# Programming Parallel Computers

Jukka Suomela · Aalto University · ppc.cs.aalto.fi

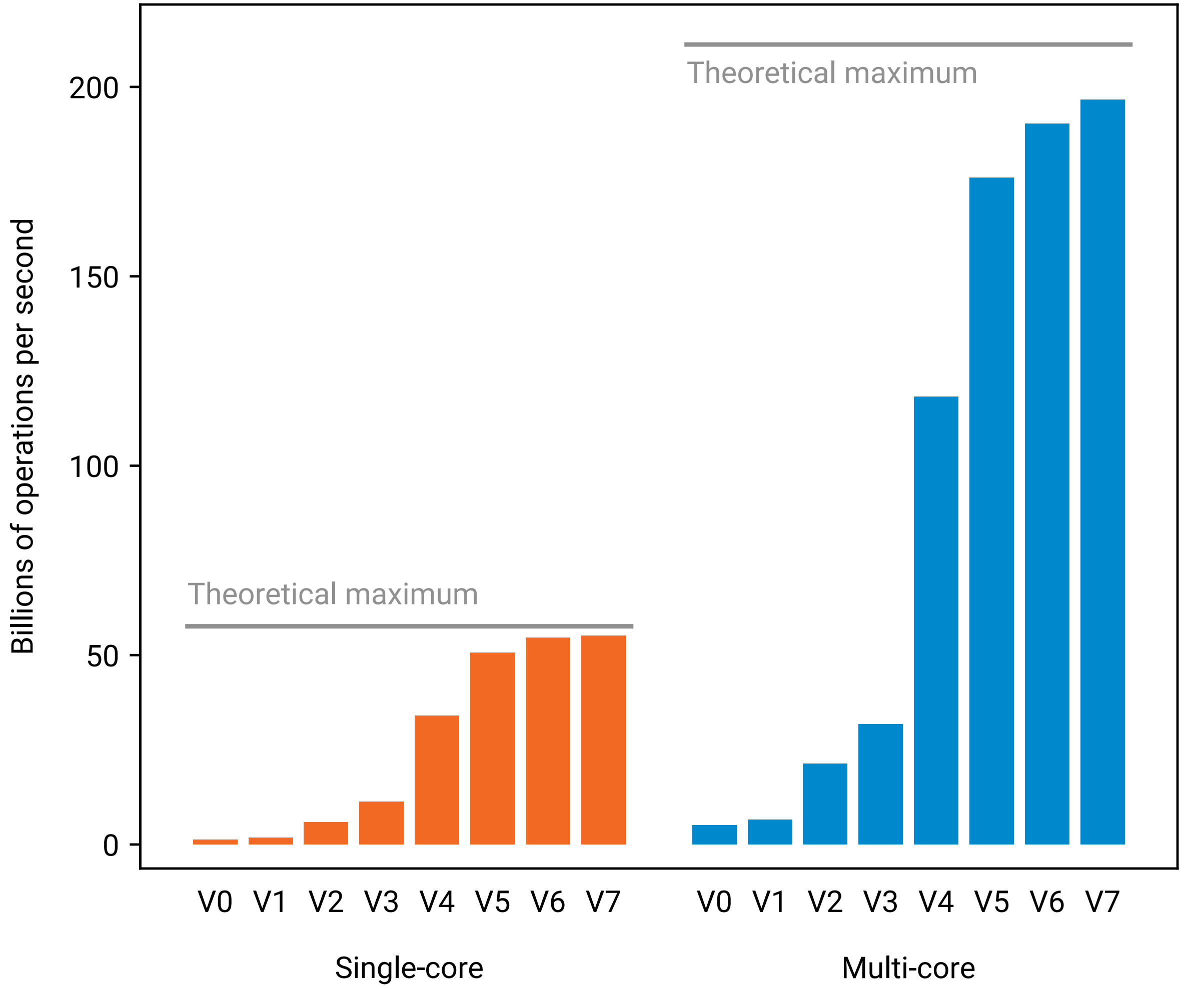
## Part 1A:
## What is this course about? · Why parallelism?

# Performance, in practice!

- **Main goal:** learning to write code that ***runs very fast*** on modern computers
- **The only way to get there:** write programs that do ***lots of independent things in parallel***

# 150-fold speedups?

On a single computer, with a 4-core processor?



