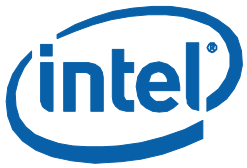

Intel(R) Threading Building Blocks

Reference Manual

Document Number 315415-016US.

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

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Revision History

Document Number	Revision Number	Description	Revision Date
315415-016	1.29	Added multifunction_node and split_node as full features of the flow graph. Made changes to method names and typedefs in tuple-based flow graph nodes. Modified flow graph nodes so that all nodes now receive a reference to the flow graph in their constructors. Added concurrent_lru_cache as a community preview feature. Made other small additions and corrections.	2012-Jan-20
315415-015	1.28	Updated concurrent_bounded_queue to add abort() method (Section 5.6 and 13.4)	2011-Nov-21
315415-014	1.27	Updated the Optimization Notice.	2011-Oct-27
315415-013	1.26	Moved the flow graph from Appendix D to Section 6 and made a number of updates as it bcomes a fully supported feature. Moved concurrent_priority_queue from Appendix D to Section 5.7 as it becomes fully supported. Added serial subset, memory pools, and parallel_deterministic_reduce to Appendix D. Made other small corrections and additions.	2011-Aug-01
315415-012	1.25	Moved task and task_group priorities from Appendix D to Section 111.3.8 and 11.6. Updated concurrent_priority_queue documentation in Section D.1 to reflect interface changes. Updated flow graph documentation in D.2 to reflect changes in the interface. Added run-time loader documentation as Section D.3.	2011-July-01
315415-011	1.24	Fix incorrect cross-reference to Tutorial in Section 11.3.5.3. Clarify left to right properties of `parallel_reduce`. Add `task_group_context` syntax and description to parallel algorithms as needed. Add `group` and `change_group` method to task. Update description of `task_group`. Add `task` and `task_group` priorities to Community Preview Features as D.3. Add two examples to D.2 and describe body objects. Update `overwrite_node`, `write_once_node` and `join_node`.	2011-Feb-24

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. The *Intel® Threading Building Blocks Design Patterns* document is another useful resource.

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`:

[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.

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`, 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 the library's namespace conventions.

2.4.1 tbb Namespace

Namespace `tbb` contains public identifiers defined by the library that you can reference in your program.

2.4.2 tb::flow Namespace

Namespace `tbb::flow` contains public identifiers related to the flow graph Community Preview Feature. See Appendix D.1 for more information.

2.4.3 tbb::interface*x* Namespace

Namespaces of the form `tbb::interface`*x* define public identifiers that the library injects into namespace `tbb`. The numeral *x* corresponds to an internal version number that serves to prevent accidental linkage of incompatible definitions. Your code should never directly reference namespaces prefixed with `tbb::interface`*x*. Instead, reference names via namespace `tbb`.

For example the header `tbb/concurrent_hash_map.h` defines the template `concurrent_hashmap<Key,T>` as `tbb::version4::concurrent_hashmap<Key,T>` and employs a `using` directive to inject it into namespace `tbb`. Your source code should reference it as `tbb::concurrent_hashmap<Key,T>`.

2.4.4 tbb::internal Namespace

Namespace `tbb::internal` serves a role similar to `tbb::interface`*x*. It is retained for backwards compatibility with older versions of the library. Your code should never

directly reference namespace `tbb::internal`. Indirect reference via a public `typedef` provided by the header files is permitted.

2.4.5 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`.

2.4.6 tbb::strict_ppl Namespace

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

2.4.7 std Namespace

The library implements some C++0x features in namespace `std`. The library version can be used by including the corresponding header in Table 3.

Table 3: C++0x Features Optonally Defined by Intel® Threading Building Blocks.

Header	Identifiers Added to `std::`	Section

tbb/compat/condition_variable	`defer_lock_t`
	`try_to_lock_t`
	`adopt_lock_t`
	`defer_lock`
	`try_to_lock`
	`adopt_lock`
	`lock_guard`
	`unique_lock`
	`swap`[2]
	`condition_variable`
	`cv_status`
	`timeout`
	`no_timeout`
		9.4

tbb/compat/thread	`thread`	14.1

[2] Adds swap of two `unique_lock` objects, not the general `swap` template function.

	this_thread	

To prevent accidental linkage with other implementations of these C++ library features, the library defines the identifiers in other namespaces and injects them into namespace `std::`. This way the "mangled name" seen by the linker will differ from the "mangled name" generated by other implementations.

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.

3 Environment

This section describes features of Intel® Threadinging Building Blocks (Intel® TB) that relate to general environment issues.

3.1 Version Information

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

3.1.1 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; that is, the major version number.
`TBB_COMPATIBLE_INTERFACE_VERSION`	Oldest major interface version still supported.

3.1.2 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`	Intel® 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.

3.1.3 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.

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.

3.2 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 6 summarizes the macros and their default values. A value of 1 enables the corresponding feature; a value of 0 disables the feature.

Table 6: Debugging Macros

Macro	Default Value	Feature
TBB_USE_DEBUG	Windows* OS: 1 if _DEBUG is defined, 0 otherwise.	Default value for all other macros in this table.
	All other systems: 0.	
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.

3.2.1 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 Microsoft Windows* operating systems, debug builds implicitly set `TBB_USE_ASSERT` to `1` by default

3.2.2 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.

3.2.3 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.

3.3 Feature macros

Macros in this section control optional features in the library.

3.3.1 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.

3.3.2 TBB_USE_EXCEPTIONS macro

The macro `TBB_USE_EXCEPTIONS` controls whether the library headers use exception-handling constructs such as `try`, `catch`, and `throw`. The headers do not use these constructs when `TBB_USE_EXCEPTIONS=0`.

For the Microsoft Windows*, Linux*, and MacOS* operating systems, the default value is 1 if exception handling constructs are enabled in the compiler, and 0 otherwise.

CAUTION: The runtime library may still throw an exception when `TBB_USE_EXCEPTIONS=0`.

3.3.3 TBB_USE_CAPTURED_EXCEPTION macro

The macro `TBB_USE_CAPTURED_EXCEPTION` controls rethrow of exceptions within the library. Because C++ 1998 does not support catching an exception on one thread and

rethrowing it on another thread, the library sometimes resorts to rethrowing an approximation called `tbb::captured_exception`.

- Define `TBB_USE_CAPTURED_EXCEPTION=1` to make the library rethrow an approximation. This is useful for uniform behavior across platforms.
- Define `TBB_USE_CAPTURED_EXCEPTION=0` to request rethrow of the exact exception. This setting is valid only on platforms that support the `std::exception_ptr` feature of C++ 200x. Otherwise a compile-time diagnostic is issued.

The default value is 1 for supported host compilers with `std::exception_ptr`, and 0 otherwise.

Section 13 describes exception handling and `TBB_USE_CAPTURED_EXCEPTION` in more detail.

4 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.

4.1 Splittable Concept

Summary

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

Requirements

Table 7 lists the requirements for a splittable type `X` with instance `x`.

Table 7: 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` (4.2.1) and `blocked_range2d` (4.2.2) represent splittable ranges. For each of these, splitting partitions the range into two subranges. See the example in Section 4.2.1.3 for the splitting constructor of `blocked_range<Value>`.

The bodies for `parallel_reduce` (4.5) and `parallel_scan` (4.6) must be splittable. For each of these, splitting results in two bodies that can be run concurrently.

4.1.1 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 {
    };
}
```

4.2 Range Concept

Summary

Requirements for type representing a recursively divisible set of values.

Requirements

Table 8 lists the requirements for a Range type `R`.

Table 8: Range Concept

Pseudo-Signature	Semantics
`R::R( const R& )`	Copy constructor.
`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` (4.2.1) 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` (4.4), `parallel_reduce` (4.5), and `parallel_scan` (4.6) 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` (4.2.1) models a one-dimensional range.

Type `blocked_range2d` (4.2.2) models a two-dimensional range.

Type `blocked_range3d` (4.2.3) models a three-dimensional range.

Concept Container Range (5.1) models a container as a range.

4.2.1 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 9. In the table, type D is the type of the expression "j-i". It can be any integral type that is convertible to `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 (4.2).

Table 9: Value Concept for blocked_range

Pseudo-Signature	Semantics
`Value::Value( const Value& )`	Copy constructor.
`Value::~Value()`	Destructor.

`void`[3] `operator=( const Value& )`	Assignment
`bool operator<( const Value& i, const Value& j )`	Value *i* precedes value j.
`D operator-( const Value& i, const Value& j )`	Number of values in range [`i`,`j`).
`Value operator+( const Value& i, D 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 (4.4), parallel_reduce (4.5), and parallel_scan (4.6) 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` (3.2.1) 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` (4.4), `parallel_reduce` (4.5), and `parallel_scan` (4.6).

---

[3]The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored by `blocked_range`.

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;
    };
}
```

4.2.1.1 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 9. 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.

4.2.1.2 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`.

4.2.1.3 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`.

4.2.1.4 size_type size() const

Requirements

`end()<begin()` is false.

Effects

Determines size of range.

Returns

`end()-begin()`

4.2.1.5 bool empty() const

Effects

Determines if range is empty.

Returns

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

4.2.1.6 size_type grainsize() const

Returns

Grain size of range.

4.2.1.7 bool is_divisible() const

Requirements

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

Effects

Determines if range can be split into subranges.

Returns

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

4.2.1.8 const_iterator begin() const

Returns

Inclusive lower bound on range.

4.2.1.9 const_iterator end() const

Returns

Exclusive upper bound on range.

4.2.2 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 9. A `blocked_range` is splittable if either axis is splittable. A `blocked_range` models the Range concept (4.2).

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.

4.2.2.1 row_range_type

Description

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

4.2.2.2 col_range_type

Description

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

4.2.2.3 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.

4.2.2.4 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)`.

4.2.2.5 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.

4.2.2.6 bool empty() const

Effects

Determines if range is empty.

Returns

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

4.2.2.7 bool is_divisible() const

Effects

Determines if range can be split into subranges.

Returns

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

4.2.2.8 const row_range_type& rows() const

Returns

Range containing the rows of the value space.

4.2.2.9 const col_range_type& cols() const

Returns

Range containing the columns of the value space.

4.2.3 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_range3d {
    public:
        // Types
        typedef blocked_range<PageValue> 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_range3d& r, split );

        // Capacity
        bool empty() const;

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

4.3 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` (4.4), `parallel_reduce` (4.5), and `parallel_scan` (4.6) 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 10. 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 10: Partitioners

Partitioner	Loop Behavior
`const auto_partitioner& (default)`[4]	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 4.2.1).

4.3.1 auto_partitioner Class

Summary

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

Syntax

```
class auto_partitioner;
```

[4] 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`.

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` (12.2.1). 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();
    }
}
```

4.3.1.1 auto_partitioner()

Construct an `auto_partitioner`.

4.3.1.2 ~auto_partitioner()

Destroy this `auto_partitioner`.

4.3.2 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.

4.3.2.1 affinity_partitioner()

Construct an `affinity_partitioner`.

4.3.2.2 ~affinity_partitioner()

Destroy this `affinity_partitioner`.

4.3.3 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();
    }
}
```

4.3.3.1 simple_partitioner()

Construct a `simple_partitioner`.

4.3.3.2 ~simple_partitioner()

Destroy this `simple_partitioner`.

4.4 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
                   [, task_group_context& group] );

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

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

where the optional *partitioner* declares any of the partitioners as shown in column 1 of Table 10.

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 (4.2). The body must model the requirements in Table 11.

Table 11: 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 (12.2), `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.

All overloads can be passed a `task_group_context` object so that the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

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≤`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.

