HIGH PERFORMANCE COMPUTING CINECA ITALY

Parallel Programming with MPI

Part I - Introduction to Point-to-Point communications

HPC SCHOOL COSTA RICA

ALESSANDRO MARANI

a.marani@cineca.it || San José 2023

AGENDA

WHAT IS MPI?
A basic introduction

MY FIRST MPI PROGRAM
Basic concepts: task, rank, communicators, first functions

POINT-TO-POINT COMMUNICATION: SEND AND RECEIVE
The message structure, datatypes, blocking vs. non-blocking

THE DREADED DEADLOCK
And how to get easily past it

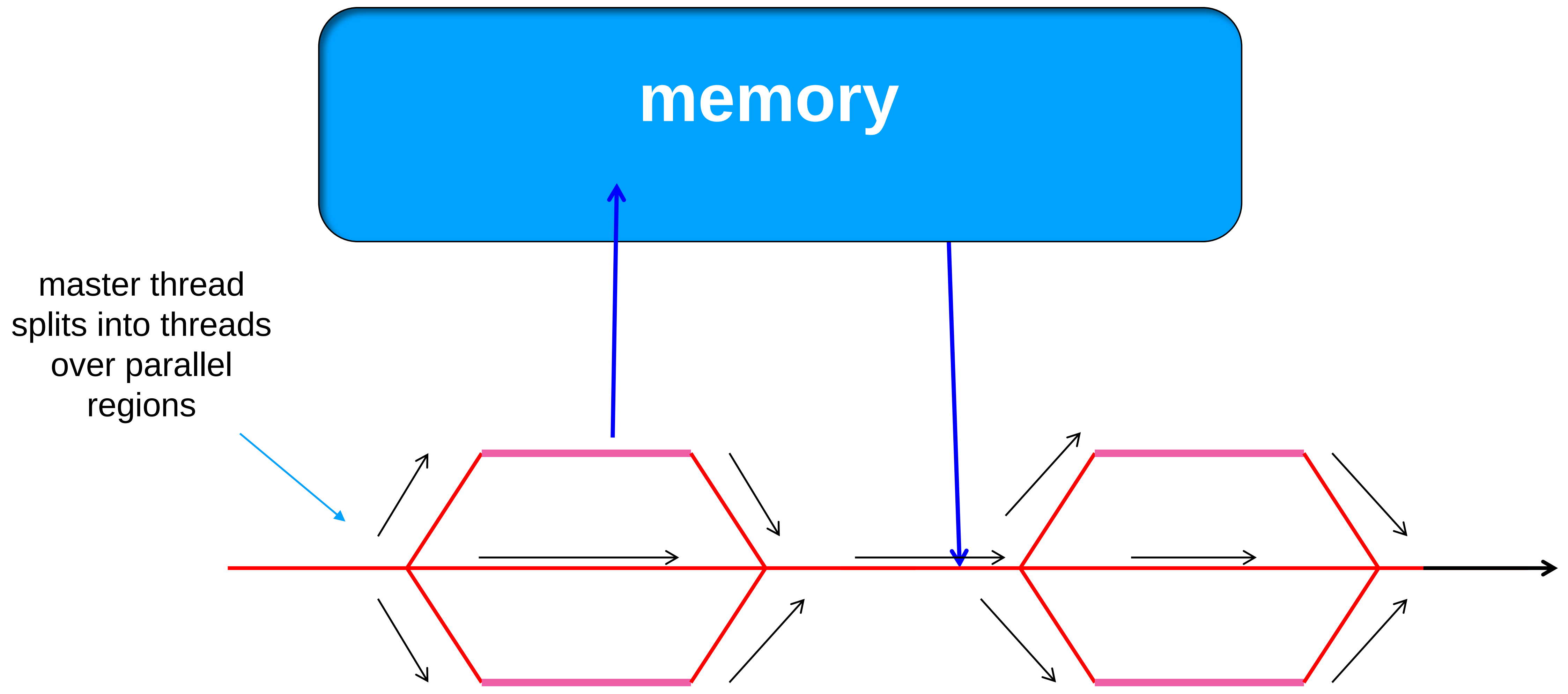
KABRÉ
SUPERCOMPUTADORA

What is MPI?

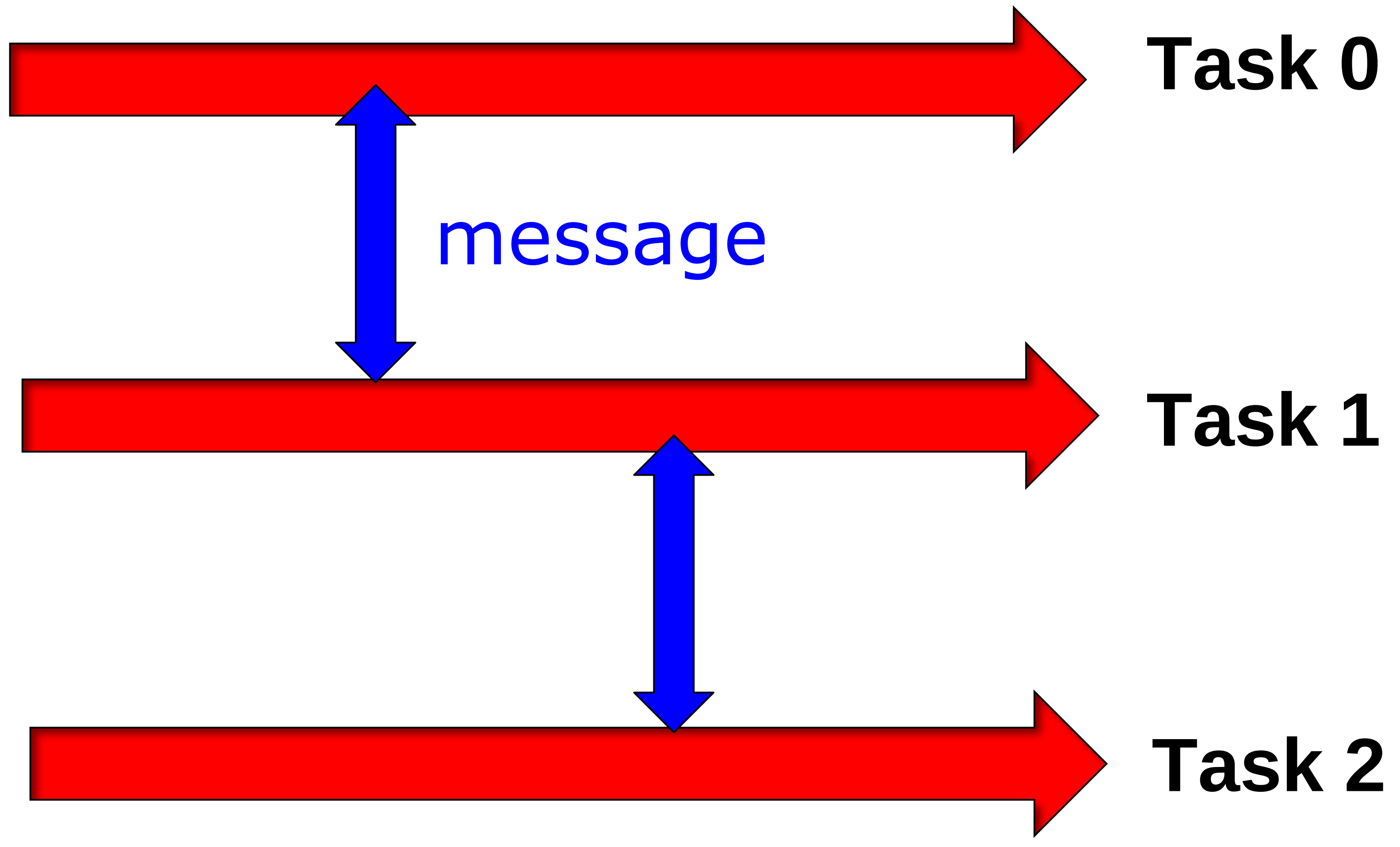
What is MPI?

