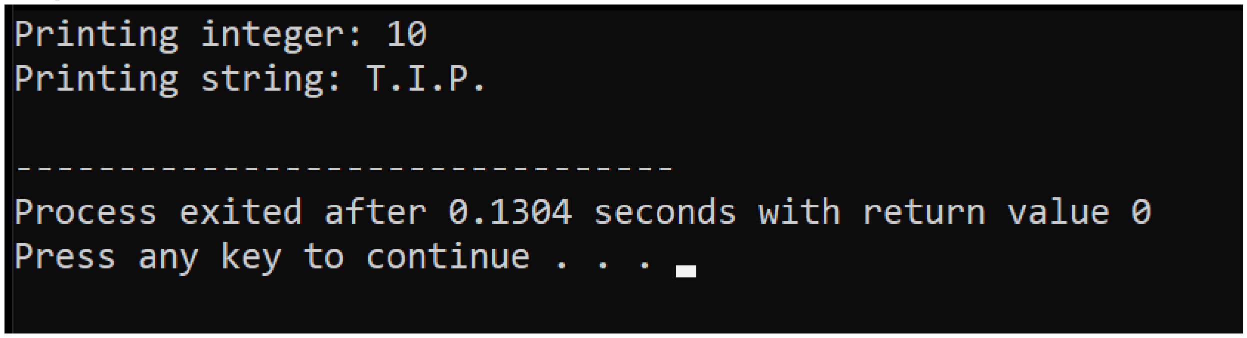
Hands-on Activity 13.1	
Parallel Algorithms and Multithreading	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/3/25
Section: CPE21S3	**Date Submitted:** 11/3/25
Name(s): Avila, Vince Gabriel V.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

Part 1:

```
#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:



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

--------------------------------
Process exited after 0.1304 seconds with return value 0
Press any key to continue . . .
```

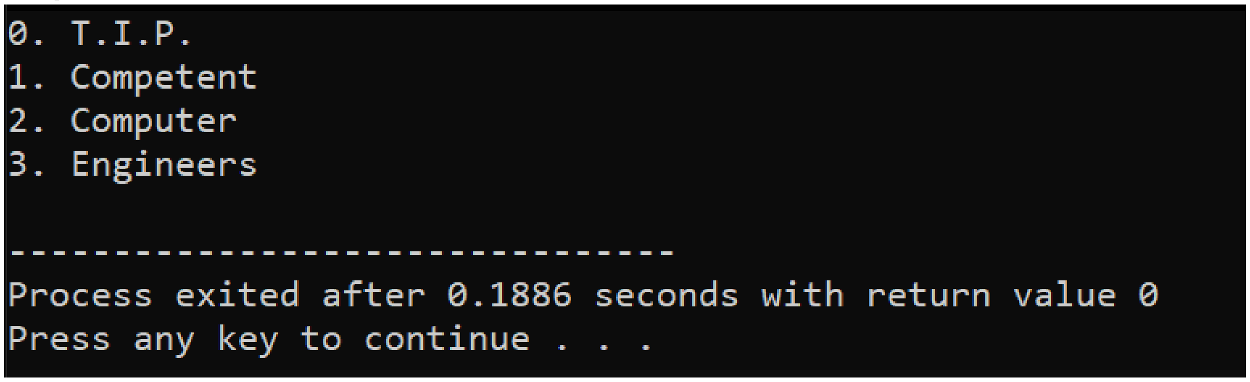
Analysis:

By using a thread to invoke a function to print an integer and a string, the following code demonstrates how threads behave in C++. It creates a thread, and joins that thread to the main program so that the thread completes execution of the function call before the main program terminates.

Part 2:

```
#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:



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

--------------------------------
Process exited after 0.1886 seconds with return value 0
Press any key to continue . . .
```

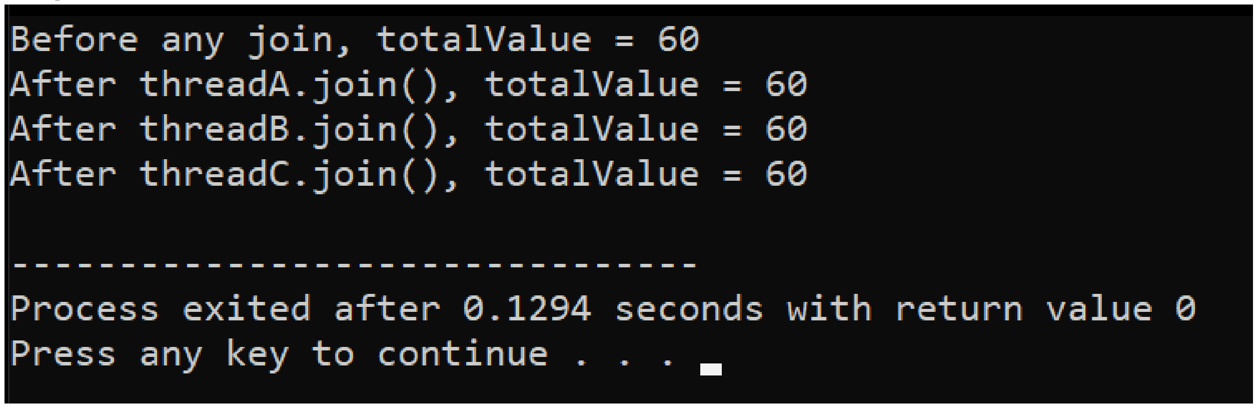
Analysis:
This code uses multiple threads to print several strings at the same time without mixing their outputs. It also uses a mutex to make sure that only one thread prints at a time, keeping the output organized and readable.

B. Answers to Supplementary Activity

Part B

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

int totalValue = 0;

void increase(int num) {
    totalValue += num;
}

int main() {
    std::thread threadA(increase, 10);
    std::thread threadB(increase, 20);
    std::thread threadC(increase, 30);

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

    threadA.join();
    std::cout << "After threadA.join(), totalValue = " << totalValue << std::endl;

    threadB.join();
    std::cout << "After threadB.join(), totalValue = " << totalValue << std::endl;

    threadC.join();
    std::cout << "After threadC.join(), totalValue = " << totalValue << std::endl;

    return 0;
}
```

Output:



```
Before any join, totalValue = 60
After threadA.join(), totalValue = 60
After threadB.join(), totalValue = 60
After threadC.join(), totalValue = 60


--------------------------------
Process exited after 0.1294 seconds with return value 0
Press any key to continue . . .
```

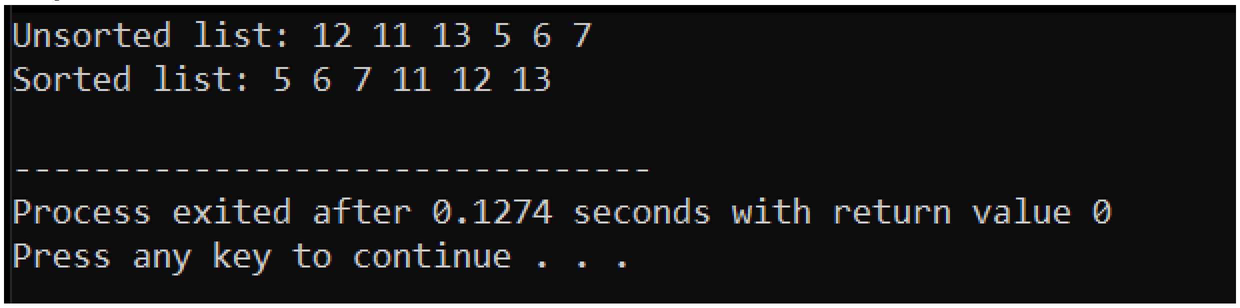
Analysis:

This code uses three threads which can modify a common variable simultaneously by adding different values. It demonstrates that threads can run concurrently, but it will not always produce the same results, as all threads can access the same variable without synchronization.

Part C

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

void combine(std::vector<int>& data, int begin, int midPoint, int end) {
    int leftSize = midPoint - begin + 1;
    int rightSize = end - midPoint;

    std::vector<int> leftPart(leftSize), rightPart(rightSize);

    for (int a = 0; a < leftSize; a++)
        leftPart[a] = data[begin + a];
    for (int b = 0; b < rightSize; b++)
        rightPart[b] = data[midPoint + 1 + b];

    int i = 0, j = 0, k = begin;
    while (i < leftSize && j < rightSize) {
        if (leftPart[i] <= rightPart[j]) {
            data[k] = leftPart[i];
            i++;
        } else {
            data[k] = rightPart[j];
            j++;
        }
        k++;
    }

    while (i < leftSize) {
        data[k] = leftPart[i];
        i++;
        k++;
    }

    while (j < rightSize) {
        data[k] = rightPart[j];
        j++;
        k++;
    }
}

void threadedMergeSort(std::vector<int>& data, int begin, int end) {
    if (begin < end) {
        int midPoint = begin + (end - begin) / 2;

        std::thread worker1(threadedMergeSort, std::ref(data), begin, midPoint);
        std::thread worker2(threadedMergeSort, std::ref(data), midPoint + 1, end);

        worker1.join();
        worker2.join();

        combine(data, begin, midPoint, end);
    }
}

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

    std::cout << "Unsorted list: ";
    for (int n : numbers)
        std::cout << n << " ";
    std::cout << std::endl;

    threadedMergeSort(numbers, 0, numbers.size() - 1);

    std::cout << "Sorted list: ";
    for (int n : numbers)
        std::cout << n << " ";
    std::cout << std::endl;

    return 0;
}
```

Output:



```
Unsorted list: 12 11 13 5 6 7
Sorted list: 5 6 7 11 12 13


--------------------------------
Process exited after 0.1274 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This code leverages multithreading to execute merge sort, which allows the sorting of numbers to occur at a faster rate by distributing the sorting task between threads. The output illustrates the sorting of each segment of the list separately and a merge of each segment into the final sorted list.

C. Conclusion & Lessons Learned

For my conclusion, I learned the role of multithreading and how multithreading makes programs run faster through this activity. I used merge sort as my algorithm and discovered that splitting the operation into threads improves it. At first, I found it difficult to know how to handle the creation of threads and how thread synchronization works. I ended up Googling some tutorials to get the code to work, and after some trial and error, I was able to figure out how to get it all running and understood how the logic worked. Even though the threads were confusing at times and I had to depend on the internet and other sources for some of the work, I still felt it was worthwhile overall because I gained a much better understanding of multithreading. This activity also gave me more confidence in writing C++ programs that successfully utilize threads.

D. Assessment Rubric

E. External References

GeeksforGeeks. (2025d, July 23). *Parallel algorithm models in parallel computing*. GeeksforGeeks.

https://www.geeksforgeeks.org/mobile-computing/parallel-algorithm-models-in-parallel-computing/

GeeksforGeeks. (2025n, October 3). *Multithreading in C++*. GeeksforGeeks.

https://www.geeksforgeeks.org/cpp/multithreading-in-cpp/