Parallel Computing
Introduction to MPI

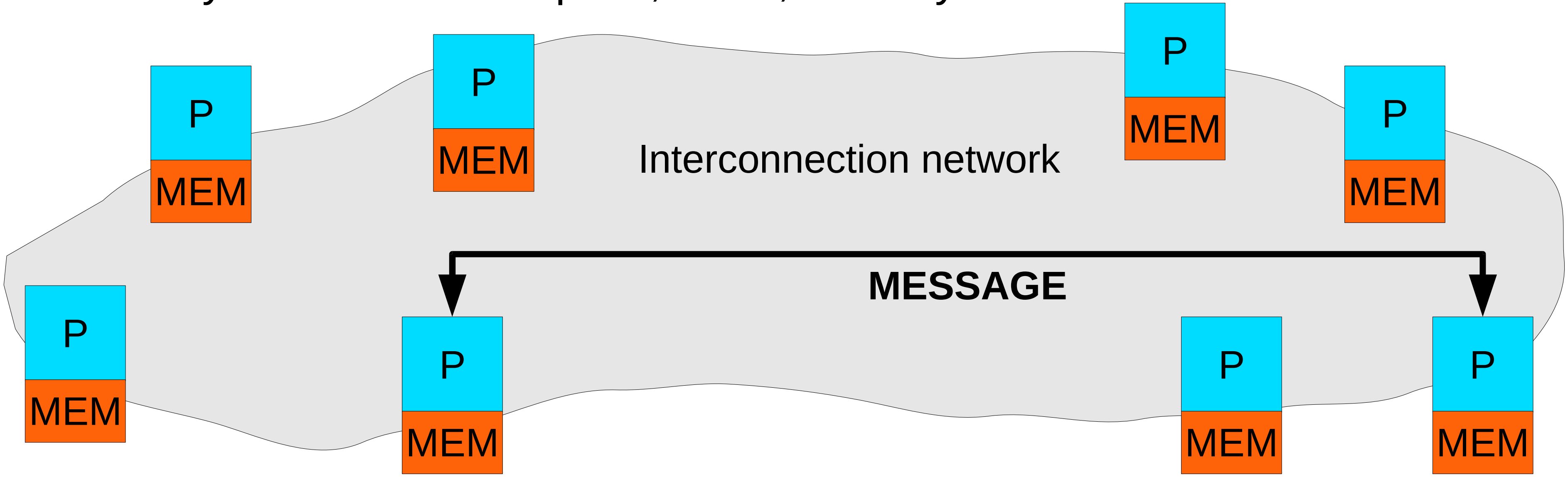
Marco Grossi
mgrossi@ectstar.eu

Ricerca ECT*, FBK Trento

Secondo corso di formazione
"Calcolo Parallelo su Grid (CSN4cluster)"
September 26th-28th, 2011
Parma

More about Message Passing paradigm

- **1** program
- **n** processes
  - Each process has it's own private address space
  - The communication between the processes it's performed with the exchange of messages
    - A message payload can contain packed data structures, but also synchronization request, … so, not only data



More about Message Passing paradigm

- **Suitable for distributed and shared memory architectures**
  - Each Processing Nodes of the CSN4 cluster has a shared memory architecture with 8 cores(2 x quadcore CPU)
  - You can start one process on each core
    - Each process has it's own private address space
    - No shared data structures available for process communication, only message passing
- **S**ingle **P**rogram **M**ultiple **D**ata(SPMD) approach(mainly)
  - Each process execute the same program but with different input

More about Message Passing paradigm

- A software library it's in charge of exchange the messages between the processes
  - If we start **n** process, we need an unique identifier in orther to distinguish one process from an other
  - When you send a message you have to specify
    - the identifier of the destination processes
      - should be more than one
    - the buffer containing the data to send
    - data size and data type
    - where the data will be left on the receiving side
  - When you want to receive a message you have to specify
    - the identifier of the sender
    - receive buffer
    - data type and size

Message Passing Interface (MPI)

- MPI is not a "complete" standard, but
  - it is a specification for **APIs** that allow many workers to communicate (**distributed memory system**)
    - it guarantees the portability for almost every distributed memory architecture
  - it provides a **language-independent** communication protocol
    - Bindings for C, C++, Fortran (and correlated languages)
- **Both cooperative (point-to-point and collective) and one-sided** communications are supported
- **Several implementations**, depending on the hardware (mainly developed by cluster vendors)
  - it guarantees the best performance on a specific hardware

MPI - Implementation

- **Different implementations**:
  - OpenMPI: http://www.open-mpi.org
  - MPICH: http://www.mcs.anl.gov/research/projects/mpich2
  - Custom MPI implementation for specific clusters (Cray, IBM, ...) and networks
  - Commercial implementations from HP, Intel, Microsoft, ...
- Each implementation decides the low-level treating of the data, depending of the hardware, in order to have the best possible performances
  - Transparent to the user
  - **Different performance (and results)** depending on the implementation: be aware of your MPI implementation!

