GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING FOR WOMEN
(AUTONOMOUS)

(Affiliated to Andhra University, Visakhapatnam)

I M. Tech II Semester Regular Examinations, August- 2026

High Performance Computing-24CS21PE3E

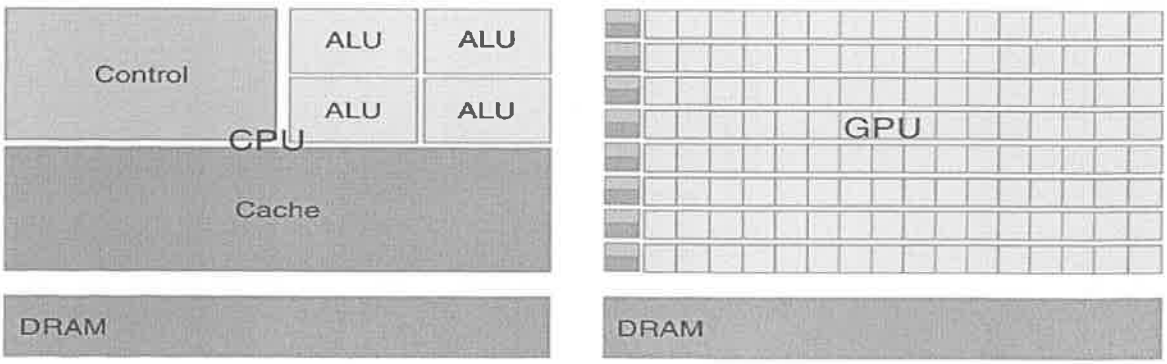
(Common to CSE, CSE(DS)

.....

Key of Valuation

1(a) Explain how GPUs act as parallel computers compared to CPUs.

- Graphics Processing Units(GPUs) have evolved into powerful parallel computing devices that emphasizes high throughput for parallel applications.
- Unlike CPUs which follow a multi-core design is optimized for sequential execution.
- GPUs contain hundreds of simpler cores capable of executing massive number of threads simultaneously.
- GPUs are widely used for computationally intensive tasks, while CPUs handle sequential operations.
- GPUs are widely used for computationally intensive tasks, while CPUs handle sequential operations, leading to hybrid CPU-GPU computing models like CUDA.



- GPUs are widely used for computationally intensive tasks, while CPUs handle sequential operations, leading to **hybrid CPU-GPU computing models** like CUDA. Their large market presence, cost-effectiveness, improved IEEE floating-point support, and ease of programming (post-CUDA) have made GPUs a practical and dominant platform for parallel computing across industries.

1(b) Describe the need for more speed and parallelism in computing.

- The demand for higher speed and parallelism arises because many modern and future applications—such as scientific simulations, high-definition video processing, gaming, and advanced user interfaces—require massive computational power and process large volumes of data simultaneously.
- GPUs can provide significant speedups (often 10× to 100× or more) for applications with **data parallelism**, but the overall performance gain depends on how much of the program can be parallelized (as explained by the Amdahl's Law). If only a small portion is parallel, speedup is limited, whereas applications with over 99% parallel execution can achieve dramatic improvements.
- Future "super applications" like molecular simulations, 3D visualization, and realistic gaming rely heavily on parallel processing to simulate real-world complexity. However, challenges such as memory bandwidth limitations and non-parallelizable (sequential) code sections must be managed carefully.
- Therefore, optimal performance is achieved using a **hybrid CPU-GPU approach**, where CPUs handle sequential tasks and GPUs accelerate parallel portions, supported by programming models like CUDA to efficiently exploit large-scale parallelism.

2(a) Discuss the concept of Unified Graphics and Computing Processors.

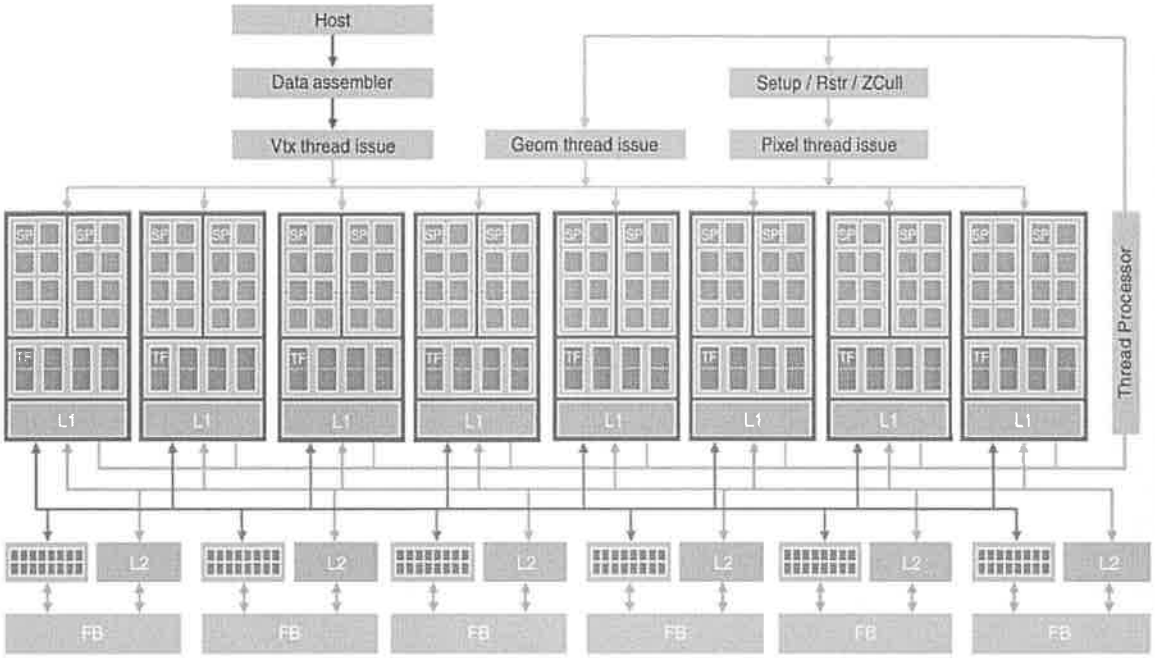
Features:

