Concurrency and Parallelism with C++17 and C++20/23

Rainer Grimm
Training, Coaching and Technology Consulting

www.ModernesCpp.de
## Concurrency and Parallelism in C++

<table>
<thead>
<tr>
<th>Year</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Memory model, Threads, Mutexes and locks, Thread local data, Condition variables, Tasks</td>
</tr>
<tr>
<td>2014</td>
<td>Reader-writer locks</td>
</tr>
<tr>
<td>2017</td>
<td>Parallel STL</td>
</tr>
<tr>
<td>2020/2023</td>
<td>Executors, std::jthread, Atomic smart pointers, std::future extensions, Latches and barriers, Coroutines, Transactional memory, Task blocks</td>
</tr>
</tbody>
</table>

### Timeline

- **C++11**: 2011
- **C++14**: 2014
- **C++17**: 2017
- **C++20/2023**:
Concurrency and Parallelism in C++17

- Memory model
- Threads
- Mutexes and locks
- Thread local data
- Condition variables
- Tasks

C++11

2011

C++14

2014

C++17

2017

C++20/23

2020/2023

- Reader-writer locks
- Parallel STL

- Executors
  - std::jthread
  - Atomic smart pointers
  - std::future extensions
  - Latches and barriers
  - Coroutines
  - Transactional memory
  - Task blocks
Parallel STL

You can choose the execution policy of an algorithm.

- **Execution policies**
  - `std::execution::seq`
    - Sequential in one thread
  - `std::execution::par`
    - Parallel
  - `std::execution::par_unseq`
    - Parallel and vectorised → SIMD
Parallel STL

const int SIZE = 8;
int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
int res[SIZE] = {0,};

int main()
{
    for (int i = 0; i < SIZE; ++i)
    {
        res[i] = vec[i] + 5;
    }
}

Not vectorised

Vectorised

```assembly
movslq -8(%rbp), %rax
movl vec(%rip), %rax
addl $5, %ecx
movslq -8(%rbp), %rax
movl %ecx, res(%rip)
```

```assembly
movdqa .LCIP0_0(%rip), %xmm0  # xmm0 = [5,5,5,5]
movdqa vec(%rip), %xmm1
paddd %xmm0, %xmm1
movdqa %xmm1, res(%rip)
paddd vec+16(%rip), %xmm0
movdqa %xmm0, res+16(%rip)
xorl %eax, %eax
```
using namespace std;
vector<int> vec = {1, 2, 3, 4, 5, .... }
Parallel STL

adjacent_difference, adjacent_find, all_of any_of, copy, copy_if, copy_n, count, count_if, equal, exclusive_scan, fill, fill_n, find, find_end, find_first_of, find_if, find_if_not, for_each, for_each_n, generate, generate_n, includes, inclusive_scan, inner_product, inplace_merge, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, merge, min_element, minmax_element, mismatch, move, none_of, nth_element, partial_sort, partial_sort_copy, partition, partition_copy, reduce, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, search, search_n, set_difference, set_intersection, set_symmetric_difference, set_union, sort, stable_partition, stable_sort, swap_ranges, transform, transform_exclusive_scan, transform_inclusive_scan, transform_reduce, uninitialized_copy, uninitialized_copy_n, uninitialized_fill, uninitialized_fill_n, unique, unique_copy
Parallel STL

std::transform_reduce

- Haskeil's function `map` is called `std::transform` in C++
- std::transform_reduce ➔ std::map_reduce

```cpp
std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};

std::size_t res = std::transform_reduce(std::execution::par,
    strVec.begin(), strVec.end(), 0,
    [](std::size_t a, std::size_t b){ return a + b; },
    [](std::string s){ return s.length(); });

std::cout << res;   // 21
```
Parallel STL

- Danger of data races or deadlocks

```cpp
int numComp = 0;
std::vector<int> vec = {1, 3, 8, 9, 10};
std::sort(std::execution::par, vec.begin(), vec.end(),
          [&numComp](int fir, int sec){
              numComp++;
              return fir < sec;
          });
```

