Parallel Computer Architecture Concepts

TDDE35 Lecture 1

Christoph Kessler

PELAB / IDA
Linköping University
Sweden

Outline

Lecture 1: Parallel Computer Architecture Concepts

- Parallel computer, multiprocessor, multicomputer
- SIMD vs. MIMD execution
- Shared memory vs. Distributed memory architecture
- Interconnection networks
- Parallel architecture design concepts
  - Instruction-level parallelism
  - Hardware multithreading
  - Multi-core and many-core
  - Accelerators and heterogeneous systems
  - Clusters
- Implications for programming and algorithm design

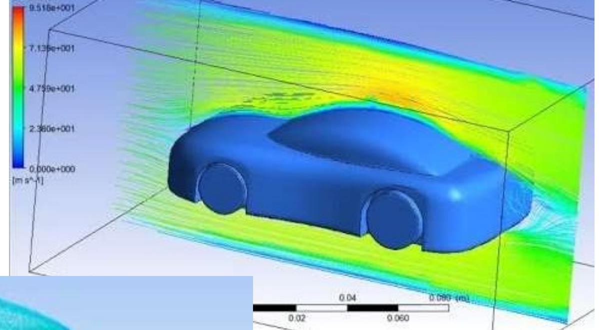
Traditional Use of Parallel Computing: Large-Scale HPC Applications

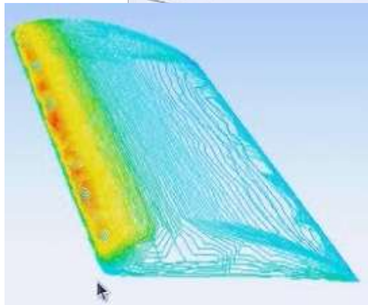
- **High Performance Computing (HPC)**
  - E.g. climate simulations, particle physics, proteine docking, …
  - Much computational work
    (in FLOPs, floatingpoint operations)
  - Often, large data sets

- Single-CPU computers and even today's multicore processors cannot provide such massive computation power

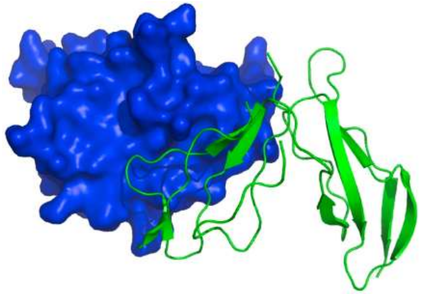
- Aggregate LOTS of computers → **Clusters**
  - Need scalable parallel algorithms
  - Need exploit multiple levels of parallelism
  - Cost of communication, memory access

High Performance Computing Application Areas (Selection)



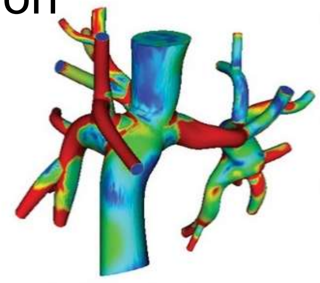


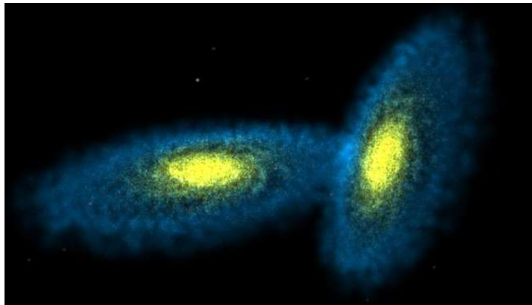
- Computational Fluid Dynamics
- Weather Forecasting and Climate Simulation
- Aerodynamics / Air Flow Simulations and Optimization
- Structural Engineering
- Fuel-Efficient Aircraft Design
- Molecular Modelling
- Material Science
- Computational Chemistry
- Battery Simulation and Optimization
- Galaxy Simulations
- Earthquake Engineering, Oil Reservoir Simulation
- Flood Prediction
- Bioinformatics (DNA Pattern Matching, Proteine Docking)
- Fluid / Structural Interaction
- Blood Flow Simulation
- fRMI Image Analysis
- ...

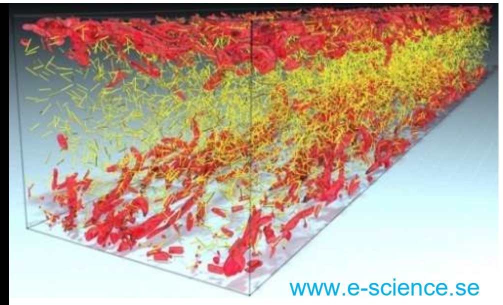


Simulation as a “third pillar” of natural sciences next to theory and traditional experimentation

“E-Science”



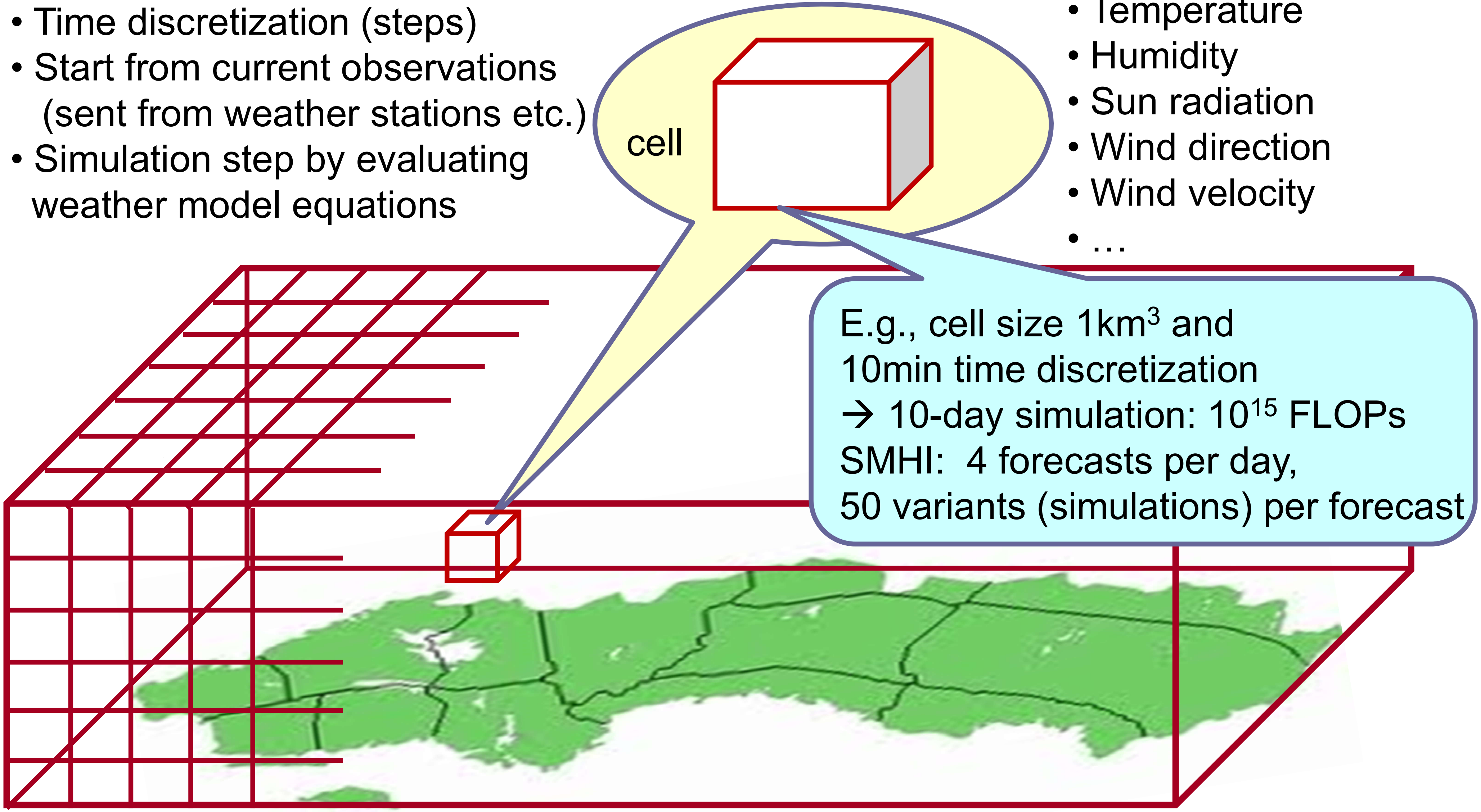




Example: Weather Forecast (very simplified…)

- 3D Space discretization (cells)
- Time discretization (steps)
- Start from current observations (sent from weather stations etc.)
- Simulation step by evaluating weather model equations

- Air pressure
- Temperature
- Humidity
- Sun radiation
- Wind direction
- Wind velocity
- …



https://www.smhi.se/kunskapsbanken/meteorologi/sa-gor-smhi-en-vaderprognos-1.4662

Another Classical Use of Parallel Computing: Parallel Embedded Computing

- **High-performance embedded computing**
  - E.g. on-board realtime image/video processing, gaming, …
  - Much computational work
    (often fixed point operations)
  - Often, in energy-constrained mobile devices

- Sequential programs on single-core computers
  cannot provide sufficient computation power
  at a reasonable power budget

- Use many small cores at low frequency
  - Need scalable parallel algorithms
  - Cost of communication, memory access
  - Energy cost (Power x Time)

More Recent Use of Parallel Computing: Big-Data Analytics Applications

- **Big Data Analytics**
  - Data access intensive (disk I/O, memory accesses)
    - Typically, very large data sets (GB … TB … PB … EB …)
  - Also some computational work for combining/aggregating data
  - E.g. data center applications, business analytics, click stream analysis, scientific data analysis, machine learning, …
  - Soft real-time requirements on interactive querys
- Single-CPU and multicore processors cannot provide such massive computation power and I/O bandwidth+capacity
- Aggregate LOTS of computers → **Clusters**
  - Need scalable parallel algorithms
  - Need to exploit multiple levels of parallelism
  - Fault tolerance

HPC vs Big-Data Computing

- Both need **parallel computing**
- **Same kind of hardware** – Clusters of (multicore) servers
- Same OS family (Linux)
- **Different programming models**, languages, and tools

HPC	Big-Data
HPC application	Big-Data application
HPC prog. languages: Fortran, C/C++ (Python)	Big-Data prog. languages: Java, Scala, Python, …
Par. programming models: MPI, OpenMP, …	Par. programming models: MapReduce, Spark, …
Scientific computing libraries: BLAS, …	Big-data storage/access: HDFS, …
OS: Linux	OS: Linux
HW: Cluster	HW: Cluster

More in TDDE31 master-level course on Big-Data Analytics

→ Let us start with the common basis: Parallel computer architecture

Parallel Computer

A parallel computer is a computer consisting of

\+ two or more processors
that can cooperate and communicate
to solve a large problem faster,

\+ one or more memory modules,

\+ an interconnection network
that connects processors with each other
and/or with the memory modules.

Multiprocessor: tightly connected processors, e.g. shared memory

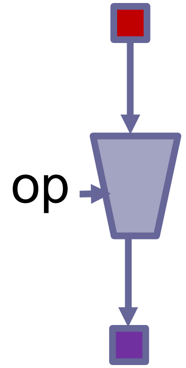
Multicomputer: more loosely connected, e.g. distributed memory

Parallel Computer Architecture Concepts

Classification of parallel computer architectures:

- by control structure
- by memory organization
  - in particular, Distributed memory vs. Shared memory
- by interconnection network topology

Classification by Control Structure [Flynn'72]



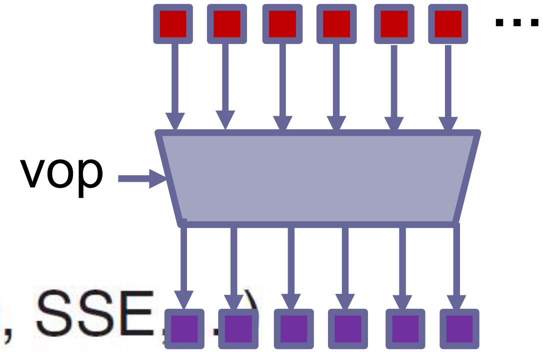
SISD single instruction stream, single data stream

+ sequential. OK where performance is not an issue.

SIMD single instruction stream, multiple data streams

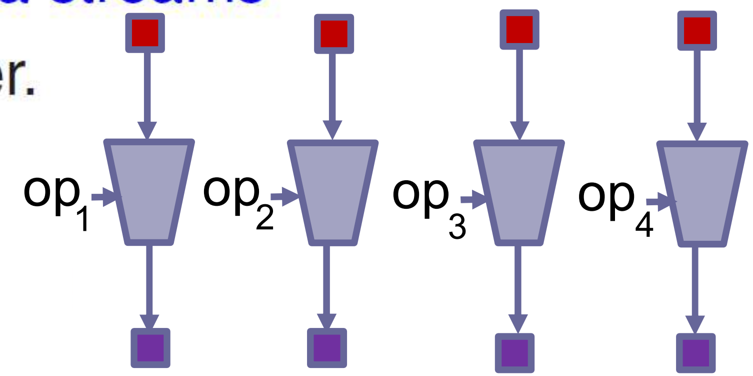
Common clock, common program memory, common program counter.

