Document number: P0076R0
Date: 2015-9-25
Project: Programming Language C++, Library Working Group
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>

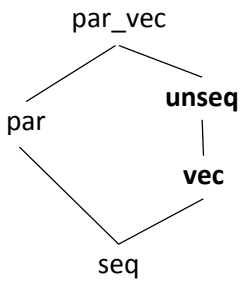
Vector and Wavefront Policies

1 Introduction

This paper proposes adding two new execution policies to the Parallelism TS and P0075R0. These policies add support for execution with relaxed sequencing restricted to a single OS thread:

- A `unseq_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 `vec_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 P0075R0.

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]



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]

[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`.

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

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 P0075R0 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`.

2 Wavefront Application

Our proposed `vec_execution_policy` gives programmers classic "vector loop" evaluation order guarantees when used with function template `for_loop` from PR0075R0. We abstract the evaluation order by defining "wavefront[3] application". Intuitively, the *wavefront application* of a function *f* over a

[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].

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_execution_policy` from our `unseq_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]

```
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 `unseq_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.

Furthermore, our `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]];
    });
}
```

our rules ensure that the result is the same as for replacing `vec` with `seq`, even if there are duplicate values in array `P`.[5] 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-

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

[5]As far as we can tell, vector hardware with support for scatter operations usually has at least an option for ordered scatters.

interleave optimizations, without introducing unnecessary relaxations that would be harmful for some loops.

2.1 Contexts

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 "contexts" that distinguish evaluating the same expression during different trips though a loop or in different invocations of a callee. Furthermore, unstructured control flows (gotos and switches like "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).

Contexts are fully defined and further explained in Section 4. For understanding the next section, it suffices to know that a context is a sequence of elements, where each element can be an integer, NaN, or lexical id of a call site. Every context begins with a call site id. The integers indicate loop nesting and number of times each loop has executed. A NaN denotes potential mischief with gotos.

2.2 Ordering Rules for Wavefront Application

The invocations of element access functions in our `for_loop` template from PR0075R0 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 explained below.

Let *f* be a function called for each argument list in a sequence of argument lists. Let c and d denote (possibly equal) contexts that do not contain a NaN. Let $X_{c,i}$ denote the evaluation of expression X within context c on the ith call, and likewise for $Y_{d,i}$, $X_{c,j}$, and $Y_{d,j}$. 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. *Wavefront application* of *f* requires that if i<j then:

1. For every expression X and Y, if $X_{c,i}$ is sequenced before $Y_{d,i}$ or $X_{c,j}$ is sequenced before $Y_{d,j}$, then $X_{c,i}$ is sequenced before $Y_{d,j}$ if both are evaluated.
2. For every expression X, all direct side effects in $X_{c,i}$ are sequenced before all direct side effects in $X_{c,j}$, if both are evaluated.

Rule 1 can be summarized graphically as shown in Figure 1.

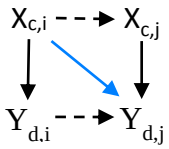


Figure 1

The black arrows denote the hypothesis; the blue arrows denote the conclusion. Solid arrows denote sequenced-before. Dashed arrows denote evaluations of the same expression in the same context, but in different applications.

Rule 2 can be summarized graphically as:

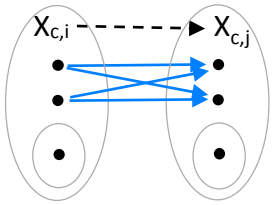


Figure 2

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

2.2.1.1 Comparison with Evaluation Order Rules from N4237

N4237's rules are presented in the context of loops. Since our `for_loop` from PR0075R0 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 context 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 the resulting sequenced-before relationships. As traditional with such diagrams, we omit arrows inferable via transitive closure.

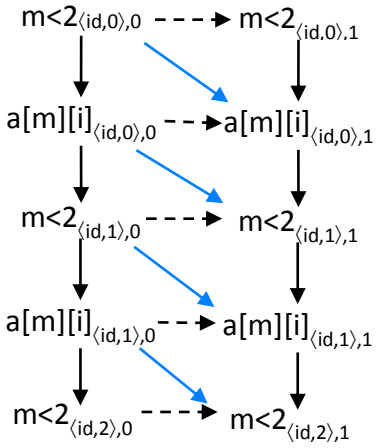


Figure 3

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 iteration statements, our rules retain classic vector evaluation order, whereas N4237 requires serialization.
- If a function executes iteration statements, and switches that jump into iteration statements or gotos, our rules retain classic vector evaluation to some degree, whereas N4237 requires serialization.

