

---

# Parallel Processing

Winter Term 2025/26

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 10, 2025

---

# Parallel Processing

Winter Term 2025/26

## 2 Basics of Parallel Processing



### Contents

- ➔ Motivation
- ➔ Parallelism
- ➔ Parallelism and data dependences
- ➔ Parallel computer architectures
- ➔ Parallel programming models
- ➔ Organisation forms for parallel programs
- ➔ A design process for parallel programs
- ➔ Performance considerations

### Literature

- ➔ Ungerer
- ➔ Grama, Gupta, Karypis, Kumar

### What is parallelism?

- ➔ In general:
  - ➔ executing more than one action at a time
- ➔ Specifically with respect to execution of programs:
  - ➔ at some point in time
    - ➔ more than one statement is executed
    - and / or
    - ➔ more than one pair of operands is processed
- ➔ Goal: faster solution of the task to be processed
- ➔ Problems: subdivision of the task, coordination overhead



### Why parallel processing?

- ➔ Applications with high computing demands, esp. simulations
  - ➔ climate, earthquakes, superconductivity, molecular design, ...
- ➔ Example: protein folding
  - ➔ 3D structure, function of proteins (Alzheimer, BSE, ...)
  - ➔  $1,5 \cdot 10^{11}$  floating point operations (Flop) / time step
  - ➔ time step:  $5 \cdot 10^{-15} \text{ s}$
  - ➔ to simulate:  $10^{-3} \text{ s}$
  - ➔  $3 \cdot 10^{22}$  Flop / simulation
  - ➔  $\Rightarrow$  1 year computation time on a PFlop/s computer!
- ➔ For comparison: world's currently fastest computer: El Capitan (LLNL, USA), 1742 PFlop/s (with 11039616 CPU cores!)



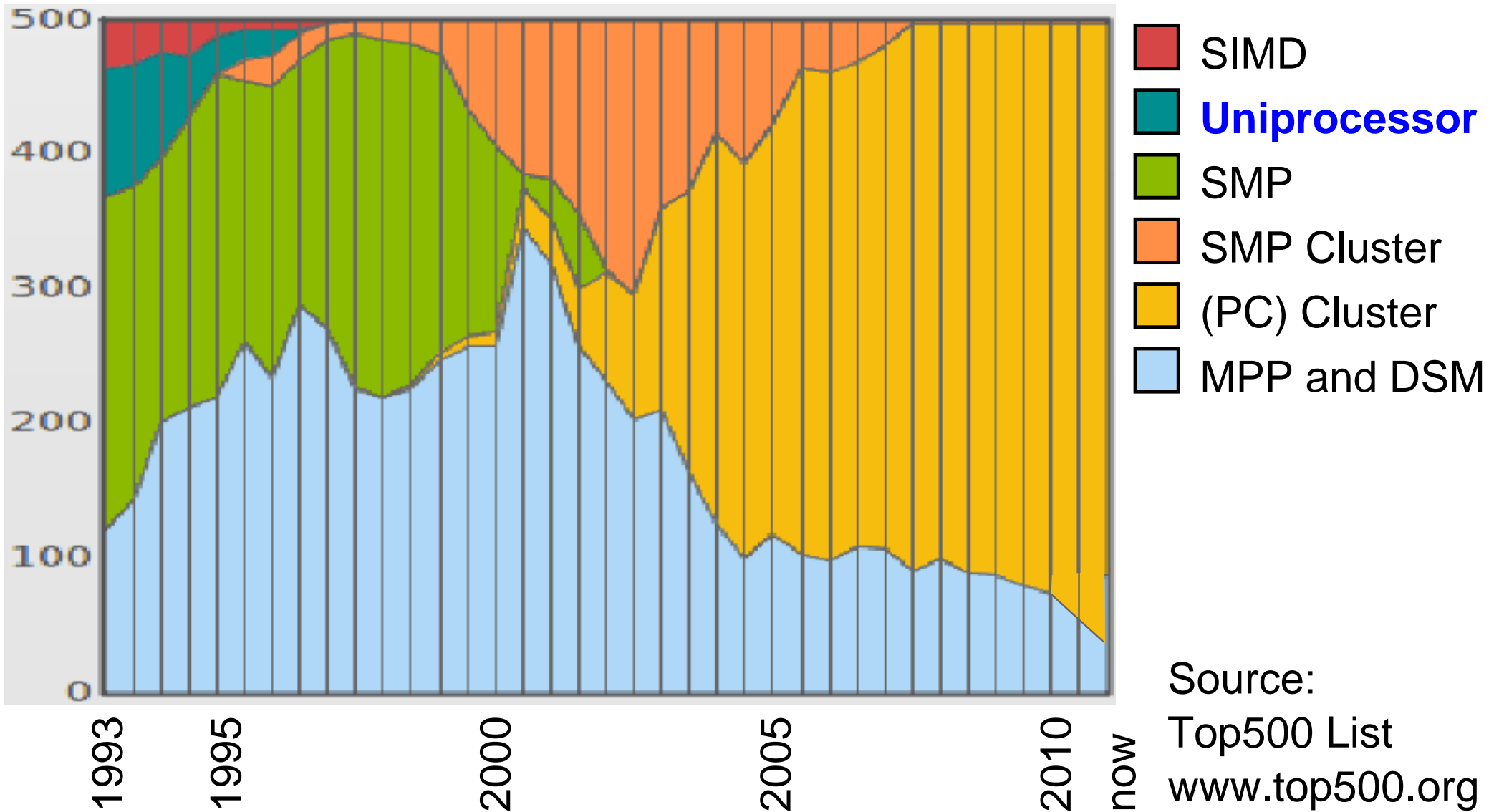
### Why parallel processing? ...

- ➔ **Moore's Law:** the computing power of a processor doubles every 18 months
  - ➔ but: memory speed increases much slower
  - ➔ 2040 the latest: physical limit will be reached
- ➔ Thus:
  - ➔ high performance computers are based on parallel processing
  - ➔ even standard CPUs use parallel processing internally
    - ➔ super scalar processors, pipelining, multicore, ...
- ➔ Economic advantages of parallel computers
  - ➔ cheap standard CPUs instead of specifically developed ones

## 2.1 Motivation ...



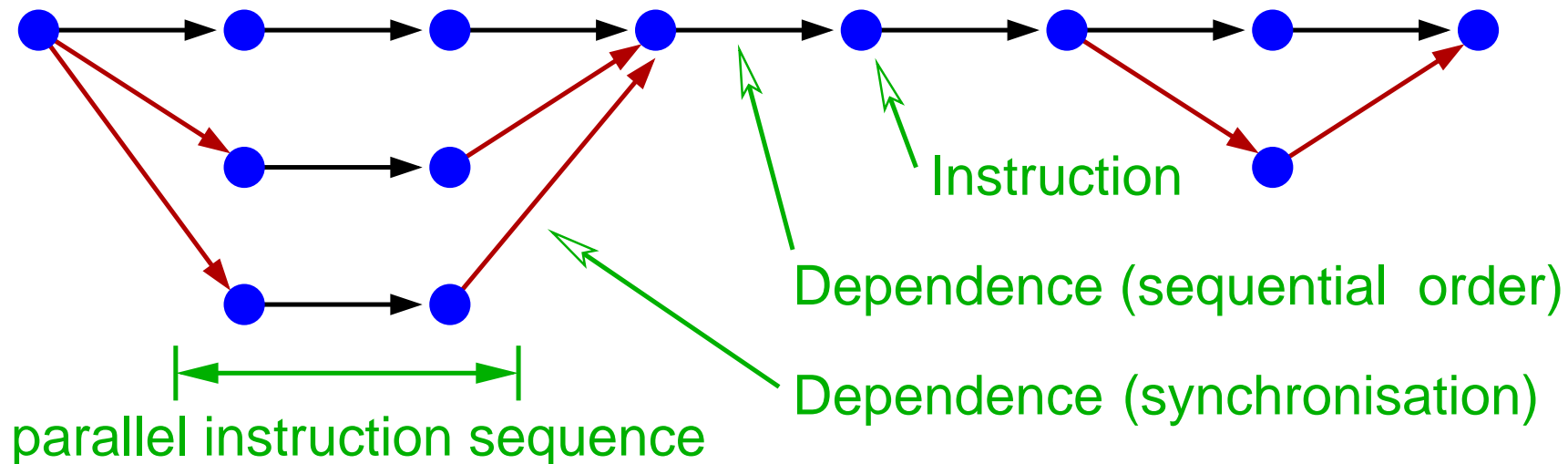
### Architecture trend of high performance computers



Source:  
Top500 List  
[www.top500.org](http://www.top500.org)

### What is a parallel programm?

- ➔ A parallel program can be viewed as a partially ordered set of instructions (activities)
- ➔ the order is given by the dependences between the instructions
- ➔ Independent instructions can be executed in parallel





### Concurrency vs. pipelining

- ➔ **Concurrency** (*Nebenläufigkeit*): instructions are executed simultaneously in different execution units
- ➔ **Pipelining**: execution of instructions is subdivided into sequential phases.  
Different phases of **different** instruction **instances** are executed simultaneously.
- ➔ Remark: here, the term “instruction” means a generic compute activity, depending on the layer of abstraction we are considering
  - ➔ e.g., machine instruction, execution of a sub-program

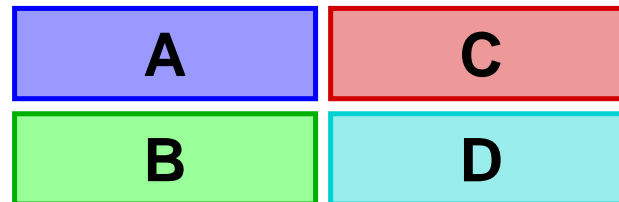


### Concurrency vs. pipelining ...

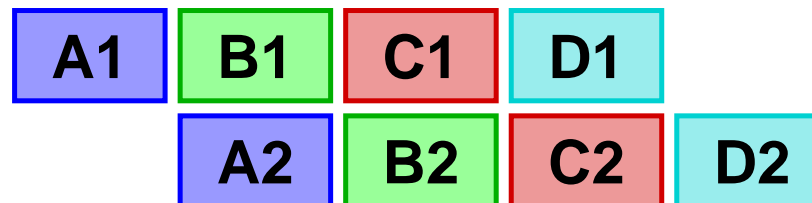
Sequential Execution



Concurrent Execution



Pipelining (2 Stages)





### At which layers of programming can we use parallelism?

- ➔ There is no consistent classification
- ➔ E.g., layers in the book from Waldschmidt, *Parallelrechner: Architekturen - Systeme - Werkzeuge*, Teubner, 1995:
  - ➔ application programs
  - ➔ cooperating processes
  - ➔ data structures
  - ➔ statements and loops
  - ➔ machine instruction

*“They are heterogeneous, subdivided according to different characteristics, and partially overlap.”*



### View of the application developer (design phase):

- ➔ “Natural parallelism”
  - ➔ e.g., computing the forces for all stars of a galaxy
  - ➔ often too fine-grained
- ➔ **Data parallelism** (domain decomposition, *Gebietsaufteilung*)
  - ➔ e.g., sequential processing of all stars in a space region
- ➔ **Task parallelism**
  - ➔ e.g., pre-processing, computation, post-processing, visualisation



### View of the programmer:

#### ➔ **Explicit parallelism**

- ➔ exchange of data (communication / synchronisation) must be explicitly programmed

#### ➔ **Implicit parallelism**

- ➔ by the compiler
  - ➔ directive controlled or automatic
  - ➔ loop level / statement level
  - ➔ compiler generates code for communication
- ➔ within a CPU (that appears to be sequential from the outside)
  - ➔ super scalar processor, pipelining, ...

### View of the system (computer / operating system):

#### ➔ Program level (job level)

- ➔ independent programs

#### ➔ Process level (task level)

- ➔ cooperating processes

- ➔ mostly with explicit exchange of messages

#### ➔ Block level

- ➔ light weight processes (threads)

- ➔ communication via shared memory

- ➔ often created by the compiler

- ➔ parallelisation of loops



### View of the system (computer / operating system): ...

#### ➔ Instruction level

- ➔ elementary instructions (operations that cannot be further subdivided in the programming language)
- ➔ scheduling is done automatically by the compiler and/or by the hardware at runtime
- ➔ e.g., in VLIW (EPIC, e.g. Itanium) and super scalar processors

#### ➔ Sub-operation level

- ➔ compiler or hardware subdivide elementary instructions into sub-operations that are executed in parallel
  - ➔ e.g., with vector or array operations



### Granularity

- ➔ Defined by the ratio between computation and communication (including synchronisation)
  - ➔ intuitively, this corresponds to the length of the parallel instruction sequences in the partial order
  - ➔ determines the requirements for the parallel computer
    - ➔ especially its communication system
  - ➔ influences the achievable acceleration (*speedup*)
- ➔ Coarse-grained: program and process level
- ➔ Mid-grained: block level
- ➔ Fine-grained: instruction level

## 2.3 Parallelisation and Data Dependences



- ➔ Important question: when can two instructions  $S_1$  and  $S_2$  be executed in parallel?
  - ➔ Answer: if there are no **dependences** between them
- ➔ Assumption: instruction  $S_1$  can and should be executed **before** instruction  $S_2$  according to the sequential code
  - ➔ e.g.:  
 $S_1: \quad x = b + 2 * a;$   
 $\quad \quad y = a * (c - 5);$   
 $S_2: \quad z = \text{abs}(x - y);$
  - ➔ but also in different iterations of a loop
- ➔ **True / flow dependence** (*echte Abhängigkeit*)  $S_1 \xrightarrow{\delta^t} S_2$

```
for (i=1; i<N; i++) {  
    a[i] = a[i-1] + b[i];  
    ...  
}
```

## 2.3 Parallelisation and Data Dependences



- ➔ Important question: when can two instructions  $S_1$  and  $S_2$  be executed in parallel?
  - ➔ Answer: if there are no **dependences** between them
- ➔ Assumption: instruction  $S_1$  can and should be executed **before** instruction  $S_2$  according to the sequential code
  - ➔ e.g.:  
 $S_1: \quad x = b + 2 * a;$   
 $\quad \quad y = a * (c - 5);$   
 $S_2: \quad z = \text{abs}(x - y);$
  - ➔ but also in different iterations of a loop
- ➔ **True / flow dependence** (*echte Abhängigkeit*)  $S_1 \xrightarrow{\delta^t} S_2$

```
for (i=1; i<N; i++) {  
    a[i] = a[i-1] + b[i];  
    ...  
}
```

$S_1: \quad a[1] = a[0] + b[1];$

$S_2: \quad a[2] = a[1] + b[2];$

## 2.3 Parallelisation and Data Dependences



- ➔ Important question: when can two instructions  $S_1$  and  $S_2$  be executed in parallel?
  - ➔ Answer: if there are no **dependences** between them
- ➔ Assumption: instruction  $S_1$  can and should be executed **before** instruction  $S_2$  according to the sequential code
  - ➔ e.g.:  
 $S_1: x = b + 2 * a;$   
 $y = a * (c - 5);$   
 $S_2: z = \text{abs}(x - y);$
  - ➔ but also in different iterations of a loop
- ➔ **True / flow dependence** (*echte Abhängigkeit*)  $S_1 \xrightarrow{\delta^t} S_2$

```
for (i=1; i<N; i++) {  
    a[i] = a[i-1] + b[i];  
    ...  
}
```

```
 $S_1: a[1] = a[0] + b[1];$   
 $S_2: a[2] = a[1] + b[2];$ 
```

## 2.3 Parallelisation and Data Dependences



- ➔ Important question: when can two instructions  $S_1$  and  $S_2$  be executed in parallel?
  - ➔ Answer: if there are no **dependences** between them
- ➔ Assumption: instruction  $S_1$  can and should be executed **before** instruction  $S_2$  according to the sequential code
  - ➔ e.g.:  
 $S_1: x = b + 2 * a;$   
 $y = a * (c - 5);$   
 $S_2: z = \text{abs}(x - y);$
  - ➔ but also in different iterations of a loop
- ➔ **True / flow dependence** (*echte Abhängigkeit*)  $S_1 \xrightarrow{\delta^t} S_2$

$S_1: a[1] = a[0] + b[1];$

$S_2: a[2] = a[1] + b[2];$

$S_1$  ( $i=1$ ) writes to  $a[1]$ , which is later read by  $S_2$  ( $i=2$ )



→ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

```
for (i=1; i<N; i++) {  
    a[i] = a[i+1];  
    ...  
}
```

➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

```
for (i=1; i<N; i++) {  
    a[i] = a[i+1];  
    ...  
}
```

$S_1: a[1] = a[2];$

$S_2: a[2] = a[3];$

➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

```
for (i=1; i<N; i++) {  
    a[i] = a[i+1];  
    ...  
}
```

$S_1$ : a[1] = a[2];

$S_2$ : a[2] = a[3];

$\delta^a$

## 2.3 Parallelisation and Data Dependences ...



➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

$S_1$ : `a[1] = a[2];`

$S_2$ : `a[2] = a[3];`

$\delta^a$

$S_1$  (i=1) read the value of a[2], which is overwritten by  $S_2$  (i=2)



➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

$S_1$ : `a[1] = a[2];`

$S_2$ : `a[2] = a[3];`

$S_1$  (i=1) read the value of a[2], which is overwritten by  $S_2$  (i=2)

➔ **Output dependence** (*Ausgabeabhängigkeit*)  $S_1 \xrightarrow{\delta^o} S_2$

```
for (i=1; i<N; i++) {  
    s = a[i];  
    ...  
}
```

➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

$S_1$ : `a[1] = a[2];`

$S_2$ : `a[2] = a[3];`

$S_1$  (i=1) read the value of a[2], which is overwritten by  $S_2$  (i=2)

➔ **Output dependence** (*Ausgabeabhängigkeit*)  $S_1 \xrightarrow{\delta^o} S_2$

```
for (i=1; i<N; i++) {  
    s = a[i];  
    ...
```

$S_1$ : `s = a[1];`

$S_2$ : `s = a[2];`

➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

$S_1$ : `a[1] = a[2];`

$S_2$ : `a[2] = a[3];`

$S_1$  (i=1) read the value of a[2], which is overwritten by  $S_2$  (i=2)

➔ **Output dependence** (*Ausgabeabhängigkeit*)  $S_1 \xrightarrow{\delta^o} S_2$

```
for (i=1; i<N; i++) {  
    s = a[i];  
    ...  
}
```

$S_1$ : `s = a[1];`

$S_2$ : `s = a[2];`

➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

$S_1$ : `a[1] = a[2];`

$S_2$ : `a[2] = a[3];`

$S_1$  (i=1) read the value of `a[2]`, which is overwritten by  $S_2$  (i=2)

➔ **Output dependence** (*Ausgabeabhängigkeit*)  $S_1 \xrightarrow{\delta^o} S_2$

$S_1$ : `s = a[1];`

$S_2$ : `s = a[2];`

$S_1$  (i=1) writes a value to `s`, which is overwritten by  $S_2$  (i=2)

➔ **Anti dependence** (*Antiabhängigkeit*)  $S_1 \xrightarrow{\delta^a} S_2$

$S_1$ : `a[1] = a[2];`

$S_2$ : `a[2] = a[3];`

$S_1$  (i=1) read the value of a[2], which is overwritten by  $S_2$  (i=2)

➔ **Output dependence** (*Ausgabeabhängigkeit*)  $S_1 \xrightarrow{\delta^o} S_2$

$S_1$ : `s = a[1];`

$S_2$ : `s = a[2];`


$S_1$  (i=1) writes a value to s, which is overwritten by  $S_2$  (i=2)

➔ **Anti** and **Output** dependences can always be removed by consistent renaming of variables

### Data dependences and synchronisation

- ➔ Two instructions  $S_1$  and  $S_2$  with a data dependence  $S_1 \rightarrow S_2$  can be distributed by different threads **only if** a correct synchronisation is performed
  - ➔  $S_2$  must be executed **after**  $S_1$
  - ➔ e.g., by using signal/wait or a message
- ➔ in the previous example:

```
x = b + 2 * a;
y = a * (c-5);
z = abs(x-y);
```

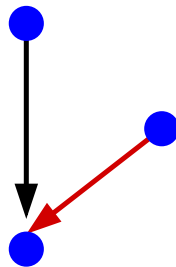


### Data dependences and synchronisation

- ➔ Two instructions  $S_1$  and  $S_2$  with a data dependence  $S_1 \rightarrow S_2$  can be distributed by different threads **only if** a correct synchronisation is performed
  - ➔  $S_2$  must be executed **after**  $S_1$
  - ➔ e.g., by using `signal/wait` or a message
- ➔ in the previous example:

Thread 1

```
x = b + 2 * a;  
  
wait (cond);  
z = abs (x-y);
```



Thread 2

```
y = a * (c-5);  
signal (cond);
```



### Classification of computer architectures according to Flynn

➔ Criteria for differentiation:

- ➔ how many **instruction streams** does the computer process at a given point in time (single, multiple)?
- ➔ how many **data streams** does the computer process at a given point in time (single, multiple)?

➔ This leads to four possible classes:

- ➔ SISD: **S**ingle **I**nstruction stream, **S**ingle **D**ata stream
  - ➔ single processor (core) systems
- ➔ MIMD: **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams
  - ➔ all kinds of multiprocessor systems
- ➔ SIMD: vector computers, vector extensions, GPUs
- ➔ MISD: empty, not really sensible



### Classes of MIMD computers

- ➔ Considering two criteria:
  - ➔ physically global vs. distributed memory
  - ➔ shared vs. distributed address space
- ➔ **NORMA: No Remote Memory Access**
  - ➔ distributed memory, distributed address space
  - ➔ i.e., no access to memory modules of non-local nodes
  - ➔ communication is only possible via messages
  - ➔ typical representative of this class:
    - ➔ ***distributed memory systems (DMM)***
      - ➔ also called MPP (massively parallel processor)
    - ➔ in principle also any computer networks (cluster, grid, cloud, ...)



### Classes of MIMD computers ...

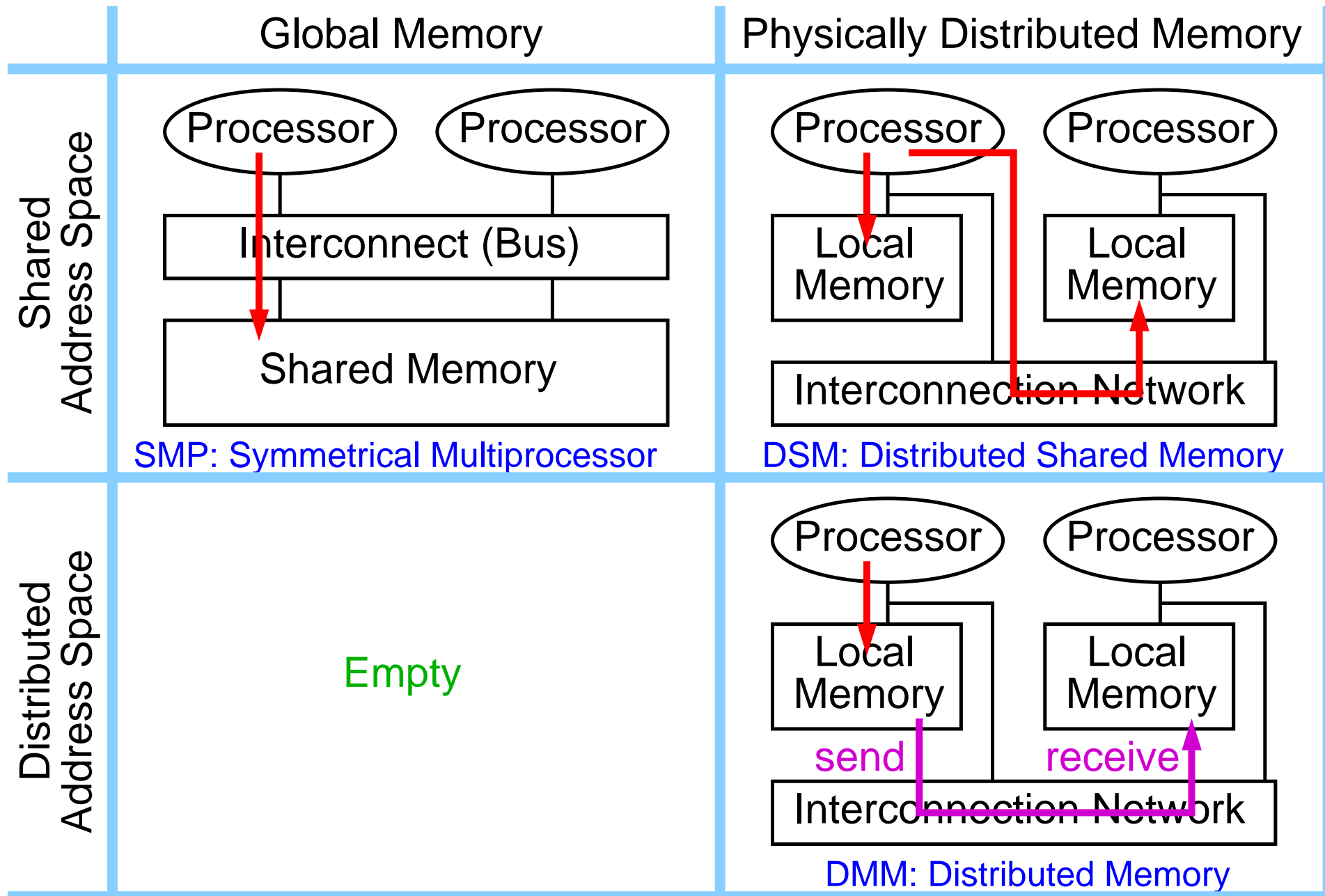
#### ➔ **UMA: Uniform Memory Access**

- ➔ global memory, shared address space
- ➔ all processors access the memory in the same way
- ➔ access time is equal for all processors
- ➔ typical representative of this class:  
**symmetrical multiprocessor (SMP)**, early multicore-CPUs

#### ➔ **NUMA: Nonuniform Memory Access**

- ➔ distributed memory, shared address space
- ➔ access to local memory is faster than access to remote one
- ➔ typical representative of this class:  
**distributed shared memory (DSM)** systems, modern multicore-CPU

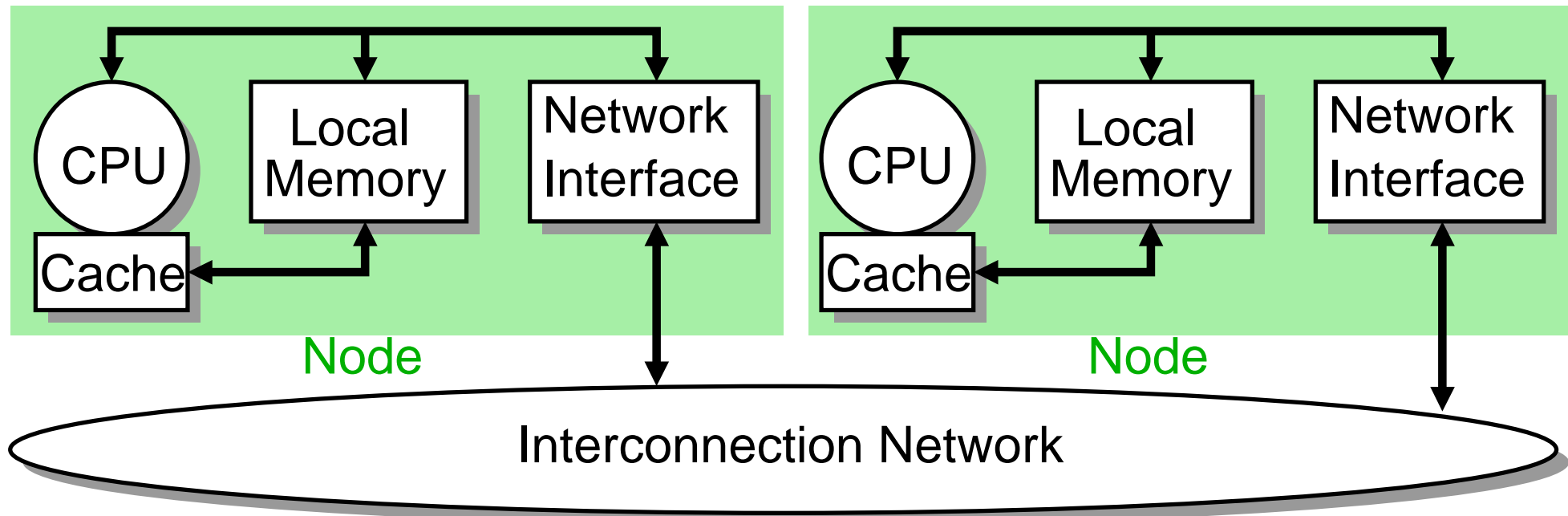
## 2.4 Parallel Computer Architectures ...



