Implementing the C++ std algorithms library in Kokkos: an overview of the main challenges, API differences and some implementation details

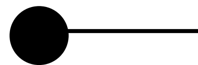
Francesco Rizzi

(NexGen Analytics)

Kokkos cafe @ CEA (Feb. 2024)

C++ std algorithms: brief timeline

The Standard Template Library (Stepanov)



October 1995

Execution policy

C++17

Ranges

C++20

The Standard Template Library

Alexander Stepanov

Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mt. View, CA 94043
stepanov@mti.sgi.com

Meng Lee

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
lee@hpl.hp.com

October 31, 1995

C++ algorithms: what are they?

More than 100 free functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on *ranges* of elements.

A range is defined (before C++20) as `[first, last)` where last refers to the element past the last element to inspect or modify.

Constrained algorithms and algorithms on ranges (C++20)

Constrained algorithms, e.g. `ranges::copy`, `ranges::sort`, ...

Execution policies (C++17)

is_execution_policy (C++17)

execution::seq (C++17)
execution::par (C++17)
execution::par_unseq (C++17)
execution::unseq (C++20)

execution::sequenced_policy (C++17)
execution::parallel_policy (C++17)
execution::parallel_unsequenced_policy (C++17)
execution::parallel_unsequenced (C++20)

Non-modifying sequence operations

Batch operations

for_each
for_each_n (C++17)

Search operations

all_of (C++11)
any_of (C++11)
none_of (C++11)
count
count_if
mismatch
equal

find
find_if
find_if_not (C++11)
find_end
find_first_of
adjacent_find
search
search_n

Modifying sequence operations

Copy operations

copy
copy_if (C++11)
copy_backward

copy_n (C++11)
move (C++11)
move_backward (C++11)

Swap operations

swap
iter_swap

swap_ranges

Transformation operations

replace
replace_if
transform

replace_copy
replace_copy_if

Generation operations

fill
fill_n

generate
generate_n

Removing operations

remove
remove_if
unique

remove_copy
remove_copy_if
unique_copy

Order-changing operations

reverse
reverse_copy
rotate
rotate_copy

random_shuffle (until C++17)
shuffle (C++11)
shift_left (C++20)
shift_right (C++20)

Sampling operations

sample (C++17)

Sorting and related operations

Partitioning operations

partition
partition_copy (C++11)
stable_partition

is_partitioned (C++11)
partition_point (C++11)

Sorting operations

sort
stable_sort
partial_sort
partial_sort_copy

is_sorted (C++11)
is_sorted_until (C++11)
nth_element

Binary search operations (on partitioned ranges)

lower_bound
upper_bound

equal_range
binary_search

Set operations (on sorted ranges)

includes
set_union
set_intersection

set_difference
set_symmetric_difference

Merge operations (on sorted ranges)

merge

inplace_merge

Heap operations

push_heap
pop_heap
make_heap

sort_heap
is_heap (C++11)
is_heap_until (C++11)

Minimum/maximum operations

max
min
minmax (C++11)
clamp (C++17)

max_element
min_element
minmax_element (C++11)

Lexicographical comparison operations

lexicographical_compare
lexicographical_compare_three_way (C++20)

Permutation operations

next_permutation
prev_permutation

is_permutation (C++11)

C library

qsort

bsearch

Numeric operations

iota (C++11)
inner_product
adjacent_difference

accumulate
reduce (C++17)
transform_reduce (C++17)

partial_sum
inclusive_scan (C++17)
exclusive_scan (C++17)

transform_inclusive_scan (C++17)
transform_exclusive_scan (C++17)

Operations on uninitialized memory

uninitialized_copy
uninitialized_move (C++17)
uninitialized_fill

uninitialized_copy_n (C++11)
uninitialized_move_n (C++17)
uninitialized_fill_n

destroy (C++17)
destroy_n (C++17)
destroy_at (C++17)
construct_at (C++20)

uninitialized_default_construct (C++17)
uninitialized_value_construct (C++17)
uninitialized_default_construct_n (C++17)
uninitialized_value_construct_n (C++17)

https://en.cppreference.com/w/cpp/algorithm

Why are std algos important? Why you should use them?

- Correctness
  - They are reliable and solid, less error-prone than writing your own
- No raw loops (Sean Parent)
  - https://www.youtube.com/watch?v=W2tWOdzgXHA
- Expressive/readable code, DRY principle
- Maintain the same "level of abstraction" (SLAP principle)
  - *Mixing different levels of abstraction in one same method can make it harder to read and understand. We should always try to keep the code inside our method at the same level of abstraction. (Uncle Bob, 2009)*

C++ STL Algorithms API

```
1 template<class InputIt, class UnaryFunction>
2 UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
3
4 template<class ExecutionPolicy, class ForwardIt, class UnaryFunction2>
5 void for_each( ExecutionPolicy&& policy,
6                ForwardIt first, ForwardIt last, UnaryFunction2 f );
```

- `[first, last)`: iterators defining a range
  - iterators' category depends on the algorithm
- `f`: unary function that is applied to the result of dereferencing each iterator in the given range (this is an example of a "strategy design pattern")
- `policy`: more on this next

Execution policy is important

Defined in header `<execution>`

`class sequenced_policy { /* unspecified */ };`	(1)	(since C++17)
`class parallel_policy { /* unspecified */ };`	(2)	(since C++17)
`class parallel_unsequenced_policy { /* unspecified */ };`	(3)	(since C++17)
`class unsequenced_policy { /* unspecified */ };`	(4)	(since C++20)

1) The execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized. The invocations of element access functions in parallel algorithms invoked with this policy (usually specified as `std::execution::seq`) are indeterminately sequenced in the calling thread.

2) The execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized. The invocations of element access functions in parallel algorithms invoked with this policy (usually specified as `std::execution::par`) are permitted to execute in either the invoking thread or in a thread implicitly created by the library to support parallel algorithm execution. Any such invocations executing in the same thread are indeterminately sequenced with respect to each other.

3) The execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized, vectorized, or migrated across threads (such as by a parent-stealing scheduler). The invocations of element access functions in parallel algorithms invoked with this policy are permitted to execute in an unordered fashion in unspecified threads, and unsequenced with respect to one another within each thread.

4) The execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items.

During the execution of a parallel algorithm with any of these execution policies, if the invocation of an element access function exits via an uncaught exception, `std::terminate` is called, but the implementations may define additional execution policies that handle exceptions differently.

How does it relate to kokkos?

Kokkos implementation of a (large, eventually growing) selection of std algorithms accepting Kokkos rank-1 `Views` or iterators.

- Header: `Kokkos_StdAlgorithms.hpp`
- Inside the `Kokkos::Experimental`
- **v3.6**: introduced API accepting execution policy instance
- **v4.2**: extended API for team-level support

- Documentation is available in the Kokkos wiki: `https://github.com/kokkos/kokkos/wiki`

	Currently Supported in Kokkos
Minimum/maximum ops	`min_element`, `max_element`, `minmax_element`
ModifyingSequence ops	`fill`, `fill_n`, `replace`, `replace_if`, `replace_copy`, `replace_copy_if`, `copy`, `copy_n`, `copy_backward`, `copy_if`, `generate`, `generate_n`, `transform`, `reverse`, `reverse_copy`, `move`, `move_backward`, `swap_ranges`, `unique`, `unique_copy`, `rotate`, `rotate_copy`, `remove`, `remove_if`, `remove_copy`, `remove_copy_if`, `shift_left`, `shift_right`
NonModifyingSequence ops	`find`, `find_if`, `find_if_not`, `for_each`, `for_each_n`, `mismatch`, `equal`, `count_if`, `count`, `all_of`, `any_of`, `none_of`, `adjacent_find`, `lexicographical_compare`, `search`, `search_n`, `find_first_of`, `find_end`
Numeric ops	`adjacent_difference`, `reduce`, `transform_reduce`, `exclusive_scan`, `transform_exclusive_scan`, `inclusive_scan`, `transform_inclusive_scan`
Partitioning ops	`is_partitioned`, `partition_copy`, `partition_point`
Sorting ops	`is_sorted_until`, `is_sorted`

