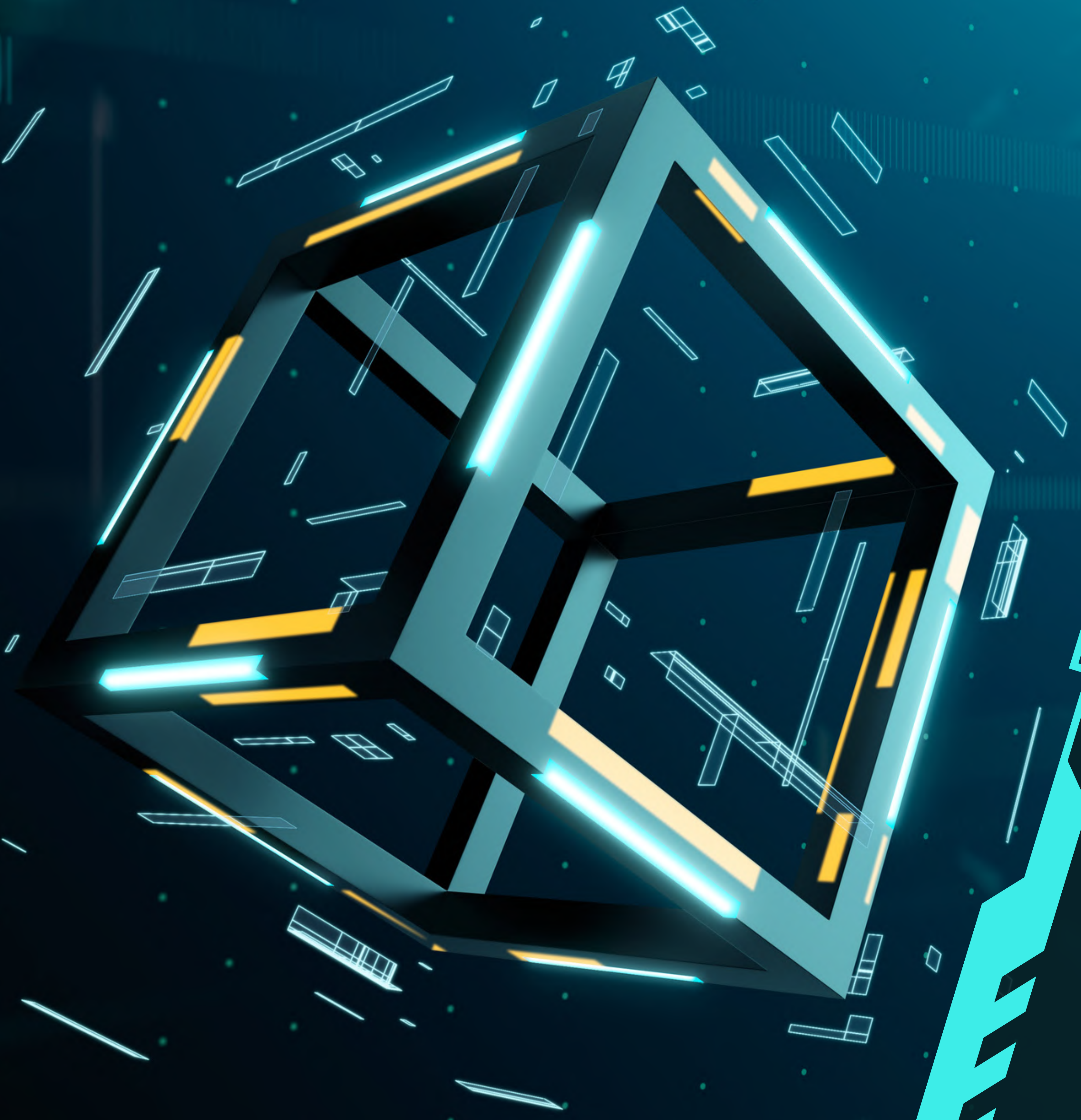


MPI

MESSAGE PASSING INTERFACE

TEAM WORK

- Djebbar Yehya
- Hadjazi Mohammed
- Mletta Mohammed
- Sahraoui Mohammed

Supervisor: Dr. Behri Mohammed

Class : Parallels programming

TABLE OF CONTENT

OBJECTIVE

- What Does Message Passing Interface (MPI) Mean?
- Why bother with MPI ?
- Should we use MPI for all distributed computing ?
- Where to install MPI ? And How (practical part)
- What MPI consists of ?
- How to write our first MPI program ?
- How to compile and execute MPI program ?
- Understand what's going on behind the seen ?
- Introduce MPI Mechanism in details (Functions and communication types)
- Reach the objective of parallels programming course

1.1 What Does Message Passing Interface (MPI) Mean?

MPI stands for Message Passing Interface and is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.

The system aims to provide a portable and efficient standard for message passing. It is widely used for message passing programs, as it defines useful syntax for routines and libraries in different computer programming languages such as Fortran, C, C++ and Java.

1.1 What Does Message Passing Interface (MPI) Mean?

MPI stands for Message Passing Interface and is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.

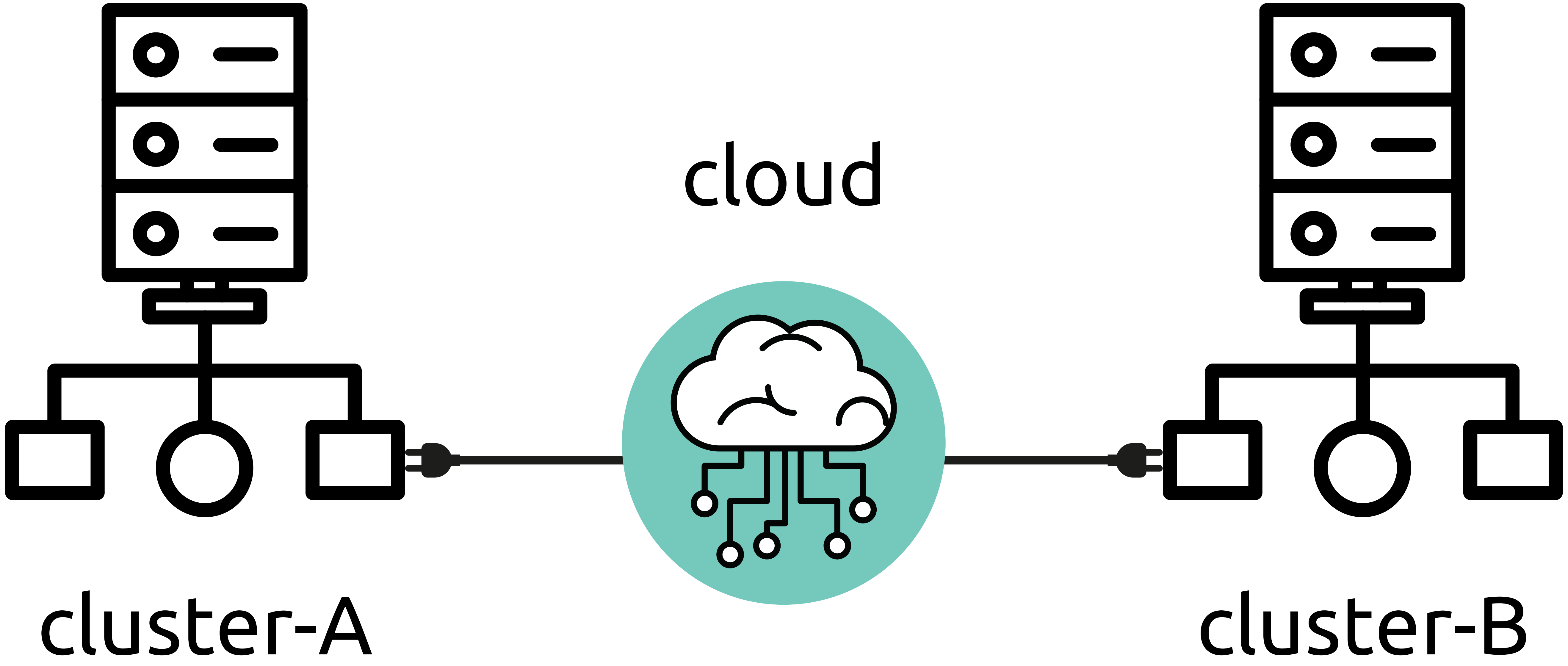
The system aims to provide a portable and efficient standard for message passing. It is widely used for message passing programs, as it defines useful syntax for routines and libraries in different computer programming languages such as Fortran, C, C++ and Java.

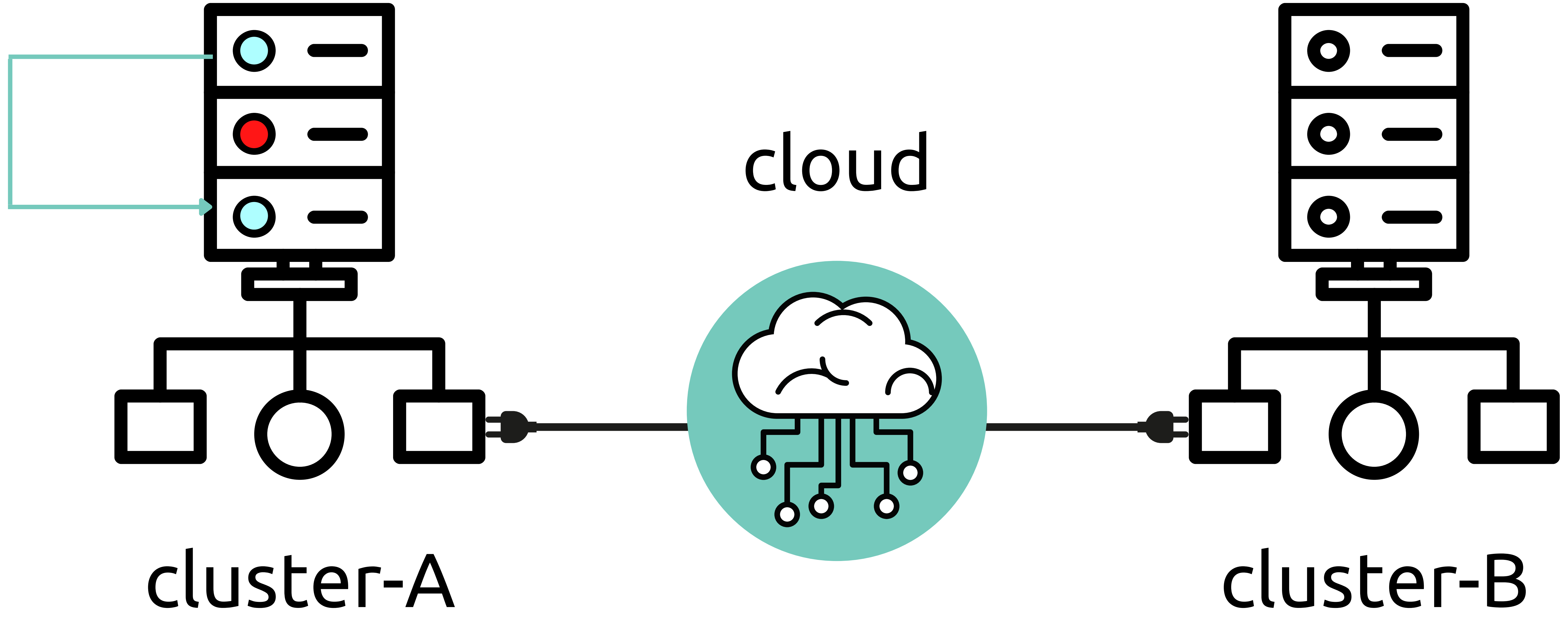
1.2 Why bother with MPI ? as we already have networking libraries for communication between processes ?

- Optimized for Performance.
- It will take advantage of the fastest transport found.
- Shared memory (within the same computer).
- Fast Cluster interconnects (Infinband, Myrinet,....) between computer nodes.
- TCP/IP if all else fails.
- Enforces other gurantees.
- Relible messages.
- In-order arrival of messages.
- Designed for multi-node technical computing.
- Completly standard (available everywhere).
- Has specialized routines for collective operations.

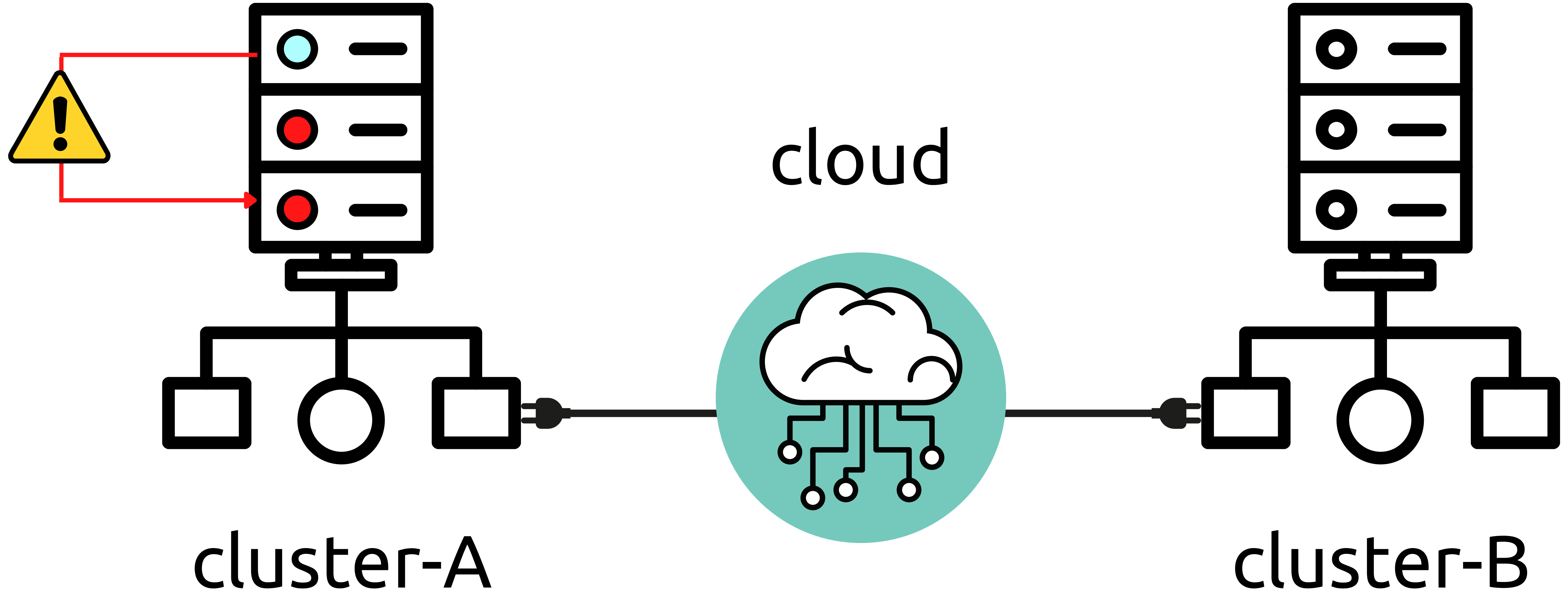
1.2 Why bother with MPI ? as we already have networking libraries for communication between processes ?

- Optimized for Performance.
- It will take advantage of the fastest transport found.
- Shared memory (within the same computer).
- Fast Cluster interconnects (Infinband, Myrinet,....) between computer nodes.
- TCP/IP if all else fails.
- Enforces other gurantees.
- Relible messages.
- In-order arrival of messages.
- Designed for multi-node technical computing.
- Complетly standard (available everywhere).
- Has specialized routines for collective operations.

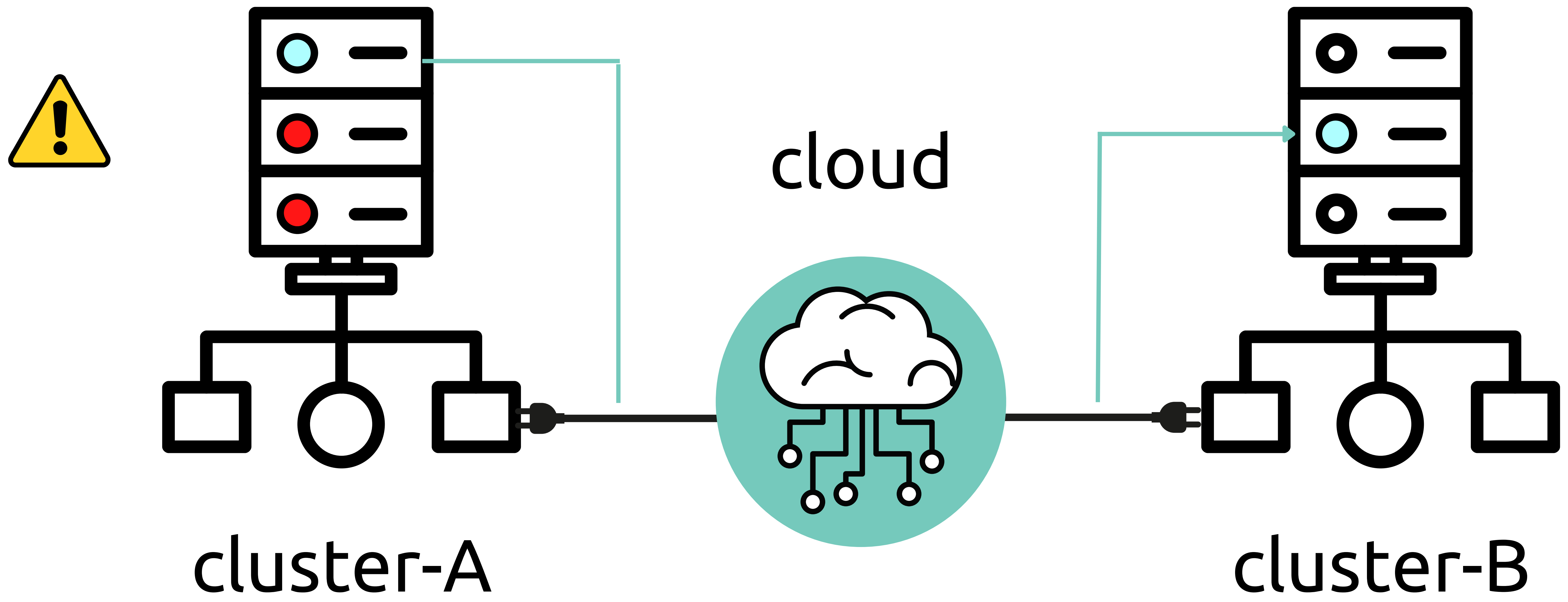

cloud
cluster-A
cluster-B


cloud
cluster-A
cluster-B

All computers are busy



Use TCP/IP to communicate



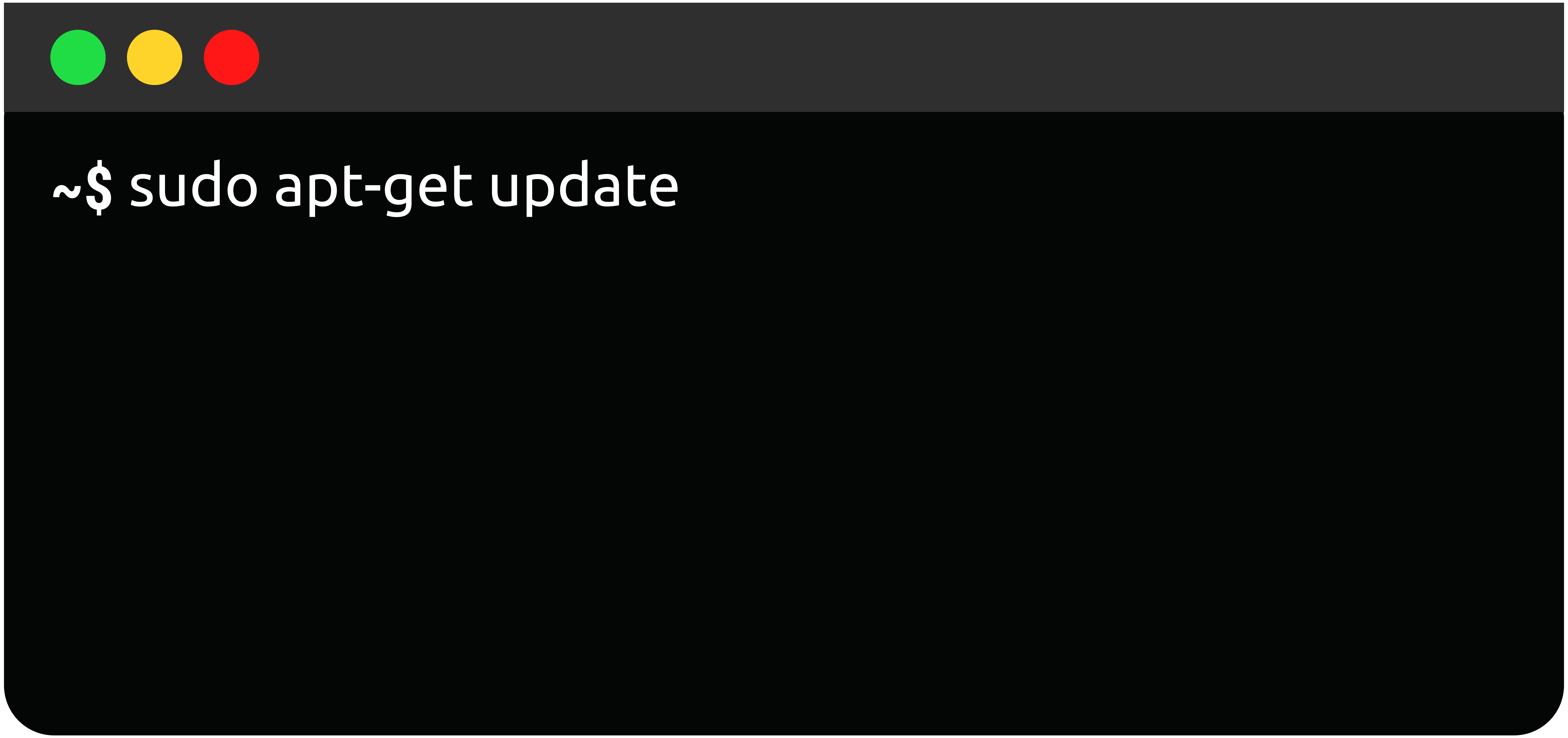
2. Where to install MPI ?