## 2.4.1 MIMD: Message Passing Systems



### Multiprocessor systems with distributed memory



- ➔ **NORMA**: No Remote Memory Access
- ➔ Good scalability (up to several 100000 nodes)
- ➔ Communication and synchronisation via message passing



### Historical evolution

- ➔ In former times: proprietary hardware for nodes and network
  - ➔ distinct node architecture (processor, network adapter, ...)
  - ➔ often static interconnection networks with *store and forward*
  - ➔ often distinct (mini) operating systems
- ➔ Today:
  - ➔ cluster with standard components (PC server)
    - ➔ usually with SMP (sometimes vector computers) as nodes
    - ➔ nodes often use accelerators (GPUs)
  - ➔ switched high performance interconnection networks
    - ➔ 100Gbit/s Ethernet, Infiniband, ...
  - ➔ standard operating systems (UNIX or Linux derivatives)



### Properties

- ➔ No shared memory or address areas between nodes
- ➔ Communication via exchange of messages
  - ➔ application layer: libraries like e.g., MPI
  - ➔ system layer: proprietary protocols or TCP/IP
  - ➔ latency caused by software often much larger than hardware latency ( $\sim 1 - 50\mu s$  vs.  $\sim 20 - 100ns$ )
- ➔ In principle unlimited scalability
  - ➔ e.g. Frontier: 135936 nodes, (8699904 cores)

---

# Parallel Processing

Winter Term 2025/26

27.10.2025

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

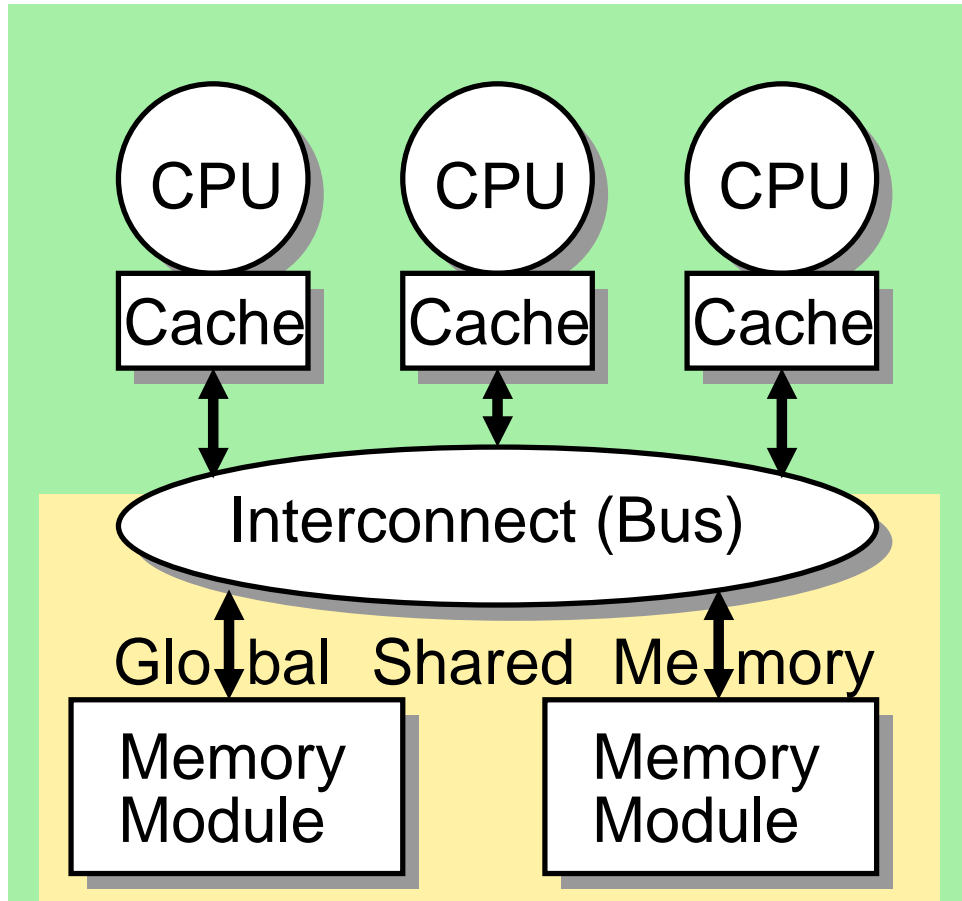
Stand: November 10, 2025



### Properties ...

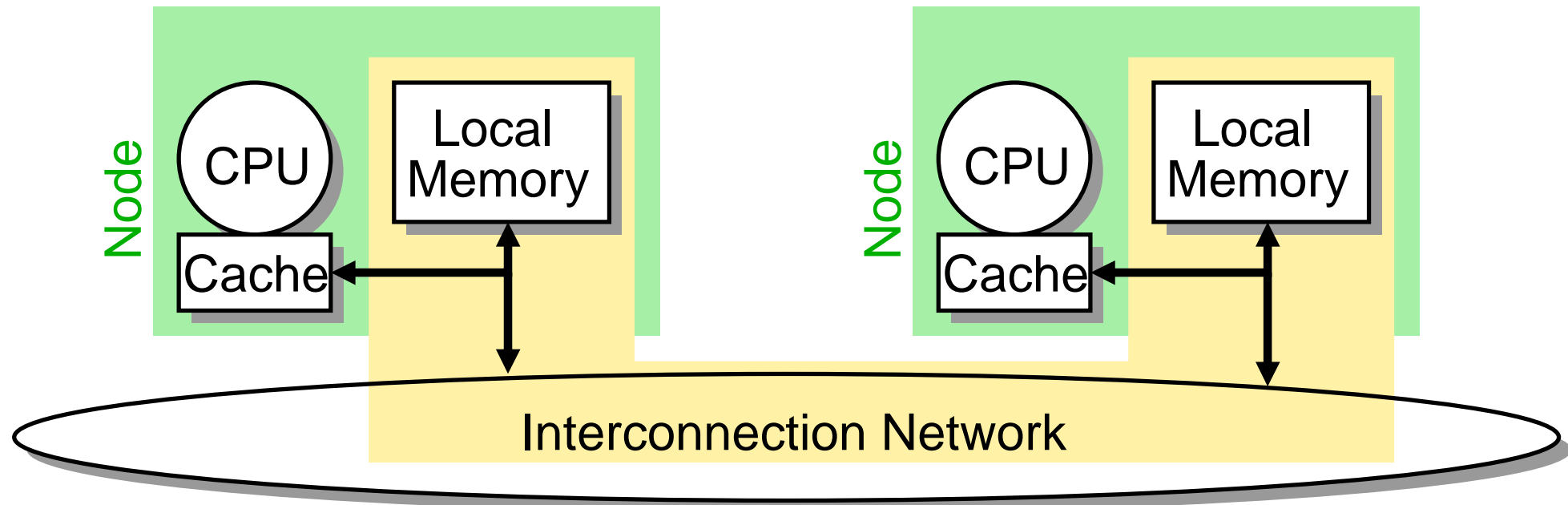
- ➔ Independent operating system on each node
- ➔ Often with shared file system
  - ➔ e.g., parallel file system, connected to each node via a (distinct) interconnection network
  - ➔ or simply NFS (in small clusters)
- ➔ Usually no *single system image*
  - ➔ user/administrator “sees” several computers
- ➔ Often no direct, interactive access to all nodes
  - ➔ *batch queueing systems* assign nodes (only) on request to parallel programs
    - ➔ often exclusively: *space sharing*, partitioning
  - ➔ often small fixed partition for login and interactive use

### Symmetrical multiprocessors (SMP)



- ➔ Global address space
- ➔ **UMA**: *uniform memory access*
- ➔ Communication and Synchronisation via shared memory
- ➔ only feasible with very few processors (ca. 2 - 32)

### Multiprocessor systems with distributed shared memory (DSM)



- ➔ Distributed memory, accessible by all CPUs
- ➔ **NUMA**: *non uniform memory access*
- ➔ Combines shared memory and scalability



### Properties

- ➔ All Processors can access all resources in the same way
  - ➔ but: different access times in NUMA architectures
    - ➔ distribute the data such that most accesses are local
- ➔ Only one instance of the operating systems for the whole computer
  - ➔ distributes processes/thread amongst the available processors
  - ➔ all processors can execute operating system services in an equal way
- ➔ *Single system image*
  - ➔ for user/administrator virtually no difference to a uniprocessor system
- ➔ Especially SMPs (UMA) only have limited scalability

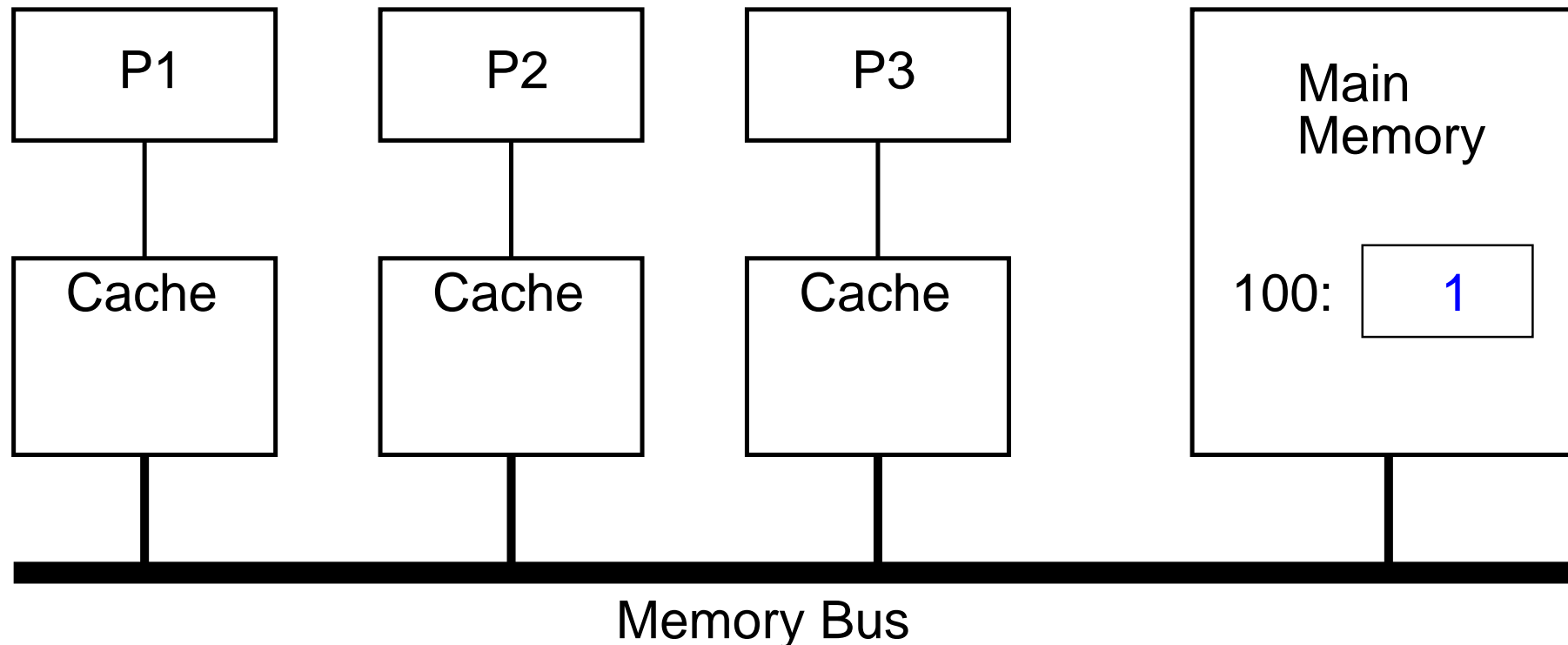


### Caches in shared memory systems

- ➔ **Cache**: fast intermediate storage, close to the CPU
  - ➔ stores copies of the most recently used data from main memory
  - ➔ when the data is in the cache: no access to main memory is necessary
    - ➔ access is 10-1000 times faster
- ➔ Cache are essential in multiprocessor systems
  - ➔ otherwise memory and interconnection network quickly become a bottleneck
  - ➔ exploiting the property of locality
    - ➔ each process mostly works on “its own” data
- ➔ But: the existance of multiple copies of data cean lead to inconsistencies: **cache coherence problem** (👉 **BS-1**)

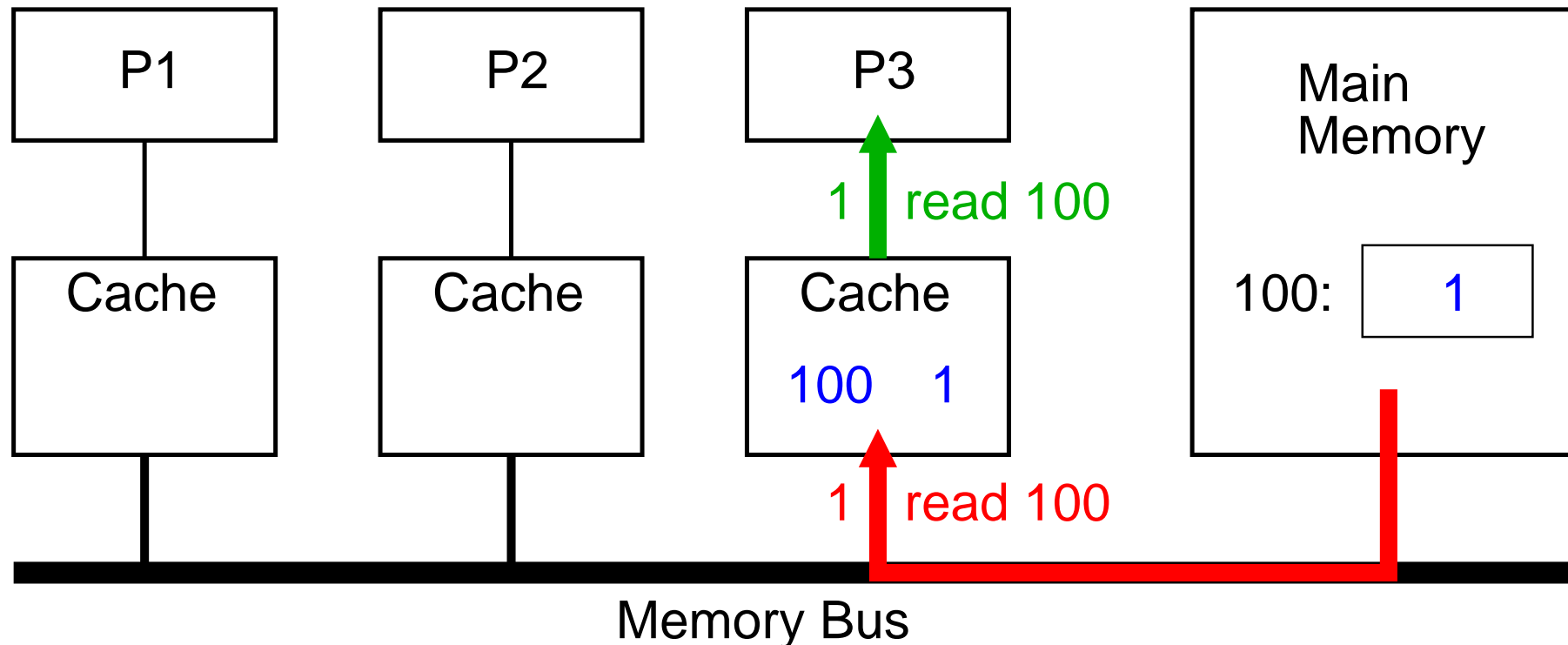
### Cache Coherence Problem: Example

- ➔ Assumption: write access directly updates main memory
- ➔ Three processors access the same memory location



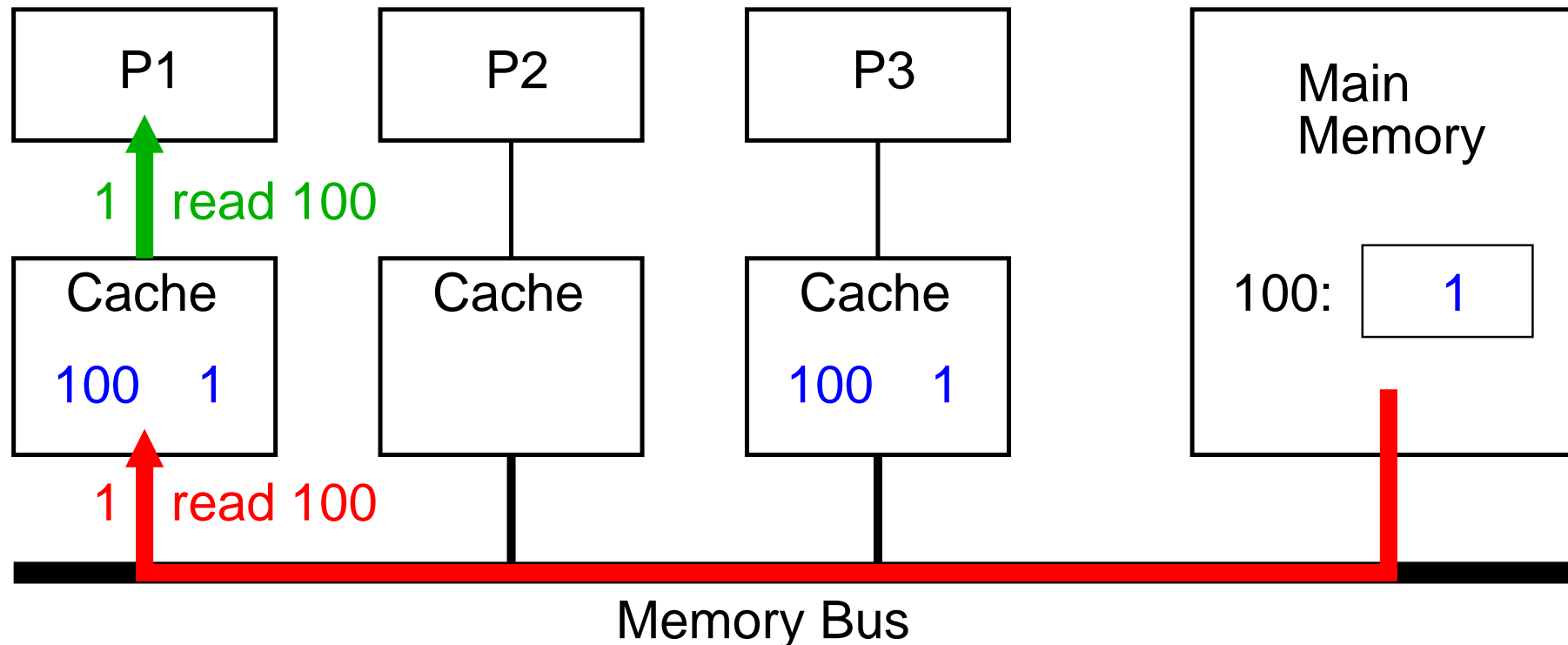
### Cache Coherence Problem: Example

- ➔ Assumption: write access directly updates main memory
- ➔ Three processors access the same memory location



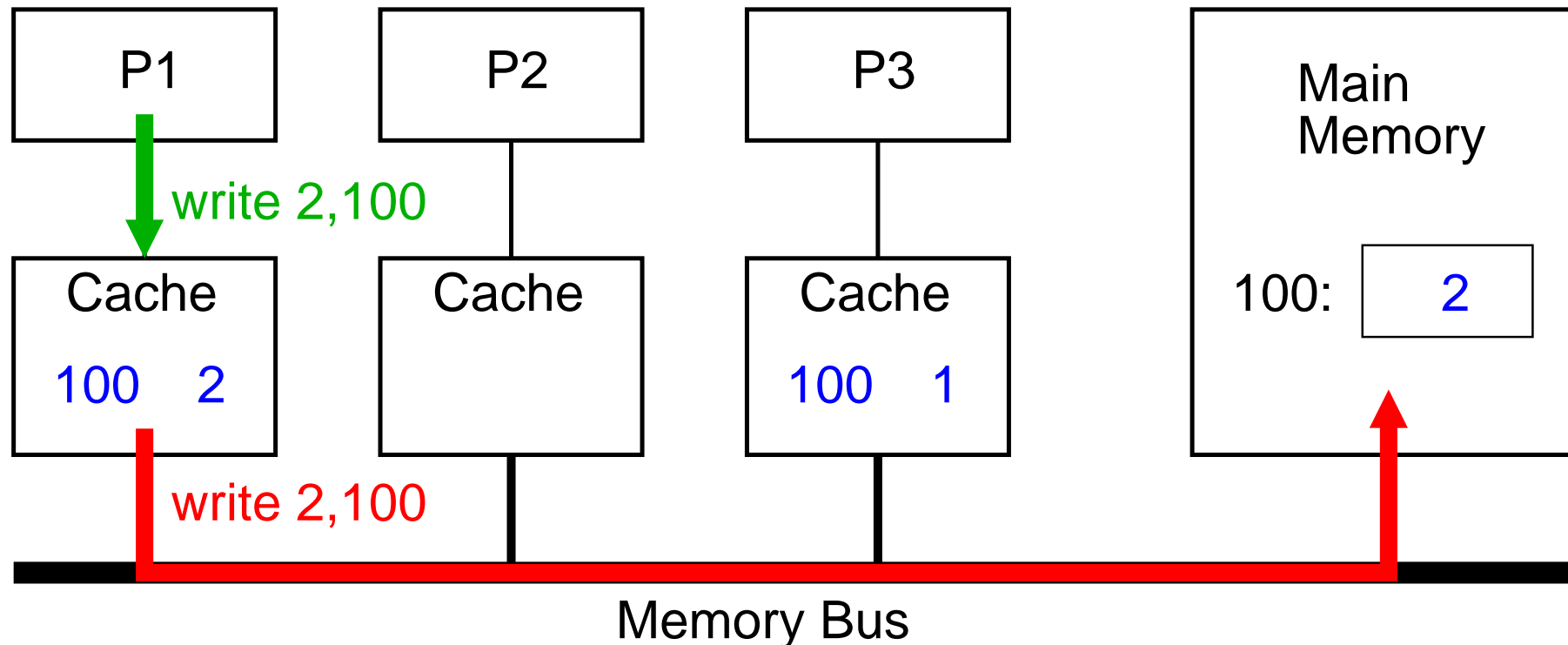
### Cache Coherence Problem: Example

- ➔ Assumption: write access directly updates main memory
- ➔ Three processors access the same memory location



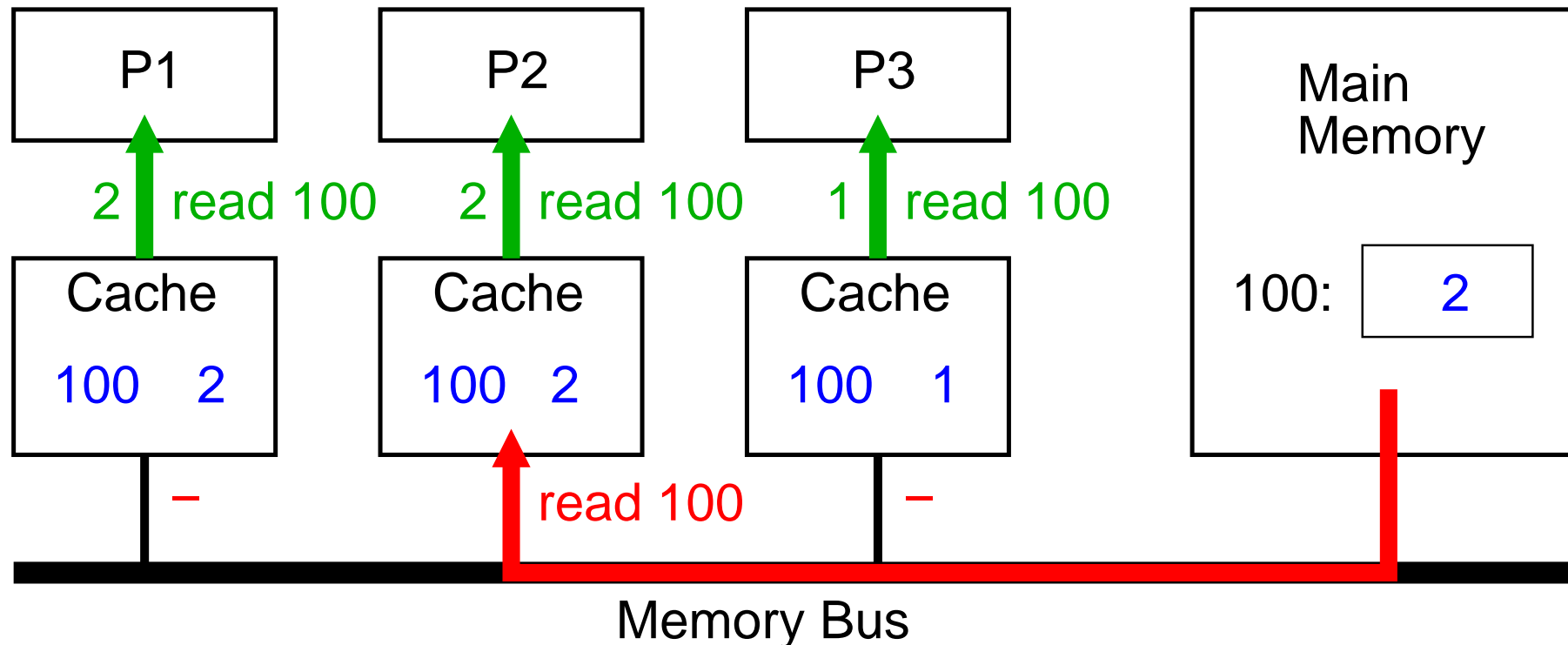
### Cache Coherence Problem: Example

- ➔ Assumption: write access directly updates main memory
- ➔ Three processors access the same memory location



### Cache Coherence Problem: Example

- ➔ Assumption: write access directly updates main memory
- ➔ Three processors access the same memory location and get different results!





### Enforcing cache coherency

- ➔ During a write access, all affected caches (= caches with copies) must be notified
  - ➔ caches invalidate or update the affected entry
- ➔ In UMA systems
  - ➔ bus as interconnection network: every access to main memory is visible for everybody (broadcast)
  - ➔ caches “listen in” on the bus (*bus snooping*)
  - ➔ (relatively) simple cache coherence protocols
    - ➔ e.g., MESI protocol
  - ➔ but: bad scalability, since the bus is a shared central resource



### Enforcing cache coherency ...

- ➔ In NUMA systems (ccNUMA: *cache coherent NUMA*)
  - ➔ accesses to main memory normally are not visible to other processors
  - ➔ affected caches must be notified explicitly
    - ➔ requires a list of all affected caches (broadcasting to all processors is too expensive)
    - ➔ message transfer time leads to additional consistency problems
  - ➔ cache coherence protocols (*directory protocols*) become very complex
  - ➔ but: good scalability



### Memory consistency (*Speicherkonsistenz*)

- ➔ Cache coherence only defines the behavior with respect to **one memory location** at a time
  - ➔ **which values** can a read operation return?
- ➔ Remaining question:
  - ➔ **when** does a processor see the value, which was written by another processor?
  - ➔ more exact: in **which order** does a processor see the write operations on **different** memory locations?



### Memory consistency: a simple example

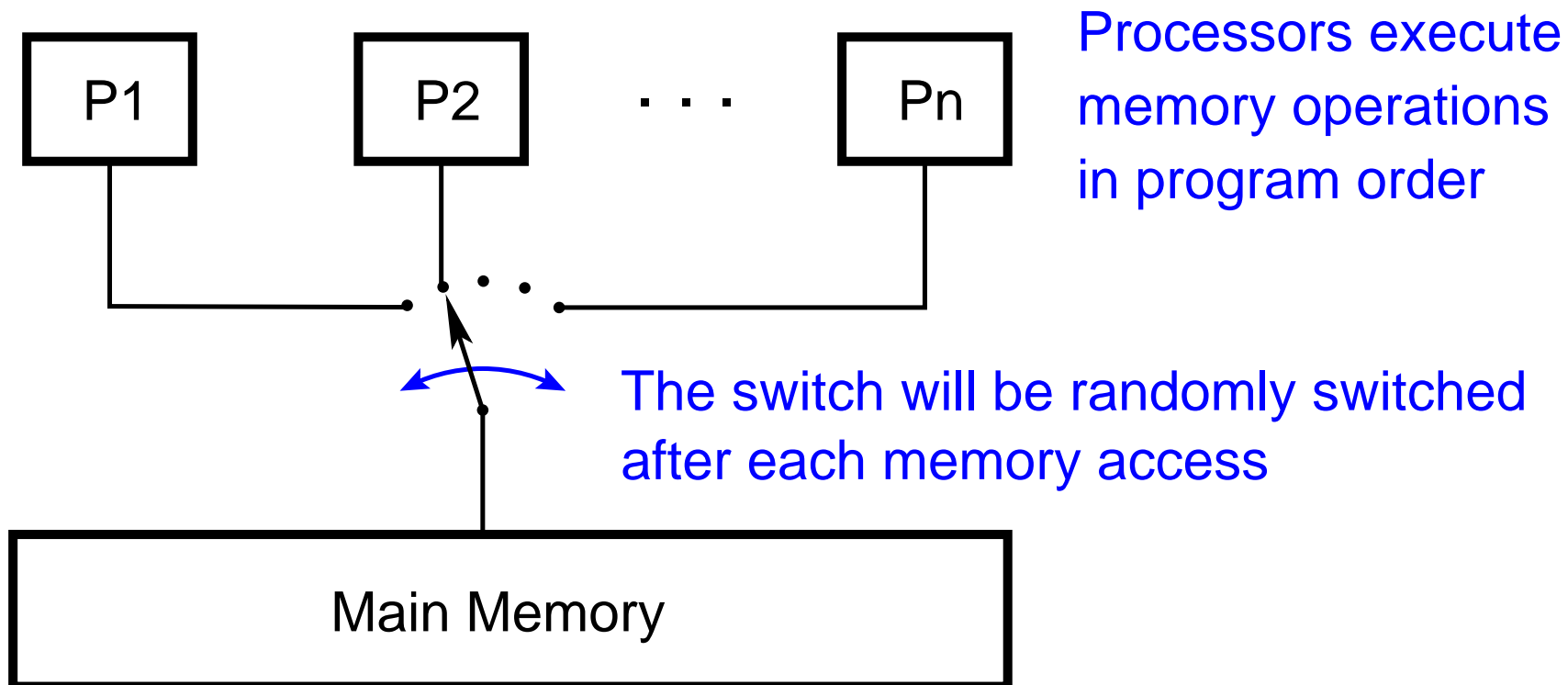
Thread $T_1$	Thread $T_2$
A = 0;	B = 0;
...;	...;
A = 1;	B = 1;
print B;	print A;

- ➔ Intuitive expectation: the output "0 0" can never occur
- ➔ But: with many SMPs/DSMs the output "0 0" is possible
  - ➔ (CPUs with dynamic instruction scheduling or write buffers)
- ➔ In spite of cache coherency: intuitively inconsistent view on the main memory:

$$T_1: A=1, B=0 \quad T_2: A=0, B=1$$

### Definition: sequential consistency

Sequential consistency is given, when the result of each execution of a parallel program can also be produced by the following abstract machine:




### Interleavings (*Verzahnungen*) in the example

Some possible execution sequences  
using the abstract machine:

A = 0	A = 0	A = 0
B = 0	B = 0	B = 0
A = 1	A = 1	B = 1
B = 1	print B	print A
print B	B = 1	A = 1
print A	print A	print B
B=1 A=1	B=0 A=1	B=1 A=0

No sequential  
consistency:

A = 0
B = 0
<b>B = 1</b>
print A
A = 1
print B
B=0 A=0





### Weak consistency models

- ➔ The requirement of sequential consistency leads to strong restrictions for the computer architecture
  - ➔ CPUs can not use instruction scheduling and write buffers
  - ➔ NUMA systems can not be realized efficiently
- ➔ Thus: parallel computers with shared memory (UMA and NUMA) use **weak consistency models!**
  - ➔ allows, e.g., swapping of write operations
  - ➔ however, each processor **always** sees **its own** write operations in program order
- ➔ Remark: also optimizing compilers can lead to weak consistency
  - ➔ swapping of instructions, register allocation, ...
  - ➔ declare the affected variables as `volatile!`



### Consequences of weak consistency: examples

➔ all variables are initially 0

Possible results with  
sequential consistency ↴

"unexpected" behavior with  
weak consistency:

<code>A=1;</code> <code>print B;</code>	<code>B=1;</code> <code>print A;</code>	0,1 1,0 1,1	due to swapping of the read and write accesses
<code>A=1;</code> <code>valid=1;</code>	<code>while (!valid);</code> <code>print A;</code>	1	due to swapping of the write accesses to A and valid



### Weak consistency models ...

- ➔ Memory consistency can (and must!) be enforced as needed, using special instructions
  - ➔ *fence / memory barrier (Speicherbarriere)*
    - ➔ all previous memory operations are completed; subsequent memory operations are started only after the barrier
  - ➔ *acquire and release*
    - ➔ *acquire*: subsequent memory operations are started only after the *acquire* is finished
    - ➔ *release*: all previous memory operations are completed
    - ➔ pattern of use is equal to mutex locks



### Enforcing consistency in the examples

➔ Here shown with memory barriers:

```
A=1;  
fence;  
print B;
```

```
B=1;  
fence;  
print A;
```

Fence ensures that the write access is finished before reading

```
A=1;  
fence;  
valid=1;
```

```
while (!valid);  
fence;  
print A;
```

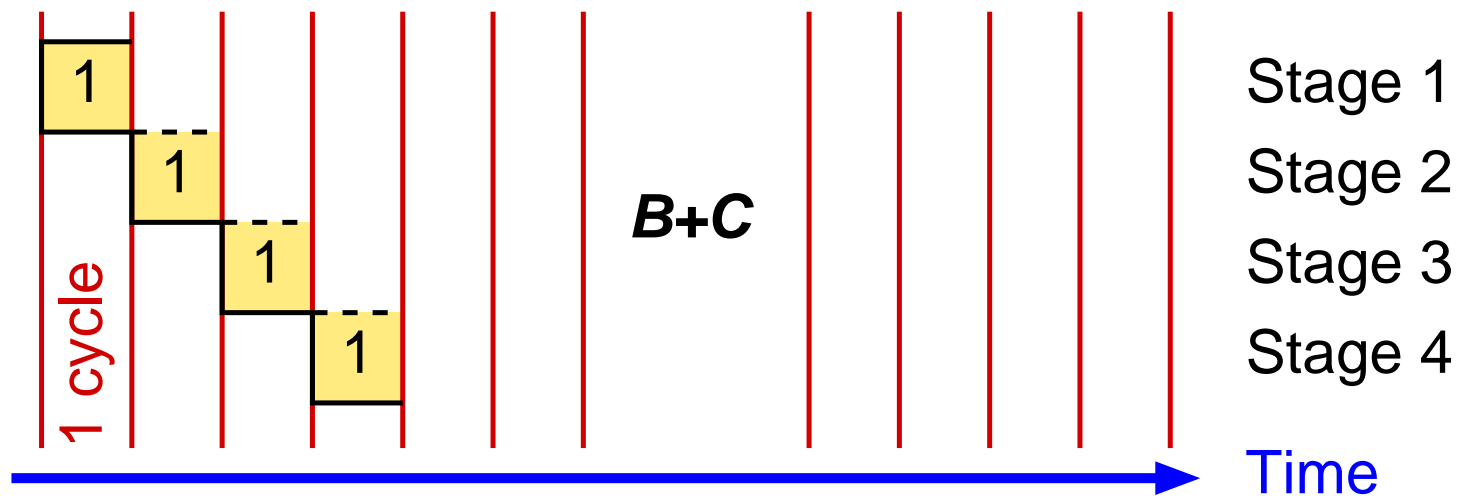
Fence ensures that 'A' is valid before 'valid' is set and that A is read only after 'valid' has been set



