TECHNISCHE UNIVERSITÄT DRESDEN

LOAD BALANCING

DISTRIBUTED OPERATING SYSTEMS, SCALABILITY, SS 2015

Hermann Härtig

ISSUES

- starting points
  - independent Unix processes and
  - block synchronous execution
- who does it
- load migration mechanism (MosiX)
- management algorithms (MosiX)
  - information dissemination
  - decision algorithms

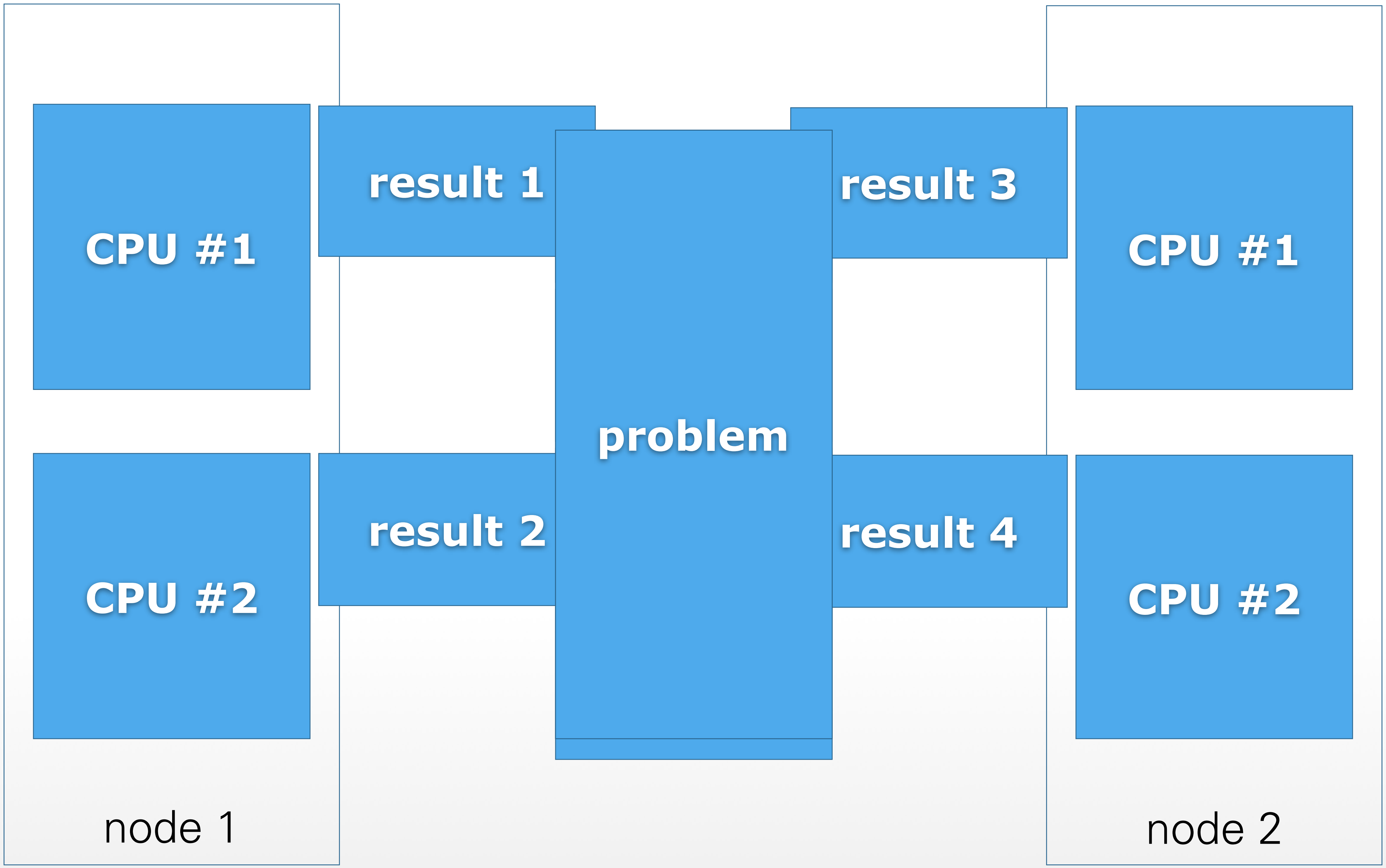
EXTREME STARTING POINTS

- independent OS processes
- block synchronous execution (HPC)
  - sequence: compute - communicate
  - all processes wait for all other processes
  - often: message passing
    for example Message Passing Library (MPI)

SIMPLE PROGRAM MULTIPLE DATA

- all processes execute same program
- while (true)
  { work; exchange data (barrier)}
- common in
  High Performance Computing:
  Message Passing Interface (MPI)
  library

DIVIDE AND CONQUER



MPI BRIEF OVERVIEW

- Library for message-oriented parallel programming
- Programming model:
  - Multiple instances of same program
  - Independent calculation
  - Communication, synchronization

