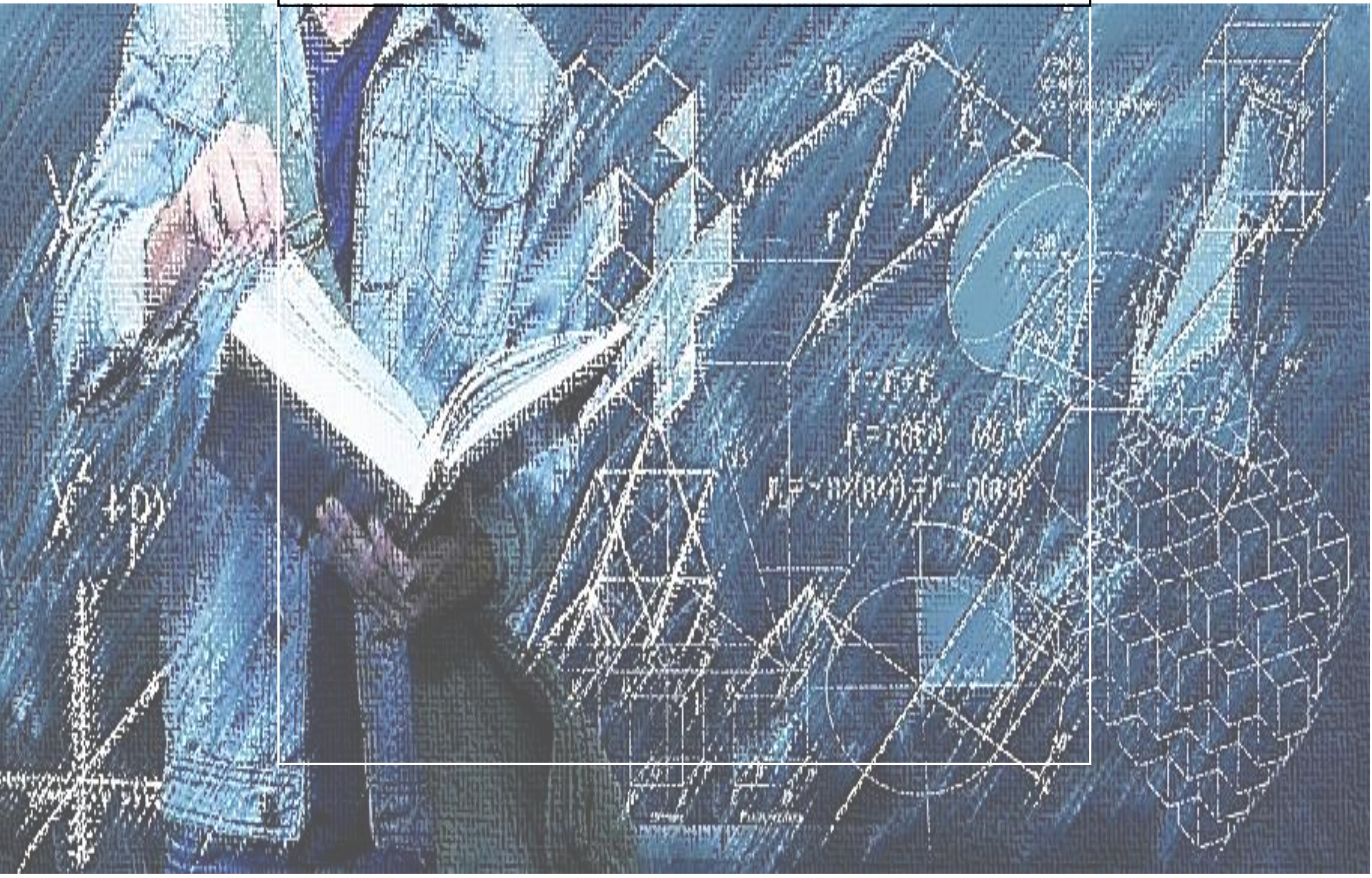
# CS417L Parallel Processing Lab

**Faculty of Computer Science and Engineering**

**Lab Manual**



# CS417L Parallel Processing Lab

## Lab Outline

**Prepared by: Mr. Usman Haider**
**Reviewed by: Dr. Taj Muhammad Khan, Ms. Nazia Shahzadi**

**Aprroved by: ______________________ Signature: ___________________**

FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)

GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNOLOGY (GIKI)

| CS417 L | Parallel Processing Parallel Processing and Distributed Computing Lab (1 CH) | Knowledge Profile: WK2 | Focus: CCP/CEP | AI/CS/DS |
|---|---|---|---|---|

**Pre-Requisite:** CS311, CS222
Instructor: **Mr. Usman Haider**
Office # F-11 FES, GIK Institute,
Email: usman.haider@giki.edu.pk
Office Hours: TBA

**Lab Introduction**

This course introduces High-performance architectures and programming languages; writing parallel programs in multiple paradigms/frameworks is covered including the use of synchronization primitives for mutual exclusion as well as other types of synchronizations. Students will be able to exploit parallel hardware to increase the performance of their programs. Shared memory systems are covered and GPU programming is also explored.

**Lab Contents**

Broadly, this lab will cover the following contents:

1. Introduction to Linux and C programming.
2. Introduction to Posix Threads pthreads -- Thread Basics, Thread Creation and Termination, Thread Synchronization.
3. Introduction to OpenMP – Basics of OpenMP API (Open Multi-Processor API), to get familiarized with OpenMP Directives
4. OpenMP --- Sharing of work among threads using Loop Construct in OpenMP, Clauses in Loop Construct
5. OpenMP --- Sharing of work among threads in an OpenMP program using ‘Sections Construct
6. Introduction to MPI --- Basics of MPI (Message Passing Interface)
7. MPI --- Communication between MPI processes.
8. MPI --- Point to-Point communication in MPI
9. MPI --- Collective communication in MPI
10. OpenCL --- OpenCL introduction and programs on vectors
11. OpenCL/ CUDA --- Programs on matrix
12. CUDA

**Mapping of CLOs and PLOs**

| Sr. No | Course Learning Outcomes+ | WA PLOs* | SW PLOs | Taxonomy level ( Psychomotor Domain) |
|---|---|---|---|---|
| CLO 1 | Implement parallel programs using synchronization primitives for shared memory CPUs. | PLO 3 | Design/ Development of Solutions | **P3** |
| CLO 2 | Implement parallel programs using synchronization primitives for GPUs. | PLO 3 | Design/ Development of Solutions | **P3** |
| CLO 3 | Implement parallel programs on distributed memory systems. | PLO 3 | Design/ Development of Solutions | **P3** |

| | | |
|---|---|---|
| | | +Please add the prefix "Upon successful completion of this course, the student will be able to" |

## CLO Assessment Mechanism

| Assessment tools | CLO_1 | CLO_2 | CLO_3 |
|---|---|---|---|
| **Lab Tasks** | **60%** | **80%** | **100%** |
| **Midterm Exam** | **20%** | **-** | **-** |
| **Open Ended Lab** | **-** | **-** | **-** |
| **Final Exam** | **20%** | **20%** | **-** |

## Overall Grading Policy

| Assessment Items | Percentage |
|---|---|
| **Lab Evaluation** | **40%** |
| **Midterm Exam** | **20%** |
| **Open Ended Lab** | **10%** |
| **Final Exam** | **30%** |

## Text and Reference Books

**Text books:**
- Lab Manual
- Online C, OpenMP, MPI, documentation
- An Introduction to Parallel Programming (2nd Edition) by Peter Pacheco, Matthew Malensek, March 15, 2020, ISBN: 978-0128046050

## Administrative Instruction

- According to institute policy, 100% attendance is *mandatory* to appear in the final examination.
- Assignments assigned must be submitted as per instructions mentioned in the assignments.
- For queries, kindly follow the office hours to avoid any inconvenience.

## Computer Usage/Software tools

- Students are encouraged to solve some assigned homework and lab tasks using the available programming software, such as DevC, Visual Studio community version and Visual Studio code

## Lecture Breakdown

| | |
|---|---|
| Lab 1 | • C Revisions: File IO, Dynamic Memory, etc. |
| Lab 2 | • Posix Threads Introduction |
| Lab 3 | • Posix Threads Synchronization |
| Lab 4 | • Posix Threads Synchronization (Mutual exclusion, cond vars, barriers) |
| Lab 5 | • OpenMP Introduction |
| Lab 6 | • OpenMP Synchronization |

| | |
|---|---|
| Lab 7 | • GPU Programming Introduction |
| Lab 8 | • GPU Programming Synchronization |
| Lab 9 | • GPU Programming Synchronization |
| Lab 10 | • MPI Introduction + Communication |
| Lab 11 | • MPI Communication (Point to point + Collective) |
| Lab 12 | • Open Ended Lab |
| | |

# Lab Evaluation Rubric

| LAB RUBRICS | | |
|---|---|---|
| Rubric 1 | Understanding of Parallel Processing Concepts | 8 points |
| Rubric 2 | Parallel Code Design | 6 points |
| Rubric 3 | Code Execution | 6 points |

## Category 1: Understanding of Parallel Processing Concepts (8 points)

**Conceptual Knowledge (5 points):**

0-2 points: Limited or no understanding.

3 points: Basic understanding.

4 points: Good understanding.

5 points: Strong understanding.

**Terminology (3 points):**

0-1 points: Incorrect or inconsistent terminology.

2 points: Some correct terminology with inconsistencies.

3 points: Consistent and correct use of terminology.

**Category 2: Implementation and Execution (12 points)**

## Parallel Code Design (6 points):

0-2 points: Poor design without parallelism.

3 points: Basic design with limited parallelism.

4 points: Good design with moderate parallelism.

5 points: Excellent design with extensive parallelism.

6 points: Outstanding design maximizing parallelism.

## Code Execution (6 points):

0-2 points: Execution issues or incorrect results.

3 points: Partially correct execution with major issues.

4 points: Mostly correct execution with minor issues.

5 points: Flawless execution with good performance.

6 points: Exceptional execution with optimal performance and scalability.

# Contents

# 1 Lab 1: Introduction to Linux and C programming

## Objectives

- Teach students how to use the terminal and execute basic commands.
- Introduce the Sublime Text editor on Ubuntu.
- Provide hands-on experience with C programming in the Ubuntu environment by solving complex problems.

## Deliverables

- Successfully navigates and performs basic tasks in the Linux environment.
- Create and compile C programs in Ubuntu.
- Complete and demonstrate two C programming examples, one involving arithmetic operations on a static array and another for matrix addition with input from a file.

## 1.1 Linux

Linux is an operating system, which is a flavor of UNIX. Linux is a multi-user and multi-tasking operating system. Because of the likeness with UNIX, all the programs written for UNIX can be compiled and run under Linux. Linux operating system runes on variety of machines like 486/Pentium, Sun SPARC, PowerPC, etc. One of the important features of Linux is communication through TCP/IP protocols. Linus Torvalds at the University of Helsinki, Finland, developed Linux. UNIX programmers around the world assisted him in the development of Linux.

## 1.2 Basics of Ubuntu

Ubuntu uses the Linux file system, which is based on a series of folders in the root directory. These folders contain important system files that cannot be modified unless you are running as the root user or use sudo. This restriction exists for both security and safety reasons; computer viruses will not be able to change the core system files, and ordinary users should not be able to accidentally damage anything vital. At the top of the hierarchy is the root directory which is denoted by . The root directory contains all other directories and files on your system. Below the root directory are the following essential directories:

- /bin and /sbin Many essential system applications (equivalent to C:/Windows).
- /etc System-wide configuration files.

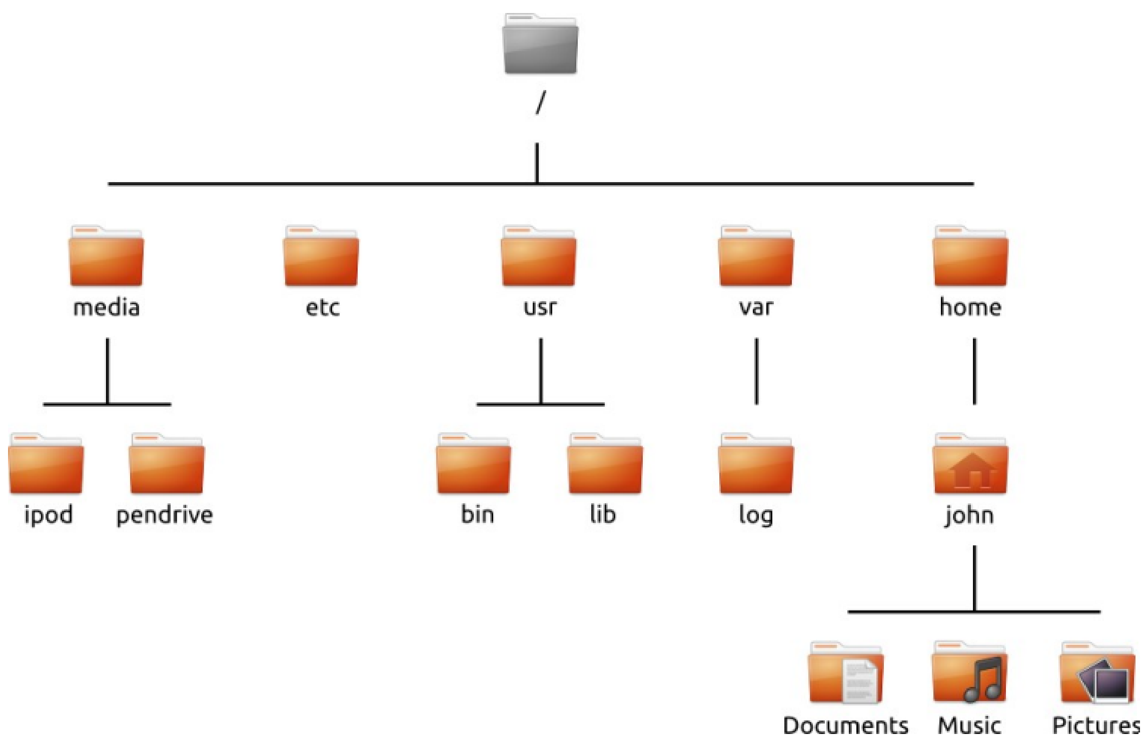
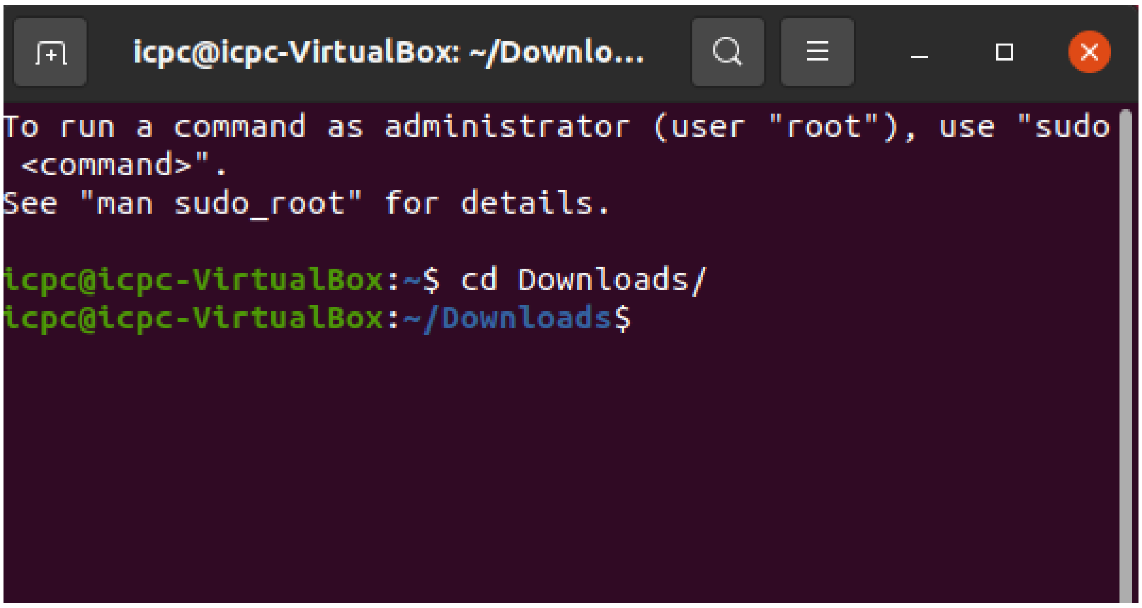


Figure 1: Ubuntu File System Structure

- /home Each user will have a subdirectory to store personal files (for example, /home/yourusername) which is equivalent to C:/Users or C:/Documents and Settings in Microsoft Windows.
- /lib Library files, similar to .dll files on Windows. Fig 1: Ubuntu File System Structure
- /media Removable media (cd-roms and usb drives) will be mounted in this directory.
- /root This contains the root user's files (not to be confused with the root directory).
- /usr Pronounced "user," it contains most program files (not to be confused with each user's home directory). This is equivalent to C:/Program Files in Microsoft Windows.
- /var/log Contains log files written by many applications.

## 1.3 Terminal

In order to fully realize the power of Ubuntu, you will need to learn how to use the terminal. Most operating systems, including Ubuntu, have two types of user interfaces. The first is a GUI. This is the desktop, windows, menus, and toolbars you click to get things done. The second, much older type of interface is the command-line interface (CLI). The terminal is Ubuntu's CLI. It is a method of controlling some aspects of Ubuntu using only commands that you type on the keyboard. The terminal gives you access to what is called a shell. When you type a command in the terminal, the shell interprets this command, resulting in the desired action. All commands in the terminal follow the same approach: Type a command, possibly followed by some parameters, and press Enter to perform the specified action. Parameters (also called switches) are extra segments of text, usually added at the end of a command, that change how the command itself is interpreted. These usually take the form of **-h** or **–help**, for example. In fact, **–help** can be added to most commands to display a short description of the command, as well as a list of any other parameters that can be used with that command. Often, some type of output will be displayed confirming the action was completed successfully, although this can depend on the command being executed. For example, using the **cd** command to change your current directory will change the prompt but will not display any output, as shown in the figure below.



```
To run a command as administrator (user "root"), use "sudo
 <command>".
See "man sudo_root" for details.

icpc@icpc-VirtualBox:~$ cd Downloads/
icpc@icpc-VirtualBox:~/Downloads$
```

Figure 2: Changing the current directory

## 1.4 Some basic commands

(a) **ls: list directory contents**: The ls command will show you the list of files in your current directory.

(b) **cd: Change Directory**: The cd command will allow you to change directories.

(c) **pwd : print the current/working directory**: The pwd command will allow you to know in which directory you are currently working.

(d) **sudo command**: run command as a super user (root)

(e) **ifconfig**: show network information

(f) **iwconfig**: show wireless information

## 1.5 Sublime Editor

Sublime Text is one of the most widely used text and source code editors for web and software development. It is very fast and it comes with lots of powerful features out of the box. You can enhance its functionality by installing new plugins and creating custom settings.

### 1.5.1 Installing Sublime Text on Ubuntu

**Before continuing with this tutorial, make sure you are logged in as a user with sudo privileges.**
To install Sublime Text 3 on your Ubuntu system, follow these steps:

1. Update the apt package list and install the dependencies necessary to fetch packages from https sources:

```
$ sudo apt update
$ sudo apt install apt-transport-https ca-certificates
curl software-properties-common
```

2. Import the repository's GPG key using the following curl command:

```
$ curl -fsSL https://download.sublimetext.com/
sublimehq-pub.gpg | sudo apt-key add
```

   Add the Sublime Text APT repository to your system's software repository list by typing:

```
$ sudo add-apt-repository "deb https://download.sublime
text.com/ apt/stable/"
```

3. Once the repository is enabled, update apt sources and install Sublime Text 3 with the following commands:

```
$ sudo apt update
$ sudo apt install sublime-text
```

That's it. Sublime Text has been installed on your Ubuntu desktop.

## 1.6 C program in Ubuntu

Before jumping right into the usage of C Programming Language, you are required to install it first on your Ubuntu. Here is the procedure for installing the C Programming Language on Ubuntu:

1. **Update system repositories:** Press “CTRL+ALT+T” to open the terminal of Ubuntu and run the below-given commands to update system repositories:

   ```
   $ sudo apt update
   ```

2. **Install build-essential package:** After updating the system repositories, execute the following command for the installation of the **“build-essential”** package:

   ```
   $ sudo apt install build-essential
   ```

   The error-free output indicates that the “build-essential” package successfully installed the collection of libraries and compiler for the C Programming Language.

3. **Check the C Compiler version:** To check the version of the installed C Compiler on your Ubuntu 22.04, utilize the **"gcc"** command with the **"–version"** option:

   ```
   $ gcc --version
   ```

After the successful installation of the C language compiler, open up your favorite text editor **(Sublime)** and write out the following simple C program in it:

```
#include <stdio.h>
int main() {
    int n1, n2, sum;
    printf("Enter two integers:");
    scanf("%d,%d",&n1,&n2);
    sum = n1+n2;
    printf("%d + %d = %d", n1, n2, sum);
    return 0;
}
```

In order to compile the above code, open the terminal and type the following commands:

```
$  gcc -o outfile infile.c
$  .\outfile
```

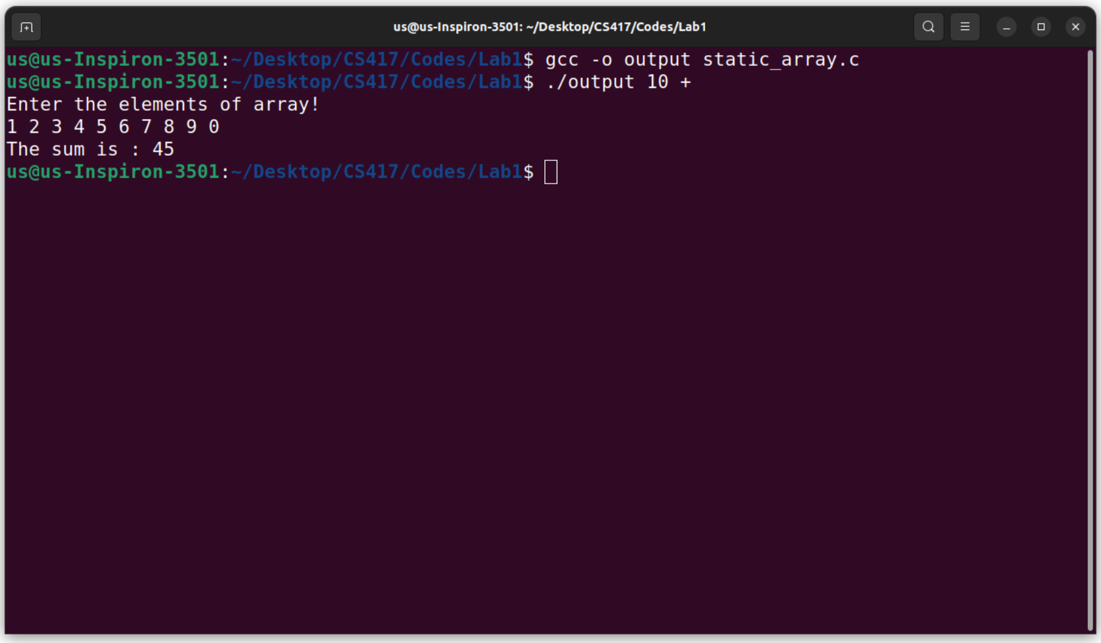
### 1.6.1 Example1 : Arithmetic operation on static array:

This C program takes command-line arguments to specify the size of an integer array and the mathematical operation to be performed on its elements. It first checks if it has received at least two command-line arguments: the program name and the operator. If not, it displays a usage message. It then converts the size argument to an integer and retrieves the operator character. The program initializes an integer array of the specified size and prompts the user to input its elements.

Depending on the operator character, it either calculates the sum or product of the array elements using a for loop. However, there is a bug in the code as it doesn't initialize the result variable before multiplication, and there's a missing semicolon in the multiplication branch of the printf statement. The program terminates by returning 0 if successful, or it displays an error message if the operator is not valid.

```
#include <stdio.h>

#include <stdlib.h>
int main (int argc, char *argv[])
{

        if (argc < 2)
        {
                printf("Missing arguments, Usage:  %s
                   ↪ size_of_array operator\n", argv[0] );
                return 0;
        }

        int size = atoi(argv[1]);
        char op = *argv[2];
        int arr[size];

        printf("Enter the elements of array!\n");
        for (int ii=0;ii<size;ii++)
        {
                scanf("%d", &arr[ii]);
        }

        int result;

        if (op == '+')
        {
                for (int ii=0;ii<size;ii++)
                {
                        result += arr[ii];
                }
                printf("The sum is : %d \n",result);
        }

        else if (op == '*')
        {
                for (int ii=0;ii<size;ii++)
                {
                        result *= arr[ii];
                }
                printf("The multiplication is : %d \n",result)
```

```
                    ↪ ;
        }
        else
        {
                printf("Operation not permitted\n");
        }
        return 0;
}
```



```
us@us-Inspiron-3501:~/Desktop/CS417/Codes/Lab1$ gcc -o output static_array.c
us@us-Inspiron-3501:~/Desktop/CS417/Codes/Lab1$ ./output 10 +
Enter the elements of array!
1 2 3 4 5 6 7 8 9 0
The sum is : 45
us@us-Inspiron-3501:~/Desktop/CS417/Codes/Lab1$ 
```

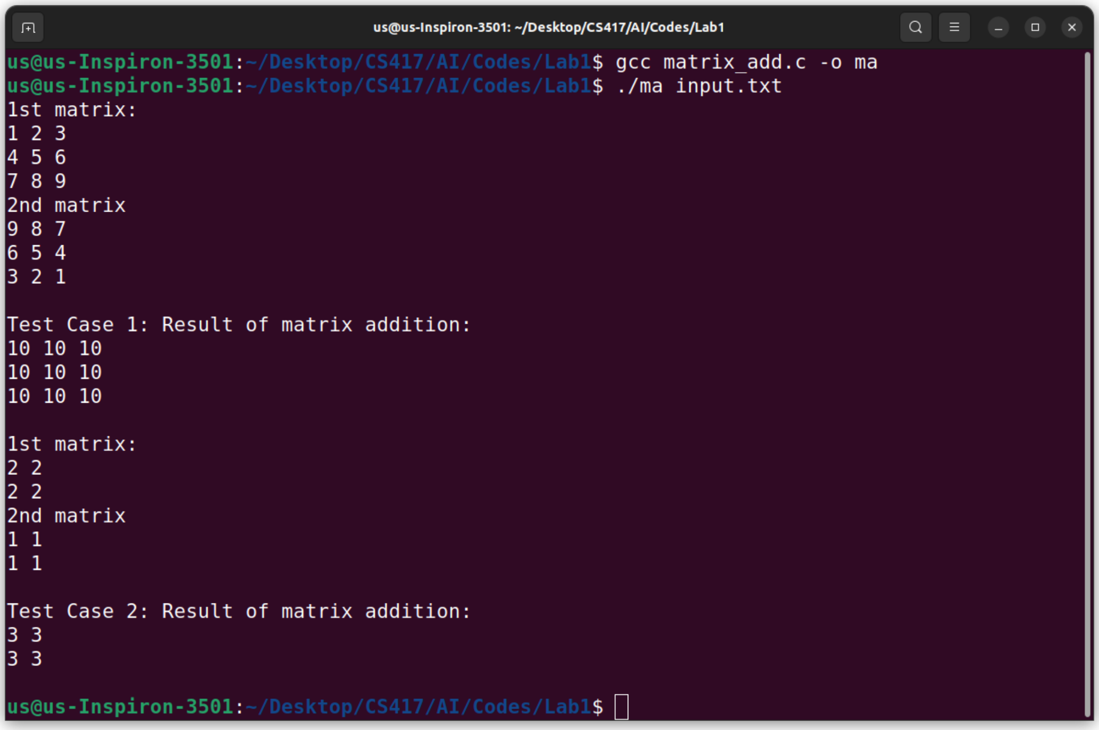
Figure 3: Output for plus operator

### 1.6.2 Example 2: Matrix addition! input from file

This C code reads matrices and performs matrix addition for multiple test cases specified in an input file. It uses functions like read_file to parse matrix data, calculate the sum of matrices, and print_result to display the results. The program iterates through the test cases, reads and adds matrices, and prints the results. It also handles file errors and memory allocation.

The input format for this program is structured to accommodate multiple test cases. Three distinct sections delineate each test case. Firstly, it begins with an integer denoting the number of test cases, labeled as 'numTestCases.' Following this, each test case consists of three consecutive lines. The first line of a test case specifies the dimensions of the first matrix, represented as two integers separated by a space. The first integer indicates the number of rows ('rows'), while the second integer signifies the number of columns ('cols') in the matrix. The second line contains the actual values of the first matrix, where spaces and organized row-wise separate integers. The third line mirrors the structure of the first two lines, providing the dimensions of the second matrix and, subsequently, the values of this matrix, also arranged row-wise.

```
#include <stdio.h>
#include <stdlib.h>
int read_file(FILE *file, int ***matrix, int *rows, int *cols)
  ↪  {

    fscanf(file, "%d %d", rows, cols);

    *matrix = (int **)malloc(*rows * sizeof(int *));
    for (int i = 0; i < *rows; i++) {
        (*matrix)[i] = (int *)malloc(*cols * sizeof(int));
    }

    for (int i = 0; i < *rows; i++) {
        for (int j = 0; j < *cols; j++) {
            fscanf(file, "%d", &(*matrix)[i][j]);
        }
    }
    return 1;
}
void free_space(int **matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}
void addition(int **matrix1, int **matrix2, int **result, int
  ↪ rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}
void print_result(int **matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }
    const char *filename = argv[1];
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    int numTestCases;
    fscanf(file, "%d", &numTestCases);
    for (int testCase = 1; testCase <= numTestCases; testCase
       ↪ ++) {
        int **matrix1, **matrix2, **result;
        int rows, cols;
        if (!read_file(file, &matrix1, &rows, &cols)) {
            fclose(file);
            return 1;
        }
        printf("1st matrix: \n");
        print_result(matrix1, rows, cols);

        if (!read_file(file, &matrix2, &rows, &cols)) {
            // free_space(matrix1, rows);
            fclose(file);
            return 1; // Error reading file
        }
        printf("2nd matrix \n");

        print_result(matrix2, rows, cols);

        result = (int **)malloc(rows * sizeof(int *));
        for (int i = 0; i < rows; i++) {
            result[i] = (int *)malloc(cols * sizeof(int));
        }
        addition(matrix1, matrix2, result, rows, cols);
        printf("\nTest Case %d: Result of matrix addition:\n",
           ↪  testCase);
        print_result(result, rows, cols);
        printf("\n");
        free_space(matrix1, rows);
        free_space(matrix2, rows);
        free_space(result, rows);
    }

    fclose(file);
    return 0;
}
```



```
us@us-Inspiron-3501:~/Desktop/CS417/AI/Codes/Lab1$ gcc matrix_add.c -o ma
us@us-Inspiron-3501:~/Desktop/CS417/AI/Codes/Lab1$ ./ma input.txt
1st matrix:
1 2 3
4 5 6
7 8 9
2nd matrix
9 8 7
6 5 4
3 2 1

Test Case 1: Result of matrix addition:
10 10 10
10 10 10
10 10 10

1st matrix:
2 2
2 2
2nd matrix
1 1
1 1

Test Case 2: Result of matrix addition:
3 3
3 3

us@us-Inspiron-3501:~/Desktop/CS417/AI/Codes/Lab1$
```

Figure 4: Matrix addition

# 2 Lab 2: Introduction to POSIX Threads (pthreads)

## Objectives

- Introduce POSIX threads (pthreads) and their API.
- Teach thread management, mutexes, condition variables, and synchronization.
- Provide hands-on experience with thread creation, termination, and joining.

## Deliverables

- Understanding of threads and POSIX threads.
- Proficiency in using pthreads for multithreaded programming.
- Successful implementation of multithreaded programs.
- Completion of provided exercises, including multithreaded array sum calculation and matrix summation.

## 2.1 What is the thread?

A thread is a semi-process, that has its own stack and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (whereas for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory and thus can access the same global variables, same heap memory, same set of file descriptors, etc.

## 2.2 POSIX threads (pthreads)

Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library, which you link with your program.
The primary motivation for using Pthreads is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication. Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU-intensive work.

- Priority/real-time scheduling: tasks, that are more important, can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling: tasks, which service events of indeterminate frequency and duration, can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

## 2.3 Pthread API

Pthreads API can be grouped into four:

### 2.3.1 Thread Management

Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes such as joinable, scheduling etc.

### 2.3.2 Mutexes

Routines that deal with synchronization, are called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

### 2.3.3 Condition variables

Routines that address communications between threads that share a mutex. Based upon programmer-specified conditions. This group includes functions to create, destroy, wait, and signal based on specified variable values. Functions to set/query condition variable attributes are also included.

### 2.3.4 Synchronization

Routines that manage read/write locks and barriers.

## 2.4 Thread creation

The main() program is a single, default thread. All other threads must be explicitly created by the programmer. The routine used to create a new thread is **pthread_create**. It creates the thread and makes it executable. The routine can be called any number of times from anywhere within the code. The prototype of the routine is given as:

```
pthread_create (pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg)
```

where the description of arguments is as follows:

- **thread:** An identifier for the new thread returned by the subroutine. This is a pointer to pthread_t structure. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread.
- **attr:** An attribute object that may be used to set thread attributes. We can specify a thread attributes object, or NULL for the default values.
- **start_routine:** The routine that the thread will execute once it is created.

  ```
      void *(*start_routine)(void *)
  ```

  We should pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. So, we can pass any type of single argument and return a pointer to any type.
- **arg:** A single argument that may be passed to start_routine. It must be passed as a void pointer. NULL may be used if no argument is to be passed.

Once created, threads are peers and may create other threads. There is no implied hierarchy or dependency between threads.

## 2.5 Terminating Threads

There are several ways in which a Pthread may be terminated:

- The thread returns from its starting routine (the main routine for the initial thread).
- The thread makes a call to the pthread_exit subroutine.
- The thread is canceled by another thread via the pthread_cancel routine
- The entire process is terminated due to a call to either the exec or exit subroutines.

**pthread_exit** is used to explicitly exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist. If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.
The example is given below:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *thread_func(void *p)
{
```

```
    pthread_t tid = pthread_self();
    printf("Child Thread ID is : %lu\n", tid);
    pthread_exit(NULL);
}

int main()
{

    pthread_t child;

    int t_status;

    printf("Parent ID is %d \n", getpid());

    t_status = pthread_create(&child, NULL, thread_func, NULL)
       ↪ ;
    if (t_status!=0)
    {
        printf("Thread creation failed.\n");
    }


    return 0;
}
```

## 2.6 Thread Join

```
int pthread_join (pthread_t th, void **thread_return)
```

The first parameter is the thread for which to wait, the identified that pthread_create filled in for us. The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure. When a thread is created, one of its attributes defines whether the thread is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
A thread can execute a thread join to wait until the other thread terminates. In our case, you - the main thread - should execute a thread join waiting for your colleague - a child thread - to terminate. In general, thread join is for a parent (P) to join with one of its child threads (C). Thread join has the following activities, assuming that a parent thread P wants to join with one of its child threads C:

- When P executes a thread join in order to join with C, which is still running, P is suspended until C terminates. Once C terminates, P resumes.
- When P executes a thread join and C has already terminated, P continues as if no such thread join has ever executed (i.e., join has no effect).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int global_data = 10;

void *thread_func(void *p)
{
        printf("Child here. Global data is %d.\n", global_data
           ↪ );
        global_data +=15;
        printf("Child after work. Global data is %d.\n",
           ↪ global_data);
}

int main()
{
        pthread_t child;

        int t_status;

        printf("Parent here. Global data is %d.\n",
           ↪ global_data);

        t_status = pthread_create(&child, NULL, thread_func,
           ↪ NULL);
        if (t_status!=0)
        {
                printf("Thread creation failed.\n");
        }

        pthread_join(child, NULL);

        printf("End of program. Global data is %d.\n",
           ↪ global_data);

        return 0;
}
```

## 2.7 Examples

### 2.7.1 Multithreaded vs. Sequential Array Sum Calculation

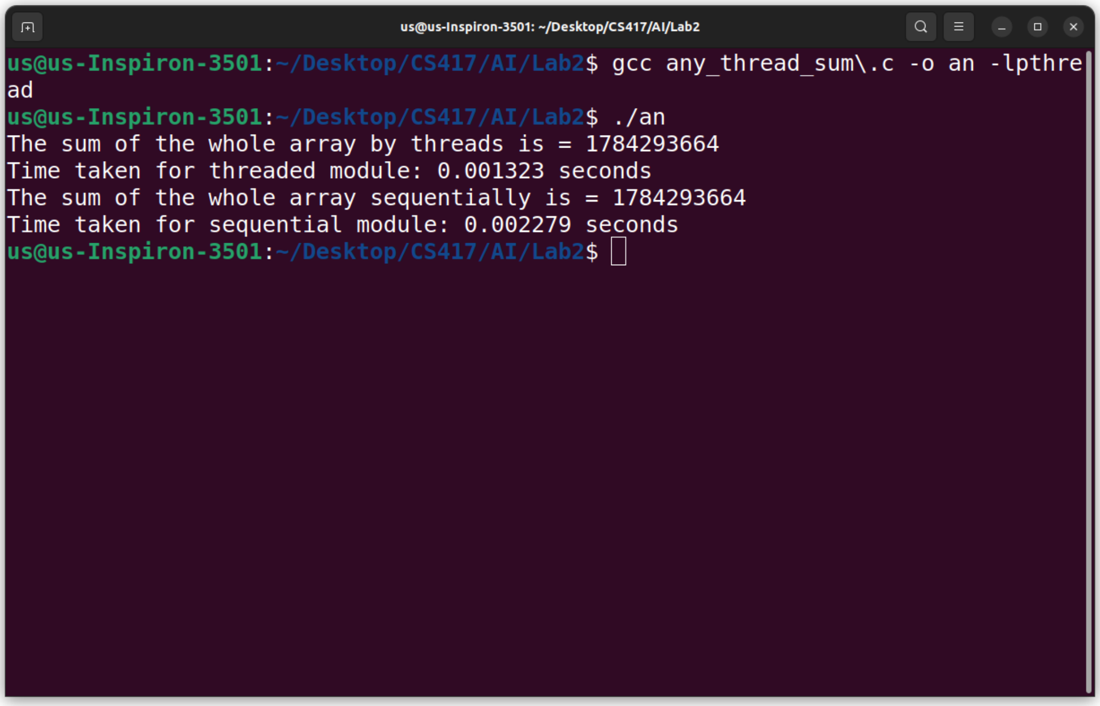
This code compares the performance of a multithreaded approach versus a sequential approach for calculating the sum of an integer array. It calculates the sum of elements in a large array. It creates multiple threads (the number is specified by num_threads) using the pthread library. Each thread is assigned a specific portion of the array to process, and their individual sums are later combined to obtain the total sum. The array partitioning ensures that each thread works on a non-overlapping segment. The program measures the time taken for both the threaded and sequential (non-parallel) approaches to compute the sum and prints the results. Finally, it cleans up memory allocation before exiting. This program illustrates the benefits of parallelism in handling computationally intensive tasks by dividing them among multiple threads, improving performance.

Furthermore, the data is partitioned as follows:

- In this program, the array partitioning is based on the total number of threads (num_threads) and each thread's number (thread_num).
- The program calculates the start and end indices for each thread's segment of the array using the following formula: start = (n * arr_size) / num_threads end = ((n + 1) * arr_size) / num_threads
  Here, n is the thread number, arr_size is the size of the array, and num_threads is the total number of threads.

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

typedef struct array_data
{
    int* arr;
    int thread_num;
    int arr_size;
    int num_threads;
} array_data;

int addition(int *arr, int arr_size)
{
    int sum = 0;
    for (int i = 0; i < arr_size; ++i)
    {
        sum += arr[i];
    }
    return sum;
}
```

```
void *array_sum(void *p)
{
    array_data* temp = (array_data*)p;
    int n = temp->thread_num;
    int* result = (int*)malloc(sizeof(int));
    result[0] = 0;

    int start = (n * temp->arr_size) / temp->num_threads;
    int end = ((n + 1) * temp->arr_size) / temp->num_threads;

    for (int i = start; i < end; ++i)
    {
        result[0] += temp->arr[i];
    }

    pthread_exit(result);
}

int main()
{
    int arr_size = 1000000;
    int num_threads = 4; // Change this to the desired number
       ↪ of threads

    int* array = (int*)malloc(sizeof(int) * arr_size);

    for (int i = 0; i < arr_size; ++i)
    {
        array[i] = i + 1;
    }

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    array_data* thread_data = (array_data*)malloc(sizeof(
       ↪ array_data) * num_threads);
    pthread_t* tid = (pthread_t*)malloc(sizeof(pthread_t) *
       ↪ num_threads);

    for (int i = 0; i < num_threads; ++i)
    {
        thread_data[i].thread_num = i;
        thread_data[i].arr = array;
        thread_data[i].arr_size = arr_size;
        thread_data[i].num_threads = num_threads;
        pthread_create(&tid[i], NULL, array_sum, &thread_data[
           ↪ i]);
    }

    int total_sum = 0;

    for (int i = 0; i < num_threads; ++i)
    {
        int* sum;
        pthread_join(tid[i], (void**)&sum);
        total_sum += *sum;
        free(sum);
    }

    printf("The sum of the whole array by threads is = %d\n",
       ↪ total_sum);

    clock_gettime(CLOCK_MONOTONIC, &end);
    double elapsed_time_threaded = (end.tv_sec - start.tv_sec)
       ↪  + (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("Time taken for threaded module: %lf seconds\n",
       ↪ elapsed_time_threaded);

    clock_gettime(CLOCK_MONOTONIC, &start);

    int sum_seq = addition(array, arr_size);

    printf("The sum of the whole array sequentially is = %d\n"
       ↪ , sum_seq);

    clock_gettime(CLOCK_MONOTONIC, &end);
    double elapsed_time_sequential = (end.tv_sec - start.
       ↪ tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("Time taken for sequential module: %lf seconds\n",
       ↪ elapsed_time_sequential);

    free(array);
    free(thread_data);
    free(tid);

    return 0;
}
```



```
us@us-Inspiron-3501:~/Desktop/CS417/AI/Lab2$ gcc any_thread_sum\.c -o an -lpthre
ad
us@us-Inspiron-3501:~/Desktop/CS417/AI/Lab2$ ./an
The sum of the whole array by threads is = 1784293664
Time taken for threaded module: 0.001323 seconds
The sum of the whole array sequentially is = 1784293664
Time taken for sequential module: 0.002279 seconds
us@us-Inspiron-3501:~/Desktop/CS417/AI/Lab2$ 
```

Figure 5: Multithreaded vs. Sequential Array Sum Calculation

### 2.7.2 Caculate the matrix sum

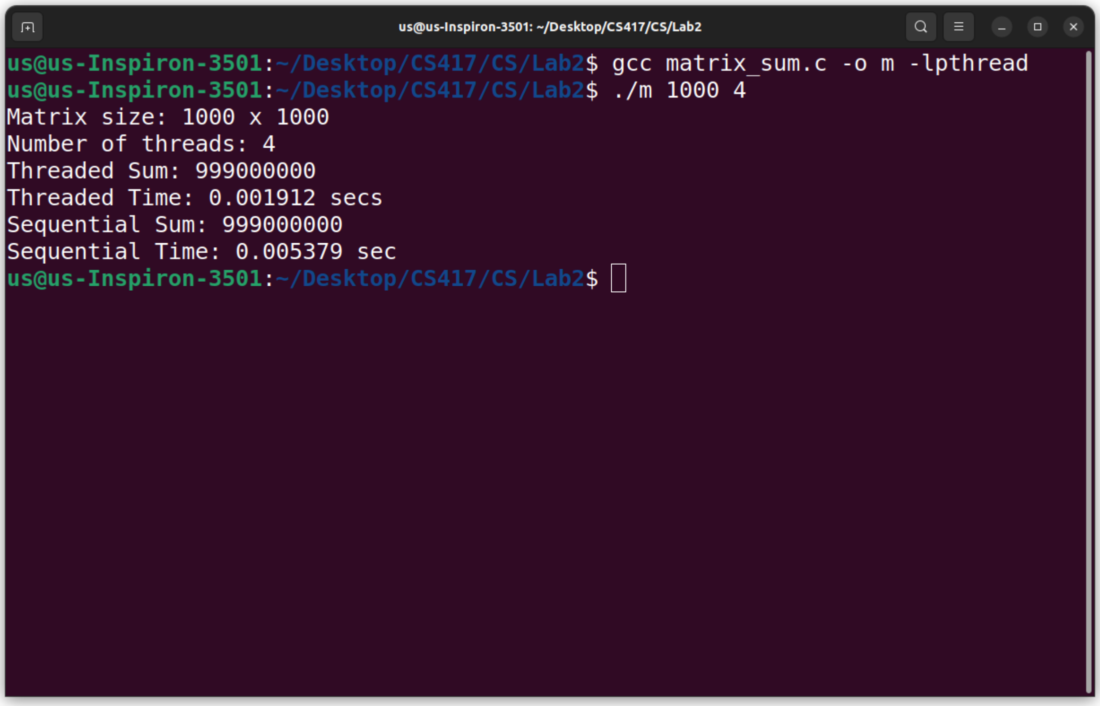
This C program utilizes pthreads to parallelize the computation of the sum of elements within a two-dimensional matrix and compares its performance against a sequential approach. In this program, global variables include the matrix itself ('mat'), its size ('mat_size'), the number of threads ('NUM_THREADS'), and an array to store thread-specific sums ('thread_sums').

The 'computeSum' function is the workhorse of the program, responsible for calculating the sum of a designated portion of the matrix by each thread. Thread IDs divide the matrix rows evenly among threads, with each thread iteratively summing its assigned rows and columns. The results are stored in the 'thread_sums' array, indexed by thread ID, before the threads exit.

The 'main' function initializes the program, accepting command-line arguments for matrix size and the number of threads. It allocates memory for the matrix and initializes its elements. An array of thread IDs and pthreads is declared, and the program records the start time. Threads are created and executed, each assigned a unique ID and tasked with running the 'computeSum' function with its respective ID. After all threads are created, the main thread waits for their completion before calculating the final threaded sum. The program then prints the threaded sum, and execution time, computes the sequential sum, and prints its results. Finally, the program deallocates memory and exits, providing a comparison of the execution times for the threaded and sequential approaches.

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

