Document number: P0076R4
Date: 2017-07-14
Project: Programming Language C++ (WG21)
Subgroup/Audience CWG and LWG
Reply to: Pablo Halpern <pablo.g.halpern@intel.com>
Arch D. Robison <arch.robison@intel.com>
Robert Geva <robert.geva@intel.com>
Clark Nelson <clark.nelson@intel.com>
Jens Maurer <jens.maurer@gmx.net>

Vector and Wavefront Policies

Contents

1 Motivation

Vector parallelism is insufficiently supported by the current Parallelism TS (N4578). The Parallelism TS does offer the `par_unseq` policy, and there is some interest in a variant that restricts execution to a single thread; the result of such a restriction is the `unseq` policy proposed in this paper. Alas, this policy, though it allows a vectorization (exploiting vector hardware), it is excessively permissive and fails to express the necessary requirements for an important set of vectorizable loops of practical interest. As defined in N4507, `par_unseq` allows:

> "The invocation of element access functions ... are permitted to execute in an unordered fashion in unspecified threads and unsequenced with respect to one another within each thread. [*Note*: this means that multiple function object invocations may be interleaved on a single thread. – *end note*]"

Merely constraining `par_unseq` to a single thread still allows permissive interleaving that would give undefined semantics to loops in the aforementioned set.

Here is a short example that falls in the gap, using `for_loop` from P0075 with `vector_policy` proposed in this paper:

```
void binomial(int n, float y[]) {
    for_loop( vec, 0, n, [&](int i) {
        y[i] += y[i+1];
    });
}
```

The call to `for_loop` is equivalent, except with more relaxed sequencing, to:

```
void binomial(int n, float y[]) {
    for( int i=0; i<n; ++i )
        y[i] += y[i+1];
}
```

The `for_loop` example cannot safely use `unseq` or `par_unseq` instead of `vec`, because that would result in unsequenced reads and writes of the same element of `y` when `n≥2`. Subsequent sections show some more examples that require `vec` instead of `unseq`.

The proposals in this paper are targeted for a future parallelism TS.

2 Change History

2.1 Changes from R3 to R4

From Core review in Toronto:

- Removed vagueness from the definition of "evaluation A *contains* evaluation B".
- Added definition of *ordered before*, which is used to correct the definition of *vertical antecedent* so that it no longer breaks when subexpressions are unsequenced or indeterminately sequenced.
- Fixed erroneous initial condition for establishing *horizontally matched*.
- Fixed uses of the term *horizontal antecedent* (replaced by *horizontally matched*), which were left over from an earlier revision.

From LWG in Toronto:

- Clarified meaning of new execution policy classes.
- Fixed up `ordered_t` member functions to inline definitions in order to avoid complex specification of return types.
- Other wording fixups.

2.2 Changes from R2 to R3

- R2 with the following revisions was approved by LEWG on Wednesday morning, 2016-06-22, and by EWG on Thursday morning, 2016-06-23, in Oulu. There was concern in EWG whether the description of wavefront execution is sufficient for creating a fresh implementation from the spec.
- Rename `vec_off` to `no_vec`.
- Add `noexcept` to `no_vec`. Add note indicating that if function invoked by `no_vec` throws, then terminate is called, consistent with `vector_policy` execution.
- Delete self-assignment operator for `ordered_update_t`.
- Add `const` to all members of `ordered_update_t`.
- Add `noexcept` to all members of `ordered_update_t`.
- Add note indicating that members of `ordered_update_t` typically return by value.
- Applied name changes from P0413r0.

2.3 Changes from R1 to R2

- As requested by SG1, removed ordered scatter rule. See section 6.5 for more details.
- Added non-normative clarifying notes to the description of wavefront execution.

2.4 Changes from R0 to R1

- Changed formal specification of wavefront ordering to use a much simpler *horizontal match* formulation instead of labeling each evaluation with a LIFO context.
- Added `ordered_update` and its helper class `ordered_update_t`.
- Changed `vec_off(f)` to return result of `f()` instead of discarding it.
- Separated the controversial "ordered scatters" rule from the rest of the proposal, so that it can be voted on separately.

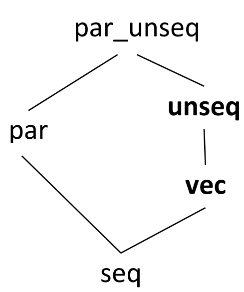
3 Execution policies for vectorization

3.1 Unsequenced and vector execution policies

This paper proposes adding two new execution policies to the Parallelism TS, assuming the adoption of P0075. These policies add support for execution with relaxed sequencing restricted to a single OS thread:

- An `unsequenced_policy` class and constant `unseq` analogous to the other policy types and constants in the Parallelism TS, with sequencing semantics similar to `parallel_unsequenced_policy`, but limited to a single OS thread.
- A `vector_policy` class and constant `vec` that is similar to the policy above, but guarantees stronger sequencing, compatible with classic work in the field of vectorization. This policy is restricted to the indexed-based loop templates proposed in P0075.

The first policy has strictly weaker sequencing guarantees than the second. The following lattice summarizes the strength of their guarantees relative to each other and existing policies, with the weakest guarantees at the top.



No compiler extensions are necessary for correct implementation; since an implementation is free to implement any policy higher on the lattice via a policy lower on the lattice, serial execution is always allowed. The goal, however, is for the implementation to exploit parallel hardware, especially vector units, for improved performance. Some combination of OpenMP directives and vendor-specific hooks are likely to be used for implementing algorithms with either policy.[1]

The ability to constrain execution to a single OS thread is commonly useful for avoiding resource interference with multi-threading designs.

Having two new policies, instead of one, and restricting `vec` to `for_loop` resolves a fundamental conflict. The `unseq` policy is generally useful and straightforward to define for the parallel algorithms in the Parallelism TS, but fails to capture guarantees critical to an important class of loops. Conversely, `vec` is critically useful for an important class of loops and definable for `for_loop`, but seems impractical to generalize to the parallel algorithms in a way that is both well-defined and beneficial to exploit.

3.2 Extensibility of Policies

Though we don't propose it for standardization at this time, we note that `vector_policy` could be subclassed to provide additional information from the programmer to the compiler. Providing this information as static const member of integral type would enable cognizant compilers to find it a compile time, as in the following example:

```
struct my_policy: vector_policy {
    static const int safelen = 8;
    static const bool vectorize_remainder = true;
};

for_loop( my_policy(), 0, 1912, [&](int i) {
    Z[i+8] = Z[i]*A;
});
```

Here, `safelen` is a *semantic* piece of information, similar to a `safelen` clause in OpenMP 4.0, that says that the (i+9)th[2] application of the function cannot start until the ith and prior applications complete. For programmers to rely on this in portable code would require standardizing it.

In contrast, `vectorize_remainder` is a performance hint, and could remain vendor specific.

4 Wavefront Application

Our proposed `vector_policy` gives programmers classic "vector loop" evaluation order guarantees when used with function template `for_loop` from P0075. We abstract the

---

[1]In particular, we implemented a performant version of vector reductions for `for_loop` in LLVM by adding special intriniscs.

[2] Yes, 9 and not 8. The wavefront semantics prevent the oldest iteration in flight from getting behind the newest iteration in flight.

evaluation order by defining "wavefront[3] application". Intuitively, the *wavefront application* of a function *f* over a sequence of argument lists applies *f* to each argument list in a way that keeps preceding applications from falling behind later application. This property distinguishes our `vector_policy` from our `unsequenced_policy`. The wavefront property has two benefits:

- It enables exploiting "forward dependencies", a common technique in classic vector codes.
- It implies that `vector_policy` is safe to use on any loop that could be auto-vectorized.

For example, consider:[4]

```
void f() {
    extern float U[], V[], A, B;
    for_loop( vec, 1, 999, [&](int i) {
        V[i] = U[i+1]*A;
        U[i] = V[i-1]+B;
    });
}
```

For this code to have the same side effects with `vec` as with the `seq` policy, it is imperative that the load of U[*k*] preceded a store into U[*k*] in a later iteration, and likewise that the store into V[*k*] precede the load of V[*k*] in a later iteration. Our wavefront semantics coupled with the subscript patterns give those guarantees. With the more relaxed ordering of our `unsequenced_policy` (or the existing `parallel_policy` or `parallel_unsequenced_policy`) the programmer would need to fission the loop into two loops, with the consequent penalty of increasing consumption of memory bandwidth.

Wavefront application provides the **necessary** conditions for vectorization on classic "long vector" machines in the tradition of Cray and Convex, vectorization on "short vector" architectures (such as Intel® SSE, Intel® AVX, ARM® NEON, and Freescale® AltiVec), as well as software pipelining and unroll-and-interleave optimizations, without introducing relaxations that would be harmful for some loops.

4.1 Horizontal Matching

Precisely defining "ahead" and "behind" can be tricky for functions with control flow that repeats evaluation of an expression. We solve the problem by refining the sequencing rules from N4237 to handle cyclic control flow. Our refinement uses "horizontal matching" that distinguish evaluating the same expression or statement during different trips though a loop or in different invocations of a callee.

---

[3]The term "wavefront" for similar orderings has a long history in the field of vector and parallel programming. An example is Figure 7 from reference [4].

[4]The example is a toy, but the dependence pattern is similar to those in staggered finite-time finite-difference codes.

Furthermore, unstructured control flows (`goto`s and `switch` statements like in "Duff's device") are handled by temporarily disabling synchronization guarantees across iterations, but in a way that limits the disabling to within a certain scope. While disabled, the `vec` policy temporarily acts like the `unseq` policy (i.e., the sequencing guarantees are relaxed).

Horizontal matching is fully defined and further explained in the proposed wording section (Section 7.6). For the next section, it suffices to understand that horizontal matching formalizes an intuitive notion of matching up corresponding evaluations in a sensible way. For example, given this code:

```
for_loop(par, 0, 4, [&](int k){
    if (k % 2)
        f(k);
    else
        g(k);
    h(k);
}
```

the rules horizontally match each row of evaluations shown in the table below.

Expression	**k=0**	**k=1**	**k=2**	**k=3**
x % 2	0 % 2	1 % 2	2 % 2	3 % 2
f(x)		f(1)		f(3)
g(x)	g(0)		g(2)	
h(x)	h(0)	h(1)	h(2)	h(3)

Executions of iterative statements are matched by matching each iteration in turn, giving up after at least one loop quits. Unstructured control-flow turns off matching until it becomes structured again. We defer the details of when this happens to the proposed wording section (7.6).

4.2 Ordering Rules for Wavefront Application

4.2.1 High-level view

The invocations of element access functions in our `for_loop` template from P0075 invoked with an execution policy of type `vector_policy` are permitted to execute in an unordered fashion in the calling thread, unsequenced with respect to one another within the calling thread, but restricted by the "wavefront application" ordering constraints formalized in the proposed wording in Section 7.6.

Figure 1 sketches the rule for the i^{th} and j^{th} invocations of the element access function, where i<j. The subscripted letters denote expression evaluations or statement executions. Dashed lines denote "horizontally matched"; solid arrows denote "sequenced before". Our rules require that if either black partial triangle exists, then the blue sequenced-before relationship must be enforced to complete the triangle.

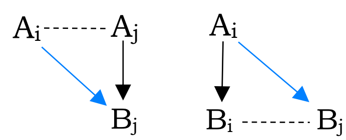


Figure 1 Horizontally matched and sequenced before relationships

Thus the j^{th} iteration cannot get ahead of the i^{th} iteration.

4.2.2 Wavefront ordering for loops within the element access function

Consider the following vector `for_loop` invocation with a serial `for` loop nested within the element-access function (a lambda expression, in this case):

```
for_loop( vec, 0, 2, [&](int i) {
    for( int m=i; m<2; ++m )
        A[m][i] = 1;
    B[i]++;
});
```

The definition of horizontal matching distinguishes the three evaluations of `m<2` and two evaluations of `A[m][i]` as five separate evaluations (in the case of `i=0`), as if the inner loop were unrolled. The dashed lines in Figure 2 show the horizontally matched relationships and the solid arrows show some of the resulting sequenced-before relationships. Evaluations of `++m` and `1` were omitted for brevity. Left side are evaluations for `i=0`; right side for `i=1`. As traditional with such diagrams, we omit some of the arrows inferable via transitive closure.

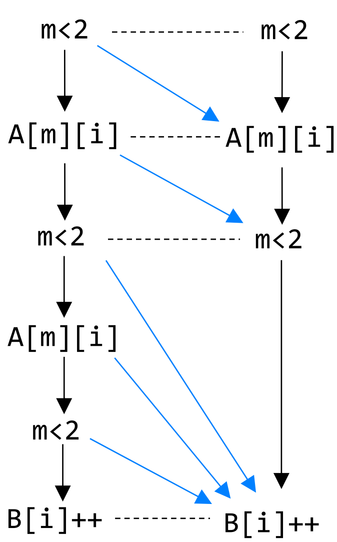


Figure 2 Horizontal Matching in a loop

Two evaluations are horizontally matched if their *vertical antecedents* (see proposed wording in Section 7.6) are horizontally matched. It is critical that *vertical antecedent*, unlike *sequenced before*, is *not* a transitive relationship, so the first evaluation of `m<2` in the first column is *not* horizontally matched with the second evaluation of `m<2` in the second column because their vertical antecedents are not horizontally matched.

If evaluations for different iterations of the inner loop were not distinguished, evaluation of the expression m<2 would be sequenced before `A[m][i]` across applications and *vice-versa*, resulting in arrows from every expression evaluation on the left to every expression evaluation on the right, which would imply serial execution order.

Note that the rules *do* produce sequenced-before relationships from each evaluation within the nested loop to evaluation of `B[i]++` immediately following the loop. This property is called “re-convergence” and is important for maximizing vector parallelism.

5 Functions for strengthening wavefront ordering

Note that if [P0335](P0335) is accepted, then the two functions described here would become member functions of `vector_policy::context_token`.

5.1 `no_vec`

It is sometimes useful to force serial sequencing of a region of code. We define a template function `no_vec` for this purpose. Here is an example:

```
extern int* p;
for_loop( vec, 0, n, [&](int i) {
    y[i] += y[i+1];
    if(y[i]<0) {
        no_vec([]{
            *p++ = i;
        });
    }
});
```

The updates `*p++=i` will occur in the same order as if the policy were `seq`.

Note that we may want to rename this function to `ordered` if P0335 is accepted and this function becomes a member of the context token. For now, LEWG felt that `ordered` is not a good name to have at `std` namespace.

5.2 `ordered_update`

The class template `ordered_update_t` and function template `ordered_update` enable concise expression of some common patterns that require tightening the sequencing rules. Given an lvalue x of type X, a call `ordered_update(x)` returns a proxy of type `ordered_update_t<X>` that sequences assignment and update operations as if they were wrapped in `no_vec`. Example patterns:

```
ordered_update(A[B[i]]) = f(i);          // Scatter
ordered_update(A[B[i]]) += f(i);         // Histogram
++ordered_update(A[B[i]]);               // Histogram
A[i] = (ordered_update(x) += f(i));      // Prefix scan
if(p(i)) A[ordered_update(j)++] = f(i); // Compress
if(p(i)) v = A[ordered_update(j)++];    // Expand
```

6 Alternative Designs Considered

At the September, 2014 meeting in Urbana, the model of vector programming presented here was known as the *wavefront* model. Its key characteristic is that *dynamically-forward loop-carried dependencies* are honored without additional syntax. Two other models described in Urbana were the *lock-step* model and the *explicit ordering-point* model (also called the *explicit barrier* model).

N4238 provides a detailed description of these models, but they can be briefly summarized as follows:

The **lock-step model** groups consecutive loop iterations into chunks of known size, with execution proceeding concurrently on all iterations within a chunk as if each iteration were executing the same operation at the same time (i.e., in lock step).

The **wavefront model** allows iterations to proceed at different rates, but does not allow execution of one iteration to "get behind" execution of a subsequent iteration. Consequently, later iterations can depend on progress guarantees that support dynamically-forward loop-carried dependencies, as in the following example:

```
extern float A[N];
parallel::for_loop(0, N - 1, [&](int i){
    // Evaluate f(A[i+1]) and store the result in A[i] occurs
    // before A[i+1] is modified in the next iteration.
    A[i] = f(A[i + 1]);
});
```

The **explicit ordering-point model** is similar to the wavefront model except that the sequencing relationships required to support dynamically-forward loop-carried dependencies would need to be made explicit by inserting *ordering point* constructs into the loop body, e.g., as in the following example.

```
extern float A[N];
parallel::for_loop(0, N - 1, [&](int i){
    auto tmp = f(A[i + 1]);
    // Ensure that evaluating f(A[i+1]) occurs
    // before A[i+1] is modified in the next iteration.
    parallel::wavefront_ordering_pt();
    A[i] = f(tmp);
});
```

6.1 Previous discussions

There was consensus before Urbana that we wish our loop-like vectorization construct to have serial equivalent semantics; i.e., it should be possible to get semantically correct results by executing the code serially. This goal conflicts with the lock-step model, which requires explicit chunking of the loop and specifies a very restrictive set of valid orderings within a chunk. Moreover, lock-step execution has a semantic whereby results calculated in one iteration of the loop may be required to be available in a *previous* iteration of the loop. Because serial ordering is not a valid ordering with the lock-step model, the lock-step programming model was not considered appropriate as the primary vector programming paradigm in C++. Both the explicit and wavefront models do support serial ordering as a valid implementation choice.

The explicit and wavefront models both had consensus support in Urbana, with the explicit model having slightly stronger support than the wavefront model. The authors of this paper deliberated long and hard on the issue and, after considering many issues, the original authors of this proposal agreed that the wavefront model was the preferred model for *vector* programming, although the explicit model may still have a role to play in some sort of *low-overhead parallel* programming which has yet to be proposed.

In Kona (October 2015), the library syntax for vector loops proposed in P0075 was well received, in general, but the question of implicit versus explicit expression of inter-iteration dependencies remained stalled. Meetings with several hardware vendors and programmers with vectorization expertise reinforced our conclusion that the wavefront model, without explicit ordering points, best expresses vectorization as it historically understood. We did, however, learn that the "ordered scatter" rule in the first version of this paper is separable from the rest of the proposal in that some existing vector systems enforce ordered scatters whereas others do not. For this reason, we have labeled this rule as "optional" and would be willing to vote on it separately.

The remainder of this section is devoted to explaining our rationale for choosing the wavefront model over the explicit model for vector programming.

6.2 The promise and disappointments of the explicit ordering-point model

Conceptually, the explicit ordering-point model is more like a parallel programming model than is the wavefront model. An ordering point would act similar to a software barrier, preventing code motion across the ordering point but allowing it between ordering points. Theoretically, less care to maintain lexical ordering would be needed in early phases of compilation thus permitting more liberal transformations.

As we analyzed this claim of better optimization, however, we discovered some issues. To be sure, there are situations where the claim is true, but there are situations where a naïve compiler could lose optimization opportunities because the ordering points are coarse-grained, and might need to be inserted in multiple places. It is possible to make the ordering points more precise, e.g., by specifying exactly the "to" and "from" points of inter-iteration dependencies. However, this would complicate the syntax in a way that we determined was too arcane and would discourage the use of vectorization.

Moreover, some expressions that are handled naturally in the wavefront model but are difficult to express using explicit ordering points. Assuming arrays A and B and loop control variable i, the expression,

```
A[i] = 2*A[i + 1];
```

requires that A[1] in iteration 1 not be modified until its value has been read in iteration 0. With the explicit ordering-point model, an ordering point would need to be inserted between the read of A[i+1] and the modification of A[i]:

```
auto tmp = A[i + 1];
parallel::wavefront_ordering_pt();
A[i] = 2*tmp;
```

Not only is the above workaround somewhat ugly and potentially error prone, but it show one of several warts that are exposed when the explicit ordering-point model is examined closely. It is not clear how many more such warts are necessary to express the entire body of vectorizable code.

Finally, the explicit model was touted as a way to express a form of parallelism more general than SIMD vectorization and software pipelining (e.g., a low-overhead parallelism that could be implemented on SIMT GPUs). While this idea has some merit, it is somewhat speculative at this point. It is not clear that the model is sufficiently rich to express the desired semantics. It is our opinion that a generalized low-overhead parallelism that can be implemented with multiple mechanisms (including SIMD) should be the subject of a future proposal, after the issues have been thoroughly explored, and with a couple of implementations. We should not hold up support for vectorization pending such exploration.

6.3 Existing Practice

The wavefront model is a formalization of the model that has been used for SIMD and long-vector architectures for decades [1][2][2]. It has been analyzed and refined in the technical literature and it has been implemented in many compilers and in many programming languages including C, C++, and Fortran (via OpenMP as well as proprietary annotations).

The experts in vector programming are familiar with the wavefront model; to them, it's what vector programming looks like. Even if we were to all agree that the explicit model is easier to learn than the wavefront model (and that is certainly not obvious), **we don't want to standardize something that is hostile to experts**.

6.4 Using vec with Other Algorithms

We considered applying `vec` to all algorithms in the Parallelism TS but we felt that it was not clear what that would mean and that assigning an arbitrary meaning would give the programmer a mistaken impression of usability. We might give `vec` a meaning to more algorithms in the future, if and when we identify a reasonable meaning for them.

6.5 Ordered scatters

Previous revisions of this paper proposed a `vec` rule to ensure that "scatters" behave in a way consistent with serial semantics. For example, given:

```
void f() {
    extern float A[], B[];
    extern int P[], Q[];
    for_loop( vec, 0, 1000, [&](int i) {
        A[P[i]] = B[Q[i]];
    });
}
```

This "ordered scatter" rule would have ensured that the result is the same as for replacing `vec` with `seq`, even if there are duplicate values in array `P`. In contrast, this example has undefined behavior if `unseq` is used and `P` has duplicate values, even if all elements of `B` are identical, because there would be unsequenced modifications of the same element of `A`. The inclusion of this rule would have reduced the uses of `ordered_update` and would have made it less likely to create program errors that result in undefined behavior. However, for architectures that support only unordered scatter instructions, the compiler would have to prove, for every store, that collisions are not possible in order to avoid serializing the store. Moreover, although `ordered_update` would not have been needed for an assignment like the above, it would still be needed for read-modify-write operations (e.g., increment), so things like the histogram pattern would not have benefited from this rule. SG1 voted to remove this rule in at the 2016-02 meeting in Jacksonville.

7 C++ Proposed Wording

The proposed edits are with respect to the current Parallelism TS assuming the adoption of P0075 and P0413.

7.1 Feature test macros

Add the following row to Table 1 in section 1.5 [parallel.general.features]

Name	Value	Header
`__cpp_lib_experimental_execution_vector_policy`	201707	`<experimental/algorithm>` `<experimental/execution>`

Editorial note: The format of this section of the TS should probably be changed to match that of the Library Fundamentals TS, which has a 6-column table that includes the name of the specific feature and the document number that proposed it.

7.2 Header <experimental/execution> synopsis

Add the following to section 2.2 [parallel.execution.synopsis] (in nested namespace `execution`):

```
class unsequenced_policy;
class vector_policy;
```

7.3 Add new execution policies

And add the following subsections:

2.x Unsequenced execution policy [parallel.execution.unseq]

`class unsequenced_policy{` *unspecified* `};`

The class `unsequenced_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, e.g., executed on a single thread using instructions that operate on multiple data items.

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

2.y Vector execution policy [parallel.execution.vec]

```
class vector_policy{ unspecified };
```

The class `vector_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. Additionally, such vectorization will result in an execution that respects the sequencing constraints of wavefront application ([parallel.alg.general.wavefront]). [*Note:* The implementation is thus making stronger guarantees than for `unsequenced_policy`, above. – *end note*]

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

7.4 Execution policy objects

Add to [parallel.execution.objects] (in nested namespace `execution`):

```
constexpr unsequenced_policy unseq{};
constexpr vector_policy      vec{};
```

7.5 Exception reporting behavior

Edit 3.1 [parallel.exeptions.behavior] paragraph 2 as shown:

- If the execution policy object is of type ~~class~~ `unsequenced_policy`, `vector_policy`, or `parallel_unsequenced_policy`, `std::terminate` shall be called.

7.6 Wavefront Application

New subsection to add to section 4.1. Shaded text is explanatory and not part of the formal wording.

Wavefront Application [parallel.alg.general.wavefront]

For the purpose of this section, an *evaluation* is a value computation or side effect of an expression, or an execution of a statement. Initialization of a temporary object is considered a subexpression of the expression that necessitates the temporary object.

An evaluation A *contains* an evaluation B if:

- A and B are not potentially concurrent ([intro.races]); and
- the start of A is the start of B or the start of A is sequenced before the start of B; and
- the completion of B is the completion of A or the completion of B is sequenced before the completion of A.

[*Note*: This includes evaluations occurring in function invocations. -- *end note*]

An evaluation A is *ordered before* an evaluation B if A is deterministically sequenced before B. [*Note:* If A is indeterminately sequenced with respect to B or A and B are unsequenced, then A is not ordered before B and B is not ordered before A. The ordered before relationship is transitive. – *end note*]

For an evaluation A ordered before an evaluation B, both contained in the same invocation of an element access function, A is a *vertical antecedent* of B if:

- there exists an evaluation S such that:
  - S contains A, and
  - S contains all evaluations C (if any) such that A is ordered before C and C is ordered before B,
  - but S does not contain B, and
- control reached B from A without executing any of the following:
  - a `goto` statement or `asm` declaration that jumps to a statement outside of S, or
  - a switch statement executed within S that transfers control into a substatement of a nested selection or iteration statement, or
  - a throw [*Note*: even if caught – *end note*], or
  - a `longjmp`.

[*Note*: Vertical antecedent is an irreflexive, antisymmetric, nontransitive relationship between two evaluations. Informally, A is a vertical antecedent of B if A is sequenced immediately before B or A is nested zero or more levels within a statement S that immediately precedes B – *end note*]

The first major bullet above describes what could informally be called "immediately precedes". If A and B are part of the *same* statement, then A is a vertical antecedent of B only if there is nothing sequenced between them. If A and B are part of *different* statements, then A is a vertical antecedent of B if, by popping out zero or more levels of nesting, you find a point where the statement *containing* A immediately precedes B. This is the point of re-convergence after a control-flow divergence.

The second major bullet is needed to handle cases where re-convergence is difficult or impossible to establish. In those cases, the guarantees degenerate to those provided by the `unsequenced_policy` until convergence is re-established at the end of the block containing both the jump statement and the jumped-to statement.

In the following, X_i and X_j refer to evaluations of the *same* expression or statement contained in the application of an element access function corresponding to the i^{th} and j^{th} elements of the input sequence. [*Note:* There might be several evaluations X_k, Y_k, etc. of a single expression or statement in application *k*, for example, if the expression or statement appears in a loop within the element access function. – *end note*]

Horizontally matched is an equivalence relationship between two evaluations of the same expression. An evaluation B_i is *horizontally matched* with an evaluation B_j if:

- both are the first evaluations in their respective applications of the element access function, or
- there exist horizontally matched evaluations A_i and A_j that are vertical antecedents of evaluations B_i and B_j, respectively.

[*Note*: *Horizontally matched* establishes a theoretical *lock-step* relationship between evaluations in different applications of an element access function. – *end note*]

The rules for establishing the horizontally matched relationship match evaluations in one application with corresponding evaluations in a separate application of the element access function. The nature of the rules are such that even nested loops work correctly. For example, given:

```
b;
while ( e )
    stmt;
c;
```

where b_i is horizontally matched with b_j. Intuitively, we would expect the k^{th} evaluation of e_i to be horizontally matched with the k^{th} evaluation of e_j, assuming both evaluations happen. Even if one of the invocations executes *e* more times than the other, all evaluations of e_i and e_j are vertical antecedents of c_i and c_j, respectively, so the horizontally matched relationship is re-established for c_i and c_j.

Let *f* be a function called for each argument list in a sequence of argument lists. *Wavefront application* of *f* requires that evaluation A_i be sequenced before evaluation B_j if $i < j$ and:

- A_i is sequenced before some evaluation B_i and B_i is horizontally matched with B_j, or
- A_i is horizontally matched with some evaluation A_j and A_j is sequenced before B_j.

[*Note: Wavefront application* guarantees that parallel applications i and j execute such that progress on application j never gets *ahead* of application i. – *end note*] [*Note:* The relationships between A_i and B_i and between A_j and B_j are *sequenced before*, not *vertical antecedent*. -- end note]

The two bullets describe the two triangles in Figure 1.

7.7 Effect of execution policies on algorithm execution

To section 4.1.2 [parallel.alg.general.exec], add:

The invocations of element access functions in parallel algorithms invoked with an execution policy of type `unsequenced_policy` are permitted to execute in an unordered fashion in the calling thread, unsequenced with respect to one another within the calling thread.

The invocations of element access functions in parallel algorithms invoked with an execution policy of type `vector_policy` are permitted to execute in an unordered fashion in the calling thread, unsequenced with respect to one another within the calling thread, subject to the sequencing constraints of wavefront application (7.6 [parallel.alg.general.wavefront]) for the last argument to `for_loop` or `for_loop_strided`.

7.8 Header <experimental/algorithm> synopsis

Add the following to 4.3.1 [parallel.alg.ops.synopsis]:

```
namespace std {
namespace experimental {
namespace parallel {
inline namespace v2 {

template<typename F>
  auto no_vec(F&& f) noexcept -> decltype(std::forward<F>(f)());

template<class T>
  class ordered_update_t;

template <class T>
  ordered_update_t<T> ordered_update(T& ref) noexcept;

}}}}
```

7.9 `no_vec`

Add this function to section 4.3 [parallel.alg.ops]:

4.3.x No_vec [parallel.alg.novec]

```
template<typename F>
  auto no_vec(F&& f) noexcept -> decltype(std::forward<F>(f)());
```

Effects: Evaluates `std::forward<F>(f)()`. When invoked within an element access function in a parallel algorithm using `vector policy`, if two calls to `no_vec` are horizontally matched within a wavefront application of an element access function over input sequence S, then the execution of `f` in the application for one element in S is sequenced before the execution of `f` in the application for a subsequent element in S; otherwise there is no effect on sequencing.

Returns: the result of the execution of `f`.

Note: If `f` exits via an exception, then `terminate` will be called, consistent with all other potentially-throwing operations invoked within `vector_policy` execution.

[*Example:*

```
extern int* p;
for_loop( vec, 0, n, [&](int i) {
    y[i] += y[i+1];
    if(y[i]<0) {
```

```
                no_vec([]{
                    *p++ = i;
                });
            }
        });
```

The updates `*p++=i` will occur in the same order as if the policy were `seq`.

– *end example*]

7.10 ordered_update

Add these subsections to section 4.3 [parallel.alg.ops]

4.3.x Ordered update class [parallel.alg.ordupdate.class]

```
template<class T>
class ordered_update_t {
  T& ref_;  // exposition only
public:
  ordered_update_t(T& loc) noexcept
    : ref_(loc) {}
  ordered_update_t(const ordered_update_t&) = delete;
  ordered_update_t& operator=(const ordered_update_t&) = delete;
  template <class U>
    auto operator=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ = std::move(rhs); }); }
  template <class U>
    auto operator+=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ += std::move(rhs); }); }
  template <class U>
    auto operator-=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ -= std::move(rhs); }); }
  template <class U>
    auto operator*=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ *= std::move(rhs); }); }
  template <class U>
    auto operator/=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ /= std::move(rhs); }); }
  template <class U>
    auto operator%=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ %= std::move(rhs); }); }
  template <class U>
    auto operator>>=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ >>= std::move(rhs); }); }
  template <class U>
    auto operator<<=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ <<= std::move(rhs); }); }
  template <class U>
    auto operator&=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ &= std::move(rhs); }); }
  template <class U>
    auto operator^=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ ^= std::move(rhs); }); }
  template <class U>
    auto operator|=(U rhs) const noexcept
```

```
      { return no_vec([&]{ return ref_ |= std::move(rhs); }); }
  auto operator++() const noexcept
    { return no_vec([&]{ return ++ref_; }); }
  auto operator++(int) const noexcept
    { return no_vec([&]{ return ref_++; }); }
  auto operator--() const noexcept
    { return no_vec([&]{ return --ref_; }); }
  auto operator--(int) const noexcept
    { return no_vec([&]{ return ref_--; }); }
};
```

An object of type `ordered_update_t<T>` is a proxy for an object of type `T` intended to be used within a parallel application of an element access function using a policy object of type `vector_policy`. Simple increments, assignments and compound assignments to the object are forwarded to proxied object, but are sequenced as though executed within a `no_vec` invocation. [*Note*: The return-value deduction of the forwarded operations results in these operators returning by value, not reference. This formulation prevents accidental collisions on accesses to the return value. – *end note*]

4.3.x Ordered update function template [parallel.alg.ordupdate.func]

```
template <class T>
  ordered_update_t<T> ordered_update(T& loc) noexcept;
```

Returns: `{ loc }`.

8 Acknowledgement

Olivier Giroux provided the ideas behind “horizontally matched” and “vertical antecedent”.

9 References

[1] CONVEX Architecture Handbook, Document No. 080-000120-000, PDF page 222, implies that the scatter instruction has serial semantics.

[2] Lee Higbie, Vectorization and Conversion of Fortran Programs for the CRAY-1 (CFG) Compiler, Undated, but seems to be from Cray-1 timeframe. PDF page 15 describes vectorization of a loop with a forward lexical dependence.

[3] Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual, Section 2.6 says “Otherwise, the Cray X1 system guarantees that B will reference memory after A only if: ... A and B are elements of the same ordered vector scatter or zero-stride vector store.”

[4] Michael Wolfe, “Loop Skewing: The Wavefront Method Revisited”, Int. J. of Parallel Programming 15(4), 1986, pp. 279-293.

[5] Robert Geva and Clark Nelson, “Language Extensions for Vector loop level parallelism”, WG21 N4237.

[6] Arch D. Robison, Pablo Halpern, Robert Geva and Clark Nelson, “Template Library for Index-Based Loops”, WG21 P0075R1.

[7] Jared Hoberock (editor), “Working Draft, Technical Specification for C++ Extensions for Parallelism Version 2”, WG21 N4578.

[8] Pablo Halpern, “Updating Parallel Execution Policy Names in the Parallelism TS”, WG21 P0413r0.