Hands-on Activity 13.1	
Parallel Algorithms and Multithreading	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/02/25
Section: CPE 010-CPE21S4	**Date Submitted:** 11/02/25
Name(s): KERWIN JAN B. CATUNGAL	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

Procedure:

ILO A & ILO B:

PART1:

```cpp
#include <iostream>
#include <string>

void print(int n, const std::string &str) {
    std::cout << "Printing integer: " << n << std::endl;
    std::cout << "Printing string: " << str << std::endl;
}

int main() {
    print(10, "T.I.P.");
    return 0;
}
```

OUTPUT:

```
Printing integer: 10
Printing string: T.I.P.

--------------------------------
Process exited after 0.2752 seconds with return value 0
Press any key to continue . . .
```

Analysis:

After doing this I focused on how one thread operates in C++. With the program running one task at a time in a sequential order it was straightforward since there was no overlap or complication between tasks. However I also realized that it can be slower if there are many tasks to handle so this part helped me understand how a norma, one threaded process behaves before learning concurrency.

PART2:

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <string>
#include <mutex>

std::mutex cout_mutex;

void print(int n, const std::string &str) {
    std::string msg = std::to_string(n) + ". " + str;

    std::lock_guard<std::mutex> guard(cout_mutex);
    std::cout << msg << std::endl;
}

int main() {
    std::vector<std::string> s = {
        "T.I.P.",
        "Competent",
        "Computer",
        "Engineers"
    };

    std::vector<std::thread> threads;

    for (int i = 0; i < s.size(); i++) {
        threads.emplace_back(print, i, s[i]);
    }

    for (auto &th : threads) {
        th.join();
    }

    return 0;
```

OUTPUT:

```
1. Competent
0. T.I.P.
2. Computer
3. Engineers

--------------------------------
Process exited after 0.3343 seconds with return value 0
Press any key to continue . . .
```

Analysis:

In the multi-threaded version, I saw how multiple tasks can run at the same time. It was interesting to see how the outputs do not always come in sequential order which illustrates the independence of threads. I also saw improved execution time because of the utilization of more CPU cores which is ref favorable for performance. However I saw that multithreaded execution is more unpredictable than the single thread version which can complicate control of the system.

B. Answers to Supplementary Activity

PART A:

Demonstrate an understanding of parallelism, concurrency, and multithreading in C++ by answering the given questions. Use of supplementary materials to support answers must be cited as reference.

Questions:

1. Write a definition of multithreading and its advantages/disadvantages.

Multithreading refers to executing stacks of I/O operations of a single program concurrently to complete several tasks simultaneously. Since the CPU is used efficiently and time is not wasted keeping the CPU idle the speed and performance of a program increases. One benefit of multithreading is the ability to perform several tasks simultaneously such as updating the interface while processing data. Multithreading problems can however, be difficult to debug because the threads can be executed in any order.

2. Rationalize the use of multithreading by providing at least 3 use-cases.

Multithreading is beneficial when programs need to perform multiple operations simultaneously. For instance, web servers manage user requests within separate threads simultaneously processing multiple requests. In games one thread manages the graphics while another manages player input or sound thereby allowing seamless gameplay. It is also used in file downloading or processing applications to allow the program to still perform other background tasks. These different applications show that multithreading improves the speed and responsiveness of programs.

3. Differentiate between parallelism and concurrency.

In computing the main difference between parallelism and concurrency comes down to the execution of multiple tasks and how they are completed and appear to finish within the systems in place. When tasks are completed within the same time frame multiple CPU cores are being used to complete tasks within parallelism. When tasks are completed within the same period multiple CPU tasks are being utilized to complete tasks within parallelism. Concurrency is when multiple tasks are managed and handled in a way that they seem to execute simultaneously without truly being executed together. There is little to no dependency between tasks in parallelism while in concurrency tasks are dependent on and share common resources. Although both are valuable they serve different purposes parallelism is concerned with achieving higher velocity and concurrency is aimed at effective task volume.

Part B:

```cpp
#include <iostream>
#include <thread>
using namespace std;

int total = 0;

void addValue(int num) {
    total += num;
}

int main() {
    thread t1(addValue, 5);
    thread t2(addValue, 10);
    thread t3(addValue, 15);

    cout << "Starting value of total = " << total << endl;

    t1.join();
    cout << "After t1.join(), total = " << total << endl;

    t2.join();
    cout << "After t2.join(), total = " << total << endl;

    t3.join();
    cout << "After t3.join(), total = " << total << endl;

    cout << "End of program, final total = " << total << endl;

    return 0;
}
```

Output:

```
Starting value of total = 30
After t1.join(), total = 30
After t2.join(), total = 30
After t3.join(), total = 30
End of program, final total = 30

--------------------------------
Process exited after 0.2853 seconds with return value 0
Press any key to continue . . .
```

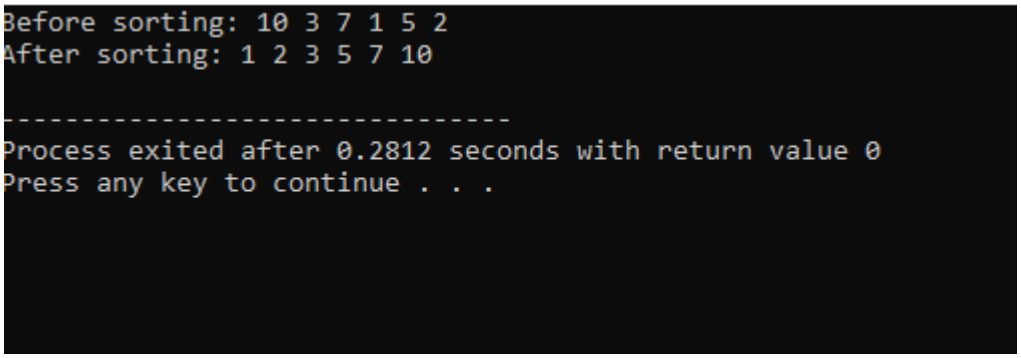
Analysis:

This program implements three threads, each of which adds different numbers to a single global variable. Almost simultaneously, each thread runs the addValue function, which means I may not always get the expected results for the total. As each thread completes its task and rejoins the main program, the output progressively updates. This illustrates the concept of multithreading and how the sequence of execution can impact shared data. The program also demonstrated that threads can lead to unpredictable results in a program when no control is implemented, which is an important consideration in program design.

PART C:

```cpp
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] < R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        thread t1(mergeSort, ref(arr), left, mid);
```

Output:



Analysis:

In the merge sort algorithm I used multithreading by allowing two threads to sort the left and right sides of the array concurrently. This speeds up the sorting process as both sides work in parallel rather than sequentially. After executing the program I confirmed that the output was indeed the numbers sorted in order. However, the array I used was small so the difference in time was not really noticeable. This still demonstrates the usefulness of multithreading for larger datasets because it significantly reduces the time spent sorting.

C. Conclusion & Lessons Learned

After doing this activity deepened my understanding of how multithreading works I find that it allows a program to perform multiple actions simultaneously and also increasing the efficiency of the program. I applied it to the merging of sorts where I observed how different portions of the thread being sorted in the array worked in parallel. And it all seemed confusing I think maybe because I didn't understand the coordination of the threads. I had to go online to find examples and guides to getting it working. Despite the struggles I gained a lot in seeing how useful and powerful multithreading is.

E. External References:

https://www.geeksforgeeks.org/cpp/cpp-concurrency/
https://www.tutorialspoint.com/multithreading-in-c
https://www.geeksforgeeks.org/cpp/multithreading-in-cpp/
https://www.programiz.com/cpp-programming/references
https://www.geeksforgeeks.org/cpp/multithreading-in-cpp/
https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm