TECHNISCHE UNIVERSITÄT DRESDEN

SCALABILITY IN LARGE COMPUTER SYSTEMS (HPC, CLUSTERS)

DISTRIBUTED OPERATING SYSTEMS, SCALABILITY, SS 2020

(THANKS TO AMNON BARAK, CARSTEN WEINHOLD, MAKSYM PLANETA, ALEX MARGOLIN, ...)

Hermann Härtig, SS 2020

THIS LECTURE

Single Admin Domain,
large number of connected Compute Nodes

- MPI (Short Intro), Partitioning
- Amdahl's law & communication & jitter
  Fault Tolerance
- Load Balancing (Case Study MosiX):
  migration mechanism
  decision making (information dissemination)

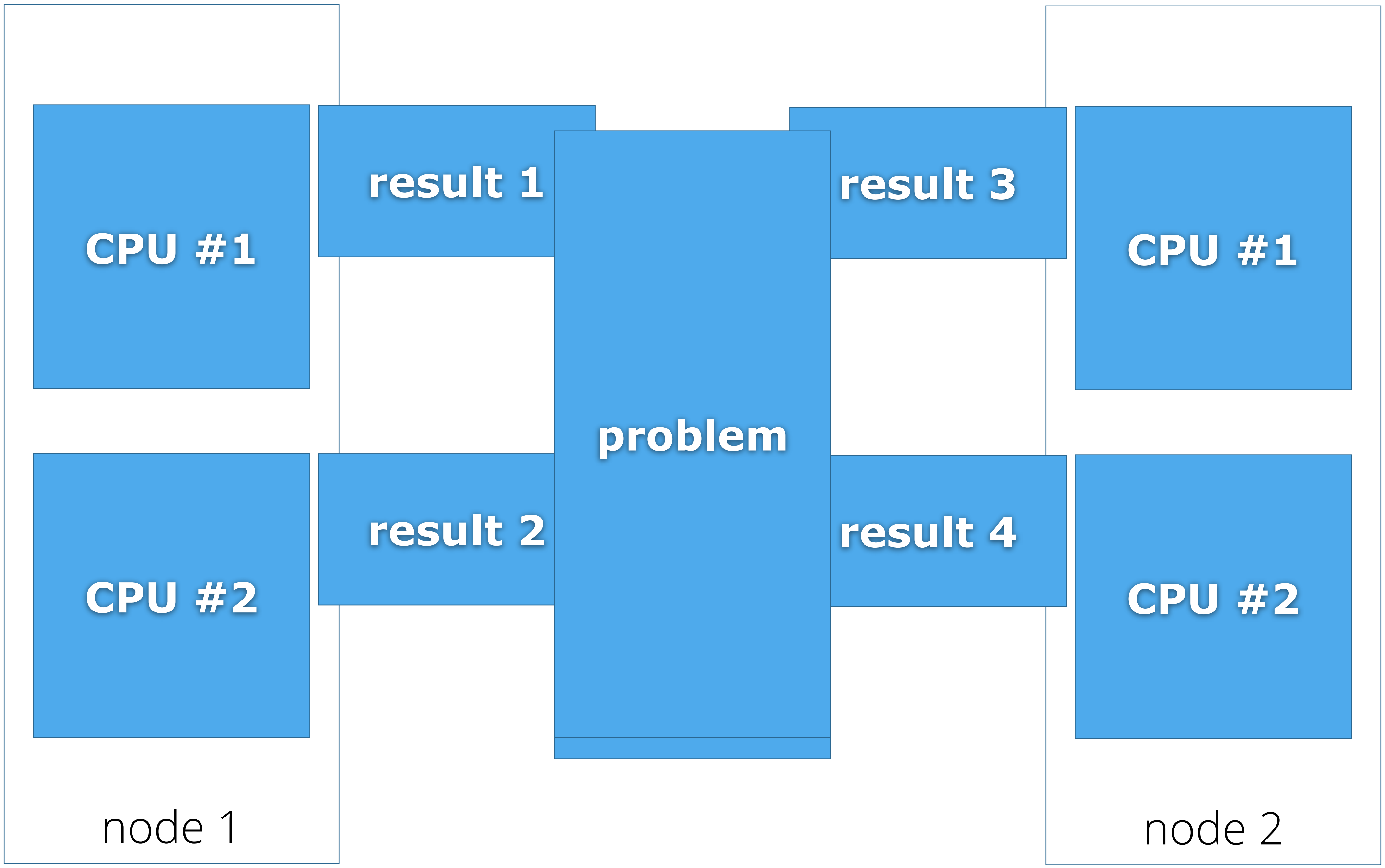
PROGRAMMING MODELS

- independent OS processes
- bulk synchronous execution (HPC)
  - iterate: compute - communicate
  - all processes wait for (all) other processes
- "task-based" …
  - usually small components within OS processes with a data driven interface

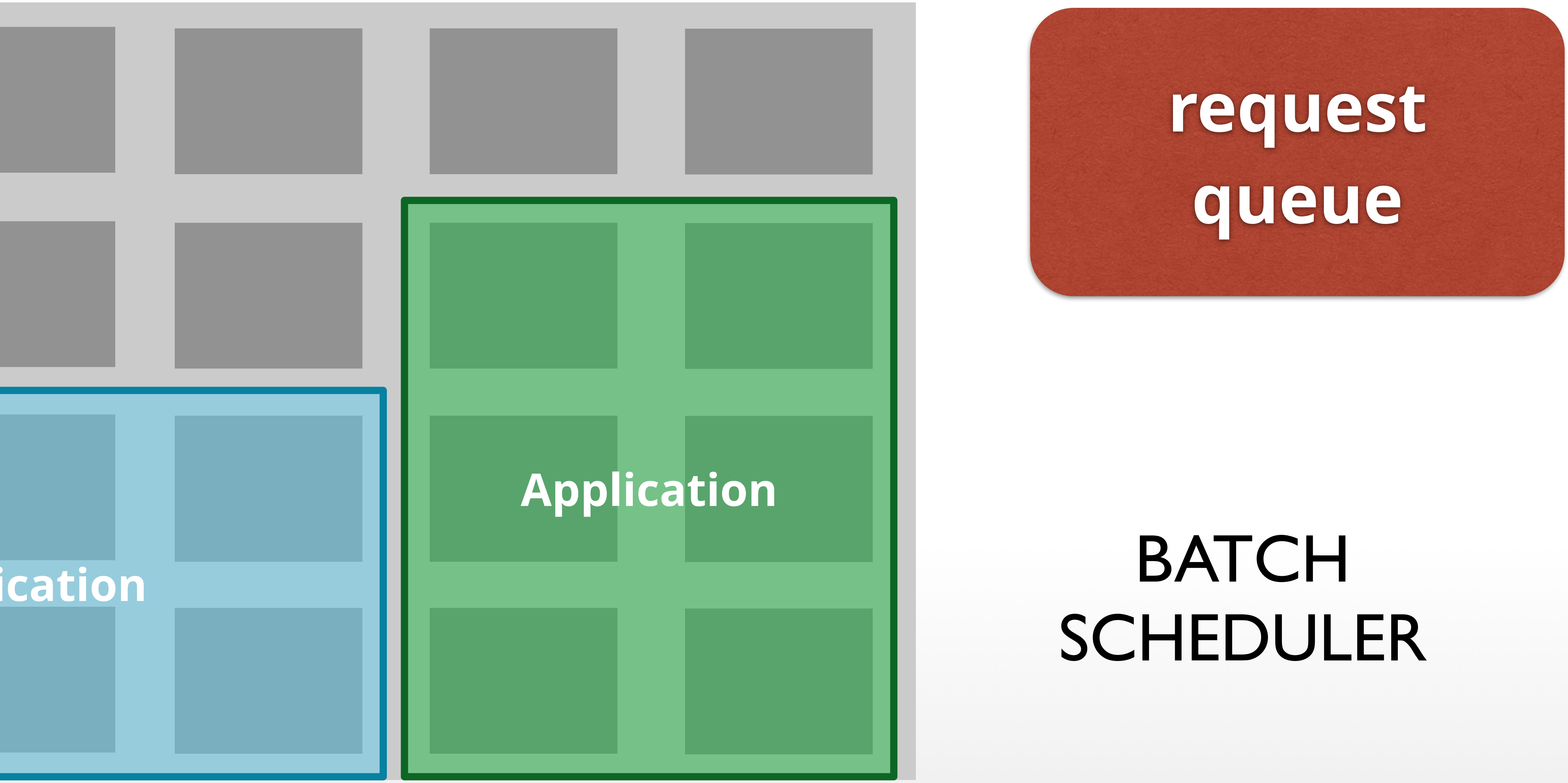
BULK SYNCHRONOUS

- all processes execute same program
- iterate
  { work; exchange data (collective operation)}
  until "result makes sense"
- common in High Performance Computing:
  Message Passing Interface (MPI)
  library

PARTITIONING



HW PARTITIONS & ENTRY QUEUE



BATCH SCHEDULER

- MPI program is started on group of processors:
  called communicator
- `MPI_Init(),MPI_Finalize()`
- `MPI_Comm_size()`
  `MPI_Comm_rank():`
  `“Rank” of process within this set`
- message passing between group members

MPI PROGRAM SKELETON

```
int my-rank, total;
MPI_Init();

MPI_Comm_rank(MPI_COMM_WORLD, &my-rank);
MPI_Comm_size(MPI_COMM_WORLD, &total);

Split (app-data, my-rank) -> my-slice ;

iterate{
 Work on my-slice;
 Exchange data via message passing
} until "result makes sense"

MPI_Finalize();
```

MPI MESSAGES

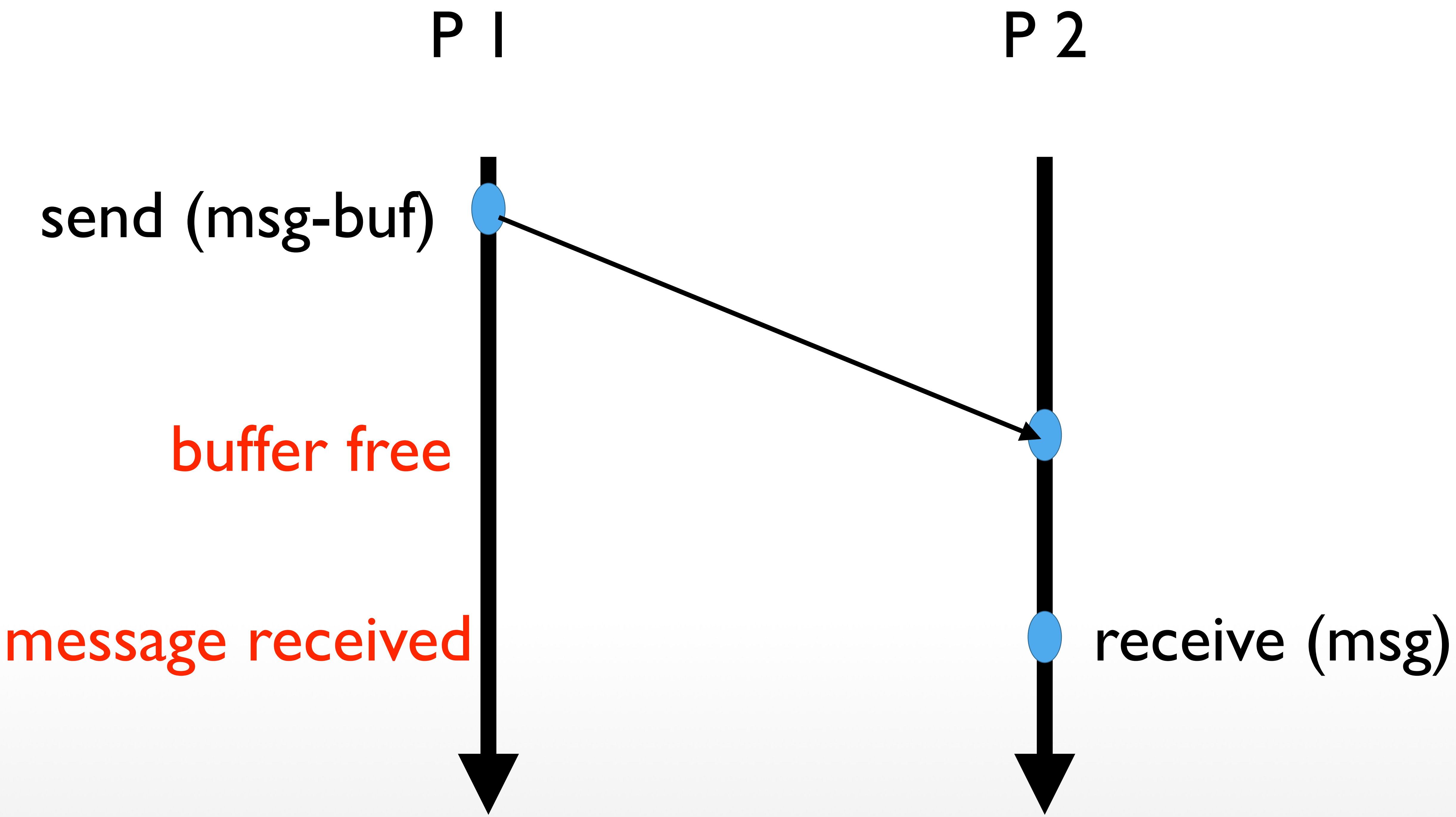
- Communication
  - Point-to-point
  - Collectives

MPI MESSAGES

- Communication
  - **Point-to-point**
  - Collectives

```
MPI_Send(
   void* buf,
   int count,
   MPI_Datatype,
   int dest,
   int tag,
   MPI_Comm comm
 )
MPI_Recv(
   void* buf,
   int count,
   MPI_Datatype,
   int source,
   int tag,
   MPI_Comm comm,
   MPI_Status *status
)
```

LATENCY HIDING

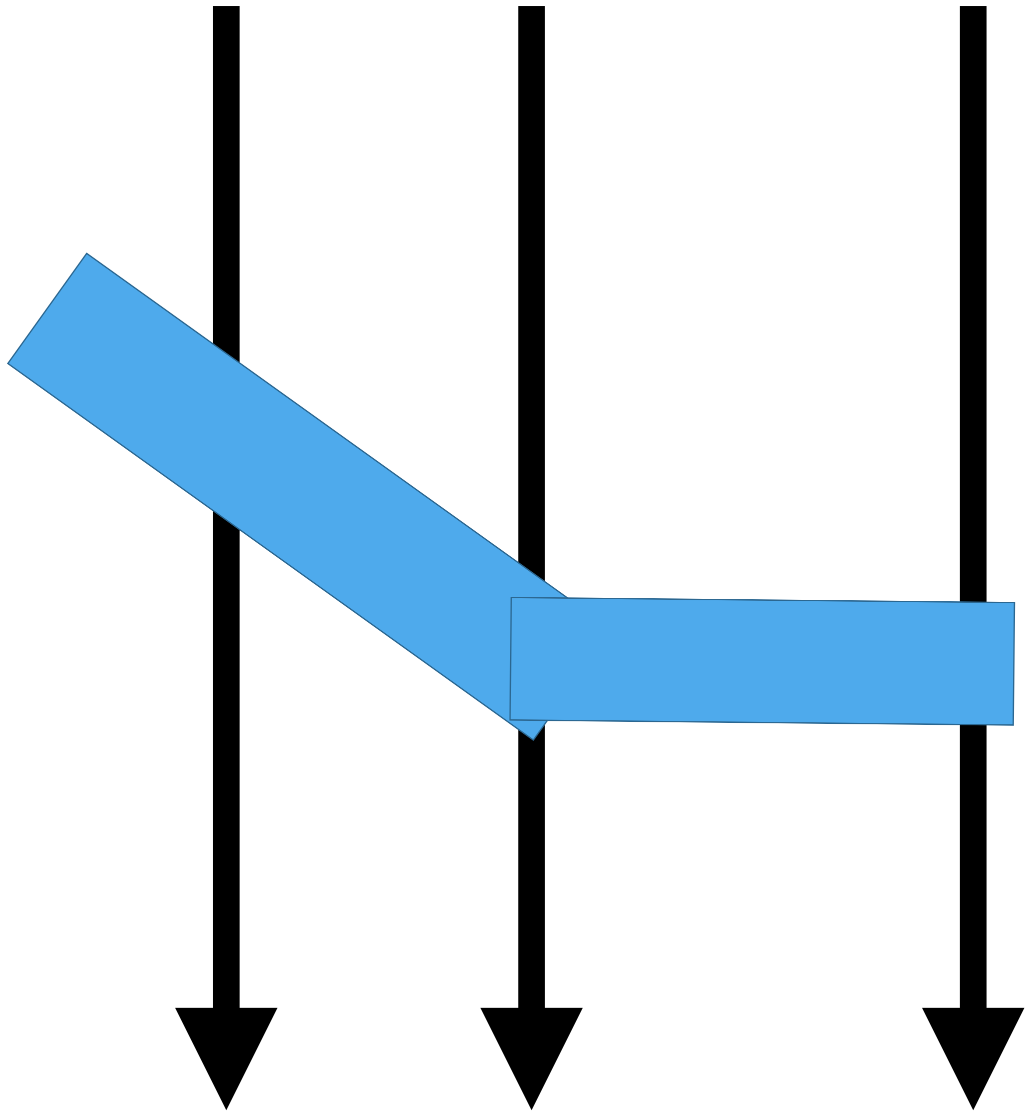


LATENCY HIDING

	blocking call	non-blocking call
synchronous communication	returns when message has been delivered (i.e. received by some)	returns immediately, sender later checks for delivery (Test/Wait)
asynchronous communication	returns when send buffer can be reused	returns immediately, sender later checks for send buffer

MPI COLLECTIVES

- Communication
  - Point-to-point
  - **Collectives all processes of communicator participate**



MPI MESSAGES

```
MPI_Barrier(
 MPI_Comm comm
)
```

- Communication
  - Point-to-point
  - **Collectives**

MPI MESSAGES

- Communication
  - Point-to-point
  - **Collectives**

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

MPI MESSAGES

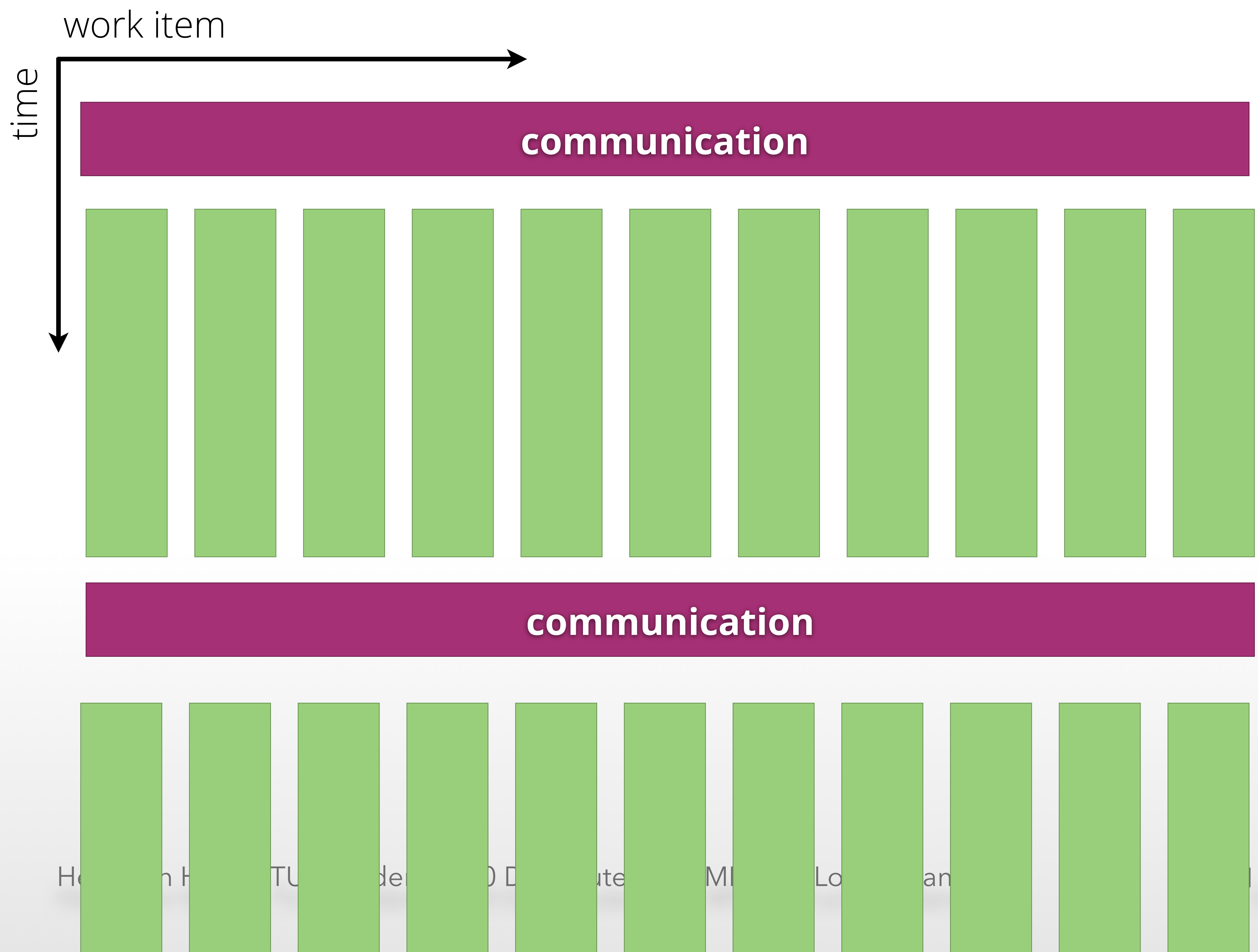
- Communication
  - Point-to-point
  - **Collectives**

```
MPI_Reduce(
   void* sendbuf,
   void *recvbuf,
   int count
   MPI_Datatype,
   MPI_Op op,
   int root,
   MPI_Comm comm
)
```

Single Admin domain,
large number of connected Compute Nodes

- MPI (Short Intro), Partitioning
- Amdahl's law & communication & jitter
  Fault Tolerance
- Load Balancing (Case Study MosiX):
  migration mechanism
  decision making (information dissemination)

AMDAHL & COMMUNICATION



REMEMBER AMDAHLS' LAW(AL)

for parallel systems:

- P: section that can be parallelized
- S: serial section (S)
- N: number of CPUs

Speedup= $$\frac{1}{S + \frac{P}{N}}$$

- next slides:

P, S per iteration step

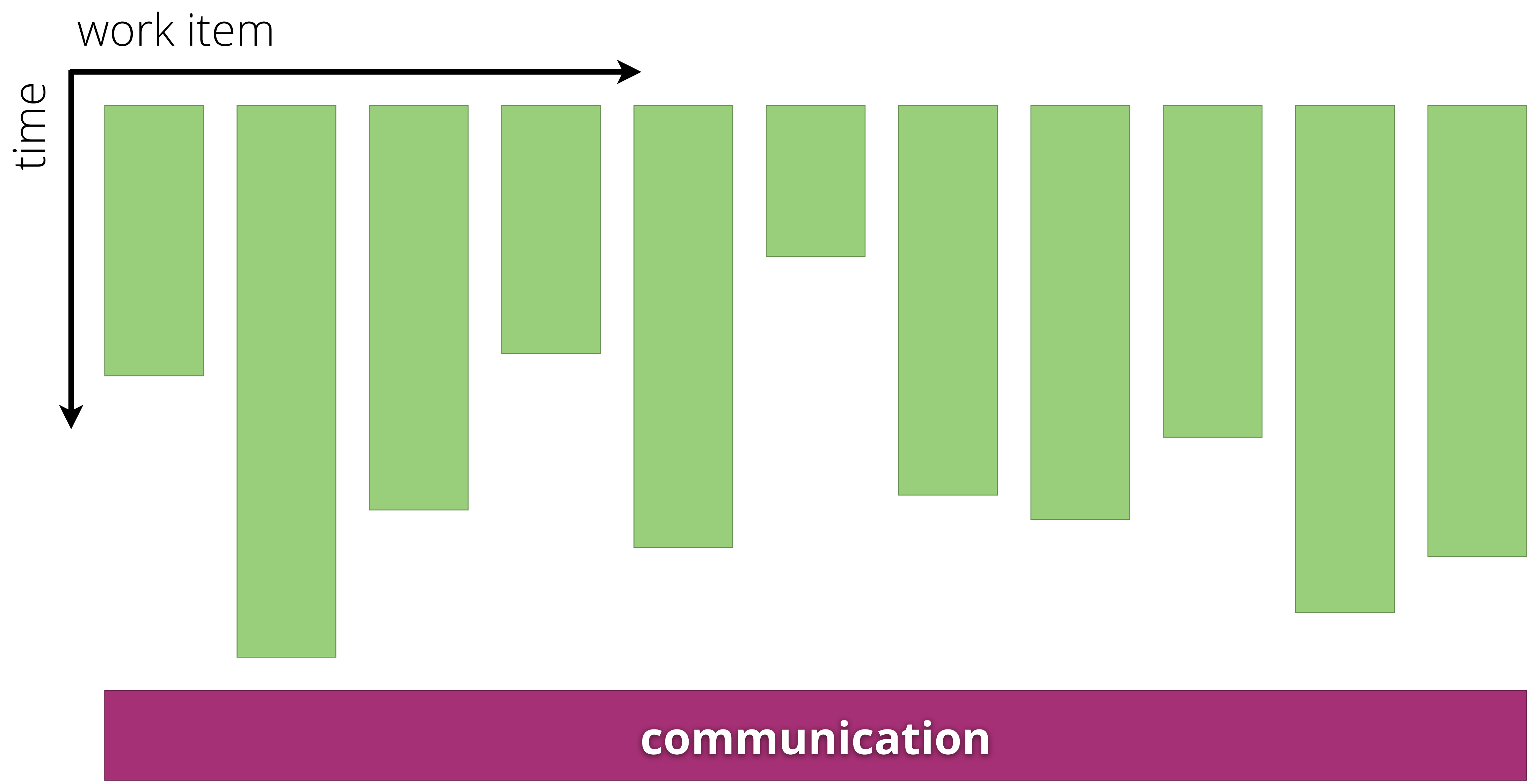
S: communication

P/N: work per process

NUMERIC EXAMPLES

P	N	P/N	S	speedup, ca
1000	1000	1	1	500
1000	10000	0.1	1	909
100	1000	0.1	1	91
10	1000	0.01	1	10
10	1000	0.01	0.01	500

AMDAHL'S LIMITATIONS



AMDAHLS' LAW (MODIFIED)

$$\frac{1}{S + \frac{P}{N}}$$

$$\frac{1}{S + LongestProcess}$$

NUMERIC EXAMPLES

P	N	per proc	S	speedup, ca
10	1000	0.01	0.01	500
10	1000	0.02	0.01	333

SOURCES FOR EXECUTION JITTER

- Hardware
- Application
- Operating system “noise”

OPERATING SYSTEM "NOISE"

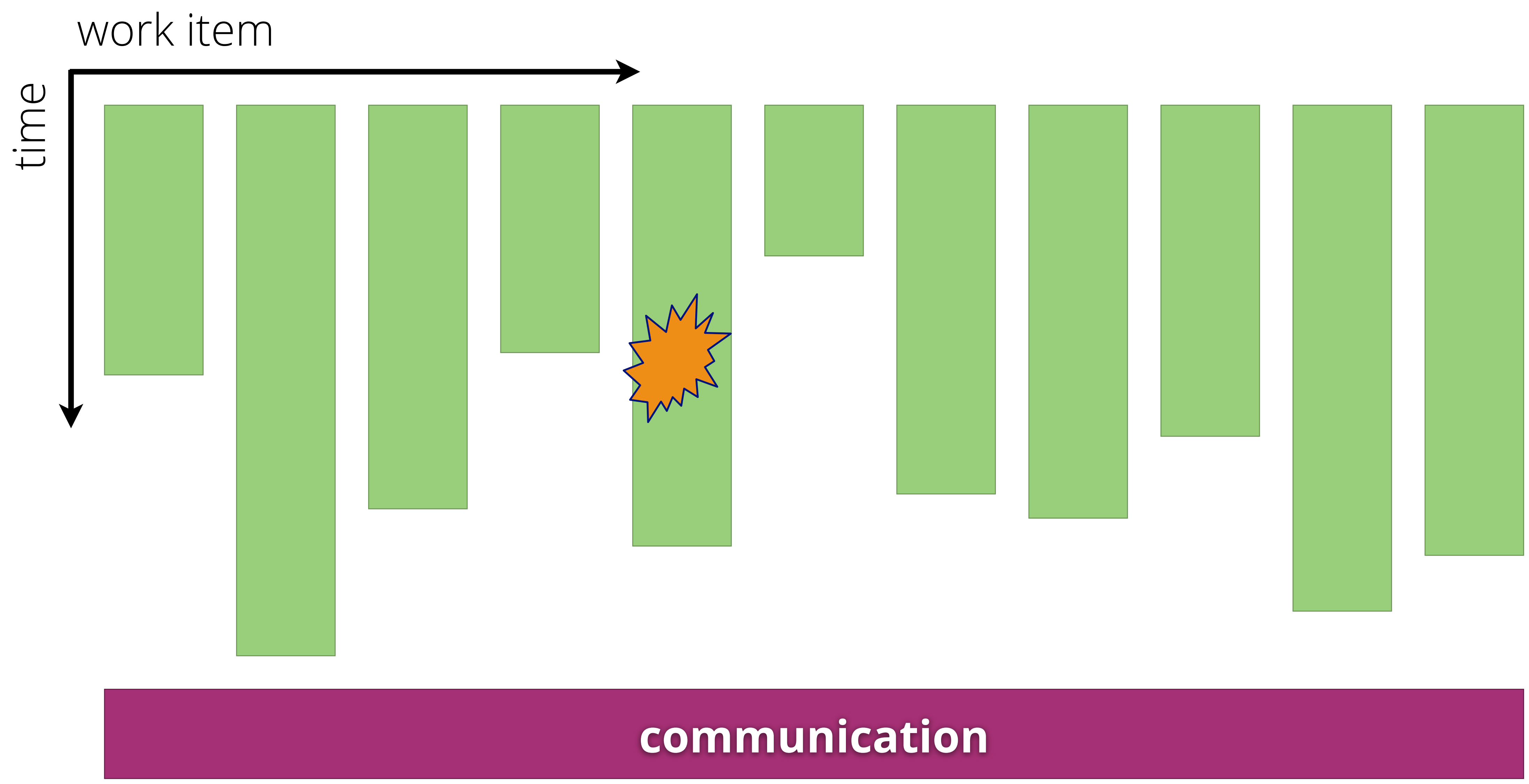
Methods to avoid:

- OS usually not directly on the critical path,
  BUT OS controls: interference via interrupts, caches, network, memory bus, (RTS techniques)
- avoid or encapsulate side activities
- small critical sections (if any)
- partition networks to isolate traffic of different applications (HW: Blue Gene)
- do not run Python scripts or printer daemons in parallel

RESEARCH TOPIC

- use small kernel to isolate

MPI & FAULT TOLERANCE



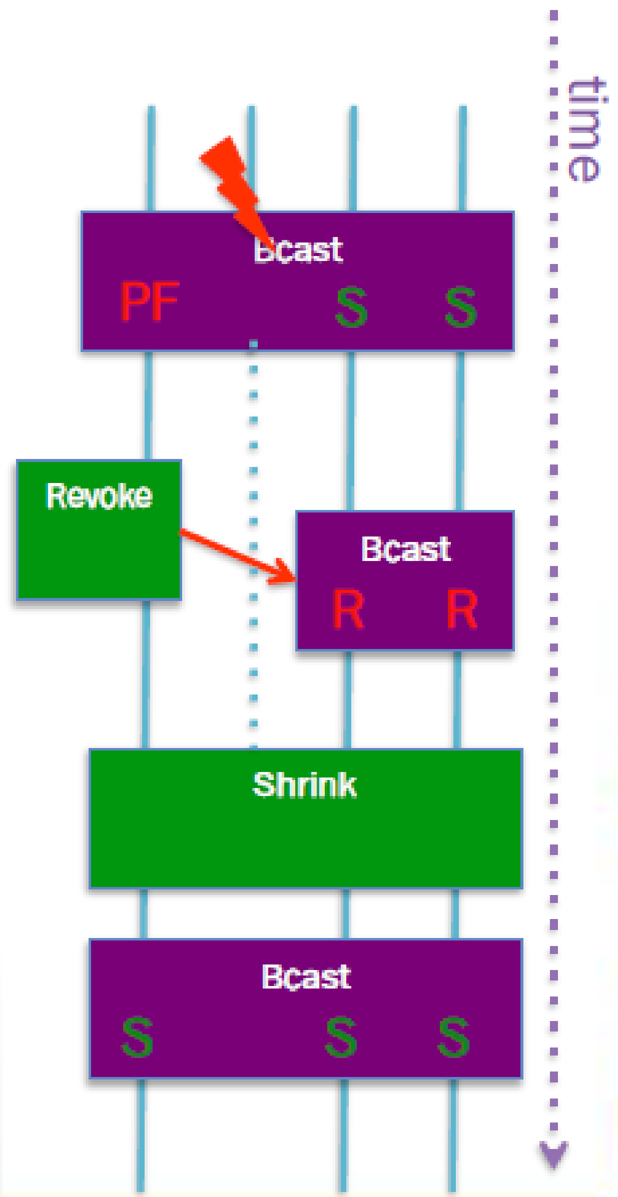
ITERATIVE ALGORITHM WITH CHECKPOINT

```
. . .

for(int t = 0; t < TIMESTEPS; t++) {
   /* ... Do work ... */

   SCR_Need_checkpoint(&flag);
   if (flag) {
     SCR_Start_checkpoint();
     SCR_Route_file(file, scr_file);
     /* save checkpoint into scr_file */
     SCR_Complete_checkpoint(1);
   }
 }
. . .
```

& RECOVER

```
MPI_Init();
SCR_Init();


if (SCR_Route_file(name, ckpt_file) ==
SCR_SUCCESS) {
  // Read checkpoint from ckpt_file
} else {
  // There is no existing checkpoint
  // Normal program startup
}
```

ULFM USER LEVEL FAULT MITIGATION



- MPI_Comm_failure_ack(comm)
  - Resumes matching for MPI_ANY_SOURCE
- MPI_Comm_failure_get_acked(comm, &group)
  - Returns to the user the group of processes acknowledged to have failed