+ VLIW processors
+ traditional vector processors
+ traditional array computers
+ SIMD instructions on wide data words (e.g. Altivec, SSE, ...)



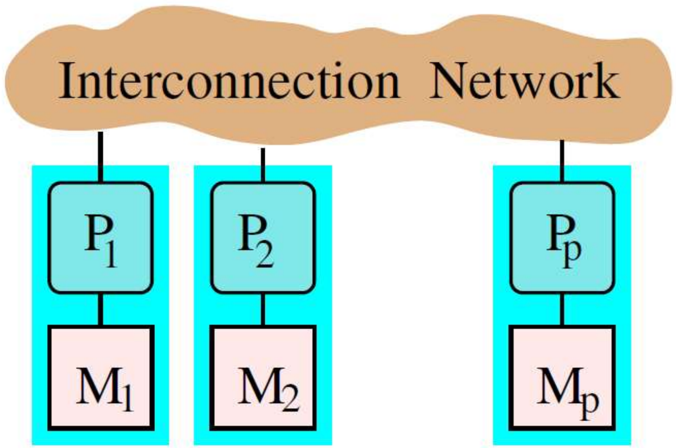
MIMD multiple instruction streams, multiple data streams

Each processor has its own program counter.



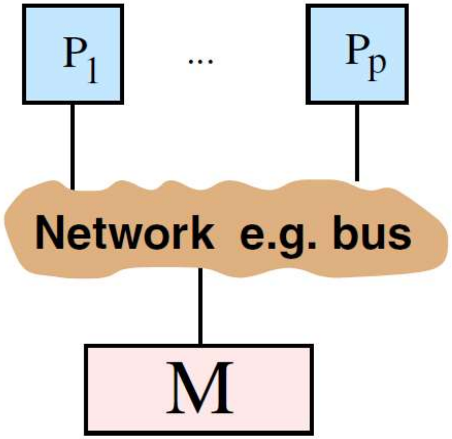
Hybrid forms

Classification by Memory Organization



Distributed memory system (DMS)

e.g. (traditional) HPC cluster



Shared memory system (SMS)

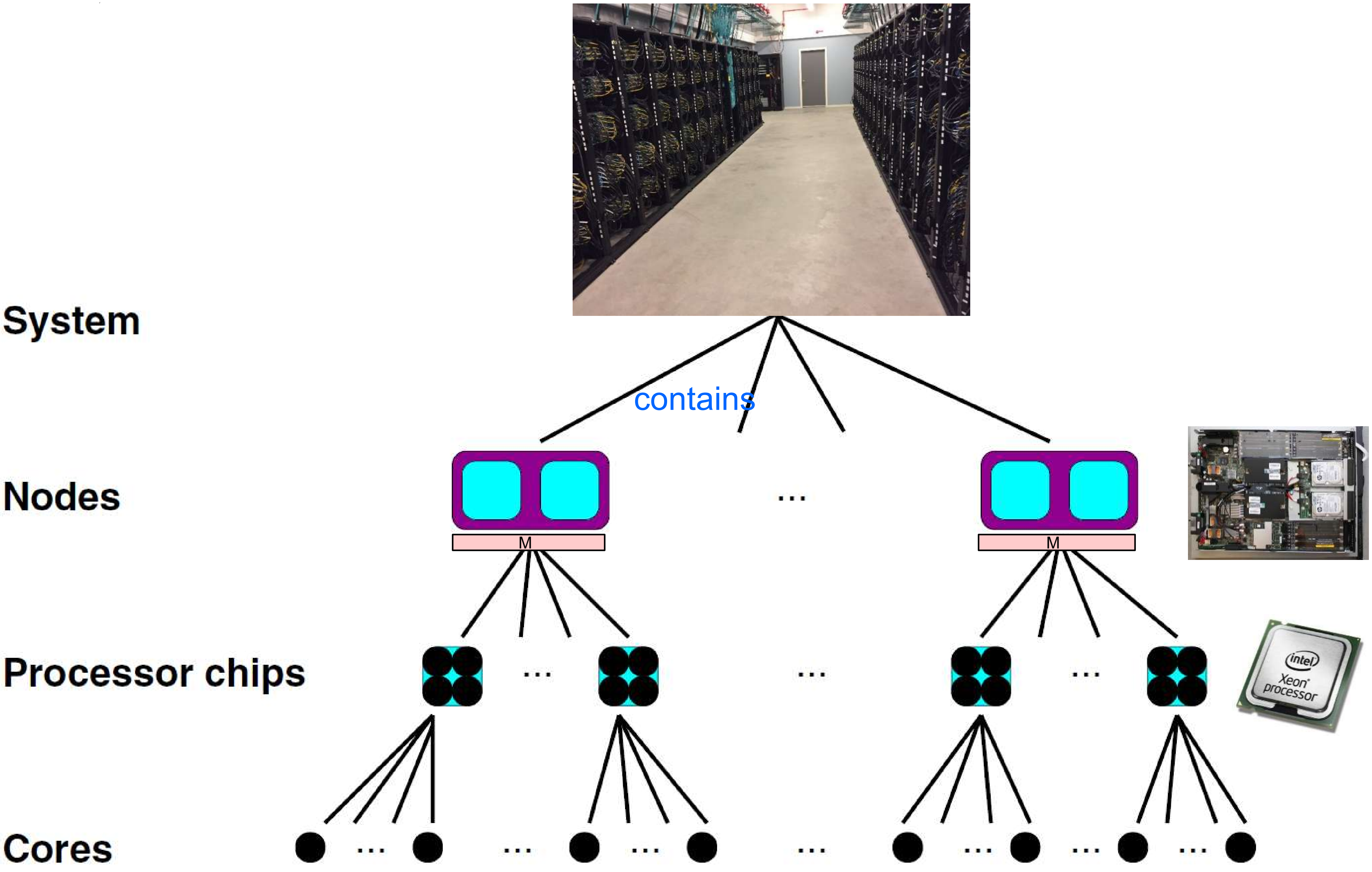
e.g. multiprocessor (SMP) or computer with a standard multicore CPU

Most common today in HPC and Data centers:

Hybrid memory system

- Cluster (distributed memory) of hundreds, thousands of shared-memory servers each containing one or several multi-core CPUs

Hybrid (Distributed + Shared) Memory



Interconnection Networks (1)

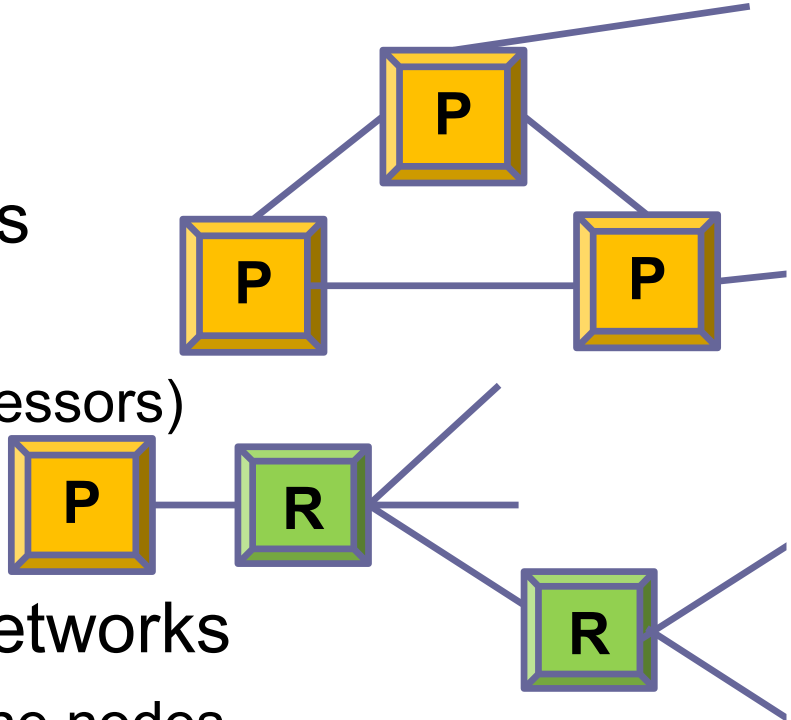
- **Network**
  = physical interconnection medium (wires, switches)
  \+ communication protocol

  (a) connecting cluster nodes with each other (DMS)

  (b) connecting processors with memory modules (SMS)

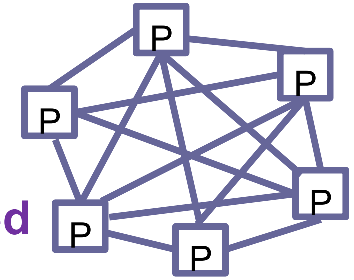
Classification

- Direct / static interconnection networks
  - connecting nodes directly to each other
  - Hardware routers (communication coprocessors) can be used to offload processors from most communication work
- Switched / dynamic interconnection networks
  - Graphs of routers (switches) connecting the nodes

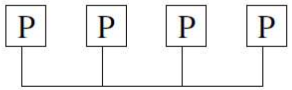


Interconnection Networks (2): Simple Topologies

fully connected

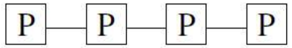


bus

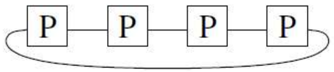


1 wire – bus saturation with many processors
e.g. Ethernet

linear array

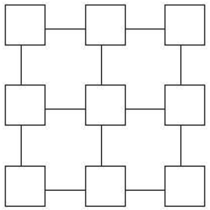


ring

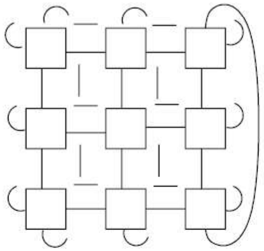


e.g. Token Ring

2D grid

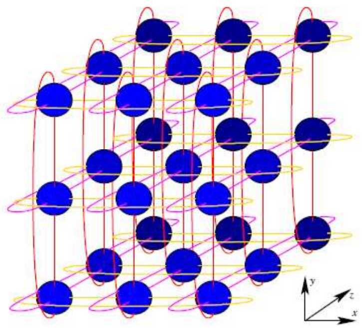


torus:

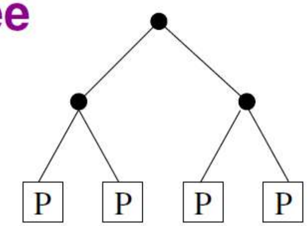


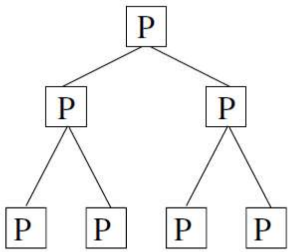
3D grid

3D torus



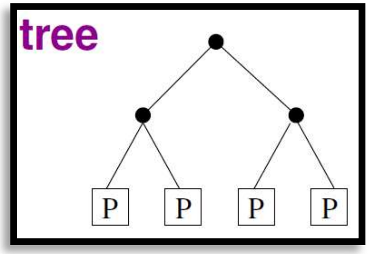
tree



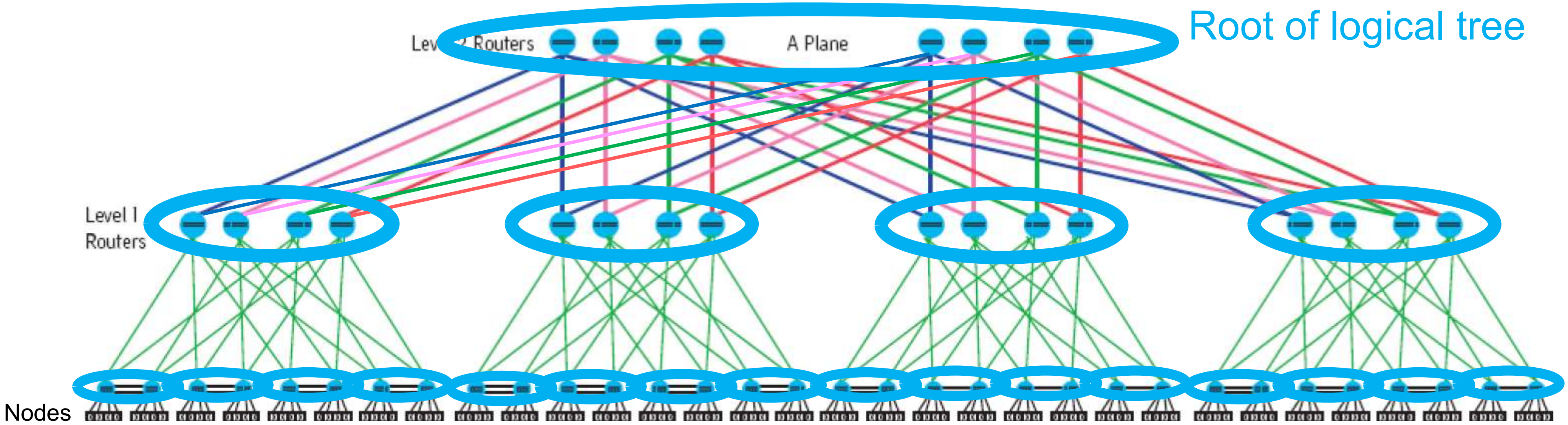


root processor
is bottleneck

Interconnection Networks (3): Fat-Tree Network



- Tree network extended for higher bandwidth (more switches, more links) closer to the root
  - Higher cost, but reduces bandwidth bottleneck



