Activity No. <13>	
<PARALLEL ALGORITHMS AND MULTITHREADING>	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 11/3/25**
Section: CPE21S4	**Date Submitted: 11/3/25**
Name(s): Alcantara, Jason P.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s):

ILO A & ILO B:

Part 1:
Main CPP:

[*] Single threaded.cpp

```cpp
#include <iostream>
#include <thread>
#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() {
    std::thread t1(print, 10, "T.I.P.");
    t1.join();
    return 0;
}

```

Output:

```
C:\Users\Joshua\Documents\Single threaded.exe
Printing integer: 10
Printing string: T.I.P.

--------------------------------
Process exited after 0.13 seconds with return value 0
Press any key to continue . . .
```

Analysis: This C++ sample is a simple demonstration of utilizing concurrency. Here, the method "print" takes an integer and a string as arguments and will print them to the screen. In the main method, a thread is created running print and the main program is allowed to continue. The join() method is there to ensure that the main program doesn't exit until the thread exits. All in all, this code provides students with an understanding of how to create and control a simple thread in C++.

Part 2:

Main CPP:

Single threaded.cpp | Multi-threaded.cpp

```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:

```
C:\Users\Joshua\Documents\Multi-threaded.exe
0. T.I.P.
1. Competent
2. Computer
3. Engineers

--------------------------------
Process exited after 0.1491 seconds with return value 0
Press any key to continue . . .
```

Analysis: The example in this program illustrates how running multiple threads in C++ allows multiple tasks to execute concurrently. Each thread prints its own message, which helps illustrate how multithreading works. Initially, the output looked jumbled because the threads were all printing at the same time. Once we added a mutex to ensure that only one thread printed to the screen at a time, the output prints nicely, each on a separate line. This example helps illustrate that while multithreading can improve performance, managing threads competing for shared resources, such as the console, is important.

B. Answers to Supplementary Activity:

Part A:

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

- Multithreading is defined as a program that runs multiple threads simultaneously, sharing a single process. All of the threads are able to run independently, but they also share memory and resources. One benefit of this is that

a multithreaded program can run faster because it runs tasks in parallel. Another benefit in major applications is it can keep a program responsive during long running operations with background threads or if the program needs to ask for user input, for example. The major disadvantage is that it is difficult to write programs since threads can conflict with each other often causing bugs like race conditions.

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

- Multithreading is meant to be utilized when some tasks can be executed simultaneously, increasing speed and performance overall. Web servers, for instance, will use multithreading to serve many users without lag since it is possible to serve requests at the same time. Similarly, video games will update graphics, sounds, and user input at the same time using multithreading. Programs which manipulate large files, also tend to read and write at the same time while running background threads. Another use-case is that multithreading can be utilized in simulations, where multiple calculations occur separately at the same time to get results sooner.

3. **Differentiate between parallelism and concurrency.**

- Concurrency occurs when tasks make progress during the same interval but do not happen at the same point in time. Parallelism occurs when tasks are running at the exact same moment on different CPU cores. Concurrency is about one task managing many others, while parallelism is about speed by doing the tasks at the same time. A program can be concurrent without being parallel those tasks might just be switching back and forth on a single core. Knowing about both allows programmers to create faster and more reliable programs.

Part B:

Main CPP:

Single threaded.cpp | Multi-threaded.cpp | multithreading_supplementary.cpp

```cpp
#include <iostream>
#include <thread>

int globalVar = 0;

void add(int value) {
    globalVar += value;
}

int main() {
    std::thread t1(add, 10);
    std::thread t2(add, 20);
    std::thread t3(add, 30);

    std::cout << "Before any join, globalVar = " << globalVar << std::endl;

    t1.join();
    std::cout << "After t1.join(), globalVar = " << globalVar << std::endl;

    t2.join();
    std::cout << "After t2.join(), globalVar = " << globalVar << std::endl;

    t3.join();
    std::cout << "After t3.join(), globalVar = " << globalVar << std::endl;

    return 0;
}

```

Output:

```
C:\Users\Joshua\Documents\multithreading_supplementary.exe
Before any join, globalVar = 60
After t1.join(), globalVar = 60
After t2.join(), globalVar = 60
After t3.join(), globalVar = 60

--------------------------------
Process exited after 0.1316 seconds with return value 0
Press any key to continue . . .
```

Analysis: In this exercise, I implement a global variable called globalVar that can be accessed by all threads. I have defined a function add which takes in an integer and adds the integer to the global variable. Next, I implemented three threads which create each thread calling add with different values so that globalVar gets updated. I called join on each thread one after the other and printed the value of globalVar after each join. This way I could see how globalVar was being altered as each thread completed, but it also demonstrated that the threads are carrying out their work simultaneously.

Part C:

Main CPP:

Single threaded.cpp | Multi-threaded.cpp | multithreading_supplementary.cpp | multi-threading_one algorithm.cpp

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

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

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

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

    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(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        std::thread t1(mergeSort, std::ref(arr), left, mid);
        std::thread t2(mergeSort, std::ref(arr), mid + 1, right);

        t1.join();
        t2.join();

        merge(arr, left, mid, right);
    }
}

int main() {
    std::vector<int> arr = {12, 11, 13, 5, 6, 7};

    std::cout << "Original array: ";
    for (int num : arr)
        std::cout << num << " ";
    std::cout << std::endl;

    mergeSort(arr, 0, arr.size() - 1);

    std::cout << "Sorted array: ";
    for (int num : arr)
        std::cout << num << " ";
    std::cout << std::endl;

    return 0;
}
```

Output:

```
C:\Users\Joshua\Documents\multi-threading_one algorithm.exe
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13

--------------------------------
Process exited after 0.1285 seconds with return value 0
Press any key to continue . . .
```

Analysis: This code is an implementation of the Merge Sort algorithm that uses multi-threading to sort an array of integers. The algorithm is recursive and divides the array into smaller sub-arrays, then combines the sub-arrays in sorted order. The multi-threading is accomplished with std::thread by creating two threads to sort the left half and right half of the array simultaneously. When both threads have completed work, the sub-arrays are merged back together. Although multi-threading can provide performance benefits, the creation of the threads causes overhead that may impede multi-threading benefits for smaller arrays when the threads cost more than the benefits of being multi-threaded.

C. Conclusion & Lessons Learned:

- In this Laboratory Activity, I was introduced to multithreading in C++ and how to run tasks in parallel to take full advantage of CPU resources. Also, I learned about join() functions that can ensure that threads are done executing and the completion of the overall thread. I was able to learn the performance benefits of multithreading, especially when tasks can operate independently from each other, but I learned that there are difficulties associated with managing shared resources and debugging. I learned of one issue that had too much overhead in creating too many threads for smaller tasks, along with the need for proper thread synchronization so data isn't inadvertently changed while consuming. I believe I still must improve my skills in multithreading, and I still have a lot to learn, especially around multithreading with different use cases and improving performance.

D. Assessment Rubric

E. External References