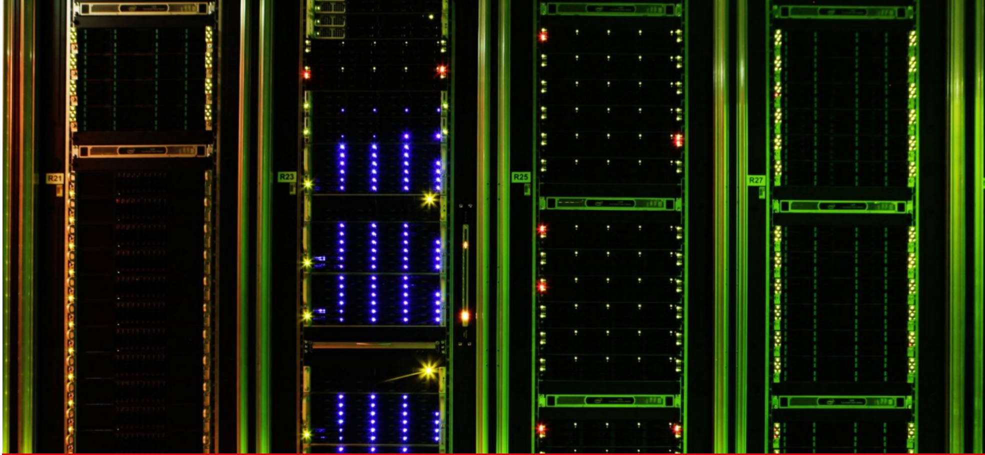
EPFL



# MATH-454 Parallel and High Performance Computing
# Lecture 6: Hybrid MPI / OpenMP and mpi4py

Pablo Antolin

Slides of N. Richart, E. Lanti, V. Keller's lecture notes

April 3 2025

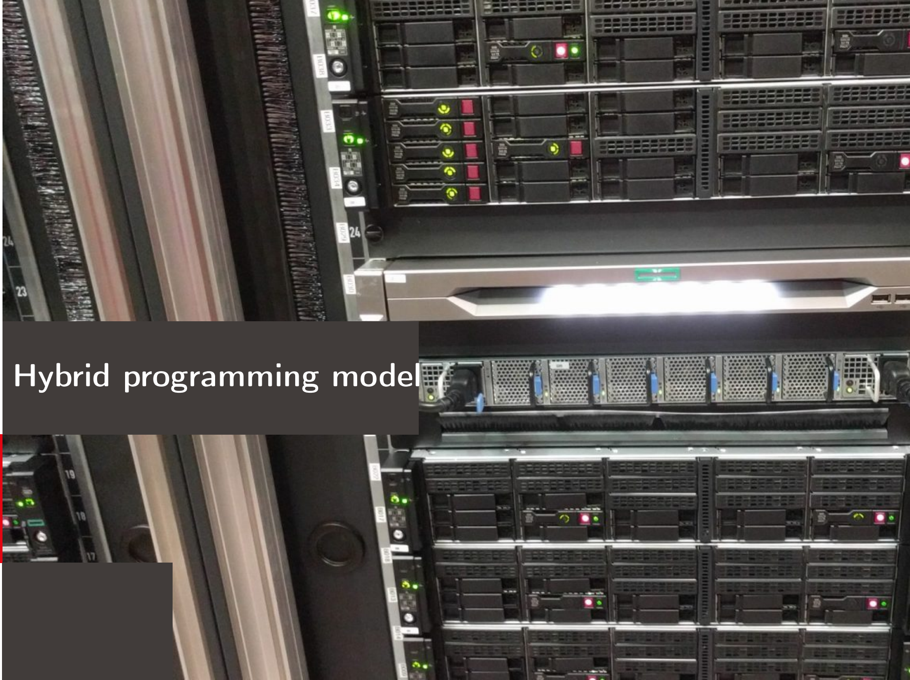
SCITAS

# What will we learn today?

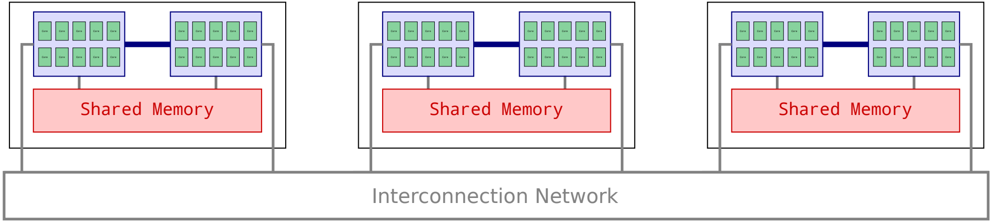
- Hybrid MPI + OpenMP programming
  - Introduction
  - Partitioned point-to-point communications
  - Matching probe/receive
- MPI for Python
- This week's exercise

EPFL

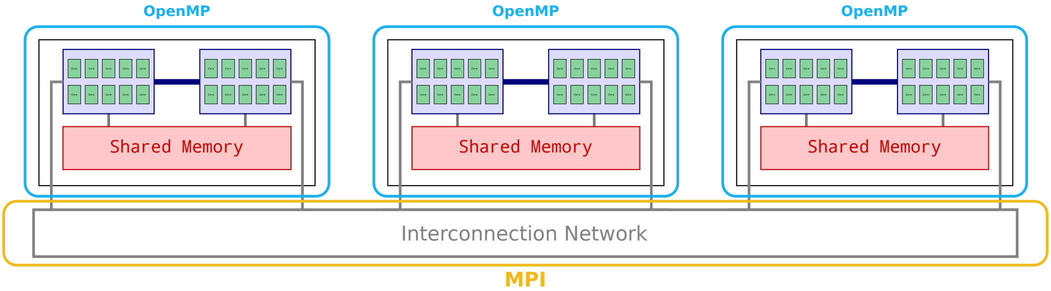
# Hybrid programming model



# Situation


Shared Memory
Shared Memory
Shared Memory
Interconnection Network

# Situation



OpenMP
OpenMP
OpenMP
Shared Memory
Shared Memory
Shared Memory
Interconnection Network
MPI

- Thread safety? Data visibility? OpenMP private?
- Which thread/process can/will call the MPI library?
- MPI process placement in the case of multi-CPU processors?
- Does my problem fit with the targeted machine?
- Levels of parallelism within my problem?

## Pure MPI

- + no code modification (portability).