1. Shared Architecture
2. Parallel Processing
3. Heterogeneous Computing

Advantages:

1. Faster Data Processing

2. Lower Power Consumption
3. Cost Effective
4. Better Performance for multimedia and AI tasks



2(b) Explain the parallel programming languages and models in GPU computing.

- Parallel programming models have evolved to support different computing architectures, with **MPI** and **Open MP** being the most widely used. MPI is designed for distributed systems without shared memory, requiring explicit message passing between nodes, which enables large-scale scalability (even over 100,000 nodes) but involves high programming complexity.
- Open MP, in contrast, uses shared memory and is easier to program but has limited scalability due to thread management overhead and cache coherence issues.
- **CUDA** combines advantages of both by providing shared memory within GPUs, simple thread management, and high scalability, making it highly effective for massively parallel applications, although it supports a narrower range of applications compared to Open MP.

- Additionally, **Open CL** was introduced as a cross-platform standard supported by major companies, allowing portability across hardware, but it is more complex and often less efficient than CUDA. Overall, CUDA remains widely preferred for performance-critical GPU computing, while MPI and Open MP continue to serve distributed and shared-memory systems respectively.

3(a) Explain the UMA and ceNUMA shared memory architectures.

- Shared memory computers allow multiple processors to access a common memory space, making communication between processors faster and easier. In Uniform Memory Access (UMA) systems, all processors access memory with equal latency, which simplifies programming but may not scale well.
- In Cache-Coherent Non-Uniform Memory Access (ccNUMA) systems, memory access time varies depending on the memory location relative to the processor, but cache coherence protocols ensure consistency across caches. These systems provide better scalability than UMA while maintaining a shared memory model.
- The simplest implementation of a UMA system is a dual core processor in which two CPUs on one chip share a single path to memory.
- It is very common in HPC to use more than one chip in a compute node, be they single core or multicore.

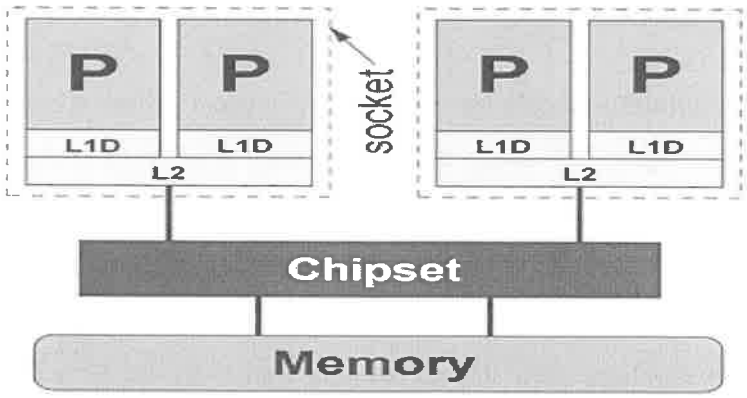


Figure 4.4: A UMA system in which the FSBs of two dual-core chips are connected separately to the chipset.

- A locality domain(LD) is a set of processors cores together with locally connected memory. This memory can be accessed in a more efficient way without resorting to a network of any kind.
- Multiple LDs are linked via coherent interconnect which allows transparent access from any processor to any other processors memory.

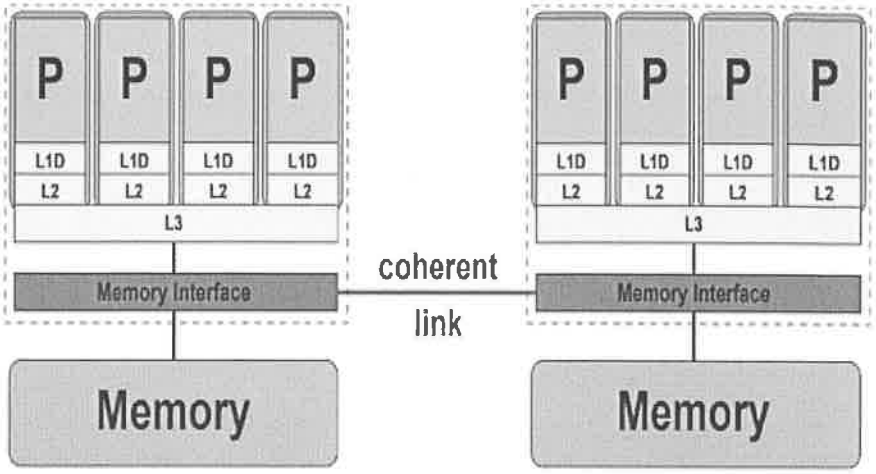


Figure 4.5: A ccNUMA system with two locality domains (one per socket) and eight cores.

