

Introduction to Parallel Programming

Bachelor Scientific Programming / Angewandte Mathematik und
Informatik

B. Küppers

kueppers@itc.rwth-aachen.de

WS 2019 / 2020

Version 1.20 (09.03.2020)

▶ Times

- ▶ Start: 09:00
- ▶ End: 17:00
- ▶ Lunchbreak: 1 hour, starting between 11.30 am. and 12.30 pm., depends...

▶ Lectures

- ▶ Monday: Introduction, Java Threads
- ▶ Tuesday, Wednesday: OpenMP (with Tim Cramer)
- ▶ Thursday, Friday: MPI

The following slides have been created and revised by the following people:

- ▶ **Prof. Dr. Hans Joachim Pflug**
- ▶ **Dr. Tatjana Streit**
- ▶ **Dr. Christian Terboven**
- ▶ **Prof. Dr. Alexander Voß**
- ▶ **Bastian Küppers, MSc.**
- ▶ **Dipl. Math. Jens Hollmann**

Introduction

Computer Architectures, Parallelization at a Glance

▶ **Computer Architectures**

- ▶ Basic Computer Architectures
- ▶ Shared-Memory Parallel Systems
- ▶ Distributed-Memory Parallel Systems

▶ **Parallelization at a Glance**

- ▶ Basic Concepts
- ▶ Parallelization Strategies
- ▶ Prominent Issues

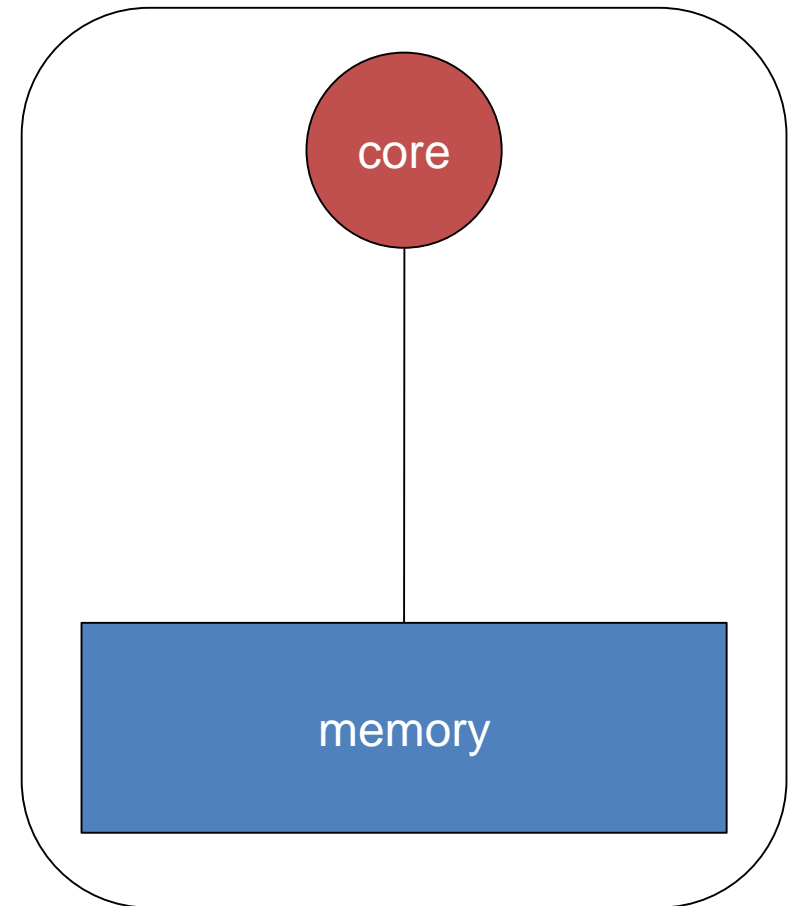
▶ Processor

- ▶ Fetch program from memory
- ▶ Execute program instructions
- ▶ Load data from memory
- ▶ Process data
- ▶ Write results back to memory

▶ Main Memory

- ▶ Store program
- ▶ Store data

▶ Input / Output is not covered here!



▶ CPU

- ▶ Fast (order of **3.0 GHz**)

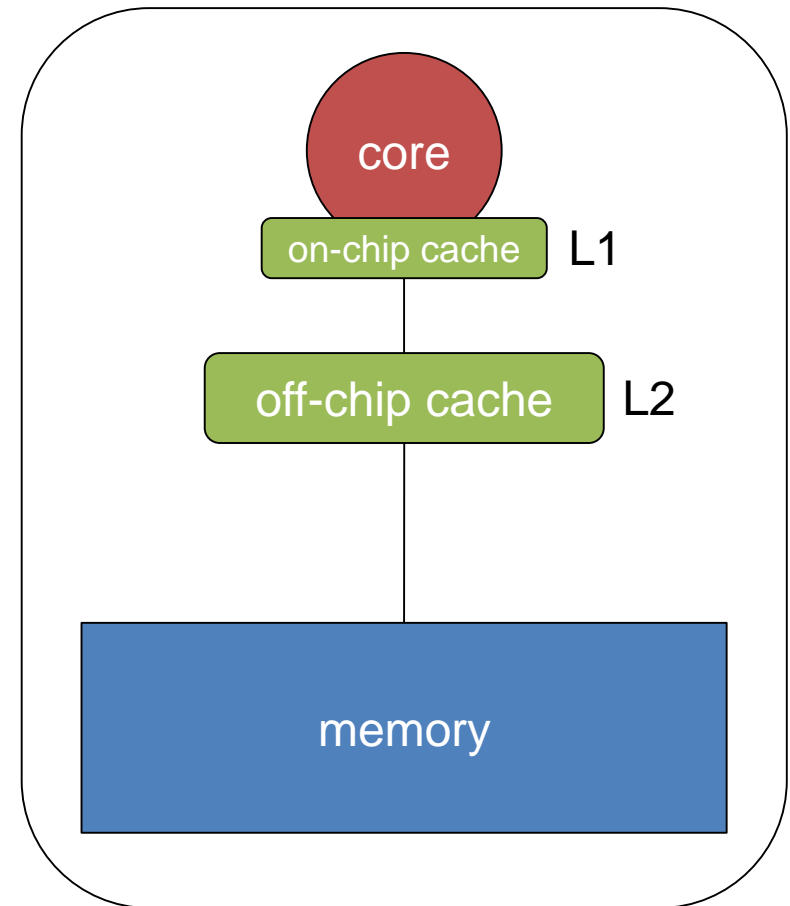
▶ Main Memory

- ▶ Slow (order of **0.3 GHz**)
- ▶ Large (order of **GB**)

