Advanced Developers Conference
Development for Professionals!

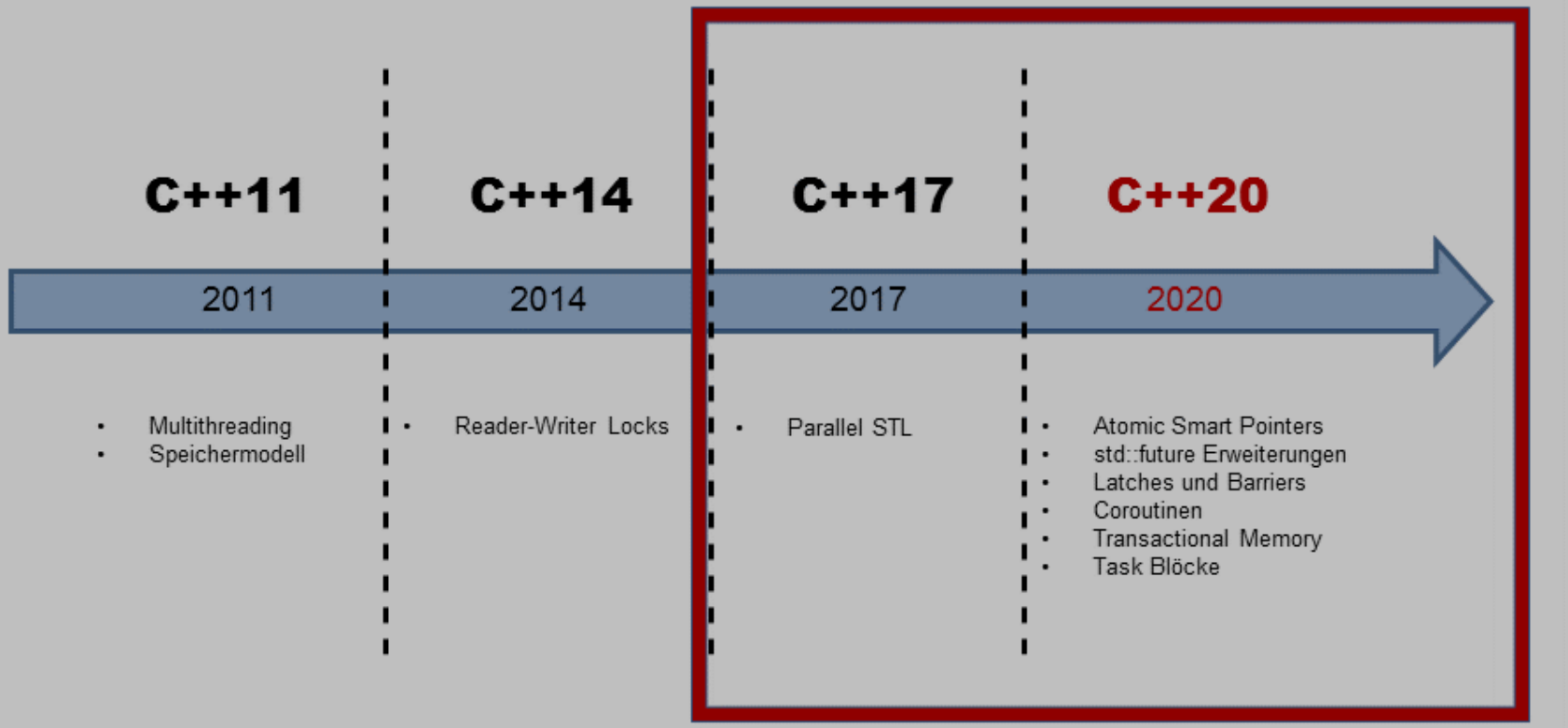
# Gleichzeitigkeit und Parallelisierung in C++17 und C++20