Example implementation (SGI):
Logically a 4-ary tree,
physically a butterfly-like network

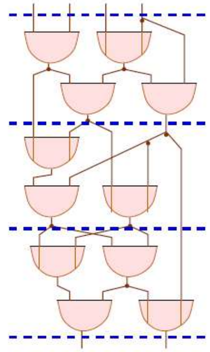
- Example: Infiniband network (Mellanox / Nvidia),
  Omnipath network (Intel)

More about Interconnection Networks

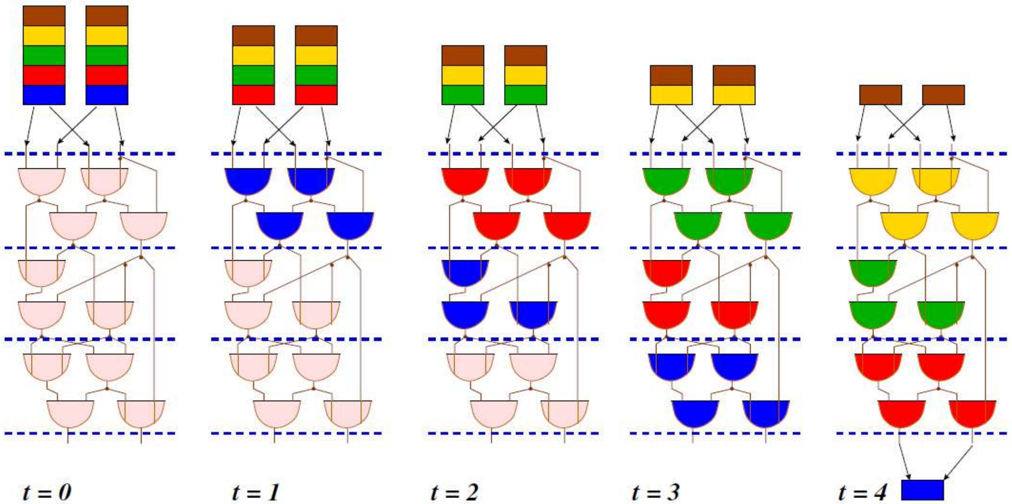
- Hypercube, Crossbar, Butterfly, Hybrid networks… → TDDE65
- Switching and routing algorithms
- **Discussion of interconnection network properties**
  - Cost (#switches, #links)
  - Scalability
    (asymptotically, cost grows not much faster than #nodes)
  - Node degree
  - Longest path (→ latency)
  - Accumulated bandwidth
  - Fault tolerance (worst-case impact of node or switch failure)
  - …

Instruction Level Parallelism (1): Pipelined Execution in the ALU

Principle: SIMD + pipelining
cf. assembly line manufacturing of cars etc.



+ Idea: partition "deep" arithmetic circuits (e.g., floatingpoint-adder) into $d > 1$ horizontal layers, called stages, of about equal depth. Reduce clock cycle time such that each stage needs one cycle.
+ Intermediate results of stage k are forwarded to stage $k + 1$
+ The operands and result(s) are vectors, sequences (arrays) of floats
+ All stages work simultaneously, but on different components of the vectors
+ Stage k works on l-th vector component in cycle $k + l$
+ First result available after d cycles, a startup phase of $d - 1$ cycles is needed to fill the pipeline



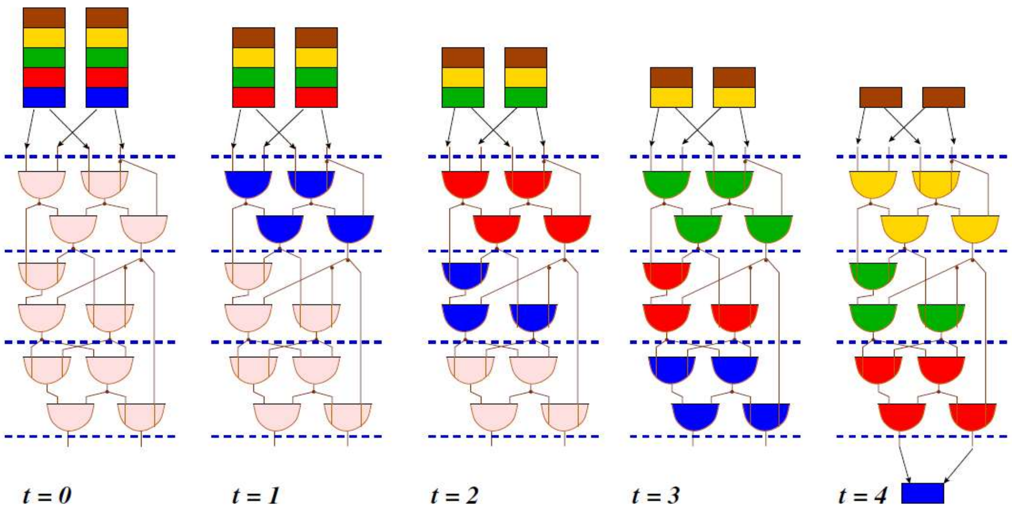
SIMD Computing with Pipelined Vector Units

Used in early supercomputers: vector supercomputers by Cray (1970s, 1980s), Fujitsu, …
Today, automatically pipelined execution also of *different* instructions is standard in CPUs

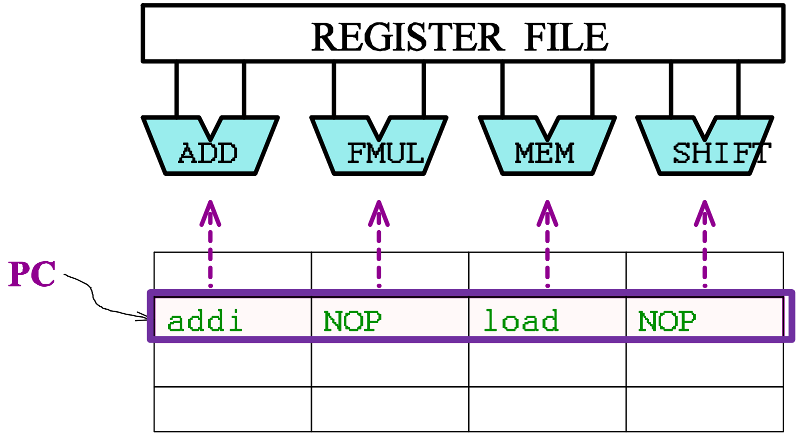
- A vector operation, e.g. $C[1:N] \leftarrow A[1:N] + B[1:N]$ (elementwise addition) takes $N+d-1$ cycles (compared to $N \times d$ cycles without pipelining)
- Condition: All component computations of a vector operation must be of same operation type and independent of each other
- Scalar operations take d cycles — no improvement.
- Programs must be vectorized (by the programmer or compiler)

\+ Stage k works on l-th vector component in cycle $k+l$

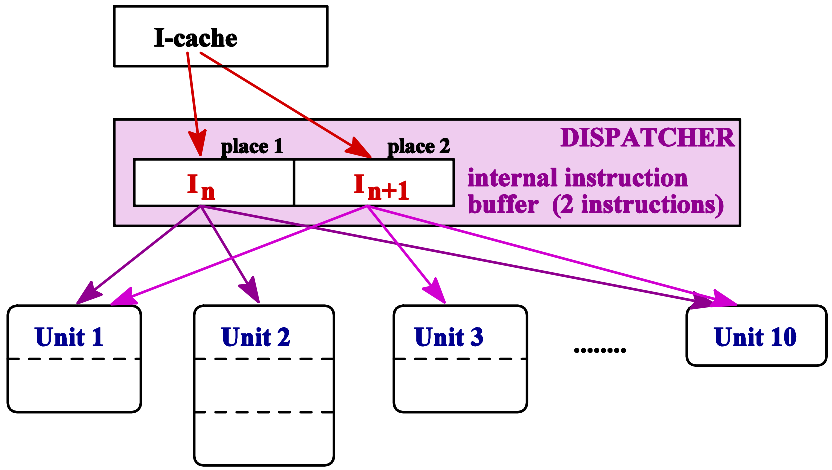
\+ First result available after d cycles, a startup phase of $d-1$ cycles is needed to fill the pipeline



Instruction-Level Parallelism (2): VLIW and Superscalar

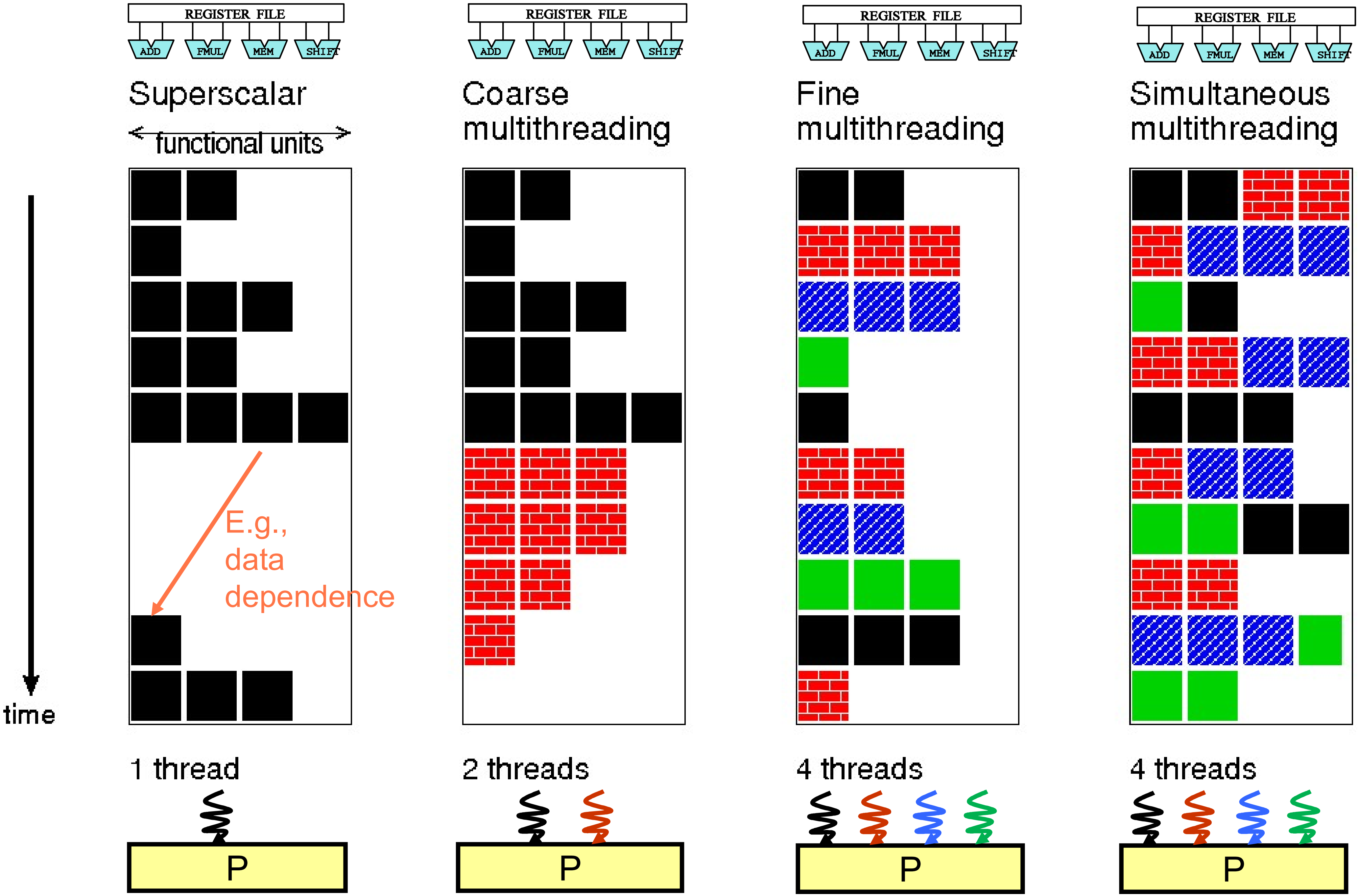


- Multiple functional units in parallel
- Try to run more than 1 instruction per cc
- **2 main paradigms:**
  - **VLIW** (very large instruction word) architecture ^
    - Parallelism is explicit, programmer-/compiler-managed (hard)
    - Energy-efficient
    - Popular in digital signal processors
  - **Superscalar** architecture →
    - Sequential instruction stream
    - Hardware-managed dispatch
    - power + area overhead



- **ILP in applications is usually limited** (= the ”ILP wall”)
  - typ. ≤ 3...4 instructions can be issued simultaneously
  - Due to control and data dependences in applications
  - Larger issue widths give at best marginal gains
- **Solution: Multithread the application and the processor**

Hardware Multithreading



Background: Hardware multithreading vs. multicore

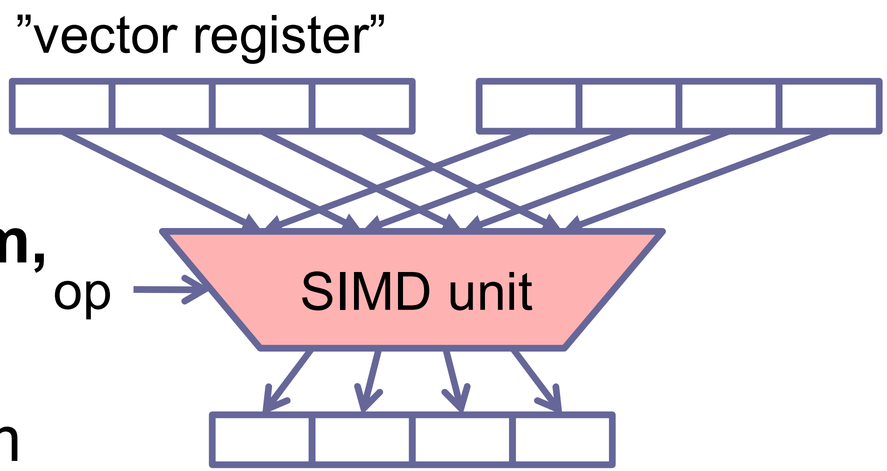
- **Multicore** = *multiple* separate processors placed on a single chip,
  - operating truly in parallel
  - sharing last-level cache and off-chip memory access interface (the "un-core").

- **Hardware multithreading**
  - a *single* processor (e.g., a core) automatically emulates *multiple* **virtual processors** (the **hardware threads**) by *timesharing* its data path (e.g., functional units)
    - Hardware threads are managed entirely by the processor's *hardware* (*not* by the OS – the OS has no influence on it).
    - Each piece of hardware (e.g., the floatingpoint unit of the processor) can only be used by *one* of the hardware threads at a time.
    - Hardware threads co-exist only by their different register sets. The *hardware* switches context by switching from one register set to the next one.
    - **Coarse-grain HW multithreading**: processor hardware context-switches on cache misses or other long-latency operations to the next hardware thread
    - **Fine-grain HW multithreading**: processor hardware context-switches *after every clock-cycle* (round-robbin hardware scheduling)
    - **Simultaneous multithreading / hyperthreading**: the HW scheduler can start execution of multiple instructions (on disjoint sub-datapaths) coming from *different* HW threads (thus, independent) in the *same* clock cycle.

Background: Hardware multithreading vs. multicore (cont.)

- Hardware multithreading only gives additional speedup if long-latency instructions (e.g. cache-missing loads) of different threads can *overlap* in time with instructions from other hardware threads, by continuing running in the (hardware) background after a hardware context switch. This is used excessively in today's GPUs, to hide the high memory latency.
- In both cases (multicore, hardware multithreading) the OS sees multiple processors sharing memory.
- Of course, both concepts can be **combined**: Today's CPUs have multiple cores, each of which is hardware-multithreaded.
- **Caution**: Hardware multithreading has *nothing* to do with *software threads* (created/managed by OS) or the OS CPU scheduler! Software threads and hardware threads are orthogonal concepts – each hardware thread can be time-shared among multiple software threads by the OS's *software* context switch and scheduler.

SIMD Instructions in modern CPUs



- Recall:
  SIMD = "Single Instruction stream, Multiple Data streams"
  - single thread of control flow
  - restricted form of data parallelism
    - apply the same primitive operation (*a single instruction*) in parallel to multiple data elements stored contiguously
- Arithmetic-logical units of CPUs: the datapath width is at least the width of widest built-in data type (e.g. `long double`, 128bit)
- SIMD-enabled arithmetic-logical units exploit full datapath width
  - use long "vector registers"
    - each holding multiple data elements of shorter data types
- Common today: 256, 512 bit SIMD extensions of the instruction set
  - MMX, SSE, SSE2, SSE3, Altivec, VMX, Neon, …
- Performance boost for operations on shorter data types
- Area- and energy-efficient
- Code to be rewritten ("vectorized") by programmer or compiler
- Does not help with the main memory access bandwidth bottleneck

The Memory Wall

- **Performance gap CPU – Memory**
- **Memory hierarchy**
- **Increasing cache sizes shows diminishing returns**
  - Costs power and chip area
    - GPUs spend the area instead on many simple cores with little memory
  - Relies on good data locality in the application
- **What if there is no / little data locality?**
  - Irregular applications,
    e.g. sorting, searching, optimization...
- **Solution: Spread out / overlap memory access delay**
  - Programmer/Compiler: Prefetching, on-chip pipelining,
    SW-managed on-chip buffers
  - Generally: Hardware multithreading, again!

Moore’s Law

Gordon Moore (1929-2023), co-founder of Intel

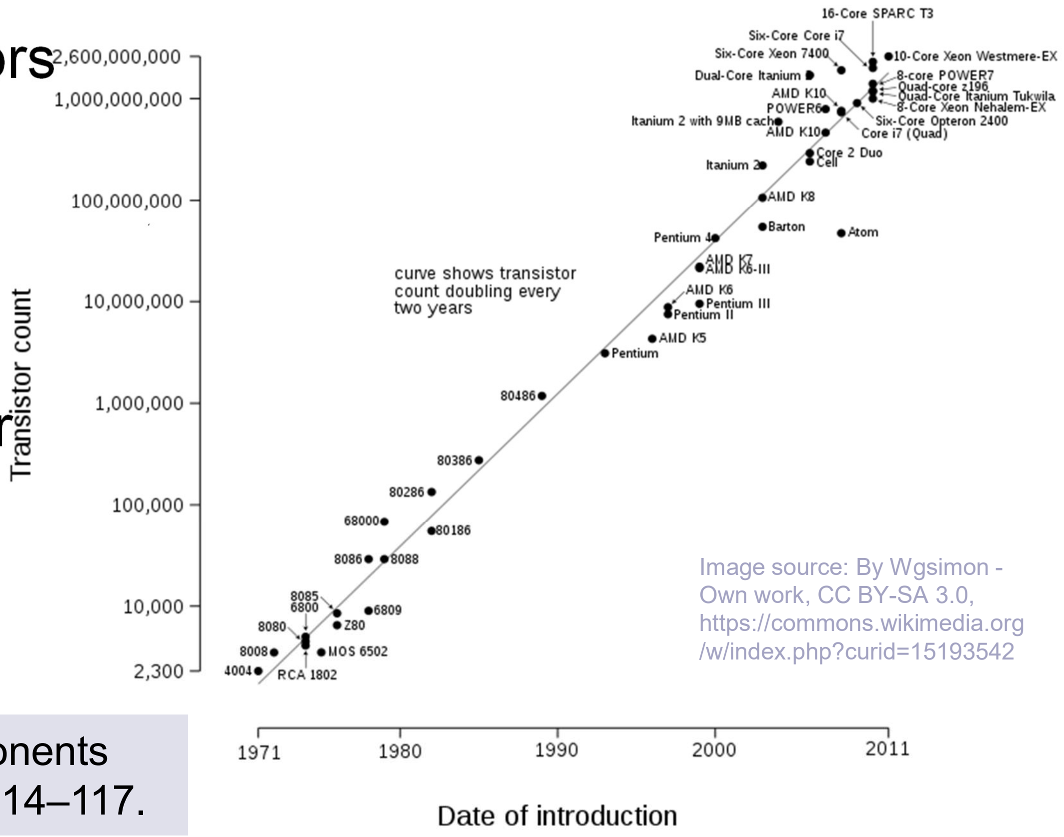
- **Prediction** (1965/1975):
  The number of transistors per mm^2 chip area doubles approximately every 2 years
  [at about equal production cost]
  - Exponential increase due to miniaturization in semiconductors

→ A self-fulfilling prophecy through 50 years!

→ Some slowdown since 2014: still exponential growth of transistor density (albeit at lower pace)

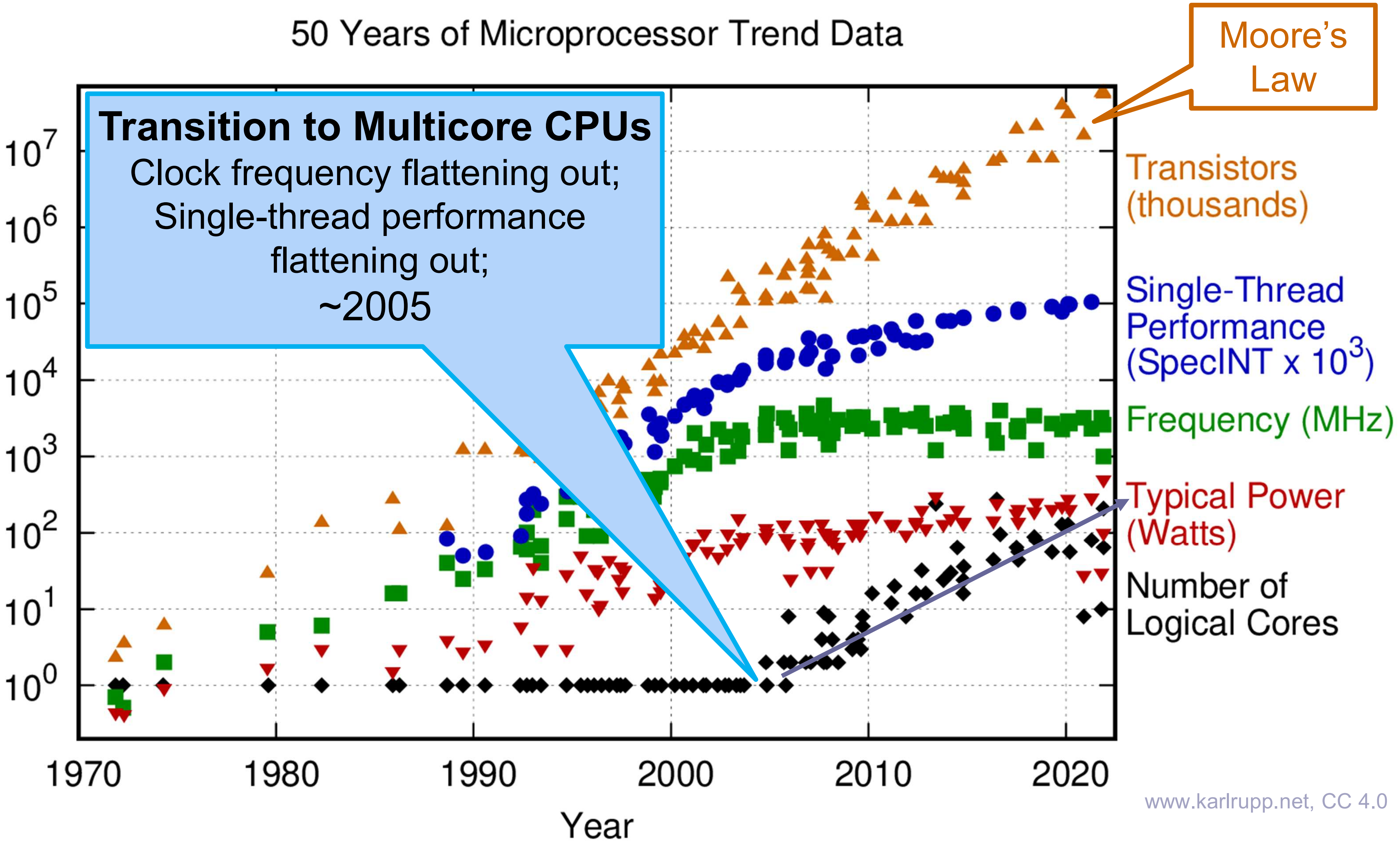
→ Soon running into physical and economical limits

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Gordon Moore (April 19, 1965). "Cramming More Components onto Integrated Circuits". *Electronics Magazine.* **38** (8): 114–117.

CPU Performance Development since 1970



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp Adapted for trend in number of cores.