Notification

- MPI_Comm_revoke(comm)
  - Non-collective collective, interrupts all operations on comm (future or active, at all ranks) by raising MPI_ERR_REVOKED

Propagation

- MPI_Comm_shrink(comm, &newcomm)
  - Collective, creates a new communicator without failed processes (identical at all ranks)
- MPI_Comm_agree(comm, &mask)
  - Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce), and the return core

Recovery

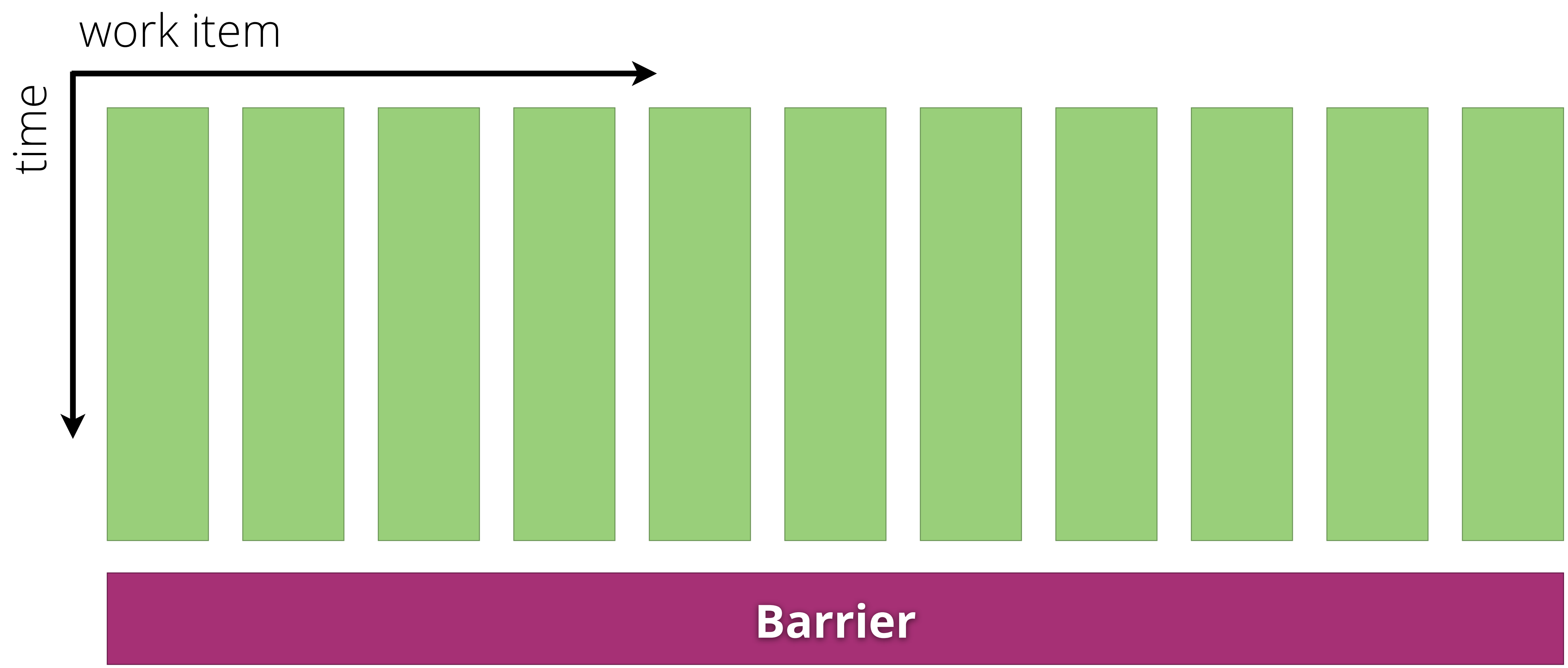
Alex Margolin

THIS LECTURE

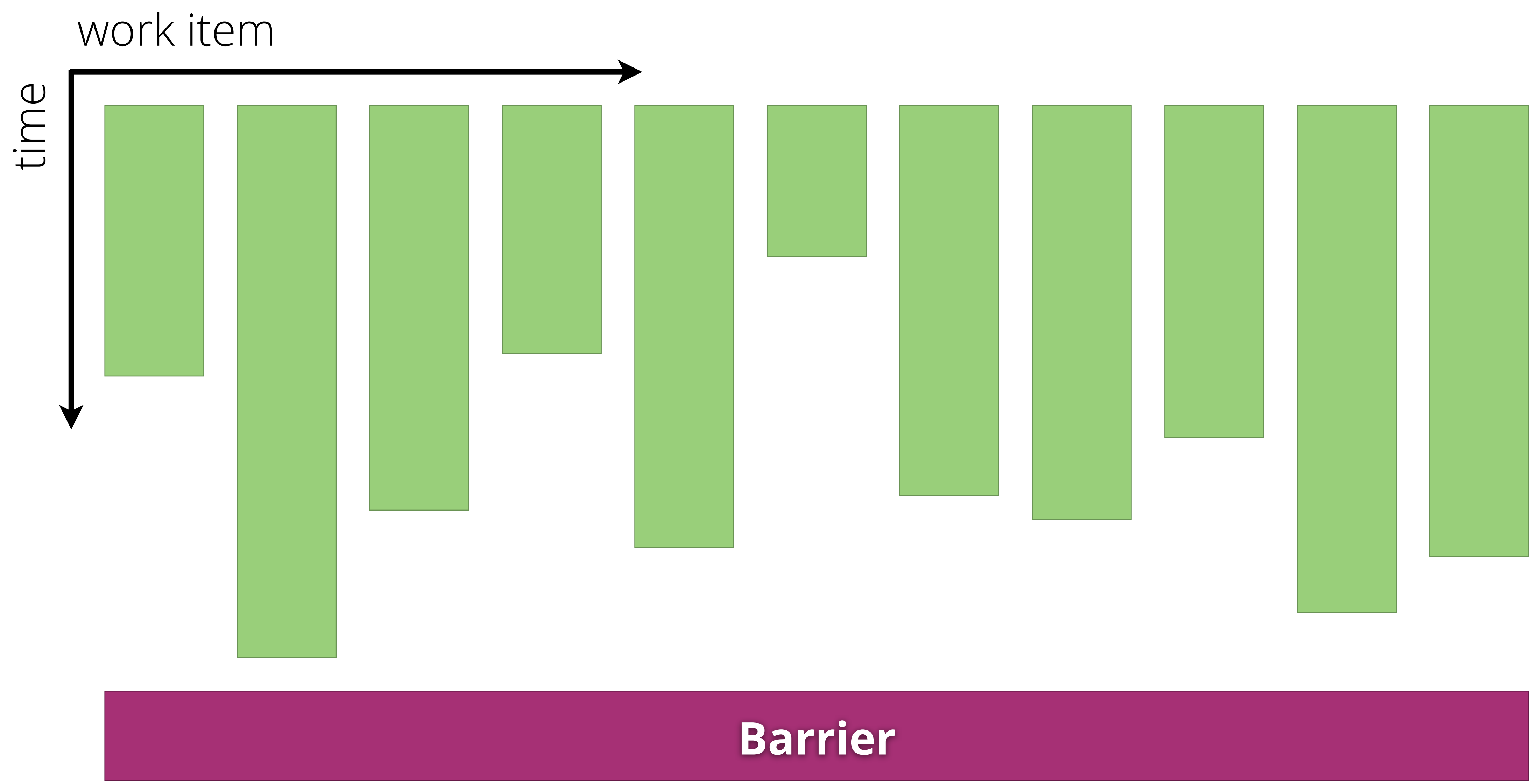
Single Admin domain,
large number of connected Compute Nodes

- MPI (Short Intro), Partitioning
- Amdahl's law & communication & jitter
  Fault Tolerance
- Load Balancing (Case Study MosiX):
  migration mechanism
  decision making (information dissemination)

