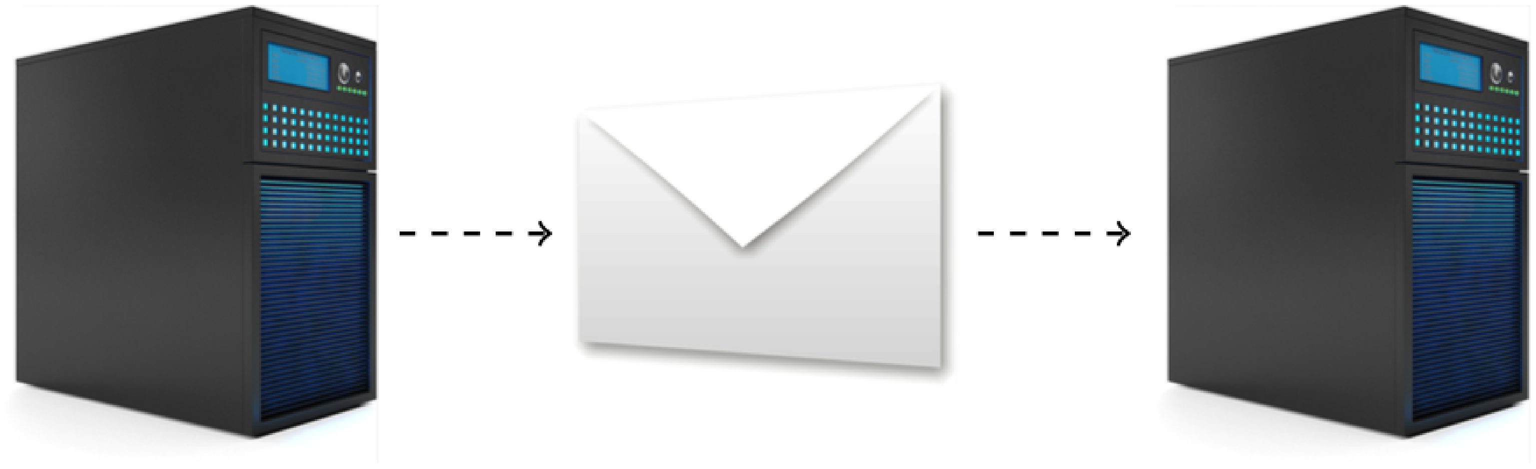
High Performance Python Lab

Term 2 2020/2021

Lecture 5. Message Passing Interface (mpi4py) continued, new tasks, crashcourse on using supercomputer

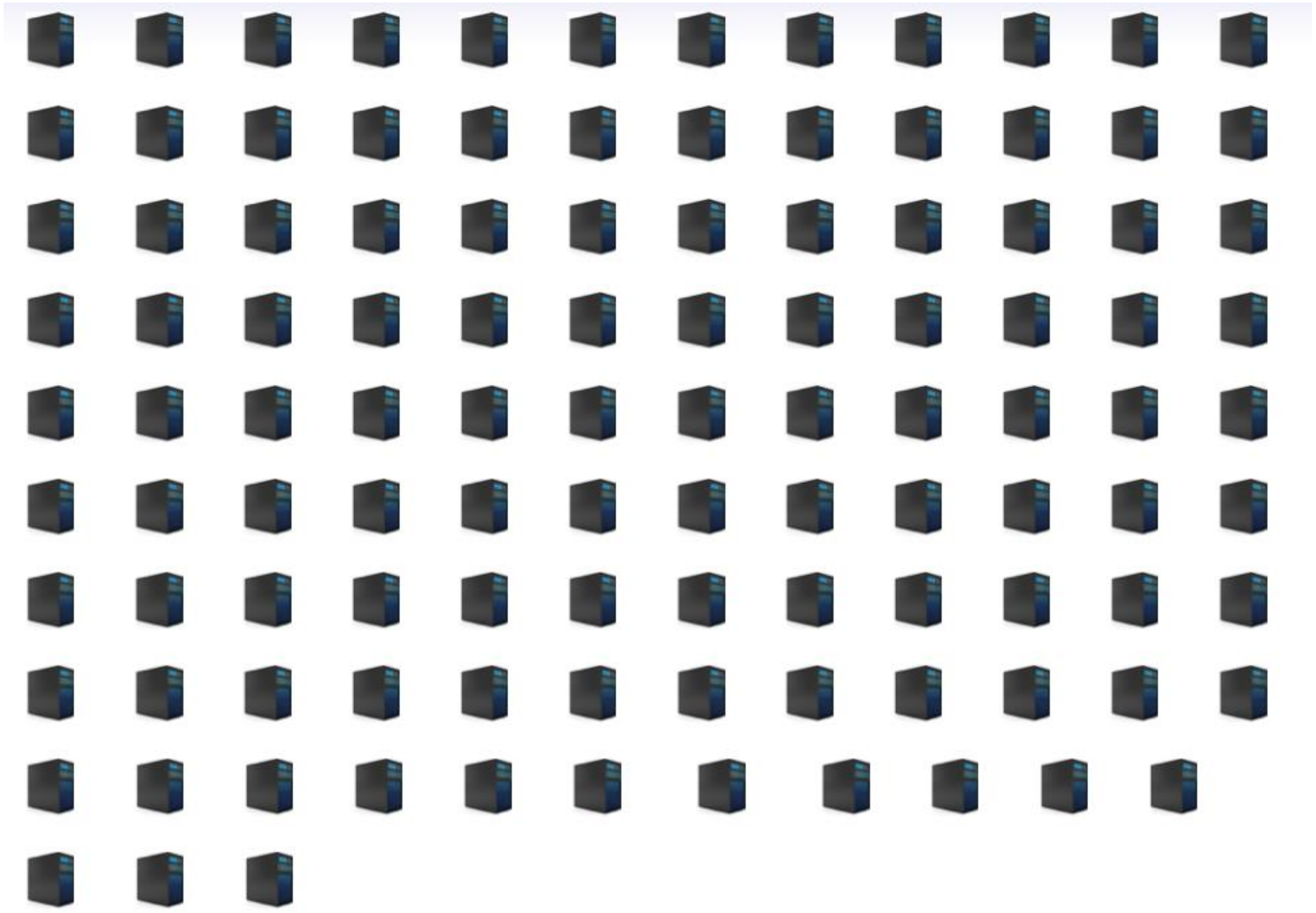
MPI

Message Passing Interface:



http://mpitutorial.com/

MPI



MPI for python

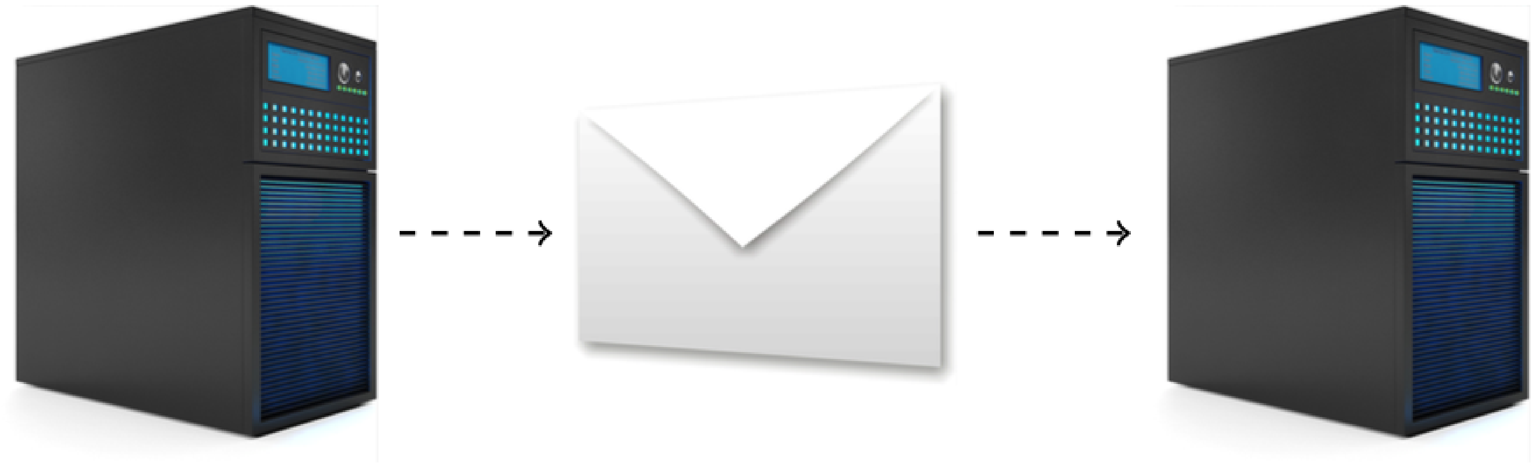
```
conda install mpi4py
```

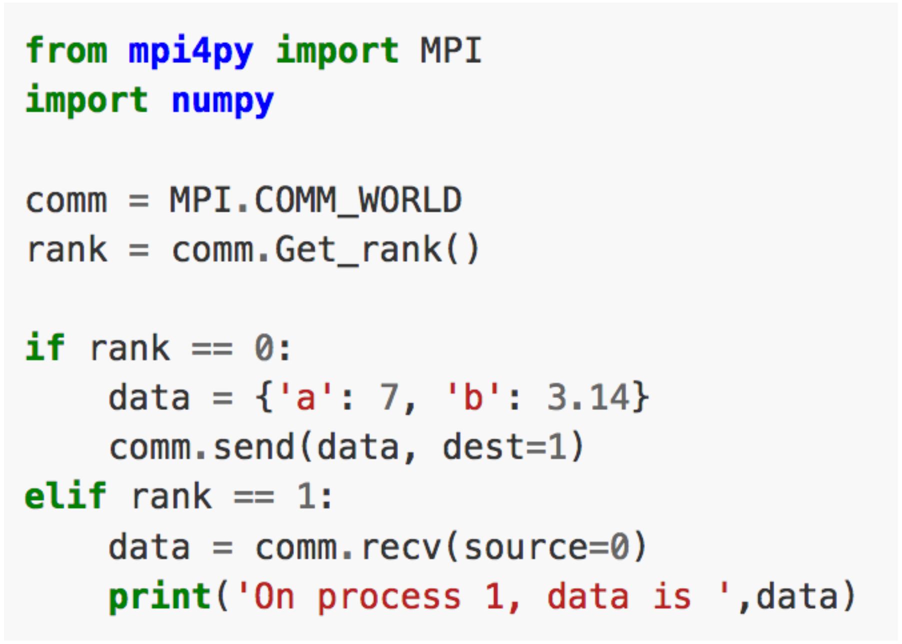
```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print('My rank is ',rank)
```

```
mpirun -n 4 python comm.py
```

MPI. Point-to-point communication

Message Passing Interface:



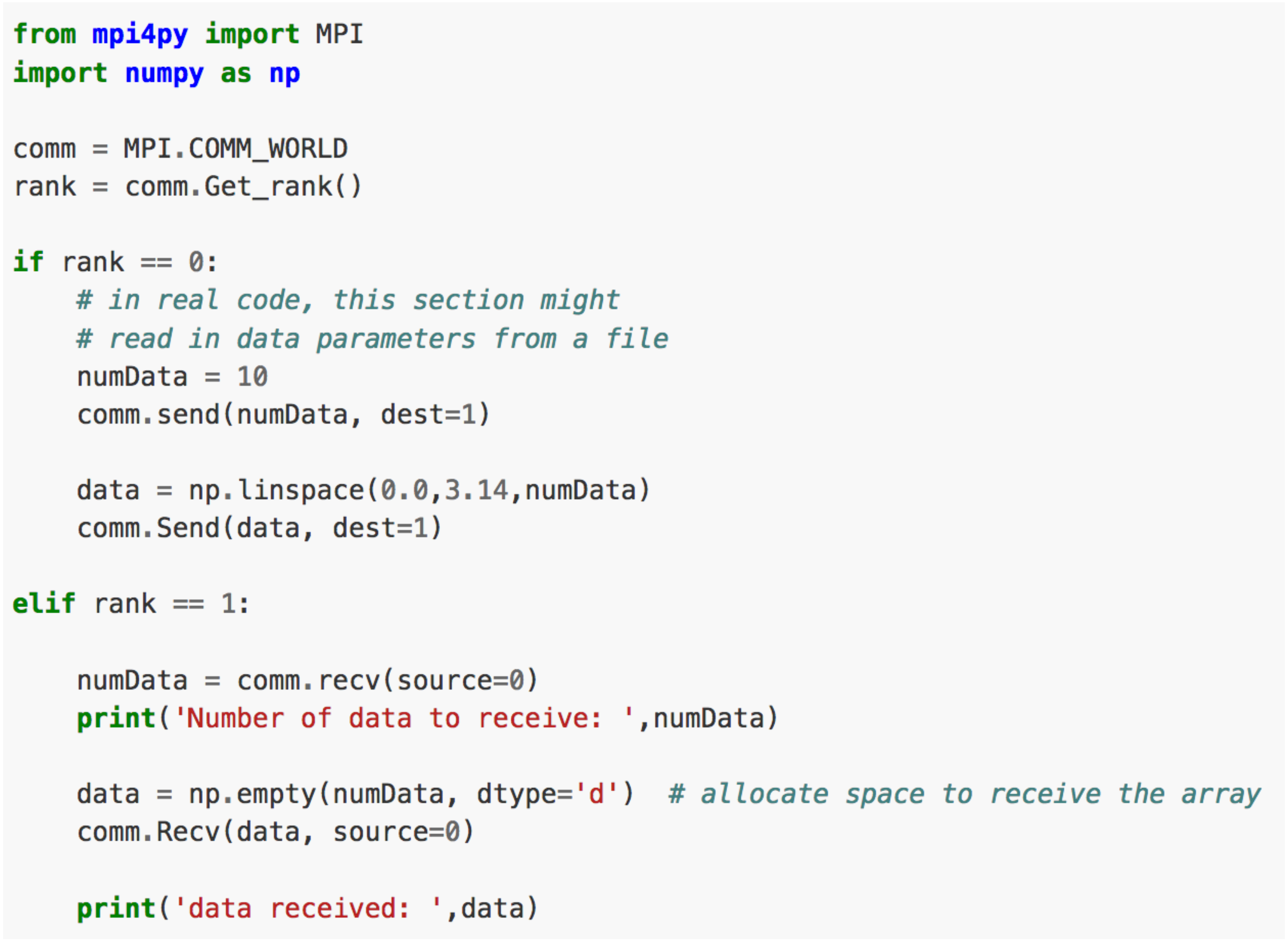


```
from mpi4py import MPI
import numpy

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source=0)
    print('On process 1, data is ',data)
```

Important:

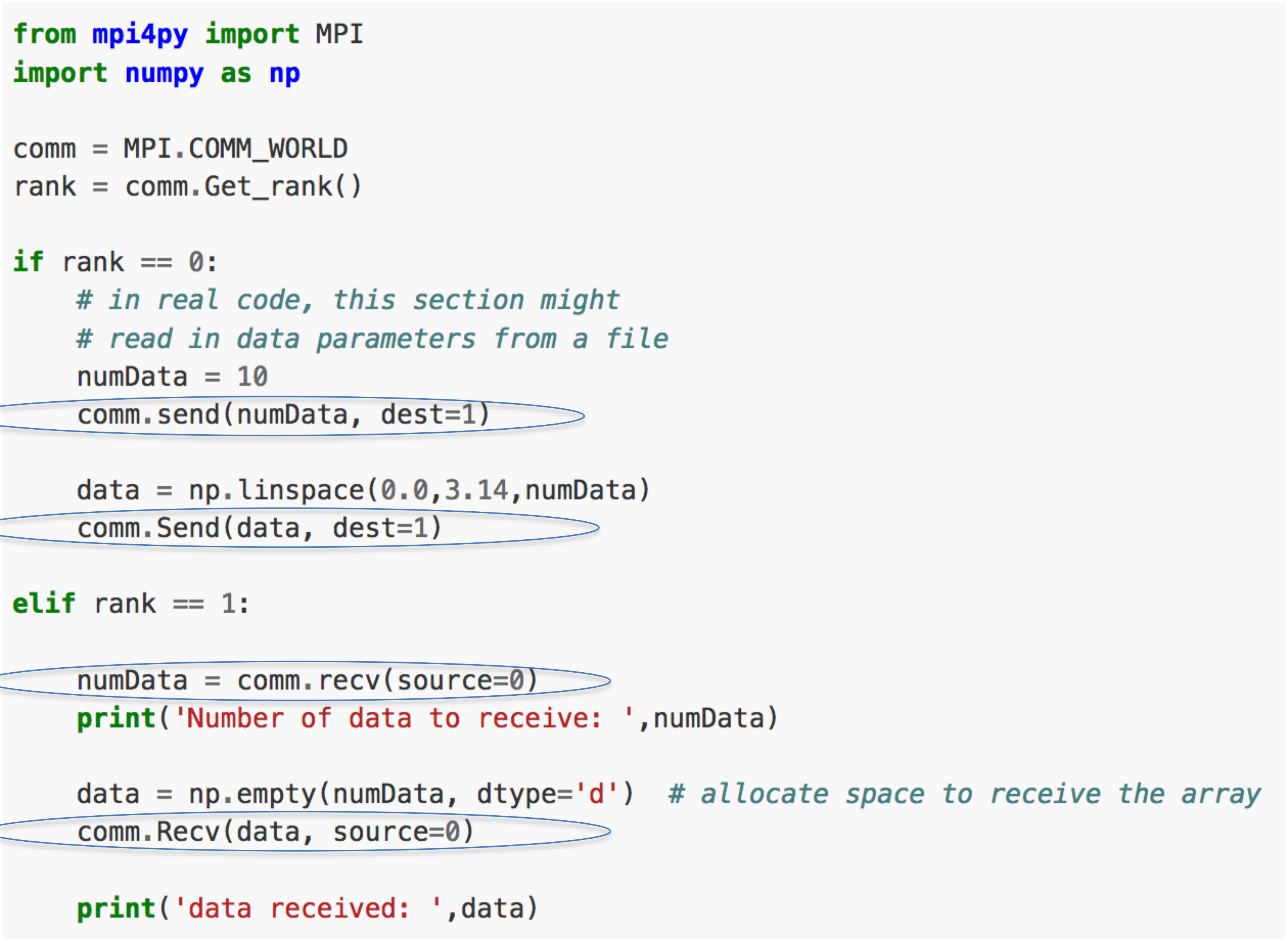
BLOCKING COMMUNICATION

MPI. Point-to-point communication



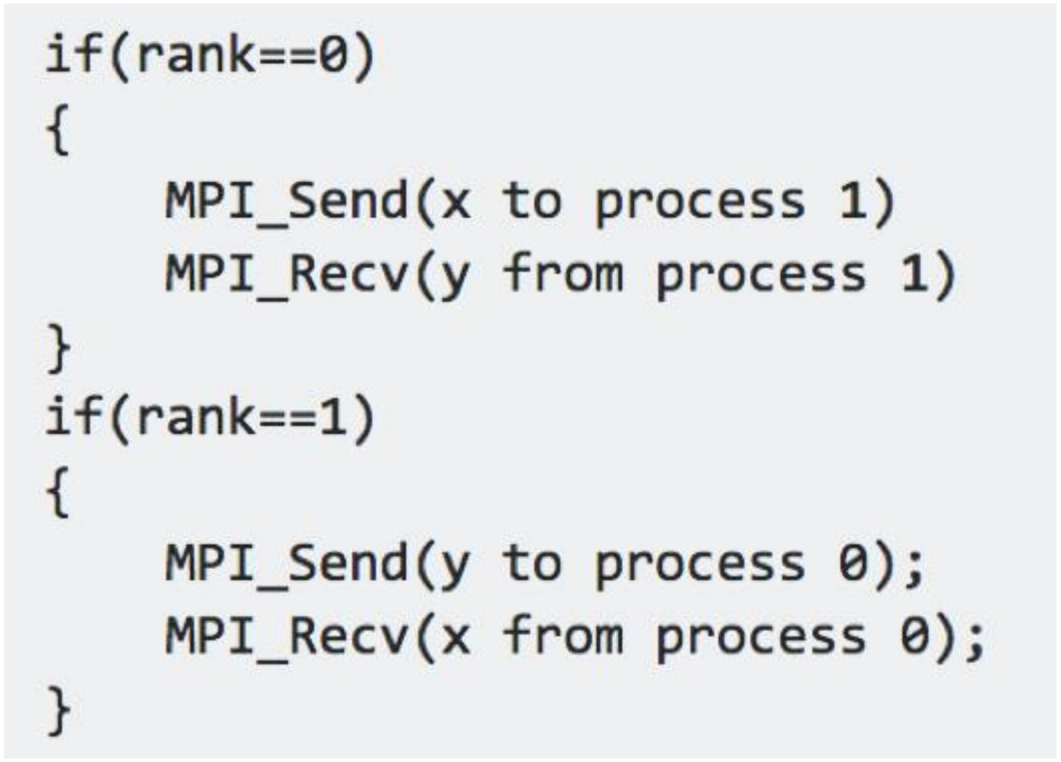
```python
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    # in real code, this section might
    # read in data parameters from a file
    numData = 10
    comm.send(numData, dest=1)

    data = np.linspace(0.0,3.14,numData)
    comm.Send(data, dest=1)

elif rank == 1:

    numData = comm.recv(source=0)
    print('Number of data to receive: ',numData)

    data = np.empty(numData, dtype='d')  # allocate space to receive the array
    comm.Recv(data, source=0)

    print('data received: ',data)
```

MPI. Point-to-point communication



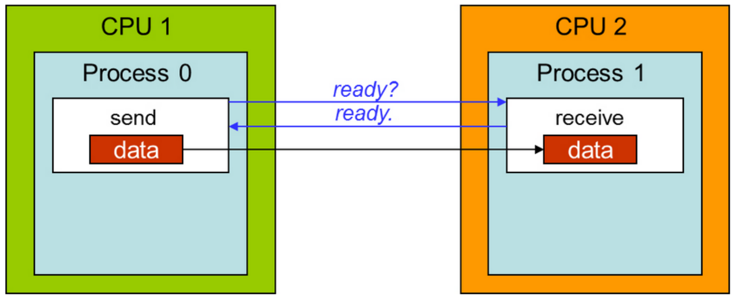
```python
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    # in real code, this section might
    # read in data parameters from a file
    numData = 10
    comm.send(numData, dest=1)

    data = np.linspace(0.0,3.14,numData)
    comm.Send(data, dest=1)

elif rank == 1:

    numData = comm.recv(source=0)
    print('Number of data to receive: ',numData)

    data = np.empty(numData, dtype='d')  # allocate space to receive the array
    comm.Recv(data, source=0)

    print('data received: ',data)
```

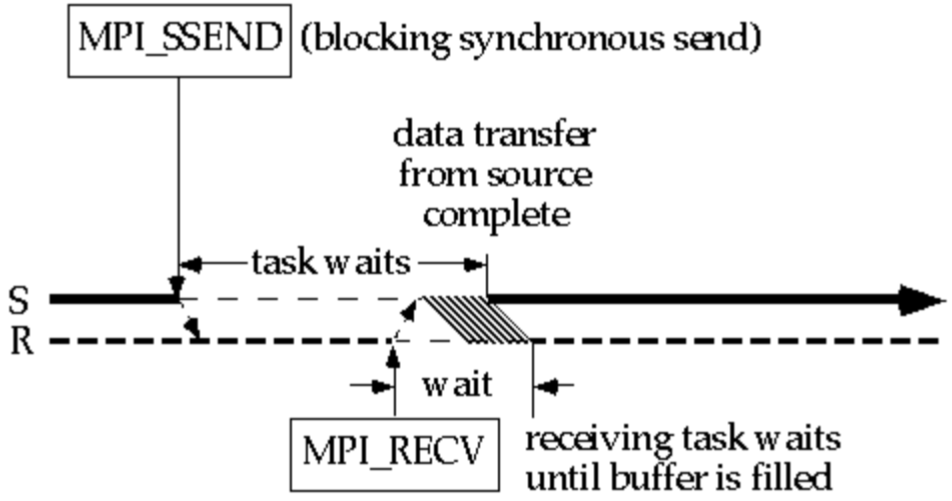
MPI_Send / MPI_Recv example



```
if(rank==0)
{
    MPI_Send(x to process 1)
    MPI_Recv(y from process 1)
}
if(rank==1)
{
    MPI_Send(y to process 0);
    MPI_Recv(x from process 0);
}
```

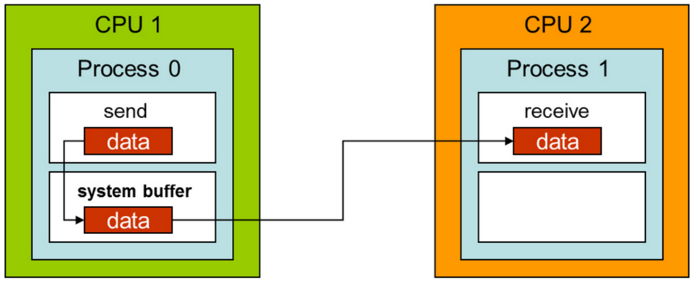
MPI Synchronous Send / Recv (SSend / Recv)

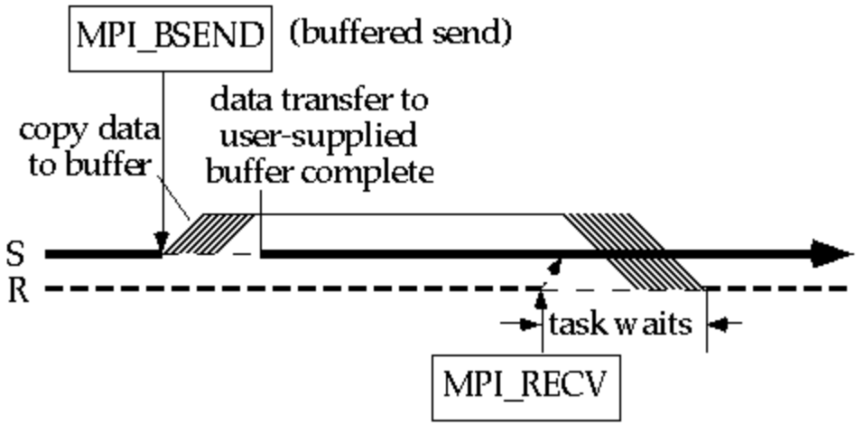



MPI_SSEND
(blocking synchronous send)
data transfer from source complete
task waits
S
R
wait
MPI_RECV
receiving task waits until buffer is filled

https://cvw.cac.cornell.edu/MPIP2P/

MPI Buffered Send / Recv (BSend / Recv)





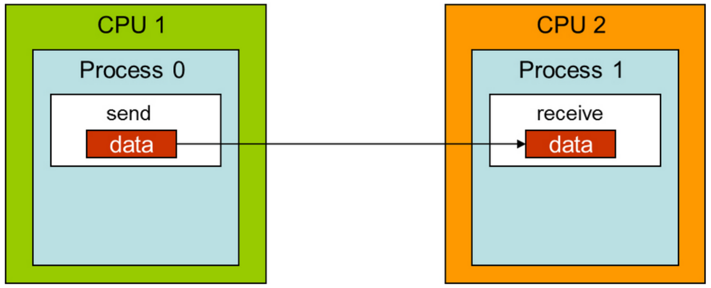
Pros:

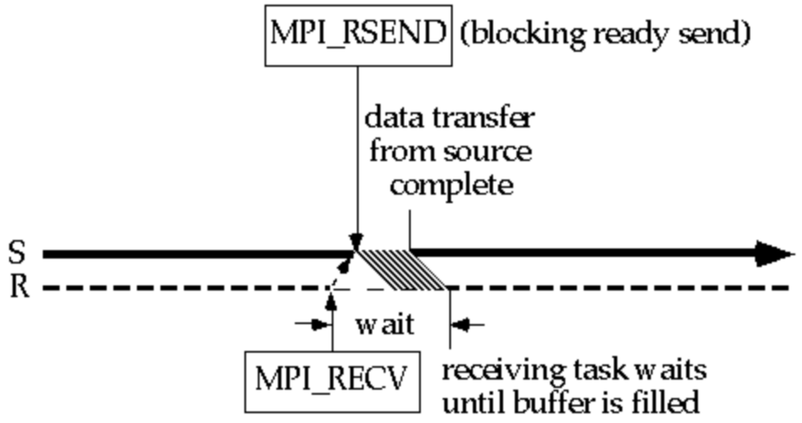
- No handshake – no need to wait for synchronization
- You can change your initial buffer
- Receiving can be done later

Cons: additional buffer

https://cvw.cac.cornell.edu/MPIP2P/

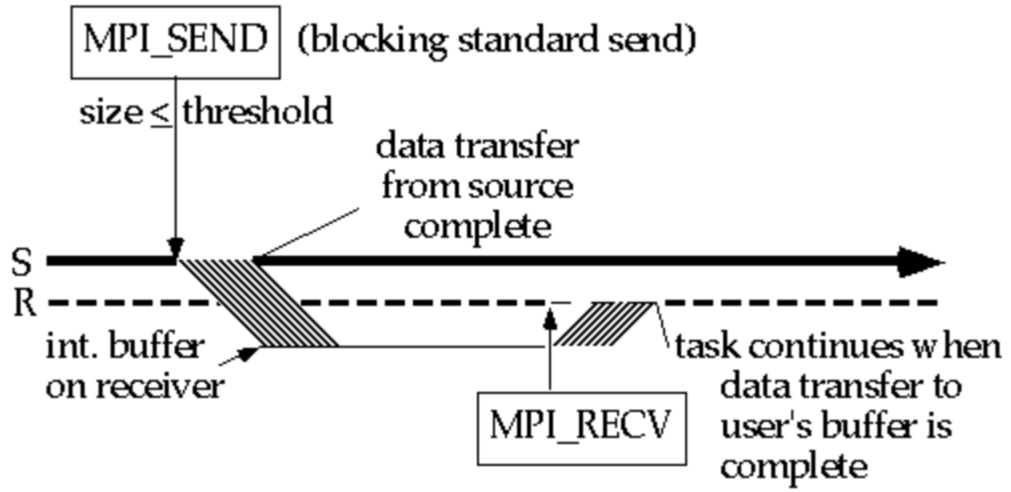
MPI Ready Send / Recv (RSend / Recv)




