Concurrency with Modern C++

What every professional C++ programmer should know about concurrency.

Rainer
Grimm

Concurrency with Modern C++

What every professional C++ programmer should know about concurrency.

Rainer Grimm

This book is for sale at http://leanpub.com/concurrencywithmodernc

This version was published on 2020-02-22

Tweet This Book!

Please help Rainer Grimm by spreading the word about this book on Twitter!

The suggested hashtag for this book is #rainer_grimm.

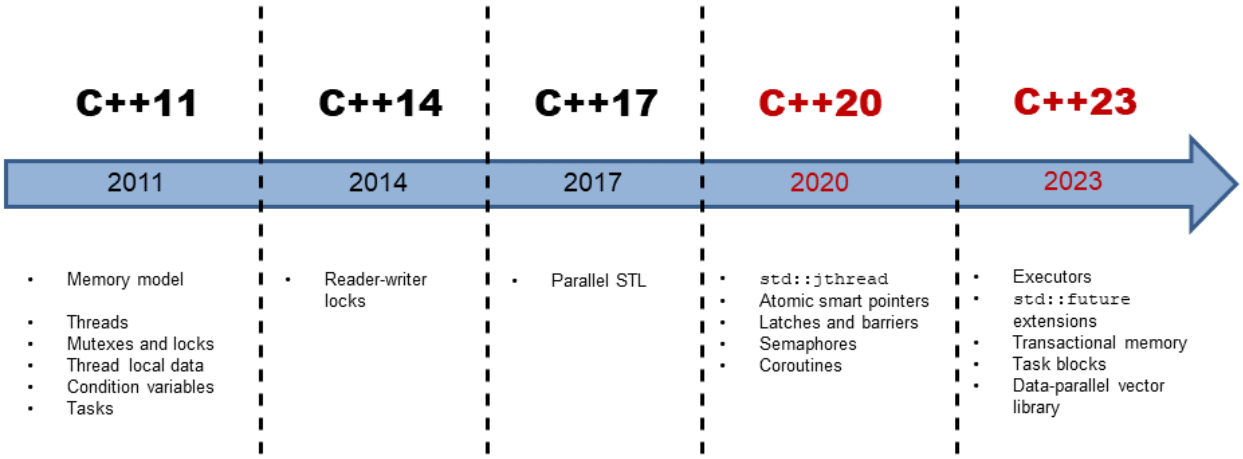
Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#rainer_grimm

Contents

Concurrency with Modern C++

With the publishing of the C++11 standard, C++ got a multithreading library and a memory model. This library has basic building blocks like atomic variables, threads, locks, and condition variables. That's the foundation on which future C++ standards such as C++20 and C++23 can establish higher abstractions. However, C++11 already knows tasks, which provide a higher abstraction than the cited basic building blocks.



Concurrency in C++

Roughly speaking, you can divide the concurrency story of C++ into three evolution steps.

C++11 and C++14: The Foundation

Multithreading was introduced in C++11. This support consists of two parts: A *well-defined* memory model, and a standardised threading interface. C++14 added reader-writer locks to the multithreading facilities of C++.

Memory Model

The foundation of multithreading is a *well-defined* memory model. This memory model has to deal with the following aspects:

- Atomic operations: operations that can be performed without interruption.
- Partial ordering of operations: a sequence of operations that must not be reordered.

- Visible effects of operations: guarantees when operations on shared variables are visible in other threads.

The C++ memory model was inspired by its predecessor: the Java memory model. Unlike the Java memory model, however, C++ allows us to break the constraints of sequential consistency, which is the default behaviour of atomic operations.

Sequential consistency provides two guarantees.

1. The instructions of a program are executed in source code order.
2. There is a global order for all operations on all threads.

The memory model is based on atomic operations on atomic data types (short atomics).

Atomics

C++ has a set of simple atomic data types. These are booleans, characters, numbers and pointers in many variants. You can define your atomic data type with the class template `std::atomic`. Atomics establish synchronisation and ordering constraints that can also hold for non-atomic types.

The standardised threading interface is the core of concurrency in C++.

Multithreading

Multithreading in C++ consists of threads, synchronisation primitives for shared data, thread-local data and tasks.

Threads

A `std::thread` represents an independent unit of program execution. The executable unit, which is started immediately, receives its work package as a callable unit. A callable unit can be a named function, a function object, or a lambda function.

