PARALLEL AND DISTRIBUTED COMPUTING
LECTURE # 1

By,

Dr. Ali Akbar Siddique

INTRODUCTION TO PARALLEL AND DISTRIBUTED COMPUTING

- **Definition of Parallel Computing**: Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously, leveraging multiple processing units.
- **Definition of Distributed Computing**: Distributed computing refers to a model where computing tasks are divided across multiple interconnected computers, working together to achieve a common goal.
- **Importance in Modern Computing**:
  - **Performance Enhancement**: Enables faster processing by distributing workloads.
  - **Scalability**: Easily handles increasing amounts of work by adding more computing resources.
  - **Fault Tolerance**: Reduces the risk of system failure by distributing tasks across multiple machines.
  - **Efficiency in Resource Utilization**: Optimizes the use of computational power across various hardware components.

KEY CONCEPTS

- **Concurrency**:
  - Concurrency is the ability of a system to handle multiple tasks in progress at the same time.
  - Tasks may not necessarily be executed simultaneously but make progress together.
  - Example: Running multiple applications on a computer, where the CPU switches between tasks.
- **Parallelism**:
  - Parallelism refers to the simultaneous execution of multiple tasks to achieve faster processing.
  - Requires multiple processing units (e.g., multi-core processors, GPUs).
  - Example: Performing matrix operations using multiple CPU cores simultaneously.
- **Distributed Systems**:
  - A distributed system is a network of independent computers that collaborate to solve a problem.
  - Each machine works on a part of the problem and communicates with others.
  - Example: Cloud computing environments like AWS, Google Cloud, and Microsoft Azure.

CONCURRENCY VS. PARALLELISM

- **Concurrency**:
  - Concurrency refers to multiple tasks making progress at overlapping time periods.
  - The system rapidly switches between tasks, creating an illusion of simultaneous execution.
  - Example: A web server handling multiple client requests concurrently using asynchronous I/O.
- **Parallelism**:
  - Parallelism involves executing multiple tasks simultaneously.
  - Requires multiple cores or processing units to run computations in parallel.
  - Example: Image processing using a GPU, where different sections of an image are processed simultaneously.

CONCURRENCY VS. PARALLELISM

Aspect	Concurrency	Parallelism	Aspect
Execution	Tasks appear to run at the same time but are actually interleaved	Tasks run at the same time on multiple processors	Execution
Dependencies	Often involves shared resources requiring synchronization	Tasks are independent and run without interference	Dependencies
Example	A user switching between multiple browser tabs	A video rendering engine using multi-core processing	Example

DISTRIBUTED SYSTEMS

- A distributed system consists of multiple independent computers working together as a unified system.
- These computers communicate over a network to share resources, perform computations, or provide services.

Characteristics:

- **Decentralization**: No single point of control; workload is distributed among multiple nodes.
- **Scalability**: Can handle increasing workloads by adding more machines.
- **Fault Tolerance**: If one node fails, others can continue functioning, improving system reliability.

Key Components:

- **Nodes**: Individual computing devices participating in the system.
- **Communication Network**: Connects the nodes for data exchange.
- **Middleware**: Software layer enabling coordination between distributed resources.

Examples:

- **Cloud Computing**: Services like AWS, Google Cloud, and Azure use distributed infrastructure.
- **Blockchain Networks**: Bitcoin and Ethereum operate on decentralized distributed systems.
- **Distributed Databases**: Systems like Apache Cassandra and Google Spanner ensure high availability and scalability.

WHY PARALLEL AND DISTRIBUTED COMPUTING?

1. Performance Enhancement

- Traditional sequential processing has limitations in speed and efficiency.
- Parallel and distributed computing enable tasks to be executed simultaneously, reducing computation time.
- **Example**: A weather simulation that would take 10 hours on a single CPU can be completed in minutes using parallel processing across multiple processors.

- **2. Efficient Resource Utilization**

- By distributing workloads across multiple processors or systems, computational resources are used optimally.
- Avoids bottlenecks by balancing loads across different machines.
- **Example**: Cloud computing providers allocate virtual machines dynamically to optimize performance and reduce idle hardware usage.

3. Scalability for Large-Scale Computations

- Parallel and distributed systems can scale horizontally (by adding more machines) or vertically (by increasing processing power).
- Handles massive datasets and complex computations effectively.
- **Example**: Google's search engine processes billions of search queries daily by distributing tasks across thousands of servers worldwide.

WHY PARALLEL AND DISTRIBUTED COMPUTING?

4. Fault Tolerance and Reliability

- Distributed systems ensure redundancy by replicating data and processing tasks across multiple nodes.
- If a node fails, others can take over, minimizing downtime.
- **Example**: Netflix uses a distributed cloud-based infrastructure to ensure seamless video streaming even if some servers fail.

5. Cost-Effectiveness

- Organizations can use distributed computing with commodity hardware instead of expensive supercomputers.
- Cloud platforms offer scalable computing resources on-demand, reducing costs.
- **Example**: A startup can use Amazon Web Services (AWS) to rent computing power instead of investing in costly data centers.

REAL-WORLD APPLICATIONS OF PARALLEL AND DISTRIBUTED COMPUTING

1. Big Data Processing

- The increasing volume of data requires powerful computational methods to process and analyze massive datasets efficiently.
- Parallel and distributed computing enable real-time data processing and analytics.
- **Technologies Used:**
- **Apache Hadoop**: Uses the MapReduce framework to distribute data processing across clusters of computers.
- **Apache Spark**: Performs in-memory distributed computing for faster data analysis.
- **Example:**
- **Netflix** uses Hadoop and Spark to analyze user preferences and provide personalized movie recommendations.

REAL-WORLD APPLICATIONS OF PARALLEL AND DISTRIBUTED COMPUTING

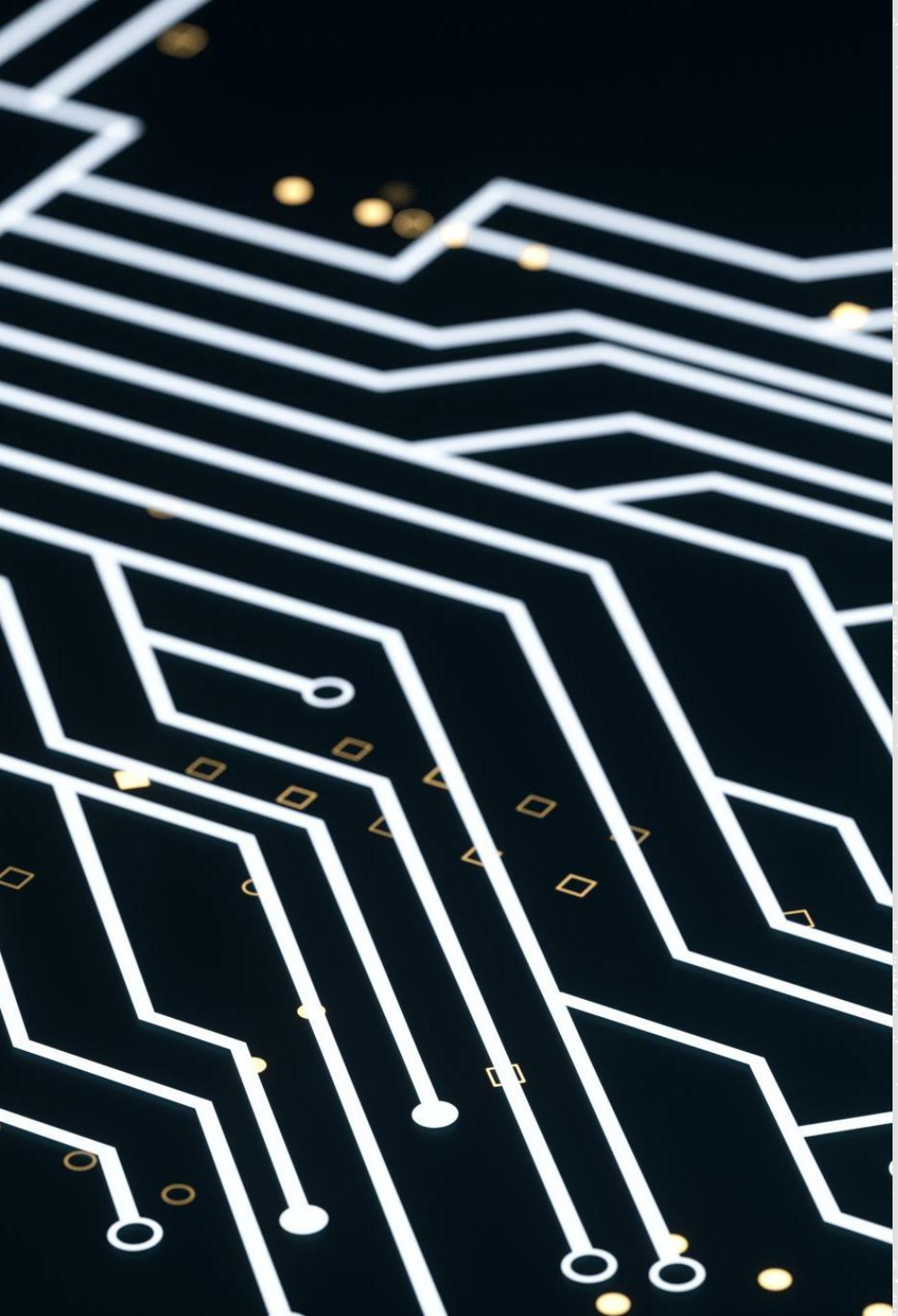
2. Artificial Intelligence (AI) and Machine Learning

- Training deep learning models requires processing large datasets and performing complex mathematical computations.
- GPUs and TPUs (Tensor Processing Units) enable parallel computation, reducing training time.
- **Technologies Used:**
- **GPUs (CUDA, TensorFlow, PyTorch)**: Perform matrix multiplications in parallel.
- **Distributed Deep Learning**: Frameworks like Horovod and TensorFlow Distributed speed up training across multiple machines.
- **Example:**
- **OpenAI's ChatGPT** and Google's **DeepMind AlphaGo** use distributed computing to train AI models on massive datasets.

REAL-WORLD APPLICATIONS OF PARALLEL AND DISTRIBUTED COMPUTING

3. Cloud Computing

- Cloud platforms use distributed computing to provide on-demand computing resources.
- Users can scale applications dynamically without investing in expensive hardware.
- **Cloud Service Providers:**
- **Amazon Web Services (AWS)**: Offers computing power (EC2), storage (S3), and databases.
- **Google Cloud Platform (GCP)**: Provides AI and machine learning services.
- **Microsoft Azure**: Supports virtual machines and enterprise applications.
- **Example:**
- **Instagram, Dropbox, and Spotify** use cloud computing to store and process massive amounts of user data across multiple servers.

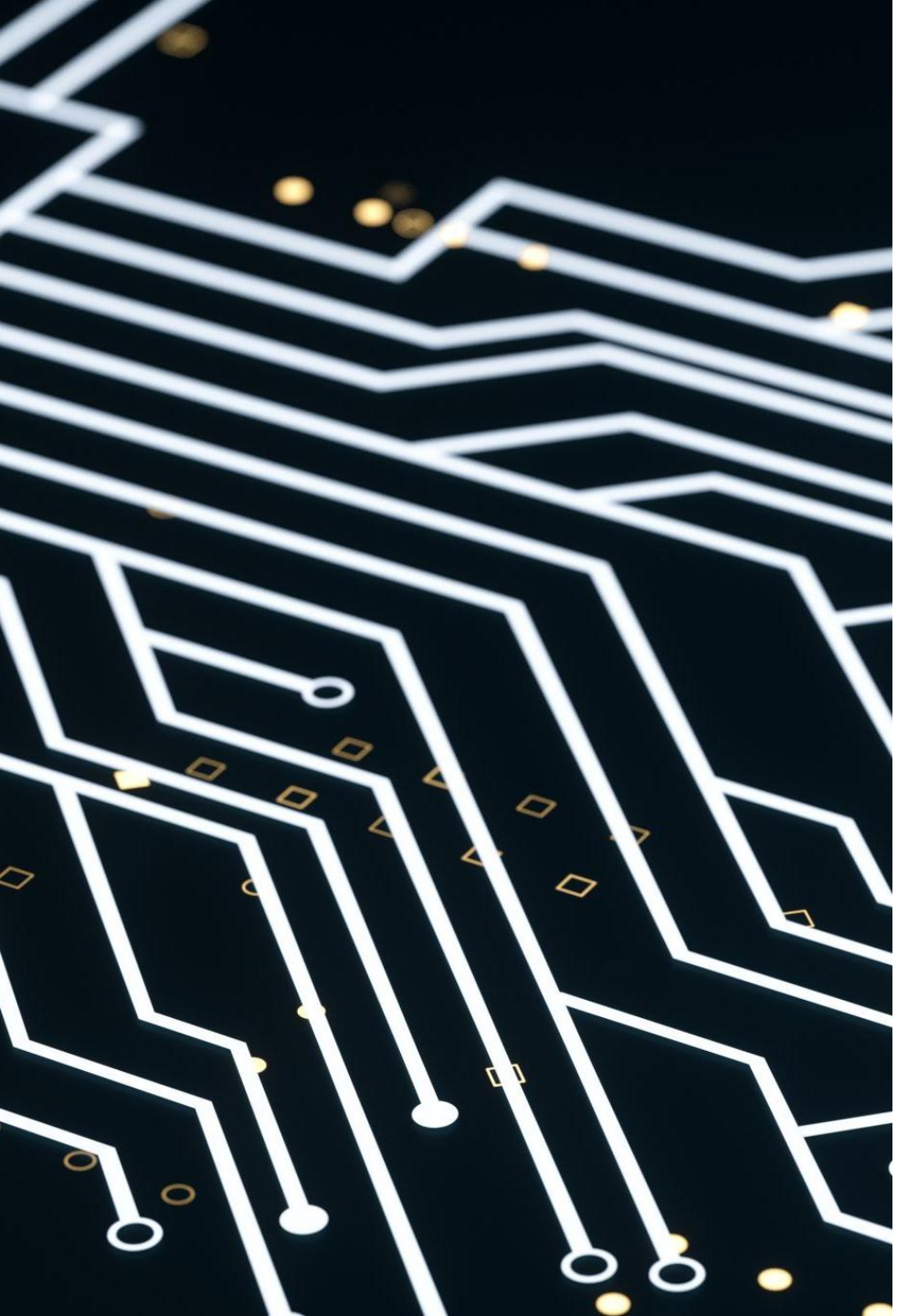


PARALLEL AND DISTRIBUTED COMPUTING IN AI

- Artificial Intelligence (AI) requires enormous computational power to train deep learning models and process vast amounts of data. Parallel and distributed computing enable efficient AI training, optimization, and real-time inference.

1. Why AI Needs Parallel and Distributed Computing?

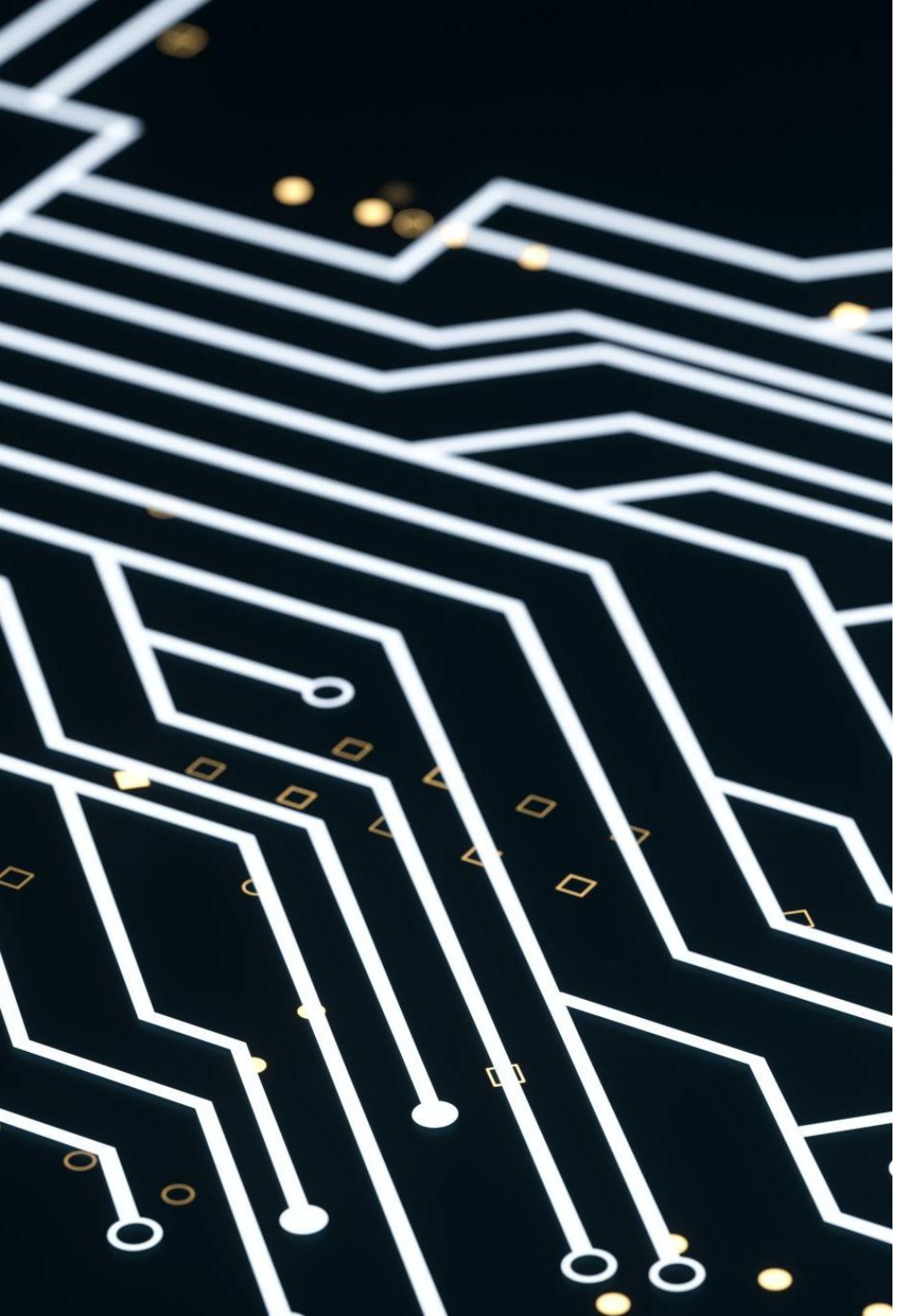
- **Massive Data Processing**: AI models require large datasets (e.g., ImageNet, GPT datasets).
- **Computational Intensity**: Deep learning involves billions of mathematical operations.
- **Real-Time Inference**: AI applications like autonomous vehicles and facial recognition require fast decision-making.



PARALLEL AND DISTRIBUTED COMPUTING IN AI

2. Training Deep Learning Models with Parallel Computing

- Training AI models involves performing millions of matrix multiplications.
- GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units) execute these operations in parallel.
- **Key Technologies:**
- **CUDA (Compute Unified Device Architecture)**: Enables GPU acceleration for AI tasks.
- **TensorFlow and PyTorch**: Use multi-core processing for deep learning.
- **NVIDIA Tensor Cores**: Optimize AI computations for faster training.
- **Example:**
- **Google's AlphaGo** used parallel computing with TPUs to train its reinforcement learning model.



PARALLEL AND DISTRIBUTED COMPUTING IN AI

3. AI in Edge and IoT Devices

- AI models can be optimized for edge devices (smartphones, IoT sensors).
- Edge AI reduces reliance on cloud computing by processing data locally.
- **Example:**
- **Tesla's Autopilot** uses parallel processing in AI chips for real-time driving decisions.

Tool/Framework	Purpose	Example Use Case
CUDA (Compute Unified Device Architecture)	Enables parallel computing using NVIDIA GPUs.	Deep learning model training with TensorFlow.
MPI (Message Passing Interface)	Manages communication in distributed systems.	Weather forecasting simulations using supercomputers.
OpenMP (Open Multi-Processing)	Supports shared-memory parallelism in C, C++, and Fortran.	Multi-threaded scientific computing applications.
Apache Hadoop	Distributed storage and processing for big data.	Processing large datasets in cloud environments.
Apache Spark	In-memory distributed computing for big data analytics.	Real-time data processing for financial transactions.
Ray	Parallel computing framework for AI and machine learning workloads.	Large-scale reinforcement learning training.

TOOLS FOR PARALLEL AND DISTRIBUTED COMPUTING

- Parallel and distributed computing rely on specialized tools and frameworks to manage computations efficiently. These tools enable developers to optimize performance, handle large-scale data, and coordinate distributed tasks.
- Several tools and frameworks are commonly used in parallel and distributed computing. Each tool serves a different purpose, from GPU acceleration to large-scale distributed processing.

KEY DIFFERENCES BETWEEN THE TOOLS

Aspect	CUDA	MPI	OpenMP	Hadoop	Spark	Ray
Computing Model	Parallel (GPU)	Distributed	Parallel (CPU)	Distributed	Distributed	Parallel & Distributed
Best For	AI, Deep Learning	Supercomputing	Scientific Computing	Big Data Processing	Real-Time Big Data	AI & ML Workloads
Memory Access	Shared GPU Memory	Distributed Memory	Shared Memory	Distributed Storage	In-Memory Processing	Distributed Execution
Ease of Use	Moderate	Complex	Easy	Moderate	Easy	Easy
Scalability	High	High	Medium	Very High	Very High	High

WHEN TO USE EACH TOOL

- **CUDA** → Best for AI, deep learning, and GPU-accelerated computations.
- **MPI** → Best for large-scale **distributed simulations** in scientific research.
- **OpenMP** → Best for multi-threaded **CPU-based parallel processing**.
- **Hadoop** → Ideal for **big data storage and batch processing**.
- **Spark** → Suitable for **real-time analytics and big data workflows**.
- **Ray** → Ideal for **AI model training, reinforcement learning, and distributed Python workloads**.


thank
you

Parallel and Distributed Computing
Lecture # 2

By,

Dr. Ali Akbar Siddique

Introduction to Hardware Architectures for Parallel Computing

- Parallel computing relies on specialized hardware to achieve high performance.
- Different architectures are designed to support parallelism at various levels.
- This lecture covers:
  - Multi-core and many-core architectures
  - GPU architectures
  - Shared vs. distributed memory models

Multi-core vs. Many-core Architectures

Multi-core Architectures

- A multi-core processor is a single computing component with two or more independent processing units (cores). Each core can execute instructions independently, allowing for parallel execution.
- **Characteristics**:

  •Typically **2 to 16 cores** in modern CPUs.

  •Each core has its own **L1 cache** but may share L2/L3 caches.

  •Designed for **general-purpose computing** and optimized for single-threaded performance.

  •Common in **desktops, laptops, and servers**.
- **Example**:

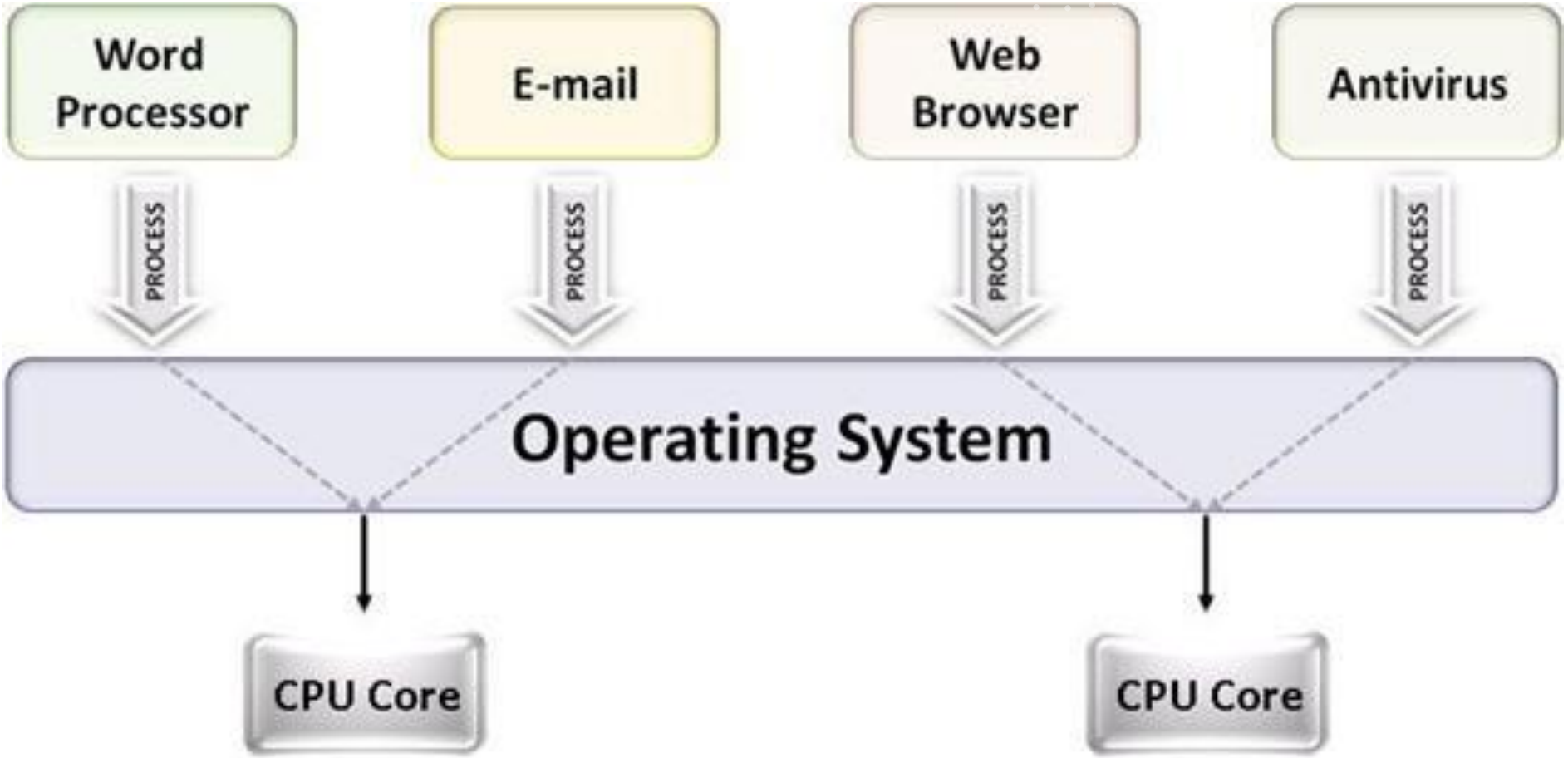
  •Intel Core i7-12700K (12 cores, 20 threads)

  •AMD Ryzen 9 5900X (12 cores, 24 threads)
- **Use Case**:

  •Running multiple applications (e.g., web browsing, video streaming, gaming).

  •Multitasking (e.g., running a code compiler while editing a document).

Task Distribution in Multi-Core Processor



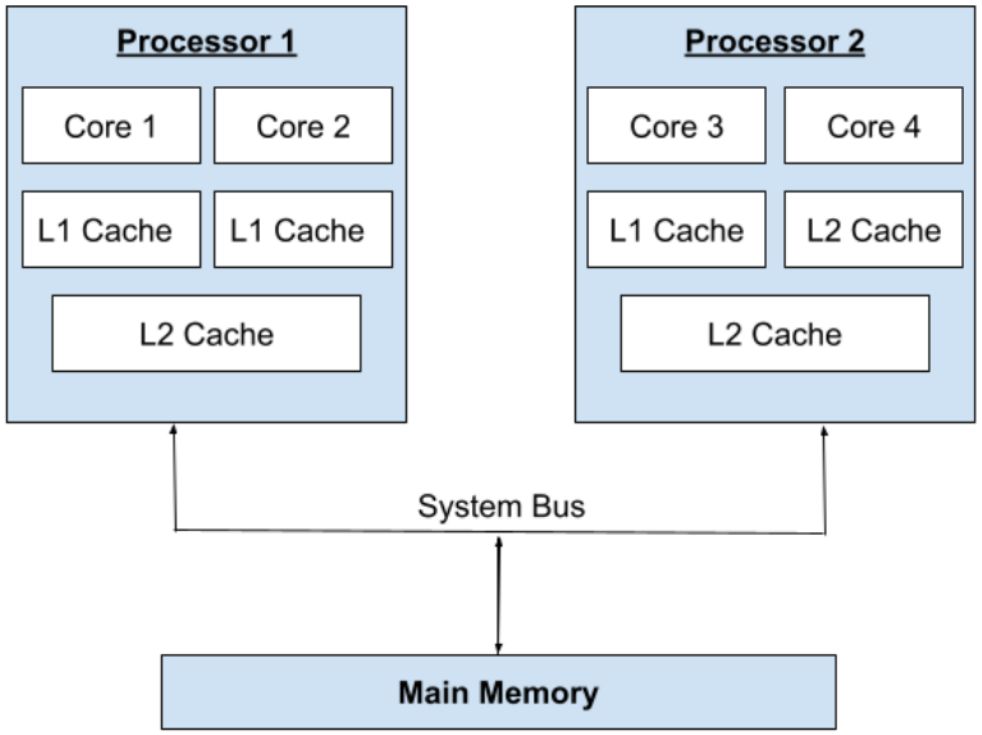
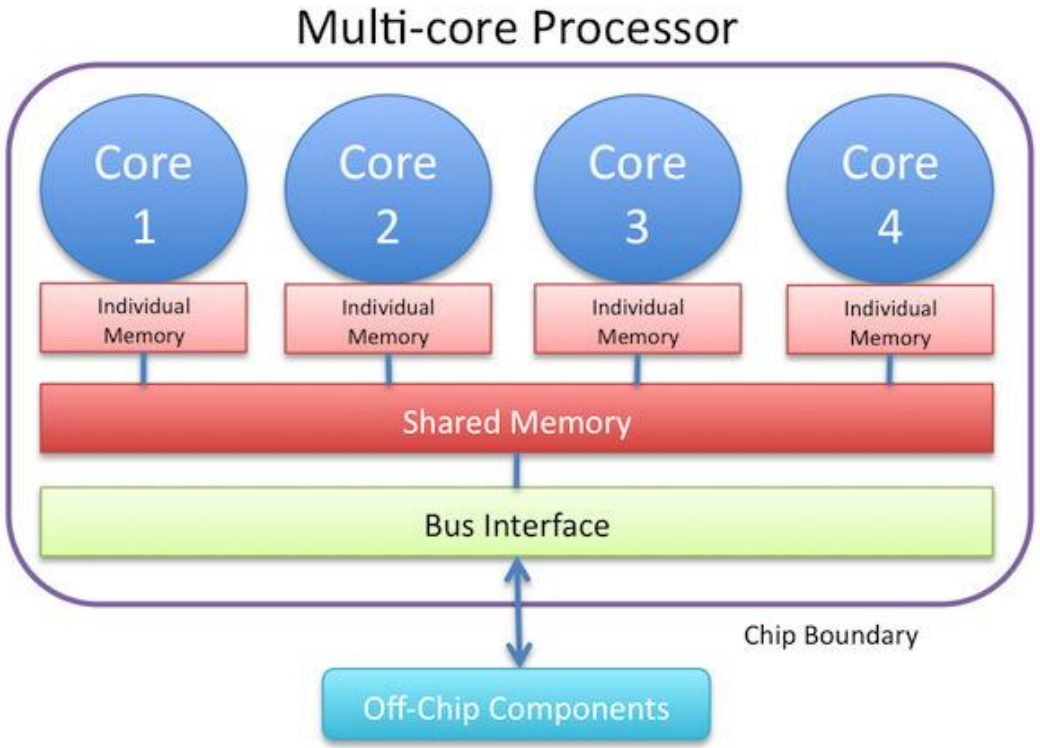


Fig. 2: Multi-core Processor Architecture



Cache Memory – L1, L2, and L3

1. What is Cache Memory?

- Cache memory is a small-sized, high-speed memory located close to the CPU.
- It stores frequently accessed data and instructions to **reduce latency** and improve processing speed.
- Faster than RAM but smaller in size.

2. Levels of Cache (L1, L2, L3)

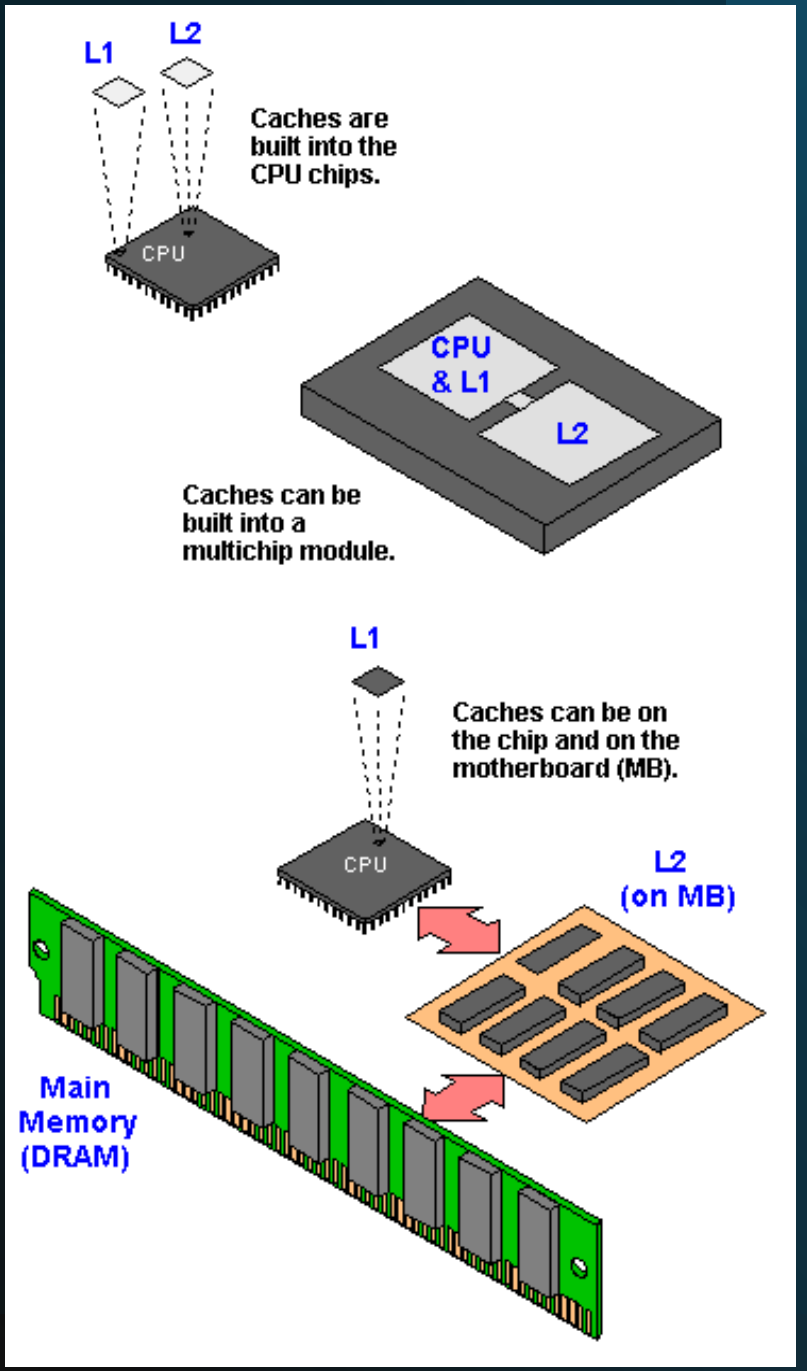
- Modern processors use a multi-level cache hierarchy to balance speed and size.

Cache Hierarchy

Cache Level	Size (Typical)	Speed (Latency)	Location	Function
L1 (Level 1)	32KB - 1MB	Fastest (1-4 cycles)	Inside CPU core	Stores most frequently used instructions & data
L2 (Level 2)	256KB - 8MB	Slower than L1 (5-20 cycles)	Shared per core or per group	Holds data evicted from L1
L3 (Level 3)	4MB - 64MB	Slowest (20-50 cycles)	Shared across all cores	Reduces access time for RAM

Cache Levels – L1

- **L1 Cache (First Level Cache)**
- Smallest but **fastest** cache (closest to the CPU).
- Stores **immediate instructions** and frequently accessed data.
- Typically divided into:
  - **L1 Instruction Cache (L1i)** – Stores CPU instructions.
  - **L1 Data Cache (L1d)** – Stores frequently used data.
- **Example:**
- When a CPU fetches an instruction, it first looks in the **L1 cache** before going to L2 or RAM.



Cache Levels – L2

L2 Cache (Second Level Cache)

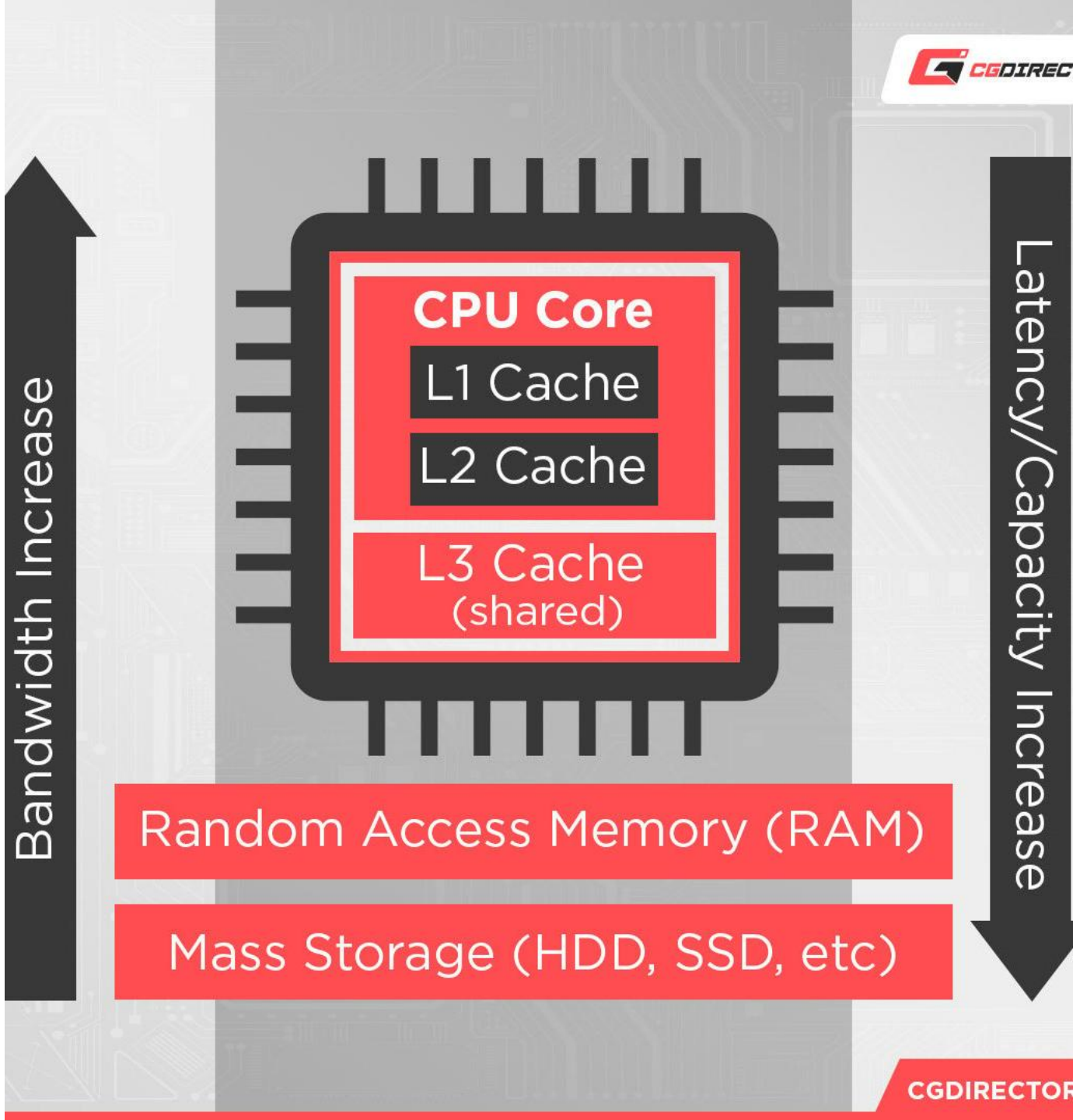
Larger than L1 but slightly slower.

Stores data that doesn't fit in L1.

Can be dedicated per core or shared among multiple cores.

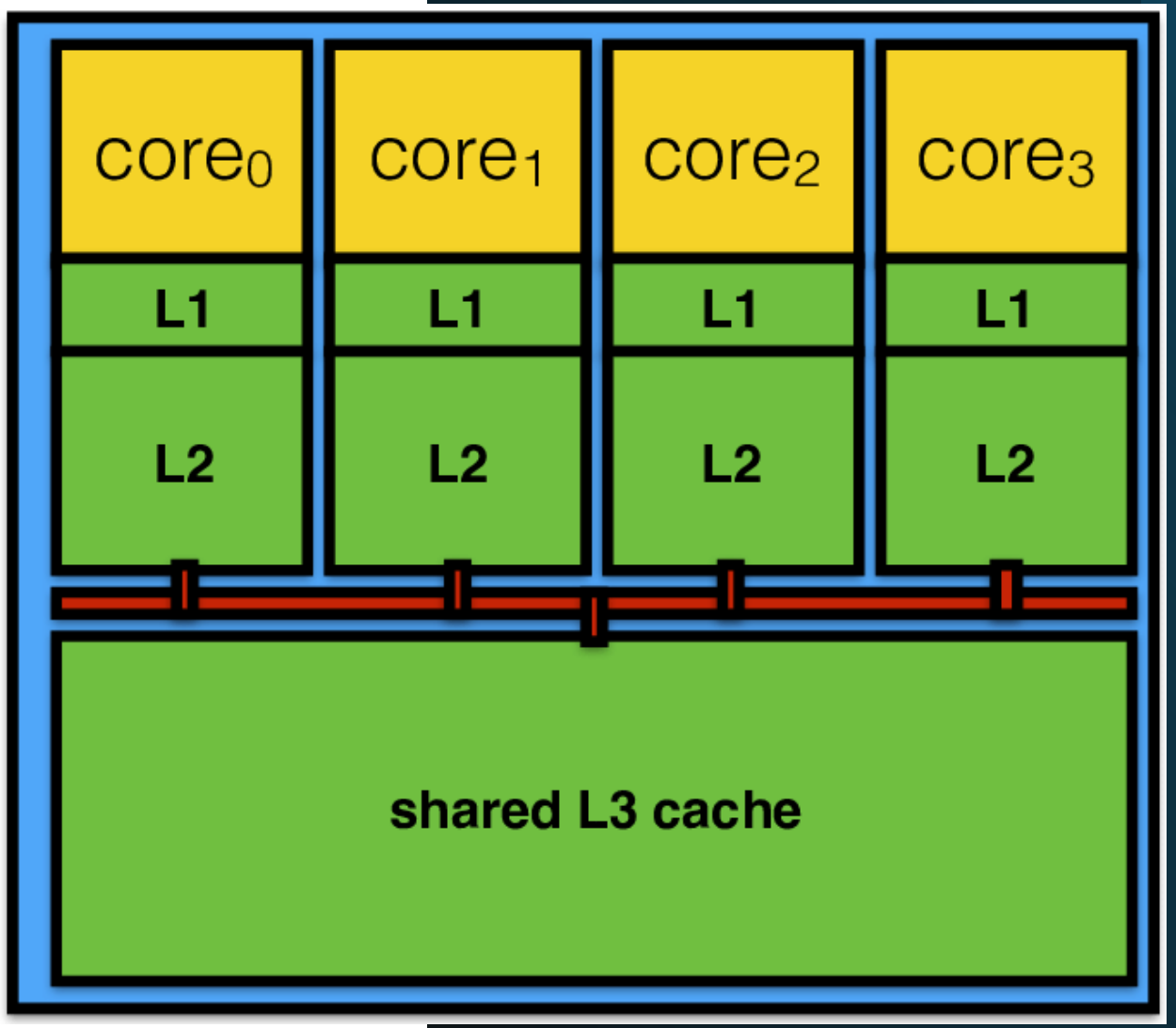
Example:

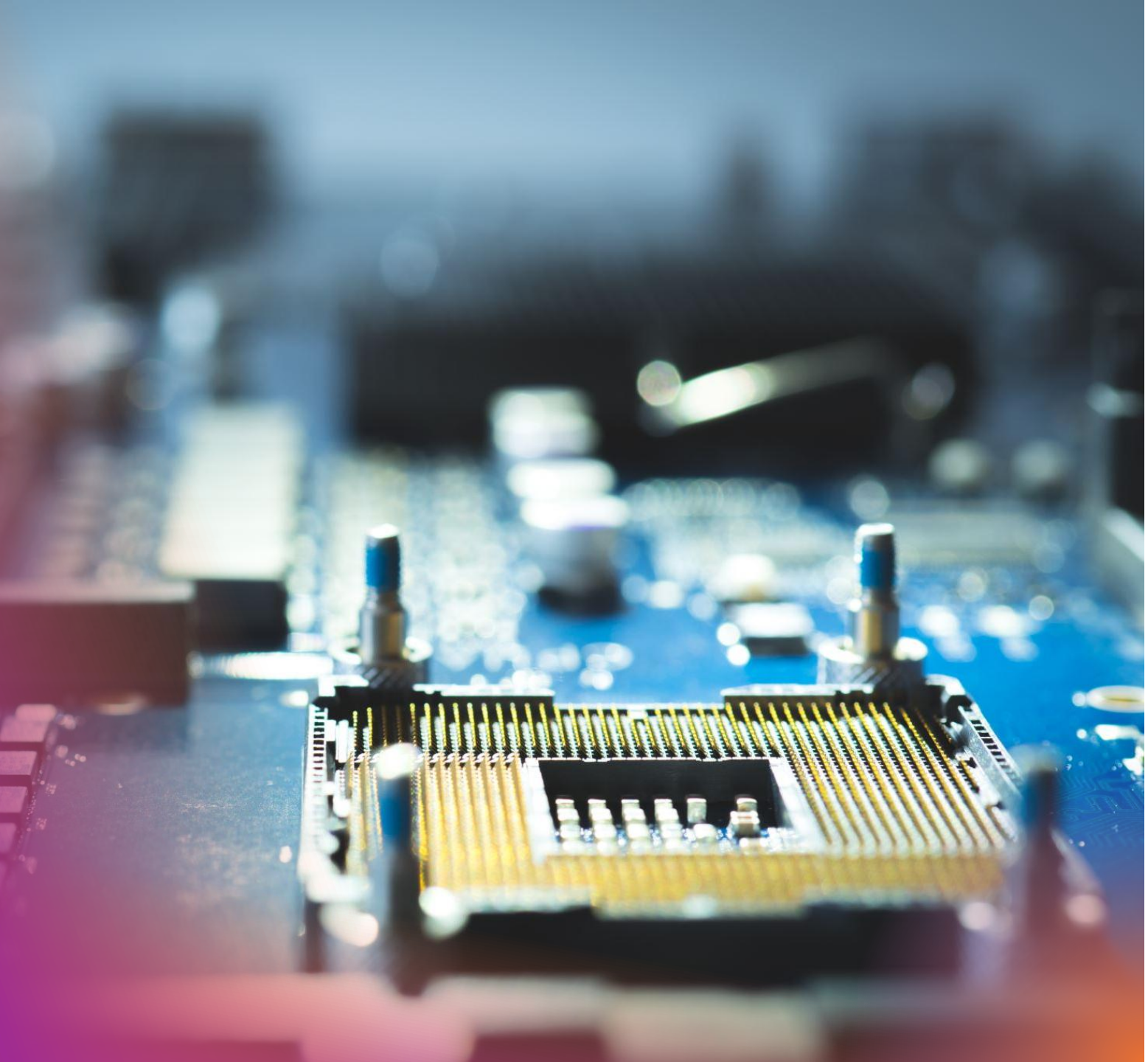
- If L1 cache misses a piece of data, it checks L2 before accessing RAM.



Cache Levels – L3

- **L3 Cache (Third Level Cache)**
  - **Largest and slowest** cache level but **still much faster than RAM**.
  - Typically **shared across all cores** in multi-core CPUs.
  - Reduces **bottlenecks** when multiple cores request the same data.
- **Example:**
  - In gaming or video rendering, **L3 cache helps multiple cores** access frequently used textures or frame data.





Cache Performance Impact

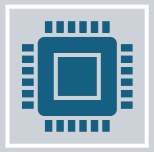
- A well-optimized cache system reduces CPU-RAM interaction, improving processing speed.
- More L1 and L2 cache means faster response times for critical operations.
- L3 cache helps in multi-threaded performance (e.g., AI training, video editing).

Real-World Example (CPU Cache Sizes)

Processor	L1 Cache	L2 Cache	L3 Cache
Intel Core i9-13900K	80KB per core	2MB per core	36MB shared
AMD Ryzen 9 7950X	64KB per core	1MB per core	64MB shared
Apple M2	192KB per core	2MB per core	16MB shared

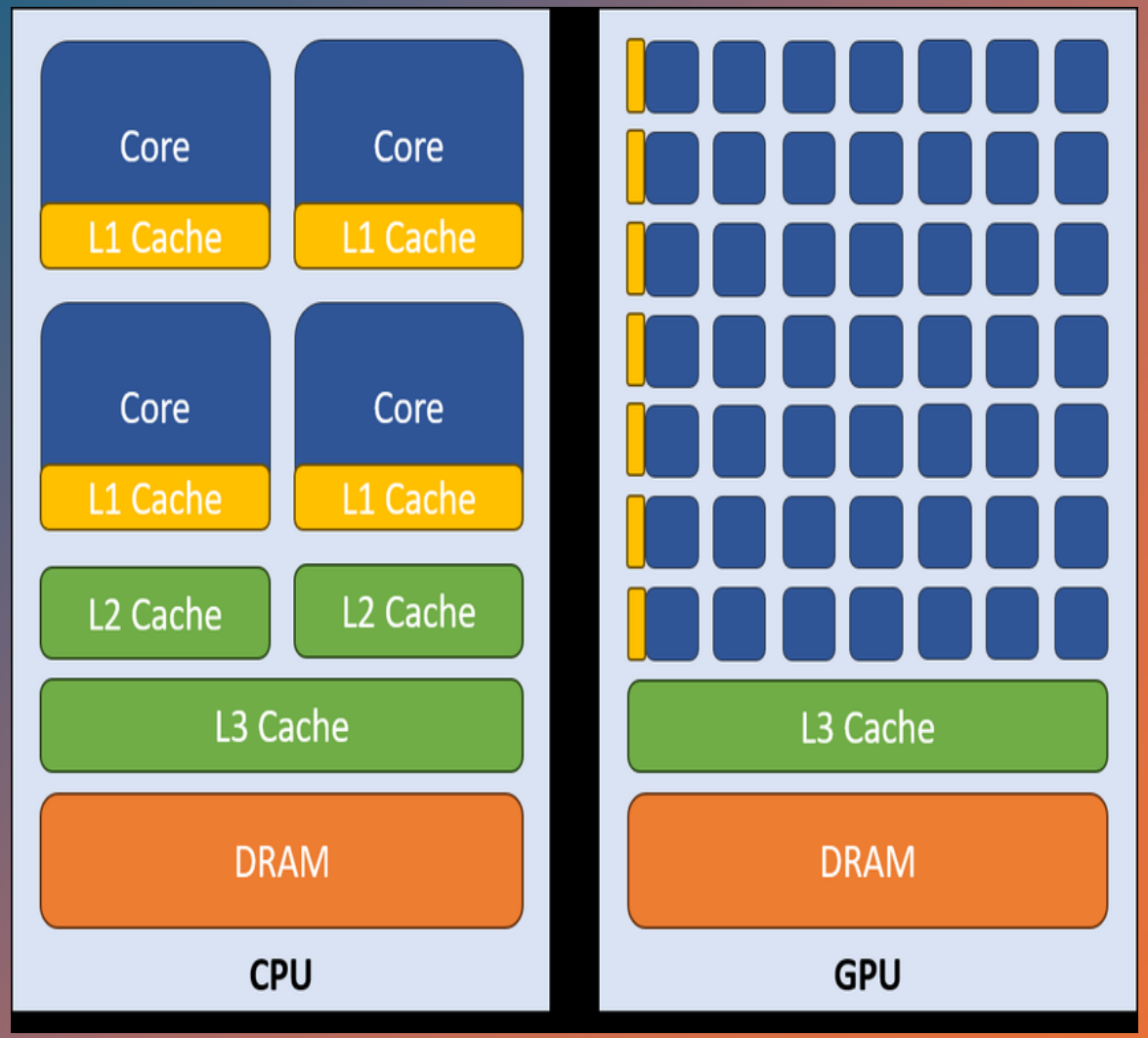
Analogy for Better Understanding

L1 Cache → Like a notepad on your desk (very fast, very small).



L2 Cache → Like a bookshelf next to your desk (a bit slower but larger).

L3 Cache → Like a library in your office building (much larger but slower to access).

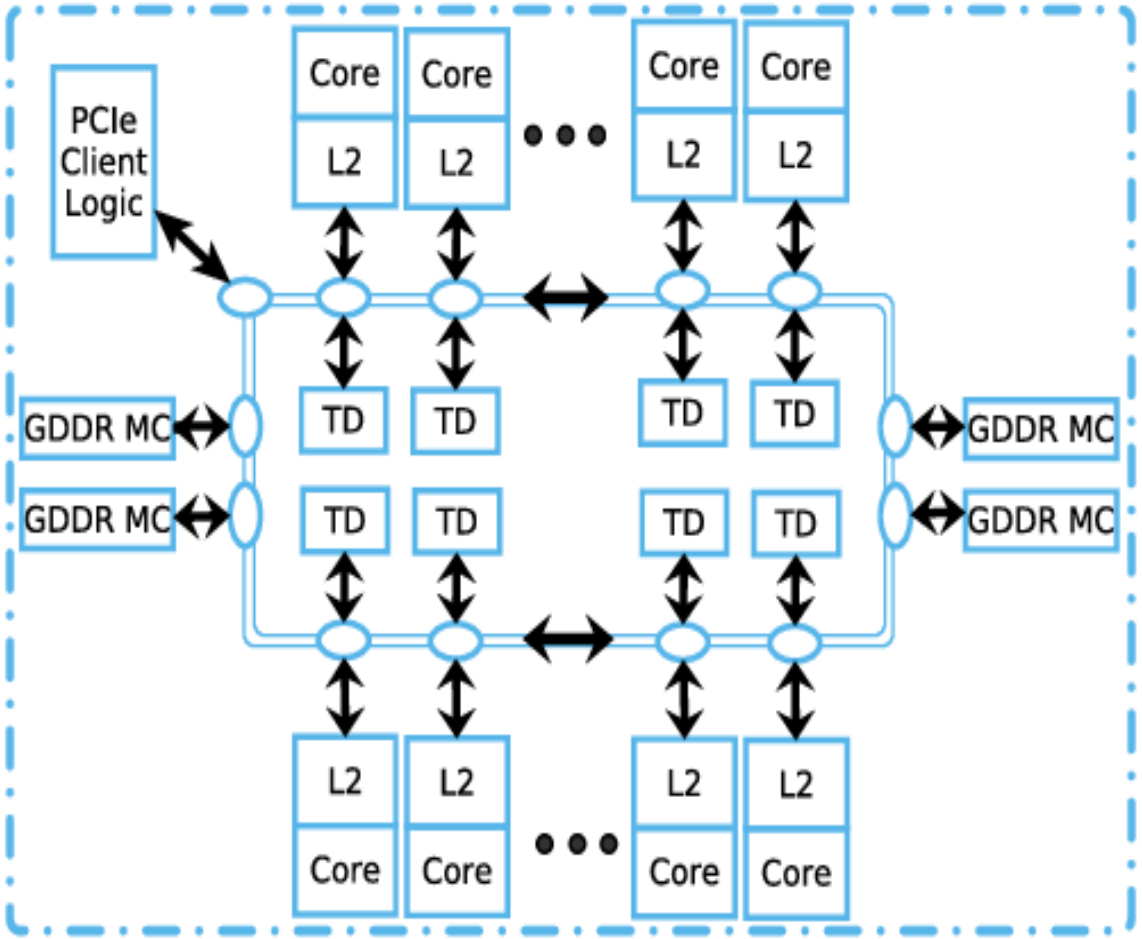


Many-core Architecture

- A many-core processor contains a large number of cores (typically dozens to hundreds) optimized for highly parallel workloads. These architectures prioritize parallelism over single-threaded performance.
- **Characteristics**:
  - Designed for **massive parallelism**.
  - Typically found in **GPUs and specialized accelerators**.
  - Cores may be **simpler and lower power** compared to multi-core CPUs.
  - Used in **scientific computing, AI, and simulations**.
- **Example:**
  - NVIDIA A100 GPU (6912 CUDA cores, 432 Tensor cores)
  - AMD Instinct MI250X (13,312 Stream processors)
- **Use Case:**
  - Training deep learning models using TensorFlow or PyTorch.
  - High-performance computing (HPC) workloads like weather modeling.
  - Rendering 3D graphics in video games or animations.

Key Components in Many Core Architecture

- **Cores & L2 Cache:**
  - Each **core** (processing unit) has access to **L2 cache**, which stores frequently used data to reduce memory access latency.
  - The **cores are organized in groups**, likely representing Streaming Multiprocessors (SMs) in a GPU.
- **TD (Thread Dispatching) Units:**
  - These units help distribute workloads efficiently among the processing cores.
  - They manage threads and optimize parallel execution.
- **GDDR Memory Controllers (MC):**
  - These handle data transfers between the **GPU and the global memory (GDDR memory).**
  - GDDR (Graphics Double Data Rate) memory is optimized for high bandwidth.
- **PCIe Client Logic:**
  - Responsible for communication between the **GPU and the CPU/system memory** via the **PCIe (Peripheral Component Interconnect Express) bus**.



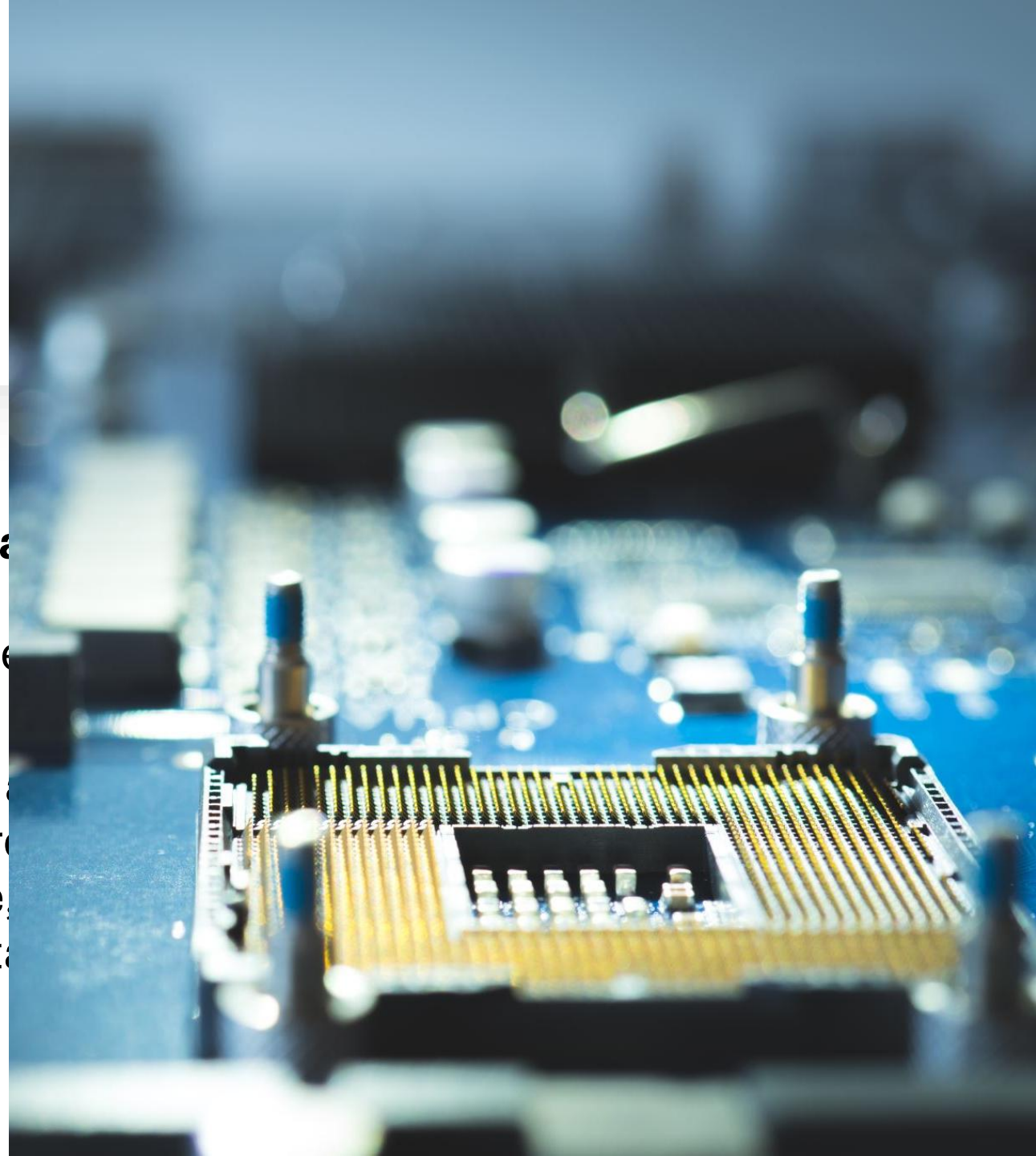
- **Interconnects (Arrows & Rings):**
  - The black arrows and circular connections represent **data flow and interconnects** that enable efficient communication between processing cores, caches, and memory controllers.

Key Differences

Feature	Multi-core Architecture	Many-core Architecture
Number of Cores	2 - 16 cores	Dozens to thousands of cores
Optimization	General-purpose computing	Highly parallel tasks
Cache Structure	L1 per core, shared L2/L3	Smaller caches, often per group of cores
Thread Handling	Efficient for multitasking	Optimized for high-thread parallelism
Use Cases	General computing, gaming, office apps	AI, scientific computing, simulations

Analogy for Better Understanding

- **Multi-core CPUs** are like **a sma**
  of highly skilled workers, each
  of doing a variety of tasks indepe
  but with limited numbers.
- **Many-core GPUs** are like **an**
  workers, each handling small, r
  tasks but at an enormous scale
  them perfect for highly parallel t

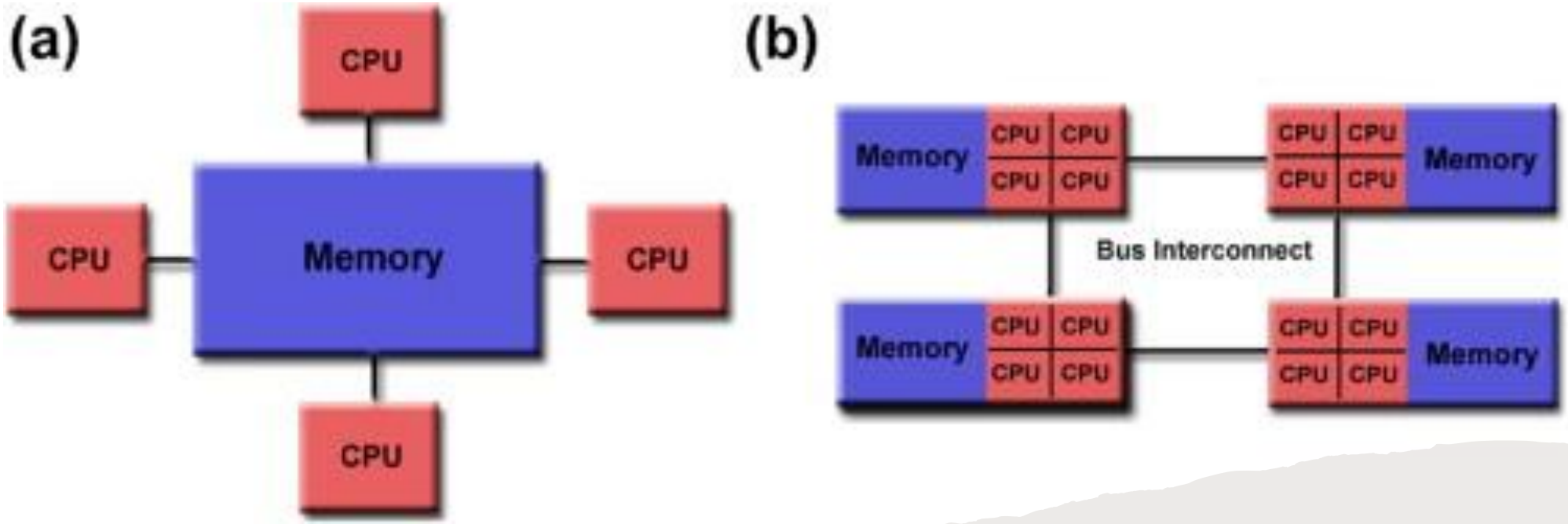


Shared Memory Model

- Multiple processors access the same memory space.
- Key Characteristics:
  - ✓ Fast data sharing between processors.
  - ✓ Requires synchronization to avoid conflicts (e.g., using mutexes and semaphores).

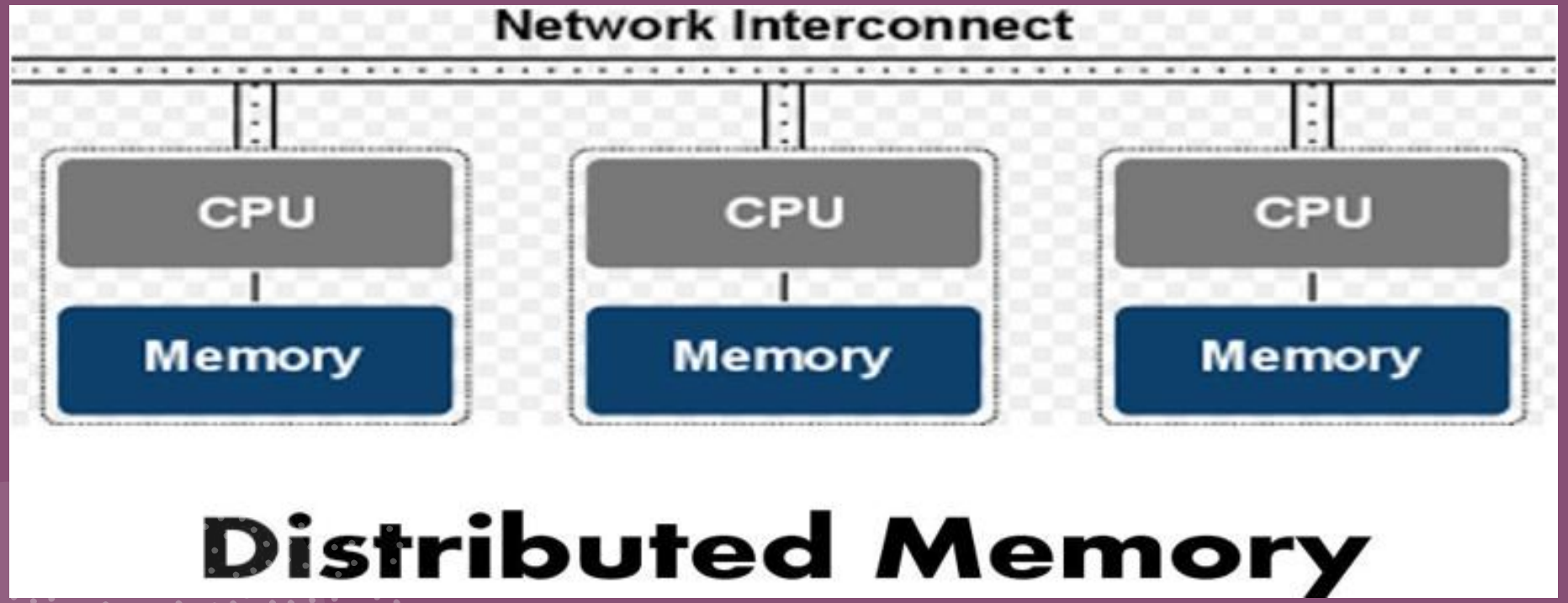
Example:

✓ Multi-threaded programming in a multi-core processor.



Distributed Memory Model

- Each processor has its own local memory and communicates via a network.
- Key Characteristics:
  - ✓ No direct memory access between processors.
  - ✓ Requires message passing (e.g., MPI - Message Passing Interface).
- Example:
  - ✓ Supercomputers using distributed clusters to solve large-scale problems



Shared vs. Distributed Memory

FEATURE	SHARED MEMORY	DISTRIBUTED MEMORY
Memory Access	Single memory space shared by processors	Each processor has its own memory
Communication	Direct memory access	Message passing (e.g., MPI)
Scalability	Limited to memory bandwidth	Highly scalable for large computations
Example	Multi-core CPUs	Cluster computing, cloud systems

Parallel & Distributed Systems

Lecture # 3

By,

Dr. Ali Akbar Sidduque

Introduction to Threads

01

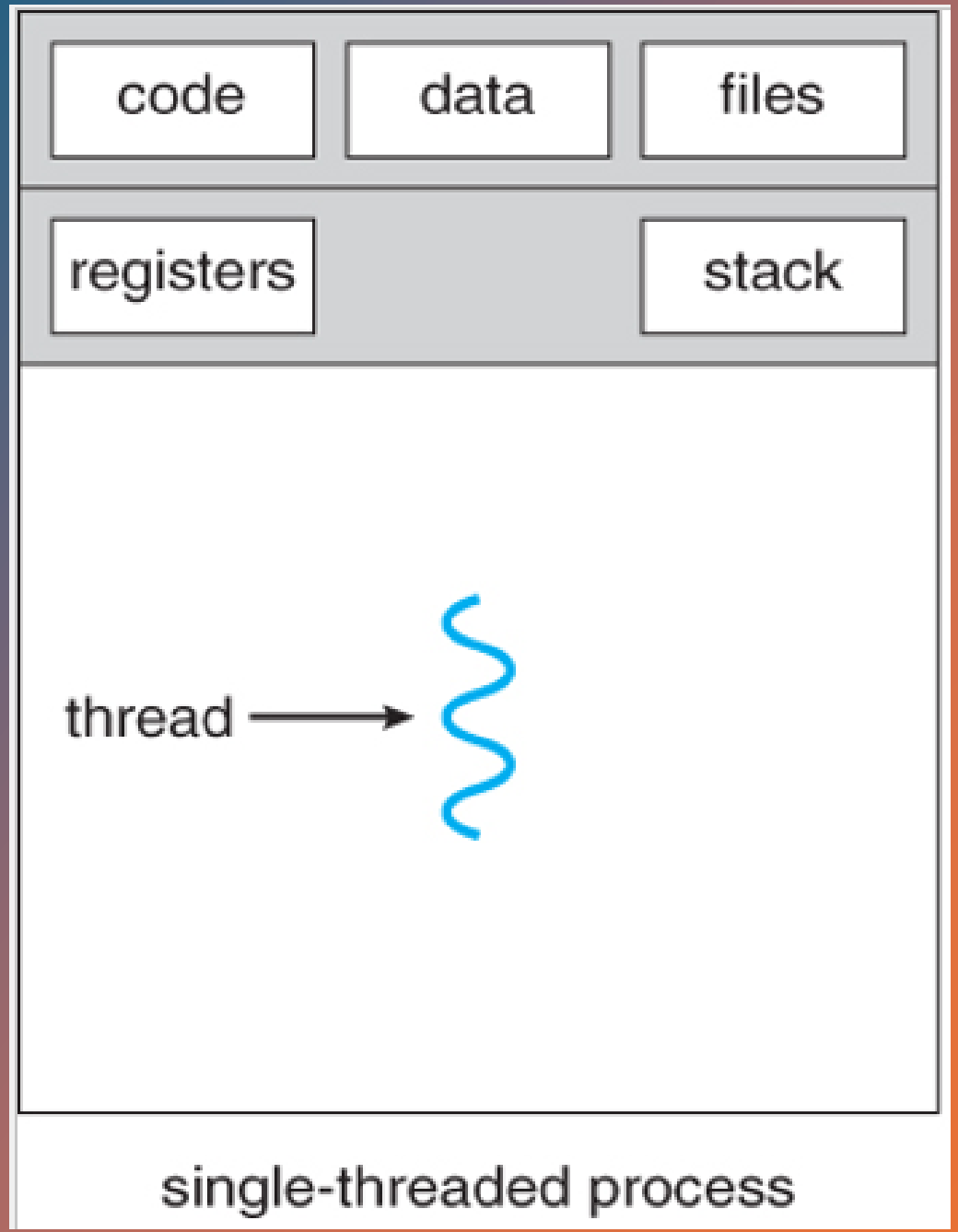
A thread is a lightweight process that runs independently within a program.

02

Threads share the same memory space but can execute different parts of a program simultaneously.

03

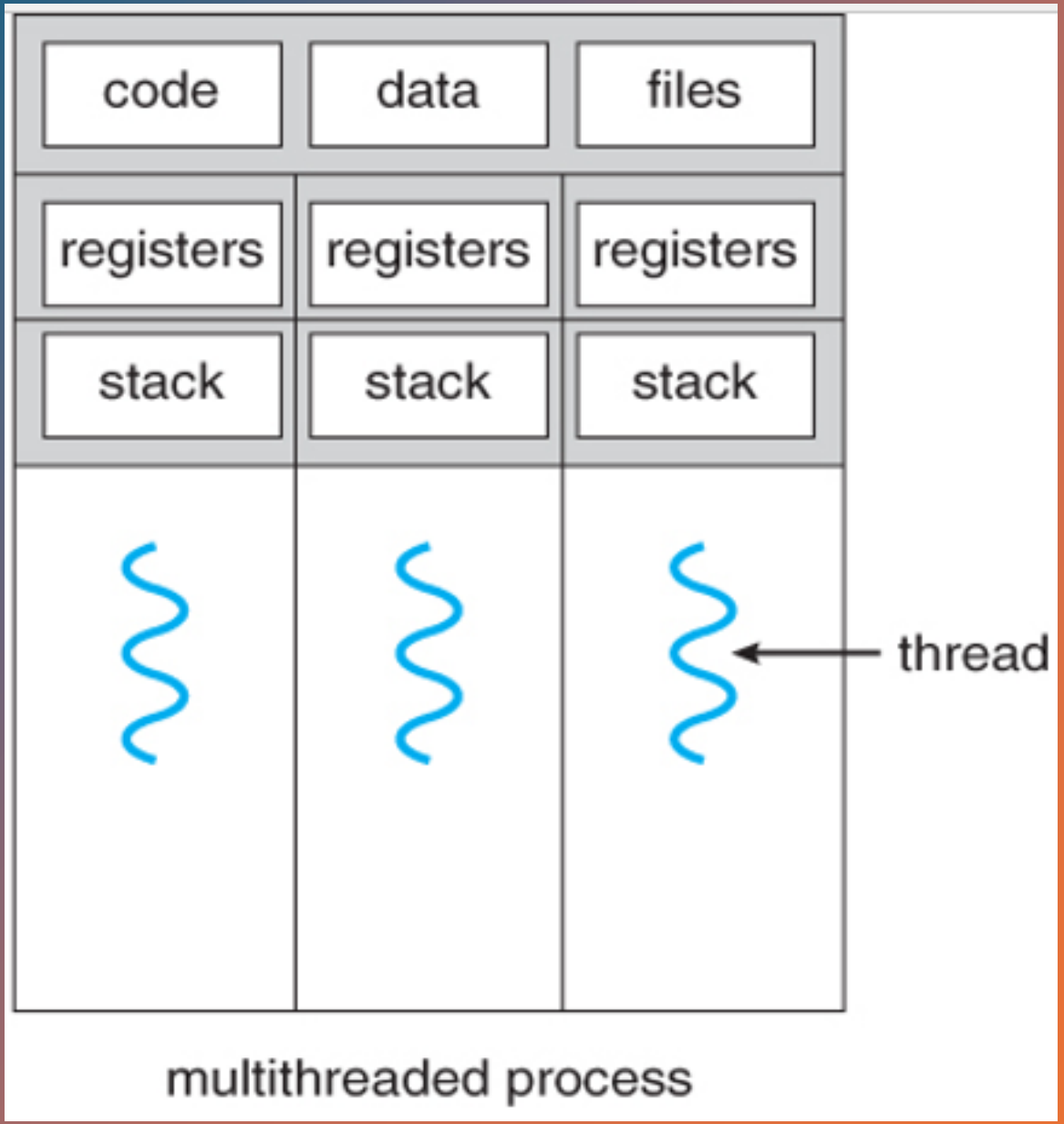
Used in modern computing to improve performance and responsiveness.



Single-threaded

Single-Threaded Process (Left Side):

- A single-threaded process contains only **one thread of execution**.
- It has a **code section, data section, and file section** that represent the program's instructions and resources.
- The **stack and registers** are dedicated to the single thread, meaning only one sequence of instructions is executed at a time.
- Since there is only **one execution thread**, tasks must be completed sequentially.



Multi-threaded processes

- A multi-threaded process consists of multiple threads within the same process.
- The code, data, and file sections are shared among all threads, allowing for efficient resource utilization.
- Each thread has its own stack and register set, enabling concurrent execution of different tasks within the same application.
- Advantage: Multi-threading improves performance by executing multiple operations in parallel, reducing CPU idle time and enhancing responsiveness.

Key Takeaways

- Single-threaded processes execute tasks one after another, leading to potential inefficiencies.
- Multi-threaded processes allow multiple tasks to run concurrently within the same program, improving performance and responsiveness.
- Multi-threading is commonly used in applications like web servers, parallel computing, and real-time systems to handle multiple operations simultaneously.

Types of Threads

Feature	User Threads	Kernel Threads
Definition	Threads managed by user-level libraries without kernel intervention.	Threads managed directly by the OS kernel.
Management	Managed by the user-space thread library (e.g., POSIX Pthreads, Java threads).	Managed by the operating system.
Speed	Faster as context switching does not involve the kernel.	Slower due to kernel involvement in scheduling.
Dependency	If one thread blocks, all threads in the process block.	If one thread blocks, other threads in the process continue execution.
Portability	More portable across different OS platforms.	Less portable as they are OS-dependent.
System Calls	Cannot take advantage of multi-core processors since the kernel sees only a single-threaded process.	Can run on multiple processors simultaneously, improving performance.
Example	Java Virtual Machine (JVM) threads, Green Threads.	Linux pthreads, Windows threads.

Thread Lifecycle

New: Thread is created but not yet started.

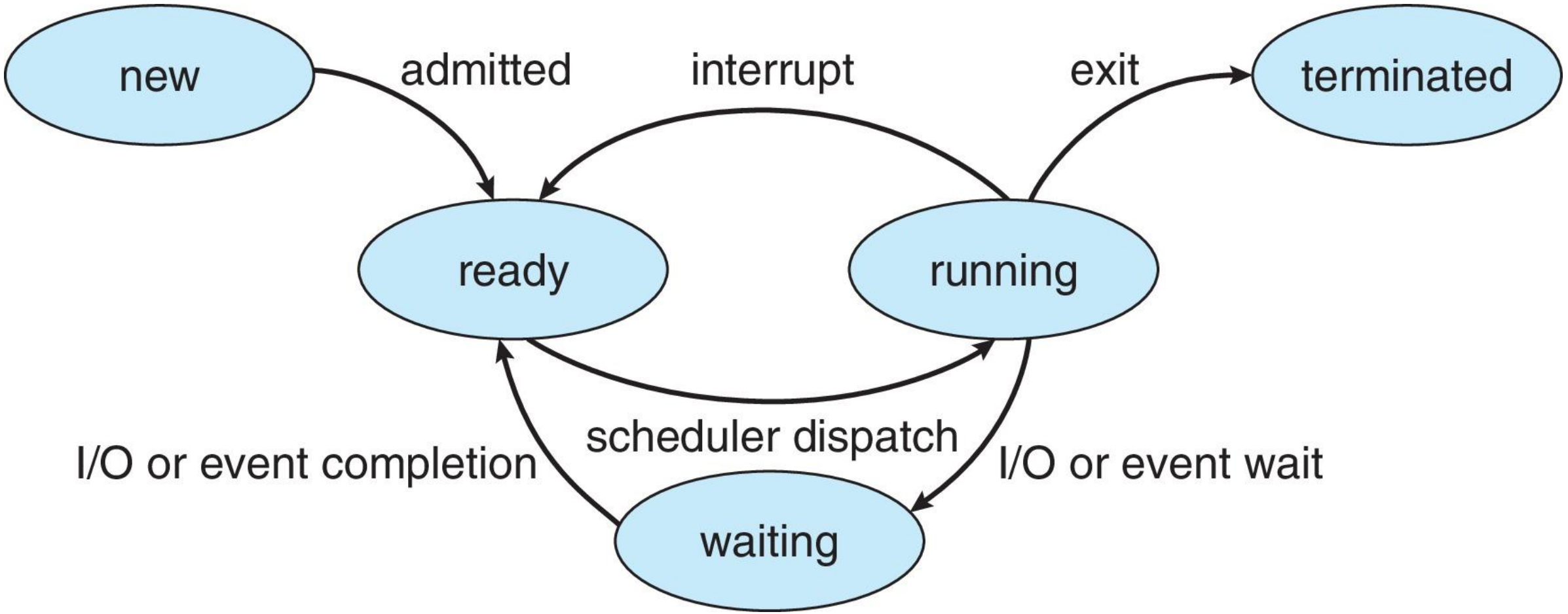
Ready: Thread is waiting for CPU allocation.

Running: Thread is executing instructions.

Blocked: Waiting for a resource or input.

Terminated: Execution completed or manually stopped.

Thread Lifecycle



Thread Life Cycle

1. **New State**:
   - The process is created but not yet ready for execution.
   - It waits for admission into the ready queue by the operating system.
2. **Ready State:**
   - The process is loaded into memory and waiting for CPU time.
   - It is ready to be assigned to a processor by the CPU scheduler.
3. **Running State**:
   - The process is currently executing on the CPU.
   - If it completes execution, it moves to the **terminated** state.
   - If an **interrupt** occurs, it moves back to the **ready** state.
4. **Waiting State**:
   - The process is waiting for an I/O operation or an event to occur.
   - It cannot execute until the required operation is completed.
5. **Terminated State**:
   - The process has completed execution or has been forcefully stopped.
   - It is removed from memory.
6. **Transitions Between States**:
   - **Admitted**: A process moves from **new** to **ready** state when it is scheduled.
   - **Scheduler Dispatch**: Moves a process from **ready** to **running** when the CPU is allocated.
   - **Interrupt**: A process in **running** can be preempted and moved back to **ready**.
   - **I/O or Event Wait**: A process moves from **running** to **waiting** if it requires an I/O operation.
   - **I/O or Event Completion**: The process returns from **waiting** to **ready** when the I/O task is completed.
   - **Exit**: A process moves from **running** to **terminated** once execution is complete.

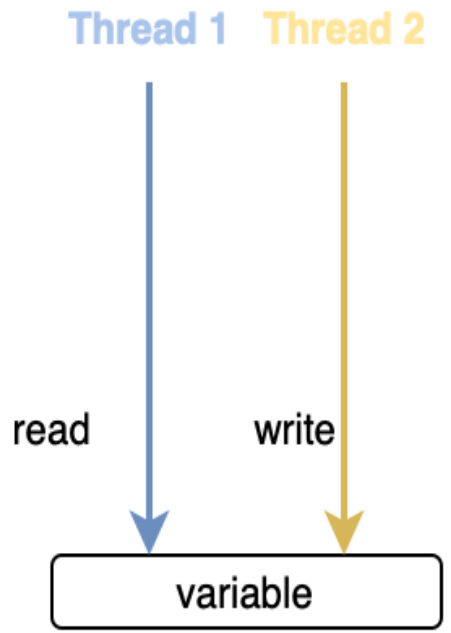
Why Use Threads?

- Improved **performance** via parallel execution.
- **Efficient resource utilization** (same memory space shared).
- **Better responsiveness** in applications like UI-based systems.
- **Simplifies complex tasks** by breaking them into smaller units.

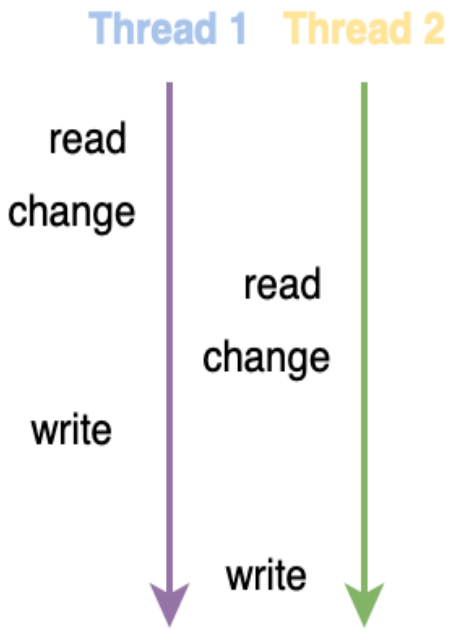
What is Synchronization?

- Synchronization ensures orderly access to shared resources.
- Prevents **race conditions**, **deadlocks**, and **inconsistent data states**.
- Common in **multithreaded applications** and **parallel computing**.

Data races



Race condition



Data Races vs. Race Condition

Data Races:

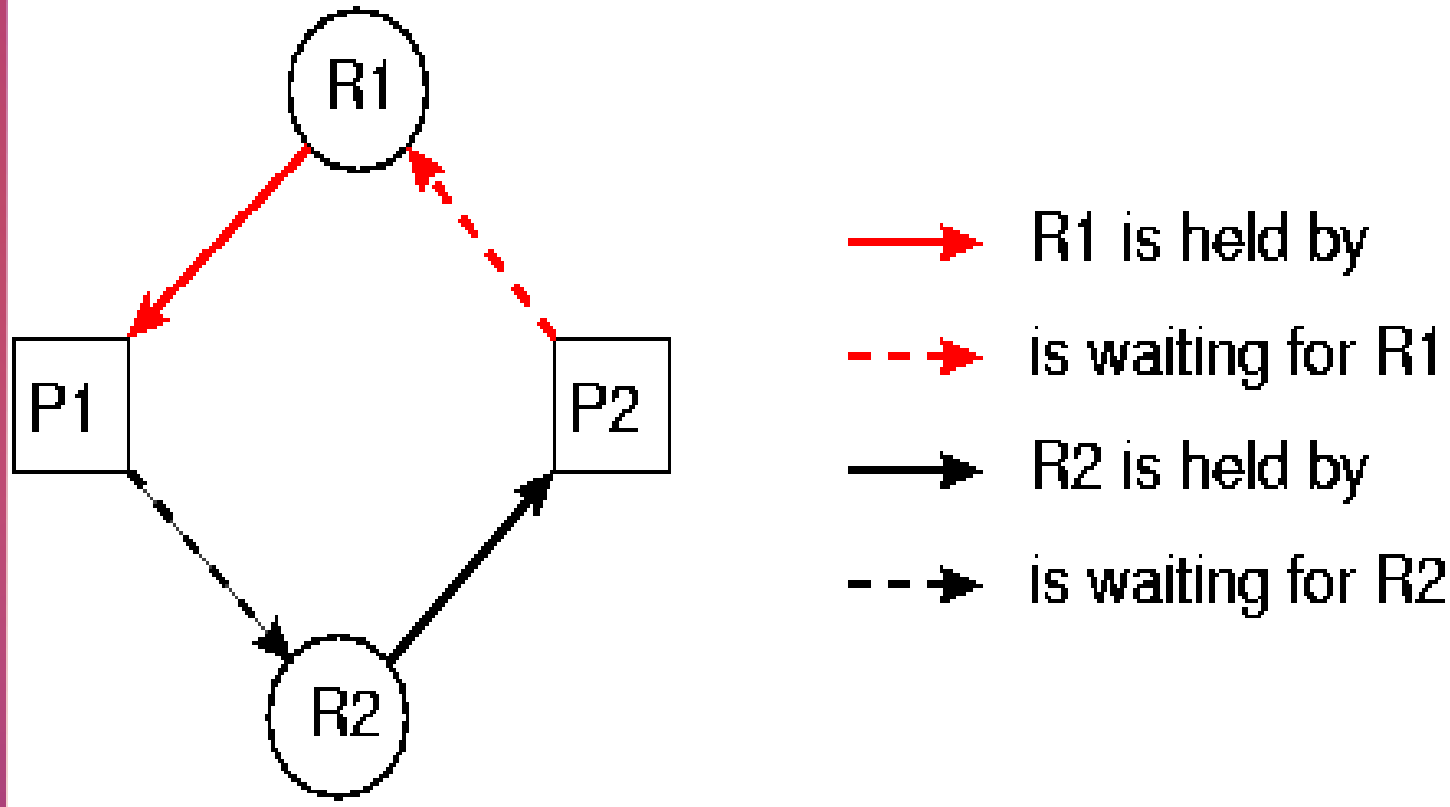
- Occurs when two threads access a shared variable concurrently.
- One thread reads while another writes at the same time.
- Leads to unpredictable or inconsistent values in the variable.

Race Condition:

- Happens when the outcome depends on the sequence/timing of threads.
- Both threads read, modify, and write the variable without proper synchronization.
- Can cause unexpected behaviors or incorrect results in a program.

Race Conditions & Deadlocks

- **Race Condition:** Two or more threads access a shared variable simultaneously, leading to unpredictable results.
- **Deadlock:** Two or more threads wait indefinitely for resources locked by the other.

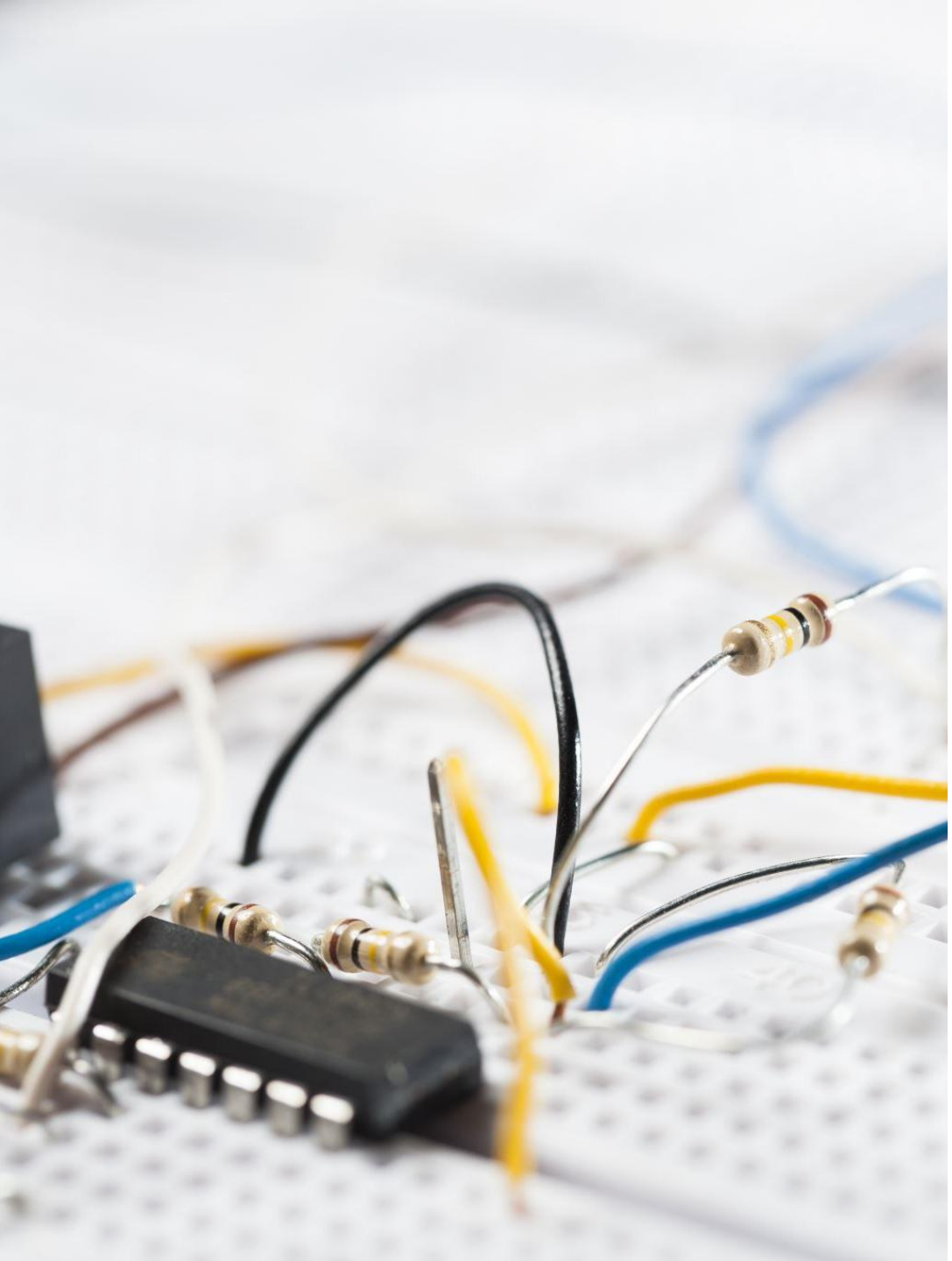


Synchronization Mechanisms

1. **Mutexes (Mutual Exclusion):** Locking mechanism to ensure only one thread accesses a resource.
2. **Semaphores:** Signaling mechanism for limited resource access.
3. **Monitors:** High-level synchronization that encapsulates shared resources.
4. **Atomic Variables:** Prevents partial updates to shared data.

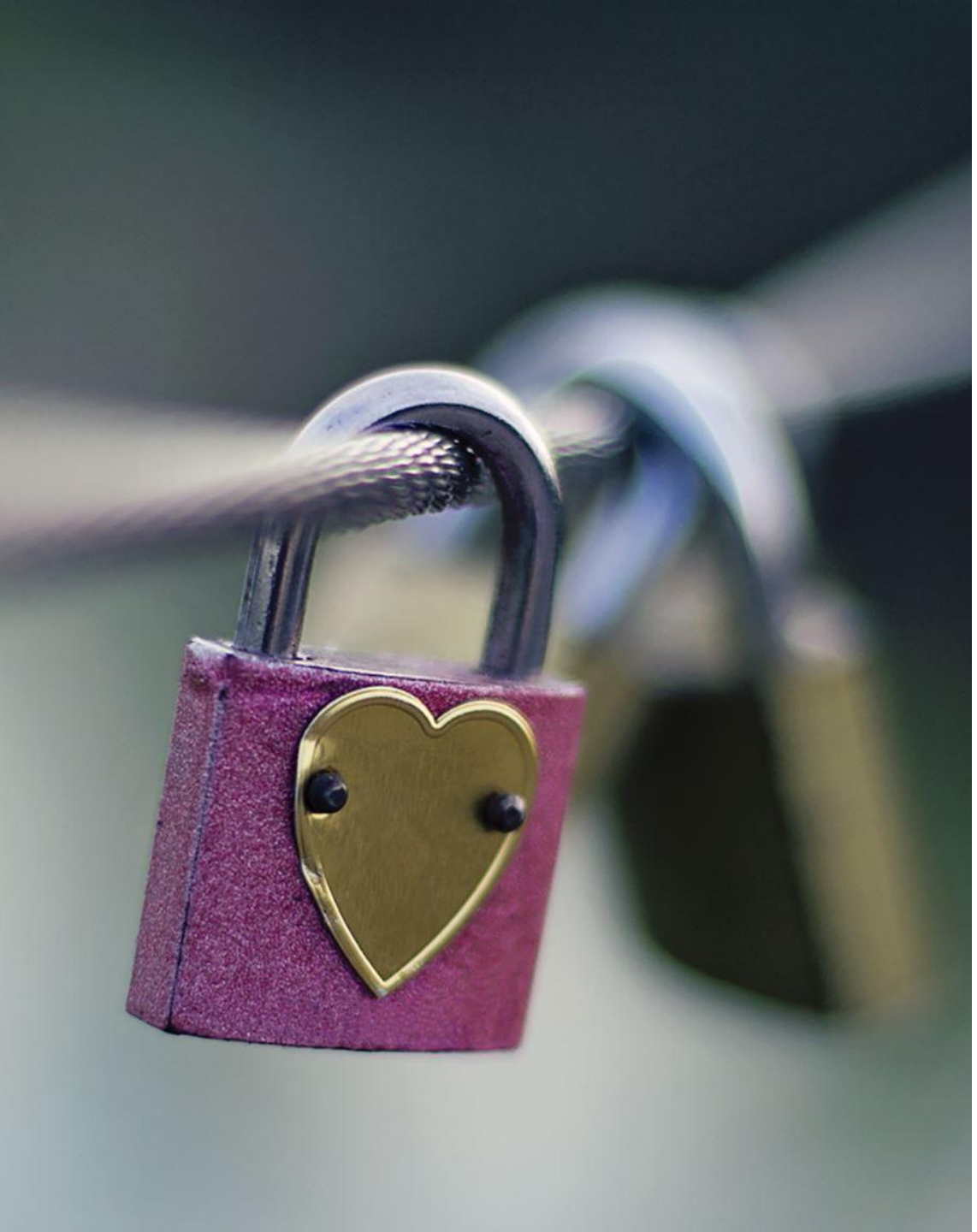
Implementing Synchronization in Python

```
import threading
lock = threading.Lock()
def critical_section():
    with lock:
        # Critical section code
        print("Thread is running safely")
thread =
threading.Thread(target=critical_section)
thread.start()
thread.join()
```



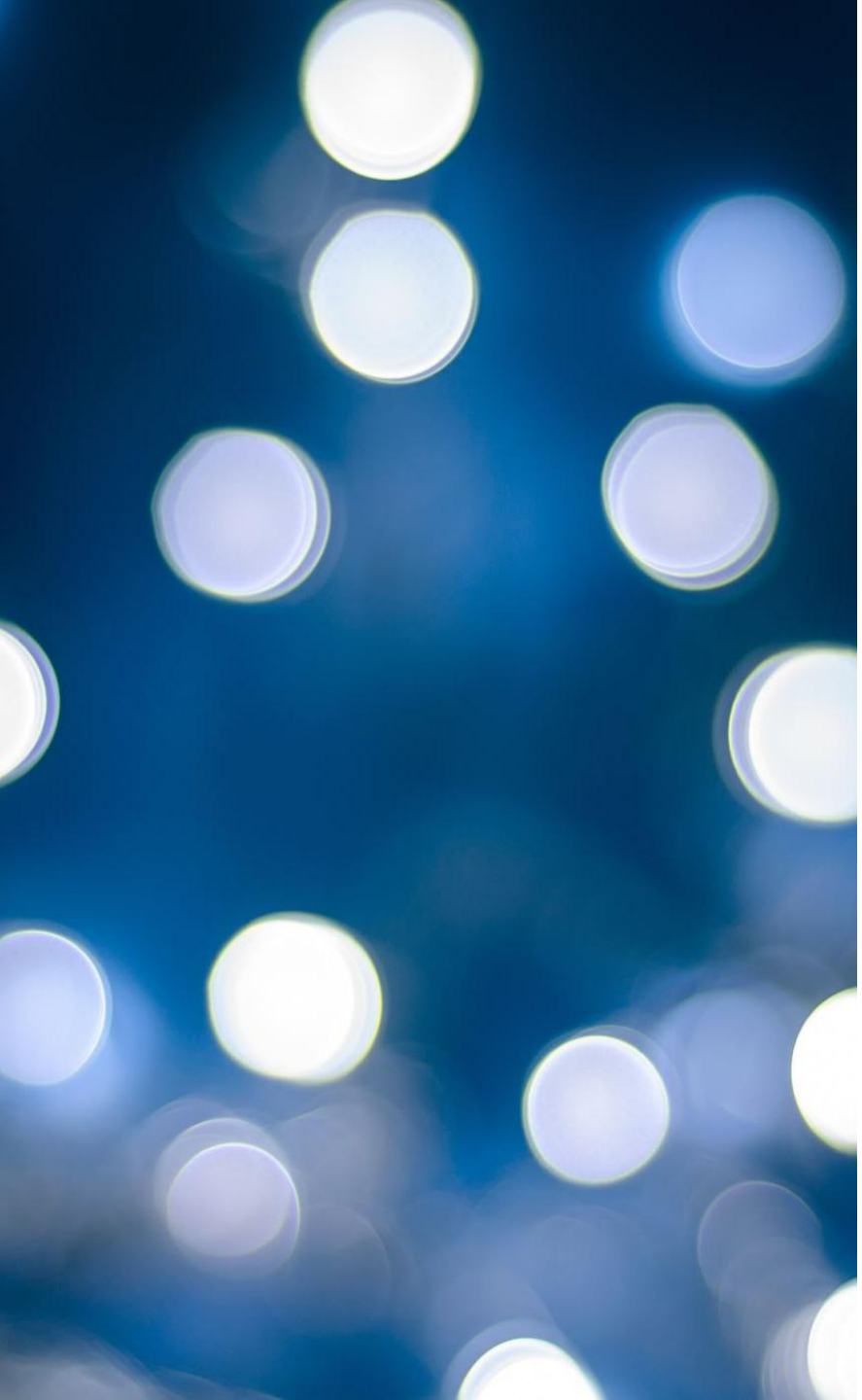
1. Importing the Threading Module

- The `threading` module provides tools for managing threads in Python.
- Used to implement synchronization mechanisms like Locks and Semaphores.



2. Creating a Lock

- `lock = threading.Lock()` creates a Lock object.
- Prevents multiple threads from accessing shared resources simultaneously.



3. Defining the Critical Section

- `with lock:` ensures only one thread enters the critical section at a time.

- Automatically releases the lock after execution to prevent deadlocks.

4. Creating and Starting a Thread

- `thread = threading.Thread(target=critical_section)` creates a new thread.

- `thread.start()` starts execution in a separate thread.

5. Waiting for the Thread to Finish

- `thread.join()` blocks the main program until the thread completes execution.

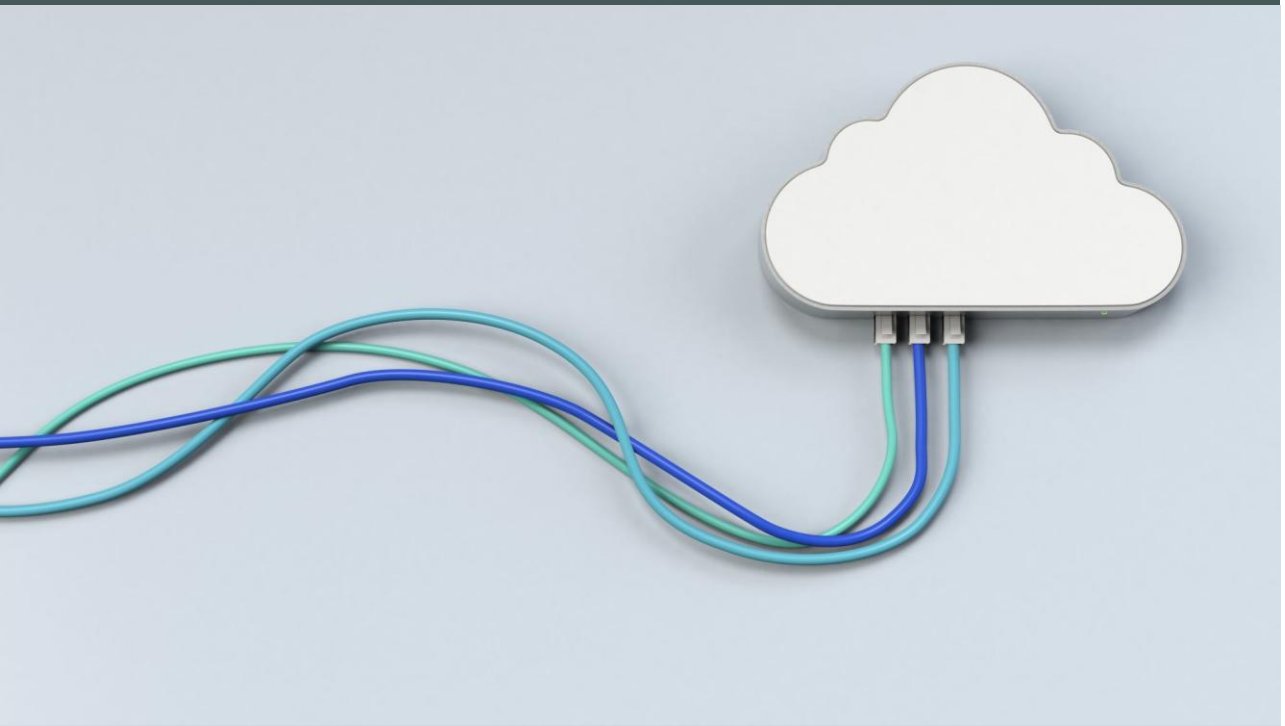
- Ensures the program does not exit prematurely.

Key Takeaways

- - Locks prevent race conditions by allowing only one thread to execute at a time.
- - `with lock:` ensures safe access to shared resources.
- - `thread.start()` runs the thread, and `thread.join()` waits for it to finish.

Lecture # 5

What is a Distributed System?



A **Distributed System** is a collection of **independent computers** that appears to its users as a **single coherent system**.

Examples:

- Google Search
- Distributed Databases (e.g., Cassandra, MongoDB)
- Online Banking Systems
- Cloud Computing Platforms

Why Use Distributed Systems?

- Scalability
- Fault tolerance
- Resource sharing
- Performance improvements

Communication in Distributed Systems

Two Models of Communication:

- Synchronous Communication
- Asynchronous Communication

Communication Types:

- Point-to-point (1-to-1)
- Broadcast or multicast (1-to-many)

Synchronous vs Asynchronous Communication

Feature	Synchronous	Asynchronous
Definition	Sender waits for reply	Sender continues without waiting
Latency	Lower control, predictable	Higher variability, less predictable
Error Handling	Easier	More complex
Use Cases	RPCs, banking systems	Email systems, message queues
Blocking?	Yes	No

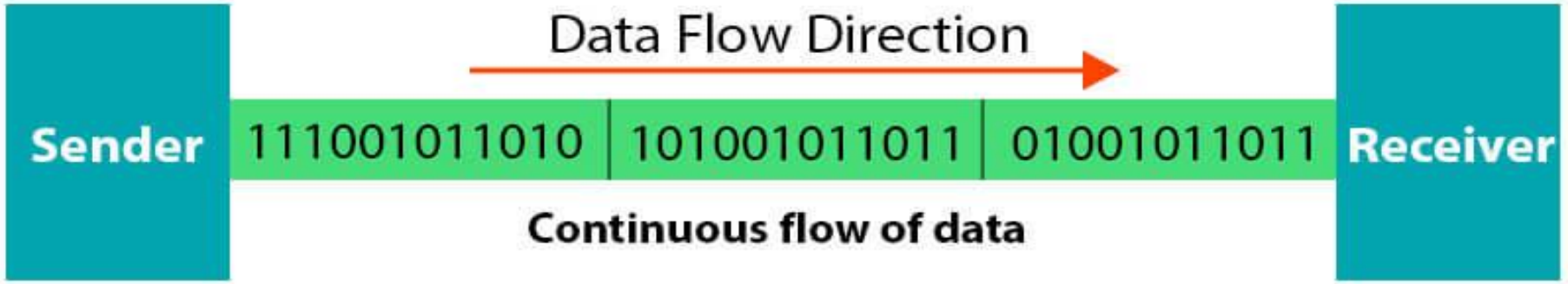
Synchronous vs. Asynchronous Communication

Communication is **at the heart** of distributed systems. Nodes (or processes) must exchange messages to coordinate tasks, share resources, and ensure consistency. The nature of this communication—**synchronous** or **asynchronous**—has a profound impact on **system design, performance, and complexity**. (Synchronous & Asynchronous)

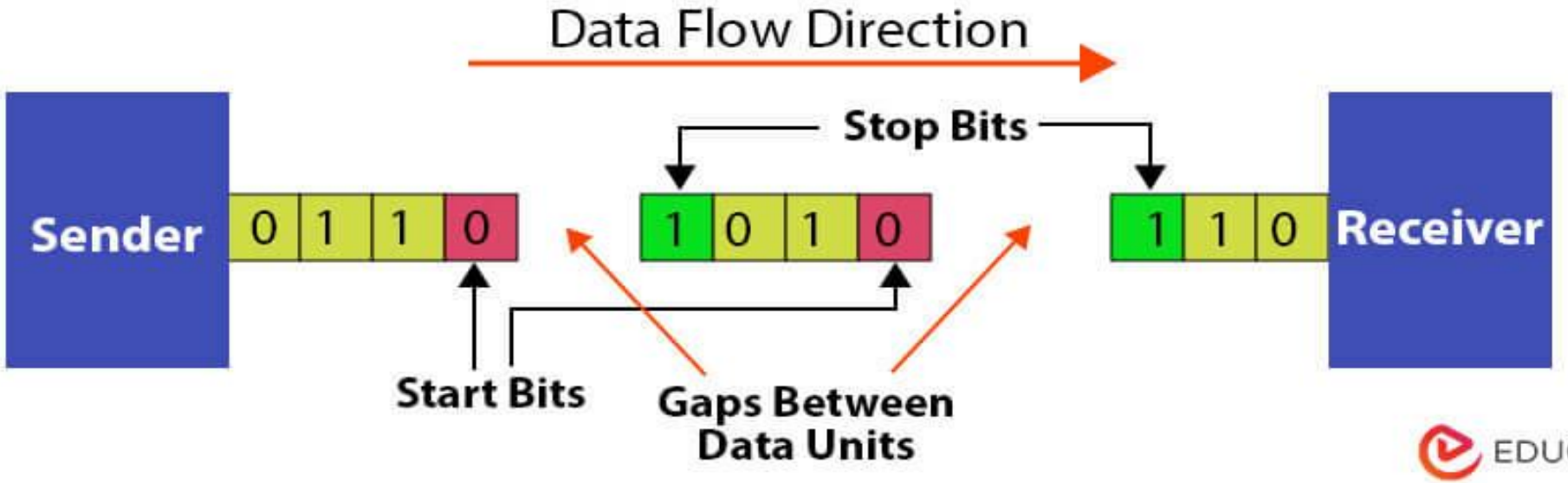
Scenario	Analogy
Synchronous	Phone call – both must be online and respond in real time.
Asynchronous	Email – send anytime; receiver reads/responds later.

Synchronous and Asynchronous Transmission

Synchronous Transmission



Asynchronous Transmission



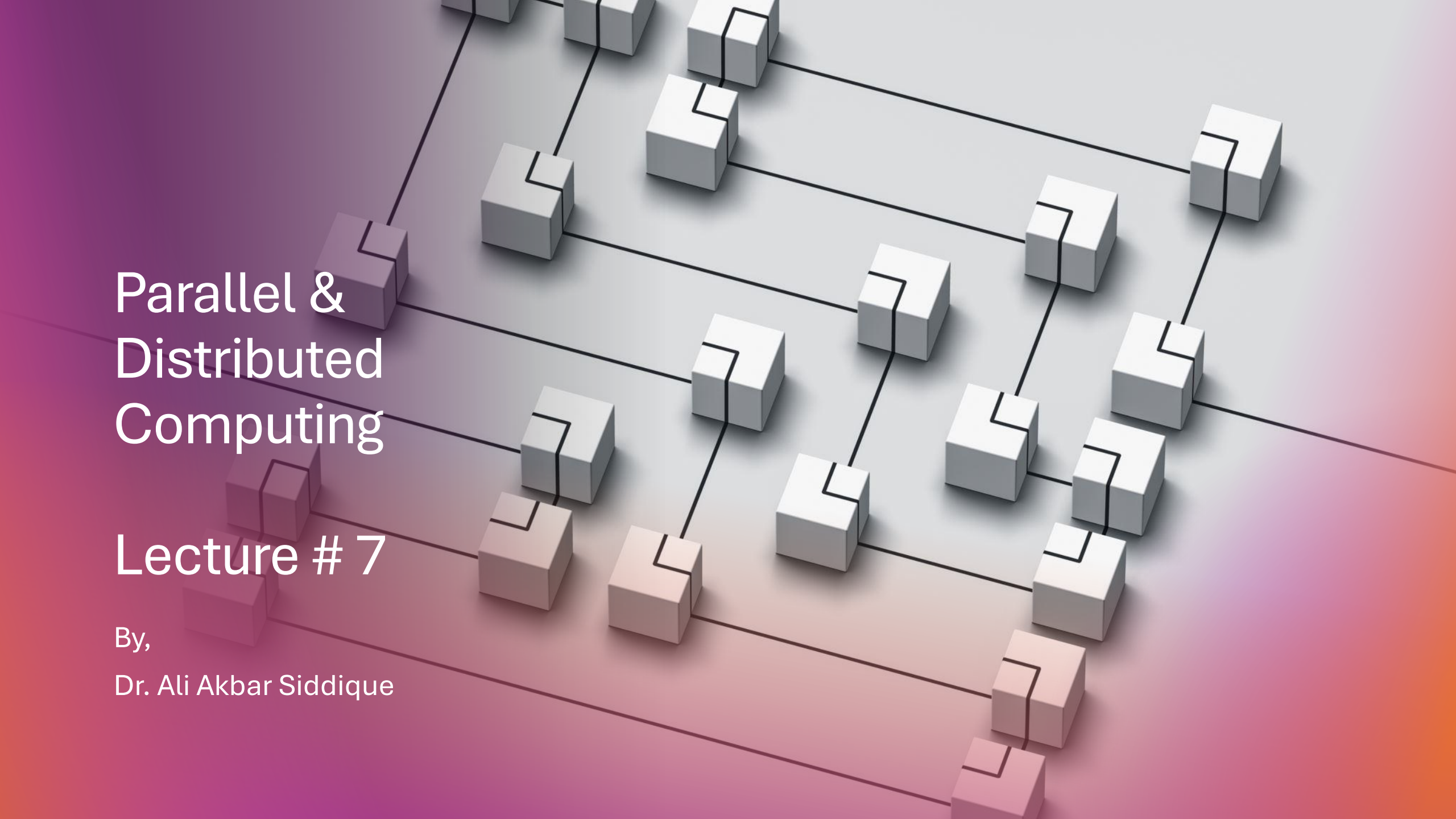
EDUCBA

Comparison Table

Feature	Synchronous	Asynchronous
Blocking	Yes (sender waits)	No (sender continues)
Message Handling	Immediate	Delayed/Queued
Coupling	Tight temporal coupling	Loose temporal coupling
Scalability	Lower	Higher
Reliability Dependency	High on receiver availability	Less dependent on receiver state
Complexity	Lower (easier to reason)	Higher (requires queues, handling)
Typical Use Cases	RPCs, DB queries	Message Queues, Microservices, IoT

Choosing Between Them

Application Requirement	Recommended Model
Real-time interaction	Synchronous
High throughput and decoupling	Asynchronous
Strict consistency	Synchronous
Fault tolerance and resilience	Asynchronous



Parallel & Distributed Computing

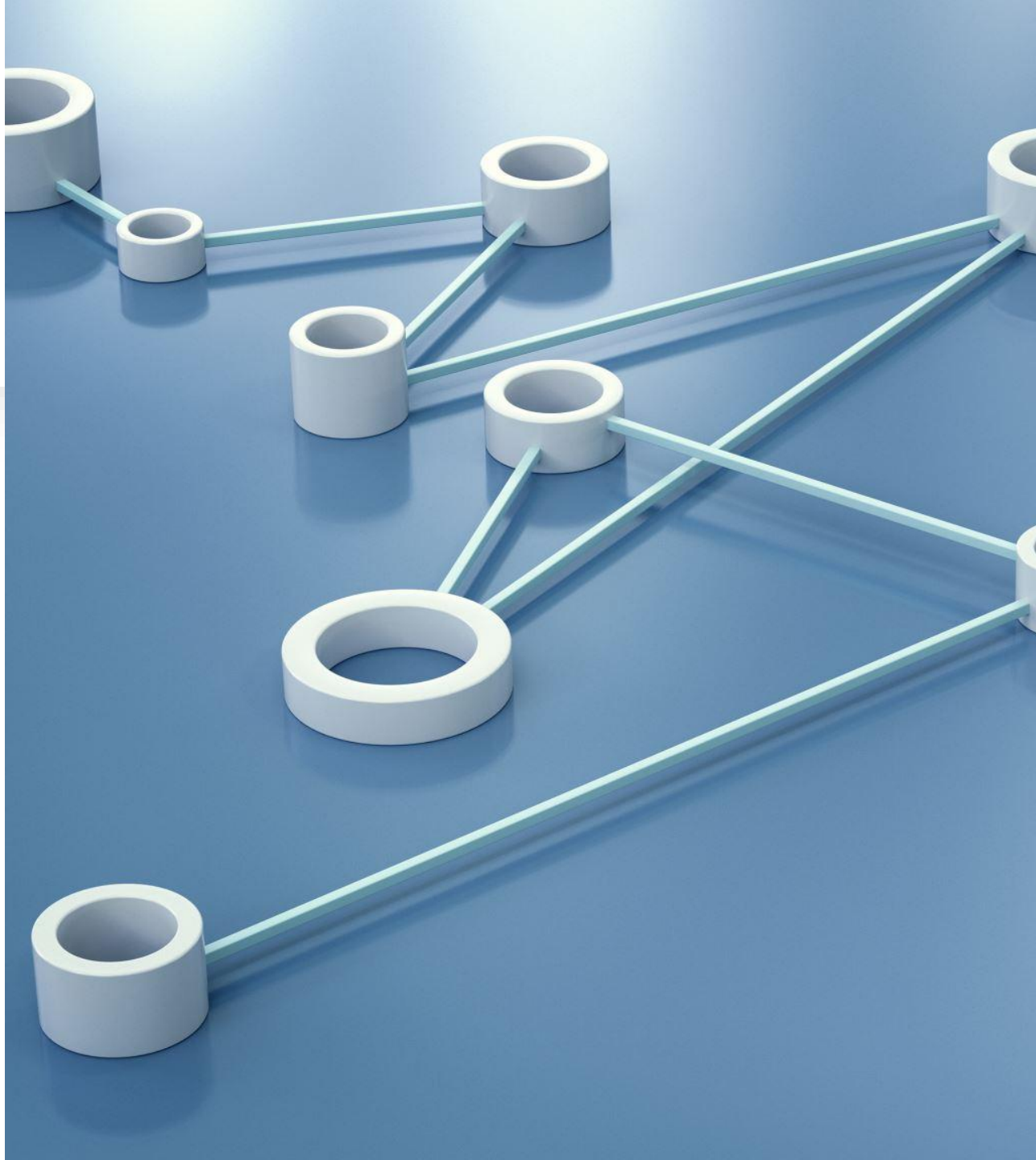
Lecture # 7

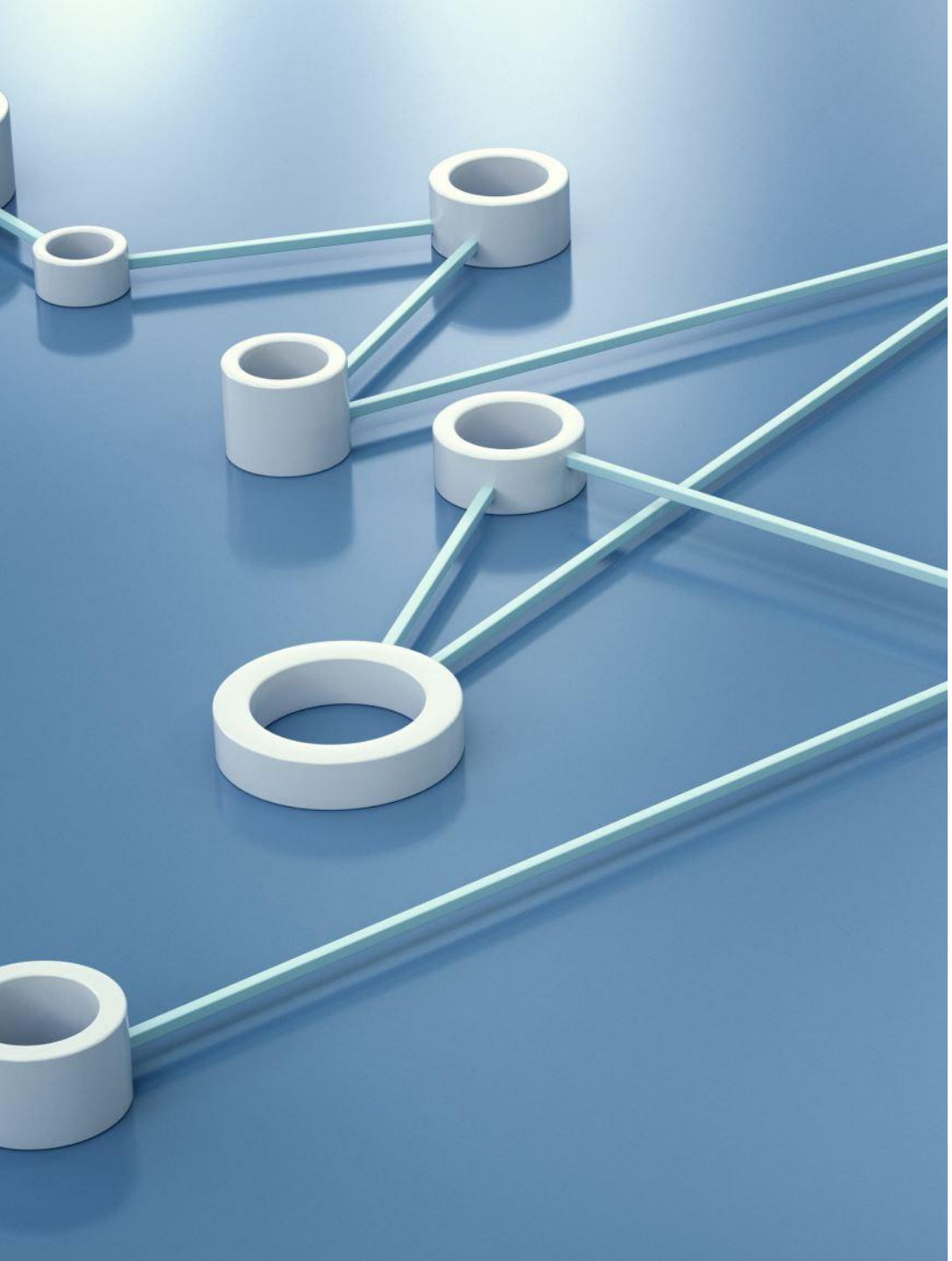
By,

Dr. Ali Akbar Siddique

What is Cloud Computing and Distributed Frameworks?

- Cloud Computing: Cloud computing is a model that provides on-demand access to computing resources (like servers, storage, databases, networking, software, and analytics) over the internet. It allows users to leverage powerful resources without owning the physical infrastructure.
- Distributed Frameworks: Distributed frameworks are software architectures that allow multiple computers (nodes) to work together on a single task. These frameworks are designed to enable parallel processing, efficient resource utilization, and fault tolerance.

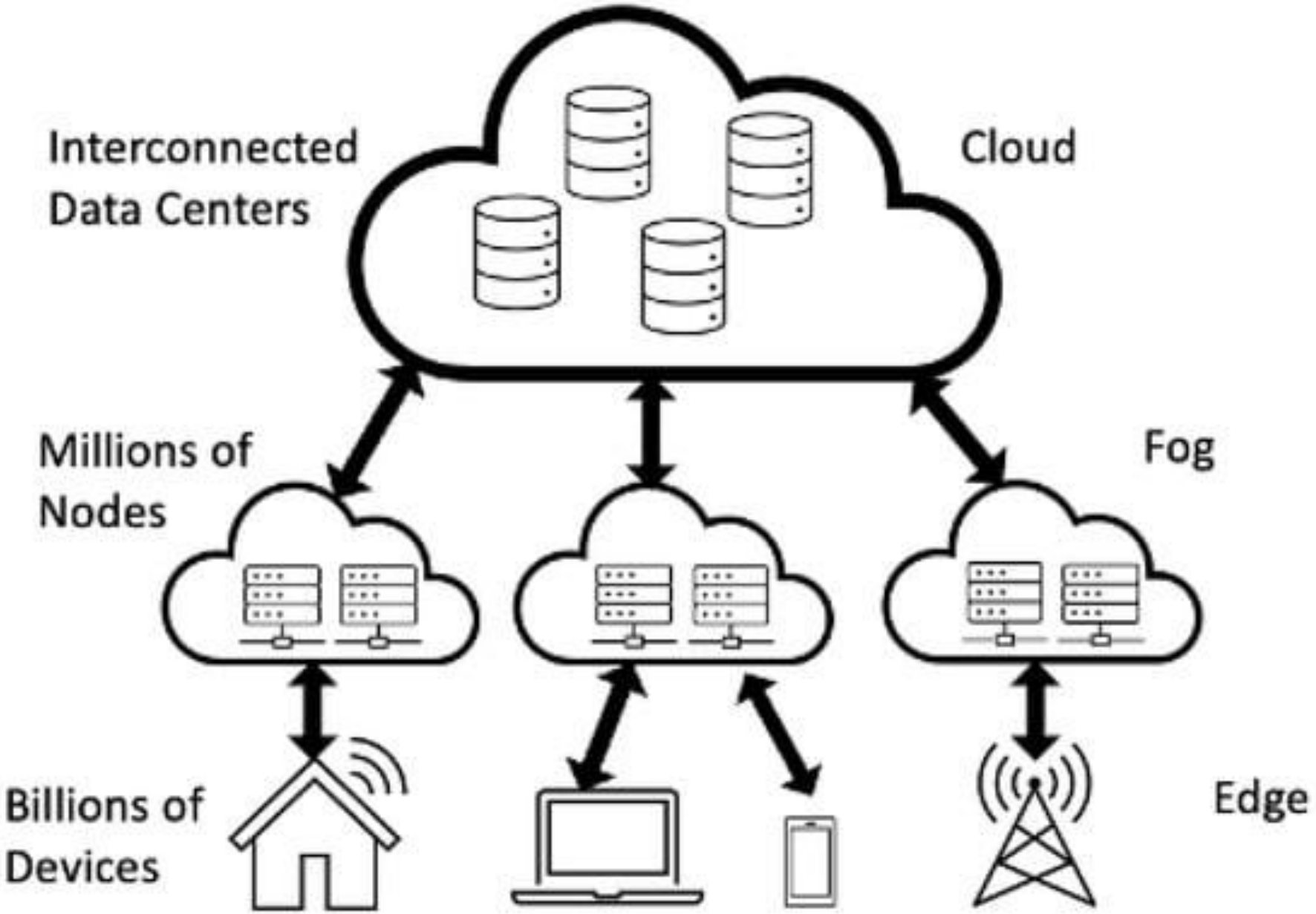




Why Study Cloud Computing and Distributed Frameworks?

- Cloud computing and distributed frameworks are the backbone of modern technology, powering everything from social media and streaming services to data analytics and artificial intelligence.
- Understanding these technologies is crucial for anyone interested in high-performance computing, big data processing, and scalable software development.

Cloud Computing



Key Characteristics of Cloud Computing

1. **On-Demand Self-Service:**
   - Users can provision computing resources (like virtual machines, storage, and applications) without requiring human intervention.
   - Example: Users can deploy a virtual server on Amazon Web Services (AWS) with a few clicks.

2. **Broad Network Access:**
   - Cloud services are accessible over the internet through various devices, including laptops, smartphones, and tablets.
   - Example: Accessing Google Drive files from any device.

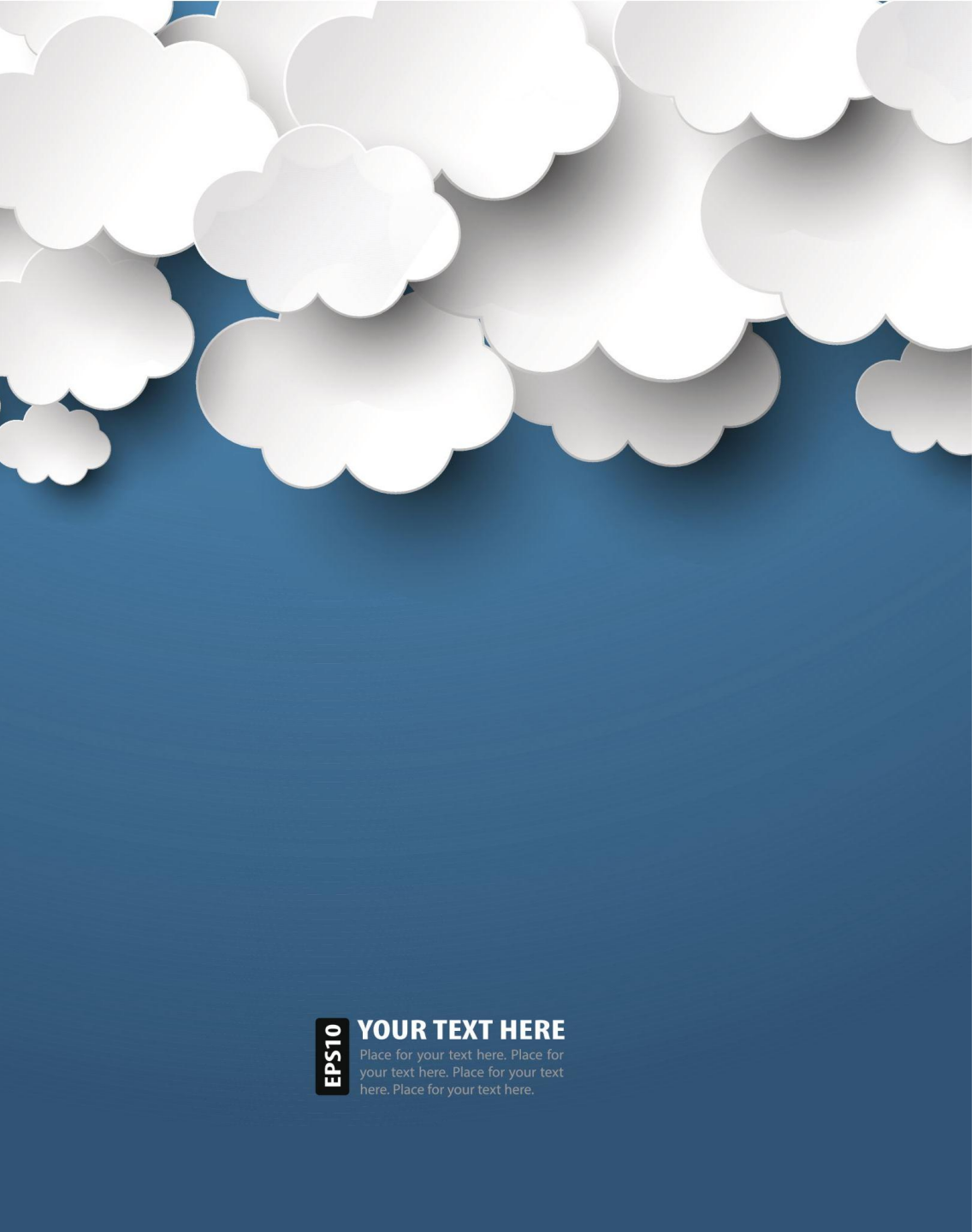
3. **Resource Pooling:**
   - Cloud providers use multi-tenant models to serve multiple customers using the same physical resources.
   - Example: Microsoft Azure uses a shared pool of servers to provide computing power to multiple users.

4. **Rapid Elasticity:**
   - Resources can be quickly scaled up or down depending on user demand.
   - Example: Netflix scales its server capacity during peak hours to support more users.

5. **Measured Service:**
   - Resource usage is monitored and billed on a pay-as-you-go or subscription basis.
   - Example: Users are billed for the amount of cloud storage or computing power they consume.

How Cloud Computing Works:

- **User Interaction**: Users access cloud services through a web browser or a dedicated application.
- **Service Delivery**: Cloud providers manage and maintain the underlying infrastructure.
- **Resource Allocation**: Virtual resources are allocated based on user requirements.



Types of Cloud Environments:

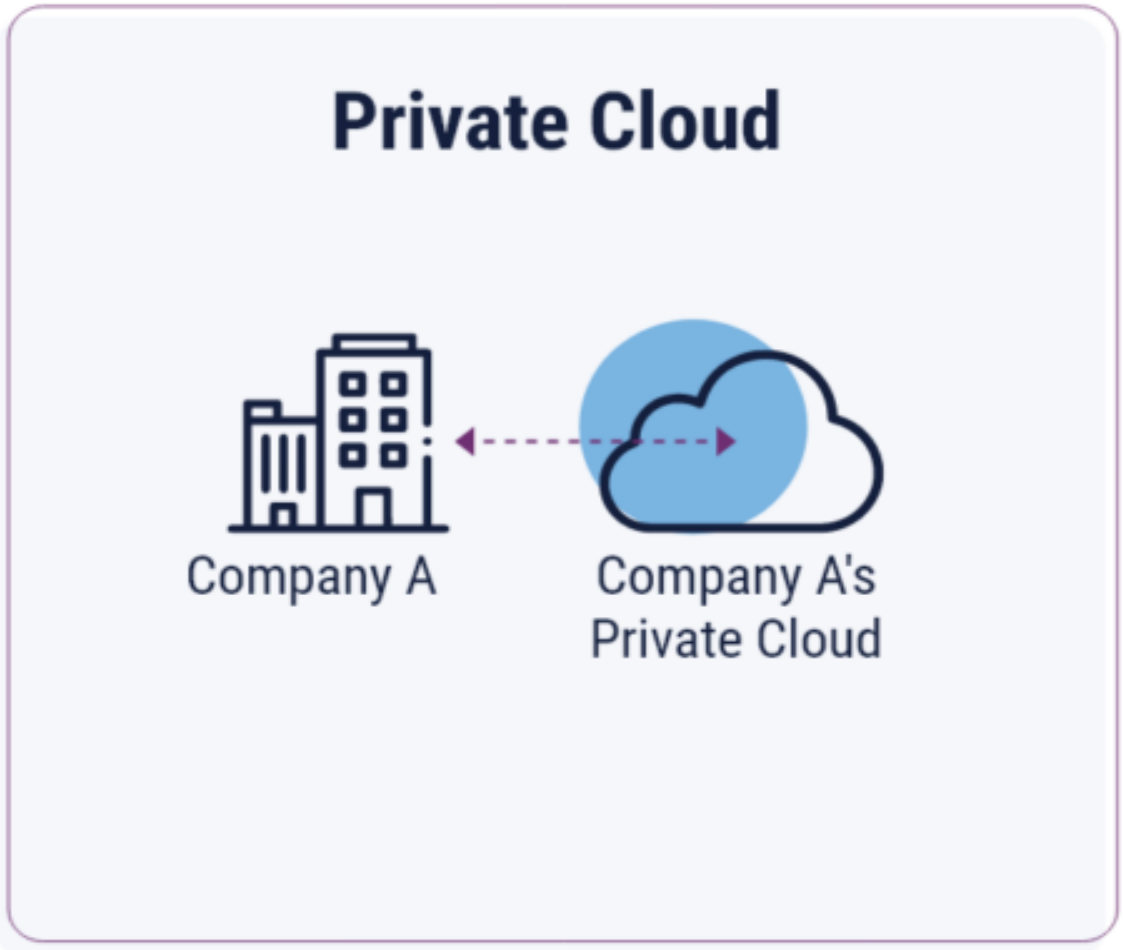
1. **Public Cloud:** Services are delivered over the internet and are accessible to anyone.
   - Example: Google Cloud Platform (GCP), Amazon Web Services (AWS).

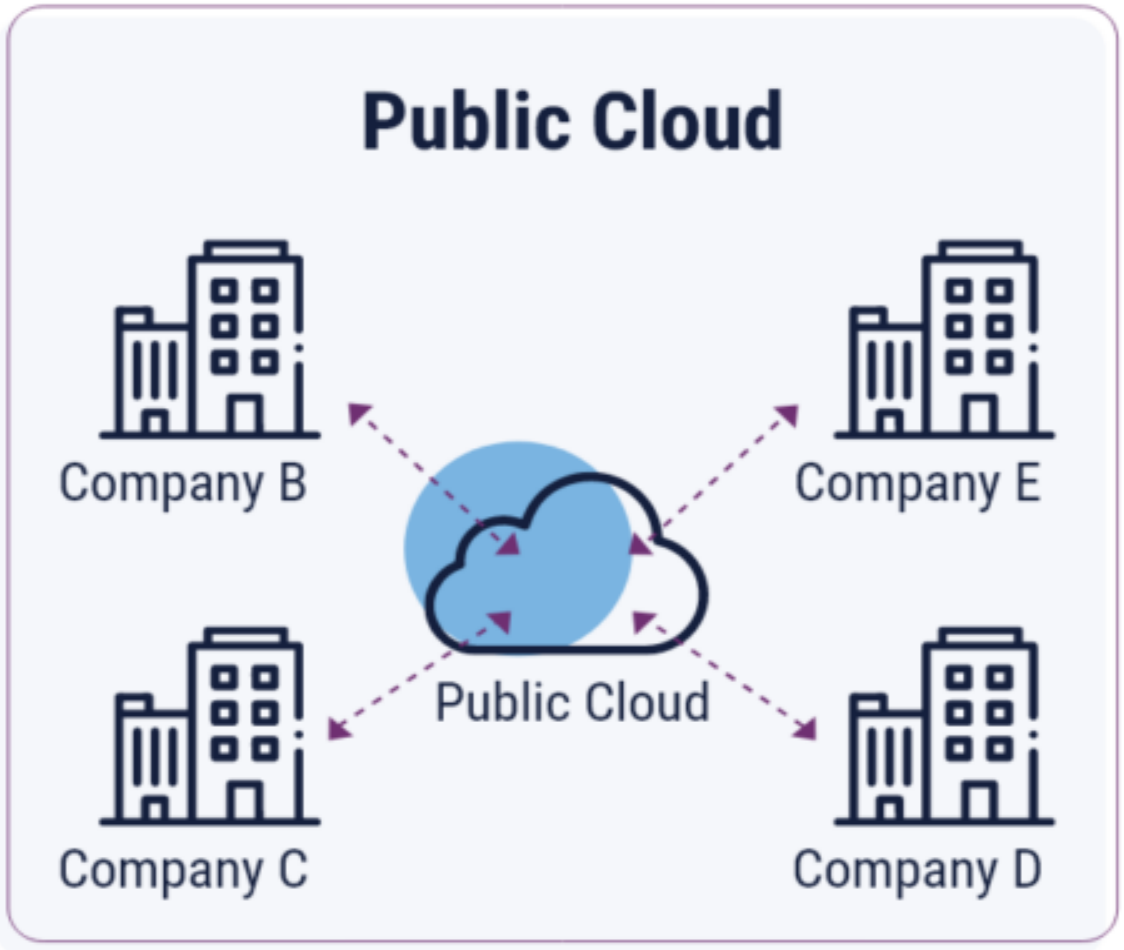
2. **Private Cloud:** Cloud infrastructure is dedicated to a single organization, providing more control and security.
   - Example: An organization’s internal cloud data center.

3. **Hybrid Cloud:** Combines public and private cloud environments, allowing data and applications to be shared between them.
   - Example: A company using both AWS for public cloud storage and a private cloud for sensitive data.

4. **Multi-Cloud:** The use of multiple cloud providers (like AWS, Azure, and Google Cloud) to avoid vendor lock-in.

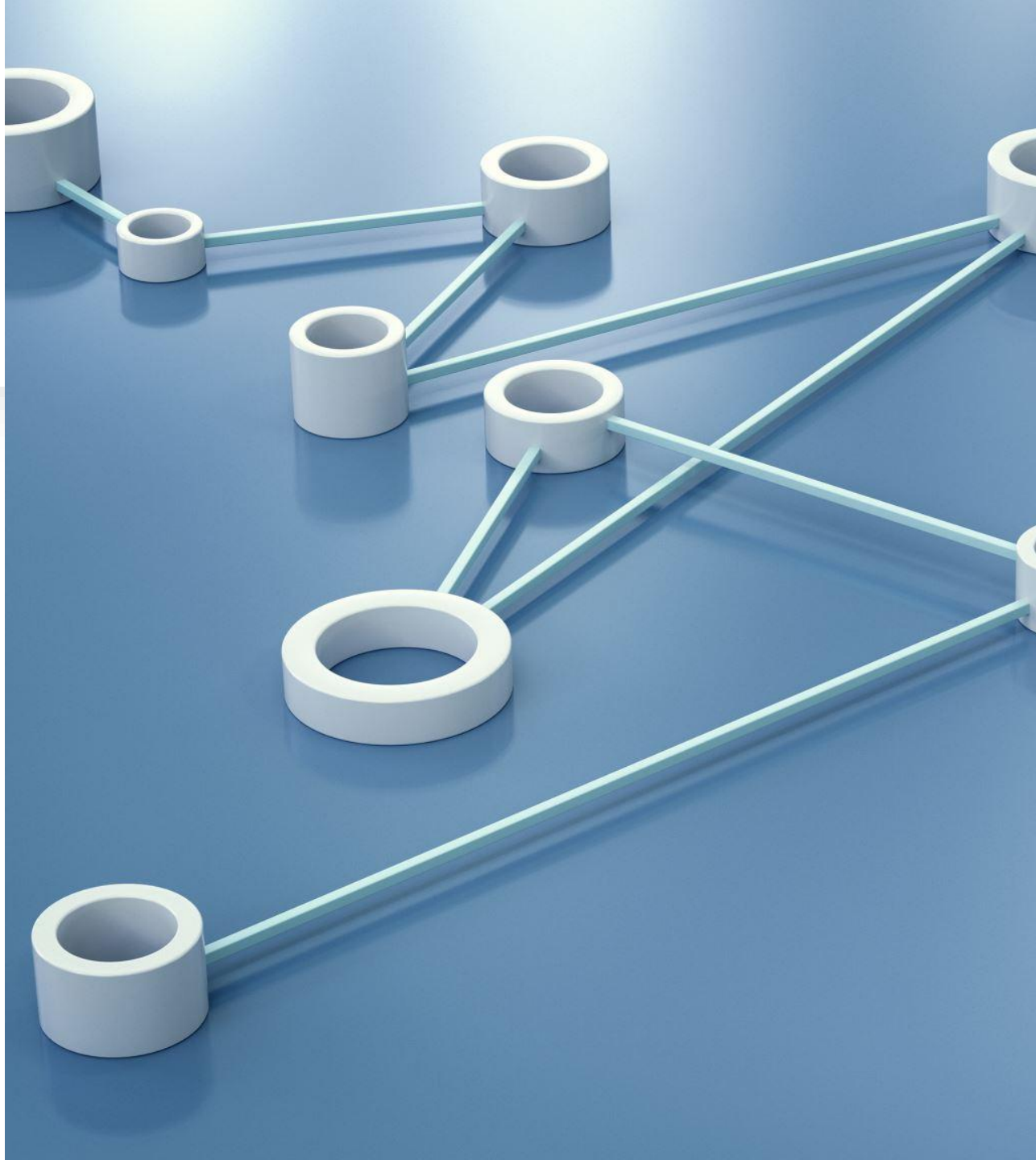
Private Vs Public Cloud





Cloud Service Models

- Cloud computing services are categorized into three main models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These models define the level of control and management provided to users.



Infrastructure as a Service (IaaS):

1. **Definition:** Provides virtualized computing resources over the internet.

2. **What it Offers:** Virtual machines, storage, networks, and operating systems.

3. **User Responsibility:** Manages OS, applications, middleware, and runtime.

4. **Examples:**
   - Amazon Web Services (AWS) EC2 (Elastic Compute Cloud).
   - Google Compute Engine (GCE).
   - Microsoft Azure Virtual Machines.

5. **Use Cases:**
   - Hosting websites and applications.
   - Storage and backup solutions.
   - Development and testing environments.

Infrastructure as a Service (IaaS):

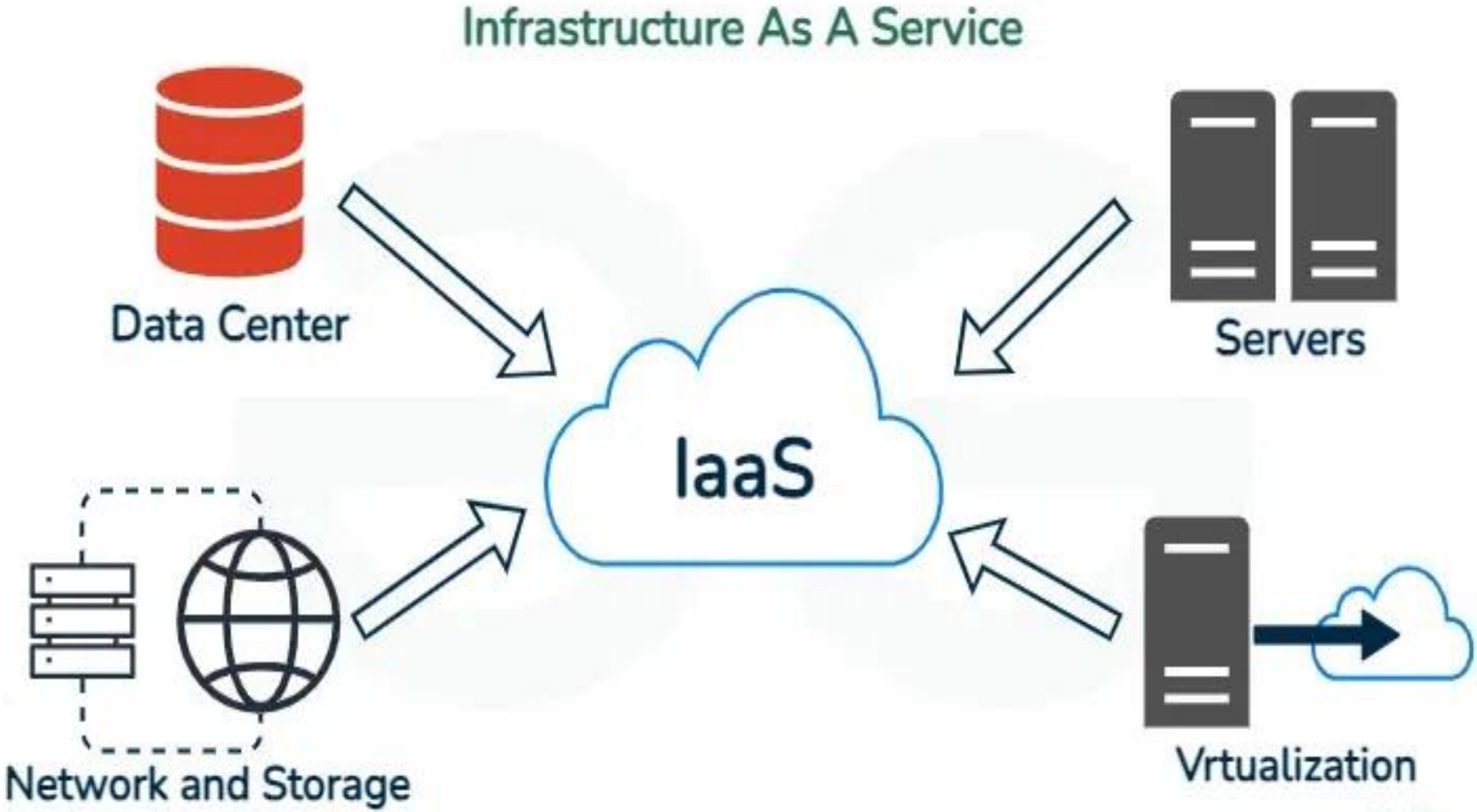
- Infrastructure as a Service (IaaS) is a cloud computing service model that gives virtualized computing resources over the web, with IaaS, associations can get to and manage versatile infrastructure assets like virtual machines, storage, and networking administration parts without the need to put resources into or keep up with actual equipment.
- IaaS allows business to outsource their whole IT infrastructure to a cloud service provider, empowering them to arrange, deploy, and manage computing resources on-demand, this adaptability allows organizations to increase their infrastructure or down in view of fluctuating interest, pay just for the resources they consume, and keep away from the expenses and intricacies related with customary on-premises infrastructure.

How does IaaS Architecture Work ?

- **On-Demand Access:** With IaaS, users can get to processing resources on-demand, allowing them to rapidly arrange and deploy infrastructure components depending on the situation. This disposes of the requirement for a forthright interest in equipment and empowers quick scaling to meet changing workload demands.
- **Self-Service Provisioning:** IaaS platforms offer self-support interfaces, for example, online interfaces or APIs, that empower users to freely arrangement and manage systems resources. This self-service model engages users to control their infrastructure deployments without depending on IT administrators.
- **Scalability:** IaaS platforms regularly offer level adaptability, allowing users to scale resources up or down based on demand, this adaptability ensures that associations can deal with changes in responsibility without encountering margin time or execution corruption.
- **Pay-Per-Use Billing:** IaaS providers normally utilize a pay-per-use billing model, where users are charged on their actual use of computing resources, this utilization based estimating model offers cost effectiveness, as associations just compensation for the resources they consume, as opposed to putting resources into excess limit.

What are the Types of Infrastructure As a Service Resources?

- **Virtual Machines (VMs):** Virtual machines are virtual instances of computing conditions that emulate the usefulness of physical servers. Users can arrangement VMs with specific configurations, including central processor, memory, storage, and operating systems, to run applications and administrations.
- **Networking:** IaaS platforms give organizing parts that empower clients to associate their virtualized infrastructure to the internet and establish communication between various resources, this includes virtual networks, subnets, firewalls, load balancers, and VPN gateways for managing network traffic and ensuring availability.
- **Load Balancers:** Load balancers convey incoming network traffic across numerous virtual machines or instances to advance execution, unwavering quality, and accessibility, they help uniformly distribute workloads and prevent overloading of individual resources, ensuring a smooth and steady user experience.
- **Databases:** A few IaaS suppliers offer managed database benefits that empower users to send and manage database in the cloud. These services incorporate relational databases like MySQL, PostgreSQL, and SQL Server, as well as NoSQL databases like MongoDB, Cassandra, and Redis.
- **Containers:** IaaS platforms may likewise offer help for containerized conditions, allowing users to deploy and manage containerized applications utilizing tools like Docker and Kubernetes, container services give a lightweight and versatile way to deal with application deployment and management, empowering quick turn of events and deployment of cloud-native applications.


Infrastructure As A Service
Data Center
Servers
IaaS
Network and Storage
Vrtualization

Platform as a Service (PaaS)

•**Definition:** Provides a complete development and deployment environment in the cloud.

•**What it Offers:** A platform for developers to build, run, and manage applications without managing the underlying infrastructure.

•**User Responsibility:** Manages the application and its data.

•**Examples:**

- Google App Engine.
- Microsoft Azure App Services.
- Heroku.

•**Use Cases:**

- Web application development.
- API development and hosting.
- Automated software deployment.

Platform As A Service (PaaS)

- Platform as a Service (PaaS) is a cloud computing model designed for developers, offering a complete environment to build, test and deploy applications. Unlike traditional infrastructure management, PaaS takes care of things like servers, storage and networking allowing developers to focus mainly on writing code and delivering applications quickly.
- In the cloud computing ecosystem, PaaS acts as a middle layer between Infrastructure as a Service (IaaS) and Software as a Service (SaaS). While IaaS provides the fundamental infrastructure like servers and storage, and SaaS delivers ready-made applications, PaaS provides developers with the necessary tools and environment to create custom applications from scratch.

How does Platform as a Service(PaaS) work?

1. **Core Infrastructure:** PaaS is built on cloud infrastructure provided by platforms like AWS, Microsoft Azure and Google Cloud. The provider handles everything behind the scenes, including servers, storage, and networking.
    - Servers: The provider manages hardware, load balancing, and scaling for you.
    - Storage: Applications and data are stored in secure cloud data centers.
    - Networking: The provider ensures secure, fast communication between resources.
2. **Built-In Platform Service:** On top of the infrastructure, PaaS offers all the tools and services you need to develop and run applications:
    - Operating Systems: Pre-configured systems like Linux or Windows.
    - Runtime Environments: Ready-to-use environments for languages like Java, Python, Node.js, Ruby or .NET.
    - Middleware: Services like caching, authentication, and messaging for applications.
    - Development Tools: Access to code editors, debugging tools, and CI/CD pipelines to streamline coding and deployment.

How does Platform as a Service(PaaS) work?

3. **Simplified Development and Deployment:** PaaS takes care of the heavy lifting in the development process
   - Development: You can write code using built-in frameworks and tools. For example, a developer might use Node.js and connect it to a pre-configured MySQL database.
   - Testing: Applications can be tested in sandbox environments that simulate real-world conditions.
   - Deployment: PaaS automates the deployment process with CI/CD pipelines, making it easy to push updates and changes.
4. **Automatic Scalability:** One of the best features of PaaS is its ability to scale based on traffic:
   - Horizontal Scaling: Adds more application instances to handle increased demand.
   - Vertical Scaling: Boosts the resources (e.g., CPU or RAM) of an existing instance.

How does Platform as a Service(PaaS) work?

5. **Easy Integration with Databases and APIs:** PaaS makes connecting to databases and third-party services straightforward:
   - Databases: Whether it's SQL (like PostgreSQL) or NoSQL (like MongoDB), PaaS simplifies setup and management.
   - APIs: You can easily integrate external services like payment systems or analytics tools to enhance your application.
6. **Built-In Security:** Security is handled by the provider, so developers can focus on building their applications:
   - Data Encryption: Ensures that data is secure both during transfer and at rest.
   - Access Control: Role-based permissions and identity management tools are included.
   - Compliance: Many providers follow regulations like GDPR or HIPAA to meet legal and industry requirements.

IaaS vs PaaS vs SaaS

Feature	IaaS	PaaS	SaaS
Definition	Provides virtualized computing resources like servers, storage, and networking.	Offers a platform with tools and environments for application development.	Delivers ready-to-use software applications over the internet.
Control Level	High: Users manage OS, middleware, apps, and data.	Medium: Users control apps and data; the provider manages infrastructure.	Low: Users only manage software configuration and usage.
Examples	AWS EC2, Microsoft Azure VM, Google Compute Engine.	AWS Elastic Beanstalk, Google App Engine, Heroku.	Google Workspace, Salesforce, Dropbox.
Target Users	IT administrators, developers requiring full control of infrastructure.	Developers looking for a managed platform to build and deploy applications.	End-users needing ready-to-use applications without technical expertise.
Use Cases	Hosting websites, storage, disaster recovery, virtual machines.	Software development, app testing, and deployment.	Email, CRM, file sharing, and collaboration tools.
Infrastructure Access	Provides direct access to virtualized hardware.	Abstracts the infrastructure, offering tools and frameworks.	No access to underlying infrastructure.

Software as a Service (SaaS)

- **Definition:** Delivers software applications over the internet on a subscription basis.

- **What it Offers:** Complete software solutions managed by the service provider.

- **User Responsibility:** Only uses the software without worrying about updates, security, or maintenance.

- **Examples:**
  - Google Workspace (Gmail, Google Drive).
  - Microsoft 365 (Word, Excel, PowerPoint).
  - Salesforce CRM.

- **Use Cases:**
  - Email communication.
  - Customer relationship management (CRM).
  - Office productivity.

Software as a Service (SaaS)

- In today's competitive market, businesses must adopt flexible, scalable and cost-effective solutions to stay ahead. Cloud services like Google Cloud, AWS, and Microsoft Azure offer these solutions. Among them, Amazon Web Services (AWS) leads the market, offering a range of services such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS).
- AWS’s SaaS solutions help businesses improve operations by making them more scalable and cost-effective while reducing infrastructure and maintenance costs. SaaS provides software applications over the Internet on a subscription basis, eliminating the need for local installation and management. It is commonly used in customer relationship management (CRM), project management, and collaboration tools.

How does a SaaS work?

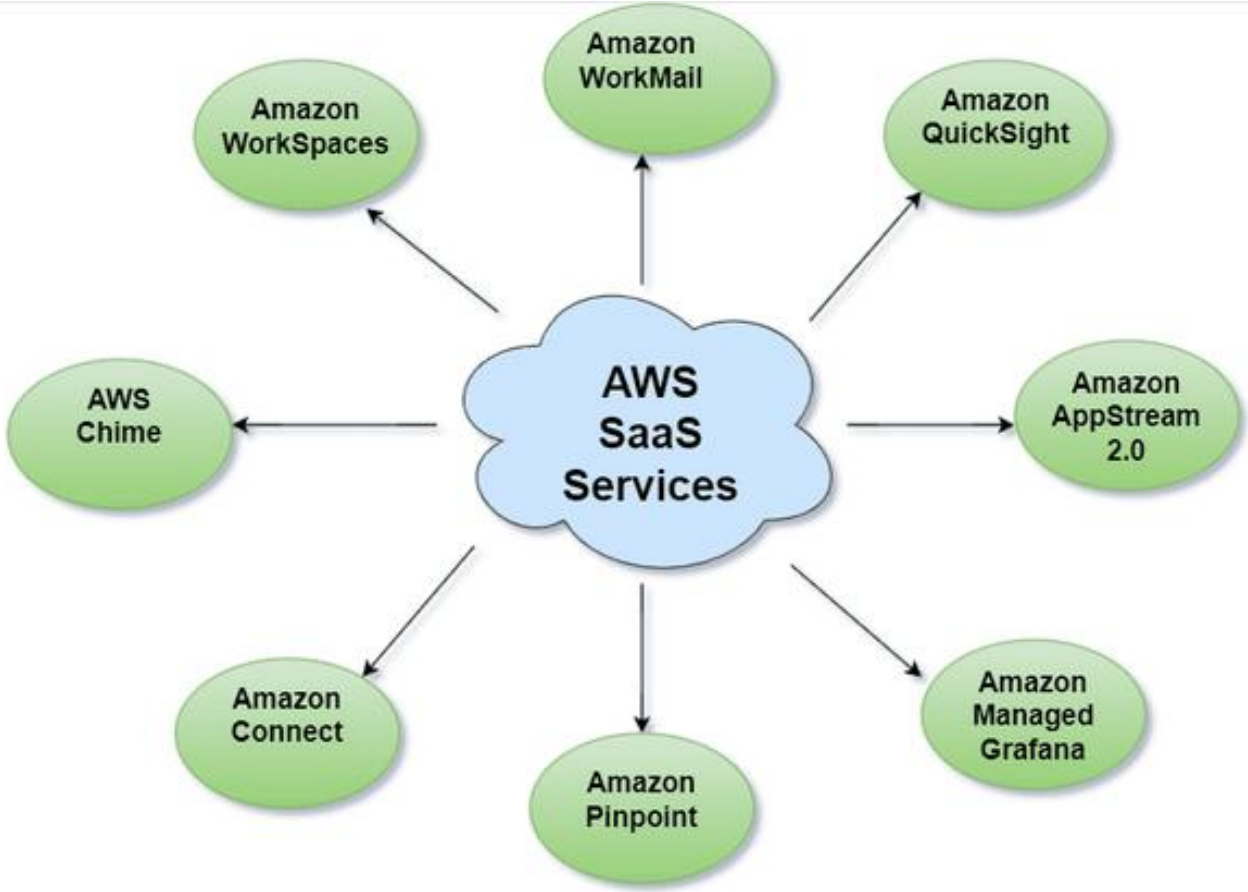
- Overview of SaaS Delivery Model

In a SaaS delivery model, the software is hosted on the provider's cloud infrastructure and made available to customers via the internet. Users access the application remotely through a browser, typically via a subscription-based pricing model. This setup eliminates the need for businesses to maintain their own servers, install software on each device or worry about software upgrades and patches.

- Cloud-based Access and Subscription Model

With SaaS, businesses only pay for the software they use often on a monthly or annual subscription basis. The subscription model is advantageous as it allows businesses to access cutting-edge software without upfront costs. Additionally, the software can be accessed from anywhere, enabling remote work and collaboration, which is especially important in today's dynamic business environment.

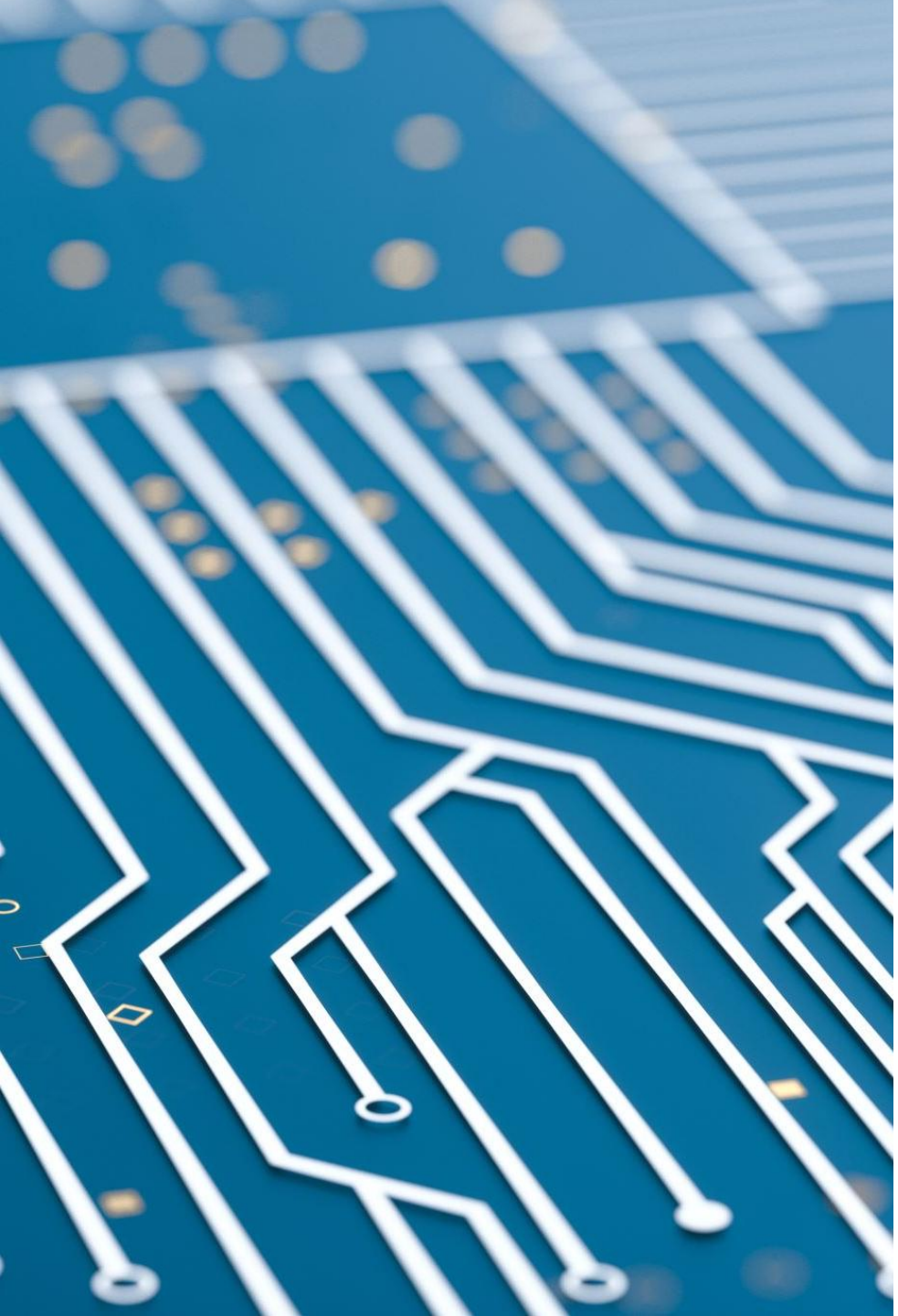
Top AWS SaaS Solutions for Modern Cloud Applications



Parallel & Distributed Computing Lecture # 4

BY,

DR. ALI AKBAR SIDDIQUE



Introduction to Performance Metrics

What are Performance Metrics?

Performance metrics are **quantitative measurements** used to evaluate the effectiveness of a parallel or distributed computing system. These metrics help determine how well a system utilizes resources such as CPUs, GPUs, memory, and bandwidth to solve computational problems efficiently.

Definition

- In parallel and distributed computing, performance metrics are used to measure and analyze how efficiently algorithms and systems perform when using multiple processing elements.
- These metrics are crucial for identifying bottlenecks, comparing systems, and optimizing applications.

Why Are They Important?

- Help determine whether parallelization improves performance.
- Allow comparison between different algorithms, architectures, or implementations.
- Guide developers to optimize resource usage and minimize computation time.
- Essential for scaling applications across larger systems.

Common Scenarios Where Performance Metrics Are Used

- Evaluating parallel algorithms in HPC (High Performance Computing) environments.
- Monitoring distributed workloads in cloud computing.
- Optimizing real-time systems and simulations.
- Benchmarking tools and profilers to identify critical code paths.

Efficiency in Parallel Computing

What is Efficiency?

- Efficiency measures how well a parallel system uses its resources. It is the ratio of speedup to the number of processors used. A high-efficiency value indicates that the system is using its processors effectively.
- Efficiency (E) is a performance metric that evaluates the degree of resource utilization in a parallel computing system.
- It shows how much useful work each processor is doing relative to the work done in the sequential version of the program.

$$\text{Efficiency (E)} = \frac{\text{Speedup (S)}}{\text{Number of Processors (P)}}$$

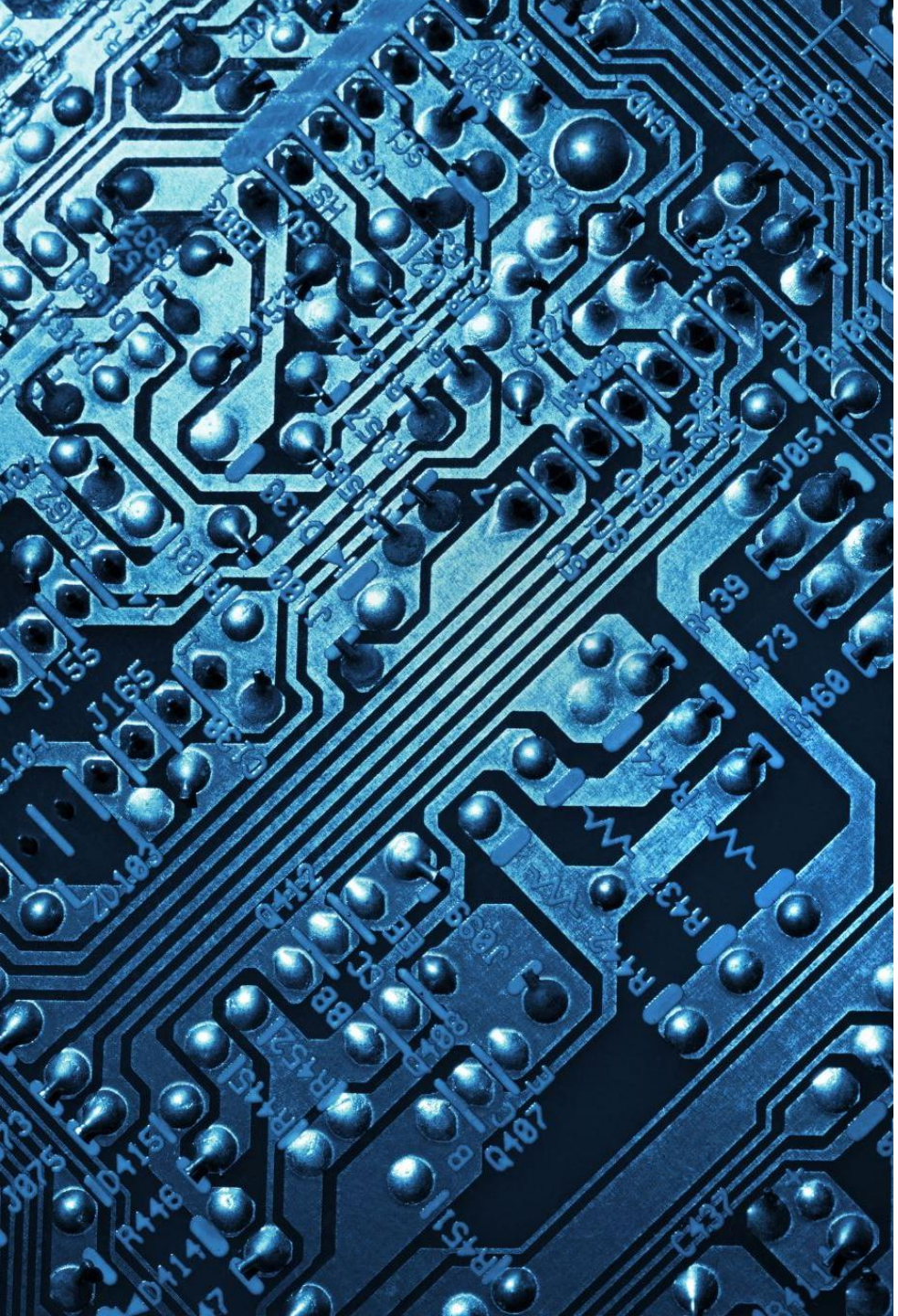
- Where:

  S = Speedup = $\frac{T_1}{T_P}$

  P = Number of processors

  T_1 = Time taken by sequential execution

  T_P = Time taken by parallel execution on P processors



Interpretation

- E = 1 (or 100%): Perfect efficiency (each processor contributes equally).
- E < 1: Some degree of waste or overhead, such as communication or synchronization delays.
- Efficiency usually decreases as the number of processors increases due to these overheads.

Why Efficiency Matters:

Helps identify if adding more processors is beneficial or just adding overhead.

Assists in resource allocation and cost justification in parallel/distributed systems.

Important when working on scalable applications—you want to maintain efficiency even as you scale up.

Real-World Analogy

- Think of a group project: If four students complete a project in 1/4th the time it takes one student alone, they’re 100% efficient. If they only save a little time, they might be stepping on each other's toes (inefficient!).

Numerical

Q: A program takes 100 seconds to execute on a single processor and 50 seconds on 2 processors. Calculate the efficiency.

Solution:

- Speedup $S = \frac{100}{50} = 2$
- Efficiency $E = \frac{2}{2} = 1.0 \rightarrow 100\%$

Numerical

Q: A parallel program takes 120 seconds on 1 processor and 40 seconds on 4 processors. Find the efficiency.

Solution:

- Speedup $S = \frac{120}{40} = 3$
- Efficiency $E = \frac{3}{4} = 0.75 \rightarrow 75\%$

Numerical

Q: A job takes 90 seconds on a single core and 30 seconds on 5 cores. Compute efficiency.

Solution:

- Speedup $S = \frac{90}{30} = 3$
- Efficiency $E = \frac{3}{5} = 0.6 \rightarrow 60\%$
- **Interpretation**: There is some overhead or idle time among processors.

Numerical

Q: A 6-processor system is operating at 80% efficiency. What is the speedup?

Solution:

- Efficiency $E = \frac{S}{P} \Rightarrow S = E \times P = 0.8 \times 6 = 4.8$
- **Speedup = 4.8**

Numerical

Q: A program has 80% of its code parallelizable. What is the efficiency when using 4 processors? (Use Amdahl's Law: $S = \frac{1}{(1-f)+\frac{f}{P}}$)

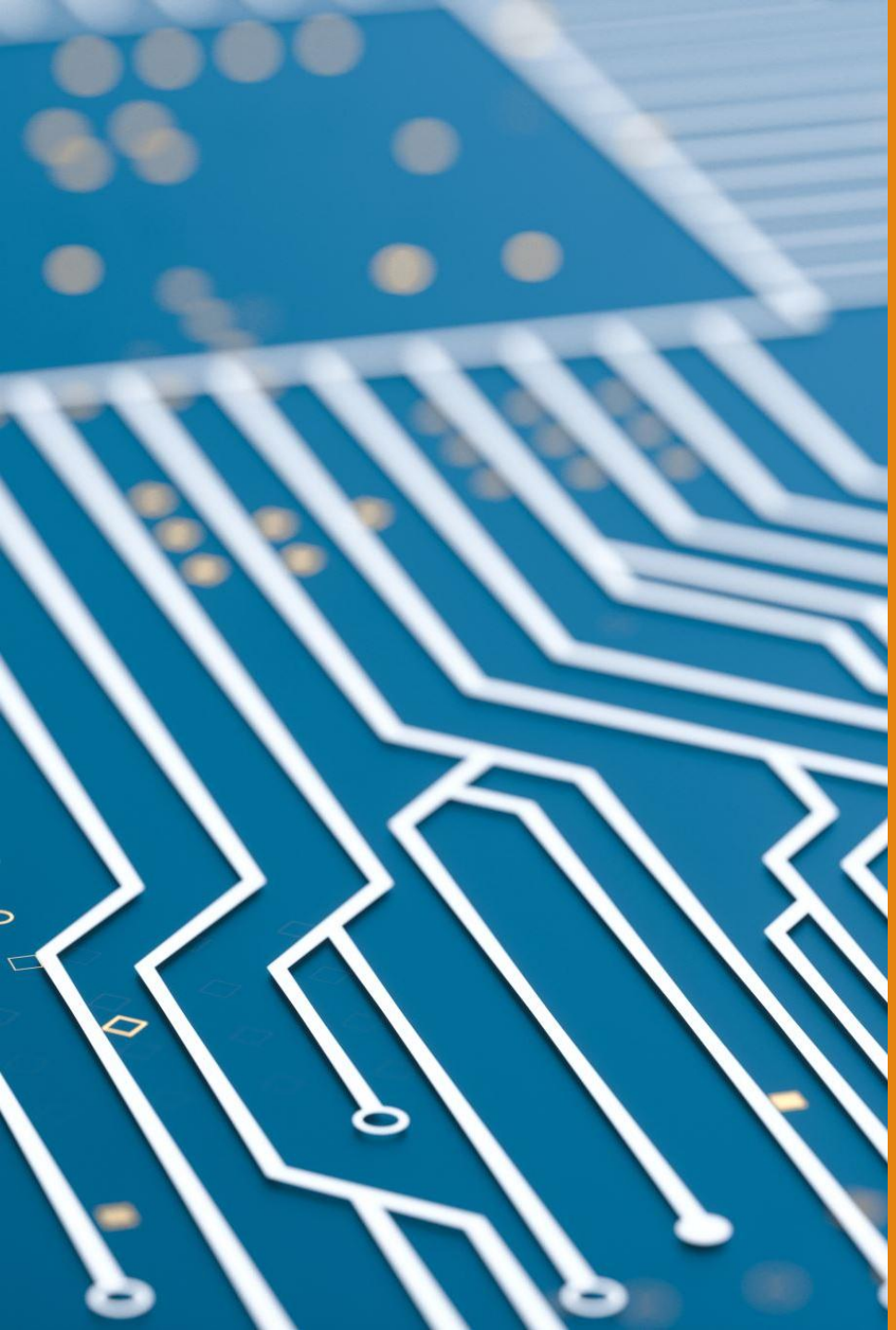
Where:

- $f = 0.8$ (parallel fraction)
- $P = 4$

Solution:

$$S = \frac{1}{(1 - 0.8) + \frac{0.8}{4}} = \frac{1}{0.2 + 0.2} = \frac{1}{0.4} = 2.5$$

- Efficiency $E = \frac{2.5}{4} = 0.625 \rightarrow 62.5\%$



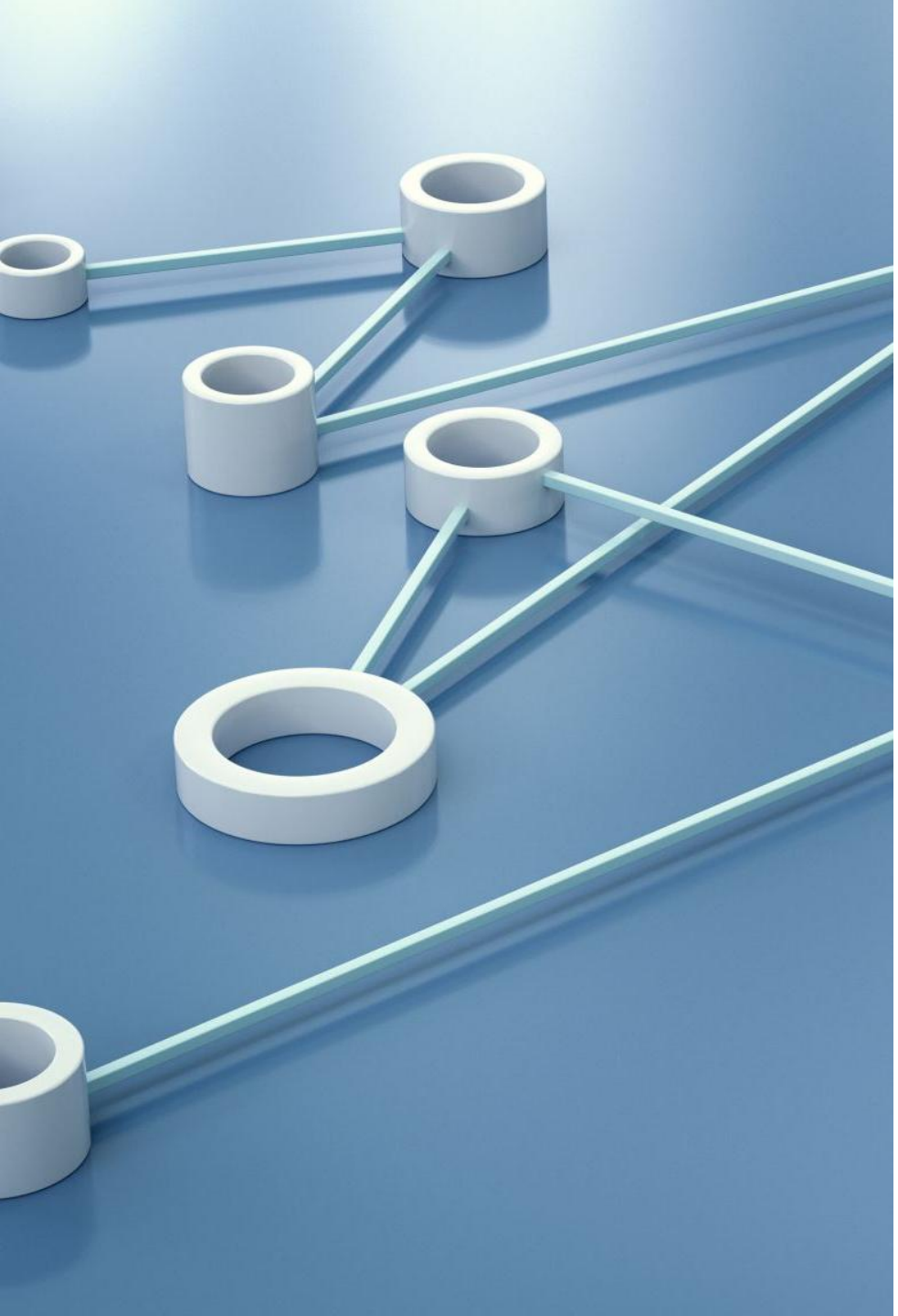
Scalability in Parallel Systems

Scalability refers to the ability of a parallel system to achieve improved performance (typically, speedup) as the number of processors increases. It measures how effectively a system can scale up in terms of performance with the addition of computing resources.

Types of Scalability:

1. **Strong Scalability**:
   - The problem size remains **fixed**, and performance improves as more processors are added.
   - Ideal for tasks with heavy computation and low communication overhead.
   - Example: Weather simulation using a fixed-size model on more cores.

2. **Weak Scalability**:
   - The problem size **increases proportionally** with the number of processors.
   - Measures how well the system can handle a growing workload.
   - Example: Processing more images in a distributed image processing task as more machines are added.



Why Scalability Matter?

- Ensures systems can handle larger problems or more users.
- Helps maximize hardware utilization and optimize cost-performance ratio.
- Critical for high-performance computing (HPC) applications and cloud-based infrastructures.

Real-World Insight:

A system with poor scalability may perform worse when more resources are added—known as **parallel slowdown**.

Amdahl's Law

Amdahl's Law provides a theoretical limit to the maximum speedup achievable when only part of a program can be parallelized. It highlights the impact of the sequential portion of a task on overall performance improvement.

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

Where:

- **P** = Proportion of the program that can be parallelized
- **1 - P** = Proportion of the program that remains sequential
- **N** = Number of processors

Interpretation

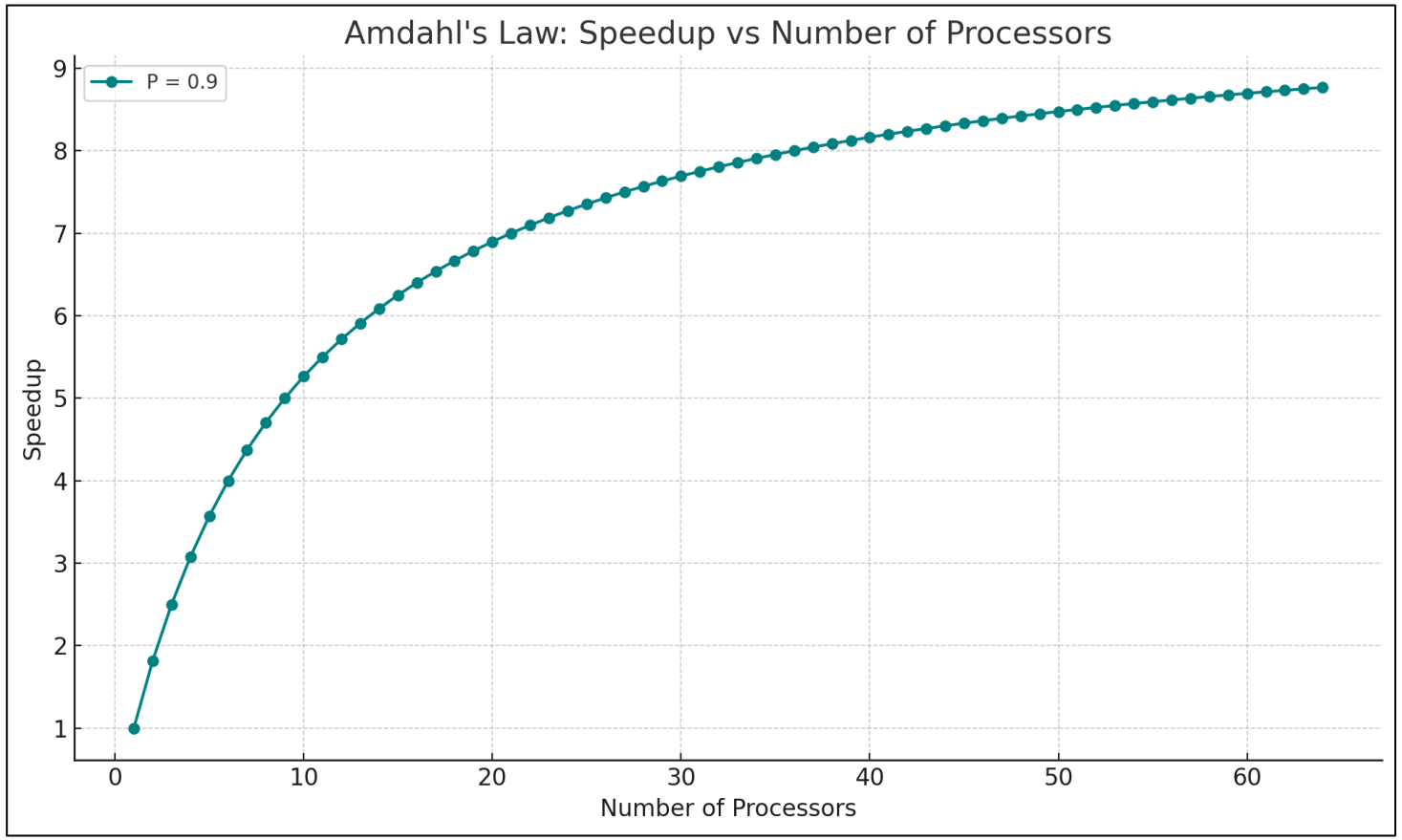
- No matter how many processors are added, the **sequential part** limits speedup.
- If **P = 0.9 (90%)** and **N = 10**, then:

$$S = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = \frac{1}{0.1 + 0.09} = \frac{1}{0.19} \approx 5.26$$

- Even with **infinite processors**, the max speedup is:

$$S = \frac{1}{1 - P}$$

Amdahl’s Law (Speed-up Vs Nos of Processors)



Here's a graphical representation of Amdahl's Law showing how speedup improves with the number of processors when 90% (P = 0.9) of the code is parallelizable. As shown, the speedup increases with more processors but plateaus due to the serial portion of the code, illustrating the law’s limitations. Let me know if you'd like graphs for other values of P or for scalability comparisons too.

Real-World Analogy

Imagine cooking dinner:

- Boiling rice (parallel task): Can be done on many burners.
- Serving (sequential task): One person has to do it.
- Even with 10 chefs, if only one can serve, your total time won’t improve significantly.

1. Synchronous Communication

Definition:

Synchronous communication requires that **both the sender and the receiver are active at the same time**. When a sender transmits a message, it **waits (blocks)** until the message is received or acknowledged before proceeding.

It's like a **phone call** – both people must be available at the same time.

- **How It Works:**

  - The sender initiates communication and waits for a response or acknowledgment.
  - The receiver processes the request immediately.
  - Only after receiving the response does the sender resume its work.

- **Characteristics:**

  - **Blocking behavior**: The sender's execution halts during communication.
  - **Tight temporal coupling**: Requires coordination in time between sender and receiver.
  - **Reliable message ordering** (usually).

- **Use Cases:**

  - Remote Procedure Calls (RPCs)
  - HTTP/HTTPS APIs
  - Database operations (traditional client-server model)
  - Distributed transactions where strict consistency is required

- **Challenges:**

  - If the receiver is slow or fails, the sender remains blocked.
  - Latency becomes a bottleneck.
  - Scalability issues in large systems (many processes waiting).

2. Asynchronous Communication

Definition:

In asynchronous communication, the sender **does not wait** for the receiver to acknowledge the message. Instead, it **sends the message and continues** its execution immediately.

It's like **sending an email** – you don't need the other person online to send your message.

- **How It Works:**

  - The sender places the message in a message queue or buffer.
  - The receiver can process it at a later time.
  - There may be delays, but the systems remain decoupled in time.

- **Characteristics:**

  - **Non-blocking**: The sender continues immediately after sending.
  - **Loose coupling**: Components operate independently.
  - **More scalable** and resilient to receiver failures.

- **Use Cases:**

  - Message queues (RabbitMQ, Kafka, ZeroMQ)
  - Event-driven systems and microservices
  - Notification systems
  - IoT systems with intermittent connectivity

- **Challenges:**

  - Message ordering can become uncertain.
  - Delivery is not guaranteed without retries or acknowledgments.
  - Complex debugging and error-handling mechanisms required.