- ➔ Only a single instruction stream, however, the instructions have **vectors** as operands  $\Rightarrow$  data parallelism
- ➔ **Vector** = one-dimensional array of numbers
- ➔ Variants:
  - ➔ vector computers
    - ➔ pipelined arithmetic units (vector units) for the processing of vectors
  - ➔ SIMD extensions in processors (SSE, AVX)
    - ➔ Intel: 256 Bit registers with, e.g., four 64 Bit double values
  - ➔ graphics processors (GPUs)
    - ➔ multiple streaming multiprocessors
    - ➔ streaming multiprocessor contains several arithmetic units (*CUDA cores*), which all execute the same instruction

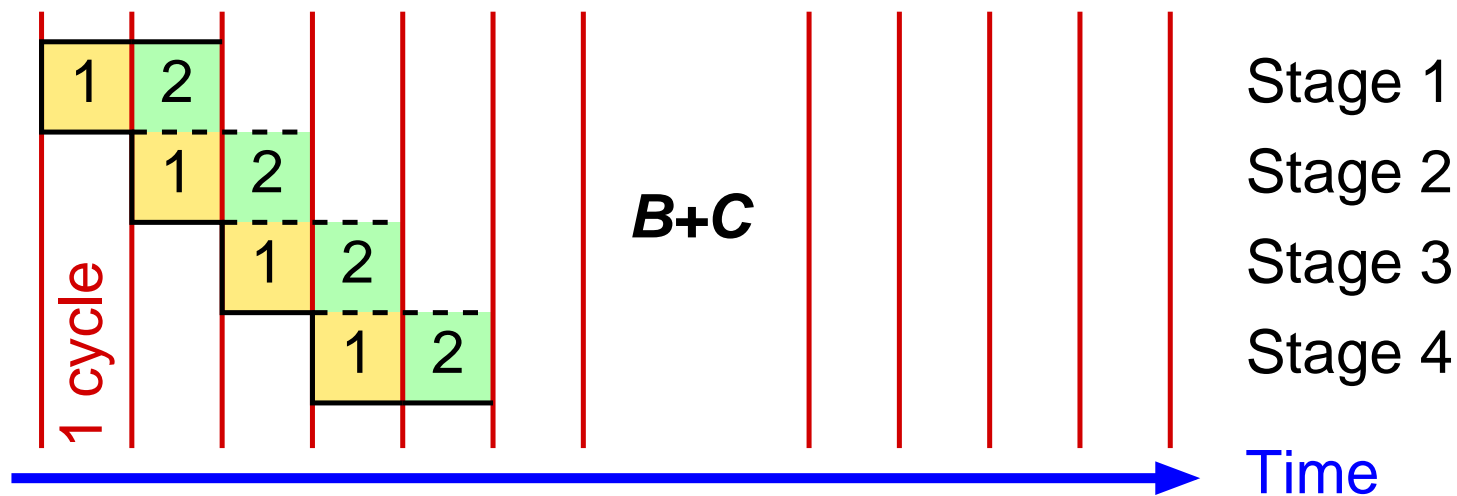
### Example: addition of two vectors

- ➔  $A_j = B_j + C_j$ , for all  $j = 1, \dots, N$
- ➔ Vector computer: the elements of the vectors are added in a pipeline: **sequentially**, but **overlapping**
- ➔ if a scalar addition takes four clock cycles (i.e., 4 pipeline stages), the following sequence will result:



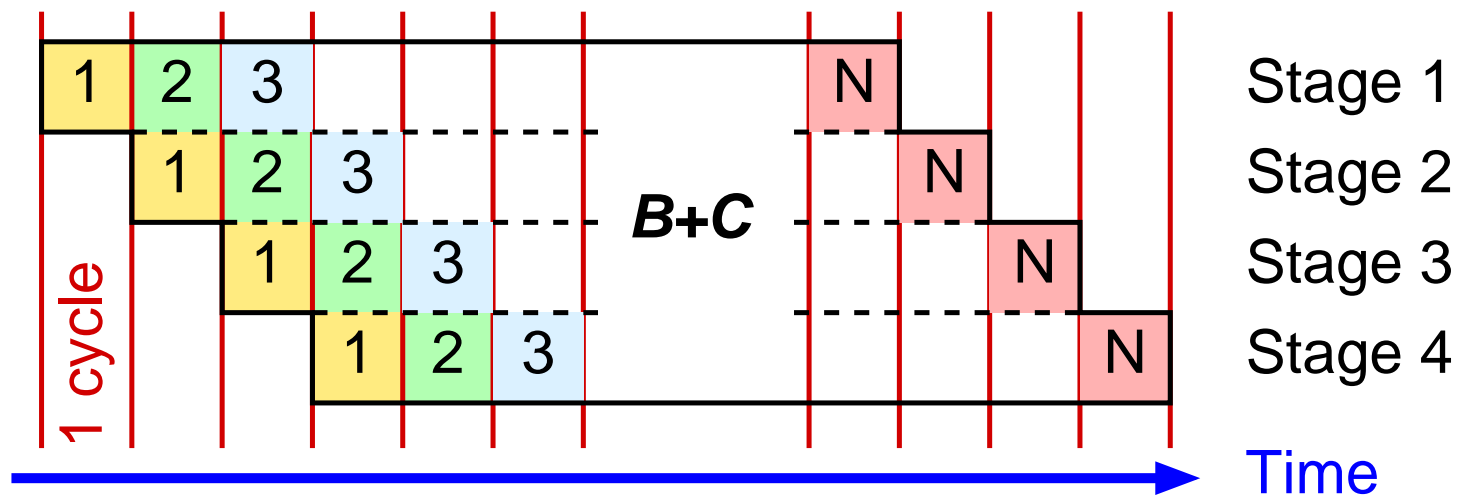
### Example: addition of two vectors

- ➔  $A_j = B_j + C_j$ , for all  $j = 1, \dots, N$
- ➔ Vector computer: the elements of the vectors are added in a pipeline: **sequentially**, but **overlapping**
- ➔ if a scalar addition takes four clock cycles (i.e., 4 pipeline stages), the following sequence will result:



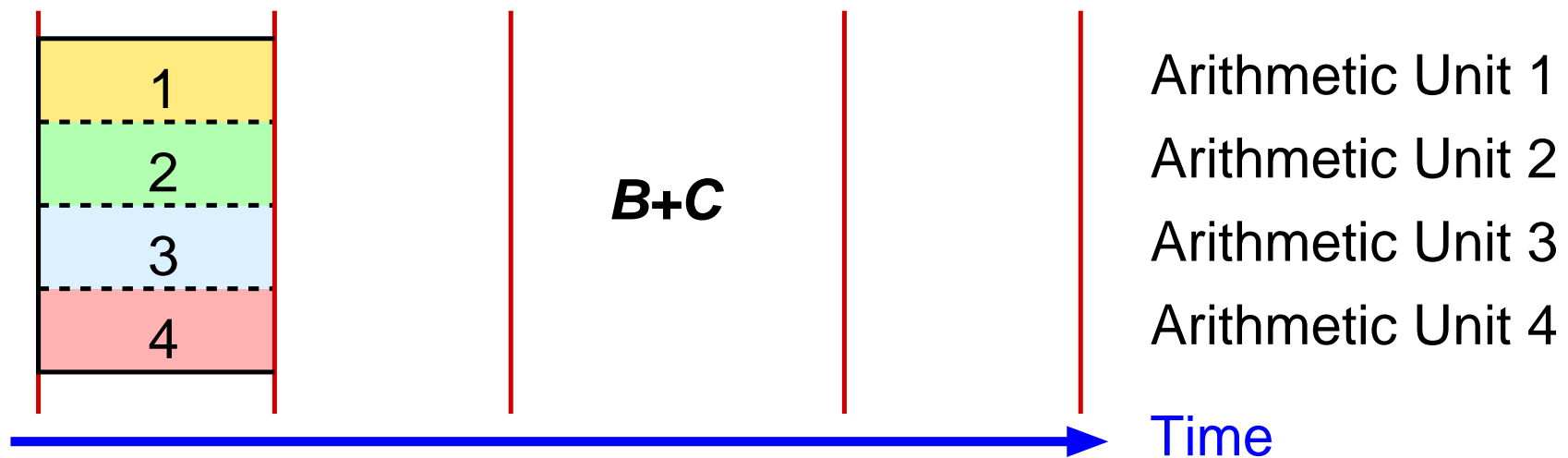
### Example: addition of two vectors

- ➔  $A_j = B_j + C_j$ , for all  $j = 1, \dots, N$
- ➔ Vector computer: the elements of the vectors are added in a pipeline: **sequentially**, but **overlapping**
- ➔ if a scalar addition takes four clock cycles (i.e., 4 pipeline stages), the following sequence will result:



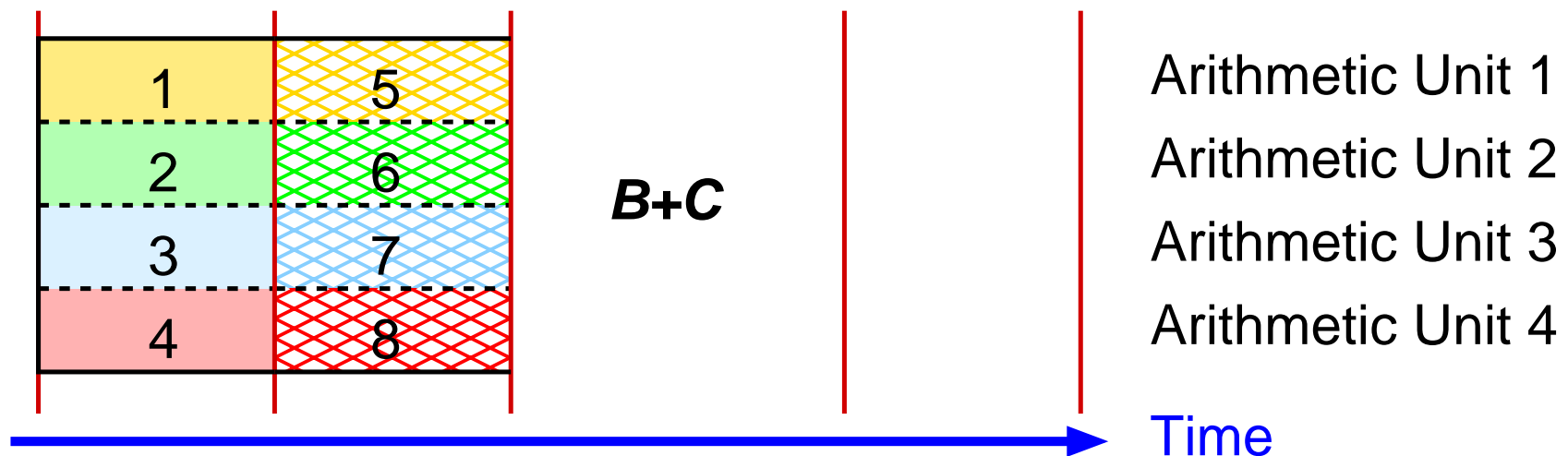
### Example: addition of two vectors

- ➔  $A_j = B_j + C_j$ , for all  $j = 1, \dots, N$
- ➔ AVX and GPU: several elements of the vectors are added **concurrently (in parallel)**
- ➔ if, e.g., four additions can be done at the same time, the following sequence will result:



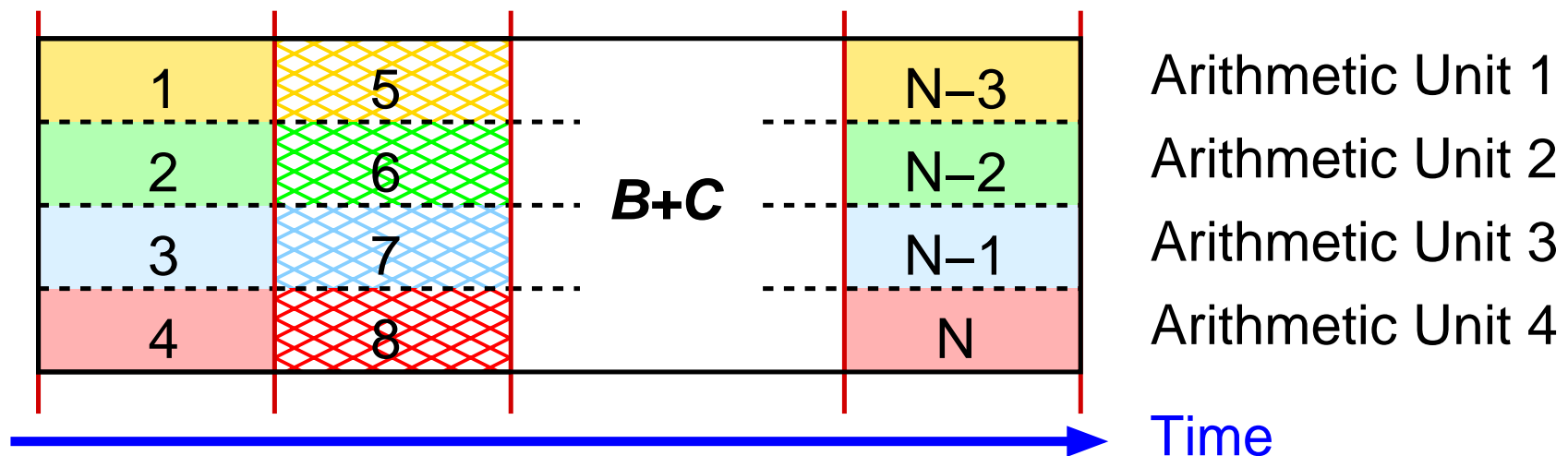
### Example: addition of two vectors

- ➔  $A_j = B_j + C_j$ , for all  $j = 1, \dots, N$
- ➔ AVX and GPU: several elements of the vectors are added **concurrently (in parallel)**
  - ➔ if, e.g., four additions can be done at the same time, the following sequence will result:



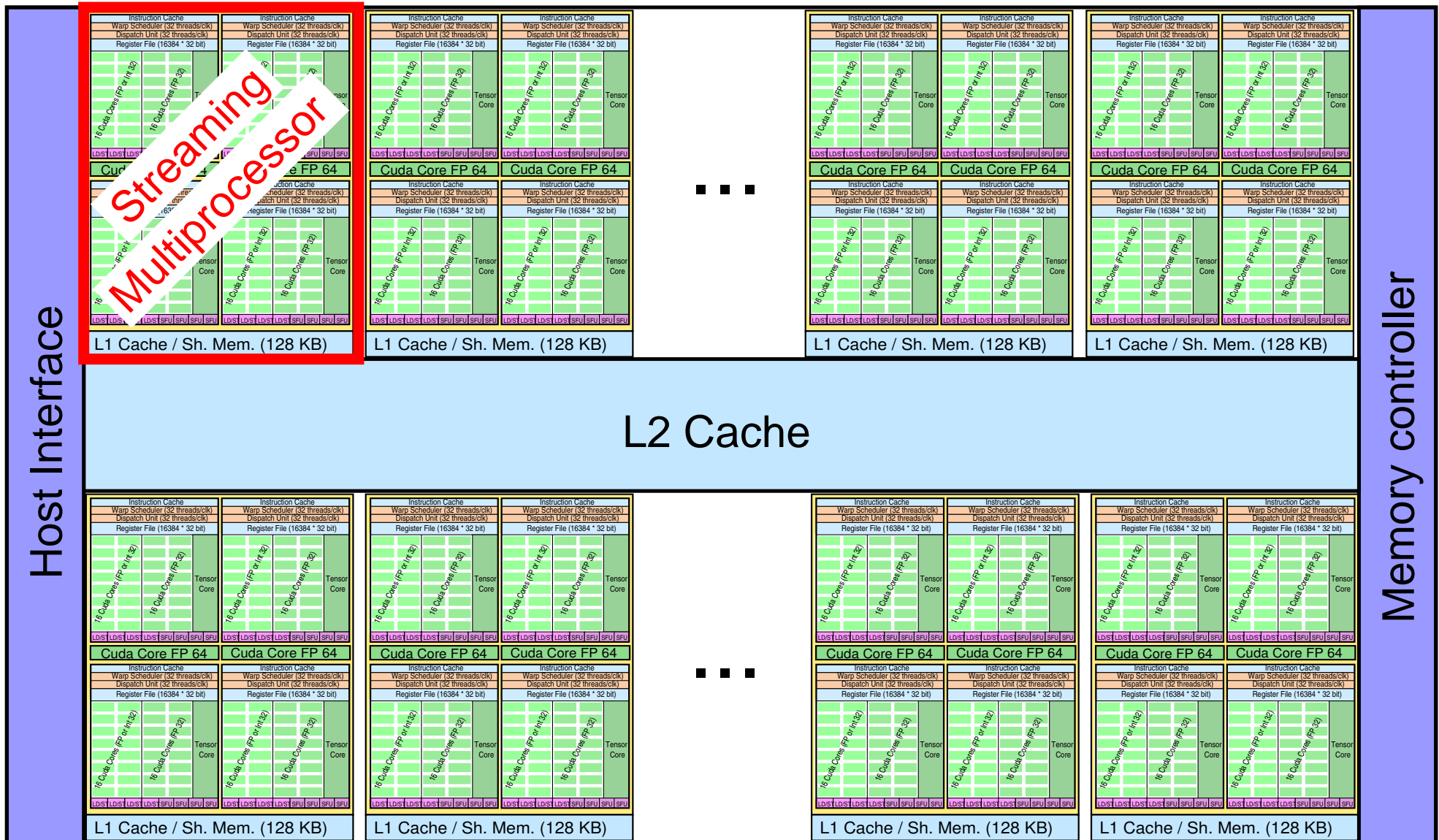
### Example: addition of two vectors

- ➔  $A_j = B_j + C_j$ , for all  $j = 1, \dots, N$
- ➔ AVX and GPU: several elements of the vectors are added **concurrently (in parallel)**
- ➔ if, e.g., four additions can be done at the same time, the following sequence will result:



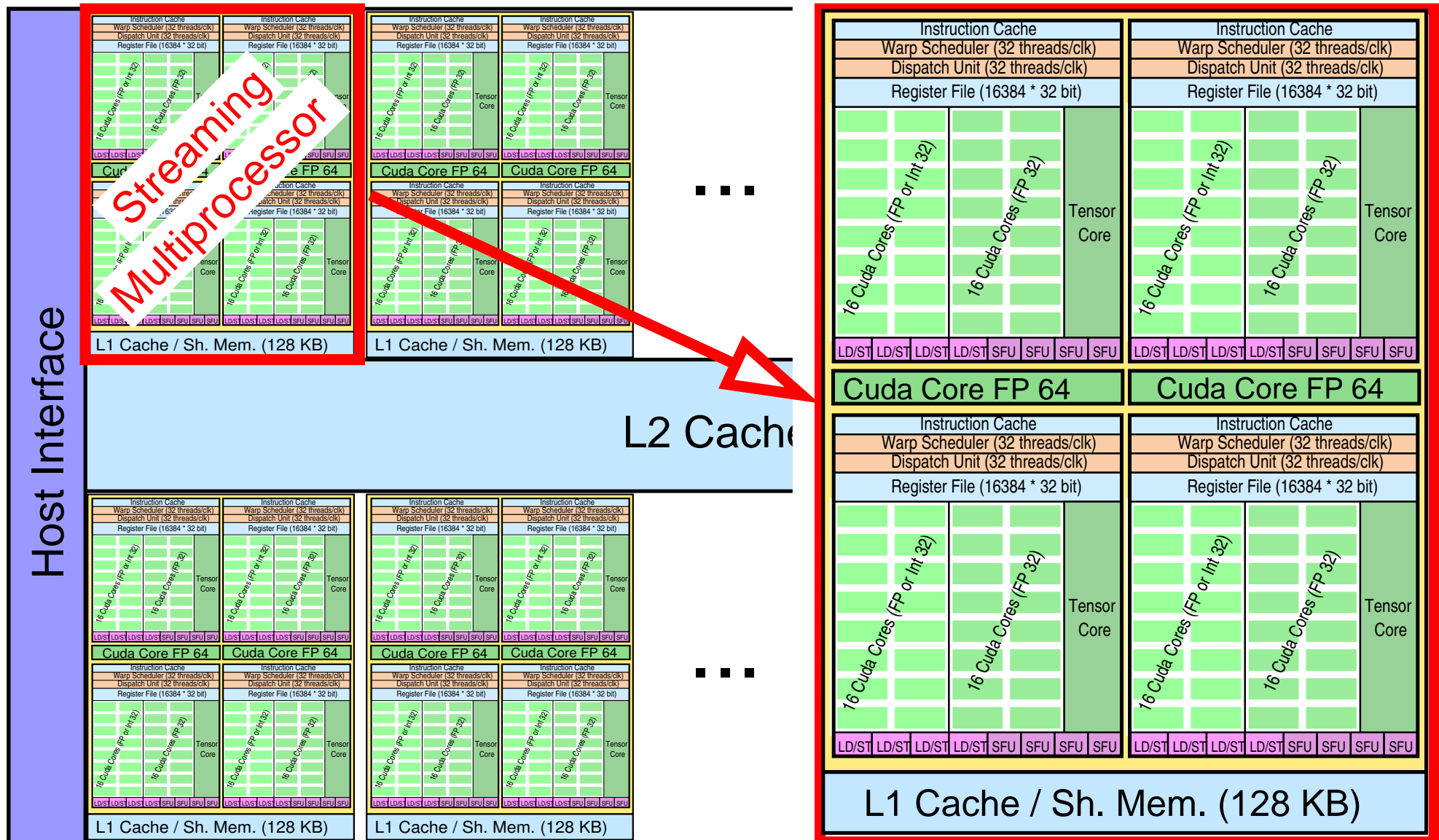


## Architecture of a GPU (NVIDIA Ada)





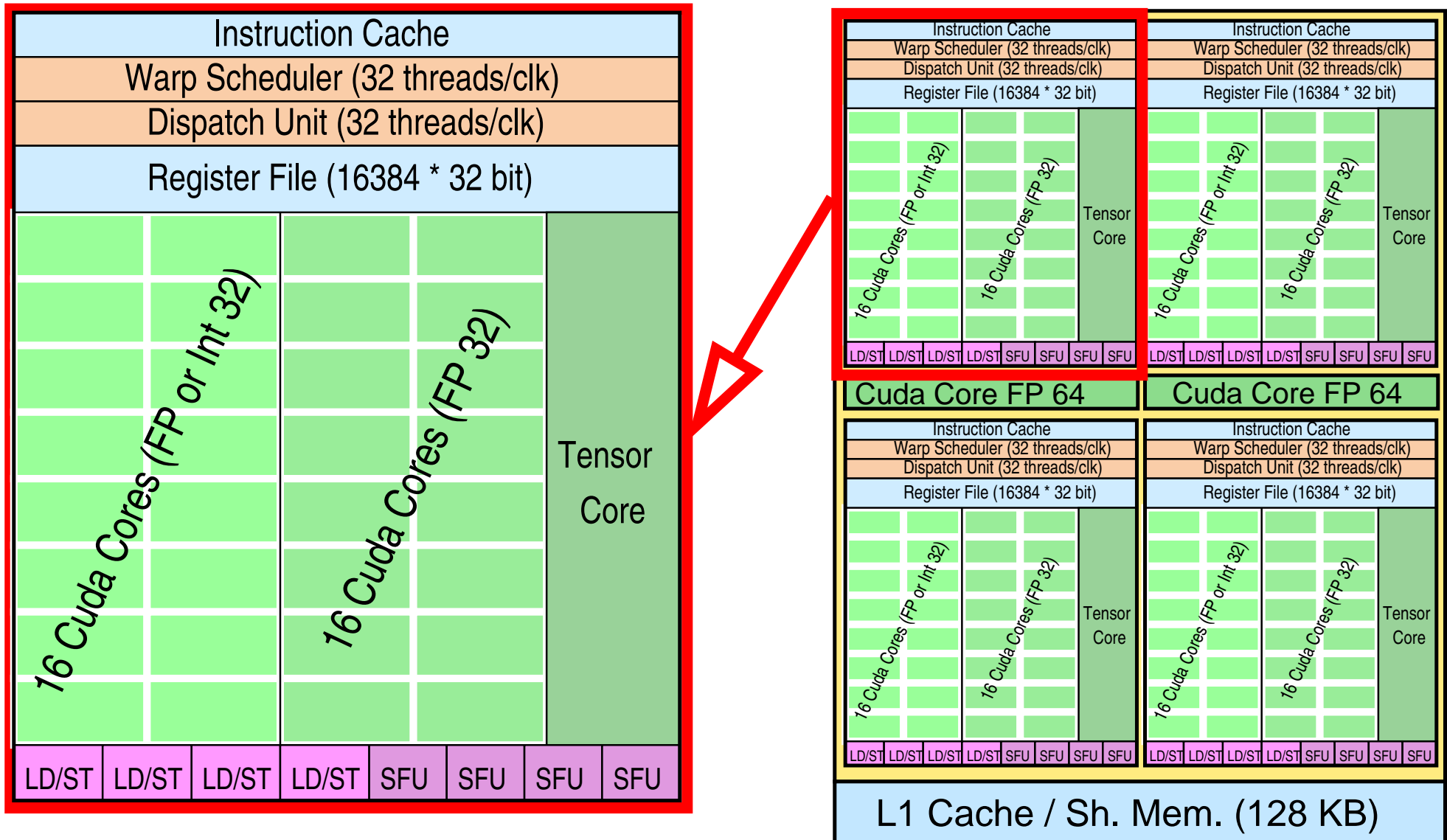
## Architecture of a GPU (NVIDIA Ada)



## 2.4.3 SIMD ...



### Architecture of a GPU (NVIDIA Ada)





### Programming of GPUs (NVIDIA Ada)

- ➔ Partitioning of the code in *blocks* of threads (max. 1024)
- ➔ *Blocks* are distributed between streaming multiprocessors (SMs)
  - ➔ no synchronisation possible between SMs
- ➔ *Warp scheduler* in SM dispatches threads to cores
  - ➔ a *warp* consists of 32 threads
  - ➔ multiple warps per scheduler support latency hiding
    - ➔ one limiting factor: number of registers
  - ➔ on a SM, threads all execute the same instruction (on different data) or none at all ( $\sim$  SIMD)
  - ➔ e.g., with `if-then-else`:
    - ➔ first some cores execute the `then` branch,
    - ➔ then the other cores execute the `else` branch



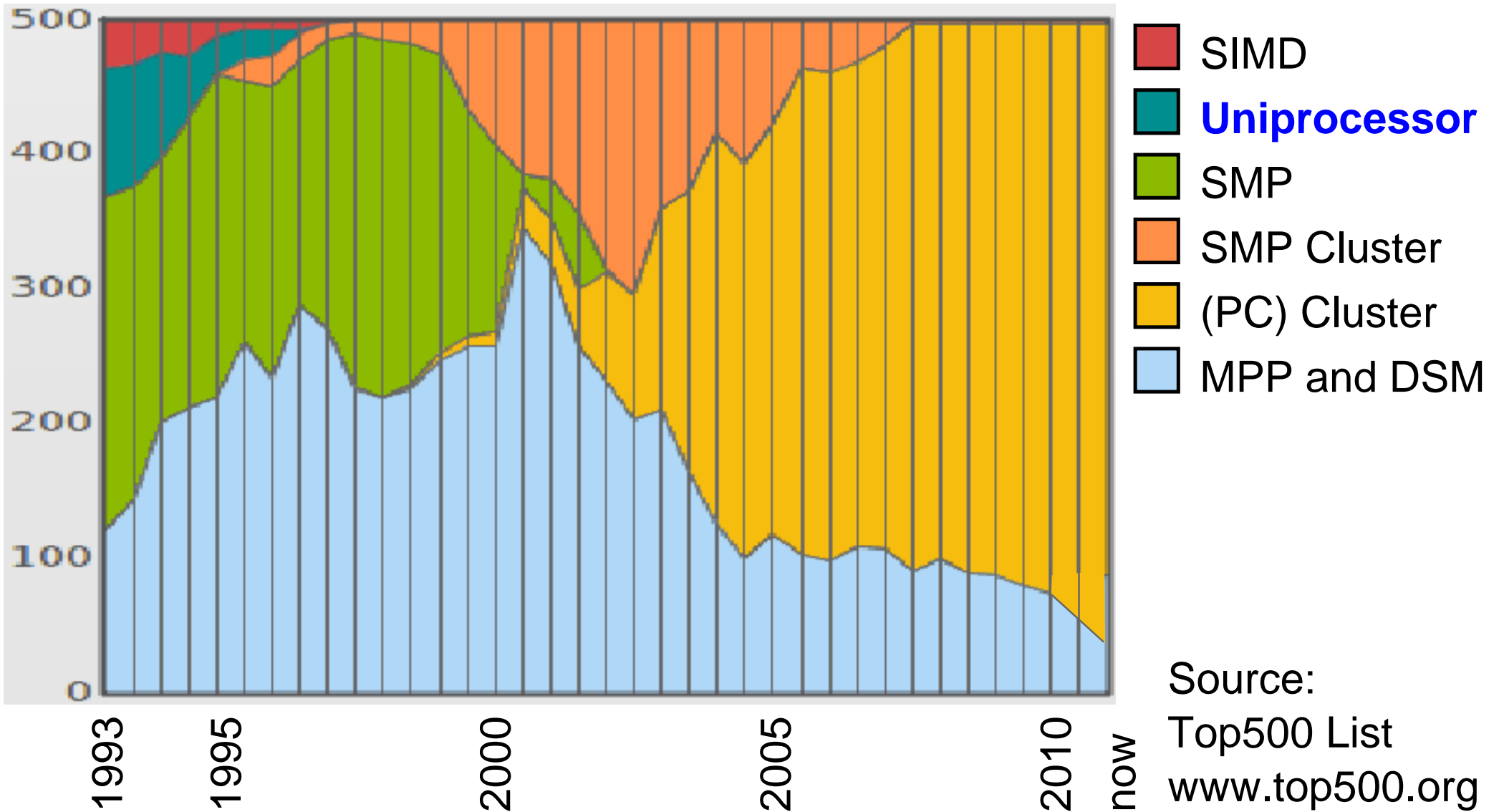
### Programming of GPUs (NVIDIA Ada) ...

- ➔ Threads in one block should address subsequent memory locations
  - ➔ only in this case, memory accesses can be merged
- ➔ 32-Bit FP performance is 64 times the 64-Bit FP performance!

