Introduction to
MPI - Message Passing Interface

Antony Spyropoulos

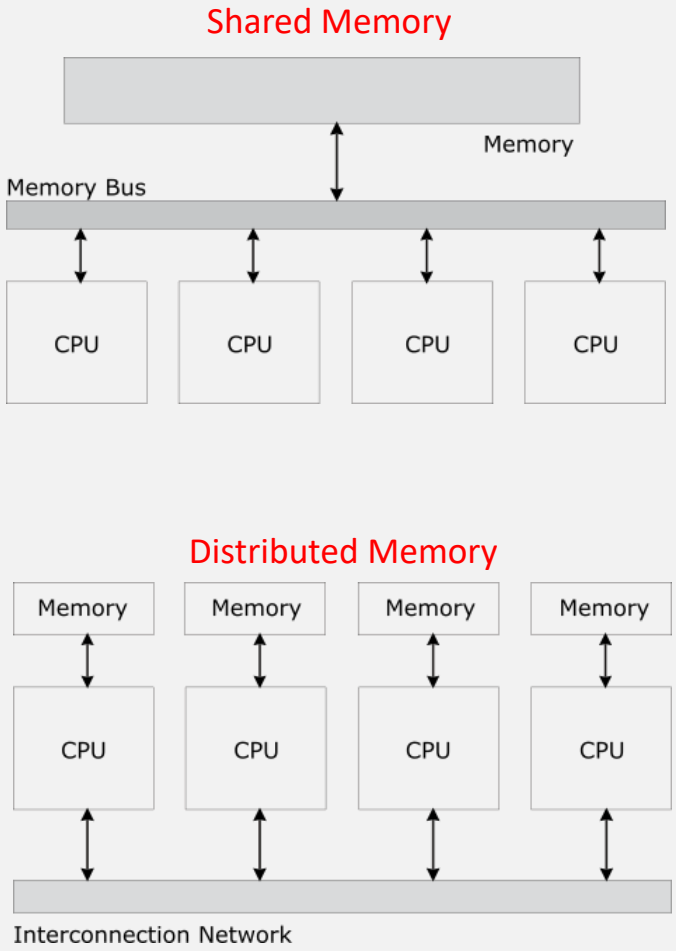
Laboratory Teaching Staff,
Department of Electrical & Computer Engineering,
University of Thessaly
email: aspyr@uth.gr

Two Parallel Programming Models

Shared Memory
- Multiple workers share the same memory
- Communication through shared variables

Distributed Memory
- Each worker has its own local memory
- Communication through message passing

Shared memory → OpenMP → threads
Distributed memory → MPI → processes



MPI is designed for distributed memory systems.

MPI Processes

- Each process **runs the same program**
- Each process has its **own memory**
- Each process has an unique ID called **rank**
  - The **rank** is an integer: 0, 1, 2, ...
  - We use the **rank** to **identify processes**

In MPI, we start multiple processes that run the same program.

The MPI Development Workflow

1. WRITE

Create your program in a text file

Include MPI calls to enable communication between processes

2. COMPILE

Use an MPI compiler wrapper

```
mpif90 -o my_program my_program.f90 (Fortran)
mpic++ -o my_program my_program.cpp (C++)
```

