EXTENDING STD::FUTURE

- Several proposals (draft technical specifications) for next C++ Standard
  - Extension for future<>
    - Compositional facilities
      - Parallel composition
      - Sequential composition
  - Parallel Algorithms
  - Parallel Task Regions
- Extended async semantics: dataflow

EXTENDING ASYNC: DATAFLOW

- What if one or more arguments to 'async' are futures themselves?
- Normal behavior: pass futures through to function
- Extended behavior: wait for futures to become ready before invoking the function:

```
template <typename F, typename… Arg>
future<typename result_of<F(Args…)>::type> dataflow(F&& f, Arg&&… arg);
```

- If ArgN is a future, then the invocation of F will be delayed
- Non-future arguments are passed through

Agenda

- Use the Proper abstractions?
- Why the rush to Massive Parallelism
- What Now?
- Hello World from C++11/14/17 Parallelism
- SYCL: C++ Heterogeneous (GPU) Programming
- Bonus: Executors
-

Concurrency and parallelism: They're not the same thing!

Concurrency

- Why: express component interactions for effective *program structure*
- How: interacting *threads* that can wait on events or each other

Parallelism

- Why: exploit *hardware* efficiently to scale *performance*
- How: independent *tasks* that can run simultaneously

A program can have both

From Pablo Halpern

Sports analogy

Concurrency

Parallelism

From Pablo Halpern

C++17 Parallel STL: Democratizing Parallelism in C++

What is Parallel STL?

Parallel STL greatly facilitates the usage of parallelism in C++ by exposing a parallel interface for the STL algorithms.

Why do I care?

Hardware architecture is becoming increasingly parallel. You cannot escape. See Herb Sutter The Free Lunch Is Over, which is ***now over 10 years old now!*** More updated version: Welcome to the jungle

What does it include?

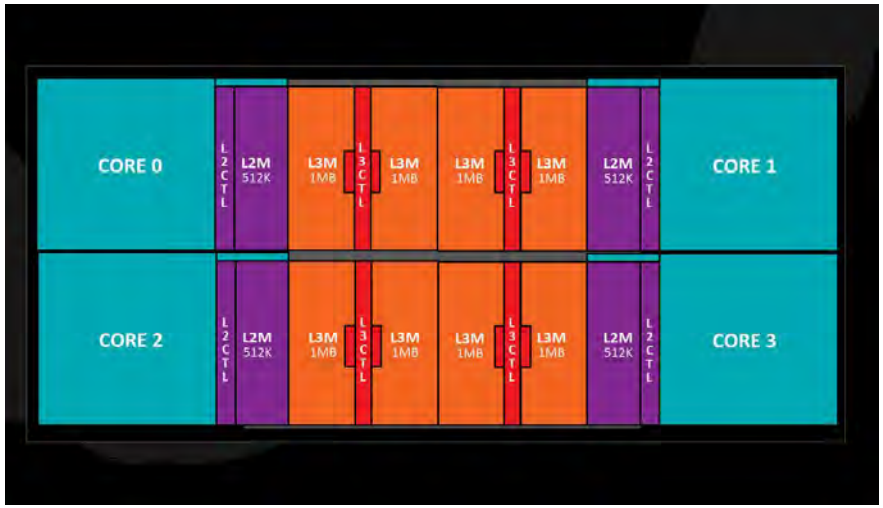
It adds wording for parallel execution on the C++ standard, and **Execution Policies** to the STL interface that enable selecting the appropriate level of parallelism.

New parallel algorithms are also added to the interface.

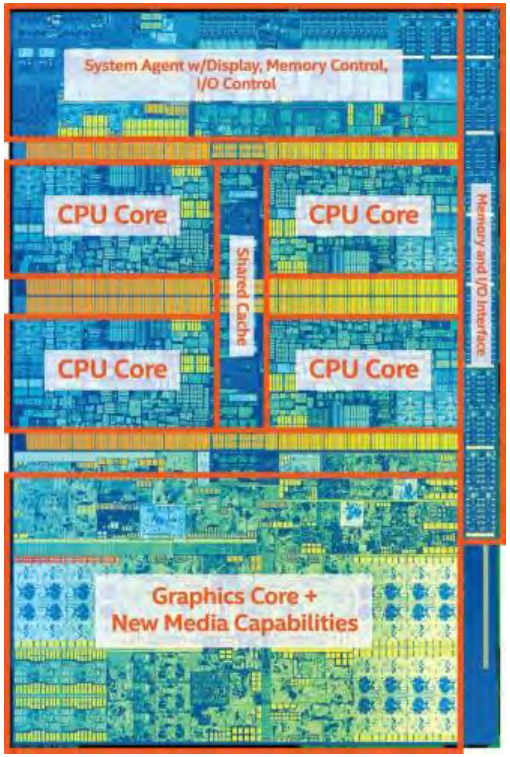
What do I take from this talk?

You will understand what Parallel STL is and learn the basic to use them. You'll be ready to use also the SYCL ParallelSTL on your accelerator.

Current “Desktop” technology



AMD Ryzen (4 cores/socket)



Intel Core i7 7th generation (4 cores + GPU / socket)

History

- Various libraries existed over the years:
  - AMD Bolt, NVIDIA Thrust, Microsoft C++ AMP algorithms...
- In 2012, two separate proposals for parallelism come to C++ standard:
  - NVIDIA (N3408) based on Thrust
  - Microsoft and Intel (N3429), based on Intel TBB and PPL/C++AMP

History

- Joint Proposal in 2013:
  - A Parallel Algorithms Library (n3554)

History

- Proposal evolved/matured for a couple of years
  - N3554, N3850, N3960, N4071, N4409...

History

- **Final proposal P0024R2 accepted for C++17 during Jacksonville**
- Many corrections and clarifications before C++17
- This is the life of a TS from birth to ratification

Sorting with the STL

```
std::vector<int> data = { 8, 9, 1, 4 };

std::sort(std::begin(data), std::end(data));

if (std::is_sorted(data)) {
    Std::cout << " Data is sorted!" << std::endl;
}
```

Normal sequential sort algorithm

```
std::vector<int> data = { 8, 9, 1, 4 };

std::sort(std::execution_policy::par,
          std::begin(data), std::end(data));

if (std::is_sorted(data)) {
    Std::cout << " Data is sorted!" << std::endl;
}
```

Extra parameter to STL algorithms enable parallelism

The Execution Policy: Standard policy classes

- Defined in the execution namespace
  - Sequenced policy
    - Never do parallel, sequenced in-order execution
    - `constexpr sequenced_policy sequenced;`
  - Parallel policy
    - Can use caller thread but may span others (std::thread)
    - Invocations do not interleave on a single thread
    - `constexpr sequenced_policy par;`
  - Parallel unsequenced
    - Can use caller thread or others (e.g std::thread)
    - Multiple invocations may be interleaved on a single thread
    - `constexpr sequenced_policy par_unseq;`

Many different existing implementations

Available today

- Microsoft: http://parallelstl.codeplex.com
- HPX: http://stellar-group.github.io/hpx/docs/html/hpx/manual/parallel.html
- HSA: http://www.hsafoundation.com/hsa-for-math-science
- Thibaut Lutz: http://github.com/t-lutz/ParallelSTL
- NVIDIA: https://thrust.github.io/doc/group__execution__policies.html
- Codeplay: http://github.com/KhronosGroup/SyclParallelSTL
- Clang: Not yet available

Expect major C++ compilers to implement it soon!

Using execution policies

```
using std::execution_policy;


// May execute in parallel
std::sort(par, std::begin(data), std::end(data))
// May be parallelized and vectorized
std::sort(std::par_unseq, std::begin(data), std::end(data));
// Will not be parallelized/vectorized
std::sort(std::sequenced, std::begin(data), std::end(data));
// Vendor-specific policy, read their documentation!
std::sort(custom_vendor_policy, std::begin(data), std::end(data));
```

Propagating the policy to the end user

```
using std::execution_policy;

template<typename Policy, typename Iterator>
void library_function(Policy p, Iterator begin,
                        Iterator end) {
   std::sort(p, begin, end);
   std::for_each(p, begin, end,
                  [&](Iterator::value_type e&) { e ++;}) ;
   std::for_each(std::sequenced, begin, end,
                  non_parallel_operation) ;
}
```

