HIGH PERFORMANCE COMPUTING ARCHITECTURE

MODULE 1: FUNDAMENTALS of COMPUTER DESIGN

Introduction;
Classes computers;
Defining computer architecture;
Trends in Technology;
Trends in power in Integrated Circuits;
Trends in cost;
Dependability,
Measuring, reporting and summarizing Performance attributes;
Quantitative Principles of computer design

High Performance because of Improvements in:
- Semiconductor technology (clock speed,...)
- Computer architecture (HLL compilers, UNIX, RISC)
- Single processor to Multi processors
- Single processor performance improvement ended in 2003

Parallelism (Data, Task)

1) Instruction Level Parallelism
- Exploits DLP
- Eg: Pipelining, Speculative Execution

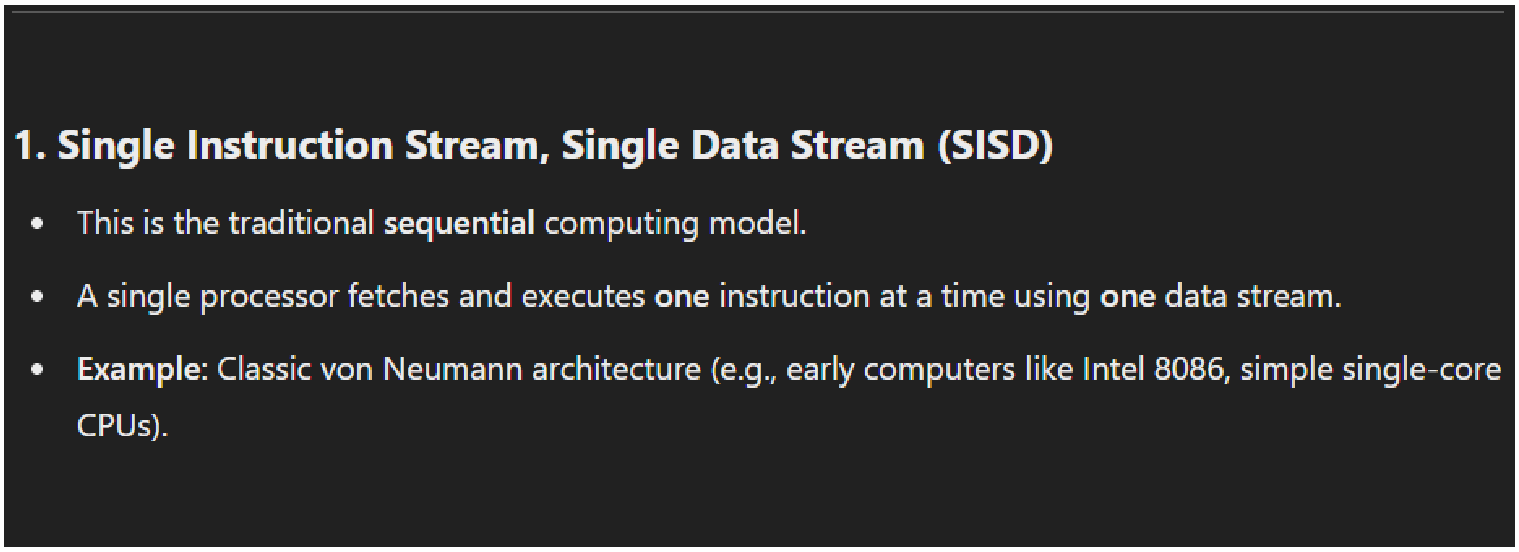
2) Graphic Processor Units
- Exploits DLP
- Eg: Applying single instruction to a collection of data in parallel

3) Thread Level Parallelism
- Exploits DLP or TLP
- Eg: In tightly couples hardware model

4) Request Level Parallelism
- Exploits parallelism in largely decoupled tasks

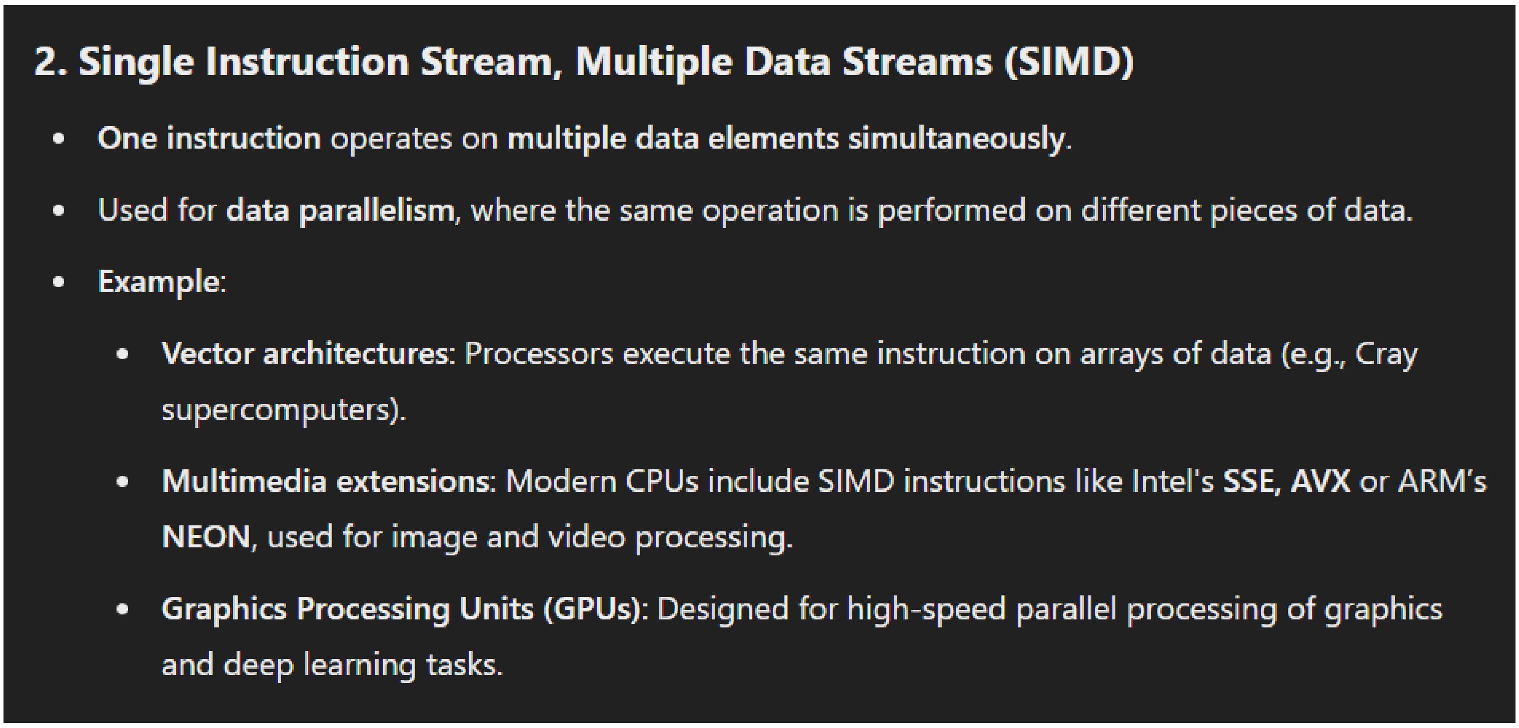
Flynn's Taxonomy (Single and Multiple, Instruction and Data)

Classification system for computer architectures based on how Instructions and Data are processed



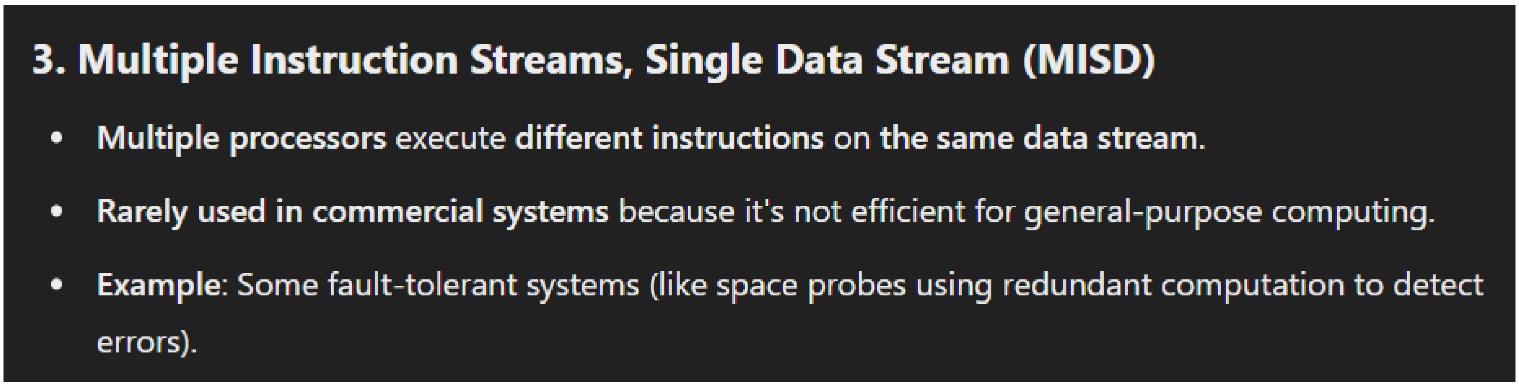
1. Single Instruction Stream, Single Data Stream (SISD)

- This is the traditional **sequential** computing model.
- A single processor fetches and executes **one** instruction at a time using **one** data stream.
- **Example**: Classic von Neumann architecture (e.g., early computers like Intel 8086, simple single-core CPUs).



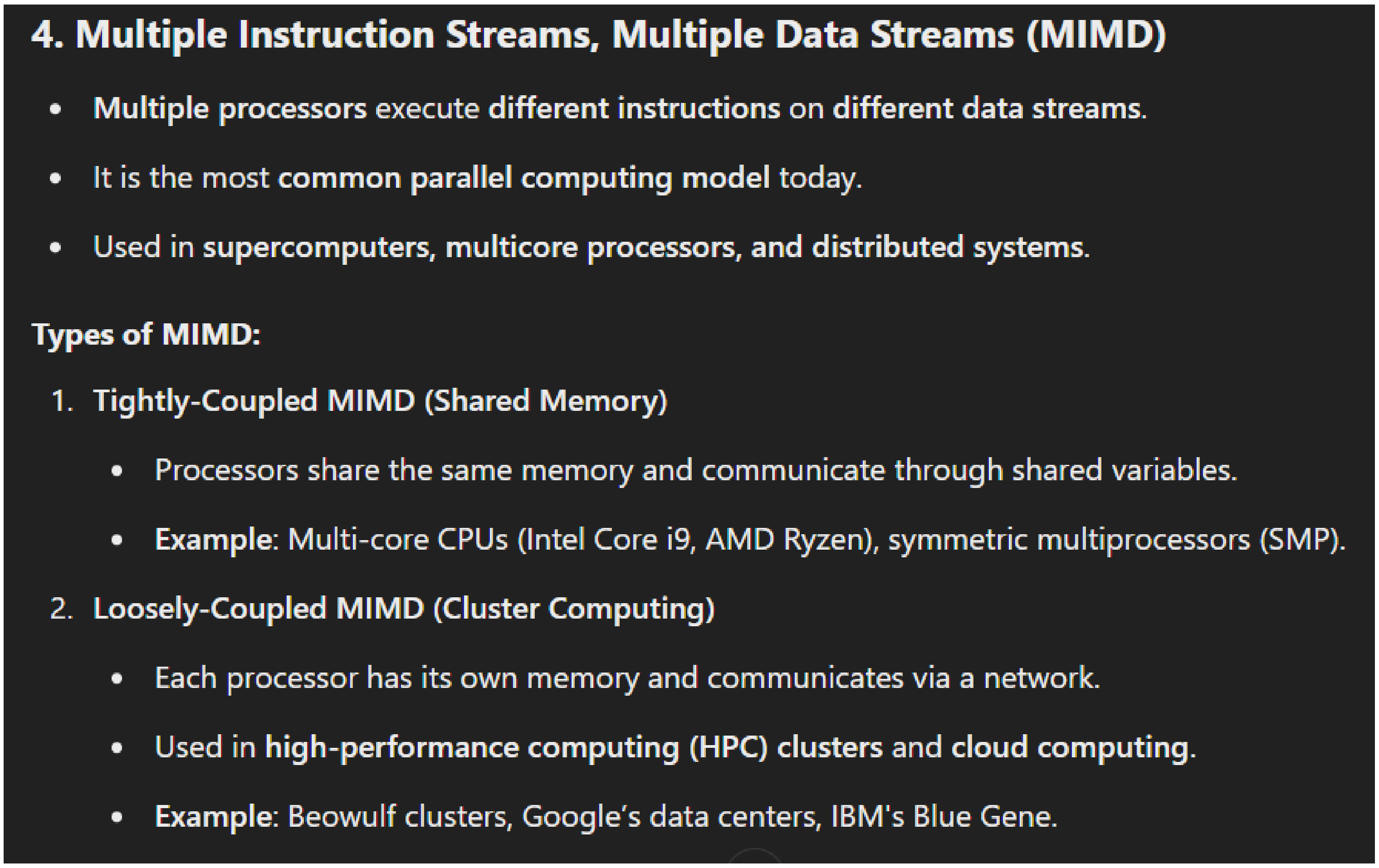
2. Single Instruction Stream, Multiple Data Streams (SIMD)

- **One instruction** operates on **multiple data elements simultaneously**.
- Used for **data parallelism**, where the same operation is performed on different pieces of data.
- **Example**:
  - **Vector architectures**: Processors execute the same instruction on arrays of data (e.g., Cray supercomputers).
  - **Multimedia extensions**: Modern CPUs include SIMD instructions like Intel's **SSE, AVX** or ARM's **NEON**, used for image and video processing.
  - **Graphics Processing Units (GPUs)**: Designed for high-speed parallel processing of graphics and deep learning tasks.



3. Multiple Instruction Streams, Single Data Stream (MISD)

- **Multiple processors** execute **different instructions** on **the same data stream**.
- **Rarely used in commercial systems** because it's not efficient for general-purpose computing.
- **Example**: Some fault-tolerant systems (like space probes using redundant computation to detect errors).

