Q1. Why is it difficult to construct a true shared-memory computer? What is the minimum number of switches for connecting p processors to shared memory with b words (where each word can be accessed independently)?

Let's tackle your question **carefully and step by step**.

Q1(a): Why is it difficult to construct a true shared-memory computer?

A **true shared-memory computer** is a system where multiple processors can **directly access the same memory locations** as if there were a single memory. While conceptually simple, there are several practical difficulties:

1. **Memory Access Conflicts**
    - If two or more processors try to read/write the same memory location simultaneously, we need **arbitration** to prevent conflicts.
    - High contention can severely reduce performance.
2. **Scalability Issues**
    - As the number of processors increases, the **bus or interconnection network** connecting processors to memory becomes a bottleneck.
    - The memory access time can increase, making the system inefficient for many processors.
3. **Synchronization Overhead**
    - To ensure correct execution, processors need locks or semaphores. Managing these across many processors is complex and can lead to **deadlocks or delays**.
4. **Hardware Complexity**
    - Each processor needs mechanisms to detect **data hazards**, manage **caches coherently**, and handle concurrent memory requests.
    - Designing such interconnection networks (e.g., crossbars, multistage switches) is **expensive and complex**.

✅ In short: The difficulty arises from **contention, synchronization, and the need for scalable hardware** to allow simultaneous access.

Q1(b): Minimum number of switches to connect `p` processors to `b` memory words

We are essentially looking at **memory interconnection networks**.

1. **Direct connection approach**:
    - If each processor can independently access any memory word, we need some network that **connects `p` processors to `b` memory modules**.
2. **Crossbar switch**:

  - The **simplest theoretical solution** is a **full crossbar**, where each processor has a dedicated path to each memory word.
  - Number of switches required:

$$\text{Number of switches} = p \times b$$

- Each processor has `b` switches (one to each memory word).

3. **Why this is minimal:**
   - Any network allowing **simultaneous access to all `b` words by all `p` processors** requires at least one switch per processor–memory word pair.
   - Using fewer switches would **force contention**, meaning some processor cannot access some memory independently.

✓ Answer (Summary)

(a) Difficulties in constructing true shared-memory computer:

- Memory access conflicts among processors
- Scalability limitations as processors increase
- Synchronization overhead for correct execution
- Complex and expensive hardware design

(b) Minimum number of switches for `p` processors and `b` memory words:

$$p \times b$$

- This is the number of switches in a **full crossbar network**, allowing independent access.

If you want, I can also **draw a small diagram for `p=3` and `b=4`** to visualize why `p × b` switches are needed. It makes this concept much easier to understand.

Q2. Consider set of task in parallel computing system. Each task has a known execution time, and they can be executed concurrently. Calculate the average degree of concurrency for the given set of tasks.

Step 1: Understand the problem

- You have **n tasks**, each with execution time T_i.
- Tasks can run concurrently (in parallel).
- **Average degree of concurrency** tells you, on average, how many tasks are running simultaneously during the execution.

Definition:

$$C_{\text{avg}} = \frac{\text{Total work}}{\text{Total execution time on parallel system}}$$

Where:

- **Total work** $W = \sum_{i=1}^{n} T_i$
- **Total execution time** T_{parallel} = overall time to complete all tasks when executed in parallel.

Step 2: Formula

$$\text{Average degree of concurrency} = \frac{\text{Sum of execution times of all tasks}}{\text{Time span during which tasks are running}}$$

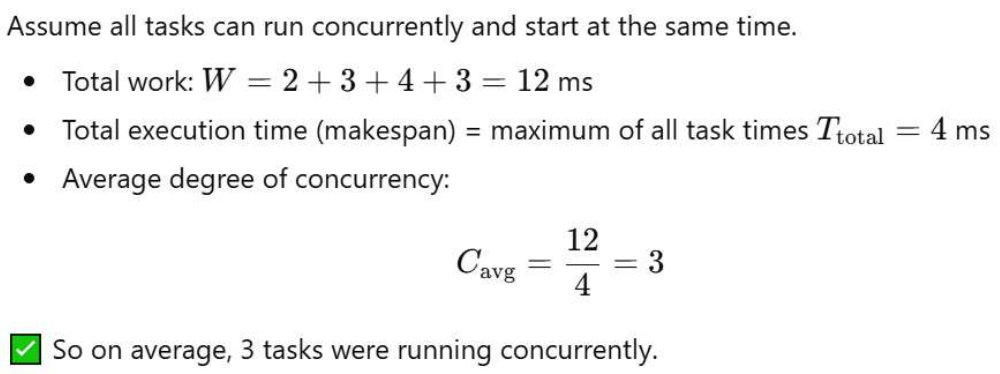
$$C_{\text{avg}} = \frac{\sum_{i=1}^{n} T_i}{T_{\text{total}}}$$

Where T_{total} is the time from the start of the first task to the completion of the last task (the **makespan**).

Step 3: Example

Suppose we have 4 tasks with execution times:

Task	Execution Time (ms)
T1	2
T2	3
T3	4
T4	3

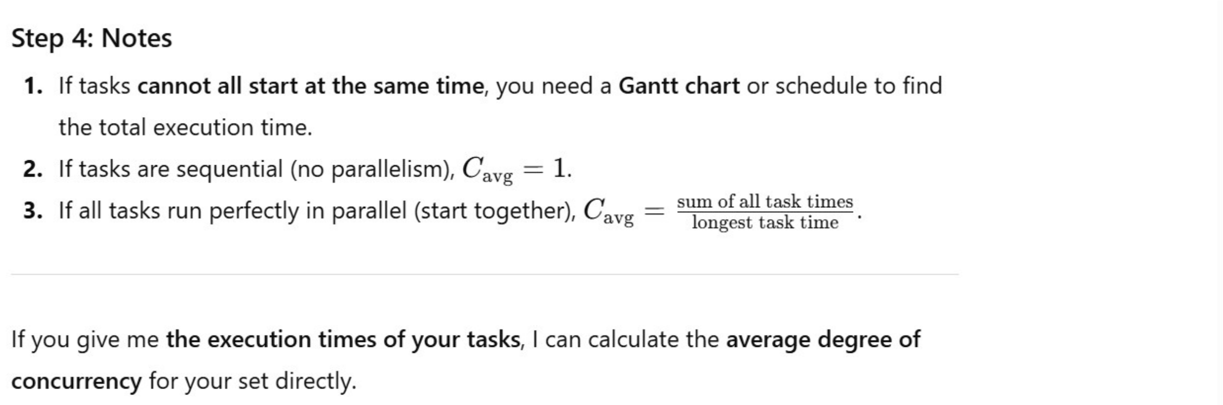


Assume all tasks can run concurrently and start at the same time.

- Total work: $W = 2 + 3 + 4 + 3 = 12$ ms
- Total execution time (makespan) = maximum of all task times $T_{\text{total}} = 4$ ms
- Average degree of concurrency:

$$C_{\text{avg}} = \frac{12}{4} = 3$$

✅ So on average, 3 tasks were running concurrently.



Step 4: Notes

1. If tasks **cannot all start at the same time**, you need a **Gantt chart** or schedule to find the total execution time.
2. If tasks are sequential (no parallelism), $C_{\text{avg}} = 1$.
3. If all tasks run perfectly in parallel (start together), $C_{\text{avg}} = \frac{\text{sum of all task times}}{\text{longest task time}}$.

If you give me **the execution times of your tasks**, I can calculate the **average degree of concurrency** for your set directly.

✅ So, this question is **directly related to analyzing task parallelism and concurrency in parallel computing systems**, which is part of **CLO-3: Analyze complex problems with shared memory programming (OpenMP) and task parallelism**.

Q3. Formulate an equestion to calculate the communication cost in a distributed system. Consider factors such as message size, bandwidth and latency.

To calculate the **communication cost** in a distributed system, you need to consider the **time it takes to send a message** from one node to another. The key factors are:

- **Message size** (M): amount of data to send (in bytes or bits)
- **Bandwidth** (B): data transfer rate of the network (bytes/sec or bits/sec)
- **Latency** (L): fixed delay in the network (propagation delay, startup time, in seconds)

Step 1: Understand the components

1. **Transmission time**: Time to send the message over the network

$$T_{\text{transmission}} = \frac{M}{B}$$

2. **Propagation time (Latency)**: Time for the first bit to reach the destination

$$T_{\text{latency}} = L$$

Step 2: Communication cost formula

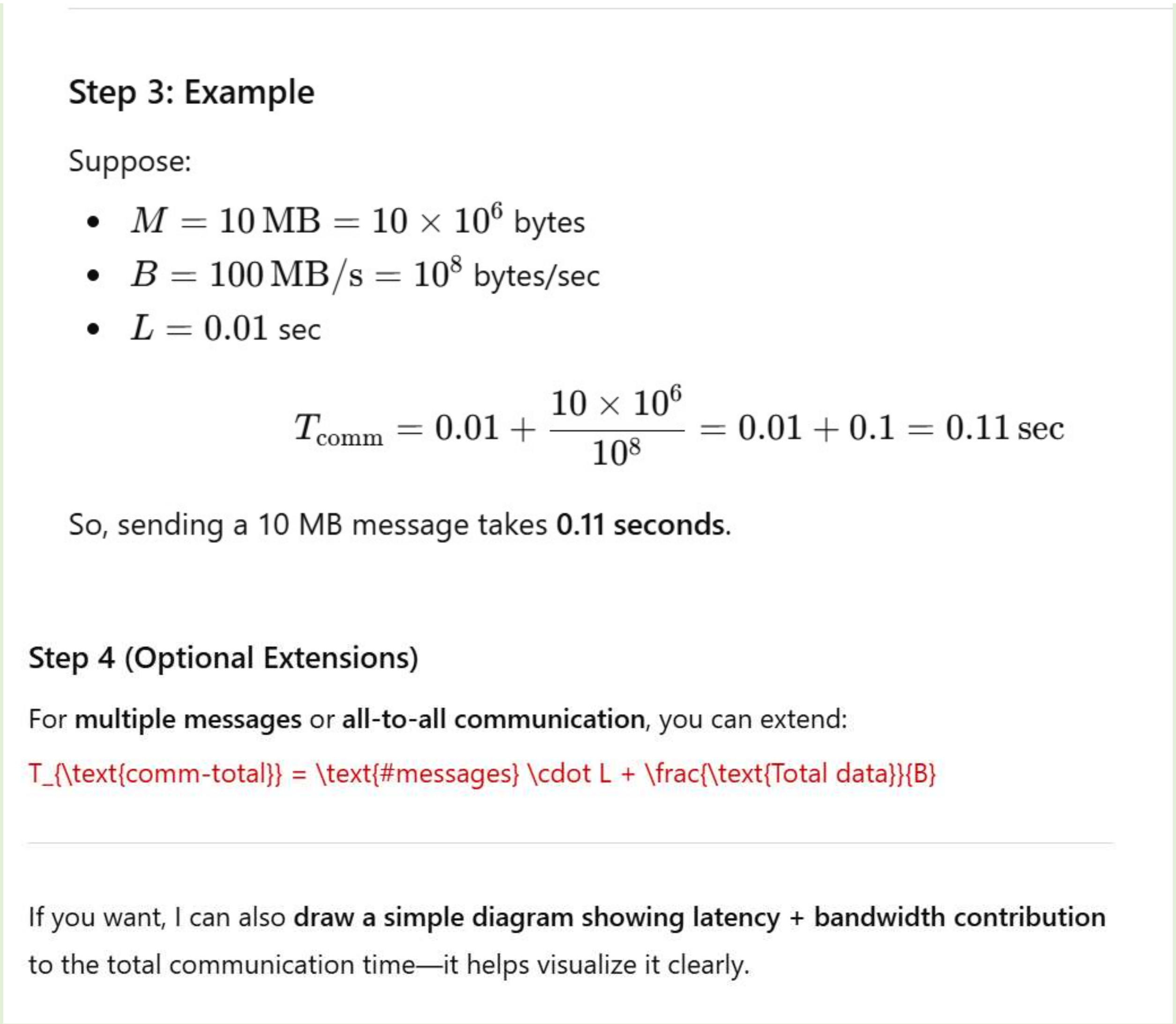
The **total communication cost** T_{comm} is the sum of **latency** and **transmission time**:

$$\boxed{T_{\text{comm}} = L + \frac{M}{B}}$$

Where:

- T_{comm} = communication cost (seconds)
- L = network latency (seconds)
- M = message size (bytes)
- B = network bandwidth (bytes/sec)

> If messages are **very small**, latency dominates.
> If messages are **very large**, transmission time dominates.



Step 3: Example

Suppose:

- $M = 10\,\mathrm{MB} = 10 \times 10^6$ bytes
- $B = 100\,\mathrm{MB/s} = 10^8$ bytes/sec
- $L = 0.01$ sec

$$T_{\text{comm}} = 0.01 + \frac{10 \times 10^6}{10^8} = 0.01 + 0.1 = 0.11\,\text{sec}$$

So, sending a 10 MB message takes **0.11 seconds.**

Step 4 (Optional Extensions)

For **multiple messages** or **all-to-all communication**, you can extend:

`T_{\text{comm-total}} = \text{#messages} \cdot L + \frac{\text{Total data}}{B}`

If you want, I can also **draw a simple diagram showing latency + bandwidth contribution** to the total communication time—it helps visualize it clearly.

- **Mam-extra: Communication Cost in Parallel Machine (lec #6)**

- Related sections:

  - Asynchronous/synchronous computation/communication (lec #3)
  - Basic Communication Operations (lec #14)

Q4. What are criteria that are used to evaluate the cost and performance of static interconnection networks?

In **parallel and distributed systems**, **static interconnection networks** are fixed networks that connect processors and memory modules (like buses, meshes, hypercubes, trees, etc.). To evaluate their **cost and performance**, several criteria are commonly used:

---

1. Network Diameter

- **Definition:** The maximum number of hops a message must traverse between any two nodes in the network.
- **Impact:** Smaller diameters → lower latency → better performance.
- **Example:** A fully connected network has diameter 1; a linear array has diameter (p-1) for (p) nodes.

---

2. Bisection Width (Bisection Bandwidth)

- **Definition:** The minimum number of links that must be cut to partition the network into two equal halves.
- **Impact:** Larger bisection width → higher potential bandwidth → better communication performance for large-scale parallel tasks.

---

3. Network Cost

- **Definition:** Typically measured as the total number of links (or switches) required to build the network.
- **Impact:** Fewer links → cheaper hardware cost; more links → higher cost but often better performance.

---

4. Node Degree

- **Definition:** Number of direct links connected to a node.
- **Impact:** Lower node degree → cheaper and simpler nodes; higher node degree → better connectivity and lower communication latency.

---

5. Average Distance (Average Hop Count)

- **Definition:** Average number of hops a message travels between all pairs of nodes.
- **Impact:** Lower average distance → lower average communication latency.

---

6. Scalability

- **Definition:** How easily the network can grow when more nodes are added.
- **Impact:** Networks with poor scalability become inefficient or expensive as the system grows.

---

7. Fault Tolerance / Reliability

- **Definition:** The ability of the network to continue functioning if some nodes or links fail.
- **Impact:** Networks with redundant paths can tolerate failures better.

---

8. Congestion / Throughput

- **Definition:** Measures how well the network handles simultaneous communication requests.
- **Impact:** High throughput → better performance for parallel tasks; congestion can cause delays.

Summary Table:

Criterion	Importance
Network Diameter	Latency between farthest nodes
Bisection Width	Maximum simultaneous communication
Network Cost	Hardware expense
Node Degree	Complexity & cost per node
Average Distance	Average message delay
Scalability	Ease of expansion
Fault Tolerance	Reliability under failures
Congestion / Throughput	Efficiency under heavy traffic

These criteria help designers **balance cost vs. performance** when choosing or designing static interconnection networks.

“Interconnection Networks in Parallel and Distributed Systems” or more specifically:

- **Clg-5: Heterogeneity, Interconnection Topologies (lec #8)**
- Related topics:
    - Network performance analysis (diameter, bisection width, throughput)
    - Scalability and cost evaluation of network designs

Q5. Define cloud computing and explain its essential characteristic. Also explain the primary security challenges in cloud computing.

Definition of Cloud Computing

Cloud computing is a model for delivering computing services—such as servers, storage, databases, networking, software, and analytics—over the internet ("the cloud"). It allows users to access and use resources on-demand without owning or managing the underlying infrastructure.

Formal definition (NIST):

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Essential Characteristics of Cloud Computing

1. **On-Demand Self-Service:**
    - Users can provision computing resources automatically as needed without human interaction from the service provider.
2. **Broad Network Access:**
    - Services are available over the network and can be accessed via standard devices like laptops, smartphones, or tablets.
3. **Resource Pooling (Multi-Tenancy):**
    - Cloud providers serve multiple customers using shared physical and virtual resources, dynamically allocating according to demand.
4. **Rapid Elasticity:**
    - Resources can be scaled up or down quickly to handle workload changes, giving the appearance of unlimited resources.
5. **Measured Service (Pay-per-Use):**
    - Resource usage is monitored, controlled, and reported, allowing users to pay only for what they consume.

Primary Security Challenges in Cloud Computing

1. **Data Breaches and Loss:**
    - Unauthorized access or accidental deletion of sensitive data stored in the cloud.
2. **Insider Threats:**

   - Malicious or careless insiders at the cloud provider or client organization can compromise data security.
3. **Data Privacy and Compliance:**
   - Ensuring that stored and processed data complies with legal and regulatory requirements (e.g., GDPR).
4. **Account Hijacking:**
   - Attackers stealing user credentials to access cloud services, manipulate data, or launch attacks.
5. **Insecure APIs and Interfaces:**
   - Vulnerabilities in cloud APIs or management interfaces can be exploited to gain unauthorized access.
6. **Denial of Service (DoS) Attacks:**
   - Cloud services may be disrupted, affecting availability and performance.
7. **Shared Technology Vulnerabilities:**
   - Multi-tenant environments may expose vulnerabilities in hypervisors, containers, or virtual networks.
8. **Data Location and Transfer Risks:**
   - Data may traverse or reside in different geographical locations, potentially exposing it to legal or security issues.

Summary Table:

Aspect	Details
Definition	On-demand delivery of computing resources via the internet
Essential Characteristics	On-demand self-service, broad network access, resource pooling, rapid elasticity, measured service
Primary Security Challenges	Data breaches, insider threats, privacy/compliance, account hijacking, insecure APIs, DoS attacks, shared tech vulnerabilities

---

If you want, I can also **draw a simple diagram showing cloud characteristics and threats** for easier understanding and exam use.

- **Course Description:** “Cloud and Distributed Tools” (like AWS, OpenStack, Hadoop, Globus)

- **Clg-15: Tools** (lec #22–#26)

- Concepts include **cloud definition, characteristics, service models (IaaS, PaaS, SaaS), and security challenges**.

Q6. What is difference between virtualization and containerization in cloud computing?

---

1. Virtualization

- **Definition:** Virtualization is the process of creating virtual instances of hardware (virtual machines) on a single physical server using a **hypervisor**. Each VM includes its own **OS, libraries, and applications**.
- **Key Features:**
  - Each VM is **isolated** from others.
  - VMs are **heavier** because they include a full OS.
  - Startup time is **slower** compared to containers.
- **Example:** VMware, VirtualBox, Hyper-V.

---

2. Containerization

- **Definition:** Containerization is a lightweight method of packaging an application with all its dependencies (libraries, binaries, configurations) but **sharing the host OS kernel**.
- **Key Features:**
  - Containers are **more lightweight** than VMs.
  - Faster startup time.
  - Better resource efficiency because multiple containers can run on the same OS.
  - Provides **process-level isolation**, not full OS isolation.
- **Example:** Docker, Kubernetes, LXC.

Comparison Table

Feature	Virtualization (VM)	Containerization
OS Required	Full OS per VM	Shares host OS kernel
Resource Usage	Higher (heavyweight)	Lower (lightweight)
Startup Time	Slower (minutes)	Faster (seconds)
Isolation Level	Hardware-level / OS-level	Process-level
Portability	Moderate	High
Examples	VMware, VirtualBox, Hyper-V	Docker, Kubernetes, LXC

Summary:

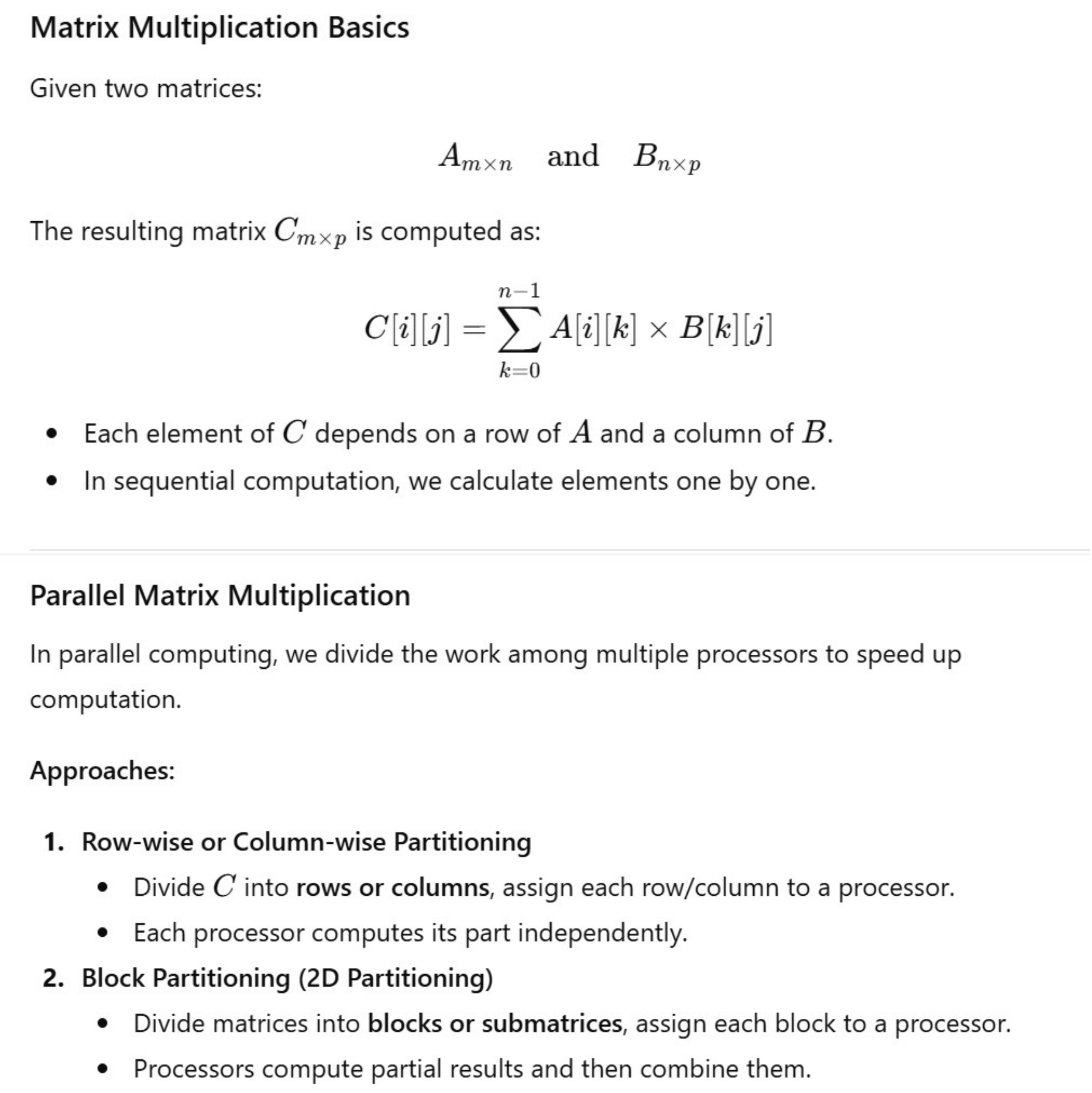
- **Virtualization** emulates hardware and runs multiple OS instances on one physical machine.
- **Containerization** packages applications with dependencies while sharing the host OS, making it **lighter, faster, and more portable**.

If you want, I can also **draw a simple diagram showing VMs vs Containers** for visual clarity—it's very useful for exams.

- **Clg-15: Tools / Cloud and Distributed Tools** (lec #22–#26)

- Concepts in **cloud architecture**: virtualization (VMs) vs containerization (Docker, Kubernetes) for resource management and deployment efficiency.

Q7. How does matrix multiplication work in parallel computing? Explain with an example.



Matrix Multiplication Basics

Given two matrices:

$$A_{m \times n} \quad \text{and} \quad B_{n \times p}$$

The resulting matrix $C_{m \times p}$ is computed as:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]$$

- Each element of C depends on a row of A and a column of B.
- In sequential computation, we calculate elements one by one.

Parallel Matrix Multiplication

In parallel computing, we divide the work among multiple processors to speed up computation.

Approaches:

1. **Row-wise or Column-wise Partitioning**
   - Divide C into **rows or columns**, assign each row/column to a processor.
   - Each processor computes its part independently.
2. **Block Partitioning (2D Partitioning)**
   - Divide matrices into **blocks or submatrices**, assign each block to a processor.
   - Processors compute partial results and then combine them.

Example: Parallel Row-wise Multiplication

Suppose we have:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

- **Resulting Matrix** $C = A \times B$:

$$C[0][0] = 1 * 5 + 2 * 7 = 19$$

$$C[0][1] = 1 * 6 + 2 * 8 = 22$$

$$C[1][0] = 3 * 5 + 4 * 7 = 43$$

$$C[1][1] = 3 * 6 + 4 * 8 = 50$$

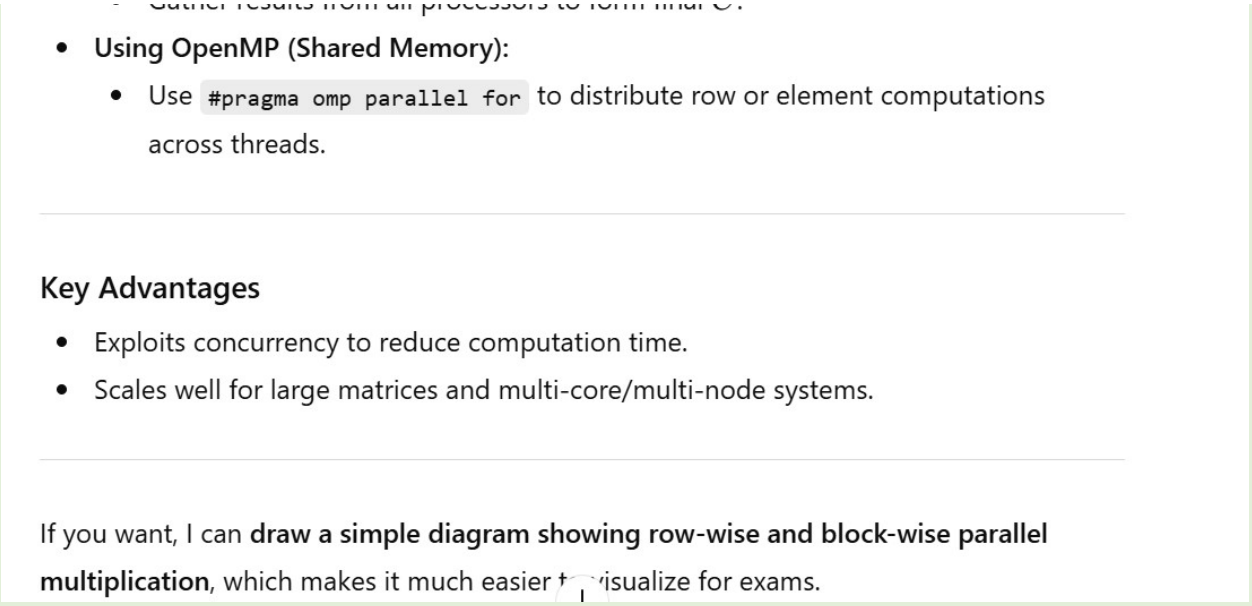
$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Parallel Execution:

- Suppose we have **2 processors**:
  - Processor 1 computes **row 0** → `[19, 22]`
  - Processor 2 computes **row 1** → `[43, 50]`
- Both processors work simultaneously → faster computation.

Implementation in Parallel Systems

- **Using MPI (Message Passing Interface):**
  - Scatter rows of A to multiple processors.
  - Broadcast matrix B to all processors.
  - Each processor computes its assigned rows of C.
  - Gather results from all processors to form final C.



  - Gather results from all processors to form final C.
- **Using OpenMP (Shared Memory):**
  - Use `#pragma omp parallel for` to distribute row or element computations across threads.

Key Advantages

- Exploits concurrency to reduce computation time.
- Scales well for large matrices and multi-core/multi-node systems.

If you want, I can **draw a simple diagram showing row-wise and block-wise parallel multiplication**, which makes it much easier to visualize for exams.

- **Clg-10: Multithreaded Programming, Parallel Algorithms & Architectures (parallel algorithm models lec #5)**

- **Main Coding Areas:**

  - **Message Passing Interface (MPI):** distributing rows/blocks of matrices across nodes
  - **OpenMP / Multithreading:** parallel computation of rows/elements

Q8. Explain Dijkstra’s algorithm for shortest paths and how it can be implemented in a distributed system.

Dijkstra's Algorithm (Sequential Version)

Purpose:

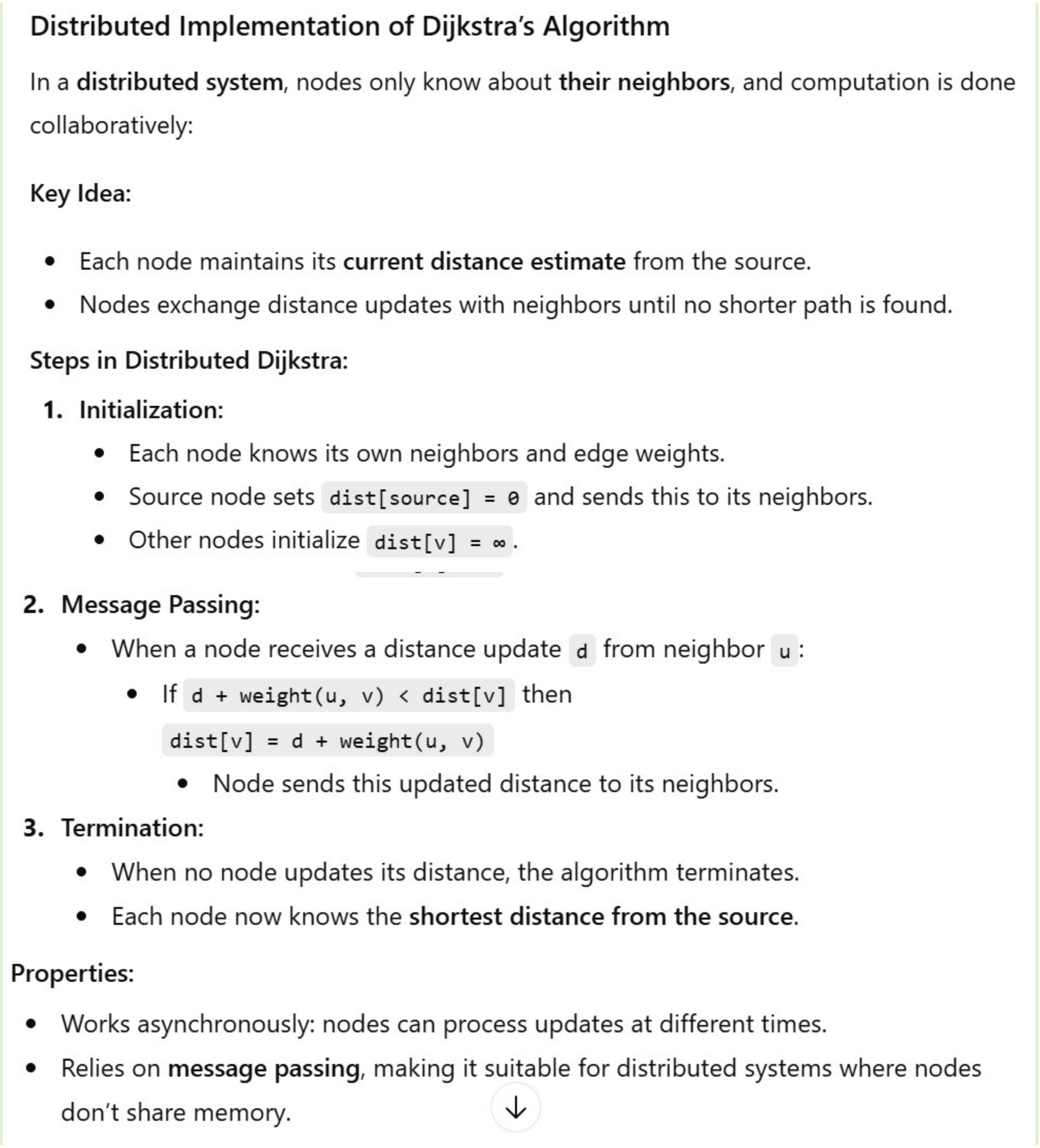
Find the **shortest path** from a source node to all other nodes in a weighted graph with **non-negative edge weights**.

Steps:

1. Initialize:
    - `dist[source] = 0`
    - `dist[v] = ∞` for all other nodes `v`
    - `visited[v] = false` for all nodes
2. Repeat until all nodes are visited:
    1. Pick the unvisited node `u` with the smallest `dist[u]`.
    2. Mark `u` as visited.
    3. For each neighbor `v` of `u`:
        - If `dist[u] + weight(u, v) < dist[v]`, then `dist[v] = dist[u] + weight(u, v)`
3. At the end, `dist[v]` gives the shortest distance from the source to node `v`.

Time Complexity:

- Using a simple array: $O(V^2)$
- Using a priority queue (min-heap): $O(E \log V)$



Distributed Implementation of Dijkstra's Algorithm

In a **distributed system**, nodes only know about **their neighbors**, and computation is done collaboratively:

Key Idea:

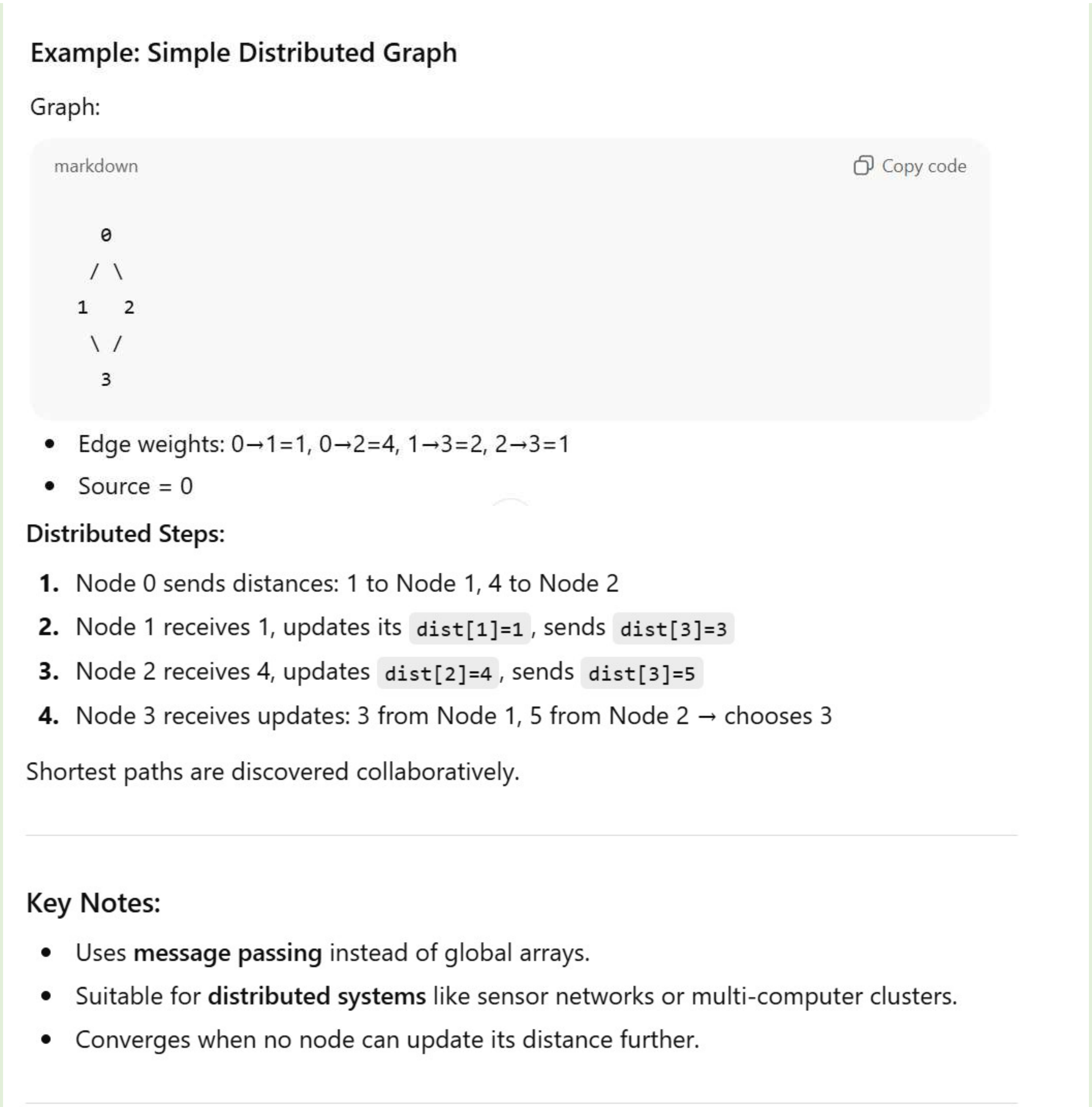
- Each node maintains its **current distance estimate** from the source.
- Nodes exchange distance updates with neighbors until no shorter path is found.

Steps in Distributed Dijkstra:

1. **Initialization:**
    - Each node knows its own neighbors and edge weights.
    - Source node sets `dist[source] = 0` and sends this to its neighbors.
    - Other nodes initialize `dist[v] = ∞`.
2. **Message Passing:**
    - When a node receives a distance update `d` from neighbor `u`:
        - If `d + weight(u, v) < dist[v]` then `dist[v] = d + weight(u, v)`
            - Node sends this updated distance to its neighbors.
3. **Termination:**
    - When no node updates its distance, the algorithm terminates.
    - Each node now knows the **shortest distance from the source**.

Properties:

- Works asynchronously: nodes can process updates at different times.
- Relies on **message passing**, making it suitable for distributed systems where nodes don't share memory.



Example: Simple Distributed Graph

Graph:

```
    0
   / \
  1   2
   \ /
    3
```

- Edge weights: 0→1=1, 0→2=4, 1→3=2, 2→3=1
- Source = 0

Distributed Steps:

1. Node 0 sends distances: 1 to Node 1, 4 to Node 2
2. Node 1 receives 1, updates its `dist[1]=1`, sends `dist[3]=3`
3. Node 2 receives 4, updates `dist[2]=4`, sends `dist[3]=5`
4. Node 3 receives updates: 3 from Node 1, 5 from Node 2 → chooses 3

Shortest paths are discovered collaboratively.

Key Notes:

- Uses **message passing** instead of global arrays.
- Suitable for **distributed systems** like sensor networks or multi-computer clusters.
- Converges when no node can update its distance further.

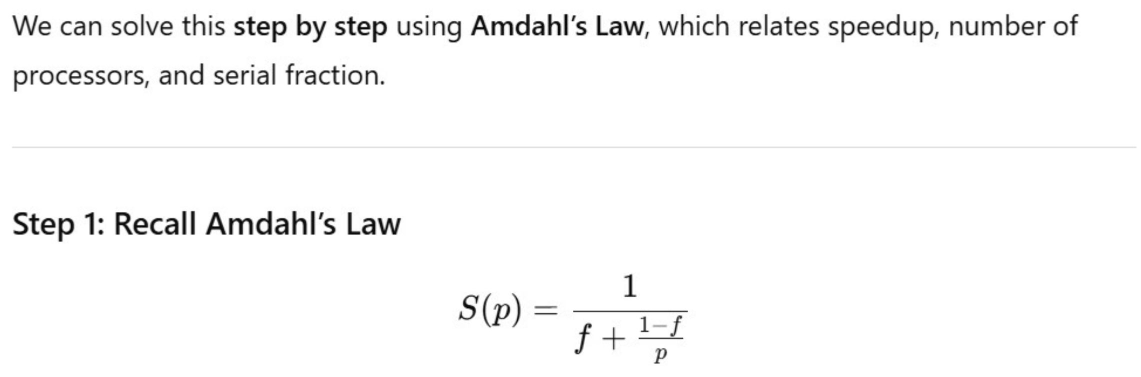
- **Clg-11: Basic Communication Operations (lec #14)** – because distributed Dijkstra relies on message passing between nodes.

- **Parallel Algorithms & Architectures (lec #5 / Clg-10)** – understanding how algorithms can be executed collaboratively across multiple nodes.

Q9. Benchmarking a parallel program on 2, …………, 8 processors produces the following speedup results:

No. of processor	2	3	4	5	6	7	8
(n,p)	1.87	2.61	3.23	3.73	4.14	4.46	4.71

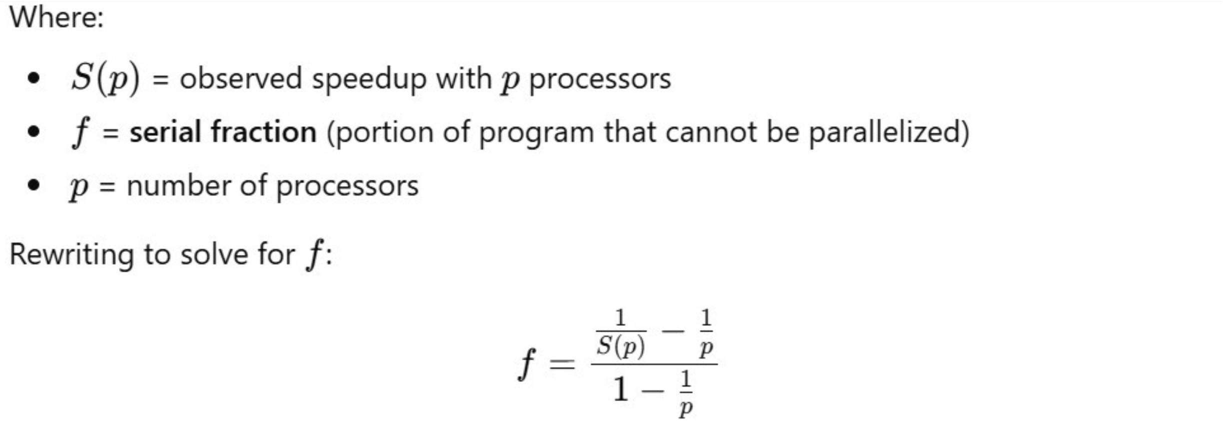
Determine serial fraction e(n,p) and find the primary reason that is contributing to the poor performance, where (n, p) denotes speed up.



We can solve this **step by step** using **Amdahl's Law**, which relates speedup, number of processors, and serial fraction.

Step 1: Recall Amdahl's Law

$$S(p) = \frac{1}{f + \frac{1-f}{p}}$$



Where:

- $S(p)$ = observed speedup with p processors
- f = **serial fraction** (portion of program that cannot be parallelized)
- p = number of processors

Rewriting to solve for f:

$$f = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Step 2: Compute Serial Fraction for Each Processor Count

For p = 2, S = 1.87

$$f = \frac{\frac{1}{1.87} - \frac{1}{2}}{1 - \frac{1}{2}} = \frac{0.5348 - 0.5}{0.5} = \frac{0.0348}{0.5} = 0.0696 \approx 0.07$$

For p = 3, S = 2.61

$$f = \frac{\frac{1}{2.61} - \frac{1}{3}}{1 - \frac{1}{3}} = \frac{0.3831 - 0.3333}{0.6667} = \frac{0.0498}{0.6667} \approx 0.0747 \approx 0.075$$

For p = 4, S = 3.23

$$f = \frac{\frac{1}{3.23} - \frac{1}{4}}{1 - \frac{1}{4}} = \frac{0.3096 - 0.25}{0.75} = \frac{0.0596}{0.75} \approx 0.0795 \approx 0.08$$

For p = 5, S = 3.73

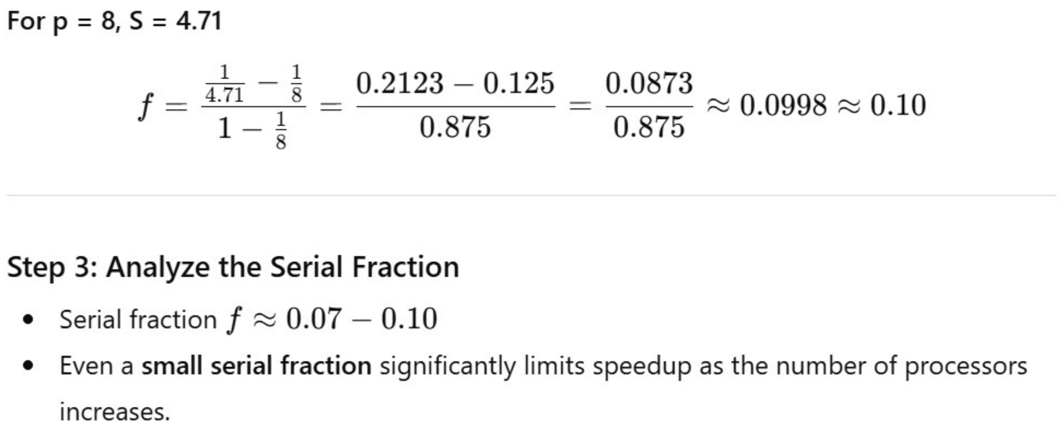
$$f = \frac{\frac{1}{3.73} - \frac{1}{5}}{1 - \frac{1}{5}} = \frac{0.268 - 0.2}{0.8} = \frac{0.068}{0.8} \approx 0.085$$

For p = 6, S = 4.14

$$f = \frac{\frac{1}{4.14} - \frac{1}{6}}{1 - \frac{1}{6}} = \frac{0.2415 - 0.1667}{0.8333} = \frac{0.0748}{0.8333} \approx 0.0897 \approx 0.09$$

For p = 7, S = 4.46

$$f = \frac{\frac{1}{4.46} - \frac{1}{7}}{1 - \frac{1}{7}} = \frac{0.2242 - 0.1429}{0.8571} = \frac{0.0813}{0.8571} \approx 0.0948 \approx 0.095$$

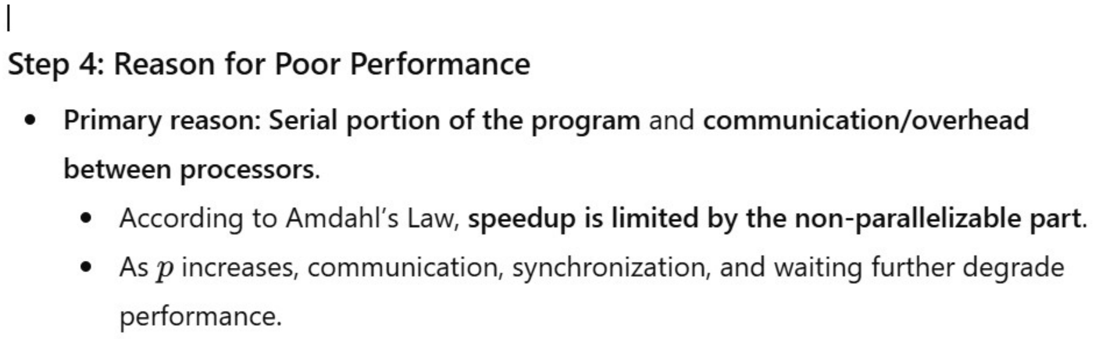


For p = 8, S = 4.71

$$f = \frac{\frac{1}{4.71} - \frac{1}{8}}{1 - \frac{1}{8}} = \frac{0.2123 - 0.125}{0.875} = \frac{0.0873}{0.875} \approx 0.0998 \approx 0.10$$

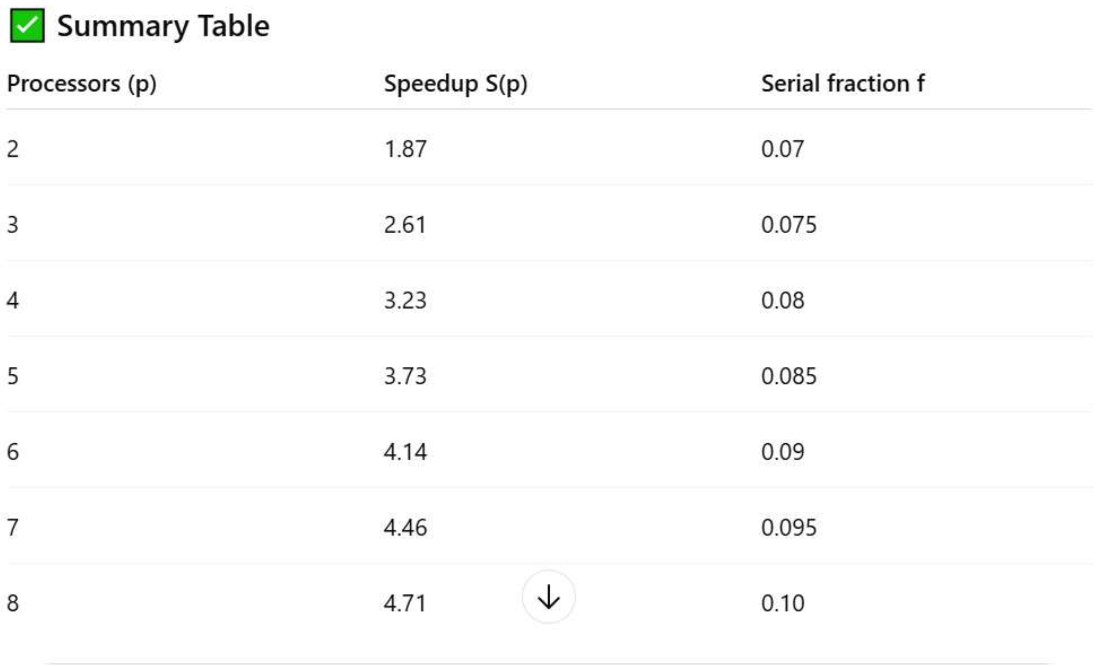
Step 3: Analyze the Serial Fraction

- Serial fraction $f \approx 0.07 - 0.10$
- Even a **small serial fraction** significantly limits speedup as the number of processors increases.



Step 4: Reason for Poor Performance

- **Primary reason: Serial portion of the program** and **communication/overhead between processors.**
  - According to Amdahl's Law, **speedup is limited by the non-parallelizable part**.
  - As p increases, communication, synchronization, and waiting further degrade performance.



✅ Summary Table

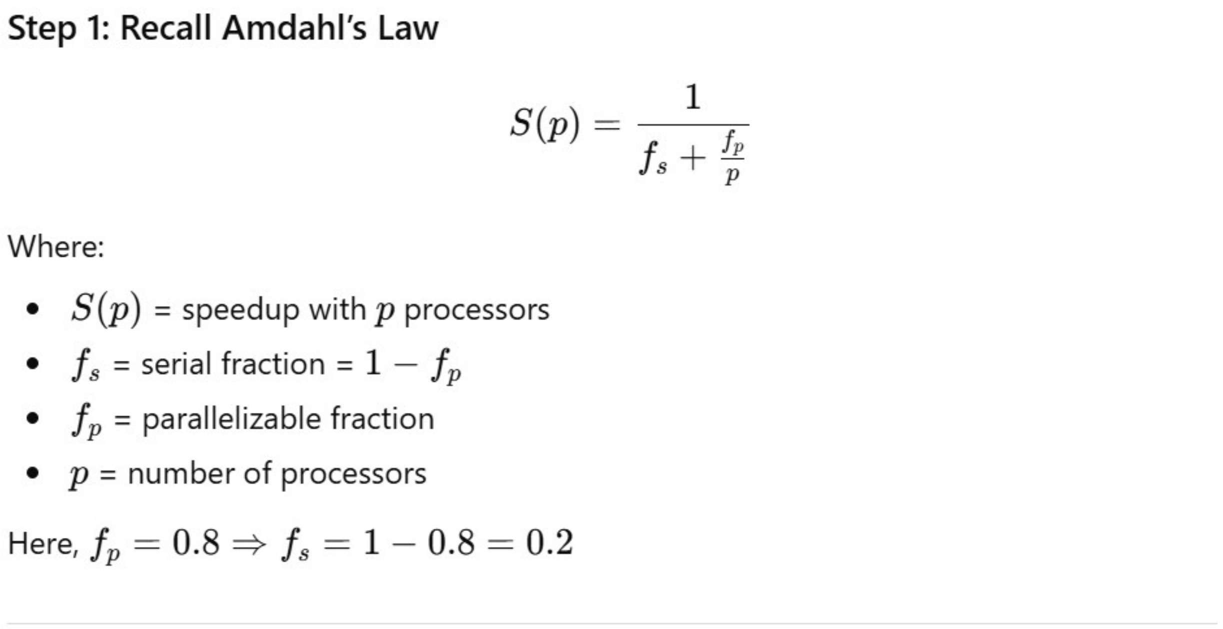
Processors (p)	Speedup S(p)	Serial fraction f
2	1.87	0.07
3	2.61	0.075
4	3.23	0.08
5	3.73	0.085
6	4.14	0.09
7	4.46	0.095
8	4.71	0.10

Conclusion:

- Serial fraction and inter-processor overhead limit parallel speedup, causing **poor performance as processors increase**.

- • **Clg-5 / Clg-13: Performance analysis and tuning, scalability and performance studies**
- • Concepts covered include **speedup, serial fraction, and factors affecting parallel program performance** in **parallel and distributed computing**.

Q10. Suppose you have a program with parallelizable fraction f= 0.8, meaning 80% of the program can be parallelized. Calculate the speedup for the program using Amdahl's Law for P=1, 2,4 and 8 processors. Also find No. of processors required for maximum speedup.



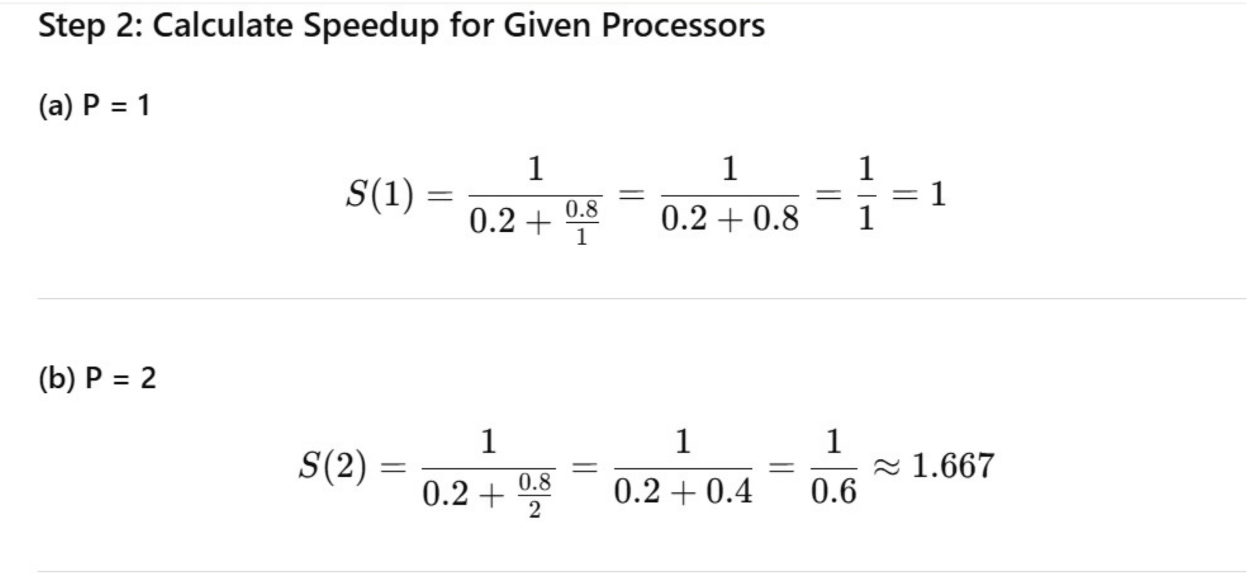
Step 1: Recall Amdahl's Law

$$S(p) = \frac{1}{f_s + \frac{f_p}{p}}$$

Where:

- $S(p)$ = speedup with p processors
- f_s = serial fraction = $1 - f_p$
- f_p = parallelizable fraction
- p = number of processors

Here, $f_p = 0.8 \Rightarrow f_s = 1 - 0.8 = 0.2$



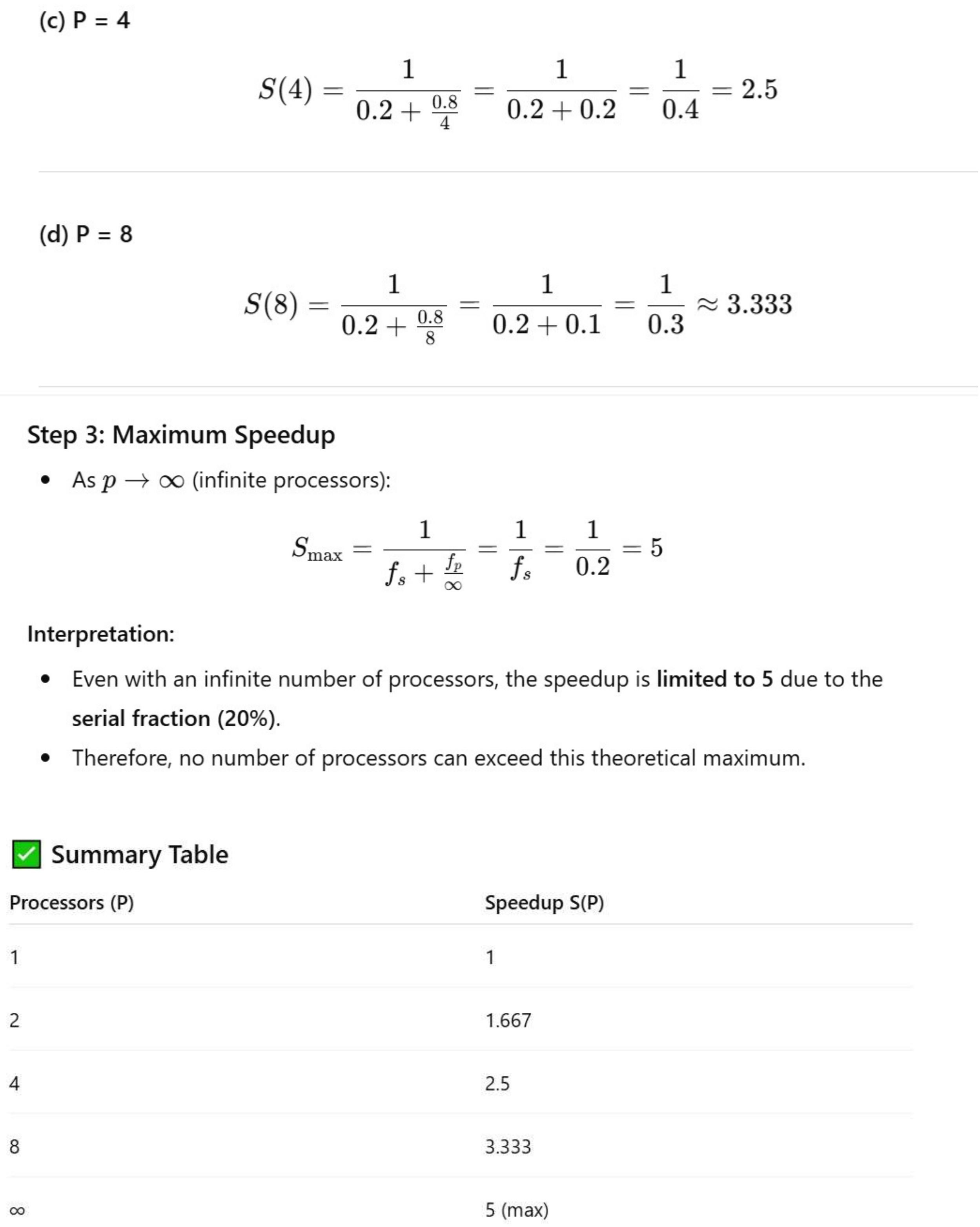
Step 2: Calculate Speedup for Given Processors

(a) P = 1

$$S(1) = \frac{1}{0.2 + \frac{0.8}{1}} = \frac{1}{0.2 + 0.8} = \frac{1}{1} = 1$$

(b) P = 2

$$S(2) = \frac{1}{0.2 + \frac{0.8}{2}} = \frac{1}{0.2 + 0.4} = \frac{1}{0.6} \approx 1.667$$



(c) P = 4

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

(d) P = 8

$$S(8) = \frac{1}{0.2 + \frac{0.8}{8}} = \frac{1}{0.2 + 0.1} = \frac{1}{0.3} \approx 3.333$$

Step 3: Maximum Speedup

- As $p \to \infty$ (infinite processors):

$$S_{\max} = \frac{1}{f_s + \frac{f_p}{\infty}} = \frac{1}{f_s} = \frac{1}{0.2} = 5$$

Interpretation:

- Even with an infinite number of processors, the speedup is **limited to 5** due to the **serial fraction (20%)**.
- Therefore, no number of processors can exceed this theoretical maximum.

✅ Summary Table

Processors (P)	Speedup S(P)
1	1
2	1.667
4	2.5
8	3.333
∞	5 (max)

Key Insight:

- The **serial portion of the program limits maximum speedup**, which is a fundamental concept in **parallel computing and performance analysis**.

◆ 1️⃣ Most Important Topics (High Focus)

Ye topics **exam, coding assignments, aur viva** ke liye critical hain.

Topic	Reason	Focus Tips
MPI (Message Passing Interface)	Core distributed programming tool; projects / labs	Example: matrix multiplication, All-to-All broadcast/reduction
OpenMP / Multithreading	Shared memory parallel programming; critical for parallel algorithms	Example: parallel sorting, sum reduction
Parallel Algorithms	Conceptual + coding questions; needed for exam	Practice: merge sort, prefix sum, search algorithms
Performance Analysis & Tuning	Bloom's Taxonomy: analyze & evaluate parallel programs	Understand speedup, efficiency, Amdahl's Law
Memory Hierarchies & Consistency Models	Important for reasoning about performance	Focus on read/write ordering, caching issues
Concurrency / Synchronization	Critical for threads and parallel execution	Mutex, semaphore, deadlock, race condition
Task Parallel / Shared vs Distributed Memory	Core concepts; exam questions	Understand difference between multithreading and message-passing
GPU Programming (CUDA)	Modern and impressive; sometimes project/viva question	Focus on kernel basics, threads vs blocks, parallelization
Cloud/Distributed Tools Overview (AWS, Hadoop, OpenStack, Globus, Condor)	Conceptual; exam/project discussion	Don't memorize everything; focus on purpose/use-case

✓ **Tip:** MPI + OpenMP + Parallel Algorithms + Performance Analysis = backbone of course.

🔹 2️⃣ Medium Priority Topics (Moderate Focus)

Ye topics **conceptually samajhne hain**, coding mein kam use hote hain.

Topic	Reason	Focus Tips
Scheduling & Load Balancing	Exam conceptual questions; real-world relevance	Focus on idea, types of scheduling (static vs dynamic)
Interconnection Topologies	Conceptual understanding; exam diagrams	Mesh, Hypercube, Star, Ring – basic properties
Decomposition Techniques	Used in algorithm design	Block, cyclic, recursive decomposition
MIMD / SIMD	Parallel architecture concept	Understand difference & example
Parallel I/O	Occasionally asked in exams	Conceptual: how data read/write happens in parallel systems
Fault Tolerance	Conceptual; may appear in viva	Replication, checkpointing

---

🔹 3️⃣ Low Priority / Optional Topics (Skip if time is limited)

Ye topics **mostly advanced / tools overview** — focus only if time permits.

Topic	Reason	Notes
FUSE (Filesystem in Userspace)	Advanced, rare in exams	Skip unless project requires
Swift / Condor / Globus	Distributed toolkits; rarely exam focus	Just know purpose, don't memorize commands
Amazon AWS / OpenStack	Cloud deployment tools	Conceptual: storage, compute, virtualization
Power / HPC-specific topics	Mostly research-level	Can skip for exam unless directly asked
Extra Lecture #6 Communication Cost	Minor detail	Conceptual understanding enough

---

🔹 4️⃣ Recommended Study Strategy

1. **Coding Priority:**

    - MPI → 40% focus
    - OpenMP / Threads → 30% focus
    - CUDA / GPU → 10–15% focus
    - Parallel Algorithms → 15% focus
2. **Conceptual Priority:**
    - Shared vs distributed memory → high
    - Synchronization, concurrency → high
    - Performance & scalability → high
    - Cloud tools / advanced toolkits → low unless project needs
3. **Viva / Exam Tip:**
    - Examiner usually **focuses on MPI, OpenMP, parallel algorithms, synchronization, and memory**.
    - Cloud / Hadoop / AWS / OpenStack = only conceptual questions.

✓ Key Takeaways

- **High Focus:** MPI, OpenMP, Parallel Algorithms, Concurrency, Memory
- **Medium Focus:** Scheduling, Load Balancing, MIMD/SIMD, Decomposition, Fault Tolerance
- **Low / Optional:** FUSE, Swift, Globus, Condor, AWS, OpenStack, Power/HPC

🔹 Coding ka scope in PDC

1. **MPI Programs** (small examples)
    - Jaise matrix addition/multiplication across nodes
    - "Hello World" communication between nodes
2. **OpenMP / Threads** (small examples)
    - Parallel sum, sorting, reduction
    - Loop-level parallelism
3. **CUDA / GPU** (optional/demo)
    - Basic kernels (vector addition, image filter)
4. **Parallel Algorithm Pseudocode**
    - Merge sort, prefix sum, parallel search
    - Mostly exam / assignment level, not production code
5. **Tool Usage Examples** (Cloud / AWS / Hadoop)
    - Conceptual commands
    - MapReduce pseudo-implementation

🔹 Reality check

- Tum **full applications** nahi likhoge, **small programs or demos** hi enough hain
- **Exam / viva** mein mostly conceptual questions + small snippets poochhe jate hain
- Focus should be on:
    1. **Logic samajhna**
    2. **Parallel execution ka flow**
    3. **Data synchronization / communication**

✓ Conclusion

- **Large-scale coding / software project nahi hota**
- **Small demos / assignments** sufficient hain
- Agar ye samajh lo, **coding tension nahi hogi**