## 2.4.4 High Performance Supercomputers



### Trends





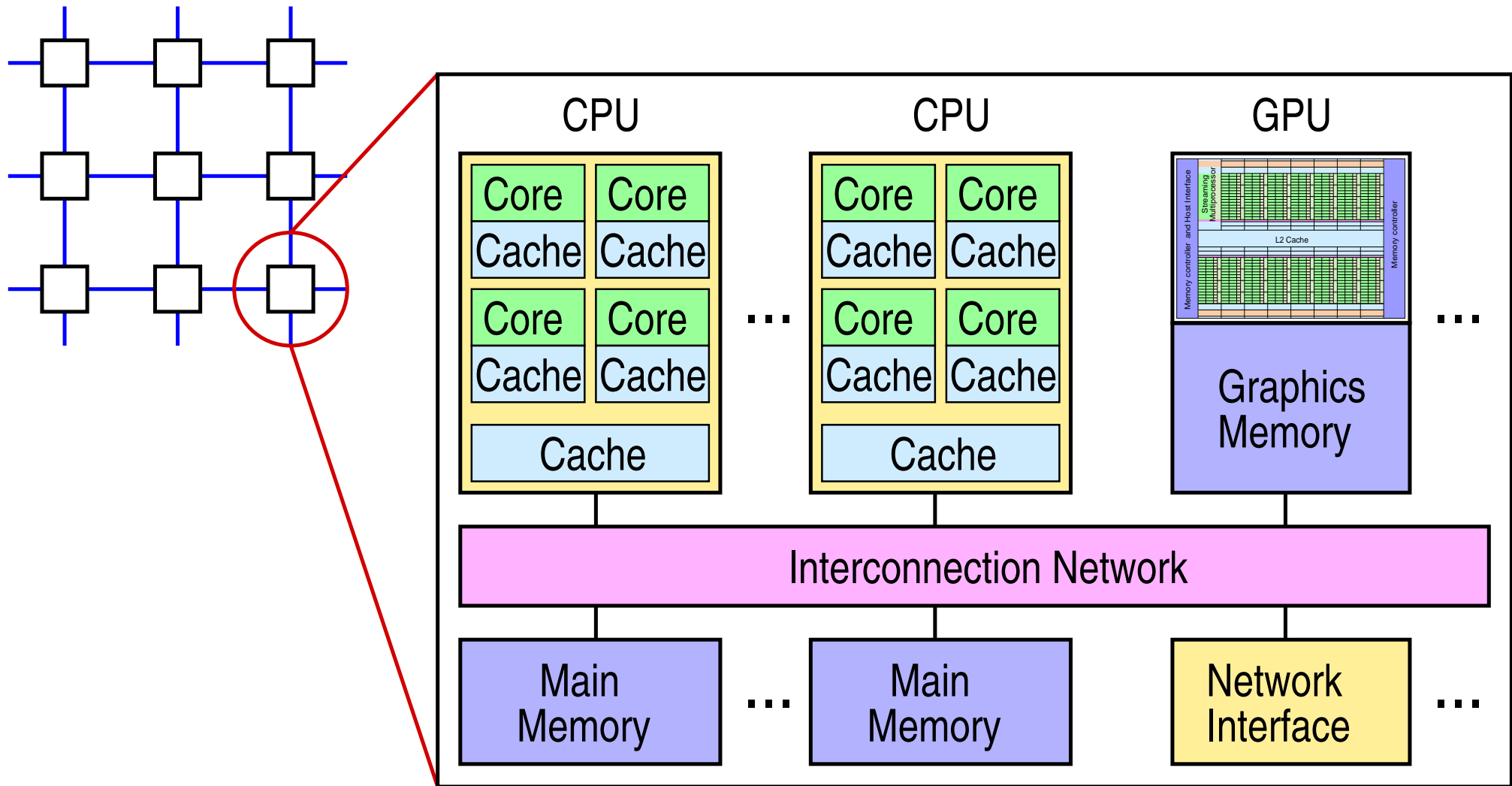
### Typical architecture:

- ➔ Message passing computers with SMP nodes and accelerators (e.g. GPUs)
  - ➔ at the highest layer: systems with distributed memory
  - ➔ nodes: NUMA systems with partially shared cache hierarchy
  - ➔ in addition one or more accelerators per node
- ➔ Compromise between scalability, programmability and performance
- ➔ Programming with hybrid programming model
  - ➔ message passing between the nodes (manually, MPI)
  - ➔ shared memory on the nodes (compiler supported, e.g., OpenMP)
  - ➔ if need be, additional programming model for accelerators

## 2.4.4 High Performance Supercomputers ...



### Typical architecture: ...



## 2.5 Parallel Programming Models



**In the following, we discuss:**

- ➔ Shared memory
  - ➔ Message passing
  - ➔ Distributed objects
  - ➔ Data parallel languages
- ➔ The list is not complete (e.g., data flow models, PGAS)



- ➔ Light weight processes (threads) share a common virtual address space
- ➔ The “more simple” parallel programming model
  - ➔ all threads have access to all data
  - ➔ also good theoretical foundation (PRAM model)
- ➔ Mostly with shared memory computers
  - ➔ however also implementable on distributed memory computers (with large performance penalty)
    - ➔ *shared virtual memory* (SVM)
- ➔ Examples:
  - ➔ PThreads, Java Threads, C++ Threads
  - ➔ Intel Threading Building Blocks (TBB)
  - ➔ OpenMP (👉 **3.1**)

### Example for data exchange

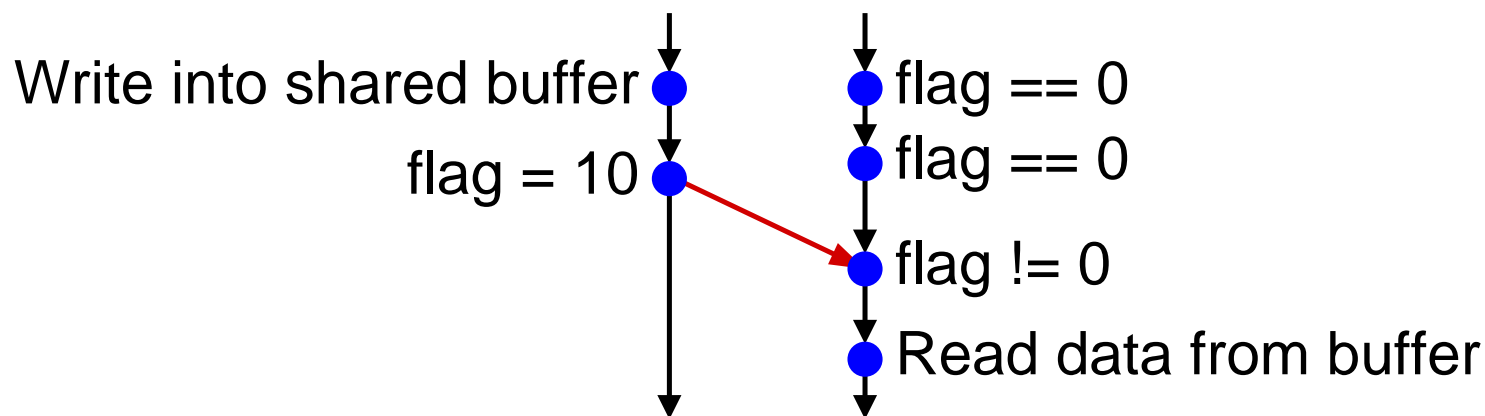
#### Producer Thread

```
for (i=0; i<size; i++)  
    buffer[i] = produce();  
flag = size;
```

#### Consumer Thread


```
while (flag==0);  
for (i=0; i<flag; i++)  
    consume(buffer[i]);
```

#### Execution Sequence:



## 2.5.2 Message Passing



- ➔ Processes with separate address spaces
- ➔ Library routines for sending and receiving messages
  - ➔ (informal) standard for parallel programming:  
MPI (*Message Passing Interface*,  4.2)
- ➔ Mostly with distributed memory computers
  - ➔ but also well usable with shared memory computers
- ➔ The “more complicated” parallel programming model
  - ➔ explicit data distribution / explicit data transfer
  - ➔ typically no compiler and/or language support
    - ➔ parallelisation is done completely manually

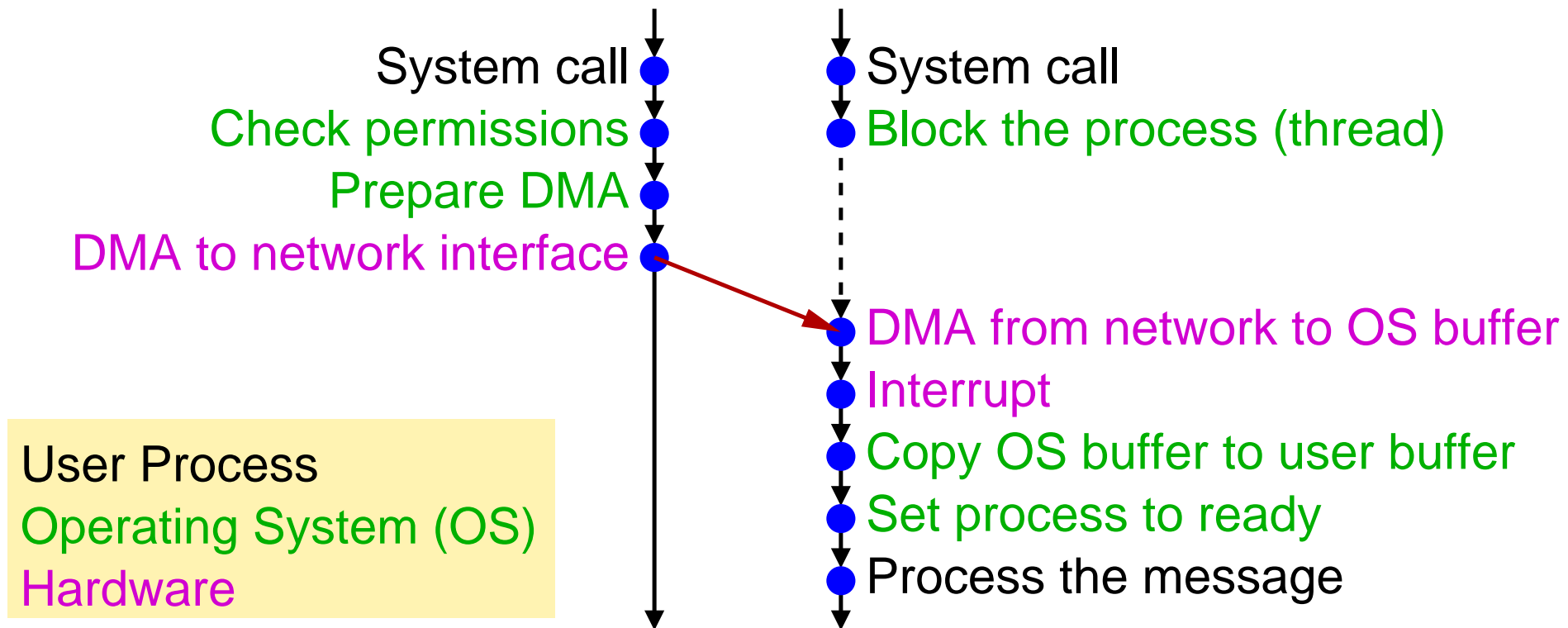
### Example for data exchange

#### Producer Process

```
send(receiver,  
      &buffer, size);
```

#### Consumer Process

```
receive(&buffer,  
       buffer_length);
```



## 2.5.3 Distributed Objects



- ➔ Basis: (purely) object oriented programming
  - ➔ access to data **only** via method calls
- ➔ Then: objects can be distributed to different address spaces (computers)
  - ➔ at object creation: additional specification of a node
  - ➔ object reference then also identifies this node
  - ➔ method calls via RPC mechanism
    - ➔ e.g., *Remote Method Invocation* (RMI) in Java
  - ➔ more about this: lecture “Distributed Systems”
- ➔ Distributed objects alone do not yet enable parallel processing
  - ➔ additional concepts / extensions are necessary
    - ➔ e.g., threads, asynchronous RPC, futures



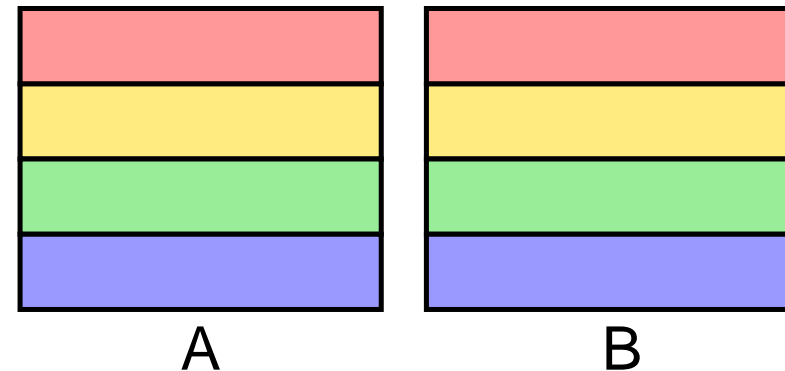
- ➔ Goal: support for data parallelism
- ➔ Sequential code is amended with compiler directives
  - ➔ Specification, how to distribute data structures (typically arrays) to processors
- ➔ Compiler automatically generates code for synchronisation or communication, respectively
  - ➔ operations are executed on the processor that “possesses” the result variable (*owner computes* rule)
- ➔ Example: HPF (*High Performance Fortran*)
- ➔ Despite easy programming not really successful
  - ➔ only suited for a limited class of applications
  - ➔ good performance requires a lot of manual optimization

### Example for HPF

```
REAL A(N,N), B(N,N)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(:, :) WITH A(:, :)

DO I = 1, N
  DO J = 1, N
    A(I, J) = A(I, J) + B(J, I)
  END DO
END DO
```

Distribution with 4 processors:



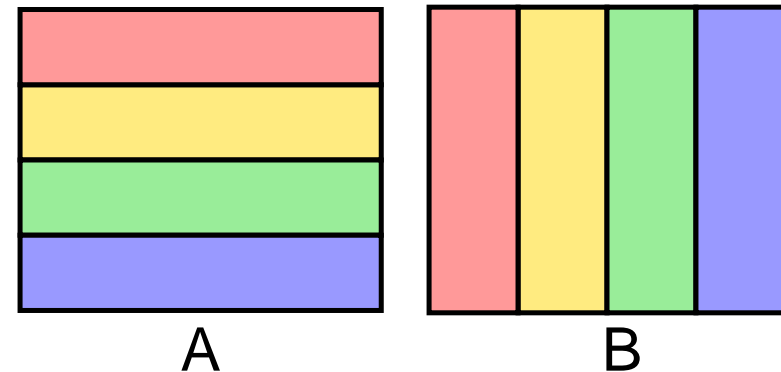
- ➔ Processor 0 executes computations for  $I = 1 .. N/4$
- ➔ Problem in this example: a lot of communication is required
  - ➔ B should be distributed in a different way

### Example for HPF

```
REAL A(N,N), B(N,N)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(j,i) WITH A(i,j)

DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I,J) + B(J,I)
  END DO
END DO
```

Distribution with 4 processors:



- ➔ Processor 0 executes computations for  $I = 1 .. N/4$
- ➔ No communication is required any more
  - ➔ but B must be redistributed, if necessary

## 2.6 Focus of this Lecture

---



- ➔ Explicit parallelism
- ➔ Process and block level
- ➔ Coarse and mid grained parallelism
  
- ➔ MIMD computers (with SIMD extensions)
  
- ➔ Programming models:
  - ➔ shared memory
  - ➔ message passing



- ➔ Models / patterns for parallel programs

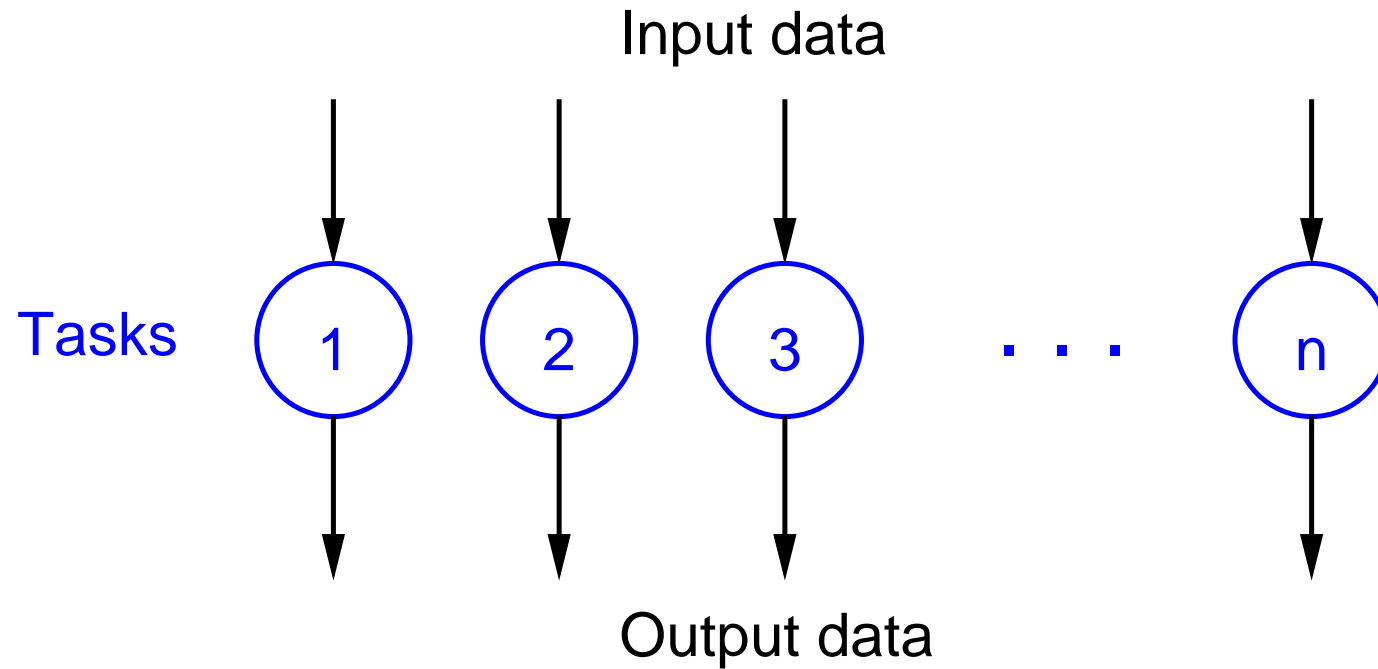
### 2.7.1 Embarrassingly Parallel

- ➔ The task to be solved can be divided into a set of **completely independent** sub-tasks
- ➔ All sub-tasks can be solved in parallel
- ➔ No data exchange (communication) is necessary between the parallel threads / processes
- ➔ Ideal situation!
  - ➔ when using  $n$  processors, the task will (usually) be solved  $n$  times faster
  - ➔ (for reflection: why only usually?)

## 2.7.1 Embarrassingly Parallel ...



### Illustration



---

# Parallel Processing

Winter Term 2025/26

28.10.2025

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 10, 2025



### Examples for embarrassingly parallel problems

- ➔ Computation of animations
  - ➔ 3D visualizations, animated cartoons, motion pictures, ...
  - ➔ each image (frame) can be computed independently
- ➔ Parameter studies
  - ➔ multiple / many simulations with different input parameters
  - ➔ e.g., weather forecast with provision for measurement errors, computational fluid dynamics for optimizing an airfoil, ...

### 2.7.2 Manager/Worker Model (Master/Slave Model)

➔ A manager process creates independent tasks and assigns them to worker processes

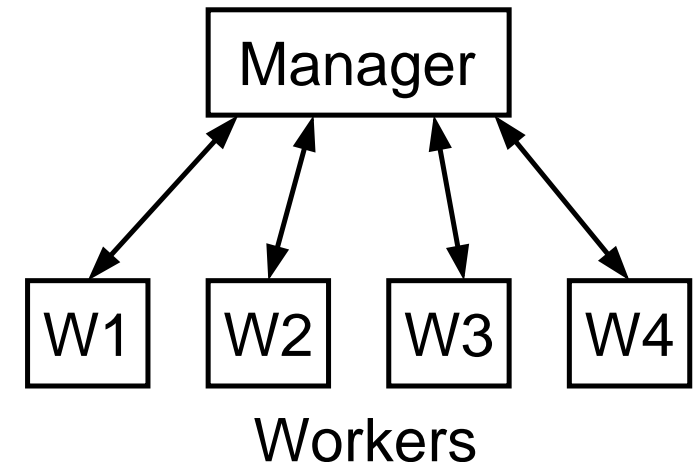
➔ several managers are possible, too

➔ a hierarchy is possible, too: a worker can itself be the manager of own workers

➔ The manager (or sometimes also the workers) can create additional tasks, while the workers are working

➔ The manager can become a bottleneck

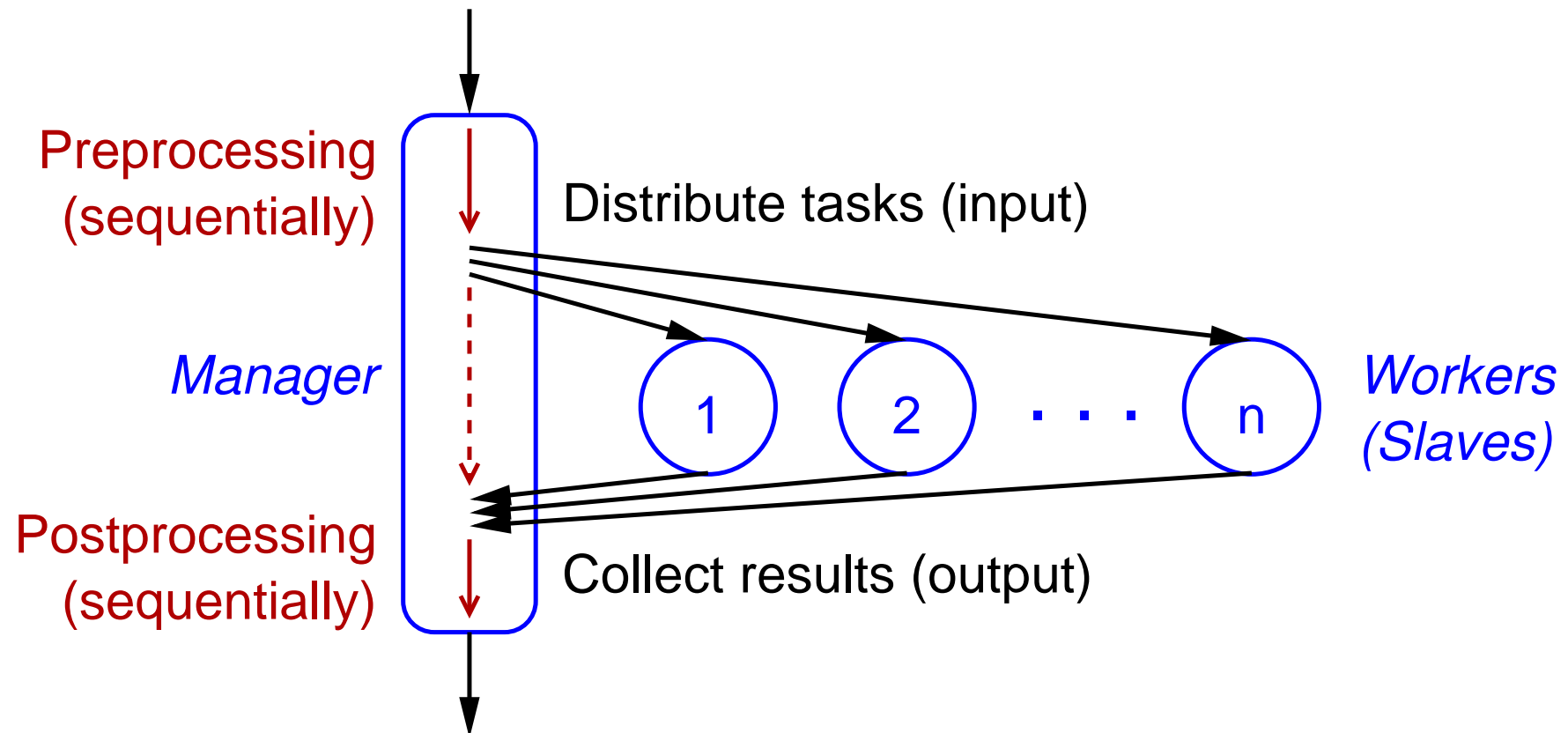
➔ The manager should be able to receive the results asynchronously (non blocking)





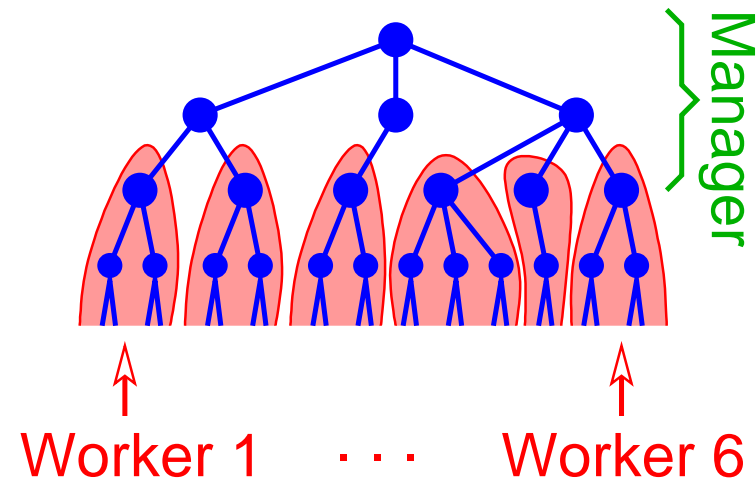
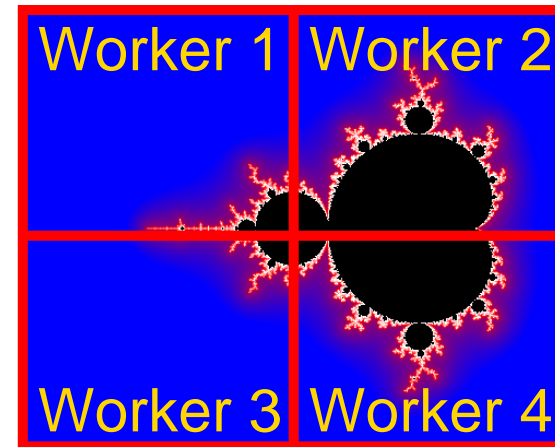
### Typical application

- ➔ Often only a part of a task can be parallelised in an optimal way
- ➔ In the easiest case, the following flow will result:



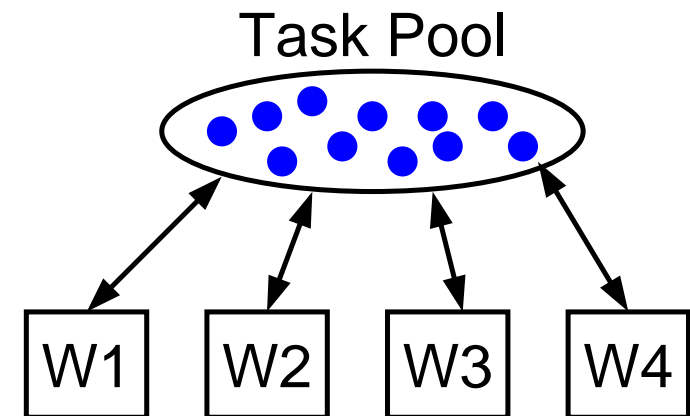
### Examples

- ➔ Image creation and processing
  - ➔ manager partitions the image into areas; each area is processed by one worker
- ➔ Tree search
  - ➔ manager traverses the tree up to a predefined depth; the workers process the sub-trees



### 2.7.3 Work Pool Model (Task Pool Model)

- ➔ Tasks are explicitly specified using a data structure
  - ➔ input data + task description, if necessary
- ➔ Centralized or distributed pool (list) of tasks
  - ➔ workers (threads or processes) fetch tasks from the pool
    - ➔ usually much more tasks than workers
    - ➔ good load balancing is possible
    - ➔ accesses must be synchronised
- ➔ Workers can put new tasks into the pool, if need be
  - ➔ e.g., with divide-and-conquer





### 2.7.4 Divide and Conquer

- ➔ **Recursive** partitioning of the task into independent sub-tasks
- ➔ Tasks dynamically create new sub-tasks
- ➔ Problem: limiting the number of tasks
  - ➔ esp. if tasks are directly implemented by threads / processes
- ➔ Solutions:
  - ➔ create a new sub-task only, if its size is larger than some minimum
  - ➔ maintain a task pool, which is executed by a fixed number of threads



### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and

$A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )

and Qsort( $A_k .. n$ )

in parallel.



### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and

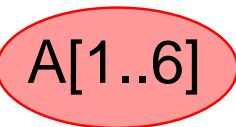
$A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )

and Qsort( $A_k .. n$ )

in parallel.

Example:

 Thread 1



### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and

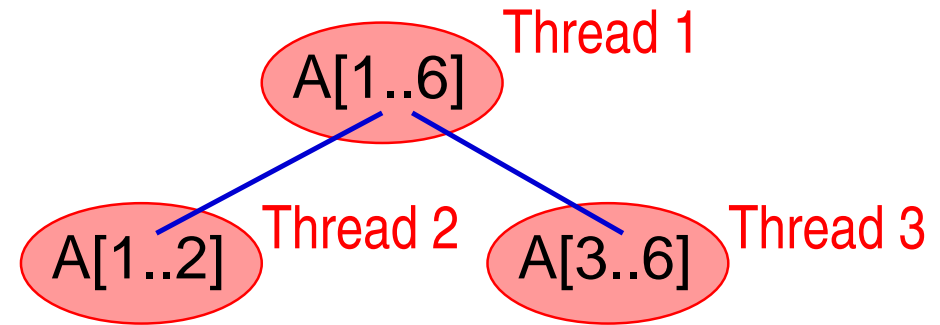
$A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )

and Qsort( $A_k .. n$ )

in parallel.

Example:





### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and

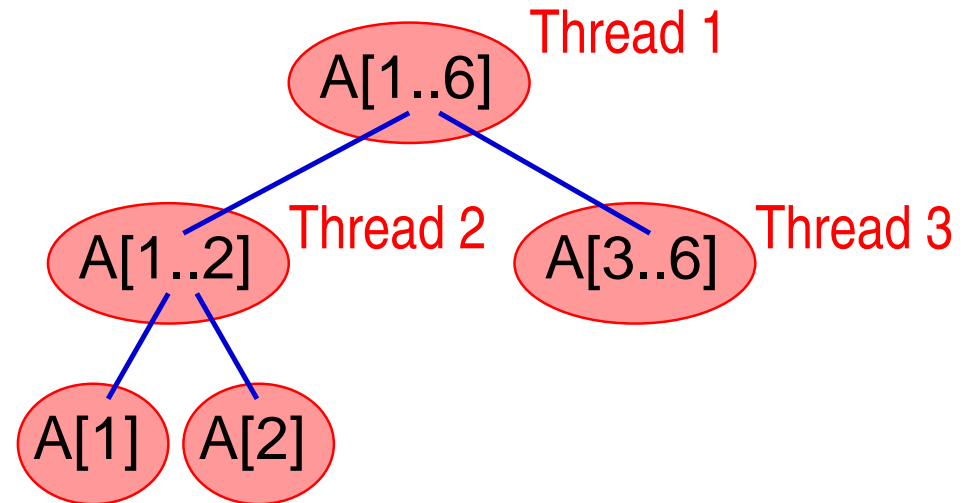
$A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )

and Qsort( $A_k .. n$ )

in parallel.

Example:





### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

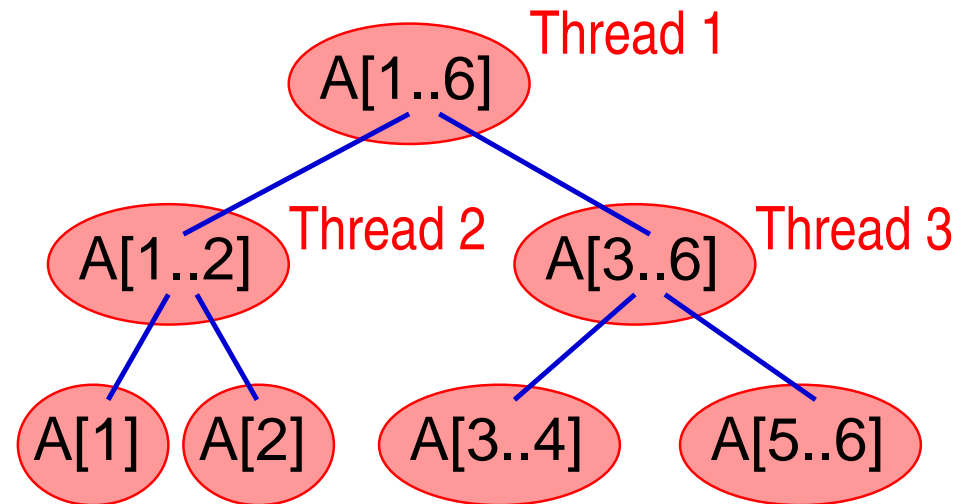
Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and  
 $A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )  
and Qsort( $A_k .. n$ )  
in parallel.