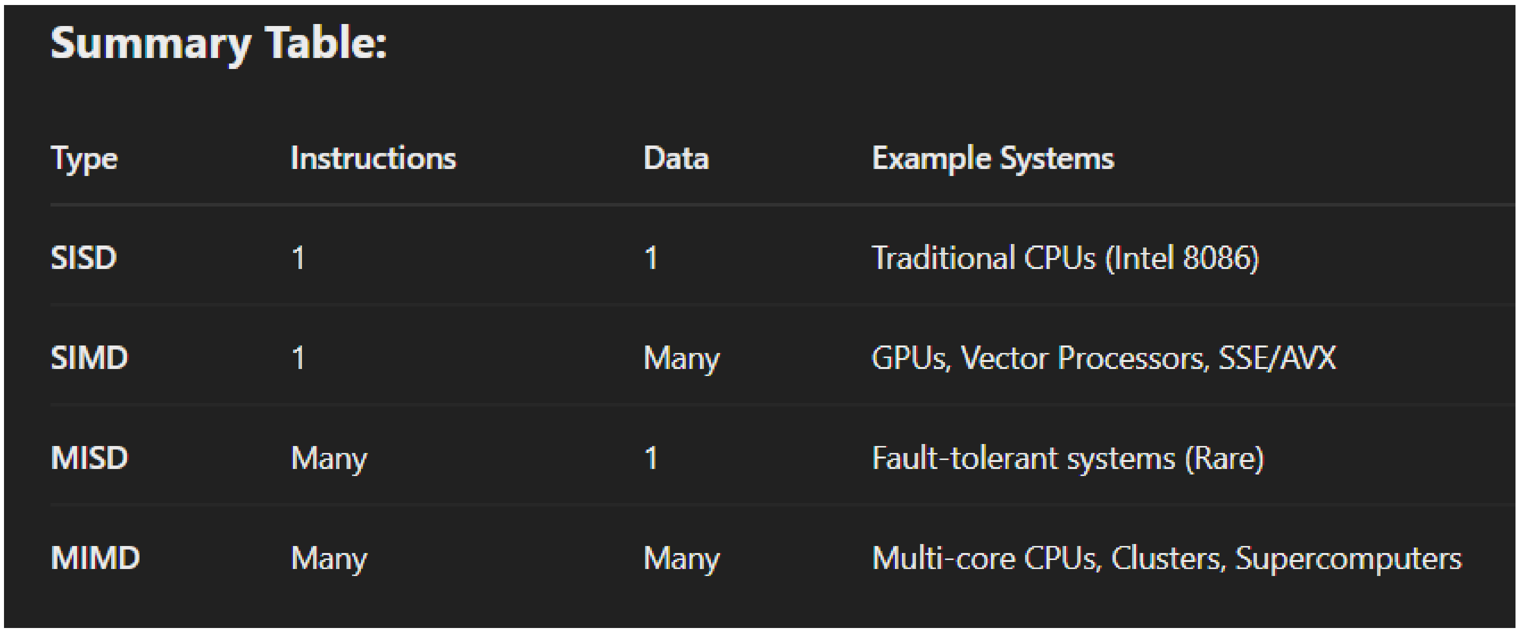


4. Multiple Instruction Streams, Multiple Data Streams (MIMD)

- **Multiple processors** execute **different instructions** on **different data streams**.
- It is the most **common parallel computing model** today.
- Used in **supercomputers, multicore processors, and distributed systems**.

Types of MIMD:

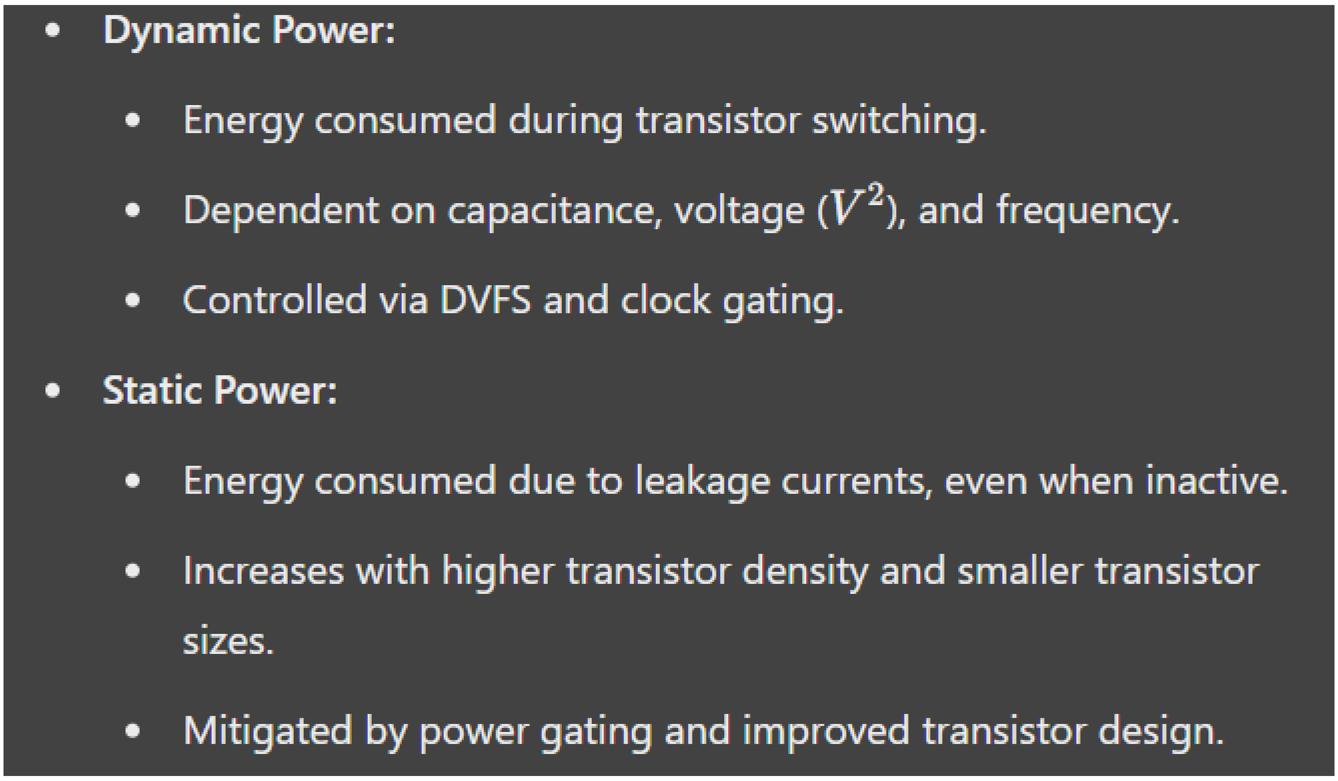
1. **Tightly-Coupled MIMD (Shared Memory)**
   - Processors share the same memory and communicate through shared variables.
   - **Example**: Multi-core CPUs (Intel Core i9, AMD Ryzen), symmetric multiprocessors (SMP).
2. **Loosely-Coupled MIMD (Cluster Computing)**
   - Each processor has its own memory and communicates via a network.
   - Used in **high-performance computing (HPC) clusters** and **cloud computing**.
   - **Example**: Beowulf clusters, Google's data centers, IBM's Blue Gene.



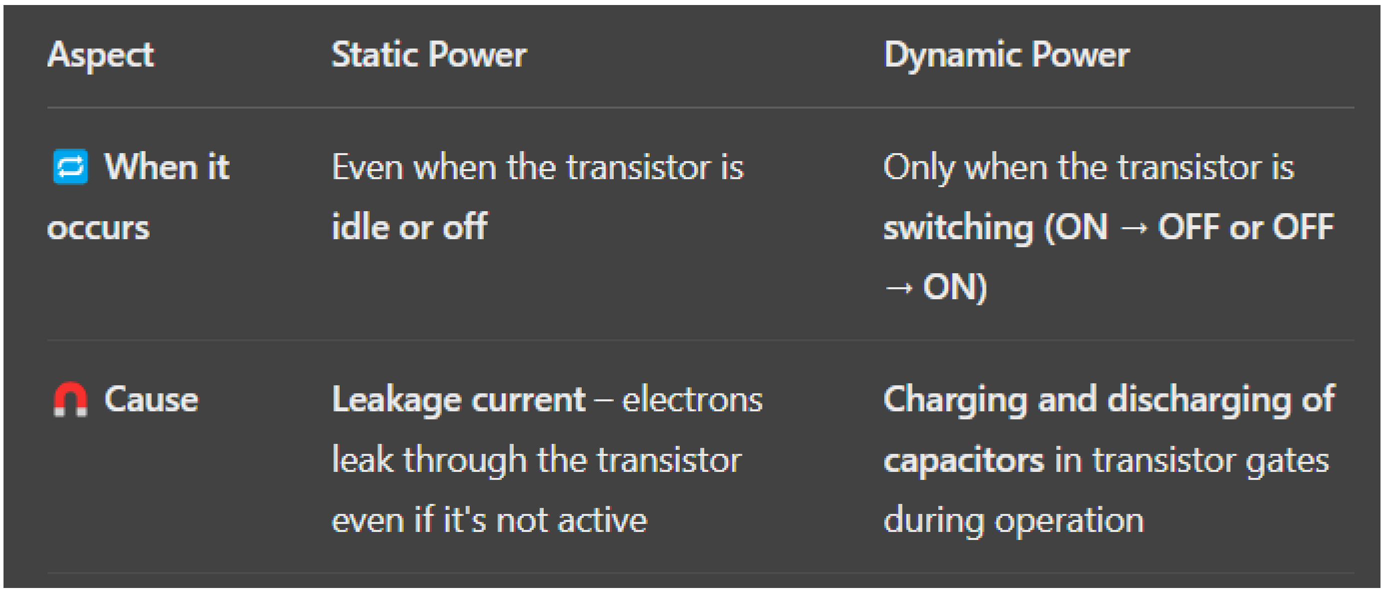
Summary Table:

Type	Instructions	Data	Example Systems
SISD	1	1	Traditional CPUs (Intel 8086)
SIMD	1	Many	GPUs, Vector Processors, SSE/AVX
MISD	Many	1	Fault-tolerant systems (Rare)
MIMD	Many	Many	Multi-core CPUs, Clusters, Supercomputers

Difference Between Dynamic and Static Power



- **Dynamic Power:**
  - Energy consumed during transistor switching.
  - Dependent on capacitance, voltage (V^2), and frequency.
  - Controlled via DVFS and clock gating.
- **Static Power:**
  - Energy consumed due to leakage currents, even when inactive.
  - Increases with higher transistor density and smaller transistor sizes.
  - Mitigated by power gating and improved transistor design.



Aspect	Static Power	Dynamic Power
🔁 When it occurs	Even when the transistor is idle or off	Only when the transistor is switching (ON → OFF or OFF → ON)
🧲 Cause	**Leakage current** – electrons leak through the transistor even if it's not active	**Charging and discharging of capacitors** in transistor gates during operation

Reducing Power Consumption

Modern microprocessors face the challenge related to power consumption

- Do nothing well
- Dynamic Voltage-Frequency Scaling
- Low power state for (DRAM, Disks)
- Overclocking, turn off cores

1. **Do Nothing Well:**
   - Microprocessors save energy by **turning off inactive components**.
   - For example, if no floating-point operations are running, the floating-point unit is disabled.
   - If some CPU cores are idle, their clocks are stopped to reduce power consumption.
2. **Dynamic Voltage-Frequency Scaling (DVFS):**
   - Adjusts voltage and clock frequency based on workload.
   - Devices like **mobile phones, laptops, and servers** lower their clock speed during periods of low activity.
   - Lowering the clock from **2.4 GHz → 1.8 GHz → 1 GHz** can reduce power usage by **10%-15% per step**.

3. **Design for Typical Case:**
   - **Power-saving modes** in memory (DRAM) and storage (hard disks) reduce energy usage in idle states.
   - However, accessing DRAM or disks requires returning to full-power mode.
   - **Emergency slowdown mechanisms** (thermal throttling) reduce CPU speed when overheating is detected.
4. **Overclocking (Turbo Mode):**
   - CPUs can temporarily **increase their clock speed** if temperature allows.
   - Example: A **3.3 GHz Core i7** can boost to **3.6 GHz** for short bursts.
   - For **single-threaded tasks**, all other cores may shut down so one core runs at an even higher speed.
   - Turbo mode can cause **unpredictable performance changes** based on room temperature.

📌 **Techniques for Energy Efficiency**
✅ **Do Nothing Well:** Turn off unused components (floating-point unit, idle cores).
✅ **DVFS (Dynamic Voltage-Frequency Scaling):** Adjusts voltage and frequency based on workload to save power.
✅ **Design for Typical Case:** DRAM and storage use power-saving modes; CPUs use **thermal throttling** for overheating protection.
✅ **Overclocking (Turbo Mode):** CPUs boost performance temporarily if temperature permits.

Defining Computer Architecture

“Old” view of CA
- Instruction Set Architecture (ISA) design

“Real” view of CA
- ISA, Microarchitecure, Hardware
- Design to maximize performance within constraints (C, P, A)

Trends in Technology
[ISA to be successful, design it to survive rapid changes]:

1) Integrated Circuit technology
2) DRAM Capacity
3) Flash Capacity
4) Magnetic Disk technology
5) Network technology

- Bandwidth (throughput) and Latency
- Transistors and Wires

Trends in Power and Energy in Integrated Circuits

- Thermal Design Power (sustained power consumption)
- Clock rate can be reduced to limit power consumption

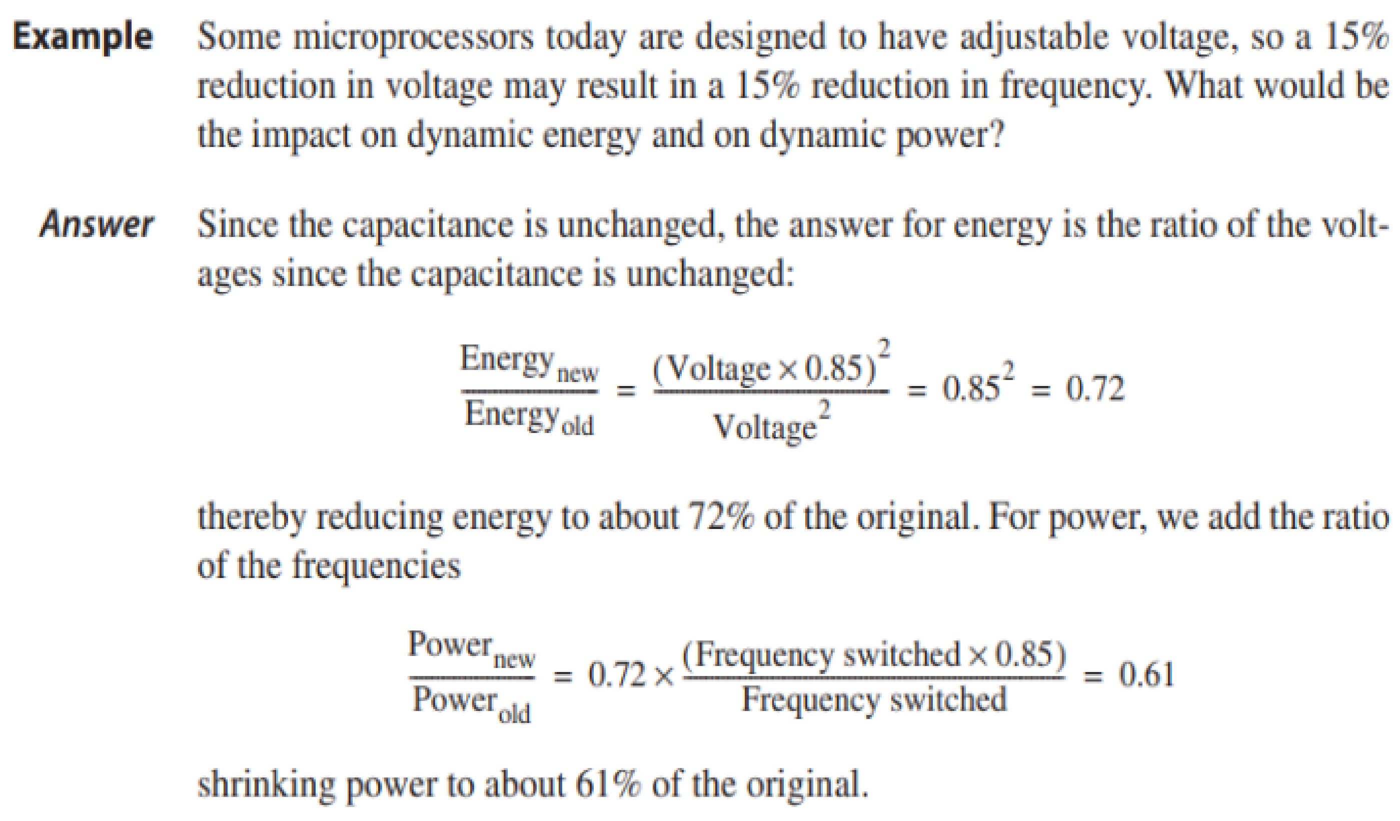
[]
Dynamic Energy = 1/2 x Capacitative Load x (Voltage)^2
Dynamic Power = Frequency switched x Dynamic Energy

# Reducing Clock Rate reduces power, not energy

Processor A = 120% power consumption, Time A = 70%
Processor B = 100% power consumption, Time B = 100%

Energy A = 1.2 x 0.7 = 84%
Energy B = 1.0 x 1.0 = 100%

#



Example Some microprocessors today are designed to have adjustable voltage, so a 15% reduction in voltage may result in a 15% reduction in frequency. What would be the impact on dynamic energy and on dynamic power?

