

Introduction to distributed memory parallel computing

W

Intermediate-level Programming for HPC Using Python, 29th May 2024

W

Giannis Koutsou,

Computation-based Science and Technology Research Center,

The Cyprus Institute

Outline

Parallel programming

- Distributed memory paradigm
- Main features and relation to shared memory parallelism

MPI Introduction

- The MPI standard and development workflow
- Using MPI with Python
- Hands-on examples

Outline

Parallel programming

- Distributed memory paradigm
- Main features and relation to shared memory parallelism

MPI Introduction

- The MPI standard and development workflow
- Using MPI with Python
- Hands-on examples

These slides

1. PDF from github: <https://github.com/CaSToRC-CyI/EuroCC-Intermediate-Training-2024>
2. Via browser: <https://slides.koutsou.net/EuroCC-2024-05-29/>

Outline

Parallel programming

- Distributed memory paradigm
- Main features and relation to shared memory parallelism

MPI Introduction

- The MPI standard and development workflow
- Using MPI with Python
- Hands-on examples

These slides

1. PDF from github: <https://github.com/CaSToRC-CyI/EuroCC-Intermediate-Training-2024>
2. Via browser: <https://slides.koutsou.net/EuroCC-2024-05-29/>

Exercises

- Available on github:
 - <https://github.com/CaSToRC-CyI/EuroCC-Intermediate-Training-2024>
- On common storage on "Cyclone"
 - /onyx/data/edu21/

The Message Passing Interface

- MPI: An Application Programmer Interface (API)
 - A *library specification*; determines functions, their names and arguments, and their functionality
- A *de facto* standard for programming *distributed memory* systems
- Current specification is version 4 (MPI-4.0), released June 9, 2021
 - For most systems you can reliably assume MPI-3.1 is in place
- Several free (open) or vendor-provided implementations, e.g.:
 - Mvapich
 - OpenMPI
 - IntelMPI

The Message Passing Interface

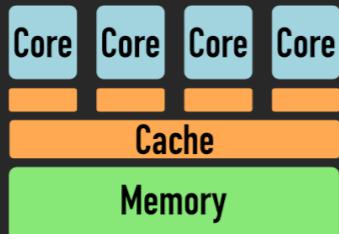
- MPI: An Application Programmer Interface (API)
 - A *library specification*; determines functions, their names and arguments, and their functionality
- A *de facto* standard for programming *distributed memory* systems
- Current specification is version 4 (MPI-4.0), released June 9, 2021
 - For most systems you can reliably assume MPI-3.1 is in place
- Several free (open) or vendor-provided implementations, e.g.:
 - Mvapich
 - OpenMPI
 - IntelMPI

Distributed memory programming

- Each process has its own memory domain
- MPI functions facilitate:
 - Obtaining environment information about the running process, e.g., process id, number of processes, etc.
 - Achieving *communication* between processes, e.g. synchronization, copying of data, etc.

Shared vs Distributed memory paradigm

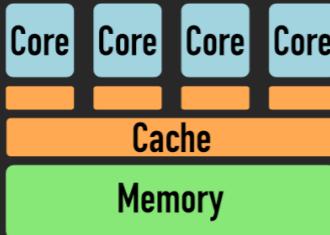
Shared memory



- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

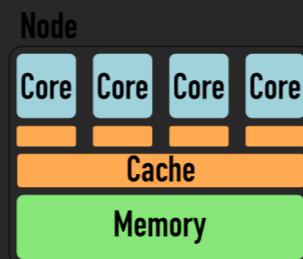
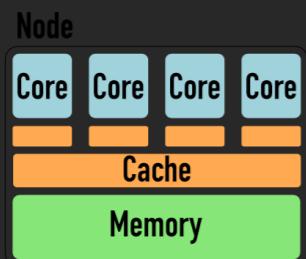
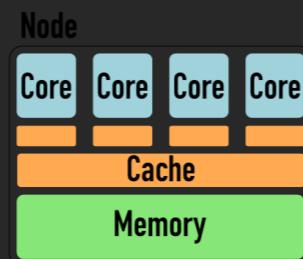
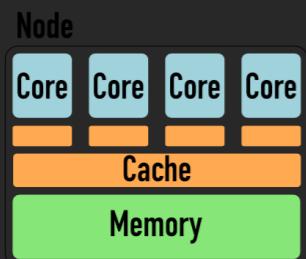
Shared vs Distributed memory paradigm

Shared memory



- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

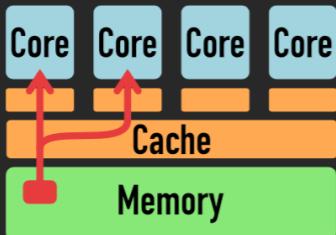
Distributed memory



- Processes have distinct memory domains (different *memory address space*)
- E.g. multiple nodes within a cluster, multiple GPUs within a node
- Programming models: **MPI**

Shared vs Distributed memory paradigm

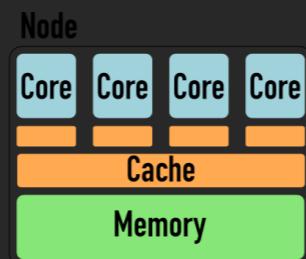
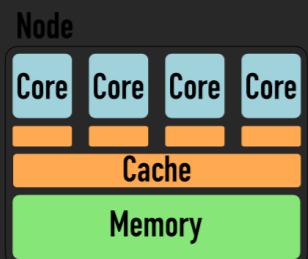
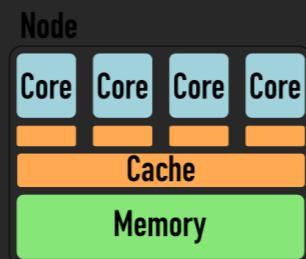
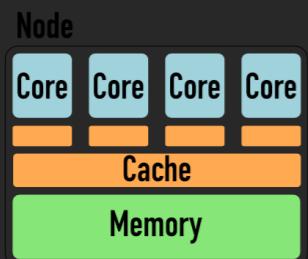
Shared memory



Data shared via memory

- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

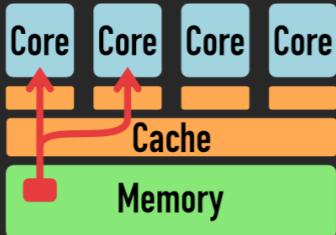
Distributed memory



- Processes have distinct memory domains (different *memory address space*)
- E.g. multiple nodes within a cluster, multiple GPUs within a node
- Programming models: **MPI**

Shared vs Distributed memory paradigm

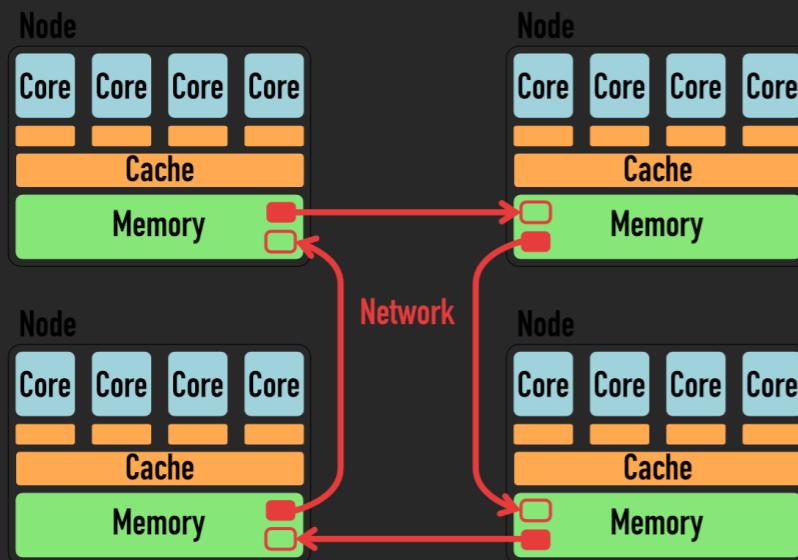
Shared memory



Data shared via memory

- Multiple processes share common memory (common *memory address space*)
- E.g. multi-core CPU, multi-socket node, GPU threads, etc.
- Programming models: OpenMP, pthreads, MPI, CUDA (sort of)

Distributed memory



Data shared via explicit communication over a network

- Processes have distinct memory domains (different *memory address space*)
- E.g. multiple nodes within a cluster, multiple GPUs within a node
- Programming models: MPI

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 python my_script.py &
ssh node02 python my_script.py &
ssh node03 python my_script.py &
```

`my_script.py` will run on each node identically

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 python my_script.py &
ssh node02 python my_script.py &
ssh node03 python my_script.py &
```

`my_script.py` will run on each node identically

- An MPI program is run in a similar way, but via a wrapper script that also initializes the parallel environment (environment variables, etc.)

```
mpirun -H node01,node02,node03 python my_script.py
```

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 python my_script.py &
ssh node02 python my_script.py &
ssh node03 python my_script.py &
```

`my_script.py` will run on each node identically

- An MPI program is run in a similar way, but via a wrapper script that also initializes the parallel environment (environment variables, etc.)

```
mpirun -H node01,node02,node03 python my_script.py
```

- In practice, a scheduler is used which determines which nodes you are currently allocated, meaning you usually will not need to explicitly specify the hostnames

```
mpirun python my_script.py
```

Running a program in parallel

- Trivially, in Linux it is simple to run a program in parallel

```
ssh node01 python my_script.py &
ssh node02 python my_script.py &
ssh node03 python my_script.py &
```

`my_script.py` will run on each node identically

- An MPI program is run in a similar way, but via a wrapper script that also initializes the parallel environment (environment variables, etc.)

```
mpirun -H node01,node02,node03 python my_script.py
```

- In practice, a scheduler is used which determines which nodes you are currently allocated, meaning you usually will not need to explicitly specify the hostnames

```
mpirun python my_script.py
```

- Depending on the system, instead of `mpirun` you may be required `mpiexec` or `srun` which take similar (but not identical) arguments

Linking against MPI/loading MPI modules

- An MPI program includes calls to MPI functions

Linking against MPI/loading MPI modules

- An MPI program includes calls to MPI functions
 - For a compiled program like C, we would include a single header file with all function definitions, macros, and constants

```
#include <mpi.h>
```

Linking against MPI/loading MPI modules

- An MPI program includes calls to MPI functions
 - For a compiled program like C, we would include a single header file with all function definitions, macros, and constants

```
#include <mpi.h>
```

- We would also need to link against MPI libraries; precise invocation depends on the compiler, the MPI implementation used, its version, etc., e.g.:

```
gcc -o my_mpi_program my_mpi_program.c -I/opt/mpi/include -L/opt/mpi/lib -lmpi
```

Linking against MPI/loading MPI modules

- An MPI program includes calls to MPI functions
 - For a compiled program like C, we would include a single header file with all function definitions, macros, and constants

```
#include <mpi.h>
```

- We would also need to link against MPI libraries; precise invocation depends on the compiler, the MPI implementation used, its version, etc., e.g.:

```
gcc -o my_mpi_program my_mpi_program.c -I/opt/mpi/include -L/opt/mpi/lib -lmpi
```

- Thankfully, knowing the locations of the MPI library and include files is never needed in practice; implementations come with wrappers that set the appropriate include paths and linker options:

```
mpicc -o my_mpi_program my_mpi_program.c
```

Linking against MPI/loading MPI modules

- An MPI program includes calls to MPI functions
 - For a compiled program like C, we would include a single header file with all function definitions, macros, and constants

```
#include <mpi.h>
```

- We would also need to link against MPI libraries; precise invocation depends on the compiler, the MPI implementation used, its version, etc., e.g.:

```
gcc -o my_mpi_program my_mpi_program.c -I/opt/mpi/include -L/opt/mpi/lib -lmpi
```

- Thankfully, knowing the locations of the MPI library and include files is never needed in practice; implementations come with wrappers that set the appropriate include paths and linker options:

```
mpicc -o my_mpi_program my_mpi_program.c
```

- For our Python examples, we will use a Python module, `mpi4py`, which implements the MPI API

```
import mpi4py
```

or, more commonly

```
from mpi4py import MPI
```

Initialization

- MPI functions begin with the `MPI_` prefix in C

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

```
MPI_Init(&argc, &argv);
```

where `argc` and `argv` are the typical names used for the command line variables passed to `main()`

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