The access to `numComp` has to be atomic.
Parallel STL

- **Support for** `std::execution::par`
  - GCC and Clang

- **MSVC with Visual Studio 2017 15.8** (`/std=c++latest`)
  - `copy, copy_n, fill, fill_n, move, reverse, reverse_copy, rotate, rotate_copy, swap_ranges`
  - `adjacent_difference, adjacent_find, all_of, any_of, count, count_if, equal, exclusive_scan, find, find_end, find_first_of, find_if, for_each, for_each_n, inclusive_scan, mismatch, none_of, partition, reduce, remove, remove_if, search, search_n, sort, stable_sort, transform, transform-exclusive_scan, transform-inclusive_scan, transform_reduce`
Concurrency and Parallelism in C++20/23

- **C++11** (2011):
  - Memory model
  - Threads
  - Mutexes and locks
  - Thread local data
  - Condition variables
  - Tasks

- **C++14** (2014):
  - Reader-writer locks

- **C++17** (2017):
  - Parallel STL

- **C++20/23**:
  - Executors
  - `std::jthread`
  - Atomic smart pointers
  - `std::future` extensions
  - Latches and barriers
  - Coroutines
  - Transactional memory
  - Task blocks
Executors

Executors are the basic building block for execution in C++. They fulfil a similar role for execution such as allocators for allocation.

An executor consists of a set of rules for a callables:
- **Where**: run on a internal or external processor
- **When**: run immediately or later
- **How**: run on a CPU or GPU
Executors

- **Using an executor**
  ```cpp
  my_executor_type my_executor = ...;
  auto future = std::async(my_executor, []{
    std::cout << "Hello world " << std::endl;
  });
  
  std::for_each(std::execution::par.on(my_executor),
    data.begin(), data.end(), func);
  
  ```

- **Obtaining an executor**
  ```cpp
  static_thread_pool pool(4);
  auto exec = pool.executor();
  task1 = long_running_task(exec);
  ```
Executors

An executor provides one or more execution functions for creating a callable.

<table>
<thead>
<tr>
<th>Name</th>
<th>Cardinality</th>
<th>Direction</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute</td>
<td>single</td>
<td>oneway</td>
<td>C++20</td>
</tr>
<tr>
<td>twoway_execute</td>
<td>single</td>
<td>twoway</td>
<td>C++23</td>
</tr>
<tr>
<td>then_execute</td>
<td>single</td>
<td>then</td>
<td>C++23</td>
</tr>
<tr>
<td>bulk_execute</td>
<td>bulk</td>
<td>oneway</td>
<td>C++20</td>
</tr>
<tr>
<td>bulk_twoway_execute</td>
<td>bulk</td>
<td>twoway</td>
<td>C++23</td>
</tr>
<tr>
<td>bulk_then_execute</td>
<td>bulk</td>
<td>then</td>
<td>C++23</td>
</tr>
</tbody>
</table>

**Cardinality**: Creation of one execution agent or a group of execution agents.

**Direction**: Directions of the execution.
Problem: `std::thread` throws `std::terminate` in its destructor if still joinable.

```cpp
std::thread t{[]{ std::cout << "New thread"; }};
std::cout << "t.joinable(): " << t.joinable();
```

```
rainer@seminar:~> thread
t.joinable(): true
terminate called without an active exception
Aborted (core dumped)
rainer@seminar:~>  
```
std::jthread

Solution: std::jthread joins automatically at the end of its scope.

std::jthread t{[] { std::cout << "New thread"; } }; 
std::cout << "t.joinable(): " << t.joinable();
std::jthread

- Instances of `std::jthread` can be interrupted

**Receiver** *(stop_token)*
- Explicit check:
  - `stop_requested`: yields, when an interrupt was signalled
  - `std::condition_variable_any wait` variations with predicate