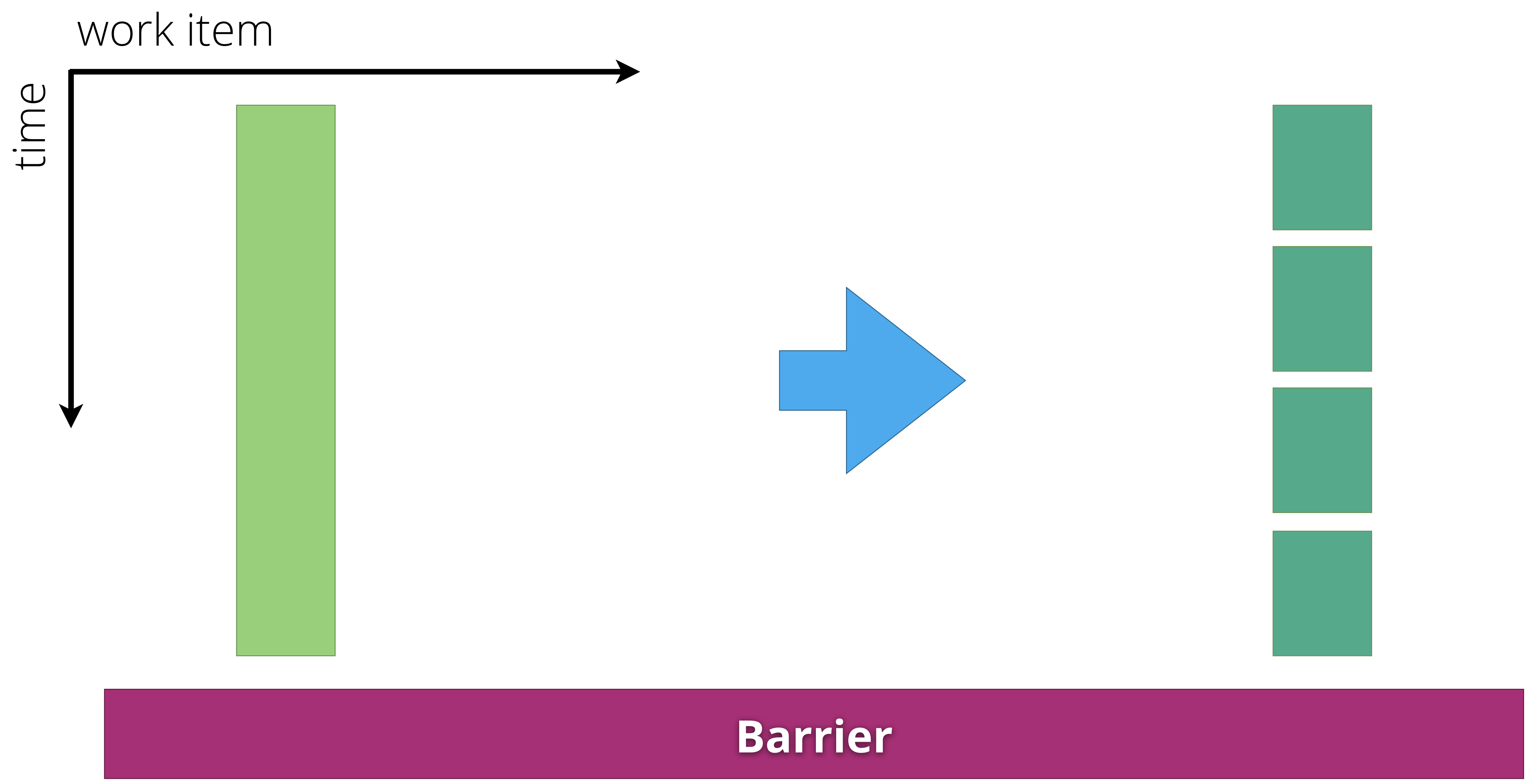
MOTIVATION FOR BALANCING



MOTIVATION FOR BALANCING

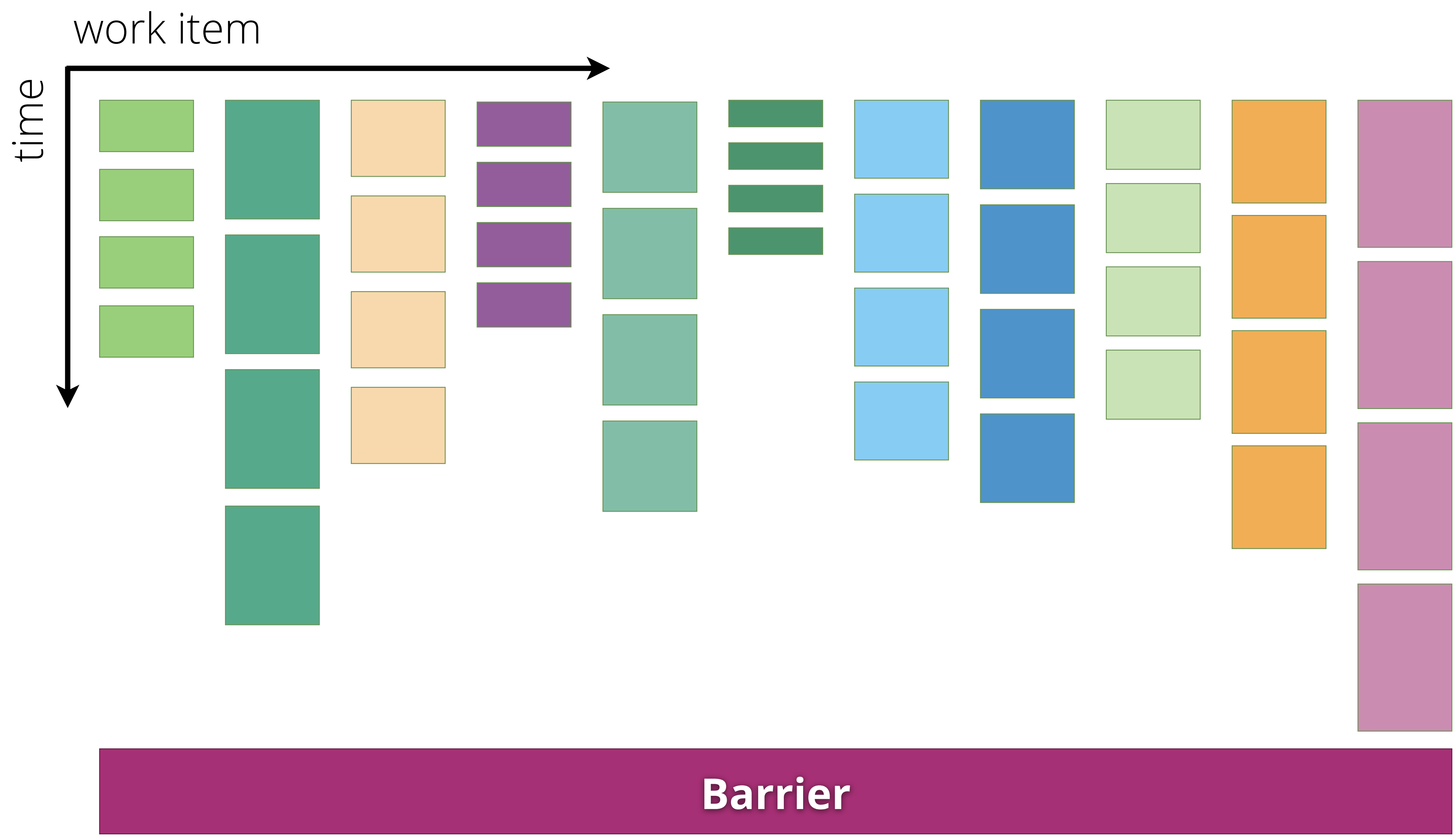


TOWARDS BALANCING



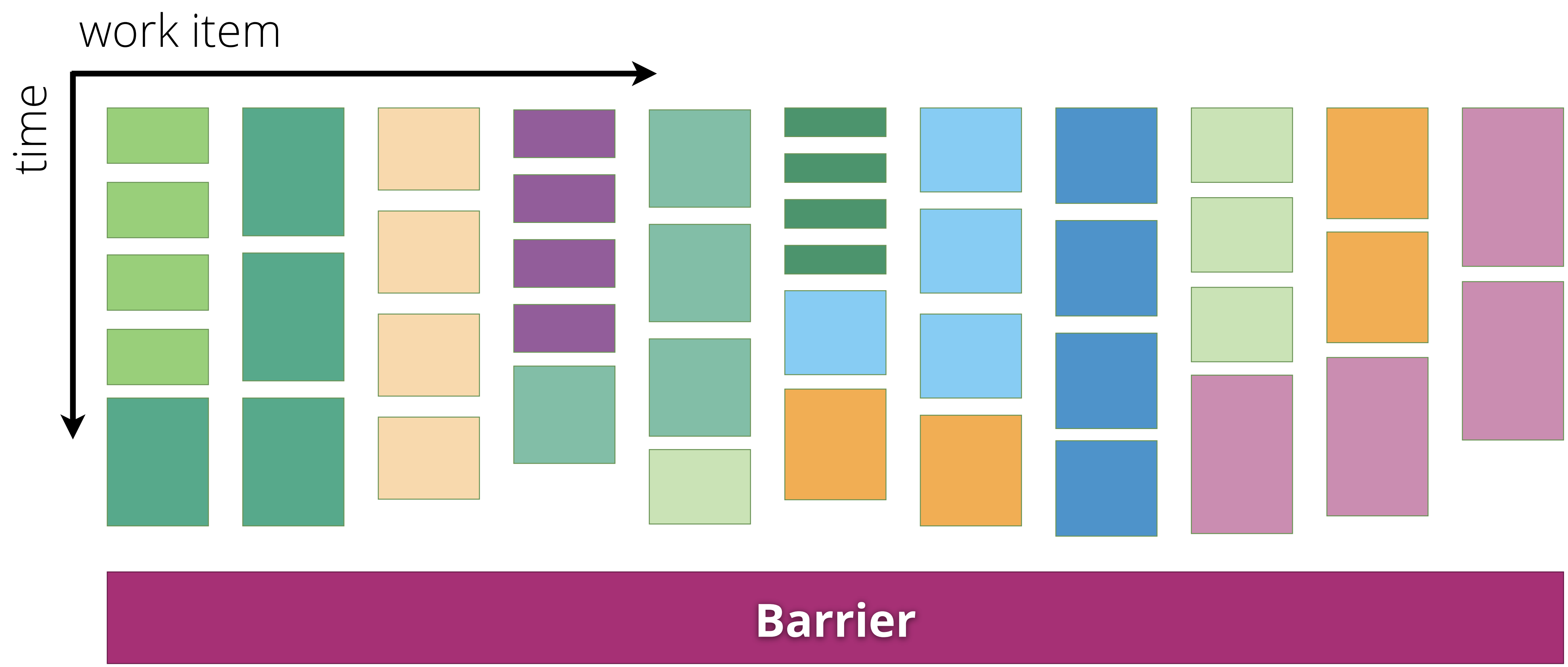
smaller pieces that can run in parallel

SPLITTING BIG JOBS



many more jobs than cores

SMALL JOBS (NO DEPS)



Execute small jobs in parallel

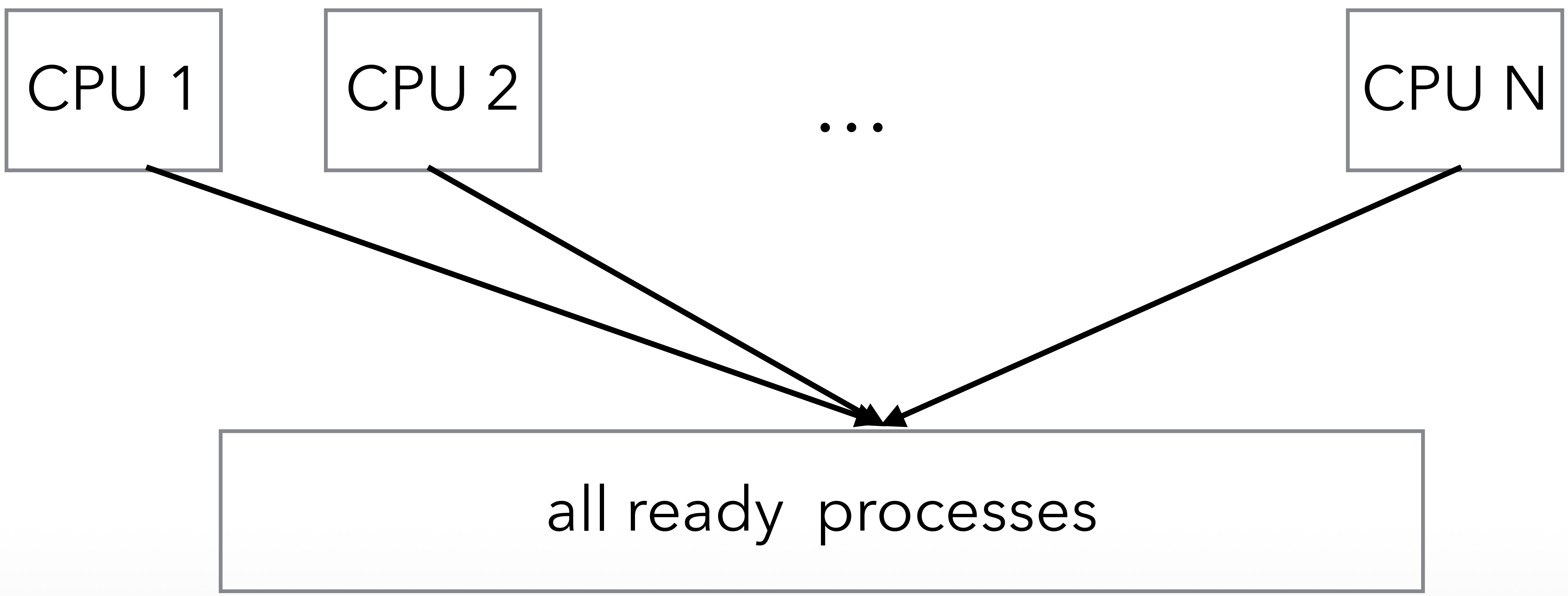
BALANCE

- if we have more pieces than CPU or are able to split into smaller pieces that can run in parallel, then use migration of load
- caveats
  - virtualization of communication needed
  - splitting per se adds cost
  - scalable decision making needed

balancing in systems architecture

- application
- run-time library (task based models)
- operating system

SCHEDULER: GLOBAL RUN QUEUE



(old) approach: global run queue

SCHEDULER: GLOBAL RUN QUEUE

- … does not scale
  - shared memory only
  - contended critical section
  - cache affinity
  - …
- ⇨ separate run queues with explicit movement of processes

OS/HW & APPLICATION

High Performance Computing

- Operating System / Hardware:
  "All" participating CPUs: active / inactive
  - Partitioning (HW)
  - Gang Scheduling (OS)
- Within Gang/Partition:
  Applications balance !!!