```
MPI_Init(&argc, &argv);
```

where `argc` and `argv` are the typical names used for the command line variables passed to `main()`

- When using `mpi4py`:
 - MPI functions are methods of the `MPI` module
 - Loading the module is equivalent to initializing MPI: `from mpi4py import MPI`

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

```
MPI_Init(&argc, &argv);
```

where `argc` and `argv` are the typical names used for the command line variables passed to `main()`

- When using `mpi4py`:
 - MPI functions are methods of the `MPI` module
 - Loading the module is equivalent to initializing MPI: `from mpi4py import MPI`
- In C, before the end of the program, call `MPI_Finalize()`, otherwise the MPI runtime may assume your program finished in error

Initialization

- MPI functions begin with the `MPI_` prefix in C
- Call `MPI_Init()` first, before any other MPI call:

```
MPI_Init(&argc, &argv);
```

where `argc` and `argv` are the typical names used for the command line variables passed to `main()`

- When using `mpi4py`:
 - MPI functions are methods of the `MPI` module
 - Loading the module is equivalent to initializing MPI: `from mpi4py import MPI`
- In C, before the end of the program, call `MPI_Finalize()`, otherwise the MPI runtime may assume your program finished in error

```
#include <mpi.h>

int
main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    /*
     ...
     ...
     */
    MPI_Finalize();
    return 0;
}
```

Initialization

- Two functions you will almost always call
 - `MPI_Comm_size()` or `MPI.COMM_WORLD.Get_size()`: gives the number of parallel process running (n_{proc})
 - `MPI_Comm_rank()` or `MPI.COMM_WORLD.Get_rank()`: determines the *rank* of the process, i.e. a unique number between 0 and $n_{\text{proc}} - 1$ that identifies the calling process
- A complete example:

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    int nproc, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf(" This is rank = %d of nproc = %d\n", rank, nproc);
    MPI_Finalize();
    return 0;
}
```

```
from mpi4py import MPI
nproc = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
print(f" This is rank = {rank} of nproc = {nproc}")
```

Initialization

- Two functions you will almost always call
 - `MPI_Comm_size()` or `MPI.COMM_WORLD.Get_size()`: gives the number of parallel process running (n_{proc})
 - `MPI_Comm_rank()` or `MPI.COMM_WORLD.Get_rank()`: determines the *rank* of the process, i.e. a unique number between 0 and $n_{\text{proc}} - 1$ that identifies the calling process
- A complete example:

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    int nproc, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf(" This is rank = %d of nproc = %d\n", rank, nproc);
    MPI_Finalize();
    return 0;
}
```

```
from mpi4py import MPI
nproc = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
print(f" This is rank = {rank} of nproc = {nproc}")
```

- `MPI_COMM_WORLD` or `MPI.COMM_WORLD` is an *MPI communicator*
 - A user can partition processes into subgroups, defining *custom communicators* (not covered here)
 - *By default*, the initial communicator is `COMM_WORLD`, i.e. all processes in one communicator

Initialization

- Two functions you will almost always call
 - `MPI_Comm_size()` or `MPI.COMM_WORLD.Get_size()`: gives the number of parallel process running (n_{proc})
 - `MPI_Comm_rank()` or `MPI.COMM_WORLD.Get_rank()`: determines the *rank* of the process, i.e. a unique number between 0 and $n_{\text{proc}} - 1$ that identifies the calling process
- A complete example:

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    int nproc, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf(" This is rank = %d of nproc = %d\n", rank, nproc);
    MPI_Finalize();
    return 0;
}
```

```
from mpi4py import MPI

nproc = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
print(f" This is rank = {rank} of nproc = {nproc}")
```

- `MPI_COMM_WORLD` or `MPI.COMM_WORLD` is an *MPI communicator*
 - A user can partition processes into subgroups, defining *custom communicators* (not covered here)
 - *By default*, the initial communicator is `COMM_WORLD`, i.e. all processes in one communicator

- **No assumptions** can safely be made about the order in which the `printf()` statements occur, i.e. the order in which each process prints is **practically random**

Initialization

- Compiling and running the previous program (assuming it is saved as `example.c` or `example.py`)

```
[user@front02 ~]$ mpicc -o example example.c  
[user@front02 ~]$ mpirun -n 5 example  
This is rank = 3 of nproc = 5  
This is rank = 1 of nproc = 5  
This is rank = 2 of nproc = 5  
This is rank = 4 of nproc = 5  
This is rank = 0 of nproc = 5
```

```
[user@front02 ~]$ mpirun -n 5 python example.py  
This is rank = 3 of nproc = 5  
This is rank = 1 of nproc = 5  
This is rank = 2 of nproc = 5  
This is rank = 4 of nproc = 5  
This is rank = 0 of nproc = 5
```

Initialization

- Compiling and running the previous program (assuming it is saved as `example.c` or `example.py`)

```
[user@front02 ~]$ mpicc -o example example.c
[user@front02 ~]$ mpirun -n 5 example
This is rank = 3 of nproc = 5
This is rank = 1 of nproc = 5
This is rank = 2 of nproc = 5
This is rank = 4 of nproc = 5
This is rank = 0 of nproc = 5
```

```
[user@front02 ~]$ mpirun -n 5 python example.py
This is rank = 3 of nproc = 5
This is rank = 1 of nproc = 5
This is rank = 2 of nproc = 5
This is rank = 4 of nproc = 5
This is rank = 0 of nproc = 5
```

- Note that the order is random

Initialization

- Compiling and running the previous program (assuming it is saved as `example.c` or `example.py`)

```
[user@front02 ~]$ mpicc -o example example.c  
[user@front02 ~]$ mpirun -n 5 example  
This is rank = 3 of nproc = 5  
This is rank = 1 of nproc = 5  
This is rank = 2 of nproc = 5  
This is rank = 4 of nproc = 5  
This is rank = 0 of nproc = 5
```

```
[user@front02 ~]$ mpirun -n 5 python example.py  
This is rank = 3 of nproc = 5  
This is rank = 1 of nproc = 5  
This is rank = 2 of nproc = 5  
This is rank = 4 of nproc = 5  
This is rank = 0 of nproc = 5
```

- Note that the order is random
- Unless any *explicit synchronization* is implemented, the order in which each process calls the print statement is unpredictable

Initialization

- Compiling and running the previous program (assuming it is saved as `example.c` or `example.py`)

```
[user@front02 ~]$ mpicc -o example example.c  
[user@front02 ~]$ mpirun -n 5 example  
This is rank = 3 of nproc = 5  
This is rank = 1 of nproc = 5  
This is rank = 2 of nproc = 5  
This is rank = 4 of nproc = 5  
This is rank = 0 of nproc = 5
```

```
[user@front02 ~]$ mpirun -n 5 python example.py  
This is rank = 3 of nproc = 5  
This is rank = 1 of nproc = 5  
This is rank = 2 of nproc = 5  
This is rank = 4 of nproc = 5  
This is rank = 0 of nproc = 5
```

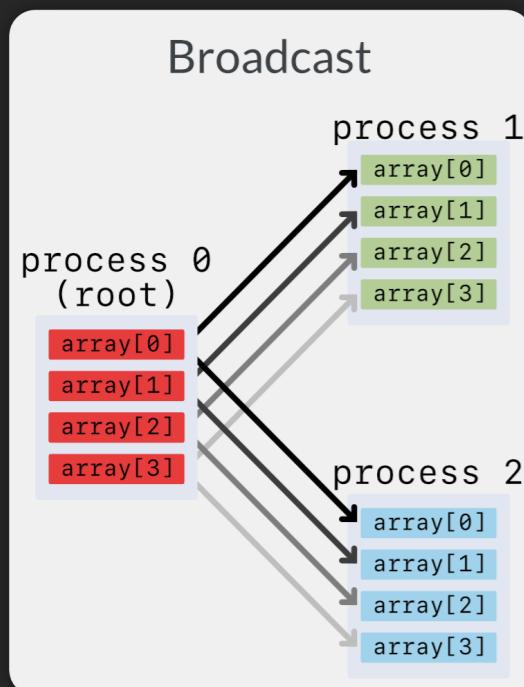
- Note that the order is random
- Unless any *explicit synchronization* is implemented, the order in which each process calls the print statement is unpredictable
- Most *collective operations* implicitly synchronize the processes

Collective operations

- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
 - Broadcast a variable from one process to all processes (Broadcast)
 - Distribute elements of an array on one process to multiple processes (Scatter)
 - Collect elements of arrays scattered over processes into a single process (Gather)
 - Sum a variable over all processes (Reduction)

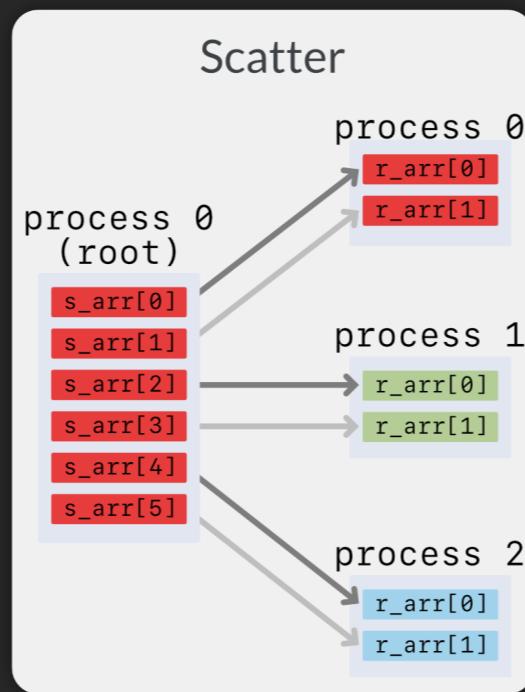
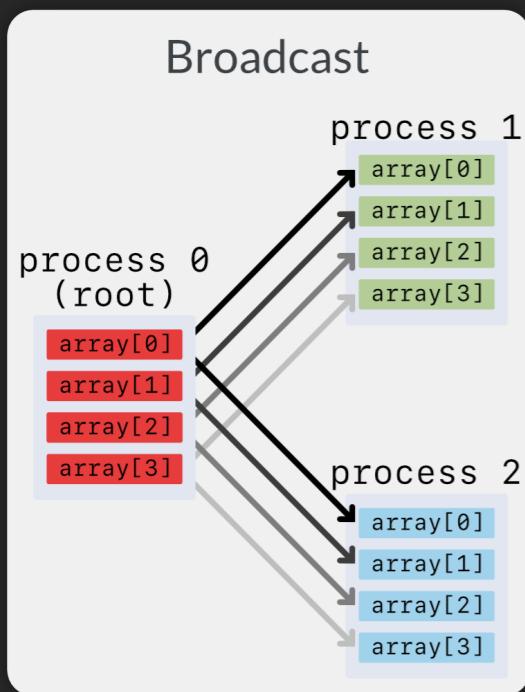
Collective operations

- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
 - Broadcast a variable from one process to all processes (Broadcast)
 - Distribute elements of an array on one process to multiple processes (Scatter)
 - Collect elements of arrays scattered over processes into a single process (Gather)
 - Sum a variable over all processes (Reduction)



