High-Performance Computing

– Introduction –

Christian Plessl

High-Performance IT Systems Group

Paderborn University

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

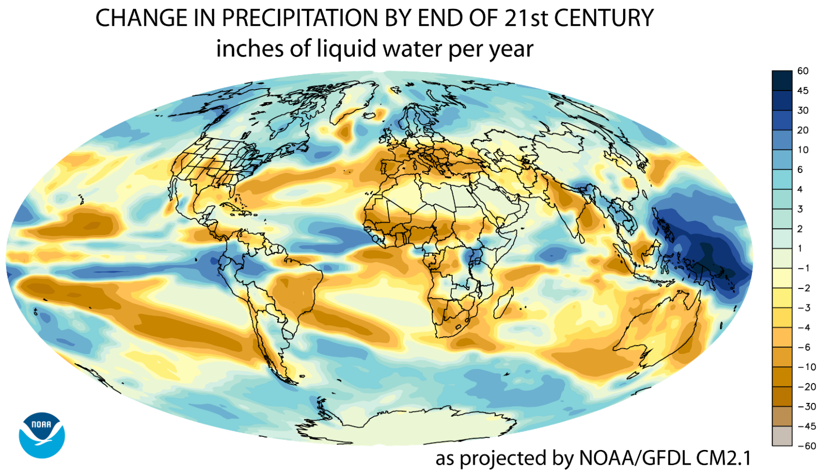
version 1.1.2 2017-10-16

Paderborn Center for Parallel Computing

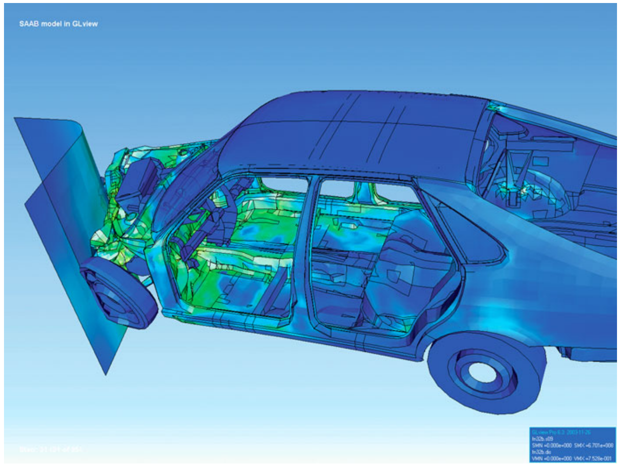
Why Do We Need High-Performance Computing

- **Simulation**
  - computational models that can make accurate quantitative predictions from first order principles in many areas of natural sciences
  - challenge: study interaction of different effects (multi-physics); desire for higher accuracy and higher temporal and spatial resolution; study dynamics behavior instead of steady state
- **Optimization**
  - identify optimal system structures by repeated simulation with varying parameters
  - challenge: same as simulation, but many iterations required
- **Analytics**
  - identify and extract interrelations in data-sets and abstract them with models
  - challenge: trend to ever larger data-sets and unstructured data

