The Software Crisis

First Software Crisis (1960s-1970s):

Problem: Early computers ran basic programs using assembly language. Assembly language programming was complex and not portable.

Solution: High-level languages like Fortran and C were introduced. They provided abstraction and portability without losing performance.

Second Software Crisis (1980s-1990s):
Problem: Building and maintaining large, complex applications with millions of lines of code.

Solution: Object-Oriented Programming (OOP) languages like C++ and Java, better tools, and software engineering methodologies.

Third Crisis (2005-Present):

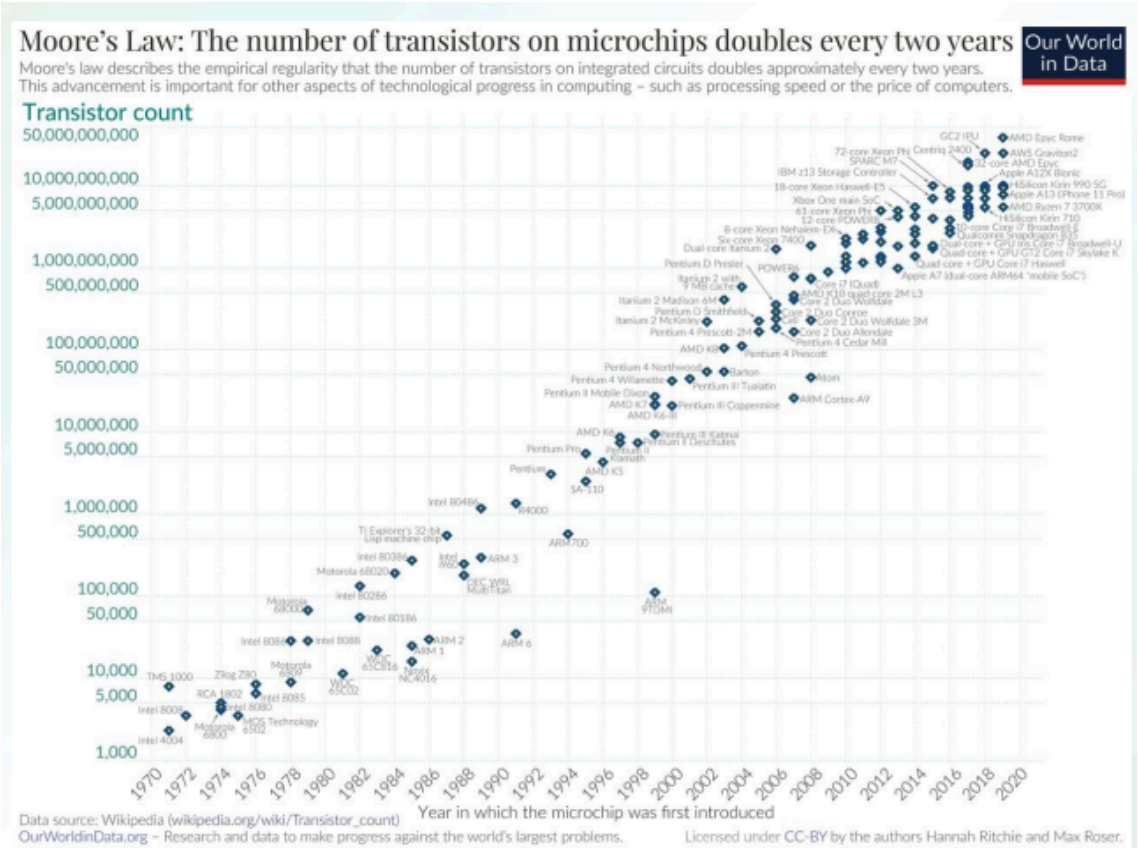
Gordon Moore's Law, introduced in 1965, states that the number of transistors on a microchip **doubles approximately every two years**, leading to exponential increases in computational power and decreases in cost per transistor.

Problem: Moore’s Law (faster processors) slowed down, creating a need for parallel computing to handle larger datasets and modern application requirements.

Sequential performance improvements are slowing down due to the end of Moore's Law.

Challenge: Need for continuous performance improvements to handle larger datasets and new features.

Prepared by: hjt



This image illustrates Moore's Law, which states that the number of transistors on a microchip doubles approximately every two years. This trend has historically enabled exponential growth in computational power and technology advancements.

1. **Exponential Growth**:
    - The graph shows transistor counts increasing logarithmically over time.
    - Microchips have evolved from **1,000 transistors in the 1970s** to **over 50 billion transistors today**.
2. **Notable Trends**:
    - Early processors (e.g., **Intel 4004** in 1971) had a small transistor count.
    - Modern processors like **Apple A15**, **AMD Epyc Rome**, and **Intel Xeon Platinum** show a massive increase.
3. **Sustained Progress**:
    - The doubling of transistors has led to improvements in **performance**, **energy efficiency**, and **cost reduction**.
    - This growth has fueled developments like smartphones, AI, and cloud computing.
4. **Future Limitations**:
    - Physical limitations (e.g., quantum effects) challenge continued transistor scaling.
    - Alternative technologies, such as quantum computing or neuromorphic chips, are emerging.

