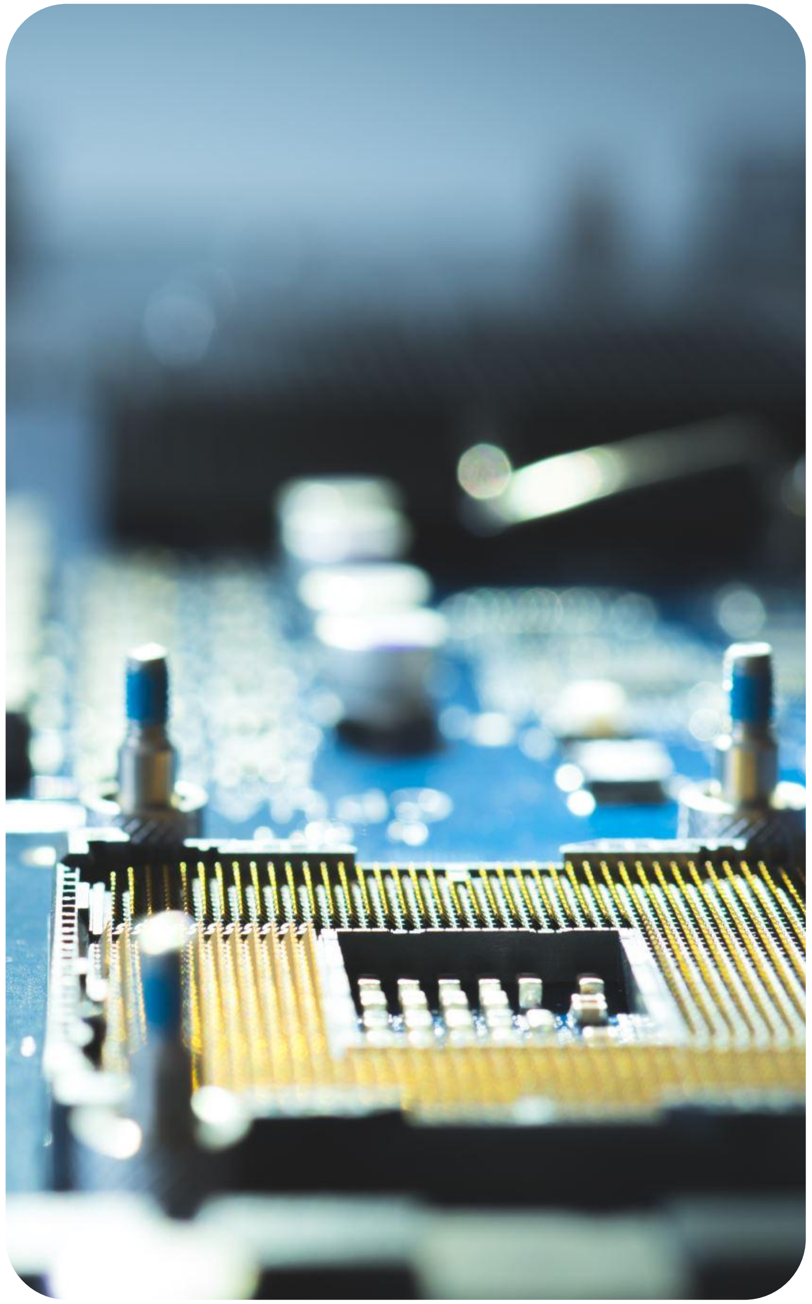
Designing for Performance

By: Bilal Ahmed

Background

- The cost of computer systems has declined significantly over time.
- System performance and capacity have increased dramatically.
- Modern laptops match the computing power of mainframes from a decade ago.
- Processing power is now effectively inexpensive.
- Microprocessors are so low-cost that they are often used in disposable applications.



Applications that require power

- Image processing (Computer Vision)
- Three-dimensional rendering (Game Development)
- Speech recognition
- Videoconferencing
- Voice and video annotation of files
- Simulation modeling

Microprocessor Speed

The power of Intel x86 processors and IBM mainframes stems from the continuous drive for higher speed by chip manufacturers.

This evolution reflects Moore’s Law, with new chip generations emerging roughly every three years.

Each new generation typically integrates four times as many transistors as its predecessor.

The raw speed of a microprocessor is effective only if it is consistently supplied with instructions.

Any disruption in instruction flow reduces processor performance.

While chipmakers focus on increasing transistor density, processor designers develop techniques to maintain efficient instruction flow.

Microprocessor Speed Techniques

- **Pipelining**: Splits instruction execution into stages (fetch, decode, execute, etc.) so multiple instructions are processed simultaneously.
- **Branch Prediction**: Anticipates future instruction paths and prefetches them to keep the processor busy.
- **Superscalar Execution**: Issues multiple instructions per clock cycle using parallel pipelines.
- **Data Flow Analysis**: Reorders instructions based on data dependencies to minimize delays.
- **Speculative Execution**: Executes likely future instructions in advance, storing results temporarily to maximize utilization.

Analogy

- **Pipelining**: Executes multiple stages at once (e.g., while one instruction is executed, the next is decoded). *Example: Assembly line in a factory.*
- **Branch Prediction**: Guesses the next instruction path. *Example: Predicting loop continuation before the condition check.*
- **Superscalar Execution**: Executes more than one instruction per cycle. *Example: Performing an addition and a multiplication simultaneously.*
- **Data Flow Analysis**: Reorders instructions to avoid stalls. *Example: Executing* $C = A + B$ *while waiting for result of* $D = E \times F$.
- **Speculative Execution**: Runs instructions in advance using predictions. *Example: Processor executes "if condition true" branch before the condition is confirmed.*

Performance Balance

Processor power has advanced very rapidly.

Other computer components have not progressed at the same pace.

This creates a performance imbalance.

System design must adjust architecture and organization to restore balance.

Modern CPUs can execute billions of instructions per second, but RAM access is still comparatively slow → creating the *memory wall problem*.

It’s like having a Ferrari (CPU) stuck in a traffic jam behind a donkey cart (memory).

Memory and CPU Issue

The biggest performance problem occurs **between processor and main memory**.

Processor speed has increased rapidly, but **memory speed** has not kept up.

This pathway is crucial for sending **instructions and data** to the processor.

If memory or the pathway is too slow, the processor enters a **wait state**, wasting valuable time.

Solutions

Make DRAM wider: Fetch more bits at once using wider memory chips and wide data buses.	**Example:** DDR memory fetching **64 bits per cycle** instead of 16.
Smarter DRAM interface: Add cache or buffers directly on the DRAM chip for faster access.	**Example:** Like keeping a **snack drawer at your desk**, so you don't keep walking to the kitchen.
Use caches to reduce memory access: Place multiple levels of cache (on-chip and off-chip) between CPU and memory to minimize slow memory calls.	**Example:** A processor with **L1, L2, and L3 cache** to cut down slow RAM fetches.
Faster connections: Improve processor–memory communication with high-speed buses and bus hierarchies for smoother data flow.	**Example: PCIe 5.0** or **Infinity Fabric** moving data at very high speeds.

Peripherals

As processors get faster, they can run more powerful and complex applications.