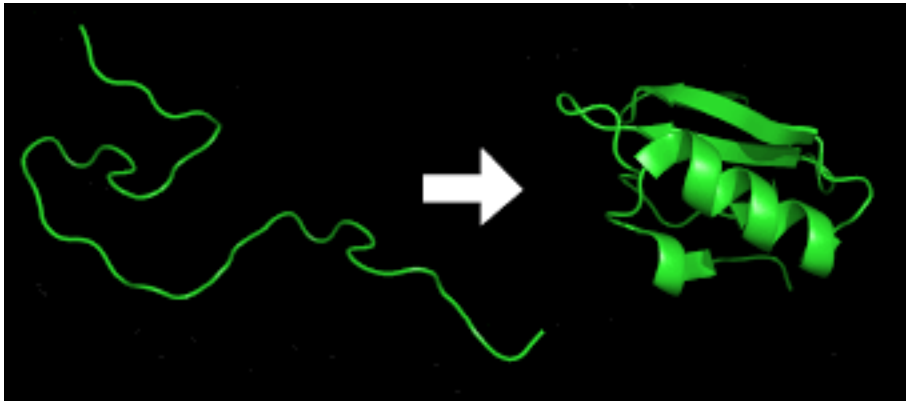
Simulation



climate prediction

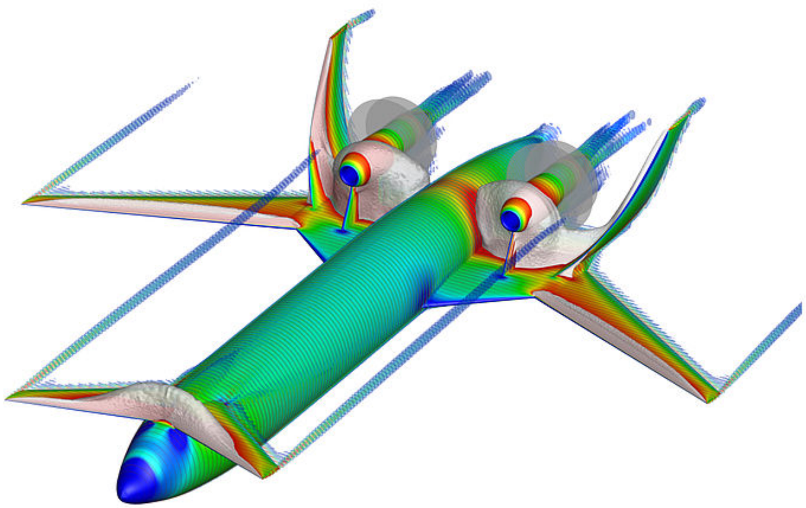


mechanical structure simulation

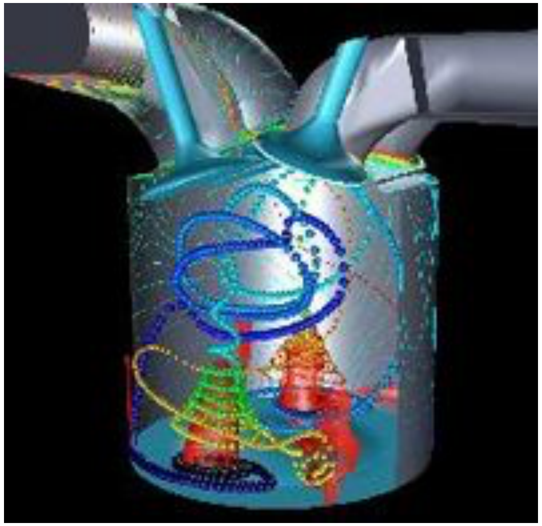


protein folding

Optimization



airflow optimization

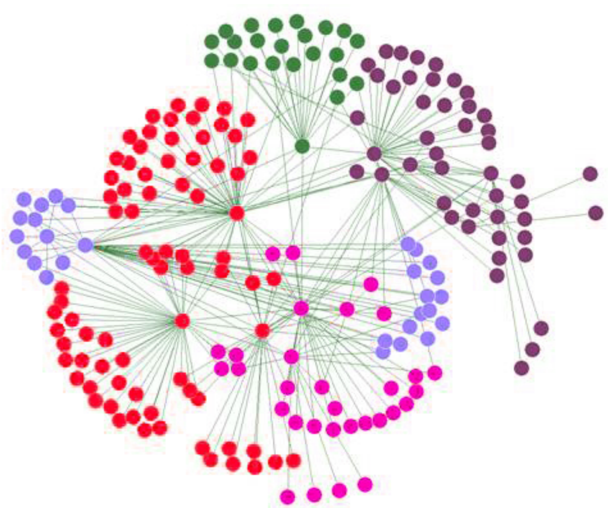


fuel combustion optimization

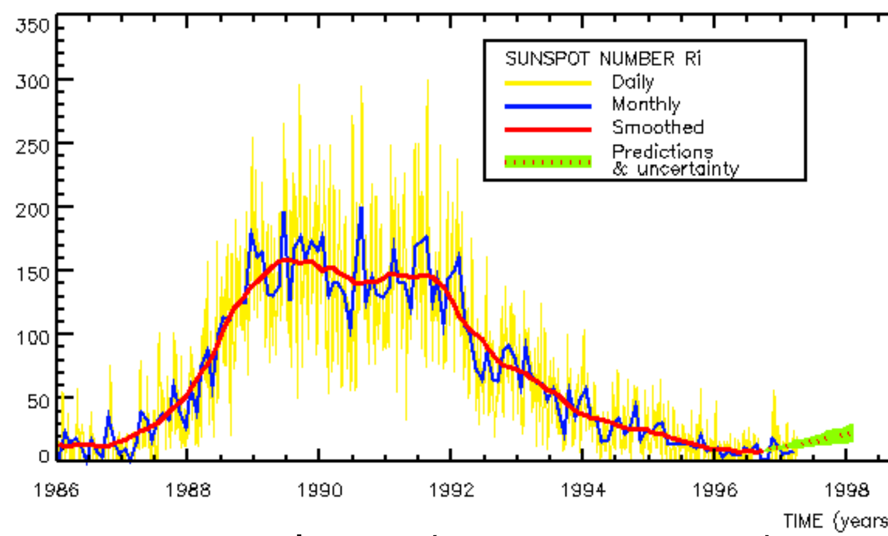
evolved antenna
(designed with
evolutionary algorithm)

Analytics

DNA sequence analysis (bio informatics)



social networks



astrophysics (sun spot activity)