Example:





### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

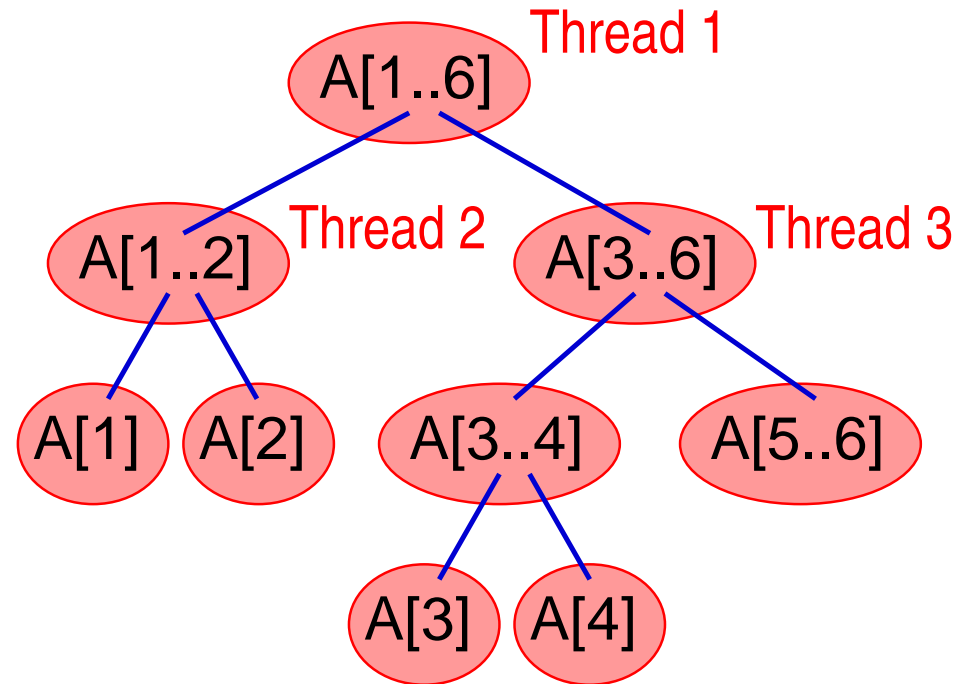
Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and  
 $A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )  
and Qsort( $A_k .. n$ )  
in parallel.

Example:





### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

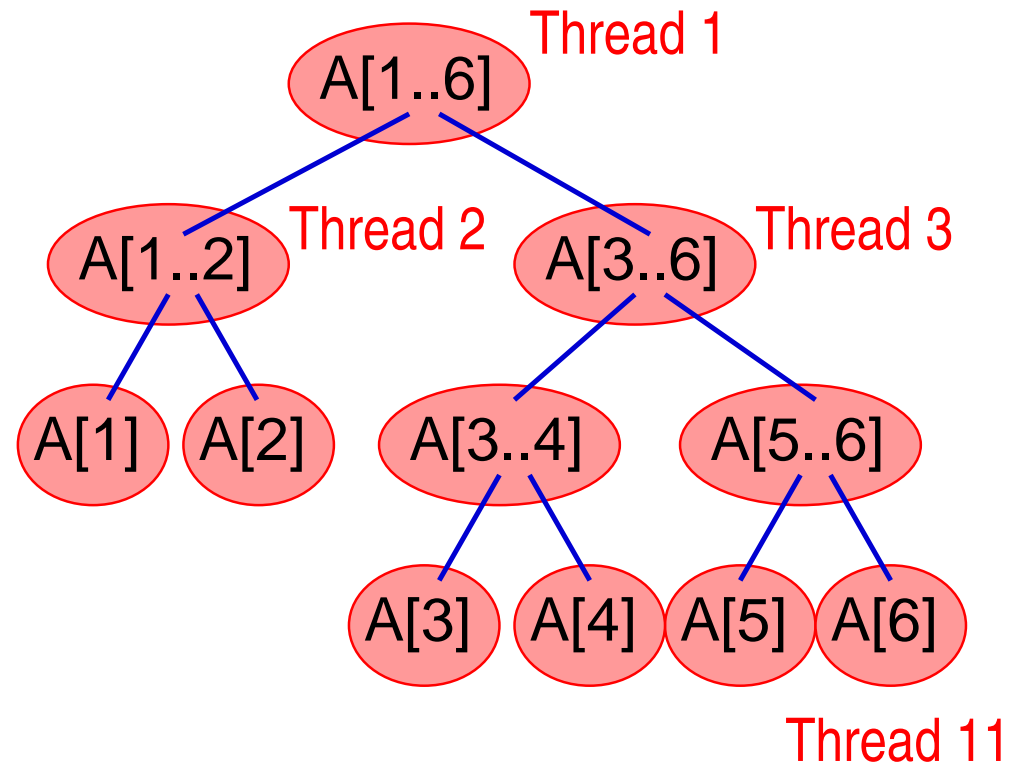
Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and  
 $A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )  
and Qsort( $A_k .. n$ )  
in parallel.

Example:



### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

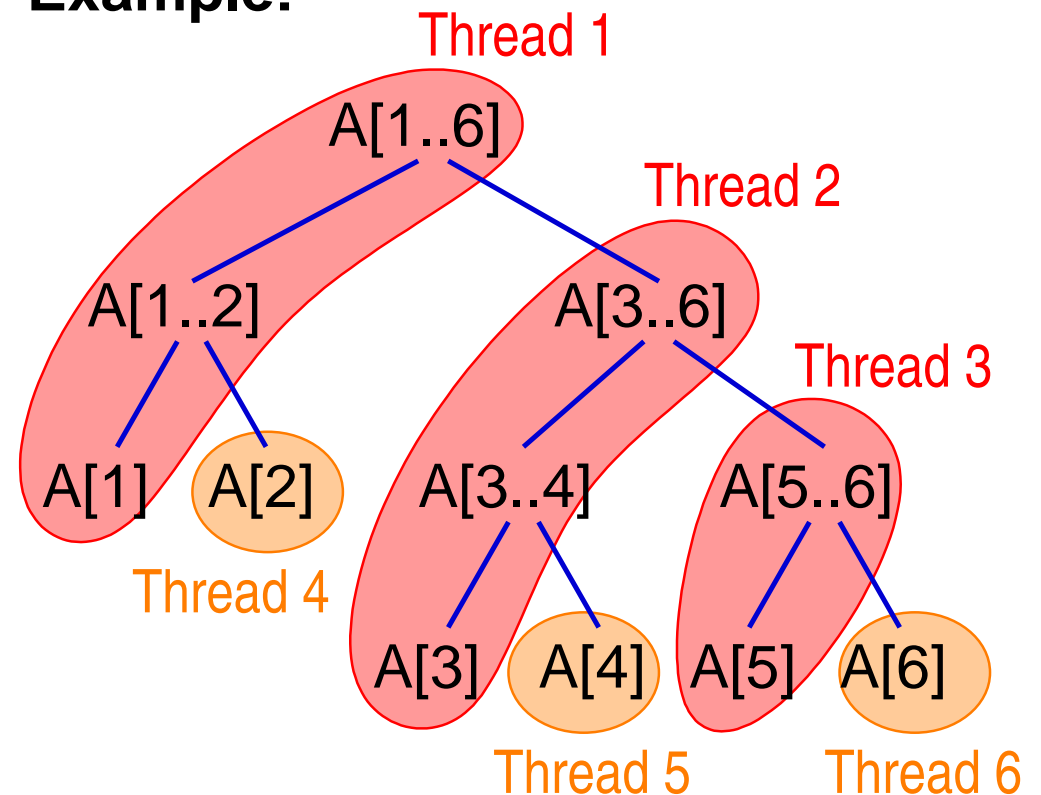
Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and  
 $A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )  
and Qsort( $A_k .. n$ )  
in parallel.

Example:



\* Assumption: thread executes first call itself and creates new thread for the second one

### Example: parallel quicksort

Qsort( $A_1 .. n$ )

If  $n = 1$ : done.

Else:

Determine the *pivot*  $S$ .

Reorder  $A$  such that

$A_i \leq S$  for  $i \in [1, k[$  and

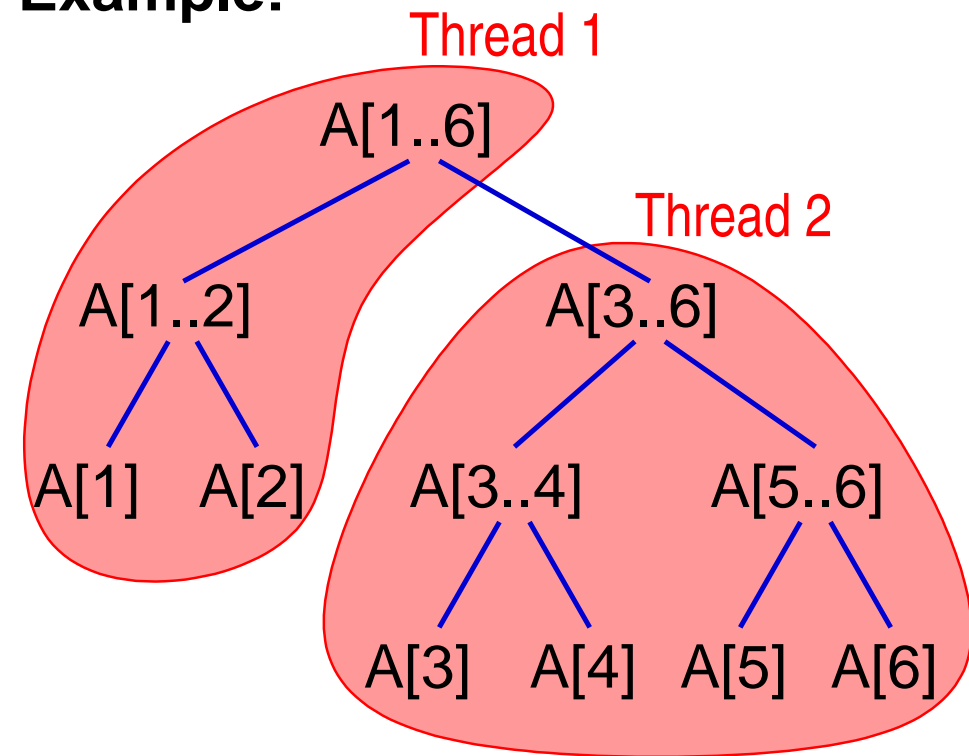
$A_i \geq S$  for  $i \in [k, n]$ .

Execute Qsort( $A_1 .. k-1$ )

and Qsort( $A_k .. n$ )

in parallel.

Example:



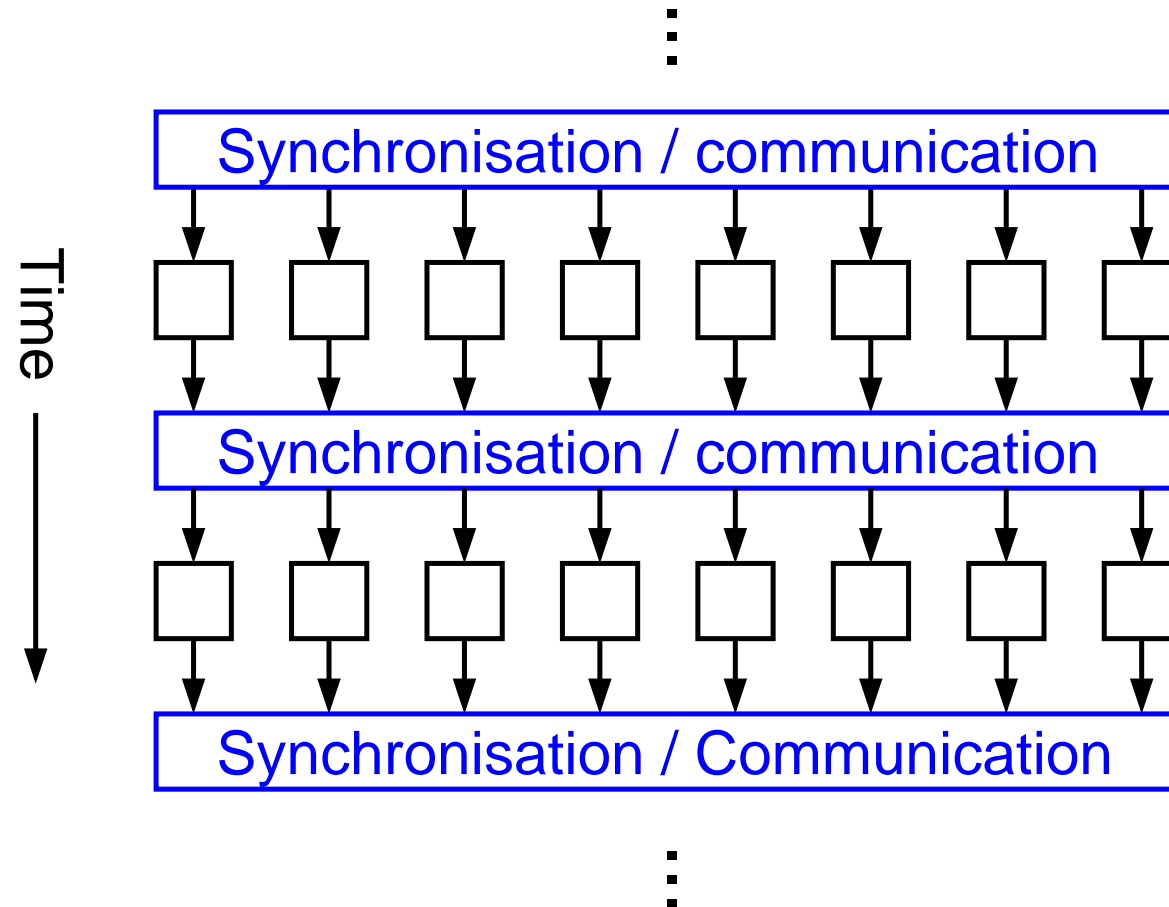
\* Additional Assumption: new thread is created only if array length > 2



### 2.7.5 Data parallel Model: SPMD

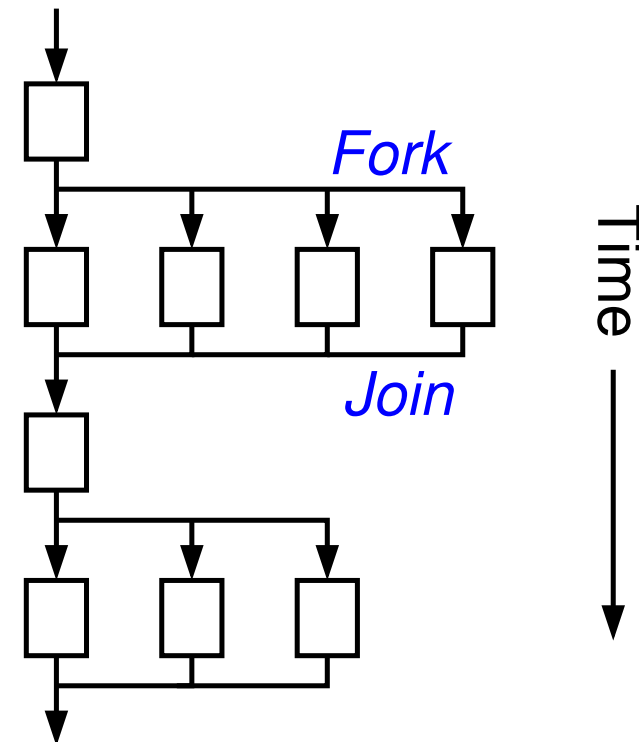
- ➔ Fixed, constant number of processes (or threads, respectively)
- ➔ One-to-one correspondence between tasks and processes
- ➔ All processes execute the same program code
  - ➔ however: conditional statements are possible ...
- ➔ For program parts which cannot be parallelised:
  - ➔ replicated execution in each process
  - ➔ execution in only one process; the other ones wait
- ➔ Usually loosely synchronous execution:
  - ➔ alternating phases with independent computations and communication / synchronisation

### Typical sequence



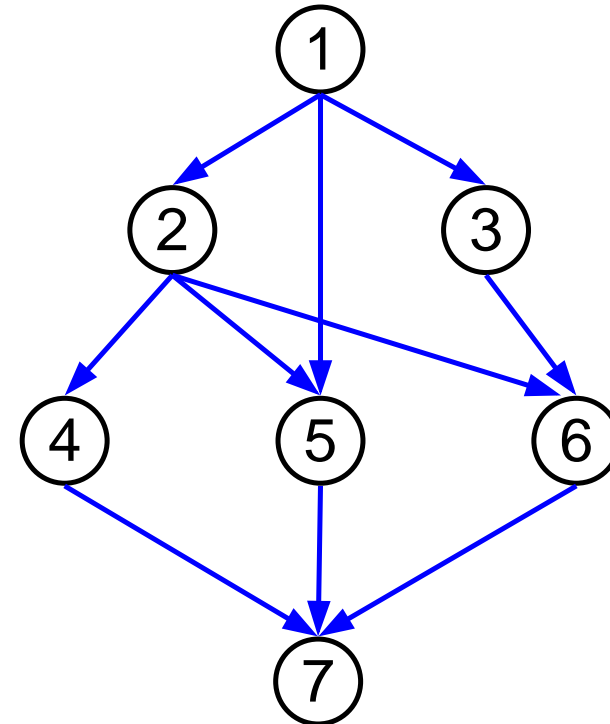
### 2.7.6 Fork/Join Model

- ➔ Program consists of sequential and parallel phases
- ➔ Thread (or processes, resp.) for parallel phases are created at run-time (*fork*)
  - ➔ one for each task
- ➔ At the end of each parallel phase: synchronisation and termination of the threads (*join*)



### 2.7.7 Task-Graph Model

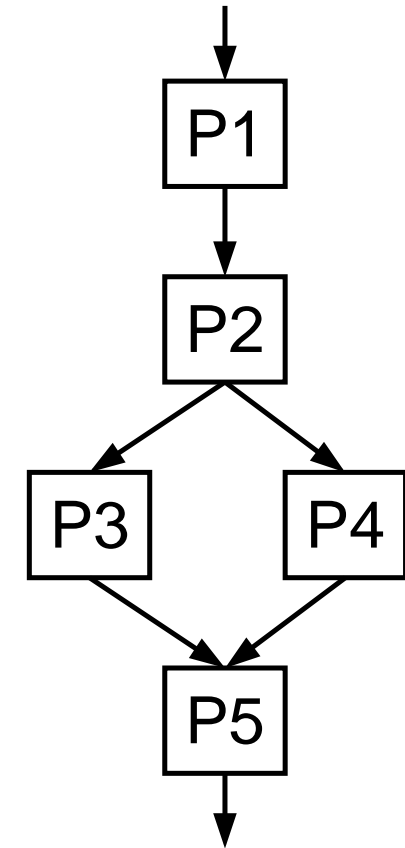
- ➔ Tasks and their dependences (data flow) are represented as a graph
- ➔ An edge in the graph denotes a data flow
  - ➔ e.g., task 1 produces data, task 2 starts execution, when this data is entirely available



- ➔ Assignment of tasks to processors usually in such a way, that the necessary amount of communication is as small as possible
  - ➔ e.g., tasks 1, 5, and 7 in one process

### 2.7.8 Pipeline Model

- ➔ A *stream* of data elements is directed through a sequence of processes
- ➔ The execution of a task starts as soon as a data element arrives
- ➔ Pipeline needs not necessarily be linear
  - ➔ general (acyclic) graphs are possible, as with the task-graph model
- ➔ Producer/consumer synchronisation between the processes





### Four design steps:

#### 1. Partitioning

- ➔ split the problem into many tasks

#### 2. Communication

- ➔ specify the information flow between the tasks
- ➔ determine the communication structure

#### 3. Agglomeration

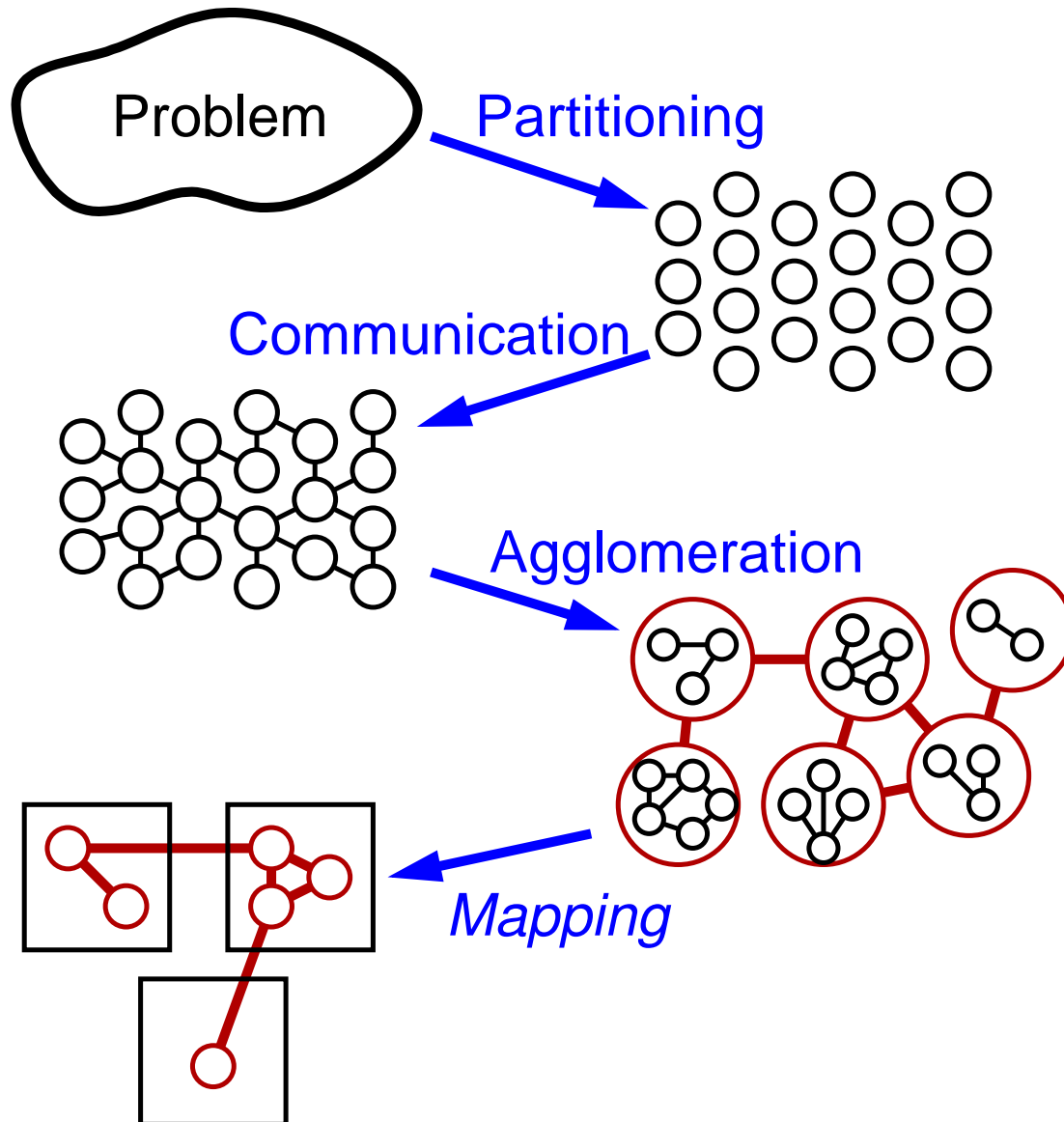
- ➔ evaluate the performance (tasks, communication structure)
- ➔ if need be, aggregate tasks into larger tasks

#### 4. Mapping

- ➔ map the tasks to processors

(See Foster: *Designing and Building Parallel Programs*, Ch. 2)

## 2.8 A Design Process for Parallel Programs ...

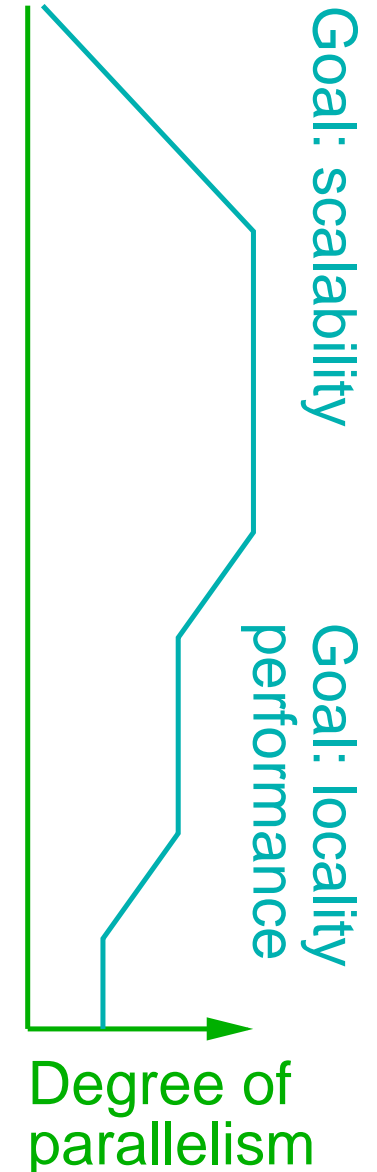


Split the problem in as many small tasks as possible

Data exchange between tasks

Merging of Tasks

Mapping to Processors



- ➔ Goal: split the problem into as many small tasks as possible

### Data partitioning (data parallelism)

- ➔ Tasks specify **identical computations** for a **part** of the data
- ➔ In general, high degree of parallelism is possible
- ➔ We can distribute:
  - ➔ input data
  - ➔ output data
  - ➔ intermediate data
- ➔ In some cases: recursive partitioning (*divide and conquer*)
- ➔ Special case: partitioning of search space in search problems

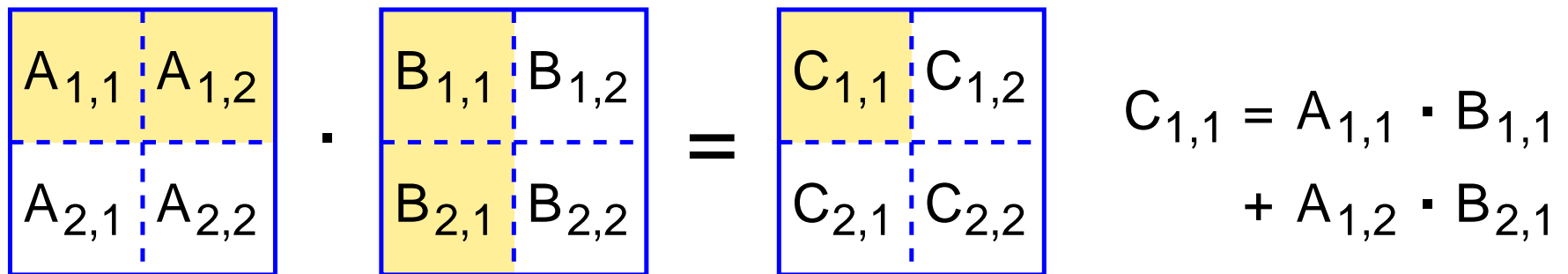
### Example: matrix multiplication

➔ Product  $C = A \cdot B$  of two square matrices

➔ 
$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}, \text{ for all } i, j = 1 \dots n$$

➔ This formula also holds when square sub-matrices  $A_{ik}$ ,  $B_{kj}$ ,  $C_{ij}$  are considered instead of single scalar elements

➔ block matrix algorithms:


$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \cdot \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$
$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

### Example: matrix multiplication ...

- ➔ Distribution of output data: each task computes a sub-matrix of  $C$
- ➔ E.g., distribution of  $C$  into four sub-matrices

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

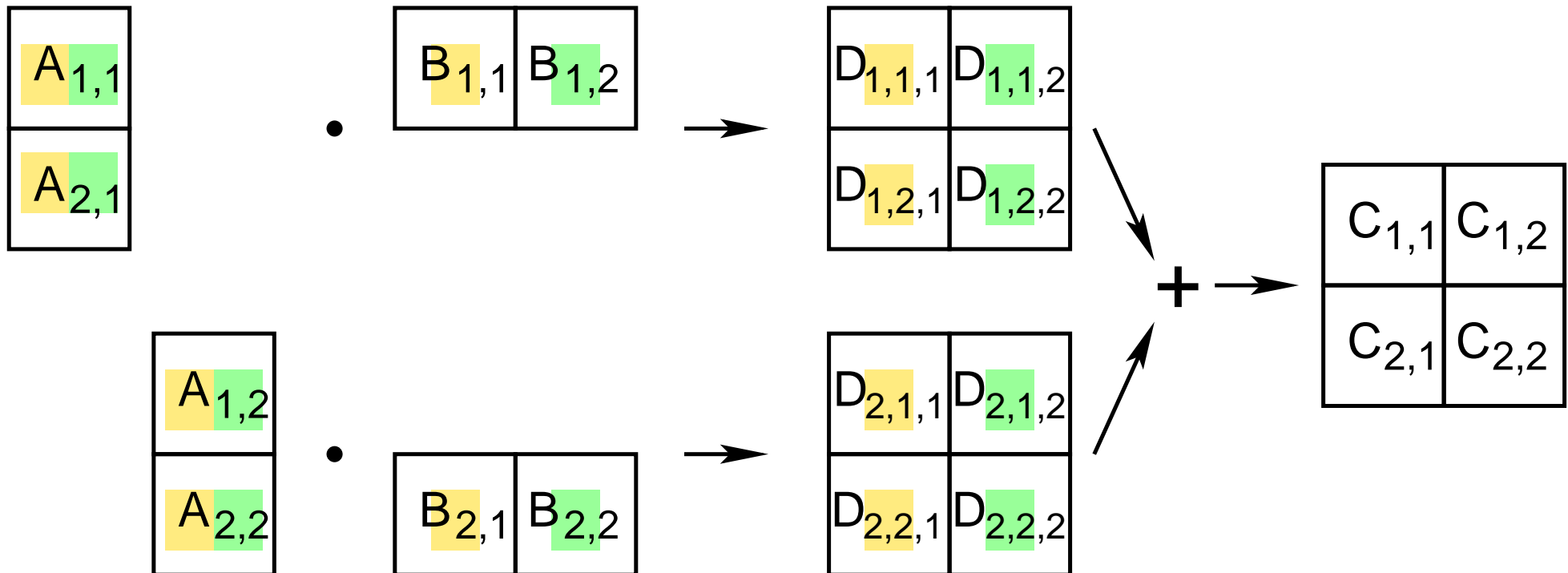
- ➔ Results in four independent tasks:
  1.  $C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$
  2.  $C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$
  3.  $C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$
  4.  $C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$

## 2.8.1 Partitioning ...



### Example: matrix multiplication $A \cdot B \rightarrow C$

- ➔ Distribution of intermediate data (higher degree of parallelism)
- ➔ here: 8 multiplications of sub-matrices

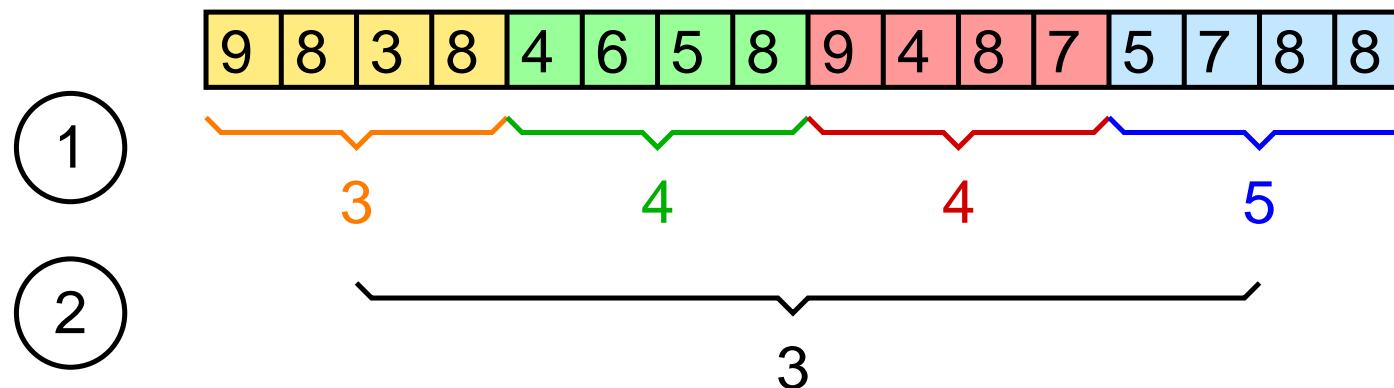


## 2.8.1 Partitioning ...



### Example: minimum of an array

- ➔ Distribution of input data
  - ➔ each threads computates its local minimum
  - ➔ afterwards: computation of the global minimum



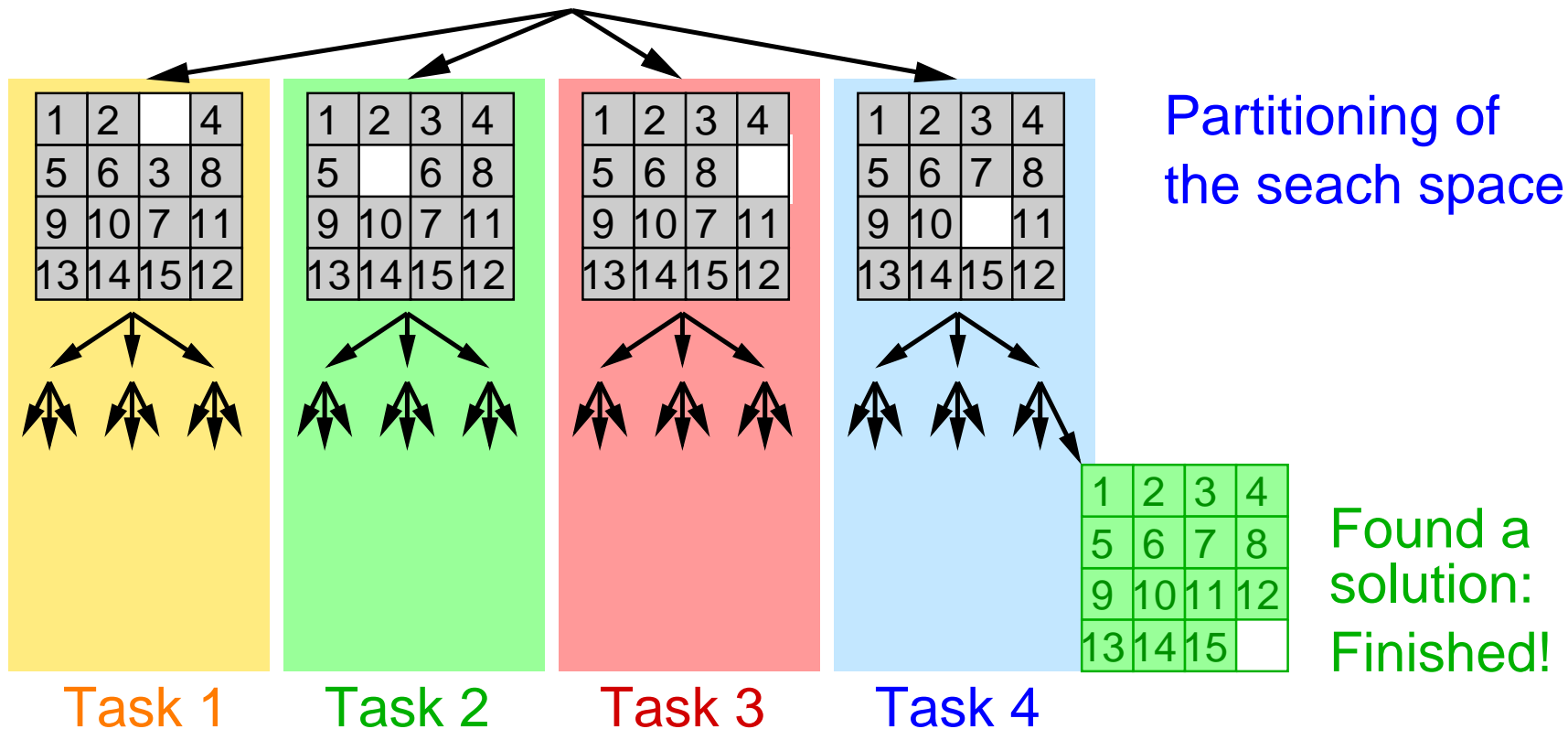
## 2.8.1 Partitioning ...