3(b) Describe the basic steps of parallelization with an example

- Parallelization is the process of dividing a problem into smaller tasks that can be executed simultaneously to reduce execution time.
- It involves identifying independent or partially dependent tasks, distributing them across multiple processors, and managing synchronization and communication between them.
- Key aspects include – task decomposition, load balancing, minimizing communication overhead and avoiding data dependencies.
- Functional Parallelism is also known as Task Parallelism, involves executing different tasks simultaneously on different processors.
- Each processor performs a distinct operation, possibly on the same or different datasets.
- This approach can be useful when a problem can be divided into independent tasks such as pipelining process or multistage computations.
- Functional Parallelism enhances performance by utilizing multiple processors efficiently but requires co-ordination to manage dependencies between tasks.

4(a) Illustrate the architecture of Distributed Memory Computers.

- Distributed Memory Computers consists of multiple processors each with its own private memory and communication occurs through message passing.
- There is no global memory space, so processors exchange data explicitly using communication protocols such as MPI (Message Passing Interface).
- These systems are highly scalable and are commonly used in clusters and super computers but programming them is more complex.

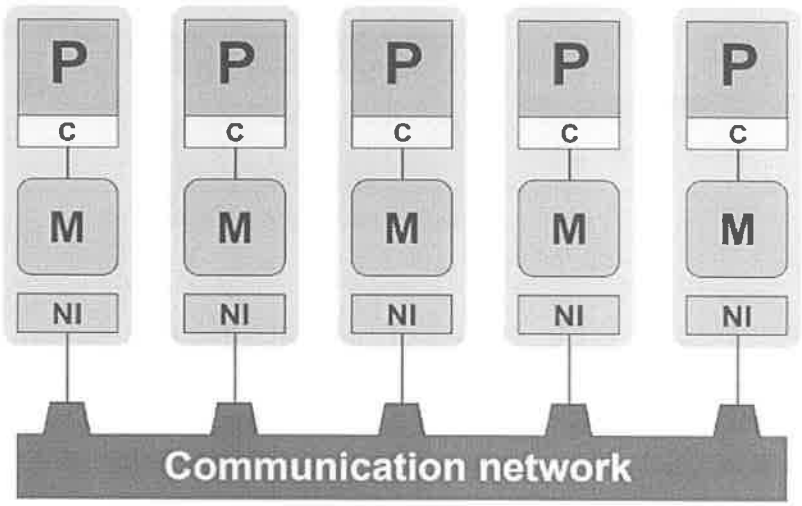


Figure 4.7: Simplified programmer's view, or "programming model," of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network. No process can access another process' memory (M) directly, although processors may reside in shared memory.

4(b) Explain Hierarchical (hybrid) parallel systems with a diagram.

- Hierarchical(hybrid) systems combine both shared and distributed memory architectures to achieve better performance and scalability.
- Typically, nodes in a cluster use shared memory (multicore processors), while communication between nodes is handled through distributed memory techniques.
- This approach leverages the advantage of both models, providing efficient intra-node communication and scalable inter-node communication.
- Hybrid programming models such as Open MP (shared memory) and MPI(distributed memory) are often used in these systems.

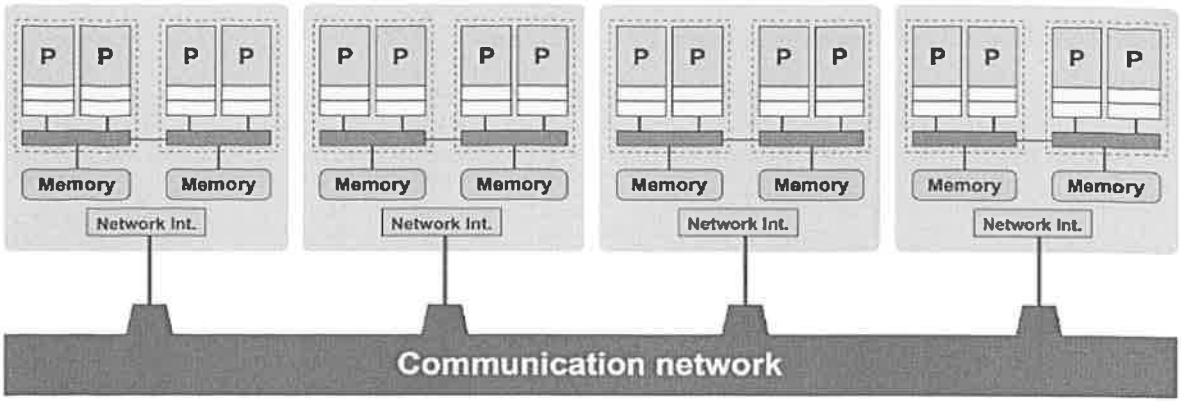
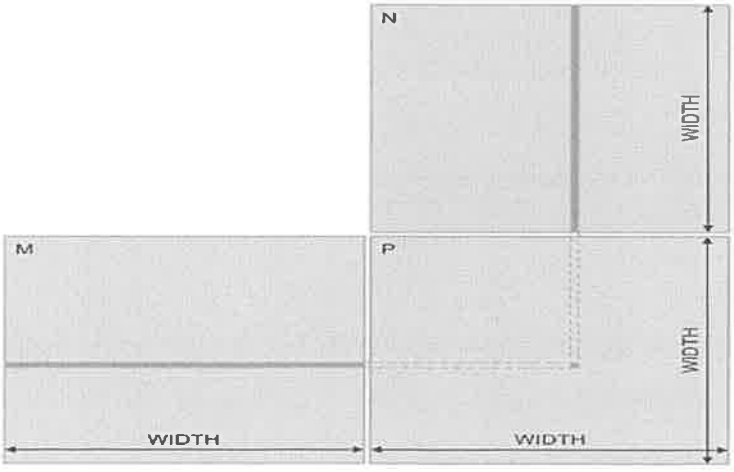


