



# An introduction to MPI

---

Ioannis E. Venetis

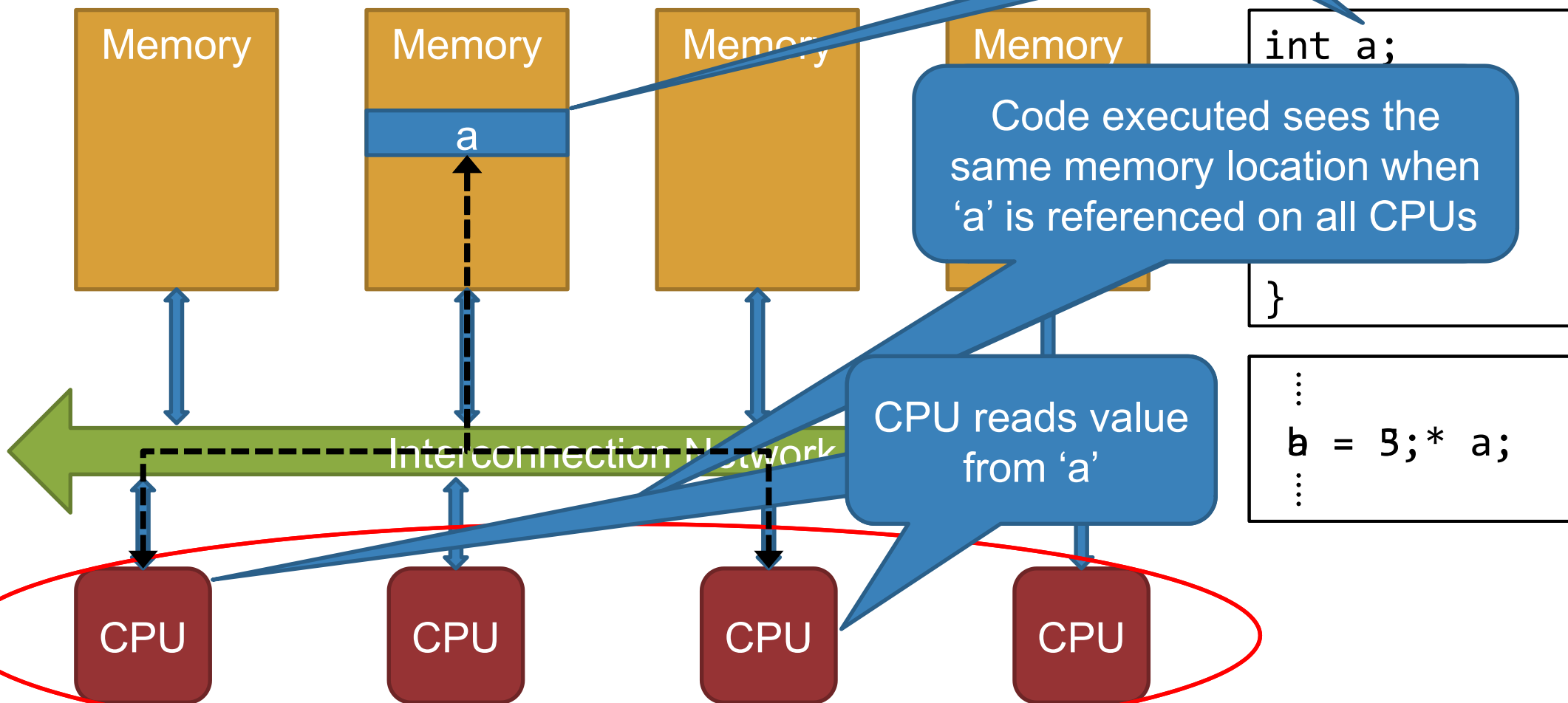
*University of Patras*



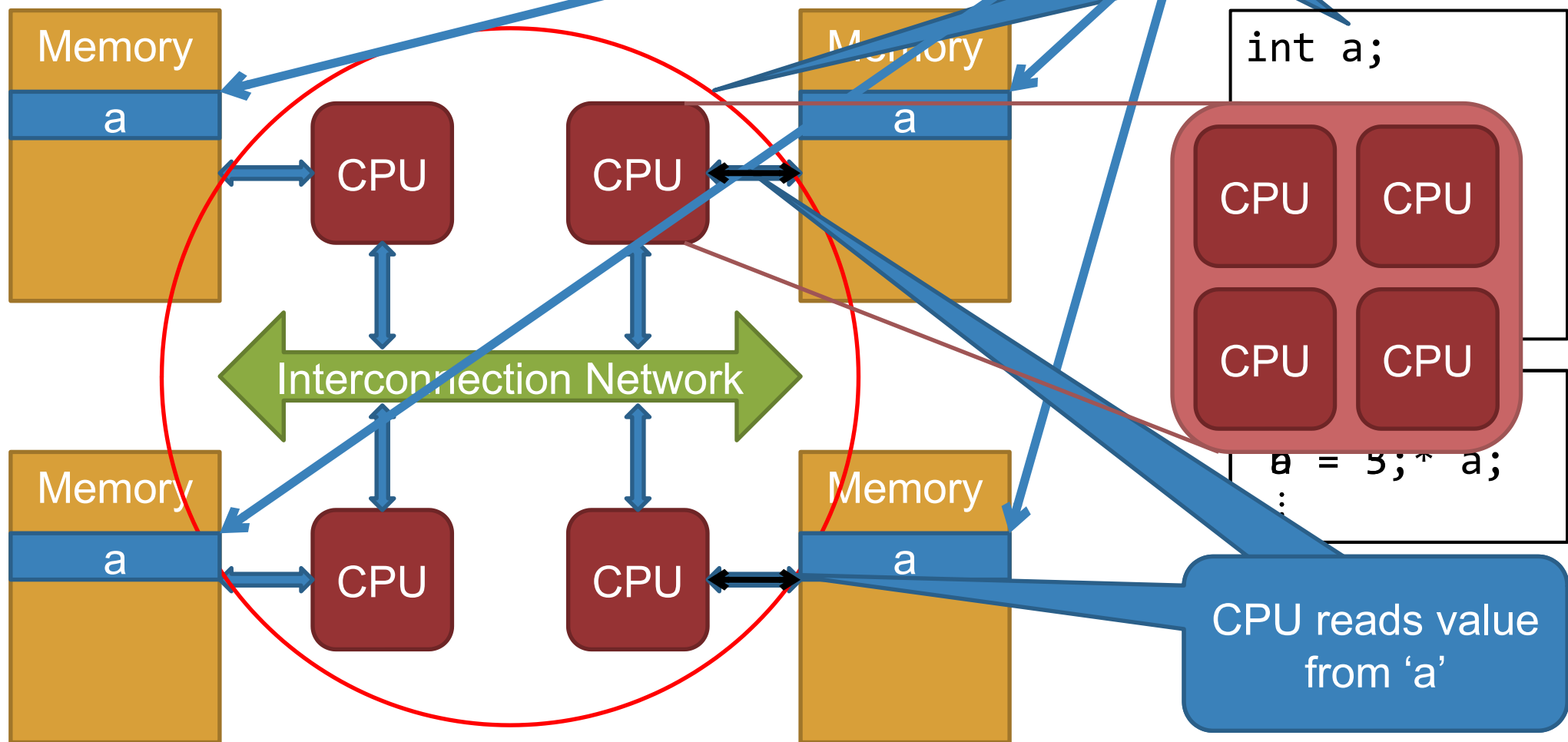
## Classification of parallel architectures

- ▶ There are many classification schemes for parallel architectures
- ▶ One of the most useful ones is according to the memory architecture of the system
  - ▶ Shared memory
  - ▶ Distributed memory
  - ▶ Distributed shared memory
- ▶ Largely determines the **programming models** that can be used to program the specific architecture

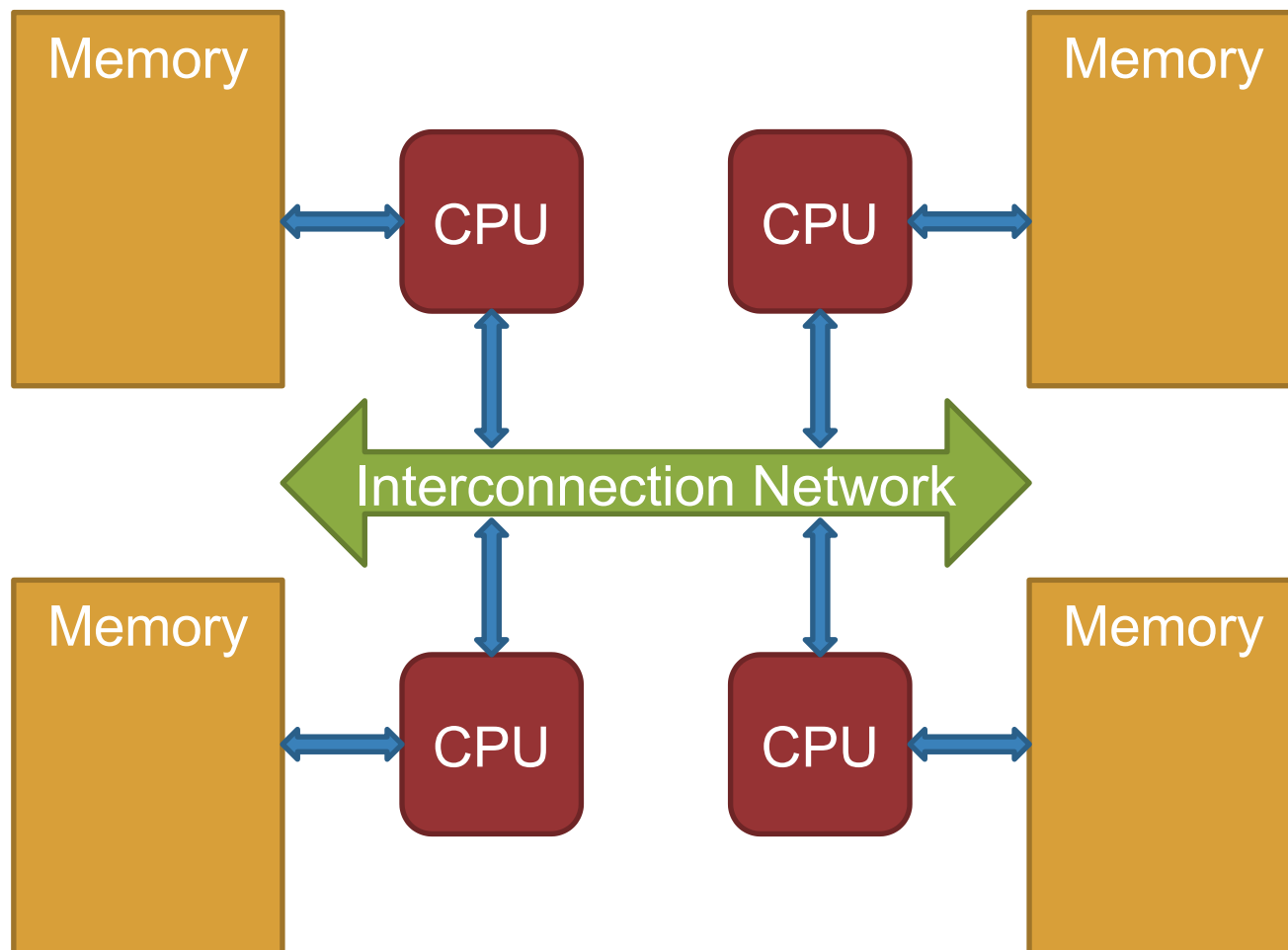
## Shared memory parallel architectures



# Distributed memory parallel architectures



## Distributed shared memory parallel architectures



- ▶ Same organization as distributed memory systems
- ▶ But memory behaves as in shared memory systems
  - ▶ Hardware ensures this kind of behavior



## Programming models for shared memory parallel architectures

- ▶ Threads
  - ▶ POSIX Threads as well as other threading libraries
- ▶ OpenMP
- ▶ Cilk
- ▶ Cilk Plus
- ▶ Threading Building Blocks
- ▶ ...



## Programming models for distributed memory parallel architectures

- ▶ MPI
- ▶ PVM
- ▶ ...