PROPERTIES HW PARTITIONS

- optimizes usage of network
- takes OS off critical path (busy waiting)
- best for strong scaling
- burdens application/library with balancing
- potentially wastes resources
- current state of the art in High Performance Computing (HPC)

BALANCING AT LIBRARY LEVEL

Programming Model

- many (small) decoupled work items
- overdecompose
  create more work items than active units
- run some balancing algorithm

Example: CHARM ++

BALANCING AT SYSTEM LEVEL

- create (many) more processes
- use OS information on run-time and system state to balance load
- examples:
    - run multiple applications
    - create more MPI processes than nodes

CAVEATS

added overhead

- additional communication between smaller work items (memory & cycles)
- more context switches
- OS on critical path (for example communication)

BALANCING ALGORITHMS

required:

- mechanism for migrating load
- information gathering
- decision algorithms

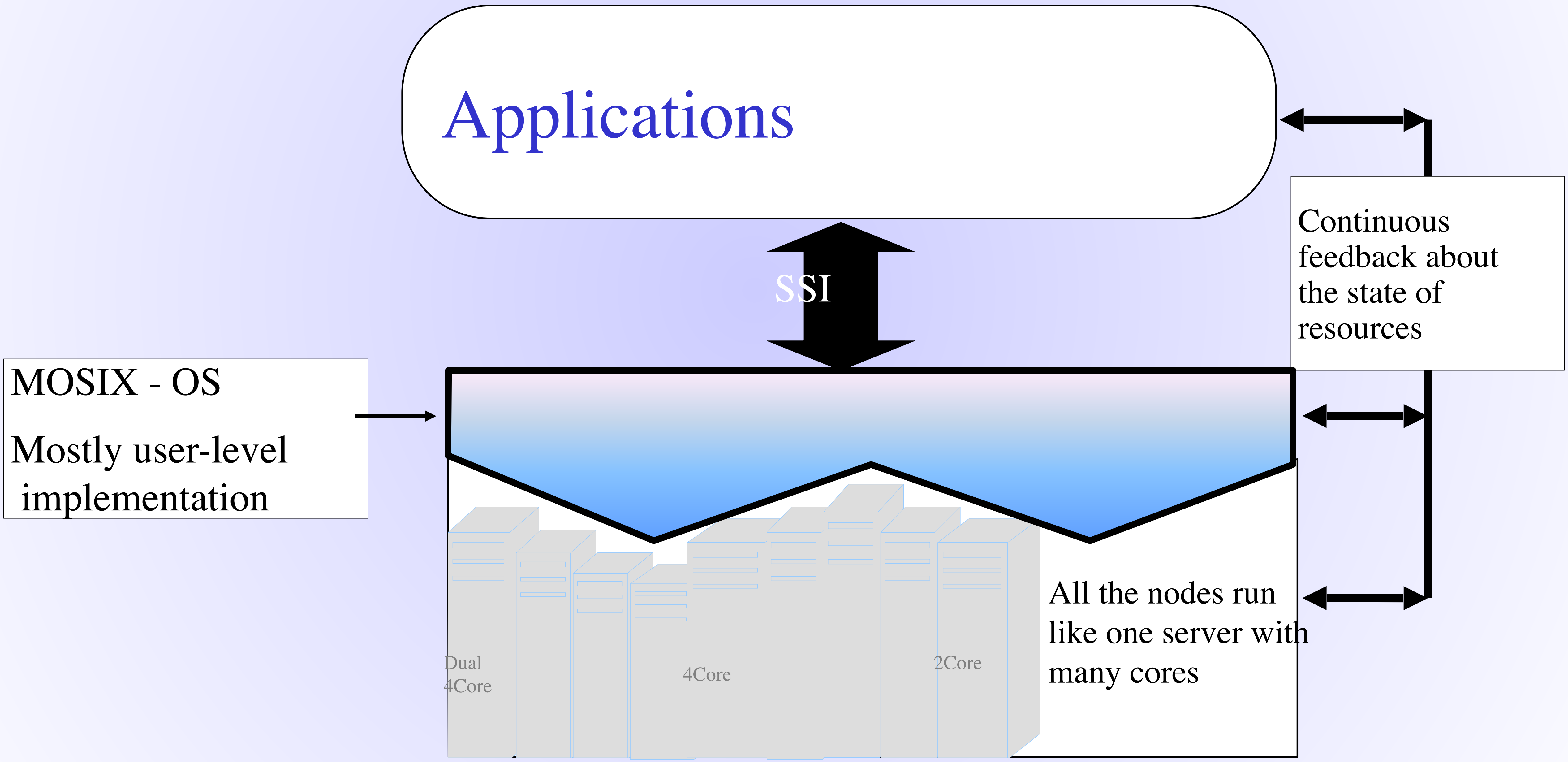
MosiX system as an example

-> Barak's slides now

Single Admin Domain,
large number of connected Compute Nodes

- MPI (Short Intro), Partitioning
- Amdahl's law & communication & jitter
  Fault Tolerance
- Load Balancing (Case Study MosiX):
  migration mechanism
  decision making (information dissemination)

MOSIX is a unifying management layer



The main software components

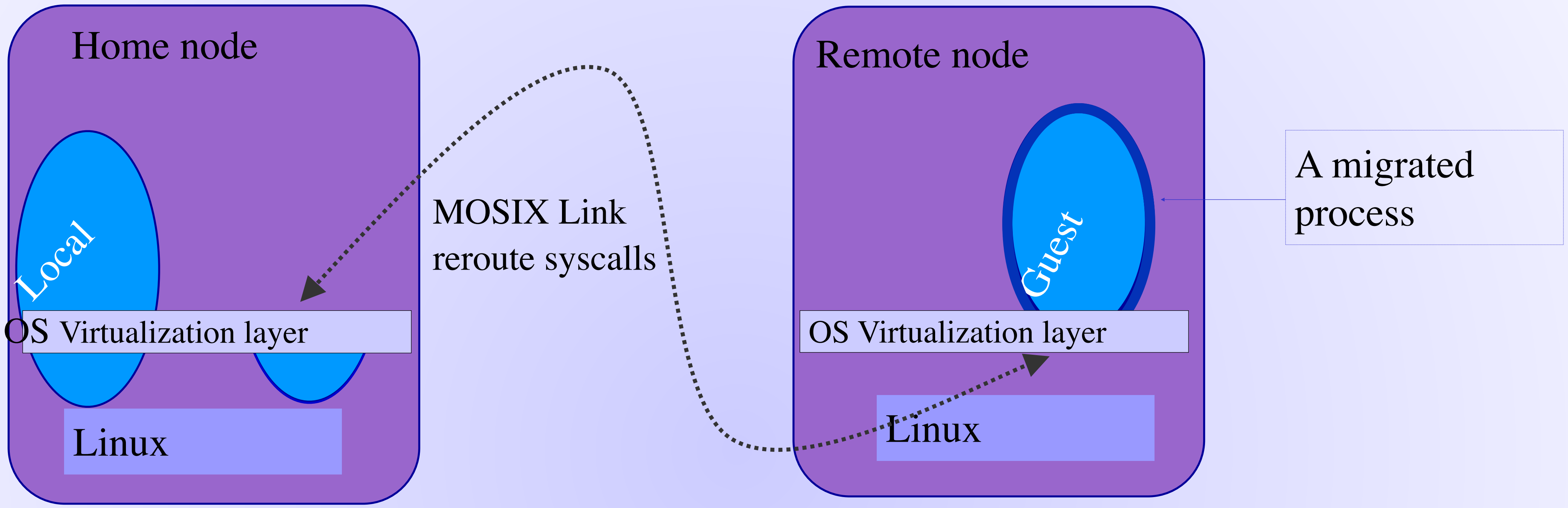
1. Preemptive process migration
   - Can migrate a running processes anytime
   - Like a course-grain context switch
     - Implication on caching, scheduling, resource utilization
2. OS virtualization layer
   - Allows a migrated process to run in remote nodes
3. On-line algorithms
   - Attempt to optimize a given goal function by process migration
     - Match between required and available resources
   - Information dissemination – based on partial knowledge

Note: features that are taken for granted in shared-memory systems, are not easy to support in a cluster

The OS virtualization layer

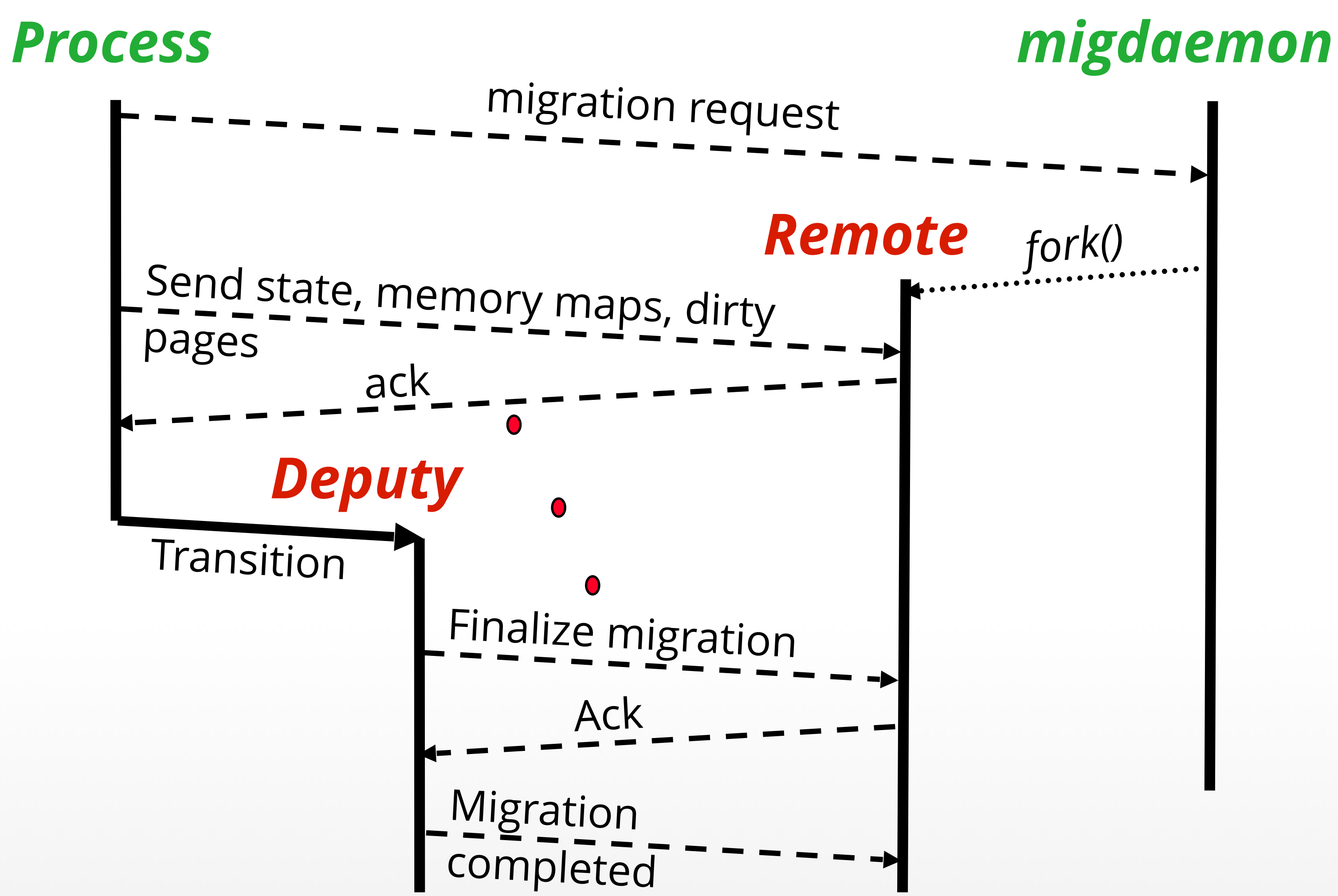
- A software layer that allows a migrated process to run in remote nodes, away from its home node
  - All system-calls are intercepted
    - Site independent sys-calls are performed locally, others are sent home
  - Migrated processes run in a sandbox
- Outcome:
  - A migrated process seems to be running in its home node
  - The cluster seems to the user as one computer
  - Run-time environment of processes are preserved - no need to change or link applications with any library, copy files or login to remote nodes
- Drawback: increased (reasonable) communication overhead

Process migration - the home node model



- **Process migration – move the process context to a remote node**
  - **System context stay at "home" thus providing a single point of entry**
- **Process partition preserves the user's run-time environment**
  - **Users need not care where their process are running**

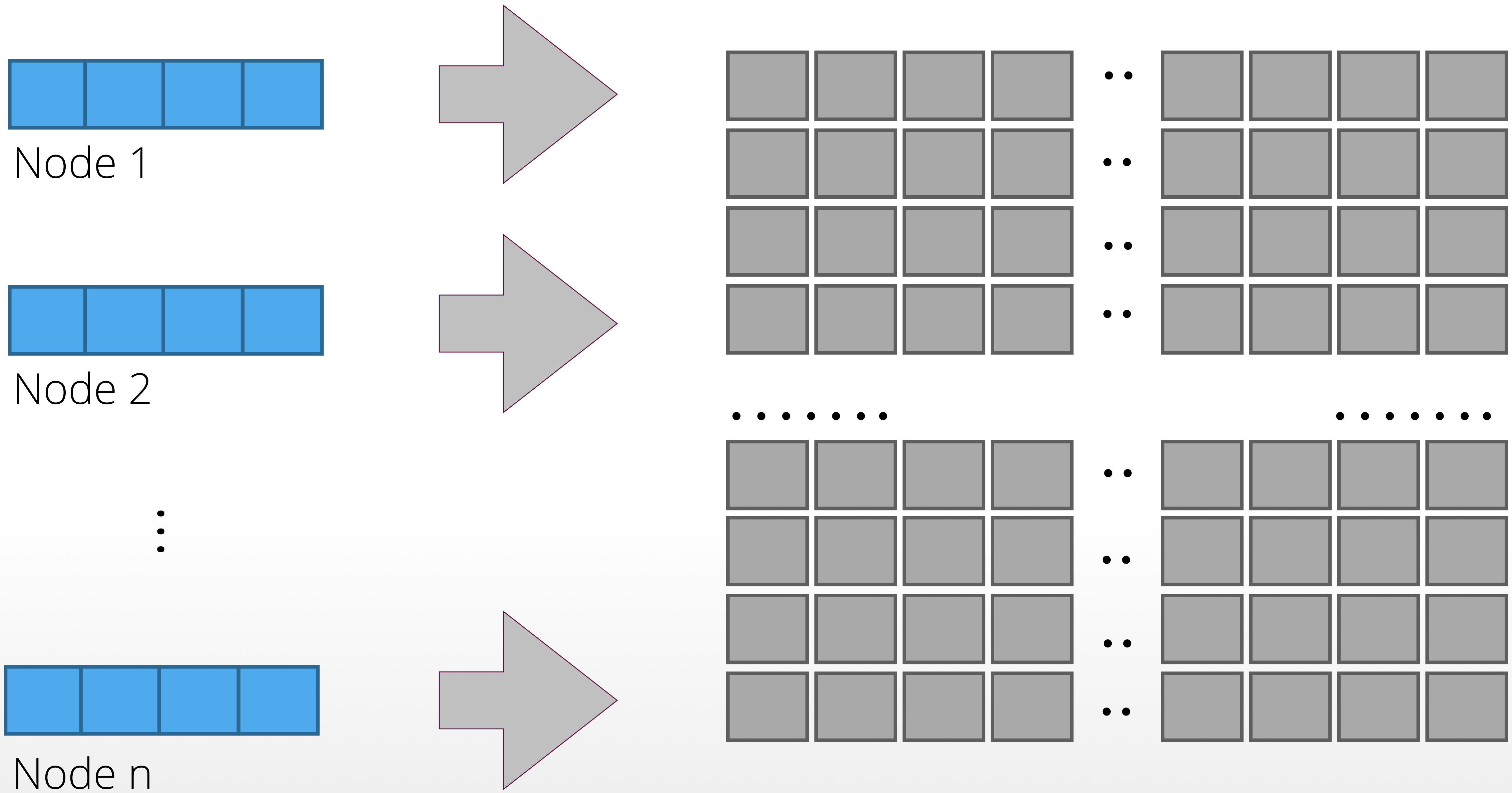
PROCESS MIGRATION IN MOSIX



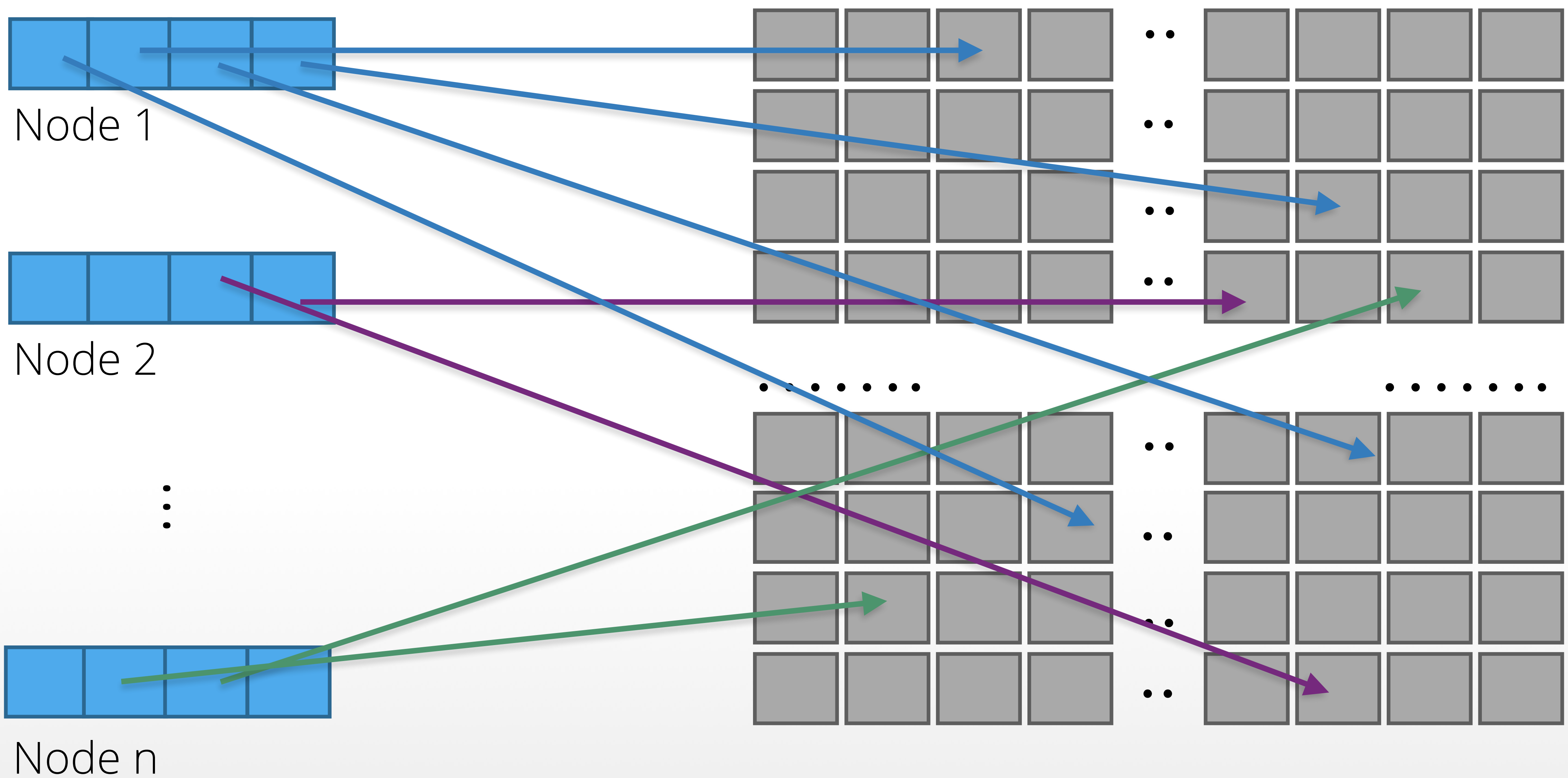
Distributed bulletin board

- **An n node cluster/Cloud system**
  - **Decentralized control**
  - **Nodes can fail at any time**
- ***Each node maintains a data structure (vector) with an entry about selected (or all) the nodes***
- **Each entry contains:**
  - **State of the resources of the corresponding node, e.g. load**
  - **Age of the information (tune to the local clock)**
- **The vector is used by each node as a distributed bulletin board**
  - **Provides information about allocation of new processes**