Figure 4.8: Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance "sweet spot" and are thus found in many commodity clusters.

5(a) Explain the concept of Data Parallelism in CUDA with an example.

- Data Parallelism is a programming paradigm where the same operation is applied simultaneously to multiple data elements.
- In GPU Computing, large datasets are divided into smaller chunks and each GPU thread processes a portion of data independently.
- This approach significantly improve performance for tasks like matrix operations, vector addition and image processing.



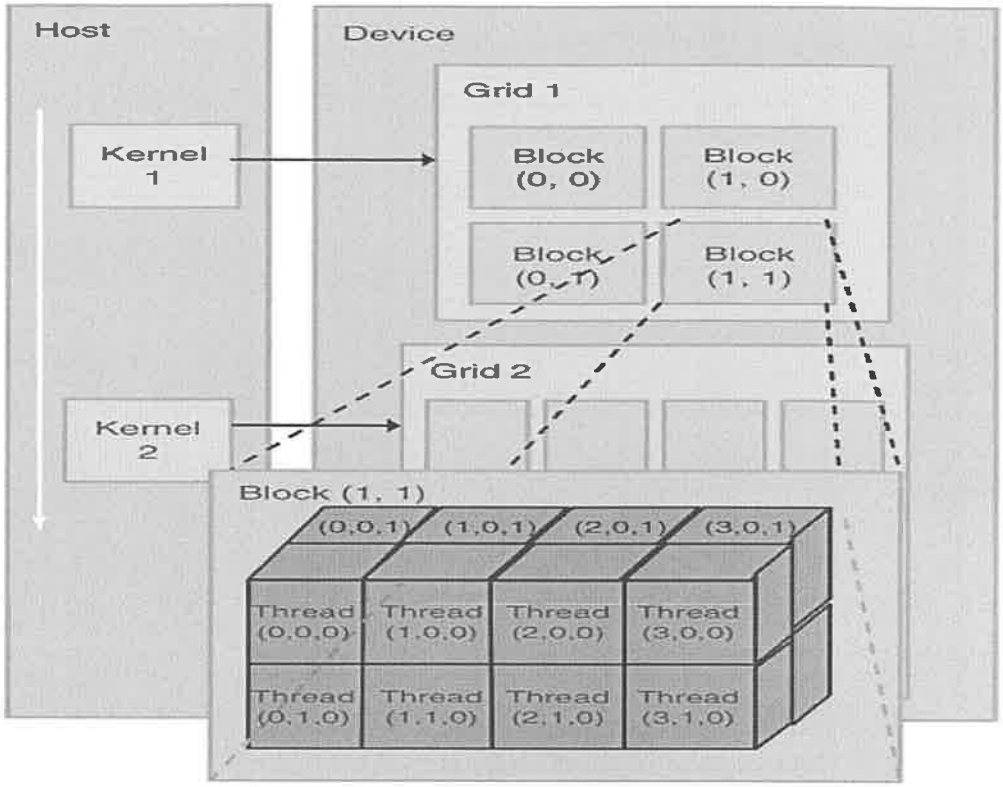
- CUDA leverages data parallelism by executing thousands of threads concurrently.

5(b) Describe the device memories and host-device data transfer in CUDA.

- CUDA provides different types of device memory such as global memory, shared memory, constant memory and registers each with different access speeds and scope.
- Global memory is large but slower, while shared memory is faster and shared among threads in a block.
- Efficient data transfer between host and device using functions like cuda Memcpy() is crucial, as memory transfer can become a performance bottleneck. Proper memory management and minimizing transfers are key to achieve high performance in CUDA programs.
- CUDA provides device memory such as global memory, shared memory, constant memory, and registers, each with different access speeds and scope. Global memory is large but slower, while shared memory is faster and shared among threads in a block.
- Efficient data transfer between host and device using functions like cudaMemcpy () is crucial, as memory transfer can become a performance bottleneck. Proper memory management and minimizing transfers are key to achieving high performance in CUDA programs.

