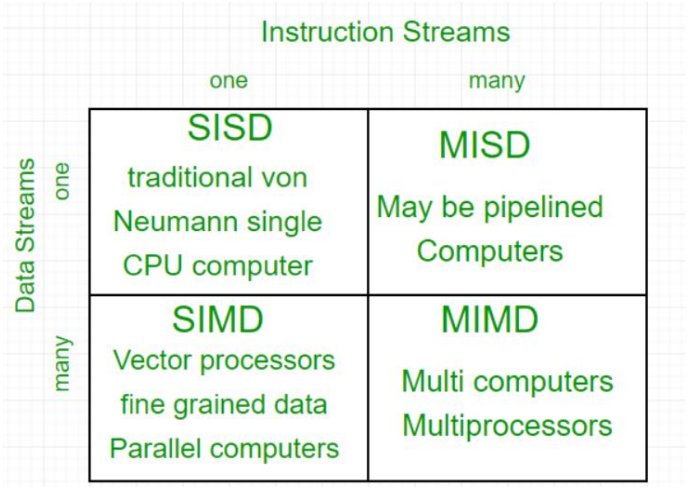
Unit 2

Computer Architecture

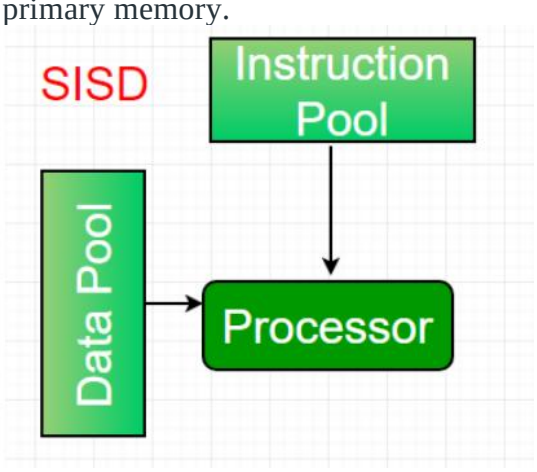
Flynn's taxonomy

Based on the number of **instructions**, **and data** streams that can be processed simultaneously, computing systems are classified into four major categories:



Single-instruction, single-data (SISD) systems

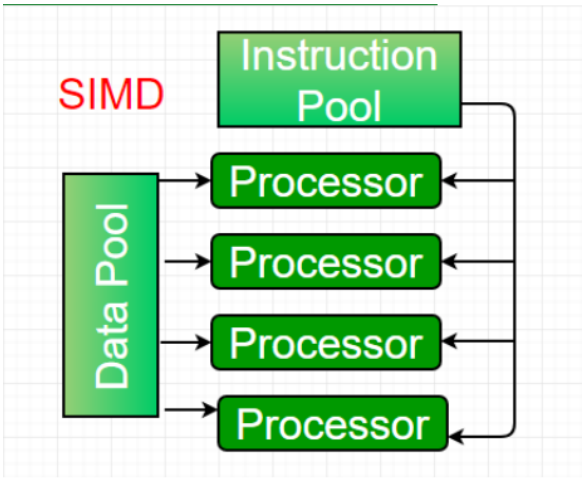
An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Most conventional computers have SISD architecture. All the instructions and data to be processed have to be stored in primary memory.



Single-instruction, multiple-data (SIMD) systems

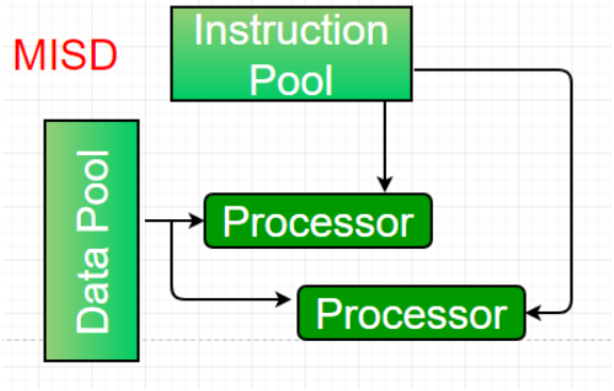
An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the

information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set.



Multiple-instruction, single-data (MISD) systems

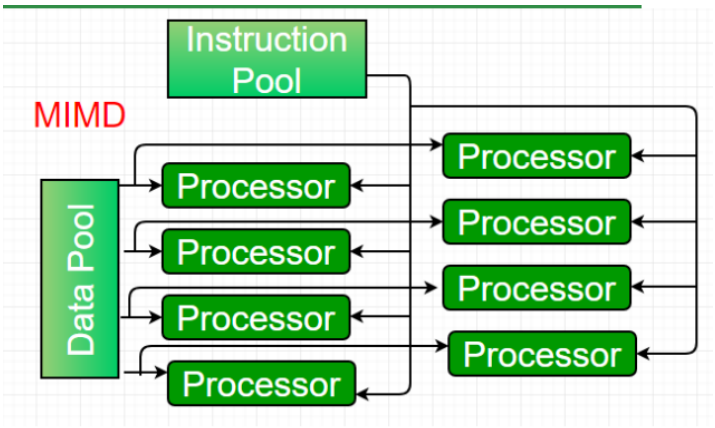
An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .



Example Z = sin(x)+cos(x)+tan(x) The system performs different operations on the same data set. Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.

Multiple-instruction, multiple-data (MIMD) systems

An MIMD system is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.

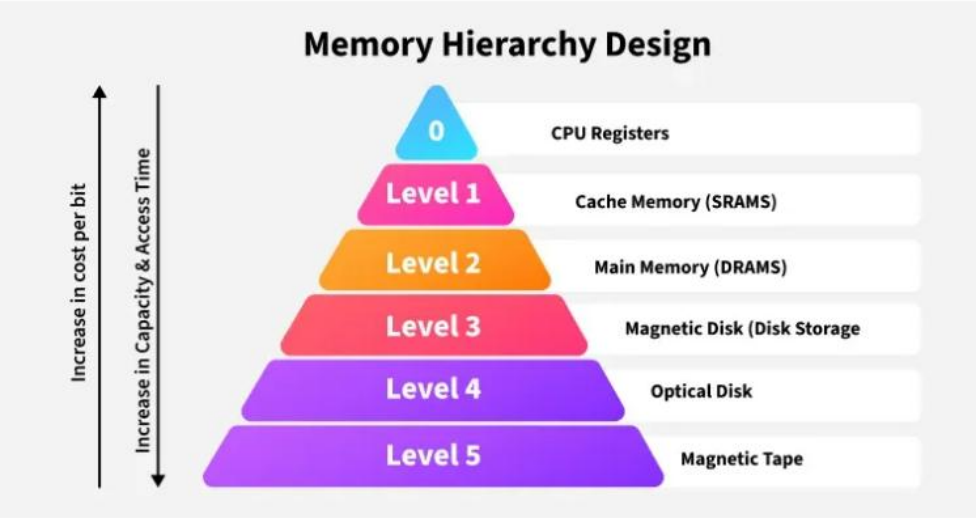


Memory Hierarchy

Memory Hierarchy helps in optimizing the memory available in the computer. There are multiple levels present in the memory, each one having a different size, different cost, etc.

Types of Memory Hierarchy
This Memory Hierarchy Design is divided into 2 main types:

- **External Memory or Secondary Memory:** Comprising of Magnetic Disk, Optical Disk, and Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via an I/O Module.
- **Internal Memory or Primary Memory:** Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.



1. Registers
Registers are small, high-speed memory units located in the CPU. They are used to store the most frequently used data and instructions. Registers have the fastest access time and the smallest storage capacity, typically ranging from 16 to 64 bits.

2. Cache Memory

Cache memory is a small, fast memory unit located close to the CPU. It stores frequently used data and instructions that have been recently accessed from the main memory. Cache memory is designed to minimize the time it takes to access data by providing the CPU with quick access to frequently used data.