2.3 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:

3. For every call X of `vec_off`, the invocation $X_{c,i}$ is sequenced before the invocation $X_{c,j}$, if both are evaluated.

2.4 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] 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.

3 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();
```

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

```
    A[i] = f(tmp);
});
```

3.1 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, 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.

3.2 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.

3.3 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**.

3.4 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 we identify a reasonable meaning for them.

4 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 unseq_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 unseq_execution_policy{ unspecified };
```

The class `unseq_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 unseq_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`, `unseq_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 `unseq_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

A *context* is a sequence of elements. Each element may be an integer, NaN, or lexical id of a call site. A lexical id can be anything that distinguishes one call site (possibly implicit) from another. [*Note*: contexts are a mathematical construct that assist definition of the sequencing constraints for `vector_execution_policy`. – *end note*]

A lexical id can be anything that distinguishes one call site from another. For example, in the expression `f()+f()` there are at least two distinct call sites, one for each invocation of `f()`, and perhaps more for implicitly invoked constructors, conversion operators, or destructors. Intuitively, the current context summarizes the call chain from an invocation of `for_loop` and which iteration each enclosing iteration statement is executing.

An application's context is the empty sequence ⟨⟩ before the function is applied. The context is updated like a LIFO during execution by the following rules:

- When entering an iteration statement, if via a `goto` or `switch` statement, push NaN. Otherwise push 0.
- When leaving the substatement of an iteration statement, increment the last element unless it is NaN.
- When leaving an iteration statement, pop the last element.

The three rules above ensure that there is exactly one context element per active iteration statement, and the elements reflect the iteration nesting and trip counts.

- Replace the last element with NaN when either of the following occur while executing a `goto`:
  - When leaving a substatement of a statement via a goto and entering another substatement of that statement via the same goto.
  - The labeled statement specified by the `goto`'s identifier is reached.

Use of `goto`, or unstructured use of `switch`, introduces a NaN until control leaves the innermost iteration statement or function containing both the `goto` and label. The rules for goto above are equivalent to the following recipe: Find the context element pushed by the innermost iteration statement that contains both the `goto` and its target label, or if there is none, the context element pushed when the function was entered. Replace that element with NaN.

- The initial context for a called function is a copy of the caller's context, or the empty sequence if the caller has no context, appended with the call site's lexical id in either case.

[*Note*: The last rule applies to the applied function itself, thus guaranteeing that the context is non-empty during evaluation of an applied function. – *end note*]

The last rule copies the caller's context, instead of using it in place, to deal with nested uses of `vec` inside permissively sequenced constructs.

Contexts are unordered. Our rules that uses contexts only depend on the notion of equal contexts, and treat a context containing NaN as if it were unequal to any other context, even itself. While a NaN is present, the `vec` policy temporarily acts like the `unseq` policy (i.e., the sequencing guarantees are relaxed).

Let *f* be a function called for each argument list in a sequence of argument lists. Let c and d denote (possibly equal) contexts that do not contain a NaN. Let $X_{c,i}$ denote the evaluation of expression X within context c on the ith call, and likewise for $Y_{d,i}$, $X_{c,j}$, and $Y_{d,j}$. 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. *Wavefront application* of *f* requires that if i<j then:

1. For every expression X and Y, if $X_{c,i}$ is sequenced before $Y_{d,i}$ or $X_{c,j}$ is sequenced before $Y_{d,j}$, then $X_{c,i}$ is sequenced before $Y_{d,j}$ if both are evaluated.
2. For every expression X, all direct side effects in $X_{c,i}$ are sequenced before all direct side effects in $X_{c,j}$, if both are evaluated.

New subsection to add to section 4.1:

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

template<typename F>
void vec_off(F&& 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 $X_{c,i}$ is sequenced before the invocation $X_{c,j}$, if both are evaluated and i<j.

Remarks: If *f* returns a result, the result is ignored.

5 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.