- + most of the libraries support multi-thread (*e.g., BLAS libraries*). Thread safety
- – does the application's topology fit the system's topology?
- – useless communications and repeated memory.

## Hybrid

- + no messages within a SMP node.
- + less (no) topology problems.
- – all threads sleep when master communicates.
- – MPI-libs must support (at least) thread safety.

# Topology problems

How to deal with:

- topology / mapping? (Which physical core is assigned to which process/thread)
- sub-domain decomposition?
- halos (ghost) size? halos shapes?
- uncessary communications?
- **computation to communication ratio?**

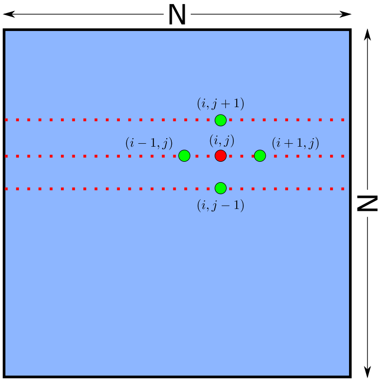
Pure MPI? Hybrid?

**A good solution may be: one MPI process per SMP node.**

# Halo regions

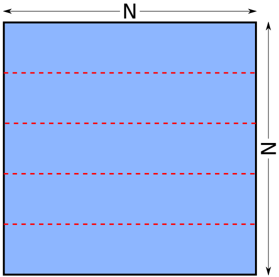
- Halo regions are local copies of remote data that are needed for communications (ghost rows in Poisson problem)
- Halo regions need to be copied frequently.
- Using threads reduces the size of halo regions copies that need to be stored.
- Reducing halo region sizes also reduces communication requirements.

