Utrecht University

# B3CC: Concurrency

## *01: Introduction*

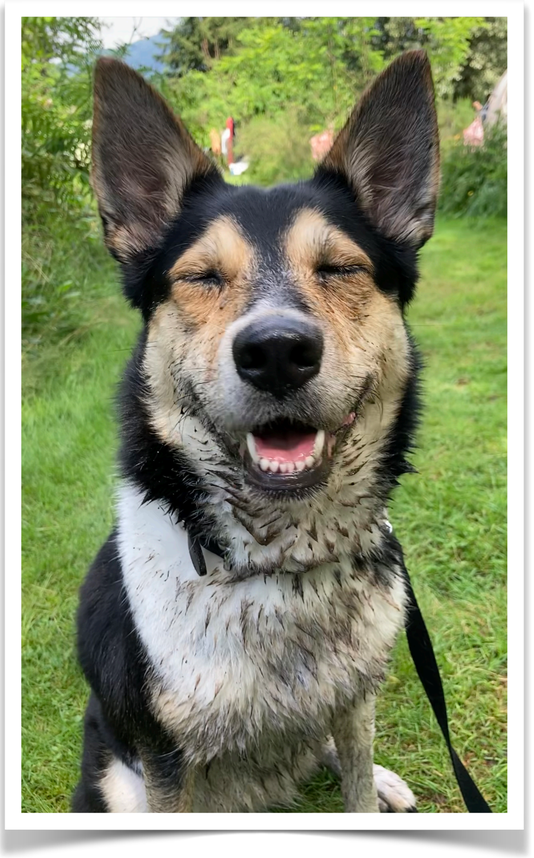
Ivo Gabe de Wolff

# Hello!

- Ivo Gabe de Wolff
  - i.g.dewolff@uu.nl
- Tom Smeding
  - t.j.smeding@uu.nl

- Most slides and materials were made by Trevor L. McDonell

- Working groups are guided by
  - Ivo Gabe de Wolff
  - Nico Naus
  - And 6 TAs



# Today

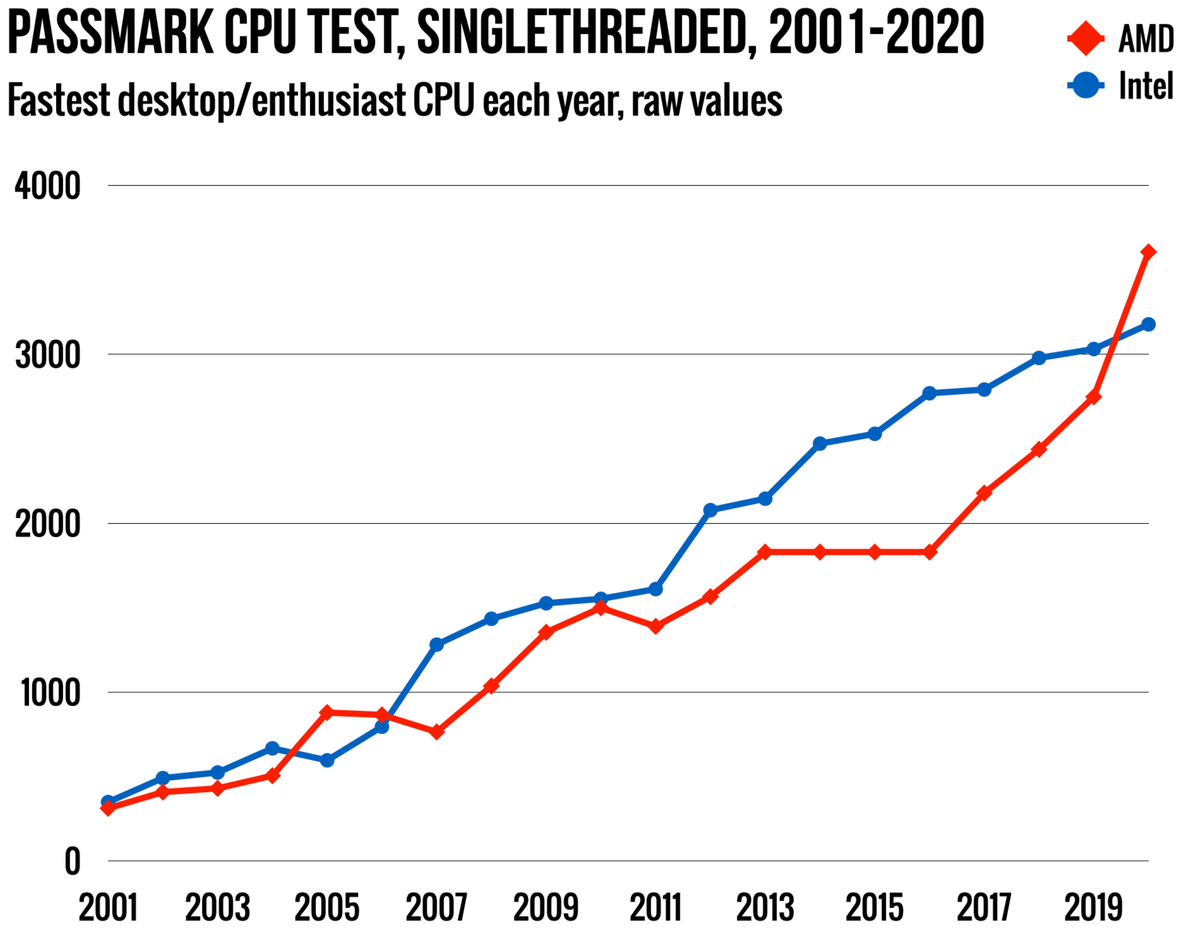