Kokkos API: accepting Views

```
1 template <class ExSpaceT, ...>
2 ret_type algo_name(const ExSpaceT& space, view(s), extra);
3
4 template <class ExSpaceT, ...>
5 ret_type algo_name(const std::string& label, const ExSpaceT& space, view(s), extra);
6
7 template <class TeamHandleT, ...>
8 KOKKOS_FUNCTION
9 ret_type algo_name(const TeamHandleT& teamHandle, view(s), extra);
```

- `space`: exec space instance
- `teamHandle`: handle given inside a parallel region when using a TeamPolicy
- `label`: passed to the implementation kernels for debugging
  For overload on line 2, defaults to "Kokkos::algo_name_view_api_default"
- view(s): rank-1, `LayoutLeft`, `LayoutRight`, `LayoutStride`;
  must be accessible from `space` or from the space associated with `teamHandle`
- extra: parameters that are specific to the algorithm

Kokkos API: accepting iterators

```
1 template <class ExSpaceT, ...>
2 ret_type algo_name(const ExSpaceT& space, iterators, extra);
3
4 template <class ExSpaceT, ...>
5 ret_type algo_name(const std::string& label, const ExSpaceT& space, iterators, extra);
6
7 template <class TeamHandleT, ...>
8 KOKKOS_FUNCTION
9 ret_type algo_name(const TeamHandleT& teamHandle, iterators, extra);
```

- `space`, `teamHandle`, `extra`: same as before
- `iterators`:
  - must be **random access iterators**
  - preferably use `Kokkos::Experimental::begin,end,cbegin,cend` (coming up)
  - must be accessible from `space` or from the exec space of `teamHandle`

Kokkos random-access iterators

`Kokkos::Experimental::{begin, cbegin, end, cend}`

Declaration:

```
template <class DataType, class... Properties>
KOKKOS_INLINE_FUNCTION
auto begin(const Kokkos::View<DataType, Properties...>& view);
```

- `view`: must be rank-1 with `LayoutLeft`, `LayoutRight`, or `LayoutStride`.
- Dereferencing iterators must be done in an execution space where ‘view‘ is accessible.

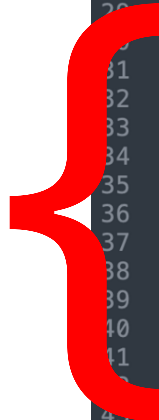
```
Kokkos::Experimental::distance(first, last);
Kokkos::Experimental::iter_swap(it1, it2);
```

General comments

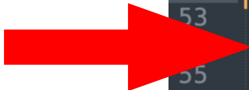
- Kokkos API accepts both random access iterators and Views directly. This is similar to C++ algorithms operating on ranges (C++20).
- The Kokkos algorithms semantically "correspond" to the C++ std algorithms using `std::execution::parallel_unsequenced_policy`
- Implemented in terms of Kokkos `parallel_{for, reduce, scan}`.
- Debug mode enables several checks, e.g.: whether iterators identify a valid range, the execution space accessibility, etc., and error messages printed.
- Currently, algorithms fence directly the execution space instance or call the team barrier for the team handle. This kinds of contradicts the Kokkos semantics and discussions are ongoing to fix this to make them potentially non-blocking.

