Parallel Programming in C++23

Jakub Marecek and team

version March 6, 2024
https://github.com/jmarecek/book-parallel-cpp

March 6, 2024

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

This copy of the book is dated March 6, 2024.

Preface

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

About the authors

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.

A wider team has contributed:

- Simon Toth has been a major source of inspiration for the project with his book The Standard Algorithms in C++. The build system for the lecture notes is taken from the book almost in verbatim, in compliance with the Creative Commons licence. The lecture notes also feature several examples from his book.

- Branislav Bosansky has been a major source of inspiration for the project with his course on OpenMP at the Czech Technical University, which I took over in 2020. Brano is the author of the matrix-multiplication examples in Chapter 6.

- Illia Kryvoviaz has fixed numerous issues with the examples and added the GodBolt links.

- Jan Mrkos has provided a number of suggestions that improved the SYCL chapter in particular.

Feedback

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

Why cc-by-sa-nc

This book is licensed CC-BY-SA-NC, which allows derivative works, but does not permit commercial use including selling printed copies.

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.

This is related to the changing "shape" of computers: Computers no longer get faster, just "wider" and "heterogeneous". Data-parallel computing is most scalable approach to the "wider" computers.

In these lecture notes, you will learn about parallel computing and data-parallel computing, in particular. 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 Computational Complexity

A large part of theoretical computer science is devoted to the study of the limitations of parallelizability of algorithms, starting with the very clear question: does every problem with a polynomial-time sequential algorithm also have an efficient parallel algorithm? In short, the answer is no.

Let us consider the P-Complete problems [1], i.e., decision problems that are in P and where every problem in P can be reduced to the P-Complete problem in logarithmic space (L). The prime examples of P-Complete problems are

- circuit evaluation: given a Boolean circuit and an input, decide what the circuit outputs

- linear programming: given a linear function subject to linear inequality constraints, find the optimum of the linear function
- many graph problems, such as Lexicographically First Depth-first Search Ordering (LFDFS): given an undirected graph with fixed ordered adjacency lists, and two designated vertices u and v, is vertex u visited before vertex v in the depth-first search of the graph?
- many compression algorithms: given strings s and t, will compressing s with LZ78 add t to the dictionary?
- many problems related to Markov decision processes: is the minimum expected cost over all policies equal to 0 (for both finite, and long-run versions).
- many tests of local optimality in combinatorial optimization: in an instance of Traveling Salesman, and a sequence of tours, is this a 2-Opt sequence?

Therein, we cannot hope for efficient solution on a parallel computer in the sense of solving in time $O(log^c n)$ using $O(n^k)$ parallel processors for some constants c and k.

In contrast, Nick's class NC^c are decision problems solvable by a uniform family of Boolean circuits with polynomial size, depth $O(log^c(n))$, and fan-in 2. More usefully, a problem in NC with input of length n can be solved in time $O(log^c n)$ using $O(n^k)$ parallel processors for some constants c and k. (Notice that for a constant k, $O(n^k)$ is a polynomial.) Thus, NC can be thought of as problems that can be efficiently solved on a parallel computer. Integer arithmetics (addition, multiplication and division), matrix arithmetics (multiplication, determinant, inverse, rank), or multiple graph problems (shortest path, maximal matching with some restrictions on the weights) are in Nick's class and hence "easier" than P-Complete problems.

2.3 Hardware support

2.3.1 Memory hierarchies

Most modern computers utilize complicated cache hierarchies. One may have 32 KB of L1 cache and 256-512 KB of L2 cache on each core, with cca. 1 ns and 3-4 ns latencies, respectively. (This will get reported as 32 MB of L2 cache on a Threadripper 3990X.) One may have many megabytes of L3 cache shared between the cores in some non-uniform fashion, with non-uniform latency of 10-20 ns. Finally, one may access the memory in cca. 60 ns. Compare these times with the 0.27 ns cycle time of a processor clocked at 3.7 GHz.

Crucially, the times above are the best-case performance. To understand the worst-case performance, one needs to understand cache snooping (with a write-invalidate protocol). Most modern cores monitor a stream of memory loads and stores across the whole shared-memory machine. If a transaction concerns store in a block cached on the core (processor), the core sets some "invalid" flag to the block in its cache. If the core wishes to read a block with the "invalid" flag, it will ask all other cores for a valid value. Obtaining the correct value takes time.

As we will see in the distributed-memory part of the course, one can also maintain the coherence otherwise, which is known as a directory-based coherency mechanism. Either way, cache coherence defines the requirements on concurrent reads and writes to the same address. To some extent, it masks the presence of cache hierarchies in any modern computing system.

2.3.2 GPGPUs

As we have mentioned in the introduction, the bulk of the computational performance of most modern computing system (from a mobile phone to a supercomputer) is in its GPGPUs. Let us briefly consider a particular example of the NVIDIA Ampere architecture of GeForce RTX 3080 or NVIDIA A100:

- There are seven Graphics Processing Clusters (GPCs), sharing up to 40 MB of L2 cache and up to 40 GB of high-speed HBM2 memory
- Within each GPC, there are 12 Streaming Multiprocessors (SMs),
- Within each SM, there are 128 cores working with single-precision floating-point (FP32) precision and two double-precision (FP64) units. There is also 128 KB of L1/Shared Memory, shared across the 128 cores.
- Each SM is partitioned into four processing blocks (or partitions), each with a few kilobytes of L0 instruction cache and one warp scheduler.

Altogether, A100 has $7 \cdot 12 \cdot 128 = 10752$ cores, but their use is rather constrained. Each warp can have at most 32 threads and runs them in lock-step on one processing blocks. Each streaming multiprocessors can run at most 64 warps of 32 threadblocks (i.e., 2048 threads per SM). Further constraints are due to the register use: each thread can use at most 255 registers, but there are only 65536 32-bit registers for the SM (yielding a limit of 257 threads per SM, at full register utilization). Further constraints are due to the use of memory hierarchy (esp. the 128 KB of shared memory, shared across the 128 cores).

Similar to the CPU, the GPU hence has a memory hierarchy:

- L1 cache with 33 cycle latency and shared memory with even lower latency, based on microbenchmarking [2]
- L2 cache with up to 2080 GB/s read bandwidth (200 cycle latency), based on microbenchmarking [2]
- on-board HBM2 memory with 1555 GB/sec bandwidth (290 cycle latency)
- intra-board NVLink with 50 Gb/sec per signal pair bandwidth
- access to RAM via PCI Express Gen 4 (PCIe Gen 4) at 31.5 GB/sec
- optionally, intra-node communication at 200 Gbit/sec using InfiniBand.

The interaction of the GPGPU memory hierarchy and CPU memory hierarchy is non-trivial, but summarized by the suggestion to reduce the number and volume of transfers between the host and the GPGPU, even at the price of increasing the volume of computation substantially. (Compare the numbers above to M.2 PCIe Gen4 SSDs with 7 GB/sec bandwidth.)

2.3.3 Memory order

Thus, there are several options for implementing synchronization primitives, known as "memory orders" or memory consistency. They define allowed behaviours of loads and stores to different addresses. 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, NVIDIA) 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, unless they use synchronization primitives (e.g. a fence).

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.

With `memory_order_seq_cst` (sequential consistency), 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 and the strongest form of consistency, but it may also seem weak, in that one

considers the ordering of the instructions the processor chose, rather than the ordering the programmer wrote down.

(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.)

See [3] for an informal overview, [4] for a formal overview, and https://www.youtube.com/watch?v=A_vAG6LIHwQ&ab_channel=ACCUConference for a nice lecture.

2.3.4 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".

```
// based on
↪ https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/atomic

template <typename _Tp>
struct atomic {
  using value_type = _Tp;

  [...]

      bool
      compare_exchange_strong(_Tp &__e, _Tp __i, memory_order __s,
                              memory_order __f) noexcept {
        return __atomic_impl::__compare_exchange(_M_i, __e, __i,
        ↪ false, __s, __f);
      }

  bool compare_exchange_weak(_Tp &__e, _Tp __i, memory_order __s,
                             memory_order __f) noexcept {
    return __atomic_impl::__compare_exchange(_M_i, __e, __i, true,
    ↪ __s, __f);
  }

  [...]
};

// Partial specialization for pointer types.
template <typename _Tp>
struct atomic<_Tp *> {
  using value_type = _Tp *;
  using difference_type = ptrdiff_t;

  [...]

      bool
      compare_exchange_strong(__pointer_type &__p1, __pointer_type
      ↪ __p2, memory_order __m1, memory_order __m2) noexcept {
        return _M_b.compare_exchange_strong(__p1, __p2, __m1,
        ↪ __m2);
      }

          [...]
};
```

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()`.

2.4 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.

2.4.1 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.

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

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

2.4.2 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` (joining threads ensures synchronization and waits for completion, while detaching threads allows them to run independently without synchronization or waiting) 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 <chrono>
2  #include <iostream>
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.

```
#include <chrono>
#include <iostream>
#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" (RAII) 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. When a resource is acquired, it is allocated in the constructor, and when the object is destroyed, the resource is automatically released in the destructor. 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 likely result in abnormal program termination if we changed jthread to std::thread. The reason for this is that with thread, we would still need to manually call ‘join‘ or ‘detach‘ to ensure proper thread management. If we forget to do so before the ‘std::thread‘ object goes out of scope, the program could terminate abnormally.

Additionally, when using standard output, it is recommended to wrap it in a ‘syncstream‘. A ‘syncstream‘ ensures thread-safe access to the standard output, preventing data races and ensuring correct output when multiple threads are involved.

An example of the use of jthread.

```
#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.

```
#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.

