Systems Programming – Part 2
Concurrent Systems Programming

Dr Lauritz Thamsen
lauritz.thamsen@glasgow.ac.uk
https://lauritzthamsen.org

Topics of Part 2 of SP(GA)

- Intro to Concurrency (with Processes and Threads)
- Process/Thread Synchronisation
- More on Process Management (from an OS Perspective)
- **Concurrency Beyond Threads & Limits of Scalability**
- Virtual Memory & Levels of Storage

Lecture Outline

- **Intricacies of (Low-Level) Concurrency**
- (Higher-Level) Abstractions for Coordination
- Options Within a Programming Language
- Scalability and Limits of Scalability

Low-Level Concurrency is Hard

A concurrent program poses all of the challenges of sequential Computation: i.e. **what** to compute.
In other words: A correct and efficient algorithm, using appropriate data structures, must be constructed.

A concurrent program must **also** specify a correct and effective strategy for **Coordination**: i.e. **how** threads should cooperate.

Concurrency Reflection

- Safely and correctly managing concurrent threads that share state is tricky
- If you use locks, you must correctly lock, unlock, and wait to arrange safe access
- You must avoid problems of

  Starvation: some thread never makes progress

  Livelock: threads run, but make no progress

  Deadlock: threads wait for each other to release locks

Some Important Coordination Aspects

Partitioning: determining what parts of the computation should be evaluated separately

- e.g. a thread to render each frame of a film

Placement: determining where threads should be executed

- e.g. allocate thread to the least busy core

Communication: when to communicate and what data to send

- e.g. film-rendering threads may hold two frames: one being processed and another ready to go next, as soon as the current frame is processed

Synchronisation: ensuring threads can cooperate without interference

- e.g. if two threads need to do work on the same frame, only one should change it at a time

University of Glasgow

Lecture Outline

- Intricacies of (Low-Level) Concurrency
- **(Higher-Level) Abstractions for Coordination**
- Options Within a Programming Language
- Scalability and Limits of Scalability

Coordination Abstraction Levels

You have probably used several notations for specifying, designing & constructing **computations** but relatively few for coordination.

Computations can be written in languages with different **levels of abstraction**, e.g.

Low-level	**Mid-level**	**High-Level**
Assembly	Java	SQL
C	Python	Prolog
		Haskell

Coordination Abstraction Levels

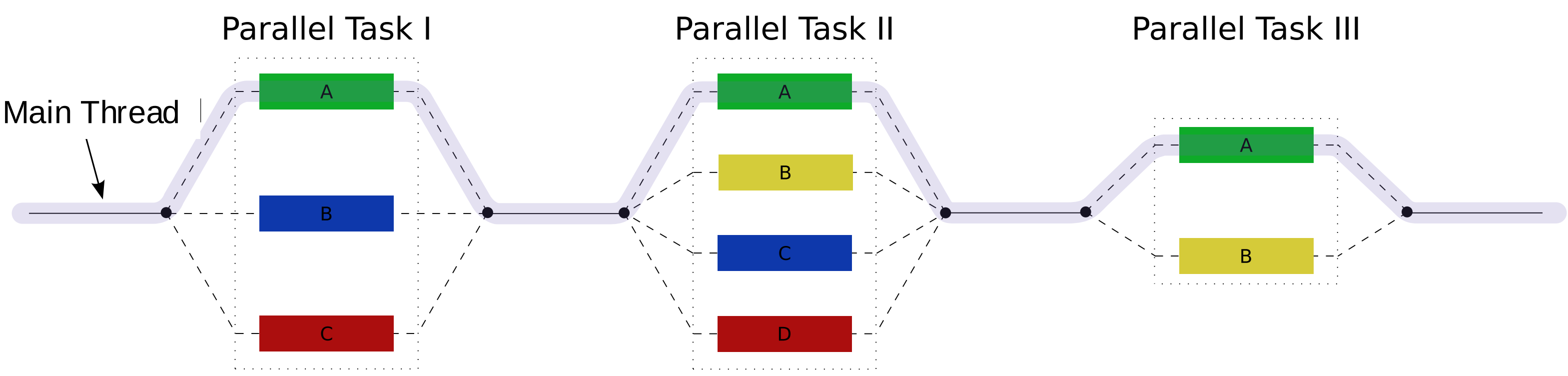
- Likewise, **coordination** can be written in languages with different levels of abstraction, e.g.

Low-level	**Mid-level**	**High-Level**
Mutexes	Go	OpenMP
Semaphores	Monitors (Java threads)	
	C++ Threads	Erlang

High-Level Example: OpenMP

- “Fork-Join” parallelism, repeatedly synchronizing independent threads (shared memory, but good practice not to share much state between threads)

```
#pragma omp parallel for
for(int i = 1; i < 100; ++i)
{
   // 100 independent computations
}
```



Lecture Outline

- Intricacies of (Low-Level) Concurrency
- (Higher-Level) Abstractions for Coordination
- **Options Within a Programming Language**
- Scalability and Limits of Scalability

Concurrent Coordination Options for Languages

A programming language may have **several concurrency means**, often competing, e.g.

C++ has:

- **Thread libraries**, e.g. POSIX
- **std threads**
- …

Java has:

- **Thread libraries,** e.g. POSIX
- **Java threads**
- **Executors**
- …

“Green Threads”

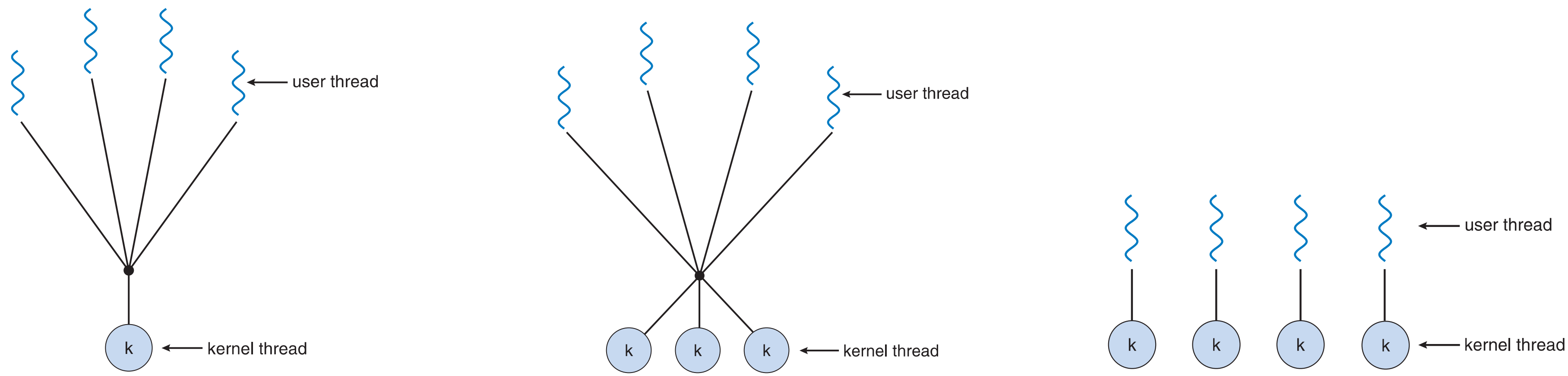
- Language-level **concurrency**: a thread that is managed by a runtime library or language virtual machine (== user threads), not natively by the operating system (== kernel threads)

- Pioneered by Java (but later abandoned for “real” threads)
- Widely used: Python’s Global Interpreter Lock, Fibers in Ruby (< 1.9), Goroutines in Go, Haskell, Erlang, …

- Either “**cooperative** multi-tasking” (== user threads have to yield voluntarily) or “**preemptive** multi-tasking” (== active user thread scheduling)

User and Kernel Thread Mappings

In the end, only kernel threads run on processors, so we need to map user threads to kernel threads:



Many-to-one
(== Green
Threads)

Many-to-many
(~ using a pool
of kernel threads)

One-to-one
(== threading in C)

Remember: We have threads <u>and</u> processes, giving us the flexibility to run also green-threaded programs in parallel (using multiple processes)

"Green Threads"

- Advantages
  - Lightweight: Creating, managing, and destroying kernel threads requires time and memory (i.e. reserving each thread's stack)
  - Concurrency in a language – independent of OS and hardware support for concurrency/parallelism
  - Easy-to-use concurrency (easy to have mutual exclusion, easier to reason about possible interleaving, especially with cooperative multi-tasking)
- Disadvantage: Mapped to a single kernel thread → can only offer concurrency, not parallelism
  - Cannot take advantage of parallel hardware
  - A single blocking user thread can block all others (if there is no preemption)
  - A user thread blocking on I/O can move all the user threads out-of-core

Lecture Outline

- Intricacies of (Low-Level) Concurrency
- (Higher-Level) Abstractions for Coordination
- Options Within a Programming Language
- **Scalability and Limits of Scalability**

A Note on Performance

- Which is the fastest?
  - … for winning on a racetrack?
  - … for transporting a family to the beach?
  - … for delivering five pizzas to different customers in a city?