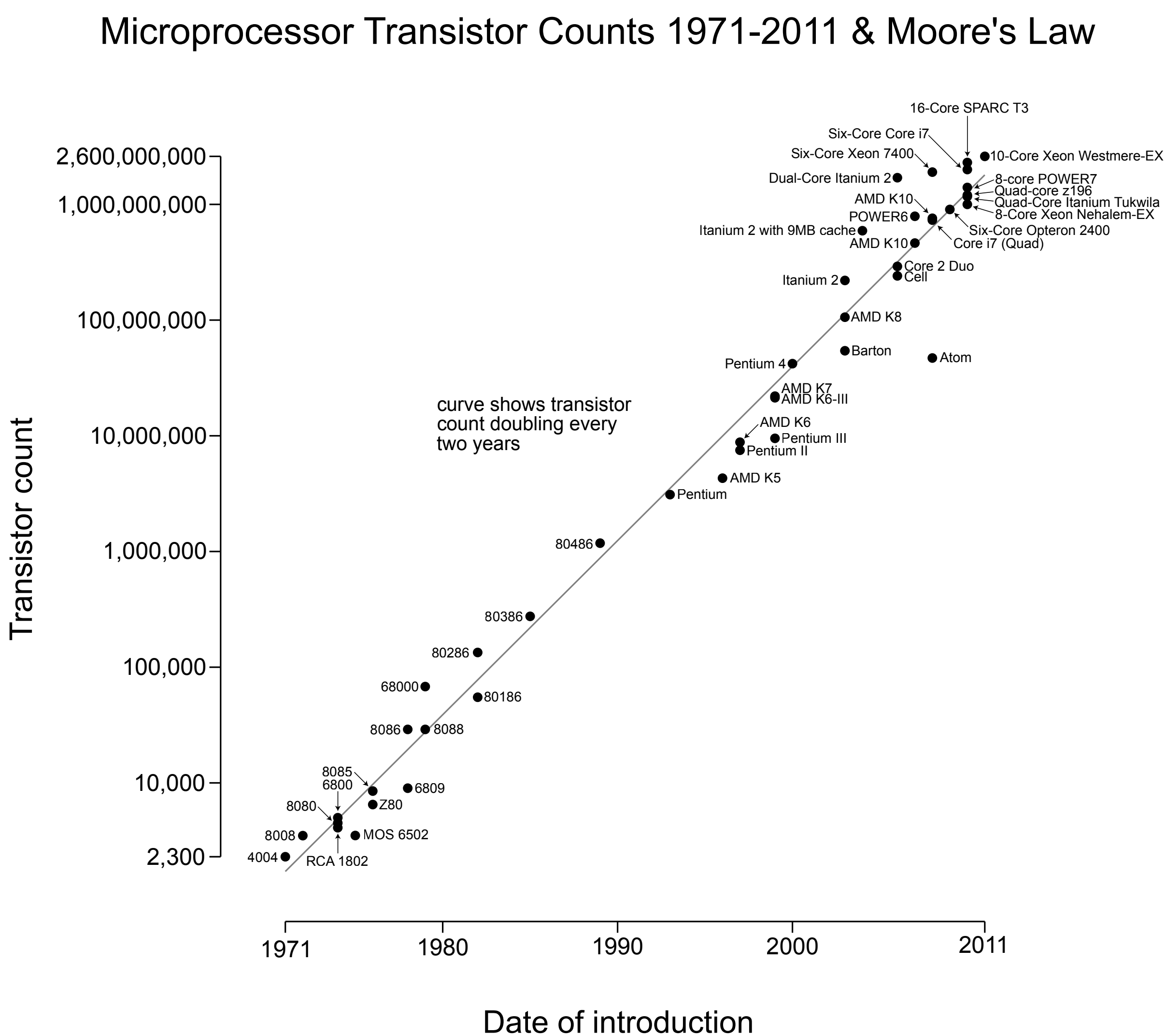
Drivers for Ever-increasing Performance

- **Engineering**
  - acceleration of time-to-market with virtual prototyping
  - optimization of design parameters to