Prepared by: hjt

12th Gen Intel® Core™ Mobile Processors SKU Comparison—HX Series

	i9-12950HX	i9-12900HX	i7-12850HX	i7-12800HX	i5-12600HX
Processor Cores	16	16	16	16	12
Processor Threads	24	24	24	24	16
Number of P-cores	8	8	8	8	4
Number of E-cores	8				
Intel® Smart Cache (L3)	30MB	30MB	25MB	25MB	18MB

Processor Cores:

- Refers to the number of physical cores within the CPU. More cores allow better performance for multi-threaded tasks (e.g., gaming, video editing, simulations).

Processor Threads:

- Threads are the virtual execution units managed by the CPU. Intel's Hyper-Threading technology enables **two threads per P-core**.

P-cores (Performance Cores):

- Designed for high-performance, latency-sensitive workloads like gaming and single-threaded tasks.

E-cores (Efficient Cores):

- Optimized for power efficiency, handling background tasks and multi-threaded workloads.

Intel® Smart Cache (L3):

- A large pool of memory shared between cores for faster access to frequently used data.

i9 Processors:

- Both **i9-12950HX** and **i9-12900HX** have identical specifications: 16 cores (8 P-cores + 8 E-cores), 24 threads, and **30MB of cache**.
- Designed for high-end performance in demanding workloads.

Prepared by: hjt

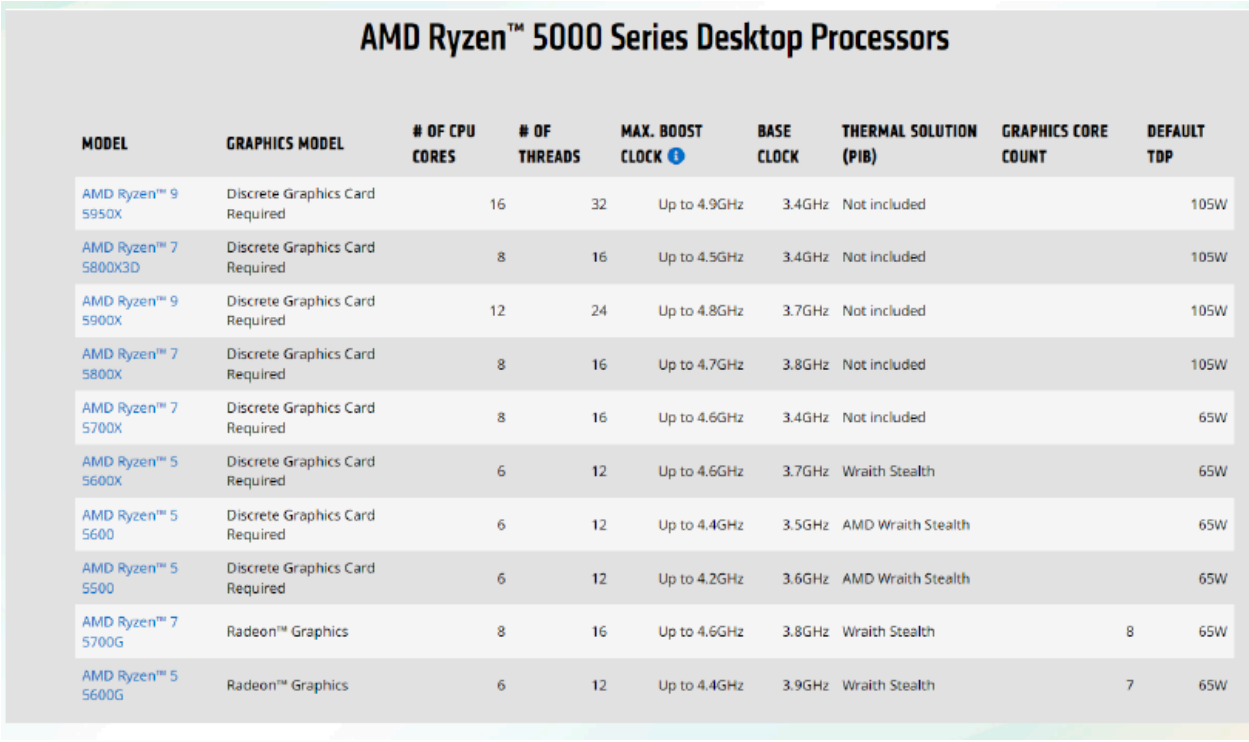
i7 Processors:

- **i7-12850HX** and **i7-12800HX** feature the same core/thread configuration as the i9 models but have **slightly less L3 cache (25MB)**.
- Targeted for powerful performance with a small compromise in cache.

i5 Processor:

- The **i5-12600HX** has fewer cores (12 total: 4 P-cores + 8 E-cores) and **16 threads**.
- It also features **less L3 cache (18MB)**, making it suitable for mid-range tasks.

i9 Models: Best for intensive applications like gaming, simulations, and video rendering.
i7 Models: Ideal for productivity tasks and general multi-threaded workloads.
i5 Model: Perfect for users who need a balance of performance and efficiency.



