Prime Minister of Italy (1867-1869)

The machine can be brought into play so as to give several results at the same time...

Task-Parallel Computing

algorithms to distribute tasks across processors

Moore/Dennard

Trends

Work: $T_1(n)$

Analysis

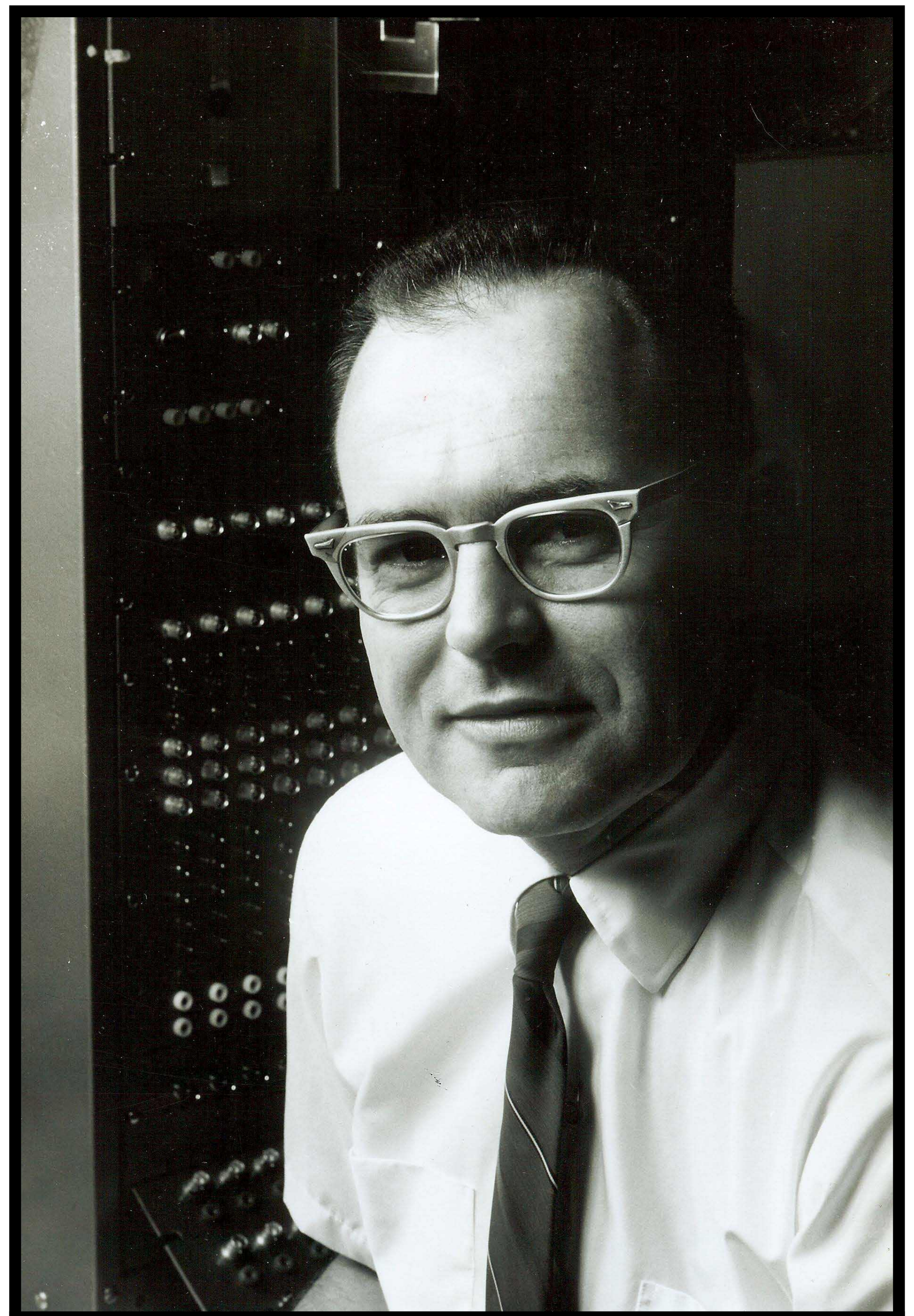
```
def par_fib(n):
  ...
```

Code

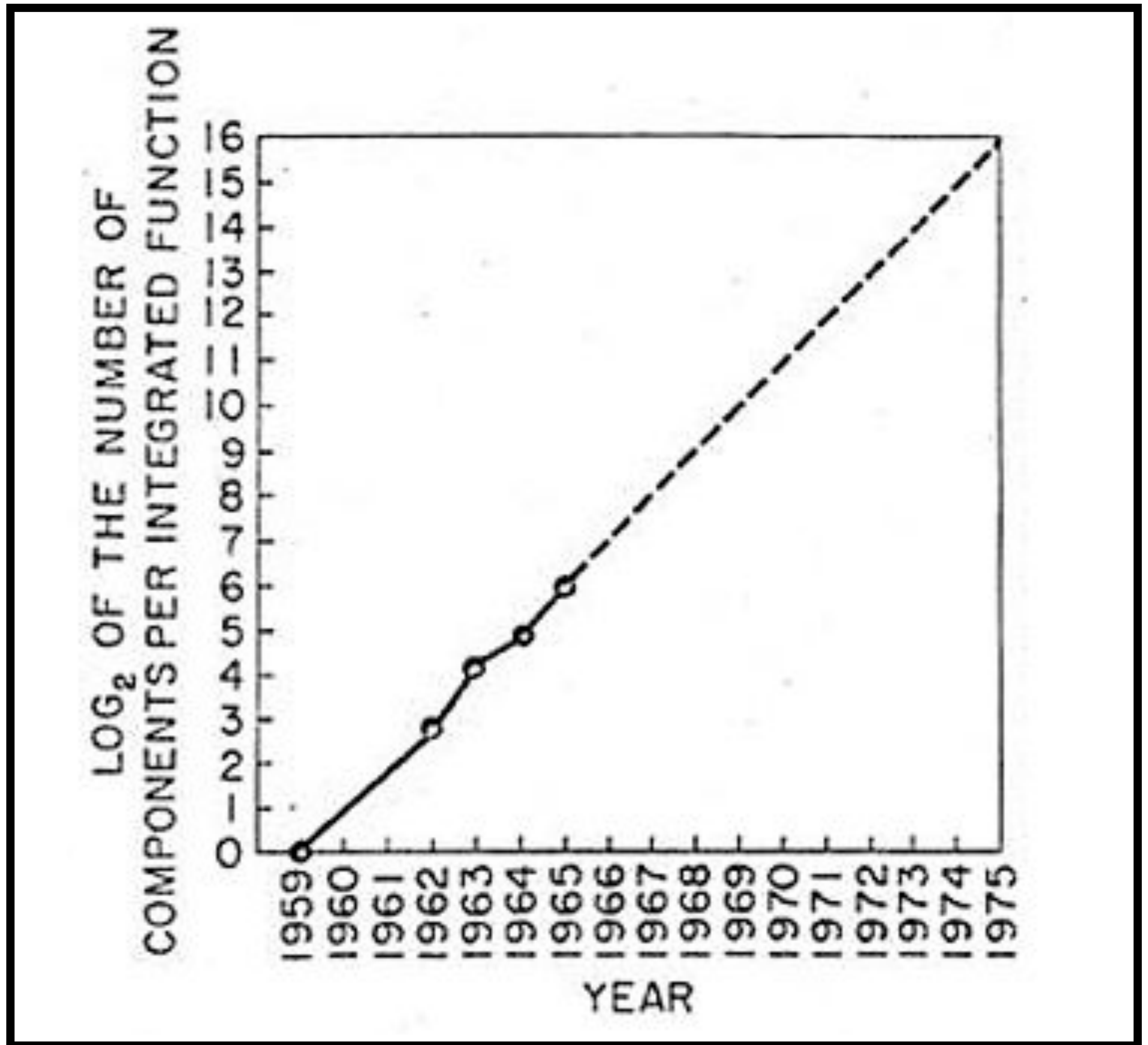
Moore's Law

Cramming more components onto *integrated circuits* (1965)

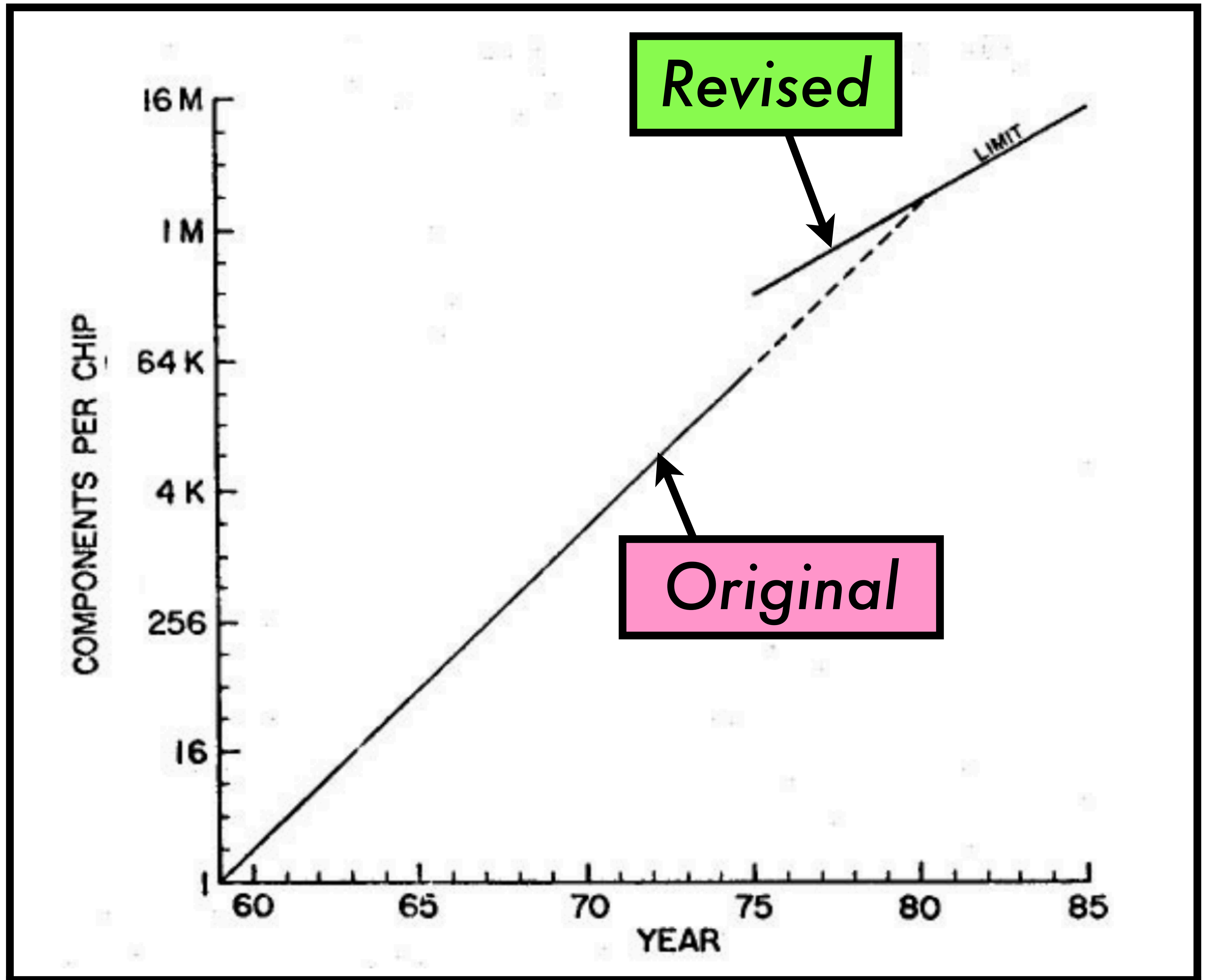
Forecast: #transistors/chip when cost-per-transistor is lowest



"ICs will lead to such wonders as home computers"



1965 *forecast: double every year for minimum component cost*

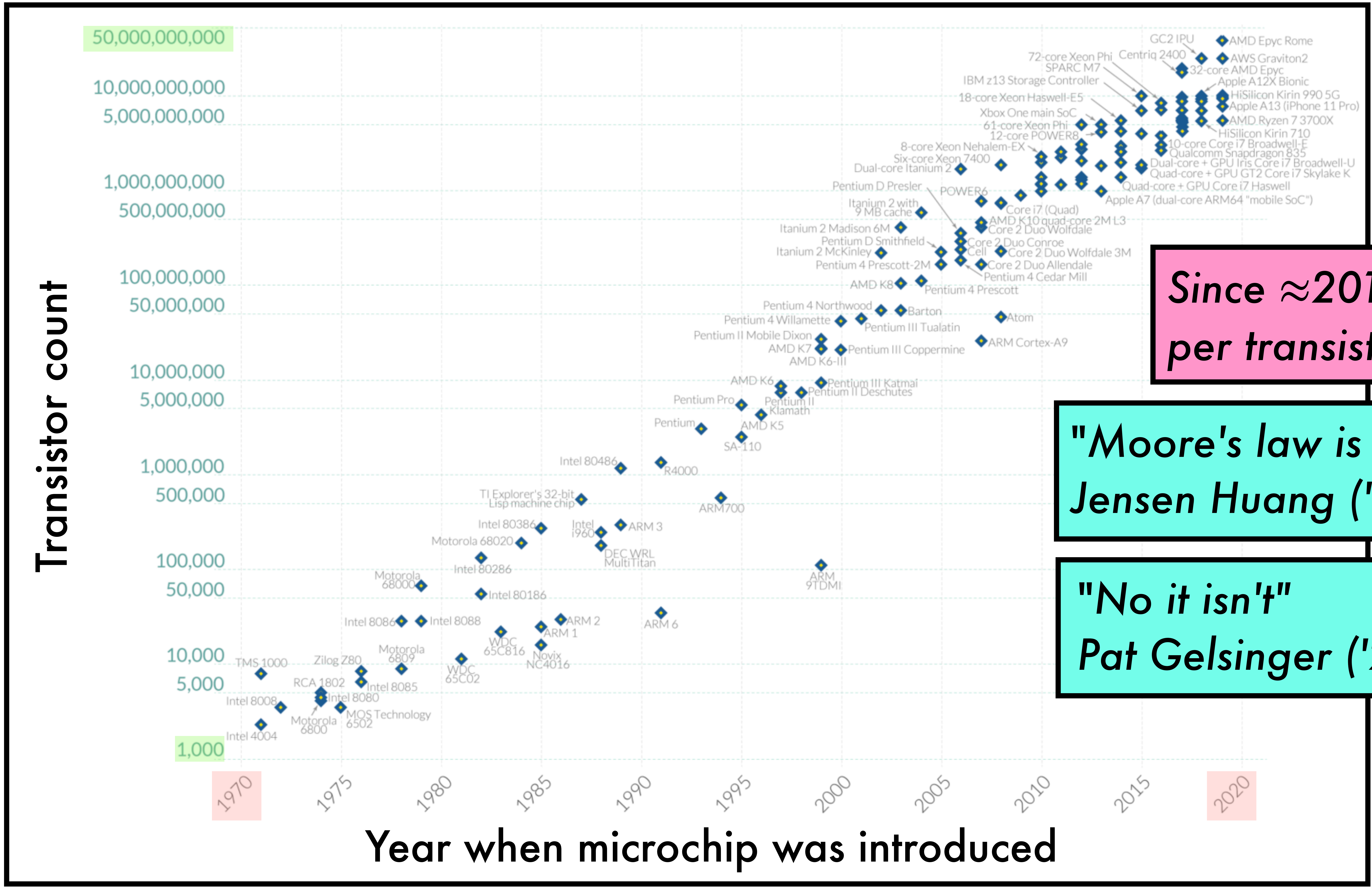


1975 *forecast: double every two years*

Progress in Digital Integrated Electronics (1975)

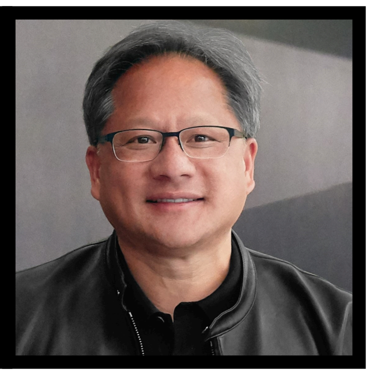
References/Notes/Image credits:
(G. Moore image) https://www.businesswire.com/news/home/20161102005463/en/Gordon-and-Betty-Moore-Foundation-Announces-Inaugural-Moore-Inventor-Fellows
(Projection figure from 1965) G. E. Moore, "Cramming more components onto integrated circuits" (1965)
(Projection figure from 1975) G. E. Moore, "Progress in digital integrated electronics", Electron devices meeting (1975)

