Lecture 3: Complexity & Efficiency

Seminar ‘Foundations of Data Science’

Prof. Dr. Karsten Donnay, Assistant: Marcel Blum

Course Outline

– Part 1: Foundations

- *Day 1: Mon. 14.06.2021*: Information Coding & Data
- *Day 2: Tue. 15.06.2021*: Programming & Algorithms
- ***Day 3: Wed. 16.06.2021*: Complexity & Efficiency**

– Part 2: Applications

- *Day 4: Thu. 17.06.2021*: Data Collection & Quality
- *Day 5: Fri. 18.06.2021*: Research on Digital Media

Overview of this Session

- Complexity
  - Basic Concepts
  - Runtime Analysis
  - O-Notation
- Efficiency
  - Parallelization
  - Hardware Parallelization
  - Software Parallelization

University of Zurich[UZH]

Department of Political Science

Complexity

Complexity

- What exactly do we mean by complexity?
  - Algorithmic complexity
    - Typically, refers to time complexity of a given algorithm
    - Depends on a particular implementation
  - Memory complexity
    - Required memory during calculation
    - Depends also on the a particular implementation
  - Problem complexity
    - Minimal effort needed to solve a problem
    - Analyzed for typical cases
    - Classification is independent of the specific algorithm

Complexity

- Why do we care about complexity?
  - Problem complexity decides whether we can in principle solve a problem in finite time
  - Algorithmic and memory complexity determine whether solutions are feasible to implement
    - Example: run time estimates (algorithmic complexity)
      - Home computer executes 10^8 operations per second
      - Supercomputer executes 10^{12} operations per second

naïve duplicate check (n^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

mergesort (n log n)

computer	thousand	million	billion
home	instant	1 sec	18 min
super	instant	instant	instant

Complexity

- Key conceptual insights
  - Memory complexity and time complexity could be very different for different algorithms
    - May favor one over the other depending on the particular application
    - Decision between memory constraints and time constraints is practically often relevant
  - Time complexity varies in how it changes with the number of elements
    - e.g., slowly rising, then sharp increase vs. sharp increase then slowly rising
    - Makes certain algorithms more favorable for small others for large number of elements
  - Depending on data, complexity may vary; distinguish between
    - Worst case complexity
    - Average case complexity
    - Best case complexity

Runtime Analysis

- The runtime sums up time units needed per instruction
  - Primitive operations
    - Assignments, comparisons, arithmetic operations, …
    - 1 time unit
  - Loops
    - (number of loops) · (time units per loop)
  - Method calls
    - Time units for method
  - Recursive calls
    - Sum of calls

Runtime Analysis

– Example: naïve duplicate check for array values

Algorithm in R

```
duplicates = function(array){
   for (i in 1:length(array){ 1
      for (j in 1:length(array){ 2
         if (i!=j & array[i]==array[j]){ 3
            return(TRUE)
         }
      }
   }
   return(FALSE)
}
```

1. get length = 1 time unit
2. get length = 1 time unit
3. two logical comparisons = 2 time units

n: length of array

Runtime Analysis

– Example: naïve duplicate check for array values

Algorithm in R

```
duplicates = function(array){
   for (i in 1:length(array){          1
      for (j in 1:length(array){                2
         if (i!=j & array[i]==array[j]){
            return(TRUE)
         }
      }
   }
   return(FALSE)
}
```

1 loop with maximal n iterations

2 loop with maximal n iterations

n: length of array

Runtime Analysis

– Example: naïve duplicate check for array values

1. get length = 1 time unit
2. get length = 1 time unit
3. two logical operations = 2 time units

1. loop with maximal n iterations
2. loop with maximal n iterations

$$n \cdot \big(1 + n \cdot (1 + 2)\big) = n + 3n^2$$

- Dominant term is n^2
- Constant terms and lower terms asymptotically influence the runtime only marginally

