Document number: P0075r1

Date: 2016-02-12

Audience: Concurrency study group (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>

Template Library for Parallel For Loops

Contents

1 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 in order to handle a number of common patterns:

- Traversing multiple sequences in the same loop, 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 often needed 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.

Target: Next revision of the Parallelism TS

2 Changes since r0

- Added `for_loop_n` and `for_loop_n_strided`.
- Added serial versions of all new algorithms, since none of them currently have serial equivalents.
- Added more rationale for reduction and inductions to store final values as side effects.
- Added precision and improved the formatting of the formal wording.

3 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, these algorithms resemble `for_each` from the Parallelism TS, but leave 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_multiplies`**, etc. create 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.

3.1 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 distinguished by name.

3.2 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. Again, 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 example below sets `c[10]`, `c[13]`, `c[16]`, and `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;
});
```

3.3 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 or 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 encapsulates 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 view containing a 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`, deliberately hiding the identically-named and closely-related variable used to create the reduction.) 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>())
```

Note that floating-point operations may be re-ordered and re-associated, thus exposing round-off errors that differ from the serial execution and, for certain execution policies, may vary from run to run. This difficulty with floating-point arithmetic is well known and consistent with other parallelism systems such as OpenMP.

3.4 Inductions (Linear Variables)

A linear induction value is a value that varies 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 encapsulates 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 copy of the loop index, and `x_, y_, z_` are initialized to `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. Note that induction values are passed to the lambda-function by value, whereas for reduction variables they are passed by modifiable reference. 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;
}
```

4 Alternative Design Choices

Below, we describe some alternatives that were considered and why we are not proposing them.

4.1 Leaving out inductions

Inductions could be omitted from this proposal, relying on users to write the equivalent math. However, doing so complicates parallelizing codes. We note that OpenMP has linear clauses for similar reason.

4.2 Leaving out reductions

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.

4.3 Returning “Live out” values of inductions and reductions

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 compromise thread safety 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 will be aware, by the very nature of the operation, that the value is modified. In general, induction and reduction variables will be local variables in the same scope as the `for_loop` function call, and there is no reason to believe that they will be any more likely than other variables to 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.

5 Future enhancements

5.1 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. By leaving the return value of the `reduction` function unspecified, we leave room for defining a user-extensible type/concept system in a future revision.

5.2 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.

6 Formal Wording

The proposed edits are with respect to the current DTS for Parallelism TS, N4507.

6.1 Feature-testing macro

Add to section [parallel.general.features], Table 1, the following row:

Doc no.	**Title**	**Primary Section**	**Macro Name Suffix**	**Value**	**Header**
P0075r1	Template Library for Parallel For Loops	4.3	`parallel_for_loop`	201602	`<experimental/memory>`

6.2 Additions to `<experimental/algorithms>` synopsis

Add the following text to to Header **`<experimental/algorithm>`** synopsis [parallel.alg.ops.synopsis]:

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

// Support for reductions (see [parallel.alg.reductions])
template <typename T, typename BinaryOp>
  see-below reduction(T& var, const T& identity, BinaryOp&& combiner);
template <typename T>
  see-below reduction_plus(T& var);
template <typename T>
  see-below reduction_multiplies(T& var);
template <typename J>
  see-below reduction_bit_and(J& var);
template <typename J>
  see-below reduction_bit_or(J& var);
template <typename J>
  see-below reduction_bit_xor(J& var);
template <typename T>
  see-below reduction_min(T& var);
template <typename T>
  see-below reduction_max(T& var);

// Support for inductions (see [parallel.alg.inductions])
template <typename T>
  see-below induction(T&& var);
template <typename T, typename S>
  see-below induction(T&& var, S stride);

// for_loop [parallel.alg.forloop]
template <typename I, typename... Rest>
  void for_loop(decay_t<I> first, I last, Rest&&... rest);
template <typename ExecutionPolicy, typename I, typename... Rest>
  void for_loop(ExecutionPolicy&& exec,
                decay_t<I> first, I last, Rest&&... rest);
template <typename I, typename S, typename... Rest>
  void for_loop_strided(decay_t<I> first, I last,
                        S stride, Rest&&... rest);
template <typename ExecutionPolicy,
         typename I, typename S, typename... Rest>
  void for_loop_strided(ExecutionPolicy&& exec,
                        decay_t<I> first, I last,
                        S stride, Rest&&... rest);
template <typename I, typename Size, typename... Rest>
  void for_loop_n(I first, Size n, Rest&&... rest);
template <typename ExecutionPolicy,
          typename I, typename Size, typename... Rest>
  void for_loop_n(ExecutionPolicy&& exec,
                  I first, Size n, Rest&&... rest);
template <typename I, typename Size, typename S, typename... Rest>
  void for_loop_n_strided(I first, Size n, S stride, Rest&&... rest);
template <typename ExecutionPolicy,
          typename I, typename Size, typename S, typename... Rest>
  void for_loop_n_strided(ExecutionPolicy&& exec,
                          I first, Size n, S stride, Rest&&... rest);

}}}}
```

6.3 New text for reductions

Add the following text to to Non-Numeric Parallel Algorithms [parallel.alg.ops] before [parallel.alg.foreach]:

Reductions [parallel.alg.reductions]

Each of the function templates in this section returns a *reduction object* of unspecified type having a *value type* and encapsulating an *identity* value for the reduction, a *combiner* function object, and a *live-out object* from which the initial value is obtained and into which the final value is stored.

A parallel algorithm uses reduction objects by allocating an unspecified number of instances, called *views*, of the reduction's value type. [*Note:* an implementation might, for example, allocate a view for each thread in its private thread pool – *end note*] Each view is initialized with the reduction object's identity value, except that the live-out object (which was initialized by the caller) comprises one of the views. The algorithm passes a reference to a view to each application of an element-access function, ensuring that no two concurrently-executing invocations share the same view. A view can be shared between two applications that do not execute concurrently, but initialization is performed only once per view.

Modifications to the view by the application of element access functions accumulate as partial results. At some point before the algorithm returns, the partial results are combined, two at a time, using the reduction object's combiner operation until a single value remains, which is then assigned back to the live-out object. [*Note:* in order to produce useful results, modifications to the view should be limited to commutative operations closely related to the combiner operation. For example if the combiner is `plus<T>`, incrementing the view would be consistent with the combiner but doubling it or assigning to it would not. – *end note*]

```
template <typename T, typename BinaryOp>
  see-below reduction(T& var, const T& identity, BinaryOp&& combiner);
```

> *Requires:* T shall meet the requirements of `CopyConstructible` and `MoveAssignable`. The expression `var = combiner(var, var)` shall be well formed.
>
> *Returns:* Returns a reduction object of unspecified type having a value type of T. When the return value is used by an algorithm, the reference to `var` is used as the live-out object, new views are initialized to a copy of `identity`, and views are combined by invoking the copy of `combiner`, passing it the two views to be combined.

```
template <typename T>
  see-below reduction_plus(T& var);
template <typename T>
  see-below reduction_multiplies(T& var);
template <typename J>
  see-below reduction_bit_and(J& var);
template <typename J>
  see-below reduction_bit_or(J& var);
template <typename J>
  see-below reduction_bit_xor(J& var);
template <typename T>
  see-below reduction_min(T& var);
template <typename T>
  see-below reduction_max(T& var);
```

> *Requires:* shall meet the requirements of `CopyConstructible` and `MoveAssignable`.
>
> *Returns:* Returns a reduction object of unspecified type having a value type of T. When the return value is used by an algorithm, the reference to `var` is used as

the live-out object, new views are initialized to a copy of the identity shown in Table 1, and views are combined by applying the combiner operation from Table 1.

Table 1 -- Reduction identities and reduction-ops

function	*identity*	*combiner operation*
`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, 0.0f, std::plus<float>()),
    [&](int i, float& t) {
        y[i] += a*x[i];
        t += y[i]*y[i];
    }
});
```

– *end example*]

6.4 New text for inductions

Inductions [parallel.alg.inductions]

Each of the function templates in this section return an *induction object* of unspecified type having a *value type* and encapsulating an initial value *i* of that type and, optionally, a *stride*.

For each element in the input range, a looping algorithm over input sequence *S* computes an *induction value* from an induction variable and ordinal position *p* within *S* by the formula *i* + *p* * *stride* if a stride was specified or *i* + *p* otherwise. This induction value is passed to the element access function.

If the `var` argument to `induction` is a non-const lvalue, then that lvalue becomes the *live-out* object for the returned induction object. For each induction object that has a live-out object, the looping algorithm assigns the value of *i* + *n* * *stride* to the live-out object upon return, where *n* is the number of elements in the input range.

```
template <typename T>
  see-below induction(T&& var);
template <typename T, typename S>
  see-below induction(T&& var, S stride);
```

Returns: Each function returns an induction object with value type T, initial value `var`, and (if specified) stride `stride`. If T is an lvalue of non-`const` type, `var` is used as the live-out object for the induction object; otherwise there is no live-out object.

6.5 New text for `parallel::for_loop`

For loop [parallel.alg.forloop]

```
template <typename I, typename... Rest>
  void for_loop(decay_t<I> first, I last, Rest&&... rest);
template <typename ExecutionPolicy, typename I, typename... Rest>
  void for_loop(ExecutionPolicy&& exec,
                decay_t<I> first, I last, Rest&&... rest);
template <typename I, typename S, typename... Rest>
  void for_loop_strided(decay_t<I> first, I last,
                        S stride, Rest&&... rest);
template <typename ExecutionPolicy,
         typename I, typename S, typename... Rest>
  void for_loop_strided(ExecutionPolicy&& exec,
                        decay_t<I> first, I last,
                        S stride, Rest&&... rest);
template <typename I, typename Size, typename... Rest>
  void for_loop_n(I first, Size n, Rest&&... rest);
template <typename ExecutionPolicy,
          typename I, typename Size, typename... Rest>
  void for_loop_n(ExecutionPolicy&& exec,
                  I first, Size n, Rest&&... rest);
template <typename I, typename Size, typename S, typename... Rest>
  void for_loop_n_strided(I first, Size n, S stride, Rest&&... rest);
template <typename ExecutionPolicy,
          typename I, typename Size, typename S, typename... Rest>
  void for_loop_n_strided(ExecutionPolicy&& exec,
                          I first, Size n, S stride, Rest&&... rest);
```

Requires: `I` shall be an integral type or meet the requirements of an input iterator type. `Size` shall be an integral type and `n` shall be non-negative. `S` shall have integral type and `stride` shall have non-zero value. `stride` shall be negative only if `I` has integral type or meets the requirements of a bidirectional iterator. The `rest` parameter pack shall have at least one element, comprising objects returned by invocations of reduction ([parallel.alg.reduction]) and/or induction ([parallel.alg.induction]) function templates followed by exactly one element invocable element-access function, *f*. If `exec` is specified, *f* shall meet the requirements of `CopyConstructible`; otherwise, *f* shall meet the requirements of `MoveConstructible`.

Effects: Applies *f* to each element in the *input sequence*, as described below, with additional arguments corresponding to the reductions and inductions in the `rest` parameter pack. The length of the input sequence is:

— `n` if specified, otherwise

— `last - first` if neither `n` nor `stride` is specified, otherwise

— `(last-first-1)/stride+1` if `stride` is positive, and `(first-last-1)/stride+1` times if stride is negative.

The first element in the input sequence is specified by `first`. Each subsequent element is generated by adding `stride` to the previous element, if `stride` is specified, otherwise by incrementing the previous element. [*Note:* As described in the C++ standard, section [algorithms.general], arithmetic on non-random-access iterators is performed using `advance` and `distance`. – *end note*] [*Note:* The order of the elements of the input sequence is important for determining ordinal position of an application of *f*, even though the applications themselves may be unordered. – *end note*]

Along with an element from the input sequence, for each member of the `rest` parameter pack excluding *f*, an additional argument is passed to each application of *f* as follows:

— If the pack member is an object returned by a call to a reduction function listed in section [parallel.alg.reductions], then the additional argument is a reference to a view of that reduction object.

— If the pack member is an object returned by a call to `induction`, then the additional argument is the induction value for that induction object corresponding to the position of the application of *f* in the input sequence.

Complexity: Applies *f* exactly once for each element of the input sequence.

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