Distributed Computing

Wolf-Tilo Balke & Pierre Senellart
IFIS, Technische Universität Braunschweig
IC2, Telecom ParisTech

Distributed Computing

- Basic Definition
  - A **distributed system** consists of multiple autonomous computers (nodes) that communicate through a computer network via message passing
  - The computers interact with each other in order to achieve a **common goal**
  - A computer program that runs in a distributed system is called a **distributed program**

Distributed Computing

- What are **economic and technical drivers** for having distributed systems?
  - **Costs:** better price/performance as long as commodity hardware is used for the component computers
  - **Performance:** by using the combined processing and storage capacity of many nodes, performance levels can be reached that are out of the scope of centralized machines
  - **Scalability/Elasticity:** resources such as processing and storage capacity can be increased incrementally
  - **Availability:** by having redundant components, the impact of hardware and software faults on users can be reduced

Applications

- Telecommunication networks
  - Telephone and cellular networks
  - Computer networks and the Internet
  - Wireless sensor networks

- Network applications
  - World Wide Web and Peer-to-Peer networks
  - Massively multiplayer online games

Applications

- Distributed databases and distributed information processing systems such as
  - Banking systems
  - Airline reservation systems
  - Real-time process control like aircraft control systems.

- Parallel computation
  - Scientific computing, including cluster and grid computing

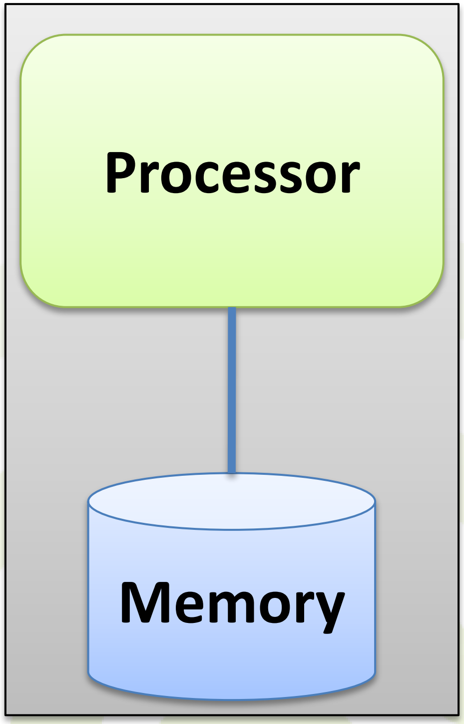
How to do it?

- The key to success
  – **Divide a problem into many tasks**: each is solved by one or more computers

- Problems
  – The system has to tolerate failures in individual nodes
  – The structure of the system (network topology, latency, number of nodes) is not known in advance
  – The system may consist of different kinds of computers and network links
  – The system may change during the execution of a distributed program
  – Each node has only a limited, incomplete view of the system

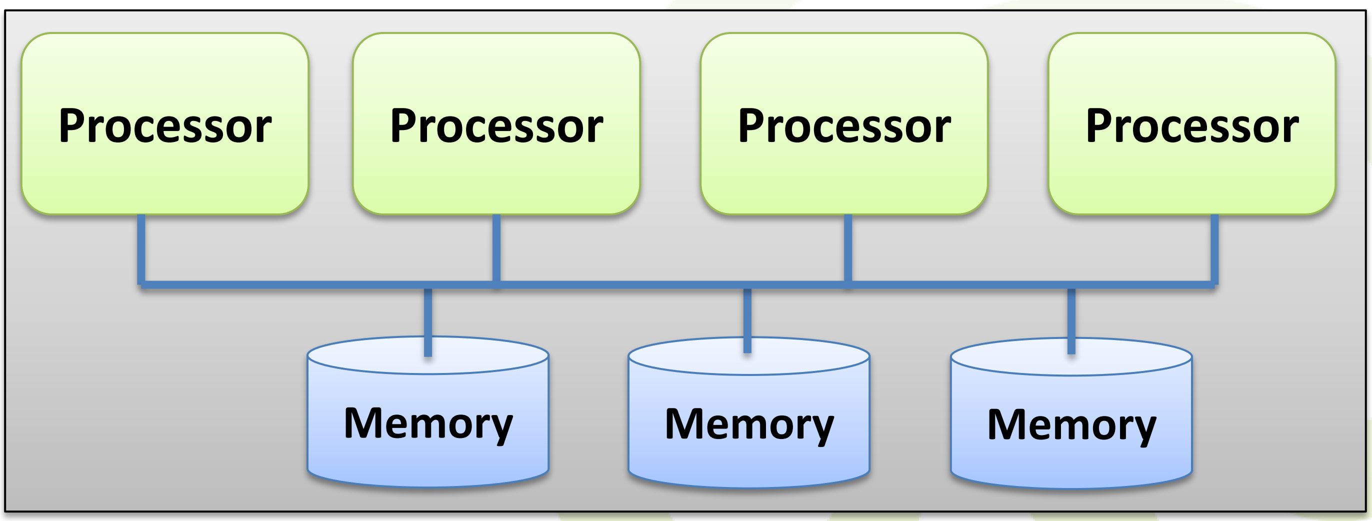
Hardware Architecture

- Uniprocessor
  - Single processor
  - Direct memory access



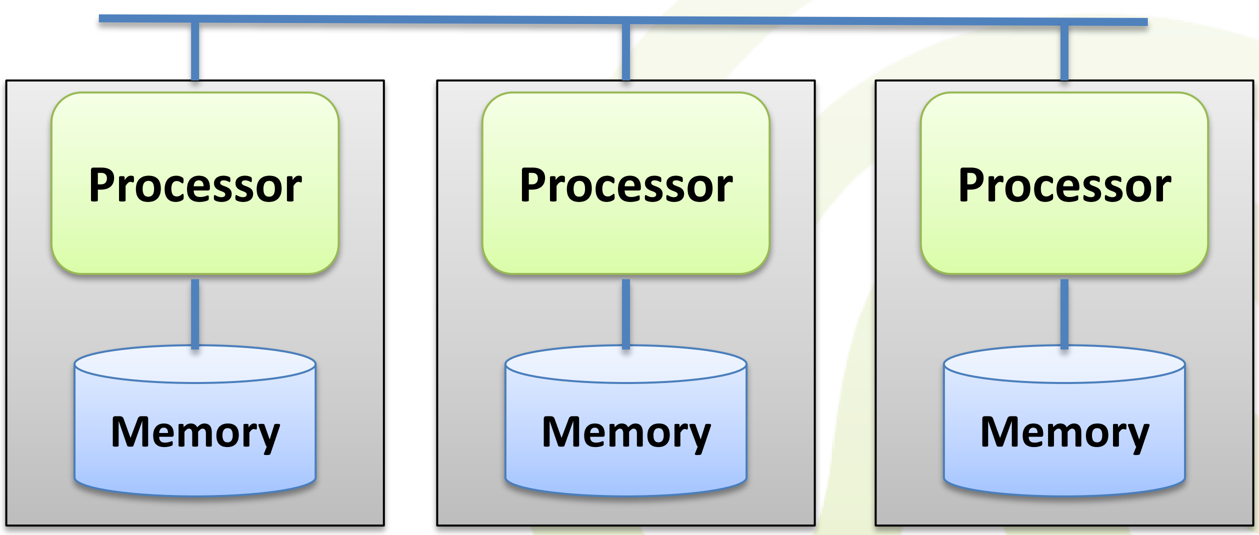
Hardware Architecture

- Multiprocessor
  - Multiple processors with direct memory access
  - Uniform memory access (e.g., SMP, multicore)
  - Nonuniform memory access (e.g., NUMA)



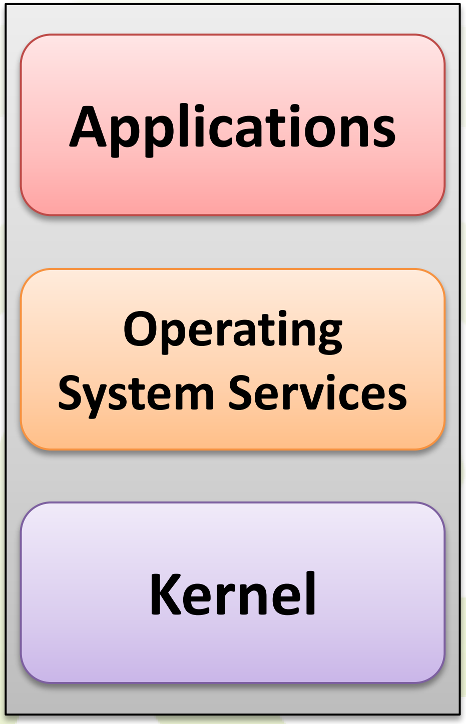
Hardware Architecture

- Multicomputer
  - Multiple computers linked via network
  - No direct memory access
  - Homogeneous vs. Heterogeneous



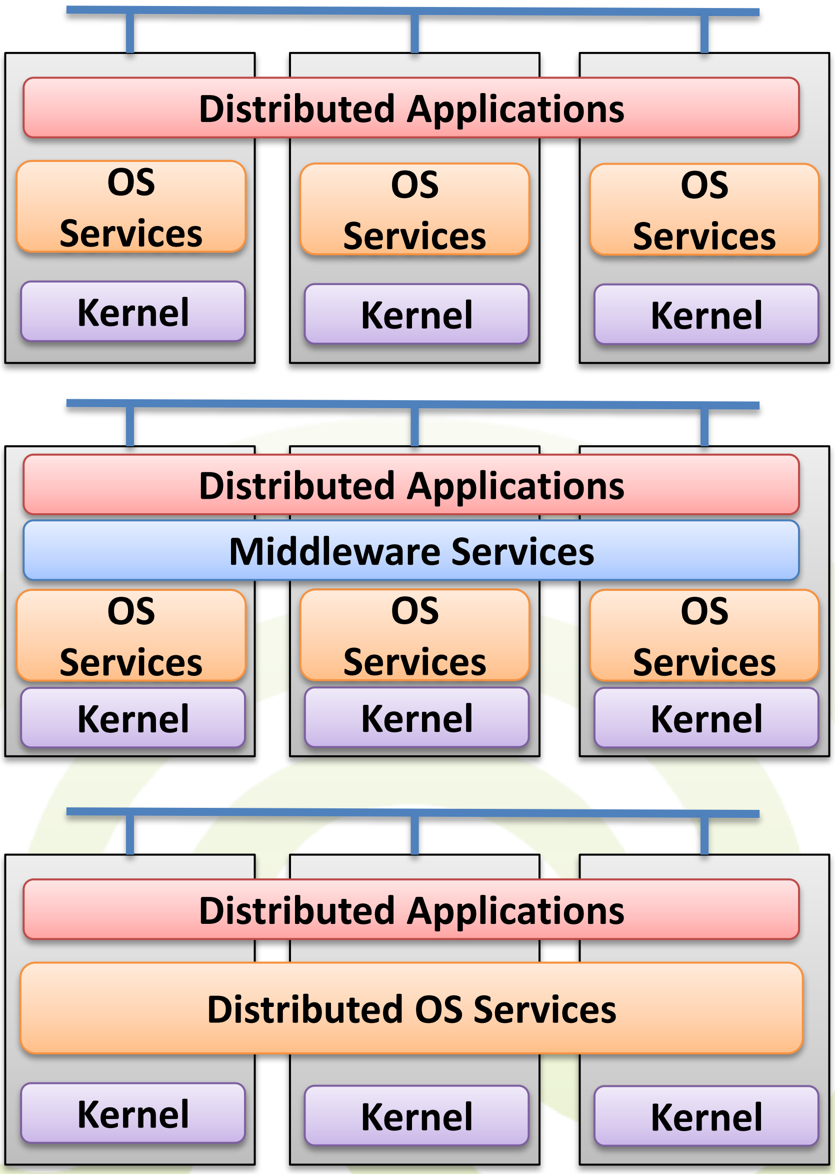
Software Architecture

- Similar for uniprocessors and multiprocessors
  - But for multiprocessors: the kernel is designed to handle multiple CPUs and the number of CPUs is transparent



