HIGH PERFORMANCE
COMPUTING CINECA
ITALY

Parallel Programming with MPI

Part V - Advanced MPI

HPC SCHOOL

COSTA RICA

ALESSANDRO MARANI

a.marani@cineca.it || San José 2023

AGENDA

MPI-2 NEW FEATURES
One-sided communications, dynamic processes with MPI_Comm_spawn

DEBUGGING AND PROFILING WITH PMPI
What is PMPI and some quick example

MPI-3 AND MPI-4
Nonblocking collectives, Neighborough collectives, new features of MPI-4

KABRÉ
SUPERCOMPUTADORA

MPI-2 new features

One-sided communications

- In two-sided (point-to-point) communications there can be a delay if the sender has to wait to send the data because the receiver is not ready.
- The MPI-2 standard added **Remote Memory Access** (RMA), also called one-sided communication, to decouple data transfer from system synchronisation.
- In RMA only one process carries out the data transfer. The **MPI_Get** and **MPI_Put** calls are non-blocking and don’t require intervention of the remote process.
- MPI-3 further extended RMA to improve functionality and performance.
- Here we only describe the simple MPI RMA functionality with MPI Get/Put and Fence synchronisation.

One-sided communications

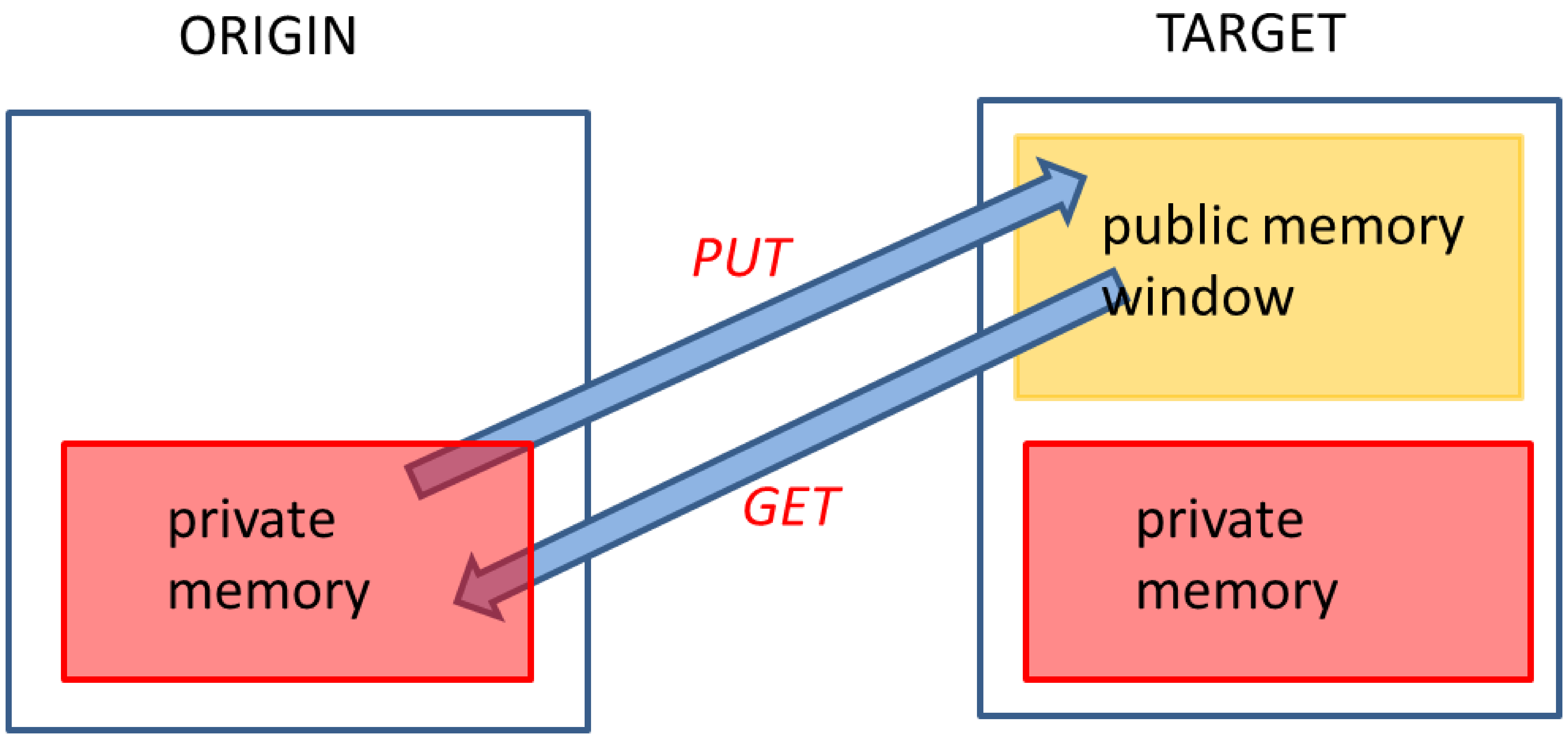
Advantages of RMA:

- ✓ With only one process taking part performance should be greater (no implicit synchronization, all data movement routines are non-blocking)
- ✓ Some programs are more easily written with RMA since the semantics are easier for programmers (as opposed to message passing)

In practice RMA may not be faster, particularly if implemented by P2P by the MPI vendor.

Using one-sided communications

1. Define an area of memory to be used for the RMA (“window”).
2. Specify the data to be moved and where to move them.
3. Specify a way to know when the data are available.



MPI_Win create

```
int MPI_Win_create(void *base, MPI_Aint size, int
disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

base initial address for the window
size size of the window in bytes
info info argument
comm communicator involved
win window object handle

MPI_Get (and MPI_Put)

```
int MPI_Get(void *origin_addr, int origin_count,
 MPI_Datatype origin_datatype, int target_rank, MPI_Aint
 target_disp, int target_count, MPI_Datatype
 target_datatype, MPI_Win *win);
```

origin_addr	address of the buffer in which to receive data
origin_count	number of entries in the receive buffer
origin_datatype	datatype of each entry in origin buffer
target_rank	rank of target
target_disp	displacement from window start to beginning of target data
target_count	number of entries to transfer
target_datatype	datatype of entries
win	window object handle

…Similarly for MPI_Put

One-sided communications synchronization

- The MPI_Get and MPI_Put calls are non-blocking.
- Need to synchronize the data transfer so that one process knows when it is safe to read the data of another.
- MPI provides various synchronization models, but we will consider only **MPI_Win_Fence**.
- This is used to *start* and *end* the PUT/GET operations. All operations complete at the second fence synchronization.

One-sided communications example

```
MPI_Win win;
MPI_Win_create(sharedbuffer, NUM_ELEMENT, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
.....
MPI_Win_fence(0, win);

if (id != 0)
    MPI_Get(&localbuffer[0], NUM_ELEMENT, MPI_INT, id-1, 0, NUM_ELEMENT, MPI_INT, win);
else
    MPI_Get(&localbuffer[0], NUM_ELEMENT, MPI_INT, num_procs-1, 0, NUM_ELEMENT, MPI_INT, win);

MPI_Win_fence(0, win);
if (id < num_procs-1)
   MPI_Put(&localbuffer[0], NUM_ELEMENT, MPI_INT, id+1, 0, NUM_ELEMENT, MPI_INT, win);
 else
   MPI_Put(&localbuffer[0], NUM_ELEMENT, MPI_INT, 0, 0, NUM_ELEMENT, MPI_INT, win);

