LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prof. Dr. D. Kranzlmüller, Dr. K. Fürlinger

Parallel Computing

WS 2017/18

Session 6: MPI Recap, Stencil Operations

Tobias Fuchs, M.Sc.

tobias.fuchs@nm.ifi.lmu.de

Two-sided vs. One-sided

Two-sided

- **Memory is private** to each process.
- When the sender calls the MPI_Send operation and the receiver calls the MPI_Recv operation, data in the sender memory is copied to a buffer then sent over the network, where it is copied to the receiver memory.
- **Drawback:** sender has to wait for the receiver to be ready to receive the data before it can send the data.
- Both sender and receiver have to state a specific call for the communication.

→ **Coupled program flow**

One-sided

- Sections in local memory are made **accessible among processes**.
- Requires **only one process to transfer data**, decouples data transfer from system synchronization.
- **MPI 3.0** supports one-sided passive target communication **without the intervention of the remote process** via Remote Direct Memory Access (RDMA).
  That is: send or receive data without any local action.

→ **Decoupled program flow**

One-sided

- Sections in local memory are made **accessible among processes**.
- Requires **only one process to transfer data**, decouples data transfer from system synchronization.
- **MPI 3.0** supports one-sided passive target communication **without the intervention of the remote process** via Remote Direct Memory Access (RDMA).
  That is: send or receive data without any local action.

→ **Decoupled program flow**

Hard to exploit in the real world

In the real world, passive RDMA in most MPI implementations is either buggy, or inefficient (barrier spin-locks and other delightful hacks) or both, but it's getting better.

One-sided Operations

Standard	Request-based
`MPI_Put`	`MPI_Rput`
`MPI_Get`	`MPI_Rget`
`MPI_Accumulate`	`MPI_Raccumulate`

- All data movement operations are **non-blocking**.
- **Requires explicit synchronization call** to ensure completion, e.g.:

```
MPI_Wait(req)
MPI_Win_fence(win)
MPI_Win_flush(win)
...
```

MPI_Get / MPI_Put

Origin: calling (i.e. local) process
Target: remote process

```
MPI_Get(oaddr, ocount, otype,
        trank, tdisp, tcount, ttype, window)
```

Transfer elements from target in `window[tdisp:tcount]` into local buffer oaddr at origin.

```
MPI_Put(oaddr, ocount, otype,
        trank, tdisp, tcount, ttype, window)
```

Transfer elements from local buffer oaddr at origin to target into `window[tdisp:tcount]`.

(Blocking / Nonblocking) x (One-sided x Two-sided)

	Blocking	**Nonblocking**
Two-sided	MPI_Send	MPI_Isend
One-sided	MPI_Put	MPI_Rput

One-sided communication can be used to implement collective operations.

Pop quiz: How would you implement a reduce operation using one-sided communication?

Example: **Find minimum value in distributed array**.

One-sided true passive RMA Example

```
MPI_Comm comm = MPI_COMM_WORLD;
// Create local window buffer of 4096 integers:
int *    winbuf;
MPI_Alloc_mem(sizeof(int)*4096, MPI_INFO_NULL, &winbuf);
// Create window, collective operation:
MPI_Win  win;
MPI_Win_create(winbuf, sizeof(int)*4096, sizeof(int),
               MPI_WIN_INFO_NULL, comm, &win);
// Start passive RMA epoch, collective operation:
MPI_Win_lock_all(0, win);

...
```

One-sided true passive RMA Example ctd.

```
// rank_0.buf[0:99] <- rank_1.win[1024:1123]

rank 1:                            |   rank 0:
-----------------------------------+------------------------------------
for (int i=0; i<128; i++)          |   int buf[100];
  winbuf[i+1024] = 42;             |   MPI_Get(buf, 100, MPI_INT,
}                                  |           1, 1024, comm);
MPI_Win_flush_all(win);            |   MPI_Win_flush(1, win);
```

What could go wrong?

One-sided true passive RMA Example ctd.

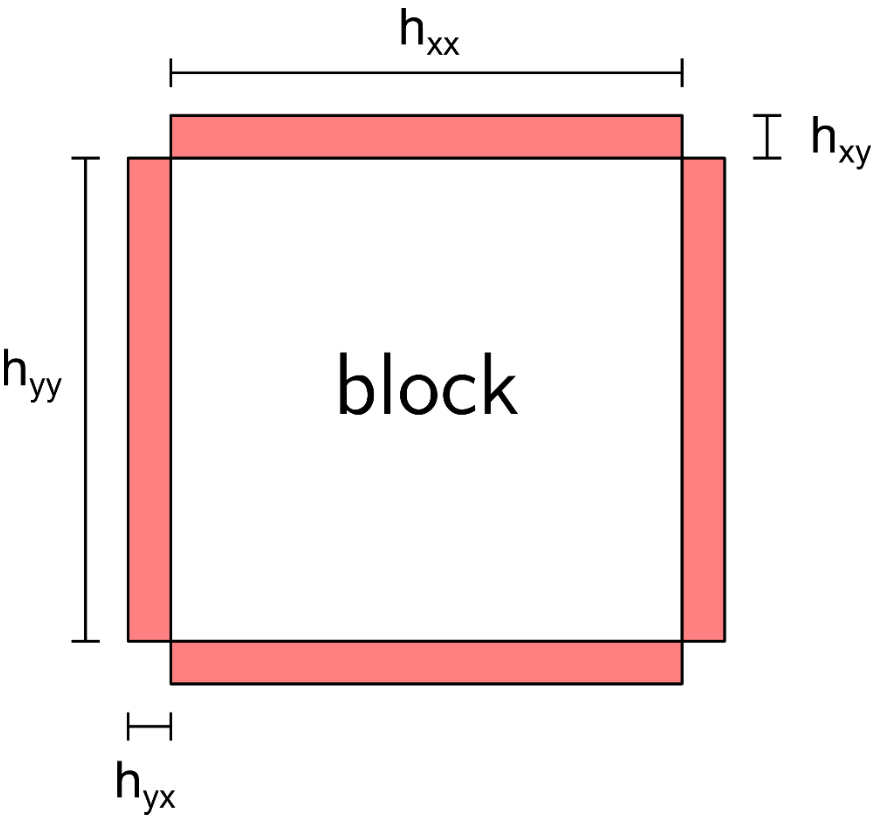
```
// rank_0.buf[0:99] <- rank_1.win[1024:1123]

rank 1:                        |   rank 0:
-------------------------------+------------------------------
for (int i=0; i<128; i++)      |   int buf[100];
  winbuf[i+1024] = 42;         |   MPI_Get(buf, 100, MPI_INT,
}                              |           1, 1024, comm);
MPI_Win_flush_all(win);        |   MPI_Win_flush(1, win);
```

Data races, as known from multi-threading.

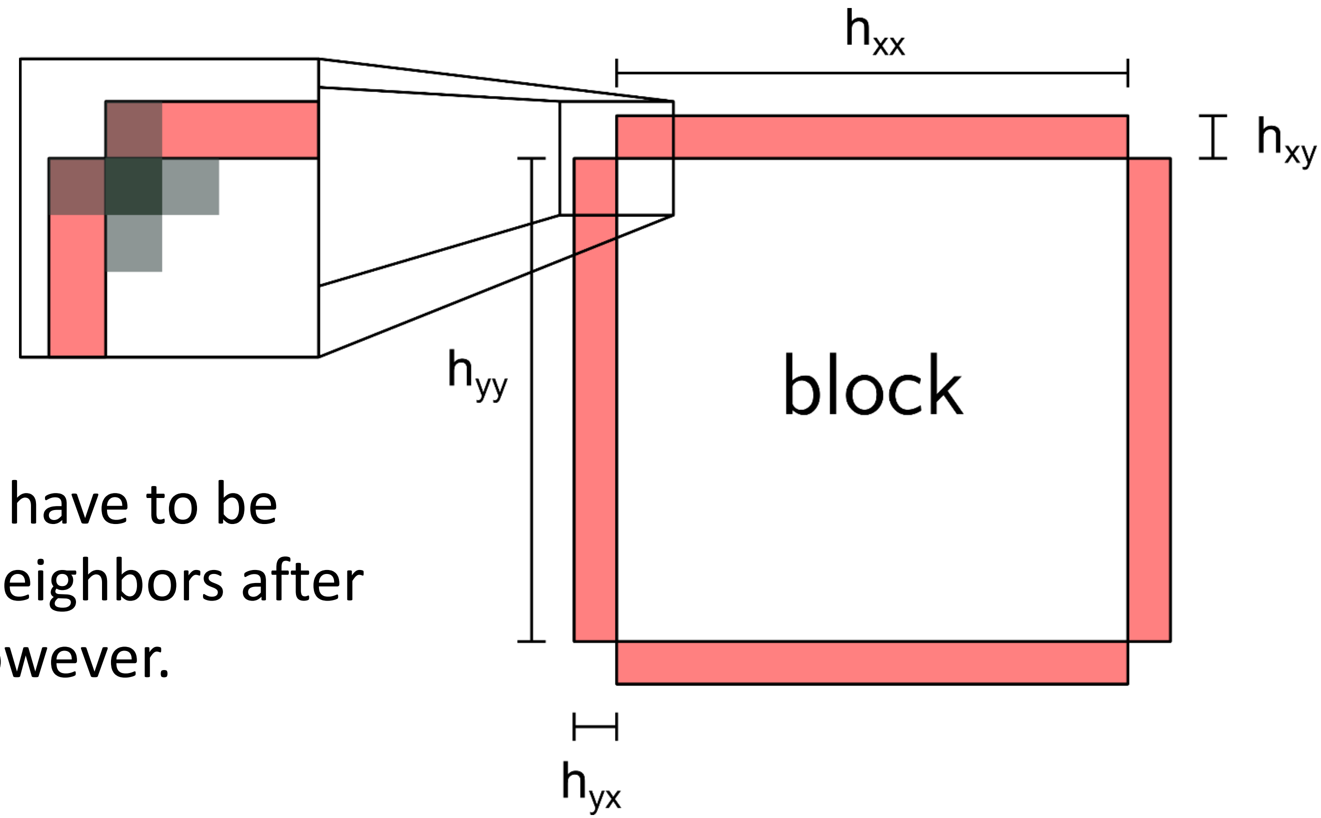
Stencil Operations

rank 0 block [0,0]	rank 1 block [0,1]
rank 2 block [1,0]	rank 3 block [1,1]

Computing the inner-most values in a local block is straight-forward (local-only).



Stencil Algorithms using MPI



Ghost cells (halo) have to be exchanged with neighbors after each iteration, however.