The creator of a thread is responsible for its lifecycle. The executable unit of the new thread ends with the end of the callable. Either the creator waits until the created thread `t` is done (`t.join()`) or the creator detaches itself from the created thread: `t.detach()`. A thread `t` is *joinable* if no operation `t.join()` or `t.detach()` was performed on it. A *joinable* thread calls `std::terminate` in its destructor, and the program terminates.

A thread that is detached from its creator is typically called a daemon thread because it runs in the background.

A `std::thread` is a variadic template. This means that it can receive an arbitrary number of arguments; either the callable or the thread can get the arguments.

Shared Data

You have to coordinate access to a shared variable if more than one thread is using it at the same time and the variable is mutable (non-const). Reading and writing a shared variable at the same time is a data race and therefore undefined behaviour. Coordinating access to a shared variable is achieved with mutexes and locks in C++.

Mutexes

A mutex (*mutual exclusion*) guarantees that only one thread can access a shared variable at any given time. A mutex locks and unlocks the critical section, to which the shared variable belongs. C++ has five different mutexes. They can lock recursively, tentatively, and with or without time constraints. Even mutexes can share a lock at the same time.

Locks

You should encapsulate a mutex in a lock to release the mutex automatically. A lock implements the RAII idiom by binding a mutex's lifetime to its own. C++ has a `std::lock_guard` / `std::scoped_lock` for the simple, and a `std::unique_lock` / `std::shared_lock` for the advanced use cases such as the explicit locking or unlocking of the mutex, respectively.

Thread-safe Initialisation of Data

If shared data is read-only, it's sufficient to initialize it in a *thread-safe* way. C++ offers various ways to achieve this including using a constant expression, a static variable with block scope, or using the function `std::call_once` in combination with the flag `std::once_flag`.

Thread Local Data

Declaring a variable as thread-local ensures that each thread gets its own copy. Therefore, there is no shared variable. The lifetime of a thread-local data is bound to the lifetime of its thread.

Condition Variables

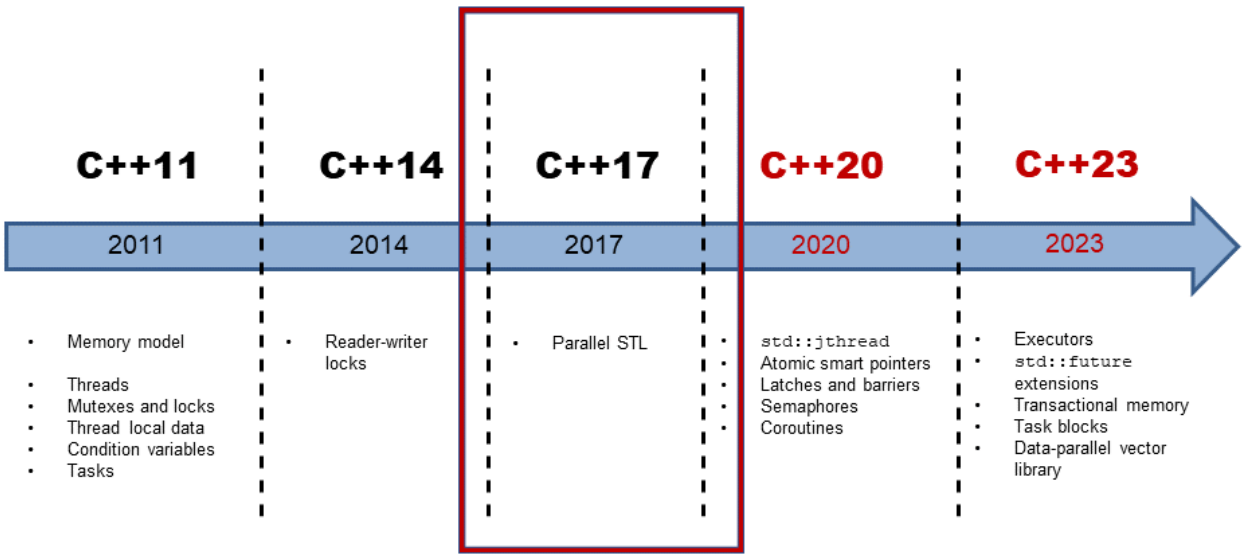
Condition variables enable threads to be synchronized via messages. One thread acts as the sender while the other one acts as the receiver of the message, where the receiver blocks waiting for the message from the sender. Typical use cases for condition variables are producer-consumer workflows. A condition variable can be either the sender or the receiver of the message. Using condition variables correctly is quite challenging; therefore, tasks are often the easier solution.