Rainer Grimm

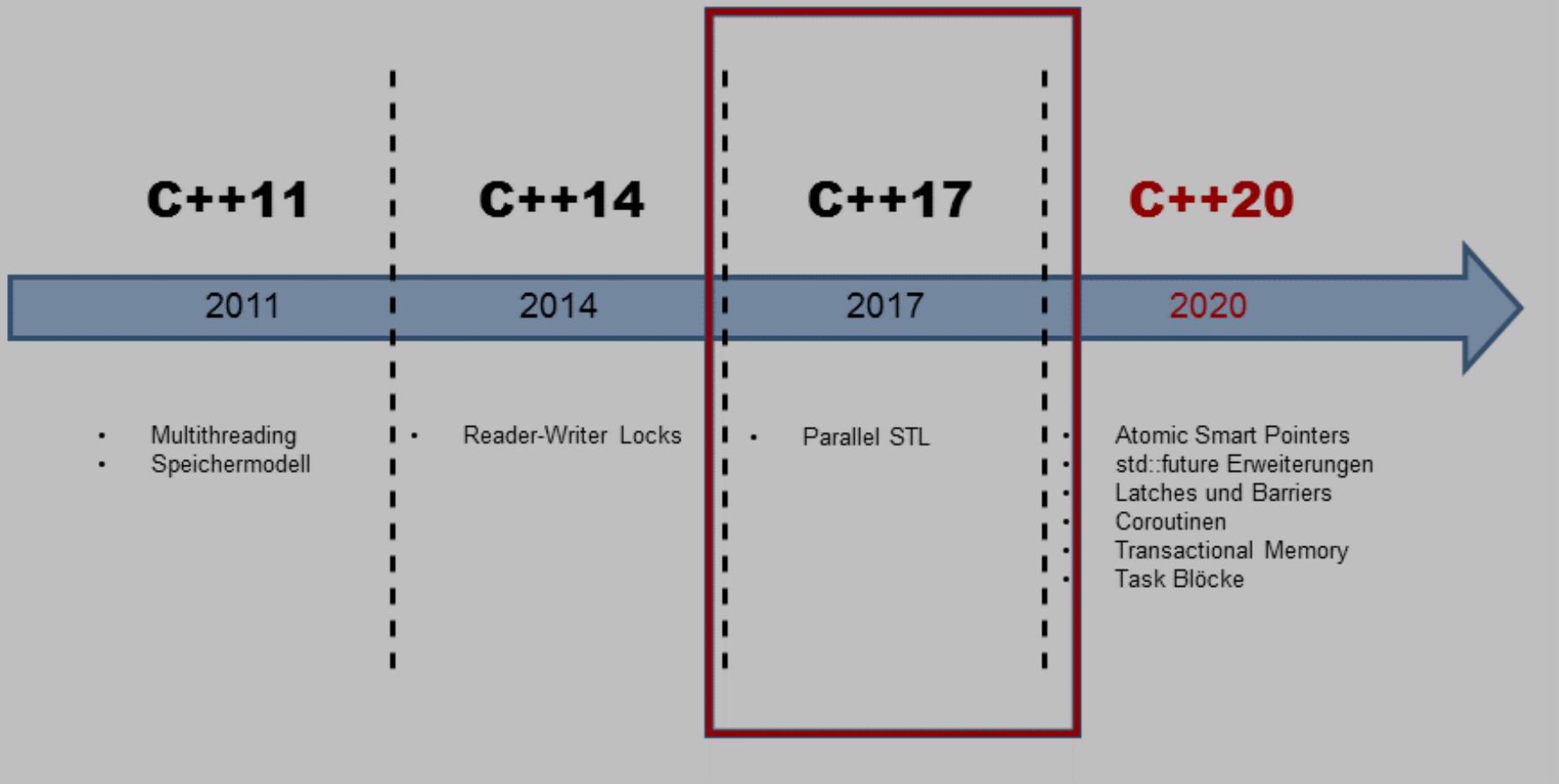
Training, Coaching und Technologieberatung

www.ModernesCpp.de

# Gleichzeitigkeit und Parallelisierung in C++



# Gleichzeitigkeit und Parallelisierung in C++17



# Parallel STL

Die Ausführungsstrategie eines STL Algorithmus kann ausgewählt werden.

- Ausführungsstrategie

`std::execution::seq`

  - Sequentiell in einem Thread

`std::execution::par`

  - Parallel

`std::execution::par_unseq`

  - Parallel und vektorisiert ➡ 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;
  }
}
```

## Nicht vektorisiert

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

## Vektorisiert

```
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::parallel::transform_reduce`

- Haskells Funktion `map` heißt in C++ `std::transform`
- `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
```

# Parallele STL

- Gefahr von kritischen Wettläufen und Verklemmungen