improve functionality or lower cost

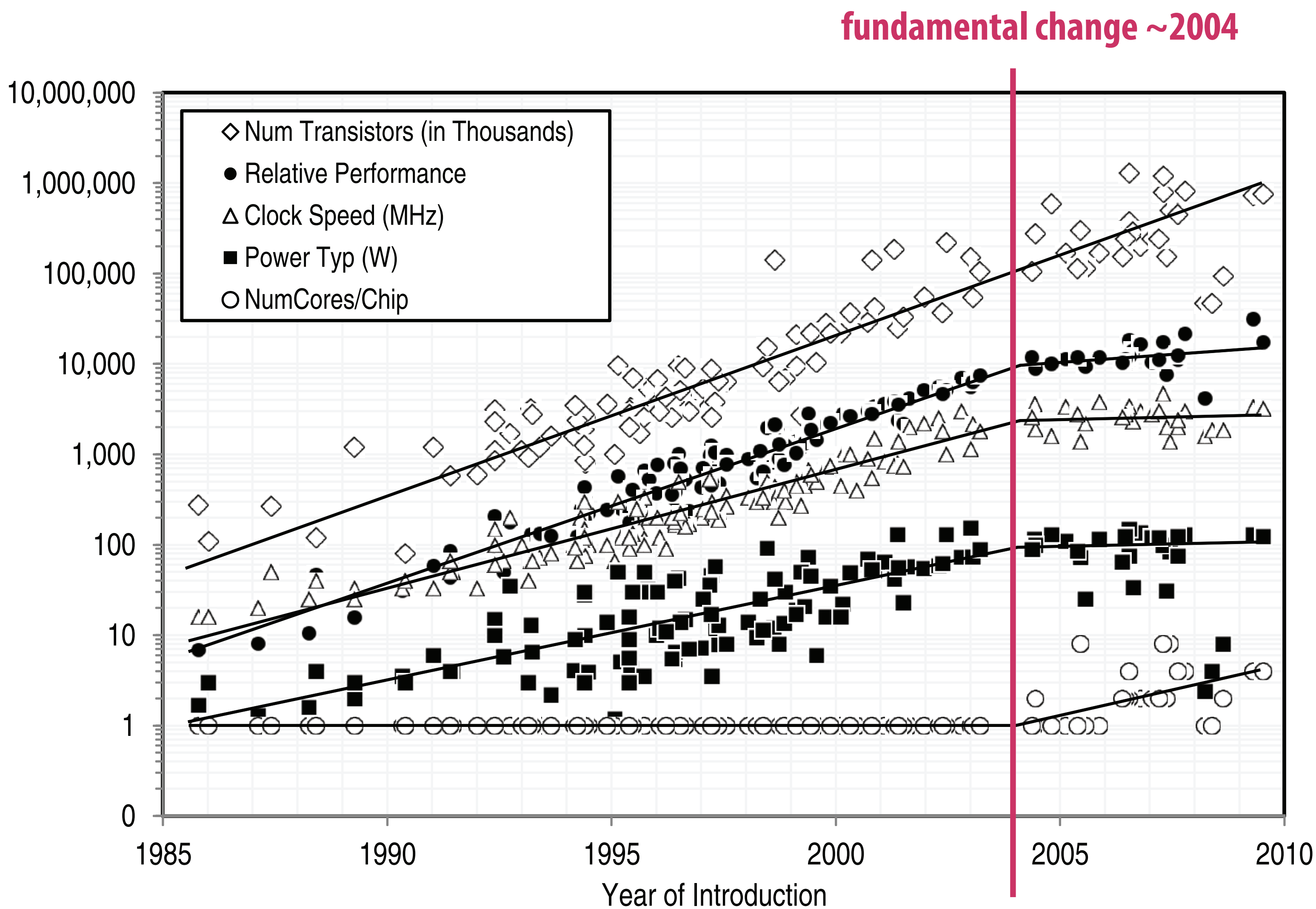
- **Science**
  - replacement of experiment and pen-and-paper theory with simulation
  - crossing of disciplinary boundaries

