MPI Basics

S. Van Criekingen
UPJV / MeCS

www.mecs.u-picardie.fr

January 28, 2015

MPI: references

Official website and specifications :

http://www.mpi-forum.org/

http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

Tutorials:

https://computing.llnl.gov/tutorials/mpi/

http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html

http://www.crihan.fr/calcul/tech/doc_ibm_pwr5/EchMsg
http://www.crihan.fr/calcul/tech/documentation-ibm-cluster-idataplex-antares/formations/OpenMP_MPI.zip/view

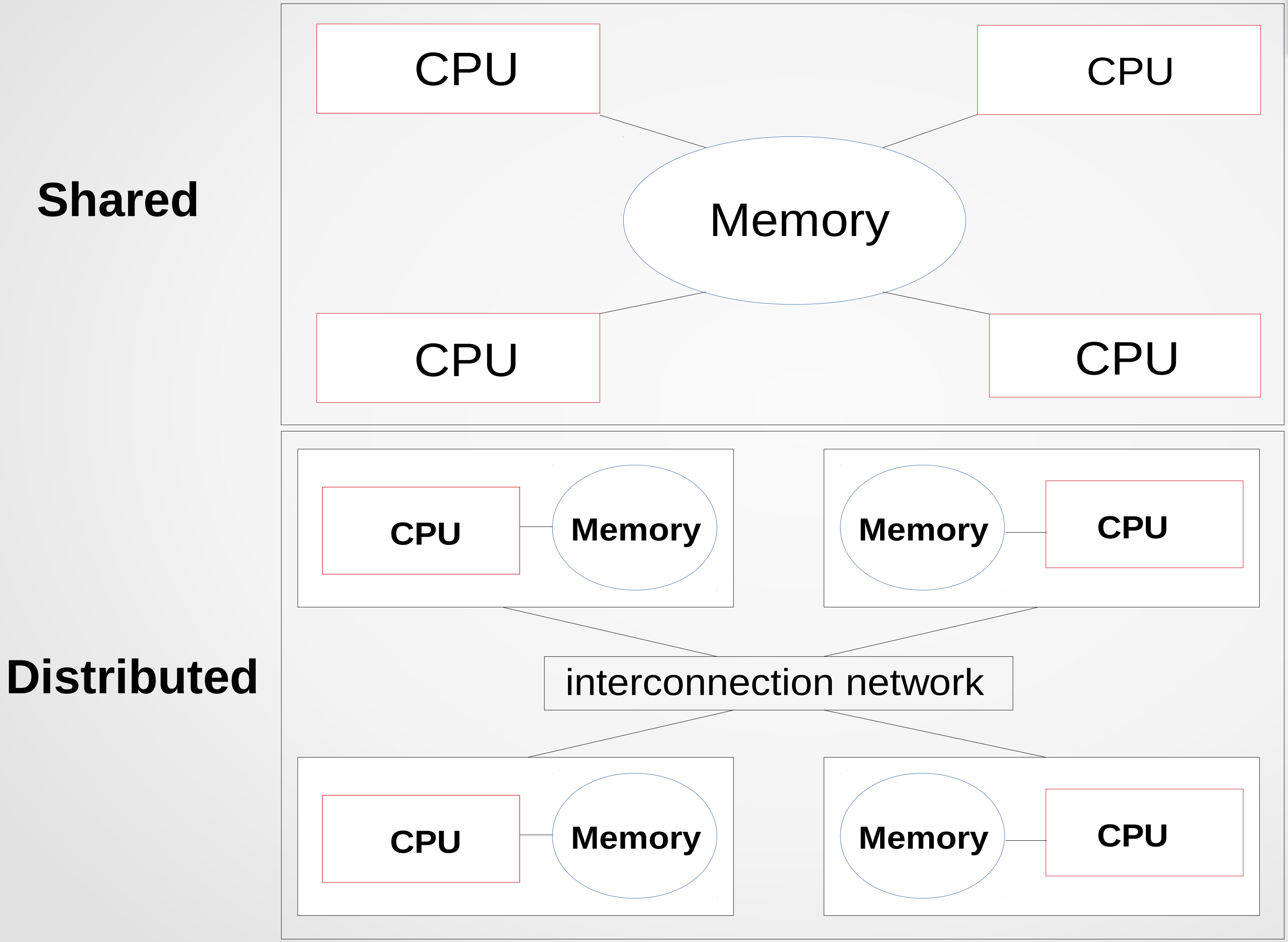
Layout

- Introduction & “Hello World”
- Point-to-point Communications
- Collective Communications
- Derived Data Types
- Communicators and Topologies
- Exercises

Layout

- Introduction & “Hello World”
- Point-to-point Communications
- Collective Communications
- Derived Data Types
- Communicators and Topologies
- Exercises

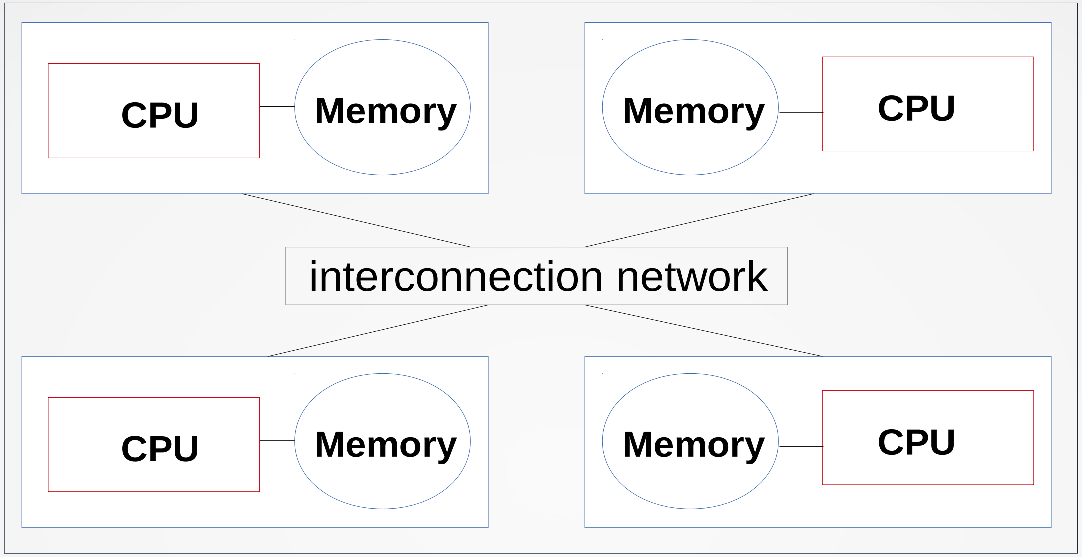
Shared vs. Distributed Memory



Programming models

- For shared memory: multi-threading (e.g. OpenMP)
- For distributed memory: message passing (e.g. MPI)

Message Passing (distributed memory)



Several processes act each on their own data and memory (own part of **distributed memory**).

Inter-process messages necessary for data exchange and synchronization.

The MPI standard

- MPI = Message Passing Interface
- First specification of standard: 1994
  Current version : MPI-3.0 (see MPI Forum)
- Various implementations:
  MPICH, MVAPICH, OpenMPI,…
- Note: MPI also works on shared-memory systems (but OpenMP might be easier to use on such systems).

MPI: basic principle

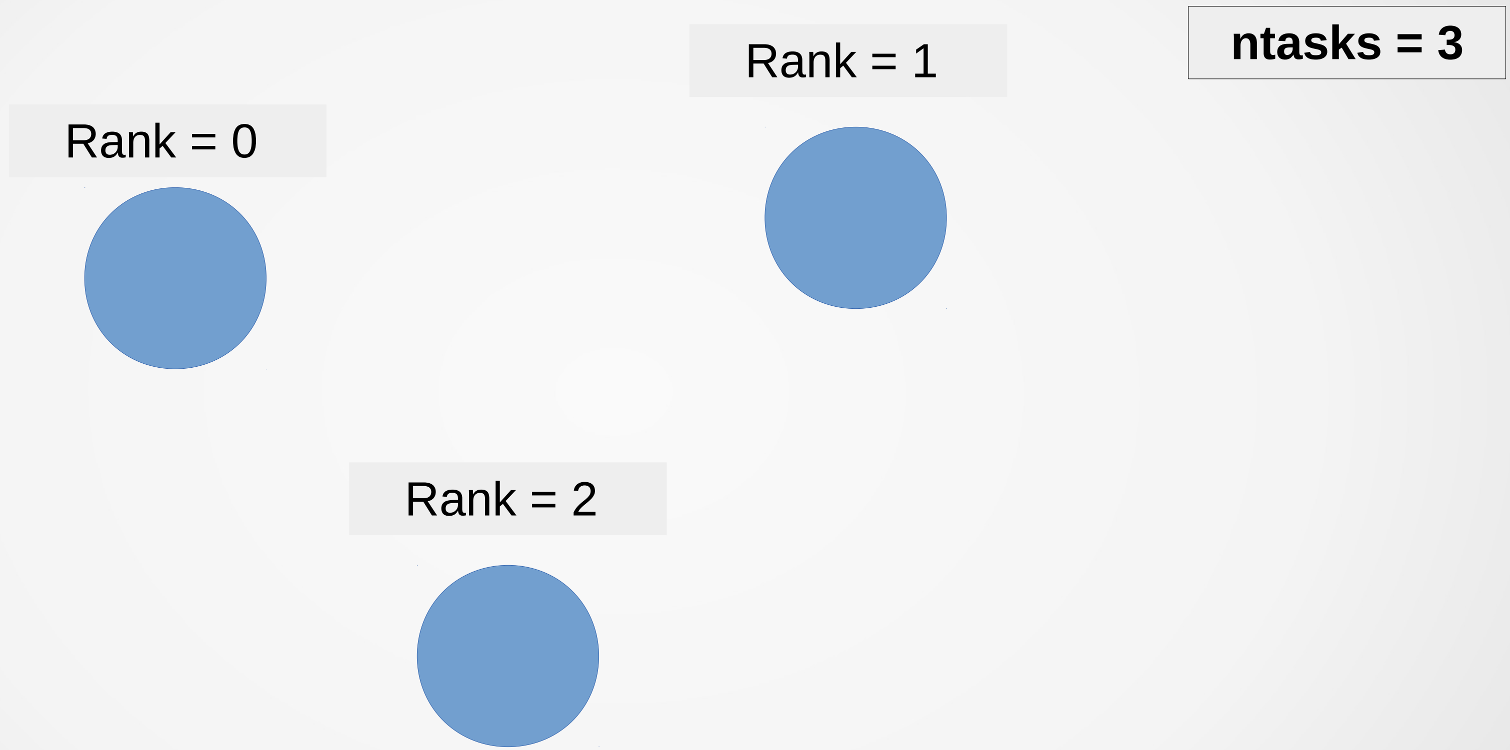
- A (single) MPI program runs on different processor cores, yielding different "MPI processes" or "MPI tasks".