Ultimate winner!



## What is the goal when using a parallel programming model?

- ▶ Problem to be solved is large
  - ▶ Requires too much time to run on single CPU
- ▶ Divide input data into smaller chunks
  - ▶ Smaller size of input  $\Rightarrow$  Faster calculation
- ▶ Assign every chunk to a CPU
  - ▶ Using appropriate software abstraction (thread or process)
- ▶ Every CPU processes its chunk and creates part of the solution
- ▶ Combine partial solutions into global solution





## MPI (Message Passing Interface)

- ▶ It is a standard
  - ▶ Not a specific implementation
- ▶ It defines a set of functions, data structures, data types, etc. to exchange messages of data among processes
  - ▶ Implemented as a library
- ▶ Layered design
- ▶ At a high level
  - ▶ Provides an API for the programmer
- ▶ At a low level
  - ▶ Interacts with the interconnection network to send and receive data
- ▶ Supports C, C++, Fortran 77 and Fortran 90



## MPI implementations

- ▶ Open MPI: <http://www.open-mpi.org>
- ▶ MPICH: <http://www-unix.mcs.anl.gov/mpi/mpich>
- ▶ MPICH2: <http://www-unix.mcs.anl.gov/mpi/mpich2>
- ▶ MPICH-GM: <http://www.myri.com/scs>
- ▶ LAM/MPI: <http://www.lam-mpi.org>
- ▶ LA-MPI: <http://public.lanl.gov/lampi>
- ▶ SCI-MPICH: <http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH>
- ▶ MPI/Pro: <http://www.mpi-softtech.com>



## A basic subset of MPI functions

- ▶ The following six MPI functions allow implementation of any MPI program:
  - ▶ MPI\_Init
  - ▶ MPI\_Finalize
  - ▶ MPI\_Comm\_size
  - ▶ MPI\_Comm\_rank
  - ▶ MPI\_Send
  - ▶ MPI\_Recv
- ▶ Why are then other functions provided?
  - ▶ Patterns occur in parallel programming
    - ▶ *Simplification of programming*
  - ▶ Performance optimization
    - ▶ *e.g., smarter algorithms, non-blocking data exchange, ...*



## Initialization/Finalization of MPI processes

- ▶ `int MPI_Init(int *argc, char ***argv);`
  - ▶ Initializes an MPI program
  - ▶ Must be called by every MPI process
    - ▶ *Must be called exactly once*
    - ▶ *Must be called before calling any other MPI function*
  - ▶ Initializes the implementation of MPI in use
- ▶ `int MPI_Finalize(void);`
  - ▶ Finalizes an MPI program
  - ▶ Must be called by every MPI process
    - ▶ *Must be called exactly once*
  - ▶ Cleans up the implementation of MPI in use



## Identification of MPI processes

- ▶ `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - ▶ Stores in variable `size` the number of MPI processes that constitute the `communicator` defined by the parameter `comm`
  
- ▶ `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - ▶ Stores in variable `rank` the rank of the calling MPI process within the `communicator` defined by the parameter `comm`

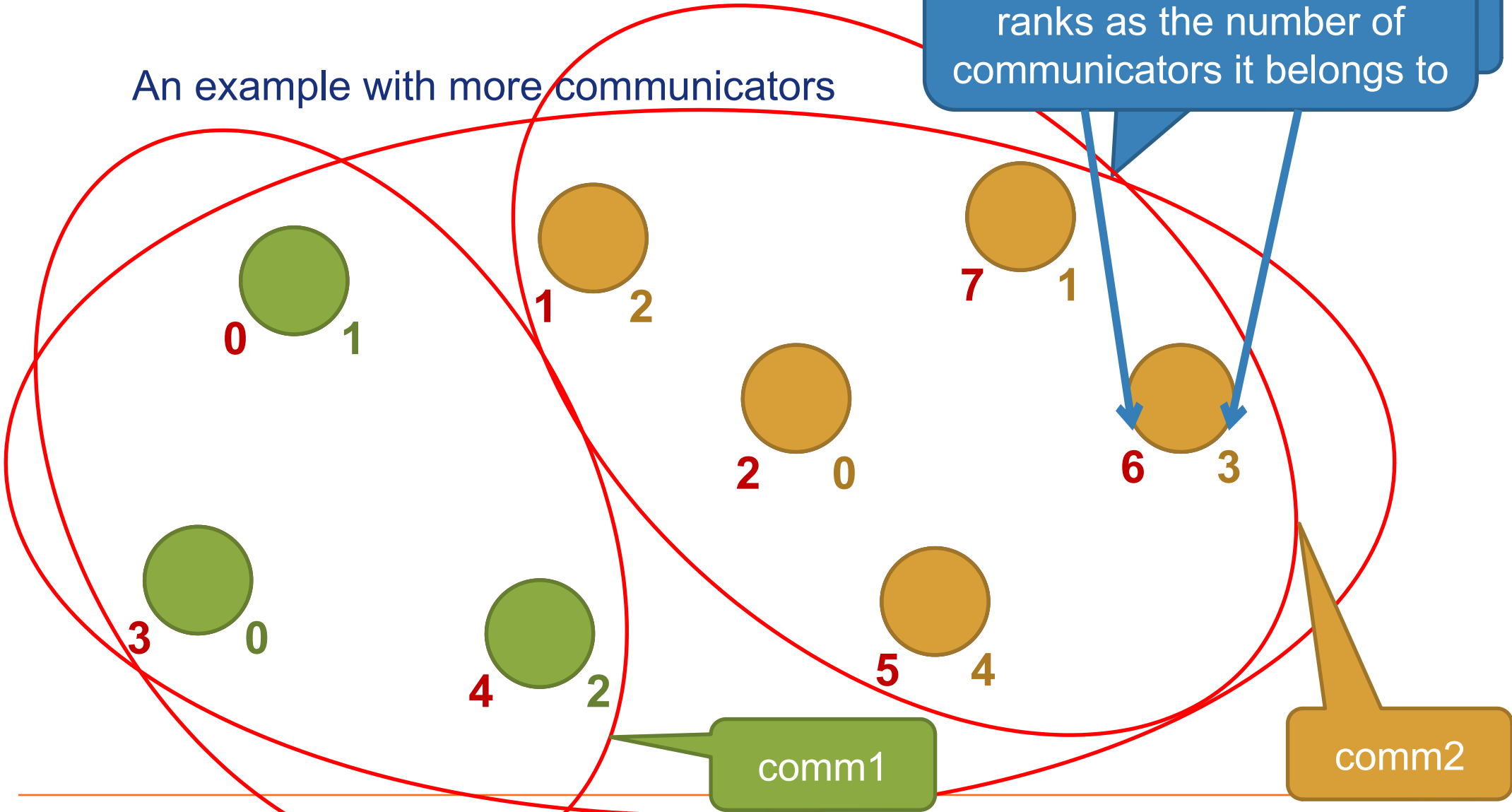


## Communicators

- ▶ Parameter **comm** in the functions presented earlier
- ▶ Defines a **communication domain**
  - ▶ A subset of the processes that constitute the MPI application
  - ▶ Processes in a subset can easily communicate
    - ▶ *Collective communication will be analyzed later*
- ▶ We will use **MPI\_COMM\_WORLD**
  - ▶ A constant defined by any implementation
  - ▶ Defines a communicator that is comprised of all processes of the MPI application

### An example with more communicators

Every process has as many ranks as the number of communicators it belongs to





## A first, simple example

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

int main(int argc, char *argv[]) {
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello world! I'm %d of %d\n", rank, size);

    MPI_Finalize();

    return(0);
}
```





## Compilation/Execution of MPI programs

- ▶ Compilation
  - ▶ `mpicc -O3 -Wall -o my_prog my_prog.c`
- ▶ Execution:
  - ▶ Typically using “mpirun” command
  - ▶ `mpirun -np 4 ./my_prog`
- ▶ Creates and executes (at least) one process on a number of nodes of the system
- ▶ Depending on the implementation of MPI there are other commands that can be used to start execution of an MPI program
- ▶ When using the SLURM batch system then “srun” should be used in the SLURM scripts
  - ▶ Works with all MPI implementations



## Differentiation of execution according to the rank

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

int main(int argc, char *argv[]) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 2) {
        printf("I am process 2 and I am different!\n");
    } else {
        printf("Hello world! I'm %d of %d\n", rank, size);
    }
    MPI_Finalize();

    return(0);
}
```

## Communication in MPI

- ▶ From one process to another: **point-to-point**
- ▶ Among sets of processes (communicators): **collective**
- ▶ Can be standard, buffered, synchronous or ready
  - ▶ When can transmission of data start, with respect to the communication operation?
- ▶ Can be blocking or non-blocking
  - ▶ When does the call to the MPI function return?
    - ▶ *Before or after the actual data transfer has taken place?*

Typically  
this is used

Typically  
this is used



# Point-to-point communication

---



## Point-to-point communication

- ▶ Exchange of data between specific processes of an MPI program
  - ▶ Identified using the ranks of these processes

## Sending a message in MPI

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

- ▶ `buf`: Starting address of data to be sent
- ▶ `count`: Number of elements to be sent
- ▶ `datatype`: Data type of each element to be sent
- ▶ `dest`: Rank of the process to receive the message
- ▶ `tag`: Tag of the message
- ▶ `comm`: Communicator

Used to determine the size (in bytes) of the message

The rank is determined according to the provided communicator



## Receiving a message in MPI

- ▶ `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,`
- ▶ `int tag, MPI_Comm comm, MPI_Status *status);`
  - ▶ `buf`: Starting address in memory where received data will be stored
  - ▶ `count`: Number of elements to be received
  - ▶ `datatype`: Data type of each element to be received
  - ▶ `source`: Rank of the process that sent the message to be received
  - ▶ `tag`: Tag of the message
  - ▶ `comm`: Communicator
  - ▶ `status`: The MPI implementation can return details about the message received



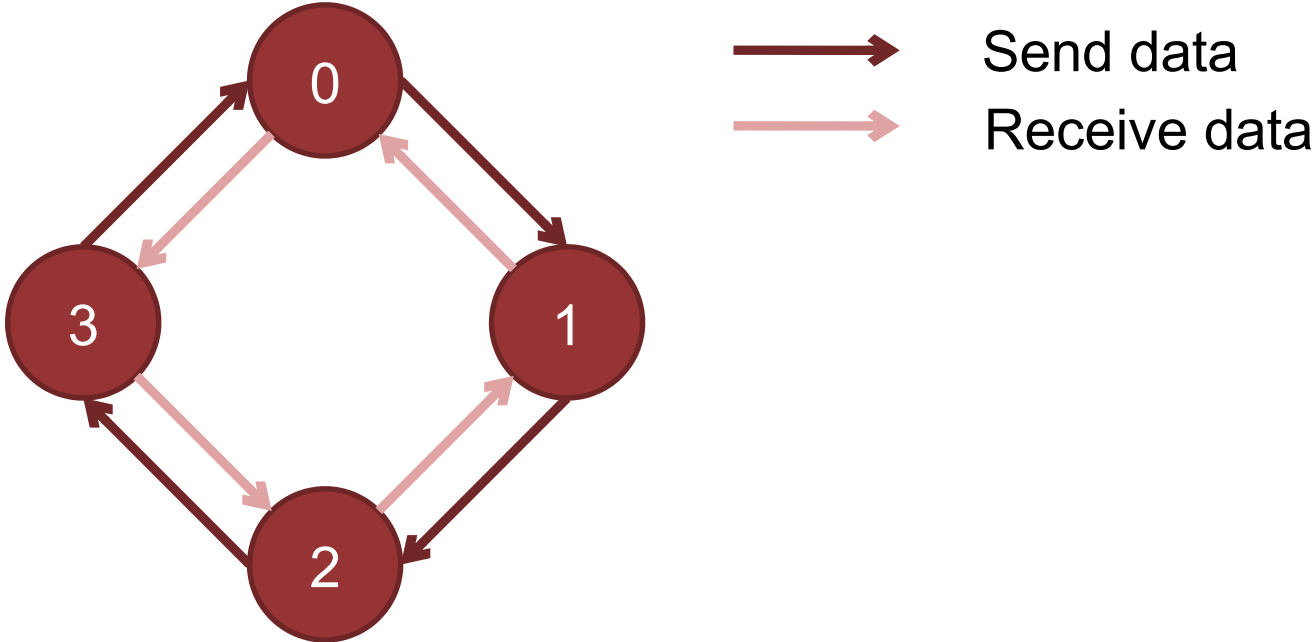
## Data types in MPI

- ▶ MPI defines its own data types
  - ▶ Allows portability of MPI programs among different architectures
  - ▶ Allows definition of new data types
    - ▶ *Derived data types*
- ▶ Most predefined data types in MPI correspond to predefined data types of the language, e.g. in C:
  - ▶ MPI\_SHORT . short int
  - ▶ MPI\_INT . int
  - ▶ MPI\_LONG . long int
  - ▶ MPI\_FLOAT . float
  - ▶ MPI\_DOUBLE . double
  - ▶ MPI\_CHAR . char
  - ▶ ...



# Creating a virtual ring (1/3)

- ▶ Each MPI process will send its rank to the “next” process
  - ▶ What happens with the last process (with the largest rank)?





## Creating a virtual ring (2/3)

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

int main(int argc, char *argv[]) {
    int my_rank, p, next_rank, prev_rank, other_rank, tag1 = 5;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank == p - 1) {
        next_rank = 0;
    } else {
        next_rank = my_rank + 1;
    }
}
```



## Creating a virtual ring (3/3)

```
if (my_rank == 0) {
    prev_rank = p - 1;
} else {
    prev_rank = my_rank - 1;
}

MPI_Send(&my_rank, 1, MPI_INT, next_rank, tag1, MPI_COMM_WORLD);
MPI_Recv(&other_rank, 1, MPI_INT, prev_rank, tag1, MPI_COMM_WORLD, &status);

printf("I am process %d and I received the value %d.\n", my_rank, other_rank);

MPI_Finalize();

return(0);
}
```

What would happen if we changed the order of the send and receive calls?



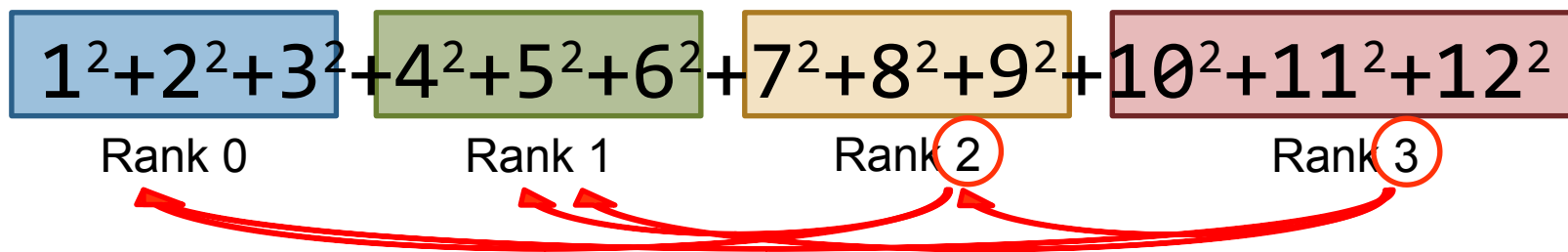
## Important notice

- ▶ Most of the examples we will look at are “embarrassingly parallel”
  - ▶ No exchange of data is required during execution
  - ▶ Except at the beginning and end of the calculation
- ▶ In this training we focus on making things clear with respect to using MPI
  - ▶ The simplicity of these examples helps towards this goal
- ▶ Real problems are typically not embarrassingly parallel

## Calculation of $1^2+2^2+\dots+N^2$

- ▶ Ask the user to provide a number N
  - ▶ Calculate in parallel  $1^2+2^2+\dots+N^2$

Assume: N = 12  
 p = 4  
 num = 3



How many elements for each process?  
 num = N / p = 3

How will each process calculate the first number it has to process?



## Calculation of $1^2+2^2+\dots+N^2$ (1/3)

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

int main(int argc, char *argv[]) {
    int my_rank, p, i, res, finres, start, end, num, N, source, target, tag1 = 5, tag2 = 6;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank == 0) {
        printf("Enter last number: ");
        scanf("%d", &N);
        for (target = 1; target < p; target++) {
            MPI_Send(&N, 1, MPI_INT, target, tag1, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&N, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status);
    }
}
```



## Calculation of $1^2+2^2+\dots+N^2$ (2/3)

```
res    = 0;
num    = N / p;
start  = (my_rank * num) + 1;
end    = start + num;

for (i = start; i < end; i++) {
    res += (i * i);
}
```



## Calculation of $1^2+2^2+\dots+N^2$ (3/3)

```
if (my_rank != 0) {
MPI_Send(&res, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
} else {
finres = res;

printf("\nResult of process %d: %d\n", my_rank, res);

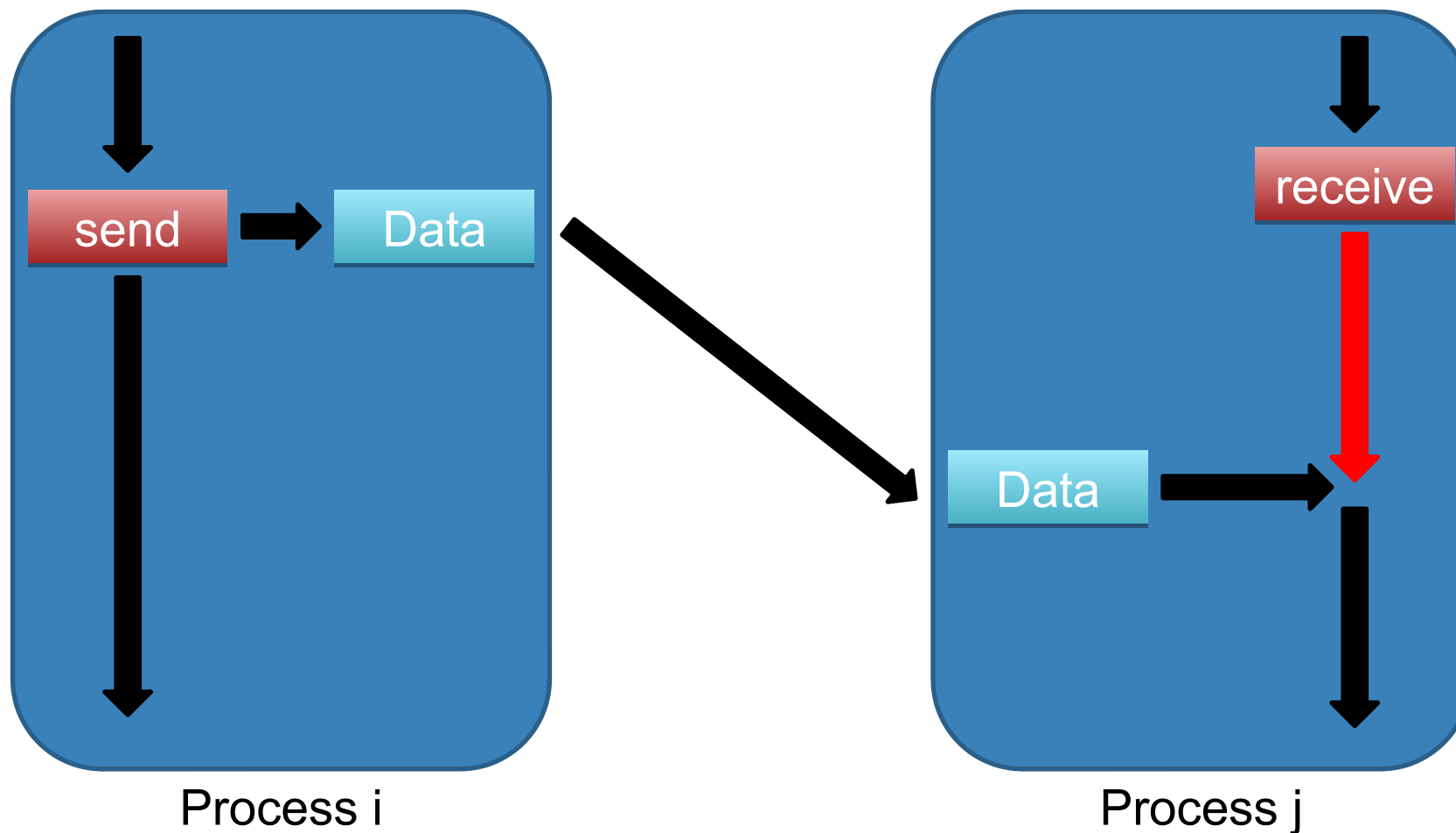
for (source = 1; source < p; source++) {
MPI_Recv(&res, 1, MPI_INT, source, tag2, MPI_COMM_WORLD, &status);
finres += res;
printf("\nResult of process %d: %d\n", source, res);
}
printf("\n\nFinal result: %d\n", finres);
}

MPI_Finalize();

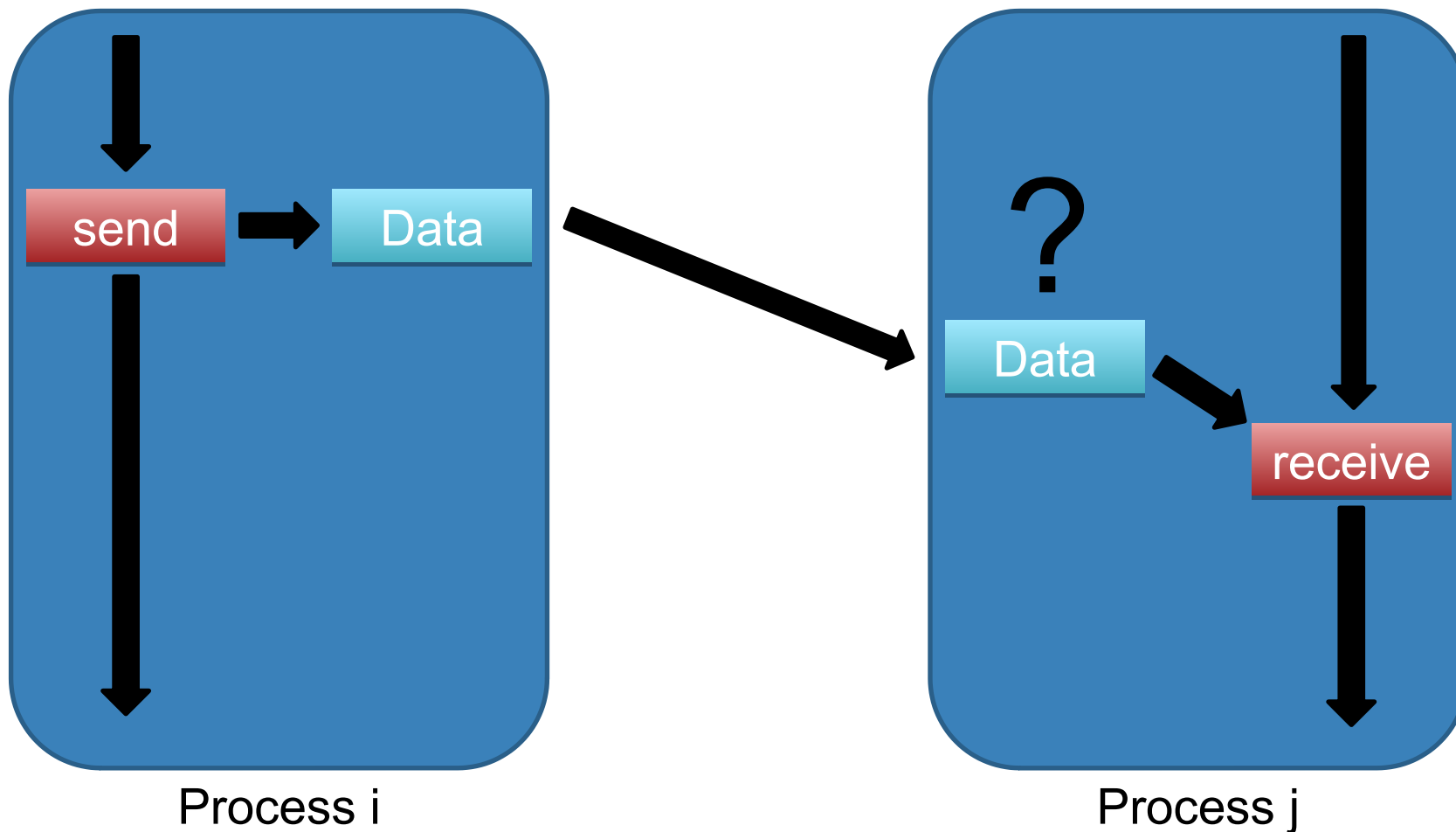
return(0);
}
```



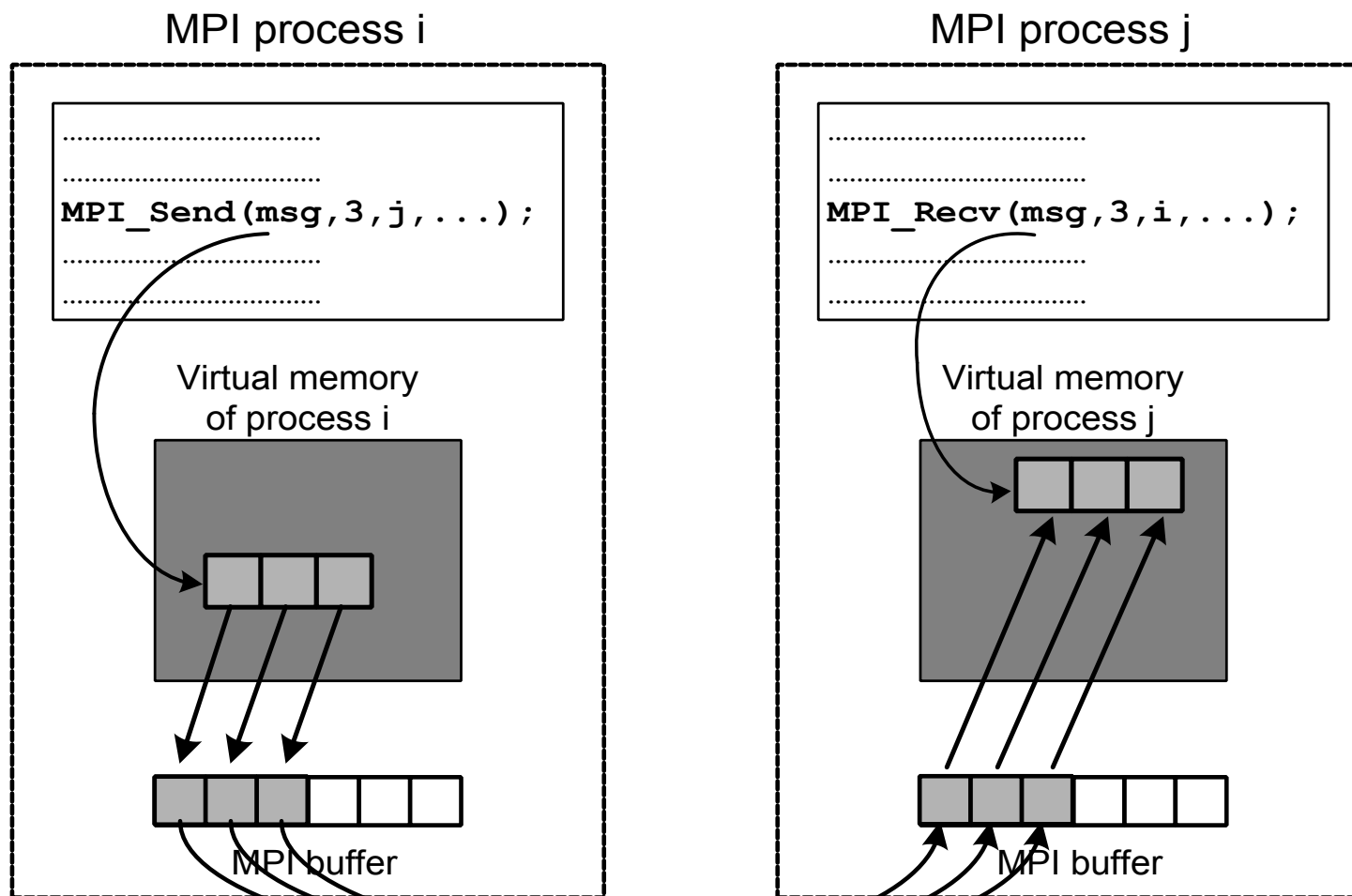
Relative execution speed:  
Receive is executed before corresponding send



Relative execution speed:  
Send is executed before corresponding receive



# Send and receive in MPI





## Sum of squares of vector elements (1/3)

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

int main(int argc, char *argv[])
{
    int my_rank, p, i, res, finres, num, N;
    int source, target;
    int tag1=50, tag2=60, tag3=70;
    int data[100];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```



## Sum of squares of vector elements (2/3)

```
if (my_rank == 0) {
    printf("Enter size of vector: ");
    scanf("%d", &N);

    printf("Enter %d vector elements: ", N);
    for (i = 0; i < N; i++) {
        scanf("%d", &data[i]);
    }

    for (target = 1; target < p; target++) {
        MPI_Send(&N, 1, MPI_INT, target, tag1, MPI_COMM_WORLD);
    }

    num = N / p;
    i = num;
    for (target = 1; target < p; target++) {
        MPI_Send(&data[i], num, MPI_INT, target, tag2, MPI_COMM_WORLD);
        i += num;
    }
} else {
    MPI_Recv(&N, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status);
    num = N / p;
    MPI_Recv(&data[0], num, MPI_INT, 0, tag2, MPI_COMM_WORLD, &status);
}
```



## Sum of squares of vector elements (3/3)

```
res = 0;
for (i = 0; i < num; i++) {
    res += (data[i] * data[i]);
}

if (my_rank != 0) {
    MPI_Send(&res, 1, MPI_INT, 0, tag3, MPI_COMM_WORLD);
} else {
    finres = res;

    printf("\nResult of process %d: %d\n", my_rank, res);

    for (source = 1; source < p; source++) {
        MPI_Recv(&res, 1, MPI_INT, source, tag3, MPI_COMM_WORLD, &status);
        finres += res;
        printf("\nResult of process %d: %d\n", source, res);
    }
    printf("\n\n\nFinal result: %d\n", finres);
}

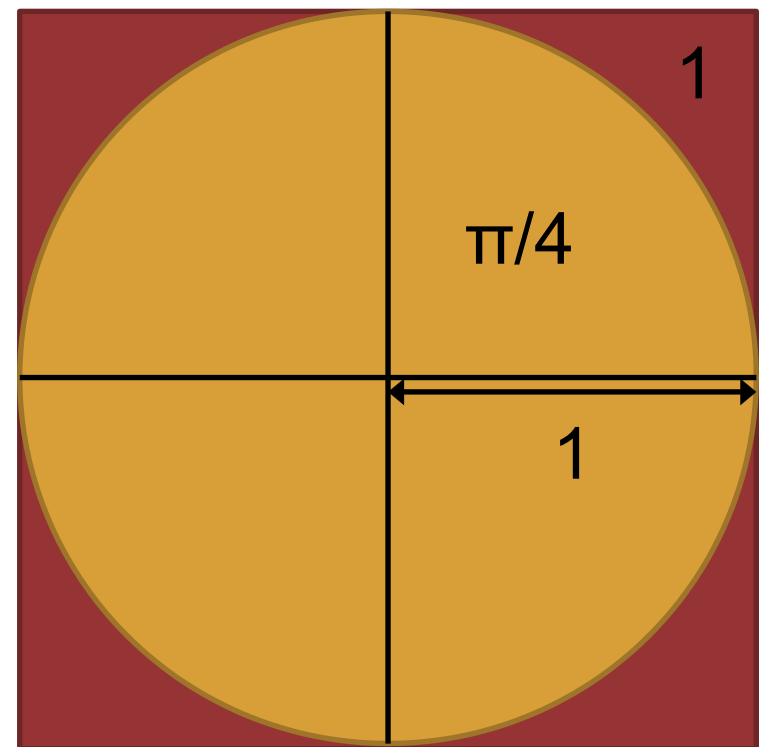
MPI_Finalize();

return(0);
}
```

## Calculating $\pi$ with a Monte Carlo method (1/3)

▶ Idea

- ▶ Create a square with a side of 1 and a quartile with a radius of 1
  - ▶ *Area of square = 1*
  - ▶ *Area of quartile =  $\pi/4$*
- ▶ Put N random points in to the square
  - ▶ *Assume that M fall into the quartile*



$$\frac{\text{Area of quartile}}{\text{Area of square}} = \frac{\pi/4}{1} \approx \frac{M}{N} \Rightarrow \pi \approx 4 \cdot \frac{M}{N}$$

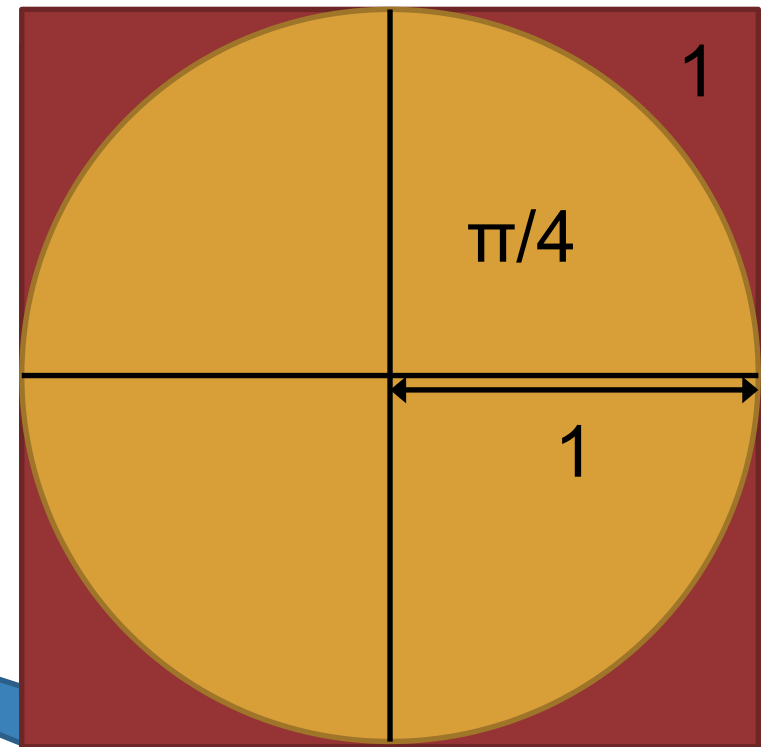
## Calculating $\pi$ with a Monte Carlo method (2/3)

```
int main(int argc, char* argv[])
{
    int i, rank, np, c, local_c, local_N, N=1000000;
    int tag = 5, source;
    double x, y;
    MPI_status status;

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

    srand48(rank);
    local_c = 0;
    local_N = N / np;

    for (i = 0; i < local_N; i++) {
        x = drand48();
        y = drand48();
        if (((x * x) + (y * y)) <= 1.0) {
            local_c++;
        }
    }
}
```



Avoids creation of the same pseudo-random series of numbers on all MPI processes





## Calculating $\pi$ with a Monte Carlo method (3/3)

```
if (rank != 0) {
MPI_Send(&local_c, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
} else {
c = local_c;

for (source = 1; source < np; source++) {
MPI_Recv(&local_c,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
c += local_c;
}
printf("Estimate of Pi = %24.16f\n", (4.0 * c) / N);
}

MPI_Finalize();

return(0);
}
```



# Collective communication

---



## Collective communication

- ▶ Allows exchange of data among processes of an MPI program that belong to the same communicator
  - ▶ Broadcast
  - ▶ Reduction
  - ▶ Gather/Gatherv, Scatter/Scatterv, AllGather/AllGatherv
  - ▶ All-to-All
  - ▶ Barrier
- ▶ Support for **topologies**
  - ▶ More on this later during the presentation
- ▶ More efficient use of MPI buffers
  - ▶ Optimized use for multiple operations
- ▶ Support for MPI data types