- **Data-driven research**
  - new sensors create unprecedented amount of multi-modal data
  - desire to extract knowledge from previously unused data

Moore's Law Blessed us with Transistors . . .



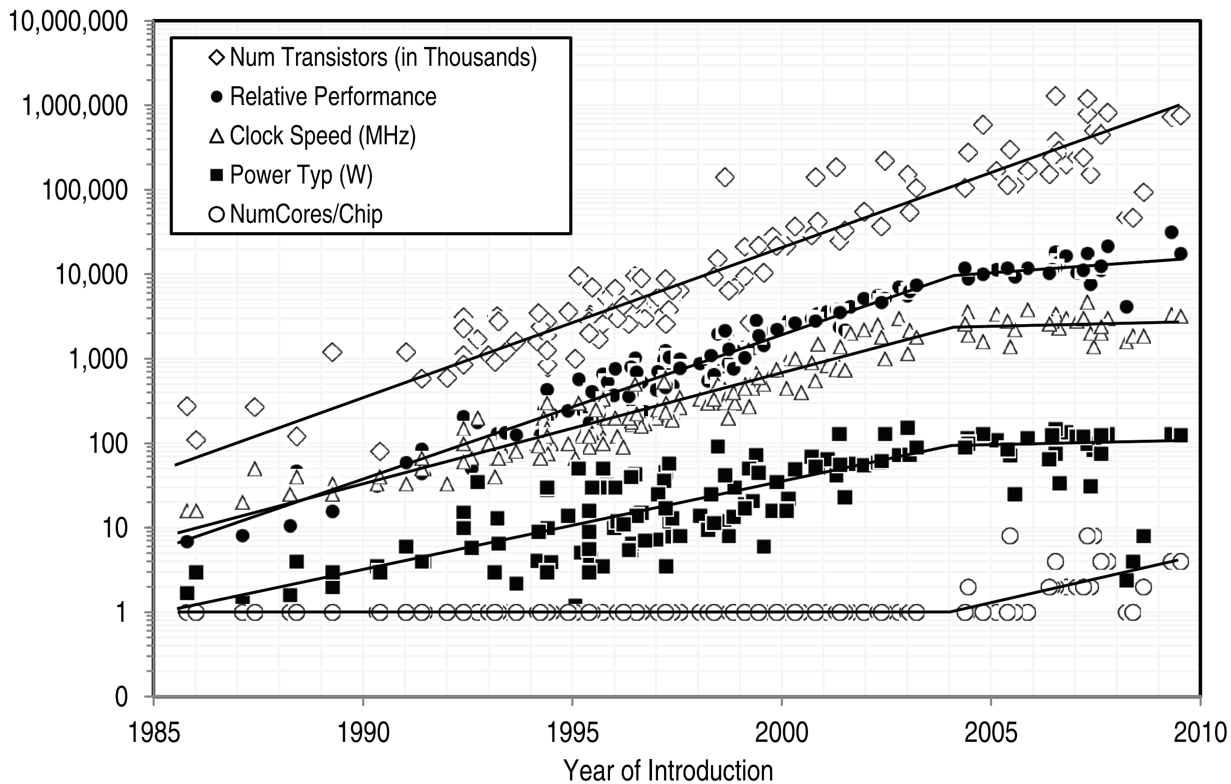
. . . and Dennard's Scaling Law with Efficiency



source: US National Academy of Sciences: Future of Computing Performance (2011)

Times Have Changed

- **Performance gains per CPU core have slowed down dramatically ~20% increase/yr instead of ~50%/year**
- **Number of CPU cores is now growing exponentially**
- **Serial programs do not benefit from these advances (in most cases)**