1. Motivation
2. Course formalities

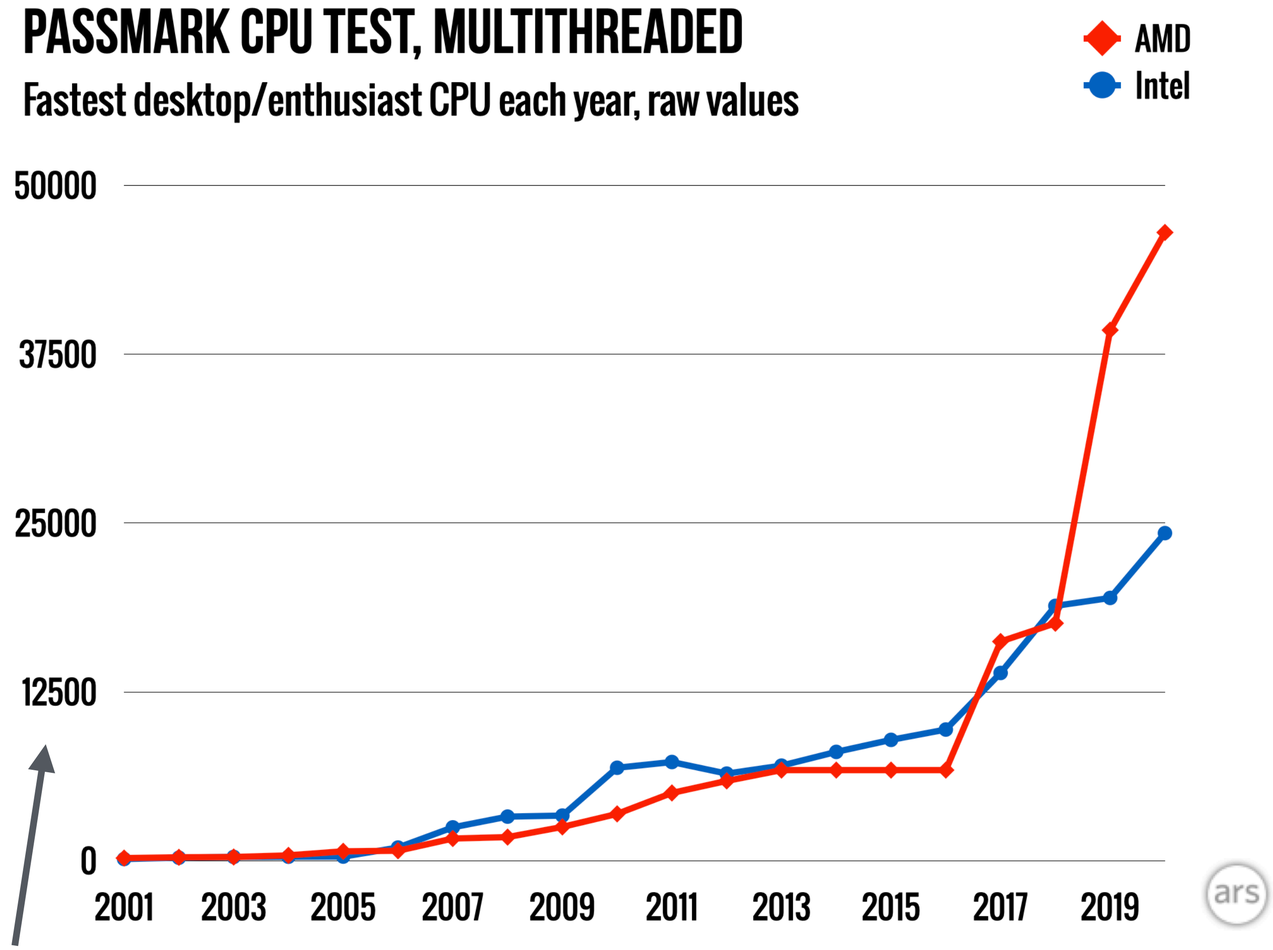
# Motivation

# Motivation

- **Concurrency:** dealing with lots of things at once
  - Collection of independently executing processes
  - Two or more threads are making progress
- **Parallelism:** doing lots of things at once
  - Simultaneous execution of (possibly related) computations
  - Two or more threads are executing simultaneously