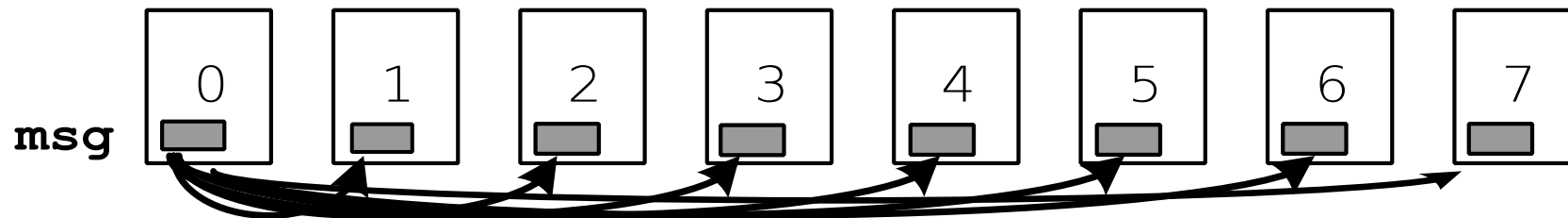
## Why support and use collective communication operations? (1/2)

- ▶ Sending the same message from process 0 to processes 1-7

```

if (rank == 0) {
for (dest = 1; dest < size; dest++)
MPI_Send(msg, count, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
} else {
MPI_Recv(msg, count, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status);
}
    
```

MPI Processes

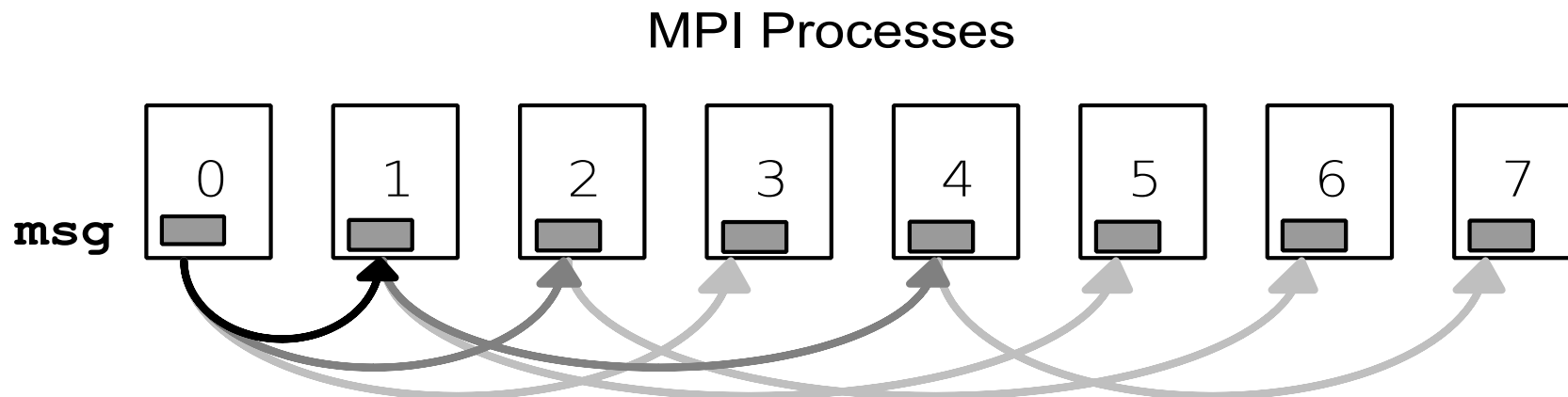


- ▶ For  $p$  processes we need  $p-1$  communication steps

## Why support and use collective communication operations? (2/2)

- ▶ Sending the same message from process 0 to processes 1-7

```
MPI_Bcast(msg, count, MPI_FLOAT, 0, MPI_COMM_WORLD);
```



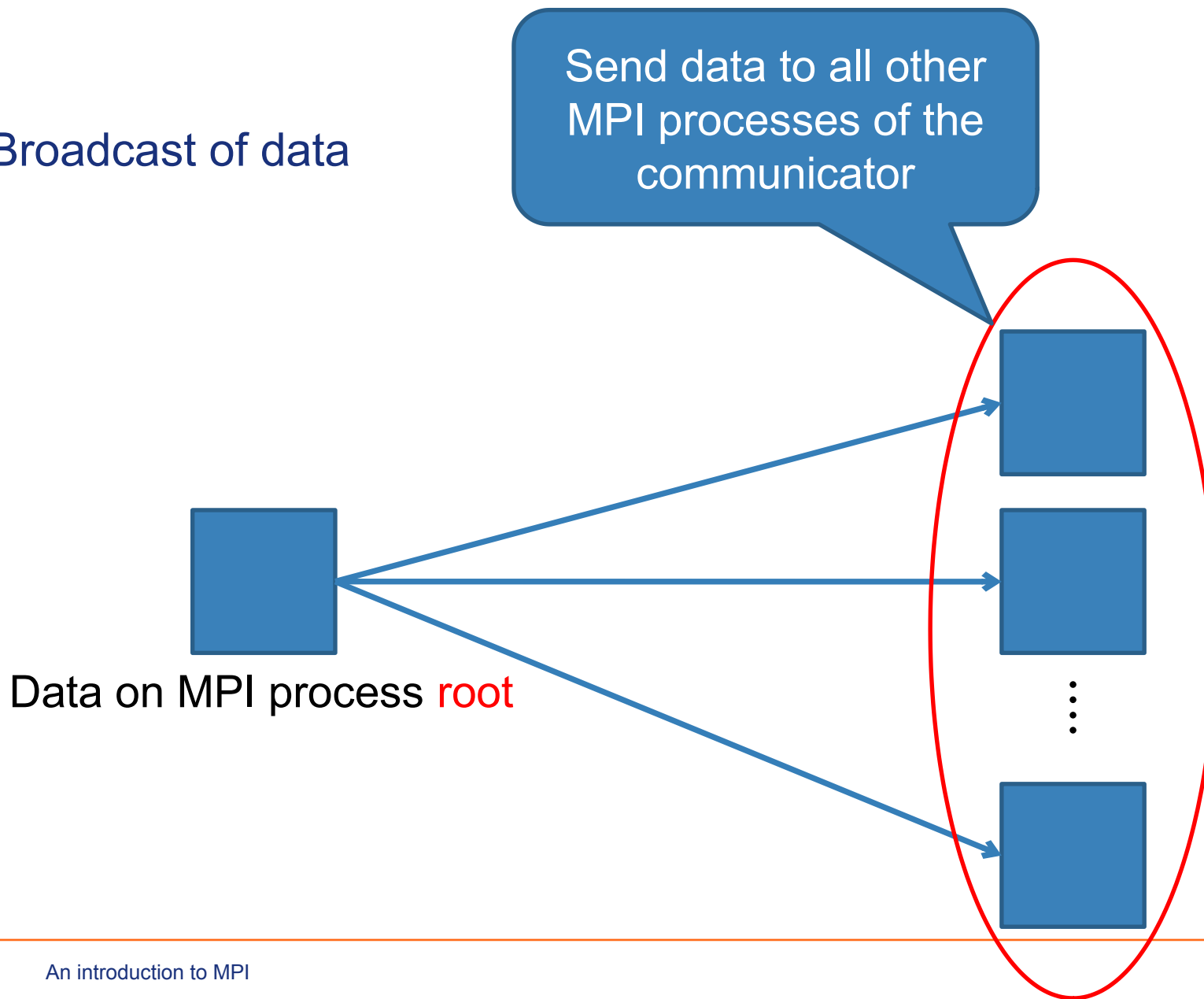
- ▶ For  $p$  processes we need  $\lceil \log_2 p \rceil$  communication steps



## Broadcast

- ▶ `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
  - ▶ buffer: Has double semantics
    - ▶ *Process that sends data: Starting address of data to be sent*
    - ▶ *Processes that receive data: Starting address in memory where received data will be stored*
  - ▶ count: Number of elements to be sent/received
  - ▶ datatype: Data type of each element to be sent/received
  - ▶ root: Rank of process that is sending data
    - ▶ *All other processes in the communicator will receive data*
  - ▶ comm: Communicator

## Broadcast of data





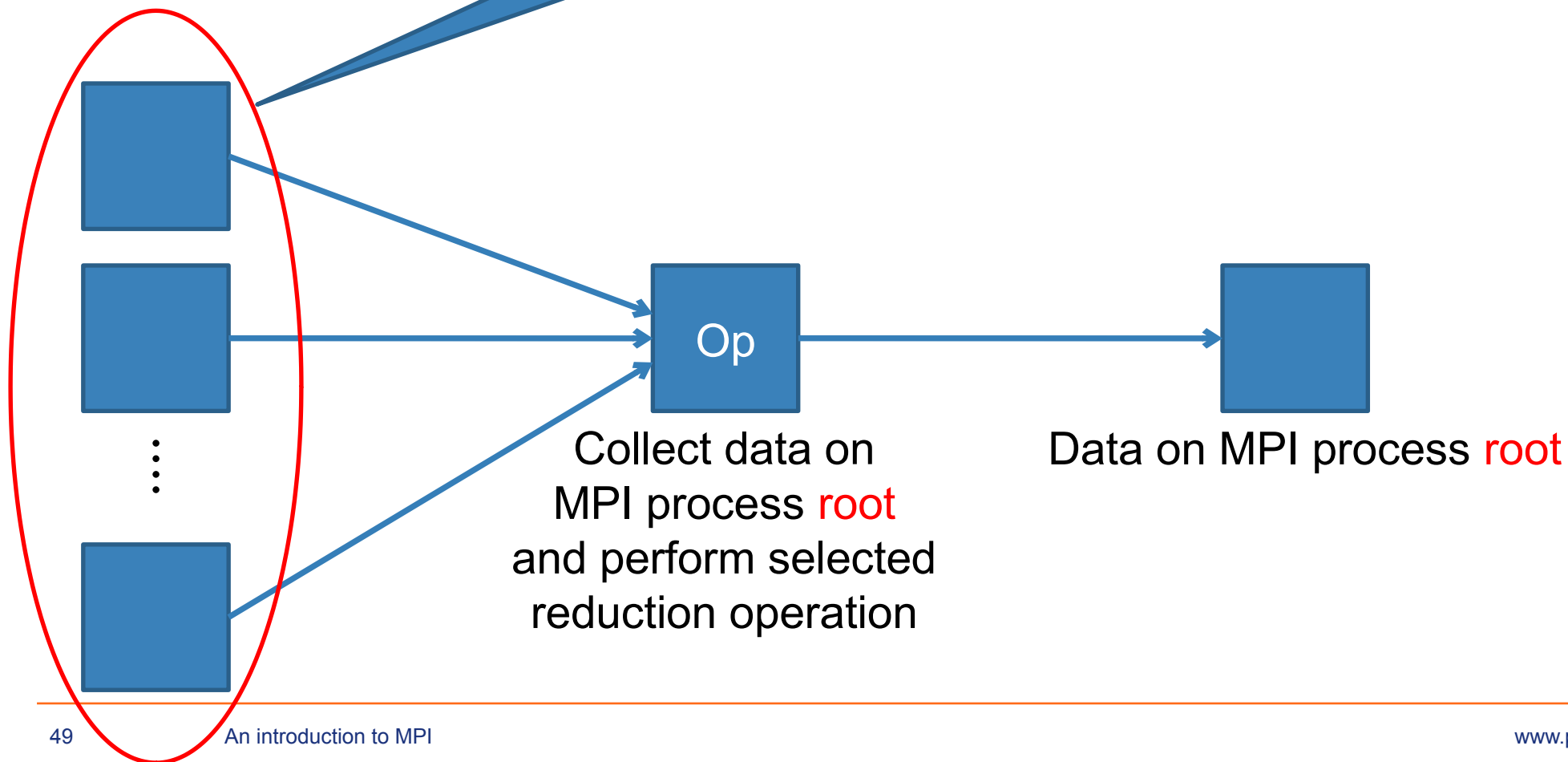
## Reduction

- ▶ `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);`
  - ▶ `sendbuf`: Starting address of data to be sent
  - ▶ `recvbuf`: Starting address in memory where received data will be stored
    - ▶ *Data stored only on the process with rank `root`*
  - ▶ `count`: Number of elements to be sent/received
  - ▶ `datatype`: Data type of each element to be sent/received
  - ▶ `op`: Reduction operation to be performed
  - ▶ `root`: The rank of the process that receives data and performs the reduction operation
    - ▶ *All processes in the communicator send data (including `root`)*
  - ▶ `comm`: Communicator