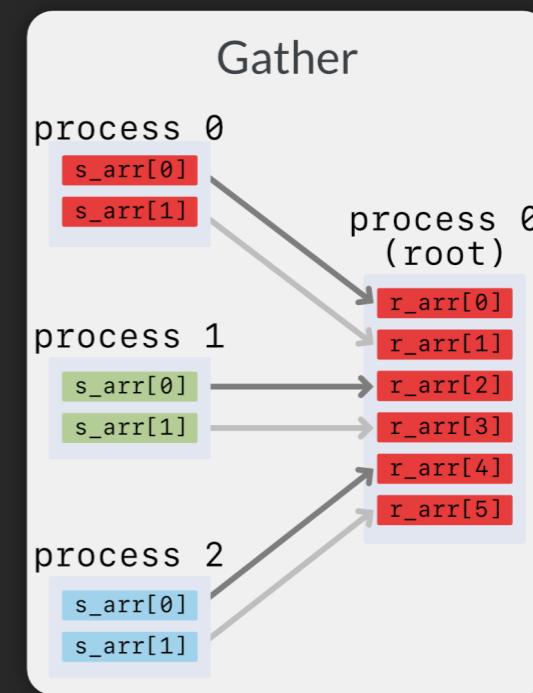
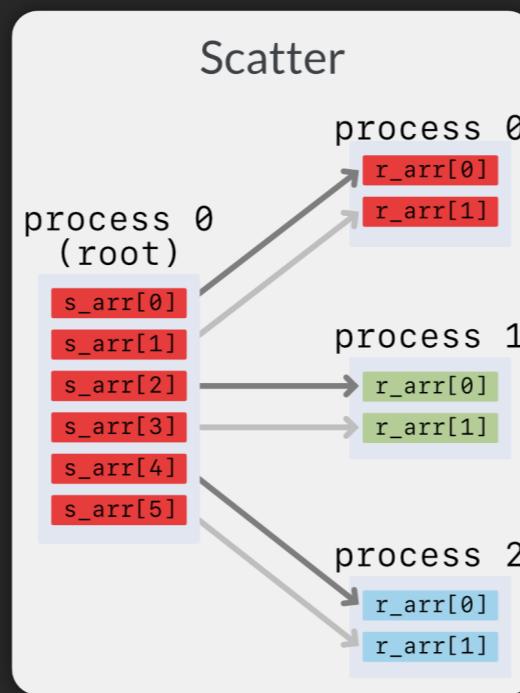
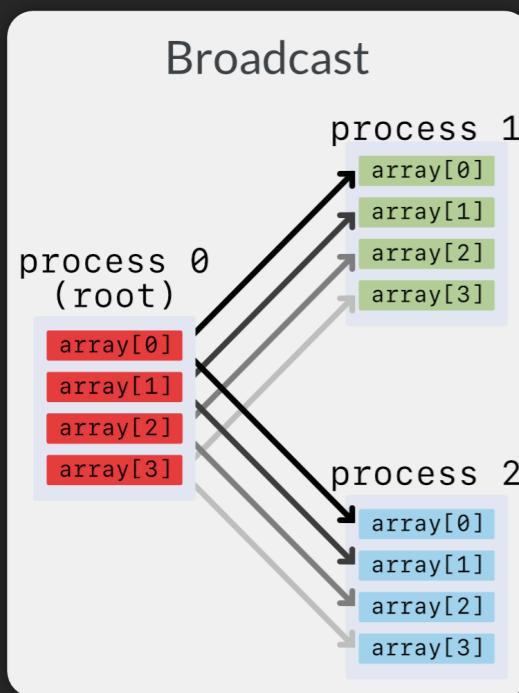
Collective operations

- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
 - Broadcast a variable from one process to all processes (Broadcast)
 - Distribute elements of an array on one process to multiple processes (Scatter)
 - Collect elements of arrays scattered over processes into a single process (Gather)
 - Sum a variable over all processes (Reduction)



Collective operations

- The first set of communication functions we will look at are *collective operations*
- Collective: all processes must be involved in the operation (as opposed to *point-to-point* communications)
- Examples (this list is not exhaustive!):
 - Broadcast a variable from one process to all processes (Broadcast)
 - Distribute elements of an array on one process to multiple processes (Scatter)
 - Collect elements of arrays scattered over processes into a single process (Gather)
 - Sum a variable over all processes (Reduction)



Collective operations: Broadcast

Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

```
MPI.COMM_WORLD.Bcast(buf: BufSpec, root: int = 0)
```

Collective operations: Broadcast

Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

```
MPI.COMM_WORLD.Bcast(buf: BufSpec, root: int = 0)
```

- *Example:* Broadcast from rank 0 (root), the four-element, double precision array arr[]

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
MPI.COMM_WORLD.Bcast(arr, 0)
```

Collective operations: Broadcast

Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

```
MPI.COMM_WORLD.Bcast(buf: BufSpec, root: int = 0)
```

- *Example:* Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
MPI.COMM_WORLD.Bcast(arr, 0)
```

- *Example:* Broadcast from rank 0 (root), the scalar integer variable `var`

```
MPI_Bcast(&var, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
var = np.array(var)
MPI.COMM_WORLD.Bcast(var, 0)
```

Collective operations: Broadcast

Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

```
MPI.COMM_WORLD.Bcast(buf: BufSpec, root: int = 0)
```

- *Example:* Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
MPI.COMM_WORLD.Bcast(arr, 0)
```

- *Example:* Broadcast from rank 0 (root), the scalar integer variable `var`

```
MPI_Bcast(&var, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
var = np.array(var)
MPI.COMM_WORLD.Bcast(var, 0)
```

- In C, the `MPI_Datatype` is important since MPI uses it to estimate the size in bytes that need to be transferred. In Python this can be deduced from the variable

Collective operations: Broadcast

Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

```
MPI.COMM_WORLD.Bcast(buf: BufSpec, root: int = 0)
```

- *Example:* Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
MPI.COMM_WORLD.Bcast(arr, 0)
```

- *Example:* Broadcast from rank 0 (root), the scalar integer variable `var`

```
MPI_Bcast(&var, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
var = np.array(var)
MPI.COMM_WORLD.Bcast(var, 0)
```

- In C, the `MPI_Datatype` is important since MPI uses it to estimate the size in bytes that need to be transferred. In Python this can be deduced from the variable
- In C, we can pass a scalar by using its memory address (second example above). In Python, we can use zero-dimensional numpy array for this

Collective operations: Broadcast

Broadcast:

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

```
MPI.COMM_WORLD.Bcast(buf: BufSpec, root: int = 0)
```

- *Example:* Broadcast from rank 0 (root), the four-element, double precision array `arr[]`

```
MPI_Bcast(arr, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
MPI.COMM_WORLD.Bcast(arr, 0)
```

- *Example:* Broadcast from rank 0 (root), the scalar integer variable `var`

```
MPI_Bcast(&var, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
var = np.array(var)
MPI.COMM_WORLD.Bcast(var, 0)
```

- In C, the `MPI_Datatype` is important since MPI uses it to estimate the size in bytes that need to be transferred. In Python this can be deduced from the variable
- In C, we can pass a scalar by using its memory address (second example above). In Python, we can use zero-dimensional numpy array for this
- Full list of types available in MPI documentation. E.g. see: <https://www.mpich.org/static/docs/latest/www3/Constants.html>

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Scatter(  
    sendbuf: BufSpec, recvbuf: BufSpec, root: int = 0  
)
```

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Scatter(  
    sendbuf: BufSpec, recvbuf: BufSpec, root: int = 0  
)
```

- `sendcount` is the number of elements to be sent to *each* process

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Scatter(  
    sendbuf: BufSpec, recvbuf: BufSpec, root: int = 0  
)
```

- `sendcount` is the number of elements to be sent to *each* process
- `sendbuf` is only relevant in the root process

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Scatter(  
    sendbuf: BufSpec, recvbuf: BufSpec, root: int = 0  
)
```

- `sendcount` is the number of elements to be sent to *each* process
 - `sendbuf` is only relevant in the root process
- *Example:* distribute a 12-element array from process 0, assuming 3 processes in total (including root)

```
double arr_all[12]; /* ← this only needs to be defined on process with rank = 0 */  
double arr_proc[4];  
MPI_Scatter(arr_all, 4, MPI_DOUBLE, arr_proc, 4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
arr_all = np.random.rand(12)  
arr_proc = np.zeros([4])  
MPI.COMM_WORLD.Scatter(arr_all, arr_proc, root = 0)
```

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Scatter(  
    sendbuf: BufSpec, recvbuf: BufSpec, root: int = 0  
)
```

- *Example:* distribute each element of a 4-element array to 4 processes in total (including root)

```
double arr[4]; /* ← this only needs to be defined on process with rank = 0 */  
double element;  
MPI_Scatter(arr, 1, MPI_DOUBLE, &element, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
arr = np.random.rand(4)  
element = np.zeros([])  
MPI.COMM_WORLD.Scatter(arr, element, root = 0)
```

Collective operations: Scatter

- Scatter:

```
MPI_Scatter(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Scatter(  
    sendbuf: BufSpec, recvbuf: BufSpec, root: int = 0  
)
```

- *Example:* distribute each element of a 4-element array to 4 processes in total (including root)

```
double arr[4]; /* ← this only needs to be defined on process with rank = 0 */  
double element;  
MPI_Scatter(arr, 1, MPI_DOUBLE, &element, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
arr = np.random.rand(4)  
element = np.zeros([])  
MPI.COMM_WORLD.Scatter(arr, element, root = 0)
```

Note: the initialization of `element` as a zero-dimensional array

Collective operations: Gather

- Gather:

```
MPI_Gather(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,  
    MPI_Comm comm  
)
```

Collective operations: Gather

- Gather:

```
MPI_Gather(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,  
    MPI_Comm comm  
)
```

- `recvcount` is the number of elements to be received by *each* process

Collective operations: Gather

- Gather:

```
MPI_Gather(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,  
    MPI_Comm comm  
)
```

- `recvcount` is the number of elements to be received by *each* process
- `recvbuf` is only relevant in the root process

Collective operations: Gather

- Gather:

```
MPI_Gather(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,  
    MPI_Comm comm  
)
```

- *Example:* collect a 9-element array at process 0, by concatenating 3 elements from each of 3 processes in total (including root)

```
double arr_all[9]; /* ← this only needs to be defined on process with rank = 0 */  
double arr_proc[3];  
MPI_Gather(arr_proc, 3, MPI_DOUBLE, arr_all, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
arr_all = np.zeros([9])  
arr_proc = np.random.rand(3)  
MPI.COMM_WORLD.Gather(arr_proc, arr_all, root = 0)
```

Collective operations: Gather

- Gather:

```
MPI_Gather(  
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,  
    MPI_Comm comm  
)
```

- *Example:* collect a 4-element array at process 0, by concatenating an element from each of 4 processes in total (including root)

```
double arr[4]; /* ← this only needs to be defined on process with rank = 0 */  
double element;  
MPI_Gather(&element, 1, MPI_DOUBLE, arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

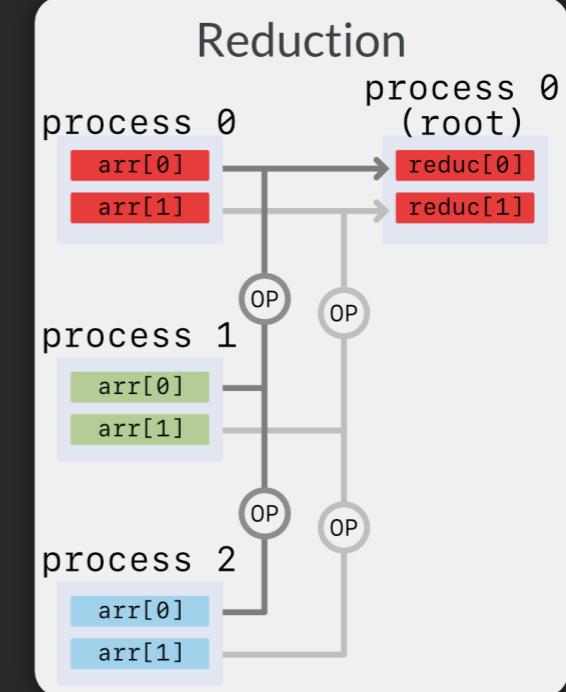
```
arr = np.zeros([4])  
element = np.random.rand(1)  
MPI.COMM_WORLD.Gather(element, arr, root = 0)
```

Collective operations: Reduction

- Reduction:

```
MPI_Reduce(  
    const void *sendbuf, void *recvbuf, int count,  
    MPI_Datatype datatype, MPI_Op op, int root,  
    MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Reduce(  
    sendbuf: BufSpec, recvbuf: BufSpec,  
    op: Op = SUM, root: int = 0  
)
```



Collective operations: Reduction

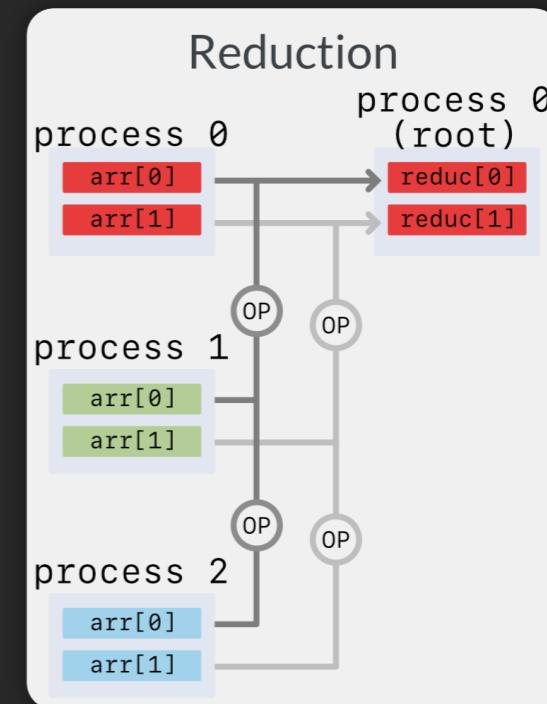
- Reduction:

```
MPI_Reduce(  
    const void *sendbuf, void *recvbuf, int count,  
    MPI_Datatype datatype, MPI_Op op, int root,  
    MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Reduce(  
    sendbuf: BufSpec, recvbuf: BufSpec,  
    op: Op = SUM, root: int = 0  
)
```

- Notes:

- `Op` is an operation, e.g. `MPI.SUM`, `MPI.PROD`, etc. (`MPI_SUM` and `MPI_PROD` in C)
- In C note the need for specifying the datatype and `count`, the number of elements of the arrays
- The operation is over all processes in `comm`, in this case `COMM_WORLD`



Collective operations: Reduction

- Reduction:

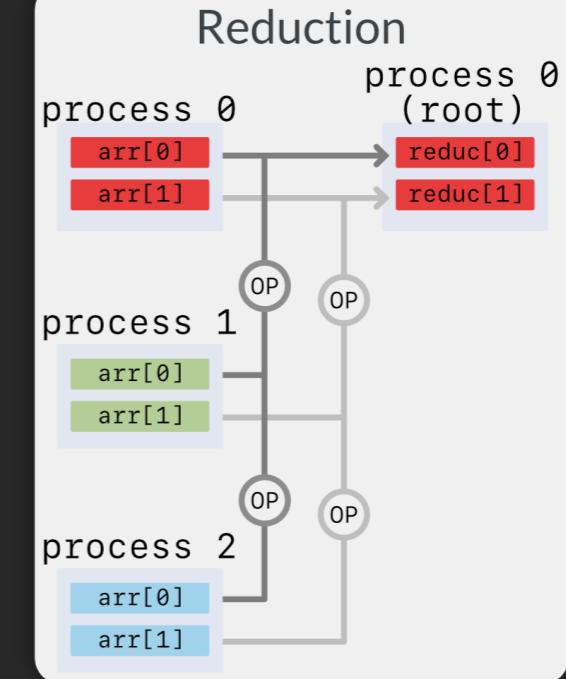
```
MPI_Reduce(  
    const void *sendbuf, void *recvbuf, int count,  
    MPI_Datatype datatype, MPI_Op op, int root,  
    MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Reduce(  
    sendbuf: BufSpec, recvbuf: BufSpec,  
    op: Op = SUM, root: int = 0  
)
```

- Example: Sum each element of a 3-element array over all processes

```
double s_arr[3];  
double r_arr[3]; /* ← only needs to      *  
                  *      be defined on root */  
MPI_Reduce(s_arr, r_arr, 3, MPI_DOUBLE,  
           MPI_SUM, 0, MPI_COMM_WORLD);
```

```
s_arr = np.random.rand(3)  
r_arr = np.zeros([3])  
MPI.COMM_WORLD.Reduce(s_arr, r_arr, MPI.SUM, root = 0)
```



Collective operations: Reduction

- Reduction:

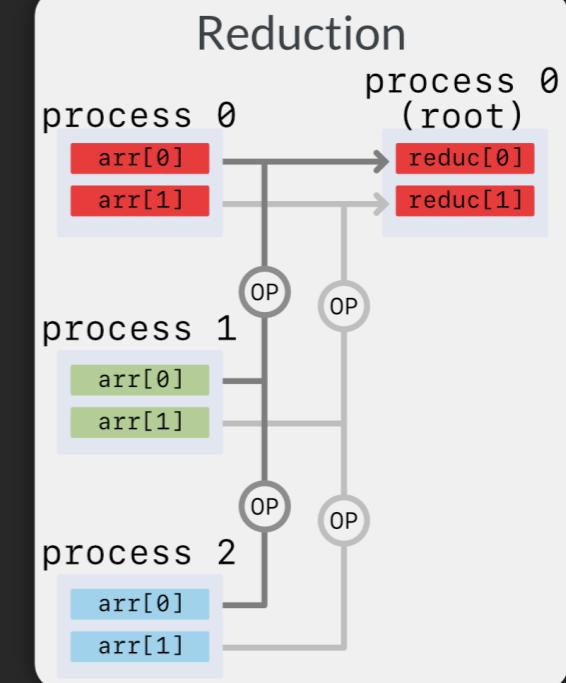
```
MPI_Reduce(  
    const void *sendbuf, void *recvbuf, int count,  
    MPI_Datatype datatype, MPI_Op op, int root,  
    MPI_Comm comm  
)
```

```
MPI.COMM_WORLD.Reduce(  
    sendbuf: BufSpec, recvbuf: BufSpec,  
    op: Op = SUM, root: int = 0  
)
```

- Example: Sum variable `var` over all processes

```
double var;  
double sum; /* ← only needs to      *  
             *      be defined on root */  
MPI_Reduce(&var, &sum, 1, MPI_DOUBLE,  
           MPI_SUM, 0, MPI_COMM_WORLD);
```

```
var = np.random.rand(1)  
sum = np.zeros([])  
MPI.COMM_WORLD.Reduce(var, sum, MPI.SUM, root = 0)
```



Collective operations

Some additional notes on variants of the collectives we have covered

- `Scatterv()` and `Gatherv()`
 - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
 - Need specifying additional arguments containing offsets of the send or receive buffer

Collective operations

Some additional notes on variants of the collectives we have covered

- `Scatterv()` and `Gatherv()`
 - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
 - Need specifying additional arguments containing offsets of the send or receive buffer
- `Allreduce()`
 - Same as `Reduce()`, but result is placed on all processes in the pool
 - Result is equivalent to `Reduce()` followed by an `Bcast()`

Collective operations

Some additional notes on variants of the collectives we have covered

- `Scatterv()` and `Gatherv()`
 - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
 - Need specifying additional arguments containing offsets of the send or receive buffer
- `Allreduce()`
 - Same as `Reduce()`, but result is placed on all processes in the pool
 - Result is equivalent to `Reduce()` followed by an `Bcast()`
- In-place operations
 - For some functions, can replace the send or receive buffer with `MPI.IN_PLACE` (`MPI_IN_PLACE` in C)
→ which buffer depends on the specific MPI function

Collective operations

Some additional notes on variants of the collectives we have covered

- `Scatterv()` and `Gatherv()`
 - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
 - Need specifying additional arguments containing offsets of the send or receive buffer
- `Allreduce()`
 - Same as `Reduce()`, but result is placed on all processes in the pool
 - Result is equivalent to `Reduce()` followed by an `Bcast()`
- In-place operations
 - For some functions, can replace the send or receive buffer with `MPI.IN_PLACE` (`MPI_IN_PLACE` in C)
→ which buffer depends on the specific MPI function
 - Instructs MPI to use the **same** buffer for receive and send

Collective operations

Some additional notes on variants of the collectives we have covered

- `Scatterv()` and `Gatherv()`
 - Allow specifying *varying* number of elements to be distributed or collected to or from the process pool
 - Need specifying additional arguments containing offsets of the send or receive buffer
- `Allreduce()`
 - Same as `Reduce()`, but result is placed on all processes in the pool
 - Result is equivalent to `Reduce()` followed by an `Bcast()`
- In-place operations
 - For some functions, can replace the send or receive buffer with `MPI.IN_PLACE` (`MPI_IN_PLACE` in C)
→ which buffer depends on the specific MPI function
 - Instructs MPI to use the **same** buffer for receive and send
 - E.g. below, the sum will be placed in `var` of the root process (process with `rank = 0`):

```
var = np.random.rand(1)
if rank != 0:
    MPI.COMM_WORLD.Reduce(var, None, MPI.SUM, root = 0)
else:
    MPI.COMM_WORLD.Reduce(MPI.IN_PLACE, var, MPI.SUM, root = 0)
```

Point-to-point communication

- Communications that involve transfer of data between two processes

Point-to-point communication

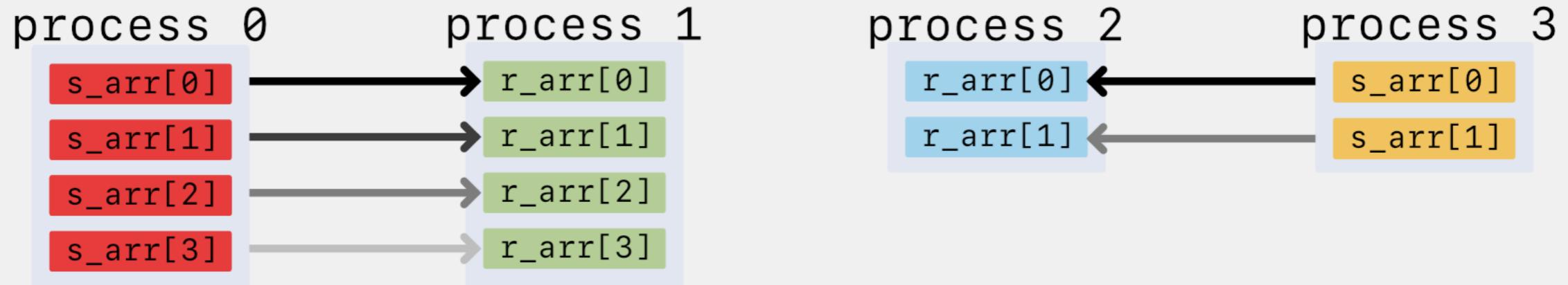
- Communications that involve transfer of data between two processes
- Most common case: send/receive
 - The sender process issues a send operation
 - The receiver process posts a receive operation

Point-to-point communication

- Communications that involve transfer of data between two processes
- Most common case: send/receive
 - The sender process issues a send operation
 - The receiver process posts a receive operation
- Asynchronous in nature: caution needed for preventing *deadlocks*, e.g.
 - Sending to a process which has not posted a matching receive
 - Posting a receive which does not have a matching send

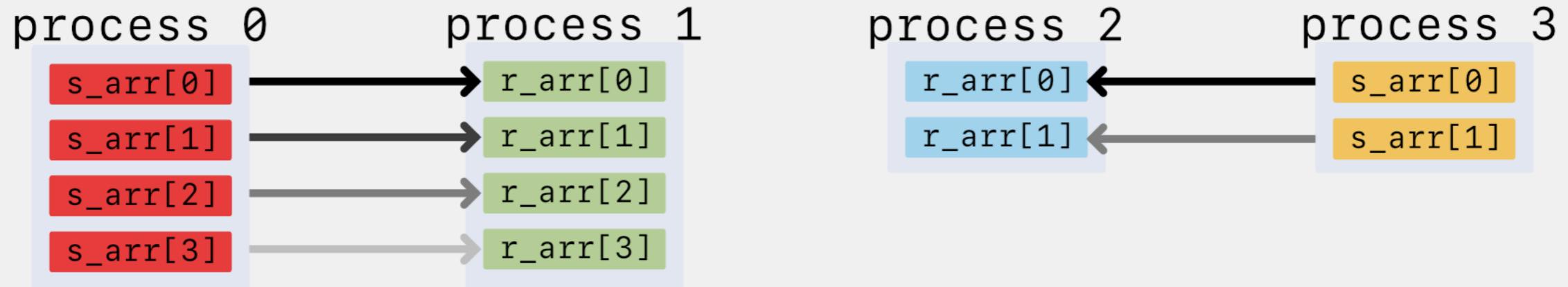
Point-to-point communication

- Communications that involve transfer of data between two processes
- Most common case: send/receive
 - The sender process issues a send operation
 - The receiver process posts a receive operation
- Asynchronous in nature: caution needed for preventing *deadlocks*, e.g.
 - Sending to a process which has not posted a matching receive
 - Posting a receive which does not have a matching send



Point-to-point communication

- Communications that involve transfer of data between two processes
- Most common case: send/receive
 - The sender process issues a send operation
 - The receiver process posts a receive operation
- Asynchronous in nature: caution needed for preventing *deadlocks*, e.g.
 - Sending to a process which has not posted a matching receive
 - Posting a receive which does not have a matching send



Two point-to-point communications are depicted above
↳ between i) process 0 and 1 and between ii) process 2 and 3

Point-to-point communication

- Send/Receive

```
MPI.COMM_WORLD.Send(buf: BufSpec, dest: int, tag: int = 0)
MPI.COMM_WORLD.Recv(buf: BufSpec, source: int = ANY_SOURCE, tag: int = ANY_TAG, status: Status = None)
```