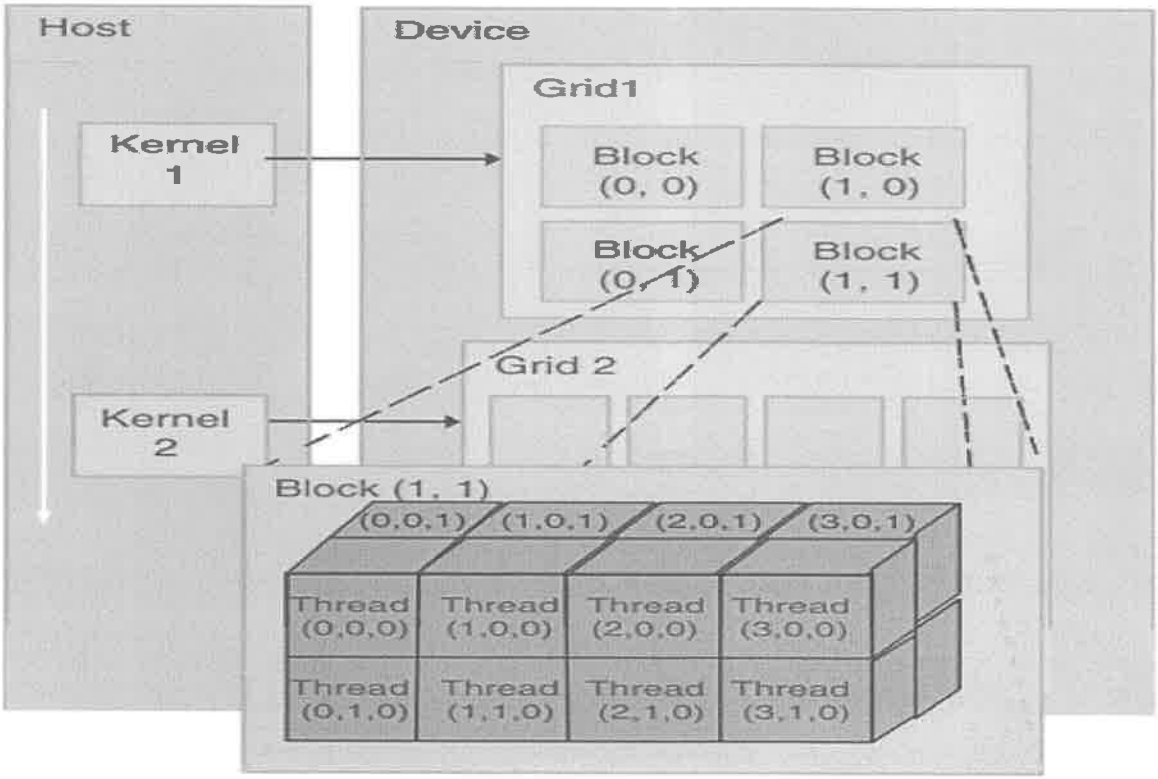
6(a) Explain the CUDA thread organization with a neat diagram.

- CUDA organizes threads hierarchically into grids, blocks and threads.
- A grid consists of multiple blocks, and each block contains multiple threads.
- Threads within the same block can cooperate via shared memory and synchronization, while blocks execute independently.
- This hierarchical structure allows scalability and efficient mapping of problems onto GPU hardware.
- Thread indexing using block Idx, thread Idx and Block Dim helps in identifying each thread role in computation.



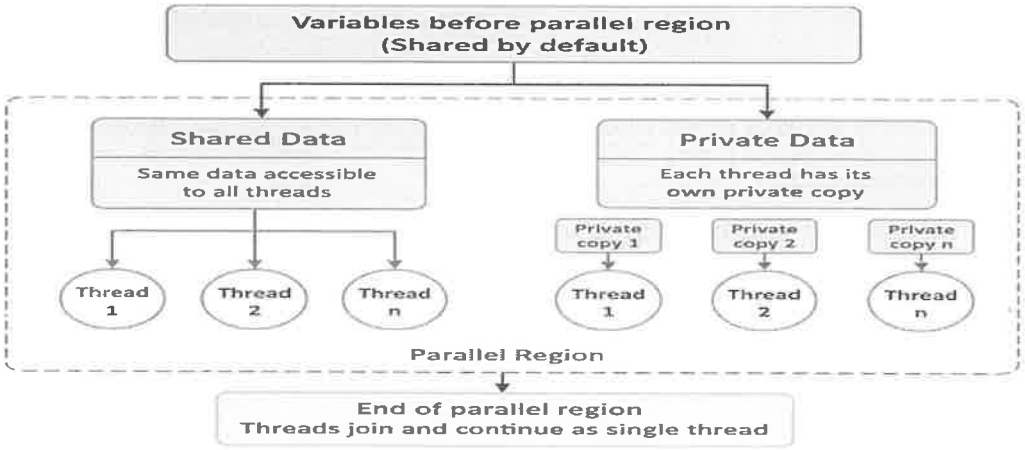
6(b) Discuss how kernel functions are defined and launched in CUDA.