Tasks

Tasks have a lot in common with threads. While you explicitly create a thread, a task is just a job you start. The C++ runtime automatically handles, in the simple case of `std::async`, the lifetime of the task.

Tasks are like data channels between two communication endpoints. They enable thread-safe communication between threads. The promise as one endpoint puts data into the data channel, the future at the other endpoint picks the value up. The data can be a value, an exception, or simply a notification. In addition to `std::async`, C++ has the class templates `std::promise` and `std::future` that give you more control over the task.

C++17: Parallel Algorithms of the Standard Template Library



Parallel algorithms in C++17

With C++17, concurrency in C++ has drastically changed, in particular the parallel algorithms of the Standard Template Library (STL). C++11 and C++14 only provide the basic building blocks for concurrency. These tools are suitable for a library or framework developer but not for the application developer. Multithreading in C++11 and C++14 becomes an assembly language for concurrency in C++17!

Execution Policy

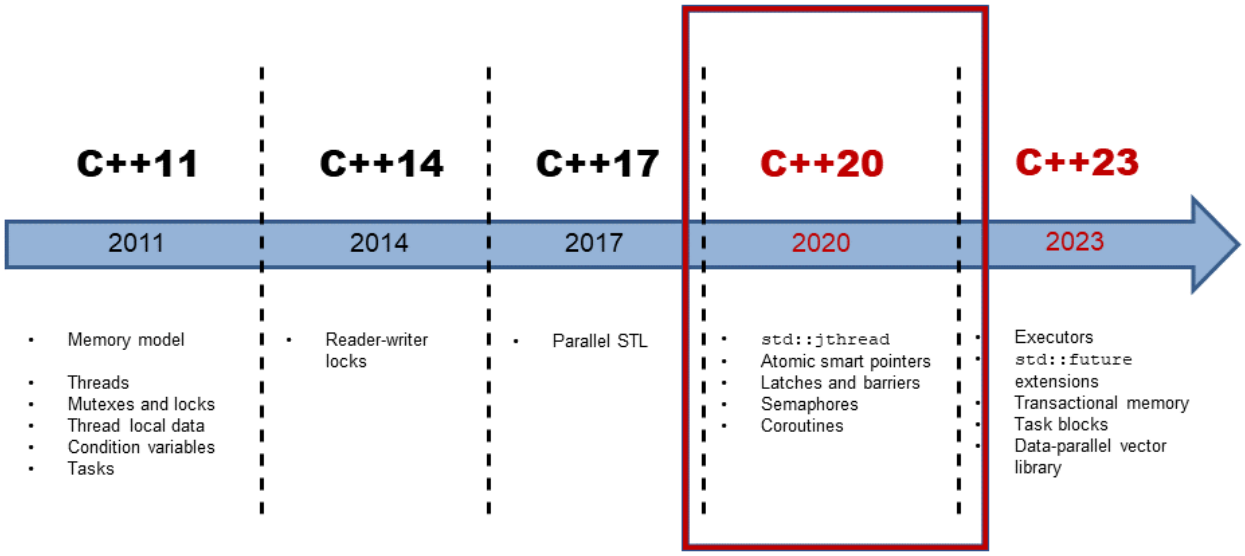
With C++17, most of the STL algorithms are available in a parallel implementation. This makes it possible for you to invoke an algorithm with a so-called execution policy. This policy specifies

whether the algorithm runs sequentially (`std::execution::seq`), in parallel (`std::execution::par`), or in parallel with additional vectorisation (`std::execution::par_unseq`).

New Algorithms

In addition to the 69 algorithms that are available in overloaded versions for parallel, or parallel and vectorised execution, we get eight additional algorithms. These new ones are well suited for parallel reducing, scanning, or transforming.

The Near Future: C++20



Concurrency in C++20

In June 2019, the C++20 standard is worked out. Here are the features, we will get.

`std::jthread`

`std::jthread` is a enhanced replacement for `std::thread`. In addition to `std::thread`, a `std::jthread` can signal an interrupt and can automatically join the started thread.

Atomic Smart Pointers

The smart pointer `std::shared_ptr`[1] and `std::weak_ptr`[2] have a conceptional issue in concurrent programs. They share intrinsically mutable state; therefore, they are prone to data races and thus,

[1] http://en.cppreference.com/w/cpp/memory/shared_ptr
[2] http://en.cppreference.com/w/cpp/memory/weak_ptr

leading to undefined behaviour. `std::shared_ptr` and `std::weak_ptr` guarantee that the incrementing or decrementing of the reference counter is an atomic operation and the resource is deleted exactly once, but neither of them can guarantee that the access to its resource is atomic. The new atomic smart pointers `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>` solve this issue. Both are partial template specialisations of `std::atomic`.

Latches and Barriers

C++14 has no semaphores. Semaphores are used to control access to a limited number of resources. Worry no longer, because C++20 gets latches and barriers. You can use latches and barriers for waiting at a synchronisation point until the counter becomes zero. The difference between latches and barriers is that a `std::latch` can only be used once while a `std::barrier` can be used more than once. Additionally, a `std::barrier` can adjust its counter after each iteration.

Semaphores

A counting semaphore is a special semaphore which has a counter that is bigger than zero. The counter is initialised when the semaphore is created. Acquiring the semaphore decreases the counter and releasing the semaphore increases the counter.

Coroutines

Coroutines are functions that can suspend and resume their execution while maintaining their state. Coroutines are often the preferred approach to implement cooperative multitasking in operating systems, event loops, infinite lists, or pipelines.

Case Studies

After presenting the theory of the memory model and the multithreading interface, I apply the theory in a few case studies.

Calculating the Sum of a Vector

Calculating the sum of an vector can be done in various ways. You can do it sequentially, or concurrently with maximum and minimum sharing of data. The performance numbers differ drastically.

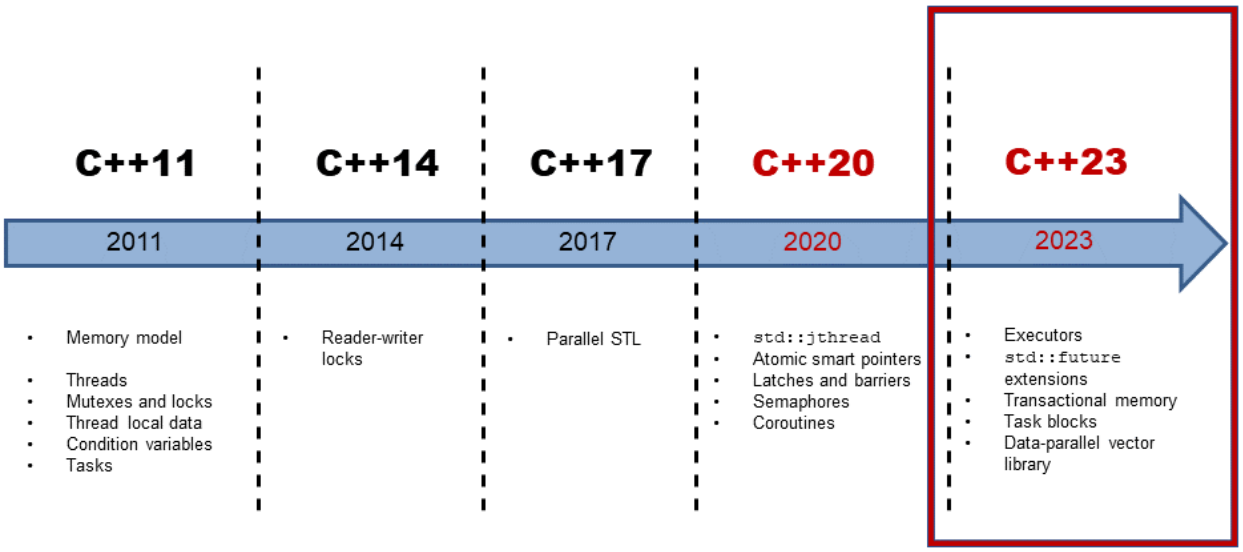
Thread-Safe Initialisation of a Singleton

Thread-safe initialisation of a singleton is the classical use-case for thread-safe initialisation of a shared variable. There are many ways to do it, with varying performance characteristics.

Ongoing Optimisation with CppMem

I start with a small program and successively improve it. I verify each step of my process of ongoing optimisation with CppMem. CppMem[3] is an interactive tool for exploring the behaviour of small code snippets using the C++ memory model.

The Future: C++23



Concurrency in C++20

It is difficult to make predictions, especially about the future (Niels Bohr[4]).

Executors

An executor consists of a set of rules about where, when and how to run a callable unit. They are the basic building block to execute and specify if callables should run on an arbitrary thread, a thread pool, or even single threaded without concurrency. The extended futures, the extensions for networking N4734[5] depend on them but also the parallel algorithms of the STL, and the new concurrency features in C++20/23 such as latches and barriers, coroutines, transactional memory, and task blocks eventually use them.

Extended futures

Tasks called promises and futures, introduced in C++11, have a lot to offer, but they also have a drawback: tasks are not composable into powerful workflows. That limitation does not hold

[3] http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/
[4] https://en.wikipedia.org/wiki/Niels_Bohr
[5] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf

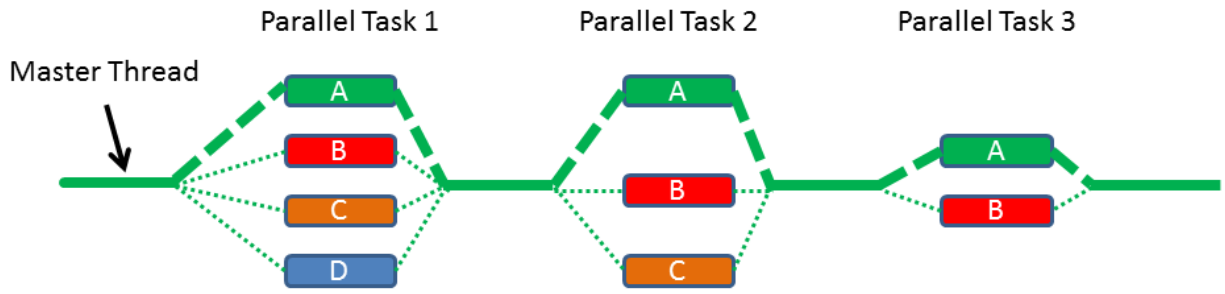
for the extended futures in C++20/23. Therefore, an extended future becomes ready, when its predecessor (`then`) becomes ready, `when_any` one of its predecessors becomes ready, or `when_all` of its predecessors become ready.

Transactional Memory

The transactional memory is based on the ideas underlying transactions in database theory. A transaction is an action that provides the first three properties of ACID database transactions: **A**tomicity, **C**onsistency, and **I**solation. The durability that is characteristic for databases holds not for the proposed transactional memory in C++. The new standard has transactional memory in two flavours: synchronised blocks and atomic blocks. Both are executed in total order and behave as if a global lock protected them. In contrast to synchronised blocks, atomic blocks cannot execute transaction-unsafe code.

Task Blocks

Task Blocks implement the fork-join paradigm in C++. The following graph illustrates the key idea of a task block: you have a fork phase in which you launch tasks and a join phase in which you synchronise them.



Task Blocks

Data-Parallel Vector Library

The data-parallel vector library provides data-parallel (SIMD) programming via vector types. SIMD means that one operation is performed on many data in parallel.

Patterns and Best Practices

Patterns are documented best practices from the best. They "... express a relation between a certain context, a problem, and a solution." Christopher Alexander[6]. Thinking about the challenges of

[6] https://en.wikipedia.org/wiki/Christopher_Alexander

concurrent programming from a more conceptional point of view provides many benefits. In contrast to the more conceptional patterns to concurrency, provides the chapter best practices pragmatic tips to overcome the concurrency challenges.

Synchronisation

A necessary prerequisite for a data races is shared mutable state. Synchronisation patterns boil down to two concerns: dealing with sharing and dealing with mutation.

Concurrent Architecture

The chapter to concurrent architecture presents three patterns. The two patterns Active Object and the Monitor Object synchronise and schedule method invocation. The third pattern Half-Sync/Half-Async has an architectural focus and decouples asynchronous and synchronous service processing in concurrent systems.

Best Practices

Concurrent programming is inherently complicated, therefore having best practices in general, but also for multithreading, and the memory model makes a lot of sense.

Data Structures

Challenges

Writing concurrent programs is inherently complicated. This is particularly true if you only use C++11 and C++14 features. Therefore I describe in detail the most challenging issues. I hope that if I dedicate a whole chapter to the challenges of concurrent programming, you become more aware of the pitfalls. I write about challenges such as race conditions, data races, and deadlocks.

Time Library

The time library is a key component of the concurrent facilities of C++. Often you let a thread sleep for a specific time duration or until a particular point in time. The time library consists of: time points, time durations, and clocks.

CppMem

CppMem is an interactive tool to get deeper inside into the memory model. It provides two very valuable services. First, you can verify your lock-free code and second, you can analyse your lock-free code and get, therefore, a more robust understanding of your code. I often uses CppMem in this book. Because the configuration options and the insights of CppMem are quite challenging, the chapter gives you a basic understanding of CppMem.

Glossary

The glossary contains non-exhaustive explanations on the most essential terms.