```
#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 <atomic>
#include <iostream>
#include <syncstream>
#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 can define coroutines, which provide a way to pause the execution of a function, return control to the calling code, and later resume from where it left off, without blocking the entire program. To get a preview, consider class template std::generator in header generator presents a view of the elements yielded by the evaluation of a coroutine:

An example of the use of coroutines, which currently does not compile in trunk of gcc or clang or MSVC 19.

```
1  #include <coroutine>
2  #include <generator>
3  #include <iostream>
4  #include <syncstream>
5
6  std::generator<int> work() {
7    for (int i = 0; i < 10; i++) {
8      co_yield i;
9    }
10 }
11
12 int main() {
13   for (int i : work()) {
14     std::osyncstream(std::cout) << i << '\n';
15   }
16 }
```

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 <memory>
7  #include <string>
8  #include <syncstream>
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    } catch (...) {
6      promise.unhandled_exception();
7    }
8    // finally
9    co_await promise.final_suspend();
10 }
```

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. It provides the necessary machinery to wait for an asynchronous operation to complete and resume the coroutine when the result is available. 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    constexpr bool await_ready() const noexcept { return false;
     ↪  }
7    constexpr void await_suspend(coroutine_handle<>) const
     ↪  noexcept {}
8    constexpr void await_resume() const noexcept {}
9  };
10
11 struct suspend_never {
12   constexpr bool await_ready() const noexcept { return true; }
13   constexpr void await_suspend(coroutine_handle<>) const
     ↪  noexcept {}
14   constexpr void await_resume() const noexcept {}
15 };
```

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 "coroutines4.h"
5  #include <coroutine>
6  #include <iostream>
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)

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

#include <atomic>
#include <chrono>
#include <coroutine>
#include <functional>
#include <iostream>
#include <stdexcept>
#include <string>
#include <thread>

#include "coroutines5a.h"
#include "coroutines5b.h"

using namespace std::chrono_literals;

int main() {

  std::cout << std::endl;

  std::cout << "Notification before waiting" << std::endl;
  Event event1{};
  auto senderThread1 = std::thread([&event1] {
  ↪ event1.notify(); });
  auto receiverThread1 = std::thread(receiver,
  ↪ std::ref(event1));

  receiverThread1.join();
  senderThread1.join();

  std::cout << std::endl;

  std::cout << "Notification after 2 seconds waiting" <<
  ↪ std::endl;
  Event event2{};
  auto receiverThread2 = std::thread(receiver,
  ↪ std::ref(event2));
  auto senderThread2 = std::thread([&event2] {
    std::this_thread::sleep_for(2s);
    event2.notify();
  });

  receiverThread2.join();
  senderThread2.join();

  std::cout << std::endl;
}
```

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 (hey ensure that operations performed on the variable are executed atomically, meaning they are indivisible and cannot be interrupted by other threads) 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.

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

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

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

Open in Compiler Explorer

An example of the use of atomic variables.

```
#include <atomic>
#include <iostream>
#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.

```
1  // based on
   ↪  https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange
2  #include <atomic>
3  #include <stack>
4
5  template <typename T>
6  struct Node {
7    T data;
8    Node *next;
9    Node(const T &data) : data(data), next(nullptr) {}
10 };
11
12 template <typename T>
13 class stack {
14   std::atomic<Node<T> *> head;
15
16 public:
17   void push(const T &data) {
18     Node<T> *new_node = new Node<T>(data);
19     new_node->next = head.load(std::memory_order_relaxed);
20     while (!head.compare_exchange_weak(new_node->next,
       ↪  new_node, std::memory_order_release,
       ↪  std::memory_order_relaxed))
21       ;
22   }
23 };
24
25 int main() {
26   std::stack<int> s;
27   s.push(1);
28   s.push(2);
29   s.push(3);
30 }
```

Open in Compiler Explorer

We will elaborate upon this later in Chapter **??**.

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

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

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>`. They allow threads to synchronize and coordinate their progress by providing a designated point in the code where all participating threads must reach before any of them can proceed further. When a thread reaches a barrier, it will be blocked until all other threads in the same synchronization group also reach the barrier. Once all threads have arrived, the barrier is released, and all threads are allowed to continue executing. 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 <algorithm>
#include <barrier>
#include <iostream>
#include <random>
#include <syncstream>
#include <thread>
#include <vector>

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

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.

```
1  struct Custom {
2    void expensive_operation() {
3      // ...
4    }
5  };
6
7  std::vector<Custom> data(10);
8
9  std::for_each(std::execution::par_unseq,
10               data.begin(), data.end(),
11               [](Custom &el) {
12                 el.expensive_operation();
13               });
```

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.

```cpp
std::vector<int> data{1, 2, 3, 4, 5};

auto sum = std::reduce(data.begin(), data.end(), 0);
// sum == 15

sum = std::reduce(std::execution::par_unseq,
                  data.begin(), data.end(), 0);
// sum == 15

auto product = std::reduce(data.begin(), data.end(), 1,
                           std::multiplies<>{});
// product == 120

product = std::reduce(std::execution::par_unseq,
                      data.begin(), data.end(), 1,
                      ↪  std::multiplies<>{});
// 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.

```cpp
std::vector<int> data1{1, 2, 3, 4, 5, 6};
std::vector<int> data2{3, 4, 5, 6, 7, 8};

std::vector<int> out(data1.size() + data2.size(), 0);
std::merge(std::execution::par_unseq,
           data1.begin(), data1.end(),
           data2.begin(), data2.end(),
           out.begin());
// 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 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, with many examples at https://github.com/OpenMP/Examples.

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 and vastly different performance. Prime implementations include

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

There are also two "subprime" implementation available in Microsoft Visual Studio, which has its own OpenMP runtime (`/openmp`) and an experimental support (`/openmp:llvm`) for the clang/llvm OpenMP runtime, in both cases at version 2.0. If you wish to use Microsoft Visual Studio, notice that Intel oneAPI release 2023.0 oficially supports Microsoft Visual Studio 2022.

OpenMP specifies 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 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 meant 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 <omp.h>
#include <syncstream>

int main() {

#pragma omp parallel default(none)
  {
    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 their value will be copied from the master thread using the default copy constructor.
- `lastprivate`(list of variables): those variables will be private to each thread, but at the end, their value will be copied to 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 programmer's 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 $+, -, *, \&, |, ^, ||, \max, \min$. The list can include array el-

ements 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 <omp.h>
3  #include <syncstream>
4
5  int main() {
6
7    omp_set_nested(1);
8    int iam, nt;
9
10 #pragma omp parallel num_threads(2) default(none) 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) default(none) 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

4.1.2 Threads and their Sizing

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.

```
1  // Based on parallel.1.c in standard OpenMP Examples 5.1
2  //
   ↪  https://www.openmp.org/wp-content/uploads/openmp-examples-5.1.pdf
3
4  #include <omp.h>
5
6  void subdomain(float *x, int istart, int ipoints) {
7    int i;
8    for (i = 0; i < ipoints; i++)
9      x[istart + i] = 123;
10 }
11
12 void sub(float *x, int npoints) {
13   int iam, nt, ipoints, istart;
14
15 #pragma omp parallel default(shared) private(iam, nt, ipoints,
   ↪  istart)
16   {
17     iam = omp_get_thread_num();
18     nt = omp_get_num_threads();
19     ipoints = npoints / nt; /* size of partition */
20     istart = iam * ipoints; /* starting array index */
21     if (iam == nt - 1)      /* last thread may do more */
22       ipoints = npoints - istart;
23     subdomain(x, istart, ipoints);
24   }
25 }
26
27 int main() {
28   float array[10000];
29   sub(array, 10000);
30   return 0;
31 }
```

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.

```
1  #include "omp.h"
2  #include <iostream>
3  #include <syncstream>
4
5  const int thread_count = 2;
6
7  void work(const int &i) {
8    int iam = omp_get_thread_num();
9    std::osyncstream(std::cout) << "Hello from work(" << i << ")
     ↪  by t = " << iam << std::endl;
10 }
11
12 int main(int argc, char *argv[]) {
13 #pragma omp parallel num_threads(thread_count)
14   {
15 #pragma omp sections
16     {
17 #pragma omp section
18       { work(1); }
19 #pragma omp section
20       { work(2); }
21     }
22   }
23   return 0;
24 }
```

Open in Compiler Explorer

An example of the use of OpenMP sections.

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

const int thread_count = 2;

void work(const int &i) {
  int iam = omp_get_thread_num();
  std::osyncstream(std::cout) << "Hello from work(" << i << ") by t = " << iam << std::endl;
}

int main(int argc, char *argv[]) {
#pragma omp parallel num_threads(thread_count)
  {
#pragma omp sections
    {
#pragma omp section
      {
        work(2);
        work(3);
      }
#pragma omp section
      { work(4); }
    }
  }
  return 0;
}
```

Open in Compiler Explorer

4.1.4 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.

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

const int thread_count = 4;

void work() {
  int iam = omp_get_thread_num();
  int nt = omp_get_num_threads();
  std::osyncstream(std::cout) << "Thread " << iam << " of " <<
  ↪  nt << std::endl;
}

int main(int argc, char *argv[]) {
#pragma omp parallel num_threads(thread_count)
  {
#pragma omp single
    {
      work();
#pragma omp task
      work();
    }
  }
}
```

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 limitations on the use of synchronization primitives and coherence among L1 caches. Ideally, one should combine the offloading of the code to the GPGU (via the target), across multiple partitions (via teams construct) etc. This can get quite non-trivial:

An example of the use of OpenMP targets and teams.

```
// Copyright EPCC, The University of Edinburgh,
↪  www.epcc.ed.ac.uk
// Creative Commons AttributionNonCommercial-ShareAlike 4.0
↪  International License.

#pragma omp target teams distribute parallel for\
map(to                      \
    : B, C),                \
    map(tofrom              \
        : sum) reduction(+ \
                         : sum)
for (int i = 0; i < N; i++) {
  sum += B[i] + C[i];
}
```

Open in Compiler Explorer

Another example of the use of OpenMP targets and teams.

```
// Based on:
//
↪  https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Intr

void saxpy(float a, float *x, float *y, int sz, int
↪  num_blocks) {
#pragma omp target teams distribute parallel for simd \
num_teams(num_blocks) map(to                          \
                          : x [0:sz]) map(tofrom      \
                                          : y [0:sz])
  for (int i = 0; i < sz; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

Open in Compiler Explorer

We could also use `nowait` as in other OpenMP constructs, but that does not match the "lockstep" execution on the GPGPUs. Since version 5.0, one can also utilize the `teams` construct on its own, optionally with a target (such a GPGPU).

The advantage is that from the "top-level perspective", the use of the example above remains standard:

Compiling the example above.

```
1  #include "target2.h"
2  #include "omp.h"
3  #include <vector>
4
5  int main() {
6    float a = 3.1415;
7    int sz = 1048576;
8    int num_blocks = 4096;
9    std::vector<float> vx(sz);
10   float *x = vx.data();
11   std::vector<float> vy(sz);
12   float *y = vy.data();
13   saxpy(a, x, y, sz, num_blocks);
14   return 0;
15 }
```

Open in Compiler Explorer

Compiling OpenMP with offloading is non-trivial. As has been mentioned in the introduction to this chapter, there are multiple implementations of OpenMP. Most likely, you have used `gcc -fopenmp`, so far. Depending on what GPGPUs you wish to target, you may need to switch:

- NVIDIA CUDA Compiler Driver NVCC may be the easiest to use with NVIDIA hardware. For A100 of the RCI cluster, use cc80: `nvcc {mp=gpu -gpu=cc80`
- clang/LLVM compilers use `clang++ -fopenmp -fopenmp-targets=<target triple` where the triple is documented at https://llvm.org/doxygen/Triple_8h_source.html such as `clang++ -fopenmp -fopenmp-targets=nvptx64 -`
- AMD ROC is built on top of Clang, so starting with `clang -fopenmp -offload-arch` is a good idea. In more detail, `-target x86_64-pc-linux-gnu -fopenmp -fopenmp-`
- Intel Compiler Collection is, since 2021, also built on top of Clang, except still uses the Intel OpenMP library, so you start with `icx -fiopenmp -fopenmp-targets=<target triple>`. Intel has support for `spir64` and `spirv64` architectures, as well as the above.
- NVIDIA on Power with IBM Open XL C/C++ for Linux on Power: `xlc {qsmp {qoffload {qtgtarch=sm_70`

We refer to: https://www.archer.ac.uk/training/course-material/2019/06/AdvOpenMP-manch/L10-OpenMPTargetOffload.pdf from which we took the first example and to https://www.openmp.org/wp-content/

uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf, from which we adapted the second example, for further details.

4.2 Synchronisation Primitives

4.2.1 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 "omp.h"
#include <iostream>
#include <syncstream>

int main() {

  int i = 0;

#pragma omp parallel
  {

#pragma omp section
    {
      int j;
#pragma omp atomic
      do {
        j = i;
      } while (j == 0);
      std::osyncstream(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.

4.2.2 Reductions

As we have briefly mentioned on page 48, 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() { return 0; }
4
5  int main() {
6    int sum = 0;
7  #pragma omp parallel for reduction(+ \
8                                     : sum)
9    for (int i = 0; i < 42; i++)
10     sum += work();
11 }
```

Open in Compiler Explorer

4.2.3 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 <omp.h>

int count;
omp_nest_lock_t countMutex;

struct RAIIMutexInit {
  RAIIMutexInit() { omp_init_nest_lock(&countMutex); }
  ~RAIIMutexInit() { omp_destroy_nest_lock(&countMutex); }
} countMutexInit; // notice the scope!

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

void work() {
  RAIIMutexHold releaseAtEndOfScope;
  count++;
}

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

Open in Compiler Explorer

4.2.4 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.

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

const int thread_count = 2;

int main() {
  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;
      int t = omp_get_thread_num();
      std::osyncstream(std::cout) << "b = " << b << " in t = " << t << "\n";
      std::osyncstream(std::cout) << "b = " << a << " in t = " << t << "\n";
    }
  }
  std::osyncstream(std::cout) << "a = " << a << "\n";
  std::osyncstream(std::cout) << "b = " << b << "\n";
  return 0;
}
```

Open in Compiler Explorer

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

4.2.5 Barrier

OpenMP provides a straightforward, explicit barrier construct:

An example of the use of a barrier.

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

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.

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

void work(int n) { std::osyncstream(std::cout) << n; }
void sub3(int n) {
  work(n);
#pragma omp barrier
  work(n);
}
void sub2(int k) {
#pragma omp parallel shared(k)
  sub3(k);
}
void sub1(int n) {
  int i;
#pragma omp parallel private(i) shared(n)
  {
#pragma omp for
    for (i = 0; i < n; i++)
      sub2(i);
  }
}
int main() {
  sub1(2);
  sub2(2);
  sub3(2);
  return 0;
}
```

Open in Compiler Explorer

Barrier is also implied by the entry and exit in `parallel` regions. There is also an implicit barrier at the end of a `for`, `sections`, `scope`, and `workshare` constructs, unless one explicitly adds a `nowait` clause.

4.2.6 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.

4.3 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.

```
// Please compile with -std=c++2b -fopenmp

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

int main(int argc, char *argv[]) {
  std::cout << "Hello from the main thread\n";

#pragma omp parallel for
  for (int i = 0; i < 10; i++)
    std::osyncstream(std::cout) << "Item " << i << " is processed by thread" << omp_get_thread_num() << std::endl;
  return 0;
}
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

Open in Compiler Explorer

An example of the use of for each.

```
1  #include "omp.h"
2  #include <iostream>
3  #include <syncstream>
4
5  int main() {
6  #pragma omp parallel
7    {
8      std::osyncstream(std::cout) << "Thread" <<
       ↪  omp_get_thread_num() << std::endl;
9
10 #pragma omp for
11     for (int i = 0; i < 10; i++)
12       std::osyncstream(std::cout) << "For i = " << i << " in t
         ↪  = " << omp_get_thread_num() << std::endl;
13   }
14
15   std::osyncstream(std::cout) << "Main thread\n";
16   return 0;
17 }
```

Open in Compiler Explorer

4.4 Exercises

A useful exercise may be to consider a simple ray tracer https://github.com/ssloy/tinyraytracer/blob/master/tinyraytracer.cpp with basic OpenMP parallelization and to try to target a GPGPU.

Chapter 5

The Syntax in SYCL

SYCL is an open specification for the design of code targeting either CPUs or GPGPUs that utilizes templates and lambda functions. By "open" in "open specification" we mean it is not tied specifically to NVIDIA. 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.

SYCL is a high-level specification, in that it builds on top of two lower-level standards, while aiming for succinct, modern C++ code, without any language extensions. First, SYCL is built on top of OpenCL and its device and execution models. Second, SYCL shares the intermediate representation of kernels, known as SPIR(-V), with OpenGL and Vulkan. SYCL aims to bring performance portability across various accelerators.

In the text, we omit the `sycl` namespace, where there is no risk of confusion, just as we have omitted the `std` earlier.

5.1 Motivation

In the previous chapter, we have seen the use of `target` to implement SAXPY. Now we clearly could have used some native, closed API, such as CUDA, if we had an NVIDIA GPGPU:

SAXPY in CUDA C++.

```
#include <cuda.h>
#include <iostream>
#include <vector>
#define N 1048576

__global__ void saxpy_kernel(float a, float *x, float *y,
↪  float *z) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    y[i] = a * x[i] + y[i];
}

int main() {
  std::vector<float> vx(N);
  float *x = vx.data();
  std::vector<float> vy(N);
  float *y = vy.data();
  std::vector<float> vz(N);
  float *z = vz.data();
  float *dx, *dy, *dz;
  cudaMalloc(&dx, N * sizeof(float));
  cudaMalloc(&dy, N * sizeof(float));
  cudaMalloc(&dz, N * sizeof(float));
  cudaMemcpy(dx, x, N * sizeof(float),
  ↪  cudaMemcpyHostToDevice);
  cudaMemcpy(dy, y, N * sizeof(float),
  ↪  cudaMemcpyHostToDevice);
  int nblocks = (n + 255) / 256;
  saxpy_kernel<<<nblocks, 256>>>(3.1415, dx, dy, dz);
  cudaMemcpy(z, dz, N * sizeof(float),
  ↪  cudaMemcpyDeviceToHost);
  cudaFree(dx);
  cudaFree(dy);
  cudaFree(dz);
  // we do not need free(x), free(y), free(z)
}
```

Open in Compiler Explorer

Admittedly, code in CUDA C++ still looks like C and is not easily portable to any other platform (AMD, Intel, ARM Mali). We could have also used Thrust, the CUDA C++ template library (https://thrust.github.io/):

SAXPY in Thrust, the CUDA C++ template library.

```cpp
// Based on
  ↪ https://github.com/NVIDIA/thrust/blob/main/examples/saxpy.cu

#include <iostream>
#include <thrust/copy.h>
#include <thrust/device_vector.h>
#include <thrust/fill.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

struct saxpy_functor {
  const float a;
  saxpy_functor(float _a) : a(_a) {}
  __host__ __device__ float operator()(const float &x, const
  ↪ float &y) const {
    return a * x + y;
  }
};

int main() {
  thrust::device_vector<float> dx(1048576);
  thrust::fill(dx.begin(), dx.end(), 1.0);
  thrust::device_vector<float> dy(1048576);
  thrust::fill(dx.begin(), dx.end(), 2.0);
  // Y <- A * X + Y
  // thrust::transform(dx.begin(), dx.end(), dy.begin(),
  ↪ dy.begin(), 3.1415f * _1 + _2);
  thrust::transform(dx.begin(), dx.end(), dy.begin(),
  ↪ dy.begin(), saxpy_functor(3.1415));
  thrust::copy(dy.begin(), dy.end(),
  ↪ std::ostream_iterator<float>(std::cout, "\n"));
}
```

Open in Compiler Explorer

The code is beautiful, but we it is not easily portable to another platform (AMD, Intel, ARM Mali). There may be several good reasons not to be tied specifically to NVIDIA:

- In consumer PC market, the biggest vendor of PC GPU is Intel with 71 percent of the market by unit count as of Q4 2022. NVIDIA has a market share of 17 percent and AMD 12 percent. In discrete PC GPU market, NVIDIA occupies 80-90% of the market, according to various estimates. Note that this market is rapidly declining.
- In mobile GPGPUs, ARM Mali is estimated to have close to 40% market share.
- In gaming consoles, both Xbox and PlayStation use accelerators

by AMD.

– In high-performance computing, NVIDIA is dominant, but the top 1 system (Frontier) currently uses AMD Instinct^{TM} 250X accelerators, and number of other systems use ARM architectures.

In this chapter, we hence consider SYCL, where the same example would resemble:

-bottom

SAXPY in SYCL, the open specification.

```
// Based on https://github.com/jeffhammond/dpcpp-tutorial

#include "CL/sycl.hpp"
#include <iostream>

class saxpy3;

int main(int argc, char *argv[]) {
  std::vector<float> vx(1048576, 1.0);
  std::vector<float> vy(1048576, 2.01);
  std::vector<float> vz(1048576, 0.0);
  sycl::queue q(sycl::default_selector{});
  try {
    const float A(aval);
    sycl::buffer<float, 1> dx{vx.data(),
    ↪  sycl::range<1>(vx.size())};
    sycl::buffer<float, 1> dy{vy.data(),
    ↪  sycl::range<1>(vy.size())};
    sycl::buffer<float, 1> dz{vz.data(),
    ↪  sycl::range<1>(vz.size())};

    q.submit([&](sycl::handler &h) {
      sycl::accessor x(dx, h, sycl::read_only);
      sycl::accessor y(dy, h, sycl::read_only);
      sycl::accessor z(dz, h, sycl::read_write);
      h.parallel_for<class saxpy3>(sycl::range<1>{length},
      ↪  [=](sycl::id<1> it) {
        const size_t i = it[0];
        z[I] += 3.1415 * x[i] + y[I];
      });
    });
    q.wait();
  } catch (sycl::exception &e) {
    std::cout << e.what() << std::endl;
    return 1;
  }
  return 0;
}
```

Open in Compiler Explorer

The code is also clearly modern C++, and can target accelerators by a number of vendors (NVIDIA, AMD, Intel, ARM), as well as FPGA.

5.2 More concepts

To make as efficient use of GPGPUs as possible, SYCL utilizes a number of concepts that we have not seen earlier. This builds upon the discussion of the GPGPU architecture in Section 2.3.2, page 12.

5.2.1 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.

In a `device_selector`, one overrides `int operator()(const sycl::device& dev) const overri` and returns an integer for the priority. The higher integer, the higher priority.

5.2.2 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 function object" for asynchronous execution.

At the simplest, a command group" is a callable (a named type, a lambda function, or std::function), which receives a "command group handler" from the SYCL as an argument of `operator()` so as to access the API. A command group submission to the queue is atomic.

A command group objects may also combine the callable and a set of requirements (edges of a task-graph), but if one declares the accessors for memory access correctly (see below), the task-graph will be constructed based on the accessors.

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()};`.

5.2.3 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 ("kernel grid" on NVIDIA), we partition the index-set of data 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). Ideally, 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.

The threads of one work group ("thread block" on NVIDIA) are sent to one Streaming Multiprocessors (SM), but one SM can execute threads by multiple work group in its multiple processing blocks. At most one work group per processing block.

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.

5.2.4 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.

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

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

using namespace sycl;

class exception1;

int main(int, char **) {
  auto exception_handler = [](exception_list exceptions) {
    for (std::exception_ptr const &e : exceptions) {
      try {
        std::rethrow_exception(e);
      } catch (exception const &e) {
        std::cout << "Caught asynchronous SYCL exception:\n"
                  << e.what() << std::endl;
      }
    }
  };

  queue q(default_selector{}, exception_handler);
  // actual use of the q

  try {
    q.wait_and_throw();
  } catch (exception const &e) {
    std::cout << "Caught synchronous SYCL exception:\n"
              << e.what() << std::endl;
  }
  return 0;
}
```

Open in Compiler Explorer

5.2.5 Unified shared memory

At the cost of some latency, one can use a unified shared memory across both the host and the device, wherein one uses 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[tid] = 0.0;
6   });
7 });
```

Open in Compiler Explorer

5.2.6 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 < 42; i++) {
12   assert(result[i] == 0);
13 }
```

Open in Compiler Explorer

5.2.7 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.

