MPI Introduction: Part 2

MPI Introduction: Part 2

- Different communication modes
- Ghost points concept
- Exercise: analyzing an MPI Code

Communications: General information

- Communications within a communicator (`MPI_COMM_WORLD` or a user-defined one)
- Exchange of vector data (a scalar is a vector of size 1)
- Exchange of typed data (`MPI_INT` , `MPI_DOUBLE` , etc., or user-defined types
- Involves MPI functions with a large number of parameters (but predictable)
- Almost all MPI functions return an integer representing an error code

Les communications: Two families

- Collective Communications
- Point-to-Point Communications

Collective Communications: Reductions

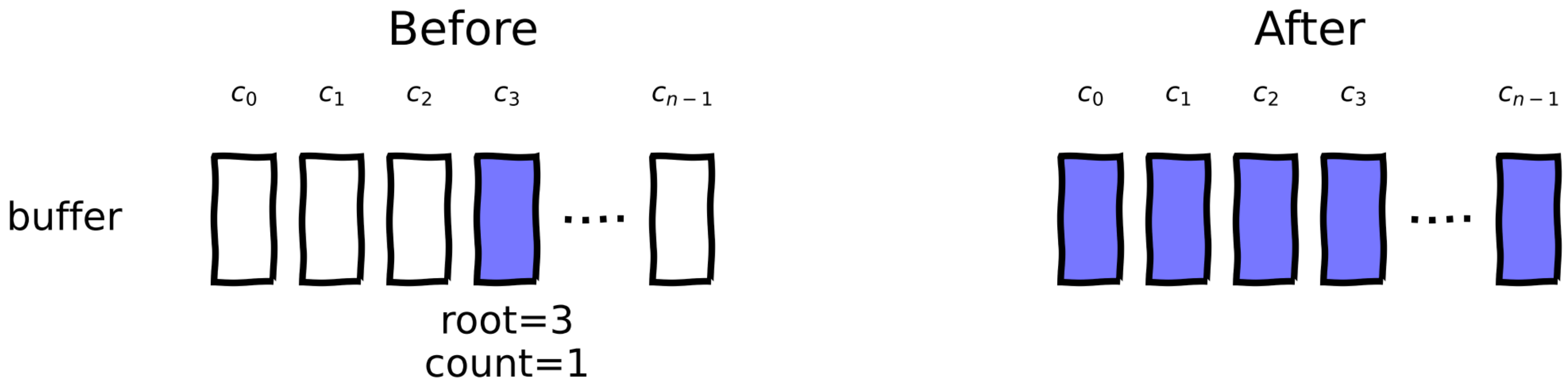
The reductions covered in the first part of this introduction to MPI are collective communications:

- `MPI_Reduce(...)`
- `MPI_Allreduce(...)`

Adding the prefix `All` to MPI functions means that the result will be available on all processes, not just on a single `root` process.

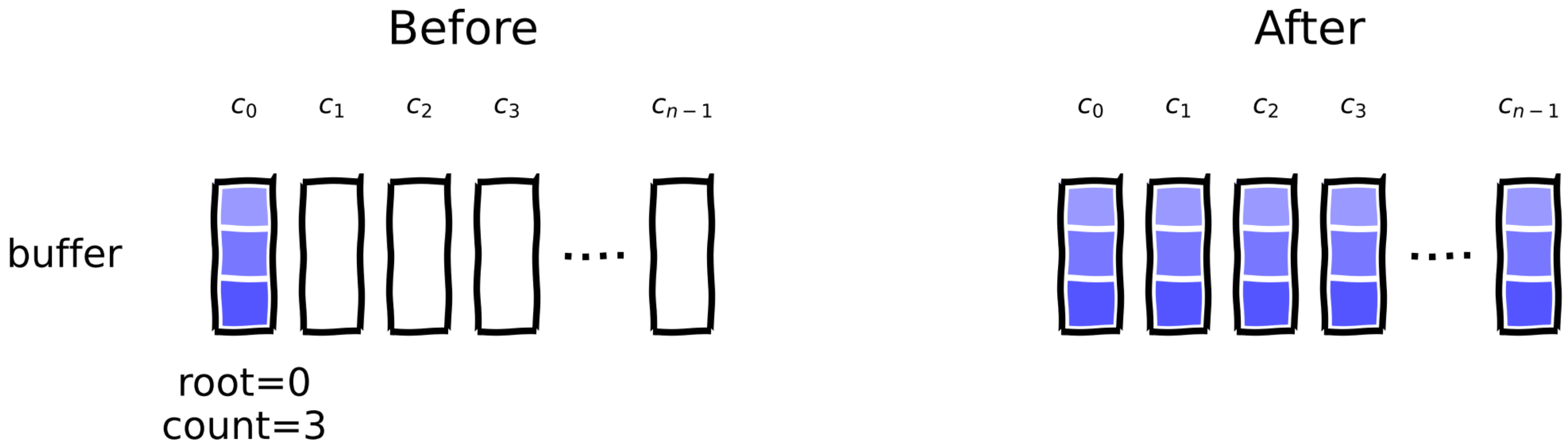
Collective Communications: MPI_Bcast()

