

# Parallel Algorithms and Programming

## MPI

Thomas Ropars

`thomas.ropars@univ-grenoble-alpes.fr`

`http://tropars.github.io/`

2018

1

## Agenda

Message Passing Systems

Introduction to MPI

Point-to-point communication

Collective communication

Other features

2

# Agenda

Message Passing Systems

Introduction to MPI

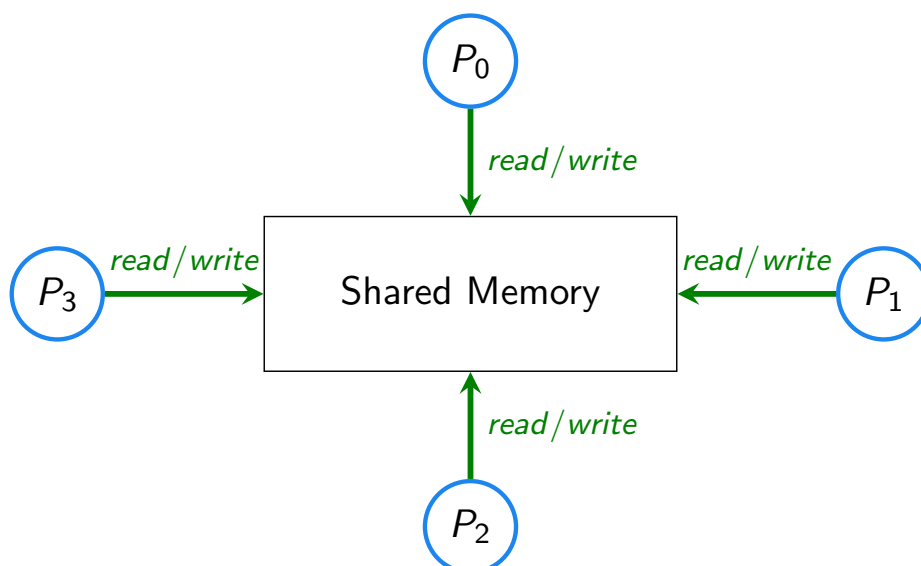
Point-to-point communication

Collective communication

Other features

3

## Shared memory model

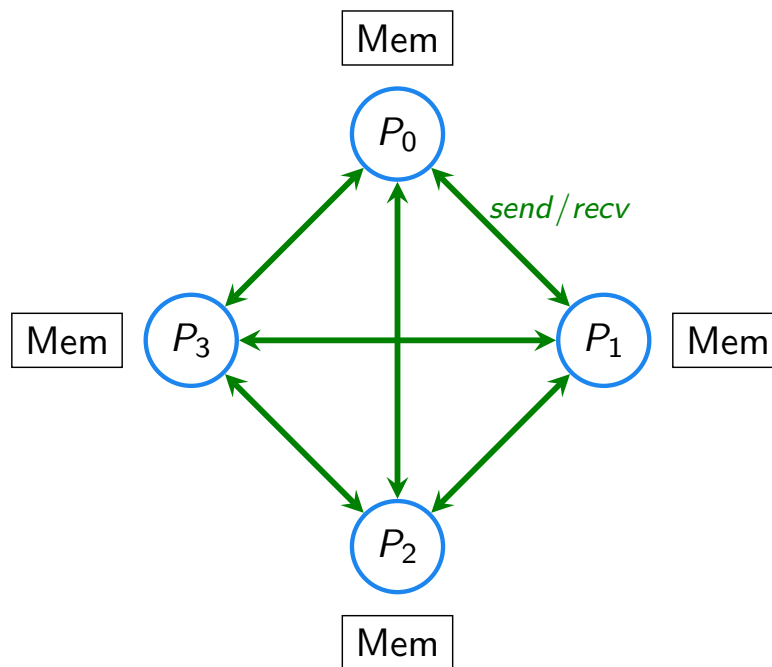


- Processes have access to a shared address space
- Processes communicate by reading and writing into the shared address space

4

# Distributed memory model

Message passing



- Each process has its own private memory
- Processes communicate by sending and receiving messages

5

## Applying the models

### Natural fit

- The **shared memory model** corresponds to threads executing on a single processor
- The **distributed memory model** corresponds to processes executing on servers interconnected through a network