- Each process/task is identified by a **rank :**

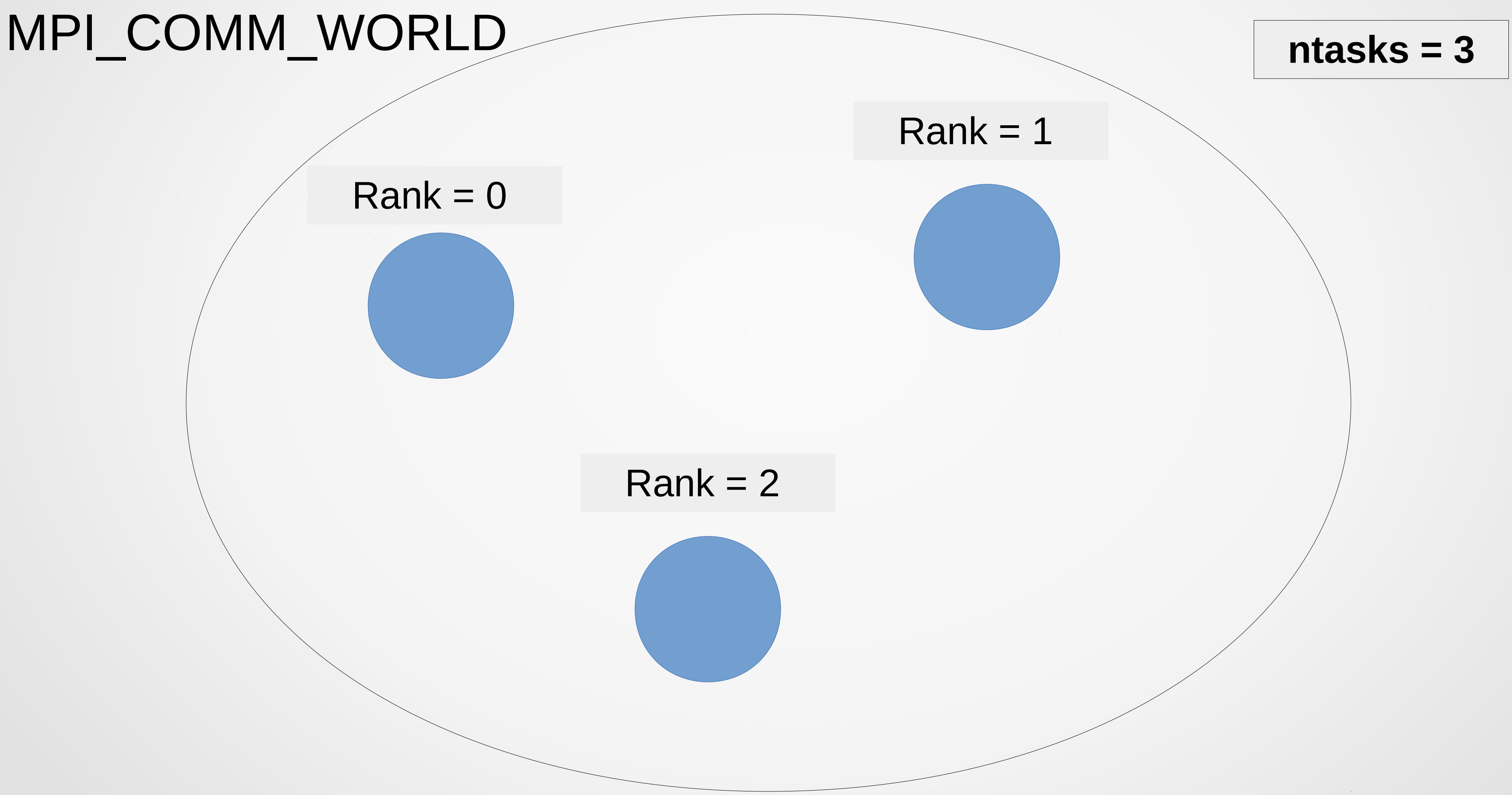
  rank = 0,1, … ntasks – 1

  where typically the number of tasks ntasks = number of available processor cores.

MPI: basic principle



MPI: basic principle



"Hello world" example (in C)

helloWorld.c

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[ ]) {
   int  ntasks, rank;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   printf ("Hello from rank %d out of %d tasks \n", rank, ntasks);
   MPI_Finalize();
}
```

MPI statements between MPI_init and MPI_Finalize

MPI_COMM_WORLD = *predefined communicator including all processes*

```
> mpicc helloWorld.c
> mpirun -np 3 ./a.out
Hello from rank 2 out of 3 tasks
Hello from rank 0 out of 3 tasks
Hello from rank 1 out of 3 tasks
```

← *set number of tasks to 3*

Order not deterministic

MPI: compile and run

Procedure to compile and run a MPI program:

1. To compile: here we use

   - **mpicc program.c**
   - **mpif90 program.f90**

   → ***a.out*** executable

2. To run on N processors:

   - **mpirun -np N ./*a.out*** (or **mpiexec**)

"Hello world" example (in Fortran 90)

helloWorld.f90

```
program main
  use mpi
  implicit none
  Integer :: ntasks, rank, ierr
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  write(*,'(a,i2,a,I2,a)') 'Hello from rank ', rank, ' out of ',ntasks,' tasks.'
  call MPI_FINALIZE(ierr)
end
```

```
> mpif90 helloWorld.f90
> mpirun -np 3 ./a.out
Hello from rank 2 out of 3 tasks.
Hello from rank 0 out of 3 tasks.
Hello from rank 1 out of 3 tasks.
```

←*set number of tasks to 3*

Order not deterministic

Main Environment Management Routines

C	Fortran	
MPI_Init (&argc,&argv)	**MPI_INIT** (ierr)	
MPI_Comm_size (comm,&size)	**MPI_COMM_SIZE** (comm,size,ierr)	Returns size = number of tasks
MPI_Comm_rank (comm,&rank)	**MPI_COMM_RANK** (comm,rank,ierr)	Returns rank in [0, size-1]
MPI_Finalize ()	**MPI_FINALIZE** (ierr)	

where *comm* = communicator = MPI_COMM_WORLD (typically)

Main Environment Management Routines

Also useful:

C	Fortran	
MPI_Wtime ()	**MPI_WTIME** ()	Timing routine
MPI_Abort (comm,errorcode)	**MPI_ABORT** (comm,errorcode,ierr)	Terminates (all) processes

where *comm* = communicator = MPI_COMM_WORLD (typically)

Layout

- Introduction & “Hello World”
- Point-to-point Communications
- Collective Communications
- Derived Data Types
- Communicators and Topologies
- Exercises

Send / Receive

Messages can be sent from one MPI process (rank) to another, which should be ready to receive it

→ an operation between two processes is ***cooperative***.

Rank n_1 : **MPI_Send** (&send_msg, …, dest = n_2, ...)

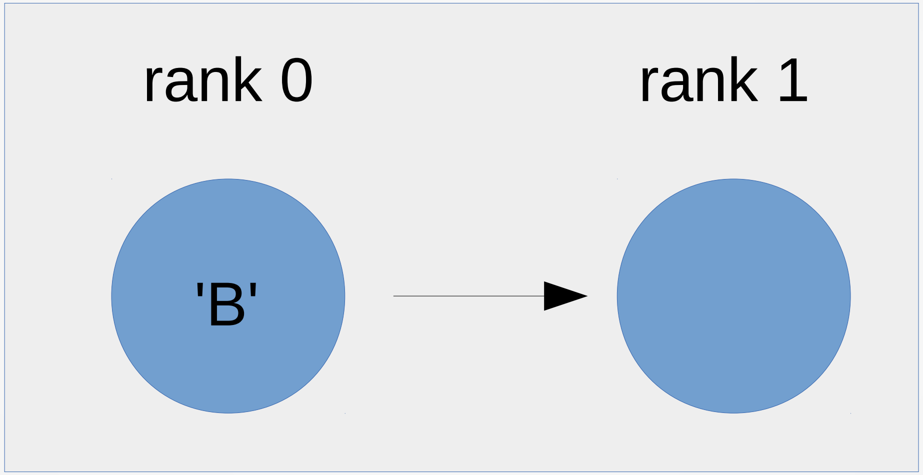
↓

Rank n_2 : **MPI_Recv** (&recv_msg, …, source = n_1, ...)

C code

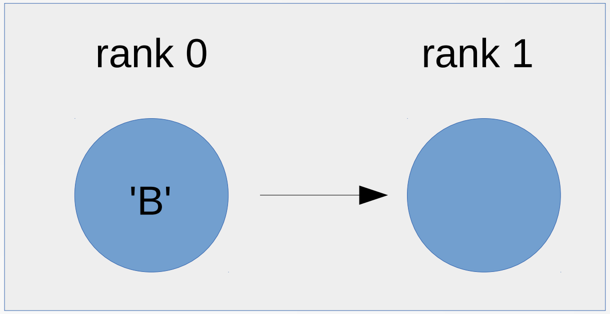
N.B. : Since MPI 2.0, there exist *one-sided* communications.

Send/Receive example



Send/Receive example (in C, zoom)

```
[…]
 count = 1, tag = 1
 if (rank == 0) {
   dest = 1;
   sent_msg = 'B';
   MPI_Send (&sent_msg, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   printf("Char sent by rank 0: %c \n", sent_msg);
 }
 else if (rank == 1) {
   source = 0;
   MPI_Recv (&recv_msg, count, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat );
   printf("Char received by rank 1: %c \n", recv_msg);
 }
[…]
```



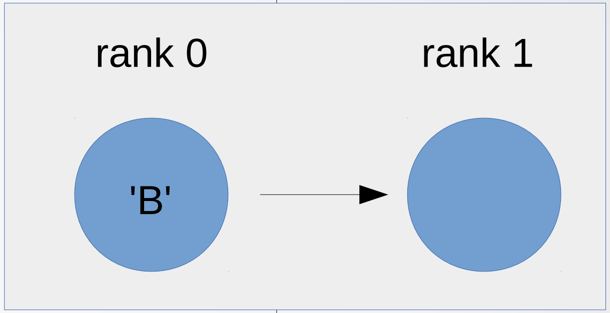
sendRecv.c

```
> mpirun -np 2 ./a.out
Char sent by rank 0: B
Char received by rank 1: B
```

Send/Receive example (in C)

sendRecv.c

```
Int main(int argc, char *argv[])  {
 int ntasks, rank, dest, source, count, tag;
 char recv_msg, sent_msg;
 MPI_Status Stat;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 count = 1, tag = 1 ;
 if (rank == 0) {
  dest = 1;
  sent_msg = 'B';
  MPI_Send (&sent_msg, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Char sent by rank 0: %c \n", sent_msg);
 }
else if (rank == 1) {
  source = 0;
  MPI_Recv (&recv_msg, count, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat );
  printf("Char received by rank 1: %c \n", recv_msg);
 }
 MPI_Finalize();
}
```




```
> mpirun -np 2 ./a.out
Char sent by rank 0: B
Char received by rank 1: B
```

Send/Receive (in C)

MPI_Send (&send_msg, count, datatype, dest, tag, comm)

MPI_Recv (&recv_msg, count, datatype, source, tag, comm, &status)

- **send_msg** / **recv_msg** = message sent / received (passed by reference)
- **count** = number of data elements sent/received
- **datatype** = type of the sent/received data
  In C, this is typically: **MPI_CHAR**, **MPI_INT**, **MPI_FLOAT**...
- **dest/source** = rank of the destination/source

Send/Receive (in C)

MPI_Send (&send_msg, count, datatype, dest, tag, comm)

MPI_Recv (&recv_msg, count, datatype, source, tag, comm, &status)

- **tag** = non-negative integer to identify the message.
  Tag in corresponding **MPI_Send** and **MPI_Recv** *must* match.
- **comm** = communicator (typically MPI_COMM_WORLD)
- **status** = information on received message

N.B. : *source + destination + tag + communicator = "**message enveloppe**"*

Send/Receive : C vs. Fortran

C code :

```
MPI_Send (&send_msg, count, datatype, dest, tag, comm)

MPI_Recv (&recv_msg, count, datatype, source, tag, comm, &status)
```

Fortran code :

```
MPI_SEND (send_msg, count, datatype, dest, tag, comm, ierr)

MPI_RECV (recv_msg, count, datatype, source, tag, comm, status, ierr)
```

C data types	Fortran data types
MPI_CHAR	MPI_CHARACTER
MPI_INT	MPI_INTEGER
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
...	...

Send/Receive example (in Fortran 90)

sendRecv.f90

```
[...]
integer :: ntasks, rank, ierr, dest, source, count = 1, tag = 1
character :: recv_msg, sent_msg
Integer :: Stat(MPI_Status_size)
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
if (rank .eq. 0) then
  dest = 1
  sent_msg = 'B'
  call MPI_Send(sent_msg, 1, MPI_CHARACTER, dest, tag, MPI_COMM_WORLD, ierr)
  Write(*,*) 'Char sent by rank 0: ', sent_msg
else if (rank .eq. 1) then
  source = 0
  call MPI_Recv(recv_msg, 1, MPI_CHARACTER, source, tag, MPI_COMM_WORLD, Stat, ierr)
  Write(*,*) 'Char received by rank 1: ', recv_msg
endif
call MPI_FINALIZE(ierr)
end
```

```
> mpirun -np 2 ./a.out
Char sent by rank 0: B
Char received by rank 1: B
```

Blocking vs. Non-blocking

MPI_Send / **MPI_Recv** are **blocking** operations: they return only when

- the sent data can be modified
- the received data is ready for use

On the opposite, equivalent **non-blocking** operations **MPI_Isend** / **MPI_Irecv** simply **request** the MPI library to perform the operation when it is able.

The user can not predict when that will happen, but can make sure it happened with a "wait" statement.

Blocking vs. Non-blocking Send/Receive

Blocking

```
[...]
 MPI_Send (...);
[...]
 MPI_Recv (...);
[...]
```

≡

Non- Blocking

```
[...]
   MPI_Isend (...);
   MPI_Wait(...);
[...]
   MPI_Irecv (...);
   MPI_Wait(...);
[...]
```

→

Non- Blocking

```
[...]
   MPI_Isend (...);
   [...]
   MPI_Wait(...);
[...]
   MPI_Irecv (...);
   [...]
   MPI_Wait(...);
[...]
```

→ Possibility to overlap communications and computations.

Blocking vs. Non-blocking Send/Receive

C code

<u>Blocking</u> :

```
MPI_Send (&send_msg, count, datatype, dest, tag, comm)

MPI_Recv (&recv_msg, count, datatype, source, tag, comm, &status)
```

<u>Non-Blocking</u> :

```
MPI_Isend (&send_msg, count, datatype, dest, tag, comm, &request)

MPI_Irecv (&recv_msg, count, datatype, source, tag, comm, &request)
```

where **request** is an output argument used to determine completion of the non-blocking operation using **MPI_Wait** :

```
MPI_Wait (&request,&status)
```

input output

Layout

- Introduction & “Hello World”
- Point-to-point Communications
- Collective Communications
- Derived Data Types
- Communicators and Topologies
- Exercises

Collective Communications

Opposite to point-to-point communications, collective communications **involve all processes in the communicator** (typically MPI_COMM_WORLD).

Types of collective communications:

- Synchronization: barrier
- Data movement: broadcast, scatter, gather
- Reductions

Synchronization

MPI_Barrier (MPI_COMM_WORLD)	C code

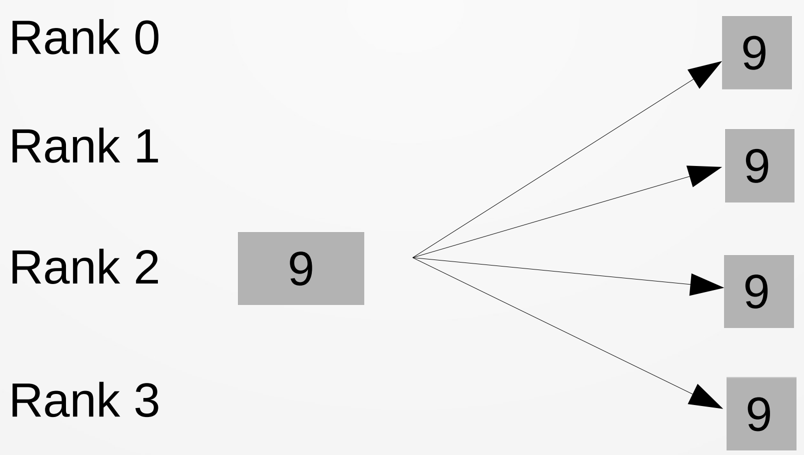
call MPI_BARRIER (MPI_COMM_WORLD, ierr)	Fortran code

At the barrier, each task blocks until all the other tasks reach the same barrier.

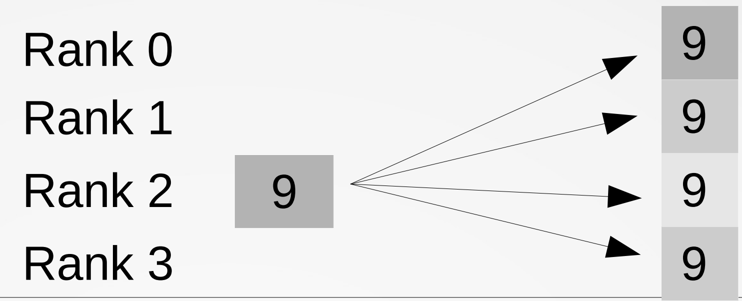
Then all tasks are free to proceed.

Broadcast

MPI_Bcast



Broadcast



C code

```
[...]
 int msg[1];
 source = 2;
 count = 1;
 if (rank == source){
   msg[0]= 9;
 }
 MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
 printf("On rank %d received: %d \n",rank,msg[0]);
 MPI_Finalize();
```

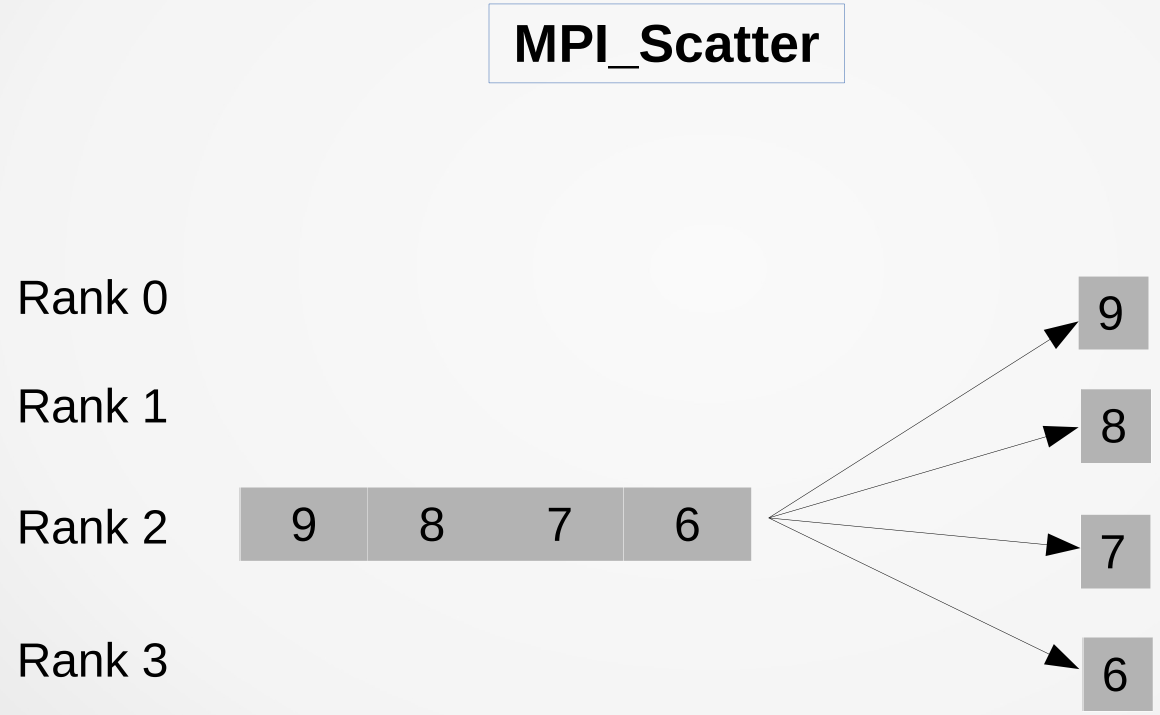
```
>  mpirun -np 4 ./a.out
On rank 0 received: 9
On rank 1 received: 9
On rank 2 received: 9
On rank 3 received: 9
```

Broadcast

MPI_Bcast (&msg**,** count, datatype, source, comm) C code

- **msg** = message broadcasted
- **count** = number of data elements broadcasted
- **datatype** = type of the broadcasted data
- **source** = rank of the source
- **comm** = communicator (typically MPI_COMM_WORLD)

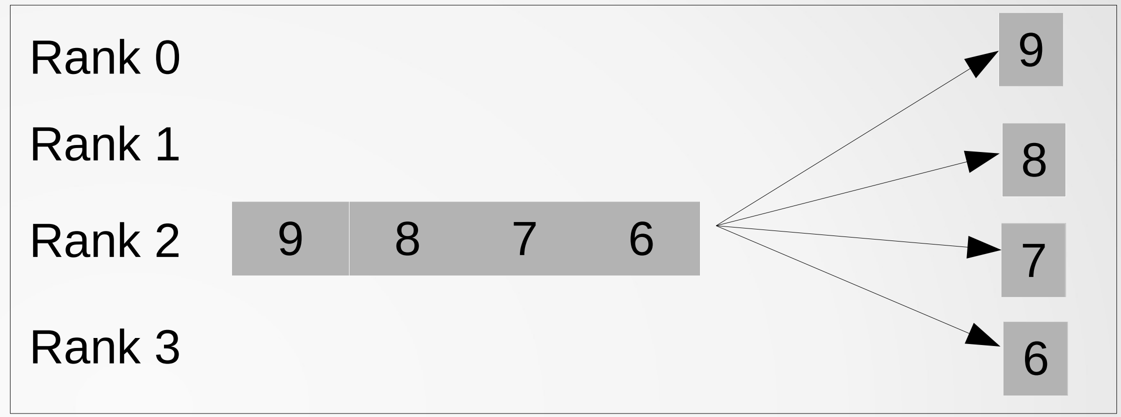
Scatter



Scatter

C code

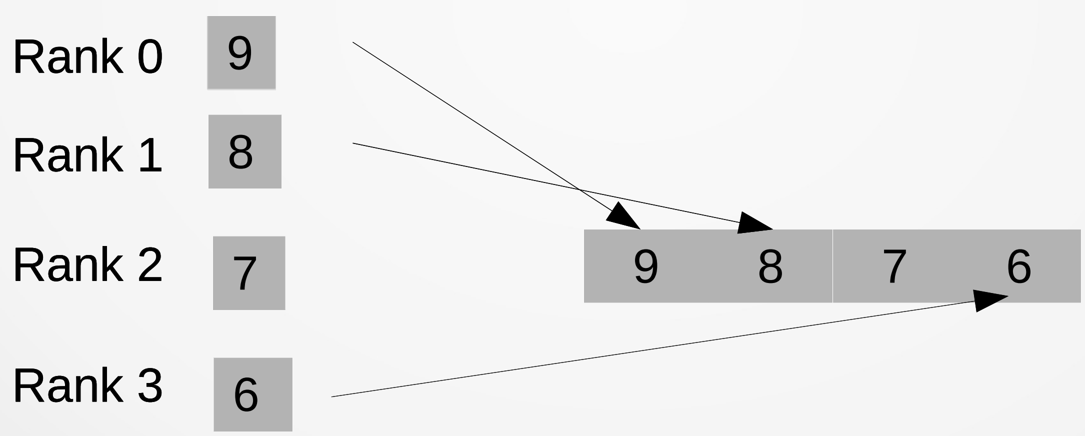
```
[...]
 int send_msg[4];
 int recv_msg[1];
 source = 2;
 count = 1;
 if (rank==source)
 {
   send_msg[0] = 9;
   send_msg[1] = 8;
   send_msg[2] = 7;
   send_msg[3] = 6;
 }
MPI_Scatter(&send_msg, count, MPI_INT, &recv_msg, count, MPI_INT, source, MPI_COMM_WORLD);
printf("On rank= %d received: %d \n",rank,recv_msg[0]);
MPI_Finalize();
```




```
> mpirun -np 4 ./a.out
On rank= 0 received: 9
On rank= 1 received: 8
On rank= 2 received: 7
On rank= 3 received: 6
```

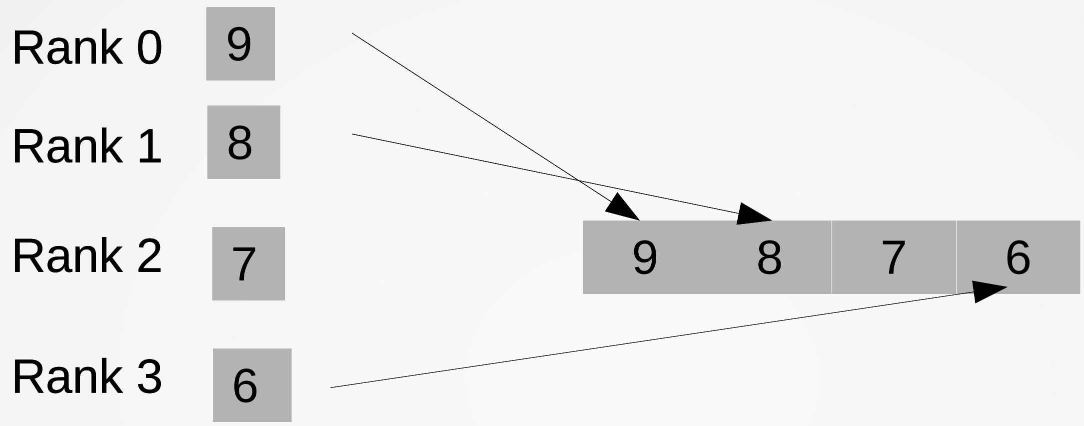
Gather

MPI_Gather



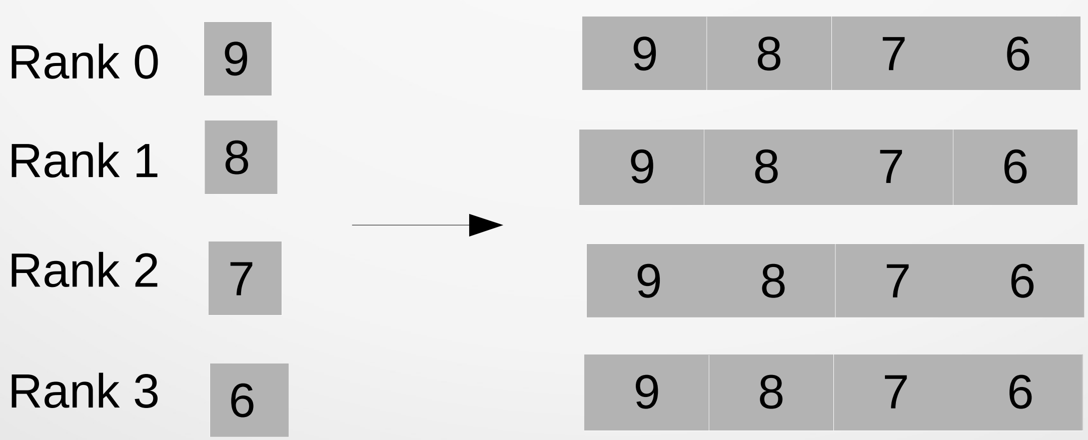
Note: Gather / AllGather

MPI_Gather



MPI_AllGather

"MPI_AllGather = MPI_Gather + MPI_Bcast"



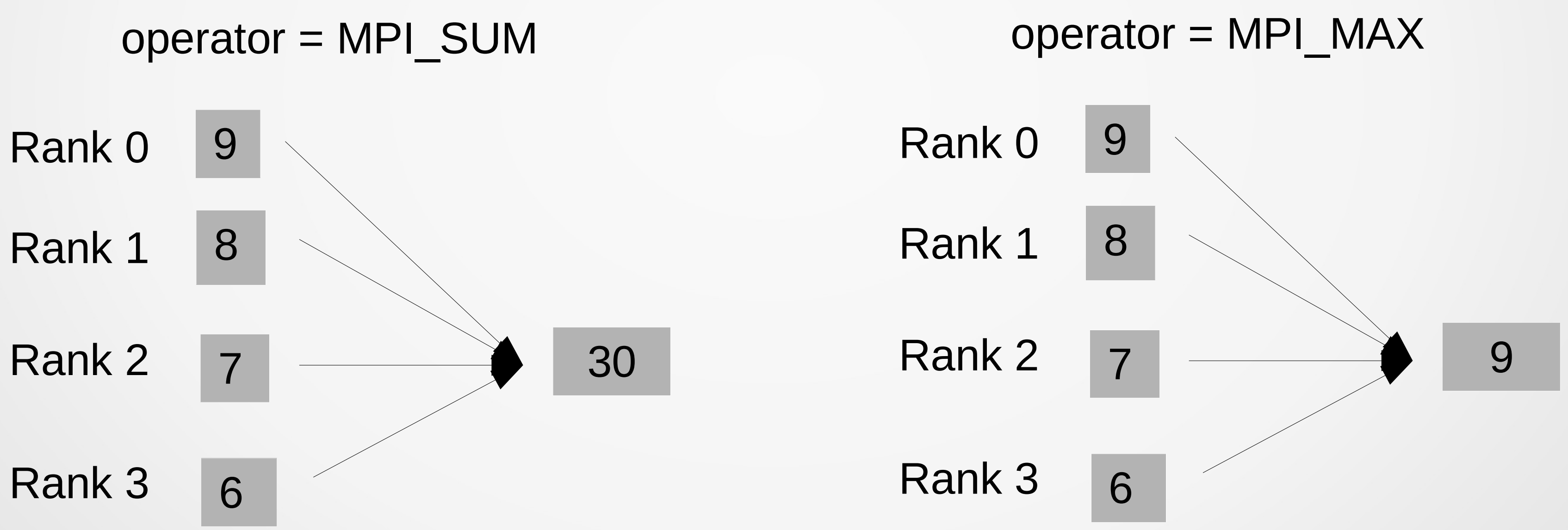
Note: All to All

MPI_Alltoall

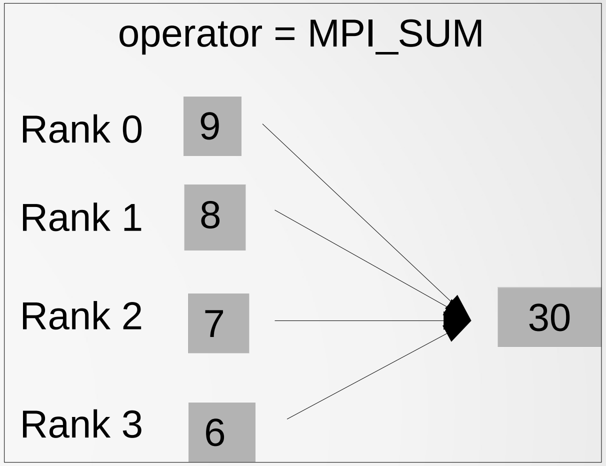
					→				
Rank 0	1	2	3	4		1	5	9	13
Rank 1	5	6	7	8		2	6	10	14
Rank 2	9	10	11	12		3	7	11	15
Rank 3	13	14	15	16		4	8	12	16

Reduce

MPI_Reduce (…, operator, ...)



Reduce



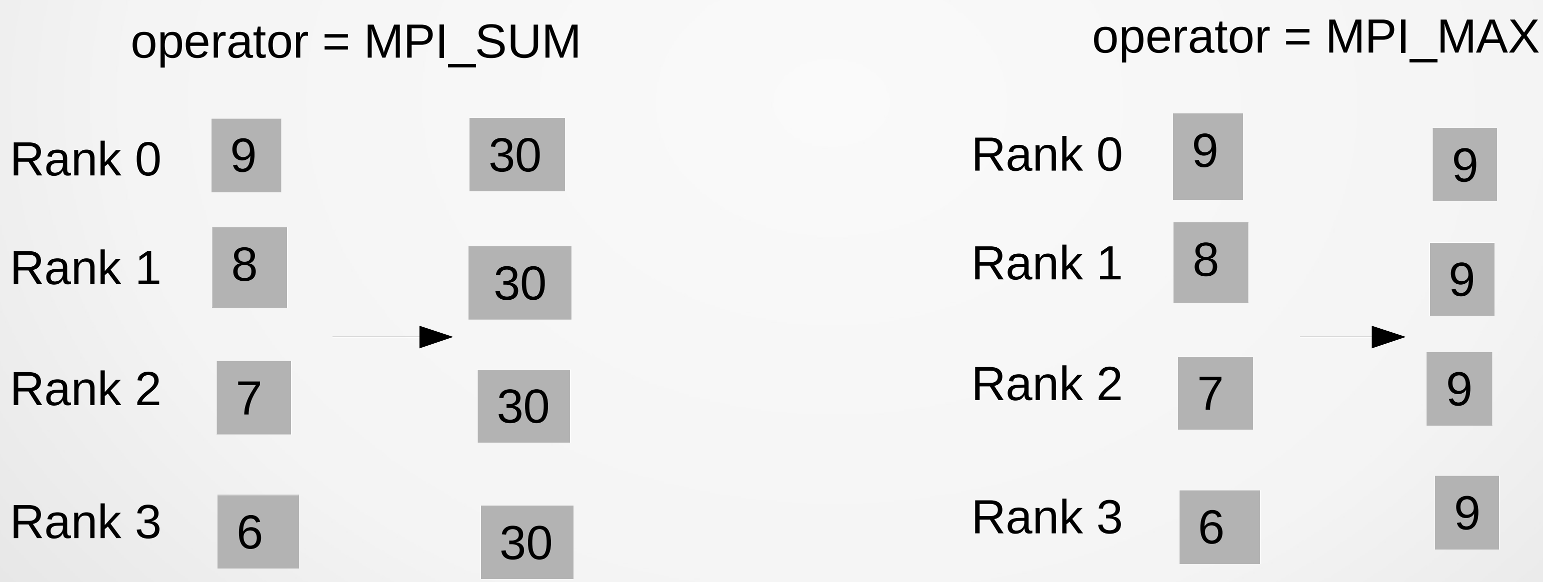
C code

```
[...]
 int send_msg[1]];
 int recv_msg[1];
 dest = 2;
 if (rank==0) send_msg[0] = 9;
 if (rank==1) send_msg[0] = 8;
 if (rank==2) send_msg[0] = 7;
 if (rank==3) send_msg[0] = 6;
 MPI_Reduce(&send_msg, &recv_msg, 1, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD);
 if (rank == dest) printf("On rank %d received: %d \n",rank,recv_msg[0]);
 MPI_Finalize();
```

```
> mpirun -np 4 ./a.out
On rank 2 received: 30
```

All Reduce

MPI_Allreduce (…, operator, ...)

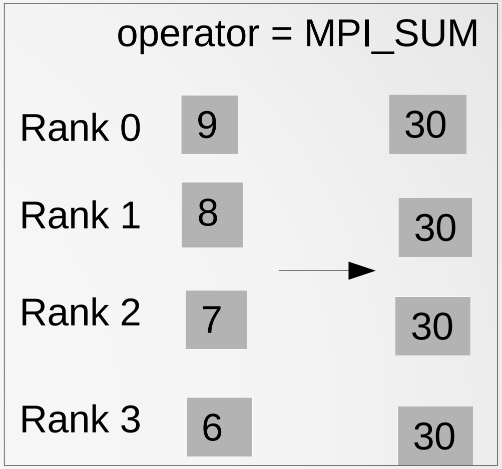


"MPI_Allreduce = MPI_Reduce + MPI_Bcast"

All Reduce

C code