- Kernel functions in CUDA are special functions executed on the GPU and defined using the __global__ keyword.
- When a kernel is launched, it runs across a large number of threads in parallel.
- Each thread executes the same code but operates on different data using unique thread indices.
- Thread level parallelism enables GPUs to perform massive computations efficiently and synchronization mechanisms are used when threads need to coordinate with a block.
- When a block is invoked or launched, it is executed as grid of parallel threads. The launch of kernel creates grid 1.
- Each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation.



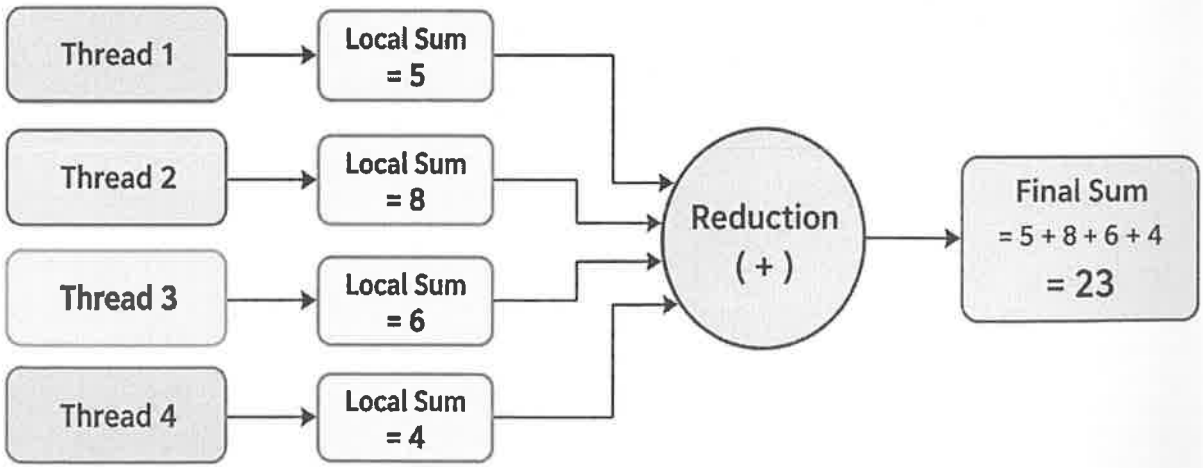
7(a) Explain data scoping in OpenMP with suitable examples.

- Data scoping determines whether a variable is shared or private among threads.
- Variables that exist before a parallel region are shared by default.
- For effective parallel execution, some variables need to be private to each thread.
- OpenMP provides a separate stack for each thread to store private variables.



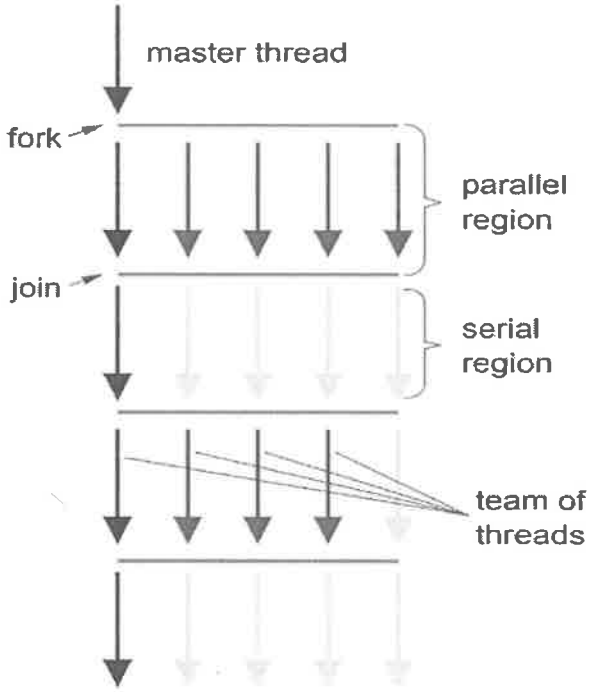
7(b) Describe the reduction operations in OpenMP with an example.

- A reduction combines multiple partial results into a single final result.
- It is commonly used in loops where several threads contribute to one variable.
- OpenMP provides the REDUCTION clause to handle this efficiently.
- The reduction variable is automatically made private for each thread.
- Each thread calculates its own partial result.
- At the end of the parallel region, OpenMP combines all partial results into the shared variable.
- **Common reduction operations include:**
- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Logical operations