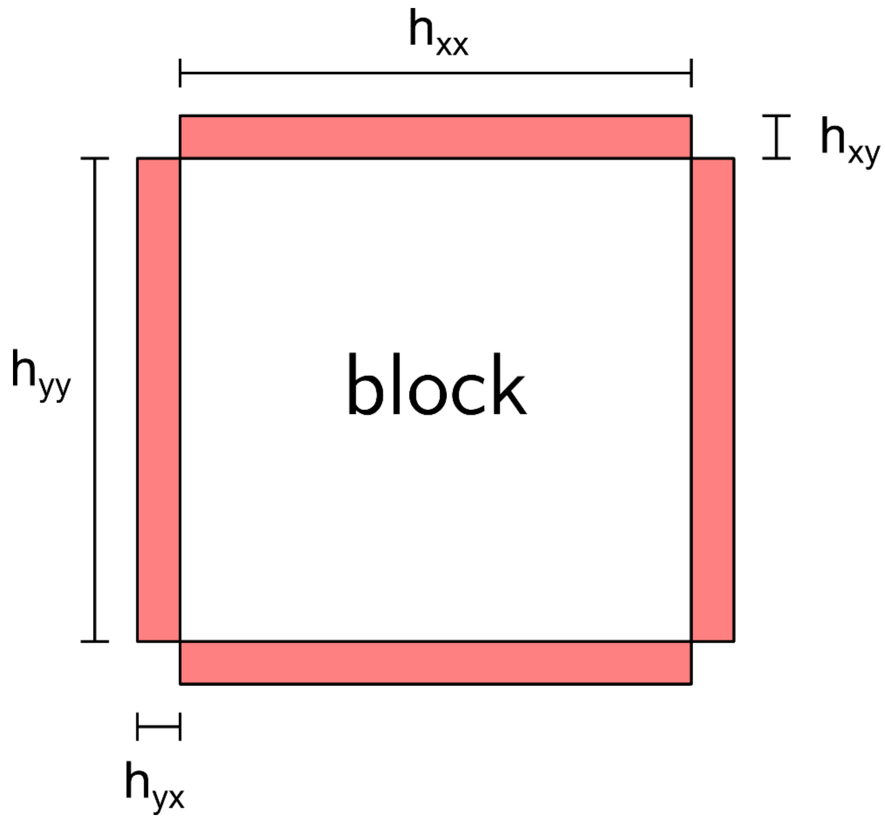
Assuming a 1-nn stencil,

block size $B = b_x \times b_y$

field size $N = n_x \times n_y$

Elements exchanged with all neighbors per block:
$4b_xb_y$

Surface-to-volume ratio?



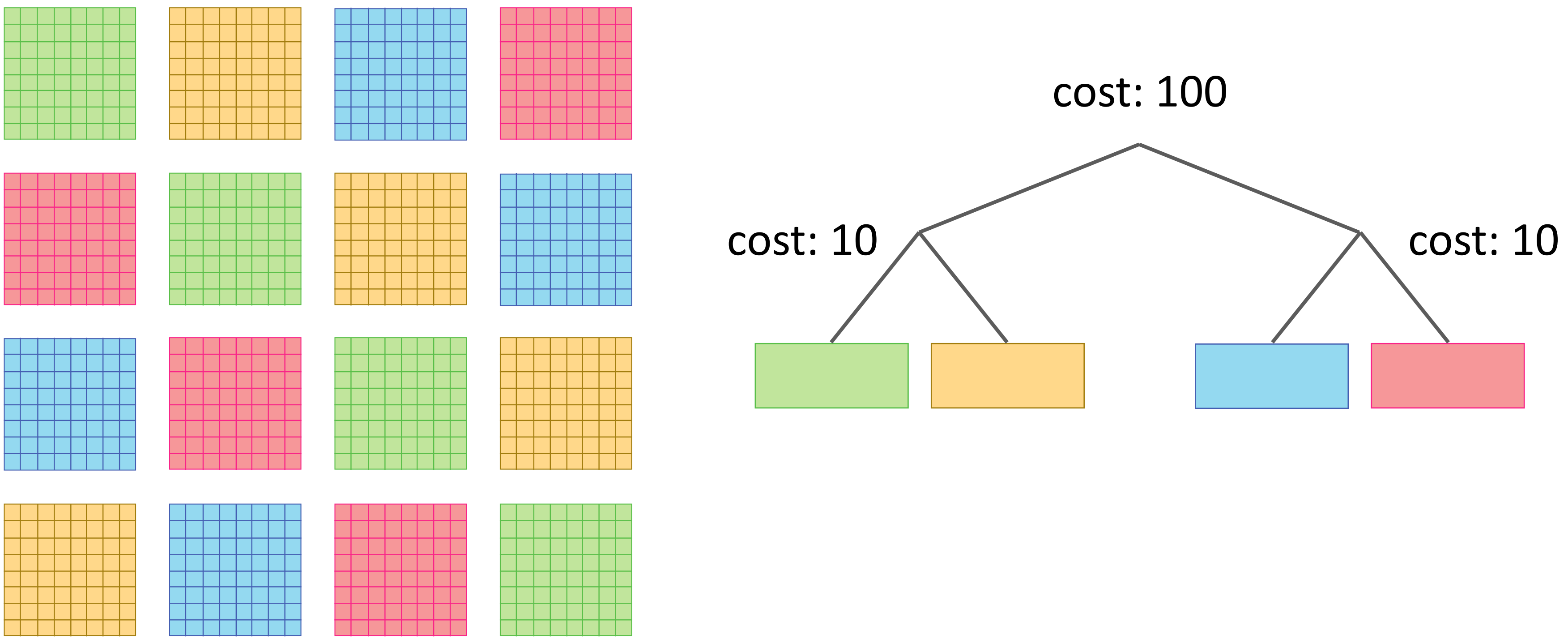
Surface/Volume Ratio

- For high degree of parallelism: **select small block size** (more processes → more blocks → smaller block size)
- But: small block size affects border exchanges, surface/volume ratio increases.
- As the size of a block increases its **volume grows faster than its surface area**.
  Square-Cube Law: O(n3) vs O(n2)
- High ratio → more the time spent on communication per iteration, less time left to spend on actual computations.