# Example of halo regions



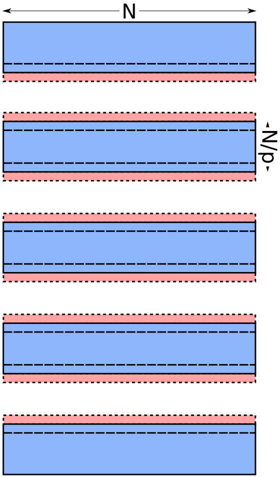

$$u(i,j) = \frac{1}{4}\left(u_{\text{old}}(i-1,j) + u_{\text{old}}(i+1,j) + u_{\text{old}}(i,j-1) + u_{\text{old}}(i,j+1) - f(i,j)\,h_m\,h_n\right)$$

# Example of halo regions



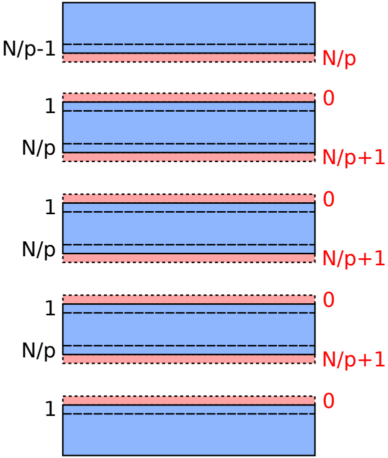

- The domain is decomposed by lines.

# Example of halo regions



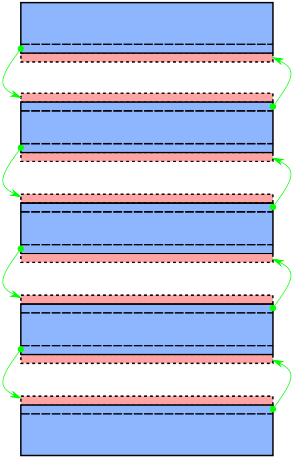

- $p$ domains of size $N/p$ each (1 per process).

# Example of halo regions




- Adding *ghost* lines (halo regions) before and after

# Example of halo regions



- Use *ghost* lines to communicate information among pairs of processes.

## Take-home messages

- Always take into account the problems related to the physical topology.
- A real application is not as easy as a *hello world*.
- Some clusters have different connectivity topologies: Match them to your problem. Examples of hardware topologies:
  - all-to-all
  - 2D/3D torus
  - three
  - ...
- One MPI process per physical node.

# Main messages

- Do not use hybrid if the pure MPI code scales ok.
- Be aware of intranode MPI behavior.
- Always observe the topology dependence of:
  - Intranode MPI.
  - Threads' overheads.
- Finally: Always compare the best pure MPI code with the best hybrid code!

# Examples that *can* benefit from a hybrid approach

- MPI codes with a lot of all-to-all communications.
- MPI codes with a very poor load balancing at the algorithmic level (less communications).
- MPI codes with memory limitations.
- MPI codes that can be easily *fine-grained* parallelized (at loop level).

# Hybrid MPI/OpenMP hello world

hybrid/hello_world.cc

```
#include <iostream>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
  int provided, size, rank, nthreads, tid;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #pragma omp parallel default(shared) private(tid, nthreads)
  {
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    std::printf("Hello from thread %i out of %i from process %i out of %i\n", tid,
    ↪  nthreads, rank, size);
  }
  MPI_Finalize();
  return 0;
```

# Hybrid MPI/OpenMP hello world

Compilation using the GNU g++ compiler:

```
$> mpicxx -fopenmp hello_world.cc -o hello_world
```

Compilation using the Intel C++ compiler:

```
$> mpiicpc -fopenmp hello_world.cc -o hello_world
```

## Submission script the clusters

```
#!/bin/bash
#SBATCH --ntasks 2
#SBATCH --nodes 2
#SBATCH --cpus-per-task 3
#SBATCH --ntasks-per-node 1
#SBATCH --qos math-454
#SBATCH --account math-454

export OMP_NUM_THREADS=3
srun ./hello_world
```

# Submission script the clusters

It will start 2 MPI processes and each one will spawn 3 threads

```
Hello from thread 0 out of 3 from process 0 out of 2
Hello from thread 1 out of 3 from process 0 out of 2
Hello from thread 0 out of 3 from process 1 out of 2
Hello from thread 1 out of 3 from process 1 out of 2
Hello from thread 2 out of 3 from process 0 out of 2
Hello from thread 2 out of 3 from process 1 out of 2
```