Moore's Law - Historical Data



Since ≈2010, cost decline per transistor is slowing

"Moore's law is dead"
Jensen Huang ('17, '22)

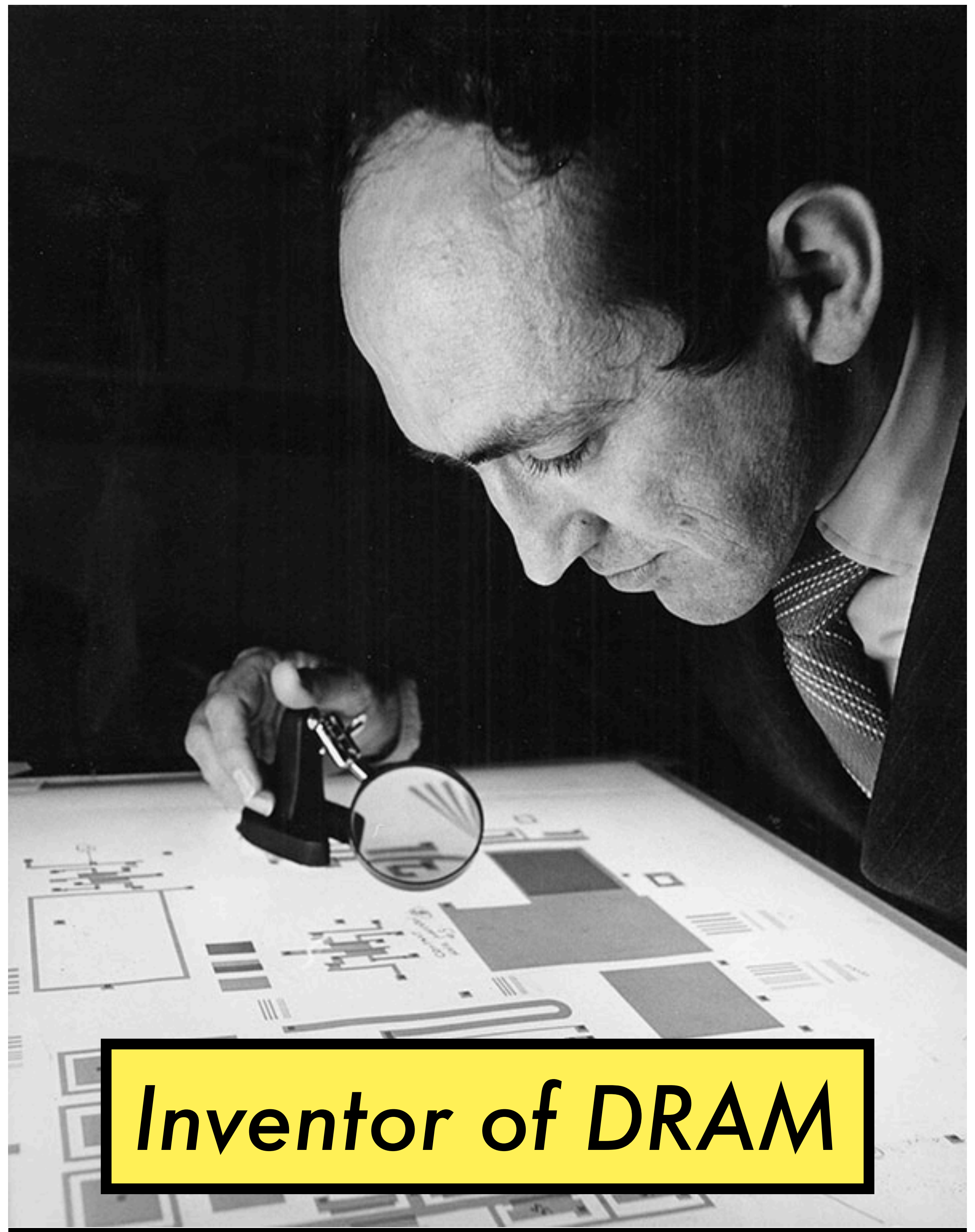


"No it isn't"
Pat Gelsinger ('22)

Image credits/references:
(Moore's law) H. Ritchie and M. Roser, https://ourworldindata.org/technological-change (last-accessed 2023-01)
(Rising costs) https://www.fabricatedknowledge.com/p/the-rising-tide-of-semiconductor

K. Flamm, "Measuring Moore's law: evidence from price, cost, and quality indexes", University of Chicago Press (2019)
(Jensen Huang image) https://nvidianews.nvidia.com/bios/jensen-huang
(Pat Gelsinger image) https://www.concordia.net/community/patrick-gelsinger/

Dennard Scaling



Inventor of DRAM

As transistors shrink, power density remains constant

Dennard's model of MOSFET scaling

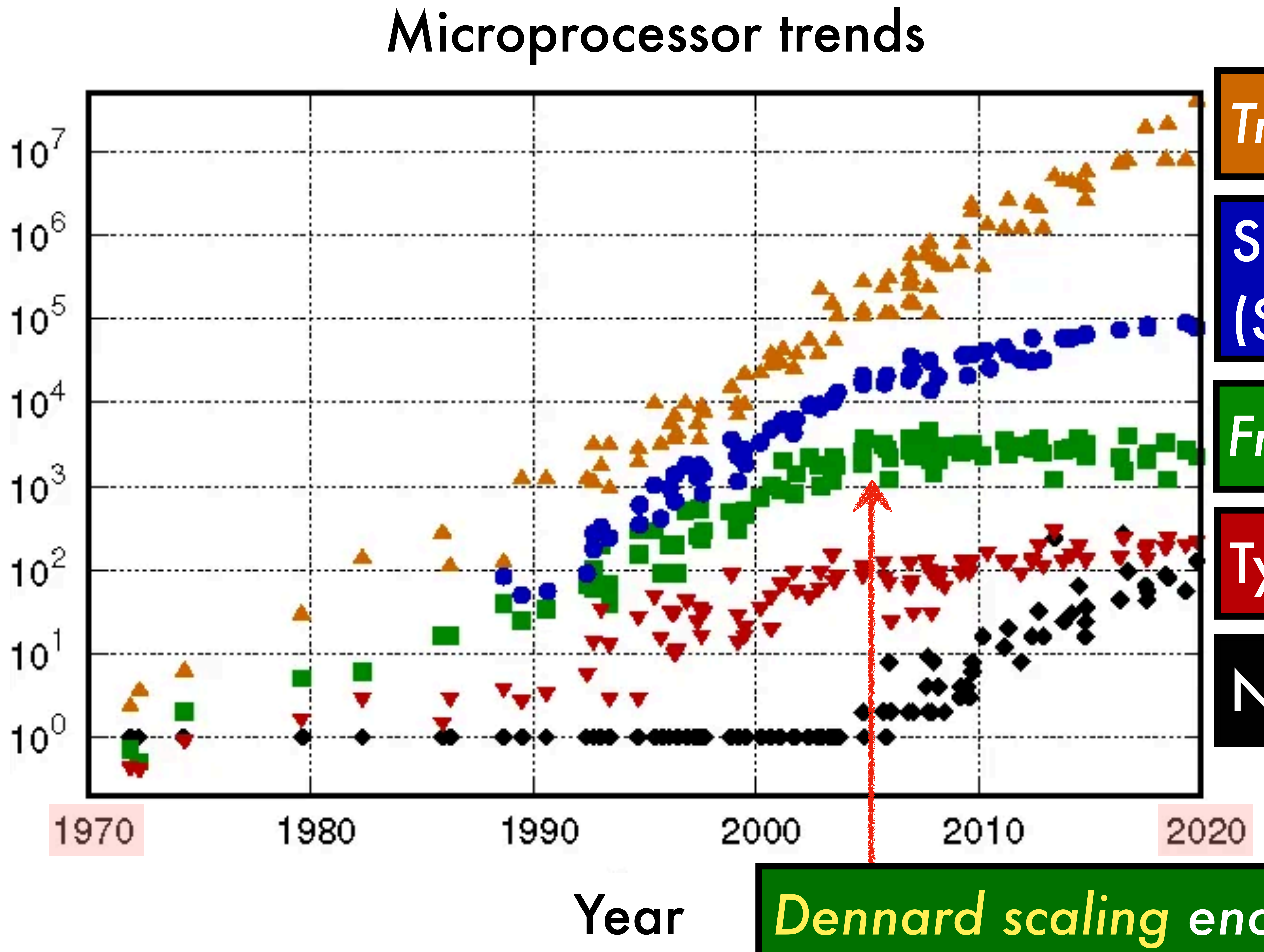
With each generation, transistor dimensions shrink by 30%

Device area shrinks by 50%	area = length × width
Capacitance shrinks by 30%	capacitance ∝ area/distance
Voltage is reduced by 30%	electric field ∝ voltage/distance
Circuit delay reduces by 30%	due to reduced gate delays
Frequency increases by 40%	frequency = 1/time period
Active power reduces by 50%	active power ∝ CV^2f

Each generation: double # transistors | same power | 40% faster

References/Notes/Image credits:
(image source) https://www.ibm.com/blogs/think/2019/11/ibms-robert-h-dennard-and-the-chip-that-changed-the-world/
R. H. Dennard et al., "Design of ion-implanted MOSFET's with very small physical dimensions", *IEEE Journal of solid-state circuits* (1974)
https://en.wikipedia.org/wiki/Dennard_scaling
S. Borkar et al., "The future of microprocessors", *Communications of the ACM* (2011)

The End Of Dennard Scaling



Transistors (thousands)

Single-thread performance (SpecINT $\times 10^3$)

Frequency (MHz)

Typical power (Watts)

Number of logical cores

Does this keep the scaling going?

Dennard scaling ends ≈2005

Dennard's model ignored "leakage current"

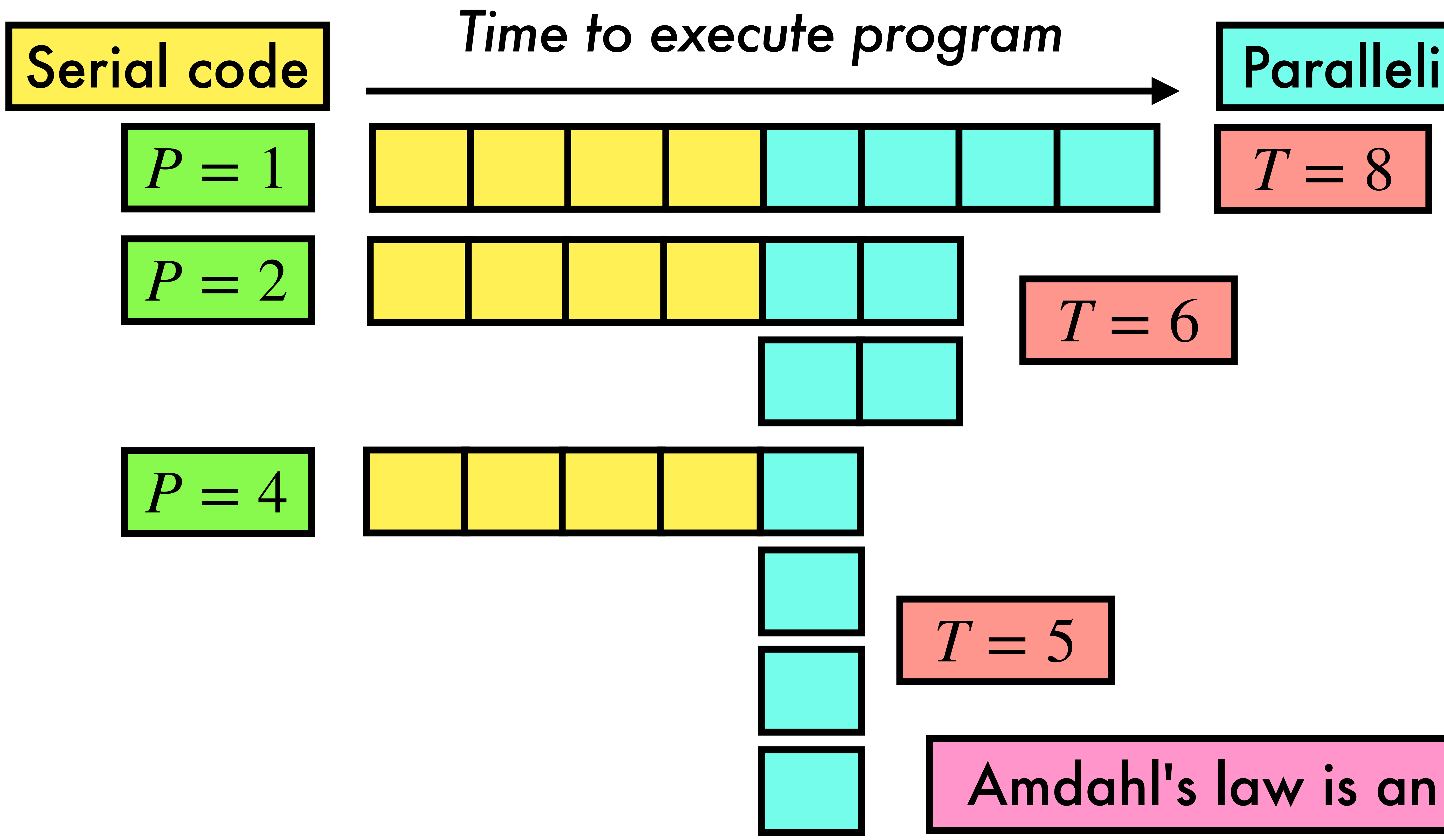
References/Image credits:
(Image credit) https://github.com/karlrupp/microprocessor-trend-data
Data up to 2010 collected/plotted by M. Horowitz et al., data from 2010-2019 by K, Rupp
M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper", *IEEE Solid-State Circuits Society Newsletter* (2007)

Amdahl's Law 💔

How far can parallel computation take us?

Many programs contain code that cannot be parallelised

Notation: Processors, P | Time, T



Amdahl's law

Let $\alpha \in [0,1]$ be the fraction of code that can be parallelised

$$T_{new} = T_{orig} \times \left((1 - \alpha) + \frac{\alpha}{P} \right)$$

$P = 4$ | $T_{new} = 8$ | $\alpha = 0.5$

$$T_{new} = 8 \cdot (0.5 + 0.125) = 5$$

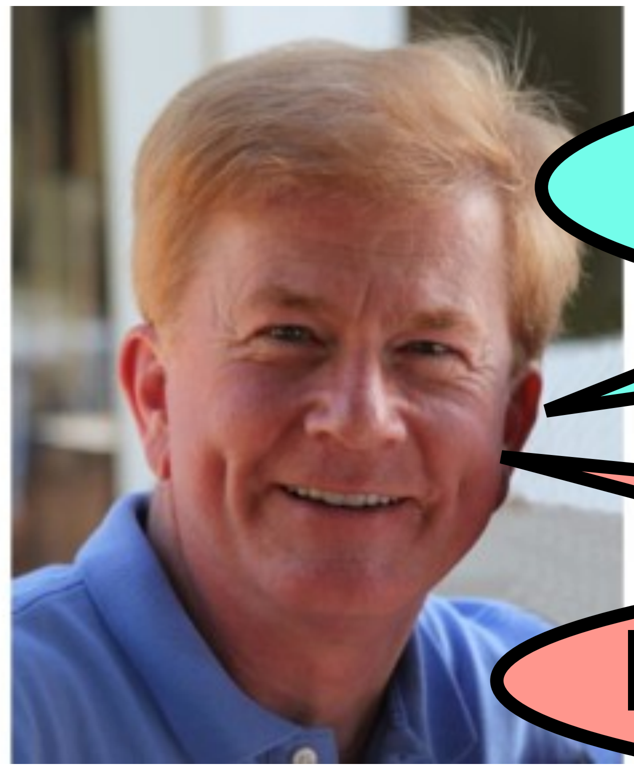
Amdahl's law is an (often loose) upper bound on parallelism

Reference:
G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", Proceedings of the Spring Joint Computer Conference (1967)
J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach", Chap. 1 (2017)

Gustafson's Law

There is a key idea underpinning the interpretation of Amdahl's law:

Problem size stays fixed even as more processors become available



This is virtually never the case!

In practice, the problem scales with # processors

Given more processing, the problem expands to use it

Better: assume runtime (not problem size) is constant

Parkinson's law (1955)

"Work expands so as to fill the time available for its completion." - Cyril Parkinson

Often, the parallel part of the program scales with problem size

P Processors

speedup $= 1 + \alpha \cdot (P - 1)$

A linear scaling law!

Key difference to Amdahl - we assume that we will scale up the parallel part of the problem

Which "law" is a better fit depends on the domain

Booting your Operating System | Amdahl

Train model on 1000s of GPUs | Gustafson

Boot stronger machine

Reference:
https://en.wikipedia.org/wiki/Gustafson's_law
J. Gustafson, "Reevaluating Amdahl's law", *Comms. of the ACM* (1988)
https://en.wikipedia.org/wiki/Parkinson's_law

Memory Models For Parallel Computing

Since the end of Dennard scaling (≈2005), multicore computers have become pervasive

Important design choice with multiple cores/multiple multiprocessors: how to organise memory

Shared Memory	Distributed Memory
Key idea: any core can directly access any location in a *shared address space*	Key idea: each core sends message (over network) to access memory belonging to another core
Typical hardware: phones laptops	Typical hardware: compute clusters
"Hard to build, easy to program"	*"Easy to build, hard to program"*

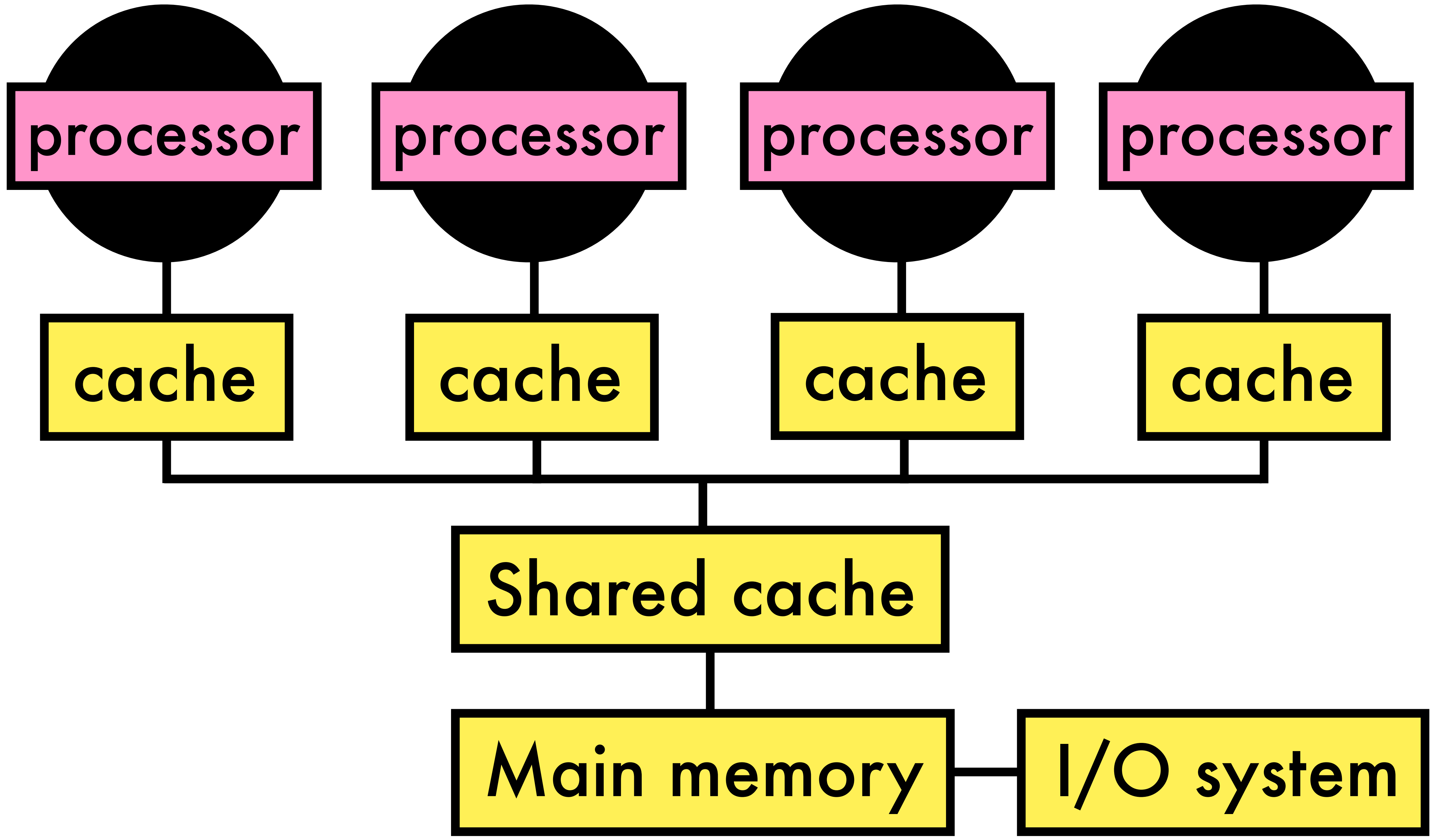
References:
J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach", Chap. 5 (2017)

Shared Memory Variants

Symmetric Multiprocessing (SMP)

Key idea: uniform access time to all memory

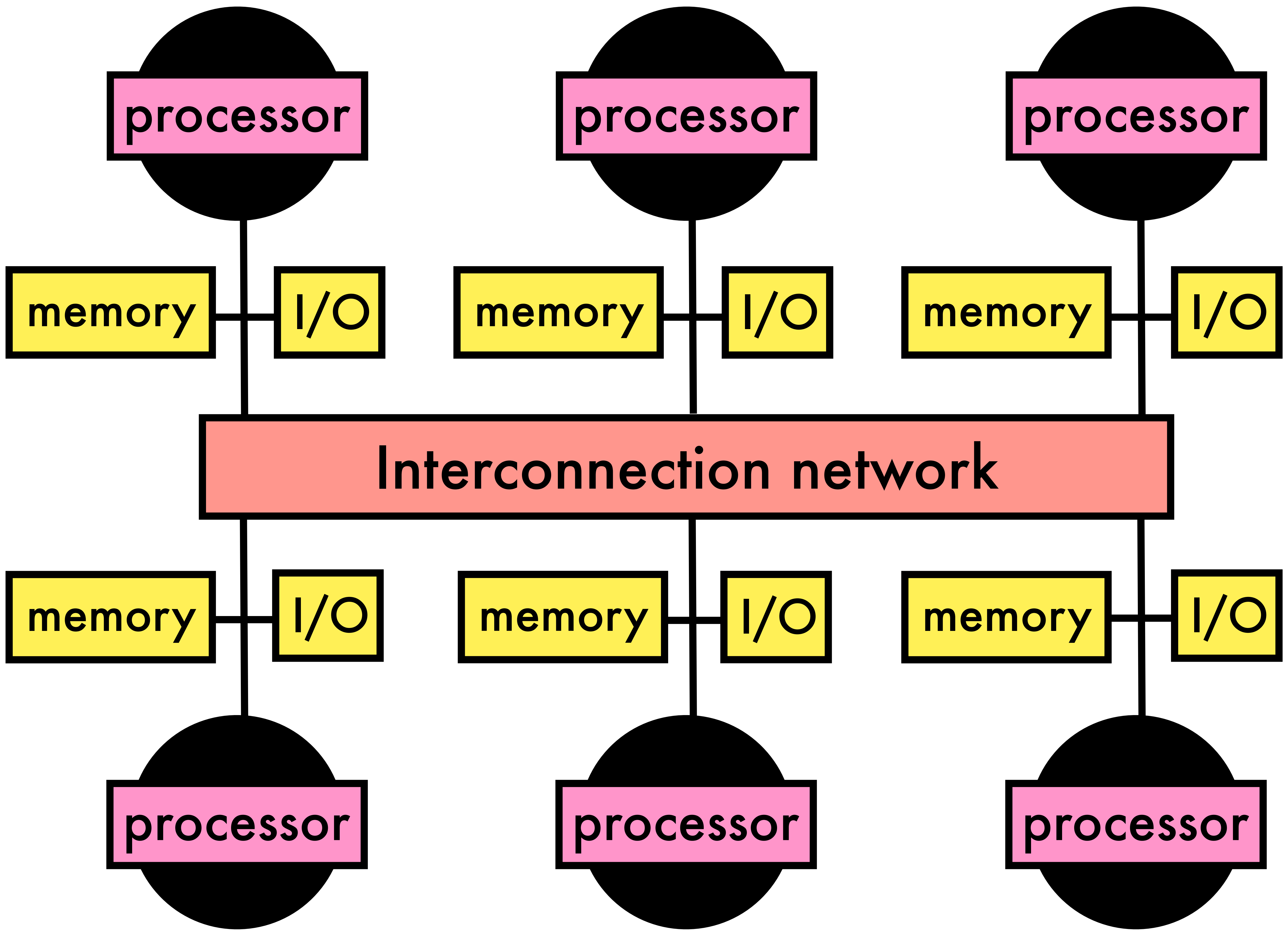
Uniform Memory Access (UMA)



Distributed Shared Memory (DSM)

Key idea: access time depends on location of data

Non-Uniform Memory Access (NUMA)



References:
J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach", Chap. 5 (2017)

Forms Of Parallelism

Data-level parallelism

Distribute data across processors

Processors perform the same task on different subsets of data in parallel

Task-level parallelism

Distribute tasks across processors

Different tasks may be run on the same data (as well as across different subsets of data)

more general

but also more complex

We will focus on task-level parallelism with a shared memory model

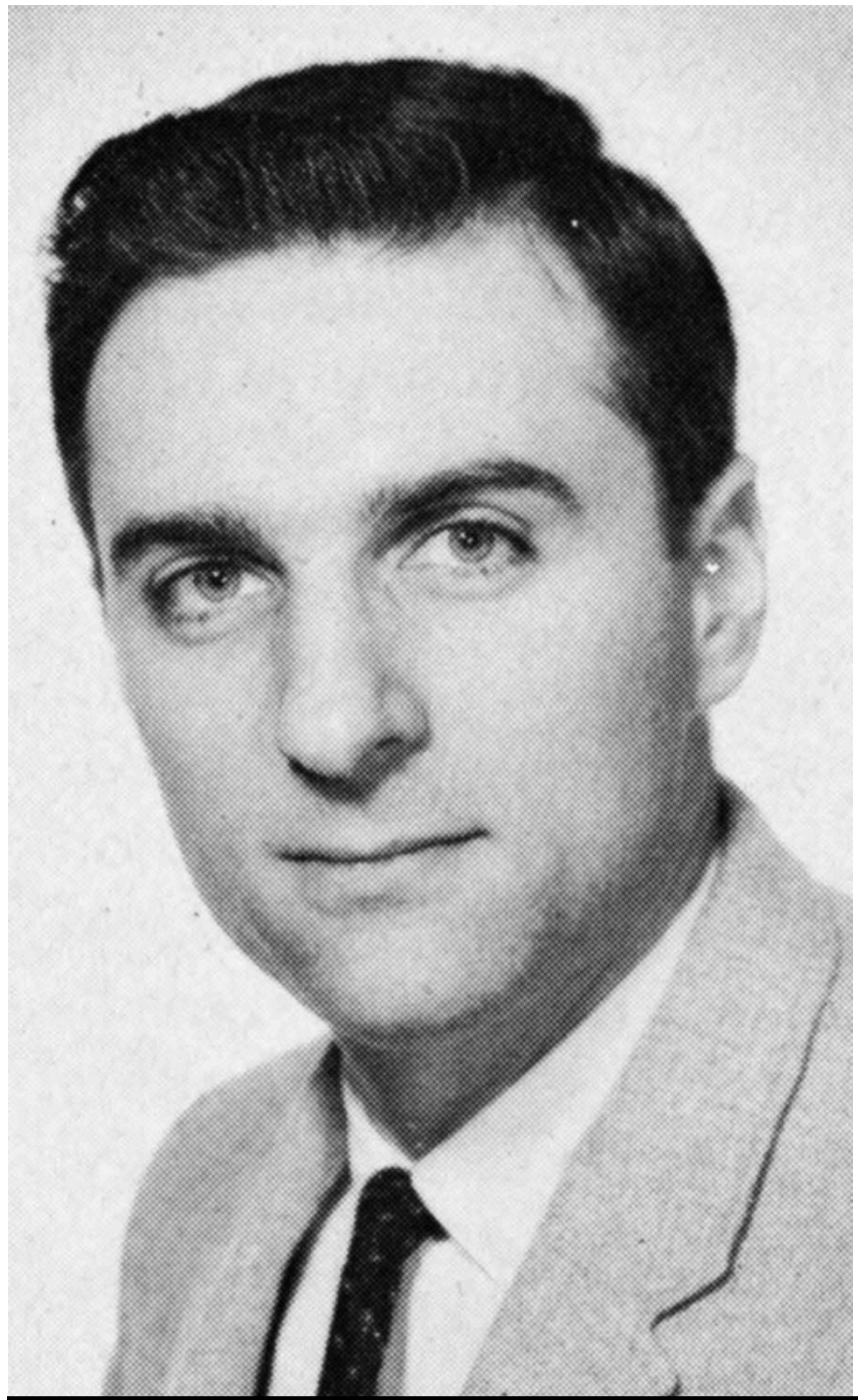
References:
https://en.wikipedia.org/wiki/Data_parallelism
https://en.wikipedia.org/wiki/Task_parallelism

Task-Parallel Platforms

Implementing task-level parallelism with threads	Task-parallel platforms
Task parallelism can be implemented with threads ("virtual processors") that share memory However, this has proven difficult to program: Scheduling/load-balancing is a challenging job	Add abstraction layer on top of threads Programmer specifies which tasks can run in parallel (but not where they run) Platform manages scheduling, balancing etc.

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)

Fork-Join Parallelism



Fork-join parallelism (1963)

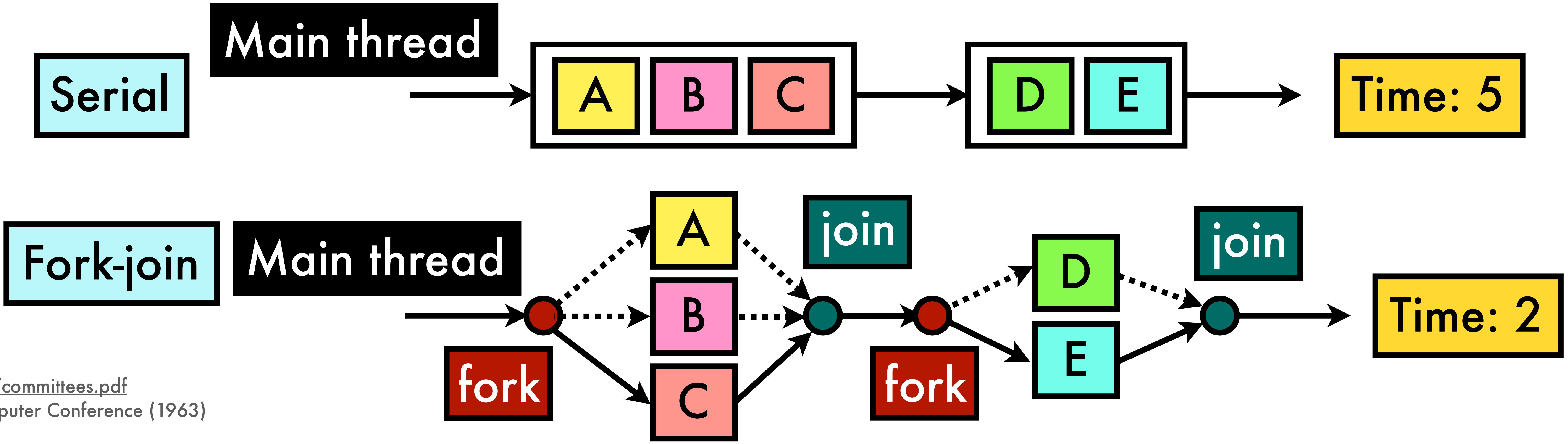
Most task-parallel platforms support fork-join Cilk OneTBB OpenMP

Spawn: "forks" - executes function while caller continues to run in parallel

Sync: "joins" - waits for spawned threads to finish before proceeding

Key concept: programmer only specifies which tasks can run in parallel, not which tasks must run in parallel