4.5 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[, task_group_context& group]] );

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

where the optional *partitioner* declares any of the partitioners as shown in column 1 of Table 10.

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 12 summarizes the type requirements on the types of *identity, func*, and *reduction.*

Table 12: Requirements for Func and Reduction

Pseudo-Signature	Semantics
`Value Identity;`	Left 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 (4.2). The body must model the requirements in Table 13.

Table 13: Requirements for parallel_reduce Body

Pseudo-Signature	Semantics
`Body::Body( Body&, split );`	Splitting constructor (4.1). 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 (12.2.1), `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.

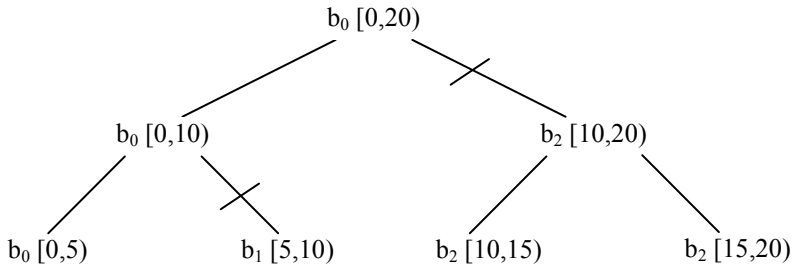

Figure 1: Execution of parallel_reduce over blocked_range<int>(0,20,5)

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 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 left to right property for a given instance of a body. However, you t must neither rely on a particular choice of body splitting nor on the subranges processed by a given body object being consecutive. `parallel_reduce` makes the choice of body splitting nondeterministically.

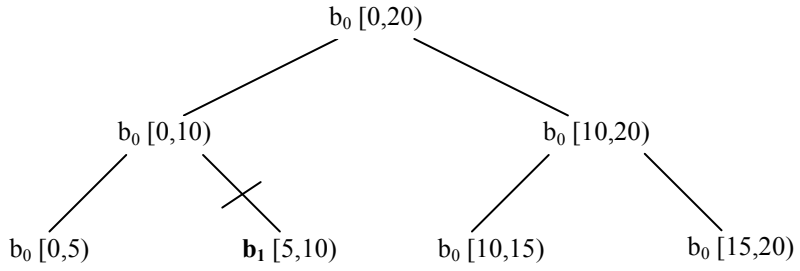

Figure 2: Example Where Body b_0 Processes Non-consecutive Subranges.

The subranges evaluated by a given body are not consecutive if there is an intervening `join`. The joined information represents processing of a gap between evaluated subranges. Figure 2 shows such an example. The body b_0 performs the following sequence of operations:

b_0([0,5))

b_0.join()(b_1) where b_1 has already processed [5,10)

b_0([10,15))

b_0([15,20))

In other words, body b_0 gathers information about all the leaf subranges in left to right order, either by directly processing each leaf, or by a join operation on a body that gathered information about one or more leaves in a similar way. When no worker threads are available, `parallel_reduce` executes sequentially from left to right in the same sense as for `parallel_for` (4.4). Sequential execution never invokes the splitting constructor or method `join`.

All overloads can be passed a `task_group_context` object so that the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

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>()
    );
}
```

4.6 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 $\oplus$ and using two passes. It may invoke $\oplus$ 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 14.

Table 14: 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.

Pseudo-Signature	Semantics
`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 3 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.

7. 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.
8. 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.
9. The pink body computes its summary for 1-8 by performing a reverse_join with the blue body.
10. The yellow body computes its summary for 1-12 by performing a reverse_join with the pink body.
11. The blue, pink, and yellow bodies compute final scans and summaries for portions of the array.
12. 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.

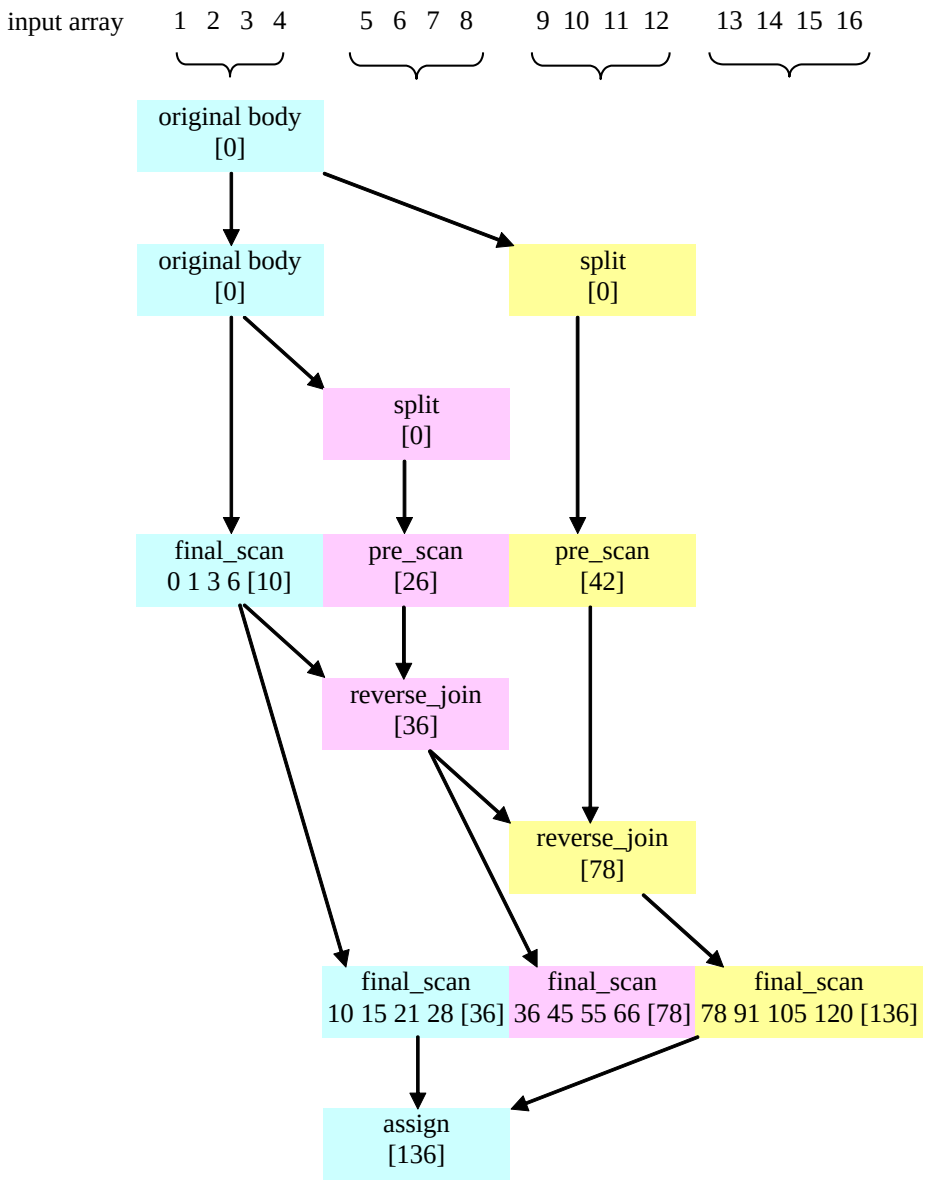

Figure 3: Example Execution of parallel_scan

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() );
```

4.6.1 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 4.6 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();
    };

}
```

4.6.1.1 bool is_final_scan()

Returns

True for a `final_scan_tag`, otherwise false.

4.7 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[, task_group_context& group] );
```

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` (4.7.1). 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` (4.7.1.1).

The requirements for input iterators are specified in Section 24.1 of the ISO C++ standard. Table 15 shows the requirements on type `Body`.

Table 15: 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 ~100,000 clock cycles. Otherwise, the internal overheads of `parallel_do` swamp the useful work.

The algorithm can be passed a `task_group_context` object so that its tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

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);
        }
    }
};
```

4.7.1 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` (4.7) 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 );
    };
}
```

4.7.1.1 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.

4.8 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
                        [, task_group_context& group]);
```

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.

If the `group` argument is specified, the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

4.9 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` (4.9.6) 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.

13. Derive each class f_i from `filter`. The constructor for f_i specifies its mode as a parameter to the constructor for base class `filter` (4.9.6.1).
14. 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.
15. Create an instance of class `pipeline`.
16. 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.
17. 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.

NOTE: Function `parallel_pipeline` provides a strongly typed lambda-friendly way to build and run pipelines.

Members

```
namespace tbb {
    class pipeline {
    public:
        pipeline();
        ~pipeline();[5]
        void add_filter( filter& f );
        void run( size_t max_number_of_live_tokens
                  [, task_group_context& group] );
        void clear();
    };
}
```

4.9.1 pipeline()

Effects

Constructs pipeline with no filters.

4.9.2 ~pipeline()

Effects

Removes all filters from the pipeline and destroys the pipeline

4.9.3 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.

[5] 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.

4.9.4 void run(size_t max_number_of_live_tokens[, task_group_context& group])

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.

If the `group` argument is specified, pipeline's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

4.9.5 void clear()

Effects

Removes all filters from the pipeline.

4.9.6 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` (0). 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` (0).

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`).

4.9.6.1 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).

4.9.6.2 ~filter()

Effects

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

4.9.6.3 bool is_serial() const

Returns

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

4.9.6.4 bool is_ordered() const

Returns

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

4.9.6.5 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.

4.9.6.6 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.

4.9.7 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` (4.9.6) 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/compat/thread"
#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
    std::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:

18. Start the pipeline on another thread.
19. Service the `thread_bound_filter` until it reaches `end_of_stream`.
20. 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.

4.9.7.1 thread_bound_filter(mode filter_mode)

Effects

Constructs a filter of the specified mode. Section 4.9.6 describes the modes.

4.9.7.2 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 16: 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 (4.9.4) would be exceeded.
end_of_stream	No more items will ever arrive at this filter.

4.9.7.3 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 16 for details.

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

4.10 parallel_pipeline Function

Summary

Strongly typed interface for pipelined execution.

Syntax

```
void parallel_pipeline( size_t max_number_of_live_tokens,
                        const filter_t<void,void>& filter_chain
                        [, task_group_context& group] );
```

Header

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

Description

Function `parallel_pipeline` is a strongly typed lambda-friendly interface for building and running pipelines. The pipeline has characteristics similar to class pipeline, except that the stages of the pipeline are specified via functors instead of class derivation.

To build and run a pipeline from functors g_0, g_1, g_2,...g_n , write:

```
parallel_pipeline( max_number_of_live_tokens,
                   make_filter<void,I1>(mode0,g0) &
                   make_filter<I1,I2>(mode1,g1) &
                   make_filter<I2,I3>(mode2,g2) &
                   ...
                   make_filter<In,void>(moden,gn) );
```

In general, functor g_i should define its `operator()` to map objects of type I_i to objects of type I_{i+1}. Functor g_0 is a special case, because it notifies the pipeline when the end of the input stream is reached. Functor g_0 must be defined such that for a flow_control

object *fc*, the expression g_0`(`*fc*`)` either returns the next value in the input stream, or if at the end of the input stream, invokes *fc.*stop() and returns a dummy value.

The value *`max_number_of_live_tokens`* has the same meaning as it does for `pipeline::run`.

If the `group` argument is specified, pipeline's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

Example

The following example uses `parallel_pipeline` compute the root-mean-square of a sequence defined by [*`first`*,*`last`*). The example is only for demonstrating syntactic mechanics. It is not as a practical way to do the calculation because parallel overhead would be vastly higher than useful work. Operator `&` requires that the output type of its first `filter_t` argument matches the input type of its second `filter_t` argument.

```
float RootMeanSquare( float* first, float* last ) {
    float sum=0;
    parallel_pipeline( /*max_number_of_live_token=*/16,
        make_filter<void,float*>(
            filter::serial,
            [&](flow_control& fc)-> float*{
                if( first<last ) {
                    return first++;
                 } else {
                    fc.stop();
                    return NULL;
                }
            }
        ) &
        make_filter<float*,float>(
            filter::parallel,
            [](float* p){return (*p)*(*p);}
        ) &
        make_filter<float,void>(
            filter::serial,
            [&](float x) {sum+=x;}
        )
    );
    return sqrt(sum);
}
```

See the Intel® Threading Building Blocks Tutorial for a non-trivial example of `parallel_pipeline`.

4.10.1 filter_t Template Class

Summary

A filter or composite filter used in conjunction with function `parallel_pipeline`.

Syntax

```
template<typename T, typename U> class filter_t;
template<typename T, typename U, typename Func>
filter_t<T,U> make_filter( filter::mode mode, const Func& f );
template<typename T, typename V, typename U>
filter_t<T,U> operator&( const filter_t<T,V>& left,
                         const filter_t<V,U>& right );
```

Header

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

Description

A `filter_t` is a strongly typed filter that specifies its input and output types. A `filter_t` can be constructed from a functor or by composing of two `filter_t` objects with `operator&`. See 4.4 for an example. The same `filter_t` object can be shared by multiple `&` expressions.

Members

```
namespace tbb {
    template<typename T, typename U>
    class filter_t {
    public:
        filter_t();
        filter_t( const filter_t<T,U>& rhs );
        template<typename Func>
        filter_t( filter::mode mode, const Func& func );
        void operator=( const filter_t<T,U>& rhs );
        ~filter_t();
        void clear();
    };

    template<typename T, typename U, typename Func>
    filter_t<T,U> make_filter( filter::mode mode, const Func& f );
    template<typename T, typename V, typename U>
    filter_t<T,U> operator&( const filter_t<T,V>& left,
                             const filter_t<V,U>& right );
}
```

4.10.1.1 filter_t()

Effects

Construct an undefined filter.

CAUTION: The effect of using an undefined filter by `operator&` or `parallel_pipeline` is undefined.

4.10.1.2 filter_t(const filter_t<T,U>& rhs)

Effects

Construct a copy of *`rhs`*.

4.10.1.3 template<typename Func> filter_t(filter::mode mode, const Func& f)

Effects

Construct a `filter_t` that uses a copy of functor *`f`* to map an input value *`t`* of type *`T`* to an output value *`u`* of type *`U`*.

NOTE: When `parallel_pipeline` uses the `filter_t`, it computes *`u`* by evaluating *`f(t)`*, unless *`T`* is `void`. In the void case *`u`* is computed by the expression *`u=f(fc)`*, where *`fc`* is of type `flow_control`.

See 4.9.6 for a description of the *`mode`* argument.

4.10.1.4 void operator=(const filter_t<T,U>& rhs)

Effects

Update `*this` to use the functor associated with *`rhs`*.

4.10.1.5 ~filter_t()

Effects

Destroy the `filter_t`.

4.10.1.6 void clear()

Effects

Set `*this` to an undefined filter.

4.10.1.7 template<typename T, typename U, typename Func> filter_t<T,U> make_filter(filter::mode mode, const Func& f)

Returns

filter_t<*T*,*U*>(*mode*,*f*)

4.10.1.8 template<typename T, typename V, typename U> filter_t<T,U> operator& (const filter_t<T,V>& left, const filter_t<V,U>& right)

Requires

The output type of `left` must match the input type of `right`.

Returns

A `filter_t` representing the composition of filters `left` and `right`. The composition behaves as if the output value of `left` becomes the input value of `right`.

4.10.2 flow_control Class

```
class flow_control;
```

Summary

Enables the first filter in a composite filter to indicate when the end of input has been reached.

Syntax

```
class flow_control;
```

Header

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

Description

Template function `parallel_pipeline` passes a `flow_control` object `fc` to the input functor of a filter_t. When the input functor reaches the end of its input, it should invoke `fc.stop()` and return a dummy value. See 4.4 for an example.

Members

```
namespace tbb {
    class flow_control {
    public:
        void stop();
```

```
    };
}
```

4.11 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 17.

Table 17: 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 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 (12.2.1), `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>());
}
```

4.12 parallel_invoke Template Function

Summary

Template function that evaluates several functions in parallel.

Syntax[6]

```
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 `f0()`, `f1()`,…`fk` possibly in parallel. There can be from 2 to 10 arguments. Each argument must have a type for 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);}
};
```

---

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

```
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);} );
}
```

5 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.

5.1 Container Range Concept

Summary

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

Requirements

A Container Range is a Range (4.2) with the further requirements listed in Table 18.

Table 18: Requirements on a Container Range R (In Addition to Table 8)

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` (5.4.4) and `concurrent_vector` (5.8.5) both have member types `range_type` and `const_range_type` that model a Container Range.

Use the range types in conjunction with `parallel_for` (4.4), `parallel_reduce` (4.5), and `parallel_scan` (4.6) to iterate over items in a container.

5.2 concurrent_unordered_map Template Class

Summary

Template class for associative container that supports concurrent insertion and traversal.

Syntax

```
template <typename Key,
          typename Element,
          typename Hasher = tbb_hash<Key>,
          typename Equality = std::equal_to<Key >,
          typename Allocator = tbb::tbb_allocator<std::pair<const
Key, Element > > >
class concurrent_unordered_map;
```

Header

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

Description

A concurrent_unordered_map supports concurrent insertion and traversal, but not concurrent erasure. The interface has no visible locking. It may hold locks internally, but never while calling user defined code. It has semantics similar to the C++0x `std::unordered_map` except as follows:

- Methods requiring C++0x language features (such as rvalue references and `std::initializer_list`) are currently omitted.
- The erase methods are prefixed with `unsafe_`, to indicate that they are not concurrency safe.
- Bucket methods are prefixed with `unsafe_` as a reminder that they are not concurrency safe with respect to insertion.
- The insert methods may create a temporary pair that is destroyed if another thread inserts the same key concurrently.
- Like `std::list`, insertion of new items does *not* invalidate any iterators, nor change the order of items already in the map. Insertion and traversal may be concurrent.
- The iterator types `iterator` and `const_iterator` are of the forward iterator category.
- Insertion does not invalidate or update the iterators returned by `equal_range`, so insertion may cause non-equal items to be inserted at the end of the range. However, the first iterator will nonethless point to the equal item even after an insertion operation.

NOTE: The key differences between classes concurrent_unordered_map and concurrent_hash_map each are:

- concurrent_unordered_map: permits concurrent traversal and insertion, no visible locking, closely resembles the C++0x unordered_map.
- concurrent_hash_map: permits concurrent erasure, built-in locking

CAUTION: As with any form of hash table, keys that are equal must have the same hash code, and the ideal hash function distributes keys uniformly across the hash code space.

Members

In the following synopsis, methods in bold may be concurrently invoked. For example, three different threads can concurrently call methods `insert`, `begin`, and `size`. Their results might be non-deterministic. For example, the result from `size` might correspond to before or after the insertion.

```
template <typename Key,
          typename Element,
          typename Hasher = tbb_hash<Key>,
          typename Equal = std::equal_to<Key>,
          typename Allocator = tbb::tbb_allocator<std::pair<const
Key, Element > > >
class concurrent_unordered_map {
public:
    // types
    typedef Key key_type;
    typedef std::pair<const Key, T> value_type;
    typedef Element mapped_type;
    typedef Hash hasher;
    typedef Equality key_equal;
    typedef Alloc allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference
const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/destroy/copy
    explicit concurrent_unordered_map(size_type n =
implementation-defined,
```

```
        const Hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
    template <typename InputIterator>
        concurrent_unordered_map(
            InputIterator first, InputIterator last,
            size_type n = implementation-defined,
            const hasher& hf = hasher(),
            const key_equal& eql = key_equal(),
            const allocator_type& a = allocator_type());
    concurrent_unordered_map(const concurrent_unordered_map&);
    concurrent_unordered_map(const Alloc&);
    concurrent_unordered_map(const concurrent_unordered_map&,
const Alloc&);
    ~concurrent_unordered_map();

    concurrent_unordered_map& operator=( const
concurrent_unordered_map&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;      // May take linear time!
    size_type size() const; // May take linear time!
    size_type max_size() const;

    // iterators
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    const_iterator cbegin() const;
    const_iterator cend() const;

    // modifiers
    std::pair<iterator, bool> insert(const value_type& x);
    iterator insert(const_iterator hint, const value_type& x);
    template<class InputIterator> void insert(InputIterator first,
                                              InputIterator last);

    iterator unsafe_erase(const_iterator position);
    size_type unsafe_erase(const key_type& k);
    iterator unsafe erase(const iterator first, const iterator
last);
    void clear();

    void swap(concurrent_unordered_map&);
```

```
    // observers
    hasher hash_function() const;
    key_equal key_eq() const;

    // lookup
    iterator find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator> equal_range(const key_type& k);
    std::pair<const_iterator, const_iterator> equal_range(const
key_type& k) const;
    mapped_type& operator[](const key_type& k);
    mapped_type& at( const key_type& k );
    const mapped_type& at(const key_type& k) const;

    // parallel iteration
    typedef implementation defined range_type;
    typedef implementation defined const_range_type;
    range_type range();
    const_range_type range() const;

    // bucket interface - for debugging
    size_type unsafe_bucket_count() const;
    size_type unsafe_max_bucket_count() const;
    size_type unsafe_bucket_size(size_type n);
    size_type unsafe_bucket(const key_type& k) const;
    local_iterator unsafe_begin(size_type n);
    const_local_iterator unsafe_begin(size_type n) const;
    local_iterator unsafe_end(size_type n);
    const_local_iterator unsafe_end(size_type n) const;
    const_local_iterator unsafe_cbegin(size_type n) const;
    const_local_iterator unsafe_cend(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    void rehash(size_type n);
};
```

5.2.1 Construct, Destroy, Copy

5.2.1.1 explicit concurrent_unordered_map (size_type n = *implementation-defined*, const hasher& hf = hasher(),const key_equal& eql = key_equal(), const allocator_type& a = allocator_type())

Effects

Construct empty table with *n* buckets.

5.2.1.2 template <typename InputIterator> concurrent_unordered_map (InputIterator first, InputIterator last, size_type n = *implementation-defined*, const hasher& hf = hasher(), const key_equal& eql = key_equal(), const allocator_type& a = allocator_type())

Effects

Construct table with *n* buckets initialized with `value_type(*i)` where `i` is in the half open interval `[first,last)`.

5.2.1.3 concurrent_unordered_map(const unordered_map& m)

Effects

Construct copy of map `m`.

5.2.1.4 concurrent_unordered_map(const Alloc& a)

Construct empy map using allocator `a`.

5.2.1.5 concurrent_unordered_map(const unordered_map&, const Alloc& a)

Effects

Construct copy of map `m` using allocator `a`.

5.2.1.6 ~concurrent_unordered_map()

Effects

Destroy the map.

5.2.1.7 concurrent_ unordered_map& operator=(const concurrent_unordered_map& m);

Effects

Set `*this` to a copy of map *m*.

5.2.1.8 allocator_type get_allocator() const;

Get copy of the allocator associated with `*this`.

5.2.2 Size and capacity

5.2.2.1 bool empty() const

Returns

`size()!=0`.

5.2.2.2 size_type size() const

Returns

Number of items in `*this`.

CAUTION: Though the current implementation takes time O(1), possible future implementations might take time O(P), where P is the number of hardware threads.

5.2.2.3 size_type max_size() const

Returns

CAUTION: Upper bound on number of items that `*this` can hold.

CAUTION: The upper bound may be much higher than what the container can actually hold.

5.2.3 Iterators

Template class `concurrent_unordered_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)` do *not* invalidate any existing iterators that point into the table. Note that an iterator obtained via begin() will no longer point to the first item if `insert` inserts an item before it.

Methods `cbegin` and `cend` follow C++0x conventions. They return `const_iterator` even if the object is non-const.

5.2.3.1 iterator begin()

Returns

`iterator` pointing to first item in the map.

5.2.3.2 const_iterator begin() const

Returns

`const_iterator` pointing to first item in in the map.

5.2.3.3 iterator end()

Returns

`iterator` pointing to immediately past last item in the map.

5.2.3.4 const_iterator end() const

Returns

`const_iterator` pointing to immediately past last item in the map.

5.2.3.5 const_iterator cbegin() const

Returns

`const_iterator` pointing to first item in the map.

5.2.3.6 const_iterator cend() const

Returns

`const_iterator` pointing to immediately after the last item in the map.

5.2.4 Modifiers

5.2.4.1 std::pair<iterator, bool> insert(const value_type& x)

Effects

Constructs copy of `x` and attempts to insert it into the map. Destroys the copy if the attempt fails because there was already an item with the same key.

Returns

`std::pair(`*iterator*,*success*). The value *iterator* points to an item in the map with a matching key. The value of *success* is true if the item was inserted; false otherwise.

5.2.4.2 iterator insert(const_iterator hint, const value_type& x)

Effects

Same as `insert(x)`.

NOTE: The current implementation ignores the hint argument. Other implementations might not ignore it. It exists for similarity with the C++0x class `unordered_map`. It hints to the implementation about where to start searching. Typically it should point to an item adjacent to where the item will be inserted.

Returns

Iterator pointing to inserted item, or item already in the map with the same key.

5.2.4.3 template<class InputIterator> void insert(InputIterator first, InputIterator last)

Effects

Does `insert(*i)` where `i` is in the half-open interval `[`*first*,*last*`)`.

5.2.4.4 iterator unsafe_erase(const_iterator position)

Effects

Remove item pointed to by *position* from the map.

Returns

Iterator pointing to item that was immediately after the erased item, or `end()` if erased item was the last item in the map.

5.2.4.5 size_type unsafe_erase(const key_type& k)

Effects

Remove item with key *k* if such an item exists.

Returns

1 if an item was removed; 0 otherwise.

5.2.4.6 iterator unsafe_erase(const_iterator first, const_iterator last)

Effects

Remove `*i` where `i` is in the half-open interval [*first*,*last*).

Returns

last

5.2.4.7 void clear()

Effects

Remove all items from the map.

5.2.4.8 void swap(concurrent_unordered_map& m)

Effects

Swap contents of `*this` and *m*.

5.2.5 Observers

5.2.5.1 hasher hash_function() const

Returns

Hashing functor associated with the map.

5.2.5.2 key_equal key_eq() const

Returns

Key equivalence functor associcated with the map.

5.2.6 Lookup

5.2.6.1 iterator find(const key_type& k)

Returns

`iterator` pointing to item with key equivalent to k, or `end()` if no such item exists.

5.2.6.2 const_iterator find(const key_type& k) const

Returns

`const_iterator` pointing to item with key equivalent to k, or `end()` if no such item exists.

5.2.6.3 size_type count(const key_type& k) const

Returns

Number of items with keys equivalent to k.

5.2.6.4 std::pair<iterator, iterator> equal_range(const key_type& k)

Returns

Range containing all keys in the map that are equivalent to k.

5.2.6.5 std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const

Returns

Range containing all keys in the map that are equivalent to k.

5.2.6.6 mapped_type& operator[](const key_type& k)

Effects

Inserts a new item if item with key equivalent to k is not already present.

Returns

Reference to x.second, where x is item in map with key equivalent to k.

5.2.6.7 mapped_type& at(const key_type& k)

Effects

Throws exception if item with key equivalent to k is not already present.

Returns

Reference to x.second, where x is the item in map with key equivalent to k.

5.2.6.8 const mapped_type& at(const key_type& k) const

Effects

Throws exception if item with key equivalent to k is not already present.

Returns

Const reference to x.second, where x is the item in map with key equivalent to k.

5.2.7 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (5.1). 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.

5.2.7.1 const_range_type range() const

Returns

`const_range_type` object representing all keys in the table.

5.2.7.2 range_type range()

Returns

`range_type` object representing all keys in the table.

5.2.8 Bucket Interface

The bucket interface is intended for debugging. It is not concurrency safe. The mapping of keys to buckets is implementation specific. The interface is similar to the bucket interface for the C++0x class `unordered_map`, except that the prefix `unsafe_` has been added as a reminder that the methods are unsafe to use during concurrent insertion.

Buckets are numbered from 0 to `unsafe_bucket_count()`−1. To iterate over a bucket use a `local_iterator` or `const_local_iterator`.

5.2.8.1 size_type unsafe_bucket_count() const

Returns

Number of buckets.

5.2.8.2 size_type unsafe_max_bucket_count() const

Returns

Upper bound on possible number of buckets.

5.2.8.3 size_type unsafe_bucket_size(size_type n)

Returns

Number of items in bucket *n*.

5.2.8.4 size_type unsafe_bucket(const key_type& k) const

Returns

Index of bucket where item with key *k* would be placed.

5.2.8.5 local_iterator unsafe_begin(size_type n)

Returns

`local_iterator` pointing to first item in bucket *n*.

5.2.8.6 const_local_iterator unsafe_begin(size_type n) const

Returns

`const_local_iterator` pointing to first item in bucket *n*.

5.2.8.7 local_iterator unsafe_end(size_type n)

Returns

`local_iterator` pointing to immediately after the last item in bucket *`n`*.

5.2.8.8 const_local_iterator unsafe_end(size_type n) const

Returns

`const_local_iterator` pointing to immediately after the last item in bucket *`n`*.

5.2.8.9 const_local_iterator unsafe_cbegin(size_type n) const

Returns

`const_local_iterator` pointing to first item in bucket *`n`*.

5.2.8.10 const_local_iterator unsafe_cend(size_type n) const

Returns

`const_local_iterator` pointing to immediately past last item in bucket *`n`*.

5.2.9 Hash policy

5.2.9.1 float load_factor() const

Returns

Average number of elements per bucket.

5.2.9.2 float max_load_factor() const

Returns

Maximum size of a bucket. If insertion of an item causes a bucket to be bigger, the implementaiton may repartition or increase the number of buckets.

5.2.9.3 void max_load_factor(float z)

Effects

Set maximum size for a bucket to *`z`*.

5.2.9.4 void rehash(size_type n)

Requirements

`n` must be a power of two.

Effects

No effect if current number of buckets is at least n. Otherwise increases number of buckets to n.

5.3 concurrent_unordered_set Template Class

Summary

Template class for a set container that supports concurrent insertion and traversal.

Syntax

```
template <typename Key,
          typename Hasher = tbb_hash<Key>,
          typename Equality = std::equal_to<Key>,
          typename Allocator = tbb::tbb_allocator<Key>
class concurrent_unordered_set;
```

Header

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

Description

A concurrent_unordered_set supports concurrent insertion and traversal, but not concurrent erasure. The interface has no visible locking. It may hold locks internally, but never while calling user defined code. It has semantics similar to the C++0x `std::unordered_set` except as follows:

- Methods requiring C++0x language features (such as rvalue references and `std::initializer_list`) are currently omitted.
- The erase methods are prefixed with `unsafe_`, to indicate that they are not concurrency safe.
- Bucket methods are prefixed with `unsafe_` as a reminder that they are not concurrency safe with respect to insertion.
- The insert methods may create a temporary pair that is destroyed if another thread inserts the same key concurrently.

- Like `std::list`, insertion of new items does *not* invalidate any iterators, nor change the order of items already in the set. Insertion and traversal may be concurrent.
- The iterator types `iterator` and `const_iterator` are of the forward iterator category.
- Insertion does not invalidate or update the iterators returned by `equal_range`, so insertion may cause non-equal items to be inserted at the end of the range. However, the first iterator will nonethless point to the equal item even after an insertion operation.

CAUTION: As with any form of hash table, keys that are equal must have the same hash code, and the ideal hash function distributes keys uniformly across the hash code space.

Members

In the following synopsis, methods in bold may be concurrently invoked. For example, three different threads can concurrently call methods `insert`, `begin`, and `size`. Their results might be non-deterministic. For example, the result from `size` might correspond to before or after the insertion.

```
template <typename Key,
          typename Hasher = tbb_hash<Key>,
          typename Equal = std::equal_to<Key>,
          typename Allocator = tbb::tbb_allocator<Key>
class concurrent_unordered_set {
public:
    // types
    typedef Key key_type;
    typedef Key value_type;
    typedef Key mapped_type;
    typedef Hash hasher;
    typedef Equality key_equal;
    typedef Alloc allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference
const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/destroy/copy
```

```
    explicit concurrent_unordered_set(size_type n =
implementation-defined,
        const Hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
    template <typename InputIterator>
    concurrent_unordered_set(
            InputIterator first, InputIterator last,
            size_type n = implementation-defined,
            const hasher& hf = hasher(),
            const key_equal& eql = key_equal(),
            const allocator_type& a = allocator_type());
    concurrent_unordered_set(const concurrent_unordered_set&);
    concurrent_unordered_set(const Alloc&);
    concurrent_unordered_set(const concurrent_unordered_set&,
const Alloc&);
    ~concurrent_unordered_set();

    concurrent_unordered_set& operator=( const
concurrent_unordered_set&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;       // May take linear time!
    size_type size() const; // May take linear time!
    size_type max_size() const;

    // iterators
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    const_iterator cbegin() const;
    const_iterator cend() const;

    // modifiers
    std::pair<iterator, bool> insert(const value_type& x);
    iterator insert(const_iterator hint, const value_type& x);
    template<class InputIterator> void insert(InputIterator first,
                                              InputIterator last);

    iterator unsafe_erase(const_iterator position);
    size_type unsafe_erase(const key_type& k);
    iterator unsafe erase(const iterator first, const iterator
last);
    void clear();
```

```
    void swap(concurrent_unordered_set&);

    // observers
    hasher hash_function() const;
    key_equal key_eq() const;

    // lookup
    iterator find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator> equal_range(const key_type& k);
    std::pair<const_iterator, const_iterator> equal_range(const
key_type& k) const;

    // parallel iteration
    typedef implementation defined range_type;
    typedef implementation defined const_range_type;
    range_type range();
    const_range_type range() const;

    // bucket interface - for debugging
    size_type unsafe_bucket_count() const;
    size_type unsafe_max_bucket_count() const;
    size_type unsafe_bucket_size(size_type n);
    size_type unsafe_bucket(const key_type& k) const;
    local_iterator unsafe_begin(size_type n);
    const_local_iterator unsafe_begin(size_type n) const;
    local_iterator unsafe_end(size_type n);
    const_local_iterator unsafe_end(size_type n) const;
    const_local_iterator unsafe_cbegin(size_type n) const;
    const_local_iterator unsafe_cend(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    void rehash(size_type n);
};
```

5.3.1 Construct, Destroy, Copy

5.3.1.1 explicit concurrent_unordered_set (size_type n = *implementation-defined*, const hasher& hf = hasher(),const key_equal& eql = key_equal(), const allocator_type& a = allocator_type())

Effects

Construct empty set with *n* buckets.

5.3.1.2 template <typename InputIterator> concurrent_unordered_set (InputIterator first, InputIterator last, size_type n = *implementation-defined*, const hasher& hf = hasher(), const key_equal& eql = key_equal(), const allocator_type& a = allocator_type())

Effects

Construct set with *n* buckets initialized with `value_type(*i)` where `i` is in the half open interval `[first,last)`.

5.3.1.3 concurrent_unordered_set(const unordered_set& m)

Effects

Construct copy of set `m`.

5.3.1.4 concurrent_unordered_set(const Alloc& a)

Construct empy set using allocator `a`.

5.3.1.5 concurrent_unordered_set(const unordered_set&, const Alloc& a)

Effects

Construct copy of set `m` using allocator `a`.

5.3.1.6 ~concurrent_unordered_set()

Effects

Destroy the set.

5.3.1.7 concurrent_ unordered_set& operator=(const concurrent_unordered_set& m);

Effects

Set `*this` to a copy of set *m*.

5.3.1.8 allocator_type get_allocator() const;

Get copy of the allocator associated with `*this`.

5.3.2 Size and capacity

5.3.2.1 bool empty() const

Returns

`size()!=0`.

5.3.2.2 size_type size() const

Returns

Number of items in `*this`.

CAUTION: Though the current implementation takes time O(1), possible future implementations might take time O(P), where P is the number of hardware threads.

5.3.2.3 size_type max_size() const

Returns

CAUTION: Upper bound on number of items that `*this` can hold.

CAUTION: The upper bound may be much higher than what the container can actually hold.

5.3.3 Iterators

Template class `concurrent_unordered_set` supports forward iterators; that is, iterators that can advance only forwards across a set. Reverse iterators are not supported. Concurrent operations (`count`, `find`, `insert)` do *not* invalidate any existing iterators that point into the set. Note that an iterator obtained via begin() will no longer point to the first item if `insert` inserts an item before it.

Methods `cbegin` and `cend` follow C++0x conventions. They return `const_iterator` even if the object is non-const.

5.3.3.1 iterator begin()

Returns

`iterator` pointing to first item in the set.

5.3.3.2 const_iterator begin() const

Returns

`const_iterator` pointing to first item in in the set.

5.3.3.3 iterator end()

Returns

`iterator` pointing to immediately past last item in the set.

5.3.3.4 const_iterator end() const

Returns

`const_iterator` pointing to immediately past last item in the set.

5.3.3.5 const_iterator cbegin() const

Returns

`const_iterator` pointing to first item in the set.

5.3.3.6 const_iterator cend() const

Returns

`const_iterator` pointing to immediately after the last item in the set.

5.3.4 Modifiers

5.3.4.1 std::pair<iterator, bool> insert(const value_type& x)

Effects

Constructs copy of `x` and attempts to insert it into the set. Destroys the copy if the attempt fails because there was already an item with the same key.

Returns

`std::pair(`*iterator*,*success*). The value *iterator* points to an item in the set with a matching key. The value of *success* is true if the item was inserted; false otherwise.

5.3.4.2 iterator insert(const_iterator hint, const value_type& x)

Effects

Same as `insert(x)`.

NOTE: The current implementation ignores the hint argument. Other implementations might not ignore it. It exists for similarity with the C++0x class `unordered_set`. It hints to the implementation about where to start searching. Typically it should point to an item adjacent to where the item will be inserted.

Returns

Iterator pointing to inserted item, or item already in the set with the same key.

5.3.4.3 template<class InputIterator> void insert(InputIterator first, InputIterator last)

Effects

Does `insert(*i)` where `i` is in the half-open interval `[`*first*,*last*`)`.

5.3.4.4 iterator unsafe_erase(const_iterator position)

Effects

Remove item pointed to by *position* from the set.

Returns

Iterator pointing to item that was immediately after the erased item, or `end()` if erased item was the last item in the set.

5.3.4.5 size_type unsafe_erase(const key_type& k)

Effects

Remove item with key *k* if such an item exists.

Returns

1 if an item was removed; 0 otherwise.

5.3.4.6 iterator unsafe_erase(const_iterator first, const_iterator last)

Effects

Remove `*i` where `i` is in the half-open interval [*first*,*last*).

Returns

last

5.3.4.7 void clear()

Effects

Remove all items from the set.

5.3.4.8 void swap(concurrent_unordered_set& m)

Effects

Swap contents of `*this` and *m*.

5.3.5 Observers

5.3.5.1 hasher hash_function() const

Returns

Hashing functor associated with the set.

5.3.5.2 key_equal key_eq() const

Returns

Key equivalence functor associcated with the set.

5.3.6 Lookup

5.3.6.1 iterator find(const key_type& k)

Returns

`iterator` pointing to item with key equivalent to k, or `end()` if no such item exists.

5.3.6.2 const_iterator find(const key_type& k) const

Returns

`const_iterator` pointing to item with key equivalent to k, or `end()` if no such item exists.

5.3.6.3 size_type count(const key_type& k) const

Returns

Number of items with keys equivalent to k.

5.3.6.4 std::pair<iterator, iterator> equal_range(const key_type& k)

Returns

Range containing all keys in the set that are equivalent to k.

5.3.6.5 std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const

Returns

Range containing all keys in the set that are equivalent to k.

5.3.7 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (5.1). 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.

5.3.7.1 const_range_type range() const

Returns

`const_range_type` object representing all keys in the set.

5.3.7.2 range_type range()

Returns

`range_type` object representing all keys in the set.

5.3.8 Bucket Interface

The bucket interface is intended for debugging. It is not concurrency safe. The mapping of keys to buckets is implementation specific. The interface is similar to the bucket interface for the C++0x class `unordered_set`, except that the prefix `unsafe_` has been added as a reminder that the methods are unsafe to use during concurrent insertion.

Buckets are numbered from 0 to `unsafe_bucket_count()`−1. To iterate over a bucket use a `local_iterator` or `const_local_iterator`.

5.3.8.1 size_type unsafe_bucket_count() const

Returns

Number of buckets.

5.3.8.2 size_type unsafe_max_bucket_count() const

Returns

Upper bound on possible number of buckets.

5.3.8.3 size_type unsafe_bucket_size(size_type n)

Returns

Number of items in bucket `n`.

5.3.8.4 size_type unsafe_bucket(const key_type& k) const

Returns

Index of bucket where item with key `k` would be placed.

5.3.8.5 local_iterator unsafe_begin(size_type n)

Returns

`local_iterator` pointing to first item in bucket `n`.

5.3.8.6 const_local_iterator unsafe_begin(size_type n) const

Returns

`const_local_iterator` pointing to first item in bucket `n`.

5.3.8.7 local_iterator unsafe_end(size_type n)

Returns

`local_iterator` pointing to immediately after the last item in bucket `n`.

5.3.8.8 const_local_iterator unsafe_end(size_type n) const

Returns

`const_local_iterator` pointing to immediately after the last item in bucket `n`.

5.3.8.9 const_local_iterator unsafe_cbegin(size_type n) const

Returns

`const_local_iterator` pointing to first item in bucket `n`.

5.3.8.10 const_local_iterator unsafe_cend(size_type n) const

Returns

`const_local_iterator` pointing to immediately past last item in bucket `n`.

5.3.9 Hash policy

5.3.9.1 float load_factor() const

Returns

Average number of elements per bucket.

5.3.9.2 float max_load_factor() const

Returns

Maximum size of a bucket. If insertion of an item causes a bucket to be bigger, the implementaiton may repartition or increase the number of buckets.

5.3.9.3 void max_load_factor(float z)

Effects

Set maximum size for a bucket to `z`.

5.3.9.4 void rehash(size_type n)

Requirements

`n` must be a power of two.

Effects

No effect if current number of buckets is at least n. Otherwise increases number of buckets to n.

5.4 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 5.4.3. 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 19.

Table 19: 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 rehash( size_type n=0 );
        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;
        size_type bucket_count() 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 Safety

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.

5.4.1 Whole Table Operations

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

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

Effects

Constructs empty table.

5.4.1.2 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 $n \geq 4P$.

5.4.1.3 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.

5.4.1.4 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*).

5.4.1.5 ~concurrent_hash_map()

Effects

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

5.4.1.6 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.

5.4.1.7 void swap(concurrent_hash_map& table)

Effects

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

5.4.1.8 void rehash(size_type n=0)

Effects

Internally, the table is partitioned into buckets. Method `rehash` reorgnizes these internal buckets in a way that may improve performance of future lookups. Raises number of internal buckets to *n* if *n*>0 and *n* exceeds the current number of buckets.

CAUTION: The current implementation never reduces the number of buckets. A future implementation might reduce the number of buckets if *n* is less than the current number of buckets.

NOTE: The ratio of items to buckets affects time and space usage by a table. A high ratio saves space at the expense of time. A low ratio does the opposite. The default ratio is 0.5 to 1 items per bucket on average.

5.4.1.9 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.

5.4.1.10 allocator_type get_allocator() const

Returns

Copy of allocator used to construct table.

5.4.2 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 20: 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.

Class	value_type	Implied Lock on pair
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.

5.4.2.1 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();
    };
}
```

5.4.2.1.1 bool empty() const

Returns

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

5.4.2.1.2 void release()

Effects

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

5.4.2.1.3 const value_type& operator*() const

Effects

Raises assertion failure if `empty()` and TBB_USE_ASSERT (3.2.1) is defined as nonzero.

Returns

Const reference to key-value pair.

5.4.2.1.4 const value_type* operator->() const

Returns

```
&operator*()
```

5.4.2.1.5 const_accessor()

Effects

Constructs const_accessor that points to nothing.

5.4.2.1.6 ~const_accessor

Effects

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

5.4.2.2 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;
    };
}
```

5.4.2.2.1 value_type& operator*() const

Effects

Raises assertion failure if `empty()` and `TBB_USE_ASSERT` (3.2.1) is defined as nonzero.

Returns

Reference to key-value pair.

5.4.2.2.2 value_type* operator->() const

Returns

```
&operator*()
```

5.4.3 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 even with const qualifier. It is unsafe to use these operations concurrently with any other operation. An exception to this rule is that `count` and `find` do not invalidate iterators if no insertions or erasures have occurred after the most recent call to method `rehash`.

TIP: In serial code, the `equal_range` method should be used instead of the `find` method for lookup, since `equal_range` is faster and does not invalidate iterators.

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);
    }
}
```

5.4.3.1 size_type count(const Key& key) const

CAUTION: This method may invalidate previously obtained iterators. In serial code, you can instead use equal_range that does not have such problems.

Returns

1 if map contains key; 0 otherwise.

5.4.3.2 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.

CAUTION: This method may invalidate previously obtained iterators. In serial code, you can instead use equal_range that does not have such problems.

Returns

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

5.4.3.3 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

CAUTION: This method may invalidate previously obtained iterators. In serial code, you can instead use equal_range that does not have such problems.

Returns

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

5.4.3.4 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.

5.4.3.5 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.

5.4.3.6 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.

5.4.3.7 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.

5.4.3.8 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.

TIP: If you do not need to access the data after insertion, use the form of insert that does not take an accessor; it may work faster and use fewer locks.

5.4.3.9 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.

5.4.3.10 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.

5.4.3.11 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.

5.4.3.12 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.

5.4.4 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (5.1). 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.

NOTE: Do not call concurrent operations, including `count` and `find` while iterating the table. Use `concurrent_unordered_map` if concurrent traversal and insertion are required.

5.4.4.1 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.

5.4.4.2 range_type range(size_t grainsize=1)

Returns

`range_type` object for the table.

5.4.5 Capacity

5.4.5.1 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.

5.4.5.2 bool empty() const

Returns

`size()==0.`

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

5.4.5.3 size_type max_size() const

Returns

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

5.4.5.4 size_type bucket_count() const

Returns

Current number of internal buckets. See method `rehash` for discussion of buckets.

5.4.6 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, An exception to this rule is that `count` and `find` do not invalidate iterators if no insertions or erasures have occurred after the most recent call to method `rehash`.

NOTE: Do not call concurrent operations, including `count` and `find` while iterating the table. Use `concurrent_unordered_map` if concurrent traversal and insertion are required.

5.4.6.1 iterator begin()

Returns

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

5.4.6.2 iterator end()

Returns

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

5.4.6.3 const_iterator begin() const

Returns

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

5.4.6.4 const_iterator end() const

Returns

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

5.4.6.5 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.

TIP: This method is serial alternative to concurrent `count` and `find` methods.

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

Description

See 5.4.6.5.

5.4.7 Global Functions

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

5.4.7.1 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 $\texttt{bool}(v_1 == v_2)$ is true.

5.4.7.2 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

$!(a==b)$

5.4.7.3 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)`

5.4.8 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`.

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>& );
};
```

5.5 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[7], subject to memory limitations on the target machine.

[7] 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.

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

Table 21: 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
        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[8]( 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;
    };
}
```

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

Effects

Constructs empty queue.

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

Effects

Constructs a copy of *src*.

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

5.5.3 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*).

5.5.4 ~concurrent_queue()

Effects

Destroys all items in the queue.

5.5.5 void push(const T& source)

Effects

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

5.5.6 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.

5.5.7 void clear()

Effects

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

5.5.8 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.

5.5.9 bool empty() const

Returns

`true` if queue has no items; false otherwise.

5.5.10 Alloc get_allocator() const

Returns

Copy of allocator used to construct the queue.

5.5.11 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;

}
```

5.5.11.1 iterator unsafe_begin()

Returns

`iterator` pointing to beginning of the queue.

5.5.11.2 iterator unsafe_end()

Returns

`iterator` pointing to end of the queue.

5.5.11.3 const_iterator unsafe_begin() const

Returns

`const_iterator` with pointing to beginning of the queue.

5.5.11.4 const_iterator unsafe_end() const

Returns

`const_iterator` pointing to end of the queue.

5.6 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`.
- Adds an `abort` operation that causes any waiting `push` or `pop` operation to abort and throw an exception.

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[9]( const T& source );
        bool try_pop[10]( T& destination );
        void abort();
        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.

---

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

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

5.6.1 void push(const T& source)

Effects

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

5.6.2 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.

5.6.3 void abort()

Effects

Wakes up any threads that are waiting on the queue via the `push` and `pop` operations and raises the `tbb::user_abort` exception on those threads. This feature is unavailable if `TBB_USE_EXCEPTIONS` is not set.

5.6.4 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.

5.6.5 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.

5.6.6 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.

5.6.7 bool empty() const

Returns

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

5.6.8 size_type capacity() const

Returns

Maximum number of values that the queue can hold.

5.6.9 void set_capacity(size_type capacity)

Effects

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

5.7 concurrent_priority_queue Template Class

Summary

Template class for priority queue with concurrent operations.

Syntax

```
template<typename T, typename Compare=std::less<T>, typename
Alloc=cache_aligned_allocator<T> >
class concurrent_priority_queue;
```

Header

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

Description

A `concurrent_priority_queue` is a container that permits multiple threads to concurrently push and pop items. Items are popped in priority order as determined by a template parameter. The queue's capacity is unbounded, subject to memory limitations on the target machine.

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

Table 43: Differences between STL priority_queue and Intel® Threading Building Blocks concurrent_priority_queue

Feature	STL `std::priority_queue`	`concurrent_priority_queue`
Choice of underlying container	Sequence template parameter	No choice of underlying container; allocator choice is provided instead
Access to highest priority item	`const value_type& top() const`	Not available. Unsafe for concurrent container

Copy and pop item if present	`bool b=!q.empty();`
	`if(b) {`
	`  x=q.top();`
	`  q.pop();`
	`}`
		`bool b = q.try_pop(x);`

Get number of items in queue	`size_type size() const`	Same, but may be inaccurate due to pending concurrent push or pop operations
Check if there are items in queue	`bool empty() const`	Same, but may be inaccurate due to pending concurrent push or pop operations

Members

```
namespace tbb {
  template <typename T, typename Compare=std::less<T>,
            typename A=cache_aligned_allocator<T> >
  class concurrent_priority_queue {
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef A allocator_type;

    concurrent_priority_queue(const allocator_type& a =
```

```
      allocator_type());
    concurrent_priority_queue(size_type init_capacity,
      const allocator_type& a = allocator_type());
    template<typename InputIterator>
    concurrent_priority_queue(InputIterator begin,
      InputIterator end, const allocator_type& a =
      allocator_type());
    concurrent_priority_queue(const
      concurrent_priority_queue& src, const
      allocator_type& a = allocator_type());
    concurrent_priority_queue& operator=(const
      concurrent_priority_queue& src);
    ~concurrent_priority_queue();

    bool empty() const;
    size_type size() const;
    void push(const_reference elem);
    bool try_pop(reference elem);
    void clear();
    void swap(concurrent_priority_queue& other);
    allocator_type get_allocator() const;
  };
}
```

5.7.1 concurrent_priority_queue(const allocator_type& a = allocator_type())

Effects

Constructs empty queue.

5.7.2 concurrent_priority_queue(size_type init_capacity, const allocator_type& a = allocator_type())

Effects

Constructs an empty queue with an initial capacity.

5.7.3 concurrent_priority_queue(InputIterator begin, InputIterator end, const allocator_type& a = allocator_type())

Effects

Constructs a queue containing copies of elements in the iterator half-open interval [`begin, end`).

5.7.4 concurrent_priority_queue (const concurrent_priority_queue& src, const allocator_type& a = allocator_type())

Effects

Constructs a copy of `src`. This operation is not thread-safe and may result in an error or an invalid copy of `src` if another thread is concurrently modifying `src`.

5.7.5 concurrent_priority_queue& operator=(const concurrent_priority_queue& src)

Effects

Assign contents of `src` to *`this`. This operation is not thread-safe and may result in an error or an invalid copy of `src` if another thread is concurrently modifying `src`.

5.7.6 ~concurrent_priority_queue()

Effects

Destroys all items in the queue, and the container itself, so that it can no longer be used.

5.7.7 bool empty() const

Returns

`true` if queue has no items; `false` otherwise. May be inaccurate when concurrent `push` or `try_pop` operations are pending. This operation reads shared data and may trigger a race condition in race detection tools when used concurrently.

5.7.8 size_type size() const

Returns

Number of items in the queue. May be inaccurate when concurrent `push` or `try_pop` operations are pending. This operation reads shared data and may trigger a race condition in race detection tools when used concurrently.

5.7.9 void push(const_reference elem)

Effects

Pushes a copy of `elem` into the queue. This operation is thread-safe with other `push` and `try_pop` operations.

5.7.10 bool try_pop(reference elem)

Effects

If the queue is not empty, copies the highest priority item from the queue and assigns it to `elem`, and destroys the popped item in the queue; otherwise, does nothing. This operation is thread-safe with other `push` and `try_pop` operations.

Returns

`true` if an item was popped; `false` otherwise.

5.7.11 void clear()

Effects

Clears the queue; results in `size()==0`. This operation is not thread-safe.

5.7.12 void swap(concurrent_priority_queue& other)

Effects

Swaps the queue contents with those of `other`. This operation is not thread-safe.

5.7.13 allocator_type get_allocator() const

Returns

Copy of allocator used to construct the queue.

5.8 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[11] allocator_type;
```

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

```
        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>
            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[12]
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();
iterator end();
const_iterator begin() const;
const_iterator end() const;
reverse_iterator rbegin();
reverse_iterator rend();
```

[12] 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.

```
        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.[13] 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 (5.8.3) and vector assignment (5.8.1) 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.

---

[13] 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.

5.8.1 Construction, Copy, and Assignment

Safety

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

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

Effects

Constructs empty vector using optionally specified allocator instance.

5.8.1.2 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.

5.8.1.3 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.

5.8.1.4 concurrent_vector(const concurrent_vector& src)

Effects

Constructs copy of *src*.

5.8.1.5 concurrent_vector& operator=(const concurrent_vector& src)

Effects

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

Returns

Reference to left hand side.

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

Assign contents of *src* to *this.

Returns

Reference to left hand side.

5.8.1.7 void assign(size_type n, const_reference t)

Assign `n` copies of `t`.

5.8.1.8 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.

5.8.2 Whole Vector Operations

Safety

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

5.8.2.1 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.

5.8.2.2 void shrink_to_fit()[14]

Effects

Compacts the internal representation to reduce fragmentation.

5.8.2.3 void swap(concurrent_vector& x)

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

5.8.2.4 void clear()

Effects

Erases all elements. Afterwards, `size()==0`. Does not free internal arrays.[15]

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

5.8.2.5 ~concurrent_vector()

Effects

Erases all elements and destroys the vector.

5.8.3 Concurrent Growth

Safety

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

---

[14] 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()`.

[15] 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()`.

5.8.3.1 iterator grow_by(size_type delta, const_reference t=T())[16]

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.

5.8.3.2 iterator grow_to_at_least(size_type n)[17]

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` (8.5). 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.

5.8.3.3 iterator push_back(const_reference value)[18]

Effects

Appends copy of `value` to the end of the vector.

---

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

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

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

Returns

Iterator that points to the copy.

5.8.4 Access

Safety

The methods described in this section may be concurrently invoked on the same vector as methods for concurrent growth (5.8.3). However, the returned reference may be to an element that is being concurrently constructed.

5.8.4.1 reference operator[](size_type index)

Returns

Reference to element with the specified index.

5.8.4.2 const_refrence operator[](size_type index) const

Returns

Const reference to element with the specified index.

5.8.4.3 reference at(size_type index)

Returns

Reference to element at specified index.

Throws

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

5.8.4.4 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.

5.8.4.5 reference front()

Returns

`(*this)[0]`

5.8.4.6 const_reference front() const

Returns

`(*this)[0]`

5.8.4.7 reference back()

Returns

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

5.8.4.8 const_reference back() const

Returns

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

5.8.5 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (5.1). 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`.

5.8.5.1 range_type range(size_t grainsize=1)

Returns

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

5.8.5.2 const_range_type range(size_t grainsize=1) const

Returns

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

5.8.6 Capacity

5.8.6.1 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 (5.8.3).

5.8.6.2 bool empty() const

Returns

`size()==0`

5.8.6.3 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.

5.8.6.4 size_type max_size() const

Returns

Highest possible size of the vector could reach.

5.8.7 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.

5.8.7.1 iterator begin()

Returns

`iterator` pointing to beginning of the vector.

5.8.7.2 const_iterator begin() const

Returns

`const_iterator` pointing to beginning of the vector.

5.8.7.3 iterator end()

Returns

`iterator` pointing to end of the vector.

5.8.7.4 const_iterator end() const

Returns

`const_iterator` pointing to end of the vector.

5.8.7.5 reverse_iterator rbegin()

Returns

`reverse iterator` pointing to beginning of reversed vector.

5.8.7.6 const_reverse_iterator rbegin() const

Returns

`const_reverse_iterator` pointing to beginning of reversed vector.

5.8.7.7 iterator rend()

Returns

`const_reverse_iterator` pointing to end of reversed vector.

5.8.7.8 const_reverse_iterator rend()

Returns

`const_reverse_iterator` pointing to end of reversed vector.

6 Flow Graph

There are some applications that best express dependencies as messages passed between nodes in a flow graph. These messages may contain data or simply act as signals that a predecessor has completed. The graph class and its associated node classes can be used to express such applications. All graph-related classes and functions are in the tbb::flow namespace.

Primary Components

There are 3 types of components used to implement a graph:

A `graph` object

Nodes

Edges

The `graph` object is the owner of the tasks created on behalf of the flow graph. Users can wait on the `graph` if they need to wait for the completion of all of the tasks related to the flow graph execution. One can also register external interactions with the `graph` and run tasks under the ownership of the flow graph.

Nodes invoke user-provided function objects or manage messages as the flow to/from other nodes. There are pre-defined nodes that buffer, filter, broadcast or order items as they flow through the graph.

Edges are the connections between the nodes, created by calls to the `make_edge` function.

Message Passing Protocol

In an Intel® TBB flow graph, edges dynamically switch between a push and pull protocol for passing messages. An Intel® TBB flow graph G = (V, S, L), where V is the set of nodes, S is the set of edges that are currently using a push protocol, and L is the set of edges that are currently using a pull protocol. For each edge (Vi, Vj), Vi is the predecessor / sender and Vj is the successor / receiver. When in the push set S, messages over an edge are initiated by the sender, which tries to *put* to the receiver. When in the pull set, messages are initiated by the receiver, which tries to *get* from the sender.

If a message attempt across an edge fails, the edge is moved to the other set. For example, if a put across the edge (Vi, Vj) fails, the edge is removed from the push set S and placed in the pull set L. This dynamic push/pull protocol is the key to performance in a non-preemptive tasking library such as Intel® TBB, where simply

repeating failed sends or receives is not an efficient option. Figure 4 summarizes this dynamic protocol.

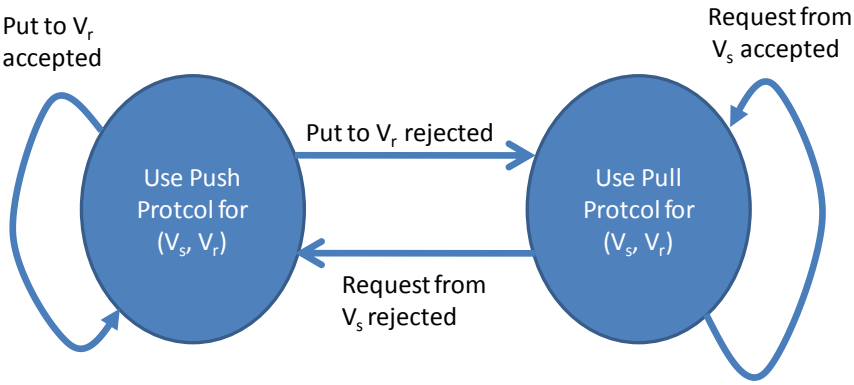

Figure 4: The dynamic push / pull protocol.

Body Objects

Some nodes execute user-provided body objects. These objects can be created by instatiating function objects or lambda expressions. The nodes that use body objects include `cotinue_node`, `function_node` and `source_node`.

CAUTION: The body objects passed to the flow graph nodes are copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the `copy_body` function described in 6.24 can be used to obtain an updated copy.

Dependency Flow Graph Example

```
#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct body {
    std::string my_name;
    body( const char *name ) : my_name(name) {}
    void operator()( continue_msg ) const {
        printf("%s\n", my_name.c_str());
    }
};

int main() {
    graph g;
```

```
    broadcast_node< continue_msg > start;
    continue_node<continue_msg> a( g, body("A"));
    continue_node<continue_msg> b( g, body("B"));
    continue_node<continue_msg> c( g, body("C"));
    continue_node<continue_msg> d( g, body("D"));
    continue_node<continue_msg> e( g, body("E"));

    make_edge( start, a );
    make_edge( start, b );
    make_edge( a, c );
    make_edge( b, c );
    make_edge( c, d );
    make_edge( a, e );

    for (int i = 0; i < 3; ++i ) {
        start.try_put( continue_msg() );
        g.wait_for_all();
    }

    return 0;
}
```

In this example, five computations A-E are setup with the partial ordering shown in Figure 5. For each edge in the flow graph, the node at the tail of the edge must complete its execution before the node at the head may begin.

NOTE: This is a simple syntactic example only. Since each node in a flow graph may execute as an independent task, the granularity of each node should follow the general guidelines for tasks as described in Section 3.2.3 of the Intel® Threading Building Blocks Tutorial.

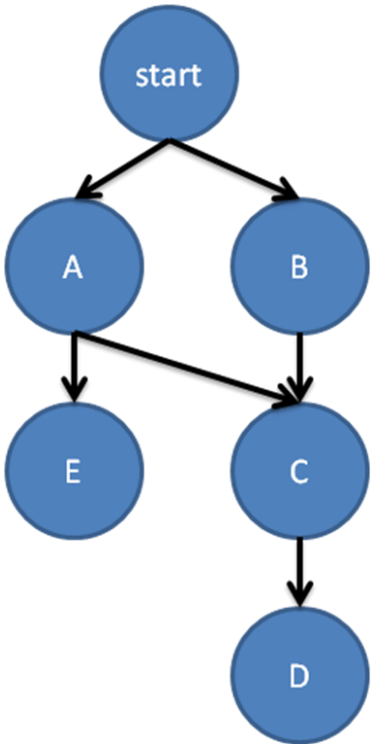

Figure 5: A simple dependency graph.

In this example, nodes A-E print out their names. All of these nodes are therefore able to use `struct body` to construct their body objects.

In function main, the flow graph is set up once and then run three times. All of the nodes in this example pass around `continue_msg` objects. This type is described in Section 6.4 and is used to communicate that a node has completed its execution.

The first line in function `main` instantiates a `graph` object, `g`. On the next line, a `broadcast_node` named `start` is created. Anything passed to this node will be broadcast to all of its successors. The node `start` is used in the `for` loop at the bottom of `main` to launch the execution of the rest of the flow graph.

In the example, five `continue_node` objects are created, named a – e. Each node is constructed with a reference to `graph g` and the function object to invoke when it runs. The successor / predecessor relationships are set up by the `make_edge` calls that follow the declaration of the nodes.

After the nodes and edges are set up, the `try_put` in each iteration of the `for` loop results in a broadcast of a `continue_msg` to both `a` and `b`. Both `a` and `b` are waiting for a single `continue_msg`, since they both have only a single predecessor, `start`.

When they receive the message from `start`, they execute their body objects. When complete, they each forward a `continue_msg` to their successors, and so on. The graph

uses tasks to execute the node bodies as well as to forward messages between the nodes, allowing computation to execute concurrently when possible.

The classes and functions used in this example are described in detail in the remaining sections in Appendix D.

Message Flow Graph Example

```
#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct square {
  int operator()(int v) { return v*v; }
};

struct cube {
  int operator()(int v) { return v*v*v; }
};

class sum {
  int &my_sum;
public:
  sum( int &s ) : my_sum(s) {}
  int operator()( std::tuple< int, int > v ) {
    my_sum +=  std::get<0>(v) + std::get<1>(v);
    return my_sum;
  }
};

int main() {
  int result = 0;

  graph g;
  broadcast_node<int> input;
  function_node<int,int> squarer( g, unlimited, square() );
  function_node<int,int> cuber( g, unlimited, cube() );
  join_node< std::tuple<int,int>, queueing > join( g );
  function_node<std::tuple<int,int>,int>
      summer( g, serial, sum(result) );

  make_edge( input, squarer );
  make_edge( input, cuber );
  make_edge( squarer, std::get<0>( join.inputs() ) );
  make_edge( cuber, std::get<1>( join.inputs() ) );
```

```
  make_edge( join, summer );

  for (int i = 1; i <= 10; ++i)
      input.try_put(i);
  g.wait_for_all();

  printf("Final result is %d\n", result);
  return 0;
}
```

This example calculates the sum of x*x + x*x*x for all x = 1 to 10.

NOTE: This is a simple syntactic example only. Since each node in a flow graph may execute as an independent task, the granularity of each node should follow the general guidelines for tasks as described in Section 3.2.3 of the Intel® Threading Building Blocks Tutorial.

The layout of this example is shown in Figure 6. Each value enters through the `broadcast_node<int> input`. This node broadcasts the value to both `squarer` and `cuber`, which calculate `x*x` and `x*x*x` respectively. The output of each of these nodes is put to one of `join`'s ports. A tuple containing both values is created by `join_node< tuple<int,int> > join` and forwarded to `summer`, which adds both values to the running total. Both `squarer` and `cuber` allow unlimited concurrency, that is they each may process multiple values simultaneously. The final `summer`, which updates a shared total, is only allowed to process a single incoming tuple at a time, eliminating the need for a lock around the shared value.

The classes `square`, `cube` and `sum` define the three user-defined operations. Each class is used to create a `function_node`.

In function `main`, the flow graph is setup and then the values 1 – 10 are put into the node `input`. All the nodes in this example pass around values of type `int`. The nodes used in this example are all class templates and therefore can be used with any type that supports copy construction, including pointers and objects.

CAUTION: Values are copied as they pass between nodes and therefore passing around large objects should be avoided. To avoid large copy overheads, pointers to large objects can be passed instead.

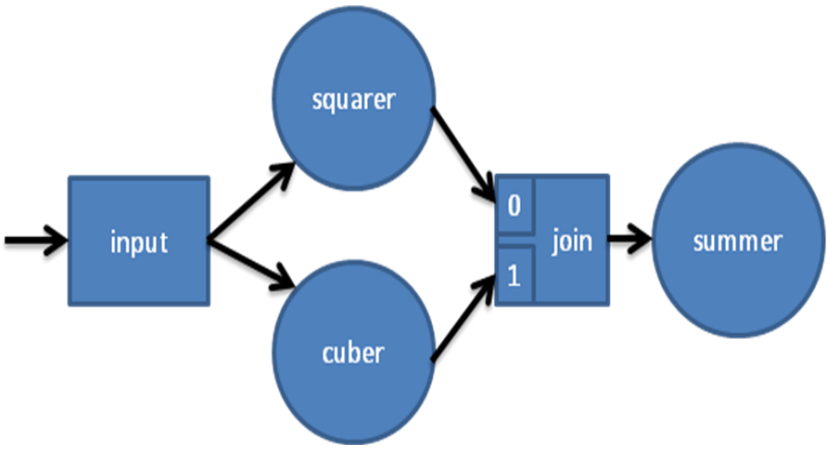

Figure 6: A simple message flow graph.

The classes and functions used in this example are described in detail in the remaining sections of Appendix D.

6.1 graph Class

Summary

Class that serves as a handle to a flow graph of nodes and edges.

Syntax

```
class graph;
```

Header

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

Description

A `graph` object contains a root task that is the parent of all tasks created on behalf of the flow graph and its nodes. It provides methods that can be used to access the root task, to wait for the children of the root task to complete, to explicitly increment or decrement the root task's reference count, and to run a task as a child of the root task.

CAUTION: Destruction of flow graph nodes before calling `wait_for_all` on their associated `graph` object has undefined behavior and can lead to program failure.

Members

```
namespace tbb {
namespace flow {
```

```
class graph {
public:

    graph();
    ~graph();

    void increment_wait_count();
    void decrement_wait_count();

    template< typename Receiver, typename Body >
    void run( Receiver &r, Body body );
    template< typename Body >
    void run( Body body );
    void wait_for_all();
    task * root_task();
};

}
}
```

6.1.1 graph()

Effects

Constructs a graph with no nodes. Instantiates a root task of class `empty_task` to serve as a parent for all of the tasks generated during runs of the graph. Sets `ref_count` of the root task to 1.

6.1.2 ~graph()

Effects

Calls `wait_for_`all on the graph, then destroys the root task.

