Parallel Programming in C++

Jakub Marecek and team

version March 8, 2023
https://github.com/jmarecek/book-parallel-cpp

March 8, 2023

Original copy of this book can be obtained at https://github.com/jmarecek/book-parallel-cpp.

This copy of the book is version March 8, 2023.

Preface

These lecture notes are intended for PDV course at the Czech Technical University, but could perhaps be of more widespread interest.

About the author

Jakub Marecek first taught a course on "Advanced C++ Programming" in 1999 in Brno, the Czech Republic. Since then, he has worked at the University of Nottingham, ARM Ltd., the University of Edinburgh, IBM Research, and the University of California, Los Angeles.

Feedback

If you like these lecture notes, please do send us feedback: positive, negative. Anything goes!

Why cc-by-sa-nc

Contents

Chapter 1

Introduction

In a modern desktop computer equipped with a GPGPU, a single-threaded application cannot easily make use of more than 0.05 % of the available performance. One hence has to learn the dark art of parallel programming to make a full use of the available performance.

Indeed, consider an AMD Threadripper 3990X with NVIDIA GeForce RTX 4090 Ti. There, a single threaded application can utilise circa 49 GFLOPS, which is less than 0.05 % of the overall performance, when one runs a single-threaded application once. A multi-threaded application can utilise perhaps 3732 GFLOPS. A multi-threaded application not making use of the GPGPU can hence make use of less than 4 % of the overall performance. In contrast, a multi-threaded application can make use of almost 100 TFLOPS. (To unlock this performance, one needs to use recent versions of OpenMP, SYCL, or vendor- or application-specific interfaces such as CUDA, OpenGL, or DirectX Compute. We will focus on OpenMP and SYCL in subsequent Chapters.) Similar breakdown is available in a modern mobile phone (with a substantial mobile GPGPU, such ARM Mali-G710 MP7 in Google Tensor G2) and modern supercomputers.

In the first three chapters, this we will learn to design multi-threaded applications. First with C++23, and subequently with OpenMP ("just in case". In the following chapters, we will extend this to GPGPU-enabled systems and SYCL, and illustrate in a number of application domains.

Chapter 2

The Concepts

Parallelism means two or more tasks can be executed simultaneously. This is an option, which the compiler and operating system and processor can exercise, but does not come with any guarantees. Often, this means no shared variables or other resources, and need not require any synchronization primitives.

Concurrency means that two or more tasks start, run, and complete in overlapping time periods, while sharing some resources. If two tasks concurrently set shared variable x to 1 and 2, it is not clear what value it would have, subsequently. More broadly, concurrent access to a mutable shared memory can result in issues without the use of synchronization primitives ("data race") and with the use of synchronization primitives ("deadlock").

2.0.1 Data race

When we need to ensure mutual exclusion in access to two or more shared variables, e.g., read value of one of the variables and add it to another variable, we may need to use some synchronization primitives (e.g., mutexes). Without the use of synchronization primitives, we are facing the risk of a data race.

For example, consider the situation in a bank, where there are two clients: Alice and Bob. Transaction T1: Bob has $100 in his account, but will be paying a $50 bill. At the same time, in Transaction T2, Alice will be paying $100 to Bob. Depending on the ordering of the reading and writing operations, one may obtain several outcomes:

- Transaction T1 will read $100 valued of Bob's account. Transaction T2 will read $100 value. Transaction T1 will write $200. Transaction T2 will write $50 value.
- Transaction T1 will read $100 valued of Bob's account. Transaction T1 will write $50. Transaction T2 will read $50 value.Transaction T2 will write $150 value.

- Transaction T1 will read \$100 valued of Bob's account. Transaction T2 will read \$100 value. Transaction T1 will write \$50. Transaction T2 will write \$200 value.

- Transaction T2 will read \$100 value. Transaction T2 will write \$200 value. Transaction T1 will read \$200 valued of Bob's account. Transaction T1 will write \$150.

Either Bob or the bank could be \$100 short.

2.0.2 Deadlock

When we need to ensure mutual exclusion in access to two or more shared variables, e.g., two temporary results associated with two mutexes, one may naively lock the first mutex first, and subsequently lock the other mutex. This, however, can lead to a deadlock. One needs to lock both mutexes at the same time.

Naively, one could run:

Locking multiple mutexes at once.

```
void thread_operation(){
    std::scoped_lock l(mutex1,mutex2);
     ...
    complicated_task();
}\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

Open in Compiler Explorer

In theory, a deadlock (Czech: "problém uváznutí") can occur when:

- each lock is owned by one thread
- each thread has locked at least one lock and needs to lock at least one more lock
- it is impossible to remove the lock ownership
- there is a cyclic dependency among the lock-using threads.

2.0.3 Concurrent programming

There are two essential models for concurrent programming: shared memory and message passing. In sharing memory, we have broadly four options:

- Confinement: Do not share memory between threads. This is often impossible.
- Immutability: Do not share any mutable data between threads.

- Thread-safe code: Use data types with additional guarantees for storing any mutable data shared between threads, or even better, use implementations of algorithms that are already parallelized and handle the concurrency issues for you. For example in C++, one can use the standard template library with a suitable execution policy. In particular, the header `execution` defines objects `std::execution::seq`, `std::execution::par`, `std::execution::par_unseq`, which can be passed as the first argument of any standard algorithm, e.g., `std::vector<int> my_data; std::sort(std::execution::par,my_data.begin(),my_data.end` See Section 3.3.1 for more examples.

- Synchronization: Use synchronization primitives to prevent accessing the variable at the same time. This option is explored in this chapter in more detail.

Eventually, we will see that message passing can be implemented using the synchronization primitives and may be the least challenging to use correctly.

2.1 Structuring code: Processes, Threads, Tasks, Coroutines

Processes, threads, tasks, and coroutines execute instructions.

A *process* provides all of the prerequisites for executing instructions: loads an executable program, sets up a virtual address space, the environment (e.g. environment variables and a security context), the process control block (PCB, often stored in registers of the processor and on a per-process stack in kernel memory), opens handles to system objects (e.g., files, sockets), and often much more. In some sense, one can imagine "a virtual machine".

Within a particular process, there is at least one *thread*. All threads of a particular process share the same virtual address space and handles to system objects. Each thread, independently, operates its own context (registers, stack, exception handlers). Unless declared otherwise, threads of a particular process share memory and are allocated "time slices" by the operating system. This can be seen as a "virtual processor" within a "a virtual machine" of a process, often with no guarantees on the time slicing.

All modern operating systems (OS) are multitasking, i.e., running multiple processes with the operating system forcibly interrupting the run one one process to execute another process after a certain amount of time ("preemptive scheduling"). Switching between the processes involves swapping the process control block (PCB). In Intel architectures, this is known as the task state segment (TSS), and there is hardware support for the switch. AMD64 does not support task switches in hardware. Consequently, neither

Windows nor Linux kernels utilize the hardware support for the switch. Context switching thus has non-trivial impact on performance.

Most modern processors are multi-core and support multithreading in some form. This means that one each process can execute multiple "hardware threads" and there is some support for switching between those. In Intel architectures, hyper-threading means each hardware core can execute multiple threads, e.g., two, to take advantage of idle time (e.g., loading data, network communications).

Most modern operating systems *do not guarantee* fairness among the threads.

Within a particular thread, one may utilize multiple *coroutines*, which can be seen as subroutines that can run in multiple steps, but sometimes can serve as a light-weight alternative to hardware threads. Coroutines can be called, can return when completed, but also can suspend themselves, yielding control and partial results, and be resumed by another co-routine. Typical uses involve generators andfactories and various other concepts within "lazy evaluation", as well as event-driven architectures within cooperative multitasking.

That is: two coroutines within one thread never run in parallel, but one can have the runs of two or more coroutines interleaved. We can suspend a co-routine in one thread and resume it within another thread.

As it turns out, the "context switch" with user-level threads has a similar cost to a function call or suspending a coroutine (`co_yield`). Indeed, coroutines are typically implemented with user-level threads, which leads to cheaper context-switch compared with hardware threads. Within the user-level threads, one can distinguish stackful and stackless versions, where coroutine state is saved on the heap (as in C++).

A *task* is a rather abstract unit of work, e.g., a function, which can be executed by any thread, but often allocated to one of a many threads within a pool.

2.2 Hardware support

2.2.1 Memory order

First, one should like to understand several options for implementing synchronization primitives, known as "memory orders". All guarantee atomicity and modification-order consistency.

In `memory_order_relaxed`, no further guarantees are provided and specifically no order is imposed on concurrent memory accesses. This is also how weakly-ordered architectures (e.g. ARM) operate, by default: if two threads access shared memory the load in one thread does not have to read a value written by another thread very recently.

With `memory_order_release` and `memory_order_acquire` specifiers, we force weakly-ordered achitectures to behave closer to strongly-ordered architectures (e.g., Intel). If one thread writes into shared memory atomically with `memory_order_release` and another thread reads the memory atomically with `memory_order_acquire`, the load in the second thread is guaranteed to read the value written by another thread.

(In earlier version of C++ standard, there were further memory models defined for the sake of DEC Alpha architecture. At least `memory_order_consume` is deprecated as of C++17.)

With `memory_order_seq_cst`, we additionally require a single total ordering of all modifications (with this specifier). A load with this specifier gets its value either from the last store with this specifier or from some store without this specifier that did not precede the most recent `memory_order_seq_cst store`. This is the default option.

See https://arxiv.org/abs/1803.04432 for a nice overview and https://www.youtube.com/watch?v=A_vAG6LIHwQ&ab_channel=ACCUConference for a nice lecture.

2.2.2 Compare and swap

Synchronization primitives are typically implemented using some hardware instructions, typically "compare-and-swap". In locking, these make it possible to test whether the lock is free, and if so, acquire the lock within a single operation that the hardware guarantees to execute atomically.

The atomic compare and swap" (CAS) instruction dereferences a pointer to an atomic variable and compares its value against a given value. If these is a match, it replaces the atomic variable with a given new value. That is:

- we declare an atomic variable (and a pointer to it)
- (*) we save the value of an atomic variable to a local, private variable (by dereferencing the pointer)
- based on the saved value in a local, private variable, we compute the new value, which we would like to store in the atomic variable
- the CAS instruction is used. If the current value matches the value saved in the local, private variable, we will overwrite the value with the newly computed value. If the current value no longer matches the value saved in the local, private variable, we wait (some random and growing from a small starting value) and repeat from (*).

In C++, the `atomic` header defines two variants of "compare and swap" and a specialization thereof for pointers:

Definitions of two variants of "compare and swap".

```
1  // based on
   ↪  https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/atomic
2
3   template<typename _Tp> struct atomic {
4     using value_type = _Tp;
5
6  [...]
7
8     bool compare_exchange_strong(_Tp& __e, _Tp __i, memory_order
      ↪  __s,
9                     memory_order __f) noexcept {
10      return __atomic_impl::__compare_exchange(_M_i, __e, __i,
        ↪  false, __s, __f);
11    }
12
13    bool compare_exchange_weak(_Tp& __e, _Tp __i, memory_order __s,
14                 memory_order __f) noexcept {
15      return __atomic_impl::__compare_exchange(_M_i, __e, __i, true,
        ↪  __s, __f);
16       }
17
18 [...]
19    };
20
21    // Partial specialization for pointer types.
22    template<typename _Tp> struct atomic<_Tp*> {
23        using value_type = _Tp*;
24        using difference_type = ptrdiff_t;
25
26 [...]
27
28      bool compare_exchange_strong(__pointer_type& __p1,
        ↪  __pointer_type __p2, memory_order __m1, memory_order __m2)
        ↪  noexcept {
29        return _M_b.compare_exchange_strong(__p1, __p2, __m1, __m2);
30      }
31
32 [...]
33
34    };
```

Open in Compiler Explorer

The former is called with the desired value `i`, the new value i, and the memory order to consider if there is a match and if there is no match. Typically, if there is a match and we want to replace the value, we may use `std::memory_order_release`. If there is no match, we are just reading the value and `std::memory_order_acquire` would suffice. In the latter variant, we pass two pointers.

The difference between the "weak" and "strong" variant is in that the weak variant may return false even if there is a match, in certain cases, but can be much faster in certain architectures. This notably entails ARM architectures (RISC-V and MIPS), where the weak variant will be implemented using the so called load-link/store-conditional pair of instructions (load exclusive register / ldxr and store exclusive register / stxr in ARM version 8). These are much faster than the comparable instructions issuing a barrier (ldaxr/stlxr in ARM version 8). All four ARM instructions utilize only two registers, compared to three registers for CAS proper in Intel architectures (Compare and exchange / cmpxchg since 80486 and cmpxchg8b and cmpxchg16b since Intel Core 2). On recent Intel and AMD processors, cmpxchg is only marginally slower than a non-cached load.

In C++, the *only* synchronization primitive that is guaranteed to be hardware implemented is a particular atomic boolean type, which is known as `std::atomic_flag`. Unlike all specializations of `std::atomic`, it is guaranteed to be lock-free. Prior to C++20, it has been very restricted, because there was no way to check the value of `std::atomic_flag` without setting it. C++20 adds method `test()`.

2.3 Synchronization primitives

Synchronization primitives make it possible to synchronize or restrict access of multiple threads to some resources (e.g., global variables, file handles, sockets). You can use them as an interface, without knowing their implementation.

2.3.1 Raw synchronization primitives

Lock, Mutex, Semaphore, Atomic, Memory Fence, Condition Variable are synchronization primitives, which make it possible to synchronize or restrict access of multiple threads to some resources.

Lock is a very general term for a synchronization primitive. Mutexes are usually used by one thread only, while semaphores are shared between multiple threads. The *binary semaphore* is the most simple type of a lock, which provides exclusive access for both reading and writing. *counting semaphore* limits the use of a single resource by at most a given number of threads.

A spinlock, the thread simply waits ("spins") until the lock becomes available. This is efficient if threads are blocked for a short time, because it avoids the overhead of operating system process re-scheduling. It is inefficient if the lock is held for a long time, or if the progress of the thread that is holding the lock depends on preemption of the locked thread. An intentionally simplistic implementation is presented below.

A silly implementation of a spin lock.

```cpp
#include <thread>
#include <queue>
#include <iostream>
#include <atomic>
#include "atomic4.h"

int main() {
    std::queue<int> numbers;
    SpinLock numbers_lock;
    int n = 0;
    for (n = 0; n < 10; ++n) {
        numbers.push(n);
    }
    for (n = 0; n < 2; n++) {
        std::jthread t{[n, &numbers, &numbers_lock] () {
            numbers_lock.lock();
            int val = numbers.front();
            numbers.pop();
            numbers_lock.unlock();
        }};
    }
}
```

Open in Compiler Explorer

2.3.2 Further synchronization features

Fences help order non-atomic and atomic memory accesses, without any associated operations. On Intel architectures (including x86-64), `atomic_thread_fence` do not issue any instructions, except for `std::atomic_thread_fence(std::memory_order::seq_cst)`.

Barrier provides a thread-coordination mechanism that blocks a group of threads until all threads in that group have reached the barrier. Such a barrier can be used repeatedly to wait until a number of threads have finished their operations.

Latch and is a downward counter, whose initial value is initialized and then threads may block on the latch until the counter is zero. One thread may decrement a latch multiple times, but no thread can increment the latch. Thus, it serves as a single-use barrier.

We will also see synchronized output streams. The synchronized buffer is flushed only when the destructor of the synchronized buffer is called, but provides for guarantees of atomicity for the access. (That is, `endl` and `std::flush` no longer flush!)

Chapter 3

The Syntax in C++23

3.1 Threads, Tasks, Coroutines

3.1.1 C++11 thread

C++11 had a very basic support for threads, in terms of `std::thread` of header `thread`. The thread starts running once the constructor is called. The object is not CopyConstructible nor CopyAssignable. The challenge in C++11 threads is that one needs to call `join` or `detach` prior to the destructor being called. If neither was called, the program was `std::aborted`. Prior to calling either, one needs to check whether the thread is `joinable()`. At the same time, it was almost impossible to handle exceptions while being able to call `join` correctly. The use of the C++11 `thread` is thus considered harmful and we will present only two short examples.

An example of the use of a C++11 thread.

```
1  #include <iostream>
2  #include <chrono>
3  #include <thread>
4
5  using namespace std::this_thread;
6  using namespace std::chrono_literals;
7
8  void A() {
9      std::cout << "a";
10     sleep_for(5s);
11     std::cout << "A";
12 }
13
14 int main() {
15     std::thread t(A);
16     t.join();
17 }
```

Open in Compiler Explorer

An example of the use of a C++11 thread.

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

using namespace std::this_thread;
using namespace std::chrono_literals;

void A() {
    std::cout << "a";
    sleep_for(5s);
    std::cout << "A";
}

void B() {
    std::cout << "b";
    sleep_for(1s);
    std::cout << "B";
}

void C() {
    std::cout << "c";
    std::thread t(A);
    t.detach();
    std::thread u(B);
    u.join();
    std::cout << "C";
}

int main() {
    C();
    std::thread t(B);
    t.join();
    A();
}
```

Open in Compiler Explorer

3.1.2 C++20 jthread

C++20 adds a new class `jthread` ("joining threads"), which does not require a call to `join` or `detach`. Instead, the destructor waits for completion of the code ("joins") automatically.

This is an example of the "resource acquisition is initialization" idiom, which is generally one of the best practices in C++. In RAII, the resource allocation is tied to an object's lifetime and is hence a class invariant. In a constructor, one allocates the resources. In a destructor, one releases the resources. There is no risk of a resource leak.

An example of the use of jthread.

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

void Hello();

int main(int argc, char* argv[]) {
    std::vector<std::jthread> threads;
    for (int cnt=0; cnt < 10; cnt++) {
        threads.push_back(std::jthread(Hello));
    }
    return 0;
}

void Hello() {
  using namespace std::chrono_literals;
  std::this_thread::sleep_for(2s);
}
```

Open in Compiler Explorer

Notice that the above example would very like result in abnormal program termination, if we changed jthread to thread. (Why?) When we use standard output, it is prudent to wrap it in a syncstream:

An example of the use of jthread.

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

void Foobar(int cnt);

int main(int argc, char* argv[]) {
    std::vector<std::jthread> threads;
    for (int cnt=0; cnt < 10; cnt++) {
        threads.push_back(std::jthread(Foobar, cnt));
    }
    std::osyncstream(std::cout) << "Main thread" << std::endl;
    return 0;
}

void Foobar(int cnt) {
    std::osyncstream(std::cout) << "Thread " << cnt << std::endl;
}
```

Open in Compiler Explorer

Rather commonly, one uses the lambda function to define the thread. (This is the `[]()`.)

An example of the use of jthread.

```cpp
#include <iostream>
#include <syncstream>
#include <thread>
using namespace std::chrono_literals;

int main() {

    auto t1 = std::jthread([](){
        std::osyncstream(std::cout) << "Another thread" <<
        ↪  std::endl;
        std::this_thread::sleep_for(1s);
    });

    std::this_thread::sleep_for(2s);
    std::osyncstream(std::cout) << "Main thread" << std::endl;
}
```

Open in Compiler Explorer

Passing arguments to threads is, nevertheless, very useful. Notably, when we pass the first argument of type `std::stop_token token`, we request the thread to stop its execution by calling `request_stop()` on the jthread object:

An example of the use of jthread.

```cpp
#include <iostream>
#include <syncstream>
#include <thread>
using namespace std::chrono_literals;

int main() {
    auto t1 = std::jthread([](std::stop_token token){
        while (!token.stop_requested()) {
            std::osyncstream(std::cout) << "A thread";
            std::this_thread::sleep_for(1s);
        }
        std::osyncstream(std::cout) << "Stop requested";
    });

    std::this_thread::sleep_for(2s);

    std::osyncstream(std::cout) << "Main thread";
    t1.request_stop();
}
```

Open in Compiler Explorer

More complicated procedures for the stopping of the thread possible. One can define `std::stop_callback` object inside the thread, whose constructor takes the stop token (`std::stop_token`) and a function. The function (in the example below another lambda) gets executed, when the thread is requested to stop via the `std::stop_token`:

An example of the use of jthread.

```cpp
#include <iostream>
#include <syncstream>
#include <atomic>
#include <thread>
using namespace std::chrono_literals;

int main() {

    auto t = std::jthread([](std::stop_token token) {
        std::osyncstream(std::cout) << "Thread " <<
        ↪  std::this_thread::get_id() << std::endl;
        std::atomic<bool> flag = false;
        std::stop_callback callback(token, [&flag]{
            std::osyncstream(std::cout) << "Stop requested" <<
            ↪  std::endl;
            flag = true;
        });
        while (!flag) {
            std::this_thread::sleep_for(1s);
        }
    });

    std::osyncstream(std::cout) << "Main thread" << std::endl;
    std::this_thread::sleep_for(3s);
    t.request_stop(); // runs all callbacks!
}
```

Open in Compiler Explorer

(For a substantially more complex example, see https://en.cppreference.com/w/cpp/thread/stop_callback.)

3.1.3 Coroutines

In C++23 and subsequent versions, we hope to see some standard syntax for defining coroutines (cf. P2502), such as:

An example of the use of coroutines, which currently does not compile in GCC 12.2.

```
#include <coroutine>
#include <generator>
#include <iostream>
#include <syncstream>

std::generator<int> work() {
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}

int main() {
    for (int i : work()) {
        std::osyncstream(std::cout) << ch << '\n';
    }
}
```

Open in Compiler Explorer

Already, there are three new keywords:

- `co_await awaiter` suspends computation and block the co-routine until the computation is resumed by another co-routine calling "resume" method of the present coroutine. In the process, it tests whether it is possible to suspend the computation using an awaiter (such as `std::suspend_always{};`) and, if so, saves all local variables to a heap-allocated handle.
- `co_yield` yields a value and suspends computation as above, and
- `co_return` returns a value. (There is no notion of an optional return type in-built.)

  Unfortunately, defining the coroutine in C++20 take some more effort. In particular, it requires:

  – defining the behaviour of the coroutine, which is known as a `promise` (different from `std::promise`), and requires one returns the type used to access the state of the coroutine on the heap, which is known as the handle,

  – defining how to store the state of the coroutine on the heap, using template class `std::coroutine_handle` parametrized by the promise

  Clearly, one needs to declare one, define the other, and then return to declare the first one. We will see how to do this later. Optionally, we

can also define an awaiter, which controls suspension and resumption behaviour.

Another difficulty in using coroutines is the fact that the coroutine may live longer than the scope it has been called from. It is hence *not* advisable to pass by reference, except perhaps `std::ref` or `std::cref`. One can either pass by value or pass, e.g., `std::unique_ptr`:

An example of the use of coroutines, which currently does not compile in GCC 12.2.

```
1  // inspired by
   ↪  https://www.incredibuild.com/blog/cpp-coroutines-lets-play-with-them
2
3  #include <coroutine>
4  #include <generator>
5  #include <iostream>
6  #include <syncstream>
7  #include <memory>
8  #include <string>
9
10 std::generator<char> split-by-value(std::string s) {
11     for (char ch : ps) {
12         co_yield ch;
13     }
14 }
15
16 std::generator<char>
   ↪  split-by-uniqueptr(std::unique_ptr<std::string> ps) {
17     for (char ch : *ps) {
18         co_yield ch;
19     }
20 }
21
22 int main() {
23     for (char ch : split-by-value("test")) {
24         std::osyncstream(std::cout) << ch << '\n';
25     }
26     for (char ch :
       ↪  split-by-uniqueptr(std::make_unique<std::string>("west")))
       ↪  {
27         std::osyncstream(std::cout) << ch << '\n';
28     }
29 }
```

Open in Compiler Explorer

Below, we will see two examples of the use of coroutines that compile with GCC 11 and 12 (enable -std=c++2b -fcoroutines). These focus on defining the promise class to be used with template class `std::coroutine_handle<promise>`. The promise class defines the

behaviour of the coroutine by implementing methods:

- `coroutine get_return_object()` is called to initialize the coroutine and create the coroutine handle, which can be the rather formulaic `coroutine_handle::from_promise(*this)`;
- `std::suspend_always initial_suspend()`, suggests whether the coroutine starts right after initialization (`std::suspend_never()`) or upon resumption (`std::suspend_always()`). (Both awaiterers are described below.)
- `std::suspend_always final_suspend() noexcept`, which can be rather formulaic `std::suspend_always()`
- `void return_void()` or `void return_value(const auto& value)`, which is called upon reaching the end of the coroutine and upon reaching `co_return`. The latter (`return_value`) often just stores the result locally.
- `void unhandled_exception()`, which can be rather formulaic `std::terminate()`, or can save the exception via `std::current_exception()`.

The promise class is instantiated for each instance of the coroutine, and its methods are called as follows:

A schema of the coroutine, in terms of its calls of the methods of the promise class.

```
1  {
2    co_await promise.initial_suspend();
3    try {
4      ...
5    }
6    catch (...) {
7      promise.unhandled_exception();
8    }
9    // finally
10   co_await promise.final_suspend();
11 }
```

Open in Compiler Explorer

Once we have a promise class, we can specialize template class `std::coroutine_handle`, which can be seen as the equivalent of a pointer and its method "destroy" as equivalent to a "free", and use the handle specialized to our own promise class to define a promise class:

An example of the use of coroutines.

```cpp
// https://en.cppreference.com/w/cpp/language/coroutines

#include <coroutine>
#include <iostream>
#include <syncstream>

struct promise;

struct coroutine : std::coroutine_handle<promise> {
    using promise_type = struct promise;
};

struct promise {
    coroutine get_return_object() { return {coroutine::from_promise(*this)}; }
    std::suspend_always initial_suspend() noexcept { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() {}
};

int main() {
    coroutine h = [](int i) -> coroutine {
        std::osyncstream(std::cout) << i;
        co_return;
    }(0);
    h.resume();
    h.destroy();
}
```

Open in Compiler Explorer

Sometimes, we also store the promise type in a `promise_type` type member, and disable (`= delete`) copy and move constructors.

Awaiters

Finally, let us consider awaiters, which can be called when a coroutine is suspended or resumed. Key methods of an awaiter include:

- `await_ready()` is called immediately before suspension of a coroutine. If it returns `true`, the coroutine will not be suspended.
- `await_suspend(handler)` is called immediately after the suspension of the coroutine. The `handler` of type `std::coroutine_handle` can be used to pass the state of the coroutine (e.g., to another

thread).

- `await_resume()` is called when the coroutine is resumed after a successful suspension. If it returns a value, this will be returned by the `co_await` routine.

The two awaiters we have seen so far (`std::suspend_never()` and `std::suspend_always()`) just returned boolean constants in `await_ready()`:

An example of two standard awaiters.

```
1  //
   ↪  https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/corout
2
3    // 17.12.5 Trivial awaitables
4    /// [coroutine.trivial.awaitables]
5    struct suspend_always
6    {
7      constexpr bool await_ready() const noexcept { return
       ↪  false; }
8      constexpr void await_suspend(coroutine_handle<>) const
       ↪  noexcept {}
9      constexpr void await_resume() const noexcept {}
10   };
11
12   struct suspend_never
13   {
14     constexpr bool await_ready() const noexcept { return true;
       ↪  }
15     constexpr void await_suspend(coroutine_handle<>) const
       ↪  noexcept {}
16     constexpr void await_resume() const noexcept {}
17   };
```

Open in Compiler Explorer

By defining `await_transform()` in the promise type, the compiler will use `co_await promise.await_transform(<expr>)` instead of any call of `co_await <expr>` in the coroutine.

An example defining a Generator (1/2).

```cpp
// This code is based on the work of Simon Toth,
//
    https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d

// The caller-level type
struct Generator {
    // The coroutine level type
    struct promise_type {
        using Handle = std::coroutine_handle<promise_type>;

        Generator get_return_object() {
            return Generator{Handle::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return
            {}; }
        std::suspend_always yield_value(int value) {
            current_value = value;
            return {};
        }
        void unhandled_exception() { }
        int current_value;
    };

    explicit Generator(promise_type::Handle coro) :
        coro_(coro) {}
    // Make move-only
    Generator(const Generator&) = delete;
    Generator& operator=(const Generator&) = delete;
    Generator(Generator&& t) noexcept : coro_(t.coro_) {
        t.coro_ = {}; }
    Generator& operator=(Generator&& t) noexcept {
        if (this == &t) return *this;
        if (coro_) coro_.destroy();
        coro_ = t.coro_;
        t.coro_ = {};
        return *this;
    }

    int get_next() {
        coro_.resume();
        return coro_.promise().current_value;
    }

private:
    promise_type::Handle coro_;
};
```

Open in Compiler Explorer

An example defining a Generator (2/2).

```
1  // This code is based on the work of Simon Toth,
2  //
   ↪  https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d
3
4  #include <coroutine>
5  #include <iostream>
6  #include "coroutines4.h"
7
8  Generator myCoroutine() {
9      int x = 0;
10     while (true) {
11         co_yield x++;
12     }
13 }
14
15 int main() {
16     auto c = myCoroutine();
17     int x = 0;
18     while ((x = c.get_next()) < 10) {
19         std::cout << x << "\n";
20     }
21 }
```

Open in Compiler Explorer

Eventually, in C++23 and C++26, the support for message-passing architectures based on https://github.com/lewissbaker/cppcoro should be available. For a small sample, see a custom implementation here:

An example of the use of coroutines in message-passing architectures (1/3).

```cpp
// This code is based on the work of Simon Toth,
//
    ↪ https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d

class Event {
 public:

    Event() = default;

    Event(const Event&) = delete;
    Event(Event&&) = delete;
    Event& operator=(const Event&) = delete;
    Event& operator=(Event&&) = delete;

    class Awaiter;
    Awaiter operator co_await() const noexcept;

    void notify() noexcept;

 private:
    friend class Awaiter;
    mutable std::atomic<void*> suspendedWaiter{nullptr};
    mutable std::atomic<bool> notified{false};

};

class Event::Awaiter {
 public:
  Awaiter(const Event& eve): event(eve) {}

  bool await_ready() const;
  bool await_suspend(std::coroutine_handle<> corHandle)
    ↪ noexcept;
  void await_resume() noexcept {}

 private:
    friend class Event;

    const Event& event;
    std::coroutine_handle<> coroutineHandle;
};

struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return
        ↪ {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};
```


Open in Compiler Explorer

An example of the use of coroutines in message-passing architectures (2/3)

```
// This code is based on the work of Simon Toth,
//
↪ https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d

bool Event::Awaiter::await_ready() const {
    // allow at most one waiter
    if (event.suspendedWaiter.load() != nullptr){
        throw std::runtime_error("More than one awaiter is not
        ↪ valid");
    }

    return event.notified; // `false' suspends the coroutine
}

bool Event::Awaiter::await_suspend(std::coroutine_handle<>
↪ corHandle) noexcept {

    coroutineHandle = corHandle;

    if (event.notified) return false;

    // store the waiter for later notification
    event.suspendedWaiter.store(this);

    return true;
}

void Event::notify() noexcept {
    notified = true;

    // try to load the waiter
    auto* waiter =
    ↪ static_cast<Awaiter*>(suspendedWaiter.load());

    // check if a waiter is available
    if (waiter != nullptr) {
        // resume the coroutine => await_resume
        waiter->coroutineHandle.resume();
    }
}

Event::Awaiter Event::operator co_await() const noexcept {
    return Awaiter{ *this };
}

Task receiver(Event& event) {
    auto start = std::chrono::high_resolution_clock::now();
    co_await event;
    std::cout << "Got the notification! " << std::endl;
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Waited " << elapsed.count() << " seconds."
    ↪ << std::endl;
}
```


Open in Compiler Explorer

An example of the use of coroutines in message-passing architectures (3/3)

```
1  // This code is based on the work of Simon Toth,
2  //
   ↪  https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d
3
4  #include <coroutine>
5  #include <chrono>
6  #include <iostream>
7  #include <functional>
8  #include <string>
9  #include <stdexcept>
10 #include <atomic>
11 #include <thread>
12
13 #include "coroutines5a.h"
14 #include "coroutines5b.h"
15
16 using namespace std::chrono_literals;
17
18 int main(){
19
20     std::cout << std::endl;
21
22     std::cout << "Notification before waiting" << std::endl;
23     Event event1{};
24     auto senderThread1 = std::thread([&event1]{
       ↪  event1.notify(); });
25     auto receiverThread1 = std::thread(receiver,
       ↪  std::ref(event1));
26
27     receiverThread1.join();
28     senderThread1.join();
29
30     std::cout << std::endl;
31
32     std::cout << "Notification after 2 seconds waiting" <<
       ↪  std::endl;
33     Event event2{};
34     auto receiverThread2 = std::thread(receiver,
       ↪  std::ref(event2));
35     auto senderThread2 = std::thread([&event2]{
36         std::this_thread::sleep_for(2s);
37         event2.notify();
38     });
39
40     receiverThread2.join();
41     senderThread2.join();
42
43     std::cout << std::endl;
44
45 }
```

Open in Compiler Explorer

For further nice examples, see also Boost Asio, e.g., https://www.boost.org/doc/libs/1_78_0/doc/html/boost_asio/example/cpp20/channels/throttling_proxy.cpp, as discussed, e.g., at https://www.youtube.com/watch?app=desktop&v=ZNttI_WswMU&ab_channel=ACCUConference. For more details, see Simon Toth's Complete guide at https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d. For the technical specification, see https://github.com/GorNishanov/coroutines-ts.

3.2 Synchronisation Primitives

3.2.1 Atomic Variables

Since C++11, there is an excellent support for atomic variables in header `<atomic>`. The primary template can be instantiated with types that are TriviallyCopyable, CopyConstructible, and CopyAssignable. First, let us consider two simple examples:

An example of the use of atomic variables.

```
1  #include <iostream>
2  #include <atomic>
3  #include <thread>
4
5  std::atomic<int> i(0);
6
7  int main() {
8      auto t1 = std::jthread([](){
9          int j;
10         do { j = i; }
11         while (j == 0);
12         std::cout << j << std::endl;
13     });
14     auto t2 = std::jthread([](){
15         i = 1;
16     });
17     return 0;
18 }
```

Open in Compiler Explorer

An example of the use of atomic variables.

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

std::atomic<int> i(0);

int main() {
    auto t1 = std::jthread([](){
        int j;
        do { j = i.load(std::memory_order_relaxed); }
        while (j == 0);
        std::cout << j << std::endl;
    });
    auto t2 = std::jthread([](){
        i.store(1, std::memory_order_relaxed);
    });
    return 0;
}
```

Open in Compiler Explorer

Finally, let us consider two, more complete examples of a stack:

A stack implemented using atomic variables.

```cpp
// based on
↪ https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange
↪
#include <atomic>
#include <stack>

template<typename T>
struct Node {
  T data;
  Node* next;
  Node(const T& data) : data(data), next(nullptr) {}
};

template<typename T> class stack {
    std::atomic<Node<T>*> head;
    public:
    void push(const T& data) {
        Node<T>* new_node = new Node<T>(data);
        new_node->next = head.load(std::memory_order_relaxed);
        while(!head.compare_exchange_weak(new_node->next,
        ↪ new_node, std::memory_order_release,
        ↪ std::memory_order_relaxed));
    }
};

int main() {
    std::stack<int> s; s.push(1); s.push(2); s.push(3);
}
```

Open in Compiler Explorer

We will elaborate upon this later in Chapter 6.

3.2.2 Mutexes and Locks

Standard Template Library in header `<mutex>` provides multiple mutexes (of type `BasicLockable` that implement `lock` and `unlock` methods): `mutex`, `recursive_mutex`, `timed_mutex`, `recursive_timed_mutex`, and `unique_lock`.

A good practice for the use of mutexes is to lock them via the RAII idiom. Since C++11, this is available as `std::unique_lock` and `std::lock_guard`, and since C++17 `scoped_lock` in header `<mutex>`. Crucially, using `scoped_lock` provides the ability to lock multiple mutexes at once, avoiding deadlock. One may hence advise to use one or more `mutex` with a `scoped_lock` on top.

An example of the use of a unique lock.

```cpp
#include <mutex>
#include <thread>
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono_literals;
    struct Shared {
        int value;
        std::mutex mux;
    };
    Shared shared{0,{}};
    auto t1 = std::jthread([&shared]{
        std::this_thread::sleep_for(1s);
        for (int i = 0; i < 10; i++) {
            std::this_thread::yield();
            {
                std::unique_lock lock(shared.mux);
                shared.value += 10;
            } // mutex unlocks!
            std::this_thread::sleep_for(1s);
        }
    });
}
```

Open in Compiler Explorer

A slightly more verbose example considers:

An example of the use of a unique lock.

```cpp
#include <mutex>
#include <thread>
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono_literals;
    struct Shared {
        int value;
        std::mutex mux;
    };
    Shared shared{0,{}};
    auto t = std::jthread([&shared]{
        std::this_thread::sleep_for(1s);
        for (int i = 0; i < 10; i++) {
            {
               std::unique_lock lock(shared.mux);
               shared.value += 1;
            }
            std::this_thread::sleep_for(1s);
        }
    });
    auto observer = std::jthread([&shared]{
        while (true) {
            {
            std::unique_lock lock(shared.mux);
            std::cout << shared.value << std::endl;
            if (shared.value == 10)
                break;
            }
            std::this_thread::sleep_for(1s);
        }
    });
}
```

Open in Compiler Explorer

3.2.3 Barrier

Since C++20, there is support for barriers in header `<barrier>`. The constructor takes an integer value, which is the number of threads that the barrier is expected to block. The key methods are `arrive_and_wait()` and `arrive_and_drop()`. The former functions as one would expect. The latter decrements the initial expected count for all uses by one, as if one thread could never reach the barrier subsequently. This can be very useful in error management:

An example of the use of a barrier.

```cpp
#include <barrier>
#include <syncstream>
#include <iostream>
#include <vector>
#include <thread>
#include <algorithm>
#include <random>

int main() {
    std::barrier b(5);
    std::vector<std::jthread> ts;
    std::generate_n(std::back_inserter(ts), 5, [&b]{
        return std::jthread([&b]{
            std::mt19937
            ↪ gen(std::hash<std::thread::id>{}(std::this_thread::get_id()));
            std::bernoulli_distribution d(0.3);
            int cnt = 1;
            while (true) {
                std::osyncstream(std::cout) <<
                ↪ std::this_thread::get_id() << "/" << cnt
                ↪ << std::endl;
                std::this_thread::yield();
                if (d(gen)) {
                    b.arrive_and_drop();
                    return;
                } else {
                    b.arrive_and_wait();
                }
                cnt++;
            }
        });
    });
}
```

Open in Compiler Explorer

More complicated uses of barriers may use the template parameter `CompletionFunction` and have a callable executed whenever the barrier is hit (reaches zero):

An example of the use of a barrier.

```cpp
#include <barrier>
#include <syncstream>
#include <iostream>
#include <vector>
#include <thread>
#include <algorithm>
#include <random>

int main() {

    std::barrier b(4,[id = 1]() mutable noexcept {
        std::osyncstream(std::cout) << id << " OK" <<
        ↪  std::endl;
        id++;
    });
    std::vector<std::jthread> runners;

    std::generate_n(std::back_inserter(runners), 4, [&b]{
        return std::jthread([&b]{
            std::osyncstream(std::cout) <<
            ↪  std::this_thread::get_id() << "/1" <<
            ↪  std::endl;
            std::this_thread::yield();
            b.arrive_and_wait();
            std::osyncstream(std::cout) <<
            ↪  std::this_thread::get_id() << "/2" <<
            ↪  std::endl;
            std::this_thread::yield();
            b.arrive_and_wait();
        });
    });
    runners.clear();
    std::osyncstream(std::cout) << std::endl;
}
```

Open in Compiler Explorer

3.3 Algorithms in the Standard Template Library

Since C++17, there is an excellent Parallel Standard Template Library in header `<algorithm>`.

3.3.1 For Each

The most useful algorithm from the Standard Template Library (STL) in terms of parallel programming is surely for each:

An example of the use of for each.

```
struct Custom {
    void expensive_operation() {
        // ...
    }
};

std::vector<Custom> data(10);

std::for_each(std::execution::par_unseq,
    data.begin(), data.end(),
    [](Custom& el) {
        el.expensive_operation();
    });
```

Open in Compiler Explorer

As in the serial version of STL, the callable within for each is permitted to change the state of elements, if the underlying range is mutable, but cannot invalidate iterators.

3.3.2 Reduce

Similarly useful is the reduce operation (also known as fold, accumulate, aggregate, compress, or inject). Indeed, a whole approach to parallel programming (Map Reduce) is named after the algorithm. There, one applies an associative operation to each piece of data to obtain a partial result, and then (perhaps using a binary-tree reduction, which also may enforce an order which may or may not be desirable) obtains the final result by applying the same associative operation to the partial results. The binary-tree reduction makes it possible to utilize $O(\log(n))$ rounds of computation on n processors.

An example of the use of reduce.

```
1  std::vector<int> data{1, 2, 3, 4, 5};
2
3  auto sum = std::reduce(data.begin(), data.end(), 0);
4  // sum == 15
5
6  sum = std::reduce(std::execution::par_unseq,
7      data.begin(), data.end(), 0);
8  // sum == 15
9
10 auto product = std::reduce(data.begin(), data.end(), 1,
11     std::multiplies<>{});
12 // product == 120
13
14 product = std::reduce(std::execution::par_unseq,
15     data.begin(), data.end(), 1, std::multiplies<>{});
16 // product == 120
```

Open in Compiler Explorer

3.3.3 Merge

Finally, in implementing parallel sorting algorithms, we will utilize the parallel merge operation:

An example of the use of a merge.

```
1  std::vector<int> data1{1, 2, 3, 4, 5, 6};
2  std::vector<int> data2{3, 4, 5, 6, 7, 8};
3
4  std::vector<int> out(data1.size()+data2.size(), 0);
5  std::merge(std::execution::par_unseq,
6      data1.begin(), data1.end(),
7      data2.begin(), data2.end(),
8      out.begin());
9  // out == {1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 8}
```

Open in Compiler Explorer

Chapter 4

The Syntax in OpenMP

OpenMP is a specification for concurrency (parallel programming of shared memory systems) in Fortran, C, and C++. The current version of the specification can be downloaded from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

The specification is built on top of the fork-join model of parallel execution (sériově-paralelní model uspořádání vláken), but generally does not provide any guarantees as to how a particular directive or function is implemented. This also means that running on a different hardware may result in different order of execution of floating-point operations or similar. Prime implementations include

- libgomp (GOMP) for GCC,
- libomp for clang, and
- liomp5 (IOMP) for ICC/Clang.

There are no guarantees that the output or performance will stay the same across multiple implementations.

OpenMP combines preprocessor directives (pragmas) and a library of functions exported via `omp.h`. Ideally one and the same program written with OpenMP should be possible to run as serial code, or with any number of threads. Many OpenMP programs use only the preprocessor directives, which makes it possible to compile them even as serial code even without OpenMP-aware compiler.

There is a long history of the evolution of OpenMP since 1997, from a 50-page long specification of OpenMP 1.0, through 150-page version 3.0 (introducing "tasks") and 250-page version 4.0 (adding GPGPU support), to much longer versions recently. Traditional implementations of OpenMP have been rather closely built on top of Pthreads,

which results in the lack of fine-grained scheduling, memory management, network management, signaling, etc. The lack of fine-grained scheduling notably means the lack of user-level threads (co-routines) and the lack of queries as to the number of hardware threads utilized by other processes, which often results in high overhead when the number of threads (across all processes!) increases above the number of hardware threads supported. Since OpenMP 5.0, the distinction between threads and tasks has been erased and thread teams are also cast into tasks. There are now also OpenMP implementations over lightweight threads, notably (BOLT is OpenMP over Lightweight Threads, https://www.bolt-omp.org/).

4.1 Threads, Tasks, Coroutines

4.1.1 OpenMP Task Region

Initially, OpenMP application starts with a single thread (initial/master thread). This can spawn parallel regions, typically with multiple threads in a thread pool ("thread team") in the fork-join manner. This means that in any thread, one can either wait for all the "sibling" threads to finish ("join them") or spawn a further, nested parallel region.

A key construct in OpenMP is thus `#pragma omp parallel`, which delineates a parallel region and opens a "team of OpenMP threads", which could be seen as a thread pool of threads or user-level threads. By default, the number of threads is set based on the available hardware threads, but this can be affected by the environment variables (`OMP_NUM_THREADS`) and modifiers of the pragma (`num_threads(2)`) and function calls (`omp_set_num_threads()`).

An example of the use of OpenMP parallel regions.

```cpp
#include <iostream>
#include <syncstream>
#include <omp.h>

int main() {

#pragma omp parallel
    {
        int iam = omp_get_thread_num();
        int nt =  omp_get_num_threads();
        std::osyncstream(std::cout) << iam << "/" << nt <<
        ↪  std::endl;
    }

    return 0;
}
```

Open in Compiler Explorer

The construct can take a number of modifiers, including:

- `num_threads`: number of threads to use in the team
- `private`(list of variables): those variables will be private to each thread
- `firstprivate`(list of variables): those variables will be private to each thread, but initially will copy a value from the master thread using the default copy constructor.
- `shared`(list of variables): these variables will be shared between the master thread and all threads in the new team. It is the programmers responsibility to keep the variables constructed as long as the parallel region is running
- `default`: values of `private`, `firstprivate`, `shared`, `none` suggest what should be the default behaviour for variables not listed above. The default is `shared`, which is suboptimal from both performance and thread-safety perspective. It is wise to issue `default(none)`.
- `reduction`(reduction-identifier : list) suggests that variables in list should be treated as shared, when they are used by the function `reduction-identifier`, which could also take the special values of $+, -, *, \&, |, \hat{}, ||, \max, \min$. The list can include array elements and, when `reduction-identifier` is a static function of a class, accessible data objects of the object.
- `proc_bind`: values of `master` and `close` and `spread` suggest how far from the master thread should be executed the new threads

(same core, close in non-uniform architectures, as far as possible in non-uniform architectures).

Whether the nested parallel regions create their own thread teams or use the existing thread teams depends on settings that we can affect through `omp_set_nested`, or environment variables `OMP_NESTED` (which can be true or false) and `OMP_MAX_ACTIVE_LEVELS`, which controls the maximum number of nested active parallel regions. See, for example:

An example of the use of OpenMP parallel regions.

```
1  #include <iostream>
2  #include <syncstream>
3  #include <omp.h>
4
5  int main() {
6
7     omp_set_nested(1);
8     int iam, nt;
9
10 #pragma omp parallel num_threads(2) private(iam,nt)
11 {
12         iam = omp_get_thread_num();
13         nt =  omp_get_num_threads();
14     std::osyncstream(std::cout) << "L1: " << iam << "/" << nt
       ↪  << std::endl;
15
16 #pragma omp parallel num_threads(2) private(iam,nt)
17 {
18         iam = omp_get_thread_num();
19         nt =  omp_get_num_threads();
20         std::osyncstream(std::cout) << "L2 " << iam << "/" <<
           ↪  nt << std::endl;
21 }
22 }
23     return 0;
24 }
```

Open in Compiler Explorer

Since version 5.0, one can also utilize the `teams` construct on its own, optionally with a target (such a GPGPU).

4.1.2 Threads

As has been mentioned above, ideally one and the same program written with OpenMP should be possible to run as serial code, or with any number of threads. This requires sizing the work in each thread

depending on the number of threads, such as in:

An example of the use of OpenMP parallel regions.

```c
// Based on parallel.1.c in standard OpenMP Examples 5.1
//
↪ https://www.openmp.org/wp-content/uploads/openmp-examples-5.1.pdf

#include <omp.h>

void subdomain(float *x, int istart, int ipoints) {
    int i;
    for (i = 0; i < ipoints; i++) x[istart+i] = 123;
}

void sub(float *x, int npoints) {
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared)
↪ private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
        ipoints = npoints / nt;    /* size of partition */
        istart = iam * ipoints;  /* starting array index */
        if (iam == nt-1)     /* last thread may do more */
            ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}

int main() {
    float array[10000];
    sub(array, 10000);
    return 0;
}
```

Open in Compiler Explorer

4.1.3 Sections

An alternative, non-iterative structuring of the code is possible with sections. Each section is a block of code executed by one thread of the current thread team (corresponding to the innermost enclosing parallel region). One can use `private`, `firstprivate`, `lastprivate`, and `reduction` modifiers, similar to the parallel construct. There is an implied barrier of the sections region, unless eliminated by a `nowait` clause. See, for example:

An example of the use of OpenMP sections.

```cpp
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 2;

void method(const int& i) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    std::cout << "Hello from method " << i << " by thread " <<
    ↪  my_rank << std::endl;
}

int main(int argc, char* argv[]) {

#pragma omp parallel num_threads(thread_count)
    {
#pragma omp sections
        {
#pragma omp section
            { method(1); }
#pragma omp section
            { method(2); }
        }
    }

    return 0;
}
```

Open in Compiler Explorer

An example of the use of OpenMP sections.

```cpp
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 2;

void method(const int& i) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    std::cout << "Hello from method " << i << " by thread " <<
    ↪  my_rank << std::endl;
}

int main(int argc, char* argv[]) {

#pragma omp parallel  num_threads(thread_count)
    {
        method(1);
#pragma omp sections
        {
#pragma omp section
            {
                method(2);
                method(3);
            }
#pragma omp section
            { method(4); }
        }
    }

    return 0;
}
```

Open in Compiler Explorer

4.1.4 Coroutines / Tasks

The closest to a coroutine in OpenMP is the concept of a task. While it does not come with a promise of an implementation with a user-level thread library (cf. Argobots, Converse threads, Qthreads, MassiveThreads, Nanos++, Maestro, GnuPth, StackThreads/MP, Protothreads, Capriccio, StateThreads, TiNy-threads, etc), it often is implemented thus.

An example of the use of OpenMP tasks.

```
1  const int thread_count = 4;
2
3  void Hello() {
4      int my_rank = omp_get_thread_num();
5      int thread_count = omp_get_num_threads();
6      std::cout << "Hello from thread " << my_rank << " of " <<
       ↪  thread_count << std::endl;
7  }
8
9  int main(int argc, char* argv[]) {
10 #pragma omp parallel num_threads(thread_count)
11     {
12 #pragma omp single
13         {
14             Hello();
15 #pragma omp task
16             Hello();
17         }
18     }
19 }
```

Open in Compiler Explorer

4.1.5 Kernels

There is a nascent support for the use of GPGPUs via `target` construct. While the block following the target construct can be arbitrary, in principle, most GPUs are not able to support arbitrary code. Specifically, there are usually no synchronization primitives available and no coherence between L1 caches. Often, block requiring anything beyond the basic arithmetics will execute only on one core ("stream multiprocessor") of the GPGPU. Ideally, one should combine the offloading of the code to the GPGU (via the target), across multiple cores (via teams construct), and then across the code. This can get quite non-trivial:

An example of the use of OpenMP targets and teams.

```
1  const int thread_count = 4;
2
3  void Hello() {
4      int my_rank = omp_get_thread_num();
5      int thread_count = omp_get_num_threads();
6      std::cout << "Hello from thread " << my_rank << " of " <<
       ↪  thread_count << std::endl;
7  }
8
9  int main(int argc, char* argv[]) {
10 #pragma omp parallel num_threads(thread_count)
11     {
12 #pragma omp single
13         {
14             Hello();
15 #pragma omp task
16             Hello();
17         }
18     }
19 }
```

Open in Compiler Explorer

We refer to: https://www.archer.ac.uk/training/course-material/2019/06/AdvOpenMP-manch/L10-OpenMPTargetOffload.pdf from which we took this example for an excellent introduction.

4.2 Synchronisation Primitives

4.2.1 Barrier

OpenMP provides a straightforward, explicit barrier construct:

An example of the use of a barrier.

```
1  #include <iostream>
2  #include <syncstream>
3  #include <thread>
4  #include "omp.h"
5
6  void work() {
7      std::osyncstream(std::cout) << n;
8  #pragma omp barrier
9      std::osyncstream(std::cout) << n;
10 }
11
12 int main() {
13 #pragma omp parallel num_threads(5)
14     work();
15 }
```

Open in Compiler Explorer

Especially with nested parallel regions, the behaviour can be quite non-trivial. All threads of the current team must complete execution of all tasks bound to the same parallel region prior to continuing past the barrier. What is the current team, however, e.g., whether it is created by the innermost enclosing parallel region, depends on the settings of the nesting:

An example of the use of a barrier, whose study is left as an exercise.

```
1  #include <iostream>
2  #include "omp.h"
3
4  void work(int n) { std::cout << n; }
5  void sub3(int n)
6  {
7      work(n);
8  #pragma omp barrier
9      work(n);
10 }
11 void sub2(int k)
12 {
13 #pragma omp parallel shared(k)
14     sub3(k);
15 }
16 void sub1(int n)
17 {
18     int i;
19 #pragma omp parallel private(i) shared(n)
20     {
21 #pragma omp for
22         for (i=0; i<n; i++)
23             sub2(i);
24     }
25 }
26 int main()
27 {
28     sub1(2);
29     sub2(2);
30     sub3(2);
31     return 0;
32 }
```

Open in Compiler Explorer

Barrier is also implied by the entry and exit in parallel regions.

4.2.2 Fences and Flushes

Close to a memory fence, the `flush` construct provides point at which a thread is guaranteed to have a consistent view of memory, similar to a fence.

The `flush` is also implied by the entry and exit in parallel regions, critical regions, operations with locks etc.

4.2.3 Atomic Variables

OpenMP has a rich support for atomic variables. At the simplest one may consider:

An example of the use of atomic variables.

```
#include <iostream>
#include "omp.h"

int main() {

int i = 0;

#pragma omp parallel
{

#pragma omp section
    {
        int j;
#pragma omp atomic
        do { j = i; } while (j == 0);
        std::cout << j << std::endl;
    }

#pragma omp section
    {
#pragma omp atomic
        i = 1;
    }

}
    return 0;
}
```

Open in Compiler Explorer

More broadly, one can specify:

- operations for which the atomicity is enforced, out of `read`, `write`, `update`, `capture`, out of which update is the default. Capture makes it possible to use operators such as `+=`, e.g., `{v = x; x binop= expr;}`.
- memory order, out of `seq_cst`, `acq_rel`, `release`, `acquire`, `relaxed` as discussed in Chapter 2. The default is relaxed-consistency shared memory model.

E.g. `#pragma omp atomic write` or `#pragma omp atomic seq_cst`. Note that you need to use `{}` to create a new block, wherein the operations would be atomic, if you wish to consider some sequence of read and write operations.

4.2.4 Reductions

As we have briefly mentioned on page 41, one may consider a reduction, which produces a single value from an associative operations such as addition, multiplication, taking of the minimum, maximum, or custom associative functions. The goal is for each thread to run the reduction with a private copy and then to produce the final result with the same reduction, perhaps in a hierarchical fashion. A simplistic blueprint is provided in the following example:

An example of the use of reductions.

```
 1 #include "omp.h"
 2
 3 int work() {
 4     return 0;
 5 }
 6
 7 int main() {
 8     int sum = 0;
 9 #pragma omp parallel for reduction(+:sum)
10     for (int i = 0; i < 42; i++) sum += work();
11 }
```

Open in Compiler Explorer

4.2.5 Mutexes

OpenMP has only a limited support for mutexes, as in does not support any RAII variant, which makes them hard to use correctly. One can, however, construct one own's RAII variant:

An example of the use of OpenMP mutexes.

```
#include <iostream>
#include <omp.h>

int count;
omp_nest_lock_t countMutex;

struct CountMutexInit {
    CountMutexInit() { omp_init_nest_lock   (&countMutex);   }
    ~CountMutexInit() { omp_destroy_nest_lock(&countMutex); }
} countMutexInit;

struct CountMutexHold {
    CountMutexHold() { omp_set_nest_lock    (&countMutex); }
    ~CountMutexHold() { omp_unset_nest_lock  (&countMutex); }
};

void Tick() {
    CountMutexHold releaseAtEndOfScope;
    count++;
}

int main() {
    Tick(); return 0;
}
```

Open in Compiler Explorer

4.2.6 Critical Sections

Instead of mutexes, one can safely use critical sections in OpenMP. A critical section is a block of code which can be executed by at most one thread at one time. This can be used to protect non-trivial non-associative update operations, for which we cannot use reductions:

An example of the use of a critical section.

```cpp
const int thread_count = 2;
int main(int argc, char* argv[]) {
    int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) private(b)
    {
        b = omp_get_thread_num()+2;
#pragma omp critical
        {
            a = a / b;
            std::cout << "Variable 'b' = " << b << " by thread " << (omp_get_thread_num()) << std::endl;
            std::cout << "Variable 'a' = " << a << " by thread " << (omp_get_thread_num()) << std::endl;
        }
    }
    std::cout << "Variable 'a' = " << a << " after omp " << std::endl;
    std::cout << "Variable 'b' = " << b << " after omp " << std::endl;
    return 0;
}
```

Open in Compiler Explorer

The use of critical sections does come with a substantial penalty in terms of performance, though.

4.3 Algorithms

While OpenMP does not really implement any algorithms, some of the data-parallel constructs are similar to algorithms in the STL library. Notably:

4.3.1 For Each

OpenMP makes it possible to use "for each":

An example of the use of for each.

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

Open in Compiler Explorer

An example of the use of for each.

```
1  #include <iostream>
2  #include <vector>
3  #include "omp.h"
4
5  int main(int argc, char* argv[]) {
6  #pragma omp parallel
7      {
8          std::cout << "this is a line printed by thread" <<
           ↪  omp_get_thread_num() << std::endl;
9
10 #pragma omp for
11         for (int i=0; i<10; i++)
12             std::cout << "for-loop line " << i << " printed by
               ↪  thread" << omp_get_thread_num() << std::endl;
13
14     }
15
16     std::cout << "Hello from the main thread\n";
17
18     return 0;
19 }
```

Open in Compiler Explorer

Chapter 5

The Syntax in SYCL

SYCL is an open specification for the design of code targeting either CPUs or GPGPUs. The current version is SYCL 2020, released in 2021. If you want to learn more about SYCL, see https://github.com/codeplaysoftware/syclacademy for an extensive tutorial.

5.1 More concepts

To make as efficient use of GPGPUs as possible, SYCL utilizes a number of concepts that we have not seen earlier.

5.1.1 Device selector

Selectors are used to pick device to run on. In header `<device_selector.h>`, there is an abstract class `device_selector` with numerous implementations such as `gpu_selector`, `host_selector`, `opencl_selector`, and `default_selector`.

One can also implement its own subclasses that specify to the runtime how to perform device selection. For example, it may query the amount of memory on the GPGPU and if it is sufficient, use GPGPU. If it were not sufficient, it could use CPU as a fallback. There, one overrides `int operator()(const sycl::device& dev) const override` and returns an integer for the priority. The higher integer, the higher priority.

5.1.2 Queues

A queue, `queue`, is an abstraction of a device, through which we orchestrate work on the device. In a constructor of a queue, we pass

the device, which cannot be changed later, but one can create further queues for the same device. A key method is `queue.submit`, which passes a "command group" for asynchronous execution.

A command group is, essentially, composed of a set of requirements and actual commands executing a kernel. The kernel then receives a "command group handler" from the SYCL as its argument, and uses it to access the API. A command group submission to the group is atomic.

Asynchronous execution also means an undefined order of execution, unless we use `wait` or suggest the dependencies between the "actions" in the form of a task graph. We can also declare the queue to be in-order, similar to sorted in OpenMP: `queue q{property::queue::in_order()};`.

5.1.3 Work items, Work groups, and Kernels

Within an action submitted to the queue, we execute kernels. Kernels are callables

- receiving an index to the run of the kernel as `auto idx` or `id<1> idx` or similar.
- returning nothing; with `void` return type
- which cannot allocate memory dynamically which cannot use certain other features (e.g., RTTI).

Within the `single_task function` method of the "command group handler" API, we pass a C++ function object as a parameter and have it executed once. Kernel can also be a class that overloads operator (), e.g. `void operator()(id<1> idx)`.

Most often, we want the kernel executed many times, in a data-parallel fashion. In the so-called nd-ranges, we partition the index-set of a data set hierarchically first into into global ranges, and then into local ranges. The local range corresponds to a work-group and each element corresponds to a work item (= single run of a kernel). Typically, all work items within a work group are executed in lock step (i.e., the same hardware instruction in all work items at the same time). The work-group local memory can often be accessed very efficiently, via `local_accessor`, and can be used to coordinate multiple work items (= single runs) within a work group.

To summarize, each work item can access:

- private memory
- work-group local memory

– global memory accessible to all work items within an nd-range, but whose access can be very expensive, as it involves copying data across PCIe bus
– constant memory, which is a part of the global memory, but which can be very cheap to access.

5.1.4 Asynchronous errors

The SYCL implementation may throw "synchronous errors" (one at a time). In contrast, asynchronous errors are produced by a command group or a kernel (with many kernels running at any point). By default, asynchronous errors are not propagated to the host. One can, however, defined and error handler and pass it to a queue `queue q(default_selector{}, exceptio`
The error handler receives an `exception_list`, wherein one can iterate over `std::exception_ptr`.

See https://www.codingame.com/playgrounds/48226/introduction-to-sycl/error-handling for a great tutorial with code that is editable, compilable, and runnable online. Let us simplify their main example somewhat:

Error handling in SYCL.

