CBCS SCHEME

USN | | | | | | | | | | |

BCS702

Seventh Semester B.E./B.Tech. Degree Examination, Dec.2025/Jan.2026
Parallel Computing

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain in detail the classification of parallel computers according to Flynn's taxonomy. Compare SIMD and MIMD systems.	10	L1	CO1
	b.	Discuss shared memory and distributed memory architectures. Explain their working principle advantages and disadvantages.	10	L2	CO1
		OR			
Q.2	a.	What is cache coherence? Explain snooping and directory based coherence mechanisms with suitable example.	10	L3	CO1
	b.	Explain non-determinism and race conditions in shared memory programs. How can they be avoided?	10	L3	CO1
		Module – 2			
Q.3	a.	Explain GPU programming in detail.	10	L2	CO2
	b.	Describe input and output handling in MIMD and GPU systems.	10	L2	CO2
		OR			
Q.4	a.	Explain Amdahl's law with example and discuss it significance.	10	L3	CO2
	b.	Write a short note on timing and performance measurement of parallel programs.	10	L3	CO2
		Module – 3			
Q.5	a.	Define and explain the following MPI functions with syntax and purpose : i) MPI_Init( ) ii) MPI_Finalize( ) iii) MPI_Comm_Size( ) iv) MPI_Comm_rank( )	10	L3	CO3
	b.	Explain the concept of point to point communication in MPI with suitable example.	10	L2	CO3
		OR			
Q.6	a.	Explain the working of trapezoidal rule program in MPI.	10	L3	CO3
	b.	Compare the traditional global sum using process 0 as collector with tree structured global sum.	10	L3	CO3

		Module – 4			
Q.7	a.	Explain the structure and working of an OpenMP "Hello world" program.	6	L2	CO4
	b.	Explain the purpose of the reduction clause in OpenMP with example.	6	L2	CO4
	c.	Write a OpenMP program to calculate n-Fibonacci using tasks.	8	L2	CO4
		OR			
Q.8	a.	Define OpenMP. Explain the key features of OpenMP and its advantages over Pthreads.	6	L1	CO4
	b.	Explain the concept of variable scope in OpenMP with suitable example.	6	L2	CO4
	c.	Estimate the value of π.	8	L3	CO4
		Module – 5			
Q.9	a.	Define GPU and GPGPU. Explain the need for GPGPU.	6	L1	CO5
	b.	Explain thread, block and grid in CUDA.	6	L2	CO5
	c.	Explain CUDA vector addition program with suitable example.	8	L3	CO5
		OR			
Q.10	a.	Compare CUDA and OpenCL.	6	L1	CO5
	b.	Explain Kernel with shared memory.	6	L2	CO5
	c.	Explain Heterogeneous computing in detail.	8	L3	CO5

* * * * *

PARALLEL COMPUTING 7TH SEMESTER -DEC 2025-JAN 2026
SUBJECT CODE: BCS702

MODULE 1

Q.1 a. Explain in detail the classification of parallel computers according to Flynn's taxonomy. Compare SIMD and MIMD systems.

Flynn's taxonomy, proposed by Michael J. Flynn, classifies parallel computer architectures based on the number of instruction streams and data streams processed simultaneously. An instruction stream refers to a sequence of instructions, while a data stream refers to the data operated upon.

Flynn classified computers into four categories:

1. SISD (Single Instruction, Single Data)

- One processor executes a single instruction on a single data item.
- No parallelism is involved.
- Example: Traditional uniprocessor systems.

2. SIMD (Single Instruction, Multiple Data)

- A single instruction stream operates on multiple data elements simultaneously.
- All processing elements execute the same instruction in a lockstep manner.
- Provides data-level parallelism.
- Examples: Vector processors, GPUs.
- Suitable for matrix operations and image processing.

3. MISD (Multiple Instruction, Single Data)

- Multiple instruction streams operate on the same data stream.
- Rare and mostly theoretical.
- Used in fault-tolerant and safety-critical systems.

4. MIMD (Multiple Instruction, Multiple Data)

- Multiple processors execute different instructions on different data streams independently.
- Supports both task-level and data-level parallelism.
- Examples: Multicore processors, clusters.
- Most commonly used architecture in modern parallel systems.

Comparison of SIMD and MIMD Systems

Feature	SIMD	MIMD
Instruction Stream	Single	Multiple
Data Stream	Multiple	Multiple
Execution	Synchronous	Asynchronous
Parallelism	Data parallelism	Task & data parallelism
Flexibility	Low	High
Examples	GPUs, Vector processors	Multicore CPUs, Clusters

b. Discuss shared memory and distributed memory architectures. Explain their working principle advantages and disadvantages.

Parallel computer architectures are commonly classified based on how memory is organized and accessed by processors. The two major architectures are Shared Memory and Distributed Memory systems.

Shared Memory Architecture

Working Principle

In a **shared memory architecture**, multiple processors access a **single, common memory space**. All processors can read and write to the same memory locations. Communication between processors occurs implicitly through shared variables stored in memory.

Shared memory systems may be:

- **UMA (Uniform Memory Access)** – equal access time to memory
- **NUMA (Non-Uniform Memory Access)** – access time depends on memory location

Examples

- Multicore processors
- Symmetric Multiprocessor (SMP) systems

Advantages

- Simple programming model
- Easy data sharing among processors
- Lower communication overhead
- Suitable for fine-grained parallelism

Disadvantages

- Limited scalability due to memory contention
- Synchronization overhead (locks, barriers)
- Cache coherence issues
- Performance degrades as number of processors increases

Distributed Memory Architecture

Working Principle

In a **distributed memory architecture**, each processor has its **own local memory**. There is no global shared memory. Processors communicate by **explicit message passing** over a network.

Data exchange is performed using communication primitives such as **send** and **receive**.