AMD Ryzen™ 5000 Series Desktop Processors

MODEL	GRAPHICS MODEL	# OF CPU CORES	# OF THREADS	MAX. BOOST CLOCK	BASE CLOCK	THERMAL SOLUTION (PIB)	GRAPHICS CORE COUNT	DEFAULT TDP
AMD Ryzen™ 9 5950X	Discrete Graphics Card Required	16	32	Up to 4.9GHz	3.4GHz	Not included		105W
AMD Ryzen™ 7 5800X3D	Discrete Graphics Card Required	8	16	Up to 4.5GHz	3.4GHz	Not included		105W
AMD Ryzen™ 9 5900X	Discrete Graphics Card Required	12	24	Up to 4.8GHz	3.7GHz	Not included		105W
AMD Ryzen™ 7 5800X	Discrete Graphics Card Required	8	16	Up to 4.7GHz	3.8GHz	Not included		105W
AMD Ryzen™ 7 5700X	Discrete Graphics Card Required	8	16	Up to 4.6GHz	3.4GHz	Not included		65W
AMD Ryzen™ 5 5600X	Discrete Graphics Card Required	6	12	Up to 4.6GHz	3.7GHz	Wraith Stealth		65W
AMD Ryzen™ 5 5600	Discrete Graphics Card Required	6	12	Up to 4.4GHz	3.5GHz	AMD Wraith Stealth		65W
AMD Ryzen™ 5 5500	Discrete Graphics Card Required	6	12	Up to 4.2GHz	3.6GHz	AMD Wraith Stealth		65W
AMD Ryzen™ 7 5700G	Radeon™ Graphics	8	16	Up to 4.6GHz	3.8GHz	Wraith Stealth	8	65W
AMD Ryzen™ 5 5600G	Radeon™ Graphics	6	12	Up to 4.4GHz	3.9GHz	Wraith Stealth	7	65W

Performance Highlights:

- **Ryzen 9 5950X**: Top-tier performance with 16 cores, 32 threads, and up to **4.9GHz boost clock**, ideal for heavy multitasking, gaming, and productivity workloads.
- **Ryzen 7 5800X3D**: Optimized for gaming, featuring AMD's **3D V-Cache™ technology** for enhanced gaming performance.

Prepared by: hjt

Thermal Solutions:

- High-performance models (e.g., 5950X, 5800X) do **not include cooling solutions**, requiring aftermarket coolers.
- Lower TDP models (like 5600X, 5600G) include **Wraith Stealth** coolers.

Graphics Integration:

- Models with a **"G"** suffix (e.g., Ryzen 7 5700G and Ryzen 5 5600G) include **Radeon™ Graphics**, eliminating the need for a discrete GPU.
    - **5700G**: 8 GPU cores.
    - **5600G**: 7 GPU cores.

TDP (Thermal Design Power):

- Higher-performance models (5950X, 5900X) require **105W TDP**, whereas mid-range options (e.g., 5600, 5600G) maintain a more power-efficient **65W TDP**.
- **Ryzen 9**: Ideal for extreme workloads (content creation, multitasking).
- **Ryzen 7**: Balanced performance for gaming and productivity.
- **Ryzen 5**: Affordable and efficient for mainstream usage.
- **G-Series**: Best for systems without discrete GPUs (integrated graphics).

Choose the right CPU based on your **budget, performance needs, and power efficiency**.

Prepared by: hjt

Flynn's Taxonomy

Flynn's Taxonomy classifies computer architectures based on instruction and data streams:

1. **SISD (Single Instruction, Single Data):**
    - **Example:** Traditional single-core processors.
    - **How it works:** One instruction operates on one data stream at a time.
2. **SIMD (Single Instruction, Multiple Data):**
    - **Example:** Graphics Processing Units (GPUs).
    - **How it works:** One instruction operates on multiple data streams simultaneously.
3. **MISD (Multiple Instruction, Single Data):**
    - **Example:** Fault-tolerant systems.
    - **How it works:** Multiple instructions operate on the same data stream.
4. **MIMD (Multiple Instruction, Multiple Data):**
    - **Example:** Multi-core processors.
    - **How it works:** Multiple instructions operate on multiple data streams independently.