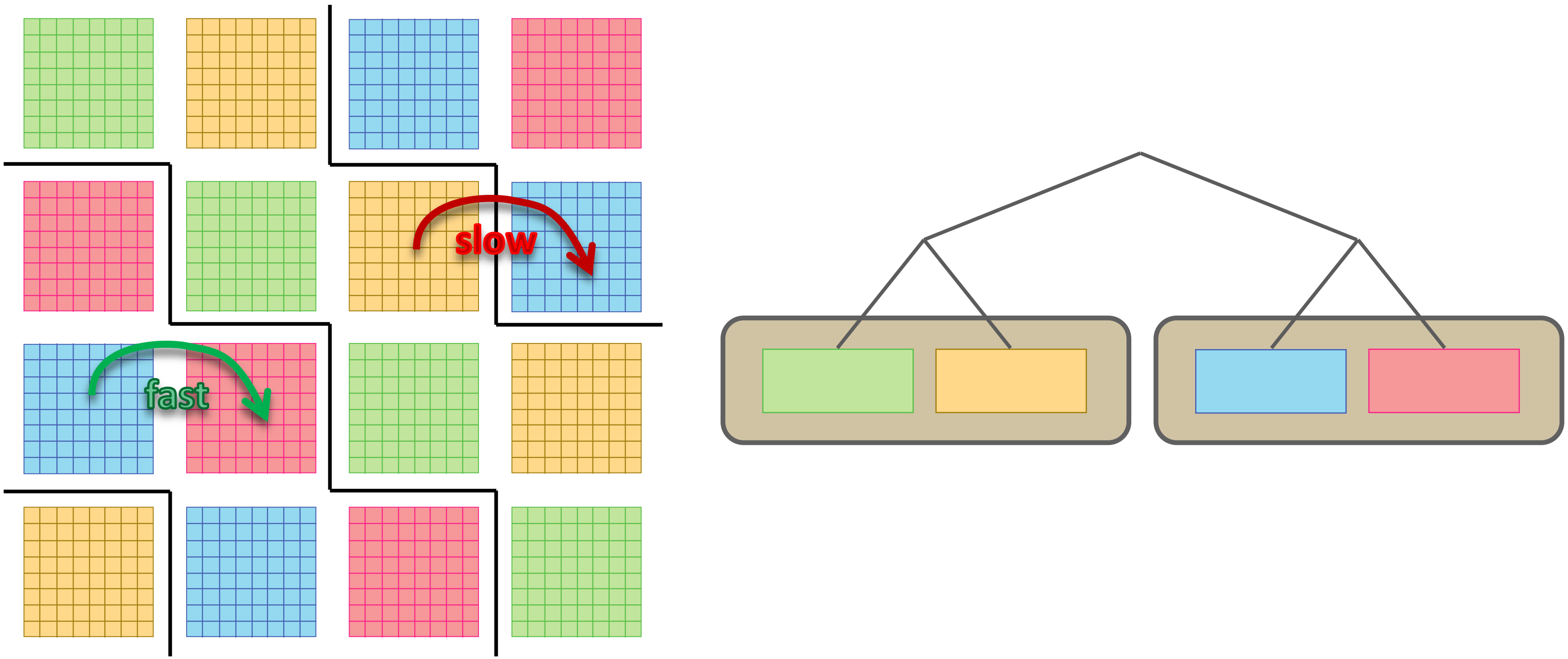
Domain Decomposition vs. Process Topology vs. Hardware Locality

Decomposition vs. Process Topology vs. Hardware

Data distribution, hardware and communication patterns are **highly interdependent with respect to performance**

