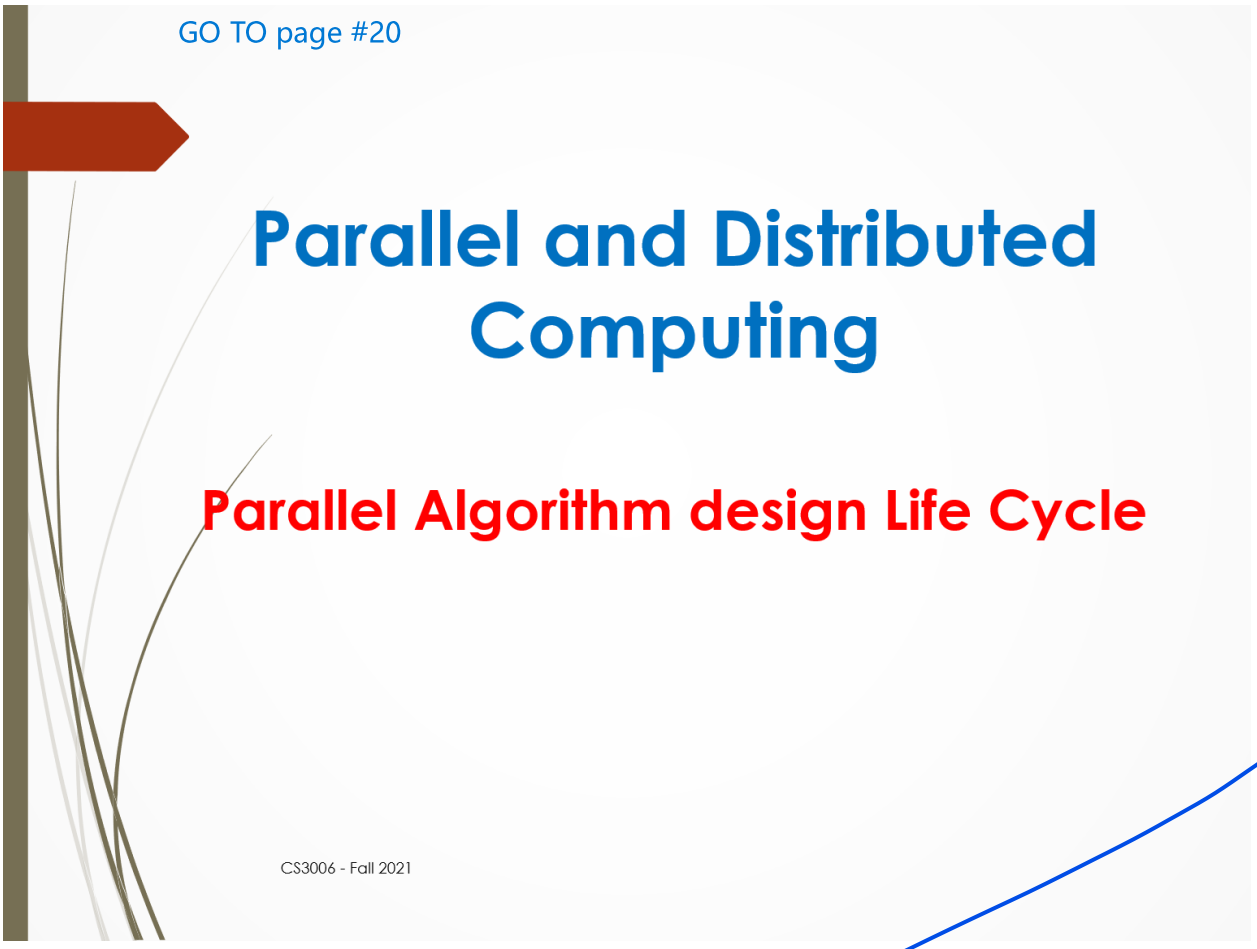


Here's a **clear, structured explanation** of the topic:

💻 Parallel and Distributed Computing

Parallel and distributed computing are two related paradigms aimed at **speeding up computations** by performing multiple operations **simultaneously**.

⚙️ Parallel Computing

- Multiple processors (or cores) work together on **a single problem**, sharing memory and data.
- Goal: **Increase speed** and **efficiency**.
- Example: Matrix multiplication using multiple CPU cores.

✅ Characteristics:

- Shared memory space
- Tight coupling (processors communicate frequently)
- Low communication delay

🌐 Distributed Computing

- Multiple **independent computers** (nodes) connected through a network.
- Each node has its **own memory** and may work on **different parts of a problem**.
- Goal: **Scalability** and **fault tolerance**.
- Example: Cloud computing, Hadoop, or blockchain networks.

✅ Characteristics:

- Separate memory space
- Loosely coupled systems
- Communication via message passing

🔄 Parallel Algorithm Design Life Cycle

Designing a **parallel algorithm** means structuring a problem so multiple processors can cooperate efficiently.

The **Parallel Algorithm Design Life Cycle** includes the following stages:

🧩 1. Problem Decomposition

Break the problem into smaller **tasks or subproblems** that can run concurrently.

Types:

- **Data decomposition:** Divide data into chunks (e.g., array blocks).
- **Functional decomposition:** Divide based on operations or functions.

🎯 *Goal:* Identify **independent tasks**.

🔗 2. Task Dependency Analysis

Determine how tasks depend on each other.

- If Task B needs Task A's output, A → B is a **dependency**.
- Represented using a **Task Dependency Graph (DAG)**.

🎯 *Goal:* Find which tasks can run in parallel and which must wait.

⚖️ 3. Concurrency Analysis

Measure how much parallelism is possible.

- **Maximum degree of concurrency:** Max number of tasks that can run simultaneously.
- **Critical path length:** Longest chain of dependent tasks.
- **Average degree of concurrency:** Total work ÷ critical path length.

🎯 *Goal:* Estimate performance gain and scalability.

⚙️ 4. Granularity Determination

Granularity = **amount of work per task** relative to communication cost.

- **Fine-grained:** Many small tasks (high communication).
- **Coarse-grained:** Few large tasks (low communication).

🎯 *Goal:* Choose balance between computation and communication overhead.

🔄 5. Task Interaction and Communication

Analyze **how tasks share data or synchronize**.

- Represented using **Task Interaction Diagrams** (arrows show communication).
- Minimize communication between processors.

🎯 *Goal:* Reduce communication cost and avoid data conflicts.

🧭 6. Process and Mapping

Assign tasks to processors (mapping).

- **Static mapping:** Fixed before execution.
- **Dynamic mapping:** Decided at runtime based on workload.

🎯 *Goal:* Achieve load balancing and minimize idle time.

🚀 7. Execution and Performance Evaluation

Run the algorithm and evaluate:

- **Speedup** = (Sequential time) / (Parallel time)
- **Efficiency** = Speedup / Number of processors
- **Scalability** = How performance changes as processors increase

🎯 *Goal:* Optimize for best performance and resource utilization.

🔁 Summary Table

Step	Description	Goal
1. Problem Decomposition	Break problem into tasks	Identify parallelism
2. Task Dependency Analysis	Identify relationships	Determine execution order
3. Concurrency Analysis	Measure parallel potential	Estimate speedup
4. Granularity	Size of tasks	Balance compute vs. communication
5. Task Interaction	Study data sharing	Reduce communication cost
6. Process & Mapping	Assign tasks to processors	Load balancing
7. Execution & Evaluation	Run & measure performance	Optimize algorithm

---

🧠 Parallel Algorithm Design Life Cycle

It defines the **stages involved in designing an efficient parallel algorithm**. The goal is to divide a problem into smaller parts that can be executed **simultaneously** to reduce computation time.

🔷 Stages:

1. **Problem Decomposition**
   - Break the problem into smaller subproblems or *tasks* that can be solved in parallel.
2. **Task Dependency Analysis**
   - Identify which tasks depend on others and which can run independently.
3. **Task Scheduling**
   - Assign tasks to processors efficiently to minimize idle time and maximize utilization.
4. **Mapping**
   - Map tasks to specific processors or cores.

5. **Execution**
    - Run the algorithm in parallel.
6. **Performance Analysis**
    - Measure speedup, efficiency, scalability, and identify bottlenecks.

🧩 Tasks, Decomposition, and Task-Dependency Graphs

🔷 Task

A *task* is the smallest unit of computation that can be executed independently.

🔷 Decomposition

The process of **breaking the main problem** into smaller subtasks that can run in parallel.
Common decomposition techniques:

- **Data decomposition:** Split data (e.g., arrays, matrices) among processors.
- **Functional decomposition:** Split different operations or functions among processors.

🔷 Task-Dependency Graph (TDG)

A **directed acyclic graph (DAG)** that shows tasks as nodes and dependencies as directed edges.

- **Edge (A → B):** Task B depends on task A.
- Helps determine **concurrency**, **critical path**, and **scheduling order**.

⚖️ Granularity

Granularity refers to **the amount of computation per communication** between tasks.

🔶 Fine-grained

- Tasks are small and numerous.
- High communication overhead.
- Example: dividing an array into hundreds of tiny chunks.
- Suitable for shared-memory systems.

🔶 Coarse-grained

- Tasks are larger, fewer, and more independent.
- Lower communication overhead.
- Example: dividing a problem into a few big modules.
- Suitable for distributed systems.

⚙️ Concurrency

Concurrency is the **number of tasks that can execute simultaneously** at a given time.

🔷 Maximum Degree of Concurrency

- The **largest number of tasks** that can run in parallel at any point in the execution.
- Determined from the task-dependency graph.

🔷 Critical Path Length

- The **longest path** from start to end in the task-dependency graph.
- Represents the **minimum possible execution time** (even with infinite processors).

🔷 Average Degree of Concurrency

$$
\text{Average Degree of Concurrency} = \frac{\text{Total Work}}{\text{Critical Path Length}}
$$

- Where “Total Work” = total sum of all task times.
- Indicates the **overall parallelism** of the algorithm.

🔄 Task-Interaction Diagrams

- Show **how tasks communicate** and share data.
- Represented as:
  - **Arrows:** communication or synchronization.
  - **Nodes:** individual tasks.
- Helps identify **bottlenecks**, **data sharing issues**, and **communication costs**.

🧭 Processes and Mapping

🔷 Processes

- A process (or thread) executes one or more tasks.
- Each process may run on a separate processor or core.

🔷 Mapping

- The **assignment of tasks to processes (or processors)**.
- Goal: minimize communication, balance load, and reduce idle time.

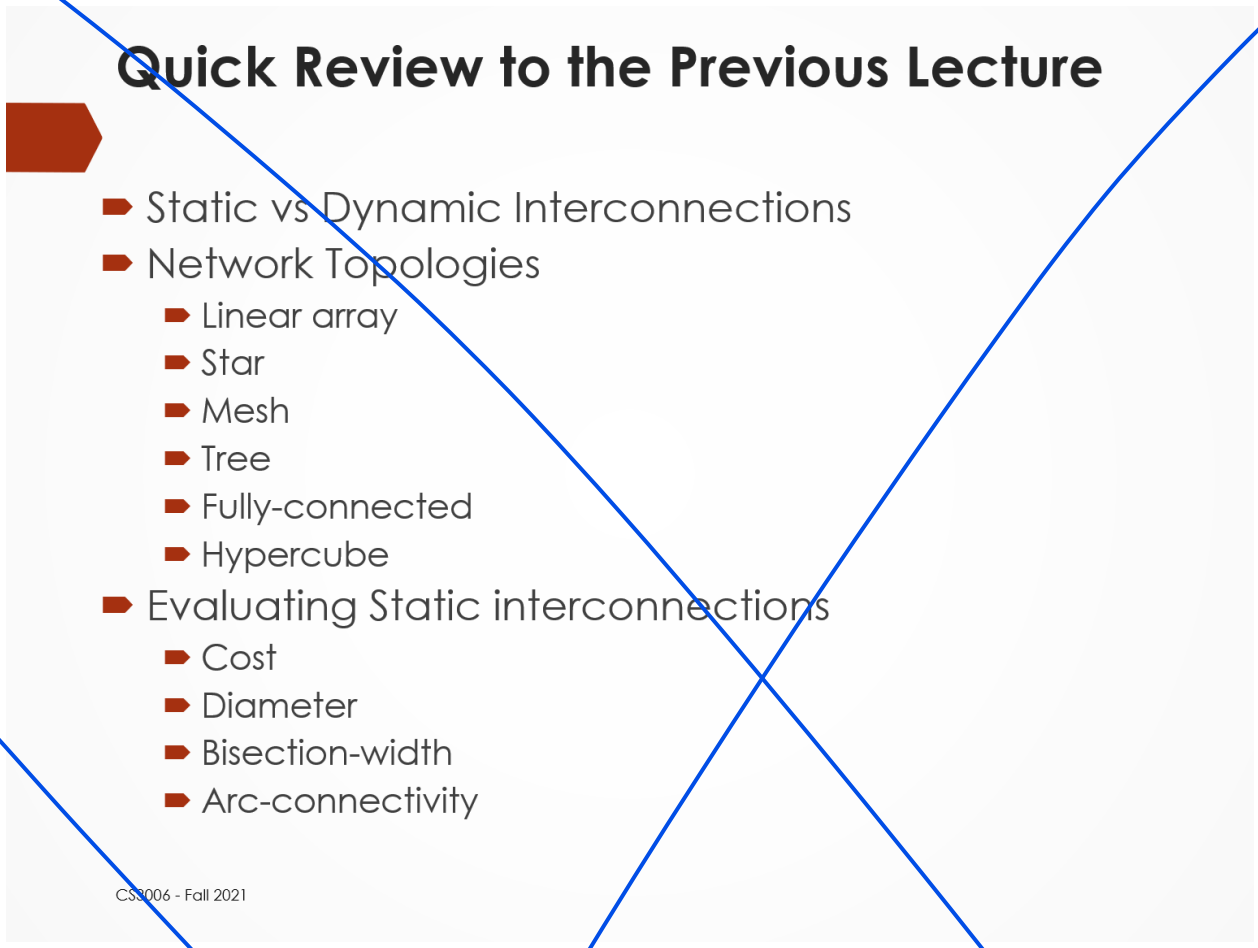
Types:

- **Static Mapping:** Fixed before execution (predictable).
- **Dynamic Mapping:** Decided at runtime (adaptive).

✅ Summary Table

Concept	Meaning
Task	Smallest independent computation unit
Decomposition	Splitting problem into tasks
Task-dependency Graph	DAG showing execution order and dependencies
Granularity	Amount of computation per communication
Fine-grained	Many small tasks, more communication
Coarse-grained	Few large tasks, less communication
Max-degree of concurrency	Highest number of tasks executable at once
Critical path	Longest dependent sequence of tasks
Avg-degree of concurrency	Total work / Critical path length
Task-interaction diagram	Shows communication between tasks
Processes & Mapping	Assigning tasks to processors efficiently

---



Perfect 👍 — here's a **complete, clear, and well-structured explanation** of these concepts from **Parallel and Distributed Computing**, written in an easy exam-oriented way 👇

⚙️ Static vs Dynamic Interconnections

When we connect multiple processors in a parallel or distributed system, the way they **communicate** is called the **interconnection network**.

There are two main types:

🔷 1. Static Interconnection Networks

- Connections between processors are **fixed** and **do not change** during execution.
- Communication occurs through **dedicated physical links**.

- Used in **message-passing multiprocessors** and **cluster systems**.

✅ Examples:

Linear array, ring, mesh, tree, star, hypercube, fully-connected.

✅ Advantages:

- Simple hardware design
- Fast communication (dedicated paths)
- Predictable performance

⚠️ Disadvantages:

- Limited flexibility
- Difficult to handle dynamic traffic patterns

🔷 2. Dynamic Interconnection Networks

- Connections between processors are **established as needed** through **switches or routers**.
- Paths change dynamically depending on communication requests.
- Used in **multistage interconnection networks** and **modern parallel systems**.

✅ Examples:

Crossbar, Omega, Banyan, and Bus-based networks.

✅ Advantages:

- Flexible and scalable
- Can handle varying communication patterns

⚠️ Disadvantages:

- More complex hardware
- Possible contention and latency

🌐 Common Static Network Topologies

Below are the most commonly used **static interconnection topologies** in parallel systems 👇

🔶 1. Linear Array

- Processors are connected in a straight line.

```
P1 — P2 — P3 — P4 — P5
```

🔷 Properties:

- Each processor connects to its **two neighbors** (except ends).
- **Simple**, but **long communication path** between distant nodes.

🔶 2. Star Topology

- One central processor connected to all others.

```
        P2
         |
P3 — P1 — P4
         |
        P5
```

🔷 Properties:

- Central node acts as a **hub**.
- Easy to manage but **failure of hub** disconnects the network.

🔶 3. Mesh Topology

- Processors arranged in a **2D grid**; each node connects to its adjacent nodes.

```
P1 — P2 — P3
|    |    |
P4 — P5 — P6
|    |    |
P7 — P8 — P9
```

🔷 Properties:

- Common in parallel computers (e.g., 2D array of processors).
- **Short local connections**, scalable.
- **Corner nodes** have fewer connections.

🔶 4. Tree Topology

- Processors connected hierarchically like a tree structure.

```
      P1
    /    \
  P2      P3
 /  \    /  \
P4  P5  P6  P7
```

🔹 Properties:

- Supports **broadcasting** efficiently (top-down).
- **Root or higher-level failure** affects the entire network.
- Used in hierarchical systems.

🔶 5. Fully-Connected (Complete) Network

- Every processor is connected to **every other processor** directly.

```
(Each node connected to all others)
```

🔹 Properties:

- Fastest communication (1 hop between any pair).
- **High cost** (many links).
- Used only for small systems.

🔶 6. Hypercube (n-Cube)

- Processors are represented by **n-bit binary addresses**.
- Two processors are connected if their addresses differ by **exactly one bit**

Example:
For 3D hypercube (8 nodes):

```
000—001
 |    |
010—011
 |    |
100—101
 |    |
110—111
```

🔷 Properties:

- Very popular in parallel computing.
- **Highly connected**, **short paths**, **$\log_2(N)$** diameter.
- **Scalable and symmetric**.

📊 Evaluating Static Interconnection Networks

When comparing or designing static networks, we use the following metrics 👇

🔶 1. Cost

$$
\text{Cost} = \text{Number of Links} + \text{Number of Processors}
$$

- Represents **total hardware cost**.
- Fewer links → cheaper but possibly slower.

🔶 2. Diameter

- The **longest shortest path** between any two nodes in the network.
- Represents the **maximum communication delay**.

📌 Example:

- Linear array (N nodes): Diameter = N − 1
- Hypercube ($N = 2^n$): Diameter = n

🔶 3. Bisection Width

- The **minimum number of links** that must be removed to divide the network into **two equal halves**.

📌 Meaning:
Higher bisection width → better **parallel data transfer**.

Example:

- Linear array (N): 1
- 2D Mesh (√N × √N): √N
- Hypercube ($N = 2^n$): N / 2

🔶 4. Arc-Connectivity (or Edge-Connectivity)

- The **minimum number of links** that must fail to **disconnect the network**.

📌 Meaning:
Higher arc-connectivity = **more fault-tolerant**.

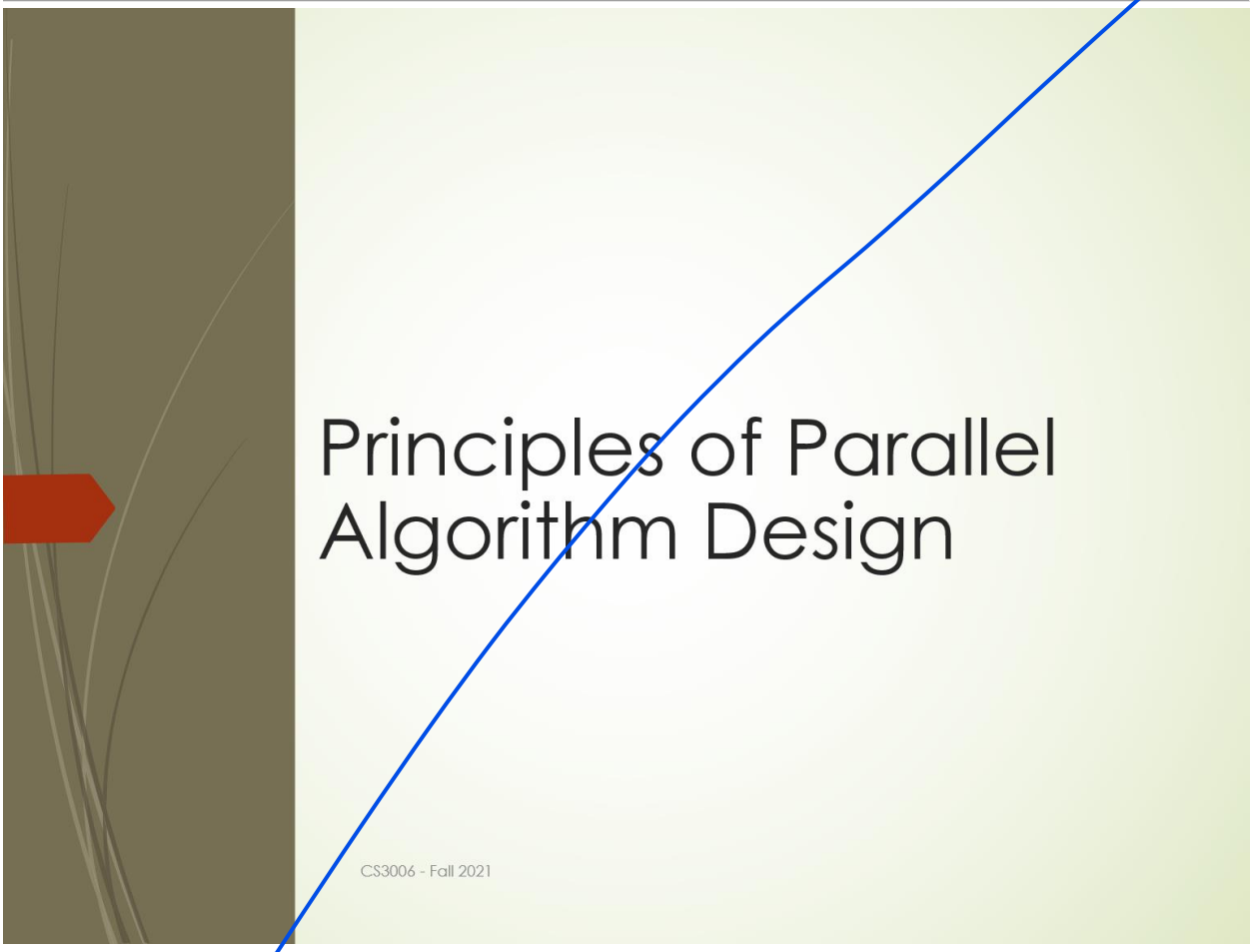
Example:

- Linear array: 1
- Mesh: 2 (for interior nodes)
- Hypercube: n (for n-cube)

✅ Summary Table

Topology	Links per Node	Diameter	Bisection Width	Fault Tolerance
Linear Array	2	N − 1	1	Low
Star	N − 1 (hub), 1 (others)	2	1	Low (hub failure)
Mesh (√N × √N)	2–4	2(√N−1)	√N	Medium

Topology	Links per Node	Diameter	Bisection Width	Fault Tolerance
Tree	$\log_2 N$	$2\log_2 N$	1	Low (root failure)
Fully Connected	$N-1$	1	$N^2/4$	Very High
Hypercube ($N=2^n$)	n	n	$N/2$	High



Excellent 👌 — here's a **complete, clear, and structured explanation** of the **Principles of Parallel Algorithm Design**, perfect for your **exam notes or understanding** in **Parallel and Distributed Computing** 👇

⚙️ Principles of Parallel Algorithm Design

Designing an efficient **parallel algorithm** means dividing a large problem into smaller parts that can be executed **simultaneously** on multiple processors — while minimizing communication and synchronization overhead.

