

**MIT Art Design and Technology University**

**MIT School of Computing, Pune**

**Department of Computer Science and Engineering**

| **Lab Manual** |
| --- |

# **Course- HPC Lab**

**Class - L.Y. (SEM-I), Core, AIEC**

**Name of the Course Coordinator**

**Dr Rashmi S Nair**

|  |
| --- |

**Team Members**

1. Dr. Sharon Christa

2. Prof. Tushar Mane

3. Prof. Aman Sinha

**A.Y. 2024 - 2025**

**Lab Experiment List**

| **Sr. No.** | **Name of Experiment** | **CO** |
| --- | --- | --- |
| 1. | Familiarization with Linux commands | CO1 |
| 2. | Familiarization with SLURM commands | CO1 |
| 3. | Write an OpenMP program to print Hello world with thread ID. | CO2 |
| 4. | Write your first Parallel Program, with which you should be able to print your NAME from 4 underline cores. | CO2 |
| 5. | Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause.   * + - * The program should perform the following steps:       * Initialize OpenMP with 4 threads.       * Declare an integer variable val and initialize it to a value of 1234.       * Print the initial value of val outside the OpenMP parallel region.       * Enter an OpenMP parallel region using the omp parallel directive, with the firstprivate clause applied to the variable val.       * Inside the parallel region, each thread should print the current value of val, increment it by 1, and then print the updated value.       * Print the final value of val outside the parallel region. | CO2 |
| 6. | Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause.  Steps to follow :   * + - * Open text editor.       * write the below program in it.       * Save the file with .c extentation.       * Compile and execuate with given commands. | CO2 |
| 7. | Write a Parallel C program where the iterations of a loop should be scheduled statically across the team of threads. A thread should perform CHUNK iterations at a time before being scheduled for the next CHUNK of work. | CO2 |
| 8. | Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads. | CO2 |
| 9. | Write MPI Program to print "Hello World".  MPI program to send and receive Hello World messages from all other processes to a Root process and print the received messages. | CO3 |
| 10. | MPI program to send two numbers (array elements) per process to a Root process and print the received messages. | CO3 |
| 11. | MPI program to find sum of first N integers using any given number of processes. Example, N=10,000 and no. of processes can be 4 or 8 or 12 etc. | CO4 |
| 12. | MPI program to find sum of n integers on in which processors are arranged in ring topology using MPI point-to-point blocking communication library calls. | CO4 |
| 13. | Write a CUDA program to perform two matrix addition. | CO5 |
| 14. | Write a CUDA program to perform two matrix multiplication. | CO5 |

**List of Hardware / Software Requirements**

* **Hardware requirements:**

1. Laptop/Desktop machine
2. Internet facility

* **Software Requirements**

1. HPC Tutor

**Experiment No 1**

## **Experiment Title: Familiarization with Linux commands**

## Problem Statement:

## Familiarization with Linux commands [CO1]

#### Objectives

1. **Understand Basic Commands**: Gain familiarity with essential Linux commands for navigation, file manipulation, and system information.
2. **File and Directory Operations**: Learn how to create, delete, move, and copy files and directories.
3. **Permission Management**: Understand and modify file and directory permissions.
4. **Text Processing**: Use commands for searching, viewing, and editing text files.
5. **System Monitoring**: Learn commands to monitor system performance and resource usage.
6. **Networking Commands**: Understand basic networking commands for connectivity and troubleshooting.
7. **Scripting Basics**: Gain an introductory understanding of shell scripting to automate tasks.

#### Theory

Linux commands are the foundation of interacting with the Linux operating system. Mastery of these commands is essential for efficient system administration and usage. The key areas include:

1. **Navigation Commands**:
   * pwd: Print working directory.
   * ls: List directory contents.
   * cd: Change directory.
2. **File and Directory Operations**:
   * touch: Create a new file.
   * mkdir: Create a new directory.
   * rm: Remove files or directories.
   * cp: Copy files or directories.
   * mv: Move or rename files or directories.
3. **Permission Management**:
   * chmod: Change file permissions.
   * chown: Change file owner and group.
   * ls -l: Display detailed file information including permissions.
4. **Text Processing**:
   * cat: Concatenate and display file content.
   * less, more: View file content page by page.
   * grep: Search text using patterns.
   * awk, sed: Text processing and transformation.
5. **System Monitoring**:
   * top: Display running processes.
   * ps: Report a snapshot of current processes.
   * df: Report file system disk space usage.
   * du: Estimate file space usage.
6. **Networking Commands**:
   * ping: Check network connectivity.
   * ifconfig or ip addr: Display network interfaces.
   * netstat: Network statistics.
   * ssh: Secure shell for remote login.
7. **Scripting Basics**:
   * Shell scripts: Basic scripts to automate tasks, usually with .sh extension.
   * Common script commands: echo, read, if, for, while.

#### Conclusion

Familiarization with Linux commands provides a fundamental skill set for working effectively in a Linux environment. By understanding and practicing these commands, users can navigate the file system, manage files and directories, monitor system performance, handle permissions, and perform basic network troubleshooting. Additionally, learning the basics of shell scripting enables automation of repetitive tasks, significantly enhancing productivity.

The hands-on experience gained through practicing these commands helps build confidence and competence in using the Linux operating system, which is crucial for both personal use and professional development in fields such as system administration, software development, and IT operations.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

1. https://www.geeksforgeeks.org/linux-commands-cheat-sheet/
2. https://ubuntu.com/tutorials/command-line-for-beginners#1-overview

## Exercise Questions

#### Navigation and Basic Commands

1. What command would you use to display the current directory you are in?
2. How do you list all files, including hidden files, in a directory?
3. Which command changes your current directory to the parent directory?

#### File and Directory Operations

1. Write the command to create a new directory named projects.
2. How do you create an empty file named file.txt?
3. What is the command to remove a file named sample.txt?
4. Write the command to copy a file named report.doc to a directory named backup.
5. How do you move a file named data.csv from the current directory to the /tmp directory?

#### Permission Management

1. How would you change the permissions of a file script.sh to make it executable by the owner only?
2. Which command would you use to change the owner of a file document.txt to user john?
3. What does the command ls -l display?

#### Text Processing

1. How do you display the contents of a file notes.txt?
2. Which command would you use to view the contents of a large file logfile.log page by page?
3. Write the command to search for the term error in a file named system.log.
4. How can you replace all occurrences of the word foo with bar in a file textfile.txt using sed?

#### System Monitoring

1. What command displays a dynamic view of the system's processes?
2. How would you list all currently running processes?
3. Write the command to check the available disk space on the file system.
4. How can you check the disk usage of the current directory?

#### Networking Commands

1. How do you check the network connectivity to www.example.com?
2. Which command displays the IP addresses and network interfaces on your system?
3. What command would you use to connect to a remote server example.com using SSH as the user alice?

**Experiment No 2**

## **Experiment Title: Familiarization with SLURM commands**

## Problem Statement: Familiarization with SLURM commands

**Objectives**

Understand the Basics of SLURM: Gain a fundamental understanding of what SLURM (Simple Linux Utility for Resource Management) is, its purpose, and its components.

Learn Basic SLURM Commands: Familiarize with essential SLURM commands used for job submission, monitoring, and management.

Execute Jobs Using SLURM: Learn how to submit, monitor, and manage jobs using SLURM commands.

Understand Job Scheduling and Resource Allocation: Understand how SLURM schedules jobs and allocates resources.

Gain Practical Experience: Practice using SLURM commands in a real or simulated high-performance computing (HPC) environment.

**Theory**

SLURM (Simple Linux Utility for Resource Management) is an open-source, highly configurable job scheduler used to manage resources in large-scale compute clusters. It is designed to allocate compute resources, schedule jobs, and manage job queues efficiently.

**Key Components of SLURM:**

Slurmctld (Controller Daemon): Manages the entire cluster, handles job queues, and schedules jobs.

Slurmd (Compute Node Daemon): Runs on each compute node and manages job execution on that node.