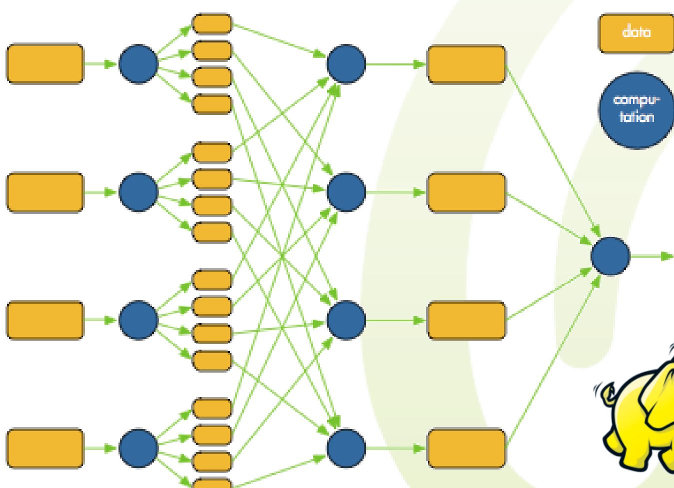
Software Architecture

- For multicomputers there are several possiblities
  - Network OS
  - **Middleware**
  - Distributed OS



This Course

- Not about architectural issues
  - A lot of open discussions that would fill our time slot completely…
- Our main focus: **scalability and time**



Response Time Models

- "Classic" cost models focus on **total resource consumption** of a task
  - Leads to good results for **heavy computational load** and **slow network connections**
    - If execution plan saves resources, many threads can be **executed in parallel** on different machines
  - However, algorithms can also be optimized for **short response** times
    - "Waste" some resources to get first results earlier
    - Take advantage of lightly loaded machines and fast connections
    - Utilize intra-thread parallelism
      - Parallelize one thread instead of cuncurrent multiple threads

Response Time Models

- **Response time models** are needed!
  - "When does the **first piece of the result** arrive?"
    - Important for Web search, query processing,
  - "When has the **final result** arrived?"

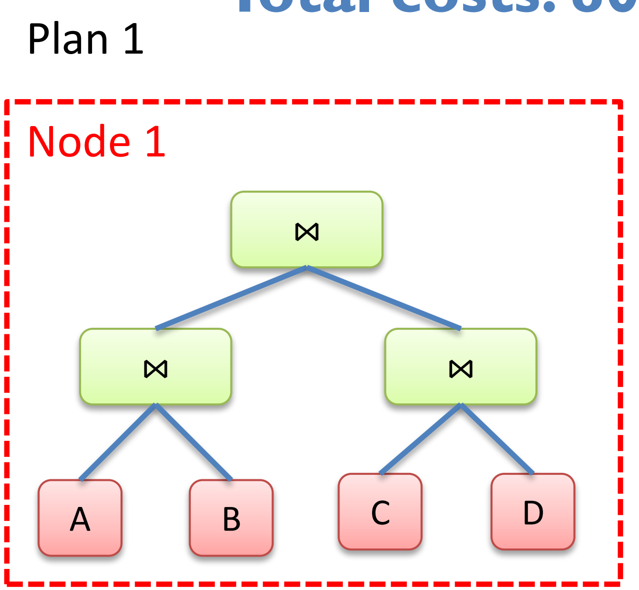
Detour

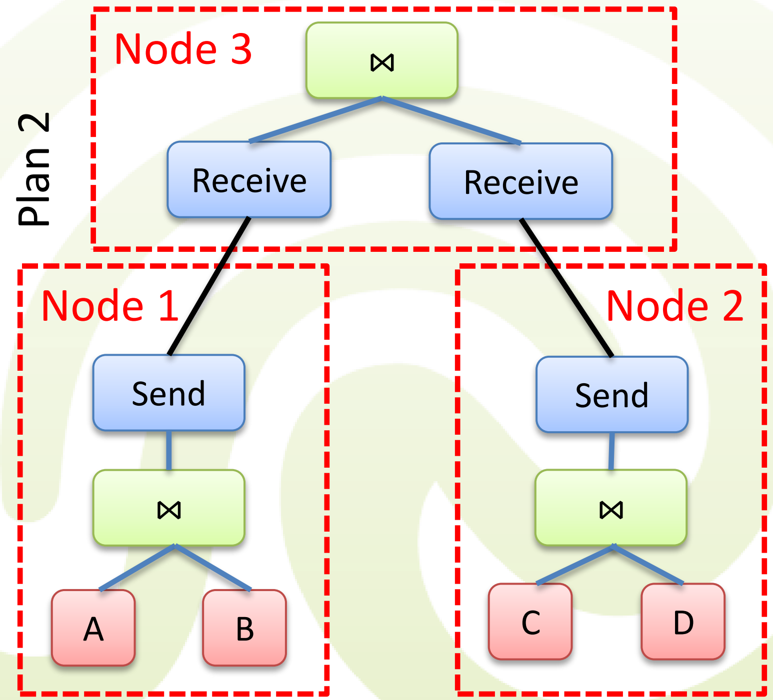
- **Example**
  - Assume relations or fragments A, B, C, and D
  - All relations/fragments are available on all nodes
    - Full replication
  - Compute $(A \bowtie B) \bowtie (C \bowtie D)$
  - **Assumptions**
    - Each join costs 20 time units (TU)
    - Transferring an intermediate result costs 10 TU
    - Accessing relations is free
    - Each node has one computation thread

Distributed Query Processing

Detour

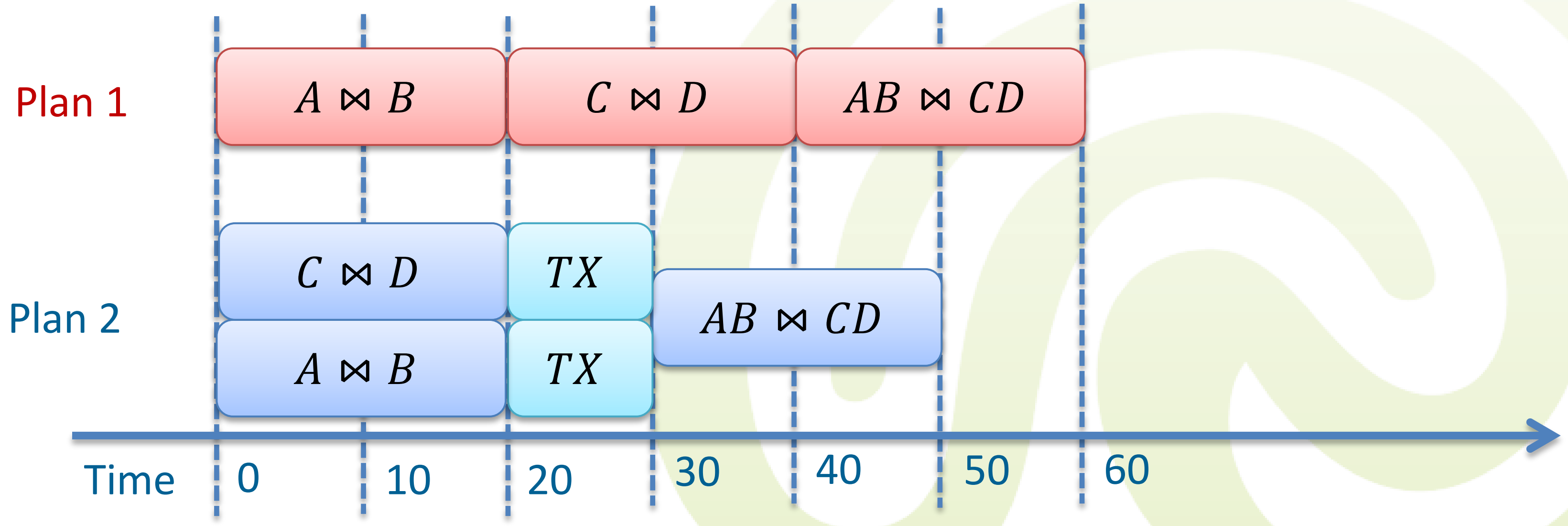
- Two plans:
  - Plan 1: Execute all operations on one node
    - **Total costs: 60**
  - Plan 2: Join on different nodes, ship results
    - **Total costs: 80**





Detour

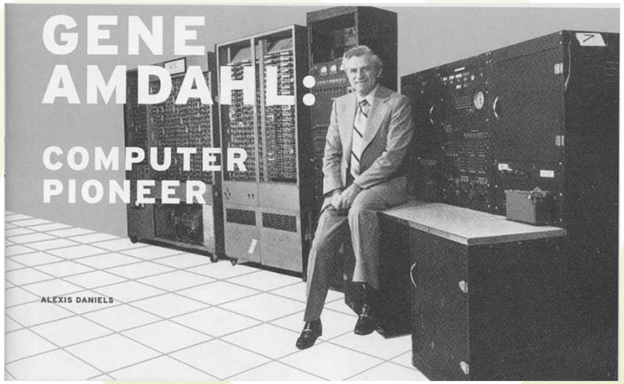
- With respect to total costs, plan 1 is better
- **Example (cont.)**
  - Plan 2 is better wrt. to response time as operations can be carried out in parallel



- **Response Time**
  - Two types of response times
    - **First Tuple** & **Full Result** Response Time
- Computing **response times**
  - Sequential execution parts
    - Full response time is sum of all computation times of all used operations
  - Multiple parallel threads
    - Maximal costs of all parallel sequences

Response Time Models

- **Considerations**:
  - How much speedup is possible due to parallelism?
    - Or: "Does kill-it-with-iron" work for parallel problems?
  - Performance **speed-up of algorithms** is limited by **Amdahl's Law**
    - Gene Amdahl, 1968
    - Algorithms are composed of parallel and sequential parts
    - **Sequential code fragments severely limit potential speedup of parallelism!**



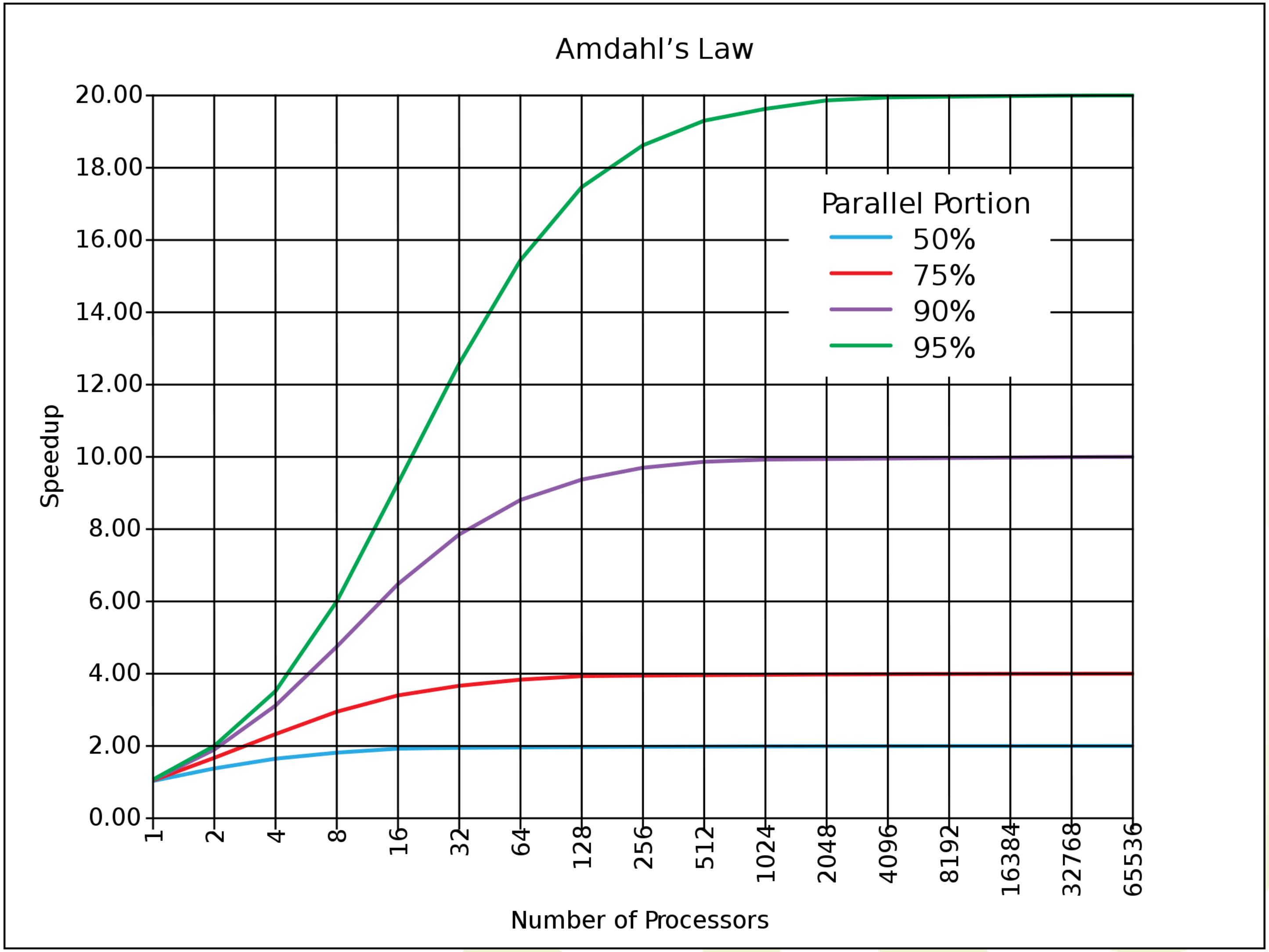
Response Time Models

– Possible maximal speed-up:
  - $maxspeedup \leq \frac{p}{1+s*(p-1)}$
  - p is number of parallel threads
  - s is percentage of single-threaded code

– e.g. if 10% of an algorithm is sequential, the maximum speed up regardless of parallelism is 10x

– For maximal efficient parallel systems, **all sequential bottlenecks** have to be identified and **eliminated!**

Response Time Models



Response Time Models

- Good **First item Response** benefits from operations executed in a pipelined fashion
  - Not **pipelined**:
    - Each operation is fully completed and a **intermediate result** is created
    - Next operation reads intermediate result and is then fully completed
      - Reading and writing of intermediate results costs resources!
  - **Pipelined**
    - Operations **do not create intermediate** results
    - Each finished tuple is fed directly into the next operation
    - Tuples "**flow**" through the operations

Response Time Models

- Usually, the result flow is controlled by **iterator interfaces** implemented by each operation
  - "Next" command
  - If execution speed of operations in the pipeline differ, results are either cached or the pipeline blocks
- Some operations are more suitable than others for pipelining
  - **Good:** selections, filtering, unions, …
  - **Tricky:** joining, intersecting, …
  - **Very Hard:** sorting