5.3 More complex examples

First, let us consider a complete, working example:

An example of vector addition in SYCL.

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

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 "
            << q.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(bufC, cgh, write_only);
      cgh.parallel_for<class 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 <cassert>
#include <cstdint>
#include <iostream>
#include <random>

#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

5.4 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.

```
1 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -fsycl
  ↪ -std=c++17")
2 set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -lOpenCL
  ↪ -lsycl")
```

Open in Compiler Explorer

If you target MKL:

Linking SYCL against MKL.

```
1 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -fsycl
  ↪ -std=c++17")
2 set(CMAKE_EXE_LINKER_FLAGS " -fsycl -lmkl_sycl
  ↪ -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core")
```

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:

Linking SYCL against AMD.

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

Open in Compiler Explorer

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

Chapter 6

Use Case 1: Linear Algebra and Machine Learning

In the first chapter, we have seen that within shared-memory parallel programming, we have broadly four options Confinement, Immutability, Use of Thread-safe Code, and Synchronization. More recently, we have seen four different implementations of a function called saxpy. Let us continue exploring the options on the use case of linear algebra and machine learning.

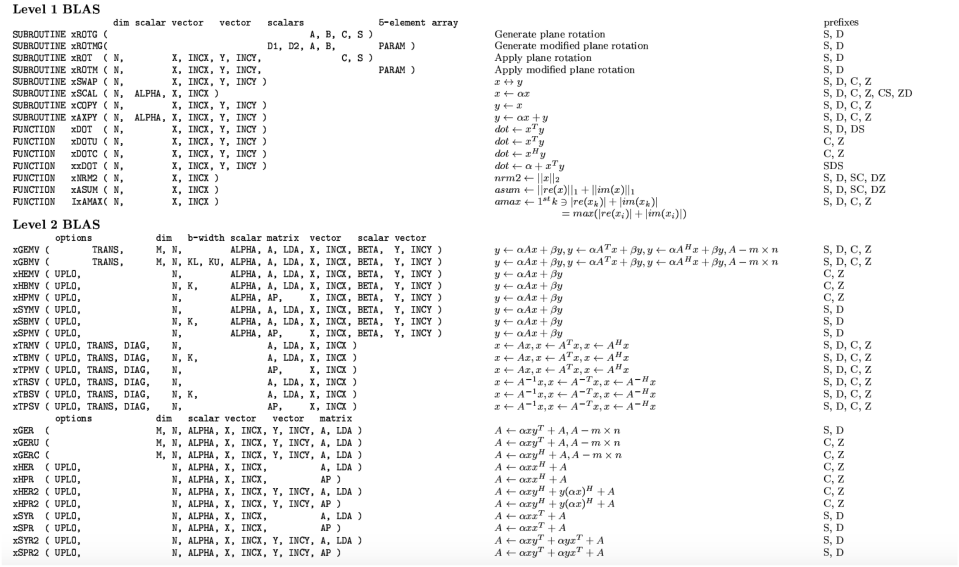
6.1 Thread-safe Code in C++20

6.1.1 BLAS

A key tool within linear algebra and machine learning are two ancient specifications, known as:

- BLAS ("Basic Linear Algebra Subprograms"), which covers vector addition, dot products, and linear combinations (this dates back to 1979). Level 2 added support for vector-matrix operations (1986), and level 3 added support for matrix-matrix operations and block-partitioned algorithms (1988).
- LAPACK ("Linear Algebra Package"), which covers matrix factorizations (LU, Cholesky and QR), eigenvalue and least squares solvers.

BLAS and LAPACK subroutines are all named naaop, where:



Level 1 BLAS

	dim scalar vector	vector	scalars	5-element array		prefixes
SUBROUTINE xROTG (			A, B, C, S)		Generate plane rotation	S, D
SUBROUTINE xROTMG(			D1, D2, A, B,	PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT (N,		X, INCX, Y, INCY,	C, S)		Apply plane rotation	S, D
SUBROUTINE xROTM (N,		X, INCX, Y, INCY,		PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP (N,		X, INCX, Y, INCY)			$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL (N,	ALPHA, X, INCX)				$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
SUBROUTINE xCOPY (N,		X, INCX, Y, INCY)			$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY (N,	ALPHA, X, INCX, Y, INCY)				$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT (N,		X, INCX, Y, INCY)			$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU (N,		X, INCX, Y, INCY)			$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC (N,		X, INCX, Y, INCY)			$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT (N,		X, INCX, Y, INCY)			$dot \leftarrow \alpha + x^T y$	SDS
FUNCTION xNRM2 (N,		X, INCX)			$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
FUNCTION xASUM (N,		X, INCX)			$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
FUNCTION IxAMAX(N,		X, INCX)			$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

options	dim	b-width	scalar	matrix	vector	scalar	vector		
xGEMV (TRANS,	M, N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xGBMV (TRANS,	M, N,	KL, KU,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xHEMV (UPLO,	N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHBMV (UPLO,	N,	K,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHPMV (UPLO,	N,		ALPHA,	AP,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xSYMV (UPLO,	N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSBMV (UPLO,	N,	K,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSPMV (UPLO,	N,		ALPHA,	AP,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xTRMV (UPLO, TRANS, DIAG,	N,			A, LDA,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTBMV (UPLO, TRANS, DIAG,	N,	K,		A, LDA,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTPMV (UPLO, TRANS, DIAG,	N,			AP,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTRSV (UPLO, TRANS, DIAG,	N,			A, LDA,	X, INCX)			$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTBSV (UPLO, TRANS, DIAG,	N,	K,		A, LDA,	X, INCX)			$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTPSV (UPLO, TRANS, DIAG,	N,			AP,	X, INCX)			$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z

options	dim	scalar	vector	matrix			$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
xGER (	M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)			
xGERU (	M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)		$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
xGERC (	M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)		$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
xHER (UPLO,	N,	ALPHA,	X, INCX,		A, LDA)		$A \leftarrow \alpha xx^H + A$	C, Z
xHPR (UPLO,	N,	ALPHA,	X, INCX,		AP)		$A \leftarrow \alpha xx^H + A$	C, Z
xHER2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)		$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xHPR2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	AP)		$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xSYR (UPLO,	N,	ALPHA,	X, INCX,		A, LDA)		$A \leftarrow \alpha xx^T + A$	S, D
xSPR (UPLO,	N,	ALPHA,	X, INCX,		AP)		$A \leftarrow \alpha xx^T + A$	S, D
xSYR2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)		$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
xSPR2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	AP)		$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Figure 6.1: A summary of the routines in BLAS levels 1 and 2, from the cheat sheet http://www.netlib.org/blas/blasqr.pdf.

- n suggests whether to use real floating-point numbers in single (S) or double (D) precision, or complex number with single (C) or double (Z) precision.
- aa denotes the assumptions on the matrix, e.g., diagonal (DI) specified by a vector, and general matrix (GE).
- op denotes the algorithm, e.g., matrix-matrix multiplication (MM), and solving linear system (SV).

SGEMM is thus matrix-matrix multiplication of general dense matrices in single precision, and DDOT is vector-vector dot product in double precision. See Figure 6.1 for an overview.

Most vendors of acclerators maintain their own BLAS implementation: AMD maintains rocBLAS, Apple maintains Accelerate, ARM maintains Arm Performance Libraries, Intel develops Intel Math Kernel Library (iMKL), and NVIDIA maintains cuBLAS and NVBLAS. Most libraries within linear algebra and machine learning supply BLAS user interface command structures, which makes it easy to transfer the code.

Notable open-source implementation focussing mostly on CPUs are ATLAS, BLIS (BLAS-like Library Instantiation Software), and OpenBLAS. A special mention should be devoted to the ATLAS library, which automatically optimizes itself for any architecture, including complicated cache hierarchies.

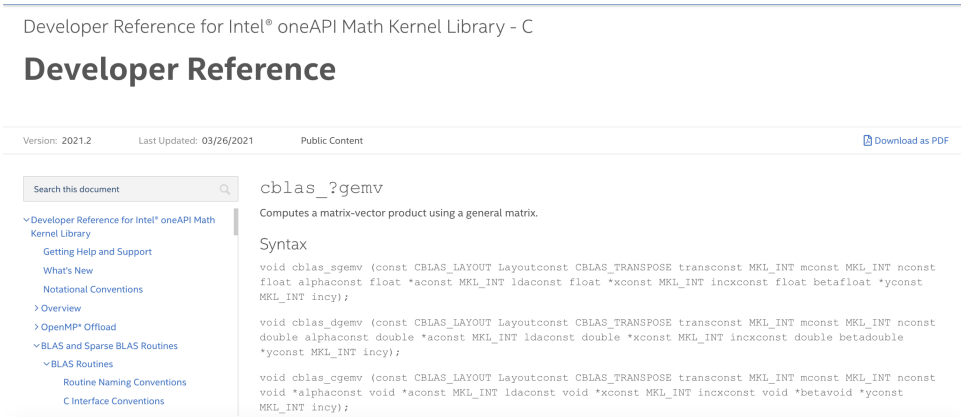


Figure 6.2: A screenshot from Intel OneAPI Mathematical Kernels.

In turn, many further libraries and toolboxes are built around BLAS, including MATLAB, Mathematica, NumPy, R, and Julia.

6.1.2 BLAS and C++

For C++ , there are multiple implementations of BLAS, loosely speaking. Libraries such as Armadillo, eigen, Intel OneAPI, LAPACK++, uBlas are sometimes linked against ancient Fortran code, but provide decent C++ interfaces. eigen and CLBlast actually provide C++ implementations too.

