Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček
jakub.marecek@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

What comes next?

What comes next?

C++

C++20:

- jthread

- Coroutines

- atomic_ref

- atomic<shared_ptr<T>>

- Floating point atomics

- Waiting on atomics

- Semaphores, latches, barriers

C++23:

- print, println ;-)

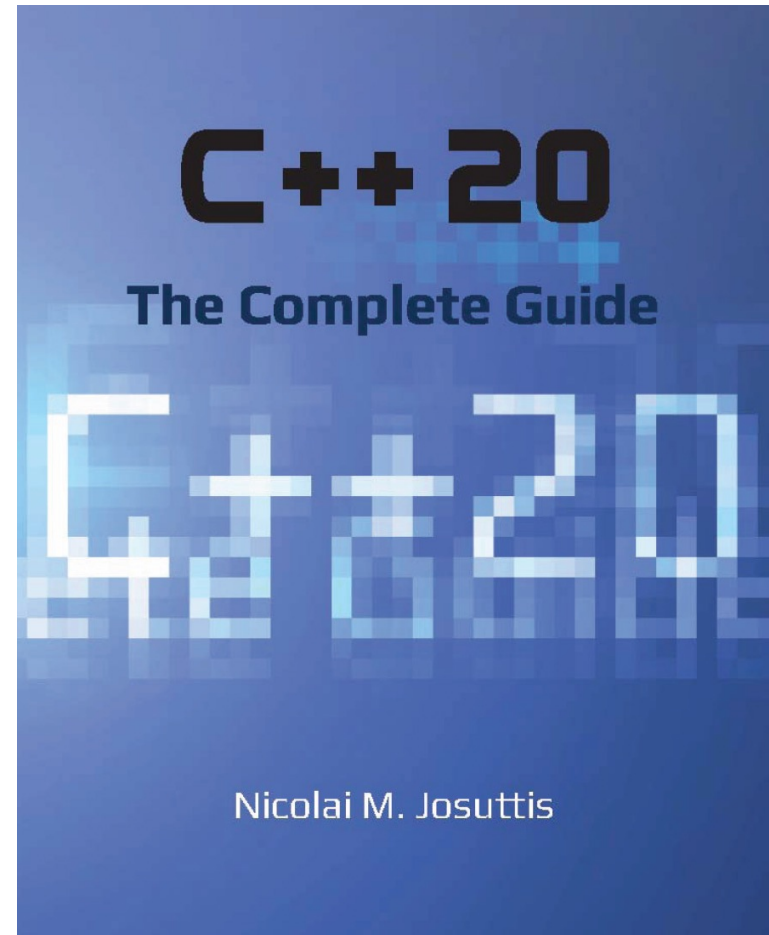- std::generator

C++26:

- Linear algebra!

What comes next?

- Nicolai M. Josuttis:
  C++20 - The Complete Guide

- http://cppstd20.com/

- Chapters:

  - 12: jthread

  - 13: Concurrency

  - 14: Coroutines

- See also
  https://en.cppreference.com/
  w/cpp/thread

What comes next?

https://github.com/jmarecek/parallel-cpp/blob/main/static/2024March6.pdf

What comes next?

- Structuring code
    - Thread
    - Jthread
    - Coroutines

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

- Structuring code
    - Coroutines

Structuring code
Threads

- 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:
- one needs to call join or detach prior to the destructor being called. If neither was called, the program was std::aborted.
- Prior to calling either, one needs to check whether the thread is joinable().
- At the same time, it is 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.

Structuring code
Threads

- The use of the C++11 thread is thus considered harmful and we will present only two short examples.

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

Structuring code
Threads

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

```cpp
1   #include <iostream>
2   #include <chrono>
3   #include <thread>
4
5   using namespace std::this_thread;
6   using namespace std::chrono_literals;
7
8   void A() {
9       std::cout << "a";
10      sleep_for(5s);
11      std::cout << "A";
12  }
13
14  void B() {
15      std::cout << "b";
16      sleep_for(1s);
17      std::cout << "B";
18  }
19

20  void C() {
21      std::cout << "c";
22      std::thread t(A);
23      t.detach();
24      std::thread u(B);
25      u.join();
26      std::cout << "C";
27  }
28
29  int main() {
30      C();
31      std::thread t(B);
32      t.join();
33      A();
34  }
```

Structuring code
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.

```cpp
1  #include <iostream>
2  #include <thread>
3  #include <vector>
4
5  void Hello();
6
7  int main(int argc, char* argv[]) {
8      std::vector<std::jthread> threads;
9      for (int cnt=0; cnt < 10; cnt++) {
10         threads.push_back(std::jthread(Hello));
11     }
12     return 0;
13 }
14
15 void Hello() {
16   using namespace std::chrono_literals;
17   std::this_thread::sleep_for(2s);
18 }
```

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

- Notice that the example would very like result in abnormal program termination, if we changed jthread to thread. (Why?)