**Sender** *(stop_source)*
- `request_stop`: signals an interrupt
```cpp
std::jthread

jthread nonInterruptable([]{
    int counter{0};
    while (counter < 10){
        this_thread::sleep_for(0.2s);
        cerr << "nonInterruptable: "
             << counter << endl;
        ++counter;
    }
});

jthread interruptable([](stop_token stoken){
    int counter{0};
    while (counter < 10){
        this_thread::sleep_for(0.2s);
        if (stoken.stop_requested()) return;
        cerr << "interruptable: "
             << counter << endl;
        ++counter;
    }
});

this_thread::sleep_for(1s);
cerr << endl;
cerr << "Main thread interrupts both jthreads" << endl;
nonInterruptable.request_stop();
interruptable.request_stop();
cout << endl;
```
std::jthread

interruptJthread

interruptable: 0
nonInterruptable: 0
interruptable: 1
nonInterruptable: 1
interruptable: 2
nonInterruptable: 2
interruptable: 3
nonInterruptable: 3

Main thread interrupts both jthreads

nonInterruptable: 4
nonInterruptable: 5
nonInterruptable: 6
nonInterruptable: 7
nonInterruptable: 8
nonInterruptable: 9
rainer@seminar:~>
Atomic Smart Pointers

C++11 has a `std::shared_ptr` for shared ownership.

- **General Rule:**
  - You should use smart pointers.

- **But:**
  - The managing of the control block and the deletion of the resource is thread-safe. The access to the resource is not thread-safe.


- **Solution:**
  - `std::atomic_shared_ptr`
  - `std::atomic_weak_ptr`
Atomic Smart Pointer

3 Reasons

- **Consistency**
  - `std::shared_ptr` is the only non-atomic data type for which atomic operations exists.

- **Correctness**
  - The correct usage of atomic operations is just based on the discipline of the user. **extremely error-prone**
    - `std::atomic_store(&sharPtr, localPtr) ≠ sharPtr = localPtr`

- **Performance**
  - `std::shared_ptr` has to be design for the special use-case.
```cpp
// Atomic Smart Pointer

template<typename T> class concurrent_stack {
  struct Node { T t; shared_ptr<Node> next; };
  atomic_shared_ptr<Node> head;
  // in C++11: remove "atomic_" and remember to use the special
  // functions every time you touch the variable
  concurrent_stack( concurrent_stack & ) = delete;
  void operator=(concurrent_stack&) = delete;

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

auto find( T t ) const {
  auto p = head.load(); // in C++11: atomic_load(&head)
  while( p && p->t != t )
    p = p->next;
  return reference(move(p));
}
auto front() const {
  return reference(head); // in C++11: atomic_load(&head)
}
void push_front( T t ) {
  auto p = make_shared<Node>();
  p->t = t;
  p->next = head; // in C++11: atomic_load(&head)
  while( !head.compare_exchange_weak(p->next, p) ){ }
  // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
}
void pop_front() {
  auto p = head.load();
  while( p && !head.compare_exchange_weak(p, p->next) ){ }
  // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
}
};
```
Atomic Smart Pointers

Atomic smart pointers are part of the ISO C++ standard.
- Partial specialisation of `std::atomic`

- `std::atomic_shared_ptr`  
  `std::atomic<std::shared_ptr<T>>`

- `std::atomic_weak_ptr`  
  `std::atomic<std::weak_ptr<T>>`
std::future Extensions

std::future doesn't support composition

- std::future Improvement ➔ Continuation
  - then: execute the next future if the previous one is done

```cpp
future<int> f1 = async([](){ return 123; });
future<string> f2 = f1.then([](future<int> f){
    return to_string(f.get()); // non-blocking
});
auto myResult = f2.get();          // blocking
```
std::future Extensions

- **when_all**: execute the future if all futures are done

```cpp
future<int> futures[] = { async([]() { return intResult(125); } ),
                        async([]() { return intResult(456); } )};
future<vector<future<int>>> all_f = when_all(begin(futures), end(futures));

vector<future<int>> myResult = all_f.get();
for (auto fut: myResult): fut.get();
```

- **when_any**: execute the future if one of the futures is done

```cpp
future<int> futures[] = {async([]() { return intResult(125); } ),
                        async([]() { return intResult(456); } )};
when_any_result<vector<future<int>>> any_f = when_any(begin(futures),
                                                       end(futures));

future<int> myResult = any_f.futures[any_f.index];
auto myResult = myResult.get();
```
make_ready_future and make_exception_future: create a future directly

```cpp
future<int> compute(int x){
    if (x < 0) return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);
    future<int> f1 = async([]{ return do_work(x); });
    return f1;
}
```

Further information

C++17: I See a Monad in Your Future! (Bartosz Milewski)
std::future Extensions

- Disadvantages of the extended futures
  - Futures and promises are coupled to std::thread.
  - Where is the .then continuation be invoked?
  - Passing futures to .then continuation is too verbose.
    