`for_each`: Kokkos Implementation

functor



arallel dispatch



```cpp
#ifndef KOKKOS_STD_ALGORITHMS_FOR_EACH_IMPL_HPP
#define KOKKOS_STD_ALGORITHMS_FOR_EACH_IMPL_HPP

#include <Kokkos_Core.hpp>
#include "Kokkos_Constraints.hpp"
#include "Kokkos_HelperPredicates.hpp"
#include <std_algorithms/Kokkos_Distance.hpp>
#include <string>

namespace Kokkos {
namespace Experimental {
namespace Impl {

template <class IteratorType, class UnaryFunctorType>
struct StdForEachFunctor {
  using index_type = typename IteratorType::difference_type;
  IteratorType m_first;
  UnaryFunctorType m_functor;

  KOKKOS_FUNCTION
  void operator()(index_type i) const { m_functor(m_first[i]); }

  KOKKOS_FUNCTION
  StdForEachFunctor(IteratorType _first, UnaryFunctorType _functor)
      : m_first(std::move(_first)), m_functor(std::move(_functor)) {}
};

template <class HandleType, class IteratorType, class UnaryFunctorType>
UnaryFunctorType for_each_exespace_impl(const std::string& label,
                                        const HandleType& handle,
                                        IteratorType first, IteratorType last,
                                        UnaryFunctorType functor) {
  Impl::static_assert_random_access_and_accessible(handle, first);
  Impl::expect_valid_range(first, last);

  const auto num_elements = Kokkos::Experimental::distance(first, last);
  ::Kokkos::parallel_for(
      label, RangePolicy<HandleType>(handle, 0, num_elements),
      StdForEachFunctor<IteratorType, UnaryFunctorType>(first, functor));
  handle.fence("Kokkos::for_each: fence after operation");

  return functor;
}
```

copy_if: Kokkos Implem

```
template <class ExecutionSpace, class InputIterator, class OutputIterator,
          class PredicateType>
OutputIterator copy_if_exespace_impl(const std::string& label,
                                     const ExecutionSpace& ex,
                                     InputIterator first, InputIterator last,
                                     OutputIterator d_first,
                                     PredicateType pred) {
  /*
    To explain the impl, suppose that our data is:

    | 1 | 1 | 2 | 2 | 3 | -2 | 4 | 4 | 4 | 5 | 7 | -10 |

    and we want to copy only the even entries,
    We can use an exclusive scan where the "update"
    is incremented only for the elements that satisfy the predicate.
    This way, the update allows us to track where in the destination
    we need to copy the elements:

    In this case, counting only the even entries, the exlusive scan
    during the final pass would yield:

    | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 6 | 6 |
              *   *       *   *   *   *           *

    which provides the indexing in the destination where
    each starred (*) element needs to be copied to since
    the starred elements are those that satisfy the predicate.
   */

  // checks
  Impl::static_assert_random_access_and_accessible(ex, first, d_first);
  Impl::static_assert_iterators_have_matching_difference_type(first, d_first);
  Impl::expect_valid_range(first, last);

  if (first == last) {
    return d_first;
  } else {
    // run
    const auto num_elements = Kokkos::Experimental::distance(first, last);

    typename InputIterator::difference_type count = 0;
    ::Kokkos::parallel_scan(label,
                            RangePolicy<ExecutionSpace>(ex, 0, num_elements),
                            // use CTAD
                            StdCopyIfFunctor(first, d_first, pred), count);

    // fence not needed because of the scan accumulating into count
    return d_first + count;
  }
}
```

```
template <class FirstFrom, class FirstDest, class PredType>
struct StdCopyIfFunctor {
  using index_type = typename FirstFrom::difference_type;

  FirstFrom m_first_from;
  FirstDest m_first_dest;
  PredType m_pred;

  KOKKOS_FUNCTION
  StdCopyIfFunctor(FirstFrom first_from,
                   FirstDest first_dest,
                   PredType pred)
      : m_first_from(std::move(first_from)),
        m_first_dest(std::move(first_dest)),
        m_pred(std::move(pred)) {}

  KOKKOS_FUNCTION
  void operator()(const index_type i, index_type& update,
                  const bool final_pass) const {
    const auto& myval = m_first_from[i];
    if (final_pass) {
      if (m_pred(myval)) {
        m_first_dest[update] = myval;
      }
    }

    if (m_pred(myval)) {
      update += 1;
    }
  }
};
```

Example usage

```
1  namespace KE = Kokkos::Experimental;
2
3  Kokkos::View<double*, Kokkos::HostSpace> myView("myView", 13);
4  // fill myView somehow
5
6  const double oldVal{2}, newVal{34};
7  auto defHostSpace = Kokkos::DefaultHostExecutionSpace();
8
9  // act on the entire view
10 KE::replace(defHostSpace, myView, oldVal, newVal);
11
12 // act on just a subset
13 auto startAt = KE::begin(myView) + 4;
14 auto endAt   = KE::begin(myView) + 10;
15 KE::replace(defHostSpace, startAt, endAt, oldVal, newVal);
16
17 // pass label and execution space (assumed enabled)
18 KE::replace("mylabel", Kokkos::OpenMP(), myView, oldVal, newVal);
```

Example usage

```cpp
template <class ValueType1, class ValueType2 = ValueType1>
struct CustomLessThanComparator {
  KOKKOS_INLINE_FUNCTION
  bool operator()(const ValueType1& a, const ValueType2& b) const
  {
    // return true if a is less than b, according to your custom logic
  }
};

int main(){
  // ...
  namespace KE = Kokkos::Experimental;
  Kokkos::View<double*, Kokkos::CudaSpace> myView("myView", 13);
  // fill a somehow
  auto res = KE::min_element(Kokkos::Cuda(), myView,
                             CustomLessThanComparator<double>());
  //...
}
```

Example usage

```
1 template <class ValueType>
2 struct GreaterThanValueFunctor {
3   ValueType m_val;
4   KOKKOS_INLINE_FUNCTION GreaterThanValueFunctor(ValueType val) : m_val(val) {}
5   KOKKOS_INLINE_FUNCTION bool operator()(ValueType v) const { return (v > m_val); }
6 };
7
8 template <class ViewType, class ValueType>
9 struct TestFunctor {
10   ViewType m_view; ValueType m_threshold; ValueType m_newVal;
11   // ...
12
13   template <class MemberType>
14   KOKKOS_INLINE_FUNCTION void operator()(const MemberType& member) const {
15     const auto myRowIndex = member.league_rank();
16     auto myRowSubView     = Kokkos::subview(m_view, myRowIndex, Kokkos::ALL());
17     GreaterThanValueFunctor predicate(m_threshold);
18     Kokkos::Experimental::replace_if(member, myRowSubView, predicate, m_newVal);
19   }
20 };
21
22 int main(){
23   // ...
24   Kokkos::View<int**> v("v", Nr, Nc); // # rows(Nr), # cols(Nc), filled somehow
25   const int threshold(151), newVal(1);
26   Kokkos::TeamPolicy<Kokkos::DefaultExecutionSpace> policy(Nr, Kokkos::AUTO());
27   Kokkos::parallel_for(policy, TestFunctor(v, threshold, newVal));
28   // ...
29 }
```

Conclusions

- Performance optimizations; keep consistency with C++ standard
- Support more algorithms (provide feedback in the survey document please)
- Kokkos "ranges" and interoperability with algorithms
  - https://github.com/fnrizzi/kokkos-tiny-ranges (fork it, contribute!)

```
Kokkos::View<int*> view("v", 1000);

auto p = view | Kokkos::nonlazy_filter(IsEven()) | Kokkos::reverse() | Kokkos::take(10);

Kokkos::parallel_for(p.size(), MyFunc(p));
```

  - Disclaimer: NOT official Kokkos work (yet), WIP but already works
  - Enabling interoperability with current algorithms API *should* be relatively smooth

Thank you

francesco.rizzi@ng-analytics.com