Structuring code

- When we use standard output, it is prudent to wrap it in a syncstream (new addition in C++20):

```
1   #include <iostream>
2   #include <syncstream>
3   #include <thread>
4   #include <vector>
5
6   void Foobar(int cnt);
7
8   int main(int argc, char* argv[]) {
9       std::vector<std::jthread> threads;
10      for (int cnt=0; cnt < 10; cnt++) {
11          threads.push_back(std::jthread(Foobar, cnt));
12      }
13      std::osyncstream(std::cout) << "Main thread" << std::endl;
14      return 0;
15  }
16
17  void Foobar(int cnt) {
18      std::osyncstream(std::cout) << "Thread " << cnt << std::endl;
19  }
```

Structuring code

- Rather commonly, one uses the lambda function to define the thread.

```cpp
1  #include <iostream>
2  #include <syncstream>
3  #include <thread>
4  using namespace std::chrono_literals;
5
6  int main() {
7
8      auto t1 = std::jthread([](){
9          std::osyncstream(std::cout) << "Another thread" <<
             ↪  std::endl;
10         std::this_thread::sleep_for(1s);
11     });
12
13     std::this_thread::sleep_for(2s);
14     std::osyncstream(std::cout) << "Main thread" << std::endl;
15 }
```

(This is the lambda [capture](args).)

Structuring code

- 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:

```cpp
1  #include <iostream>
2  #include <syncstream>
3  #include <thread>
4  using namespace std::chrono_literals;
5
6  int main() {
7      auto t1 = std::jthread([](std::stop_token token){
8          while (!token.stop_requested()) {
9              std::osyncstream(std::cout) << "A thread";
10             std::this_thread::sleep_for(1s);
11         }
12         std::osyncstream(std::cout) << "Stop requested";
13     });
14
15     std::this_thread::sleep_for(2s);
16
17     std::osyncstream(std::cout) << "Main thread";
18     t1.request_stop();
19 }
```
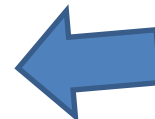
This is another example of ``resource acquisition is initialization" (RAII).

request_stop()

Structuring code

C++20 jthread

```cpp
1  #include <iostream>
2  #include <syncstream>
3  #include <atomic>
4  #include <thread>
5  using namespace std::chrono_literals;
6
7  int main() {
8
9      auto t = std::jthread([](std::stop_token token) {
10         std::osyncstream(std::cout) << "Thread " <<
           ↪  std::this_thread::get_id() << std::endl;
11         std::atomic<bool> flag = false;
12         std::stop_callback callback(token, [&flag]{
13             std::osyncstream(std::cout) << "Stop requested" <<
               ↪  std::endl;
14             flag = true;
15         });
16         while (!flag) {
17             std::this_thread::sleep_for(1s);
18         }
19     });
20
21     std::osyncstream(std::cout) << "Main thread" << std::endl;
22     std::this_thread::sleep_for(3s);
23     t.request_stop(); // runs all callbacks!
24 }
```

One can define std::stop_callback object inside the thread, whose constructor takes the stop token and a function.

(Notice the flag is "captured".)

The function gets executed, when the thread is requested to stop via the std::stop_token.

Structuring code
C++20 jthread

```cpp
 1  #include <iostream>
 2  #include <syncstream>
 3  #include <atomic>
 4  #include <thread>
 5  using namespace std::chrono_literals;
 6
 7  int main() {
 8
 9      auto t = std::jthread([](std::stop_token token) {
10          std::osyncstream(std::cout) << "Thread " <<
                ↪  std::this_thread::get_id() << std::endl;
11          std::atomic<bool> flag = false;
12          std::stop_callback callback(token, [&flag]{
13              std::osyncstream(std::cout) << "Stop requested" <<
                    ↪  std::endl;
14              flag = true;
15          });
16          while (!flag) {
17              std::this_thread::sleep_for(1s);
18          }
19      });
20
21      std::osyncstream(std::cout) << "Main thread" << std::endl;
22      std::this_thread::sleep_for(3s);
23      t.request_stop(); // runs all callbacks!
24  }
```

One can define std::stop_callback object inside the thread, whose constructor takes the stop token and a function.
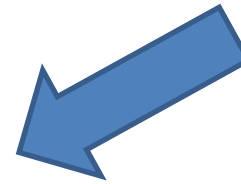
(Notice the flag is "captured".)