There are **two major principles** (or phases) generally followed in parallel algorithm design:

1. **Algorithmic Design Principles** (How to identify parallelism)
2. **Performance Optimization Principles** (How to use that parallelism efficiently)

Let's go step by step 👇

🧩 1. Algorithmic Design Principles

These principles focus on how to **decompose** and **structure** the algorithm for parallel execution.

🔷 A. Decomposition

Definition:
Divide the problem into smaller **tasks or subproblems** that can be solved concurrently.

There are two main types:

- **Data Decomposition:**
  Divide the data into parts; each processor performs the same operation on different data.
  🧠 Example: Matrix multiplication — each processor handles a block of the matrix.
- **Functional Decomposition:**
  Divide tasks by functions or operations.
  🧠 Example: One processor reads data, another processes it, another writes results.

🎯 **Goal:** Expose as much parallelism as possible.

🔷 B. Task Dependency and Concurrency

After decomposition, identify **dependencies** among tasks.

- **Independent tasks** → can run in parallel.
- **Dependent tasks** → must follow order.

Represented using a **Task Dependency Graph (TDG)**.

🎯 **Goal:** Maximize **concurrency** and minimize **sequential dependencies**.

🔷 C. Mapping Tasks to Processes

Once tasks are defined, assign them to processors.

- **Static Mapping:** Tasks assigned before execution.
- **Dynamic Mapping:** Tasks assigned during execution (based on workload).

🎯 **Goal:** Ensure **load balancing** — all processors have roughly equal work.

🔷 D. Granularity

Granularity = size of a task relative to the communication overhead.

- **Fine-grained:** Many small tasks → high communication cost.
- **Coarse-grained:** Fewer large tasks → less communication, easier to manage.

🎯 **Goal:** Choose a balance that maximizes computation and minimizes communication.

🔷 E. Communication

Tasks must exchange data efficiently.

- **Shared Memory Systems:** via variables.
- **Distributed Systems:** via message passing.

🎯 **Goal:** Reduce communication frequency and volume.

🔷 F. Synchronization

Ensure tasks that depend on others **wait** for the required results.

- **Barrier synchronization:** All tasks reach a common point before continuing.
- **Locks / Semaphores:** Used for shared data.

🎯 **Goal:** Maintain **correctness** with **minimal waiting time**.

⚙️ 2. Performance Optimization Principles

After designing the basic structure, focus on optimizing performance by minimizing bottlenecks.

🔷 A. Load Balancing

Distribute the total workload evenly across processors.

- Avoid **idle processors** and **hotspots**.
- Use **work stealing** or **dynamic scheduling** if needed.

🎯 **Goal:** Maximize processor utilization.

🔷 B. Minimizing Communication

Communication between processors is expensive.

- Group related tasks on the same processor.
- Use **local data access** as much as possible.

🎯 **Goal:** Reduce data transfer time.

🔷 C. Minimizing Synchronization

Synchronization introduces **waiting**.

- Reduce dependencies between tasks.
- Avoid unnecessary locks or barriers.

🎯 **Goal:** Improve concurrency and reduce idle time.

🔷 D. Reducing Overheads

Parallelization adds extra work (thread creation, message passing, etc.)

- Combine smaller tasks.
- Reuse processes/threads.

🎯 **Goal:** Keep overhead < computation time gain.

🔷 E. Scalability

A good parallel algorithm should maintain efficiency when:

- Input size increases, or
- Number of processors increases.

🎯 **Goal:** Algorithm remains efficient even with many processors.

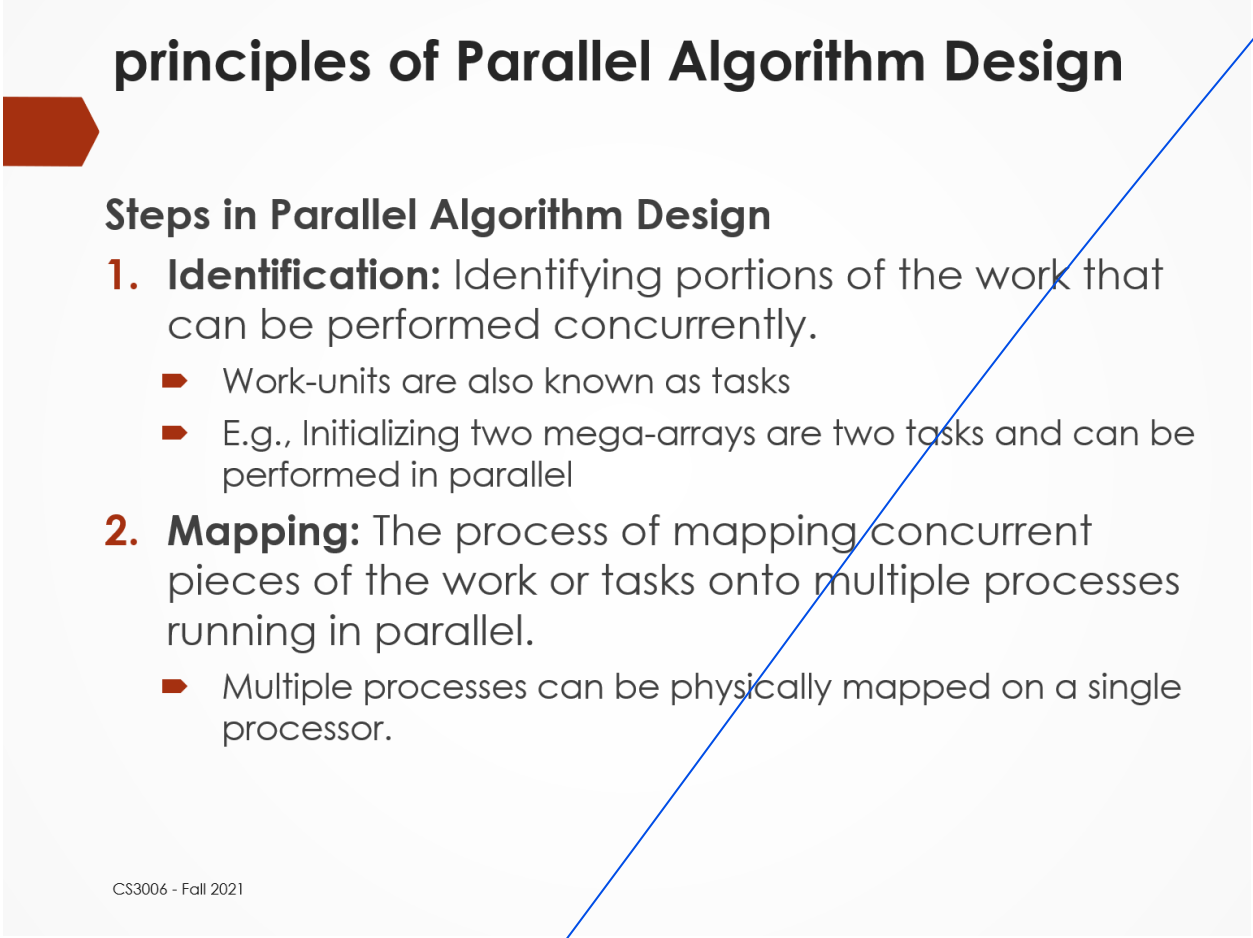
🧭 Summary Table

Principle	Description	Goal
Decomposition	Break problem into subproblems	Identify parallel tasks
Dependency Analysis	Find which tasks depend on others	Maximize concurrency
Mapping	Assign tasks to processors	Balance load
Granularity	Task size vs. communication	Optimize computation
Communication	Exchange of data	Reduce communication cost
Synchronization	Manage task timing	Maintain correctness
Load Balancing	Equal work for all processors	Avoid idle processors
Scalability	Efficiency with more processors	Long-term performance

---

🚀 Key Objectives of Parallel Algorithm Design

1. **Correctness** – The result must match the sequential version.
2. **Efficiency** – Minimize idle time, overhead, and communication.
3. **Scalability** – Performance should improve as processors increase.
4. **Portability** – Should work on different architectures.



Process is logical agent for computation over a physical processing element(processor).

Perfect 👌 — here's a **clear and well-structured explanation** of the **Steps in Parallel Algorithm Design**, including your given points (Identification and Mapping) — explained in simple, exam-ready language 👇

⚙️ Steps in Parallel Algorithm Design

Designing a parallel algorithm involves several key steps that help identify parallelism, manage dependencies, and efficiently map work to processors.

The **main goal** is to divide the problem into smaller, concurrent tasks and execute them efficiently on multiple processors.

🔷 1. Identification (Finding Parallelism)

Definition:

The **first step** in designing a parallel algorithm is to **identify portions of the work** that can be performed **concurrently (in parallel)**.

These independent pieces of work are called **tasks** or **work-units**.

Key Points:

- Identify **independent computations** that don't depend on each other's results.
- Each independent computation can be executed as a **separate task**.
- Tasks may represent:
    - Independent data elements
    - Independent functions
    - Independent loop iterations

Example:

If you need to **initialize two large arrays**, you can do both simultaneously:

```
Task 1 → Initialize array A
Task 2 → Initialize array B
```

Both tasks are independent, so they can run **in parallel** on different processors.

Goal of Identification:

- Find **maximum concurrency** (the most parallel work possible).
- Ensure tasks are **independent** to avoid unnecessary synchronization.

🔷 2. Mapping (Assigning Tasks to Processes)

Definition:

Once tasks are identified, the next step is to **map** these concurrent tasks onto multiple **processes** (logical computation agents).

Each **process** represents a unit of execution that can run one or more tasks.

Key Concept:

"Mapping" is the process of deciding **which task runs on which process** (and indirectly on which processor).

- Multiple **processes** can be **physically mapped** to a **single processor**.
- This happens when the number of tasks > number of processors.

Terminology Clarification:

- **Process:** A *logical agent* responsible for executing a task or part of the work.
- **Processor:** A *physical hardware element* (CPU core or node).
- A processor may **host multiple processes** (especially in multitasking or time-sharing systems).

Types of Mapping:

Type	Description
Static Mapping	Mapping is decided **before execution** and remains fixed.
Dynamic Mapping	Mapping is decided **at runtime** depending on system load or task completion.

Goal of Mapping:

- Balance the **workload** among processors.
- Minimize **communication overhead** between tasks.
- Reduce **idle time** (ensure all processors are busy).

Example:

Suppose we have **4 tasks** and **2 processors**:

Task	Description	Mapped Processor
T1	Initialize array A	Processor 1
T2	Initialize array B	Processor 2
T3	Compute sum of A	Processor 1
T4	Compute sum of B	Processor 2

If tasks finish at different times, dynamic mapping may reassign remaining tasks to idle processors.

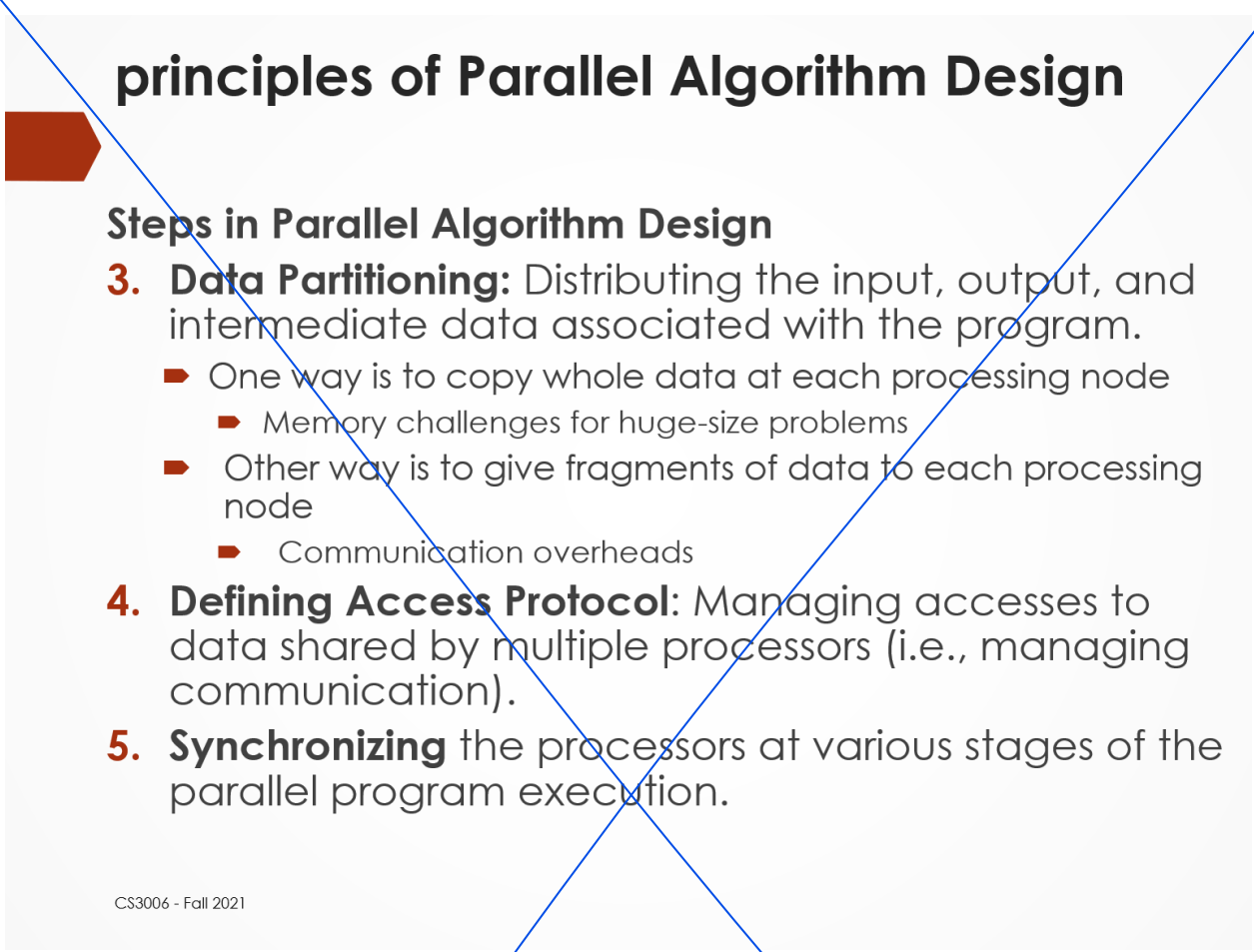
🧭 Summary Table

Step	Description	Goal
Identification	Find independent tasks that can execute in parallel	Maximize concurrency
Mapping	Assign tasks to processes (and processors)	Balance workload, minimize communication

---

🧠 Remember:

- Identification → *"What can be done in parallel?"*
- Mapping → *"Who will do which task?"*



Array sum

Excellent 👏 — you're covering the **core stages of Parallel Algorithm Design** beautifully. Let's now complete your topic with a **clear, structured, and easy-to-understand explanation** of the next steps: **Data Partitioning** and **Defining Access Protocol** (with an example of *Array Sum*).

⚙️ Steps in Parallel Algorithm Design

After **Identification** and **Mapping**, the next important steps are **Data Partitioning** and **Defining Access Protocol** — both are crucial for efficient data handling and synchronization between processors.

🔷 3. Data Partitioning

Definition:

Data Partitioning means **distributing the program's data** — input, output, and intermediate results — among multiple processors.

It ensures that **each processor gets a portion of the data** it needs to work on.

Types of Data Partitioning:

🟢 1. Replicated Data (Whole Copy)

- Each processor gets a **complete copy** of all data.
- Simple but wastes memory.
- Works well for **small datasets**.

📌 **Example:**
If we have an array of 100 elements, and 4 processors:

- Each processor gets the *entire array*.

➡️ **Problem:** For very large arrays (e.g., millions of elements), copying to each node becomes **memory expensive**.

🟣 2. Distributed Data (Fragmented Copy)

- The data is **divided into chunks**, and each processor gets only a **fragment**.
- Saves memory but requires **communication** between processors (to combine results).

📌 **Example:**
Same array of 100 elements → divided among 4 processors:

- P1 → elements 1–25
- P2 → elements 26–50
- P3 → elements 51–75
- P4 → elements 76–100

Each processor works on its fragment.

➡️ **Challenge:** After computation, results must be **combined** (causing communication overhead).

Goal of Data Partitioning:

- Distribute data **evenly** among processors.
- Reduce **memory usage**.
- Minimize **data transfer and communication**.

⚠️ Trade-Offs

Method	Advantage	Disadvantage
Whole Copy	No communication overhead	High memory usage
Fragmented Copy	Efficient memory usage	Requires communication and synchronization

---

🔷 4. Defining Access Protocol

Definition:

When multiple processors need to **access or modify shared data**, we must define clear **access rules (protocols)** to prevent conflicts and ensure correctness.

Goals of Access Protocols:

- Manage **shared data access** safely.
- Prevent **data races** (when two processors modify the same data simultaneously).
- Coordinate **communication** and **synchronization** among processors.

Key Concepts:

🟢 1. Communication Management

Processors must **exchange data** when their results depend on each other.

- In **shared-memory systems:** use shared variables.
- In **distributed-memory systems:** use **message passing** (like MPI send/receive).

🎯 *Goal:* Ensure processors have the right data at the right time.

🔵 2. Synchronization

Processors must coordinate their progress so that **dependent tasks** do not execute prematurely.

- **Barrier synchronization:** All processors must reach a common point before proceeding.
- **Locks/Semaphores:** Used to control access to shared variables.

🎯 *Goal:* Ensure correctness while minimizing idle waiting.

💡 Example: Array Sum

Problem:

Find the **sum of all elements** in a large array using 4 processors.

Step 1: Identification

Each part of the array can be summed independently.
So, the work is **divided into 4 parallel tasks.**

Step 2: Mapping

Each processor (P1–P4) gets one part of the array.

```
Array = [A1, A2, …, A100]
P1 → 1-25
P2 → 26-50
P3 → 51-75
P4 → 76-100
```

Step 3: Data Partitioning

- **Fragmented copy approach**: Each processor holds its own portion of data.
- Each processor computes **partial sum**:
- `S1 = sum(1-25)`
- `S2 = sum(26-50)`
- `S3 = sum(51-75)`

- `S4 = sum(76–100)`

Step 4: Defining Access Protocol

- After local sums are computed, processors must **share their partial sums**.
- One processor (say P1) gathers all partial results and computes:
- `Total Sum = S1 + S2 + S3 + S4`
- **Synchronization point:** All processors must finish their partial sums before combining.

✅ Result:

Each processor does 1/4th of the work → total time reduces significantly.
However, there's a **small communication cost** at the end during the combination step.

🧭 Summary Table

Step	Description	Example (Array Sum)
Identification	Find concurrent tasks	Each portion of array is independent
Mapping	Assign tasks to processors	Each processor handles one portion
Data Partitioning	Divide data among processors	Each gets 25 elements
Access Protocol	Manage communication & synchronization	Combine partial sums safely

---

💬 In short:

- **Data Partitioning** = "Who gets which data?"
- **Access Protocol** = "How do they share and synchronize safely?"

---

principles of Parallel Algorithm Design

- **Decomposition:**
  - The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel.
- ***Tasks***
  - **Programmer-defined units of computation** into which the main computation is subdivided by means of decomposition
  - Tasks can be of **arbitrary size**, but once defined, they are regarded as **indivisible units of computation**.
  - The tasks into which a problem is decomposed may not all be of the same size
  - Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem.

Problem is divided into tasks (chunks), could be of same and different size

We could use task dependency graph to explain decomposition e.g. adding 8 numbers using 4 processors

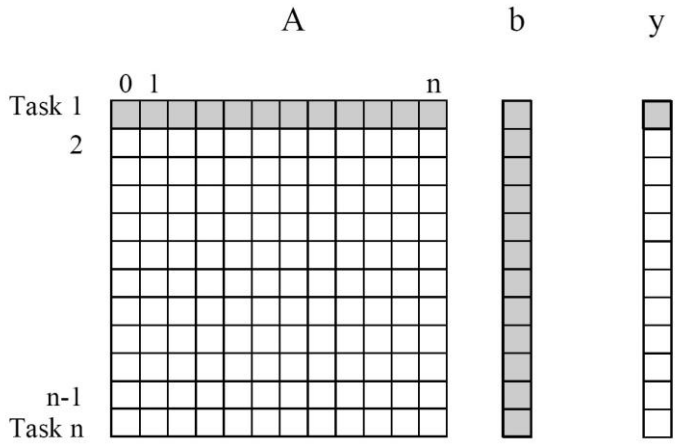


Figure 3.1 Decomposition of dense matrix-vector multiplication into n tasks, where n is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

Computation of each y[i] is regarded as a single task.

Matrix multiplication rule: no. of columns in first matrix A should be equal to no. of rows in second b
Resulting matrix would have the no. of rows equivalent of first matrix and no. of columns of second matrix

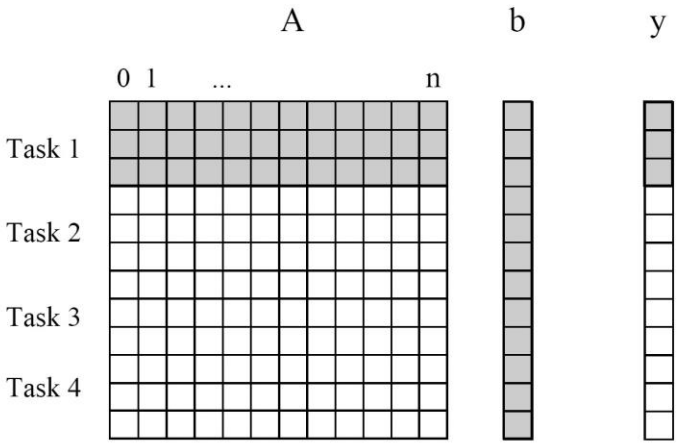


Figure 3.4 Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

Computation of three y[i] entries is regarded as a task

Perfect 👌 — here's a **clear and concise explanation** of your topic "**Decomposition in Parallel Algorithm Design**" that you can use in your notes or slides:

Decomposition in Parallel Algorithm Design

Definition:
Decomposition is the process of **dividing a computation into smaller parts** (subproblems or tasks), some or all of which can be executed **concurrently** on multiple processors.

Key Concepts:

- **Tasks:**
  - These are **programmer-defined units of computation**.

- Each task represents a part of the overall work (e.g., computing a sum, sorting a portion of data, etc.).
- Once defined, a task is considered **indivisible** during execution.
- Tasks may vary in **size and complexity** — some might take longer than others.

Purpose of Decomposition:

- To **identify independent or partially independent computations** that can run in parallel.
- To **increase concurrency**, reducing total execution time.
- To **balance workload** among processors to avoid idle time.

Important Notes:

- The **quality of decomposition** directly affects performance — poor decomposition can lead to uneven load or excessive communication overhead.
- There is often a **trade-off** between having too few large tasks (less concurrency) and too many tiny tasks (high overhead).

Example:

Array Sum

- Suppose we need to find the sum of all elements in a large array.
- The array can be **divided into smaller segments**, and each segment can be summed by a separate task.
- Finally, the partial sums from each task can be combined to get the total sum.

---

principles of Parallel Algorithm Design

Task-Dependency Graph

- The tasks in the previous examples are independent and can be performed in any sequence.
- In most of the problems, there exist some sort of dependencies between the tasks.
- An abstraction used to express such **dependencies** among tasks and their **relative order of execution** is known as a **task-dependency graph**
- It is a **directed acyclic graph** in which node are tasks and the directed edges indicate the dependencies between them
- The task corresponding to a node can be executed when all tasks connected to this node by incoming edges have completed.

Some tasks may use data produced by other tasks and thus may need to wait for these tasks to finish execution

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Table 3.1 A database storing information about used vehicles.

Total records are 10.

Select ID, model, year, color from dbtable where model='civic' and year='2001' and (color='White' OR color='Green')

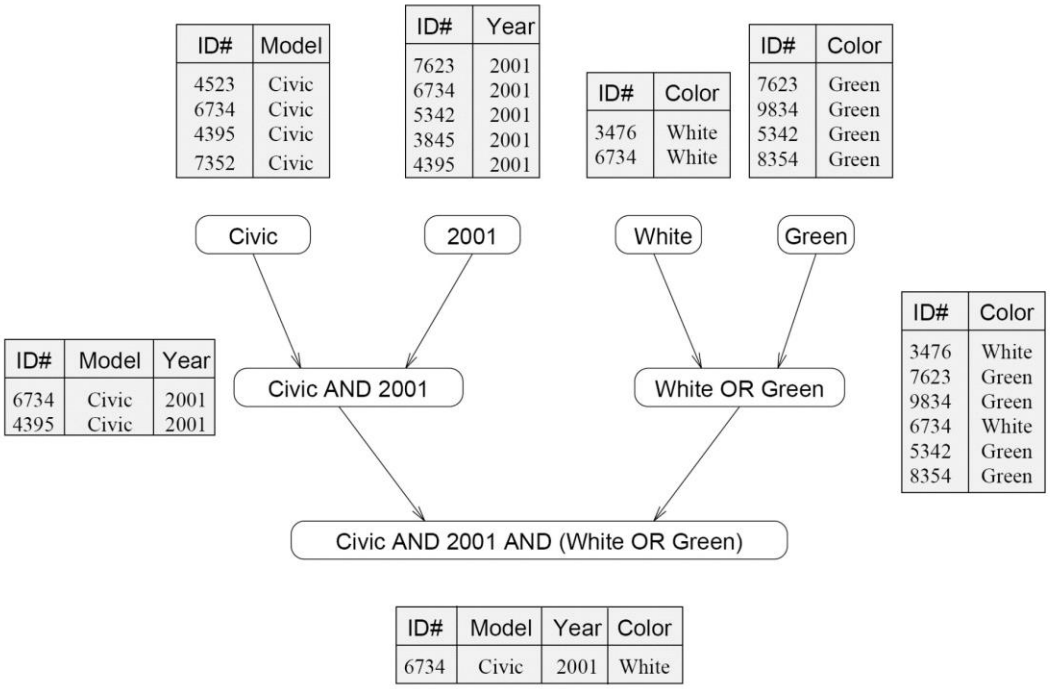


Figure 3.2 The different tables and their dependencies in a query processing operation.

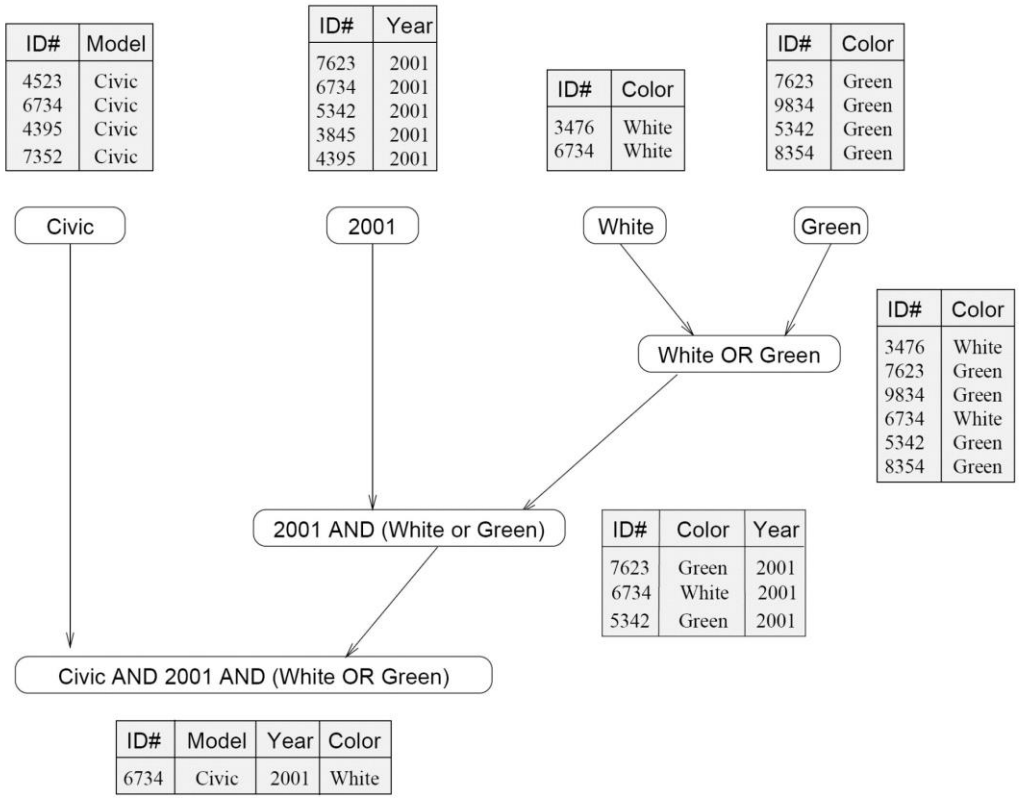


Figure 3.3 An alternate data-dependency graph for the query processing operation.

Perfect — here's your concept on **Task-Dependency Graph** written in a clear, structured, and easy-to-understand format for notes or slides 👇

Task-Dependency Graph

Definition:
A **Task-Dependency Graph (TDG)** is an abstraction used to represent **dependencies** among tasks in a parallel program and to define the **relative order of their execution**.

Key Characteristics:

- It is a **Directed Acyclic Graph (DAG)**.
- **Nodes** represent **tasks**.
- **Directed edges** represent **dependencies** between tasks.
- An edge from Task A → Task B means **Task B cannot start** until **Task A finishes**.

Execution Rule:

A task (node) can only be executed **after all tasks with incoming edges** to it have been completed.

Purpose:

- To identify **parallelism** and **synchronization points**.
- To understand **which tasks can be executed concurrently** and **which must wait**.
- To assist in **scheduling and load balancing** across processors.

Example (Simple Illustration):

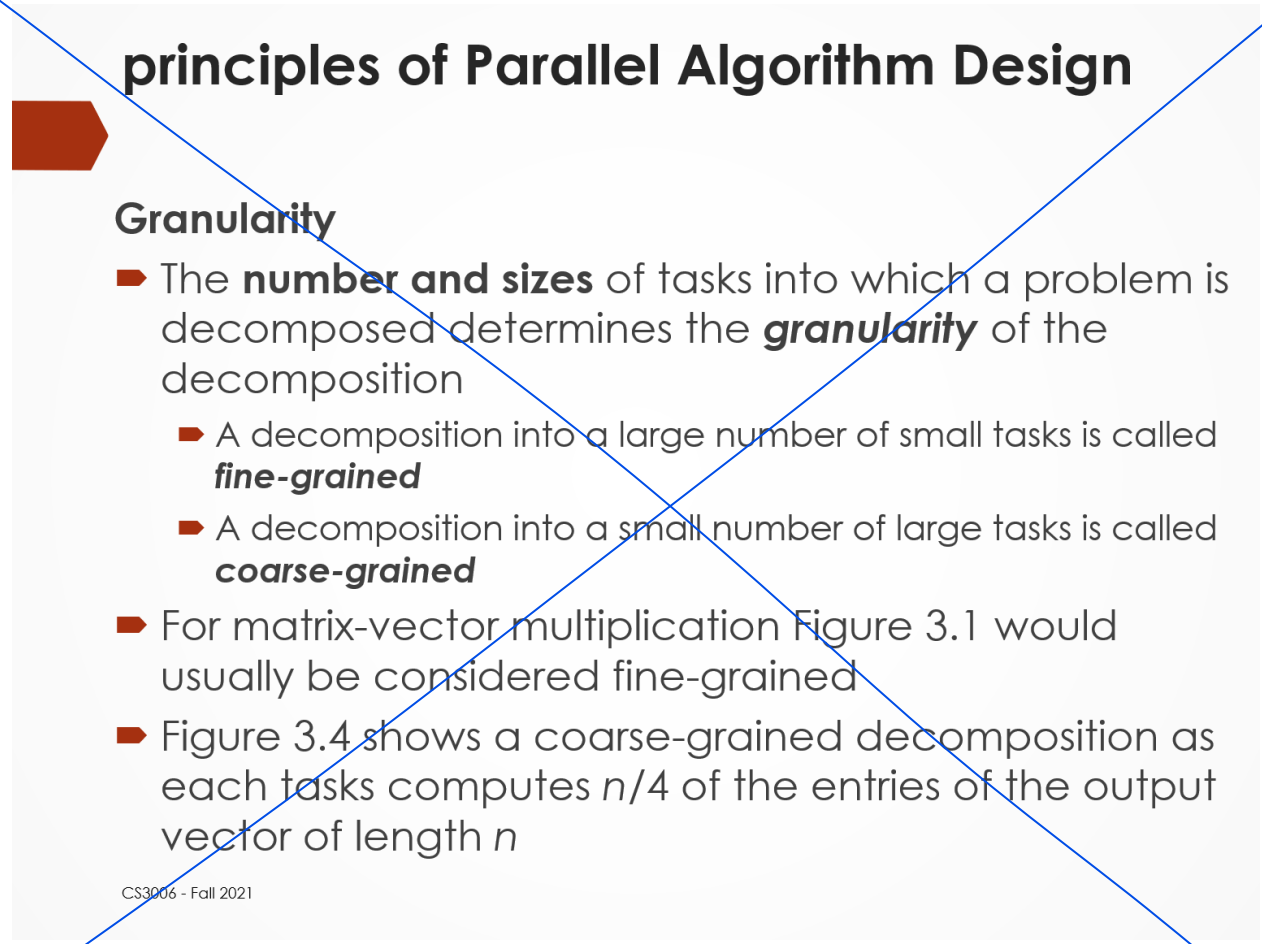
Suppose we have four tasks:
T1 → T2 → T3, and T1 → T4

Explanation:

- T1 must complete before T2 and T4 can start.
- T3 can only begin after T2 is done.
- T4 runs independently after T1 — so T3 and T4 can execute **in parallel** once T1 and T2 finish respectively.

Visualization (Text Representation):

```
   T1
  /  \
 T2   T4
 |
 T3
```



Granularity: roughness (means consisting of small grains or particles)

Go to the previous slides for better illustration

Excellent 👌 — here's your topic on **Granularity in Parallel Algorithm Design**, written clearly and neatly for your study notes or presentation slides:

Granularity

Definition:
The **granularity** of a parallel program refers to the **number and size of tasks** into which a problem is decomposed.
It indicates **how much computation** is performed within a task **before communication** or synchronization occurs.

Types of Granularity:

1. **Fine-Grained Decomposition**
    - The problem is divided into a **large number of small tasks**.
    - Each task performs a **small amount of computation**.
    - Requires **frequent communication** and **synchronization** between processors.
    - Example: Each task computes **a single entry** of a matrix-vector multiplication.
    - ✅ High parallelism
    - ❌ High communication overhead

---

2. **Coarse-Grained Decomposition**
    - The problem is divided into a **small number of large tasks**.
    - Each task performs **a significant portion of the total computation**.
    - **Less communication** and **synchronization overhead**.
    - Example: Each task computes **n/4 entries** of the output vector of length *n* in matrix-vector multiplication.
    - ✅ Low overhead
    - ❌ Less parallelism

Trade-Off:

Choosing the right granularity is a **balance** between:

- **Too fine-grained:** High communication overhead
- **Too coarse-grained:** Poor load balancing and less concurrency

---

In Summary:

Type	Number of Tasks	Task Size	Communication Overhead	Parallelism
Fine-Grained	Large	Small	High	High
Coarse-Grained	Small	Large	Low	Low

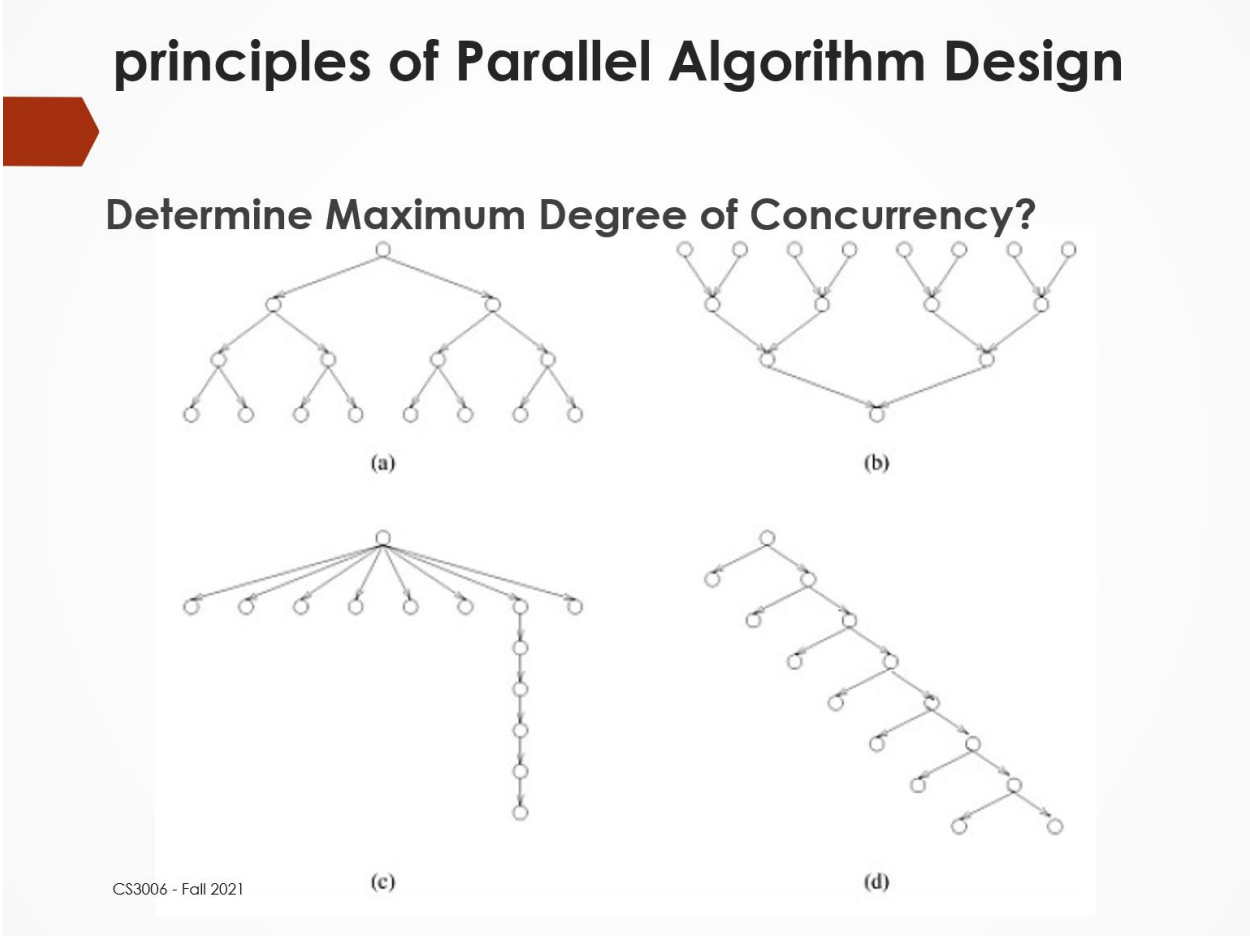
---

principles of Parallel Algorithm Design

Maximum Degree of Concurrency

- The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its *maximum degree of concurrency*
- Usually, it is always less than total number of tasks due to dependencies.
- E.g., max-degree of concurrency in the task-graphs of Figures 3.2 and 3.3 is 4.
- **Rule of thumb:** For task-dependency graphs that are trees, the maximum degree of concurrency **is always equal to the number of leaves in the tree**

Go to the previous slides for better illustration



Perfect 🌟 — here's your topic **Maximum Degree of Concurrency** written in a clear and structured way for your notes or slides:

Maximum Degree of Concurrency

Definition:
The **maximum degree of concurrency** is the **largest number of tasks** that can be executed **simultaneously** at any given point during the execution of a parallel program.

Key Points:

- It indicates the **upper limit** of how many tasks can run in parallel **at once**.
- It depends on the **task-dependency graph (TDG)** — specifically on which tasks are **independent** at a certain stage.
- It is **usually less than the total number of tasks** because some tasks must **wait** for others to finish (due to dependencies).

Example:

If a task-dependency graph has 10 tasks but only 4 of them can run concurrently at any given time,
then the **maximum degree of concurrency = 4**.

Rule of Thumb:

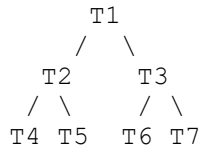
For task-dependency graphs that form **tree structures**,

The **maximum degree of concurrency** is equal to the **number of leaves** in the tree.

Importance:

- Helps in determining **how many processors** can be effectively utilized.
- If the maximum degree of concurrency is small, adding more processors **won't improve** performance significantly.

Example Visualization (Conceptual):



Here, tasks **T4, T5, T6, and T7** are the **leaves**.
➡️ **Maximum degree of concurrency = 4**

Would you like me to explain the next concept — **Critical Path Length** — which is directly related to this one?

principles of Parallel Algorithm Design

Average Degree of Concurrency

- A relatively better measure for the performance of a parallel program
- The average number of tasks that can run concurrently over the entire duration of execution of the program
- The ratio of the ***total amount of work*** to the ***critical-path length***
  - So, what is the critical path in the graph**?**

principles of Parallel Algorithm Design

Average Degree of Concurrency

- **Critical Path:** The longest directed path between any pair of start and finish nodes is known as the critical path.
- **Critical Path Length:** The sum of the weights of nodes along this path
  - the weight of a node is the size or the amount of work associated with the corresponding task.
- A shorter critical path favors a higher average-degree of concurrency.
- Both, maximum and average degree of concurrency increases as tasks become smaller(finer)

principles of Parallel Algorithm Design

Average Degree of Concurrency

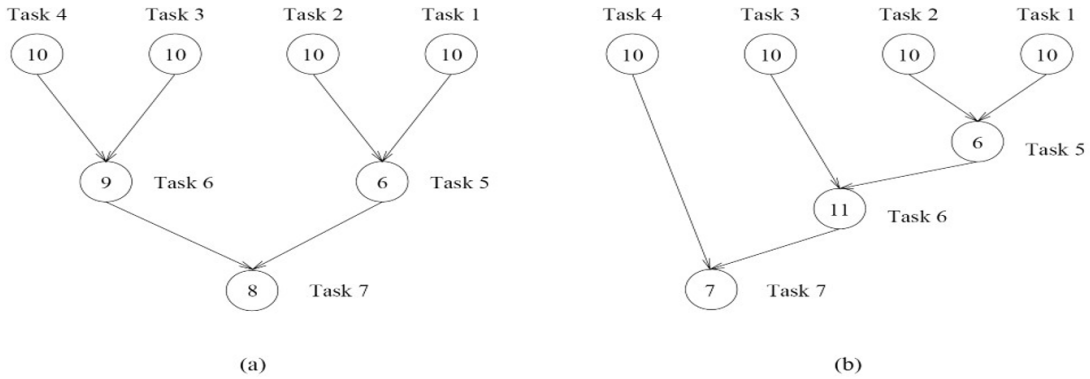


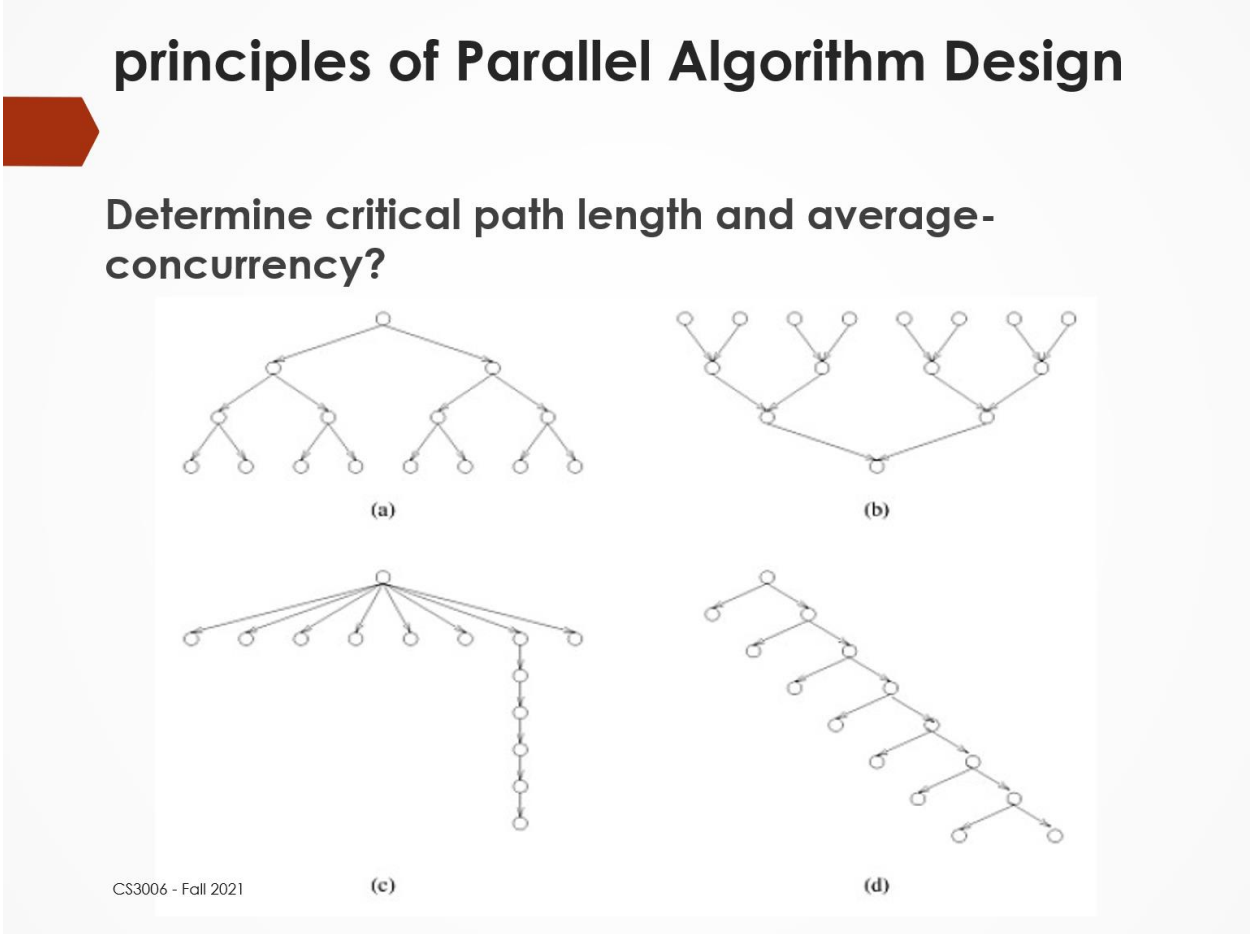
Figure 3.5 Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.

Critical path lengths: 27 and 34

Total amount of work: 63 and 64

Average degree of concurrency: 2.33 and 1.88

63/27. Although max degree of concurrency is same (40 in both cases), average degree is different. So, shape of graph also effects degree of concurrency



Critical path length=sum of weights along critical path= 4,4,7,8

Average concurrency= 15/4,15/4, 2, 15/8

Excellent 👍 — here's a clean and complete explanation combining both **Critical Path** and **Average Degree of Concurrency**, written perfectly for your notes or presentation:

Critical Path and Average Degree of Concurrency

---

1. Critical Path

Definition:
The **Critical Path** in a task-dependency graph is the **longest directed path** between any **start node** and **finish node**.

Critical Path Length:

- It is the **sum of the weights** (amount of work or execution time) of all tasks along the critical path.
- The **weight of a node** represents the **size or computation time** of that task.

Meaning:

- The **critical path length** represents the **minimum possible time** required to execute the program — even with an **infinite number of processors**.
- Tasks on this path **cannot be parallelized** further; they must be executed **sequentially**.

Example:
If a dependency graph has tasks where:

- Path 1 (T1 → T2 → T3) takes total time = 10
- Path 2 (T1 → T4) takes total time = 7

Then, the **critical path length = 10**, because it's the **longest**.

---

2. Average Degree of Concurrency

Definition:
The **Average Degree of Concurrency (ADC)** represents the **average number of tasks** that can be executed **simultaneously** throughout the entire execution of the program.

Formula:
[
\text{Average Degree of Concurrency} = \frac{\text{Total Amount of Work}}{\text{Critical Path Length}}
]

Where:

- **Total Amount of Work** = Sum of execution times (weights) of all tasks in the graph
- **Critical Path Length** = Time taken along the longest path (minimum sequential time)

Interpretation:

- A **larger value** of ADC means **better parallel performance**.
- A **shorter critical path** or **finer-grained tasks** increases the average concurrency.

Key Relationships:

Term	Meaning
Maximum Degree of Concurrency	Maximum number of tasks that can run at once
Critical Path Length	Minimum sequential execution time
Average Degree of Concurrency	Average number of concurrent tasks = total work ÷ critical path length

---

Observation:

- As **tasks become smaller (fine-grained)** → both **maximum** and **average** degree of concurrency **increase**.
- A **shorter critical path** → higher **average concurrency** → better performance.

---

principles of Parallel Algorithm Design

Task Interact Graph

- Depicts pattern of interaction between the tasks
- Dependency graphs only show that how output of first task becomes input to the next level task.
- But how the tasks interact with each other to access distributed data is only depicted by task interaction graphs
- The nodes in a task-interaction graph represent tasks
- The edges connect tasks that interact with each other

For example, in dense matrix-vector diagram, all the tasks can be shown independent in dependency graph. Since originally there is only one copy of the vector *b*, tasks may have to send and receive messages for all of them to access the entire vector in the distributed-memory paradigm

principles of Parallel Algorithm Design

Task Interact Graph

- The edges in a task interaction graph are usually **undirected**
  - but directed edges can be used to indicate the direction of flow of data, if it is unidirectional.
- The edge-set of a task-interaction graph is usually a **superset** of the edge-set of the task-dependency graph
- In database query processing example, the task-interaction graph is the **same** as the task-dependency graph.

For example, all the tasks can be shown independent in dependency graph of matrix-vector multiplication. Since originally there is only one copy of the vector *b*, tasks may have to send and receive messages for all of them to access the entire vector in the distributed-memory paradigm

principles of Parallel Algorithm Design

Task Interact Graph (Sparse-matrix multiplication)

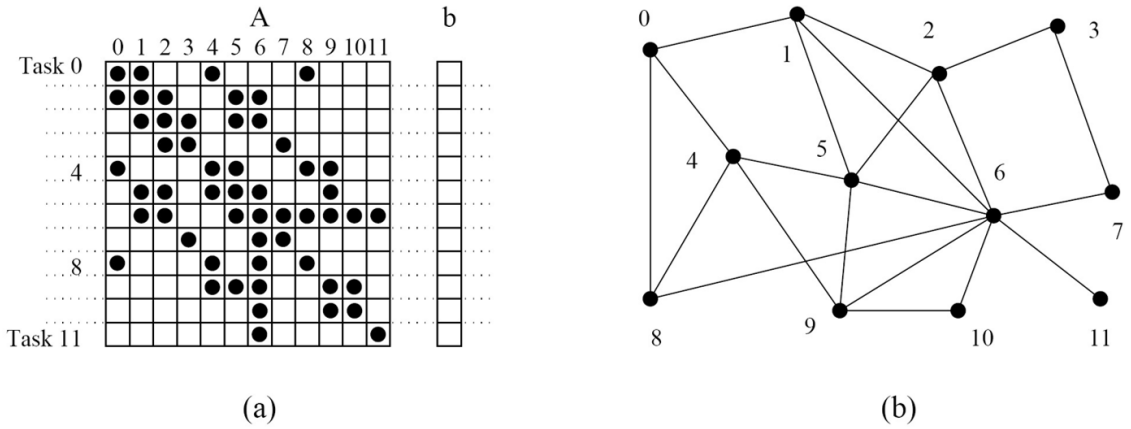


Figure 3.6 A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \le j \le 11, A[i,j] \neq 0} A[i, j].b[j]$.

Each task is assigned computation of single element of output vector y;

Furthermore, we make task(i) the owner of row A[i,*] and input vector element b[i];
In simple words:
Tasks: each task computes an entry of y[]
Assign ith row of A to Task i. Also assign b[i] to Task i.

Perfect 👏 — here's your topic **Task-Interaction Graph** rewritten clearly, concisely, and structured for easy learning or slide presentation:

Task-Interaction Graph

Definition:

A **Task-Interaction Graph (TIG)** is used to depict the **pattern of interaction** (data exchange or communication) **between tasks** in a parallel program.

It focuses on **how tasks communicate and share data**, rather than just how they depend on each other.

Difference from Task-Dependency Graph:

Aspect	Task-Dependency Graph	Task-Interaction Graph
Purpose	Shows **precedence and dependency** (which task must finish before another starts).	Shows **communication and data exchange** between tasks.
Edges	Directed edges (represent task order).	Usually **undirected edges** (represent communication).
Focus	Execution order.	Data interaction pattern.
Edge Set	Subset.	Usually a **superset** of dependency graph edges.

Structure:

- **Nodes:** Represent **tasks**.
- **Edges:** Represent **communication or interaction** between tasks.
  - Usually **undirected**, meaning communication is two-way.
  - Can be **directed** if communication is **one-way** (unidirectional data flow).

Example 1 — Dense Matrix-Vector Multiplication:

- In a dependency graph:
  Each task (computing one element of the result vector) appears **independent** — no dependencies.
- In a **task-interaction graph**:
  Since there is **only one copy of vector `b`**, each task must **send and receive** parts of `b` to access it completely.
  → This shows **interaction edges** between all tasks that share `b`.

