|  |  |
| --- | --- |
| Document number: | D0075r1 |
| Date: | 2016-01-29 |
| 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> |

**Template Library for Parallel For Loops**

# Introduction

In order to maximally exploit parallelism, a parallel loop construct must be able to predict iterations, even before the loop begins executing. Thus, parallel loops are necessarily more restrictive than the general-purpose (serial) for loop at the C++ language level. The looping construct in the existing parallelism TS, parallel::for\_each, while convenient for traversing a sequence of elements in parallel, would require tricky and convoluted code to handle a number of common patterns:

* Traversing multiple sequences in the same loop is difficult, e.g., A[i] = B[i].
* Referring to elements before or after the current element, e.g., iter[0] = iter[1].
* Performing computations based on the position in the loop, e.g., A[i] += i % 2 ? 1 : -1;

Critically, the patterns above are critical for exploiting vector parallelism, as described in P0076. This paper proposes support for all aforementioned patterns as a pure-library extension. Our proposal is pure-library extension of the Parallelism TS, and adds support for indexed-based loops with reduction and induction variables.

# Changes since r0

* Added for\_loop\_n and for\_loop\_n\_strided.
* Added more rationale for reduction and inductions to store final values as side effects.

# Summary of proposal

The proposal adds the following new function templates to the Parallelism TS:

* **for\_loop**, **for\_loop\_strided**, **for\_loop\_n**, and **for\_loop\_n\_strided** implement loop functionality over a range specified by integral or iterator bounds. For the iterator case, it resembles for\_each from the Parallelism TS, but leaves to the programmer when and if to dereference the iterator.
* **reduction** provides a flexible way to specify reductions in conjunction with for\_loop.
* **reduction\_plus**, **reduction\_mutiplies**, ... etc. creating reduction descriptors for common cases such as addition, multiplication, etc.
* **induction** provides a flexible way to specify indices or iterators that vary linearly with the primary index of the loop.

Here is a short example:

void saxpy\_ref(int n, float a, float x[], float y[]) {

for\_loop( seq, 0, n, [&](int i) {

y[i] += a\*x[i];

});

}

The call to for\_loop is equivalent to:

void saxpy\_ref(int n, float a, float x[], float y[]) {

for( int i=0; i<n; ++i )

y[i] += a\*x[i];

}

The loop can be parallelized by replacing seq with par. Our library interface permits the “loop index” to have integral type or be an iterator. As with the current Parallelism TS, the iterator case does not require a random-access iterator. For example, for\_loop enables the following general implementation of for\_each from the Parallelism TS.

template<class ExecutionPolicy,

class InputIterator, class Function>

void for\_each(ExecutionPolicy&& exec, InputIterator first, InputIterator last,

Function f) {

for\_loop( exec, first, last, [&](InputIterator i){f(\*i);} );

}

When exec is not sequential\_execution\_policy, random-access iterators may yield better performance, because unaggressive implementations are likely to fall back to using a serial loop for other kinds of iterators.

## Range and counted variants

For each proposed function template, there are two variants: A range-based version and a counted version. The normal (range based) version takes a starting index (or iterator) and an ending index (or iterator) and iterates over the half-open range [start, end). The counted variants take a starting index (or iterator) and a count of iterations. Because the two variants are nearly impossible to distinguish using overloading alone, the latter have “\_n” in their names, in the same way as for\_each and for\_each\_n are named.

## Strided variants

Our proposal also adds a function template for strided loops. Though these can be expressed from unit-stride loops and mathematical machinations, we think code is clearer when loops can be expressed in natural strided form. To alleviate template overload trickiness and potential hazards, the function templates for strided loops have different names from their unstrided variants. The situation is somewhat akin to the motivations for giving for\_each and for\_each\_n different names.

The stride parameter follows the second bound on the index space. The stride=3 example below sets c[10], c[13], c[16], c[19] to true.

for\_loop\_strided( par, 10, 20, 3, [&](int k) {

c[k] = true;

});

Negative strides are allowed. The following sets the same elements of c to true as the previous example.

for\_loop\_strided( par, 19, 9, -3, [&](int k) {

c[k] = true;

});

## Reductions

A reduction is the parallel application of a mutating operation on a variable in such a way that races are avoided (without locks) and the final value of the variable is the same as it would be if the computation were performed serially. This is accomplished by giving each concurrent task a different *view* of the variable and combining the separate views at the end of the computation.

The for\_loop template allows specification of one more reduction variables, with a syntax inspired by OpenMP, but done with a pure library approach. Here is an example:

float dot\_saxpy(int n, float a, float x[], float y[]) {

float s = 0;

for\_loop( par, 0, n,

reduction(s,0.0f,std::plus<float>()),

[&](int i, float& s\_) {

y[i] += a\*x[i];

s\_ += y[i]\*y[i];

});

return s;

}

Here, reduction is a function that returns an implementation-specific *reduction object* that specifies three things:

* a reduction lvalue s
* the identity value for the reduction operation
* the reduction operation

In the lambda expression, i is a value of the loop index, and s\_ is a reference to a private partial sum. There is one such reference for each reduction argument to for\_loop, and association is positional. (We suspect that in practice, most programmers will name the local reference just s.) The example is equivalent, except with more relaxed sequencing and reduction order, to the following serial code:

float serial\_dot\_saxpy (int n, float a, float x[], float y[]) {

float s = 0;

for( int i=0; i<n; ++i ) {

y[i] += a\*x[i];

s += y[i]\*y[i];

}

return s;

}

For convenience, we supply shorthand functions for common reductions. For example:

reduction\_plus(s)

is equivalent to:

reduction(s,0.0f,std::plus<float>())

## Inductions (Linear Variables)

A linear induction value is a value that varies in a predictable way (typically linearly) with the loop iteration count. Although an induction value can always be computed from the iteration count, requiring the programmer to do so is inconvenient and error prone.

The for-loop template allows specification of induction variables, using a scheme somewhat similar to that for reduction variables. Here is an example with three induction variables:

float\* zipper(int n, float\* x, float \*y, float \*z) {

for\_loop( par, 0, n,

induction(x),

induction(y),

induction(z,2),

[&](int i, float\* x\_, float\* y\_, float\* z\_) {

\*z\_++ = \*x\_++;

\*z\_++ = \*y\_++;

});

return z;

}

Here induction is a function that returns an implementation-specific type that specifies two things:

* An initial value (lvalue or rvalue) for the induction (e.g. x)
* An optional stride for that value. Here the stride is implicitly 1 for x and y, and explicitly 2 for z.

In the lambda expression, i is a value of the loop index, and x\_, y\_, z\_ are initialized with x+i, y+i, and z+2\*i respectively. As with reduction arguments, association is positional. A function can have both reduction and induction arguments. When the for\_loop finishes, any lvalues used to initialize the inductions are set to the same live-out values as if the loop had been written sequentially. For example, the following serial code returns the same value as the previous example:

float\* zipper(int n, float\* x, float \*y, float \*z) {

for( int i=0; i<n; ++i ) {

\*z++ = \*x++;

\*z++ = \*y++;

}

return z;

}

# Alternative Designs

## Leaving out inductions

It is possible to leave induction out and rely on users to write the equivalent math. However, doing so complicates parallelizing codes. We note that OpenMP has linear clauses for similar reason.

The current Parallel STL has support for reductions. However, these are tightly tied to specific algorithms and require “tuple-fying” values (and defining reduction operations on the tuples) for code that needs to perform more than one reduction. Our approach brings the flexibility that OpenMP users have enjoyed from the start.

## Eliminating the “live out” value of inductions and reductions through side effects

During the October 2014 meeting in Kona, there was concern that the lvalue passed into the reduction and induction functions is modified (i.e., there is a side effect) when the for\_loop completes. The argument was made that this could cause races via “action at a distance,” and we were encouraged to consider alternative designs such as returning the final values as a tuple.

Our analysis indicates that the risk of races is no more significant than any other function call that takes an argument by reference. The for\_loop itself does not modify the reduction or induction variable concurrently, and the user should be aware that the value is modified by the call to for\_loop. In general, induction and reduction variables will be local variables in the same scope as the for\_loop function call, and will not be shared by other threads or parallel tasks.