```
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; }
);
```

➡ Der Zugriff auf **numComp** muss atomar sein.

# Parallel STL

- Statische Ausführungsstrategie

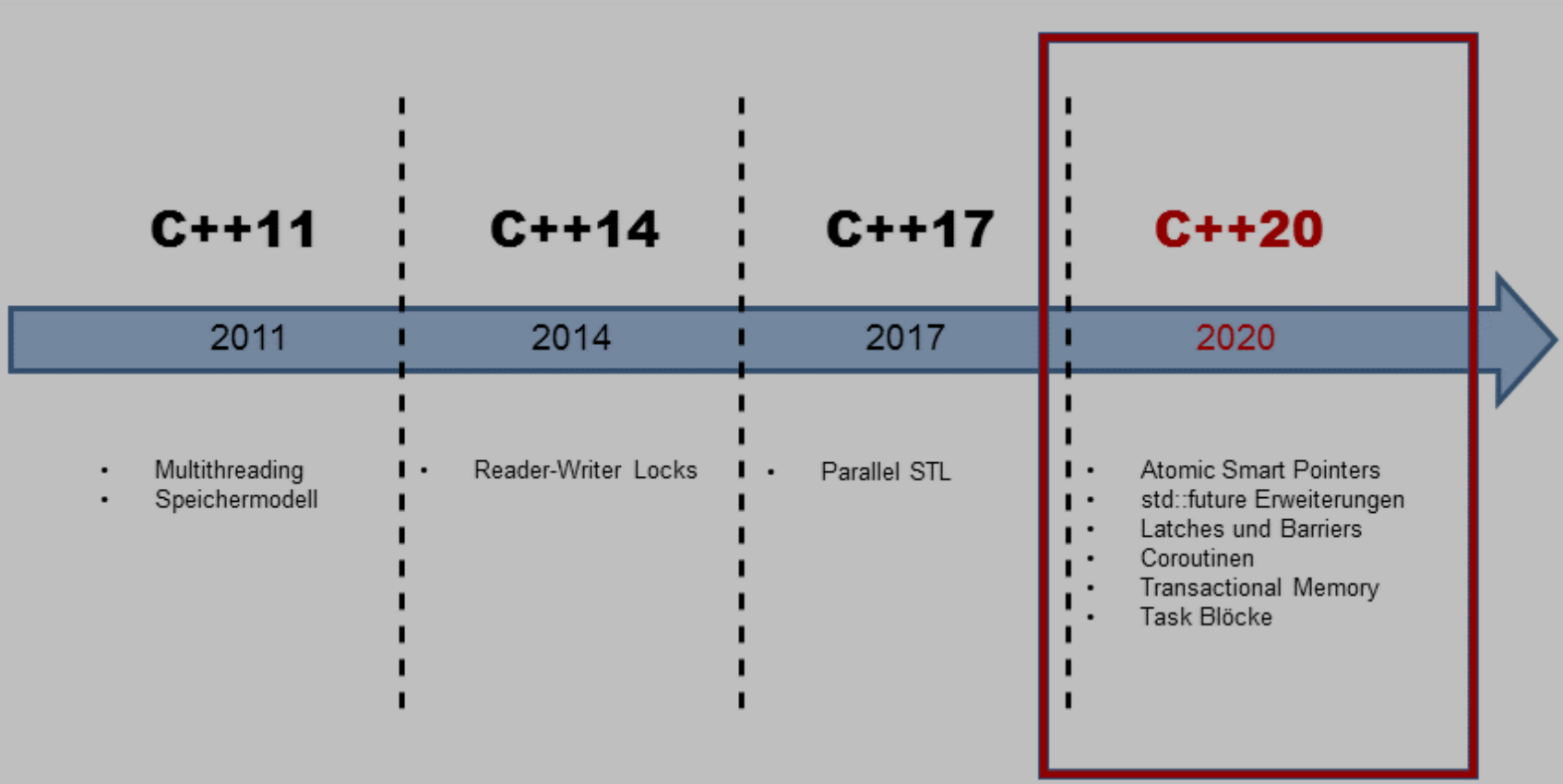
```
template <class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last){
  if(first == last) return;
  auto pivot = *next(first, distance(first,last)/2);
  ForwardIt middle1 = partition(parallel::par, first, last,
                    [pivot](const auto& em){ return em < pivot; });
  ForwardIt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

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

- Dynamische Ausführungsstrategie

```
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);
}
```

# Gleichzeitigkeit und Parallelisierung in C++20



# Atomic Smart Pointers

C++11 besitzt einen `std:shared_ptr` für geteilte Besitzverhältnisse.

