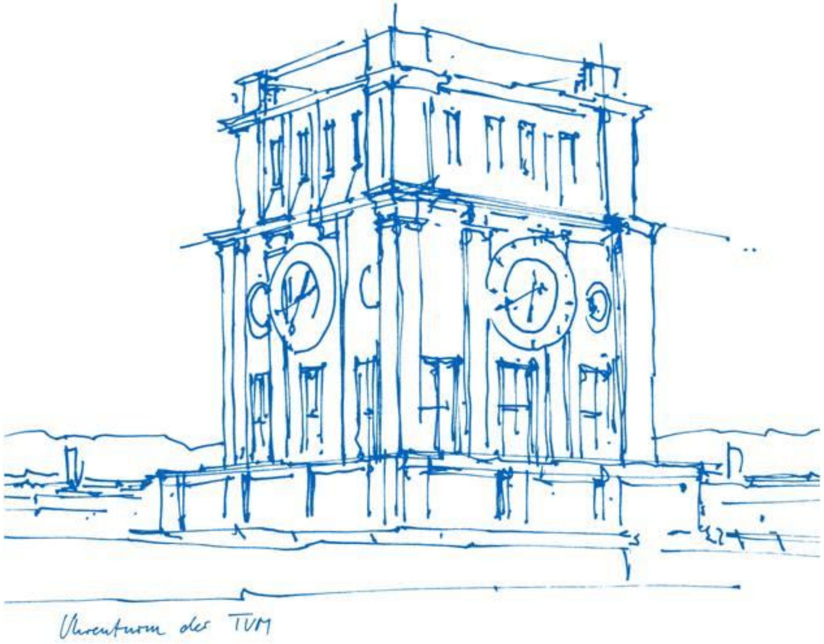
Concurrency in C++17: Parallel STL

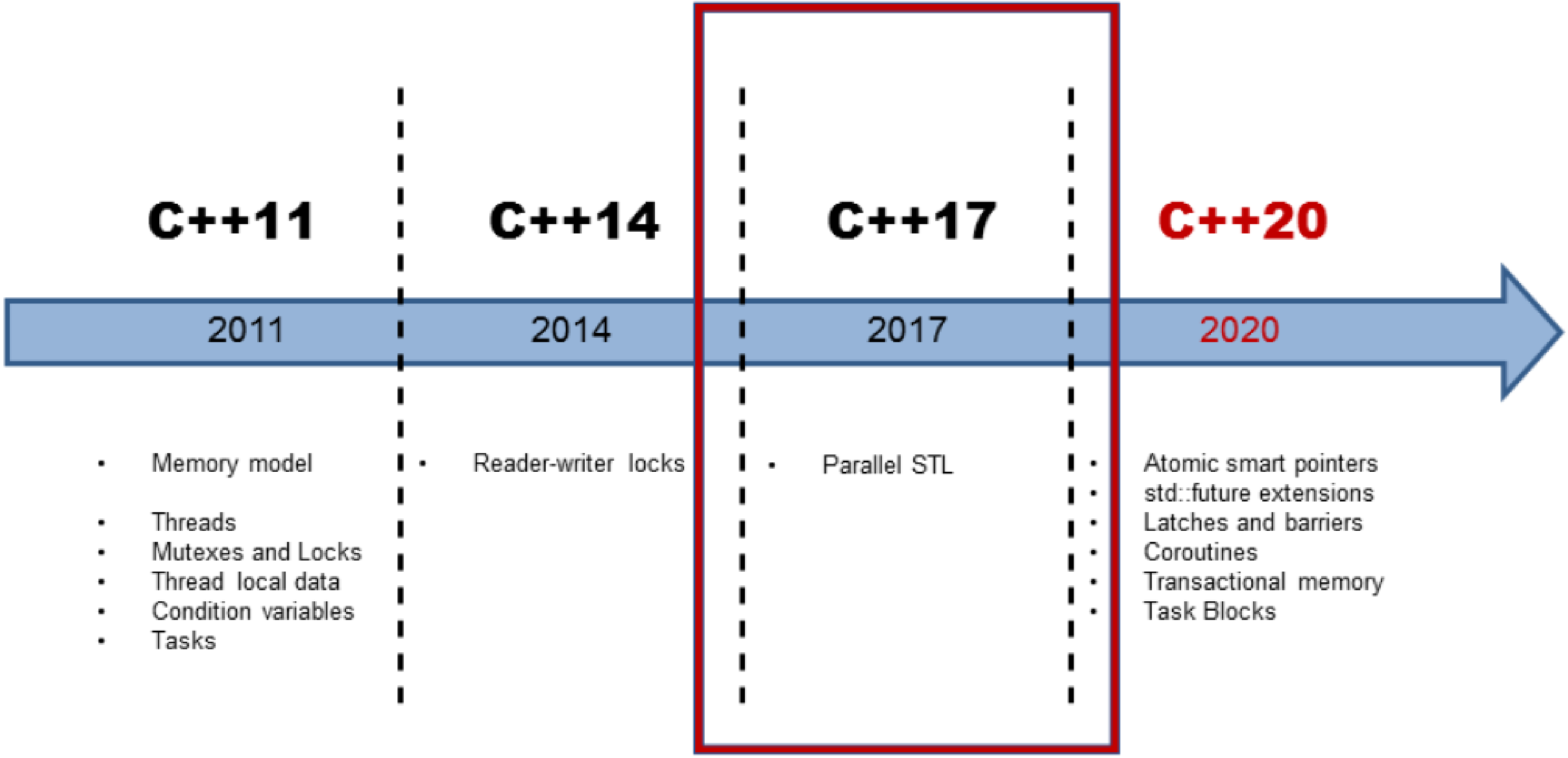
Philipp Bock
Technical University of Munich
Department of Informatics
Chair for Computer Aided Medical Procedures
& Augmented Reality

Garching, 19th of June, 2018



Content

- What is new in C++17?
- Recap : STL algorithms & parallelism
- Execution Policy
- New STL algorithms
- Important Notes
- Outlook
- Exercises



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

2020

- Atomic smart pointers
- std::future extensions
- Latches and barriers
- Coroutines
- Transactional memory
- Task Blocks

Q.: What are STL algorithms?

Q.: What are STL algorithms?

→Generic operations on sequences

Types of operations

Non-modifying sequence operations

Mutating sequence operations

Numeric operations

Sorting and related operations

Q.: What is parallelism?

Concurrency: tasks executed during the same time period

Parallelism: tasks literally run at the same time

Concept of execution policies

- Permits concurrent execution of STL algorithms

Concept of execution policies

- Permits concurrent execution of STL algorithms

- Implementation is compiler specific

  →Three standard execution policies

  →no personal execution policies

  → compiler implementation can do anything with respect to the policy constraints (including GPU support)

  → there can be compiler specific execution policies

Execution Policies

Execution policies are defined in #include <execution>

std::execution::seq

std::execution::par

std::execution::par_unseq

sequential_policy

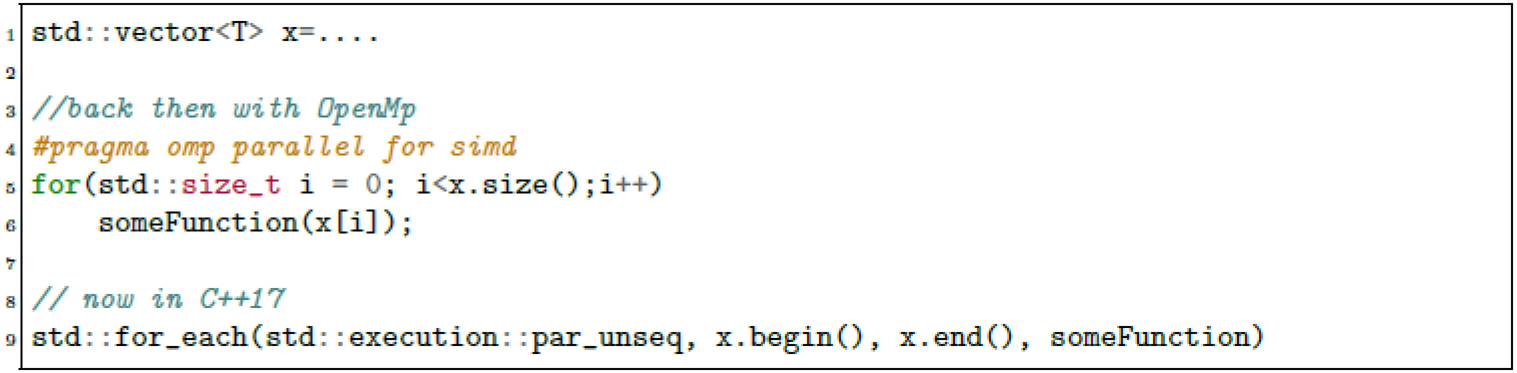
- Algorithm executes indeterminately sequenced