# Performance improvements over the years




*Different scale*

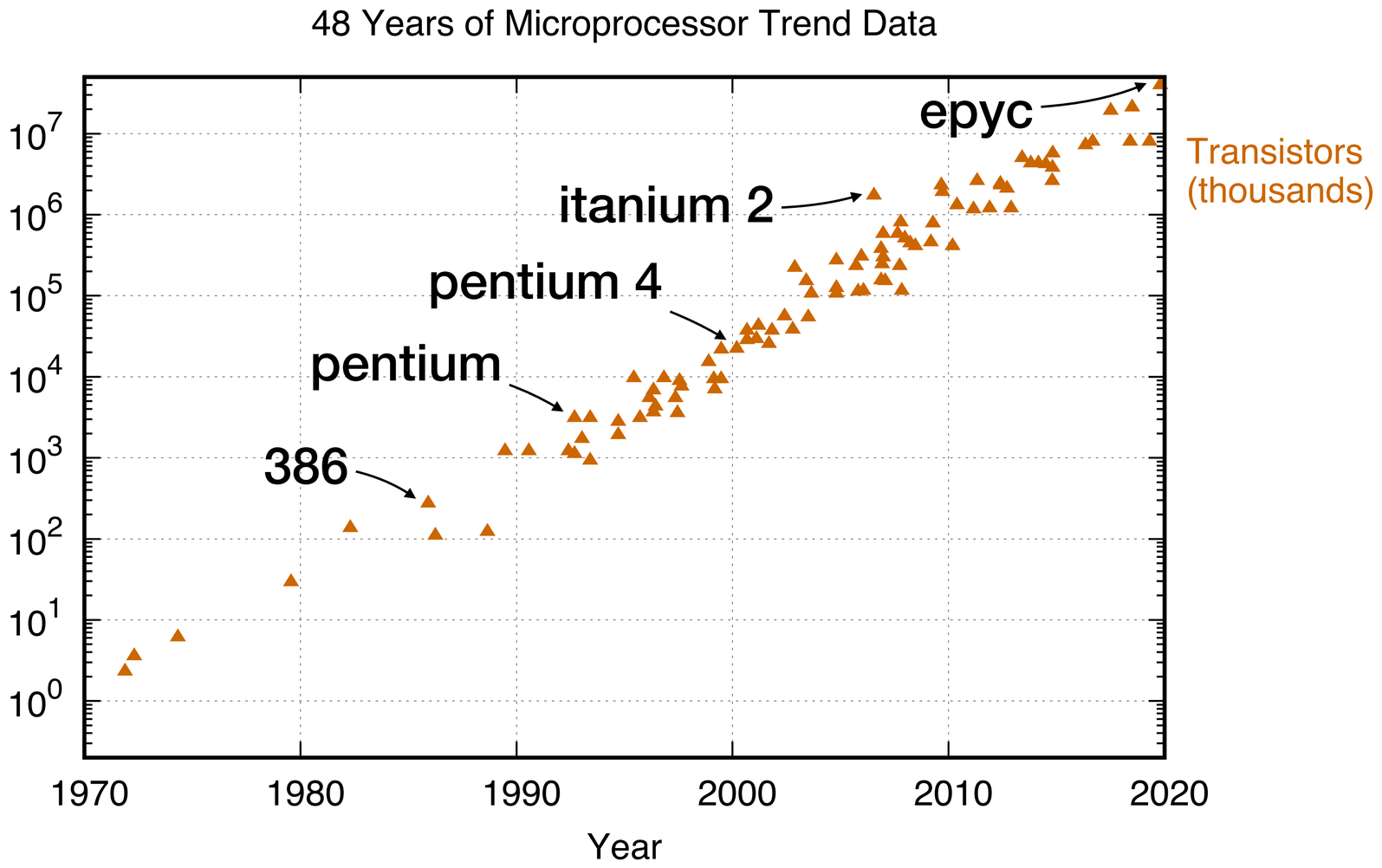
https://arstechnica.com/gadgets/2020/11/a-history-of-intel-vs-amd-desktop-performance-with-cpu-charts-galore/

# The free lunch is over

- “The free lunch is over” (2005)
  - Today virtually all processors include multiple cores/processing elements
  - This has become the primary method for increasing performance
  - This has consequences for the programmer