▶ Caches

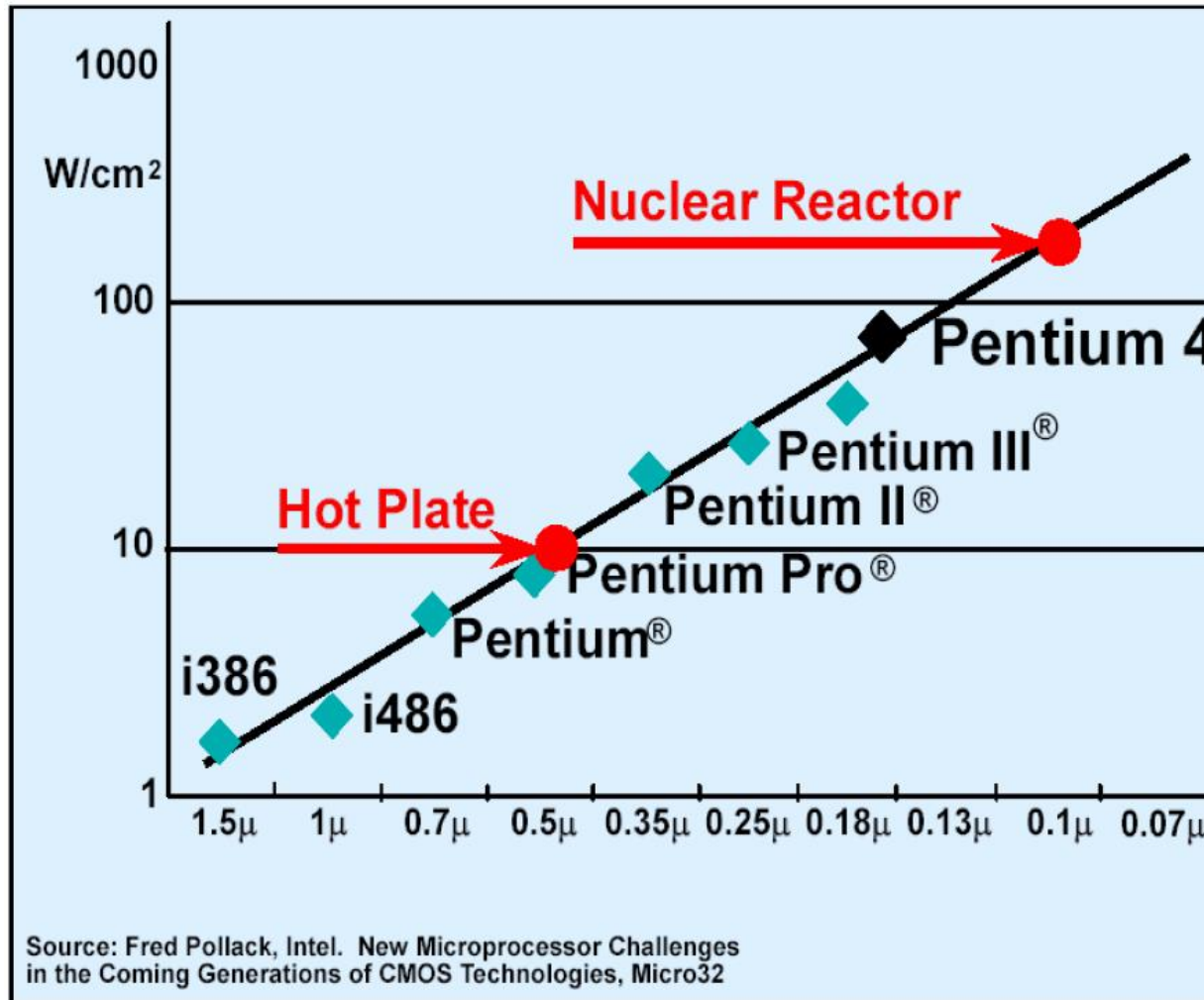
- ▶ Fast, but expensive
- ▶ Small (order of **MB**)

▶ Usage of Cache is mandatory for good performance on parallel applications.



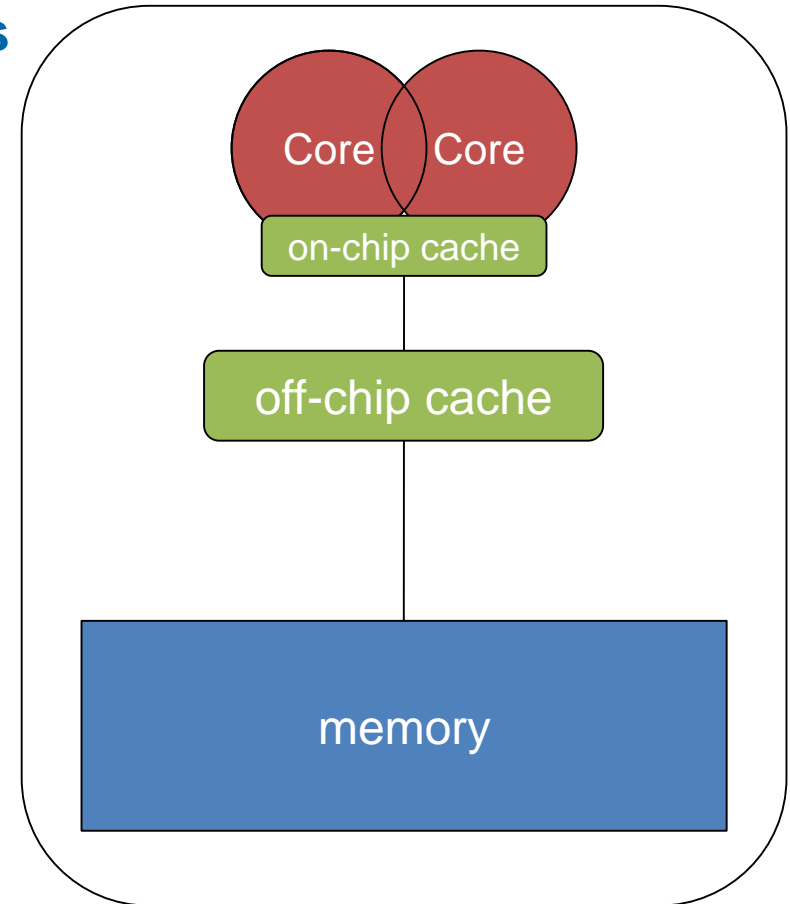
Why aren't CPUs getting faster anymore?

► The CPU would get too hot!



Fast clock cycles make processor chips more expensive, hotter and more power consuming.

- ▶ **Since 2005/2006 dual-core processors are produced for the home user.**
- ▶ **Number of cores per chip increases since then**
 - ▶ Today: up to 8 cores per chip for a standard CPU
- ▶ **Any recently bought PC or Laptop is a multi-core system already.**



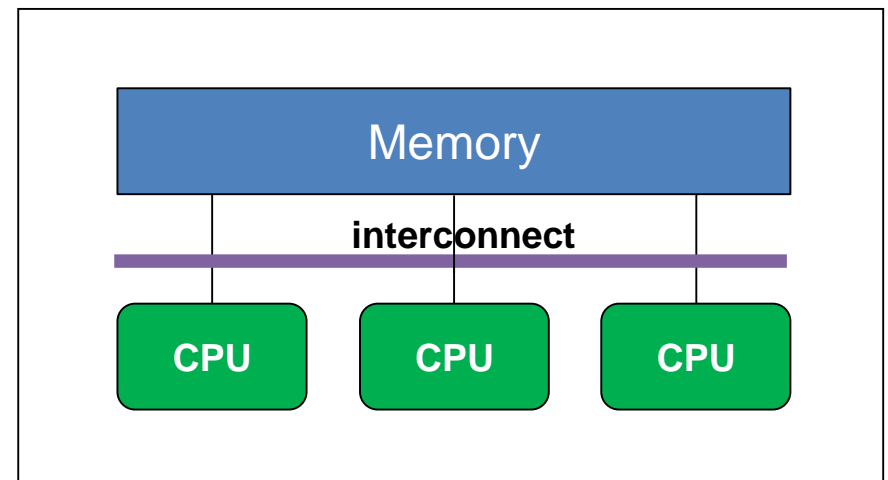
▶ **Computer Architectures**

- ▶ Basic Computer Architectures
- ▶ Shared-Memory Parallel Systems
- ▶ Distributed-Memory Parallel Systems

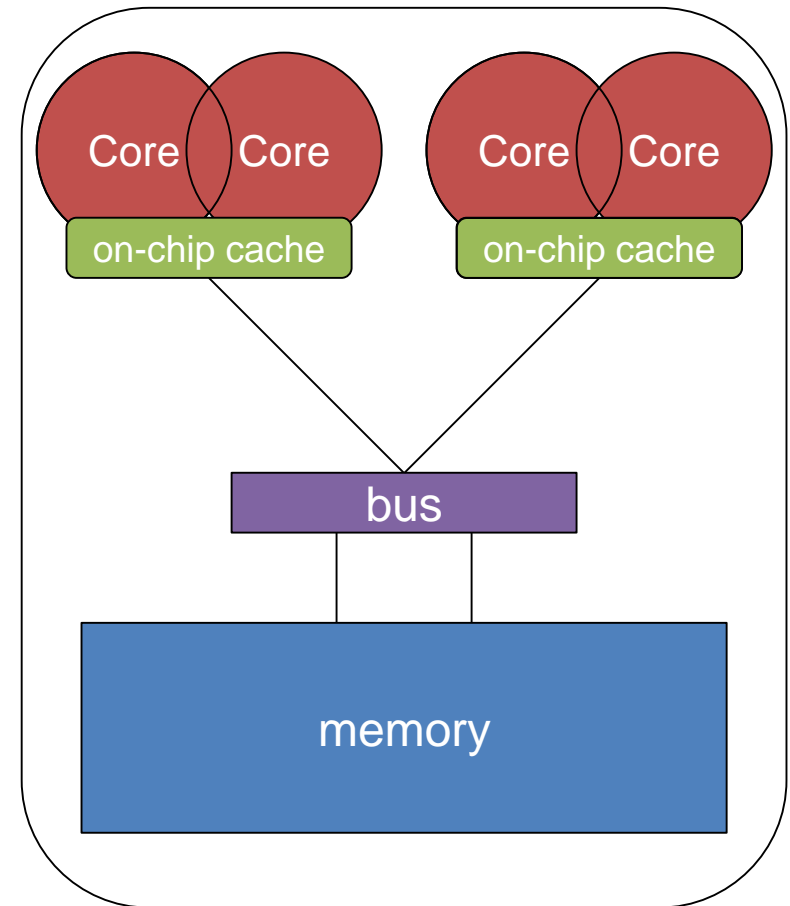
▶ **Parallelization at a Glance**

- ▶ Basic Concepts
- ▶ Parallelization Strategies
- ▶ Prominent Issues

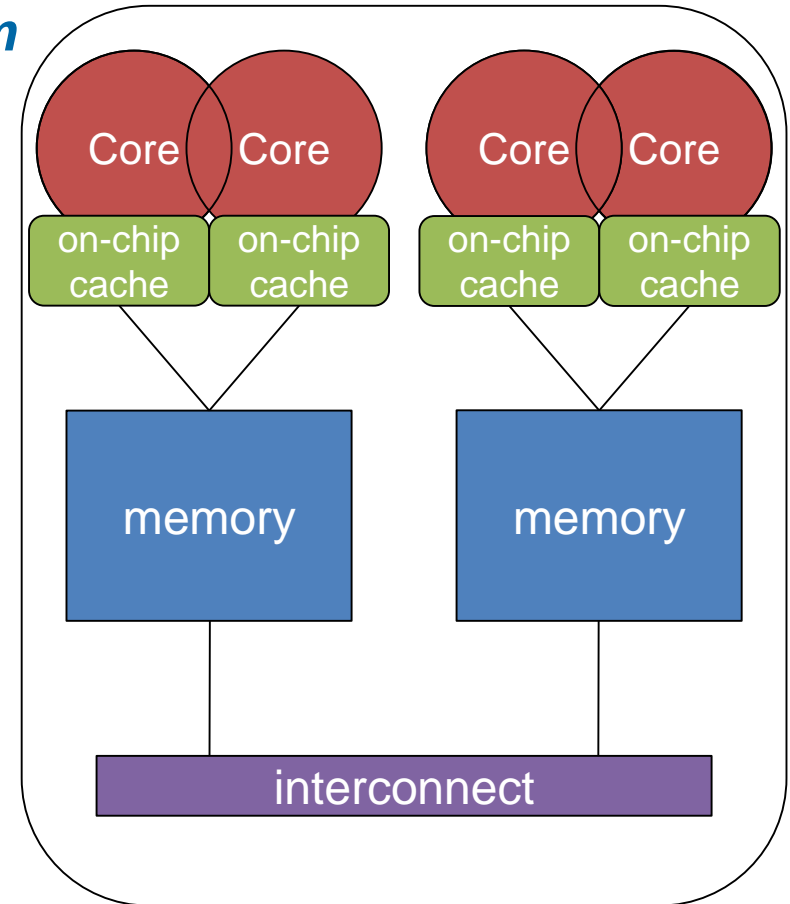
- ▶ **Implicit data distribution**
- ▶ **Implicit communication**
- ▶ **Different types of shared-memory architectures**
- ▶ **Programming via ...**
 - ▶ OpenMP
 - ▶ Java-Threads



- ▶ Abbr. for *Symmetric Multi Processing*
- ▶ Memory access time is uniform on all cores
- ▶ Limited scalability
- ▶ Example: *Intel Woodcrest*
 - ▶ Two cores per chip, 3.0 GHz
 - ▶ Each chip has 4 MB of L2 cache on-chip, shared by both cores
 - ▶ No off-chip cache
 - ▶ Bus: Frontsidebus



- ▶ Abbr. for *cache-coherent Non-Uniform Memory Architecture*
- ▶ Memory access time is non-uniform
- ▶ Scalable
- ▶ Example: *AMD Opteron*
 - ▶ Two cores per chip, 2.4 GHz
 - ▶ Each core has separate 1 MB of L2-cache on-chip
 - ▶ No off-chip cache
 - ▶ Interconnect: Hypertransport



- ▶ If there are multiple caches not shared by all cores in the system, the system takes care of the cache coherence.

- ▶ **Example:**

```
int a[some_number]; //shared by all threads
thread 1: a[0] = 23;      thread 2: a[1] = 42;
--- thread + memory synchronization (barrier) ---
thread 1: x = a[1];      thread 2: y = a[0];
```

- ▶ Both `a[0]` and `a[1]` are stored in caches of thread 1 and 2
- ▶ Changes to data in the cache is at first only visible for the CPU that modified its cache
- ▶ After synchronization point all threads need to have the same view of (shared) main memory

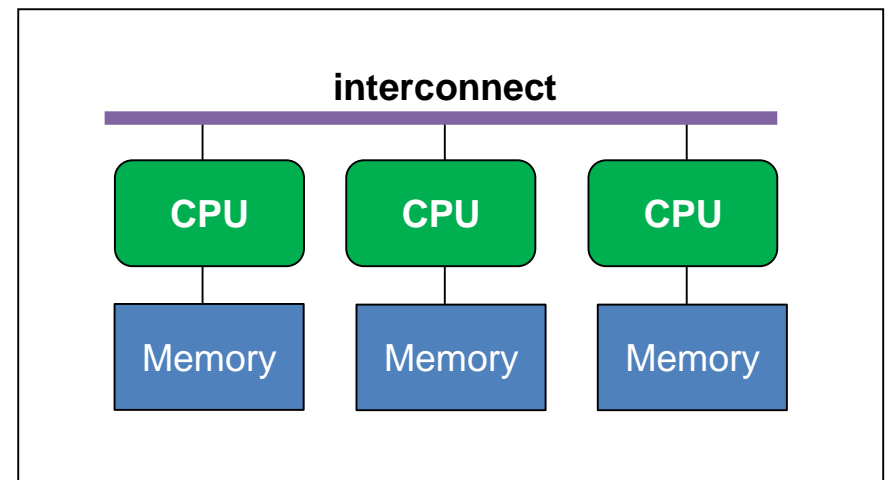
▶ **Computer Architectures**

- ▶ Basic Computer Architectures
- ▶ Shared-Memory Parallel Systems
- ▶ Distributed-Memory Parallel Systems