- Usage:
  - →debugging
  - →fallback for efficency reasons

parallel_policy

- Functions are permitted to execute within a new thread
- Invocations executing in the same thread are indeterminately sequenced
- Locks are allowed

parallel_unsequenced_policy



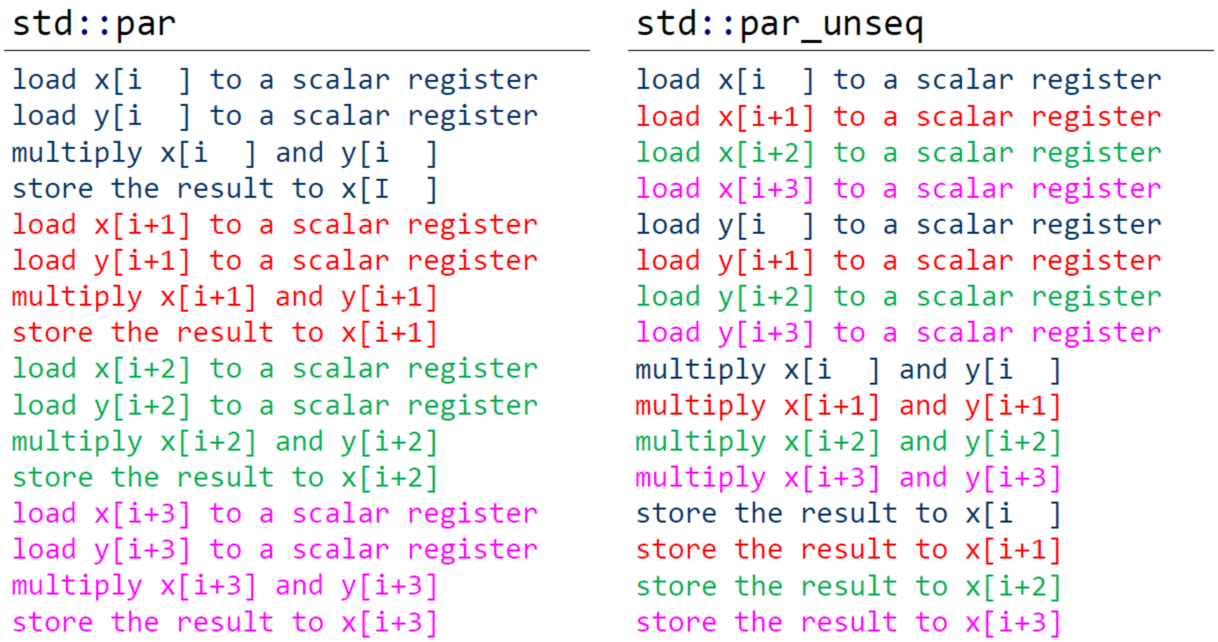
```
std::vector<T> x=....

//back then with OpenMp
#pragma omp parallel for simd
for(std::size_t i = 0; i<x.size();i++)
    someFunction(x[i]);

// now in C++17
std::for_each(std::execution::par_unseq, x.begin(), x.end(), someFunction)
```