- Probleme:
  - Der Kontrollblock und das Löschen der Ressource ist thread-safe, die Ressource ist nicht thread-safe.
  - Smart Pointer sollen in Multithreading Programmen verwendet werden.
- *Lösung:*
  - C++11 besitzt atomare Operationen für `std::shared_ptr`.

➡ Neue atomare Datentypen

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

# Atomic Smart Pointer

## 3 Gründe

- Konsistenz:
  - Der `std::shared_ptr` ist der einzige nicht-atomare Datentyp, für den atomare Operationen existieren.
- Korrektheit:
  - Die richtige Verwendung der atomaren Operationen basiert auf der Disziplin des Programmierers. ➡ Sehr fehleranfällig

    `std::atomic_store(&sharPtr, localPtr)` ≠ `sharPtr = localPtr`
- Performanz
  - `std::shared_ptr` müssen für den speziellen Anwendungsfall Multithreading entworfen werden

# 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 Erweiterungen

`std::future` unterstützt keine Komposition.

- `std::future` Verbesserung ➡ Continuation
  - `then`: Führe den nächsten Future aus, wenn der Vorgänger fertig ist.

```
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 Erweiterungen

- **`when_all`:** Führe den Future aus, wenn alle Futures fertig sind.

```
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`:** Führe den Future aus, wenn ein Future fertig ist.

```
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 Erweiterungen

- **`make_ready_future und make_exception_future:`**
  Erzeugen direkt einen Future ohne einen Promise.

```
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;
}
```

## Weitere Informationen

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

# Latches und Barriers

C++ besitzt keine Semaphoren ➡ Latches und Barriers

- Zentrale Idee

  Ein Thread wartet gegebenenfalls an einem Synchronisationspunkt bis ein Zähler den Wert null besitzt.

  - `latch` ist für den einmaligen Gebrauch konzipiert
    - `count_down_und_wait:` Dekrementiert den Zähler und wartet, bis dieser null ist.
    - `count_down:` Dekrementiert den Zähler
    - `is_ready:` Prüft den Zähler
    - `wait:` Wartet, bis der Zähler null ist

# Latches und Barriers

- `barrier` kann mehrmals verwendet werden
  - `arrive_und_wait`: Wartet am Synchronisationspunkt
  - `arrive_und_drop`: Entfernt sich selbst aus dem Synchronisationsmechanismus

- `flex_barrier` ist ein wiederverwendbare und anpassbare Barriere
  - Der Konstruktor bekommt eine aufrufbare Einheit.
  - Die aufrufbare Einheit wird in der *completion phase* ausgeführt.
  - Die aufrufbare Einheit muss einen Zahl zurückgeben, die den Zähler für die nächste Iteration festlegt.
  - Kann als einzige Barriere ihren Wert erhöhen.

# Latches und 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();
}
```

# Coroutinen

Coroutinen sind verallgemeinerte Funktionen, die ihre Ausführung unterbrechen und wieder aufnehmen können und dabei ihren Zustand behalten.

- Typische Einsatzgebiete
  - Kooperative Tasks
  - Eventschleifen
  - Unendliche Datenströme
  - Pipelines

# Coroutinen

Design Principles (James McNellis)

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

# Coroutinen

| | Function | Coroutine |
|---|---|---|
| invoke | `func(args)` | `func(args)` |
| return | `return statement` | `co_return statement` |
| suspend | | `co_await expression`<br>`co_yield expression` |
| resume | | `coroutine_handle<>::resume()` |

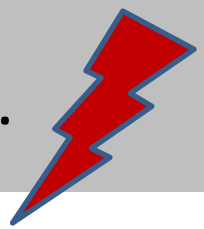
Ein Funktion ist ein Coroutine, falls sie einen Aufruf `co_return`, `co_await`, `co_yield` oder ein Range-basierte for-Schleife `co_await` enthält.

