|  |  |
| --- | --- |
| Document number: | D0076R1 |
| Date: | 2016-02-09 |
| Project: | Programming Language C++, SG1 |
| Reply to: | Arch D. Robison <arch.robison@intel.com>  Pablo Halpern <pablo.g.halpern@intel.com >  Robert Geva <robert.geva@intel.com>  Clark Nelson <clark.nelson@intel.com> |

**Vector and Wavefront Policies**

# Introduction

This paper proposes adding two new execution policies to the Parallelism TS and P0075R1. These policies add support for execution with relaxed sequencing restricted to a single OS thread:

* A unsequenced\_execution\_policy and constant unseq analogous to the other policy types and constants in the Parallelism TS, with sequencing semantics similar to parallel\_vector\_execution\_policy, but limited to a single OS thread.
* A vector\_execution\_policy 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 P0075R1.

The first policy is strictly weaker 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.[[1]](#footnote-2)

seq

par

**unseq**

par\_vec

**vec**

No compiler extensions are necessary for correct implementation. An implementation is free to implement any policy higher on the lattice via a policy lower on the lattice, although it is not currently clear what the vec policy might mean for most of the STL algorithms, other than ones that iteratively apply a single function. Hidden vendor-specific hooks may aid an optimizing implementation of for\_loop reductions with either policy.[[2]](#footnote-3)

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.

The Parallelism TS offers the par\_vec policy, and there is some interest in a variant that restricts execution to a single thread. The result of such a restriction is our unseq policy. Alas, this policy, though **sufficient** for vectorization (exploiting vector hardware), is excessively permissive and fails to express the **necessary** requirements. The gap between sufficient and necessary contains many vectorizable loops of practical interest. As defined in N4507, par\_vec 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\_vec to a single thread still allows permissive interleaving that would give undefined semantics to loops in the aforementioned gap.

Here is a short example that falls in the gap, using for\_loop from P0075R1 with vector\_execution\_policy:

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 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.

# Changes since R0

* Changed formal specification to vertical/horizontal instead of LIFO contexts.
* Added ordered\_update and its helper class ordered\_update\_t.
* Changed vec\_off(f) to return result of f() instead of discarding it.

# Wavefront Application

Our proposed vector\_execution\_policy gives programmers classic “vector loop” evaluation order guarantees when used with function template for\_loop from PR0075R1. We abstract the evaluation order by defining “wavefront[[3]](#footnote-4) application*”*. Intuitively, the *wavefront application* of a function *f* over a sequence of argument listsapplies *f* to each argument list in a way that keeps preceding applications from falling behind later application. This property distinguishes our vector\_execution\_policy from our unsequenced\_execution\_policy. The property has two benefits:

* It enables exploiting “forward dependencies”, a common technique in classic vector codes.
* It implies that vector\_execution\_policy is safe to use on any loop that could be auto-vectorized.

For example, consider:[[4]](#footnote-5)

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\_execution\_policy (or the existing parallel\_execution\_policy or parallel\_vector\_execution\_policy) the programmer would need to fission the loop into two loops, with consequent penalty of increasing consumption of memory bandwidth.

We optionally propose vec rules 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]];

});

}

The optional rule would ensure that the result is the same as for replacing vec with seq, even if there are duplicate values in array P.[[5]](#footnote-6) 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.

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 unnecessary relaxations that would be harmful for some loops.

## 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. Furthermore, unstructured control flows (gotos and switches like “[Duff’s device](https://en.wikipedia.org/wiki/Duff%27s_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 Section 4. For understanding 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);

}

our 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 Section 5.

## Ordering Rules for Wavefront Application

The invocations of element access functions in our for\_loop template from PR0075R1 invoked with an execution policy of type vector\_execution\_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 5.

Figure 1 sketches the rule for the ith and jth 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.

Ai

Aj

Bj

′

**Figure 1**

Ai

Bi

~~B~~

Bj

~~B′~~

′

Thus the jth iteration cannot get ahead of the ith iteration.

