Document Number: P0350R1
Date: 2017-07-30
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG

Integrating simd with parallel algorithms

ABSTRACT

This paper discusses a new execution policy for integrating `simd` with *parallel algorithms*.

CONTENTS

0 Remarks

- This documents talks about "vector" types/objects. In general this will not refer to the `std::vector` class template. References to the container type will explicitly call out the `std` prefix to avoid confusion.
- [P0214R5] is the last paper on `simd`.

1 Changelog

1.1 Changes from R0

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.

2 Straw polls

2.1 SG1 at Oulu

Poll: Ship it to LEWG?

SF	F	N	A	SA
6	6	2	0	0

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.

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

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

As a consequence, the applicability of the execution policy is limited to iterators where `simd<Iterator::value_type>` is a valid template instantiation of `simd`. A future extension of `simd` may lift this restriction by allowing certain (or all) user-defined types as first template argument to `simd`.

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` 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 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

```cpp
template <size_t N>
void epilogue(ContiguousIterator first, ContiguousIterator last,
              ContiguousIterator::value_type first_value);

template <>
inline void epilogue<0>(ContiguousIterator, ContiguousIterator,
                        ContiguousIterator::value_type) {}

template <size_t N>
inline void epilogue(ContiguousIterator first, ContiguousIterator last,
                     ContiguousIterator::value_type first_value) {
  if (distance(first, last) >= N) {
    using T = ContiguousIterator::value_type;
    using V = simd<T, abi_for_size_t<N>>;
    const V init = V([&](auto i) { return T(i); }) + first_value;
    store(init, std::addressof(*first), flags::element_aligned);
    first += V::size();
  }
  epilogue<V::size() / 2>(first, last, init[V::size() - 1] + 1);
}

void iota(execution::simd_policy, ContiguousIterator first, ContiguousIterator last,
          float first_value) {
  using T = ContiguousIterator::value_type;
  using V = simd<T, simd_abi::native>;
  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), flags::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.

```cpp
template <size_t N>
void epilogue(ContiguousIterator first, ContiguousIterator last, UnaryFunction f);

template <>
inline void epilogue<0>(ContiguousIterator, ContiguousIterator, UnaryFunction) {}

template <size_t N>
inline void epilogue(ContiguousIterator first, ContiguousIterator last,
                     UnaryFunction f) {
  using V = simd<ContiguousIterator::value_type, abi_for_size_t<N>>;
  V tmp(std::addressof(*first), flags::element_aligned);
  f(tmp);
  if (is_functor_argument_mutable<UnaryFunction, V>::value) {
    store(tmp, std::addressof(*first), flags::element_aligned);
  }
  epilogue<V::size() / 2>(first, last, f);
}

void for_each(execution::simd_policy, ContiguousIterator first,
              ContiguousIterator last, UnaryFunction f) {
  using V = simd<ContiguousIterator::value_type, simd_abi::native>;
  for (; distance(first, last) >= V::size(); first += V::size()) {
    V tmp(std::addressof(*first), flags::element_aligned);
    f(tmp);
    if (is_functor_argument_mutable<UnaryFunction, V>::value) {
      store(tmp, std::addressof(*first), flags::element_aligned);
    }
  }
  epilogue<V::size() / 2>(first, last, f);
}
```

Listing 3: Implementation idea for the `for_each` function used in Listing 1.

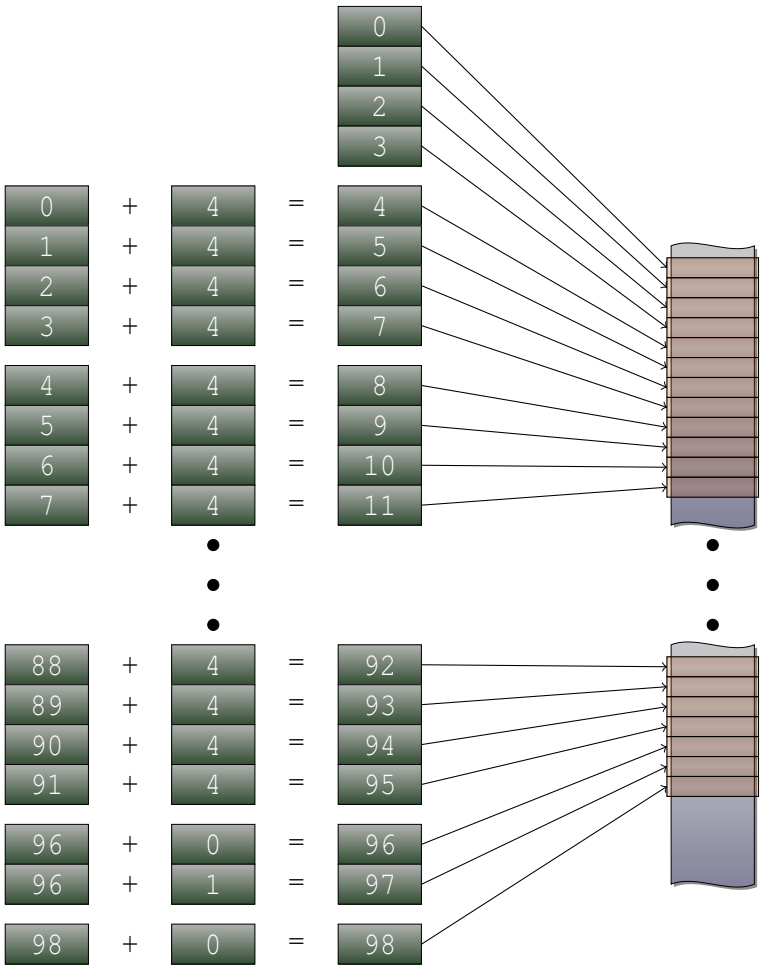

Figure 1: Visualization of chunking the `iota` call with $\mathcal{W}_{\mathrm{T}} = 4$ in Listing 1.

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.

4.2 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;
```

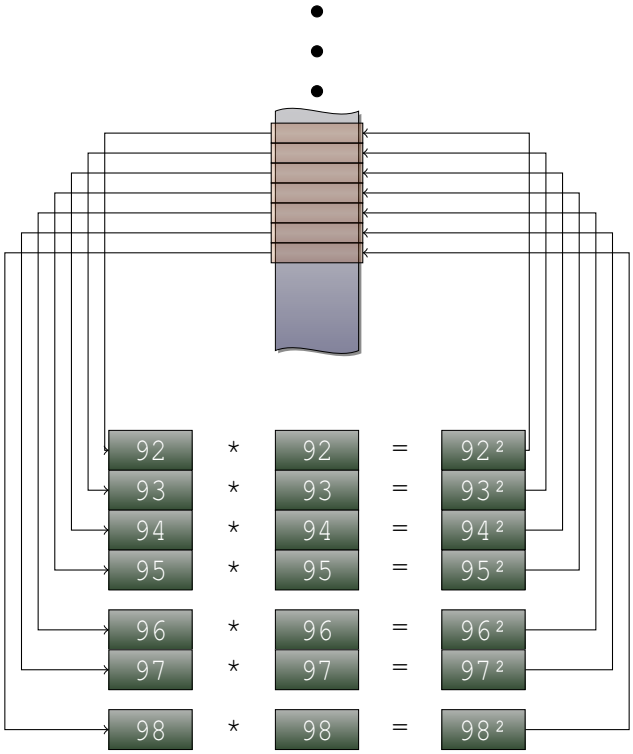

Figure 2: Visualization of chunking the `foreach` call with $\mathcal{W}_{\mathrm{T}} = 4$ in Listing 1.

```
// 23.19.~~7~~8, 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]:

```
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]:

```
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:

§28.4.2 [algorithms.parallel.user]

4 Function objects passed into parallel algorithms instantiated with the `execution::simd` execution policy shall be callable with any argument of type `simd<T, Abi>`, where `T` is the type obtained from dereferencing the iterator.

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).

§28.4.3 [algorithms.parallel.exec]

12 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.

4.3 Wording for individual algorithms

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 Acknowledgements

This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).

B Bibliography

[P0214R5] Matthias Kretz. *P0214R5: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0214r5.pdf.

[P0350R0] Matthias Kretz. *P0350R0: Integrating datapar with parallel algorithms and executors*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0350r0.pdf.

[N4659] Richard Smith, ed. *Working Draft, Standard for Programming Language C++*. ISO/IEC JTC1/SC22/WG21, 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.