# Changes to your code

- Change your MPI initialization routine
  - **MPI_Init** is replaced by **MPI_Init_thread**
  - **MPI_Init_thread** has two additional parameters for the level of thread support required, and for the level of thread support provided by the library implementation

```c
int MPI_Init_thread(int *argc, char ***argv, int required, int
    *provided)
```

- Make sure that the *provided* support matches the *required* one

```c
if (provided < required)
  MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
```

- Add OpenMP directives as long as you stick to the level of thread safety you specified in the call to **MPI_Init_thread**

# The 4 options for thread support

- MPI_THREAD_SINGLE
  - Only one thread will execute (no multi-threading)
  - Standard MPI-only application
- MPI_THREAD_FUNNELED
  - Only the Master Thread will make calls to the MPI library
  - A thread can determine whether it is the master thread by a call to `MPI_Is_thread_main`
- MPI_THREAD_SERIALIZED
  - Only one thread at a time will make calls to the MPI library, but all threads are eligible to make such calls
- MPI_THREAD_MULTIPLE
  - Any thread may call the MPI library at any time

# The 4 options for thread support

- **MPI_THREAD_SINGLE**
  - Only one thread will execute (no multi-threading)
  - Standard MPI-only application
- **MPI_THREAD_FUNNELED**
  - Only the Master Thread will make calls to the MPI library
  - A thread can determine whether it is the master thread by a call to `MPI_Is_thread_main`
- **MPI_THREAD_SERIALIZED**
  - Only one thread at a time will make calls to the MPI library, but all threads are eligible to make such calls
- **MPI_THREAD_MULTIPLE**
  - Any thread may call the MPI library at any time

In most cases **MPI_THREAD_FUNNELED** provides the best choice for hybrid programs

# The 4 options for thread support

- Thread support values are monotonic, i.e. **MPI_THREAD_SINGLE** < **MPI_THREAD_FUNNELED** < **MPI_THREAD_SERIALIZED** < **MPI_THREAD_MULTIPLE**
- Gets the maximum level of thread support provided by the MPI library

```c
int MPI_Query_thread(int *thread_level_provided);
```

- Different processes in **MPI_COMM_WORLD** can have different thread safety
- The level(s) of provided thread support depends on the implementation

# MPI partitioned communications

# MPI partitioned communications

- New feature from MPI 4.0 standard (June 2021!)
- We have already talked about persistent point-to-point communications
- Partitioned comms are just persistent comms where the message is constructed in partitions
- Typical case: multi-threading with each thread building a portion of the message

# MPI partitioned communications

- Remember the typical cycle for persistent point-to-point communications

  Init (Start Test/Wait)* Free

  where * means zero or more
- Partitioned are very similar

  PInit (Start PReady)* Free

```
MPI_Psend_init(msg, parts, count, MPI_INT, dest, tag, info, MPI_COMM_WORLD, &request);
MPI_Start(&request);
#pragma omp parallel for shared(request)
for (int i = 0; i < parts; ++i) {
    /* compute and fill partition #i of msg, then mark ready: */
    MPI_Pready(i, request);
  }
while(!flag) {
    /* Do useful work */
    MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
    /* Do useful work */
  }
MPI_Request_free(&request);
```

# MPI matching probe

# Probing reminder

**Syntax**

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status);