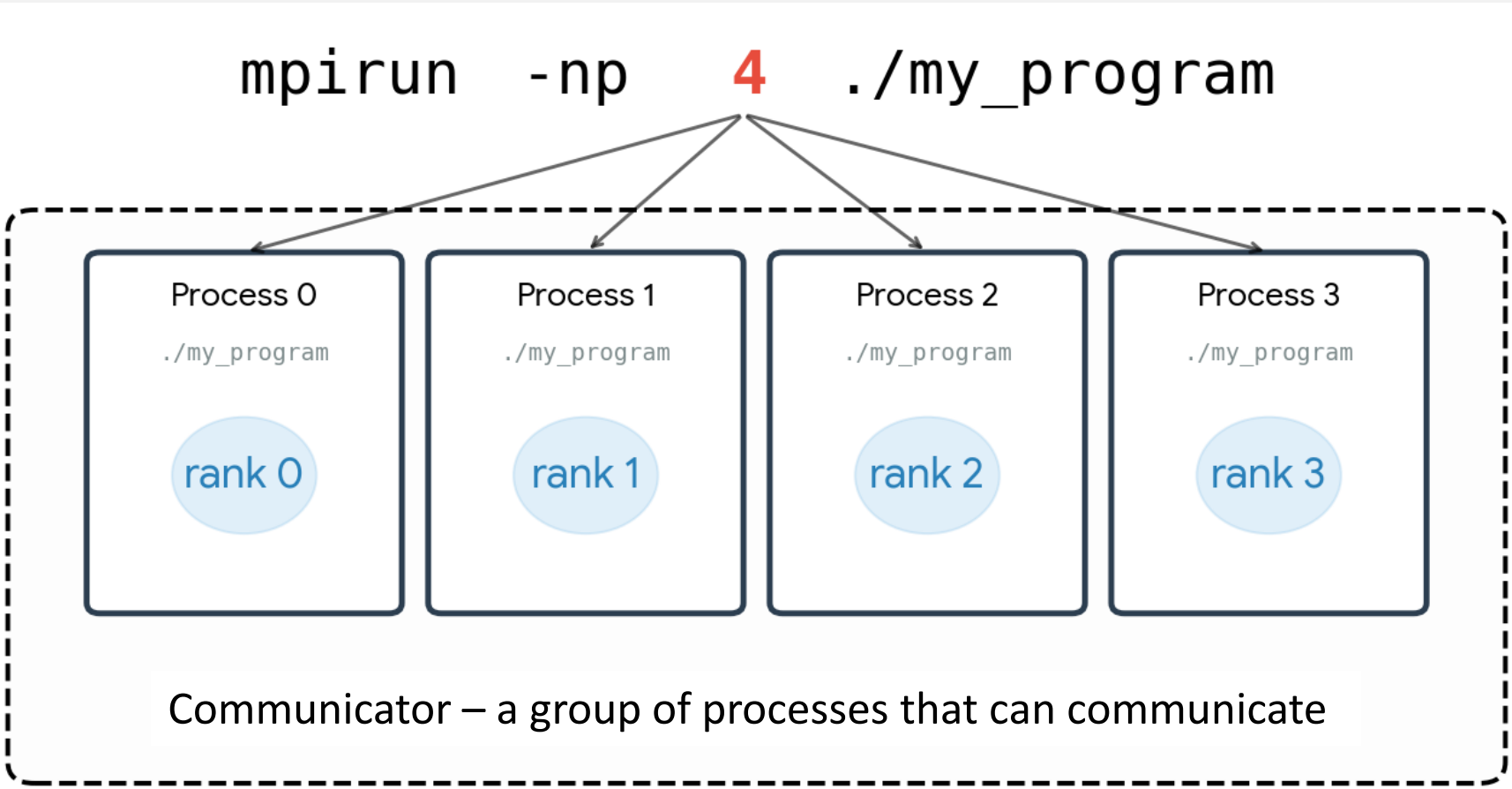
Automatically links MPI libraries

3. EXECUTE

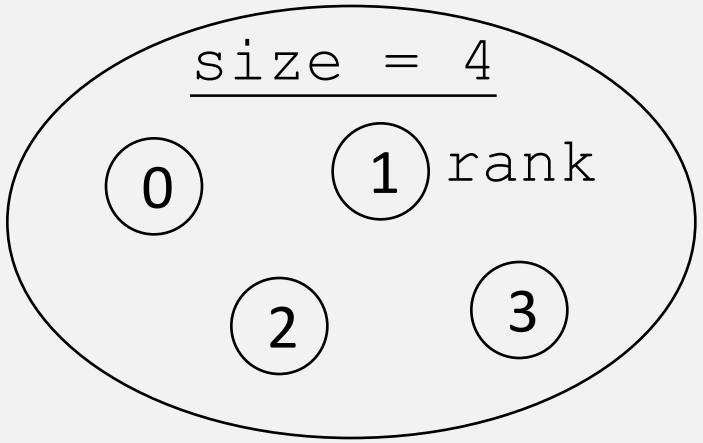
```
mpirun -np 4 ./my_program
```

Starts **4** processes that run the same program

The Execution Model



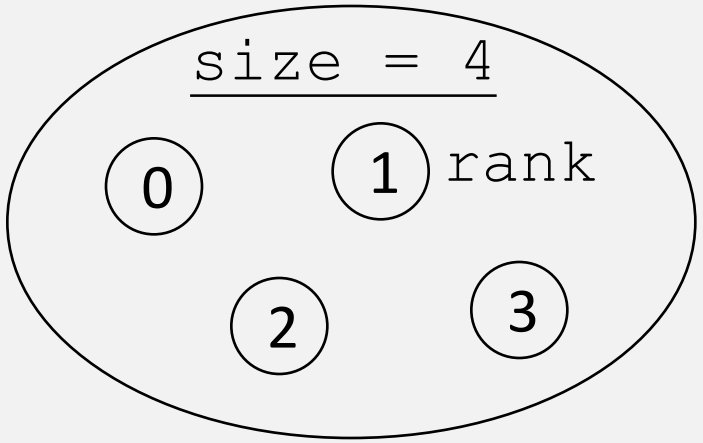
Structure of an MPI Program (Fortran)




```
include 'mpif.h'


call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)

... parallel work ...

call MPI_FINALIZE(error)
```

Structure of an MPI Program (C/C++)

MPI_COMM_WORLD (communicator)




```
#include <mpi.h>


MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

... parallel work ...

MPI_Finalize();
```

Example 1: My first MPI code

```
program hello
implicit NONE
include 'mpif.h'
integer rank, size, error

call MPI_INIT(error)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)

print *, 'Hello world from ', rank

call MPI_FINALIZE(error)

end
```

Example 2: My second code in MPI

```
program hello2
implicit NONE
include 'mpif.h'
integer rank, size, error

call MPI_INIT(error)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)

print *, 'Hello world from ', rank

if (rank == 0) then
  print *, 'Number of processes:', size
endif

call MPI_FINALIZE(error)

end
```