```
#include <pthread.h>
#include <time.h>

int **mat;
int mat_size;
int NUM_THREADS;
int *thread_sums;

void* computeSum(void* arg) {
    int thread_id = *((int*)arg);
    int

    int local_sum = 0;

    for (int i = start_row; i < end_row; i++) {
        for (int j = 0; j < mat_size; j++) {
            local_sum += mat[i][j];
        }
    }

    thread_sums[thread_id] = local_sum;
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <mat_size> <num_threads>\n"
           ↪ , argv[0]);
        return 1;
    }

    mat_size = atoi(argv[1]);
    NUM_THREADS = atoi(argv[2]);

    // Allocate memory for the mat
    mat = (int **)malloc(mat_size * sizeof(int *));
    for (int i = 0; i < mat_size; i++) {
        mat[i] = (int *)malloc(mat_size * sizeof(int));
    }

    thread_sums = (int *)malloc(NUM_THREADS * sizeof(int));

    for (int i = 0; i < mat_size; i++) {
        for (int j = 0; j < mat_size; j++) {
            mat[i][j] = i + j;
        }
    }

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, computeSum, &
           ↪ thread_ids[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // Calculate the final threaded sum
    int total_threaded_sum = 0;
    for (int i = 0; i < NUM_THREADS; i++) {
        total_threaded_sum += thread_sums[i];
    }

    printf("Threaded Sum: %d\n", total_threaded_sum);

    clock_gettime(CLOCK_MONOTONIC, &end);
    double elapsed_time_threaded = (end.tv_sec - start.tv_sec)
       ↪  + (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("Threaded Time: %lf secs\n", elapsed_time_threaded)
       ↪ ;



    clock_gettime(CLOCK_MONOTONIC, &start);
    int sequential_sum=0;
    for (int i = 0; i < mat_size; i++) {
        for (int j = 0; j < mat_size; j++) {
            sequential_sum += mat[i][j];
        }
    }

    printf("Sequential Sum: %d\n", sequential_sum);
    clock_gettime(CLOCK_MONOTONIC, &end);
    double elapsed_time_sequential = (end.tv_sec - start.
        ↪ tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("Sequential Time: %lf sec\n",
        ↪ elapsed_time_sequential);
    return 0;
}
```



```
us@us-Inspiron-3501:~/Desktop/CS417/CS/Lab2$ gcc matrix_sum.c -o m -lpthread
us@us-Inspiron-3501:~/Desktop/CS417/CS/Lab2$ ./m 1000 4
Matrix size: 1000 x 1000
Number of threads: 4
Threaded Sum: 999000000
Threaded Time: 0.001912 secs
Sequential Sum: 999000000
Sequential Time: 0.005379 sec
us@us-Inspiron-3501:~/Desktop/CS417/CS/Lab2$ 
```

Figure 6: Multithreaded vs. Sequential Matrix Element Sum Calculation

# 3 Lab 3: Thread Synchronization and Mutexes

## Objectives

- Explain the concept of mutual exclusion using mutexes.
- Demonstrate the characteristics and uses of mutexes.
- Introduce the concept of barriers in multithreading.

## Deliverables

- Proficiency in using mutexes for thread synchronization.
- Successful implementation of parallel algorithms using mutexes.
- Completion of exercises, including parallel counter implementation, counting unique elements in an array, and using barriers in multithreading.

## 3.1 Introduction

In this lab, we will study thread synchronization and mutexes. Synchronization is the cooperative action of two or more threads that ensure that each thread reaches a known point of operation concerning other threads before continuing. Thread synchronization is the concurrent execution of two or more threads that share critical resources. Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable simultaneously. A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time. Thus, a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access to the resource. Upon releasing the resource, the thread unlocks the mutex.

## 3.2 Mutexes (Mutual Exclusion)

A mutex, short for "mutual exclusion," is a synchronization primitive used in multi-threaded programming to control access to shared resources. Mutexes serve as a mechanism to ensure that only one thread can access a particular resource at any given time, thereby preventing race conditions and maintaining data consistency. Mutexes are widely used to coordinate and synchronize concurrent threads to achieve thread safety. The pseudo-code is given as:

```
Mutex Initialization:
  Initialize a mutex variable, mutex.
```

```
Thread A:
  Lock the mutex (pthread_mutex_lock).
  // Critical section - Access shared resource
  Unlock the mutex (pthread_mutex_unlock).

Thread B:
  Lock the mutex (pthread_mutex_lock).
  // Critical section - Access shared resource
  Unlock the mutex (pthread_mutex_unlock).

Thread C:
  Lock the mutex (pthread_mutex_lock).
  // Critical section - Access shared resource
  Unlock the mutex (pthread_mutex_unlock).
```

This pseudo-code demonstrates how to initialize a mutex and how multiple threads (Thread A, Thread B, Thread C) can use the mutex to safely access and modify a shared resource within their respective critical sections. The mutex ensures that only one thread can access the critical section simultaneously, preventing data corruption and race conditions.

### 3.2.1 Mutex Characteristics

**Exclusive Locking:** Mutexes provide exclusive locking. When a thread acquires a mutex lock, it gains exclusive access to the protected resource or critical section, preventing other threads from accessing it until the lock is released.
**Blocking and Waiting:** If a thread attempts to acquire a mutex already locked by another thread, it will be blocked (put to sleep) until it becomes available. This blocking behavior allows for controlled access to shared resources.

## 3.3 Uses of Mutexes

**Protection of Shared Resources:** The primary use of mutexes is to protect shared resources, such as variables, data structures, or critical sections of code, from simultaneous access by multiple threads. Mutexes ensure that only one thread can access the protected resource at any given time, preventing data corruption and race conditions.
**Thread Safety:** Mutexes are essential for achieving thread safety in multithreaded programs. By using mutexes to guard access to shared data, you can ensure that multiple threads can work together cooperatively without interfering with each other's operations.
**Orderly Access to Resources:** Mutexes help establish a specific order of access to shared resources. This is crucial in scenarios where the order of operations matters or where you want to ensure that resources are accessed in a predictable sequence. **Resource Management:** Mutexes are used to manage access to finite

resources, such as connections, file handles, and hardware devices. By using mutexes, you can prevent resource contention and efficiently allocate resources among multiple threads. **Preventing Deadlocks:** Mutexes, when used correctly, can help prevent deadlocks in multithreaded programs by imposing a disciplined order of acquiring locks and releasing them. This is critical for maintaining program stability.

## 3.4 Barriers

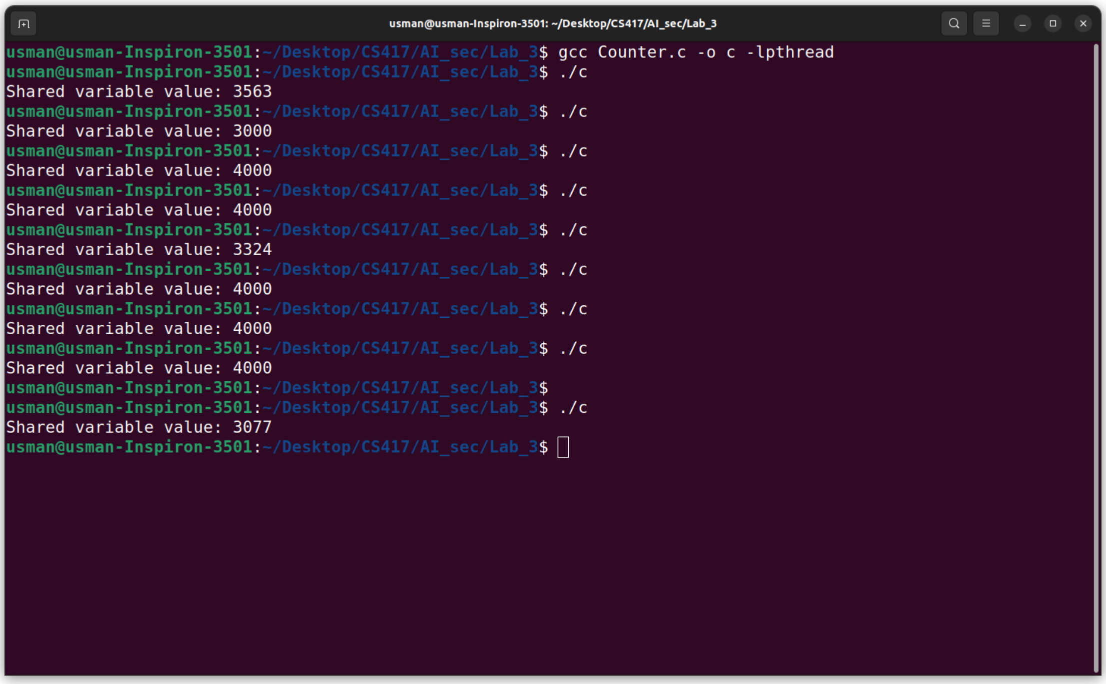
A barrier in the C language, specifically in the context of multithreaded programming with pthreads (POSIX Threads), serves as a synchronization mechanism that allows multiple threads to coordinate their execution. Threads often perform different tasks, and there are situations where you want them to pause and synchronize at a specific point in their execution before proceeding further. The barrier ensures that no thread can proceed beyond the barrier until all participating threads have reached it. To use a barrier, you first initialize it with the desired number of threads that need to synchronize at that point. Then, each thread, when it reaches the synchronization point, calls **pthread_barrier_wait()**. This function blocks the thread until all threads have reached the barrier. Once all threads are present, the barrier is released, and all threads can continue their execution past the barrier point.

For example, if you have four threads in a program and you want them to coordinate their actions at a specific point, you initialize a barrier with a count of 4. Each thread works independently until it reaches the barrier, where it waits until all four threads are present. This synchronization ensures that the threads proceed together beyond the barrier, making it a powerful tool for managing concurrent operations in multithreaded programs and avoiding race conditions or data inconsistencies that might occur without proper synchronization.

## 3.5 Example 1: Parallel implementation of counter

In this code, multiple threads are incrementing a shared counter without any synchronization mechanism. This can lead to race conditions, and the final value of the shared counter may not be the sum of all increments.

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

#define NUM_THREADS 4
#define ITER 1000

int shared_var = 0;

void *incrementer(void*)
{
        for (int i = 0; i < ITER; ++i)
        {
                shared_var++;
        }
        pthread_exit(NULL);
}

int main(){

        pthread_t threads[NUM_THREADS];

        for (int i = 0; i < NUM_THREADS; ++i)
        {
                pthread_create(&threads[i], NULL, incrementer,
                   ↪ NULL);
        }
        for (int i = 0; i < NUM_THREADS; ++i)
        {
                pthread_join(threads[i],NULL);
        }

        printf("Shared variable value: %d\n", shared_var );

        return 0;
}
```

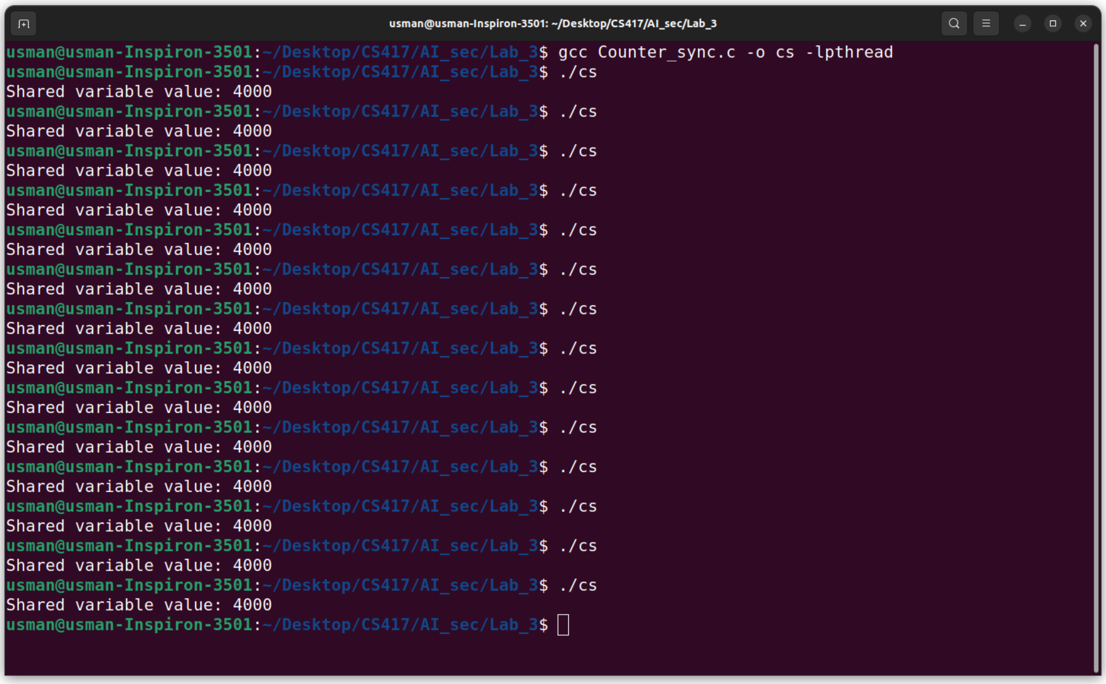


```
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ gcc Counter.c -o c -lpthread
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 3563
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 3000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 3324
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./c
Shared variable value: 3077
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$
```

Figure 7: Counter without synchronization machenism

In the above code, multiple threads concurrently access and increment the shared variable. Since any synchronization mechanism does not protect these operations, issues like race conditions and deadlocks can arise. The solution to these problems is to introduce proper synchronization to ensure that only one thread can increment shared variable at any given time. The updated code is given as follows:

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

#define NUM_THREADS 4
#define ITER 1000

int shared_var = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


void *incrementer(void*)
{

        for (int i = 0; i < ITER; ++i)
        {
                pthread_mutex_lock(&mutex);
        shared_var++;
        pthread_mutex_unlock(&mutex);
        }
        pthread_exit(NULL);
}

int main(){

        pthread_t threads[NUM_THREADS];


        for (int i = 0; i < NUM_THREADS; ++i)
        {
                pthread_create(&threads[i], NULL, incrementer,
                   ↪ NULL);
        }
        for (int i = 0; i < NUM_THREADS; ++i)
        {
                pthread_join(threads[i],NULL);
        }

        printf("Shared variable value: %d\n", shared_var );

        return 0;
}
```



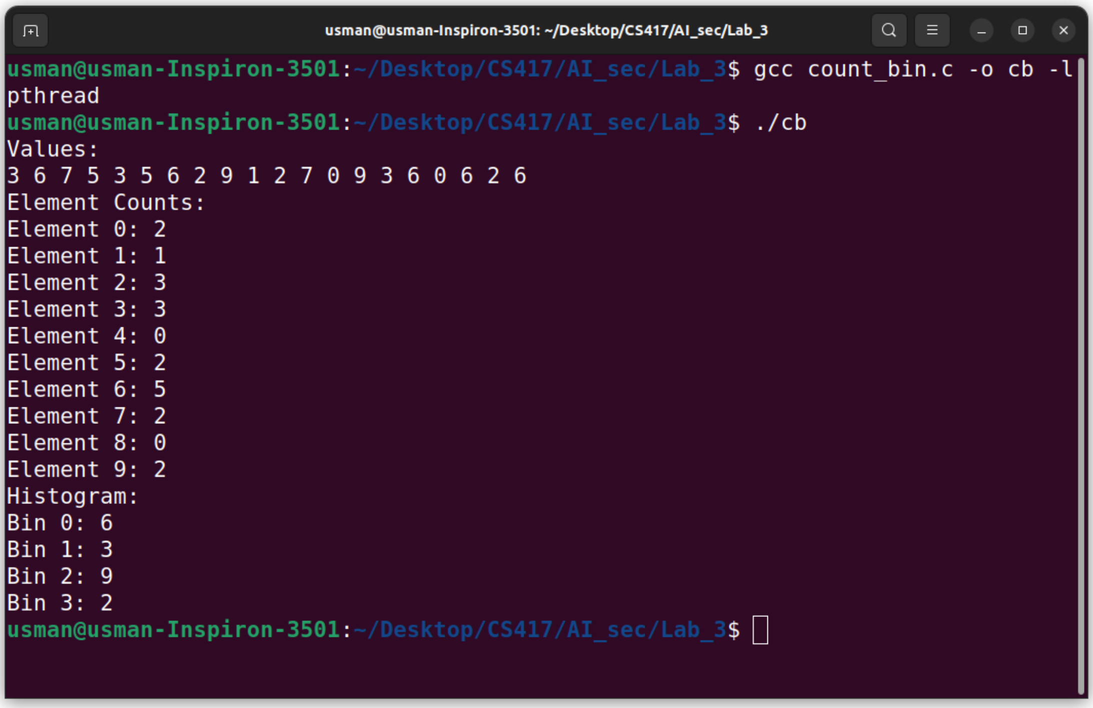
```
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ gcc Counter_sync.c -o cs -lpthread
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cs
Shared variable value: 4000
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$
```

Figure 8: Counter with synchronization mechanism

## 3.6 Example 2: Count the unique elements and generate a histogram in the array

The problem involves efficiently counting the occurrence of unique elements within an integer array using multiple threads while ensuring thread safety. Given an array of integers, the objective is to employ parallel processing, with each thread responsible for a distinct portion of the array. Mutexes are utilized to prevent race conditions, guaranteeing that the counts of individual elements are accurately updated. The result is a count of how many times each unique element appears in the array, providing valuable insights into the data distribution. This problem exemplifies concurrent programming and synchronization techniques, allowing for faster processing of large datasets in a multi-threaded environment.
This code further introduces the concept of a histogram. A histogram is a graphical representation of data distribution, where data values are grouped into intervals or "bins," and the frequency of values falling into each bin is counted. In this code, the histogram is divided into four bins. Instead of merely counting individual elements, the program assigns each element to one of these bins based on its value. This creates a more concise summary of the data distribution and allows for easier visualization.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 4
#define arr_len 20
#define val_limit 10
#define bins 4

int array[arr_len];
int element_counts[val_limit] = {0};
int histogram[bins] = {0};
pthread_mutex_t mutex[val_limit];

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    int chunk_size = arr_len / NUM_THREADS;
    int start = thread_id * chunk_size;
    int end = (thread_id == NUM_THREADS - 1) ? arr_len : start
       ↪  + chunk_size;

    for (int i = start; i < end; i++) {
        int element = array[i];
        pthread_mutex_lock(&mutex[element]);
        element_counts[element]++;
        pthread_mutex_unlock(&mutex[element]);
        // Update the histogram based on element value
        int bin = element * bins / val_limit;
        pthread_mutex_lock(&mutex[bin]);
        histogram[bin]++;
        pthread_mutex_unlock(&mutex[bin]);
    }
    pthread_exit(NULL);
}

int main() {

    for (int i = 0; i < arr_len; i++) {
        array[i] = rand() % val_limit;
    }
    printf("Values: \n");
    for (int i = 0; i < arr_len; ++i)
    {
        printf("%d ", array[i] );
    }
    printf("\n");

    for (int i = 0; i < val_limit; i++) {
        pthread_mutex_init(&mutex[i], NULL);
    }

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &
           ↪ thread_ids[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Element Counts:\n");
    for (int i = 0; i < val_limit; i++) {
        printf("Element %d: %d\n", i, element_counts[i]);
    }

    printf("Histogram:\n");
    for (int i = 0; i < bins; i++) {
        printf("Bin %d: %d\n", i, histogram[i]);
    }

    for (int i = 0; i < val_limit; i++) {
        pthread_mutex_destroy(&mutex[i]);
    }

    return 0;
}
```



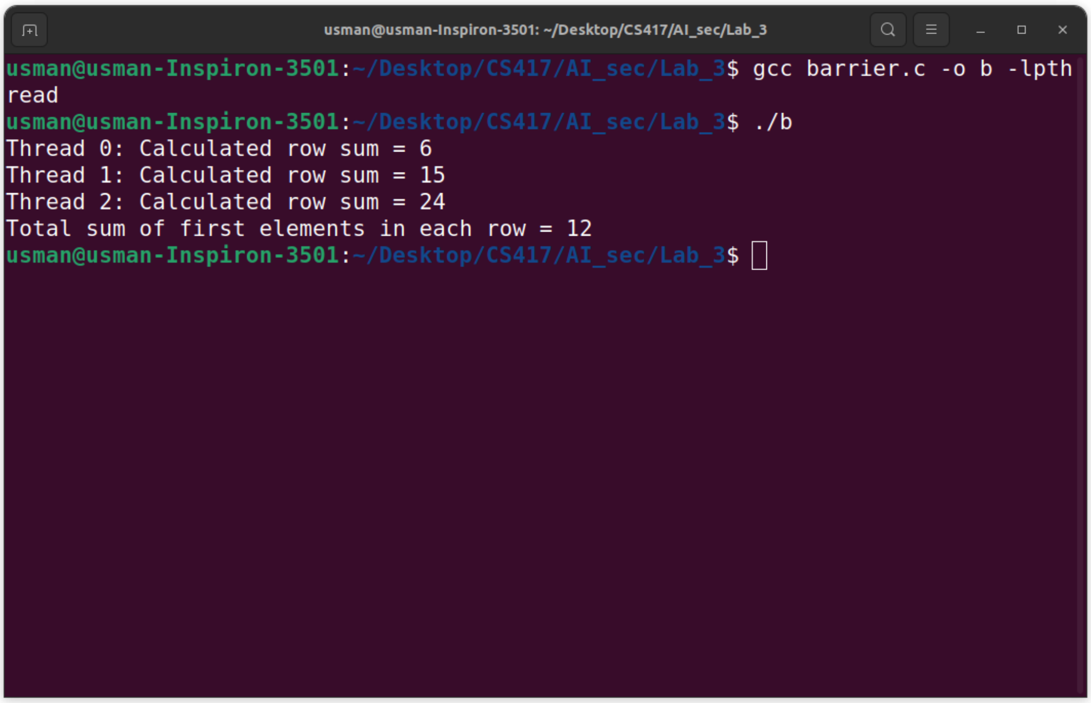
```
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ gcc count_bin.c -o cb -l
pthread
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./cb
Values:
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6
Element Counts:
Element 0: 2
Element 1: 1
Element 2: 3
Element 3: 3
Element 4: 0
Element 5: 2
Element 6: 5
Element 7: 2
Element 8: 0
Element 9: 2
Histogram:
Bin 0: 6
Bin 1: 3
Bin 2: 9
Bin 3: 2
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ 
```

Figure 9: Count of elements in any array

## 3.7 Example 3: Barrier example

This C code demonstrates the use of pthread barriers for synchronization in a multi-threaded program that performs calculations on a 3x3 matrix. The code spawns multiple threads, each responsible for calculating the sum of elements in its assigned row of the matrix. After completing their respective calculations, the threads synchronize using a pthread barrier to ensure that all calculations are finished.
Finally, one of the threads calculates and displays the total sum of the first elements in each row of the matrix. This example illustrates the effective use of pthread barriers to coordinate the execution of threads in a parallel computing environment, allowing for efficient and synchronized computation of complex tasks involving shared data structures.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 3
#define MATRIX_SIZE 3

int matrix[MATRIX_SIZE][MATRIX_SIZE] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

pthread_barrier_t barrier;

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    int row_sum = 0;

    for (int i = 0; i < MATRIX_SIZE; i++) {
        row_sum += matrix[thread_id][i];
    }

    printf("Thread %d: Calculated row sum = %d\n", thread_id,
       ↪ row_sum);

    pthread_barrier_wait(&barrier);

    if (thread_id == 0) {
        int total_sum = 0;
        for (int i = 0; i < NUM_THREADS; i++) {
            total_sum += matrix[i][0];
        }
        printf("Total sum of first elements in each row = %d\n
           ↪ ", total_sum);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    pthread_barrier_init(&barrier, NULL, NUM_THREADS);
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &
           ↪ thread_ids[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }


    pthread_barrier_destroy(&barrier);

    return 0;
}
```



usman@usman-Inspiron-3501: ~/Desktop/CS417/AI_sec/Lab_3

```
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ gcc barrier.c -o b -lpth
read
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ ./b
Thread 0: Calculated row sum = 6
Thread 1: Calculated row sum = 15
Thread 2: Calculated row sum = 24
Total sum of first elements in each row = 12
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_3$ 
```

Figure 10: Barrier example

# 4 Lab 4: Working with Conditional Variables in Multithreaded C Programming

## Objectives

- Introduce conditional variables and their role in multithreaded programs.
- Explain how to initialize, wait, and signal conditional variables.

## Deliverables

- Understanding of conditional variables and their usage.
- Proficiency in implementing multithreaded programs with conditional variables.
- Successful completion of exercises, including scenarios involving conditional variables, such as matrix sum and producer-consumer simulation.

## 4.1 Introduction

In multithreaded programs, threads often need to coordinate their actions to prevent race conditions and ensure shared resources are accessed safely. Synchronization mechanisms control the order of execution of threads and manage access to shared data. Mutexes and conditional variables are two key synchronization primitives used in multithreaded C programming.

Mutexes, short for "mutual exclusion," are used to protect critical code sections. They ensure that only one thread can access a particular resource. Threads attempting to access the resource protected by a mutex will be blocked until they acquire it.

```
ead_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
ead_mutex_lock(&mutex);
ritical section - protected resource accessed here
ead_mutex_unlock(&mutex);
```

## 4.2 Conditional Variables

Conditional variables are synchronization primitives that allow threads to coordinate and communicate regarding the state of shared resources. They are often used in conjunction with mutexes. Conditional variables allow threads to wait for a specific condition to become true before proceeding.

```
ead_cond_t cond = PTHREAD_COND_INITIALIZER;
ead_cond_wait(&cond, &mutex); // Wait on the condition
   ↪ variable
```

```
ead_cond_signal(&cond); // Signal the condition variable
```

## 4.3 Working of Conditional Variable

### 4.3.1 Initialization

Conditional variables must be initialized using pthread_cond_init before use. This function allocates and initializes the necessary resources.

### 4.3.2 Waiting

Threads that want to wait for a condition to become true use pthread_cond_wait. When a thread calls this function, it releases the associated mutex and enters a blocked state, waiting for another thread to signal the condition variable. This allows the waiting thread to efficiently wait without using CPU resources.

### 4.3.3 Signaling

Threads that modify shared resources and want to notify waiting threads when the condition has been met use pthread_cond_signal or pthread_cond_broadcast. pthread_cond_signal wakes up one waiting thread, allowing it to check the condition. pthread_cond_broadcast wakes up all waiting threads.

## 4.4 Example Scenario:

A classic example of using conditional variables is in the producer-consumer problem. In this scenario, one or more producer threads produce data, and consumer threads consume the data. Conditional variables are used to signal when data is ready for consumption.

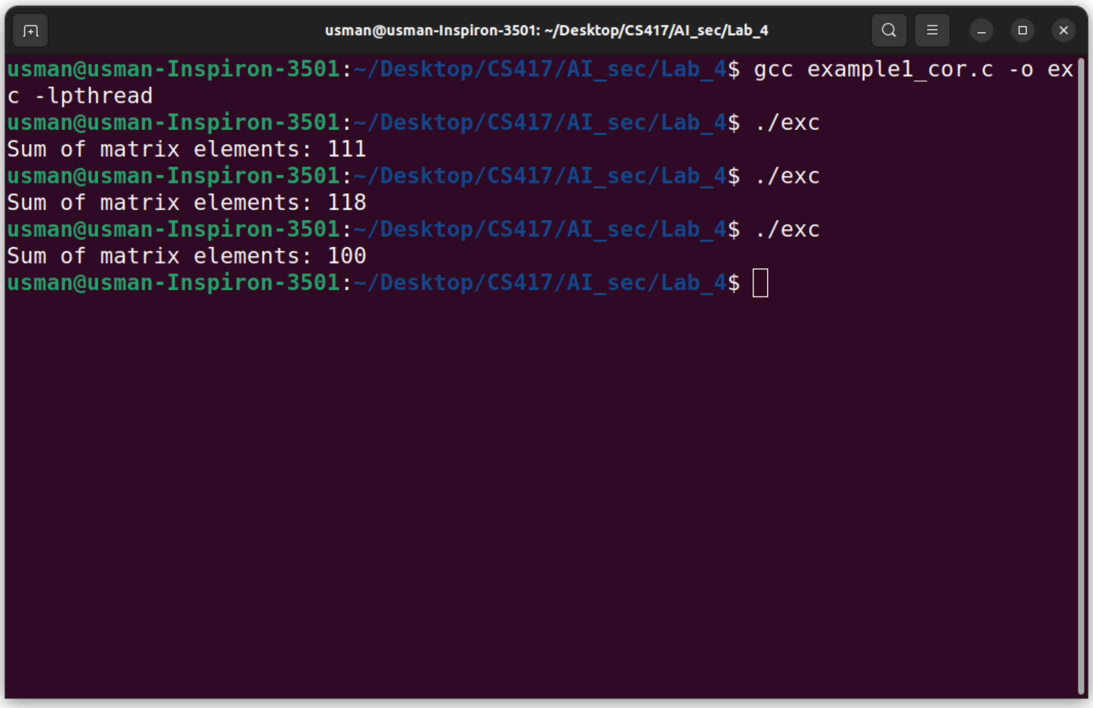
- The producer threads produce data, update a shared buffer and signal the condition variable when data is available.
- The consumer threads wait on the condition variable until data is available, consume the data and repeat the process.

## 4.5 Example 1. Matrix sum in producer-consumer fashion

The problem involves initializing a matrix and calculating its sum using multiple threads in a producer-consumer fashion. The challenge is ensuring the matrix is fully initialized before any threads start calculating the sum to prevent data inconsistencies.

Within this program, a dedicated thread named "initializeMatrix" is responsible for populating the matrix and issuing signals to inform other threads of its readiness. The synchronization of threads is facilitated by using a conditional variable named "cond," ensuring that calculation threads remain in a waiting state until the initialization process reaches completion. To concurrently calculate the sum of matrix elements, multiple threads named "calculateSum" threads are employed. These threads collaborate seamlessly while guaranteeing the safety of updates to the global "sum" variable by utilizing a mutex, aptly named "mutex." Furthermore, the main thread undertakes the responsibilities of managing thread creation, ensuring their proper joining, and ultimately presenting the computed sum as the program's output. In essence, this solution effectively and securely orchestrates the tasks of matrix initialization and sum calculation within a multi-threaded environment, all while adhering to the principles inherent in producer-consumer scenarios.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

#define MATRIX_SIZE 5
#define NUM_THREADS 3

int matrix[MATRIX_SIZE][MATRIX_SIZE];
int sum = 0;
int initialization_done = 0; // New variable to track
   ↪ initialization
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void* initializeMatrix(void* arg) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            matrix[i][j] = rand() % 10;
        }
    }

    pthread_mutex_lock(&mutex);
    initialization_done = 1; // Signal that initialization is
       ↪ complete
    pthread_cond_broadcast(&cond); // Broadcast the signal
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

void* calculateSum(void* arg) {
    pthread_mutex_lock(&mutex);
    while (!initialization_done) {
        pthread_cond_wait(&cond, &mutex); // Wait until
           ↪ initialization is complete
    }
    pthread_mutex_unlock(&mutex);

    int thread_id = *(int*)arg;
    int partial_sum = 0;

    for (int i = thread_id; i < MATRIX_SIZE; i += NUM_THREADS)
       ↪  {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            partial_sum += matrix[i][j];
        }
    }

    pthread_mutex_lock(&mutex);
    sum += partial_sum;
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS + 1];
    int thread_ids[NUM_THREADS];

    srand(time(NULL));

    pthread_create(&threads[0], NULL, initializeMatrix, NULL);
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i + 1], NULL, calculateSum, &
           ↪ thread_ids[i]);
    }

    for (int i = 0; i < NUM_THREADS + 1; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Sum of matrix elements: %d\n", sum);

    return 0;
}
```



```
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ gcc example1_cor.c -o ex
c -lpthread
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ ./exc
Sum of matrix elements: 111
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ ./exc
Sum of matrix elements: 118
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ ./exc
Sum of matrix elements: 100
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ 
```

Figure 11: Matrix sum

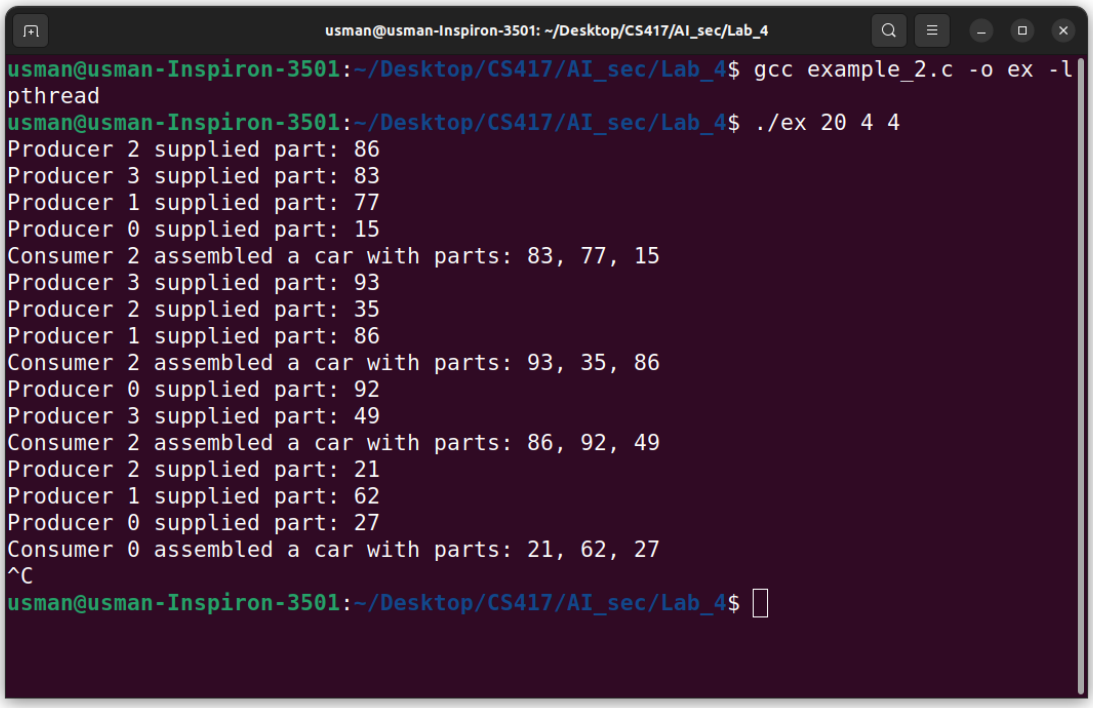
## 4.6 Example 2. Multi-threaded Producer-Consumer Simulation with Thread Synchronization, Car assembly

The provided C program tackles the challenge of managing a multi-threaded system comprising both producers and consumers interacting with a shared buffer. Its primary aim is to demonstrate the intricacies of thread synchronization and coordination within a simplified assembly line scenario reminiscent of car production. Several key elements and issues are present in this program, starting with thread management, where the program dynamically creates producer and consumer threads as specified by the command-line arguments NUM_PRODUCERS and NUM_CONSUMERS. A shared buffer, represented by the array variable, serves as the central point of interaction. Producers add parts to this buffer, while consumers assemble cars using these parts. Concurrency issues are a significant concern in this setup, as multiple threads access the shared buffer simultaneously, potentially leading to race conditions and data integrity problems. To address this, the program employs synchronization techniques, specifically a mutex (mutex), which ensures mutual exclusion when threads access the shared buffer, preventing data corruption.

The buffer's size, determined by the BUFFER_SIZE command-line argument, dictates the maximum number of parts the buffer can store. Producer and consumer threads operate in infinite loops, continuously generating parts and assembling cars. The solution involves careful initialization, dynamic memory allocation for the buffer, and mutex initialization for synchronization. Each producer creates a separate thread, generating random parts and attempting to add them to the shared buffer while maintaining thread safety through mutex locking and unlocking. Similarly, consumer threads, also managed in separate threads, continuously attempt to assemble cars if enough parts are available.

The program employs mutex locking and unlocking effectively to provide mutual exclusion, ensuring that only one thread can access the shared buffer at a time. After threads complete their operations, they unlock the mutex, allowing other threads to access the buffer. To avoid premature program termination, the main function waits for all producer and consumer threads to finish using pthread_join. Finally, it properly releases system resources by destroying the mutex with pthread_mutex_destroy.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int BUFFER_SIZE;
int NUM_PRODUCERS;
int NUM_CONSUMERS;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t buffer_not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t buffer_not_empty = PTHREAD_COND_INITIALIZER;

int *array;
int count = 0;

void *producer(void *arg) {
    int thread_no = *((int *)arg);
    while (1) {
        sleep(1);
        int part = rand() % 100;

        pthread_mutex_lock(&mutex);
        while (count >= BUFFER_SIZE) {
            pthread_cond_wait(&buffer_not_full, &mutex);
        }

        array[count] = part;
        printf("Producer %d supplied part: %d\n", thread_no,
           ↪ part);
        count++;

        pthread_cond_signal(&buffer_not_empty);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *consumer(void *arg) {
    int thread_no = *((int *)arg);
    while (1) {
        sleep(1);
        pthread_mutex_lock(&mutex);
        while (count < 3) {
            // Buffer does not have enough parts to assemble a
               ↪  car, wait for a signal from producers
            pthread_cond_wait(&buffer_not_empty, &mutex);
        }

        printf("Consumer %d assembled a car with parts: %d, %d
            ↪ , %d\n", thread_no, array[count-3], array[count
            ↪ -2], array[count-1]);
        count -= 3;

        pthread_cond_signal(&buffer_not_full);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    if(argc != 4){
        printf("Invalud arguments");
        return 1;
    }

    BUFFER_SIZE = atoi(argv[1]);
    NUM_PRODUCERS = atoi(argv[2]);
    NUM_CONSUMERS = atoi(argv[3]);

    array = (int*)malloc(BUFFER_SIZE * sizeof(int));

    pthread_t producers[NUM_PRODUCERS];
    pthread_t consumers[NUM_CONSUMERS];

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        int *producer_no = malloc(sizeof(int));
        *producer_no = i;
        pthread_create(&producers[i], NULL, producer,
            ↪ producer_no);
    }
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        int *comsumer_no = malloc(sizeof(int));
        *comsumer_no = i;
        pthread_create(&consumers[i], NULL, consumer,
           ↪ comsumer_no);
    }

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&buffer_not_full);
    pthread_cond_destroy(&buffer_not_empty);

    return 0;
}
```



```
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ gcc example_2.c -o ex -l
pthread
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ ./ex 20 4 4
Producer 2 supplied part: 86
Producer 3 supplied part: 83
Producer 1 supplied part: 77
Producer 0 supplied part: 15
Consumer 2 assembled a car with parts: 83, 77, 15
Producer 3 supplied part: 93
Producer 2 supplied part: 35
Producer 1 supplied part: 86
Consumer 2 assembled a car with parts: 93, 35, 86
Producer 0 supplied part: 92
Producer 3 supplied part: 49
Consumer 2 assembled a car with parts: 86, 92, 49
Producer 2 supplied part: 21
Producer 1 supplied part: 62
Producer 0 supplied part: 27
Consumer 0 assembled a car with parts: 21, 62, 27
^C
usman@usman-Inspiron-3501:~/Desktop/CS417/AI_sec/Lab_4$ 
```

Figure 12: Multi-threaded Producer-Consumer Simulation with Thread Synchronization

# 5 Lab 5: Introduction to OpenMP

## Objectives

- Provide an introduction to OpenMP programming model.
- Familiarize students with OpenMP terms and directives.
- Teach the loop construct and reduction operators in OpenMP.

## Deliverables

- Proficiency in using OpenMP for parallel programming.
- Successful implementation of OpenMP programs with loop constructs and reduction operators.
- Completion of exercises, including calculating factorial using reduction operators.

## 5.1 Basics

OpenMP is a parallel programming framework for C/C++ and Fortran. It has gained quite a bit of traction in recent years, primarily due to its simplicity while still providing good performance. In this lab we will be taking a quick peek at a small fraction of its features. OpenMP uses shared memory, meaning all threads can access the same address space. The alternative to this is distributed memory, which is prevalent on clusters where data must be explicitly moved between address spaces. Many programmers find shared memory easier to program since they do not have to worry about moving their data, but it is usually harder to implement in hardware in a scalable way. Later in the lab we will declare some memory to be thread local (accessible only by the thread that created it) for performance reasons, but the programming framework provides the flexibility for threads to share memory without programmer effort.
**Goals of OpenMP**

- **Standardization:** Provide a standard among a variety of shared memory architectures/platforms
- **Lean and Mean:** Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use:** Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach and also provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:** Supports Fortran (77, 90, and 95), C, and C++

## 5.2 OpenMP Programming Model

There are many types of parallelism and patterns for exploiting it, and OpenMP chooses to use a nested fork-join model. By default, an OpenMP program is a normal sequential program, except for regions that the programmer explicitly declares to be executed in parallel. In the parallel region, the framework creates (fork) a set number of threads. Typically these threads all execute the same instructions, just on different portions of the data. At the end of the parallel region, the framework waits for all threads to complete (join) before it leaves that region and continues sequentially. The main components of OpenMP API are given as:

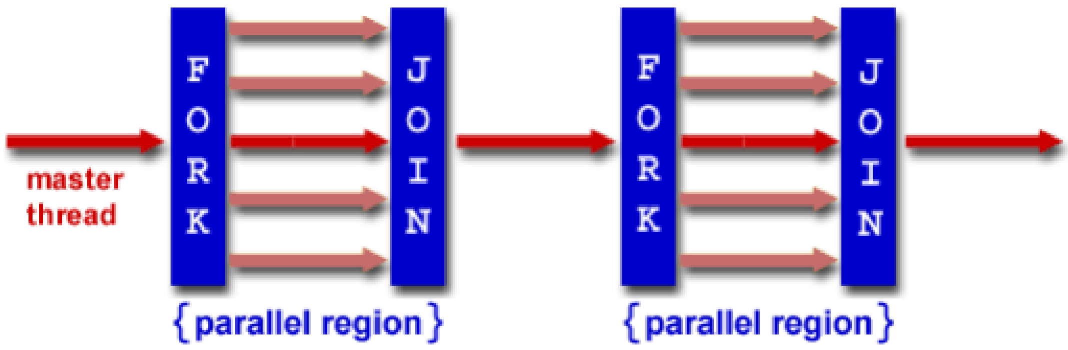


Figure 13: OpenMP programming model

- Compiler Directives
- Runtime library routines
- Environment Variables

The basic code structure of OpenMP in C/C++ is given as:

```
main()
{
     Serial code ..
     .
     #pragma omp parallel
     {
     //parallel region
     }
     Resume serial code
}
```

## 5.3 Important Terms for OpenMP environment

**Construct:** A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct.
**directive:** A C or C++ #pragma followed by the omp identifier, other text, and a new line. The directive specifies program behavior.
**Region:** A dynamic extent **Thread:** An execution entity having a serial flow of control, a set of private variables, and access to shared variables.
**master thread:** The thread that creates a team when a parallel region is entered.
**serial region:** Statements executed only by the master thread outside of the dynamic extent of any parallel region.
**parallel region:** Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.
**Variable:** An identifier, optionally qualified by namespace names, that names an object.
**Private:** A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within a parallel region, a threadprivate directive, a private, firstprivate, lastprivate, or reduction clause, or use of the variable as a forloop control variable in a for loop immediately following a for or parallel for directive.
**Shared:** A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.
**Team:** One or more threads cooperating in the execution of a construct.

## 5.4 First OpenMP Program

The following program shows how to get the number of threads and print "Hello world" using threads.

```
#include<omp.h>
int main()
{
   # pragma omp parallel
   {
     int thread_ID = omp_get_thread_num();
     printf("hello world%d\n", thread_ID);
   }
 }
```

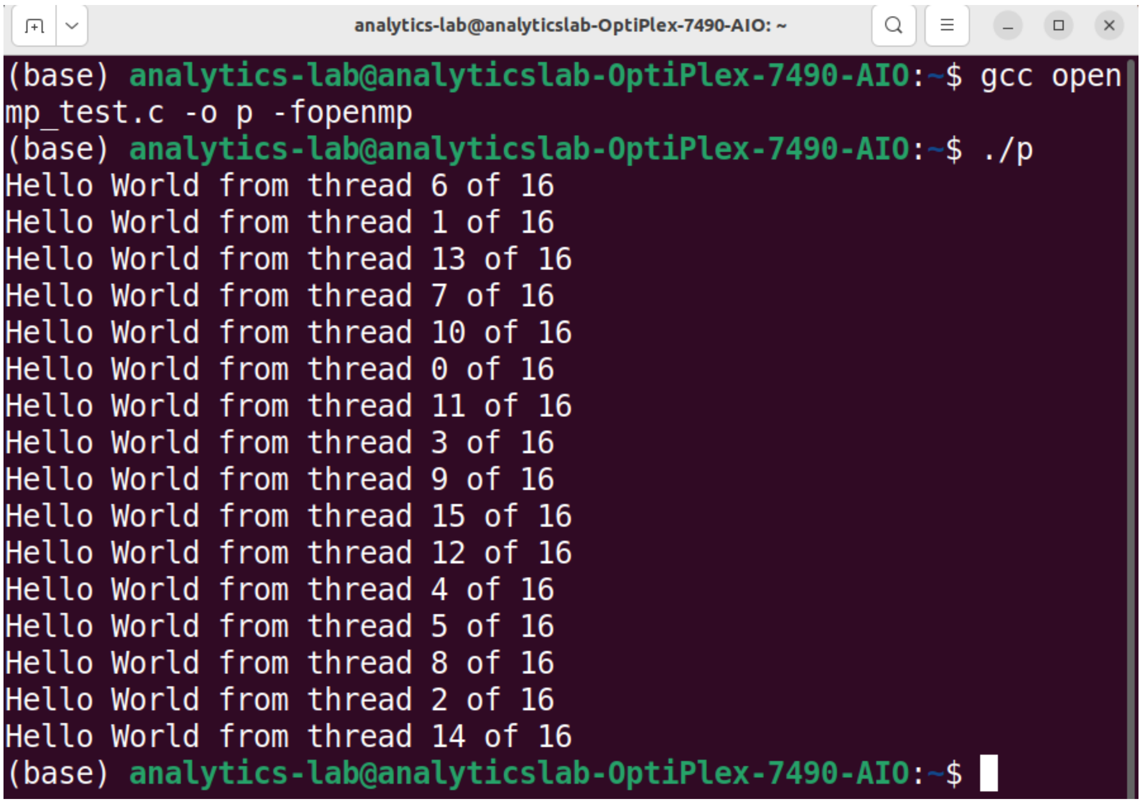
In order to execute the OpenMP code and set the number of threads, the following commands should be used:

```
export OMP_NUM_THREADS=2
gcc -fopenmp hello.c -o h
./h
```

However, the number of threads can be specified in the code instead of the terminal. The following code will help us to do this. All the threads are used in the output and perform their respective tasks.

```
#include<stdlib.h>
#include<stdio.h>
#include<omp.h>
int main()
{
   # pragma omp parallel
   {
     int thread_ID = omp_get_thread_num();
     int num_threads = omp_get_num_threads();
     print("Hello World from thread %d of %d \n", thread_id
        ↪ , num_threads);
   }
   return 0;
 }
```

**What are problems here?**
**The order of the print is not correct. What is the possible solution? It is a home task.**



```
(base) analytics-lab@analyticslab-OptiPlex-7490-AIO:~$ gcc open
mp_test.c -o p -fopenmp
(base) analytics-lab@analyticslab-OptiPlex-7490-AIO:~$ ./p
Hello World from thread 6 of 16
Hello World from thread 1 of 16
Hello World from thread 13 of 16
Hello World from thread 7 of 16
Hello World from thread 10 of 16
Hello World from thread 0 of 16
Hello World from thread 11 of 16
Hello World from thread 3 of 16
Hello World from thread 9 of 16
Hello World from thread 15 of 16
Hello World from thread 12 of 16
Hello World from thread 4 of 16
Hello World from thread 5 of 16
Hello World from thread 8 of 16
Hello World from thread 2 of 16
Hello World from thread 14 of 16
(base) analytics-lab@analyticslab-OptiPlex-7490-AIO:~$ 
```

Figure 14: Output

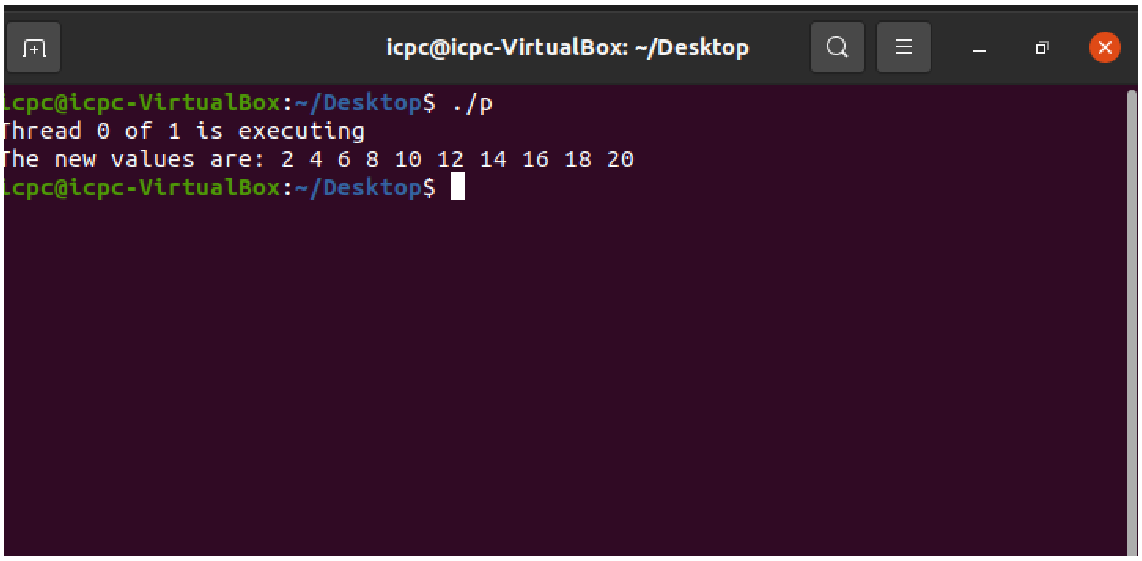
Another example is to sum the two arrays into a single array within a parallel region.

**Think!!! If it is executing in parallel?**

```
#include<stdlib.h>
#include<stdio.h>
#include<omp.h>
int main(int argc, char const *argv[])
{
     int A[10] = {1,2,3,4,5,6,7,8,9,10};
     int B[10] = {1,2,3,4,5,6,7,8,9,10};
     int C[10];
   # pragma omp parallel
   {
     int thread_ID = omp_get_thread_num();
     int num_threads = omp_get_num_threads();
     print("Thread %d of %d is executing \n", thread_id,
        ↪ num_threads);
     for (int i=0; i<10; i++)
     {
         C[i]= A[i]+B[i];
     }
     printf("The new values are:");
     for (int j=0;j<10;j++)
     {
         printf(" %d", C[j]);
     }
     print("\n");
   }

   return 0;
 }
```

## 5.5 Sharing of work among threads using Loop Construct

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region to have an effect. It is ignored if a work-sharing directive is encountered in an inactive parallel region or the sequential part of the program. Since work-sharing directives may occur in procedures invoked both from within a parallel region and outside of any parallel regions, they may be exploited during some calls and ignored during others.



```
icpc@icpc-VirtualBox:~/Desktop$ ./p
Thread 0 of 1 is executing
The new values are: 2 4 6 8 10 12 14 16 18 20
icpc@icpc-VirtualBox:~/Desktop$ 
```

Figure 15: Output

The work-sharing constructs are listed below.

- #pragma omp for: Distribute iterations over the threads

The two main rules regarding work-sharing constructs are as follows:

- All threads must encounter each work-sharing region in a team or by none.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier to entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its work. However, the programmer can suppress this by using the nowait clause.

## 5.6 The Loop Construct

The loop construct causes the iterations of the loop immediately following it to be executed in parallel. At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features.

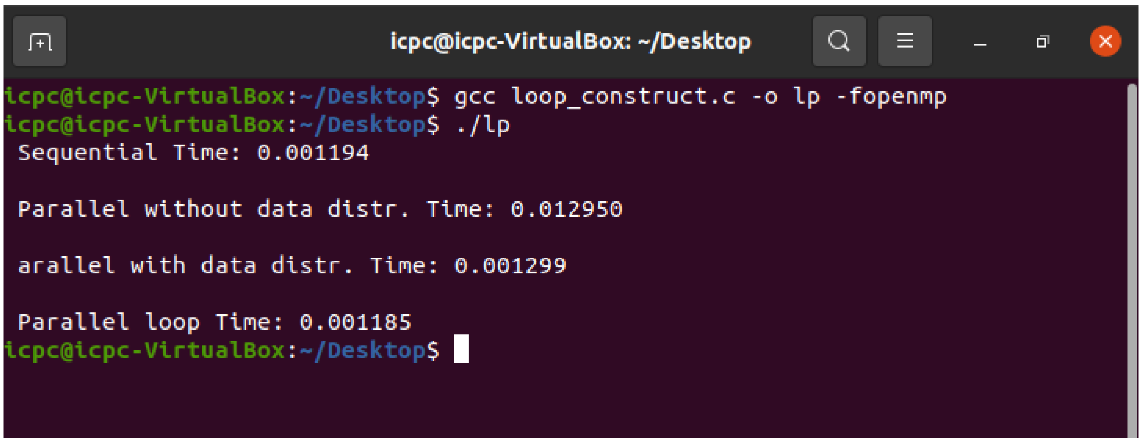
### 5.6.1 Syntax and Example:

The loop construct with time profiling is given below. The code first computes the addition of two arrays sequentially, then computes the sum in parallel regions without data distribution. After that, the data distribution is done, and finally, a loop construct is used to automatically distribute the data.

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

int main()
{
        int sz=500000;
        int A[sz];
        int B[sz];
        int C[sz];
    for(int i=0;i<sz;i++)
    {
        A[i]=rand()%100;
                B[i]=rand()%10;
        }
        //sequential code
        clock_t begin= clock();
        for (int i=0;i<sz;i++)
        {
                C[i]=A[i]*B[i];
        }
        clock_t end= clock();
        double time_spent = (double) (end-begin)/
           ↪ CLOCKS_PER_SEC;
        printf(" Sequential Time: %f\n", time_spent );

        printf("\n");
        omp_set_num_threads(8);
        //OpenMP parallel region without data distribution
        double s_time = omp_get_wtime();
        #pragma omp parallel
        {
                for (int i=0;i<sz;i++)
                {
                        C[i]=A[i]*B[i];
                }
        }
        double time_omp = omp_get_wtime() - s_time;
        printf(" Parallel without data distr. Time: %f\n",
           ↪ time_omp );
        printf("\n");
        //OpenMP parallel region with data distribution
        double s_time_1 = omp_get_wtime();
        #pragma omp parallel
        {
                int t_id, thread_num, st_ind, end_ind;
                t_id = omp_get_thread_num();
                thread_num = omp_get_num_threads();

                st_ind = t_id * sz / thread_num;
                end_ind = (t_id+1) * sz / thread_num;

                if (t_id==thread_num-1)
                {
                        end_ind=sz;
                }

                for (int i=st_ind;i<end_ind;i++)
                {
                        C[i]=A[i]*B[i];
                }
        }
        double time_omp_1 = omp_get_wtime() - s_time_1;
        printf(" arallel with data distr. Time: %f\n",
           ↪ time_omp_1 );
        printf("\n");
        double s_time_2 = omp_get_wtime();
        #pragma omp parallel
        {
                #pragma omp for

                for (int i=0;i<sz;i++)
                {
                        C[i]=A[i]*B[i];
                }
        }
        double time_omp_2 = omp_get_wtime() - s_time_2;
        printf(" Parallel loop Time: %f\n", time_omp_2);

        return 0;
}
```

## 5.7 Reduction Operator

The REDUCTION clause performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.



```
icpc@icpc-VirtualBox:~/Desktop$ gcc loop_construct.c -o lp -fopenmp
icpc@icpc-VirtualBox:~/Desktop$ ./lp
 Sequential Time: 0.001194

 Parallel without data distr. Time: 0.012950

 arallel with data distr. Time: 0.001299

 Parallel loop Time: 0.001185
icpc@icpc-VirtualBox:~/Desktop$ 
```

Figure 16: Arrays sum

### 5.7.1 Format: REDUCTION (OPERATOR: LIST)

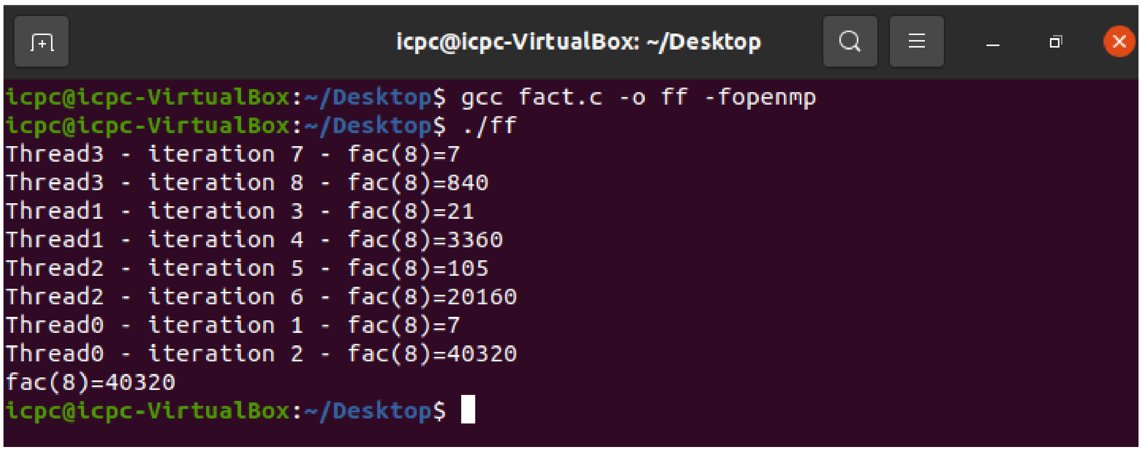
- Variables in the list must be named scalar variables. They can not be array or structure-type variables. They must also be declared SHARED in the enclosing context.
- Reduction operations may not be associative for real numbers.
- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements that have one of the following forms:

```
xx = x op expr
x = expr op x (except subtraction)
x binop = expr
x++
++x
x--
--x
```

### 5.7.2 Example: Calculate the factorial of the given number.

The code computes the factorial of a given number:

```
#include <stdio.h>
#include <omp.h>
int main(){
        int i,n=8;
        int fac=1;
        omp_set_num_threads(4);
        #pragma omp parallel for ordered shared(n)
        private(i)      reduction(*:fac)
        for(i=1;i<=n;i++){
                fac*=i;
                printf("Thread%d - iteration %d - fac(%d)=%d\n
                   ↪ ",
                omp_get_thread_num(),i,n,fac);

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



```
icpc@icpc-VirtualBox:~/Desktop$ gcc fact.c -o ff -fopenmp
icpc@icpc-VirtualBox:~/Desktop$ ./ff
Thread3 - iteration 7 - fac(8)=7
Thread3 - iteration 8 - fac(8)=840
Thread1 - iteration 3 - fac(8)=21
Thread1 - iteration 4 - fac(8)=3360
Thread2 - iteration 5 - fac(8)=105
Thread2 - iteration 6 - fac(8)=20160
Thread0 - iteration 1 - fac(8)=7
Thread0 - iteration 2 - fac(8)=40320
fac(8)=40320
icpc@icpc-VirtualBox:~/Desktop$ 
```

Figure 17: Factorial

# 6 Lab 6: Sharing of work among threads in an OpenMP program using Sections Construct

## Objectives

- Introduce the Sections Construct in OpenMP.
- Explain how to share work among threads using Sections.
- Introduce the Tasks directive in OpenMP.
- Apply these concepts to solve complex problems.

## Deliverables

- Proficiency in using the Sections Construct for work distribution.
- Ability to use the Tasks directive in OpenMP.
- Successful resolution of complex problems involving Sections and Tasks.

## 6.1 Introduction

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region in order to have an effect. If a work-sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is ignored. Since work-sharing directives may occur in procedures invoked both from within a parallel region and outside of any parallel regions, they may be exploited during some calls and ignored during others.
The work-sharing constructs are listed below.

- #pragma omp for: Distribute iterations over the threads
- #pragma omp sections: Distribute independent work units
- #pragma omp single: Only one thread executes the code block

The two main rules regarding work-sharing constructs are as follows:

- All threads must encounter each work-sharing region in a team or by none.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its work. However, the programmer can suppress this by using the nowait clause.

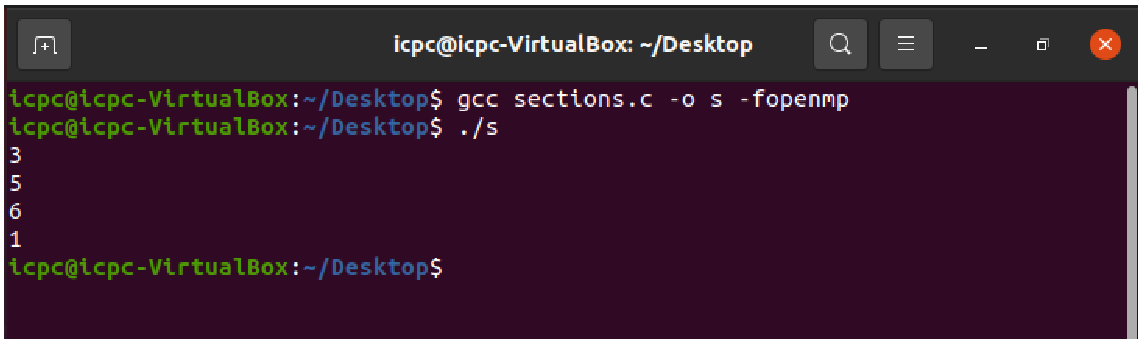
## 6.2 Sections Construct

The sections construct is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which will be executed by one of the threads. It consists of two directives: first, **#pragma omp sections:** to indicate the start of the construct and second, the **#pragma omp section:** to mark each distinct section. Each section must be a structured block of code that is independent of the other sections.
At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once. If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks. If there are fewer code blocks than threads, the remaining threads will be idle. Note that the assignment of code blocks to threads is implementation-dependent.

## 6.3 Example

The example shows the computation of three different operations in parallel:

```
#include <stdio.h>
#include <omp.h>
int summation(int x)
{
        return x+2;
}
int product(int x)
{
        return x*2;
}
int sub(int x)
{
        return x-2;
}
int main(){
        int x;
        scanf("%d",&x);

        #pragma omp parallel sections num_threads(6)
        {
                #pragma omp section
                        printf("%d \n", summation(x));
                #pragma omp section
                        printf("%d \n", product(x));
                #pragma omp section
                        printf("%d \n", sub(x));
        }

        return 0;
}
```



```
icpc@icpc-VirtualBox:~/Desktop$ gcc sections.c -o s -fopenmp
icpc@icpc-VirtualBox:~/Desktop$ ./s
3
5
6
1
icpc@icpc-VirtualBox:~/Desktop$
```

Figure 18: Three operations

## 6.4 Tasks directive

OpenMP Tasks suitable for irregular problems (problems without loops, unbounded loops, recursive algorithms, etc.) Tasks are independent units of work. Threads are assigned to perform the work of each task.

- Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above

An example is shown below:

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

int summation(int *array, int N)
{
        if (N==0) return 0;

        else if (N==1) return *array;

        int half =N/2;
```

```
         return summation(array, half) + summation(array+half,
         N-half);
}

int summation_1(int *A, int N)
{
        if (N==0) return 0;

        else if (N==1) return *A;

        int half =N/2;
        int x,y;
        omp_set_num_threads(4);
        #pragma omp parallel
        #pragma omp single
        {
                #pragma omp task shared(x)
                x = summation_1(A, half);
                #pragma omp task shared(y)
                y = summation_1(A+half, N-half);
                   ↪

                #pragma omp taskwait
                x += y;
        }

        return x;
}

int main(){

        int sz=500;
        int A[sz];
    for(int i=0;i<sz;i++)
    {
        A[i]=rand()%10;
        }


        printf("%d\n",  summation(A,sz));
        #pragma omp parallel num_threads(4) private(tid)
        {
                #pragma omp single
                {
                 tid = omp_get_thread_num();
                 printf("Hello world from (%d) \n", tid);
```

```
                        printf("Sum = %d by %d \n",, fib(A,sz), tid);
                            ↪
                    }
            }
//          printf("%d\n",  summation_1(A,sz));

            return 0;
}
```

**Why is a single construct being used?** One thread enters a single construct and creates all tasks (you don't want all threads to enter a construct and create duplicate tasks). This single thread will create a pipeline (task queue) on which other tasks will be placed. By default, all threads will execute tasks only when a single construct (thread) finishes its job. This can be overridden by passing nowait clause to the single construct.

1. **Lab Task: Fibonacci Series f(1) = 1, f(2) = 1, f(n) = f(n - 1) + f(n - 2)**
2. **Travese Linkedlist in parallel**

# 7 Lab 7: Introduction to CUDA

## Objectives

- Understand fundamental GPU architecture concepts.
- Differentiate between CPU and GPU functionalities.
- Familiarize with CUDA programming principles and grasp basic CUDA syntax and structure.
- Successfully install CUDA toolkit and drivers and ensure functional GPU environment for practical exercises.