- **Linux users:** 💎
  OpenMPI
  http://www.open-mpi.org/

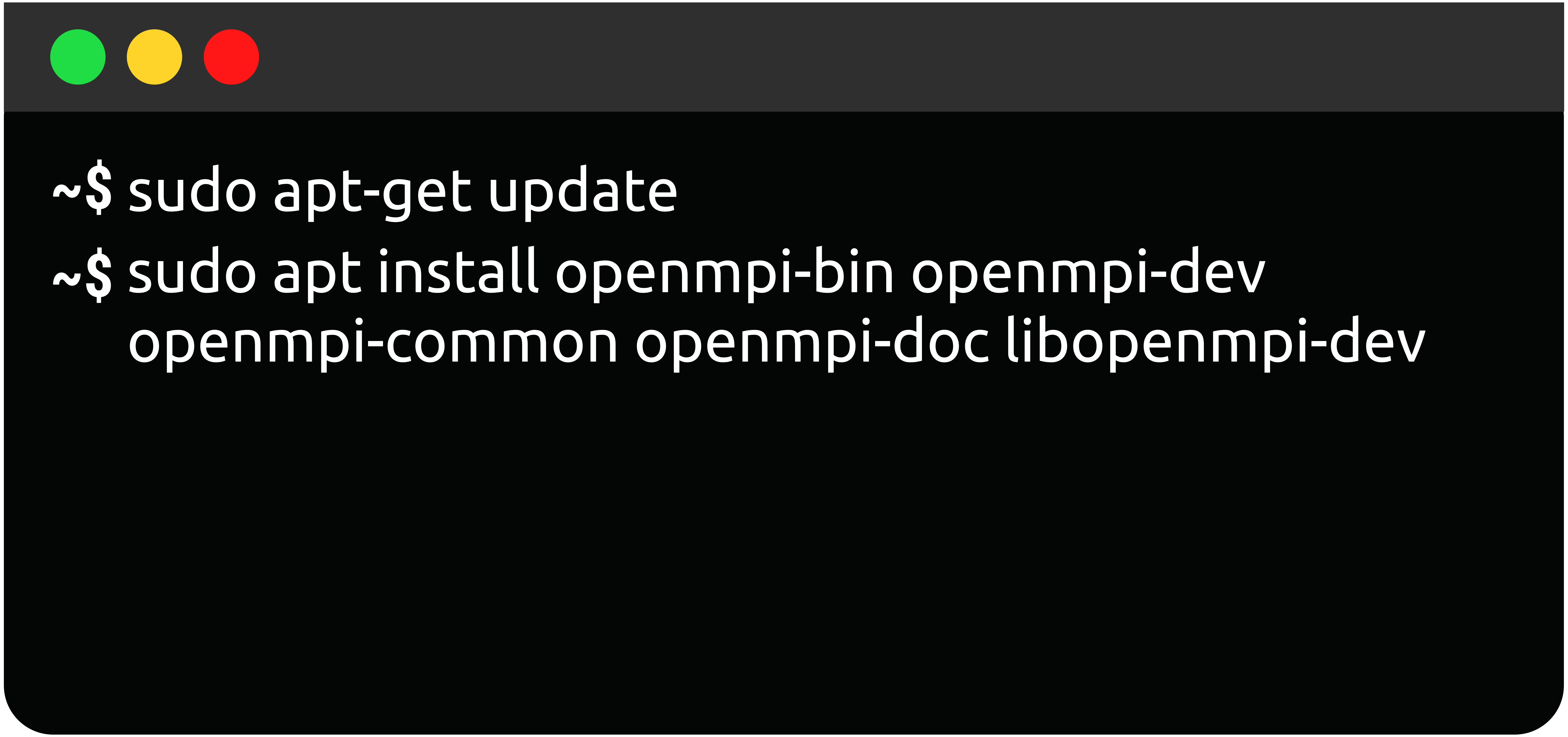
  MPICH2
  http://www.mcs.anl.gov/research/projects/mpich2/index.php

- **Microsoft:**
  https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi

Installation for Linux

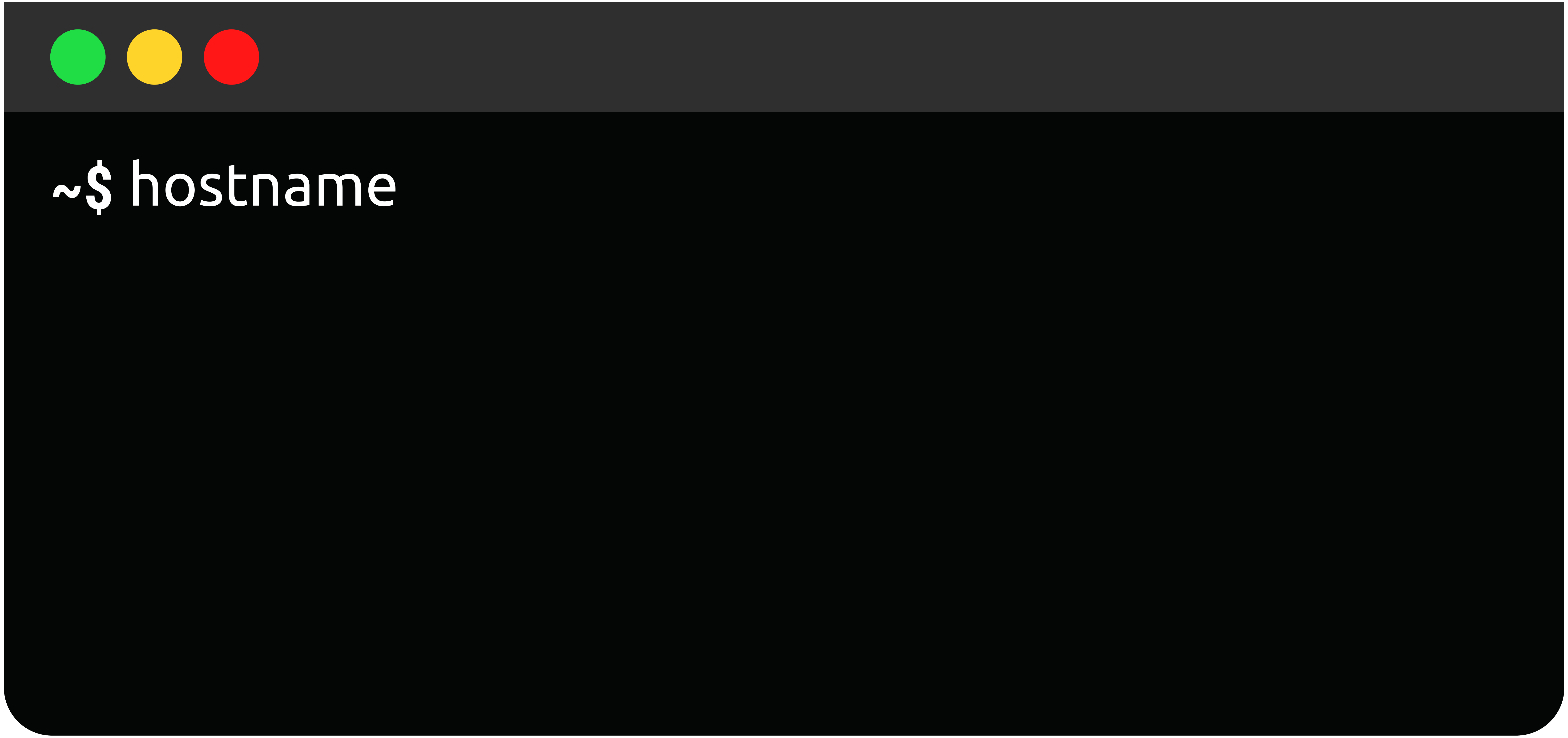


```
~$ sudo apt-get update
```

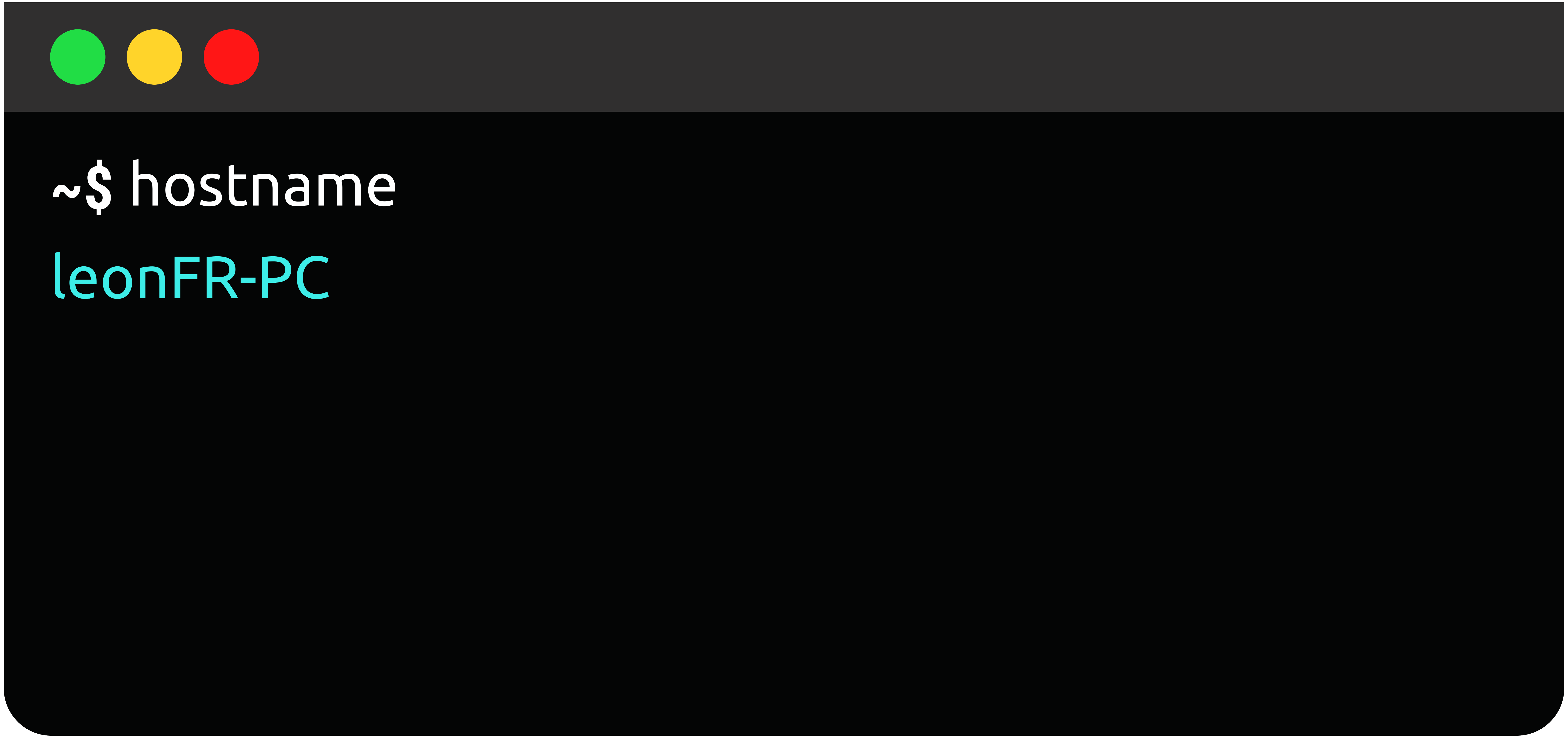


```
~$ sudo apt-get update
~$ sudo apt install openmpi-bin openmpi-dev
   openmpi-common openmpi-doc libopenmpi-dev
```

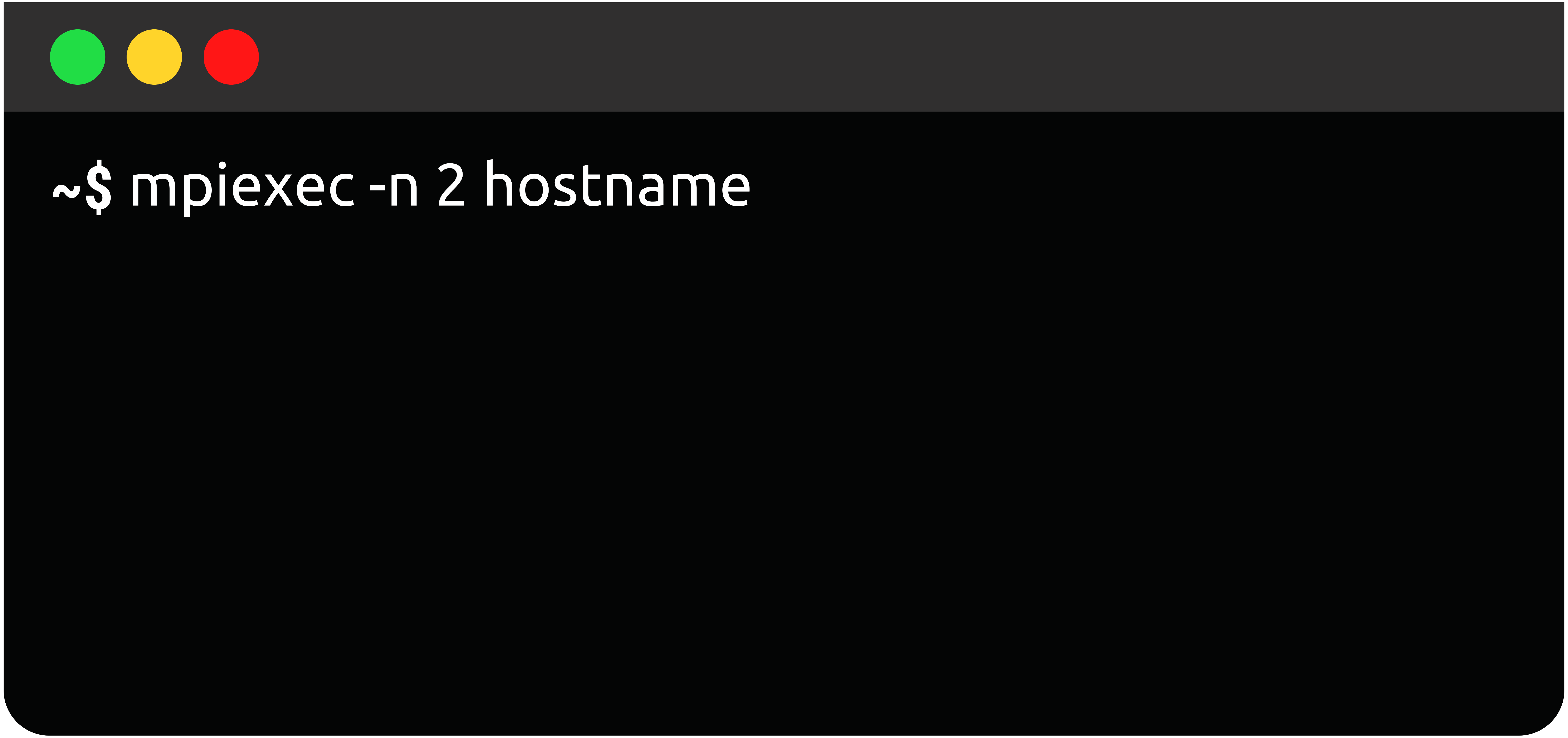
Installation for Microsoft



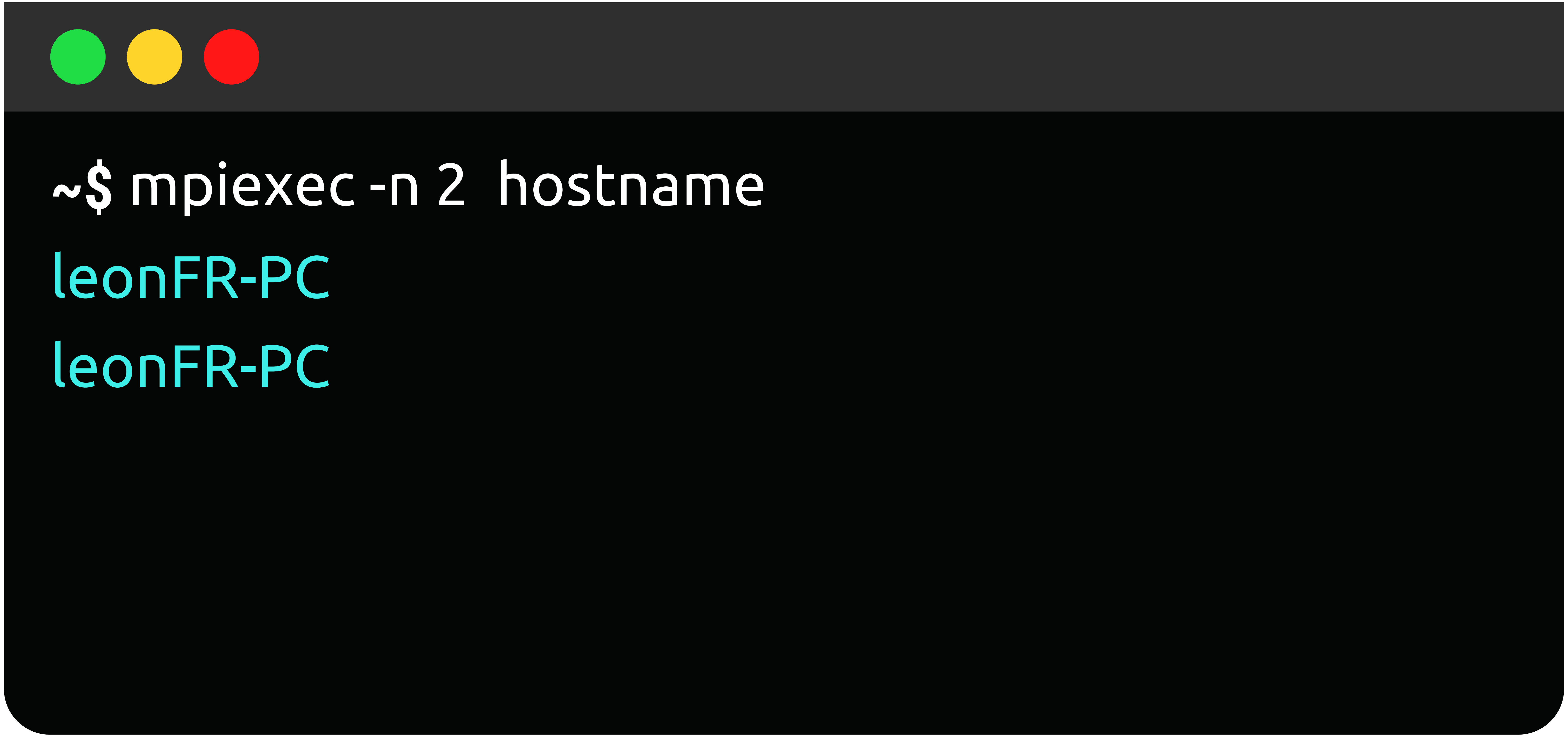
```
~$ hostname
```



```
~$ hostname
leonFR-PC
```



```
~$ mpiexec -n 2 hostname
```

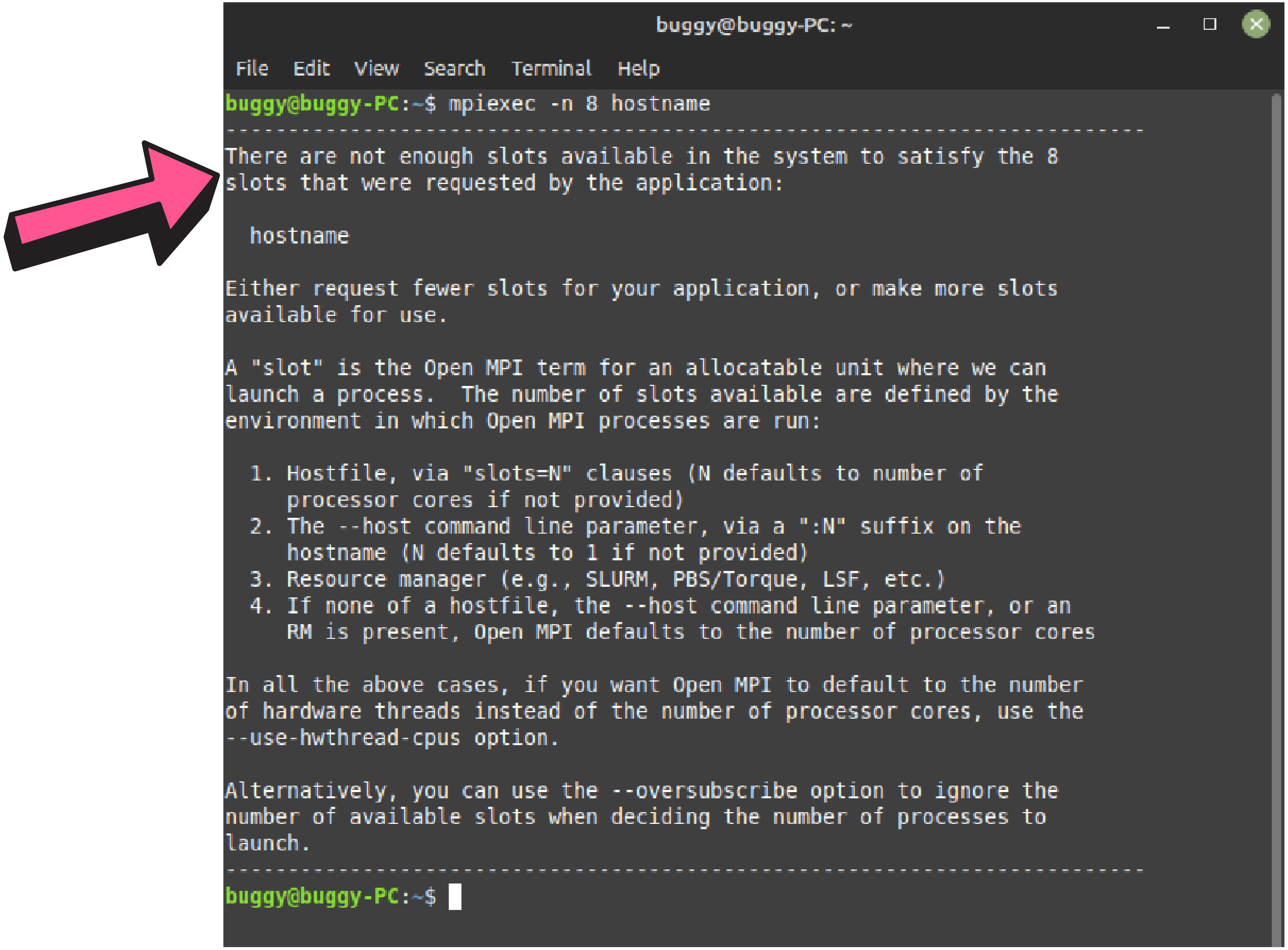


```
~$ mpiexec -n 2  hostname
leonFR-PC
leonFR-PC
```

2.2 What happened ?

- mpiexec luanched 2 proceses on my computer
- Each process ran the program hostname
- each ran independently in its oun enviroment
- you can do as many as you want as long you have the numbers of processors that supports it. (ex: if you have an 8 core cpu you can run it 8 times independently.)