http://www.gotw.ca/publications/concurrency-ddj.htm

# Why?




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

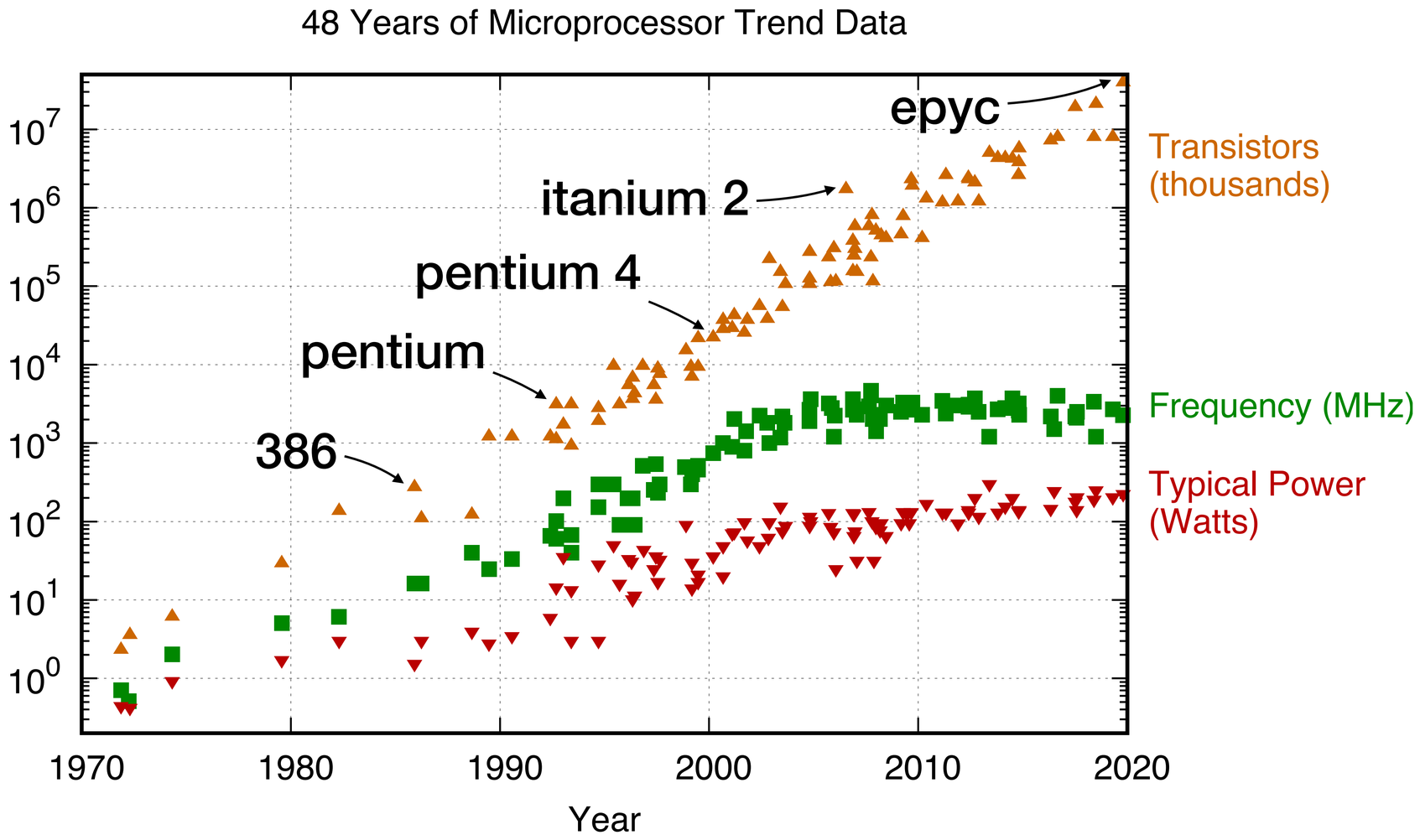
# Why?

- Moore's curve (1965)
  - Observation that the number of transistors in an integrated circuit doubles roughly every two years
  - Based on production cost and yield (success rate of production) of chips
  - Not a law in any sense of the word (don't call it that)

# Why?

- Dennard scaling: As transistors get smaller, power density remains constant
  - Voltage and current decrease at same rate as transistor size
- Smaller transistors allowed higher clock frequencies
  - As signal delays are proportional to transistor size