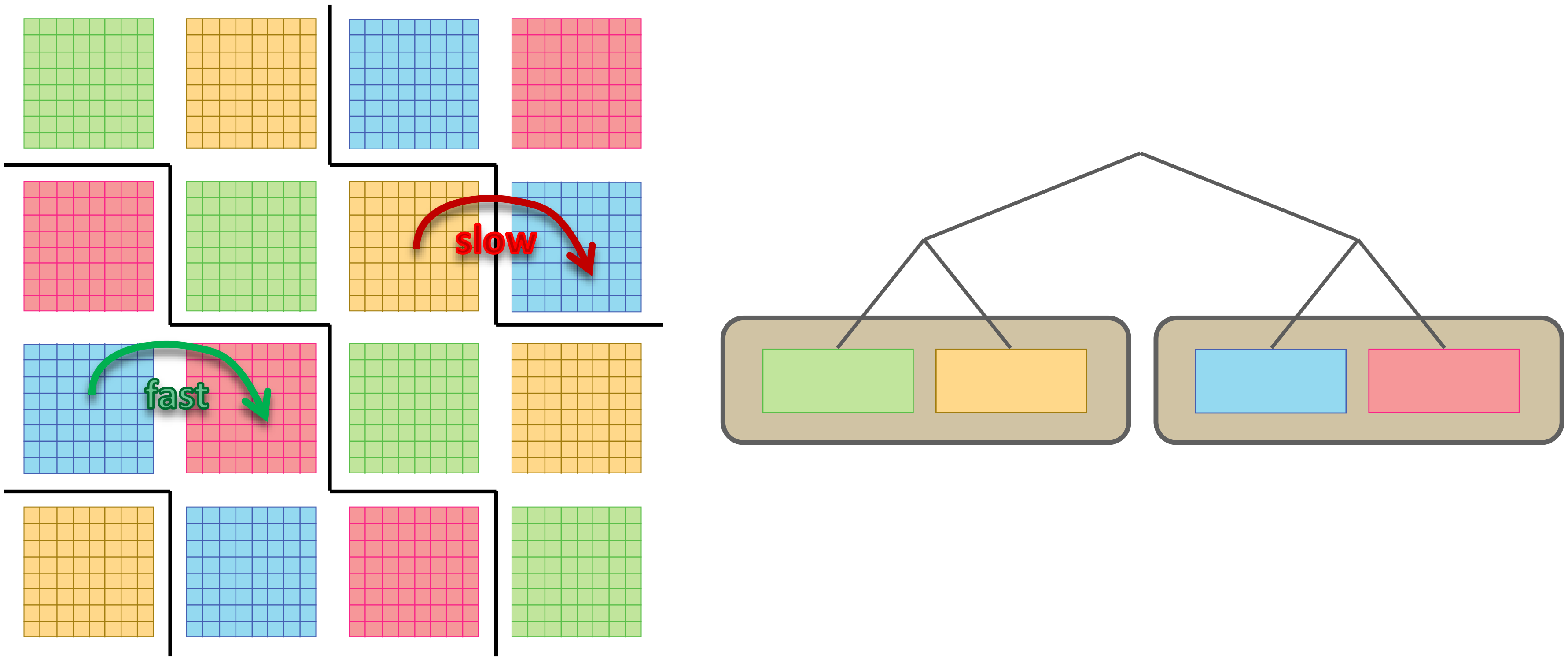


slow boundaries in halo exchange: 12

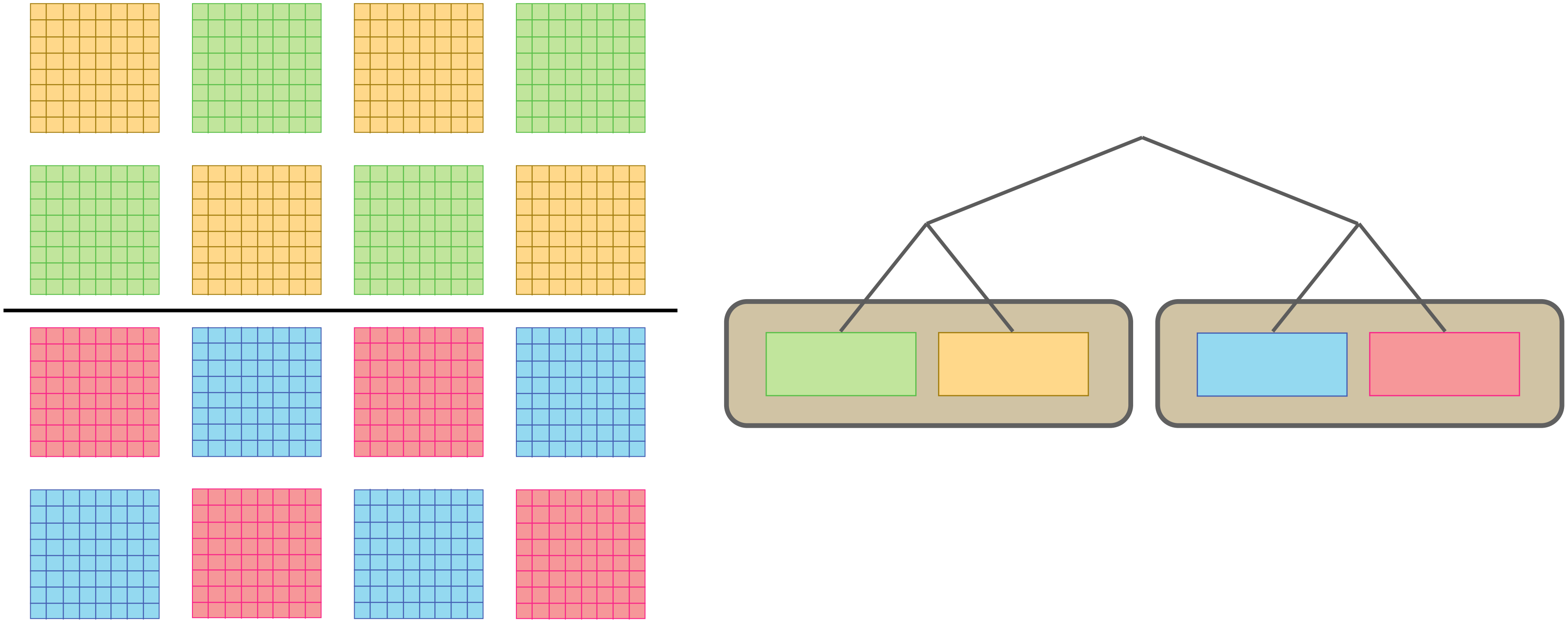


slow boundaries in halo exchange: 12

distinct neighbor ranks per process: 2

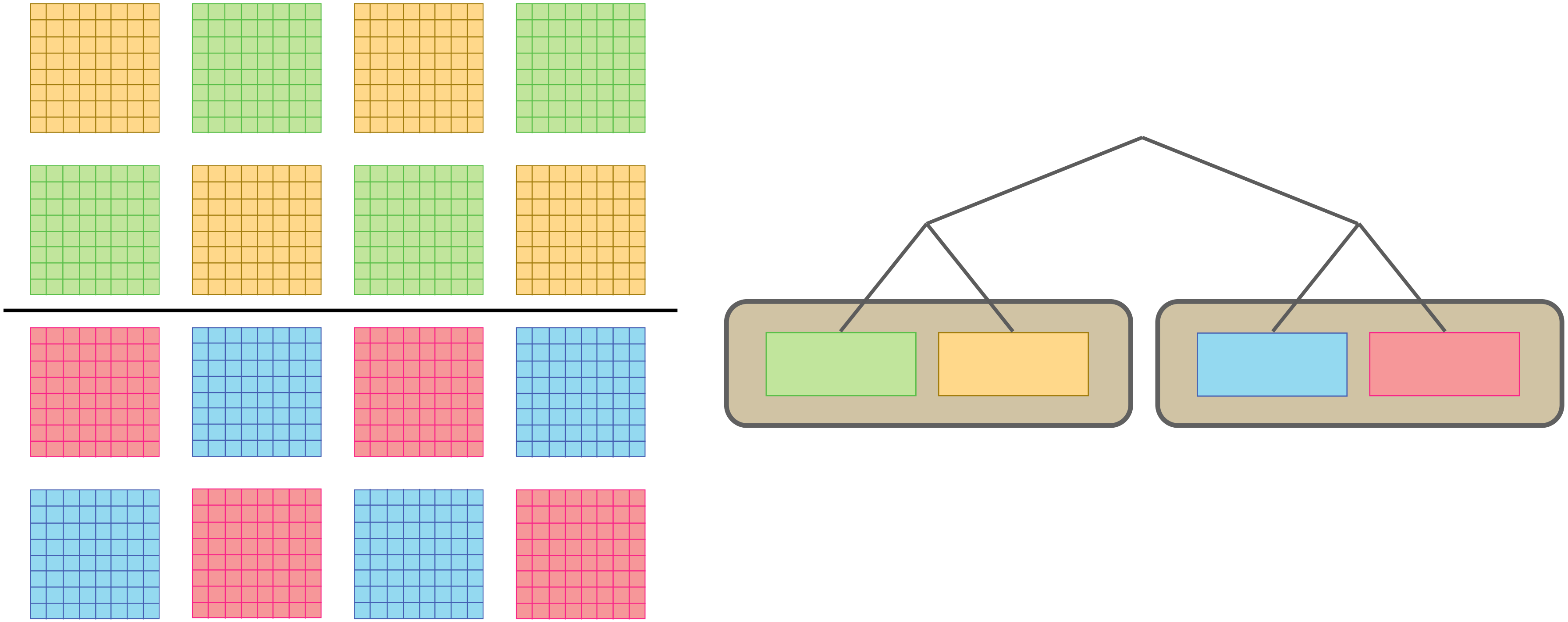