The Power Issue

- Power = Static (leakage) power + Dynamic (switching) power
- Dynamic power ~ Voltage^2 * Clock frequency
  where Clock frequency approx. ~ voltage

  → Dynamic power ~ Frequency^3
- Total power ~ #processors

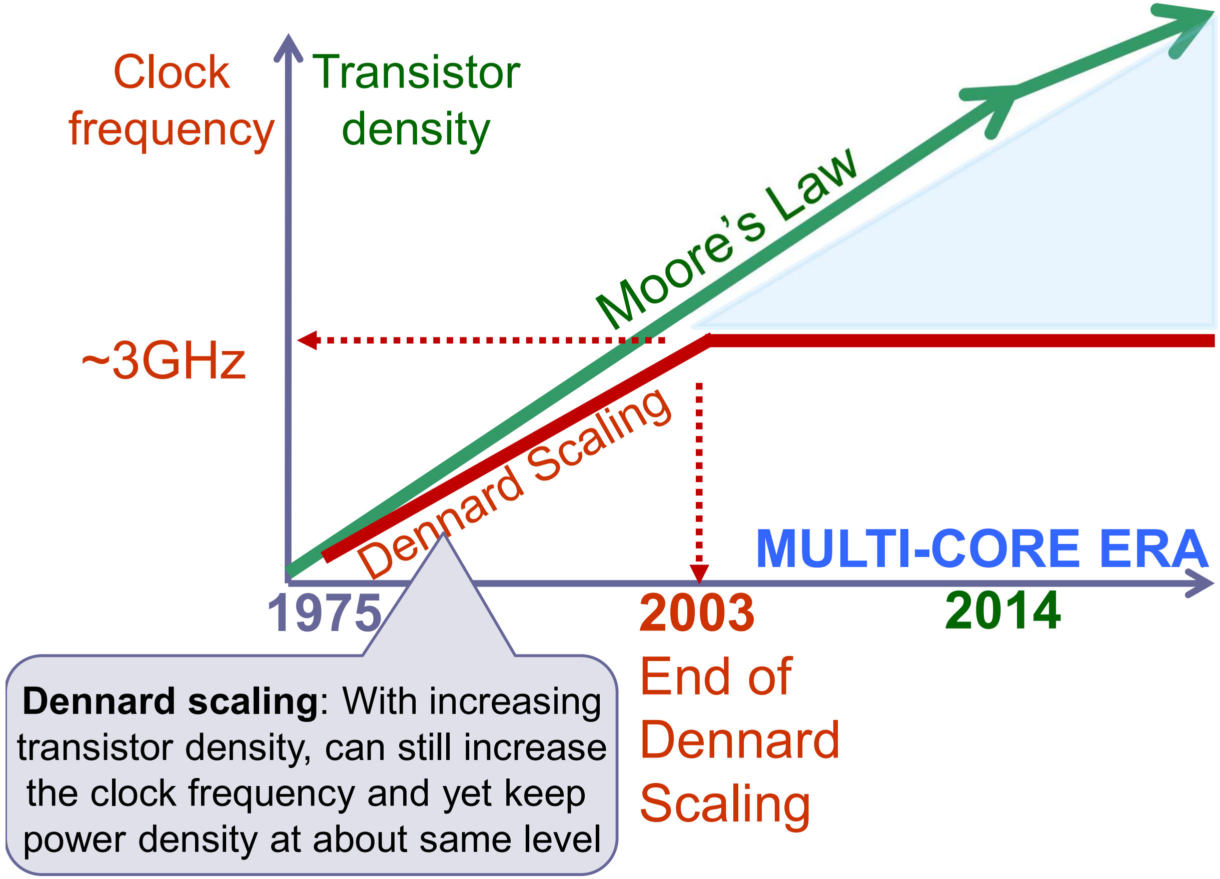
Processor architecture	#cores	Voltage	Frequency	Performance	Power	Power efficiency [Gflops/W]
Classical superscalar	1x	1x	1x	1x	1x	1x
"Faster" superscalar	1x	1.5x	1.5x	1.5x	3.3x	0.45x
Multi-core	2x	0.75x	0.75x	1.5x	0.8x	1.88x

Source: J. Dongarra, 2009

→ Preferable to use multiple slower processors than one superfast processor

... PROVIDED THAT the application can be parallelized efficiently!

Moore's Law vs. Clock Frequency



- **#Transistors / mm^2 still growing exponentially according to Moore's Law** (but with slightly lower slope since ~2014)
- **Clock speed hitting thermal limits of air-cooled CMOS ~2003**, due to end of Dennard Scaling

More transistors + Limited frequency
⇒ More cores

Solution for CPU Design: Multicore + Multithreading

- Single-thread performance does not improve any more since ca. 2003
  - ILP wall
  - Memory wall
  - Power wall (end of “Dennard Scaling”)

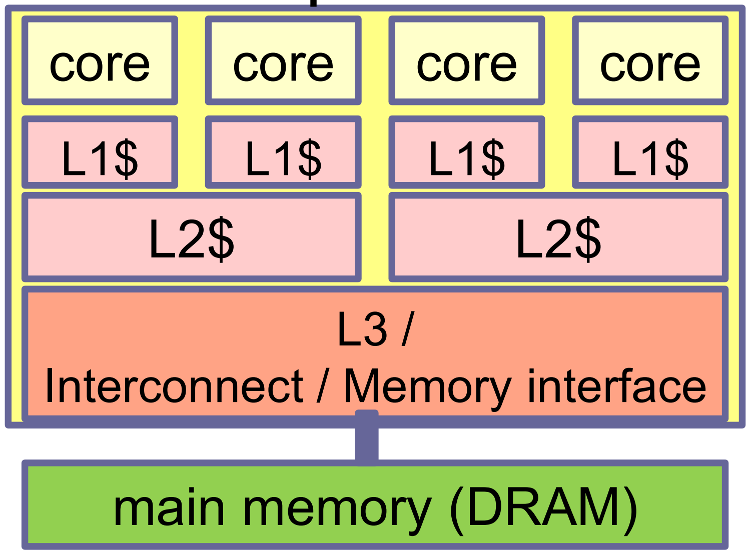
- but thanks to Moore’s Law continuing, we could still put more cores on a chip
  - And hardware-multithread the cores to hide (some) memory latency
  - All major chip manufacturers produce multicore CPUs today

Main features of a multicore system

- A parallel computer
- There are multiple computational cores on the same CPU chip.
  - Homogeneous multicore (same core type)
  - Heterogeneous multicore (different core types)
- The cores might have (small) private on-chip memory modules and/or access to on-chip memory shared by several cores.
- The cores have access to a common off-chip main memory
- There is a way by which these cores communicate with each other and/or with the environment.

Standard CPU Multicore Designs

- Standard desktop/server CPUs have a few ... up to ~32 cores with shared off-chip main memory
  - On-chip cache (typ., 3 levels)
    - L1-cache mostly core-private
    - L2-cache often shared by groups of cores, L3 often by all
  - Memory access interface shared by all or groups of cores

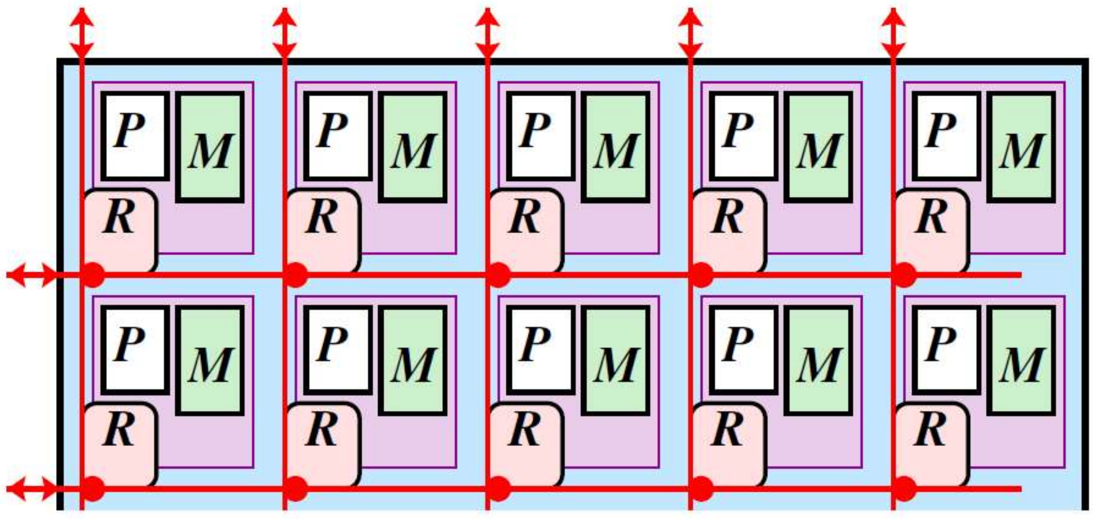


- Caching → multiple copies of the same data item
  - Writing to one copy (only) causes inconsistency
  - Shared memory coherence mechanism to enforce automatic updating or invalidation of all copies around

→ More about shared-memory architecture, caches, data locality, consistency issues and coherence protocols in TDDE65/TDDD56

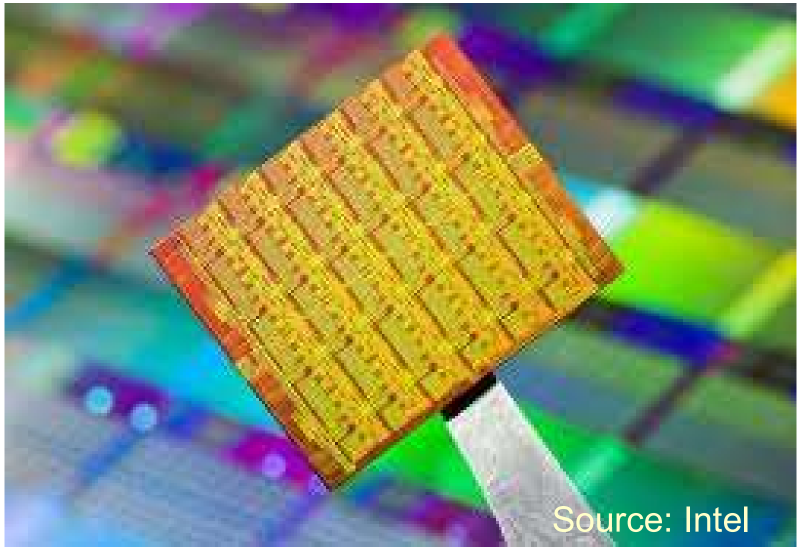
Scaling Up: Network-On-Chip

- Cache-coherent shared memory (hardware-controlled) – does not scale well to many cores
  - power- and area-hungry
  - signal latency across whole chip
  - not well predictable access times
- NCC-NUMA – non-cache-coherent, non-uniform memory access
  - Physically distributed on-chip [cache] memory,
  - on-chip network, connecting PEs or coherent ”tiles” of few PEs
  - global shared address space,
  - but *software* is responsible for maintaining coherence
- Examples:
  - STI Cell/B.E.,
  - Tilera TILE64,
  - Intel SCC, Kalray MPPA



Towards Many-Core CPUs...

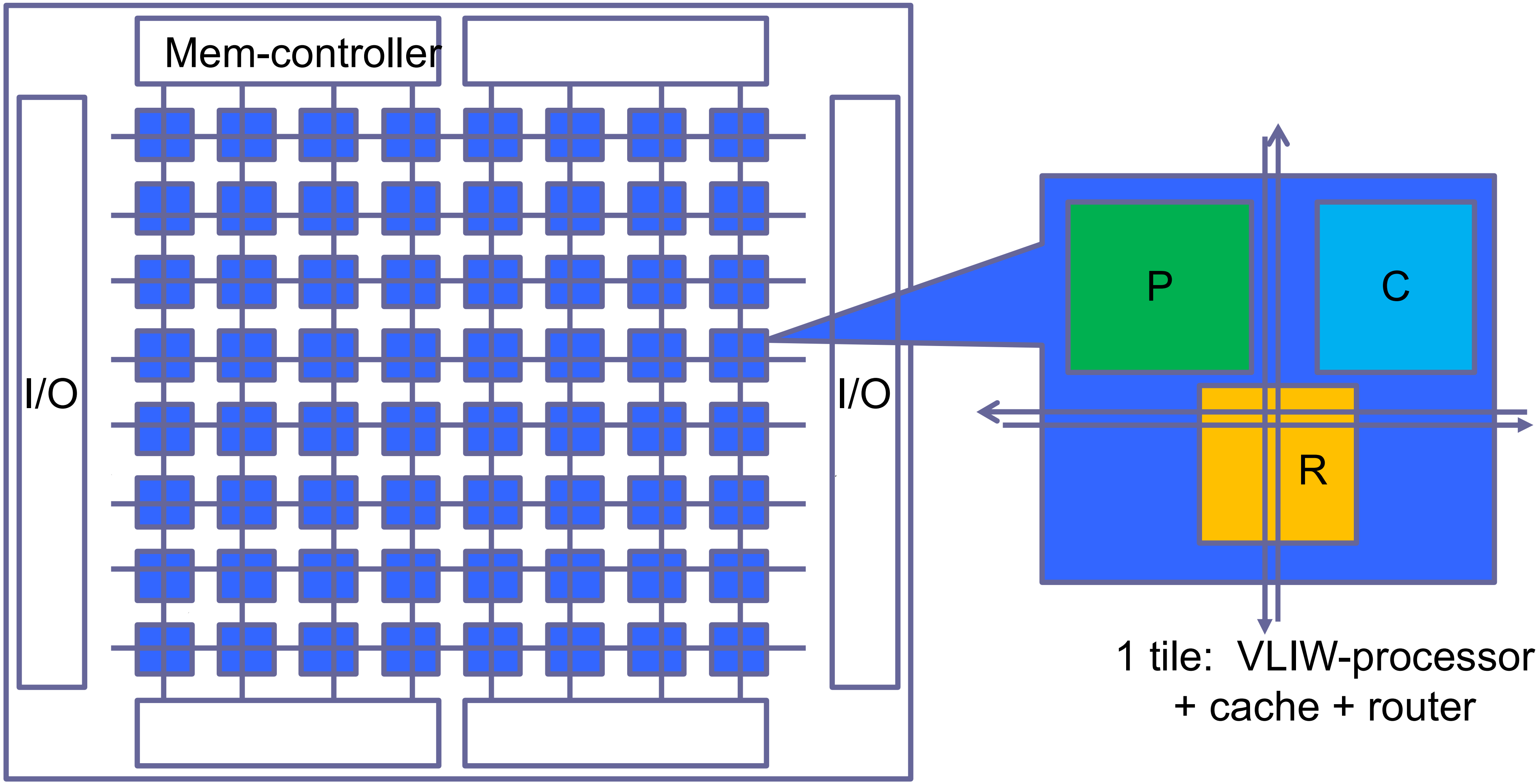
- For low-power, throughput-oriented computing
- Many (today: >100) but small (energy-efficient) CPU cores on the chip
  - No longer fully cache coherent over the entire chip
  - MPI-like message passing over 2D mesh network on chip



Source: Intel

Towards Many-Core Architectures

- Tilera TILE64 (2007): 64 cores, 8x8 2D-mesh on-chip network



1 tile: VLIW-processor
+ cache + router

(Image simplified)

Clustered Many-core CPU: Kalray MPPA-256

- 16 tiles
  with 16 VLIW compute cores each
  plus 1 control core per tile
- Message passing network on chip
- Virtually unlimited array extension by clustering several chips
- First version ca. 2012
- 28 nm CMOS technology
- Low power dissipation, typ. 5 W

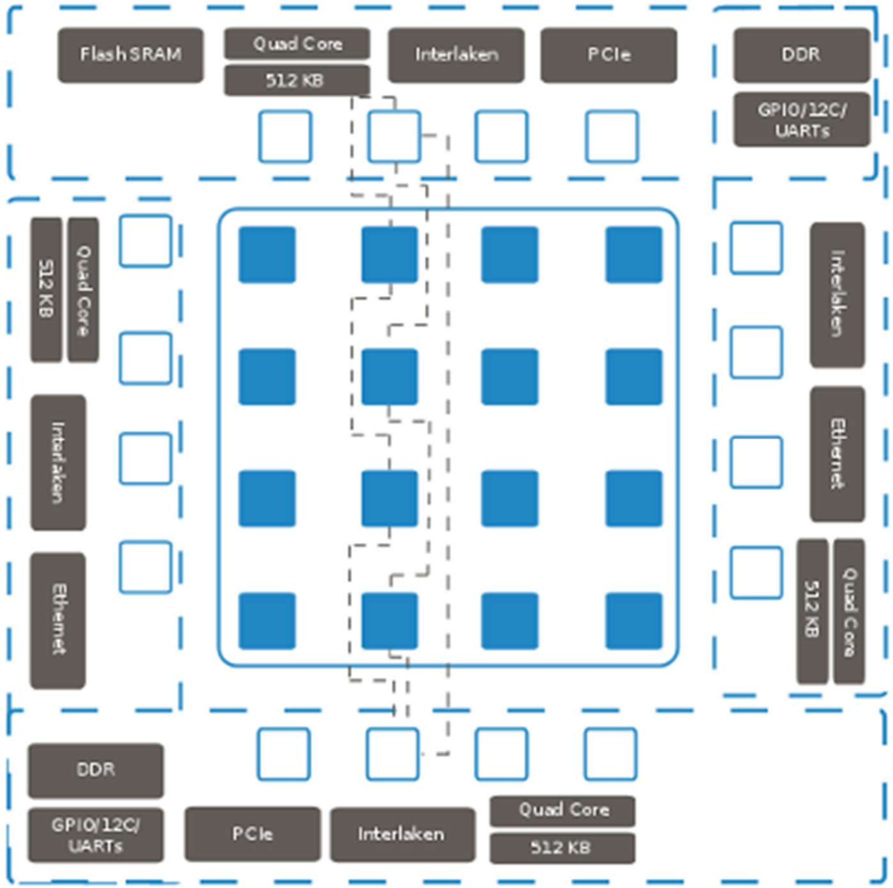
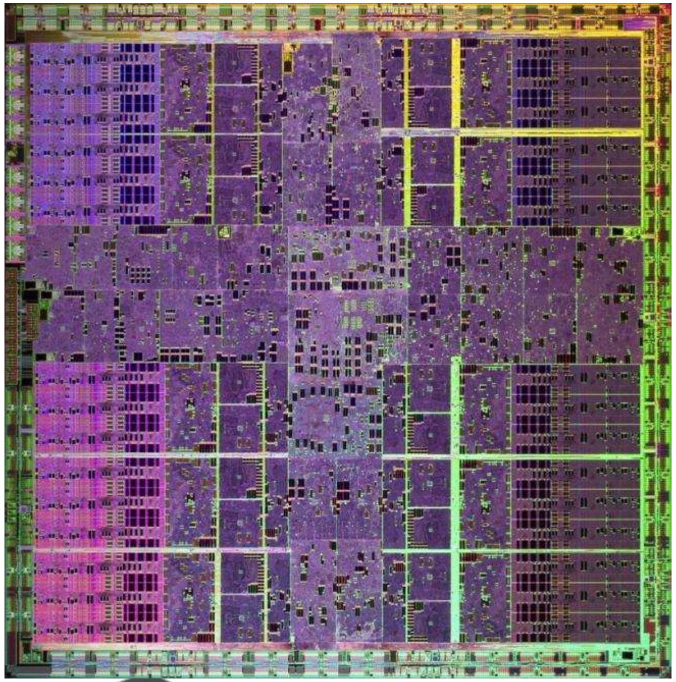


Image source: Kalray

”General-purpose” GPUs

- Optimized for high throughput rather than single-thread execution time
- **Example**:
  High-end NVIDIA GPUs (e.g. A100) have ~5000 CUDA cores
  - Each CUDA core has a
    - Floating point / integer unit
    - Logic unit
    - Move, compare unit
    - Branch unit

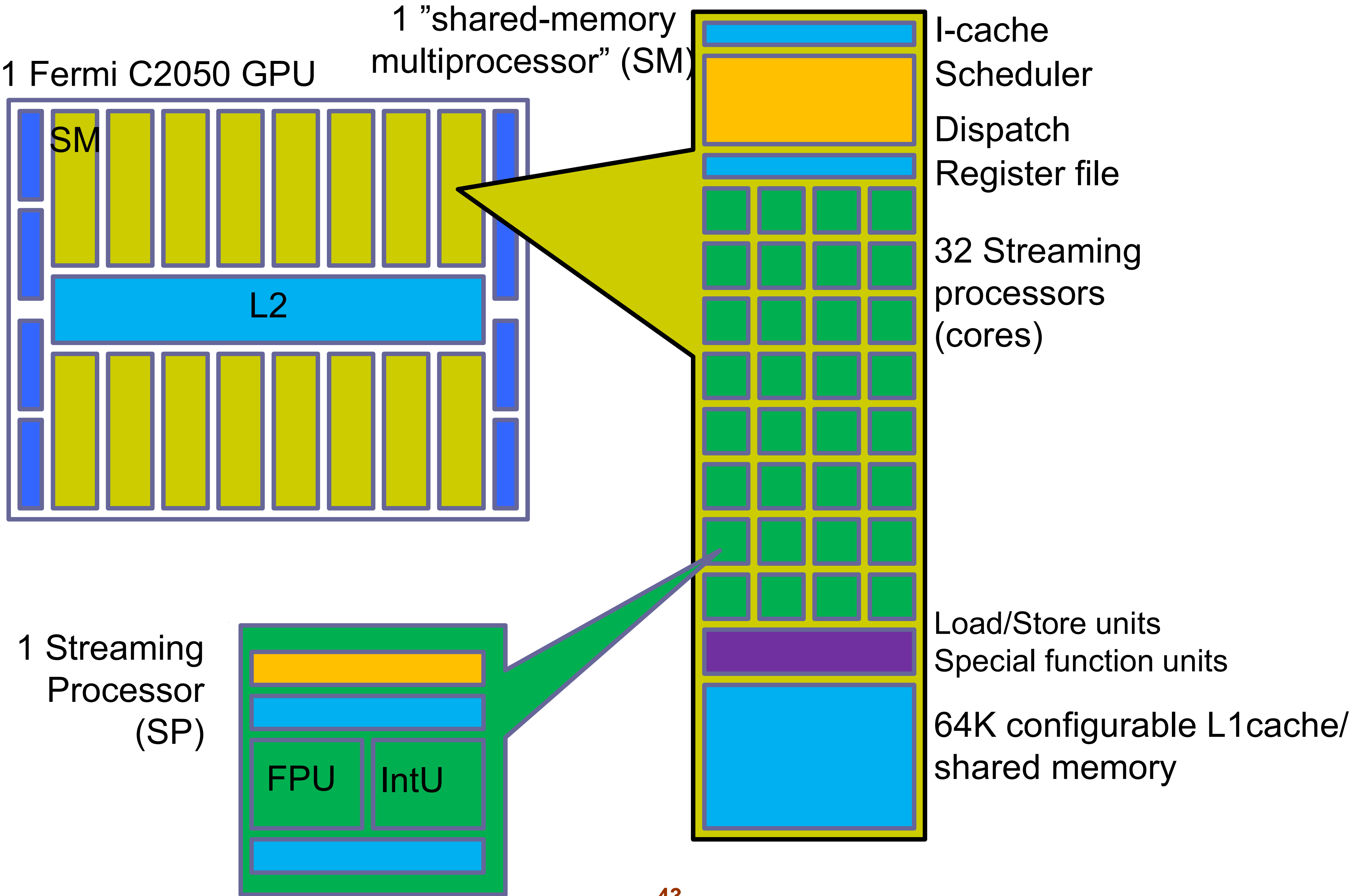
    and is highly hardware-multithreaded to hide the high memory access latency
  - Cores managed by thread manager
    - Hardware scheduler, can manage 100,000+ threads in flight
    - Zero overhead thread switching



Source: NVidia

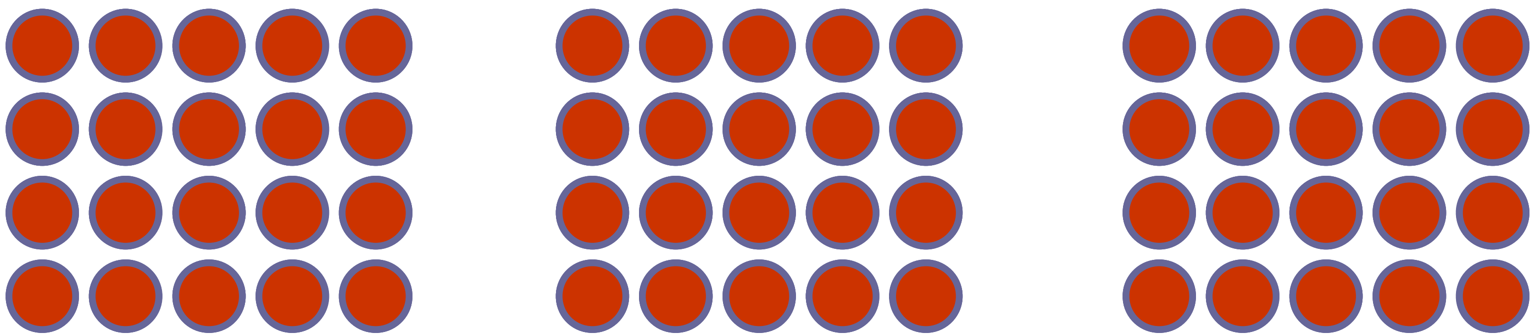
Nvidia Tesla C1060:
933 Gflops (~2009)

Nvidia Fermi (2010): 512 cores



GPU Architecture Paradigm

- Optimized for high throughput
  - In theory, ~10x to ~100x higher throughput than CPU is possible
- Massive hardware-multithreading hides memory access latency
- Massive parallelism
- GPUs are good at data-parallel computations
  - multiple threads executing the <u>same</u> instruction on different data, preferably located adjacently in memory



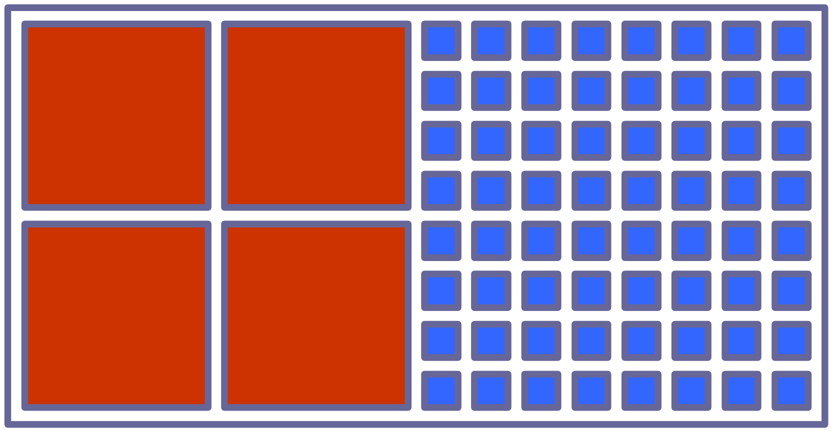
The future will be heterogeneous!

Need 2 kinds of cores – often on same chip:

- For non-parallelizable code:
  Parallelism only from running several serial applications simultaneously on different cores
  (e.g. on desktop: word processor, email, virus scanner, … not much more)

  → **Few (ca. 4-8) ”fat” cores – designed for low latency**
  (power-hungry, area-costly,
  large caches, out-of-order issue / speculation)
  for high single-thread performance

- For well-parallelizable code:
  → **hundreds of *simple* cores –
  designed for high throughput
  at low power consumption**
  (power + area efficient)



Heterogeneous / Hybrid Multi-/Manycore

Key concept: Master-worker parallelism, offloading

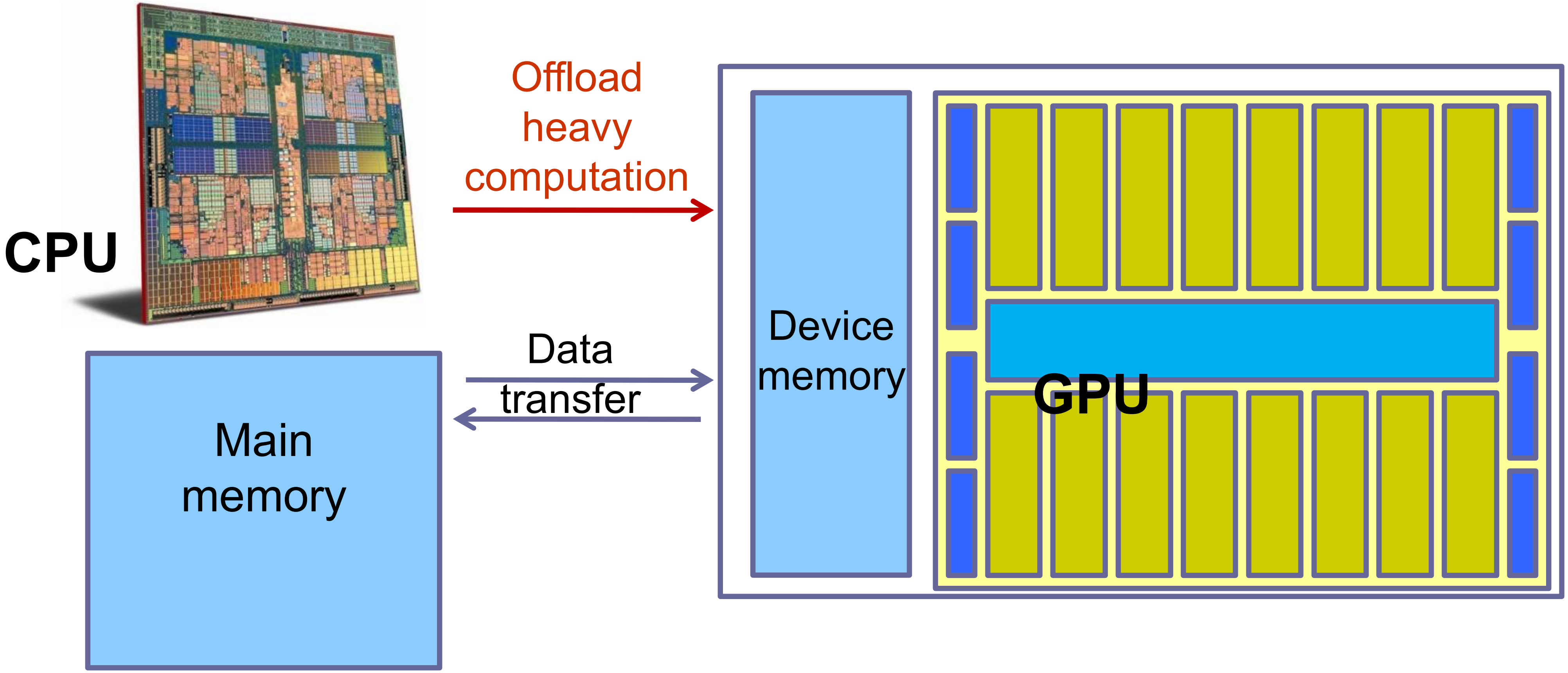
- General-purpose CPU (master) processor controls execution of worker processors by submitting tasks to them and transfering operand data to the workers' local memory

  → Master <u>offloads</u> computation to the slaves

- Workers often optimized for heavy throughput computing
  - Master could do something else while waiting for the result, or switch to a power-saving mode
- Master and worker cores might reside
  on the same chip (e.g., Cell/B.E.)
  or on different chips (e.g., systems with GPU graphics cards)
- Workers might have access to off-chip main memory (e.g., Cell) or not (e.g., most GPUs)

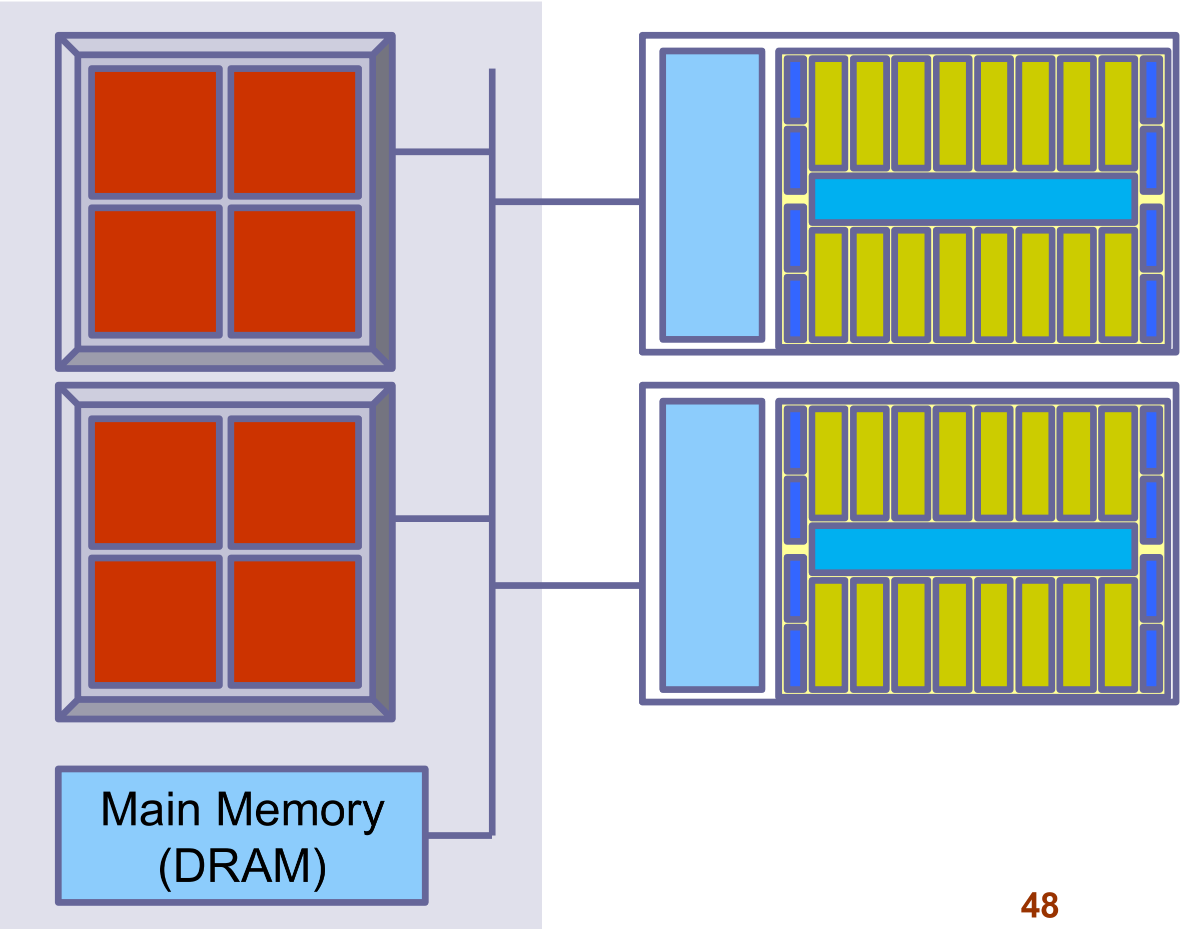
Heterogeneous / Hybrid Multi-/Manycore Systems

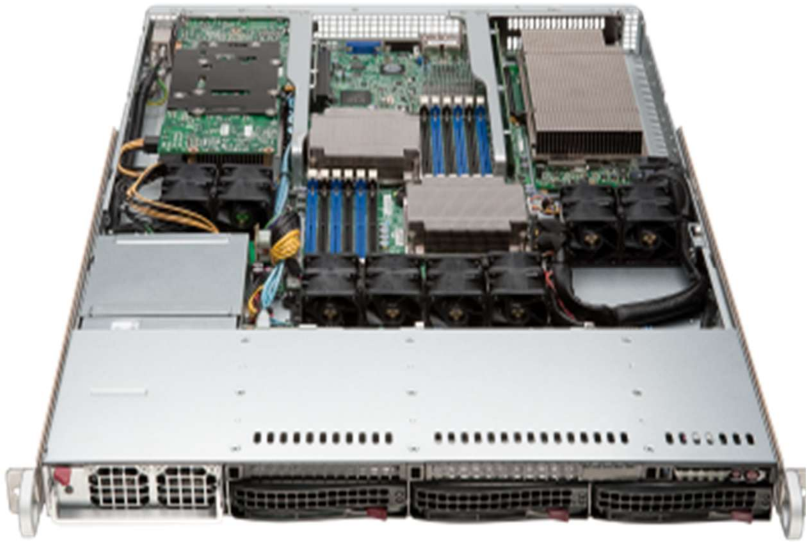
- Example: GPU-based system:



Multi-GPU Systems

- Connect one or few general-purpose (CPU) multicore processors with shared off-chip memory to several GPUs
- Increasingly popular in high-performance computing, DNN
  - Cost and (quite) energy effective if offloaded computation fits GPU architecture well



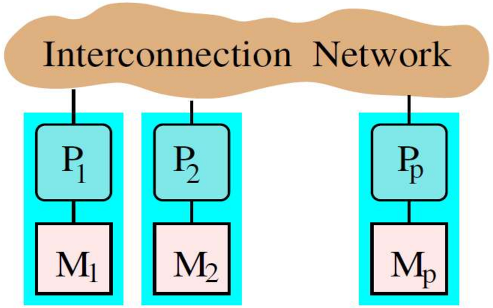


Reconfigurable Computing Units

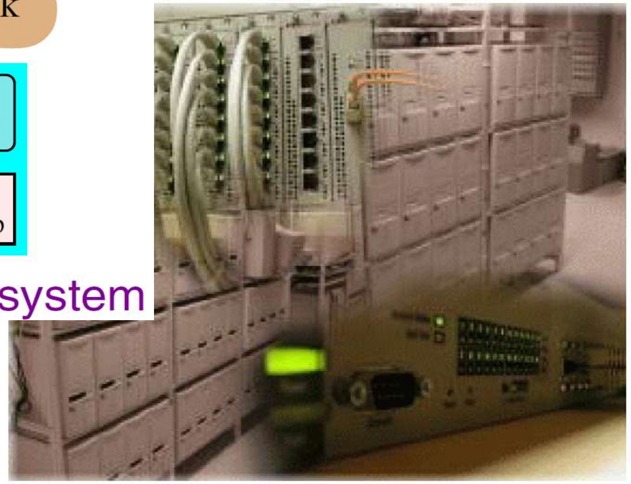
- **FPGA** – Field Programmable Gate Array

"Altera StratixIVGX FPGA" by Altera Corp.

Example: Beowulf-class PC Clusters



Distributed memory system



Characteristics:

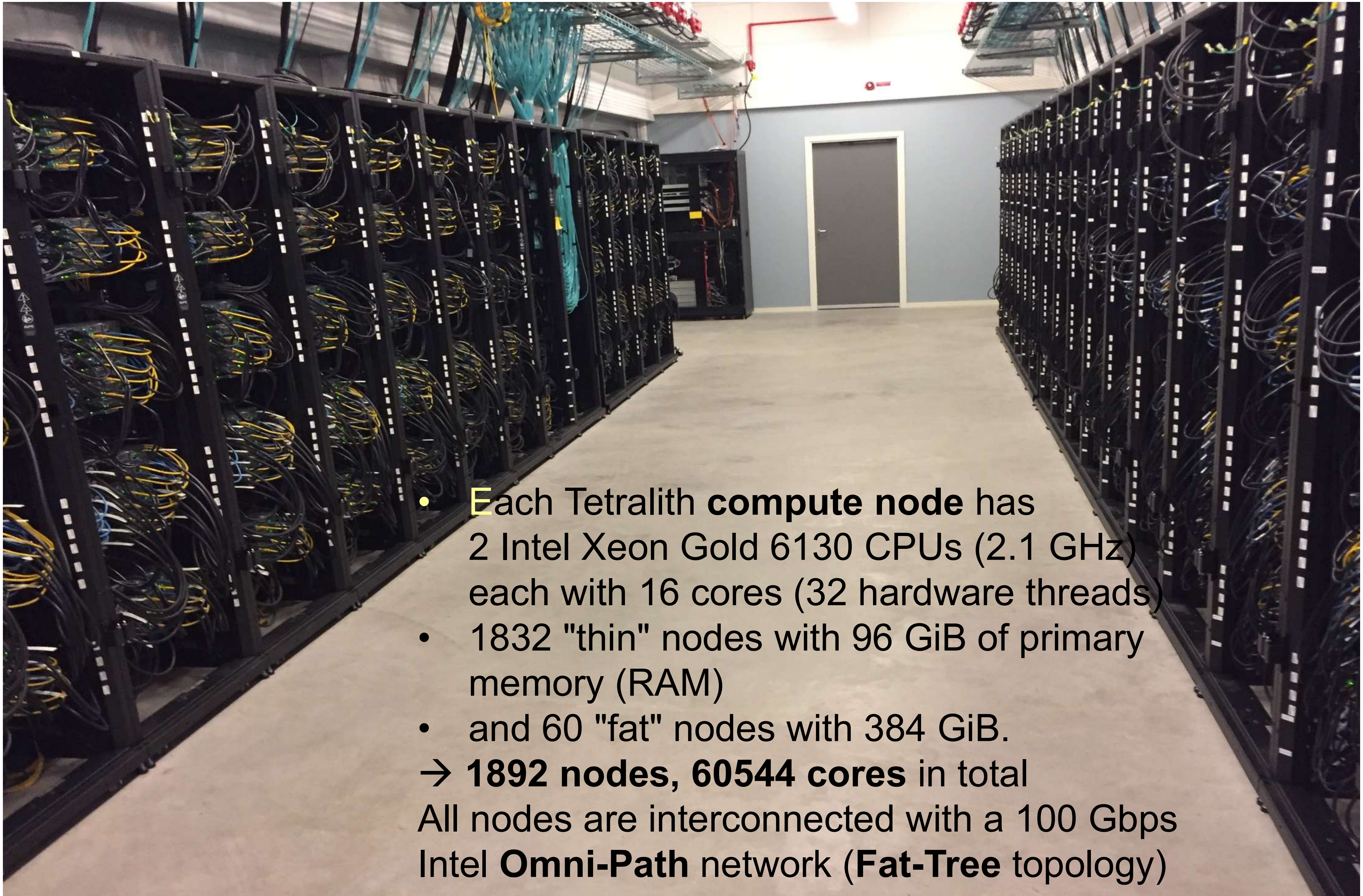
- off-the-shelf (PC) nodes
  with off-the-shelf CPUs
  (Xeon, Opteron, …)
- commodity interconnect
  G-Ethernet, Myrinet, Infiniband, SCI
- Open Source Unix
  Linux, BSD
- Message passing computing
  MPI, PVM

