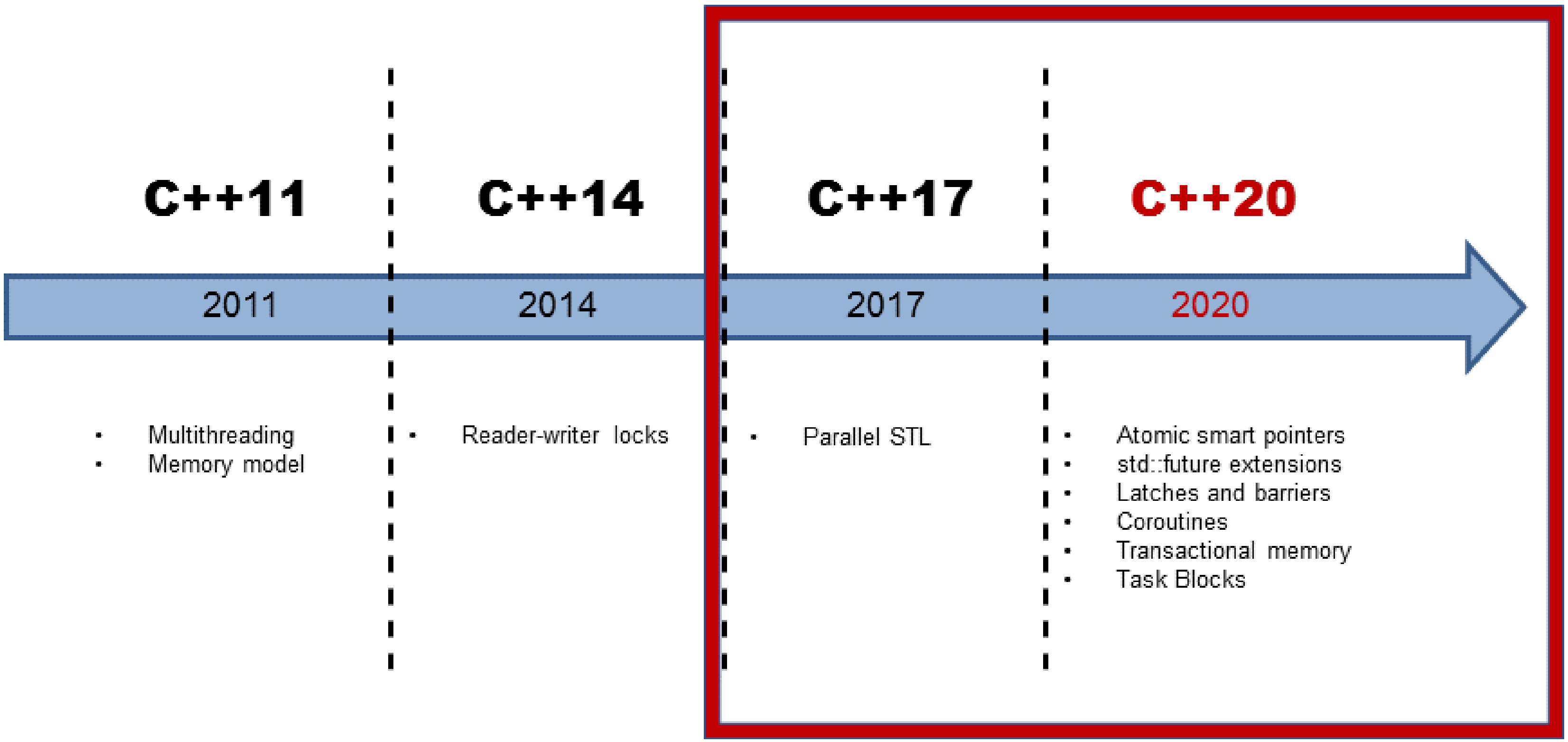
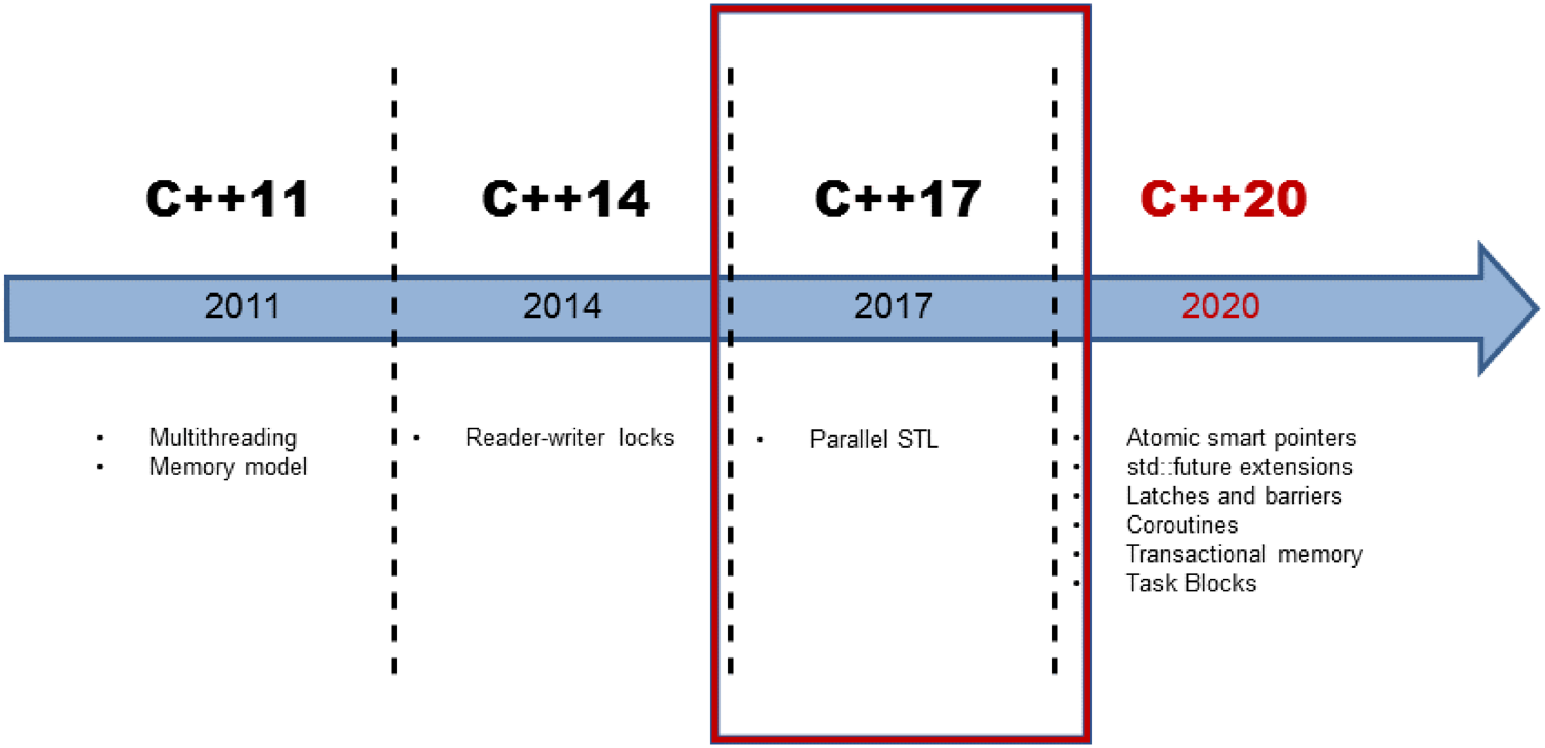
Parallelism and Concurrency in C++17 and C++20

Rainer Grimm
Training, Coaching and, Technology Consulting
www.grimm-jaud.de

Multithreading and Parallelism in C++



Multithreading in C++17



Parallel STL

The execution policy of the STL algorithm can be chosen.

- Execution policy

  `std::execution::seq`
  - Sequential execution on calling thread

  `std::execution::par`
  - Parallel

  `std::execution::par_unseq`
  - Parallel and vectorized
  - Performed on multiple data at the same time ➡ SIMD

Parallel STL

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

// static decision
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

// dynamic decision
size_t threshold= ...
execution_policy exec = execution::seq;
if(vec.size() > threshold) exec = execution::par;
sort(exec, vec.begin(), vec.end());
```

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

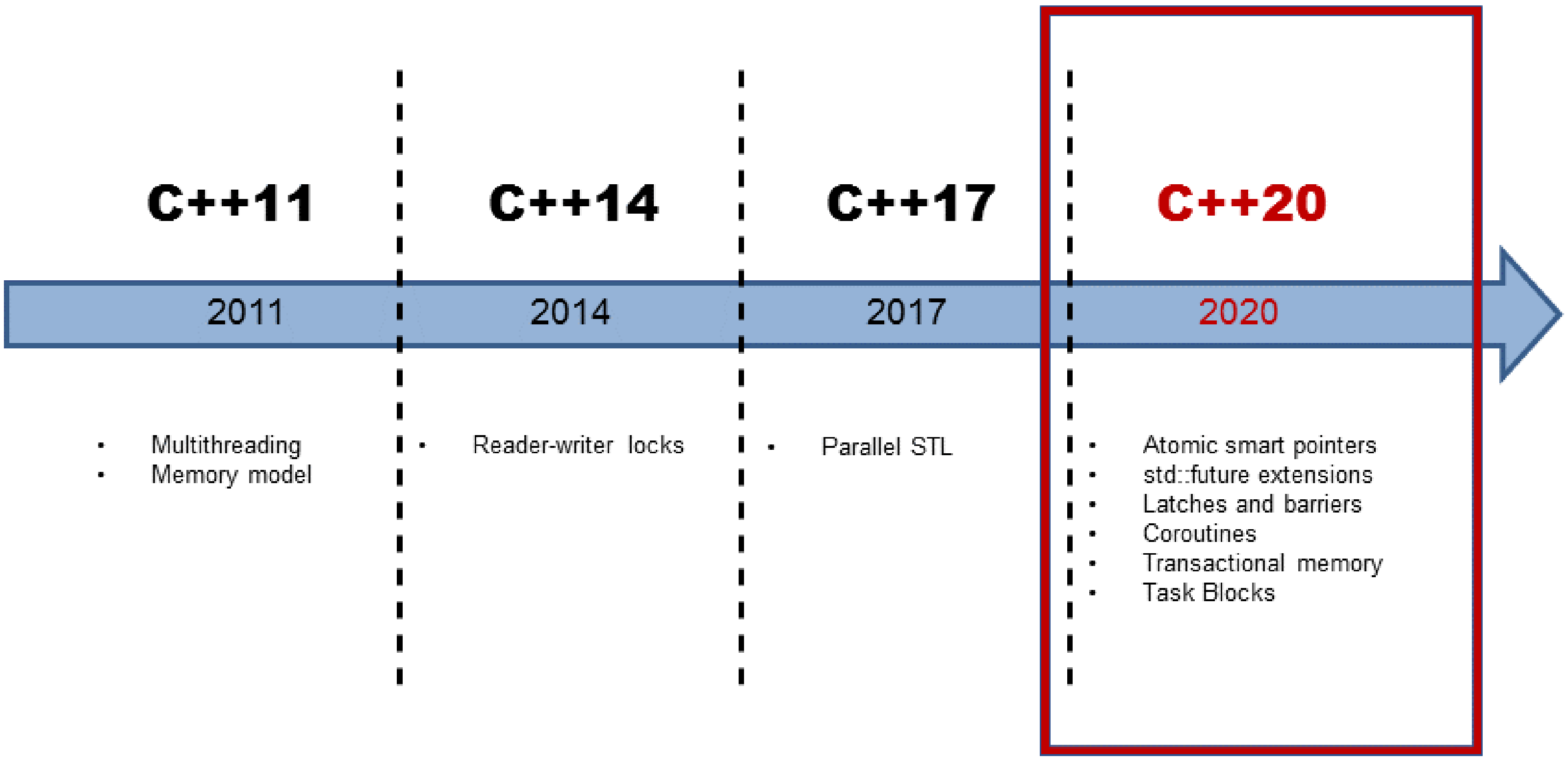
Parallel STL

`std::parallel::transform_reduce`

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

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


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

std::cout << result << std::endl;      //   21
```

Multithreading in C++20



Atomic Smart Pointers

C++11 has a

- `std::shared_ptr:` Shared ownership
- `std::weak_ptr:` Breaks cyclic references

- Issues:
  - The control block and the deletion of the resource is thread-safe, but not the resource.
  - C++11 has atomic operations for `std::shared_ptr`.

➡ New atomic data types:

- `std::atomic_shared_ptr`
- `std::atomic_weak_ptr`

std::future extensions

`std::future` support no function composition.

- `std::future` Improvements ➡ Continuation
  - `then`: Execute the second future, if the first one is done.

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

std::future extensions

- **`when_all:`** Execute the future when all of the 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 when any 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();
```

Latches and Barriers

C++ has no semaphores. ➡ Latches and barriers

- Concepts

  A thread waits eventually at a synchronization point until the counter is 0.

  - `latch` is a single-use barrier
    - `count_down_and_wait:` Decrements the counter and block until 0
    - `count_down:` Decrements the counter
    - `is_ready:` Checks the counter
    - `wait:` Waits until the counter is 0

Latches and Barriers

- `barrier` is a reusable barrier
  - `arrive_and_wait`: Waits at the synchronization point.
  - `arrive_and_drop`: Removes itself from the synchronization set.

- `flex_barrier` is a reusable and flexible barrier
  - The constructor can get a callable.
  - The callable will be executed in the completion phase.
  - The callable must return a value which specifies the counter for the next iteration.
  - It's the only barrier that can increase the counter.

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 generalized functions that can suspend and resume execution while keeping their state.

- Programming concept for
  - Cooperative task
  - Event loops
  - Iterators
  - Infinite lists
  - Pipes

Coroutines

Design Principles (James McNellis)

- **Scalable**, to billions of concurrent coroutines
- **Efficient**: Suspend/resume operations comparable in cost to function call overhead
- **Open-Ended**: Library designers can develop coroutines libraries
- **Seamless Interaction** with existing facilities with no overhead.
- **Usable** in environments where exceptions are forbidden or not available.

Coroutines: Generators

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

int main(){
  auto numbers= generatorForNumbers(-10);
  for (int i= 1; i <= 20; ++i) std::cout << numbers << " ";
  for (auto n: getForNumbers(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 soc= 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 transaction idea of databases applied to the software development.

- A transaction has the ACID property excluding **D**urability

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

- **A**tomicity: All or no statement will be executed.
- **C**onsistency: The system is always in a consistent state.
- **I**solation: A transaction runs in total isolation.
- **D**urability: The result of a committed transaction remains.

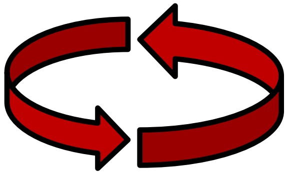
Transactional Memory

- Transactions
  - Execute in a single total order
  - Are protected (behave like a **global** lock)

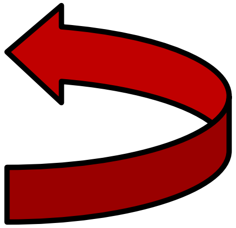
  ⇒ Use optimistic concurrency ≠ Locks

- Workflow

Retry



Rollback



A transaction remembers its initial state.

The transaction runs without synchronization.

The system detects a conflict with the initial state.

The transaction will be committed.

Transactional Memory

- Two forms
  - Synchronized block:
    - Relaxed transactions
    - Are no transactions in the strict sense.

  ➡ Can call `transaction-unsafe` code

  - Atomic blocks:
    - Atomic transactions
    - Are available in three forms.

  ➡ Can only call `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(); });
```

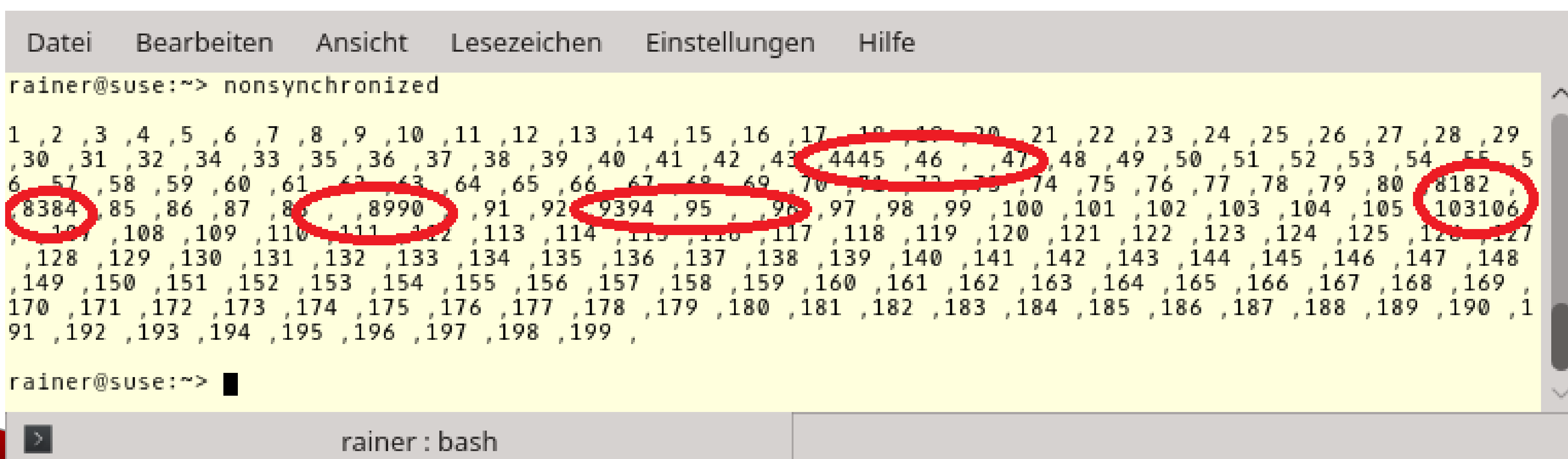
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
rainer@suse:~> synchronized

1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14 ,15 ,16 ,17 ,18 ,19 ,20 ,21 ,22 ,23 ,24 ,25 ,26 ,27 ,28 ,29
0 ,31 ,32 ,33 ,34 ,35 ,36 ,37 ,38 ,39 ,40 ,41 ,42 ,43 ,44 ,45 ,46 ,47 ,48 ,49 ,50 ,51 ,52 ,53 ,54 ,55 ,56
7 ,58 ,59 ,60 ,61 ,62 ,63 ,64 ,65 ,66 ,67 ,68 ,69 ,70 ,71 ,72 ,73 ,74 ,75 ,76 ,77 ,78 ,79 ,80 ,81 ,82 ,83
4 ,85 ,86 ,87 ,88 ,89 ,90 ,91 ,92 ,93 ,94 ,95 ,96 ,97 ,98 ,99 ,100 ,

rainer@suse:~>
```

rainer : bash

Transactional Memory: Synchronized blocks

```
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 occurs
  - `atomic_noexcept`:
    - `std::abort` is called.
  - `atomic_cancel`:
    - `std::abort` is called unless it was a `transaction_safe` exception. => Cancel the transaction, set the atomic block to is initial state and throw the exception.
  - `atomic_commit`:
    - Commit the transaction and throw the exception.

Transactional Memory: Atomic blocks

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

The transaction can only executed transaction-safe code.

➡ **Compile time error**

Transactional memory: transaction_safe

A function be

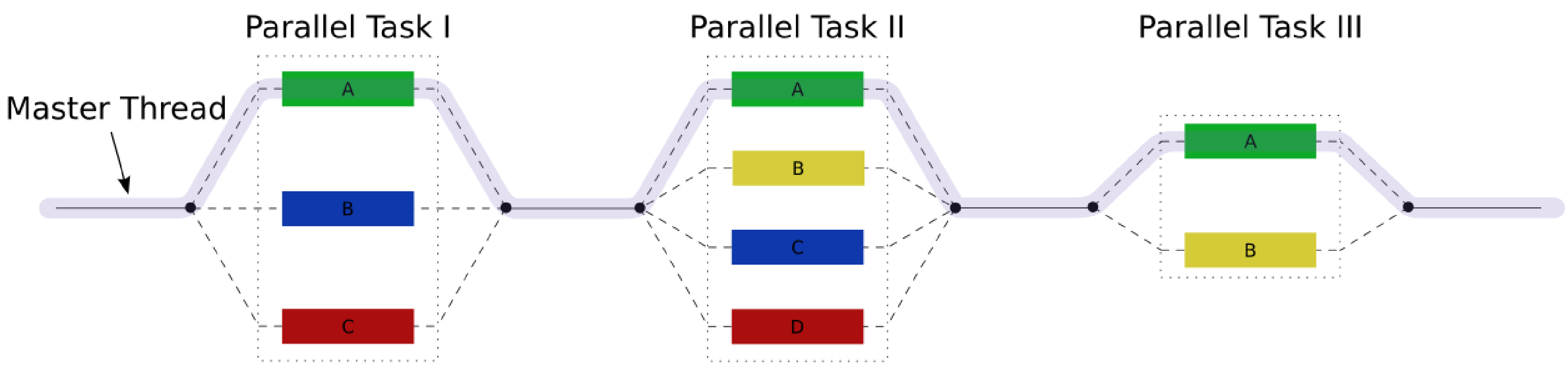
- be declared `transaction_safe`
- have a `transaction_unsafe` attribute.

```
int transactionSafeFunction() transaction_safe;
[[transaction_unsafe]] int transactionUnsafeFunction();
```

- `transaction_safe` is part of the type of the function.

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 potentially run
- The end of task block joins the tasks

run: Runs a task

Task Blocks

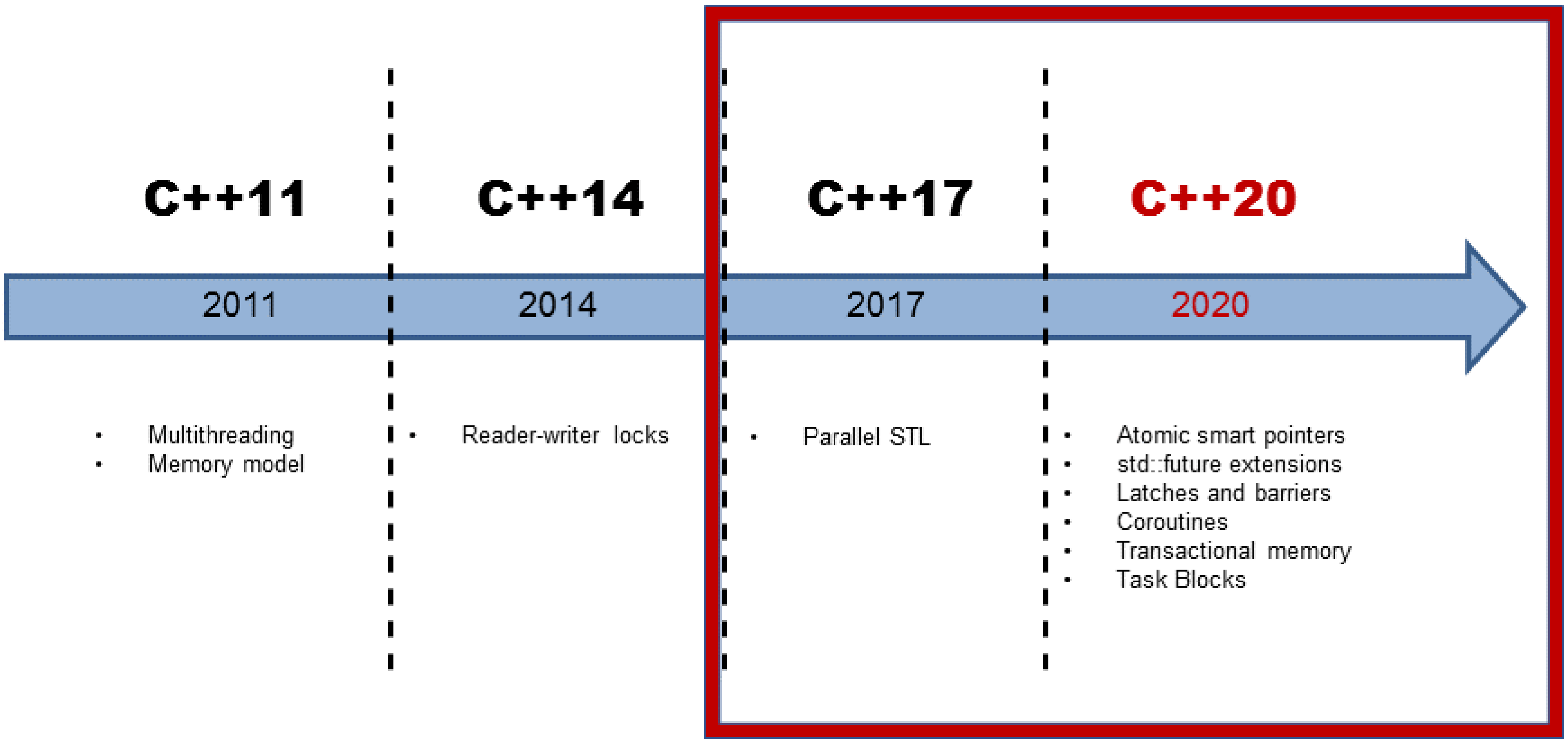
define_task_block_restore_thread

```
...
define_task_block([&](auto& tb)
  tb.run([&]{[] func(); });
  define_task_block_restore_thread([&](auto& tb){
    tb.run([&]([]{ func2(); });
    define_task_block([&](auto& tb){
      tb.run([&]{ func3(); }
    });
    ...
    ...
  });
  ...
  ...
});
...
...
```

wait

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

Multithreading and Parallelism in C++



Further Information

- **Modernes C++:** Training, coaching, and technology consulting by Rainer Grimm
  - www.ModernesCpp.de

- Blog to modern C++
  - www.grimm-jaud.de (German)
  - www.ModernesCpp.com (English)

- Contact
  - @rainer_grimm
  - schulungen@grimm-jaud.de

Rainer Grimm
Training, Coaching, and Technology Consulting
www.grimm-jaud.de