# Coroutinen: Generatoren

```
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 ...**



# Coroutinen: Warten statt blockieren

## Blockieren

```
Acceptor accept{443};

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

## Warten

```
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* ist die Idee der Transaktion aus der Datenbank auf die Software angewandt.

- Ein Transaktion besitzt die ACID Eigenschaften ohne ***D****urability*

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

- **A**tomicity: Alle oder keine Anweisung wird ausgeführt.
- **C**onsistency: Das System ist immer in einem konsistenten Zustand.
- **I**solation: Ein Transaktion läuft in vollkommener Isolation.
- **D**urability: Das Ergebnis einer Transaktion wird gespeichert.

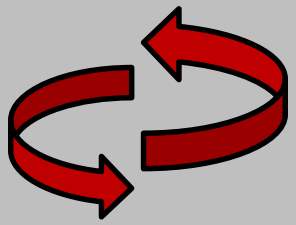
# Transactional Memory

- Transaktionen
  - werden in einer totalen Ordnung ausgeführt.
  - verhalten sie wie wenn sie ein globales Lock verwenden.

    ➡ Optimistischer Ansatz ≠ Lock

- Arbeitsablaufs

***Retry***



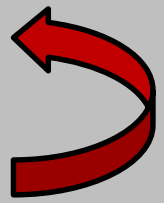
Die Transaktion merkt sich ihren Anfangszustand.

Die Transaktion wird ohne Synchronisation ausgeführt.

Die Laufzeit entdeckt einen Konflikt mit ihrem Anfangszustand.

Die Transaktion wird wiederholt.

***Rollback***



# Transactional Memory

- Zwei Formen
  - `synchronized` Blöcke
    - *relaxed* Transaktion
    - Sind im strengen Sinn keine Transaktionen

    ➡ Können `transaction-unsafe` Code ausführen

  - `atomic` Blöcke
    - Atomare Transaktionen
    - Gibt es in drei Variationen

    ➡ Können nur `transaction-safe` Code ausführen

# Transactional Memory: synchronized Blöcke

```
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 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 << " ,"; });
```

Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
rainer@suse:~> nonsynchronized

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
,30 ,31 ,32 ,34 ,33 ,35 ,36 ,37 ,38 ,39 ,40 ,41 ,42 ,43 ,4445 ,46 , ,47 ,48 ,49 ,50 ,51 ,52 ,53 ,54 ,55 ,5
6 ,57 ,58 ,59 ,60 ,61 ,62 ,63 ,64 ,65 ,66 ,67 ,68 ,69 ,70 ,71 ,72 ,73 ,74 ,75 ,76 ,77 ,78 ,79 ,80 ,8182 ,
,8384 ,85 ,86 ,87 ,88 , ,8990 ,91 ,92 ,9394 ,95 , ,96 ,97 ,98 ,99 ,100 ,101 ,102 ,103 ,104 ,105 ,103106
, ,107 ,108 ,109 ,110 ,111 ,112 ,113 ,114 ,115 ,116 ,117 ,118 ,119 ,120 ,121 ,122 ,123 ,124 ,125 ,126 ,127
,128 ,129 ,130 ,131 ,132 ,133 ,134 ,135 ,136 ,137 ,138 ,139 ,140 ,141 ,142 ,143 ,144 ,145 ,146 ,147 ,148
,149 ,150 ,151 ,152 ,153 ,154 ,155 ,156 ,157 ,158 ,159 ,160 ,161 ,162 ,163 ,164 ,165 ,166 ,167 ,168 ,169 ,
170 ,171 ,172 ,173 ,174 ,175 ,176 ,177 ,178 ,179 ,180 ,181 ,182 ,183 ,184 ,185 ,186 ,187 ,188 ,189 ,190 ,1
91 ,192 ,193 ,194 ,195 ,196 ,197 ,198 ,199 ,

rainer@suse:~>
```

rainer : bash

# Transactional Memory

- `atomic` Blöcke

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

- Ausnahme
  - `atomic_noexcept`:
    - `std::abort` wird ausgeführt.
  - `atomic_cancel`:
    - `std::abort` wird ausgeführt, falls es eine `transaction_safe` Ausnahme war. ➡ Beende die Transaktion, setze den atomaren Block auf seinen Anfangszustand und führe die Ausnahme aus.
  - `atomic_commit`:
    - Veröffentliche die Transaktion und führe die Ausnahme aus.

# Transactional Memory: Atomic Blöcke

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

Die Transaktion kann nur `transaction-safe` Code ausführen.