Advantages:

+ best price-performance ratio
+ low entry-level cost
+ vendor independent
+ scalable
+ rapid technology tracking

T. Sterling: The scientific workstation of the future may be a pile of PCs.
Communications of the ACM **39**(9), Sep. 1996

Example: Tetralith (NSC, 2018/2019)



- Each Tetralith **compute node** has 2 Intel Xeon Gold 6130 CPUs (2.1 GHz) each with 16 cores (32 hardware threads)
- 1832 "thin" nodes with 96 GiB of primary memory (RAM)
- and 60 "fat" nodes with 384 GiB.

→ **1892 nodes, 60544 cores** in total

All nodes are interconnected with a 100 Gbps Intel **Omni-Path** network (**Fat-Tree** topology)

The Challenge

- **Today, basically *all* computers are parallel computers!**
  - Single-thread performance stagnating
  - Dozens of cores and hundreds of HW threads available per server
  - May even be heterogeneous (core types, accelerators)
  - Data locality matters
  - Large clusters for HPC and Data centers, require message passing
- Utilizing more than one CPU core requires thread-level parallelism
- One of the biggest *software* challenges: **Exploiting parallelism**
  - Need LOTS of (mostly, independent) tasks to keep cores/HW threads busy and overlap waiting times (cache misses, I/O accesses)
  - All application areas, not only traditional HPC
    - General-purpose, data mining, graphics, games, embedded, DSP, …
  - Affects HW/SW system architecture, programming languages, algorithms, data structures …
  - Parallel programming is more error-prone (deadlocks, races, further sources of inefficiencies)
    - And thus more expensive and time-consuming

Can’t the compiler fix it for us?

- **Automatic parallelization?**
  - at compile time:
    - Requires static analysis – not effective for pointer-based languages
    - needs programmer hints / rewriting ...
    - ok for few benign special cases:
      - (Fortran) loop SIMDization,
      - extraction of instruction-level parallelism, …
  - at run time (e.g. speculative multithreading)
    - High overheads, not scalable

- More about parallelizing compilers in TDDD56 + TDDE65

And worse yet,

- A lot of variations/choices in hardware
  - Many will have performance implications
  - No standard parallel programming model
    - portability issue
- Understanding the hardware will make it easier to make programs get high performance
  - Performance-aware programming gets more important also for single-threaded code
  - Adaptation leads to portability issue again
- How to write future-proof parallel programs?

Python Programming is Not Suitable for Resource-Aware Computing

- Using a native programming language can give 1-2 orders of magnitude in speedup
- Exploit multiple levels of parallelism and optimizations

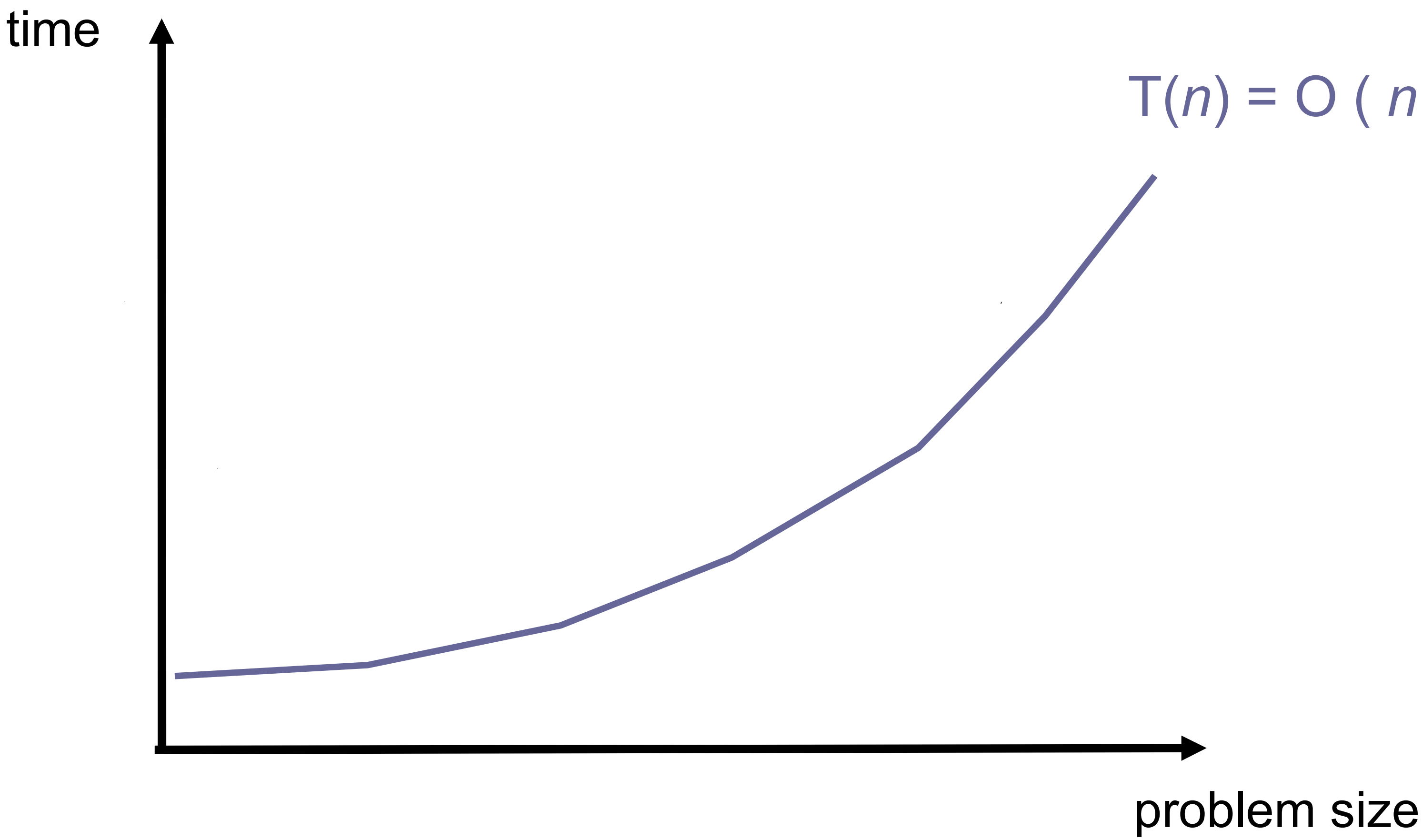
Example:
Matrix-Multiply: relative speedup to a Python version (18 core Intel Xeon CPU)

Version	Speedup	Optimization
Python	1	
C	47	Rewrite in a static, compiled (“native”) progr. language
C with parallel loops	366	Extract multi-core parallelism (OpenMP)
C with loops and memory optimization	6,727	Loop tiling for data locality
Loop vectorization using Intel AVX SIMD instructions	62,806	Extract SIMD parallelism

Table source: Turing award lecture by J. Hennessy and D. Patterson, 2018. See also:
J. Hennessy, D. Patterson: A New Golden Age for Computer Architecture. *Communications of the ACM* 62(2):48-60, Feb. 2019.

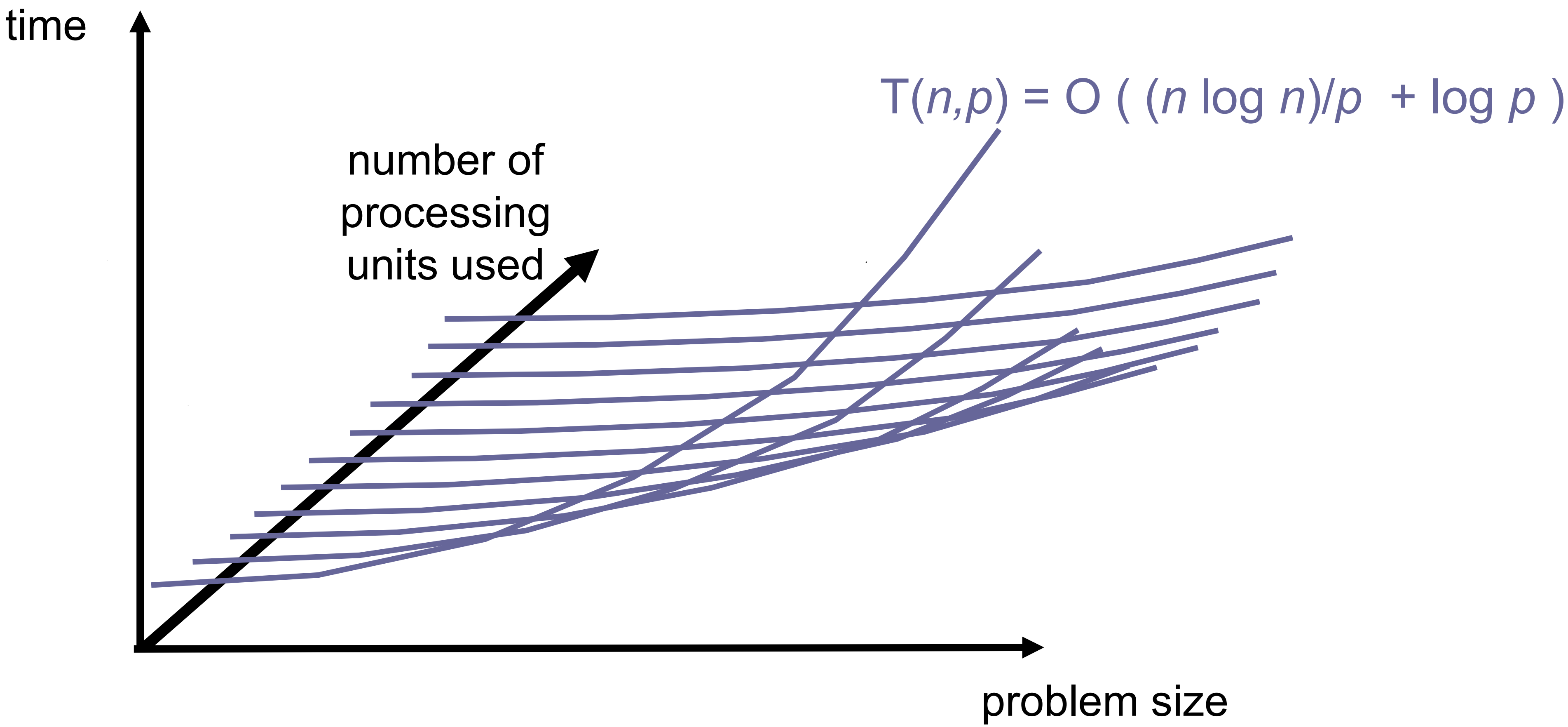
What we had learned so far …

- Sequential **von-Neumann model**
  programming, algorithms, data structures, complexity
  - Sequential / few-threaded languages: C/C++, Java, Python, ...
    not designed for exploiting massive parallelism



… and what we need now

- **Parallel programming!**
  - Parallel algorithms and data structures
  - Analysis / cost model: parallel time, work, cost; scalability;
  - Performance-awareness: data locality, load balancing, communication



Questions?

Homework

- Explain the difference between software multithreading and hardware multithreading.
- Explain the difference between hardware multithreading and multicore.
- For your own computer / smartphone, find out which CPU it has, with how many cores and hardware threads.