slow boundaries in halo exchange: 4



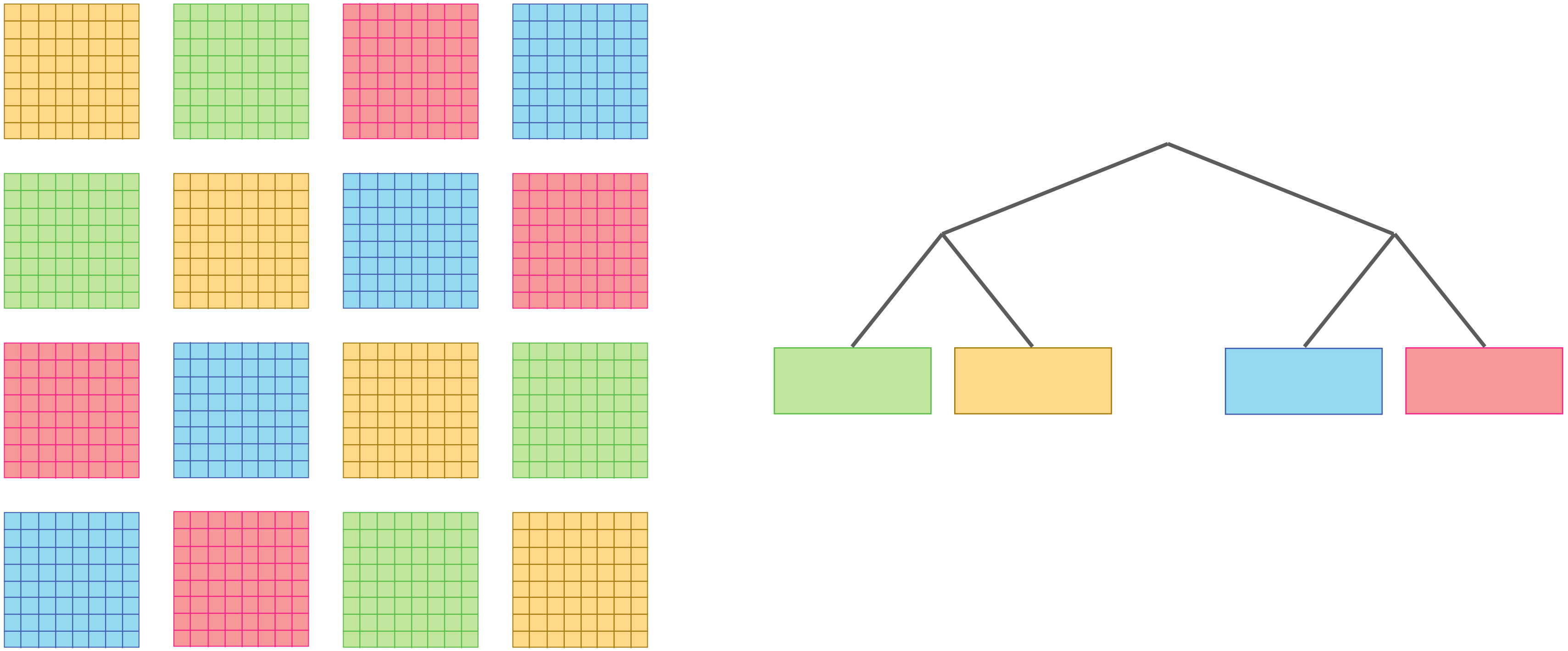
… but isn't time to completion the same?

Time spent in communication per iteration = max. transfer time



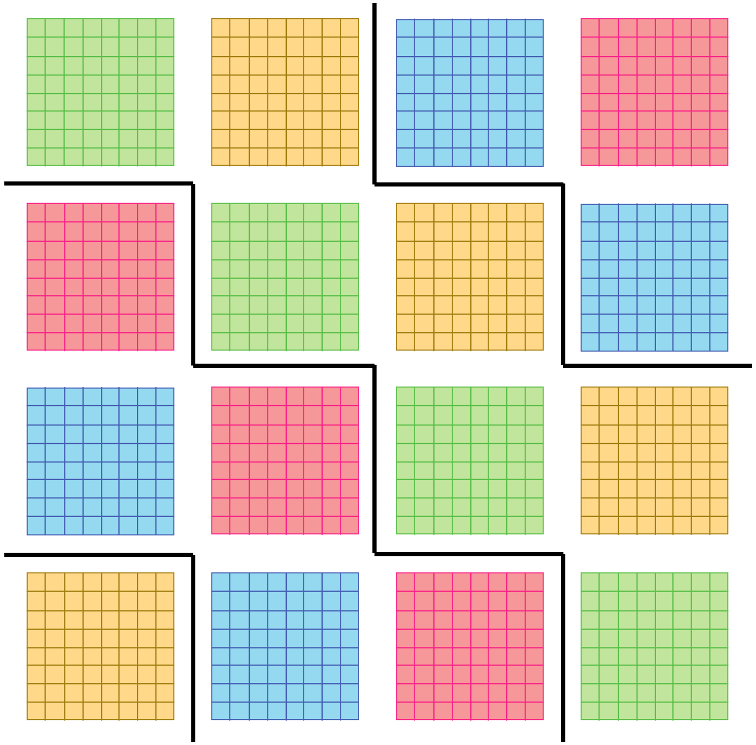
Decomposition vs. Process Topology vs. Hardware