Point-to-point communication

- Send/Receive

```
MPI.COMM_WORLD.Send(buf: BufSpec, dest: int, tag: int = 0)
MPI.COMM_WORLD.Recv(buf: BufSpec, source: int = ANY_SOURCE, tag: int = ANY_TAG, status: Status = None)
```

- Note the need to specify a source and destination rank

Point-to-point communication

- Send/Receive

```
MPI.COMM_WORLD.Send(buf: BufSpec, dest: int, tag: int = 0)
MPI.COMM_WORLD.Recv(buf: BufSpec, source: int = ANY_SOURCE, tag: int = ANY_TAG, status: Status = None)
```

- Note the need to specify a source and destination rank
- The `tag` variable tags the message. In the receiving process, it must match what the sender specified, or can be set to `MPI.ANY_TAG`

Point-to-point communication

- Send/Receive

```
MPI.COMM_WORLD.Send(buf: BufSpec, dest: int, tag: int = 0)
MPI.COMM_WORLD.Recv(buf: BufSpec, source: int = ANY_SOURCE, tag: int = ANY_TAG, status: Status = None)
```

- Note the need to specify a source and destination rank
- The `tag` variable tags the message. In the receiving process, it must match what the sender specified, or can be set to `MPI.ANY_TAG`
- Use of `MPI.ANY_SOURCE` in `Recv()` means "accept data from any source"

Point-to-point communication

- Send/Receive

```
MPI.COMM_WORLD.Send(buf: BufSpec, dest: int, tag: int = 0)
MPI.COMM_WORLD.Recv(buf: BufSpec, source: int = ANY_SOURCE, tag: int = ANY_TAG, status: Status = None)
```

- Note the need to specify a source and destination rank
- The `tag` variable tags the message. In the receiving process, it must match what the sender specified, or can be set to `MPI.ANY_TAG`
- Use of `MPI.ANY_SOURCE` in `Recv()` means "accept data from any source"
- `status` can be used to query the result of the receive (e.g. how many elements were received). We will leave to its default value, `None`, which ignores the status

Point-to-point communication

- Send/Receive; a trivial example



Point-to-point communication

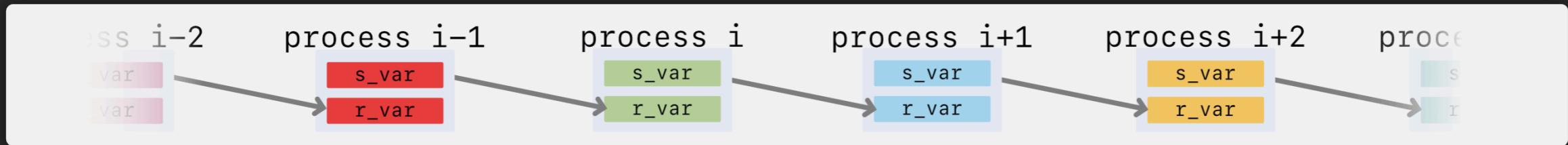
- Send/Receive; a trivial example



```
s_arr = np.random.rand(4)
r_arr = np.zeros([4])
if rank == i:
    MPI.COMM_WORLD.Send(s_arr, j)
if rank == j:
    MPI.COMM_WORLD.Recv(r_arr, i)
```

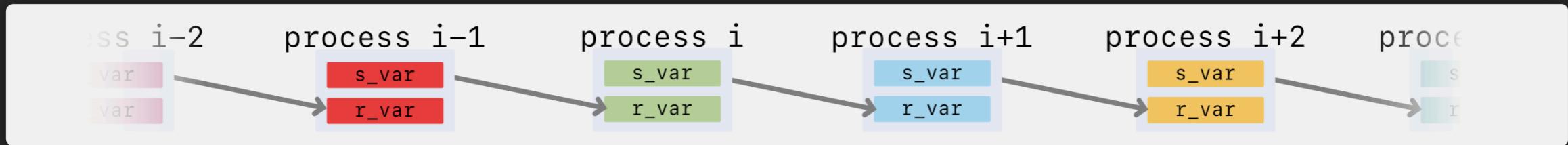
Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process

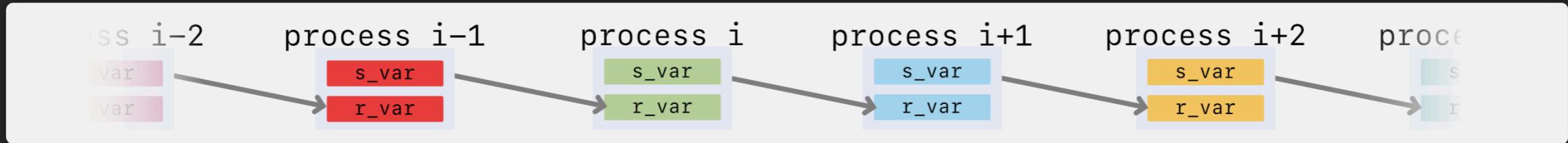


- This will **not** work:

```
MPI.COMM_WORLD.Send(s_var, rank+1);
MPI.COMM_WORLD.Recv(r_var, rank-1);
```

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



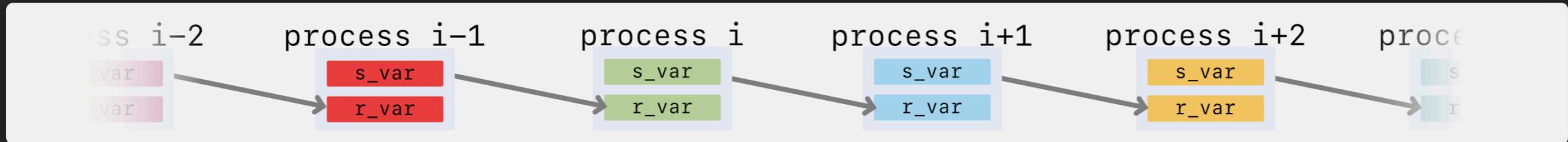
- This will **not** work:

```
MPI.COMM_WORLD.Send(s_var, rank+1);
MPI.COMM_WORLD.Recv(r_var, rank-1);
```

- It results in a **deadlock**:
 - an `Recv()` can only be posted once an `Send()` completes
 - an `Send()` can only complete if a matching `Recv()` is posted

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



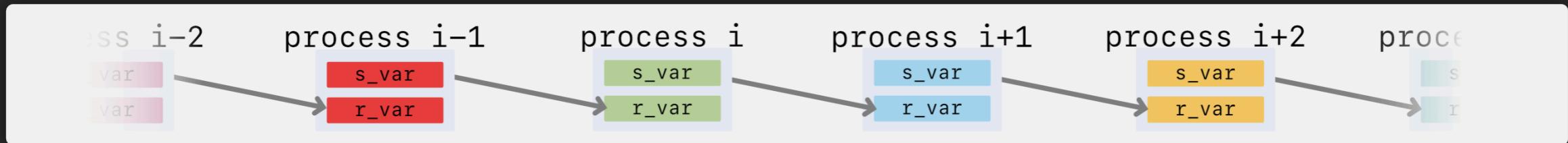
- This will **not** work:

```
MPI.COMM_WORLD.Send(s_var, rank+1);
MPI.COMM_WORLD.Recv(r_var, rank-1);
```

- It results in a **deadlock**:
 - an `Recv()` can only be posted once an `Send()` completes
 - an `Send()` can only complete if a matching `Recv()` is posted
- One can serialize the communications, i.e. use a loop to determine the order of send/receives

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



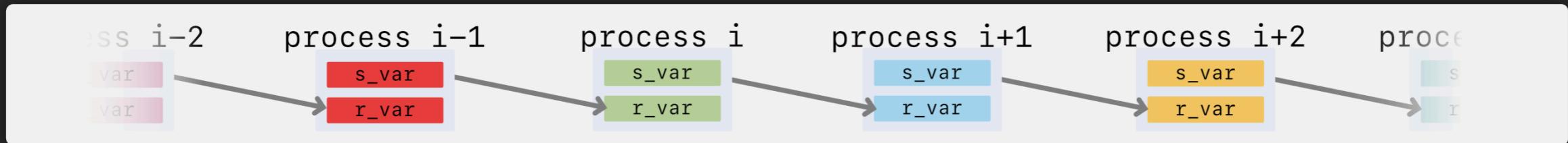
- This will **not** work:

```
MPI.COMM_WORLD.Send(s_var, rank+1);
MPI.COMM_WORLD.Recv(r_var, rank-1);
```

- It results in a **deadlock**:
 - an `Recv()` can only be posted once an `Send()` completes
 - an `Send()` can only complete if a matching `Recv()` is posted
- One can serialize the communications, i.e. use a loop to determine the order of send/receives
 - Serializes communications that would otherwise be done faster in parallel

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



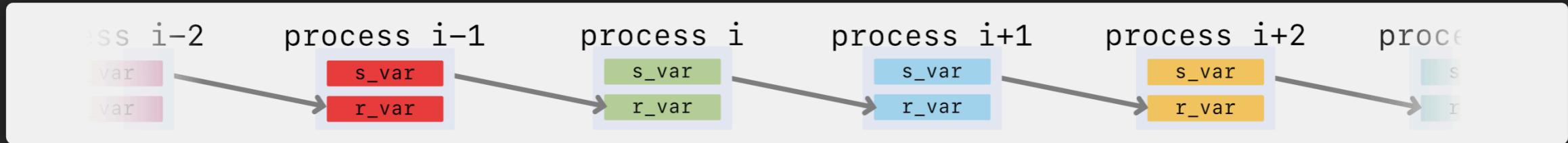
- This will **not** work:

```
MPI.COMM_WORLD.Send(s_var, rank+1);
MPI.COMM_WORLD.Recv(r_var, rank-1);
```

- It results in a **deadlock**:
 - an `Recv()` can only be posted once an `Send()` completes
 - an `Send()` can only complete if a matching `Recv()` is posted
- One can serialize the communications, i.e. use a loop to determine the order of send/receives
 - Serializes communications that would otherwise be done faster in parallel
 - Inlegant, obscure, and error-prone

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process

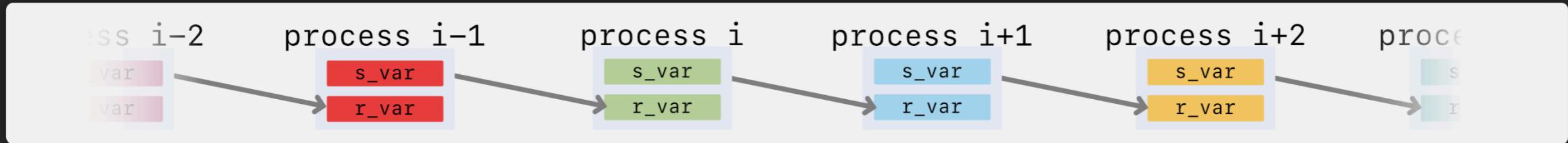


- A more efficient and elegant solution is to use `Sendrecv()`:

```
MPI.COMM_WORLD.Sendrecv(  
    sendbuf: BufSpec, dest: int, sendtag: int = 0,  
    recvbuf: BufSpec, source: int = ANY_SOURCE, recvtag: int = ANY_TAG,  
    status: Status = None  
)
```

Point-to-point communication

- It is common in parallel applications to require that every process communicates with another process, e.g. a neighboring process



- A more efficient and elegant solution is to use `Sendrecv()`:

```
MPI.COMM_WORLD.Sendrecv(  
    sendbuf: BufSpec, dest: int, sendtag: int = 0,  
    recvbuf: BufSpec, source: int = ANY_SOURCE, recvtag: int = ANY_TAG,  
    status: Status = None  
)
```

- For the depicted example:

```
MPI.COMM_WORLD.Sendrecv(s_var, rank+1, 0, r_var, rank-1)
```

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- Some MPI functions allow omitting the mandatory buffer argument. The buffer is returned instead, e.g.:

```
s_arr = np.random.rand(4)
if rank == i:
    MPI.COMM_WORLD.send(s_arr, j)
if rank == j:
    r_arr = MPI.COMM_WORLD.recv(source = i)
```

Note: lower-case versions of `send()` and `recv()`

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- Some MPI functions allow omitting the mandatory buffer argument. The buffer is returned instead, e.g.:

```
s_arr = np.random.rand(4)
if rank == i:
    MPI.COMM_WORLD.send(s_arr, j)
if rank == j:
    r_arr = MPI.COMM_WORLD.recv(source = i)
```

Note: lower-case versions of `send()` and `recv()`

- `Irecv()` and `Irecv()`
 - *Non-blocking* variants. The `I` stands for "immediate"

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- Some MPI functions allow omitting the mandatory buffer argument. The buffer is returned instead, e.g.:

```
s_arr = np.random.rand(4)
if rank == i:
    MPI.COMM_WORLD.send(s_arr, j)
if rank == j:
    r_arr = MPI.COMM_WORLD.recv(source = i)
```

Note: lower-case versions of `send()` and `recv()`

- `Irecv()` and `Irecv()`
 - *Non-blocking* variants. The `I` stands for "immediate"
 - Functions return immediately, i.e. the functions don't block waiting for `sendbuf` to be sent or `recvbuf` to be received

Point-to-point communications

Some additional notes on variants of the point-to-point communications we have covered

- Some MPI functions allow omitting the mandatory buffer argument. The buffer is returned instead, e.g.:

```
s_arr = np.random.rand(4)
if rank == i:
    MPI.COMM_WORLD.send(s_arr, j)
if rank == j:
    r_arr = MPI.COMM_WORLD.recv(source = i)
```

Note: lower-case versions of `send()` and `recv()`

- `Isend()` and `Irecv()`
 - *Non-blocking* variants. The `I` stands for "immediate"
 - Functions return immediately, i.e. the functions don't block waiting for `sendbuf` to be sent or `recvbuf` to be received
 - The function `wait()` is used to block until the operation has complete

```
request = MPI.COMM_WORLD.Isend(sendbuf, destrank);
/*
 * More code can come here, provided it
 * does not modify sendbuf, which is
 * assumed to be "in-flight"
 */
request.wait()
```

Exercises

Exercises

```
cp -r /nvme/scratch/edu21/MPI/ex?? .
```

- Exercises follow a common structure

Exercises

```
cp -r /nvme/scratch/edu21/MPI/ex?? .
```

- Exercises follow a common structure
- It is assumed that you already:
 - Know how to login to Cyclone, navigate the filesystem, and modify files
 - Can use Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.

Exercises

```
cp -r /nvme/scratch/edu21/MPI/ex?? .
```

- Exercises follow a common structure
- It is assumed that you already:
 - Know how to login to Cyclone, navigate the filesystem, and modify files
 - Can use Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
- Each folder includes (where `${n}` below is the exercise number, i.e. `01`, `02`, etc.):
 - A `.py` source code file (`ex${n}.py`)
 - A submit script (`sub-${n}.sh`)

Exercises

```
cp -r /nvme/scratch/edu21/MPI/ex?? .
```

- Exercises follow a common structure
- It is assumed that you already:
 - Know how to login to Cyclone, navigate the filesystem, and modify files
 - Can use Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
- Each folder includes (where `${n}` below is the exercise number, i.e. `01`, `02`, etc.):
 - A `.py` source code file (`ex${n}.py`)
 - A submit script (`sub-${n}.sh`)
- Our workflow will typically be:
 - Modify `ex${n}.py` as instructed
 - Submit the job script `sbatch sub-${n}.sh`
 - Look at the output, which can be found in `ex${n}-output.txt`

Exercises

```
cp -r /nvme/scratch/edu21/MPI/ex?? .
```

- Exercises follow a common structure
- It is assumed that you already:
 - Know how to login to Cyclone, navigate the filesystem, and modify files
 - Can use Slurm, the job scheduler, and its commands: `sbatch`, `squeue`, etc.
- Each folder includes (where `${n}` below is the exercise number, i.e. `01`, `02`, etc.):
 - A `.py` source code file (`ex${n}.py`)
 - A submit script (`sub-${n}.sh`)
- Our workflow will typically be:
 - Modify `ex${n}.py` as instructed
 - Submit the job script `sbatch sub-${n}.sh`
 - Look at the output, which can be found in `ex${n}-output.txt`
- Note that if you have modified `ex${n}.py` correctly, the job should complete in **less than one minute**

Exercises

- Exercises are mostly complete but require some minor modifications by you

Exercises

- Exercises are mostly complete but require some minor modifications by you
- This is mostly to "*encourage*" reading and understanding the code

Exercises

- Exercises are mostly complete but require some minor modifications by you
- This is mostly to "*encourage*" reading and understanding the code
- The MPI functions demonstrated in each exercise are:
 - ex01: Use of `Get_rank()` and `Get_size()`
 - ex02: Use of `Bcast()`, `Scatter()`, and `Reduce()`
 - ex03: Use of `Gather()`
 - ex04: Use of `Bcast()`, `Scatter()`, `Sendrecv()`, and `Gather()`

Exercises

- Exercises are mostly complete but require some minor modifications by you
- This is mostly to "*encourage*" reading and understanding the code
- The MPI functions demonstrated in each exercise are:
 - ex01: Use of `Get_rank()` and `Get_size()`
 - ex02: Use of `Bcast()`, `Scatter()`, and `Reduce()`
 - ex03: Use of `Gather()`
 - ex04: Use of `Bcast()`, `Scatter()`, `Sendrecv()`, and `Gather()`
- All exercises have been tested with specific versions of OpenMPI, the GNU Compiler, and numpy and mpi4py. Please use:

```
module load SciPy-bundle/2023.07-gfbf-2023a mpi4py/3.1.4-gompi-2023a
```

for all exercises.

Exercises

Ex01

- Modify ex01.py to call `Get_size()` and `Get_rank()` appropriately

```
#  
# TODO: call the appropriate MPI functions here  
#  
rank = # TODO  
size = # TODO
```

Exercises

Ex01

- Modify `ex01.py` to call `Get_size()` and `Get_rank()` appropriately

```
#  
# TODO: call the appropriate MPI functions here  
#  
rank = # TODO  
size = # TODO
```

- A job script has been prepared to run `ex01`:

```
[user@front02 ex01]$ cat sub-01.sh  
#!/bin/bash  
#SBATCH --job-name=01  
#SBATCH --nodes=2  
#SBATCH --ntasks=8  
#SBATCH --ntasks-per-node=4  
#SBATCH --output=ex01-output.txt  
#SBATCH --time=00:01:00  
#SBATCH --reservation=edu21  
#SBATCH -A edu21  
  
module load SciPy-bundle/2023.07-gfbf-2023a mpi4py/3.1.4-gompi-2023a  
mpirun python ex01.py
```

Exercises

Ex01

- Modify `ex01.py` to call `Get_size()` and `Get_rank()` appropriately

```
#  
# TODO: call the appropriate MPI functions here  
#  
rank = # TODO  
size = # TODO
```

- A job script has been prepared to run `ex01`:

```
[user@front02 ex01]$ cat sub-01.sh  
#!/bin/bash  
#SBATCH --job-name=01  
#SBATCH --nodes=2  
#SBATCH --ntasks=8  
#SBATCH --ntasks-per-node=4  
#SBATCH --output=ex01-output.txt  
#SBATCH --time=00:01:00  
#SBATCH --reservation=edu21  
#SBATCH -A edu21  
  
module load SciPy-bundle/2023.07-gfbf-2023a mpi4py/3.1.4-gompi-2023a  
mpirun python ex01.py
```

- 2 nodes, 8 processes, meaning 4 processes per node

Exercises

Ex01

- Modify `ex01.py` to call `Get_size()` and `Get_rank()` appropriately

```
#  
# TODO: call the appropriate MPI functions here  
#  
rank = # TODO  
size = # TODO
```

- A job script has been prepared to run `ex01`:

```
[user@front02 ex01]$ cat sub-01.sh  
#!/bin/bash  
#SBATCH --job-name=01  
#SBATCH --nodes=2  
#SBATCH --ntasks=8  
#SBATCH --ntasks-per-node=4  
#SBATCH --output=ex01-output.txt  
#SBATCH --time=00:01:00  
#SBATCH --reservation=edu21  
#SBATCH -A edu21  
  
module load SciPy-bundle/2023.07-gfbf-2023a mpi4py/3.1.4-gompi-2023a  
mpirun python ex01.py
```

- 2 nodes, 8 processes, meaning 4 processes per node
- program output will be redirected to file `ex01-output.txt`

Exercises

Ex01

- Modify `ex01.py` to call `Get_size()` and `Get_rank()` appropriately

```
#  
# TODO: call the appropriate MPI functions here  
#  
rank = # TODO  
size = # TODO
```

- A job script has been prepared to run `ex01`:

```
[user@front02 ex01]$ cat sub-01.sh  
#!/bin/bash  
#SBATCH --job-name=01  
#SBATCH --nodes=2  
#SBATCH --ntasks=8  
#SBATCH --ntasks-per-node=4  
#SBATCH --output=ex01-output.txt  
#SBATCH --time=00:01:00  
#SBATCH --reservation=edu21  
#SBATCH -A edu21  
  
module load SciPy-bundle/2023.07-gfbf-2023a mpi4py/3.1.4-gompi-2023a  
mpirun python ex01.py
```

- 2 nodes, 8 processes, meaning 4 processes per node
- program output will be redirected to file `ex01-output.txt`
- requests 1 minute. If not done by then, the scheduler will kill the job

Exercises

Ex01

- Submit the job script:

```
[user@front02 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front02 ex01]$
```

Exercises

Ex01

- Submit the job script:

```
[user@front02 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front02 ex01]$
```

- Check status of job:

```
[user@front02 ex01]$ squeue -u $USER
JOBID PARTITION      NAME      USER ST      TIME   NODES NODELIST(REASON)
69712      cpu      01      user  R      0:00      2 cn[01-02]
[user@front02 ex01]$
```

Exercises

Ex01

- Submit the job script:

```
[user@front02 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front02 ex01]$
```

- Check status of job:

```
[user@front02 ex01]$ squeue -u $USER
JOBID PARTITION      NAME      USER ST      TIME   NODES NODELIST(REASON)
69712      cpu      01      user    R      0:00      2 cn[01-02]
[user@front02 ex01]$
```

- If status is CF, i.e. "configuring", you have been allocated nodes that were in power-saving mode and are now booting. It may take several minutes until they boot.

Exercises

Ex01

- Submit the job script:

```
[user@front02 ex01]$ sbatch sub-01.sh
Submitted batch job 69711
[user@front02 ex01]$
```

- Check status of job:

```
[user@front02 ex01]$ squeue -u $USER
JOBID PARTITION      NAME      USER ST      TIME   NODES NODELIST(REASON)
69712      cpu        01      user    R      0:00      2 cn[01-02]
[user@front02 ex01]$
```

- If status is `CF`, i.e. "configuring", you have been allocated nodes that were in power-saving mode and are now booting. It may take several minutes until they boot.
- Otherwise, the job runs very quickly. You may see no output above if the job has finished:

```
[user@front02 ex01]$ squeue -u $USER
JOBID PARTITION      NAME      USER ST      TIME   NODES NODELIST(REASON)
[user@front02 ex01]$
```

Exercises

Ex01

- If done, the file `ex01-output.txt` should have been created
- Inspect the file:

```
[user@front02 ex01]$ cat ex01-output.txt
This is rank = 2 of nproc = 8 on node: cn03
This is rank = 3 of nproc = 8 on node: cn03
This is rank = 4 of nproc = 8 on node: cn03
This is rank = 5 of nproc = 8 on node: cn03
This is rank = 6 of nproc = 8 on node: cn13
This is rank = 0 of nproc = 8 on node: cn13
This is rank = 1 of nproc = 8 on node: cn13
This is rank = 7 of nproc = 8 on node: cn13
[user@front02 ex01]$
```

Exercises

Ex01

- If done, the file `ex01-output.txt` should have been created
- Inspect the file:

```
[user@front02 ex01]$ cat ex01-output.txt
This is rank = 2 of nproc = 8 on node: cn03
This is rank = 3 of nproc = 8 on node: cn03
This is rank = 4 of nproc = 8 on node: cn03
This is rank = 5 of nproc = 8 on node: cn03
This is rank = 6 of nproc = 8 on node: cn13
This is rank = 0 of nproc = 8 on node: cn13
This is rank = 1 of nproc = 8 on node: cn13
This is rank = 7 of nproc = 8 on node: cn13
[user@front02 ex01]$
```

- Note the order is nondeterministic; whichever process reaches the print statement first prints