Advantages

Highly scalable

No shared memory contention

Better performance for large-scale applications

Suitable for coarse-grained parallelism

Disadvantages

- Complex programming model
- Explicit communication required
- Higher communication latency
- Data consistency must be handled by the programmer

Comparison of Shared Memory and Distributed Memory Architectures

Feature	Shared Memory	Distributed Memory
Memory Organization	Single shared memory	Private local memory
Communication	Through shared variables	Message passing
Programming	Easier	More complex
Scalability	Limited	High
Synchronization	Required	Less frequent
Examples	SMP, Multicore CPUs	Clusters, Supercomputers

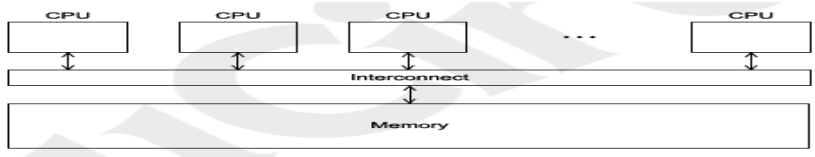


FIGURE 2.3
A shared-memory system.

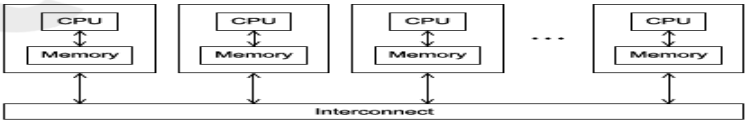
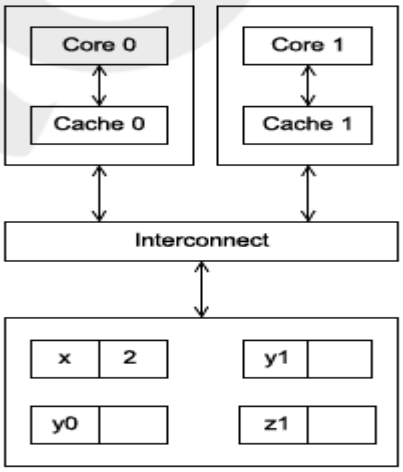


FIGURE 2.4
A distributed-memory system.

Q.2 a. What is cache coherence? Explain snooping and directory based coherence mechanisms with suitable example.

Cache coherence refers to the problem of maintaining a consistent view of shared data stored in multiple caches in a shared-memory multiprocessor system. When multiple processors have local caches and share common memory, updates made by one processor must be visible to all other processors to ensure correctness. If cache coherence is not maintained, processors may operate on stale or inconsistent data leading to wrong exe.



Snooping cache coherence

There are two main approaches to ensuring cache coherence: **snooping cache coherence** and **directory-based cache coherence**. The idea behind snooping com bus-based systems: When the cores share a bus, any signal transmitted on the be "seen" by all the cores connected to the bus. Thus when core 0 updates t of x stored in its cache, if it also broadcasts this information across the bus core 1 is "snooping" the bus, it will see that x has been updated, and it can r copy of x as invalid. This is more or less how snooping cache coherence wor principal difference between our description and the actual snooping protoc the broadcast only informs the other cores that the *cache line* containing x h updated, not that x has been updated.

A couple of points should be made regarding snooping. First, it's not essen the interconnect be a bus, only that it support broadcasts from each process

the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches, there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks, broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated. So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Fig. 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable x stored in core 1's memory, by simply executing a statement such as y = x. (Of course, accessing the memory attached to another core will be slower than accessing "local" memory, but that's another story.) Such a system can, in principle, scale to very large numbers of cores. However, snooping cache coherence is clearly a problem, since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated, indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches will invalidate those lines.

b. **Explain non-determinism and race conditions in shared memory programs. How can they be avoided?**

Non-determinism occurs in a parallel program when the output or behavior varies from one execution to another, even when the input remains the same. In shared memory systems, non-determinism arises due to uncontrolled access to shared variables by multiple threads.

The execution order of threads depends on factors such as:

- Thread scheduling
- Timing of execution
- Processor availability

As a result, different interleavings of operations can produce different outcomes.

Race Conditions

A **race condition** occurs when:

- Two or more threads access a **shared variable concurrently**
- At least one of the accesses is a **write**
- The final result depends on the **order of execution**

Race conditions are a primary cause of non-deterministic behavior.

Example

```
Shared variable: count = 0


Thread 1: count = count + 1

Thread 2: count = count + 1
```

If both threads execute simultaneously without synchronization, the final value of `count` may be **1 instead of 2** due to overlapping read-modify-write operations.

Relationship Between Non-Determinism and Race Conditions

- Race conditions lead to **unpredictable execution order**
- This unpredictability results in **non-deterministic program output**
- Not all non-determinism is harmful, but race-induced non-determinism is usually incorrect

---

Avoiding Non-Determinism and Race Conditions

1. Mutual Exclusion

- Protect critical sections using **locks, mutexes, or semaphores**
- Ensure only one thread accesses shared data at a time

2. Atomic Operations

- Use atomic instructions for read-modify-write operations
- Prevent partial updates of shared variables

3. Synchronization Constructs

- Use **barriers, condition variables**
- Ensure proper ordering of thread execution

4. Minimize Shared Data

- Use **private variables** wherever possible
- Reduce shared state to avoid conflicts

5. Structured Parallel Programming

- Use high-level constructs like **OpenMP critical, atomic, and reduction**

- These enforce correct synchronization automatically

MODULE 2

Q.3 a. Explain GPU programming in detail.

GPU programming refers to developing programs that execute on a Graphics Processing Unit (GPU) to exploit its massive parallel processing capability. Unlike CPUs, which are optimized for sequential and control-intensive tasks, GPUs are designed for data-parallel computations where the same operation is applied to a large number of data elements simultaneously. GPUs follow the SIMD / SIMT (Single Instruction, Multiple Threads) execution model and are widely used in scientific computing, machine learning, image processing, and simulations.

GPUs are usually not "standalone" processors. They don't ordinarily run an operating system and system services, such as direct access to secondary storage. So programming a GPU also involves writing code for the CPU "host" system, which runs on an ordinary CPU. The memory for the CPU host and the GPU memory are usually separate. So the code that runs on the host typically allocates and initializes storage on both the CPU and the GPU. It will start the program on the GPU, and it is responsible for the output of the results of the GPU program. Thus GPU programming is really heterogeneous programming, since it involves programming two different types of processors.The GPU itself will have one or more processors. Each of these processors is capable of running hundreds or thousands of threads. In the systems we'll be using, the processors share a large block of memory, but each individual processor has a small block of much faster memory that can only be accessed by threads running on that processor. These blocks of faster memory can be thought of as a programmer-managed cache.
The threads running on a processor are typically divided into groups: the threads within a group use the SIMD model, and two threads in different groups can run independently. The threads in a SIMD group may not run in lockstep. That is, they may not all execute the same instruction at the same time. However, no thread in the group will execute the next

instruction until all the threads in the group have completed executing the current instruction. If the threads in a group are executing a branch, it may be necessary to idle some of the threads. For example, suppose there are 32 threads in a SIMD group, and each thread has a private variable rank_in_gp that ranges from 0 to 31. Suppose also that the threads are executing the following code.

```
// Thread private variables
int rank_in_gp, my_x;
...
if (rank_in_gp < 16)
   my_x += 1;
else
   my_x -= 1;
```

Then the threads with rank < 16 will execute the first assignment, while the threads with rank \(\ge \) 16 are idle. After the threads with rank < 16 are done, the roles will be reversed: the threads with rank < 16 will be idle, while the threads with rank > 16 will execute the second assignment. Of course, idling half the threads for two instructions isn't a very efficient use of the available resources. So it's up to the programmer to minimize branching, where the threads within a SIMD group take different branches.Another issue in GPU programming that's different from CPU programming is how the threads are scheduled to execute. GPUs use a hardware scheduler (unlike CPUs, which use software to schedule threads and processes), and this hardware scheduler uses very little overhead. However, the scheduler will choose to execute an instruction when all the threads in the SIMD group are ready. In the preceding example, before executing the test, we would want the variable rank_in_gp stored in a register by each thread. So, to maximize use of the hardware, we usually create a large number of SIMD groups. When this is the case, groups that aren't ready to execute (e.g., they're waiting for data from memory, or waiting for the completion of a

previous instruction) can be idled, and the scheduler can choose a SIMD group that is ready.

Applications of GPU Programming

- Machine learning and deep learning
- Image and video processing
- Scientific simulations
- Weather forecasting
- Financial modeling

---

b. **Describe input and output handling in MIMD and GPU systems.**

In MIMD systems, input and output handling depends on whether the architecture is shared memory or distributed memory. In shared memory MIMD systems, I/O devices are usually shared among processors, and typically a single processor or a master thread performs input and output operations. The data obtained from input devices is placed in shared memory, from where other processors can access it. Synchronization mechanisms such as locks and barriers are used to prevent multiple processors from accessing the same I/O device simultaneously. In distributed memory MIMD systems, each processor has its own local memory and I/O may be handled either centrally by one process or in parallel by multiple processes. Input data read by one processor can be communicated to others using message-passing techniques such as MPI, and parallel file systems are often employed to support large-scale I/O operations.

In GPU systems, input and output handling follows a heterogeneous computing model in which the CPU (host) is responsible for all I/O operations, while the GPU (device) is used only for computation. The CPU reads input data from files or input devices and transfers it to the GPU's global memory. GPU kernels then execute in parallel on this data and store the results in device memory. After kernel execution, the computed results are copied back from the GPU memory to the host memory, and the CPU performs the output operations such as writing results to files or displaying them. GPUs cannot directly interact with I/O devices, and therefore minimizing host–device data transfers is essential for achieving good performance.

Q.4 a Explain Amdahl's law with example and discuss its significance.

Back in the 1960s, Gene Amdahl made an observation [3] that's become known as Amdahl's law. It says, roughly, that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited-regardless of the number of cores available. Suppose, for example, that we're able to parallelize 90% of a serial program. Furthermore, suppose that the parallelization is "perfect," that is, regardless of the number of cores p we use, the speedup of this part of the program will be p. If the seria run-time is T_{serial} = 20 seconds, then the run-time of the parallelized part will be $0.9 \times T_{serial}/p = 18/p$ and the run-time of the "unparallelized" part will be $0.1 \times T_{serial} = 2$. The overall parallel run-time will be

$$T_{parallel} = 0.9 \times T_{serial}/p + 0.1 \times T_{serial} = 18/p + 2$$

and the speedup will be

$$S = \frac{T_{serial}}{0.9 \times T_{serial}/p + 0.1 \times T_{serial}} = \frac{20}{18/p + 2}$$

Now as p gets larger and larger, $0.9 \times T_{serial}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1 \times T_{serial} = 2$. That is, the denominator in S can't be smaller than $0.1 \times T_{serial} = 2$. The fraction S must therefore satisfy the inequality

$$S < \frac{T_{serial}}{0.1 \times T_{serial}} = \frac{20}{2} = 10.$$

That is, $S \leq 10$. This is saying that even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

More generally, if a fraction r of our serial program remains unparallelized, then Amdahl's law says we can't get a speedup better than 1/r. In our example, $r = 1 - 0.9 = 1/10$, so we couldn't get a speedup better than 10. Therefore if a.

Amdahl's Law provides a theoretical limit on the maximum speedup achievable by parallelizing a program, given that some portion of the program must remain sequential.It states that if a fraction s of a program is serial (cannot be parallelized) and the remaining fraction (1 − s) can be perfectly parallelized using p processors, then the overall speedup S(p) is:

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

Example

Assume a program where **20% (s = 0.2)** of the code is sequential and **80%** can be parallelized. If the program is executed on **4 processors**, the speedup is:

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

Even with **infinite processors**, the maximum possible speedup would be:

$$S(\infty) = \frac{1}{s} = \frac{1}{0.2} = 5$$

This shows that the sequential portion limits performance gains, regardless of how many processors are added.