# Why?



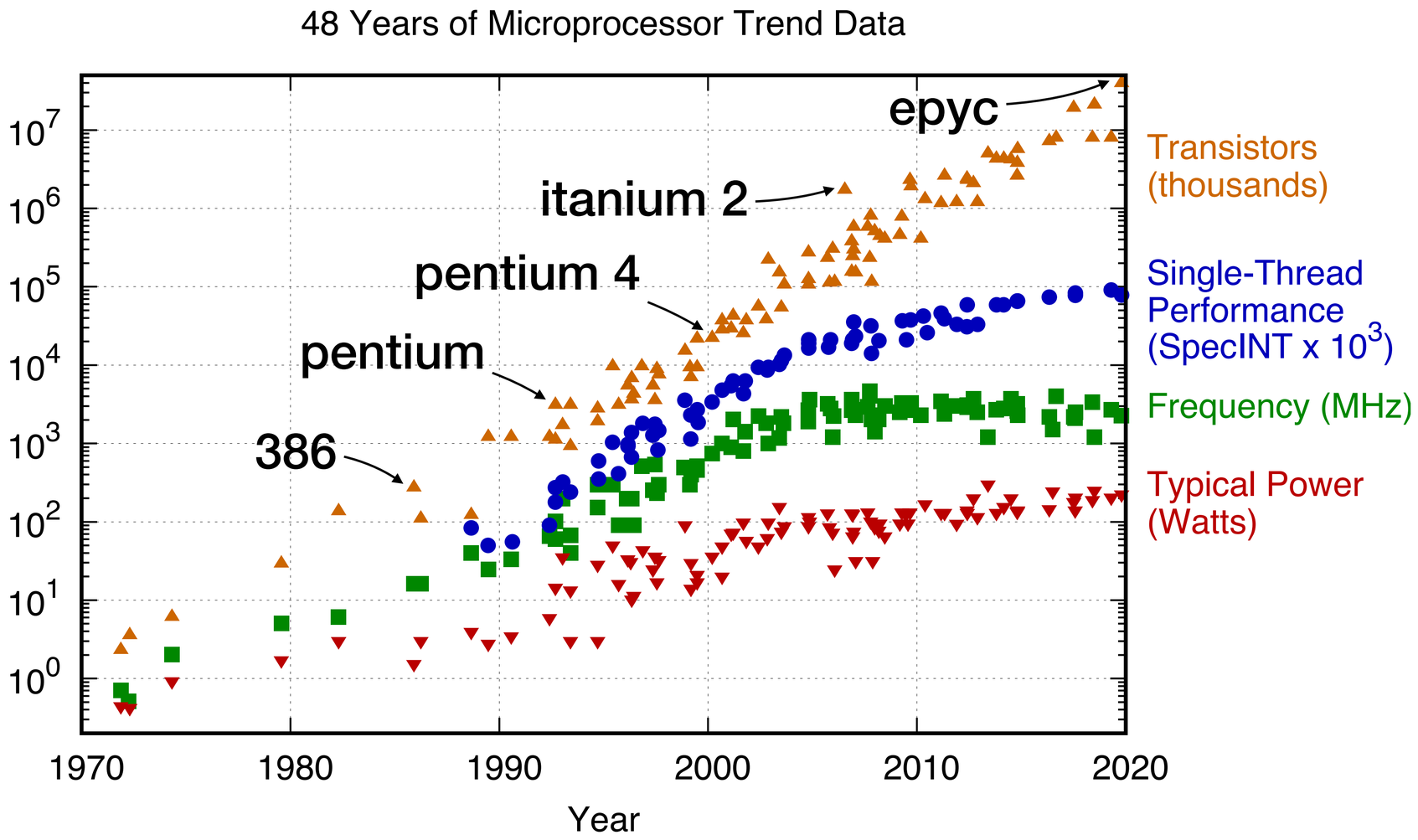

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

# Why?

- Since ~2005 Dennard scaling breaks down
  - Current leakage increased the power usage; power density wasn't constant any more
  - Consequence: can no longer improve performance through frequency scaling alone

# Why?




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

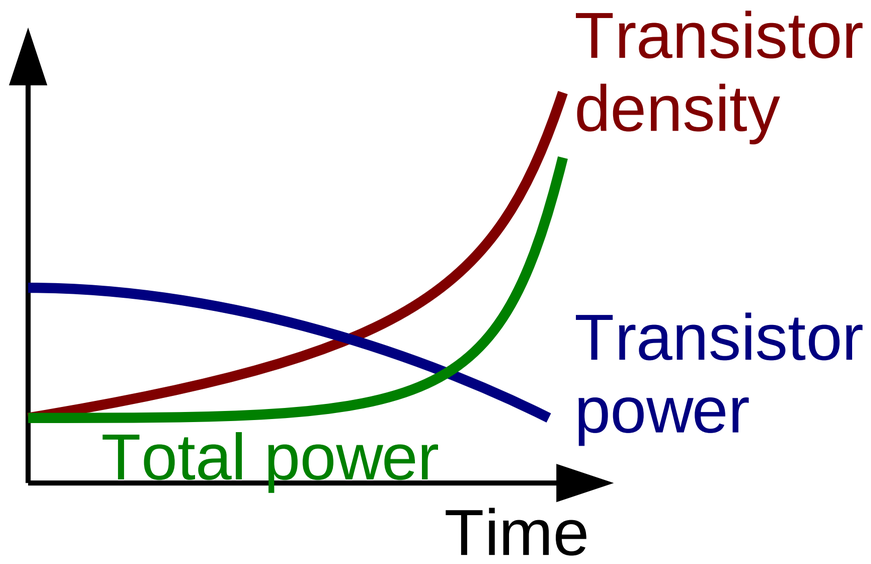
https://github.com/karlrupp/microprocessor-trend-data

# Why?

- Traditional approaches to increasing CPU performance:
  - Frequency scaling
  - Caches
  - Micro-architectural improvements
    - Out of order execution (increase utilisation of execution hardware)
    - Branch prediction (guess the outcome of control flow)
    - Speculative execution (do work before knowing if it will be needed)