MPI_Win_fence(0, win);
MPI_Win_free(&win);
MPI_Finalize();
```

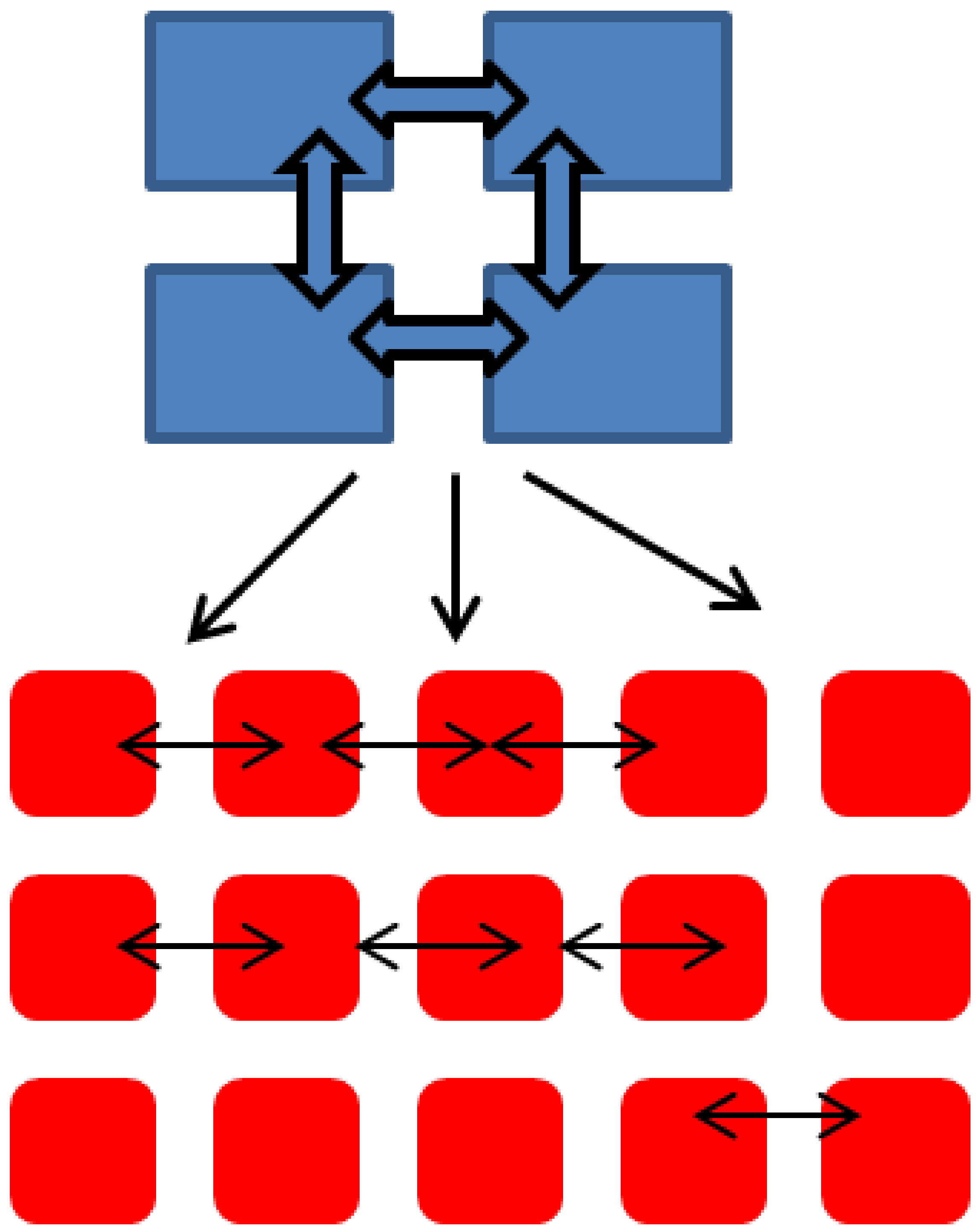
Dynamic processes

- Normally MPI tasks are fixed (e.g. by mpirun) at the start of execution.
- But can be useful to add or create tasks “on the fly”:

  - Master–slave type codes, or on heterogenous architectures (normal nodes + accelerators).

  - Client-server or peer-to-peer

- Handling faults failures



MPI_Comm_spawn

MPI-2 provides ***spawn functionality***

– `MPI_Comm_spawn`

starts a new set of processes with the same command lines (SPMD model)

– `MPI_Comm_spawn_multiple`

starts a new set of processes with potentially different command lines (i.e. different executables and arguments = MPMD)

Group of parents collectively call spawn:

- Launches a new set of child processes
- Child processes become an MPI job
- An intercommunicator is created between parents and children.
- Parents and children can then use MPI functions to communicate.

MPI_Comm_spawn

- Not all MPI implementations support MPI spawning.
- The MPI implementation may require particular runtime options.
- Remember that if working in a batch environment you should allocate resources to cover the spawned processes as well.
- MPI_UNIVERSE_SIZE is often used to set the total number of processes available (i.e. including spawned processes)
- Not commonly used in HPC environments. May be used in heterogenous (i.e. with accelerators), although OpenMP task creation is more likely.

MPI_Comm_spawn example

```
#define NUM_SPAWNS 2
int main(int argc, char* argv[])
{
   int np=NUM_SPAWNS;
   MPI_Comm parentcomm, intercomm;
   int errcodes[NUM_SPAWNS];
   MPI_Init( &argc, &argv );
   MPI_Comm_get_parent( &parentcomm );
    if (parentcomm == MPI_COMM_NULL)
    {
    // Create 2 more processes - example must be called spawn_example.exe for this to work
        MPI_Comm_spawn( "./spawnexample", MPI_ARGV_NULL, np, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
         &intercomm, errcodes);
        printf("I'm the parent.\n");
    }
    else
       printf("I'm the spawned.\n");
    MPI_Finalize();
    return 0;    }
```

Debugging and profiling with PMPI

Debugging and profiling with PMPI

- MPI implementations also provide a profiling interface called **PMPI**.
- In PMPI each standard MPI function (MPI_) has an equivalent function with prefix PMPI_ (e.g. *PMPI_Send*, *PMI_Recv*, etc).
- With PMPI it is possible to customize normal MPI commands to provide extra information useful for profiling or debugging.
- No modifications to source code – just link in your modified MPI library

```
mpicc my_mpi.c -c my_mpi.o
mpicc my_code.c -c my_code.o
mpicc my_mpi.o my_code.c -o my_exe.x
```

PMPI examples

```
// profiling example
static int send_count=0;
int MPI_Send(void *start,int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
{
send_count++;
printf(“Updated number of MPI_Send calls: %d\n”,send_count);
return PMPI_Send(start, count, datatype, dest, tag, comm);
}
```

```
Updated number of MPI_Send calls: 13
```

```
// Unsafe use of MPI_Send
// MPI_Send can be implemented as MPI_Isend
int MPI_Send(void *start, int count, MPI_Datatype datatype, int dest, int tag, MPI_comm comm)
 {
  MPI_Request req;
  return PMPI_Isend(start, count, datatype, dest, tag, comm, &req);
 }
```

MPI-3 and MPI-4

MPI-3

MPI 3.0 was approved in 2012. **MPI 3.1** was approved in 2015.

Features include:

- Non-blocking collectives
- Neighbourhood collectives
- New one sided communications
- Plus enhancements for many other features of MPI-2 (e.g. Remote Memory Access).

New collective calls in MPI-3

- Collective calls (MPI_Bcast, MPI_Reduce, etc) are very often performance bottlenecks in MPI codes. For Exascale, with potentially millions of process, their impact could be serious.
- MPI-3 has introduced several enhancements to minimise performance loss due to collectives. These include:

1. Non-blocking collectives
2. Neighbourhood collectives.

Non-blocking collectives

- Work in the same way to the usual blocking collectives, except that they return almost immediately after being called, i.e. a task does not wait for other tasks to make the call.
- Naming convention just like non-blocking point-to-point calls: MPI_Iallreduce, MPI_Ibarrier, MPI_Ibcast ..
- Used with MPI_Test or MPI_Wait to increase overlap of calculation and computation.

Neighborough collectives

- A special type of collective call for sparse communication patterns, i.e where communications occur between a few processes in a communicator.
- In a neighbourhood call each process makes the call but communication only occurs between nearest neighbours.

Example:

```
MPI_Neighbor_allgather(void* sendbuf, int sendcount,
 MPI_Datatype sendtype, void* recvbuf, int recvcount,
 MPI_Datatype recvtype, MPI_Comm comm)
```

This sends the same data element to all neighbor processes and receives a distinct data element from each of the neighbors.

MPI-4.0

Latest release June 9, 2021.

Highlights include:

- Large-count versions of many routines to go beyond int32 (i.e. int or INTEGER).
- Persistent collectives (persistent P2P in 3.0).
- Partitioned communications
- Alternative ways to initialize MPI (Sessions)
- Info assertions and error handling

MPI 4.0 features

Persistent collectives

– Many programs execute a collective operation with the same arguments many times.

– With persistent collectives a programmer can specify operations which can be used frequently.

– Should be able to offer large performance benefits.

Sessions

Will allow a more dynamic model for creating MPI tasks:

– No more implicit MPI_COMM_WORLD

– Resource isolation

– Create communicators without parent communicators

– Re-initialization of MPI.

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

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