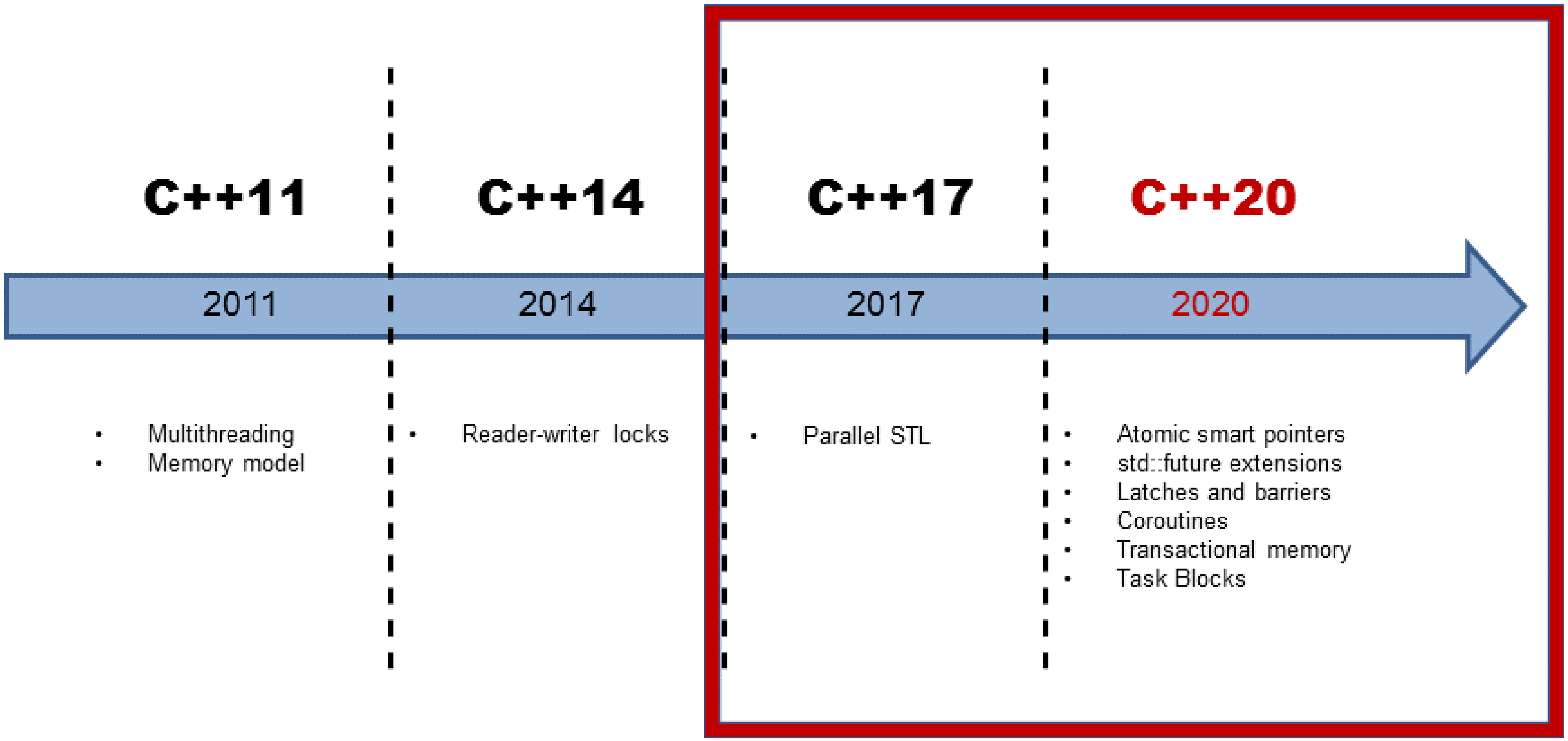
Concurrency and Parallelism with C++17 and C++20

Rainer Grimm

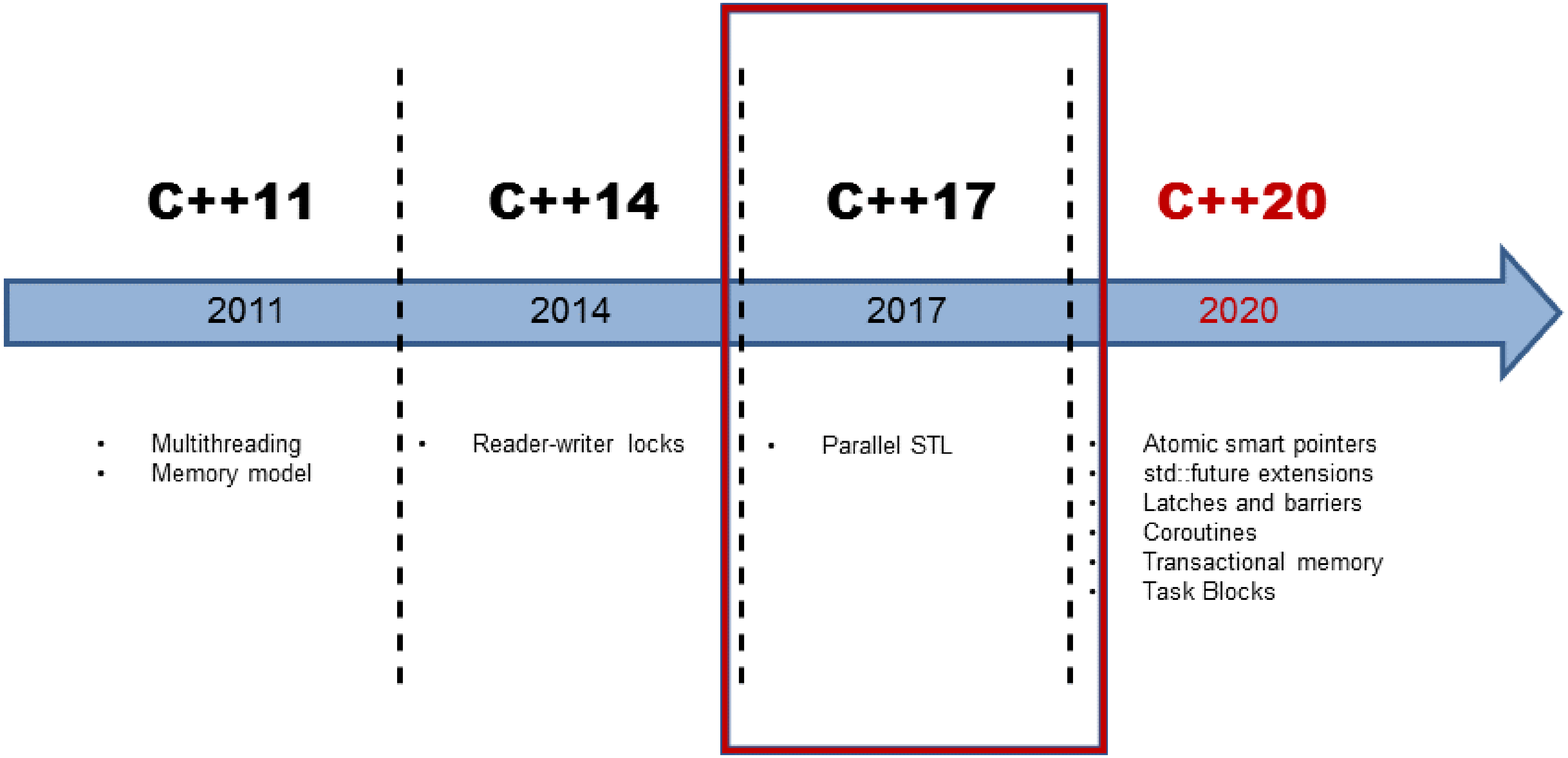
Training, Coaching and, Technology Consulting

www.ModernesCpp.de

Concurrency and Parallelism in C++



Concurrency and Parallelism in C++17



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

```
movslq  -8(%rbp), %rax
movl    vec(,%rax,4), %ecx
addl    $5, %ecx
movslq  -8(%rbp), %rax
movl    %ecx, res(,%rax,4)
```

Vectorised

```
movdqa  .LCPI0_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
```

Parallel STL

```
using namespace std;
vector<int> vec = {1, 2, 3, 4, 5, .... }


sort(vec.begin(), vec.end());     // sequential as ever

sort(execution::seq, vec.begin(), vec.end());       // sequential
sort(execution::par, vec.begin(), vec.end());       // parallel
sort(execution::par_unseq, vec.begin(), vec.end()); // par + vec
```

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`

- Haskells function `map` is called `std::transform` in C++
- `std::transform_reduce` → `std::map_reduce`

```
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 and deadlocks

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

➡ The access to **`numComp`** has to be atomic.

Parallel STL

- Static execution strategy

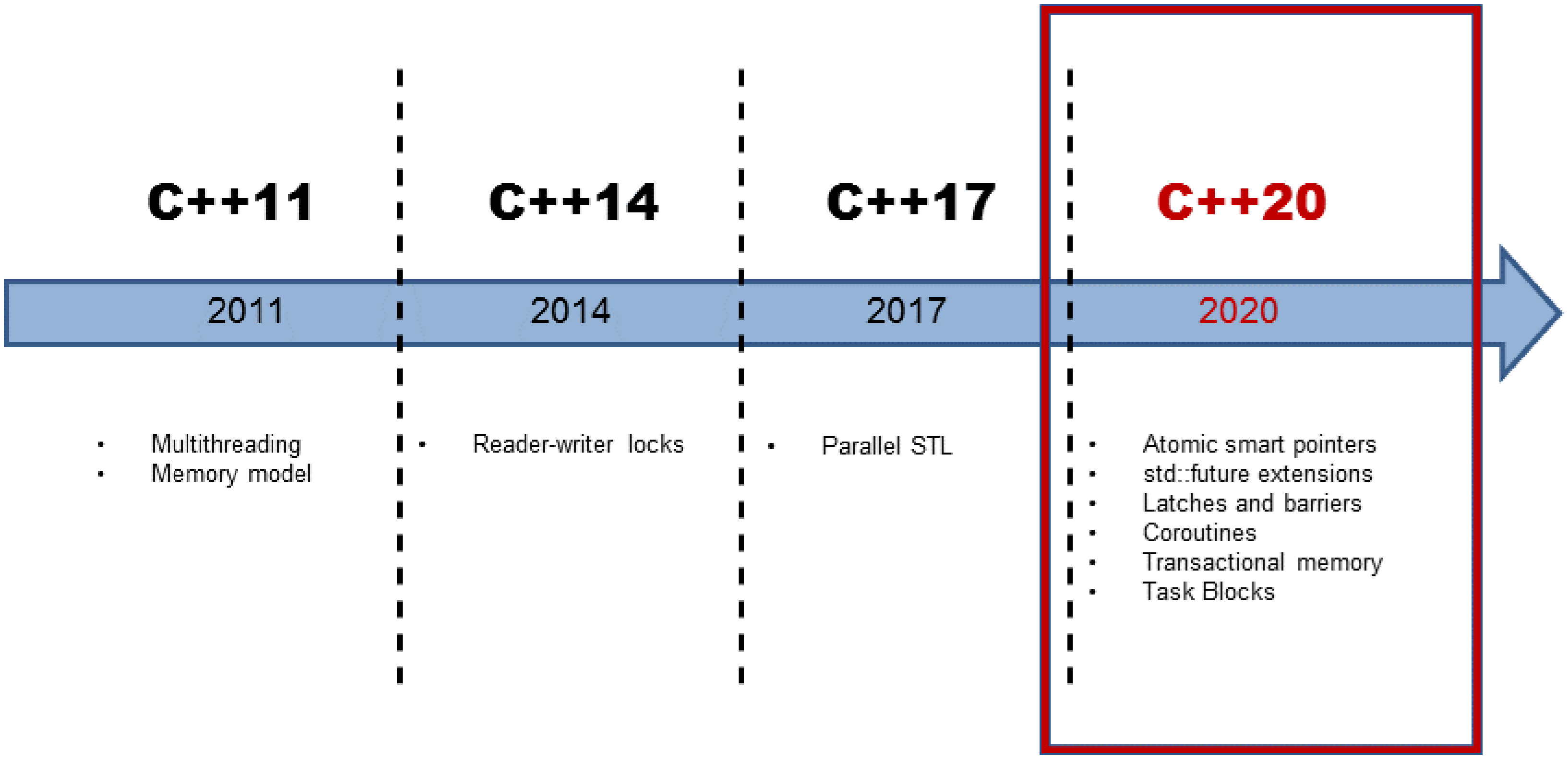
```
template <class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last){
  if(first == last) return;
  auto pivot = *next(first, distance(first,last) / 2);
  ForwardIt middle1 = partition(parallel::par, first, last,
                       [pivot](const auto& em){ return em < pivot; });
  ForwardIt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

~~Parallel STL~~ => C++20

- Dynamic execution strategy

```
std::size_t threshold = .... ; // some value

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

Concurrency and Parallelism in C++20



Atomic Smart Pointers

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

- Issues:
  - The managing of the control block and the deletion of the resource is thread-safe. The access to the ressource is not thread-safe.
  - You should use smart pointers.
- *Solution:*
  - C++11 allows atomic operations on `std::shared_ptr`.

New atomic smart pointers

- `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. → extremly error-prone

    `std::atomic_store(&sharPtr, localPtr)` ≠ `sharPtr = localPtr`
- Performance
  - `std::shared_ptr` has to be design for the special use-case.

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);
    }
};
```

std::future Extensions

`std::future` doesn't support composition

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

```
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

```
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

```
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();
```

std::future Extensions

- **make_ready_future and make_exception_future:** create a future directly

```
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;
}
```

Futher information

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

Latches and Barriers

C++ has no semaphor ➡ 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 = 0)`: 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 sychronisation 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.

Latches and Barriers

```
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
  - 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.

Coroutines

	Function	Coroutine
invoke	`func(args)`	`func(args)`
return	`return statement`	`co_return someValue`
suspend		`co_await someAwaitable` `co_yield someValue`
resume		`coroutine_handle<>::resume()`

A function is a coroutine if it has a `co_return`, `co_await`, `co_yield` call or if it has a range-based for loop with a `co_await` call.

Coroutines: Generators

```
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: Waiting instead of Blocking

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_await accept.accept();
  auto req = co_await so.read();
  auto resp = handleRequest(req);
  co_await so.write(resp);
}
```

Transactional Memory

Transactional Memory is the idea of transactions from the data base theory applied to software.

- A transaction has the ACID properties without ***D****urability*