```
[...]
int send_msg[1]];
int recv_msg[1];
MPI_Init(&argc,&argv);
[...]
if (rank==0) send_msg[0] = 9;
if (rank==1) send_msg[0] = 8;
if (rank==2) send_msg[0] = 7;
if (rank==3) send_msg[0] = 6;
MPI_Allreduce(&send_msg,&recv_msg,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
printf("On rank= %d received: %d \n",rank,recv_msg[0]);
MPI_Finalize();
```

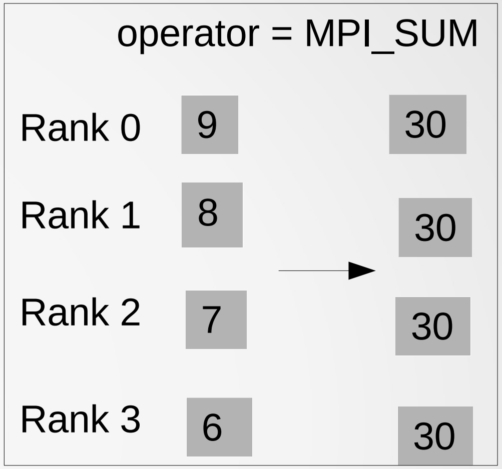



```
> mpirun -np 4 ./a.out
On rank= 0 received: 30
On rank= 1 received: 30
On rank= 2 received: 30
On rank= 3 received: 30
```

All Reduce

Fortran 90 code

```
[...]
integer, dimension(1) :: send_msg, recv_msg
call MPI_Init(ierr)
[...]
if (rank==0) send_msg(1) = 9
if (rank==1) send_msg(1) = 8
if (rank==2) send_msg(1) = 7
if (rank==3) send_msg(1) = 6
call MPI_Allreduce(send_msg,recv_msg,1,MPI_INTEGER,MPI_SUM, MPI_COMM_WORLD,ierr)
write(*,'(a,i2,a,i2)') 'On rank ', rank, ' received: ', recv_msg(1)
call MPI_Finalize();
```




```
> mpirun -np 4 ./a.out
On rank= 0 received: 30
On rank= 1 received: 30
On rank= 2 received: 30
On rank= 3 received: 30
```

Layout

- Introduction & “Hello World”
- Point-to-point Communications
- Collective Communications
- Derived Data Types
- Communicators and Topologies
- Exercises

Primitive vs. Derived Data types

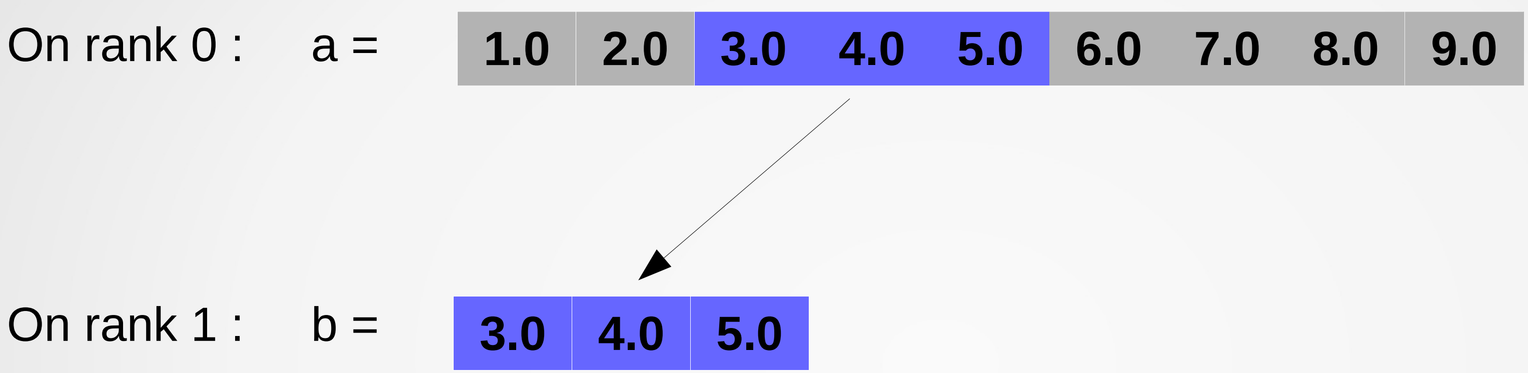
Recall the (primitive) data types usable in MPI communications:

C data types	Fortran data types
MPI_CHAR	**MPI_CHARACTER**
MPI_INT	**MPI_INTEGER**
MPI_FLOAT	**MPI_REAL**
MPI_DOUBLE	**MPI_DOUBLE_PRECISION**
...	...

More complex data types can be exchanged with MPI: these are the **Derived Data Types**.

These are typically used to exchange data extracted from existing vectors and matrices.

MPI_Type_contiguous



MPI_Type_contiguous (n_extracted, basictype, &newType) — C code

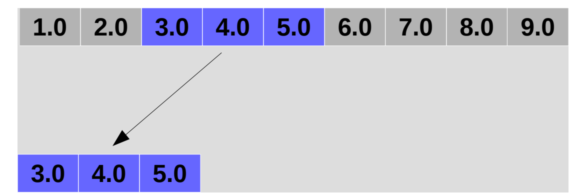
- **n_extracted = 3**
- **basicType = MPI_FLOAT**

→ "newType" generated

MPI_Type_contiguous

DdtContiguous1D.c

```c
[...]
 float a[9] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
 int n_extracted = 3;
 float b[n_extracted];
 MPI_Datatype newType;
 MPI_Type_contiguous (n_extracted, MPI_FLOAT, &newType);
 MPI_Type_commit(&newType);
 if (rank == 0) {
   dest = 1;
   MPI_Send(&a[2], 1, newType, dest, tag, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
   source = 0;
   MPI_Recv(b, n_extracted, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   for (i=0;i< n_extracted;++i) printf("On rank %d  b[%d]= %2.1f \n", rank, i, b[i]);
 }
 MPI_Type_free(&newType);
[...]
```

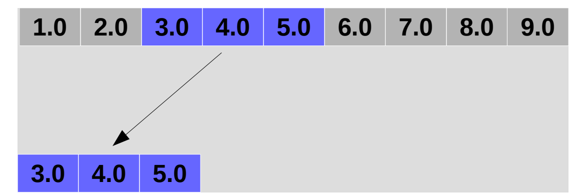



```
>  mpirun -np 2 ./a.out
On rank 1  b[0]= 3.0
On rank 1  b[1]= 4.0
On rank 1  b[2]= 5.0
```

MPI_Type_contiguous

DdtContiguous1D.c

```
[...]
 float a[9] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
 int n_extracted = 3;
 float b[n_extracted];
 MPI_Datatype newType;
 MPI_Type_contiguous (n_extracted, MPI_FLOAT, &newType);
 MPI_Type_commit(&newType);
 if (rank == 0) {
   dest = 1;
   MPI_Send(&a[5], 1, newType, dest, tag, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
   source = 0;
   MPI_Recv(b, n_extracted, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   for (i=0;i< n_extracted;++i) printf("On rank %d  b[%d]= %2.1f \n", rank, i, b[i]);
 }
 MPI_Type_free(&newType);
[...]
```




```
>  mpirun -np 2 ./a.out
On rank 1  b[0]= 6.0
On rank 1  b[1]= 7.0
On rank 1  b[2]= 8.0
```

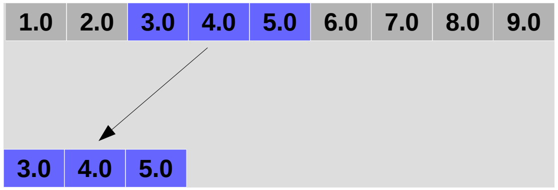
MPI_Type_contiguous

DdtContiguous1D.c

```
[...]
 float a[9] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
 int n_extracted = 3;
 float b[n_extracted];
 MPI_Datatype newType;
 MPI_Type_contiguous (n_extracted, MPI_FLOAT, &newType);
 MPI_Type_commit(&newType);
 if (rank == 0) {
   dest = 1;
   MPI_Send(&a[2], 1, newType, dest, tag, MPI_COMM_WORLD);
 }
 else if (rank == 1) {
   source = 0;
   MPI_Recv(b, n_extracted, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   for (i=0;i< n_extracted;++i) printf("On rank %d  b[%d]= %2.1f \n", rank, i, b[i]);
 }
 MPI_Type_free(&newType);
[...]
```

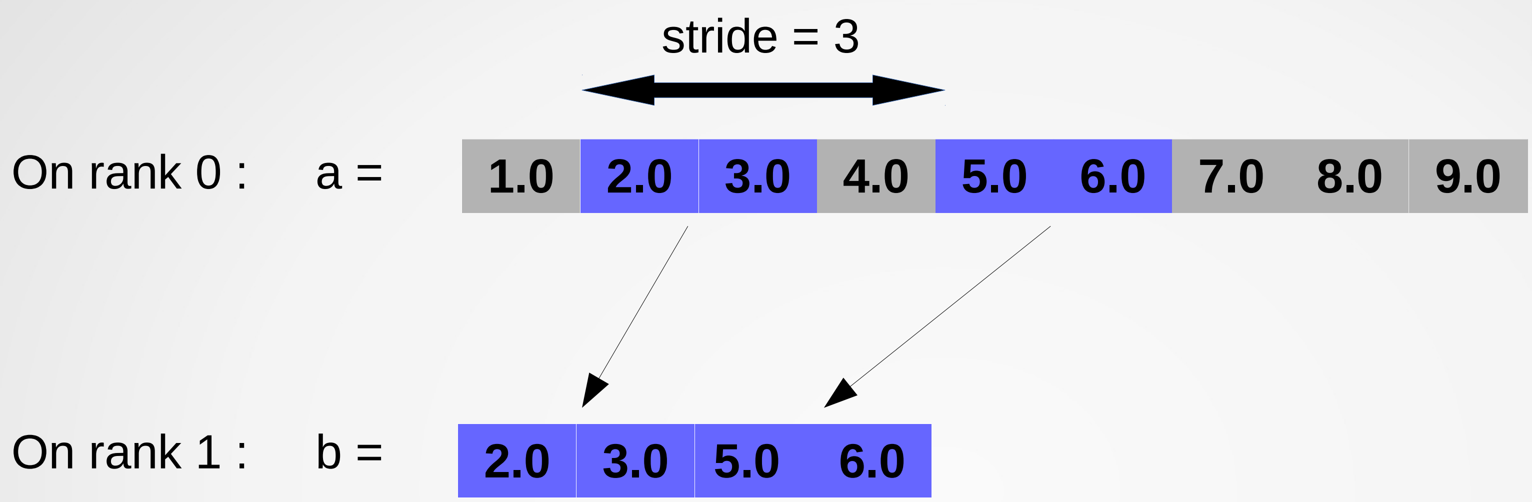
← commit required

← so that **newType** can be re-used




