Moore's Law Explanation

Moore's law states

Moore's Law is an observation made by Gordon Moore, the co-founder of Intel, in 1965. It states that:

"The number of transistors on a microchip doubles approximately every two years, while the cost of computers is halved."

This principle has driven exponential growth in computing power over decades, enabling advances in technology, from smaller devices to more powerful processors.

Although Moore's Law is not a physical law and is more of a trend, it has been a guiding principle for the semiconductor industry. In recent years, the pace of transistor miniaturization has slowed, leading to discussions about the eventual end of Moore's Law due to physical and economic limitations. However, innovations like multi-core processors and 3D chip designs continue to push the boundaries of computing performance.

5. What is Parallelism?
6. What is Concurrency?
7. Differentiate between Parallelism and Concurrency.
8. Types of parallelism.
9. Differentiate between explicit and implicit parallelism. explain each as a long answers

5. What is Parallelism?

Parallelism refers to the simultaneous execution of multiple tasks or processes to achieve faster computation. It involves dividing a problem into smaller subproblems, executing these subproblems concurrently, and then combining the results. Parallelism typically requires multiple processing units (e.g., CPUs, GPUs) or cores to execute tasks simultaneously.

For example:

- A GPU rendering multiple frames of a video simultaneously.
- Sorting different parts of a dataset at the same time using multiple threads.

---

6. What is Concurrency?

Concurrency is the ability of a system to handle multiple tasks or processes at the same time, not necessarily simultaneously. It focuses on managing the execution of tasks in such a way that they appear to run in parallel. Concurrency can occur on a single-core processor by interleaving the execution of tasks through scheduling.

For example:

- Running a web server that handles multiple client requests concurrently.
- Multithreading in an application where threads share a single processor core.

---

7. Differentiate between Parallelism and Concurrency

Aspect	Parallelism	Concurrency
Definition	Simultaneous execution of tasks on multiple cores/units.	Management of multiple tasks, not necessarily simultaneous.
Objective	Improve performance by dividing and executing tasks.	Efficiently manage resource usage and task execution.
Execution	Tasks run at the same time.	Tasks may run interleaved or partially overlapped.
Dependencies	Usually tasks are independent.	Tasks may have interdependencies.
Example	Processing a large dataset on a multi-core system.	Handling multiple requests on a single-threaded web server.
Hardware Dependency	Requires multiple cores or processors.	Can run on a single processor with scheduling.

---

8. Types of Parallelism

1. **Data Parallelism**: Distributes subsets of data across multiple processors and performs the same operation on each subset.
   Example: Applying filters to sections of an image simultaneously.
2. **Task Parallelism**: Distributes different tasks or functions across processors.
   Example: One core handles data reading while another processes computations.
3. **Bit-level Parallelism**: Executes operations on multiple bits of data simultaneously.
   Example: Optimizing bitwise operations in processors.
4. **Instruction-level Parallelism (ILP)**: Overlaps execution of instructions within a CPU pipeline.
   Example: A processor executing multiple instructions in a single clock cycle.
5. **Pipeline Parallelism**: Breaks a task into stages and executes each stage on a different processor.
   Example: Assembly line in manufacturing.

---

9. Differentiate Between Explicit and Implicit Parallelism

Aspect	Explicit Parallelism	Implicit Parallelism
Definition	Programmer explicitly defines parallel tasks and controls synchronization.	Compiler/system handles parallelization automatically.
Control	Requires manual control by the programmer.	Automated by the compiler/runtime.
Ease of Use	Complex to implement due to the need for manual intervention.	Easier for developers as the system abstracts complexity.
Flexibility	Offers greater control over parallel execution.	Limited control; relies on the system's efficiency.
Example	Using multithreading libraries (e.g., OpenMP, MPI).	Automatic parallelization in high-level languages like Python's Numpy.

Explanation

- **Explicit Parallelism**:
  In this approach, the programmer must identify parallelizable parts of a program and use tools like threads, libraries, or APIs to execute them in parallel. The programmer is responsible for managing synchronization, task distribution, and potential conflicts.
  Example: Writing a multithreaded program using pthreads to calculate matrix multiplication.
- **Implicit Parallelism**:
  Here, the compiler or runtime system detects opportunities for parallel execution without programmer intervention. It automatically optimizes the code to take advantage of multi-core processors. This approach abstracts complexity but may not be as efficient for specialized use cases.
  Example: A modern compiler that breaks a loop into parallelizable chunks during optimization.

Whatarethe different types of parallelism?
11.
Explain Instruction level parallelism.

Types of Parallelism

Parallelism can be categorized based on how tasks are distributed and executed. The main types include:

1. **Data Parallelism**:
   - Focuses on distributing data across multiple processors or cores.
   - Each processor performs the same operation on its subset of data simultaneously.
   - **Example**: Applying transformations to different parts of an image.
2. **Task Parallelism**:
   - Different tasks or operations are executed simultaneously on separate processors or cores.

- Tasks are independent but may communicate or synchronize.
- **Example**: One processor handles input data while another performs calculations.

3. **Instruction-Level Parallelism (ILP)**:
   - Overlaps the execution of multiple instructions in a single program within a processor.
   - Achieved using techniques like pipelining, superscalar execution, and out-of-order execution.
4. **Bit-Level Parallelism**:
   - Involves performing operations on multiple bits simultaneously within a single instruction.
   - Common in low-level hardware optimizations.
   - **Example**: Arithmetic operations on 64-bit registers.
5. **Pipeline Parallelism**:
   - Tasks are broken into smaller stages, and each stage is executed concurrently in a pipeline fashion.
   - Often used in assembly lines, graphics rendering, and stream processing.
   - **Example**: In CPUs, instruction fetch, decode, and execute stages.
6. **Thread-Level Parallelism**:
   - Involves running multiple threads concurrently, either in the same or different programs.
   - Exploited in multi-threaded applications.
   - **Example**: A web browser loading multiple tabs concurrently.
7. **Process-Level Parallelism**:
   - Multiple processes are executed in parallel, often on separate systems or cores.
   - Used in distributed systems or parallelized applications.
   - **Example**: Cluster computing for large datasets.

---

11. Explain Instruction-Level Parallelism (ILP)

Instruction-Level Parallelism (ILP) is a form of parallelism where a CPU executes multiple instructions simultaneously. It focuses on exploiting opportunities within a single program to overlap the execution of instructions.