# Performance, in practice!

- **“Solve this problem, using this computer, for this input, as fast as possible”**
  - you will write a program
  - we will measure how long it takes to run
- **Grading:** ***correct solution & good performance***

# Performance, in practice!

- We will focus on the ***good parts***
  - getting the job done, with minimal effort, in practice
  - tools that are **as simple as possible** — without sacrificing performance
- Emphasis on ***understanding***
  - demystifying hardware
  - learning to **predict** performance
- This is ***engineering***
  - based on understanding, math, science, and good practices
  - but requires **creativity** and **experimentation**

# Prerequisites

- **Necessary:**
  - good understanding of computer programming, algorithms and data structures
  - ***working knowledge of C or C++***
- **Not needed:**
  - knowledge of parallel programming

# Why parallelism?

The only way to get good performance nowadays

# Modern computers are massively parallel

- Multiple ***CPU cores***
- Multiple ***execution units*** per core
- Execution units can perform ***vector operations***
- Execution units are ***pipelined***
  - no need to wait for one operation to finish before starting the next one
- And then there is a ***massively parallel GPU***…
  - we can do general-purpose computation on the graphics processor

# All new performance comes from parallelism

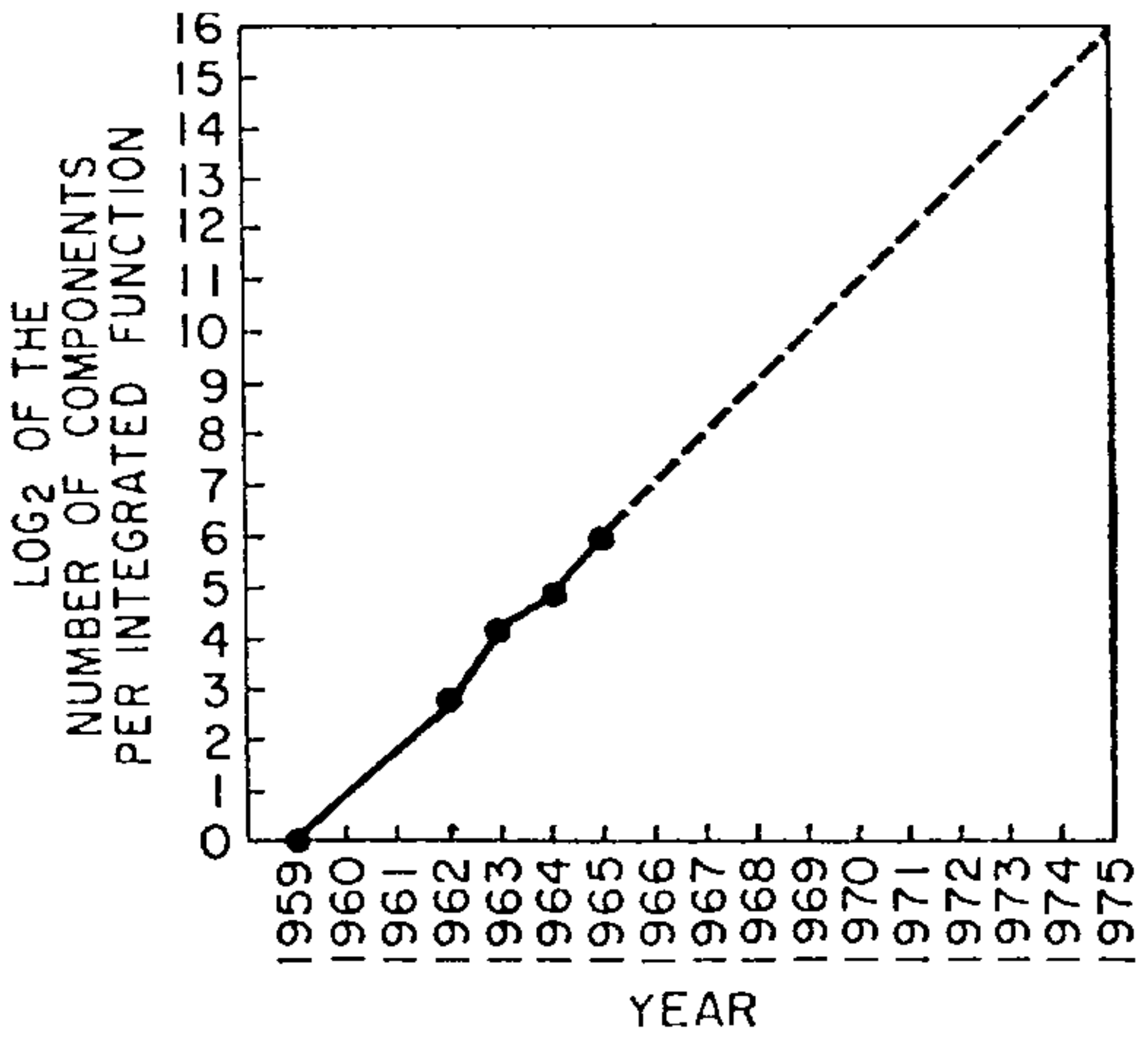
- Sequential performance stopped improving around 2000
- ***All new performance comes from parallelism***
- New code is needed
- Traditional C++ code might use ***less than 1%*** of the capabilities of your computer

Gordon E. Moore: "Cramming more components onto integrated circuits", Electronics Magazine 1965 (reprinted in Proc. IEEE, vol. 86, issue 1, 1998)

# Moore's law

1965 prediction:

***number of transistors in integrated circuits grows exponentially***



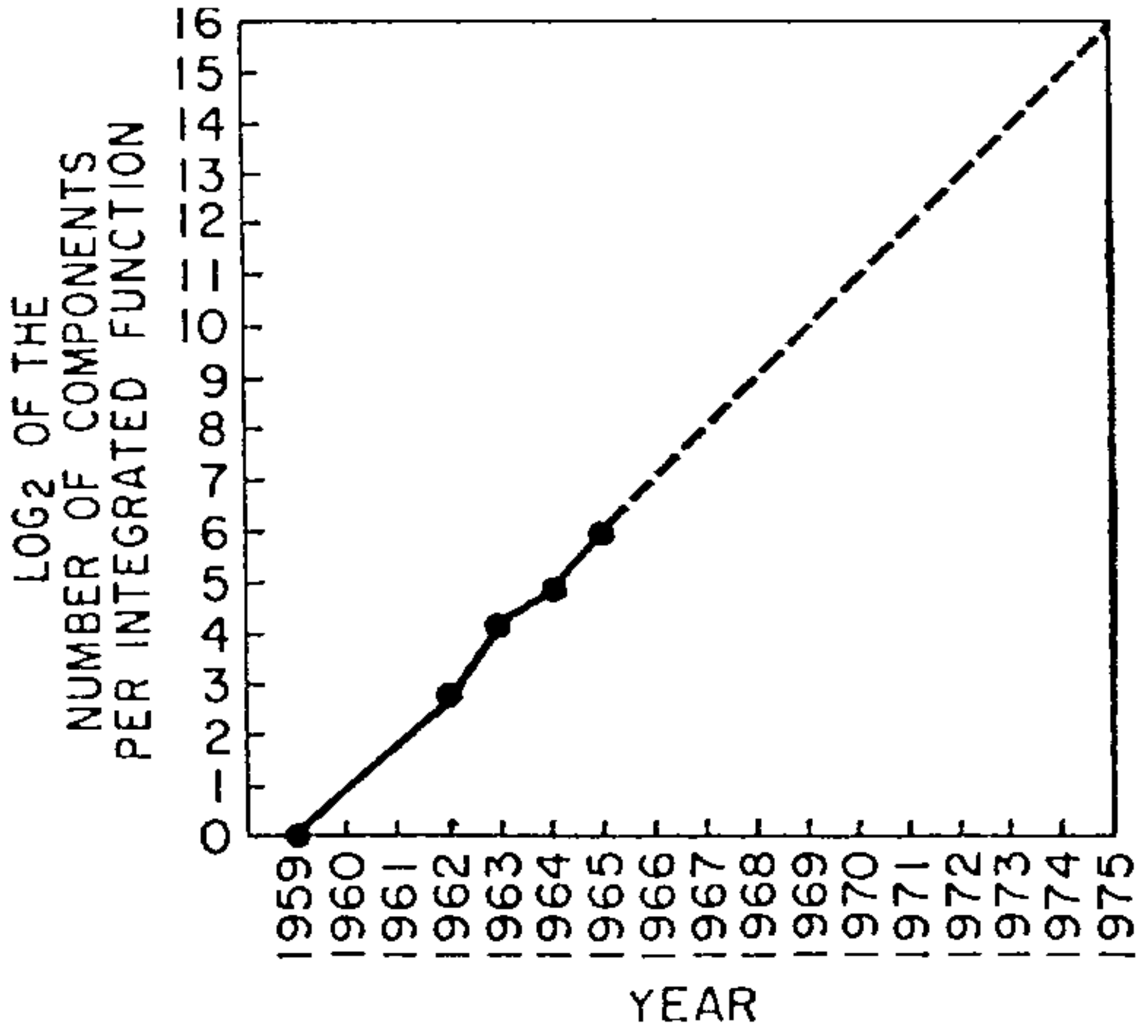
Gordon E. Moore: “Cramming more components onto integrated circuits”, Electronics Magazine 1965 (reprinted in Proc. IEEE, vol. 86, issue 1, 1998)

# Moore’s law

1965 prediction:

***number of transistors in integrated circuits grows exponentially***

2020: yes, still true!



# Moore's law

Still going strong!

But something
has changed...

| Year | Transistors | CPU model |
|---|---:|---|
| 1975 | 3 000 | 6502 |
| 1979 | 30 000 | 8088 |
| 1985 | 300 000 | 386 |
| 1989 | 1 000 000 | 486 |
| 1995 | 6 000 000 | Pentium Pro |
| 2000 | 40 000 000 | Pentium 4 |
| 2005 | 100 000 000 | 2-core Pentium D |
| 2008 | 700 000 000 | 8-core Nehalem |
| 2014 | 6 000 000 000 | 18-core Haswell |
| 2017 | 20 000 000 000 | 32-core AMD Epyc |
| 2019 | 40 000 000 000 | 64-core AMD Rome |

***Sequential*** performance improving (1975–2000)

***Parallel*** performance improving (2000–2019)

| Year | Transistors | CPU model |
|---|---:|---|
| 1975 | 3 000 | 6502 |
| 1979 | 30 000 | 8088 |
| 1985 | 300 000 | 386 |
| 1989 | 1 000 000 | 486 |
| 1995 | 6 000 000 | Pentium Pro |
| 2000 | 40 000 000 | Pentium 4 |
| 2005 | 100 000 000 | 2-core Pentium D |
| 2008 | 700 000 000 | 8-core Nehalem |
| 2014 | 6 000 000 000 | 18-core Haswell |
| 2017 | 20 000 000 000 | 32-core AMD Epyc |
| 2019 | 40 000 000 000 | 64-core AMD Rome |

It takes ***less time*** to complete one operation

We can do ***several*** operations in parallel

| Year | Transistors | CPU model |
|---|---:|---|
| 1975 | 3 000 | 6502 |
| 1979 | 30 000 | 8088 |
| 1985 | 300 000 | 386 |
| 1989 | 1 000 000 | 486 |
| 1995 | 6 000 000 | Pentium Pro |
| 2000 | 40 000 000 | Pentium 4 |
| 2005 | 100 000 000 | 2-core Pentium D |
| 2008 | 700 000 000 | 8-core Nehalem |
| 2014 | 6 000 000 000 | 18-core Haswell |
| 2017 | 20 000 000 000 | 32-core AMD Epyc |
| 2019 | 40 000 000 000 | 64-core AMD Rome |

Lower *latency*

Higher *throughput*

| Year | Transistors | CPU model |
|---|---:|---|
| 1975 | 3 000 | 6502 |
| 1979 | 30 000 | 8088 |
| 1985 | 300 000 | 386 |
| 1989 | 1 000 000 | 486 |
| 1995 | 6 000 000 | Pentium Pro |
| 2000 | 40 000 000 | Pentium 4 |
| 2005 | 100 000 000 | 2-core Pentium D |
| 2008 | 700 000 000 | 8-core Nehalem |
| 2014 | 6 000 000 000 | 18-core Haswell |
| 2017 | 20 000 000 000 | 32-core AMD Epyc |
| 2019 | 40 000 000 000 | 64-core AMD Rome |

# Latency vs. throughput

- ***Latency:*** time to perform operation, from start to finish
- ***Throughput:*** how many operations are completed per time unit
    - in the long run
- **Example:** MSc degrees at Aalto
    - latency: ≈ ***2 years***
    - throughput: ≈ ***1960 degrees/year***
    - Aalto is massively parallel!
    - education in a sequential manner would yield only ***0.5 degrees/year***

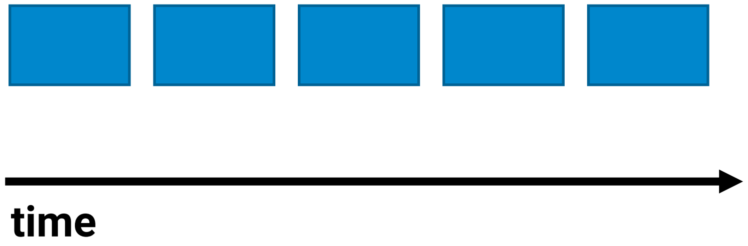
# Latency vs. throughput

- ***Latency:*** time to perform operation, from start to finish
- ***Throughput:*** how many operations are completed per time unit
  - in the long run

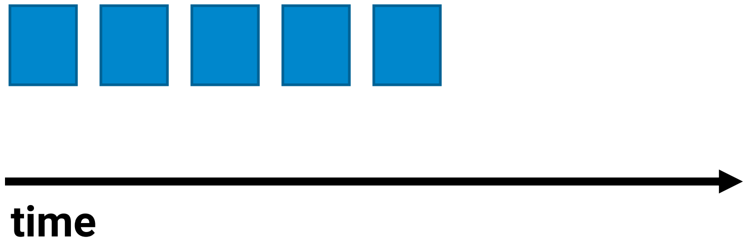
- **Formerly:** lower latency → higher throughput
- **Nowadays:** more parallelism → higher throughput

# Progress used to look like this

## High latency

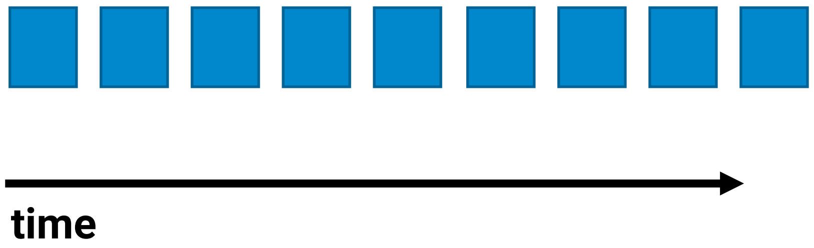



## Low latency

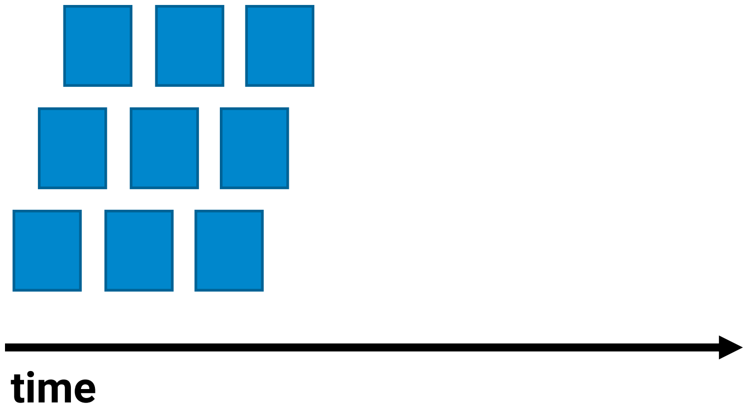


# New kind of progress

## No parallelism




## Lots of parallelism



# An example

- Typical modern desktop CPU: **Intel Core i5-6500** (4 cores)
- Operation: **single-precision floating-point multiplication**
- Latency: ***4 clock cycles***
- Sequential throughput: ***0.25 operations / cycle***
- Parallel throughput: ***64 operations / cycle***
  - we can have 256 operations simultaneously on the fly!
- ***200 billion*** operations per second (clock speed ≈ 3.3 GHz)

# An example

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

# An example

**Parallel computing: much more than just multithreading!**

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

# An example

**Not only for high-end servers: your laptop can do all of this!**

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors