Intel(R) Threading Building Blocks

Reference Manual

Document Number 315415-005US

World Wide Web: http://www.intel.com

Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
315415-005	1.18	Clarify that concurrent operations on `concurrent_hash_map` invalidate iterators.	2009-Oct-14
315415-004	1.17	Revise `task_group::run_and_wait` signatures. Reorder presentation of `task_group.h` interfaces. Clarify exception safety for `concurrent_vector`. Add constructor of `concurrent_hash_map` that preallocates buckets. Add `scalable_msize`.	2009-Sep-21
315415-003	1.16	Relax task ownership constraints.	2009-Jul-29

Document Number	Revision Number	Description	Revision Date
315415-002	1.15	Type atomic<T> allows T to be an enumeration type. Add appendices B and C. Add zero_allocator. Change return types for concurrent_vector methods push_back, grow_by and grow_to_at_least. Rename concurrent_vector method compact as shrink_to_fit. Remove depth methods from class task. Add methods increment_ref_count and decrement_ref_count to class task. Change default partitioner from simple_partitioner to auto_partitioner. Add tbb_thread::operator=. Add parallel_invoke. Add task_group. Add combinable, enumerable_thread_specific, and flattened2d. Add thread_bound_filter. Add parallel_for_each. Add tbb_hash_compare and tbb_hasher.	2009-Jun-25
	1.14	Clarify semantics of concurrent_hash_map methods insert and erase.	2009-Mar-13
	1.13	Update copyright to 2009. Remove requirement to clear a pipeline before destroying its filters. Add null_mutex and null_rw_mutex.	2009-Feb-6
	1.12	Add task_scheduler_init::is_active(). Document new debugging macros and deprecate old ones. Add filter::serial_out_of_order and filter::finalize(). Describe automatic reset of cancellation state by task::wait_for_all(). Clarify behavior of concurrent_vector::clear(). Add TBB_runtime_interface_version(). Add new constructors to concurrent_queue. Consistently use typename keyword for template parameters.	2008-Dec-5

Contents

1 Overview

Intel® Threading Building Blocks (Intel® TBB) is a library that supports scalable parallel programming using standard ISO C++ code. It does not require special languages or compilers. It is designed to promote scalable data parallel programming. Additionally, it fully supports nested parallelism, so you can build larger parallel components from smaller parallel components. To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner.

Many of the library interfaces employ generic programming, in which interfaces are defined by requirements on types and not specific types. The C++ Standard Template Library (STL) is an example of generic programming. Generic programming enables Intel® Threading Building Blocks to be flexible yet efficient. The generic interfaces enable you to customize components to your specific needs.

The net result is that Intel® Threading Building Blocks enables you to specify parallelism far more conveniently than using raw threads, and at the same time can improve performance.

This document is a reference manual. It is organized for looking up details about syntax and semantics. You should first read the *Intel® Threading Building Blocks Getting Started Guide* and the *Intel® Threading Building Blocks Tutorial* to learn how to use the library effectively.

TIP: Even experienced parallel programmers should read the *Intel® Threading Building Blocks Tutorial* before using this reference guide because Intel® Threading Building Blocks uses a surprising recursive model of parallelism and generic algorithms.

2 General Conventions

This section describes conventions used in this document.

2.1 Notation

Literal program text appears in Courier font. Algebraic placeholders are in *monospace italics*. For example, the notation blocked_range<*Type*> indicates that blocked_range is literal, but Type is a notational placeholder. Real program text replaces *Type* with a real type, such as in blocked_range<int>.

Class members are summarized by informal class declarations that describe the class as it seems to clients, not how it is actually implemented. For example, here is an informal declaration of class Foo:

```
class Foo {
public:
    int x();
    int y;
    ~Foo();
};
```

The actual implementation might look like:

```
namespace internal {
    class FooBase {
    protected:
        int x();
    };

    class Foo_v3: protected FooBase {
    private:
        int internal_stuff;
    public:
        using FooBase::x;
        int y;
    };
}

typedef internal::Foo_v3 Foo;
```

The example shows two cases where the actual implementation departs from the informal declaration:

- Foo is actually a typedef to Foo_v3.

- Method `x()` is inherited from a protected base class.
- The destructor is an implicit method generated by the compiler.

The informal declarations are intended to show you what you need to know to use the class without the distraction of irrelevant clutter particular to the implementation.

2.2 Terminology

This section describes terminology specific to Intel® Threading Building Blocks (Intel® TBB).

2.2.1 Concept

A *concept* is a set of requirements on a type. The requirements may be syntactic or semantic. For example, the concept of "sortable" could be defined as a set of requirements that enable an array to be sorted. A type `T` would be sortable if:

- `x < y` returns a boolean value, and represents a total order on items of type `T`.
- `swap(x,y)` swaps items `x` and `y`

You can write a sorting template function in C++ that sorts an array of any type that is sortable.

Two approaches for defining concepts are *valid expressions* and *pseudo-signatures*[1]. The ISO C++ standard follows the valid expressions approach, which shows what the usage pattern looks like for a concept. It has the drawback of relegating important details to notational conventions. This document uses pseudo-signatures, because they are concise, and can be cut-and-pasted for an initial implementation.

For example, Table 1 shows pseudo-signatures for a sortable type `T`:

Table 1: Pseudo-Signatures for Example Concept "sortable"

Pseudo-Signature	Semantics
`bool operator<(const T& x, const T& y)`	Compare `x` and `y`.
`void swap(T& x, T& y)`	Swap `x` and `y`.

A real signature may differ from the pseudo-signature that it implements in ways where implicit conversions would deal with the difference. For an example type `U`, the real signature that implements operator< in Table 1 can be expressed as `int operator<( U x, U y )`, because C++ permits implicit conversion from `int` to `bool`,

[1] See Section 3.2.3 of *Concepts for C++0x* available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf for further discussion of valid expressions versus pseudo-signatures.

and implicit conversion from `U` to `(const U&)`. Similarly, the real signature `bool operator<( U& x, U& y )` is acceptable because C++ permits implicit addition of a const qualifier to a reference type.

2.2.2 Model

A type *models* a concept if it meets the requirements of the concept. For example, type `int` models the sortable concept in Table 1 if there exists a function `swap(x,y)` that swaps two `int` values `x` and `y`. The other requirement for sortable, specifically `x<y`, is already met by the built-in `operator<` on type `int`.

2.2.3 CopyConstructible

The library sometimes requires that a type model the `CopyConstructible` concept, which is defined by the ISO C++ standard. Table 2 shows the requirements for `CopyConstructible` in pseudo-signature form.

Table 2: CopyConstructible Requirements

Pseudo-Signature	Semantics
`T( const T& )`	Construct copy of const T.
`~T()`	Destructor.
`T* operator&()`	Take address.
`const T* operator&() const`	Take address of const T.

2.3 Identifiers

This section describes the identifier conventions used by Intel® Threading Building Blocks.

2.3.1 Case

The identifier convention in the library follows the style in the ISO C++ standard library. Identifiers are written in underscore_style, and concepts in PascalCase.

2.3.2 Reserved Identifier Prefixes

The library reserves the prefix `__TBB` for internal identifiers and macros that should never be directly referenced by your code.

2.4 Namespaces

This section describes reserved namespaces used by Intel® Threading Building Blocks.

2.4.1 tbb Namespace

The library puts all public classes and functions into the namespace `tbb`.

2.4.2 tbb::internal Namespace

The library uses the namespace `tbb::internal` for internal identifiers. Client code should never directly reference the namespace `tbb::internal` or the identifiers inside it. Indirect reference via a public `typedef` provided by the header files is permitted.

An example of the distinction between direct and indirect use is type `concurrent_vector<T>::iterator`. This type is a `typedef` for an internal class `internal::vector_iterator<Container,Value>`. Your source code should use the `iterator` typedef.

2.4.3 tbb::deprecated Namespace

The library uses the namespace `tbb::deprecated` for deprecated identifiers that have different default meanings in namespace `tbb`. Compiling with `TBB_DEPRECATED=1` causes such identifiers to replace their counterpart in namespace `tbb`.

For example, `tbb::concurrent_queue` underwent changes in Intel® TBB 2.2 that split its functionality into `tbb::concurrent_queue` and `tbb::concurrent_bounded_queue` and changed the name of some methods. For sake of legacy code, the old Intel® TBB 2.1 functionality is retained in `tbb::deprecated::concurrent_queue`, which is injected into namespace tbb when compiled with `TBB_DEPRECATED=1`.

2.4.4 tbb::strict_ppl

The library uses the namespace `tbb::strict_ppl` for identifiers that are put in namespace `Concurrency` when `tbb/compat/ppl.h` is included.

2.5 Thread Safety

Unless otherwise stated, the thread safety rules for the library are as follows:

- Two threads can invoke a method or function concurrently on different objects, but not the same object.
- It is unsafe for two threads to invoke concurrently methods or functions on the same object.

Descriptions of the classes note departures from this convention. For example, the concurrent containers are more liberal. By their nature, they do permit some concurrent operations on the same container object.

2.6 Enabling Debugging Features

Four macros control certain debugging features. In general, it is useful to compile with these features on for development code, and off for production code, because the features may decrease performance. Table 3 summarizes the macros and their default values. A value of 1 enables the corresponding feature; a value of 0 disables the feature.

Table 3: Debugging Macros

Macro	Default Value	Feature
TBB_USE_DEBUG	Windows* OS: 1 if _DEBUG is defined, 0 otherwise. All other systems: 0.	Default value for all other macros in this table.
TBB_USE_ASSERT	TBB_USE_DEBUG	Enable internal assertion checking. Can significantly slow performance.
TBB_USE_THREADING_TOOLS		Enable full support for Intel® Parallel Studio and Intel® Threading Tools.
TBB_USE_PERFORMANCE_WARNINGS		Enable warnings about performance issues.

2.6.1 TBB_USE_ASSERT Macro

The macro TBB_USE_ASSERT controls whether error checking is enabled in the header files. Define TBB_USE_ASSERT as 1 to enable error checking.

If an error is detected, the library prints an error message on `stderr` and calls the standard C routine `abort`. To stop a program when internal error checking detects a failure, place a breakpoint on `tbb::assertion_failure`.

TIP: On Windows* operating systems, debug builds implicitly set TBB_USE_ASSERT to 1 by default

2.6.2 TBB_USE_THREADING_TOOLS Macro

The macro TBB_USE_THREADING_TOOLS controls support for Intel® Threading Tools:

- Intel® Parallel Inspector
- Intel® Parallel Amplifier
- Intel® Thread Profiler
- Intel® Thread Checker.

Define TBB_USE_THREADING_TOOLS as 1 to enable full support for these tools.

That is full support is enabled if error checking is enabled. Leave TBB_USE_THREADING_TOOLS undefined or zero to enable top performance in release builds, at the expense of turning off some support for tools.

2.6.3 TBB_USE_PERFORMANCE_WARNINGS Macro

The macro TBB_USE_PERFORMANCE_WARNINGS controls performance warnings. Define it to be 1 to enable the warnings. Currently, the warnings affected are:

- Some that report poor hash functions for concurrent_hash_map. Enabling the warnings may impact performance.
- Misaligned 8-byte atomic stores on Intel® IA-32 processors.

2.7 Version Information

Intel® TBB has macros, an environment variable, and a function that reveal version and run-time information.

2.7.1 Version Macros

The header tbb/tbb_stddef.h defines macros related to versioning, as described in Table 4. You should not redefine these macros.

Table 4: Version Macros

Macro	Description of Value
TBB_INTERFACE_VERSION	Current interface version. The value is a decimal numeral of the form *xyyy* where *x* is the major version number and *y* is the minor version number.
TBB_INTERFACE_VERSION_MAJOR	TBB_INTERFACE_VERSION/1000; i.e., the major version number.
TBB_COMPATIBLE_INTERFACE_VERSION	Oldest major interface version still supported.

2.7.2 TBB_VERSION Environment Variable

Set the environment variable TBB_VERSION to 1 to cause the library to print information on stderr. Each line is of the form "TBB: *tag value*", where *tag* and *value are* described in Table 5.

Table 5: Output from TBB_VERSION

Tag	Description of Value
VERSION	TBB product version number.
INTERFACE_VERSION	Value of macro TBB_INTERFACE_VERSION when library was compiled.
BUILD_...	Various information about the machine configuration on which the library was built.
TBB_USE_ASSERT	Setting of macro TBB_USE_ASSERT
DO_ITT_NOTIFY	1 if library can enable instrumentation for Intel® Parallel Studio and Intel® Threading Tools; 0 or undefined otherwise.
ITT	yes if library has enabled instrumentation for Intel® Parallel Studio and Intel® Threadng Tools, no otherwise. Typically yes only if the program is running under control of Intel® Parallel Studio or Intel® Threadng Tools.
ALLOCATOR	Underlying allocator for tbb::tbb_allocator. It is scalable_malloc if the Intel® TBB malloc library was successfully loaded; malloc otherwise.

CAUTION: This output is implementation specific and may change at any time.

2.7.3 TBB_runtime_interface_version Function

Summary

Function that returns the interface version of the Intel® TBB library that was loaded at runtime.

Syntax

```
extern "C" int TBB_runtime_interface_version();
```

Header

```
#include "tbb/tbb_stddef.h"
```

Description

The value returned by TBB_runtime_interface_version() may differ from the value of TBB_INTERFACE_VERSION obtained at compile time. This can be used to identify whether an application was compiled against a compatible version of the Intel® TBB headers.

CAUTION: In general, the run-time value `TBB_runtime_interface_version()` must be greater than or equal to the compile-time value of `TBB_INTERFACE_VERSION`. Otherwise the application may fail to resolve all symbols at run time.

2.8 TBB_DEPRECATED macro

The macro `TBB_DEPRECATED` controls deprecated features that would otherwise conflict with non-deprecated use. Define it to be 1 to get deprecated Intel® TBB 2.1 interfaces. Appendix A describes deprecated features.

3 Algorithms

Most parallel algorithms provided by Intel® Threading Building Blocks (Intel® TBB) are generic. They operate on all types that model the necessary concepts. Parallel algorithms may be nested. For example, the body of a `parallel_for` can invoke another `parallel_for`.

CAUTION: When the body of an outer parallel algorithm invokes another parallel algorithm, it may cause the outer body to be re-entered for a different iteration of the outer algorithm.

For example, if the outer body holds a global lock while calling an inner parallel algorithm, the body will deadlock if the re-entrant invocation attempts to acquire the same global lock. This ill-formed example is a special case of a general rule that code should not hold a lock while calling code written by another author.

3.1 Splittable Concept

Summary

Requirements for a type whose instances can be split into two pieces.

Requirements

Table 6 lists the requirements for a splittable type X with instance x.

Table 6: Splittable Concept

Pseudo-Signature	Semantics
`X::X(X& x, Split)`	Split x into x and newly constructed object.

Description

A type is splittable if it has a *splitting constructor* that allows an instance to be split into two pieces. The splitting constructor takes as arguments a reference to the original object, and a dummy argument of type `Split`, which is defined by the library. The dummy argument distinguishes the splitting constructor from a copy constructor. After the constructor runs, *x* and the newly constructed object should represent the two pieces of the original *x.* The library uses splitting constructors in three contexts:

- *Partitioning* a range into two subranges that can be processed concurrently.
- *Forking* a body (function object) into two bodies that can run concurrently.

The following model types provide examples.

Model Types

blocked_range (3.2.1) and blocked_range2d (3.2.2) represent splittable ranges. For each of these, splitting partitions the range into two subranges. See the example in Section 3.2.1.3 for the splitting constructor of blocked_range<Value>.

The bodies for parallel_reduce (3.5) and parallel_scan (3.6) must be splittable. For each of these, splitting results in two bodies that can be run concurrently.

3.1.1 split Class

Summary

Type for dummy argument of a splitting constructor.

Syntax

```
class split;
```

Header

```
#include "tbb/tbb_stddef.h"
```

Description

An argument of type split is used to distinguish a splitting constructor from a copy constructor.

Members

```
namespace tbb {
    class split {
    };
}
```

3.2 Range Concept

Summary

Requirements for type representing a recursively divisible set of values.

Requirements

Table 7 lists the requirements for a Range type R.

Table 7: Range Concept

Pseudo-Signature	Semantics
R::R(const R&)	Copy constructor.

Pseudo-Signature	Semantics
`R::~R()`	Destructor.
`bool R::empty() const`	True if range is empty.
`bool R::is_divisible() const`	True if range can be partitioned into two subranges.
`R::R( R& r, split )`	Split `r` into two subranges.

Description

A Range can be recursively subdivided into two parts. It is recommended that the division be into nearly equal parts, but it is not required. Splitting as evenly as possible typically yields the best parallelism. Ideally, a range is recursively splittable until the parts represent portions of work that are more efficient to execute serially rather than split further. The amount of work represented by a Range typically depends upon higher level context, hence a typical type that models a Range should provide a way to control the degree of splitting. For example, the template class `blocked_range` (3.2.1) has a *grainsize* parameter that specifies the biggest range considered indivisible.

The constructor that implements splitting is called a *splitting constructor*. If the set of values has a sense of direction, then by convention the splitting constructor should construct the second part of the range, and update the argument to be the first half. Following this convention causes the `parallel_for` (3.4), `parallel_reduce` (3.5), and `parallel_scan` (3.6) algorithms, when running sequentially, to work across a range in the increasing order typical of an ordinary sequential loop.

Example

The following code defines a type `TrivialIntegerRange` that models the Range concept. It represents a half-open interval [lower,upper) that is divisible down to a single integer.

```
struct TrivialIntegerRange {
    int lower;
    int upper;
    bool empty() const {return lower==upper;}
    bool is_divisible() const {return upper>lower+1;}
    TrivialIntegerRange( TrivialIntegerRange& r, split ) {
        int m = (r.lower+r.upper)/2;
        lower = m;
        upper = r.upper;
        r.upper = m;
    }
};
```

`TrivialIntegerRange` is for demonstration and not very practical, because it lacks a grainsize parameter. Use the library class `blocked_range` instead.

Model Types

Type blocked_range (3.2.1) models a one-dimensional range.

Type blocked_range2d (3.2.2) models a two-dimensional range.

Type blocked_range3d (3.2.3) models a three-dimensional range.

Concept Container Range (4.1) models a container as a range.

3.2.1 blocked_range<Value> Template Class

Summary

Template class for a recursively divisible half-open interval.

Syntax

```
template<typename Value> class blocked_range;
```

Header

```
#include "tbb/blocked_range.h"
```

Description

A blocked_range<Value> represents a half-open range [*i*,*j*) that can be recursively split. The types of *i* and *j* must model the requirements in Table 8. Because the requirements are pseudo-signatures, signatures that differ by implicit conversions are allowed. For example, a blocked_range<int> is allowed, because the difference of two int values can be implicitly converted to a size_t. Examples that model the Value requirements are integral types, pointers, and STL random-access iterators whose difference can be implicitly converted to a size_t.

A blocked_range models the Range concept (3.2).

Table 8: Value Concept for blocked_range

Pseudo-Signature	Semantics
Value::Value(const Value&)	Copy constructor.
Value::~Value()	Destructor.
bool operator<(const Value& i, const Value& j)	Value *i* precedes value j.
size_t operator−(const Value& i, const Value& j)	Number of values in range [i,j).
Value operator+(const Value& i, size_t k)	*k*th value after *i*.

A blocked_range<Value> specifies a *grainsize* of type size_t. A blocked_range is splittable into two subranges if the size of the range exceeds *grain size*. The ideal grain size depends upon the context of the blocked_range<Value>, which is typically as the range argument to the loop templates parallel_for, parallel_reduce, or parallel_scan. A too small grainsize may cause scheduling overhead within the loop templates to swamp speedup gained from parallelism. A too large grainsize may unnecessarily limit parallelism. For example, if the grain size is so large that the range can be split only once, then the maximum possible parallelism is two.

Here is a suggested procedure for choosing *grainsize*:

1. Set the grainsize parameter to 10,000. This value is high enough to amortize scheduler overhead sufficiently for practically all loop bodies, but may be unnecessarily limit parallelism.
2. Run your algorithm on *one* processor.
3. Start halving the grainsize parameter and see how much the algorithm slows down as the value decreases.

A slowdown of about 5-10% is a good setting for most purposes.

TIP: For a blocked_range [i,j) where j<i, not all methods have specified behavior. However, enough methods do have specified behavior that parallel_for (3.4), parallel_reduce (3.5), and parallel_scan (3.6) iterate over the same iteration space as the serial loop for(Value index=i; index<j; ++index)... , even when j<i. If TBB_USE_ASSERT (2.6.1) is nonzero, methods with unspecified behavior raise an assertion failure.

Examples

A blocked_range<Value> typically appears as a range argument to a loop template. See the examples for parallel_for (3.4), parallel_reduce (3.5), and parallel_scan (3.6).

Members

```
namespace tbb {
    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=1);
        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;
    };
}
```

3.2.1.1 size_type

Description

The type for measuring the size of a blocked_range. The type is always a size_t.

```
const_iterator
```

Description

The type of a value in the range. Despite its name, the type const_iterator is not necessarily an STL iterator; it merely needs to meet the Value requirements in Table 8. However, it is convenient to call it const_iterator so that if it is a const_iterator, then the blocked_range behaves like a read-only STL container.

3.2.1.2 blocked_range(Value begin, Value end, size_t grainsize=1)

Requirements

The parameter grainsize must be positive. The debug version of the library raises an assertion failure if this requirement is not met.

Effects

Constructs a blocked_range representing the half-open interval [begin,end) with the given grainsize.

Example

The statement "blocked_range<int> r(5, 14, 2);" constructs a range of int that contains the values 5 through 13 inclusive, with a grainsize of 2. Afterwards, r.begin()==5 and r.end()==14.

3.2.1.3 blocked_range(blocked_range& range, split)

Requirements

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. Each subrange 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 specifiy a grain size. The statement blocked_range<int> r(i,j,g) constructs a blocked_range<int> that represents [i,j) with grain size g. Running the statement blocked_range<int> s(r,split); subsequently causes r to represent [i, i +(j −i)/2) and s to represent [i +(j −i)/2, j), both with grain size g.

3.2.1.4 size_type size() const

Requirements

end()<begin() is false.

Effects

Determines size of range.

Returns

end()−begin()

3.2.1.5 bool empty() const

Effects

Determines if range is empty.

Returns

!(begin()<end())

3.2.1.6 size_type grainsize() const

Returns

Grain size of range.

3.2.1.7 bool is_divisible() const

Requirements

```
!(end()<begin())
```

Effects

Determines if range can be split into subranges.

Returns

True if `size()>grainsize()`; false otherwise.

3.2.1.8 const_iterator begin() const

Returns

Inclusive lower bound on range.

3.2.1.9 const_iterator end() const

Returns

Exclusive upper bound on range.

3.2.2 blocked_range2d Template Class

Summary

Template class that represents recursively divisible two-dimensional half-open interval.

Syntax

```
template<typename RowValue, typename ColValue> class
blocked_range2d;
```

Header

```
#include "tbb/blocked_range2d.h"
```

Description

A blocked_range2d<*RowValue*,*ColValue*> represents a half-open two dimensional range $[i_0,j_0)\times[i_1,j_1)$. Each axis of the range has its own splitting threshold. The *RowValue* and *ColValue* must meet the requirements in Table 8. A blocked_range is splittable if either axis is splittable. A blocked_range models the Range concept (3.2).

Members

```
namespace tbb {
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;
    };
}
```

Example

The code that follows shows a serial matrix multiply, and the corresponding parallel matrix multiply that uses a blocked_range2d to specify the iteration space.

```
const size_t L = 150;
const size_t M = 225;
const size_t N = 300;

void SerialMatrixMultiply( float c[M][N], float a[M][L], float
b[L][N] ) {
    for( size_t i=0; i<M; ++i ) {
        for( size_t j=0; j<N; ++j ) {
            float sum = 0;
            for( size_t k=0; k<L; ++k )
                sum += a[i][k]*b[k][j];
            c[i][j] = sum;
        }
    }
}
```

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
```

```
using namespace tbb;

const size_t L = 150;
const size_t M = 225;
const size_t N = 300;

class MatrixMultiplyBody2D {
    float (*my_a)[L];
    float (*my_b)[N];
    float (*my_c)[N];
public:
    void operator()( const blocked_range2d<size_t>& r ) const {
        float (*a)[L] = my_a;
        float (*b)[N] = my_b;
        float (*c)[N] = my_c;
        for( size_t i=r.rows().begin(); i!=r.rows().end(); ++i ){
            for( size_t j=r.cols().begin(); j!=r.cols().end();
++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k]*b[k][j];
                c[i][j] = sum;
            }
        }
    }
    MatrixMultiplyBody2D( float c[M][N], float a[M][L], float
b[L][N] ) :
        my_a(a), my_b(b), my_c(c)
    {}
};

void ParallelMatrixMultiply(float c[M][N], float a[M][L], float
b[L][N]){
    parallel_for( blocked_range2d<size_t>(0, M, 16, 0, N, 32),
                  MatrixMultiplyBody2D(c,a,b) );
}
```

The `blocked_range2d` enables the two outermost loops of the serial version to become parallel loops. The `parallel_for` recursively splits the `blocked_range2d` until the pieces are no larger than 16×32. It invokes `MatrixMultiplyBody2D::operator()` on each piece.

3.2.2.1 row_range_type

Description

A `blocked_range<RowValue>`. That is, the type of the row values.

3.2.2.2 col_range_type

Description

A blocked_range<ColValue>. That is, the type of the column values.

3.2.2.3 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 statement “blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2);” 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.

3.2.2.4 blocked_range2d<RowValue,ColValue>(RowValue row_begin, RowValue row_end, ColValue col_begin, ColValue col_end)

Effects

Same as blocked_range2d(row_begin,row_end,1,col_begin,col_end,1).

3.2.2.5 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 intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective row and column grain sizes. For example, if the row_grainsize is twice col_grainsize, the subranges will tend towards having twice as many rows as columns.

3.2.2.6 bool empty() const

Effects

Determines if range is empty.

Returns

```
rows().empty()||cols().empty()
```

3.2.2.7 bool is_divisible() const

Effects

Determines if range can be split into subranges.

Returns

```
rows().is_divisible()||cols().is_divisible()
```

3.2.2.8 const row_range_type& rows() const

Returns

Range containing the rows of the value space.

3.2.2.9 const col_range_type& cols() const

Returns

Range containing the columns of the value space.

3.2.3 blocked_range3d Template Class

Summary

Template class that represents recursively divisible three-dimensional half-open interval.

Syntax

```
template<typename PageValue, typename RowValue, typename
ColValue> class blocked_range3d;
```

Header

```
#include "tbb/blocked_range3d.h"
```

Description

A blocked_range3d<*PageValue*,*RowValue*,*ColValue*> is the three-dimensional extension of blocked_range2d.

Members

```
namespace tbb {
template<typename PageValue, typename RowValue=PageValue,
typename ColValue=RowValue>
    class blocked_range2d {
    public:
        // Types
        typedef blocked_range<ColValue> page_range_type;
        typedef blocked_range<RowValue> row_range_type;
        typedef blocked_range<ColValue> col_range_type;

        // Constructors
        blocked_range3d( PageValue page_begin, PageValue
page_end,
                         typename page_range_type::size_type
page_grainsize,
                         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_range3d( PageValue page_begin, PageValue
page_end,
                         RowValue row_begin, RowValue row_end,
                         ColValue col_begin, ColValue col_end);
        blocked_range3d( blocked_range2d& r, split );

        // Capacity
        bool empty() const;

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

3.3 Partitioners

Summary

A partitioner specifies how a loop template should partition its work among threads.

Description

The default behavior of the loop templates parallel_for (3.4), parallel_reduce (3.5), and parallel_scan (3.6) tries to recursively split a range into enough parts to keep processors busy, not necessarily splitting as finely as possible. An optional partitioner parameter enables other behaviors to be specified, as shown in Table 9. The first column of the table shows how the formal parameter is declared in the loop templates. An affinity_partitioner is passed by non-const reference because it is updated to remember where loop iterations run.

Table 9: Partitioners

Partitioner	Loop Behavior
const **auto_partitioner**& (default)[2]	Performs sufficient splitting to balance load, not necessarily splitting as finely as Range::is_divisible permits. When used with classes such as blocked_range, the selection of an appropriate grainsize is less important, and often acceptable performance can be achieved with the default grain size of 1.
affinity_partitioner&	Similar to auto_partitioner, but improves cache affinity by its choice of mapping subranges to worker threads. It can improve performance significantly when a loop is re-executed over the same data set, and the data set fits in cache.
const **simple_partitioner**&	Recursively splits a range until it is no longer divisible. The Range::is_divisible function is wholly responsible for deciding when recursive splitting halts. When used with classes such as blocked_range, the selection of an appropriate grainsize is critical to enabling concurrency while limiting overheads (see the discussion in Section 3.2.1).

3.3.1 auto_partitioner Class

Summary

Specify that a parallel loop should optimize its range subdivision based on work-stealing events.

---

[2] In Intel® TBB 2.1, simple_partitioner was the default. Intel® TBB 2.2 changed the default to auto_partitioner to simplify common usage of the loop templates. To get the old default, compile with the preprocessor symbol TBB_DEPRECATED=1.

Syntax

```
class auto_partitioner;
```

Header

```
#include "tbb/partitioner.h"
```

Description

A loop template with an auto_partitioner attempts to minimize range splitting while providing ample opportunities for work-stealing.

The range subdivision is initially limited to S subranges, where S is proportional to the number of threads specified by the task_scheduler_init (10.2.1). Each of these subranges is not divided further unless it is stolen by an idle thread. If stolen, it is further subdivided to create additional subranges. Thus a loop template with an auto_partitioner creates additional subranges only when necessary to balance load.

TIP: When using auto_partitioner and a blocked_range for a parallel loop, the body may be passed a subrange larger than the blocked_range's grainsize. Therefore do not assume that the grainsize is an upper bound on the size of the subrange. Use a simple_partitioner if an upper bound is required.

Members

```
namespace tbb {
    class auto_partitioner {
    public:
        auto_partitioner();
        ~auto_partitioner();
    }
}
```

3.3.1.1 auto_partitioner()

Construct an auto_partitioner.

3.3.1.2 ~auto_partitioner()

Destroy this auto_partitioner.

3.3.2 affinity_partitioner

Summary

Hint that loop iterations should be assigned to threads in a way that optimizes for cache affinity.

Syntax

```
class affinity_partitioner;
```

Header

```
#include "tbb/partitioner.h"
```

Description

An `affinity_partitioner` hints that execution of a loop template should assign iterations to the same processors as another execution of the loop (or another loop) with the same `affinity_partitioner` object.

Unlike the other partitioners, it is important that the same `affinity_partitioner` object be passed to the loop templates to be optimized for affinity. The Tutorial (Section 3.2.3 “Bandwidth and Cache Affinity”) discusses affinity effects in detail.

TIP: The affinity_partitioner generally improves performance only when:

- The computation does a few operations per data access.
- The data acted upon by the loop fits in cache.
- The loop, or a similar loop, is re-executed over the same data.
- There are more than two hardware threads available.

Members

```
namespace tbb {
    class affinity_partitioner {
    public:
        affinity_partitioner();
        ~affinity_partitioner();
    }
}
```

Example

The following example can benefit from cache affinity. The example simulates a one dimensional additive automaton.

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#include "tbb/partitioner.h"

using namespace tbb;

const int N = 1000000;
typedef unsigned char Cell;
Cell Array[2][N];
int FlipFlop;
```

```
struct TimeStepOverSubrange {
    void operator()( const blocked_range<int>& r ) const {
        int j = r.end();
        const Cell* x = Array[FlipFlop];
        Cell* y = Array[!FlipFlop];
        for( int i=r.begin(); i!=j; ++i )
            y[i] = x[i]^x[i+1];
    }
};


void DoAllTimeSteps( int m ) {
    affinity_partitioner ap;
    for( int k=0; k<m; ++k ) {
        parallel_for( blocked_range<int>(0,N-1),
                      TimeStepOverSubrange(),
                      ap );
        FlipFlop ^= 1;
    }
}
```

For each time step, the old state of the automaton is read from Array[FlipFlop], and the new state is written into Array[!FlipFlop]. Then FlipFlop flips to make the new state become the old state. The aggregate size of both states is about 2 MByte, which fits in most modern processors' cache. Improvements ranging from 50%-200% have been observed for this example on 8 core machines, compared with using an auto_partitioner instead.

The affinity_partitioner must live between loop iterations. The example accomplishes this by declaring it outside the loop that executes all iterations. An alternative would be to declare the affinity partitioner at the file scope, which works as long as DoAllTimeSteps itself is not invoked concurrently. The same instance of affinity_partitioner should not be passed to two parallel algorithm templates that are invoked concurrently. Use separate instances instead.

3.3.2.1 affinity_partitioner()

Construct an affinity_partitioner.

3.3.2.2 ~affinity_partitioner()

Destroy this affinity_partitioner.

3.3.3 simple_partitioner Class

Summary

Specify that a parallel loop should recursively split its range until it cannot be subdivided further.

Syntax

```
class simple_partitioner;
```

Header

```
#include "tbb/partitioner.h"
```

Description

A `simple_partitioner` specifies that a loop template should recursively divide its range until for each subrange *r*, the condition `!r.is_divisible()` holds. This is the default behavior of the loop templates that take a range argument.

TIP: When using `simple_partitioner` and a `blocked_range` for a parallel loop, be careful to specify an appropriate grainsize for the blocked_range. The default grainsize is 1, which may make the subranges much too small for efficient execution.

Members

```
namespace tbb {
    class simple_partitioner {
    public:
        simple_partitioner();
        ~simple_partitioner();
    }
}
```

3.3.3.1 simple_partitioner()

Construct a `simple_partitioner`.

3.3.3.2 ~simple_partitioner()

Destroy this `simple_partitioner`.

3.4 parallel_for Template Function

Summary

Template function that performs parallel iteration over a range of values.

Syntax

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index_type last, const Func& f);

template<typename Index, typename Func>
Func parallel_for(Index first, Index_type last,
                  Index step, const Func& f);

template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body,
                   [, partitioner] );
```

where the optional *partitioner* declares any of the partitioners as shown in column 1 of Table 9.

Header

```
#include "tbb/parallel_for.h"
```

Description

A parallel_for(*first*, *last*,*step*,*f*) represents parallel execution of the loop:

for(auto *i*=*first*; *i*<*last*; *i*+=step) *f*(*i*);

The index type must be an integral type. The loop must not wrap around. The step value must be positive. If omitted, it is implicitly 1. There is no guarantee that the iterations run in parallel. Deadlock may occur if a lesser iteration waits for a greater iteration. The partitioning strategy is always auto_partitioner.

A parallel_for(*range*,*body*,*partitioner*) provides a more general form of parallel iteration. It represents parallel execution of *body* over each value in *range*. The optional *partitioner* specifies a partitioning strategy. Type Range must model the Range concept (3.2). The body must model the requirements in Table 10.

Table 10: Requirements for parallel_for Body

Pseudo-Signature	Semantics
Body::Body(const Body&)	Copy constructor.
Body::~Body()	Destructor.
void Body::operator()(Range& range) const	Apply body to range.

A parallel_for recursively splits the range into subranges to the point such that is_divisible() is false for each subrange, and makes copies of the body for each of these subranges. For each such body/subrange pair, it invokes Body::operator(). The invocations are interleaved with the recursive splitting, in order to minimize space overhead and efficiently use cache.

Some of the copies of the range and body may be destroyed after parallel_for returns. This late destruction is not an issue in typical usage, but is something to be aware of when looking at execution traces or writing range or body objects with complex side effects.

When worker threads are available (10.2), `parallel_for` executes iterations is non-deterministic order. Do not rely upon any particular execution order for correctness. However, for efficiency, do expect `parallel_for` to tend towards operating on consecutive runs of values.

When no worker threads are available, `parallel_for` executes iterations from left to right in the following sense. Imagine drawing a binary tree that represents the recursive splitting. Each non-leaf node represents splitting a subrange r by invoking the splitting constructor `Range(r,split())`. The left child represents the updated value of r. The right child represents the newly constructed object. Each leaf in the tree represents an indivisible subrange. The method `Body::operator()` is invoked on each leaf subrange, from left to right.

Complexity

If the range and body take O(1) space, and the range splits into nearly equal pieces, then the space complexity is O(P log(N)), where N is the size of the range and P is the number of threads.

Example

This example defines a routine `ParallelAverage` that sets `output[i]` to the average of `input[i-1]`, `input[i]`, and `input[i+1]`, for $1 \le i < n$.

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Average {
    const float* input;
    float* output;
    void operator()( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end(); ++i )
            output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
    }
};

// Note: Reads input[0..n] and writes output[1..n-1].
void ParallelAverage( float* output, const float* input, size_t n
) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 1, n ), avg );
}
```

Example

This example is more complex and requires familiarity with STL. It shows the power of parallel_for beyond flat iteration spaces. The code performs a parallel merge of two sorted sequences. It works for any sequence with a random-access iterator. The algorithm (Akl 1987) works recursively as follows:

1. If the sequences are too short for effective use of parallelism, do a sequential merge. Otherwise perform steps 2-6.
2. Swap the sequences if necessary, so that the first sequence [begin1,end1) is at least as long as the second sequence [begin2,end2).
3. Set m1 to the middle position in [begin1,end1). Call the item at that location *key.*
4. Set m2 to where *key* would fall in [begin2,end2).
5. Merge [begin1,m1) and [begin2,m2) to create the first part of the merged sequence.
6. Merge [m1,end1) and [m2,end2) to create the second part of the merged sequence.

The Intel® Threading Building Blocks implementation of this algorithm uses the range object to perform most of the steps. Predicate is_divisible performs the test in step 1, and step 2. The splitting constructor does steps 3-6. The body object does the sequential merges.

```
#include "tbb/parallel_for.h"
#include <algorithm>

using namespace tbb;

template<typename Iterator>
struct ParallelMergeRange {
    static size_t grainsize;
    Iterator begin1, end1; // [begin1,end1) is 1st sequence to be
merged
    Iterator begin2, end2; // [begin2,end2) is 2nd sequence to be
merged
    Iterator out;               // where to put merged sequence
    bool empty()   const {return (end1-begin1)+(end2-begin2)==0;}
    bool is_divisible() const {
        return std::min( end1-begin1, end2-begin2 ) > grainsize;
    }
    ParallelMergeRange( ParallelMergeRange& r, split ) {
        if( r.end1-r.begin1 < r.end2-r.begin2 ) {
            std::swap(r.begin1,r.begin2);
            std::swap(r.end1,r.end2);
        }
        Iterator m1 = r.begin1 + (r.end1-r.begin1)/2;
        Iterator m2 = std::lower_bound( r.begin2, r.end2, *m1 );
        begin1 = m1;
```

```
        begin2 = m2;
        end1 = r.end1;
        end2 = r.end2;
        out = r.out + (m1-r.begin1) + (m2-r.begin2);
        r.end1 = m1;
        r.end2 = m2;
    }
    ParallelMergeRange( Iterator begin1_, Iterator end1_,
                        Iterator begin2_, Iterator end2_,
                        Iterator out_ ) :
        begin1(begin1_), end1(end1_),
        begin2(begin2_), end2(end2_), out(out_)
    {}
};

template<typename Iterator>
size_t ParallelMergeRange<Iterator>::grainsize = 1000;

template<typename Iterator>
struct ParallelMergeBody {
    void operator()( ParallelMergeRange<Iterator>& r ) const {
        std::merge( r.begin1, r.end1, r.begin2, r.end2, r.out );
    }
};

template<typename Iterator>
void ParallelMerge( Iterator begin1, Iterator end1, Iterator
begin2, Iterator end2, Iterator out ) {
    parallel_for(
       ParallelMergeRange<Iterator>(begin1,end1,begin2,end2,out),
       ParallelMergeBody<Iterator>(),
       simple_partitioner()
    );
}
```

Because the algorithm moves many locations, it tends to be bandwidth limited. Speedup varies, depending upon the system.

3.5 parallel_reduce Template Function

Summary

Computes reduction over a range.

Syntax

```
template<typename Range, typename Value,
         typename Func, typename Reduction>
Value parallel_reduce( const Range& range, const Value& identity,
   const Func& func, const Reduction& reduction,
   [, partitioner] );

template<typename Range, typename Body>
void parallel_reduce( const Range& range, const Body& body
   [, partitioner] );
```

where the optional *partitioner* declares any of the partitioners as shown in column 1 of Table 9.

Header

```
#include "tbb/parallel_reduce.h"
```

Description

The `parallel_reduce` template has two forms. The functional form is designed to be easy to use in conjunction with lambda expressions. The imperative form is designed to minimize copying of data.

The functional form `parallel_reduce(`*range*,*identity*,*func*,*reduction*`)` performs a parallel reduction by applying *func* to subranges in *range* and reducing the results using binary operator *reduction.* It returns the result of the reduction. Parameter *func* and *reduction* can be lambda expressions. Table 11 summarizes the type requirements on the types of *identity*, *func*, and *reduction.*

Table 11: Requirements for Func and Reduction

Pseudo-Signature	Semantics
`Value Identity;`	Right identity element for `Func::operator()`.
`Value Func::operator()(const Range& range, const Value& x)`	Accumulate result for subrange, starting with initial value `x`.
`Value Reduction::operator()(const Value& x, const Value& y);`	Combine results `x` and `y`.

The imperative form `parallel_reduce(`*range*,*body*`)` performs parallel reduction of *body* over each value in *range*. Type Range must model the Range concept (3.2). The body must model the requirements in Table 12.

Table 12: Requirements for parallel_reduce Body

Pseudo-Signature	Semantics
`Body::Body( Body&, split );`	Splitting constructor (3.1). Must be able to run concurrently with `operator()` and method `join`.
`Body::~Body()`	Destructor.
`void Body::operator()(const Range& range);`	Accumulate result for subrange.
`void Body::join( Body& rhs );`	Join results. The result in rhs should be merged into the result of `this`.

A `parallel_reduce` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange. A `parallel_reduce` uses the splitting constructor to make one or more copies of the body for each thread. It may copy a body while the body's `operator()` or method `join` runs concurrently. You are responsible for ensuring the safety of such concurrency. In typical usage, the safety requires no extra effort.

When worker threads are available (10.2.1), `parallel_reduce` invokes the splitting constructor for the body. For each such split of the body, it invokes method `join` in order to merge the results from the bodies. Define `join` to update this to represent the accumulated result for this and rhs. The reduction operation should be associative, but does not have to be commutative. For a noncommutative operation *op*, "*left*.join(*right*)" should update *left* to be the result of *left op right*.

A body is split only if the range is split, but the converse is not necessarily so. Figure 1 diagrams a sample execution of `parallel_reduce`. The root represents the original body b0 being applied to the half-open interval [0,20). The range is recursively split at each level into two subranges. The grain size for the example is 5, which yields four leaf ranges. The slash marks (/) denote where copies (b_1 and b_2) of the body were created by the body splitting constructor. Bodies b_0 and b_1 each evaluate one leaf. Body b_2 evaluates leaf [10,15) and [15,20), in that order. On the way back up the tree, `parallel_reduce` invokes b_0.join(b_1) and b_0.join(b_2) to merge the results of the leaves.

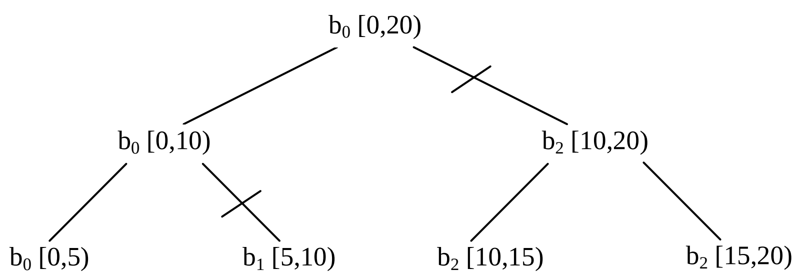

Figure 1: Example Execution of parallel_reduce Over blocked_range<int>(0,20,5)

Figure 1 shows only one possible execution. Other valid executions include splitting b_2 into b_2 and b_3, or doing no splitting at all. With no splitting, b_0 evaluates each leaf in left to right order, with no calls to join. A given body always evaluates one or more consecutive subranges in left to right order. For example, in Figure 1, body b_2 is guaranteed to evaluate [10,15) before [15,20). You may rely on the consecutive left to right property for a given instance of a body, but must not rely on a particular choice of body splitting. parallel_reduce makes the choice of body splitting nondeterministically.

When no worker threads are available, parallel_reduce executes sequentially from left to right in the same sense as for parallel_for (3.4). Sequential execution never invokes the splitting constructor or method join.

Complexity

If the range and body take O(1) space, and the range splits into nearly equal pieces, then the space complexity is O(P log(N)), where N is the size of the range and P is the number of threads.

Example (Imperative Form)

The following code sums the values in an array.

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

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

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n ),
                     total );
    return total.value;
}
```

The example generalizes to reduction for any associative operation *op* as follows:

- Replace occurrences of 0 with the identity element for *op*
- Replace occurrences of += with *op*= or its logical equivalent.
- Change the name `Sum` to something more appropriate for *op*.

The operation may be noncommutative. For example, *op* could be matrix multiplication.

Example with Lambda Expressions

The following is analogous to the previous example, but written using lambda expressions and the functional form of `parallel_reduce`.

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

float ParallelSum( float array[], size_t n ) {
    return parallel_reduce(
        blocked_range<float*>( array, array+n ),
        0.f,
        [](const blocked_range<float*>& r, float init)->float {
            for( float* a=r.begin(); a!=r.end(); ++a )
                init += *a;
            return init;
        },
        []( float x, float y )->float {
            return x+y;
        }
    );
}
```

STL generalized numeric operations and functions objects can be used to write the example more compactly as follows:

```
#include <numeric>
#include <functional>
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

float ParallelSum( float array[], size_t n ) {
    return parallel_reduce(
        blocked_range<float*>( array, array+n ),
        0.f,
        [](const blocked_range<float*>& r, float value)->float {
            return std::accumulate(r.begin(),r.end(),value);
```

```
        },
        std::plus<float>()
    );
}
```

3.6 parallel_scan Template Function

Summary

Template function that computes parallel prefix.

Syntax

```
template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body );

template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body, const
auto_partitioner& );

template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body, const
simple_partitioner& );
```

Header

```
#include "tbb/parallel_scan.h"
```

Description

A `parallel_scan(`*range*`,`*body*`)` computes a parallel prefix, also known as parallel scan. This computation is an advanced concept in parallel computing that is sometimes useful in scenarios that appear to have inherently serial dependences.

A mathematical definition of the parallel prefix is as follows. Let $\oplus$ be an associative operation $\oplus$ with left-identity element $id_{\oplus}$. The parallel prefix of $\oplus$ over a sequence x_0, x_1, ...x_{n-1} is a sequence y_0, y_1, y_2, ...y_{n-1} where:

- $y_0 = id_{\oplus} \oplus x_0$
- $y_i = y_{i-1} \oplus x_i$

For example, if $\oplus$ is addition, the parallel prefix corresponds a running sum. A serial implementation of parallel prefix is:

```
T temp = id⊕;
for( int i=1; i<=n; ++i ) {
    temp = temp ⊕ x[i];
    y[i] = temp;
}
```

Parallel prefix performs this in parallel by reassociating the application of ⊕ and using two passes. It may invoke ⊕ up to twice as many times as the serial prefix algorithm. Given the right grain size and sufficient hardware threads, it can out perform the serial prefix because even though it does more work, it can distribute the work across more than one hardware thread.

TIP: Because parallel_scan needs two passes, systems with only two hardware threads tend to exhibit small speedup. parallel_scan is best considered a glimpse of a technique for future systems with more than two cores. It is nonetheless of interest because it shows how a problem that appears inherently sequential can be parallelized.

The template parallel_scan<Range,Body> implements parallel prefix generically. It requires the signatures described in Table 13.

Table 13: parallel_scan Requirements

Pseudo-Signature	Semantics
void Body::operator()(const Range& r, pre_scan_tag)	Accumulate summary for range r.
void Body::operator()(const Range& r, final_scan_tag)	Compute scan result and summary for range r.
Body::Body(Body& b, split)	Split b so that this and b can accumulate summaries separately. Body *this is object a in the table row below.
void Body::reverse_join(Body& a)	Merge summary accumulated by a into summary accumulated by this, where this was created earlier from a by a's splitting constructor. Body *this is object b in the table row above.
void Body::assign(Body& b)	Assign summary of b to this.

A summary contains enough information such that for two consecutive subranges *r* and *s*:

- If *r* has no preceding subrange, the scan result for *s* can be computed from knowing s and the summary for *r*.
- A summary of r concatenated with s can be computed from the summaries of r and s.

For example, if computing a running sum of an array, the summary for a range *r* is the sum of the array elements corresponding to *r*.

Figure 2 shows one way that parallel_scan might compute the running sum of an array containing the integers 1-16. Time flows downwards in the diagram. Each color denotes a separate Body object. Summaries are shown in brackets.

1. The first two steps split the original blue body into the pink and yellow bodies. Each body operates on a quarter of the input array in parallel. The last quarter is processed later in step 5.
2. The blue body computes the final scan and summary for 1-4. The pink and yellow bodies compute their summaries by prescanning 5-8 and 9-12 respectively.
3. The pink body computes its summary for 1-8 by performing a reverse_join with the blue body.
4. The yellow body computes its summary for 1-12 by performing a reverse_join with the pink body.
5. The blue, pink, and yellow bodies compute final scans and summaries for portions of the array.
6. The yellow summary is assigned to the blue body. The pink and yellow bodies are destroyed.

Note that two quarters of the array were not prescanned. The `parallel_scan` template makes an effort to avoid prescanning where possible, to improve performance when there are only a few or no extra worker threads. If no other workers are available, parallel_scan processes the subranges without any pre_scans, by processing the subranges from left to right using final scans. That's why final scans must compute a summary as well as the final scan result. The summary might be needed to process the next subrange if no worker thread has prescanned it yet.

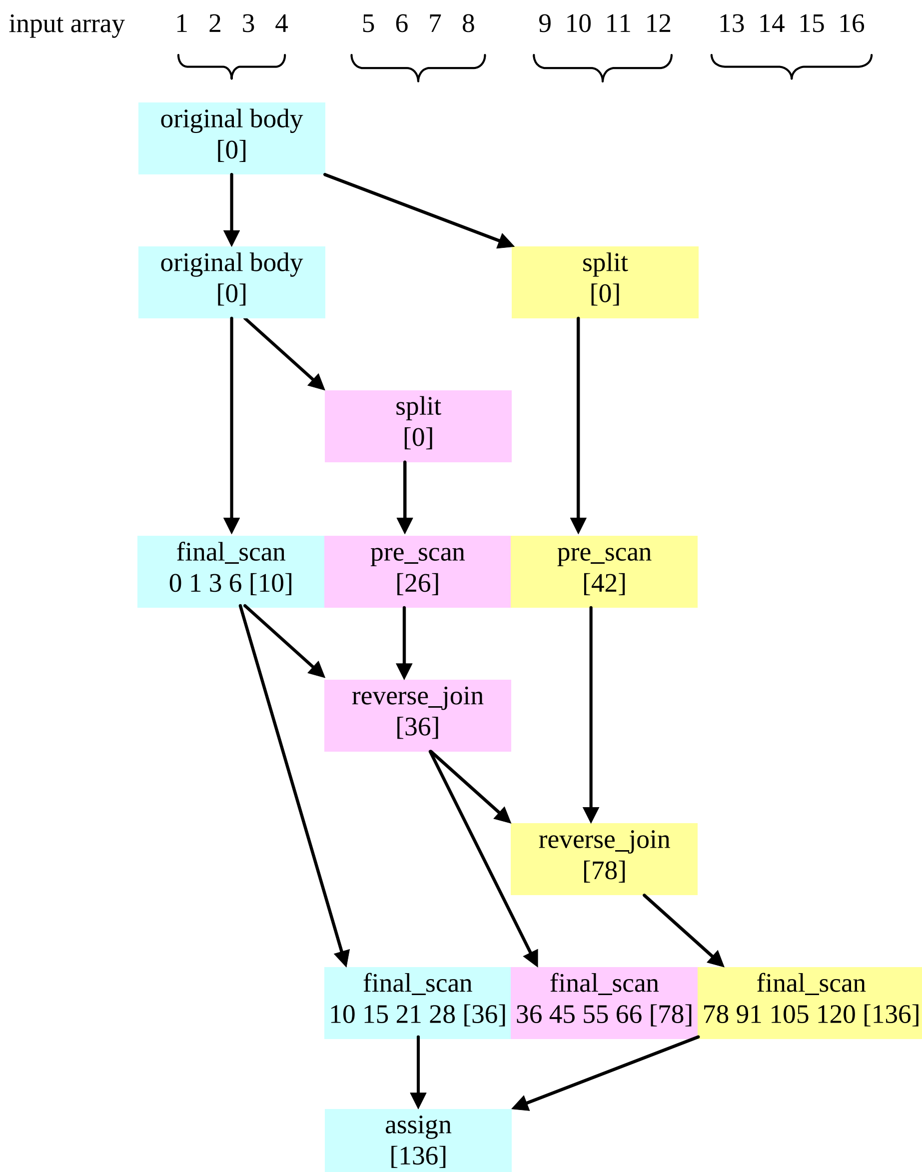

Figure 2: Example Execution of parallel_scan

The following code demonstrates how the signatures could be implemented to use `parallel_scan` to compute the same result as the earlier sequential example involving ⊕.

```
using namespace tbb;

class Body {
    T sum;
    T* const y;
    const T* const x;
```

```
public:
    Body( T y_[], const T x_[] ) : sum(id⊕), x(x_), y(y_) {}
    T get_sum() const {return sum;}

    template<typename Tag>
    void operator()( const blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp ⊕ x[i];
            if( Tag::is_final_scan() )
                y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, split ) : x(b.x), y(b.y), sum(id⊕) {}
    void reverse_join( Body& a ) { sum = a.sum ⊕ sum;}
    void assign( Body& b ) {sum = b.sum;}
};

float DoParallelScan( T y[], const T x[], int n ) {
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n), body );
    return body.get_sum();
}
```

The definition of `operator()` demonstrates typical patterns when using `parallel_scan`.

- A single template defines both versions. Doing so is not required, but usually saves coding effort, because the two versions are usually similar. The library defines static method `is_final_scan()` to enable differentiation between the versions.
- The prescan variant computes the ⊕ reduction, but does not update y. The prescan is used by `parallel_scan` to generate look-ahead partial reductions.
- The final scan variant computes the ⊕ reduction and updates y.

The operation `reverse_join` is similar to the operation `join` used by `parallel_reduce`, except that the arguments are reversed. That is, `this` is the *right* argument of ⊕. Template function `parallel_scan` decides if and when to generate parallel work. It is thus crucial that ⊕ is associative and that the methods of Body faithfully represent it. Operations such as floating-point addition that are somewhat associative can be used, with the understanding that the results may be rounded differently depending upon the association used by `parallel_scan`. The reassociation may differ between runs even on the same machine. However, if there are no worker threads available, execution associates identically to the serial form shown at the beginning of this section.

If you change the example to use a `simple_partitioner`, be sure to provide a grainsize. The code below shows the how to do this for a grainsize of 1000:

```
    parallel_scan(blocked_range<int>(0,n,1000), total,
                                                  simple_partitioner() );
```

3.6.1 pre_scan_tag and final_scan_tag Classes

Summary

Types that distinguish the phases of `parallel_scan`..

Syntax

```
struct pre_scan_tag;
struct final_scan_tag;
```

Header

```
#include "tbb/parallel_scan.h"
```

Description

Types `pre_scan_tag` and `final_scan_tag` are dummy types used in conjunction with `parallel_scan`. See the example in Section 3.6 for how they are used in the signature of `operator()`.

Members

```
namespace tbb {

    struct pre_scan_tag {
        static bool is_final_scan();
    };

    struct final_scan_tag {
        static bool is_final_scan();
    };

}
```

3.6.1.1 bool is_final_scan()

Returns

True for a `final_scan_tag`, otherwise false.

3.7 parallel_do Template Function

Summary

Template function that processes work items in parallel.

Syntax

```
template<typename InputIterator, typename Body>
void parallel_do( InputIterator first, InputIterator last, Body
body );
```

Header

```
#include "tbb/parallel_do.h"
```

Description

A parallel_do(*first, last, body*) applies a function object *body* over the half-open interval [*first*,*last*). Items may be processed in parallel. Additional work items can be added by body if it has a second argument of type parallel_do_feeder (3.7.1). The function terminates when *body*(x) returns for all items *x* that were in the input sequence or added to it by method parallel_do_feeder::add (3.7.1.1).

The requirements for input iterators are specified in Section 24.1 of the ISO C++ standard. Table 14 shows the requirements on type Body.

Table 14: parallel_do Requirements for Body B and its Argument Type T

Pseudo-Signature	Semantics

B::operator()(
    cv-qualifiers T& item,
    parallel_do_feeder<T>& feeder
) const
OR
B::operator()(*cv-qualifiers* T& item) const
	Process item. Template parallel_do may concurrently invoke operator() for the same this but different item.
	The signature with feeder permits additional work items to be added.

T(const T&)	Copy a work item.
~T::T()	Destroy a work item.

For example, a unary function object, as defined in Section 20.3 of the C++ standard, models the requirements for B.

CAUTION: Defining both the one-argument and two-argument forms of operator() is not permitted.

TIP: The parallelism in parallel_do is not scalable if all of the items come from an input stream that does not have random access. To achieve scaling, do one of the following:

- Use random access iterators to specify the input stream.
- Design your algorithm such that the body often adds more than one piece of work.
- Use `parallel_for` instead.

To achieve speedup, the grainsize of `B::operator()` needs to be on the order of at least ~10,000 instructions. Otherwise, the internal overheads of `parallel_do` swamp the useful work.

Example

The following code sketches a body with the two-argument form of `operator()`.

```
struct MyBody {
    void operator()(item_t item,
                    parallel_do_feeder<item_t>& feeder ) {
        for each new piece of work implied by item do {
            item_t new_item = initializer;
            feeder.add(new_item);
        }
    }
};
```

3.7.1 parallel_do_feeder<Item> class

Summary

Inlet into which additional work items for a `parallel_do` can be fed.

Syntax

```
template<typename Item>
class parallel_do_feeder;
```

Header

```
#include "tbb/parallel_do.h"
```

Description

A `parallel_do_feeder` enables the body of a `parallel_do` to add more work items.

Only class `parallel_do` (3.7) can create or destroy a `parallel_do_feeder`. The only operation other code can perform on a `parallel_do_feeder` is to invoke method `parallel_do_feeder::add`.

Members

```
namespace tbb {
    template<typename Item>
    struct parallel_do_feeder {
        void add( const Item& item );
```

```
    };
}
```

3.7.1.1 void add(const Item& item)

Requirements

Must be called from a call to *body*.operator() created by parallel_do. Otherwise, the termination semantics of method operator() are undefined.

Effects

Adds item to collection of work items to be processed.

3.8 parallel_for_each Template Function

Summary

Parallel variant of std::for_each.

Syntax

```
template<typename InputIterator, typename Func>
void parallel_for_each (InputIterator first, InputIterator last,
                        const Func& f);
```

Header

```
#include "tbb/parallel_for_each.h"
```

Description

A parallel_for_each(*first, last*,*f*) applies *f* to the result of dereferencing every iterator in the range [*first*,*last*), possibly in parallel. It is provided for PPL compatibility and equivalent to parallel_do(*first*,*last*,*f*) without "feeder" functionality.

3.9 pipeline Class

Summary

Class that performs pipelined execution.

Syntax

```
class pipeline;
```

Header

```
#include "tbb/pipeline.h"
```

Description

A pipeline represents pipelined application of a series of filters to a stream of items. Each filter operates in a particular mode: parallel, serial in order, or serial out of order (MacDonald 2004). See class filter (3.9.6) for details.

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 its mode as a parameter to the constructor for base class filter (3.9.6.1).
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. An instance of a filter can be added at most once to a pipeline. A filter should never be a member of more than one pipeline at a time.
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. See the Tutorial, in the section on class pipeline, for more about effective use of max_number_of_live_tokens.

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

If there is other work to do while the pipeline is running, the call to method pipeline::run can be replaced by a pair of calls pipeline::start_run and pipeline::finish_run, and the calling thread can do other work between the calls. The example in Section 3.9.7 has an example.

CAUTION: If there are no worker threads, the pipeline does not process any items until the call to pipeline::finish_run.

Members

```
namespace tbb {
    class pipeline {
    public:
        pipeline();
```

```
        ~pipeline();[3]
        void add_filter( filter& f );
        void run( size_t max_number_of_live_tokens );
        void clear();
    };
}
```

3.9.1 pipeline()

Effects

Constructs pipeline with no filters.

3.9.2 ~pipeline()

Effects

Removes all filters from the pipeline and destroys the pipeline

3.9.3 void add_filter(filter& f)

Effects

Appends filter *f* to sequence of filters in the pipeline. The filter *f* must not already be in a pipeline.

3.9.4 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 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.

A pipeline can be run multiple times. It is safe to add stages between runs. Concurrent invocations of `run` on the same instance of pipeline are prohibited.

---

[3] Though the current implementation declares the destructor `virtual`, do not rely on this detail. The virtual nature is deprecated and may disappear in future versions of Intel® TBB.

3.9.5 void clear()

Effects

Removes all filters from the pipeline.

3.9.6 filter Class

Summary

Abstract base class that represents a filter in a pipeline.

Syntax

```
class filter;
```

Header

```
#include "tbb/pipeline.h"
```

Description

A `filter` represents a filter in a `pipeline` (3.9). There are three modes of filters:

A **`parallel`** filter can process multiple items in parallel and in no particular order.

A **`serial_out_of_order`** filter processes items one at a time, and in no particular order.

A **`serial_in_order`** filter processes items one at a time. All `serial_in_order` filters in a pipeline process items in the same order.

The mode of filter is specified by an argument to the constructor. Parallel filters are preferred when practical because they permit parallel speedup. If a filter must be serial, the out of order variant is preferred when practical because it puts less contraints on processing order.

Class `filter` should only be used in conjunction with class `pipeline` (3.9).

TIP: Use a `serial_in_order` input filter if there are any subsequent `serial_in_order` stages that should process items in their input order.

CAUTION: Intel® TBB 2.0 and prior treated parallel input stages as serial. Later versions of Intel® TBB can execute a parallel input stage in parallel, so if you specify such a stage, ensure that its `operator()` is thread safe.

Members

```
namespace tbb {
    class filter {
    public:
        enum mode {
            parallel = implementation-defined,
```

```
            serial_in_order = implementation-defined,
            serial_out_of_order = implementation-defined
        };
        bool is_serial() const;
        bool is_ordered() const;
        virtual void* operator()( void* item ) = 0;
        virtual void finalize( void* item ) {}
        virtual ~filter();
    protected:
        filter( mode );
    };
}
```

Example

See the example filters `MyInputFilter`, `MyTransformFilter`, and `MyOutputFilter` in the Tutorial (`doc/Tutorial.pdf`).

3.9.6.1 filter(mode filter_mode)

Effects

Constructs a filter of the specified mode.

NOTE: Intel® TBB 2.1 and prior had a similar constructor with a `bool` argument `is_serial`. That constructor exists but is deprecated (Section A.2.1).

3.9.6.2 ~filter()

Effects

Destroys the filter. If the filter is in a `pipeline`, it is automatically removed from that pipeline.

3.9.6.3 bool is_serial() const

Returns

False if filter mode is `parallel`; true otherwise.

3.9.6.4 bool is_ordered() const

Returns

True if filter mode is `serial_in_order`, false otherwise.

3.9.6.5 virtual void* operator()(void * item)

Description

The derived filter should override this method to process an item and return a pointer to an item to be processed by the next `filter`. The item parameter is NULL for the first filter in the pipeline.

Returns

The first filter in a `pipeline` should return NULL if there are no more items to process. The result of the last filter in a `pipeline` is ignored.

3.9.6.6 virtual void finalize(void * item)

Description

A pipeline can be cancelled by user demand or because of an exception. When a pipeline is cancelled, there may be items returned by a filter's operator() that have not yet been processed by the next filter. When a pipeline is cancelled, the next filter invokes `finalize()` on each item instead of `operator()`. In contrast to `operator()`, method `finalize()` does not return an item for further processing. A derived filter should override `finalize()` to perform proper cleanup for an item. A pipeline will not invoke any further methods on the item.

Effects

The default definition has no effect.

3.9.7 thread_bound_filter Class

Summary

Abstract base class that represents a filter in a pipeline that a thread must service explicitly.

Syntax

```
class thread_bound_filter;
```

Header

```
#include "tbb/pipeline.h"
```

Description

A `thread_bound_filter` is a special kind of `filter` (3.9.6) that is explicitly serviced by a particular thread. It is useful when a filter must be executed by a particular thread.

CAUTION: Use `thread_bound_filter` only if you need a filter to be executed on a particular thread. The thread that services a `thread_bound_filter` must not be the thread that calls `pipeline::run()`.

Members

```
namespace tbb {
    class thread_bound_filter: public filter {
    protected:
        thread_bound_filter(mode filter_mode);
    public:
        enum result_type {
            success,
            item_not_available,
            end_of_stream
        };
        result_type try_process_item();
        result_type process_item();
    };
}
```

Example

The example below shows a pipeline with two filters where the second filter is a `thread_bound_filter` serviced by the main thread.

```
#include <iostream>
#include "tbb/pipeline.h"
#include "tbb/tbb_thread.h"
#include "tbb/task_scheduler_init.h"

using namespace tbb;

char InputString[] = "abcdefg\n";

class InputFilter: public filter {
    char* my_ptr;
public:
    void* operator()(void*) {
        if (*my_ptr)
            return my_ptr++;
        else
            return NULL;
    }
    InputFilter() :
        filter( serial_in_order ), my_ptr(InputString)
    {}
```

```
};

class OutputFilter: public thread_bound_filter {
public:
    void* operator()(void* item) {
        std::cout << *(char*)item;
        return NULL;
    }
    OutputFilter() : thread_bound_filter(serial_in_order) {}
};

void RunPipeline(pipeline* p) {
    p->run(8);
}

int main() {
    // Construct the pipeline
    InputFilter f;
    OutputFilter g;
    pipeline p;
    p.add_filter(f);
    p.add_filter(g);

    // Another thread initiates execution of the pipeline
    tbb_thread t(RunPipeline,&p);

    // Process the thread_bound_filter with the current thread.
    while (g.process_item()!=thread_bound_filter::end_of_stream)
        continue;

    // Wait for pipeline to finish on the other thread.
    t.join();
    return 0;
}
```

The main thread does the following after constructing the pipeline:

1. Start the pipeline on another thread.
2. Service the `thread_bound_filter` until it reaches `end_of_stream`.
3. Wait for the other thread to finish.

The pipeline is run on a separate thread because the main thread is responsible for servicing the `thread_bound_filter` g. The roles of the two threads can be reversed. A single thread cannot do both roles.

3.9.7.1 thread_bound_filter(mode filter_mode)

Effects

Constructs a filter of the specified mode. Section 3.9.6 describes the modes.

3.9.7.2 result_type try_process_item()

Effects

If an item is available and it can be processed without exceeding the token limit, process the item with `filter::operator()`.

Returns

Table 15: Return Values From `try_process_item`

Return Value	Description
`success`	Applied `filter::operator()` to one item.
`item_not_available`	No item is currently available to process, or the token limit (3.9.4) would be exceeded.
`end_of_stream`	No more items will ever arrive at this filter.

3.9.7.3 result_type process_item()

Effects

Like `try_process_item`, but waits until it can process an item or the end of the stream is reached.

Returns

Either success or `end_of_stream`. See Table 15 for details.

CAUTION: The current implementation spin waits until it can process an item or reaches the end of the stream.

3.10 parallel_sort Template Function

Summary

Sort a sequence.

Syntax

```
template<typename RandomAccessIterator>
```

```
void parallel_sort(RandomAccessIterator begin,
RandomAccessIterator end);

template<typename RandomAccessIterator, typename Compare>
void parallel_sort(RandomAccessIterator begin,
RandomAccessIterator end,
                     const Compare& comp );
```

Header

```
#include "tbb/parallel_sort.h"
```

Description

Performs an *unstable* sort of sequence [*begin1*, *end1*). An unstable sort might not preserve the relative ordering of elements with equal keys. The sort is deterministic; sorting the same sequence will produce the same result each time. The requirements on the iterator and sequence are the same as for `std::sort`. Specifically, `RandomAccessIterator` must be a random access iterator, and its value type *T* must model the requirements in Table 16.

Table 16: Requirements on Value Type T of RandomAccessIterator for parallel_sort

Pseudo-Signature	Semantics
`void swap( T& x, T& y )`	Swap x and y.
`bool Compare::operator()( const T& x, const T& y )`	True if x comes before y; false otherwise.

A call `parallel_sort(i,j,comp)` sorts the sequence `[i,j)` using the second argument `comp` to determine relative orderings. If `comp(x,y)` returns `true` then `x` appears before `y` in the sorted sequence.

A call `parallel_sort(i,j)` is equivalent to `parallel_sort(i,j,std::less<T>)`.

Complexity

`parallel_sort` is comparison sort with an average time complexity of O(N log (N)), where N is the number of elements in the sequence. When worker threads are available (10.2.1), `parallel_sort` creates subtasks that may be executed concurrently, leading to improved execution times.

Example

The following example shows two sorts. The sort of array a uses the default comparison, which sorts in ascending order. The sort of array b sorts in descending order by using `std::greater<float>` for comparison.

```
#include "tbb/parallel_sort.h"
#include <math.h>

using namespace tbb;
```

```
const int N = 100000;
float a[N];
float b[N];

void SortExample() {
    for( int i = 0; i < N; i++ ) {
       a[i] = sin((double)i);
       b[i] = cos((double)i);
    }
    parallel_sort(a, a + N);
    parallel_sort(b, b + N, std::greater<float>());
}
```

3.11 parallel_invoke Template Function

Summary

Template function that evaluates several functions in parallel.

Syntax[4]

```
template<typename Func0, typename Func1>
void parallel_invoke(const Func0& f0, const Func1& f1);

template<typename Func0, typename Func1, typename Func2>
void parallel_invoke(const Func0& f0, const Func1& f1, const
Func2& f2);
…
template<typename Func0, typename Func1 … typename Func9>
void parallel_invoke(const Func0& f0, const Func1& f1 … const
Func9& f9);
```

Header

```
#include "tbb/parallel_invoke.h"
```

Description

The expression `parallel_invoke(`f_0`,`f_1…f_k`)` evaluates f_0`()`, f_1`()`,…f_k possibly in parallel. There can be from 2 to 10 arguments. Each argument must have a type for

[4] When support for C++0x rvalue references become prevalent, the formal parameters may change to rvalue references instead of values.

which operator() is defined. Typically the arguments are either function objects or pointers to functions. Return values are ignored.

Example

The following example evaluates f(), g(), and h() in parallel. Notice how g and h are function objects that can hold local state.

```
#include "tbb/parallel_invoke.h"

using namespace tbb;

void f();
extern void bar(int);

class MyFunctor {
    int arg;
public:
    MyFunctor(int a) : arg(a) {}
    void operator()() const {bar(arg);}
};

void RunFunctionsInParallel() {
    MyFunctor g(2);
    MyFunctor h(3);
    tbb::parallel_invoke(f, g, h );
}
```

Example with Lambda Expressions

Here is the previous example rewritten with C++0x lambda expressions, which generate function objects.

```
#include "tbb/parallel_invoke.h"

using namespace tbb;

void f();
extern void bar(int);

void RunFunctionsInParallel() {
    tbb::parallel_invoke(f, []{bar(2);}, []{bar(3);} );
}
```

4 Containers

The container classes permit multiple threads to simultaneously invoke certain methods on the same container.

Like STL, Intel® Threading Building Blocks (Intel® TBB) containers are templated with respect to an allocator argument. Each container uses its allocator to allocate memory for user-visible items. A container may use a different allocator for strictly internal structures.

4.1 Container Range Concept

Summary

View set of items in a container as a recursively divisible range.

Requirements

A Container Range is a Range (3.2) with the further requirements listed in Table 17.

Table 17: Requirements on a Container Range R (In Addition to Table 7)

Pseudo-Signature	Semantics
R::value_type	Item type
R::reference	Item reference type
R::const_reference	Item const reference type
R::difference_type	Type for difference of two iterators
R::iterator	Iterator type for range
R::iterator R::begin()	First item in range
R::iterator R::end()	One past last item in range
R::size_type R::grainsize() const	Grain size

Model Types

Classes concurrent_hash_map (4.2.4) and concurrent_vector (4.5.5) both have member types range_type and const_range_type that model a Container Range.

Use the range types in conjunction with parallel_for (3.4), parallel_reduce (3.5), and parallel_scan (3.6) to iterate over items in a container.

4.2 concurrent_hash_map Template Class

Summary

Template class for associative container with concurrent access.

Syntax

```
template<typename Key, typename T,
         typename HashCompare=tbb_hash_compare<Key>,
         typename A=tbb_allocator<std::pair<Key, T> > >
class concurrent_hash_map;
```

Header

```
#include "tbb/concurrent_hash_map.h"
```

Description

A concurrent_hash_map maps keys to values in a way that permits multiple threads to concurrently access values. The keys are unordered. There is at most one element in a concurrent_hash_map for each key. The key may have other elements in flight but not in the map as described in Section 4.2.3. The interface resembles typical STL associative containers, but with some differences critical to supporting concurrent access. It meets the Container Requirements of the ISO C++ standard.

Types Key and T must model the CopyConstructible concept (2.2.3).

Type HashCompare specifies how keys are hashed and compared for equality. It must model the HashCompare concept in Table 18.

Table 18: HashCompare Concept

Pseudo-Signature	Semantics
`HashCompare::HashCompare( const HashCompare& )`	Copy constructor.
`HashCompare::~HashCompare ()`	Destructor.
`bool HashCompare::equal( const Key& j, const Key& k ) const`	True if keys are equal.
`size_t HashCompare::hash( const Key& k ) const`	Hashcode for key.

CAUTION: As for most hash tables, if two keys are equal, they must hash to the same hash code. That is for a given HashCompare h and any two keys j and k, the following assertion must hold: "!h.equal(j,k) || h.hash(j)==h.hash(k)". The importance of this property is the reason that concurrent_hash_map makes key equality and hashing function travel together in a single object instead of being separate objects. The hash code of a key must not change while the hash table is non-empty.

CAUTION: Good performance depends on having good pseudo-randomness in the low-order bits of the hash code.

Example

When keys are pointers, simply casting the pointer to a hash code may cause poor performance because the low-order bits of the hash code will be always zero if the pointer points to a type with alignment restrictions. A way to remove this bias is to divide the casted pointer by the size of the type, as shown by the underlined blue text below.

```
size_t MyHashCompare::hash( Key* key ) const {
    return reinterpret_cast<size_t>(key)/sizeof(Key);
}
```

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare,
             typename Alloc=tbb_allocator<std::pair<Key,T> > >
    class concurrent_hash_map {
    public:
        // types
        typedef Key key_type;
        typedef T mapped_type;
        typedef std::pair<const Key,T> value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        typedef value_type* pointer;
        typedef const value_type* const_pointer;
        typedef value_type& reference;
        typedef Alloc allocator_type;

        // whole-table operations
        concurrent_hash_map(
            const allocator_type& a=allocator_type() );
        concurrent_hash_map(
            size_type n,
            const allocator_type &a = allocator_type() );
        concurrent_hash_map(
            const concurrent_hash_map&,
            const allocator_type& a=allocator_type() );
        template<typename InputIterator>
            concurrent_hash_map(
                InputIterator first, InputIterator last,
                const allocator_type& a = allocator_type())
        ~concurrent_hash_map();
        concurrent_hash_map operator=( const concurrent_hash_map&
);
        void clear();
```

```
        allocator_type get_allocator() const;

        // concurrent access
        class const_accessor;
        class accessor;

        // concurrent operations on a table
        bool find( const_accessor& result, const Key& key )
const;
        bool find( accessor& result, const Key& key );
        bool insert( const_accessor& result, const Key& key );
        bool insert( accessor& result, const Key& key );
        bool insert( const_accessor& result, const value_type&
value );
        bool insert( accessor& result, const value_type& value );
        bool insert( const value_type& value );
        template<typename I> void insert( I first, I last );
        bool erase( const Key& key );
        bool erase( const_accessor& item_accessor );
        bool erase( accessor& item_accessor );

        // parallel iteration
        typedef implementation defined range_type;
        typedef implementation defined const_range_type;
        range_type range( size_t grainsize=1 );
        const_range_type range( size_t grainsize=1 ) const;

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

        // iterators
        typedef implementation defined iterator;
        typedef implementation defined const_iterator;
        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
        std::pair<iterator, iterator> equal_range( const Key& key
);
        std::pair<const_iterator, const_iterator>
            equal_range( const Key& key ) const;
    };
```

```
    template<typename Key, typename T, typename HashCompare,
             typename A1, typename A2>
    bool operator==(
        const concurrent_hash_map<Key,T,HashCompare,A1> &a,
        const concurrent_hash_map<Key,T,HashCompare,A2> &b);

    template<typename Key, typename T, typename HashCompare,
             typename A1, typename A2>
    bool operator!=(const
        concurrent_hash_map<Key,T,HashCompare,A1> &a,
        const concurrent_hash_map<Key,T,HashCompare,A2> &b);

    template<typename Key, typename T, typename HashCompare,
             typename A>
    void swap(concurrent_hash_map<Key,T,HashCompare,A>& a,
              concurrent_hash_map<Key,T,HashCompare,A>& b)
}
```

Exception Safey

The following functions must not throw exceptions:

- The hash function
- The destructors for types Key and T.

The following hold true:

- If an exception happens during an insert operation, the operation has no effect.
- If an exception happens during an assignment operation, the container may be in a state where only some of the items were assigned, and methods size() and empty() may return invalid answers.

4.2.1 Whole Table Operations

These operations affect an entire table. Do not concurrently invoke them on the same table.

4.2.1.1 concurrent_hash_map(const allocator_type& a = allocator_type())

Effects

Constructs empty table.

4.2.1.2 concurrent_hash_map(size_type n, const allocator_type& a = allocator_type())

Effects

Construct empty table with preallocated buckets for at least *n* items.

NOTE: In general, thread contention for buckets is inversely related to the number of buckets. If memory consumption is not an issue and *P* threads will be accessing the concurrent_hash_map, set $n \geq 4P$.

4.2.1.3 concurrent_hash_map(const concurrent_hash_map& table, const allocator_type& a = allocator_type())

Effects

Copies a table. The table being copied may have const operations running on it concurrently.

4.2.1.4 template<typename InputIterator> concurrent_hash_map(InputIterator first, InputIterator last, const allocator_type& a = allocator_type())

Effects

Constructs table containing copies of elements in the iterator half-open interval [*first*,*last*).

4.2.1.5 ~concurrent_hash_map()

Effects

Invokes clear(). This method is not safe to execute concurrently with other methods on the same concurrent_hash_map.

4.2.1.6 concurrent_hash_map& operator= (concurrent_hash_map& source)

Effects

If source and destination (this) table are distinct, clears the destination table and copies all key-value pairs from the source table to the destination table. Otherwise, does nothing.

Returns

Reference to the destination table.

4.2.1.7 void swap(concurrent_hash_map& table)

Effects

Swaps contents and allocators of `this` and `table`.

4.2.1.8 void clear()

Effects

Erases all key-value pairs from the table. Does not hash or compare any keys.

If TBB_USE_PERFORMANCE_WARNINGS is nonzero, issues a performance warning if the randomness of the hashing is poor enough to significantly impact performance.

4.2.1.9 allocator_type get_allocator() const

Returns

Copy of allocator used to construct table.

4.2.2 Concurrent Access

Member classes `const_accessor` and `accessor` are called *accessors.* Accessors allow multiple threads to concurrently access pairs in a shared `concurrent_hash_map`. An accessor acts as a smart pointer to a pair in a `concurrent_hash_map`. It holds an implicit lock on a pair until the instance is destroyed or method `release` is called on the accessor.

Classes `const_accessor` and `accessor` differ in the kind of access that they permit.

Table 19: Differences Between const_accessor and accessor

Class	value_type	Implied Lock on pair
`const_accessor`	`const std::pair<const Key,T>`	Reader lock – permits shared access with other readers.
`accessor`	`std::pair<const Key,T>`	Writer lock – permits exclusive access by a thread. Blocks access by other threads.

Accessors cannot be assigned or copy-constructed, because allowing such would greatly complicate the locking semantics.

4.2.2.1 const_accessor

Summary

Provides read-only access to a pair in a `concurrent_hash_map`.

Syntax

```
template<typename Key, typename T, typename HashCompare, typename
A>
class concurrent_hash_map<Key,T,HashCompare,A>::const_accessor;
```

Header

```
#include "tbb/concurrent_hash_map.h"
```

Description

A `const_accessor` permits read-only access to a key-value pair in a `concurrent_hash_map`.

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare,
typename A>
    class
concurrent_hash_map<Key,T,HashCompare,A>::const_accessor {
    public:
        // types
        typedef const std::pair<const Key,T> value_type;

        // construction and destruction
        const_accessor();
        ~const_accessor();

        // inspection
        bool empty() const;
        const value_type& operator*() const;
        const value_type* operator->() const;

        // early release
        void release();
    };
}
```

4.2.2.1.1 bool empty() const

Returns

True if instance points to nothing; false if instance points to a key-value pair.

4.2.2.1.2 void release()

Effects

If `!empty()`, releases the implied lock on the pair, and sets instance to point to nothing. Otherwise does nothing.

4.2.2.1.3 const value_type& operator*() const

Effects

Raises assertion failure if `empty()` and TBB_USE_ASSERT (2.6.1) is defined as nonzero.

Returns

Const reference to key-value pair.

4.2.2.1.4 const value_type* operator->() const

Returns

```
&operator*()
```

4.2.2.1.5 const_accessor()

Effects

Constructs const_accessor that points to nothing.

4.2.2.1.6 ~const_accessor

Effects

If pointing to key-value pair, releases the implied lock on the pair.

4.2.2.2 accessor

Summary

Class that provides read and write access to a pair in a `concurrent_hash_map`.

Syntax

```
template<typename Key, typename T, typename HashCompare,
         typename Alloc>
class concurrent_hash_map<Key,T,HashCompare,A>::accessor;
```

Header

```
#include "tbb/concurrent_hash_map.h"
```

Description

An accessor permits read and write access to a key-value pair in a concurrent_hash_map. It is derived from a const_accessor, and thus can be implicitly cast to a const_accessor.

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare,
typename Alloc>
    class concurrent_hash_map<Key,T,HashCompare,Alloc>::accessor:

concurrent_hash_map<Key,T,HashCompare,Alloc>::const_accessor {
    public:
        typedef std::pair<const Key,T> value_type;
        value_type& operator*() const;
        value_type* operator->() const;
    };
}
```

4.2.2.2.1 value_type& operator*() const

Effects

Raises assertion failure if empty() and TBB_USE_ASSERT (2.6.1) is defined as nonzero.

Returns

Reference to key-value pair.

4.2.2.2.2 value_type* operator->() const

Returns

```
&operator*()
```

4.2.3 Concurrent Operations

The operations count, find, insert, and erase are the only operations that may be concurrently invoked on the same concurrent_hash_map. These operations search the table for a key-value pair that matches a given key. The find and insert methods each have two variants. One takes a const_accessor argument and provides read-only access to the desired key-value pair. The other takes an accessor argument and provides write access. Additionally, insert has a variant without any accessor.

CAUTION: The concurrent operations (count, find, insert, and erase) invalidate any iterators pointing into the affected instance. It is unsafe to use these operations concurrently with any other operation.

TIP: If the nonconst variant succeeds in finding the key, the consequent write access blocks any other thread from accessing the key until the accessor object is destroyed. Where possible, use the const variant to improve concurrency.

Each map operation in this section returns true if the operation succeeds, false otherwise.

CAUTION: Though there can be at most one occurrence of a given key in the map, there may be other key-value pairs in flight with the same key. These arise from the semantics of the `insert` and `erase` methods. The `insert` methods can create and destroy a temporary key-value pair that is not inserted into a map. The erase methods remove a key-value pair from the map before destroying it, thus permitting another thread to construct a similar key before the old one is destroyed.

TIP: To guarantee that only one instance of a resource exists simultaneously for a given key, use the following technique:

- To construct the resource: Obtain an `accessor` to the key in the map before constructing the resource.
- To destroy the resource: Obtain an `accessor` to the key, destroy the resource, and then erase the key using the accessor.

Below is a sketch of how this can be done.

```
extern tbb::concurrent_hash_map<Key,Resource,HashCompare> Map;

void ConstructResource( Key key ) {
    accessor acc;
    if( Map.insert(acc,key) ) {
        // Current thread inserted key and has exclusive access.
        ...construct the resource here...
    }
    // Implicit destruction of acc releases lock
}

void DestroyResource( Key key ) {
    accessor acc;
    if( Map.find(acc,key) ) {
        // Current thread found key and has exclusive access.
        ...destroy the resource here...
        // Erase key using accessor.
        Map.erase(acc);
    }
}
```

4.2.3.1 size_type count(const Key& key) const

Returns

1 if map contains key; 0 otherwise.

4.2.3.2 bool find(const_accessor& result, const Key& key) const

Effects

Searches table for pair with given key. If key is found, sets result to provide read-only access to the matching pair.

Returns

True if key was found; false if key was not found.

4.2.3.3 bool find(accessor& result, const Key& key)

Effects

Searches table for pair with given key. If key is found, sets result to provide write access to the matching pair

Returns

True if key was found; false if key was not found.

4.2.3.4 bool insert(const_accessor& result, const Key& key)

Effects

Searches table for pair with given key. If not present, inserts new `pair(key,T())` into the table. Sets *`result`* to provide read-only access to the matching pair.

Returns

True if new pair was inserted; false if key was already in the map.

4.2.3.5 bool insert(accessor& result, const Key& key)

Effects

Searches table for pair with given key. If not present, inserts new `pair(key,T())` into the table. Sets *`result`* to provide write access to the matching pair.

Returns

True if new pair was inserted; false if key was already in the map.

4.2.3.6 bool insert(const_accessor& result, const value_type& value)

Effects

Searches table for pair with given key. If not present, inserts new pair copy-constructed from *value* into the table. Sets `result` to provide read-only access to the matching pair.

Returns

True if new pair was inserted; false if key was already in the map.

4.2.3.7 bool insert(accessor& result, const value_type& value)

Effects

Searches table for pair with given key. If not present, inserts new pair copy-constructed from *value* into the table. Sets `result` to provide write access to the matching pair.

Returns

True if new pair was inserted; false if key was already in the map.

4.2.3.8 bool insert(const value_type& value)

Effects

Searches table for pair with given key. If not present, inserts new pair copy-constructed from *value* into the table.

Returns

True if new pair was inserted; false if key was already in the map.

4.2.3.9 template<typename InputIterator> void insert(InputIterator first, InputIterator last)

Effects

For each pair *p* in the half-open interval [`first`,`last`), does `insert(p)`. The order of the insertions, or whether they are done concurrently, is unspecified.

CAUTION: The current implementation processes the insertions in order. Future implementations may do the insertions concurrently. If duplicate keys exist in [first,last), be careful to not depend on their insertion order.

4.2.3.10 bool erase(const Key& key)

Effects

Searches table for pair with given key. Removes the matching pair if it exists. If there is an accessor pointing to the pair, the pair is nonetheless removed from the table but its destruction is deferred until all accessors stop pointing to it.

Returns

True if pair was removed by the call; false if key was not found in the map.

4.2.3.11 bool erase(const_accessor& item_accessor)

Requirements

item_accessor.empty()==false

Effects

Removes pair referenced by *item_accessor*. Concurrent insertion of the same key creates a new pair in the table.

Returns

True if pair was removed by this thread; false if pair was removed by another thread.

4.2.3.12 bool erase(accessor& item_accessor)

Requirements

item_accessor.empty()==false

Effects

Removes pair referenced by *item_accessor*. Concurrent insertion of the same key creates a new pair in the table.

Returns

True if pair was removed by this thread; false if pair was removed by another thread.

4.2.4 Parallel Iteration

Types const_range_type and range_type model the Container Range concept (4.1). The types differ only in that the bounds for a const_range_type are of type const_iterator, whereas the bounds for a range_type are of type iterator.

4.2.4.1 const_range_type range(size_t grainsize=1) const

Effects

Constructs a `const_range_type` representing all keys in the table. The parameter `grainsize` is in units of hash table buckets. Each bucket typically has on average about one key-value pair.

Returns

`const_range_type` object for the table.

4.2.4.2 range_type range(size_t grainsize=1)

Returns

`range_type` object for the table.

4.2.5 Capacity

4.2.5.1 size_type size() const

Returns

Number of key-value pairs in the table.

NOTE: This method takes constant time, but is slower than for most STL containers.

4.2.5.2 bool empty() const

Returns

```
size()==0.
```

NOTE: This method takes constant time, but is slower than for most STL containers.

4.2.5.3 size_type max_size() const

Returns

Inclusive upper bound on number of key-value pairs that the table can hold.

4.2.6 Iterators

Template class `concurrent_hash_map` supports forward iterators; that is, iterators that can advance only forwards across a table. Reverse iterators are not supported. Concurrent operations (`count`, `find`, `insert`, and `erase`) invalidate any existing iterators that point into the table.

4.2.6.1 iterator begin()

Returns

`iterator` pointing to beginning of key-value sequence.

4.2.6.2 iterator end()

Returns

`iterator` pointing to end of key-value sequence.

4.2.6.3 const_iterator begin() const

Returns

`const_iterator` with pointing to beginning of key-value sequence.

4.2.6.4 const_iterator end() const

Returns

`const_iterator` pointing to end of key-value sequence.

4.2.6.5 std::pair<iterator, iterator> equal_range(const Key& key);

Returns

Pair of iterators (*i*,*j*) such that the half-open range [*i*,*j*) contains all pairs in the map (and only such pairs) with keys equal to key. Because the map has no duplicate keys, the half-open range is either empty or contains a single pair.

4.2.6.6 std::pair<const_iterator, const_iterator> equal_range(const Key& key) const;

Description

See 4.2.6.5.

4.2.7 Global Functions

These functions in namespace `tbb` improve the STL compatibility of `concurrent_hash_map`.

4.2.7.1 template<typename Key, typename T, typename HashCompare, typename A1, typename A2> bool operator==(const concurrent_hash_map<Key,T,HashCompare,A1>& a, const concurrent_hash_map<Key,T,HashCompare,A2>& b);

Returns

True if a and b contain equal sets of keys and for each pair $(k,v_1) \in a$ and pair $,v_2) \in b$, the expression `bool(`v_1`==`v_2`)` is true.

4.2.7.2 template<typename Key, typename T, typename HashCompare, typename A1, typename A2> bool operator!=(const concurrent_hash_map<Key,T,HashCompare,A1> &a, const concurrent_hash_map<Key,T,HashCompare,A2> &b);

Returns

`!(`*a*`==`*b*`)`

4.2.7.3 template<typename Key, typename T, typename HashCompare, typename A> void swap(concurrent_hash_map<Key, T, HashCompare, A> &a, concurrent_hash_map<Key, T, HashCompare, A> &b)

Effects

`a.swap(`*b*`)`

4.2.8 tbb_hash_compare Class

Summary

Default `HashCompare` for `concurrent_hash_map`.

Syntax

```
template<typename Key> struct tbb_hash_compare;
```

Header

```
#include "tbb/concurrent_hash_map.h"
```

Description

A `tbb_hash_compare<Key>` is the default for the `HashCompare` argument of template class `concurrent_hash_map`. The built-in definition relies on `operator==` and `tbb_hasher` as shown in the Members description. For your own types, you can define a template specialization of `tbb_hash_compare` or define an overload of `tbb_hasher`.

There are built-in definitions of tbb_hasher for the following Key types:

- Types that are convertible to a size_t by static_cast<T>
- Pointer types
- std::basic_string
- std::pair<*K1*,*K2*> where *K1* and *K2* are hashed using tbb_hasher.

You may add overloads of tbb_hash_compare for your own types.

Members

```
namespace tbb {
    template<typename Key>
    struct tbb_hash_compare {
        static size_t hash(const Key& a) {
            return tbb_hasher(a);
        }
        static bool equal(const Key& a, const Key& b) {
            return a==b;
        }
    };

    template<typename T>
    size_t tbb_hasher(const T&);

    template<typename T>
    size_t tbb_hasher(T*);

    template<typename T, typename Traits, typename Alloc>
    size_t tbb_hasher(const std::basic_string<T, Traits,Alloc>&);

    template<typename T1, typename T2>
    size_t tbb_hasher(const std::pair<T1,T2>& );
};
```

4.3 concurrent_queue Template Class

Summary

Template class for queue with concurrent operations.

Syntax

```
template<typename T, typename Alloc=cache_aligned_allocator<T> >
class concurrent_queue;
```

Header

```
#include "tbb/concurrent_queue.h"
```

Description

A concurrent_queue is a first-in first-out data structure that permits multiple threads to concurrently push and pop items. Its capacity is unbounded[5], subject to memory limitations on the target machine.

The interface is similar to STL std::queue except where it must differ to make concurrent modification safe.

Table 20: Differences Between STL queue and Intel® Threading Building Blocks concurrent_queue

Feature	STL std::queue	concurrent_queue
Access to front and back	Methods front and back	Not present. They would be unsafe while concurrent operations are in progress.
size_type	unsigned integral type	*signed* integral type
unsafe_size()	Returns number of items in queue	Returns number of items in queue. May return incorrect value if any push or try_pop operations are concurrently in flight.

Copy and pop item unless queue q is empty.	`bool b=!q.empty();`
	`if(b) {`
	`    x=q.front();`
	`    q.pop();`
	`}`
		bool b = *q*.try_pop (x)

Members

```
namespace tbb {
    template<typename T,
             typename Alloc=cache_aligned_allocator<T> >
    class concurrent_queue {
    public:
        // types
```

---

[5] In Intel® TBB 2.1, a concurrent_queue could be bounded. Intel® TBB 2.2 moves this functionality to concurrent_bounded_queue. Compile with TBB_DEPRECATED=1 to restore the old functionality, or (recommended) use concurrent_bounded_queue instead.

```
        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef std::ptrdiff_t size_type;
        typedef std::ptrdiff_t difference_type;
        typedef Alloc allocator_type;

        explicit concurrent_queue(const Alloc& a = Alloc ());
        concurrent_queue(const concurrent_queue& src,
                         const Alloc& a = Alloc());
        template<typename InputIterator>
        concurrent_queue(InputIterator first, InputIterator last,
                         const Alloc& a = Alloc());
        ~concurrent_queue();

        void push( const T& source );
        bool try_pop[6]( T& destination );
        void clear() ;

        size_type unsafe_size() const;
        bool empty() const;
        Alloc get_allocator() const;

        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;

        // iterators (these are slow and intended only for debugging)
        iterator unsafe_begin();
        iterator unsafe_end();
        const_iterator unsafe_begin() const;
        const_iterator unsafe_end() const;
    };
}
```

4.3.1 concurrent_queue(const Alloc& a = Alloc ())

Effects

Constructs empty queue.

---

[6] Called pop_if_present in Intel® TBB 2.1. Compile with TBB_DEPRECATED=1 to use the old name.

4.3.2 concurrent_queue(const concurrent_queue& src, const Alloc& a = Alloc())

Effects

Constructs a copy of *src*.

4.3.3 template<typename InputIterator> concurrent_queue(InputIterator first, InputIterator last, const Alloc& a = Alloc())

Effects

Constructs a queue containing copies of elements in the iterator half-open interval [*first*,*last*).

4.3.4 ~concurrent_queue()

Effects

Destroys all items in the queue.

4.3.5 void push(const T& source)

Effects

Pushes a copy of `source` onto back of the queue.

4.3.6 bool try_pop (T& destination)

Effects

If value is available, pops it from the queue, assigns it to destination, and destroys the original value. Otherwise does nothing.

Returns

True if value was popped; false otherwise.

4.3.7 void clear()

Effects

Clears the queue. Afterwards `size()==0`.

4.3.8 size_type unsafe_size() const

Returns

Number of items in the queue. If there are concurrent modifications in flight, the value might not reflect the actual number of items in the queue.

4.3.9 bool empty() const

Returns

true if queue has no items; false otherwise.

4.3.10 Alloc get_allocator() const

Returns

Copy of allocator used to construct the queue.

4.3.11 Iterators

A concurrent_queue provides limited iterator support that is intended solely to allow programmers to inspect a queue during debugging. It provides iterator and const_iterator types. Both follow the usual STL conventions for forward iterators. The iteration order is from least recently pushed to most recently pushed. Modifying a concurrent_queue invalidates any iterators that reference it.

CAUTION: The iterators are relatively slow. They should be used only for debugging.

Example

The following program builds a queue with the integers 0..9, and then dumps the queue to standard output. Its overall effect is to print 0 1 2 3 4 5 6 7 8 9.

```
#include "tbb/concurrent_queue.h"
#include <iostream>

using namespace std;
using namespace tbb;

int main() {
    concurrent_queue<int> queue;
    for( int i=0; i<10; ++i )
        queue.push(i);
    typedef concurrent_queue<int>::iterator iter;
    for(iter i(queue.unsafe_begin()); i!=queue.unsafe_end(); ++i)
```

```
        cout << *i << " ";
    cout << endl;
    return 0;

}
```

4.3.11.1 iterator unsafe_begin()

Returns

`iterator` pointing to beginning of the queue.

4.3.11.2 iterator unsafe_end()

Returns

`iterator` pointing to end of the queue.

4.3.11.3 const_iterator unsafe_begin() const

Returns

`const_iterator` with pointing to beginning of the queue.

4.3.11.4 const_iterator unsafe_end() const

Returns

`const_iterator` pointing to end of the queue.

4.4 concurrent_bounded_queue Template Class

Summary

Template class for bounded dual queue with concurrent operations.

Syntax

```
template<typename T, class Alloc=cache_aligned_allocator<T> >
class concurrent_bounded_queue;
```

Header

```
#include "tbb/concurrent_queue.h"
```

Description

A concurrent_bounded_queue is similar to a concurrent_queue, but with the following differences:

- Adds the ability to specify a capacity. The default capacity makes the queue practically unbounded.
- Changes the push operation so that it waits until it can complete without exceeding the capacity.
- Adds a waiting pop operation that waits until it can pop an item.
- Changes the size_type to a *signed* type.
- Changes the size() operation to return the number of push operations minus the number of pop operations. For example, if there are 3 pop operations waiting on an empty queue, size() returns -3.

Members

To aid comparison, the parts that differ from concurrent_queue are in bold and annotated.

```
namespace tbb {
    template<typename T, typename
             Alloc=cache_aligned_allocator<T> >
    class concurrent_bounded_queue {
    public:
        // types
        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef Alloc allocator_type;
        // size_type is signed type
        typedef std::ptrdiff_t size_type;
        typedef std::ptrdiff_t difference_type;

        explicit concurrent_bounded_queue(const allocator_type& a
= allocator_type());
        concurrent_bounded_queue( const concurrent_bounded_queue&
src, const allocator_type& a = allocator_type());
        template<typename InputIterator>
        concurrent_bounded_queue( InputIterator begin,
InputIterator end, const allocator_type& a = allocator_type());
        ~concurrent_bounded_queue();

        // waits until it can push without exceeding capacity.
        void push( const T& source );
        // waits if *this is empty
        void pop( T& destination );
```

```
        // skips push if it would exceed capacity.
        bool try_push[7]( const T& source );
        bool try_pop[8]( T& destination );
        void clear() ;

        // safe to call during concurrent modification, can return negative size.
        size_type size() const;
        bool empty() const;
        size_type capacity() const;
        void set_capacity( size_type capacity );
        allocator_type get_allocator() const;

        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;

        // iterators (these are slow an intended only for
debugging)
        iterator unsafe_begin();
        iterator unsafe_end();
        const_iterator unsafe_begin() const;
        const_iterator unsafe_end() const;
    };
}
```

Because concurrent_bounded_queue is similar to concurrent_queue, the following subsections described only methods that differ.

4.4.1 void push(const T& source)

Effects

Waits until size()<capacity, and then pushes a copy of source onto back of the queue.

---

[7] Method try_push was called push_if_not_full in Intel® TBB 2.1.

[8] Method try_pop was called pop_if_present in Intel® TBB 2.1.

4.4.2 void pop(T& destination)

Effects

Waits until a value becomes available and pops it from the queue. Assigns it to `destination`. Destroys the original value.

4.4.3 bool try_push(const T& source)

Effects

If size()<capacity, pushes a copy of `source` onto back of the queue.

Returns

True if a copy was pushed; false otherwise.

4.4.4 bool try_pop(T& destination)

Effects

If a value is available, pops it from the queue, assigns it to destination, and destroys the original value. Otherwise does nothing.

Returns

True if a value was popped; false otherwise.

4.4.5 size_type size() const

Returns

Number of pushes minus number of pops. The result is negative if there are pop operations waiting for corresponding pushes. The result can exceed `capacity()` if the queue is full and there are push operations waiting for corresponding pops.

4.4.6 bool empty() const

Returns

```
size()<=0
```

4.4.7 size_type capacity() const

Returns

Maximum number of values that the queue can hold.

4.4.8 void set_capacity(size_type capacity)

Effects

Sets the maximum number of values that the queue can hold.

4.5 concurrent_vector

Summary

Template class for vector that can be concurrently grown and accessed.

Syntax

```
template<typename T, class Alloc=cache_aligned_allocator<T> >
class concurrent_vector;
```

Header

```
#include "tbb/concurrent_vector.h"
```

Description

A `concurrent_vector` is a container with the following features:

- Random access by index. The index of the first element is zero.
- Multiple threads can grow the container and append new elements concurrently.
- Growing the container does not invalidate existing iterators or indices.

A `concurrent_vector` meets all requirements for a Container and a Reversible Container as specified in the ISO C++ standard. It does not meet the Sequence requirements due to absence of methods `insert()` and `erase()`.

Members

```
namespace tbb {
    template<typename T, typename
Alloc=cache_aligned_allocator<T> >
    class concurrent_vector {
    public:
```

```
        typedef size_t size_type;
        typedef allocator-A-rebound-for-T [9] allocator_type;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T* pointer;
        typedef const T *const_pointer;
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef implementation-defined reverse_iterator;
        typedef implementation-defined const_reverse_iterator;

        // Parallel ranges
        typedef implementation-defined range_type;
        typedef implementation-defined const_range_type;
        range_type range( size_t grainsize );
        const_range_type range( size_t grainsize ) const;

        // Constructors
        explicit concurrent_vector( const allocator_type& a =
                                    allocator_type() );
        concurrent_vector( const concurrent_vector& x );
        template<typename M>
            concurrent_vector( const concurrent_vector<T, M>& x
);

        explicit concurrent_vector( size_type n,
            const T& t=T(),
            const allocator_type& a = allocator_type() );
        template<typename InputIterator>
            concurrent_vector(InputIterator first, InputIterator
last,
           const allocator_type& a=allocator_type());

        // Assignment
        concurrent_vector& operator=( const concurrent_vector& x
);
        template<class M>
```

---

[9] This rebinding follows practice established by both the Microsoft and GNU implementations of `std::vector`.

```
            concurrent_vector& operator=( const
concurrent_vector<T, M>& x );
        void assign( size_type n, const T& t );
        template<class InputIterator >
            void assign( InputIterator first, InputIterator last
);

        // Concurrent growth operations[10]
        iterator grow_by( size_type delta );
        iterator grow_by( size_type delta, const T& t );
        iterator grow_to_at_least( size_type n );
        iterator push_back( const T& item );

        // Items access
        reference operator[]( size_type index );
        const_reference operator[]( size_type index ) const;
        reference at( size_type index );
        const_reference at( size_type index ) const;
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;

        // Storage
        bool empty() const;
        size_type capacity() const;
        size_type max_size() const;
        size_type size() const;
        allocator_type get_allocator() const;

        // Non-concurrent operations on whole container
        void reserve( size_type n );
        void compact();
        void swap( concurrent_vector& vector );
        void clear();
        ~concurrent_vector();

        // Iterators
        iterator begin();
```

[10] The return types of the growth methods are different in Intel® TBB 2.2 than in prior versions. See footnotes in the descriptions of the individual methods for details.

```
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
        reverse_iterator rbegin();
        reverse_iterator rend();
        const_reverse_iterator rbegin() const;
        const_reverse_iterator rend() const;

        // C++0x extensions
        const_iterator cbegin() const;
        const_iterator cend() const;
        const_reverse_iterator crbegin() const;
        const_reverse_iterator crend() const;
    };

    // Template functions
    template<typename T, class A1, class A2>
        bool operator==( const concurrent_vector<T, A1>& a,
                         const concurrent_vector<T, A2>& b );

   template<typename T, class A1, class A2>
       bool operator!=( const concurrent_vector<T, A1>& a,
                        const concurrent_vector<T, A2>& b );

   template<typename T, class A1, class A2>
   bool operator<( const concurrent_vector<T, A1>& a,
                   const concurrent_vector<T, A2>& b );

   template<typename T, class A1, class A2>
       bool operator>( const concurrent_vector<T, A1>& a,
                       const concurrent_vector<T, A2>& b );

   template<typename T, class A1, class A2>
       bool operator<=( const concurrent_vector<T, A1>& a,
                        const concurrent_vector<T, A2>& b );

   template<typename T, class A1, class A2>
       bool operator>=(const concurrent_vector<T, A1>& a,
                       const concurrent_vector<T, A2>& b );

   template<typename T, class A>
       void swap(concurrent_vector<T, A>& a, concurrent_vector<T,
A>& b);

}
```

Exception Safety

Concurrent growing is fundamentally incompatible with ideal exception safety.[11] Nonetheless, `concurrent_vector` offers a practical level of exception safety.

Element type T must meet the following requirements:

- Its destructor must not throw an exception.
- If its default constructor can throw an exception, its destructor must be non-virtual and work correctly on zero-filled memory.

Otherwise the program's behavior is undefined.

Growth (4.5.3) and vector assignment (4.5.1) append a sequence of elements to a vector. If an exception occurs, the impact on the vector depends upon the cause of the exception:

- If the exception is thrown by the constructor of an element, then all subsequent elements in the appended sequence will be zero-filled.
- Otherwise, the exception was thrown by the vector's allocator. The vector becomes broken. Each element in the appended sequence will be in one of three states:
  - constructed
  - zero-filled
  - unallocated in memory

Once a vector becomes broken, care must be taken when accessing it:

- Accessing an unallocated element with method `at` causes an exception `std::range_error`. Any other way of accessing an unallocated element has undefined behavior.
- The values of `capacity()` and `size()` may be less than expected.
- Access to a broken vector via `back()` has undefined behavior.

However, the following guarantees hold for broken or unbroken vectors:

- Let *k* be an index of an unallocated element. Then `size()`≤`capacity()`≤*k*.
- Growth operations never cause `size()` or `capacity()` to decrease.

If a concurrent growth operation successfully completes, the appended sequence remains valid and accessible even if a subsequent growth operations fails.

---

[11] For example, consider P threads each appending N elements. To be perfectly exception safe, these operations would have to be serialized, because each operation has to know that the previous operation succeeded before allocating more indices.

Fragmentation

Unlike a `std::vector`, a `concurrent_vector` never moves existing elements when it grows. The container allocates a series of contiguous arrays. The first reservation, growth, or assignment operation determines the size of the first array. Using a small number of elements as initial size incurs fragmentation across cache lines that may increase element access time. The method `shrink_to_fit()` merges several smaller arrays into a single contiguous array, which may improve access time.

4.5.1 Construction, Copy, and Assignment

Safety

These operations must not be invoked concurrently on the same vector.

4.5.1.1 concurrent_vector(const allocator_type& a = allocator_type())

Effects

Constructs empty vector using optionally specified allocator instance.

4.5.1.2 concurrent_vector(size_type n, const_reference t=T(), const allocator_type& a = allocator_type());

Effects

Constructs vector of *n* copies of *t*, using optionally specified allocator instance. If *t* is not specified, each element is default constructed instead of copied.

4.5.1.3 template<typename InputIterator> concurrent_vector(InputIterator first, InputIterator last, const allocator_type& a = allocator_type())

Effects

Constructs vector that is copy of the sequence [*first*, *last*), making only N calls to the copy constructor of T, where N is the distance between first and last.

4.5.1.4 concurrent_vector(const concurrent_vector& src)

Effects

Constructs copy of *src*.

4.5.1.5 concurrent_vector& operator=(const concurrent_vector& src)

Effects

Assigns contents of *src* to **this*.

Returns

Reference to left hand side.

4.5.1.6 template<typename M> concurrent_vector& operator=(const concurrent_vector<T, M>& src)

Assign contents of *src* to *this.

Returns

Reference to left hand side.

4.5.1.7 void assign(size_type n, const_reference t)

Assign *n* copies of `t`.

4.5.1.8 template<class InputIterator > void assign(InputIterator first, InputIterator last)

Assign copies of sequence [*first*,*last*), making only N calls to the copy constructor of T, where N is the distance between first and last.

4.5.2 Whole Vector Operations

Safety

Concurrent invocation of these operations on the same instance is not safe.

4.5.2.1 void reserve(size_type n)

Effects

Reserves space for at least *n* elements.

Throws

`std::length_error` if *n*`>max_size()`. It can also throw an exception if the allocator throws an exception.

Safety

If an exception is thrown, the instance remains in a valid state.

4.5.2.2 void shrink_to_fit()[12]

Effects

Compacts the internal representation to reduce fragmentation.

4.5.2.3 void swap(concurrent_vector& x)

Swap contents of two vectors. Takes O(1) time.

4.5.2.4 void clear()

Effects

Erases all elements. Afterwards, `size()==0`. Does not free internal arrays.[13]

TIP: To free internal arrays, call `shrink_to_fit()` after `clear()`.

4.5.2.5 ~concurrent_vector()

Effects

Erases all elements and destroys the vector.

4.5.3 Concurrent Growth

Safety

The methods described in this section may be invoked concurrently on the same vector.

---

[12] Method `shrink_to_fit` was called `compact()` in Intel® TBB 2.1. It was renamed to match the C++0x `std::vector::shrink_to_fit()`.

[13] The original release of Intel® TBB 2.1 and its "update 1" freed the arrays. The change in "update 2" reverts back to the behavior of Intel® TBB 2.0. The motivation for not freeing the arrays is to behave similarly to `std::vector::clear()`.

4.5.3.1 iterator grow_by(size_type delta, const_reference t=T())[14]

Effects

Appends a sequence comprising *delta* copies of *t* to the end of the vector. If *t* is not specified, the new elements are default constructed.

Returns

Iterator pointing to beginning of appended sequence.

4.5.3.2 iterator grow_to_at_least(size_type n) [15]

Effects

Appends minimal sequence of elements such that vector.size()>=*n*. The new elements are default constructed. Blocks until all elements in range [0..*n*) are allocated (but not necessarily constructed if they are under construction by a different thread).

TIP: If a thread must know whether construction of an element has completed, consider the following technique. Instantiate the concurrent_vector using a zero_allocator (6.5). Define the constructor *T*() such that when it completes, it sets a field of *T* to non-zero. A thread can check whether an item in the concurrent_vector is constructed by checking whether the field is non-zero.

Returns

Iterator that points to beginning of appended sequence, or pointer to (*this)[*n*] if no elements were appended.

4.5.3.3 iterator push_back(const_reference value)[16]

Effects

Appends copy of value to the end of the vector.

Returns

Iterator that points to the copy.

---

[14] Return type was size_type in Intel® TBB 2.1.

[15] Return type was void in Intel® TBB 2.1.

[16] Return type was size_type in Intel® TBB 2.1.

4.5.4 Access

Safety

The methods described in this section may be concurrently invoked on the same vector as methods for concurrent growth (4.5.3). However, the returned reference may be to an element that is being concurrently constructed.

4.5.4.1 reference operator[](size_type index)

Returns

Reference to element with the specified index.

4.5.4.2 const_refrence operator[](size_type index) const

Returns

Const reference to element with the specified index.

4.5.4.3 reference at(size_type index)

Returns

Reference to element at specified index.

Throws

`std::out_of_range` if *index* ≥ `size()`.

4.5.4.4 const_reference at(size_type index) const

Returns

Const reference to element at specified index.

Throws

`std::out_of_range` if *index* ≥ `size()` or *index* is for broken portion of vector.

4.5.4.5 reference front()

Returns

`(*this)[0]`

4.5.4.6 const_reference front() const

Returns

`(*this)[0]`

4.5.4.7 reference back()

Returns

`(*this)[size()-1]`

4.5.4.8 const_reference back() const

Returns

`(*this)[size()-1]`

4.5.5 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (4.1). The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type `iterator`.

4.5.5.1 range_type range(size_t grainsize=1)

Returns

Range over entire `concurrent_vector` that permits read-write access.

4.5.5.2 const_range_type range(size_t grainsize=1) const

Returns

Range over entire `concurrent_vector` that permits read-only access.

4.5.6 Capacity

4.5.6.1 size_type size() const

Returns

Number of elements in the vector. The result may include elements that are allocated but still under construction by concurrent calls to any of the growth methods (4.5.3).

4.5.6.2 bool empty() const

Returns

`size()==0`

4.5.6.3 size_type capacity() const

Returns

Maximum size to which vector can grow without having to allocate more memory.

NOTE: Unlike an STL vector, a `concurrent_vector` does not move existing elements if it allocates more memory.

4.5.6.4 size_type max_size() const

Returns

Highest possible size of the vector could reach.

4.5.7 Iterators

Template class `concurrent_vector<T>` supports random access iterators as defined in Section 24.1.4 of the ISO C++ Standard. Unlike a `std::vector`, the iterators are not raw pointers. A `concurrent_vector<T>` meets the reversible container requirements in Table 66 of the ISO C++ Standard.

4.5.7.1 iterator begin()

Returns

`iterator` pointing to beginning of the vector.

4.5.7.2 const_iterator begin() const

Returns

`const_iterator` pointing to beginning of the vector.

4.5.7.3 iterator end()

Returns

`iterator` pointing to end of the vector.

4.5.7.4 const_iterator end() const

Returns

const_iterator pointing to end of the vector.

4.5.7.5 reverse_iterator rbegin()

Returns

reverse iterator pointing to beginning of reversed vector.

4.5.7.6 const_reverse_iterator rbegin() const

Returns

const_reverse_iterator pointing to beginning of reversed vector.

4.5.7.7 iterator rend()

Returns

const_reverse_iterator pointing to end of reversed vector.

4.5.7.8 const_reverse_iterator rend()

Returns

const_reverse_iterator pointing to end of reversed vector.

5 Thread Local Storage

Intel® Threading Building Blocks (Intel® TBB) provides two template classes for thread local storage. Both provide a thread-local element per thread. Both lazily create the elements on demand. They differ in their intended use models:

combinable provides thread-local storage for holding per-thread subcomputations that will later be reduced to a single result. It is PPL compatible.

enumerable_thread_specific provides thread-local storage that acts like a STL container with one element per thread. The container permits iterating over the elements using the usual STL iteration idioms.

This chapter also describes template class flatten2d, which assists a common idiom where an enumerable_thread_specific represents a container partitioner across threads.

5.1 combinable Template Class

Summary

Template class for holding thread-local values during a parallel computation that will be merged into to final.

Syntax

```
template<typename T> class combinable<T>;
```

Header

```
#include "tbb/combinable.h"
```

Description

A combinable<T> provides each thread with its own local instance of type T.

Members

```
namespace tbb {
    template <typename T>
    class combinable {
    public:
        combinable();

        template <typename FInit>
        combinable(FInit finit);}
```

```
        combinable(const combinable& other);

        ~combinable();

        combinable& operator=( const combinable& other);
        void clear();

        T& local();
        T& local(bool & exists);

        template<typename FCombine> T combine(FCombine fcombine);
        template<typename Func> void combine_each(Func f);
    };
}
```

5.1.1 combinable()

Effects

Constructs combinable such that any thread-local instances of T will be created using default construction.

5.1.2 template<typename FInit> combinable(FInit finit)

Effects

Constructs combinable such that any thread-local element will be created by copying the result of *finit*().

NOTE: The expression finit() must be safe to evaluate concurrently by multiple threads. It is evaluated each time a thread-local element is created.

5.1.3 combinable(const combinable& other);

Effects

Construct a copy of *other*, so that it has copies of each element in *other* with the same thread mapping.

5.1.4 ~combinable()

Effects

Destroy all thread-local elements in *this.

5.1.5 combinable& operator=(const combinable& other)

Effects

Set `*this` to be a copy of *`other`*.

5.1.6 void clear()

Effects

Remove all elements from `*this`.

5.1.7 T& local()

Effects

If thread-local element does not exist, create it.

Returns

Reference to thread-local element.

5.1.8 T& local(bool& exists)

Effects

Similar to `local()`, except that *`exists`* is set to true if an element was already present for the current thread; false otherwise.

Returns

Reference to thread-local element.

5.1.9 template<typename FCombine>T combine(FCombine fcombine)

Requires

Parameter *`fcombine`* should be an associative binary functor with the signature `T(T,T)` or `T(const T&,const T&)`.

Effects

Computes reduction over all elements using binary functor *`fcombine`*. If there are no elements, creates the result using the same rules as for creating a thread-local element.

Returns

Result of the reduction.

5.1.10 template<typename Func> void combine_each(Func f)

Requires

Parameter *f* should be a unary functor with the signature `void(T)` or `void(const T&)`.

Effects

Evaluates `f(`*x*`)` for each instance *x* of T in `*this`.

5.2 enumerable_thread_specific Template Class

Summary

Template class for thread local storage.

Syntax

```
enum ets_key_usage_type {
    ets_no_native_tls_keys,
    ets_tls_key_per_instance
};

template <typename T,
         typename Allocator=cache_aligned_allocator<T>,
         ets_key_usage_type ETS_key_type=ets_no_native_tls_keys>
class enumerable_thread_specific;
```

Header

```
#include "tbb/enumerable_thread_specific.h"
```

Description

An `enumerable_thread_specific` provides thread local storage (TLS) for elements of type *T*. An enumerable_thread_specific acts as a container by providing iterators and ranges across all of the thread-local elements.

The thread-local elements are created lazily. A freshly constructed `enumerable_thread_specific` has no elements. When a thread requests access to a `enumerable_thread_specific`, it creates an element corresponding to that thread. The number of elements is equal to the number of distinct threads that have accessed the enumerable_thread_specific and not the number of threads in use by the application. Clearing a enumerable_thread_specific removes all of its elements.

The ETS_key_usage_type parameter can be used to select between an implementation that consumes no native TLS keys and a specialization that offers higher performance but consumes 1 native TLS key per enumerable_thread_specific instance. If no ETS_key_usage_type parameter is provided, ets_no_native_tls_keys is used by default.

CAUTION: The number of native TLS keys is limited and can be fairly small, for example 64 or 128. Therefore it is recommended to restrict the use of the ets_tls_key_per_instance specialization to only the most performance critical cases.

Example

The following code shows a simple example usage of enumerable_thread_specific. The number of calls to null_parallel_for_body::operator() and total number of iterations executed are counted by each thread that participates in the parallel_for, and these counts are printed at the end of main.

```
#include <cstdio>
#include <utility>

#include "tbb/task_scheduler_init.h"
#include "tbb/enumerable_thread_specific.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

typedef enumerable_thread_specific< std::pair<int,int> >
        CounterType;

CounterType MyCounters (std::make_pair(0,0));

struct Body {
    void operator()(const tbb::blocked_range<int> &r) const {
        CounterType::reference my_counter = MyCounters.local();
        ++my_counter.first;
        for (int i = r.begin(); i != r.end(); ++i)
            ++my_counter.second;
    }
};

int main() {
```

```
    parallel_for( blocked_range<int>(0, 100000000), Body());

    for (CounterType::const_iterator i = MyCounters.begin();
         i != MyCounters.end();
         ++i)
    {
        printf("Thread stats:\n");
        printf("  calls to operator(): %d", i->first);
        printf("  total # of iterations executed: %d\n\n",
               i->second);
    }
}
```

Example with Lambda Expressions

Class enumerable_thread_specific has a method combine(*f*) that does reduction using binary functor *f*, which can be written using a lambda expression. For example, the previous example can be extended to sum the thread-local values by adding the following lines to the end of function main:

```
    std::pair<int,int> sum =
        MyCounters.combine([](std::pair<int,int> x,
                              std::pair<int,int> y) {
            return std::make_pair(x.first+y.first,
                                  x.second+y.second);
        });
    printf("Total calls to operator() = %d, "
           "total iterations = %d\n", sum.first, sum.second);
```

Members

```
namespace tbb {
    template <typename T,
              typename Allocator=cache_aligned_allocator<T>,
              ets_key_usage_type ETS_key_type=ets_single_key >
    class enumerable_thread_specific {
    public:
        // Basic types
        typedef Allocator allocator_type;
        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T* pointer;
        typedef implementation-dependent size_type;
        typedef implementation-dependent difference_type;

        // Iterator types
        typedef implementation-dependent iterator;
```

```
        typedef implementation-dependent const_iterator;

        // Parallel range types
        typedef implementation-dependent range_type;
        typedef implementation-dependent const_range_type;

        // Whole container operations
        enumerable_thread_specific();
        enumerable_thread_specific(
            const enumerable_thread_specific &ets
        );
        enumerable_thread_specific(const T &_exemplar);
        ~enumerable_thread_specific();
        void clear();

        // Concurrent operations
        reference local();
        reference local(bool& existis);
        size_type size() const;
        bool empty() const;

        // Combining
        template<typename FCombine> T combine(FCombine fcombine);
        template<typename Func> void combine_each(Func f);

        // Parallel iteration
        range_type range( size_t grainsize=1 );
        const_range_type range( size_t grainsize=1 ) const;

        // Iterators
        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
    };
}
```

5.2.1 Whole Container Operations

Safety

These operations must not be invoked concurrently on the same instance of `enumerable_thread_specific`.

5.2.1.1 enumerable_thread_specific()

Effects

Constructs an `enumerable_thread_specific` where each local copy will be default constructed.

5.2.1.2 enumerable_thread_specific(const enumerable_thread_specific &e)

Effects

Copy construct an `enumerable_thread_specific`. The values are copy constructed from the values in *e* and have same thread correspondence.

5.2.1.3 enumerable_thread_specific(const &exemplar)

Effects

Constructs an `enumerable_thread_specific` where each local copy will be copy constructed from *`exemplar`*.

5.2.1.4 ~enumerable_thread_specific()

Effects

Destroys all elements in `*this`. Destroys any native TLS keys that were created for this instance.

5.2.1.5 void clear()

Effects

Destroys all elements in *this. Destroys and then recreates any native TLS keys used in the implementation.

NOTE: In the current implementation, there is no performance advantage of using clear instead of destroying and reconstructing an `enumerable_thread_specific`.

5.2.2 Concurrent Operations

5.2.2.1 reference local()

Returns

A reference to the element of `*this` that corresponds to the current thread.

Effects

If there is no current element corresponding to the current thread, then constructs a new element. A new element is copy-constructed if an exemplar was provided to the constructor for `*this`, otherwise a new element is default constructed.

5.2.2.2 T& local(bool& exists)

Effects

Similar to `local()`, except that *exists* is set to true if an element was already present for the current thread; false otherwise.

Returns

Reference to thread-local element.

5.2.2.3 size_type size() const

Returns

The number of elements in `*this`. The value is equal to the number of distinct threads that have called local() after *this was constructed or most recently cleared.

5.2.2.4 bool empty() const

Returns

```
size()==0.
```

5.2.3 Combining

The methods in this section iterate across the entire container.

5.2.3.1 template<typename FCombine>T combine(FCombine fcombine)

Requires

Parameter *fcombine* should be an associative binary functor with the signature `T(T,T)` or `T(const T&,const T&)`.

Effects

Computes reduction over all elements using binary functor fcombine. If there are no elements, creates the result using the same rules as for creating a thread-local element.

Returns

Result of the reduction.

5.2.3.2 template<typename Func> void combine_each(Func f)

Requires

Parameter f should be a unary functor with the signature `void(T)` or `void(const T&)`.

Effects

Evaluates `f(`*x*`)` for each instance *x* of T in `*this`.

5.2.4 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (4.1). The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type iterator.

5.2.4.1 const_range_type range(size_t grainsize=1) const

Returns

A `const_range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

5.2.4.2 range_type range(size_t grainsize=1)

Returns

A `range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

5.2.5 Iterators

Template class `enumerable_thread_specific` supports random access iterators, which enable iteration over the set of all elements in the container.

5.2.5.1 iterator begin()

Returns

`iterator` pointing to beginning of the set of elements.

5.2.5.2 iterator end()

Returns

`iterator` pointing to end of the set of elements.

5.2.5.3 const_iterator begin() const

Returns

`const_iterator` pointing to beginning of the set of elements.

5.2.5.4 const_iterator end() const

Returns

`const_iterator` pointing to the end of the set of elements.

5.3 flattened2d Template Class

Summary

Adaptor that provides a flattened view of a container of containers.

Syntax

```
template<typename Container>
class flattened2;

template <typename Container>
flattened2d<Container> flatten2d(const Container &c);

template <typename Container>
flattened2d<Container> flatten2d(
    const Container &c,
    const typename Container::const_iterator b,
    const typename Container::const_iterator e);
```

Header

```
#include "tbb/enumerable_thread_specific.h"
```

Description

A `flattened2d` provides a flattened view of a container of containers. Iterating from `begin()` to `end()`visits all of the elements in the inner containers. This can be useful when traversing a `enumerable_thread_specific` whose elements are containers.

The utility function `flatten2d` creates a `flattened2d` object from a container.

Example

The following code shows a simple example usage of flatten2d and flattened2d. Each thread collects the values of i that are evenly divisible by K in a thread-local vector. In main, the results are printed by using a `flattened2d` to simplify the traversal of all of the elements in all of the local vectors.

```
#include <iostream>
#include <utility>
#include <vector>

#include "tbb/task_scheduler_init.h"
#include "tbb/enumerable_thread_specific.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

// A VecType has a separate std::vector<int> per thread
typedef enumerable_thread_specific< std::vector<int> > VecType;
VecType MyVectors;
int K = 1000000;

struct Func {
    void operator()(const blocked_range<int>& r) const {
        VecType::reference v = MyVectors.local();
        for (int i=r.begin(); i!=r.end(); ++i)
            if( i%k==0 )
                v.push_back(i);
    }
};

int main() {
    parallel_for(blocked_range<int>(0, 100000000),
                 Func());

    flattened2d<VecType> flat_view = flatten2d( MyVectors );
    for( flattened2d<VecType>::const_iterator
         i = flat_view.begin(); i != flat_view.end(); ++i)
        cout << *i << endl;
    return 0;
}
```

Members

```
namespace tbb {

    template<typename Container>
    class flattened2d {

    public:
        // Basic types
        typedef implementation-dependent size_type;
        typedef implementation-dependent difference_type;
        typedef implementation-dependent allocator_type;
        typedef implementation-dependent value_type;
        typedef implementation-dependent reference;
        typedef implementation-dependent const_reference;
        typedef implementation-dependent pointer;
        typedef implementation-dependent const_pointer;

        typedef implementation-dependent iterator;
        typedef implementation-dependent const_iterator;

        flattened2d( const Container& c );

        flattened2d( const Container& c,
                     typename Container::const_iterator first,
                     typename Container::const_iterator last );

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;

        size_type size() const;
    };

    template <typename Container>
    flattened2d<Container> flatten2d(const Container &c);

    template <typename Container>
    flattened2d<Container> flatten2d(
        const Container &c,
        const typename Container::const_iterator first,
        const typename Container::const_iterator last);
}
```

5.3.1 Whole Container Operations

Safety

These operations must not be invoked concurrently on the same `flattened2d`.

5.3.1.1 flattened2d(const Container& c);

Effects

Constructs a `flattened2d` representing the sequence of elements in the inner containers contained by outer container c.

5.3.1.2 flattened2d(const Container& c, typename Container::const_iterator first, typename Container::const_iterator last)

Effects

Constructs a flattened2d representing the sequence of elements in the inner containers in the half-open intervale [*first*, *last*) of Container c.

5.3.2 Concurrent Operations

Safety

These operations may be invoked concurrently on the same `flattened2d`.

5.3.2.1 size_type size() const

Returns

The sum of the sizes of the inner containers that are viewable in the `flattened2d`.

5.3.3 Iterators

Template class `flattened2d` supports foward iterators only.

5.3.3.1 iterator begin()

Returns

`iterator` pointing to beginning of the set of local copies.

5.3.3.2 iterator end()

Returns

`iterator` pointing to end of the set of local copies.

5.3.3.3 const_iterator begin() const

Returns

`const_iterator` pointing to beginning of the set of local copies.

5.3.3.4 const_iterator end() const

Returns

`const_iterator` pointing to the end of the set of local copies.

5.3.4 Utility Functions

5.3.4.1 template <typename Container> flattened2d<Container> flatten2d(const Container &c, const typename Container::const_iterator b, const typename Container::const_iterator e)

Returns

Constructs and returns a flattened2d that provides iterators that traverse the elements in the containers within the half-open range [b, e) of Container c.

5.3.4.2 template <typename Container> flattened2d(const Container &c)

Returns

Constructs and returns a flattened2d that provides iterators that traverse the elements in all of the containers within Container c.

6 Memory Allocation

This section describes classes related to memory allocation.

6.1 Allocator Concept

The allocator concept for allocators in Intel® Threading Building Blocks is similar to the "Allocator requirements" in Table 32 of the ISO C++ Standard, but with further guarantees required by the ISO C++ Standard (Section 20.1.5 paragraph 4) for use with ISO C++ containers. Table 21 summarizes the allocator concept. Here, A and B represent instances of the allocator class.

Table 21: Allocator Concept

Pseudo-Signature	Semantics
`typedef T* A::pointer`	Pointer to *T*.
`typedef const T* A::const_pointer`	Pointer to const *T*.
`typedef T& A::reference`	Reference to *T*.
`typedef const T& A::const_reference`	Reference to const *T*.
`typedef T A::value_type`	Type of value to be allocated.
`typedef size_t A::size_type`	Type for representing number of values.
`typedef ptrdiff_t A::difference_type`	Type for representing pointer difference.

`template<typename U> struct rebind {`
`    typedef A<U> A::other;`
`};`
	Rebind to a different type *U*

`A() throw()`	Default constructor.
`A( const A& ) throw()`	Copy constructor.
`template<typename U> A( const A& )`	Rebinding constructor.
`~A() throw()`	Destructor.
`T* A::address( T& x ) const`	Take address.
`const T* A::const_address( const T& x ) const`	Take const address.
`T* A::allocate( size_type n, const void* hint=0 )`	Allocate space for n values.
`void A::deallocate( T* p, size_t  n )`	Deallocate n values.
`size_type A::max_size() const throw()`	Maximum plausible

Pseudo-Signature	Semantics
	argument to method allocate.
`void A::construct( T* p, const T& value )`	`new(p) T(value)`
`void A::destroy( T* p )`	`p->T::~T()`
`bool operator==( const A&, const B& )`	Return true.
`bool operator!=( const A&, const B& )`	Return false.

Model Types

Template classes `tbb_allocactor` (6.2), `scalable_allocator` (6.3), and `cached_aligned_allocator` (6.4), and `zero_allocator` (6.5) model the Allocator concept.

6.2 tbb_allocator Template Class

Summary

Template class for scalable memory allocation if available; possibly non-scalable otherwise.

Syntax

```
template<typename T> class tbb_allocator
```

Header

```
#include "tbb/tbb_allocator.h"
```

Description

A `tbb_allocator` allocates and frees memory via the Intel® TBB malloc library if it is available, otherwise it reverts to using malloc and free.

TIP: Set the environment variable `TBB_VERSION` to 1 to find out if the Intel® TBB malloc library is being used. Details are in Section 2.7.2.

6.3 scalable_allocator Template Class

Summary

Template class for scalable memory allocation.

Syntax

```
template<typename T> class scalable_allocator;
```

Header

```
#include "tbb/scalable_allocator.h"
```

Description

A scalable_allocator allocates and frees memory in a way that scales with the number of processors. A scalable_allocator models the allocator requirements described in Table 21. Using a scalable_allocator in place of std::allocator may improve program performance. Memory allocated by a scalable_allocator should be freed by a scalable_allocator, not by a std::allocator.

CAUTION: The scalable_allocator requires that the tbb malloc library be available. If the library is missing, calls to the scalable allocator fail. In contrast, tbb_allocator falls back on malloc and free if the tbbmalloc library is missing.

Members

See Allocator concept (6.1).

Acknowledgement

The scalable memory allocator incorporates McRT technology developed by Intel's PSL CTG team.

6.3.1 C Interface to Scalable Allocator

Summary

Low level interface for scalable memory allocation.

Syntax

```
extern "C" {
    // Scalable analogs of C memory allocator
    void* scalable_malloc( size_t size );
    void  scalable_free( void* ptr );
    void* scalable_calloc( size_t nobj, size_t size );
    void* scalable_realloc( void* ptr, size_t size );

    // Analog of _msize/malloc_size/malloc_usable_size.
    size_t scalable_msize( void* ptr );

    // Scalable analog of posix_memalign
    int scalable_posix_memalign( void** memptr,
                                 size_t alignment, size_t size );
```

```
    // Aligned allocation
    void* scalable_aligned_malloc( size_t size,
                                   size_t alignment);
    void scalable_aligned_free( void* ptr );
    void* scalable_aligned_realloc( void* ptr, size_t size,
                                    size_t alignment );

}
```

Header

```
#include "tbb/scalable_allocator.h"
```

Description

These functions provide a C level interface to the scalable allocator. Each routine scalable_*x* behaves analogously to library function *x*. The routines form the two families shown in Table 22. Storage allocated by a scalable_*x* function in one family must be freed or resized by a scalable_*x* function in the same family, not by a C standard library function. Likewise storage allocated by a C standard library function should not be freed or resized by a scalable_*x* function.

Table 22: C Interface to Scalable Allocator

Family	Allocation Routine	Deallocation Routine	Analogous Library
1	`scalable_malloc`	`scalable_free`	C standard library
	`scalable_calloc`		
	`scalable_realloc`		
	`scalable_posix_memalign`		POSIX*[17]
2	`scalable_aligned_malloc`	`scalable_aligned_free`	Microsoft* C run-time library
	`scalable_aligned_free`		
	`scalable_aligned_realloc`		

[17] See "The Open Group* Base Specifications Issue 6", IEEE* Std 1003.1, 2004 Edition for the definition of `posix_memalign`.

6.3.1.1 size_t scalable_msize(void* ptr)

Returns

The usable size of the memory block pointed to by *ptr* if it was allocated by the scalable allocator. Returns zero if *ptr* does not point to such a block.

6.4 cache_aligned_allocator Template Class

Summary

Template class for allocating memory in way that avoids false sharing.

Syntax

```
template<typename T> class cache_aligned_allocator;
```

Header

```
#include "tbb/cache_aligned_allocator.h"
```

Description

A cache_aligned_allocator allocates memory on cache line boundaries, in order to avoid false sharing. False sharing is when logically distinct items occupy the same cache line, which can hurt performance if multiple threads attempt to access the different items simultaneously. Even though the items are logically separate, the processor hardware may have to transfer the cache line between the processors as if they were sharing a location. The net result can be much more memory traffic than if the logically distinct items were on different cache lines.

A cache_aligned_allocator models the allocator requirements described in Table 21. It can be used to replace a std::allocator. Used judiciously, cache_aligned_allocator can improve performance by reducing false sharing. However, it is sometimes an inappropriate replacement, because the benefit of allocating on a cache line comes at the price that cache_aligned_allocator implicitly adds pad memory. The padding is typically 128 bytes. Hence allocating many small objects with cache_aligned_allocator may increase memory usage.

Members

```
namespace tbb {

    template<typename T>
    class cache_aligned_allocator {
    public:
        typedef T* pointer;
        typedef const T* const_pointer;
        typedef T& reference;
```

```
        typedef const T& const_reference;
        typedef T value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        template<typename U> struct rebind {
            typedef cache_aligned_allocator<U> other;
        };

    #if _WIN64
        char* _Charalloc( size_type size );
    #endif /* _WIN64 */

        cache_aligned_allocator() throw();
        cache_aligned_allocator( const cache_aligned_allocator& )
throw();
        template<typename U>
        cache_aligned_allocator( const
cache_aligned_allocator<U>& ) throw();
        ~cache_aligned_allocator();

        pointer address(reference x) const;
        const_pointer address(const_reference x) const;

        pointer allocate( size_type n, const void* hint=0 );
        void deallocate( pointer p, size_type );
        size_type max_size() const throw();

        void construct( pointer p, const T& value );
        void destroy( pointer p );
    };

    template<>
    class cache_aligned_allocator<void> {
    public:
        typedef void* pointer;
        typedef const void* const_pointer;
        typedef void value_type;
        template<typename U> struct rebind {
            typedef cache_aligned_allocator<U> other;
        };
    };

    template<typename T, typename U>
    bool operator==( const cache_aligned_allocator<T>&,
                     const cache_aligned_allocator<U>& );
```

```
    template<typename T, typename U>
    bool operator!=( const cache_aligned_allocator<T>&,
                     const cache_aligned_allocator<U>& );

}
```

For sake of brevity, the following subsections describe only those methods that differ significantly from the corresponding methods of `std::allocator`.

6.4.1 pointer allocate(size_type n, const void* hint=0)

Effects

Allocates *size* bytes of memory on a cache-line boundary. The allocation may include extra hidden padding.

Returns

Pointer to the allocated memory.

6.4.2 void deallocate(pointer p, size_type n)

Requirements

Pointer *p* must be result of method `allocate(n)`. The memory must not have been already deallocated.

Effects

Deallocates memory pointed to by p. The deallocation also deallocates any extra hidden padding.

6.4.3 char* _Charalloc(size_type size)

NOTE: This method is provided only on 64-bit Windows* OS platforms. It is a non-ISO method that exists for backwards compatibility with versions of Window's containers that seem to require it. Please do not use it directly.

6.5 zero_allocator

Summary

Template class for allocator that returns zeroed memory.

Syntax

```
template <typename T,
          template<typename U> class Alloc = tbb_allocator>
class zero_allocator: public Alloc<T>;
```

Header

```
#include "tbb/tbb_allocator.h"
```

Description

A zero_allocator allocates zeroed memory. A zero_allocator<*T*,*A*> can be instantiated for any class *A* that models the Allocator concept. The default for *A* is tbb_allocator. A zero_allocator forwards allocation requests to *A* and zeros the allocation before returning it.

Members

```
namespace tbb {
    template <typename T, template<typename U> class Alloc =
tbb_allocator>
    class zero_allocator : public Alloc<T> {
    public:
        typedef Alloc<T> base_allocator_type;
        typedef typename base_allocator_type::value_type
                                              value_type;
        typedef typename base_allocator_type::pointer pointer;
        typedef typename base_allocator_type::const_pointer
                                              const_pointer;
        typedef typename base_allocator_type::reference
                                              reference;
        typedef typename base_allocator_type::const_reference
                                              const_reference;
        typedef typename base_allocator_type::size_type
                                              size_type;
        typedef typename base_allocator_type::difference_type
                                              difference_type;
        template<typename U> struct rebind {
            typedef zero_allocator<U, Alloc> other;
        };

        zero_allocator() throw() { }
        zero_allocator(const zero_allocator &a) throw();
        template<typename U>
        zero_allocator(const zero_allocator<U> &a) throw();

        pointer allocate(const size_type n, const void* hint=0);
    };
```

```
}
```

6.6 aligned_space Template Class

Summary

Uninitialized memory space for an array of a given type.

Syntax

```
template<typename T, size_t N> class aligned_space;
```

Header

```
#include "tbb/aligned_space.h"
```

Description

An aligned_space occupies enough memory and is sufficiently aligned to hold an array *T*[*N*]. The client is responsible for initializing or destroying the objects. An aligned_space is typically used as a local variable or field in scenarios where a block of fixed-length uninitialized memory is needed.

Members

```
namespace tbb {
    template<typename T, size_t N>
    class aligned_space {
    public:
        aligned_space();
        ~aligned_space();
        T* begin();
        T* end();
    };
}
```

6.6.1 aligned_space()

Effects

None. Does not invoke constructors.

6.6.2 ~aligned_space()

Effects

None. Does not invoke destructors.

6.6.3 T* begin()

Returns

Pointer to beginning of storage.

6.6.4 T* end()

Returns

```
begin()+N
```

7 Synchronization

The library supports mutual exclusion and atomic operations.

7.1 Mutexes

Mutexes provide MUTual EXclusion of threads from sections of code.

In general, strive for designs that minimize the use of explicit locking, because it can lead to serial bottlenecks. If explicitly locking is necessary, try to spread it out so that multiple threads usually do not contend to lock the same mutex.

7.1.1 Mutex Concept

The mutexes and locks here have relatively spartan interfaces that are designed for high performance. The interfaces enforce the *scoped locking pattern,* which is widely used in C++ libraries because:

1. Does not require the programmer to remember to release the lock
2. Releases the lock if an exception is thrown out of the mutual exclusion region protected by the lock

There are two parts to the pattern: a *mutex* object, for which construction of a *lock* object acquires a lock on the mutex and destruction of the *lock* object releases the lock. Here's an example:

```
{
    // Construction of myLock acquires lock on myMutex
    M::scoped_lock myLock( myMutex );
    ... actions to be performed while holding the lock ...
    // Destruction of myLock releases lock on myMutex
}
```

If the actions throw an exception, the lock is automatically released as the block is exited.

Table 23 shows the requirements for the Mutex concept for a mutex type M

Table 23: Mutex Concept

Pseudo-Signature	Semantics
`M()`	Construct unlocked mutex.
`~M()`	Destroy unlocked mutex.
`typename M::scoped_lock`	Corresponding scoped-lock type.
`M::scoped_lock()`	Construct lock without acquiring

Pseudo-Signature	Semantics
	mutex.
M::scoped_lock(M&)	Construct lock and acquire lock on mutex.
M::~scoped_lock()	Release lock (if acquired).
M::scoped_lock::acquire(M&)	Acquire lock on mutex.
bool M::scoped_lock::try_acquire(M&)	Try to acquire lock on mutex. Return true if lock acquired, false otherwise.
M::scoped_lock::release()	Release lock.
static const bool M::is_rw_mutex	True if mutex is reader-writer mutex; false otherwise.
static const bool M::is_recursive_mutex	True if mutex is recursive mutex; false otherwise.
static const bool M::is_fair_mutex	True if mutex is fair; false otherwise.

Table 24 summarizes the classes that model the Mutex concept.

Table 24: Mutexes that Model the Mutex Concept

	Scalable	Fair	Reentrant	Long Wait	Size
mutex	OS dependent	OS dependent	No	Blocks	≥ 3 words
recursive_mutex	OS dependent	OS dependent	Yes	Blocks	≥ 3 words
spin_mutex	No	No	No	Yields	1 byte
queuing_mutex	✓	✓	No	Yields	1 word
spin_rw_mutex	No	No	No	Yields	1 word
queuing_rw_mutex	✓	✓	No	Yields	1 word
null_mutex	-	Yes	Yes	-	empty
null_rw_mutex	-	Yes	Yes	-	empty

See the Tutorial, Section 6.1.1, for a discussion of the mutex properties and the rationale for null mutexes.

7.1.1.1 C++0x Compatibility

Classes mutex, recursive_mutex, spin_mutex, and spin_rw_mutex support the C++0x methods described in Table 25.

Table 25: C++0x Methods Available for Some Mutexes.

Pseudo-Signature	Semantics
void M::lock()	Acquire lock.
bool M::try_lock()	Try to acquire lock on mutex. Return true if lock

	acquired, false otherwise.
`void M::unlock()`	Release lock.

Class `spin_rw_mutex` also has methods `read_lock()` and `try_read_lock()` for corresponding operations that acquire reader locks.

7.1.2 mutex Class

Summary

Class that models Mutex Concept using underlying OS locks.

Syntax

```
class mutex;
```

Header

```
#include "tbb/mutex.h"
```

Description

A `mutex` models the Mutex Concept (7.1.1). It is a wrapper around OS calls that provide mutual exclusion. The advantages of using `mutex` instead of the OS calls are:

- Portable across all operating systems supported by Intel® Threading Building Blocks.
- Releases the lock if an exception is thrown from the protected region of code.

Members

See Mutex Concept (7.1.1).

7.1.3 recursive_mutex Class

Summary

Class that models Mutex Concept using underlying OS locks and permits recursive acquisition.

Syntax

```
class recursive_mutex;
```

Header

```
#include "tbb/recursive_mutex.h"
```

Description

A recursive_mutex is similar to a mutex (7.1.2), except that a thread may acquire multiple locks on it. The thread must release all locks on a recursive_mutex before any other thread can acquire a lock on it.

Members

See Mutex Concept (7.1.1).

7.1.4 spin_mutex Class

Summary

Class that models Mutex Concept using a spin lock.

Syntax

```
class spin_mutex;
```

Header

```
#include "tbb/spin_mutex.h"
```

Description

A spin_mutex models the Mutex Concept (7.1.1). A spin_mutex is not scalable, fair, or recursive. It is ideal when the lock is lightly contended and is held for only a few machine instructions. If a thread has to wait to acquire a spin_mutex, it busy waits, which can degrade system performance if the wait is long. However, if the wait is typically short, a spin_mutex significantly improve performance compared to other mutexes.

Members

See Mutex Concept (7.1.1).

7.1.5 queuing_mutex Class

Summary

Class that models Mutex Concept that is fair and scalable.

Syntax

```
class queuing_mutex;
```

Header

```
#include "tbb/queuing_mutex.h"
```

Description

A queuing_mutex models the Mutex Concept (7.1.1). A queuing_mutex is scalable, in the sense that if a thread has to wait to acquire the mutex, it spins on its own local cache line. A queuing_mutex is fair. Threads acquire a lock on a mutex in the order that they request it. A queuing_mutex is not recursive.

The current implementation does busy-waiting, so using a queuing_mutex may degrade system performance if the wait is long.

Members

See Mutex Concept (7.1.1).

7.1.6 ReaderWriterMutex Concept

The ReaderWriterMutex concept extends the Mutex Concept to include the notion of reader-writer locks. It introduces a boolean parameter write that specifies whether a writer lock (write =true) or reader lock (write =false) is being requested. Multiple reader locks can be held simultaneously on a ReaderWriterMutex if it does not have a writer lock on it. A writer lock on a ReaderWriterMutex excludes all other threads from holding a lock on the mutex at the same time.

Table 26 shows the requirements for a ReaderWriterMutex RW. They form a superset of the Mutex Concept (7.1.1).

Table 26: ReaderWriterMutex Concept

Pseudo-Signature	Semantics
`RW()`	Construct unlocked mutex.
`~RW()`	Destroy unlocked mutex.
`typename RW::scoped_lock`	Corresponding scoped-lock type.
`RW::scoped_lock()`	Construct lock without acquiring mutex.
`RW::scoped_lock(RW&, bool write=true)`	Construct lock and acquire lock on mutex.
`RW::~scoped_lock()`	Release lock (if acquired).
`RW::scoped_lock::acquire(RW&, bool write=true)`	Acquire lock on mutex.
`bool RW::scoped_lock::try_acquire(RW&, bool write=true)`	Try to acquire lock on mutex. Return `true` if lock acquired, `false` otherwise.
`RW::scoped_lock::release()`	Release lock.
`bool RW::scoped_lock::upgrade_to_writer()`	Change reader lock to writer lock.

Pseudo-Signature	Semantics
`bool RW::scoped_lock::downgrade_to_reader()`	Change writer lock to reader lock.
`static const bool RW::is_rw_mutex = true`	True.
`static const bool RW::is_recursive_mutex`	True if mutex is reader-writer mutex; false otherwise. For all current reader-writer mutexes, false.
`static const bool RW::is_fair_mutex`	True if mutex is fair; false otherwise.

The following subsections explain the semantics of the ReaderWriterMutex concept in detail.

Model Types

`spin_rw_mutex` (7.1.7) and `queuing_rw_mutex` (7.1.8) model the ReaderWriterMutex concept.

7.1.6.1 ReaderWriterMutex()

Effects

Constructs unlocked `ReaderWriterMutex`.

7.1.6.2 ~ReaderWriterMutex()

Effects

Destroys unlocked `ReaderWriterMutex`. The effect of destroying a locked `ReaderWriterMutex` is undefined.

7.1.6.3 ReaderWriterMutex::scoped_lock()

Effects

Constructs a `scoped_lock` object that does not hold a lock on any mutex.

7.1.6.4 ReaderWriterMutex::scoped_lock(ReaderWriterMutex& rw, bool write =true)

Effects

Constructs a `scoped_lock` object that acquires a lock on mutex *rw*. The lock is a writer lock if *write* is true; a reader lock otherwise.

7.1.6.5 ReaderWriterMutex::~scoped_lock()

Effects

If the object holds a lock on a `ReaderWriterMutex`, releases the lock.

7.1.6.6 void ReaderWriterMutex:: scoped_lock:: acquire(ReaderWriterMutex& rw, bool write=true)

Effects

Acquires a lock on mutex *rw*. The lock is a writer lock if *write* is true; a reader lock otherwise.

7.1.6.7 bool ReaderWriterMutex:: scoped_lock::try_acquire(ReaderWriterMutex& rw, bool write=true)

Effects

Attempts to acquire a lock on mutex *rw*. The lock is a writer lock if *write* is true; a reader lock otherwise.

Returns

`true` if the lock is acquired, `false` otherwise.

7.1.6.8 void ReaderWriterMutex:: scoped_lock::release()

Effects

Releases lock. The effect is undefined if no lock is held.

7.1.6.9 bool ReaderWriterMutex:: scoped_lock::upgrade_to_writer()

Effects

Changes reader lock to a writer lock. The effect is undefined if the object does not already hold a reader lock.

Returns

`false` if lock was released in favor of another upgrade request and then reacquired; `true` otherwise.

7.1.6.10 bool ReaderWriterMutex:: scoped_lock::downgrade_to_reader()

Effects

Changes writer lock to a reader lock. The effect is undefined if the object does not already hold a writer lock.

Returns

`false` if lock was released and reacquired; `true` otherwise.

NOTE: Intel's current implementations for `spin_rw_mutex` and `queuing_rw_mutex` always return `true`. Different implementations might sometimes return `false`.

7.1.7 spin_rw_mutex Class

Summary

Class that models ReaderWriterMutex Concept that is unfair and not scalable.

Syntax

```
class spin_rw_mutex;
```

Header

```
#include "tbb/spin_rw_mutex.h"
```

Description

A `spin_rw_mutex` models the ReaderWriterMutex Concept (7.1.6). A `spin_rw_mutex` is not scalable, fair, or recursive. It is ideal when the lock is lightly contended and is held for only a few machine instructions. If a thread has to wait to acquire a `spin_rw_mutex`, it busy waits, which can degrade system performance if the wait is long. However, if the wait is typically short, a `spin_rw_mutex` significantly improve performance compared to other mutexes..

Members

See ReaderWriterMutex concept (7.1.6).

7.1.8 queuing_rw_mutex Class

Summary

Class that models ReaderWriterMutex Concept that is fair and scalable.

Syntax

```
class queuing_rw_mutex;
```

Header

```
#include "tbb/queuing_rw_mutex.h"
```

Description

A queuing_rw_mutex models the ReaderWriterMutex Concept (7.1.6). A queuing_rw_mutex is scalable, in the sense that if a thread has to wait to acquire the mutex, it spins on its own local cache line. A queuing_rw_mutex is fair. Threads acquire a lock on a queuing_rw_mutex in the order that they request it. A queuing_rw_mutex is not recursive.

Members

See ReaderWriterMutex concept (7.1.6).

7.1.9 null_mutex Class

Summary

Class that models Mutex Concept buts does nothing.

Syntax

```
class null_mutex;
```

Header

```
#include "tbb/null_mutex.h"
```

Description

A null_mutex models the Mutex Concept (7.1.1) syntactically, but does nothing. It is useful for instantiating a template that expects a Mutex, but no mutual exclusion is actually needed for that instance.

Members

See Mutex Concept (7.1.1).

7.1.10 null_rw_mutex Class

Summary

Class that models ReaderWriterMutex Concept but does nothing.

Syntax

```
class null_rw_mutex;
```

Header

```
#include "tbb/null_rw_mutex.h"
```

Description

A `null_rw_mutex` models the ReaderWriterMutex Concept (7.1.6) syntactically, but does nothing. It is useful for instantiating a template that expects a ReaderWriterMutex, but no mutual exclusion is actually needed for that instance..

Members

See ReaderWriterMutex concept (7.1.6).

7.2 atomic Template Class

Summary

Template class for atomic operations.

Syntax

```
template<typename T> atomic;
```

Header

```
#include "tbb/atomic.h"
```

Description

An `atomic<T>` supports atomic read, write, fetch-and-add, fetch-and-store, and compare-and-swap. Type T may be an integral type, enumeration type, or a pointer type. When T is a pointer type, arithmetic operations are interpreted as pointer arithmetic. For example, if *x* has type `atomic<float*>` and a float occupies four bytes, then ++*x* advances *x* by four bytes. Arithmetic on `atomic<T>` is not allowed if T is an enumeration type, `void*`, or `bool`.

Some of the methods have template method variants that permit more selective memory fencing. On IA-32 and Intel® 64 architecture processors, they have the same effect as the non-templated variants. On IA-64 architecture (Itanium®) processors, they may improve performance by allowing the memory subsystem more latitude on the orders of reads and write. Using them may improve performance. Table 27 shows the fencing for the non-template form.

Table 27: Operation Order Implied by Non-Template Methods

Kind	Description	Default For
acquire	Operations after the atomic operation never move over it.	read
release	Operations before the atomic operation never move over it.	write
sequentially consistent	Operations on either side never move over it and furthermore, the sequentially consistent atomic operations have a global order.	`fetch_and_store`, `fetch_and_add`, `compare_and_swap`

CAUTION: The copy constructor for class `atomic<T>` is not atomic. To atomically copy an `atomic<T>`, default-construct the copy first and assign to it. Below is an example that shows the difference.

```
atomic<T> y(x);  // Not atomic
atomic<T> z;
z=x;             // Atomic assignment
```

The copy constructor is not atomic because it is compiler generated. Introducing any non-trivial constructors might remove an important property of atomic<T>: namespace scope instances are zero-initialized before namespace scope dynamic initializers run. This property can be essential for code executing early during program startup.

To create an atomic<T> with a specific value, default-construct it first, and afterwards assign a value to it.

Members

```
namespace tbb {
    enum memory_semantics {
        acquire,
        release
    };

    struct atomic<T> {
        typedef T value_type;

        template<memory_semantics M>
        value_type compare_and_swap( value_type new_value,
                                     value_type comparand );

        value_type compare_and_swap( value_type new_value,
                                     value_type comparand );

        template<memory_semantics M>
```

```
        value_type fetch_and_store( value_type new_value );

        value_type fetch_and_store( value_type new_value );

        operator value_type() const;

        value_type operator=( value_type new_value );
        atomic<T>& operator=( const atomic<T>& value );

        // The following members exist only if T is an integral
        // or pointer type.

        template<memory_semantics M>
        value_type fetch_and_add( value_type addend );

        value_type fetch_and_add( value_type addend );

        template<memory_semantics M>
        value_type fetch_and_increment();

        value_type fetch_and_increment();

        template<memory_semantics M>
        value_type fetch_and_decrement();

        value_type fetch_and_decrement();

        value_type operator+=(value_type);
        value_type operator-=(value_type);
        value_type operator++();
        value_type operator++(int);
        value_type operator--();
        value_type operator--(int);
    };
}
```

So that an atomic<T*> can be used like a pointer to T, the specialization atomic<T*> also defines:

```
        T* operator->() const;
```

7.2.1 memory_semantics Enum

Description

Defines values used to select the template variants that permit more selective control over visibility of operations (see Table 27).

7.2.2 value_type fetch_and_add(value_type addend)

Effects

Let *x* be the value of `*this`. Atomically updates *x* = *x* + addend.

Returns

Original value of *x*.

7.2.3 value_type fetch_and_increment()

Effects

Let *x* be the value of `*this`. Atomically updates *x* = *x* + 1.

Returns

Original value of *x*.

7.2.4 value_type fetch_and_decrement()

Effects

Let *x* be the value of `*this`. Atomically updates *x* = *x* – 1.

Returns

Original value of *x*.

7.2.5 value_type compare_and_swap

```
value_type compare_and_swap( value_type new_value, value_type
comparand )
```

Effects

Let *x* be the value of `*this`. Atomically compares *x* with comparand, and if they are equal, sets *x*=new_value.

Returns

Original value of *x*.

7.2.6 value_type fetch_and_store(value_type new_value)

Effects

Let *x* be the value of `*this`. Atomically exchanges old value of *x* with new_value.

Returns

Original value of *x*.

8 Timing

Parallel programming is about speeding up *wall clock* time, which is the real time that it takes a program to run. Unfortunately, some of the obvious wall clock timing routines provided by operating systems do not always work reliably across threads, because the hardware thread clocks are not synchronized. The library provides support for timing across threads. The routines are wrappers around operating services that we have verified as safe to use across threads.

8.1 tick_count Class

Summary

Class for computing wall-clock times.

Syntax

```
class tick_count;
```

Header

```
#include "tbb/tick_count.h"
```

Description

A `tick_count` is an absolute timestamp. Two `tick_count` objects may be subtracted to compute a relative time `tick_count`::interval_t, which can be converted to seconds.

Example

```
using namespace tbb;

void Foo() {
    tick_count t0 = tick_count::now();
    ...action being timed...
    tick_count t1 = tick_count::now();
    printf("time for action = %g seconds\n", (t1-t0).seconds() );
}
```

Members

```
namespace tbb {

    class tick_count {
    public:
```

```
        class interval_t;
        static tick_count now();
    };

    tick_count::interval_t  operator-( const tick_count& t1,
                                       const tick_count& t0 );
} // tbb
```

8.1.1 static tick_count tick_count::now()

Returns

Current wall clock timestamp.

CAUTION: On Windows* operating systems, the current implementation uses the function `QueryPerformanceCounter`. Microsoft warns that some systems may have bugs in their basic input/output system (BIOS) or hardware abstraction layer (HAL) that cause different processors to return different results.

8.1.2 tick_count::interval_t operator−(const tick_count& t1, const tick_count& t0)

Returns

Relative time that t1 occurred after t0.

8.1.3 tick_count::interval_t Class

Summary

Class for relative wall-clock time.

Syntax

```
class tick_count::interval_t;
```

Header

```
#include "tbb/tick_count.h"
```

Description

A `tick_count::interval_t` represents relative wall clock duration.

Members

```
namespace tbb {
```

```
    class tick_count::interval_t {
    public:
        interval_t();
        explicit interval_t( double sec );
        double seconds() const;
        interval_t operator+=( const interval_t& i );
        interval_t operator-=( const interval_t& i );
    };

    tick_count::interval_t operator+(
        const tick_count::interval_t& i,
        const tick_count::interval_t& j );

    tick_count::interval_t operator-(
        const tick_count::interval_t& i,
        const tick_count::interval_t& j );

} // namespace tbb
```

8.1.3.1 interval_t()

Effects

Constructs `interval_t` representing zero time duration.

8.1.3.2 interval_t(double sec)

Effects

Constructs interval_t representing specified number of seconds.

8.1.3.3 double seconds() const

Returns

Time interval measured in seconds.

8.1.3.4 interval_t operator+=(const interval_t& i)

Effects

```
*this = *this + i
```

Returns

Reference to `*this`.

8.1.3.5 interval_t operator-=(const interval_t& i)

Effects

```
*this = *this - i
```

Returns

Reference to *this.

8.1.3.6 interval_t operator+ (const interval_t& i, const interval_t& j)

Returns

Interval_t representing sum of intervals *i* and *j*.

8.1.3.7 interval_t operator- (const interval_t& i, const interval_t& j)

Returns

Interval_t representing difference of intervals *i* and *j*.

9 Task Groups

This chapter covers the high-level interface to the task scheduler. Chapter 10 covers the low-level interface. The high-level interface lets you easily create groups of potentially parallel tasks from functors or lambda expressions. The low-level interface permits more detailed control, such as control over exception propogation and affinity.

Summary

High-level interface for running functions in parallel.

Syntax

```
template<typename Func> task_handle;
template<typename Func> task_handle<Func> make_task( const Func&
f );
enum task_group_status;
class task_group;
class structured_task_group;
bool is_current_task_group_canceling();
```

Header

```
#include "tbb/task_group.h"
```

Requirements

Functor arguments for various methods in this chapter should meet the requirements in Table 28.

Table 28: Requirements on functor arguments

Pseudo-Signature	Semantics
`Func::Func (const Func&)`	Copy constructor.
`Func::~Func ()`	Destructor.
`void Func::operator()() const;`	Evaluate functor.

9.1 task_group Class

Description

A task_group represents concurrent execution of a group of tasks. Tasks may be dynamically added to the group as it is executing.

Example with Lambda Expressions

```
#include "tbb/task_group.h"

using namespace tbb;

int Fib(int n) {
    if( n<2 ) {
        return n;
    } else {
        int x, y;
        task_group g;
        g.run([&]{x=Fib(n-1);}); // spawn a task
        g.run([&]{y=Fib(n-2);}); // spawn another task
        g.wait();                // wait for both tasks to complete
        return x+y;
    }
}
```

CAUTION: Creating a large number of tasks for a single task_group is not scalable, because task creation becomes a serial bottleneck. If creating more than a small number of concurrent tasks, consider using `parallel_for` (3.4) or `parallel_invoke` (3.11) instead, or structure the spawning as a recursive tree.

Members

```
namespace tbb {
    class task_group {
    public:
        task_group();
        ~task_group();

        template<typename Func>
        void run( const Func& f );

        template<typename Func>
        void run( task_handle<Func>& handle );

        template<typename Func>
        void run_and_wait( const Func& f );

        template<typename Func>
        void run_and_wait( task_handle<Func>& handle );

        task_group_status wait();
        bool is_canceling();
        void cancel();
```

```
    }
}
```

9.1.1 task_group()

Constructs an empty task group.

9.1.2 ~task_group()

Requires

Method wait must be called before destroying a task_group, otherwise the destructor throws an exception.

9.1.3 template<typename Func> void run(const Func& f)

Effects

Spawn a task that computes *f*() and return immediately.

9.1.4 template<typename Func> void run (task_handle<Func>& handle);

Effects

Spawn a task that computes *handle*() and return immediately.

9.1.5 template<typename Func> void run_and_wait(const Func& f)

Effects

Equivalent to {run(*f*); wait();}, but guarantees that *f* runs on the current thread.

NOTE: Template method run_and_wait is intended to be more efficient than separate calls to run and wait.

9.1.6 template<typename Func> void run _and_wait(task_handle<Func>& handle);

Effects

Equivalent to `{run(handle); wait();}`, but guarantees that *handle*() runs on the current thread.

NOTE: Template method `run_and_wait` is intended to be more efficient than separate calls to `run` and `wait`.

9.1.7 task_group_status wait()

Effects

Wait for all tasks in the group to complete or be cancelled.

9.1.8 bool is_canceling()

Returns

True if this task group is cancelling its tasks.

9.1.9 void cancel()

Effects

Cancel all tasks in this `task_group`.

9.2 task_group_status Enum

A `task_group_status` represents the status of a `task_group`.

Members

```
namespace tbb {
    enum task_group_status {
        not_complete,  // Not cancelled and not all tasks in group have completed.
        complete,      // Not cancelled and all tasks in group have completed
        canceled       // Task group received cancellation request
    };
}
```

9.3 task_handle Template Class

Summary

Template function for creating a task_handle from a function or functor.

Description

Class task_handle is used primarily in conjunction with class structured_task_group. For sake of uniformity, class task_group also accepts task_handle arguments.

Members

```
template<typename Func>
class task_handle {
public:
    task_handle( const Func& f );
    void operator()() const;
};
```

9.4 make_task Template Function

Summary

Template function for creating a task_handle from a function or functor.

Syntax

```
template<typename Func>
task_handle<Func> make_task( const Func& f );
```

Returns

task_handle<*Func*>(*f*)

9.5 structured_task_group Class

Description

A structured_task_group is like a task_group, but has only a subset of the functionality. It may permit performance optimizations in the future. The restrictions are:

- Methods run and run_and_wait take only task_handle arguments, not general functors.

- Methods run and run_and_wait do not copy their task_handle arguments. The caller must not destroy those arguments until after wait or run_and_wait returns.
- Methods run, run_and_wait, cancel, and wait should be called only by the thread that created the structured_task_group.
- Method wait (or run_and_wait) should be called only once on a given instance of structured_task_group.

Example

The function fork_join below evaluates f1() and f2(), in parallel if resources permit.

```
#include "tbb/task_group.h"

using namespace tbb;

template<typename Func1, typename Func2>
void fork_join( const Func1& f1, const Func2& f2 ) {
    structured_task_group group;

    task_handle<Func1> h1(f1);
    group.run(h1);              // spawn a task

    task_handle<Func2> h2(f2);
    group.run(h2);              // spawn another task

    group.wait();               // wait for both tasks to complete
    // now safe to destroy h1 and h2
}
```

Members

```
namespace tbb {
    class structured_task_group {
    public:
        structured_task_group();
        ~structured_task_group();

        template<typename Func>
        void run( task_handle<Func>& handle );

        template<typename Func>
        void run_and_wait( task_handle<Func>& handle );

        task_group_status wait();
        bool is_canceling();
```

```
        void cancel();
    };
}
```

9.6 is_current_task_group_canceling Function

Returns

True if innermost task group executing on this thread is cancelling its tasks.

10 Task Scheduler

Intel Threading Building Blocks (Intel® TBB) provides a task scheduler, which is the engine that drives the algorithm templates (Section 3) and task groups (Section 9). You may also call it directly. Using tasks is often simpler and more efficient than using threads, because the task scheduler takes care of a lot of details.

The tasks are quanta of computation. The scheduler maps these onto physical threads. The mapping is non-preemptive. Each thread has a method `execute()`. Once a thread starts running `execute()`, the task is bound to that thread until `execute()` returns. During that time, the thread services other tasks only when it waits on child tasks, at which time it may run the child tasks, or if there are no pending child tasks, service tasks created by other threads.

The task scheduler is intended for parallelizing computationally intensive work. Because task objects are not scheduled preemptively, they should not make calls that might block for long periods, because meanwhile that thread is precluded from servicing other tasks.

CAUTION: There is no guarantee that *potentially* parallel tasks *actually* execute in parallel, because the scheduler adjusts actual parallelism to fit available worker threads. For example, given a single worker thread, the scheduler creates no actual parallelism. For example, it is generally unsafe to use tasks in a producer consumer relationship, because there is no guarantee that the consumer runs at all while the producer is running.

Potential parallelism is typically generated by a split/join pattern. Two basic patterns of split/join are supported. The most efficient is continuation-passing form, in which the programmer constructs an explicit "continuation" task. The parent task splits child tasks and specifies a continuation task to be executed when the children complete. The continuation inherits the parent's ancestor. The parent task then exits; i.e., it does not block on its children. The children subsequently run, and after they (or their continuations) finish, the continuation task starts running. Figure 3 shows the steps. The running tasks at each step are shaded.

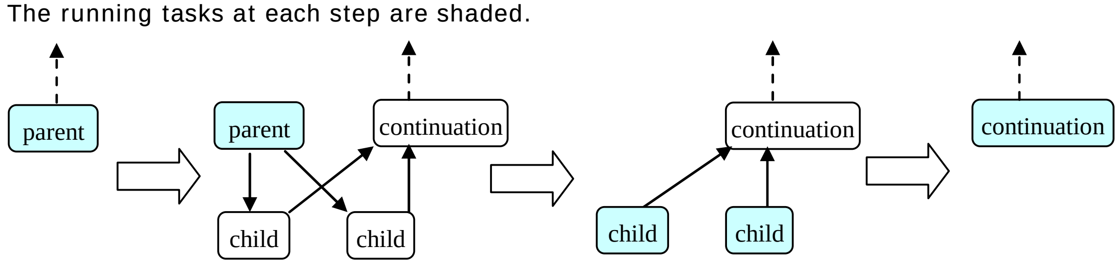

Figure 3: Continuation-passing Style

Explicit continuation passing is efficient, because it decouples the thread's stack from the tasks. However, it is more difficult to program. A second pattern is "blocking style", which uses implicit continuations. It is sometimes less efficient in performance, but more convenient to program. In this pattern, the parent task blocks until its children complete, as shown in Figure 4.

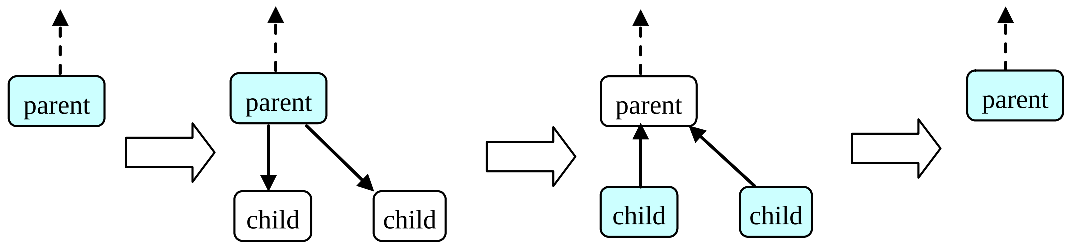

Figure 4: Blocking Style

The convenience comes with a price. Because the parent blocks, its thread's stack cannot be popped yet. The thread must be careful about what work it takes on, because continually stealing and blocking could cause the stack to grow without bound. To solve this problem, the scheduler constrains a blocked thread such that it never executes a task that is less deep than its deepest blocked task. This constraint may impact performance because it limits available parallelism, and tends to cause threads to select smaller (deeper) subtrees than they would otherwise choose.

10.1 Scheduling Algorithm

The scheduler employs a technique known as *work stealing*. Each thread keeps a "ready pool" of tasks that are ready to run. The ready pool is structured as a deque of `task` objects. A thread chooses its next task according to the first rule below that applies:

1. The task returned by `task::execute()` that the thread invoked previously.
2. The task whose lastly completed child was completed by this thread.
3. A task popped from the end of the deque.
4. A task with affinity for the thread.
5. A task popped from in the beginning of another randomly chosen thread's deque.

When a thread spawns a task, it pushes it onto the end of its own deque. Hence rule (3) above gets the task most recently spawned by the thread, whereas rule (5) gets the least recently spawned task of another thread.

Work stealing tends to strike a good balance between locality of reference, space efficiency, and parallelism. The work-stealing algorithm in the task scheduler is similar to that used by Cilk (Blumofe 1995). The notion of work-stealing dates back to the 1980s (Kumar 1987). The thread affinity support is more recent (Acar 2000).

10.2 task_scheduler_init Class

Summary

Class that explicity represents thread's interest in task scheduling services.

Syntax

```
class task_scheduler_init;
```

Header

```
#include "tbb/task_scheduler_init.h"
```

Description

Using `task_scheduler_init` is optional in Intel® TBB 2.2. By default, Intel® TBB 2.2 automatically creates a task scheduler the first time that a thread uses task scheduling services and destroys it when the last such thread exits.

An instance of `task_scheduler_init` can be used to control the following aspects of the task scheduler:

- When the task scheduler is constructed and destroyed.
- The number of threads used by the task scheduler.
- The stack size for worker threads.

To override the automatic defaults for task scheduling, a `task_scheduler_init` must become active before the first use of task scheduling services.

A `task_scheduler_init` is either "active" or "inactive".

The default constructor for a `task_scheduler_init` activates it, and the destructor deactivates it. To defer activation, pass the value `task_scheduler_init::deferred` to the constructor. Such a `task_scheduler_init` may be activated later by calling method `initialize`. Destruction of an active `task_scheduler_init` implicitly deactivates it. To deactivate it earlier, call method `terminate`.

An optional parameter to the constructor and method `initialize` allow you to specify the number of threads to be used for `task` execution. This parameter is useful for scaling studies during development, but should not be set for production use.

TIP: The reason for not specifying the number of threads in production code is that in a large software project, there is no way for various components to know how many threads would be optimal for other threads. Hardware threads are a shared global resource. It is best to leave the decision of how many threads to use to the task scheduler.

To minimize time overhead, it is best to rely upon automatic creation of the task scheduler, or create a single `task_scheduler_init` object whose activation spans all uses of the library's task scheduler. A `task_scheduler_init` is not assignable or copy-constructible.

Example

```
// Sketch of one way to do a scaling study
#include <iostream>
#include "tbb/task_scheduler_init"

int main() {
    int n = task_scheduler_init::default_num_threads();
    for( int p=1; p<=n; ++p ) {
        // Construct task scheduler with p threads
        task_scheduler_init init(p);
        tick_count t0 = tick_count::now();
         ... execute parallel algorithm using task or
             template algorithm here...
        tick_count t1 = tick_count::now();
        double t = (t1-t0).seconds();
        cout << "time = " << t << " with " << p << "threads\n";
        // Implicitly destroy task scheduler.
    }
    return 0;
}
```

Members

```
namespace tbb {
    typedef unsigned-integral-type stack_size_type;

    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,
                             stack_size_type thread_stack_size=0
);
        ~task_scheduler_init();
        void initialize( int number_of_threads=automatic );
        void terminate();
        static int default_num_threads();
    };
} // namespace tbb
```

10.2.1 task_scheduler_init(int number_of_threads=automatic, stack_size_type thread_stack_size=0)

Requirements

The value number_of_threads shall be one of the values in Table 29.

Effects

If number_of_threads==task_scheduler_init::deferred, nothing happens, and the task_scheduler_init remains inactive. Otherwise, the task_scheduler_init is activated as follows. If the thread has no other active task_scheduler_init objects, the thread allocates internal thread-specific resources required for scheduling task objects. If there were no threads with active task_scheduler_init objects yet, then internal worker threads are created as described in Table 29. These workers sleep until needed by the task scheduler.

The optional parameter thread_stack_size specifies the stack size of each worker thread. A value of 0 specifies use of a default stack size.

Table 29: Values for number_of_threads

number_of_threads	Semantics
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.

10.2.2 ~task_scheduler_init()

Effects

If the task_scheduler_init is inactive, nothing happens. Otherwise, the task_scheduler_init is deactivated as follows. If the thread has no other active task_scheduler_init objects, the thread deallocates internal thread-specific resources required for scheduling task objects. If no existing thread has any active task_scheduler_init objects, then the internal worker threads are terminated.

10.2.3 void initialize(int number_of_threads=automatic)

Requirements

The task_scheduler_init shall be inactive.

Effects

Similar to constructor (10.2.1).

10.2.4 void terminate()

Requirements

The `task_scheduler_init` shall be active.

Effects

Deactivates the `task_scheduler_init` without destroying it. The description of the destructor (10.2.2) specifies what deactivation entails.

10.2.5 int default_num_threads()

Returns

One more than the number of worker threads that `task_scheduler_init` creates by default.

10.2.6 bool is_active() const

Returns

True if `*this` is active as described in Section 10.2; false otherwise.

10.2.7 Mixing with OpenMP

Mixing OpenMP with Intel® Threading Building Blocks is supported. Performance may be less than a pure OpenMP or pure Intel® Threading Building Blocks solution if the two forms of parallelism are nested.

An OpenMP parallel region that plans to use the `task` scheduler should create a `task_scheduler_init` inside the parallel region, because the parallel region may create new threads unknown to Intel® Threading Building Blocks. Each of these new OpenMP threads, like native threads, must create a `task_scheduler_init` object before using Intel® Threading Building Blocks algorithms. The following example demonstrates how to do this.

```
void OpenMP_Calls_TBB( int n ) {
#pragma omp parallel
    {
        task_scheduler_init init;
#pragma omp for
        for( int i=0; i<n; ++i ) {
            ...can use class task or
```

```
                Intel® Threading Building Blocks algorithms here
...
        }
    }
}
```

10.3 task Class

Summary

Base class for tasks.

Syntax

```
class task;
```

Header

```
#include "tbb/task.h"
```

Description

Class `task` is the base class for tasks. You are expected to derive classes from `task`, and at least override the virtual method `task* task::execute()`.

Each instance of `task` has associated attributes, that while not directly visible, must be understood to fully grasp how `task` objects are used. The attributes are described in Table 30.[18]

Table 30: Task Attributes[19]

Attribute	Description
parent	Either null, or a pointer to another task whose refcount field will be decremented after the present task completes. Typically, the parent is the task that allocated the present task, or a task allocated as the continuation of that parent.
refcount	The number of Tasks that have this is their parent. Increments and decrement of refcount are always atomic.

TIP: Always allocate memory for `task` objects using special overloaded new operators (10.3.2) provided by the library, otherwise the results are undefined. Destruction of a `task` is normally implicit. The copy constructor and assignment operators for task are

[18] The depth attribute in Intel® TBB 2.1 no longer exists (A.6).

[19] The ownership attribute and restrictions in Intel® TBB 2.1 no longer exist.

not accessible. This prevents accidental copying of a task, which would be ill-defined and corrupt internal data structures.

Notation

Some member descriptions illustrate effects by diagrams such as Figure 5.

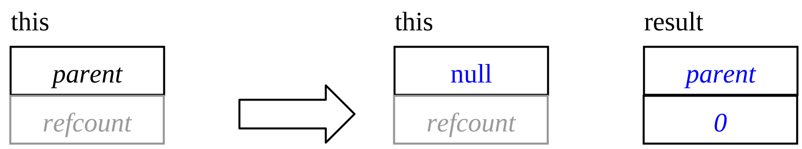

Figure 5: Example Effect Diagram

Conventions in these diagrams are as follows:

- The big arrow denotes the transition from the old state to the new state.
- Each task's state is shown as a box divided into *parent* and *refcount* sub-boxes.
- Gray denotes state that is ignored. Sometimes ignored state is simply left blank..
- Black denotes state that is read.
- Blue denotes state that is written.

Members

In the description below, types *proxy1*…*proxy5* are internal types. Methods returning such types should only be used in conjunction with the special overloaded new operators, as described in Section (10.3.2).

```
namespace tbb {
    class task {
    protected:
        task();

    public:
        virtual ~task() {}

        virtual task* execute() = 0;

        // task allocation and destruction
        static proxy1 allocate_root();
        static proxy2 allocate_root( task_group_context& );
        proxy3 allocate_continuation();
        proxy4 allocate_child();
        proxy5 allocate_additional_child_of( task& );

        // Explicit task destruction
        void destroy( task& victim );
```

```
        // Recycling
        void recycle_as_continuation();
        void recycle_as_child_of( task& new_parent );
        void recycle_to_reexecute();

        // Synchronization
        void set_ref_count( int count );
        void increment_ref_count();
        int decrement_ref_count();
        void wait_for_all();
        void spawn( task& child );
        void spawn( task_list& list );
        void spawn_and_wait_for_all( task& child );
        void spawn_and_wait_for_all( task_list& list );
        static void spawn_root_and_wait( task& root );
        static void spawn_root_and_wait( task_list& root );

        // task context
        static task& self();
        task* parent() const;
        bool is_stolen_task() const;

        // Cancellation
        bool cancel_group_execution();
        bool is_cancelled() const;

        // Affinity
        typedef implementation-defined-unsigned-type affinity_id;
        virtual void note_affinity( affinity_id id );
        void set_affinity( affinity_id id );
        affinity_id affinity() const;

        // task debugging
        enum state_type {
            executing,
            reexecute,
            ready,
            allocated,
            freed
        };
        int ref_count() const;
        state_type state() const;
    };
} // namespace tbb
```

```
void *operator new( size_t bytes, const proxy1& p );
void operator delete( void* task, const proxy1& p );
void *operator new( size_t bytes, const proxy2& p );
void operator delete( void* task, const proxy2& p );
void *operator new( size_t bytes, const proxy3& p );
void operator delete( void* task, const proxy3& p );
void *operator new( size_t bytes, proxy4& p );
void operator delete( void* task, proxy4& p );
void *operator new( size_t bytes, proxy5& p );
void operator delete( void* task, proxy5& p );
```

10.3.1 task Derivation

Class `task` is an abstract base class. You **must** override method `task::execute`. Method `execute` should perform the necessary actions for running the task, and then return the next `task` to execute, or NULL if the scheduler should choose the next task to execute. Typically, if non-NULL, the returned task is one of the children of `this`. Unless one of the recycle/reschedule methods described in Section (10.3.4) is called while method `execute()` is running, the `this` object will be implicitly destroyed after method `execute` returns.

Override the virtual destructor if necessary to release resources allocated by the constructor.

Override `note_affinity` to improve cache reuse across tasks, as described in Section 10.3.8.

10.3.1.1 Processing of execute()

When the scheduler decides that a thread should begin executing a *task*, it performs the following steps:

1. Invokes `execute()` and waits for it to return.
2. If the task has not been marked by a method `recycle_*`:
    a. Calls the task's destructor.
    b. If the task's *parent* is not null, then atomically decrements *parent->refcount*, and if becomes zero, puts the *parent* into the ready pool.
    c. Frees the memory of the task for reuse.
3. If the task has been marked for recycling:
    a. If marked by `recycle_to_reexecute`, puts the task back into the ready pool.
    b. Otherwise it was marked by `recycle_as_child` or `recycle_as_continuation`.

10.3.2 task Allocation

Always allocate memory for `task` objects using one of the special overloaded new operators. The allocation methods do not construct the `task`. Instead, they return a proxy object that can be used as an argument to an overloaded version of operator new provided by the library.

In general, the allocation methods must be called before any of the tasks allocated are spawned. The exception to this rule is `allocate_additional_child_of(t)`, which can be called even if `task` *t* is already running. The proxy types are defined by the implementation. The only guarantee is that the phrase "`new(proxy) T(...)`"allocates and constructs a task of type *T*. Because these methods are used idiomatically, the headings in the subsection show the idiom, not the declaration. The argument `this` is typically implicit, but shown explicitly in the headings to distinguish instance methods from static methods.

TIP: Allocating tasks larger than 216 bytes might be significantly slower than allocating smaller tasks. In general, task objects should be small lightweight entities.

10.3.2.1 new(task::allocate_root(task_group_context& group)) *T*

Allocate a `task` of type *T* with the specified cancellation group. Figure 6 summarizes the state transition.

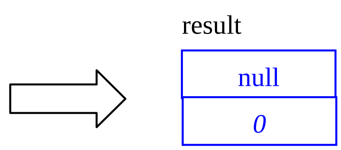

Figure 6: Effect of task::allocate_root()

Use method `spawn_root_and_wait` (10.3.5.9) to execute the `task`.

10.3.2.2 new(task::allocate_root()) *T*

Like `new(task::allocate_root(task_group_context&))` except that cancellation group is the current innermost cancellation group.

10.3.2.3 new(x.allocate_continuation()) *T*

Allocates and constructs a task of type *T*, and transfers the *parent* from *x* to the new task. No reference counts change. Figure 7 summarizes the state transition.

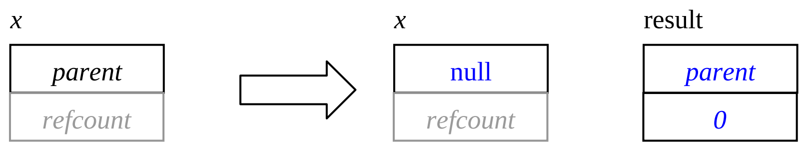

Figure 7: Effect of allocate_continuation()

10.3.2.4 new(*x*.allocate_child()) T

Effects

Allocates a `task` with `this` as its *parent*. Figure 8 summarizes the state transition.

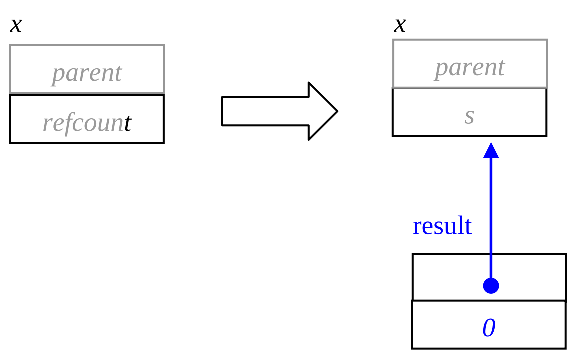

Figure 8: Effect of allocate_child()

If using explicit continuation passing, then the continuation, not the parent, should call the allocation method, so that *parent* is set correctly.

If the number of tasks is not a small fixed number, consider building a `task_list` (10.5) of the children first, and spawning them with a single call to `task::spawn` (10.3.5.5). If a `task` must spawn some children before all are constructed, it should use `task::allocate_additional_child_of(*this)` instead, because that method atomically increments *refcount*, so that the additional child is properly accounted. However, if doing so, the `task` must protect against premature zeroing of *refcount* by using a blocking-style task pattern.

10.3.2.5 new(*x*.task::allocate_additional_child_of(*y*))

Effects

Allocates a `task` as a child of another `task` *y*. The result becomes a child of *y*, not *x*. Task *y* may be already running or have other children running. Figure 9 summarizes the state transition.

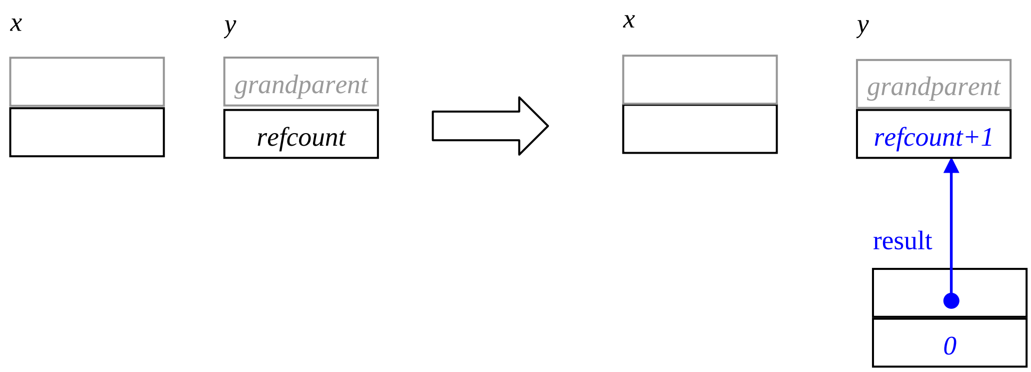

Figure 9: Effect of allocate_additional_child_of(parent)

Because `parent` may already have running children, the increment of parent.*refcount* is thread safe (unlike the other allocation methods, where the increment is not thread safe). When adding a child to a parent with other children running, it is up to the programmer to ensure that the parent's *refcount* does not prematurely reach 0 and trigger execution of the parent before the child is added.

10.3.3 Explicit task Destruction

Usually, a `task` is automatically destroyed by the scheduler after its method `execute` returns. But sometimes `task` objects are used idiomatically (e.g. for reference counting) without ever running `execute`. Such tasks should be disposed with method `destroy`.

10.3.3.1 void destroy(task& victim)

Requirements

The reference count of *victim* should be 0. This requirement is checked in the debug version of the library.

Effects

Calls destructor and deallocates memory for *victim*. If this has non-null *parent*, atomically decrements *parent->refcount*. The *parent* is **not** put into the ready pool if *parent->refcount* becomes zero. Figure 10 summarizes the state transition.

The implicit argument `this` is used internally, but not visibly affected. A `task` is allowed to destroy itself; e.g., "`this->destroy(*this)`" is permitted unless the `task` has been spawned but has not yet completed method `execute`.

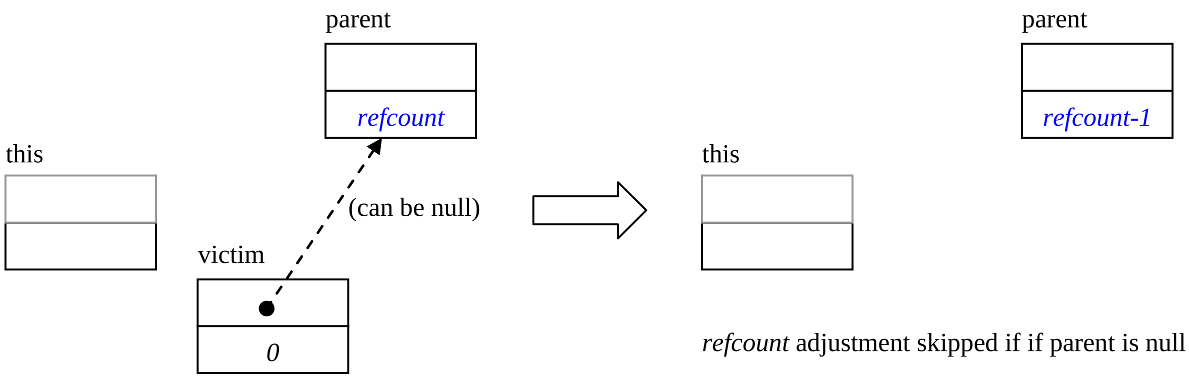

Figure 10: Effect of destroy(victim)

10.3.4 Recycling Tasks

It is often more efficient to recycle a `task` object rather than reallocate one from scratch. Often the parent can become the continuation, or one of the children.

10.3.4.1 void recycle_as_continuation()

Requirements

Must be called while method `execute()` is running.

The *refcount* for the recycled task should be set to n, where n is the number of children of the continuation task.

NOTE: The caller must guarantee that the task's *refcount* does not become zero until after the method `execute()` returns. If this is not possible, use the method `recycle_as_safe_continuation()` instead, and set refcount to `n+1`.

Effects

Causes `this` to not be destroyed when method `execute()` returns.

10.3.4.2 void recycle_as_safe_continuation()

Requirements

Must be called while method `execute()` is running.

The *refcount* for the recycled task should be set to `n+1`, where n is the number of children of the continuation task. The additional `+1` represents the task to be recycled.

Effects

Causes `this` to not be destroyed when method `execute()` returns.

This method avoids race conditions that can arise from using the method `recycle_as_continuation`. The race occurs when:

The method `execute()` recycles `this` as a continuation.

The continuation creates children.

All the children finish before method `execute()` completes, so the continuation executes before the scheduler is done running `this`, which corrupts the scheduler.

Method `recycle_as_safe_continuation` avoids this race because the additional +1 in the *refcount* prevents the continuation from executing until the task completes.

10.3.4.3 void recycle_as_child_of(task& new_parent)

Requirements

Must be called while method `execute()` is running.

Effects

Causes `this` to become a child of *new_parent*, and not be destroyed when method `execute()` returns.

10.3.4.4 void recycle _to_reexecute()

Requirements

Only valid to call while method `execute()` is running. When method `execute()` returns, it must return a pointer to another task.

Effects

Causes `this` to be automatically spawned after `execute()` returns.

10.3.5 Synchronization

Spawning a task *task* either causes the calling thread to invoke *task*.`execute()`, or causes *task* to be put into the ready pool. Any thread participating in task scheduling may then acquire the task and invoke *task*.`execute()`. Section 10.1 describes the structure of the ready pool.

The calls that spawn come in two forms:

- Spawn a single `task`.
- Spawn multiple `task` objects specified by a `task_list` and clear `task_list`.

The calls distinguish between spawning root tasks and child tasks. A root task is one that was created using method `allocate_root`.

Important

A task should not spawn any child until it has called method `set_ref_count` to indicate both the number of children and whether it intends to use one of the "wait_for_all" methods.

10.3.5.1 void set_ref_count(int count)

Requirements

count≥0.[20] If the intent is to subsequently spawn *n* children and wait, then *count* should be *n*+1. Otherwise *count* should be *n.*

Effects

Sets the *refcount* attribute to *count*.

10.3.5.2 void increment_ref_count();

Effects

Atomically increments *refcount* attribute.

10.3.5.3 int decrement_ref_count();

Effects

Atomically decrements *refcount* attribute.

Returns

New value of *refcount* attribute.

NOTE: Explicit use of `increment_ref_count` and `decrement_ref_count` is typically necessary only when a task has more than one immediate successor task. Section 10.5 of the Tutorial ("General Acyclic Graphs of Tasks") explains more.

10.3.5.4 void wait_for_all()

Requirements

refcount=n+1, where *n* is the number of children who are still running.

---

[20] Intel® TBB 2.1 had the stronger requirement *count*>0.

Effects

Executes tasks in ready pool until *refcount* is 1. Afterwards sets *refcount* to 0. Figure 11 summarizes the state transitions.

Also, `wait_for_all()`automatically resets the cancellation state of the `task_group_context` implicitly associated with the task (10.6), when all of the following conditions hold:

- The task was allocated without specifying a context.
- The calling thread is a user-created thread, not an Intel® TBB worker thread.
- It is the outermost call to `wait_for_all()` by the thread.

TIP: Under such conditions there is no way to know afterwards if the `task_group_context` was cancelled. Use an explicit task_group_context if you need to know.

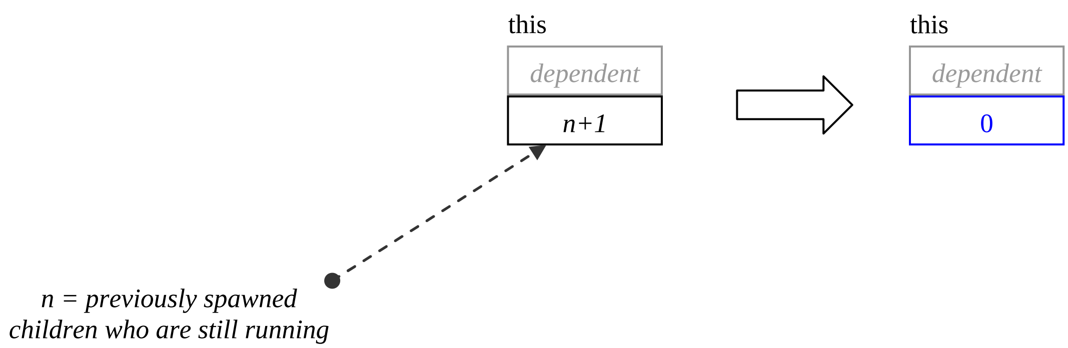

Figure 11: Effect of wait_for_all

10.3.5.5 void spawn(task& t)

Requirements

`this->refcount>0`

Effects

Puts task *t* into the ready pool and immediately returns. Task *t* does not have to be a child of `this`. A `task` may spawn itself.

The parent must call `set_ref_count` before spawning any child tasks, because once the child tasks are going, their completion will cause *refcount* to be decremented asynchronously. The debug version of the library detects when a required call to `set_ref_count` is not made, or is made too late.

10.3.5.6 void spawn (task_list& list)

Requirements

`this->refcount>0`

Effects

Equivalent to executing spawn on each task in *`list`* and clearing *`list`*, but more efficient. If *`list`* is empty, there is no effect.

10.3.5.7 void spawn_and_wait_for_all(task& t)

Requirements

Any other children of `this` must already be spawned. The `task` *`t`* must have a non-null attribute *parent*. There must be a chain of *parent* links from *`t`* to the calling `task`. Typically, this chain contains a single link. That is, *`t`* is typically a child of `this`.

Effects

Similar to `{spawn(`*task*`); wait_for_all();}`, but often more efficient. Furthermore, it guarantees that *task* is executed by the current thread. This constraint can sometimes simplify synchronization. Figure 12 illustrates the state transitions.

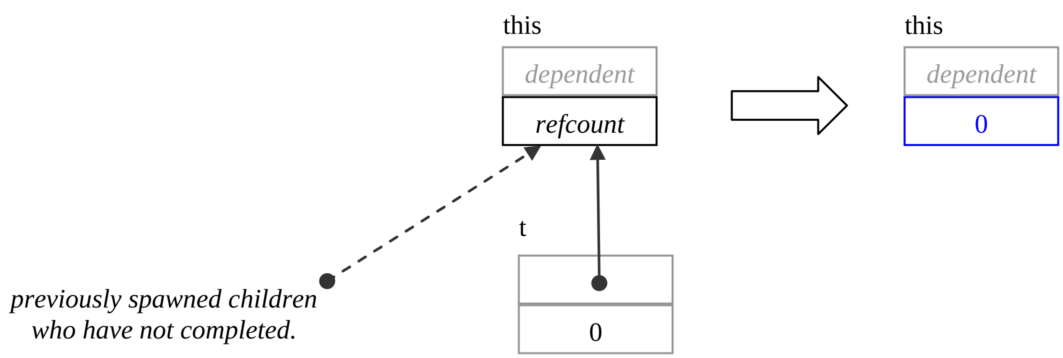

Figure 12: Effect of spawn_and_wait_for_all

10.3.5.8 void spawn_and_wait_for_all(task_list& list)

Effects

Similar to `{spawn(`*list*`); wait_for_all();}`, but often more efficient.

10.3.5.9 static void spawn_root_and_wait(task& root)

Requirements

The memory for task *root* was allocated by `task::allocate_root()`.

Effects

Sets *parent* attribute of *root* to an undefined value and execute *root* as described in Section 10.3.1.1. Destroys *root* afterwards unless *root* was recycled.

10.3.5.10 static void spawn_root_and_wait(task_list& root_list)

Requirements

Each `task` object *t* in *root_list* must meet the requirements in Section 10.3.5.9.

Effects

For each `task` object *t* in *root_list*, performs `spawn_root_and_wait(`*t*`)`, possibly in parallel. Section 10.3.5.9 describes the actions of `spawn_root_and_wait(`*t*`)`.

10.3.6 task Context

These methods expose relationships between `task` objects, and between `task` objects and the underlying physical threads.

10.3.6.1 static task& self()

Returns

Reference to innermost `task` that calling thread is running. A task is considered "running" if its methods `execute()`, `note_affinity()`, or destructor are running. If the calling thread is a user-created thread that is not running any task, `self()` returns a reference to an implicit dummy task associated with the thread.

10.3.6.2 task* parent() const

Returns

Value of the attribute *parent*. The result is an undefined value if the task was allocated by `allocate_root` and is currently running under control of `spawn_root_and_wait`.

10.3.6.3 bool is_stolen_task() const

Requirements

The attribute *parent* is not null and `this.execute()` is running. The calling task must not have been allocated with allocate_root.

Returns

true if task is running on a thread different than the thread that spawned it.

10.3.7 Cancellation

A `task` is a quantum of work that is cancelled or executes to completion. A cancelled task skips its method `execute()` if that method has not yet started. Otherwise cancellation has no direct effect on the task. A task can poll `task::is_cancelled()` to see if cancellation was requested after it started running.

Tasks are cancelled in groups as explained in Section 10.6.

10.3.7.1 bool cancel_group_execution()

Effects

Requests cancellation of all tasks in its group and its subordinate groups.

Returns

False if the task's group already received a cancellation request; true otherwise.

10.3.7.2 bool is_cancelled() const

Returns

True if task's group has received a cancellation request; false otherwise.

10.3.8 Affinity

These methods enable optimizing for cache affinity. They enable you to hint that a later task should run on the same thread as another task that was executed earlier. To do this:

1. In the earlier task, override `note_affinity(`*`id`*`)` with a definition that records *id*.
2. Before spawning the later task, run `set_affinity(`*`id`*`)` using the *`id`* recorded in step 1,

The *`id`* is a hint and may be ignored by the scheduler.

10.3.8.1 affinity_id

The type `task::affinity_id` is an implementation-defined unsigned integral type. A value of 0 indicates no affinity. Other values represent affinity to a particular thread. Do not assume anything about non-zero values. The mapping of non-zero values to threads is internal to the Intel® TBB implementation.

10.3.8.2 virtual void note_affinity (affinity_id id)

The task scheduler invokes `note_affinity` before invoking `execute()` when:

- The task has no affinity, but will execute on a thread different than the one that spawned it.
- The task has affinity, but will execute on a thread different than the one specified by the affinity.

You can override this method to record the id, so that it can be used as the argument to `set_affinity(id)` for a later task.

Effects

The default definition has no effect.

10.3.8.3 void set_affinity(affinity_id id)

Effects

Sets affinity of this task to *`id`*. The *`id`* should be either 0 or obtained from `note_affinity`.

10.3.8.4 affinity_id affinity() const

Returns

Affinity of this task as set by `set_affinity`.

10.3.9 task Debugging

Methods in this subsection are useful for debugging. They may change in future implementations.

10.3.9.1 state_type state() const

CAUTION: This method is intended for debugging only. Its behavior or performance may change in future implementations. The definition of `task::state_type` may change in future implementations. This information is being provided because it can be useful for diagnosing problems during debugging.

Returns

Current state of the task. Table 31 describes valid states. Any other value is the result of memory corruption, such as using a `task` whose memory has been deallocated.

Table 31: Values Returned by task::state()

Value	Description
allocated	Task is freshly allocated or recycled.
ready	Task is in ready pool, or is in process of being transferred to/from there.
executing	Task is running, and will be destroyed after method execute() returns.
freed	Task is on internal free list, or is in process of being transferred to/from there.
reexecute	Task is running, and will be respawned after method execute() returns.

Figure 13 summarizes possible state transitions for a `task`.

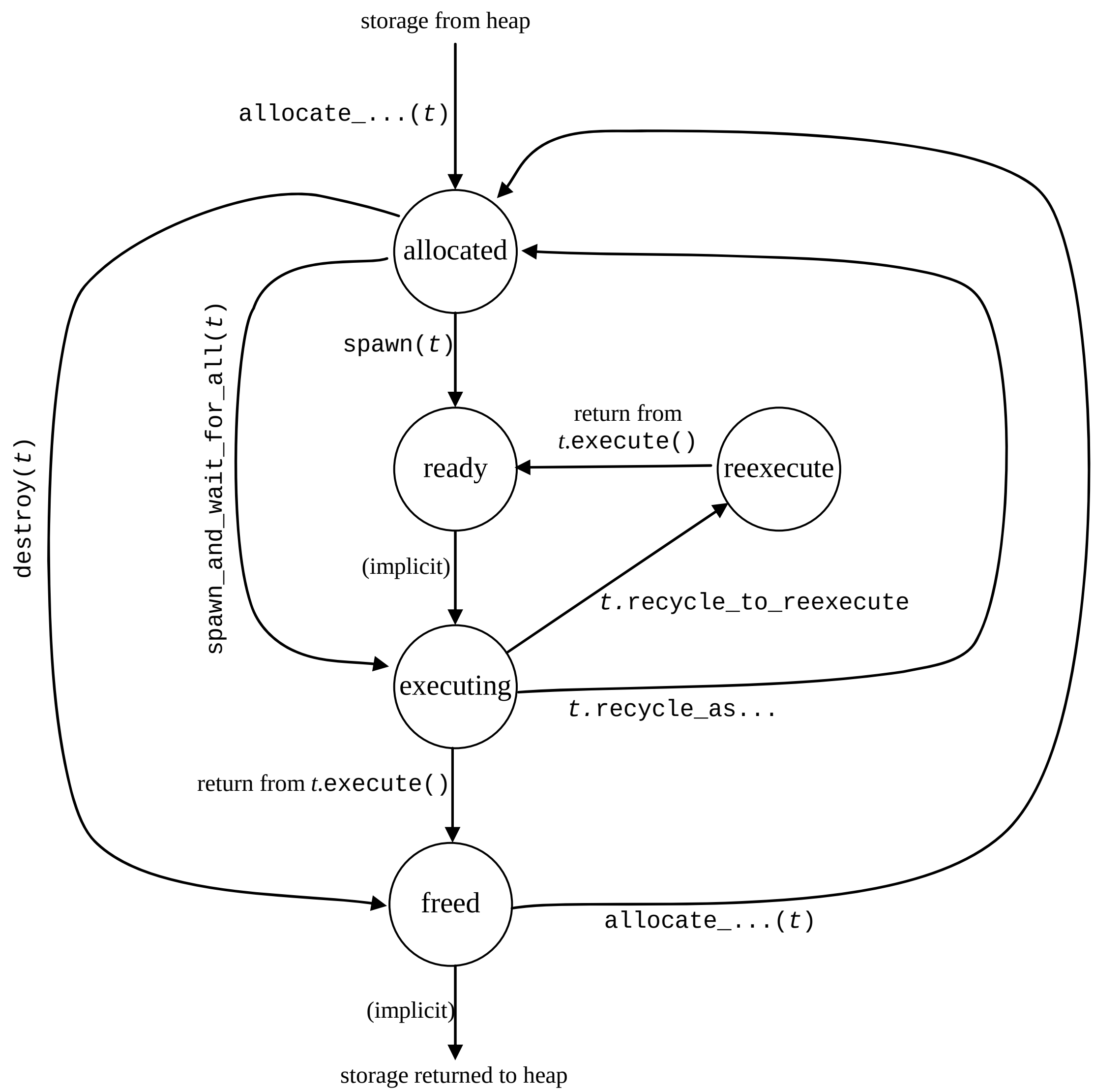

Figure 13: Typical task::state() Transitions

10.3.9.2 int ref_count() const

CAUTION: This method is intended for debugging only. Its behavior or performance may change in future implementations.

Returns

The value of the attribute *`refcount`*.

10.4 empty_task Class

Summary

Subclass of task that represents doing nothing.

Syntax

```
class empty_task;
```

Header

```
#include "tbb/task.h"
```

Description

An empty_task is a task that does nothing. It is useful as a continuation of a parent task when the continuation should do nothing except wait for its children to complete.

Members

```
namespace tbb {
    class empty_task: public task {
        /*override*/ task* execute() {return NULL;}
    };
}
```

10.5 task_list Class

Summary

List of task objects.

Syntax

```
class task_list;
```

Header

```
#include "tbb/task.h"
```

Description

A task_list is a list of references to task objects. The purpose of task_list is to allow a task to create a list of child tasks and spawn them all at once via the method task::spawn(task_list&), as described in 10.3.5.6.

A task can belong to at most one task_list at a time, and on that task_list at most once. A task that has been spawned, but not started running, must not belong to a task_list. A task_list cannot be copy-constructed or assigned.

Members

```
namespace tbb {
   class task_list {
    public:
        task_list();
        ~task_list();
        bool empty() const;
        void push_back( task& task );
        task& pop_front();
        void clear();
    };
}
```

10.5.1 task_list()

Effects

Constructs an empty list.

10.5.2 ~task_list()

Effects

Destroys the list. Does not destroy the task objects.

10.5.3 bool empty() const

Returns

True if list is empty; false otherwise.

10.5.4 push_back(task& task)

Effects

Inserts a reference to *task* at back of the list.

10.5.5 task& task pop_front()

Effects

Removes a task reference from front of list.

Returns

The reference that was removed.

10.5.6 void clear()

Effects

Removes all task references from the list. Does not destroy the task objects.

10.6 task_group_context

Summary

A cancellable group of tasks.

Syntax

```
class task_group_context;
```

Header

```
#include "tbb/task.h"
```

Description

A task_group_context represents a group of tasks that can be cancelled together. The task_group_context objects form a forest of trees. Each tree's root is a task_group_context constructed as isolated.

A task_group_context is cancelled explicitly by request, or implicitly when an exception is thrown out of a task. Cancelling a task_group_context causes the entire subtree rooted at it to be cancelled.

Each user thread that creates a task_scheduler_init (10.2) implicitly has an isolated task_group_context that acts as the root of its initial tree. This context is associated with the dummy task returned by task::self() when the user thread is not running any task (10.3.6.1).

Members

```
namespace tbb {
    class task_group_context {
```

```
    public:
        enum kind_t {
            isolated = implementation-defined,
            bound = implementation-defined
        };
        task_group_context( kind_t relation_to_parent = bound );
        ~task_group_context();
        void reset();
        bool cancel_group_execution();
        bool is_group_execution_cancelled() const;
    };
}
```

10.6.1 task_group_context(kind_t relation_to_parent=bound)

Effects

Constructs an empty `task_group_context`. If *`relation_to_parent`* is bound, the `task_group_context` becomes a child of the current innermost `task_group_context` and becomes the new innermost `task_group_context`. If *`relation_to_parent`* is `isolated`, it has no parent `task_group_context`.

10.6.2 ~task_group_context()

Effects

Destroys an empty task_group_context. It is a programmer error if there are still extant tasks in the group.

10.6.3 bool cancel_group_execution()

Effects

Requests that tasks in group be cancelled.

Returns

False if group is already cancelled; true otherwise. If concurrently called by multiple threads, exactly one call returns true and the rest return false.

10.6.4 bool is_group_execution_cancelled() const

Returns

True if group has received cancellation.

10.6.5 void reset()

Effects

Reinitializes this to uncancelled state.

CAUTION: This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

10.7 task_scheduler_observer

Summary

Class that represents thread's interest in task scheduling services.

Syntax

```
class task_scheduler_observer;
```

Header

```
#include "tbb/task_scheduler_observer.h"
```

Description

A `task_scheduler_observer` permits clients to observe when a thread starts or stops participating in task scheduling. You typically derive your own observer class from task_scheduler_observer, and override virtual methods `on_scheduler_entry` or `on_scheduler_exit`. An instance has a state *observing* or *not observing.* Remember to call `observe()` to enable observation.

Members

```
namespace tbb {
    class task_scheduler_observer {
    public:
        task_scheduler_observer();
        virtual ~task_scheduler_observer();
        void observe( bool state=true );
        bool is_observing() const;
```

```
        virtual void on_scheduler_entry( bool is_worker ) {}
        virtual void on_scheduler_exit( bool is_worker } {}
    };
}
```

10.7.1 task_scheduler_observer()

Effects

Constructs instance with observing disabled.

10.7.2 ~task_scheduler_observer()

Effects

Disables observing. Waits for extant invocations of on_scheduler_entry or on_scheduler_exit to complete.

10.7.3 void observe(bool state=true)

Effects

Enables observing if state is true; disables observing if state is false.

10.7.4 bool is_observing() const

Returns

True if observing is enabled; false otherwise.

10.7.5 virtual void on_scheduler_entry(bool is_worker)

Description

The task scheduler invokes this method on each thread that starts participating in task scheduling, if observing is enabled. If observing is enabled after threads started participating, then this method is invoked once for each such thread, before it executes the first task it steals afterwards.

The flag is_worker is true if the thread was created by the task scheduler; false otherwise.

NOTE: If a thread enables observing before spawning a task, it is guaranteed that the thread that executes the task will have invoked on_scheduler_entry before executing the task.

Effects

The default behavior does nothing.

10.7.6 virtual void on_scheduler_exit(bool is_worker)

Description

The task scheduler invokes this method when a thread stops participating in task scheduling, if observing is enabled.

CAUTION: Sometimes `on_scheduler_exit` is invoked for a thread but not `on_scheduler_entry`. This situation can arise if a thread never steals a task.

CAUTION: A process does not wait for Intel® TBB worker threads to clean up. Thus a process can terminate before `on_scheduler_exit` is invoked.

Effects

The default behavior does nothing.

10.8 Catalog of Recommended task Patterns

This section catalogues recommended task patterns. In each pattern, class T is assumed to derive from class `task`. Subtasks are labeled t_1, t_2, ... t_k. The subscripts indicate the order in which the subtasks execute if no parallelism is available. If parallelism is available, the subtask execution order is non-deterministic, except that t_1 is guaranteed to be executed by the spawning thread.

Recursive task patterns are recommended for efficient scalable parallelism, because they allow the task scheduler to unfold potential parallelism to match available parallelism. A recursive task pattern begins by creating a root task t_0 and running it as follows.

```
T& t0 = *new(allocate_root()) T(...);
task::spawn_root_and_wait(*t0);
```

The root task's method `execute()` recursively creates more tasks as described in subsequent subsections.

10.8.1 Blocking Style With *k* Children

The following shows the recommended style for a recursive task of type *T* where each level spawns *k* children.

```
task* T::execute() {
    if( not recursing any further ) {
        ...
```

```
    } else {
        set_ref_count(k+1);
        task& tk = *new(allocate_child()) T(...);  spawn(tk);
        task& tk-1= *new(allocate_child()) T(...);  spawn(tk-1);
        ...
        task& t1 = *new(allocate_child()) T(...);
        spawn_and_wait_for_all(t1);
    }
    return NULL;
}
```

Child construction and spawning may be reordered if convenient, as long as a task is constructed before it is spawned.

The key points of the pattern are:

- The call to `set_ref_count` uses *k*+1 as its argument. The extra 1 is critical.
- Each task is allocated by `allocate_child`.
- The call `spawn_and_wait_for_all` combines spawning and waiting. A more uniform but slightly less efficient alternative is to spawn all tasks with spawn and wait by calling `wait_for_all`.

10.8.2 Continuation-Passing Style With *k* Children

There are two recommended styles. They differ in whether it is more convenient to recycle the parent as the continuation or as a child. The decision should be based upon whether the continuation or child acts more like the parent.

Optionally, as shown in the following examples, the code can return a pointer to one of the children instead of spawning it. Doing so causes the child to execute immediately after the parent returns. This option often improves efficiency because it skips pointless overhead of putting the task into the task pool and taking it back out.

10.8.2.1 Recycling Parent as Continuation

This style is useful when the continuation needs to inherit much of the state of the parent and the child does not need the state. The continuation must have the same type as the parent.

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
    } else {
        set_ref_count(k);
        recycle_as_continuation();
        task& tk  = *new(allocate_child()) T(...); spawn(tk);
        task& tk–1 = *new(allocate_child()) T(...); spawn(tk–1);
        ...
```

```
        // Return pointer to first child instead of spawning it,
        // to remove unnecessary overhead.
        task& t1 = *new(allocate_child()) T(...);
        return &t1;
    }
}
```

The key points of the pattern are:

- The call to `set_ref_count` uses *k* as its argument. There is no extra +1 as there is in blocking style discussed in Section 10.8.1.
- Each child task is allocated by `allocate_child`.
- The continuation is recycled from the parent, and hence gets the parent's state without doing copy operations.

10.8.2.2 Recycling Parent as a Child

This style is useful when the child inherits much of its state from a parent and the continuation does not need the state of the parent. The child must have the same type as the parent. In the example, C is the type of the continuation, and must derive from class `task`. If C does nothing except wait for all children to complete, then C can be the class `empty_task` (10.4).

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
    } else {
        set_ref_count(k);
        // Construct continuation
        C& c = allocate_continuation();
        // Recycle self as first child
        task& tk  = *new(c.allocate_child()) T(...); spawn(tk);
        task& tk−1 = *new(c.allocate_child()) T(...); spawn(tk−1);
        ...
        task& t2 = *new(c.allocate_child()) T(...);  spawn(t2);
        // task t1 is our recycled self.
        recycle_as_child_of(c);
        update fields of *this to subproblem to be solved by t1
        return this;
    }
}
```

The key points of the pattern are:

- The call to `set_ref_count` uses *k* as its argument. There is no extra 1 as there is in blocking style discussed in Section 10.8.1.

- Each child task except for t_1 is allocated by `c.allocate_child`. It is critical to use `c.allocate_child`, and not `(*this).allocate_child`; otherwise the task graph will be wrong.
- Task t_1 is recycled from the parent, and hence gets the parent's state without performing copy operations. Do not forget to update the state to represent a child subproblem; otherwise infinite recursion will occur.

10.8.3 Letting Main Thread Work While Child Tasks Run

Sometimes it is desirable to have the main thread continue execution while child tasks are running. The following pattern does this by using a dummy `empty_task` (10.4).

```
task* dummy = new( task::allocate_root() ) empty_task;
dummy->set_ref_count(k+1);
task& tk = *new( dummy->allocate_child() ) T;  dummy->spawn(tk);
task& tk-1= *new( dummy->allocate_child() ) T;  dummy->spawn(tk-1);
...
task& t1 = *new( dummy->allocate_child() ) T;  dummy->spawn(t1);
...do any other work...
dummy->wait_for_all();
dummy->destroy(*dummy);
```

The key points of the pattern are:

- The dummy task is a placeholder and never runs.
- The call to `set_ref_count` uses *k*+1 as its argument.
- The dummy task must be explicitly destroyed.

11 Exceptions

Intel® Threading Building Blocks (Intel® TBB) propagates exceptions along logical paths in a tree of tasks. Because these paths cross between thread stacks, support for moving an exception between stacks is necessary.

When an exception is thrown out of a task, it is caught inside the Intel® TBB run-time and handled as follows:

1. If the cancellation group for the task has already been cancelled, the exception is ignored.
2. Otherwise the exception is captured as follows:
   a. If it is a tbb_exception *x*, it is captured by x.move()
   b. If it is a std::exception x, it is captured as a `captured_exception(typeid(x).name(),x.what())`.
   c. Otherwise it is captured as a captured exception with implementation-specified value for `name()` and `what()`.
3. The captured exception is rethrown from the root of the cancellation group after all tasks in the group have completed or have been successfully cancelled.

11.1 tbb_exception

Summary

Exception that can be moved to another thread.

Syntax

```
class tbb_exception;
```

Header

```
#include "tbb/tbb_exception.h"
```

Description

In a parallel environment, exceptions sometimes have to be propagated across threads. Class `tbb_exception` subclasses std::exception to add support for such propagation.

Members

```
namespace tbb {
```

```
    class tbb_exception: public std::exception {
        virtual tbb_exception* move() = 0;
        virtual void destroy() throw() = 0;
        virtual void throw_self() = 0;
        virtual const char* name() throw() = 0;
        virtual const char* what() throw() = 0;
    };
}
```

Derived classes should define the abstract virtual methods as follows:

- move() should create a pointer to a copy of the exception that can outlive the original. It may move the contents of the original.
- destroy() should destroy a copy created by move().
- throw_self() should throw *this.
- name() typically returns the RTTI name of the originally intercepted exception.
- what() returns a null-terminated string describing the exception.

11.2 captured_exception

Summary

Class used by Intel® TBB to propagate an exception that is not a tbb_exception.

Syntax

```
class captured_exception;
```

Header

```
#include "tbb/tbb_exception.h"
```

Description

When a task throws an exception and the exception is not a tbb_exception, Intel® TBB converts the exception to a captured_exception before propagating it. Conversion is necessary so that the exception can be propagated across threads.

Members

```
namespace tbb {
    class captured_exception: public tbb_exception {
        captured_exception(const captured_exception& src);
        captured_exception(const char* name, const char* info);
        ~captured_exception() throw();
        captured_exception& operator=(const captured_exception&);
        captured_exception* move() throw();
```

```
        void destroy() throw();
        void throw_self();
        const char* name() const throw();
        const char* what() const throw();
    };
}
```

Only the additions that captured_exception makes to tbb_exception are described here. Section 11.1 describes the rest of the interface.

11.2.1 captured_exception(const char* name, const char* info)

Effects

Constructs a captured_exception with the specified *name* and *info*.

11.3 movable_exception<ExceptionData>

Summary

Subclass of tbb_exception interface that supports propagating copy-constructible data.

Syntax

```
template<typename ExceptionData> class movable_exception;
```

Header

```
#include "tbb/tbb_exception.h"
```

Description

This template provides a convenient way to implement a subclass of tbb_exception that propagates arbitrary copy-constructible data.

Members

```
namespace tbb {
    template<typename ExceptionData>
    class movable_exception: public tbb_exception {
    public:
        movable_exception( const ExceptionData& src );
        movable_exception( const movable_exception& src )throw();
        ~movable_exception() throw();
        movable_exception& operator=( const movable_exception&
src );
```

```
        ExceptionData& data() throw();
        const ExceptionData& data() const throw();
        movable_exception* move() throw();
        void destroy() throw();
        void throw_self();
        const char* name() const throw();
        const char* what() const throw();
    };
}
```

Only the additions that movable_exception makes to tbb_exception are described here. Section 11.1 describes the rest of the interface.

11.3.1 movable_exception(const ExceptionData& src)

Effects

Construct movable_exception containing copy of src.

11.3.2 ExceptionData& data() throw()

Returns

Reference to contained data.

11.3.3 const ExceptionData& data() const throw()

Returns

Const reference to contained data.

11.4 missing_wait

Summary

Exception thrown by task_group and structured_task_group when call to wait() is missing.

Syntax

```
class missing_wait;
```

Header

```
#include "tbb/task_group.h"
```

Description

If an instance of `task_group` or `structured_task_group` is destroyed before method `wait()` is invoked, the instance throws the exception `missing_wait`.

Members

```
namespace tbb {
    class missing_wait: public std::exception {
    public:
        const char* what() const throw();
    };
}
```

12 Threads

Intel® Threading Building Blocks (Intel® TBB) provides a wrapper around the platform's native threads, based upon proposal N2497 for C++ 200x. Using this wrapper has two benefits:

- It makes threaded code portable across platforms.
- It eases later migration to ISO C++ 200x threads.

The significant departures from N2497 are shown in Table 32.

Table 32: Differences Between N2497 and Intel® TBB Thread Class

N2497	TBB
std::thread	tbb::tbb_thread
std::this_thread	tbb::tbb_this_thread
std::this_thread::sleep (system_time);	tbb::tbb_this_thread::sleep(tick_count ::interval_t)
rvalue reference parameters	Parameter changed to plain value, or function removed, as appropriate.
constructor for std::thread takes arbitrary number of arguments.	constructor for tbb::tbb_thread takes 0-3 arguments.

The name changes prevent identifier collisions when using directives are employed. The other changes are for compatibility with the current C++ standard or Intel® TBB. For example, constructors that have an arbitrary number of arguments require the variadic template features of C++ 200x.

CAUTION: Threads are heavy weight on most systems, and running too many threads on a system can seriously degrade performance. Consider using a task based solution instead if practical.

12.1 tbb_thread Class

Summary

Represents a thread of execution.

Syntax

```
class tbb_thread;
```

Header

```
#include "tbb/tbb_thread.h"
```

Description

Class tbb_thread provides a platform independent interface to native threads. An instance represents a thread. A platform-specific thread handle can be obtained via method native_handle().

Members

```
namespace tbb {
    class tbb_thread {
    public:
#if _WIN32||_WIN64
        typedef HANDLE native_handle_type;
#else
        typedef pthread_t native_handle_type;
#endif // _WIN32||_WIN64

        class id;

        tbb_thread();
        template <typename F> explicit tbb_thread(F f);
        template <typename F, typename X> tbb_thread(F f, X x);
        template <typename F, typename X, typename Y>
            tbb_thread (F f, X x, Y y);
        tbb_thread& operator=(tbb_thread& x);
        ~tbb_thread();

        bool joinable() const;
        void join();
        void detach();
        id get_id() const;
        native_handle_type native_handle();

        static unsigned hardware_concurrency();
    };
}
```

12.1.1 tbb_thread()

Effects

Constructs tbb_thread that does not represent a thread of execution, with get_id()==id().

12.1.2 template<typename F> tbb_thread(F f)

Effects

Construct tbb_thread that evaluates f()

12.1.3 template<typename F, typename X> tbb_thread(F f, X x)

Effects

Constructs tbb_thread that evaluates f(x).

12.1.4 template<typename F, typename X, typename Y> tbb_thread(F f, X x, Y y)

Effects

Constructs tbb_thread that evaluates f(x,y).

12.1.5 tbb_thread& operator=(tbb_thread& x)

Effects

If `joinable()`, calls `detach()`. Then assigns the state of *x* to `*this` and sets *x* to default constructed state.

CAUTION: Assignment moves the state instead of copying it.

12.1.6 ~tbb_thread

Effects

`if( joinable() ) detach()`.

12.1.7 bool joinable() const

Returns

```
get_id()!=id()
```

12.1.8 void join()

Requirements

`joinable()==true`

Effects

Wait for thread to complete. Afterwards, `joinable()==false`.

12.1.9 void detach()

Requirements

`joinable()==true`

Effects

Sets *this to default constructed state and returns without blocking. The thread represented by *this continues execution.

12.1.10 id get_id() const

Returns

id of the thread, or a default-constructed id if *this does not represent a thread.

12.1.11 native_handle_type native_handle()

Returns

Native thread handle. The handle is a `HANDLE` on Windows* operating systems and a `pthread_t` on Linux* and Mac OS* X operating systems. For these systems, `native_handle()` returns 0 if `joinable()==false`.

12.1.12 static unsigned hardware_concurrency()

Returns

The number of hardware threads. For example, 4 on a system with a single Intel® Core™2 Quad processor.

12.2 tbb_thread:: id

Summary

Unique identifier for a thread.

Syntax

```
class tbb_thread::id;
```

Header

```
#include "tbb/tbb_thread.h"
```

Description

A `tbb_thread::id` is an identifier value for a thread that remains unique over the thread's lifetime. A special value tbb_thread::id() represents no thread of execution. The instances are totally ordered.

Members

```
namespace tbb {
    class tbb_thread::id {
    public:
        id();
    };
    template<typename charT, typename traits>
    std::basic_ostream<charT, traits>&
        operator<< (std::basic_ostream<charT, traits> &out,
                    tbb_thread::id id)

    bool operator==(tbb_thread::id x, tbb_thread::id y);
    bool operator!=(tbb_thread::id x, tbb_thread::id y);
    bool operator<(tbb_thread::id x, tbb_thread::id y);
    bool operator<=(tbb_thread::id x, tbb_thread::id y);
    bool operator>(tbb_thread::id x, tbb_thread::id y);
    bool operator>=(tbb_thread::id x, tbb_thread::id y);
} // namespace tbb
```

12.3 this_tbb_thread Namespace

Description

Namespace this_tbb_thread contains global functions related to threading.

Members

```
namepace tbb {
    namespace this_tbb_thread {
        tbb_thread::id get_id();
        void yield();
        void sleep( const tick_count::interval_t );
    }
}
```

12.3.1 tbb_thread::id get_id()

Returns

Id of the current thread.

12.3.2 void yield()

Effects

Offers to suspend current thread so that another thread may run.

12.3.3 void sleep(const tick_count::interval_t & i)

Effects

Current thread blocks for at least time interval *i*.

Example

```
using namespace tbb;

void Foo() {
    // Sleep 30 seconds
    this_tbb_thread::sleep( tick_count::interval_t(30) );
}
```

13 References

Umut A. Acar, Guy E. Blelloch, Robert D. Blumofe, The Data Locality of Work Stealing. *ACM Symposium on Parallel Algorithms and Architectures* (2000):1-12.

Robert D.Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (July 1995):207–216.

H. Hinnant, L. Crowl, Beman Dawes, A. WIlliams, J. Garland, Multi-threading Library for Standard C++ (Revision 1), ISO/IEC JTC1 SC22 WG21 N2497.

Steve MacDonald, Duane Szafron, and Jonathan Schaeffer. Rethinking the Pipeline as Object-Oriented States with Transformations. *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (April 2004):12-21.

V. Kumar and V. N. Rao, "Parallel Depth First Search. Part II. Analysis", *International Journal of Parallel Programming* (December 1987): 501-519.

ISO/IEC 14882, *Programming Languages* – C++

Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence 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.

S. G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts", *IEEE Transactions on Computers*, Vol. C-36 No. 11, Nov. 1987.

Appendix A Compatibility Features

This appendix describes features of Intel Threading Building Blocks (Intel® TBB) that remain for compatibility with previous versions. These features are deprecated and may disappear in future versions of Intel® TBB. Some of these features are available only if the preprocessor symbol TBB_DEPRECATED is non-zero.

A.1 parallel_while Template Class

Summary

Template class that processes work items.

TIP: This class is deprecated. Use parallel_do (3.7) instead.

Syntax

```
template<typename Body>
class parallel_while;
```

Header

```
#include "tbb/parallel_while.h"
```

Description

A parallel_while<Body> performs parallel iteration over items. The processing to be performed on each item is defined by a function object of type Body. The items are specified in two ways:

- A stream of items.
- Additional items that are added while the stream is being processed.

Table 33 shows the requirements on the stream and body.

Table 33: parallel_while Requirements for Stream S and Body B

Pseudo-Signature	Semantics
`bool S::pop_if_present( B::argument_type& item )`	Get next stream item. parallel_while does not concurrently invoke the method on the same this.
`B::operator()( B::argument_type& item ) const`	Process item. parallel_while may concurrently invoke the operator for the same this but

Pseudo-Signature	Semantics
	different item.
`B::argument_type()`	Default constructor.
`B::argument_type( const B::argument_type& )`	Copy constructor.
`~B::argument_type()`	Destructor.

For example, a unary function object, as defined in Section 20.3 of the C++ standard, models the requirements for B. A concurrent_queue (4.3) models the requirements for S.

TIP: To achieve speedup, the grainsize of `B::operator()` needs to be on the order of at least ~10,000 instructions. Otherwise, the internal overheads of `parallel_while` swamp the useful work. The parallelism in `parallel_while` is not scalable if all the items come from the input stream. To achieve scaling, design your algorithm such that method add often adds more than one piece of work.

Members

```
namespace tbb {
    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 );
    };
}
```

A.1.1 parallel_while<Body>()

Effects

Constructs a `parallel_while` that is not yet running.

A.1.2 ~parallel_while<Body>()

Effects

Destroys a `parallel_while`.

A.1.3 Template <typename Stream> void run(Stream& stream, const Body& body)

Effects

Applies *body* to each item in *stream* and any other items that are added by method add. Terminates when both of the following conditions become true:

- *`stream`*`.pop_if_present` returned false.
- *body*`(x)` returned for all items *x* generated from the stream or method add.

A.1.4 void add(const value_type& item)

Requirements

Must be called from a call to *body*`.operator()` created by `parallel_while`. Otherwise, the termination semantics of method run are undefined.

Effects

Adds item to collection of items to be processed.

A.2 Interface for constructing a pipeline filter

The interface for constructing a filter evolved over several releases of Intel® TBB. The two following subsections describe obsolete aspects of the interface.

A.2.1 filter::filter(bool is_serial)

Effects

Constructs a serial in order filter if `is_serial` is true, or a parallel filter if `is_serial` is false. This deprecated constructor is superseded by the constructor `filter( filter::mode )` described in Section 3.9.6.1.

A.2.2 filter::serial

The filter mode value filter::serial is now named filter::serial_in_order. The new name distinguishes it more clearly from the mode filter::serial_out_of_order.

A.3 Debugging Macros

The names of the debugging macros have changed as shown in Table 34. If you define the old macros, Intel® TBB sets each undefined new macro in a way that duplicates the behavior the old macro settings.

The old TBB_DO_ASSERT enabled assertions, full support for Intel® Threading Tools, and performance warnings. These three distinct capabilities are now controlled by three separate macros as described in Section 2.6.

TIP: To enable all three capabilities with a single macro, define TBB_USE_DEBUG to be 1. If you had code under "#if TBB_DO_ASSERT" that should be conditionally included only when assertions are enabled, use "#if TBB_USE_ASSERT" instead.

Table 34: Deprecated Macros

Deprecated Macro	New Macro
TBB_DO_ASSERT	TBB_USE_DEBUG or TBB_USE_ASSERT, depending on context.
TBB_DO_THREADING_TOOLS	TBB_USE_THREADING_TOOLS

A.4 tbb::deprecated::concurrent_queue<T,Alloc> Template Class

Summary

Template class for queue with concurrent operations. This is the concurrent_queue supported in Intel® TBB 2.1 and prior. New code should use the Intel® TBB 2.2 unbounded concurrent_queue or concurrent_bounded_queue.

Syntax

```
template<typename T, typename Alloc=cache_aligned_allocator<T> >
class concurrent_queue;
```

Header

```
#include "tbb/concurrent_queue.h"
```

Description

A tbb::deprecated::concurrent_queue is a bounded first-in first-out data structure that permits multiple threads to concurrently push and pop items. The default bounds are large enough to make the queue practically unbounded, subject to memory limitations on the target machine.

NOTE: Compile with TBB_DEPRECATED=1 to inject tbb::deprecated::concurrent_queue into namespace tbb. Consider eventually migrating to the new queue classes.

- Use the new tbb::concurrent_queue if you need only the non-blocking operations (push and try_pop) for modifying the queue.
- Otherwise use the new tbb::concurrent_bounded_queue. It supports both blocking operations (push and try_pop) and non-blocking operations.

In both cases, use the new method names in Table 35.

Table 35: Method Name Changes for Concurrent Queues

Method in tbb::deprecated::concurrent_queue	Equivalent method in tbb::concurrent_queue or tbb::concurrent_bounded_queue
pop_if_present	try_pop
push_if_not_full	try_push (not available in tbb::concurrent_queue)
begin	unsafe_begin
end	unsafe_end

Members

```
namespace tbb {
  namespace deprecated {
    template<typename T,
             typename Alloc=cache_aligned_allocator<T> >
    class concurrent_queue {
    public:
        // types
        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef std::ptrdiff_t size_type;
        typedef std::ptrdiff_t difference_type;

        concurrent_queue(const Alloc& a = Alloc());
        concurrent_queue(const concurrent_queue& src,
                         const Alloc& a = Alloc());
        template<typename InputIterator>
        concurrent_queue(InputIterator first, InputIterator last,
```

```
                         const Alloc& a = Alloc());
        ~concurrent_queue();

        void push(const T& source);
        bool push_if_not_full(const T& source);
        void pop(T& destination);
        bool pop_if_present(T& destination);
        void clear() ;

        size_type size() const;
        bool empty() const;
        size_t capacity() const;
        void set_capacity(size_type capacity);
        Alloc get_allocator() const;

        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;

        // iterators (these are slow and intended only for
debugging)
        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
    };
  }
#if TBB_DEPRECATED
  using deprecated::concurrent_queue;
#else
  using strict_ppl::concurrent_queue;
#endif
}
```

A.5 Interface for concurrent_vector

The return type of methods grow_by and grow_to_at_least changed in Intel® TBB 2.2. Compile with the preprocessor symbol TBB_DEPRECATED set to nonzero to get the old methods.

Table 36: Change in Return Types

Method	Deprecated Return Type	New Return Type
grow_by (4.5.3.1)	size_type	iterator

grow_to_at_least (4.5.3.2)	void	iterator
push_back (4.5.3.3)	size_type	iterator

A.5.1 void compact()

Effects

Same as shrink_to_fit() (4.5.2.2).

A.6 Depth interface for class task

Intel® TBB 2.2 eliminates the notion of task depth that was present in prior versions of Intel® TBB. The members of class task that related to depth have been retained under TBB_DEPRECATED, but do nothing.

Deprecated Members of class task

```
namespace tbb {
    class task {
        ...
#if TBB_DEPRECATED
        // task depth
        typedef implementation-defined-signed-integral-type
depth_type;
        depth_type depth() const {return 0;}
        void set_depth( depth_type new_depth ) {}
        void add_to_depth( int delta ){}
#endif
        ...
    };
}
```

Appendix B PPL Compatibility

Intel Threading Building Blocks (Intel® TBB) 2.2 introduces features based on joint discussions between the Microsoft Corporation and Intel Corporation. The features establish some degree of compatibility between Intel® TBB and Microsoft Parallel Patterns Library (PPL) development software.

Table 37 lists the features. Each feature appears in namespace `tbb`. Each feature can be injected into namespace `Concurrency` by including the file "`tbb/compat/ppl.h`"

Table 37: PPL Compatibility Features

Section	Feature
3.4	`parallel_for(`*`first,last,step,f`*`)`
3.8	`parallel_for_each`
3.11	`parallel_invoke`
9.3	`task_handle`
9.2	`task_group_status`
9.1.1	`task_group`
9.5	`structured_task_group`
9.6	`is_current_task_group_cancelling`
11.4	`missing_wait`

For `parallel_for`, only the signature that takes `(`*`first,last,step,f`*`)` is injected into namespace `Concurrency`.

CAUTION: Because of different environments and evolving specifications, the behavior of the features can differ between the Intel® TBB and PPL implementations.

Appendix C Known Issues

This section explains known issues with using Intel® Threading Building Blocks (Intel® TBB).

C.1 Windows* OS

Some Intel® TBB header files necessarily include the header file `<windows.h>`, which by default defines the macros `min` and `max`, and consequently breaks the ISO C++ header files `<limits>` and `<algorithm>`. Defining the preprocessor symbol `NOMINMAX` causes `<windows.h>` to not define the offending macros. Thus programs using Intel® TBB and either of the aforementioned ISO C++ headers should be compiled with `/DNOMINMAX` as a compiler argument.