Key Concepts in ILP:

1. **Pipelining**:
   - Instructions are divided into stages (e.g., fetch, decode, execute).
   - Multiple instructions are processed simultaneously in different stages of the pipeline.
   - **Example**: While one instruction is being executed, another is being decoded, and a third is being fetched.
2. **Superscalar Architecture**:
   - CPUs have multiple execution units (e.g., arithmetic logic units, floating-point units).
   - Allows multiple instructions to execute in parallel within a single clock cycle.
   - **Example**: Performing addition and multiplication at the same time.
3. **Out-of-Order Execution**:
   - Instructions are executed as soon as their operands are available, even if earlier instructions are delayed.
   - **Example**: Executing an independent instruction before a stalled one.

4. **Branch Prediction**:
    - Predicts the outcome of conditional instructions (e.g., `if` statements) to keep the pipeline full.
    - Reduces delays caused by branching.
5. **Loop Unrolling**:
    - Reduces overhead by increasing the number of instructions executed per iteration.
    - **Example**: Converting a loop with 10 iterations into 2 loops with 5 instructions each.

Benefits of ILP:

- Increases CPU efficiency by maximizing the utilization of execution units.
- Reduces idle time between instructions, improving performance.

Challenges of ILP:

- **Data Dependencies**: Instructions may depend on the results of previous instructions.
- **Control Dependencies**: Branches can disrupt the instruction pipeline.
- **Hardware Complexity**: Advanced techniques like out-of-order execution require sophisticated hardware.

Example of ILP in a Pipelined Processor:

Consider the following instructions:

1. `A = B + C` (Fetch Stage)
2. `D = A * E` (Decode Stage)
3. `F = G - H` (Execute Stage)

Each instruction is at a different stage of execution, allowing the CPU to work on all three simultaneously.

Explain the concept of pipelining.
13. Explain superscalar architecture.
14. Differentiate between Superscalar and Pipelining
15. Explain VLIW architecture

12. Explain the Concept of Pipelining

Pipelining is a technique used in computer architecture to improve the throughput of a CPU by overlapping the execution of instructions. Instead of executing one instruction at a time, the processor breaks each instruction into smaller stages and executes multiple stages concurrently.

Stages of a Typical Pipeline:

1. **Fetch**: Retrieve the instruction from memory.
2. **Decode**: Interpret the instruction and prepare operands.
3. **Execute**: Perform the operation (e.g., arithmetic, logic).
4. **Memory Access**: Read/write data to/from memory.
5. **Write Back**: Store the result back to a register.

Key Features:

- Each stage works on a different instruction at the same time.
- New instructions are "pushed" into the pipeline as previous ones progress.

Advantages:

- Increases instruction throughput.
- Reduces idle time in the CPU.
- Efficient use of hardware resources.

Challenges:

- **Data Hazards**: Dependencies between instructions can cause delays.
- **Control Hazards**: Branches or jumps disrupt the pipeline flow.
- **Structural Hazards**: Resource conflicts when multiple stages need the same hardware.

---

13. Explain Superscalar Architecture

Superscalar Architecture is a CPU design approach that enables the simultaneous execution of multiple instructions within a single clock cycle. It uses multiple execution units to achieve parallelism at the instruction level.

Key Characteristics:

1. **Multiple Execution Units**:
   - Includes units for arithmetic, logic, floating-point, etc.
2. **Instruction Dispatch**:
   - Decodes and sends multiple instructions to available units in the same cycle.
3. **Dynamic Scheduling**:
   - Determines the order of instruction execution based on data availability.

Advantages:

- Increased performance by leveraging instruction-level parallelism.
- More efficient hardware utilization.

Challenges:

- Complex hardware to handle dependencies and conflicts.
- Requires advanced branch prediction and out-of-order execution.

Example:

A CPU with two ALUs can simultaneously execute `A = B + C` and `D = E - F` if there are no dependencies between them.

---

14. Differentiate Between Superscalar and Pipelining

Aspect	Pipelining	Superscalar Architecture
Concept	Overlaps different stages of instruction execution.	Executes multiple instructions simultaneously in one cycle.
Execution Units	One instruction processed per stage at a time.	Multiple execution units for parallel instruction execution.
Objective	Increases throughput by dividing instructions into stages.	Increases throughput by executing instructions in parallel.
Dependencies	Focuses on sequential execution with overlap.	Handles multiple independent instructions concurrently.
Hardware Complexity	Relatively simpler hardware.	More complex hardware with additional execution units.
Example	Assembly line processing in a factory.	Multiple workers performing different tasks simultaneously.

---

15. Explain VLIW Architecture

VLIW (Very Long Instruction Word) architecture is a computer design approach that executes multiple operations simultaneously by packing them into a single, long instruction word. Each word contains instructions for different functional units, and all operations in the word are executed in parallel.

Key Features:

1. **Fixed Instruction Format**:
   - Each instruction word specifies operations for multiple execution units.
2. **Compiler-Driven Parallelism**:
   - The compiler determines which operations can be executed in parallel and schedules them in the instruction word.
3. **Simple Hardware**:
   - Unlike superscalar CPUs, VLIW relies on the compiler rather than complex hardware for instruction scheduling.

Advantages:

- Simplifies CPU design as no dynamic scheduling is needed.
- High performance for parallelizable workloads.

Challenges:

- Increased burden on the compiler for parallelism detection.

- Wasted execution slots if parallelism is not fully exploited.

Example:

A VLIW instruction might include:

- Add two numbers in one functional unit.
- Multiply two numbers in another unit.
- Load data from memory in a third unit.

This ensures all units work concurrently, provided the compiler schedules the instructions effectively.

Define decomposition.
17. Define granularity.
18. Typesof granularity

16. Define Decomposition

Decomposition in parallel computing refers to the process of dividing a computational task into smaller, more manageable sub-tasks that can be executed independently or concurrently. The goal of decomposition is to exploit parallelism, improve efficiency, and reduce the overall execution time.

Types of Decomposition:

1. **Task Decomposition**: Breaking the computation into smaller tasks that can be executed independently.
   Example: Dividing a software application into modules for different functionalities.
2. **Data Decomposition**: Dividing the data into smaller chunks and applying the same operation on each chunk concurrently.
   Example: Splitting an image into parts and applying filters to each part simultaneously.