Intel OneAPI Mathematical Kernels comes with excellent documentation, examples, and support, but rather cumbersome naming conventions:

Example of the use of BLAS in C++ via Intel OneAPI.

```
1  #include "mkldnn.hpp"
2  #include <iostream>
3  #include <vector>
4
5  using namespace mkldnn;
6  using dim_t = mkldnn::memory::dim;
7
8  int main(int argc, char **argv) {
9    try {
10     const dim_t n = 64;
11     // column-major order (sloupce souvisle)
12     std::vector<float> A(n * n, 1.0f);
13     std::vector<float> B(n * n, 1.0f);
14     std::vector<float> C(n * n, 1.0f);
```

```
15      //
        ↪  https://oneapi-src.github.io/oneDNN/v0/group__c__api__blas.html
16      mkldnn_status_t status = mkldnn_sgemm('N', 'N', n, n, n,
17                                            1.f, A.data(), n,
                                              ↪  B.data(), n,
18                                            0.f, C.data(), n);
19      std::cerr << "status: " << status << std::endl;
20    } catch (error &e) {
21      std::cerr << "status: " << e.status << std::endl;
22      std::cerr << "message: " << e.message << std::endl;
23    }
24    return 0;
25  }
```

Open in Compiler Explorer

Boost.org uBlas provides a much more modern C++20-only syntax, and can be linked to arbitrary vendor-provided BLAS library. The example above could be rewritten, e.g., as:

Example of the use of BLAS in C++ via Boost.org uBlas.

```
1  #include <boost/numeric/ublas/tensor.hpp>
2  #include <iostream>
3
4  int main() {
5    using namespace boost::numeric::ublas::index;
6    using tensor = boost::numeric::ublas::tensor_dynamic<float>;
7    auto ones = boost::numeric::ublas::ones<float>{};
8
9    tensor A = ones(64, 64);
10   tensor B = ones(64, 64);
11
12   tensor C = A(_i, _j) * B(_i, _j);
13   std::cout << "C=" << C << ";" << std::endl;
14 }
```

Open in Compiler Explorer

As is clear from the preceding example, one can extend this well beyond matrices:

A more elaborate example of the use of BLAS in C++ via Boost.org uBlas.

```
1  #include <boost/numeric/ublas/tensor.hpp>
2  #include <iostream>
3
4  int main() {
```

```
5    using namespace boost::numeric::ublas::index;
6    using tensor = boost::numeric::ublas::tensor_dynamic<float>;
7    auto ones = boost::numeric::ublas::ones<float>{};
8
9    tensor A = ones(3, 4, 5);
10   tensor B = ones(4, 6, 3, 2);
11
12   tensor C = 2 * ones(5, 6, 2) + A(_i, _j, _k) * B(_j, _l, _i,
     ↪  _m) + 5;
13   std::cout << "C=" << C << ";" << std::endl;
14 }
```

Open in Compiler Explorer

6.1.3 Scaling to Supercomputers

As you may know, the tensor computations underlie much of modern machine learning, in the form of training deep neural networks. The TensorFlow and PyTorch are key contenders there, again easy to link against any vendor-provided BLAS.

Libraries such as TensorFlow make make it possible to exploit much of the theoretically available processing power. See, for example, the plots in Figure 6.3 and many further plots at https://code.ornl.gov/olcf-analytics/summit/distributed-deep-learning-examples. This concerns the scaling of a code for training ResNet deep neural network on ImageNet benchmark on a supercomputer at the Oak Ridge National Laboratory in Tennessee, USA. The supercomputer has 9,216 POWER9 22-core CPUs and 27,648 NVIDIA Tesla V100 GPUs, each of which has 5,120 CUDA Cores. In total, this means 141,557,760 cores with circa 200 petaFLOPS performance. With a substantial amount of engineering and auto-tuning (using Horovod, cf. [5]), one can achieve near-linear scaling of performance (in terms of numbers of images trained per second) as a function of the number of cores. This, effectively, renders implementations "from scratch" unnecessary.

6.2 Synchronization

While it is very hard to see why one would like to implement one's own replacement of BLAS or LAPACK, it is instructive to see several steps of the development of common linear-algebraic routines. Let us explore matrix-matrix multiplication and solving of linear systems, as two examples.

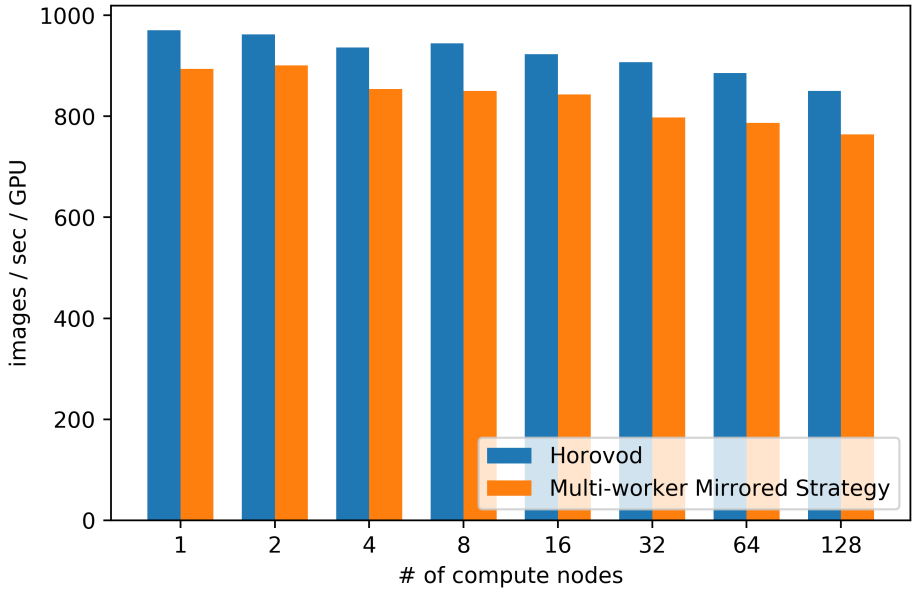


Figure 6.3: A summary of scaling of training ResNet on ImageNet in TensorFlow on Summit.

6.2.1 Matrix-Vector Multiplication

Matrix-vector multiplication is a key primitive throughout linear algebra. Considering that the computation:

$$y_j = \sum_i a_{ji} x_i \tag{6.1}$$

for each coordinate j in the resulting vector y, can be computed in parallel, independently of any other, it is almost embarrassingly (data) parallel. Still, there are several important improvements over the basic version:

An example of matrix multiplication in OpenMP.

```
1  void multiply(std::vector<int> &A, std::vector<int> &x,
   ↪  std::vector<int> &y) {
2  #pragma omp declare reduction(vec_int_plus        \
3                                : std::vector <int> \
4                                :
   ↪  std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus <int>()))
   ↪  initializer(omp_priv = omp_orig)
5
```

```
6  #pragma omp parallel for num_threads(thread_count)
   ↪  reduction(vec_int_plus \
7                                                          :
   ↪  y)
8    for (int i = 0; i < ROWS; i++) {
9      for (int j = 0; j < COLS; j++) {
10       y[i] += A[i * COLS + j] * x[j];
11     }
12   }
13 }
```

Open in Compiler Explorer

Notably, one could aim to parallelize the processing of each row of the matrix. This is difficult, due to the need for synchronization in the addition into the result. One option would be to consider some auxiliary variable z, which would be computed column-wise:

$$z_{ij} = a_{ij} x_j \tag{6.2}$$

$$y_i = \sum_j z_i j \tag{6.3}$$

This would be difficult, in terms of the cache hierarchies. We could, however, reorder the memory:

An example of matrix multiplication in OpenMP.

```
1  ...
2      // data has to be ordered by columns in memory
3      for (int i = 0; i < COLS; i++) {
4    x[i] = rand() % 1000;
5    for (int j = 0; j < ROWS; j++) {
6      A[i * ROWS + j] = rand() % 1000;
7    }
8  }
9  ...
10
11     void
12     multiply_column(std::vector<int> &A, std::vector<int> &x,
       ↪  std::vector<int> &y) {
13 #pragma omp declare reduction(vec_int_plus        \
14                                : std::vector <int> \
15                                :
   ↪  std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus <int>()))
   ↪  initializer(omp_priv = omp_orig)
16
```

```
17  #pragma omp parallel for num_threads(thread_count)
    ↪  reduction(vec_int_plus \
18                                                          :
    ↪  y)
19    for (int i = 0; i < COLS; i++) {
20      for (int j = 0; j < ROWS; j++) {
21        y[j] += A[i * ROWS + j] * x[i];
22      }
23    }
24  }
```

Open in Compiler Explorer

Alternatively, one can use the single most useful and embarrassingly simple trick of parallel programming: introducing local variable.

An example of matrix multiplication in OpenMP.

```
1   void multiply(std::vector<int> &A, std::vector<int> &x,
    ↪  std::vector<int> &y) {
2   #pragma omp declare reduction(vec_int_plus        \
3                                 : std::vector <int> \
4                                 :
    ↪  std::transform(omp_out.begin(), omp_out.end(),
    ↪  omp_in.begin(), omp_out.begin(), std::plus <int>()))
    ↪  initializer(omp_priv = omp_orig)
5
6     int tmp;
7   #pragma omp parallel for num_threads(thread_count)
    ↪  reduction(vec_int_plus \
8                                                           :
    ↪  y)
9     for (int i = 0; i < ROWS; i++) {
10      tmp = 0;
11      for (int j = 0; j < COLS; j++) {
12        tmp += A[i * COLS + j] * x[j];
13      }
14      y[i] += tmp;
15    }
16  }
```

Open in Compiler Explorer

6.2.2 Matrix-Matrix Multiplication

Matrix-matrix multiplication is, likewise, very common and to some extent embarrassingly parallel. For large-enough matrices, the trivial parallelization leads to too many small(ish) tasks.

An example of matrix multiplication in OpenMP.

```
1  void multiply(std::vector<int> &A, std::vector<int> &B,
   ↪  std::vector<int> &C) {
2  #pragma omp declare reduction(vec_int_plus        \
3                                : std::vector <int> \
4                                :
   ↪  std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus <int>()))
   ↪  initializer(omp_priv = omp_orig)
5    int tmp;
6  #pragma omp parallel for collapse(2) num_threads(thread_count)
   ↪  reduction(vec_int_plus \
7
   ↪  : C)
8    for (int i = 0; i < ROWS; i++) {
9      for (int j = 0; j < COLS; j++) {
10       tmp = 0;
11       for (int k = 0; k < ROWS; k++) {
12         tmp += A[i * COLS + k] * B[k * COLS + j];
13       }
14       C[i * COLS + j] += tmp;
15     }
16   }
17 }
```

Open in Compiler Explorer

A more elaborate version considers blocking. This is actually very close to the state of the art algorithms:

An example of matrix multiplication in OpenMP.

```
1  void multiply_blocks(std::vector<int> &A, std::vector<int> &B,
   ↪  std::vector<int> &C) {
2  #pragma omp declare reduction(vec_int_plus        \
3                                : std::vector <int> \
4                                :
   ↪  std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus <int>()))
   ↪  initializer(omp_priv = omp_orig)
5
6    const int ROWS_IN_BLOCK = 10;
7    const int BLOCKS_IN_ROW = ROWS / ROWS_IN_BLOCK;
8    int tmp;
9
10 #pragma omp parallel for collapse(2) num_threads(thread_count)
   ↪  reduction(vec_int_plus \
11
   ↪  : C) private(tmp)
```

```
12    for (int br1 = 0; br1 < BLOCKS_IN_ROW; br1++) {
13      for (int bb = 0; bb < BLOCKS_IN_ROW; bb++) {
14        for (int bc2 = 0; bc2 < BLOCKS_IN_ROW; bc2++) {
15          for (int r = br1 * ROWS_IN_BLOCK; r < (br1 + 1) *
            ↪  ROWS_IN_BLOCK; r++) {
16            for (int c = bc2 * ROWS_IN_BLOCK; c < (bc2 + 1) *
              ↪  ROWS_IN_BLOCK; c++) {
17              tmp = 0;
18              for (int k = 0; k < ROWS_IN_BLOCK; k++) {
19                tmp += A[r * COLS + (k + bb * ROWS_IN_BLOCK)] *
                  ↪  B[(bb * ROWS_IN_BLOCK + k) * COLS + c];
20              }
21              C[r * COLS + c] += tmp;
22            }
23          }
24        }
25      }
26    }
27  }
```

Open in Compiler Explorer

6.2.3 Solving Linear Systems

In commonly-known approaches to solving linear systems, the parallelization is less trivial. For example in Gauss elimination, once you pick a pivot, a large submatrix (whose size depends on the pivot) needs to change. See for example:

An example of Gauss in OpenMP.

```
1  void gauss_par(std::vector<double> &A) {
2  #pragma omp declare reduction(vec_int_plus           \
3                                : std::vector <double> \
4                                :
   ↪  std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus <double>()))
   ↪  initializer(omp_priv = omp_orig)
5    for (int i = 0; i < ROWS; i++) {
6      // Make all rows below this one 0 in current column
7  #pragma omp parallel for num_threads(thread_count)
8      for (int k = i + 1; k < ROWS; k++) {
9        double c = -A[k * COLS + i] / A[i * COLS + i];
10       for (int j = i; j < ROWS; j++) {
11         if (i == j) {
12           A[k * COLS + j] = 0;
13         } else {
14           A[k * COLS + j] += c * A[i * COLS + j];
15         }
```

```
16        }
17      }
18    }
19  }
```

Open in Compiler Explorer

Notice, however, that one can rephrase the solving of a linear system in terms of solving a least squares problem (minimizing the residual) and that modern optimization methods are competitive with tailor-made solvers for linear systems cf. [**?**]. One can hence focus on parallelizing optimization methods.

6.2.4 Optimization for Machine Learning

Much of machine learning is embarassingly parallel, incl. hyperparameter search, or data processing in Computer Vision, where one often has to process several streams of video in parallel, and sometimes can process several images or frames of a video in parallel.

Much else in machine learning is actually optimization, which can be parallelized efficiently, either in coordinate descent methods, or via stochastic gradient methods.

For one example of such embarrassingly parallel computation, see [6], which was originally developed for monitoring data from a city-wide camera network of a European capital. In one version of the code, similar to the snippet at https://github.com/jmarecek/OnlineLowRank, the goal was to speed up the computation per single stream of video, such that each core handles one stream from a 4K camera in real-time.

For another example of parallel programming in machine learning, see [7], and the corresponding code at: https://github.com/optml/ac-dc/blob/master/cpp/src/solver/matrixCompletion/parallel/parallel_mc_opemmp.h.

Chapter 7

Use Case 2: Sorting

In the first chapter, we have seen that within shared-memory parallel programming, we have broadly four options Confinement, Immutability, Use of Thread-safe Code, and Synchronization. Let us now explore the latter two options on the use case of sorting. As Table 7.1 suggests, sorting is Nick's class (cf. p. 11): for n items, we can consider $O(n^2)$ pairs of items in parallel, compare them to obtain a binary value, and then for each item, obtain its rank in the sorted order by adding the binary values. This is known as the parallel ranking.

There are many other parallel algorithms based on picking minimum and maximum from a small set, as well as algorithms based on hashing.

Algorithm	$p(n)$ Processors	Time
Sequential algorithms	1	$O(n \log n)$
Parallel divide and conquer	$O(1)$	$O(n \log n)$
	$O(\log n)$	$O\left(\frac{n(\log n)}{p(n)}\right)$
	$\omega(\log n)$	$O(n)$
Parallel Ranking	$O(n^2)$	$O(\log n)$

Table 7.1: Parallelisability of common sorting algorithms, following [1].

7.1 Thread-safe Code in C++20

A prime example of the use of thread-safe code in C++ is the standard template library (STL) with a suitable execution policy. We have seen that 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
```

Example of parallel sorting in STL.

```cpp
#include <algorithm>
#include <chrono>
#include <execution>
#include <iostream>
#include <random>
#include <vector>
using namespace std::chrono;

int main() {
  const int N = 1000000;
  std::vector<int> v(N);
  std::mt19937 rng;
  rng.seed(std::random_device()());
  std::uniform_int_distribution<int> dist(0, 255);
  std::generate(begin(v), end(v), [&]() { return dist(rng); });
  auto start = high_resolution_clock::now();
  std::sort(std::execution::par, begin(v), end(v));
  auto finish = high_resolution_clock::now();
  auto duration = duration_cast<milliseconds>(finish - start);
  std::cout << "\nElapsed time = " << duration.count() << " ms\n";
  return 0;
}
```

Open in Compiler Explorer

Now, knowing what your implementation of standard template library (STL) would actually use is non-trivial. One may imagine a code such as:

Example of a quick sort using STL.

```cpp
template <class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last) {
  if (first == last)
    return;
  std::size_t distance = std::distance(first, last);
  auto pivot = *std::next(first, distance / 2);
  ForwardIt middle1;
  ForwardIt middle2;
  if (distance < threshold) {
    middle1 = std::partition(std::execution::seq, first, last,
    ↪ [pivot](const auto &em) { return em < pivot; });
    middle2 = std::partition(std::execution::seq, middle1,
    ↪ last, [pivot](const auto &em) { return !(pivot < em);
    ↪ });
```

```
12    } else {
13      middle1 = std::partition(std::execution::par, first, last,
        ↪  [pivot](const auto &em) { return em < pivot; });
14      middle2 = std::partition(std::execution::par, middle1,
        ↪  last, [pivot](const auto &em) { return !(pivot < em);
        ↪  });
15    }
16    quicksort(first, middle1);
17    quicksort(middle2, last);
18  }
```

Open in Compiler Explorer

In the Intel Thread Building Blocks (TBB) backend of the GCC implementation, you would actually find:

Example of parallel sorting implementation in STL.

```
1  template <class _ExecutionPolicy, typename
   ↪  _RandomAccessIterator, typename _Compare, typename
   ↪  _LeafSort>
2  void __parallel_stable_sort(_ExecutionPolicy &&,
   ↪  _RandomAccessIterator __xs, _RandomAccessIterator __xe,
   ↪  _Compare __comp,
3                              _LeafSort __leaf_sort, std::size_t
                               ↪  __nsort = 0) {
4    tbb::this_task_arena::isolate([=, &__nsort]() {
5      // sorting based on task tree and parallel merge
6      typedef typename
       ↪  std::iterator_traits<_RandomAccessIterator>::value_type
       ↪  _ValueType;
7      typedef typename
       ↪  std::iterator_traits<_RandomAccessIterator>::difference_type
       ↪  _DifferenceType;
8      const _DifferenceType __n = __xe - __xs;
9      if (__nsort == __n)
10       __nsort = 0; // 'partial_sort' becames 'sort'
11
12     const _DifferenceType __sort_cut_off =
       ↪  _PSTL_STABLE_SORT_CUT_OFF;
13     if (__n > __sort_cut_off) {
14       __buffer<_ValueType> __buf(__n);
15       __root_task<__stable_sort_func<_RandomAccessIterator,
         ↪  _ValueType *, _Compare, _LeafSort>> __root{
16           __xs, __xe, __buf.get(), true, __comp, __leaf_sort,
             ↪  __nsort, __xs, __buf.get()};
17       __task::spawn_root_and_wait(__root);
18       return;
19     }
20     // serial sort
```

```
21      __leaf_sort(__xs, __xe, __comp);
22    });
23  }
```

Open in Compiler Explorer

This uses a "sorting based on task tree and parallel merge", while making use of several non-trivial tricks, including `tbb::task_scheduler_init`, `std::thread::hardware_concurrency()`, and `std::hardware_constructive_interfer` (Contrast this with the serial version of GCC sort, which uses a multi-way mergesort, and GCC stable_sort, which uses a quicksort.) We wish to make use of the STL, rather than redevelop it, in the first instance.

Even making full use of the STL is quite non-trivial. Here, we present an overview of the sorting-related routines in verbatim from the fantastic book "A Complete Guide to Standard C++ Algorithms" of Simon Toth [8], in compliance with the license.

7.1.1 `strict_weak_ordering`

First, however, let us introduce the `strict_weak_ordering` "comparator" required, following the material of Simon Toth. Implementing a `strict_weak_ordering` for a custom type at minimum requires providing an overload of `operator<` with the following behaviour:

- irreflexive $\neg f(a, a)$
- anti-symmetric $f(a, b) \Rightarrow \neg f(b, a)$
- transitive $(f(a, b) \wedge f(b, c)) \Rightarrow f(a, c)$

A good default for a `strict_weak_ordering` implementation is lexicographical ordering. Lexicographical ordering is also the ordering provided by standard containers. C++20 introduced the spaceship operator, whereby user-defined types can easily access the default version of lexicographical ordering.

C++20 does not enforce that the comparator is thread safe, although that matters greatly for the thread-safe nature of the whole sorting algorithm!

Example of three approaches to implementing lexicographical comparison for a custom type *that are not thread safe!*.

```cpp
struct Point {

  int x;
  int y;

  // pre-C++20 lexicographical less-than
  friend bool operator<(const Point &left, const Point &right) {
    if (left.x != right.x)
      return left.x < right.x;
    return left.y < right.y;
  }

  // default C++20 spaceship version of lexicographical comparison
  friend auto operator<=>(const Point &, const Point &) = default;

  // manual version of lexicographical comparison using operator <=>
  friend auto operator<=>(const Point &left, const Point &right) {
    if (left.x != right.x)
      return left.x <=> right.x;
    return left.y <=> right.y;
  }
};
```

Open in Compiler Explorer

The default lexicographical ordering (line 14) works recursively. It starts with the object's bases first, left-to-right, depth-first and then non-static members in declaration order (processing arrays element by element, left-to-right).

The type returned for the spaceship operator is the common comparison category type for the bases and members, one of:

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`

7.1.2 `std::lexicographical_compare`

Lexicographical `strict_weak_ordering` for ranges is exposed through the `std::lexicographical_compare` algorithm.

lex...compare	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	(`input_range`, `input_range`)	
parallel domain	(`forward_range`, `forward_range`)	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `lexicographical_compare` and the built-in less than operator to compare vectors of integers.

