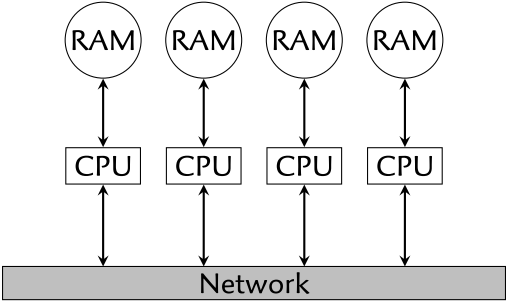
Lecture #2: Message Passing Interface

September 19, 2023

Distributed Memory Machine



- Each processor has its own memory
- Communication $=$ messages exchanged on the network

Message Passing Interface

In the early days

- IBM, Cray, Silicon Graphics, ... had **their own** communication middleware
- Or course, not compatible with each others
- Serious obstacle to code portability

In 1994

- v1.0 of the **MPI specification**
- Consortium with the usual suspects (Cray, IBM, ...)
- New standard for writing "Message-passing programs". Goals:
  - **Widely-used**
  - **Practical**, **portable**, **efficient**, and **flexible**

Some Context

- Single Program / Multiple Data
- Many **processes**, all running the same code
- Each process has a (unique) **rank**

$\boxed{P_0}$

$\boxed{P_1}$

$\boxed{P_2}$

- Very flexible & portable

Some Context

- Single Program / Multiple Data
- Many **processes**, all running the same code
- Each process has a (unique) **rank**

$\boxed{P_0}$ $\boxed{P_1}$ $\boxed{P_2}$

$\boxed{P_3}$ $\boxed{P_4}$ $\boxed{P_5}$

- Very flexible & portable

Some Context

- Single Program / Multiple Data
- Many **processes**, all running the same code
- Each process has a (unique) **rank**

P_0 P_1 P_2

P_3 P_4 P_5

P_6 P_7 P_8

P_9 P_a P_b

P_c P_d P_e

P_f P_g P_h

- Very flexible & portable

What MPI offers

1. Runtime Environment

- The `mpiexec` program starts an MPI application
- Usual arguments
  - Program to execute
  - # processes to start
  - List of target nodes
  - (optional) process placement directives

2. Library

- `mpi.h` (≈ 370 functions...)

3. Compiler Wrapper

- Use `mpicc` to compile MPI applications
  - It invokes `gcc` with the right arguments

MPI Startup and Termination

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    /* your code goes here */