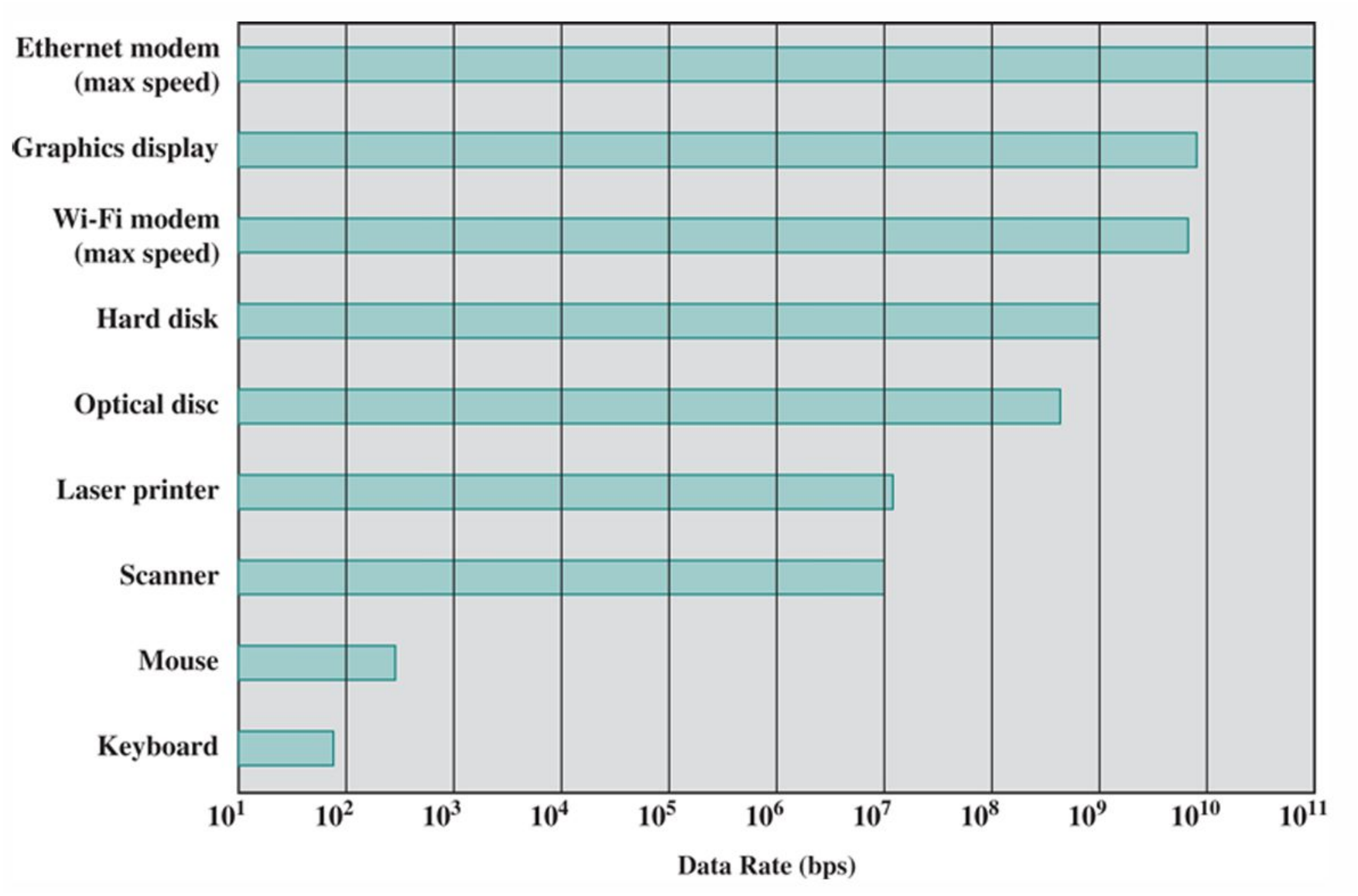
These advanced applications (e.g., video editing, AI, gaming) need peripherals that handle large amounts of input/output (I/O).

Examples of such peripherals: high-speed storage, GPUs, network cards, VR headsets.

This puts pressure on the computer’s I/O system to manage data transfer quickly and efficiently.

Therefore, I/O handling becomes a key design focus to avoid bottlenecks.

Peripherals Speed



The key is to balance

Computer design requires **balance** between processor, memory, I/O devices, and interconnections.

Challenge: Different components improve at **different rates** (e.g., CPU vs memory).

Challenge: **New applications and devices** change instruction use and data access patterns.

Result: Computer design must **constantly adapt and evolve**.