➡ **Compiler Fehler**

# Transactional Memory

Eine Funktion kann

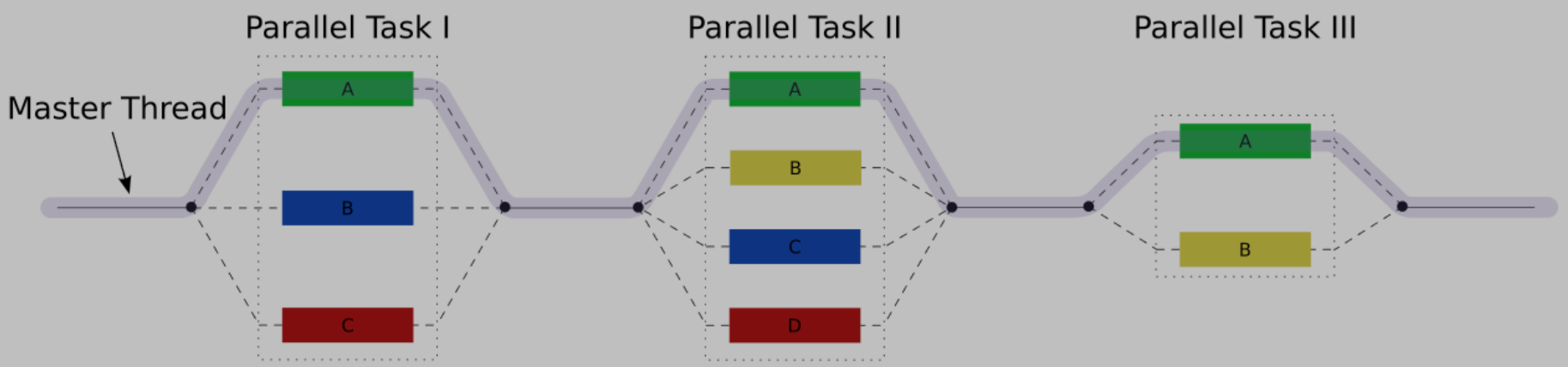
- `transaction_safe` deklariert werden.
- das `transaction_unsafe` Attribut besitzen.

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

- `transaction_safe` gehört zum Typ einer Funktion.

# Task Blöcke

## Fork-join Parallelisierung mit Task Blöcken.



# Task Blöcke

```
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 könne ausgeführt werden
  - Am Ende des Task Blocks werden die Tasks synchronisiert
- **`run:`** Startet einen Task

# Task Blöcke

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

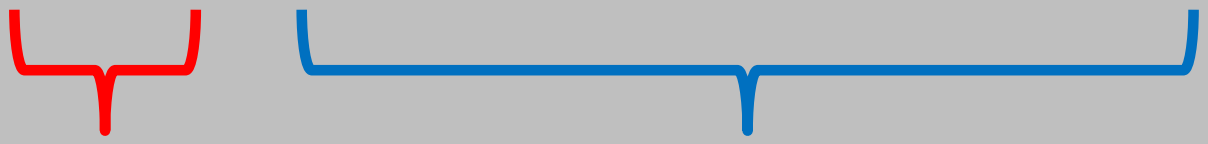
## wait

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

# Task Blöcke

- Der Scheduler

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



**Parent** **Child**

- **Child stealing**: Der Scheduler klaut die Aufgabe und führt sie aus.
- **Parent stealing**: Der Task Block für seine Aufgabe selber aus. Der Scheduler schnappt sich den Parent.

➡ Beide Strategien sind in C++20 möglich.

# Weitere Proposals

- Executors
  - Objekt für das Erzeugen von *execution agents*
  - *Executors* beantworten die Frage
    - Was soll ausgeführt werden?
    - Wann soll es ausgeführt werden?
    - Wo soll es ausgeführt werden?
    - Wie soll es ausgeführt werden?
  - Vorgeschlagene *executors*
    - `thread_per_task`
    - `thread_pool_executor`
    - `serial_executor`
    - `loop_exector`
    - `system_executor` (default executor)

# Weitere Proposals

- Concurrent ungeordnete Container
  - Name: `concurrent_unordered_map`
  - Geben ein `std::optional` Objekt zurück
  - Interface:
    - `find, insert, exchange, erase, reduce, clear, size, for_each, is_lock_free`

- Concurrent Queue
  - Ist an das Interface einer `std::deque` angelehnt
  - Zwei Implementierungen einer Queue fester Länge
    - Locking Buffer Queue
    - Lock-Free Buffer Queue

# Weitere Proposals

- Pipelines
  - *Parallele Unix-Pipelines* (Googles Open Source Implementierung)

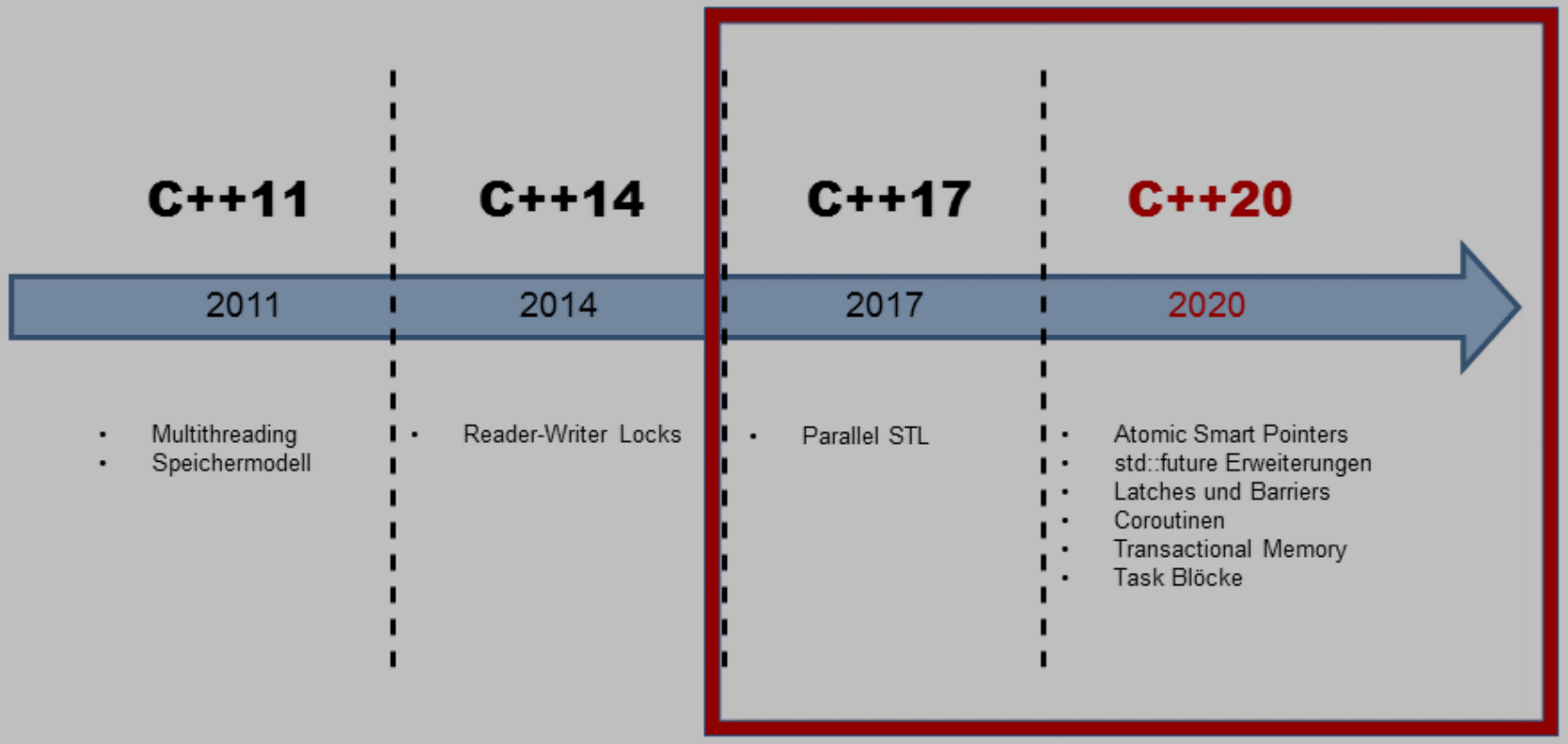
```
pipeline::execution task(
  pipeline::from(filenames) |
  pipeline::parallel(read_file | grep_fn, 8) | vgrep_fn | sed_fn |
  output_queue).run(&thread_pool);

pipeline::execution task(
  pipeline::from(filenames) |
  pipeline::parallel(read_file | grep_fn | vgrep_fn | sed_fn, 8) |
  output_queue).run(&thread_pool);
```

- Verteilte Zähler
  - Zähler werden lokal gepuffert und können synchronisiert werden (*push* oder *pull*)

# Gleichzeitigkeit und Parallelisierung in C++

## Multithreading

## Gleichzeitigkeit und Parallelisierung



# Gleichzeitig und Parallelisierung in C++

# Meine Blogs

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

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de