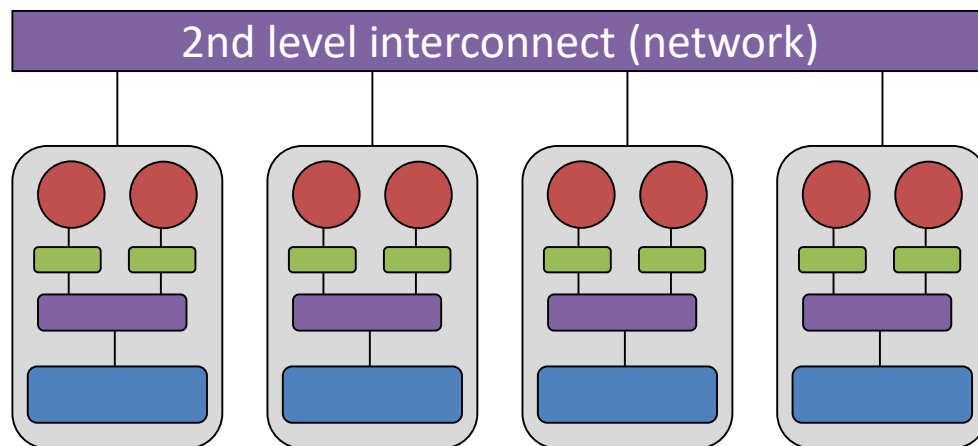
▶ **Parallelization at a Glance**

- ▶ Basic Concepts
- ▶ Parallelization Strategies
- ▶ Prominent Issues

- ▶ **Explicit data distribution**
- ▶ **Explicit communication**
- ▶ **Scalable**
- ▶ **Programming via MPI**



- ▶ **Various independent computers are connected to each other over a non-cache-coherent *second level interconnect***
 - ▶ Infiniband
 - ▶ Latency: $\leq 5 \mu\text{s}$
 - ▶ Bandwidth: $\geq 1200 \text{ MB/s}$
 - ▶ GigaBit Ethernet
 - ▶ Latency: $\leq 60 \mu\text{s}$
 - ▶ Bandwidth: $\geq 100 \text{ MB/s}$



Latency:
Time required to send a message of
size zero
(time to setup the communication)

Bandwidth:
Rate at which large messages (≥ 2
MB) are transferred

- ▶ **Computer Architectures**
 - ▶ Basic Computer Architectures
 - ▶ Shared-Memory Parallel Systems
 - ▶ Distributed-Memory Parallel Systems

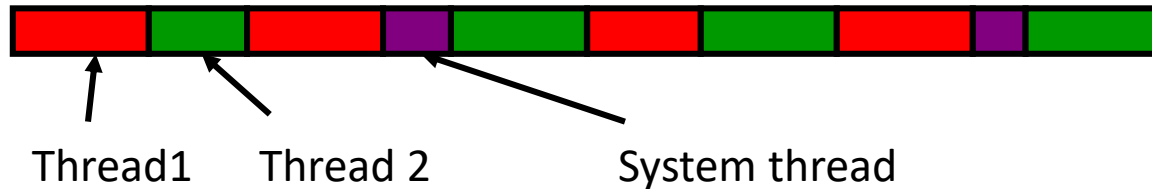
- ▶ **Parallelization at a Glance**
 - ▶ Basic Concepts
 - ▶ Parallelization Strategies
 - ▶ Prominent Issues

- ▶ **A process is the abstraction of a program in execution**
- ▶ **It can be in different states**
 - ▶ Running
 - ▶ Blocked
 - ▶ Waiting
- ▶ **Each process has its own address-space**
 - No common variables between processes

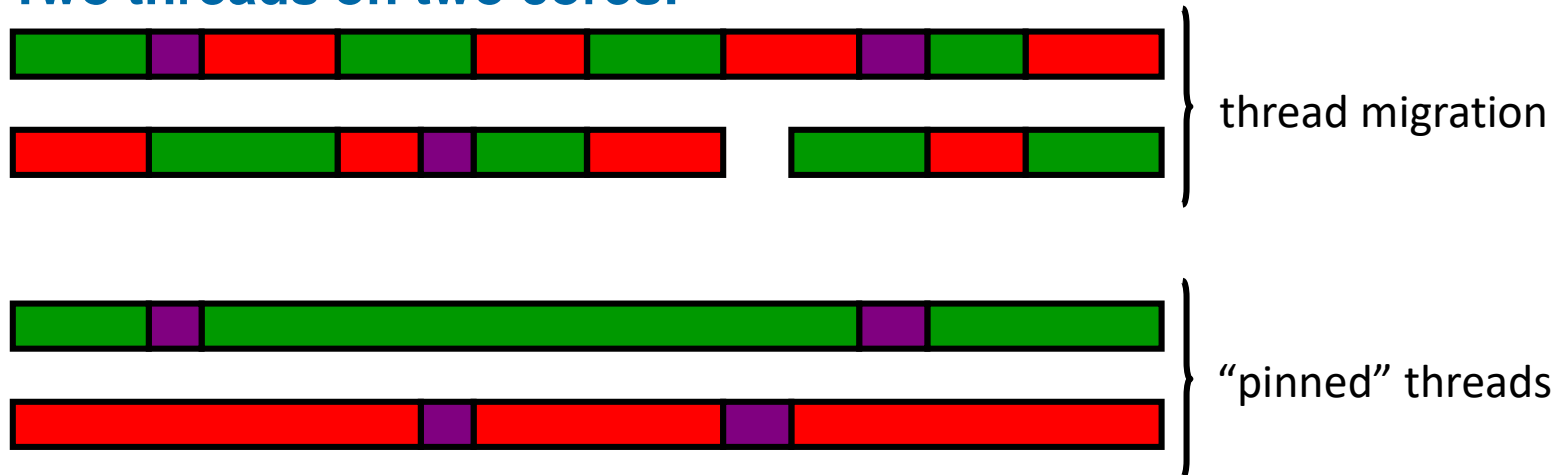
- ▶ A thread is a *lightweight* process
- ▶ In difference to a process, a thread shares the address-space with all other threads of the process it belongs to, but has its own stack.
 - Common variables between threads

- ▶ Even on a multi-socket / multi-core system you should not make any assumption which process / thread is executed when and where!