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status
 ↪  *status);
```

- Check incoming messages without receiving.
- Immediate variant returns **true** if matching message exists.
- Can be used in combination with a successive `MPI_Get_count` for deducing the size of an incoming message before actually recieving it. Thus, we can allocate a buffer for holding the message.

# MPI matching probe

- We have already talked before about **MPI_Probe** to obtain information about a message waiting to be received
- This is typically used when the size of the message is unknown (probe, allocate, receive)

# MPI matching probe

- We have already talked before about **MPI_Probe** to obtain information about a message waiting to be received
- This is typically used when the size of the message is unknown (probe, allocate, receive)
- Care must be taken because it is a stateful method:
  *A subsequent receive [...] will receive the message that was matched by the probe,* ***if no other intervening receive occurs after the probe*** *[...]*

# MPI matching probe

- We have already talked before about **MPI_Probe** to obtain information about a message waiting to be received
- This is typically used when the size of the message is unknown (probe, allocate, receive)
- Care must be taken because it is a stateful method:
  *A subsequent receive [...] will receive the message that was matched by the probe,* ***if no other intervening receive occurs after the probe [...]***
- Problem with multi-threading!
- Imagine two threads $A$ and $B$ that must do a Probe, Allocation, and Receive

$$A_P \longrightarrow A_A \longrightarrow A_R \longrightarrow B_P \longrightarrow B_A \longrightarrow B_R$$

but may also be

$$A_P \longrightarrow B_P \longrightarrow B_A \longrightarrow B_R \longrightarrow A_A \longrightarrow A_R$$

Thread $B$ stole thread $A$'s message!

# MPI matching probe

- The solution of this problem is the matching probe
- MPI provides two versions, **MPI_Improbe** and **MPI_Mprobe**
- It allows to receive only a **MPI_Message** matching a specific probe

# MPI matching probe

- The solution of this problem is the matching probe
- MPI provides two versions, **MPI_Improbe** and **MPI_Mprobe**
- It allows to receive only a **MPI_Message** matching a specific probe

- Counterpart operations are the matching receive **MPI_Imrecv** and **MPI_Mrecv**
- They are used to receive messages that have been previously matched by a matching probe

## Concluding remarks

- Always keep in mind that you are mixing (OpenMP) threads and (MPI) processes
- You will need to test your code performance on every machine
- There are no magic rules on the best configuration to use
- Often 1 MPI task per NUMA region seems to give the best performance

# MPI for Python

SCITAS

# mpi4py

- Python wrappers for MPI.
- Covers most of the features.
- Much simpler than Fortran and C interfaces.
- Based on `pickle` serialization.

# Learning by example: Hello, World

mpi4py/ex_0.py

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print(f'I am process {rank} out of
↪ {size}.')
```

Output

```
I am process 1 out of 4.
I am process 3 out of 4.
I am process 0 out of 4.
I am process 2 out of 4.
```

```
$> module load intel intel-oneapi-mpi
$> # or module load gcc openmpi
$> module load python py-mpi4py
$> srun -n 4 python ex_0.py
```

# Learning by example: Simple communication

mpi4py/ex_1.py

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
else:
    data = None
print(rank, data)
```

Output

```
0 {'a': 7, 'b': 3.14}
2 None
3 None
1 {'a': 7, 'b': 3.14}
```

# Learning by example: Non-blocking communication

mpi4py/ex_2.py

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
else:
    data = None
print(rank, data)
```

Output

```
0 {'a': 7, 'b': 3.14}
1 {'a': 7, 'b': 3.14}
2 None
3 None
```

# Learning by example: efficient numpy arrays

mpi4py/ex_3.py

```python
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# passing MPI datatypes explicitly
data = None
if rank == 0:
    data = np.arange(10, dtype='i')
    comm.Send([data, MPI.INT], dest=1,
    ↪  tag=77)
elif rank == 1:
    data = np.empty(10, dtype='i')
    comm.Recv([data, MPI.INT], source=0,
    ↪  tag=77)
print(rank, data)
```

Output

```
0 [0 1 2 3 4 5 6 7 8 9]
1 [0 1 2 3 4 5 6 7 8 9]
2 None
3 None
```

# Learning by example: efficient numpy arrays

mpi4py/ex_4.py

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# automatic MPI datatype discovery
data = None
if rank == 0:
    data = np.arange(10, dtype=np.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = np.empty(10, dtype=np.float64)
    comm.Recv(data, source=0, tag=13)
print(rank, data)
```

Output

```
0 [0. 1. 2. 3. 4. 5. 6.
↪  7. 8. 9.]
1 [0. 1. 2. 3. 4. 5. 6.
↪  7. 8. 9.]
2 None
3 None
```

# Learning by example: collective communications (broadcast)

mpi4py/ex_5.py

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : 0,
            'key2' : ('abc', 'xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
print(rank, data)
```