Parallel sections can fork recursively until reaching a given task granularity



References:
(M. Conway image) http://www.melconway.com/Home/pdf/committees.pdf
M. Conway, "A multiprocessor system design", Fall Joint Computer Conference (1963)
https://en.wikipedia.org/wiki/Fork-join_model
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)
R. Blumofe et al., "Cilk: An efficient multithreaded runtime system", *ACM SigPlan* (1995)
(OneTBB) https://github.com/oneapi-src/oneTBB

An Example: Fibonacci

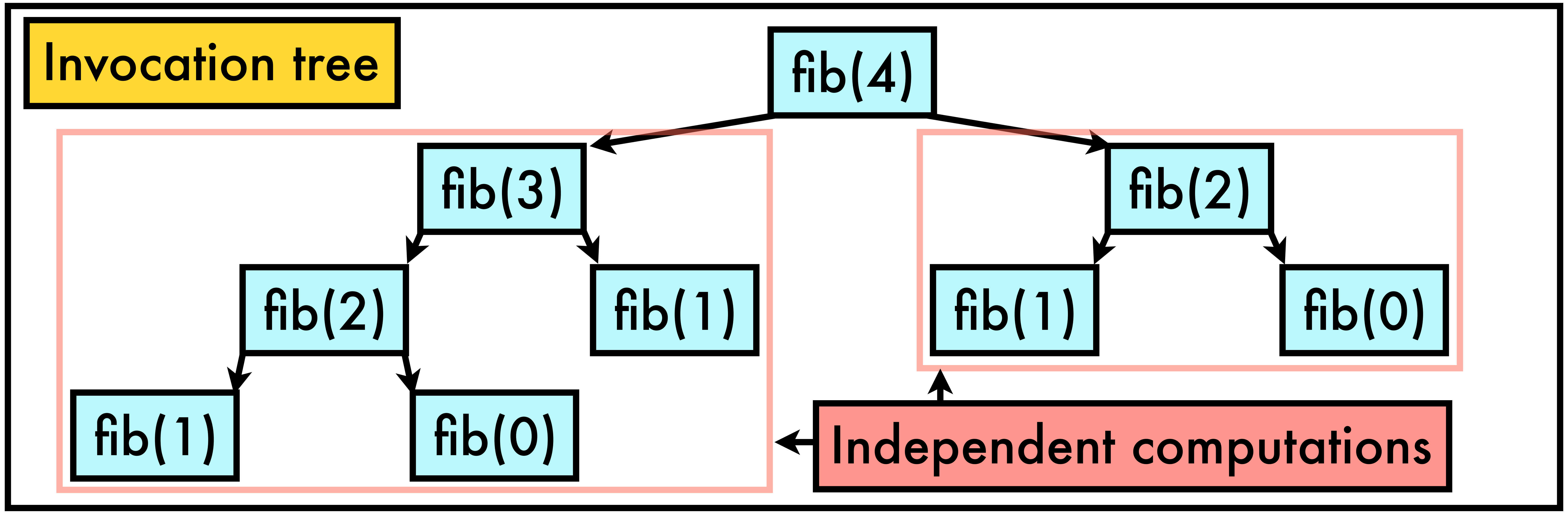
Fibonacci Numbers

```
def fib(n):
    if n < 2:
        return n
    x = fib(n - 1)
    y = fib(n - 2)
    return x + y
```

Not very efficient (no memoization)

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$ (exponential)



References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)

Parallel Code

Parallel pseudocode (CLRS/cilk)

```
def par_fib(n):
    if n < 2:
        return n
    x = spawn par_fib(n - 1)
    y = par_fib(n - 2)
    sync
    return x + y
```

Parallel version (Python) — CPython GIL 🐢

Use `nogil` + coarsening for speedup

```
def par_fib(n):
    if n < 2:
        return n
    with ThreadPoolExecutor() as exec:
        x_future = exec.submit(par_fib, n - 1) # spawn
        y = par_fib(n - 2)
        x = x_future.result() # sync
    return x + y
```

`spawn` says main thread can execute in parallel with the spawned child, not that it must

`sync` says parent must wait for all spawned children to finish (join)

`spawn/sync` express the logical parallelism of the tasks

It is the responsibility of the scheduler to assign the tasks to processors

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)
R. Blumofe et al., "Cilk: An efficient multithreaded runtime system", *ACM SigPlan* (1995)
(nogil python) https://nogil.dev/

Computation DAG

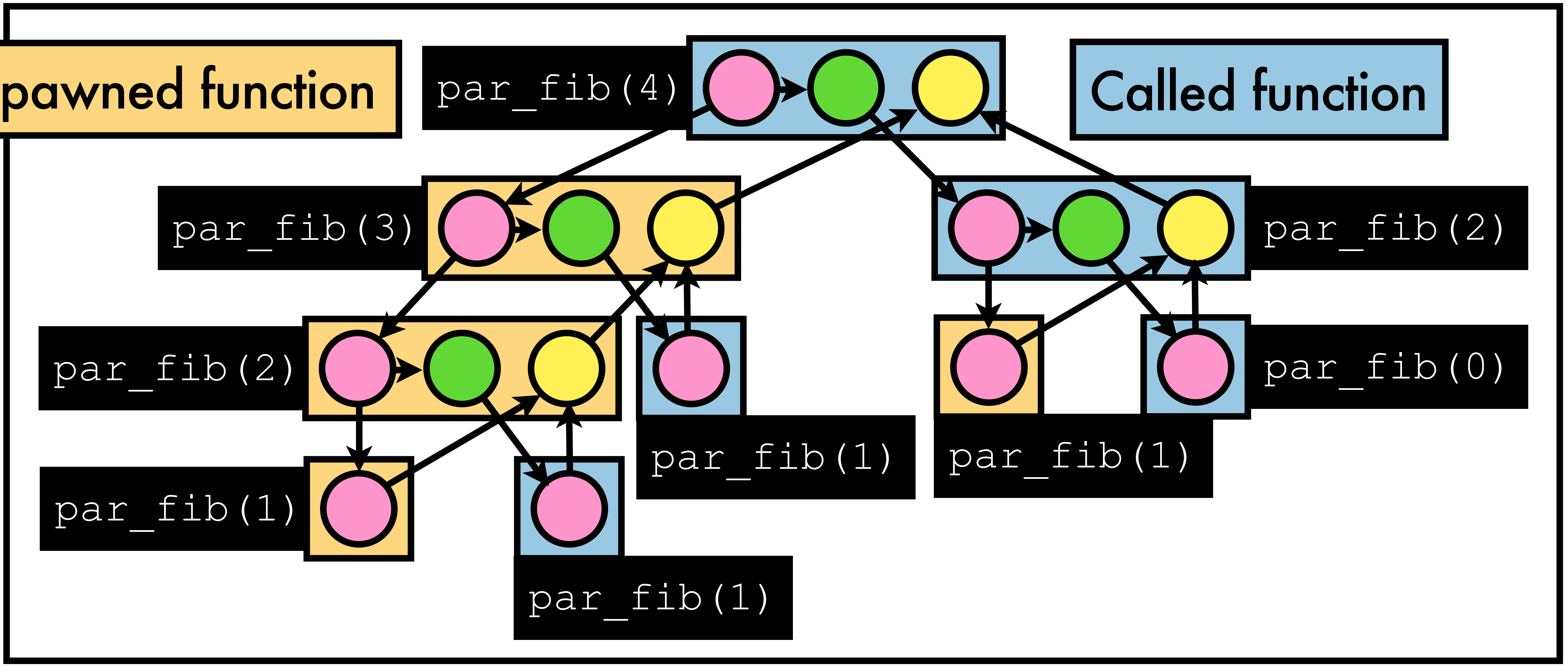
We can view execution as a computation DAG (a.k.a. "parallel trace"): $G = (V, E)$

Executed instructions: vertices in V

Dependencies between instructions: edges in E

To avoid clutter: group chains of instructions with no parallel/procedural control into "strands"

```
def par_fib(n):
    if n < 2:
        return n
    x = spawn par_fib(n - 1)
    y = par_fib(n - 2)
    sync
    return x + y
```

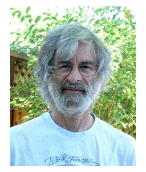


References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)
https://www.csd.uwo.ca/~mmorenom/cs3101_Winter_2015/Multithreaded_Parallelism_and_Performance_Measures.pdf

Parallel Computation Analysis: Assumptions

<u>Assumptions for analysis</u>

1. We have an ideal parallel computer:
   - multiple processors
   - sequentially consistent memory
2. Processors have equal computing power
3. No overhead for scheduling