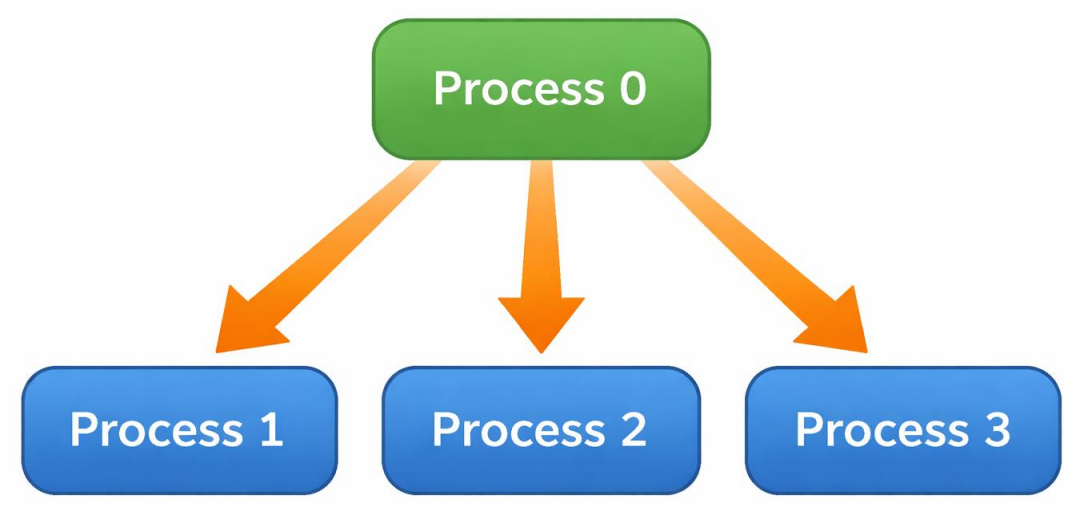
MPI Communication Types

Point-to-Point

One process sends data to another process

Collective

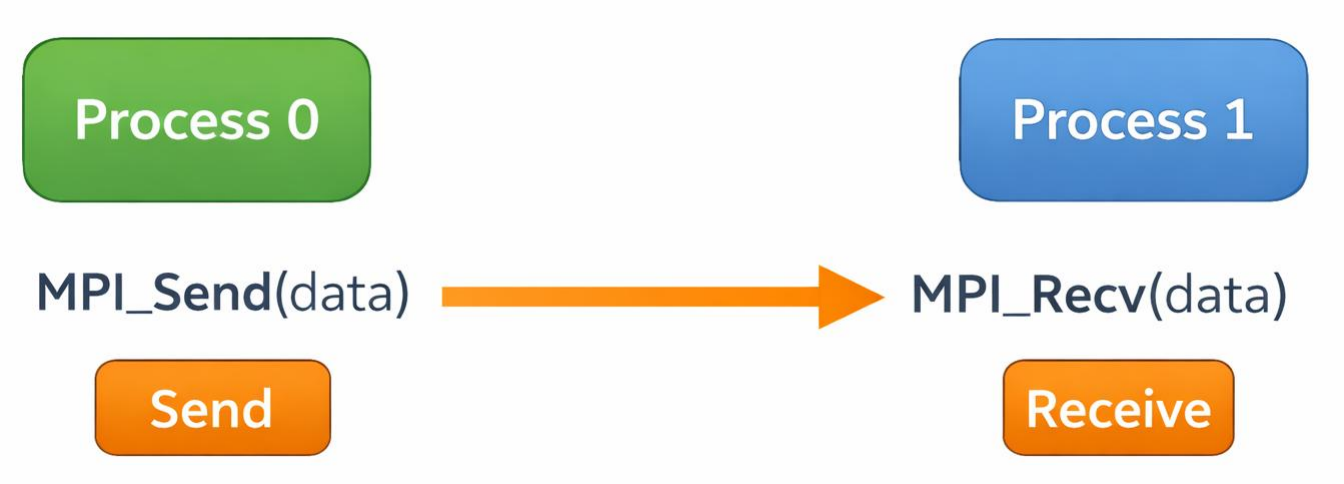
All processes participate together



Point-to-Point communication

Communication between **two processes**

- One process **sends** data
- One process **receives** data



One process sends a message (data) and another process receives it.

Point-to-Point communication: MPI_SEND

```
MPI_SEND(data, count, datatype, dest, tag, comm, error)
```

Arguments

- `data` → data to send
- `count` → number of elements
- `datatype` → type of data
- `dest` → destination process
- `tag` → message label
- `comm` → communicator

```
datatype: MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, ...
comm: MPI_COMM_WORLD
```

Point-to-Point communication: MPI_RECV

```
MPI_RECV(data, count, datatype, source, tag, comm, status, error)
```