Output

```
0 {'key1': 0, 'key2':
↪ ('abc', 'xyz')}
1 {'key1': 0, 'key2':
↪ ('abc', 'xyz')}
3 {'key1': 0, 'key2':
↪ ('abc', 'xyz')}
2 {'key1': 0, 'key2':
↪ ('abc', 'xyz')}
```

# Learning by example: collective communications (scatter)

mpi4py/ex_6.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
print(rank, data)
```

Output

```
0 1
1 4
2 9
3 16
```

# Learning by example: collective communications (gather)

mpi4py/ex_7.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
else:
    assert data is None
print(rank, data)
```

Output

```
2 None
0 [1, 4, 9, 16]
1 None
3 None
```

# Learning by example: numpy collective communications (broadcast)

mpi4py/ex_8.py

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(5, dtype='i')
else:
    data = np.empty(5, dtype='i')
comm.Bcast(data, root=0)
for i in range(5):
    assert data[i] == i
print(rank, data)
```

Output

```
0 [0 1 2 3 4]
1 [0 1 2 3 4]
2 [0 1 2 3 4]
3 [0 1 2 3 4]
```

# Learning by example: numpy collective communications (gather)

mpi4py/ex_9.py

```python
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = np.zeros(5, dtype='i') + rank
recvbuf = None
if rank == 0:
    recvbuf = np.empty([size, 5], dtype='i')
comm.Gather(sendbuf, recvbuf, root=0)
if rank == 0:
    for i in range(size):
        assert np.allclose(recvbuf[i,:], i)
print(rank, recvbuf)
```

Output

```
3 None
0 [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]]
2 None
1 None
```

# Learning by example: numpy collective communications (scatter)

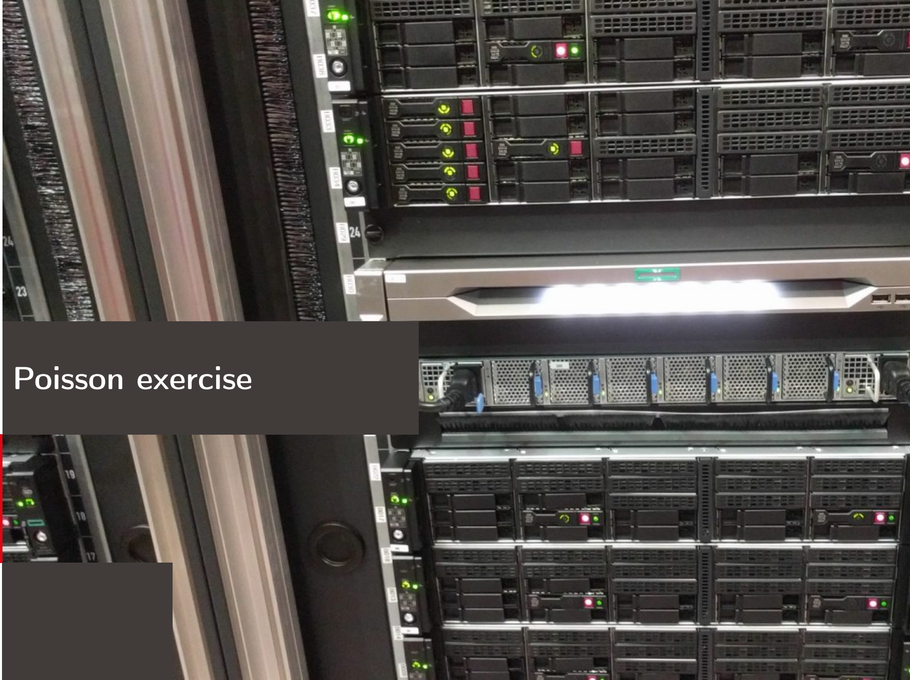
mpi4py/ex_10.py

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = None
if rank == 0:
    sendbuf = np.empty([size, 5], dtype='i')
    sendbuf.T[:,:] = range(size)
recvbuf = np.empty(5, dtype='i')
comm.Scatter(sendbuf, recvbuf, root=0)
assert np.allclose(recvbuf, rank)
print(rank, recvbuf)
```