MPI (**M**essage **P**assing **I**nterface) is a standard for parallel programming where:

- Each parallel process has its own memory space.
- Data communication or synchronization is explicit and occurs via function calls.
- Communication is possible both within shared memory nodes (intranode) and between nodes (internode)



Parallelism via Shared Memory (OpenMP)



Message Passing parallelism (MPI)

Advantages and Disadvantages

- ✓ Communications hardware and software are important components of an HPC system and often very highly optimised;
- ✓ Portable and scalable;
- ✓ Long history (many applications already written for it)

- X Explicit nature of message-passing is error-prone and discourages frequent communications;
- X Most serial programs need to be completely re-written;
- X High memory overheads.
- X Parallelism must be introduced explicitly.

The most important concept in Message Passing is…

TO MINIMIZE MESSAGE PASSING AS MUCH AS POSSIBLE

⚠ Excessive communication or synchronization degrades program performance and should be avoided unless absolutely necessary !

MPI standards and implementations

MPI FORUM

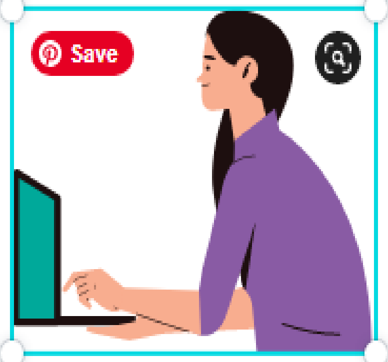
the MPI Forum defines standards, i.e. the commands need to run FORTRAN and C codes

MPI-1 (May 1994)

MPI-2 (April 1997)

MPI-3 (September 2012)

MPI-4 (June 2021)



FORTRAN or C/C++ program

intelmpi

openmpi

mpich

compilation and execution is implementation dependent

More on MPI implementations

- MPI is implemented as libraries, header files and other options which can be used with standard compilers (e.g. gcc)

- To avoid remembering these details it is usual for the MPI vendor to provide a *wrapped version* of the compiler and a program to launch the MPI executable.

- Be aware of which compiler you are using - for some implementations (e.g. openmpi, intelmpi) multiple compilers are available

```
$ module load openmpi

$ mpif90 -show
gcc -I/opt/tools/openmpi-4.0.5/include -pthread -Wl,-rpath -Wl,/opt/tools/openmpi-4.0.5/lib -Wl,--enable-new-dtags -L/opt/tools/openmpi-4.0.5/lib -lmpi
```

Sample compilation and execution

```
$ module load openmpi

# C++ program
$ mpicxx -o my_prog.exe
program.cpp
$ mpirun -n4 ./my_prog.exe

# C program with multiple source
files
$ mpicc -c f1.c
$ mpicc -c f2.c
$ mpicc -o prog.exe *.o
$ srun ./prog.exe
```

You will generally have to load a compiler module to have access to the MPI wrappers and launchers.

A launcher program, often mpirun, mpiexec or srun (if using SLURM).

Wrapped compilers for compilation and linking.

Do not run MPI on login nodes of a cluster - this could cause a crash or make the logins unusable for everyone.

The MPI compilers for C++ can have different names: mpicxx, mpic++, mpiCC. There is no real difference between these compilers.

Programming languages for MPI

The main programming languages for MPI are **Fortran** (not discussed here) and **C**

Programming with **C++** is possible, but:
“The C++ API was dropped from MPI-3 since it offered no real advantage over the C bindings, instead being a simple wrapper layer.”

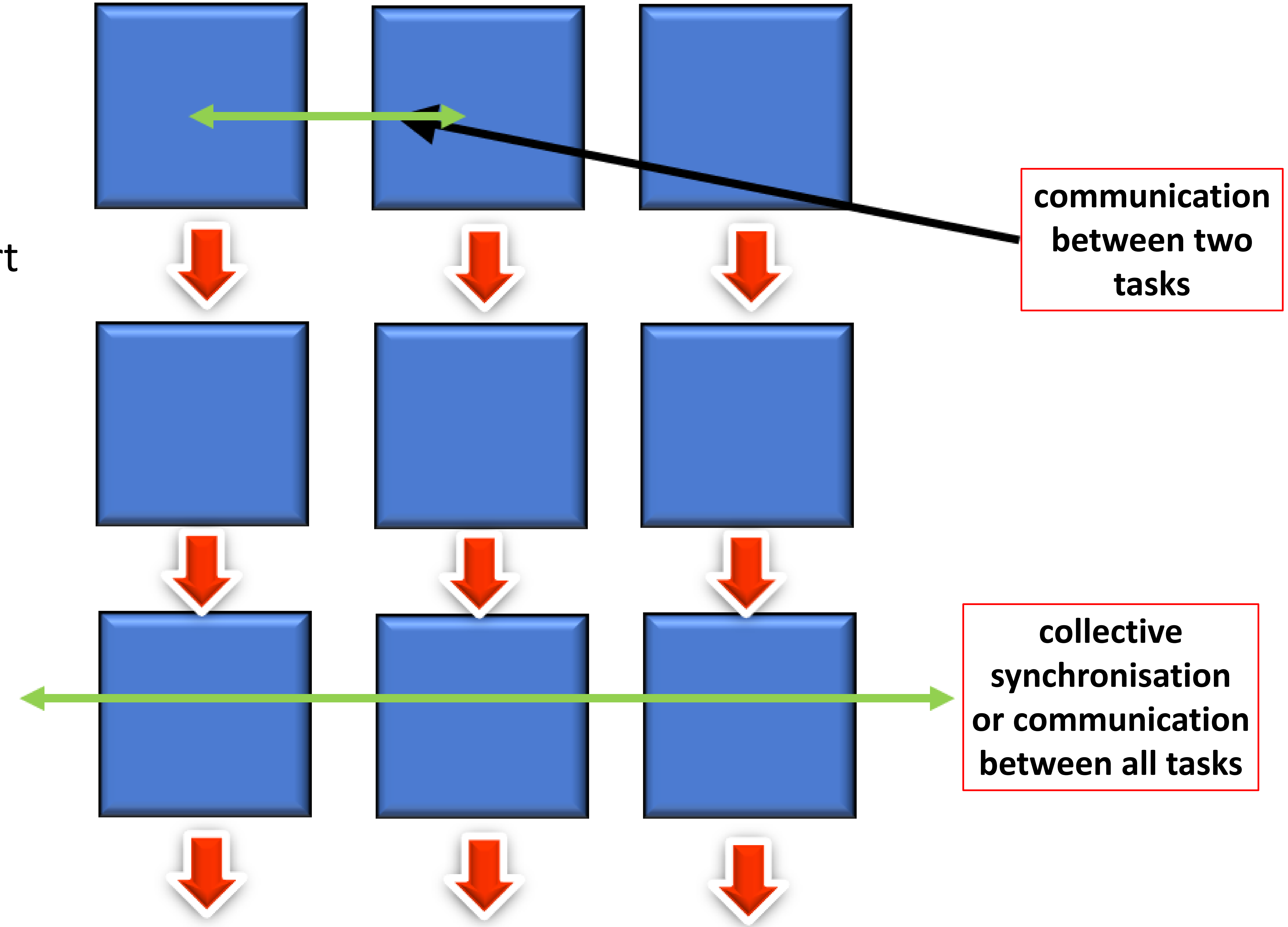
(from Stack Overflow)

Therefore, it is still possible to program with C++ as if it were C, but support is deprecated: standards do not define C++ APIs

There is also the possibility for **Python** users to program with MPI, taking advantage of the “mpi4py” Python package.

Ok, but where do I start?

- Useful to start with the SPMD (Single Program Multiple Data) Model.
- Many MPI ***tasks*** are launched at the start of program execution.
- Each task has its own local memory which is completely separate from the others unless…
- …a communication step transfers some data.
- Synchronization may be needed to ensure the parallel program is correct



MPI Commands available

1. Calls used to initialize, manage, and terminate communications.
2. Calls used to communicate between pairs of processors (point to point communication).
3. Calls used to communicate among groups of processors (collective communication).
4. Calls to create data types and topologies.

There are very many MPI calls, probably hundreds, so we will describe just the most important ones.

My first MPI program

Let’s get started – Hello World

C

```c
#include <stdio.h>
#include <mpi.h>
void main(int argc, char * argv[])
{
   int err;
   err = MPI_Init(&argc, &argv);
   printf(“Hello world!\n”);
   err = MPI_Finalize();
}
```

C++

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;
void main(int argc, char * argv[]
){
   int err;
   err = MPI_Init(&argc, &argv);
   cout << “Hello world!” << endl;
   err = MPI_Finalize();
}
```

Note: “**mpi.h**” is mandatory – it’s the header file including all the variables and functions defined by MPI standard

MPI Function format

```
int error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

- C requires that MPI is all uppercase and what is after the underscore has the first letter uppercase and the rest lowercase
- The return value is an integer, that returns 0 if the MPI call completed successfully and a number otherwise (depending on the error code)
- It’s up to you whether you want to store the return value in a variable. It can be useful for debugging purposes

Initializing MPI

```
int MPI_Init(int *argc, char **argv);
```

- Must be the first MPI call: initializes the message passing routines
- The arguments in MPI_Init in the C version are not used but some compilers insist they are there.

Finalizing MPI

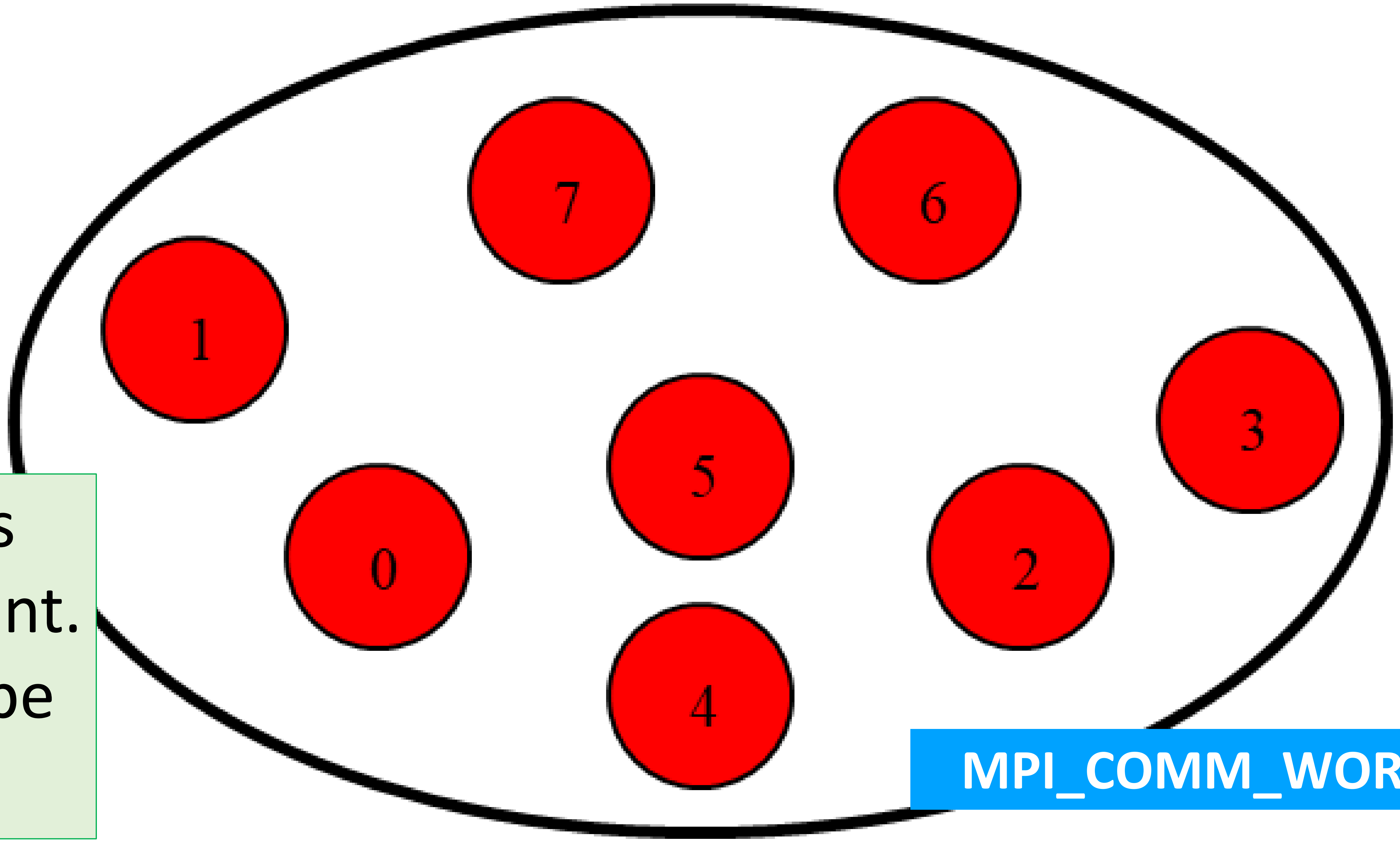
```
int MPI_Finalize();
```

These two subprograms should be called by all processes, and no other MPI calls are allowed before **MPI_Init** and after **MPI_Finalize**.

However the program can go on as a serial program

MPI Communications and communicators

- It is possible to divide the total number of tasks into groups called *communicators*.
- The variable identifying a communicator identifies those tasks which can communicate with each other.
- The default communicator is called MPI_COMM_WORLD and by default includes all the tasks available to the program.



MPI_COMM_WORLD with 8 tasks

- All communication commands have a communicator argument.
- Multiple communicators can be defined at any one time.

Using communicators

C

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char * argv[] ){
   int err;
   int nprocs, my_rank;
   err = MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
   printf("Hello I am %d of %d procs \n",
     my_rank, nprocs);
   err = MPI_Finalize();
}
```

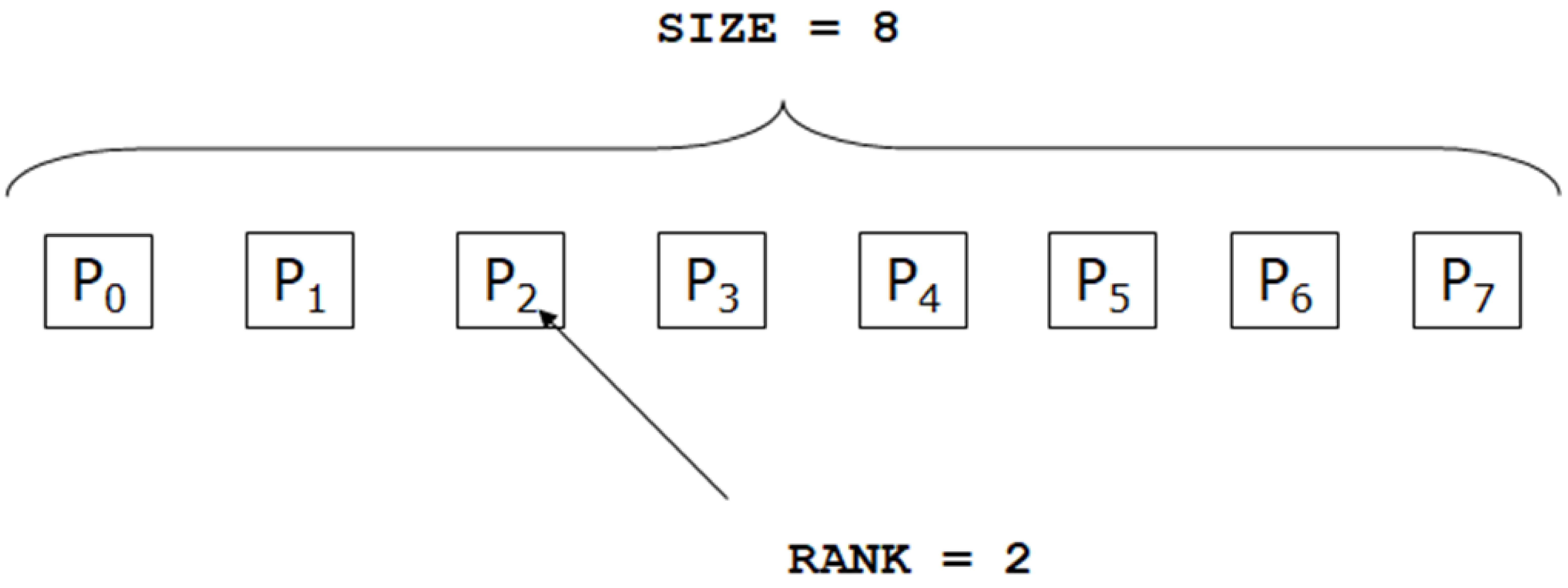
C++

```
#include <iostream>
#include <mpi.h>
using namespace std;
void main(int argc, char * argv[] ){
   int err;
   int nprocs, my_rank;
   err = MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
   cout << "Hello I am " << my_rank << " of "
        << nprocs << " procs" << endl;

   err = MPI_Finalize();  }
```

Each task in a communicator is identified by its ***rank***
The rank varies **from 0 to n-1**, where n=the number of tasks in the processors.
A rank in MPI identifies a **task**, not necessarily linked to a hardware processor or core.

Communicator size and process rank

How many processes are contained within a communicator?



size is the number of processes associated to the communicator

rank is the index of the process within a group associated to a communicator (**rank** = 0,1,...,n-1)
The rank is used to identify the source and destination process in a communication

Communicator size

How many processes are associated with a communicator?

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

After the call, size = number of processes

Process rank

How can you identify different processes?
What is the ID of a processor in a group?

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

rank is an integer that identifies the Process inside the communicator *comm*

MPI_Comm_rank is used to find the rank (the name or identifier) of the Process running the code

Remember that every process is running the same code independently: at the end of the call, *rank* will have a **different value** for every process!

Same example – makes more sense now?

C

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

void main(int argc, char * argv[] ){

    int err;

    int nprocs, my_rank;

    err = MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs)

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)

    printf("Hello I am %d of %d procs \n",
      my_rank, nprocs);

    err = MPI_Finalize();

}
```

C++

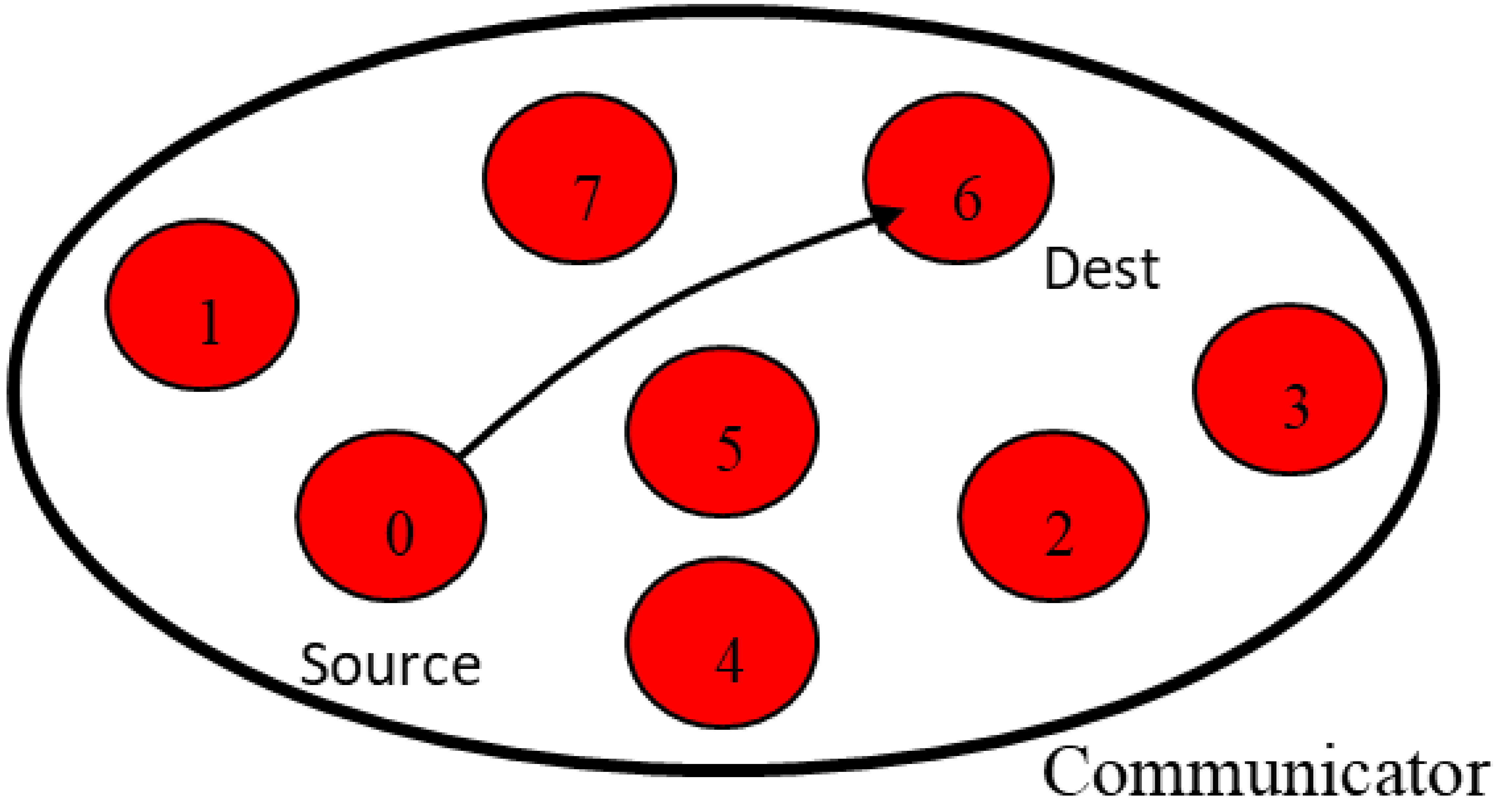
```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

void main(int argc, char * argv[] ){

    int err;

    int nprocs, my_rank;

    err = MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs)

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)

    cout << "Hello I am " << my_rank << " of "
         << nprocs << " procs" << endl;

    err = MPI_Finalize();  }
```

```
Hello I am 1 of 4 procs
Hello I am 0 of 4 procs
Hello I am 3 of 4 procs
Hello I am 2 of 4 procs
```

Point-to-point communication: Send and Receive

Point-to-point communication

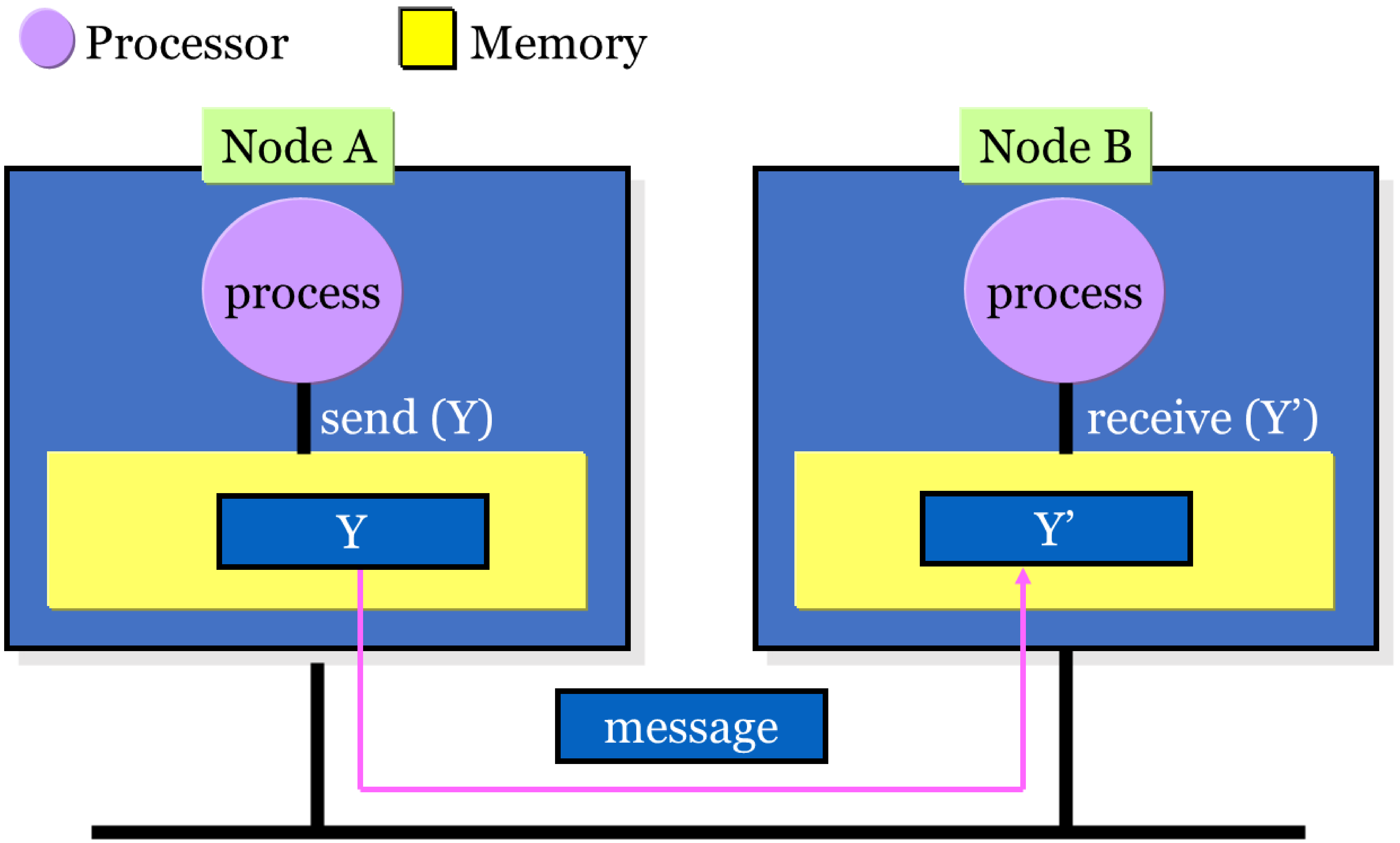
- It is the basic communication method provided by MPI library - communication between 2 processes.
- It is conceptually simple: source process **A** sends a message to destination process **B**; **B** then receives the message from **A**.
- Communication take places within a communicator.
- Source and Destination are identified by their rank in the communicator.



Quick example of point-to-point

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0) {
 MPI_Send(a,2,MPI_FLOAT,1,10,MPI_COMM_WORLD);
}
else if (my_rank==1)  {
 MPI_Recv(b,2,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
}
```

MPI Point-to-point programming model



The message

- Data is exchanged in the buffer, an array of *count* elements of some particular MPI data type
- One argument that usually must be given to MPI routines is the *type* of the data being passed.
- This allows MPI programs to run automatically in heterogeneous environment.

Messages are identified by their envelopes. A message could be exchanged only if the sender and receiver specify the correct envelope

Message Structure

body			envelope			
buffer	count	datatype	source	destination	communicator	tag

MPI Data types

- MPI Data types can be:
  - Basic types (portability)
  - Derived types (MPI_Type_xxx functions)

- A derived type can be built up from basic types

- User-defined data types allows MPI to automatically scatter and gather data to and from non-contiguous buffers

- MPI defines ‘handles’ to allow programmers to refer to data types and structures

- C/C++ handles are macros to structs (#define MPI_INT …)

i

MPI Derived types will be discussed in a further lesson.

MPI intrinsic datatypes - C

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

For a communication to succeed…

1. **Sender** must specify a valid **destination rank**.
2. **Receiver** must specify a valid **source rank**.
3. The **communicator** must be **the same**.
4. **Tags must match**.
5. **Buffers** must be **large enough**.

Must check very carefully all the arguments of the commands: the command may succeed, but with wrong data!

Completion

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. The MPI implementation is able to deal with storing data when the two tasks are out of sync.

- Completion of the communication means that memory locations used in the message transfer can be safely accessed:
    - Send: variable sent can be reused after completion
    - Receive: variable received can be used after completion

Blocking communication

Most of the MPI point-to-point routines can be used in either **blocking** or **non-blocking** mode.

Blocking mode:

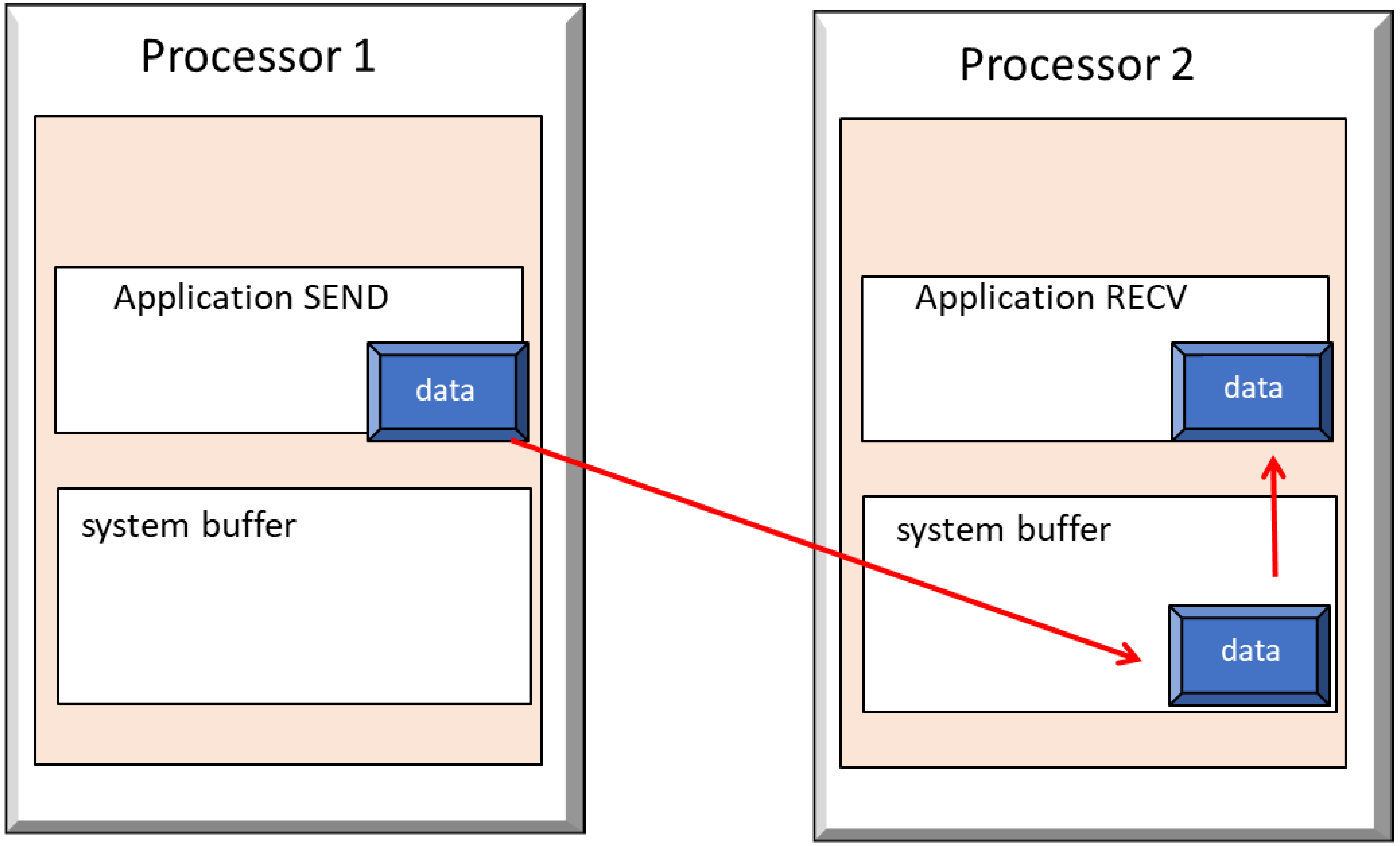
- A blocking send returns after it is safe to modify the application buffer (your send data) for reuse. **Safe does not imply that the data was actually received** - it may very well be sitting in a system buffer.

- A blocking send can be synchronous.

- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.

- **A blocking receive only "returns" after the data has arrived and is ready for use by the program.**

A blocking communication



How buffers are used is implementation specific - MPI does not define the use of buffers, only the semantics of blocking communications (excl. buffered sends).

Blocking send and receive

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

```
int MPI_Recv(void *buf, int count, MPI_Datatype type, int source,
             int tag, MPI_Comm comm, MPI_Status *status);
```

buf	array of type **type**(see table).
count	number of element of **buf** to be sent
type	MPI type of **buf**
dest	rank of the destination process
source	rank origin of send process
tag	number identifying the message
comm	communicator of the sender and receiver
status	array of size **MPI_STATUS_SIZE** containing communication status information (Orig Rank, Tag, Number of elements received)

Message body

Message envelope

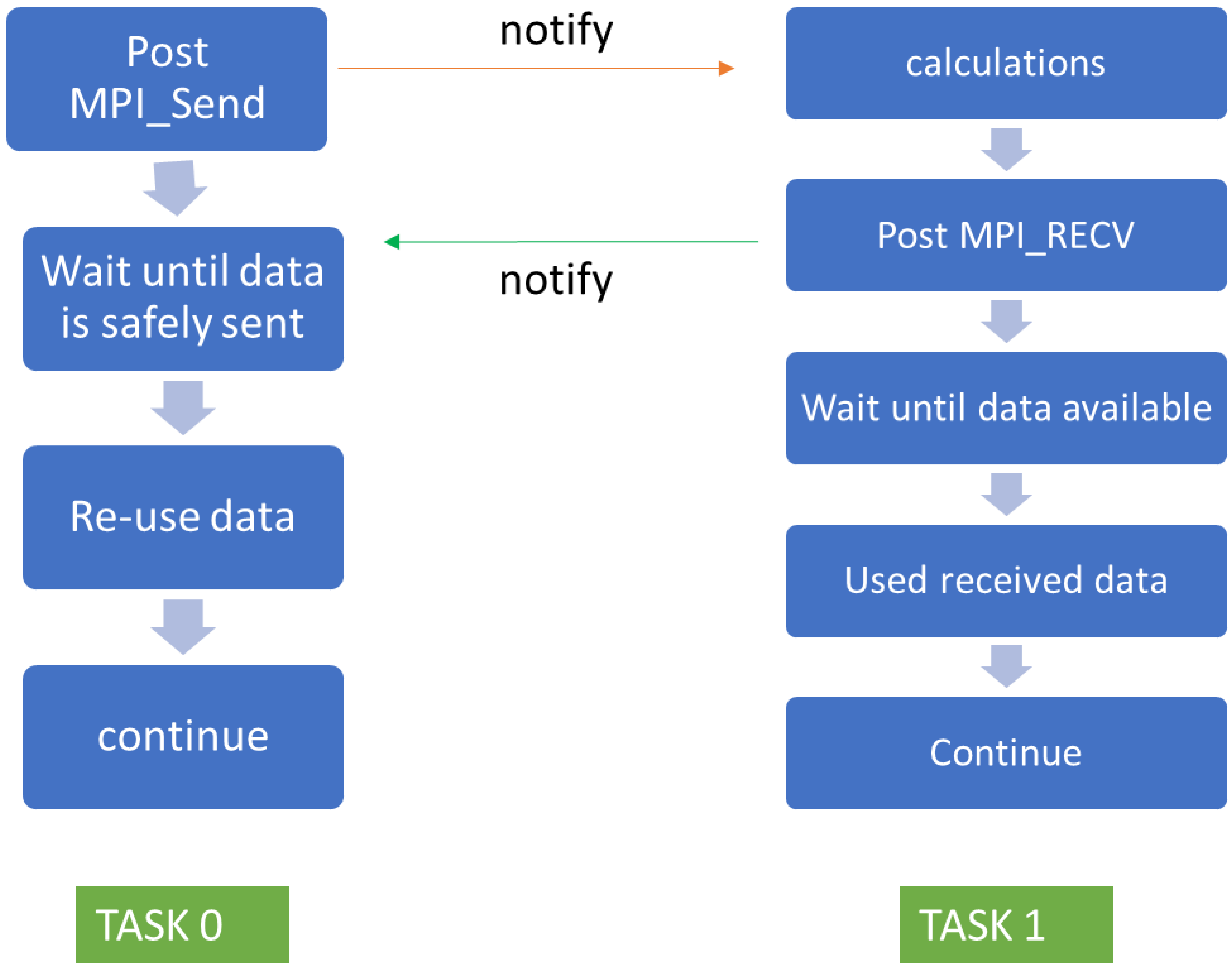
WILDCARDS

Wildcards are also accepted,

- To receive from any source: MPI_ANY_SOURCE
- To receive with any tag: MPI_ANY_TAG

Actual source and tag are returned in the receiver's status parameter

Blocking communication



Blocking send and receive – code example

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

if (my_rank==0) {
 a[0]=3.0;
 a[1]=5.0;
 MPI_Send(a,2,MPI_FLOAT,1,10,MPI_COMM_WORLD);
}

else if (my_rank==1)  {
 MPI_Recv(b,2,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
 printf("My rank is %d: b[0]=%f, b[1]=%f\n", my_rank, b[0], b[1]);
}

MPI_Finalize();
```

At this point we can modify A in rank 0

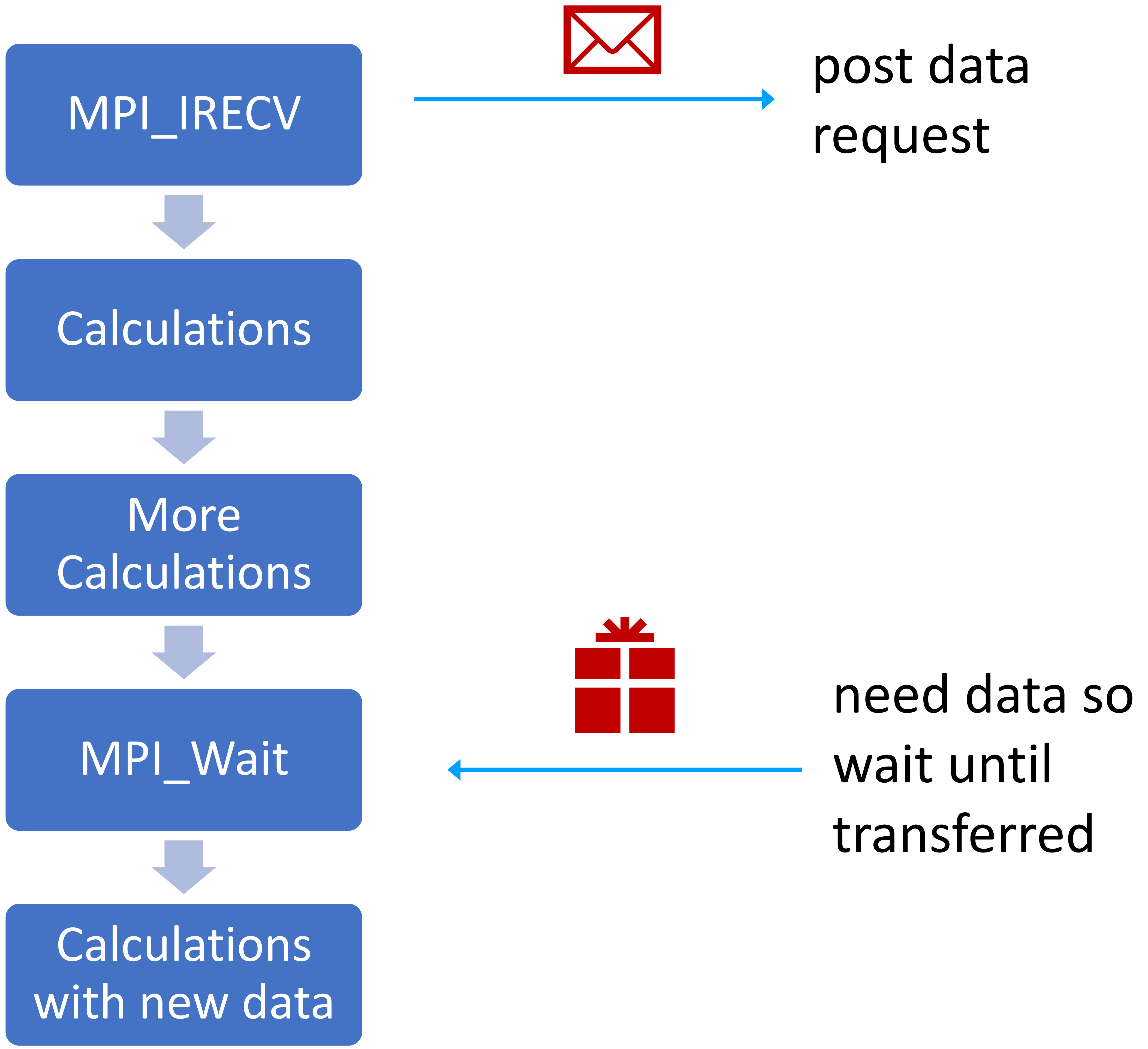
At this point we can use B in rank 1

Non-blocking communication

Most of the MPI point-to-point routines can be used in either **blocking** or **non-blocking** mode.

Non-blocking mode:

- Non-blocking send and receive routines will return almost immediately. They do not wait for any communication events to complete.
- Non-blocking operations simply "**request**" the MPI library to perform the operation when it is possible. The user can not predict when that will happen.
- It is unsafe to modify the application buffer until you know for a fact that the requested non-blocking operation was actually performed by the library. There are "**wait**" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication. This is very important for performance reasons!



Non-blocking send and receive

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest,
                  int tag, MPI_Comm comm, MPI_Request *req);
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype type, int source,
                  int tag, MPI_Comm comm, MPI_Request *req);
```

buf	array of type **type**(see table).	Message body
count	number of element of **buf** to be sent	
type	MPI type of **buf**	
dest	rank of the destination process	Message envelope
source	rank origin of send process	
tag	number identifying the message	
comm	communicator of the sender and receiver	
req	output, identifier of the communications handle	

i

- The **I** indicates non-blocking also for other MPI commands
- Note that in MPI_Irecv the **status** is missing

Waiting for completion

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
```

A call to this subroutine causes the code to wait until the communication pointed by req is complete.

req : input, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

status : output, array of size **MPI_STATUS_SIZE**. if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

```
int MPI_Waitall(int count, MPI_Request **array_of_requests,
                MPI_Status **array_of_statuses);
```

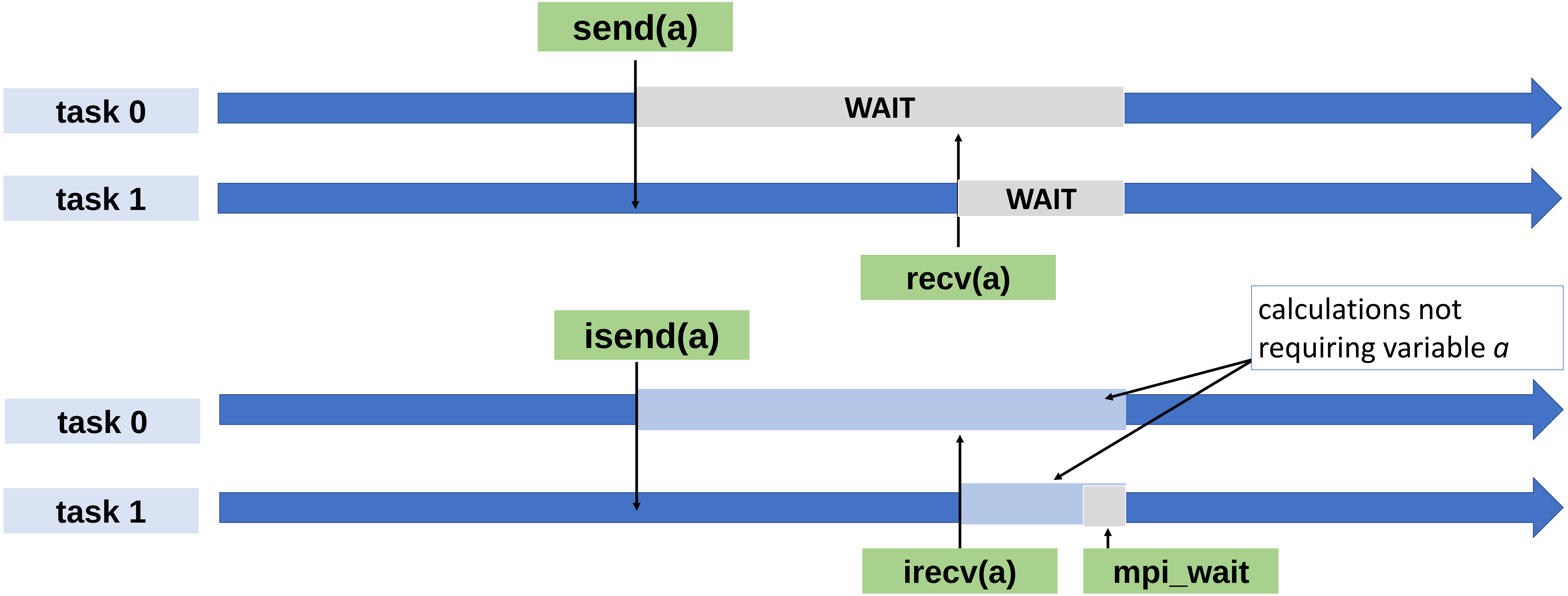
MPI_Waitall waits for multiple non-blocking calls at once, hence the use of array of statuses and requests.
The additional parameter "**count**" is the dimension of the arrays, i.e. the number of Isend/Irecv it has to wait for

Example of non-blocking point-to-point

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

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

int size, rank;
double a[2],b[2];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Request request1,request0;
MPI_Status status;

if (rank ==0) {
   a[0]=2.0;
   a[1]=4.0;
   MPI_Irecv(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD,&request0);
   MPI_Send(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
}
else if (rank==1) {
   a[0]=3.0;
   a[1]=5.0;
   MPI_Irecv(b,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD,&request1);
   MPI_Send(a,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD);
   MPI_Wait(&request1,MPI_STATUS_IGNORE);

}

printf("%d b[0]=%lf, b[1]=%lf \n",rank,b[0],b[1]);