```
atomic{
  statement1;
  statement2;
  statement3;
}
```

- **A**tomicity: all or no statement will be performed
- **C**onsistency: the system is always in a consitent state
- **I**solation: a transaction runs total isolation
- **D**urability: the result of a transaction will be stored

Transactional Memory

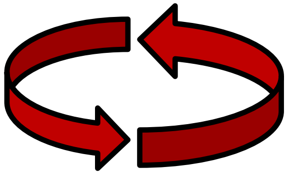
- Transactions
  - build a total order
  - feel like a global lock

  → Optimistic approach ≠ lock

- Workflow

Retry

Rollback



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 performend once more.

Transactional Memory

- Two forms
  - `synchronized` blocks
    - *relaxed* transaction
    - are not transaction in the pure sense

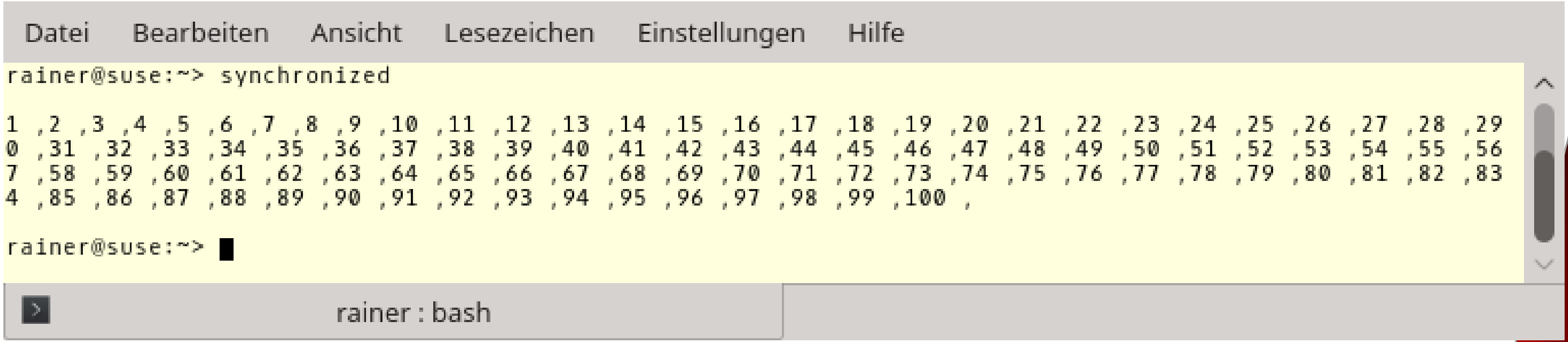
  ➡ can have `transaction-unsafe` code

  - `atomic` blocks
    - atomic blocks
    - are available in three variations

  ➡ can only execute `transaction-safe` code

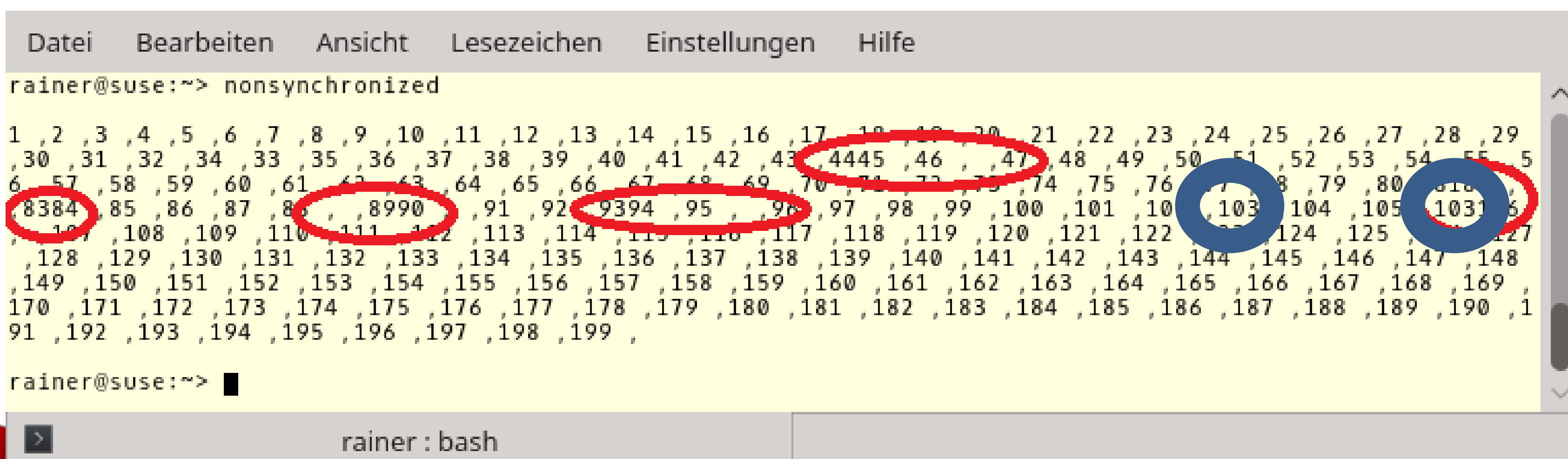
Transactional Memory: `synchronized` Blocks

```
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(); });
```



Transactional Memory: `synchronized` Blöcke