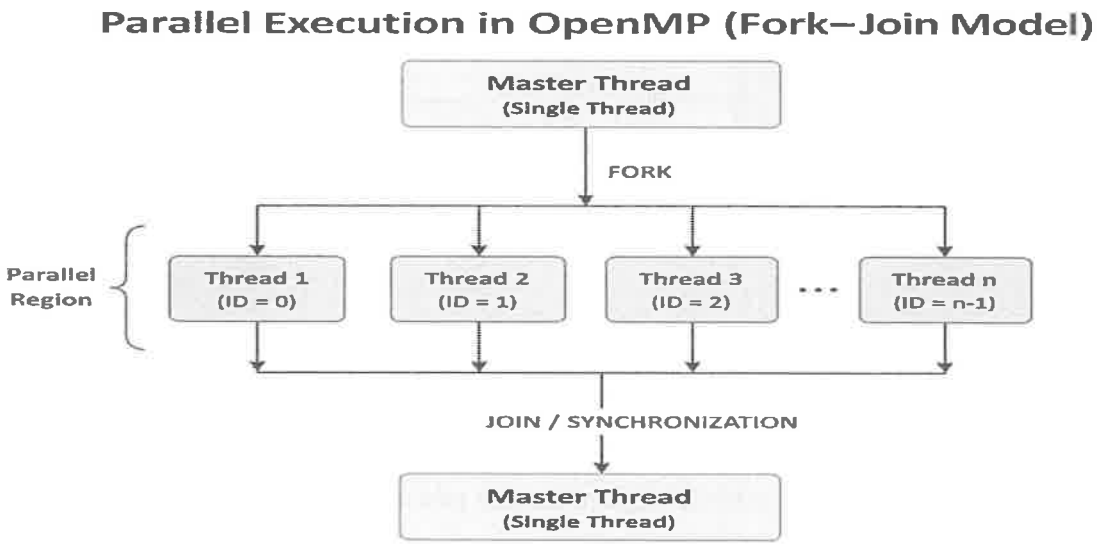


8(a). Explain the OpenMP parallel execution model with a diagram.

- The master thread starts the program.
- At a parallel region, the master thread forks a team of threads.
- Threads execute tasks concurrently using shared memory.
- At the end of the parallel region, threads join the master thread.
- The number of threads can vary between different parallel regions.

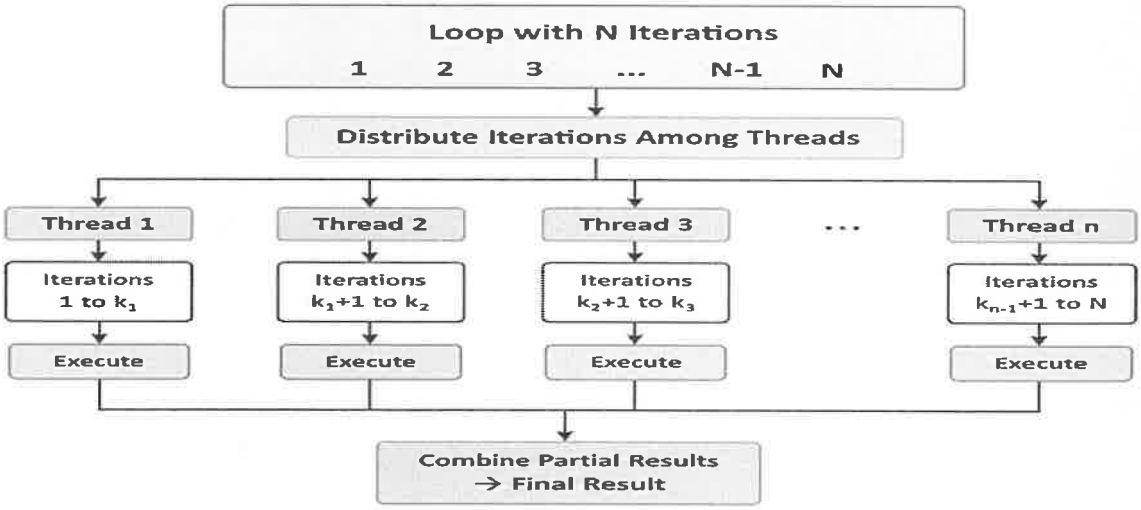


- Allows multiple threads to execute tasks concurrently.
- Helps reduce execution time and improve performance.
- Uses #pragma omp parallel to create a team of threads.
- The master thread initiates the parallel region.
- Each thread executes the code inside the parallel region.



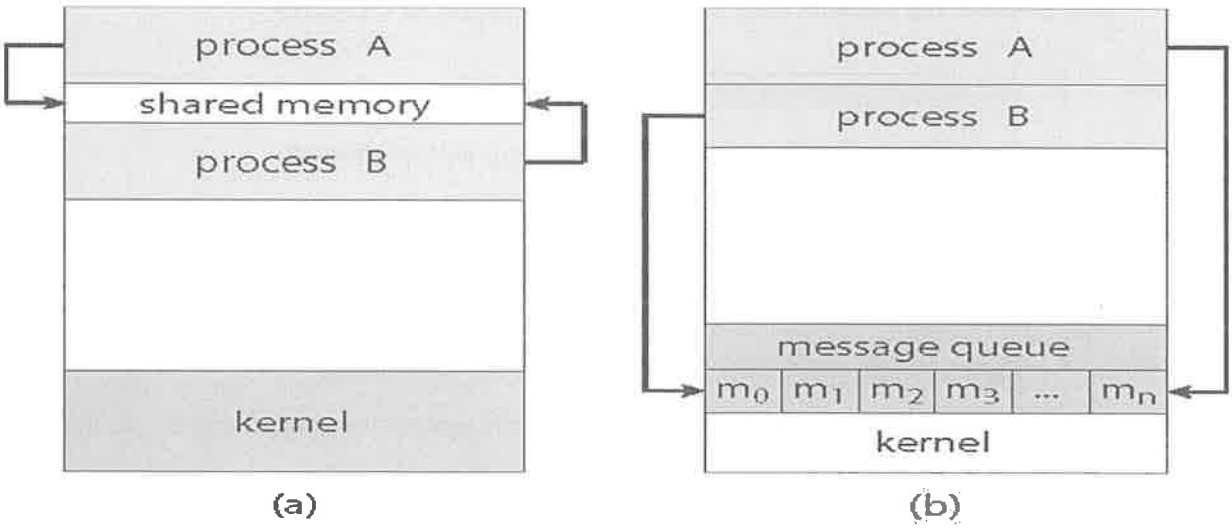
8(b) Discuss the various loop scheduling techniques in OpenMP.