Significance of Amdahl's Law

Amdahl's Law highlights that the sequential part of a program is the primary bottleneck in parallel computing. It emphasizes the importance of minimizing serial sections to achieve higher speedup. The law also shows that adding more processors yields diminishing returns once the parallel portion is saturated. It is useful for evaluating whether parallelization is worthwhile and for setting realistic expectations about performance improvement. However, it assumes a fixed problem size and does not account for increased workload with more processors, which limits its applicability to real-world scalable systems.

---

b. **Write a short note on timing and performance measurement of parallel programs.**

The timing and performance measurement of parallel programs is essential to evaluate the efficiency of parallelization and identify bottlenecks. The total execution time of a parallel program is usually measured using wall-clock time, which records the elapsed real time from start to finish, or CPU time, which measures the processor time consumed. Important metrics include speedup, which is the ratio of execution time of the sequential program to the parallel program; efficiency, which is speedup divided by the number of processors; and throughput, indicating the amount of work done per unit time. Tools like timers in MPI (MPI_Wtime), OpenMP timing

routines, or profiling tools help record execution times. Measuring performance allows developers to identify load imbalance, communication overhead, and synchronization delays, helping to optimize parallel programs for better scalability and resource utilization.

1. **Speedup (S)**: The ratio of sequential execution time (T_1) to parallel execution time on p processors (T_p):

$$S = \frac{T_1}{T_p}$$

It indicates how much faster the parallel program runs compared to the sequential version.

2. **Efficiency (E)**: Measures processor utilization and is given by:

$$E = \frac{S}{p} = \frac{T_1}{p \cdot T_p}$$

It shows how effectively the processors are used.

3. **Throughput**: Amount of work completed per unit time, important for workload-intensive applications.
4. **Scalability**: How performance improves ↓ the number of processors or problem size increases.

Tools for Timing and Measurement:

- MPI programs: MPI_Wtime() for measuring start and end times
- OpenMP programs: omp_get_wtime() function
- Profilers: Tools like gprof, nvprof, or Intel VTune to measure computation, communication, and synchronization overhead

Significance:

Timing and performance measurements help identify load imbalance, communication delays, and synchronization overhead. Optimizing these factors improves parallel efficiency, resource utilization, and scalability. Accurate measurement also aids in comparing different parallel algorithms and architectures.

MODULE 3

Q.5 a. Define and explain the following MPI functions with syntax and purpose:

i) MPI_Init()

i) `MPI_Init()`

- **Purpose:** Initializes the MPI environment.
- **Usage:** Must be called **before any other MPI function**.
- **Syntax:**

int MPI_Init(int *argc, char ***argv);

---

ii) MPI_Finalize()

ii) `MPI_Finalize()`

- **Purpose:** Terminates the MPI environment.
- **Usage:** Called **after all MPI operations are completed.**
- **Syntax:**

int MPI_Finalize(void);

---

iii) MPI_Comm_Size()

iii) `MPI_Comm_Size()`

- **Purpose:** Determines the **total number of processes** in a communicator.
- **Usage:** Often used to get the total number of processors in the program.
- **Syntax:**

int MPI_Comm_size(MPI_Comm comm, int *size);

- **Parameters:**
  - `comm` – Communicator (usually `MPI_COMM_WORLD`)
  - `size` – Pointer to integer that returns number of processes

---

iv) MPI_Comm_rank()

iv) `MPI_Comm_rank()`

- **Purpose**: Determines the **rank (ID) of the calling process** in a communicator.
- **Usage**: Helps each process know its identity to perform different tasks.
- **Syntax**:

int MPI_Comm_rank(MPI_Comm comm, int *rank);

- **Parameters**:
  - `comm` – Communicator (usually `MPI_COMM_WORLD`)
  - `rank` – Pointer to integer that returns the process rank

---

b. Explain the concept of point to point communication in MPI with suitable example.

Point-to-point communication in MPI refers to the direct exchange of data between two processes. One process sends data, and another process receives it. This is the simplest form of communication in MPI and is the foundation of most MPI programs.

Key Concepts

1. Send and Receive Operations
   - MPI_Send(): Used by the sending process to send data to a specific destination process.
   - MPI_Recv(): Used by the receiving process to receive data from a specific source process.
2. Communicator
   - Communication occurs within a communicator, usually MPI_COMM_WORLD, which includes all processes.
3. Tags
   - Each message can have a tag, which helps the receiver identify the type of message.

---

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);

Parameters:

- `buf` – pointer to the data buffer
- `count` – number of elements
- `datatype` – type of data (e.g., `MPI_INT`, `MPI_FLOAT`)
- `dest` / `source` – rank of destination or source process
- `tag` – message tag
- `comm` – communicator (usually `MPI_COMM_WORLD`)
- `status` – stores information about received message

Example: Sending an Integer from Process 0 to Process 1

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

int main(int argc, char *argv[]) {
   int rank, size;
   int data;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 0) {
      data = 100;
      MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // Send data to process 1
      printf("Process 0 sent %d to Process 1\n", data);
   }
   else if (rank == 1) {
```

```
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE); // Receive data from process 0
        printf("Process 1 received %d from Process 0\n", data);
    }

    MPI_Finalize();
    return 0;
}
```

Q.6 **a.** **Explain the working of trapezoidal rule program in MPI.**

Working of Trapezoidal Rule Program in MPI

The **Trapezoidal Rule** is a numerical method for approximating the definite integral of a function:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right]$$

Where:

- `a` = lower limit, `b` = upper limit
- `n` = number of subintervals
- `h = (b - a) / n` = width of each trapezoid

In **MPI**, the computation can be **parallelized** by dividing the interval among multiple processes.

Steps in MPI Trapezoidal Rule

1. Initialize MPI Environment
    - Call MPI_Init() to start MPI
    - Get the total number of processes (MPI_Comm_size())
    - Get the rank of each process (MPI_Comm_rank())

---

2. **Divide the Interval**
   - The interval `[a, b]` is divided into `n` trapezoids.
   - Each process computes the **local integral** over its subinterval `[loca local_b]`.
   - Subinterval width for each process:

$$local\_n = n/p, \quad h = (b-a)/n$$

   - `local_a = a + rank * local_n * h`
   - `local_b = local_a + local_n * h`

2. **Compute Local Integral**

   - Each process calculates its partial sum using the trapezoidal formula:

$$local\_integral = \frac{h}{2}\left[f(local\_a) + 2\sum_{i=1}^{local\_n-1} f(local\_a + i \cdot h\right.$$

3. **Communication and Summation**

   - Partial integrals from all processes are collected to process 0 using `MPI_Reduce()` with `MPI_SUM`.
     Process 0 computes the total integral by summing all local integrals.

4. **Finalize MPI**

   - **Call `MPI_Finalize()` to terminate MPI environment.**

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

double f(double x) { return x*x; } // Example function: f(x) = x^2
```

```
int main(int argc, char *argv[]) {
    int rank, size, n = 1000;
    double a = 0.0, b = 10.0, h, local_a, local_b;
    double local_integral = 0.0, total_integral = 0.0;
    int local_n;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    h = (b - a) / n;
    local_n = n / size;
    local_a = a + rank * local_n * h;
    local_b = local_a + local_n * h;

    // Compute local integral
    local_integral = (f(local_a) + f(local_b)) / 2.0;
    for (int i = 1; i < local_n; i++) {
        local_integral += f(local_a + i * h);
    }
    local_integral *= h;

    // Reduce all local integrals to total_integral in process 0
    MPI_Reduce(&local_integral, &total_integral, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Approximate integral = %lf\n", total_integral);
    }

    MPI_Finalize();
    return 0;
}
```

Working Explanation

1. The interval `[a, b]` is **split among all processes**, so each process handles only a part of the computation.
2. Each process computes its **local trapezoidal sum** independently, reducing computation time.
3. `MPI_Reduce()` is used to **combine local results** into the final total integral.
4. Only process 0 prints the result, ensuring a **single output**.
5. This parallel approach significantly **reduces execution time**, especially for large `n`.

Significance

- Efficiently **utilizes multiple processors** to compute integrals in parallel.
- Reduces computation time compared to sequential implementation.
- Demonstrates **MPI point-to-point and collective communication** concepts (especially `MPI_Reduce`).

---

b. **Compare the traditional global sum using process 0 as collector with tree structured global sum.**

In parallel computing, after each process computes a local value (e.g., a local integral in the trapezoidal rule), these values must be combined into a global sum. Two common methods are traditional global sum and tree-structured global sum.

1. Traditional Global Sum (Process 0 as Collector)

Working Principle:

- All processes send their local results to process 0.
- Process 0 receives the values and sums them to compute the total.

Steps:

1. Each process executes MPI_Send() to send its local result to process 0.
2. Process 0 executes multiple MPI_Recv() calls to collect results.
3. Process 0 sums all received values and obtains the global sum.

Characteristics:

- Simple to implement
- Communication bottleneck at process 0
- Scalability is poor as the number of processes increases

Example:

```
if (rank != 0) {
   MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
} else {
   total_sum = local_sum;
   for (i = 1; i < size; i++) {
      MPI_Recv(&recv_sum, 1, MPI_DOUBLE, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      total_sum += recv_sum;
   }
}
```

---

2. Tree-Structured Global Sum

Working Principle:

- Processes are organized in a binary tree structure.
- In each step, pairs of processes exchange and sum their local values.
- The result moves up the tree until process 0 (root) obtains the final sum.

Steps:

1. Processes are paired (e.g., 0 & 1, 2 & 3) and each pair computes partial sums.
2. Partial sums are sent to the next higher level in the tree.

3. Repeated until the root process (process 0) receives the final sum.

Characteristics:

- Reduces communication overhead from O(p) to O(log p), where p = number of processes
- Better scalability for large number of processes
- Slightly more complex to implement than the traditional method

Example:

- For 8 processes, the tree-based sum completes in $\log_2 8 = 3$ steps instead of 7 steps in the traditional method.

---

3. Comparison Table

Feature	Traditional Global Sum	Tree-Structured Global Sum
Communication Pattern	All processes → Process 0	Pairwise, hierarchical (binary tree)
Number of Communication Steps	$p - 1$	$\log_2 p$
Communication Bottleneck	Process 0	Minimal, distributed
Scalability	Poor for large p	Excellent
Implementation Complexity	Simple	Moderate
Suitable For	Small number of processes	Large-scale parallel systems

MODULE 4

Q.7 **a.** **Explain the structure and working of an OpenMP "Hello world" program.**

OpenMP (Open Multi-Processing) is an API for shared-memory parallel programming that allows a program to run multiple threads concurrently. A basic OpenMP “Hello World” program includes three main components: first, the OpenMP header #include <omp.h> which provides OpenMP functions; second, the main function where the program execution begins; and third, the parallel region directive #pragma omp parallel that defines a block of code to be executed by multiple threads. Inside the parallel region, each thread can identify itself using omp_get_thread_num() and execute statements independently. For example:

```
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel
   {
      int thread_id = omp_get_thread_num();
      printf("Hello World from thread %d\n", thread_id);
   }
   return 0;
}
```

In this program, the parallel region forks multiple threads, each executing the printf() statement simultaneously. The number of threads can be controlled using OMP_NUM_THREADS or omp_set_num_threads(). After executing the parallel block, all threads join back to a single thread, continuing sequential execution. The output shows a “Hello World” message from each thread along with its thread ID, though the order may vary due to concurrent execution. This program demonstrates the fork-join model, shared memory usage, and thread identification in OpenMP, making it a fundamental example of parallel programming.

---

b. **Explain the purpose of the reduction clause in OpenMP with example.**

In OpenMP, the reduction clause is used to perform parallel reduction operations on a variable that is updated by multiple threads inside a parallel region. Its primary purpose is to avoid race conditions when multiple threads try to update a shared variable simultaneously while

---

computing a sum, product, maximum, minimum, or any associative operation.

Working of reduction Clause

1. Private Copy per Thread:
    - OpenMP automatically creates a private copy of the reduction variable for each thread.
2. Local Computation:
    - Each thread performs its computation on its private copy without interfering with other threads.
3. Combine Results:
    - At the end of the parallel region, OpenMP combines all private copies into the original shared variable using the specified operation (sum, product, max, min, etc.).

#pragma omp parallel for reduction(operator : variable)

```
#include <stdio.h>
#include <omp.h>

int main() {
   int i;
   int sum = 0;
   int arr[100];

   // Initialize array
   for(i = 0; i < 100; i++) arr[i] = i + 1;

   #pragma omp parallel for reduction(+:sum)
   for(i = 0; i < 100; i++) {
      sum += arr[i];
   }

   printf("Sum of array elements = %d\n", sum);
   return 0;
}
```

Each thread computes a partial sum of its portion of the array.

The reduction(+:sum) clause ensures that OpenMP combines all partial sums safely into the original sum variable at the end of the parallel region.

This avoids race conditions and produces the correct total sum.

---

c. **Write a OpenMP program to calculate n-Fibonacci using tasks.**

```
#include <stdio.h>
#include <omp.h>

// Recursive function to calculate Fibonacci using OpenMP tasks
int fib(int n) {
   int x, y;

   if (n < 2)
      return n;

   #pragma omp task shared(x)
   x = fib(n - 1);

   #pragma omp task shared(y)
   y = fib(n - 2);

   #pragma omp taskwait // Wait for both tasks to finish
   return x + y;
}

int main() {
   int n = 10; // Example: calculate 10th Fibonacci number
   int result;

   // Parallel region with a single thread to start tasks
   #pragma omp parallel
   {
      #pragma omp single
      result = fib(n);  // Start the first task
   }

   printf("Fibonacci number F(%d) = %d\n", n, result);
   return 0;
}
```

Q.8 a. Define OpenMP. Explain the key features of OpenMP and its advantages over Pthreads.

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) for shared-memory parallel programming in C, C++, and Fortran. It provides compiler directives, library routines, and environment variables to write parallel programs easily for multi-core and multi-processor systems. OpenMP follows a fork-join model, where a master thread forks multiple parallel threads to execute a task, and then joins back to the master thread after completing the parallel region.

Key Features of OpenMP

1. **Shared Memory Parallelism**
   - All threads share the **same address space**, allowing easy access to global variables.
2. **Fork-Join Model**
   - Program starts with a **single master thread**.
   - When a parallel region begins, the master thread **forks worker threads** to execute code concurrently.
   - Threads **join** back after the parallel region ends.
3. **Compiler Directives (Pragmas)**
   - `#pragma omp parallel`, `#pragma omp for`, `#pragma omp task`, etc.
   - Allow easy **annotation of existing sequential code** for parallel execution without major rewriting.
4. **Synchronization Constructs**
   - `critical`, `atomic`, `barrier`, `taskwait`, etc., to prevent race conditions and manage dependencies.
5. **Thread Identification**
   - `omp_get_thread_num()` and `omp_get_num_threads()` to manage and identify threads dynamically.
6. **Reduction Clause**
   - Supports **safe aggregation of variables** (sum, product, max, min) across threads without race conditions.
7. **Environment Variables**
   - Control runtime parameters such as number of threads (`OMP_NUM_THREADS`), scheduling, and thread affinity.

.

Advantages of OpenMP over Pthreads

Feature	OpenMP	Pthreads
Programming Model	Directive-based; simpler to write	API-based; requires explicit thread creation and management
Ease of Use	Easier to parallelize loops and sections	Manual handling of threads, locks, and synchronization
Shared Memory Support	Automatic shared variable handling	Programmer must manage shared vs private variables
Synchronization	Built-in constructs (`critical`, `atomic`, `barrier`)	Manual use of mutexes, condition variables
Portability	Supported by many compilers (GCC, Intel, Clang)	Platform-dependent; more effort to port
Dynamic Thread Management	Automatic scheduling of ads	Must be explicitly implemented

b. Explain the concept of variable scope in OpenMP with suitable example.

In OpenMP, variable scope determines whether a variable is shared among all threads or private to each thread within a parallel region. Understanding variable scope is essential to avoid race conditions and ensure correct parallel execution.

OpenMP supports two main types of variable scope:

1. Shared Variables
   - A shared variable is accessible by all threads in a parallel region.
   - Any modification by one thread is visible to all other threads.
   - Typically, global variables and variables defined outside the parallel region are shared by default.