Parallel Computing is a Necessity

- **Energy-efficiency**
  - many but more efficient processor cores in a CPU

- **Performance by parallel execution on all levels in a server …**
  - data-level (vector instructions)
  - thread-level (simultaneous multi threading)
  - chip-level (multi-cores)
  - server-level (symmetric multi-processing)
- **… and in the compute center**
  - cluster-level (shared/distributed memory systems with fast interconnect)

Approaches to Make a Serial Program Parallel

- **Runtime environment takes care of parallelism**
  - e.g. Matlab parallel computing toolbox, NumPy
- **Auto-parallelizing compilers**
  - e.g. GNU compiler, Intel ICC
- **Optimized, parallel libraries**
  - e.g. Intel Math Kernel Library (MKL), NAG parallel libraries, Tensorflow
- **Languages extensions and APIs for serial programming languages**
  - Pthreads, OpenMP, MPI, OpenCL, C++ parallel STL
- **Parallel programming languages**
  - Chapel, Julia, Go

Automatic Parallelization

- **Compilers for automatic conversion of serial programs to parallel programs**
  - studied for decades by so far limited success
  - resulting programs frequently inefficient
- **In many cases the best parallel solution <u>does not</u> correspond to a parallelized version of the best serial code**
  - requires to step back and devise an entirely new algorithm

Example

- **Objective: Compute n values and add them together**
- **Serial solution:**

```
sum = 0;
for (i=0;  i<n;  i++) {
  x = compute_next_value(…);
  sum += x;
}
```

Example (cont.)

- **We have p cores, $p \ll n$**
- **Each core performs a partial sum of approximately n/p values**

```
my_sum = 0;
my_first_i = …;
my_last_i = …;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
  my_x = compute_next_value(…);
  my_sum += my_x;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example (cont.)

- **After each core completes execution of the code, its private variable my_sum contains the sum of the values computed by its calls to compute_next_value**

- **Example: 8 cores, n = 24 and the calls to compute_next_value return:**

1,4,3,   9,2,8,   5,1,1,   5,2,7,   2,5,0,   4,1,8,   6,5,1,   2,3,9

Example (cont.)

- **Once all the cores are done computing their sum my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.**