### However

- Shared memory can be implemented on top of the distributed memory model
  - Distributed shared memory
  - Partitionable Global Address Space
- The distributed memory model can be implemented on top of shared memory
  - Send/Recv operations can be implemented on top of shared memory

6

## In a supercomputer

A large number of servers:

- Interconnected through a high-performance network
- Equipped with multicore multi-processors and accelerators

### What programming model to use?

- Hybrid solution
  - ▶ Message passing for inter-node communication
  - ▶ Shared memory inside a node
- Message passing everywhere
  - ▶ Less and less used as the number of cores per node increases

7

## Message Passing Programming Model

### Differences with the shared memory model

- Communication is explicit
  - ▶ The user is in charge of managing communication
  - ▶ The programming effort is bigger
- No good automatic techniques to parallelize code
- More efficient when running on a distributed setup
  - ▶ Better control on the data movements

8

# The Message Passing Interface (MPI)

<http://mpi-forum.org/>

MPI is the most commonly used solution to program message passing applications in the HPC context.

## What is MPI?

- MPI is a standard
  - ▶ It defines a set of operations to program message passing applications.
  - ▶ The standard defines the semantic of the operations (not how they are implemented)
  - ▶ Current version is 3.1 (<http://mpi-forum.org/mpi-31/>)
- Several implementations of the standard exist (libraries)
  - ▶ Open MPI and MPICH are the two main open source implementations (provide C and Fortran bindings)

9

## Agenda

Message Passing Systems

Introduction to MPI

Point-to-point communication

Collective communication

Other features

## My first MPI program

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

int main(int argc, char *argv[])
{
    char msg[20];
    int my_rank;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        strcpy(msg, "Hello!");
        MPI_Send(msg, strlen(msg), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(msg, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("I received %s!\n", msg);
    }
    MPI_Finalize();
}
```

11

## SPMD application

MPI programs follow the SPMD execution model:

- Each process executes the same program at independent points
- Only the data differ from one process to the others
- Different actions may be taken based on the rank of the process

12

# Compiling and executing

## Compiling

- Use `mpicc` instead of `gcc` (`mpicxx`, `mpif77`, `mpif90`)

```
mpicc -o hello_world hello_world.c
```

## Executing

```
mpirun -n 2 -hostfile machine_file ./hello_world
```

- Creates 2 MPI processes that will run on the 2 first machines listed in the `machine_file` (implementation dependent)
- If no `machine_file` is provided, the processes are created on the local machine

13

# Back to our example

## Mandatory calls (by every process)

- `MPI_Init()`: Initialize the MPI execution environment
  - ▶ No other MPI calls can be done before `Init()`.
- `MPI_Finalize()`: Terminates MPI execution environment
  - ▶ To be called before terminating the program

Note that all MPI functions are prefixed with `MPI_`

14

# Communicators and ranks

## Communicators

- A communicator defines a **group** of processes that can communicate in a communication **context**.
- Inside a group, processes have a unique rank
- Ranks go from 0 to  $p - 1$  in a group of size  $p$
- At the beginning of the application, a default communicator including all application processes is created:  
`MPI_COMM_WORLD`
- Any communication occurs in the context of a communicator
- Processes may belong to multiple communicators and have a different rank in different communicators

15

## Communicators and ranks: Retrieving basic information

- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`: Get rank of the process in `MPI_COMM_WORLD`.
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`: Get the number of processes belonging to the group associated with `MPI_COMM_WORLD`.

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int size, rank;
    char name[256];

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

    gethostname(name, 256);
    printf("Hello from %d on %s (out of %d procs.)\n", rank, name, size);

    MPI_Finalize();
}
```

16



## MPI Messages

A MPI message includes a payload (the data) and metadata (called the envelope).

### Metadata

- Processes rank (sender and receiver)
- A Communicator (the context of the communication)
- A message tag (can be used to distinguish between messages inside a communicator)

### Payload

The payload is described with the following information:

- Address of the beginning of the buffer
- Number of elements
- Type of the elements

17

## Signature of send/recv functions

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

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

18

## Elementary datatypes in C

MPI datatype	C datatype
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	1 Byte
MPI_PACKED	see MPI_Pack()

19

## A few more things

### The status object

Contains information about the communication (3 fields):

- MPI\_SOURCE: the id of the sender.
- MPI\_TAG: the tag of the message.
- MPI\_ERROR: the error code

The status object has to be allocated by the user.