```
void inc() {
  synchronized{
    std::cout << ++i << " ,";
    this_thead::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 << " ,"; });
```



Transactional Memory

- `atomic` blocks

```
atomic_<Exception_specifier>{  // begin transaction
  ...
} // end transaction
```

- Exception
  - `atomic_noexcept`:
    - `std::abort` will be performend
  - `atomic_cancel`:
    - `std::abort` will be performed. If it was a `transaction_safe` exception for cancelling the transaction. ➡ Stops the transaction, puts the atomic block to its initial state and executes the exception.
  - `atomic_commit`:
    - publishes the transaction

Transactional Memory: Atomic Blocks

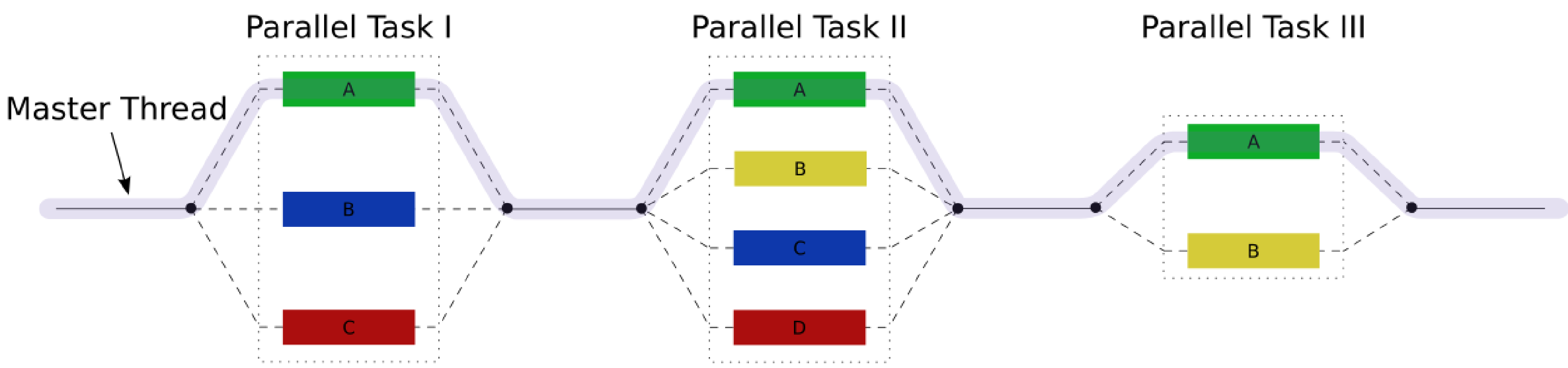
```
int i = 0;
void func() {
  atomic_noexcept{
    cout << ++i << " ,"; // non transaction-safe code
  }
}
```

A transaction can only perform `transaction-safe` code

➡ **compiler error**

Task Blocks

Fork-join parallelism with task blocks.



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;
}
```

- **`define_task_block`**
  - tasks can be perfomed
  - tasks will be synchronised at the end of the task block
- **`run`**: starts a task

Task Blocks

define_task_block_restore_thread

```
      ...
(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)   });
      ...
      ...
```

wait

```
define_task_block([&](auto& tb){
  tb.run([&]{ process(x1, x2); });
  if (x2 == x3) tb.wait();
  process(x3, x4);
});
```

Task Blocks

- The scheduler

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

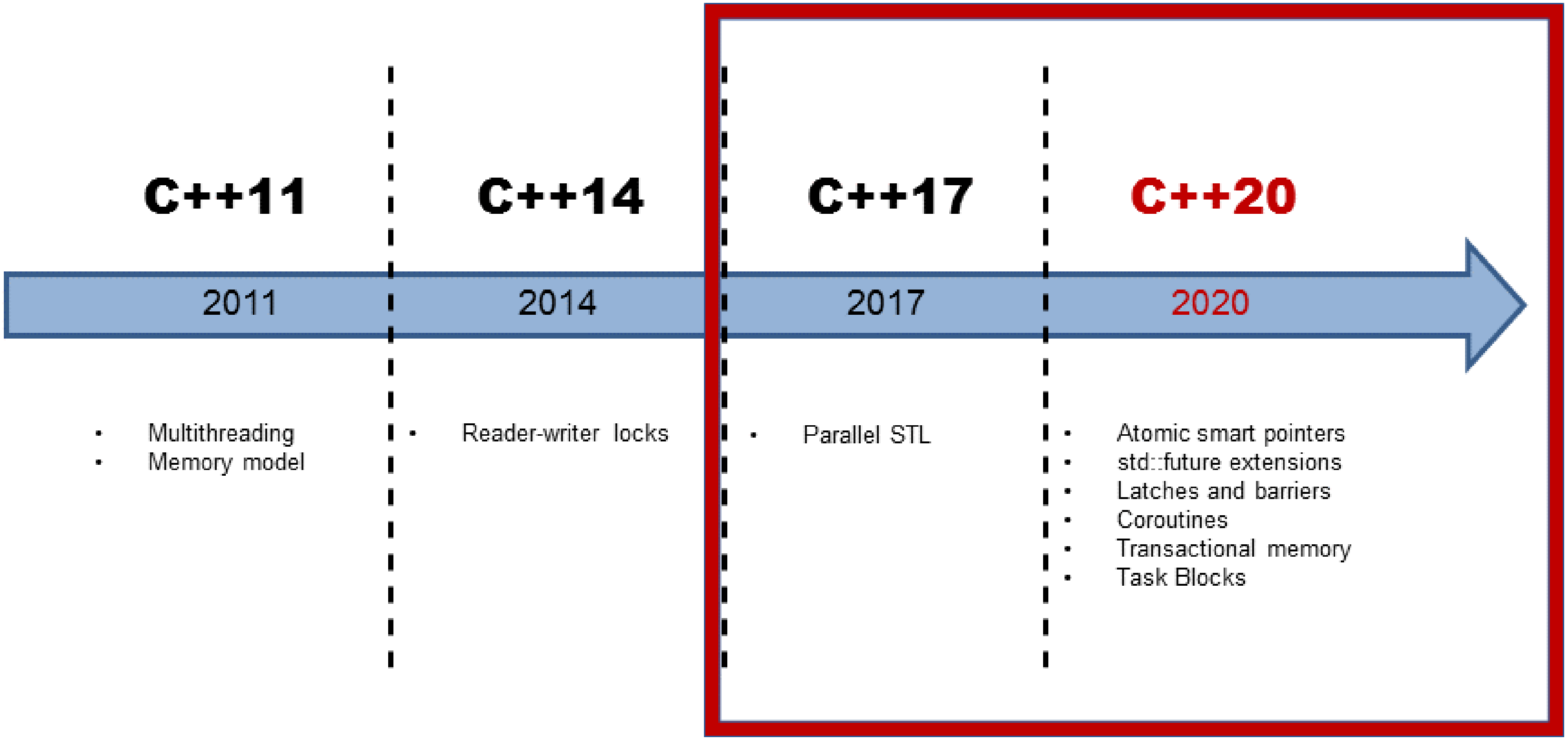
Parent **Child**

- **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 **Concurrency and Parallelism**



Concurrency and Parallelism in C++

Proposals

- Atomic smart pointers: N4058 (2014)
- `std::future` extensions: N4107 (2014)
- Latches and barriers: P0666R0 (2017)
- Coroutines: N4402 (2015)
- Transactional memory: N3919 (2014)
- Task blocks: N4411 (2015)
- Executors: P0443R2 (2017)
- Concurrent unordered associative containers: N3732 (2013)
- Concurrent Queue: P0260r0 (2016)
- Pipelines: N3534 (2013)
- Distributed counters: P0261r1 (2016)

Blogs

[www.grimm-jaud.de [De]](http://www.grimm-jaud.de)
[www.ModernesCpp.com [En]](http://www.ModernesCpp.com)

Rainer Grimm

Training, Coaching and,
Technology Consulting

www.ModernesCpp.de