- Loops are natural candidates for parallelization when individual iterations are independent.
- OpenMP divides the loop iterations among multiple threads.
- Each thread receives its own set of iterations to execute.
- Threads perform their assigned iterations concurrently.
- The loop counter is automatically made private for each thread.
- Partial results can be combined to obtain the final result.
- Care must be taken when multiple threads update the same variable, as this can cause a race condition.
- OpenMP can combine parallel execution and loop work sharing using PARALLEL DO.



9(a) Explain the Message Passing Model and its characteristics.

- The **Message Passing Model** is a parallel programming model in which multiple processes communicate by sending and receiving messages. Each process has its own private memory, and data is exchanged explicitly through communication operations.
- This model is widely used in distributed memory systems, such as computer clusters and supercomputers, where processors cannot directly access each other's memory. It provides scalability, flexibility, and efficient execution of large-scale parallel applications.



Characteristics of Message Passing Model:

1. Distributed Memory – Each process has its own private/local memory.
2. Explicit Communication – Processes communicate explicitly using send and receive operations.
3. Process-Based Parallelism – A problem is divided among multiple processes.
4. Data Distribution – Data is divided and distributed among different processes.
5. Synchronization – Processes can synchronize their execution when required.
6. Scalability – Can be used with a large number of processors.
7. Communication Overhead – Sending and receiving messages requires communication time.

9(b) Describe point-to-point communication in MPI with an example.

- Point-to-point communication is the basic communication mechanism in MPI, where one process sends a message directly to another process. The primary functions used are MPI_Send() for sending data and MPI_Recv() for receiving data.
- Each communication involves a sender, a receiver, a message, a tag, and a communicator. This communication method is suitable for exchanging data between specific processes and forms the foundation for more complex communication patterns.
- `MPI_Send()` → sends data from one process to another.

- `MPI_Recv()` → receives data from another process

Example

Consider two processes, P0 and P1.

- P0 sends the number 100 to P1.
- P1 receives the number and displays it.

10(a) Explain the collective communication operations in MPI.

- Collective communication involves all processes within a communicator participating in a communication operation. MPI provides functions such as **MPI_Bcast()** for broadcasting data, **MPI_Scatter()** for distributing data, **MPI_Gather()** for collecting data, **MPI_Reduce()** for combining results, and **MPI_Barrier()** for synchronization.
- These operations simplify communication, improve performance, and optimize data movement among multiple processes in parallel applications.

Collective Communication Operations in MPI:

1. **MPI_Bcast()** – Broadcast data from one process to all processes.
2. **MPI_Scatter()** – Distribute different portions of data from one process to all processes.
3. **MPI_Gather()** – Collect data from all processes to one process.
4. **MPI_Reduce()** – Combine data from all processes and give the result to one process.
5. **MPI_Allreduce()** – Combine data from all processes and give the result to all processes.
6. **MPI_Allgather()** – Collect data from all processes and distribute the combined data to all processes.
7. **MPI_Alltoall()** – Exchange data among all processes.
8. **MPI_Barrier()** – Synchronize all processes.

10(b) Compare blocking and non-blocking communication in MPI.

- Nonblocking point-to-point communication allows a process to initiate a send or receive operation and continue executing without waiting for the communication to complete. MPI provides functions such as **MPI_Isend()** and **MPI_Irecv()** for nonblocking communication, along with **MPI_Wait()** and **MPI_Test()** to check for completion.
- This approach enables overlap of communication and computation, reduces idle time, improves processor utilization, and enhances the performance of parallel applications, especially in large distributed systems.

Blocking vs Non-Blocking Communication in MPI

	Blocking Communication	Non-Blocking Communication
Meaning	Process waits for communication to complete.	Process can continue execution while communication is in progress.
Functions	MPI_Send(), MPI_Recv()	MPI_Isend(), MPI_Irecv()
Waiting	Process may be blocked during communication.	Process is not blocked immediately.
Execution	Communication and computation are generally sequential.	Communication and computation can overlap.
Performance	May take more time when communication is slow.	Can improve performance by overlapping communication and computation.
Programming	Simple and easy to implement.	More complex to implement.
Completion	Operation completes before the program proceeds as required by the operation's semantics.	Completion is checked using MPI_Wait() or MPI_Test().

Prepared by: Dr.S.Sumahasan

Verified by:Dr.K.P.Naidu