Exercises

Ex02

- `ex02.py` demonstrates the use of `Scatter()` and `Reduce()`
- The file with name `array.txt` includes 55,440 floating point numbers, one per line:

```
[user@front02 02]$ head array.txt
7.913676052329088328e-01
1.879167007836126668e-01
2.343674804515035737e-01
4.707043244181141617e-02
6.272795840838938375e-01
2.725799268304553991e-01
5.803516013116442052e-01
2.356271465482765448e-01
2.982738904468156260e-01
5.372364132030218453e-01
[ikoutsou@front02 02]$
```

Exercises

Ex02

- We would like:

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`
 - Each process to initialize an empty array `sub[]` with number of elements `nloc = ntot / size`

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`
 - Each process to initialize an empty array `sub[]` with number of elements `nloc = ntot / size`
 - The root process to scatter the elements of array `array[]` to all processes
 - ↳ Each process should receive `nloc` elements

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`
 - Each process to initialize an empty array `sub[]` with number of elements `nloc = ntot / size`
 - The root process to scatter the elements of array `array[]` to all processes
 - ↳ Each process should receive `nloc` elements
 - Each process to sum its local elements, storing the result into `sum_loc`

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`
 - Each process to initialize an empty array `sub[]` with number of elements `nloc = ntot / size`
 - The root process to scatter the elements of array `array[]` to all processes
 - ↳ Each process should receive `nloc` elements
 - Each process to sum its local elements, storing the result into `sum_loc`
 - To use a reduction operation to obtain the grand total over all 55,440 elements in the root rank

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`
 - Each process to initialize an empty array `sub[]` with number of elements `nloc = ntot / size`
 - The root process to scatter the elements of array `array[]` to all processes
 - ↳ Each process should receive `nloc` elements
 - Each process to sum its local elements, storing the result into `sum_loc`
 - To use a reduction operation to obtain the grand total over all 55,440 elements in the root rank
- Look at `ex02.py`. You **only need to complete some parts**, as instructed by the comments with `TODO`

Exercises

Ex02

- We would like:
 - The root process to read all elements into an array `array[]`
 - The root process to *broadcast* the total number of elements of the array, `ntot`
 - Each process to initialize an empty array `sub[]` with number of elements `nloc = ntot / size`
 - The root process to scatter the elements of array `array[]` to all processes
 - ↳ Each process should receive `nloc` elements
 - Each process to sum its local elements, storing the result into `sum_loc`
 - To use a reduction operation to obtain the grand total over all 55,440 elements in the root rank
- Look at `ex02.py`. You **only need to complete some parts**, as instructed by the comments with `TODO`
- The correct result, which will be in `ex02-output.txt` should be:

```
Sum: 27777.25711
```

Exercises

Ex03

- This exercise demonstrates `Gather()`
- A file `filenames.txt` includes the filenames of 8 files:

```
[user@front02 ex03]$ cat filenames.txt  
00.txt  
01.txt  
02.txt  
03.txt  
04.txt  
05.txt  
06.txt  
07.txt  
[user@front02 ex03]$
```

Exercises

Ex03

- This exercise demonstrates `Gather()`
- A file `filenames.txt` includes the filenames of 8 files:

```
[user@front02 ex03]$ cat filenames.txt  
00.txt  
01.txt  
02.txt  
03.txt  
04.txt  
05.txt  
06.txt  
07.txt  
[user@front02 ex03]$
```

- In `ex03.py`, the root process (process with `rank = 0`) reads the filenames and scatters one to each process

Exercises

Ex03

- This exercise demonstrates `Gather()`
- A file `filenames.txt` includes the filenames of 8 files:

```
[user@front02 ex03]$ cat filenames.txt  
00.txt  
01.txt  
02.txt  
03.txt  
04.txt  
05.txt  
06.txt  
07.txt  
[user@front02 ex03]$
```

- In `ex03.py`, the root process (process with `rank = 0`) reads the filenames and scatters one to each process
- Each process then computes the Fletcher 32 checksum of one file

Exercises

Ex03

- This exercise demonstrates `Gather()`
- A file `filenames.txt` includes the filenames of 8 files:

```
[user@front02 ex03]$ cat filenames.txt  
00.txt  
01.txt  
02.txt  
03.txt  
04.txt  
05.txt  
06.txt  
07.txt  
[user@front02 ex03]$
```

- In `ex03.py`, the root process (process with `rank = 0`) reads the filenames and scatters one to each process
- Each process then computes the Fletcher 32 checksum of one file
- You need to write an appropriate `Gather()` operation to collect the checksums into the root process such that it prints them correctly

Exercises

Ex03

- If done correctly, `ex03-output.txt` should include the following output:

```
[user@front02 ex03]$ cat ex03-output.txt
[user@front02 ex03]$
00.txt → 04D70552
01.txt → 19708CD4
02.txt → ED737A1C
03.txt → 0C40E2D2
04.txt → F7BDE74D
05.txt → 562DDD6C
06.txt → 6F2CD2F1
07.txt → 016DB6C6
[user@front02 03]$
```

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:
 - The array in each rank has two additional elements, `sub = np.empty(shape=(nloc+2))`, where `nloc = ntot / size`

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:
 - The array in each rank has two additional elements, `sub = np.empty(shape=(nloc+2))`, where `nloc = ntot / size`
 - Add a missing `Scatter()` to distribute to each processes the corresponding `nloc` elements. These should be received in `sub[1:-1]` of each rank

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:
 - The array in each rank has two additional elements, `sub = np.empty(shape=(nloc+2))`, where `nloc = ntot / size`
 - Add a missing `Scatter()` to distribute to each processes the corresponding `nloc` elements. These should be received in `sub[1:-1]` of each rank
 - Use `Sendrecv()` to fill the first and last elements element of `sub[]`, i.e. `sub[0]` and `sub[-1]`, with the corresponding elements of the neighboring ranks, needed to carry out the derivative

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:
 - The array in each rank has two additional elements, `sub = np.empty(shape=(nloc+2))`, where `nloc = ntot / size`
 - Add a missing `Scatter()` to distribute to each processes the corresponding `nloc` elements. These should be received in `sub[1:-1]` of each rank
 - Use `Sendrecv()` to fill the first and last elements element of `sub[]`, i.e. `sub[0]` and `sub[-1]`, with the corresponding elements of the neighboring ranks, needed to carry out the derivative
 - Now your program can proceed to correctly compute the derivative:

```
deriv_loc[i] = sub[i-1] - 2*sub[i] + sub[i+1]
```

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:
 - The array in each rank has two additional elements, `sub = np.empty(shape=(nloc+2))`, where `nloc = ntot / size`
 - Add a missing `Scatter()` to distribute to each processes the corresponding `nloc` elements. These should be received in `sub[1:-1]` of each rank
 - Use `Sendrecv()` to fill the first and last elements element of `sub[]`, i.e. `sub[0]` and `sub[-1]`, with the corresponding elements of the neighboring ranks, needed to carry out the derivative
 - Now your program can proceed to correctly compute the derivative:

```
deriv_loc[i] = sub[i-1] - 2*sub[i] + sub[i+1]
```

- Use a `Gather()` to collect the full array of the derivative into the root process (rank = 0). The root process will then write the array to a new file `deriv.txt`

Exercises

Ex04

- In this exercise, a large array of 55,440 elements is read by rank 0 (`ntot = 55400`)
- Our objective is to compute a second order discrete derivative of the data:

```
deriv[i] = array[i-1] + array[i+1] - 2*array[i]
```

- The goal is to do this *in parallel*, such that each rank computed it for a *subset of the data*
- We will proceed as follows:
 - The array in each rank has two additional elements, `sub = np.empty(shape=(nloc+2))`, where `nloc = ntot / size`
 - Add a missing `Scatter()` to distribute to each processes the corresponding `nloc` elements. These should be received in `sub[1:-1]` of each rank
 - Use `Sendrecv()` to fill the first and last elements element of `sub[]`, i.e. `sub[0]` and `sub[-1]`, with the corresponding elements of the neighboring ranks, needed to carry out the derivative
 - Now your program can proceed to correctly compute the derivative:

```
deriv_loc[i] = sub[i-1] - 2*sub[i] + sub[i+1]
```

- Use a `Gather()` to collect the full array of the derivative into the root process (rank = 0). The root process will then write the array to a new file `deriv.txt`
- If done correctly, `deriv.txt` will contain all zeros, except for the first and last element

Exercises

Ex04

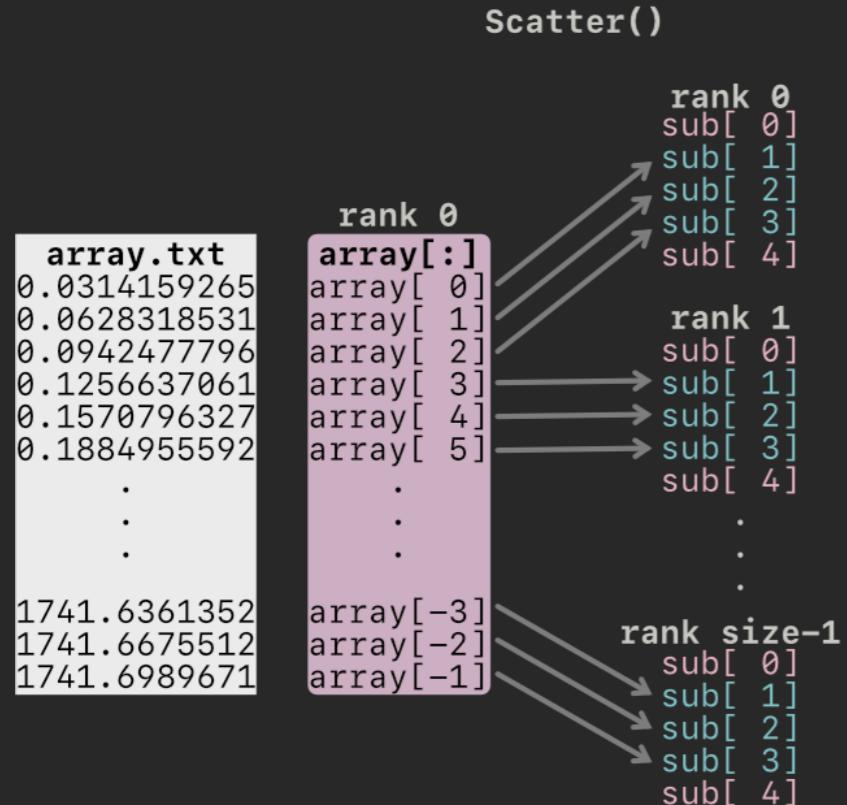
Exercises

Ex04

array.txt	rank 0
0.0314159265	array[::]
0.0628318531	array[0]
0.0942477796	array[1]
0.1256637061	array[2]
0.1570796327	array[3]
0.1884955592	array[4]
.	array[5]
.	.
.	.
1741.6361352	array[-3]
1741.6675512	array[-2]
1741.6989671	array[-1]