Runtime Analysis

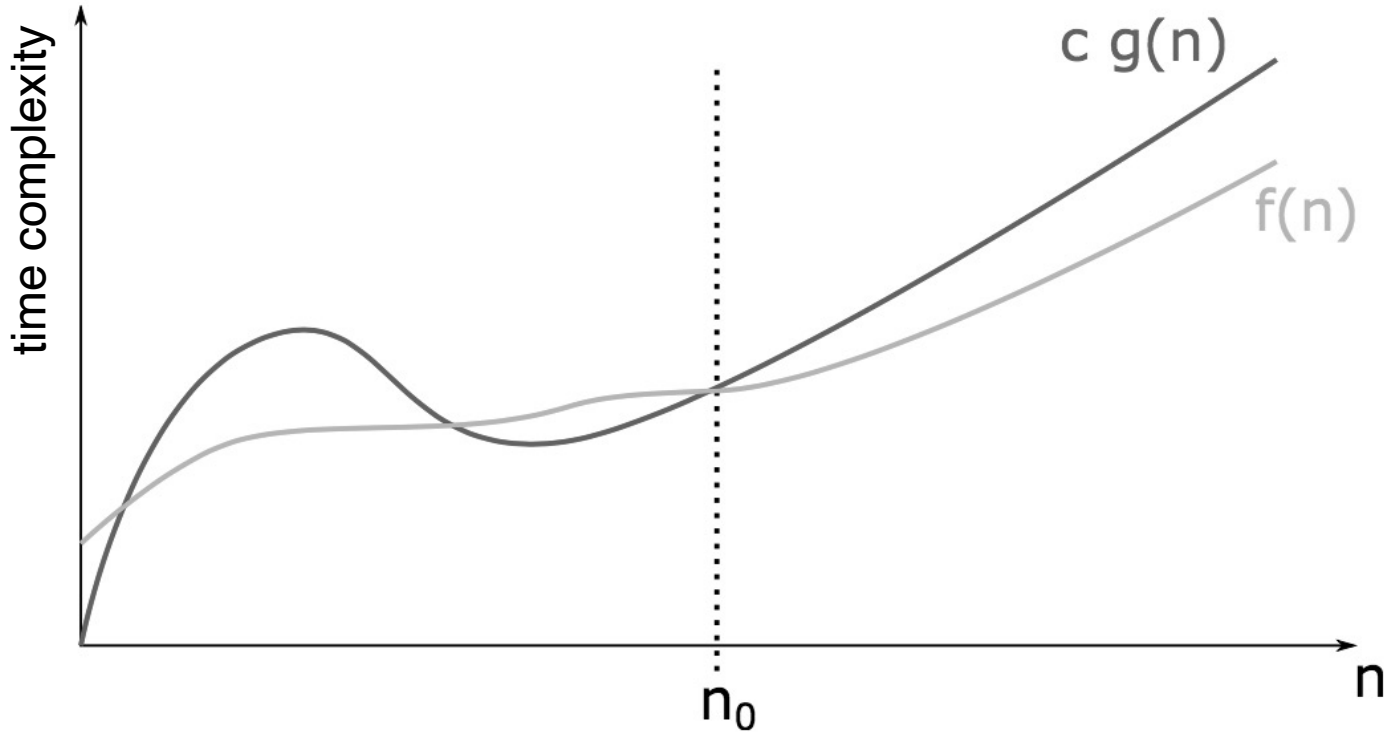
- Example: naïve duplicate check for array values
  - What we looked at here is actually **worst case complexity**
    - The calculation assumed that we have to look through the entire array
    - But this is only the case if the duplicate entry is the last entry of the array
  - The algorithm is likely much faster for other cases
    - e.g., the duplicate entry is in the first index of the array, search finishes already after one(!) iteration, i.e., that is the **best case complexity**
    - Probably here more relevant for performance to look at **average case complexity**
  - Approach these intuitions more formally with O-Notation next…

O-Notation

- Definition
  - O-Notation describes classes of functions with specific (asymptotic) properties
  - Also referred to as Big O notation (or Landau's symbol)
  - Values of a given function are after a given point n_0
    - Always larger or equal, or
    - Always smaller or equal to a given function
  - Constants are not considered or expressed by an (arbitrary) chosen pre-factor
  - Separates into three different notations where O is the most commonly used
    - O refers to the upper bound
    - Ω to the lower bound
    - Θ to the exact bound

