Concurrency

Moore's law 1965
https://en.wikipedia.org/wiki/Moore27s_law#/media/File:Moore's_Law_Transistor_Count_1971-2018.png

# of integrated circuit transistors duplicates in 24 month
(at least for the next 10 years)
for long time the speed also duplicated in the same period

Herb Sutter: The Free lunch is Over 2005 DrDobb's Journal
http://www.gotw.ca/publications/concurrency-ddj.htm
The clock speed cannot be increased
--> applications should be utilize multicore systems

Concurrency =/= Parallelism

Why to use concurrent solutions?
More intuitive to solve the problem
- Server or graphical applications
- X-Window over DOS - non-preemptive scheduling
(thread runs until some external event occurs, mostly I/O)
Running on different devices
More efficient/scalable

Concurrency levels
==========

o Process level
+ Separate address space: better for security and stability
+ No shared memory: no data race
+ Can run on physically different hardware nodes
- Costly to start a new process (spawn or fork-exec)
- Costly to switch between processes
- Costly to communicate
(IPC = signals, messages, semaphores, shared mem., pipe)
o Thread level
o Thread as user space lib
+ Faster start and context switch
- I/O can block the whole branch of threads
o Thread as kernel lib (M:N)
+ Real concurrent behavior
- More expensive start and switch
(still slightly cheaper than start a new process)
o Corutines
o Stackless
o Stackfull

Amdahl's law:
P portion of parallelly executable code
N execution units

Performance gain is:

$$\frac{1}{(1-P) + \frac{P}{N}} \xrightarrow{n \to \infty} \frac{1}{1-P}$$

e.g. if P = 0.95 N->OO 20x
P = 0.90 10x

Communication and syncronization models
==========

Issues
======

Data race
==========
Concurrent non-atomic actions on the same memory location
at least one of them update

Solution:
busy waiting / spin lock
mutex/semaphore (semaphore: Dijkstra 1968)
lock_guards
conditional variables
spurious wake up

data channels (Golang)
(future-promise in C++)

Deadlock
========
- Job interview :)

```
bool operator<(A a, B b)
{
 lock_guard(a);
 lock_guard(b);
 return a < b;
}
```

t1: x < y t2 : y < x

All of these conditions should occure:
- mutual exclusion
- hold and wait locking
- no preemption (like database manages)
- circular dependences

Solution:
 algoritms, like Banker's one (???)
 setting the lock always in some predefined order
 detecting deadlock and interact (databases)

std::lock(Args... args);

Starvation
==========
(resource) starvation
e.g. naive RW lock approaches

C++11 memory model
================
std::atomic<>

sequential consistency (Leslie Lampard 1977)
release-acquare
release-consume
relaxed

- promise-future
- std::async()
- C++17 parallel STL

Others
======

SIMD == Single Instruction Multiple Data
RCU == Read-copy-update patterns

False claims:
 If it concurrent: it is faster
 - Actually, it can be even slower
 - Start a (system) thread is expensive.
   Parallel STL in C++17 starts new threads only over ~10000 elems
Easy to write a sequential prototype and than rewrite it parallel

Hard to debug
  n statements, t threads   (nt)!
possible execution paths = ------
                            n!^t