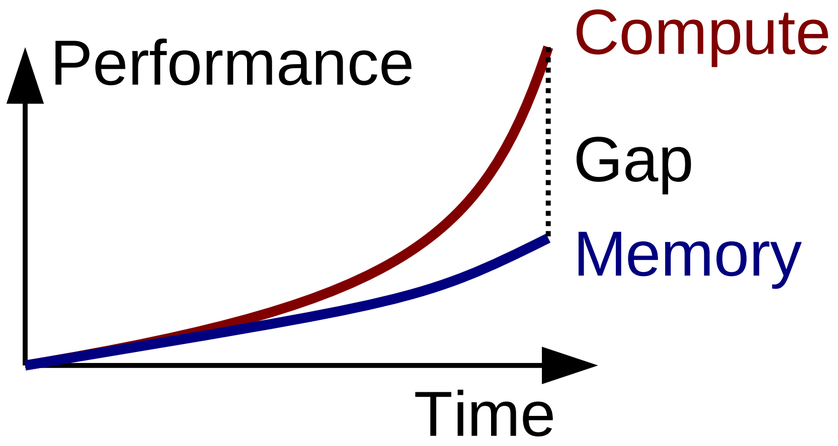
# Why?

- Frequency scaling: The Power Wall
  - Power consumption of transistors does not decrease as fast as density increases
  - Performance limited by power consumption (& dissipation)



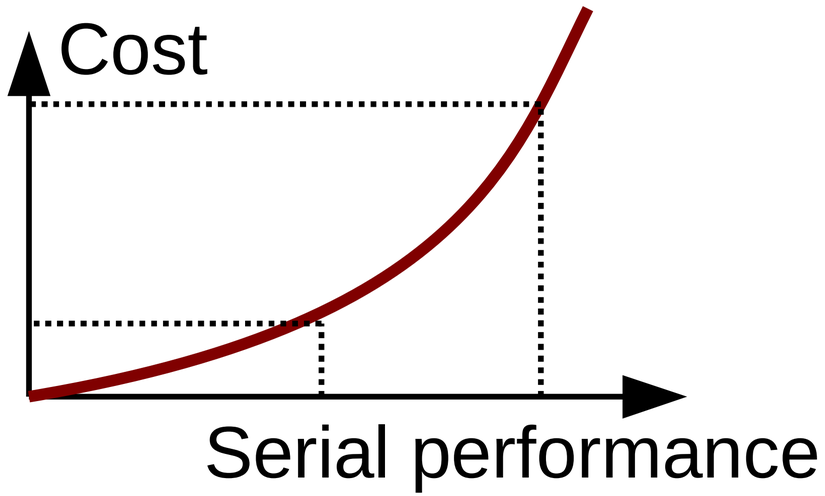
# Why?

- Caches: The Memory Wall
  - Memory speed does not increase as fast as computing speed
  - Increasingly difficult to hide memory latency

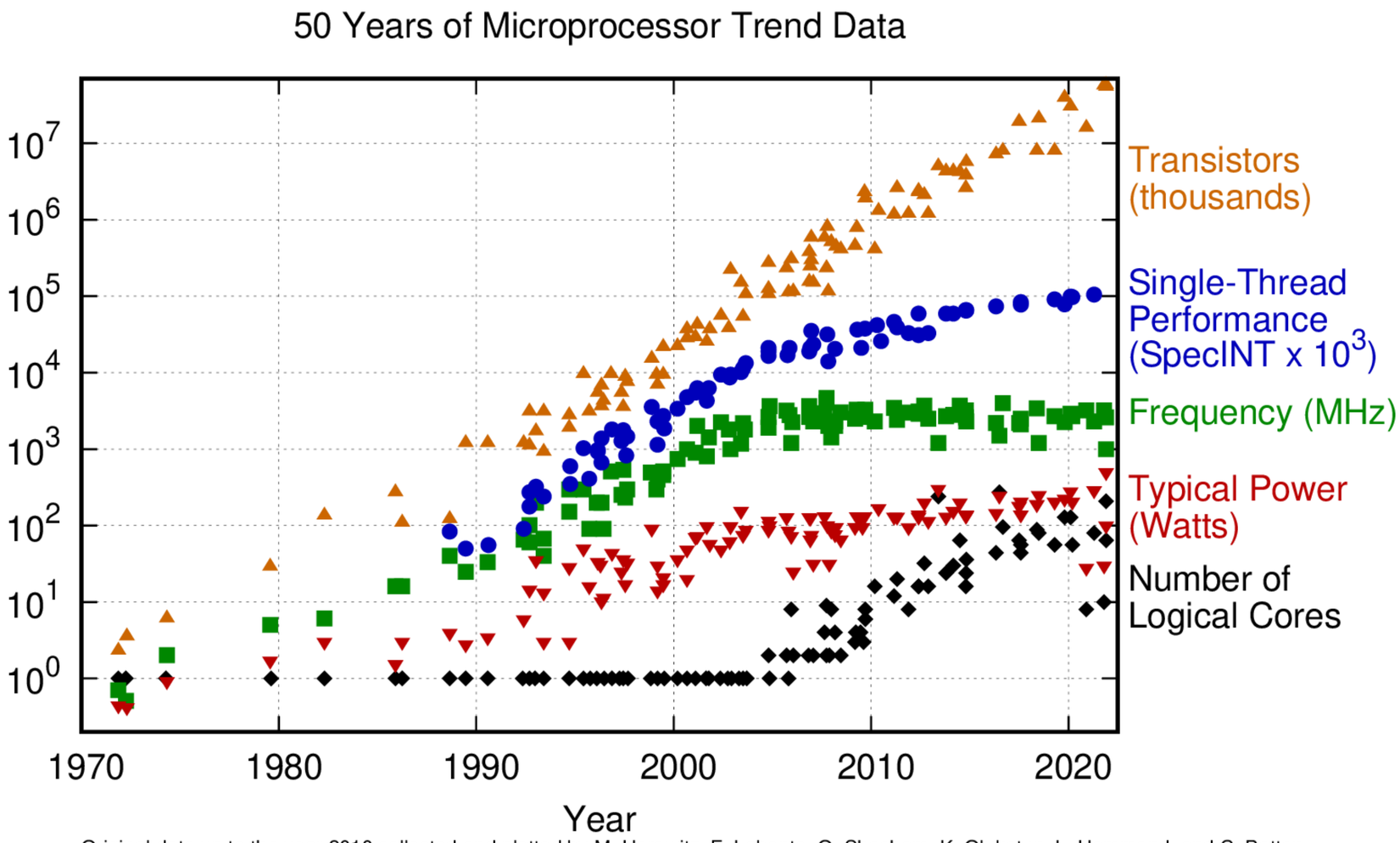


# Why?

- Microarchitecture improvements: Instruction Level Parallelism Wall
  - Law of diminishing returns
  - Pollack rule: performance $\propto \sqrt{\text{complexity}}$



# Why?




https://github.com/karlrupp/microprocessor-trend-data

# Performance walls

Limitations for single core performance:

- Power Wall
- Memory Wall
- ILP Wall

Instead, processors became faster by adding more cores

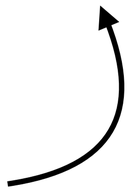
# Demo: Counting in parallel

# Counting in parallel

Say we need to count the number of persons in this room.

We can do that in parallel!

- Stand up, and remember the number 1
- Form pairs, if possible
- Add your two numbers, and now remember that number instead
- One per pair sits down
- Again form pairs of two standing persons, and repeat



# Concurrency bugs

# Therac-25

- Computer controlled medical radiation device (1982)
  - Two operating modes: a low-current electron beam; or high-energy x-rays
- Involved in at least six incidents, resulting in serious injury or death
  - A *race condition* could cause the high-power electron beam to be administered directly to the patient
  - Resulted in radiation doses 100x higher than normal
  - Additional problems related to poor software development practices

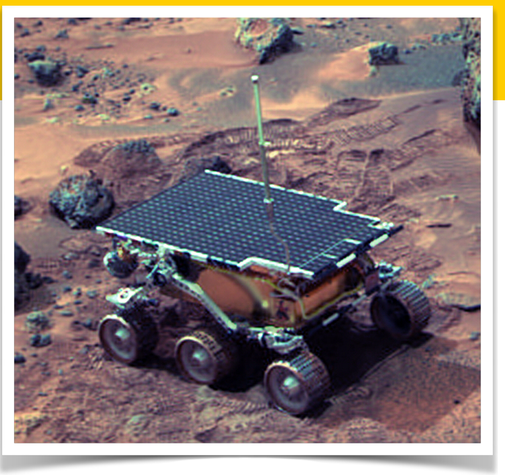
https://en.wikipedia.org/wiki/Therac-25

# Northeast blackout (2003)

- Widespread power outage throughout USA and Canada
  - Second most widespread blackout in history (at the time)
  - Affected an estimated 10M people in Ontario and 45M in 8 US states
  - A *race condition* prevented an alarm from going off
  - Operators were unaware of the need to redistribute power—a minor problem—which cascaded into complete collapse of the electrical grid

https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

# Mars Pathfinder



- Launched by NASA in 1996
- *Sojourner* became the first rover to operate outside the Earth-Moon system
- Control computer contained a *priority inversion* bug
  - Triggered under certain high loads causing a system reset
  - Successfully patched remotely

https://en.wikipedia.org/wiki/Mars_Pathfinder

# Meta AI glasses

- https://www.youtube.com/watch?v=I5_JrfvO4G8
- Live demo failed because of a race condition
  - Screen went to sleep at the exact same time a call was incoming

# Conclusion?

## Theory

## Practice

https://www.reddit.com/r/aww/comments/2oagj8/multithreaded_programming_theory_and_practice/

# Course formalities

# Topics

- Program a multithreaded application
  - Managing threads
  - Synchronise with locks, etc.
  - Software transactional memory
  - Parallelism
- Analyse parallel algorithms with work & span
- Design and implement concurrent algorithms / data structures

# Goals

- By the end of the course you should be able to:
  - Design and implement a multithreaded application
  - Understand the difference between concurrency and parallelism
  - Reason about the properties/complexity of parallel algorithms

# Homepage

- https://ics-websites.science.uu.nl/docs/vakken/b3cc/
  - Feel free to let me know if there are broken links, missing slides, etc.

