Doc. no.: N2104=06-0174
Date: 2006-09-29
Project: Programming Language C++
Reply to: Arch D. Robison <arch.robison@intel.com>
Intel Corporation
1906 Fox Drive
Champaign IL USA 61820

# A Proposal to Add Parallel Iteration to the Standard Library

## 1 Table of Contents

## 2 Motivation

Multi-core processors are becoming common, yet writing even a simple parallel for-loop is tedious with existing threading packages. Writing an efficient one is much harder. This document proposes templates for common parallel iteration patterns that enable programmers to get speedup from multiple cores/processors without having to be experts in synchronization, load balancing, and cache optimization.

The goal of this proposal is parallelism for performance. It is not a general replacement for threading libraries.

## 3 Impact on the Standard

The proposal introduces the notion of multiple threads of control. It assumes that a memory consistency model will be added to the standard via other proposals. This proposal has been implemented in standard C++ using POSIX or Windows threads, and some common atomic operations (compare-and-swap, fetch-and-decrement).

This proposal can be implemented as a pure extension, because conforming implementations may choose to do no threading at all, as explained in Section 4.1.2.

Partial order diagrams are used to state concurrency constraints. Using pictures for normative information would be new in the standard, but seems to be most concise.

## 4 Design Decisions

This proposal is based on experience with a larger library known as Intel® Threading Building Blocks [TBB06].

### 4.1 General Principles

#### 4.1.1 Be generic

Generic programming enables high-quality implementations of broadly applicable algorithms. For example, the proposed template `parallel_for` applies to any recursively divisible space, not only integers or random access iterators.

#### 4.1.2 Parallelism is allowed, not required

This proposal allows parallelism, but does not require it. For example, a valid implementation of the proposed `parallel_for` is the following recursive routine:

```
template<class RecursiveRange, class Body>
void parallel_for (const RecursiveRange& range, const Body& body) {
  if (!range.empty())
    if (range.is_divisible()) {
      body(range);
    } else {
      RecursiveRange rest(range,split());
      parallel_for(range,body);
      parallel_for(rest,body);
    }
}
```

Some parallel programming paradigms require concurrency; e.g., a producer-consumer relationship with a bounded buffer requires concurrency to avoid deadlock. Mandatory concurrency is often inefficient when there are more runnable software threads than hardware threads to service them. Parallel performance is usually best when the system is allowed to throttle how much *available* parallelism it turns into *real* parallelism.

One possible parallel implementation of `parallel_for`, based on a `cobegin/coend` extension, is shown below.

```
template<class RecursiveRange, class Body>
void parallel_for (const RecursiveRange& range, const Body& body) {
  if (!range.empty())
    if (range.is_divisible()) {
      body(range);
    } else {
      const Body b2(body);
      RecursiveRange r2(range,split());
      cobegin
          parallel_for(range,body);
          parallel_for(r2,b2);
      coend
    }
}
```

A good way to throttle this parallelism is the work-stealing algorithm from Cilk [BJKLRZ95], which yields the benefits detailed in Section 4.2.2.

### 4.1.3 Full support for nested parallelism

Software components are built from other components. Each level of component may be able to express parallelism. The proposal's interfaces permit efficient implementation of nested parallelism, such as the approach outlined in Section 4.2.2.

### 4.1.4 Hide the number of physical threads

The number of available physical threads is often not really known, because some of them may be servicing other processes on the system. Furthermore, even on a single-program system, support of nested parallelism implies that not all threads are available to service a routine if there are other routines running in parallel. Thus this proposal deliberately omits providing any portable way to find out the number of available physical threads. The implementation should be free to choose dynamically the appropriate number of underlying physical threads to use at any given moment.

### 4.1.5 Be efficient without compiler heroics

Doing parallel programming for performance inherently demands efficiency, for if the parallel routine is slower than the corresponding sequential routine, it was not worth parallelizing. A parallel programming interface should not rely upon heroic compiler support for practical implementation. Examples of heroics would be whole program analysis or solving Diophantine equations. Besides putting a burden on implementers, depending on such heroics puts a burden on programmers, because the boundary separating where heroics succeed and where they fail is often murky and implementation dependent. This proposal can be implemented efficiently as a library using existing C++ compilers.

## 4.2 Implications

### 4.2.1 Program with task patterns, not threads.

Traditional pre-emptive threads flunk some of the principles stated in Section 4.1, namely they require concurrency, and do poorly for nested parallelism because nesting leads to an exponential explosion of software threads.

Traditional threads are too heavy for small parallel tasks. An alternative would be lighter weight thread implementations that support nesting [Narlikar99], but that would require significant OS support.

Therefore, in this proposal the programmer specifies logical tasks as classes. For example, the *body* argument to a `parallel_for` specifies the task to be performed for each subrange in a range, somewhat analogous to the function argument to `std::for_each`. Using tasks instead of threads permits practical finer grained parallelism than traditional threads, because tasks can be executed by mechanisms that are typically much faster than starting up and shutting down a thread.

### 4.2.2 Divide and conquer with work stealing

Published work on Cilk [Cilk][BJKLRZ95] indicates that a divide-and-conquer strategy, combined with work stealing, yields all of the following:

- throttling actual parallelism to available parallelism
- good load balancing
- good cache usage
- efficient nested parallelism

The interfaces of the proposal permit an efficient implementation on top of Cilk or similar run-time. It is straightforward to build a similar underlying run-time on top of standard C++ and common threading interfaces without the Cilk syntactic extensions.

### 4.2.3 Recursive ranges, not iterators

An early prototype of `parallel_for` took three arguments similar to `std::for_each`. Two of the arguments specified: the "body" argument and two random-access iterators: begin and end. Experience was that the traditional [begin,end) specification was awkward for parallel programming, for several reasons:

- Only random access iterators sufficed. Sequential access iterators (input, output, forward, bidirectional) precluded scalable parallelism.
- We often wanted to parallelize over two dimensions. There were ways to contort iterators to do multi-dimensional traversal with divide-and-conquer, but they were unnatural.

Ranges have been proposed as a way to encapsulate iterators [Ottosen06]. Recursive ranges are fundamentally more general, because they permit recursive descriptions of iteration spaces that do not necessarily have iterators. Hence this proposal uses "recursive range" to distinguish from the other range proposal.

Section 4.3.4 describes how `blocked_range` provides a way to construct a recursive range from a [*begin,end)* pair of random-access iterators.

STAPL [STAPL], another generic parallel C++ library, has a similar notion of recursive parallel ranges, albeit more complex because it tackles issues in distributed memory programming.

## 4.3 Specific Decisions

### 4.3.1 No automagic startup/shutdown

Any user-created thread must initialize the library using `task_scheduler_init`. An early prototype had automatic startup/shutdown, but our OpenMP group, based on their experience, strongly warned against this because it is extremely difficult to implement the shutdown part correctly on some operating systems. In particular, always knowing when a thread shut down was problematic on some operating systems.

### 4.3.2 Splitting syntax

The notion of recursive splitting is central to this proposal. The requirements for recursive ranges and the requirements for a parallel_reduce's body both require splitting. The proposal uses the splitting constructor notation R s(r, `split()`) to split r of type R into subranges r and s. The dummy type `split` serves to distinguish it from normal copy construction. The splitting constructor destructively updates r, and so is allowed to recycle resources from r.

Several alternatives were considered:

- R s = r.split(). This form had the disadvantage of requiring extra object copying. In principle, the compiler can eliminate the extra copies in most copies, but that depends upon the compiler having the optimization and the user knowing how to cater to it.
- r.split(s). This form would require prior initialization of s, and thus possibly incur extra overhead compared to constructing s on the spot.
- r.split(s1,s2) where s1 and s2 would be the new subranges, and r is a const R. This form would preclude recycling resources in the original range for one of the subranges. This is a serious overhead in some situations. For example, if the range is a collection, the collection would have to be copied.
- pair<R> s1s2 = r.split() where r is const R. This form suffers both from object copying and not being able to recycle.

Allowing splitting into more than two pieces was considered, but seems to add a lot of complexity with little gain in power, because a K-way split can be implemented as K-1 two-way splits.

### 4.3.3 affinity_record

Repeated application of a parallel loop can be inefficient if iteration *i* is run on a different processor each time, which incurs a cost of moving data associated with *i* between processors (and thus often between caches). The performance increase in our prototype from using an `affinity_record` with a simple stencil computation is ~25%. The prototyped uses the work stealing modification described by Acar, Blelloch, and Blumofe [ABB].

### 4.3.4 blocked_range

A `blocked_range` abstracts the loop "for(Value i=begin; i<end; ++i)", in a way amenable to parallel execution. Values are now required to implement LessThanComparable, which gives the loop well defined semantics even when end<begin.

The original prototype of `blocked_range` abstracted the loop "for(Value i=begin; i!=end; ++i) in a way amenable to parallel execution. In that prototype, Values had to be EqualityComparable, not LessThanComparable. Unfortunately, that meant that parallel algorithms had no way to detect when the user passed in a backwards range, e.g., `[3,-5)`, and the results of such executions were wild. Early experience with users indicated that detecting such errors was valuable. Hence the change to the current requirements.

A random-access iterator with a `size_type` of `size_t` can serve as a Value for a `blocked_range`. It would be nice if the sizes in `blocked_range` were generalized to the "size type" of the value type, instead of coerced to `size_t`. Because Values are not necessarily iterators, deducing the "size type" is non-trivial. If the decltype [JSR04] becomes part of the language, then perhaps deducing the "size type" would be easier to deduce.

Using a signed integral type for the size of a blocked_range was rejected as inconsistent with containers. The interface for `blocked_range` attempts to give it the look and feel of a const container of values.

The grainsize controls amortization of parallel scheduling overhead. Alas, the optimal grainsize will depend upon implementations. Automatic determination of grainsize is still a research problem. A default grainsize such as (*end-begin*)/(P*4) for a P-processor machine *usually* works well, but has the bad behavior that amortization gets worse with increasing numbers processors, causing parallel slowdown instead of parallel speedup. Having a grainsize independent of the number of processors tends to keep, in common cases, the parallel scheduling overhead in a constant proportion to real work. However, feedback from users is that they want automatic grainsize determination, even if it is suboptimal, so the grainsize parameter is optional.

The grainsize is in the `blocked_range`, not the body argument to the `parallel_for` or `parallel_reduce`. Here's the implications for the alternatives:

- Putting it in the `blocked_range` allows the decision "is divisible" to be made without referring to the body. It allows commonly reused splitting logic to be encapsulated in `blocked_range`.
- Putting the grainsize in the body would have required adding some kind of requirement to a recursive range (e.g. "size") that could be queried by the body argument to a `parallel_for` or `parallel_reduce`. This would require adding another required signature to `parallel_for`, which would preclude using function objects as the body for `parallel_for`. The preclusion would be particularly annoying if lambda expressions are added to C++ [WJGSL06][Samko06].

### 4.3.5 blocked_range2d

Template class `blocked_range2d` is included because beneficial uses for it showed up commonly. Parallelizing over two dimensions instead of one often yields more parallelism and better cache behavior than parallelizing over only one dimension.

A generic "cartesian product of any two ranges" was not attempted because it was not clear how the splitting constructor should behave when both dimensions are divisible.

Extensions to three or more dimensions, or making the number of dimensions a parameter, were considered but rejected as adding too much complexity with little practically motivating cases.

A constructor for `blocked_range2d` that takes two `blocked_range` arguments was considered as an alternative to the 6-argument constructor, but so far practice has shown that such a constructor just adds extra clutter.

### 4.3.6 parallel_for

The Body type must be copy constructible so that implementations have freedom to make thread-private copies of a body. Because implementations are not required to merge the copies afterwards, Body`::operator()` is required to be const qualified, as a syntactic guard against trying to accumulate side effects in *body* which would be lost by the thread-private copies.

The specification of `parallel_for` is general enough to enable coding a crude parallel quicksort without explicit recursion. The trick is to define a RecursiveRange that represents the key sequence, using the is_divisible and splitting constructor to do the partitioning logic, and making Body`::operator()` perform the leaf-case insertion sorts.

### 4.3.7 parallel_reduce

The proposal attempts to make `parallel_reduce` similar to `parallel_for`. The principle difference is that thread-private copies of *body* must be merged at the end.

The requirements are in the form of requiring that alternative execution sequences have the same net side effects. Thus the specification circumvents trying to define what a pure function is.

### 4.3.8 parallel_while

The proposed `parallel_while` usually does not provide scalable parallelism if method `add` is not used, because the input stream typically acts as a bottleneck. However, this bottleneck is broken if the stream is used to "prime a pump" and further items come from invocations of method `add` caused by previous items.

Even in the non-scalable case, `parallel_while` covers a commonly requested idiom of walking a sequential structure (e.g. a linked list) and dispatching concurrent work for each item in the structure.

The proposed `parallel_while` requires that the programmer supply an object with method `pop_if_present`(). The implementation is free to concurrently invoke that method on the same object. An alternative design would be to define the input stream with a range [*begin*,*end*) defined in terms of input iterators. This would have a performance drawback of requiring that the implementation of `parallel_while` fetch an item by executing something like:

```
acquire lock
bool okay=i!=end;
if(okay) item=*i++;
release lock
```

Experience has been that programmers can often supply an atomic equivalent that is significantly more efficient; e.g., via an atomic operations library [Boehm06].

### 4.3.9 pipeline

Pipelined processing is common in multi-media applications. The classical thread-per-stage implementation suffers two problems:

1. The speedup is limited to the number of stages.
2. When a thread finishes a stage, it must pass its data to another thread.

This proposal follows the recommendations of [MSS] to eliminate these problems. The programmer specifies whether a stage is serial or parallel. A serial stages processes items one at a time in order. A parallel stage may process items out of order or concurrently.

A pipeline contains one or more filters, denoted here as $f_i$ , where *i* denotes the position of the filter in the pipeline. The pipeline starts with filter $f_0$, followed by $f_1$, $f_2$, etc. The following steps describe how to use class pipeline.

1. Derive classes $f_i$ from `filter`. The constructor for $f_i$ specifies whether it is serial or not via the bool parameter to the constructor for base class `filter`.
2. Override virtual method `filter::operator()` to perform the filter's action on the item, and return a pointer to the item to be processed by the next filter. The first filter $f_0$ generates the stream. It should return NULL if there are no more items in the stream. The return value for the last filter is ignored.
3. Create an instance of class `pipeline`.
4. Create instances of the filters $f_i$ and add them to the pipeline, in order from first to last.
5. Call method `pipeline::run`. The parameter `max_number_of_live_tokens` puts an upper bound on the number of stages that will be run concurrently. Higher values may increase concurrency at the expense of more memory consumption from having more items in flight.

Given sufficient processors and tokens, the throughput of the pipeline is limited to the throughput of the slowest serial filter.

### 4.3.10 filter

The proposed form of a filter takes a (void*) as an argument, and returns a (void*). It would be nice if the arguments were more precisely typed, but it seems difficult to combine static typing of filters with the ability to dynamically build a pipeline. To have stronger typing would seem to require restricting pipelines to a fixed number and sequence of stages at compilation time.

## 4.4 Comparison with OpenMP Scheduling

OpenMP [OpenMP02] is the most successful parallel extension to date. It is a language extension consisting of pragmas, routines, and environment variables. OpenMP lets the programmer choose between three scheduling approaches (static, guided, dynamic) for scheduling loop iterations:

**Static** scheduling partitions the index space into chunks of equal size. If different iterations contain differing amounts of work, or the system processors have other tasks too, such scheduling may incur a load imbalance penalty.

**Dynamic** scheduling deals out work as worker threads demand it. This scheme provides optimal load balancing, but often incurs high communication cost because the dealing mechanism becomes a source of contention.

**Guided** scheduling deals out work in chunks of exponentially decreasing size. This scheme provides a good tradeoff of communication versus load balancing, because it hands out big chunks early, and fills in the load-balancing gaps with small work later. It's like most people pack a suitcase: big items first and socks last. The dealing mechanism can become a contention problem, though less than for dynamic scheduling.

In contrast to the three OpenMP scheduling schemes, this proposal has a single divide-and-conquer approach. Implemented with work stealing, it compares favorably to dynamic or guided scheduling, but without a centralized dealer. Static scheduling is sometimes faster on systems undisturbed by other processes (or concurrent sibling code). However, divide-and-conquer comes close enough for most purposes, and fits well with nested parallelism. Furthermore, generic static scheduling would impose a more complex "split yourself into *p* equal pieces" requirement, instead of the proposals "split yourself into two approximately equal pieces" requirement.

## 5 Proposed Text for the Standard

This clause describes components for parallel programming.

### 5.1 Requirements

#### 5.1.1 Recursive Range

A recursive range is a set of values that may be recursively partitioned.

A class `R` satisfies the requirements of a recursive range. In Table 1, `R` is a type to be supplied by a C++ program instantiating a template, r1 and r2 are values of type R, and `s` is a value of type const R.

**Table 1 – RecursiveRange requirements (in addition to CopyConstructible)**

| **expression** | **return type** | **requirements** |
|---|---|---|
| `s.empty()` | convertible to `bool` | return true iff range is empty |
| `s.is_divisible()` | convertible to `bool` | return true iff range should be partitioned into two subranges. |
| `R r2(r1,split())` | | pre: `r1.is_divisible()`<br>split `r1` into two subranges r1 and r2. |

**Header <parallel> synopsis**

```
namespace std {
  class split {};
  class task_scheduler_init;
  class task_affinity_record;
  template<typename Value> class blocked_range;
  template<typename Value> class blocked_range2d;
  template<typename RecursiveRange, typename Body>
    void parallel_for(const RecursiveRange& range, const Body& body );
  template<typename RecursiveRange, typename Body>
    void parallel_reduce(const RecursiveRange& range,
                         const Body& body );
  template<typename Body> class parallel_while;
  class pipeline;
  class filter;
}
```

## 5.2 Class task_scheduler_init

A `task_scheduler_init` represents a thread's request for task scheduling services. A `task_scheduler_init` is either active or inactive. A thread must have at least one active `task_scheduler_init` while it executes a `parallel_for` or `parallel_reduce`.

```
namespace std {
  class task_scheduler_init {
  public:
    static const int automatic = implementation-defined;
    static const int deferred = implementation-defined;
    task_scheduler_init( int number_of_threads=automatic );
    ~task_scheduler_init();
    void initialize( int number_of_threads=automatic );
    void terminate();
  };
}
```

### 5.2.1 Startup

```
task_scheduler_init( int number_of_threads=automatic );
```

**Requires:** The value *`number_of_threads`* shall be one of the values in Table 2.

**Effects:** If *`number_of_threads`*`==task_scheduler_init::deferred`, the `task_scheduler_init` is initialized as inactive. Otherwise, the `task_scheduler_init` is initialized as active. If there were no other extant active `task_scheduler_init` objects, then the argument is a hint at the number of internal worker threads to create, as described in Table 2.

**Table 2: Values for *number_of_threads***

| ***`number_of_threads`*** | **effects** |
|---|---|
| `task_scheduler_init::automatic` | Let library determine *`number_of_threads`* based on hardware configuration. |
| `task_scheduler_init::deferred` | Defer activation actions. |
| positive integer | If no worker threads exist yet, create *`number_of_threads`*–1 worker threads. If worker threads exist, do not change the number of worker threads. |

[*Note:*The positive integer values are intended to assist scaling studies during program development.]

```
void initialize( int number_of_threads=automatic )
```

**Requires:** The `task_scheduler_init` shall be inactive.

**Effects:** Make `task_scheduler_init` active. The *`number_of_threads`* has the same meaning as in the constructor.

### 5.2.2 Shutdown

```
~task_scheduler_init()
```

**Effects:** If the `task_scheduler_init` is active, make `task_scheduler_init` inactive before destroying it.

```
void terminate()
```

**Requires:** The `task_scheduler_init` shall be active.

**Effects:** Make `task_scheduler_init` inactive.

The description of the destructor specifies what deactivation entails.

## 5.3 Class affinity_record

An `affinity_record` is an optional argument to `parallel_for` and `parallel_reduce` that hints that performance might be improved if iterations are run on (or near) the same processors as for the previous execution with that `affinity_record`.

```
namespace std {
  class affinity_record {
  public:
    affinity_record();
    ~affinity_record();
  };
}
```

[*Exampl*e:

```
affinity_record a, b;
for( int i=0; i<2; ++i ) {
    parallel_for(r,f,a); // #1
    parallel_for(r,g,b); // #2
    parallel_for(r,h,a); // #3
}
```

Lines #1 and Line #3 hint that they should be run with a similar iteration↔processor mapping each time. Likewise for Line #2. The separate hint says it is unrelated to #1 and #3.
--*end example*]

## 5.4 Template class blocked_range

```
namespace std {
  template<typename Value>
  class blocked_range {
  public:
    // types
    typedef size_t size_type;
    typedef Value const_iterator;

    // constructors
    blocked_range(Value begin, Value end, size_type grainsize);
    blocked_range(Value begin, Value end);
    blocked_range(blocked_range& r, split);
```

```
        // capacity
        size_type size() const;
        bool empty() const;

        // access
        size_type grainsize() const;
        bool is_divisible() const;

        // iterators
        const_iterator begin() const;
        const_iterator end() const;
      };
    }
```

A `blocked_range` represents a half-open range [*i*,*j*) that can be recursively split when the number of values is greater than its grainsize. It satisfies all of the requirements for a recursive range.

In Table 3, Value is a type supplied by a C++ program instantiating a `blocked_range`, `i` and `j` are values of type const Value, and `k` is of type `size_t`.

**Table 3 – requirements on Value for a blocked_range (in addition to CopyConstructible)**

| **expression** | **return type** | **requirements** |
|---|---|---|
| `i<j` | convertible to `bool` | return true if Value `i` precedes value j |
| `i-j` | convertible to `size_t` | number of values in range [`i`,`j`). |
| `i+k` | Value | `k`th value after `i` |

```
    blocked_range(Value begin, Value end, size_t grainsize)
    blocked_range(Value begin, Value end)
```

**Effects:** Constructs a `blocked_range` representing the half-open interval [*`begin`*,*`end`*). If a grain size is not provided, the implementation chooses a grain size, which may vary dynamically.

[*Example:*: Let `i` and `j` be random-access iterators of type T with a size_type convertible to size_t. Then the following constructs a recursive range over [begin,end) that breaks the range into chunks of size less than or equal to 16.

```
    blocked_range<T>(i,j,16)
```

--*end example*]

```
    blocked_range(blocked_range& range, split)
```

**Requires:** *`range.`*`is_divisible()` is true.

**Effects:** Partitions *`range`* into two subranges. The newly constructed `blocked_range` is approximately the second half of the original `range`, and `range` is updated to be the remainder of the original range. The `blocked_range` has the same *`grainsize`* as the original `range`.

[*Example:*: Let `i` and `j` be integers that define a half-open interval [*`i`*,*`j`*) and let `g` specify a grain size. The statements

```
    blocked_range<int> r(i,j,g);
    blocked_range<int> s(r,split);
```

cause r to represent [`i`, `i` +(`j` –`i`)/2) and *`s`* to represent [`i` +(`j` –`i`)/2, `j`), both with grain size `g`.
--*end example*]

```
size_type size() const
```

**Requires:** `!(end()<begin())`

**Returns:** `end()–begin()`

```
bool empty() const
```

**Returns:** `!(begin()<end())`

```
size_type grainsize() const
```

**Returns:** grain size of the `blocked_range` as specified when it was constructed.

```
bool is_divisible() const
```

**Requires:** `!(end()<begin())`

**Returns:** `grainsize()<size()`

```
const_iterator begin() const
```

**Returns:** Inclusive lower bound on range.

```
const_iterator end() const
```

**Returns:** Exclusive upper bound on range.

## 5.5 Template class blocked_range2d

```
namespace std {
  template<typename RowValue, typename ColValue=RowValue>
  class blocked_range2d {
  public:
    // types
    typedef blocked_range<RowValue> row_range_type;
    typedef blocked_range<ColValue> col_range_type;

    // constructors
    blocked_range2d(RowValue row_begin, RowValue row_end,
                    typename row_range_type::size_type row_grainsize,
                    ColValue col_begin, ColValue col_end,
                    typename col_range_type::size_type col_grainsize);
    blocked_range2d(RowValue row_begin, RowValue row_end,
                    ColValue col_begin, ColValue col_end);
    blocked_range2d(blocked_range2d& r, split);

    // capacity
    bool empty() const;

    // access
    bool is_divisible() const;
    const row_range_type& rows() const;
    const col_range_type& cols() const;
  };
}
```

A `blocked_range2d` represents a two dimensional range over the cartesian product $[i_0,j_0)\times[i_1,j_1)$. It satisfies all of the requirements for a recursive range.

```
blocked_range2d<RowValue,ColValue>(RowValue row_begin, RowValue
row_end, typename row_range_type::size_type row_grainsize, ColValue
col_begin, ColValue col_end, typename col_range_type::size_type
col_grainsize)
```

**Effects:** Constructs a `blocked_range2d` representing a two dimensional space of values. The space is the half-open Cartesian product [`row_begin`,`row_end`)× [`col_begin`,`col_end`), with the given grain sizes for the rows and columns.

*[Example:* The following statement constructs a two-dimensional space that contains all value pairs of the form (i, j), where i ranges from `'a'` to `'z'` with a grain size of `3`, and j ranges from 0 to 9 with a grain size of 2.

```
blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2);
```

--*end example*]

```
blocked_range2d<RowValue,ColValue> (blocked_range2d& range, split)
```

**Effects:** Partitions `range` into two subranges. The newly constructed `blocked_range2d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. The split is either by rows or columns. The choice of which axis to split is implementation defined.

```
bool empty() const
```

**Returns:** `rows().empty ()||cols().empty().`

```
bool is_divisible() const
```

**Returns:** `rows().is_divisible()||cols().is_divisible()`

```
const row_range_type& rows() const
```

**Returns:** `blocked_range2d` containing the rows of the value space.

```
const col_range_type& cols() const
```

**Returns:** `blocked_range2d` containing the columns of the value space.

## 5.6 Parallel For

```
template<class RecursiveRange, class Body>
  void parallel_for (const RecursiveRange& range, const Body& body);
template<class RecursiveRange, class Body>
  void parallel_for (const RecursiveRange& range, const Body& body,
                     task_affinity_record& affinity_record);
```

**Requires:** *range* must be RecursiveRange and *body* must be a unary function object such that for RecursiveRange r and const Body b, the expression b(r) applies b to r.
**Effects:** Recursively partitions *range* into subranges $r_i$ such that ! $r_i$.`is_divisible`() and applies *body* to each $r_i$. The partitioning and application may occur concurrently by the invoking thread and other threads, subject to the following:

- An instance of *body* is never applied to more than one $r_i$ concurrently.
- An implementation may invoke the copy constructor for *body*, or a copy thereof, concurrently with application of *body*.

Figure 1 shows the maximal concurrency allowed when processing distinct subranges $r_i$ and $r_j$.

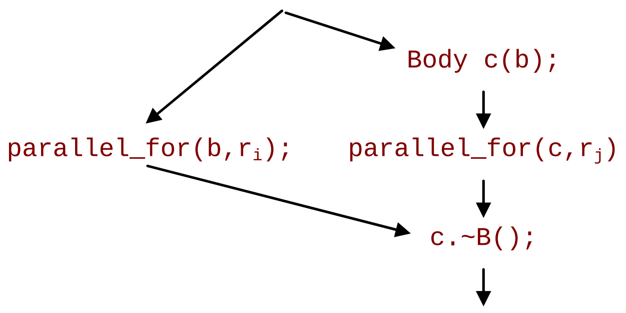

**Figure 1 -- maximal concurrency allowed for parallel_for implementation**

The same instance of *body* may be applied to multiple $r_i$ as long as those applications are totally ordered.

[*Example:* Function `ParallelApplyFoo` evaluates `Foo(i)` for `i` in the range [0,`n`). Each invocation of `operator()` operates on a subrange of ten or fewer values.

```
#include <parallel>
using namespace std;

class ApplyFoo {
public:
    void operator()(const blocked_range<size_t>& r) const {
        for(size_t i=r.begin(); i!=r.end(); ++i)
            Foo(i);
    }
};
void ParallelApplyFoo(size_t n) {
    parallel_for(blocked_range<size_t>(0,n,10), ApplyFoo());
}
```

An implementation is free to process the subranges in parallel, even if concurrent invocations of Foo cause races.--*end example*]

## 5.7 Parallel Reduce

```
template<class RecursiveRange, class Body>
  void parallel_reduce(const RecursiveRange& range, Body& body);
template<class RecursiveRange, class Body>
  void parallel_reduce(const RecursiveRange& range, Body& body,
                       task_affinity_record& affinity_record);
```

**Requires:** In Table 4, R and B are the types used respectively to instantiate RecursiveRange and Body, b and c are values of type B, and r is a value of type R.

**Table 4 --body requirements for parallel_reduce**

| **expression** | **return type** | **requirement** |
|---|---|---|
| `B c(b,split())` | | create new body to process subrange concurrently. |
| `b.~B()` | | |
| `b(r)` | | accumulate result for subrange |
| `b.join(c);` | | update b to summary result for b and c, where c was created by the expression `Body c(b,split())`. |

If r and s are the result of R s(r,split()), the programmer must ensure that the two concurrent execution sequences in Figure 2 have the same net effect; that is, are confluent. The implementation may choose, non-deterministically, which alternative is used for a given range or subrange.

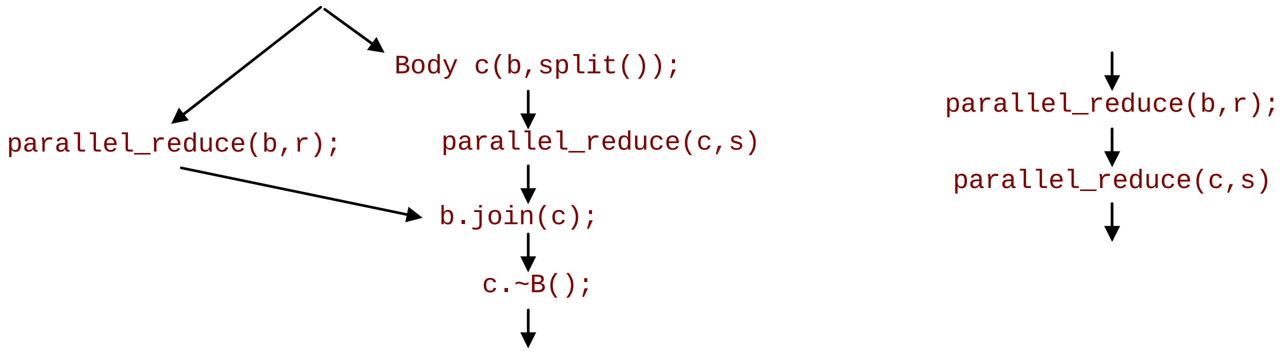

**Figure 2: confluence requirement on arguments to parallel_reduce**

**Effect:** Recursively partitions range into subranges $r_i$ such that `!` $r_i$`.is_divisible()` and applies body to each $r_i$. The implementation may do the partitioning and application concurrently using the invoking thread and other threads. Invocations of the splitting constructor are always paired with a corresponding invocation of `join`.

*[Note: The* function represented by type Body should be associative, but does not have to commutative.]

*[Example:* The following code performs an exclusive-or reduction**:**

```
    #include <parallel>
    using namespace std;

    struct Xor {
        int value;
        Xor() : value(0) {}
        Xor(Xor& s, split) {value = 0;}
        void operator()( const blocked_range<int*>& range ) {
            int temp = value;
            for( int* a=range.begin(); a!=range.end(); ++a ) {
                temp ^= *a;
            }
            value = temp;
        }
        void join(Xor& rhs) {value ^= rhs.value;}
    };

    float ParallelXor( int array[], size_t n ) {
        Xor total;
        parallel_reduce(blocked_range<int*>( array, array+n, 1000 ),
                        total);
        return total.value;
}
```

--*end example*]

## 5.8 Template class parallel_while

```
    namespace std {
```

```
    template<typename Body>
    class parallel_while {
    public:
      parallel_while();
      ~parallel_while();
      typedef typename Body::argument_type value_type;
      template<typename Stream>
        void run( Stream& stream, const Body& body );
      void add( const value_type& item );
    };
  }
```

The class `parallel_while` performs parallel iteration over a collection of items. Items may be added to the initial collection while the parallel iteration is running.

Table 5 shows the requirements on the types Body and Stream used to instantiate the template class and its template member. In Table 5, B is the type used to instantiate Body, b is a value of type B, x is of type B::argument type, and s is a value of the type used to instantiate Stream. Type B::argument type must be copy constructible and have a default constructor.

**Table 5: parallel_while requirements**

| expression | return type | requirements |
|---|---|---|
| `B::argument_type` | argument type of b | |
| `s.pop_if_present(x)` | `bool` | get next item from s and return true; or return false if no more items are available. |
| `b(x)` | | process item `x`. |

```
parallel_while<Body>()
```

**Effects:** Construct a `parallel_while` that is not running.

```
~parallel_while<Body>()
```

**Effects:** Destroy a `parallel_while`.

```
template<typename Stream>
void run( Stream& stream, const Body& body )
```

**Effects:** Repeatedly invokes applies *body* to each item in the associated collection. The associated collection consists of items obtained from *stream*.pop_if_present() and provided by method `add`. Returns when both of the following conditions become true:

1. *stream*.pop_if_present has returned false
2. *body*(*x*) has returned for all items *x* generated from the stream or method add.

An implementation may concurrently invoke *stream*.`pop_if_present(`$x_i$`)` and *body*($x_i$) on the same `parallel_while` object w with distinct $x_i$.

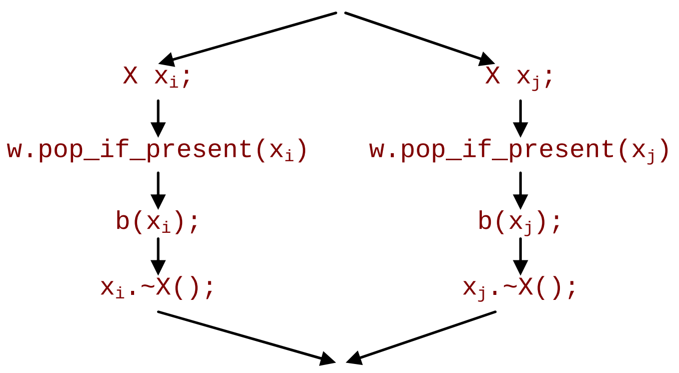

**Figure 3 -- concurrency of parallel_while for i≠j**

```
void add( const value_type& item )
```

**Requires:** Must be called directly or indirectly from an invocation of *body*`.operator()` created by `parallel_while`. Otherwise, behavior is undefined.

**Effects:** Add item to associated collection. Multiple concurrent invocations of w.add(*item*) are permitted for a instance w of `parallel_while`.

## 5.9 Class pipeline

```
namespace std {
  class pipeline {
  public:
    pipeline();
    virtual ~pipeline();
    void add_filter( filter& f );
    void run(size_t max_number_of_live_tokens);
    void clear();
  }
}
```

A `pipeline` represents pipelined application of a series of filters to a stream of items. Each filter is parallel or serial. See class `filter` (5.10) for details.

```
pipeline()
```

**Effects:** Constructs pipeline with no filters.

```
~pipeline()
```

**Effects:** Remove all filters from the pipeline and destroy the pipeline

```
void add_filter(filter& f)
```

**Effects:** Append filter *f* to sequence of filters in the pipeline. If filter *f* is already in a pipeline, the effect is undefined.

```
void run(size_t max_number_of_live_tokens)
```

**Effects:** Runs the pipeline until the first filter returns NULL and each subsequent filter has processed all items from its predecessor. The number of items actually processed in parallel depends upon the structure of the pipeline and number of available threads. At most `max_number_of_live_tokens` are in flight at any given time.

```
void clear()
```

**Effects:** Remove all filters from the pipeline.

### 5.10 Class filter

```
namespace std {
  class filter {
  protected:
    filter(bool is_serial);
  public:
    bool is_serial() const;
    virtual void* operator()( void* item ) = 0;
    virtual ~filter();
  };
}
```

A `filter` represents a filter in a `pipeline`. A filter is parallel or serial. An implementation may apply a parallel filter concurrently or out of order on multiple items. An implementation must apply a serial filter to items one at a time in the original order they were returned by the first filter in the pipeline. [*Note*: Throughput is limited by the throughput of the slowest serial filter. ]

```
filter(bool is_serial)
```

**Effects:** Constructs a serial filter if *`is_serial`* is true, or a parallel filter if *`is_serial`* is false.

```
~filter()
```

**Requires:** The filter must not be in a `pipeline`.

**Effects:** Destroys the filter.

```
bool is_serial() const
```

**Returns:** True iff filter is serial.

```
virtual void* operator()(void * item) = 0
```

**Effects:** The pipeline calls this method to process an item pointed to by *item*. The method shall return a pointer to item to be processed by the next filter in the pipeline. The item parameter shall be NULL for the first filter in the pipeline.

**Returns:** pointer to the result of the filter. The first filter in a pipeline shall return NULL if there are no more items to process. [Note: The result of the last filter in a pipeline is ignored. --end note]

## 6 Acknowledgements

Timmie Smith implemented a task affinity prototype while an intern at Intel. Dave Poulsen, Anton Malakhov, Sanjiv Shah, and Mike Voss provided comments on an earlier draft.

## 7 References

[ABB] U. Acar, G. Blelloch, and R. Blumofe. "The Data Locality of Work Stealing", in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures* (Bar Harbor, Maine, United States, July 09 - 13, 2000). SPAA '00, pp. 1-12.

[BJKLRZ95] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System", *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* PPOPP '95. July 19-21, 1995, Santa Barbara, California, pp. 207-216.

[Boehm06] H. Boehm. *An Atomic Opertions Library for C++*. C++ standards committee document N2047, June 2006.

[Cilk] Cilk Papers. http://supertech.csail.mit.edu/cilk/papers/index.html

[JSR04] J. Järvi, B. Stroustrup, and G. Reis. *Decltype and auto (revision 4).* C++ standards committee document N1705=04-0145, September 2004.

[MSS] S. MacDonald, D. Szafron, and J. Schaeffer. "Rethinking the Pipeline as Object–Oriented States with Transformations", Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04).

[Narlikar99] G.Narlikar and G. Blelloch, "Space-Efficient Scheduling of Nested Parallelism", ACM Transactions on Programming Languages and Systems, 21(1), January 1999, pp. 138-173.

[OpenMP02] *OpenMP C and C++ Application Program Interface*, Version 2.0 March 2002. http://www.openmp.org

[Ottosen06] T. Ottosen. *Range Library Core.* C++ standards committee document N2068, September 2006.

[Samko06] V. Samko. *A proposal to add lambda functions to the C++ standard.* C++ standards committee document N1958=06-028, February 2006.

[STAPL] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger. "STAPL: An Adaptive, Generic Parallel C++ Library", *Workshop on Language and Compilers for Parallel Computing* (LCPC 2001), Cumberland Falls, Kentucky Aug 2001. Lecture Notes in Computer Science 2624 (2003): 193-208. See also STAPL home page at http://parasol.tamu.edu/groups/rwergergroup/research/stapl/.

[TBB06] Reference for Intel® Threading Building Blocks, Version 1.3. http://www.intel.com/support/performancetools/tbb/sb/CS-023279.htm .

[WJGSL06] J. Willcock, J. Järvi, D. Gregor, B. Stroustrup, A. Lumsdaine, *Lambda Expressions and Closures for C++*, N1968-06-0038.