- Unsequenced execution → functions can interleave

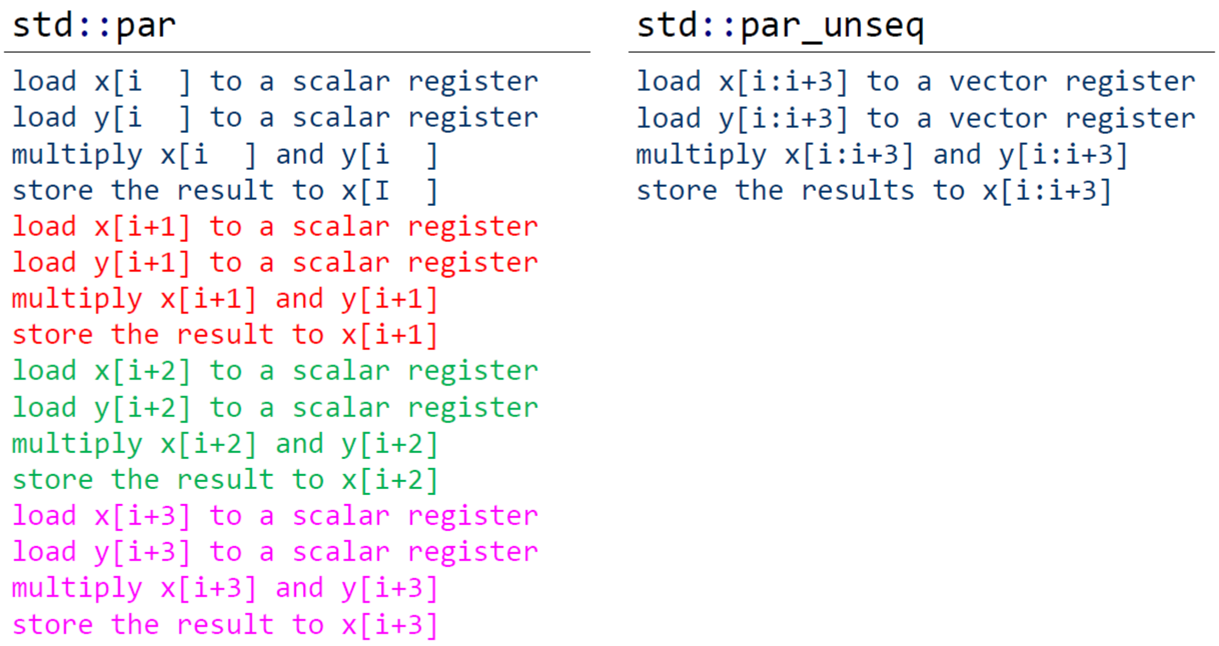
Std::execution::par_unseq



std::par	std::par_unseq
load x[i] to a scalar register	load x[i] to a scalar register
load y[i] to a scalar register	load x[i+1] to a scalar register
multiply x[i] and y[i]	load x[i+2] to a scalar register
store the result to x[I]	load x[i+3] to a scalar register
load x[i+1] to a scalar register	load y[i] to a scalar register
load y[i+1] to a scalar register	load y[i+1] to a scalar register
multiply x[i+1] and y[i+1]	load y[i+2] to a scalar register
store the result to x[i+1]	load y[i+3] to a scalar register
load x[i+2] to a scalar register	multiply x[i] and y[i]
load y[i+2] to a scalar register	multiply x[i+1] and y[i+1]
multiply x[i+2] and y[i+2]	multiply x[i+2] and y[i+2]
store the result to x[i+2]	multiply x[i+3] and y[i+3]
load x[i+3] to a scalar register	store the result to x[i]
load y[i+3] to a scalar register	store the result to x[i+1]
multiply x[i+3] and y[i+3]	store the result to x[i+2]
store the result to x[i+3]	store the result to x[i+3]