Answer Since the capacitance is unchanged, the answer for energy is the ratio of the voltages since the capacitance is unchanged:

$$\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2}{\text{Voltage}^2} = 0.85^2 = 0.72$$

thereby reducing energy to about 72% of the original. For power, we add the ratio of the frequencies

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = 0.72 \times \frac{(\text{Frequency switched} \times 0.85)}{\text{Frequency switched}} = 0.61$$

shrinking power to about 61% of the original.

Trends in Cost

Dependability

MTTF
MTTR
MTBF
Availability = 1/3

Measuring, Reporting, Summarizing PERFORMANCE

Performance metrics: Throughput, Response Time

Execution Time:
- Wall Clock time
- CPU time

Benchmarks:
- Kernels
- Toy Programs
- Benchmark suites

Quantitative Principles of Computer Design

1) Take Advantage of Parallelism
2) Principle of Locality
3) Focus on Common Case
4) Amdahl's Law

1) Take Advantage of Parallelism

Performance can be significantly improved by using parallelism. Dividing tasks into smaller units that can be executed simultaneously.

System-Level Parallelism:

Performance can be enhanced by using multiple processors and disks to handle tasks concurrently.

Scalability, the ability to expand memory, processors, and disks, is crucial for improving server performance.

Processor-Level Parallelism:

On individual processors, parallelism is achieved through techniques like pipelining. Pipelining overlaps the execution of instructions to reduce overall execution time, enabling instruction-level parallelism.

Digital Design Parallelism:

At the hardware level, parallelism is utilized in designs such as set-associative caches, where memory banks are searched in parallel, and carry-lookahead in ALUs, which speeds up arithmetic operations by reducing the time complexity from linear to logarithmic.

2) Principle of Locality

It states that programs tend to reuse data and instructions they have accessed recently.

This principle is based on two key observations:
temporal locality and **spatial locality**

Temporal locality suggests that recently accessed items are likely to be accessed again in the near future

Spatial locality indicates that items located near each other in memory tend to be accessed together.

By exploiting these patterns, systems can predict which data and instructions will be needed soon, enabling efficient memory usage and speeding up execution through techniques like caching.

3) Focus on the Common Case

In computer design, determining what the frequent case is and optimizing for it can significantly improve performance and resource efficiency.

This principle applies to resource allocation, power management, and system dependability.

For example:

- In a processor, the instruction fetch and decode unit is used much more frequently than a multiplier, so it should be optimized
- When adding numbers, overflow is a rare occurrence, so performance can be improved by optimizing for the case with no overflow.

Amdahl’s Law helps quantify the benefits of this approach by showing how much performance can be improved by making the common case faster.

4) Amdahl’s Law

Performance gain obtained by improving some portions of the computer can be calculated using Amdahl’s Law

Speedup = Execution time without Enhancement / with Enhancement

Speedup = Time to execute a program in serial (one processor) /
Time to execute a program in parallel (multi processor)

There are strictly serial part of the program = B x T(1)
Then the strictly parallel part of the program = ((1-B) x T(1)) / N

Speedup = N / [(BN) + (1-B)]

# Processor Performance Equation

CPU Time = Clock Cycles for Program x Clock Cycle Time
where,
Clock Cycle Time = 1 / Clock Rate
Clock Cycles for Program = Instructions Count x Cycles per Inst (CPI)

so new,
CPU Time = Clock Cycles for Program / Clock Rate
CPU Time = IC x CPI x CCT
CPU Time = Seconds/Programs

- Different instruction types having different CPIs

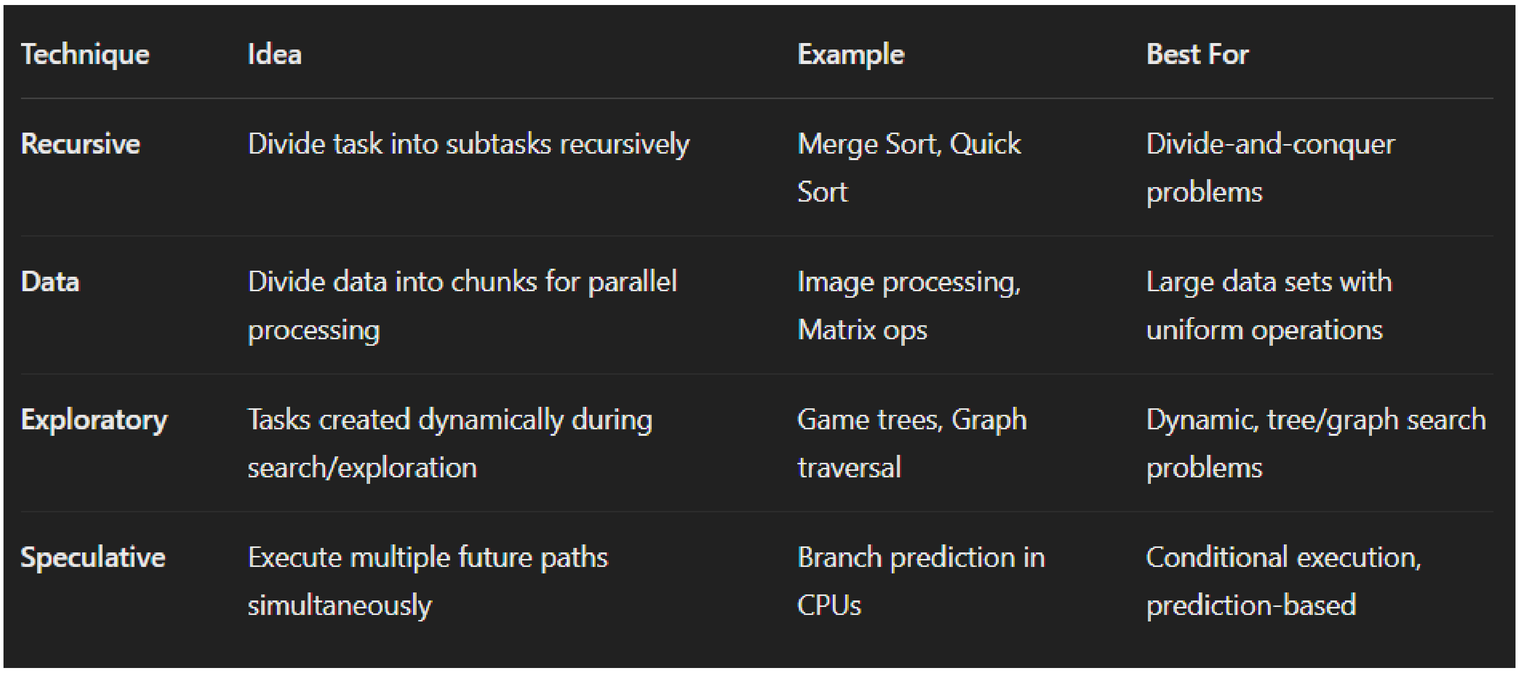
$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

UNIT 2

Introduction to Parallel Programming

Motivation,
Scope of Parallel Computing,
Principles of Parallel Algorithm design:
Preliminaries,
Decomposition Techniques,
Characteristics of Tasks and Interactions,
Mapping Techniques for Load Balancing,
Methods for containing Interaction Overheads,
Parallel Algorithms Models



Technique	Idea	Example	Best For
Recursive	Divide task into subtasks recursively	Merge Sort, Quick Sort	Divide-and-conquer problems
Data	Divide data into chunks for parallel processing	Image processing, Matrix ops	Large data sets with uniform operations
Exploratory	Tasks created dynamically during search/exploration	Game trees, Graph traversal	Dynamic, tree/graph search problems
Speculative	Execute multiple future paths simultaneously	Branch prediction in CPUs	Conditional execution, prediction-based

DECOMPOSITION Techniques

- Recursive decomposition
- Data decomposition
- Exploratory decomposition
- Speculative decomposition

RECURSIVE DECOMPOSITION

It is generally suited to problems that are solved using the divide-and-conquer strategy.

The problem is first decomposed into a set of sub-problems.

These sub-problems are recursively decomposed further until a desired granularity is reached.

Eg: Quick Sort, Merge Sort

•) Applying it to the matrix vector multiplication problem.

Let $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ $b = \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix}$.

$\Rightarrow Ab = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} a_{11} b_{11} + a_{12} b_{11} \\ a_{21} b_{11} + a_{22} b_{21} \end{bmatrix} = \begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix}$.

Result vector. V

4

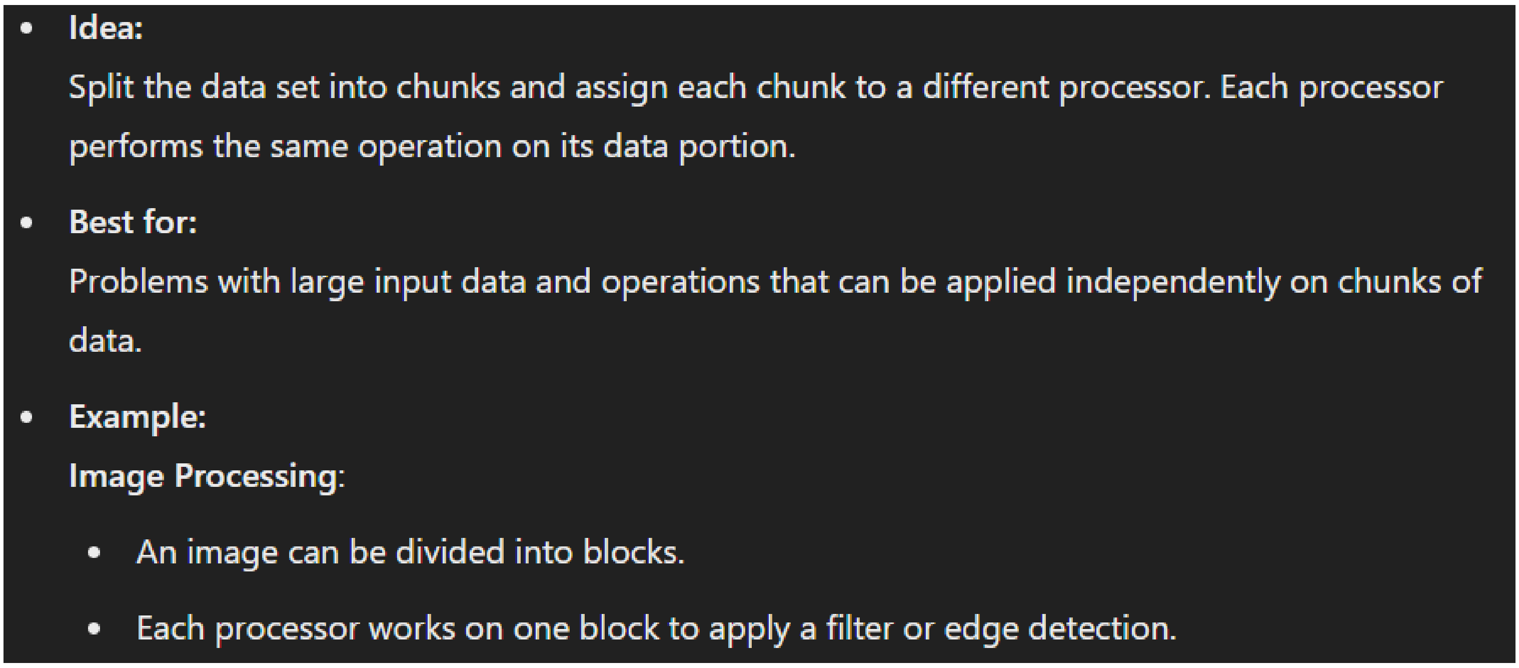
$$\begin{tikzcd}
 & & \begin{bmatrix} a_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} \arrow[dl] \arrow[dr] & & \\
 & \begin{bmatrix} a_{11} \mid a_{12} \end{bmatrix}\begin{bmatrix} b_{11} \\ \hline b_{21} \end{bmatrix} \arrow[dl] \arrow[d] & & \begin{bmatrix} a_{21} \mid a_{22} \end{bmatrix}\begin{bmatrix} b_{11} \\ \hline b_{21} \end{bmatrix} \arrow[d] \arrow[dr] & \\
 \begin{bmatrix} a_{11} \end{bmatrix}\begin{bmatrix} b_{11} \end{bmatrix} \arrow[r, phantom, "+"] & \begin{bmatrix} a_{12} \end{bmatrix}\begin{bmatrix} b_{21} \end{bmatrix} \arrow[dr] & & \begin{bmatrix} a_{21} \end{bmatrix}\begin{bmatrix} b_{11} \end{bmatrix} \arrow[r, phantom, "+"] \arrow[dl] & \begin{bmatrix} a_{22} \end{bmatrix}\begin{bmatrix} b_{21} \end{bmatrix} \\
 & & \begin{bmatrix} a_{11} b_{11} + a_{12} b_{21} \\ a_{21} b_{11} + a_{22} b_{21} \end{bmatrix} = V. & &
\end{tikzcd}$$

DATA DECOMPOSITION

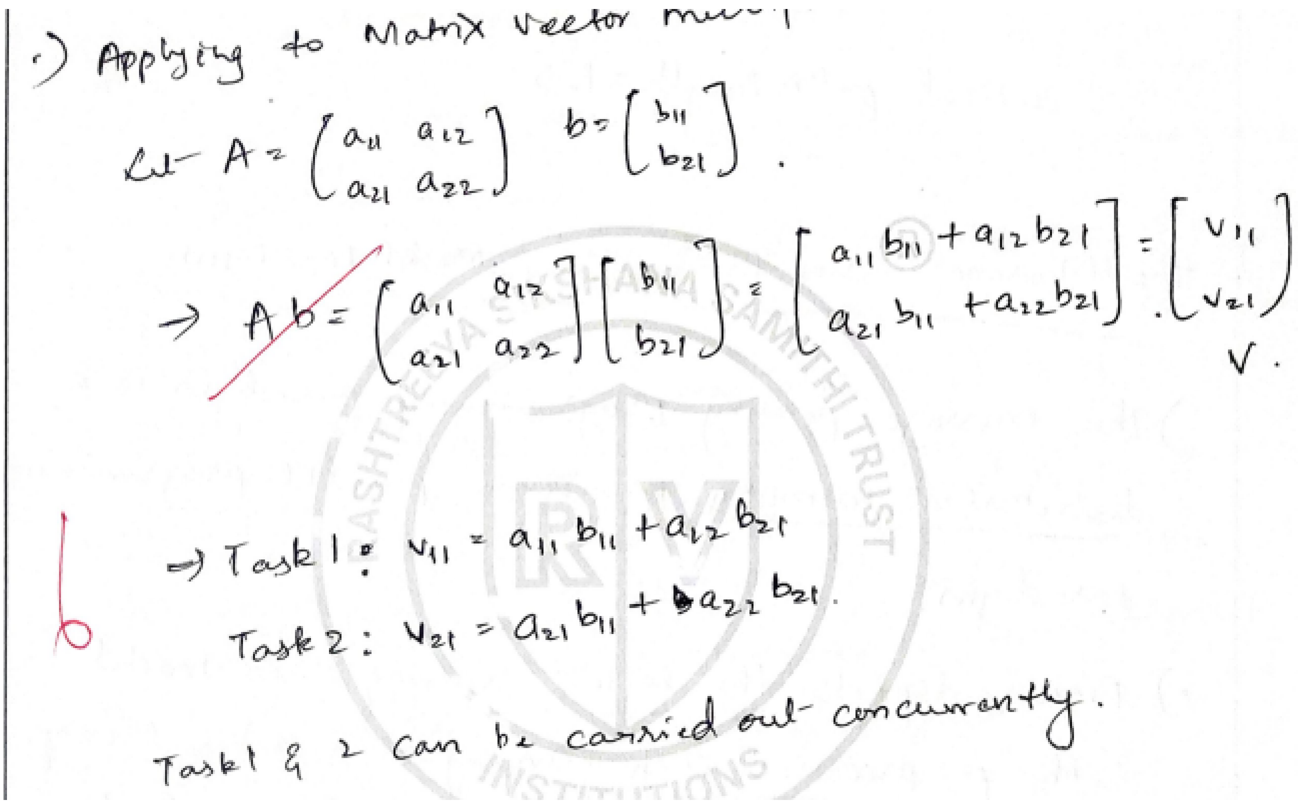
Identify the data on which computations are performed.
Partition this data across various tasks.

This partitioning induces a decomposition of the problem.

Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.



- **Idea:**
  Split the data set into chunks and assign each chunk to a different processor. Each processor performs the same operation on its data portion.
- **Best for:**
  Problems with large input data and operations that can be applied independently on chunks of data.
- **Example:**
  Image Processing:
  - An image can be divided into blocks.
  - Each processor works on one block to apply a filter or edge detection.



:) Applying to Matrix vector mul

Let $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ $b = \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix}$.

$\rightarrow \cancel{Ab} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} a_{11} b_{11} + a_{12} b_{21} \\ a_{21} b_{11} + a_{22} b_{21} \end{bmatrix} = \begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix}$.
V.

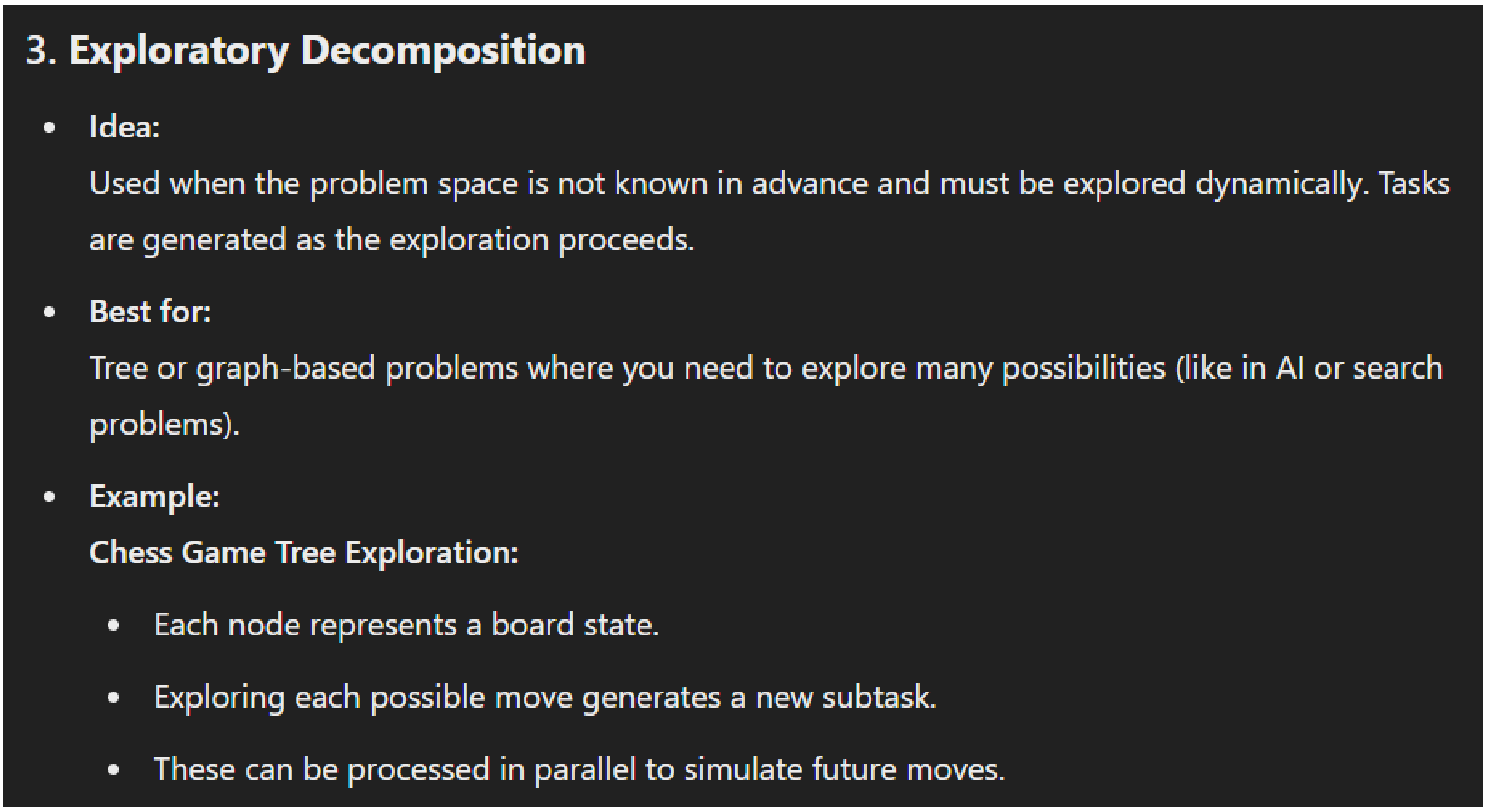
6

$\rightarrow$ Task 1 : $v_{11} = a_{11} b_{11} + a_{12} b_{21}$

Task 2 : $v_{21} = a_{21} b_{11} + a_{22} b_{21}$.

Task 1 & 2 can be carried out concurrently.

EXPLORATORY DECOMPOSITION

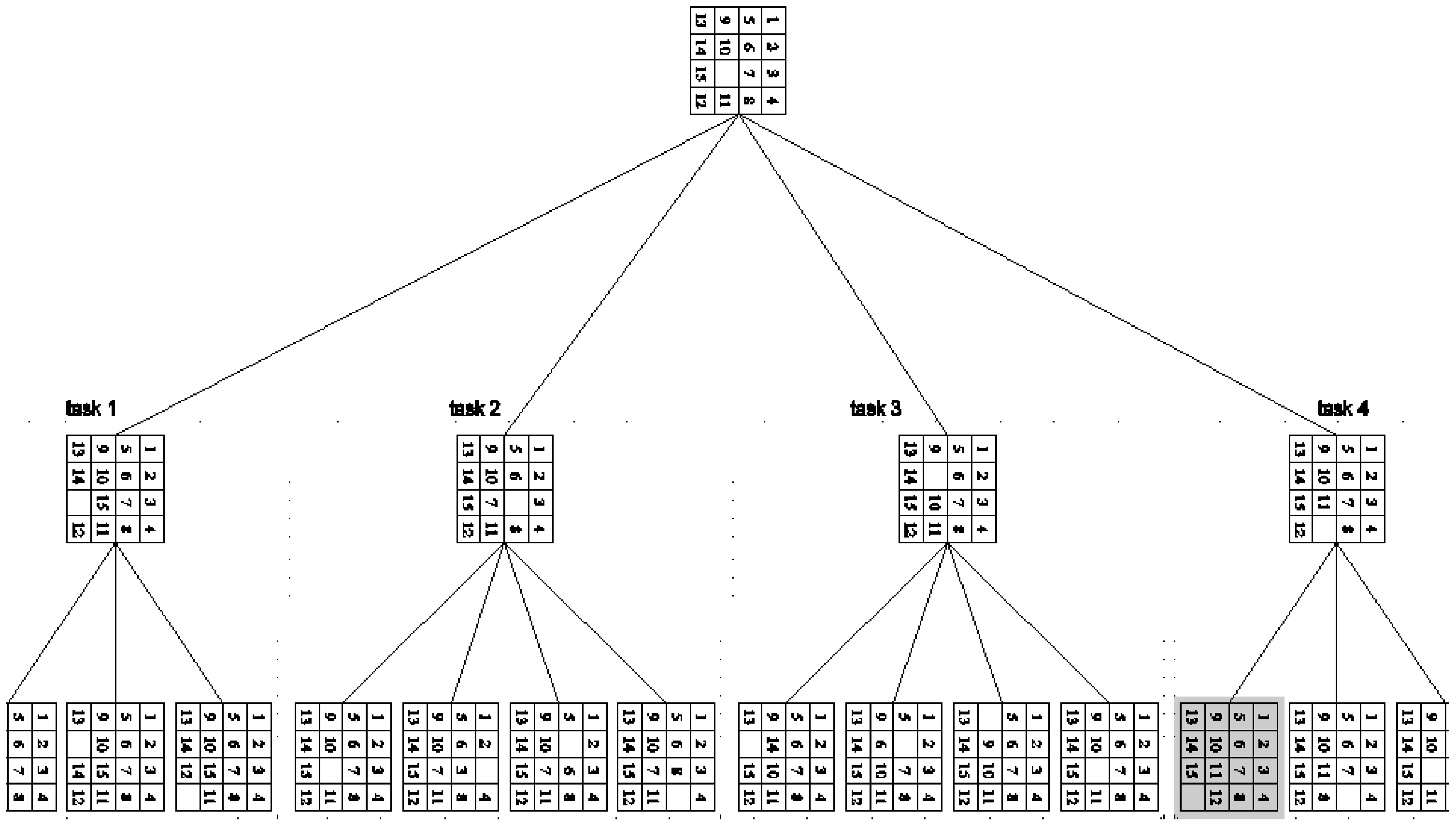


3. Exploratory Decomposition

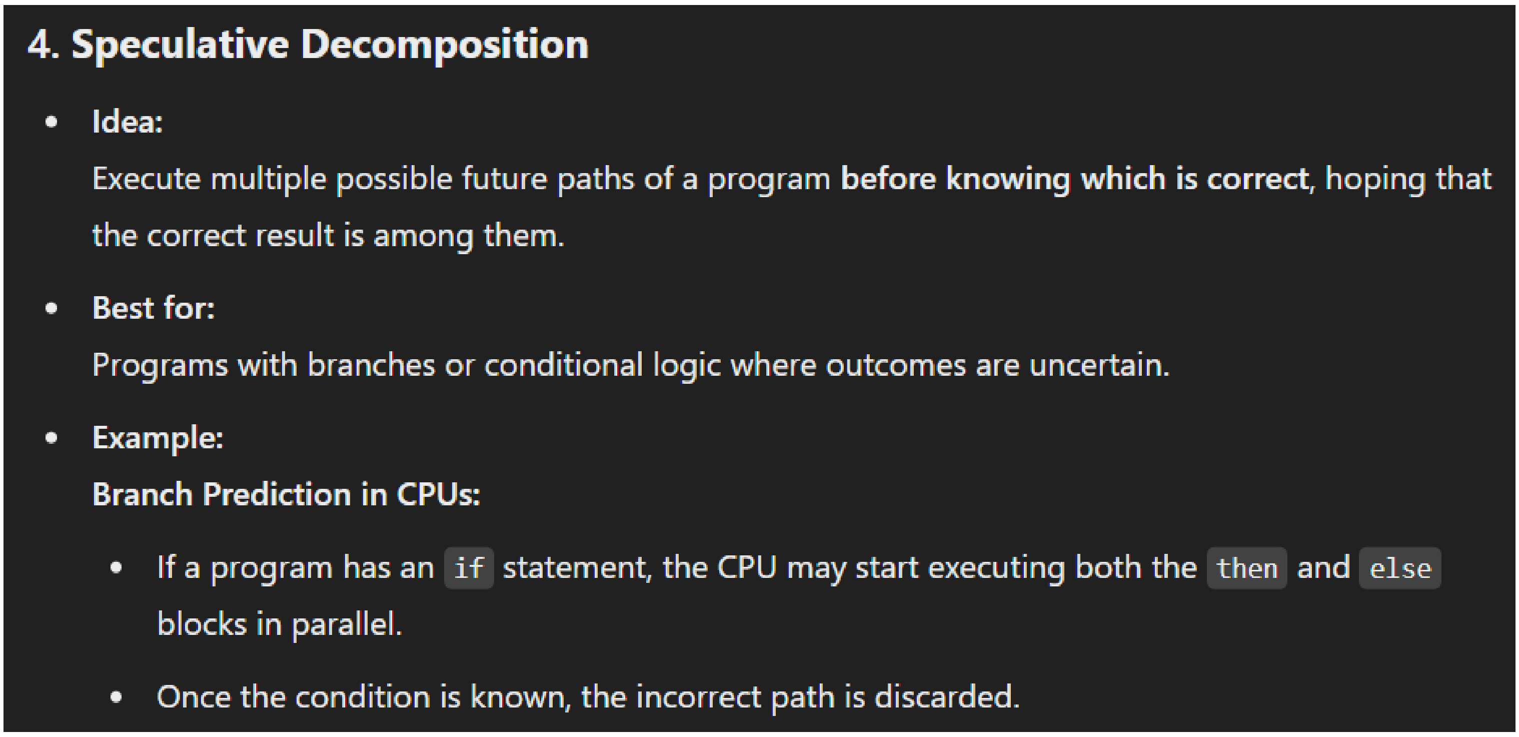
- **Idea:**
  Used when the problem space is not known in advance and must be explored dynamically. Tasks are generated as the exploration proceeds.
- **Best for:**
  Tree or graph-based problems where you need to explore many possibilities (like in AI or search problems).
- **Example:**
  Chess Game Tree Exploration:
  - Each node represents a board state.
  - Exploring each possible move generates a new subtask.
  - These can be processed in parallel to simulate future moves.

Problems in this class include a variety of discrete optimization problems: Theorem Proving, Game Playing, ...

The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



SPECULATIVE DECOMPOSITION



4. Speculative Decomposition

- **Idea:**
  Execute multiple possible future paths of a program **before knowing which is correct**, hoping that the correct result is among them.
- **Best for:**
  Programs with branches or conditional logic where outcomes are uncertain.
- **Example:**
  Branch Prediction in CPUs:
  - If a program has an `if` statement, the CPU may start executing both the `then` and `else` blocks in parallel.
  - Once the condition is known, the incorrect path is discarded.