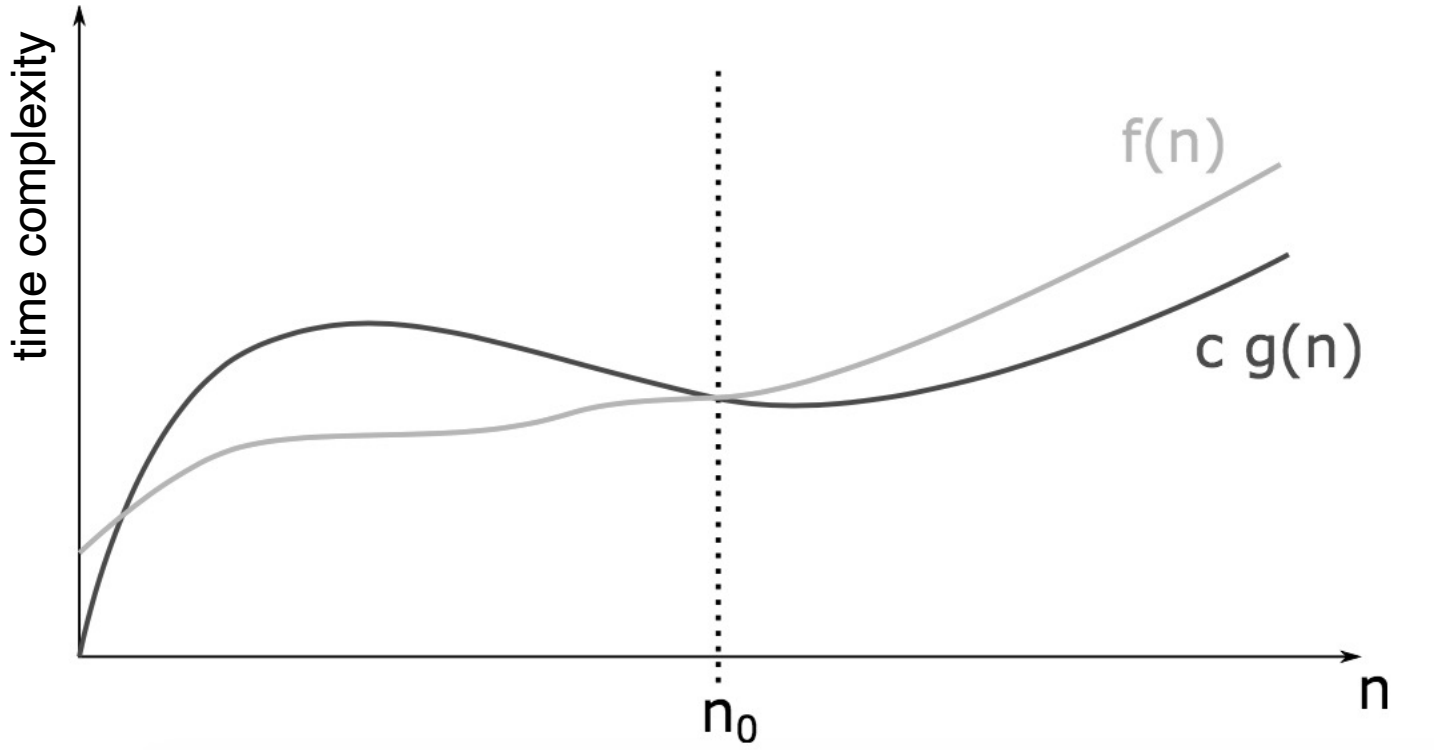
Upper Bound O

– $f \in O(g) \Leftrightarrow \exists c > 0 \; \exists \, n_0 \; \forall n > n_0 : |f(n)| \leq c \cdot |g(n)|$