```
>  mpirun -np 2 ./a.out
On rank 1  b[0]= 3.0
On rank 1  b[1]= 4.0
On rank 1  b[2]= 5.0
```

MPI_Type_vector

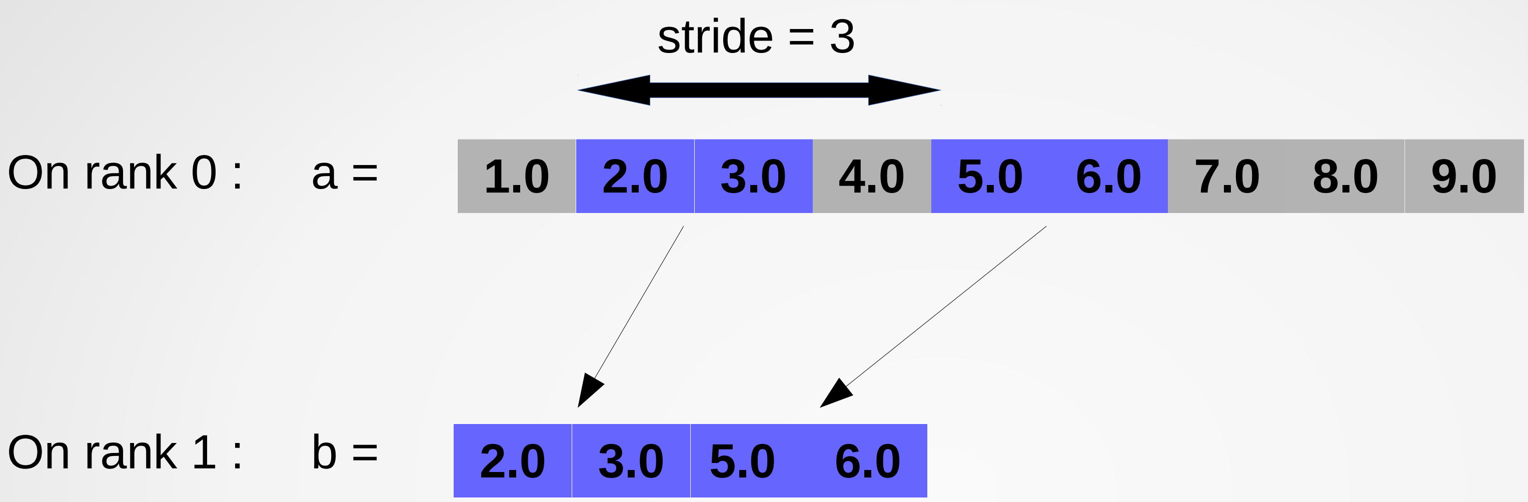


MPI_Type_vector (n_blocks, blocklength, stride, basictype, &newtype)	C code

- **n_blocks = 2**
- **blocklength = 2**
- **stride = 3**
- **basicType** = **MPI_FLOAT**

→ "newType" generated

MPI_Type_vector



1.0	2.0	3.0
4.0	5.0	6.0
7.0	8.0	9.0

"row major order"

Matrices in C and Fortran

C:

```
float a[3][3] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
```

C code

$$\longrightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

"row major order"

Fortran:

```
real, dimension(3,3) :: A
a = reshape( (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /), (/ 3, 3/) )
```

F90 code

$$\longrightarrow \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

"column major order"

Matrices in C and Fortran

C:

```
float a[3][3] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
```

C code

$$\rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

MPI_Type_contiguous
extracts **rows**

Fortran:

```
real, dimension(3,3) :: A
a = reshape( (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /), (/ 3, 3/) )
```

F90 code

$$\rightarrow \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

MPI_TYPE_CONTIGUOUS
extracts **columns**

Matrices in C and Fortran

C:

```
float a[3][3] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
```

C code

$$\rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

MPI_Type_vector
extracts **columns**

Fortran:

```
real, dimension(3,3) :: A
a = reshape( (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /), (/ 3, 3/) )
```

F90 code

$$\rightarrow \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

MPI_TYPE_VECTOR
extracts **rows**

Moreover...

C code

```
MPI_Type_contiguous (n_extracted, basicType, &newType)
```

```
MPI_Type_vector (n_blocks, blocklength, stride, basicType, &newtype)
```

```
MPI_Type_indexed (n_blocks, blocklengths[ ], offsets[ ], basicType, &newtype)
```

```
MPI_Type_struct (n_blocks, blocklengths[ ], offsets[ ], basicTypes[ ], &newtype)
```

Layout

- Introduction & “Hello World”
- Point-to-point Communications
- Collective Communications
- Derived Data Types
- Communicators and Topologies
- Exercises

Beyond MPI_COMM_WORLD...

There exists MPI statements to create and handle **communicators different from MPI_COMM_WORLD.**

→ enable for instance collective communications operations across a **subset** of tasks.

Ex : **MPI_COMM_SPLIT**(**comm**, **color**, key, **new_comm**)

partitions the communicator **comm** according to the value of **color**.

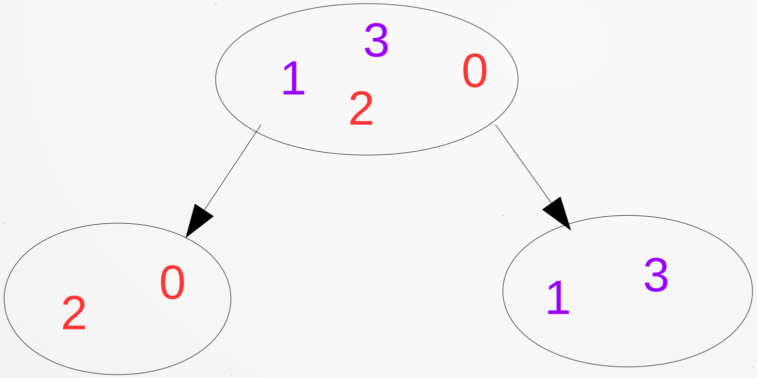
→ new communicators **new_comm** are created, each containing all the processes of the same color.

MPI_Comm_split

MPI_COMM_SPLIT(comm, color, key, new_comm)

Example with 2 colors:

- comm = MPI_COMM_WORLD
- color = 0 on even ranks and color = 1 on odd ranks

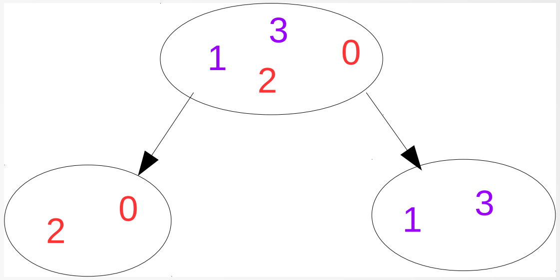


NB : The same name **new_comm** then refers to different communicators on processes of different colors.

Even/Odd Rank Split

```
[…]
 int orig_rank, new_rank, color, key=0;
 MPI_Comm  new_comm;
 MPI_Init (&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &orig_rank);
 color = orig_rank%2;
 MPI_Comm_split (MPI_COMM_WORLD, color, key, &new_comm);
 MPI_Comm_rank (new_comm, &new_rank);
 printf("orig_rank = %d  color = %d  new_rank = %d \n", orig_rank, color, new_rank);
 MPI_Comm_free (&new_comm);
 MPI_Finalize();
[…]
```

evenOddSplit.c




```
> mpirun -np 4 ./a.out
orig_rank = 0  color = 0  new_rank = 0
orig_rank = 1  color = 1  new_rank = 0
orig_rank = 2  color = 0  new_rank = 1
orig_rank = 3  color = 1  new_rank = 1
```

Virtual topologies

Communicators can be defined to represent a topology, i.e., a mapping of MPI processes (i.e., MPI ranks) into a geometric pattern.

Here we consider cartesian topologies. Example:

2 (0,2)	5 (1,2)	8 (2,2)
1 (0,1)	4 (1 ,1)	7 (2,1)
0 (0,0)	3 (1,0)	6 (2,0)

Interest : provide tools to easily determine neighbors, correspondance between the rank and the coordinates in the grid.

Note : May not necessarliy correspond to physical CPU layout
→ « virtual ».

Cartesian topologies

MPI_Cart_create(MPI_COMM_WORLD, **ndims**, **dims**, …, **&cart_comm**)

→ creates a new communicator **cart_comm** representing a cartesian topology in **ndims** dimensons, of size **dims**.

2 (0,2)	5 (1,2)	8 (2,2)
1 (0,1)	4 (1 ,1)	7 (2,1)
0 (0,0)	3 (1,0)	6 (2,0)

ndims = 2
dims = (3,3)

Coordinates from rank

MPI_Cart_coords(**cart_comm, rank, ndims, coords**)

↓ output

2 (0,2)	5 (1,2)	8 (2,2)
1 (0,1)	4 (1 ,1)	7 (2,1)
0 (0,0)	3 (1,0)	6 (2,0)

```
> mpirun -np 9 ./a.out | sort
rank= 0 →  coords= 0 0
rank= 1 →  coords= 0 1
rank= 2 →  coords= 0 2
rank= 3 →  coords= 1 0
[...]
rank= 8 →  coords= 2 2
```

Rank from coordinates

MPI_Cart_rank (cart_comm, coords, &the_rank);

↓ output

2 (0,2)	5 (1,2)	8 (2,2)
1 (0,1)	4 (1 ,1)	7 (2,1)
0 (0,0)	3 (1,0)	6 (2,0)

```
> mpirun -np 9 ./a.out | sort
coords= 1 2 →   the_rank= 5
```

Neighbors

MPI_Cart_shift (cart2D_comm, direction, displacement, &neighbors[...], &neighbors[...]);

output

2 (0,2)	5 (1,2)	8 (2,2)
1 (0,1)	4 (1 ,1)	7 (2,1)
0 (0,0)	3 (1,0)	6 (2,0)

```
> mpirun -np 9 ./a.out | sort
rank= 0 → neighbors(w,e,s,n)= -2 3 -2 1
...
rank= 3 → neighbors(w,e,s,n)= 0 6 -2 4
rank= 4 → neighbors(w,e,s,n)= 1 7 3 5
...
rank= 8 → neighbors(w,e,s,n)= 5 -2 7 -2
```

Automatic dimensioning

MPI_Dims_create (ntasks, ndims, dims);

output (dims)

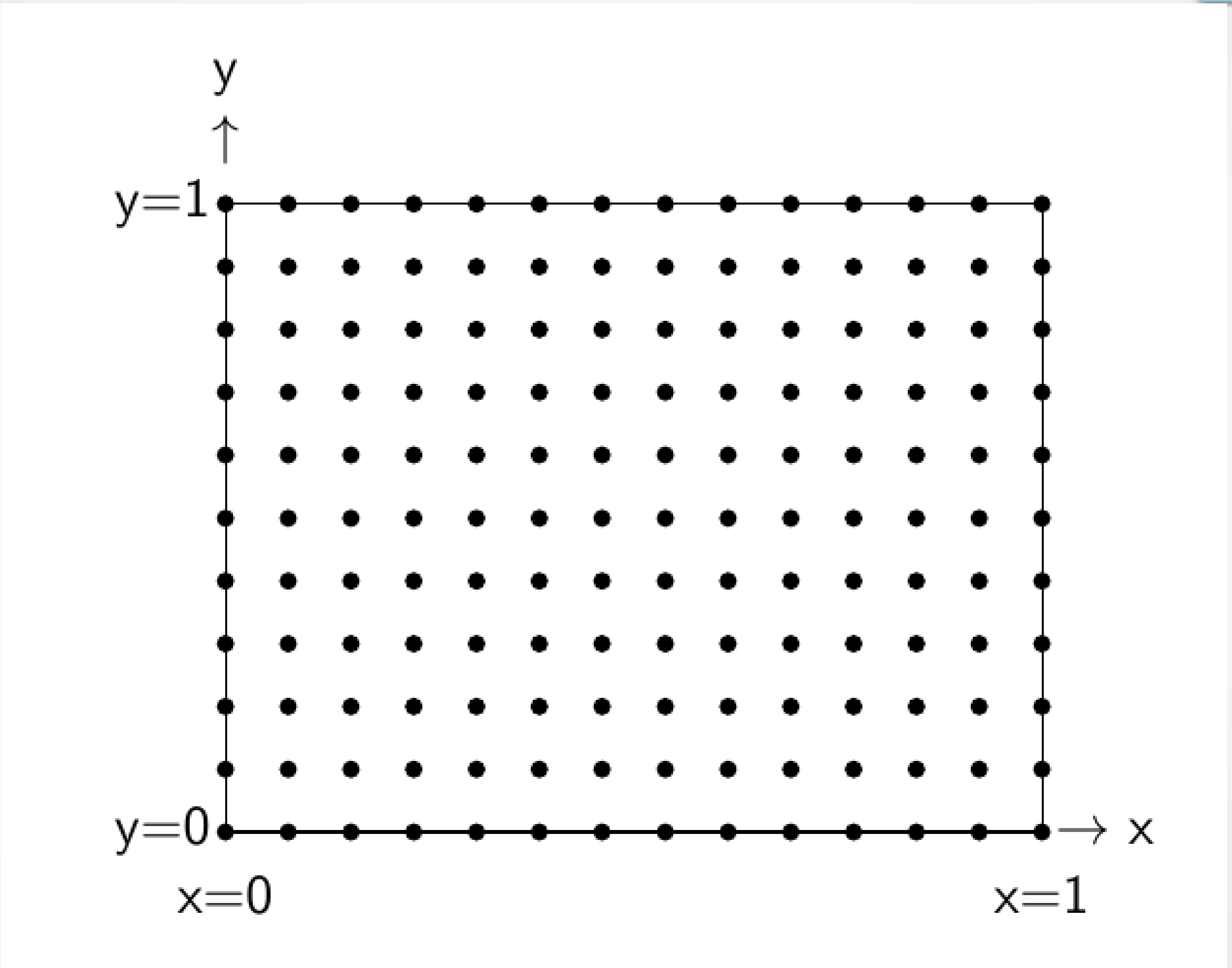
```
> mpirun -np 4 ./a.out
dims[0]= 2 dims[1]= 2