Parallel overloads available

Table 1 — Table of parallel algorithms

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			

New algorithms into the STL: Parallel For Each

```
template<class ExecutionPolicy, class InputIterator, class Function>
void for_each(ExecutionPolicy && exec,InputIterator first, InputIterator
last, Function f);

template<class ExecutionPolicy, class InputIterator, class Size, class
Function>
InputIterator for_each_n(ExecutionPolicy && exec,
                         InputIterator first, Size n,
                         Function f) ;

template<class InputIterator, class Size, class Function>
InputIterator for_each_n(InputIterator first, Size n, Function f);
```

- **for_each**: Applies f to elements in range [first, last).
- **for_each_n**: Applies f to elements in [first, first + n)

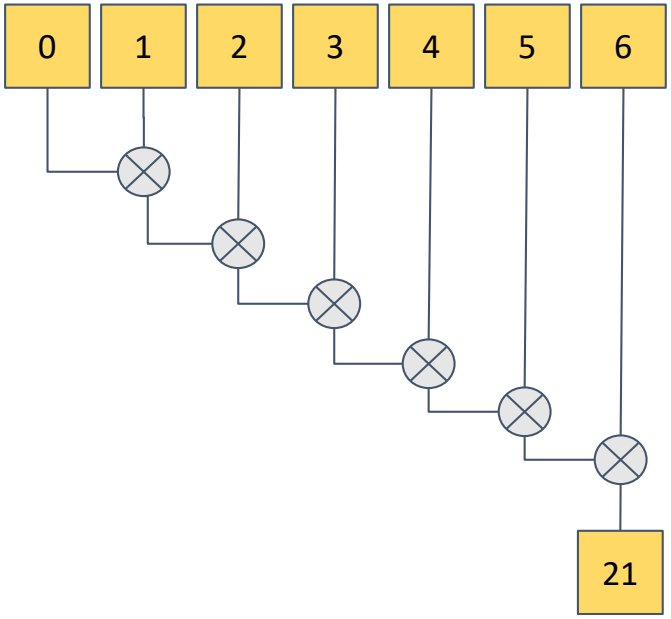
New algorithms into the STL

Numerical Parallel Algorithms

```
template < class InputIterator >
typename iterator_traits < InputIterator >:: value_type
reduce ( InputIterator first , InputIterator last ) ;

template < class InputIterator , class T >
T reduce ( InputIterator first , InputIterator last , T init ) ;

template < class InputIterator , class T , class BinaryOperation >
T reduce ( InputIterator first , InputIterator last , T init ,
BinaryOperation binary_op ) ;
```

Implements a reduction operation (the order of the binary_op is not relevant). The sequential equivalent is accumulate

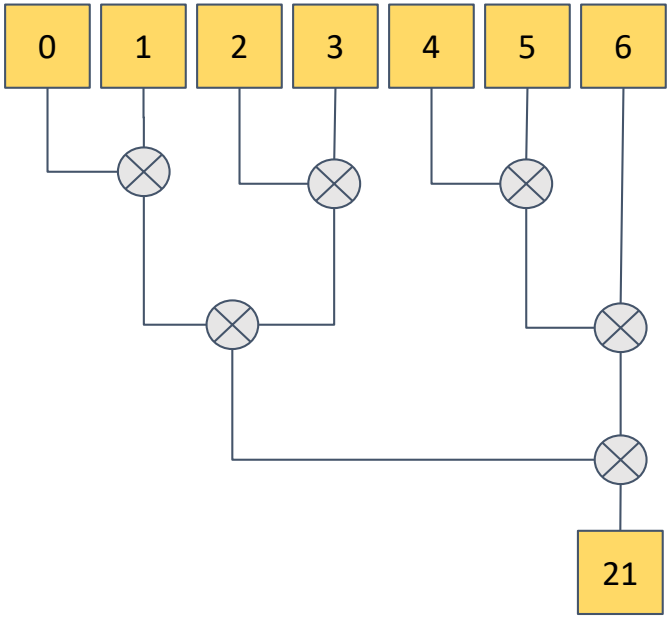
New algorithms into the STL (Serial Reduction pattern)




```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::accumulate(std::begin(v1), nElems, 1);
```

Only one core is used for the different additions.