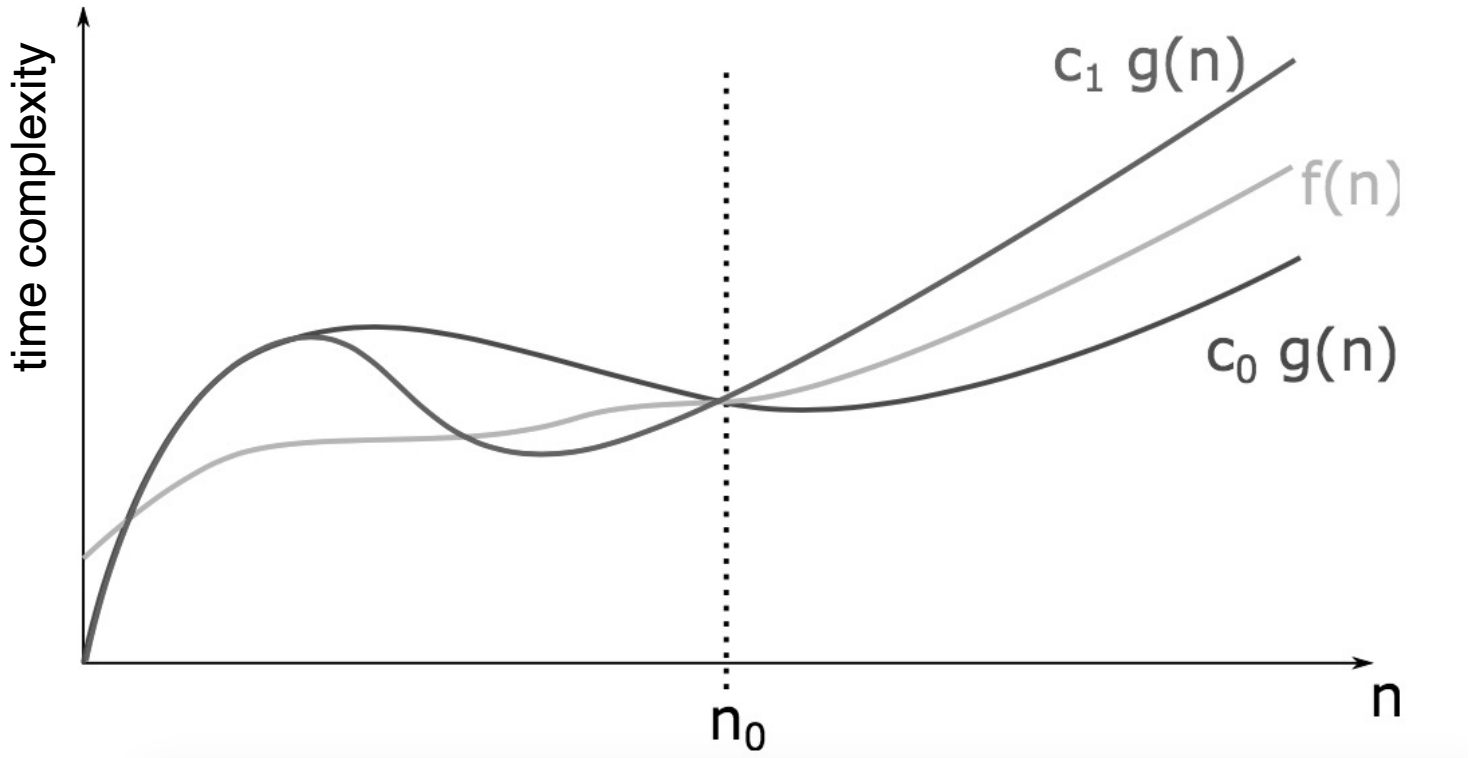
Lower Bound Ω

– $f \in \Omega(g) \Leftrightarrow \exists c > 0 \; \exists \, n_0 \; \forall n > n_0 : |f(n)| \geq c \cdot |g(n)|$