    MPI_Finalize();
}
```

Dead Weight of History

- `MPI_Init` used to inspect/modify `argc, argv` in early MPI implementations
- It is legal (and simpler) to call `MPI_Init(NULL, NULL);`

MPI Startup and Termination

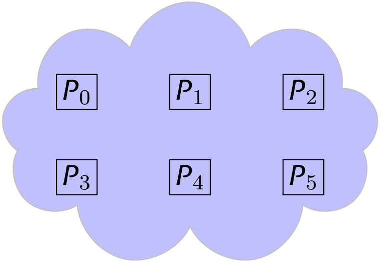
Library functions

- `int MPI_Init(int *argc, char ***argv)`
- `int MPI_Finalize()`

Remarks

- (almost) All MPI functions return an error code
- Default MPI error handler = ☠ **KILL** ☠ your application
  - ⇝ No need to check the error codes...
- Default behavior can be changed

Communicators



- Process group + communication context
  - Ranks start at zero
- Opaque object of type `MPI_Comm`
- On startup, `MPI_COMM_WORLD` contains all processes
- (advanced technique) new communicators can be created

MPI Library functions

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`

Messages

- **data** (array of values) + **envelope** ("metadata")

Message data

buffer	Memory address of the first element
count	Number of elements
datatype	Type of the elements

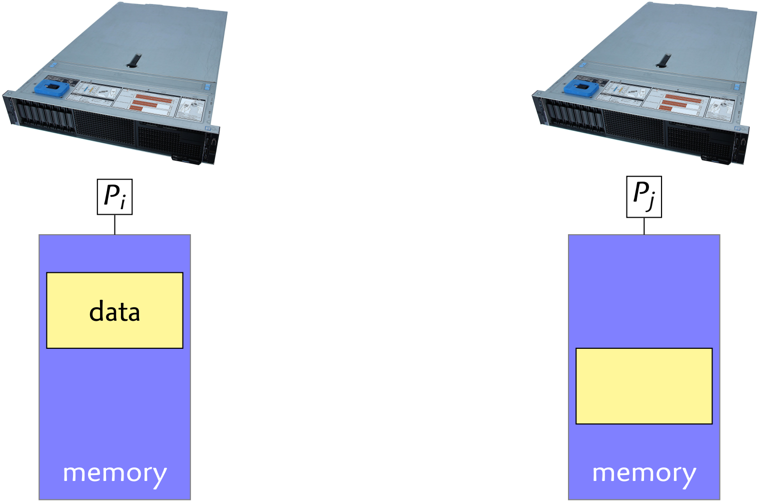
Envelope

source	rank of the sending process
destination	rank of the receiving process
tag	arbitrary integer ("nature" of the message)
communicator	messages belong to a single communicator

Basic Data Types

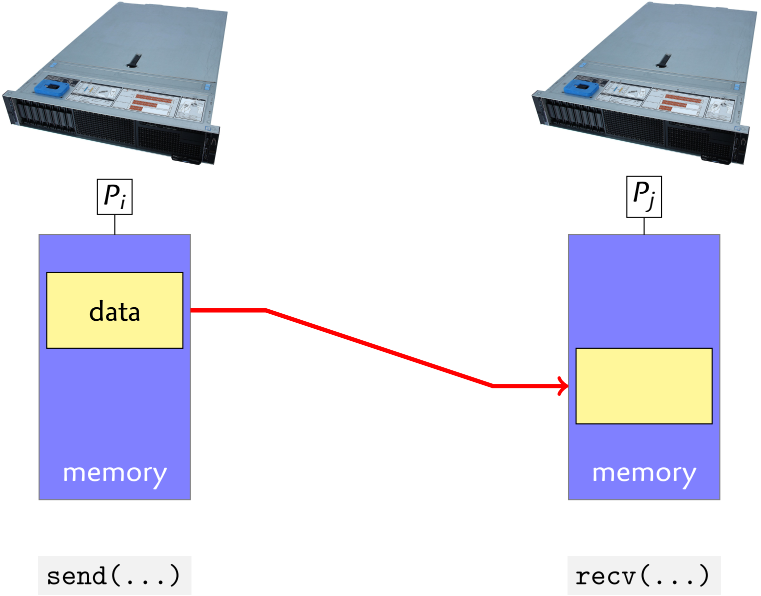
MPI type	C type
`MPI_CHAR`	`char`
`MPI_SHORT`	`short int`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_LONG_LONG_INT`	`long long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_INT8_T`	`int8_t`
`MPI_INT16_T`	`int16_t`
`MPI_INT32_T`	`int32_t`
`MPI_INT64_T`	`int64_t`
`MPI_BYTE`	

Point-to-Point Communication

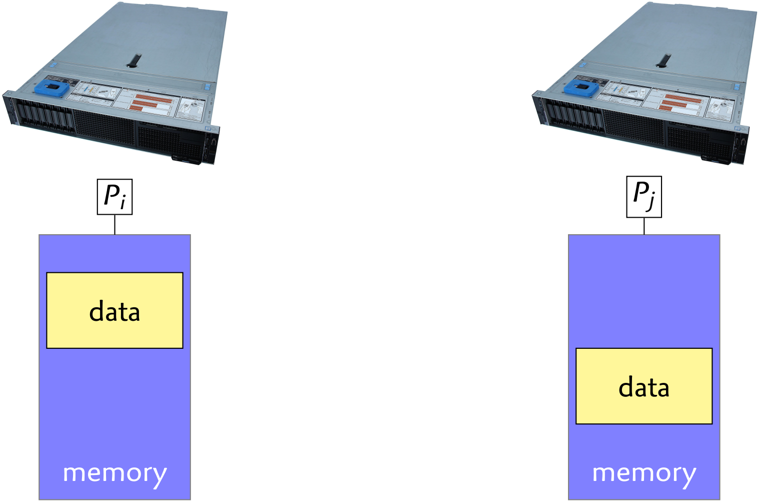


`send(...)` `recv(...)`

Point-to-Point Communication



Point-to-Point Communication



`send(...)`

`recv(...)`

Point-to-Point Communication (sending)

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

Comments

- Read a message in memory from send buffer
  - `count` elements of type `datatype` at address `buf`
- Send it to process of rank `dest` in communicator `comm`
- Message “labelled” with `tag`
  - $0 \leq$ `tag` $< 2^{15}$
- Function returns when message has been entirely read
  - The send buffer can be safely overwritten

Point-to-Point Communication (sending)

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

MPI data types

- With basic datatypes:
  - Message = array of **contiguous** values in memory
- Create more complex MPI types using MPI functions
  - Array of `struct`
  - Non-contiguous (jumps between values)
  - See MPI spec ; out of the scope of this lecture

Point-to-Point Communication (receiving)

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

Comments about `MPI_Recv`

- Wait for matching message
  - Labelled `tag` from process `source` in communicator `comm`
- Write it in memory in receive buffer (at address `buf`)
- Write metadata in `status`
- Size of the message must be $\leq$ `count`
  - Actual number of received values may be smaller

Point-to-Point Communication (receiving, continued)

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

Can use **wildcards**

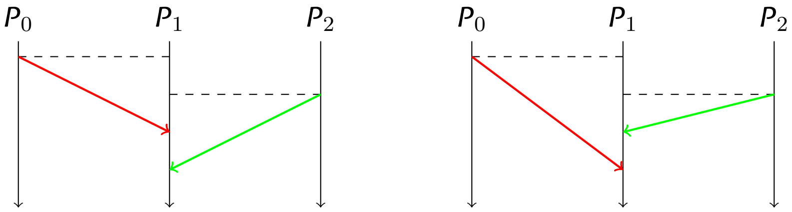
- `tag == MPI_ANY_TAG`
- `source == MPI_ANY_SOURCE`

Inspecting metadata in `status`

- Not written if `status == MPI_STATUS_IGNORE`
- `MPI_status` is a `struct`
  - fields `MPI_SOURCE` and `MPI_TAG`
- ```c
  int MPI_Get_count(MPI_Status *status,
                    MPI_Datatype datatype, int *count)
  ```
  ```

Semantics of Point-to-Point Communication

- Messages are not lost
- Messages are not duplicated
- Messages do not overtake
  - P_i sends two messages M, M' in succession to P_j
  - P_j posts a receive maching both messages
  - P_j receives M first
- No guarantees when senders are different
  - Inherent nondeterminism



Minimal MPI

- `MPI_Init()`
- `MPI_Comm_size()`
- `MPI_Comm_rank()`
- `MPI_Send()`
- `MPI_Recv()`
- `MPI_Finalize()`

Major advantages

- Works over any kind of weird network
- Don't need to care about network addresses