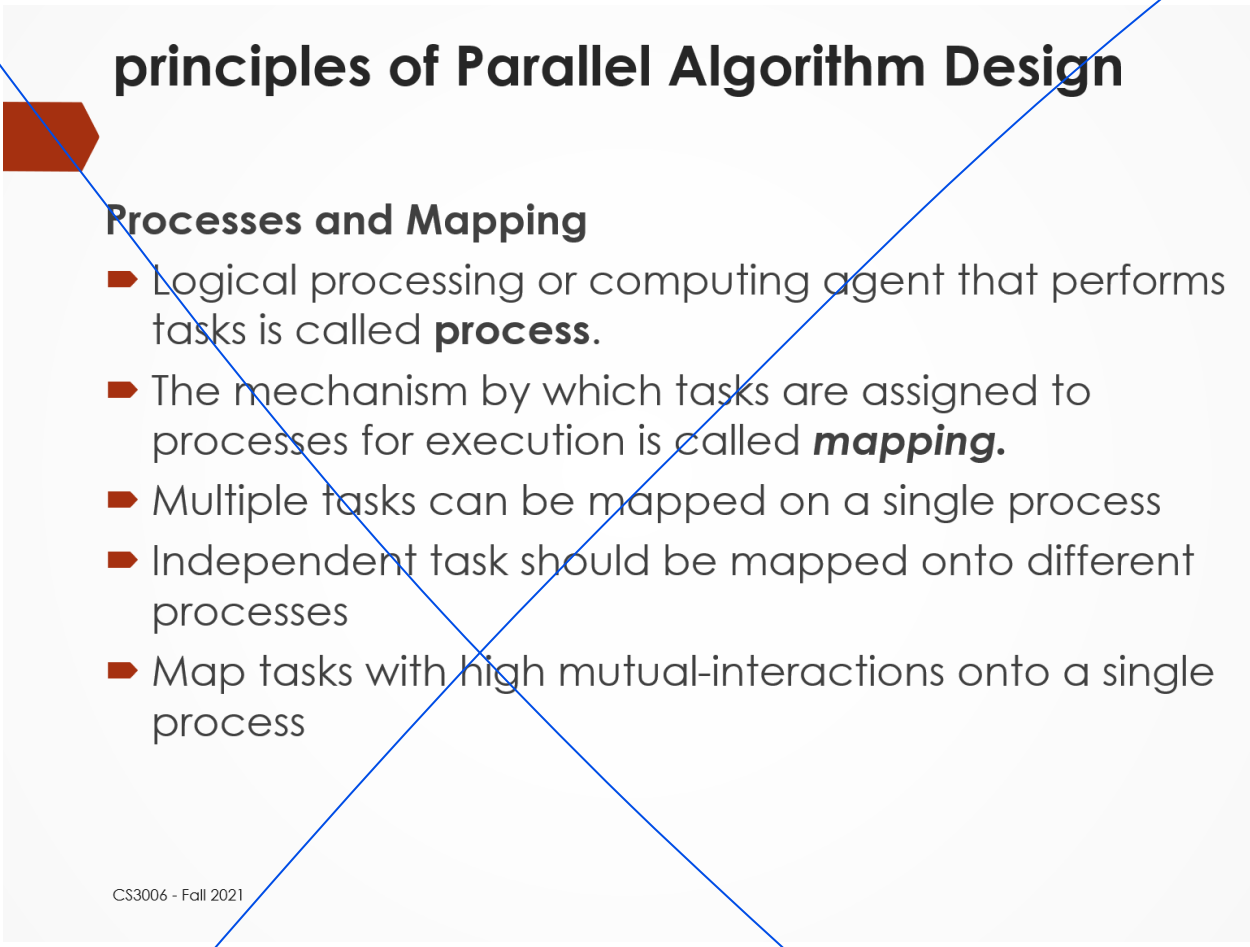
Example 2 — Database Query Processing:

- In some cases, such as a **database query**, the **task-interaction graph** and the **task-dependency graph** may be **identical**, because dependencies and interactions overlap.

Key Takeaways:

- TIGs reveal **communication overheads** and **data-sharing complexity**.
- They help in designing efficient **data distribution** and **communication strategies**.
- The **more edges** a TIG has, the **higher** the potential **communication cost** in parallel execution.

Would you like me to move next to **"Processes and Mapping"** — which is the next part in the Parallel Algorithm Design topic?



A parallel program must have several processes active and simultaneously working on different tasks to gain a significant speedup over the sequential program.

principles of Parallel Algorithm Design

Processes and Mapping

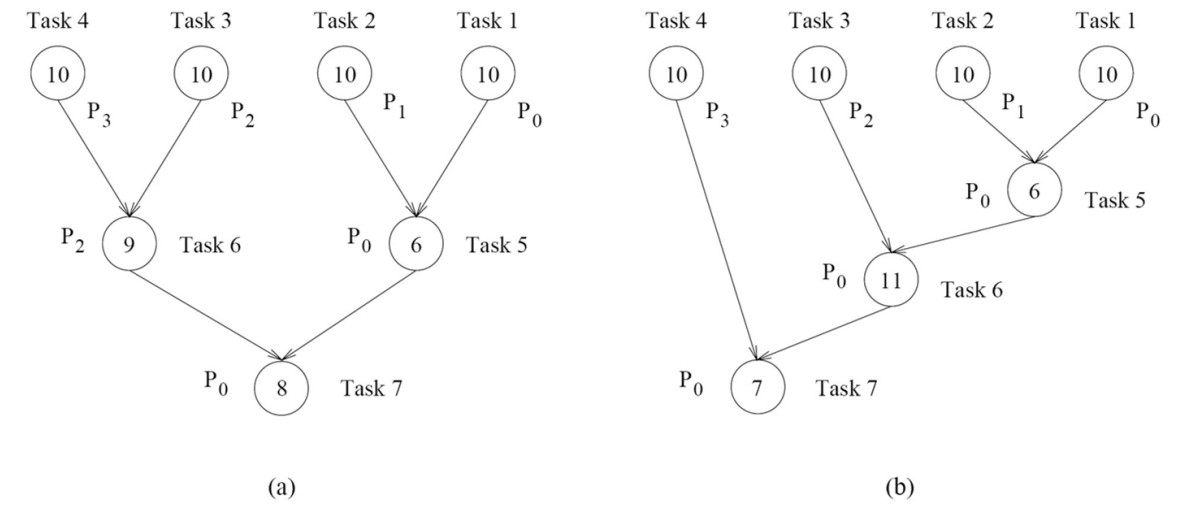
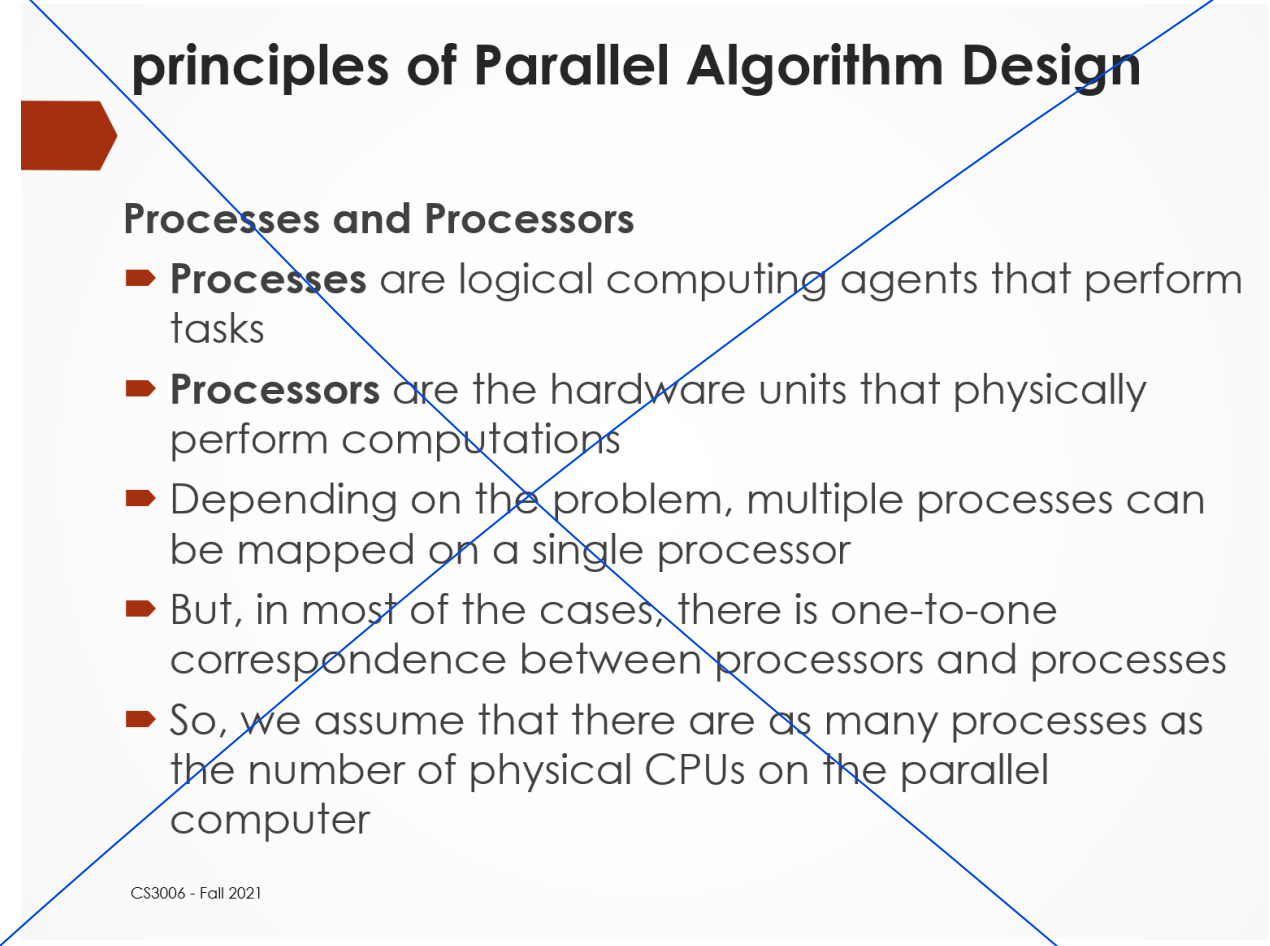


Figure 3.7 Mappings of the task graphs of Figure 3.5 onto four processes.

What if in 3.7 (b), Task 5 is assigned to P2?



Excellent 👌 — here's a well-structured, complete explanation of **Processes and Mapping** (including the distinction between processes and processors), written clearly for your notes or slides:

Processes and Mapping

---

1. Process

- A **process** is a **logical computing agent** responsible for performing one or more **tasks** in a parallel program.
- It represents a **software unit of execution** that carries out computations and communicates with other processes.

---

2. Mapping

Definition:
The mechanism by which **tasks are assigned to processes** for execution is called **mapping**.

Mapping Guidelines:

1. **Independent tasks** should be **mapped onto different processes** so they can execute **concurrently**.
2. **Tasks with high interaction or communication** should be **mapped onto the same process** to **reduce communication overhead**.
3. **Multiple small tasks** can be assigned to a **single process** to balance workload.

Purpose of Mapping:

- To **maximize parallelism**.
- To **minimize communication costs** between processes.
- To **balance the computational load** across all available resources.

---

3. Processes and Processors

Concept	Meaning
Process	A logical or software-based computing agent that performs tasks.
Processor	A physical hardware unit (like a CPU core) that executes instructions.

Relationship Between Processes and Processors:

- Multiple **processes** can be mapped onto a **single processor**, especially if the number of tasks exceeds available hardware units.
- However, in most practical parallel systems, a **one-to-one correspondence** is assumed between **processes** and **processors** for simplicity.

[
\text{No. of Processes} = \text{No. of Physical CPUs (Processors)}
]

---

4. Key Point:

A parallel program achieves **significant speedup** only when **multiple processes** are **actively executing different tasks simultaneously**.

Summary Table

Concept	Description
Process	Logical computing unit performing tasks
Processor	Physical hardware executing computations
Mapping	Assigning tasks to processes
Goal of Mapping	Balance workload and reduce communication
Rule of Thumb	Map dependent or interacting tasks close together

---

Would you like me to continue next with the topic **"Static vs Dynamic Interconnections and Network Topologies"** (the next logical section after this)?