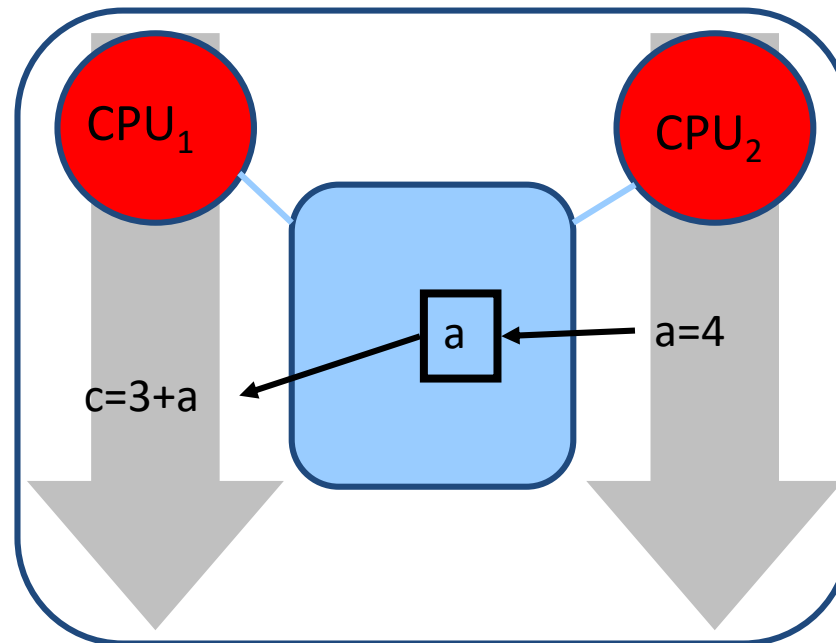
- ▶ Two threads on one core:



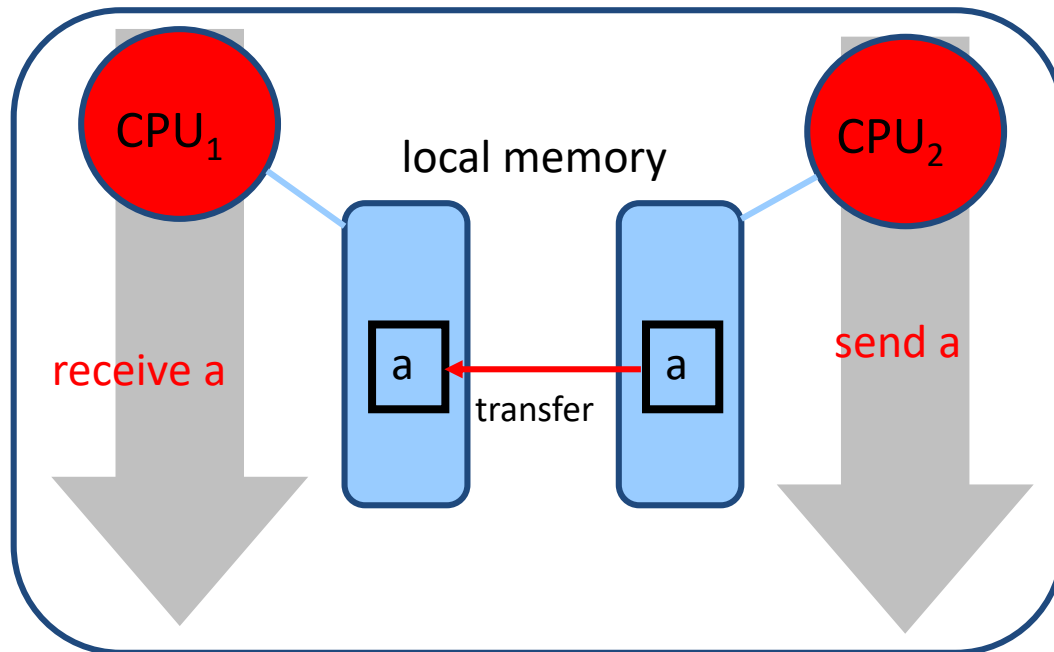
- ▶ Two threads on two cores:



- ▶ Memory can be accessed by several threads running on different cores in a multi-socket / multi-core system



- ▶ Each process has its own distinct memory
- ▶ Communication via *Message Passing*



- ▶ **Computer Architectures**
 - ▶ Basic Computer Architectures
 - ▶ Shared-Memory Parallel Systems
 - ▶ Distributed-Memory Parallel Systems

- ▶ **Parallelization at a Glance**
 - ▶ Basic Concepts
 - ▶ **Parallelization Strategies**
 - ▶ Prominent Issues

Speedup and Efficiency (1 / 2)

▶ Time using 1 CPU: $T(1)$

▶ Time using p CPUs: $T(p)$

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

▶ Measures how much fast the parallel computation is

▶ Efficiency E :
$$E(p) = \frac{S(p)}{p}$$

▶ Example:

- ▶ $T(1) = 6s, T(2) = 4s$

- $S(2) = \frac{6}{4} = \frac{3}{2} = 1.5$

- $E(2) = \frac{1.5}{2} = \frac{3}{4} = 0.75$

▶ Ideal case: $T(p) = T(1)/p$

- ▶ $S(p) = p$

- ▶ $E(p) = 1.0$

- ▶ Describes the influence of the serial part onto scalability (without taking any overhead into account).

- ▶ $S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{f \cdot T(1) + (1-f) \cdot \frac{T(1)}{p}} = \frac{1}{f + \frac{1-f}{p}}$

- ▶ f : serial part ($0 < f \leq 1$)

- ▶ $T(1)$: time using 1 CPU

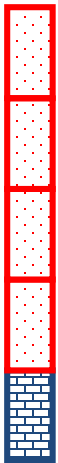
- ▶ $T(p)$: time using p CPUs

- ▶ $S(p)$: speedup; $S(p) = \frac{T(1)}{T(p)}$

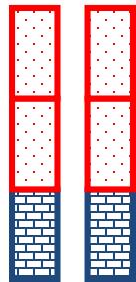
- ▶ $E(p)$: efficiency; $E(p) = \frac{S(p)}{p}$

- ▶ It is rather easy to scale to a small number of cores, but any parallelization is limited by the serial part of the program!