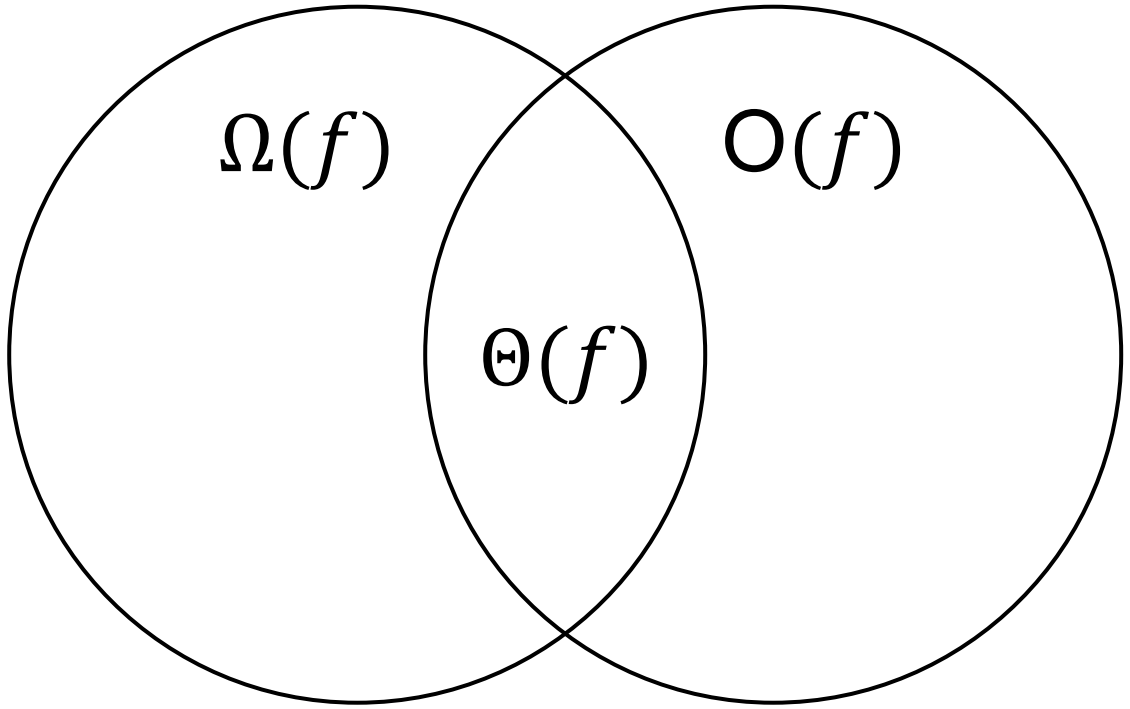


Exact Bound Θ

– $f \in \Theta(g) \Leftrightarrow \exists c_0 > 0 \exists c_1 > 0 \exists\, n_0\, \forall n > n_0 : c_0 \cdot |g(n)| \leq |f(n)| \leq c_1 \cdot |g(n)|$



O-Notation



O-Notation

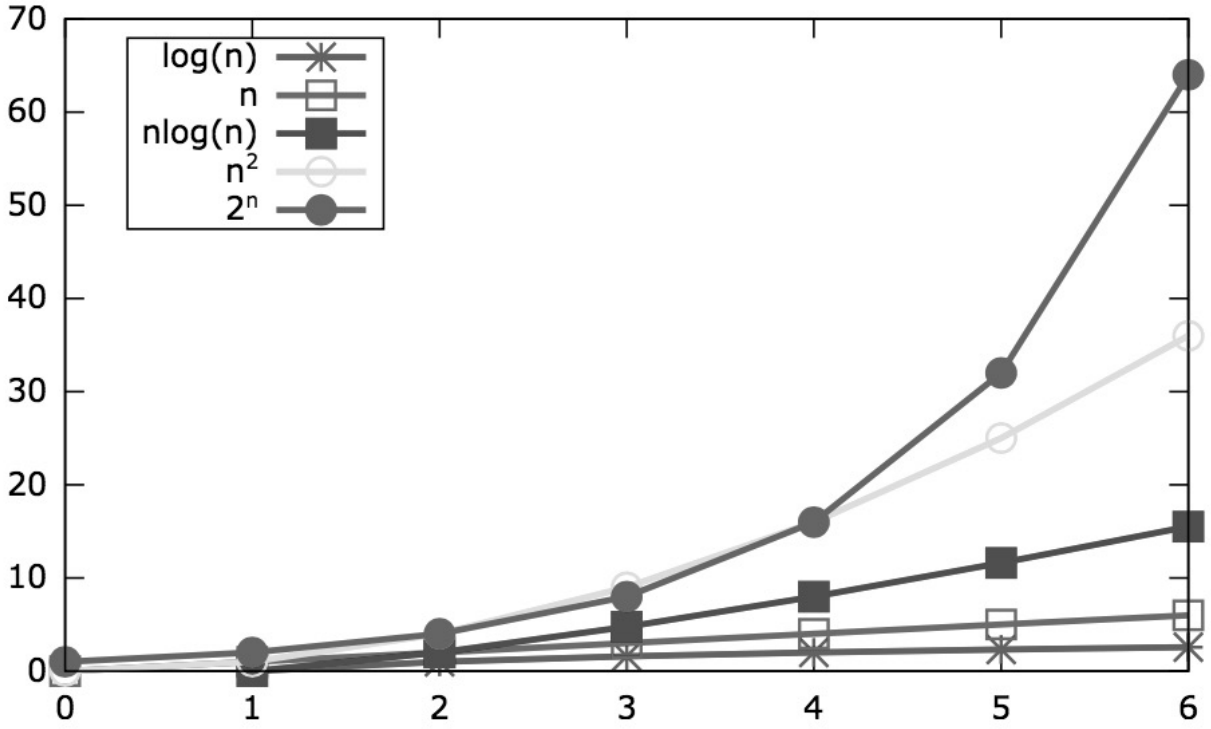
- Using O-Notation
  - Correct notation
    - People often write $f = O(g)$ instead of $f \in O(g)$
    - Strictly speaking that is mathematically incorrect but it is usually tolerated
  - Transformation rules
    - $f \in O(g)$
    - $c \cdot f \in O(g)$, for constant *c*
    - $O(O(f)) \in O(f)$, i.e., for $f \in O(g)$, $c \cdot f \in O(g)$, for constant *c*
  - Rules for addition
    - $f + g \in O(\max(f, g))$, i.e., for $f \in O(g)$, $f + g \in O(g)$
    - Also implies for polynomial *p* of degree *m*, $p \in O(n^m)$

O-Notation

- Use of the O-Notation for classification of functions:
  - Describes asymptotic upper, lower and exact bounds of a given function for large values
  - Interest is in dominant terms for large values only
    - More precise analyses are often very cumbersome
    - Constant factors may often not be of interest (depending on the application they may become relevant though…)
    - Linear acceleration is typically easy to achieve (new hardware, …)
  - Goal: description of complexity through classes of functions
    - Literally: “it can not get worse than f” if complexity is O(f)

Growth of Functions

– Growth of different classes of functions



Growth of Functions

– Growth of different classes of functions

n	1	10	100	1.000	10.000
$\log_{10} n$	0	1	2	3	4
$\log_2 n$	0	≈ 3	≈ 7	≈ 10	≈ 13
$\sqrt{n}$	1	≈ 3	10	≈ 32	100
n^2	1	100	10.000	1.000.000	100.000.000
n^3	1	1.000	1.000.000	1.000.000.000	1 Billion
2^n	2	1.024	$\approx 10^{30}$	$\approx 10^{301}$	$> 10^{3000}$
$1,1^n$	≈ 1	≈ 3	≈ 13.781	$\approx 2 \cdot 10^{41}$	$> 10^{414}$
$n!$	1	3.628.800	$\approx 9 \cdot 10^{157}$	...	...
n^n	1	10.000.000.000	10^{200}	10^{3000}	10^{40000}

O-Notation

– Common classes of functions

name	class
constant	$O(1)$
logarithmic	$O(\log n)$
polylogarithmic	$O(\log^c n)$
linear	$O(n)$
n-log-n growth	$O(n \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$O(n^c)$
exponential	$O(c^n)$

O-Notation

– What does this imply?

- Remember our earlier examples

naïve duplicate check (n^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

mergesort (n log n)

computer	thousand	million	billion
home	instant	1 sec	18 min
super	instant	instant	instant

Social Science Applications

- Why do we care about complexity?
  - Practical considerations
    - Algorithmic efficiency is often a pre-condition to doing any kind of data analysis in a reasonable time
    - Understanding which implementation choices make an algorithm inefficient is thus basic tradecraft
    - There are often simple problems in day-to-day code that can lead to massive slow-downs

Social Science Applications

- Why do we care about complexity?
  - "Hard" social science problems
    - There are a number of rather common social science applications that have very high problem complexity
    - Many of those examples are related to graphs/networks e.g., similar to the classical "traveling salesman" problem
    - We will not focus on those in this class but it is important to keep in mind if you ever work on (social) network problems in future…

Efficiency

Why Parallelize?

- How to solve problems/ do tasks more quickly
  - We could simply use faster processors
    - Until a few years ago realized through ever faster CPUs

year	CPU speed	processor
1990	33 MHz	486
1992	66 MHz	486DX2
1994	100 MHz	486DX4
1996	200 MHz	Pentium
1998	300 MHz	Pentium 2
2000	1 GHz	Pentium 3
2004	3,8 GHz	Pentium 4
2006	3,6 GHz	Pentium D
2008	3,2 GHz	Core 2
2010	3,2 GHz	Core i7
2011	3,9 GHz	AMD FX-8170
2012	4,2 GHz	AMD FX-4350
2013	4,7 GHz	AMD FX-9590

Why Parallelize?

- Fundamental limitations to increasing CPU speed
  - Physical and practical limitations
    - Silicon structures have to get ever smaller to further increase CPU clock speed
    - Smaller structures lead to more leakage and thus higher clock speed requires larger voltages
    - Larger voltage implies greater energy consumption
    - Classical example: Pentium 4 processor generation
      - Processor design specifically tuned towards high clock speeds BUT requires up to 115W power
      - Comparison: Newest generation MacBook Pro requires only 40-50W to run everything (including display etc.)

Why Parallelize?