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



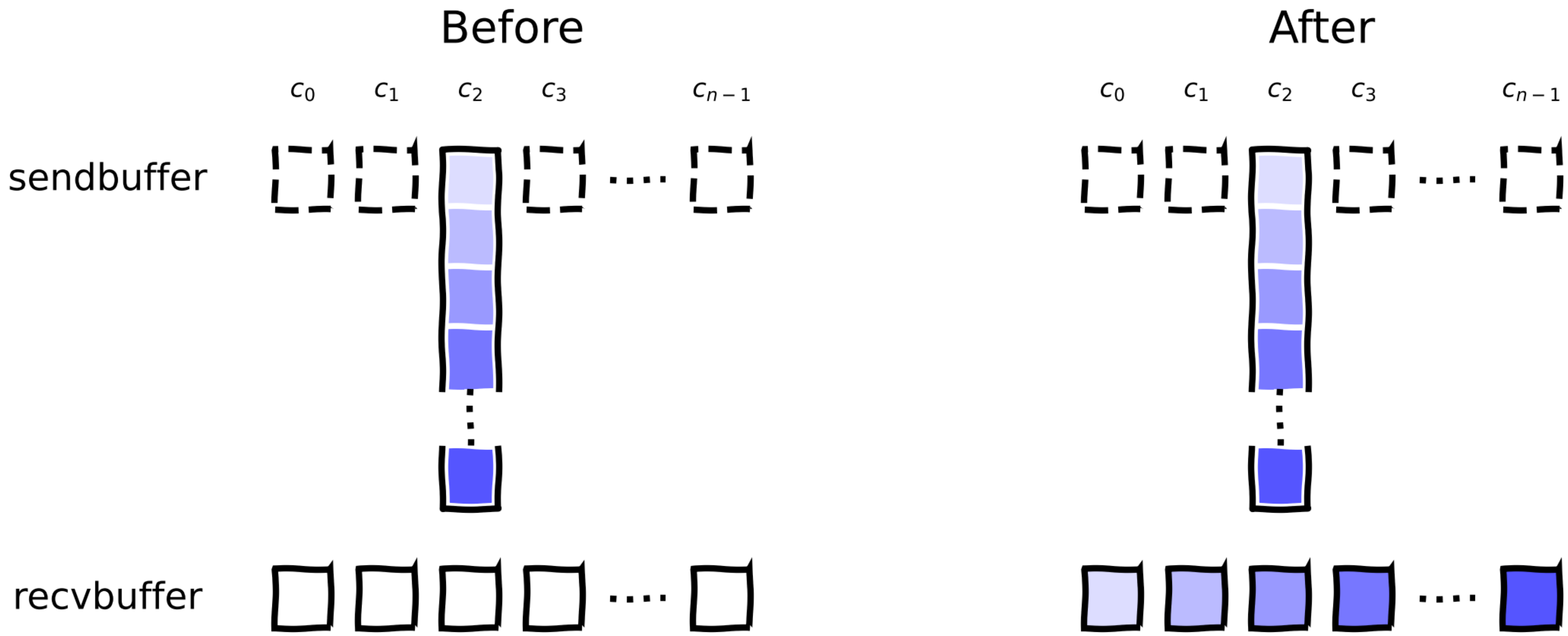
Collective Communications: MPI_Bcast()

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



Collective Communications: MPI_Scatter()

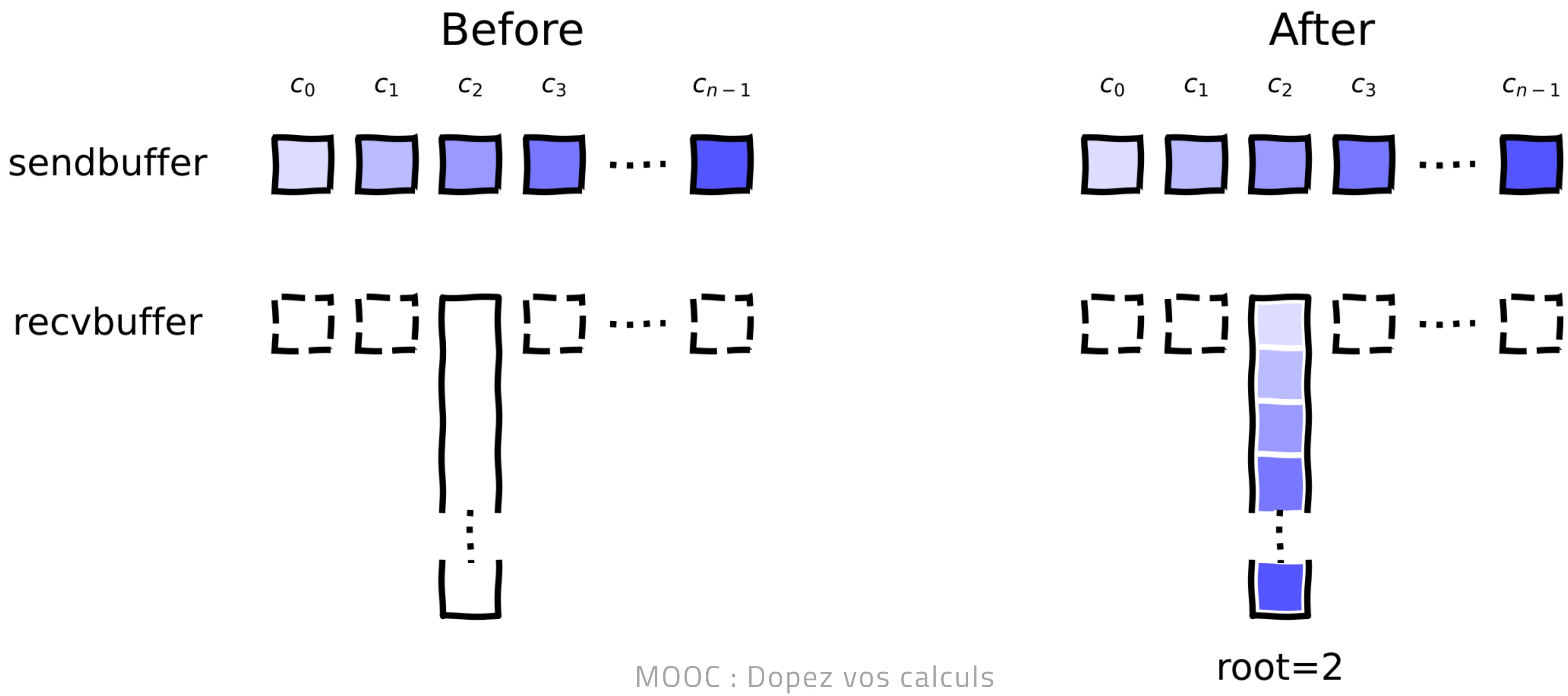
```
int MPI_Scatter(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
                void *recvbuffer, int recvcount, MPI_Datatype recvtype,
                int root,MPI_Comm comm);
```



root=2

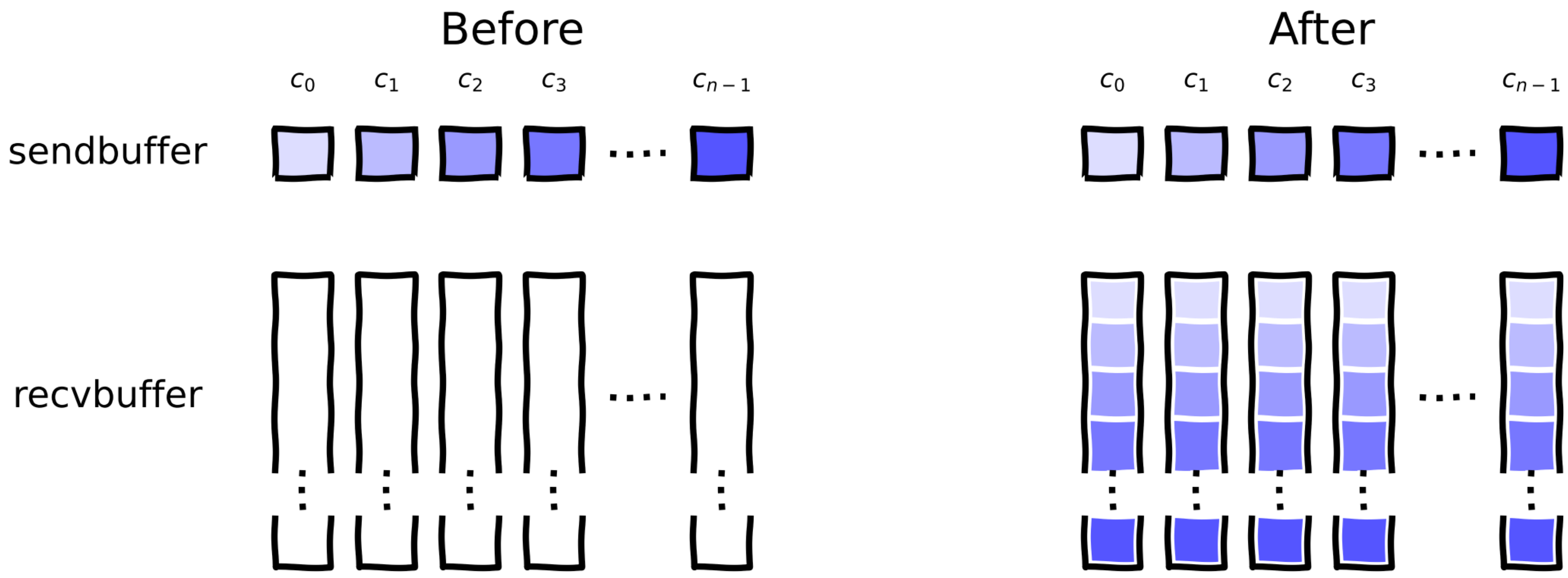
Collective Communications: MPI_Gather()

```
int MPI_Gather(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
               void *recvbuffer, int recvcount, MPI_Datatype recvtype,
               int root,MPI_Comm comm);
```



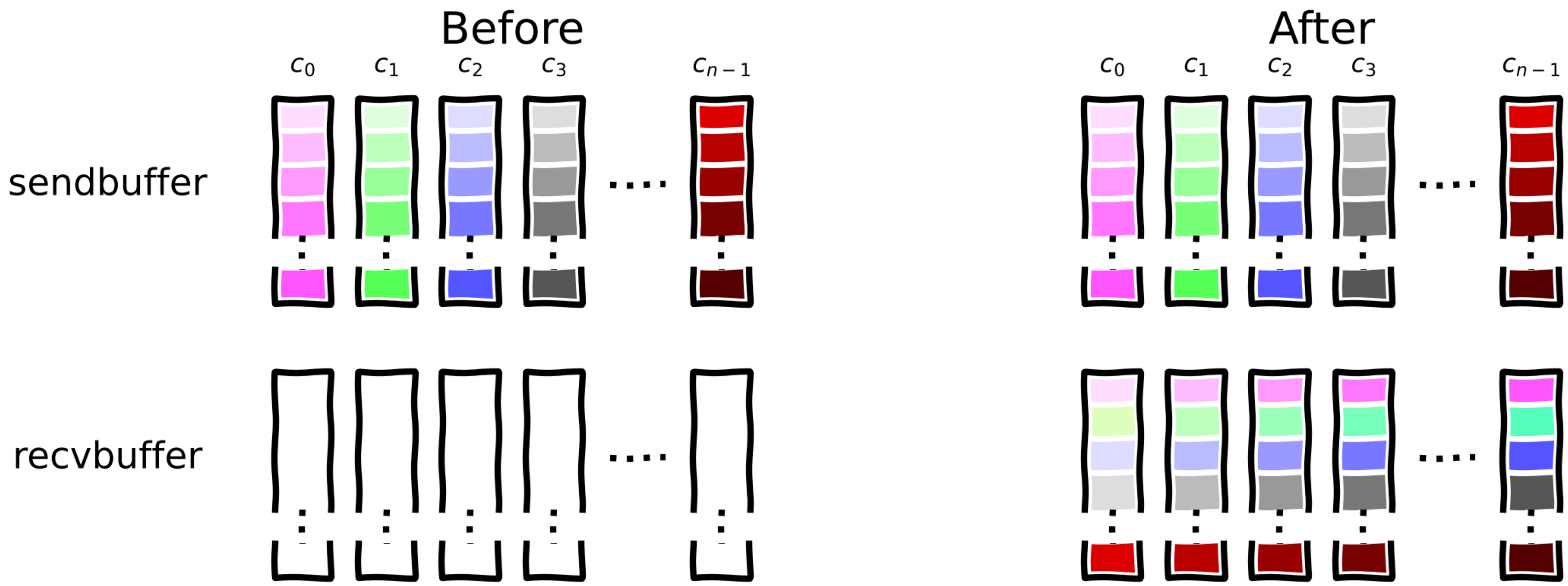
Collective Communications: MPI_Allgather()

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



Collective Communications: MPI_Alltoall()

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



Collective Communications: Heterogeneous version

For heterogeneous data in terms of quantity, you can add a **'v'** to the function name and provide additional arguments.

- MPI_Scatterv(...)
- MPI_Gatherv(...)
- MPI_Alltoallv(...)

```
int MPI_Alltoallv(
  void *sbuffer,int *scounts,int *sdispls,MPI_Datatype stype,
  void *rbuffer,int *rcounts,int *rdispls,MPI_Datatype rtype,
  MPI_Comm comm
);
```

P2P Communications

These are communications between:

- a sender who performs the message sending (**SEND**)
- a receiver who receives the message (**RECV**).

There are also two types of P2P communication:

- synchronous communications
- asynchronous communications

P2P Communications: Synchronous

- sending a message to `dest` with the identifier `tag`

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

- receiving a message from `src` with the identifier `tag`

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

- possible special parameters: `MPI_STATUS_IGNORE`, `MPI_ANY_SOURCE`, `MPI_ANY_TAG`

P2P Communications: Asynchrones prefix 'I'

- sending a message to `dest` with the identifier `tag`

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

- receiving a message from `src` with the identifier `tag`

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

- waiting for the completion of an action

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

Ghost Points: A local processing pattern

To achieve efficient parallelization of an operation, you need a **local** processing **pattern or kernel**.

Example: Image processing, noise reduction (Gaussian filter)

3x3 Filter	5x5 Filter

3x3 Filter: $\frac{1}{30}$ ×

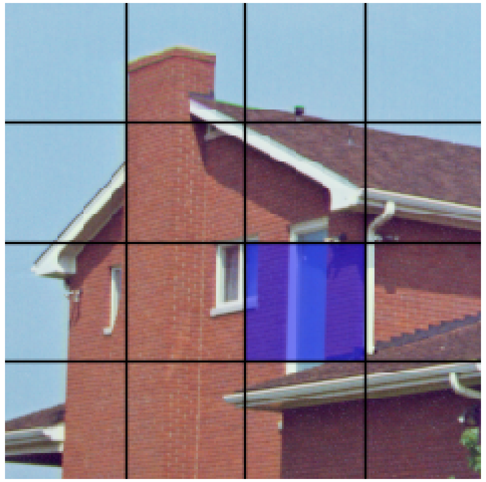
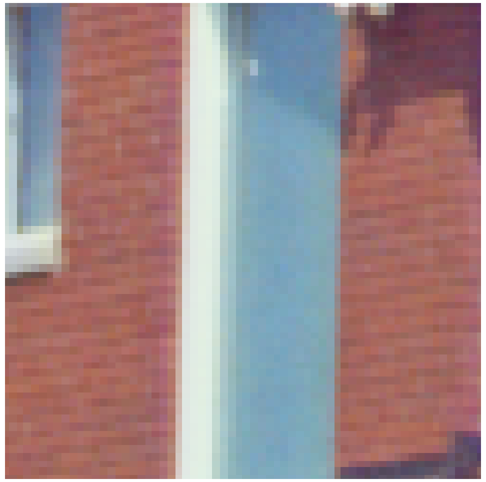
1	2	1
2	4	2
1	2	1

5x5 Filter: $\frac{1}{331}$ ×

1	4	7	4	1
4	20	33	20	4
7	33	55	33	7
4	20	33	20	4
1	4	7	4	1

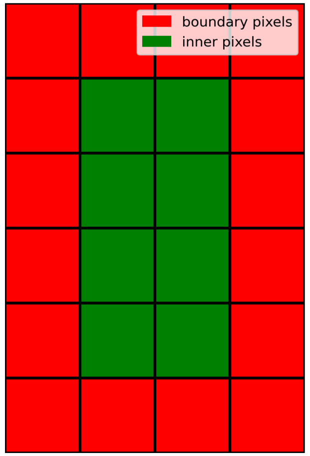
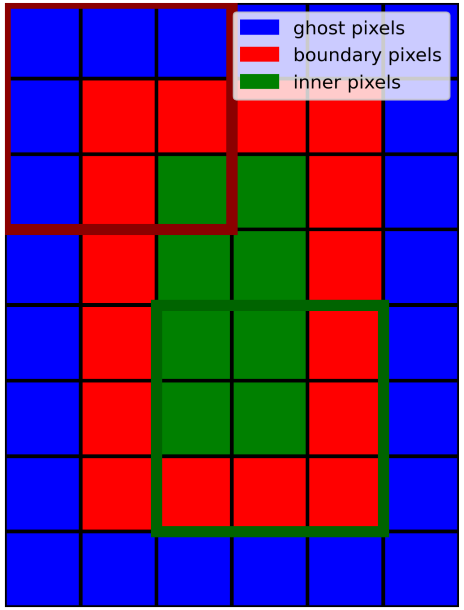
Ghost Points: Subdomain partitioning

The image is **partitioned** into multiple subdomains hosted on different cores (here, 16 cores).

Original Image	Partitioned Image	Sub-Image
		

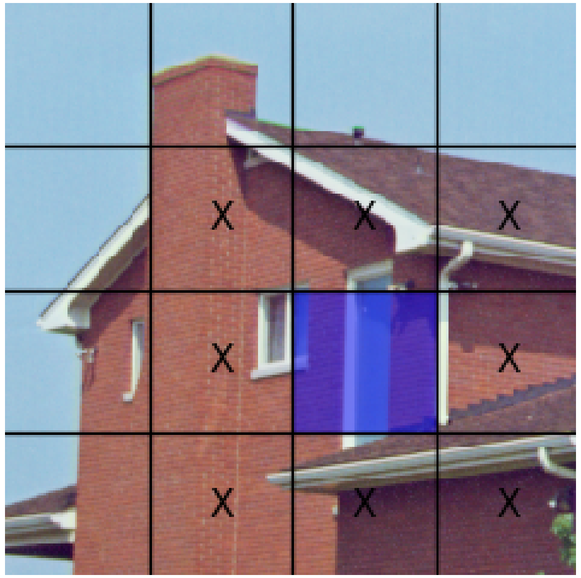
Ghost Points: Extended subdomains

The subdomains are **extended** with a layer of **ghost pixels** to store a copy of neighboring pixels.

Sub-Image	Extended Sub-Image
	

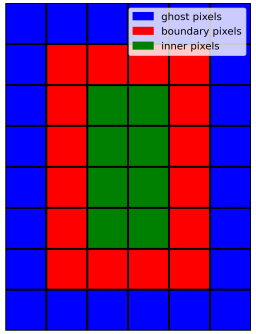
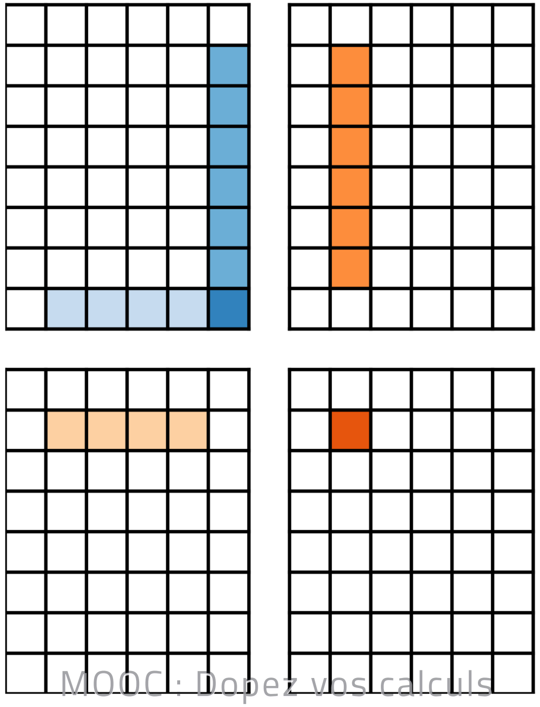
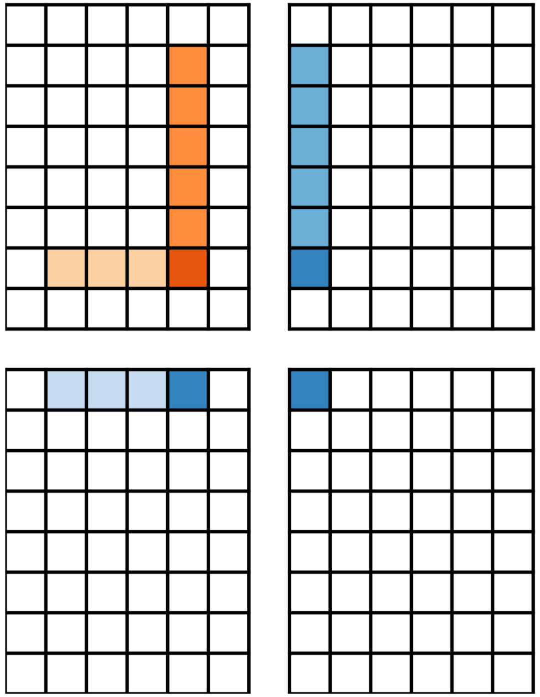
Ghost Points: Update & Communications

Updating the ghost pixels requires communications only with neighboring subdomains/cores (here marked with **X** and up to 8 at most).



Ghost Points: Update & Communications

Illustration of exchanges (symmetry: lower right corner only)

Sub-Image	Reception	Sends
		

Ghost Points: Remarks

- Each subdomain/core only communicates with its close neighbors.
  - here, a maximum of 8 even on thousands of cores.
- The number of exchanged pixels is **small** compared to the number of pixels in the subdomain.
  - for a sub-image of **512x512 pixels**, only 4104 = 2* (4*512+4) pixels are exchanged out of 262,144 (so **only 1.6%**).

Parallélisation avec MPI

You have grasped:

- the different communication modes: collective, synchronous point-to-point, and asynchronous point-to-point
- the setup of ghost points, which are widely used in MPI codes

Exercise: Analyze an MPI code