MPI Speed on Grid5000

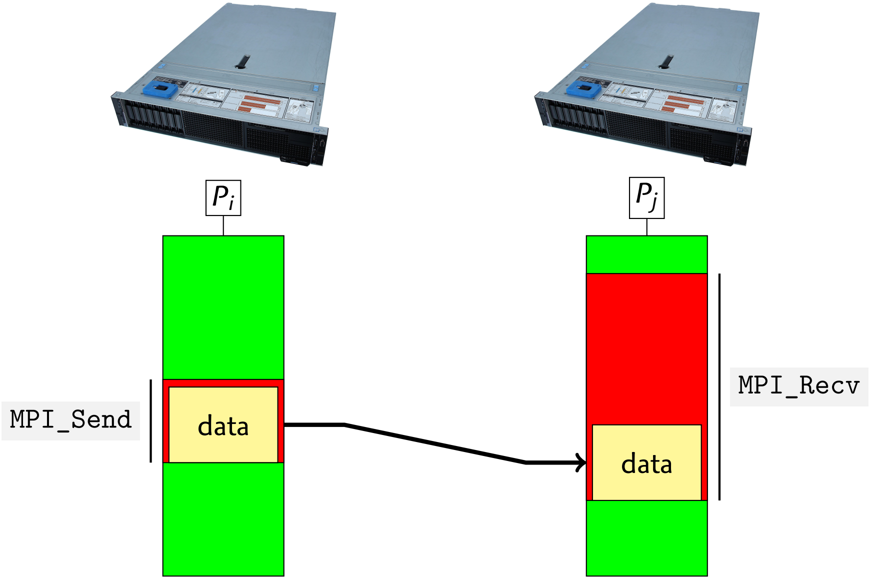
```c
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);

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

Cluster	Site	Year	Interconnect	Bandwith	Latency
PPTI	jussieu	20??	1Gbit ethernet	64MB/s	50.0μs
sagitaire	lyon	2006	1Gbit ethernet	116MB/s	65.0μs
taurus	lyon	2012	10Gbit ethernet	1156MB/s	25.9μs
gros	nancy	2019	25Gbit ethernet	2480MB/s	8.9μs
grcinq	nancy	2013	56Gbit InfiniBand	2488MB/s	1.2μs
grimoire	nancy	2016	56Gbit InfiniBand	4248MB/s	1.1μs
drac	grenoble	2015	100Gbit InfiniBand	7760MB/s	1.7μs
grvingt	nancy	2018	100Gbit OmniPath	12100MB/s	0.9μs

(measure : ping-pong between two nodes)

Blocking Receive



More Point-to-Point Functions

Receiving

Blocking	`MPI_Recv`
Non-blocking	`MPI_Irecv`

More Point-to-Point Functions

Receiving

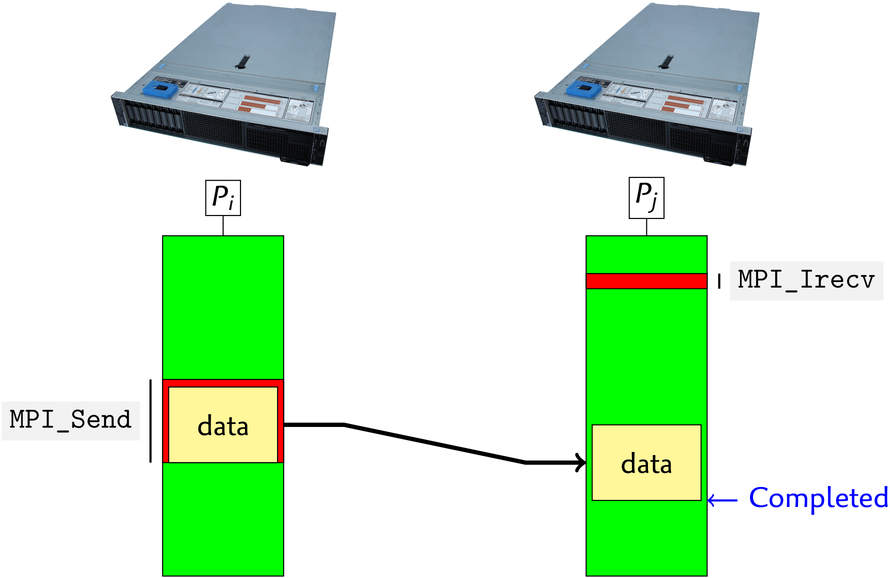
Blocking	`MPI_Recv`
Non-blocking	`MPI_Irecv`

Sending

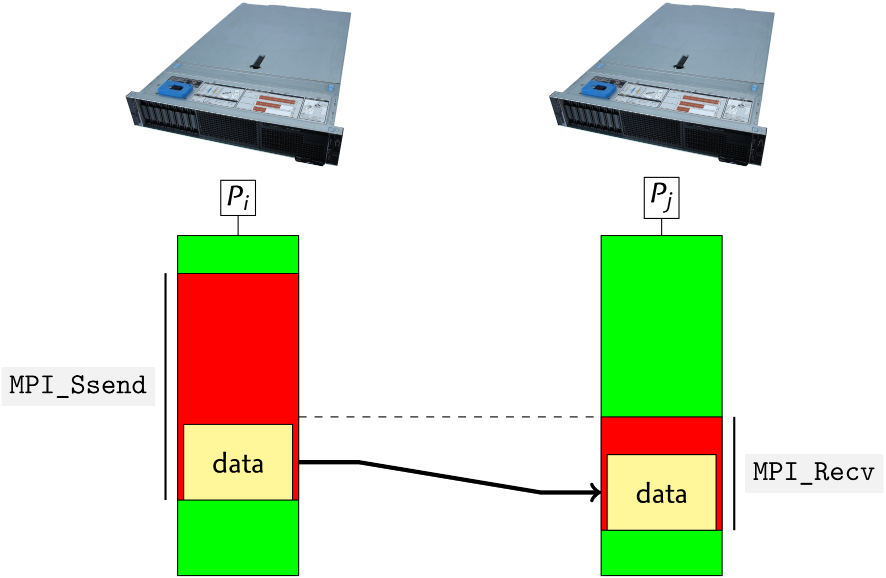
	Synchronous	Buffered	Standard
Blocking	`MPI_Ssend`	`MPI_Bsend`	`MPI_Send`
Non-blocking	`MPI_Issend`	`MPI_Ibsend`	`MPI_Isend`

- I = "Immediate"
- Non-blocking functions return as soon as possible
- Operation runs in the background
  - $\Rightarrow$ Overlap communication with computation

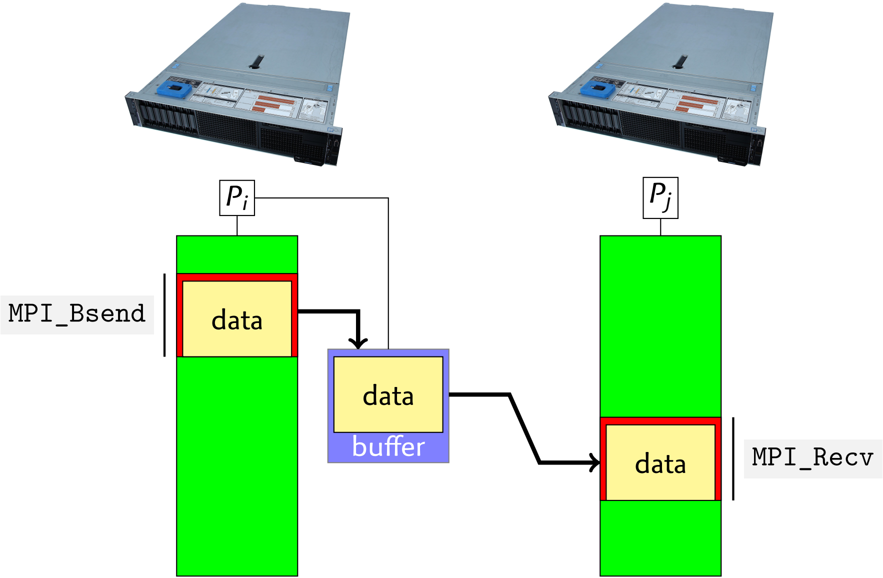