Slurmdbd (Database Daemon): Manages the accounting and usage information in a database.

Slurm Commands: Tools for users to interact with the SLURM system.

Essential SLURM Commands:

sbatch: Submits a batch script to the SLURM scheduler.

Example: sbatch myscript.sh

salloc: Allocates resources for a job and runs a command in a new shell.

Example: salloc -N 1 -n 4

srun: Used to submit jobs for execution or to launch tasks within a job allocation.

Example: srun myprogram

scancel: Cancels a pending or running job.

Example: scancel job\_id

squeue: Shows the status of jobs in the queue.

Example: squeue -u username

sinfo: Provides information about nodes and partitions.

Example: sinfo

sacct: Displays accounting information for jobs.

Example: sacct -j job\_id

**Conclusion**

Familiarizing yourself with SLURM commands is crucial for efficiently managing and utilizing resources in a high-performance computing environment. Understanding the theory behind SLURM and practicing essential commands will allow you to effectively submit, monitor, and manage jobs on a compute cluster. As you gain more experience, you can explore advanced features of SLURM to optimize job scheduling and resource allocation, ultimately enhancing the performance and efficiency of your computational tasks.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 3**

## **Experiment Title: OpenMP program 1**

## Problem Statement: Write an OpenMP program to print Hello world with thread ID.

## **Objectives**

## Understand OpenMP: Gain a basic understanding of OpenMP (Open Multi-Processing), its purpose, and how it enables parallel programming.

## Learn Basic OpenMP Directives: Familiarize yourself with essential OpenMP directives for parallelization.

## Implement a Simple OpenMP Program: Write and execute a simple OpenMP program to print "Hello, World" along with the thread ID.

## Analyze Program Output: Understand how OpenMP handles thread creation and management by analyzing the program output.

### Theory

## OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It is widely used for writing parallel applications, taking advantage of multi-core processors. OpenMP uses a set of compiler directives, library routines, and environment variables to specify parallel regions, control thread behavior, and manage the execution of parallel tasks.

#### Key Components of OpenMP:

## Parallel Regions: Sections of code that run in parallel.

## Work-sharing Constructs: Distribute the execution of code blocks among threads.

## Synchronization Constructs: Ensure correct program execution by managing access to shared resources.

#### Basic OpenMP Directives:

## #pragma omp parallel: Defines a parallel region.

## #pragma omp for: Distributes loop iterations among threads.

## #pragma omp critical: Ensures that a code block is executed by only one thread at a time.

#### 

#### 

#### 

#### Conclusion

#### OpenMP provides a simple and efficient way to write parallel programs for shared-memory architectures. By using basic directives like #pragma omp parallel, you can quickly parallelize sections of your code. The "Hello, World" program demonstrates the fundamental concept of creating and managing threads with OpenMP, showcasing how each thread can execute a portion of the code and identify itself with a unique thread ID. This foundational knowledge will help you explore more advanced features of OpenMP and develop more complex parallel applications.

## Source Code and Output /Screenshots**:**

## ***(To be provided by the student)***

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## 

## Exercise Questions

Q1.

**Experiment No 4**

## **Experiment Title: OpenMP Program 2**

Problem Statement: Write your first Parallel Program, with which you should be able to print your NAME from 4 underline cores.

**Objectives**

**Understand Parallel Programming**: Gain a fundamental understanding of parallel programming concepts.

**Learn Basic OpenMP Directives**: Familiarize yourself with essential OpenMP directives for creating parallel regions and managing threads.

**Implement a Simple Parallel Program**: Write and execute a simple OpenMP program to print your name using 4 threads.

**Analyze Program Output**: Understand how OpenMP distributes tasks among threads by analyzing the program output.

**Theory**

Parallel programming involves dividing a problem into sub-problems, solving them simultaneously (in parallel), and combining the results. This approach leverages multiple processing cores to enhance performance and reduce execution time.

OpenMP (Open Multi-Processing) is an API that supports shared-memory parallel programming in C, C++, and Fortran. It allows developers to write parallel code using compiler directives, library routines, and environment variables.

**Key Concepts**:

**Parallel Regions**: Blocks of code executed by multiple threads.

**Thread Management**: Creating, executing, and synchronizing threads.

**Work-sharing Constructs**: Dividing work among threads.

### Conclusion

Writing your first parallel program with OpenMP provides a hands-on introduction to parallel programming concepts. By setting the number of threads and creating a parallel region, you can see how OpenMP distributes tasks among multiple threads. This simple program demonstrates the power of parallel programming, where multiple threads can execute code concurrently, leading to more efficient use of processing resources. This foundational knowledge sets the stage for exploring more advanced parallel programming techniques and applications.

## Source Code and Output /Screenshots:

## ***(To be provided by the student)***

| ***//Source code and output/screenshot should be available here*** |
| --- |

# Useful Resource:

## Exercise Questions

## 

**Experiment No 5**

## **Experiment Title: OpenMP Program 3**

## Problem Statement:

Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause.

The program should perform the following steps:

* Initialize OpenMP with 4 threads.
* Declare an integer variable val and initialize it to a value of 1234.
* Print the initial value of val outside the OpenMP parallel region.
* Enter an OpenMP parallel region using the omp parallel directive, with the firstprivate clause applied to the variable val.
* Inside the parallel region, each thread should print the current value of val, increment it by 1, and then print the updated value.
* Print the final value of val outside the parallel region.

## **Objectives**

## Understand OpenMP Clauses: Learn about the private and firstprivate clauses in OpenMP.

## Demonstrate Variable Scoping in Parallel Regions: Observe how variable scoping works within OpenMP parallel regions.

## Implement and Analyze a Parallel Program: Write and execute a program to see the effects of the firstprivate clause on a variable in a parallel region.

## **Theory**

## OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It allows developers to write parallel code using compiler directives, library routines, and environment variables.

## Key OpenMP Clauses:

## private: Specifies that each thread should have its own instance of a variable. Variables are uninitialized at the start of the parallel region.

## firstprivate: Similar to private, but each thread's private instance of the variable is initialized with the value of the variable before the parallel region.

## These clauses are used to control the scope and sharing of variables in parallel regions.

## **Conclusion**

## This program demonstrates the behavior of the firstprivate clause in OpenMP. By using firstprivate, each thread gets its own copy of the variable val, initialized with the value before entering the parallel region. The changes made to val by each thread are independent and do not affect the original variable or other threads' copies. This example highlights the importance of variable scoping and initialization in parallel programming, ensuring correct and predictable behavior when multiple threads operate concurrently.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 6**

## **Experiment Title: OpenMP Program 4**

## 

## Problem Statement:

Write a C program utilizing OpenMP directives to demonstrate the behavior of the private clause.

Steps to follow :

Open text editor.

write the below program in it.

Save the file with .c extentation.

Compile and execuate with given commands.

**Objectives**

Understand OpenMP Clauses: Learn about the private clause in OpenMP.

Demonstrate Variable Scoping in Parallel Regions: Observe how variable scoping works within OpenMP parallel regions using the private clause.

Implement and Analyze a Parallel Program: Write and execute a program to see the effects of the private clause on a variable in a parallel region.

**Theory**

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It allows developers to write parallel code using compiler directives, library routines, and environment variables.

Key OpenMP Clause:

private: Specifies that each thread should have its own instance of a variable. Variables are uninitialized at the start of the parallel region.

This clause is used to control the scope and sharing of variables in parallel regions, ensuring that each thread has its own copy of the variable and that changes made to the variable by one thread do not affect others.

**Conclusion**

This program demonstrates the behavior of the private clause in OpenMP. By using private, each thread gets its own uninitialized copy of the variable val. Changes made to val by each thread are independent and do not affect the original variable or other threads' copies. This example highlights the importance of variable scoping in parallel programming, ensuring correct and predictable behavior when multiple threads operate concurrently. Understanding how to use the private clause is crucial for managing data correctly in parallel regions, preventing unintended data sharing and race conditions.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 7**

## **Experiment Title: OpenMP Program 5**

## 

## Problem Statement: Write a Parallel C program where the iterations of a loop should be scheduled statically across the team of threads. A thread should perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.

**Objectives**

Understand Loop Scheduling in OpenMP: Learn how loop scheduling works in OpenMP, specifically the static scheduling.

Implement Static Scheduling: Write a parallel C program that uses static scheduling to divide loop iterations among threads.

Analyze Performance: Observe the behavior and performance of the program with different chunk sizes.

**Theory**

OpenMP (Open Multi-Processing) is an API that supports shared-memory parallel programming in C, C++, and Fortran. It allows developers to write parallel code using compiler directives, library routines, and environment variables.

**Key Concepts:**

Static Scheduling: In static scheduling, the iterations of a loop are divided into chunks of a fixed size, and each chunk is assigned to a thread before the loop starts. This method can lead to predictable performance and low overhead.

Chunk Size: The number of iterations a thread performs in one go before being scheduled for the next chunk of work.

**Conclusion**

This program demonstrates the use of static scheduling in OpenMP to distribute loop iterations among threads. By specifying a chunk size, each thread is assigned a fixed number of iterations to process at a time. This method provides predictable performance and minimizes scheduling overhead. The example illustrates how the chunk size affects the distribution of work among threads and helps understand the behavior of static scheduling in parallel programming. Understanding and applying loop scheduling techniques like static scheduling is crucial for optimizing the performance of parallel applications.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 8**

## **Experiment Title: OpenMP Program 6**

## 

## Problem Statement:

## Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads.

## Objectives

Understand Thread Creation and Management in OpenMP: Learn how to create and manage multiple threads in OpenMP.

Implement Parallel Tasks: Write a parallel C program where different threads execute different tasks.

Print Multiple Series Using Different Threads: Demonstrate how to print two series of numbers (2s and 4s) using separate threads.

## Theory

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It allows developers to write parallel code using compiler directives, library routines, and environment variables.

In OpenMP, parallel regions can be used to execute code concurrently across multiple threads. By dividing tasks among threads, different threads can execute different portions of the code simultaneously.

Key Concepts:

Parallel Regions: Sections of code that run in parallel.

Task Directive: Defines a block of code to be executed by a thread.

Conclusion

This program demonstrates how to use OpenMP to execute different tasks on separate threads. By using the parallel, single, and task directives, different threads can execute different portions of the code concurrently. This example shows how to print two different series of numbers (2s and 4s) using separate threads, illustrating the flexibility and power of OpenMP for parallel programming. Understanding how to manage and utilize threads effectively is crucial for optimizing the performance of parallel applications.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 9**

## **Experiment Title: MPI Program 1**

## 

## Problem Statement: Write MPI Program to print "Hello World".

## MPI program to send and receive Hello World messages from all other processes to a Root process and print the received messages.

### Objectives

1. **Understand MPI Basics**: Gain a basic understanding of MPI (Message Passing Interface) and its use in parallel programming.
2. **Learn MPI Communication**: Familiarize yourself with MPI functions for sending and receiving messages between processes.
3. **Implement a Simple MPI Program**: Write and execute a simple MPI program to print "Hello World" and send/receive messages between processes.

### Theory

MPI (Message Passing Interface) is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. MPI provides a way for processes to communicate with one another by sending and receiving messages.

#### Key MPI Functions:

1. **MPI\_Init**: Initializes the MPI environment.
2. **MPI\_Comm\_size**: Determines the size of the communicator (number of processes).
3. **MPI\_Comm\_rank**: Determines the rank of the calling process within the communicator.
4. **MPI\_Send**: Sends a message to a specified process.
5. **MPI\_Recv**: Receives a message from a specified process.
6. **MPI\_Finalize**: Terminates the MPI environment.

### Conclusion

This program demonstrates the basics of MPI, including initializing the environment, retrieving process information, and performing message passing between processes. By printing "Hello World" from each process and sending/receiving messages to/from the root process, you can see how MPI enables parallel programs to communicate effectively. Understanding MPI's fundamental functions is essential for developing more complex parallel applications that require efficient inter-process communication.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 10**

## **Experiment Title: MPI Program 2**

## 

## Problem Statement: MPI program to send two numbers (array elements) per process to a Root process and print the received messages.

### Objectives

1. **Understand Advanced MPI Communication**: Learn how to send and receive arrays of data between processes in MPI.
2. **Implement Collective Communication**: Write a parallel MPI program where processes send data to a root process, which then collects and prints the received data.
3. **Demonstrate Practical Use of MPI**: Implement a practical example of data distribution and collection using MPI.

### Theory

MPI (Message Passing Interface) is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. It enables communication between processes by sending and receiving messages, which can include arrays or other data structures.

#### Key MPI Functions for Array Communication:

1. **MPI\_Send**: Sends a message to a specified process.
2. **MPI\_Recv**: Receives a message from a specified process.
3. **MPI\_Gather**: Collects data from all processes and sends it to the root process.

### Conclusion

This program demonstrates the use of MPI to send and receive arrays of data between processes. Each process sends two numbers to the root process, which collects and prints the received data. This example illustrates the practical use of MPI for data distribution and collection, showcasing how MPI can be used to manage communication between multiple processes in parallel applications. Understanding these concepts is essential for developing efficient parallel programs that require inter-process communication.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 11**

## **Experiment Title: MPI Program 3**

## 

## Problem Statement: MPI program to find sum of first N integers using any given number of processes. Example, N=10,000 and no. of processes can be 4 or 8 or 12 etc.

### Objectives

1. **Understand Parallel Reduction**: Learn how to use MPI for parallel reduction operations, such as computing the sum of integers.
2. **Distribute Computation Across Processes**: Write an MPI program that divides the task of summing integers across multiple processes.
3. **Implement and Analyze Parallel Performance**: Implement the program and observe how the computation scales with different numbers of processes.

### Theory

MPI (Message Passing Interface) is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. It enables communication between processes by sending and receiving messages. Parallel reduction is a common pattern in parallel computing where a large dataset is divided among multiple processes, each of which computes a partial result. The partial results are then combined (reduced) to obtain the final result.

#### Key MPI Functions:

1. **MPI\_Init**: Initializes the MPI environment.
2. **MPI\_Comm\_size**: Determines the size of the communicator (number of processes).
3. **MPI\_Comm\_rank**: Determines the rank of the calling process within the communicator.
4. **MPI\_Reduce**: Combines values from all processes and returns the result to the root process.
5. **MPI\_Finalize**: Terminates the MPI environment.

### Conclusion

This program demonstrates the use of MPI for parallel reduction to compute the sum of the first N integers. By dividing the range of numbers among multiple processes and using MPI\_Reduce to combine the partial results, the program efficiently computes the sum in parallel. Understanding and implementing parallel reduction is essential for developing efficient parallel applications that require collective operations across multiple processes.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 12**

## **Experiment Title: MPI Program 4**

## 

## Problem Statement: MPI program to find sum of n integers in which processors are arranged in ring topology using MPI point-to-point blocking communication library calls.

Objectives

1. **Understand Ring Topology in MPI**: Learn how to simulate a ring communication pattern using MPI point-to-point communication.
2. **Implement Point-to-Point Communication**: Use MPI send and receive operations to pass data between processes arranged in a ring.
3. **Compute Sum Using Ring Communication**: Write an MPI program to find the sum of n integers where processes communicate in a ring topology.

### Theory

MPI (Message Passing Interface) supports various communication patterns, including point-to-point communication, which allows processes to send and receive messages directly to and from other processes. A ring topology in MPI arranges processes in a circular manner, where each process communicates with its neighbors.

#### Key MPI Functions for Point-to-Point Communication:

1. **MPI\_Init**: Initializes the MPI environment.
2. **MPI\_Comm\_size**: Determines the size of the communicator (number of processes).
3. **MPI\_Comm\_rank**: Determines the rank of the calling process within the communicator.
4. **MPI\_Send**: Sends a message to a specified process.
5. **MPI\_Recv**: Receives a message from a specified process.
6. **MPI\_Finalize**: Terminates the MPI environment.

### Conclusion

This MPI program demonstrates the use of point-to-point communication to compute the sum of n integers in a ring topology. Each process calculates its local sum and communicates with its neighbors to combine results until the total sum is computed by the root process. Understanding ring communication and point-to-point operations in MPI is essential for developing scalable and efficient parallel applications that utilize different communication patterns.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 13**

## **Experiment Title: CUDA Program 1**

## 

## Problem Statement: Write a CUDA program to perform two matrix additions.

Objectives

1. **Introduction to CUDA Programming**: Learn the basics of CUDA programming for parallel computing on NVIDIA GPUs.
2. **Matrix Addition Using CUDA**: Implement a CUDA program to perform matrix addition on GPU.
3. **Understand GPU Architecture and Memory Management**: Gain insights into CUDA threads, blocks, and memory management for efficient parallel execution.

### Theory

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows developers to use NVIDIA GPUs for general-purpose processing, significantly accelerating computational tasks compared to traditional CPU-based processing.

#### Key Concepts in CUDA Programming:

1. **Threads and Blocks**: CUDA organizes parallel execution using threads grouped into blocks. Each block executes independently, and threads within a block can cooperate using shared memory.
2. **Memory Hierarchy**: CUDA provides different types of memory:
   * **Global memory**: Accessed by all threads, but with higher latency.
   * **Shared memory**: Faster memory accessible by threads within the same block.
   * **Registers**: Private memory for each thread.
3. **Kernel Functions**: Functions executed on the GPU are called kernels. Kernels are written with a specific syntax (\_\_global\_\_) and are called from the host (CPU).

### Conclusion

This CUDA program demonstrates how to perform matrix addition using GPU parallelism. By utilizing CUDA's kernel function and memory management, matrix addition can be efficiently parallelized across GPU threads. Understanding CUDA programming concepts such as threads, blocks, memory management, and kernel functions is essential for leveraging GPU acceleration in compute-intensive applications. This example serves as a foundational starting point for developing more complex CUDA programs for various computational tasks.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions

**Experiment No 14**

## **Experiment Title: CUDA Program 2**

## 

## Problem Statement: Write a CUDA program to perform two matrix multiplications.

Objectives

1. **Introduction to CUDA Programming**: Learn the basics of CUDA programming for parallel computing on NVIDIA GPUs.
2. **Matrix Multiplication Using CUDA**: Implement a CUDA program to perform matrix multiplication on GPU.
3. **Understand GPU Architecture and Memory Management**: Gain insights into CUDA threads, blocks, and memory management for efficient parallel execution of matrix operations.

### Theory

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows developers to use NVIDIA GPUs for general-purpose processing, significantly accelerating computational tasks compared to traditional CPU-based processing.

#### Key Concepts in CUDA Programming:

1. **Threads and Blocks**: CUDA organizes parallel execution using threads grouped into blocks. Each block executes independently, and threads within a block can cooperate using shared memory.
2. **Memory Hierarchy**: CUDA provides different types of memory:
   * **Global memory**: Accessed by all threads, but with higher latency.
   * **Shared memory**: Faster memory accessible by threads within the same block.
   * **Registers**: Private memory for each thread.
3. **Kernel Functions**: Functions executed on the GPU are called kernels. Kernels are written with a specific syntax (\_\_global\_\_) and are called from the host (CPU).

### Conclusion

This CUDA program demonstrates how to perform matrix multiplication using GPU parallelism. By utilizing CUDA's kernel function and memory management, matrix multiplication can be efficiently parallelized across GPU threads. Understanding CUDA programming concepts such as threads, blocks, memory management, and kernel functions is essential for leveraging GPU acceleration in compute-intensive applications. This example serves as a foundational starting point for developing more complex CUDA programs for various computational tasks involving matrix operations.

## Source Code and Output /Screenshots:

## *(To be provided by the student)*

| *//Source code and output/screenshot should be available here* |
| --- |

## Useful Resource:

## Exercise Questions