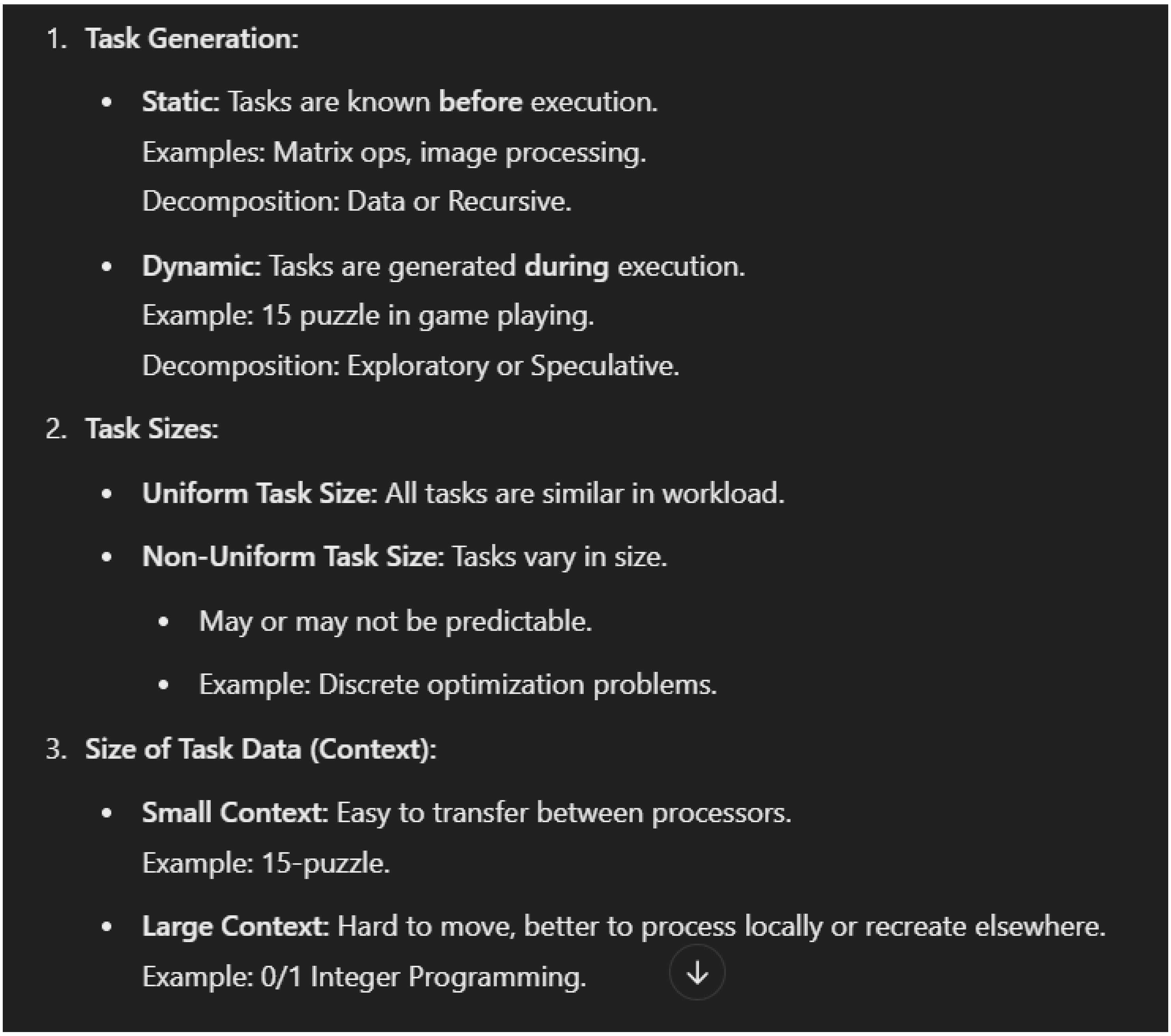
Dependencies between tasks are not known a-priori. It is impossible to identify independent tasks.

There are generally two approaches to dealing with such applications:

Conservative approaches: which identify independent tasks only when they are guaranteed to not have dependencies.
Yields little concurrency

Optimistic approaches: which schedule tasks even when they may potentially be erroneous.
Require roll-back mechanism in case of error

CHARACTERISTICS OF TASKS



1. **Task Generation:**
   - **Static:** Tasks are known **before** execution.
     Examples: Matrix ops, image processing.
     Decomposition: Data or Recursive.
   - **Dynamic:** Tasks are generated **during** execution.
     Example: 15 puzzle in game playing.
     Decomposition: Exploratory or Speculative.
2. **Task Sizes:**
   - **Uniform Task Size:** All tasks are similar in workload.
   - **Non-Uniform Task Size:** Tasks vary in size.
     - May or may not be predictable.
     - Example: Discrete optimization problems.
3. **Size of Task Data (Context):**
   - **Small Context:** Easy to transfer between processors.
     Example: 15-puzzle.
   - **Large Context:** Hard to move, better to process locally or recreate elsewhere.
     Example: 0/1 Integer Programming.

Methods for containing Interaction Overheads

Reducing the interaction overhead among concurrent tasks is important for an efficient parallel program.

- Maximizing Data Locality
- Minimizing Contention and Hot Spots
- Overlapping Computations with Interactions
- Replicating Data or Computations
- Using Optimized Collective Interaction Operations
- Overlapping Interactions with Other Interactions

Maximizing Data Locality

By doing this, different processes access the shared data.

It minimizes the amount of non-local data accessed,
maximizing reuse of recently accessed data, and
reducing the frequency of interactions between processes.

Minimize Volume of Data-Exchange:

Minimize the amount of shared data that processes need to access. By maximizing *temporal data locality* (i.e., accessing the same data multiple times), programs can avoid frequent data retrieval from memory.

For instance, when mapping computations, using higher-dimensional distributions (as opposed to 1D) can reduce the volume of shared data. This is similar to modern CPU cache optimizations that aim to reuse cached data as much as possible.

Example: In matrix multiplication, a two-dimensional mapping can reduce shared data access compared to a one-dimensional mapping.

Intermediate Results: Another approach is to use local data to store intermediate results, only interacting with shared data to store the final result. This minimizes access to shared memory locations.

Minimize Frequency of Interactions:
Reducing how often interactions occur is essential.
One way is to restructure the algorithm so that large blocks of shared data are accessed in one go, thus reducing the number of interactions overall. This concept is similar to improving *spatial locality*, ensuring that data accessed consecutively are close together, either in memory or through network transfers.

Minimizing Contention and Hot Spots

In parallel programs, **contention** occurs when multiple tasks attempt to access the same resource simultaneously, such as shared memory or interconnection links. This can significantly increase interaction overheads because only one operation can proceed at a time, while others must wait their turn. Contention can arise in various scenarios, like when multiple tasks access the same block of memory or send messages to the same process.

Reducing Contention: To minimize contention, the parallel algorithm can be redesigned to access data in a way that avoids simultaneous access to the same resources. For the matrix multiplication example, the computation order can be rearranged to ensure that each task accesses a unique block of matrices A and B, eliminating contention. This ensures that tasks working on the same row or column of the result don’t try to access the same data at the same time.

Centralized vs. Distributed Mapping: Centralized dynamic mapping schemes can also cause contention, especially with shared data structures or communication channels. Switching to a distributed mapping scheme can help reduce contention, even though it might be harder to implement.

Overlapping Computations with Interactions

One way to reduce interaction overhead is to **overlap computations with interactions**, which allows tasks to continue working while waiting for shared data or additional work. This can be done in several ways:

1. **Initiate Interactions Early**: By starting interactions ahead of time, they can be completed before the program needs the data. This is possible if the interaction pattern is predictable or if multiple tasks are ready to execute on the same process. However, this may increase overhead by reducing task granularity, so it should be used carefully.
2. **Anticipating Work Transfers**: In dynamic mapping schemes, a process that is about to run out of work can request additional tasks in advance and continue working while waiting for the new tasks. This requires anticipating when a task will run out of work and initiating a request ahead of time.
3. **Programming and Hardware Support**: Overlapping computations and interactions often require support from the **programming paradigm**, **operating system**, and **hardware**. For example, message-passing systems with non-blocking primitives allow a program to initiate communication and continue computing without waiting for the message to complete. On **shared-memory systems**, prefetching hardware can anticipate future memory accesses and begin loading data before it’s needed, reducing waiting time.

In summary, overlapping computations with interactions helps reduce idle time, improving overall parallel program performance. However, this technique depends on careful implementation and support from the system's hardware and software.

Replicating Data or Computations

Replication involves creating copies of data or computations to reduce interaction overheads. This technique is useful when multiple processes frequently need read-only access to shared data structures, like hash tables, in parallel algorithms.

- **Data Replication**: If memory usage allows, it's often beneficial to replicate shared data on each process. After the initial data replication interaction, subsequent accesses to this data will be free of interaction overhead.
    - **Example**: In shared-memory systems, caches often handle data replication automatically. However, in message-passing paradigms (where data sharing is more complex), explicit replication of data is especially useful as it simplifies the program and reduces interaction overhead.
- **Cost of Replication**: While replication reduces overhead, it increases memory usage. The total memory required grows linearly with the number of processes. For this reason, replication should only be used for small amounts of data to avoid excessive memory consumption.
- **Replicating Computations**: In some cases, it may be more efficient for a process to compute intermediate results instead of fetching them from another process. This is particularly useful when the interaction overhead for sharing results is high.
    - **Example**: In the Fast Fourier Transform (FFT), rather than sharing "twiddle factors" (complex coefficients), each process can compute the factors it needs locally. While this increases the number of computations, it may still lead to faster execution compared to fetching the data from other processes

Using Optimized Collective Interaction Operations

In parallel programs, some interaction patterns are static and regular, often involving multiple tasks working together. These patterns are common in operations like broadcasting data or performing collective computations, such as summing values across processes. Such interactions are known as **collective operations**.

- **Types of Collective Operations**:
    1. **Data Access Operations**: Tasks accessing shared data across processes.
    2. **Communication-Intensive Computations**: Tasks performing computations that involve significant communication between processes.
    3. **Synchronization Operations**: Operations ensuring tasks are synchronized, such as barrier synchronization.
- **Optimized Implementations**: Optimized algorithms for these collective operations are available to minimize data transfer overhead and contention. For example, the **MPI (Message Passing Interface)** provides highly optimized collective operations like broadcasting or summing data, allowing programmers to focus on the functionality rather than the underlying implementation details.
- **Custom Collective Operations**: Sometimes, the interaction pattern in a program may benefit from a custom implementation of collective operations. This can be the case when the default optimized operations do not suit the specific needs of the algorithm or architecture.

In summary, optimized collective operations reduce interaction overheads and simplify parallel programming. However, in some cases, designing custom operations tailored to the problem may offer additional benefits.

Overlapping Interactions with Other Interactions

Overlapping interactions refers to executing multiple communication operations concurrently to reduce the total communication time. This is possible if the underlying hardware allows for simultaneous data transfers between different pairs of processes. By overlapping interactions, the overall volume of communication can be reduced, leading to more efficient parallel communication.

Example of Overlapping Interactions (Broadcast):

Consider the common operation of **broadcasting data from one process (P0) to all other processes** in a message-passing paradigm. Here’s how the interaction can be overlapped:

- **Step-by-Step Overlap**: In the first step, P0 sends data to P2. In the second step, P0 sends data to P1, and simultaneously, P2 sends the data it received from P0 to P3. The entire operation is completed in two steps, as illustrated in Figure 3.41(a). This is more efficient than a naive broadcast where data is sent sequentially from P0 to P1, P1 to P2, and P2 to P3, which takes three steps (Figure 3.41(b)).
- **Increased Overlap with Naive Algorithm**: Interestingly, the naive broadcast method (Figure 3.41(b)) can sometimes be adapted to increase overlap, especially when multiple data structures need to be broadcast. For example, when broadcasting four messages, the naive method can reduce the total number of steps from eight (in the optimized method) to six. In this case, P0 sends the first message to P1, then sends the second message while P1 sends the first message to P2, and so on. The messages "pipeline" through the system, reducing the total steps.

When to Use Overlapping Interactions:

- **Efficiency Considerations**: While overlapping interactions can be highly effective, they might not always be the best choice for a single broadcast operation, as they could incur high costs. However, in certain situations, like broadcasting multiple data structures, the naive algorithm may perform better by reducing the total communication time.

- **Custom Collective Communication**: In some cases, the programmer may need to create custom communication functions to take advantage of these overlapping strategies. This approach might deviate from the typical collective communication libraries but can provide performance improvements based on the specific interaction pattern of the algorithm.

In summary, overlapping interactions can reduce communication overhead, but it requires careful consideration of the interaction patterns and hardware capabilities. Depending on the context, writing custom communication functions may be necessary to optimize performance.

Parallel Algorithms Models

Parallel programming models provide structured approaches to decompose problems and map them onto multiple processing units to achieve efficient computation. These models aim to reduce communication overhead, optimize data movement, and improve computational performance. Below are the main parallel algorithm models discussed:

1. Data-Parallel Model

- **Concept**: The data-parallel model is a straightforward parallelism strategy where tasks perform identical operations on different pieces of data. This is called *data parallelism*. The problem is divided into smaller tasks, each of which processes a partition of the data.
- **Mapping**: Tasks are statically or semi-statically mapped to processes. A uniform partitioning of data ensures load balance, and data interaction is minimized by utilizing locality-preserving decompositions.
- **Advantages**: This model is simple to implement and scales well with problem size. Data parallelism increases as the problem size grows, making it suitable for large-scale computations.
- **Use Cases**: A classic example is **dense matrix multiplication**, where tasks operate on different parts of the matrix.
- **Paradigms**: The data-parallel model can be implemented in both **shared-address-space** and **message-passing** paradigms, with the message-passing paradigm offering better control over locality.

2. Task Graph Model

- **Concept**: This model leverages the dependencies between tasks in the form of a *task-dependency graph*, where tasks are connected based on their data and computation dependencies.
- **Mapping**: Tasks are mapped based on the structure of the dependency graph, optimizing for locality and reducing communication costs. The mapping can be static or dynamic depending on the problem.
- **Advantages**: It is effective for problems where the computational load is uneven across tasks. By focusing on task interactions and locality, this model reduces the volume and frequency of communication.
- **Use Cases**: Algorithms such as **parallel quicksort** and **sparse matrix factorization** fit well in this model.
- **Paradigms**: The task graph model is more suitable for message-passing paradigms, where tasks have distinct interaction patterns, but it can also be adapted for shared-address-space systems.

3. Work Pool Model

- **Concept**: The work pool (or task pool) model involves dynamically assigning tasks to processes for load balancing, where any process can execute any task. Tasks may be generated statically at the beginning or dynamically throughout execution.
- **Mapping**: Tasks are stored in a centralized or decentralized data structure, such as a priority queue or hash table. Dynamic mapping ensures that tasks are distributed efficiently based on process availability.
- **Advantages**: It allows dynamic load balancing and flexibility in task assignment. The granularity of tasks can be adjusted to optimize performance.
- **Use Cases**: **Parallel tree search** and **chunk scheduling** for loop parallelization are examples where the work pool model is beneficial.

- **Paradigms**: This model is commonly used in the **message-passing paradigm**, where data interaction overhead is minimal, and task movement can be done easily.

4. Master-Slave Model

- **Concept**: In the master-slave model, one or more "master" processes manage work allocation to "worker" processes. The master generates or assigns tasks, and workers execute the tasks. The model can be hierarchical with multiple levels of masters and workers.
- **Mapping**: There is no fixed mapping of tasks to processes; tasks are dynamically assigned. This model ensures that work is evenly distributed but requires careful attention to avoid bottlenecks, especially if the master becomes overwhelmed.
- **Advantages**: Simple to implement and flexible in task management. It is particularly useful when the master can efficiently generate tasks and workers can perform tasks independently.
- **Use Cases**: Tasks such as parallel simulations or phased computations (e.g., **matrix factorization**) work well with this model.
- **Paradigms**: Suitable for both **shared-address-space** and **message-passing paradigms**, depending on how data and tasks are distributed.

5. Pipeline or Producer-Consumer Model

- **Concept**: This model uses a series of processes that form a pipeline. Each process performs some computation on a data stream, with each stage (process) being a producer and a consumer. The next process in the pipeline consumes data from the previous one and produces data for the next.
- **Mapping**: Tasks are mapped in a sequence along the pipeline. While the pipeline is usually statically mapped, the process can be extended to more complex forms like multidimensional arrays or directed graphs.