    ```cpp
    std::future f1 = std::async([]{ return 123; });
    std::future f2 = f1.then([](std::future f){
        return std::to_string(f.get()); }
    );
    std::future f2 = f1.then(std::to_string);
    ```

  - Future blocks in its destructor.
  - Futures und values should be easily composable.
    ```cpp
    bool f(std::string, double, int);
    std::future<std::string> a = /* ... */;
    std::future<int> c = /* ... */;
    f(a.get(), 3.14, c.get())
    std::future<bool> d2 = when_all(a, 3.14, c).then(f);
    ```

C++ has no semaphores ➔ latches and barriers

- **Key idea**
  A thread is waiting at the synchronisation point until the counter becomes zero.

  - **latch** is for the one-time use-case
    - `count_down_and_wait`: decrements the counter until it becomes zero
    - `count_down(n = 1)`: decrements the counter by n
    - `is_ready`: checks the counter
    - `wait`: waits until the counter becomes zero
Latches and Barriers

- **barrier** can be reused
  - **arrive_and_wait**: waits at the synchronisation point
  - **arrive_and_drop**: removes itself from the synchronisation mechanism

- **flex_barrier** is a reusable and adaptable barrier
  - The constructor gets a callable.
  - The callable will be called in the completion phase.
  - The callable returns a number which stands for the counter in the next iteration.
  - Can change the value of the counter for each iteration.
void doWork(threadpool* pool) {
    latch completion_latch(NUMBER_TASKS);
    for (int i = 0; i < NUMBER_TASKS; ++i) {
        pool->add_task([&] {
            // perform the work
            ...
            completion_latch.count_down();
        }));
    }
    // block until all tasks are done
    completion_latch.wait();
}
Coroutines

Coroutines are generalised functions that can be suspended and resumed while keeping their state.

- **Typical use-case**
  - Cooperative Tasks (protection from data races)
  - Event loops
  - Infinite data streams
  - Pipelines
Coroutines

Design Principles

- **Scalable**, to billions of concurrent coroutines
- **Efficient**: Suspend/resume operations comparable in cost to function call overhead
- **Open-Ended**: Library designers can develop coroutine libraries
- **Seamless Interaction** with existing facilities with no overhead
- **Usable** in environments where exceptions are forbidden or not available.
A function is a coroutine if it has a `co_return`, `coAwait`, `co_yield` call or if it has a range-based for loop with a `coAwait` call.

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Coroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>invoke</code></td>
<td><code>func(args)</code></td>
</tr>
<tr>
<td><code>return</code></td>
<td><code>return statement</code></td>
</tr>
<tr>
<td><code>suspend</code></td>
<td><code>coAwait someAwaitable</code></td>
</tr>
<tr>
<td></td>
<td><code>co_yield someValue</code></td>
</tr>
<tr>
<td><code>resume</code></td>
<td><code>coroutine_handle&lt;&gt;::resume()</code></td>
</tr>
</tbody>
</table>
Coroutines

generator<int> genForNumbers(int begin, int inc= 1){
    for (int i = begin;; i += inc){
        co_yield i;
    }
}

int main(){
    auto numbers = genForNumbers(-10);
    for (int i = 1; i <= 20; ++i) std::cout << numbers << " ";
    for (auto n: genForNumbers(0, 5)) std::cout << n << " ";
}

-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 . . . .
Coroutines

Blocking

Acceptor accept{443};

while (true){
    Socket so = accept.accept(); // block
    auto req = so.read();          // block
    auto resp = handleRequest(req);
    so.write(resp);                // block
}

Waiting

Acceptor accept{443};

while (true){
    Socket so = co_wait accept.accept();
    auto req = co_wait so.read();
    auto resp = handleRequest(req);
    co_wait so.write(resp);
}
Transactional Memory

*Transactional Memory* is the idea of transactions from the database theory applied to software.

- A transaction has the ACID properties without *Durability*
  ```
  atomic{
    statement1;
    statement2;
    statement3;
  }
  ```

- **Atomicity**: all or no statement will be performed
- **Consistency**: the system is always in a consistent state
- **Isolation**: a transaction runs total isolation
- **Durability**: the result of a transaction will be stored
Transactional Memory

- **Transactions**
  - build a total order
  - feel like a global lock
    - Optimistic approach ≠ lock

- **Workflow**

  **Retry**
  A transaction stores its initial state.
  The transaction will be performed without synchronisation.
  The runtime experiences a violation to the initial state.
  The transaction will be performed once more.

  **Rollback**
Transactional Memory

- **Two forms**
  - *synchronized blocks*
    - *relaxed transaction*
    - are not transaction in the pure sense
      - can have *transaction-unsafe code*
  - *atomic blocks*
    - atomic blocks
    - are available in three variations
      - can only execute *transaction-safe code*
int i = 0;

void inc() {
  synchronized{
    cout << ++i << " ,";
  }
}

vector<thread> vecSyn(10);
for(auto& t : vecSyn)
  t = thread([]{ for(int n = 0; n < 10; ++n) inc(); });
void inc() {
    synchronized{
        std::cout << ++i << " ,";
        this_thread::sleep_for(1ns);
    }
}

vector<thread> vecSyn(10), vecUnsyn(10);
for(auto& t: vecSyn)
    t= thread[]{ for(int n = 0; n < 10; ++n) inc(); }
for(auto& t: vecUnsyn)
    t= thread[]{ for(int n = 0; n < 10; ++n) cout << ++i << " ,"; };

Transaction Memory
Transactional Memory

```cpp
int i = 0;
void func() {
    atomic_noexcept{
        cout << ++i << " ,"; // non transaction-safe code
    }
}
```

A transaction can only perform **transaction-safe code**

*compiler error*
Task Blocks

Fork-join parallelism with task blocks.
template <typename Func>
int traverse(node& n, Func && f){
    int left = 0, right = 0;
    define_task_block(
        [&](task_block& tb) {
            if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
            if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
        }
    );
    return f(n) + left + right;
}
Task Blocks

\begin{itemize}
\item \texttt{define_task_block_restore_thread}
\item \texttt{define_task_block}
\item \texttt{wait}
\end{itemize}

\begin{verbatim}
(1) define_task_block([&](auto& tb)
    tb.run([&]{ func(); });
(2) define_task_block_restore_thread([&](auto& tb){
    tb.run([&]{ func2(); });
(3) define_task_block([&](auto& tb){
    tb.run([&]{ func3(); }
(3) });
... ...
(2) });
... ...
(1) });
\end{verbatim}

\begin{verbatim}
define_task_block([&](auto& tb){
    tb.run([&]{ process(x1, x2); });
    if (x2 == x3) tb.wait();
    process(x3, x4);
});
\end{verbatim}
Task Blocks

- The scheduler

```cpp
tb.run( [&] { process(x1, x2); } );
```

Parent  Child

- **Child stealing**: the scheduler steals the job and executes it
- **Parent stealing**: the task block performs the child; the scheduler steals the parent

Both strategies are possible in C++20
Concurrency and Parallelism in C++

**Multithreading**

- Memory model
- Threads
- Mutexes and locks
- Thread local data
- Condition variables
- Tasks

- Reader-writer locks

- Parallel STL

- Executors
  - `std::jthread`
  - Atomic smart pointers
  - `std::future` extensions
  - Latches and barriers
  - Coroutines
  - Transactional memory
  - Task blocks

C++11: 2011
C++14: 2014
C++17: 2017
C++20/23: 2020/2023
Concurrency and Parallelism in C++
Proposals

- Stop tokens and a joining thread [P0660R8](#) (2019)
- Atomic smart pointers: [N4162](#) (2014)
- `std::future` extensions: [N4107](#) (2014) and [P070r1](#) (2017)
- Latches and barriers: [P0666R0](#) (2017)
- Coroutines: [N4723](#) (2018)
- Task blocks: [N4411](#) (2015)
Blogs

www.grimm-jaud.de [De]
www.ModernesCpp.com [En]