```
if (I'm the master core) {
  sum = my_x;
  for each core other than myself {
    receive value from core;
    sum += value;
  }
} else {
    send my_x to the master;
}
```

Example (cont.)

Core	0	1	2	3	4	5	6	7
numbers	1,4,3	9,2,8	5,1,1	5,2,7	2,5,0	4,1,8	6,5,1	2,3,9
my_sum	8	19	7	15	7	13	12	14

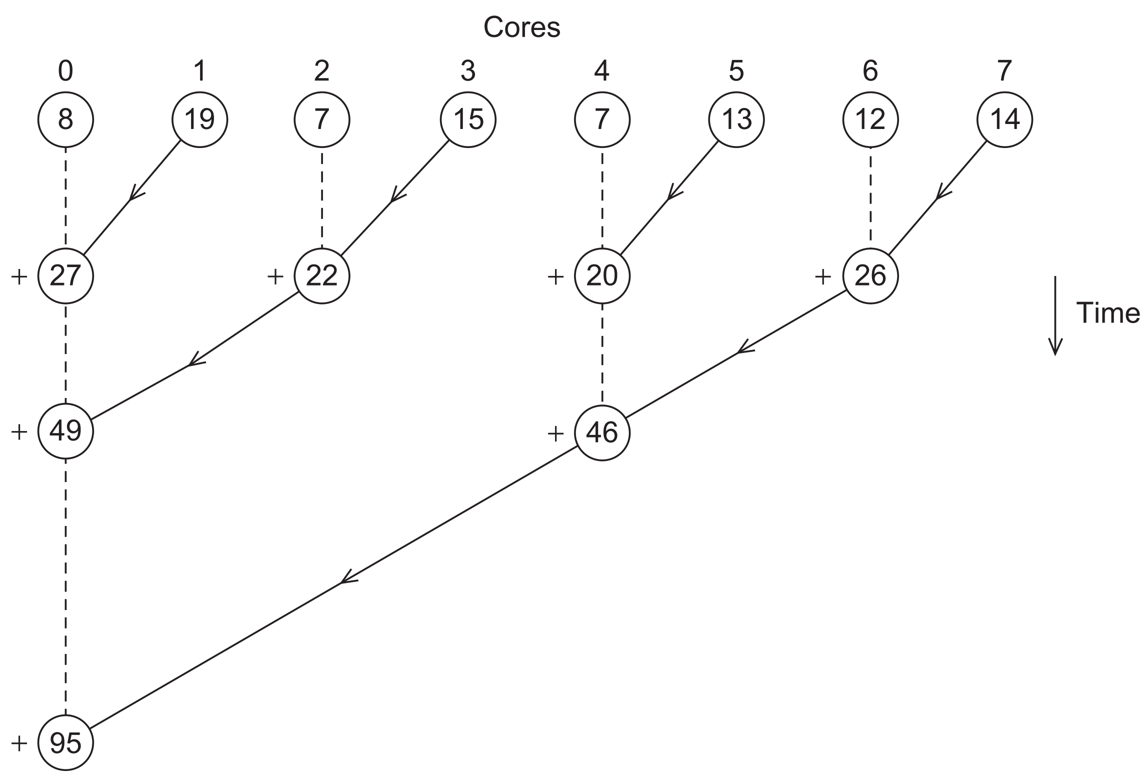
Global sum

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Code is Correct but Inefficient

- **Computation of parallel sum in master core, two problems**
  - master is loaded with all the computation time for parallel sum (serial code)
  - master aggregates all the communication latencies (serial code)
- **Share work for parallel sum among all cores**
  - use summation tree



Analysis

- **Number of operations**
  - in the naïve implementation the master core performs 7 receives and 7 additions
  - in the summation tree the master performs 3 receives and 3 additions
  - **the improvement is more than a factor of 2**

- **The difference is more dramatic with a larger number of cores, for 1000 cores the master**
  - performs 999 receives and 999 additions in the naïve implementation
  - but only 10 receives and 10 additions in the summation tree
  - **that's an improvement of almost a factor of 100!**

- **When communication over network $t_{communication} \gg t_{addition}$ the communication latency may be the main performance limitation**

Task vs. Data Parallelism

- **Task parallelism**
  - Partition the problem into a sequence of tasks (that perform a different function and possibly depend on each other)
  - Distribute the tasks among the cores

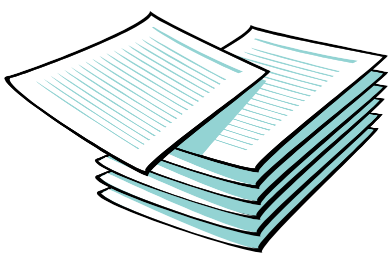
- **Data parallelism**
  - Partition the data that needs to be processed to solve the problem between the cores
  - Let each core perform the same or very similar operations on its part of the data

- **Frequently both approaches are combined**
  - Not an either-or decision
  - Application typically work in phases (tasks), which are data parallel themselves (nested parallelism)

Example: Grading of exams

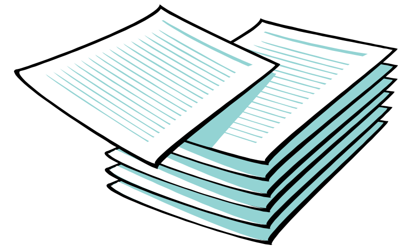
15 questions

300 exams



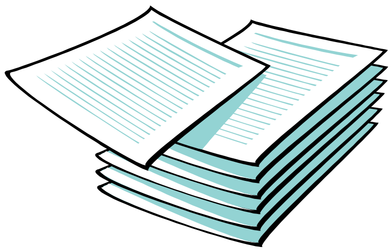
Division of Work – Data Parallelism

teaching assistant 2

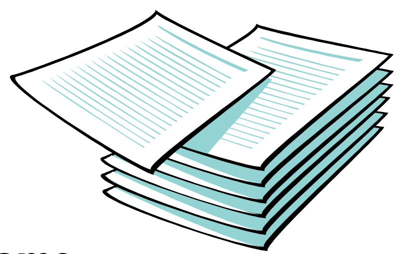


100 exams

teaching assistant 1



100 exams



100 exams

teaching assistant 3

Division of Work – Task Parallelism

teaching assistant 1

Questions 1–5

teaching assistant 2

Questions 11–15

teaching assistant 3

Questions 6–10

Division of Work – Data parallelism

```
sum = 0;
for (i=0;  i<n;  i++) {
  x = compute_next_value(…);
  sum += x;
}
```

Division of Work – Task Parallelism