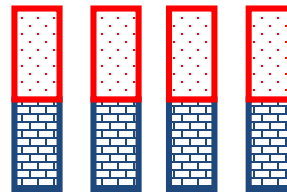
- ▶ If 80% (measured in program runtime) of your work can be parallelized and “just” 20% are still running sequential, then your speedup will be:



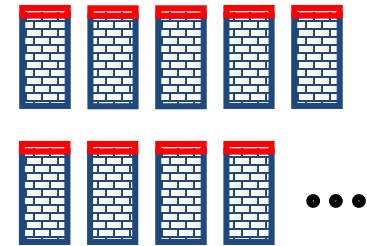
1 processor:
time: 100%
speedup: 1



2 processors:
time: 60%
speedup: 1.7

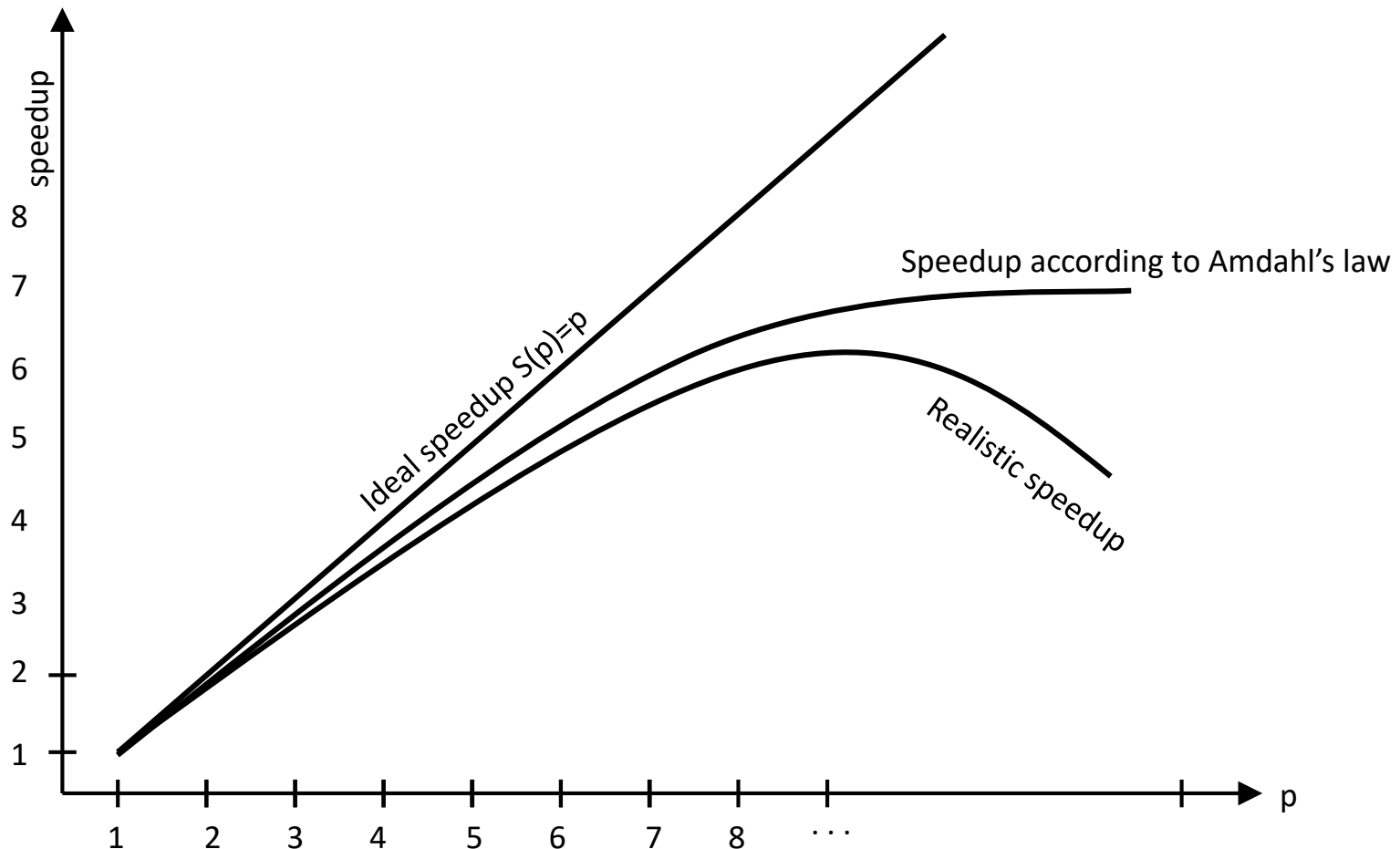


4 processors:
time: 40%
speedup: 2.5



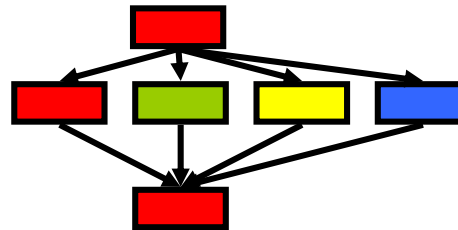
∞ processors:
time: 20%
speedup: 5

- ▶ After the initial parallelization of a program, you will typically see speedup curves like this:

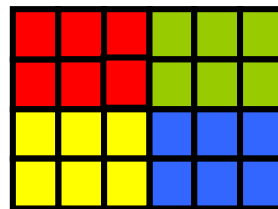


► Chances for concurrent execution:

- Look for tasks that can be executed simultaneously
(task decomposition)



- Decompose data into distinct chunks to be processed independently
(data decomposition)

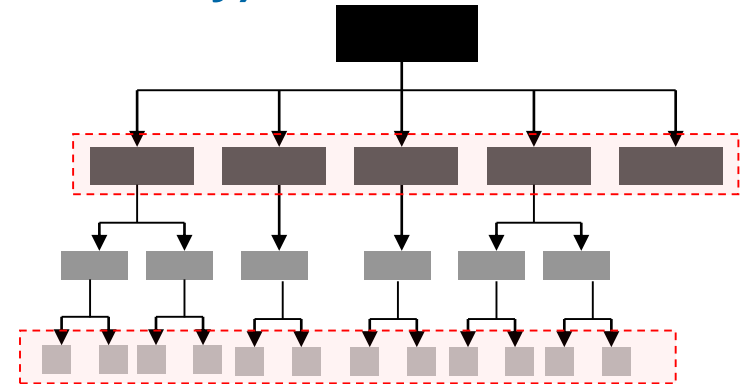


▶ Parallelization on a High Level (low granularity)

- ▶ Chances of low synchronization / communication cost
- ▶ Danger of load balancing issues

▶ Parallelization on a Low Level (high granularity)

- ▶ Danger of high synchronization / communication cost
- ▶ Chances of avoiding load balancing issues



- ▶ **Compute intensive programs may employ multiple levels of parallelization, maybe even with multiple parallelization paradigms (hybrid parallelization).**

- ▶ **Computer Architectures**
 - ▶ Basic Computer Architectures
 - ▶ Shared-Memory Parallel Systems
 - ▶ Distributed-Memory Parallel Systems

- ▶ **Parallelization at a Glance**
 - ▶ Basic Concepts
 - ▶ Parallelization Strategies
 - ▶ Prominent Issues

- ▶ **You can still run into all issues of Serial Programming ☹ !**
- ▶ **Additional issues:**
 - ▶ Is your parallelization correct?
 - ▶ It is harder to debug parallel code than serial code!
- ▶ **Specific issues of Parallel Programming:**
 - ▶ Introduction of overhead by parallelization
 - ▶ Data Races / Race Conditions
 - ▶ Deadlocks
 - ▶ Load Balancing
 - ▶ Serialization
 - ▶ Irreproducibility / Different numerical results

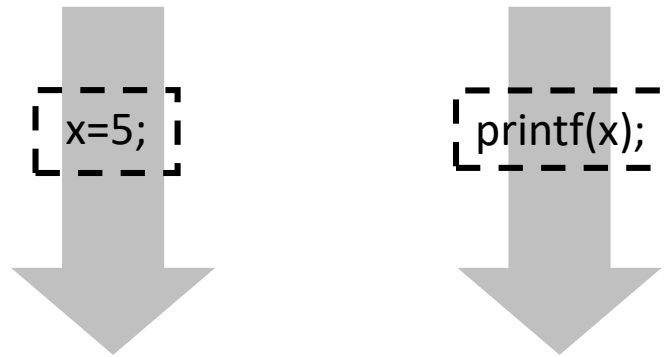
- ▶ **Overhead introduced by the parallelization:**
 - ▶ Time to start / end / manage threads
 - ▶ Time to send / exchange data
 - ▶ Time spent in synchronization of threads / processes

- ▶ **With parallelization:**
 - ▶ The total CPU time increases,
 - ▶ The Wall time decreases,
 - ▶ The System time stays the same.

- ▶ **Efficient parallelization is about minimizing the overhead introduced by the parallelization itself!**

- ▶ **Data Race: Concurrent access of the same memory location by multiple threads without proper synchronization**

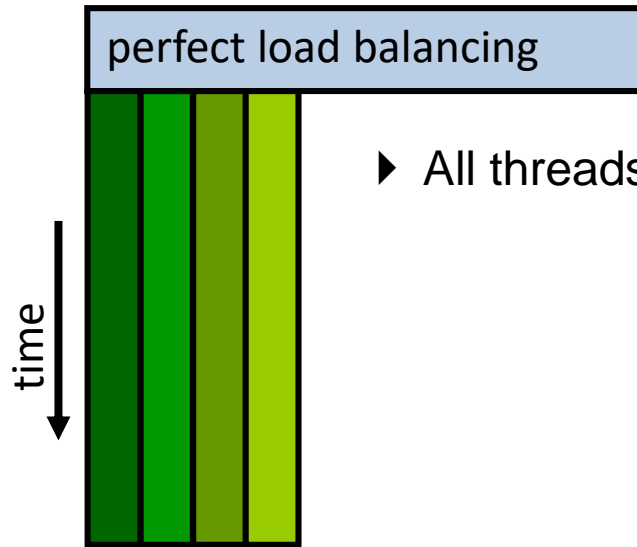
- ▶ Let x be initialized with 1



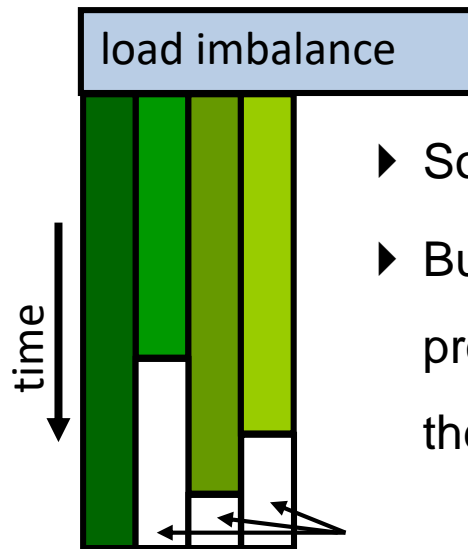
- ▶ Depending on which thread is faster, you will see either 1 or 5
 - ▶ Result is nondeterministic (i.e. depends on OS scheduling)
-
- ▶ **Data Races (and how to detect and avoid them) will be covered in more detail later!**

- ▶ When two or more threads / processes are waiting for another to release a resource in a circular chain, the program appears to „hang“:





- ▶ All threads / processes finish at the same time



- ▶ Some threads / processes take longer than others
- ▶ But: All threads / processes have to wait for the slowest thread / process, which is thus limiting the scalability

- ▶ **Serialization: When threads / processes wait „too much“**
 - ▶ Limited scalability, if at all
- ▶ **Simple (and stupid) example:**