Our optionally proposed rule for scatters can be summarized graphically as:

Xc,i

Xc,j

**Figure 2**

The dots represent side effects. The ellipses circumscribe side effects of an expression. The inner ellipses circumscribe subexpressions. Lines and arrows have the same meaning as in the first picture.

#### Comparison with Evaluation Order Rules from N4237

N4237’s rules are presented in the context of loops. Since our for\_loop from PR0075R1 takes the loop body as a function, our rules are phrased in terms of applying that function.

Our rule 1 is essentially a narrowing of rule 1 from p. 7 of N4237. Our revision narrows it in two ways:

* Only applications i and j are used. Evaluations in other applications (“k” in N4237) have no impact on the sequencing relationships between evaluations in applications i and j.
* Only evaluations within the same context can be used to establish new sequencing relationships.

The latter narrowing is critical for enabling vector evaluation of nested loops. Consider:

for\_loop( vec, 0, 2, [&](int i) {

for( int m=0; m<2; ++m )

A[m][i] = 1;

});

Our definition of horizontal matching lets our rules see the three evaluations of m<2 and two evaluations of A[m][i] as five separate evaluations, as if the inner loop was unrolled. The solid arrows in Figure 3 shows 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 arrows inferable via transitive closure.

**Figure 3**

m<2

a[m][i]

m<2

a[m][i]

m<2

m<2

a[m][i]

m<2

a[m][i]

m<2

Without context distinctions, the expression m<3 is sequenced before A[m][i] 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.

The difference in power between our rule 1 and the rules in N4237 can be summarized as:

* If a function executes no iteration statements and no gotos, they are equivalent.
* If a function executes no iteration statements, but does execute gotos, our rules have more relaxed sequencing than N4237.
* If a function executes iteration statements, but no gotos, and no switches that jump into selection or iteration statements, our rules retain classic vector evaluation order, whereas N4237 requires serialization.
* If a function executes iteration statements, and has gotos or switches that jump into iteration statements, our rules retain classic vector evaluation to some degree, whereas N4237 requires serialization.

## vec\_off

It is sometimes useful to force serial sequencing of a region of code. We define a template function vec\_off 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) {

vec\_off([]{

\*p++ = i;

});

}

});

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

The syntactic definition of vec\_off is:

template<typename F>

void vec\_off( F&& f ) {f();}

but subject to a sequenced-before variant of our rule 2, where an entire call to vec\_off acts as a “direct side effect”. Using the notation and assumption i<j from the other two rules, the semantics of vec\_off are:

1. For every call X of vec\_off, the invocation Xc,i is sequenced before the invocation Xc,j, if both are evaluated.

## Extensibility of Policies

Though we don’t propose it for standardization at this time, we note that vector\_execution\_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\_execution\_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[[6]](#footnote-7) 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.

# 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](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4238.pdf) 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);

});

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

## The promise and disappointments of the explicit model

Conceptually, the explicit 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 and in a way that we determined was too arcane and would discourage the use of vectorization.

There were two classes of expression 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, an examples of the first expression is:

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

The first expression requires that A[1] not be modified until its value has been read in iteration 0. With the explicit 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;

though, with a small helper function, this could be simplified to:

A[i] = 2\*parallel::wavefront\_rvalue(A[i + 1]);

The second class of expression is more problematic:

A[B[i]] = expr;

Given that B[i] is not necessarily unique for all i, the only way to achieve consistent results is to require strict (left-to-right) sequencing of the assignment operation. There is no place where one could insert a wavefront ordering point that would make this work. The way to get the correct result would be something like:

auto tmp = expr;

auto& ref = A[B[i]];

parallel::wavefront\_off([&]{ ref = tmp; });

Again, a helper function could simplify things:

parallel::wavefront\_assign(A[B[i]], expr);

or

parallel::wavefront\_assign(A[B[i]]) = expr;

Not only are the above workarounds somewhat ugly and potentially error prone, but it shows warts that are exposed when the explicit model is examined closely and 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.

## 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**.

## 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 we identify a reasonable meaning for them.

# C++ Proposed Wording

The proposed edits are with respect to the current Parallelism TS and PR0076R0.

**Header <experimental/execution\_policy> synopsis**

class vector\_execution\_policy;

class unsequenced\_execution\_policy;

**Add section after section on Parallel+Vector execution policy**

class vector\_execution\_policy{ unspecified };

The class vector\_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, but must respect wavefront evaluation order

class unsequenced\_execution\_policy{ unspecified };

The class unsequenced\_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.

**Execution policy objects**

Add:

constexpr vector\_execution\_policy vec{};

constexpr unsequenced\_execution\_policy unseq{};

**Exception reporting behavior**

Edit 3.1 paragraph 2 as shown:

If the execution policy object is of type class vector\_execution\_policy, unsequenced\_execution\_policy, or parallel\_vector\_execution\_policy, std::terminate shall be called.

To “Effect of execution policies on algorithm execution”, add:

The invocations of element access functions in parallel algorithms invoked with an execution policy of type unsequenced\_execution\_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 for\_loop or for\_loop\_strided invoked with an execution policy of type vector\_execution\_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 constraints of wavefront application order for the last argument to for\_loop or for\_loop\_strided.

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

**Wavefront Application**

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

An evaluation C of an expression is *sequenced-between* two evaluations A and B (of possibly different expressions) if A is sequenced before C and C is sequenced before B.

An evaluation A *contains* an evaluation B if evaluation of B occurs as part of evaluation of A. [*Note*: This includes evaluations occurring in function invocations.]

*Vertical antecedent* is an irreflexive, nonsymmetric, nontransitive relationship between two evaluations. For an evaluation A sequenced 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
  + contains all said evaluations between A and B,
  + but does not contain B, and
* control reached B from A without executing any of the following:
  + a goto statement 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.

The first and second major bullets above describes the notion related to “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 third 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\_execution\_policy until convergence is re-established at the end of the block containing both the jump statement and the jumped-to statement.

*Horizontally matched* is an equivalence relationship between two evaluations of the same expression. Within two applications of an element access function, given evaluations Bi and Bj of an expression, Bi is *horizontally matched* with Bj if:

* neither evaluation has a vertical antecedent, or
* there exist evaluations Ai and Aj of a (possibly different) expression that are the vertical antecedents of evaluations Bi and Bj respectively, and A is a horizontal antecedent of A’.

[*Note:* Horizontal antecedent does not imply a sequenced-before relationship. --  *end note*]

The rules for establishing the horizontal antecedent relationship match evaluations in one application with corresponding evaluations in a logically-later application of the element access function. The nature of the rules are such that even nested loops work correctly. For example, given[[7]](#footnote-8):

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

where *b* is a horizontal antecedent of *b’*. Intuitively, we would expect the kth evaluation *e* to be the horizontal antecedent of the kth evaluation of *e*′, assuming both evaluations happen. Even if one of the invocations executes *e* more times than the other, the last evaluations of *e* and *e*’ that is common to both invocations are vertical antecedents of *c* and *c*’, so the horizontal antecedent relationship is re-established for *c* and *c*’.

Let *f* be a function called for each argument list in a sequence of argument lists. Let Ai and Bj denote two evaluations where the context of Ai precedes the context of Bj. *Wavefront application* of *f* requires that Ai be sequenced before Bj if:

* there exists an evaluation Bi horizontally matched with Bj such that Ai is sequenced before Bi, or
* there exists an evaluation Ai horizontally matched with Aj such that Ai is sequenced before Bj .

The two bullets describe the two triangles in Figure 1. Note that the vertical relationships are *sequenced before*, not *vertical antecedent.*

**Optional clause for supporting scatter pattern**

The *direct side effects* of a an expression X are those caused by evaluating X, but not including side effects caused by evaluating its sub-expressions. For any two evaluations Ai and Aj such that Ai is a horizontal antecedent of Aj, all direct side effects in Ai are sequenced before all direct side effects in Aj.