The function gets executed, when the thread is requested to stop via the std::stop_token.

Structuring code
Coroutines

- 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.

https://blog.eiler.eu/posts/20210512/images/coroutines.png
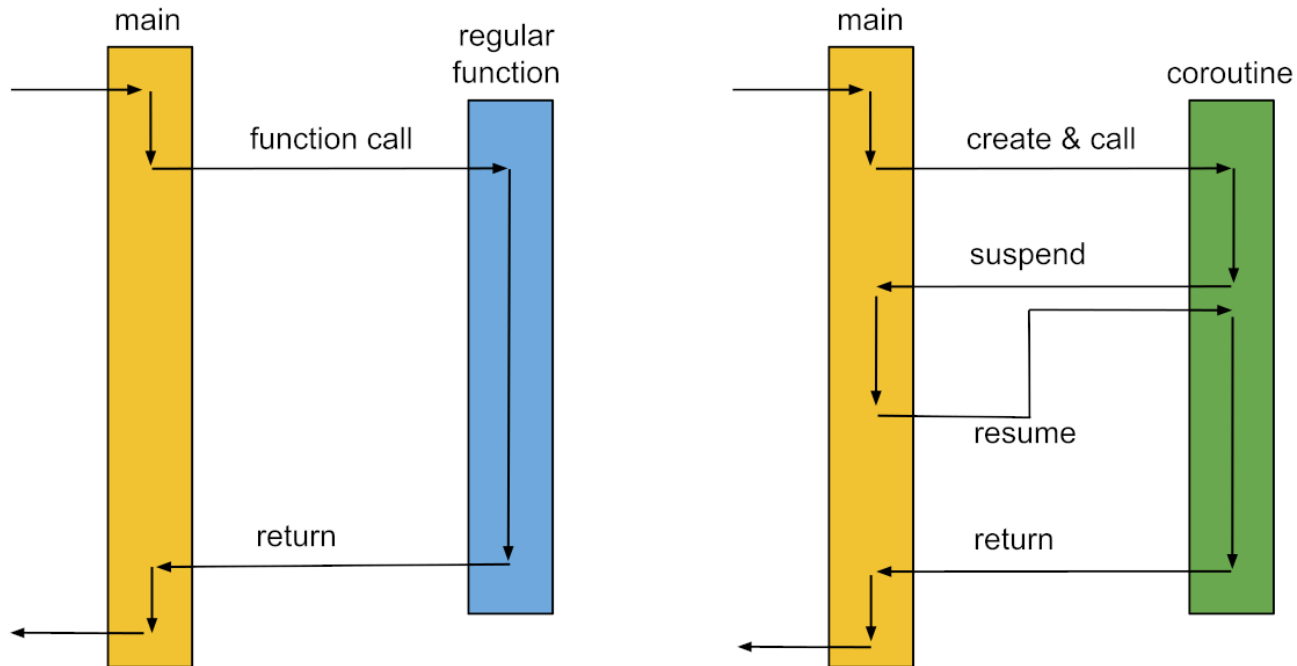
Structuring code
Coroutines

In C++23, 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. Needs -std=c++2b or similar.

```cpp
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 }
```
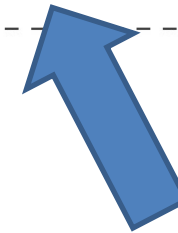
Needs
-std=gnu++2b
or
-std=c++2b

https://godbolt.org/z/
Kn8s48aqT

Structuring code
Coroutines

```cpp
 7  struct promise;
 8
 9  struct coroutine : std::coroutine_handle<promise> {
10      using promise_type = struct promise;
11  };
12
13  struct promise {
14      coroutine get_return_object() { return
        ↪  {coroutine::from_promise(*this)}; }
15      std::suspend_always initial_suspend() noexcept { return
        ↪  {}; }
16      std::suspend_always final_suspend() noexcept { return {};
        ↪  }
17      void return_void() {}
18      void unhandled_exception() {}
19  };
20
21  int main() {
22      coroutine h = [](int i) -> coroutine {
23          std::osyncstream(std::cout) << i;
24          co_return;
25      }(0);
26      h.resume();
27      h.destroy();
28  }
```
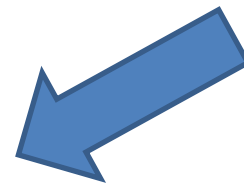
Later, we will see how to implement a generator ourselves.