## Deliverables

- Step-by-step documentation for CUDA toolkit and GPU driver installation.
- A collection of simple CUDA programs showcasing syntax and structure.
- A verified CUDA installation report

## 7.1 Introduction to GPU Computing

In high-performance computing and parallel processing, General-Purpose Graphics Processing Units (GPGPUs) have emerged as powerful tools. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface (API) created by NVIDIA. It allows developers to harness the computational power of NVIDIA GPUs for general-purpose computing tasks beyond graphics rendering. CUDA has become increasingly popular due to its ability to accelerate data-intensive and computationally intensive applications dramatically. In this section, we will delve into CUDA's significance in the realm of parallel computing and provide an overview of GPU architecture.

### 7.1.1 What is CUDA and its importance in parallel computing

CUDA is a parallel computing platform developed by NVIDIA, designed to allow developers to access the immense computational power of NVIDIA GPUs for non-graphics applications. It provides a framework for writing highly parallel programs that can execute on the thousands of cores within a GPU. The importance of CUDA in parallel computing lies in its capability to address tasks that require massive parallelism efficiently. By programming for CUDA, developers can take advantage of the parallel nature of GPUs and achieve substantial speedups in various domains, such as scientific simulations, deep learning, image and signal processing, and more.

### 7.1.2 Concept of GPU architecture

GPU architecture is fundamentally different from CPU architecture. A CPU (Central Processing Unit) is optimized for sequential processing and is designed for tasks that require fast single-threaded execution. In contrast, a GPU (Graphics Processing Unit) is engineered for parallelism. GPUs contain many smaller processing units known as CUDA cores, which work in unison to perform computations. This parallel architecture allows GPUs to process thousands of tasks simultaneously, making them ideal for highly parallel workloads.

### 7.1.3 Advantages of using CUDA for parallel programming

CUDA offers several significant advantages for parallel programming, making it a preferred choice for developers:

- **Massive Parallelism:** CUDA enables the utilization of thousands of CUDA cores on a GPU, resulting in massive parallelism. This is particularly advantageous for applications that can be parallelized, as it can lead to dramatic performance improvements.
- **Heterogeneous Computing** CUDA allows developers to use both the CPU and GPU in a heterogeneous computing environment, optimizing the strengths of each processor for the respective workload.
- **Access to Cutting-Edge GPU Features:** CUDA provides access to advanced GPU features and libraries, allowing developers to leverage the latest innovations in GPU technology for their applications.

### 7.1.4 GPU vs. CPU

GPUs and CPUs differ significantly in terms of architecture and purpose:
**Parallelism:** GPUs are designed for massively parallel execution, with thousands of cores working together on tasks, whereas CPUs are optimized for sequential processing. **Execution:** CPUs are well-suited for tasks that require low-latency, single-threaded execution, such as operating system operations and general-purpose computation. GPUs excel in data-parallel tasks, where the same operation is performed on multiple data elements simultaneously. **Throughput vs. Latency:** GPUs prioritize throughput, aiming to complete many tasks in parallel, while CPUs prioritize low-latency execution of individual tasks. **Memory Hierarchy** CPUs often have a complex memory hierarchy with various caches, while GPUs have simpler, shared memory structures designed for parallel workloads.

## 7.2 Getting started with CUDA in C

### 7.2.1 System Requirements

To use NVIDIA CUDA on your system, you will need the following installed:

- CUDA-capable GPU
- A supported version of Linux with a gcc compiler and toolchain
- CUDA Toolkit (available at https://developer.nvidia.com/cuda-downloads)

Some actions must be taken before the CUDA Toolkit, and Driver can be installed on Linux:

- Verify the system has a CUDA-capable GPU.
- Verify the system is running a supported version of Linux.
- Verify the system has gcc installed.
- Verify the system has the correct kernel headers and development packages installed.
- Download the NVIDIA CUDA Toolkit.
- Handle conflicting installation methods.

### 7.2.2 Verify the prerequisites

To verify that your GPU is CUDA-capable, go to your distribution's equivalent of System Properties, or, from the command line, enter:

```
    lspci | grep -i nvidia
```

If you do not see any settings, update the PCI hardware database Linux maintains by entering update-pciids (generally found in /sbin) at the command line and rerunning the previous lspci command.
The CUDA Development Tools are only supported on some specific distributions of Linux. These are listed in the CUDA Toolkit release notes. To determine which distribution and release number you're running, type the following at the command line:

```
    uname -m && cat /etc/*release
```

You should see output similar to the following, modified for your particular system:

```
x86_64
Red Hat Enterprise Linux Workstation release 6.0 (Santiago)
```

The x86_64 line indicates you are running on a 64-bit system. The remainder gives information about your distribution.
To verify the version of gcc installed on your system, type the following on the command line:

```
gcc --version
```

If an error message displays, you must install the development tools from your Linux distribution (See Lab 1).

### 7.2.3 Installation of CUDA

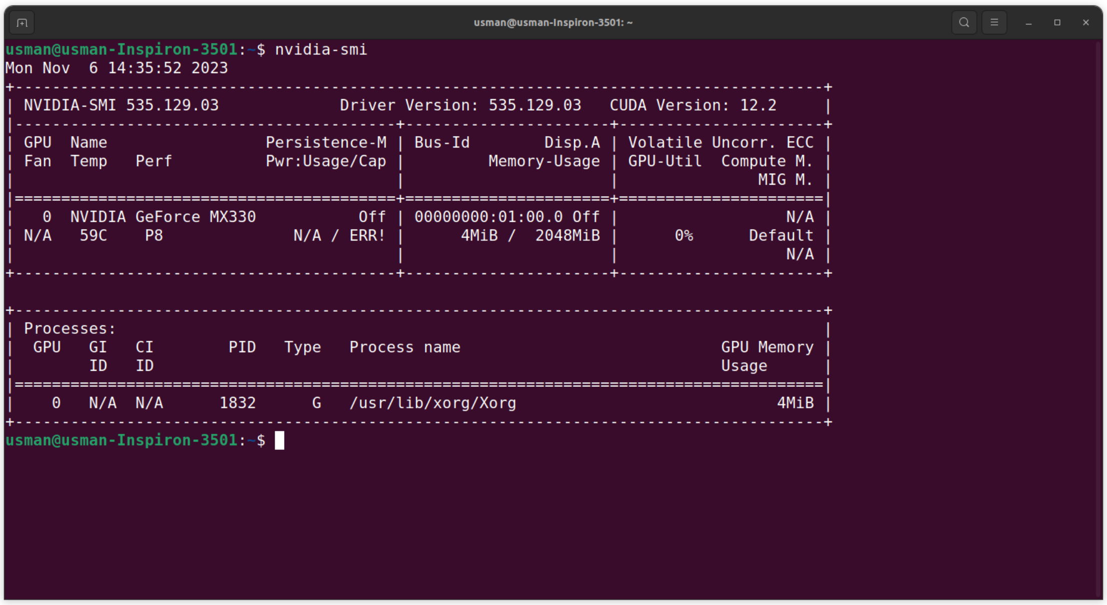
Setting up CUDA on Ubuntu for C language programming involves several steps. Below, a general process is given. Keep in mind that the specific versions of CUDA and Ubuntu may vary, so make sure to use the latest versions available.
Install the NVIDIA GPU drivers. Open a terminal and use the following command to install the recommended driver package:

```
sudo ubuntu-drivers autoinstall
```

Confirm that the driver is installed correctly by running:

```
nvidia-smi
```

After the successful installation of the drivers, you will get the following output
Next, we will install the CUDA toolkit using the following commands. Please note



```
usman@usman-Inspiron-3501:~$ nvidia-smi
Mon Nov  6 14:35:52 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.129.03             Driver Version: 535.129.03   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce MX330           Off | 00000000:01:00.0 Off |                  N/A |
| N/A   59C    P8              N/A / ERR! |      4MiB /  2048MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1832      G   /usr/lib/xorg/Xorg                            4MiB |
+---------------------------------------------------------------------------------------+
usman@usman-Inspiron-3501:~$
```

Figure 19: Nvidia Drivers

that, the commands are customizable and varies. You can get commands for your system at (https://developer.nvidia.com/cuda-downloads).

```
wget https://developer.download.nvidia.com/compute/cuda/repos/
    ↪ ubuntu2204/x86_64/cuda-ubuntu2204.pin
sudo mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-
    ↪ repository-pin-600
wget https://developer.download.nvidia.com/compute/cuda
    ↪ /12.3.0/local_installers/cuda-repo-ubuntu2204-12-3-
    ↪ local_12.3.0-545.23.06-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu2204-12-3-local_12
    ↪ .3.0-545.23.06-1_amd64.deb
sudo cp /var/cuda-repo-ubuntu2204-12-3-local/cuda-*-keyring.
    ↪ gpg /usr/share/keyrings/
```

```
sudo apt-get update
sudo apt-get -y install cuda-toolkit-12-3
```

Edit your /.bashrc file to set the environment variables for CUDA. Add the following lines to the file:

```
    export PATH=/usr/local/cuda-11/bin:$PATH
    export LD_LIBRARY_PATH=/usr/local/cuda-11/lib64:
       ↪ $LD_LIBRARY_PATH
```

After modifying the file, run source /.bashrc to apply the changes. At the end, to confirm that CUDA is installed, run:

```
    nvcc --version
```

## 7.3 First CUDA Program

### 7.3.1 Example 1:

This code is a simple example of GPU programming using CUDA to perform addition of two numbers. It consists of a CPU part and a GPU part. The CPU part initializes two integer variables, 'a' and 'b', with values 5 and 7, respectively. It then allocates GPU memory to store the result of the addition and launches a GPU kernel called 'addGPU.' This kernel takes 'a' and 'b' as parameters, adds them together, and stores the result in the allocated GPU memory. After the GPU kernel execution, the code copies the result from the GPU back to the CPU memory. Finally, it frees the GPU memory. The CPU and GPU results are then displayed, demonstrating the addition of 'a' and 'b' both on the CPU and GPU. This code serves as a basic introduction to GPU programming concepts, including memory allocation, kernel execution, and data transfer between the CPU and GPU.

```
#include <iostream>

// CUDA kernel to perform addition on the GPU
__global__ void addGPU(int a, int b, int* result_gpu) {
    *result_gpu = a + b;
}

int main() {
    // CPU variables
    int a = 5;
    int b = 7;
    int result_gpu = 0;
    int* d_result_gpu;  // Pointer to store the GPU result

    // Allocate GPU memory for the result
    cudaMalloc((void**)&d_result_gpu, sizeof(int));
    // Launch GPU kernel
    addGPU<<<1, 1>>>(a, b, d_result_gpu);

    // Copy result from GPU to CPU
    cudaMemcpy(&result_gpu, d_result_gpu, sizeof(int),
       ↪ cudaMemcpyDeviceToHost);

    // Free GPU memory
    cudaFree(d_result_gpu);

    std::cout << "CPU Result: " << a + b << std::endl;
    std::cout << "GPU Result: " << result_gpu << std::endl;

    return 0;
}
```

### 7.3.2 Example 2:

This CUDA program calculates the sum of an array using both GPU and CPU computations while measuring and displaying their respective execution times. The code first initializes an array with data and allocates memory on the GPU for the array and the result. It then launches a GPU kernel to compute the sum in parallel. Simultaneously, a CPU function computes the sum on the CPU. The code records and displays the execution times for both GPU and CPU calculations, providing a clear comparison of their processing speed. This example showcases the advantages of GPU parallelism for data-intensive tasks and demonstrates the difference in execution times between GPU and CPU computations for summing an array.

```
#include <iostream>
#include <ctime>

// CUDA kernel to calculate the sum of an array
__global__ void sumArrayGPU(int* array, int* result, int n) {
    int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;

    int sum = 0;

    for (int i = threadID; i < n; i += stride) {
        sum += array[i];
    }

    atomicAdd(result, sum);
}
```

```
// CPU function to calculate the sum of an array
int sumArrayCPU(int* array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += array[i];
    }
    return sum;
}

int main() {
    const int arraySize = 1000000; // Size of the array
    int array[arraySize];

    // Initialize the array with some data
    for (int i = 0; i < arraySize; i++) {
        array[i] = i;
    }

    int* d_array; // GPU pointer to store the array
    int* d_result; // GPU pointer to store the result

    // Allocate GPU memory for the array and result
    cudaMalloc((void**)&d_array, arraySize * sizeof(int));
    cudaMalloc((void**)&d_result, sizeof(int));

    // Copy the array to the GPU
    cudaMemcpy(d_array, array, arraySize * sizeof(int),
       ↪ cudaMemcpyHostToDevice);

    // Initialize the result on the GPU to 0
    cudaMemset(d_result, 0, sizeof(int));

    // Define the number of threads and blocks for GPU
       ↪ calculation
    int threadsPerBlock = 256;
    int blocksPerGrid = (arraySize + threadsPerBlock - 1) /
       ↪ threadsPerBlock;

    // Measure GPU execution time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    // Launch the GPU kernel to calculate the sum
    sumArrayGPU<<<blocksPerGrid, threadsPerBlock>>>(d_array,
       ↪ d_result, arraySize);
    cudaEventRecord(stop);

    // Copy the result from GPU to CPU
    int result;
    cudaMemcpy(&result, d_result, sizeof(int),
       ↪ cudaMemcpyDeviceToHost);

    // Free GPU memory
    cudaFree(d_array);
    cudaFree(d_result);

    // CPU calculation
    int cpuResult = sumArrayCPU(array, arraySize);

    // Measure CPU execution time
    clock_t cpuStart = clock();
    int cpuSum = sumArrayCPU(array, arraySize);
    clock_t cpuEnd = clock();
    double cpuTime = (double)(cpuEnd - cpuStart) /
       ↪ CLOCKS_PER_SEC;

    std::cout << "GPU Result: " << result << std::endl;
    std::cout << "CPU Result: " << cpuResult << std::endl;
    std::cout << "GPU Execution Time: ";
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    std::cout << milliseconds << " ms" << std::endl;
    std::cout << "CPU Execution Time: " << cpuTime << " s" <<
       ↪ std::endl;

    return 0;
```

# 8 Lab 8: Using CUDA in Google Colab and Matrix Multiplication

- Learn to configure Google Colab for CUDA-enabled GPU usage and understand the steps to install and verify the CUDA toolkit in a Colab environment.
- Grasp the basics of matrix multiplication and understand the parallelizable nature of matrix operations.
- Learn how to write a simple CUDA program for matrix multiplication.
- Evaluate the performance gains achieved by using CUDA for matrix multiplication.

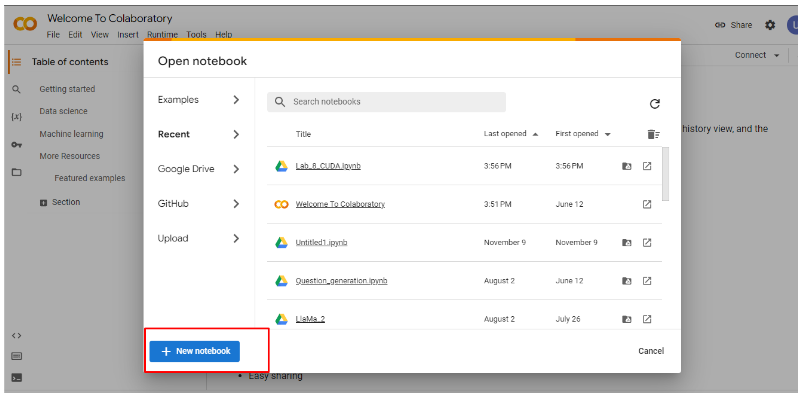
## Deliverables

- Step-by-step guide on configuring Google Colab for CUDA and GPU support.
- CUDA code for matrix multiplication with clear comments and explanations.
- Report showcasing the performance improvements achieved using CUDA in matrix multiplication.
- A Colab notebook containing examples and exercises for hands-on practice.

## 8.1 Setting up Environment on Google Colab

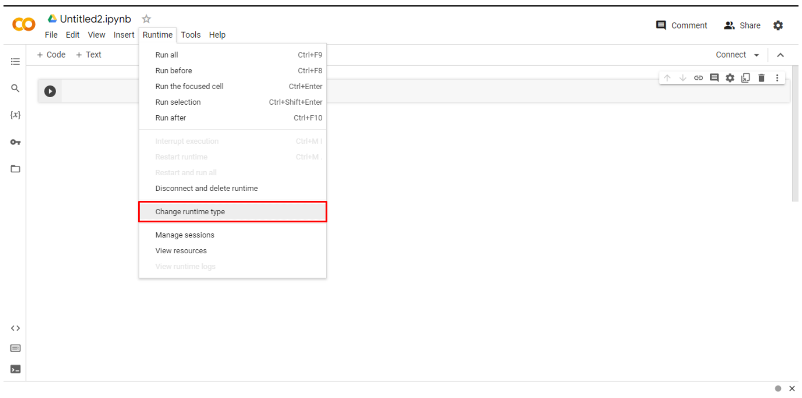
Google Colab is a free cloud service, and the most important feature that distinguishes Colab from other free cloud services is that Colab offers GPU and is completely free! With Colab, you can work on the GPU with CUDA C/C++ for free! CUDA code will not run on AMD CPU or Intel HD graphics unless you have NVIDIA hardware inside your machine. On Colab, you can take advantage of Nvidia GPU as well as being a fully functional Jupyter Notebook with pre-installed Tensorflow and some other ML/DL tools.

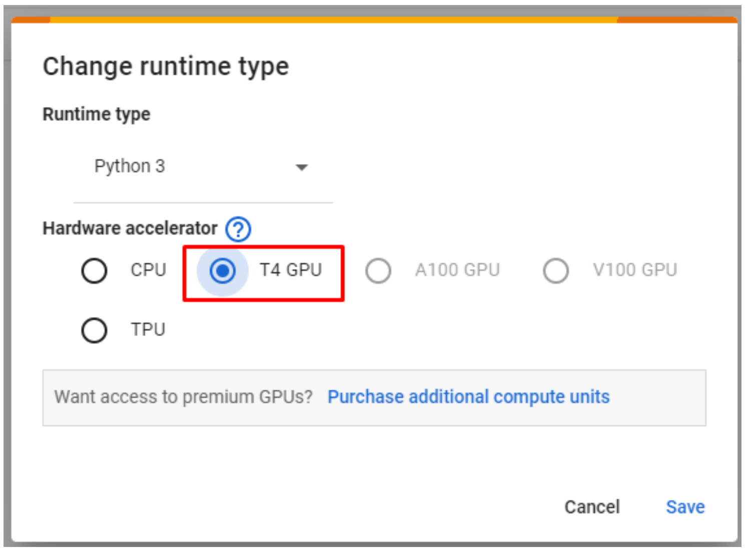
### 8.1.1 Go to https://colab.research.google.com in Browser and Click on New Notebook.



| Title | Last opened | First opened |
|---|---|---|
| Lab_8_CUDA.ipynb | 3:56 PM | 3:56 PM |
| Welcome To Colaboratory | 3:51 PM | June 12 |
| Untitled1.ipynb | November 9 | November 9 |
| Question_generation.ipynb | August 2 | June 12 |
| LlaMa_2 | August 2 | July 26 |

### 8.1.2 We must switch our runtime from CPU to GPU. Click on Runtime > Change runtime type > Hardware Accelerator > GPU > Save.





Runtime type: Python 3

Hardware accelerator: CPU [✓ T4 GPU] A100 GPU V100 GPU TPU

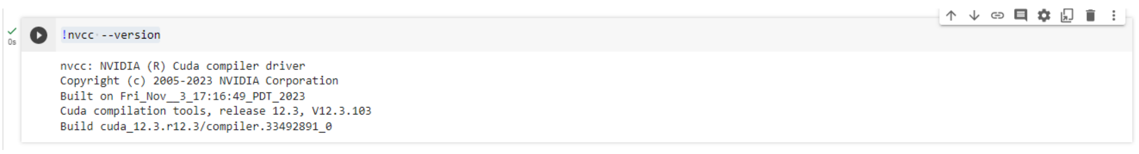
Want access to premium GPUs? Purchase additional compute units

### 8.1.3 Install CUDA

```
!wget https://developer.download.nvidia.com/compute/cuda/repos
   ↪ /ubuntu2204/x86_64/cuda-ubuntu2204.pin
!mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-repository
   ↪ -pin-600
!wget https://developer.download.nvidia.com/compute/cuda
   ↪ /12.3.1/local_installers/cuda-repo-ubuntu2204-12-3-
   ↪ local_12.3.1-545.23.08-1_amd64.deb
!dpkg -i cuda-repo-ubuntu2204-12-3-local_12.3.1-545.23.08-1
   ↪ _amd64.deb
!sudo cp /var/cuda-repo-ubuntu2204-12-3-local/cuda-*-keyring.
   ↪ gpg /usr/share/keyrings/
!apt-get update
!apt-get -y install cuda-toolkit-12-3
```

### 8.1.4 Now you can check your CUDA installation by running the command given below.

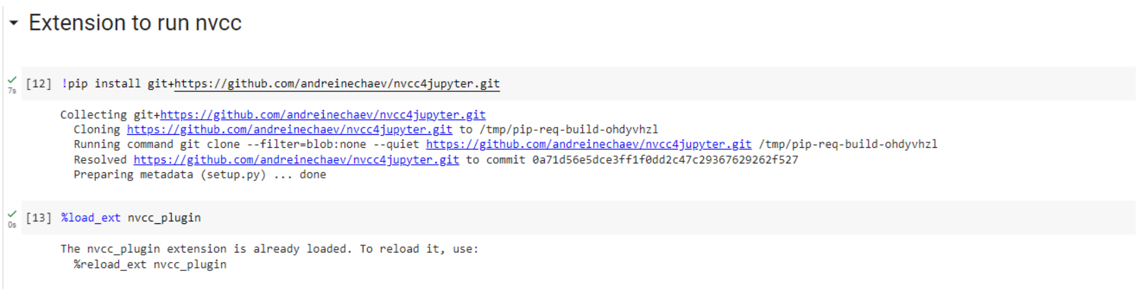
```
!nvcc --version
```



```
!nvcc --version

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Fri_Nov__3_17:16:49_PDT_2023
Cuda compilation tools, release 12.3, V12.3.103
Build cuda_12.3.r12.3/compiler.33492891_0
```

### 8.1.5 Run the given command to install a small extension to run nvcc from the Notebook cells.

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter
   ↪ .git
// To load the extension:
%load_ext nvcc_plugin
```

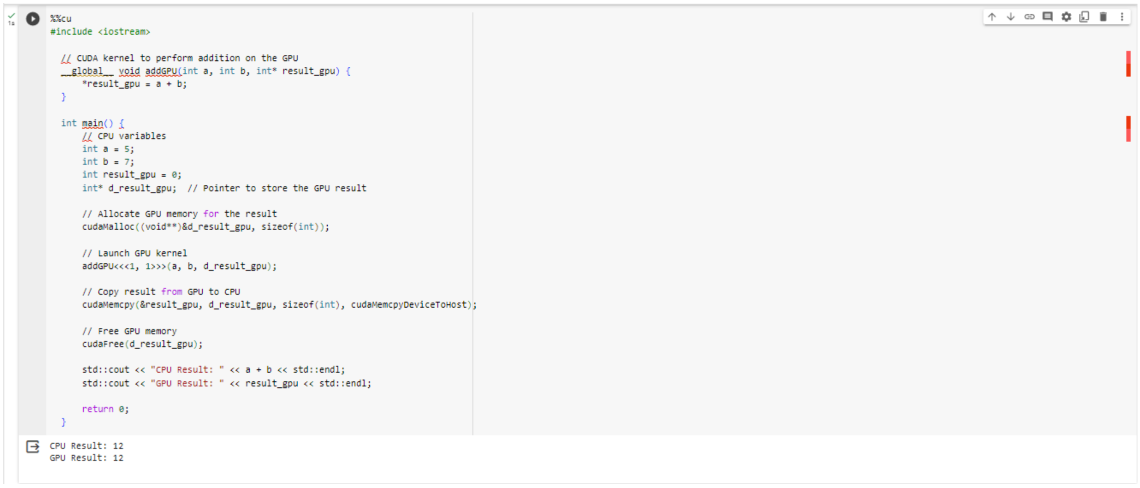


Extension to run nvcc

```
[12] !pip install git+https://github.com/andreinechaev/nvcc4jupyter.git

Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-ohdyvhzl
  Running command git clone --filter=blob:none --quiet https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-ohdyvhzl
  Resolved https://github.com/andreinechaev/nvcc4jupyter.git to commit 0a71d56e5dce3ff1f0dd2c47c29367629262f527
  Preparing metadata (setup.py) ... done

[13] %load_ext nvcc_plugin

The nvcc_plugin extension is already loaded. To reload it, use:
  %reload_ext nvcc_plugin
```

### 8.1.6 Execute the code given below to check if CUDA is working or not.

The code used in this snippet is from the previous lab, i.e., Example 1.