- Performance depends not only on the **execution environment**, but also on the **workload**!

5 x

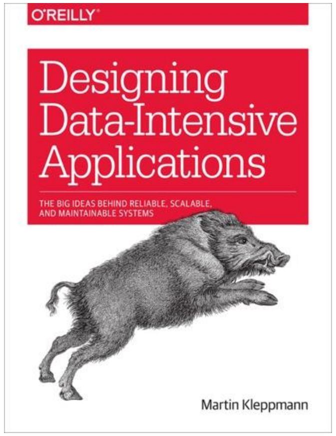
Scalability

- **Performance**: load a system can handle (at a certain scale)
  - Usually calculated as the mean, median, or x-percentile of load measurements (usually as throughput or latency)

- **Load**: amount and properties of work, used to measure scalability
  - e.g. requests per second, read/write ratios, cache hit rates

- **Scalability**: describes a system's ability to cope with increased load:

  "The system supports growths
  (in data volume, traffic volume, or complexity)
  with reasonable ways of dealing with it
  (e.g. more resources)."



Types of Scaling

- ***Strong scaling***: Fixed load, different resources
  - Best case: linear scalability (e.g. twice the VMs → half the runtime)
  - There are usually diminishing returns from some point

- ***Weak scaling***: Scale resources *and* load proportionally
  - Can we use e.g. twice as many resources to handle twice the load (with the same performance?
  - Typically, this is less bounded

Theoretical Limit: Amdahl's Law

"The speedup of a program using multiple processors for parallel computing is limited by the sequential fraction of the program"

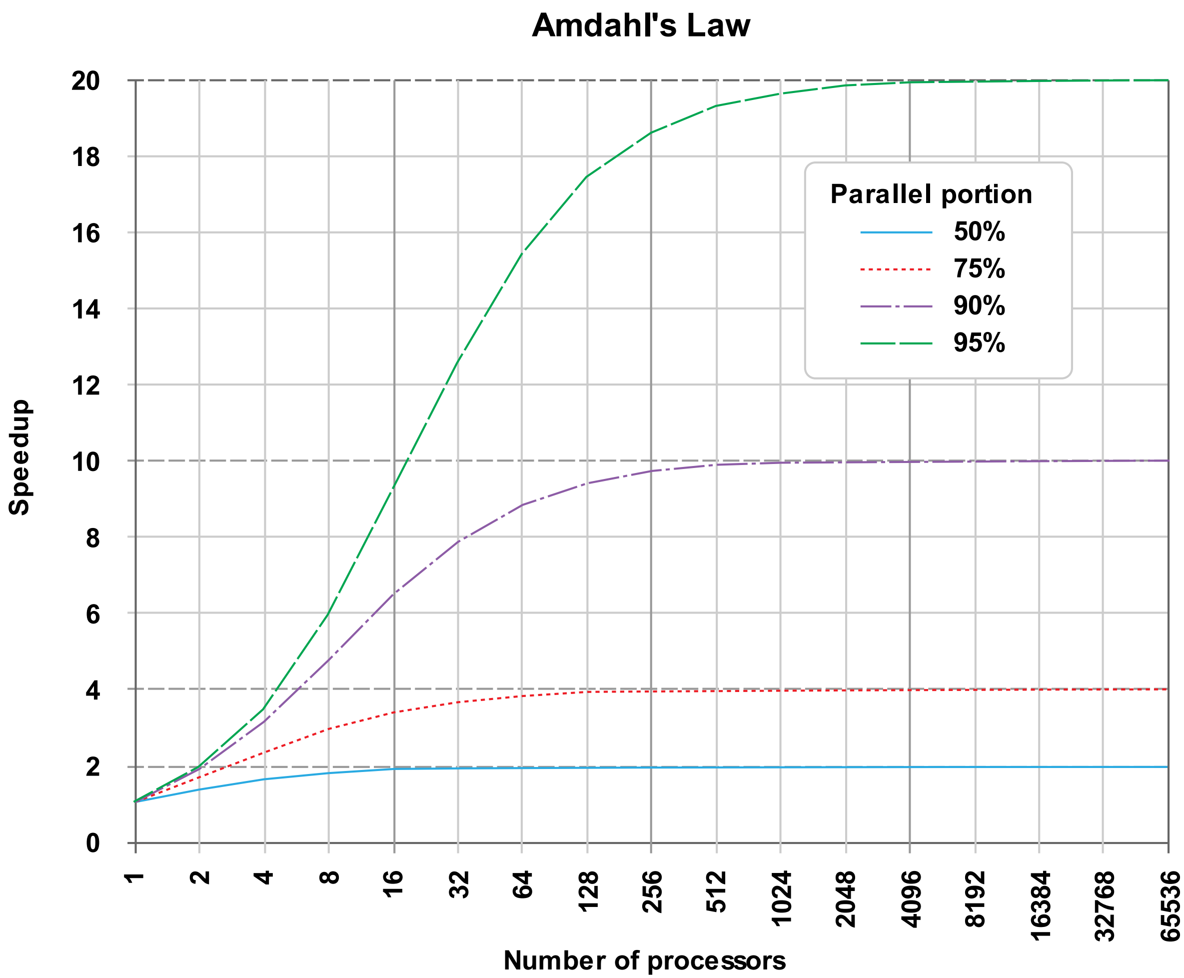
$$S_{total} = \frac{1}{(1-p) + \frac{p}{s}}$$

p: fraction of the program that benefits from parallelization

s: degree of parallelization (e.g. number of cores used)

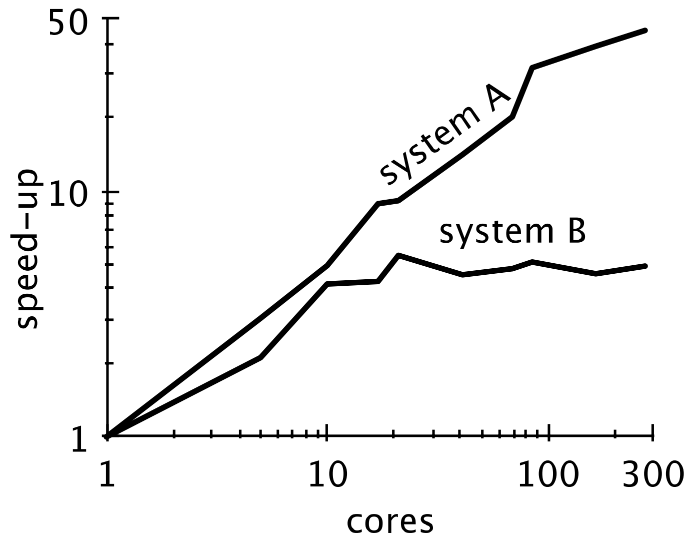
- Note: as s approaches infinity, $\frac{p}{s}$ approaches 0, so S_{total} becomes $\frac{1}{(1-p)}$
  - e.g. 50% parallel code, p = 0.5 → speedup of at most 2
  - e.g. 95% parallel code, p = 0.95 → speedup of at most 20

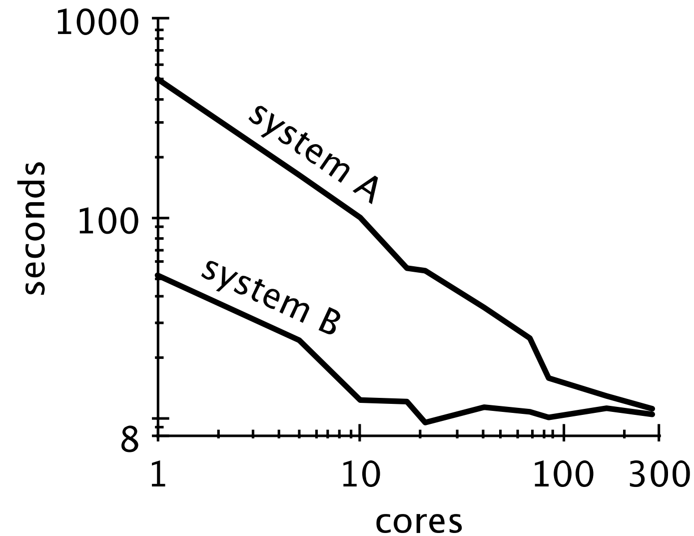
Theoretical Limit: Amdahl's Law



A Word of Caution

- Easiest way to reach good scalability? → Have a poor baseline speed!





- Scalability of System B is worse… because of its better performance!

Recommended Reading

- Recommended reading:
  - Silberschatz, Galvin, Gagne, Operating Systems Concepts, Sections 4.2-4

- Further reading recommendations (beyond this course):
  - C. Breshears, “The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications”, 2009
  - M. Kleppmann, “Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”, 2017
  - F. McSherry; M. Isard; D. G. Murray: “Scalability! But at what COST?”, HotOS XI, 2015.