> mpirun -np 6 ./a.out
dims[0]= 3 dims[1]= 2
```

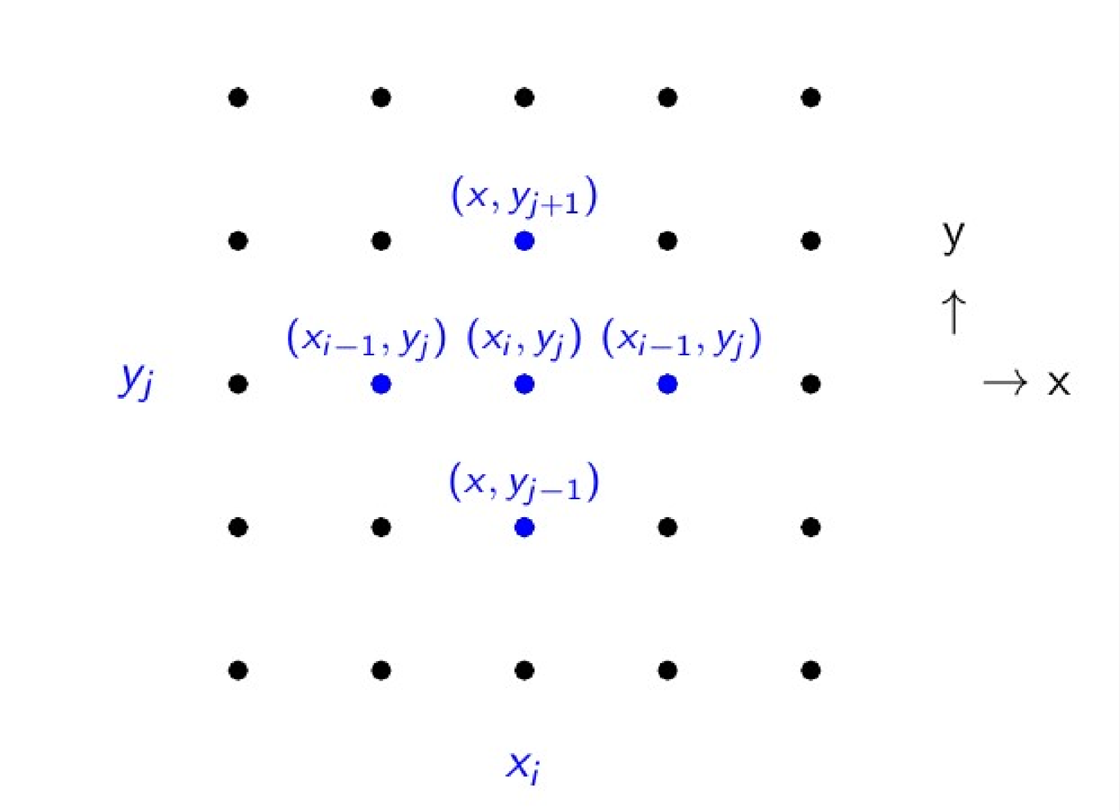
1 (0,1)	**3 (1 ,1)**
0 (0,0)	**2 (1,0)**

1 (0,1)	**3 (1 ,1)**	**5 (2,1)**
0 (0,0)	**2 (1,0)**	**4 (2,0)**

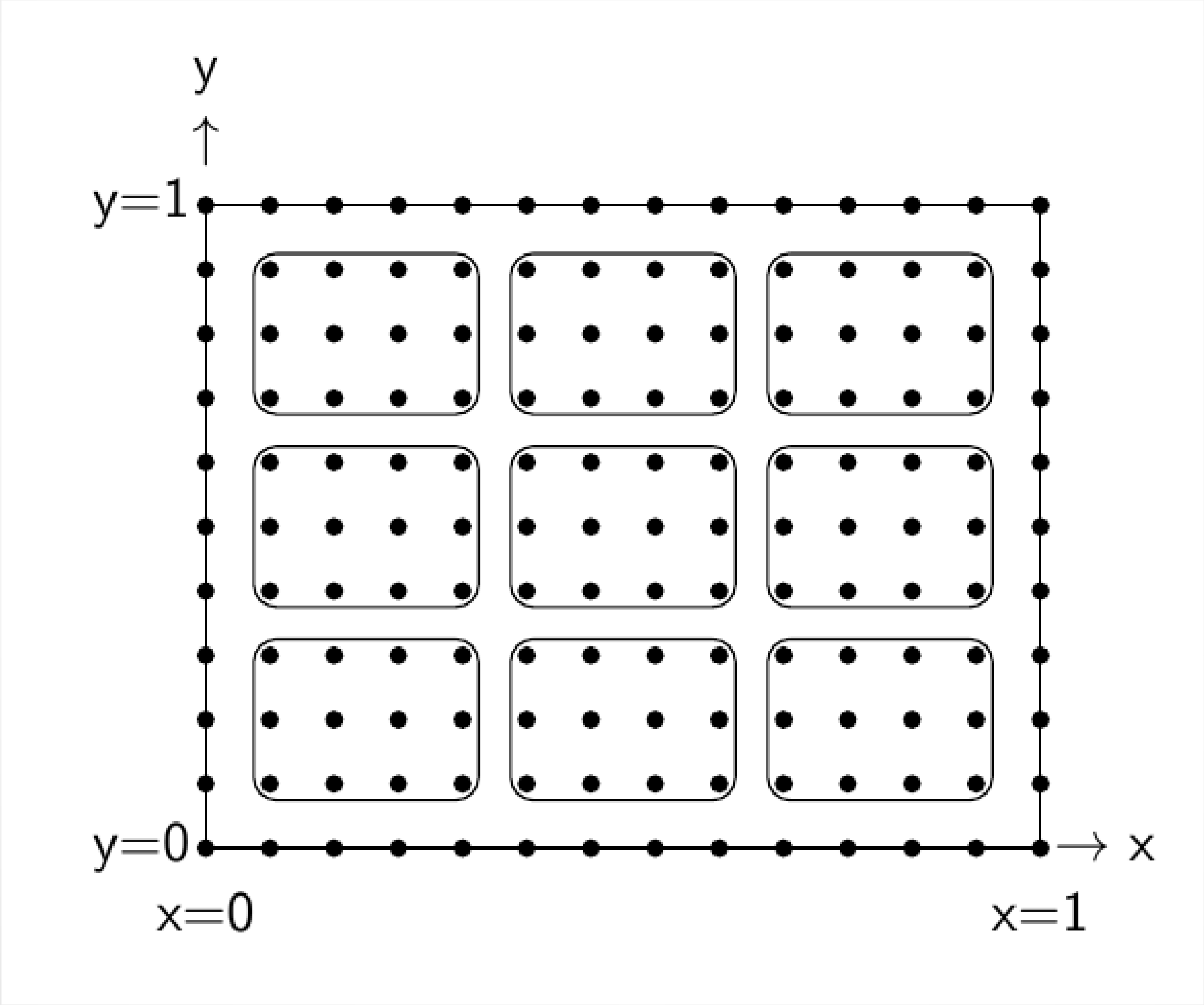
Illustration: PDE on a finite difference grid



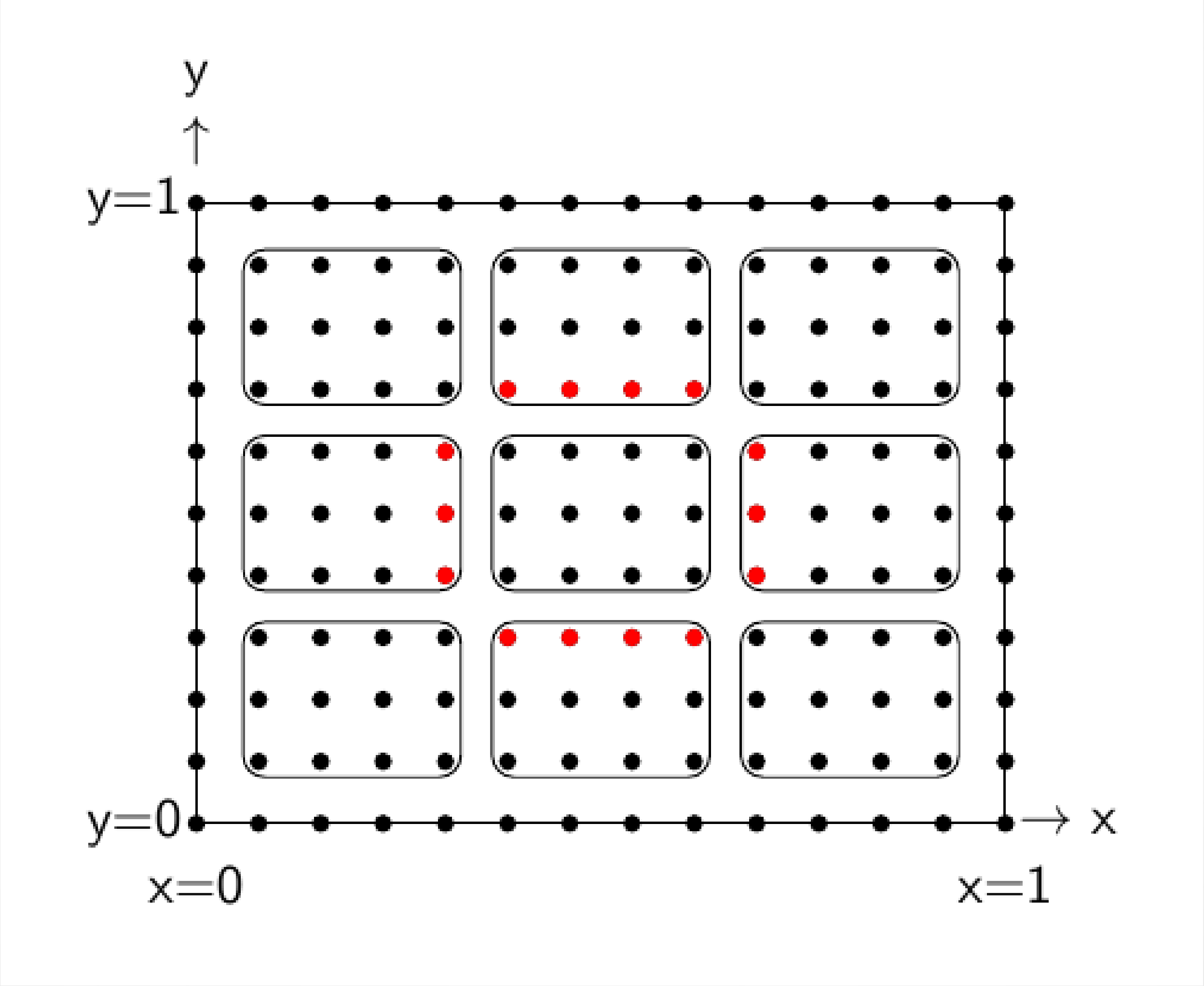
5-point finite difference scheme



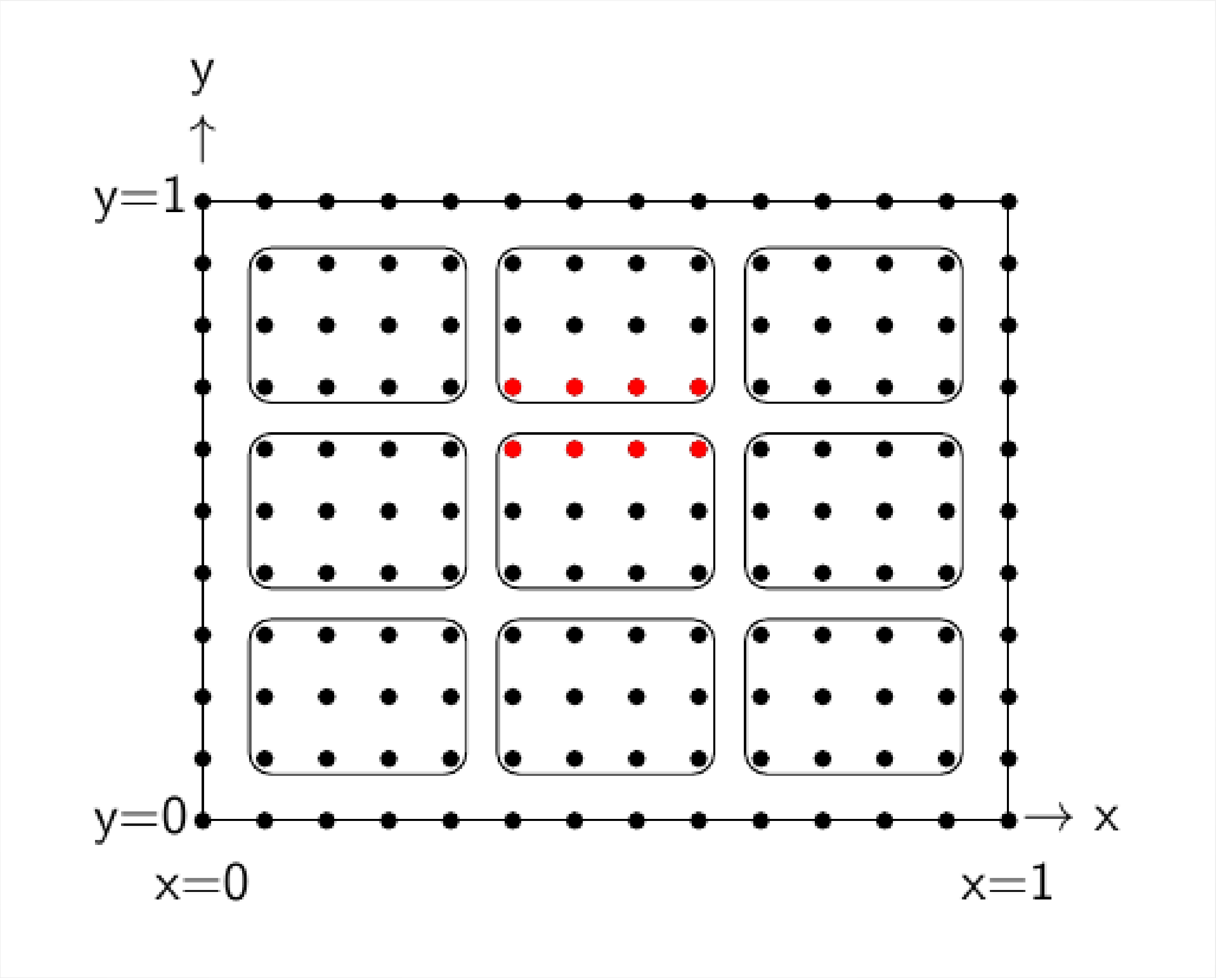
Let ntasks = 9



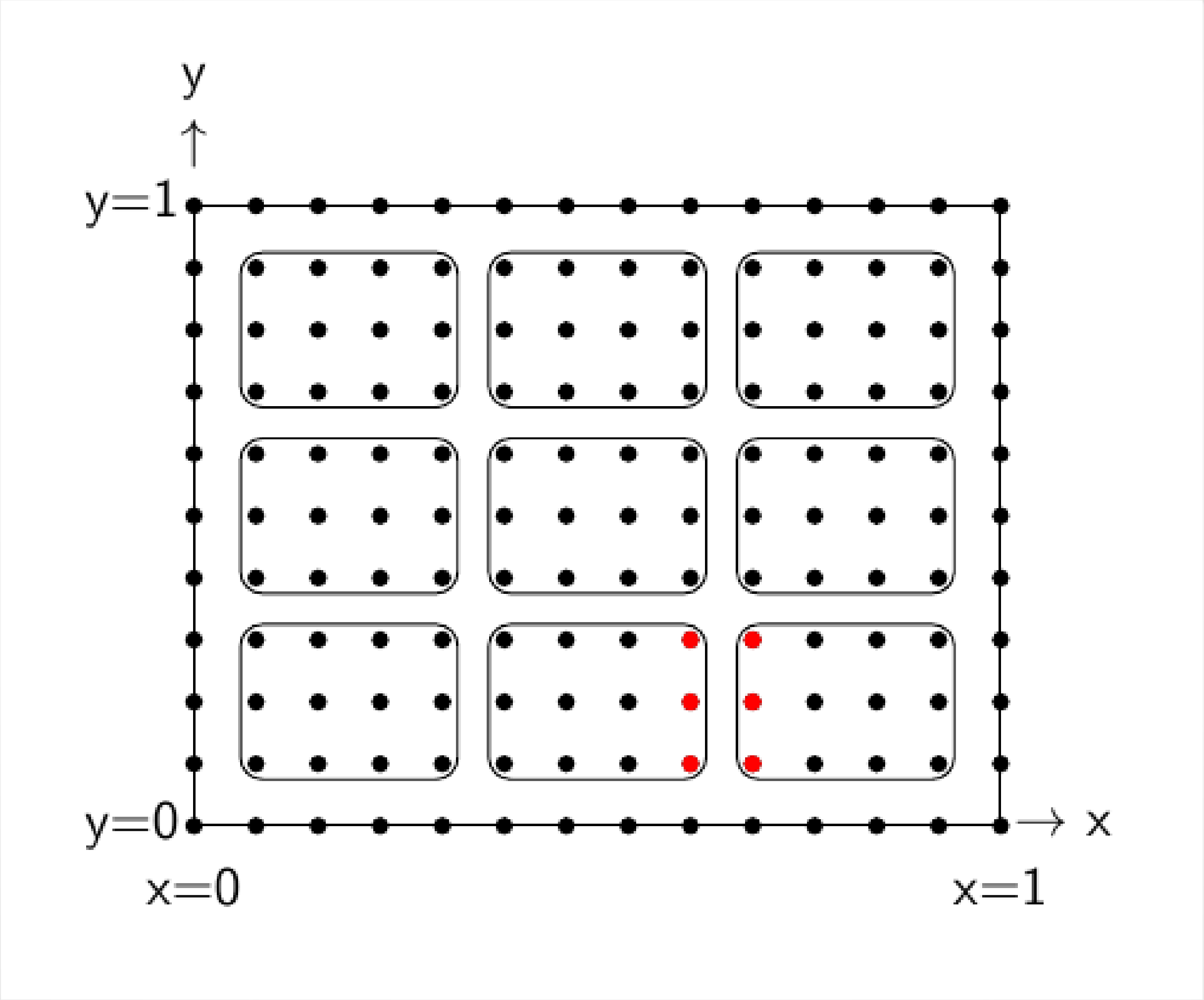
Ghost points



Exchanges needed...

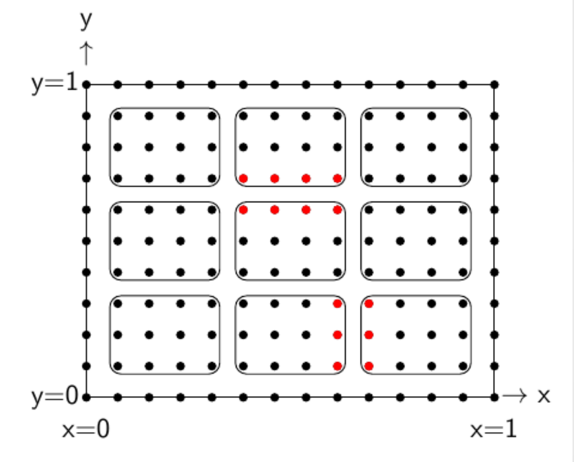


Exchanges needed...



Use cartesian topology! (and DDT)

- **MPI_Cart_create** → create topology
- **MPI_Cart_coords** → coordinates from rank
- **MPI_Cart_shift** → neighbors
- **Derived data types**:
  - ➔ “rowType” for North-South exchanges
  - ➔ “columnType” for East-West exchanges
- **MPI_Dims_create** for automatic dimensioning



Numerical results on MeCS' uv100

Poisson equation, Gauss-Seidel Iteration.

Ncpus	200*200 mesh
1	*32*
2	*16*
4	*8*
8	*4*

Perfect scaling

Times in seconds

Numerical results on MeCS' uv100

Poisson equation, Gauss-Seidel Iteration.

Ncpus	200*200 mesh
1	*32*
2	*16*
4	*8*
8	*4*
16	*8*

Perfect scaling

… but not above 8 cpus.

Times in seconds

Numerical results on MeCS' uv100

Poisson equation, Gauss-Seidel Iteration.

Ncpus	200*200 mesh	800*800 mesh
1	*32*	
2	*16*	
4	*8*	
8	*4*	*1052*
16	*8*	*524*

Times in seconds

Numerical results on MeCS' uv100

Poisson equation, Gauss-Seidel Iteration.

Ncpus	200*200 mesh	800*800 mesh
1	*32*	
2	*16*	
4	*8*	
8	*4*	*1052*
16	*8*	*524*
32		*900*

Times in seconds

N.B.: In real life...

Don't program PDE solver from scratch on your own!

Use existing tools, like:
PETSc, Trilinos, FeniCS, FreeFEM,...

Exercises

Exercise 1

- Compile and run the "Hello World" example on different numbers of processors by varying the -np argument. Have only a given MPI process (e.g. rank 0) print the "Hello World" message.

- Compile and run the given Send/Recv example **sendRecv.c/f90** on N = 2 processors. See what happens when launching on N ≠ 2 processors. Modify the code to have the integer 2 sent instead of the letter 'B' from rank 0 to rank 1.

Exercise 2

Write a MPI code such that each process exchanges its rank with its "partner" whose rank is defined as:

partner_rank = ntasks – (rank+1)

Have each MPI process print the integer it receives to check that it corresponds to the rank of its partner.

```
> mpirun -np 3 ./a.out
Rank 0 has partner rank 2
Rank 1 has partner rank 1
Rank 2 has partner rank 0
Integer received by rank 1 : 1
Integer received by rank 0 : 2
Integer received by rank 2 : 0
```

Exercise 3

From D. Lecas et al. (IDRIS)
http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html

The **coinTossSerial.c** program (next slide) simulates coin tossing (« *pile ou face* ») on one processor.

From there, write a parallel program that simulates simultaneous coin tossing on different MPI processes.

Then, build a program that observes toss results on all the MPI processes, and repeats the simultaneous coin tossing until unanimity is reached or until a given number of maximum attempts is reached.

Exercise 3

coinTossSerial.c

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

  srand(time(NULL));
  tossResult = (int) ((double)rand() / ((double)RAND_MAX + 1) * 2) ;
  // tossResult = 0 or 1

  printf("tossResult=%d \n", tossResult);
}
```

Exercise 3

coinTossSerial.f90

```
program main
!  use mpi
   implicit none
   include 'mpif.h'
   integer ::  tossResult, K
   integer, dimension(8) :: timeValues
   real :: random
   call date_and_time(values=timeValues)
   call random_seed(size=K)
   call random_seed(put=timeValues(1:K))
   call random_number(random)
   tossResult = nint(random)
   write(*,*) 'tossResult = ', tossResult
end
```