1 to 37 pages krny hai

---

## Parallel and Distributed Computing

- **Parallel Computing:** Multiple tasks are performed **simultaneously** using multiple processors within a **single computer system**.
- **Distributed Computing:** Tasks are performed across **multiple computers (nodes) connected via a network**, working together to solve a problem.

Think of it like this:

- Parallel computing → a single kitchen with many chefs working at the same time.
- Distributed computing → many kitchens in different locations working together to cook a huge feast.

---

## Flynn's Taxonomy

Flynn's Taxonomy is a classification of computer architectures based on the number of **instruction streams** and **data streams** they can handle **simultaneously**. It was introduced by Michael J. Flynn in 1966.

It has **four main types**:

| Type | Instruction Stream | Data Stream | Explanation | Real-World Example |
|---|---|---|---|---|
| **SISD** (Single Instruction, Single Data) | 1 | 1 | Traditional sequential computer. Executes **one instruction on one data item at a time**. | Typical desktop PC running simple programs |
| **SIMD** (Single Instruction, Multiple Data) | 1 | Many | Executes **same instruction on multiple data simultaneously**. Great for vector/matrix operations. | Graphics Processing Units (GPUs), Image processing, Video rendering |
| **MISD** (Multiple Instruction, Single Data) | Many | 1 | Rare. Multiple instructions operate on the **same data**. | Fault-tolerant systems, some redundant aerospace computations |
| **MIMD** (Multiple Instruction, Multiple Data) | Many | Many | Multiple processors execute **different instructions on different data** simultaneously. | Modern multi-core CPUs, distributed computing clusters, cloud computing |

---

**Real-World Analogy**

- **SISD:** One chef cooking one dish step by step.
- **SIMD:** One chef using several hands (or many assistants) to stir multiple pots in the same way at once.
- **MISD:** Multiple chefs all cooking different recipes but using the **same ingredient**.
- **MIMD:** Multiple chefs each cooking different dishes in parallel, independently.


Sure! Let's go **step by step**, explaining all these topics clearly and with real-world examples where possible.

---

# 1. Flynn's Taxonomy

Flynn's Taxonomy classifies computer architectures based on **instruction streams** (how many instruction sequences) and **data streams** (how many data items processed) simultaneously.

| Type | Instruction Stream | Data Stream | Explanation | Real-World Example |
|---|---|---|---|---|
| **SISD** (Single Instruction, Single Data) | 1 | 1 | Executes **one instruction on one data item at a time**. | Normal PC running Word or Excel |
| **SIMD** (Single Instruction, Multiple Data) | 1 | Many | Executes the **same instruction on multiple data elements simultaneously**. | GPUs for image/video processing, AI neural networks |
| **MISD** (Multiple Instruction, Single Data) | Many | 1 | Multiple instructions operate on the **same data**. Rare. | Fault-tolerant systems in aerospace (redundant computations) |
| **MIMD** (Multiple Instruction, Multiple Data) | Many | Many | Multiple processors execute **different instructions on different data**. | Multi-core CPUs, distributed cloud servers, supercomputers |

---

# 2. Physical Organization of Parallel Platforms

Parallel computing platforms are organized in terms of **how processors and memory are connected**. Main types:

1. **Shared Memory Systems**
   - All processors share a **common memory**.
   - Easier to program.
   - Limitation: Memory access contention.
   - Example: Multi-core CPU in a PC.
2. **Distributed Memory Systems**
   - Each processor has its **own private memory**.
   - Processors communicate via a **network** (message passing).
   - Example: Cluster of computers running MPI (Message Passing Interface).
3. **Hybrid Systems**
   - Combination of shared and distributed memory.
   - Example: Modern supercomputers (e.g., nodes with multi-core CPUs, each node connected via high-speed network).

---

# 3. PRAM (Parallel Random Access Machine)

- **PRAM** is an abstract model used to **design parallel algorithms**.
- **Concept:** Multiple processors can access a **shared memory** simultaneously.
- **Types based on concurrent access:**
  1. **EREW (Exclusive Read, Exclusive Write):** No simultaneous reads or writes.

2. **CREW (Concurrent Read, Exclusive Write):** Multiple processors can read at the same time, but writes are exclusive.
3. **CRCW (Concurrent Read, Concurrent Write):** Multiple processors can read/write simultaneously. Conflicts are resolved by rules.

**Example:** Solving large matrix multiplication in parallel.

---

# 4. Routing Techniques and Costs

Routing refers to **how messages are sent between processors** in a parallel or distributed system.

**Routing Techniques:**

1. **Static Routing**
   - Path is determined **before execution**.
   - Simple, predictable.
   - Limitation: Not flexible for dynamic load.
2. **Dynamic Routing**
   - Path is determined **during execution**.
   - Adapts to network congestion.
   - Example: Internet packet routing.

**Costs in Parallel Systems:**

1. **Communication Cost**
   - Time to send a message between processors.
   - Depends on distance, network speed, and message size.
2. **Computation Cost**
   - Time taken to perform calculations.
3. **Total Execution Time**
   - **Parallel Time = Computation Time + Communication Time**

**Example:** In a cluster, sending data from node A to node B over a network may take 5 ms (communication) while processing the data takes 20 ms (computation).

---

✅ **Summary Table of Key Concepts**

| Concept | Key Idea | Example |
|---|---|---|
| SISD | 1 instruction, 1 data | Single-core PC |
| SIMD | 1 instruction, multiple data | GPU |
| MISD | Multiple instructions, 1 data | Redundant aerospace systems |

| Concept | Key Idea | Example |
|---|---|---|
| MIMD | Multiple instructions, multiple data | Cloud servers |
| PRAM | Abstract shared-memory model | Parallel matrix multiplication |
| Routing | Path of message in network | Static vs dynamic routing |
| Cost | Communication + computation | Cluster message passing |

Go to pg# 13

---

# Quick Review to the Previous Lecture

- **Amdahl's Law of Parallel Speedup**
  - Purpose, derivation, and examples
- **Karp-Flatt Metric**
  - Finding sequential fraction in the given parallel setup
- **Types of Parallelism**
  - Data-parallelism
    - Same operation on different data elements
  - Functional-parallelism
    - Different independent tasks with different operations on different data elements can be parallelized
  - Pipelining
    - Overlapping the instructions in a single instruction cycle to achieve parallelism

## 1. Amdahl's Law of Parallel Speedup

### Purpose

- Amdahl's Law gives a **theoretical limit on the speedup** you can achieve by parallelizing a task.
- It shows that the speedup is limited by the **sequential portion** of the program.

---

### Formula / Derivation

Let:

- $S$ = Fraction of the program that is **sequential** (cannot be parallelized)
- $P$ = Fraction that is **parallelizable** ($P = 1 - S$)
- $N$ = Number of processors
- $\text{Speedup}$ = How many times faster the parallel version is compared to sequential

**Speedup formula:**

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

**Step-by-step derivation:**

1. Total execution time sequential: 1 unit
2. Sequential part takes $S$ units → cannot be improved
3. Parallel part takes $P$ units → divided among N processors → $P/N$
4. Total parallel time: $T_{\text{parallel}} = S + P/N$
5. Speedup = Sequential time / Parallel time = $1/(S + P/N)$

---

### Example

- Suppose 30% of a program is sequential ($S = 0.3$) and 70% can be parallelized ( $P = 0.7$)
- Using **4 processors**:

**Example**

- Suppose 30% of a program is sequential ($S = 0.3$) and 70% can be parallelized ( $P = 0.7$)
- Using **4 processors**:

$$\text{Speedup} = \frac{1}{0.3 + \frac{0.7}{4}} = \frac{1}{0.3 + 0.175} = \frac{1}{0.475} \approx 2.11$$

✅ Even with 4 processors, the speedup is **just 2.11x** because the sequential part limits performance.

---

## 2. Karp-Flatt Metric

- Measures the **apparent serial fraction** of a parallel program.
- Helps **identify overheads** like communication or load imbalance.

**Formula:**

$$e = \frac{\frac{1}{\text{Speedup}} - \frac{1}{N}}{1 - \frac{1}{N}}$$

Where:

- $e$ = apparent serial fraction
- $N$ = number of processors
- Speedup = measured parallel speedup

**Purpose:** Shows how much **non-parallelizable work + overhead** is present.

**Example:**

**Example:**

- Measured speedup on 4 processors = 3.2

$$e = \frac{\frac{1}{3.2} - \frac{1}{4}}{1 - \frac{1}{4}} = \frac{0.3125 - 0.25}{0.75} = 0.0833$$

✅ Only **8.33% of the program behaves as sequential**.

---

## 3. Finding Sequential Fraction

- From **Amdahl's Law**, given speedup $S_p$ and processors $N$:

$$S = \frac{1/S_p - 1/N}{1 - 1/N}$$

- This is basically **using Karp-Flatt metric** to find sequential portion.

## 4. Types of Parallelism

### a) Data-Parallelism

- Same operation is applied to **different data elements** at the same time.
- Example:
  - Adding two arrays element-wise in parallel.
  - GPU rendering pixels of an image simultaneously.

---

### b) Functional-Parallelism

- Different operations run **independently on different data**.
- Example:
  - One processor compresses images, another encrypts text, another sorts a database, all simultaneously.

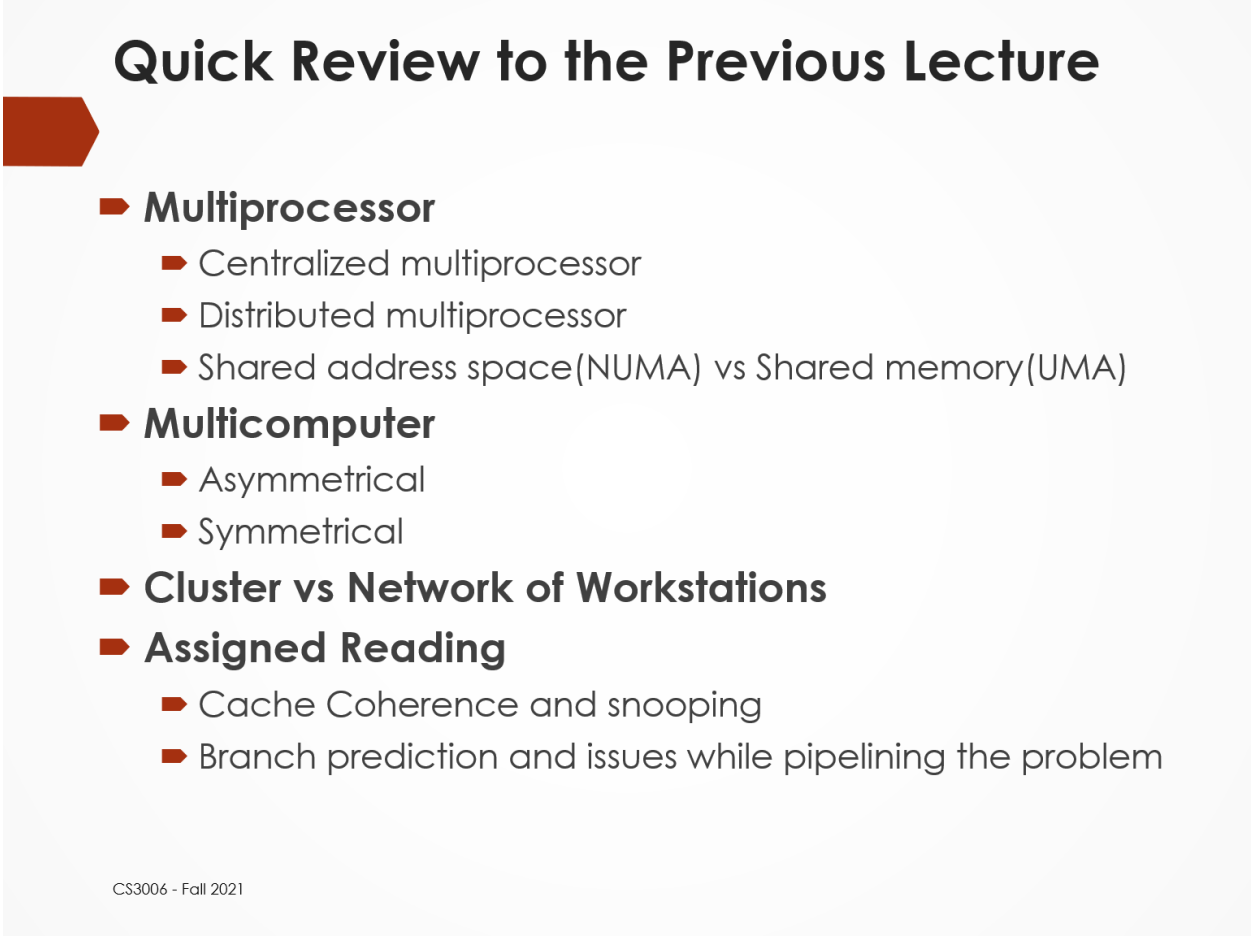
### c) Pipelining

- Overlaps execution of instructions to achieve parallelism **within a single instruction cycle**.
- Each stage of a pipeline handles a part of the task.
- Example:
  - In CPU: Instruction Fetch → Decode → Execute → Write Back
  - While one instruction is executing, the next can be decoded.

---

## ✅ Summary Table

| Concept | Key Idea | Example |
|---|---|---|
| Amdahl's Law | Limits speedup due to sequential part | 70% parallelizable program on 4 cores → Speedup ≈ 2.11 |
| Karp-Flatt Metric | Measures apparent sequential fraction | Speedup 3.2 on 4 processors → e ≈ 0.083 |
| Data-Parallelism | Same operation, different data | GPU pixel processing, array addition |
| Functional-Parallelism | Different operations on different data | Video encoding + encryption + sorting in parallel |
| Pipelining | Overlapping instruction execution | CPU instruction stages, assembly line analogy |




---

# 1. Multiprocessor

- A **multiprocessor system** has **more than one processor** working together to perform tasks.
- Advantages: Higher performance, reliability, throughput.
- Can be categorized as **centralized** or **distributed**.

---

# 2. Centralized Multiprocessor

- **Definition:** All processors are connected to a **single shared memory** via a common bus or interconnect.
- Processors share **resources** (memory, I/O).
- Types:
  - **UMA (Uniform Memory Access):** All processors access memory **equally fast**.

- **NUMA (Non-Uniform Memory Access):** Memory access time varies depending on location relative to processor.

**Example:** Multi-core CPUs, IBM POWER systems.

---

# 3. Distributed Multiprocessor (Multicomputer)

- Each processor has **its own private memory**.
- Processors communicate via a **network** (message passing).
- More scalable than centralized systems.

**Types:**

- **Symmetric Multiprocessing (SMP):** All processors are **equal**, share tasks.
- **Asymmetric Multiprocessing (AMP):** One master processor controls others.

**Example:** Cluster computing, supercomputers.

---

# 4. Shared Address Space vs Shared Memory

| Feature | UMA (Uniform Memory Access) | NUMA (Non-Uniform Memory Access) |
|---|---|---|
| Memory Access | Same latency | Different latency depending on memory location |
| Shared by | All processors equally | Processors may access local memory faster than remote memory |
| Example | Traditional multi-core CPU | High-end servers, NUMA architectures in Intel/AMD |

**Note:** UMA is simpler, NUMA scales better for large systems.

---

# 5. Multicomputer

- A collection of **independent computers** connected via a **network** to solve a common problem.
- Each has **its own CPU and memory**, communicates through messages.
- Example: Beowulf cluster, Google data centers.

---

# 6. Asymmetrical vs Symmetrical Multiprocessing

| Feature | Asymmetrical (AMP) | Symmetrical (SMP) |
|---|---|---|
| Control | One master processor | All processors equal |
| Task Allocation | Master assigns tasks | Any processor can execute any task |
| Complexity | Simpler | More complex |
| Example | Older IBM mainframes | Modern multi-core CPUs |

---

# 7. Cluster vs Network of Workstations (NOW)

| Feature | Cluster | Network of Workstations (NOW) |
|---|---|---|
| Hardware | Dedicated nodes | Existing workstations connected |
| Performance | High, designed for HPC | Medium, uses idle cycles |
| Communication | Fast, low-latency interconnect | Slower network (Ethernet) |
| Example | Supercomputer cluster | Volunteer computing (SETI@home) |

---

# 8. Cache Coherence and Snooping

- **Cache coherence:** Ensures **all caches have a consistent view of memory**.
- Problem: When multiple processors modify the same memory, other caches may have **stale data**.

**Snooping Protocol:**

- Each cache monitors (snoops) the **bus** for write operations.
- Keeps caches consistent.

**Example:** MESI protocol (Modified, Exclusive, Shared, Invalid states).

---

# 9. Branch Prediction & Pipelining Issues

- **Pipelining:** CPU executes instructions in stages (Fetch → Decode → Execute → Memory → Write Back).
- **Branch instructions** (if/else, loops) create uncertainty: CPU may fetch wrong instructions.
- **Branch Prediction:** CPU guesses the next instruction path to keep pipeline full.

**Issues if prediction fails:**

- Pipeline must **flush incorrect instructions** → Performance penalty.
- More mispredictions → more stalls.

**Example:** Modern CPUs like Intel and AMD use **dynamic branch prediction** to improve efficiency.

---

## ✅ Summary Table

| Concept | Key Idea | Example |
|---|---|---|
| Centralized Multiprocessor | Shared memory, one interconnect | Multi-core CPU |
| Distributed Multiprocessor | Private memory, communicates via network | Cluster computing |
| UMA vs NUMA | Memory access uniform vs non-uniform | UMA: multi-core CPU, NUMA: server CPU |
| Asymmetrical vs Symmetrical | Master-controlled vs equal processors | AMP: old IBM, SMP: modern CPU |
| Cluster vs NOW | Dedicated HPC vs networked PCs | Cluster: supercomputer, NOW: SETI@home |
| Cache Coherence | Ensure consistent data in caches | MESI protocol |
| Snooping | Monitor bus to maintain coherence | CPU cache systems |
| Branch Prediction | Guess next instruction to keep pipeline full | Intel/AMD CPUs |
| Pipeline Issues | Wrong guesses → flush pipeline | Mis-predicted branches |

---

1. SISD (Single Instruction, Single Data)

Executes one instruction at a time on one piece of data.

Sequential execution model (no parallelism).

Used in traditional uniprocessor systems.

Example Architecture:

Intel 8086

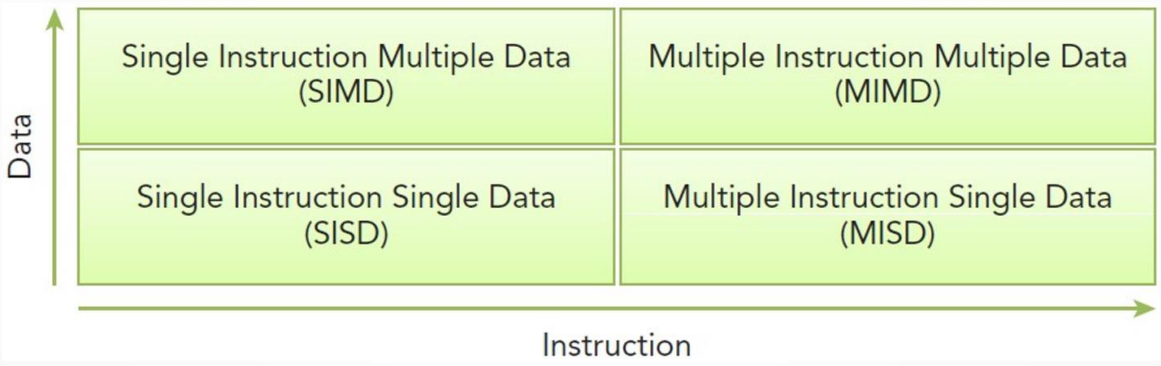
Old PCs or Microcontrollers

Diagram (Conceptual):

Instruction 1 → Processor → Data 1

# Flynn's Taxonomy

- Widely used architectural classification scheme
- Classifies architectures into four types
- The classification is based on how data and instructions flow through the cores.




🏵 2. SIMD (Single Instruction, Multiple Data)

One instruction operates on multiple data elements simultaneously.

Used for data-level parallelism (e.g., image processing, graphics).

Example Architecture:

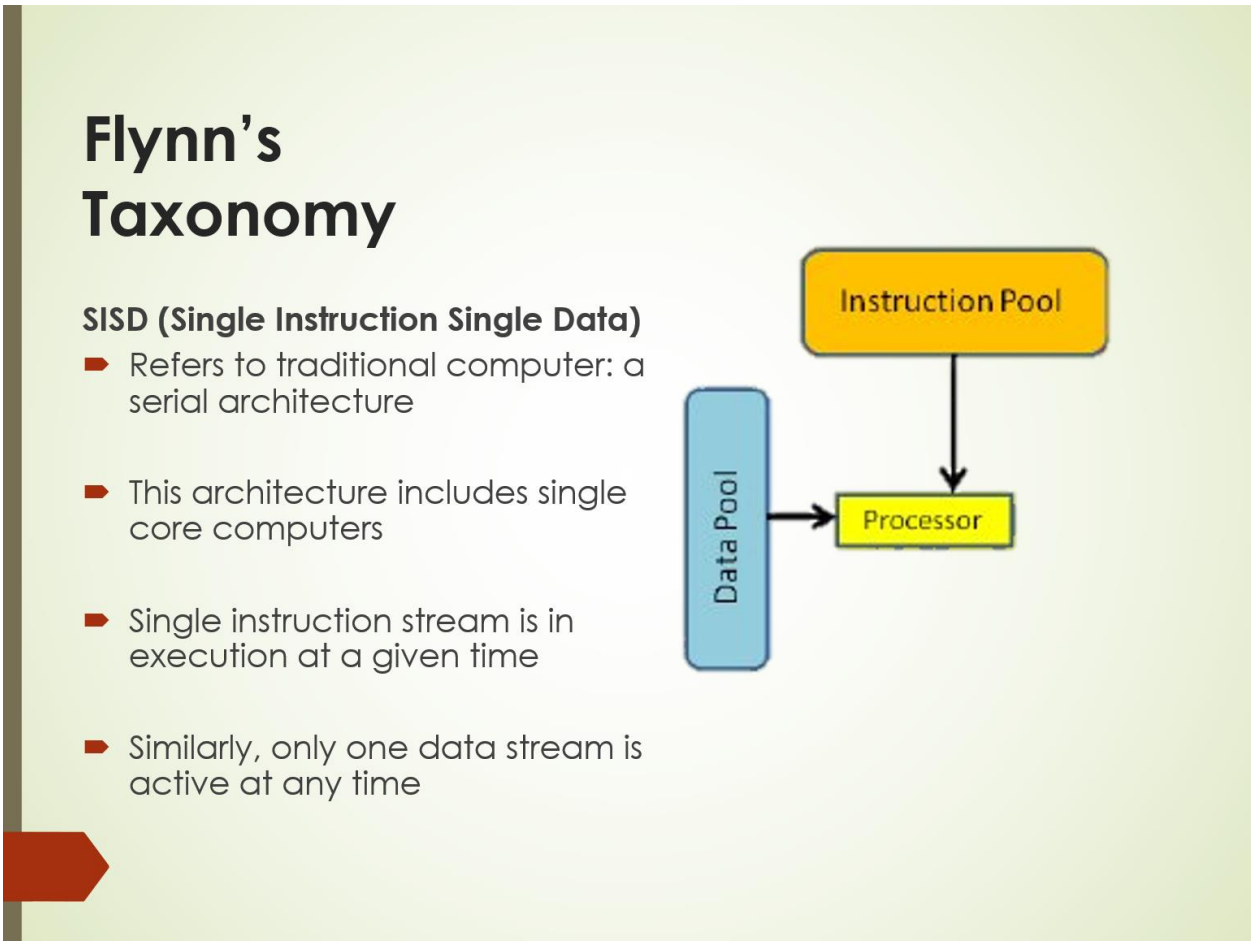
GPUs (Graphics Processing Units)

Vector processors (e.g., CRAY-1 supercomputer)

MMX/SSE instruction sets in Intel CPUs

Diagram:

Instruction 1 → [Processor 1 → Data 1]
[Processor 2 → Data 2]
[Processor 3 → Data 3]




Not parallel, classical Von Neumann architecture

Parallelism can be introduced using pipelining

🧩 3. MISD (Multiple Instruction, Single Data)

Multiple instructions operate on the same data.

Rare in real systems.

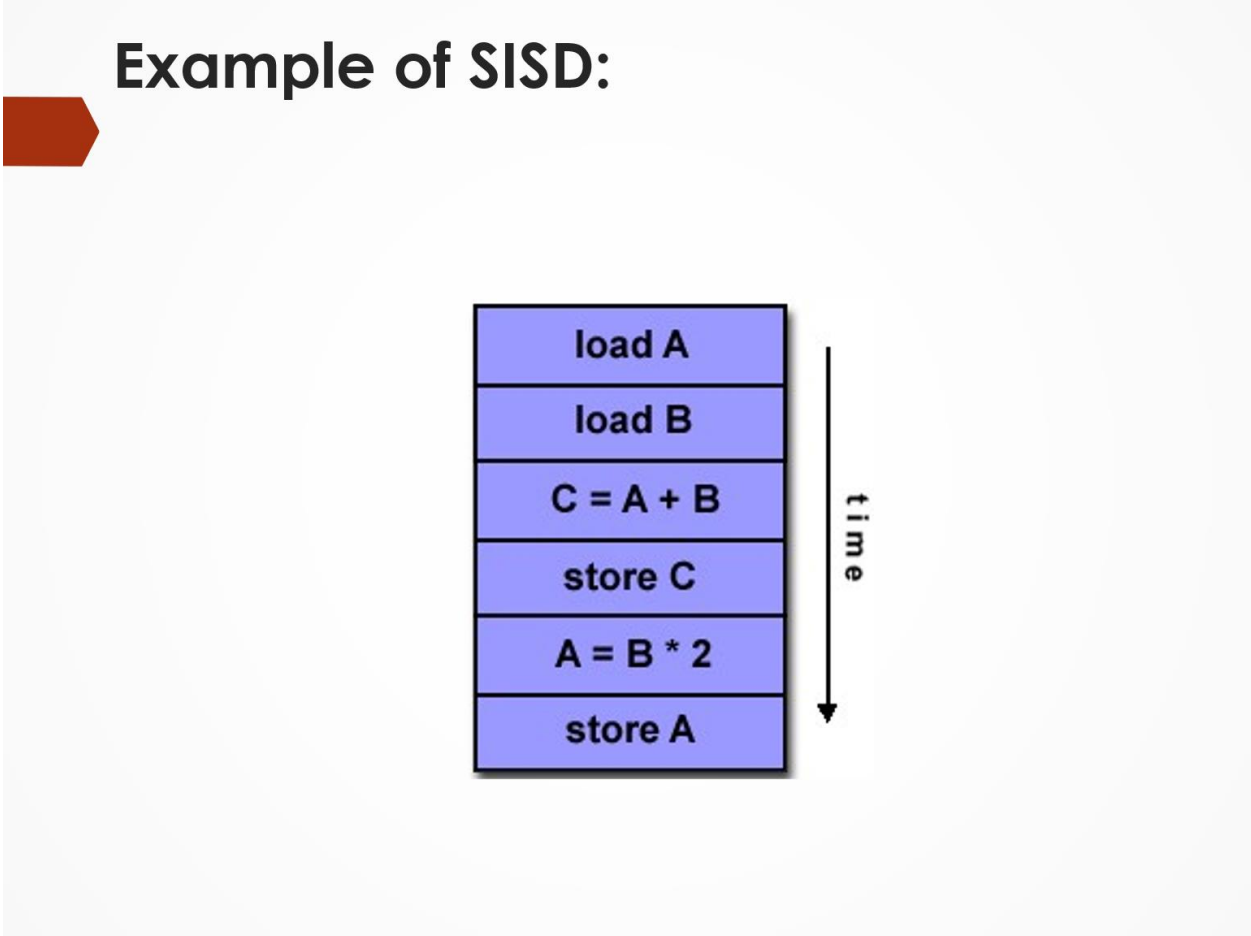
Used in redundant systems where multiple units perform different operations on the same data for fault tolerance.

Example Architecture:

Space shuttle flight control systems (fault-tolerant pipelines)

Diagram:

[Instruction 1, Instruction 2, Instruction 3] → Data 1




💠 4. MIMD (Multiple Instruction, Multiple Data)

Multiple processors executing different instructions on different data.

Provides task-level parallelism.

Common in modern multi-core and distributed systems.

Example Architecture:

Multi-core processors (e.g., Intel Core i7)

Cluster computers

Cloud computing systems

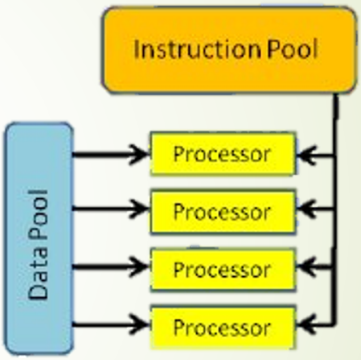
Diagram:

Instruction 1 → Processor 1 → Data 1
Instruction 2 → Processor 2 → Data 2
Instruction 3 → Processor 3 → Data 3

# Flynn's Taxonomy

**SIMD (Single Instruction Multiple Data)**

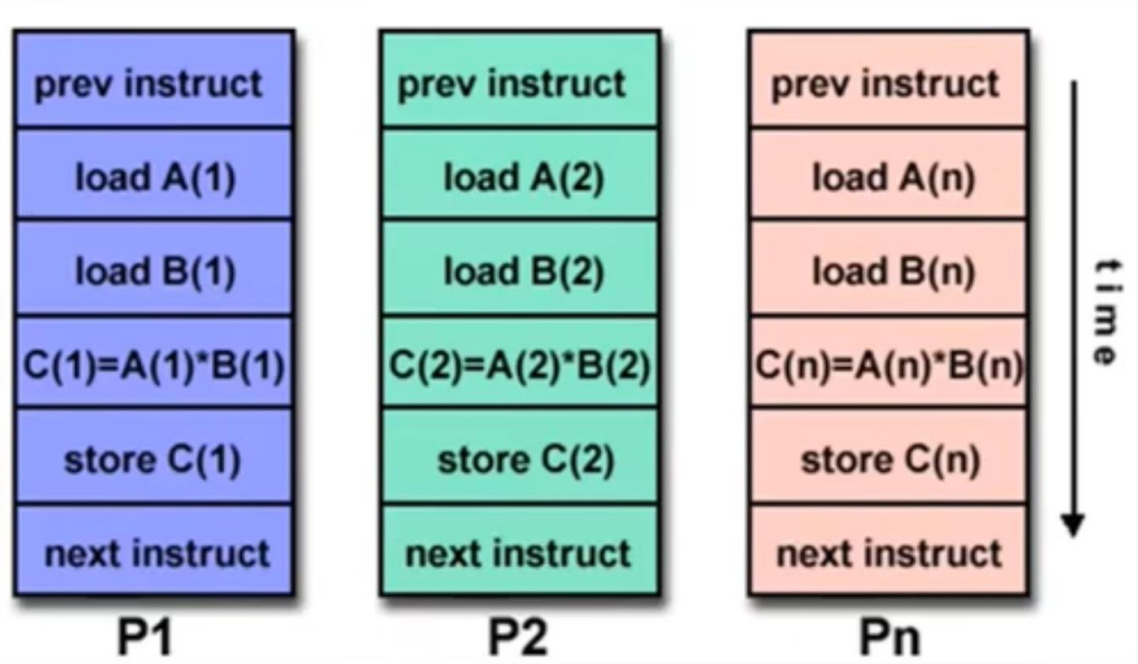
- Refers to parallel architecture with multiple cores
- All the cores execute the same instruction stream at any time but, data stream is different for the each.
- Well-suited for the scientific operations requiring large matrix-vector operations
- Vector computers (Cray vector processing machine) and Intel co-processing unit 'MMX' fall under this category.
- Used with array operations, image processing and graphics




Array: same operations on different array elements. Replaces the loops

Image: Applying same operation on different pixels

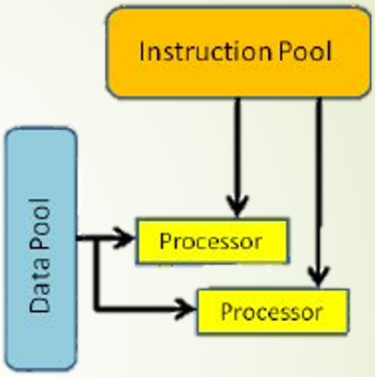
# Example of SIMD:



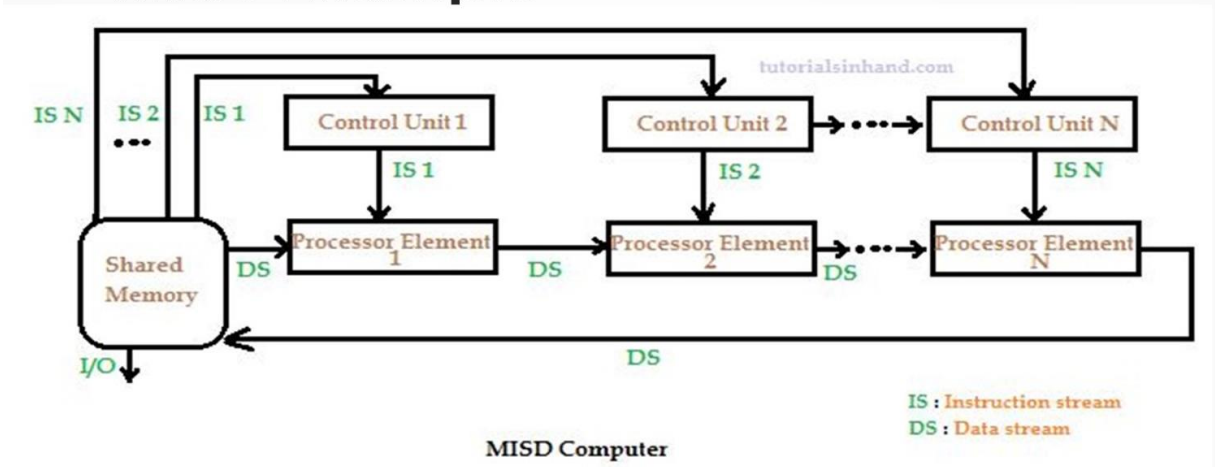
# Flynn's Taxonomy

**MISD (Multiple Instructions Single Data)**

- Multiple instruction stream and single data stream
  - A pipeline of multiple independently executing functional units
  - Each operating on a single stream of data and forwarding results from one to the next
- Rarely used in practice
- E.g., Systolic arrays : network of primitive processing elements that pump data.

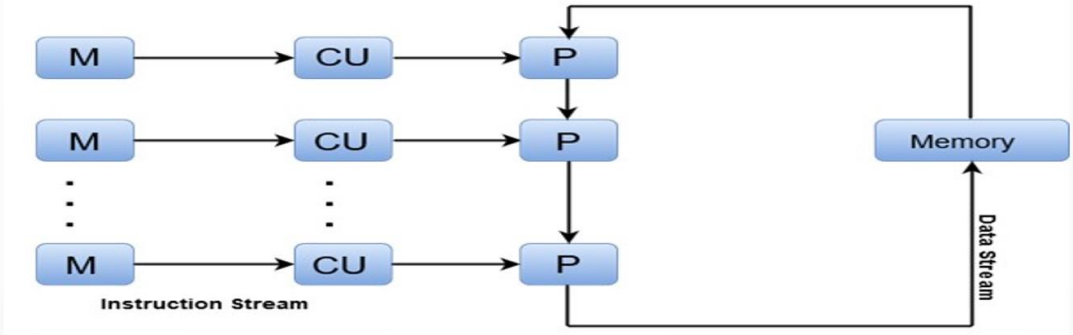


# MISD Example

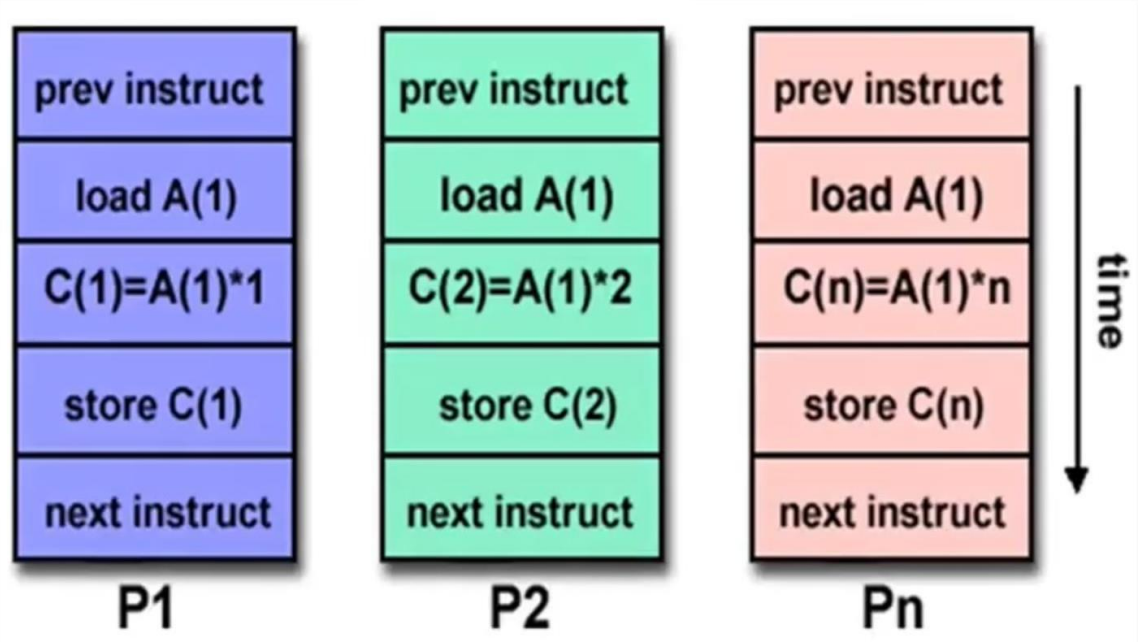



MISD Computer

**MISD:**



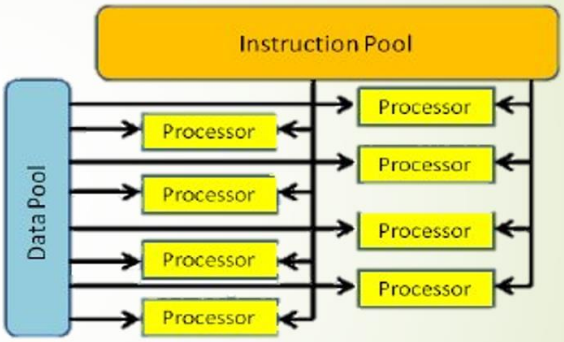
# Example of MISD:



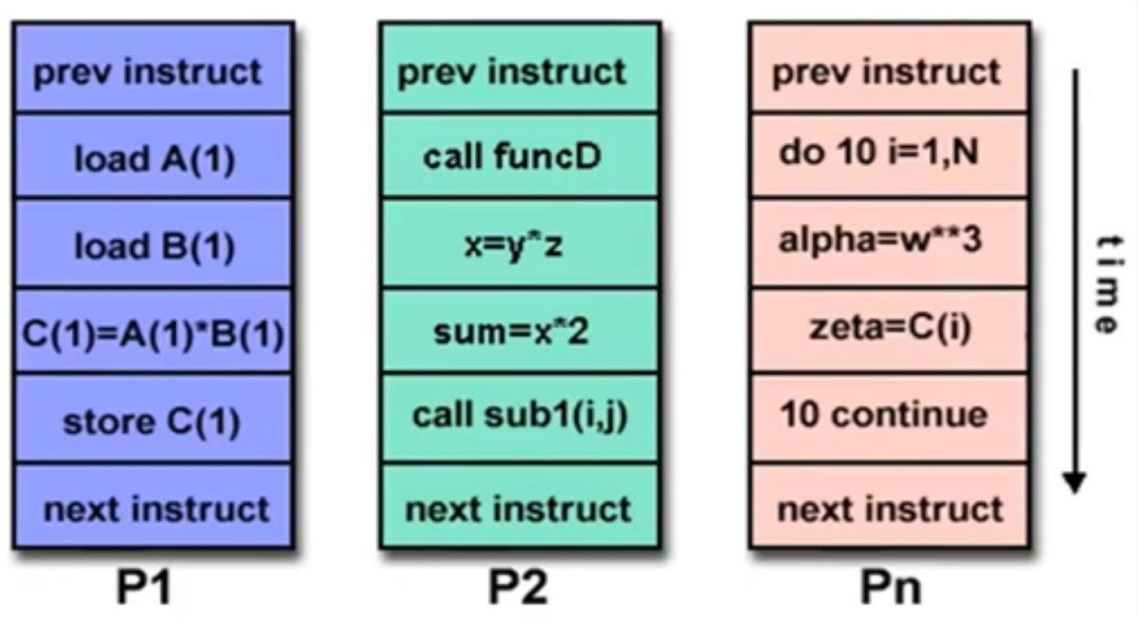
# Flynn's Taxonomy

**MIMD (Multiple Instructions Multiple Data)**

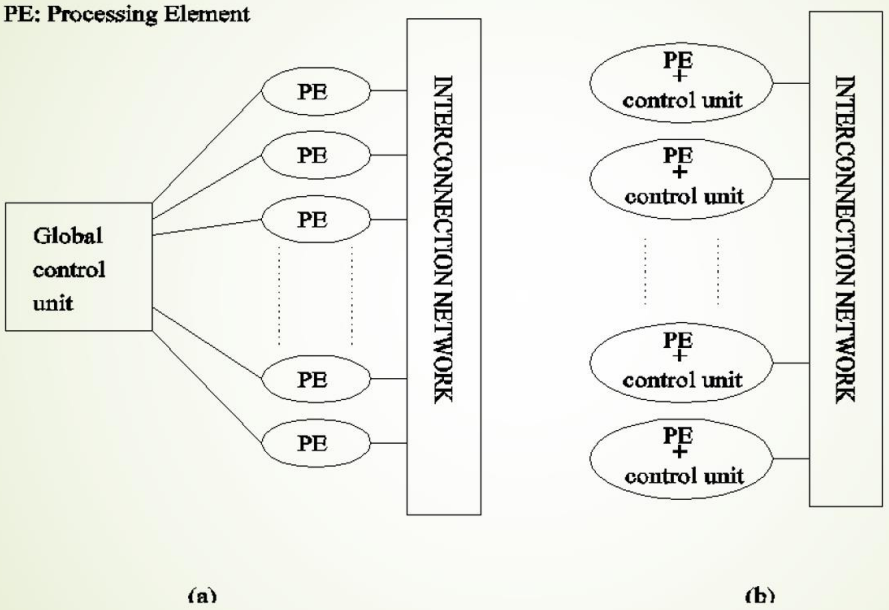
- Multiple instruction streams and multiple data streams
- Different CPUs can simultaneously execute different instruction streams manipulating different data
- Most of the modern parallel architectures fall under this category e.g., **Multiprocessor** and **multicomputer** architectures
- Many MIMD architectures include SIMD executions by default.



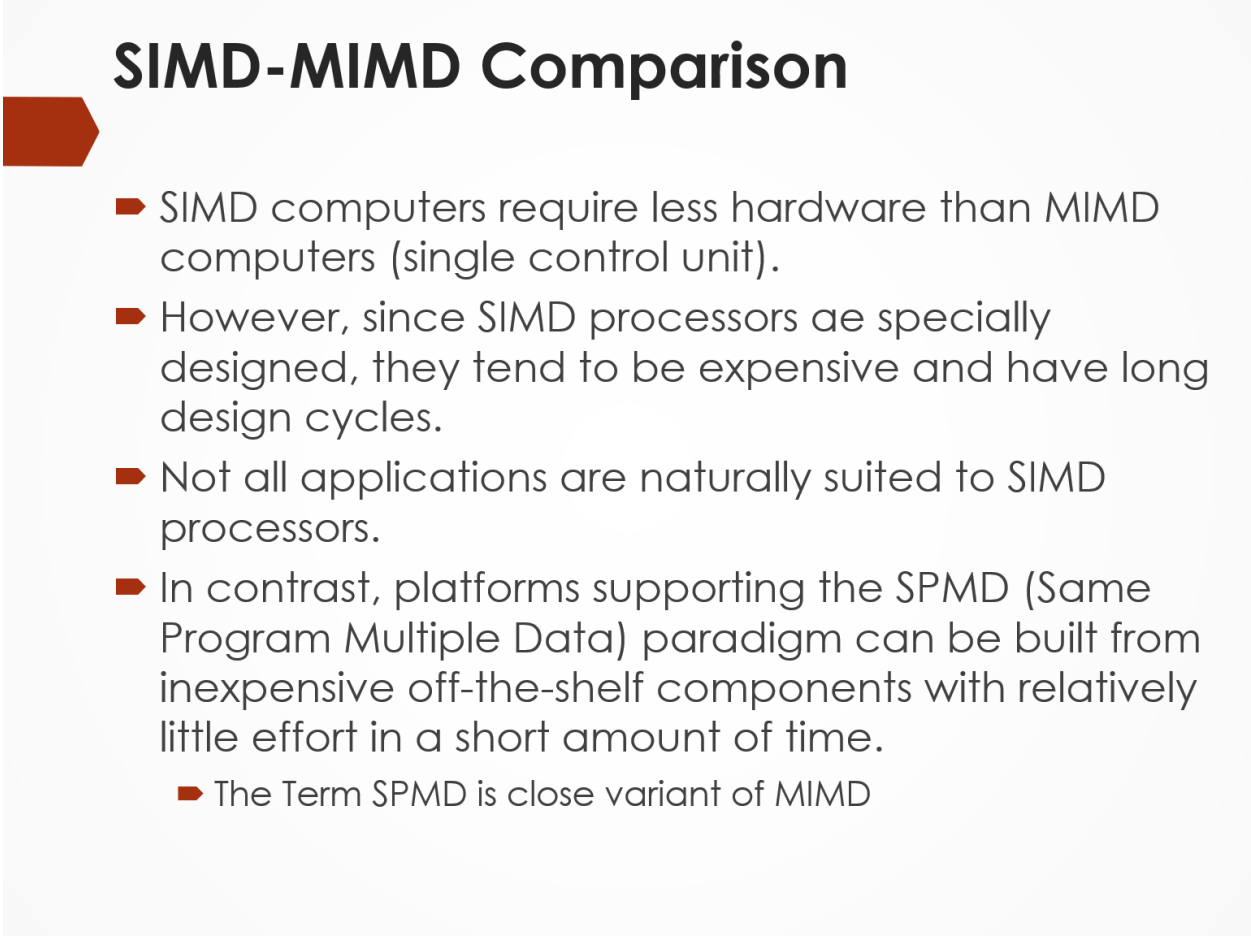
# Example of MIMD:



# Flynn's Taxonomy




A typical SIMD architecture (a) and a typical MIMD architecture (b).




Assigned reading: → SPMD(Running same program on multiple computers e.g., clusters of workstations),

---

# 1. Flynn's Taxonomy Recap

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

- **SISD:** Single instruction, single data → traditional serial computer.
- **SIMD:** Single instruction, multiple data → all cores execute the **same instruction** on **different data**.
- **MISD:** Multiple instructions, single data → rare, e.g., systolic arrays.
- **MIMD:** Multiple instructions, multiple data → multiple processors executing different instructions on different data.

---

# 2. SIMD vs MIMD vs SPMD

## SIMD

- **Single Instruction, Multiple Data**
- **All cores execute the same instruction**, but on **different pieces of data**.
- Works well for **vectorized operations, graphics, image processing, and scientific computing**.
- Less hardware complexity (single control unit), but not all problems fit this model.
- **Examples:** Cray vector machines, GPUs, Intel MMX.

---

## MIMD

- **Multiple Instructions, Multiple Data**
- **Each CPU/core can run a different instruction stream** on different data.
- Very flexible, can run general-purpose parallel programs.
- Most modern parallel computers (multi-core CPUs, clusters) are **MIMD architectures**.
- Many MIMD systems can execute **SIMD-like operations** if needed.

---

## SPMD (Same Program Multiple Data)

- A **programming model**, not strictly a hardware type.
- Multiple processors execute **the same program** independently on **different data**.
- Conceptually, it's similar to **MIMD**, because each processor executes its **own instruction stream**, but in practice, the instruction stream is **logically the same program**.
- SPMD is **a subset of MIMD architectures**.
    - Hardware: MIMD
    - Programming model: SPMD
- Very common in distributed computing and HPC.

**Example:**

- You write a parallel matrix multiplication program. Each CPU runs the same program (loop structure, function calls) on a **different chunk of the matrix**. This is **SPMD**, running on an **MIMD machine**.

---

## Key Distinctions

| Term | Hardware/Software | Instruction Stream | Data Stream | Example |
|---|---|---|---|---|
| SIMD | Hardware | Single | Multiple | GPU, Cray vector machine |
| MIMD | Hardware | Multiple | Multiple | Multi-core CPU, cluster |
| SPMD | Programming model | Each executes same program | Multiple | MPI-based matrix multiplication |

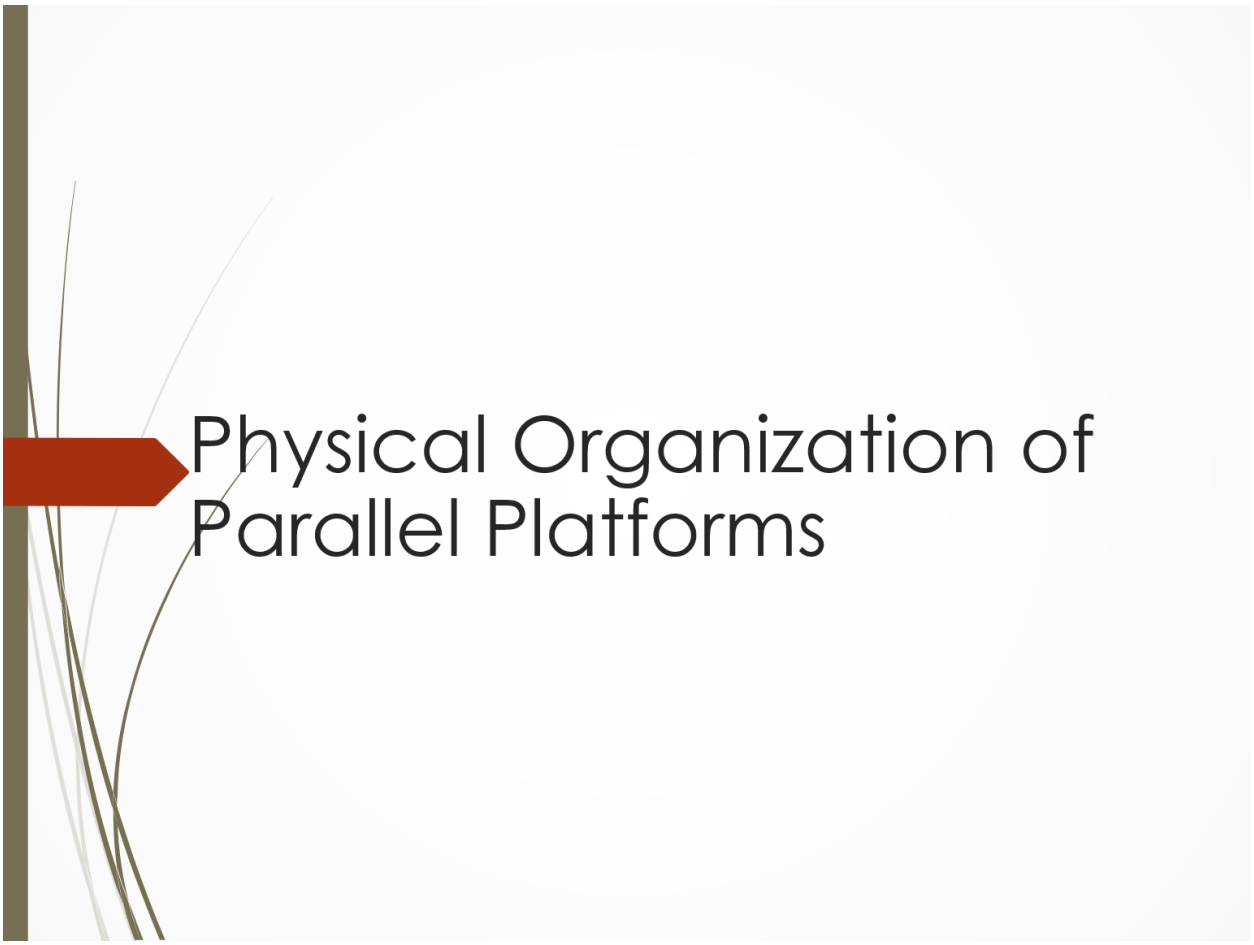
- **SPMD ≈ MIMD in hardware**, but emphasizes that the **program is the same** on all processors, just the data differs.
- It allows building **parallel programs on cheap off-the-shelf MIMD systems** without specialized SIMD hardware.

---

✅ **In short:**

- **SIMD:** Hardware-level single control unit → multiple data.
- **MIMD:** Hardware-level multiple CPUs → multiple instruction streams + data streams.
- **SPMD:** Software-level model → same program on multiple processors; runs on MIMD hardware.

---


Physical Organization of
Parallel Platforms

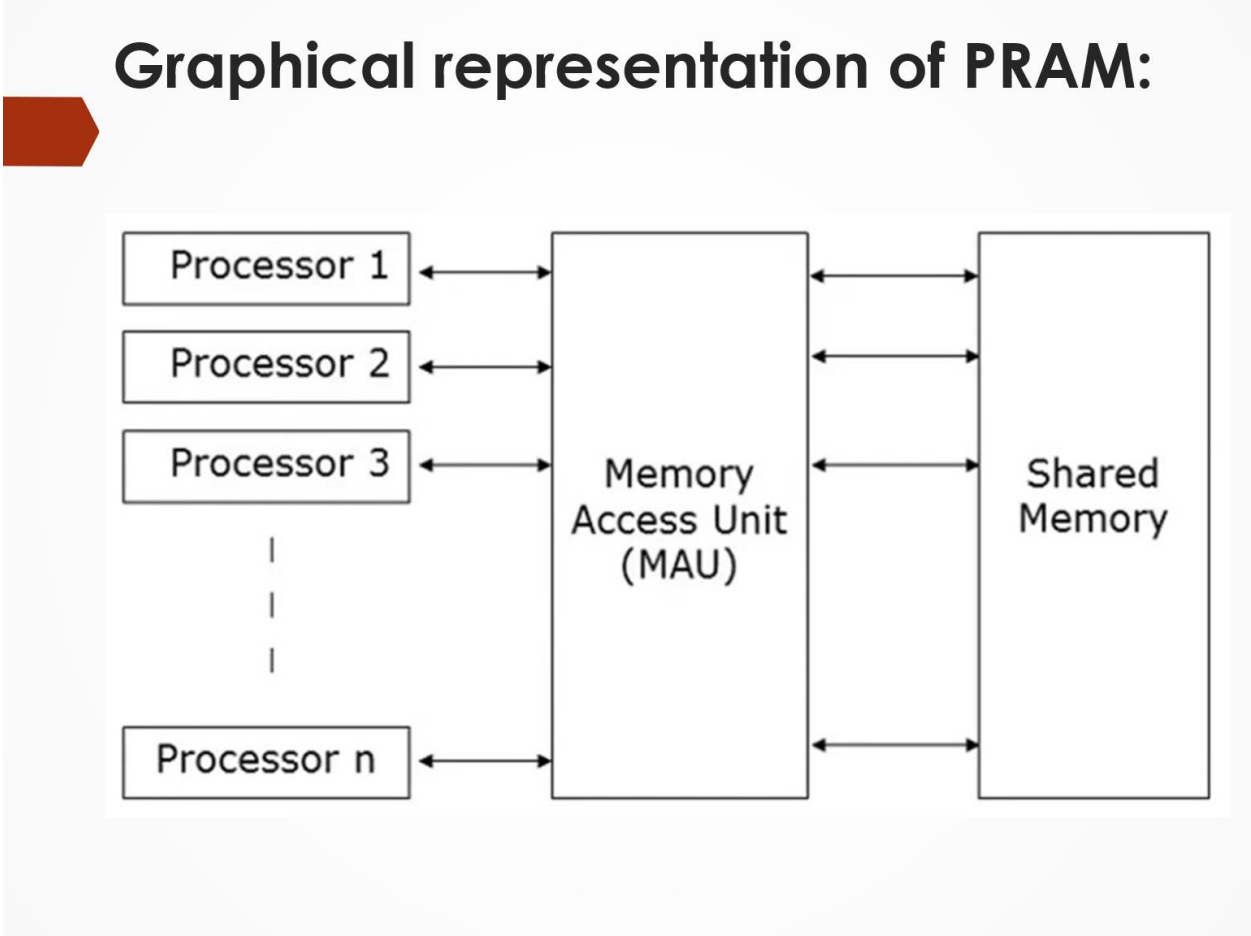
# Architecture of an Ideal Parallel Computer

## Parallel Random Access Machine (PRAM)

- An extension to ideal sequential model: random access machine (RAM)
- PRAMs consist of ***p*** processors
- A global memory
  - Unbounded size
  - Uniformly accessible to all processors with same address space
- Processors share a common clock but may execute different instructions in each cycle.
- Based on simultaneous memory access mechanisms, PRAM can further be classified.

What is the problem here?

Ans: N no. of processors can perform independent operations on N no. of data in a given time, this might lead to simultaneous access of same memory location by different processors




PRAM has a set of similar type of processors

Processors communicate with each other using the shared memory

# Architecture of an Ideal Parallel Computer

## Parallel Random Access Machine (PRAM)

- PRAMs can be divided into four subclasses.

1. Exclusive-read, exclusive-write (EREW) PRAM
   - No two processors can perform read/write operations concurrently
   - Weakest PRAM model, provides minimum memory access concurrency
2. Concurrent-read, exclusive-write (CREW) PRAM
   - All processors can read concurrently but can't write at same time
   - Multiple write accesses to a memory location are serialized
3. Exclusive-read, concurrent-write (ERCW) PRAM
   - No two processors can perform read operations concurrently, but can write
4. Concurrent-read, concurrent-write (CRCW) PRAM
   - Most powerful PRAM model

To solve the simultaneous access of same memory location problem we have these subclasses

Exclusive meaning: excluding (not doing other things)

Isn't concurrent write would be an issue?

# Architecture of an Ideal Parallel Computer

**Parallel Random Access Machine (PRAM)**

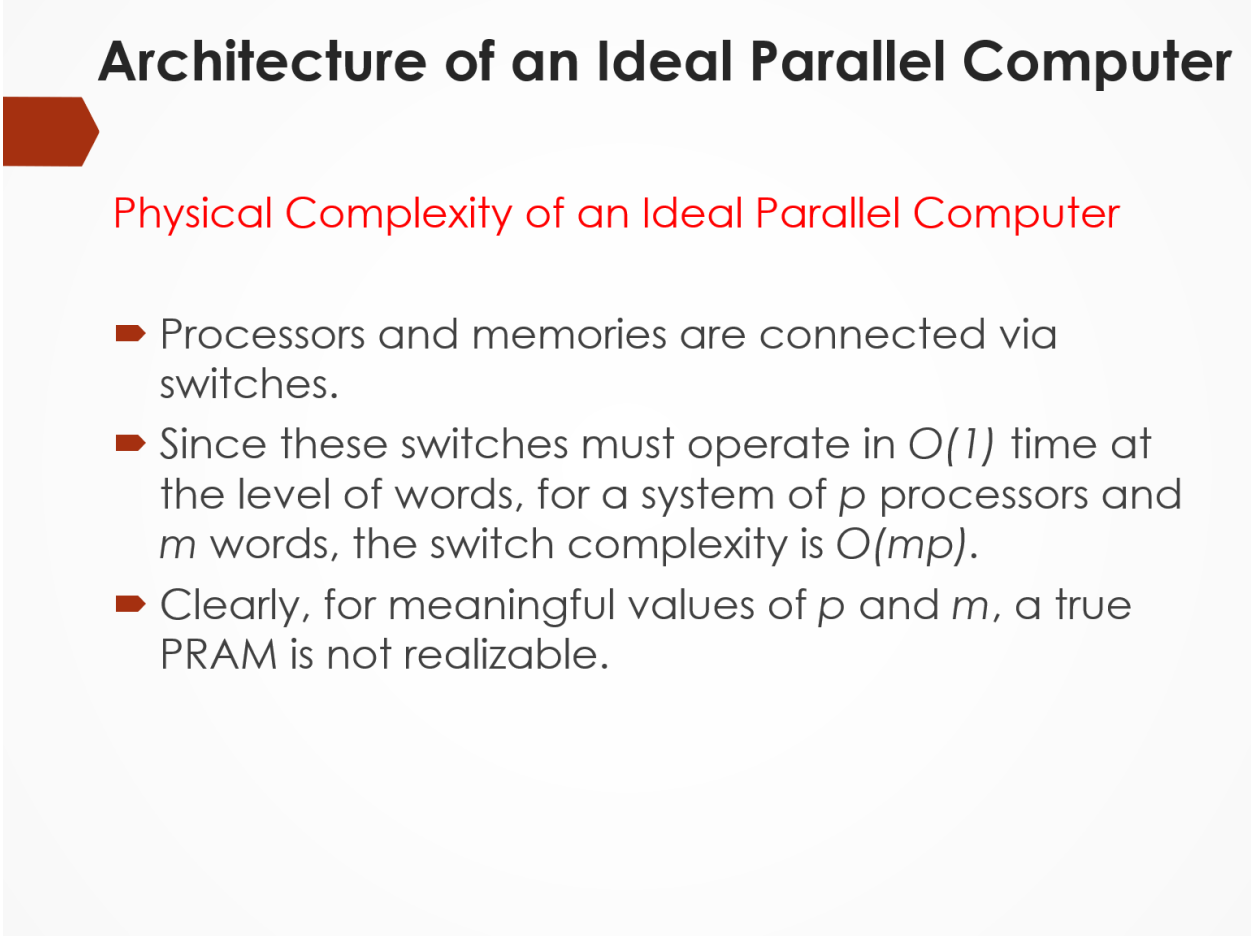
- Concurrent reads do not create any semantic inconsistencies
- But, What about concurrent write?
- Need of an arbitration(mediation) mechanism to resolve concurrent write access

# Architecture of an Ideal Parallel Computer

## Parallel Random Access Machine (PRAM)

- Mostly used arbitration protocols: -
  - **Common:** write only if all values that the processors are attempting to write are identical
  - **Arbitrary:** write the data from a randomly selected processor and ignore the rest.
  - **Priority:** follow a predetermined priority order. Processor with highest priority succeeds and the rest fail.
  - **Sum:** Write the sum of the data items in all the write requests. The sum -based write conflict resolution model can be extended for any of the associative operators, that is defined for data being written .

Max, Multiplication, XOR




these switches determine the memory word being accessed by each processor

Switch is a device that opens or closes access to certain data bank or word

---

# Physical Organization of an Ideal Parallel Computer (PRAM)

1. **Concept of PRAM**
   - PRAM = **Parallel Random Access Machine**.
   - It's like an **idealized computer for learning parallel algorithms**.
   - Components:
     - **p processors** → multiple CPUs working at the same time
     - **Global shared memory** → all processors can access the same memory
     - **Single clock** → processors are synchronized
   - Goal: N processors can operate on N pieces of data simultaneously.

---

2. **The Problem**
    - If multiple processors try to **read or write the same memory location at the same time**, conflicts occur.
    - **Reading the same memory** → usually fine
    - **Writing to the same memory** → needs conflict resolution

---

3. **PRAM Subclasses (Based on Memory Access)**

| PRAM Type | Reads | Writes | Notes |
|---|---|---|---|
| EREW | Exclusive | Exclusive | No two processors read/write same location at same time → simplest model |
| CREW | Concurrent | Exclusive | Multiple reads allowed, but writes must be serialized |
| ERCW | Exclusive | Concurrent | Multiple writes allowed, but reads cannot be simultaneous |
| CRCW | Concurrent | Concurrent | Most powerful, multiple reads and writes allowed → needs arbitration |

---

4. **Concurrent Write Problem**

- **Why it's an issue:** Two processors trying to write different values to the same memory can lead to inconsistent data.
- **Solution:** Use **arbitration (mediation) mechanisms**:
    1. **Common:** Write only if all processors want the same value.
    2. **Arbitrary:** Randomly select one processor's value to write.
    3. **Priority:** Processor with highest priority wins.
    4. **Sum/Associative:** Write the sum (or any associative operation) of all values.

**Example:**

- Memory location = 0
- Processor 1 → write 5
- Processor 2 → write 10
- Using **arbitrary arbitration**, memory might end up = 5 (randomly) or using **sum**, memory = 15

---

5. **Switch in PRAM (Physical Organization)**

- Think of **switch as a gate** that controls which processor can access which memory word.
- **Each processor → switch → memory word**

- Switches open/close paths so multiple processors can access memory **simultaneously without interference**.

**Example (Simple):**

- 2 processors, 4 memory words
- Switch connects processor 1 → memory word 3
- Processor 2 → memory word 2
- Switches allow **both to access memory at the same time**.
- But if both try word 3 → arbitration resolves it (common, priority, sum, etc.)
- **Complexity:**
  - For p processors and m memory words, switch complexity = O(p × m)
  - Meaning: In real life, building a perfect PRAM is **not practical**, but it helps to **design parallel algorithms**.

---

## ✅ Exam-Friendly Version

**PRAM (Parallel Random Access Machine)**:

- Ideal parallel computer with p processors + shared global memory.
- **Problem:** Multiple processors may try to read/write same memory.
- **Memory Access Types:** EREW, CREW, ERCW, CRCW
- **Concurrent Write Solution:** Use arbitration → common, arbitrary, priority, sum.
- **Switch:** A gate that connects processor to memory word, allows simultaneous access; resolves conflicts using arbitration.

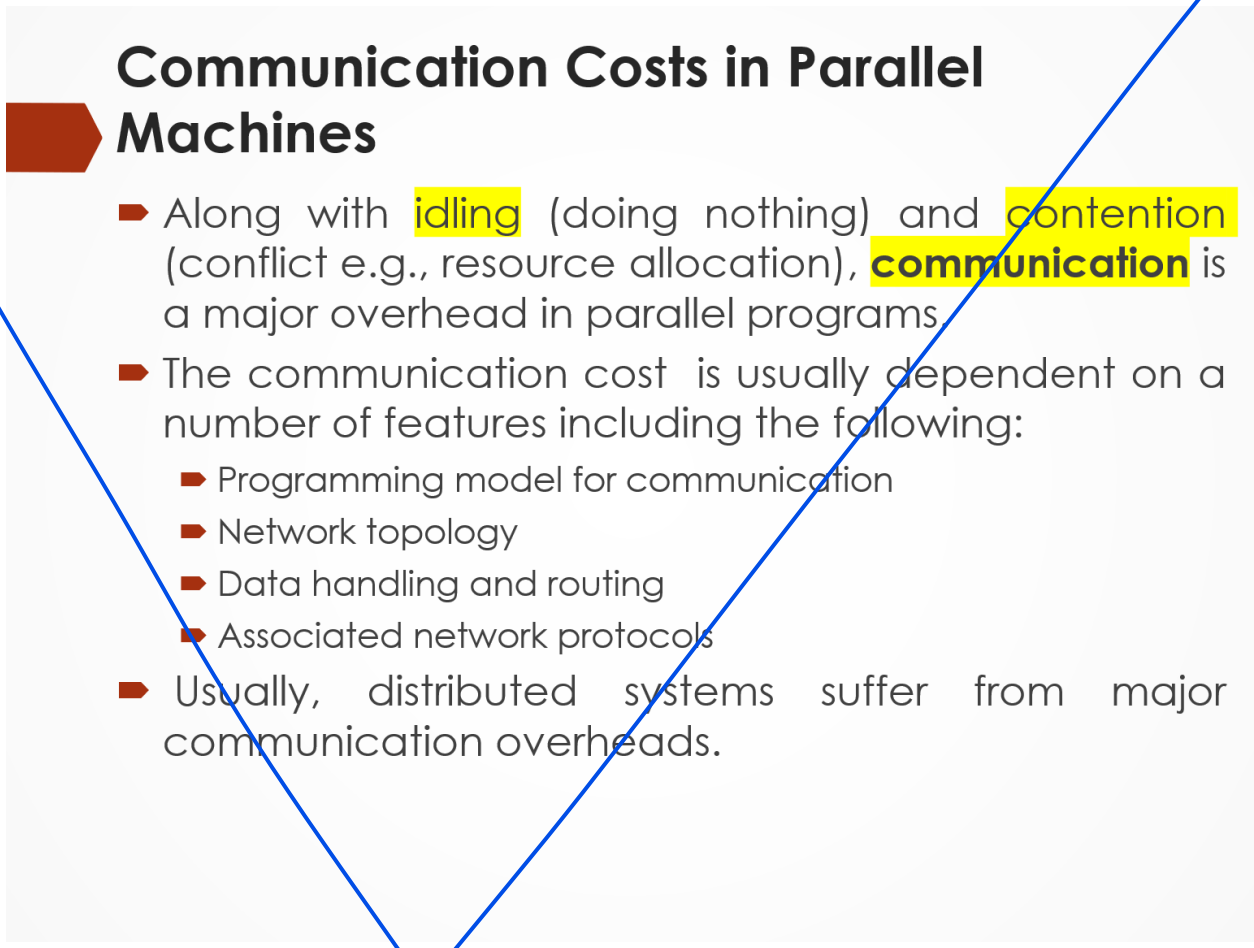
**Example:**

- Processor 1 → memory 5
- Processor 2 → memory 5
- Concurrent write → use "sum" → memory = 10

---

Also do with this lec#6

# Communication Costs in Parallel Machines




Programming model for communication—Required pattern of the communication in the program

Associated protocols: Security assessments UDP,TCP,

# Message Passing Costs in Parallel Computers

- The total time to transfer a message over a network comprises of the following:
  - ***Startup time*** **($t_s$):** Time spent at sending and receiving nodes (preparing the message[adding headers, trailers, and parity information ] , executing the routing algorithm, establishing interface between node and router, etc.).
  - ***Per-hop time*** **($t_h$):** This time is a function of number of hops (steps) and includes factors such as switch latencies, network delays, etc.
    - Also known as **node latency**.
  - ***Per-word transfer time*** **($t_w$):** This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, and buffering overheads, etc.

$t_h$ also accounts for the latency to take decision of choosing next channel to which this message shall be forwarded

If channel bandwidth is r words/s then each word take $t_w = 1/r$ to traverse the link.

# Message Passing Costs in Parallel Computers

## Store-and-Forward Routing

- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size ***m*** words to traverse ***l*** communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, $t_h$ is small and the above expression can be approximated by

$$t_{comm} = t_s + mlt_w.$$

*Cost of header transfer at each hop (step) $t_h$.*

*Ts is startup time*

*Mtw id cost of transferring m words over this link*

# Message Passing Costs in Parallel Computers

## Packet Routing

- Store-and-forward makes poor use of communication resources.
- Packet routing breaks messages into packets and pipelines them through the network.
- Since packets may take different paths, each packet must carry routing information, error checking, sequencing, and other related header information.
- The total communication time for packet routing is approximated by: $t_{comm} = t_s + t_h l + t_w m$.
- Here factor $t_w$ also accounts for overheads in packet headers.



Error checking---parity information

Sequencing---packet order number

Related headers: layers headers, addressing headers

# Message Passing Costs in Parallel Computers

## Cut-Through Routing

- Takes the concept of packet routing to an extreme by further dividing messages into basic units called **flits** or flow control digits.
- Since flits are typically small, the header information must be minimized.
- This is done by forcing all flits to take the same path, in sequence.
- A tracer message first programs all intermediate routers. All flits then take the same route.
- Error checks are performed on the entire message, as opposed to flits.
- No sequence numbers are needed.

Sequencing information is not needed as all the packets are following same path which ensures in-order delivery

# Message Passing Costs in Parallel Computers

## Cut-Through Routing

- The total communication time for cut-through routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

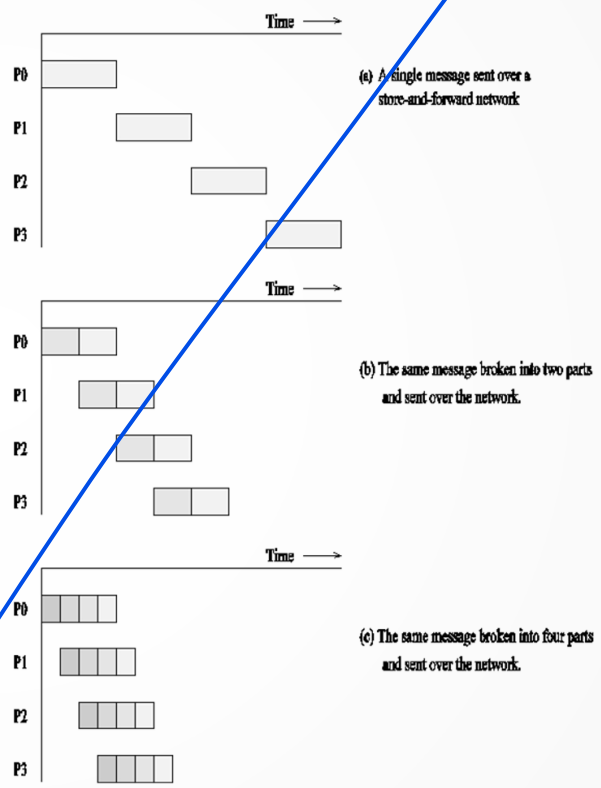
- This is identical to packet routing, however, $t_w$ is typically much smaller.

Header of the message takes l* $t_h$ to reach the destination and entire message arrives in time **m**$t_w$ after the message header

# Message Passing Costs in Parallel Computers

(a) through a store-and-forward communication network;

b) and (c) extending the concept to cut-through routing.




Shaded regions here represent the time where message is in transit (travel)

The startup time associated with this message transfer is assumed to be zero

# Message Passing Costs in Parallel Computers

## Simplified Cost Model for Communicating Messages

- The cost of communicating a message between two nodes $l$ hops away using cut-through routing is given by

$$t_{comm} = t_s + lt_h + t_w m.$$

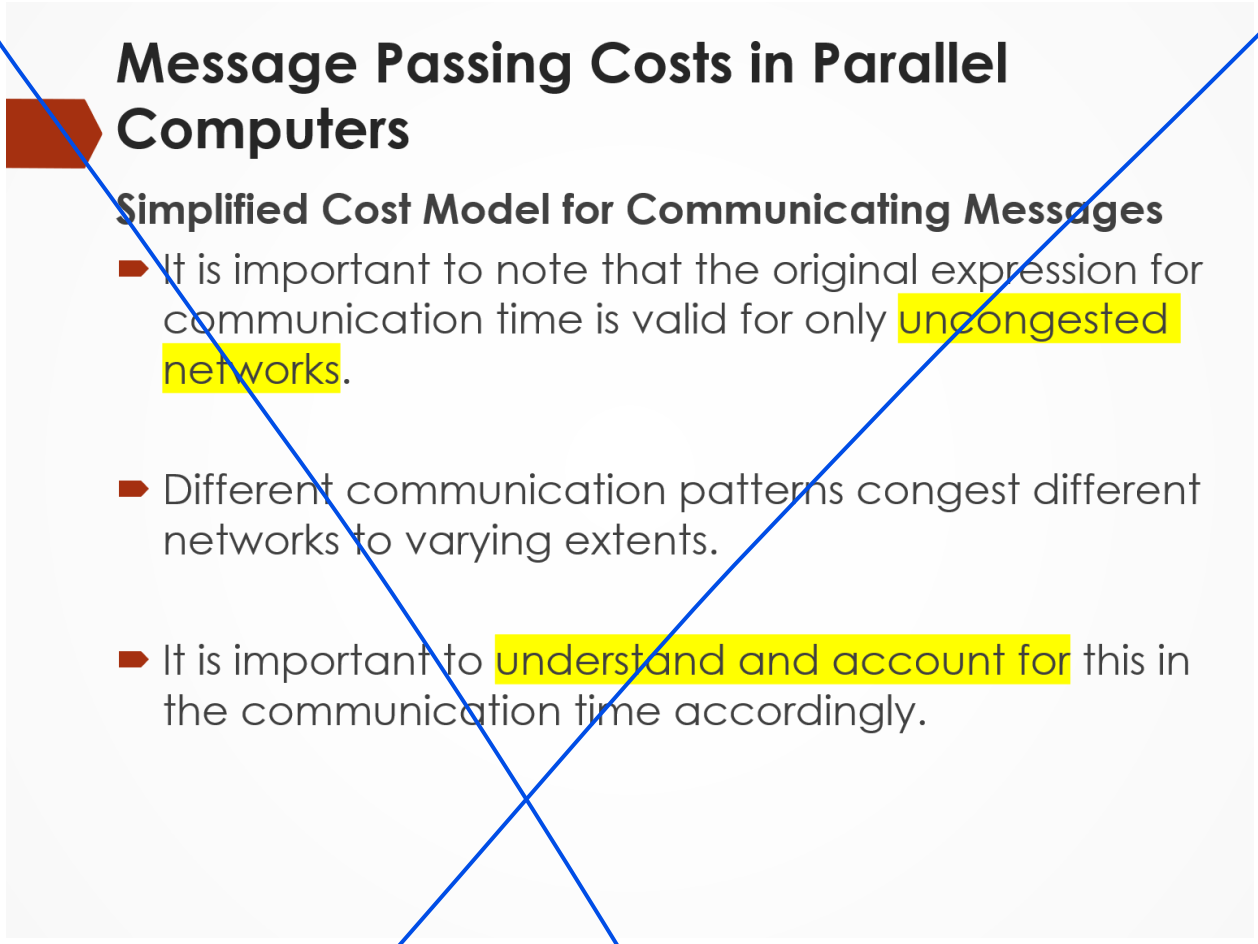
- In this expression, $t_h$ is typically smaller than $t_s$ and $t_w$. For this reason, the second term in the RHS does not show, particularly, when $m$ is large.

- For these reasons, we can approximate the cost of message transfer by

$$t_{comm} = t_s + t_w m.$$



For communication using flits, start-up time dominates the node latencies.




---

## 1. Switches in Parallel Computers

- **Switch:** A device that connects **processors to memory words or banks**.
- **Function:** Opens or closes the path for a processor to access a particular memory location.
- **Problem:** If multiple processors want the **same memory word** at the same time → **contention**.

**Example:**

- Processor 1 wants memory word 5
- Processor 2 also wants memory word 5
- Switch mediates access → either one writes first, or an arbitration policy is used.

---

## 2. Communication in Parallel Programs

Communication = **sending data between processors**.

### Major overheads in parallel systems:

1. **Idling:** Processor waits for data.
2. **Contention:** Multiple processors want the same resource.
3. **Communication:** Sending messages between processors (network delay).

**Factors affecting communication cost:**

- Programming model (how communication is coded)
- Network topology (how processors are connected)
- Data handling & routing
- Network protocols (TCP, UDP, etc.)

---

## 3. Programming Models for Communication

- **Message Passing:** Processors explicitly send/receive messages.
- **Communication cost** depends on **message size, network hops, routing method**.

---

## 4. Routing Techniques

### a) Store-and-Forward Routing

- Message travels **hop by hop**.
- Each intermediate node **receives the full message**, then forwards it.
- Communication cost:

$$
T = l \cdot t_h + m \cdot t_w
$$

Where:

- $l$ = number of hops
- $t_h$ = time for header to traverse each hop
- $t_w$ = time per word to transfer
- $m$ = number of words in message

**Problem:** Slow for large networks because nodes must fully receive messages before forwarding.

### b) Packet Routing

- Message divided into **packets**, pipelined through network.
- Packets may take different paths.
- Each packet has: routing info, error checking, sequence number, headers.
- Communication cost ≈ **similar to store-and-forward**, but pipelining reduces delay.

**Example:** Internet TCP/IP packets

---

### c) Cut-Through Routing

- Message divided into **flits** (small units).
- Header programs the network first → all flits follow same path.
- **No sequence numbers**, error checking at end.
- Faster than store-and-forward.

**Approximate cost:**

[
T \approx t_s + l \cdot t_h + m \cdot t_w
]

Where:

- (t_s) = startup time (prepare message, headers, trailers, parity)
- (t_h) = per-hop latency
- (t_w) = per-word transfer time

**Example:** Supercomputer interconnects, InfiniBand networks

---

# 5. Communication Cost Components (Simplified)

| Term | Meaning | Example |
|---|---|---|
| (t_s) (Startup) | Time at sending/receiving node to prepare message | Adding headers, setting up link |
| (t_h) (Per-hop) | Time for message to traverse one network hop | Switch delay, router decision latency |
| (t_w) (Per-word) | Time per word/message length | Bandwidth delay, buffering overheads |

**Notes:**

- For large messages, startup dominates per-hop delays.
- Uncongested networks → formula valid; congestion increases communication time.

---

# 6. Exam-Friendly Summary

- **Switch:** Opens/closes access for processor → memory. Mediates conflicts.
- **Parallel communication overheads:** Idling, contention, message transfer.
- **Routing types:**
    1. Store-and-forward → full message at each hop
    2. Packet routing → divide message into packets
    3. Cut-through → divide message into flits, follow same path
- **Communication cost:**

$$
T \approx t_s + l \cdot t_h + m \cdot t_w
$$

- **Components:**
    - $t_s$ → startup time (headers, node setup)
    - $t_h$ → per-hop latency (switch/router delay)
    - $t_w$ → per-word transfer time (bandwidth-dependent)

**Example:**

- Supercomputer sending 1000 words over 4 hops:
    - Startup = 2 ms, per-hop = 0.1 ms, per-word = 0.001 ms
    - Total ≈ $2 + (4 \times 0.1) + (1000 \times 0.001) = 2 + 0.4 + 1 = 3.4$ ms

---

---