- Fundamental limitations to increasing CPU speed
  - Idea for solution implies a paradigm shift
    - Use several processors with lower clock speeds
      - More energy efficient
      - Allows for smaller and lighter machines
    - Different ways to “multiply” processing
      - **Multiprocessor**: more than one CPU
      - **Multicore**: one CPU with several cores
      - **Multithreading**: single core executes more than one task in parallel
    - Overall performance of machines with this new paradigm is actually much higher…

Concurrent vs. Parallel

- Concurrency
  - Also referred to as virtual or logical parallelization
  - Task can be executed in several (independent) steps
  - Order of execution of tasks is not completely fixed
    - If ordering of tasks A and B is not fixed, the system can first execute A, then B or vice versa or execute both in parallel
    - Does not lead to speed gains by itself but enables them
- Parallelization
  - Tasks are at certain times executed in parallel
  - Requires hardware (and software) that permits the execution of tasks in parallel

Concurrent vs. Parallel

- Advantages of Concurrency
  - Good model of many real-world software tasks
    - Often have natural parallelization
    - Avoids artificially enforcing sequential execution when it is not necessary
  - Powerful paradigm for layered task execution
    - Example: loading a website
      - Text and graphics do not have to be loaded completely in sync
      - Natural parallelization of tasks within the loading task
  - Good starting point to achieve real parallelization
    - Requires that multiprocessing or multicores are available

Hardware Parallelization

- Parallelization Paradigms
  - **Instruction-level parallelism**
    - Parallel execution of machine instructions by the CPU
      - e.g., CPU with more than one execution unit
  - **Coprocessors**
    - Specialized additional processor optimized for specific operations
      - e.g., for floating point arithmetic or graphics
  - **Vector processor**
    - Implements instruction set on vectors rather than on single data items
      - Used in video-game consoles (e.g. PlayStation 5)

Hardware Parallelization

- Parallelization Paradigms (continued)
  - **Multicores**
    - several independent cores in a given system
      - e.g., your laptop in all likelihood is a multicore machine
  - **Cluster**
    - linking many computers with very high bandwidth connections
      - e.g., the Science Cloud at UZH, the national supercomputing cluster in Lugano
  - **Grid**
    - Linking of several clusters

Hardware Parallelization

- Coprocessors
  - Specialized additional processor optimized for specific operations
  - Mathematical coprocessors
    - Example: Intel 8087 (introduce in 1980) accelerated floating point calculation by a factor of up to 50 times
    - Today a standard component of practically any CPU
  - Other examples:
    - PhysX (Ageia) to accelerate simulations of physical space
      - Used for graphics in video games
    - Network processing units to speed up network processing

Hardware Parallelization

- Multicores
  - Originally implemented as symmetric multiprocessing (SMP)
    - several identical and independent CPUs, shared RAM and bus
    - relative expensive since they require specialize hardware
  - Today implemented as multicores
    - several processing cores on the same chip
    - much smaller distance between cores compared to SMP
  - Cores independently execute computing tasks on any data
  - Special case is simultaneous multithreading (SMT)
    - several components of the core are doubled, e.g. the register sets
    - most modern CPUs support this kind of multithreading with at least 2 fully parallel threads

Hardware Parallelization

- Cluster
  - Linking many computers together with very high bandwidth connections
  - Swiss National Supercomputing Center (CSCS) in Lugano is largest
    - Operated by ETH Zurich since 1991
    - Provides overall infrastructure to house compute nodes and home to different computing clusters
    - Example: EULER cluster of ETH
      - Several thousand nodes, each with between 12 and 64 cores
      - Uses shareholder model: users pay for initial hardware investment, cluster pays for power, maintenance, support etc.
  - Most similar infrastructure at UZH is the “Science Cloud” service

Software Parallelization

- Memory Sharing
  - All cores in a system have access to the same memory
    - Very fast
    - Needs hardware support to ensure cache-coherence
  - Every program only sees its own memory segment
    - Accessing other memory segment aborts execution
  - Shared Memory
    - Special memory segment provided by the OS that can be used by several programs at the same time
    - can be used like normal memory
    - programs have to use same identifier
    - segment has a fixed size

Software Parallelization

- Process vs. Thread
  - Idea: programming software to enable multi-threading, i.e., running different tasks in parallel within a given process (i.e. program)
  - Process
    - Program or routine is usually executed as a process
    - Process subsumes program code, memory, open data, network connections etc.
    - Every process has at least one thread
  - Thread
    - Sequential execution of elementary operations
    - Shares resources with other threads in the same process except stack memory

Software Parallelization

- Thread-based parallelization
  - OS distributes executable threads onto cores
    - Concurrence if only one core
    - Real parallelization if multicore
  - Communication via shared memory segment of process
  - Processor can quickly switch between threads
  - Starting new (sub)threads by forking, i.e., splitting an execution thread into several
  - Simplest and fastest parallelization strategy
    - Degree of parallelizability only limited by hardware
    - In practice, many small hidden problems

