|  |  |
| --- | --- |
| Document number: | P0075R0 |
| 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> |

**Template Library for Index-Based Loops**

# Introduction

Indexed-based loops are well established in programming languages. Though C++ has language-level support for sequential forms of these loops, it has none for parallel or parallel-vector forms. This paper proposes support for all aforementioned forms 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.

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

* **for\_loop** and **for\_loop\_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 a random-access 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.

## Strided Loops

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, we the function template for strided loops has a different name. 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

The for\_loop template also 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-specified *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)

The for-loop template also 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-specified type that specifies two things:

* a reference to an induction lvalue (e.g. x)
* a optional stride for that lvalue. 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, x, y, z 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

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.

# 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-specified 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 Op>

*implementation-specified* reduction( T& var, T&& identity, Op&& op );

// Shorthand for plus reduction

template<typename T>

*implementation-specified* reduction\_plus( T& var );

// Shorthand for multiplies reduction

template<typename T>

*implementation-specified* reduction\_multiplies( T& var );

// Shorthand for bit\_and reduction

template<typename J>

*implementation-specified* reduction\_bit\_and( J& var );

// Shorthand for bit\_or reduction

template<typename J>

*implementation-specified* reduction\_bit\_or( J& var );

// Shorthand for bit\_xor reduction

template<typename J>

*implementation-specified* reduction\_bit\_xor( J& var );

// Shorthand for min reduction

template<typename T>

*implementation-specified* reduction\_min( T& var );

// Shorthand for max reduction

template<typename T>

*implementation-specified* reduction\_max( T& var );

}}}}

Each function returns a “reduction object” that specifies a *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 as follows:

|  |  |  |
| --- | --- | --- |
| *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 | std::numeric\_limits<T>::max() | std::min(x,y) |
| reduction\_max | std::numeric\_limits<T>::lowest() | 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>

*implementation-specified* induction( T& var );

template<typename T, typename S>

*implementation-specified* induction( T& var, S stride );

}}}}

Each function returns an “induction object” that specifies a *var*, and optionally a *stride.*  See description of for\_loop for how these are used.

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

namespace std {

namespace experimental {

namespace parallel {

inline namespace v2 {

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

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

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

void for\_loop\_strided( Policy&& policy, I first, I last, S stride, Rest&&... rest );

}}}}

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

stride shall be non-zero.

The type of *f* shall meet the requirements of MoveConstructible if Policy is sequential\_execution\_policy, otherwise it shall meet the requirements of CopyConstructible. For for\_loop, I shall be an integral type or meet the requirements of an input iterator. For for\_loop\_strided, 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. For for\_loop, the value of the first argument is successive values in the range [first,last). For for\_loop\_strided with a positive stride, the value of the first argument is successive values in the range [first,last) starting with first and advancing by stride. For for\_loop\_strided with negative stride, the value of the first argument is successive values in the range (last,first], starting with first and advancing (backwards) by stride.

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:* for\_loop applies f exactly last-first times. for\_loop\_strided 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]*