This clause allows for code such as:

U[V[i]] = expr(i);

to produce deterministic results even if V[i] contains duplicate elements (sometimes called the *overlapping scatter pattern*). If this clause is adopted, we will also want a library function, unordered\_update, having a syntax similar to ordered\_update, that relaxes this guarantee and allows the generation of faster code on architectures with scatter instructions that do not support ordered writes. The unrdered\_update function should be used only when V[i] is known not to contain duplicates.

**New subsection to add to section 4.1:**

namespace std {

namespace experimental {

namespace parallel {

inline namespace v2 {

template<typename F>

auto vec\_off(F&& f) -> decltype(f());

}}}}

*Effects*: Evaluates f() subject to the constraint that given a function call expression call X of the form vec\_off(*expr*), if two evaluations of X are dynamically inside an invocation of for\_loop with vector\_execution\_policy, the invocation Xc,i is sequenced before the invocation Xc,j, if both are evaluated and i<j.

*Remarks*: Returns result of f().

**New subsection to add to section 4.1:**

namespace std {

namespace experimental {

namespace parallel {

inline namespace v2 {

template<class T>

class ordered\_update\_t {

T& *ref*; // exposition only

public:

ordered\_update\_t(T& loc);

T operator=(const T& rhs);

template<class U>

   T operator+=(const U& rhs);

  T operator++();

  T operator++(int);

};

template <class T>

ordered\_update\_t<T> ordered\_update(T& ref);

}}}}

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 simd\_off. 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*

ordered\_update\_t(T& loc);

*Effect*: Initialize *ref* with loc.

T operator=(const T& rhs);

template<class U> T operator+=(const U& rhs);

T operator++();

T operator++(int);

Effect*:* Perform the corresponding operation ref=rhs, ref+=rhs, ref++, or ++ref as if all accesses to ref are wrapped simd\_off. Evaluation of rhs is not necessarily wrapped. For example, operator+= can behave as if written:

    template <class U>

    T operator+=(const U& rhs) {

        U value = rhs;

        return vec\_off([&]{return this->ref + value;});

    }

*Returns:* value returned by evaluation above.

template <class T>

ordered\_update\_t<T> ordered\_update(T& ref);  
*Returns*: ordered\_update(ref)

If the implicit scatter rule is included, then we will want a way to turn it off when it is not required. If the committee goes that direction, we will add a function unordered\_update that turns the rule off, similar in style to how ordered\_update turns the rule on.

# References

1. [CONVEX Architecture Handbook](http://bitsavers.informatik.uni-stuttgart.de/pdf/convex/080-000120-000_CONVEX_Architecture_Handbook_1984.pdf), 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](http://bitsavers.informatik.uni-stuttgart.de/pdf/cray/CFT/2240207_Vectorization_and_Conversion_of_Fortran_Programs_for_the_CFT_Compiler.pdf), 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](http://docs.cray.com/books/S-2314-51/html-S-2314-51/x3724.html) 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.

1. We also recommend that the existing par\_vec be renamed par\_unseq since the top lattice point’s relaxations are the union of the relaxations of par and unseq, or dually the top lattice point’s guarantees are the intersection of the guarantees of par and unseq. [↑](#footnote-ref-2)
2. In particular, we implemented a performant version of vector reductions for for\_loop in LLVM by adding special intriniscs. [↑](#footnote-ref-3)
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]. [↑](#footnote-ref-4)
4. The example is a toy, but the dependence pattern is similar to those in staggered finite-time finite-difference codes. [↑](#footnote-ref-5)
5. As far as we can tell, vector hardware with support for scatter operations usually has at least an option for ordered scatters. [↑](#footnote-ref-6)
6. Yes, 9 and not 8. The wavefront semantics prevent the oldest iteration in flight from getting behind the newest iteration in flight. [↑](#footnote-ref-7)
7. The notation here is informal. In the code, the letters *b, c, e* denote expressions. In the discussion text they refer to the evaluations of those expressions. [↑](#footnote-ref-8)