## Reduction of data

Send data from all MPI processes of the communicator (including process **root**) to process **root**





## Reduction operations

- ▶ `MPI_MAX` Maximum value
- ▶ `MPI_MIN` Minimum value
- ▶ `MPI_SUM` Sum
- ▶ `MPI_PROD` Product
- ▶ `MPI_LAND` Logical AND
- ▶ `MPI_BAND` Bitwise AND
- ▶ `MPI_LOR` Logical OR
- ▶ `MPI BOR` Bitwise OR
- ▶ `MPI_LXOR` Logical Exclusive OR (XOR)
- ▶ `MPI_BXOR` Bitwise Exclusive OR (XOR)
- ▶ `MPI_MAXLOC` Maximum value and position within a vector
- ▶ `MPI_MINLOC` Minimum value and position within a vector



## Calculation of $1^2+2^2+\dots+N^2$ (1/2)

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

int main(int argc, char *argv[])
{
    int my_rank, p, i, res, finres, start, end, num, N;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank == 0) {
        printf("Enter last number: ");
        scanf("%d", &N);
    }
}
```



## Calculation of $1^2+2^2+\dots+N^2$ (2/2)

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

res    = 0;
num    = N / p;
start  = (my_rank * num) + 1;
end    = start + num;
for (i = start; i < end; i++) {
    res += (i * i);
}

printf("\nResult of process %d: %d\n", my_rank, res);

MPI_Reduce(&res, &finres, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (my_rank == 0) {
    printf("\n Total result for N = %d is equal to : %d \n", N, finres);
}
MPI_Finalize();
return(0);
}
```



## Scatter

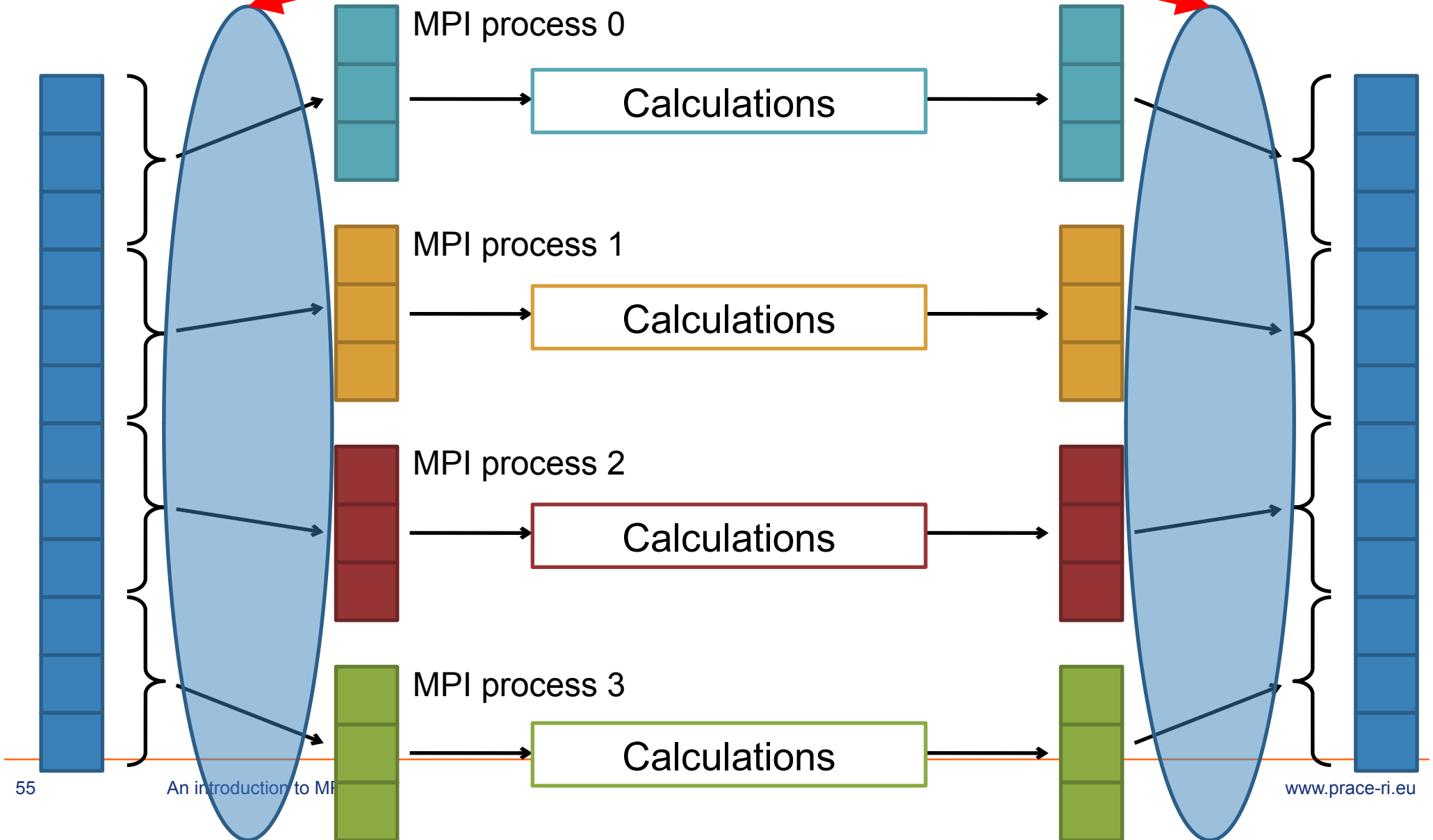
- ▶ `int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtpe, int root, MPI_Comm comm);`
- ▶
  - ▶ `sendbuf`: Starting address of data to be sent
    - ▶ *Only on the process with rank `root`*
  - ▶ `sendcount`: Number of elements to be sent **to each process**
  - ▶ `sendtype`: Data type of each element to be sent
  - ▶ `recvbuf`: Starting address in memory where received data will be stored
    - ▶ *On all processes, including `root`*
  - ▶ `recvcount`: Number of elements to be received **on each process**
  - ▶ `recvtpe`: Data type of each element to be received
  - ▶ `root`: The rank of the process that will distribute the data
    - ▶ *All other processes in the communicator will receive part of the data*
  - ▶ `comm`: Communicator



## Gather

- ▶ `int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`
  - ▶ `sendbuf`: Starting address of data to be collected
    - ▶ *From every process, including `root`*
  - ▶ `sendcount`: Number of elements to be sent **from each process**
  - ▶ `sendtype`: Data type of each element to be sent
  - ▶ `recvbuf`: Starting address in memory where received data will be stored
    - ▶ *Only on the process with rank `root`*
  - ▶ `recvcount`: Number of elements to be received **from each process**
  - ▶ `recvtype`: Data type of each element to be received
  - ▶ `root`: The rank of the process that will collect all the data
    - ▶ *All other processes in the communicator will send part of the data*
  - ▶ `comm`: Communicator

# Scatter/Gather





## Scalar-Vector multiplication (1/3)

```
int main(int argc, char *argv[])
{
    int my_rank, p, i, num, b, size, A[100], local_A[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank == 0) {
        printf("Calculating b * A\n\n");
        printf("Enter value for b: ");
        scanf("%d", &b);
        printf("Enter size of vector A:");
        scanf("%d", &size);
        printf("Enter values of vector elements: ");
        for (i = 0; i < size; i++) {
            scanf("%d", &A[i]);
        }
    }
}
```





## Scalar-Vector multiplication (2/3)

```
MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&b, 1, MPI_INT, 0, MPI_COMM_WORLD);

num = size / p;

MPI_Scatter(A, num, MPI_INT, local_A, num, MPI_INT, 0, MPI_COMM_WORLD);

for (i = 0; i < num; i++) {
    local_A[i] *= b;
}

printf("\nLocal results for process %d:\n", my_rank);
for (i = 0; i < num; i++) {
    printf("%d ", local_A[i]);
}
printf("\n\n");
```



## Scalar-Vector multiplication (3/3)

```
MPI_Gather(local_A, num, MPI_INT, A, num, MPI_INT, 0, MPI_COMM_WORLD);

if (my_rank == 0) {
    printf("\nFinal result:\n");
    for (i = 0; i < size; i++) {
        printf("%d ", A[i]);
    }
    printf("\n\n");
}

MPI_Finalize();

return(0);
}
```



## Scatterv

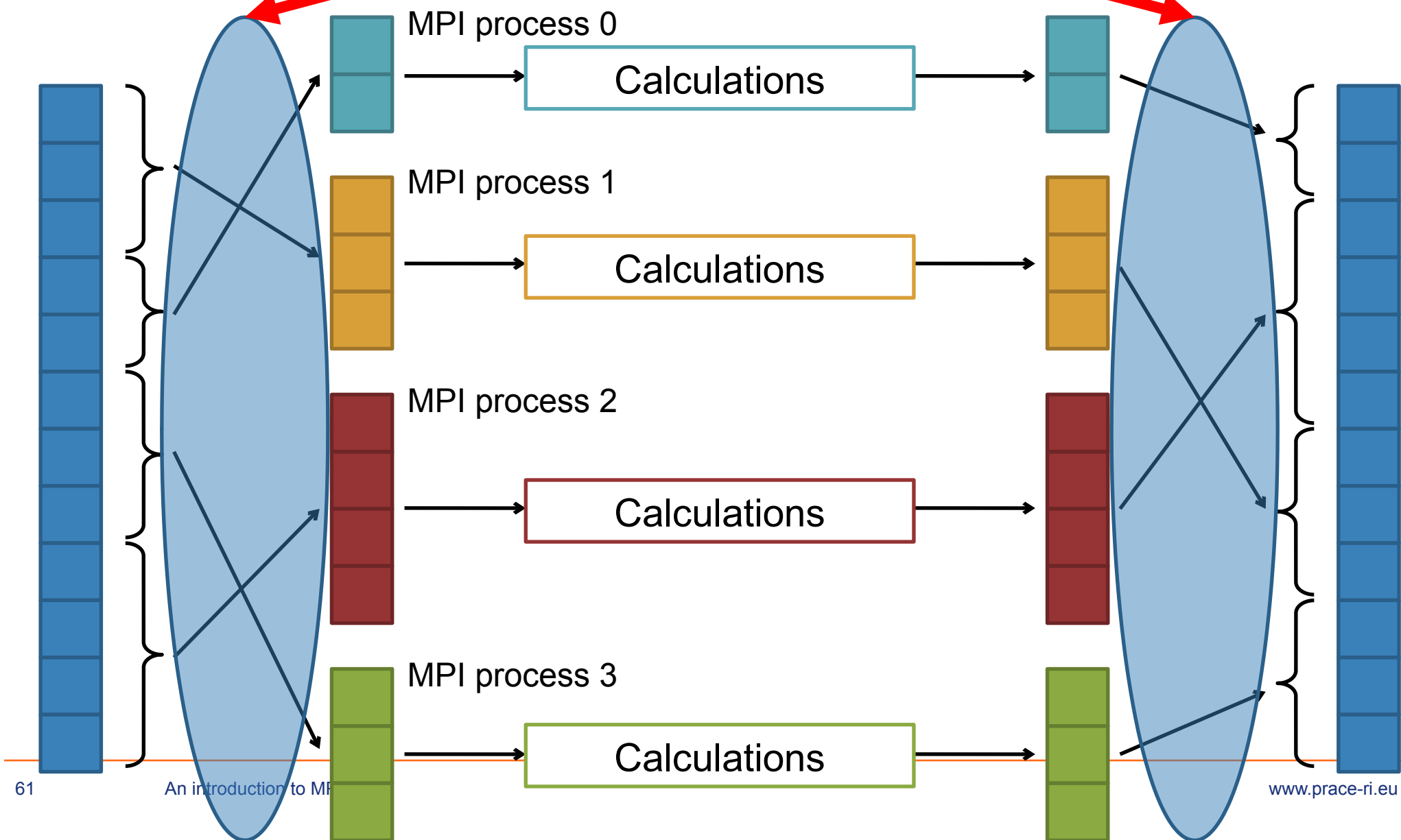
- ▶ `int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtpe, int root, MPI_Comm comm);`
  - ▶ “sendbuf”: Η διεύθυνση των δεδομένων προς διαμοίραση
    - ▶ *Στον επεξεργαστή root*
  - ▶ “sendcounts”: Διάνυσμα με το πλήθος στοιχείων που αποστέλλονται προς κάθε επεξεργαστή
  - ▶ “displs”: Διάνυσμα με τις μετατοπίσεις από την αρχή του διανύσματος “sendbuf” για κάθε επεξεργαστή
  - ▶ “sendtype”: Ο τύπος κάθε στοιχείου που αποστέλλεται
  - ▶ “recvbuf”: Η διεύθυνση αποθήκευσης των δεδομένων που θα παραληφθούν
    - ▶ *Σε κάθε επεξεργαστή*
  - ▶ “recvcnt”: Πλήθος στοιχείων που παραλαμβάνονται από κάθε επεξεργαστή
  - ▶ “recvtpe”: Ο τύπος κάθε στοιχείου που παραλαμβάνεται
  - ▶ “root”: Ο επεξεργαστής ο οποίος αποστέλλει δεδομένα
    - ▶ *Όλοι οι άλλοι στον communicator θα παραλάβουν*
  - ▶ “comm”: Communicator