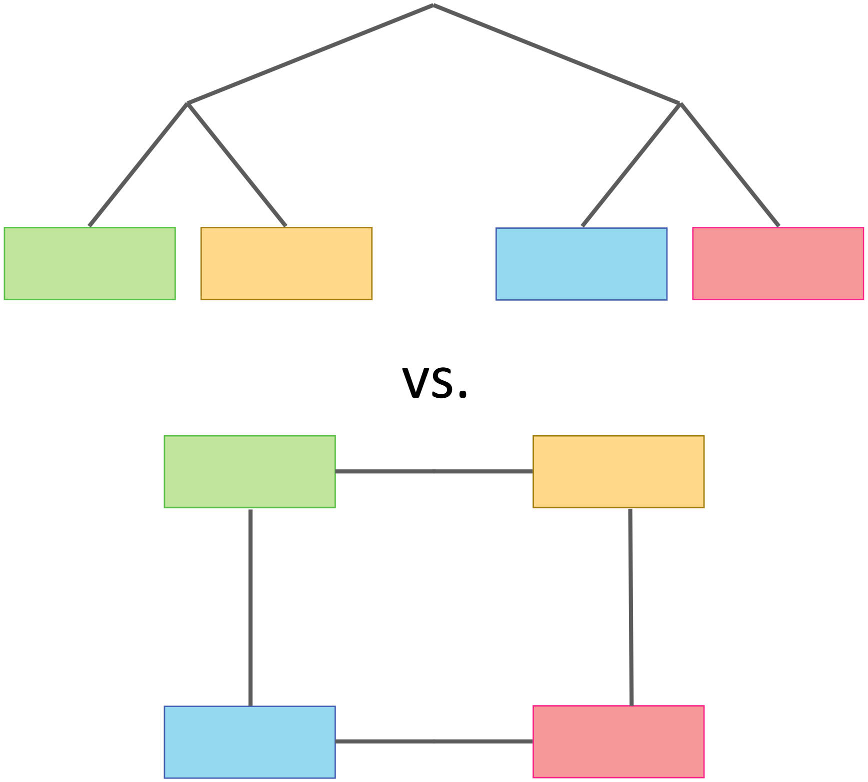
How many slow boundaries for this mapping scheme?



Decomposition vs. Process Topology vs. Hardware

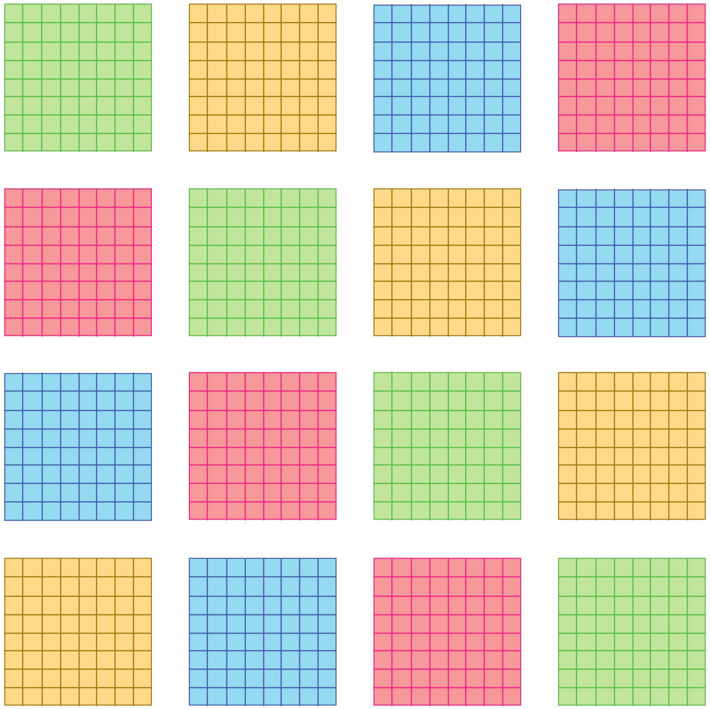
Would a more interconnected, flat hardware topology help?

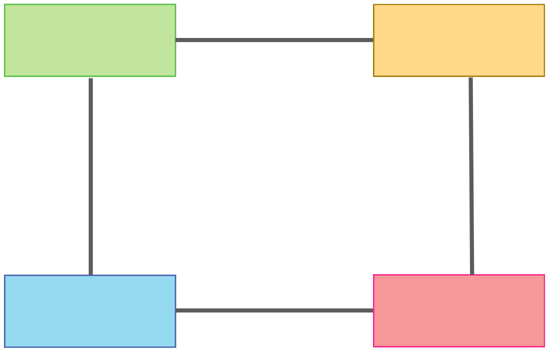




Decomposition vs. Process Topology vs. Hardware

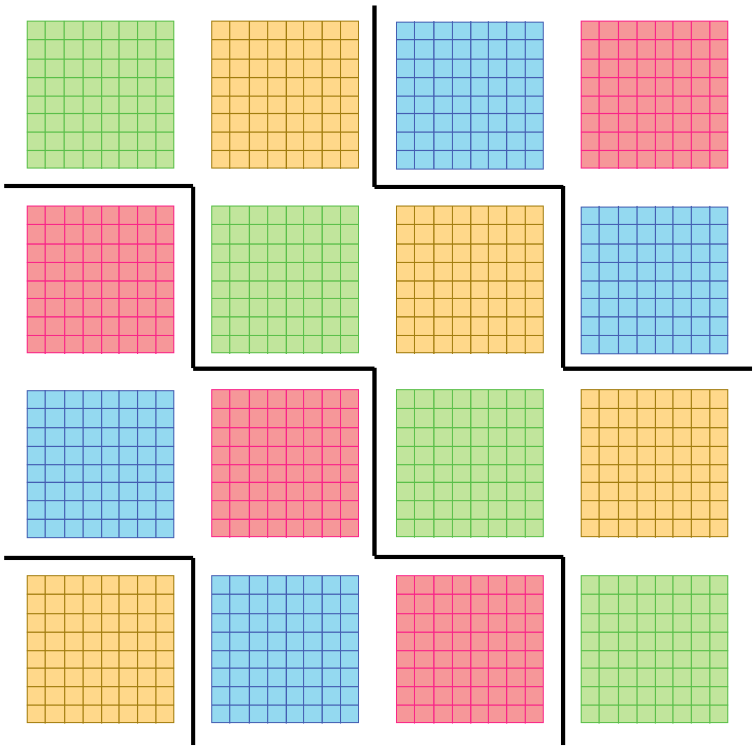
It should!