```
if (I’ m the master core) {
  sum = my_x;
  for each core other than myself {
    receive value from core;
    sum += value;
  }
} else {
    send my_x to the master;
}
```

Tasks

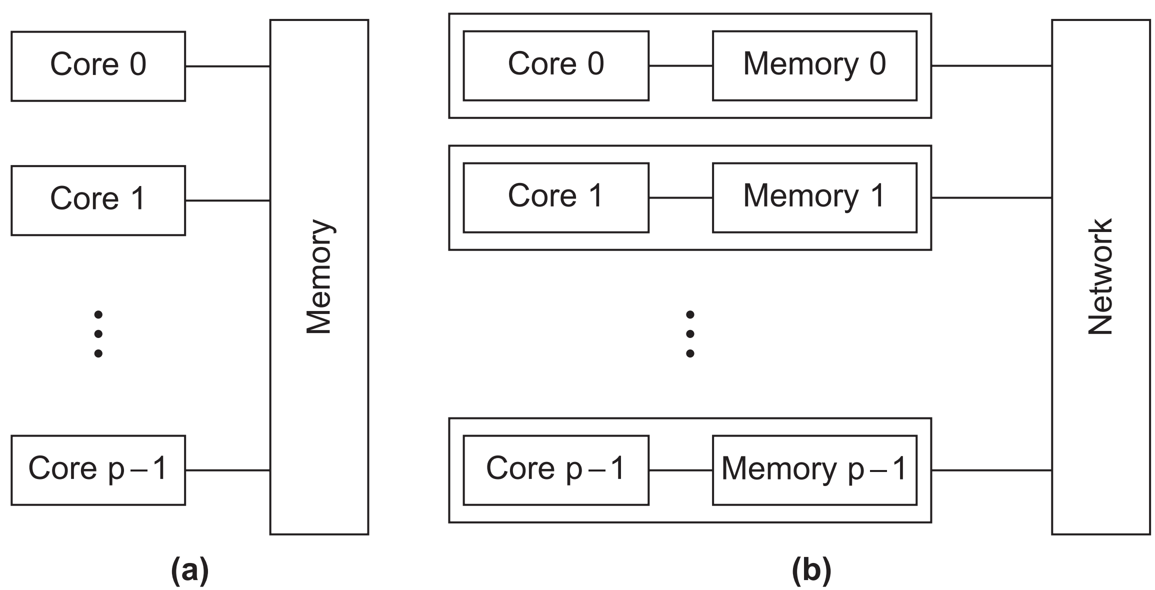
1) **Receiving**
2) **Addition**

Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core
- Load balancing – share the work evenly among the cores so that one is not heavily loaded
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest

Type of Parallel Systems

- **Shared-memory**
  - All cores shares access to the computer's memory
  - The cores coordinate and communicate by examining and update shared memory locations
- **Distributed-memory**
  - Each core has its own, private memory
  - The cores must communicate explicitly by sending messages across a network



Shared-memory **Distributed-memory**

Concurrent vs. Parallel vs. Distributed

- **Concurrent computing** – a program is one in which multiple tasks can be in progress at any instant.
- **Parallel computing** – a program is one in which multiple tasks cooperate closely to solve a problem
- **Distributed computing** – a program may need to cooperate with other programs to solve a problem.

Summary

- The laws of physics and limitations of semiconductor technology have brought us to the doorstep of multicore technology
- Serial programs typically don't benefit from multiple cores
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers
- Learning to write parallel programs involves learning how to coordinate the cores
- Parallel programs are usually very complex and therefore, require sound program techniques and development

Acknowledgements

- **Peter S. Pacheco / Elsevier**
    - for providing the lecture slides on which this presentation is based

Change log

- **1.1.3 (2017-10-17)**
  - consistent capitalization of titles
  - clarifies text on slide 20
- **1.1.2 (2017-10-16)**
  - fix typo on slide 18
  - clarify text on slides 20, 27
- **1.1.1 (2017-10-10)**
  - cosmetics
- **1.1.0 (2017-10-08)**
  - updated for winter term 2017/18
- **1.0.1 (2016-10-28)**
  - fix typo on slide 14; add page numbers; cosmetics
- **1.0.0 (2016-10-26)**
  - initial version of slides