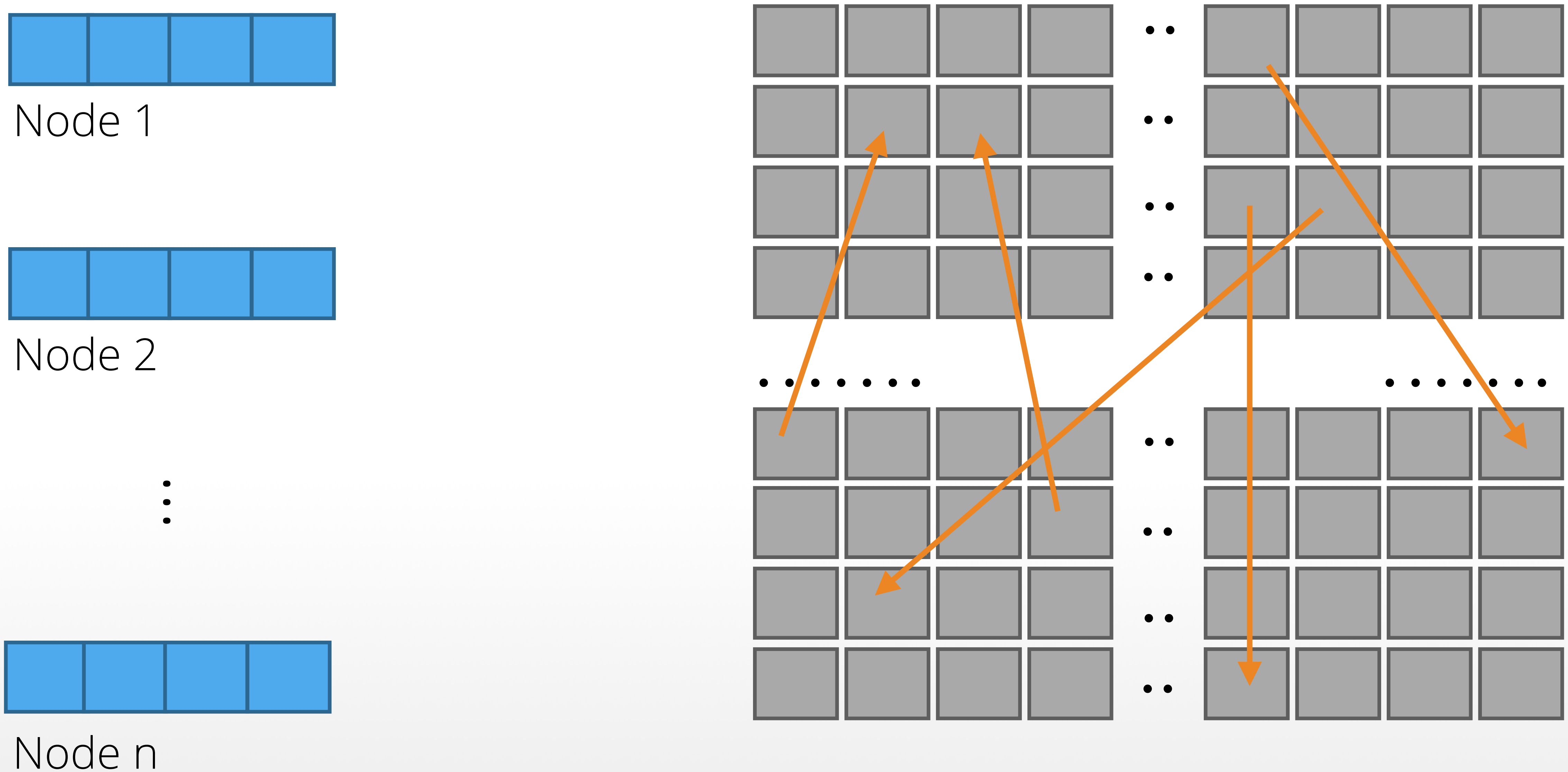
DECENTRALIZED GLOBAL STATE



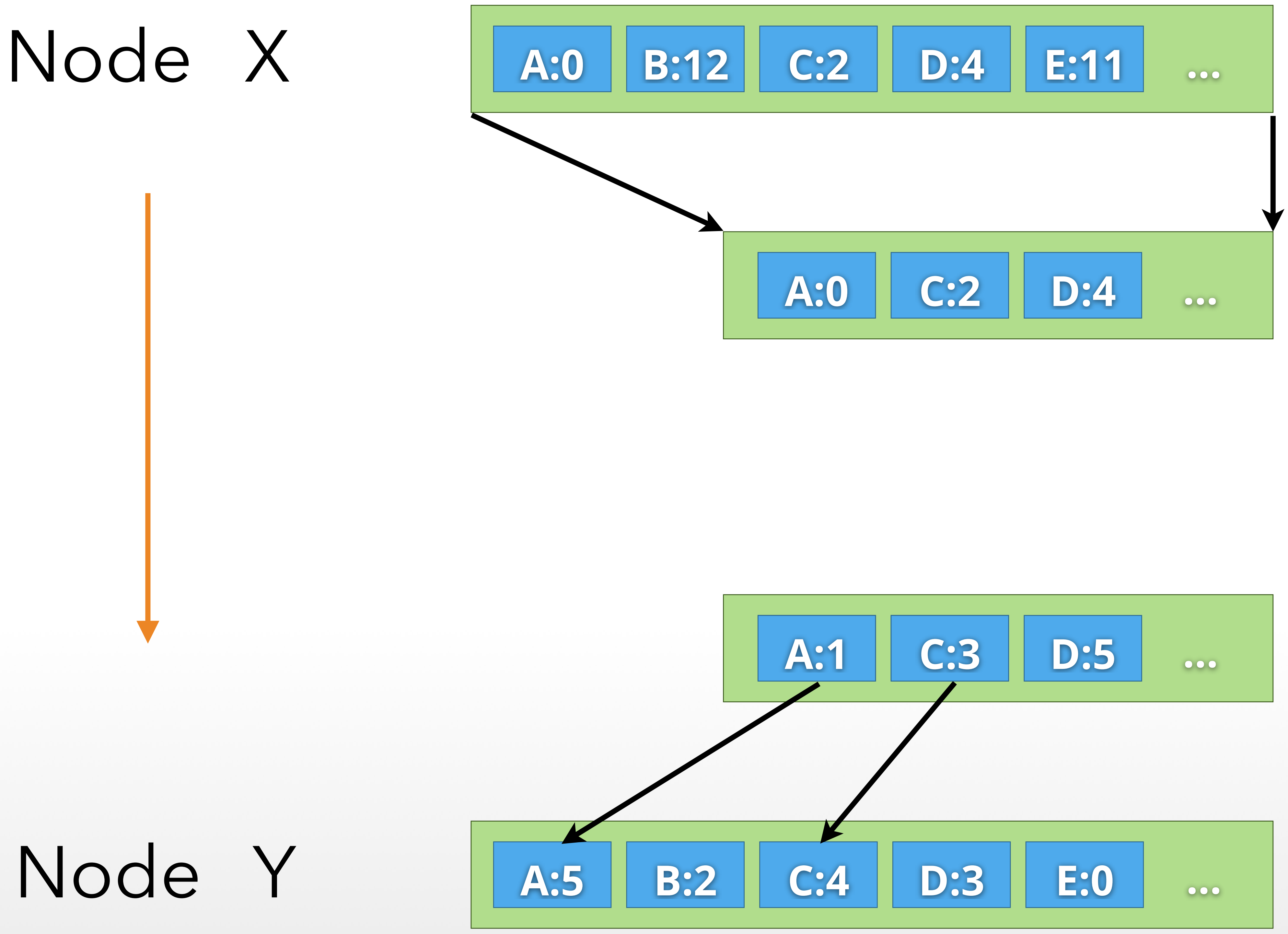
DECENTRALIZED GLOBAL STATE



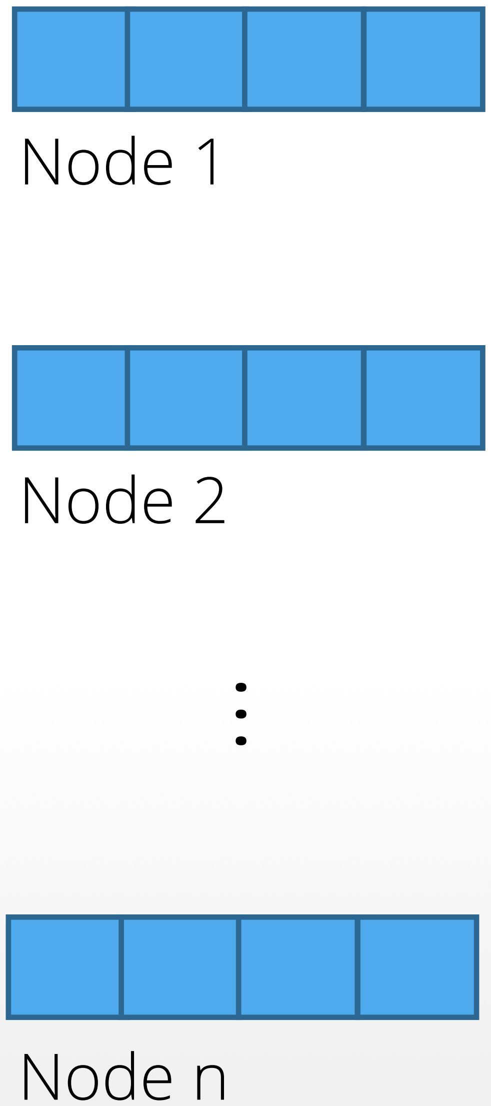
GOSSIP



GOSSIP



WWW



When

- **M:** load diff erence discovered
- anomaly discovered
- anticipated

Where

- **M:** memory, cycles, comm
- consider topology
- application knowledge

Which

- **M:** past predicts future
- application knowledge

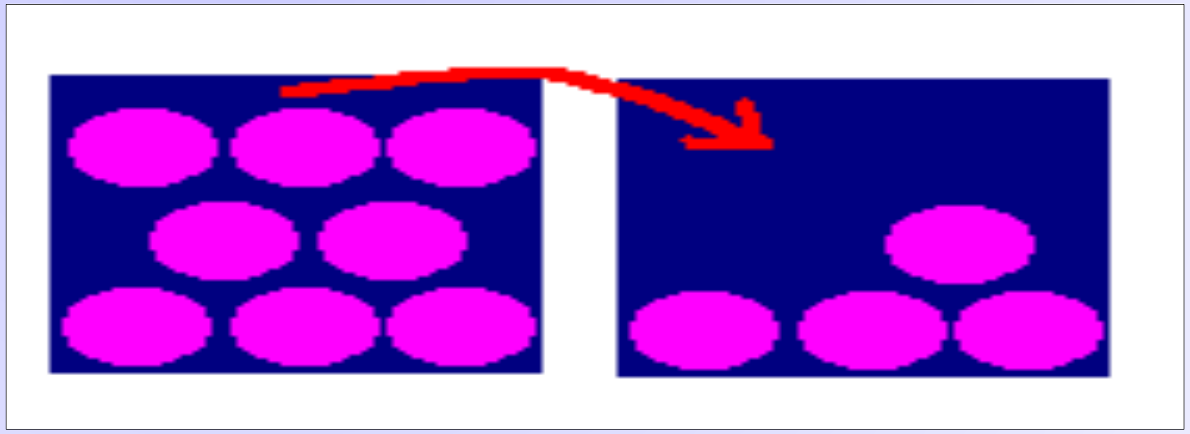
Load balancing algorithms

- **When** - Load difference between a pair of nodes is above a threshold value
- **Which** - Oldest process (assumes past-repeat)
- **Where** - To the known node with the lowest load
- Many other heuristics

- Performance: our online algorithm is only ~2% slower than the optimal algorithm (which has complete information about all the processes)

Memory ushering

- **Heuristics: initiate process migration from a node with no free memory to a node with available free memory**
- **Useful: when non-uniform memory usage (many users) or nodes with different memory sizes**
- **Overrides load-balancing**



- Recall: **placement problem is NP-hard**

Memory ushering algorithm

- **When** - free memory drops below a threshold
- **Where** - the node with the lowest load, to avoid unnecessary follow-up migrations
- **Which** - smallest process that brings node under threshold
  - To reduce the communication overhead

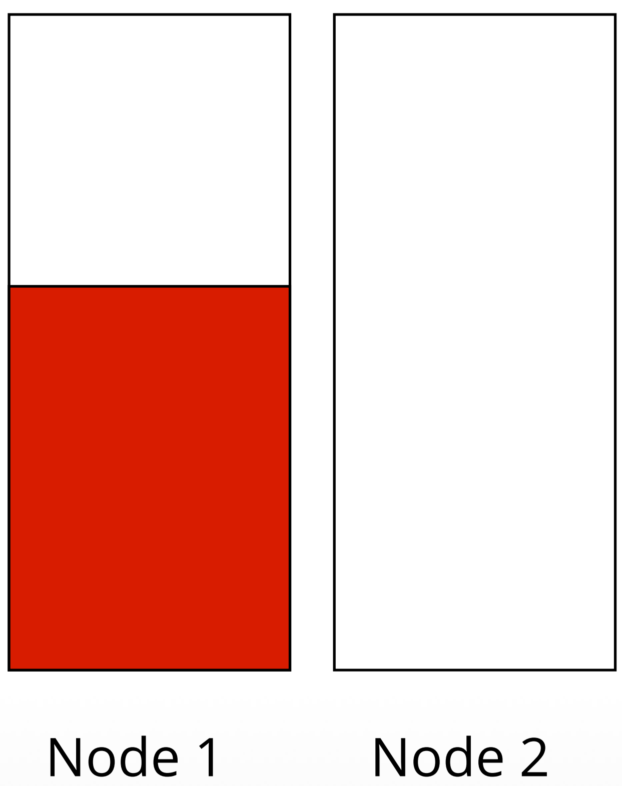
PRECEDENCE

- memory
- cpu load
- IPC

SOME PRACTICAL PROBLEMS

- flooding
  all processes jump to one new empty node
  => decide immediately before migration commitment
  extra communication, piggy packed
- ping pong
  if thresholds are very close, processes moved back and forth
  => tell a little higher load than real

PING PONG



One process two nodes

Scenario:
compare load on nodes 1 and 2
node 1 moves process to node 2

Solutions:
add one + little bit to load
average over time

Solves short peaks problem as well (short cron processes)

LESSONS

- execution/communication time jitter matters (Amdahl)
- HPC approaches: partition ./. balance
- dynamic balance components:
  migration mechanism,
  information bulletin,
  decision: which, when, where