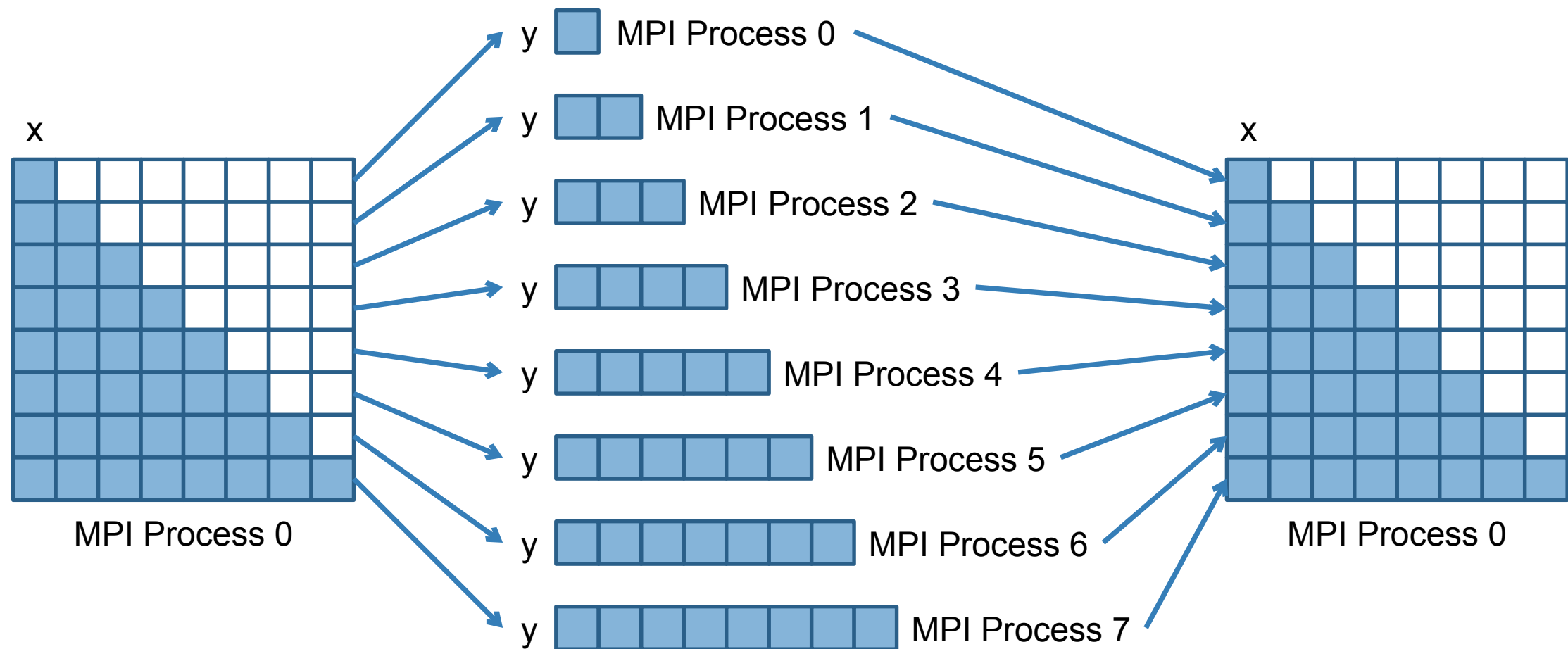
## Gatherv

- ▶ `int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);`
  - ▶ “sendbuf”: Η διεύθυνση των δεδομένων προς συγκέντρωση
    - ▶ Σε κάθε επεξεργαστή
  - ▶ “sendcount”: Πλήθος στοιχείων που αποστέλλονται προς τον επεξεργαστή root από κάθε επεξεργαστή
  - ▶ “sendtype”: Ο τύπος κάθε στοιχείου που αποστέλλεται
  - ▶ “recvbuf”: Η διεύθυνση αποθήκευσης των δεδομένων που θα παραληφθούν
    - ▶ Στον επεξεργαστή root
  - ▶ “recvcounts”: Διάνυσμα με το πλήθος στοιχείων που παραλαμβάνονται από κάθε επεξεργαστή
  - ▶ “displs”: Διάνυσμα με τις μετατοπίσεις από την αρχή του διανύσματος “recvbuf” για κάθε επεξεργαστή
  - ▶ “recvtype”: Ο τύπος κάθε στοιχείου που παραλαμβάνεται
  - ▶ “root”: Ο επεξεργαστής ο οποίος συγκεντρώνει δεδομένα
    - ▶ Όλοι οι άλλοι στον *communicator* θα αποστείλουν
  - ▶ “comm”: Communicator

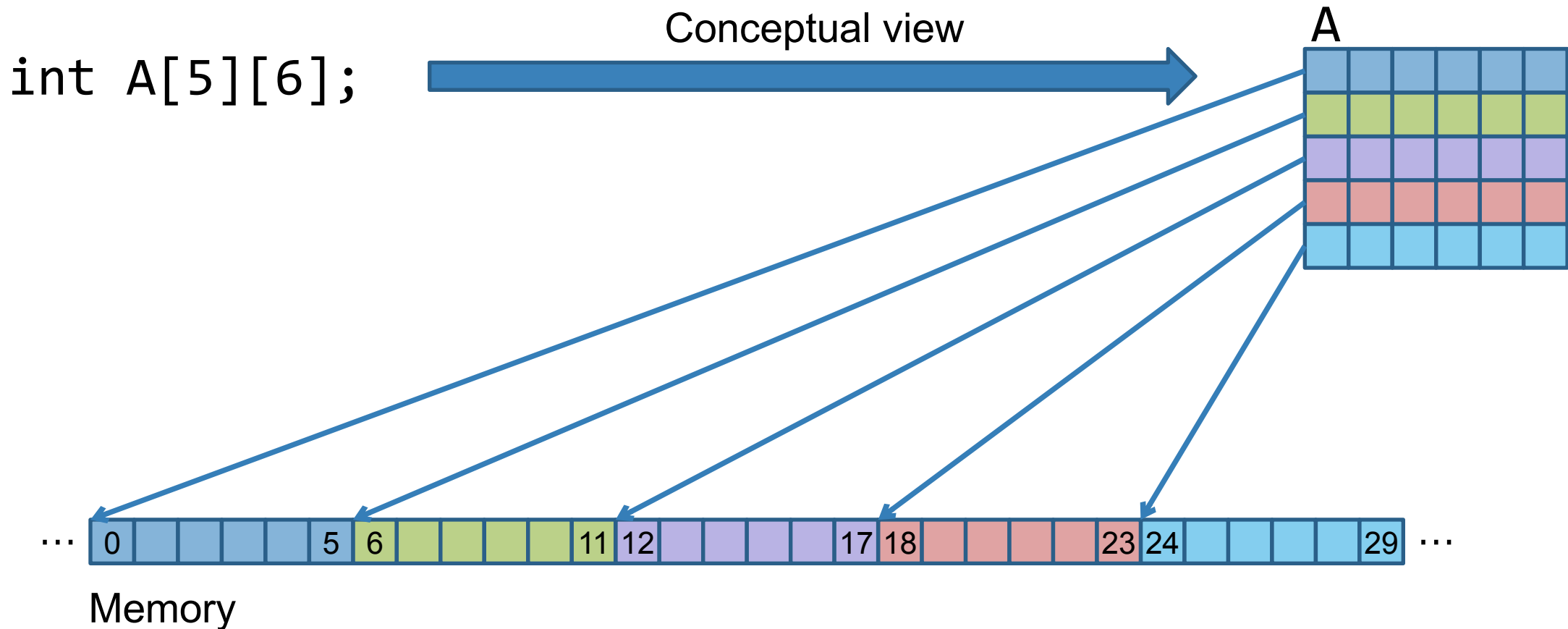
# Scatterv/Gatherv



## Example: Handling a lower triangular matrix



## 2-D matrix representation in memory (row-major)





## Handling a lower triangular matrix (1/7)

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

#define MAXPROC 8          /* Max number of procsses */
#define LENGTH 8          /* Size of matrix is LENGTH * LENGTH */

int main(int argc, char *argv[]) {
    int i, j, np, my_rank;
    int x[LENGTH][LENGTH]; /* Send buffer */
    int y[LENGTH]; /* Receive buffer */
    int res[LENGTH][LENGTH]; /* Final receive buffer */
    int *sendcount, *recvcount; /* Arrays for sendcounts and recvcounts */
    int *displs1, *displs2; /* Arrays for displacements */
```





## Handling a lower triangular matrix (2/7)

```
MPI_Init(&argc, &argv);          /* Initialize MPI      */
MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get # of processes */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /* Get own rank */

/* Check that we have one process for each row in the matrix */
if (np != LENGTH) {
    if (my_rank == 0) {
        printf("You have to use %d processes\n", LENGTH);
    }
    MPI_Finalize();
    exit(0);
}
```



## Handling a lower triangular matrix (3/7)

```
/* Allocate memory for the sendcount, recvcount and displacements arrays */
sendcount = (int *)malloc(LENGTH*sizeof(int));
recvcount = (int *)malloc(LENGTH*sizeof(int));
displs1    = (int *)malloc(LENGTH*sizeof(int));
displs2    = (int *)malloc(LENGTH*sizeof(int));

if (my_rank == 0) {
    for (i = 0; i < LENGTH; i++) {
        for (j = 0; j < LENGTH; j++) {
            x[i][j] = i * LENGTH + j;
        }
    }
}
```



## Handling a lower triangular matrix (4/7)

```
/* Initialize sendcount and displacements arrays */
for (i = 0; i < LENGTH; i++) {
    sendcount[i] = i + 1;
    displs1[i] = i * LENGTH;
}

/* Scatter the lower triangular part of array x to all proceses, place it in y */
MPI_Scatterv(x, sendcount, displs1, MPI_INT, y, sendcount[my_rank], MPI_INT, 0,
            MPI_COMM_WORLD);
```



## Handling a lower triangular matrix (5/7)

```
if (my_rank == 0) {
    /* Initialize the result matrix res with 0 */
    for (i = 0; i < LENGTH; i++) {
        for (j = 0; j < LENGTH; j++) {
            res[i][j] = 0;
        }
    }

    /* Print out the result matrix res before gathering */
    printf("The result matrix before gathering is\n");
    for (i = 0; i < LENGTH; i++) {
        for (j = 0; j < LENGTH; j++) {
            printf("%4d ", res[i][j]);
        }
        printf("\n");
    }
}
```



## Handling a lower triangular matrix (6/7)