Benefits:

- Enables parallel execution.
- Improves scalability and resource utilization.
- Simplifies problem-solving by focusing on smaller tasks.

---

17. Define Granularity

Granularity refers to the size or amount of computation involved in a single task or process within a parallel computing system. It measures the ratio of computation to communication.

Key Factors:

- **Fine-Grained Tasks**: Small tasks with minimal computation but frequent communication.
- **Coarse-Grained Tasks**: Larger tasks with significant computation and less communication.

Impact of Granularity:

- Fine granularity requires efficient communication mechanisms to avoid overhead.
- Coarse granularity is better suited for distributed systems with limited communication capabilities.

---

18. Types of Granularity

1. **Fine-Grained Parallelism**:
   - Tasks involve a small amount of computation and frequent communication.
   - **Example**: Element-wise operations in a matrix.
   - **Advantages**: High degree of parallelism.
   - **Disadvantages**: High communication overhead.
2. **Medium-Grained Parallelism**:
   - Balances the amount of computation and communication.
   - **Example**: Partitioning a matrix into submatrices and processing them in parallel.
   - **Advantages**: Better trade-off between parallelism and overhead.
3. **Coarse-Grained Parallelism**:
   - Tasks involve significant computation and infrequent communication.
   - **Example**: Distributing entire subroutines to different processors.
   - **Advantages**: Low communication overhead, suitable for distributed systems.
   - **Disadvantages**: Lower degree of parallelism compared to fine-grained.

Choosing the Right Granularity:

The choice of granularity depends on:

- **Hardware Capabilities**: Processors, network bandwidth, etc.
- **Task Dependencies**: Extent of communication required between tasks.
- **Performance Goals**: Balancing computation speed with communication overhead.

Explain the techniques of decomposition.

Techniques of Decomposition in Parallel Computing

Decomposition techniques are methods used to break down a computational problem into smaller subproblems that can be executed independently or concurrently. The choice of technique depends on

the nature of the problem and the goals of the computation.

---

1. Task Decomposition

- **Definition**: Divides the problem into distinct tasks or functions that can be performed independently.
- **Characteristics**:
  - Tasks may perform different operations.
  - Requires less communication between tasks.
- **Example**:
  - A weather simulation where one task calculates wind patterns and another computes temperature changes.

---

2. Data Decomposition

- **Definition**: Divides the data into smaller chunks and applies the same operation to each chunk in parallel.
- **Characteristics**:
  - Suitable for problems with large datasets.
  - Tasks are often similar but operate on different portions of data.
- **Example**:
  - Processing different parts of an image for filtering or compression.

Types of Data Decomposition:

1. **Block Decomposition**: Divides data into contiguous blocks.
   - Example: Splitting a matrix row-wise or column-wise.
2. **Cyclic Decomposition**: Distributes data elements cyclically across tasks.
   - Example: Distributing rows of a matrix in a round-robin fashion.

---

3. Recursive Decomposition

- **Definition**: Recursively breaks a problem into smaller subproblems until they are small enough to solve directly.
- **Characteristics**:
  - Suitable for problems that naturally divide into similar subproblems.
  - Often used in divide-and-conquer algorithms.
- **Example**:
  - Merge Sort or Quick Sort.

---

4. Pipeline Decomposition

- **Definition**: Divides the computation into a sequence of stages where each stage processes part of the input and passes the output to the next stage.
- **Characteristics**:
  - Suitable for problems with a sequential flow of data.

    - Tasks operate concurrently on different parts of the data.
- **Example**:
    - Video encoding: One stage compresses the frame, another applies filters, and another encodes the result.

---

5. Hybrid Decomposition

- **Definition**: Combines two or more decomposition techniques to optimize performance.
- **Characteristics**:
    - Useful for complex problems with both task and data parallelism.
- **Example**:
    - A computational fluid dynamics simulation that uses task decomposition for different physical phenomena (e.g., heat transfer, fluid flow) and data decomposition for partitioning the simulation grid.

---

6. Functional Decomposition

- **Definition**: Breaks a problem based on the functions or modules of a program.
- **Characteristics**:
    - Tasks correspond to different functionalities of the system.
    - Often involves task parallelism.
- **Example**:
    - A compiler: One task handles lexical analysis, another handles syntax analysis, and another performs code generation.

Choosing a Decomposition Technique

The appropriate technique depends on:

1. **Problem Characteristics**: Task dependencies, data size, and complexity.
2. **Hardware Resources**: Number of processors and memory availability.
3. **Performance Goals**: Minimize communication overhead and maximize load balancing.

Explain the different types of Mapping Techniques.
21. Write the techniques of Parallel Algorithm Model.
22. Whatdoyoumeanbydegree of concurrency, average degree of concurrency and critical path?

20. Different Types of Mapping Techniques

Mapping in parallel computing refers to assigning tasks or data to processors for execution. The goal is to balance the workload among processors, minimize communication overhead, and maximize

performance.

Types of Mapping Techniques

1. **Static Mapping**:
   - The assignment of tasks or data to processors is determined before execution and remains unchanged.
   - Suitable for systems with predictable workloads.
   - **Advantages**:
     - Simple to implement.
     - Minimal runtime overhead.
   - **Disadvantages**:
     - May lead to load imbalance if the workload changes.
   - **Example**:
     - Dividing a matrix row-wise among processors for matrix-vector multiplication.
2. **Dynamic Mapping**:
   - Tasks or data are assigned to processors during runtime based on current workload and system state.
   - Suitable for systems with unpredictable workloads.
   - **Advantages**:
     - Better load balancing.
     - Adapts to dynamic changes in workload.
   - **Disadvantages**:
     - Higher runtime overhead due to frequent reassignment.
   - **Example**:
     - A web server dynamically allocating incoming requests to available processors.
3. **Hierarchical Mapping**:
   - Combines static and dynamic mapping approaches.
   - Tasks are grouped and assigned to processor groups (static), and within groups, tasks are dynamically assigned.
   - **Advantages**:
     - Combines the benefits of static and dynamic mapping.
   - **Example**:
     - Distributed computing systems where tasks are assigned to clusters statically and nodes within the cluster handle them dynamically.

---

21. Techniques of Parallel Algorithm Models

Parallel algorithms are designed using specific models that dictate how tasks are structured and coordinated. The key techniques include:

1. **Data Parallelism Model**:
   - Focuses on distributing data across processors.
   - Each processor performs the same operation on its chunk of data.
   - **Example**: Vector addition where each processor adds a subset of vector elements.
2. **Task Parallelism Model**:

- Different tasks are assigned to different processors.
- Tasks may perform distinct computations and communicate as needed.
- **Example**: Sorting a dataset where one task partitions the data and others sort the partitions.

3. **Pipeline Model**:
   - Breaks the computation into stages.
   - Each stage processes a part of the data and passes the result to the next stage.
   - **Example**: Image processing where one stage filters, another compresses, and another stores.
4. **Hybrid Model**:
   - Combines data and task parallelism to optimize performance.
   - **Example**: Weather simulation where data is divided across processors, and each handles multiple aspects (temperature, pressure, etc.).
5. **Recursive Decomposition Model**:
   - Divides the problem into smaller subproblems recursively.
   - Solves subproblems concurrently and combines results.
   - **Example**: Recursive matrix multiplication.

---

22. Degree of Concurrency, Average Degree of Concurrency, and Critical Path

Degree of Concurrency:

- **Definition**: The number of tasks or operations that can be executed simultaneously at any point in a parallel program.
- **Importance**:
  - Indicates the potential parallelism of an algorithm.
  - A high degree of concurrency suggests better utilization of available processors.
- **Example**:
  - In a matrix multiplication program, multiple row-column multiplications can occur concurrently.

Average Degree of Concurrency:

- **Definition**: The average number of tasks that can run concurrently over the entire execution of the parallel program.
- **Formula**:

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

- **Example**:
  - If a program has 100 independent tasks over a critical path length of 10 steps, the average degree of concurrency is 10.

Critical Path:

- **Definition**: The longest sequence of dependent tasks in a parallel program that determines the minimum execution time.
- **Significance**:

    - Represents the bottleneck of the program.
    - Reducing the critical path can significantly improve performance.
- **Example**:
    - In a task graph, the critical path is the path with the maximum weight (time or dependencies) among all paths.

Relationship:

- The degree of concurrency is influenced by the critical path.
- Shorter critical paths lead to higher concurrency and better parallel efficiency.

23. Whatarethe characteristics of task and interaction?
24. Differentiate between static v/s dynamic interaction
25. Whatisthe process?
26. Whatdoyoumeanbytask?

23. Characteristics of Task and Interaction

Tasks:

- **Definition**: Independent or dependent units of work that a parallel program divides its computation into.
- **Characteristics**:
    1. **Granularity**: Size of computation in a task (fine or coarse).
    2. **Independence**: Tasks may be independent or require synchronization.
    3. **Execution Time**: Varies depending on workload and dependencies.
    4. **Resource Requirements**: Tasks may require specific computational or memory resources.
    5. **Communication**: Some tasks share data or coordinate through communication mechanisms.

Interaction:

- **Definition**: Refers to communication and synchronization between tasks in a parallel computing system.
- **Characteristics**:
    1. **Frequency**: Determines overhead based on how often tasks interact.
    2. **Data Dependency**: Tasks may need data from others to proceed.
    3. **Latency**: Time delay involved in communication.
    4. **Synchronization**: Ensures tasks execute in the correct order or share resources correctly.
    5. **Overhead**: Excessive interaction may reduce parallel efficiency.

---

24. Differentiate Between Static and Dynamic Interaction

Aspect	Static Interaction	Dynamic Interaction
Definition	Fixed pattern of interaction determined at compile time.	Interaction established at runtime, based on system state or workload.
Task Assignment	Predefined and unchanging.	Adjusts dynamically based on execution conditions.
Communication Pattern	Predictable and regular.	Unpredictable and irregular.
Overhead	Lower communication and management overhead.	Higher overhead due to runtime adjustments.
Flexibility	Rigid and less adaptable to workload changes.	Highly adaptable to changing workloads.
Example	Matrix multiplication with fixed block assignments.	Dynamic load balancing in a server environment.

---

25. What is the Process?

A **process** is an independent program in execution within a computer system. It is the basic unit of work that a processor executes, with its own memory space, resources, and state.

Characteristics of a Process:

1. **Independent Execution**: A process operates independently with its own resources.
2. **Address Space**: Each process has a unique memory space.
3. **States**:
    - **Running**: Actively executing on a processor.
    - **Waiting**: Waiting for resources or input/output operations.
    - **Ready**: Prepared to execute when a processor is available.
4. **Communication**: Processes may communicate using Inter-Process Communication (IPC) mechanisms like shared memory or message passing.

Example:

Running a browser and a text editor simultaneously—each is a separate process.

---

26. What Do You Mean by Task?

A **task** is a smaller unit of computation within a parallel program, often representing a piece of the problem that can be executed independently or concurrently with other tasks.

Characteristics of a Task:

1. **Granularity**: Defines the size of the task (fine-grained or coarse-grained).
2. **Dependencies**: May depend on data or results from other tasks.
3. **Parallelism**: Tasks can often be executed concurrently.
4. **Communication**: Some tasks require data exchange with others.
5. **Runtime Behavior**: Tasks can be created dynamically or statically.

Example:

In sorting a dataset, tasks might include sorting different segments of the dataset in parallel.

Define Message Passing interface.

Message Passing Interface (MPI)

The **Message Passing Interface (MPI)** is a standardized and portable communication protocol designed for programming parallel computers. MPI provides a set of functions and libraries that allow processes to communicate with each other by sending and receiving messages.

Key Features of MPI:

1. **Standardized Interface**:
   - Provides a standard for message-passing systems, ensuring portability across different hardware and software platforms.
2. **Process Communication**:
   - Facilitates communication between processes in distributed memory systems through point-to-point and collective communication.
3. **Scalability**:
   - Efficient for both small-scale (e.g., multicore processors) and large-scale (e.g., supercomputers) systems.
4. **Language Support**:
   - Available for multiple programming languages, including C, C++, and Fortran.
5. **Portability**:
   - Platform-independent and works across various operating systems and hardware architectures.
6. **Rich Set of Functions**:
   - Includes functions for data transfer, synchronization, and process management.

Key Concepts in MPI:

1. **Point-to-Point Communication**:
   - Exchange of messages between two specific processes.
   - **Example Functions**: `MPI_Send`, `MPI_Recv`.
2. **Collective Communication**:
   - Communication involving a group of processes.
   - **Examples**:
     - `MPI_Bcast`: Broadcast data from one process to all others.
     - `MPI_Reduce`: Aggregate data from all processes.