```
1  std::vector<int> range1{1, 2, 3};
2  std::vector<int> range2{1, 3};
3  std::vector<int> range3{1, 3, 1};
4
5  bool cmp1 = std::lexicographical_compare(range1.begin(),
   ↪  range1.end(),
6                                           range2.begin(),
                                            ↪  range2.end());
7  // same as
8  bool cmp2 = range1 < range2;
9  // cmp1 == cmp2 == true
10
11 bool cmp3 = std::lexicographical_compare(range2.begin(),
   ↪  range2.end(),
12                                          range3.begin(),
                                            ↪  range3.end());
13 // same as
14 bool cmp4 = range2 < range3;
15 // cmp3 == cmp4 == true
```

Open in Compiler Explorer

Because the standard containers already offer a built-in lexicographical comparison, the algorithm mainly finds use for comparing raw C arrays and in cases when we need to specify a custom comparator.

Example of using `lexicographical_compare` for C-style arrays and customizing the comparator.

```cpp
// for demonstration only, prefer std::array
int x[] = {1, 2, 3};
int y[] = {1, 4};

bool cmp1 = std::lexicographical_compare(&x[0], &x[3], &y[0], &y[2]);
// cmp1 == true

std::vector<std::string> names1{"Zod", "Celeste"};
std::vector<std::string> names2{"Adam", "Maria"};

bool cmp2 = std::ranges::lexicographical_compare(names1, names2,
                                                 [](const std::string &left, const std::string &right) {
                                                   return left.length() < right.length();
                                                 });
// different than
bool cmp3 = names1 < names2; // Zod > Adam
// cmp2 == true, cmp3 == false
```

Open in Compiler Explorer

7.1.3 `std::lexicographical_compare_three_way`

The `std::lexicographical_compare_three_way` is the spaceship operator equivalent to `std::lexicographical_compare`. It returns one of:

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`

The type depends on the type returned by the elements' spaceship operator.

lex...three_way	
introduced	C++20
constexpr	C++20
parallel	N/A
rangified	N/A

constraints		
domain	`(input_range, input_range)`	
invocable	default	custom
	`operator<=>`	`strong_ordering`, `weak_ordering`, `partial_ordering`

Example of using `std::lexicographical_compare_three_way`.

```
1  std::vector<int> data1 = {1, 1, 1};
2  std::vector<int> data2 = {1, 2, 3};
3
4  auto cmp = std::lexicographical_compare_three_way(
5      data1.begin(), data1.end(),
6      data2.begin(), data2.end());
7  // cmp == std::strong_ordering::less
```

Open in Compiler Explorer

sort	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

7.1.4 `std::sort`

The `std::sort` algorithm is the canonical $O(n \log n)$ sort.

constraints		
domain	`random_access_range`	
parallel domain	`random_access_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Due to the $O(n \log n)$ complexity guarantee, `std::sort` only operates on `random_access` ranges. Notably, `std::list` offers a method with approximately $n \log n$ complexity.

Basic example of using `std::sort` and `std::list::sort`.

```
1  std::vector<int> data1 = {9, 1, 8, 2, 7, 3, 6, 4, 5};
2  std::sort(data1.begin(), data1.end());
3  // data1 == {1, 2, 3, 4, 5, 6, 7, 8, 9}
4
5  std::list<int> data2 = {9, 1, 8, 2, 7, 3, 6, 4, 5};
6  // std::sort(data.begin(), data.end()); // doesn't compile
7  data2.sort();
8  // data2 == {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Open in Compiler Explorer

With C++20, we can take advantage of projections to sort by a method or member:

Example of using a projection in conjunction with a range algorithm. The algorithm will sort the elements based on the values obtained by invoking the method `value` on each element.

```
1 struct Account {
2   double value() { return value_; }
3   double value_;
4 };
5
6 std::vector<Account> accounts{{0.1}, {0.3}, {0.01}, {0.05}};
7 std::ranges::sort(accounts, std::greater<>{},
  ↪  &Account::value);
8 // accounts = { {0.3}, {0.1}, {0.05}, {0.01} }
```

Open in Compiler Explorer

Before C++14, you would have to fully specify the type of the comparator, i.e. `std::greater<double>{}`. The type erased variant `std::greater<>{}` relies on type deduction to determine the parameter types. Projections accept an unary invocable, including pointers to members and member functions.

7.1.5 `std::stable_sort`

The `std::sort` is free to re-arrange equivalent elements, which can be undesirable when re-sorting an already sorted range. The `std::stable_sort` provides the additional guarantee of preserving the relative order of equal elements.

stable_sort	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

constraints		
domain	`random_access_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

If additional memory is available, stable_sort remains $O(n \log n)$. However, if it fails to allocate, it will degrade to an $O(n \log n \log n)$ algorithm.

Example of re-sorting a range using `std::stable_sort`, resulting in a guaranteed order of elements.

```cpp
struct Record {
  std::string label;
  int rank;
};

std::vector<Record> data{{"q", 1}, {"f", 1}, {"c", 2}, {"a", 1}, {"d", 3}};

std::ranges::stable_sort(data, {}, &Record::label);
std::ranges::stable_sort(data, {}, &Record::rank);
// Guarantted order: a-1, f-1, q-1, c-2, d-3
```

Open in Compiler Explorer

7.1.6 `std::is_sorted`

is_sorted	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::is_sorted` algorithm is a linear check returning a boolean denoting whether the ranges elements are in non-descending order.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::less`	`strict_weak_ordering`

Example of testing a range using `std::is_sorted`.

```cpp
std::vector<int> data1 = {1, 2, 3, 4, 5};
bool test1 = std::is_sorted(data1.begin(), data1.end());
// test1 == true

std::vector<int> data2 = {5, 4, 3, 2, 1};
bool test2 = std::ranges::is_sorted(data2);
// test2 == false
bool test3 = std::ranges::is_sorted(data2, std::greater<>{});
// test3 == true
```

Open in Compiler Explorer

7.1.7 `std::is_sorted_until`

is_sorted_until	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::is_sorted_until` algorithm returns the first out-of-order element in the given range, thus denoting a sorted sub-range.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::less`	`strict_weak_ordering`

Example of testing a range using `std::is_sorted_until`.

```
1 std::vector<int> data{1, 5, 9, 2, 4, 6};
2 auto it = std::is_sorted_until(data.begin(), data.end());
3 // *it == 2
```

Open in Compiler Explorer

Note that because of the behaviour of `std::is_sorted_until`, the following is always true:
`std::is_sorted(r.begin(), std::is_sorted_until(r.begin(), r.end()))`

7.1.8 `std::partial_sort`

The `std::partial_sort` algorithm reorders the range's elements such that the leading sub-range is in the same order it would when fully sorted. However, the algorithm leaves the rest of the range in an unspecified order.

partial_sort	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(random_access_range, random_access_iterator)`	
parallel domain	`(random_access_range, random_access_iterator)`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

The benefit of using a partial sort is faster runtime — approximately $O(n \log k)$, where k is the number of elements sorted.

Example of using `std::partial_sort` to sort the first three elements of a range.

```
1 std::vector<int> data{9, 8, 7, 6, 5, 4, 3, 2, 1};
2 std::partial_sort(data.begin(), data.begin() + 3, data.end());
3 // data == {1, 2, 3, -unspecified order-}
4
5 std::ranges::partial_sort(data, data.begin() + 3,
  ↪  std::greater<>());
6 // data == {9, 8, 7, -unspecified order-}
```

Open in Compiler Explorer

7.1.9 `std::partial_sort_copy`

partial_sort_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::partial_sort_copy` algorithm has the same behaviour as `std::partial_sort`; however, it does not operate inline. Instead, the algorithm writes the results to a second range.

constraints		
domain	`input_range -> random_access_range`	
parallel domain	`forward_range -> random_access_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

The consequence of writing output to a second range is that the source range does not have to be mutable nor provide random access.

Example of using `std::partial_sort_copy` to iterate over ten integers read from standard input and storing the top three values in sorted order.

```
1  // input == "0 1 2 3 4 5 6 7 8 9"
2  std::vector<int> top(3);
3
4  auto input = std::istream_iterator<int>(std::cin);
5  auto cnt = std::counted_iterator(input, 10);
6
7  std::ranges::partial_sort_copy(cnt, std::default_sentinel,
8                                 top.begin(), top.end(),
9                                 std::greater<>{});
10 // top == { 9, 8, 7 }
```

Open in Compiler Explorer

7.2 Synchronization

Let us now use synchronization primitives to implement simple sorting algorithms.

7.2.1 Quick sort with user-level threads

For instance, let us consider quick-sort, for a start:

An example of parallel quick sort in OpenMP.

```
1  void qs(std::vector<int> &vector_to_sort, int from, int to) {
2    if (to - from <= base_size) {
```

```
3       std::sort(vector_to_sort.begin() + from,
        ↪  vector_to_sort.begin() + to);
4       return;
5     }
6
7     // cf. the pivot (vector_to_sort[from])
8     int part2_start = partition(vector_to_sort, from, to,
      ↪  vector_to_sort[from]);
9
10    if (part2_start - from > 1) {
11  #pragma omp task shared(vector_to_sort) firstprivate(from,
    ↪  part2_start)
12      {
13        qs(vector_to_sort, from, part2_start);
14      }
15    }
16    if (to - part2_start > 1) {
17      qs(vector_to_sort, part2_start, to);
18    }
19  }
```

Open in Compiler Explorer

Intel suggests a three-way quick-sort (https://software.intel.com/content/www/us/en/develop/articles/an-efficient-parallel-three-way-quicksort-using-html) using the task construct:

An example of three-way parallel quick sort in OpenMP.

```
1   template <class RanIt, class _Pred>
2   void qsort3w(RanIt _First, RanIt _Last, _Pred compare) {
3     if (_First >= _Last)
4       return;
5
6     std::size_t _Size = 0L;
7     g_depth++;
8     if ((_Size = std::distance(_First, _Last)) > 0) {
9       RanIt _LeftIt = _First, _RightIt = _Last;
10      bool is_swapped_left = false, is_swapped_right = false;
11      typename std::iterator_traits<RanIt>::value_type _Pivot =
        ↪  *_First;
12
13      RanIt _FwdIt = _First + 1;
14      while (_FwdIt <= _RightIt) {
15        if (compare(*_FwdIt, _Pivot)) {
16          is_swapped_left = true;
17          std::iter_swap(_LeftIt, _FwdIt);
18          _LeftIt++;
19          _FwdIt++;
```

```
20        } else if (compare(_Pivot, *_FwdIt)) {
21          is_swapped_right = true;
22          std::iter_swap(_RightIt, _FwdIt);
23          _RightIt--;
24        } else
25          _FwdIt++;
26      }
27
28      if (_Size >= cutoff) {
29  #pragma omp taskgroup
30        {
31  #pragma omp task untied mergeable
32          if ((std::distance(_First, _LeftIt) > 0) &&
            ↪  (is_swapped_left))
33            qsort3w(_First, _LeftIt - 1, compare);
34
35  #pragma omp task untied mergeable
36          if ((std::distance(_RightIt, _Last) > 0) &&
            ↪  (is_swapped_right))
37            qsort3w(_RightIt + 1, _Last, compare);
38        }
39      } else {
40  #pragma omp task untied mergeable
41        {
42          if ((std::distance(_First, _LeftIt) > 0) &&
            ↪  is_swapped_left)
43            qsort3w(_First, _LeftIt - 1, compare);
44
45          if ((std::distance(_RightIt, _Last) > 0) &&
            ↪  is_swapped_right)
46            qsort3w(_RightIt + 1, _Last, compare);
47        }
48      }
49    }
50  }
```

Open in Compiler Explorer

7.2.2 Merge sort with user-level threads

Similarly, one could parallelize merge sort. Let us consider a simple merge sort:

An example of merge sort.

```
1  void ms_serial(std::vector<int> &vector_to_sort, int from, int
   ↪  to) {
2    if (to - from <= 1) {
3      return;
```

```
  }
  int middle = (to - from) / 2 + from;

  ms_serial(vector_to_sort, from, middle);
  ms_serial(vector_to_sort, middle, to);
  std::inplace_merge(vector_to_sort.begin() + from,
  ↪  vector_to_sort.begin() + middle, vector_to_sort.begin()
  ↪  + to);
}

void ms(std::vector<int> &vector_to_sort, int from, int to) {
  if (to - from <= base_size) {
    ms_serial(vector_to_sort, from, to);
    return;
  }
  int middle = (to - from) / 2 + from;

  ms(vector_to_sort, from, middle);
  ms(vector_to_sort, middle, to);

  std::inplace_merge(vector_to_sort.begin() + from,
  ↪  vector_to_sort.begin() + middle, vector_to_sort.begin()
  ↪  + to);
}
```

Open in Compiler Explorer

Using the task construct in OpenMP, we can parallelize this as:

An example of merge sort in OpenMP.

```
void ms_parallel(std::vector<int> &vector_to_sort, int from,
↪  int to) {
  if (to - from <= 1) {
    return;
  }
  int middle = (to - from) / 2 + from;

  ms_serial(vector_to_sort, from, middle);
  ms_serial(vector_to_sort, middle, to);
  std::inplace_merge(vector_to_sort.begin() + from,
  ↪  vector_to_sort.begin() + middle, vector_to_sort.begin()
  ↪  + to);
}

void ms(std::vector<int> &vector_to_sort, int from, int to) {
  if (to - from <= base_size) {
    ms_serial(vector_to_sort, from, to);
    return;
  }
```

```
17    int middle = (to - from) / 2 + from;
18
19  #pragma omp task shared(vector_to_sort) firstprivate(from,
    ↪  middle)
20    ms(vector_to_sort, from, middle);
21
22    ms(vector_to_sort, middle, to);
23
24  #pragma omp taskwait
25    std::inplace_merge(vector_to_sort.begin() + from,
      ↪  vector_to_sort.begin() + middle, vector_to_sort.begin()
      ↪  + to);
26  }
```

Open in Compiler Explorer

These examples of quick sort and merge sort are actually competitive. On GPGPUs, one may wish to consider Odd-Even Merge Sort:

An example of odd-even merge sort.

```
1  void odd - even - merge(std::vector<int> &vector_to_sort, int
   ↪  from, int to, int step) {
2    auto new_step = step * 2;
3    if (new_step < to - from) {
4      odd - even - merge(vector_to_sort, from, to, new_step);
5      odd - even - merge(vector_to_sort, from + step, to,
       ↪  new_step);
6      for (int i = from + step; i < to - step; i += new_step) {
7        compare_and_swap(vector_to_sort, i, i + step);
8      }
9    } else {
10     compare_and_swap(vector_to_sort, from, from + step);
11   }
12 }
```

Open in Compiler Explorer

which could be vectorized. Notice that many current compilers are able to vectorize the code for you, without the need to use intrinsics:

An example of vectorized subroutine of the odd-even merge sort.

```
1  #include <immintrin.h>
2
3  int main() {
4
5    std::vector<int> vec1 = std::vector<int>(SIZE);
```

```
6    std::vector<int> vec2 = std::vector<int>(SIZE);
7
8    for (int i = 0; i < SIZE; i++) {
9      vec1[i] = rand() % 10000;
10     vec2[i] = rand() % 10000;
11   }
12
13   auto t_start = std::chrono::high_resolution_clock::now();
14
15   __m256i v1;
16   __m256i v2;
17   __m256i r1, r2;
18
19   for (int i = 0; i < SIZE; i += 8) {
20     v1 = _mm256_loadu_si256((__m256i *)&vec1[i]);
21     v2 = _mm256_loadu_si256((__m256i *)&vec2[i]);
22     r1 = _mm256_min_epi32(v1, v2);
23     r2 = _mm256_max_epi32(v1, v2);
24     _mm256_storeu_si256((__m256i *)&vec1[i], r1);
25     _mm256_storeu_si256((__m256i *)&vec2[i], r2);
26   }
27
28   auto t_end = std::chrono::high_resolution_clock::now();
29   double elapsed = std::chrono::duration<double,
     ↪  std::milli>(t_end - t_start).count() / 1000.0;
30
31   std::cout << "compared in " << elapsed << " s" << std::endl;
32   return 0;
33 }
```

Open in Compiler Explorer

An example of vectorized subroutine of the odd-even merge sort.

```
1  int SIZE = 8;
2  std::vector<int> vec1 = std::vector<int>(SIZE);
3  for (int i = 0; i < SIZE; i++) {
4    vec1[i] = rand() % 10000;
5    std::cout << vec1[i] << " ";
6  }
7  __m128i mask_llhhllhh = _mm_set_epi32(0xffffffff, 0,
   ↪  0xffffffff, 0);
8  __m128i mask_hhllhhll = _mm_set_epi32(0, 0xffffffff, 0,
   ↪  0xffffffff);
9  __m128i v1;
10 __m128i v2;
11 __m128i r1, r2;
12 for (int i = 0; i < SIZE; i += 4) {
13   v1 = _mm_loadu_si128((__m128i *)&vec1[i]);
```

```
14    v2 = _mm_alignr_epi8(_mm_setzero_si128(), v1, 1 * 4);
15    r1 = _mm_min_epi32(v1, v2);
16    r1 = _mm_and_si128(r1, mask_hhllhhll);
17    v2 = _mm_alignr_epi8(v1, _mm_setzero_si128(), 3 * 4);
18    r2 = _mm_max_epi32(v1, v2);
19    r2 = _mm_and_si128(r2, mask_llhhllhh);
20    r1 = _mm_or_si128(r1, r2);
21    _mm_storeu_si128((__m128i *)&vec1[i], r1);
22  }
```

Open in Compiler Explorer

See https://xhad1234.github.io/Parallel-Sort-Merge-Join-in-Peloton/ for a state-of-the-art implementation.

7.2.3 Bubble sort with vectorisation

One may wish to "do better" by considering alternative algorithms and alternative synchronisation primitives. Quite possibly the simplest sorting algorithm is the bubble sort:

An example of bubble sort.

```
1  bool compare_swap(std::vector<int> &vector_to_sort, const int
   ↪  &val1, const int &val2) {
2    if (vector_to_sort[val1] > vector_to_sort[val2]) {
3      std::iter_swap(vector_to_sort.begin() + val1,
       ↪  vector_to_sort.begin() + val2);
4      return true;
5    }
6    return false;
7  }
8
9  void bubble(std::vector<int> &vector_to_sort, int from, int
   ↪  to) {
10   bool change = true;
11   while (change) {
12     change = false;
13     for (int i = from + 1; i < to; i++) {
14       change |= compare_swap(vector_to_sort, i - 1, i);
15     }
16   }
17 }
```

Open in Compiler Explorer

This can be parallelized with OpenMP:

An example of bubble sort in OpenMP.

```
1  void parallel_bubble(std::vector<int> &vector_to_sort,
   ↪  unsigned int from, unsigned int to) {
2    while (change) {
3      change = false;
4  #pragma omp parallel for num_threads(thread_count)
   ↪  schedule(static) shared(vector_to_sort) reduction(| \
5
   ↪  : change)
6      for (int i = from + 1; i < to; i += 2) {
7        change |= compare_swap(vector_to_sort, i - 1, i);
8      }
9
10 #pragma omp parallel for num_threads(thread_count)
   ↪  schedule(static) shared(vector_to_sort) reduction(| \
11
   ↪  : change)
12     for (int i = from + 2; i < to; i += 2) {
13       change |= compare_swap(vector_to_sort, i - 1, i);
14     }
15   }
16 }
```

Open in Compiler Explorer

7.2.4 Sample sort

Another alternative is to consider sample sort. This is implemented in Boost (https://www.boost.org/doc/libs/develop/libs/sort/doc/html/sort/parallel.html), and as with many ideas implemented in Boost, this is a great idea.

Bibliography

[1] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to parallel computation: P-completeness theory*. Oxford University Press on Demand, 1995.

[2] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. A. Badawy, "Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–8.

[3] M. Pöter and J. L. Träff, "Memory models for C/C++ programmers," *CoRR*, vol. abs/1803.04432, 2018. [Online]. Available: http://arxiv.org/abs/1803.04432

[4] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design*, vol. 41, no. 2, pp. 178–210, 2012. [Online]. Available: https://doi.org/10.1007/s10703-012-0161-5

[5] J. Yin, S. Gahlot, N. Laanait, K. Maheshwari, J. Morrison, S. Dash, and M. Shankar, "Strategies to deploy and scale deep learning on the summit supercomputer," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, Nov 2019, pp. 84–94.

[6] R. M. Gower and P. Richtárik, "Randomized iterative methods for linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 4, pp. 1660–1690, 2015.

[7] A. Akhriev, J. Marecek, and A. Simonetto, "Pursuit of low-rank models of time-varying matrices robust to sparse and measurement noise," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, arXiv preprint arXiv:1809.03550.

[8] J. Mareček, P. Richtárik, and M. Takáč, "Distributed block coordinate descent for minimizing partially separable functions," in *Numerical Analysis and Optimization: NAO-III, Muscat, Oman, January 2014*. Springer, 2015, pp. 261–288.

[9] S. Toth, *A Complete Guide to Standard C++ Algorithms*. https://github.com/HappyCerberus/book-cpp-algorithms, 2023.