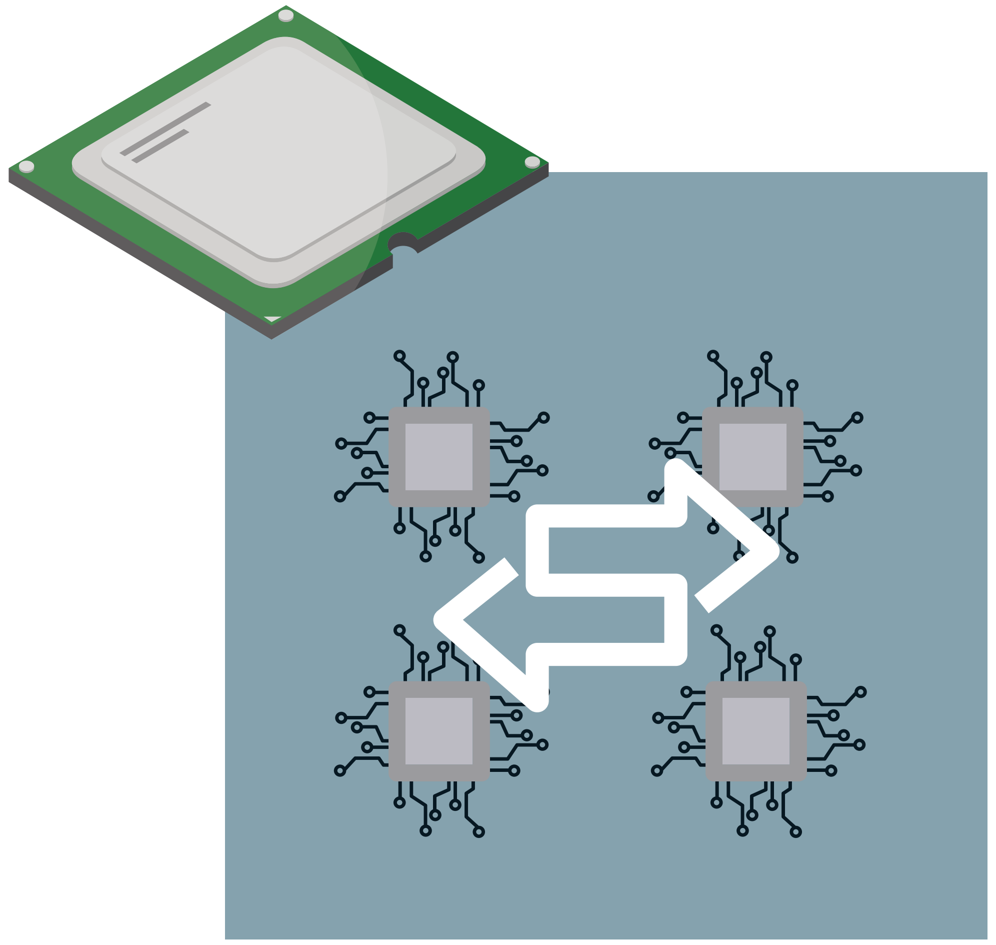
What if you want to use more processor that existed one ?
Simply an error will occurred

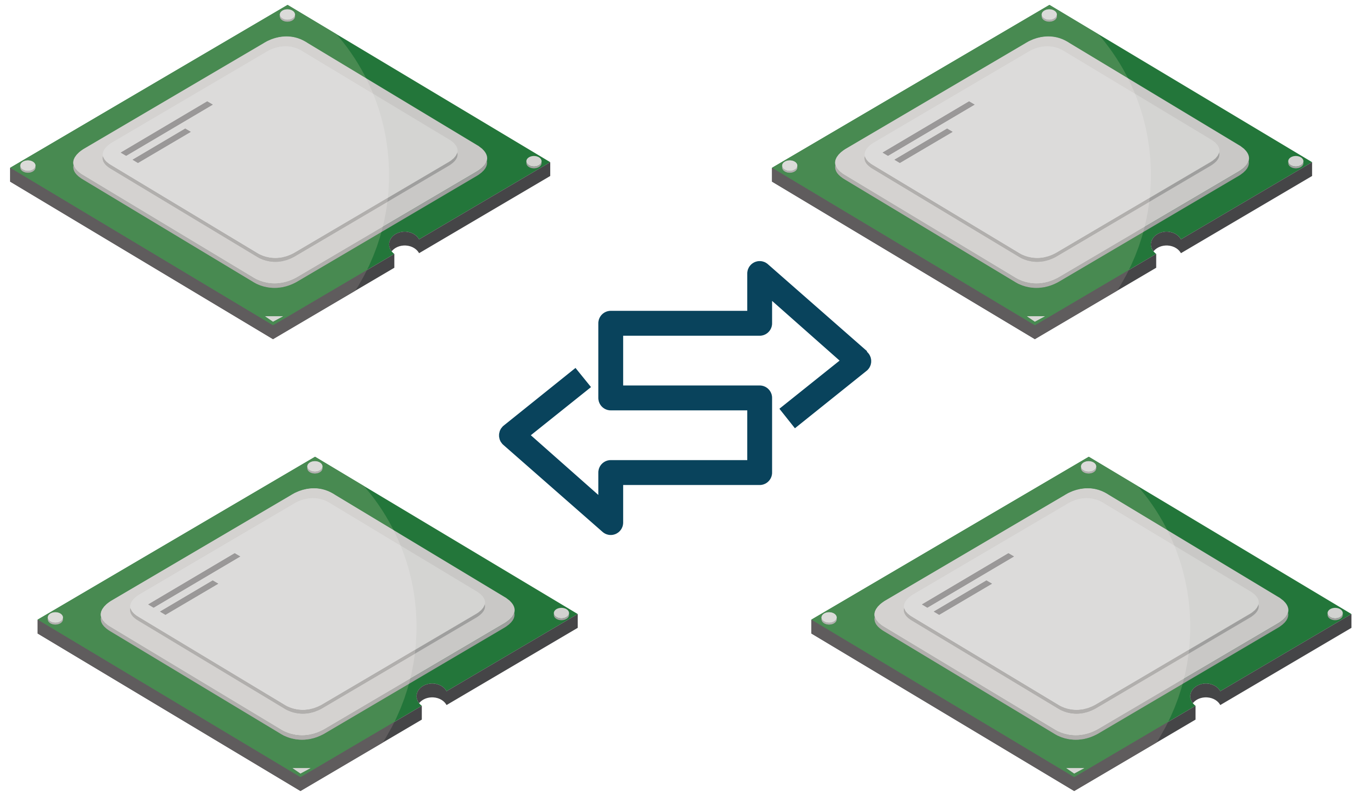


```
buggy@buggy-PC:~$ mpiexec -n 8 hostname
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 8
slots that were requested by the application:

  hostname

Either request fewer slots for your application, or make more slots
available for use.

A "slot" is the Open MPI term for an allocatable unit where we can
launch a process.  The number of slots available are defined by the
environment in which Open MPI processes are run:

  1. Hostfile, via "slots=N" clauses (N defaults to number of
     processor cores if not provided)
  2. The --host command line parameter, via a ":N" suffix on the
     hostname (N defaults to 1 if not provided)
  3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
  4. If none of a hostfile, the --host command line parameter, or an
     RM is present, Open MPI defaults to the number of processor cores

In all the above cases, if you want Open MPI to default to the number
of hardware threads instead of the number of processor cores, use the
--use-hwthread-cpus option.

Alternatively, you can use the --oversubscribe option to ignore the
number of available slots when deciding the number of processes to
launch.
--------------------------------------------------------------------------
buggy@buggy-PC:~$
```

2.3 So....

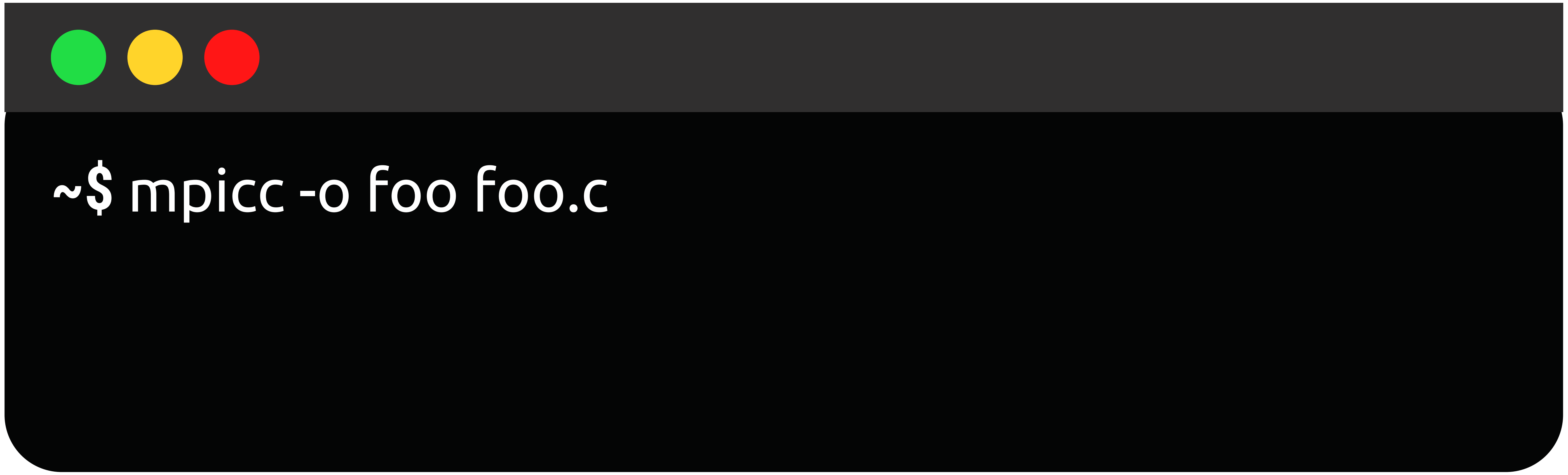
- The real power of MPI is being able to use multiple computers.
- Easiest if those computers can be access without needing a password (eg. via SSH with keys).





3. Compiling:

- Compiling a program for MPI is almost just like compiling a regular C or C++ program.



```
~$ mpicc -o foo foo.c
```

- The C compiler is mpicc and the C++ compiler is mpic++.

4. What MPI consists of :

- a header file mpi.h
- a library of routines and functions
- a runtime system.

```c
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] )
{

int rank;
int number_of_processes;
MPI_Init( &argc , &argv );

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

```
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] )
{

int rank;
int number_of_processes;
MPI_Init( &argc , &argv );

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

A c language library used to include all input/output binary functions after being preprocessed

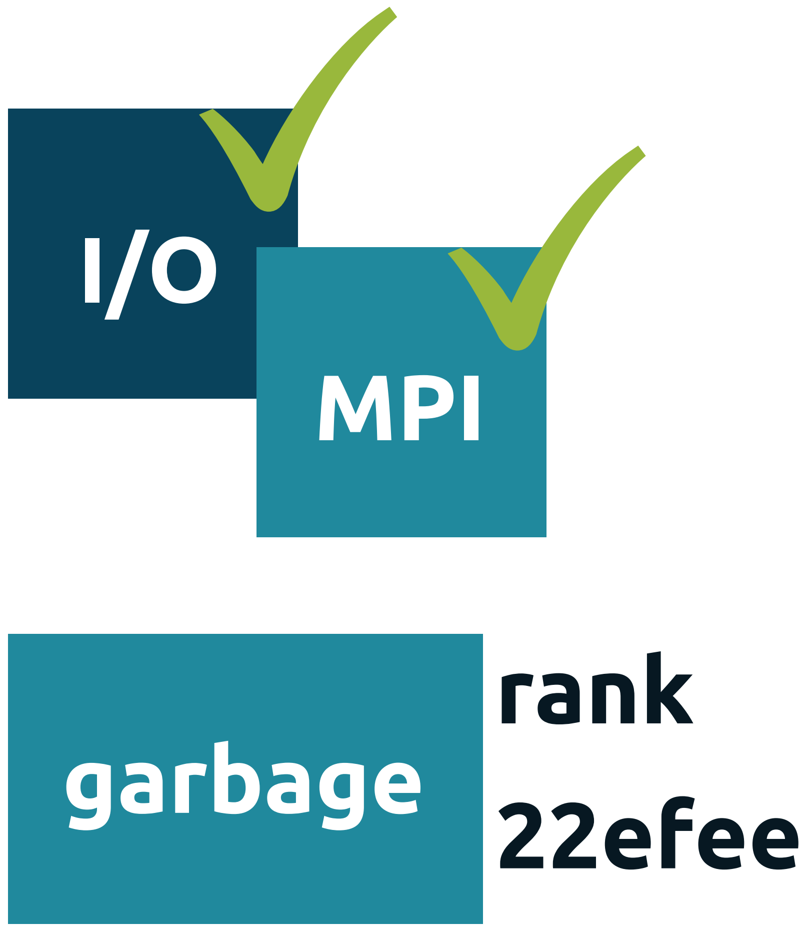
I/O

```
#include <stdio.h>
#include <mpi.h>
int main( int argc ,    ar *argv[] )
{

int rank;
int number_of_processes;
MPI_Init( &argc , &argv );

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

Message passing interface library that holds all c binary functions of passing-message. here where all MPI functionalities ships.

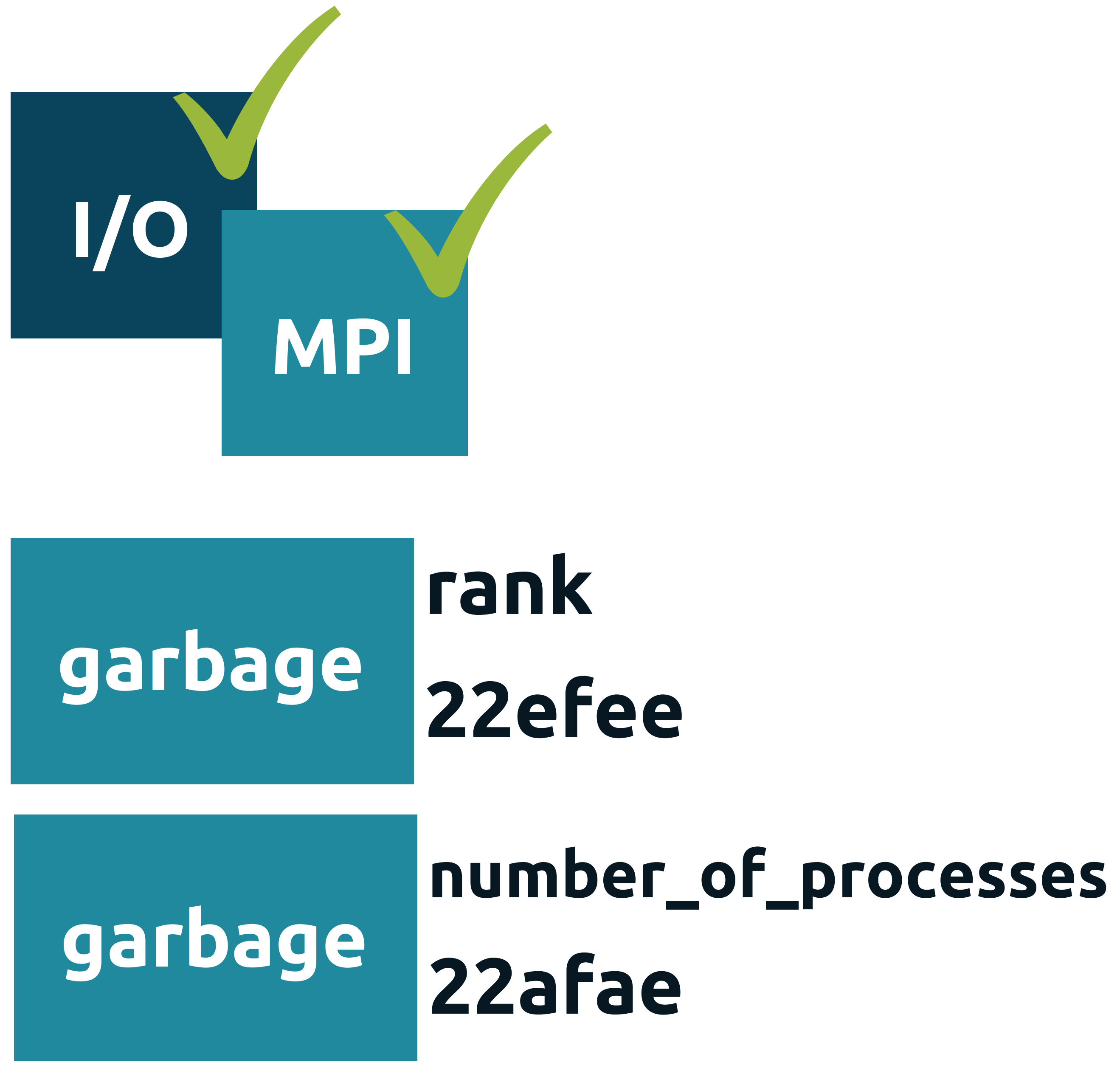
```c
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] )
{

int rank;
int number_of_processes;
MPI_Init( &argc , &argv );

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

declaration of an integer variable with the label (identifier) rank which used to store the process ID



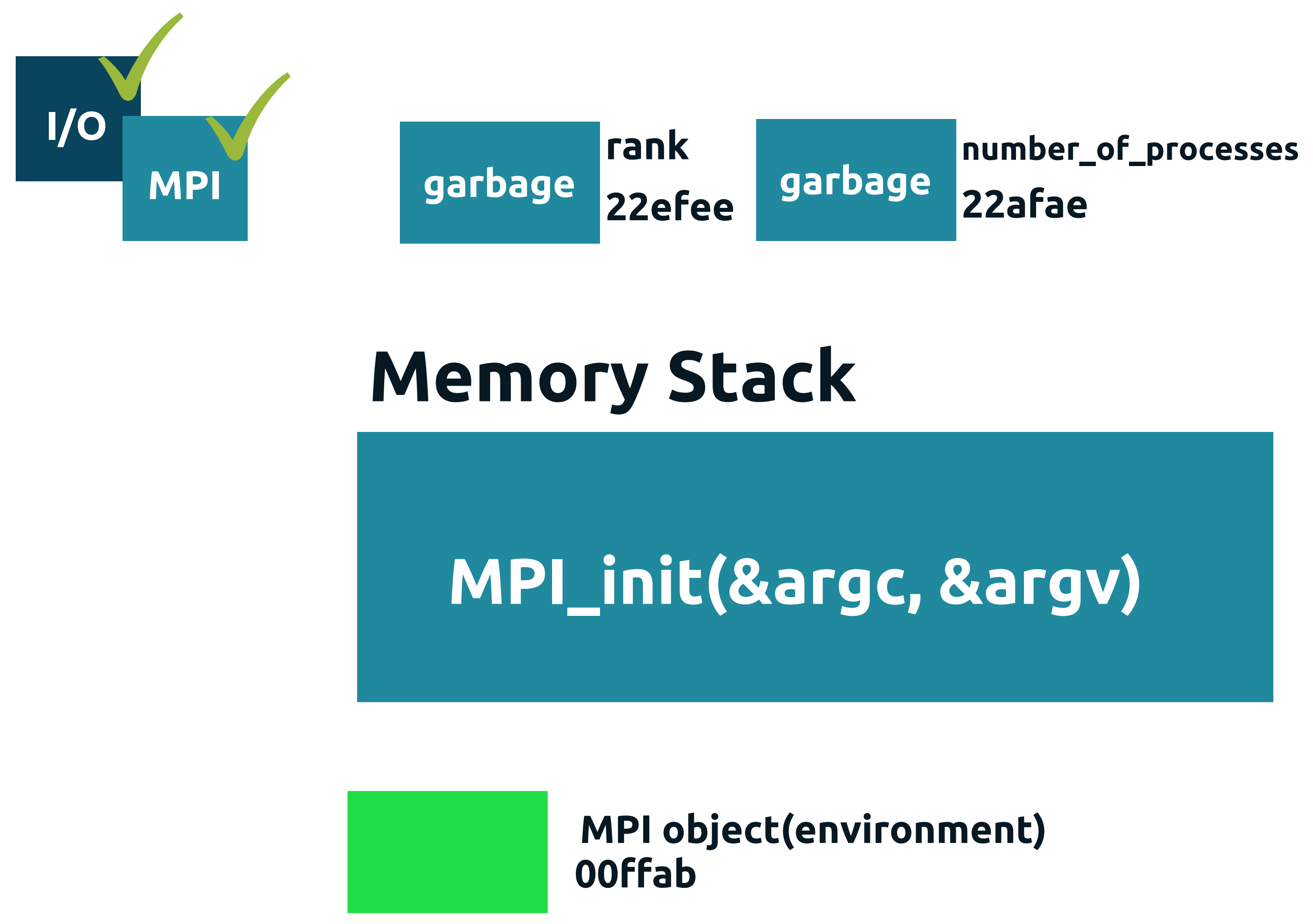
```c
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] )
{

int rank;
int number_of_processes;
MPI_Init( &argc , &argv );

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

declaration of an integer variable with the label (identifier) which used to store the number of all process available



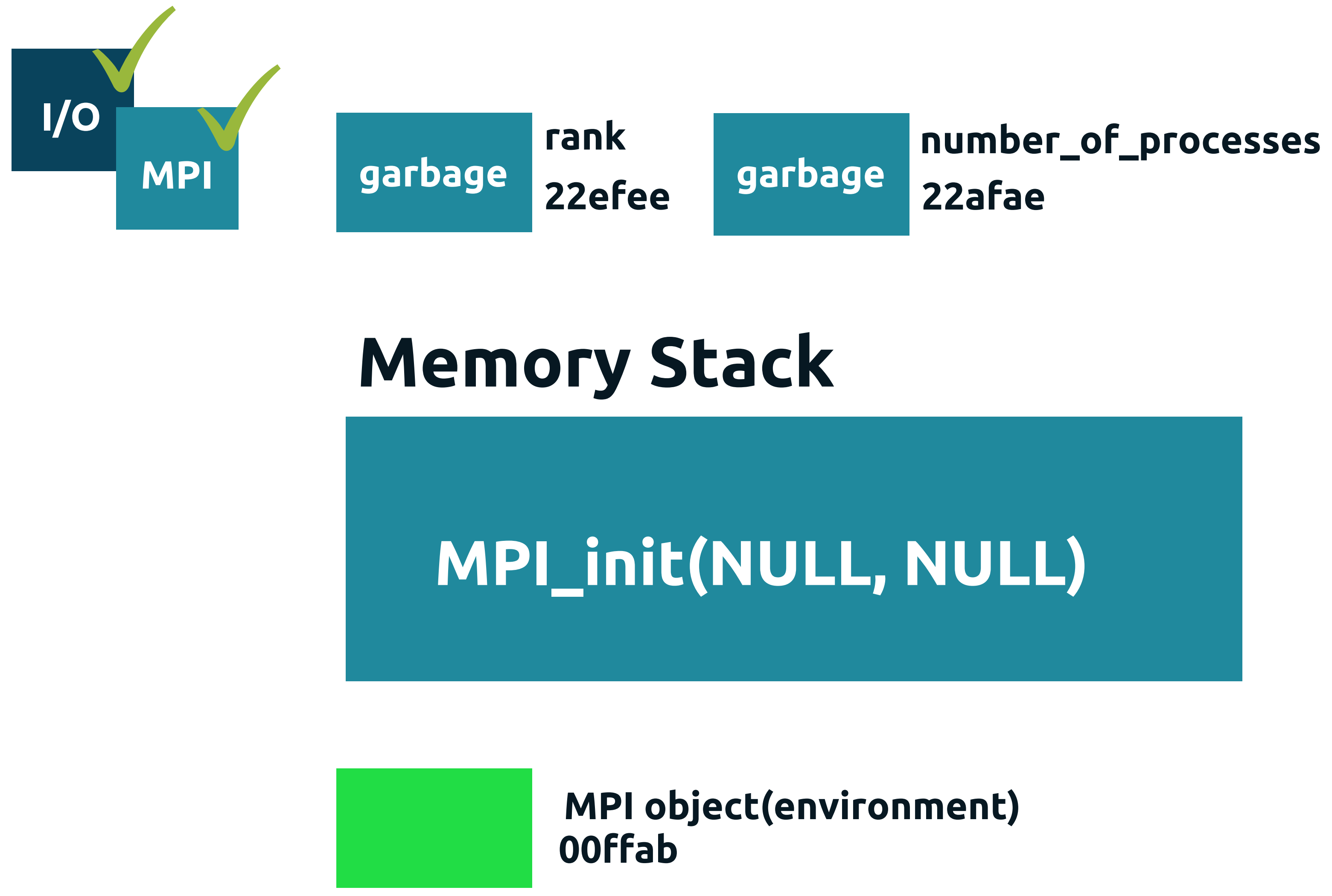
```c
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] )
{

int rank;
int number_of_processes;
MPI_Init( &argc , &argv );