What comes next?

- Structuring code
    - Thread
    - Jthread
    - Coroutines

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

- Structuring code
    - Coroutines

Synchronisation Primitives

Atomic Variables

Since C++11, there is an excellent support for atomic variables in header <atomic>. The primary template can be instantiated with types that are TriviallyCopyable, CopyConstructible, and CopyAssignable.

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

Synchronisation Primitives
Atomic Variables

Since C++11, there is an excellent support for atomic variables in header <atomic>. The primary template can be instantiated with types that are TriviallyCopyable, CopyConstructible, and CopyAssignable.

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

```cpp
 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> class stack {
13      std::atomic<Node<T>*> head;
14      public:
15      void push(const T& data) {
16          Node<T>* new_node = new Node<T>(data);
17          new_node->next = head.load(std::memory_order_relaxed);
18          while(!head.compare_exchange_weak(new_node->next,
              ↪   new_node, std::memory_order_release,
              ↪   std::memory_order_relaxed));
19      }
20  };
21
22  int main() {
23      std::stack<int> s; s.push(1); s.push(2); s.push(3);
24  }
```

Synchronisation Primitives

Barrier

- Since C++20, there is support for barriers in header <barrier>.
- The constructor takes an integer value, which is the number of threads that the barrier is expected to block.
- **arrive_and_wait():** blocking wait until the number of threads arrive at the same spot
- **arrive_and_drop():** 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.

```cpp
 1  #include <barrier>
 2  #include <syncstream>
 3  #include <iostream>
 4  #include <vector>
 5  #include <thread>
 6  #include <algorithm>
 7  #include <random>
 8
 9  int main() {
10      std::barrier b(5);
11      std::vector<std::jthread> ts;
12      std::generate_n(std::back_inserter(ts), 5, [&b]{
13          return std::jthread([&b]{
14              std::mt19937
                  ↪  gen(std::hash<std::thread::id>{}(std::this_thread::get_id()));
15              std::bernoulli_distribution d(0.3);
16              int cnt = 1;
17              while (true) {
18                  std::osyncstream(std::cout) <<
                      ↪  std::this_thread::get_id() << "/" << cnt
                      ↪  << std::endl;
19                  std::this_thread::yield();
20                  if (d(gen)) {
21                      b.arrive_and_drop();
22                      return;
23                  } else {
24                      b.arrive_and_wait();
25                  }
26                  cnt++;
27              }
28          });
29      });
30  }
```

Synchronisation Primitives

Barrier

More complicated uses of barriers may use the template parameter CompletionFunction and have a callable executed whenever the barrier hits 0:

```
11      std::barrier b(4,[id = 1]() mutable noexcept {
12          std::osyncstream(std::cout) << id << " OK" <<
            ↪   std::endl;
13          id++;
14      });
15      std::vector<std::jthread> runners;
16
17      std::generate_n(std::back_inserter(runners), 4, [&b]{
18          return std::jthread([&b]{
19              std::osyncstream(std::cout) <<
                ↪   std::this_thread::get_id() << "/1" <<
                ↪   std::endl;
20              std::this_thread::yield();
21              b.arrive_and_wait();
22              std::osyncstream(std::cout) <<
                ↪   std::this_thread::get_id() << "/2" <<
                ↪   std::endl;
23              std::this_thread::yield();
24              b.arrive_and_wait();
25          });
26      });
27      runners.clear();
28      std::osyncstream(std::cout) << std::endl;
29  }
```

Synchronisation Primitives

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.