MPI – Program structure

MPI include file

Declarations, prototypes, etc.

Program Begins

.
.
.

Serial code

Initialize MPI environment ***Parallel code begins***

.
.
.

Do work & make message passing calls

.
.
.

Terminate MPI environment ***Parallel code ends***

.
.
.

Serial code

Program Ends

Let's start!

hello_world.c

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

#include <mpi.h>

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

  int err, my_rank, comm_size;

  err = MPI_Init(&argc, &argv);

  err = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  printf(”Hello World from process %d of %d!\n”,
         my_rank, comm_size);

  err = MPI_Finalize();

  exit(0);
}
```

```
Compile → mpicc -W -Wall hello_world.c -o hello_world
Execute → mpirun -np k ./hello_world
```

MPI – C binding details

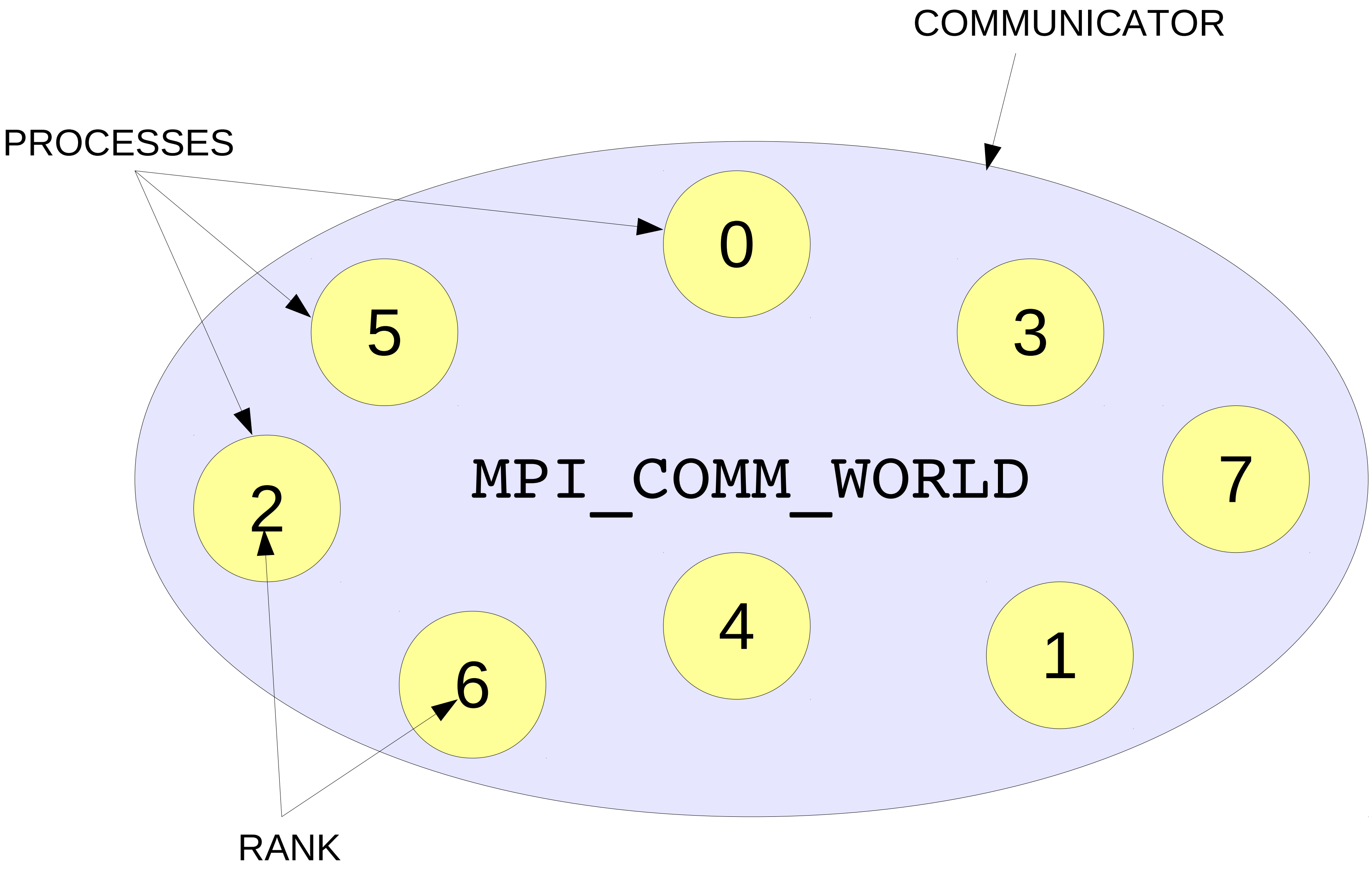
```
int MPI_Xxxx(...)
```

- All MPI names have an `”MPI_”` prefix
  - Defined constants are in all capital letters
- Programs must not declare variables or functions with names beginning with the prefix `”MPI_”`.
  - To also support the profiling interface, avoid the `”PMPI_”` prefix
- Almost all C functions return an error code
  - The successful return code will be `MPI_SUCCESS`, but failure return codes are implementation dependent