3. **Communicators**:
    - Define groups of processes that can communicate with each other.
    - Default communicator: `MPI_COMM_WORLD` (includes all processes).
4. **Rank**:
    - Each process in MPI is assigned a unique identifier (rank) within a communicator, used to specify communication partners.

Advantages of MPI:

1. **Flexibility**: Suitable for a wide range of parallel computing architectures.
2. **Efficiency**: Optimized for high performance on distributed memory systems.
3. **Scalability**: Handles thousands of processes efficiently.
4. **Portability**: Programs written in MPI can run on different systems without modification.

Example Use Case:

- **Parallel Matrix Multiplication**: Divide a large matrix into smaller blocks, distribute them among processes, and use MPI for communication and aggregation of results.

MPI is widely used in scientific computing, engineering simulations, and any application requiring large-scale parallel computation.

Whatarethe different types of MPI datatypes.

Types of MPI Datatypes

In MPI (Message Passing Interface), data sent between processes must conform to specific datatypes so that the communication routines can correctly interpret the data. MPI defines a range of **datatypes** that can represent basic data types, derived/custom types, and special cases.

---

1. Predefined Datatypes

MPI provides predefined datatypes that correspond to standard data types in common programming languages like C, C++, and Fortran.

Examples in C:

MPI Datatype	Corresponding C Type
`MPI_CHAR`	`char`
`MPI_INT`	`int`

MPI Datatype	Corresponding C Type
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_BYTE`	Raw byte data
`MPI_LONG`	`long`
`MPI_UNSIGNED`	`unsigned int`

Examples in Fortran:

MPI Datatype	Corresponding Fortran Type
`MPI_INTEGER`	`INTEGER`
`MPI_REAL`	`REAL`
`MPI_DOUBLE_PRECISION`	`DOUBLE PRECISION`

---

2. Derived Datatypes

MPI allows users to define custom datatypes to represent complex or non-contiguous data structures. Derived datatypes are built using MPI's datatype construction functions.

Common Derived Datatypes:

1. **Contiguous**:
   - Combines multiple instances of a basic datatype into a single, contiguous block.
   - **Function**: `MPI_Type_contiguous`.
2. **Vector**:
   - Represents a regular pattern of data with a fixed stride.
   - **Function**: `MPI_Type_vector`.
3. **Indexed**:
   - Allows arbitrary patterns of data with varying block sizes.
   - **Function**: `MPI_Type_indexed`.
4. **Struct**:
   - Combines different datatypes into a single compound type.
   - Useful for representing complex structures (e.g., structs in C).
   - **Function**: `MPI_Type_create_struct`.
5. **Subarray**:
   - Represents a subarray within a multidimensional array.
   - **Function**: `MPI_Type_create_subarray`.
6. **Heterogeneous Data**:
   - Combines data from multiple different memory locations.
   - Achieved using `MPI_Type_create_hindexed` or `MPI_Type_create_struct`.

---

3. Special Datatypes

1. **Packed Datatype**:
   - Represents a packed sequence of multiple basic or derived types.
   - **Function**: `MPI_Pack` and `MPI_Unpack`.
2. **Null Datatype**:
   - Represents an empty or null datatype.
   - **Constant**: `MPI_DATATYPE_NULL`.

Key Advantages of Using MPI Datatypes:

1. **Flexibility**: Handles non-contiguous and complex data structures efficiently.
2. **Performance**: Reduces communication overhead by enabling efficient packing/unpacking and transmission.
3. **Portability**: Abstracts hardware details, ensuring portability across platforms.

Usage Example (Derived Datatype - Struct):

```c
#include <mpi.h>
#include <stdio.h>

typedef struct {
    int id;
    float value;
} CustomData;

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    // Define the custom datatype
    int block_lengths[2] = {1, 1};
    MPI_Aint offsets[2];
    offsets[0] = offsetof(CustomData, id);
    offsets[1] = offsetof(CustomData, value);
    MPI_Datatype types[2] = {MPI_INT, MPI_FLOAT};
    MPI_Datatype custom_type;

    MPI_Type_create_struct(2, block_lengths, offsets, types, &custom_type);
    MPI_Type_commit(&custom_type);

    // Use custom_type in communication
    CustomData data = {42, 3.14};
    MPI_Send(&data, 1, custom_type, 1, 0, MPI_COMM_WORLD);

    MPI_Type_free(&custom_type);
    MPI_Finalize();
    return 0;
}
```

This example demonstrates the creation and use of a custom `struct` datatype for communication in MPI.

Define Synchronization and its types

Synchronization in Parallel Computing

Synchronization refers to the coordination and management of processes or threads to ensure the correct sequence of execution, consistent access to shared resources, and proper communication in parallel or distributed systems.

Synchronization ensures:

1. **Correctness**: Prevents race conditions by ensuring orderly access to shared data.
2. **Coordination**: Allows processes to communicate and share results effectively.
3. **Efficiency**: Minimizes contention and maximizes parallel performance.

Types of Synchronization

1. Process Synchronization

- Ensures proper sequence of execution between processes to prevent conflicts.
- **Examples**:
  - Synchronizing a producer and a consumer in a producer-consumer problem.
  - Coordinating processes in a distributed database.

---

2. Data Synchronization

- Ensures consistent access and updates to shared data by multiple processes or threads.
- **Examples**:
  - Ensuring a counter shared between threads reflects the correct value.
  - Synchronizing read/write access to a shared memory buffer.

Synchronization Mechanisms

A. Barrier Synchronization

- All processes or threads must reach a specific point before any can proceed.
- **Use Case**:
  - In matrix multiplication, threads synchronize after computing partial results to proceed to the next stage.
- **Example in MPI**: `` `MPI_Barrier` ``.

B. Mutex (Mutual Exclusion)

- Ensures only one process or thread can access a critical section at a time.
- Prevents race conditions in shared resources.
- **Example**: Locking a shared variable to avoid simultaneous updates.

C. Semaphore

- Allows controlled access to a resource by a fixed number of threads or processes.
- Can be:
  - **Binary Semaphore**: Acts as a lock (similar to a mutex).
  - **Counting Semaphore**: Allows a specific number of threads to access a resource.
- **Example**: Managing access to a limited number of database connections.

D. Spinlock

- A lock mechanism where a thread waits in a loop ("spins") until the lock becomes available.
- **Use Case**:
  - Low-latency scenarios where waiting time is minimal.

E. Condition Variables

- Used to block a thread until a specific condition is met.
- Often used with a mutex to signal state changes.
- **Example**:
  - A consumer thread waits for a producer thread to signal that data is ready.

F. Event-based Synchronization

- Uses events to signal threads when specific conditions occur.
- **Example**:
  - Thread A sets an event after producing data, allowing Thread B to consume it.

G. Message Passing

- Synchronization through the exchange of messages between processes in distributed systems.
- **Example in MPI**: Using `MPI_Send` and `MPI_Recv` to synchronize processes.

H. Read-Write Locks

- Allows multiple threads to read a resource simultaneously but provides exclusive access for writing.
- **Use Case**:
  - Scenarios with more read operations than write operations.

Summary Table

Type	Description	Example
Barrier	Ensures all processes reach a point before proceeding.	`MPI_Barrier` in MPI.

Type	Description	Example
Mutex	Mutual exclusion for accessing critical sections.	Locking a shared counter.
Semaphore	Controlled access to resources for multiple threads.	Managing database connections.
Spinlock	Waits in a loop until the lock is available.	Low-latency lock mechanisms.
Condition Variables	Blocks thread until a condition is met.	Waiting for a signal from another thread.
Event-based	Signals threads when conditions occur.	Event-driven thread signaling.
Message Passing	Synchronizes processes via message exchange.	`MPI_Send`, `MPI_Recv`.
Read-Write Locks	Allows multiple readers or one writer.	Reader-writer problem.

Proper synchronization is crucial for building reliable, efficient, and scalable parallel and distributed systems.

Whatarethe different types of job scheduling techniques?

Job Scheduling Techniques

In parallel and distributed systems, **job scheduling** involves allocating computational tasks (jobs) to available processors to optimize performance and resource utilization. The primary goals include minimizing execution time, maximizing resource utilization, ensuring fairness, and meeting deadlines.

Types of Job Scheduling Techniques

1. Static Scheduling

- **Definition**: Job scheduling is determined before execution starts and remains unchanged.
- **Characteristics**:
  - Jobs and resources are allocated at compile time.
  - Assumes prior knowledge of tasks and system resources.
  - Low runtime overhead.
- **Advantages**:
  - Simplicity and predictability.
  - No runtime scheduling overhead.
- **Disadvantages**:
  - Inflexible to workload changes.
  - Poor performance in dynamic environments.
- **Examples**:
  - Assigning fixed rows of a matrix to specific processors for matrix multiplication.

---

2. Dynamic Scheduling

- **Definition**: Job scheduling decisions are made during runtime based on the current system state and workload.
- **Characteristics**:
  - Adjusts to dynamic changes in task loads or resource availability.
  - Requires monitoring of system state.
  - Higher runtime overhead.
- **Advantages**:
  - Flexible and adaptive to workload changes.
  - Handles unpredictable task behavior efficiently.
- **Disadvantages**:
  - Increased scheduling overhead.
  - More complex to implement.
- **Examples**:
  - A web server dynamically assigning requests to available threads.

---

3. Batch Scheduling

- **Definition**: Jobs are grouped into batches and scheduled collectively.
- **Characteristics**:
  - Suitable for long-running, non-interactive jobs.
  - Jobs in the batch queue are processed based on priority or order.
- **Advantages**:
  - Simple implementation.
  - High throughput for batch jobs.
- **Disadvantages**:
  - High latency for individual jobs.
- **Examples**:
  - Scheduling large simulations in a supercomputer environment.

---

4. Real-Time Scheduling

- **Definition**: Ensures jobs are completed within specified time constraints.
- **Characteristics**:
  - Divided into **hard real-time** (strict deadlines) and **soft real-time** (flexible deadlines).
  - Critical for systems like embedded or time-sensitive applications.
- **Advantages**:
  - Guarantees timely completion of critical tasks.
- **Disadvantages**:
  - Requires accurate estimations of execution times.
  - Complex to implement in non-dedicated systems.
- **Examples**:
  - Scheduling tasks in automotive control systems.

---

5. Fair Scheduling

- **Definition**: Ensures that all jobs get a fair share of system resources.

- **Characteristics**:
  - Prevents resource monopolization by high-priority tasks.
  - Suitable for multi-user or multi-tenant environments.
- **Advantages**:
  - Ensures fairness across jobs.
  - Balances workload distribution.
- **Disadvantages**:
  - May not optimize performance for high-priority jobs.
- **Examples**:
  - Hadoop's Fair Scheduler in distributed computing.

---

6. Priority Scheduling

- **Definition**: Jobs are assigned priorities, and higher-priority jobs are executed first.
- **Characteristics**:
  - Can be preemptive (interrupt lower-priority tasks) or non-preemptive.
  - Priorities can be static or dynamic.
- **Advantages**:
  - Efficient for time-critical tasks.
  - High-priority jobs are executed faster.
- **Disadvantages**:
  - Lower-priority jobs may face starvation.
- **Examples**:
  - Real-time operating systems prioritizing emergency tasks.

---

7. Load Balancing Scheduling

- **Definition**: Focuses on evenly distributing tasks across processors to avoid overloading any single processor.
- **Characteristics**:
  - Balances computational load dynamically or statically.
  - Minimizes idle time for processors.
- **Advantages**:
  - Maximizes system throughput.
  - Reduces execution time.
- **Disadvantages**:
  - Overhead for continuous load monitoring.
- **Examples**:
  - Dynamic workload distribution in cloud computing.

---

8. Hybrid Scheduling

- **Definition**: Combines multiple scheduling techniques to leverage their benefits.
- **Characteristics**:
  - Customizable for specific applications or systems.
  - Provides flexibility and adaptability.

- **Advantages**:
  - Balances efficiency and fairness.
- **Disadvantages**:
  - Increased implementation complexity.
- **Examples**:
  - Combining batch and priority scheduling for heterogeneous workloads.

Summary Table

Scheduling Type	Key Feature	Best Used For
Static Scheduling	Predefined scheduling at compile time	Predictable, uniform workloads
Dynamic Scheduling	Runtime decision-making	Dynamic or unpredictable workloads
Batch Scheduling	Processes jobs in batches	Long-running, non-interactive jobs
Real-Time Scheduling	Meets strict deadlines	Time-critical systems
Fair Scheduling	Ensures resource fairness	Multi-user or multi-tenant systems
Priority Scheduling	Executes jobs by priority	Time-sensitive tasks
Load Balancing Scheduling	Distributes workload evenly	Distributed or parallel systems
Hybrid Scheduling	Combines multiple approaches	Complex and diverse workloads

---

By choosing the appropriate scheduling technique, systems can achieve better performance, fairness, and efficiency tailored to their specific workload characteristics.

WhatisGPGPU?
42. Whatarethe components of GPGPU?

What is GPGPU?

GPGPU (General-Purpose computing on Graphics Processing Units) refers to the use of GPUs (Graphics Processing Units) for performing computations that are traditionally handled by CPUs, particularly tasks unrelated to graphics rendering. GPGPU leverages the massive parallel processing capabilities of GPUs to accelerate a wide range of computational tasks, such as scientific simulations, machine learning, image processing, and data analysis.

Key Features of GPGPU

1. **Massive Parallelism**:
   - GPUs consist of hundreds or thousands of cores capable of processing multiple tasks simultaneously.
2. **High Throughput**:

- Ideal for tasks that involve large datasets and repetitive computations.

3. **Programming Frameworks**:
   - GPGPU uses specialized programming models such as **CUDA** (NVIDIA) and **OpenCL** (vendor-neutral).
4. **Applications**:
   - Machine learning, scientific computing, financial modeling, cryptography, and real-time signal processing.

Components of GPGPU

1. **Streaming Multiprocessors (SMs)**:
   - The primary computational units within a GPU.
   - Contain multiple cores, responsible for executing instructions in parallel.
   - Include schedulers and warp (group of threads) controllers.
2. **GPU Cores**:
   - Perform arithmetic and logic operations.
   - Highly optimized for floating-point operations.
3. **Registers**:
   - Provide fast local storage for each thread.
   - Enable low-latency access to frequently used data.
4. **Shared Memory**:
   - A small, fast memory block shared among threads within the same block.
   - Enables inter-thread communication and data sharing.
5. **Global Memory**:
   - Accessible by all threads across different blocks.
   - High latency compared to shared memory, but suitable for storing large datasets.
6. **Texture and Constant Memory**:
   - Specialized memory regions optimized for specific types of data:
     - **Texture Memory**: Efficient for spatially localized data (e.g., images).
     - **Constant Memory**: Optimized for read-only data shared across threads.
7. **Memory Controllers**:
   - Manage data transfer between the GPU's cores and its memory hierarchy.
   - Includes techniques like coalescing to improve memory access efficiency.
8. **PCIe Interface**:
   - Connects the GPU to the CPU and system memory.
   - Transfers data between the host and device.
9. **Execution Units**:
   - Perform vectorized operations on multiple data points in parallel.
10. **Control Logic**:

- Manages task scheduling, thread execution, and instruction flow.

11. **Caches**:

- Hierarchical memory caches (L1, L2) for faster access to frequently used data.

GPGPU vs Traditional GPU Rendering

Feature	GPGPU	Traditional GPU Rendering
Purpose	General-purpose computation	Graphics rendering (e.g., games, visuals)
Focus	Computational workloads (e.g., ML, HPC)	Visual tasks (e.g., rasterization, shaders)
Programming Models	CUDA, OpenCL	DirectX, Vulkan, OpenGL

GPGPU has revolutionized fields requiring high computational power by unlocking the full potential of GPUs for non-graphical tasks.

Write about DGX architecture.

DGX Architecture

NVIDIA's **DGX systems** are purpose-built hardware platforms designed to accelerate AI workloads, including machine learning, deep learning, and data analytics. DGX stands for **"Deep Learning Supercomputing Platform"** and incorporates cutting-edge hardware and software to provide unmatched computational performance and scalability for AI research and production.

Key Features of DGX Architecture

1. **High-Performance GPUs**:
    - DGX systems leverage NVIDIA GPUs, such as **A100 Tensor Core GPUs**, with advanced parallel processing capabilities.
    - Each GPU includes:
        - **Tensor Cores** for mixed-precision computations.
        - **CUDA Cores** for general-purpose GPU computing.
2. **NVLink and NVSwitch Interconnect**:
    - High-bandwidth interconnects allow fast communication between GPUs.
    - **NVLink**: A direct GPU-to-GPU communication interface providing speeds up to 600 GB/s.
    - **NVSwitch**: Connects multiple GPUs in a mesh topology, enabling full-bandwidth, all-to-all connectivity in multi-GPU systems.
3. **Memory Architecture**:
    - Supports **HBM2e (High Bandwidth Memory)**, offering terabytes per second of memory bandwidth per GPU.
    - Unified memory architecture enables seamless data sharing between GPUs and CPUs.
4. **DGX Software Stack**:
    - Preloaded with optimized software for AI and HPC workloads, including:

    - NVIDIA **CUDA Toolkit**.
    - NVIDIA **Deep Learning SDK**.
    - **NVIDIA NGC** (NVIDIA GPU Cloud) containers with pre-configured AI frameworks like TensorFlow, PyTorch, and MXNet.
  - Enables rapid deployment of AI applications without manual configuration.
5. **AI Accelerators**:
   - Integrated support for AI-specific optimizations, such as sparsity-aware computations and mixed-precision processing.
6. **Scalability**:
   - DGX systems can function as standalone units or scale across multiple nodes in a data center using NVIDIA **InfiniBand** and **Ethernet** technologies for inter-node communication.
7. **Optimized Cooling and Power**:
   - Advanced thermal design ensures efficient cooling for dense computational workloads.
   - Energy-efficient architecture reduces power consumption while delivering high performance.

DGX Systems Overview

1. **DGX Station**:
   - A workstation-class AI system for smaller teams.
   - Typically includes 4 GPUs interconnected via NVLink.
2. **DGX A100**:
   - A data center-class system featuring 8 A100 GPUs.
   - Provides up to **5 petaflops** of AI performance.
   - Optimized for training and inference tasks.
3. **DGX SuperPOD**:
   - A cluster of DGX systems interconnected to provide supercomputing capabilities.
   - Designed for large-scale AI research and industrial deployment.
   - Can achieve **exaflop-scale performance**.