Non-Blocking Receive



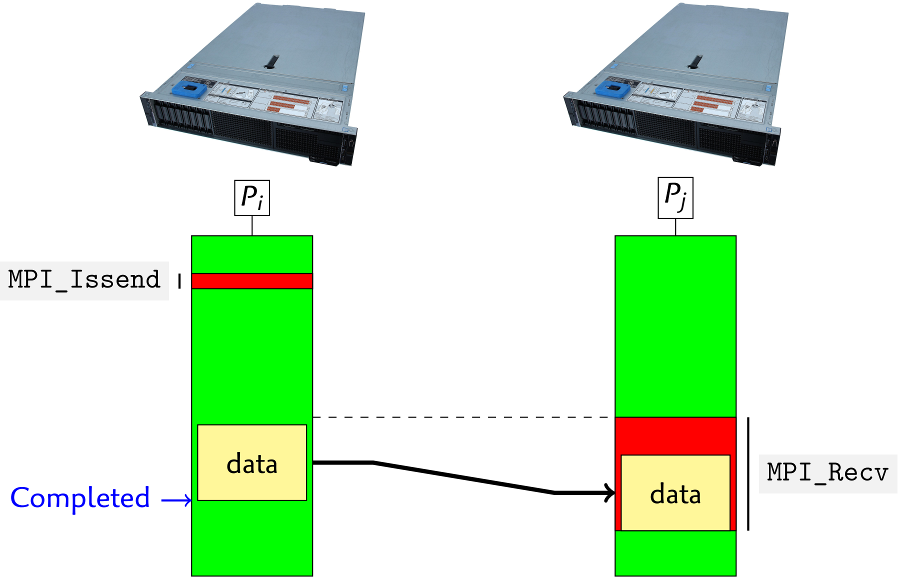
Blocking Synchronous Send



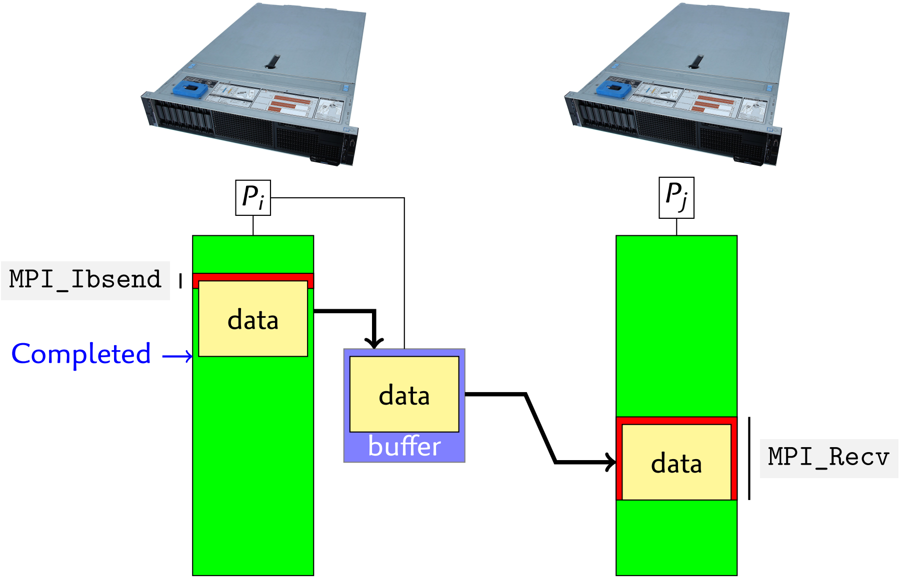
Blocking Buffered Send



Non-Blocking Synchronous Send



Non-Blocking Buffered Send



Summary

Blocking / Non-blocking

- blocking send: send buffer can be overwritten
- blocking Receive: data is ready in receive buffer
- non-blocking: buffer must not be touched until completion

Send: Synchronous / Buffered / Standard

- Buffered: may complete before a matching receive is posted
  - Needs more memory
- Synchronous: completes only if a matching receive started
  - No need for additional memory
  - **Synchronizes** processes
- Standard: either Buffered or Synchronous (best perf.)

Non-Blocking Functions

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

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Request *request)

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

Rules

- Non-blocking operations use resources
  - Maintain state in an `MPI_Request`
- They **must** be waited for
  - `MPI_Wait` releases resources
  - Buffer can be read/overwritten after waiting

MPI : Simultanenous Send / Receive

- Recurring pattern
  - Shift operation across a chain of processes
- Naive solution (`MPI_Send; MPI_Recv`) ⇝ ☠ deadlock ☠

```c
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);
```

- `source == dest` is legal
- The send and receive buffers must not overlap

```c
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
                         int dest, int sendtag, int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status);
```

- May allocate memory

Collective Operations

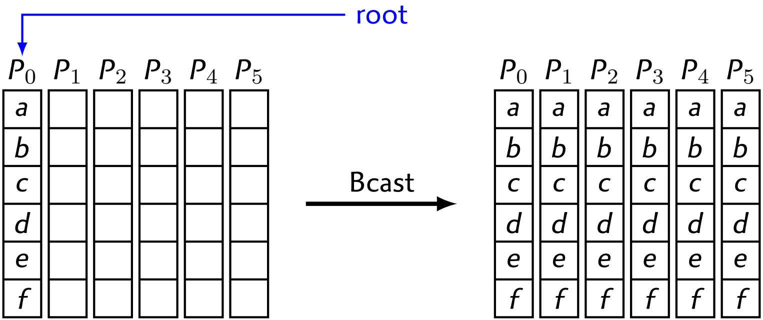
- Point-to-Point operations (two processes)
- Collective operations (all processes)
  - Broadcast, Gather, Scatter, Reduction, ...

Rule

- Collective communications take place in a communicator
- All processes of a communicator **MUST** call the function
  - It requires some action from them
  - They can't do it if you don't allow them to

Broadcast

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



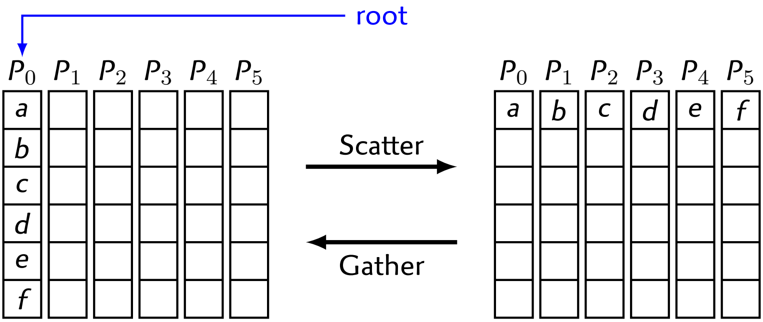
Remark

- All processes must know `count` in advance
  - Do `MPI_Bcast` with a single `int` first

Gather / Scatter

```c
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm);

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



Gather / Scatter

Comments about the Arguments

Some arguments only make sense when `rank == root`

- Gather: the receive buffer
- Scatter: the send buffer
- **Ignored** otherwise

Important point

- `recvcount == sendcount` = #elements **per process**

`root` Copy `count` items from `sendbuf` to `recvbuf`

- Can be disabled with `sendbuf == MPI_IN_PLACE`
  - Ignore send buffer
  - “own” data **already** at the right place in the receive buffer

Gather / Scatter

Comments about array sizes

Big array divided into **equally sized** slices

- `count` elements per process
- `p` processes
- Total size of the array is `count` $\times$ `p`

Potential mismatch

- #process imposed by the **hardware**
- Array size imposed by **user input**

Two potential solutions

1. Padding
2. `MPI_Gatherv` and `MPI_Scatterv`

Array of Size n Must be Divided in p Slices

Idea #1: pad the array

- Slices of size $k = \lceil n/p \rceil =$ `(n + p - 1) / p`
- Allocate a (larger) array A of size kp
- Only `A[0:n]` contains meaningful data

P_0				P_1				P_2				P_3			
0	1	2	3	4	5	6	7	8	9	a	b	c	d		

Alternative

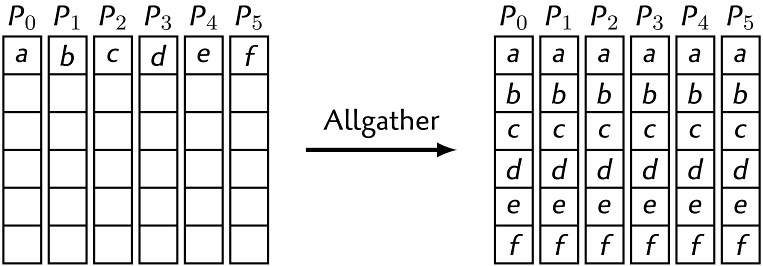
- Process i has `A[i*n/p : (i+1)*n/p]`

P_0			P_1				P_2			P_3			
0	1	2	3	4	5	6	7	8	9	a	b	c	d

- Requires the use of `MPI_Gatherv`, `MPI_Scatterv`

Gather-to-All

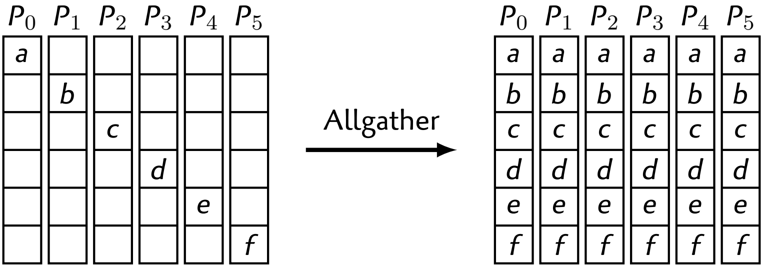
```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
```



- See also : `MPI_Allgatherv`

Gather-to-All

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



- See also : `MPI_Allgatherv`
- `sendbuf == MPI_IN_PLACE` (for all processes)
  - Send buffers ignored
  - **Read** from the **receive** buffers

All-to-All Scatter/Gather

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

P_0	P_1	P_2	P_3	P_4	P_5
a_0	b_0	c_0	d_0	e_0	f_0
a_1	b_1	c_1	d_1	e_1	f_1
a_2	b_2	c_2	d_2	e_2	f_2
a_3	b_3	c_3	d_3	e_3	f_3
a_4	b_4	c_4	d_4	e_4	f_4
a_5	b_5	c_5	d_5	e_5	f_5

$\xrightarrow{\text{Alltoall}}$

P_0	P_1	P_2	P_3	P_4	P_5
a_0	a_1	a_2	a_3	a_4	a_5
b_0	b_1	b_2	b_3	b_4	b_5
c_0	c_1	c_2	c_3	c_4	c_5
d_0	d_1	d_2	d_3	d_4	d_5
e_0	e_1	e_2	e_3	e_4	e_5
f_0	f_1	f_2	f_3	f_4	f_5

- See also : `MPI_Alltoallv`, `MPI_Alltoallw`
- `sendbuf == MPI_IN_PLACE` (for all processes)
  - Send buffers ignored
  - **receive** buffer sent then overwritten with received data

Network Torture Test

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

Settings

- 16 nodes, 4GB per node (64GB in total)

⇝ Each node sends/receives 256MB to each other node

Cluster	Site	Year	Interface	T	aggregated BW
PPTI	jussieu	20??	1Gbit ethernet	254s	250Mo/s

Network Torture Test

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

Settings

- 16 nodes, 4GB per node (64GB in total)
- ⇝ Each node sends/receives 256MB to each other node

Cluster	Site	Year	Interface	T	aggregated BW
PPTI	jussieu	20??	1Gbit ethernet	254s	250Mo/s
sagitaire	lyon	2006	1Gbit ethernet	?	

Network Torture Test

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

Settings

- 16 nodes, 4GB per node (64GB in total)

⇝ Each node sends/receives 256MB to each other node

Cluster	Site	Year	Interface	T	aggregated BW
PPTI	jussieu	20??	1Gbit ethernet	254s	250Mo/s
sagitaire	lyon	2006	1Gbit ethernet	?	
paravance	rennes	2015	10Gbit ethernet	14.9s	4Go/s

Network Torture Test

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

Settings

- 16 nodes, 4GB per node (64GB in total)

$\rightsquigarrow$ Each node sends/receives 256MB to each other node

Cluster	Site	Year	Interface	T	aggregated BW
PPTI	jussieu	20??	1Gbit ethernet	254s	250Mo/s
sagitaire	lyon	2006	1Gbit ethernet	?	
paravance	rennes	2015	10Gbit ethernet	14.9s	4Go/s
gros	nancy	2019	25Gbit ethernet	6.8s	9.4Go/s

Network Torture Test

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

Settings

- 16 nodes, 4GB per node (64GB in total)
- ⇝ Each node sends/receives 256MB to each other node

Cluster	Site	Year	Interface	T	aggregated BW
PPTI	jussieu	20??	1Gbit ethernet	254s	250Mo/s
sagitaire	lyon	2006	1Gbit ethernet	?	
paravance	rennes	2015	10Gbit ethernet	14.9s	4Go/s
gros	nancy	2019	25Gbit ethernet	6.8s	9.4Go/s
grcinq	nancy	2013	56Gbit InfiniBand	4.8s	13Go/s

Network Torture Test

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

Settings

- 16 nodes, 4GB per node (64GB in total)
- ⇝ Each node sends/receives 256MB to each other node

Cluster	Site	Year	Interface	T	aggregated BW
PPTI	jussieu	20??	1Gbit ethernet	254s	250Mo/s
sagitaire	lyon	2006	1Gbit ethernet	?	
paravance	rennes	2015	10Gbit ethernet	14.9s	4Go/s
gros	nancy	2019	25Gbit ethernet	6.8s	9.4Go/s
grcinq	nancy	2013	56Gbit InfiniBand	4.8s	13Go/s
grvingt	nancy	2018	100Gbit OmniPath	2.2s	29Go/s

Network Torture Test

With *real* HPC hardware

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

- *n* nodes, 6.75GB per node, $6.75n$GB in total

Network Torture Test

With *real* HPC hardware

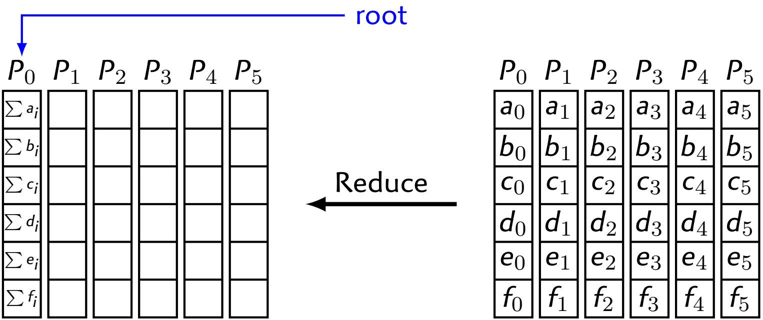
```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm);
```

- *n* nodes, 6.75GB per node, $6.75n$GB in total

# nodes	Data	T (s)	Aggregated BW
2	13.5GB	2.7	6.8GB/s
4	27GB	2.9	9.3GB/s
⋮	⋮	⋮	⋮
64	430GB	4.1	105GB/s
128	861GB	8.9	96GB/s
256	1.7TB	13.0	130GB/s
512	3.4TB	13.7	248GB/s
1024	6.9TB	15.9	434GB/s
2048	13.4TB	17.0	788GB/s
4096	27.5TB	18.7	1470GB/s

Reduce

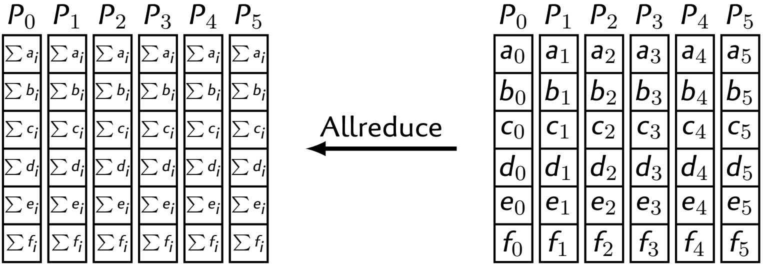
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm);
```



- `sendbuf == MPI_IN_PLACE` works at `root`
- Predefined operations: $+$, $\times$, $\min$, $\max$, AND, OR, XOR, ...
- User-definable operations

All-Reduce

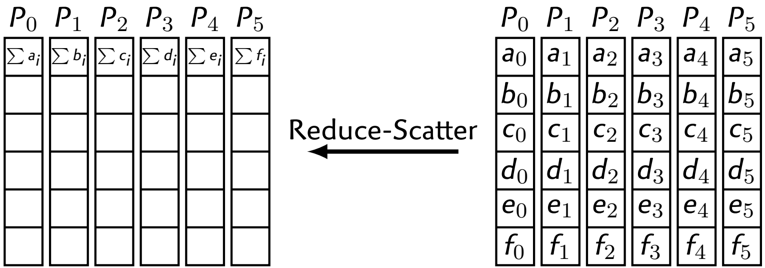
```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```



- `sendbuf == MPI_IN_PLACE` works (all processes)
- Predefined operations: $+$, $\times$, $\min$, $\max$, AND, OR, XOR, ...
- User-definable operations

Reduce-Scatter

```
int MPI_Reduce_scatter_block(void* sendbuf, void* recvbuf,
                             int recvcount, MPI_Datatype datatype,
                             MPI_Op op, MPI_Comm comm)
```



- `sendbuf == MPI_IN_PLACE` works (all processes)
- Predefined operations: $+$, $\times$, $\min$, $\max$, AND, OR, XOR, ...
- User-definable operations
- See also : `MPI_Reduce_scatter`

Scan

A.k.a. Prefix-sum

```c
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

a_0	b_0	c_0
a_0+a_1	b_0+b_1	c_0+c_1
$a_0+a_1+a_2$	$b_0+b_1+b_2$	$c_0+c_1+c_2$
$a_0+\cdots+a_3$	$b_0+\cdots+b_3$	$c_0+\cdots+c_3$
$a_0+\cdots+a_4$	$b_0+\cdots+b_4$	$c_0+\cdots+c_4$
$a_0+\cdots+a_5$	$b_0+\cdots+b_5$	$c_0+\cdots+c_5$

$\xleftarrow{\text{Scan}}$

a_0	b_0	c_0
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_4	c_4
a_5	b_5	c_5

- `sendbuf == MPI_IN_PLACE` works (all processes)
- Predefined operations: $+$, $\times$, $\min$, $\max$, AND, OR, XOR, ...
- User-definable operations
- See also : `MPI_Exscan`

Guess What?

Non-blocking collective operations

- Only since MPI v3.0 (2012)
- `MPI_Ibcast`, `MPI_Iscatter`, `MPI_Igather`, `MPI_Iallgather`, `MPI_Ialltoall`, `MPI_Ireduce`, `MPI_Iallreduce`, `MPI_Ireduce_scatter`, etc.

Collective Operations on a Subgroup

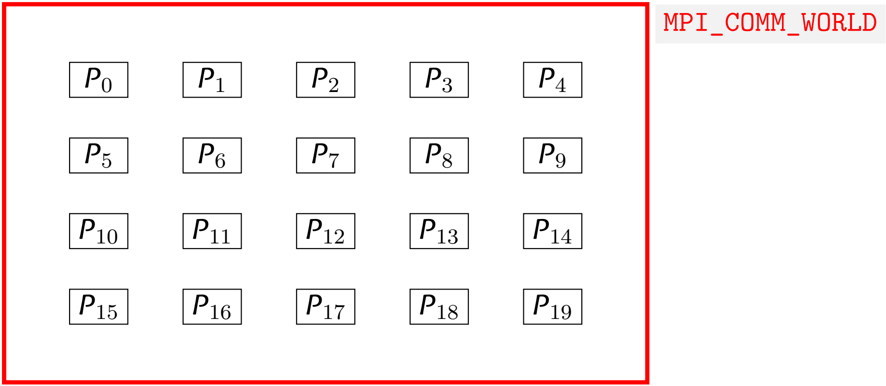
- Collective operations = **all** processes in a communicator
- Initially, a single communicator (`MPI_COMM_WORLD`)
- Possible to create **sub**-communicators

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

Description

- Collective operation
- `newcomm` contains all processes in `comm` with the same `color`, ordered by `key` (ties broken with rank in `comm`)
  - This **partitions** `comm`
  - Distinct sub-communicator for each value of `key`

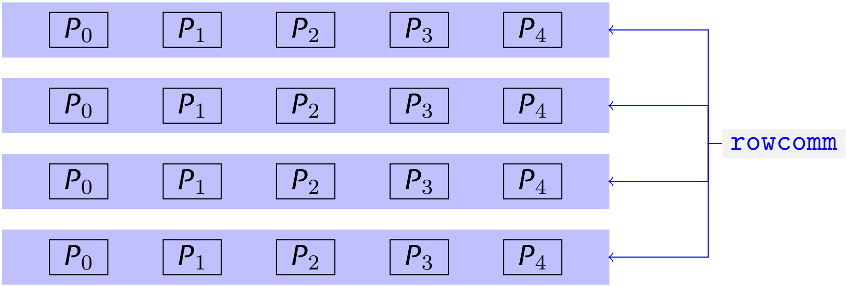
Typical Use-Case: 2D Process Grid




```
MPI_Comm rowcomm, colcomm;
int i = rank / n, j = rank % n;
MPI_Comm_split(MPI_COMM_WORLD, i, j, &rowcomm);
MPI_Comm_split(MPI_COMM_WORLD, j, i, &colcomm);
```

- See also `MPI_Cart_create` and `MPI_Cart_Sub`

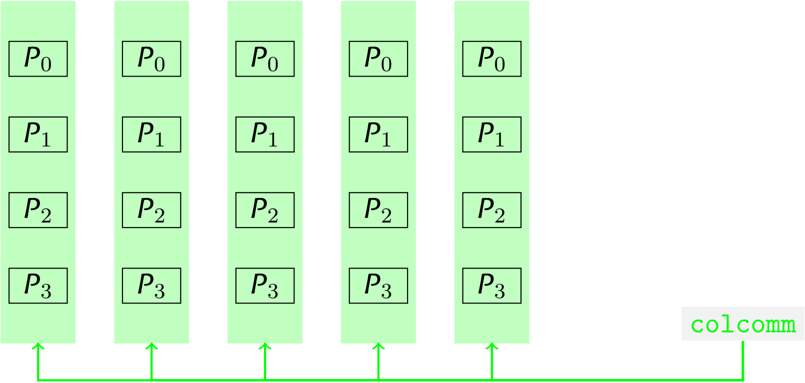
Typical Use-Case: 2D Process Grid




```
MPI_Comm rowcomm, colcomm;
int i = rank / n, j = rank % n;
MPI_Comm_split(MPI_COMM_WORLD, i, j, &rowcomm);
MPI_Comm_split(MPI_COMM_WORLD, j, i, &colcomm);
```

- See also `MPI_Cart_create` and `MPI_Cart_Sub`

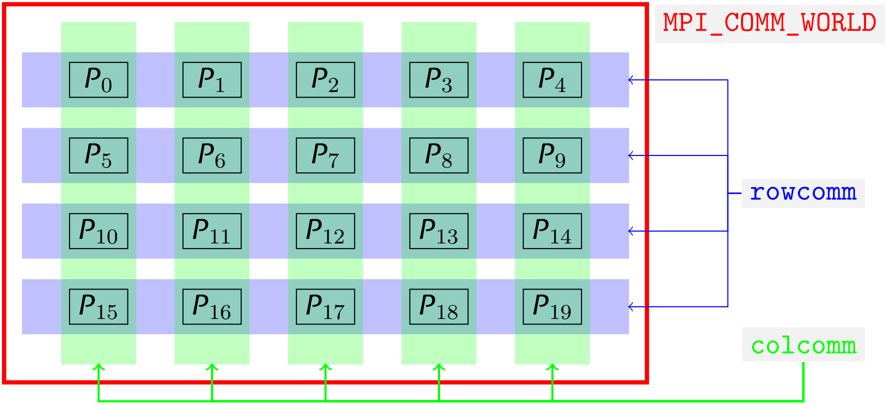
Typical Use-Case: 2D Process Grid




```
MPI_Comm rowcomm, colcomm;
int i = rank / n, j = rank % n;
MPI_Comm_split(MPI_COMM_WORLD, i, j, &rowcomm);
MPI_Comm_split(MPI_COMM_WORLD, j, i, &colcomm);
```

- See also `MPI_Cart_create` and `MPI_Cart_Sub`

Typical Use-Case: 2D Process Grid




```
MPI_Comm rowcomm, colcomm;
int i = rank / n, j = rank % n;
MPI_Comm_split(MPI_COMM_WORLD, i, j, &rowcomm);
MPI_Comm_split(MPI_COMM_WORLD, j, i, &colcomm);
```

- See also `MPI_Cart_create` and `MPI_Cart_Sub`