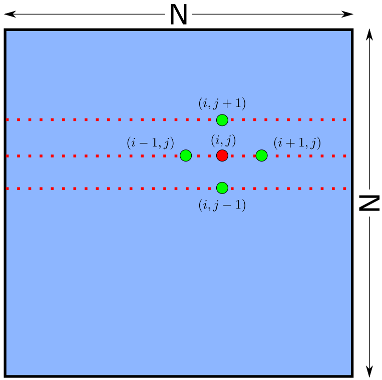
Output

```
3 [3 3 3 3 3]
1 [1 1 1 1 1]
2 [2 2 2 2 2]
0 [0 0 0 0 0]
```

EPFL



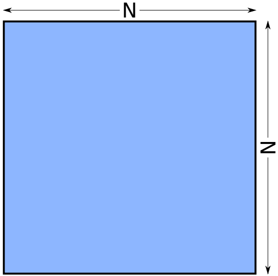
# Poisson exercise




This finite difference (5 points stencil) computes the solution of the Poisson equation in 2D in an iterative manner. The equation is given by:

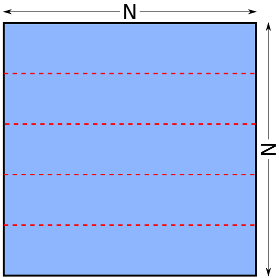
$$u(i,j) = \frac{1}{4}\left(u_{\text{old}}(i-1,j) + u_{\text{old}}(i+1,j)\right.$$
$$\left. + u_{\text{old}}(i,j-1) + u_{\text{old}}(i,j+1) - f(i,j)\, h_m\, h_n\right)$$

# Parallelization of the Poisson code

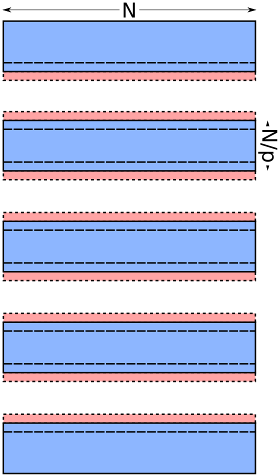



- Parallelize the Poisson 2D problem using the Messages Passing Interface (MPI)

# Parallelization of the Poisson code

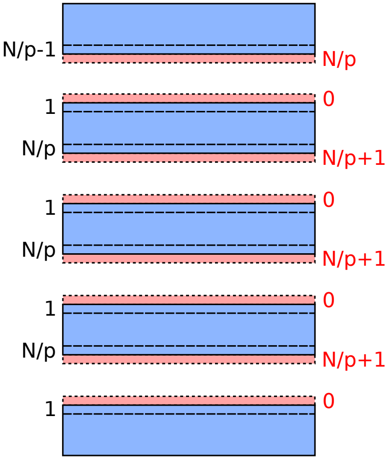



- The memory allocation is done in the C default manner, "Row-Major Order": make your domain decomposition by lines



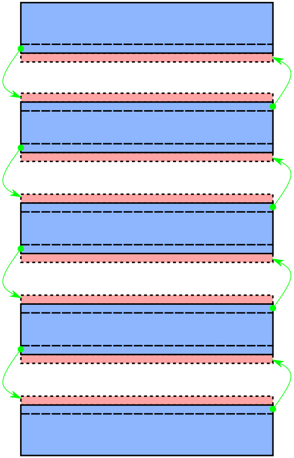

- $p$ domains of size $N/p$ each (1 per process)

# Parallelization of the Poisson code



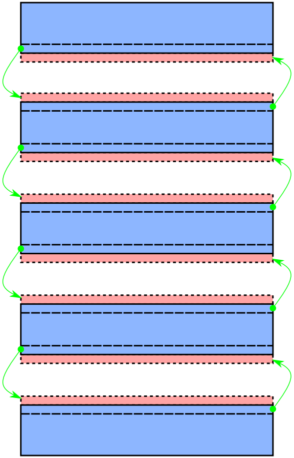

- Adding *ghost* lines before and after

# Parallelization of the Poisson code



- Use the *ghost* lines to receive the missing local data

# Parallelization of the Poisson code



- Start using `MPI_Sendrecv` to implement the communications
- You can use the number of iterations as a check
- Once it is working try to use *non-blocking* communications