Improvements in Chip Organization and Architecture

Increase processor speed → Achieved by shrinking transistor (logic gate) size so more fit on the chip. Smaller gates = shorter signal paths = faster propagation. **Additionally, raising the clock speed** (running cycles faster) makes each instruction execute more quickly.

Improve caches → Larger and faster caches, especially when placed on the CPU chip, reduce the delay of fetching instructions/data from slower main memory.

Enhance processor organization/architecture → Use techniques like pipelining, parallelism, superscalar execution, or multiple cores so the processor can handle more work at once.

Problems with Clock Speed and Login Density

Power & heat → Smaller, faster chips consume more power and are harder to cool.

RC delay → Thinner and closer wires increase resistance & capacitance, slowing signal travel.

Memory gap → Memory speed (latency) and data transfer (throughput) lag behind processor speed.

Processor Trends

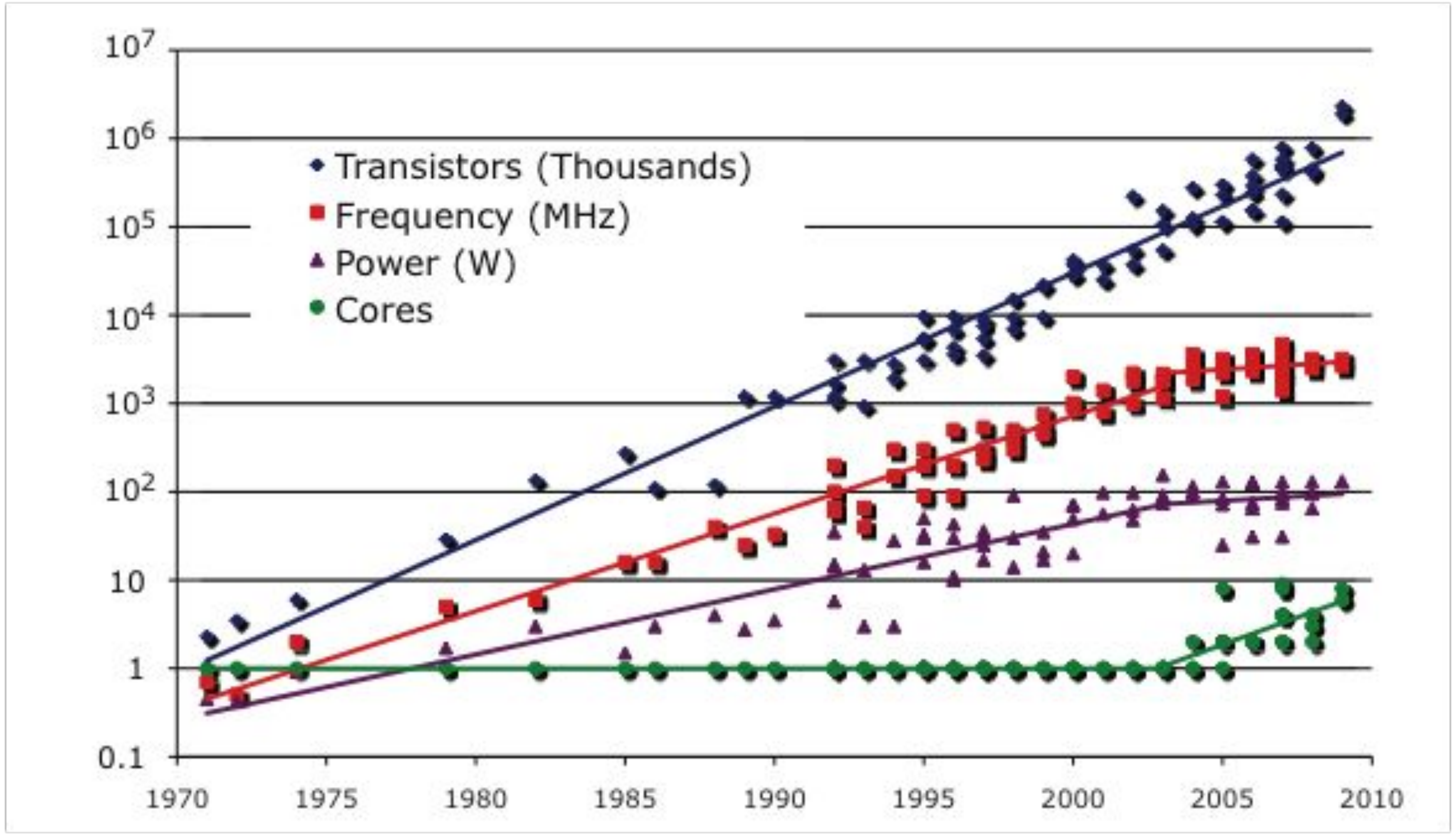


Figure 2.11 Processor Trends

From Clock Speed to Multicore

Before (Past Strategy)

Performance improved by **increasing clock speeds** and **adding more transistors**.

Challenge (Problem Faced)

Higher clock speeds caused **power and heat issues**, limiting further gains.

Solution (New Approach)

With transistor counts still rising, designers shifted to **multicore processors**, enabling performance gains through **parallel processing**.

Multicore

To overcome limits of clock speed and heat, designers now place **multiple processors (cores) on one chip** with a shared cache.

This **multicore approach** boosts performance without raising clock speed.

A single complex processor gives limited gains (performance grows ~√complexity).

But if software supports it, **adding more cores nearly doubles performance**.

Strategy: **use multiple simpler cores instead of one complex processor**.

Multicore

Adding more processors (cores) made **larger caches worthwhile**, since caches use less power than processing logic.

As chip density increased → designers kept adding **more cores (2 → 4 → 8 → 16 …)** and **bigger caches**.