### Wildcards for receptions

- MPI\_ANY\_SOURCE: receive from any source
- MPI\_ANY\_TAG: receive with any tag

20

# Agenda

Message Passing Systems

Introduction to MPI

Point-to-point communication

Collective communication

Other features

21

## Blocking communication

`MPI_Send()` and `MPI_Recv()` are blocking communication primitives.

What does blocking means in this context?

22

## Blocking communication

`MPI_Send()` and `MPI_Recv()` are blocking communication primitives.

What does blocking means in this context?

- **Blocking send:** When the call returns, it is safe to reuse the buffer containing the data to send.
  - ▶ It does not mean that the data has been transferred to the receiver.
  - ▶ It might only be that a local copy of the data has been made
  - ▶ It may complete before the corresponding receive has been posted
- **Blocking recv:** When the call returns, the received data are available in the buffer.

22

## Communication Mode

- **Standard (`MPI_Send()`)**
  - ▶ The send may buffer the message locally or wait until a corresponding reception is posted.
- **Buffered (`MPI_BSend()`)**
  - ▶ Force buffering if no matching reception has been posted.
- **Synchronous (`MPI_SSend()`)**
  - ▶ The send cannot complete until a matching receive has been posted (the operation is not local)
- **Ready (`MPI_RSend()`)**
  - ▶ The operation fails if the corresponding reception has not been posted.
  - ▶ Still, send may complete before reception is complete

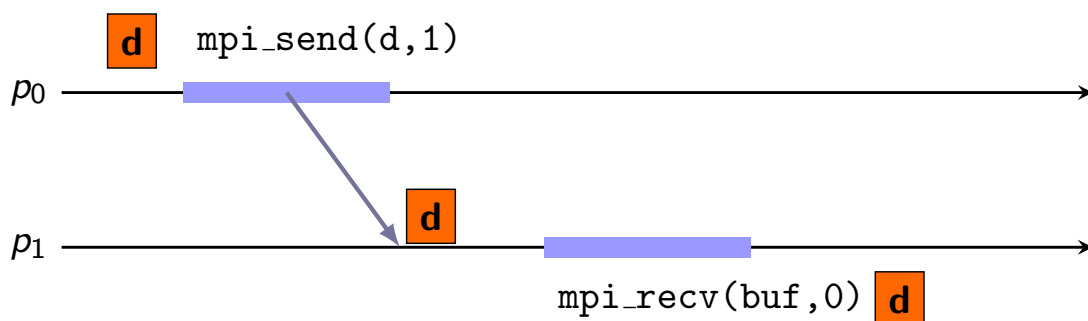
23

# Protocols for standard mode

A taste of the implementation

## Eager protocol

- Data sent assuming receiver can store it
- The receiver may not have posted the corresponding reception
- This solution is used only for small messages (typically < 64kB)
  - This solution has low synchronization delays
  - It may require an extra message copy on destination side



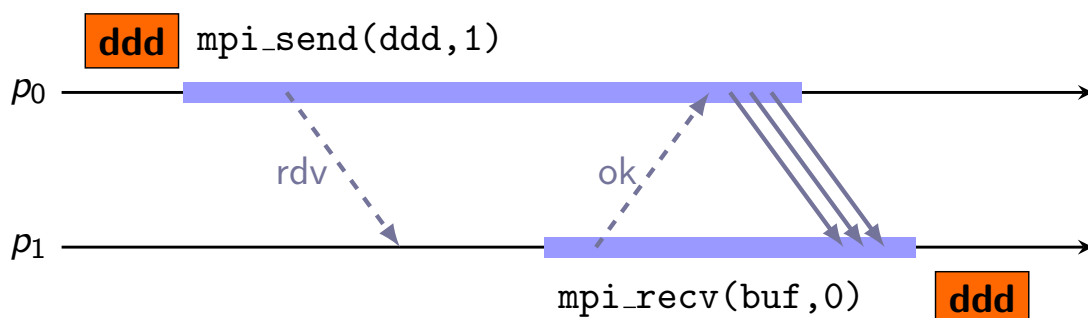
24

# Protocols for standard mode

A taste of the implementation

## Rendezvous protocol

- Message is not sent until the receiver is ok
- Protocol used for *large messages*
  - Higher synchronization cost
  - If the message is big, it should be buffered on sender side.



25

## Non blocking communication

Basic idea: dividing communication into two logical steps

- Posting a request: Informing the library of an operation to be performed
- Checking for completion: Verifying whether the action corresponding to the request is done

### Posting a request

- Non-blocking send: `MPI_Isend()`
- Non-blocking recv: `MPI_Irecv()`
- They return a `MPI_Request` to be used to check for completion

26

## Non blocking communication

### Checking request completion

- Testing if the request is completed : `MPI_Test()`
  - ▶ Returns true or false depending if the request is completed
- Other versions to test several requests at once (suffix `_any`, `_some`, `_all`)

### Waiting for request completion

- Waiting until the request is completed : `MPI_Wait()`
- Other versions to wait for several requests at once (suffix `_any`, `_some`, `_all`)

27

## Overlapping communication and computation

Non-blocking communication primitives allow trying to overlap communication and computation

- Better performance if the two occur in parallel

```
MPI_Isend(..., req);  
...  
/* run some computation */  
...  
MPI_Wait(req);
```

However, things are not that simple:

- MPI libraries are not multi-threaded (by default)
  - The only thread is the application thread (no progress thread)
- The only way to get overlapping is through specialized hardware
  - The network card has to be able to manage the data transfer alone

28

## Matching incoming messages and reception requests

MPI communication channels are *First-in-First-out* (FIFO)

- Note however that a communication channel is defined in the context of a communicator

### Matching rules

- When the reception request is named (source and tag defined), it is matched with the next arriving message from the source with correct tag.
- When the reception request is anonymous (`MPI_ANY_SOURCE`), it is matched with next message from any process in the communicator
  - Note that the matching is done when the envelope of the message arrives.

29

# Discussion about performance of P2P communication

Things to have in mind to get good communication performance:

- Avoid extra copies of the messages
  - ▶ Reception requests should be posted before corresponding send requests
- Reduce synchronization delays
  - ▶ Same solution as before
  - ▶ The latency of the network also has an impact
- Take into account the topology of the underlying network
  - ▶ Contention can have a dramatic impact on performance

30

## Agenda

Message Passing Systems

Introduction to MPI

Point-to-point communication

**Collective communication**

Other features

31



## Collective communication

A collective operation involves all the processes of a communicator.

All the classic operations are defined in MPI:

- Barrier (global synchronization)
- Broadcast (one-to-all)
- Scatter/ gather
- Allgather (gather + all members receive the result)
- AllToAll
- Reduce, AllReduce (Example of op: sum, max, min)
- etc.

There are **v** versions of some collectives (Gatherv, Scatterv, Allgatherv, Alltoallv):

- They allow using a vector of send or recv buffers.

32

## Example with broadcast

### Signature

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

### Broadcast Hello

```
#include <mpi.h>
int main(int argc, char *argv[])
{
    char msg[20];
    int my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0)
        strcpy(msg, "Hello from 0!");
    MPI_Bcast(msg, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
    printf("rank %d: I received %s\n", my_rank, msg);
    MPI_Finalize();
}
```

33

## About collectives and synchronization

### What the standard says

A collective communication call may, or may not, have the effect of synchronizing all calling processes.

- It cannot be assumed that collectives synchronize processes
  - ▶ Synchronizing here means that no process would complete the collective operation until the last one entered the collective
  - ▶ `MPI_Barrier()` still synchronizes the processes
- Why is synchronization useful?
  - ▶ Ensure correct message matching when using anonymous receptions
  - ▶ Avoid too many *unexpected* messages (where the reception request is not yet posted)

34

## About collectives and synchronization

### What about real life?

- In most libraries, collectives imply a synchronization
  - ▶ An implementation without synchronization is costly
- A user program that assumes no synchronization is erroneous

### Incorrect code (High risk of deadlock)

```
if(my_rank == 1)
    MPI_Recv(0);

MPI_Bcast(...);

if(my_rank == 0)
    MPI_Send(1);
```

35

# Implementation of collectives

- MPI libraries implement several algorithms for each collective operation
- Different criteria are used to select the best one for a call, taking into account:
  - ▶ The number of processes involved
  - ▶ The size of the message
- A supercomputer may have its own custom MPI library
  - ▶ Take into account the physical network to optimize collectives

36

## Agenda

Message Passing Systems

Introduction to MPI

Point-to-point communication

Collective communication

Other features

37

## Derived datatypes

We have already introduced the basic datatypes defined by MPI

- They allow sending contiguous blocks of data of one type

Sometimes one will want to:

- Send non-contiguous data (a sub-block of a matrix)
- Buffers containing different datatypes (an integer count, followed by a sequence of real numbers)

One can defined derived datatypes

38

## Derived datatypes

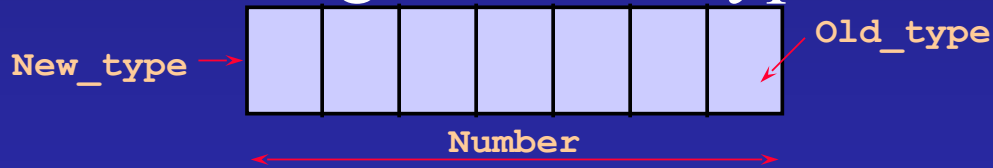
- A derived datatype is defined based on a type-map
  - ▶ A type-map is a sequence of pairs {dtype, displacement}
  - ▶ The displacement is an address shift relative to the basic address

### Committing types

- `MPI_Type_commit()`
  - ▶ Commits the definition of the new datatype
  - ▶ A datatype has to be committed before it can be used in a communication
- `MPI_Type_free()`
  - ▶ Mark the datatype object for de-allocation

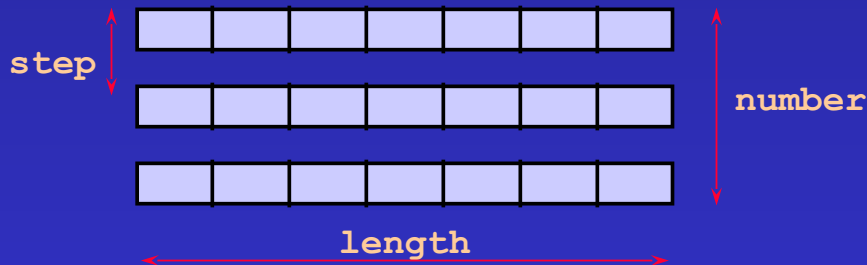
39

# Contiguous Datatype



`MPI_Type_contiguous(number, Old_type, &New_type)`

# Vector Datatype



`MPI_Type_(h)vector(number, length, step, Old_type, &New_type)`  
 step in bytes

```
MPI_Datatype Col_Type, Row_Type;
MPI_Comm comm;
```

```
MPI_Type_contiguous(6, MPI_REAL, &Col_Type);
MPI_Type_commit(&Col_Type);
```

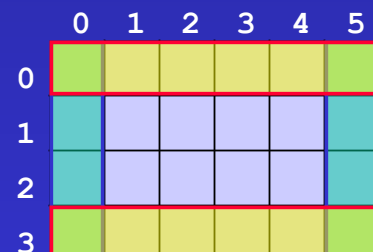


```
MPI_Type_vector(4, 1, 6, MPI_REAL, &Row_Type);
MPI_Type_commit(&Row_Type);
```

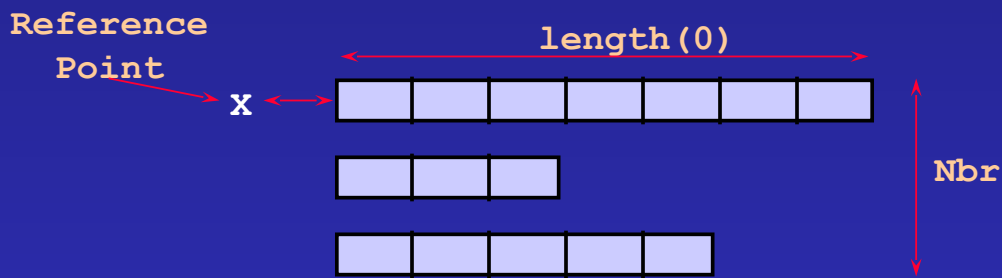


```
...
MPI_Send(A(0,0), 1, Col_Type, west, 0, comm);
MPI_Send(A(0,5), 1, Col_Type, east, 0, comm);
MPI_Send(A(0,0), 1, Row_Type, north, 0, comm);
MPI_Send(A(3,0), 1, Row_Type, south, 0, comm);
...
```

```
MPI_Type_free(&Col_Type);
MPI_Type_free(&Row_Type);
```



# Indexed Datatype



`MPI_Type_(h) indexed(Nbr, length, where, Old_type, &New_type)`  
 where in bytes

- The `New_type` is made of `Nbr` arrays `i` of size `length(i)`, each one being at `where(i)` of the `Old_type` (`where(i)` in number of items except for `MPI_Type_hindexed`).

## Performance with derived datatypes

Derived datatypes should be used carefully:

- By default, the data are copied into a contiguous buffer being sent (no zero-copy)
- Special hardware support is required to avoid this extra copy

# Operations on communicators

New communicators can be created by the user:

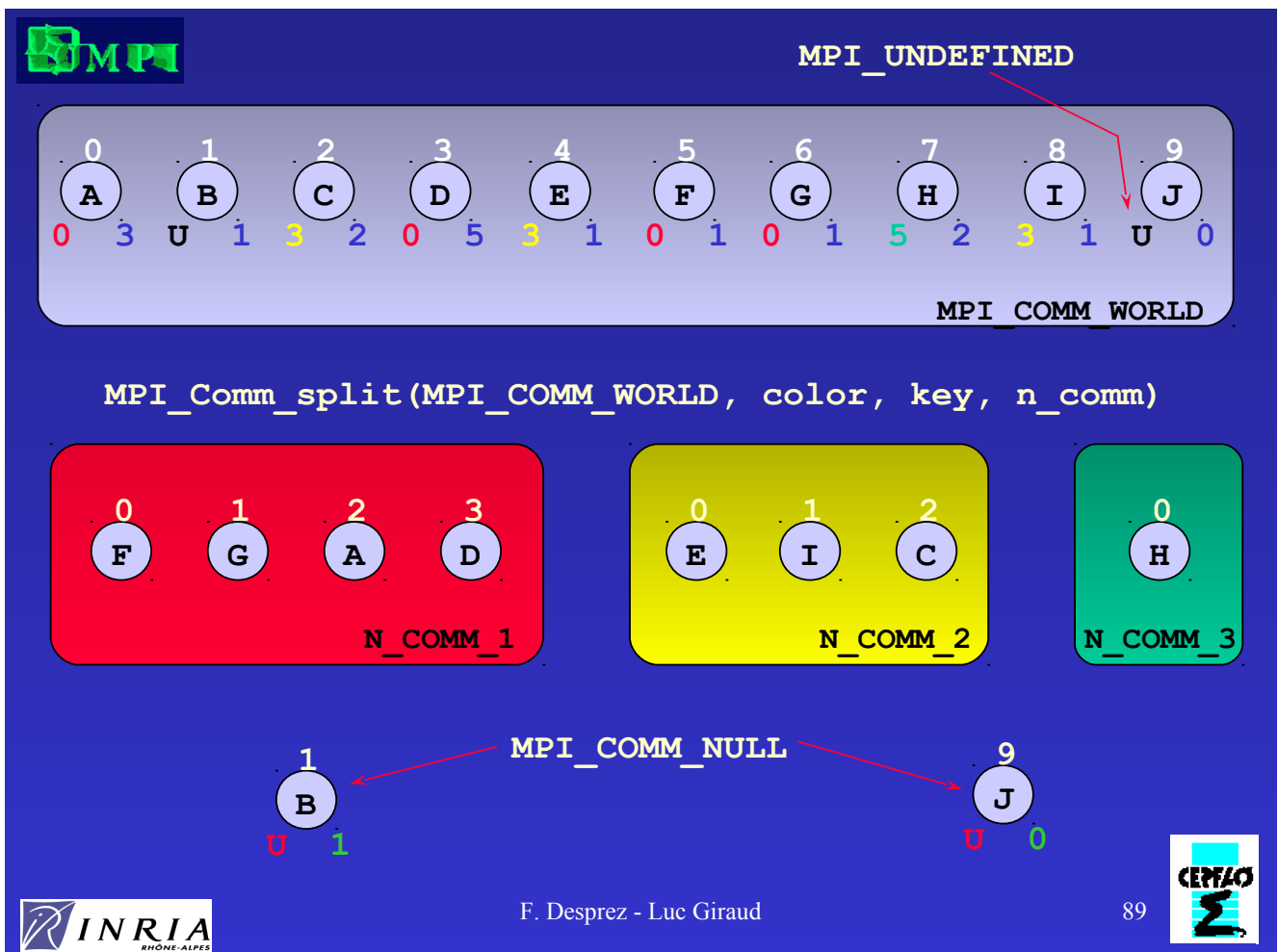
- Duplicating a communicator (`MPI_Comm_dup()`)
  - ▶ Same group of processes as the original communicator
  - ▶ New communication context

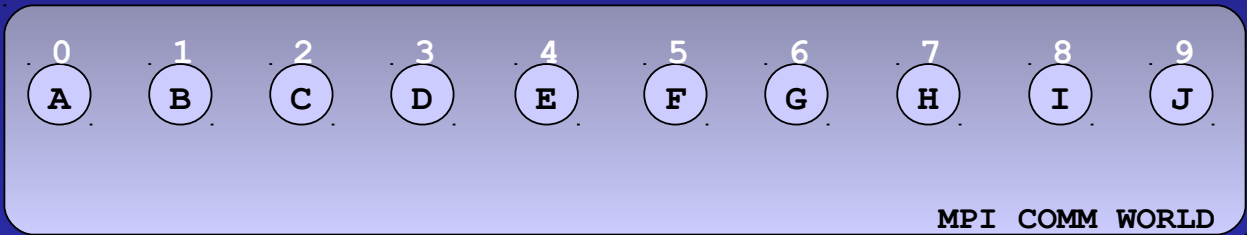
```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
```

- Splitting a communicator (`MPI_Comm_split()`)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

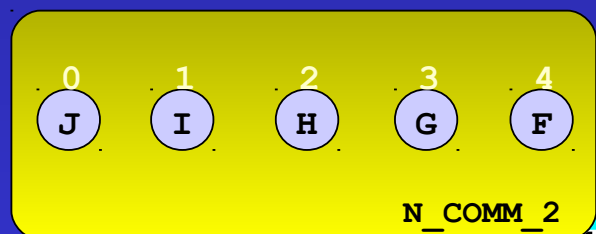
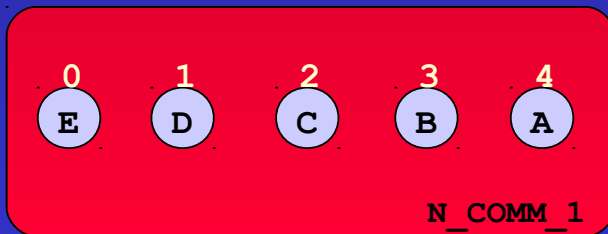
- ▶ Partitions the group associated with `comm` into disjoint subgroups, one for each value of `color`.
- ▶ Each subgroup contains all processes of the same `color`.
- ▶ Within each subgroup, the processes are ranked in the order defined by the value of the argument `key`.
- ▶ Useful when defining hierarchy of computation





```
MPI_Comm_rank(MPI_COMM_WORLD, rank);  
MPI_Comm_size(MPI_COMM_WORLD, size);  
color = 2*rank/size;  
key = size - rank - 1
```

```
MPI_Comm_split(MPI_COMM_WORLD, color, key, n_comm)
```



## Warning

The goal of this presentation is only to provide an overview of the MPI interface.

Many more features are available, including:

- One-sided communication
- Non-blocking collectives
- Process management
- Inter-communicators
- etc.

MPI 3.1 standard is a 836-page document



## References

- Many resources available on the Internet
- The man-pages
- The specification documents are available at:  
<http://mpi-forum.org/docs/>