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

MPI_init which used to store the process ID and all MPI's mechanisms



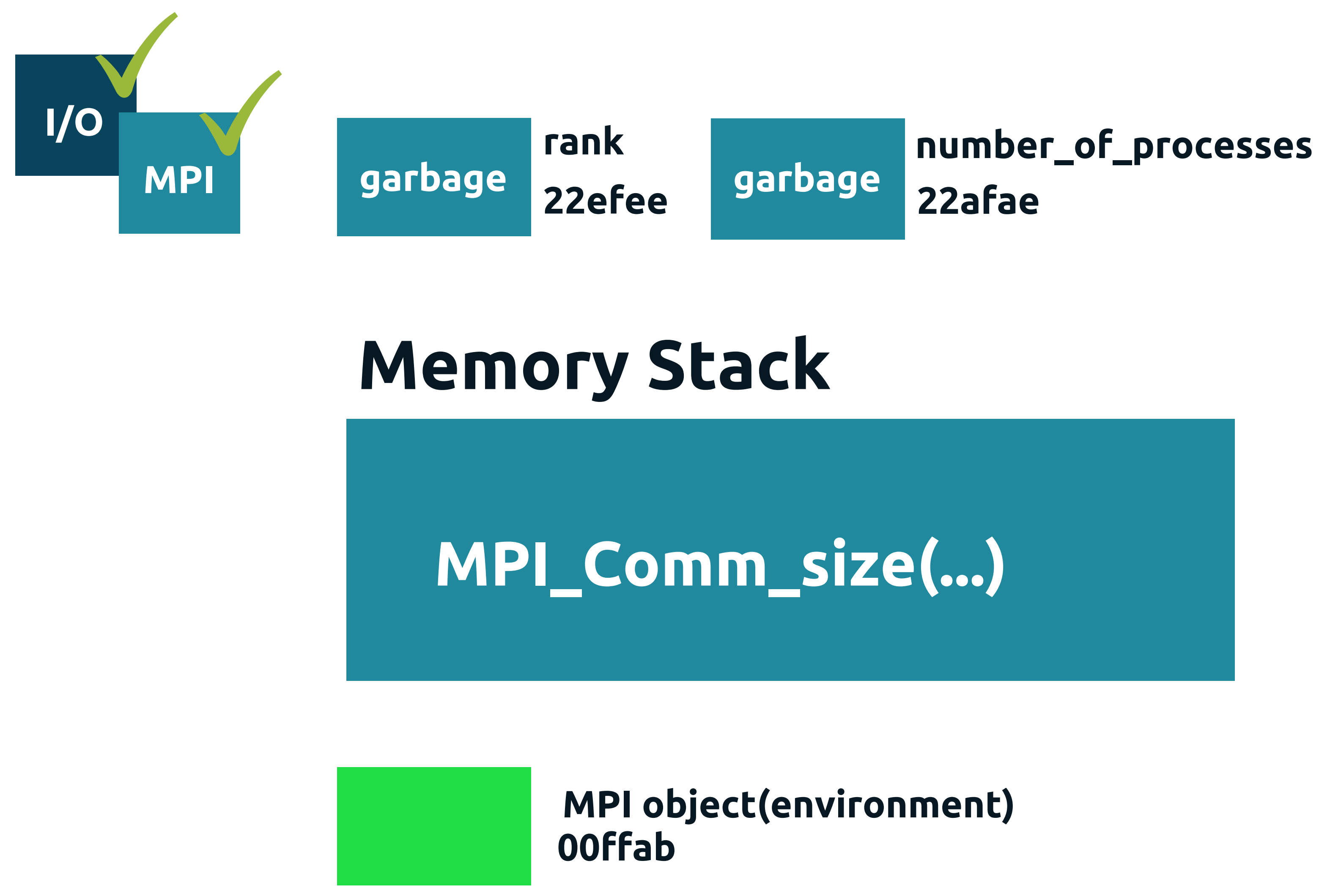
```c
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

MPI_init which used to store the process ID and all MPI's mechanisms



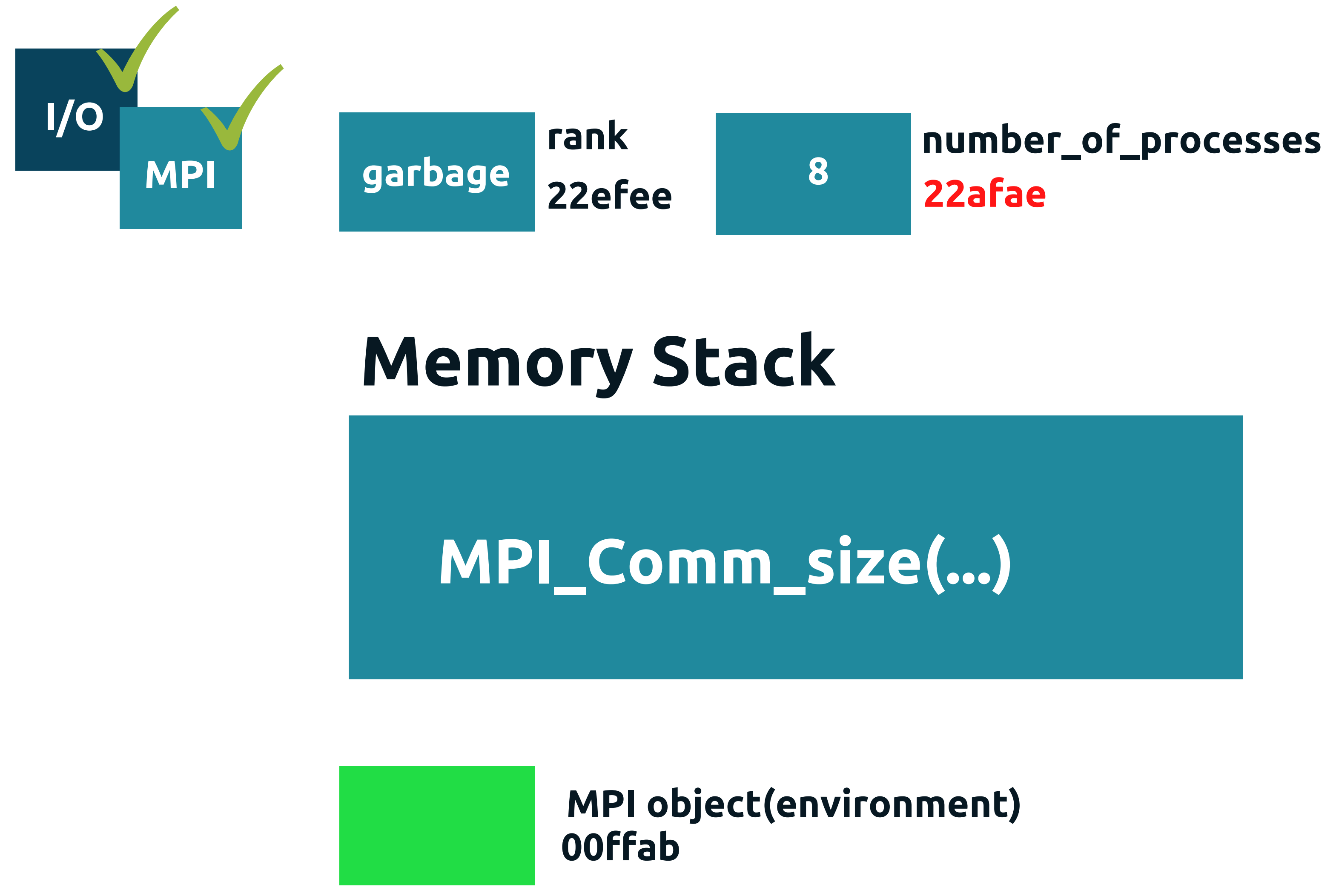
```c
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

gets the number of processes, Np



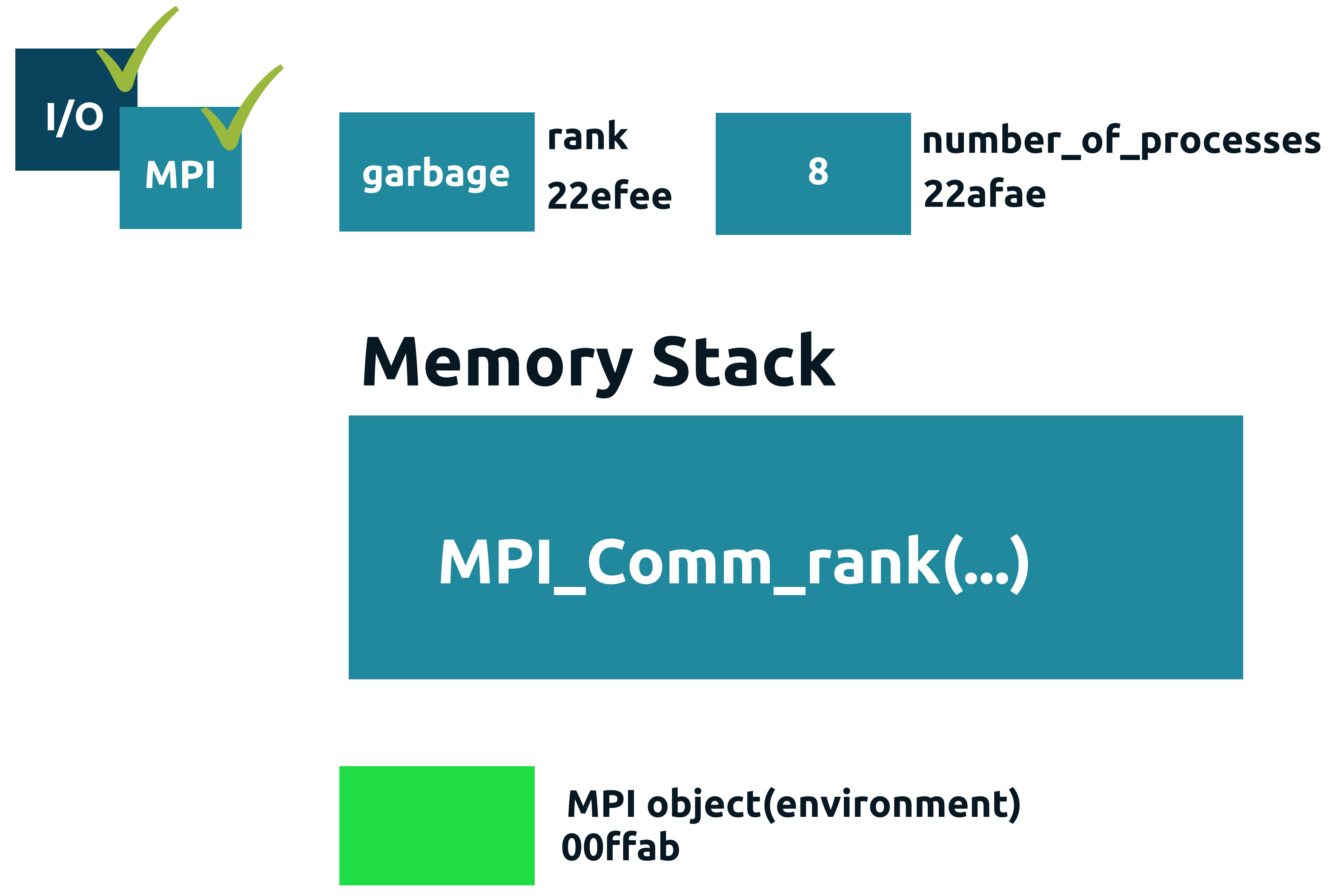
```c
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

gets the number of processes, Np



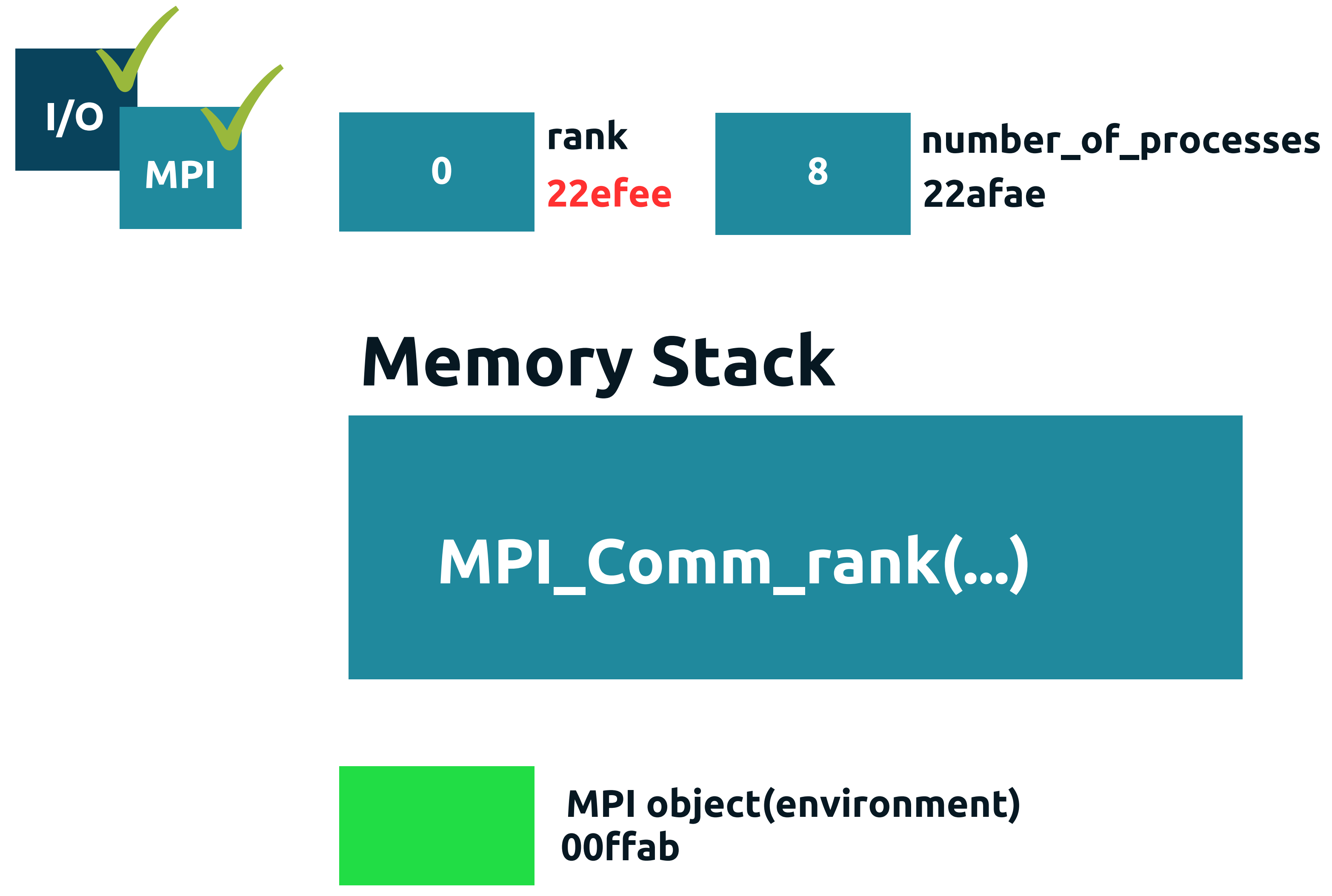
```c
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

gets the process ID of the current process which is between 0 and Np – 1, inclusive



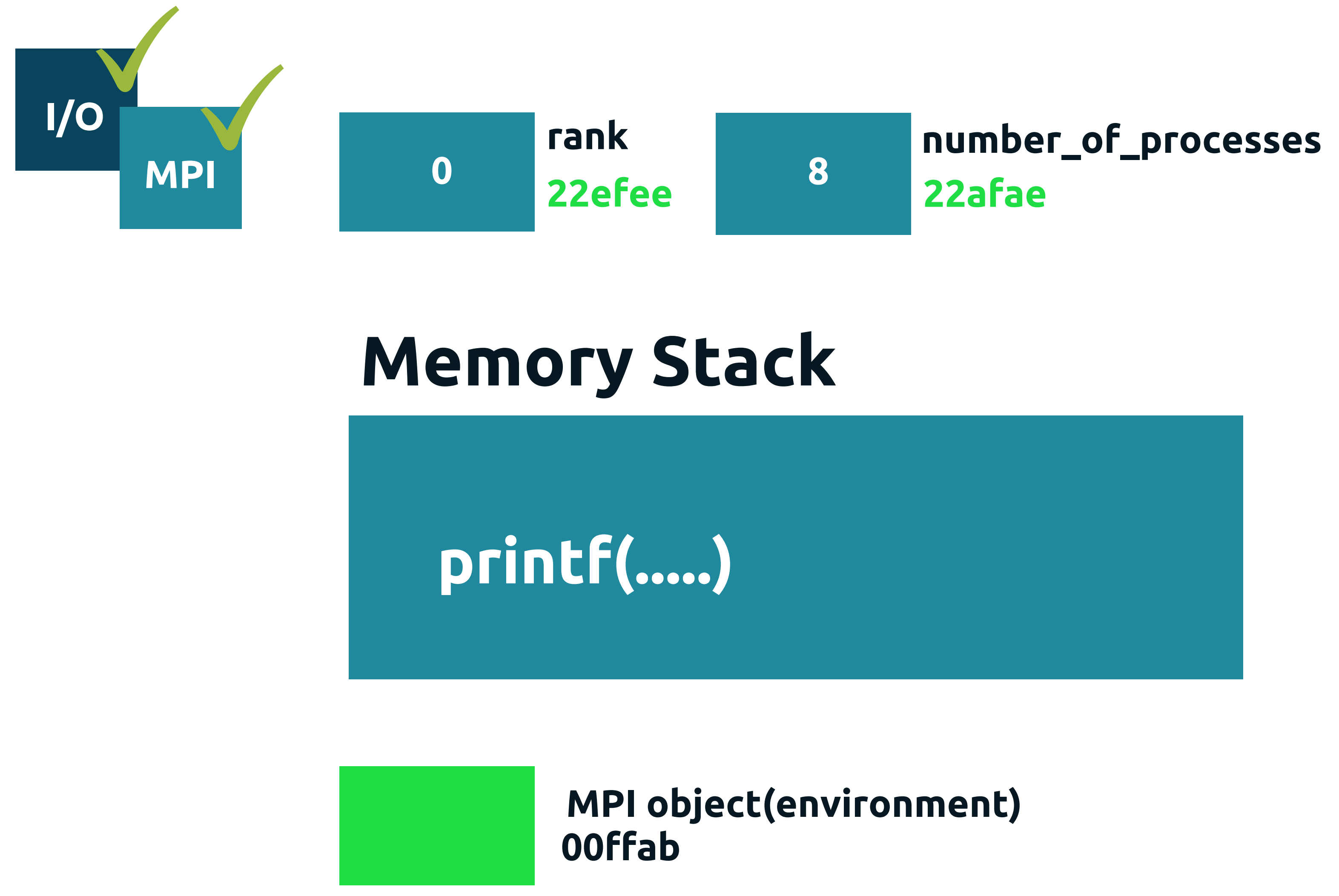
```
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

gets the process ID of the current process which is between 0 and Np – 1, inclusive



```
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

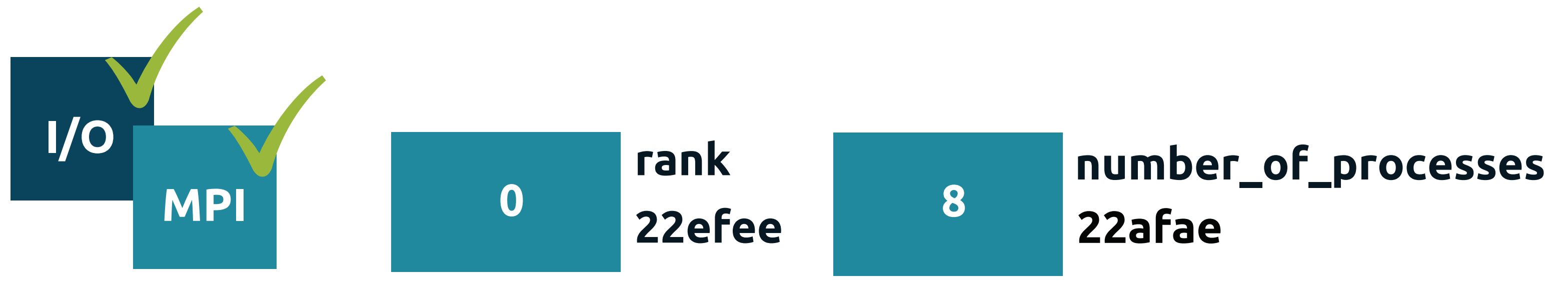
a format standard output function that used to print output data to user via the monitor using the default stream.



```
~ $hello from process 0 of 8
```

```c
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

shuts down the MPI runtime environment

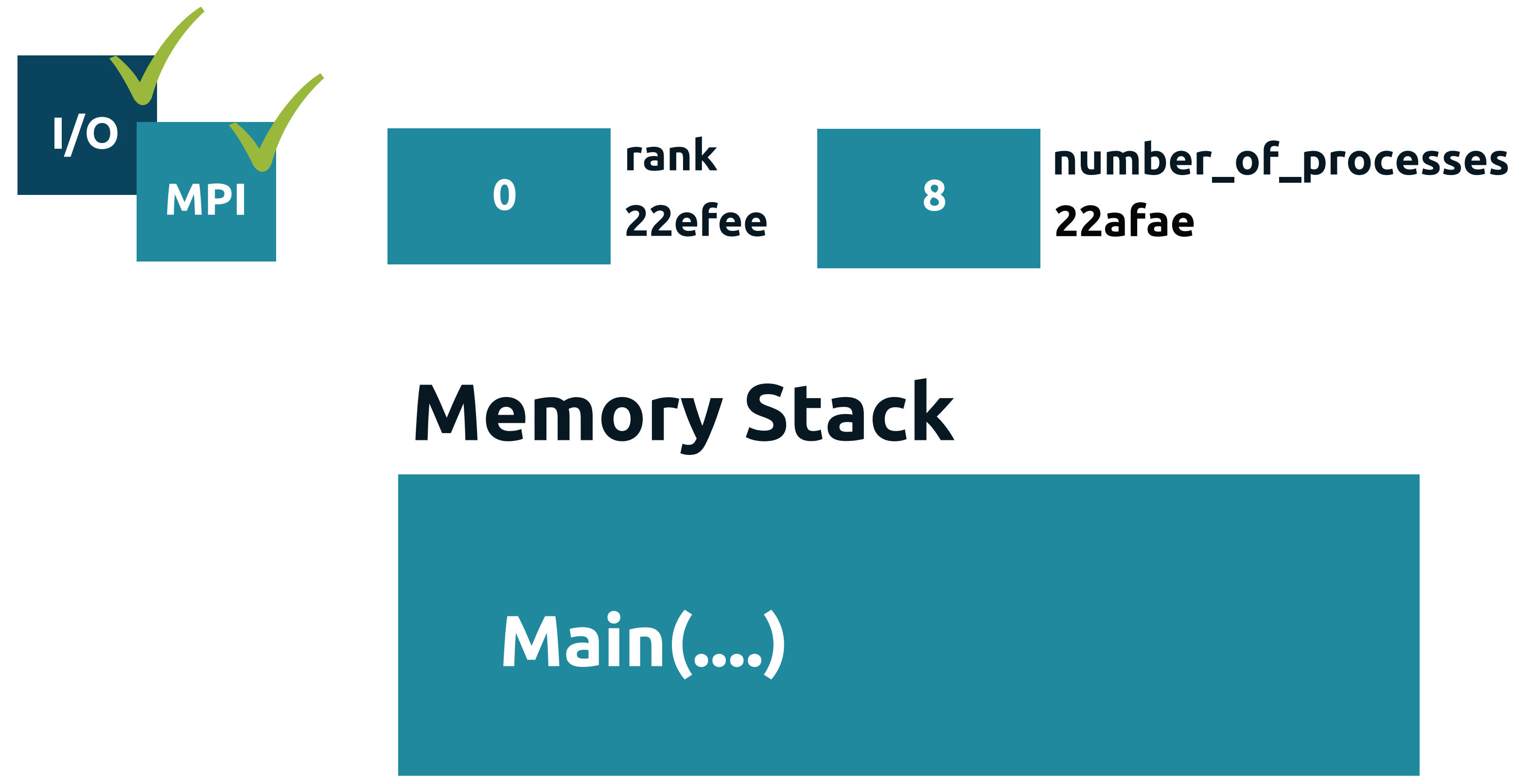


Memory Stack

MPI_Finalize(...)

```c
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

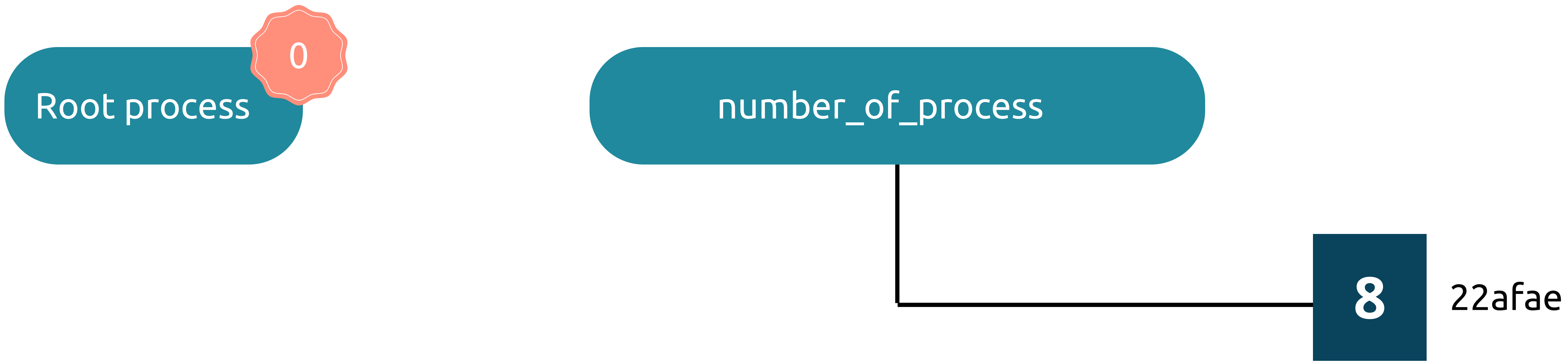
return 0 to operating system to indicate the successful termination of program

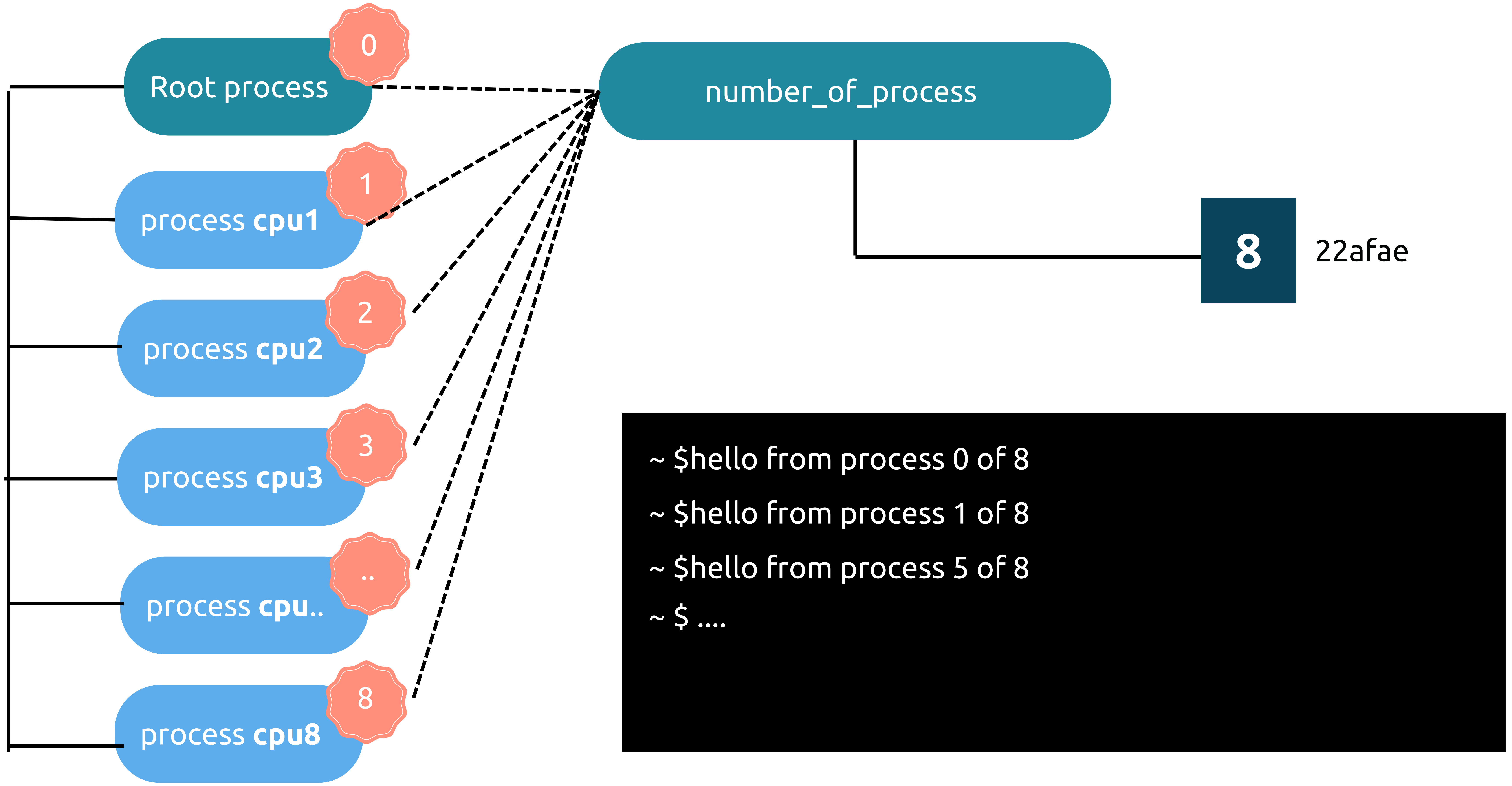


```
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int number_of_processes;
MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &number_of_processes );

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

printf("hello from process %d of %d \n", rank , number_of_processes);

MPI_Finalize();
return 0;
}
```

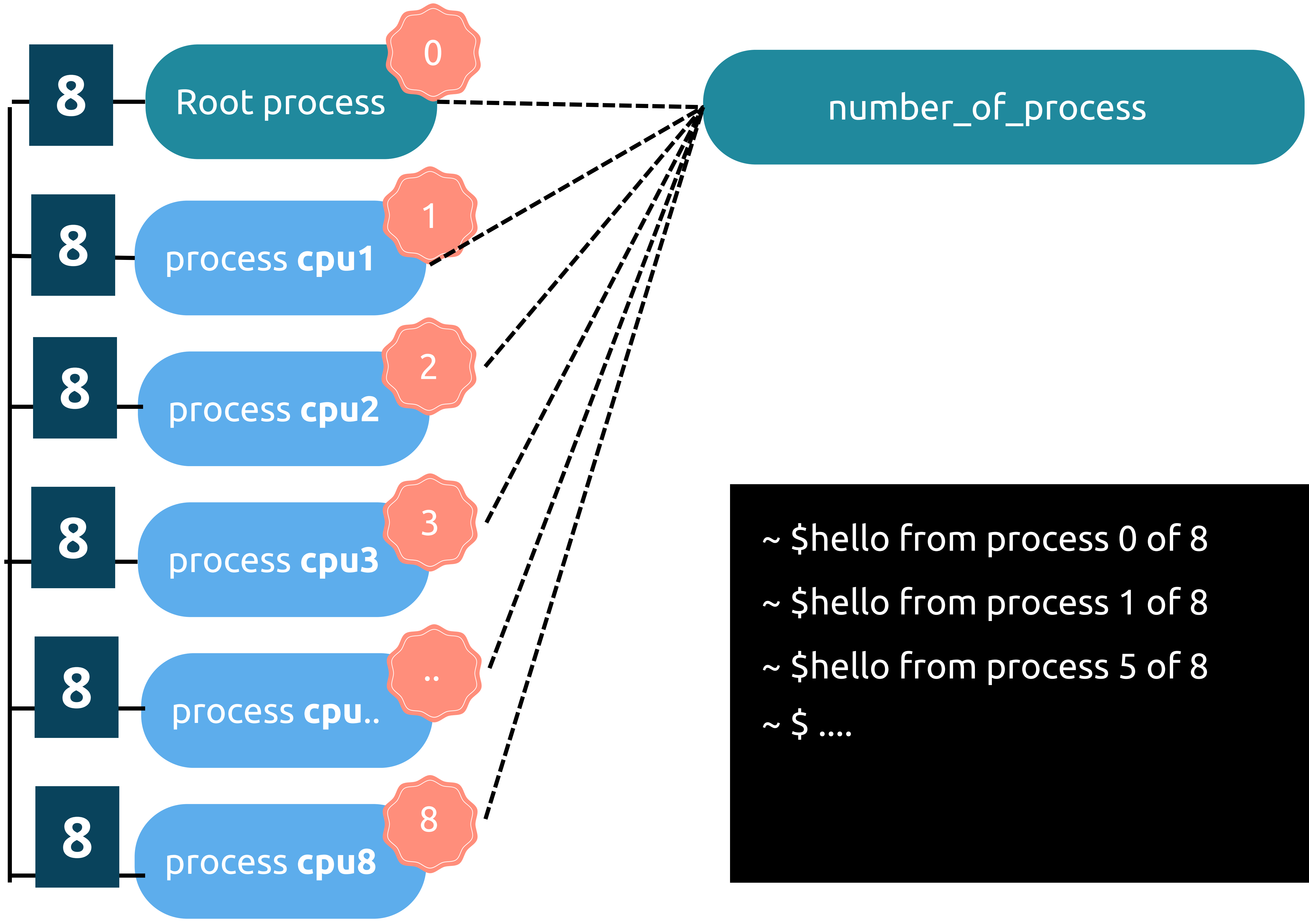
The closing coli braces to indelicate the end of the execution

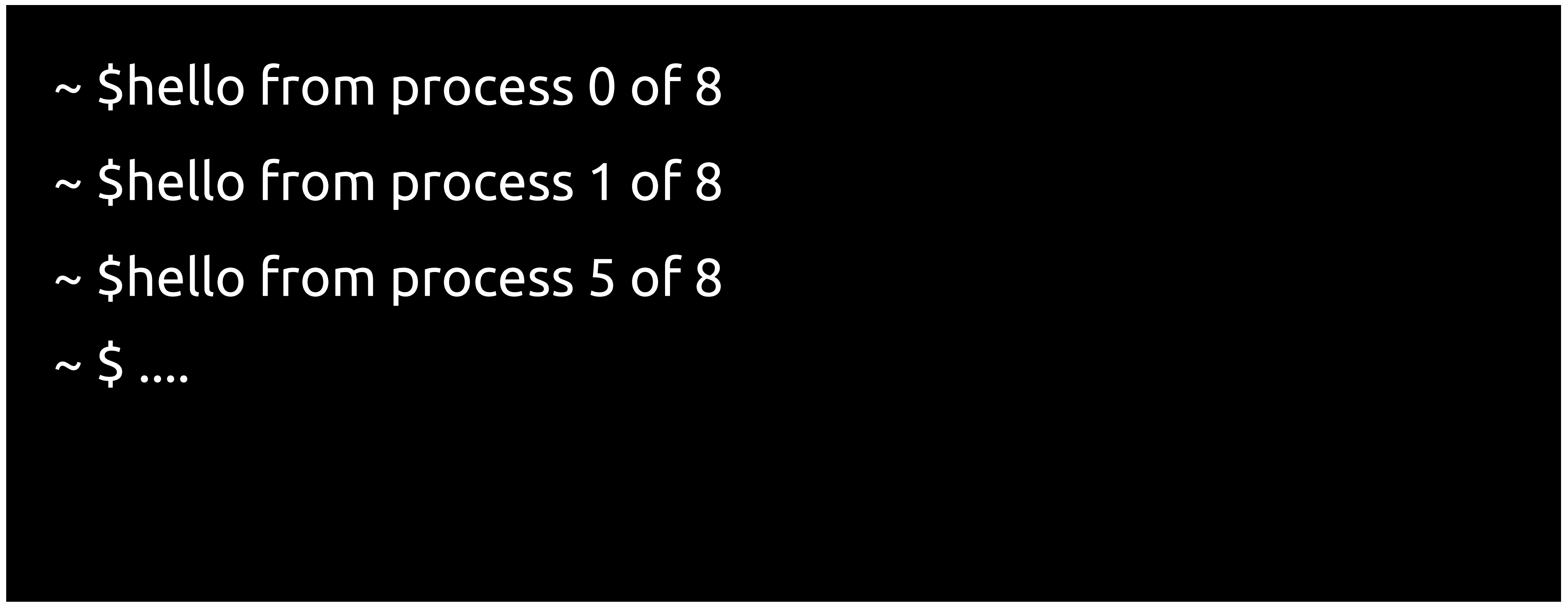
Memory finally clear


Root process
0
number_of_process
8
22afae



```
~ $hello from process 0 of 8
~ $hello from process 1 of 8
~ $hello from process 5 of 8
~ $ ....
```





```
~ $hello from process 0 of 8
~ $hello from process 1 of 8
~ $hello from process 5 of 8
~ $ ....
```

- Note:
  - All MPI processes (normally) run the same executable.
  - Each MPI process knows which rank it is.
  - Each MPI process knows how many processes are part of the same job.
  - The processes run in a non-deterministic order.

Executing examples

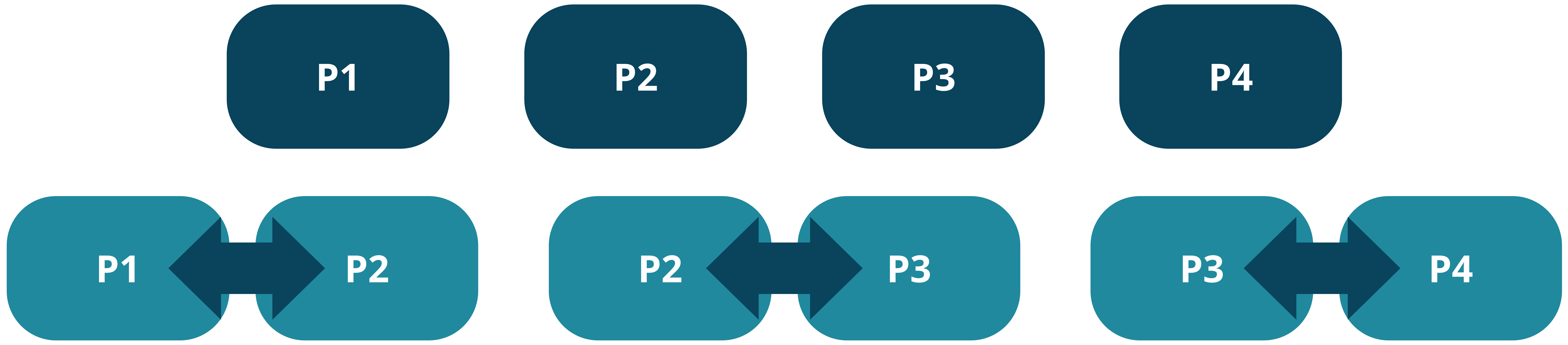
6. Point-to-point communication

Send() and Recv() functions

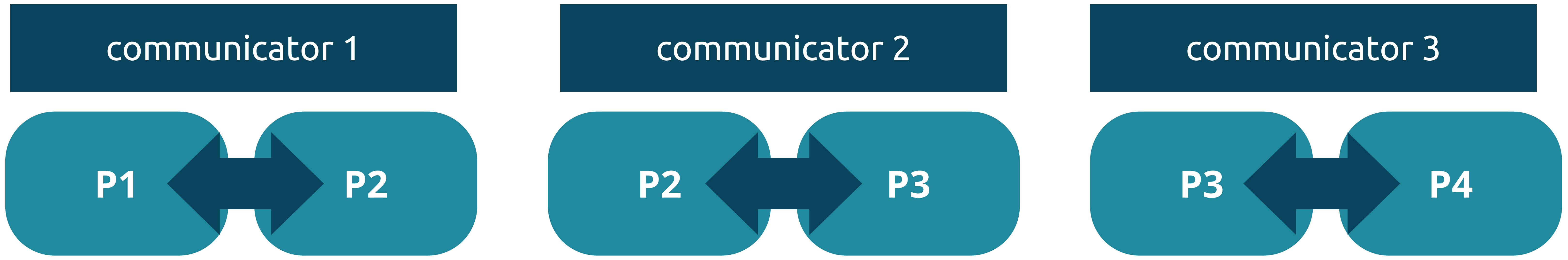
In message passing interface the important concept is the communication, The communication needs identifiers for both sender and receiver.

- Receiver must know from where to receiver the message (source)
- Sender must know the identification of whom receiving the message

This basic idea needs a way to organize process that communicate so there where communicators are introduced.



For each communication we have to define a communicator between process



Organize how processes communicate with each other.
A single communicator, **MPI_COMM_WORLD**, is created by **MPI_Init()** and all the processes running the program have access to it.

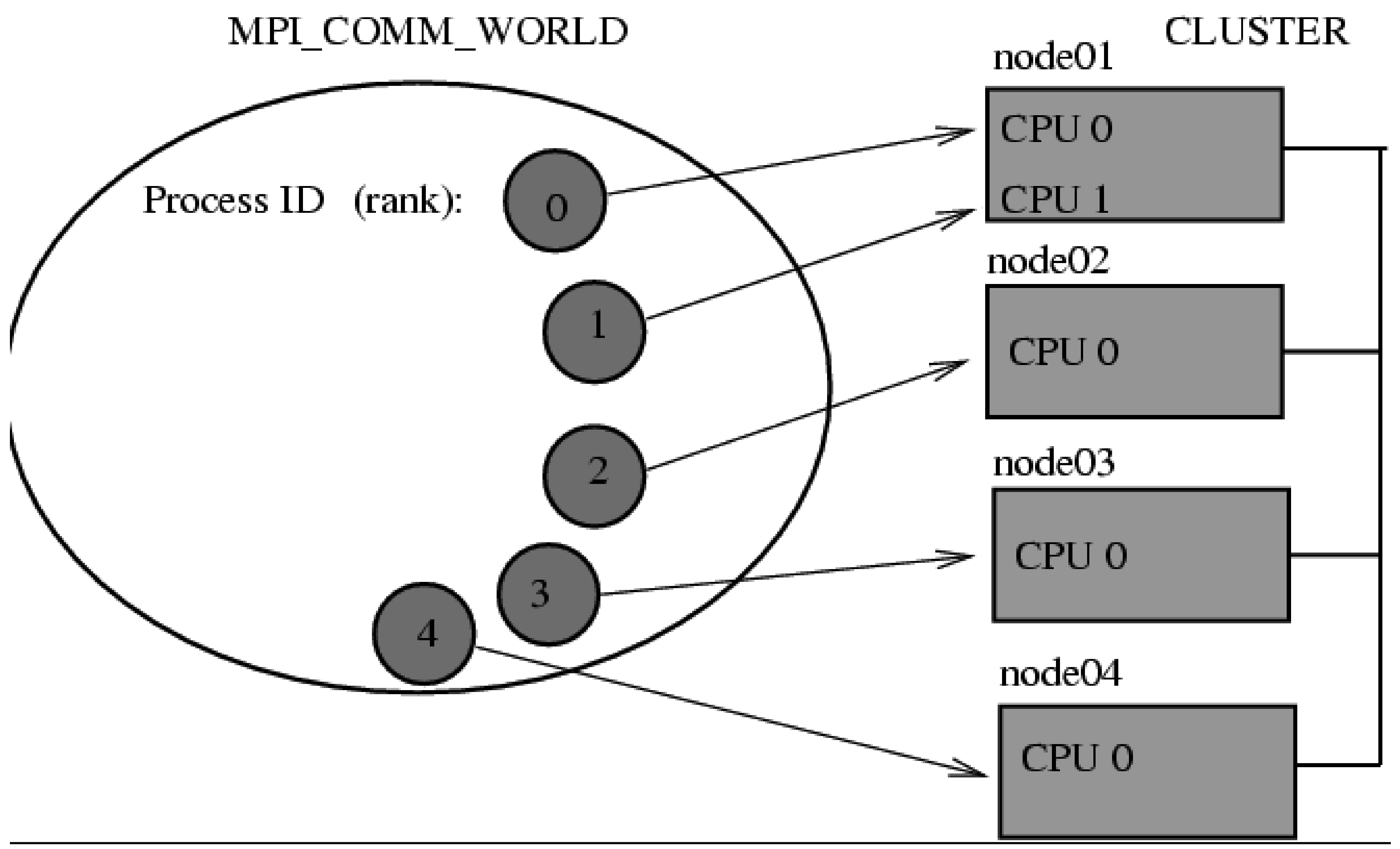


figure - https://www.codingame.com/

6. Point-to-point communication

Send() and Recv() functions

MPI provides two main routines for point-to-point communication

MPI_Send() – Send to a message to another process
MPI_Recv() – Receive a message from another process

Both of these have several variants that we'll mention here and see some of later.

6.1 Send()

Function Parameters

The calling sequence for MPI_Send() is

```
int MPI_Send (
void * buf , /* pointer to send buffer */
int count , /* number of items to send */
MPI_Datatype datatype , /* datatype of buffer elements */
int dest , /* rank of destination process */
int tag , /* message type identifier */
MPI_Comm comm ) /* MPI communicator to use */
```

6.1 Send()

Function Returns

- Nearly all MPI functions return an integer status code:

MPI_SUCCESS if function completed without error, otherwise an error code is returned

6.1 Send()

Functionality

- The **MPI_Send()** function initiates a blocking send. Here **“blocking”** does not indicate that the sender waits for the message to be received, but rather that the sender waits for the message to be accepted by the MPI system.