L1 cache → private to each core.

L2 and L3 caches → initially shared, later L2 often made private, with L3 shared.

Many Integrated Core (MIC)

Places **dozens to hundreds of simple cores** on one chip.

Designed for **parallel computing** (HPC, AI, simulations).

Focuses on **throughput** rather than single-core speed.

Example: **Intel Xeon Phi**.

Limitation: Needs **parallel-optimized software** to benefit.

GPUs

Multicore & MIC strategy → Many general-purpose processors on one chip (all similar).

Alternative design → Combine general-purpose CPUs with specialized cores (GPU).

Specialized cores → Include GPUs (for graphics) and video-processing units.

GPU role → Handles parallel operations on graphics and video data.

MIC = Many identical CPU-like cores (homogeneous).

GPU = Specialized cores for highly parallel tasks (heterogeneous).

Summary

Feature	**Traditional CPU**	**Multicore CPU**	**MIC (Many Integrated Core)**	**GPU (Graphics Processing Unit)**
Core Type	Single general-purpose	Few general-purpose cores	Many general-purpose cores (homogeneous)	Specialized parallel cores (heterogeneous)
Parallelism	Very limited	Moderate	High (many threads)	Very high (massive parallelism)
Focus	Sequential performance	Balanced: sequential + parallel	General-purpose parallel workloads	Graphics, video, AI, and data-parallel tasks
Cache Usage	Large, per CPU	Shared + private caches	Larger shared cache across many cores	Small cache per core, relies on memory bandwidth
Typical Use	General computing tasks	Modern desktops, servers	HPC, scientific workloads	Graphics rendering, AI, ML, simulations

Amdahl's Law

- System performance can be improved by faster processors, caches, memory, or I/O.
- However, improving only one part of the system gives limited overall gain.
- Proposed by Gene Amdahl in 1967.
- **Amdahl's Law**: Explains the **potential speedup of a program** when using multiple processors compared to a single processor.
- Performance gain is **limited by the portion of the program that must run sequentially**.
- Even with many processors, the sequential part becomes a **bottleneck**.