Sequential vs. Parallel Computing

Sequential Computing:

Sequential computing refers to the execution of **one task at a time** in a specific order. Instructions are processed **linearly**, meaning each step must be completed before moving to the next one. This is the traditional model used in single-core systems.

- **Definition:** Tasks are executed one after another in a single sequence.
- **Example:** Cooking step-by-step: chop vegetables, then cook them, then serve.

How Sequential Computing Works

- Tasks are broken down into smaller steps.
- Each step is executed one after the other in **sequential order**.
- There is no parallel execution or overlap between tasks.

Pros of Sequential Computing

1. **Simplicity**:
    - Easier to design, understand, and debug as tasks execute in a clear step-by-step order.
    - Example: Writing basic programs or algorithms in languages like Python or C.
2. **Predictability**:
    - Execution behavior is deterministic; results are consistent since only one instruction executes at a time.
3. **Low Overhead**:
    - No need for **task synchronization** or complex resource management.
    - No communication delays as all operations occur on a single core.
4. **Suitable for Small Tasks**:
    - Ideal for simple programs or systems where tasks are not time-intensive.
    - Example: Single-user programs like calculators or text-based applications.

Cons of Sequential Computing

1. **Limited Performance**:
    - Can only execute **one task at a time**, leading to slower execution for large, complex tasks.
    - Cannot leverage **multi-core** or distributed systems.
2. **Poor Scalability**:
    - Adding more resources (e.g., multiple cores or CPUs) does not improve performance.
    - The system is limited by the processing power of a single core.
3. **Inefficient for Complex Problems**:
    - Tasks that can be broken into **independent subtasks** (e.g., large datasets, simulations) are better suited for parallel computing.
4. **Resource Underutilization**:
    - In modern systems with multiple cores, sequential computing may leave other cores idle, reducing efficiency.
5. **Longer Execution Time**:
    - Processing large tasks sequentially takes significantly more time compared to parallel execution.

Comparison Example

Imagine sorting a deck of 52 cards:

- **Sequential Approach**: One person sorts the entire deck alone, step by step.
- **Parallel Approach**: Multiple people divide the deck and sort different sections simultaneously, then merge the results.

When to Use Sequential Computing:

1. Small programs or tasks that don't require heavy computation.
2. Environments where **deterministic execution** is critical (e.g., safety systems).
3. Systems with **limited resources** (e.g., single-core devices).

Sequential computing is straightforward, reliable, and ideal for smaller tasks. However, it becomes a bottleneck for performance and scalability when handling modern, large-scale problems.

Concurrent Computing:

Concurrent computing refers to the execution of multiple tasks that appear to run simultaneously but are not necessarily executing **at the same instant**. It involves tasks sharing a **single-core CPU** by switching between tasks rapidly to create the **illusion of parallelism**.

- **Definition:** Tasks appear to run simultaneously due to fast switching by the CPU.
- **Example:** A single-core CPU switching between downloading and rendering tasks.

How Concurrent Computing Works

- Tasks are broken into smaller parts or threads.
- The system switches between these tasks efficiently to overlap execution times.
- In a **single-core CPU**, only one task executes at a time, but the CPU alternates between tasks.

Pros of Concurrent Computing

1. **Improved Resource Utilization**:
    - Enables efficient use of CPU cycles by handling **I/O-bound tasks** while waiting for input/output operations (e.g., reading a file, downloading data).
2. **Responsiveness**:

    - Systems remain responsive by allowing multiple tasks to appear active simultaneously.
    - Example: A video player downloading a file while playing the video.
3. **Simplified Program Design**:
    - Suitable for applications with tasks that can overlap without dependencies.
    - Example: Managing user input, animations, and network requests in a game.
4. **Supports Multitasking**:
    - Allows users to perform multiple operations (e.g., editing a document while printing) by switching between tasks seamlessly.
5. **Low Overhead**:
    - Does not require multiple processors or cores. Concurrency can work effectively on single-core CPUs.

Cons of Concurrent Computing

1. **Task Switching Overhead**:
    - Frequent switching between tasks introduces overhead (context switching), which reduces efficiency.
2. **Not True Parallelism**:
    - On single-core CPUs, only one task runs at any given time; others are waiting, so it is not true simultaneous execution.
3. **Synchronization Complexity**:
    - If tasks share resources (e.g., a shared file), **synchronization** issues like race conditions and deadlocks can arise.
    - Example: Two tasks trying to update the same file simultaneously.
4. **Difficult Debugging**:
    - Identifying bugs becomes harder because tasks execute in an interleaved manner, leading to unpredictable behaviors.
5. **Performance Limitations**:
    - If tasks are CPU-bound (require heavy computation), concurrent computing may not improve performance.

Example of Concurrent Computing

Imagine a single cashier handling multiple customers:

1. **Task 1**: Scans items for Customer A.
2. **Task 2**: While waiting for Customer A's payment, starts scanning items for Customer B.
3. **Task 3**: Goes back to Customer A to confirm payment and finish.

When to Use Concurrent Computing

1. Applications with **I/O-bound tasks** (e.g., web servers handling multiple requests).
2. Systems requiring **responsiveness** (e.g., GUI applications, games).
3. Programs that benefit from task overlap but do not require true parallel execution.

Concurrent computing allows tasks to appear as if they run simultaneously by efficiently switching between them. While it improves responsiveness and resource usage, it does not achieve true parallel execution. For CPU-bound tasks, **parallel computing** on multi-core systems is more appropriate.

Parallel Computing:

Parallel computing is a computational model where **multiple tasks or processes are executed simultaneously** across multiple processors, cores, or machines. This model improves performance by breaking tasks into smaller parts that can run independently in parallel.

- **Definition:** Tasks are executed simultaneously across multiple processors.
- **Example:** Cooking with friends: one chops vegetables, another cooks, and another serves.

How Parallel Computing Works

1. **Tasks are divided into independent subtasks.**
2. **These subtasks run simultaneously on multiple processing units (multi-core CPUs, GPUs, or distributed systems).**
3. **Once completed, results from all parallel tasks are combined to produce the final output.**

Prepared by: hjt

Pros of Parallel Computing

1. **Improved Performance:**
    - **Multiple tasks execute simultaneously, significantly reducing execution time for large computations.**
    - **Example: Weather simulations, large data analysis.**
2. **Better Resource Utilization:**
    - **Maximizes the use of multi-core processors or GPUs.**
    - **Ensures all available processing units are actively working.**
3. **Handles Large-Scale Problems:**
    - **Essential for solving complex, time-consuming problems like scientific simulations, machine learning, and cryptography.**
4. **Scalability:**
    - **Systems can scale by adding more processors or nodes to handle larger workloads.**
    - **Example: Cloud computing infrastructure.**
5. **Energy Efficiency:**
    - **Tasks finish faster, reducing overall power consumption when compared to sequential processing for the same workload.**
6. **Fault Tolerance (in Distributed Systems):**
    - **In some parallel systems (e.g., Hadoop clusters), failures in individual nodes do not stop the entire system.**

Cons of Parallel Computing

1. **Complexity of Development:**
    - **Writing parallel programs is more complex due to the need for task division, synchronization, and data sharing.**
    - **Bugs like race conditions, deadlocks, and resource contention are common.**
2. **Overhead from Communication:**
    - **Coordination and communication between processors add overhead, reducing efficiency.**
    - **Example: Dividing and merging tasks in distributed computing.**
3. **Not All Tasks are Parallelizable:**
    - **Some tasks have sequential dependencies (e.g., A must finish before B starts), which limits parallelization.**
    - **Amdahl's Law highlights this limitation.**

4. **Hardware Dependency:**
    - **Requires specialized hardware (multi-core processors, GPUs, or clusters).**
    - **Cost of multi-processor systems can be high.**
5. **Load Balancing Issues:**
    - **Uneven distribution of tasks may cause some processors to sit idle while others are overloaded.**
6. **Memory Overhead:**
    - **Managing data across multiple cores or processors can increase memory usage and latency.**

Example of Parallel Computing

- **Sorting a List:**
    - **Sequential: Sort the list from start to finish in one step.**
    - **Parallel: Divide the list into smaller parts, sort them simultaneously on multiple processors, then merge the results.**
- **Real-World Applications:**
    - **Machine learning training (using GPUs).**
    - **Weather forecasting simulations.**
    - **Video rendering or large-scale image processing.**
    - **Distributed systems like Google's MapReduce.**

When to Use Parallel Computing

1. **Compute-Intensive Tasks: Large-scale simulations, rendering, and scientific calculations.**
2. **Big Data Processing: Systems like Hadoop and Spark.**
3. **Applications with Independent Tasks: Workloads that can be broken into smaller, parallelizable units.**

Parallel computing offers immense benefits for performance, scalability, and resource utilization, especially for large-scale and complex problems. However, it introduces development complexity and hardware dependency. By understanding its trade-offs, developers can determine when to use parallel computing effectively.

Prepared by: hjt

Subtopic 2

Amdahl's Law

Amdahl's Law provides a theoretical framework to understand the maximum speedup that can be achieved by parallelizing a program. It highlights the limitations of parallel computing, emphasizing that the sequential portion of a program restricts the overall performance improvement.

What is Amdahl's Law?