Source: Bryce Adelstein Lelbach | cppcon 2016, Bellevue Washington

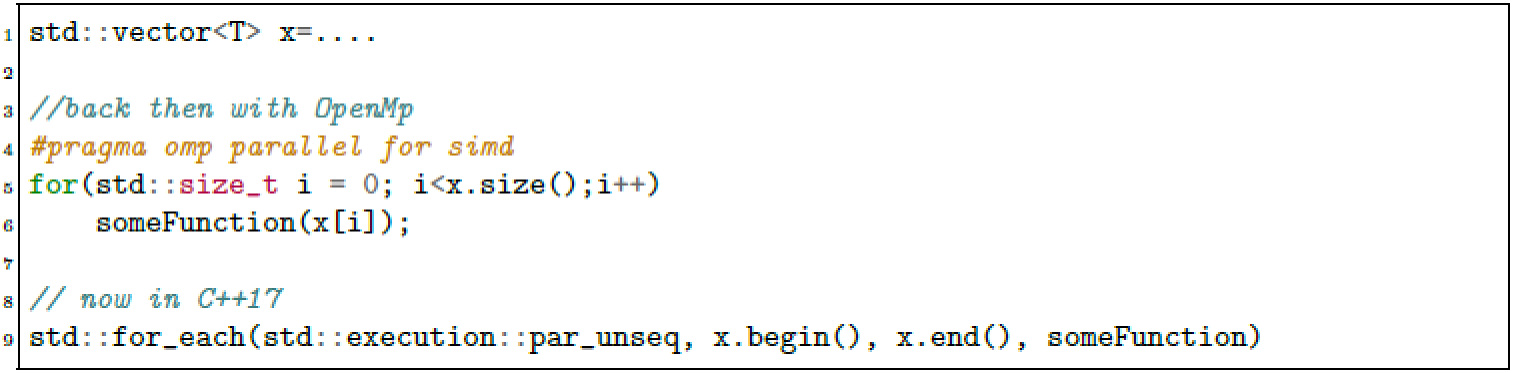
Std::execution::par_unseq



std::par	std::par_unseq
load x[i] to a scalar register	load x[i:i+3] to a vector register
load y[i] to a scalar register	load y[i:i+3] to a vector register
multiply x[i] and y[i]	multiply x[i:i+3] and y[i:i+3]
store the result to x[I]	store the results to x[i:i+3]
load x[i+1] to a scalar register	
load y[i+1] to a scalar register	
multiply x[i+1] and y[i+1]	
store the result to x[i+1]	
load x[i+2] to a scalar register	
load y[i+2] to a scalar register	
multiply x[i+2] and y[i+2]	
store the result to x[i+2]	
load x[i+3] to a scalar register	
load y[i+3] to a scalar register	
multiply x[i+3] and y[i+3]	
store the result to x[i+3]	

Source: Bryce Adelstein Lelbach | cppcon 2016, Bellevue Washington

parallel_unsequenced_policy



```
std::vector<T> x=....

//back then with OpenMp
#pragma omp parallel for simd
for(std::size_t i = 0; i<x.size();i++)
    someFunction(x[i]);

// now in C++17
std::for_each(std::execution::par_unseq, x.begin(), x.end(), someFunction)
```

- Unsequenced execution → functions can interleave
- Not allowed:
    - Allocation / Deallocation of memory
    - Acquiring mutex
    - Generally vectorization-unsafe operations

- Not every hardware does support SIMD

New STL Algorithms

Execution Policy signature:

```
std::algorithm_name(ExecutionPolicy&& policy, /* normal args... */);
```

New STL Algorithms

Execution Policy signature:

```
std::algorithm_name(ExecutionPolicy&& policy, /* normal args... */);
```

- overloads for 69 algorithms
- 8 new algorithms

New STL Algorithms (parallelized)

std::for_each | std::for_each_n

“Unordered” versions of “ordered” algorithms
- std::reduce
- std::inclusive_scan
- std::exclusive_scan

Fused Algorithms
- std::transform_reduce
- std::transform_inclusive_scan
- std::transform_exclusive_scan

std::for_each | std::for_each_n

std::for_each

```
template< class ExecutionPolicy, class ForwardIt, class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, UnaryFunction2 f );
```

std::for_each_n

```
template< class ExecutionPolicy, class ForwardIt, class Size, class UnaryFunction2 >
ForwardIt for_each_n( ExecutionPolicy&& policy, ForwardIt first, Size n, UnaryFunction2 f );
```

std::reduce – unordered std::accumulate

```
template<class ExecutionPolicy, class ForwardIt, class T, class BinaryOp>
T reduce(ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init, BinaryOp binary_op);
```

- Returns a generalized sum of a initial value and a sequence over a Binary operation

std::reduce – unordered std::accumulate

```
Vector<int> x={1,2,3,4};
int result = 0, result1 = 0, result2 = 0, init = 0;

int sum(int a,int b){return a+b;}
std::accumulate(x.begin(), x.end(),sum) == std::reduce(x.begin(), x.end(),sum) // true!
 ↪  (6==6)
//but we only know the execution order for std::accumulate, which will be
/*  result = init;
    result = result + x[0]; 1
    result = result + x[1]; 3
    result = result + x[2]; 7
    result = result + x[3]; 10  */
```

std::reduce – unordered std::accumulate

```
Vector<int> x={1,2,3,4};
int result = 0, result1 = 0, result2 = 0, init = 0;

int sum(int a,int b){return a+b;}
std::accumulate(x.begin(), x.end(),sum) == std::reduce(x.begin(), x.end(),sum) // true!
↪   (6==6)
//but we only know the execution order for std::accumulate, which will be
/*  result = init;
    result = result + x[0]; 1
    result = result + x[1]; 3
    result = result + x[2]; 7
    result = result + x[3]; 10  */
//while std::reduce has a random order as for example:
/*  result1 = x[3] + x[2]; 7
    result2 = x[0] + x[1]; 3
    result  = init + result1 + result2; 10  */
```

std::reduce – unordered std::accumulate

```
Vector<int> x={1,2,3,4};
int result = 0, result1 = 0, result2 = 0, init = 0;

int sum(int a,int b){return a+b;}
std::accumulate(x.begin(), x.end(),sum) == std::reduce(x.begin(), x.end(),sum) // true!
 ↪   (6==6)
//but we only know the execution order for std::accumulate, which will be
/*  result = init;
    result = result + x[0]; 1
    result = result + x[1]; 3
    result = result + x[2]; 7
    result = result + x[3]; 10  */
//while std::reduce has a random order as for example:
/*  result1 = x[3] + x[2]; 7
    result2 = x[0] + x[1]; 3
    result  = init + result1 + result2; 10  */
```

What would happen if we apply minus() as binary_op?

std::reduce – unordered std::accumulate

```
Vector<int> x={1,2,3,4};
int result = 0, result1 = 0, result2 = 0, init = 0;

int minus(int a,int b){return a-b;}
// Undetermined! (-10==??)
std::accumulate(x.begin(), x.end(),minus) == std::reduce(x.begin(), x.end(),minus)
//but we only know the execution order for std::accumulate, which will be
/*  result = init;
    result = result - x[0]; -1
    result = result - x[1]; -3
    result = result - x[2]; -7
    result = result - x[3]; -10  */
//while std::reduce has a random order as for example:
/*  result1 = x[3] - x[2]; 1
    result2 = init - x[0] - x[1]; -3
    result  = init + result1 + result2; -2  */
```

std::accumulate does not equal std::reduce for non-commutative and non-associative operations!

std::reduce – unordered std::accumulate

```
template<class ExecutionPolicy, class ForwardIt, class T, class BinaryOp>
T reduce(ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init, BinaryOp binary_op);
```

- Returns a generalized sum of a initial value and a sequence over a Binary operation

- Supports only commutative and associative operations.
  For example:
    - integer addition
    - integer multiplication

- Integer subtraction has non-deterministic behaviour
  because: x-y !=y-x → non-commutative

std::inclusive_scan – unordered std::partial_sum

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class BinaryOperation, class T >
ForwardIt2 inclusive_scan( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last,
ForwardIt2 d_first, BinaryOperation binary_op, T init );
```

```
(output)*   = init + first*;
(output+1) = init + first* + (first+1)*;
(output+2) = init + first* + (first+1)* + (first+2)*;
```

Unspecified grouping → Binary_op has to be associative!

std::exclusive_scan – unordered std::partial_sum

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class T, class BinaryOperation >
ForwardIt2 exclusive_scan( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last,
ForwardIt2 d_first, T init, BinaryOperation binary_op);
```

```
(output)*   = init; //n-th element is excluded
(output+1) = init + first*;
(output+2) = init + first* + (first+1)*;
```

n-th element is excluded

Unspecified grouping → Binary_op has to be associative!

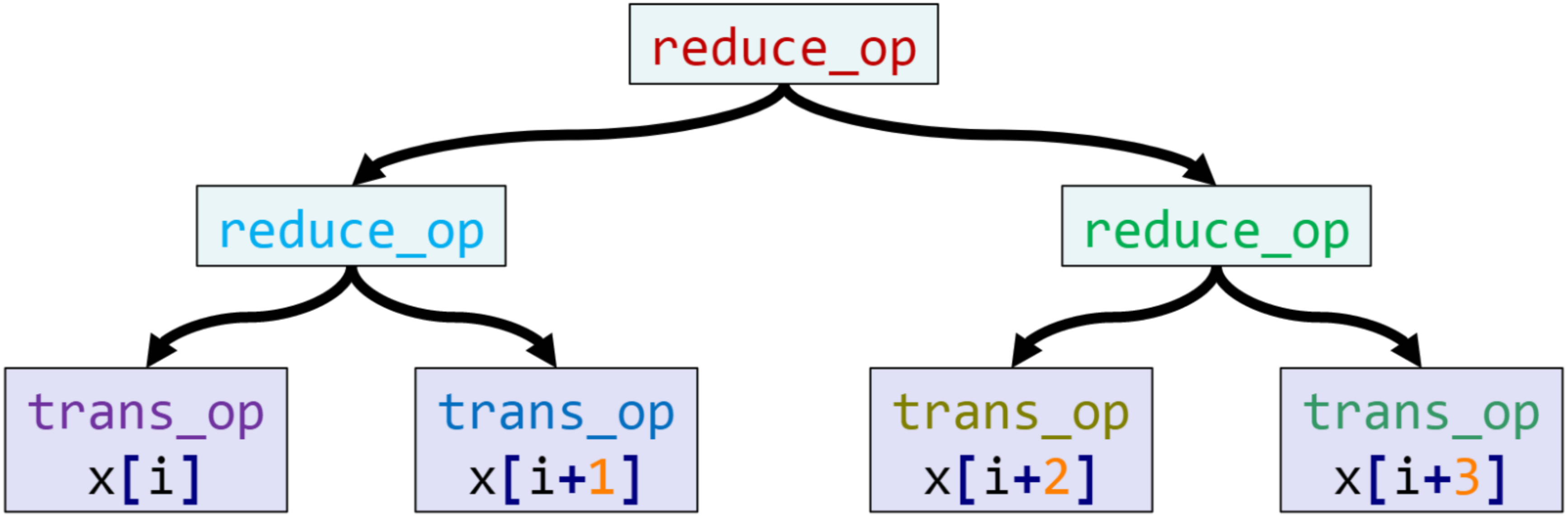
std::transform_reduce – unordered std::inner_product

```
template<class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class T,
class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy, ForwardIt1 first1, ForwardIt1 last1,
ForwardIt2 first2, T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2);
```