MPI STARTUP & TEARDOWN

- MPI program is started on all processors
- `MPI_Init()`, `MPI_Finalize()`
- Communicators (e.g., MPI_COMM_WORLD)
  - `MPI_Comm_size()`
  - `MPI_Comm_rank()`: "Rank" of process within this set
  - Typed messages
- Dynamically create and spread processes using `MPI_Spawn()` (since MPI-2)

MPI EXECUTION

- Communication
  - Point-to-point
  - Collectives
- Synchronization
  - Test
  - Wait
  - Barrier

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

BLOCK AND SYNC

	blocking call	non-blocking call
synchronous communication	returns when message has been delivered	returns immediately, following test/wait checks for delivery
asynchronous communication	returns when send buffer can be reused	returns immediately, following test/wait checks for send buffer

EXAMPLE

```
int rank, total;
MPI_Init();
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &total);

MPI_Bcast(...);
/* work on own part, determined by rank */

if (id == 0) {
  for (int rr = 1; rr < total; ++rr)
    MPI_Recv(...);
  /* Generate final result */
} else {
    MPI_Send(...);
}
MPI_Finalize();
```

AMDAHLS' LAW

interpretation for parallel systems:

- P: section that can be parallelized
- 1-P: serial section
- N: number of CPUs

$$\text{Speedup}(P,N) = \frac{1}{\left(1 - P + \frac{P}{N}\right)}$$

AMDAHL'S LAW

Serial section:
communicate, longest sequential section

	Parallel,	Serial,	possible speedup:	
■	1ms,	100 µs:	1/0.1	→ 10
■	1ms,	1 µs:	1/0.001	→ 1000
■	10 µs,	1 µs:	0.01/0.001	→ 10
■	…			

WEAK VS. STRONG SCALING

Strong:

- accelerate same problem size

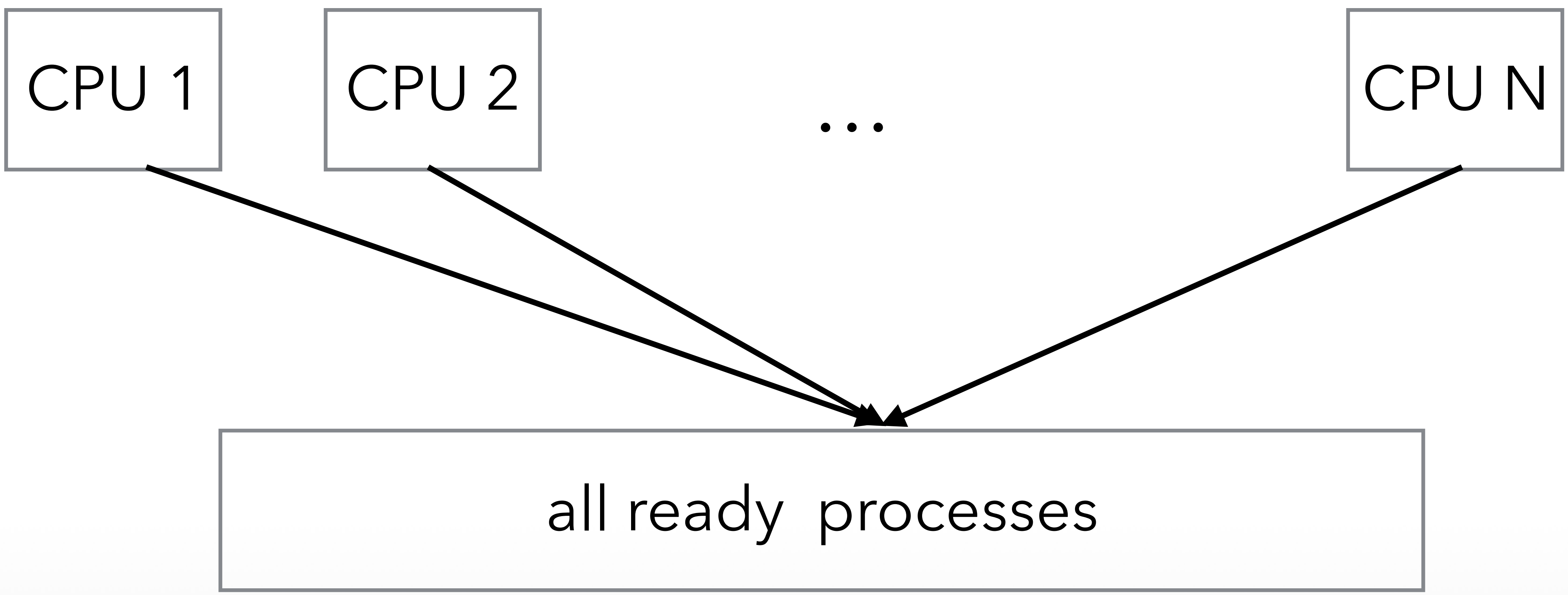
Weak:

- extend to larger problem size

WHO DOES IT

- application
- run-time library
- operating system

SCHEDULER: GLOBAL RUN QUEUE



immediate approach: global run queue

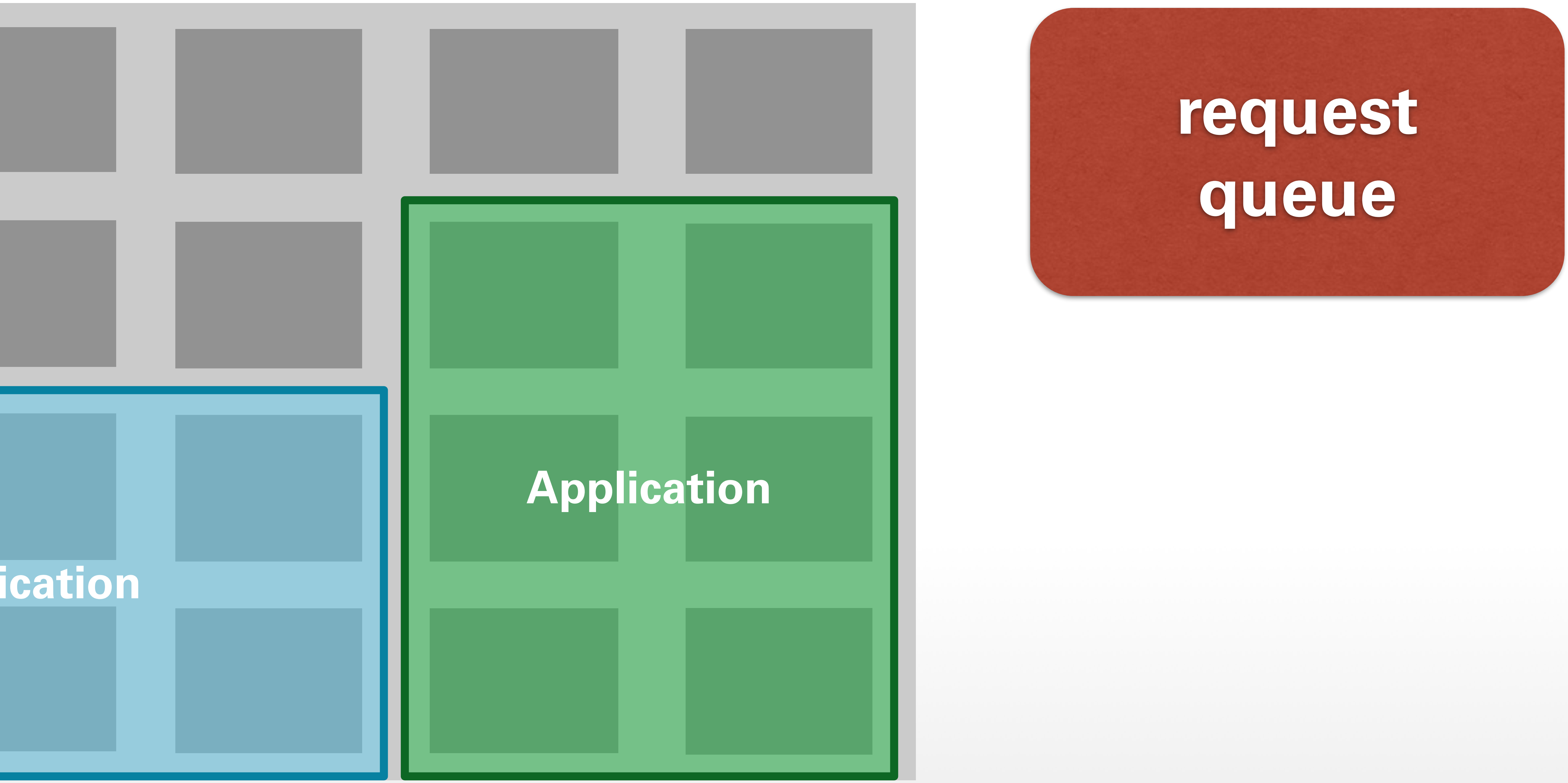
SCHEDULER: GLOBAL RUN QUEUE

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

OS/HW & APPLICATION

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

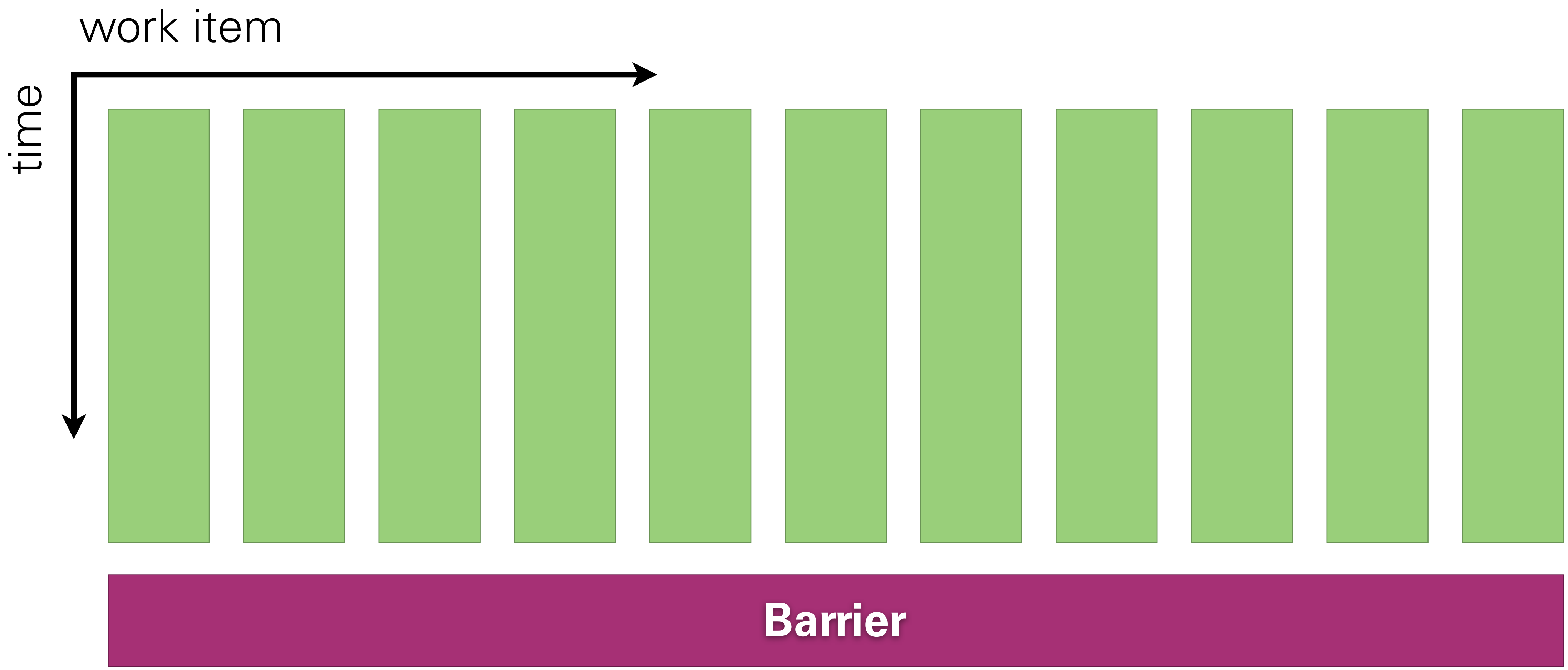
HW PARTITIONS & ENTRY QUEUE



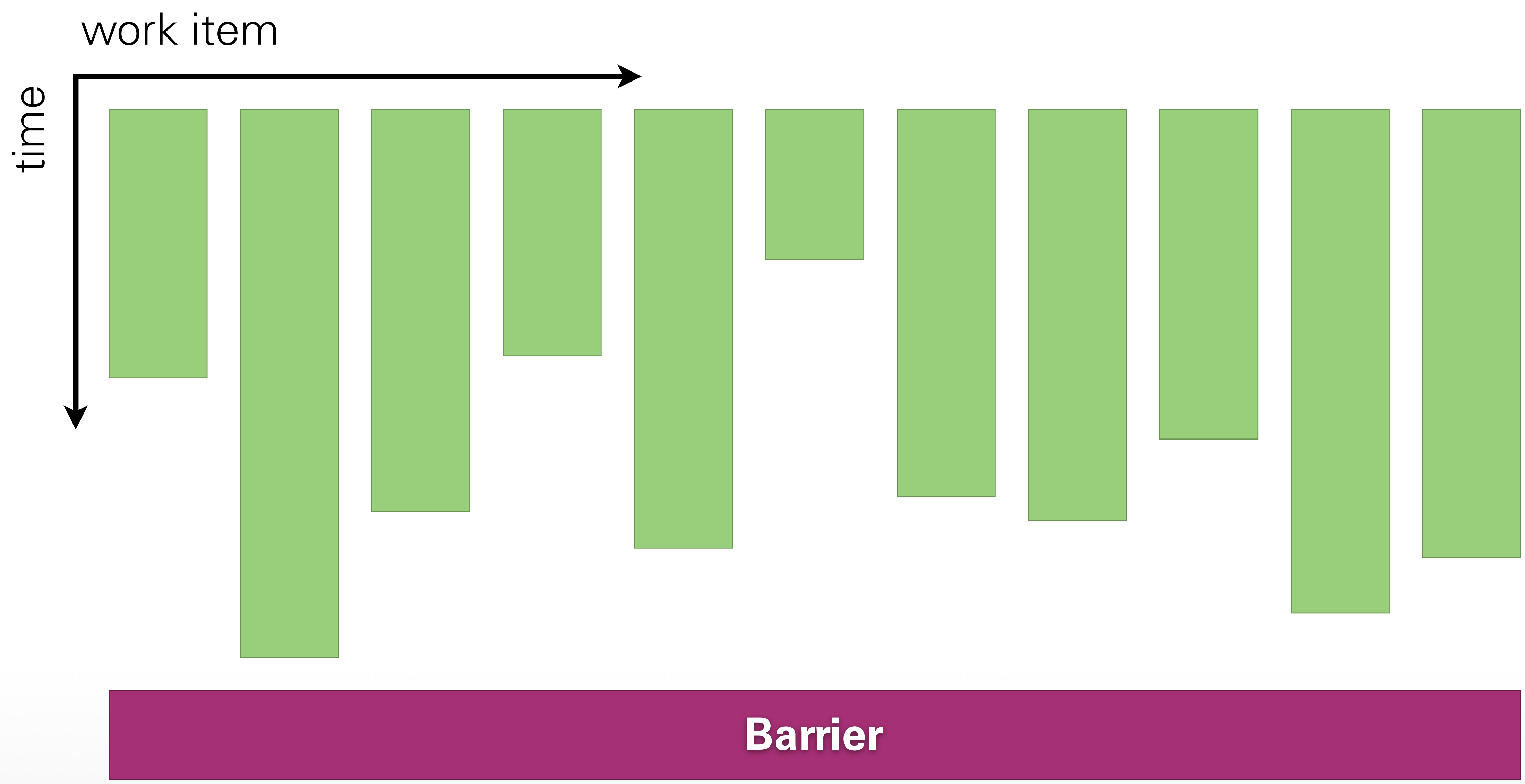