- **Definition: The theoretical speedup of a program using parallel computing is limited by the fraction of the program that cannot be parallelized.**
- **Formula:**

$$\text{Speedup} = \frac{1}{(1 - P) + \frac{P}{N}}$$

  - P **= Fraction of the program that can be parallelized.**
  - N **= Number of processors.**

Example:

- **Task: Searching through a pile of index cards.**
  - **Sequential Part: Dividing the cards and collecting results (1 second).**
  - **Parallel Part: Searching through the cards (3 seconds).**
  - **Total Time: 5 seconds (sequential) vs. 3 seconds (parallel with 2 friends).**
  - **Speedup: 40%.**

Amdahl's Law states that the speedup of a program using parallel computing is limited by the fraction of the program that cannot be parallelized.

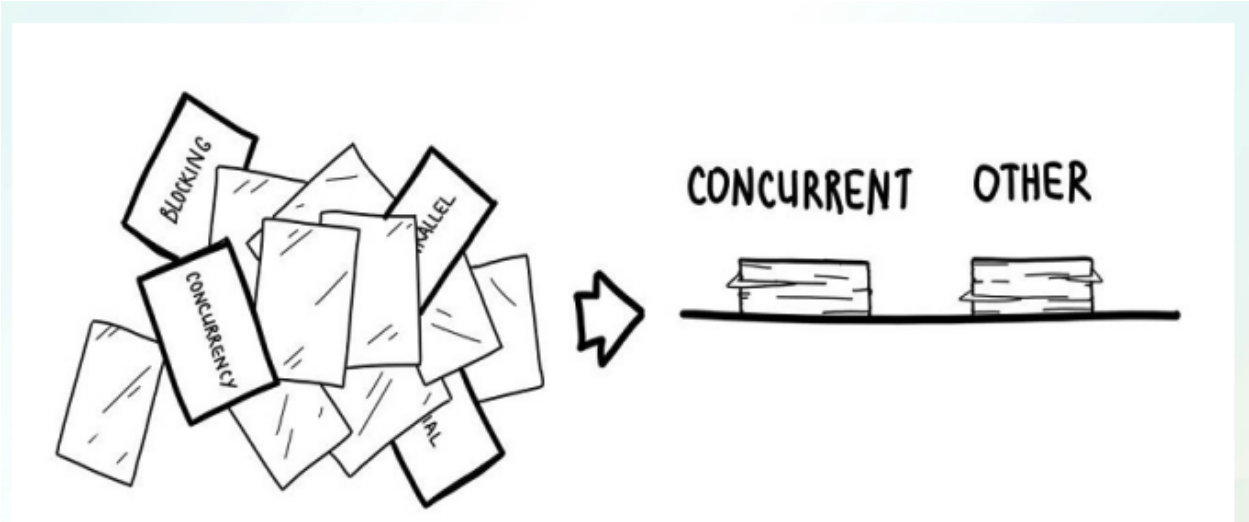
The example of searching through index cards illustrates how Amdahl's Law works. The speedup is limited by the sequential parts of the task, even as more processors are added.

Amdahl's Law tells us that when we try to speed up a program by using multiple processors, the speedup is limited by the parts of the program that cannot be parallelized (the sequential parts).

Imagine you have a pile of index cards, and you need to find specific cards. The task has three steps:

1. **Divide the cards (1 second).**
2. **Search through the cards (3 seconds).**
3. **Collect the results (1 second).**

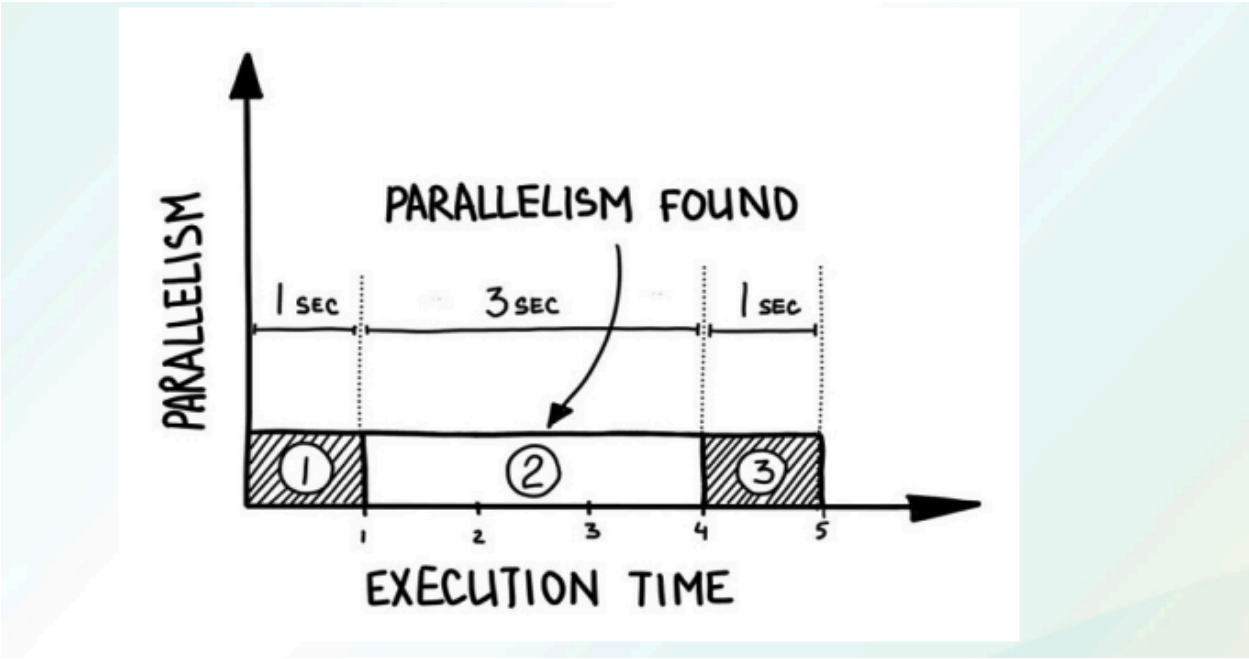
- **The total time if you do it alone is 5 seconds.**