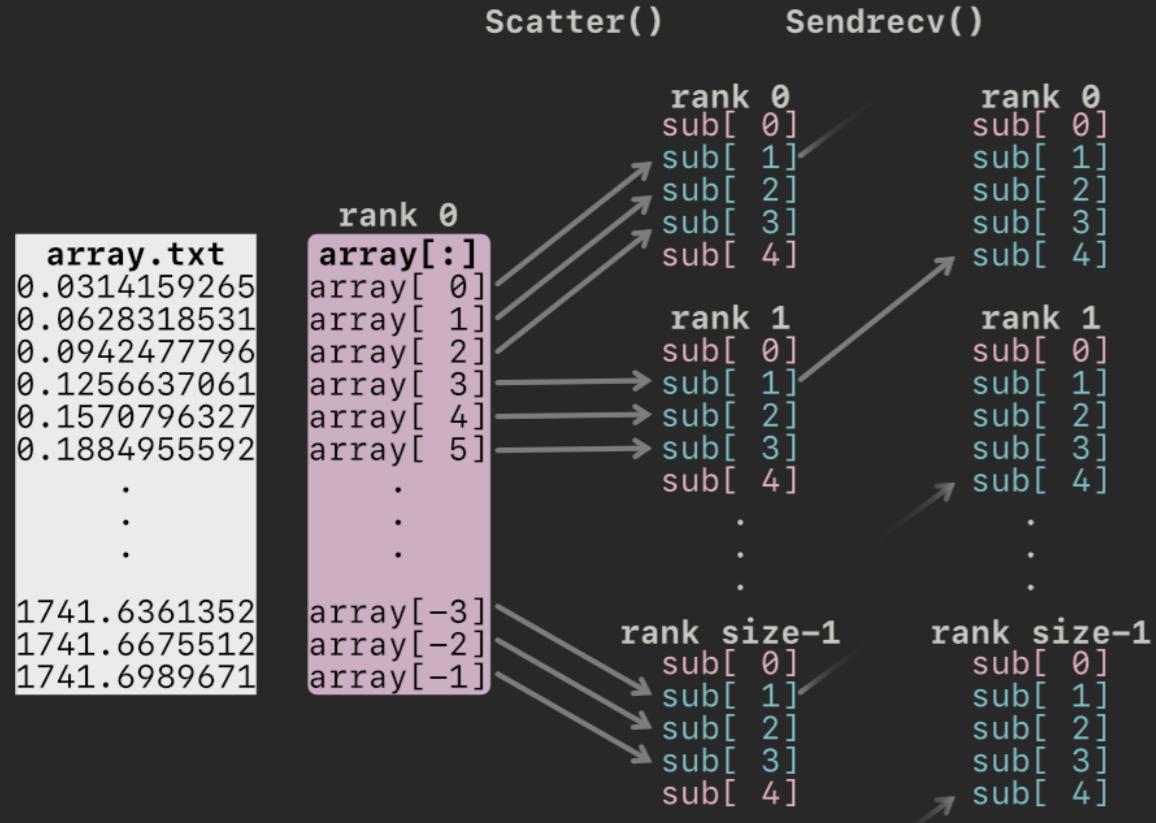
Exercises

Ex04



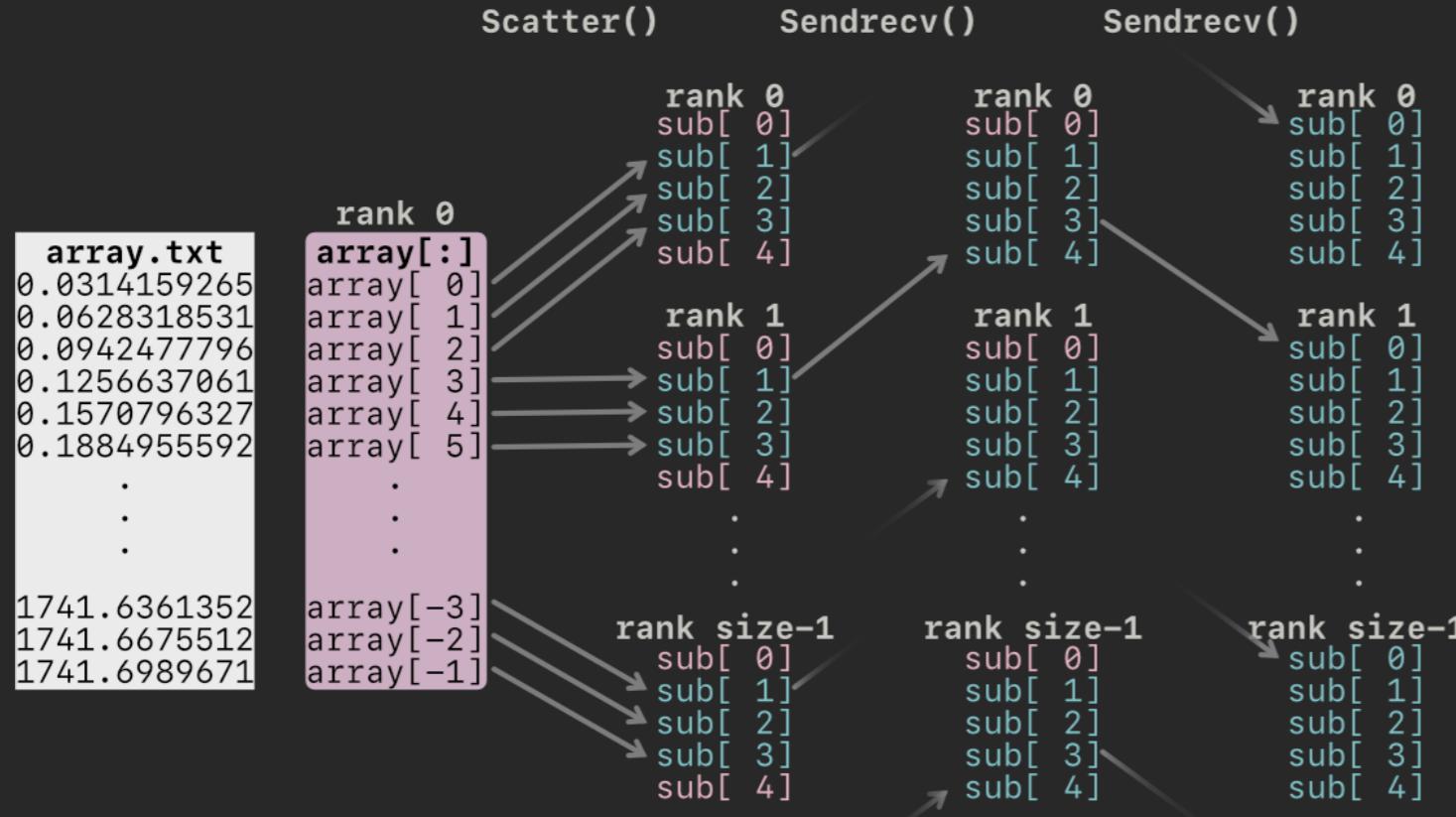
Exercises

Ex04



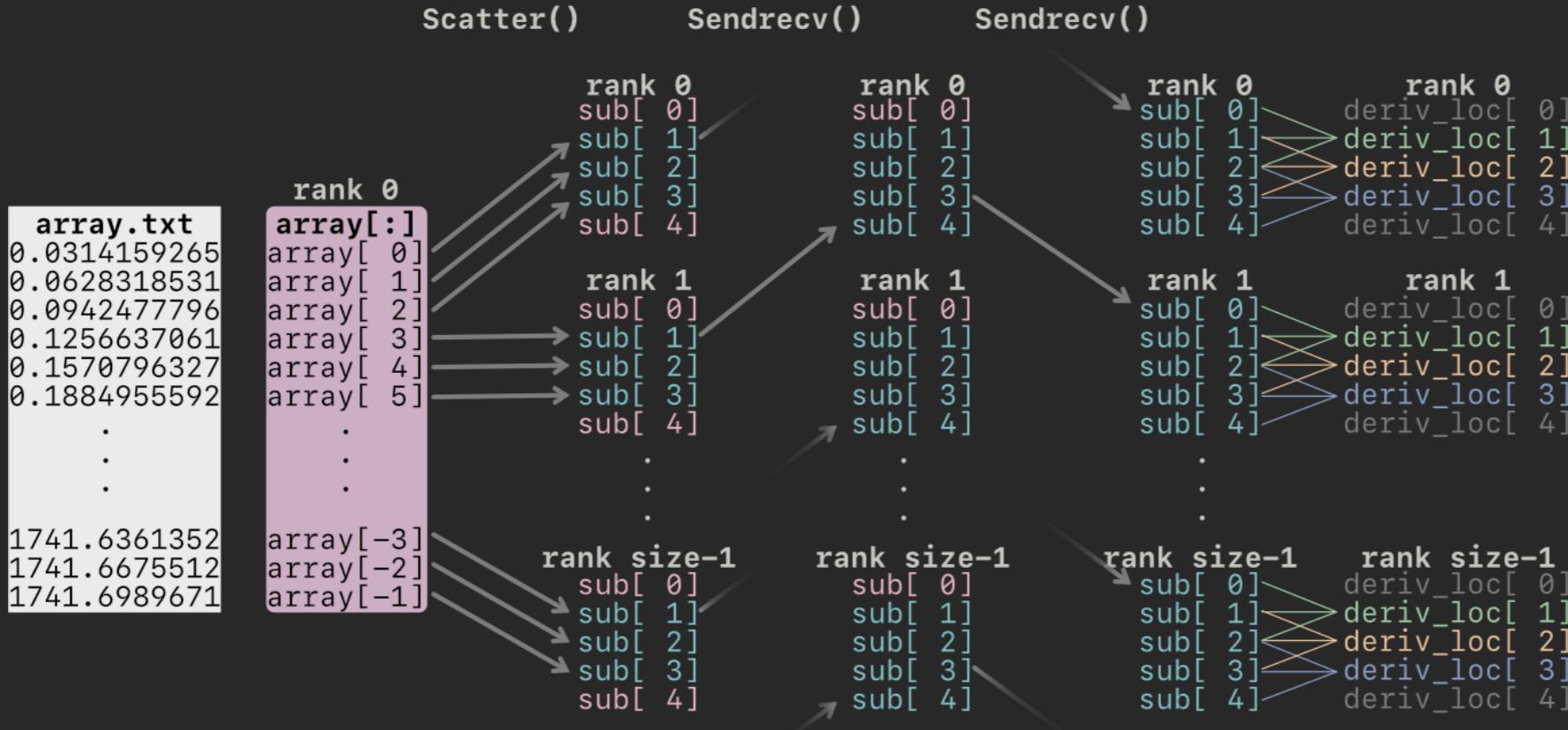
Exercises

Ex04



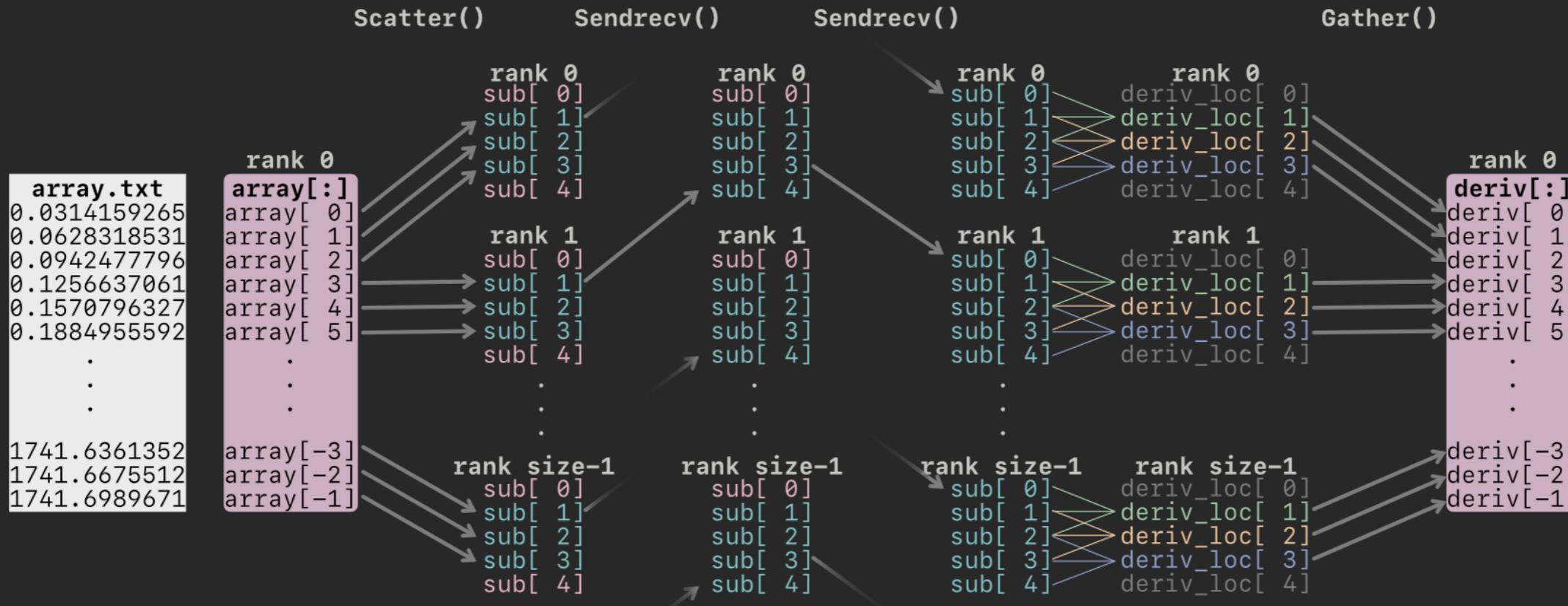
Exercises

Ex04



Exercises

Ex04



Exercises

Ex04

