Hands-on Activity 13.1	
Parallel Algorithms and Multithreading	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11 - 03 - 25
Section: CPE21S4	**Date Submitted:** 11 - 03 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

ILO A: Part 1: Simple One-Threaded Example

Table 13-1. Simple One-Threaded Example

Code:

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

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();
   std::cout << std::endl;


   return 0;
}
```

Output

```
"D:\College\Year 2 CPE\Y2
Printing integer: 10
Printing string: T.I.P.
```

Analysis

When I experimented by removing the join() line, I saw an error that said “terminate called without an active

exception." As I analyzed it, it is because I never joined the thread object in the main thread, thus, while the main thread finishes executing, the thread object is still running and is needed to be terminated. As I looked for more information online, they reflected my analysis as well. This error is a useful diagnostic tool when dealing with multithreaded operations. It's important to account for all threads created and running and must be properly joined and handled to ensure that no errors or undefined behaviors will happen.

Reference:
https://codemia.io/knowledge-hub/path/c_terminate_called_without_an_active_exception

ILO B: Part 2: Multi-Threaded Example

Table 13-2. Multithreaded Example.

Code:

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


void printMulti(int n, const std::string &str) {
   std::string msg = std::to_string(n) + " : " + str;
   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.push_back(std::thread(printMulti, i, s[i]));
   }
   for (auto &th : threads) {
       th.join();
   }
   return 0;
}
```

Output:

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

Process finished with exit code 0
```

Analysis:

In this code, although the loops join the threads starting from 0 to 3, the resulting output is not in order of joining. This is because these threads run concurrently or at the same time during runtime with the main thread. And since there is nothing to control the output of threads or synchronization between them, the output will be unpredictable—as seen in the output above.

7. 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.

   Multi-threading is a technique that allows multiple instructions or threads to run concurrently or simultaneously within the same process. Multithreading also allows each thread to share the same memory and resources, thus maximizing performance. This is especially useful when we want to maximize computer resources to speed up processes. An advantage is that it speeds up processes since it utilizes more CPU resources, which improves performance. This is due to the fact that threads can share the same resources, hence, the response time would be faster when working on the same set of data. For example, sorting the same set of data through multi-threading would be much faster since we're using one process but dividing the work into multiple threads. However, data sharing also poses a critical downside—data races. This occurs when the threads aren't properly synchronized or ordered, so they end up accessing a single variable at the same time, which messes the process. However, since multi-threading is such a powerful technique, it is worth the risk of data races; and besides, there are workarounds like mutex and data synchronization to combat this.

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

   Business transactions are one of the most common use-cases of multi-threading. Since we're handling many transactions at the same time, multi-threading allows us to handle these transactions simultaneously. Another common use case is through web services like browsers. Browsers utilize multi-threading by maximizing computer resources when having multiple tabs open and active. These tabs are running simultaneously which allows the user to seamlessly switch between them without having to load each tab every time. Of course, this still risks overuse of resources, hence why modern web browsers and computer systems limit the amount of

times a tab is running but currently in the background. Lastly, modern computer systems are probably the best example of multi-threading. CPUs started from single cores to dual cores to multiple cores, and since we can only fit so many cores in a tiny chip, the best way to improve performance is to maximize a single core by letting it process multiple threads. When multiple cores handle multiple threads concurrently, this achieves efficient and fast processing because idle resources on a single core are also being utilized. This is why modern CPUs, even with less cores, can be faster than previous generations with more cores.

3. Differentiate between parallelism and concurrency.

The main difference is that concurrency imitates parallelism by allowing multiple threads to take turns handling and utilizing the same resources on the same processor. This increases response time and mimics parallelism, where it seems that multiple processes are being handled at once, but in reality, the processor is just letting each thread alternate between the same resources. Meanwhile, true parallelism allows multiple processes to happen simultaneously—but they are handled by different processors. For example, concurrency is when Processor A lets Threads 1 and 2 to handle the same array and computer resources but in a controlled manner. This accomplishes the tasks of each thread quickly since we're maximizing Processor A, but we are not actually executing thread 1 and 2 in parallel. Meanwhile, parallelism is when Processor A handles Thread 1, while Processor B handles Thread 2. In this case, true parallelism occurs because both processors are active and are executing both threads individually but simultaneously.

References:

https://www.geeksforgeeks.org/operating-systems/multithreading-in-operating-system/

https://www.geeksforgeeks.org/operating-systems/difference-between-concurrency-and-parallelism/

Part B: Create C++ Code and show a solution that satisfies the given requirements below.

- Create a global variable of type integer.
- Create an add function that will take an integer parameter and add that value to the global variable.
- Use multi-threading techniques to create 3 threads; individually pass the add function to the threads.
- Display the value of global variables at different combinations of using the join() per thread. Such that:
  - Display
  - T1.join()
  - Display
  - T2.join()
  - Display
  - T3.join()

Provide an analysis based on the outputs of the multi-threading exercise.

Code:

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

int global = 0;

void addToGlobal(int n) {
```

```
    global += n;
    std::cout << "Adding " << n << " to global: " << global <<
std::endl;
}
int main() {

    std::cout << "Global: " << global << std::endl;

    std::thread t1(addToGlobal, 5);
    std::thread t2(addToGlobal, 6);
    std::thread t3(addToGlobal, 7);

    t1.join();
    std::cout << "t1 Global: " << global << std::endl;
    t2.join();
    std::cout << "t2 Global: " << global << std::endl;
    t3.join();
    std::cout << "t3 Global: " << global << std::endl;

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010
Global: 0
Adding 5 to global: 18
Adding 6 to global: 18
Adding 7 to global: 18
t1 Global: 18
t2 Global: 18
t3 Global: 18

Process finished with exit code 0
```