Software Parallelization

- Thread-based parallelization: requirements for programming language
  - Generating and starting new threads
    - Every process has initially one thread
    - Programming language has to enable starting new threads at runtime
  - Coordination among threads
    - Thread A waits for (intermediate) result of thread B
    - Thread A and thread B may want to change the state of an object at the same time
    - Can be tricky because no thread knows the current state of other threads
  - Control of thread execution
    - Scheduler distributes processor resources to threads
    - Different types of scheduling (ahead of time or interactive)

Software Parallelization

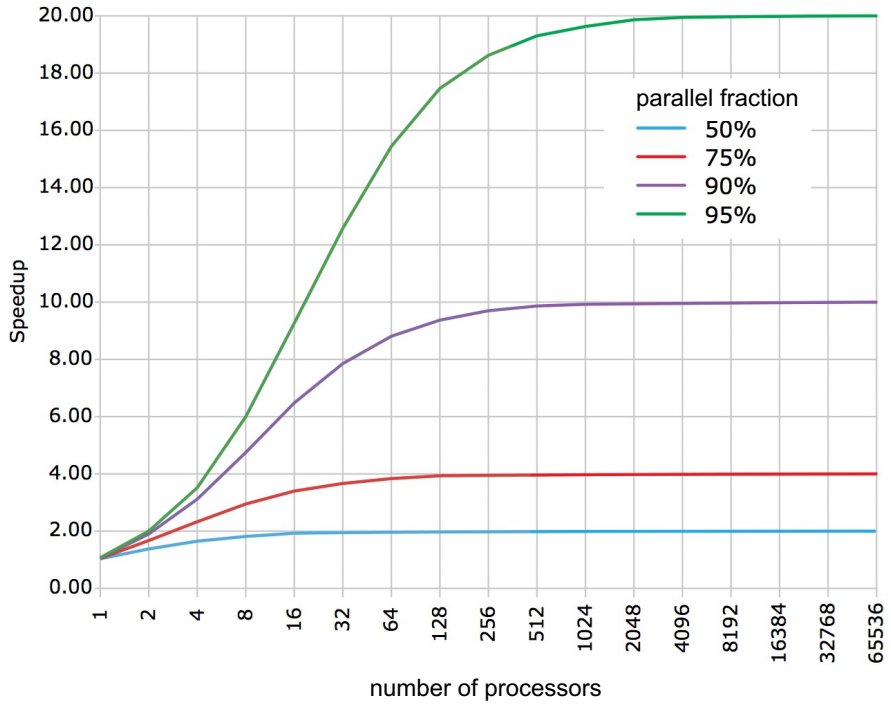
- Process communication
  - Several processes often work in parallel on the same problem
    - Processes on same machine
    - Processes on cluster
  - Coordination through messages
    - Within the OS
    - Through the network
  - Example protocols
    - IPC (inter-process communication)
    - MPI (message passing interface)

Limits to Parallelization

- Can you actually achieve an optimal speedup?
  - Gene Amdahl (1967): there is an upper limit
    - Intuition:
      - There is always a part of the code, that can not be parallelized
      - With increasing parallelization, this part dominates the overall runtime of the program
    - Formally:
      - maximal speedup: $S = \frac{1}{(1-P)+\frac{P}{N}}$
        - P: fraction of parallelized processing, e.g. 80%
        - N: number of processors / threads

Limits to Parallelization

– Amdahl's law



Social Science Applications

- Parallel programming in social science practice
  - Parallelization within the program/ algorithm
    - Use packages that, whenever possible, apply multi-threading etc. to computationally heavy operations
      - e.g., parallel execution of (independent) for-loops
      - Optimized statistical software (e.g. for Monte-Carlo calculations)
    - This can include using several cores on cluster computers but may also simply use multicore in your laptop

Social Science Applications

- Parallel programming in social science practice
  - Massive parallel computations
    - Suitable for tasks that are inherently independent
      - Bootstrapping in statistics
      - Enumerating a simulation model
    - Distribute independent tasks on cluster computers using a LSF (platform load sharing facility) job scheduler
    - After execution run a simple script to collect and “summarize” results of parallel runs

Up Next

- Exercise (this afternoon)
  - Efficient data management
  - apply vs. dplyr
- Next lecture (tomorrow morning)
  - (Digital) data sources
  - Data quality and collection
  - Implications for research