Sequentially consistent (Lamport, 1979 ):
Instruction execution preserves the partial ordering of DAG

Attain via sequential processors; FIFO memory (processors communicate through memory)

References/image credits:
T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)
(Sequential consistency) L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE ToC* (1979)
(Image of L. Lamport) https://en.wikipedia.org/wiki/Leslie_Lamport#/media/File:Leslie_Lamport.jpg

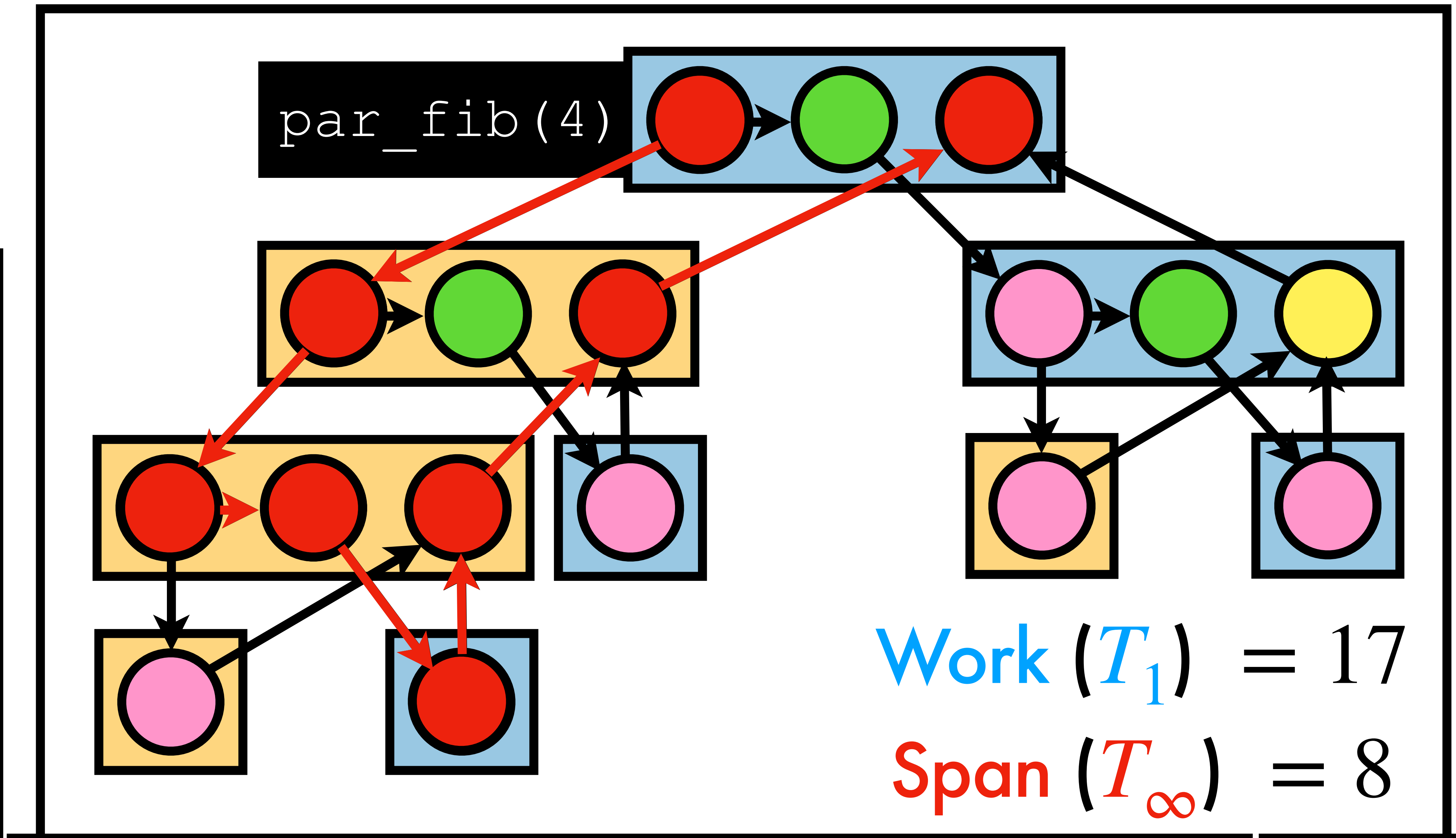
Work/Span Analysis

Let T_P denote runtime of program on P processors

Work (T_1): time to execute program on 1 processor

Span (T_∞): time to execute program on ∞ processors

The span is the sum of the runtimes of strands on the "critical path" - the longest path in the computation DAG



Parallelism $(T_1/T_\infty) = 17/8 = 2.125$ $n = 4$

Work law: $T_P \geq T_1/P$ (P processors can achieve at most P units of work per time step)

Span law: $T_P \geq T_\infty$ (with ∞ processors, we can emulate the P processors and leave rest idle)

Speedup: T_1/T_P (by work law, $T_1/T_P \leq P$ - speedup on our ideal parallel machine is at most P)

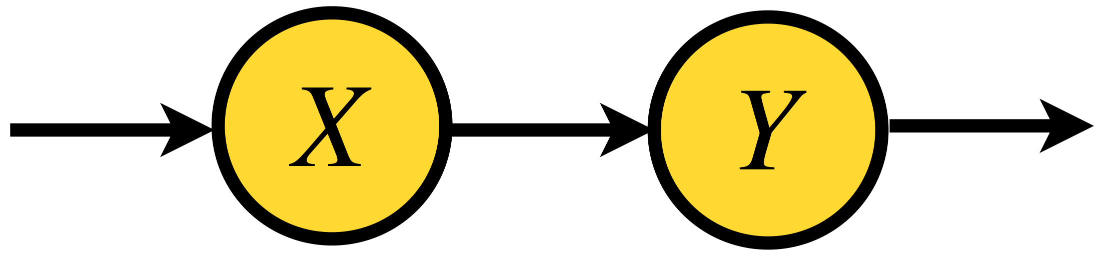
Linear speedup: $T_1/T_P = \Theta(P)$ Perfect linear speedup: $T_1/T_P = P$ super-linear impossible in this model

Parallelism: T_1/T_∞ (maximum possible speed up with any number of processors)

References:
T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)
https://en.wikipedia.org/wiki/Analysis_of_parallel_algorithms

Parallel Analysis

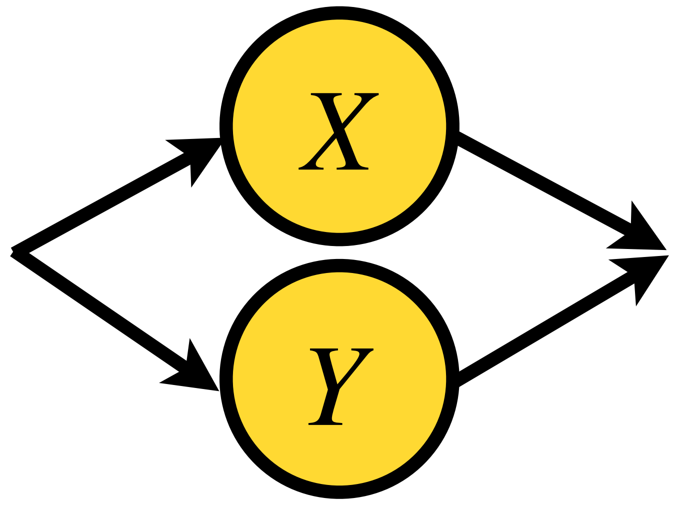
Serial



Work: $T_1(X \cup Y) = T_1(X) + T_1(Y)$

Span: $T_\infty(X \cup Y) = T_\infty(X) + T_\infty(Y)$

Parallel



Work: $T_1(X \cup Y) = T_1(X) + T_1(Y)$

Span: $T_\infty(X \cup Y) = \max(T_\infty(X), T_\infty(Y))$

Work $T_1(n)$ for `par_fib(n)`:

$$T_1(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Span $T_\infty(n)$ for `par_fib(n)`:

$$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$$

$$= T_\infty(n-1) + \Theta(1) \qquad T_\infty = \Theta(n)$$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n \Big/ n\right)$

Grows fast with n

Lots of parallelism

Reference:
T. Cormen et al., "Introduction to algorithms", Chap 26, MIT press (2022)