Example:

```
#include <stdio.h>
#include <omp.h>

int main() {
   int sum = 0; // shared variable by default
   #pragma omp parallel for shared(sum)
   for (int i = 1; i <= 5; i++) {
      sum += i;
```

```
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

2. Private Variables
   - A private variable is unique to each thread.
   - Each thread gets its own copy; changes by one thread do not affect others.
   - Useful for loop counters or temporary computations within a thread.

Example:

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for private(i)
    for (int i = 1; i <= 5; i++) {
        printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

3. Default Scoping Rules
   - shared: Variables outside the parallel block are shared by default.
   - private: Variables declared inside the parallel block are private by default.
   - OpenMP also supports firstprivate (initialize private copy with original value) and lastprivate (store last private value into shared variable).

---

c. Estimate the value of π.

Estimating the Value of π

The value of π can be **numerically estimated** using the **Monte Carlo method** or **numerical integration**. Here, we explain the **numerical integration approach** using the function:

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

1. Divide the interval `[0,1]` into `n` small subintervals of width `h = 1/n`.
2. Use the **midpoint or trapezoidal rule** to approximate the integral:

$$\pi \approx 4 \sum_{i=0}^{n-1} \frac{1}{1+x_i^2} \cdot h$$

3. Increasing `n` improves the accuracy.

Parallel Program Using OpenMP

```
#include <stdio.h>
#include <omp.h>

int main() {
    int n = 1000000;
    double pi = 0.0, h = 1.0 / n;

    #pragma omp parallel for reduction(+:pi)
    for (int i = 0; i < n; i++) {
        double x = (i + 0.5) * h;
        pi += 4.0 / (1.0 + x * x);
    }

    pi *= h;
    printf("Estimated value of pi (parallel) = %lf\n", pi);
    return 0;
}
```

Sample Output (n = 1,000,000)

```
Estimated value of pi = 3.141592
```

MODULE 5

Q.9 **a.** Define GPU and GPGPU. Explain the need for GPGPU.

Definition of GPU and GPGPU

GPU (Graphics Processing Unit):
A GPU is a specialized processor designed for rendering graphics and performing computations on large amounts of data in parallel. It contains thousands of small, efficient cores optimized for data-parallel tasks, making it ideal for graphics rendering, image processing, and simulations.

GPGPU (General-Purpose Computing on GPU):
GPGPU refers to the use of a GPU to perform general-purpose computations beyond graphics. In GPGPU, the massively parallel architecture of GPUs is leveraged to accelerate scientific, engineering, and data-intensive applications, such as matrix multiplication, neural networks, or simulations.

Need for GPGPU

1. **Massive Parallelism**
   - Modern GPUs contain thousands of cores, allowing **thousands of threads to run concurrently**.
   - CPU architectures have far fewer cores, limiting parallel computation speed.
2. **High Throughput for Data-Parallel Tasks**
   - GPUs are optimized for **vector and matrix operations**, which are common in scientific and machine learning computations.
3. **Cost-Effective Performance**
   - Using GPUs for computation can achieve **significant speedup** at a lower cost compared to building large CPU clusters.
4. **Energy Efficiency**
   - GPUs deliver **higher FLOPS per watt** than traditional CPUs for data-parallel workloads.
5. **Support for HPC and AI Workloads**
   - Many high-performance computing (HPC), deep learning, and simulation applications require **massive computations** that are more efficiently handled by GPUs.

Example Applications of GPGPU

- Scientific simulations (climate modeling, physics simulations)
- Machine learning and AI (deep neural network training)

- Financial modeling (Monte Carlo simulations)
- Image and video processing

b. Explain thread, block and grid in CUDA.

Threads, Blocks, and Grids in CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform that allows programming GPUs for general-purpose computation (GPGPU). CUDA uses a hierarchical thread organization to manage thousands of parallel threads efficiently. The three main components of this hierarchy are threads, blocks, and grids.

---

1. Thread

- A thread is the smallest unit of execution in CUDA.
- Each thread executes a kernel function, which is the function that runs on the GPU.
- Threads have a unique thread ID (threadIdx) used to identify which portion of the data they process.
- Threads are lightweight, allowing thousands to be created simultaneously.

---

2. Block

- A block is a collection of threads that execute together on a single Streaming Multiprocessor (SM).

- Threads within a block can communicate via shared memory and synchronize using __syncthreads().
- Each thread in a block has a unique thread index (threadIdx) within the block.
- Blocks can be 1D, 2D, or 3D depending on the application.

Example:

dim3 threadsPerBlock(16, 16); // 16x16 = 256 threads per block

3. Grid

- A grid is a collection of blocks that covers the entire dataset.
- Each block in the grid has a unique block index (blockIdx) and size (blockDim) that helps calculate the global thread ID.
- Grids can also be 1D, 2D, or 3D, depending on the problem.

Global thread ID calculation:

int globalThreadId = threadIdx.x + blockIdx.x * blockDim.x;

---

c. Explain CUDA vector addition program with suitable example.

Vector addition is a common example to demonstrate parallel computing with CUDA. It involves adding two arrays (vectors) element-wise to produce a third vector.

- Given two vectors `A` and `B` of size `N`, compute:

$$C[i] = A[i] + B[i] \quad \text{for } i = 0 \text{ to } N-1$$

- Each element addition is **independent**, making it **ideal for parallel execution**.
- In CUDA, **each thread handles one element** of the vector.

#include <stdio.h>

```
#include <cuda.h>

__global__ void vectorAdd(int *A, int *B, int *C, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // Global thread ID
    if (tid < N)
        C[tid] = A[tid] + B[tid];                // Perform addition
}

int main() {
    int N = 1000;
    size_t size = N * sizeof(int);

    int *h_A = (int*)malloc(size);
    int *h_B = (int*)malloc(size);
    int *h_C = (int*)malloc(size);

    // Initialize vectors
    for (int i = 0; i < N; i++) {
        h_A[i] = i;
        h_B[i] = i * 2;
    }

    int *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Launch kernel
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
N);

    // Copy result back to host
```

```
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Display first 10 results
    for (int i = 0; i < 10; i++)
        printf("C[%d] = %d\n", i, h_C[i]);

    // Free memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);

    return 0;
}
```

Q.10 **a.** **Compare CUDA and OpenCL.**

Comparison of CUDA and OpenCL

CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) are two popular frameworks for parallel programming on GPUs and other accelerators, but they have important differences in terms of platform, portability, and usage.

CUDA: A parallel computing platform and programming model developed by NVIDIA for their GPUs. It allows developers to write GPU programs in C, C++, or Fortran.

OpenCL: An open standard for parallel programming developed by Khronos Group. It supports heterogeneous computing across CPUs, GPUs, FPGAs, and other devices from multiple vendors.

Comparison:

Feature	CUDA	OpenCL
Developer / Vendor	NVIDIA	Khronos Group (multi-vendor)
Hardware Support	Only NVIDIA GPUs	GPUs, CPUs, FPGAs, DSPs from vendors
Programming Language	C, C++, Fortran (extensions)	C-based kernel language, suppo in C/C++/Python
Portability	Platform-specific (NVIDIA only)	Platform-independent; runs on devices
Ease of Use	Easier to learn; high-level abstractions	More complex; requires explicit management
Performance	Highly optimized for NVIDIA GPUs	Performance may vary across de requires tuning
Memory Model	Shared memory, global memory, constant memory	Similar memory model but mor across devices
Ecosystem / Libraries	Rich libraries (cuBLAS, cuFFT, cuDNN)	Limited standard libraries; requi specific implementations
Community Support	Large NVIDIA-focused community	Cross-platform, growing commu

b. Explain Kernel with shared memory.

In CUDA, a kernel is a function executed on the GPU by multiple threads in parallel. Each thread executes the same kernel function but works on different data elements, identified by thread and block indices.

Shared memory is a type of fast, on-chip memory that is shared among threads within the same block. Using shared memory improves performance because it is much faster than global memory, but it is limited in size.

1. Concept of Shared Memory

- Scope: Shared memory is visible to all threads in a block, but not accessible by threads in other blocks.

- Purpose: Reduces global memory accesses, enabling faster data sharing and cooperation among threads.

- Synchronization: Threads must synchronize using __syncthreads() when reading/writing shared memory to ensure correct results.

---

2. Structure of Kernel with Shared Memory

1. Declare shared memory inside the kernel using the __shared__ keyword.
2. Load data from global memory to shared memory (optional, for reuse).
3. Perform computation using shared memory.
4. Write results back to global memory.

```
#include <stdio.h>
#include <cuda.h>

__global__ void vectorAddShared(int *A, int *B, int *C, int N) {
    extern __shared__ int temp[];   // Shared memory array
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N) {
        // Load data into shared memory
        temp[threadIdx.x] = A[tid] + B[tid];
        __syncthreads(); // Ensure all threads have written to shared
memory

        // Copy result from shared memory to global memory
        C[tid] = temp[threadIdx.x];
    }
}

int main() {
    int N = 1024;
    size_t size = N * sizeof(int);

    int *h_A = (int*)malloc(size);
    int *h_B = (int*)malloc(size);
```

```
    int *h_C = (int*)malloc(size);

    for (int i = 0; i < N; i++) {
        h_A[i] = i;
        h_B[i] = i * 2;
    }

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;

    vectorAddShared<<<blocksPerGrid, threadsPerBlock,
threadsPerBlock * sizeof(int)>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    printf("First 10 results:\n");
    for (int i = 0; i < 10; i++)
        printf("C[%d] = %d\n", i, h_C[i]);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);

    return 0;
}
```

c. **Explain Heterogeneous computing in detail.**

Heterogeneous computing refers to a computing system that uses more than one type of processor or cores to perform different types of tasks efficiently. Typically, it combines CPUs (Central Processing Units) and accelerators such as GPUs (Graphics Processing Units), FPGAs (Field Programmable Gate Arrays), or DSPs (Digital Signal Processors) in a single system to achieve higher performance and energy efficiency.

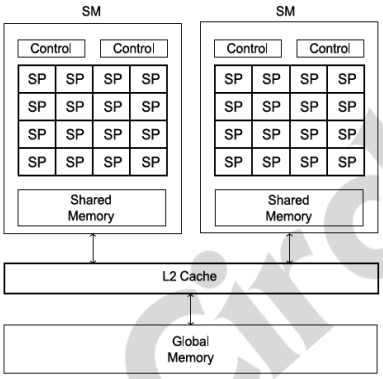


FIGURE 6.1
Simplified block diagram of a GPU.

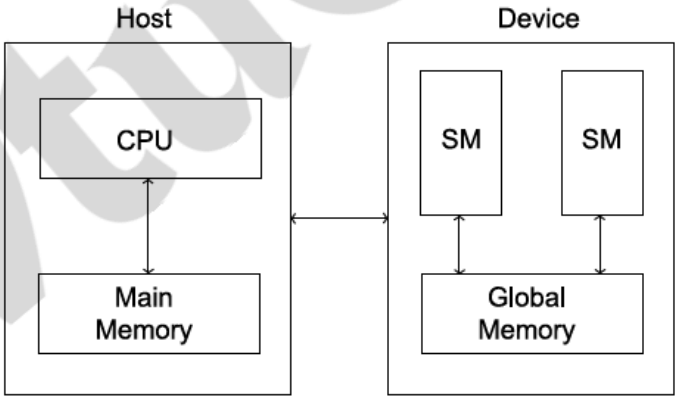


FIGURE 6.2
Simplified block diagram of a CPU and a GPU.

Concept

- Different processors are optimized for different workloads:
  CPU: General-purpose, sequential tasks, control-intensive operations.
  GPU: Data-parallel tasks, computation-intensive operations.
  FPGA/DSP: Specialized tasks requiring custom hardware acceleration.
  Tasks are divided based on their nature, and each processor executes tasks it is best suited for.

The system works collaboratively, with a CPU often acting as a host coordinating tasks and a GPU or accelerator performing heavy computations.

Architecture

A typical heterogeneous computing system consists of:

1. Host (CPU)
   Manages program execution, I/O, and task scheduling.
   Handles sequential or control-intensive computations.

2. Device (GPU/Accelerator/FPU)

Executes data-parallel or compute-intensive tasks.
Provides massive parallelism for high throughput.

3. Memory Hierarchy
   CPU and GPU have separate memory spaces (global memory on GPU, RAM on CPU).
   Data must be transferred between host and device efficiently.