Arguments

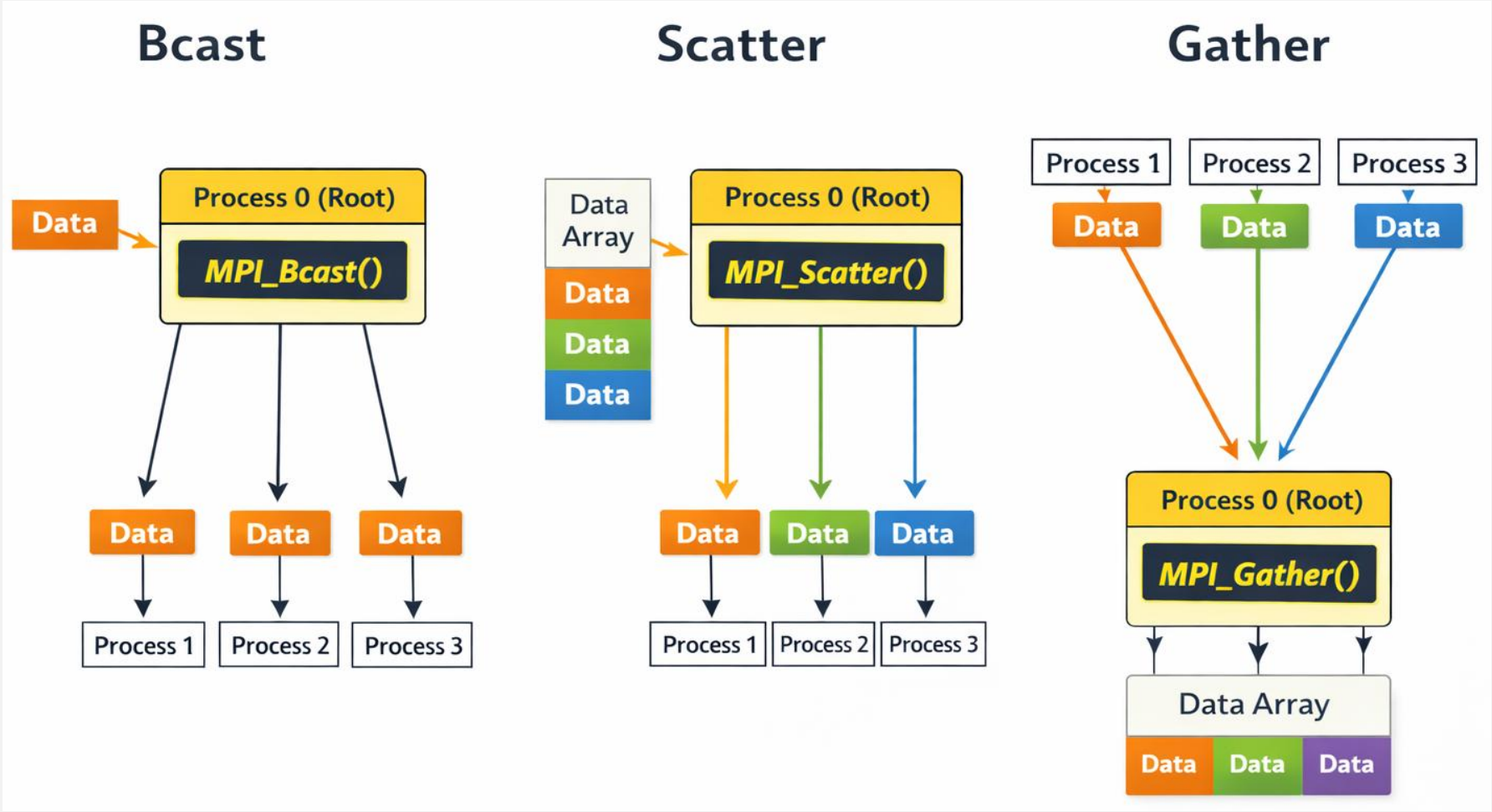
- `data` → buffer to receive data
- `count` → number of elements
- `datatype` → type of data
- `source` → source process
- `tag` → message label
- `comm` → communicator
- `status` → contains information about the received message

Point-to-Point communication: Simple Example

```
tag = 0

if (rank == 0) then
  dest = 1
  data = 10
  call MPI_SEND(data, 1, MPI_INTEGER, dest, tag, &
                MPI_COMM_WORLD, error)
endif

if (rank == 1) then
  source = 0
  call MPI_RECV(data, 1, MPI_INTEGER, source, tag, &
                MPI_COMM_WORLD, status, error)
endif
```

Every send must have a matching receive.

Collective communication: Broadcast, Scatter and Gather



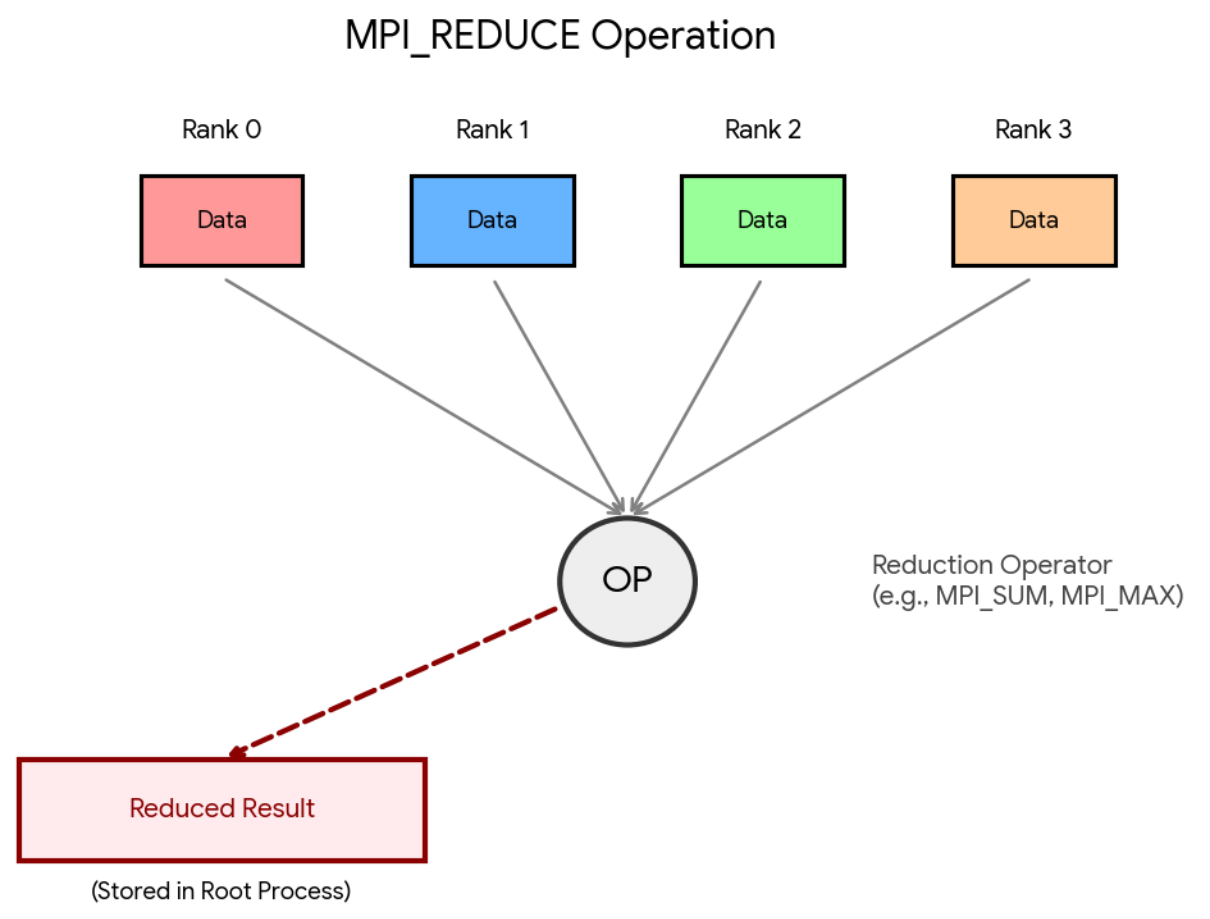
Example: `MPI_BCAST(data, count, datatype, root, comm, error)`

