Document Number: P0350R0
Date: 2016-05-24
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: SG1

INTEGRATING DATAPAR WITH PARALLEL ALGORITHMS AND EXECUTORS

ABSTRACT

This paper discusses a new execution policy for integrating datapar 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.
- [P0214R1] is the last paper on `datapar`.

1 INTRODUCTION

1.1 PARALLEL ALGORITHMS

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 `datapar_execution`. `datapar_execution` requires user-provided function objects to be callable with `datapar<T, Abi>` arguments instead of the `T` arguments the `std::sequential` variant would use. The algorithm therefore processes chunks of `datapar<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 `datapar<Iterator::value_type>` is a valid template instantiation of `datapar`. A future extension of `datapar` may lift this restriction by allowing certain (or all) user-defined types as first template argument to `datapar`.

1.2 EXECUTORS

Executors abstract execution resources (see e.g. [P0058R1]). One of the execution resources this covers is SIMD units (or any other comparable data-parallel execution). [P0058R1] shows an example for a `vector_executor` implementation using `#pragma simd`. An alternative approach (competing or complementary) uses `datapar` to express the data parallelism via the type system. The user-provided function object to the executor's `execute` function follows the same idea as for the parallel algorithms. The executor passes an index object to the user-provided function object to identify the partition of the work the function needs to process. For a `datapar_executor` this index object could be a new type identifying an index range. Overloads of the subscript operator (or other functions) can be used to load/store `datapar` objects using this index range object.

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

Listing 1: Example using `datapar_execution` with `iota` and `for_each`.

1.3 FUTURE WORK

Finally, though not covered in this paper, we should consider using `datapar` as the ABI type that enables calling into vectorized functions from code executed via `std::par_vec` or from a `vector_executor` as suggested in [P0058R1].

2 PARALLEL ALGORITHMS

2.1 EXAMPLE

Consider the example in Listing 1. The `iota` and `for_each` functions each could create an internal `datapar` 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 `datapar` object back. Otherwise, the store is optimized away.

Figure 1 shows a visualization how the `iota` implementation works. The `init datapar` object is stored via vector stores to 4 (native `datapar::size()`) elements in the `std::vector`. In each iteration the `init` object is incremented by `datapar::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 datapar` 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 `datapar` object, and executes

```
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 V = datapar<ContiguousIterator::value_type, abi_for_size_t<N>>;
    const V init = sequence_from_zero<V>() + first_value;
    store(init, std::addressof(*first), flags::unaligned);
    first += V::size();
  }
  epilogue<V::size() / 2>(first, last, init[V::size() - 1] + 1);
}

void iota(datapar_execution_policy, ContiguousIterator first, ContiguousIterator last,
          float first_value) {
  using V = datapar<ContiguousIterator::value_type, datapar_abi::native>;
  V init = sequence_from_zero<V>() + first_value;
  const V stride = static_cast<float>(V::size());
  for (; distance(first, last) >= V::size(); first += V::size(), init += stride) {
    store(init, std::addressof(*first), flags::unaligned);
  }
  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 = datapar<ContiguousIterator::value_type, abi_for_size_t<N>>;
  V tmp = load<V>(std::addressof(*first), flags::unaligned);
  f(tmp);
  if (is_functor_argument_mutable<UnaryFunction, V>::value) {
    store(tmp, std::addressof(*first), flags::unaligned);
  }
  epilogue<V::size() / 2>(first, last, f);
}

void for_each(datapar_execution_policy, ContiguousIterator first,
              ContiguousIterator last, UnaryFunction f) {
  using V = datapar<ContiguousIterator::value_type, datapar_abi::native>;
  for (; distance(first, last) >= V::size(); first += V::size()) {
    V tmp = load<V>(std::addressof(*first), flags::unaligned);
    f(tmp);
    if (is_functor_argument_mutable<UnaryFunction, V>::value) {
      store(tmp, std::addressof(*first), flags::unaligned);
    }
  }
  epilogue<V::size() / 2>(first, last, f);
}
```

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

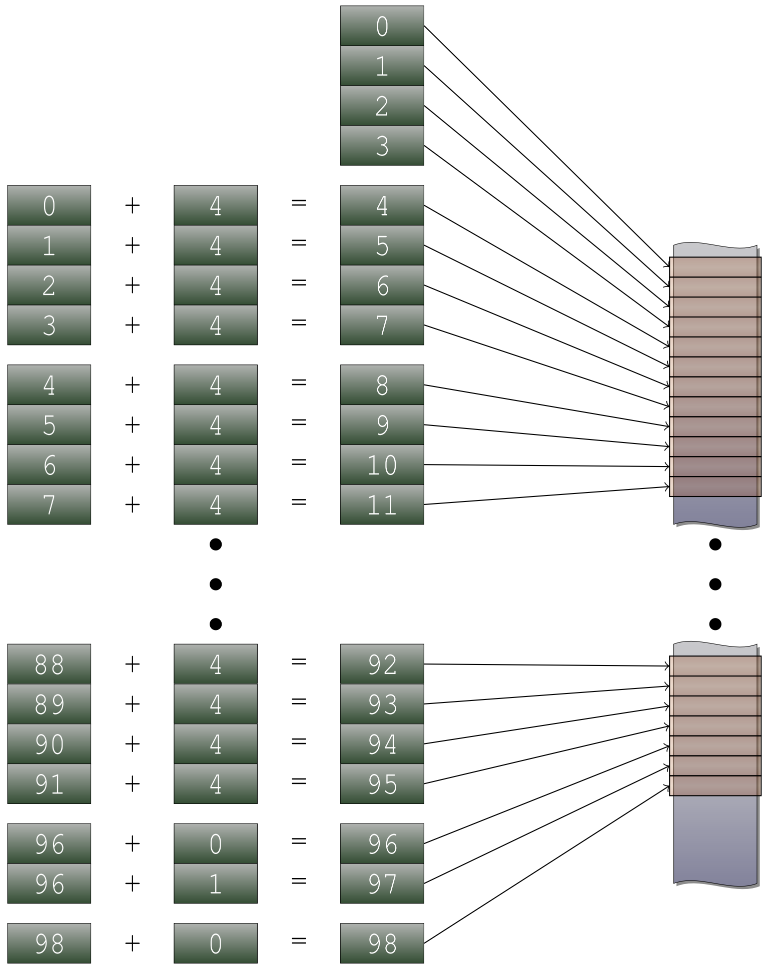

Figure 1: Visualization of chunking the `iota` call with $\mathcal{W}_{\mathtt{T}} = 4$ in Listing 1.

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 `datapar` objects with size 1.

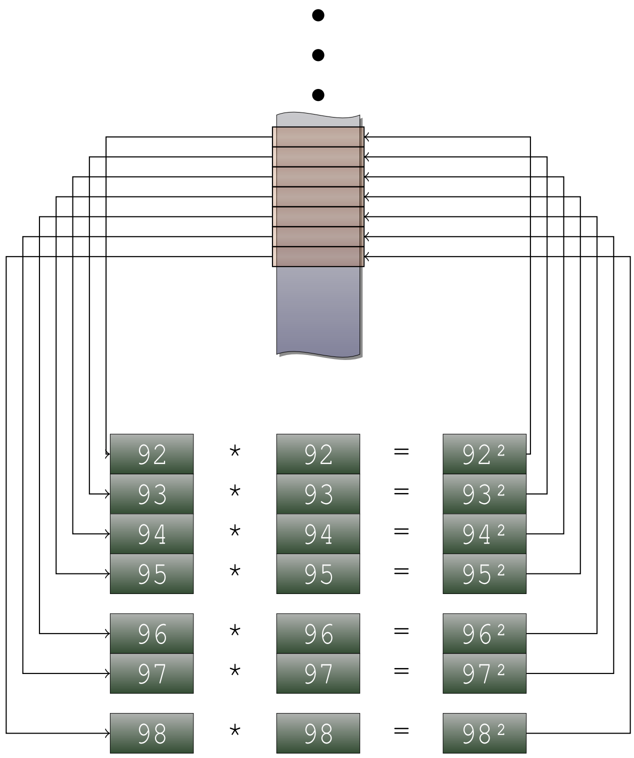

Figure 2: Visualization of chunking the `foreach` call with $\mathcal{W}_{\mathrm{T}} = 4$ in Listing 1.

2.2 WORDING FOR THE POLICY

Add a new execution policy to [N4582, §20.18.2]:

§20.18.2 [execpol.syn]

```
// 20.18.6, parallel+vector execution policy:
class parallel_vector_execution_policy;

// 20.18.7, datapar execution policy:
class datapar_execution_policy;

// 20.18.78, execution policy objects:
constexpr sequential_execution_policy sequential{ unspecified };
constexpr parallel_execution_policy par{ unspecified };
constexpr parallel_vector_execution_policy par_vec{ unspecified };
constexpr datapar_execution_policy datapar_execution{ unspecified };
```

Renumber §20.18.7 to §20.18.8 and add §20.18.7 [execpol.datapar]:

```
class datapar_execution_policy { unspecified };
```

1 The class `datapar_execution_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 `datapar` for interfacing with user-provided functionality.

Add to §20.18.8 [parallel.execpol.objects]:

```
constexpr datapar_execution_policy datapar_execution{ unspecified };
```

[N4582, §25.2.2] defines requirements on user-provided function objects. This might be the right place to add:

§25.2.2 [algorithms.parallel.user]

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

The following subsection in [N4582, §25.2.3] defines the semantics of the execution policies. A new paragraph for `datapar_execution` 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 `datapar` chunks are created from sequential iteration over the iterator(s).

§25.2.3 [algorithms.parallel.exec]

9 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `datapar_execution_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 `datapar<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 `datapar` object is equal to the order of the corresponding elements in the sequence argument. The invocation order of user-provided function objects is sequential.

[N4582, §25.2.4 (2.2)] needs to add `datapar_execution_policy`.

§25.2.4 (2.2) [algorithms.parallel.exceptions]

If the execution policy object is of type `sequential_execution_policy`, `datapar_execution_policy`, or `parallel_execution_policy`, the execution of the algorithm exits via an exception.

There is no need for multiple exceptions when applying user-provided function objects. The need for exception lists only arises in the vector-parallel execution of iterator operations.

2.3 WORDING FOR INDIVIDUAL ALGORITHMS

I have not identified the need for any additional wording in the subsections on the individual algorithms for the `datapar_execution_policy` at this point.

It might be useful to only require `MoveConstructible` user-provided functions instead of the stricter requirement of `CopyConstructible`.

```
1 std::vector<float> data = ...;
2 datapar_executor exec;
3 exec.execute([&](auto idx) {
4   auto x = data[idx];  // decltype(x) is datapar<float, Abi>
5   where(x < 0, x) += 360.f;
6   data[idx] = x;
7 }, data.size());
```

Listing 4: Example use of the `datapar_executor`.

3 EXECUTORS

Consider the example in Listing 4. The line 3 requests the `datapar_executor` to generate index objects for the index range `0–data.size()`. The type of the index object is determined via deduction and can be different in subsequent invocations of the callable. For example, if `data.size()` is 13, the first `idx` object may denote the range 0–7, the second `idx` object denotes 8–11, and the third `idx` object denotes 12. An overload of the subscript operator of `std::vector` in line 4 turns the expression into an efficient SIMD vector load operation.[1] Line 5 modifies the elements of `x` that are negative. Line 6 finally stores the result back to `data`.

The example shows how the executor solves the "load store problem" of `datapar`: Requiring the user to explicitly partition the loop into different chunk sizes and call loads and stores explicitly is more low-level than we want the average user to work. The executor solves this and at the same time enables better composition with the upcoming facilities for concurrency in C++.

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).

1 Note that the executor cannot know anything about the alignment of `data`. Therefore, the conservative approach must default to unaligned loads and stores. Load-store flags, applicable to load and store operations of `datapar`, could be incorporated into the type of `idx`. The question remains, how the `execute` function determines those flags. This likely needs to be a template parameter of the `datapar_executor` class.

B BIBLIOGRAPHY

[P0058R1] Jared Hoberock, Michael Garland, and Giroux Olivier. *P0058R1: An Interface for Abstracting Execution*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0058r1.pdf.

[P0214R1] Matthias Kretz. *P0214R1: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0214r1.pdf.

[N4582] Richard Smith, ed. *Working Draft, Standard for Programming Language C++*. ISO/IEC JTC1/SC22/WG21, 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4582.pdf.