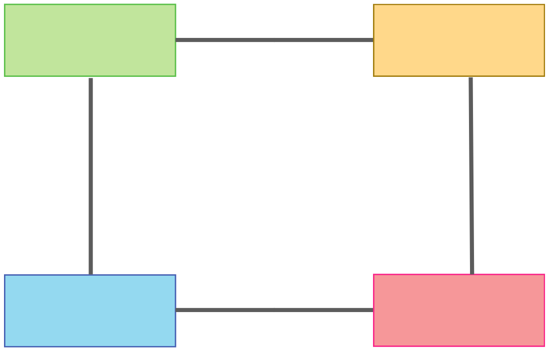




Decomposition vs. Process Topology vs. Hardware

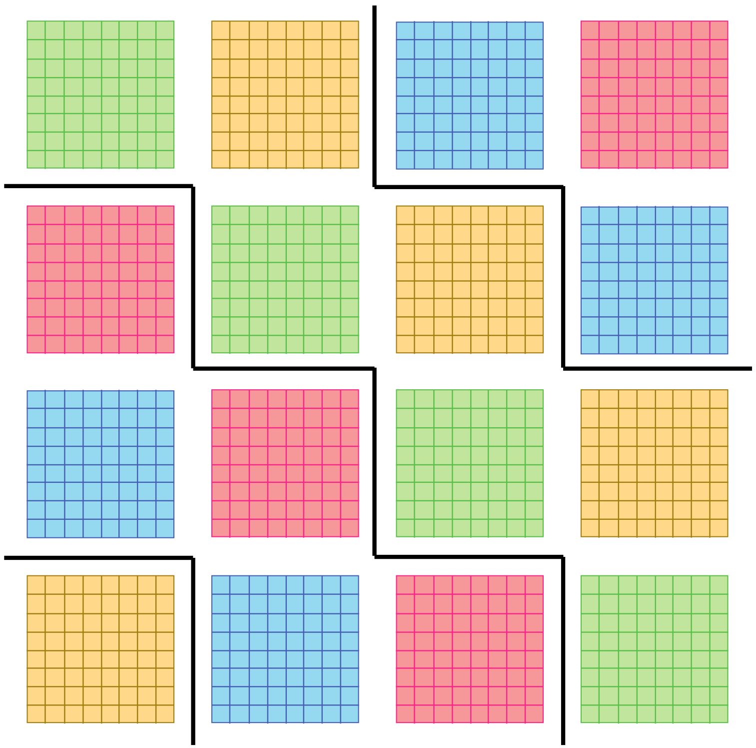
It should! But domain decomposition scheme prevents to exploit it here.

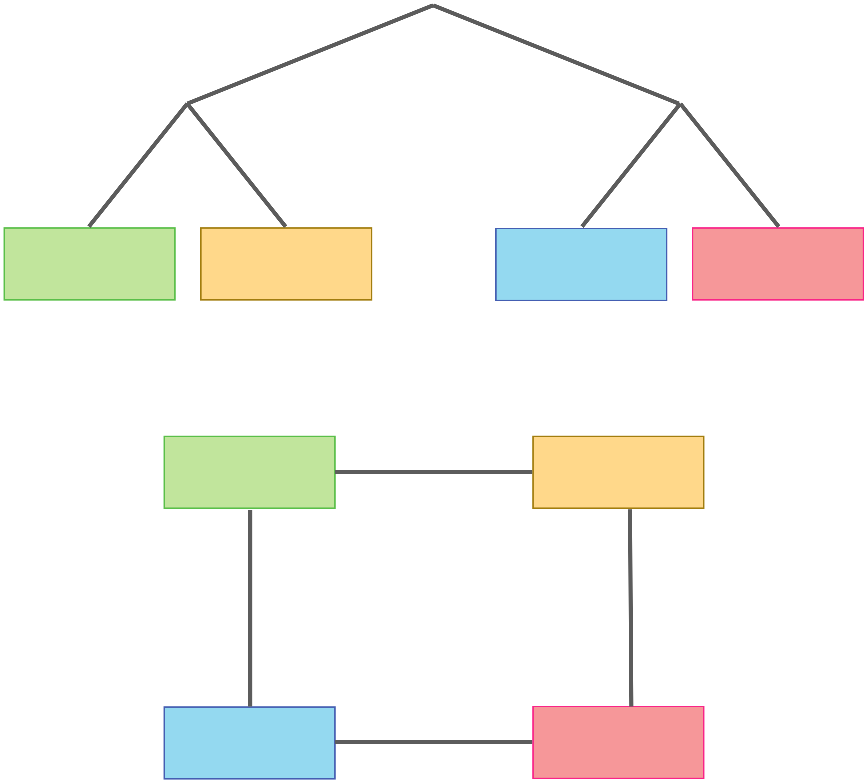




Decomposition vs. Process Topology vs. Hardware

Just for fun, slow boundaries are even identical for either topology in this example.





The one MPI tutorial you all want to read:

Basics:	https://cvw.cac.cornell.edu/MPI/
P2P:	https://cvw.cac.cornell.edu/MPIP2P/
RMA:	https://cvw.cac.cornell.edu/MPIoneSided/
Advanced:	https://cvw.cac.cornell.edu/MPIAdvTopics/

Official MPI 3.1 documentation (Index):
http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm#Node0

Again, a collection of documented MPI examples:
http://www.mcs.anl.gov/~thakur/sc14-mpi-tutorial/

Tobias Fuchs

tobias.fuchs@nm.ifi.lmu.de

www.mnm-team.org/~fuchst