3. Main Memory

Main memory, also known as RAM (Random Access Memory), is the primary memory of a computer system. It has a larger storage capacity than cache memory, but it is slower. Main memory is used to store data and instructions that are currently in use by the CPU.

Types of Main Memory

- **Static RAM:** Static RAM stores the binary information in flip flops and information remains valid until power is supplied. Static RAM has a faster access time and is used in implementing cache memory.
- **Dynamic RAM:** It stores the binary information as a charge on the capacitor. It requires refreshing circuitry to maintain the charge on the capacitors after a few milliseconds. It contains more memory cells per unit area as compared to SRAM.

4. Secondary Storage

Secondary storage, such as hard disk drives (HDD) and solid-state drives (SSD) , is a non-volatile memory unit that has a larger storage capacity than main memory. It is used to store data and instructions that are not currently in use by the CPU. Secondary storage has the slowest access time and is typically the least expensive type of memory in the memory hierarchy.

5. Magnetic Disk

Magnetic Disks are simply circular plates that are fabricated with either a metal or a plastic or a magnetized material. The Magnetic disks work at a high speed inside the computer and these are frequently used.

6. Magnetic Tape

Magnetic Tape is simply a magnetic recording device that is covered with a plastic film. Magnetic Tape is generally used for the backup of data. In the case of a magnetic tape, the access time for a computer is a little slower and therefore, it requires some amount of time for accessing the strip.

Characteristics of Memory Hierarchy

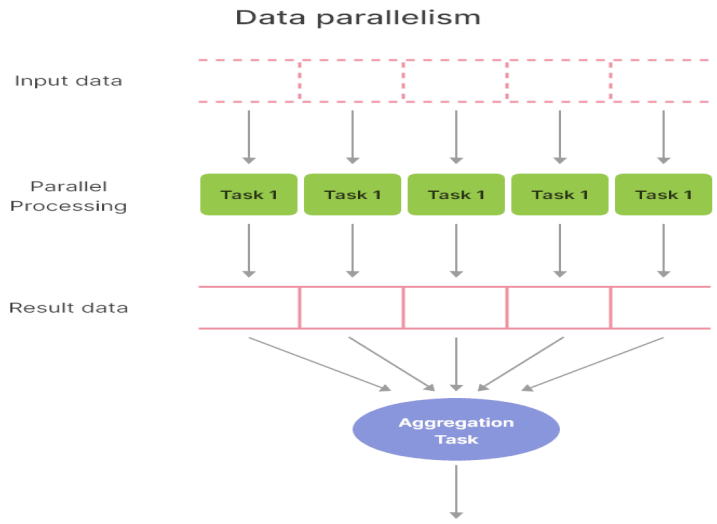
- **Capacity:** It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.
- **Access Time:** It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.
- **Performance:** The Memory Hierarch design ensures that frequently accessed data is stored in faster memory to improve system performance.
- **Cost Per Bit:** As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Data parallelism

Data parallelism is a parallel computing paradigm in which a large task is divided into smaller, independent, simultaneously processed subtasks. Via this approach, different processors or

computing units perform the same operation on multiple pieces of data at the same time. The primary goal of data parallelism is to improve computational efficiency and speed.

Let's take an example, summing the contents of an array of size N. For a single-core system, one thread would simply sum the elements [0] . . . [N − 1]. For a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1] and while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1]. So the Two threads would be running in parallel on separate computing cores.



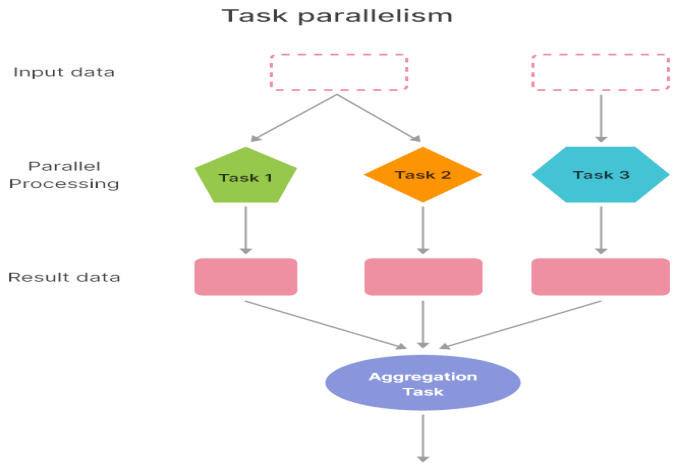
Data parallelism works by:

1. **Dividing data into chunks**: The first step in data parallelism is breaking down a large data set into smaller, manageable chunks. This division can be based on various criteria, such as dividing rows of a matrix or segments of an array.
2. **Distributed processing**: Once the data is divided into chunks, each chunk is assigned to a separate processor or thread. This distribution allows for parallel processing, with each processor independently working on its allocated portion of the data.
3. **Simultaneous processing**: Multiple processors or threads work on their respective chunks simultaneously. This simultaneous processing enables a significant reduction in the overall computation time, as different portions of the data are processed concurrently.
4. **Operation replication**: The same operation or set of operations is applied to each chunk independently. This ensures that the results are consistent across all processed chunks. Common operations include mathematical computations, transformations, or other tasks that can be parallelized.
5. **Aggregation**: After processing their chunks, the results are aggregated or combined to obtain the final output. The aggregation step might involve summing, averaging, or otherwise combining the individual results from each processed chunk

Task Parallelism

Task parallelism is a parallel programming paradigm where different tasks (functions, operations, or threads) are executed concurrently on multiple processing elements. Unlike data parallelism, where the same operation is applied to different chunks of data, in task parallelism different operations are applied to either the same or different sets of data.

Consider an example of task parallelism, where two threads each perform a unique statistical operation on the same array of elements. The threads run in parallel on separate computing cores, but each carries out a different operation.



Steps in Task Parallelism

1. **Task Identification**: The first step is identifying different tasks or operations that can be executed concurrently. Each task may perform a unique function, such as filtering, sorting, or statistical computation.
2. **Task Assignment**: Once tasks are identified, they are assigned to separate processors or threads. Each processor/thread is responsible for performing a distinct operation, either on the same dataset or on different datasets.
3. **Simultaneous Execution**: Multiple processors or threads execute their assigned tasks at the same time. This concurrency allows the system to complete complex workflows faster, as independent tasks are carried out in parallel.
4. **Operation Diversity**: Unlike data parallelism, here the tasks are not identical. Each processor/thread can perform a different operation, making task parallelism suitable for heterogeneous workloads such as pipelines or multi-stage processing.
5. **Result Combination**: After execution, the outputs of different tasks are combined (if needed) to form the final result. In some cases, tasks may feed results into each other (pipeline), while in others they may independently contribute to the overall output.

Data Parallelisms	Task Parallelisms
1. Same task are performed on different subsets of same data.	1. Different task are performed on the same or different data.
2. Synchronous computation is performed.	2. Asynchronous computation is performed.
3. As there is only one execution thread operating on all sets of data, so the speedup is more.	3. As each processor will execute a different thread or process on the same or different set of data, so speedup is less.
4. Amount of parallelization is proportional to the input size.	4. Amount of parallelization is proportional to the number of independent tasks is performed.
5. It is designed for optimum load balance on multiprocessor system.	5. Here, load balancing depends upon on the e availability of the hardware and scheduling algorithms like static and dynamic scheduling.

Bit-level Parallelism

Bit-level parallelism is a form of parallel computing that increases a processor's efficiency by allowing it to process **more bits per instruction**. It is directly related to the **word size (bit size)** of the processor, such as 8-bit, 16-bit, 32-bit, or 64-bit.

The main idea is:

- A processor with a **larger word size** can handle larger chunks of data at once.
- This reduces the **number of instructions** needed to perform computations on large data values.
- As the processor size increases, tasks that once required multiple steps can now be completed in fewer steps.

How It Works

1. **Processor Word Size**

   The word size is the number of bits a processor can handle in one operation.

   Example: An 8-bit processor can handle 8 bits at a time, while a 16-bit processor can handle 16 bits.

2. **Instruction Reduction**

   With a smaller word size, large data must be split into multiple chunks, requiring multiple instructions.

With a larger word size, the same operation can be performed in fewer instructions because the processor can handle more bits simultaneously.

3. **Performance Improvement**

   Fewer instructions mean less overhead and faster execution.

   This is especially useful in arithmetic operations, data transfers, and logical operations.

Example

- Suppose we want to add two **16-bit integers**:
- On an **8-bit processor**:
    - Step 1: Add the **lower 8 bits** of both numbers.
    - Step 2: Add the **higher 8 bits** of both numbers (including carry, if any).
    - Requires **2 instructions**.

On a **16-bit processor**:

- Step 1: Add all **16 bits** at once.
- Requires **1 instruction**.

This demonstrates how increasing the processor word size improves efficiency.

Applications of Bit-level Parallelism

Bit-level parallelism is essential in various fields that require efficient data processing and high-speed computation:

- **Cryptography:** Efficient handling of large bit-lengths improves encryption and decryption speeds.
- **Digital Signal Processing (DSP)**: Processes like audio and image compression benefit from parallel data handling.
- **High-performance Computing:** Scientific simulations and complex numerical algorithms leverage increased word sizes for faster calculations.
- **Graphics Processing:** Enhances rendering and processing of high-definition images and videos.
- **Database Management:** Bit-level parallelism aids in faster querying and data manipulation.

Instruction Level Parallelism

Instruction-Level Parallelism refers to the ability of a processor to execute multiple instructions simultaneously within a single CPU cycle. Instead of running one instruction after another

(sequential execution), ILP allows overlapping or parallel execution of instructions, improving overall performance.

Instruction Level Parallelism (ILP) is used to refer to the architecture in which multiple operations can be performed parallelly in a particular process, with its own set of resources - address space, registers, identifiers, state, and program counters.

It refers to the compiler design techniques and processors designed to execute operations, like memory load and store, integer addition, and float multiplication, in parallel to improve the performance of the processors.

Classification of ILP Architectures

The classification of ILP architectures can be done in the following ways -

- **Sequential Architecture:** Here, the program is not expected to explicitly convey any information regarding parallelism to hardware, like superscalar architecture.
- **Dependence Architectures:** Here, the program explicitly mentions information regarding dependencies between operations like dataflow architecture.
- **Independence Architecture:** Here, programme m gives information regarding which operations are independent of each other so that they can be executed instead of the 'nops.

Advantages of Instruction-Level Parallelism

- **Improved Performance:** ILP can significantly improve the performance of processors by allowing multiple instructions to be executed simultaneously or out-of-order. This can lead to faster program execution and better system throughput.
- **Efficient Resource Utilization:** ILP can help to efficiently utilize processor resources by allowing multiple instructions to be executed at the same time. This can help to reduce resource wastage and increase efficiency.
- **Reduced Instruction Dependency:** ILP can help to reduce the number of instruction dependencies, which can limit the amount of instruction-level parallelism that can be exploited. This can help to improve performance and reduce bottlenecks.
- **Increased Throughput:** ILP can help to increase the overall throughput of processors by allowing multiple instructions to be executed simultaneously or out-of-order. This can help to improve the performance of multi-threaded applications and other parallel processing tasks.

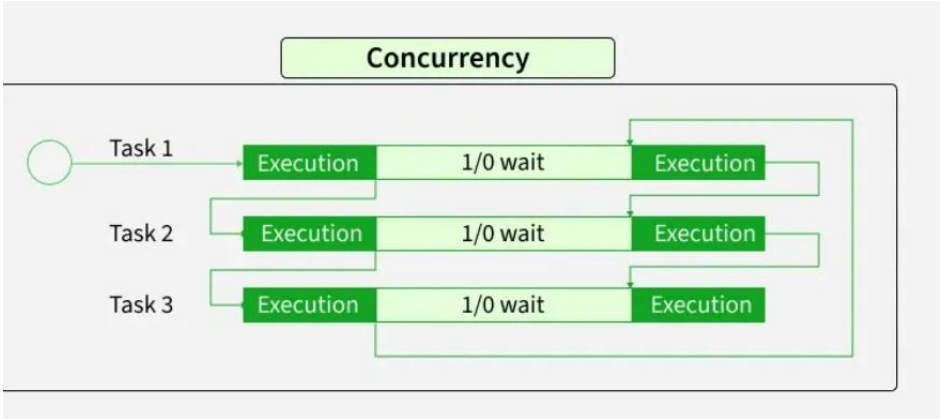
Concurrency

Concurrency refers to the ability of a system to **manage multiple tasks at the same time**. It does not necessarily mean that tasks are running *literally* in parallel (simultaneously on different processors). Instead, concurrency means the system **interleaves execution** of multiple tasks to give the *illusion* that they are running together.

Concurrency refers to an application that is doing many tasks at the same time. Concurrency is a technique for reducing system response time by utilizing a single processing unit. Concurrency offers the illusion of parallelism, however, the chunks of a job aren't performed in parallel, and inside the application, more than one task is being executed at the same time. It does not complete one work before beginning the next.

Key Characteristics:

- Tasks start, run and complete in overlapping time periods.
- A single processor may switch between tasks quickly (context switching), giving the illusion of simultaneous execution.
- Common in environments where responsiveness is important (e.g., handling multiple user requests).



Here, first Task 1 is executing then it went to I/O stage, during this Task 2 starts executing and then it also went to I/O stage and then task 3 and so on. Finally task 1 finish its I/O stage and starts executing remaining part, Task 2 follows it and then Task 3.
Example: A single-core CPU running multiple threads: the CPU rapidly switches between threads so each makes progress.

Concurrency is achieved by the interleaving of processes on the central processing unit (CPU), also known as context switching. That is the logic behind parallel processing. It enhances the quantity of work that can be completed at one time.

Concurrency in computing means handling **multiple tasks at once**. There are **three basic kinds of concurrent computing techniques**:

- **Threading:** A concurrency technique where a program is divided into multiple threads that run independently but share the same memory space.
- **Asynchrony:** A concurrency technique where tasks are executed without waiting for each other, allowing the program to continue other work while some tasks run in the background.
- **Pre-emptive Multitasking:** A concurrency technique where the operating system interrupts tasks and allocates CPU time to multiple processes or threads, giving the illusion that they run simultaneously.

Concurrency	Parallelism
Concurrency is the problem of simultaneously conducting and managing numerous computations.	Parallelism is the challenge of conducting numerous calculations at the same time.
Concurrency is achieved by the interleaving of processes on the central processing unit (CPU), also known as context switching.	While it is accomplished through the use of several central processing units (CPUs).
Concurrency can be accomplished by utilising a single processing unit.	While this cannot be accomplished with a single processing unit. It necessitates the use of numerous processing units.
Concurrency increases the quantity of work that may be completed at one moment.	While increasing the system's throughput and processing performance.
Concurrency deals with many things at the same time.	While it does a number of things at the same time.
The non-deterministic control flow technique is called concurrency.	While the method is deterministic control flow.
Debugging in concurrency is extremely difficult.	While debugging is difficult, it is simpler than concurrency.

Decomposition in Parallel Computing

When a computer uses multiprocessing (multiple processors/cores), a large problem must be broken down into smaller subproblems so that each processor can work on a part of it.

This process is called **Problem Decomposition**.

- It splits a problem into **subproblems (or subprograms)**.
- Each processor works on its assigned subproblem **concurrently**.
- It is the **foundation of parallel computing**.

Types of Problem Decomposition

The two most common techniques are Domain Decomposition and Functional Decomposition.

1. Domain Decomposition (Data Decomposition)

Domain decomposition, also known as data decomposition, focuses on splitting the data into smaller parts while applying the same computation to each part. Every processor works on a different portion of the data set but executes the same operation.

Example:
Suppose we want to apply a filter to a large digital image. Instead of processing the entire image

on one processor, the image is divided into four equal blocks: top-left, top-right, bottom-left, and bottom-right. Each processor is assigned one block and applies the same filter operation, such as edge detection or blurring. Once all four processors finish their part, the results are merged to form the final processed image.

This method is particularly useful in problems like numerical simulations, weather prediction, matrix computations, and image processing, where the same operation needs to be repeated over large datasets.

2. Functional Decomposition (Task Decomposition)

Functional decomposition, also known as task decomposition, focuses on splitting the functions or tasks of a program rather than splitting the data. Each processor performs a different operation on the same or related data. This approach is suitable when the workload consists of distinct operations that can be executed concurrently.

Example:
Consider a video processing system. The program can be divided into multiple stages:

- Processor 1 reads video frames from the input device (e.g., camera or disk).
- Processor 2 decodes the compressed video stream.
- Processor 3 applies filters such as color correction or noise reduction.
- Processor 4 compresses or renders the processed video for display or storage.

Here, each processor is performing a different task but contributes to the same overall goal of processing the video. This pipeline-like arrangement is an excellent example of functional decomposition.

Mapping in Parallel Programming

Once a problem has been decomposed into smaller tasks or data chunks, the next important step in parallel programming is Mapping. Mapping refers to the process of assigning these decomposed tasks or data units to processing elements (PEs) such as CPU cores, GPU threads, or even nodes in a distributed cluster.

The efficiency of mapping is crucial because it determines how well the workload is balanced across processors. A good mapping strategy ensures that all processors are kept busy, no processor is overloaded or idle, and execution time is minimized.

There are two main types of mapping techniques: Static Mapping and Dynamic Mapping.

1. Static Mapping (Compile-Time Mapping): In static mapping, the assignment of tasks to processors is done before execution begins (at compile time).Each task is permanently bound to a particular processor, and the assignment does not change during execution.

Example: Suppose we want to multiply two large matrices using four processors. The matrices are divided into four equal blocks, and each processor is assigned one block to compute from start to finish. Since the workload is evenly distributed, static mapping works efficiently here.

2. Dynamic Mapping (Run-Time Mapping): In dynamic mapping, the assignment of tasks to processors is done at run time while the program is executing. Tasks are placed in a task pool, and processors request new tasks when they become free. This allows flexible assignment depending on current system load.

Characteristics:

Adaptive to load imbalance (some tasks may take longer than others).
Can handle unpredictable situations like processor failures or unequal workloads.
Common in distributed computing and heterogeneous systems.

Example: Imagine a distributed system running scientific simulations where some tasks (e.g., simulating complex regions) take more time than others. With dynamic mapping, as soon as a processor finishes one task, it fetches another from the task pool. This prevents faster processors from staying idle while others are still working.

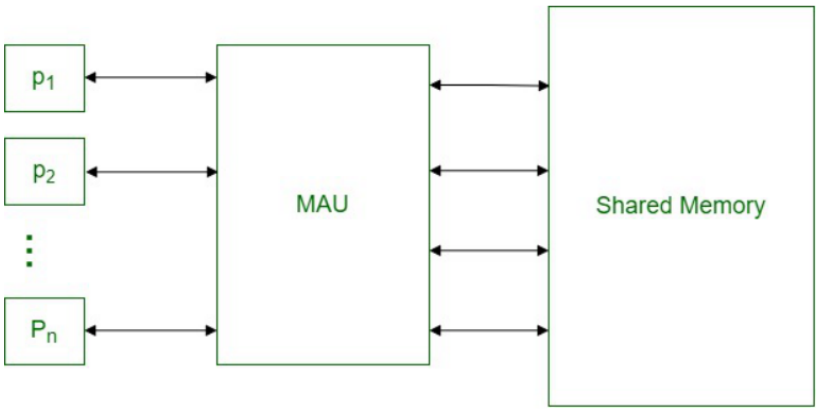
PRAM or Parallel Random Access Machines

Parallel Random Access Machine, also called **PRAM** is a model considered for most of the parallel algorithms. It helps to write a precursor parallel algorithm without any architecture constraints and also allows parallel-algorithm designers to treat processing power as unlimited. It ignores the complexity of inter-process communication. PRAM algorithms are mostly theoretical but can be used as a basis for developing an efficient parallel algorithm for practical machines and can also motivate building specialized machines.

PRAM Architecture Model

The following are the modules of which a PRAM consists of:

1. It consists of a control unit, global memory, and an unbounded set of similar processors, each with its own private memory.
2. An active processor reads from global memory, performs required computation, and then writes to global memory.
3. Therefore, if there are N processors in a PRAM, then N number of independent operations can be performed in a particular unit of time.



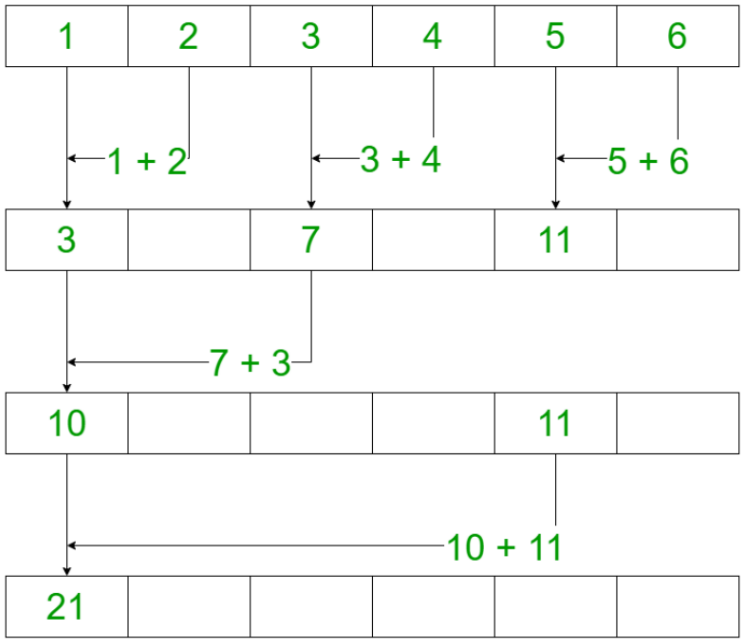
Parallel Random Access Machines

Models of PRAM

While accessing the shared memory, there can be conflicts while performing the read and write operation (i.e.), a processor can access a memory block that is already being accessed by another processor. Therefore, there are various constraints on a PRAM model which handles the read or write conflicts. They are:

- **EREW:** also called Exclusive Read Exclusive Write is a constraint that doesn't allow two processors to read or write from the same memory location at the same instance.
- **CREW:** also called Concurrent Read Exclusive Write is a constraint that allows all the processors to read from the same memory location but are not allowed to write into the same memory location at the same time.
- **ERCW:** also called Exclusive Read Concurrent Write is a constraint that allows all the processors to write to the same memory location but are now allowed to read the same memory location at the same time.
- **CRCW:** also called Concurrent Read Concurrent Write is a constraint that allows all the processors to read from and write to the same memory location parallelly.

Example: Suppose we wish to add an array consisting of N numbers. We generally iterate through the array and use N steps to find the sum of the array. So, if the size of the array is N and for each step, let's assume the time taken to be 1 second. Therefore, it takes N seconds to complete the iteration. The same operation can be performed more efficiently using a CRCW model of a PRAM. Let there be N/2 parallel processors for an array of size N, then the time taken for the execution is 4 which is less than N = 6 seconds in the following illustration.


1
2
3
4
5
6
1 + 2
3 + 4
5 + 6
3
7
11
7 + 3
10
11
10 + 11
21