- **Advantages**: Efficient for tasks where data flows sequentially through various stages. It is useful for continuous data processing and stream-based parallelism.
- **Use Cases**: **LU factorization** and real-time data processing can be effectively modeled using this strategy.
- **Paradigms**: This model typically benefits from **shared-address-space** systems, but **message-passing** paradigms can also be used with appropriate synchronization mechanisms.

6. Hybrid Models

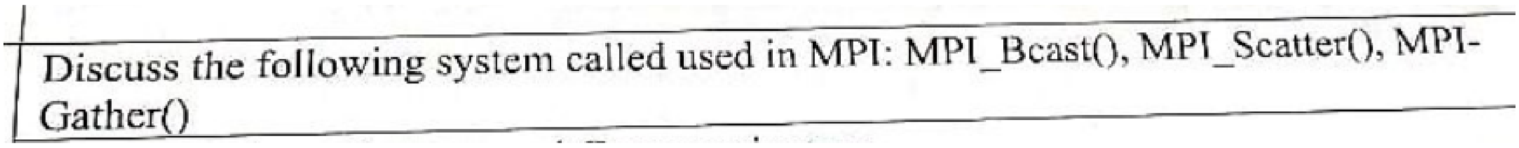
- **Concept**: Hybrid models combine multiple parallel algorithm models for different phases or parts of an algorithm. This flexibility allows adapting the best strategy for each part of the computation.
- **Mapping**: The problem may be divided hierarchically or sequentially into subproblems, with each part being suited for different parallel models. For example, a task graph may have pipelined subgraphs or data-parallel components.
- **Advantages**: It leverages the strengths of various models to maximize performance. The ability to tailor the parallel strategy to different algorithm phases ensures optimal use of resources.
- **Use Cases**: **Parallel quicksort** is an example of an algorithm that benefits from hybrid parallelism.
- **Paradigms**: Hybrid models can integrate multiple paradigms (message-passing and shared-address-space) to optimize computation and communication.

UNIT 3

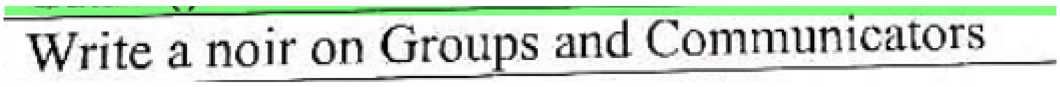
Programming Using the Message Passing Paradigm

Principles of Message Passing Programming,
Building Blocks,
MPI,
Topologies and Embedding,

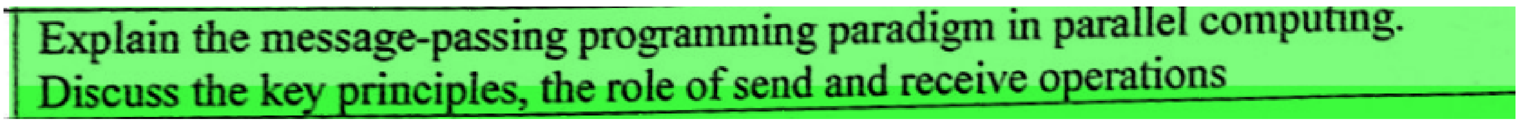
Overlapping Communication with computation,
Collective Communication and computation operations,
Groups and Communicators



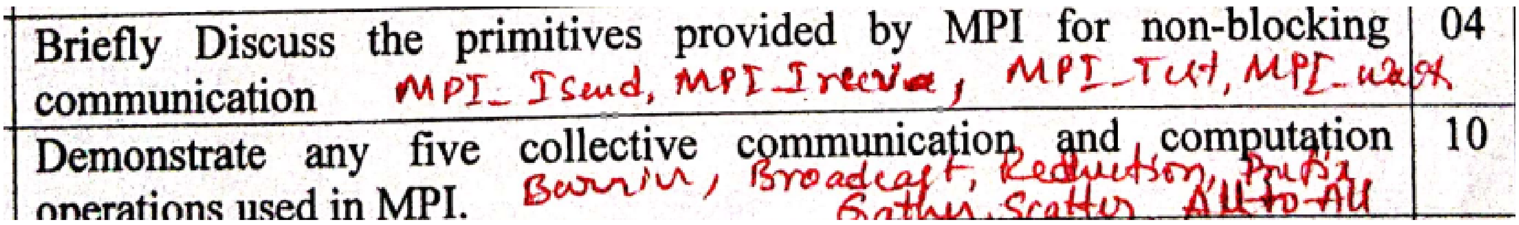
Discuss the following system called used in MPI: MPI_Bcast(), MPI_Scatter(), MPI-Gather()



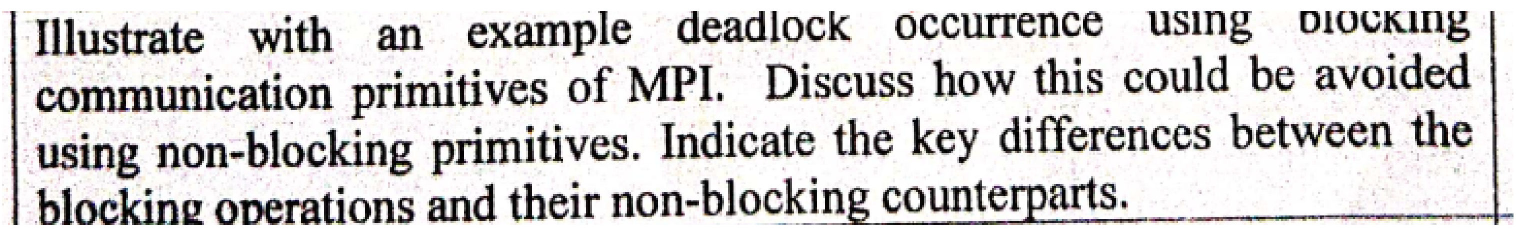
Write a noir on Groups and Communicators



Explain the message-passing programming paradigm in parallel computing. Discuss the key principles, the role of send and receive operations



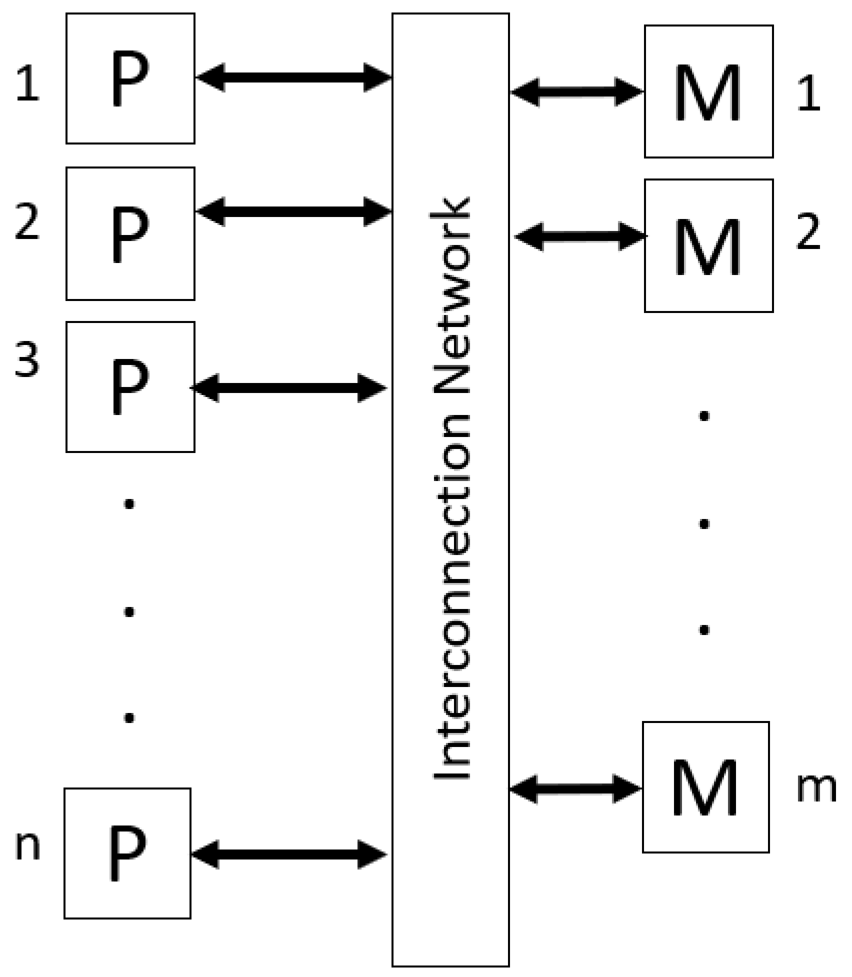
Briefly Discuss the primitives provided by MPI for non-blocking communication MPI_Isend, MPI_Irecva, MPI_Test, MPI_wait	04
Demonstrate any five collective communication and computation operations used in MPI. Barrier, Broadcast, Reduction, Prefix, Gather, Scatter, All-to-All	10



Illustrate with an example deadlock occurrence using blocking communication primitives of MPI. Discuss how this could be avoided using non-blocking primitives. Indicate the key differences between the blocking operations and their non-blocking counterparts.

Principles of Message Passing Programming

Message Passing is a parallel programming model where processes communicate by sending messages

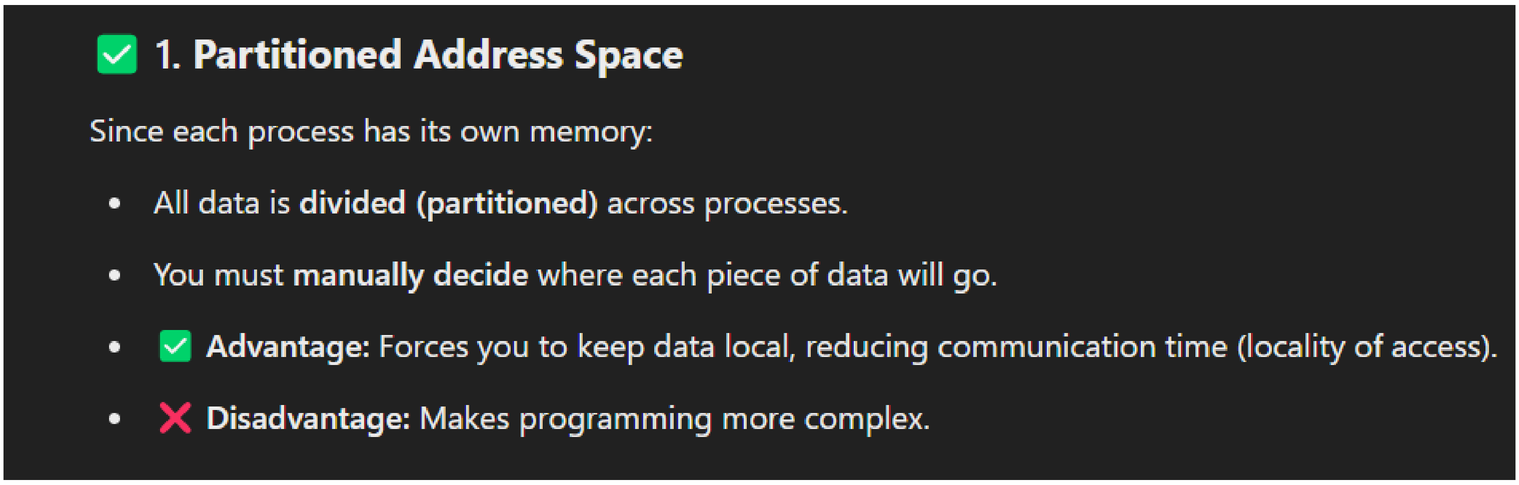


- Each process has its own memory (no shared data)
- It is highly scalable and suitable for large-scale distributed systems
- It is complex to program and can lead to deadlocks

Principles of Message-passing programming:

1) Partitioned address space
2) Explicit parallelization
3) Structure (Asynchronous, Loosely Synchronous)
4) Single Program Multiple Data (SPMD)
5) Basic Operations (Send, Receive)

1) Partitioned Address Space



✅ 1. Partitioned Address Space

Since each process has its own memory:

- All data is **divided (partitioned)** across processes.
- You must **manually decide** where each piece of data will go.
- ✅ **Advantage:** Forces you to keep data local, reducing communication time (locality of access).
- ❌ **Disadvantage:** Makes programming more complex.

2) Explicit Parallelization

The programmer is responsible for identifying and decomposing tasks to extract concurrency, making this approach intellectually demanding but capable of high performance and scalability when done correctly.

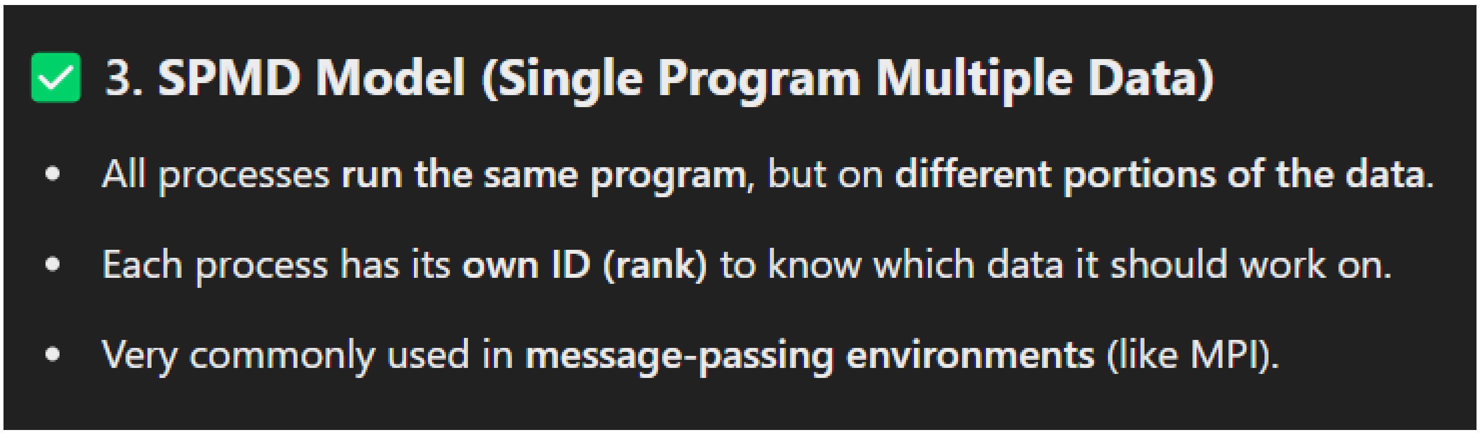
3) Structure of Message-Passing Programs:

Message-passing programs are typically written in two paradigms:

Asynchronous programs allow for fully concurrent execution but can be challenging to reason about due to race conditions.

Loosely Synchronous programs offer a balance, with tasks synchronizing for interactions but executing asynchronously between them, making the program easier to manage.

4) Single Program Multiple Data



✅ 3. SPMD Model (Single Program Multiple Data)

- All processes **run the same program**, but on **different portions of the data.**
- Each process has its **own ID (rank)** to know which data it should work on.
- Very commonly used in **message-passing environments** (like MPI).

BUILDING BLOCKS

In message-passing programming, interactions between processes are performed using the basic operations: **send** and **receive**.

- **send**(void *sendbuf, int nelems, int dest)
- **receive**(void *recvbuf, int nelems, int source)

- sendbuf and recvbuf are pointers to the data buffers for sending and receiving data.
- nelems is the number of data elements to be transferred.
- dest and source are the identifiers of the receiving and sending processes.

Though these operations might appear straightforward, their implementation involves complexities that can affect both program correctness and performance.

P0

```
a = 100;
send(&a,1, 1);
a = 0;
```

P1

```
receive(&a, 1, 0)
printf("%d\n", a);
```

The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0

Non-Buffered Blocking Message Passing Operations

📌 **Definition:**

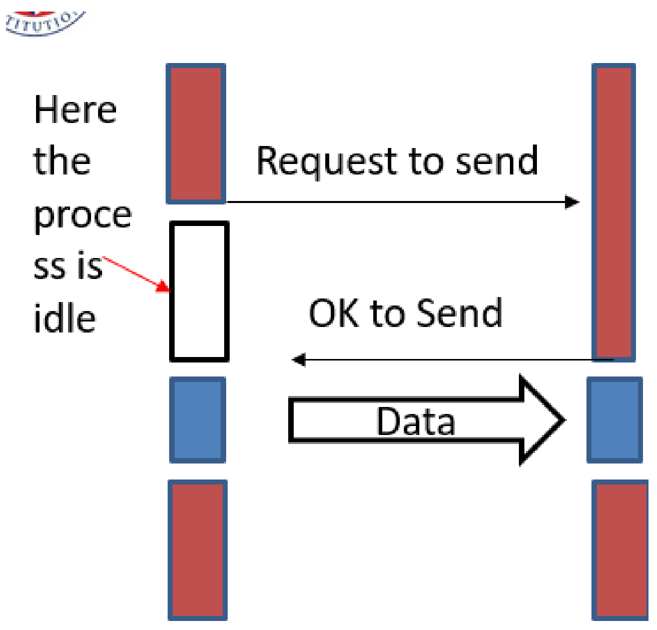
In **non-buffered blocking message passing**, the **sender** is *blocked* (i.e., it waits and cannot proceed) **until the receiver is ready** to accept the message.

There is **no intermediate storage (buffer)** to temporarily hold the message.

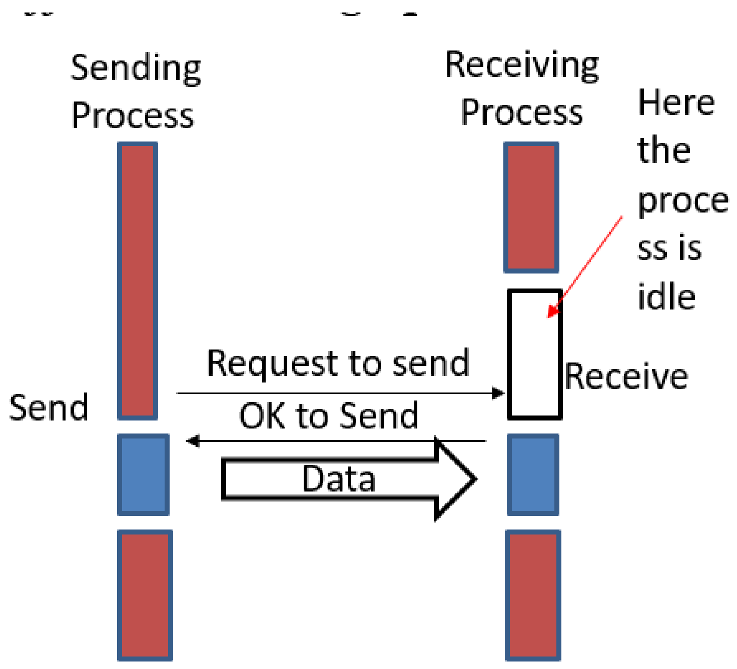
Simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.

This involves a handshake between the sending and receiving processes

The sending process sends a request to communicate to the receiving process. When the receiving process encounters the target receive, it responds to the request. The sending process upon receiving this response initiates a transfer operation.



Sender comes first; idling at sender



Here the receiver is ready but the sender is not posted.

Message Passing Interface (MPI)

It is a standardized means of exchanging messages between multiple computers running parallel program across distributed memory

- Library Specification for message passing
- Defines syntax and semantics of core routines
- Routines help to write message passing programs

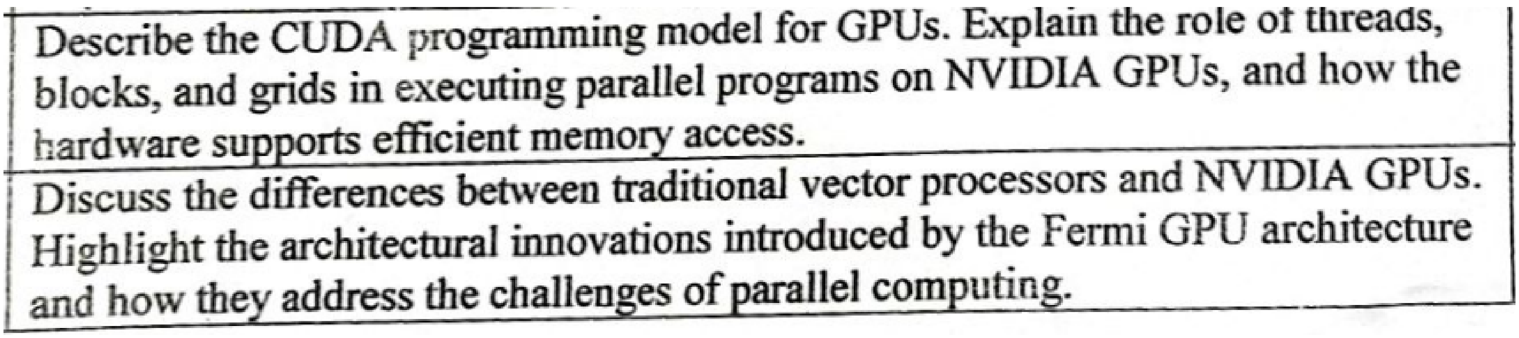
- Header file “mpi.h”
- It is for parallel computers, clusters, heterogeneous systems
- Can be used with C/C++ and Fortran

MPI Routines	**Descriptions**
MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of the calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.

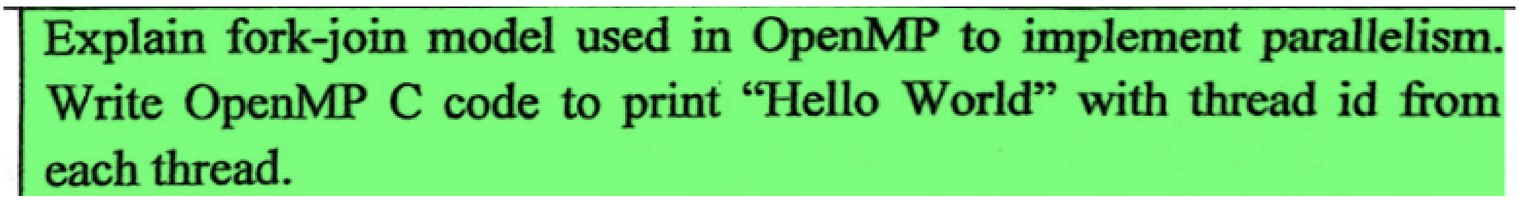
UNIT 4

Overview of OpenMP and GPU Architectures
Introduction,
the idea of OpenMP,
the feature set,
OpenMP Language Features,
Parallel Construct,
Sharing the work among threads in an OpenMP program,
Clauses to control parallel and
Work-Sharing Constructs,
OpenMP Synchronization Constructs.

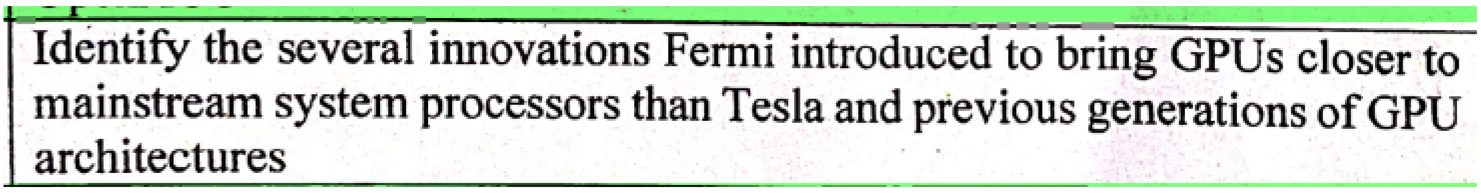
Introduction to Graphics Processing Units,
Detecting and Enhancing Loop-Level Parallelism,
Mobile versus Server GPUs and
Tesla versus Core i7,
GPU programming using CUDA



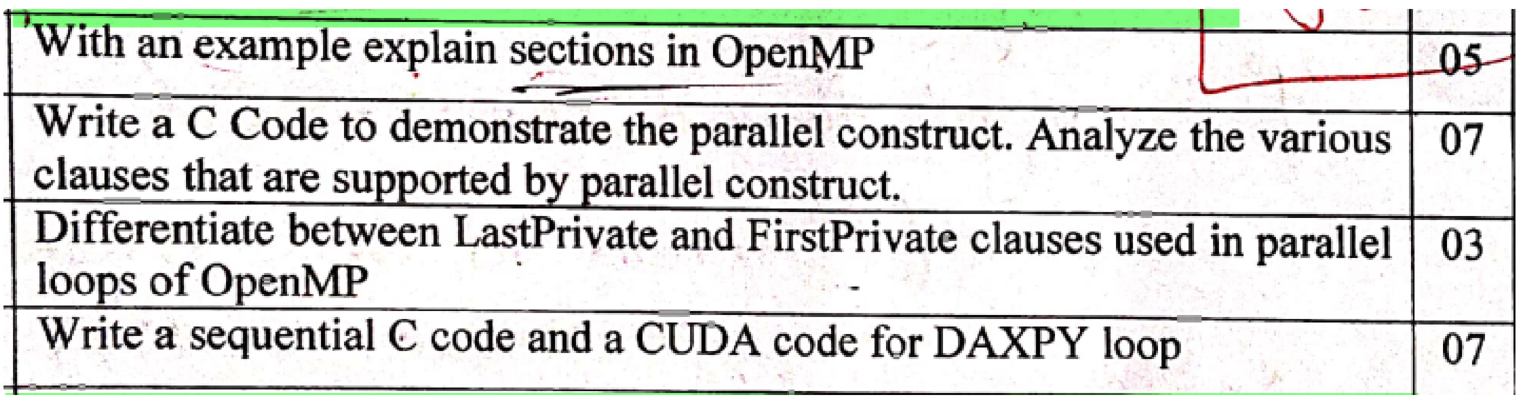
Describe the CUDA programming model for GPUs. Explain the role of threads, blocks, and grids in executing parallel programs on NVIDIA GPUs, and how the hardware supports efficient memory access.
Discuss the differences between traditional vector processors and NVIDIA GPUs. Highlight the architectural innovations introduced by the Fermi GPU architecture and how they address the challenges of parallel computing.



Explain fork-join model used in OpenMP to implement parallelism. Write OpenMP C code to print “Hello World” with thread id from each thread.



Identify the several innovations Fermi introduced to bring GPUs closer to mainstream system processors than Tesla and previous generations of GPU architectures



With an example explain sections in OpenMP	05
Write a C Code to demonstrate the parallel construct. Analyze the various clauses that are supported by parallel construct.	07
Differentiate between LastPrivate and FirstPrivate clauses used in parallel loops of OpenMP	03
Write a sequential C code and a CUDA code for DAXPY loop	07

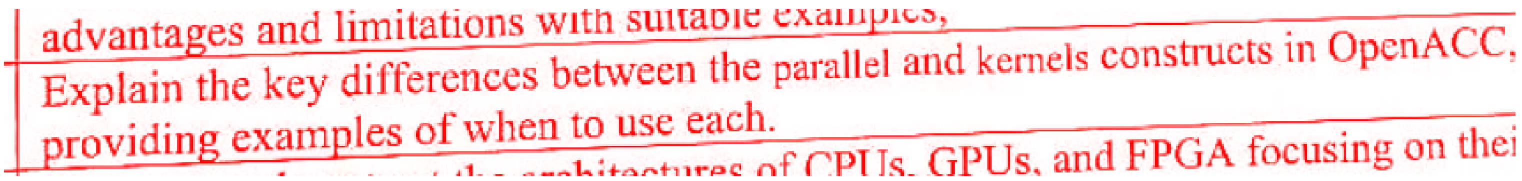
UNIT 5

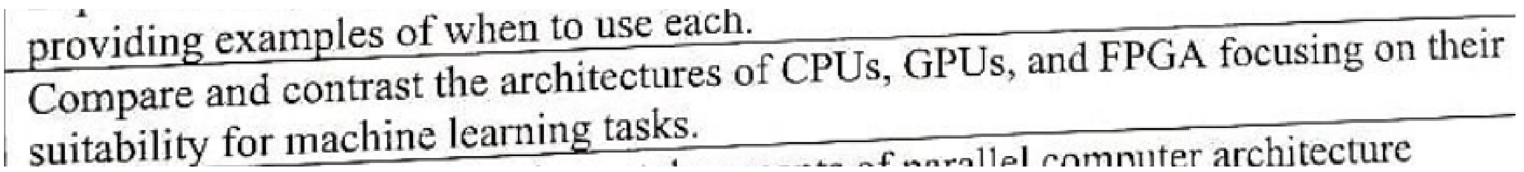
Heterogeneous Computing From Serial To Parallel Programming Using OpenAcc,
a simple data-parallel loop,
the OpenAcc kernels and parallel construct,
the various forms of OpenAcc parallelism.

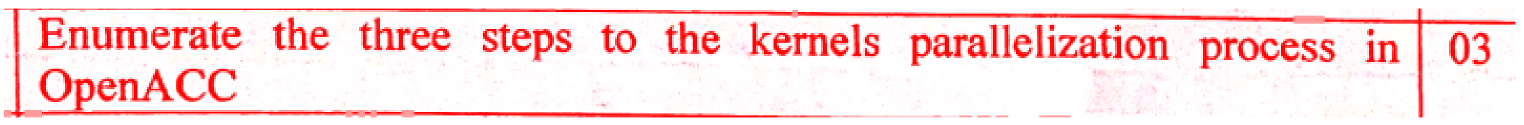
Intel FPGAs:
Introduction to Intel FPGAs and
Intel Quartus Prime
Design Software - FPGA design and implementation.

