ACTIVITY NO. 13

PARALLEL ALGORITHMS AND MULTITHREADING	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/31/25**
Section: CPE21S4	**Date Submitted: 10/31/25**
Name: Cruz, Axl Waltz E.	**Instructor:** Engr. Jimlord Quejado

1. Objective(s)

Create C++ code that uses modern programming techniques to implement parallel algorithms, multithreading and concurrency.

2. Intended Learning Outcomes (ILOs)

After this activity, the student should be able to:

- Create C++ code that uses parallel algorithms for better performance.
- Create C++ code that implements multithreading and concurrency.

3. Discussion

In the modern tech climate, concurrency has become an essential skill for all CPP programmers. As programs continue to get more complex, computers are designed with more CPU cores to match.

The best way for you to make use of these multicore machines is the coding technique of concurrency.

What is concurrency?

Concurrency occurs when multiple copies of a program run simultaneously while communicating with each other. Simply put, concurrency is when two tasks are overlapped. A simple concurrent application will use a single machine to store the program's instruction, but that process is executed by multiple, different threads.

This setup creates a kind of control flow, where each thread executes its instruction before passing to the next one. The threads act independently and to make decisions based on the previous thread as well. However, some issues can arise in concurrency that make it tricky to implement.

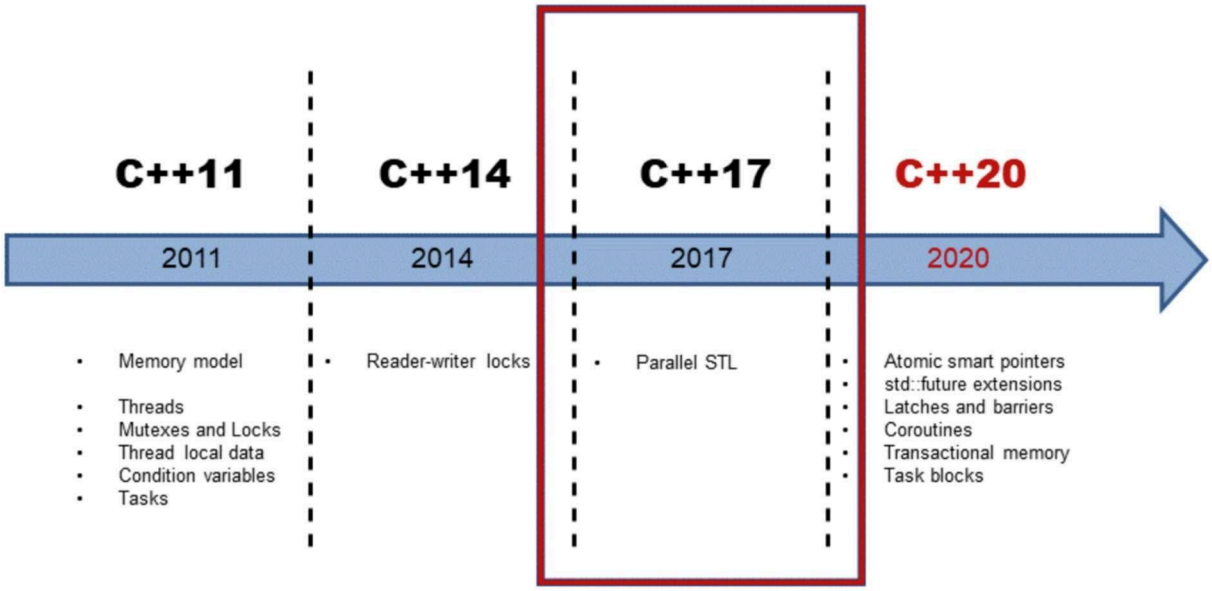
Example:

- A data race is a common issue you may encounter in C++ concurrency and multi-threaded processes. Data races in C++ occur when at least two threads can simultaneously access a variable or memory location, and at least one of those threads tries to access that variable.

This can result in undefined behavior. Regardless of its challenges, concurrency is very important for handling multiple tasks at once.

History of Concurrency

C++11 was the first C++ standard to introduce concurrency, including threads, the C++ memory model, conditional variables, mutex, and more. The C++11 standard changes drastically with C++17. The addition of parallel algorithms in the Standard Template Library (STL) greatly improved concurrent code.



Concurrency vs. Parallelism

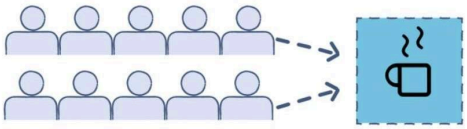
Concurrency and parallelism often get mixed up, but it's important to understand the difference. In parallelism, we run multiple copies of the same program simultaneously, but they are executed on different data.

Example:

- You could use parallelism to send requests to different websites but give each copy of the program a different set of URLs. These copies are not necessarily in communication with each other, but they are running at the same time in parallel.

As we explained above, concurrent programming involves a shared memory location, and the different threads actually "read" the information provided by the previous threads.

Concurrent: *Two Queues* & a *Single* Espresso machine.



Parallel: *Two Queues* & *Two* Espresso machines.

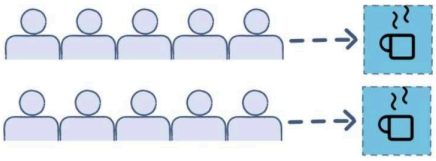


Image from EdPresso shot, "What is concurrent programming?"

Methods of Implementing Concurrency

In C++, the two most common ways of implementing concurrency are through multithreading and parallelism. While these can be used in other programming languages, C++ stands out for its concurrent capabilities with lower-than-average overhead costs as well as its capacity for complex instruction.

C++ multithreading involves creating and using thread objects, seen as `std::thread` in code, to carry out delegated sub-tasks independently.

New threads are passed a function to complete, and optionally some parameters for that function.

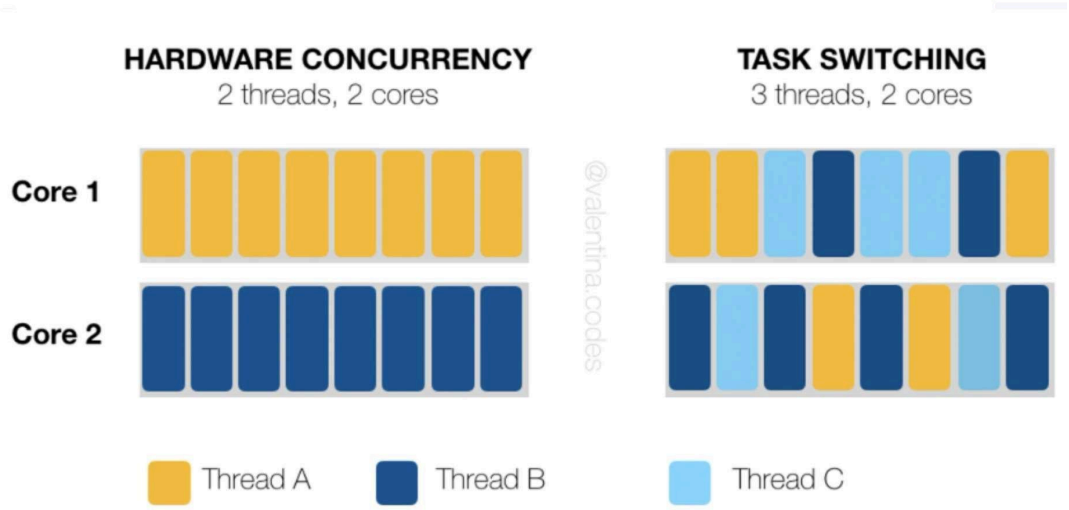


Image from Medium, [C++] Concurrency by Valentina

While each individual thread can complete only one function at a time, thread pools allow us to recycle and reuse thread objects to give programs the illusion of unlimited multitasking. Not only does this take advantage of multiple CPU cores, but it also allows the developer to control the number of tasks taken on by manipulating the thread pool size. The program can then use the computer resources efficiently without overloading becoming overloaded.

To better understand thread pools, consider the relationship of worker bees to a hive queen:

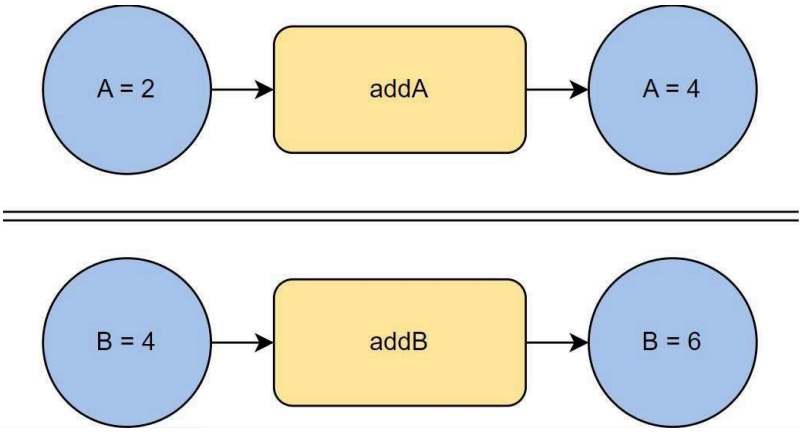
The queen (the program) has a broader goal to accomplish (the survival of the hive) while the workers (the threads) only have their individual tasks given by the queen. Once these tasks are completed, the bees return to the queen for further instruction. At any one time, there is a set number of these workers being commanded by the queen, enough to utilize all of its hive space without overcrowding it.

Parallelism

Creating different threads is typically expensive in terms of both time and memory overhead for the program. Multithreading can therefore be wasteful when dealing with short simpler functions. Parallel functions can significantly speed up operations because they automatically use more of the

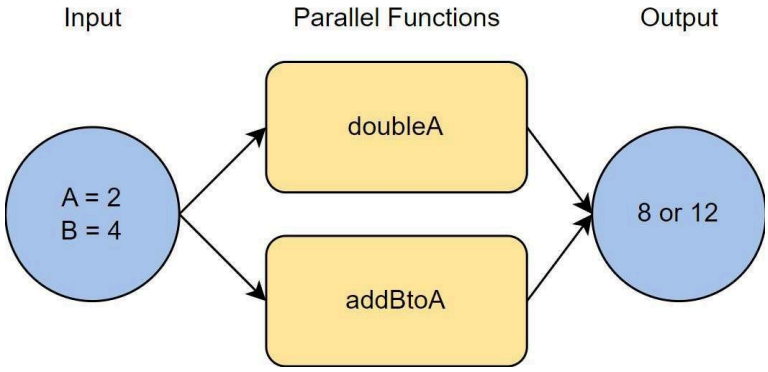
computer’s CPU resources.

However, it is best saved for functions that have little interaction with other functions using dependencies or data editing. This is because while they are worked on concurrently, there is no way to know which will complete first, meaning the result is unpredictable unless synchronization such as mutex or condition variables are used.



Parallel execution of separate threads

Imagine we have two variables, A and B, and create functions addA and addB, which add 2 to their value. We could do so with parallelism, as the behavior of addA is independent on the behavior of the other parallel function addB, and therefore has no problem executing concurrently.



Parallel functions that create unexpected outcomes

However, if the functions both impacted the same variable, we would instead want to use sequential execution. Imagine that we instead had one which multiplied variable A by two, doubleA, and another which added B to A, addBtoA.

In this case, we would not want to use parallel execution as the outcome of this set of functions depends on which is completed first and would, therefore, result in a race condition.

While both multithreading and parallelism are helpful concepts for implementing concurrency in a C++ program, multithreading is more widely applicable due to its ability to handle complex operations. In the next section, we'll look at a code example of multithreading at its most basic.

4. Materials and Equipment

Personal Computer with C++ IDE

Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A & ILO B:

Part 1: Simple One-Threaded Example

Since all threads must be given a function to complete at their creation, we first must declare a function for it to be given. We'll name this function print and will design it to take int and string arguments when called.

When executed, this code will simply report the data values passed in.

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

In the next section, we'll initialize a thread and have it execute the above function. To do this, we'll have the main function, the default executor present in all C++ applications, initialize the thread for the print function.

After that, we use another handy multithreading command, `join()`, pausing the main function's thread until the specified thread, in this case t1, has finished its task. Without `join()` here, the main thread would finish its task before t1 would complete print, resulting in an error.

```
int main() {
    std::thread t1(print, 10, "T.I.P.");
    t1.join();
return 0;
}
```

Show your output and analysis in section 6 as its

own table. Part 2: Multi-Threaded Example

While the outcome of the single thread example above could easily be replicated without using multithreaded

code, we can truly see concurrency's benefits when we attempt to complete print multiple times with different sets of data. Without multithreading, this would be done by simply having the main thread repeat print one at a time until completion.

To do this with concurrency in mind, we instead use a for loop to initialize multiple threads, pass them the print function and arguments, which they then complete concurrently. This multithreading option would be faster one using only the main thread as more of the total CPU is being used.

Runtime difference between multithreading and non-multithreading solutions increasing as more print executions are needed.

Let's see what a many-thread version of the above code would look like:

```
void print(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(print, i, s[i]));
  }

  for (auto &th : threads) {
    th.join();
  }
  return 0;
}
```

Show your output and analysis in section 6 as its own table.

6. Output

Table 13-1. Simple One-Threaded Example

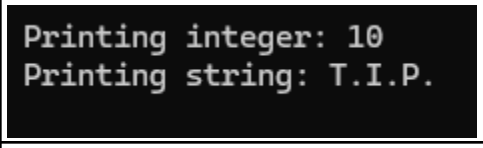
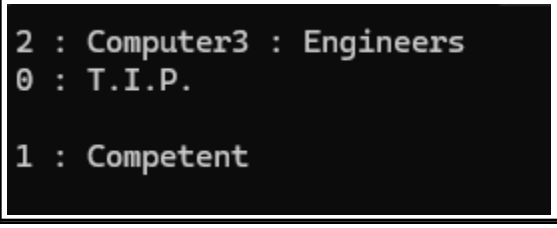
Screenshot	
Analysis	The program demonstrates parallelism and multithreading by creating a single thread that runs the print function separately from the main program. The thread prints an integer and a string while the main thread waits for it to finish. Even though there is only one thread, it still shows how tasks can run independently. By joining the thread at the end, the program ensures that the thread completes its work before the program exits.

Table 13-2. Multithreaded Example.

Screenshot	

Analysis	The program shows parallelism and multithreading by creating multiple threads that run the print function at the same time. Each thread prints its own message like 0 : T.I.P. or 1 : Competent independently of the others. Because threads execute concurrently, the order of the printed lines may differ each time the program runs, showing real parallel behavior. By joining all threads at the end, the program ensures all parallel tasks complete before it exits.

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.

Multithreading allows a program to run multiple parts, called threads, at the same time. It helps improve performance, speed, and responsiveness by doing many tasks together. The main advantages are faster execution, better use of system resources, and smoother performance. However, it can be hard to debug and may cause errors if threads share data incorrectly. It is used in web servers, games, and data processing systems to handle many tasks efficiently. Source: https://phoenixnap.com/glossary/what-is-multithreading?utm_source

2. Rationalize the use of multithreading by providing at least 3 use-cases.
   Parallelism means doing many tasks at the same time using multiple processors. Concurrency means handling many tasks at once, even if not all run together. Parallelism focuses on speeding up execution, while concurrency focuses on managing tasks efficiently. Both are important for improving performance in modern programs.
   Source: https://www.geeksforgeeks.org/difference-between-concurrency-and-parallelism

3. Differentiate between parallelism and concurrency.

- Multithreading is used to make programs faster and more efficient. It is helpful in applications that need to do many actions at the same time, like responding to users or processing large amounts of data. Examples include web servers that serve many clients, games that run physics and graphics together, and apps that download or calculate multiple things at once.

  Source: https://www.geeksforgeeks.org/benefits-of-multithreading-in-operating-system

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 variable 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.

Main.cpp 44392xst5

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

int globalVar = 0;
mutex mtx;

void add(int value) {
    lock_guard<mutex> lock(mtx);
    globalVar += value;
    cout << "Thread adding " << value << ", globalVar = " << globalVar << endl;
}

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

    t1.join();
    cout << "After T1.join(), globalVar = " << globalVar << endl;

    t2.join();
    cout << "After T2.join(), globalVar = " << globalVar << endl;

    t3.join();
    cout << "After T3.join(), globalVar = " << globalVar << endl;

    cout << "Final globalVar = " << globalVar << endl;
    return 0;
}
```

STDIN

Input for the program (Optional)

Output:

```
Thread adding 10, globalVar = 10
Thread adding 5, globalVar = 15
After T1.join(), globalVar = 15
After T2.join(), globalVar = 15
Thread adding 15, globalVar = 30
After T3.join(), globalVar = 30
Final globalVar = 30
```

This program uses three threads to safely update a shared global variable, showing how multithreading allows multiple tasks to run independently. Joining threads ensures that the updates happen in a controlled order, preventing conflicts and demonstrating synchronization.

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

Main.cpp 44392xst5

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

int totalSum = 0;
mutex mtx;

void partialSum(vector<int>& arr, int start, int end) {
    int localSum = 0;
    for (int i = start; i < end; ++i)
        localSum += arr[i];
    lock_guard<mutex> lock(mtx);
    totalSum += localSum;
}

int main() {
    vector<int> numbers = {1,2,3,4,5,6,7,8,9};
    int n = numbers.size();

    thread t1(partialSum, ref(numbers), 0, n/3);
    thread t2(partialSum, ref(numbers), n/3, 2*n/3);
    thread t3(partialSum, ref(numbers), 2*n/3, n);

    t1.join();
    t2.join();
    t3.join();

    cout << "Total Sum of array using threads: " << totalSum << endl;
    return 0;
}
```

STDIN

Input for the program (Optional)

Output:

```
Total Sum of array using threads: 45
```

This program divides an array into three parts and lets three threads compute partial sums in parallel, demonstrating parallelism. Using multithreading speeds up the calculation and safely combines results using a mutex, showing how tasks can run simultaneously for better performance.

8. Conclusion

Provide the following:

- Summary of lessons learned
  - We learned how multithreading allows multiple tasks to run at the same time, improving program performance. Using threads safely with mutex prevents data conflicts when sharing variables. Parallelism can speed up calculations by splitting work across multiple threads.
- Analysis of the procedure
  - Creating threads and using join() helped control execution order and synchronize tasks. The global variable example showed the importance of safe access in multithreaded programs. The array sum example demonstrated how dividing work among threads achieves parallel computation efficiently.
- Analysis of the supplementary activity
  - The exercises reinforced how threads can run independently yet share data safely. Using multithreading improved responsiveness and computation speed in our examples.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
  - I think I did well in understanding and applying multithreading and parallelism in C++. I need to improve on debugging complex threaded programs and optimizing thread usage. Overall, this activity helped me see practical benefits of threads and parallel computation.

9. Assessment Rubric