```
1  // based on error_handling.cpp of "Introduction to SYCL",
2  //
   ↪  https://www.codingame.com/playgrounds/48226/introduction-to-sycl
3
4  #include <iostream>
5  #include <sycl/sycl.hpp>
6
7  using namespace sycl;
8
9  class exception1;
10
11 int main(int, char**) {
12    auto exception_handler = [] (exception_list exceptions) {
13       for (std::exception_ptr const& e : exceptions) {
14          try {
15             std::rethrow_exception(e);
16          } catch(exception const& e) {
17          std::cout << "Caught asynchronous SYCL exception:\n"
18                    << e.what() << std::endl;
19          }
20       }
21    };
22
23    queue q(default_selector{}, exception_handler);
24    // actual use of the q
25
26    try {
27       q.wait_and_throw();
28    } catch (exception const& e) {
29       std::cout << "Caught synchronous SYCL exception:\n"
30       << e.what() << std::endl;
31    }
32    return 0;
33 }
```

Open in Compiler Explorer

5.1.5 Unified shared memory

At the cost of some latency, one can use a unified shared memory across both the host and the device, wherein oneuses the same pointer on both the host and the device. This requires:

```
template <typename T> T* malloc_shared(size_t count,                cons
```

and the corresponding

```
void free(void* ptr, sycl::queue& syclQueue)
```

See the example below:

An example of the use of unified shared memory.

```
1 auto shared = malloc_shared<double>(42, q);
2
3 q.submit([&](handler& cgh){
4   cgh.parallel_for(range{42}, [=](id<1> tid){
5     shared[i] = 0.0;
6   }
7 });
```

Open in Compiler Explorer

5.1.6 Buffers and accessors

A buffer is a constrained view of a 1-, 2-, or 3-dimensional array. The constraints specify how it can be accessed on the host, the device or both. A buffer is constructed with a pre-allocated, trivially copyable C++ objects (e.g., STL container). Within the contract for the use of the buffer, one promises not to amend the memory used to initialise the buffer during the lifetime of the buffer. Buffer promises to update the memory in the host upon destruction, in RAII spirit.

In the case of one-dimensional arrays, one can call the constructor with an interator: `template <typename InputIterator> buffer(InputIterator first, InputItera`

Once in a kernel, an `accessor` specifies constraints on the use of a buffer therein. Two key choices are:

- access mode: `read`, `write`, and `read_write`, where write access mode also implicitly defines dependencies between tasks
- access target: `global_memory` suggests that the data resides in the global memory space of the device. Other options are device specific. `no_init` suggests that the initial data can be discarded (not moved to the device).

See the example below:

An example of the use of buffers and accessors.

```
1  buffer<double> A{range{42}};
2
3  q.submit([&](handler& cgh){
4     accessor aA{A, cgh};
5     cgh.parallel_for(range{42}, [=](id<1> & idx){
6       aA[idx] = 0.0;
7     })
8  });
9
10 host_accessor result{A};
11 for (int i = 0; i < N; i++) {
12   assert(result[i] == 0);
13 }
```

Open in Compiler Explorer

5.1.7 Barrier

Depending on the details of the use of a barrier, one may wish to use `sycl::queue::wait()` and `sycl::queue::wait_and_throw()`, or `item::barrier(access::fence_space)` within a kernel.

5.1.8 More complex examples

First, let us consider a complete, working example:

An example of vector addition in SYCL.

```
#include <iostream>
#include <vector>
#include <CL/sycl.hpp>

using namespace std;
using namespace cl::sycl;
class buffer3;

int main(int, char**) {

   std::vector<float> a { 2.0, 3.0, 7.0, 4.0 };
   std::vector<float> b { 4.0, 6.0, 1.0, 3.0 };
   std::vector<float> c { 0.0, 0.0, 0.0, 0.0 };

   default_selector device_selector;

   queue q(device_selector);

   std::cout << "Running on "
             <<
             ↪  queue.get_device().get_info<info::device::name>()
             << "\n";
   {
      buffer bufA(a);
      buffer bufB(b);
      buffer bufC(c);

      q.submit([&] (handler& cgh) {

         auto accA = accessor(bufA, cgh, read_only);
         auto accB = accessor(bufB, cgh, read_only);
         auto accC = accessor(buff, cgh, write_only);
         cgh.single_task<buffer3>(bufC.get_range(), [=](id<1>
         ↪  i) {
           accC[i] = accA[i] + accB[i];
         });
      });
      q.wait_and_throw();
   }

   return 0;
}
```

Open in Compiler Explorer

Next, let us consider an example that uses work groups and local memory in an attempt to utilize more of the performance available in the GPGPU.

An example of the use of work groups and local memory.

```
// based on memory4.cpp of "Introduction to SYCL",
//
//  https://www.codingame.com/playgrounds/48226/introduction-to-sycl

  #include <array>
  #include <cstdint>
  #include <iostream>
  #include <random>
  #include <cassert>

  #include <CL/sycl.hpp>

  class reduction_kernel;
  namespace sycl = cl::sycl;

  int main(int, char**) {
    std::array<int32_t, 16> arr;

    std::mt19937 mt_engine(std::random_device{}());
    std::uniform_int_distribution<int32_t> idist(0, 10);

    std::cout << "Data: ";
    for (auto& el : arr) {
      el = idist(mt_engine);
      std::cout << el << " ";
    }
    std::cout << std::endl;

    sycl::buffer<int32_t, 1> buf(arr.data(),
    ↪  sycl::range<1>(arr.size()));

    sycl::device device =
    ↪  sycl::default_selector{}.select_device();

    sycl::queue queue(device, [] (sycl::exception_list el) {
       for (auto ex : el) { std::rethrow_exception(ex); }
    } );

    size_t wgroup_size = 32;
    auto part_size = wgroup_size * 2;

    auto has_local_mem = device.is_host()
        ||
        ↪  (device.get_info<sycl::info::device::local_mem_type>()
        != sycl::info::local_mem_type::none);
    auto local_mem_size =
    ↪  device.get_info<sycl::info::device::local_mem_size>();
    if (!has_local_mem
      || local_mem_size < (wgroup_size * sizeof(int32_t)))
    {
       throw "Device doesn't have enough local memory!";
    }
```



```
#include "buffer4.h"

    auto acc = buf.get_access<sycl::access::mode::read>();
    std::cout << "Sum: " << acc[0] << std::endl;

    return 0;
  }
```

Open in Compiler Explorer

5.1.9 Building code

Compiling code with SYCL should require passing "-fsycl" to a compiler that supports the SYCL 2020 specification, but it often turns out to be substantially more complicated than with standard C++23 or OpenMP.

There are many implementations of the SYCL specification (incl. OpenSYCL, Intel DPC++, CodePlay ComputeCPP) with minor differences. One way of getting around the complexity is to install the OpenSYCL compiler: https://github.com/OpenSYCL/OpenSYCL (formerly known as hipSYCL).

Independent of the compiler, to build code with SYCL that targets a GPGPU, you need to link with the appropriate libraries, which depends on what is your target. If you target, e.g., common OpenCL variants:

Linking SYCL against OpenCL.

Open in Compiler Explorer

If you target MKL:

Linking SYCL against MKL.

Open in Compiler Explorer

If you target CUDA (or NVIDIA OpenCL), you may need to set up paths to CUDA, in addition to the use of:

```
-fsycl -fsycl-targets=nvptx64-nvidia-cuda
```

Similarly for AMD:

```
-fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend {offload-arch=gfx906
```

For details of the use of SYCL with CUDA, see https://github.com/codeplaysoftware/SYCL-For-CUDA-Examples

Chapter 6

Introducing parallel data structures

In Section 3.2.1, we have seen an example of a simple parallel stack developed with Atomic variables. Let us now consider a more elaborate version:

A more complete example of a stack.

```cpp
// inspired by https://www.modernescpp.com/index.php/atomic-smart-pointers

#include <iostream>
#include <atomic>
#include <thread>
#include <memory>

template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
    concurrent_stack( concurrent_stack &) = delete;
    void operator=(concurrent_stack&) = delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
       reference(shared_ptr<Node> p_) : p{p_} { }
       T& operator* () { return p->t; }
       T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load();
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
       return reference(head);
    }
    void push_front( T t ) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;
      while( !head.compare_exchange_weak(p->next, p) ){ }
    }
    void pop_front() {
       auto p = head.load();
       while( p && !head.compare_exchange_weak(p, p->next) ){ }
    }
};
```

Open in Compiler Explorer