INTEGRATING SIMD WITH PARALLEL ALGORITHMS

ABSTRACT

This paper discusses a new execution policy for integrating SIMD with parallel algorithms.

CONTENTS

1 Changelog 1
2 Straw Polls 1
3 Introduction 2
4 Parallel Algorithms 2
A Bibliography 13
1

CHANGEOLOG

1.1 changes from revision 0

Previous revision: [P0350R0]

• Update to apply against C++17 wording.

• Removed executors discussion because the executors design has not left SG1 yet.

• Updated example code to reflect changes in P0214.

1.2 changes from revision 1

Previous revision: [P0350R1]

• Updated code to match [N4744].

• Fixed a bug in the for_each example implementation.

• Improved iota and for_each example implementations with constexpr-if.

• Discuss impact on all algorithms.

1.3 changes from revision 2

Previous revision: [P0350R2]

• Discuss ABI tag of std::generate callables.

• Add a Tony Table.

• Note that remove and remove_copy are implicitly vectorizable.

2

STRAW POLLS

2.1 sg1 at oulu

Poll: Ship it to LEWG?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
2.2 lewg at albuquerque

Poll: Forward the paper to LWG?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

→ Paper needs a revision: LEWG wants a list of affected algorithms and an update to concept requirements.

3 INTRODUCTION

Parallel Algorithms enable implementations of the existing STL algorithms to use non-sequential semantics when executing the user-supplied code (explicit callable or implicit operator call). The first argument to the algorithm function determines this change in execution semantics via an execution policy. This paper introduces a new execution policy, called execution::simd. execution::simd requires user-provided function objects to be callable with simd<T, Abi> arguments instead of the T arguments the std::execution::seq variant would use. The algorithm therefore processes chunks of simd<T, Abi>::size() objects concurrently. The execution order of the chunks retains the sequential semantics of the non-parallel algorithms.

As a consequence, the applicability of the execution policy is limited to iterators where Iterator::value_type is a vectorizable type [N4744, parallel simd.general]]. A future extension of simd may lift this restriction by allowing certain (or all) user-defined types as first template argument to simd. A different conceivable extension is a recursive destructuring applied inside the algorithm, subsequent creation of a corresponding number of simd objects, and a call to the function object with a corresponding number of arguments. (E.g. application of an algorithm on std::vector<std::pair<float, float>> calls the function object with simd<float>, simd<float> instead of simd<std::pair<float, float>>.)

4 PARALLEL ALGORITHMS

4.1 example

Consider the example in Listing 1. The iota and for_each functions each could create an internal simd iterator adaptor, depending on the iterator category. Being able to determine whether the storage, the iterator points to, is contiguous, is most important in this context as it enables vector loads and stores. Since the std::vector

1 An alternative suggestion for the name is execution::simd_type.
Table 1: Tony Table

<table>
<thead>
<tr>
<th>before</th>
<th>after (with optimized epilogue)</th>
</tr>
</thead>
</table>

**Using** V = stdx::native_simd<float>;  
constexpr int N = 60;  

**Template** <class T> T something(T);  

auto f(const std::array<float, N>& data)  
{
  std::array<float, N> output;
  size_t i = 0;
  for (; i + V::size() <= N; i += V::size()) {
    V x(data[i], stdx::element_aligned);
    x = something(x + 1);
    x.copy_to(&output[i], stdx::element_aligned);
  }
  for (; i < N; ++i) {
    output[i] = something(data[i] + 1);
  }
  return output;
}

**Using** V = stdx::native_simd<float>;  
constexpr int N = 60;  

**Template** <class T> T something(T);  

auto f(const std::array<float, N>& data)  
{
  std::array<float, N> output;
  stdx::transform(std::execution::simd,  
data.begin(), data.end(), output.begin(),  
[](auto x) {
    return something(x + 1);
  });
  return output;
}

Listing 1: Example using execution::simd with iota and for_each.

1. std::vector<float> data;
2. data.resize(99);
3. iota(execution::simd, data.begin(), data.end(), 0.f);
4. for_each(execution::simd, data.begin(), data.end(), [](auto &x) {
  x *= x;
});

Iterators are contiguous iterators, the example implementations shown in Listing 2 and Listing 3 could be used for the example.