# Sessions

- Lectures:
  - Tue 15:15 - 17:00
  - Thu 11:00 - 12:45
  - Recorded (at your risk, please remind us if the recording light is off!)
- Working groups:
  - Tue 13:15 - 15:00
  - Thu 9:00 - 11:45
- Participation is expected (please ask questions!)
- **Completing the working group sets is the best way to prepare for both the exams *and* the practicals**

# Course components

- Exam (50%)
  - Mid session exam: 15-12-2025 (50%)
  - Final exam: 26-01-2026 (50%)
- Practicals (50%)
  - Assignment 1: 28-11-2025 (individual) (20%)
  - Assignment 2: 19-12-2025 (in pairs) (40%)
  - Assignment 3: 23-01-2026 (in pairs) (40%)
  - If you want to reuse practical grades from last year, send me an email (i.g.dewolff@uu.nl)

# Haskell

- General concepts for concurrency are similar across programming languages,
- but some provide better support for concurrency.
- Haskell:
  - Separates pure and impure code, e.g. thread-local work and actual concurrency work
  - Has good support for building and studying abstractions

# Haskell

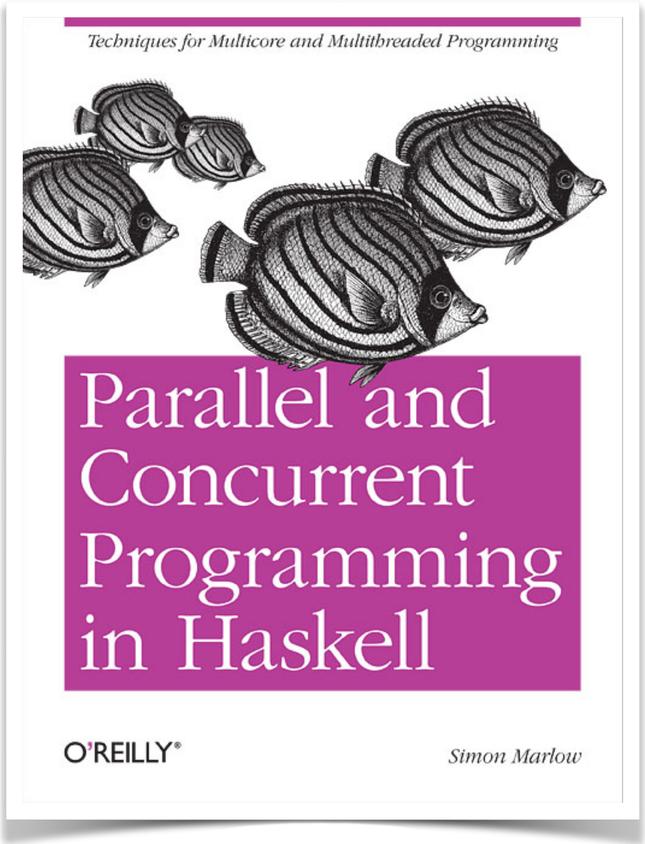
- Haskell allows us to study both low-level and high-level aspects of concurrency.
  - P1: basics of forking threads, locks and communication
  - P3: design and implementation of a quicksort-like algorithm for GPUs
- Thursday (9:00) during working groups: Haskell recap, focusing on the important parts for this course
  - Very boring if you just passed Functional Programming
  - Who is interested?
  - BBG 201
  - Regular working groups in other rooms (209, 214)
- Sometimes we use C in examples in the slides, to focus on low-level details

# Rust

- We study concepts and apply them in Haskell
  - They also apply to other languages including Rust
- Like Haskell, Rust uses types to make concurrency safe
- Object is mutable only if you are the single owner, or via concurrency primitives
  - Prevents race conditions
- Allowed as alternative for Haskell in P1, maybe in P2
  - At your own risk
  - Only advised if you already know Rust

# Resources

- Parallel and Concurrent Programming in Haskell
  - https://simonmar.github.io/pages/pcph.html
- Many more on the website
  - https://ics-websites.science.uu.nl/docs/vakken/b3cc/resources.html



# Practicals

- P1: already available, you can start with a sequential (single-threaded) implementation
  - https://ics-websites.science.uu.nl/docs/vakken/b3cc/assessment.html

# Software installation

- A recent version of GHC (9.6.7)
  - Instructions on our website

# Next time...

Thursday 9:00

- Haskell recap (BBG 201)
- Or: Working groups: start with P1 (BBG 209, 214)

Thursday 11:00

- Lecture: Threads 1