- Array arguments are indexed from zero
- Logical flags are integers with value `0` meaning `“false”` and a non-zero value meaning `“true”`

Rank & Communicator

- If we start the hello_world program with 8 processes, after calling MPI_Init each process:
  - has it's own unique identifier, called **rank**(an integer from 0 to 7)
  - belongs to the default **communicator**: `MPI_COMM_WORLD`
- A communicator it's an opaque object of type `MPI_Comm`
  - It's a group of procs that can exchange data between each other
  - opaque object: size and shape are not visible to the users; accessed by handles, which exist in user space
  - `MPI_COMM_WORLD` it's the default communicator, available from the call to `MPI_Init` until `MPI_Finalize`
  - You can create multiple communicator of different size(a process may have different rank on different communicator)
    - Requires special inter-communicator routines

Rank & Communicator



MPI – Point-to-point communication modes

blocking standard
non-blocking standard
buffered
ready
synchronous

send

blocking
non-blocking

recv

blocking

combined sendrecv

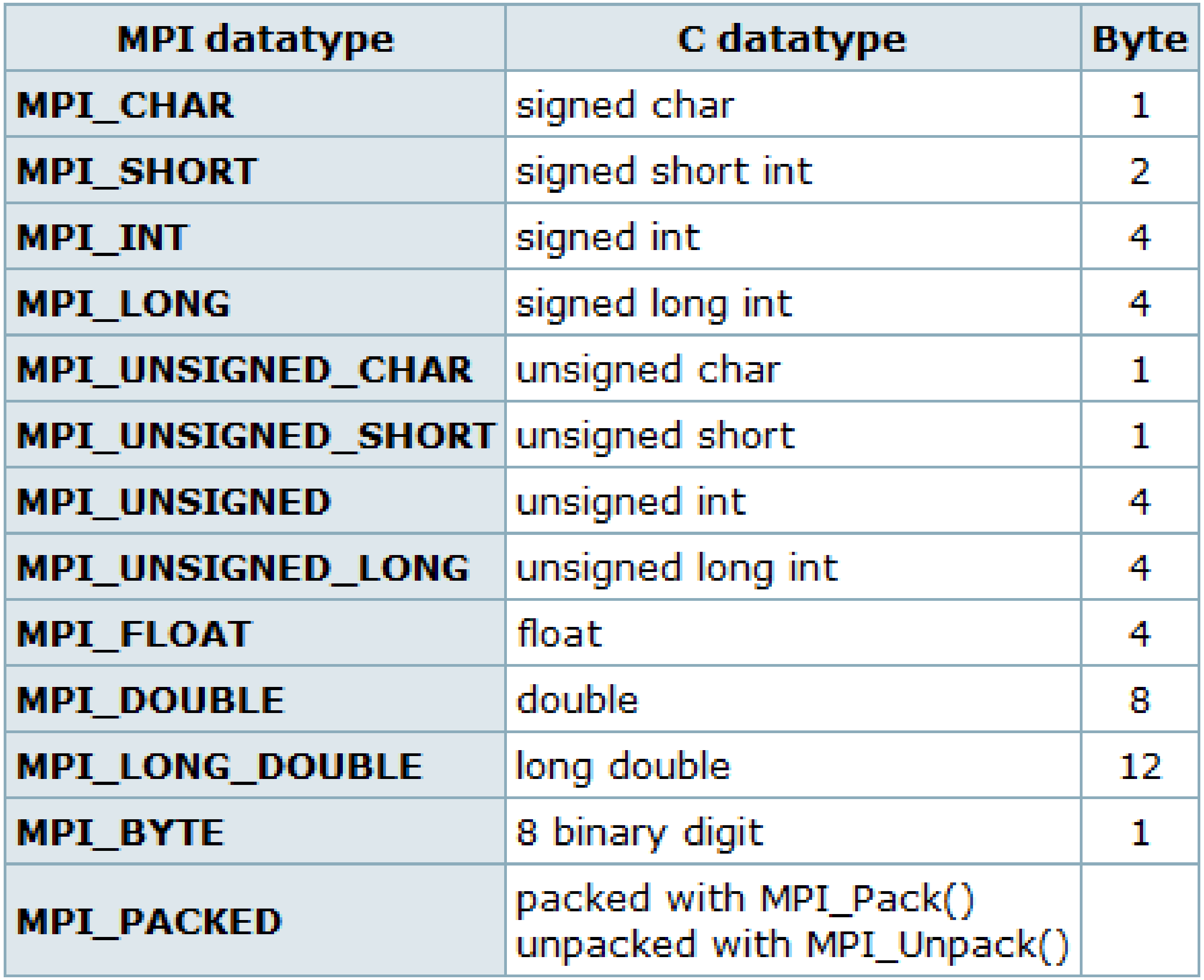
MPI – blocking: send & recv

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm)
```

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

`buffer`	a pointer to data to send or recv
`datatype`	type of the element in the buffer
`count`	how many element in the buffer to send or recv
`dest/source`	rank of the process to send to or recv from
`tag`	each rank has different mailbox where the message can be received; the `tag` it's the identifier of a specific mailbox. Normally it's set to `0`
`comm`	communicator of the sender and receiver
`status`	data structure that contain details on sender, tag and data count. Normally it's set to `MPI_STATUS_IGNORE`

MPI – C data types



MPI datatype	C datatype	Byte
MPI_CHAR	signed char	1
MPI_SHORT	signed short int	2
MPI_INT	signed int	4
MPI_LONG	signed long int	4
MPI_UNSIGNED_CHAR	unsigned char	1
MPI_UNSIGNED_SHORT	unsigned short	1
MPI_UNSIGNED	unsigned int	4
MPI_UNSIGNED_LONG	unsigned long int	4
MPI_FLOAT	float	4
MPI_DOUBLE	double	8
MPI_LONG_DOUBLE	long double	12
MPI_BYTE	8 binary digit	1
MPI_PACKED	packed with MPI_Pack() unpacked with MPI_Unpack()	

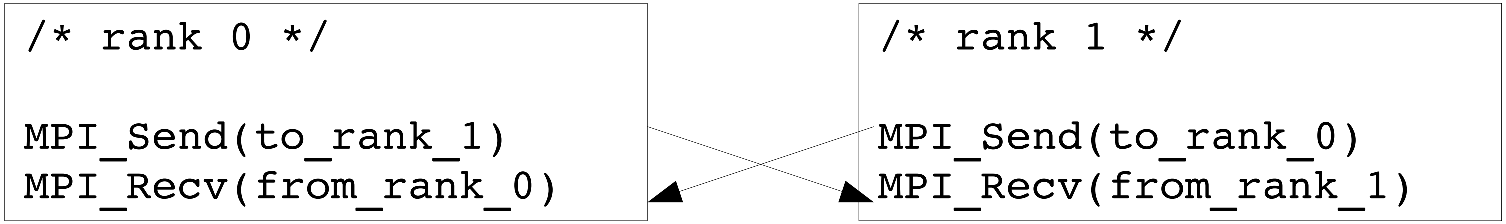
MPI – combined sendrecv

- In orther to avoid deadlock due to the lack of free buffer space you can use the Sendrecv primitive

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

MPI – send/recv pattern

:-(

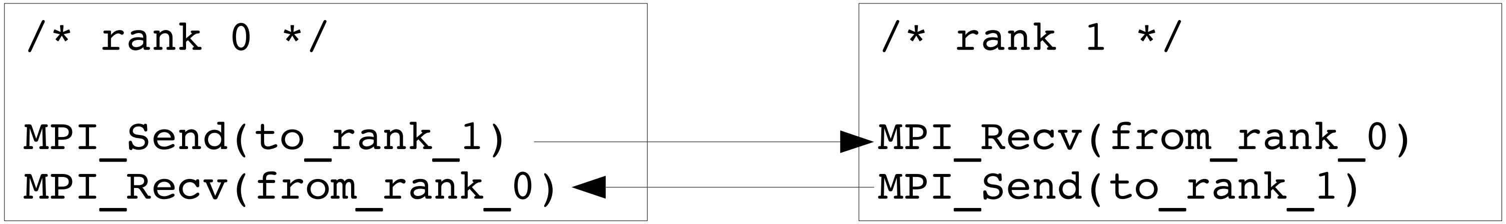


```
/* rank 0 */

MPI_Send(to_rank_1)
MPI_Recv(from_rank_0)
```

```
/* rank 1 */

MPI_Send(to_rank_0)
MPI_Recv(from_rank_1)
```

