Raphael Braun
Ayesha Feroz
Kyowon Jeong
Matteo Pilz
Lukas Ruppert
Arslan Siraj
Faezeh S. Zakeri

Eberhard Karls
Universität
Tübingen

Programming in C++

Sheet 9

Submission date: 14.10.2021

Concurrency - first steps in multi-threaded programming

Modern PCs typically have multiple physical cores. It is thus just natural that we want to leverage the additional compute power by processing tasks concurrently. But it is not only about mere compute power. There are many scenarios where concurrent tasks are necessary to make a program work. For example, a web browser receives data through an internet connection, but we want to start rendering the webpage as soon as possible (e.g., while data is still transmitted). Another common use case is a graphical user interface needs to stay responsive even if there are calculations done in the background.
In general, we are safe to say that concurrency is a difficult topic with many hidden traps. This is also true for concurrency in C++ and whole books have been written on the topic (see e.g., C++ Concurrency in action). Today we will skim over some selected topics: classic synchronization tasks and how they can be implemented in C++. Note: We tried to make the tasks automatically testable but testing multithreaded code is challenging. Code that passes a test once doesn't need to be correct. We would thus like you to add clear comments to the code you need to add.

9.1 Synchronization with mutexes (30P)

C++

Synchronisation of threads is necessary to prevent race conditions when parallel running threads are reading and writing shared data. Please take a look at the handed out template and try to understand the code. You will find a set of stressed out salespersons that sell a shared stock of items.
Note: Please run the code multiple times on your computer and carefully check the printed message to check if the current implementation has obvious problems.
First, you should identify problematic accesses to shared data by reading the code. Use the global `std::mutex`es to prevent possible race conditions in the `sell_book` and `sell_toy` function. Run the program a few times to ensure that your implementation does not lead to dead-locks before submitting.

9.2 Waiting on condition variables (optional 30P)

C++

The producer-consumer problem is a classical multi-threaded synchronization problem in concurrent programming. One or several producers (e.g., a network connection, a file stream, some incoming data from hardware) generate data in an asynchronous way. On the other end, a consumer (e.g., a web browser, file viewer, hardware driver) receives the data and processes it. Here we model a simple scenario with a fast and slow producer that yields numbers with different speed and a consumer that takes a pair of numbers (one from the slow and one from the fast producer) and processes the pair of numbers in the order they were produced. We want to make sure that the consumer always has corresponding values from the two consumers to process (e.g., the n-th element from the slow producer should be processed with the n-th elmenent of the fast producer). To ensure this, both producers push their newly generated values at the end of their queues. Because we want to start consuming the pair of values as soon as possible we use `std::condition_variable`s to notify the consumer when an item is ready to pop from the slow queue. Once notified, the producer pops one element and uses the other condition variable to wait for the fast producer (Note: which is faster and should already have some values added to his queue - ready to pop).

In this task, most of the code is again already given, and your first job is to understand the code.
Hint: add `notify_one()` in the producers to indicate that a new value is available and two `wait()` statements in the consumer to wait for the value from slow and fast producer. Think about the predicate and make sure that you don't forget to unlock. All lines were you should add code are indicated in the student template.

9.3 Using STL functionality for parallel processing (40P)

C++

Luckily, many multi-threading tasks require less complex or no synchronization at all. For these cases, you can get good speedups by simply replacing STL algorithms with versions that use parallel execution policies. Use the STL std::sort algorithm to sort the vector of one million random numbers using serial (aka normal execution) and parallel execution by passing execution::par as execution policy. Do the same for std::transform and the expensive trigonometric function. Do you observe a speedup? Note: To make these execution policies work on your PC you might need to have libtbb installed (see Sheet00).

9.4 Using OpenMP for parallel processing (30P + optional 20P [0P + 10P + 10P + 10P + optional 20P]

C++

OpenMP is a framework for multi-platform shared-memory parallel programming. It uses annotated C++ source code using pragmas to define how e.g. loops are processed in parallel. It predates many of the modern C++ features for parallel programming and is still widely used. As it will not go away in the foreseeable future, it is a good idea to take a closer look how to program with OpenMP. Our toy example calculates and plots histograms of a normal distribution using different synchronization mechanisms available in OpenMP. We will benchmark the different mechanisms and check that no sampled point get's lost. Take a look at the source code and associated tasks:

- hist_serial is the default version. Here you don't need to change anything as you don't run into synchronization problems.
- In hist_critical we marked the for loop for parallel execution. Your task is here to add an OpenMP "critical section" to synchronize concurrent access to the histogram. Without protecting the histogram for concurrent access, two threads might read the same value v from same histogram bin, increment v and both write v+1 back. Effectively losing one count.
- In hist_atomic you need to synchronize concurrent access to the histogram again but use the atomic keyword in OpenMP.
- In hist_element_lock you need to synchronize concurrent access to individual bins in the histogram using many OpenMP locks. Think about why this could be faster than the previous two methods. Will it be in the end?
- (optional) In hist_lockfree we use our brain to come up with a solution that needs no synchronization at all.