HW PARTITIONS

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

TOWARDS OS/RT BALANCING



EXECUTION TIME JITTER



SOURCES FOR EXECUTION JITTER

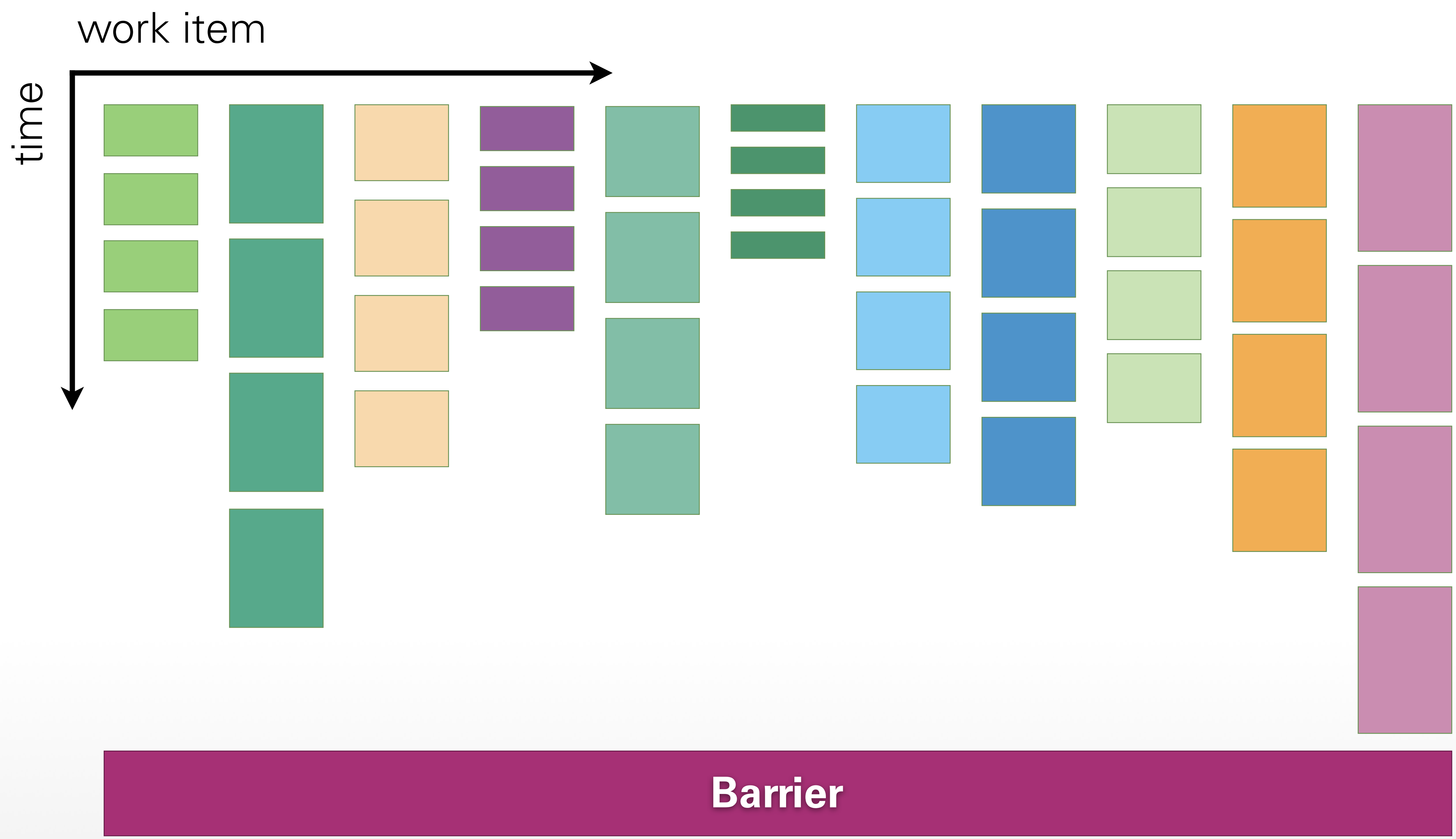
- Hardware ???
- Application
- Operating system “noise”

OPERATING SYSTEM "NOISE"

Use common sense to avoid:

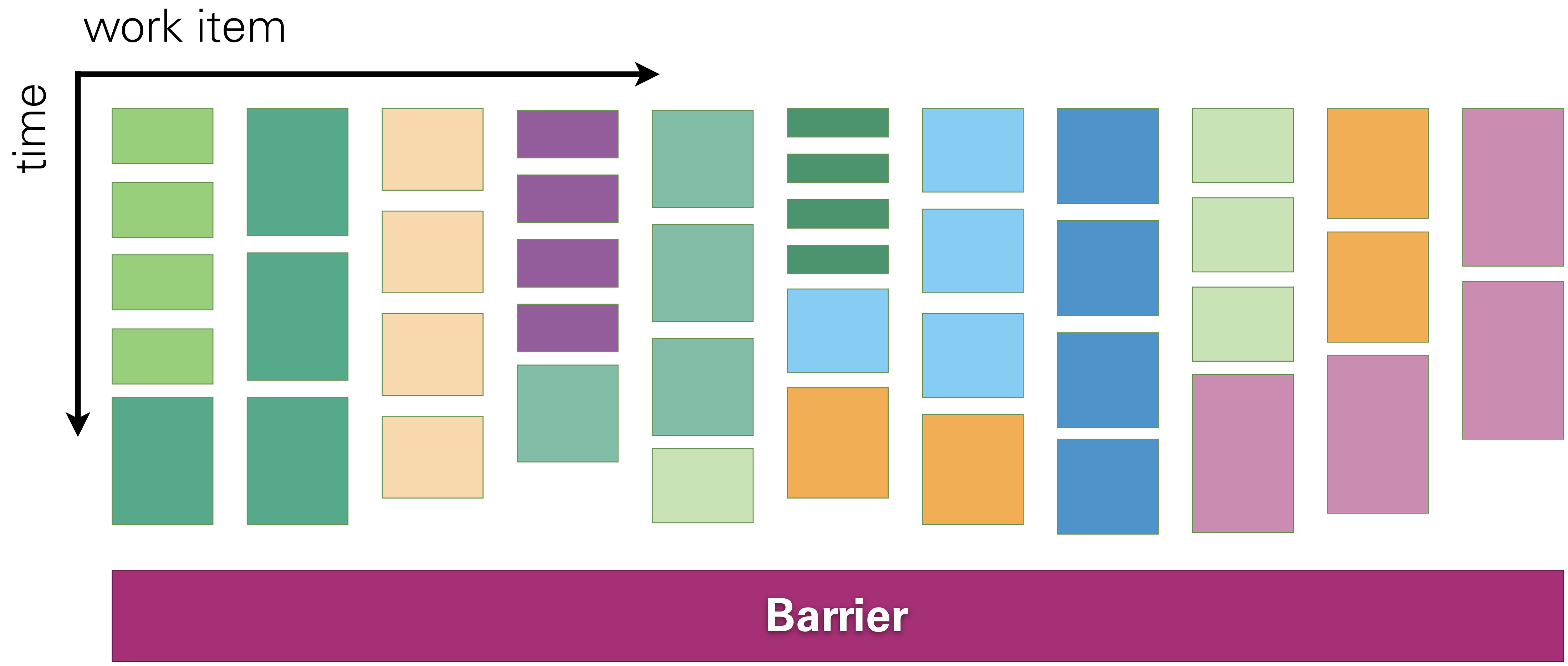
- OS usually not directly on the critical path, BUT OS controls: interference via interrupts, caches, network, memory bus, (RTS techniques)
- avoid or encapsulate side activities
- small critical sections (if any)
- partition networks to isolate traffic of different applications (HW: Blue Gene)
- do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

SPLITTING BIG JOBS



overdecomposition & “oversubscription”

SMALL JOBS (NO DEPS)



Execute small jobs in parallel (if possible)

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:
  - create more MPI processes than nodes
  - run multiple applications

CAVEATS

added overhead

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

BALANCING ALGORITHMS

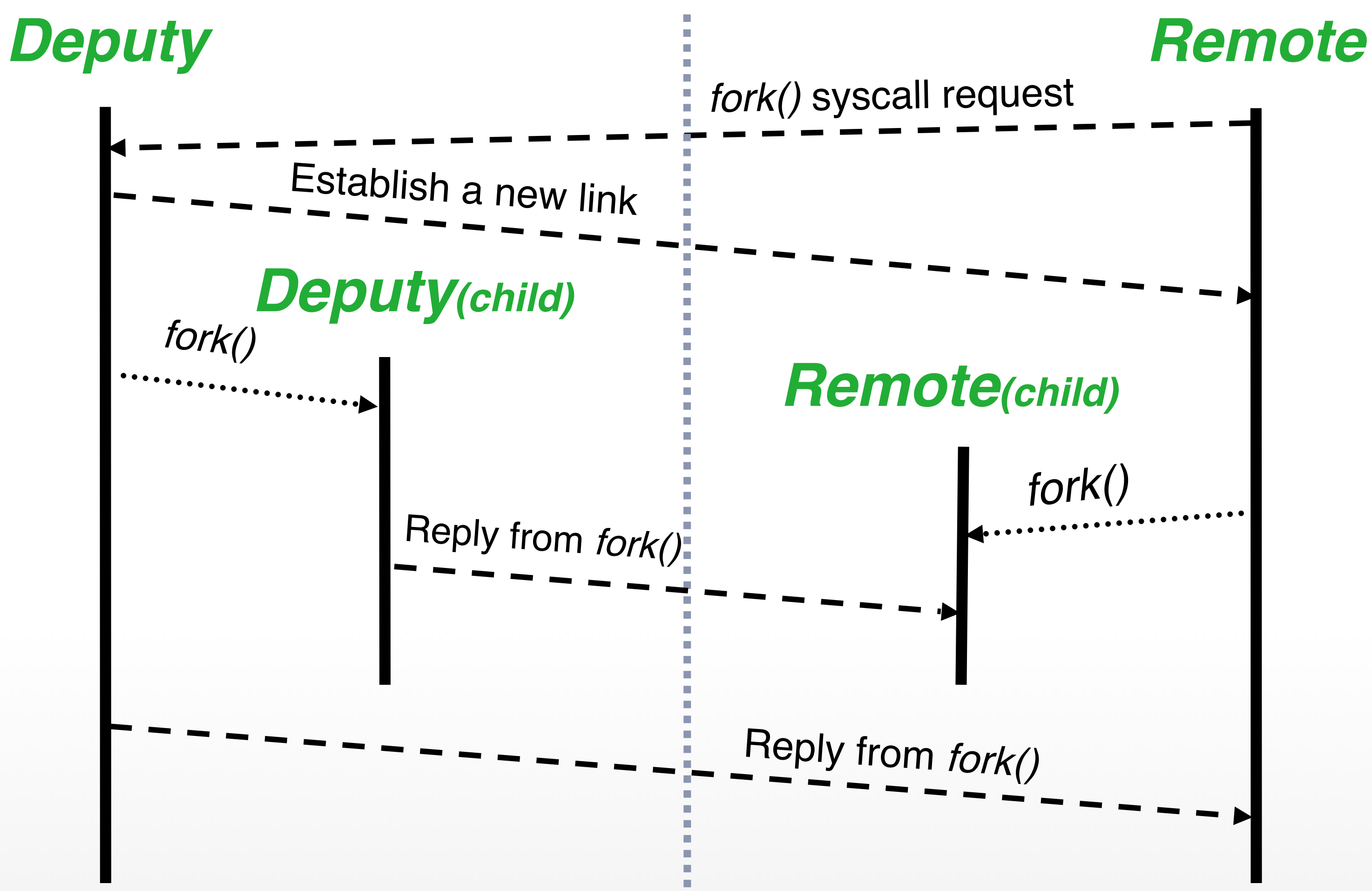
required:

- mechanism for migrating load
- information gathering
- decisions

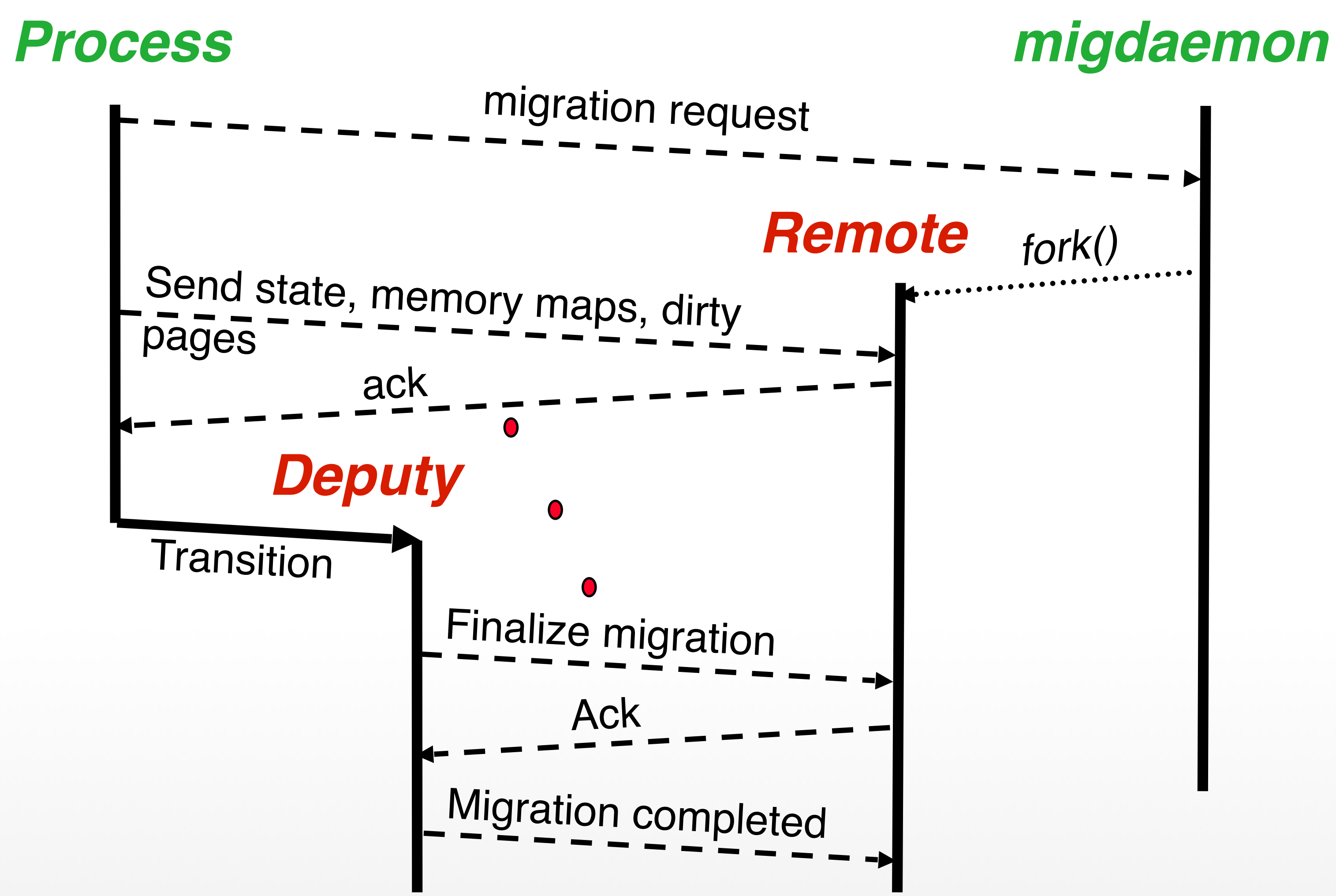
MosiX system as an example

-> Barak’s slides now

FORK IN MOSIX



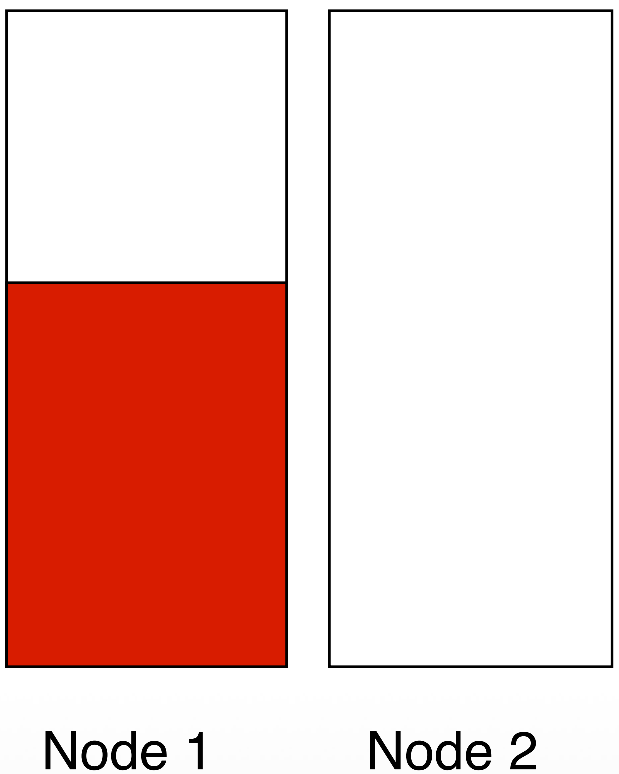
PROCESS MIGRATION IN MOSIX



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 time jitter matters (Amdahl)
- approaches: partition ./. balance
- dynamic balance components:
  migration of load, information, decision
  who, when, where