This could lead to a deadlock → Game Over

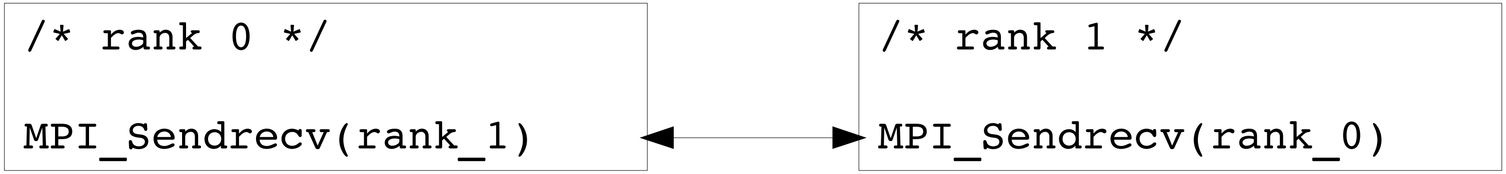
:-)



```
/* rank 0 */

MPI_Send(to_rank_1)
MPI_Recv(from_rank_0)
```

```
/* rank 1 */

MPI_Recv(from_rank_0)
MPI_Send(to_rank_1)
```

:-)



```
/* rank 0 */

MPI_Sendrecv(rank_1)
```

```
/* rank 1 */

MPI_Sendrecv(rank_0)
```

MPI – Blocking vs. non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- Blocking:
  - **A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse**. Safe means that modifications will not affect the data intended for the receive task. **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** which means there is handshaking occurring with the receive task to confirm a safe send.
      - **asynchronous(standard mode)** if a system buffer is used to hold the data for eventual delivery to the receive.

MPI – Blocking vs. Non-blocking

- Non-blocking:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - **Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen**.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact 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 and exploit possible performance gains.

MPI – non-blocking: send & recv

```
int MPI_Isend(void *buffer, int count, MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Irecv(void *buffer, int count, MPI_Datatype datatype,
              int source, int tag,
              MPI_Comm comm, MPI_Status *status, MPI_Request *request)
```

`request` The request can be used later to query the status(with `MPI_Test`) of the communication or wait(with `MPI_Wait`) for its completion.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)

    It's a blocking call
```

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

    It's a non-blocking call
    If (flag != 0) than the operation identified by request is completed
```

MPI – systolic exchange with non-blocking primitives

- All rank have to send to their neighbours on the right and receive from the left

```
int neigh_left, neigh_right;
double *buf_tx[BUF_SIZE], *buf_rx[BUF_SIZE];

[…]

/* Buffer allocation, aligned at page size */
if (posix_memalign(&buf_tx, sysconf(_SC_PAGESIZE), RXBUF_SIZE) != 0) {
    perror(...); exit(-1);
}

[…]

MPI_Request req_recv;

for (...) {
  MPI_Irecv(buf_rx, BUF_SIZE, MPI_DOUBLE, neigh_left, 0,
            MPI_COMM_WORLD, &req_recv);
  [… calc …]
  MPI_Send(buf_tx, BUF_SIZE, MPI_DOUBLE, neigh_right, 0, MPI_COMM_WORLD);
  MPI_Wait(&req_recv, MPI_STATUS_IGNORE);
}
```

MPI – Persistent communication request

- In the previous example we had continued to issue the `Irecv+Send+Wait` call sequence with the same buffer and data count/type to the same src/dest ranks
- In such a situation, **it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages**
- This construct allows reduction of the overhead
- It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa

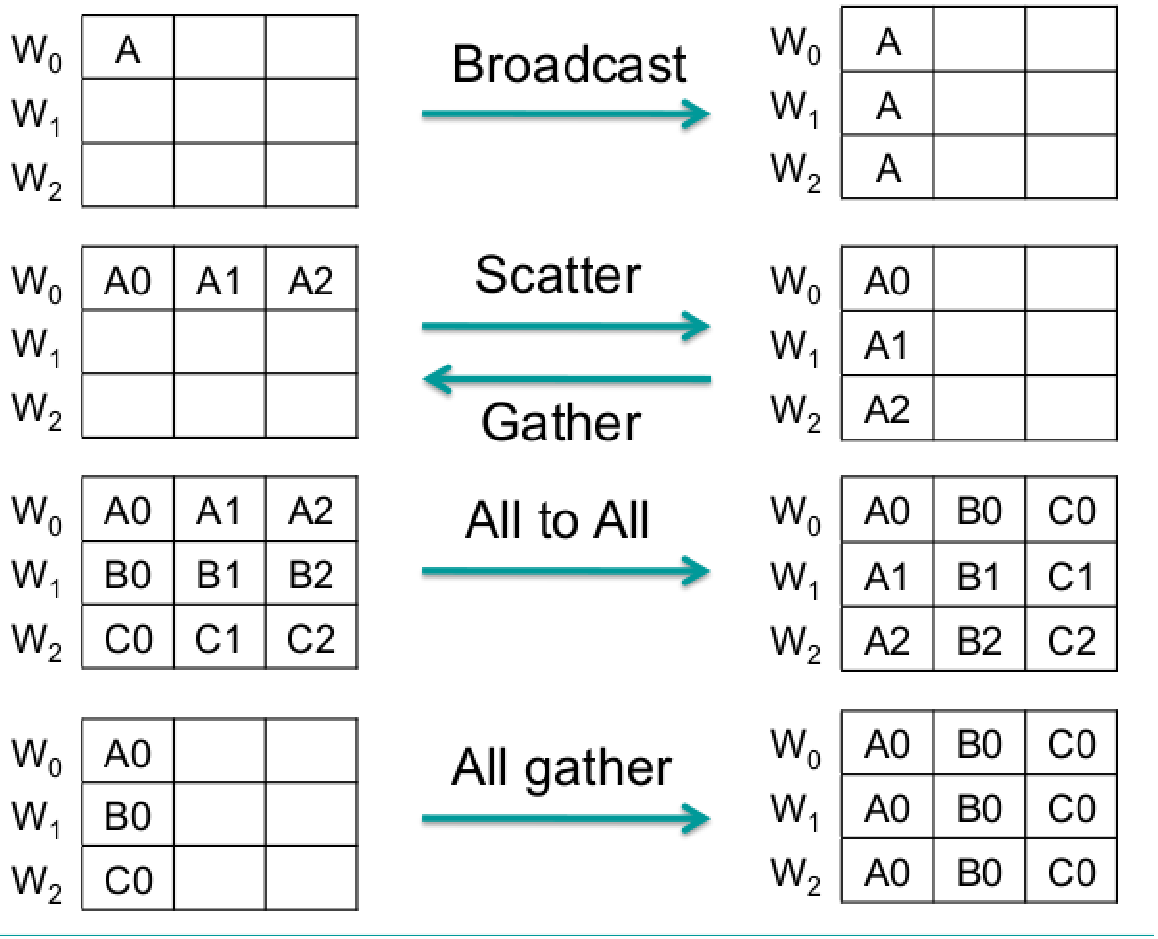
MPI – systolic exchange with persistent communication

```
/* Buffer allocation and all other initializations */

[…]

MPI_Request request_array[2]; /* index 0: recv -  index 1: send */

MPI_Recv_init(buf_tx, BUF_SIZE, MPI_DOUBLE, neigh_right, 0,
              MPI_COMM_WORLD, &request_array[0]);
MPI_Send_init(buf_tx, BUF_SIZE, MPI_DOUBLE, neigh_right, 0,
              MPI_COMM_WORLD, &request_array[1]);

[…]

for (...) {
  MPI_Start(&request_array[0]); /* Irecv */

  [… calc …]

  MPI_Start(&request_array[1]); /* Isend */

  /* Wait for completion of Irecv+Isend */
  MPI_Waitall(2, &request_array, MPI_STATUSES_IGNORE);
}
```

MPI – Collective communications

- All or None:
  - **Collective communication MUST involve all processes in the scope of a communicator**.
  - It is the programmer's responsibility to insure that all processes within a communicator participate in any collective operations
- Types of Collective Operations:
  - **Synchronization** - processes wait until all members of the group have reached the synchronization point
  - **Data Movement** - broadcast, scatter/gather, all to all
  - **Collective Computation (reductions**) - collects data from all ranks and perform an operation (min, max, add, multiply, etc.) on that data; return the result to one rank or all communicator
- **Collective operations are blocking**

MPI – Collective communications



MPI - Barrier, Reduce

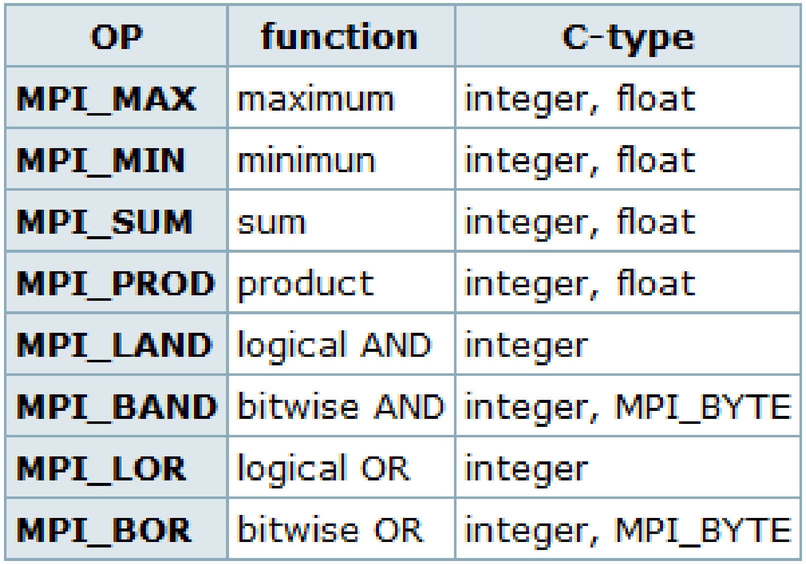
```
int MPI_Barrier(MPI_Comm comm)
```

Blocks until all processes have reached this routine

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

Combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value only in the output buffer of the process with rank **root**.