Example 3: MPI_BCAST

```
program broadcast
implicit NONE
include 'mpif.h'
integer rank, size, error, root
integer value

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)

root = 0

if (rank == root) then
  value=10
endif

call MPI_BCAST(value, 1, MPI_INTEGER, root, MPI_COMM_WORLD, error)

print *, 'Process', rank, 'has value', value

call MPI_FINALIZE(error)

end
```

Collective communication: MPI_REDUCE

Combines data from all processes into a single result using a specific operator (MPI_SUM, MPI_MAX, etc.) and stores it on the root process.



Collective communication: MPI_REDUCE

```
MPI_REDUCE(local_data, reduced_data, count, datatype, op,
           root, comm, error)
```

Arguments

`local_data` → data from each process
`reduced_data` → buffer at root to store the final result
`count` → number of elements
`datatype` → type of data
`op` → operation to combine values
`root` → process that receives the final result
`comm` → communicator

op: `MPI_MAX, MPI_MIN, MPI_SUM, ...`

Example 4: MPI_REDUCE

```
program reduce
implicit NONE
include 'mpif.h'
integer rank, size, error, root
integer local_data, reduced_data

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)

local_data = rank
root = 0

call MPI_REDUCE(local_data, reduced_data, 1, MPI_INTEGER, MPI_MAX, &
                root, MPI_COMM_WORLD, error)

if (rank == root) then
  print *, reduced_data
endif

call MPI_FINALIZE(error)

end
```

Dot Product with MPI

Given:

$$x = [x_1, x_2, \dots, x_N] \epsilon \boldsymbol{R}^N, \quad x_i = \frac{i}{i+1}, \quad i = 1,2, \dots, N$$

Goal:

Compute the dot product $x^T x = x_1^2 + x_2^2 + \cdots + x_N^2$
using P processes (assume N divisible by P)

Data partitioning across processes

Each process handles $N_{local} = {}^{N}/_{P}$ elements of the vector x

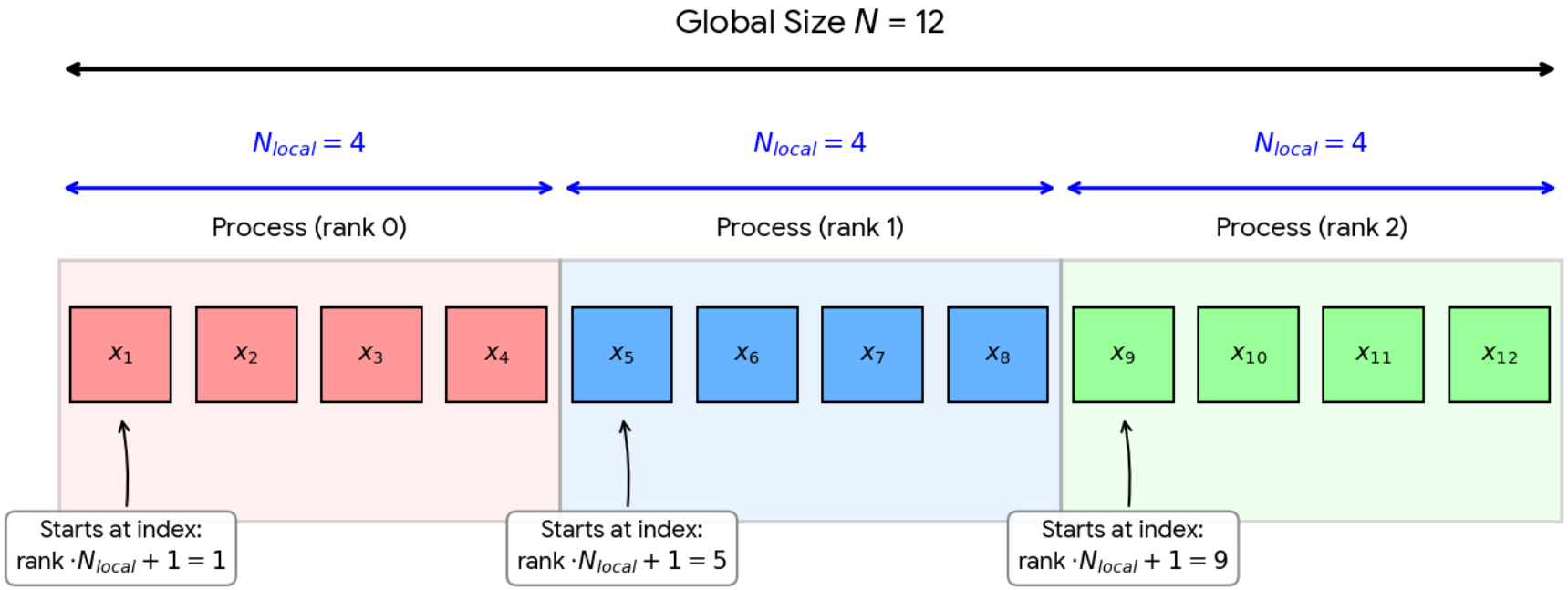
Elements before the process's part of the vector x:

$$rank \cdot N_{local}$$

Elements handled by each process:

$$\frac{rank \cdot N_{local} + i}{rank \cdot N_{local} + i+1}, \quad i = 1,2, \dots, N_{local}$$

Distribution of data across processes



Global Index = (rank · N_{local}) + i

MPI Program for Dot Product

```
program dot_product
implicit NONE
include 'mpif.h'
integer, parameter :: N = 10000
integer rank, size, error, i
integer N_local, elements_before
real(8), allocatable, dimension(:) :: x_local
real(8) sum_local, sum_total

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)

!Partitioning
  N_local = N / size
  elements_before = rank*N_local

  allocate( x_local(N_local) )
  do i=1,N_local
    x_local(i) = ( elements_before + i )/( elements_before + i + 1.0d0 )
  enddo
```

MPI Program for Dot Product

```
! Main computation done here
! Gains: loop runs only from 1 to N_local instead of 1 to N
! Memory-efficient: only local array x_local is allocated, not full x(N)
   sum_local = 0.0d0
   do i=1,N_local
     sum_local = sum_local + x_local(i)**2
   enddo

! Communication
   call MPI_REDUCE(sum_local, sum_total, 1, MPI_DOUBLE_PRECISION, &
                   MPI_SUM, 0, MPI_COMM_WORLD, error)

   if (rank == 0) then
     print *, 'Dot product = ', sum_total
   endif

   call MPI_FINALIZE(error)
end
```

Communication Time

Communication time = System overhead + Synchronization overhead

System overhead

- Time spent for **data transfer**
- Depends on the **network performance**

(We need a fast network!)

Synchronization overhead

- Time spent **waiting for other processes**

(We need good programming practices!)

Why it matters

- Present only in parallel programs
- **Slows down** execution compared to serial code
- Minimizing it is key to good parallel performance

Fast networks help, but good programming practices matter even more.

Workshop Announcement

HPC Workshop

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

EuroCC@Greece

Large-Scale Scientific Computations

School of Chemical Engineering
Computer Center

School of Mechanical Engineering
Parallel CFD and Optimization Unit

School of Electrical and Computer Engineering
Computing Systems Laboratory

Designed for students with some programming experience to learn three parallel programming models: **MPI | CUDA | OpenMP**

- **Beginner-friendly**, but includes advanced topics like parallelization of a Krylov-type solver
- **Hands-on sessions** in C and Fortran → test and understand the main building blocks

Format: On-Site, Hands-on
When: Every year in July
Where: NTUA School of Chemical Engineering (PC-Lab)

More Info: https://hpc-workshop.chemeng.ntua.gr

EuroCC@Greece