Understanding Law

- Imagine a program has **two parts**:
  - A part that can run in **parallel** (many processors can work on it).
  - A part that is **sequential** (only one processor can do it).
- **Sequential part = the limit.** No matter how many processors you add, that sequential portion will always take the same time.
- Example:
  - A program takes **10 hours**.
  - **8 hours** can run in parallel, **2 hours** must run sequentially.
  - With 100 processors → the 8 hours shrink almost to zero, but the **2 hours remain**.
  - So, the program **still takes at least 2 hours**.
- **Key Idea:** Adding more processors helps only up to a point. The part that can’t be parallelized sets a **hard limit** on speedup.

Formula

$$S = \frac{1}{(1 - F) + \frac{F}{N}}$$

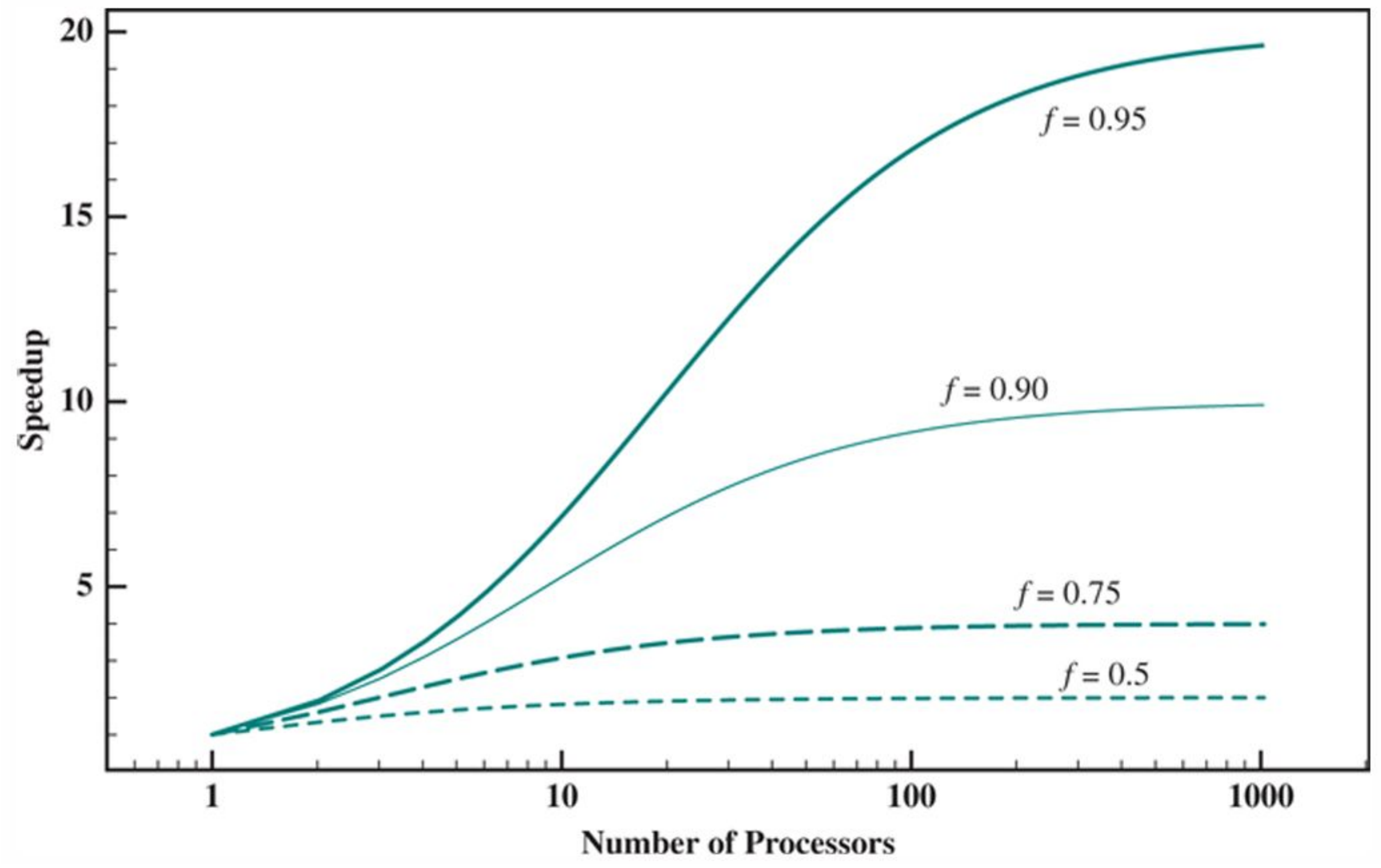
Where:

- **S** = Speedup (overall performance gain)
- **F** = Fraction of program that can be parallelized ($0 \le F \le 1$)
- **N** = Number of processors

- 1. When f is small, the use of parallel processors has little effect.
- 2. As N approaches infinity, speedup is bound by 1 / (1−f), for using more processors.

Amdahl’s Law and Number of Processors

- The speed-up of a program from parallelization is limited by how much of the program can be parallelized.
- For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used.
- If 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10×



Simple examples

Cooking analogy:

- You can chop vegetables (parallel: many people chopping).
- But you **can’t start cooking** until the vegetables are ready (sequential step).

Program example:

- Loading data from disk (sequential).
- Once loaded, different cores can process different parts (parallel).
- But final results may need to be combined in order (sequential again).

Practical Examples

- **1. Operating System (OS)**
- **Boot sequence**: The OS must load the kernel before drivers, and drivers before applications.
  - You can’t run them in parallel because each step depends on the previous one.
- **2. Graphics (GPU rendering)**
- **Frame rendering pipeline**:
  - You must first **calculate geometry** before applying **textures**.
  - Finally, you **display** the frame.
  - Each stage depends on the previous result, so parts must be sequential even if others run in parallel.
- **3. Robotics**
- **Sensor → Processing → Action loop**:
  - Robot must **read sensor data** → then **process/decide** → then **send motor commands**.
  - You can parallelize sensor fusion or path planning, but the **decision-to-action flow is sequential**, otherwise the robot might act on incomplete info.
- That’s why Amdahl’s Law matters: even if you have **100 processors**, the sequential parts (like boot, rendering order, or sensor-to-action flow) will always limit the maximum speedup.

Problem

- Suppose that a task makes extensive use of floating-point operations, with 40% of the time consumed by floating-point operations. With a new hardware design, the floating-point module is sped up by a factor of K. Find maximum Speed-Up?
- Both **K (hardware speedup factor)** and **N (number of processors)** play the *same role* in Amdahl's Law:
- They are just **different ways** of expressing *how much faster* the improvable part becomes.
- In one case (processors) → speedup comes from **parallelism**.
- In the other case (floating-point unit, cache, I/O, etc.) → speedup comes from **better hardware design**.
- So you can think of **K as analogous to N** — both represent *how many times faster* the affected portion of the workload runs.

Solution

Given:

- Fraction of program using floating-point (**F**) = **0.40**
- Fraction not affected = **1 − F = 0.60**
- Speedup of floating-point part = **K**

Amdahl's Law:

$$S = \frac{1}{(1 - F) + \frac{F}{K}}$$

Substitute values:

$$S = \frac{1}{0.60 + \frac{0.40}{K}}$$

Solution

Now let's compute for some example values of K:

- **If K = 2** (floating-point 2× faster):

$$S = \frac{1}{0.60 + \frac{0.40}{2}} = \frac{1}{0.60 + 0.20} = \frac{1}{0.80} = 1.25$$

- **If K = 5**:

$$S = \frac{1}{0.60 + \frac{0.40}{5}} = \frac{1}{0.60 + 0.08} = \frac{1}{0.68} \approx 1.47$$

- **If K → ∞** (perfect floating-point hardware):

$$S = \frac{1}{0.60 + 0} = \frac{1}{0.60} \approx 1.67$$

No matter how fast the floating-point unit gets, **the maximum possible speedup is 1.67×** (limited by the 60% non-floating part).