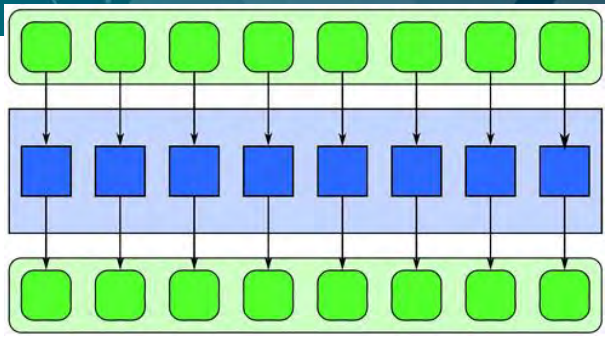
New algorithms into the STL (Parallel Reduction Pattern)




```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::reduce(std::execution_policy::par,
                std::begin(v1), nElems, 1);
```

If operation is commutative and associative, can be run in parallel. Reduction uses all cores!

Transform



```
std::transform(std::execution::par,
               v1.begin(), v1.end(),
               v2.begin(), output.begin(),
               [=](int val1, int val2)
                  { return val1 + val2 + 1; });
```

- transform (a.k.a map) applies a function to an input range and stores the result on the output range. Operation is out of order.

Transform reduce

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

(4) (since C++17)

```
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);
```

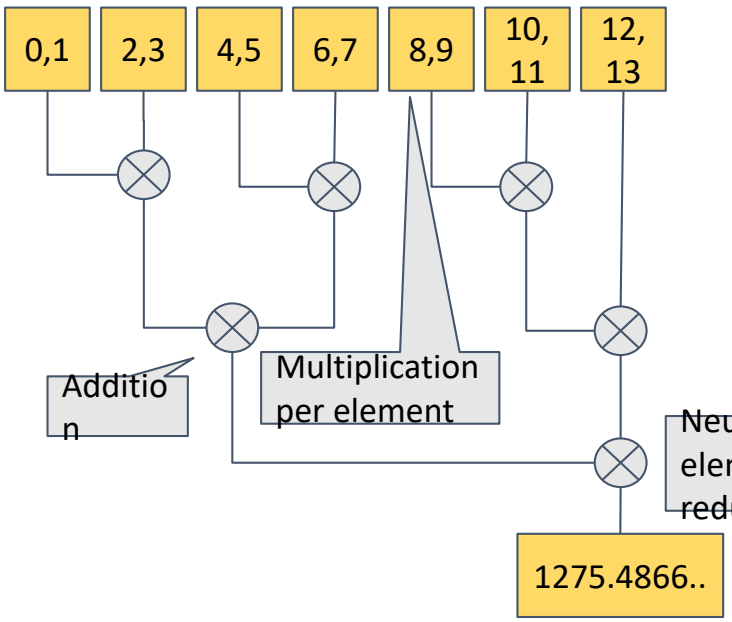
(5) (since C++17)

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

(6) (since C++17)

- transform_reduce applies a function to an input range and then applies the binary operation to reduce the values

Transform Reduce example



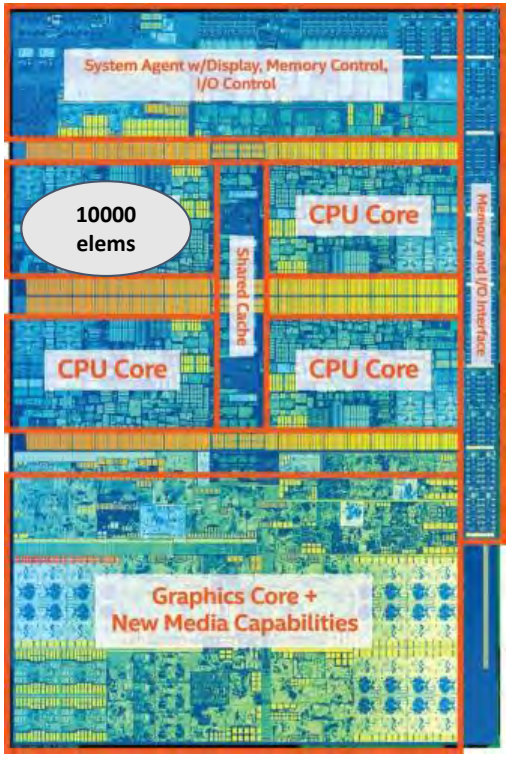

```
struct elem {
float a;
float b;
} elem;

auto transformReduce = [&]() {
   float pi = PI;
   float res = std::experimental::parallel::transform_reduce(
       snp, std::begin(v), std::end(v), [=](elem x) { return x.a * x.b * pi; },
       0,                                                  // map multiplication
       [=](float a, float b) { return a + b; });  // reduce addition
}
```

Apply to each element of v

Neutral element of reduction

Reduction function: addition

What can I do with a Parallel For Each?

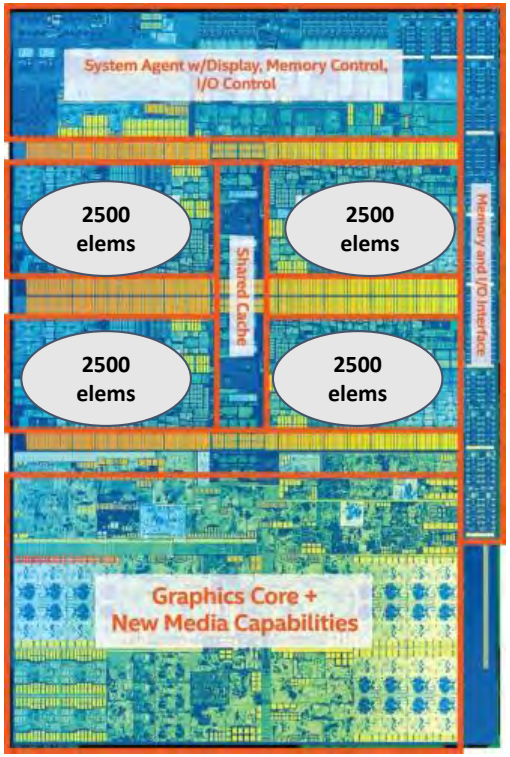


Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::begin(v1), nElems, 1);

std::for_each(std::begin(v), std::end(v),
                    [=](float f) { f * f + f });
```

Traditional for each uses only one core, rest of the die is unutilized!

What can I do with a Parallel For Each?



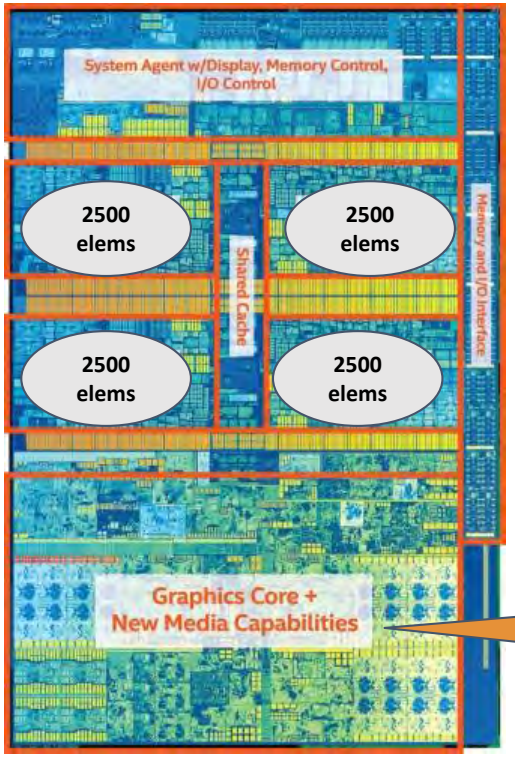
Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::execution_policy::par,
            std::begin(v1), nElems, 1);

std::for_each(std::execution_policy::par,
              std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?



Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::execution_policy::par,
            std::begin(v1), nElems, 1);

std::for_each(std::execution_policy::par,
              std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

What about this part?

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)