Intel SoC FPGAs:
Introduction to Intel SoC FPGAs -
IP design and Platform designer,
Embedded System design using Cyclone V and
ARM -SoC Design Flow

Discuss the architecture of an FPGA.


advantages and limitations with suitable examples,
Explain the key differences between the parallel and kernels constructs in OpenACC,
providing examples of when to use each.
the architectures of CPUs, GPUs, and FPGA focusing on thei

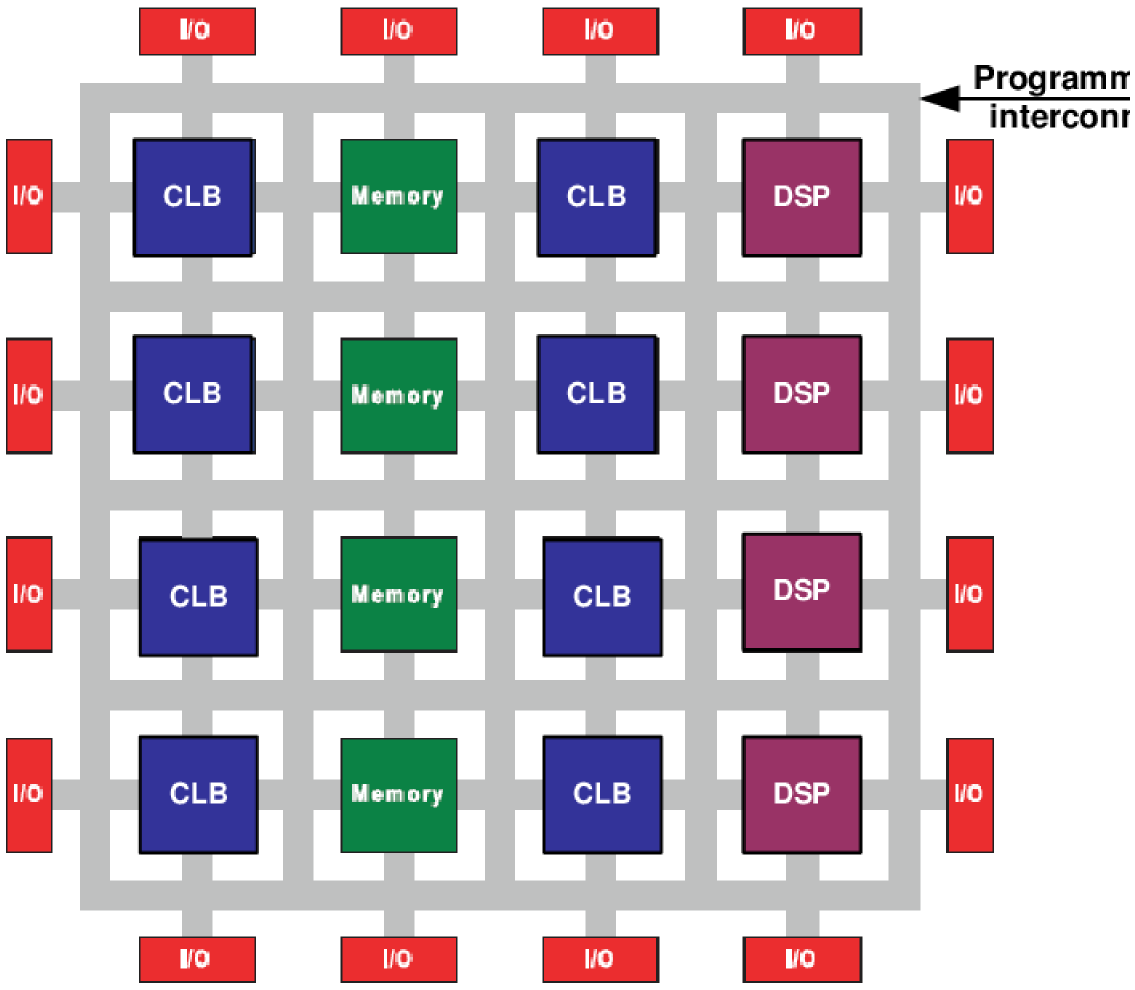

providing examples of when to use each.
Compare and contrast the architectures of CPUs, GPUs, and FPGA focusing on their
suitability for machine learning tasks.
of parallel computer architecture

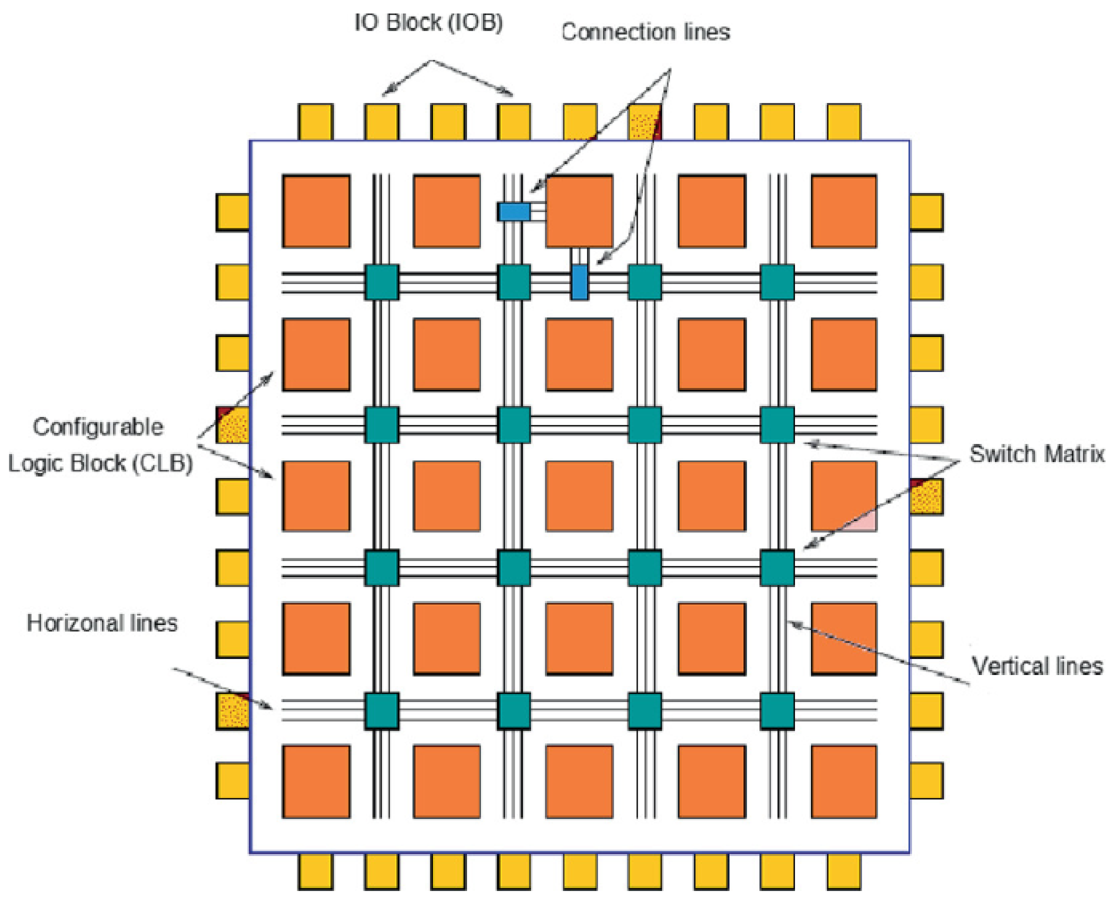

Enumerate the three steps to the kernels parallelization process in OpenACC	03

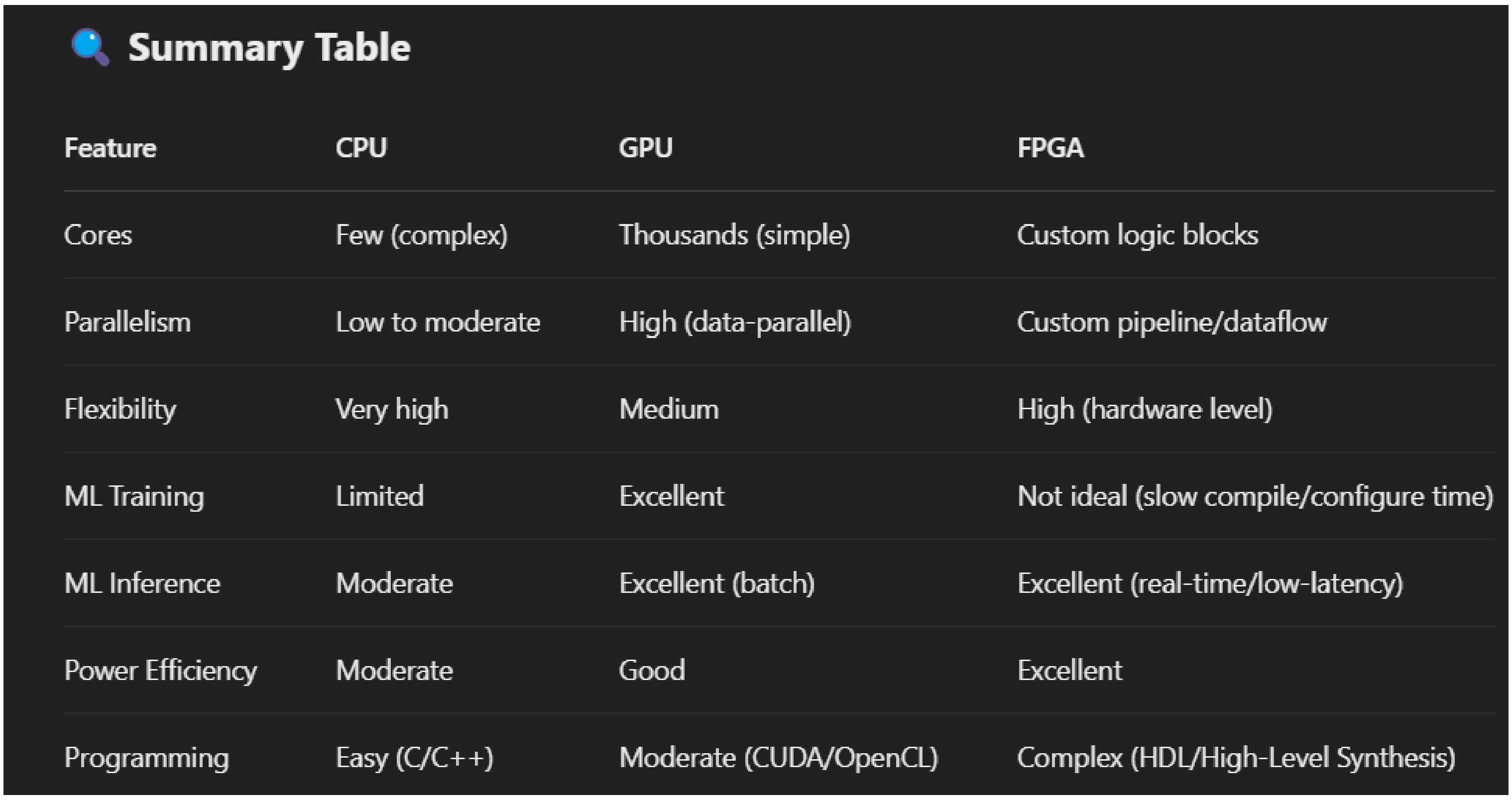
Identify the three levels of parallelism in OpenACC execution model.	03

Briefly Discuss FPGAs (Field Programmable Gate Arrays) with their benefits

FPGA (Field Programmable Gate Array)

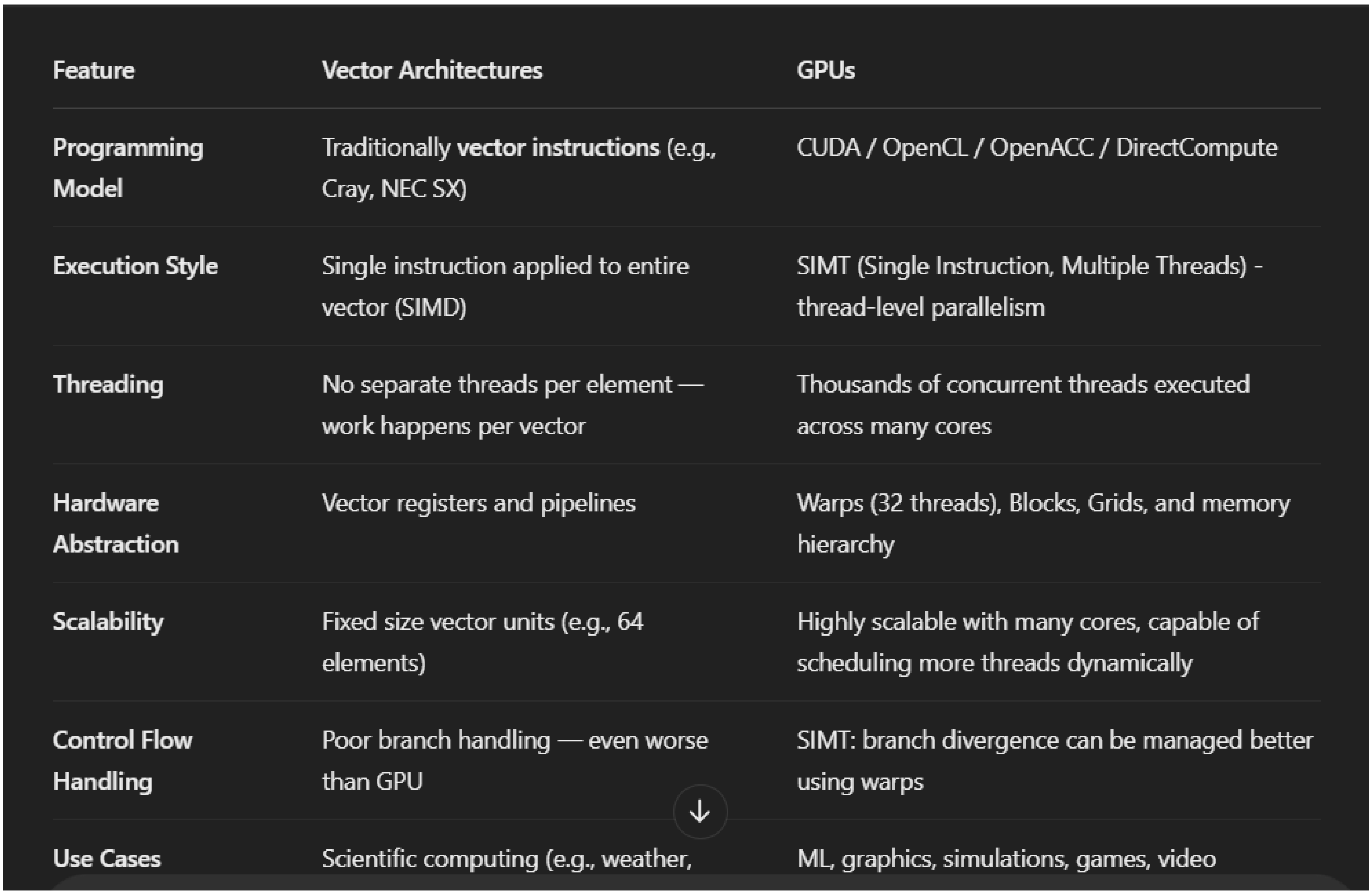






🔍 Summary Table

Feature	CPU	GPU	FPGA
Cores	Few (complex)	Thousands (simple)	Custom logic blocks
Parallelism	Low to moderate	High (data-parallel)	Custom pipeline/dataflow
Flexibility	Very high	Medium	High (hardware level)
ML Training	Limited	Excellent	Not ideal (slow compile/configure time)
ML Inference	Moderate	Excellent (batch)	Excellent (real-time/low-latency)
Power Efficiency	Moderate	Good	Excellent
Programming	Easy (C/C++)	Moderate (CUDA/OpenCL)	Complex (HDL/High-Level Synthesis)



Feature	Vector Architectures	GPUs
Programming Model	Traditionally **vector instructions** (e.g., Cray, NEC SX)	CUDA / OpenCL / OpenACC / DirectCompute
Execution Style	Single instruction applied to entire vector (SIMD)	SIMT (Single Instruction, Multiple Threads) - thread-level parallelism
Threading	No separate threads per element — work happens per vector	Thousands of concurrent threads executed across many cores
Hardware Abstraction	Vector registers and pipelines	Warps (32 threads), Blocks, Grids, and memory hierarchy
Scalability	Fixed size vector units (e.g., 64 elements)	Highly scalable with many cores, capable of scheduling more threads dynamically
Control Flow Handling	Poor branch handling — even worse than GPU	SIMT: branch divergence can be managed better using warps
Use Cases	Scientific computing (e.g., weather,	ML, graphics, simulations, games, video