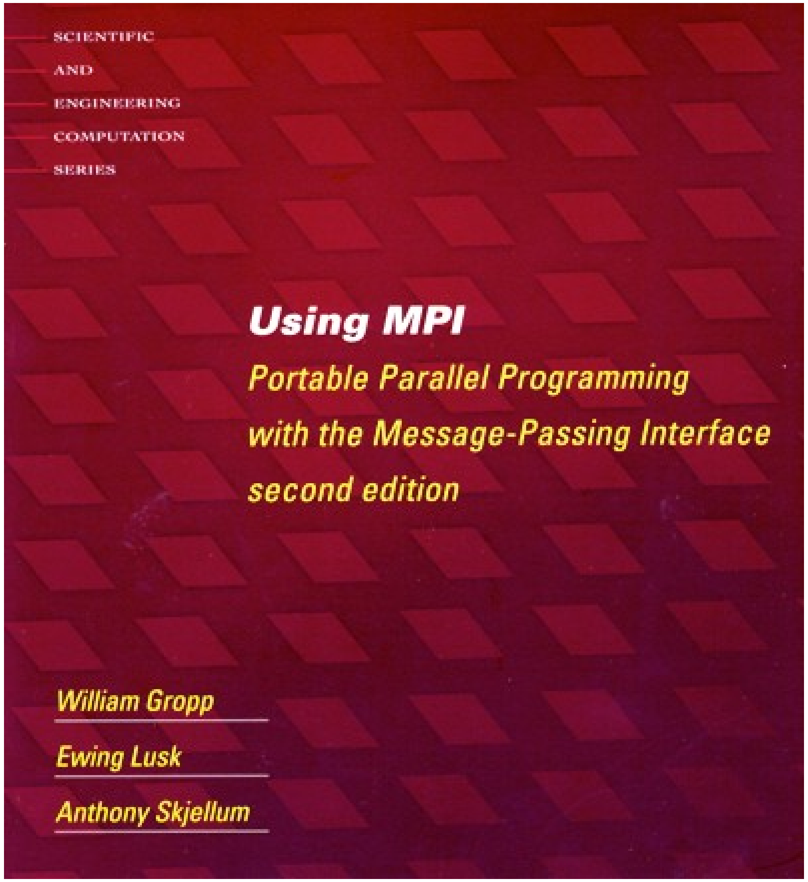
The reduction operation can be either one of a predefined list of operations, or a user-defined operation(see MPI_Op_create).



OP	function	C-type
MPI_MAX	maximum	integer, float
MPI_MIN	minimun	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bitwise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bitwise OR	integer, MPI_BYTE

References

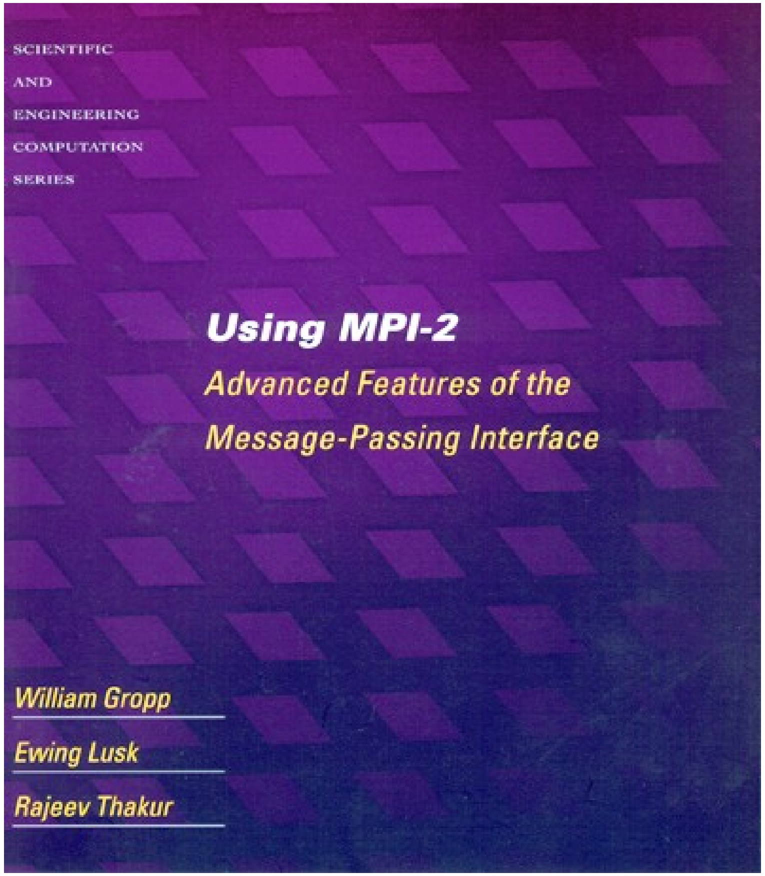
- https://computing.llnl.gov/tutorials/mpi/
- http://www.open-mpi.org/
- http://www.mcs.anl.gov/research/projects/mpi/usingmpi/
- http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI
Portable Parallel Programming
with the Message-Passing Interface
second edition