Analysis

This code perfectly demonstrates the issue of unsynchronized multithreading resulting in a data race. Although I joined and displayed the results of each thread in a specific order, they do not reflect the expected output from order alone. They are accessing a common global variable, and since these are multithreaded threads, they accessed the global variable simultaneously, resulting in all three instances being applied to the global variable instead of one before the other. It's an expected flaw in multithreading as this leads to unpredictable and wrong results, which is why it is important to apply data synchronization through different methods like mutex or atomic synchronization.

Reference:
https://www.geeksforgeeks.org/cpp/data-races-in-cpp/

Part C: Use multi-threading with one of the algorithms previously developed in the course; provide an analysis of the result.

Code:

Header/Algorithm to be tested:

```
//
// Created by David Owen A. Santiago on 16/09/2025.
//
#ifndef SEARCHING_H
#define SEARCHING_H

#include <iostream>

// Linear search function template
template <typename T>
bool arrayLinearSearch(const T data[], int size, T item) {
    for (int i = 0; i < size; ++i) {
        if (data[i] == item) {
            std::cout << "Searching is successful. Found at index: " <<
i << "\n";
            return true;
        }
    }
    std::cout << "Searching is not successful.\n";
    return false;
}


template <typename T>
bool arrayBinarySearch(const T arr[], int size, T item) {
    int low = 0;
    int up = size - 1;

    while (low <= up) {
        int mid = (low + up) / 2;

        if (arr[mid] == item) {
            std::cout << "Search element is found at index: " << mid <<
"\n";
            return true;
        } else if (item < arr[mid]) {
            up = mid - 1;
```

```
        } else {
            low = mid + 1;
        }
    }

    std::cout << "Search element is not found.\n";
    return false;
}

template <typename T>
int arrayLinearSearchComparisons(const T data[], int size, T item) {
    int comparisons = 0;
    for (int i = 0; i < size; ++i) {
        comparisons++;
        if (data[i] == item) break;
    }
    return comparisons;
}

template <typename T>
int countOccurrencesArray(const T data[], int size, T item) {
    int count = 0;
    for (int i = 0; i < size; ++i)
        if (data[i] == item) count++;
    return count;
}

template <typename T>
int recursiveBinarySearch(const T arr[], T low, T up, T key) {
    if (low > up) return -1;

    int mid = (low + up) / 2;

    if (arr[mid] == key)
        return mid;
    else if (key < arr[mid])
        return recursiveBinarySearch(arr, low, mid - 1, key);

    else
        return recursiveBinarySearch(arr, mid + 1, up, key);
}

#endif // SEARCHING_H
```

Main:

```
//
// Created by David Owen A. Santiago on 03/11/2025.
//
```

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include "searching.h"

void runBinarySearch(int* arr, int size, int target) {
    arrayBinarySearch<int>(arr, size, target);
}

void runLinearSearch(int* arr, int size, int target) {
    arrayLinearSearch<int>(arr, size, target);
}

int main() {

    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 5;

    std::thread t1(runBinarySearch, arr, size, target);
    std::thread t2(runLinearSearch, arr, size, target);
    t1.join();
    t2.join();


    return 0;
}
```

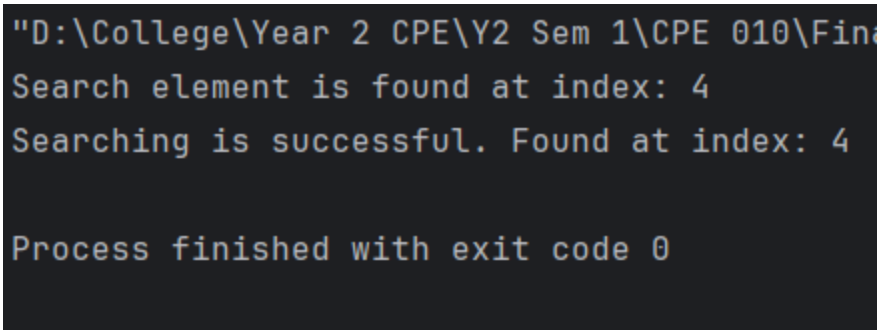
Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Finals\HOA 13\cmake-build-debug\HOA_13.exe"
Search element is found at index: Searching is successful. Found at index: 4
4

Process finished with exit code 0
```

Analysis:

I decided to test multi-threading with the old linear and binary searching algorithm that I created previously. Here, I wanted to see what would happen if I ran both concurrently. When I run the code, the printed output shows mixed outputs from both. The expected output would've been like this:



But since we're running it concurrently, I believe what's happening is that the program is accessing std::cout at the same time when running the two functions, thus, resulting in the jumbled results. When run sequentially, the output will be like the expected output, where I join thread 1 before creating and joining thread 2. Despite the weird output, I believe with more additions, like data synchronization using mutex, we can utilize multi-threading much more efficiently. One of the ways is to run linear search concurrently through threads such that thread 1 searches the first half while thread 2 searches the second half of the array. This results in a much faster execution since we're cutting the searching in half instead of iterating through each element one by one. I can see this being especially useful when dealing with larger arrays that would require more resources to sift through.

8. Conclusion

In conclusion, concurrency and parallelism are both powerful techniques to maximize performance and make algorithms faster. They are integral concepts in computer architecture since they not only improve performance but they also make the most of the resources in a computer. Concurrency is when a single processor lets multiple threads handle the same resources and memory. This imitates parallelism since these threads are being handled quickly, but it still isn't true parallelism because these threads are only taking turns accessing the resources, they aren't actually accessing the data simultaneously, otherwise this would cause issues like data races.

Concurrency doesn't necessarily emphasize improving overall performance, instead, it practices the maximization of existing and idle resources on a single processor to execute threads faster. Meanwhile, parallelism is when multiple processors handle multiple threads at the same time. This with the combination of concurrency maximizes performance even more; this is the main reason why modern CPUs are so much faster than previous generations, even if they have less cores. Efficiency is the biggest contribution of concurrency and parallelism. Multiple processors practicing concurrency when running in parallel provide the best and most performance out of a single chip.

These are also core concepts in algorithm structures, as programming and computer hardware go hand-in-hand. Parallel and concurrency in programming allows us to maximize resource usage in our program and still be as efficient as possible. We can utilize more idle resources which cut the runtime significantly, which is important when creating algorithms for broad cases like web services and financial transactions. Hence, in the modern technology centered world, parallelism and concurrency can be considered the backbone of algorithms in computers all over the world. It is what helps make our lives easier and faster.

9. Assessment Rubric