**Example: sliding puzzle** (partitioning of search space)

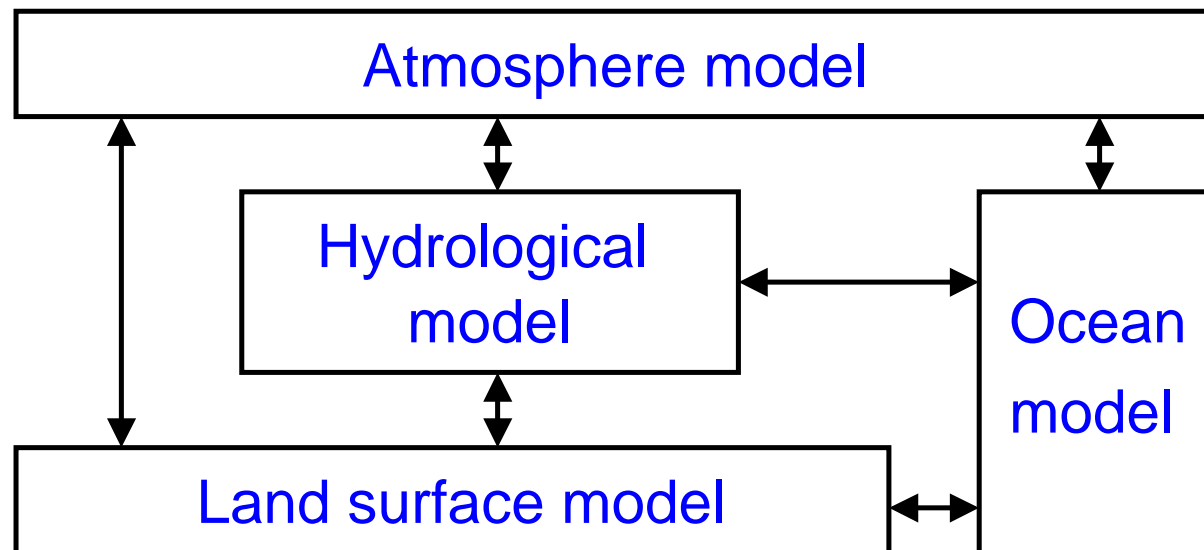
1	2	3	4
5	6	*	8
9	10	7	11
13	14	15	12

Goal: find a sequence of moves, which leads to a sorted configuration



### Task partitioning (task parallelism)

- ➔ Tasks are **different** sub-problems (execution steps) of a problem
- ➔ E.g., climate model



- ➔ Tasks can work concurrently or in a pipeline
- ➔ max. gain: number of sub-problems (typically small)
- ➔ often in addition to data partitioning



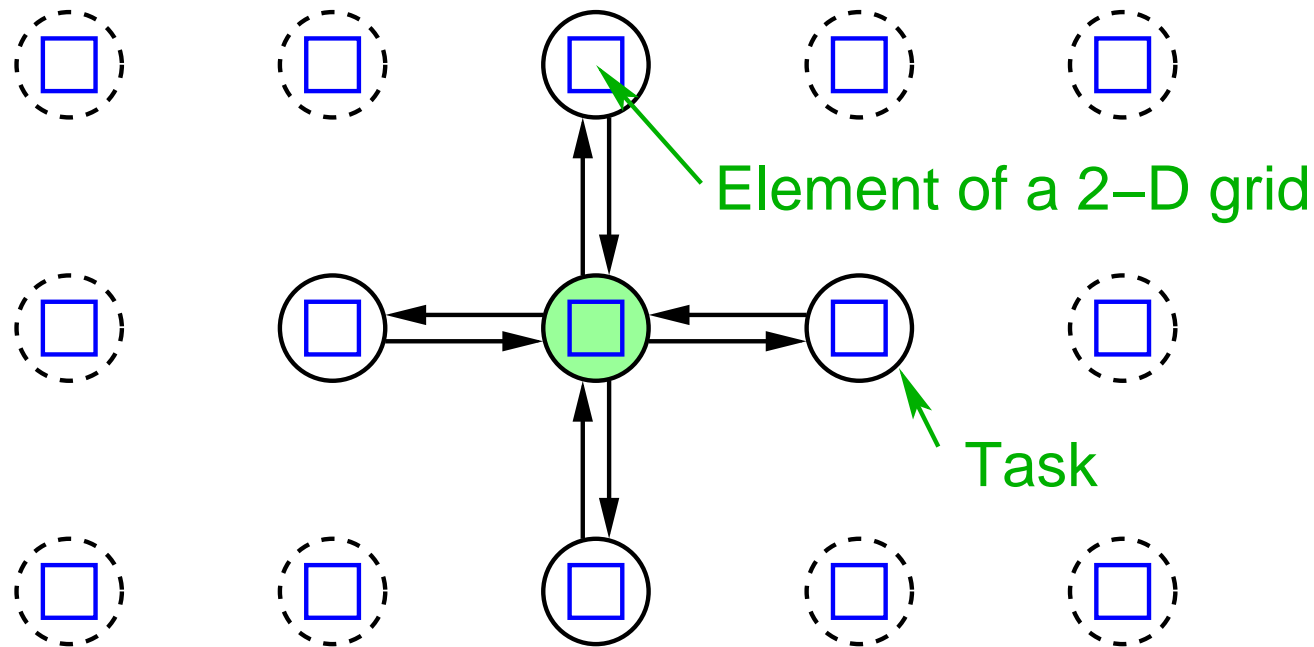
- ➔ Two step approach
  - ➔ definition of the communication structure
    - ➔ who must exchange data with whom?
    - ➔ sometimes complex when using data partitioning
    - ➔ often simple when using task partitioning
  - ➔ definition of the messages to be sent
    - ➔ which data must be exchanged when?
    - ➔ taking data dependences into account



### Different communication patterns:

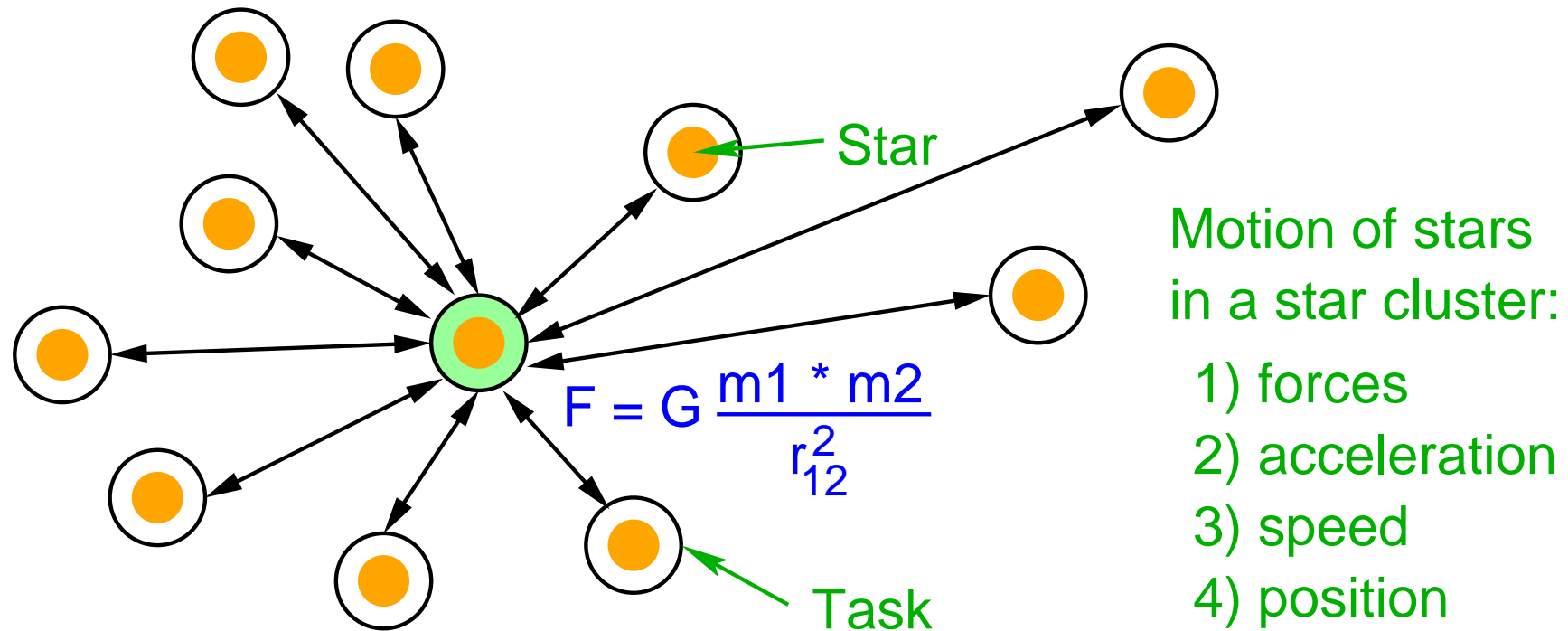
- ➔ Local vs. global communication
  - ➔ lokal: task communicates only with a small set of other tasks (its “neighbors”)
  - ➔ global: task communicates with many/all other tasks
- ➔ Structured vs. unstructured communication
  - ➔ structured: regular structure, e.g., grid, tree
- ➔ Static vs. dynamic communication
  - ➔ dynamic: communication structure is changing during run-time, depending on computed data
- ➔ Synchronous vs. asynchronous communication
  - ➔ asynchronous: the task owning the data does not know, when other tasks need to access it

### Example for local communication: stencil algorithms



- ➔ Here: 5-point stencil (also others are possible)
- ➔ Examples: Jacobi or Gauss-Seidel methods, filters for image processing, ...

### Example for global communication: N-body problem

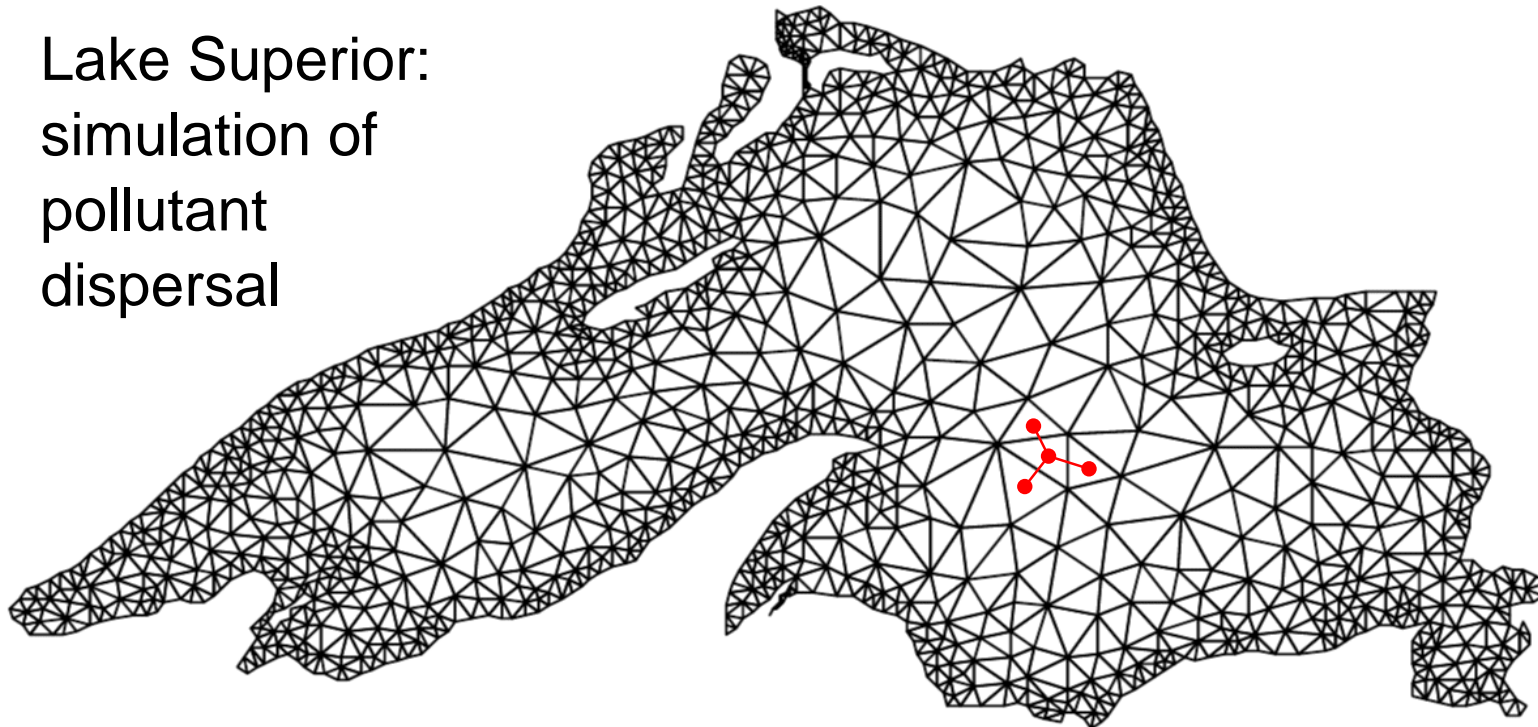


- ➔ The effective force on a star in a star cluster depends on the masses and locations of all other stars
  - ➔ possible approximation: restriction to relatively close stars
    - ➔ will, however, result in dynamic communication

### Example for structured / unstructured communication

- ➔ Structured: stencil algorithms
- ➔ Unstructured: “unstructured grids”

Lake Superior:  
simulation of  
pollutant  
dispersal



- ➔ grid points are defined at different density
- ➔ edges: neighborhood relation (communication)



- ➔ So far: abstract parallel algorithms
- ➔ Now: concrete formulation for real computers
  - ➔ limited number of processors
  - ➔ costs for communication, process creation, process switching, ...
- ➔ Goals:
  - ➔ reducing the communication costs
    - ➔ aggregation of tasks
    - ➔ replication of data and/or computation
  - ➔ retaining the flexibility
    - ➔ sufficiently fine-grained parallelism for mapping phase



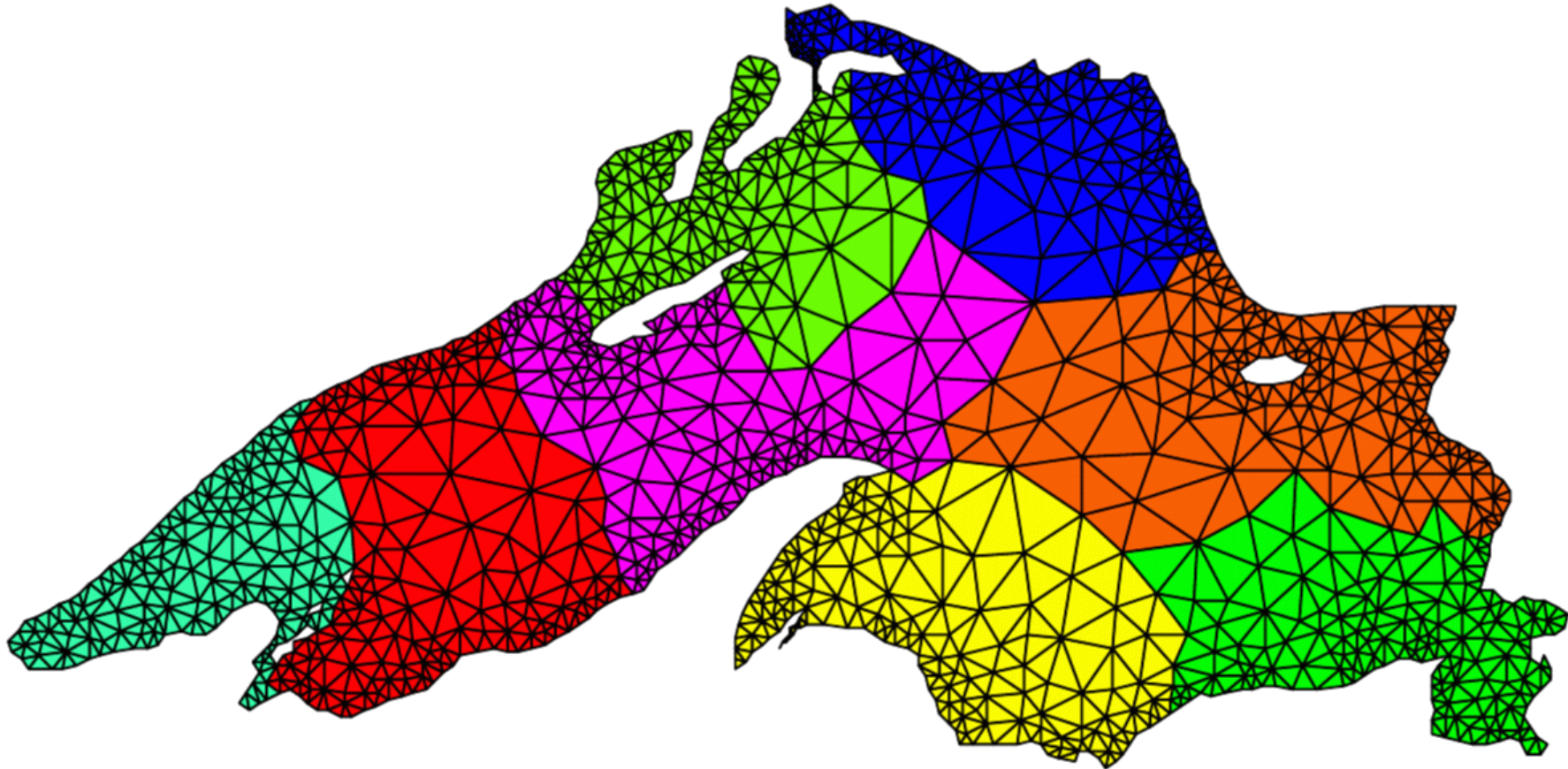
- ➔ Task: assignment of tasks to available processors
- ➔ Goal: minimizing the execution time
- ➔ Two (conflicting) strategies:
  - ➔ map concurrently executable tasks to different processors
    - ➔ high degree of parallelism
  - ➔ map communicating tasks to the same processor
    - ➔ higher locality (less communication)
- ➔ Constraint: load balancing
  - ➔ (roughly) the same computing effort for each processor
- ➔ The mapping problem is NP complete



### Variants of mapping techniques

- ➔ Static mapping
  - ➔ fixed assignment of tasks to processors when program is started
  - ➔ for algorithms on arrays or Cartesian grids:
    - ➔ often manually, e.g., block wise or cyclic distribution
  - ➔ for unstructured grids:
    - ➔ graph partitioning algorithms, e.g., greedy, recursive coordinate bisection, recursive spectral bisection, ...
- ➔ Dynamic mapping (dynamic load balancing)
  - ➔ assignment of tasks to processors at runtime
  - ➔ variants:
    - ➔ tasks stay on their processor until their execution ends
    - ➔ task migration is possible during runtime

### Example: static mapping with unstructured grid



- ➔ (Roughly) the same number of grid points per processor
- ➔ Short boundaries: small amount of communication

---

# Parallel Processing

Winter Term 2025/26

03.11.2025

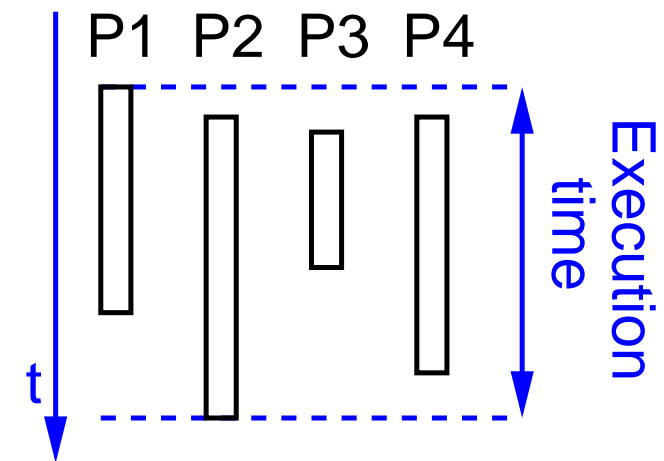
Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 10, 2025

## 2.9 Performance Considerations



- ➔ Which performance gain results from the parallelisation?
- ➔ Possible performance metrics:
  - ➔ execution time, throughput, memory requirements, processor utilisation, development cost, maintenance cost, ...
- ➔ In the following, we consider execution time
  - ➔ **execution time** of a parallel program: time between the start of the program and the end of the computation on the last processor





### Speedup (*Beschleunigung*)

➔ Reduction of execution time due to parallel execution

➔ **Absolute speedup**

$$S(p) = \frac{T_s}{T(p)}$$

➔  $T_s$  = execution time of the sequential program (or the best sequential algorithm, respectively)

➔  $T(p)$  = execution time of the parallel program (algorithm) with  $p$  processors



### Speedup (*Beschleunigung*) ...

➔ **Relative speedup** (for “sugarcoated” results ...)

$$S(p) = \frac{T(1)}{T(p)}$$

➔  $T(1)$  = execution time of the parallel program (algorithm) with one processor

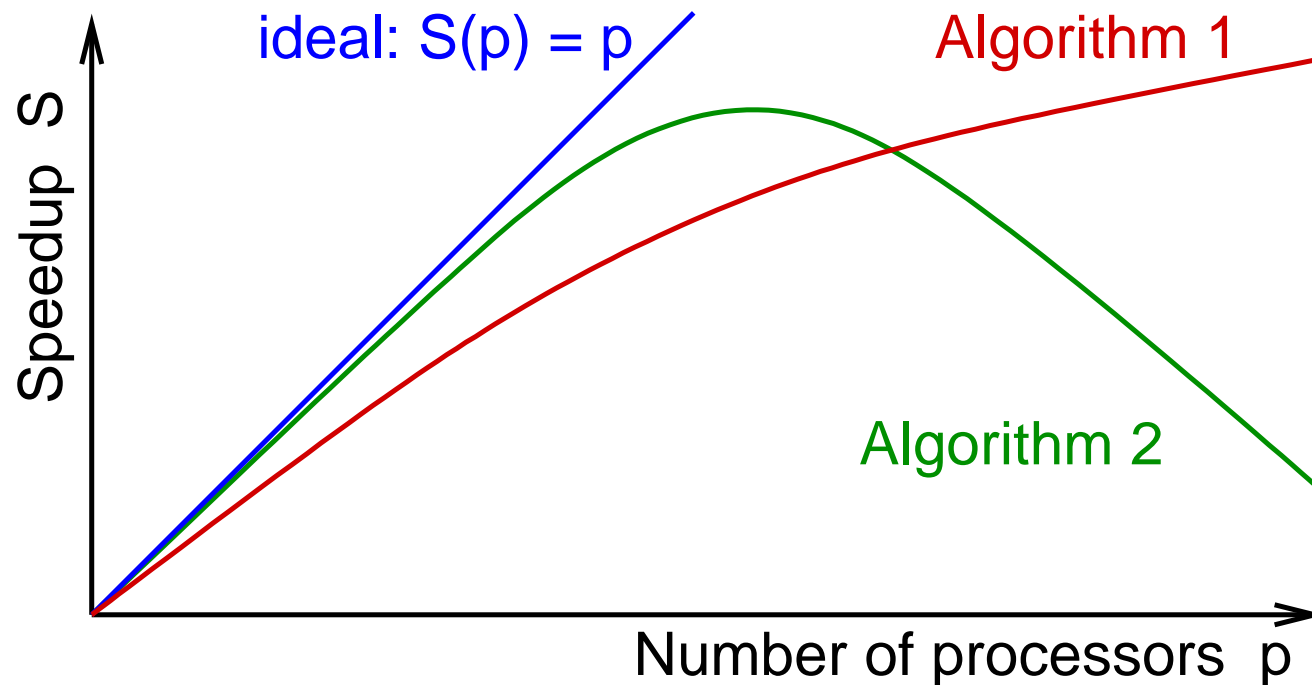
➔ Optimum:  $S(p) = p$

➔ Often: with **fixed** problem size,  $S(p)$  declines again, when  $p$  increases

➔ more communication, less computing work per processor

### Speedup (*Beschleunigung*) ...

➔ Typical trends:



➔ Statements like “speedup of 7.5 with 8 processors” can not be extrapolated to a larger number of processors



### Amdahl's Law

- ➔ Defines an upper limit for the achievable speedup
- ➔ Basis: usually, not all parts of a program can be parallelized
  - ➔ due to the programming effort
  - ➔ due to data dependences
- ➔ Let  $a$  be the **portion of time** of these program parts in the **sequential** version of the program. Then:

$$S(p) = \frac{T_s}{T(p)} \leq \frac{1}{a + (1 - a)/p} \leq \frac{1}{a}$$

- ➔ With a 10% sequential portion, this leads to  $S(p) \leq 10$



### Superlinear speedup

- ➔ Sometimes we observe  $S(p) > p$ , although this should actually be impossible
- ➔ Causes:
  - ➔ cache effects
    - ➔ with  $p$  processors, the amount of cache is  $p$  times higher than with one processor
    - ➔ thus, we also have higher cache hit rates
  - ➔ implicit change in the algorithm
    - ➔ e.g., with parallel tree search: several paths in the search tree are traversed simultaneously
      - ➔ limited breadth-first search instead of depth-first search



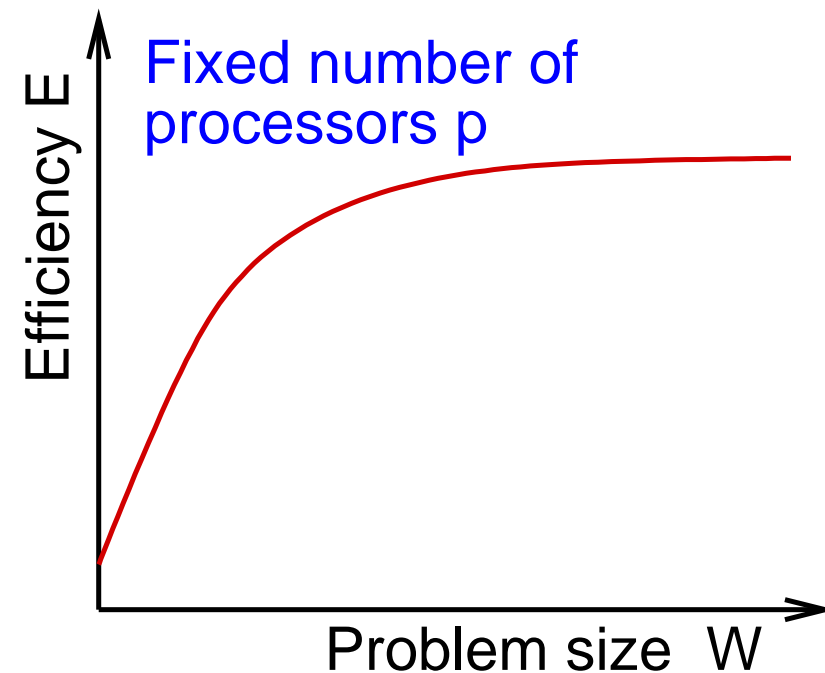
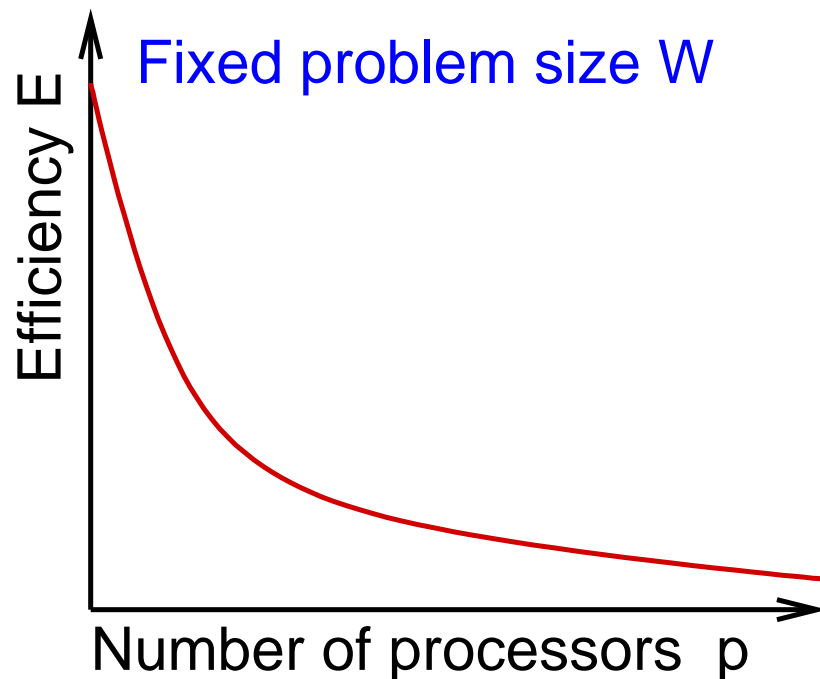
### Efficiency

$$E(p) = \frac{S(p)}{p}$$

- ➔ Metrics for the utilisation of a parallel computer
- ➔  $E(p) \leq 1$ , the optimum would be  $E(p) = 1$

### Scalability

➔ Typical observations:



➔ Reason: with increasing  $p$ : less work per processor, but the same amount of (or even more) communication



### Scalability ...

- ➔ How must the problem size  $W$  increase with increasing number of processors  $p$ , such that the efficiency stays the same?
- ➔ Answer is given by the **isoefficiency function**
- ➔ Parallel execution time

$$T(p) = \frac{W + T_o(W, p)}{p}$$

- ➔  $T_o(W, p)$  = overhead of parallel execution
- ➔  $T$  and  $W$  are measured as the number of elementary operations
- ➔ Thus:

$$W = \frac{E(p)}{1 - E(p)} \cdot T_o(W, p)$$



### Scalability ...

- ➔ Isoefficiency function  $I(p)$ 
  - ➔ solution of the equation  $W = K \cdot T_o(W, p)$  w.r.t.  $W$
  - ➔  $K = \text{constant}$ , depending on the required efficiency
- ➔ Good scalability:  $I(p) = \mathcal{O}(p)$  or  $I(p) = \mathcal{O}(p \log p)$
- ➔ Bad scalability:  $I(p) = \mathcal{O}(p^k)$  for  $k > 1$
- ➔  $T_o(W, p)$  is determined by analysing the parallel algorithm
  - ➔ how much time is needed for communication / synchronisation and potentially additional computations?
  - ➔ more details and examples in chapter [2.9.5](#)



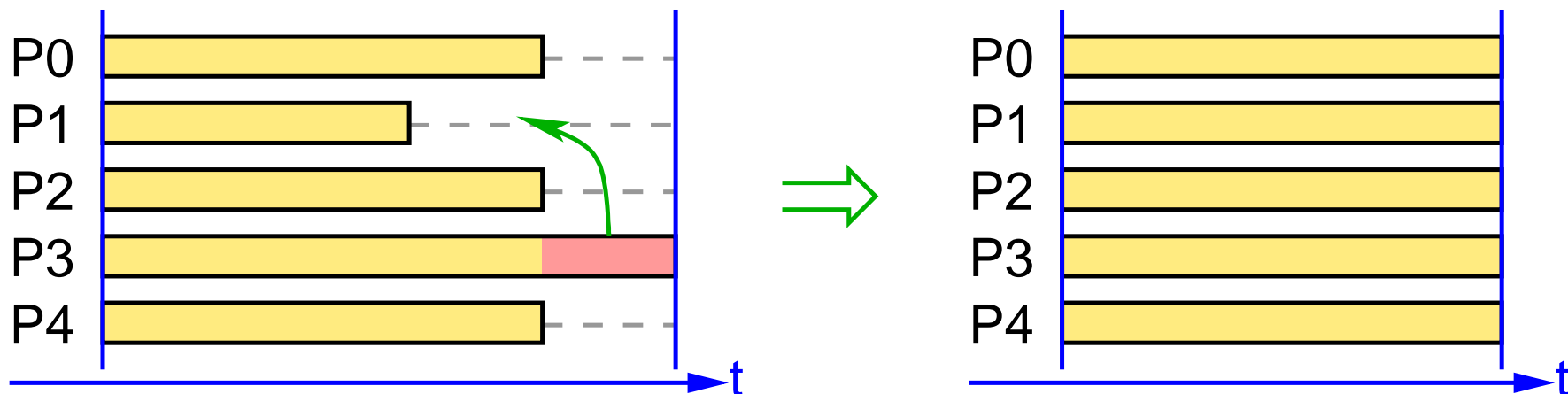
- ➔ **Access losses** due to data exchange between tasks
  - ➔ e.g., message passing, remote memory access
- ➔ **Conflict losses** due to shared use of resources by multiple tasks
  - ➔ e.g., conflicts when accessing the network, mutual exclusion when accessing data
- ➔ **Utilisation losses** due to insufficient degree of parallelism
  - ➔ e.g., waiting for data, load imbalance
- ➔ **Complexity losses** due to additional work necessary for the parallel execution
  - ➔ e.g., partitioning of unstructured grids



- ➔ **Algorithmic losses** due to modifications of the algorithms during the parallelisation
  - ➔ e.g., worse convergence of an iterative method
- ➔ **Dumping losses** due to computations, which are executed redundantly but not used later on
  - ➔ e.g., lapsed search in branch-and-bound algorithms
- ➔ **Breaking losses** when computations should end
  - ➔ e.g., with search problems: all other processes must be notified that a solution has been found

### Introduction

- ➔ For optimal performance: processors should compute equally long between two (global) synchronisations
- ➔ synchronisation: includes messages and program start / end



- ➔ Load in this context: execution time between two synchronisations
- ➔ other load metrics are possible, e.g., communication load
- ➔ Load balancing is one of the goals of the mapping phase



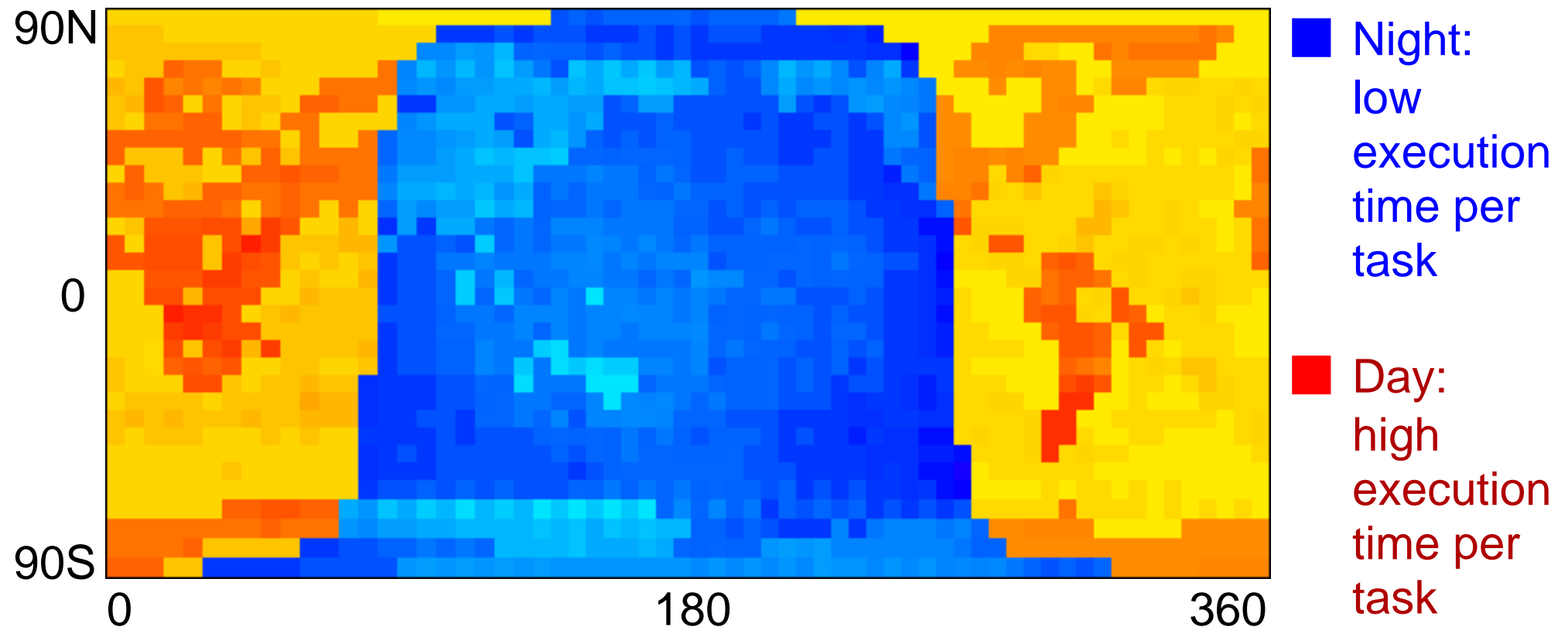
### Reasons for load imbalance

- ➔ Unequal computational load of the tasks
  - ➔ e.g., atmospheric model: areas over land / water
- ➔ Heterogeneous execution platform
  - ➔ e.g., processors (cores) with different performance
- ➔ Computational load of the tasks changes dynamically
  - ➔ e.g., in atmospheric model, depending on the simulated time of day (solar radiation)
- ➔ Background load on the processors
  - ➔ e.g., in a PC cluster

static

dynamic

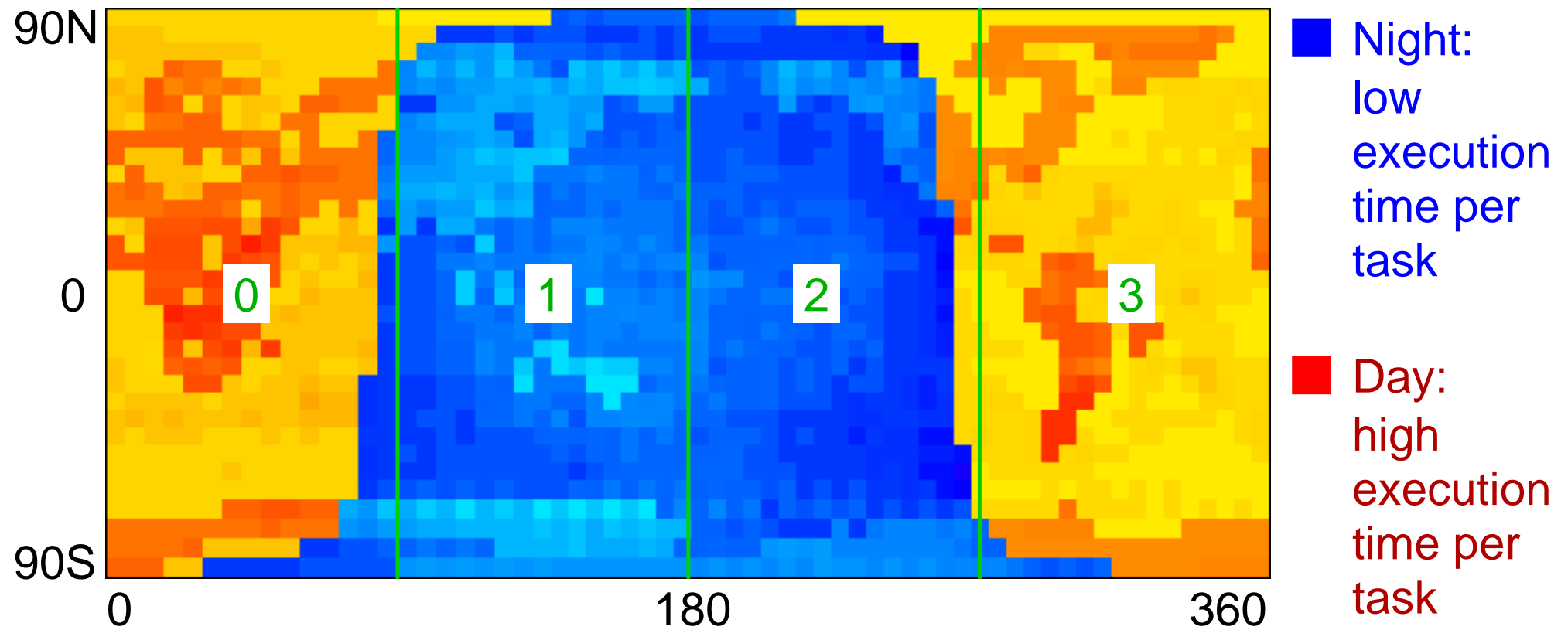
### Example: atmospheric model



➔ Continents: static load imbalance

➔ Border between day and night: dynamic load imbalance

### Example: atmospheric model



➔ Continents: static load imbalance

➔ Border between day and night: dynamic load imbalance



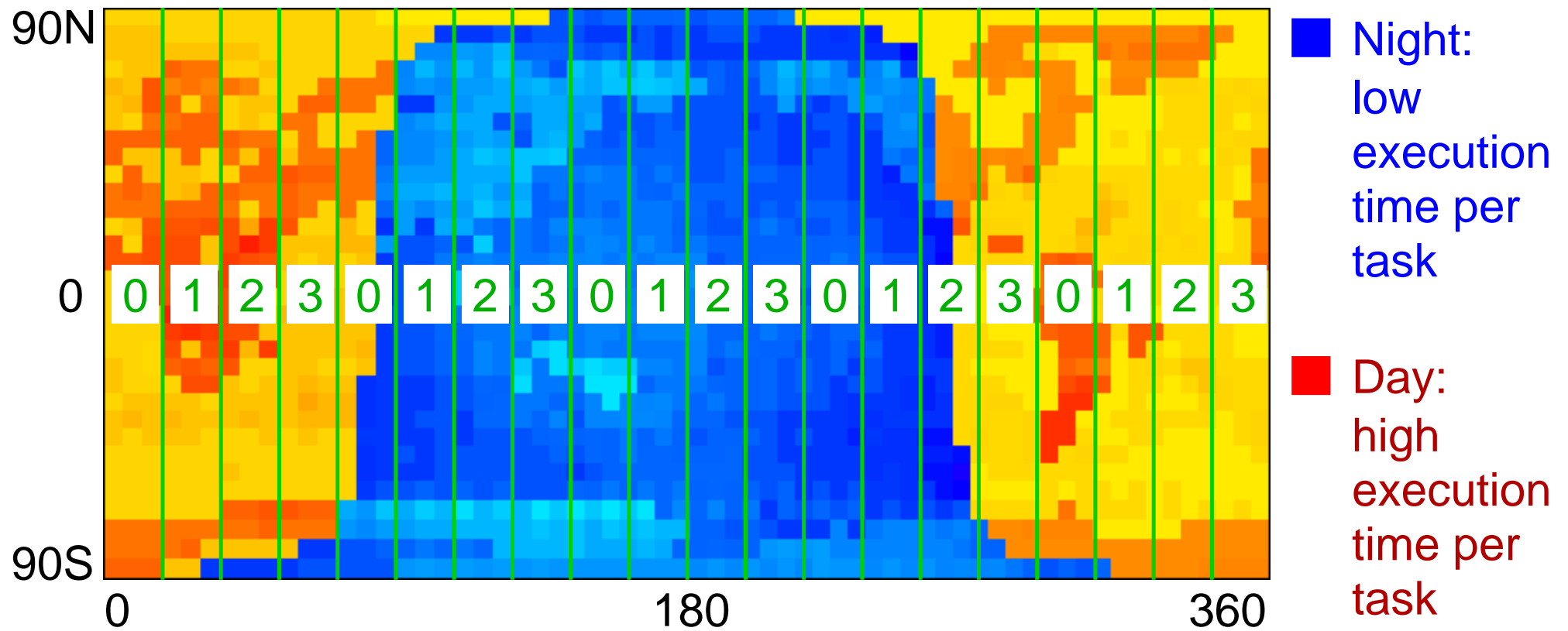
### Static load balancing

- ➔ Goal: distribute the tasks to the processors at / before program start, such that the computational load of the processors is equal
- ➔ Two fundamentally different approaches:
  - ➔ take into account the tasks' different computational load when mapping them to processors
    - ➔ extension of graph partitioning algorithms
    - ➔ requires a good estimation of a task's load
    - ➔ no solution, when load changes dynamically
  - ➔ fine grained cyclic or random mapping
    - ➔ results (most likely) in a good load balancing, even when the load changes dynamically
    - ➔ price: usually higher communication cost

## 2.9.3 Load Balancing ...



### Example: atmospheric model, cyclic mapping



➔ Each processor has tasks with high and low computational load



### Dynamic load balancing

- ➔ Independent (often dyn. created) tasks (e.g., search problem)
  - ➔ goal: processors do not idle, i.e., always have a task to process
    - ➔ even at the end of the program, i.e., all processes finish at the same time
  - ➔ tasks are dynamically **allocated** to processors and stay there until their processing is finished
    - ➔ optimal: allocate task with highest processing time first
- ➔ Communicating tasks (SPMD, e.g., stencil algorithm)
  - ➔ goal: equal computing time between synchronisations
  - ➔ if necessary, tasks are **migrated** between processors during their execution



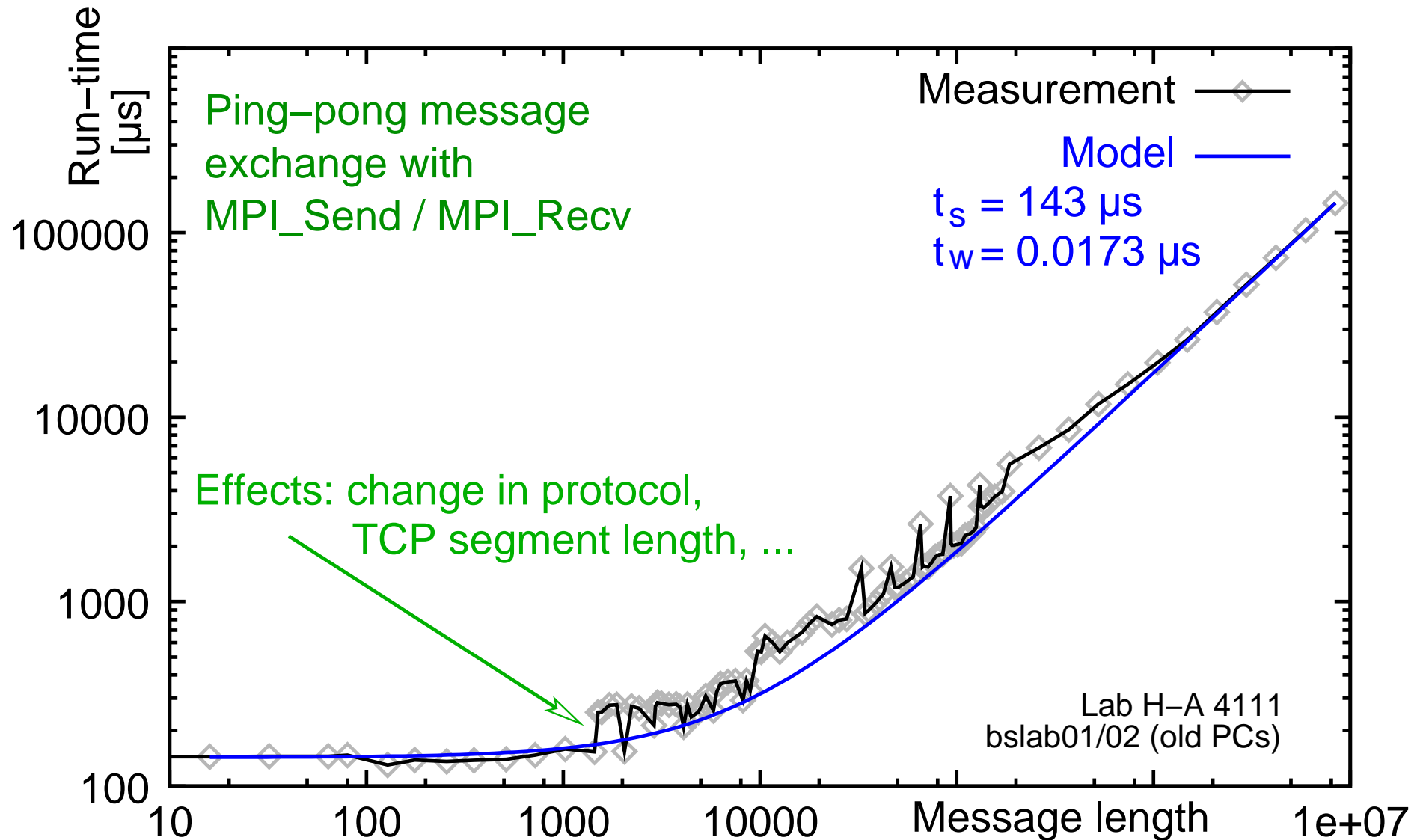
### How to determine performance metrics

- ➔ Analytical model of the algorithm
  - ➔ approach: determine computation and communication time
    - ➔  $T(p) = t_{comp} + t_{comm}$
    - ➔ computation/communication ratio  $t_{comp}/t_{comm}$  allows a rough estimation of performance
  - ➔ requires a computation model (model of the computer hardware)
- ➔ Measurement with the real programm
  - ➔ explicit timing measurement in the code
  - ➔ performance analysis tools

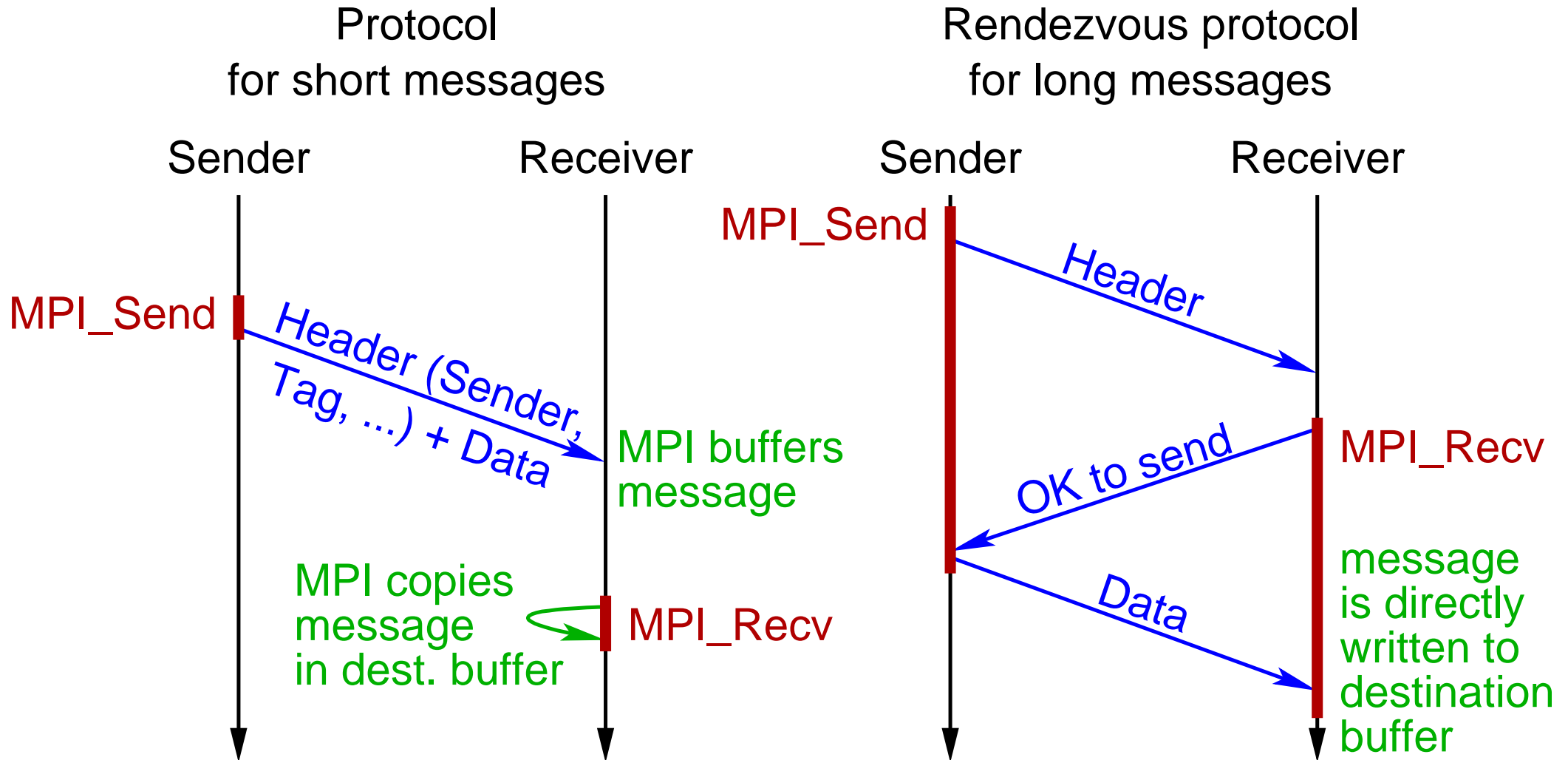
### Models for communication time

- ➔ E.g., for MPI (following Rauber: “*Parallele und verteilte Programmierung*”)
  - ➔ point-to-point send:  $t(m) = t_s + t_w \cdot m$
  - ➔ broadcast:  $t(p, m) = \tau \cdot \log p + t_w \cdot m \cdot \log p$
- ➔ Parameters ( $t_s, t_w, \tau$ ) are obtained via **micro benchmarks**
  - ➔ selectively measure a single aspect of the system
  - ➔ also allow the deduction of implementation characteristics
  - ➔ fitting, e.g., using the least square method
    - ➔ e.g., for point-to-point send:  
PCs H-A 4111 (remote):  $t_s = 75.3 \mu s, t_w = 9 ns$   
PCs H-A 4111 (local):  $t_s = 0.45 \mu s, t_w = 0.11 ns$

### Example: results of the micro benchmark SKaMPI



### Communication protocols in MPI





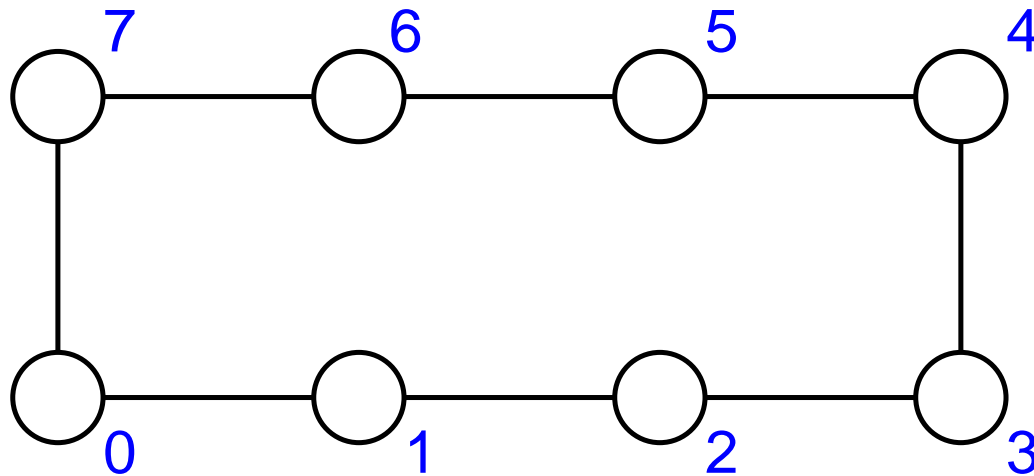
### Example: matrix multiplication

- ➔ Product  $C = A \cdot B$  of two square matrices
- ➔ Assumption:  $A, B, C$  are distributed blockwise on  $p$  processors
  - ➔ processor  $P_{ij}$  has  $A_{ij}$  and  $B_{ij}$  and computes  $C_{ij}$
- ➔  $P_{ij}$  needs  $A_{ik}$  and  $B_{kj}$  for  $k = 1 \dots \sqrt{p}$
- ➔ Chosen approach:
  - ➔ all-to-all broadcast of the  $A$  blocks in each row of processors
  - ➔ all-to-all broadcast of the  $B$  blocks in each column of processors

- ➔ computation of  $C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} \cdot B_{kj}$

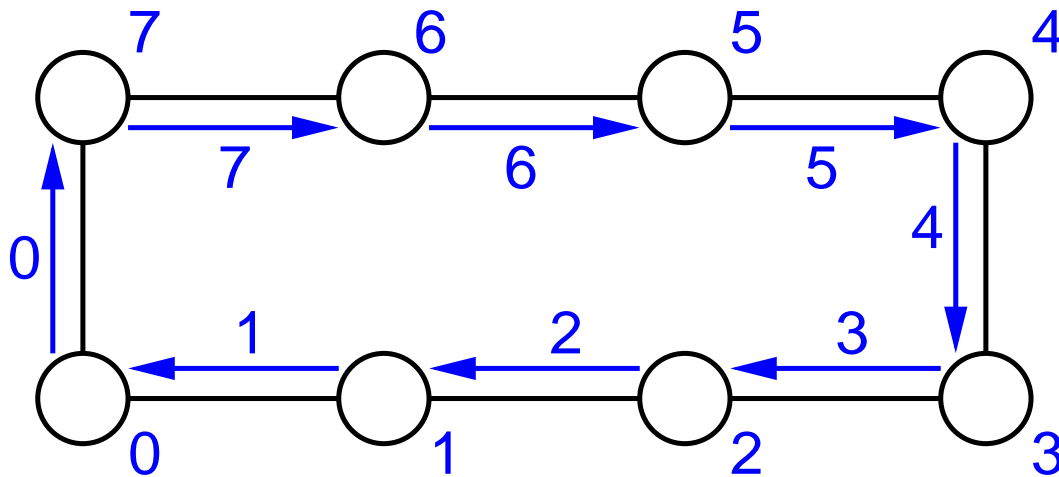
### All-to-all broadcast

- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
  - ➔ who can directly communicate with whom?
- ➔ Example: ring topology



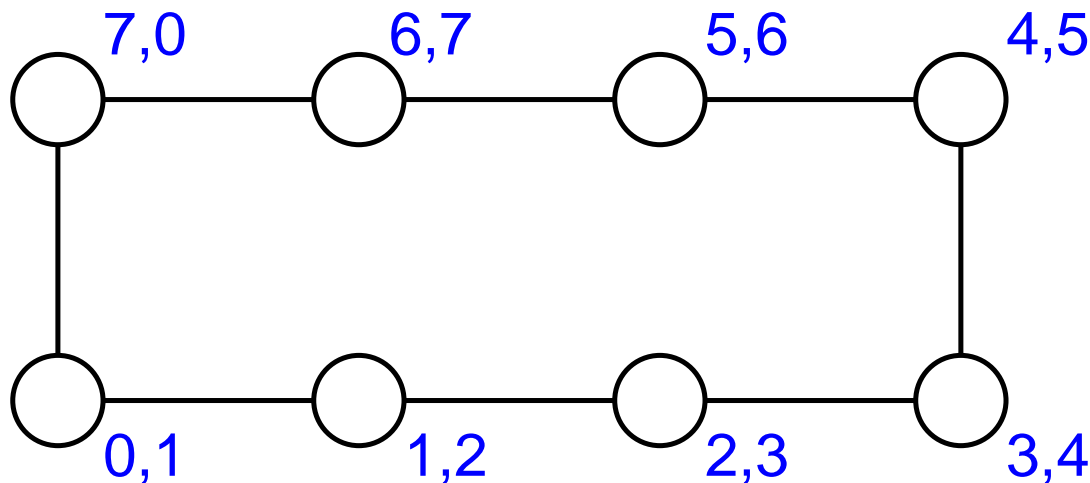
### All-to-all broadcast

- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
  - ➔ who can directly communicate with whom?
- ➔ Example: ring topology



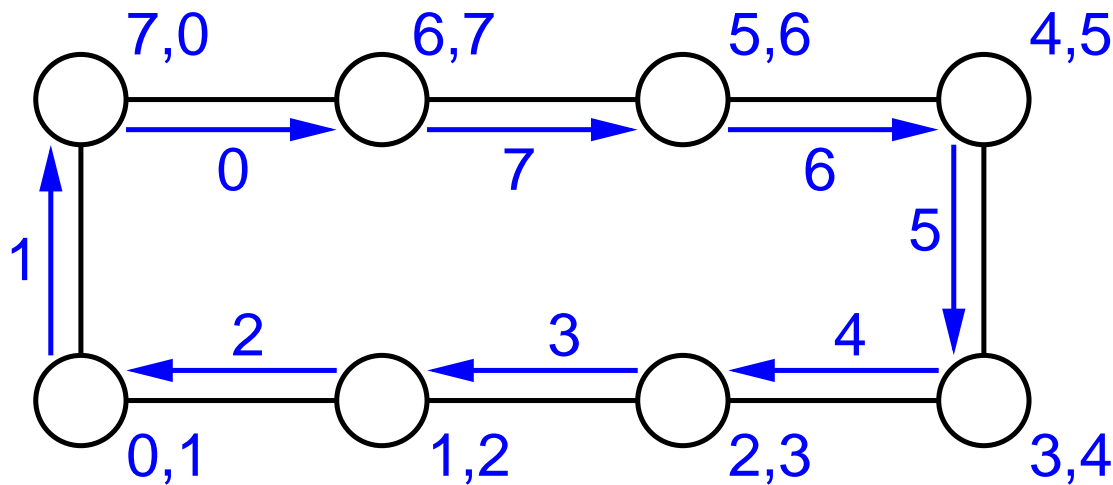
### All-to-all broadcast

- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
  - ➔ who can directly communicate with whom?
- ➔ Example: ring topology



### All-to-all broadcast

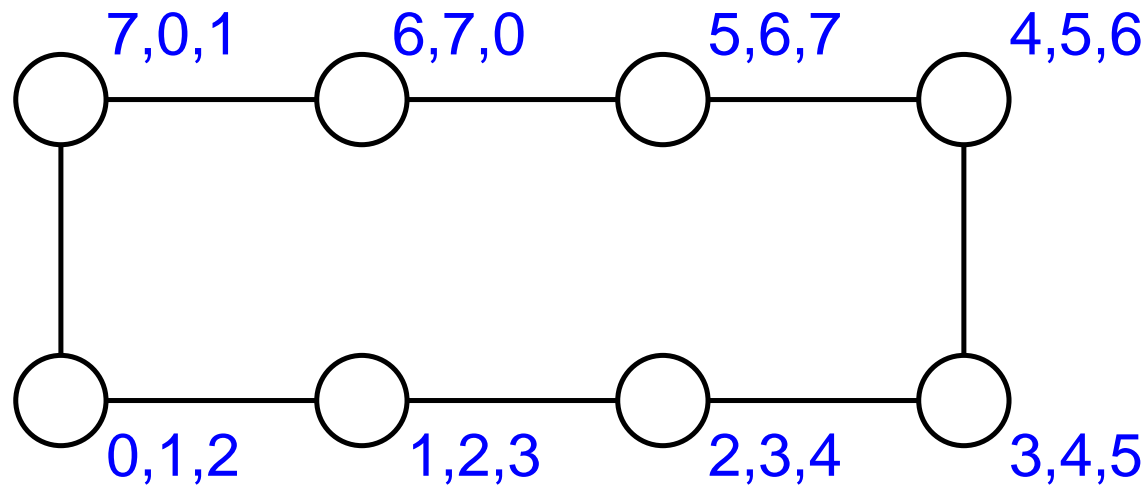
- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
  - ➔ who can directly communicate with whom?
- ➔ Example: ring topology





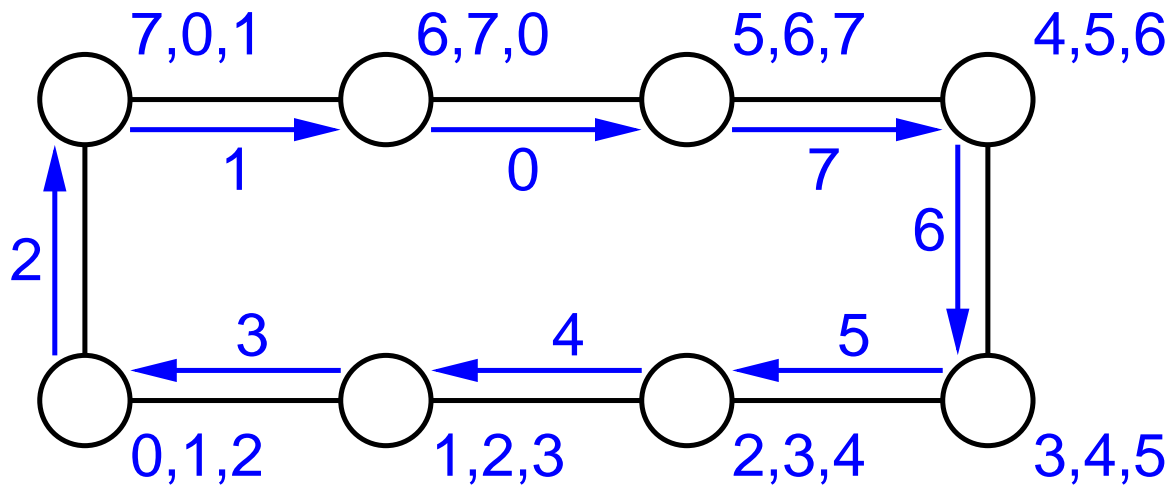
### All-to-all broadcast

- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
  - ➔ who can directly communicate with whom?
- ➔ Example: ring topology



### All-to-all broadcast

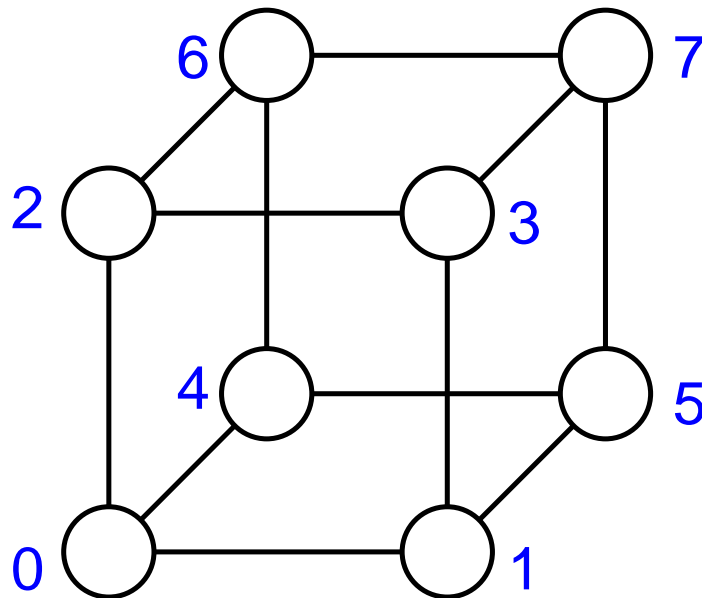
- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
  - ➔ who can directly communicate with whom?
- ➔ Example: ring topology



- ➔ Cost:  $t_s(p - 1) + t_w m(p - 1)$  ( $m$ : data length)

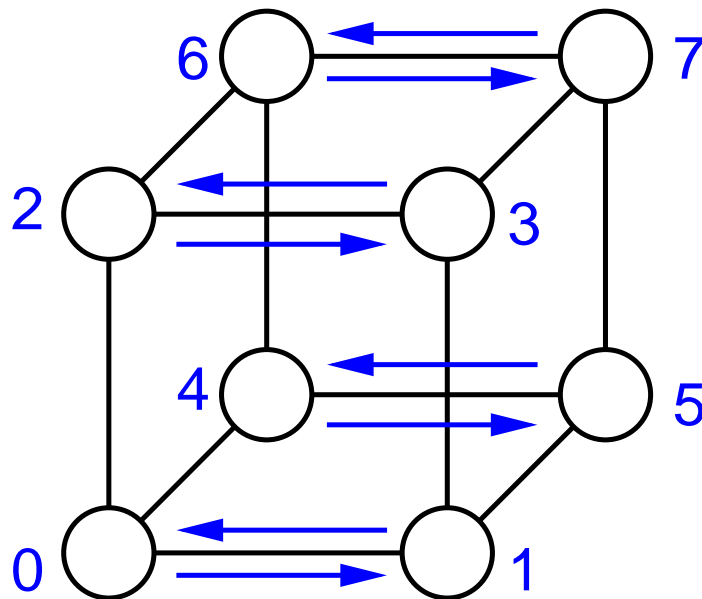
### All-to-all broadcast ...

- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



### All-to-all broadcast ...

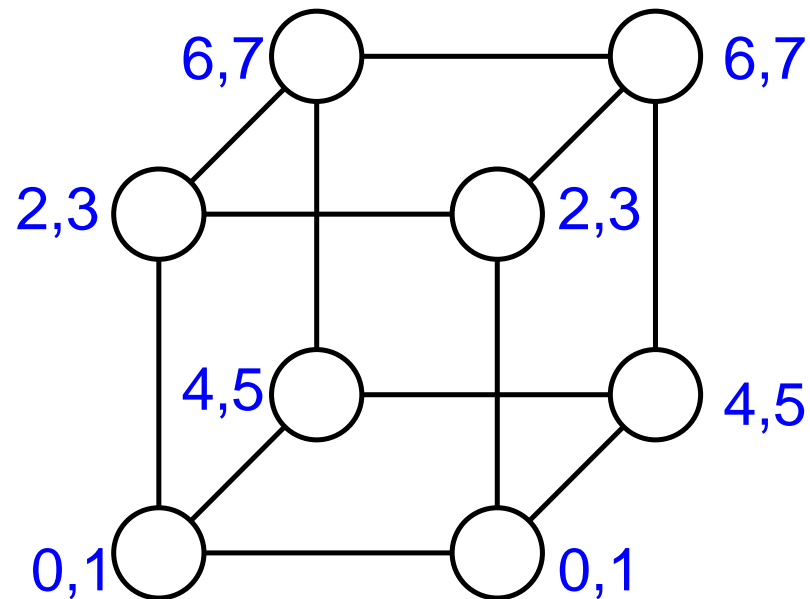
- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



1. Pairwise exchange  
in x direction

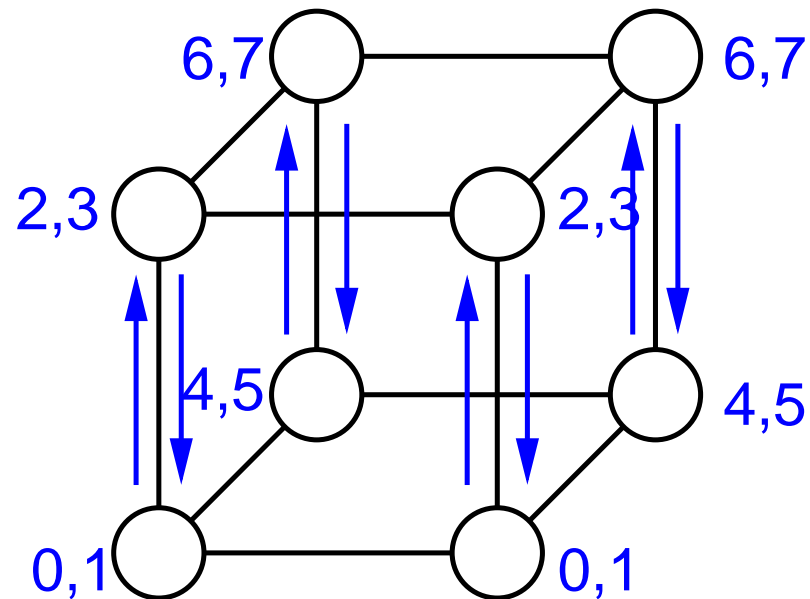
### All-to-all broadcast ...

- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



### All-to-all broadcast ...

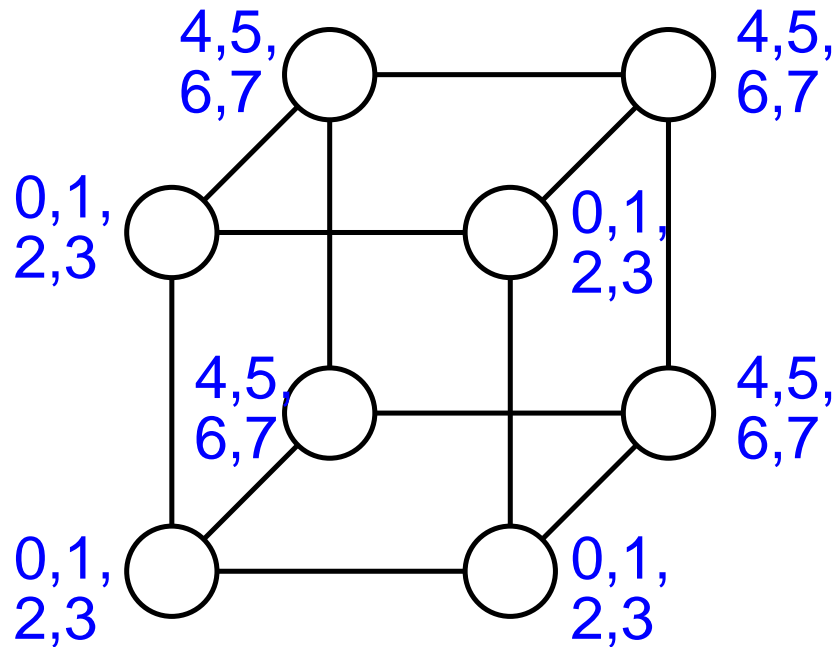
- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



2. Pairwise exchange  
in y direction

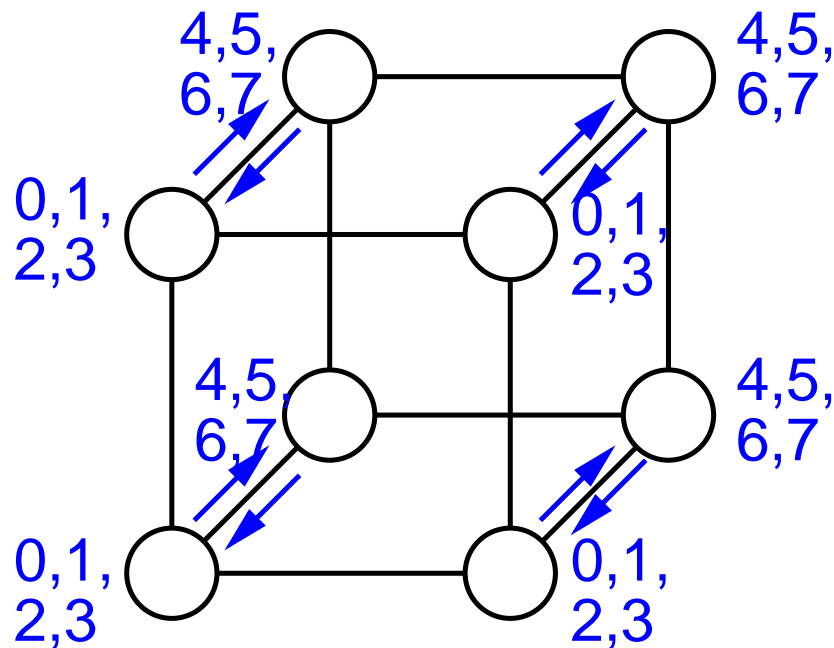
### All-to-all broadcast ...

- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



### All-to-all broadcast ...

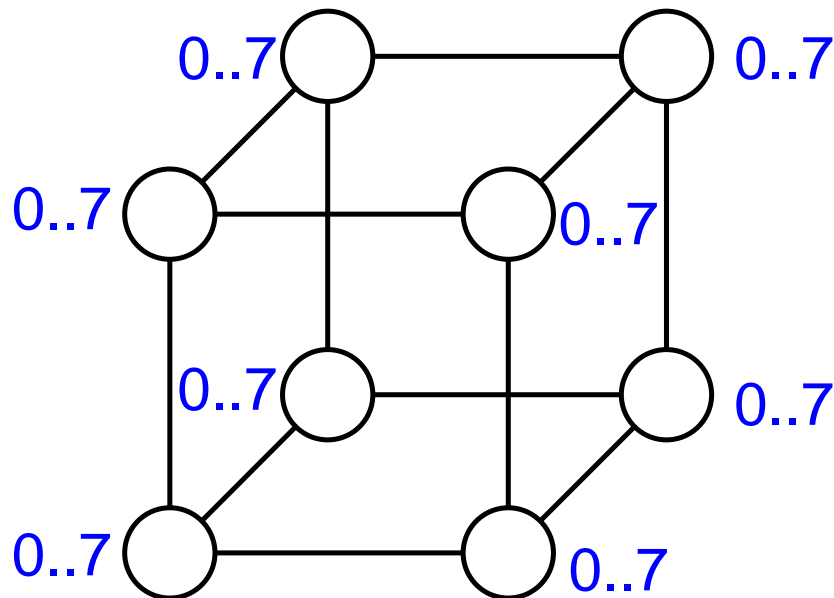
- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



3. Pairwise exchange  
in z direction

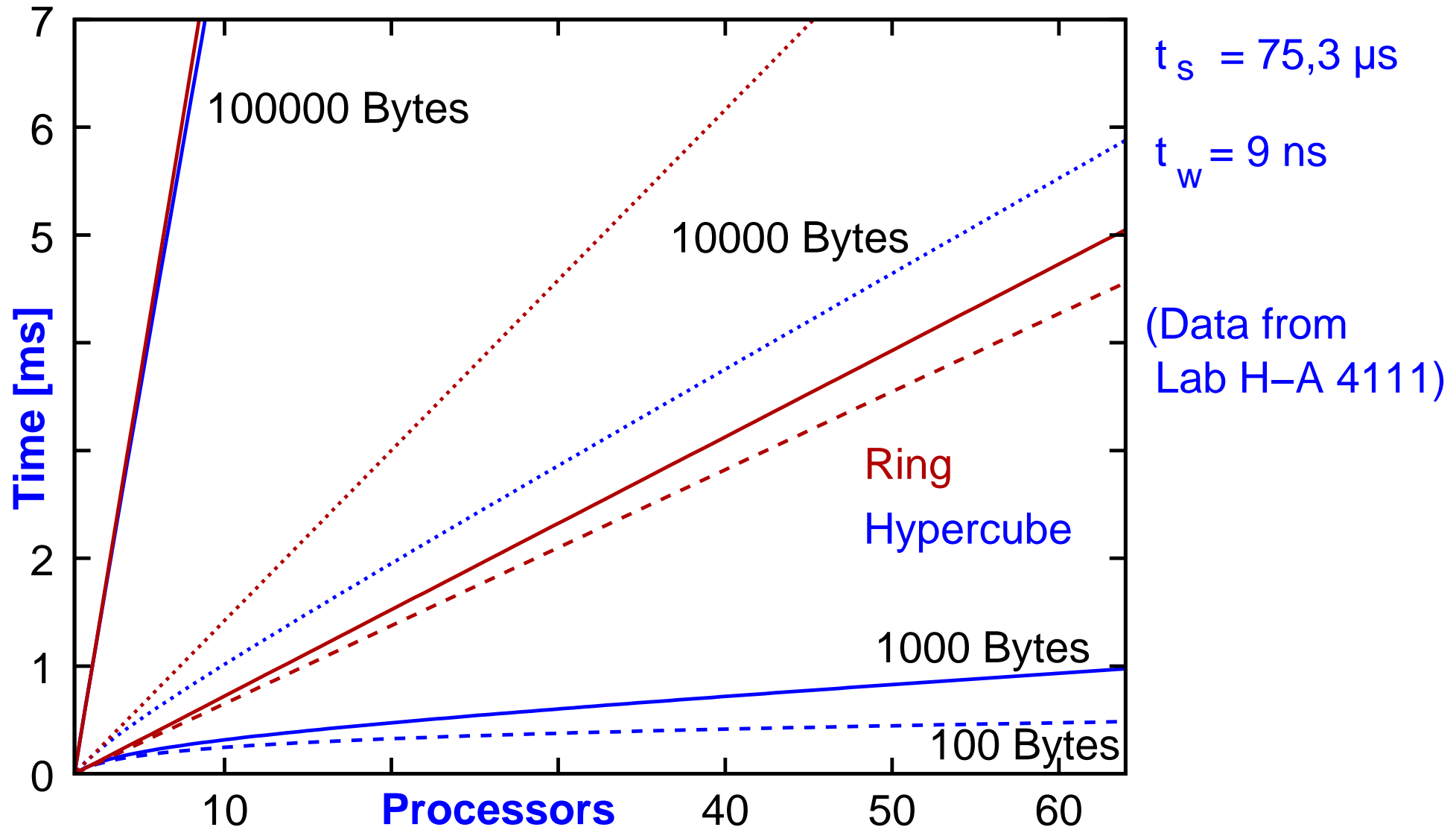
### All-to-all broadcast ...

- ➔ Example: communication along a hyper cube
- ➔ requires only  $\log p$  steps with  $p$  processors



➔ Cost: 
$$\sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m) = t_s \log p + t_w m (p - 1)$$

### All-to-all broadcast ...

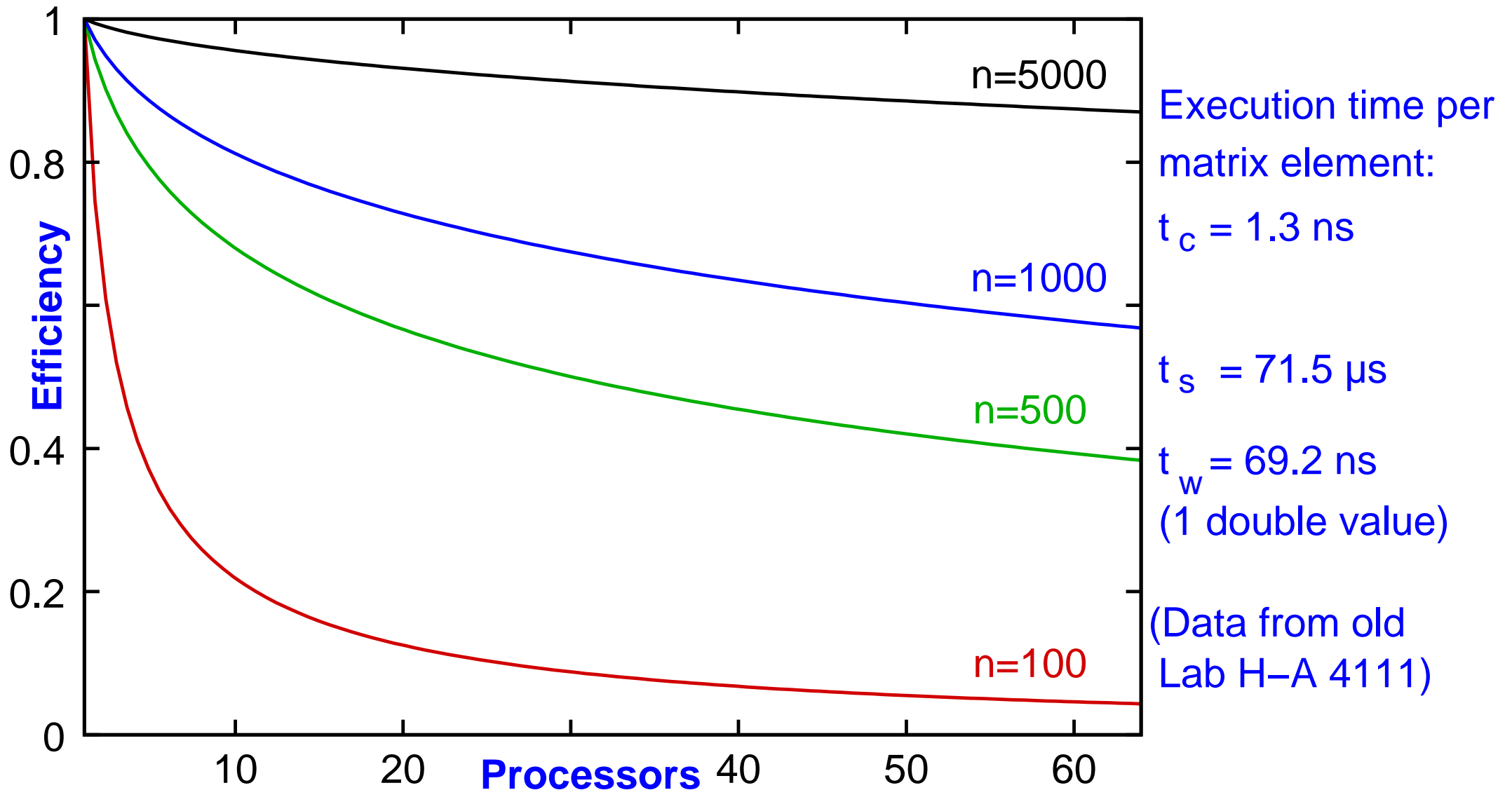


### Complete analysis of matrix multiplication

- ➔ Two all-to-all broadcast steps between  $\sqrt{p}$  processors
  - ➔ each step concurrently in  $\sqrt{p}$  rows / columns
- ➔ Communication time:  $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p} - 1))$
- ➔  $\sqrt{p}$  multiplications of  $(n/\sqrt{p}) \times (n/\sqrt{p})$  sub-matrices
- ➔ Computation time:  $t_c \sqrt{p} \cdot (n/\sqrt{p})^3 = t_c n^3 / p$
- ➔ Parallel run-time:  $T(p) \approx t_c n^3 / p + t_s \log p + 2t_w(n^2 / \sqrt{p})$
- ➔ Sequential run-time:  $T_s = t_c n^3$

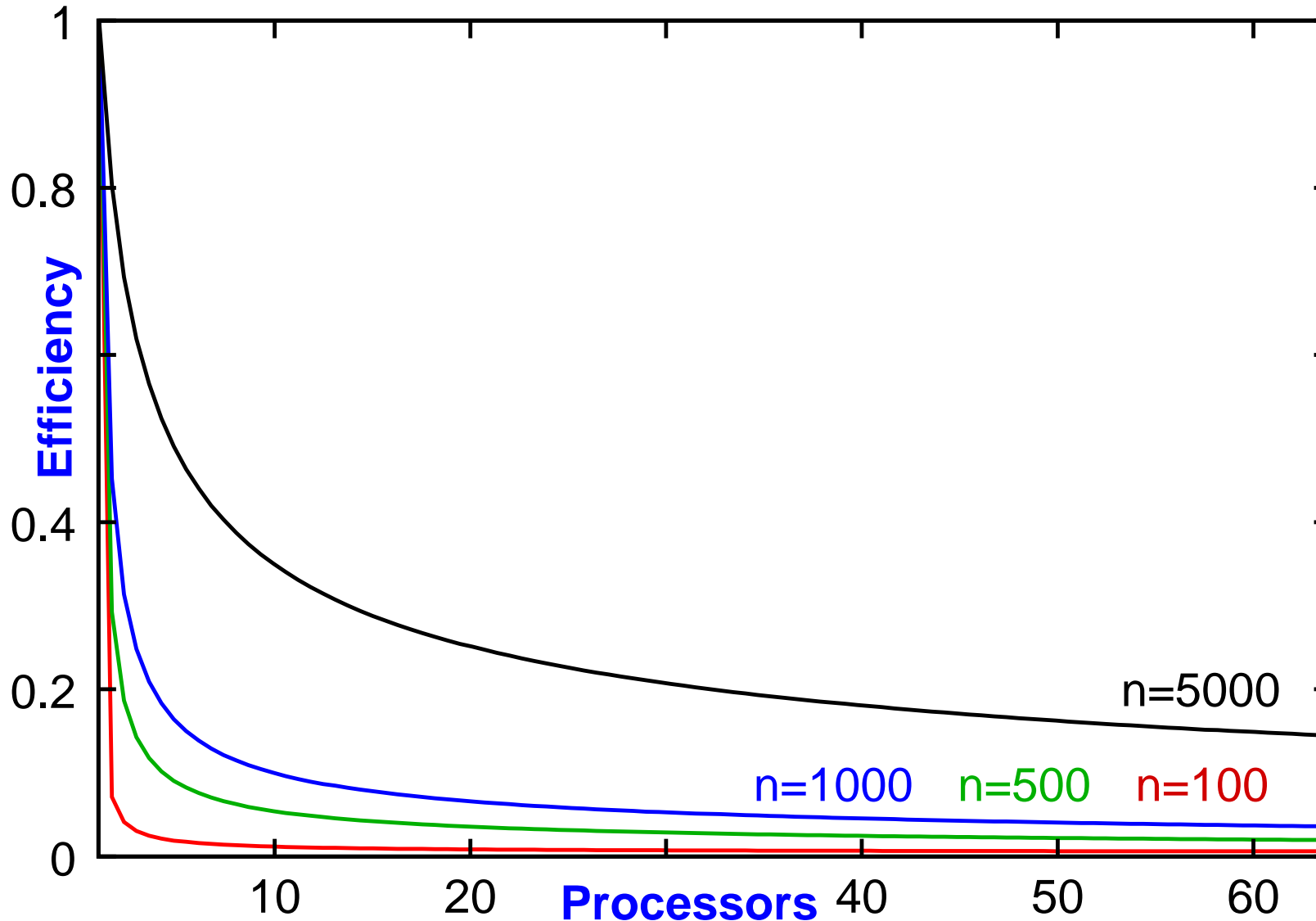


### Efficiency of matrix multiplication





### Efficiency of matrix multiplication



Execution time per matrix element:

$$t_c = 0.033 \text{ ns}$$

$$t_s = 75.3 \text{ } \mu\text{s}$$

$$t_w = 72 \text{ ns}$$

(1 double value)

(Data from new Lab H-A 4111)



- ➔ Goal: performance debugging, i.e., finding and eliminating performance bottlenecks
- ➔ Method: measurement of different quantities (metrics), if applicable separated according to:
  - ➔ execution unit (compute node, process, thread)
  - ➔ source code position (procedure, source code line)
  - ➔ time
- ➔ Tools are very different in their details
  - ➔ method of measurement, required preparation steps, processing of information, ...
- ➔ Some tools are also usable to visualise the program execution



### Metrics for performance analysis

- ➔ CPU time (assessment of computing effort)
- ➔ Wall clock time (includes times where thread is blocked)
- ➔ Communication time and volume
- ➔ Metrics of the operating system:
  - ➔ page faults, process switches, system calls, signals
- ➔ Hardware metrics (only with hardware support in the CPU):
  - ➔ CPU cycles, floating point operations, memory accesses
  - ➔ cache misses, cache invalidations, ...

---

# Parallel Processing

Winter Term 2025/26

04.11.2025

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 10, 2025



### Sampling (sample based performance analysis)

- ➔ Program is interrupted periodically
- ➔ Current value of the program counter is read (and maybe also the call stack)
- ➔ The full measurement value is assigned to this place in the program, e.g., when measuring CPU time:
  - ➔ periodic interruption every  $10ms$  CPU time
  - ➔ `CPU_time[current_PC_value] += 10ms`
- ➔ Mapping to source code level is done offline
- ➔ Result: measurement value for each function / source line



### Profiling and tracing (event based performance analysis)

- ➔ Requires an **instrumentation** of the programs, e.g., insertion of measurement code at interesting places
  - ➔ often at the beginning and end of library routines, e.g., MPI\_Recv, MPI\_Barrier, ...
- ➔ Tools usually do the instrumentation automatically
  - ➔ typically, the program must be re-compiled or re-linked
- ➔ Analysis of the results is done during the measurement (profiling) or after the program execution (tracing)
- ➔ Result:
  - ➔ measurement value for each measured function (profiling, tracing)
  - ➔ development of the measurement value over time (tracing)



### Example: measurement of cache misses

➔ Basis: hardware counter for cache misses in the processor

➔ Sampling based:

➔ when a certain counter value (e.g., 419) is reached, an interrupt is triggered

➔ `cache_misses[current_PC_value] += 419`

➔ Event based:

➔ insertion of code for reading the counters:

```
old_cm = read_hw_counter(25);  
for (j=0; j<1000; j++)  
    d += a[i][j];  
cache_misses += read_hw_counter(25) - old_cm;
```



### Pros and cons of the methods

- ➔ Sampling
  - ➔ low and predictable overhead; reference to source code
  - ➔ limited precision; no resolution in time
- ➔ Tracing
  - ➔ acquisition of all relevant data with high resolution in time
  - ➔ relatively high overhead; large volumes of data
- ➔ Profiling
  - ➔ reduced volume of data, but less flexible