William Gropp
Ewing Lusk
Anthony Skjellum



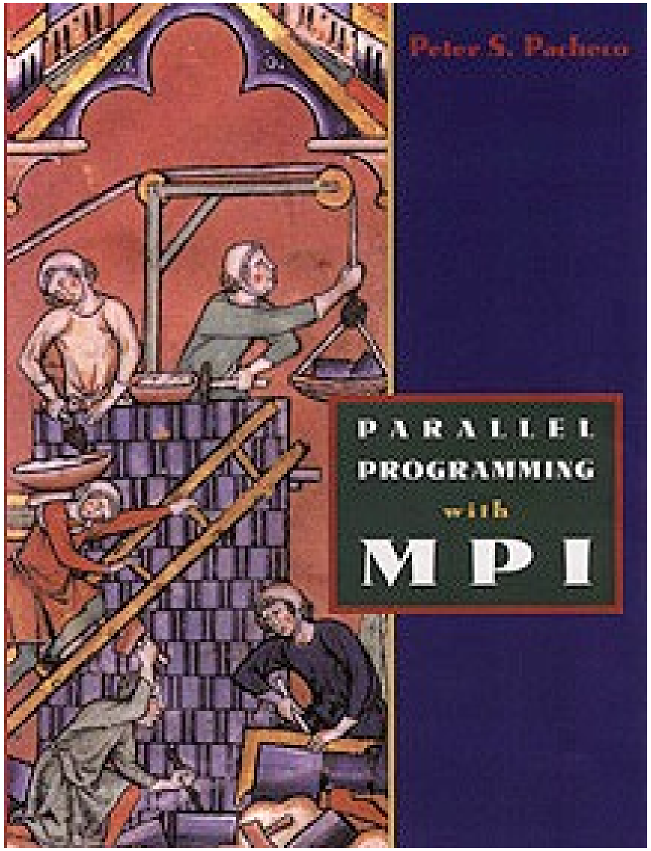
SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

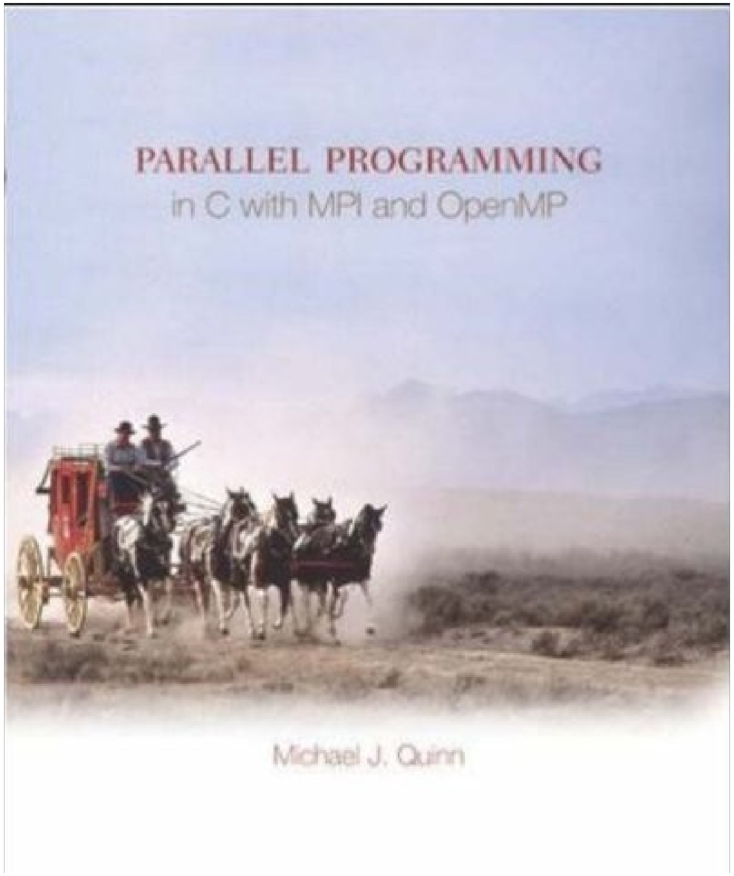
Using MPI-2
Advanced Features of the
Message-Passing Interface

William Gropp
Ewing Lusk
Rajeev Thakur

References

- http://www.cs.usfca.edu/~peter/ppmpi/





Your questions & hints

Thank you for your attention!
For any questions and hints
please send an email to

`mgrossi@ectstar.eu`