- It does mean that once this function returns the send buffer may be changed with out impacting the send operation.

6.2 Recv()

Function parameters

The calling sequence for MPI_Recv() is

```
int MPI_Recv (
void * buf , /* pointer to send buffer */
int count , /* number of items to send */
MPI_Datatype datatype , /* datatype of buffer elements */
int source , /* rank of sending process */
int tag , /* message type identifier */
MPI_Comm comm , /* MPI communicator to use */
MPI_Status * status ) /* MPI status object */
```

6.2 Recv()

Function return

- Nearly all MPI functions return an integer status code:

MPI_SUCCESS if function completed without error, otherwise an error code is returned

6.2 Recv()

Functionality

The **MPI_Recv()** function initiates a blocking receive. It will not return to its caller until a message with the specified tag is received from the specified source.

MPI_ANY_SOURCE may be used to indicate the message should be accepted from any source.
MPI_ANY_TAG may be used to indicate the message should be accepted regardless of its tag.

P0

```
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int nb;
MPI_status status;

MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &nb);

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

if(rank == 0)
{
   int number;
   number = 10;
   MPI_Send(&number, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
}else
{
  int received;
  MPI_Recv(&received,1,MPI_INT, 0,1,MPI_COMM_WORLD,&status);
}

MPI_Finalize();
return 0;
}
```

P1

```
#include <stdio.h>
#include <mpi.h>
int main( )
{

int rank;
int nb;
MPI_status status;

MPI_Init( NULL , NULL);

MPI_Comm_size( MPI_COMM_WORLD , &nb);

MPI_Comm_rank( MPI_COMM_WORLD , &rank );

if(rank == 0)
{
  int number;
  number = 10;
  MPI_Send(&number, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
}else
{

  int received;
  MPI_Recv(&received,1,MPI_INT, 0,1,MPI_COMM_WORLD,&status);

}

MPI_Finalize();
return 0;
}
```

7. grouping communication

MPI_Bcast() broadcasts a message from one process to all of the others.

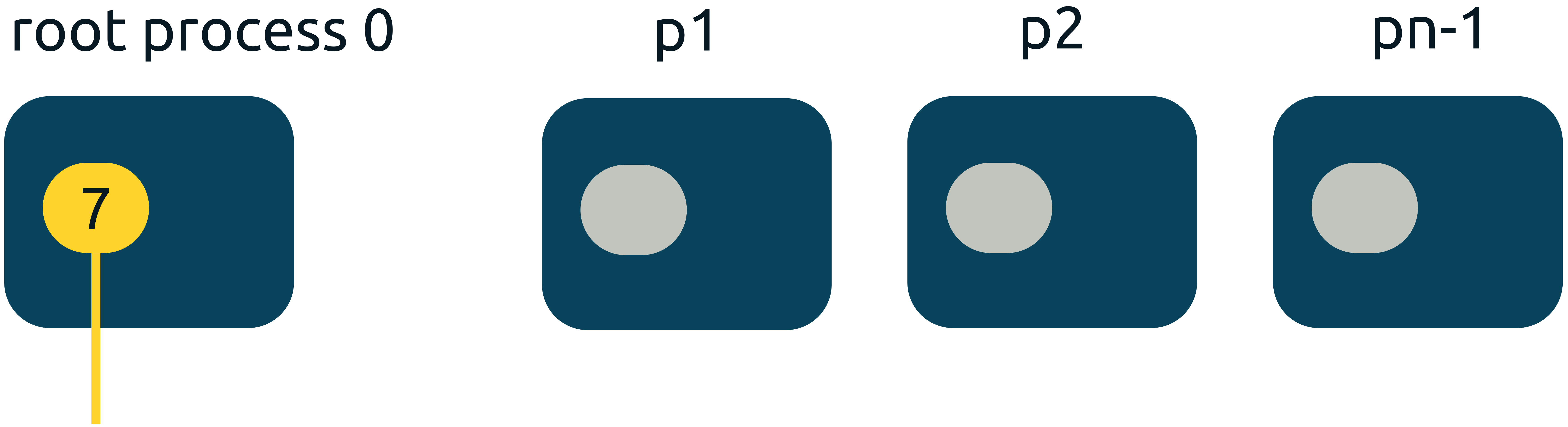
MPI_Reduce() performs a reduction (e.g. a global sum, maximum, etc.) of a variable in all processes, with the result ending up in a single process.

MPI_Allreduce() performs a reduction of a variable in all processes, with the result ending up in all processes.

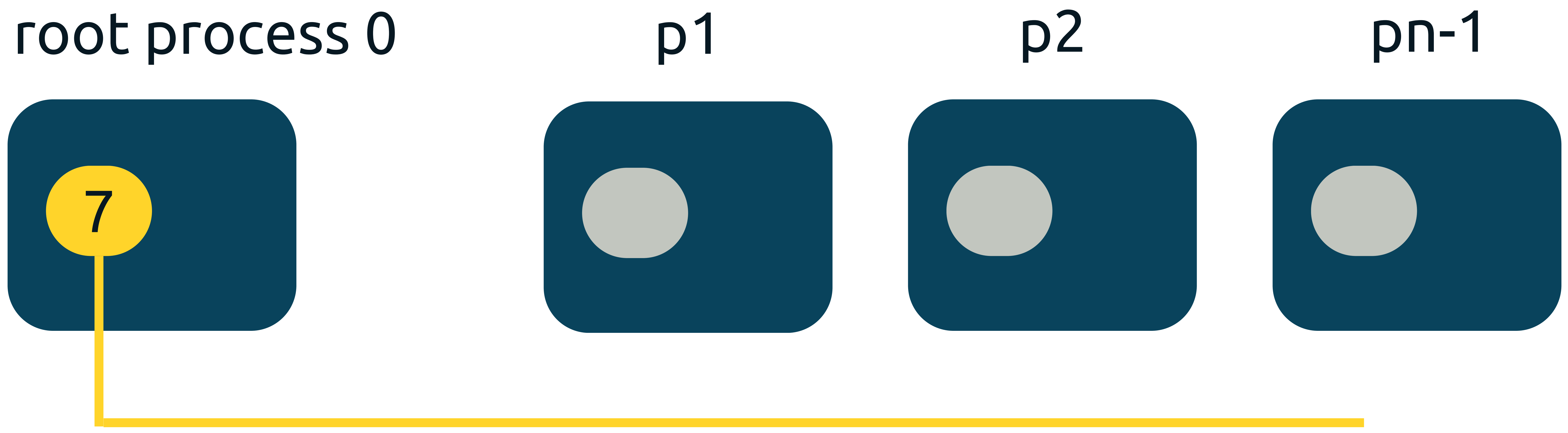
MPI_scatter() Spread all data from one process to many we call this scatter.

MPI_gather() It's the inverse operation of scatter where we grather data from many processes to one process called the root.

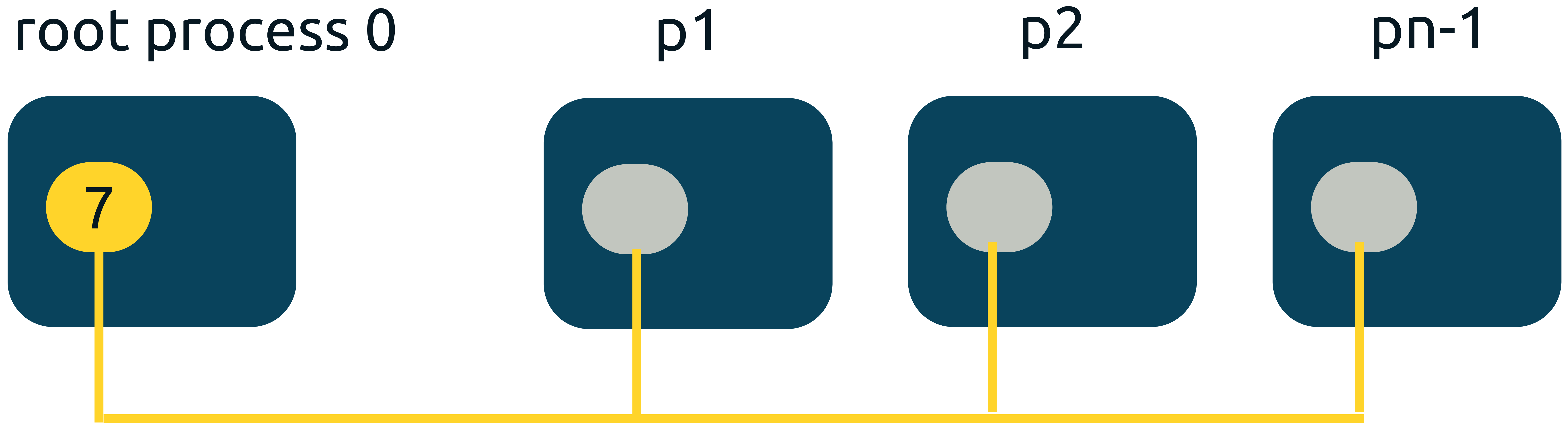
MPI_Bcast()



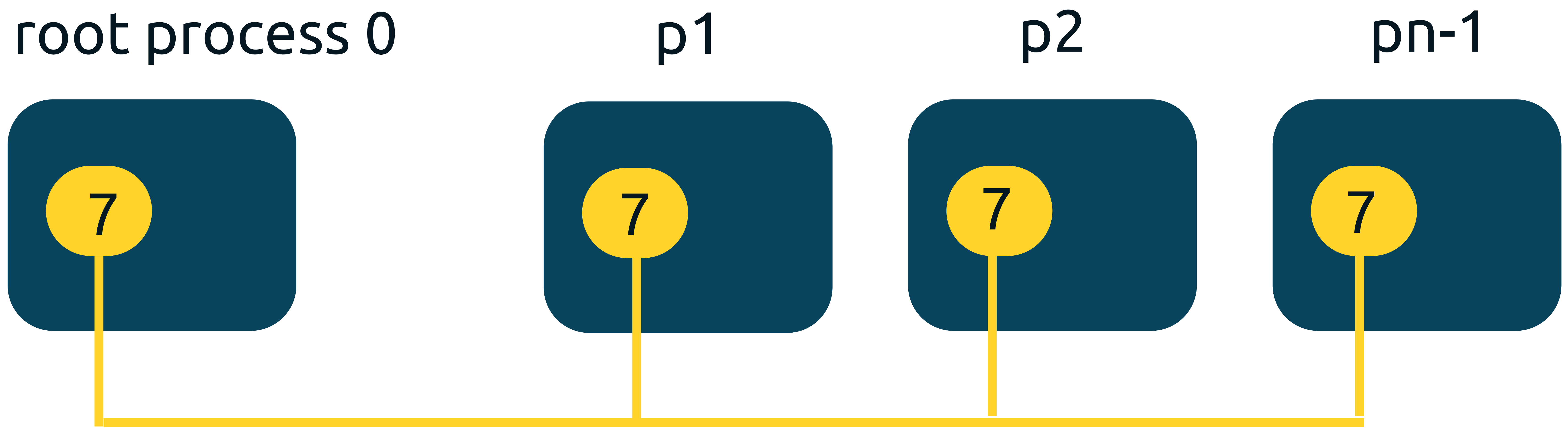
MPI_Bcast()



MPI_Bcast()



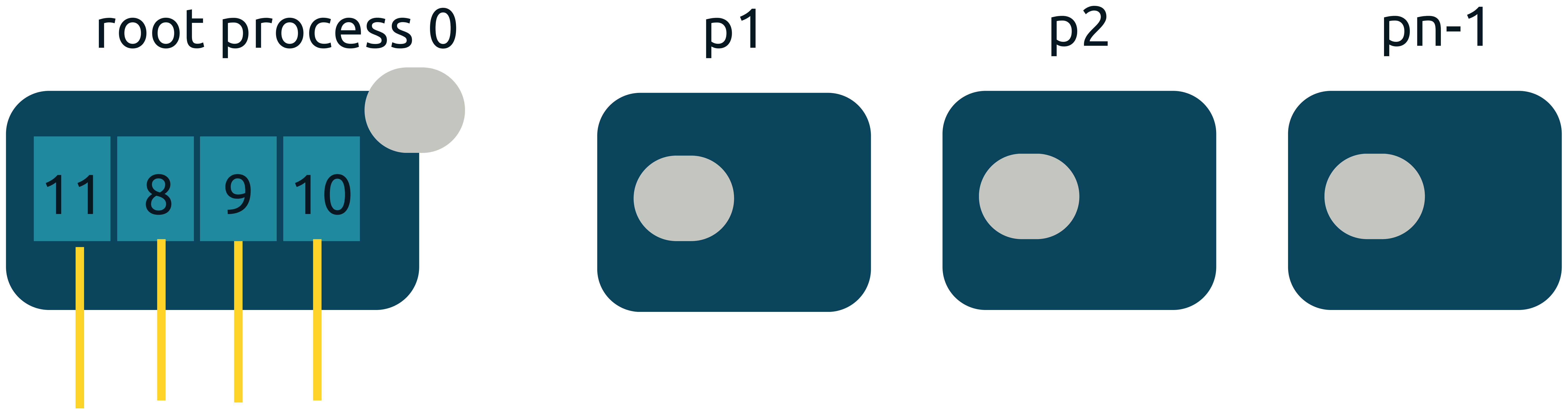
MPI_Bcast()



example

breaking a cipher using crypto analysis

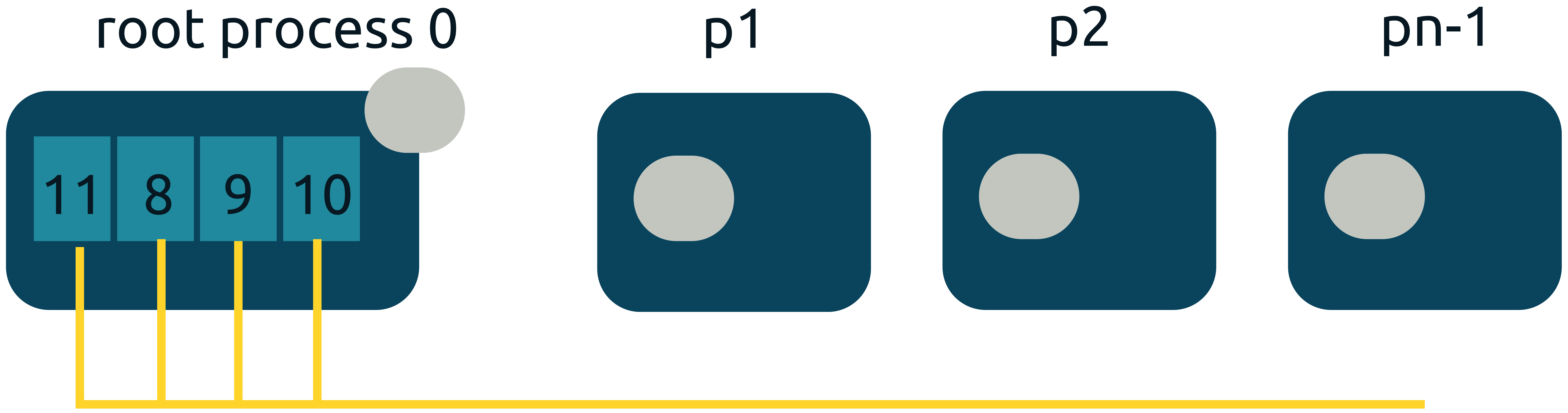
MPI_Scatter()



example

mean calculation of RSSI, WIC and ISI

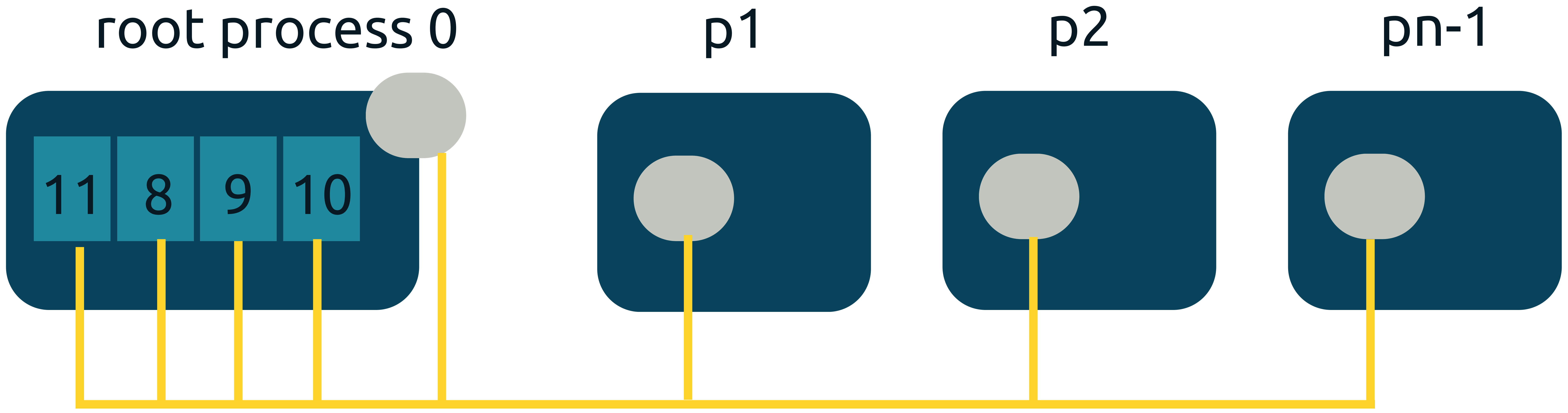
MPI_Scatter()



example

mean calculation of RSSI, WIC and ISI

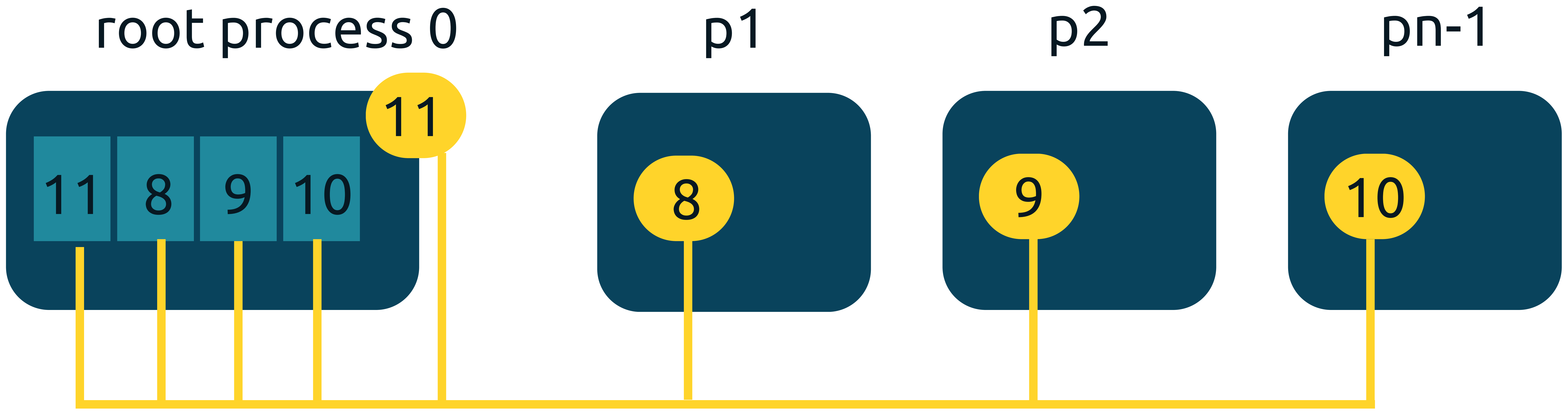
MPI_Scatter()



example

mean calculation of RSSI, WIC and ISI

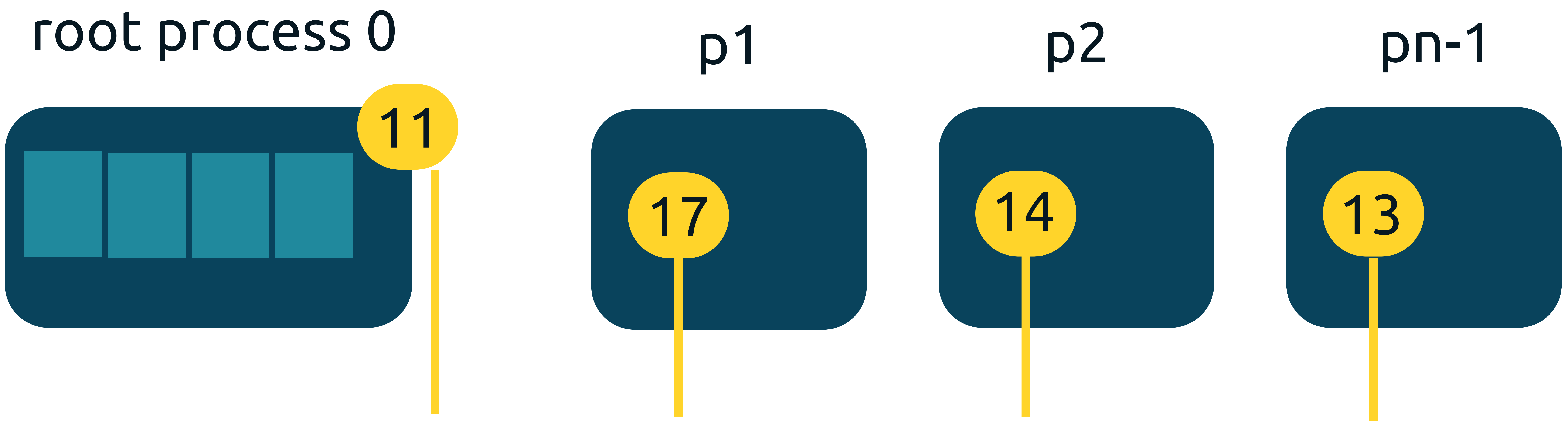
MPI_Scatter()



example

mean calculation of RSSI, WIC and ISI, SS

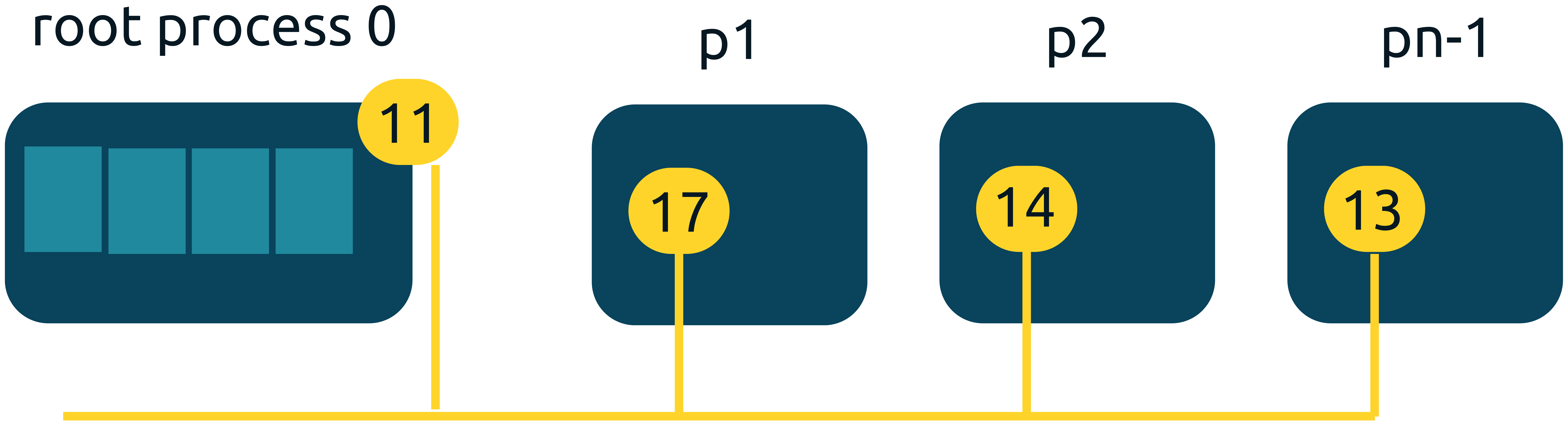
MPI_Gather()