Synchronisation Primitives

Mutexes and Locks
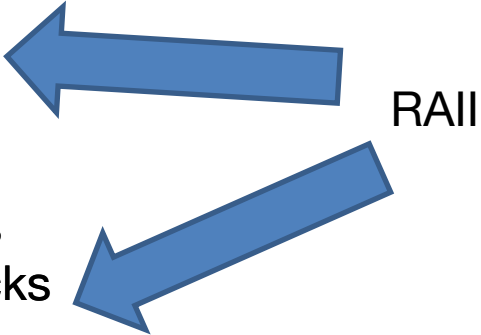
```cpp
 6  int main() {
 7      using namespace std::chrono_literals;
 8      struct Shared {
 9          int value;
10          std::mutex mux;
11      };
12      Shared shared{0,{}};
13      auto t1 = std::jthread([&shared]{
14          std::this_thread::sleep_for(1s);
15          for (int i = 0; i < 10; i++) {
16              std::this_thread::yield();
17              {
18                  std::unique_lock lock(shared.mux);
19                  shared.value += 10;
20              } // mutex unlocks!
21              std::this_thread::sleep_for(1s);
22          }
23      });
24  }
```

Synchronisation Primitives

Mutexes and Locks

```cpp
 6  int main() {
 7      using namespace std::chrono_literals;
 8      struct Shared {
 9          int value;
10          std::mutex mux;
11      };
12      Shared shared{0,{}};
13      auto t = std::jthread([&shared]{
14          std::this_thread::sleep_for(1s);
15          for (int i = 0; i < 10; i++) {
16              {
17                  std::unique_lock lock(shared.mux);
18                  shared.value += 1;
19              }
20              std::this_thread::sleep_for(1s);
21          }
22      });
23      auto observer = std::jthread([&shared]{
24          while (true) {
25              {
26              std::unique_lock lock(shared.mux);
27              std::cout << shared.value << std::endl;
28              if (shared.value == 10)
29                  break;
30              }
31              std::this_thread::sleep_for(1s);
32          }
33      });
34  }
```

What comes next?

- Structuring code
    - Thread
    - Jthread
    - Coroutines   RAII

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

- Structuring code
    - Coroutines

Algorithms in the Standard Template Library

For each

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

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

```cpp
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     });
```

- Similarly useful is the reduce operation (also known as fold, accumulate, aggregate, compress, or inject).
- In Map Reduce, one applies an associative operation to each piece of data to obtain a partial result, and then 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.

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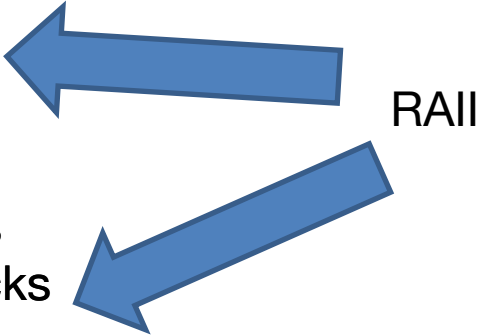
Algorithms in the Standard Template Library

Merge

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

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

What comes next?

- Structuring code
  - Thread
  - Jthread
  - Coroutines   RAII

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

- Structuring code
  - Coroutines

Structuring code
Coroutines

- 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 and factories and various
  other concepts within ``lazy evaluation'', as well as event-
  driven architectures within cooperative multi-tasking.

- 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.

Structuring code
Coroutines

There are three new keywords:

- **co_await** tries to suspends computation and block the co-routine until the computation is resumed by another coroutine calling ``resume'' method of the present coroutine. In the process, it tests whether it is possible to suspend the computation using an awaiter 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.)

A 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:

```
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  }
```

Structuring code
Coroutines

Defining the coroutine in C++20 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.

Structuring code
Coroutines

A **promise** class implements:
- coroutine **get_return_object()** is called to initialize the coroutine and create the coroutine handle
- std::suspend_always **initial_suspend()**, suggests whether the coroutine starts right after initialization
  std::suspend_always **final_suspend()** noexcept which can be rather formulaic std::suspend_always()
- void **return_void()** or void **return_value**(cons 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().

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

Structuring code
Coroutines

```cpp
 7  struct promise;
 8
 9  struct coroutine : std::coroutine_handle<promise> {
10      using promise_type = struct promise;
11  };
12
13  struct promise {
14      coroutine get_return_object() { return
        ↪  {coroutine::from_promise(*this)}; }
15      std::suspend_always initial_suspend() noexcept { return
        ↪  {}; }
16      std::suspend_always final_suspend() noexcept { return {};
        ↪  }
17      void return_void() {}
18      void unhandled_exception() {}
19  };
20
21  int main() {
22      coroutine h = [](int i) -> coroutine {
23          std::osyncstream(std::cout) << i;
24          co_return;
25      }(0);
26      h.resume();
27      h.destroy();
28  }
```

Structuring code

Finally, let us consider awaiters, which can be called when a coroutine is suspended or resumed.

Key methods of an awaiter include:

- **await_ready()** is called immediately before suspension of a coroutine. If it returns true, the coroutine will not be suspended.
- **await_suspend**(handle) 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 awaiters we have seen so far (std::suspend_never() and std::suspend_always()) returned boolean constants in await_ready()

Structuring code

Awaiters

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.

```
5    struct suspend_always
6    {
7      constexpr bool await_ready() const noexcept { return
         ↪  false; }
8      constexpr void await_suspend(coroutine_handle<>) const
         ↪  noexcept {}
9      constexpr void await_resume() const noexcept {}
10   };
11
12   struct suspend_never
13   {
14     constexpr bool await_ready() const noexcept { return true;
         ↪  }
15     constexpr void await_suspend(coroutine_handle<>) const
         ↪  noexcept {}
16     constexpr void await_resume() const noexcept {}
17   };
```

Structuring code

Our Own Generator

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

```cpp
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 }
```

```cpp
48  Generator myCoroutine() {
49      int x = 0;
50      while (true) {
51          co_yield x++;
52      }
53  }
```

Structuring code
Our Own Generator

```cpp
 7  // The caller-level type
 8  struct Generator {
 9      // The coroutine level type
10      struct promise_type {
11          using Handle = std::coroutine_handle<promise_type>;
12
13          Generator get_return_object() {
14              return Generator{Handle::from_promise(*this)};
15          }
16          std::suspend_always initial_suspend() { return {}; }
17          std::suspend_always final_suspend() noexcept { return
    ↪    {}; }
18          std::suspend_always yield_value(int value) {
19              current_value = value;
20              return {};
21          }
22          void unhandled_exception() { }
23          int current_value;
24      };
```

```
26      explicit Generator(promise_type::Handle coro) :
        ↪   coro_(coro) {}
27      // Make move-only
28      Generator(const Generator&) = delete;
29      Generator& operator=(const Generator&) = delete;
30      Generator(Generator&& t) noexcept : coro_(t.coro_) {
        ↪   t.coro_ = {}; }
31      Generator& operator=(Generator&& t) noexcept {
32          if (this == &t) return *this;
33          if (coro_) coro_.destroy();
34          coro_ = t.coro_;
35          t.coro_ = {};
36          return *this;
37      }
38
39      int get_next() {
40          coro_.resume();
41          return coro_.promise().current_value;
42      }
```

Structuring code
Our Own Message-Passing

```cpp
 4  class Event {
 5   public:
 6
 7      Event() = default;
 8
 9      Event(const Event&) = delete;
10      Event(Event&&) = delete;
11      Event& operator=(const Event&) = delete;
12      Event& operator=(Event&&) = delete;
13
14      class Awaiter;
15      Awaiter operator co_await() const noexcept;
16
17      void notify() noexcept;
18
19   private:
20      friend class Awaiter;
21      mutable std::atomic<void*> suspendedWaiter{nullptr};
22      mutable std::atomic<bool> notified{false};
23
24  };
```

Structuring code
Our Own Message-Passing

```cpp
26  class Event::Awaiter {
27   public:
28     Awaiter(const Event& eve): event(eve) {}
29
30     bool await_ready() const;
31     bool await_suspend(std::coroutine_handle<> corHandle)
      ↪    noexcept;
32     void await_resume() noexcept {}
33
34   private:
35       friend class Event;
36
37       const Event& event;
38       std::coroutine_handle<> coroutineHandle;
39  };
```

Structuring code
Our Own Message-Passing

```cpp
41  struct Task {
42      struct promise_type {
43          Task get_return_object() { return {}; }
44          std::suspend_never initial_suspend() { return {}; }
45          std::suspend_never final_suspend() noexcept { return
            ↪  {}; }
46          void return_void() {}
47          void unhandled_exception() {}
48      };
49  };
```

What have we seen so far?

- Structuring code
    - Thread
    - Jthread
    - Coroutines

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

- Structuring code
    - Coroutines