MPI_RSEND
(blocking ready send)
data transfer from source complete
S
R
wait
MPI_RECV
receiving task waits until buffer is filled

https://cvw.cac.cornell.edu/MPIP2P/

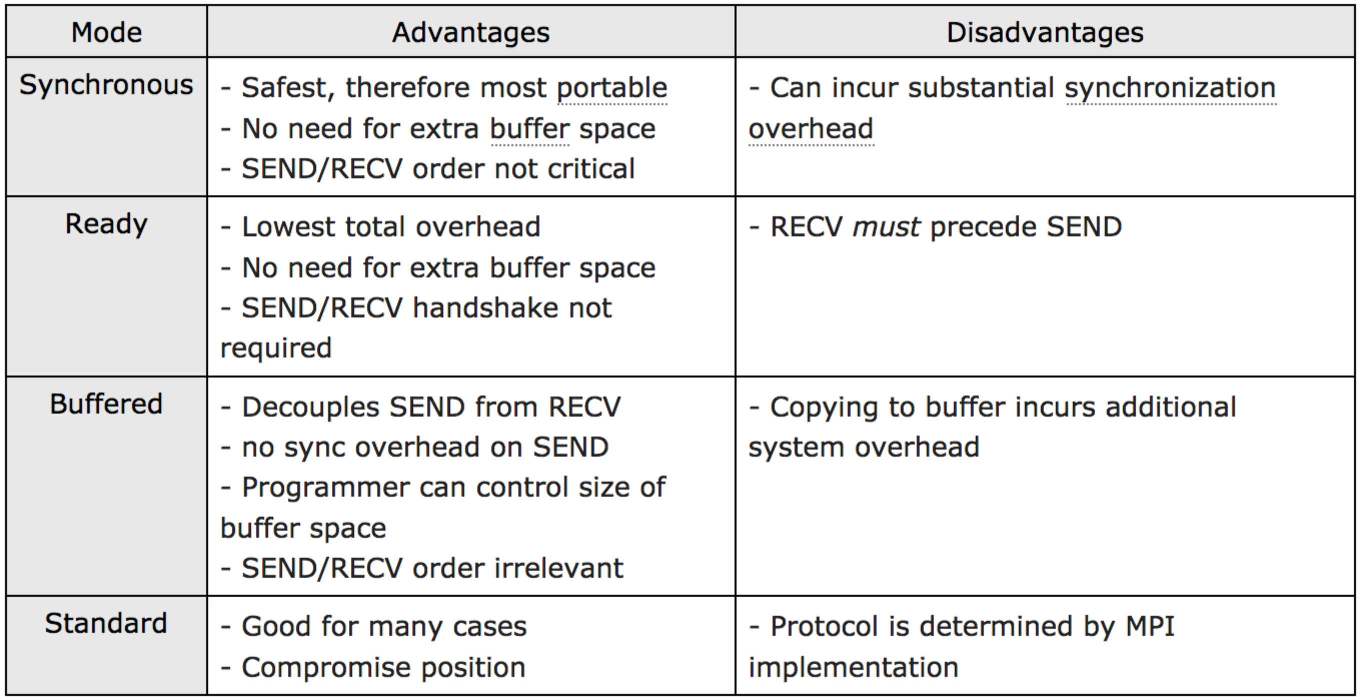
MPI Standard Send / Recv (Send / Recv)



MPI_SEND
(blocking standard send)
size ≤ threshold
data transfer from source complete
S
R
int. buffer on receiver
MPI_RECV
task continues when data transfer to user's buffer is complete

https://cvw.cac.cornell.edu/MPIP2P/

MPI Standard Send / Recv (Send / Recv)



Mode	Advantages	Disadvantages
Synchronous	- Safest, therefore most portable - No need for extra buffer space - SEND/RECV order not critical	- Can incur substantial synchronization overhead
Ready	- Lowest total overhead - No need for extra buffer space - SEND/RECV handshake not required	- RECV *must* precede SEND
Buffered	- Decouples SEND from RECV - no sync overhead on SEND - Programmer can control size of buffer space - SEND/RECV order irrelevant	- Copying to buffer incurs additional system overhead
Standard	- Good for many cases - Compromise position	- Protocol is determined by MPI implementation

https://cvw.cac.cornell.edu/MPIP2P/

MPI non-blocking P2P communication

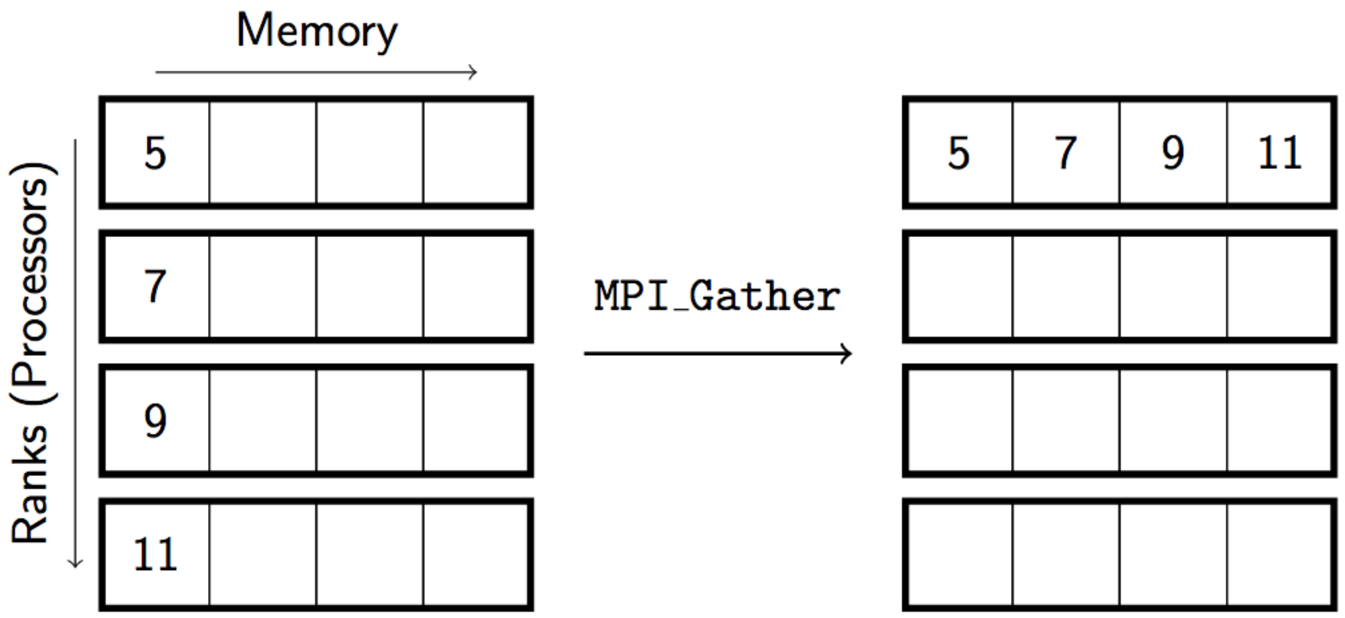
```
MPI_Status status;
MPI_Request request;

MPI_Isend(
    &count, 1, MPI_INT, dest, prank, MPI_COMM_WORLD, &request
);

MPI_Irecv(
    &count, 1, MPI_INT, source, source, MPI_COMM_WORLD, &request
);
```

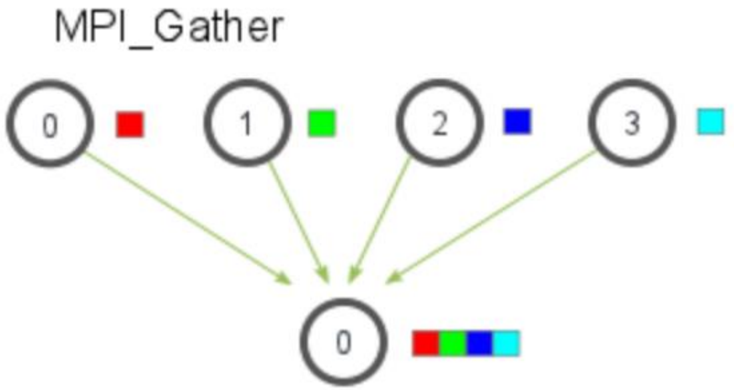
Testing whether the message has arrived:

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

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

MPI. Collective communications

Gather





MPI Gather example

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

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

numDataPerRank = 10
sendbuf = np.linspace(rank*numDataPerRank+1,(rank+1)*numDataPerRank,numDataPerRank)
print('Rank: ',rank, ', sendbuf: ',sendbuf)

recvbuf = None
if rank == 0:
    recvbuf = np.empty(numDataPerRank*size, dtype='d')

comm.Gather(sendbuf, recvbuf, root=0)

if rank == 0:
    print('Rank: ',rank, ', recvbuf received: ',recvbuf)
```

Examples and snippets on Canvas/box

- Try them out
- Understand them
- Use them

Computation on a mesh (grid)

For example, Laplace eqn:

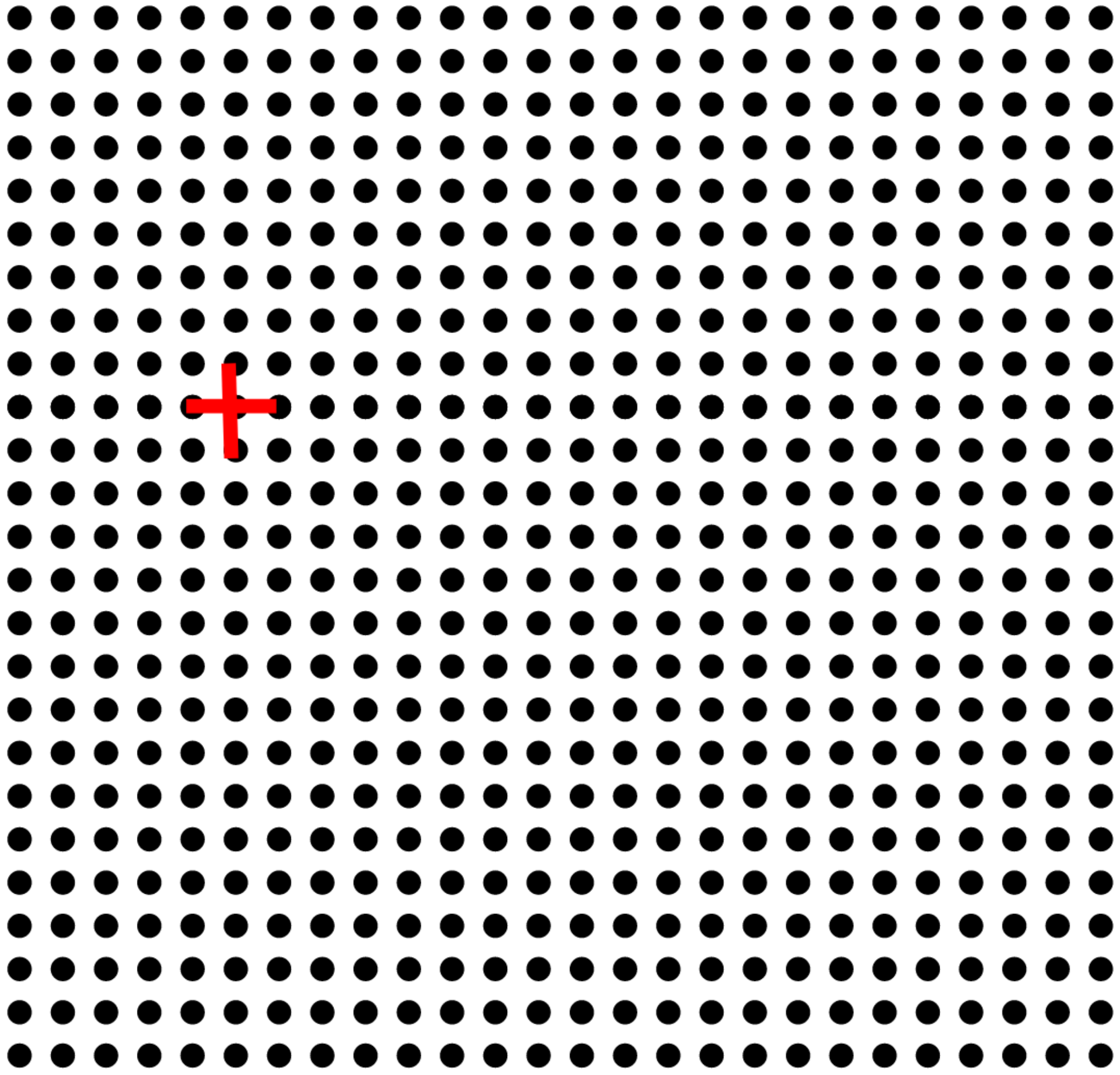
$$\nabla^2 f = 0$$

Or image blurring
Or cellular automata

Each circle is a grid point

The red “plus“ is called a numerical stencil

Need to communicate with neighbours on the grid



Computation on a mesh (grid)

For example, Laplace eqn:

$$\nabla^2 f = 0$$

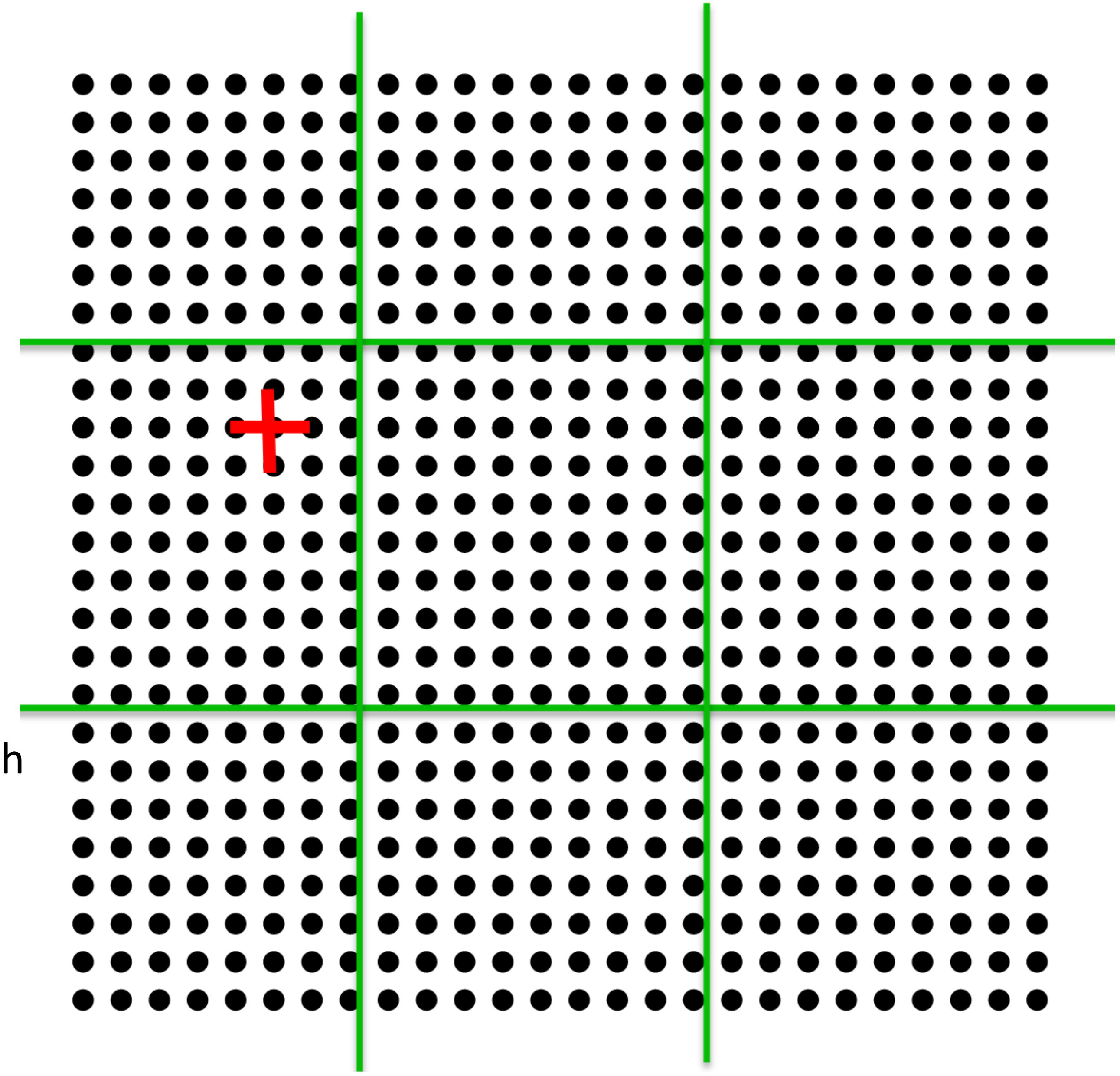
Or image blurring
Or cellular automata

Each circle is a grid point

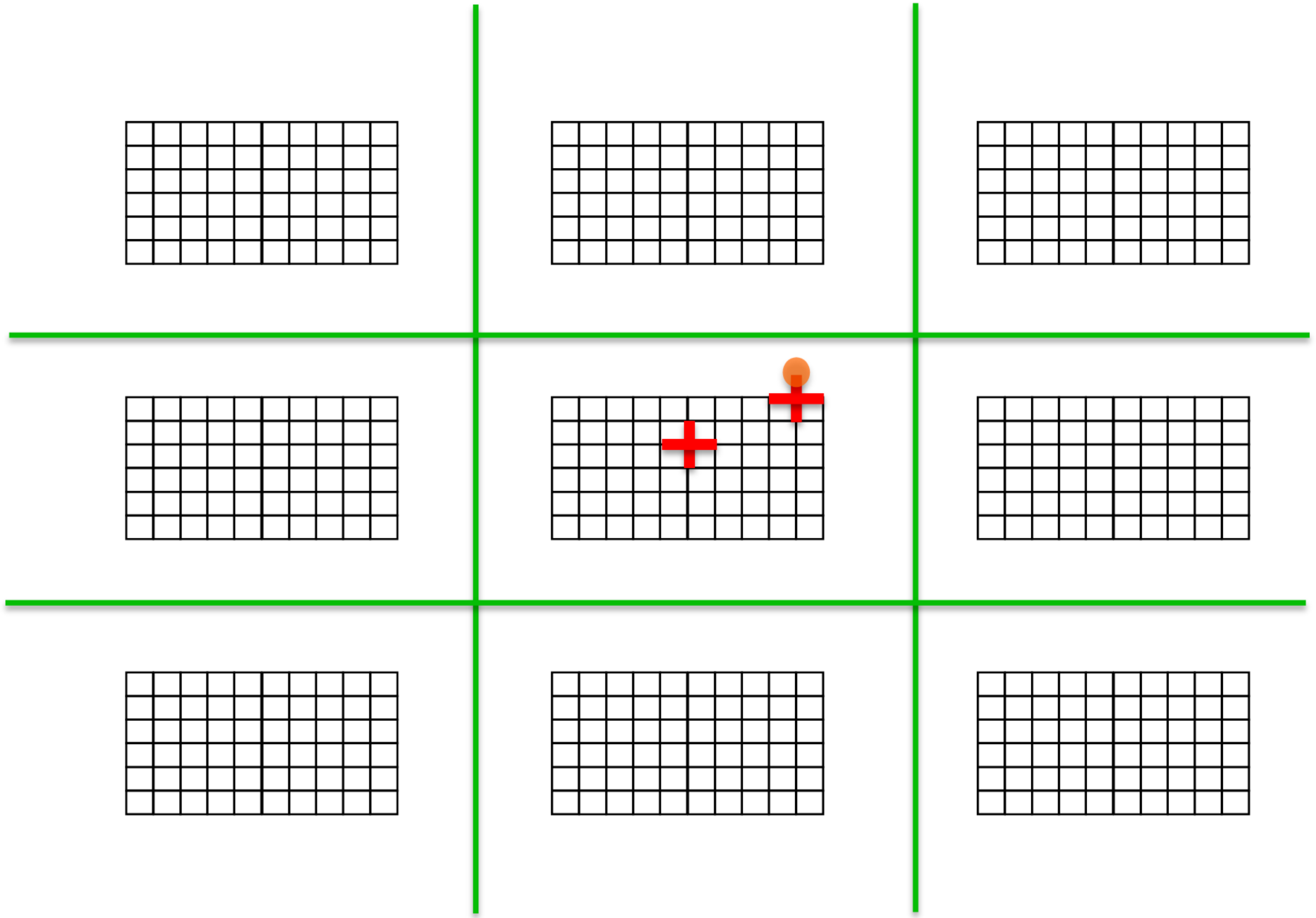
The red “plus“ is called a numerical stencil

Need to communicate with neighbours on the grid

Decompose the grid into (equal) chunks, each on a separate processor

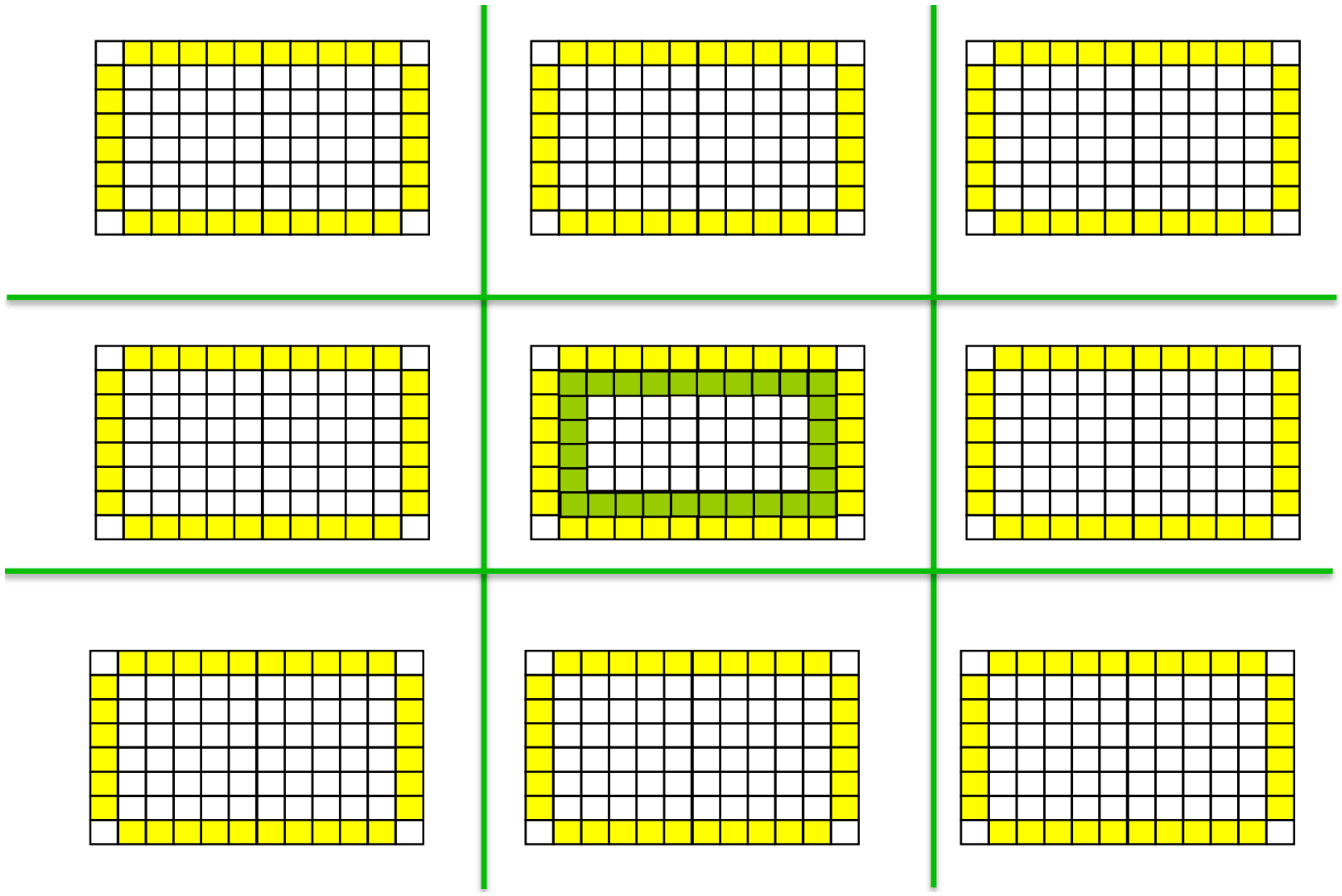


Necessary data transfers



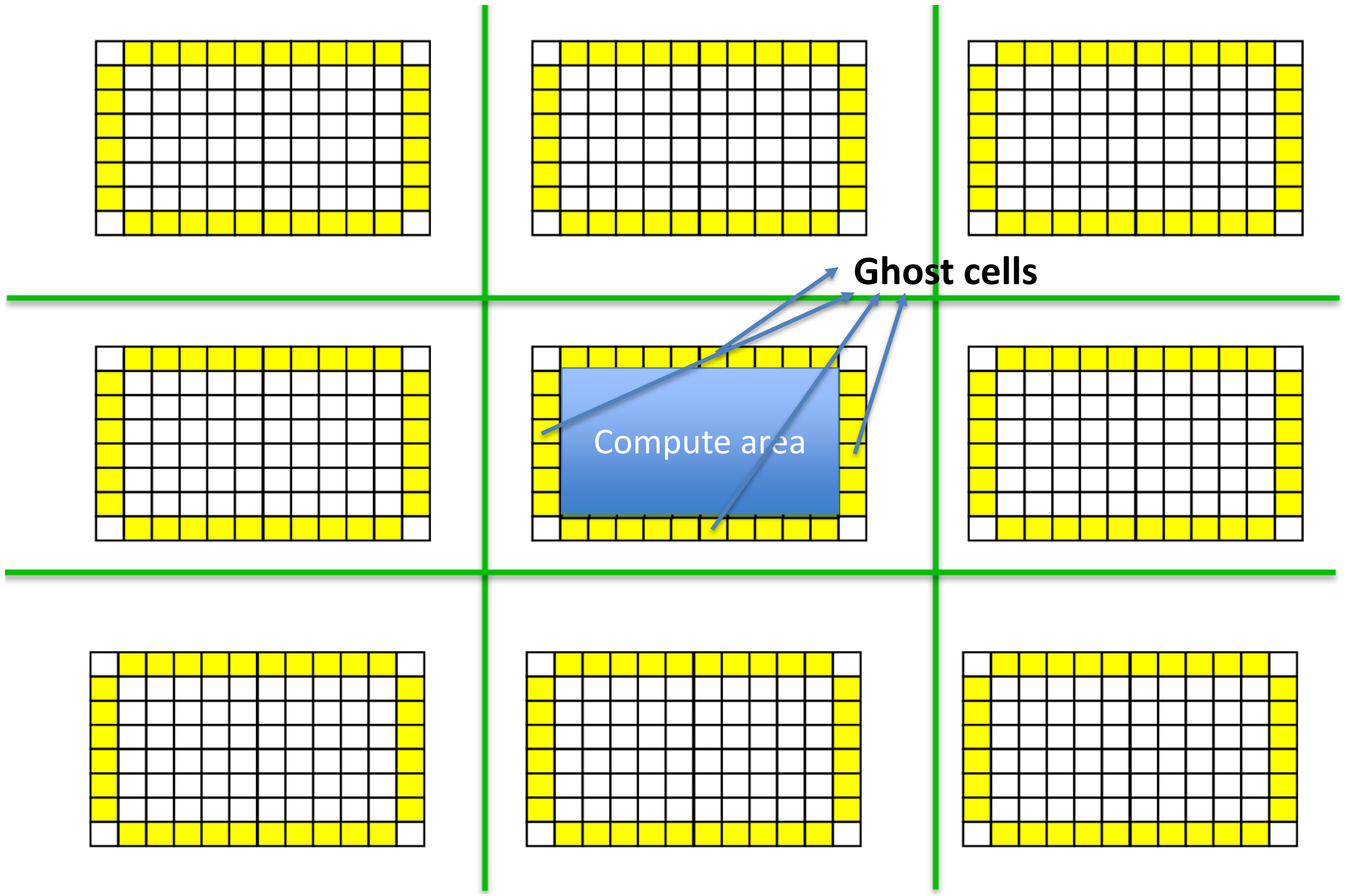
Necessary data transfers

Ghost (halo) cells exchange

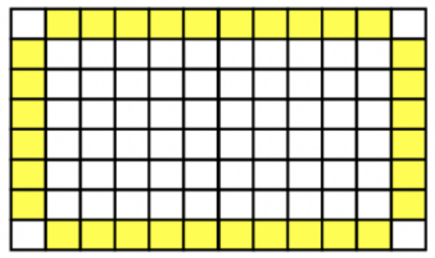


Necessary data transfers

Ghost (halo) cells exchange



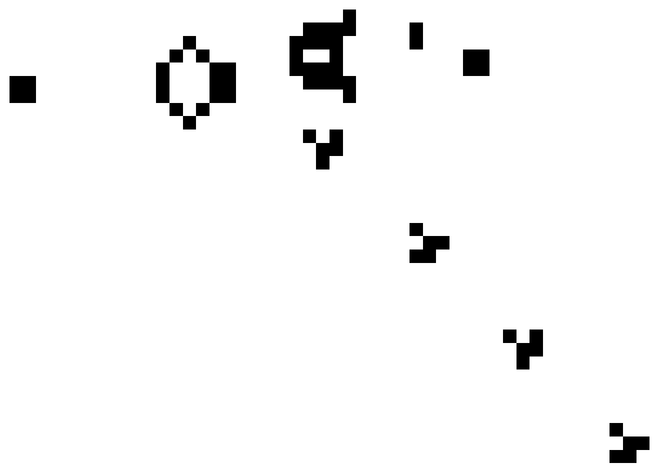
Algorithm



Time (convergence) loop ……

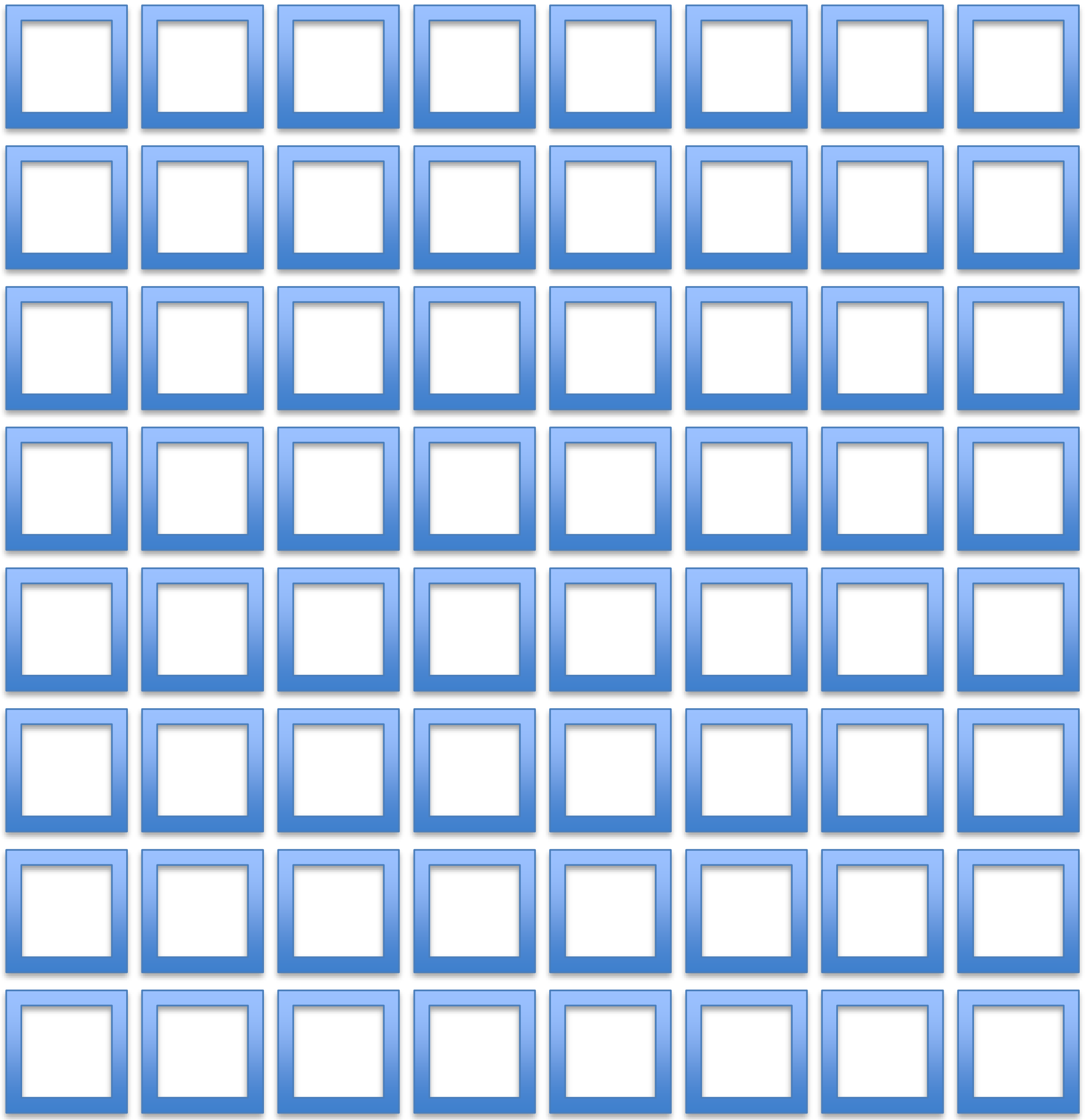
1. Exchange ghost cells
    1. MPI Send/Recv. Isend, Irecv etc
2. Perform computation as usual
    1. May use OpenMP, CUDA etc

Example: Game of Life

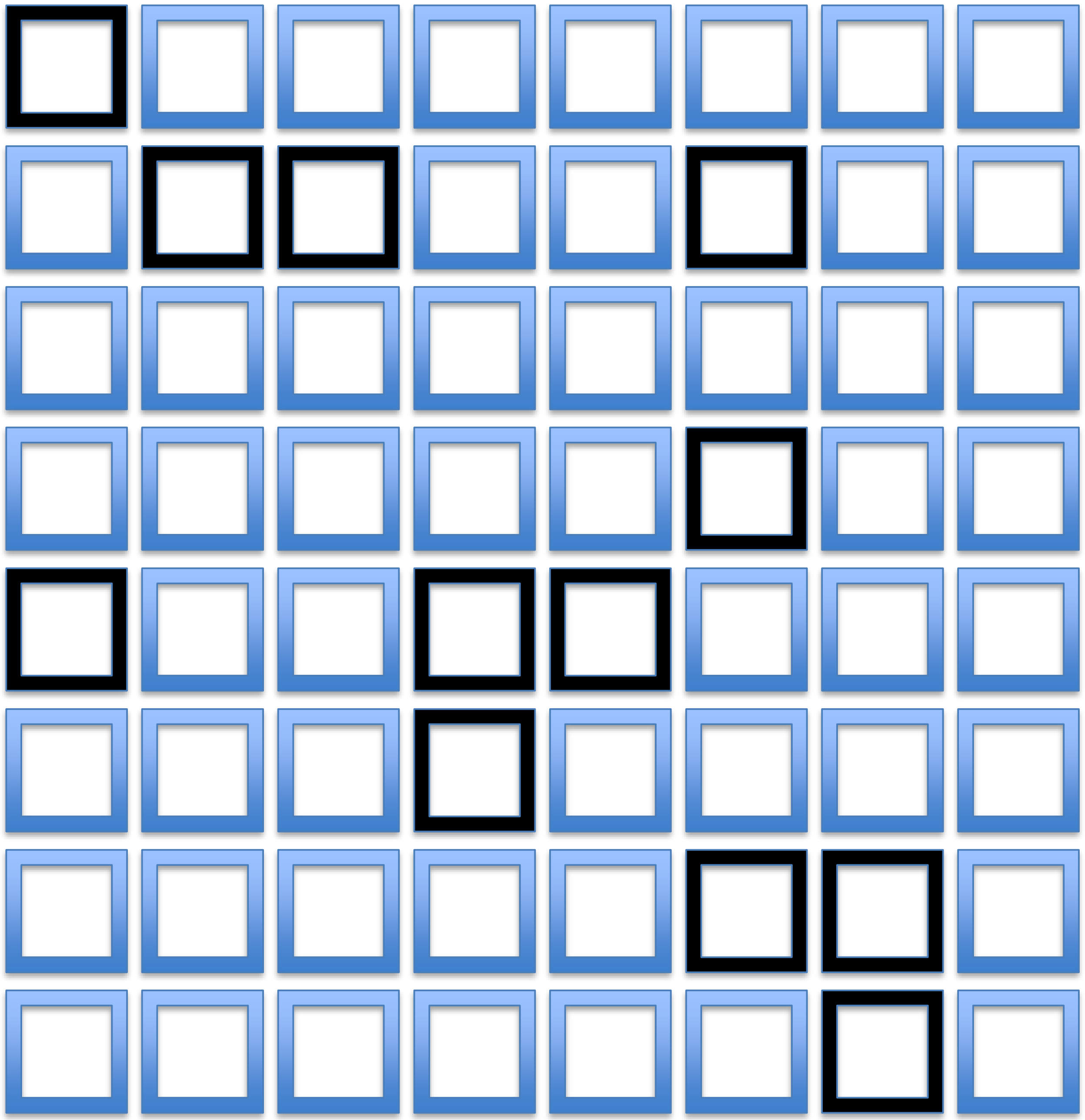


1. Any live cell with two or three live neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

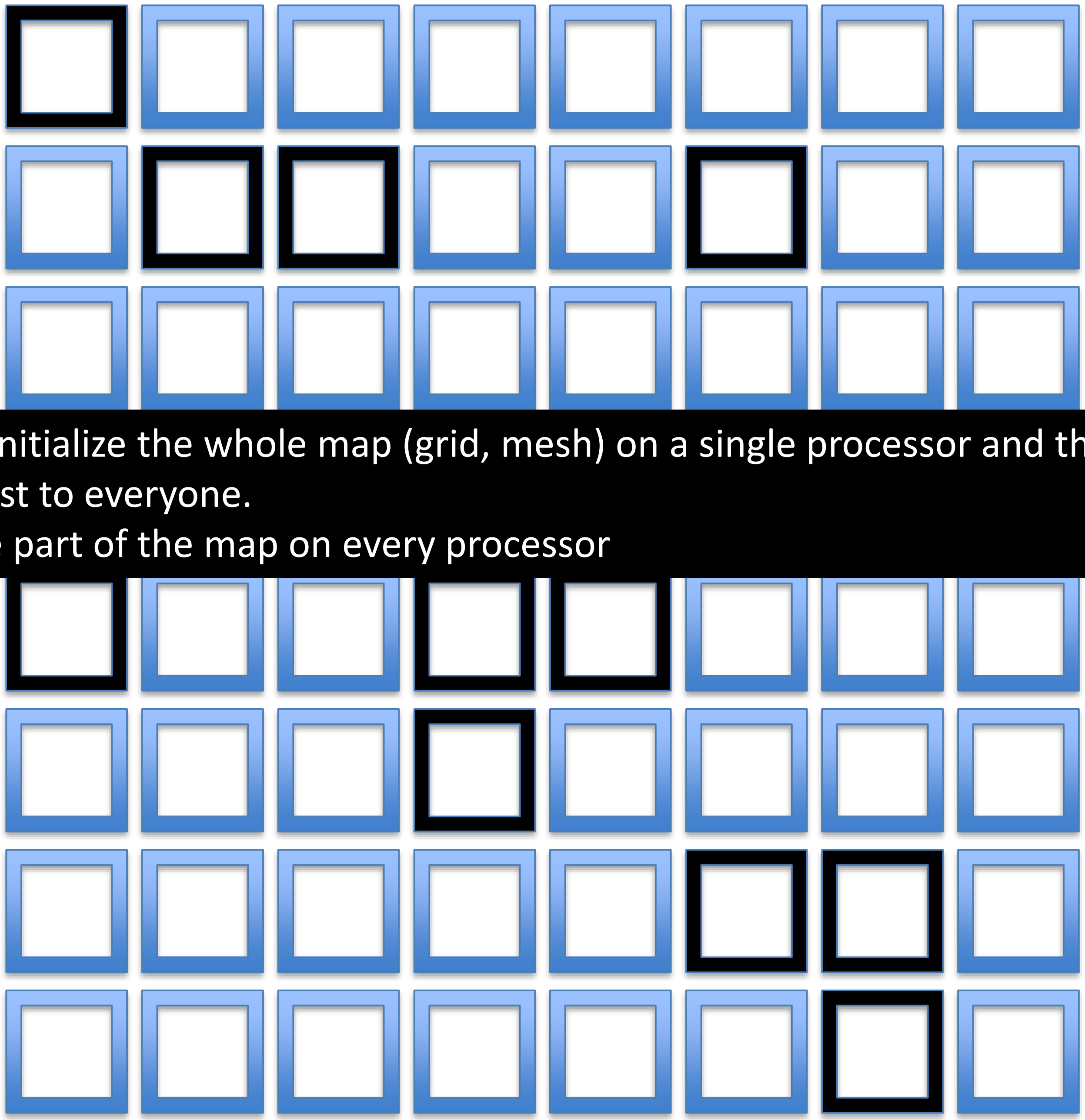
Game of Life 1: initialization (random)



Game of Life 1: initialization (random)

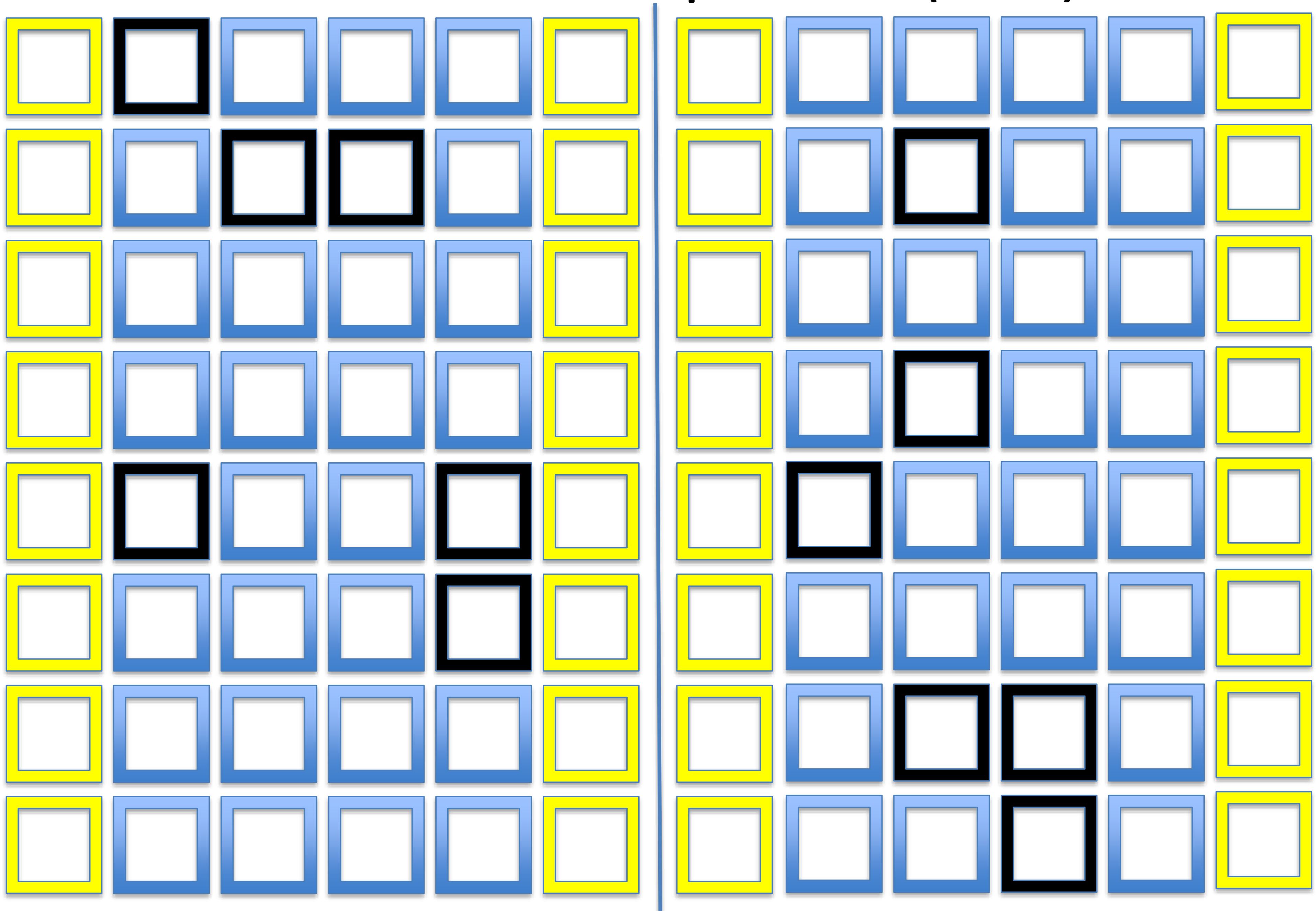


Initialization in parallel (MPI)

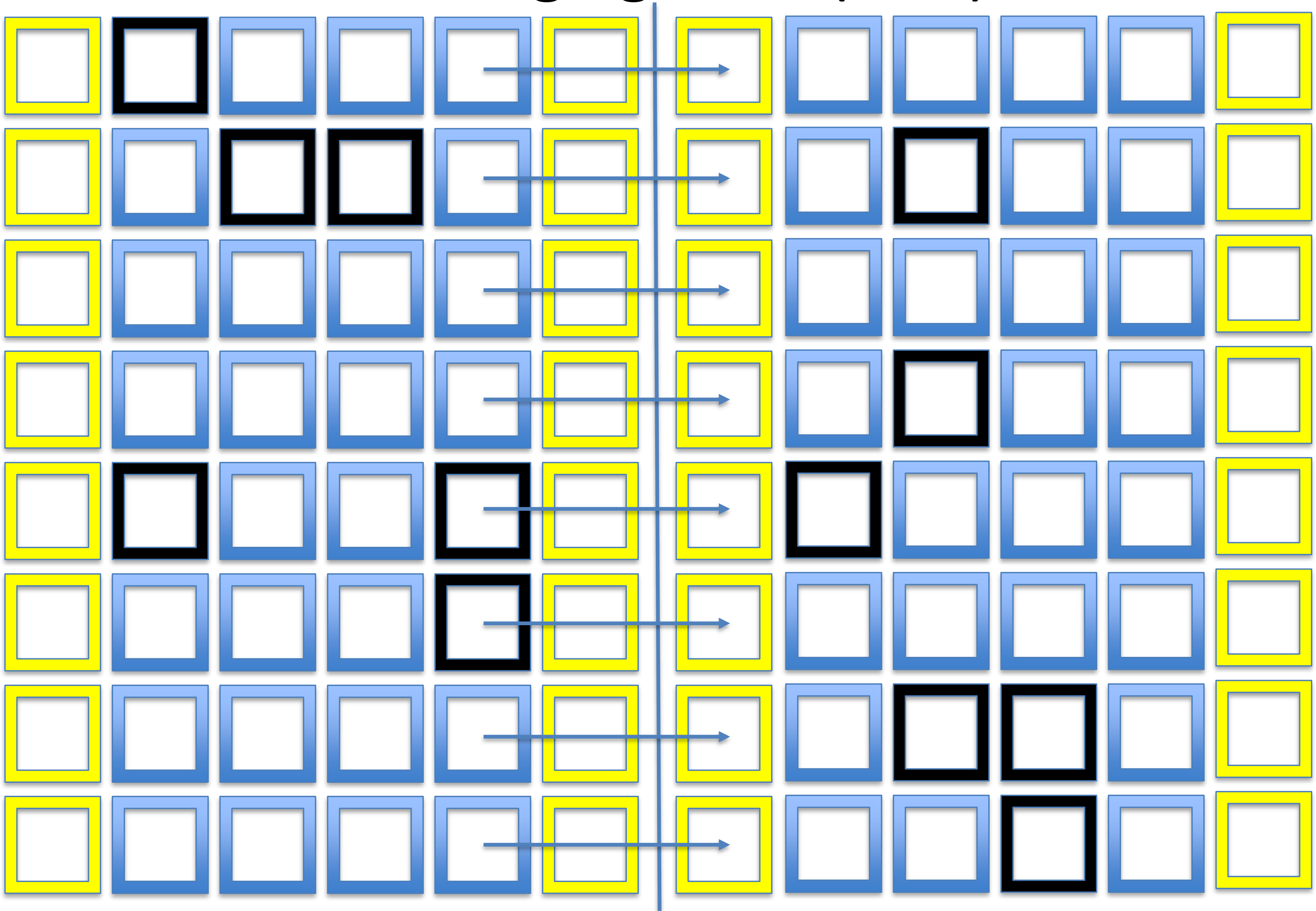


Do not initialize the whole map (grid, mesh) on a single processor and then broadcast to everyone.
Initialize part of the map on every processor

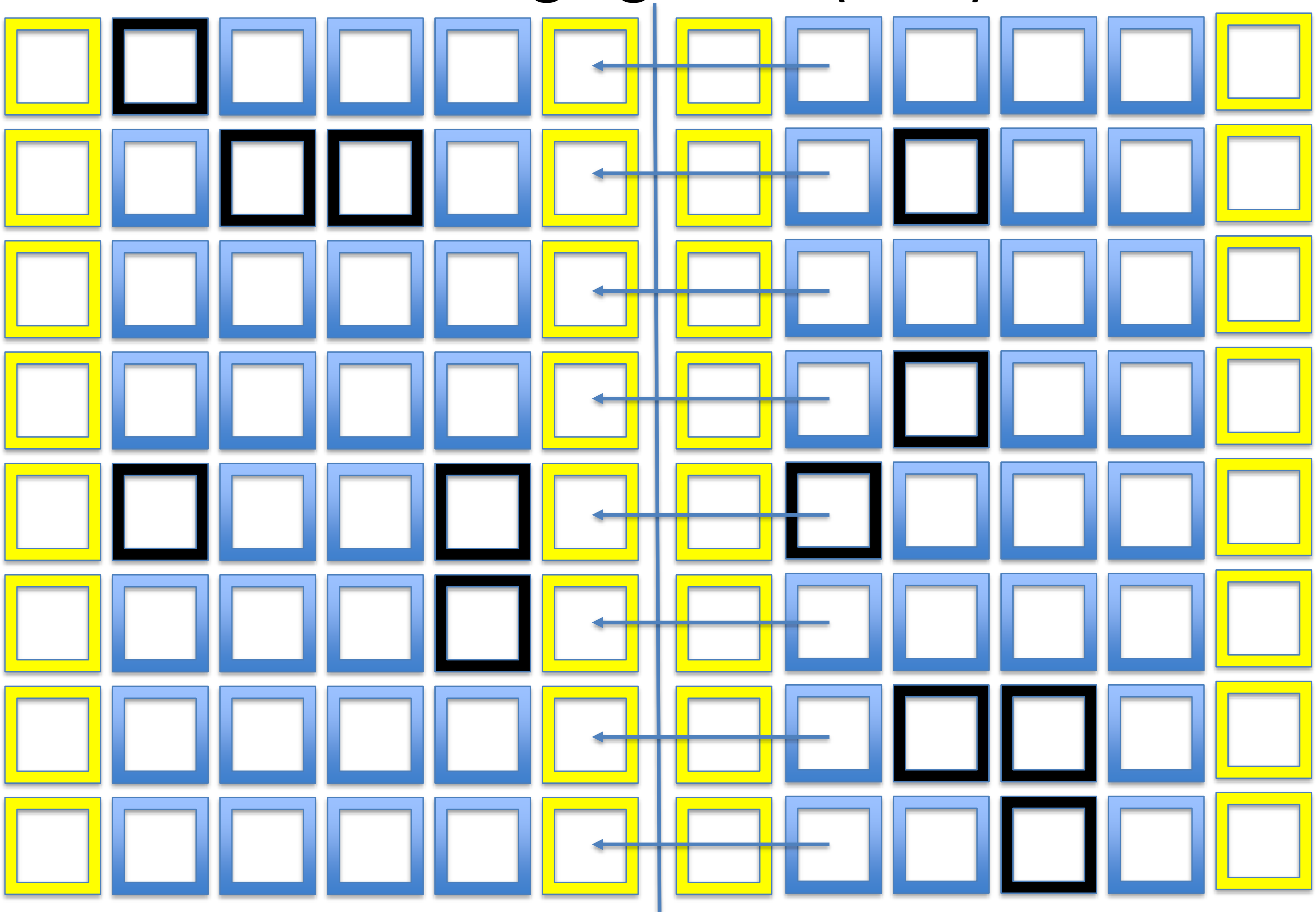
Initialization in parallel (MPI)



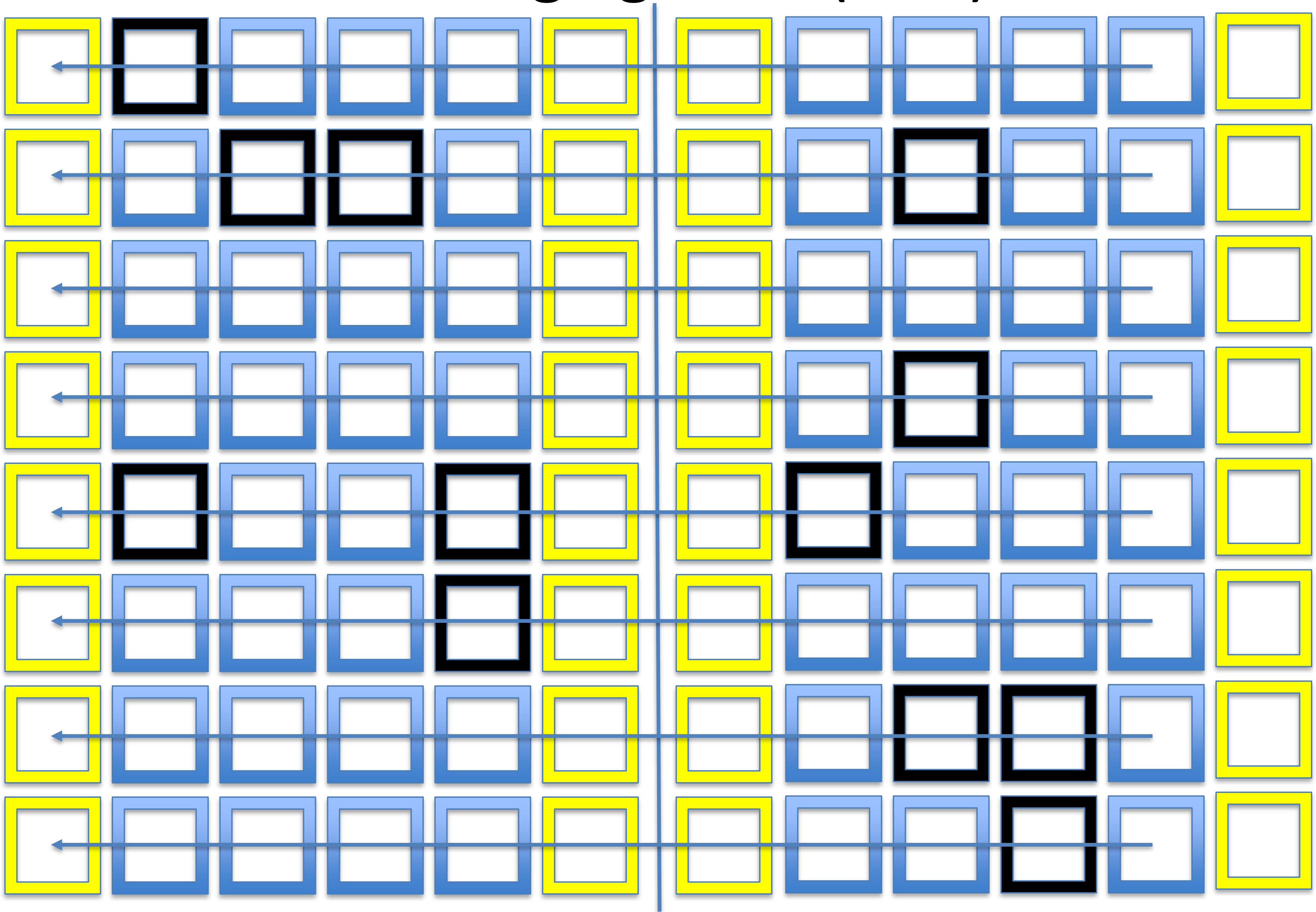
Exchange ghosts (MPI)



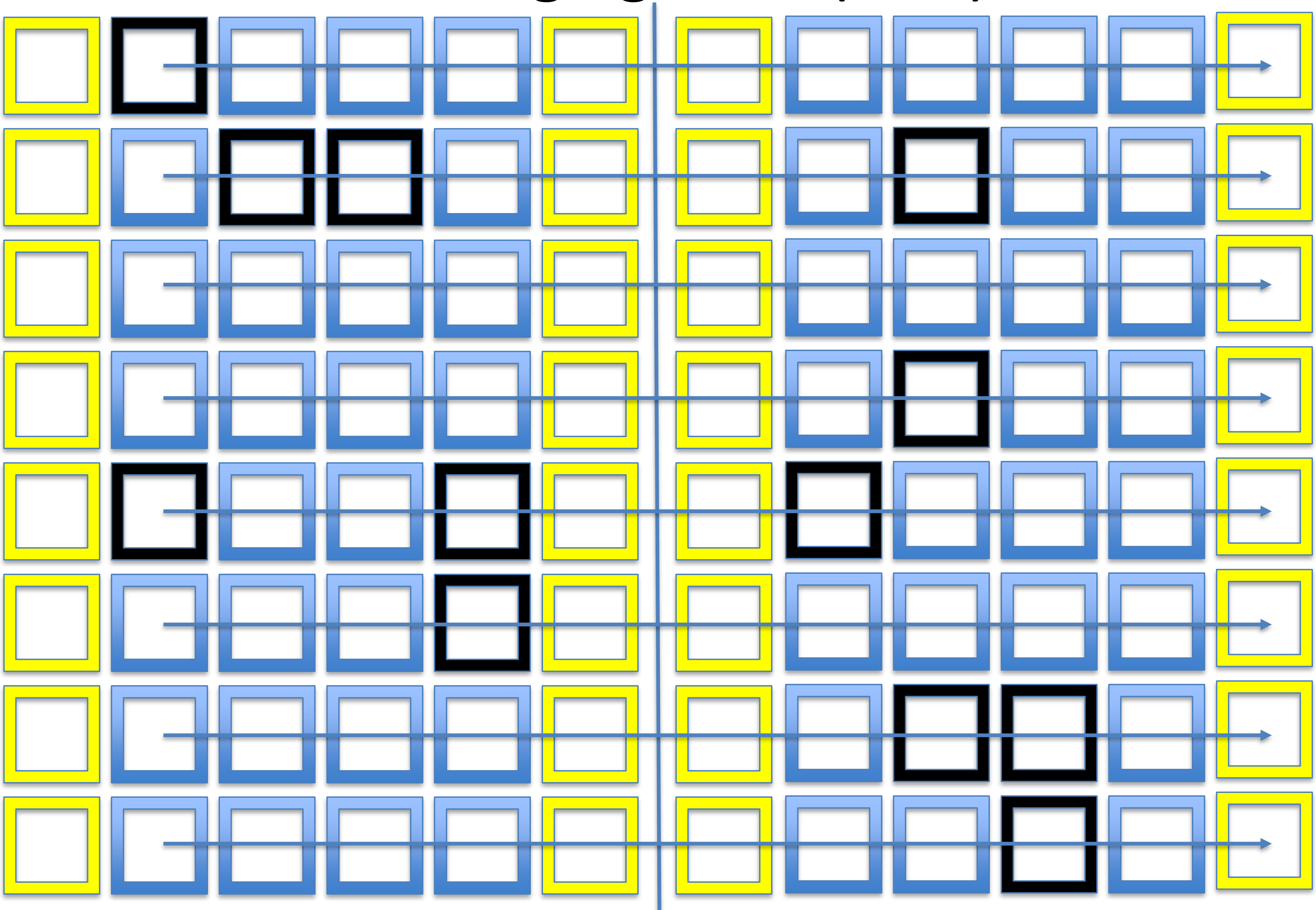
Exchange ghosts (MPI)



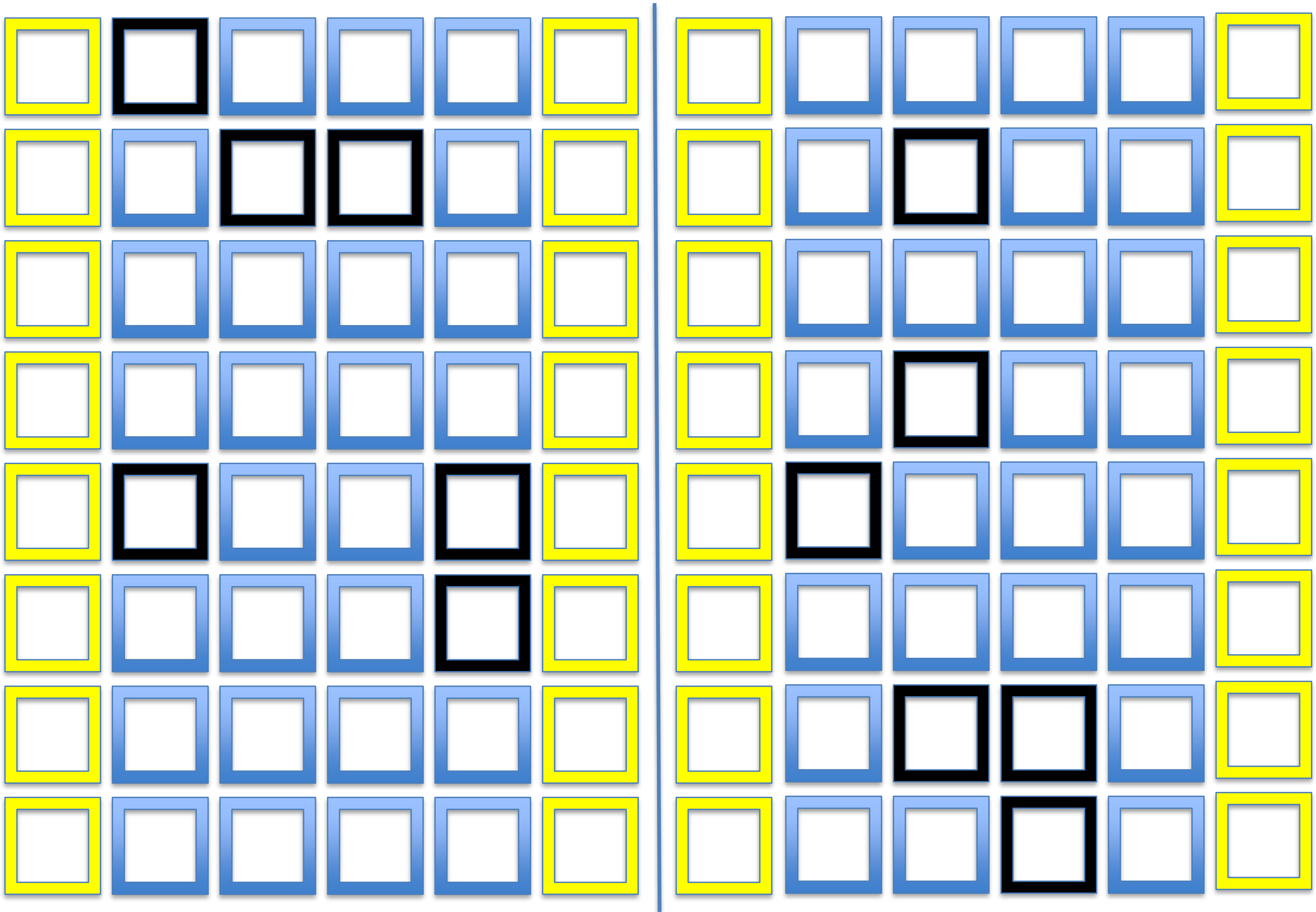
Exchange ghosts (MPI)



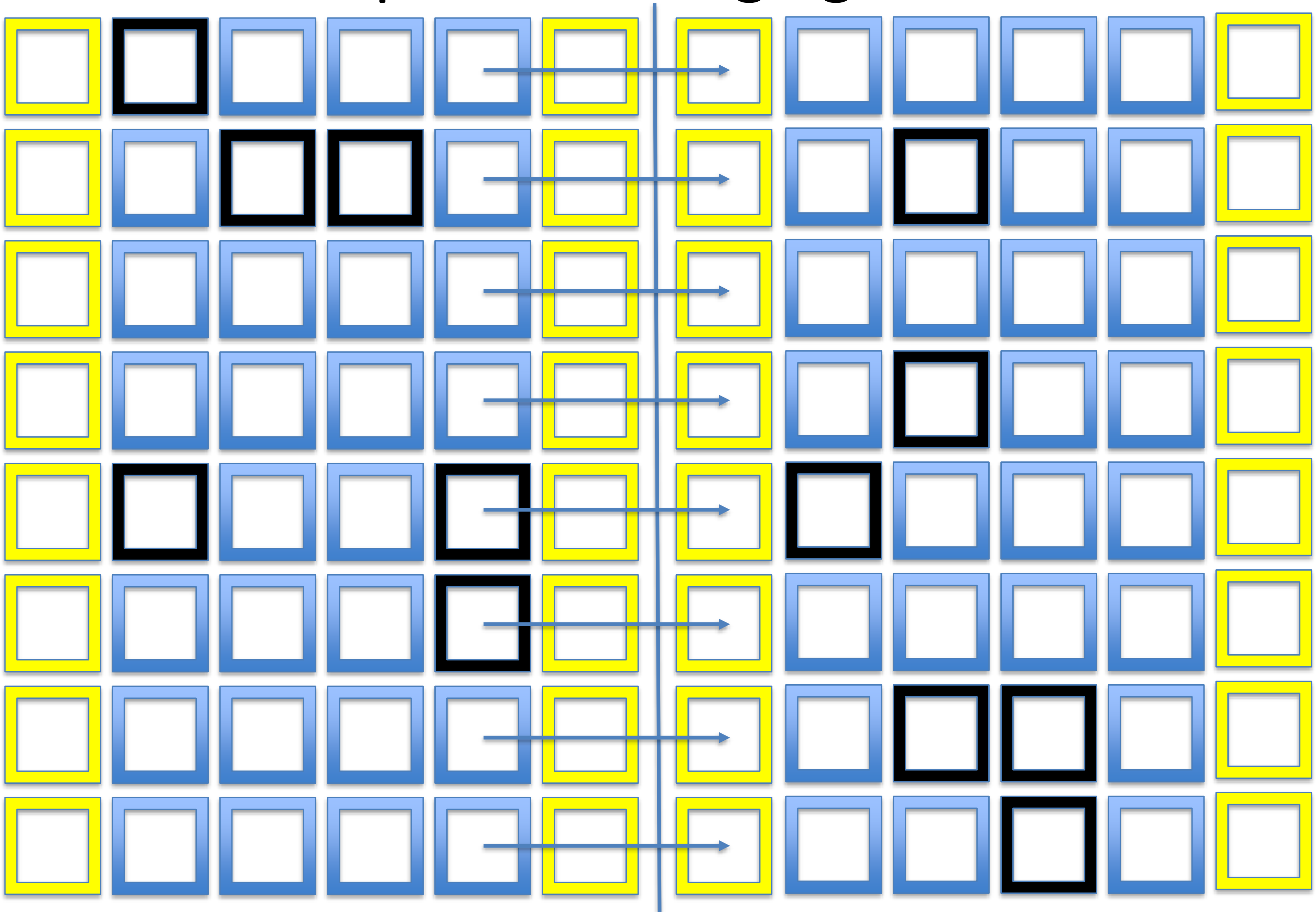
Exchange ghosts (MPI)



Do the calculation



Repeat exchange ghosts



How to save the grid

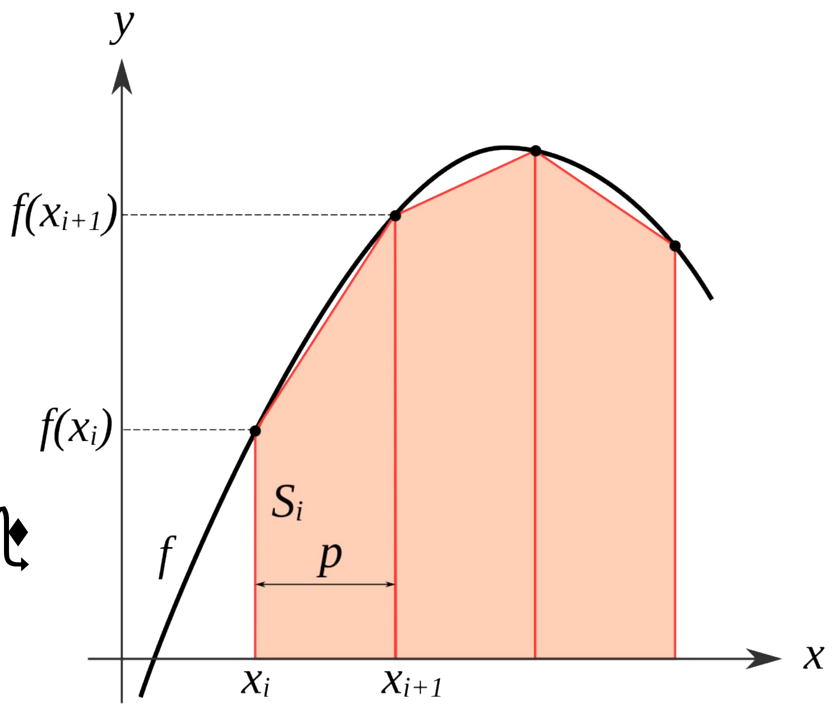
1. Each processor dumps its own part of the grid to a separate file: “grid-000.dat, grid-001.dat…”
2. Parallel I/O (for example, HDF5 library)
3. Do processing of data on the fly (OpenGL, other postprocessing modules)

Tasks

- Study an integral
- Columnwise shifted pictures
- Conway’s Game of Life

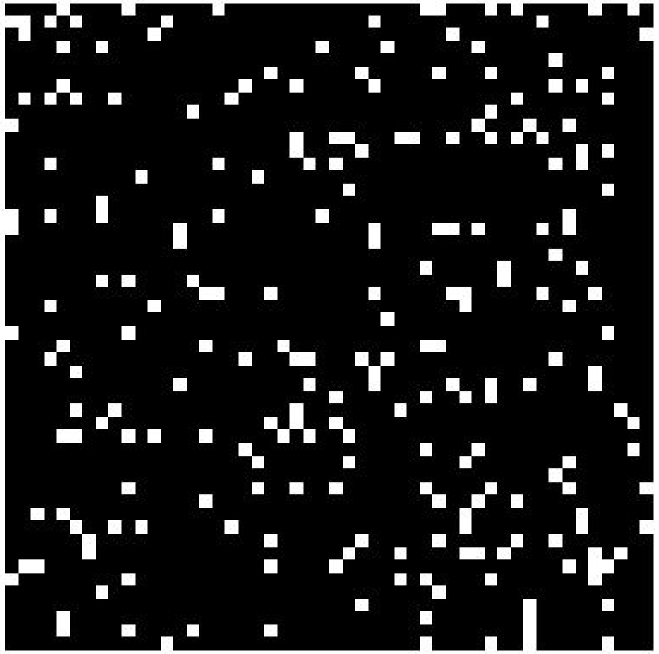
Task 7: Study an integral

- [illegible]
- [illegible] Reduce [illegible]
- [illegible]
- [illegible]



Task 8: Columnwise shifteded pictures

- Take a picture
- Split the picture columns between processes
- Shift the columns cyclically

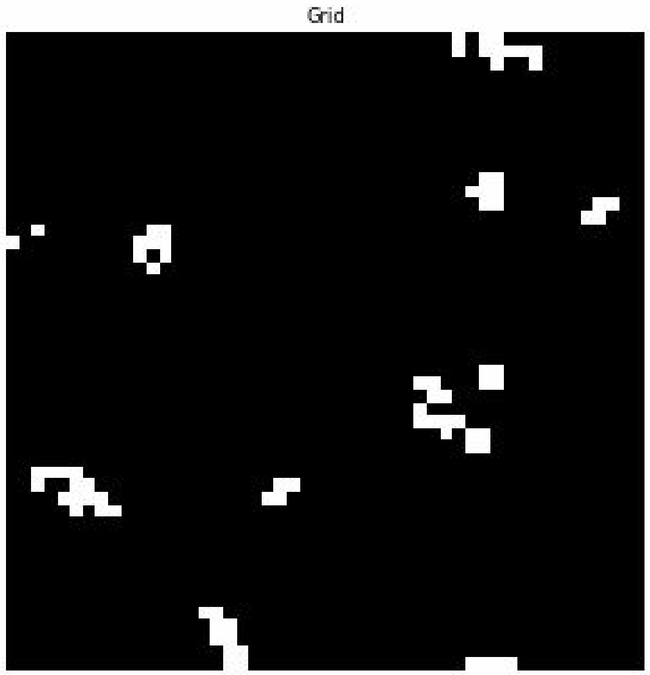


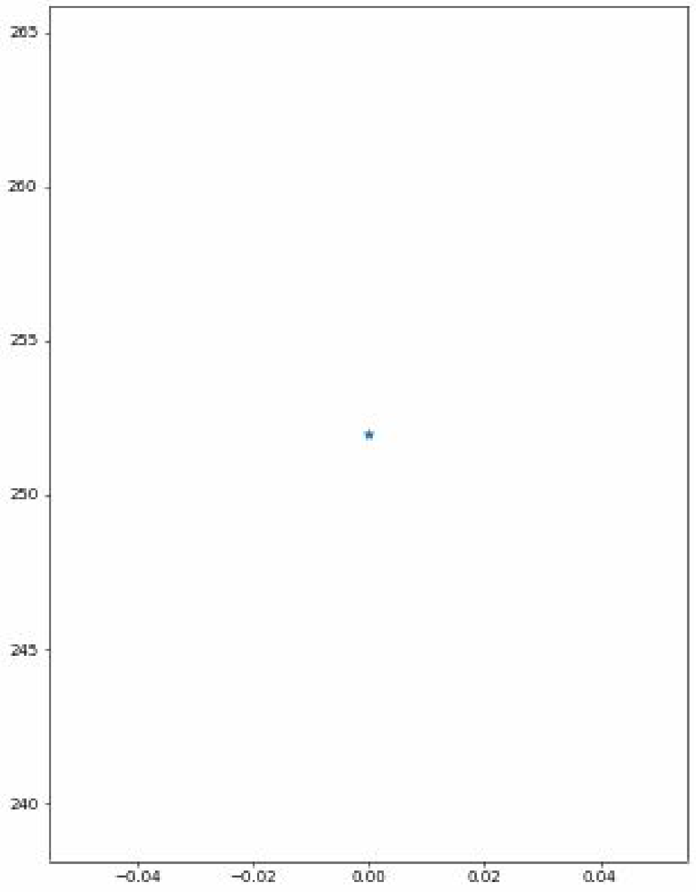
Task 9: Conway’s Game of Life

Like Schelling’s model, but easier to parallel!

- A grid is a square of N x N cells
- Each cell is either dead or alive
- Each cell has 8 neighbours (the grid is periodic)
- For each cell apply a rule:
    - If cell is dead -- become alive only if exactly 3 neigbours are alive
    - If cell is alive -- stay alive only if it has 2 or 3 alive neighbours

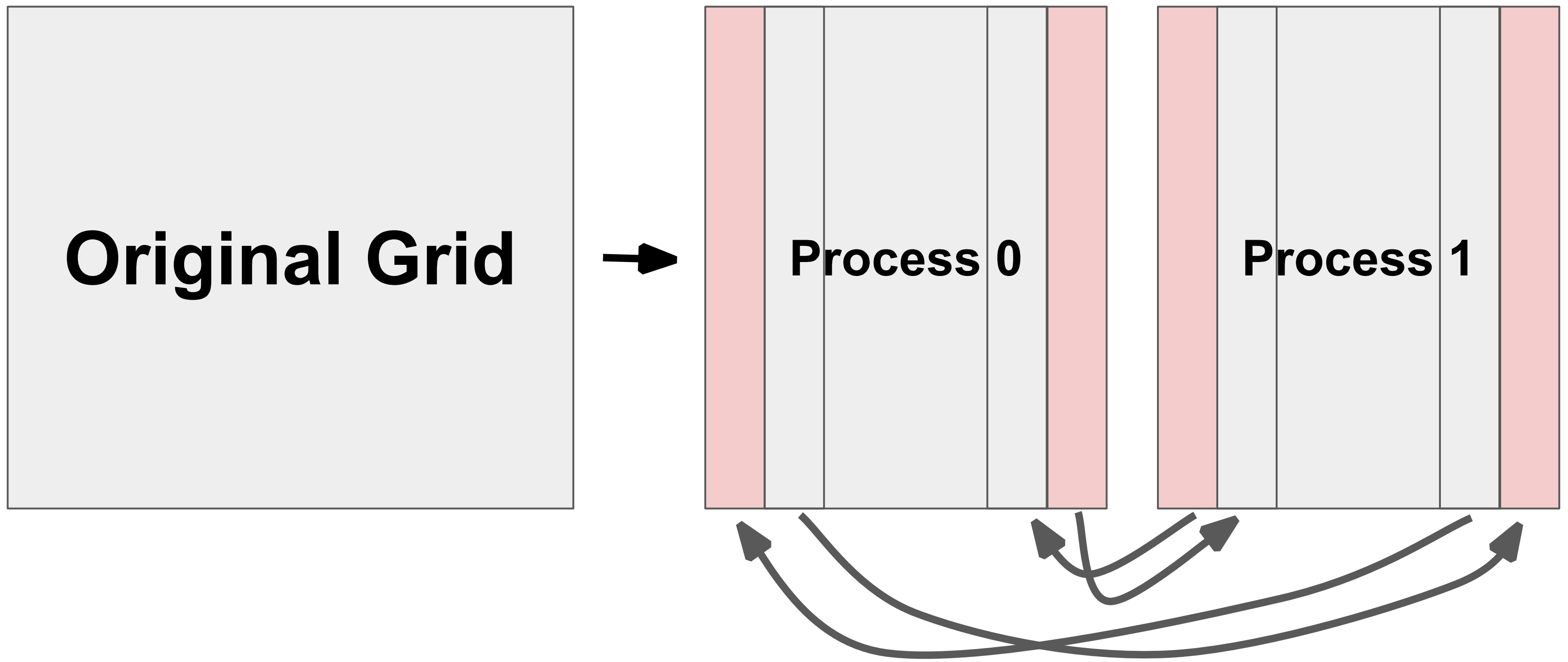
Task 9: Conway’s Game of Life





Task 9: Conway’s Game of Life

Parallelize using “ghost” cells (red):

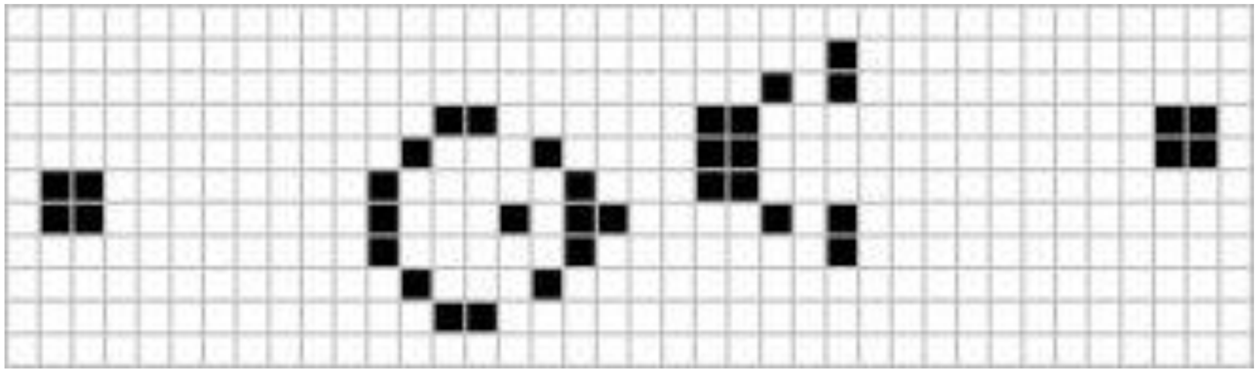


Task 9: Conway's Game of Life

Try different initial conditions:

For example -- Gosper's glider gun:

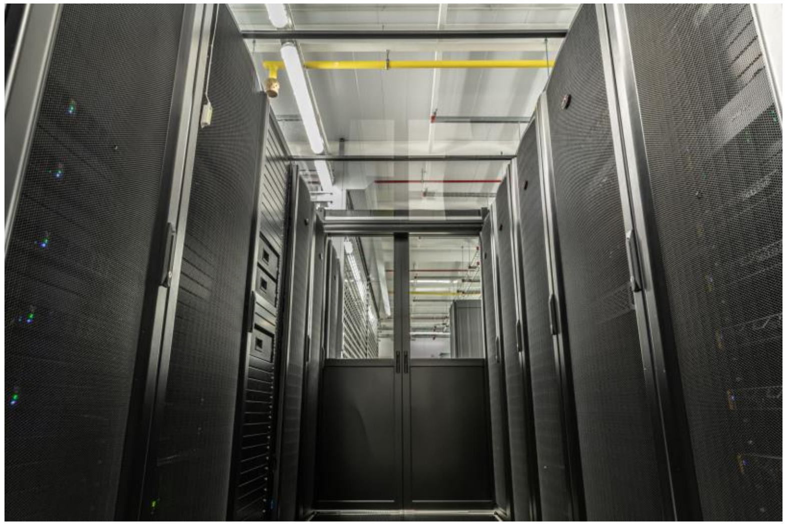
https://tinyurl.com/yx5hy26m



Flagship supercomputer “Zhores” for AI, Big data and HPC

Hybrid energy-efficient architecture

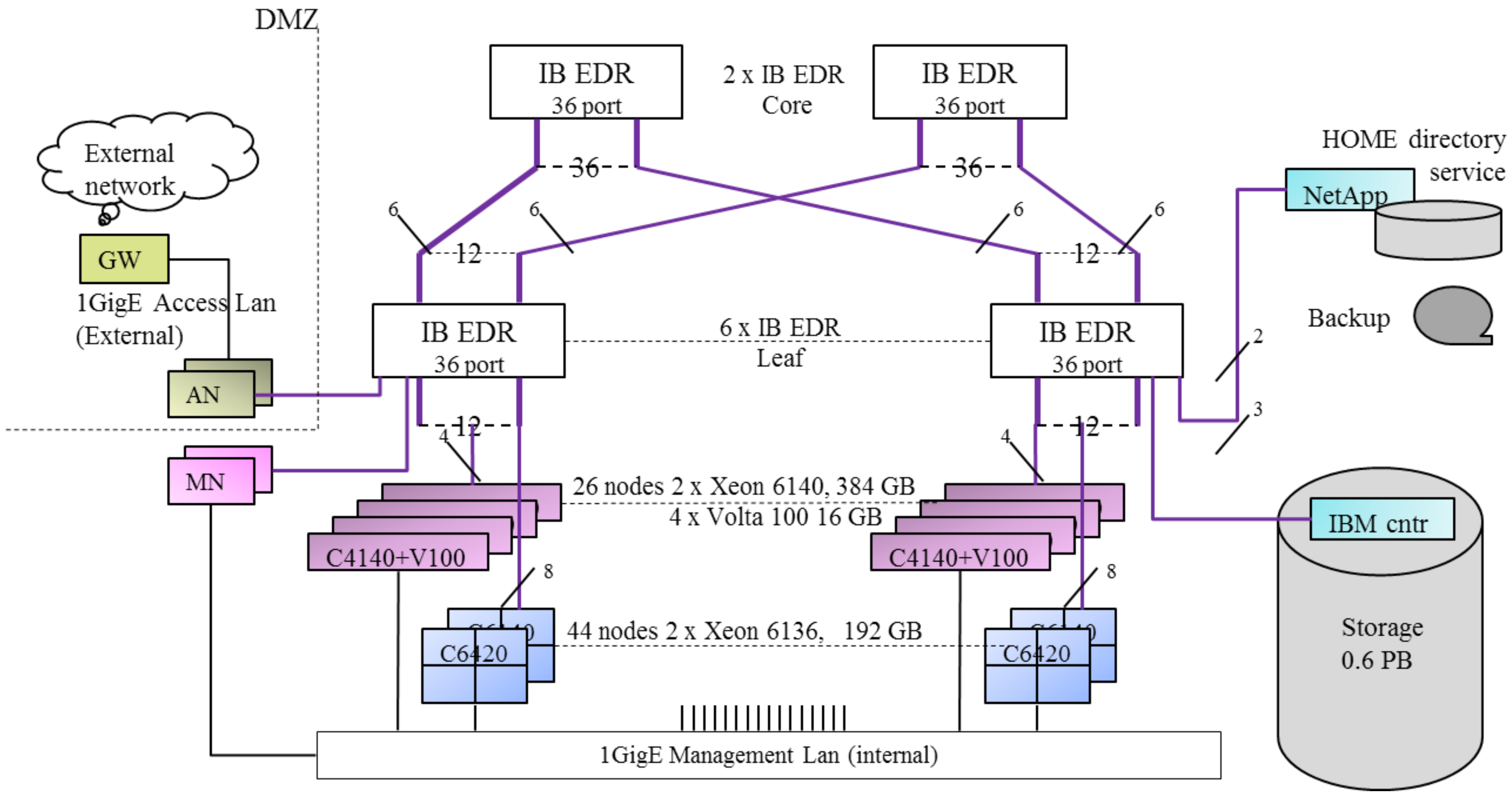
- 74 compute nodes
- 26 nodes with powerful graphic cards Nvidia Tesla V100 (NVLink + RDMA)
- tensor cores for deep learning;
- 90 kWatt power consumption;
- 1PFlops peak performance;
- 0.5 Pbytes storage system
- #6 in Russia
- was installed by our own small team



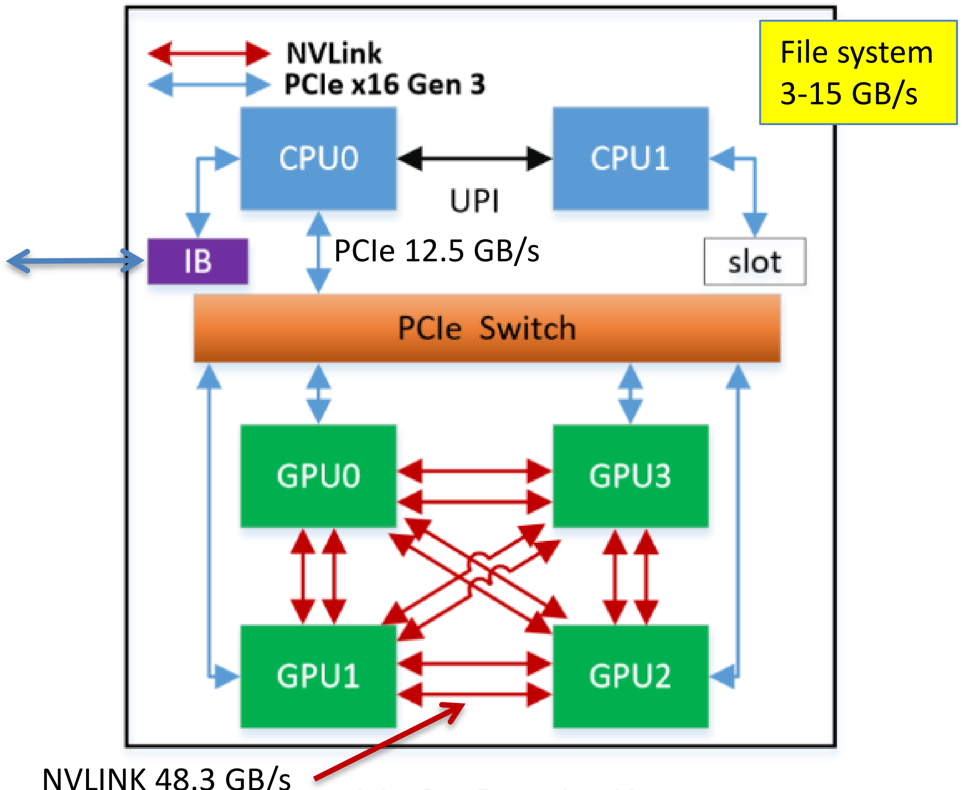
«Zhores» is a unique for Russia supercomputer capable of solving a wide range of interdisciplinnary problems in machine learning, data science and mathematical modeling in such areas as: biomedicine, image classification, Digital Pharma, Photonics, predictive maintenance, new X- and gamma-ray sources

ZHORES SUPERCOMPUTER
HIGH PERFORMANCE COMPUTING AND BIG DATA
SKOLTECH CDISE

Zhores Cluster Structure



Main power is in the GPU nodes



(a) Configuration K

Platform	PowerEdge C4140 (configuration K and M)
CPU	2xCPU Intel 6140 (18c, 2.3 GHz)
Memory	384 GB DDR4 @ 2666MHz
GPU	NVidia Tesla V100 SXM2 (16 Gb @ 900 GB/s)

„Zhores“ configuration details

PART	Nodes name #		Node characteristics							total		
			CPU	#cores	F[GHz]	M [GB]	S [GB]	#IB	[TF/s]	#cores	M[GB]	P[ǂ][TF/s]
C6420	cn	44	6136	2 x 12	3.0	192	480	1	2.3	1056	8448	101.4
C4140	gn	26	6140	2 x 18	2.3	384		2	2.6	936	9984	68.9
			V100	4 x 5120	1.52	4x16		NVL	31.2	532480	1664	811.2
C6420	hd	4	6136	2 x 12	3.0	192	9 TB	1	2.3		768	9.2
	an	2	6136	2 x 12	3.0	256		1	2.3		512	4.6
	vn	2	6134	2 x 8	3.2	384			1.6		768	3.2
	anlab	4	6134	4 x 8	3.2	192		1	3.3		768	13.1
Totals	82			2296*		21248				2296	21248*	1011.6

Notes:

Xeon DP Performance per core: F[GHz] x 32 [DP/clock] , i.e. 3.0 x 32 = 96 GF/s , 2.3 x 32 = 73.6 GF/s

V100 DP Performance per GPU: 1.53 [GHz] x 1 [DP/clock] x #cores i.e. 1.53x5120 = 7.8 Tflop/s

ǂ Theoretical performance is calculated using the base (non AVX) frequency for Intel Xeon.

*GPU memory and cores are not included in the total