The task can be broken into:

1. **Serial Part 1: Dividing the cards (1 second).**
2. **Parallel Part: Searching through the cards (3 seconds).**
3. **Serial Part 2: Collecting the results (1 second).**

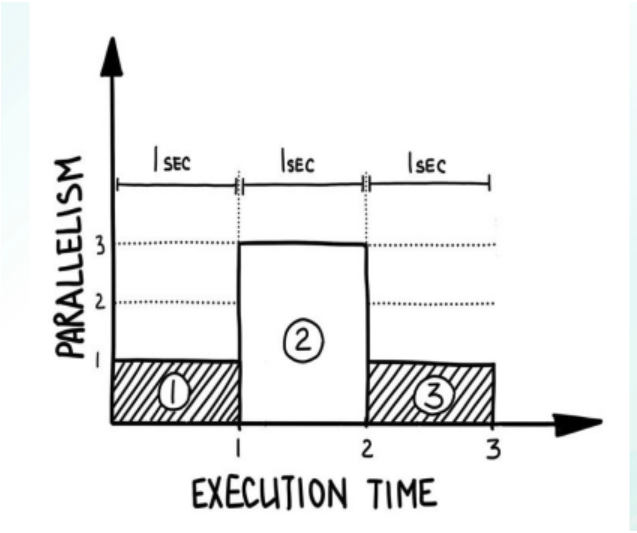
- **The serial parts (dividing and collecting) cannot be sped up, even with more people.**



If you do the task alone, it takes 5 seconds. But if you get friends to help with the searching part, you can reduce the time. However, the dividing and collecting parts will still take 1 second each.

If you have 2 friends help with the searching, the 3-second search can be reduced to 1.5 seconds. So the total time becomes:

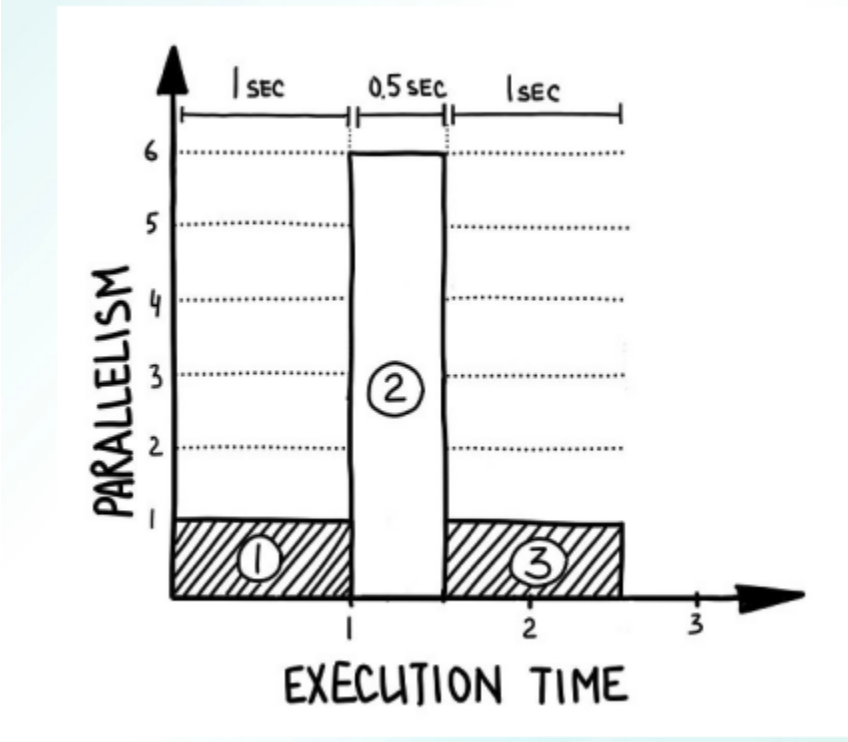
- **1 (dividing) + 1.5 (searching) + 1 (collecting) = 3.5 seconds.**



Prepared by: hjt

If you have 2 friends and reduce the searching time to 1 second, the total time becomes:

- **1 (dividing) + 1 (searching) + 1 (collecting) = 3 seconds.**
- **This is a 40% speedup compared to the original 5 seconds.**



If you add more friends, say 6 friends, the searching time can be reduced to 0.5 seconds. So the total time becomes:

- **1 (dividing) + 0.5 (searching) + 1 (collecting) = 2.5 seconds.**
- **This is a 50% speedup compared to the original 5 seconds.**

Even if you add more friends, the total time cannot be less than 2 seconds because the dividing and collecting parts take 1 second each and cannot be sped up.

Adding too many friends might actually slow things down because of the time it takes to coordinate and communicate between them.

The key takeaway is that no matter how many friends (processors) you add, the total time cannot be less than the time it takes to do the serial parts (dividing and collecting).

Prepared by: hjt

Example 1: Video Rendering

Scenario:

You're rendering a video, and the process consists of:

1. **Loading the Project (Sequential):** Takes 2 minutes.
2. **Rendering the Video (Parallel):** Takes 10 minutes on a single processor.
3. **Saving the Output (Sequential):** Takes 3 minutes.

Applying Amdahl's Law:

- **Total Sequential Time:** 2 (loading) + 10 (rendering) + 3 (saving) = **15 minutes**.
- **Parallel Part (P):** Rendering the video (10 minutes out of 15 minutes).

$$P = \frac{10}{15} \approx 0.67$$

- **Number of Processors (N):** Let's say you use **4 processors**.

Calculating Speedup:

$$\text{Speedup} = \frac{1}{(1 - 0.67) + \frac{0.67}{4}} = \frac{1}{0.33 + 0.1675} = \frac{1}{0.4975} \approx 2.01$$

- **Interpretation:** The task will run **2.01 times faster** with 4 processors compared to doing it sequentially.

Total Time with 4 Processors:

- **Parallel Rendering Time:** 10 minutes / 4 = 2.5 minutes.
- **Total Time:** 2 (loading) + 2.5 (rendering) + 3 (saving) = **7.5 minutes**.

Example 2: Searching Through a Database

Scenario:

You need to search through a large database for specific records. The process consists of:

1. **Setting Up the Query (Sequential):** Takes 1 minute.
2. **Searching the Database (Parallel):** Takes 5 minutes on a single processor.
3. **Displaying the Results (Sequential):** Takes 1 minute.

Applying Amdahl's Law:

- **Total Sequential Time:** 1 (setup) + 5 (searching) + 1 (displaying) = **7 minutes**.
- **Parallel Part (P):** Searching the database (5 minutes out of 7 minutes).

$$P = \frac{5}{7} \approx 0.71$$

- **Number of Processors (N):** Let's say you use **5 processors**.

Calculating Speedup:

$$\text{Speedup} = \frac{1}{(1 - 0.71) + \frac{0.71}{5}} = \frac{1}{0.29 + 0.142} = \frac{1}{0.432} \approx 2.31$$

- **Interpretation:** The task will run **2.31 times faster** with 5 processors compared to doing it sequentially.

Total Time with 5 Processors:

- **Parallel Searching Time:** 5 minutes / 5 = 1 minute.
- **Total Time:** 1 (setup) + 1 (searching) + 1 (displaying) = **3 minutes**.

Example 3: Image Processing

Scenario:

You're processing a large image, and the process consists of:

1. **Loading the Image (Sequential):** Takes 1 minute.
2. **Applying Filters (Parallel):** Takes 4 minutes on a single processor.
3. **Saving the Processed Image (Sequential):** Takes 1 minute.

Applying Amdahl's Law:

- **Total Sequential Time:** 1 (loading) + 4 (applying filters) + 1 (saving) = **6 minutes**.
- **Parallel Part (P):** Applying filters (4 minutes out of 6 minutes).

$$P = \frac{4}{6} \approx 0.67$$

- **Number of Processors (N):** Let's say you use **4 processors**.

Calculating Speedup:

$$\text{Speedup} = \frac{1}{(1 - 0.67) + \frac{0.67}{4}} = \frac{1}{0.33 + 0.1675} = \frac{1}{0.4975} \approx 2.01$$

- **Interpretation:** The task will run **2.01 times faster** with 4 processors compared to doing it sequentially.

Total Time with 4 Processors:

- **Parallel Filtering Time:** 4 minutes / 4 = 1 minute.
- **Total Time:** 1 (loading) + 1 (applying filters) + 1 (saving) = **3 minutes**.