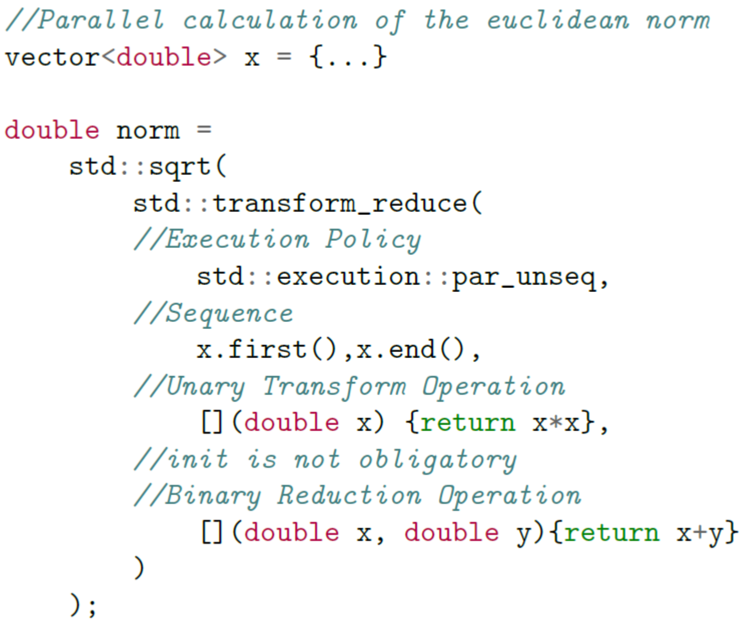
From Haskell known as map_reduce

1. applies *R trans_op(T const&)* on all sequence elements
2. reduces the sequence over *R reduce_op(R const&, R const&)*

std::transform_reduce – unordered std::inner_product



std::transform_reduce – unordered std::inner_product



```
//Parallel calculation of the euclidean norm
vector<double> x = {...}

double norm =
    std::sqrt(
        std::transform_reduce(
        //Execution Policy
            std::execution::par_unseq,
        //Sequence
            x.first(),x.end(),
        //Unary Transform Operation
            [](double x) {return x*x},
        //init is not obligatory
        //Binary Reduction Operation
            [](double x, double y){return x+y}
        )
    );
```

std::transform_inclusive_scan

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class BinaryOperation,
class UnaryOperation, class T >
ForwardIt2 transform_inclusive_scan( ExecutionPolicy&& policy, ForwardIt1 first,
ForwardIt1 last, ForwardIt2 d_first, BinaryOperation binary_op,
UnaryOperation unary_op, T init );
```

std::transform_exclusive_scan

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class T,
class BinaryOperation, class UnaryOperation >
ForwardIt2 transform_exclusive_scan( ExecutionPolicy&& policy, ForwardIt1 first,
ForwardIt1 last, ForwardIt2 d_first, T init, BinaryOperation binary_op,
UnaryOperation unary_op );
```

Important Notes

Element access functions

Exception Handling

Element access functions

Functions that are passed to the STL algorithms

Have to apply policy specific constraints:

```
std::vector<int> a ={...};
std::vector<int> b;
std::for_each(std::execution::seq, std::begin(a), std::end(a),[&](int i) {
        b.push_back(i);
});
```

Element access functions

Functions that are passed to the STL algorithms

Have to apply policy specific constraints:

```
std::vector<int> a ={...};
std::vector<int> b;
std::for_each(std::execution::seq, std::begin(a), std::end(a),[&](int i) {
		b.push_back(i);
});
```

Element access functions

```
std::vector<int> a ={...};
std::vector<int> b;
…
std::for_each(std::execution::par, std::begin(a), std::end(a),[&](int i) {
            b.push_back(i);
});
```

Element access functions

```
std::vector<int> a ={...};
std::vector<int> b;
…
//Error: data race because of parallel execution policy
std::for_each(std::execution::par, std::begin(a), std::end(a),[&](int i) {
                b.push_back(i);
});
```

Element access functions

```
std::vector<int> a ={...};
std::vector<int> b;
…
//No data race because vector gets locked before access
std::mutex m;
std::for_each(std::execution::par, std::begin(a), std::end(a),[&](int i) {
            m.lock();
            b.push_back(i);
            m.unlock();
});
```

Element access functions

```
std::vector<int> a ={...};
std::vector<int> b;
…
//No data race because vector gets locked before access
std::mutex m;
std::for_each(std::execution::par, std::begin(a), std::end(a),[&](int i) {
		m.lock();
		b.push_back(i);
		m.unlock();
});
```

Can we implement the same functionality with std::execution::par_unseq?

Q.:What happens if an Exception is thrown?

→std::terminate is called

→std::bad::alloc if out of memory

→compiler specific execution policies may define different behavior

Outlook

Dynamic Execution Policies

Executors

Parallel STL algorithms that return futures

Summary

C++11/14	low-level concurrency primitives
C++17	higher-level generic abstractions