example

The calculation of the max of sections means

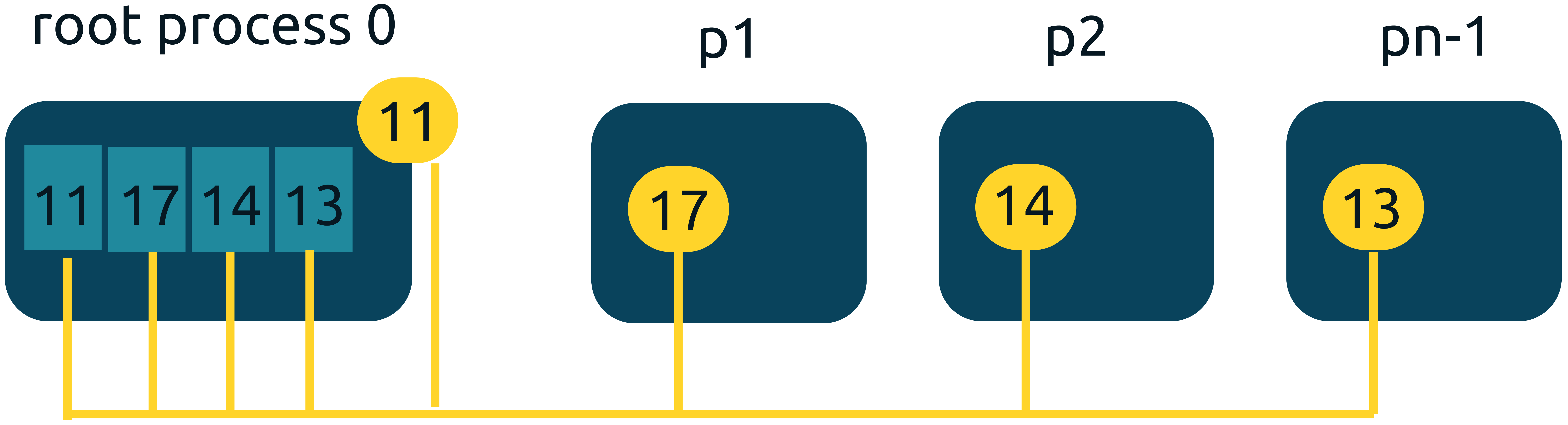
MPI_Gather()



example

The calculation of the max of sections means

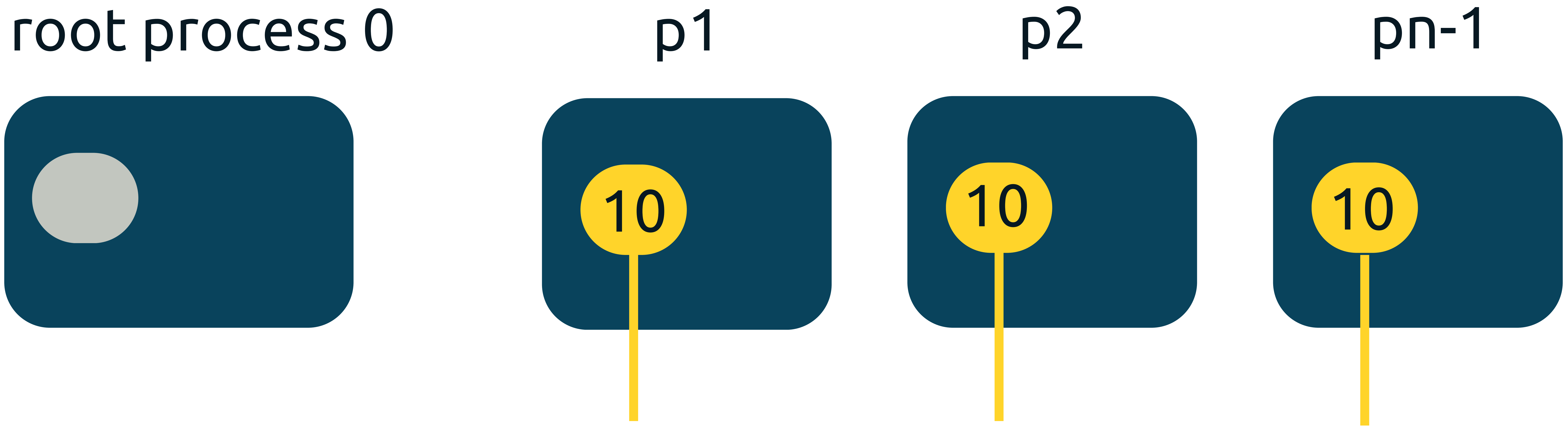
MPI_Gather()



example

The calculation of the max of sections means

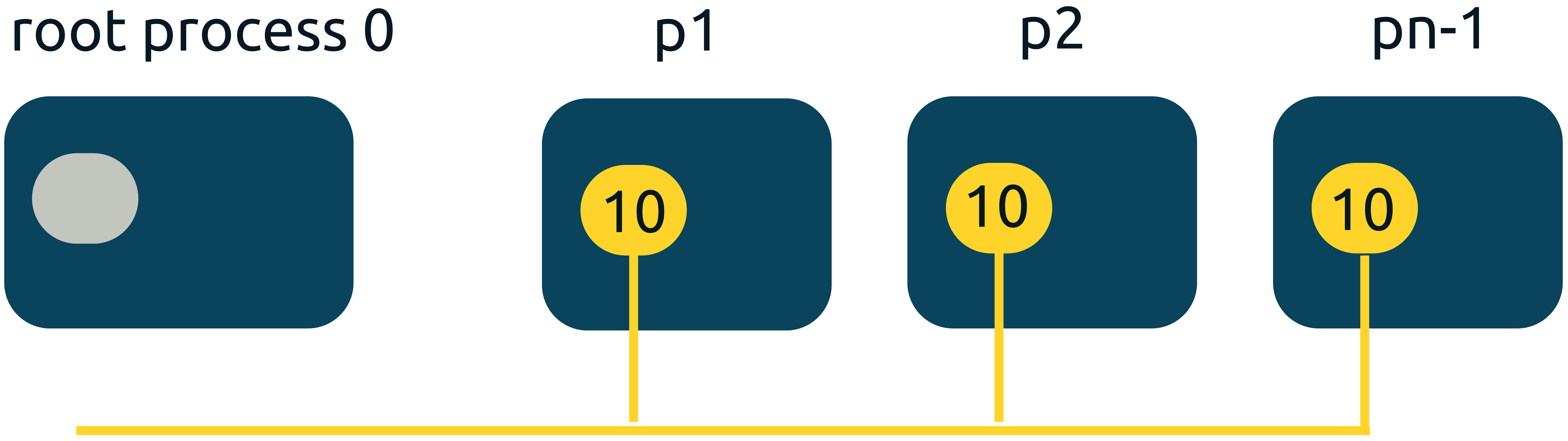
MPI_Reduce()



example

perform an operation of all other resolve values

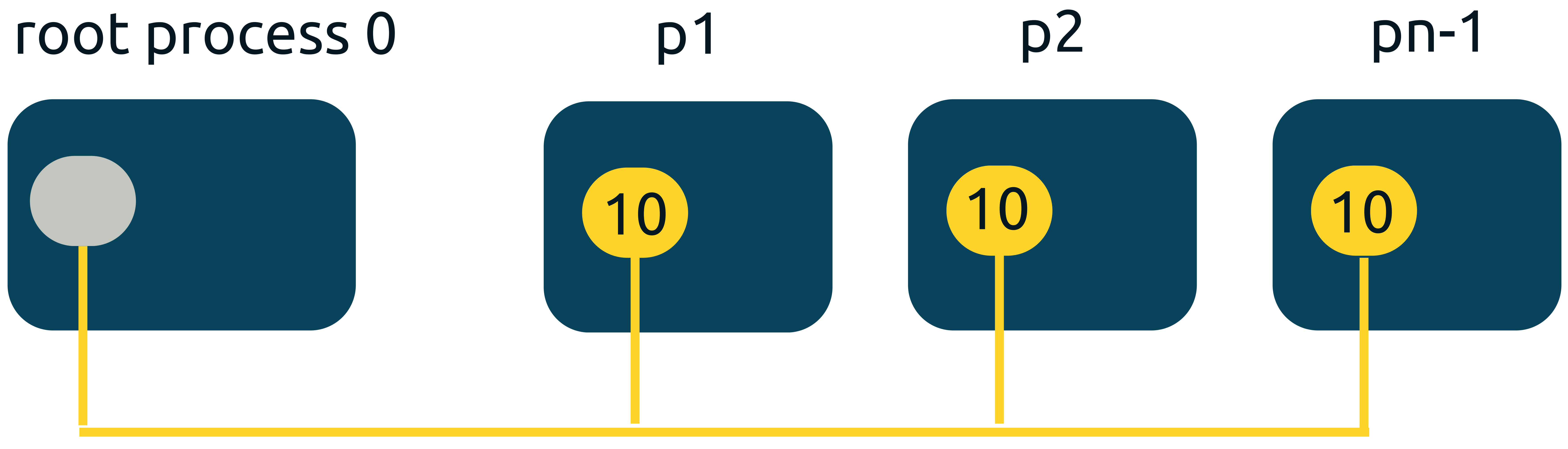
MPI_Reduce()



example

perform an operation of all other resolve values

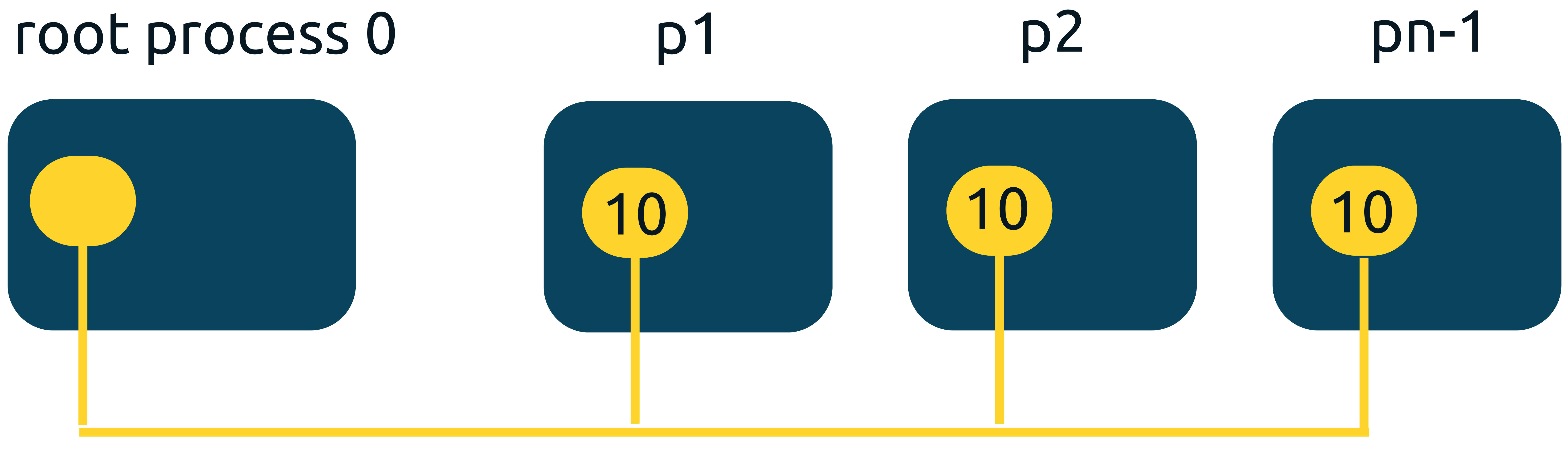
MPI_Reduce()



example

perform an operation of all other resolve values

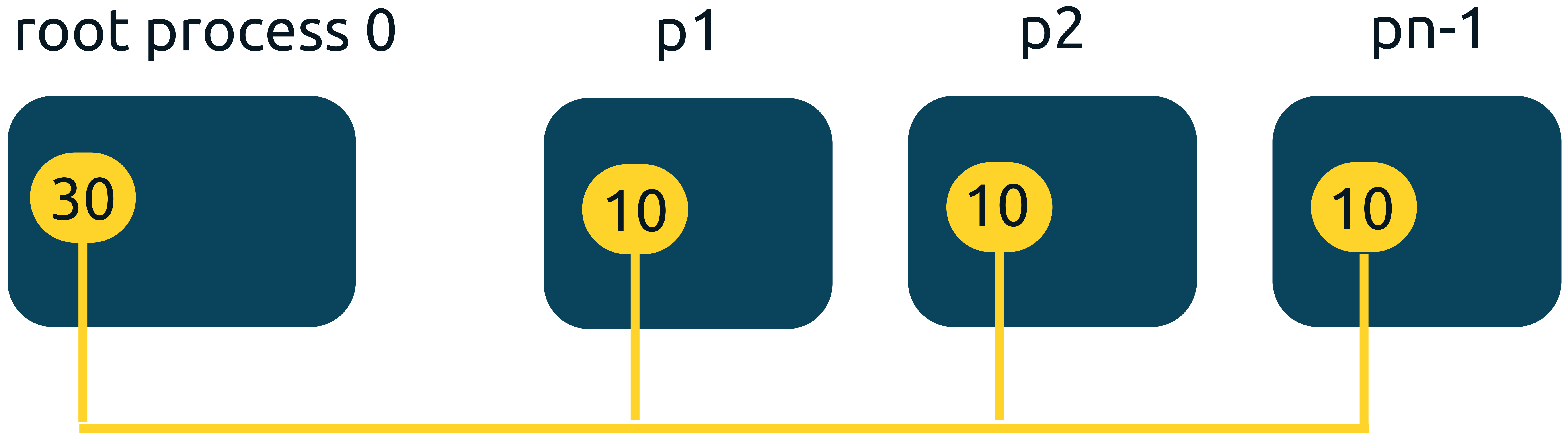
MPI_Reduce()



example

perform an operation of all other resolve values

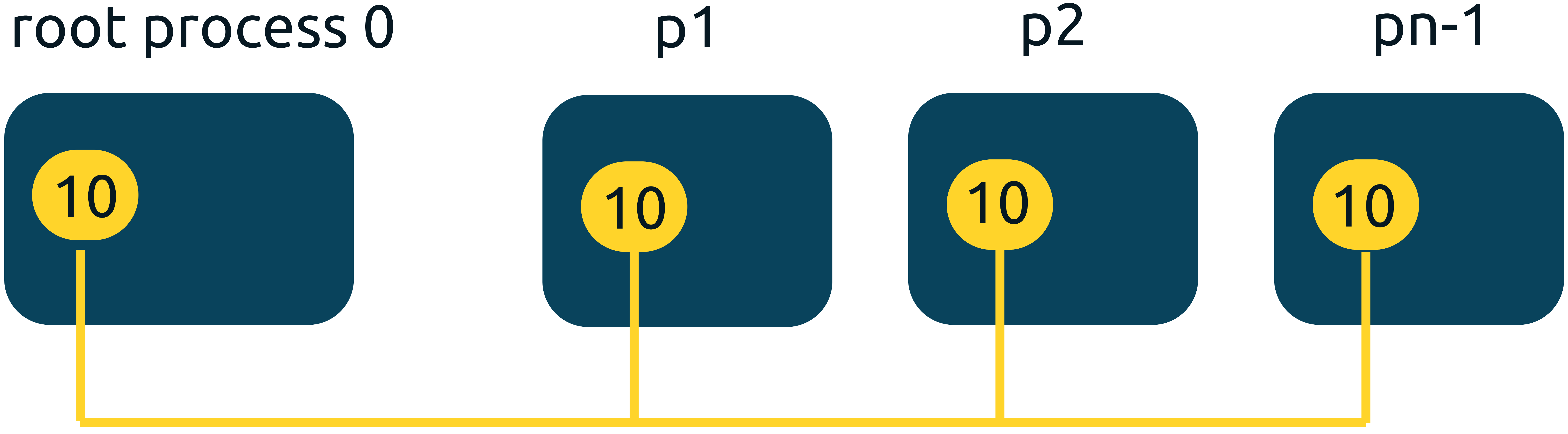
MPI_Reduce()



example

perform an operation of all other resolve values

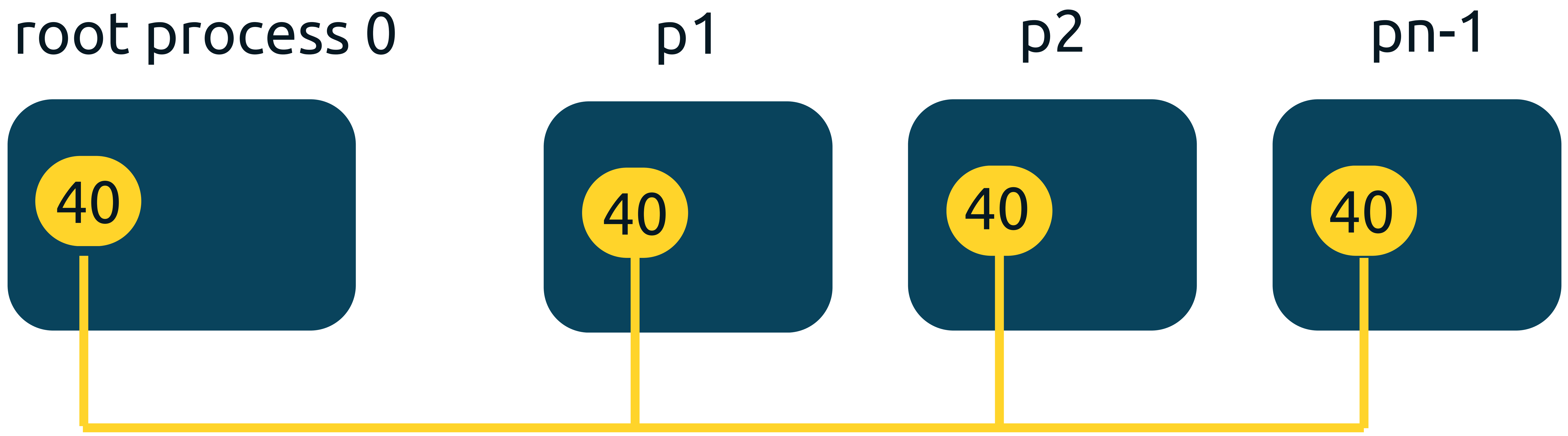
MPI_AllReduce()



example

perform an operation of all other resolve values

MPI_AllReduce()



example

perform an operation of all other resolve values

Synchronization

In a parallels computations there are various global operations:

- Synchronization
- Data movement **(MPI_Bcast())**
- Collective computations (**MPI_Reduce()**)

The synchronization is used when a processor with a rank n' does relay to another processor n, so n' won't continue execution until n finish processing data that n' need to continue executing.

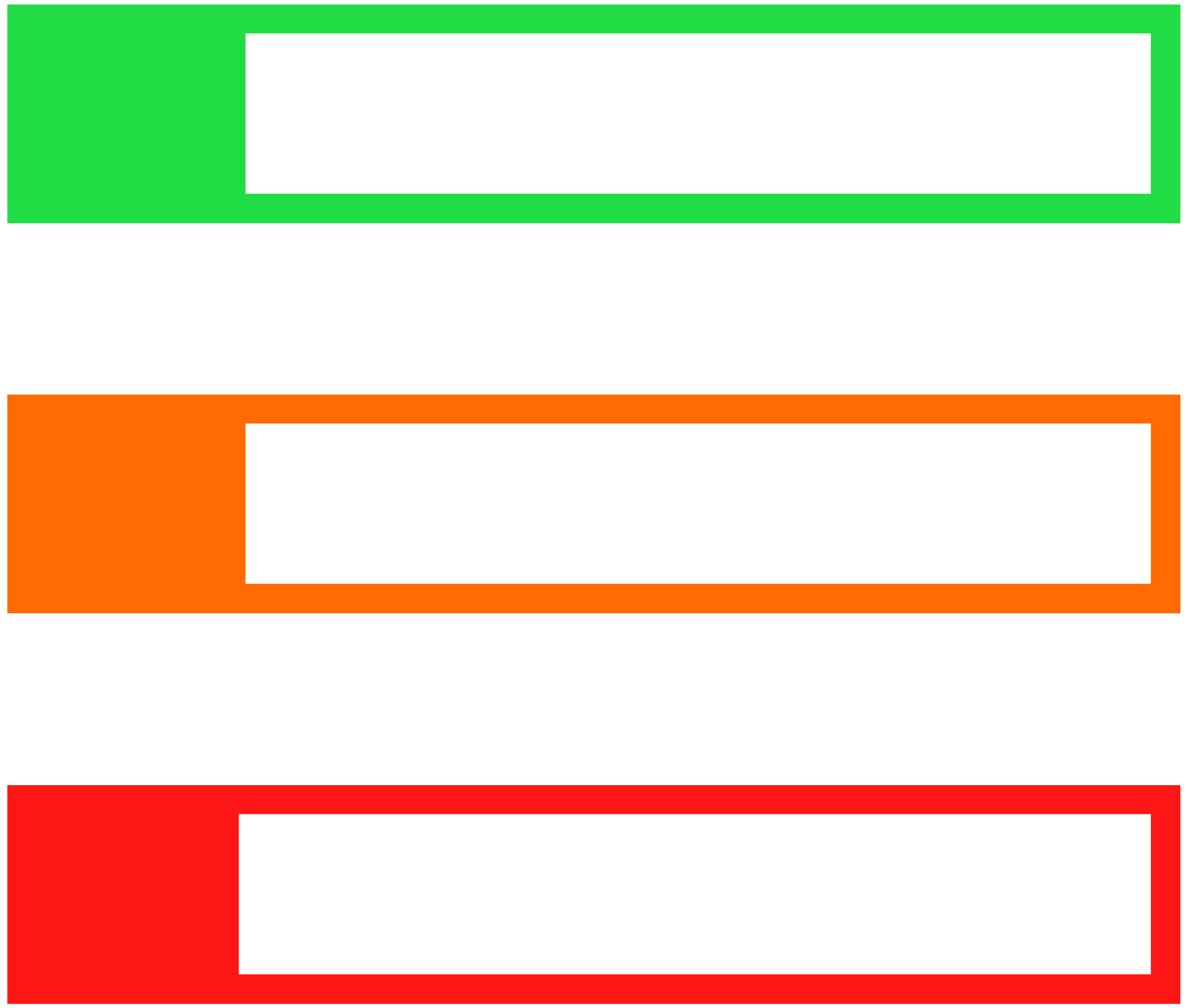
print all results in order sequences, suppose we have 3 processor

and let's try to execute this code:

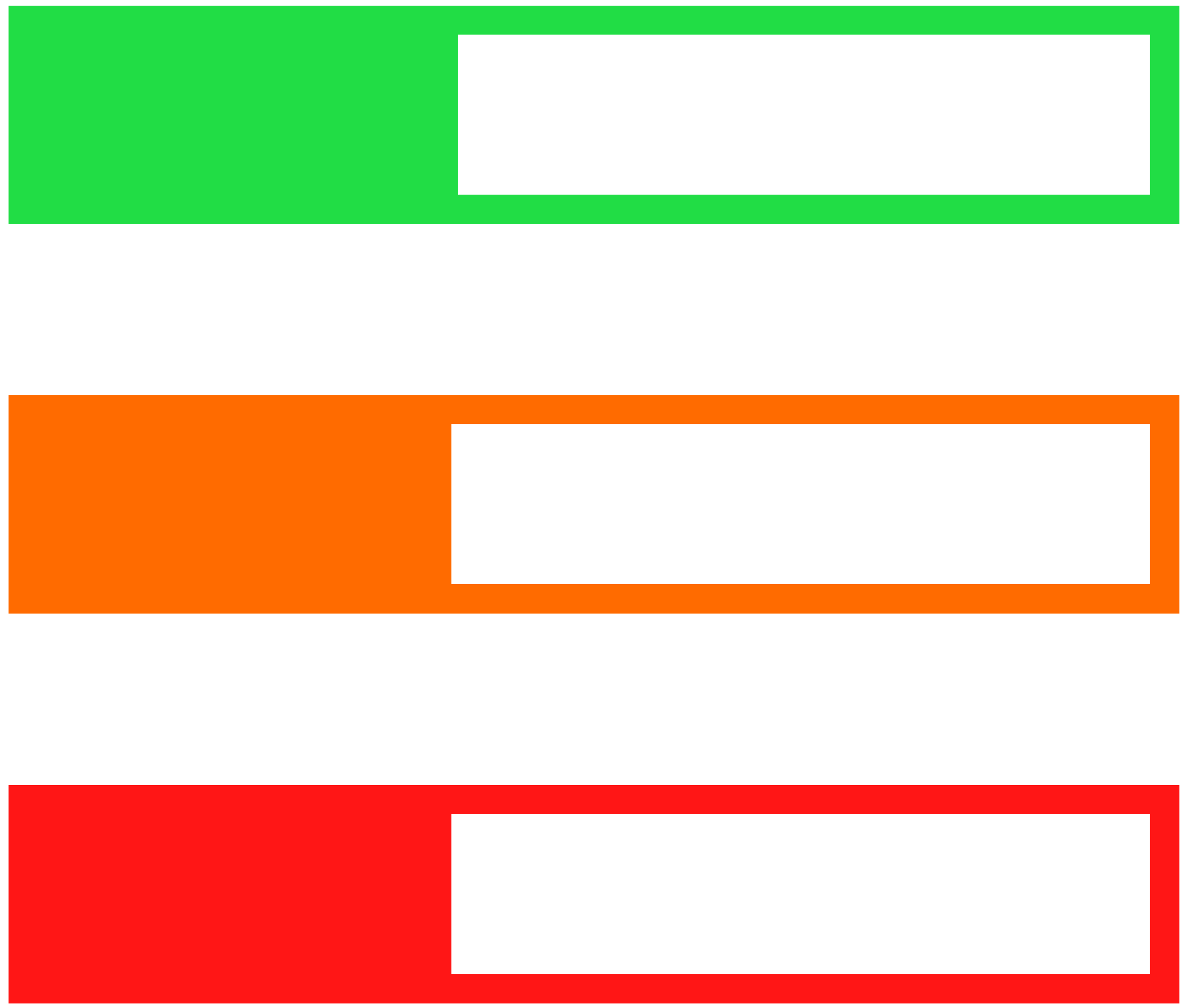
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```

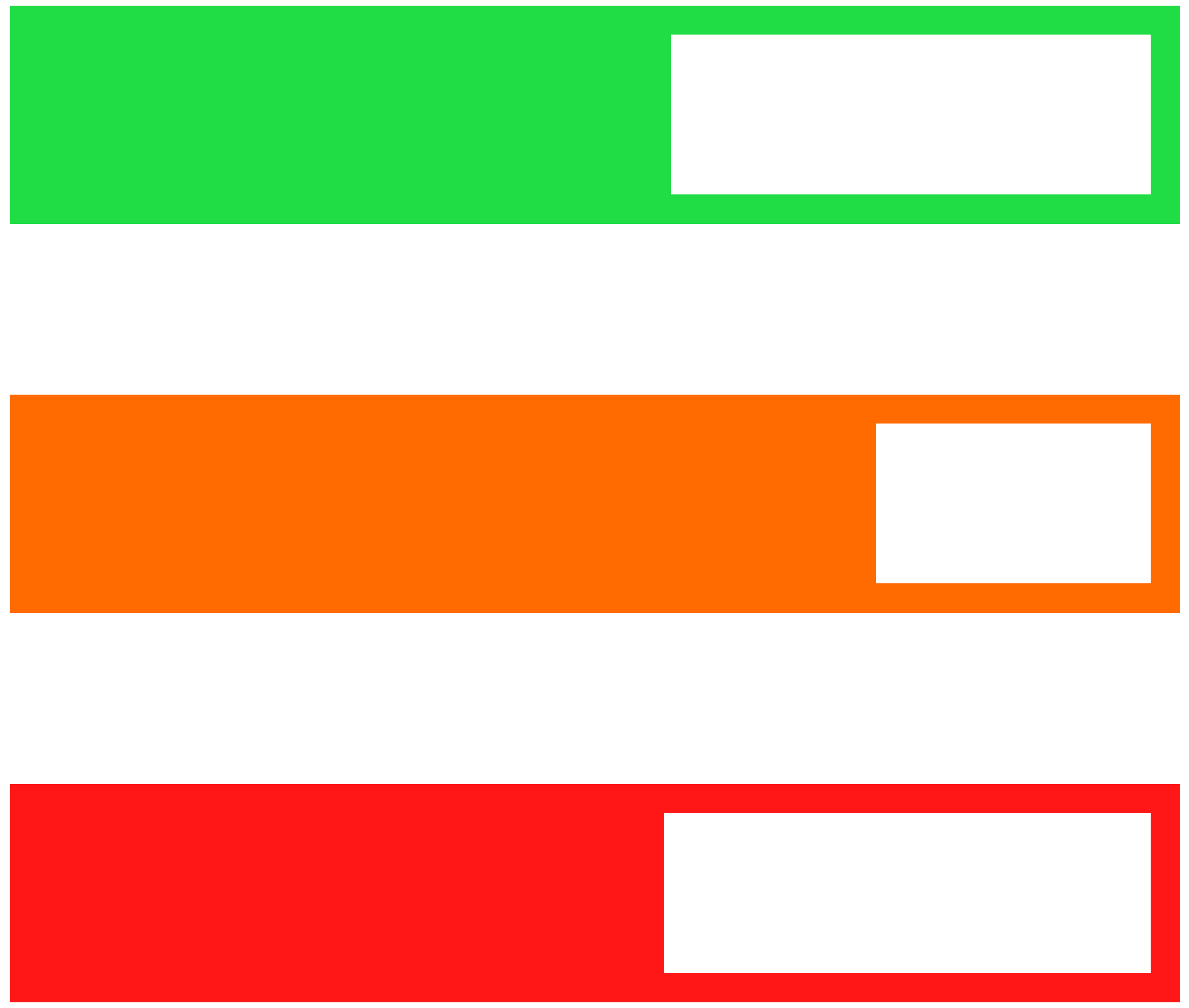
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
 int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```



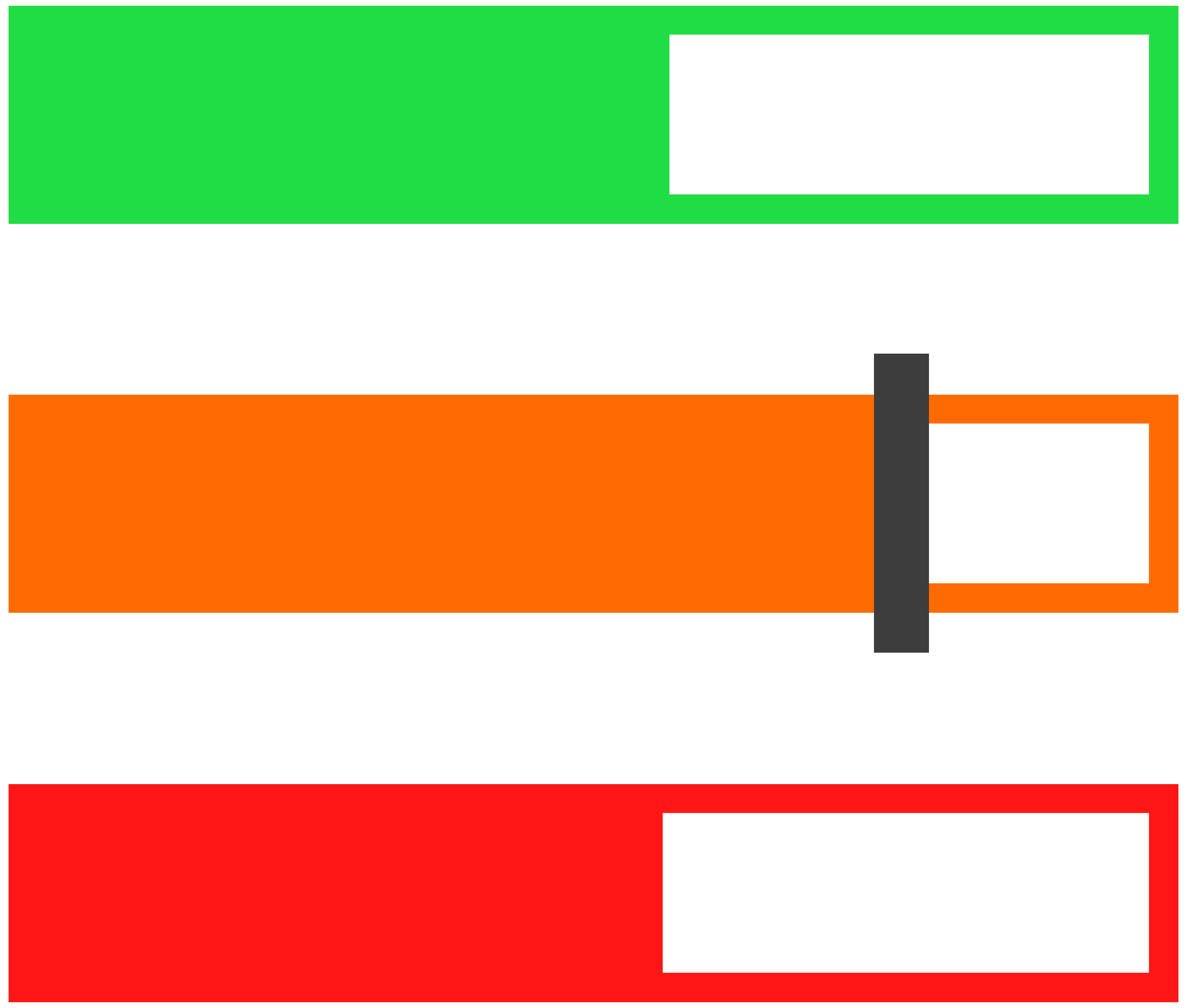
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Hello, world. I am %d of %d\n", rank, nprocs);
  fflush(stdout);
  MPI_Finalize();
  return 0;
}
```



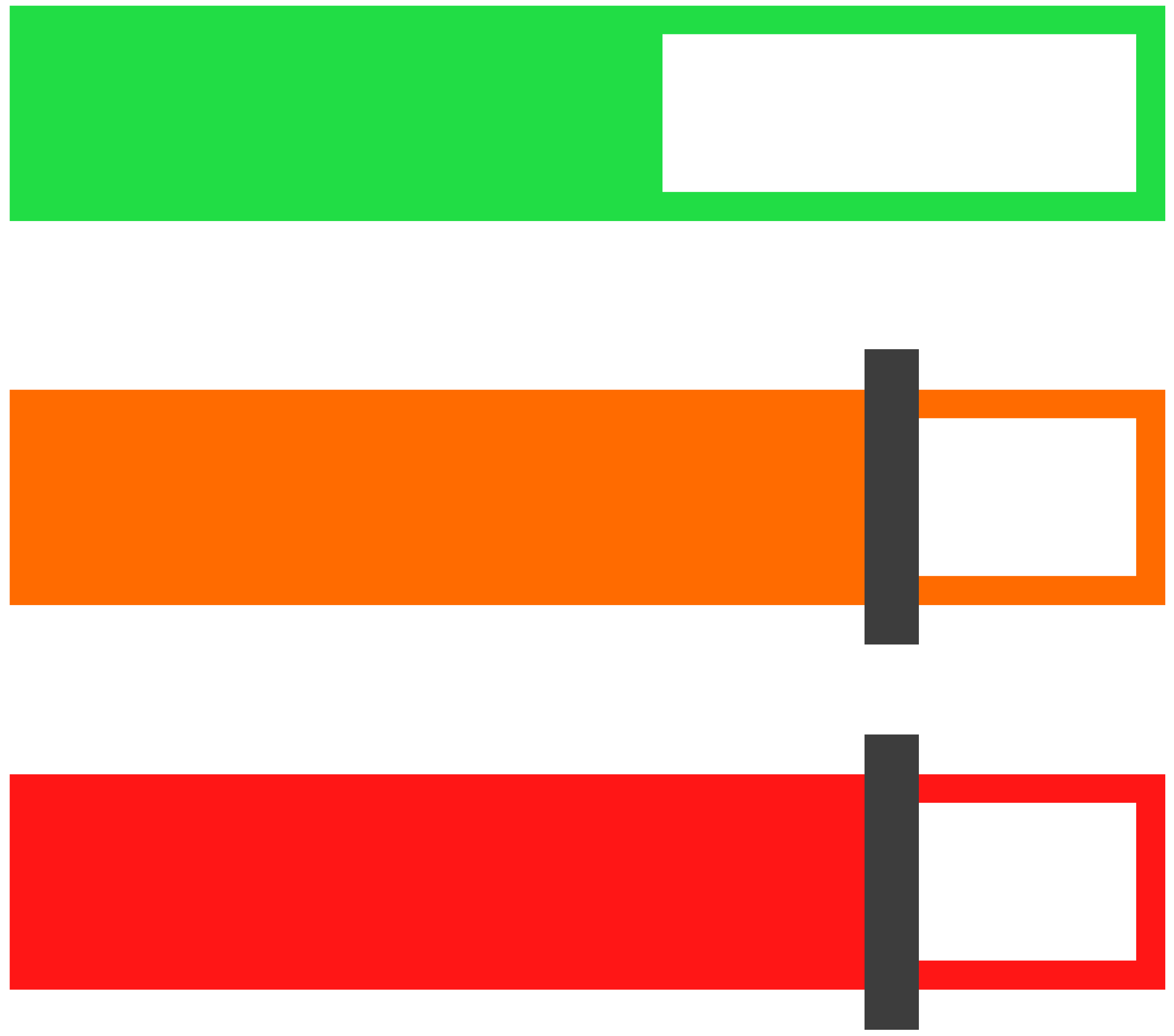
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```



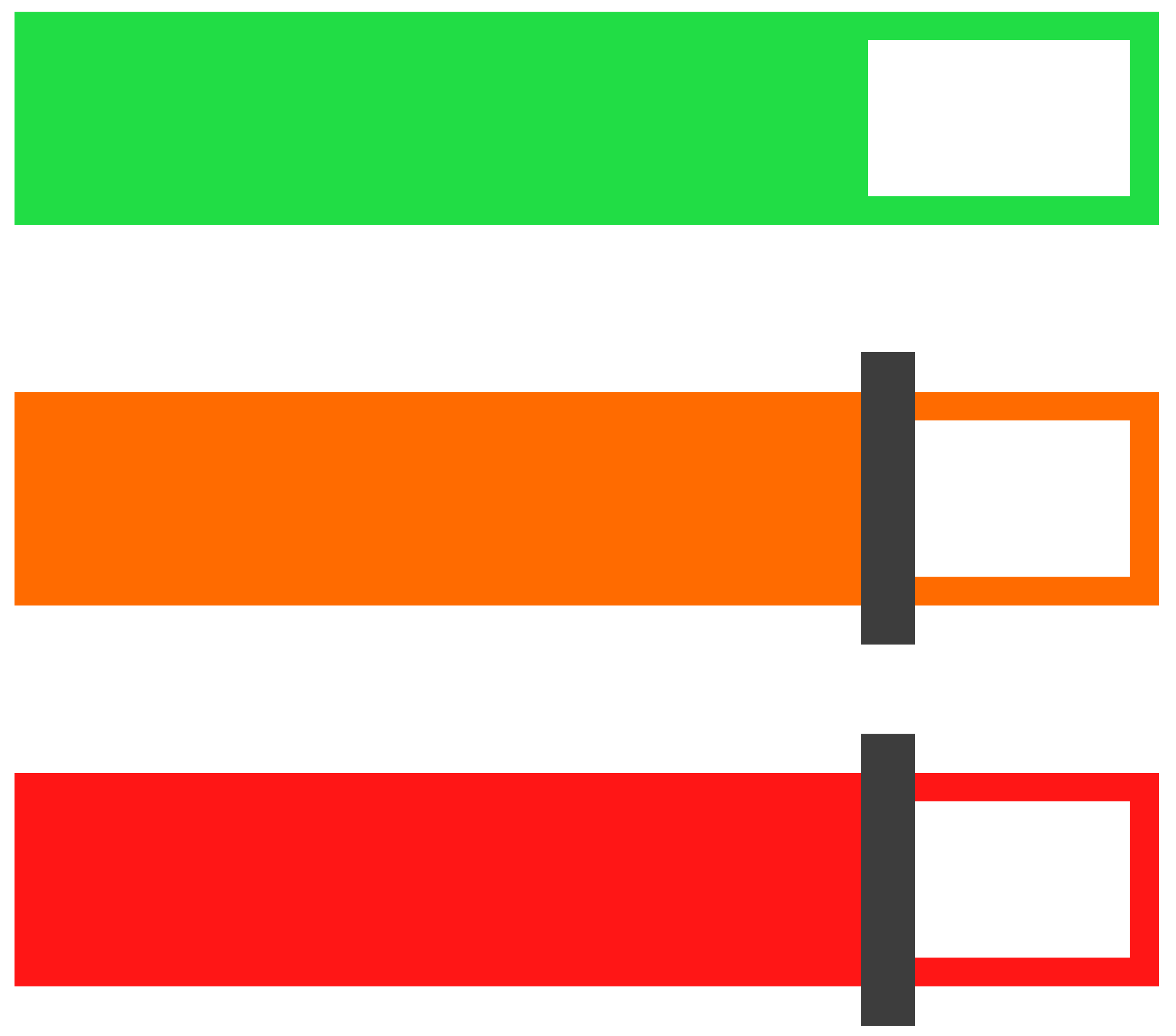
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```



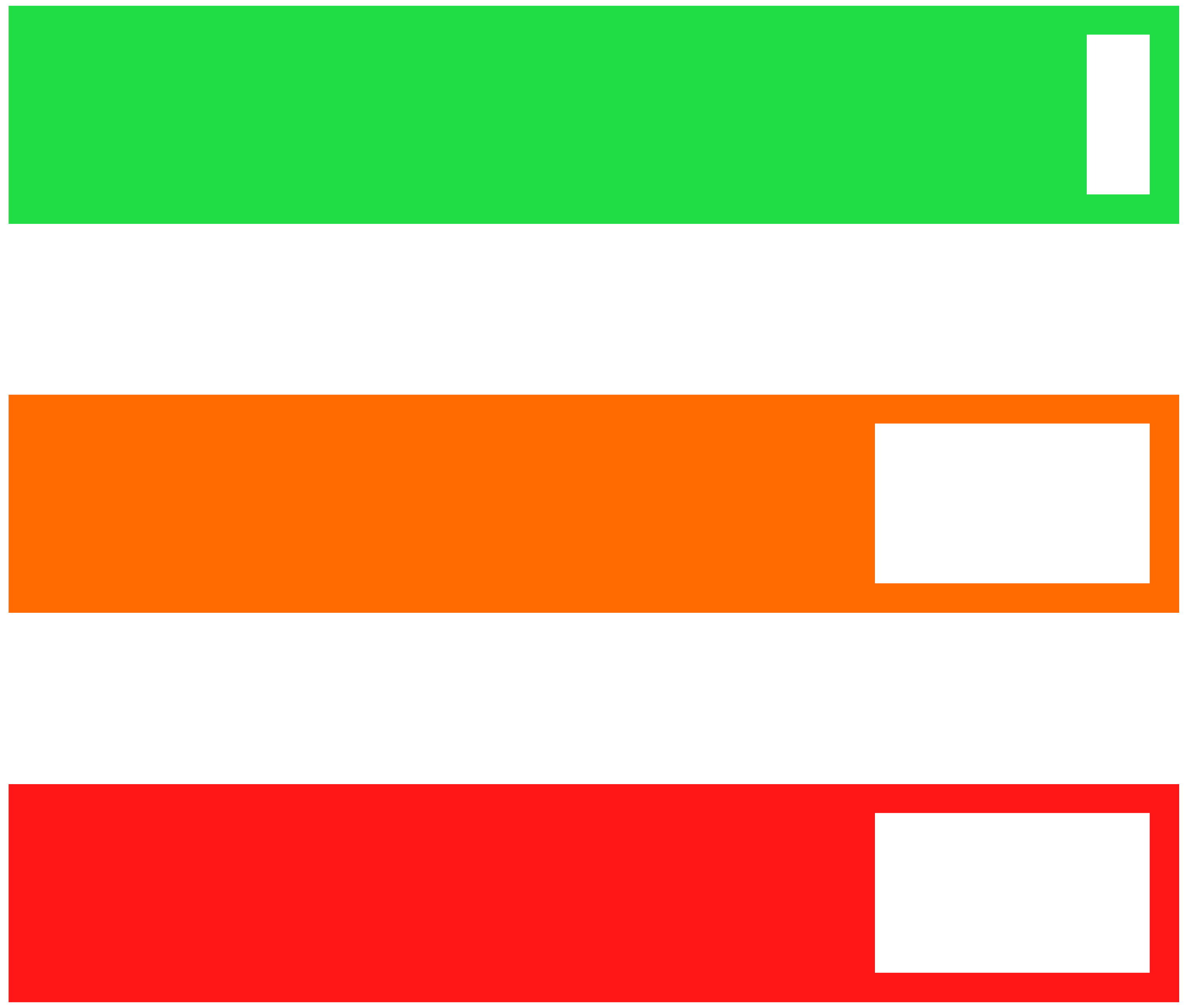
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```



```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```



```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```



```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world. I am %d of %d\n", rank, nprocs);
   fflush(stdout);
   MPI_Finalize();
   return 0;
}
```

Thank you.