Both implementations might be improved with a prologue that enables aligned loads and stores. Also note that for_each allows the Function parameter to mutate the argument if the iterator is a mutable iterator. The implementation uses a compile-time trait to determine whether the function f uses a reference parameter, in which case it stores the temporary simd object back. Otherwise, the store is optimized away.

Figure 1 shows a visualization how the iota implementation works. The init simd object is stored via vector stores to 4 (assuming native simd::size() == 4) elements in the std::vector. In each iteration the init object is incremented by simd::size() and stored to the following elements in the std::vector. Since the std::vector has 99 elements, the last three elements cannot be initialized with a vector store of four.
```cpp
template <size_t N, class ContiguousIterator>
inline void epilogue(ContiguousIterator first, ContiguousIterator last,
                      typename ContiguousIterator::value_type first_value) {
  if constexpr (N > 0) {
    if (distance(first, last) >= N) {
      using T = ContiguousIterator::value_type;
      using V = simd<T, simd_abi::deduce_t<T, N>>;
      const V init = V([&](auto i) { return T(i); }) + first_value;
      store(init, std::addressof(*first), element_aligned);
      first += V::size();
      epilogue<V::size() / 2>(first, last, init[V::size() - 1] + 1);
    }
  }
}

template <class ContiguousIterator>
void iota(execution::simd_policy, ContiguousIterator first, ContiguousIterator last,
          typename ContiguousIterator::value_type first_value) {
  using T = ContiguousIterator::value_type;
  using V = native_simd<T>;
  V init = V([&](auto i) { return T(i); }) + first_value;
  const V stride = T(V::size());
  for (; distance(first, last) >= V::size(); first += V::size(), init += stride) {
    store(init, std::addressof(*first), element_aligned);
  }
  epilogue<V::size() / 2>(first, last, init[V::size() - 1] + 1);
}
```

Listing 2: Implementation idea for the `iota` function used in Listing 1.
template <size_t N, class ContiguousIterator, class UnaryFunction>
inline void epilogue(ContiguousIterator first, ContiguousIterator last,
                        UnaryFunction f) {
  if constexpr (N > 0) {
    using T = ContiguousIterator::value_type;
    using V = simd<T, simd_abi::deduce_t<T, N>>;
    if (distance(first, last) >= V::size()) {
      V tmp(std::addressof(*first), element_aligned);
      f(tmp);
      if constexpr (is_functor_argument_mutable_v<UnaryFunction, V>) {
        store(tmp, std::addressof(*first), element_aligned);
      }
    }
  }
  epilogue<V::size() / 2>(first, last, f);
}

template <class ContiguousIterator, class UnaryFunction>
void for_each(execution::simd_policy, ContiguousIterator first,
              ContiguousIterator last, UnaryFunction f) {
  using V = native_simd<ContiguousIterator::value_type>;
  for (; distance(first, last) >= V::size(); first += V::size()) {
    V tmp(std::addressof(*first), element_aligned);
    f(tmp);
    if constexpr (is_functor_argument_mutable_v<UnaryFunction, V>) {
      store(tmp, std::addressof(*first), element_aligned);
    }
  }
  epilogue<V::size() / 2>(first, last, f);
}

Listing 3: Implementation idea for the for_each function used in Listing 1.
Elements. Instead the epilogue recursion generates a new init simd object for size 2 and subsequently for size 1.

Figure 2 visualizes the end of the for_each implementation. The main for loop processes four elements of the std::vector in parallel. It executes a vector load, calls the user-provided function with the temporary simd object, and executes a vector store back to the same memory location. The remaining three elements are again handled by an epilogue recursion which divides the number of processed elements by 2 with every step.

For both algorithms it would be perfectly valid to implement the epilogue as a sequential loop using simd objects with size 1.
Figure 2: Visualization of chunking the \texttt{for_each} call with $W_T = 4$ in Listing 1.
In general, the execution::simd policy requires algorithms to make a copy from the input sequence. For now, since simd only supports arithmetic types and simd does not return lvalue references to its values, it is not observable whether a copy was made. With two exceptions:

- Modification of the input sequence via different means than the function parameter(s) will not modify the value of the function parameter(s).
- Using mutable iterators, assignment to the simd (lvalue reference) parameter of the user-supplied function object will not modify the output sequence until after the function has returned (cf. Listing 3).

Note that most non-modifying sequence operations allow modification of the sequence by using a non-const lvalue reference parameter for the user-supplied function object.

Algorithms that take a predicate returning a bool have two possible vectorization strategies:

1. The predicate still returns bool. In this case, every predicate must execute a simd_mask reduction. This makes it simple to short-circuit in the algorithm implementation but may unnecessarily restrict the achievable parallelization.

2. The predicate returns simd_mask. In this case reductions can happen in parallel. Short-circuiting is still possible, but requires a simd_mask reduction on each step (QoI question).

I recommend to allow both. Let the algorithm switch the strategy depending on the return type of the predicate. Let the user decide on the trade-offs.

For many algorithms, the complexity requirement states “Applies f exactly last – first times”. In the execution::simd case, the number of applications of f is reduced by an unspecified factor.

The Compare function object type is required to return a value that is contextually convertible to bool. For sorting, it is important that overloads using the execution::simd policy work with simd_mask instead of bool. It is not useful for the sort algorithm to know whether all/any/some/none of the compared values are “less than”. It requires a mask object to know the “less than” relation for each individual value.
4.3 design alternative

There are subtle differences in how the `std::simd` specializations need to be used (e.g., `std::generate` currently requires the generator function to return objects that can be assigned to a dereferenced `ForwardIt`; the `std::simd` specialization requires the generator function to return objects of type `std::simd<ForwardIt::value_type>`). An attempt to fit `std::simd_policy` into the existing wording results in some special-casing in the algorithm specifications. This observation leads to the question whether a new execution policy is really the best approach. The alternative would be a duplication of algorithms to variants with a `simd_` prefix in their name. Example:

```cpp
std::simd_for_each(data.begin(), data.end(), [](auto &x) {
  x *= x;
});
```

This alternative would not reduce the amount of wording/complexity though, since now a lot of the algorithm wording would need to be duplicated. However, this would allow a very simple reduction of the number of algorithms that support `std::simd` execution.

4.4 affected algorithms

The following algorithms have an `std::execution::simd_policy` overload and can work with a `std::simd_policy` specialization:

- `all_of`, `any_of`, `none_of`
- `for_each`, `for_each_n`
- `find`, `find_if`, `find_if_not`
- `find_end`
- `find_first_of`
- `adjacent_find`
- `count`, `count_if`
- `mismatch`
- `equal`
- `search`, `search_n`
• copy, copy_n (no real need; can be implicitly vectorized)
• copy_if
• swap (no real need; can be implicitly vectorized)
• transform
• replace, replace_if, replace_copy, replace_copy_if
• fill, fill_n (no real need; can be implicitly vectorized)
• generate, generate_n

Note that the generator function passed to generate/generate_n does not expect any arguments and thus has no interface for the algorithm to request a certain ABI tag from the function (template). Consequently, either the user could choose the ABI tag (via the return type) and expect values at the tail (beyond end) to be discarded. Alternatively, the algorithm could pass an arbitrary (or default- or zero-initialized) data-parallel object to the function. This would communicate the expected return type of the generator function. See Figure 3 for an example. It is possible to allow both variants.

• remove, remove_copy (no real need; can be implicitly vectorized)
• remove_if, remove_copy_if
• unique, unique_copy
• reverse, reverse_copy (no real need; can be implicitly vectorized)
• rotate, rotate_copy (no real need; can be implicitly vectorized)
• is_partitioned, partition, stable_partition, partition_copy, partition_point
• sort, stable_sort, partial_sort, partial_sort_copy, is_sorted, is_sorted_until
• nth_element
• merge, inplace_merge
• includes, set_union, set_intersection, set_difference, set_symmetric_difference
std::array<float, N> data;

// let the generator function choose the ABI tag and discard
// N % native_simd<float>::size() values beyond data.end():
std::generate(std::execution:: simd, data.begin(), data.end(), []() {
    return native_simd<float>();
});

// Alternative: the algorithm tells the generator function via the argument
// what data-parallel type it expects to get.
std::generate(std::execution:: simd, data.begin(), data.end(), [](auto x) {
    return x = 0;
});

Figure 3: Generator function return type example.

- min_element, max_element, minmax_element
- lexicographical_compare

The remaining algorithms have no obvious use for the specialization:

- move makes no sense until we can create simd<T> types for pointers (likely) and
class types (less likely).

  lower_bound, upper_bound, equal_range, and binary_search may benefit from
simd usage, but currently do not provide ExecutionPolicy overloads.
I have not considered is_heap and is_heap_until yet.

4.5 initial wording for the policy

Add a new execution policy to [N4659, §23.19.2]:

§23.19.2 [execution.syn]

// 23.19.6, parallel and unsequenced execution policy
class parallel_unsequenced_policy;

// 23.19.7, simd execution policy
class simd_policy;

// 23.19.28, execution policy objects:
inline constexpr sequenced_policy seq{ unspecified };  
inline constexpr parallel_policy par{ unspecified };    
inline constexpr parallel_unsequenced_policy par_unseq{ unspecified };  
inline constexpr simd_policy simd{ unspecified };
Renumber §23.19.7 to §23.19.8 and add §23.19.7 [execpol.simd]:

```cpp
class simd_policy { unspecified; }
```

1. The class `simd_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm’s execution may be vectorized using `simd` for interfacing with user-provided functionality.

2. During the execution of a parallel algorithm with the `execution::simd_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

Add to §23.19.8 [execpol.objects]:

```cpp
inline constexpr execution::simd_policy execution::simd{ unspecified; }
```

[N4659, §28.4.2] defines requirements on user-provided function objects. This might be the right place to add:

Function objects passed into parallel algorithms instantiated with the `execution::simd` execution policy shall:

- be callable with arguments of type `simd<Iterator::value_type, Abi>`, for any ABI tag `Abi`, for all arguments that otherwise would be of type `Iterator::value_type`;
- return objects of type `simd<Iterator::value_type, Abi>`, if the function object is otherwise expected to return objects assignable to a dereferenced `Iterator` object;
- return objects of type `simd_mask<Iterator::value_type, Abi>` or `bool`, if the function object is otherwise expected to return `bool`.

The following subsection in [N4659, §28.4.3] defines the semantics of the execution policies. A new paragraph for `execution::simd` is needed. The intent is to

1. constrain execution to the calling thread,

2. allow implementations to assume unordered access for all internal element access functions (most importantly loads and stores),

3. apply user-provided function objects in the order the `simd` chunks are created from sequential iteration over the iterator(s).
The invocations of element access functions in parallel algorithms invoked with an execution policy object of type execution::simd_policy are permitted to execute in an unordered fashion in the calling thread, except for the application of user-provided function objects. User-provided function objects are called with an implementation-defined number of sequence elements combined into a simd<T, Abi> object. The type for Abi is chosen by the implementation. It may be different for subsequent applications of the user-provided function in the same parallel algorithm invocation. The type for T is the decayed type of the sequence elements. The order of elements in the simd object is equal to the order of the corresponding elements in the sequence argument. The invocation order of user-provided function objects is sequential.

It is my understanding that we do not want to add anything to [N4659, §28.4.4 [algorithms.parallel.exceptions]] at this point. The situation is simpler for the execution::simd policy. It is almost equivalent to the seq policy.

4.6 wording for individual algorithms

Compare is a function object type. The return value of the function call operation applied to an object of type Compare, when contextually converted to bool, yields true if the first argument of the call is less than the second, and false otherwise. If the ExecutionPolicy is execution::simd_policy, the return type of the function call operation applied to an object of type Compare is a specialization of simd_mask. Its i-th element in the simd_mask yields true if the value of the i-th element of the first argument of the call is less than the corresponding element of the second, and false otherwise. Compare comp is used throughout for algorithms assuming an ordering relation. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

I have not identified the need for any additional wording in the subsections on the individual algorithms for the execution::simd_policy at this point.

A BIBLIOGRAPHY