Advantages of DGX Architecture

1. **Extreme AI Performance**:
   - Accelerates training and inference for large-scale deep learning models.
   - Supports workloads like NLP, computer vision, and generative AI.
2. **Ease of Use**:
   - Turnkey solution with pre-installed software stacks for quick deployment.
3. **Scalable Design**:
   - Supports multi-node configurations for enterprise-level AI tasks.
4. **Efficient Resource Utilization**:
   - Optimized interconnects and memory reduce bottlenecks in data-intensive applications.

Applications of DGX Architecture

1. **AI Research**:
   - Training state-of-the-art models like GPT, BERT, and Stable Diffusion.
2. **Data Analytics**:
   - Accelerating big data frameworks like Apache Spark.
3. **HPC Workloads**:
   - Simulating complex phenomena in weather modeling, physics, and genomics.
4. **Enterprise AI**:
   - Enabling predictive analytics, recommendation systems, and fraud detection.

Conclusion

The NVIDIA DGX architecture is a cornerstone for accelerating AI and HPC workloads in both research and industry. Its cutting-edge hardware and software integration deliver unparalleled performance, scalability, and efficiency, making it a leader in the era of AI-driven innovation.

what is GProf

What is GProf?

GProf (GNU Profiler) is a performance analysis tool used to profile programs written in C, C++, and Fortran. It provides insights into a program's runtime behavior, helping developers identify bottlenecks, optimize code, and improve application performance. GProf was originally developed as part of the GNU project.

Features of GProf

1. **Flat Profile**:
   - Displays the amount of time spent in each function of the program.
   - Helps identify functions consuming the most execution time.
2. **Call Graph**:
   - Shows how functions interact with each other during execution.
   - Details time spent in each function and its descendants (functions it calls).
3. **Execution Time Analysis**:
   - Measures the time taken by individual functions and their cumulative impact on the total execution time.
4. **Supports Multiple Languages**:

- Works with programs written in C, C++, and Fortran.

5. **Lightweight Profiling**:
   - Introduces minimal overhead, ensuring accurate profiling without significantly affecting performance.

How GProf Works

1. **Instrumenting the Code**:
   - Compile the program with the `-pg` flag to enable profiling.

```bash
gcc -pg -o my_program my_program.c
```

2. **Running the Program**:
   - Execute the compiled program as usual. This generates a file called `gmon.out`, which contains profiling data.

```bash
./my_program
```

3. **Analyzing the Profiling Data**:
   - Use GProf to analyze the `gmon.out` file and produce a readable report.

```bash
gprof my_program gmon.out > analysis.txt
```

4. **Reviewing the Report**:
   - Examine the generated `analysis.txt` file for function-level details and call graph information.

Output of GProf

1. **Flat Profile**:
   - Lists all functions with:
     - Total time spent in each function.
     - Number of times each function was called.
     - Average time per call.
2. **Call Graph**:

- Shows relationships between functions:
  - Caller functions (functions that call others).
  - Callee functions (functions being called).
- Time spent in each function and its descendants.

Benefits of Using GProf

1. **Identifies Bottlenecks**:
   - Highlights functions consuming the most time for targeted optimization.
2. **Improves Code Efficiency**:
   - Helps developers focus on optimizing critical parts of the program.
3. **Simple Integration**:
   - Works seamlessly with common compilers like GCC and G++.
4. **Detailed Analysis**:
   - Provides insights into both individual functions and their interactions.

Limitations of GProf

1. **Sampling-Based**:
   - GProf uses sampling, which may miss very short-lived functions.
   - Precision depends on the sampling frequency.
2. **Lack of Multithreading Support**:
   - Doesn't handle multi-threaded programs well, leading to inaccurate results for such applications.
3. **Code Instrumentation Overhead**:
   - Requires recompiling the code with the `-pg` flag, which may not be feasible for all projects.

When to Use GProf

- Identifying hotspots in a single-threaded application.
- Analyzing execution time distribution among functions.
- Optimizing performance-critical sections of the code.

Alternatives to GProf

While GProf is a powerful tool, modern alternatives may offer better support for multi-threaded or more complex applications:

1. **Perf** (Linux Performance Profiler).
2. **Valgrind (Callgrind)**.
3. **Intel VTune Profiler**.
4. **gperftools**.

5. **Visual Studio Profiler** (for Windows).

GProf remains a robust and accessible tool for profiling single-threaded programs and understanding their runtime behavior.

State Adhalm's law.

Amdahl's Law

Amdahl's Law is a formula used to predict the theoretical speedup of a task achieved by parallelizing portions of it. It was formulated by computer scientist Gene Amdahl in 1967. The law highlights the limitations of parallel computing, emphasizing that the fraction of the task that cannot be parallelized constrains the overall performance improvement.

Mathematical Representation

The speedup S is given by:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

Where:

- S: The speedup of the program when using N processors.
- P: The parallelizable portion of the task (a fraction between 0 and 1).
- $(1 - P)$: The sequential portion of the task.
- N: The number of processors.

Key Insights from Amdahl's Law

1. **Sequential Bottleneck**:
   - The performance improvement is limited by the portion of the task that must be executed sequentially $(1 - P)$.
2. **Diminishing Returns**:
   - As the number of processors (N) increases, the speedup approaches an upper limit determined by the sequential portion of the task.
3. **Maximum Speedup**:
   - When $N \to \infty$, the maximum theoretical speedup is:

$$S_{\max} = \frac{1}{1 - P}$$

Example

If 80% of a program ($P = 0.8$) can be parallelized and 20% ($1 - P = 0.2$) must be executed sequentially:

- With $N = 4$ processors:

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

- Even with infinite processors:

$$S_{\max} = \frac{1}{1 - 0.8} = 5$$

Implications

1. **Focus on Reducing Sequential Portions**:
   - To achieve significant speedup, efforts should be made to parallelize as much of the program as possible.
2. **Practical Limits on Processor Count**:
   - Adding more processors yields diminishing returns if the parallel portion (P) is not close to 1.
3. **Balanced Resource Allocation**:
   - Over-provisioning processors for tasks with significant sequential components results in inefficient resource utilization.

Applications

- Used in performance modeling and scalability analysis of parallel systems.
- Guides software and hardware optimization for high-performance computing.

Amdahl's Law provides a foundational understanding of the trade-offs and constraints in parallel computing.