```
for (i = 0; i < LENGTH; i++) {  
    recvcount[i] = i + 1;  
    displs2[i] = i * LENGTH;  
}
```

```
/* Gather the local elements of each process to form a triangular matrix in the root */  
MPI_Gatherv(y, recvcount[my_rank], MPI_INT, res, recvcount, displs2, MPI_INT, 0,  
            MPI_COMM_WORLD);
```



## Handling a lower triangular matrix (7/7)

```
if (my_rank == 0) {
    /* Print out the result matrix after gathering */
    printf("The result matrix after gathering is\n");
    for (i = 0; i < LENGTH; i++) {
        for (j = 0; j < LENGTH; j++) {
            printf("%4d ", res[i][j]);
        }
        printf("\n");
    }
}

MPI_Finalize();

exit(0);
}
```

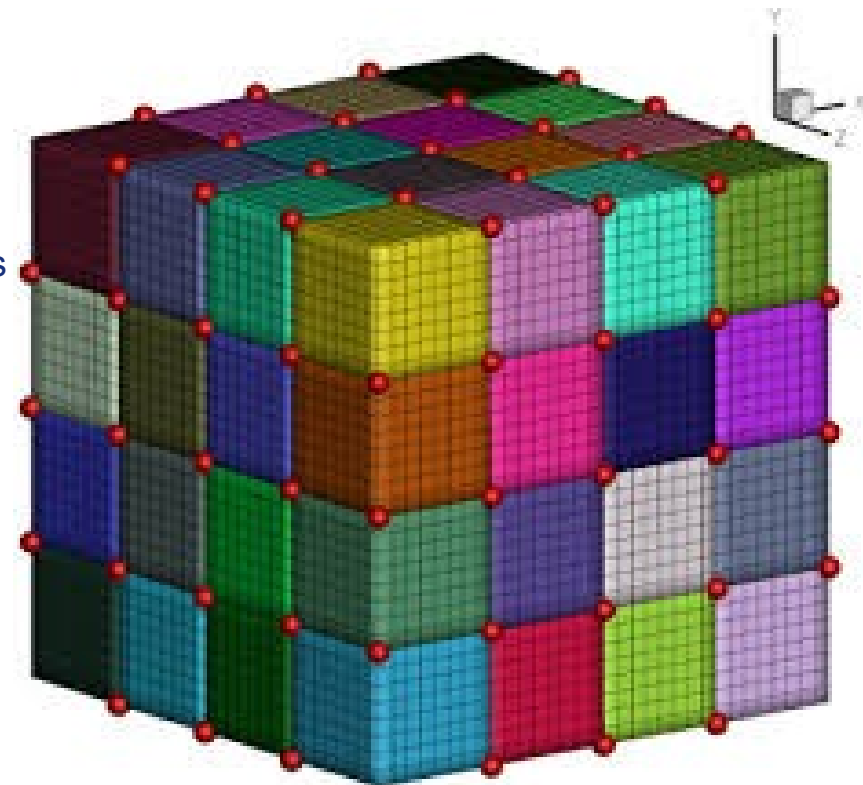


# Virtual Topologies

---

# Virtual Topologies

- ▶ Physical problems are in 1D, 2D or 3D
  - ▶ Domain is decomposed into smaller parts
  - ▶ Each part is assigned to an MPI process
- ▶ Algorithms that solve such problems typically require MPI processes to exchange data that resides close to the borders of each part
- ▶ Virtual topologies
  - ▶ Allows easy determination of neighboring MPI processes that must exchange data
  - ▶ Two types supported
    - ▶ *Cartesian (regular mesh)*
    - ▶ *Graph*







## Creating a communication topology

- ▶ `int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)`
  - ▶ “comm\_old”: The original communicator that includes the MPI processes to be used in creating the Cartesian topology
  - ▶ “ndims”: Dimensionality of the Cartesian topology
  - ▶ “dims”: integer array (of size “ndims”) that defines the number of processes in each dimension
  - ▶ “periods”: array (of size “ndims”) that defines the periodicity of each dimension
    - ▶ *0 for a dimension ⇒ Not periodic*
    - ▶ *1 for a dimension ⇒ Periodic*
  - ▶ “reorder”: MPI is allowed to renumber (reassign ranks) of the MPI processes
  - ▶ “comm\_cart”: New communicator that includes information about the Cartesian topology



## Ranks and coordinates

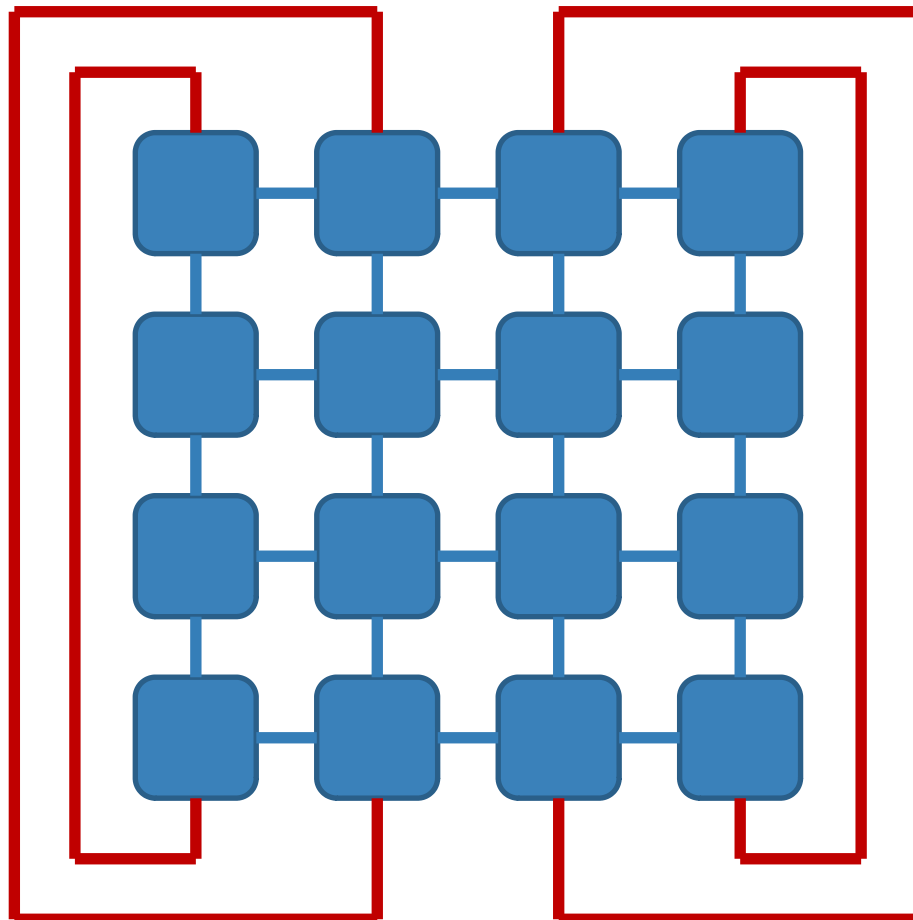
- ▶ Translate a rank to coordinates within a Cartesian topology
- ▶ `int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)`
  - ▶ “comm”: Cartesian communicator
  - ▶ “rank”: Rank of MPI process whose coordinates in the Cartesian topology are to be found
  - ▶ “maxdim”: Dimension of “coords”
  - ▶ “coords”: Coordinates in the Cartesian topology that correspond to “rank”
- ▶ Translate coordinates of a Cartesian topology to a rank
- ▶ `int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)`
  - ▶ “comm”: Cartesian communicator
  - ▶ “coords”: Array of coordinates
  - ▶ “rank”: The rank of the MPI processes corresponding to “coords”



## Finding neighbors

- ▶ Find ranks of neighboring MPI processes on a single dimension
- ▶ `int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)`
  - ▶ “comm”: Cartesian communicator
  - ▶ “direction”: Shift direction, e.g., dimension for which neighbors will be returned
    - ▶ *0 on 1D topology*
    - ▶ *0 or 1 on 2D topology*
    - ▶ *0, 1 or 2 for 3D dimension, etc*
  - ▶ “disp”: Shift displacement
  - ▶ “rank\_source”: Rank of source MPI process
  - ▶ “rank\_dest”: Rank of destination process

## 2D grid with periodicity on one dimension





## 2D grid with periodicity on one dimension (1/3)

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

int main(int argc, char *argv[])
{
    int numtasks, rank, source, dest, outbuf, i, tag=1, dims[2]={4,4}, periods[2]={0,1}, nbrs[4],
        inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},reorder=0,coords[2];

    MPI_Request reqs[8];
    MPI_Status stats[8];
    MPI_Comm cartcomm;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```



## 2D grid with periodicity on one dimension (2/3)

```
if (numtasks == SIZE) {  
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);  
  
    MPI_Comm_rank(cartcomm, &rank);  
  
    MPI_Cart_coords(cartcomm, rank, 2, coords);  
  
    MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);  
    MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);  
}
```



## 2D grid with periodicity on one dimension (3/3)

```
outbuf = rank;

for (i = 0; i < 4; i++) {
    dest = nbrs[i];
    source = nbrs[i];
    MPI_Isend(&outbuf, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &reqs[i]);
    MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag, MPI_COMM_WORLD, &reqs[i+4]);
}
MPI_Waitall(8, reqs, stats);

printf("rank= %d, coords= (%d, %d), neighbors(u,d,l,r) = (%d %d %d %d)\n",
       rank, coords[0], coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT], nbrs[RIGHT]);

printf("rank= %d,                               inbuf(u,d,l,r) = (%d %d %d %d)\n",
       rank, inbuf[UP], inbuf[DOWN], inbuf[LEFT], inbuf[RIGHT]);
} else {
    printf("Must specify %d processors. Terminating.\n", SIZE);
}
MPI_Finalize();
}
```



# Communication modes

---





## Send modes

- ▶ To this point, we have studied non-blocking send routines using **standard mode**
  - ▶ In standard mode, the implementation determines whether buffering occurs
  
- ▶ MPI includes three other send modes that give the user explicit control over buffering.
  - ▶ Buffered
  - ▶ Synchronous
  - ▶ Ready
  
- ▶ Corresponding MPI functions
  - ▶ MPI\_Bsend
  - ▶ MPI\_Ssend
  - ▶ MPI\_Rsend



## MPI\_Bsend

- ▶ Buffered Send: allows user to explicitly create buffer space and attach buffer to send operations:
  - ▶ `MPI_BSend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)`
    - ▶ *Note: this is same as standard send arguments*
  - ▶ `MPI_Buffer_attach(void *buf, int size)`
    - ▶ *Create buffer space to be used with BSend*
  - ▶ `MPI_Buffer_detach(void *buf, int *size)`
    - ▶ *Note: call blocks until message has been safely sent*
- ▶ It is up to the user to properly manage the buffer and ensure space is available for any Bsend call



## MPI\_Ssend

- ▶ Synchronous Send
- ▶ Ensures that no buffering is used
- ▶ Couples send and receive operation – send cannot complete until matching receive is posted and message is fully copied to remote processor
- ▶ Very good for testing buffer safety of program



## MPI\_Rsend

- ▶ Ready Send
- ▶ Matching receive must be posted before send, otherwise program is incorrect
- ▶ Can be implemented to avoid handshake overhead when program is known to meet this condition
- ▶ Not very typical + dangerous



## MPI's Non-blocking Operations

- ▶ Non-blocking operations return (immediately) “request handles” that can be tested and waited on.
  - ▶ `MPI_Isend(start, count, datatype, dest, tag, comm, request)`
  - ▶ `MPI_Irecv(start, count, datatype, dest, tag, comm, request)`
  - ▶ `MPI_Wait(&request, &status)`
- ▶ One can also test without waiting:
  - ▶ `MPI_Test(&request, &flag, status)`



## Multiple Completions

- ▶ It is sometimes desirable to wait on multiple requests:
  - ▶ `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - ▶ `MPI_Waitany(count, array_of_requests, &index, &status)`
  - ▶ `MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)`
  
- ▶ There are corresponding versions of test for each of these.



**THANK YOU FOR YOUR ATTENTION**

[www.prace-ri.eu](http://www.prace-ri.eu)