MPI_Finalize();
```

Why use non-blocking communications?

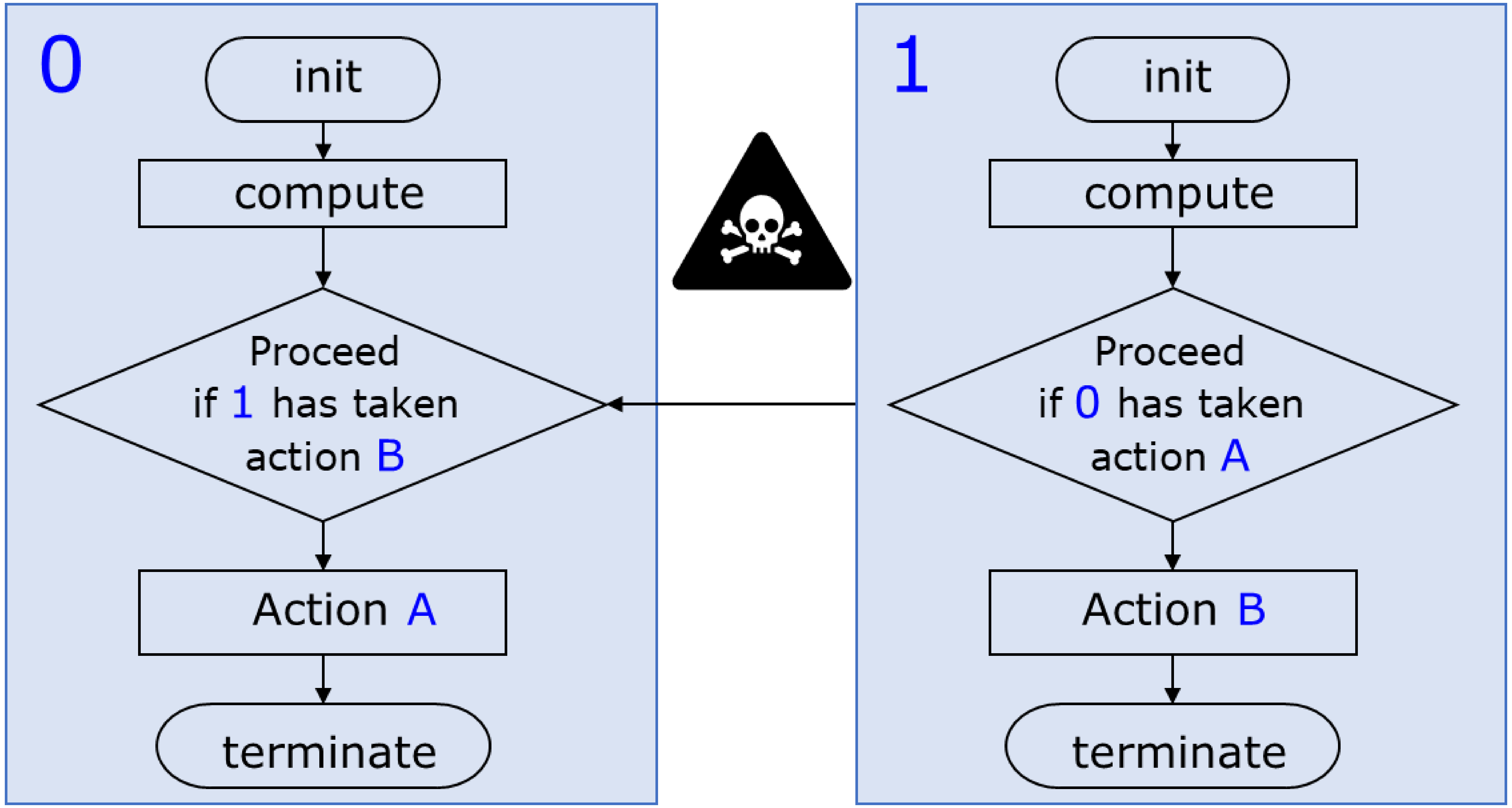
- A major inefficiency in using blocking communications is due to the waiting time between sending and receiving messages.
- With non-blocking communications it may be possible to overlap communication and calculations and reduce the waiting time.



The dreaded DEADLOCK

Deadlock

A **Deadlock** or a Race condition occurs when 2 (or more) processes are blocked, and each is waiting for the other to make progress.



One result is that the allocated time (and budget) may expire but no work is actually done.

MPI Deadlock example

This is an example of deadlock. Why ?

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

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

int size, rank;
double a[2],b[2];

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

if (rank ==0) {
   a[0]=2.0;
   a[1]=4.0;
   MPI_Recv(b,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   MPI_Send(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
}
else if (rank==1) {
   a[0]=3.0;
   a[1]=5.0;
   MPI_Recv(b,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   MPI_Send(a,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD);
}
printf("%d b[0]=%lf, b[1]=%lf \n",rank,b[0],b[1]);


MPI_Finalize();


}
```

Avoiding deadlock

Reasons for deadlock in MPI include:

- Incorrect ordering of MPI_Send and MPI_Recv;
- Misaligned tags;
- Other situations with misaligned sends/recvs (e.g. a rank is not included in the communicator);

Deadlock can be avoided by:

- Checking correct alignment of sends/recvs and matching tags;
- Non-blocking send/recv;
- MPI_Sendrecv.

This code avoids deadlock (send-recv inverted)

```
if (my_rank==0) {
  a[0]=2.0; a[1]=4.0;
  MPI_Send(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
  MPI_Recv(b,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
else if (my_rank==1)  {
  a[0]=3.0; a[1]=5.0;
  MPI_Recv(b,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  MPI_Send(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
}
```

Question: what about this one?

```
if (my_rank==0) {
  a[0]=2.0; a[1]=4.0;
  MPI_Send(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
  MPI_Recv(b,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
else if (my_rank==1)  {
  a[0]=3.0; a[1]=5.0;
  MPI_Send(a,2,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
  MPI_Recv(b,2,MPI_DOUBLE,0,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
```

MPI_Sendrecv

```
int MPI_Sendrecv(void *snd_buf, int snd_count, MPI_Datatype snd_type,
    int dest, int tag, void *rcv_buf, int rcv_count, MPI_Datatype
    rcv_type, int src, int tag, MPI_Comm comm, MPI_Status status);
```

Green= Sender specifics (buffer, size, datatype, dest, tag)

Gold= Receiver specifics (buffer, size, datatype, source, tag)

Black= General specifics (communicator, status)

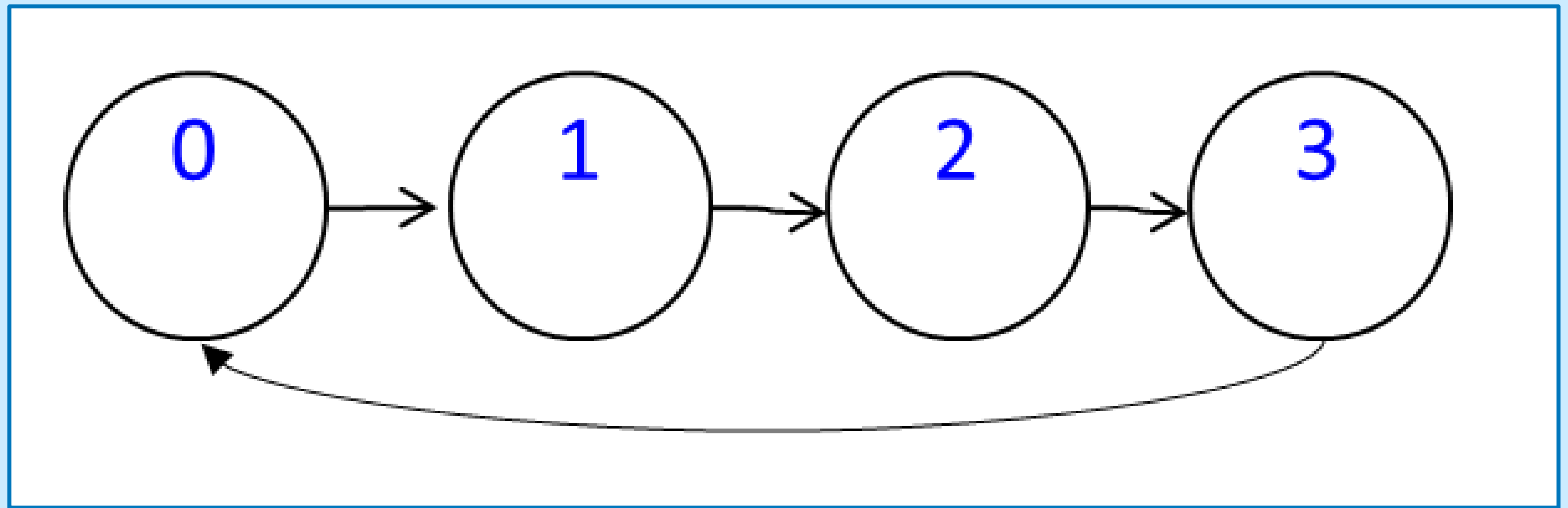
- **MPI_SendRecv** simultaneously posts an MPI_Send and an MPI_Recv.

- This allows 3 ranks to be involved in the call:
  1. The calling process
  2. The process providing the data
  3. The process receiving data.

- This makes it useful for **cyclic communication patterns**.

MPI_Sendrecv example

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

int main(int argc, char *argv[])
{
    int my_rank, procs, left, right;
    int buffer_s, buffer_r;
    MPI_Status status;

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

    right = (my_rank + 1) % procs;
    left = my_rank - 1;
    if (left < 0)
        left = procs - 1;

    buffer[0]=my_rank;
    MPI_Sendrecv(&buffer_s, 1, MPI_INT, right, 123, &buffer_r, 1, MPI_INT, left, 123, MPI_COMM_WORLD, &status);
    MPI_Finalize();
    return 0;
 }
```



rank 1 receives data from 0 and sends data to 2

Summary

Point-to-point is the most basic communication method in MPI, using 2 or 3 processes.

Important to recognise the difference between blocking and non-blocking methods.

Blocking should be safer, but you may get better performance with non-blocking.

Beware of deadlocks: check send/recv order or consider non-blocking or MPI sendrecv

Current MPI Standard to be found at: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
Credits to A. Emerson and many other colleagues from CINECA for the slides