Furthermore, returning the values as a tuple is cumbersome, error-prone, and just as dangerous as modifying them through a reference. Consider:

int a = 100;

float b = 1.0;

tie(a, b) = parallel::for\_loop(0, 100, reduction\_plus(b), induction(a),

[&](int i, float& b, int a){

// Code that uses i and a and updates b.

});

Because of the limitation of using a library syntax, the reduction and inductions variables must be specified at least twice: (1) as arguments to reduction and induction and (2) as arguments to the lambda expression. Returning the final values as a tuple would require that they be specified a third time, and, in fact, the above code has an error in that the tie expression has its arguments reversed. Moreover, the tie expression stores the references in a way that is no less race prone than the original proposed formulation. For these reasons, we elected to leave the definitions of reduction and induction unchanged in this respect.

# Future enhancements

## More general reductions

This proposal does not describe a *concept* for the value returned by the reduce function template. It might be desirable in the future for users to be able to create more sophisticated reductions, e.g., that use allocators or generate identity objects in interesting ways.

## Non-commutative reductions

Some parallel languages (such as Cilk Plus) allow reductions on non-commutative operations such as list append. The runtime library is required to combine partial results such that the left-to-right ordering is preserved. For thread-parallelism, this presents very little overhead, but for vectorization the overhead can be significant. In this proposal, we do not make any such guarantees, but a future proposal might add reductions that are specifically tagged as non-commutative.

# C++ Proposed Wording

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

**Reduction Support for for\_loop [Addition to Non-Numeric Parallel Algorithms]**

Reduction objects add a flexible reduction capability to std::for\_loop. Reduction objects have implementation-specific types, and are created by the function template reduction.

namespace std {

namespace experimental {

namespace parallel {

inline namespace v2 {

// General form for reduction

template<typename T, typename BinaryOp>

*see-below* reduction( T& var, T&& identity, BinaryOp&& op );

// Shorthand for plus reduction

template<typename T>

*see-below* reduction\_plus( T& var );

// Shorthand for multiplies reduction

template<typename T>

*see-below* reduction\_multiplies( T& var );

// Shorthand for bit\_and reduction

template<typename J>

*see-below* reduction\_bit\_and( J& var );

// Shorthand for bit\_or reduction

template<typename J>

*see-below* reduction\_bit\_or( J& var );

// Shorthand for bit\_xor reduction

template<typename J>

*see-below* reduction\_bit\_xor( J& var );

// Shorthand for min reduction

template<typename T>

*see-below* reduction\_min( T& var );

// Shorthand for max reduction

template<typename T>

*see-below* reduction\_max( T& var );

}}}}

Each function returns a “reduction object” having a *value type* of T and encapsulating a reference to *var*, an *identity* value for the reduction, and a *reduction-op*. See description of for\_loop for how these are used. The implicit *identity* and *reduction-op* are shown in Table 1.

Table 1 -- Reduction identities and reduction-ops

|  |  |  |
| --- | --- | --- |
| *function* | *identity* | *reduction-op* |
| reduction\_plus | T() | x+y |
| reduction\_multiplies | T(1) | x\*y |
| reduction\_bit\_and | ~(T()) | x&y |
| reduction\_bit\_or | T() | x|y |
| reduction\_bit\_xor | T() | x^y |
| reduction\_min | *var* | std::min(x,y) |
| reduction\_max | *var* | std::max(x,y) |

[*Example:*

The following code updates each element of y and sets s to the sum of the squares.

float s = 0;

for\_loop( vec, 0, n,

reduction(s,std::plus<float>()),

[&](int i, float& t) {

y[i] += a\*x[i];

t += y[i]\*y[i];

}

});

--*end example*]

**Induction objects**

Induction objects add a flexible capability to specify secondary index variables to std::for\_loop.

namespace std {

namespace experimental {

namespace parallel {

inline namespace v2 {

template<typename T>

*see-below* induction( T&& var );

template<typename T, typename S>

*see-below* induction( T&& var, S stride );

}}}}

Each function returns an “induction object” that encapsulates a *var*, and optionally a *stride.* If T is an rvalue type, then only the initial value of *var* is used – *var* is not modified on return from the for\_loop. If T is a const lvalue type, the program is ill-formed. See description of for\_loop for how these are used.

**For loop [Addition to Non-Numeric Parallel Algorithms]**

**In <experimental/algorithm> synopsis [parallel.alg.ops.synopsis], add**:

namespace std {

namespace experimental {

namespace parallel {

inline namespace v2 {

template<typename Policy, typename I, typename... Rest>

void for\_loop (Policy&& policy, common\_type\_t<I> first, I last,

Rest&&... rest );

template<typename Policy, typename I, typename S, typename... Rest>

void for\_loop\_strided(Policy&& policy, common\_type\_t<I> first, I last,

S stride, Rest&&... rest );

template<typename Policy, typename I, typename Size, typename... Rest>

void for\_loop\_n(Policy&& policy, I first, Size n, Rest&&... rest );

template<typename Policy, typename I, typename Size, typename S,

typename... Rest>

void for\_loop\_n\_strided(Policy&& policy, I first, Size n, S stride,

Rest&&... rest );

}}}}

**New section: For loop [parallel.alg.forloop]:**

template<typename Policy, typename I, typename... Rest>

void for\_loop (Policy&& policy, common\_type\_t<I> first, I last,

Rest&&... rest );

template<typename Policy, typename I, typename S, typename... Rest>

void for\_loop\_strided(Policy&& policy, common\_type\_t<I> first, I last,

S stride, Rest&&... rest );

template<typename Policy, typename I, typename Size, typename... Rest>

void for\_loop\_n(Policy&& policy, I first, Size n, Rest&&... rest );

template<typename Policy, typename I, typename Size, typename S,

typename... Rest>

void for\_loop\_n\_strided(Policy&& policy, I first, Size n, S stride,

Rest&&... rest );

*Requires:* The parameter pack rest shall have at least one element; the last element shall be an invocable object, *f*, callable with an argument list composed as described below. Each of the remaining elements of the parameter pack shall be the result of invoking an instance of one of the reduction or induction function templates.

The type of *f* shall meet the requirements of MoveConstructible if Policy is sequential\_execution\_policy, otherwise it shall meet the requirements of CopyConstructible.

If specified, n shall be non-negative.

If specified, stride shall be non-zero.

I shall be an integral type or meet the requirements of an input iterator if stride is positive or not specified, otherwise S shall be an integral type and:

* I shall be an integral type or
* I shall meet the requirements of an input iterator if stride is positive or
* I shall meet the requirements of a bidirectional iterator if stride is negative.

*Effects*: Applies *f* to a sequence of argument lists. Let *s* be stride if specified or 1 otherwise. Let *e* be last if specified and first+*s*\*n otherwise. For a positive *s*, the value of the first argument is successive values in the range [first,*e*) starting with first and advancing by *s*. For negative *s*, the value of the first argument is successive values in the range (*e*,first], starting with first and advancing (backwards) by *s*. [*Note:* whether or not the order of elements in the range is significant depends on the execution policy – *end note*]

Each argument list contains an additional iteration-local value corresponding, by positional order, to each parameter in rest except the last.

For a reduction object, its iteration-localvalue is a reference to a temporary of its reduction type. Each temporary is copy-constructed from the reduction’s *identity* value. The reduction object’s *var* is updated with the result of applying *reduction-op* to *var* and any temporaries that were generated.When these updates occur is implementation specific.

For induction object with no stride, its iteration-localvalue for the *i*th iteration is *original* + *i* and its *final value* is *var* + *n*, where *n* s the number of times that *f* was applied. For an induction object with a *stride*, its iteration-localvalue for the *i*th iteration is *original* + *i*\**stride* and its *final value* is *var* + *n*\**stride*. The *var* is set to the *final value.* All applications of *f* are sequenced before *var* is set.

If Policy is sequential\_execution\_policy, the application shall start with the first argument list in the sequence and proceed to the last one.

*Complexity:* If stride is not specified, applies f exactly last-first times; otherwise, applies f exactly ⎣(last-first-1)/stride⎦+1 if stride is positive, and ⎣(first-last-1)/stride⎦+1 times if stride is negative.

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

*[Note:* When i and j are iterators, for\_loop(policy,i,j,f) differs from for\_each(policy,i,j,f) in that the latter dereferences elements in [first,last) *--note]*