```
%%cu
#include <iostream>

  // CUDA kernel to perform addition on the GPU
  __global__ void addGPU(int a, int b, int* result_gpu) {
      *result_gpu = a + b;
  }

  int main() {
      // CPU variables
      int a = 5;
      int b = 7;
      int result_gpu = 0;
      int* d_result_gpu;  // Pointer to store the GPU result

      // Allocate GPU memory for the result
      cudaMalloc((void**)&d_result_gpu, sizeof(int));

      // Launch GPU kernel
      addGPU<<<1, 1>>>(a, b, d_result_gpu);

      // Copy result from GPU to CPU
      cudaMemcpy(&result_gpu, d_result_gpu, sizeof(int), cudaMemcpyDeviceToHost);

      // Free GPU memory
      cudaFree(d_result_gpu);

      std::cout << "CPU Result: " << a + b << std::endl;
      std::cout << "GPU Result: " << result_gpu << std::endl;

      return 0;
  }
```

```
CPU Result: 12
GPU Result: 12
```

## 8.2 Matrix Operations in CUDA

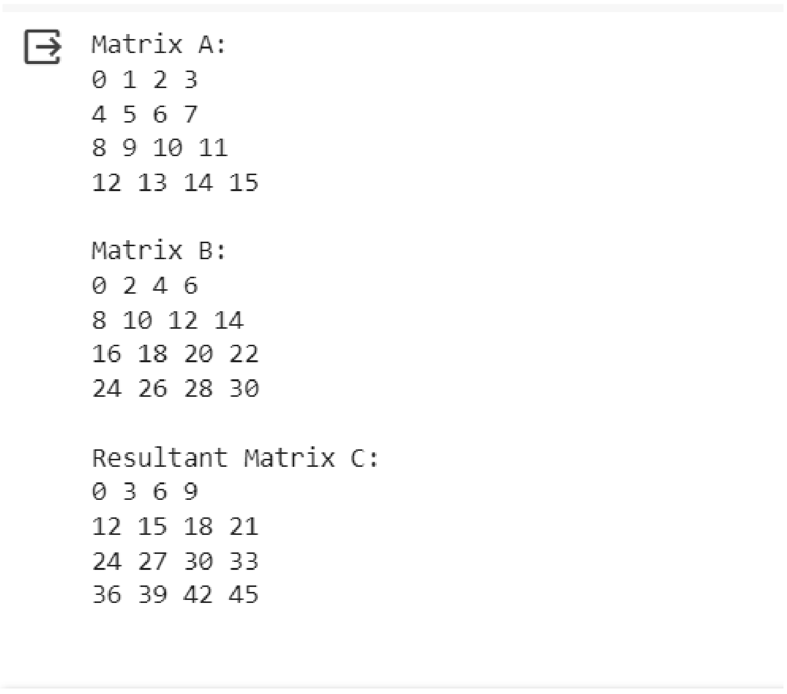
In the realm of high-performance computing, the use of Graphics Processing Units (GPUs) has become increasingly prevalent for accelerating complex computations. CUDA (Compute Unified Device Architecture), a parallel computing platform and programming model developed by NVIDIA, allows developers to harness the power of GPUs for general-purpose computing tasks. In this lab, we delve into matrix operations in CUDA, exploring how parallel processing can significantly enhance the efficiency of these fundamental mathematical operations.

**Matrix addition** is a fundamental operation that can be parallelized efficiently using CUDA. In this lab, we will design a CUDA kernel to perform matrix addition in parallel. The key is to distribute the workload across multiple threads, taking advantage of the parallel architecture of the GPU. We will discuss thread indexing, block configuration, and shared memory utilization to optimize performance.

### 8.2.1 Example Code:

The given C++ CUDA code implements matrix addition on the GPU using NVIDIA's CUDA parallel computing platform. The program begins by including necessary header files, defining the size of the square matrices (4x4 in this case), and declaring pointers for the host (CPU) and device (GPU) matrices. Memory is allocated on both the host and the device for input matrices ('h_a' and 'h_b') and the output matrix ('h_c'). The matrices are then initialized on the host, and their values are transferred to the GPU.

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>

const int N = 4;  // Size of the matrices (N x N)

// CUDA kernel to add two matrices
__global__ void matrixAdd(int *a, int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < n && j < n) {
        int index = i * n + j;
        c[index] = a[index] + b[index];
    }
}

int main() {
    int *h_a, *h_b, *h_c;  // Host matrices
    int *d_a, *d_b, *d_c;  // Device matrices

    // Allocate memory on the host
    h_a = new int[N * N];
    h_b = new int[N * N];
    h_c = new int[N * N];

    // Initialize matrices on the host
    for (int i = 0; i < N * N; ++i) {
        h_a[i] = i;
        h_b[i] = 2 * i;
    }

    // Allocate memory on the device
    cudaMalloc((void**)&d_a, N * N * sizeof(int));
    cudaMalloc((void**)&d_b, N * N * sizeof(int));
    cudaMalloc((void**)&d_c, N * N * sizeof(int));

    // Copy matrices from host to device
    cudaMemcpy(d_a, h_a, N * N * sizeof(int),
       ↪ cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, N * N * sizeof(int),
       ↪ cudaMemcpyHostToDevice);

    // Define block and grid sizes
    dim3 blockSize(16, 16);
    dim3 gridSize(ceil(static_cast<float>(N) / blockSize.x),
       ↪ ceil(static_cast<float>(N) / blockSize.y));

    // Launch the kernel
    matrixAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, N);

    // Copy the result from device to host
    cudaMemcpy(h_c, d_c, N * N * sizeof(int),
       ↪ cudaMemcpyDeviceToHost);
    // Print the result
    std::cout << "Matrix A:\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            std::cout << h_a[i * N + j] << " ";
        }
        std::cout << "\n";
    }

    std::cout << "\nMatrix B:\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            std::cout << h_b[i * N + j] << " ";
        }
        std::cout << "\n";
    }

    std::cout << "\nResultant Matrix C:\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            std::cout << h_c[i * N + j] << " ";
        }
        std::cout << "\n";
    }

    // Free memory on the device
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Free memory on the host
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;

    return 0;
}
```



```
Matrix A:
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15

Matrix B:
0 2 4 6
8 10 12 14
16 18 20 22
24 26 28 30

Resultant Matrix C:
0 3 6 9
12 15 18 21
24 27 30 33
36 39 42 45
```

The CUDA kernel function ‘matrixAdd‘ is responsible for performing element-wise addition of the matrices. It is executed in parallel by multiple threads organized into blocks and grids. The block and grid sizes are defined, and the kernel is launched on the GPU.

After the kernel execution, the result is copied back from the device to the host. The program concludes by printing the original matrices and the resultant matrix on the console for verification. Finally, memory allocated on both the host and the device is freed. This code serves as a basic example of parallelizing matrix operations using CUDA, showcasing the memory management, kernel launching, and data transfer between the host and the GPU. The block and grid configuration enables efficient parallelization of the matrix addition operation, enhancing performance compared to a sequential CPU implementation for large matrices.

### 8.2.2 Example 2: Row vector multiplication with column vector

The CUDA C++ program takes user input for the sizes and values of a row vector and a column vector, checks if the number of columns in the row vector is equal to the number of rows in the column vector, and then performs matrix multiplication on the GPU. The program dynamically allocates memory for the vectors on both the host and the device, initializes the vectors with user-provided values, launches a CUDA kernel to multiply the vectors element-wise, and prints the resulting scalar value. This version of the code ensures that the resultant vector is a single scalar value (1x1), reflecting the multiplication of a row vector with a column vector, and frees allocated memory on both the host and the device. The program incorporates error checks to handle invalid input sizes or matrix dimensions.

```
#include <iostream>
#include <cuda_runtime.h>
```

```
// CUDA kernel to multiply a row vector with a column vector
__global__ void vectorMultiply(int *rowVector, int *colVector,
    ↪  int *result, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i == 0) {
        result[0] = 0;  // Initialize the result to 0
        for (int j = 0; j < n; ++j) {
            result[0] += rowVector[j] * colVector[j];
        }
    }
}

int main() {
    int numRows, numCols;

    // Get the size of the row vector from the user
    std::cout << "Enter the number of rows in the row vector:
        ↪ ";
    std::cin >> numRows;

    if (numRows <= 0) {
        std::cerr << "Error: The number of rows must be
            ↪ greater than 0.\n";
        return 1;
    }

    // Get the size of the column vector from the user
    std::cout << "Enter the number of columns in the column
        ↪ vector: ";
    std::cin >> numCols;

    if (numCols <= 0) {
        std::cerr << "Error: The number of columns must be
            ↪ greater than 0.\n";
        return 1;
    }

    if (numCols != numRows) {
        std::cerr << "Error: The number of columns in the row
            ↪ vector must be equal to the number of rows in
            ↪ the column vector.\n";
        return 1;
    }
```

```
int *h_rowVector, *h_colVector, *h_result;  // Host
   ↪ vectors
int *d_rowVector, *d_colVector, *d_result;  // Device
   ↪ vectors

// Allocate memory on the host
h_rowVector = new int[numRows];
h_colVector = new int[numCols];
h_result = new int[1]; // Resultant vector size is set to
   ↪ 1x1

// Get values for the row vector from the user
std::cout << "Enter values for the row vector:\n";
for (int i = 0; i < numRows; ++i) {
    std::cout << "Value " << i + 1 << ": ";
    std::cin >> h_rowVector[i];
}

// Get values for the column vector from the user
std::cout << "Enter values for the column vector:\n";
for (int i = 0; i < numCols; ++i) {
    std::cout << "Value " << i + 1 << ": ";
    std::cin >> h_colVector[i];
}

// Allocate memory on the device
cudaMalloc((void**)&d_rowVector, numRows * sizeof(int));
cudaMalloc((void**)&d_colVector, numCols * sizeof(int));
cudaMalloc((void**)&d_result, 1 * sizeof(int)); //
   ↪ Resultant vector size is set to 1x1

// Copy vectors from host to device
cudaMemcpy(d_rowVector, h_rowVector, numRows * sizeof(int)
   ↪ , cudaMemcpyHostToDevice);
cudaMemcpy(d_colVector, h_colVector, numCols * sizeof(int)
   ↪ , cudaMemcpyHostToDevice);

// Define block and grid sizes
dim3 blockSize(256);
dim3 gridSize((1 + blockSize.x - 1) / blockSize.x); //
   ↪ Grid size set to 1

// Launch the kernel
vectorMultiply<<<gridSize, blockSize>>>(d_rowVector,
   ↪ d_colVector, d_result, numRows);
```

```
    // Copy the result from device to host
    cudaMemcpy(h_result, d_result, 1 * sizeof(int),
       ↪ cudaMemcpyDeviceToHost);

    // Print the result
    std::cout << "Resultant Scalar: " << h_result[0] << "\n";

    // Free memory on the device
    cudaFree(d_rowVector);
    cudaFree(d_colVector);
    cudaFree(d_result);

    // Free memory on the host
    delete[] h_rowVector;
    delete[] h_colVector;
    delete[] h_result;

    return 0;
}
```

# 9 Lab 9: Use of GPU memory

## Objectives

- Explore the utilization of global memory, constant memory, and shared memory in parallel algorithms.
- Understand the impact of different memory types on performance and identify optimization opportunities.
- Familiarize oneself with CUDA kernel design and execution for parallel processing on NVIDIA GPUs.

## Deliverables

- Complete and functional CUDA C++ code for vector addition with distinct kernels utilizing global, constant, and shared memory.
- Documentation detailing the rationale and methodology behind memory type selection for each kernel.
- Performance analysis report comparing the execution times and efficiency of the different memory implementations.

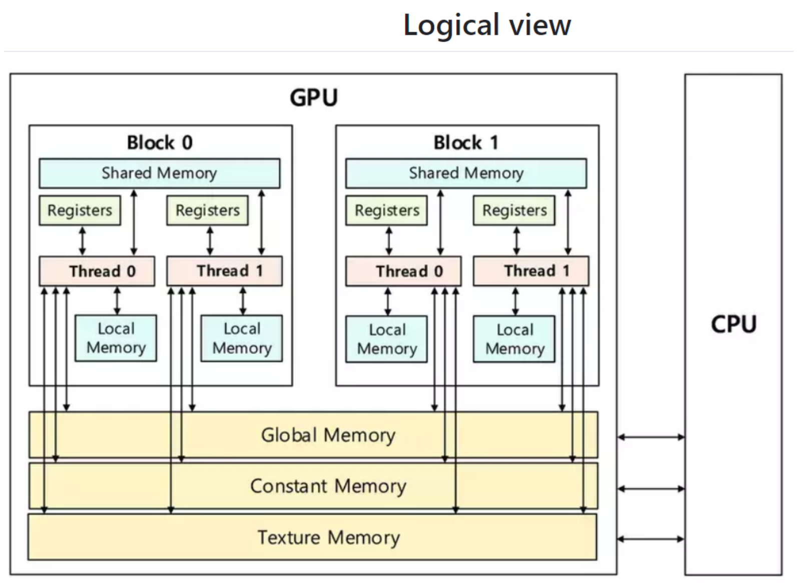
## 9.1 GPU memory

Before delving into the various memory types within a GPU, it's important to understand that when we talk about memory, we generally categorize it into two main types: physical memory and logical memory.

**Physical Memory:** This refers to the actual hardware memory in a computer. It includes components such as RAM modules and storage devices like hard drives (HDD/SSD). Physical memory is where data and programs are stored directly and can be accessed quickly by the processor.

**Logical Memory (Virtual Memory):** This is the address space that the operating system and programs can access. Logical memory doesn't necessarily have a direct one-to-one correspondence with physical memory. The operating system typically manages the mapping between logical addresses and physical addresses. This management helps allocate and manage memory for programs running on the system.

You can understand it in a simple way: when we code, we interact with logical memory, and once the code is finished, the data located in logical memory will be mapped to physical memory (meaning the computer will operate in physical memory). Now that we have a foundational understanding of memory, let's explore the specific memory types within a GPU and their purposes.



As mentioned, Blocks and Threads are logical concepts, and due to the SIMT mechanism, it's important to understand how Threads and Blocks are distributed and managed within the logical memory of the GPU. Here, we have a familiar concept known as scope, which plays a crucial role in understanding how resources like Threads and Blocks are allocated and managed within the logical memory of the GPU.

- **Local Memory:** Each Thread can use its own local memory, where it can store temporary variables. This has the smallest scope and is dedicated to each individual Thread.
- **Shared Memory:** Threads within the same Block can share data through shared memory. This allows Threads within the same Block to communicate and access data faster compared to accessing global memory.
- **Global Memory:** Global memory in GPU programming is a fundamental storage space accessible by all threads in all blocks. It serves as the primary repository for data that needs to be shared across threads. However, it's crucial to be mindful of its relatively slower access speed compared to other memory types. To optimize performance, developers should minimize global memory accesses and adopt coalesced memory access patterns. Coalescing, which involves arranging memory accesses in a way that maximizes throughput, can significantly enhance the efficiency of global memory usage.
- **Texture Memory and Constant Memory:** Constant memory provides a read-only, cached space that is shared among all threads within a block. This type of memory is ideal for storing values that remain constant across multiple threads, offering fast and efficient access. To harness the benefits of constant memory, developers should focus on using it for read-only data and ensuring that access patterns are aligned and coalesced. It's essential to be mindful of the limited capacity of the constant cache and avoid exceeding it to maintain optimal performance.

- **Shared memory** is a high-speed, block-specific memory that is shared among threads within the same block. It is explicitly managed by the programmer and serves as a space for creating local copies of data that multiple threads need to access and modify concurrently. To maximize the advantages of shared memory, developers should efficiently use it to reduce reliance on global memory accesses. Strategies include minimizing bank conflicts, which occur when multiple threads access the same memory bank simultaneously, and exploiting shared memory for data reuse and inter-thread communication.

## 9.2 Code Example:

This CUDA C++ code implements a vector addition program, showcasing three different kernels for parallel execution on a GPU. The first kernel ('vectorAddGlobal') utilizes global memory for vector addition, with each thread handling an element of the vectors. The second kernel ('vectorAddConstant') demonstrates the use of constant memory for one of the vectors, enhancing read-only data access. The third kernel ('vectorAddShared') employs shared memory to cache portions of both vectors, reducing global memory accesses and promoting inter-thread communication within a block. The example initializes host vectors, allocates and copies data to the GPU device, performs vector addition using each kernel, and finally retrieves the results back to the host for verification. While this example may not exhibit significant performance gains due to the simplicity of vector addition, the principles illustrated become crucial for optimizing more complex GPU algorithms with intricate memory access patterns.

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>

const int N = 1024; // Size of the vectors

// Kernel for vector addition using global memory
__global__ void vectorAddGlobal(float* a, float* b, float* c)
   ↪ {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        c[i] = a[i] + b[i];
    }
}

// Kernel for vector addition using constant memory
__constant__ float constVector[N]; // Constant memory
   ↪ declaration

__global__ void vectorAddConstant(float* a, float* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        c[i] = a[i] + constVector[i];
    }
}

// Kernel for vector addition using shared memory
__global__ void vectorAddShared(float* a, float* b, float* c)
   ↪ {
    __shared__ float sharedA[N];
    __shared__ float sharedB[N];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        sharedA[i] = a[i];
        sharedB[i] = b[i];
        __syncthreads(); // Ensure all threads have finished
           ↪ copying

        c[i] = sharedA[i] + sharedB[i];
    }
}

int main() {
    // Host vectors
    float h_a[N], h_b[N], h_c[N];

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

    // Allocate device memory
    float *d_a, *d_b, *d_c;
    cudaMalloc((void**)&d_a, N * sizeof(float));
    cudaMalloc((void**)&d_b, N * sizeof(float));
    cudaMalloc((void**)&d_c, N * sizeof(float));

    // Copy host data to device
    cudaMemcpy(d_a, h_a, N * sizeof(float),
       ↪ cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, N * sizeof(float),
       ↪ cudaMemcpyHostToDevice);

    // Global memory version
    vectorAddGlobal<<<(N + 255) / 256, 256>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(h_c, d_c, N * sizeof(float),
       ↪ cudaMemcpyDeviceToHost);

    // Check results...

    // Reset device memory for the next kernel
    cudaMemset(d_c, 0, N * sizeof(float));

    // Constant memory version
    cudaMemcpyToSymbol(constVector, h_b, N * sizeof(float));
       ↪ // Copy data to constant memory
    vectorAddConstant<<<(N + 255) / 256, 256>>>(d_a, d_c);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, N * sizeof(float),
       ↪ cudaMemcpyDeviceToHost);

    // Reset device memory for the next kernel
    cudaMemset(d_c, 0, N * sizeof(float));

    // Shared memory version
    vectorAddShared<<<(N + 255) / 256, 256>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, N * sizeof(float),
       ↪ cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

# 10 Lab 10: Introduction to MPI (Message Passing Interface

## Objectives

- Explain the structure of MPI programs.
- Introduce MPI environment management routines.
- Provide practical examples of MPI usage.
- Provide examples of communication between MPI processes.

## Deliverables

- Understanding of the structure of MPI programs.
- Proficiency in using MPI environment management routines.
- Successful implementation of MPI examples.
- Proficiency in implementing communication between MPI processes.

In order to reduce the execution time, work is carried out in parallel. Two types of parallel programming are:

- Explicit parallel programming
- Implicit parallel programming

**Explicit parallel programming** – These are languages where the user has full control and has to provide all the details explicitly. Compiler effort is minimal.
**Implicit parallel programming** – These are sequential languages where the compiler has full responsibility for extracting the parallelism in the program.
**Parallel Programming Models:**

- Message Passing Programming
- Shared Memory Programming

**Message Passing Programming:**

- In message passing programming, programmers view their programs (Applications) as a collection of co-operating processes with private (local) variables.
- The only way for an application to share data among processors is for programmer to explicitly code commands to move data from one processor to another.

**Message Passing Libraries:** There are two message passing libraries available. They are:

- PVM – Parallel Virtual Machine
- MPI – Message Passing Interface. It is a set of parallel APIs which can be used with languages such as C and FORTRAN.

## 10.1 Structure of MPI Proggram

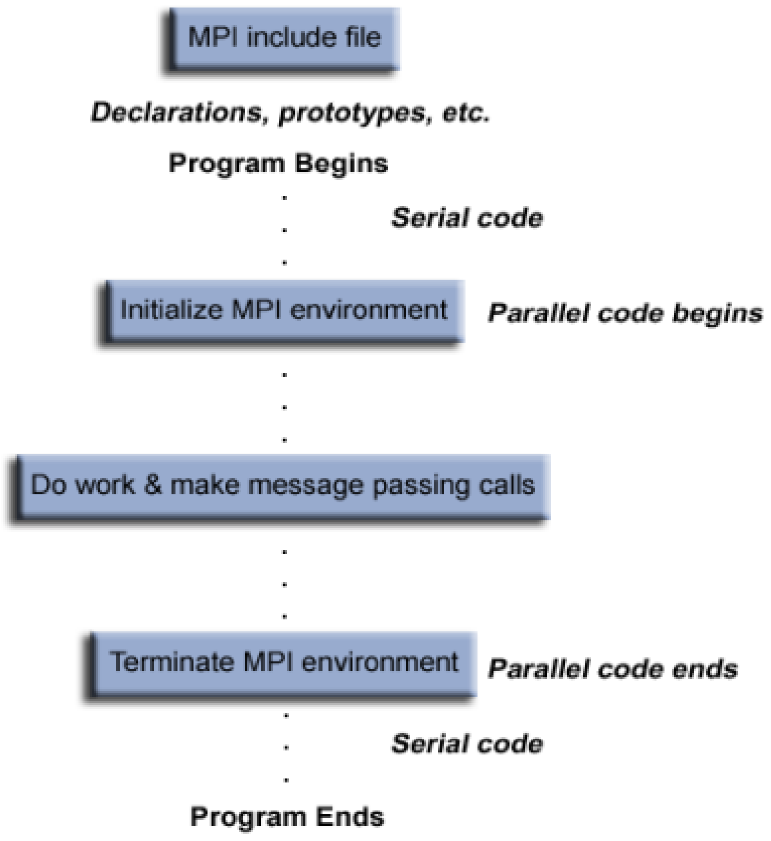

MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

Figure 20: Structure of MPI Proggram

**Communicators and Groups:**

- MPI assumes static processes.
- All the processes are created when the program is loaded.
- No process can be created or terminated during program execution.
- There is a default process group consisting of all such processes identified by MPI_COMM_WORLD.

## 10.2 MPI Environment Management Routines

**MPI_Init:** Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
(&argc,&argv);
```

**MPI_Comm_size:** Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI_COMM_WORLD.

```
_size(Comm,&size);
```

**MPI_Comm_rank:** Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size-1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a process ID.

```
_rank(Comm,&rank);
```

**MPI_Finalize:** Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

```
lize();
```

## 10.3 Example

A simple MPI program to print total number of process and rank of each process.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank,size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("My rank is %d in total %d process",rank,size);
    MPI_Finalize();
    return 0;
}
```

## 10.4 Communication between MPI processes

It is important to observe that when a program runs with MPI, all processes use the same compiled binary, and hence all processes are running the exact same code. What in an MPI distinguishes a parallel program running on P processors from the serial version of the code running on P processors? Two things distinguish the parallel program:

- Each process uses its process rank to determine what part of the algorithm instructions are meant for it.
- Processes communicate with each other in order to accomplish the final task.

Even though each process receives an identical copy of the instructions to be executed, this does not imply that all processes will execute the same instructions. Because each process is able to obtain its process rank (using MPI_Comm_rank).It can determine which part of the code it is supposed to run. This is accomplished through the use of IF statements. Code that is meant to be run by one particular process should be enclosed within an IF statement, which verifies the process identification number of the process. If the code is not placed with in IF statements specific to a particular id, then the code will be executed by all processes.
The second point, communicating between processes; MPI communication can be summed up in the concept of sending and receiving messages. Sending and receiving is done with the following two functions: MPI Send and MPI Recv.
**MPI_Send**

```
Send( void* message /* in */, int count /* in */,
type datatype /* in */, int dest /* in */,
/* in */, MPI Comm comm /* in */ )
```

**MPI_Recv**

```
Recv( void* message /* out */, int count /* in */,
type datatype /* in */, int source /* in */,
/* in */, MPI Comm comm /* in */,
us* status /* out */)
```

Understanding the Argument Lists

- **message** - starting address of the send/recv buffer.
- **count** - number of elements in the send/recv buffer.
- **datatype** - data type of the elements in the send buffer.
- **source** - process rank to send the data.
- **dest** - process rank to receive the data.
- **tag** - message tag.
- **comm** - communicator.
- **status** - status object.

## 10.5 Examples

**Example 1: Sending a value to another process**

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

int main(int argc, char const *argv[])
{
        MPI_Init(NULL, NULL);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if (world_size < 2)
        {
                fprintf(stderr, "World size must be greater
                   ↪ than 1 for %s\n", argv[0]);
                MPI_Abort(MPI_COMM_WORLD,1);
        }

        int number;

        if (world_rank ==0)
        {
                number = -1;
                MPI_Send(&number, 1, MPI_INT, 1, 0,
                   ↪ MPI_COMM_WORLD);
        }
        else if (world_rank==1) {
                MPI_Recv(&number, 1, MPI_INT, 0, 0,
                   ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                printf("Process 1 recieved number %d from
                   ↪ process 0\n", number );
        }
        MPI_Finalize();

        return 0;
}
```

**Example 2: ping pong code** This program processes use MPI_Send and MPI_Recv to continually bounce messages off of each other until they decide to stop.

```
lude <mpi.h>
```

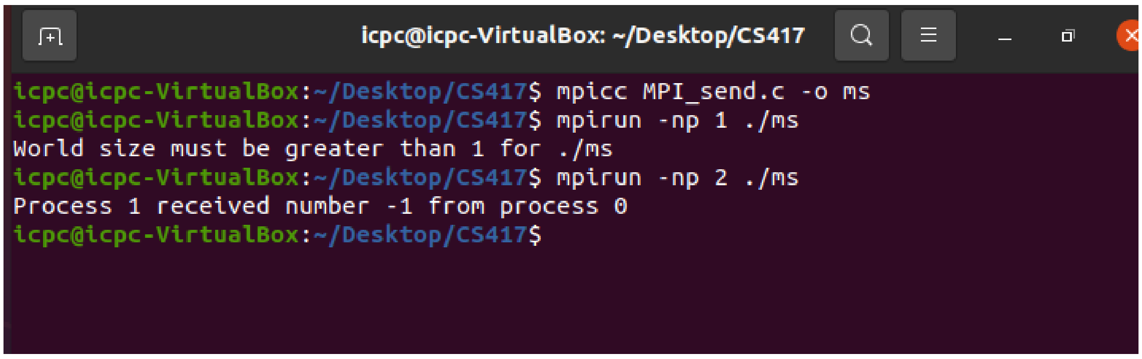

```
icpc@icpc-VirtualBox:~/Desktop/CS417$ mpicc MPI_send.c -o ms
icpc@icpc-VirtualBox:~/Desktop/CS417$ mpirun -np 1 ./ms
World size must be greater than 1 for ./ms
icpc@icpc-VirtualBox:~/Desktop/CS417$ mpirun -np 2 ./ms
Process 1 received number -1 from process 0
icpc@icpc-VirtualBox:~/Desktop/CS417$
```

Figure 21: Output

```
lude <stdio.h>
lude <stdlib.h>

main(int argc, char const *argv[])

const int PING_PONG_LIMIT = 10;
MPI_Init(NULL, NULL);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

if (world_size != 2)
{
    fprintf(stderr, "World size must be must be 2 for %s\n",
       ↪ argv[0]);
    MPI_Abort(MPI_COMM_WORLD,1);
}

int ping_pomg_count =0;
int partner_rank= (world_rank+1)%2;
while (ping_pomg_count<PING_PONG_LIMIT)
{
    if (world_rank==ping_pomg_count%2)
    {
        ping_pomg_count++;
        MPI_SEND(&ping_pomg_count, 1, MPI_INT, partner_rank,
           ↪ 0, MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pomg_count: %d to
           ↪  %d \n", world_rank, ping_pomg_count,
           ↪ partner_rank);
    } else {
        MPI_Recv(&ping_pomg_count, 1, MPI_INT, partner_rank,
           ↪ 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d recieving ping_pomg_count: %d to %d \n",
           ↪ world_rank, ping_pomg_count,partner_rank);
    }
}
MPI_Finalize();

return 0;
```

# 11 Lab 11: Collective communication in MPI

## Objectives

- Explore collective communication patterns in MPI.
- Optimize collective communication performance in MPI.
- Benchmark and evaluate optimized strategies.

## Deliverables

- Collective communication analysis report.
- Optimized collective communication strategies documentation.
- Benchmarking results and analysis report.

## 11.1 MPI_SCATTER, MPI_GATHER

**Collective Communication routines:**
When all processes in a group participate in a global communication operation, the resulting communication is called collective communication.
**MPI_Bcast:**

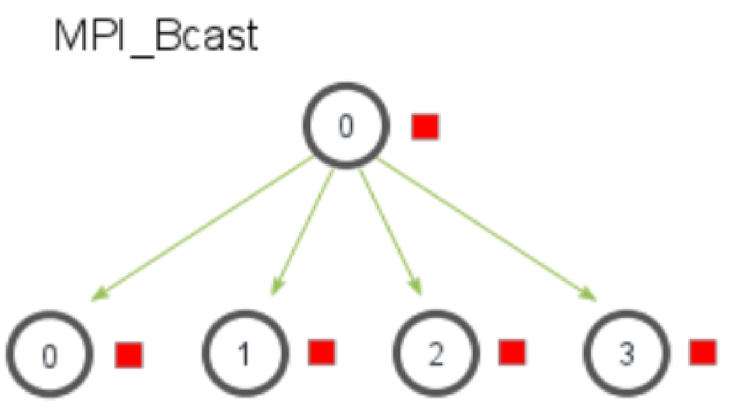
```
MPI_Bcast (Address, Count, Datatype, Root, Comm);
```

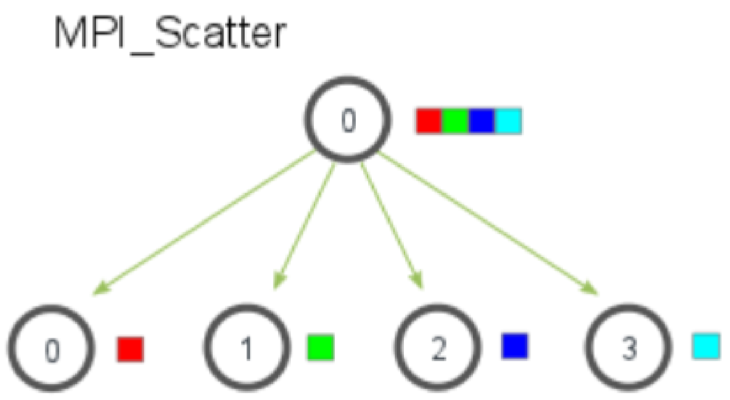
The process ranked Root sends the same message whose content is identified by the triple (Address,Count,Datatype) to all processes(including itself) in the communicator Comm.

**MPI_Scatter:** MPI_Scatter is a collective routine that is very similar to MPI_Bcast. MPI_Scatter involves a designated root process sending data to all processes in a communicator. The primary difference between MPI_Bcast and MPI_Scatter is small but important. MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of data to different processes. Check out the illustration below for further clarification.

```
MPI_Scatter( SendBuff, Sendcount, SendDatatype, RecvBuff,
t, RecvDatatype, Root, Comm);
```

Ensures that the Root process sends out personalized messages, which are in rank order in its send buffer, to all the N processes (including itself).
**MPI_Gather:** PI_Gather is the inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching. Below is a simple illustration of this algorithm. Similar to MPI_Scatter, MPI_Gather takes elements from each process and gathers them to the root process. The elements

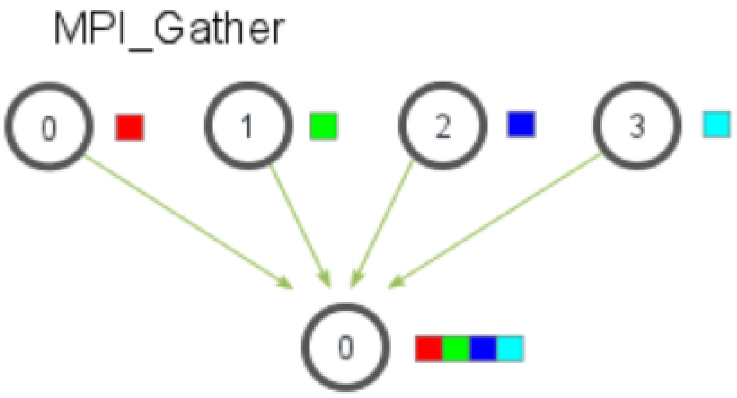




are ordered by the rank of the process from which they were received. The function prototype for MPI_Gather is identical to that of MPI_Scatter.

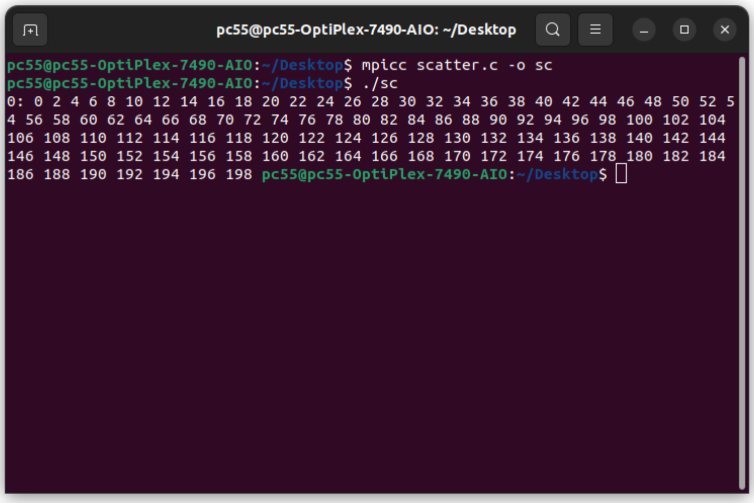
```
MPI_Gather( SendAddress, Sendcount, SendDatatype,
ess, RecvCount, RecvDatatype, Root, Comm);
```

The root process receives a personalized message from all N processes. These N received messages are concatenated in rank order and stored in the receive buffer of the root process.



**Example: MPI Scatter**

```
lude <mpi.h>
lude <stdio.h>
lude <stdlib.h>

main(int argc, char const *argv[])

    int myRank;
    int totalProcesses;
    MPI_Init(NULL, NULL);

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int fullSize =100, reducedSize = fullSize/totalProcesses;
    int *buffer1 = malloc(size(int)*fullSize);
    int *buffer2 = malloc(size(int)*fullSize);
    int *buf = malloc(sizeof(int)*reducedSize);
    i = 0;

    if(myRank == 0)
    {
            for (i = 0; i < fullSize; i++)
            {
                    buffer1[i] = i;
            }
    }

    MPI_Scatter(buffer1, reducedSize, MPI_INT, buf,
       ↪ reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
    for (int i = 0; i < reducedSize; ++i)
    {
            buf[i]*=2;
    }

    MPI_Gather(buf, reducedSize, MPI_INT, buffer2, reducedSize
       ↪ , MPI_INT, 0, MPI_COMM_WORLD);
    if (myRank==0)
    {
            printf("%d\n", myRank );
            for (int i = 0; i < fullSize; ++i)
            {
                    printf("%d\n", buffer2[i]);
            }
    }
    MPI_Finalize();

    return 0;
```



```
pc55@pc55-OptiPlex-7490-AIO:~/Desktop$ mpicc scatter.c -o sc
pc55@pc55-OptiPlex-7490-AIO:~/Desktop$ ./sc
0: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 5
4 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104
106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144
146 148 150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184
186 188 190 192 194 196 198 pc55@pc55-OptiPlex-7490-AIO:~/Desktop$
```

Figure 22: Output

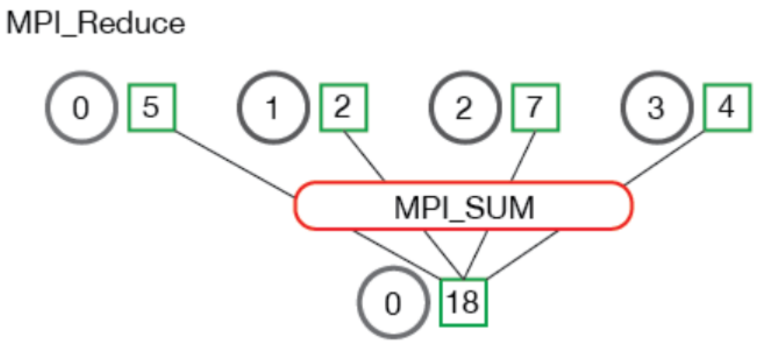
## 11.2 MPI_REDUCE

Reduce is a classic concept from functional programming. Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. For example, let's say we have a list of numbers [1, 2, 3, 4, 5]. Reducing this list of numbers with the sum function would produce sum([1, 2, 3, 4, 5]) = 15. Similarly, the multiplication reduction would yield multiply([1, 2, 3, 4, 5]) = 120. As you might have imagined, it can be very cumbersome to apply reduction functions across a set of distributed numbers. Along with that, it is difficult to efficiently program non-commutative reductions, i.e. reductions that must occur in a set order. Luckily, MPI has a handy function called MPI_Reduce that will handle almost all of the common reductions that a programmer needs to do in a parallel application.

**MPI_REDUCE** Similar to MPI_Gather, MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for MPI_Reduce looks like this:

```
MPI_Reduce (SendAddress, RecvAddress, Count, Datatype,
, Comm);
```

Below is an illustration of the communication pattern of MPI_Reduce. The send_data

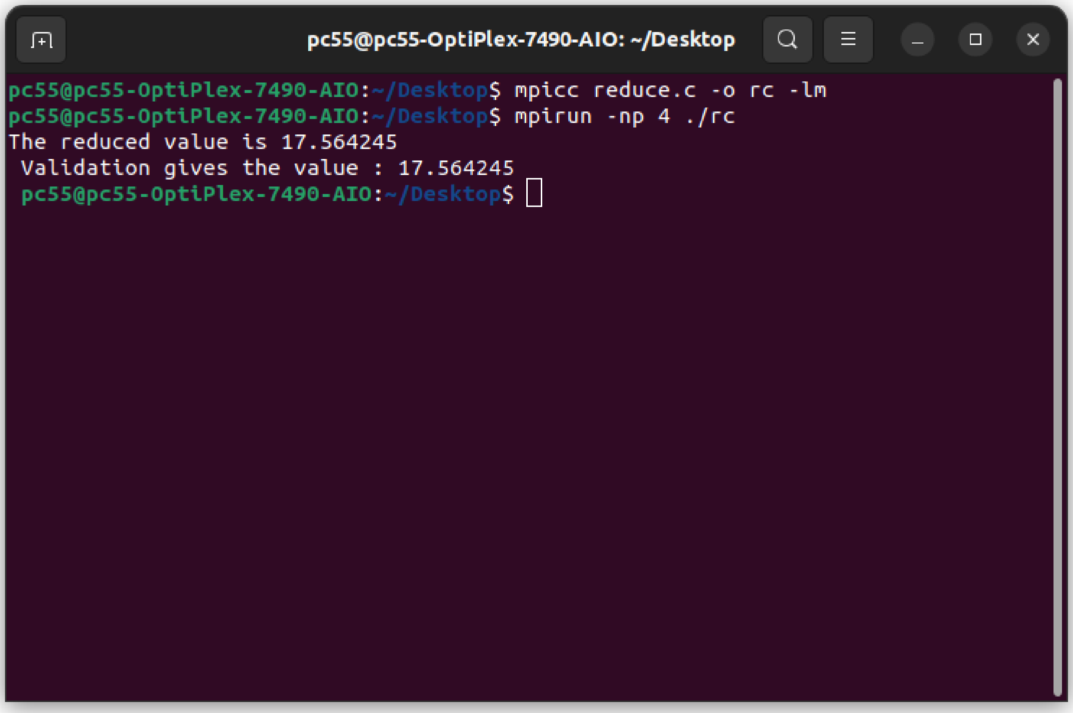


parameter is an array of elements of type datatype that each process wants to reduce. The recv_data is only relevant on the process with a rank of root. The recv_data array contains the reduced result and has a size of sizeof(datatype) count. The op parameter is the operation that you wish to apply to your data. MPI contains a set of common reduction operations that can be used. Although custom reduction operations can be defined, it is beyond the scope of this lesson. The reduction operations defined by MPI include:

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical and across the elements.
- MPI_LOR - Performs a logical or across the elements.
- MPI_BAND - Performs a bitwise and across the bits of the elements.
- MPI_BOR - Performs a bitwise or across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

**Example: MPI_Reduce:** Consider a system where you have processes. The goal of the game is to compute the dot product of two vectors in parallel. Now the dot product of two vectors and, for those who forgot, is the following operation : As you can imagine, this is highly parallelizable. If you have processes, each process can compute the intermediate value . Then, the program needs to find a way to sum all of these values. This is where the reduction comes into play. We can ask MPI to sum all those value and store them either on only one process (for instance process 0) or to redistribute the value to every process. Here is how we would do it in C++ :

```
lude <mpi.h>
lude <stdio.h>
lude <stdlib.h>

int main(int argc, char const *argv[])

    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    float u_i = rank*rank;
    float v_i = log(rank+1.0);

    float tmp = u_i * v_i;

    float result;

    MPI_Reduce(&tmp, &result, 1, MPI_FLOAT, MPI_SUM, 0,
       ↪ MPI_COMM_WORLD);

    if (rank==0)
    {
            printf("The reduced value is %f \n", result);

            float validation - 0.0f;
            for (int i = 0; i < size; ++i)
            {
                    validation += i*i +log(i+1.0f);
            }
            printf("Validation gives the value: %f \n",
               ↪ validation);
    }

    MPI_Finalize();
    return 0;
```



```
pc55@pc55-OptiPlex-7490-AIO:~/Desktop$ mpicc reduce.c -o rc -lm
pc55@pc55-OptiPlex-7490-AIO:~/Desktop$ mpirun -np 4 ./rc
The reduced value is 17.564245
 Validation gives the value : 17.564245
 pc55@pc55-OptiPlex-7490-AIO:~/Desktop$ 
```

Figure 23: Output