6.1.3 void increment_wait_count()

Description

Used to register that an external entity may still interact with the graph.

Effects

Increments the `ref_count` of the root task.

6.1.4 void decrement_wait_count()

Description

Used to unregister an external entity that may have interacted with the graph.

Effects

Decrements the `ref_count` of the root task.

6.1.5 template< typename Receiver, typename Body > void run(Receiver &r, Body body)

Description

This method can be used to enqueue a task that runs a body and puts its output to a specific receiver. The task is created as a child of the graph's root task and therefore `wait_for_all` will not return until this task completes.

Effects

Enqueues a task that invokes `r.try_put( body() )`. It does not wait for the task to complete. The enqueued task is a child of the root task.

6.1.6 template< typename Body > void run(Body body)

Description

This method enqueues a task that runs as a child of the graph's root task. Calls to `wait_for_all` will not return until this enqueued task completes.

Effects

Enqueues a task that invokes `body()`. It does not wait for the task to complete.

6.1.7 void wait_for_all()

Effect

Blocks until all tasks associated with the root task have completed and the number of `decrement_wait_count` calls equals the number of `increment_wait_count` calls. Because it calls `wait_for_all` on the root graph task, the calling thread may participate in work-stealing while it is blocked.

6.1.8 task *root_task()

Retuns

Returns a pointer to the root task of the flow graph.

6.2 sender Template Class

Summary

An abstract base class for nodes that act as message senders.

Syntax

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

Header

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

Description

The `sender` template class is an abstract base class that defines the interface for nodes that can act as senders. Default implementations for several functions are provided.

Members

```
namespace tbb {
namespace flow {

template< typename T >
class sender {
public:
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    virtual ~sender();
    virtual bool register_successor( successor_type &r ) = 0;
    virtual bool remove_successor( successor_type &r ) = 0;
    virtual bool try_get( output_type & ) { return false; }
    virtual bool try_reserve( output_type & ) { return false; }
    virtual bool try_release( ) { return false; }
    virtual bool try_consume( ) { return false; }
};

}
}
```

6.2.1 ~sender()

Description

The destructor.

6.2.2 bool register_successor(successor_type & r) = 0

Description

A pure virtual method that describes the interface for adding a successor node to the set of successors for the sender.

Returns

True if the successor is added. False otherwise.

6.2.3 bool remove_successor(successor_type & r) = 0

Description

A pure virtual method that describes the interface for removing a successor node from the set of successors for a sender.

Returns

True if the successor is removed. False otherwise.

6.2.4 bool try_get(output_type &)

Description

Requests an item from a sender.

Returns

The default implementation returns false.

6.2.5 bool try_reserve(output_type &)

Description

Reserves an item at the sender.

Returns

The default implementation returns false.

6.2.6 bool try_release( )

Description

Releases the reservation held at the sender.

Returns

The default implementation returns false.

6.2.7 bool try_consume( )

Description

Consumes the reservation held at the sender.

Effect

The default implementation returns false.

6.3 receiver Template Class

Summary

An abstract base class for nodes that act as message receivers.

Syntax

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

Header

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

Description

The `receiver` template class is an abstract base class that defines the interface for nodes that can act as receivers. Default implementations for several functions are provided.

Members

```
namespace tbb {
namespace flow {

template< typename T >
class receiver {
public:
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    virtual ~receiver();
    virtual bool try_put( const input_type &v ) = 0;
    virtual bool register_predecessor( predecessor_type &p ) {
        return false; }
    virtual bool remove_predecessor( predecessor_type &p ) {
        return false; }
};

}
}
```

6.3.1 ~receiver()

Description

The destructor.

6.3.2 bool register_predecessor(predecessor_type & p)

Description

Adds a predecessor to the node's set of predecessors.

Returns

True if the predecessor is added. False otherwise. The default implementation returns false.

6.3.3 bool remove_predecessor(predecessor_type & p)

Description

Removes a predecessor from the node's set of predecessors.

Returns

True if the predecessor is removed. False otherwise. The default implementation returns false.

6.3.4 bool try_put(const input_type &v) = 0

Description

A pure virtual method that represents the interface for putting an item to a receiver.

6.4 continue_msg Class

Summary

An empty class that represent a continue message. This class is used to indicate that the sender has completed.

Syntax

```
class continue_msg;
```

Header

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

Members

```
namespace tbb { namespace flow { class continue_msg {}; } }
```

6.5 continue_receiver Class

Summary

An abstract base class for nodes that act as receivers of `continue_msg` objects. These nodes call a method `execute` when the number of `try_put` calls reaches a threshold that represents the number of known predecessors.

Syntax

```
class continue_receiver;
```

Header

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

Description

This type of node is triggered when its method `try_put` has been called a number of times that is equal to the number of known predecessors. When triggered, the node calls the method `execute`, then resets and will fire again when it receives the correct number of `try_put` calls. This node type is useful for dependency graphs, where each node must wait for its predecessors to complete before executing, but no explicit data is passed across the edge.

Members

```
namespace tbb {
namespace flow {

class continue_receiver : public receiver< continue_msg > {
public:
    typedef continue_msg input_type;
    typedef sender< input_type > predecessor_type;
    continue_receiver( int num_predecessors = 0 );
    continue_receiver( const continue_receiver& src );
    virtual ~continue_receiver();
    virtual bool try_put( const input_type &v );
    virtual bool register_predecessor( predecessor_type &p );
    virtual bool remove_predecessor( predecessor_type &p );

protected:
    virtual void execute() = 0;
};

}
}
```

6.5.1 continue_receiver(int num_predecessors = 0)

Effect

Constructs a `continue_receiver` that is initialized to trigger after receiving `num_predecessors` calls to `try_put`.

6.5.2 continue_receiver(const continue_receiver& src)

Effect

Constructs a `continue_receiver` that has the same initial state that `src` had after its construction. It does not copy the current count of `try_puts` received, or the current known number of predecessors. The `continue_receiver` that is constructed will only have a non-zero threshold if `src` was constructed with a non-zero threshold.

6.5.3 ~continue_receiver( )

Effect

Destructor.

6.5.4 bool try_put(const input_type &)

Effect

Increments the count of `try_put` calls received. If the incremented count is equal to the number of known predecessors, a call is made to `execute` and the internal count of `try_put` calls is reset to zero. This method performs as if the call to `execute` and the updates to the internal count occur atomically.

Returns

True.

6.5.5 bool register_predecessor(predecessor_type & r)

Effect

Increments the number of known predecessors.

Returns

True.

6.5.6 bool remove_predecessor(predecessor_type & r)

Effect

Decrements the number of know predecessors.

CAUTION: The method `execute` is not called if the count of `try_put` calls received becomes equal to the number of known predecessors as a result of this call. That is, a call to `remove_predecessor` will never call `execute`.

6.5.7 void execute() = 0

Description

A pure virtual method that is called when the number of `try_put` calls is equal to the number of known predecessors. Must be overridden by the child class.

CAUTION: This method should be very fast or else enqueue a task to offload its work, since this method is called while the sender is blocked on `try_put`.

6.6 graph_node Class

Summary

A base class for all graph nodes.

Syntax

```
class graph_node;
```

Header

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

Description

The class `graph_node` is a base class for all flow graph nodes. The virtual destructor allows flow graph nodes to be destroyed through pointers to `graph_node`. For example, a `vector< graph_node * >` could be used to hold the addresses of flow graph nodes that will later need to be destroyed.

Members

```
namespace tbb {
namespace flow {

class graph_node {
public:
    virtual ~graph_node() {}
};

}
}
```

6.7 continue_node Template Class

Summary

A template class that is a `graph_node`, `continue_receiver` and a `sender<T>`. It executes a specified body object when triggered and broadcasts the generated value to all of its successors.

Syntax

```
template< typename Output > class continue_node;
```

Header

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

Description

This type is used for nodes that wait for their predecessors to complete before executing, but no explicit data is passed across the incoming edges. The output of the node can be a `continue_msg` or a value.

An `continue_node` maintains an internal threshold, T, and an internal counter, C. If a value for the number of predecessors is provided at construction, then T is set to the provided value and C=0. Otherwise, C=T=0.

At each call to method `register_predecessor`, the threshold T is incremented. At each call to method `remove_predecessor`, the threshold T is decremented. The functions `make_edge` and `remove_edge` appropriately call `register_predecessor` and `remove_predecessor` when edges are added to or removed from a `continue_node`.

At each call to method `try_put`, C is incremented. If after the increment, C>=T, then C is reset to 0 and a task is enqueued to broadcast the result of `body()` to all successors. The increment of C, enqueueing of the task, and the resetting of C are all

done atomically with respect to the node. If after the increment, C<T, no additional action is taken.

The value generated by an execution of the body object is broadcast to all successors. Rejection of messages by successors is handled using the protocol in Figure 4.

A `continue_node` can serve as a terminal node in the graph. The convention is to use an `Output` of `continue_msg` and attach no successor.

The Body concept for `continue_node` is shown in Table 22.

Table 22: continue_node<Output> Body Concept

Pseudo-Signature	Semantics
`B::B( const B& )`	Copy constructor.
`B::~B()`	Destructor.
`void`[19] `operator=( const B& )`	Assignment
`Output B::operator()(const continue_msg &v) const`	Perform operation and return value of type Output.

CAUTION: The body object passed to a `continue_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the `copy_body` function described in 6.24 can be used to obtain an updated copy.

`Output` must be copy-constructible and assignable.

Members

```
namespace tbb {
namespace flow {

template< typename Output >
class continue_node :
  public graph_node, public continue_receiver,
  public sender<Output> {
public:
    template<typename Body>
    continue_node( graph &g, Body body );
    template<typename Body>
    continue_node( graph &g, int number_of_predecessors,
```

[19]The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored.

```
                        Body body );
    continue_node( const continue_node& src );

    // continue_receiver
    typedef continue_msg input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<Output>
    typedef Output output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.7.1 template< typename Body> continue_node(graph &g, Body body)

Effect

Constructs an `continue_node` that will invoke `body`.

6.7.2 template< typename Body> continue_node(graph &g, int number_of_predecessors, Body body)

Effect

Constructs an `continue_node` that will invoke `body`. The threshold T is initialized to `number_of_predecessors`.

6.7.3 continue_node(const continue_node & src)

Effect

Constructs a `continue_node` that has the same initial state that `src` had after its construction. It does not copy the current count of `try_puts` received, or the current known number of predecessors. The `continue_node` that is constructed will have a reference to the same `graph` object as `src`, have a copy of the initial `body` used by `src`, and only have a non-zero threshold if `src` was constructed with a non-zero threshold.

CAUTION: The new body object is copy constructed from a copy of the original body provided to `src` at its construction. Therefore changes made to member variables in `src`'s body after the construction of `src` will not affect the body of the new `continue_node`.

6.7.4 bool register_predecessor(predecessor_type & r)

Effect

Increments the number of known predecessors.

Returns

True.

6.7.5 bool remove_predecessor(predecessor_type & r)

Effect

Decrements the number of know predecessors.

CAUTION: The body is not called if the count of `try_put` calls received becomes equal to the number of known predecessors as a result of this call. That is, a call to `remove_predecessor` will never invoke the body.

6.7.6 bool try_put(const input_type &)

Effect

Increments the count of `try_put` calls received. If the incremented count is equal to the number of known predecessors, a task is enqueued to execute the `body` and the internal count of `try_put` calls is reset to zero. This method performs as if the

enqueueing of the body task and the updates to the internal count occur atomically. It does not wait for the execution of the body to complete.

Returns

True.

6.7.7 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

True.

6.7.8 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

True.

6.7.9 bool try_get(output_type &v)

Description

The `continue_node` does not contain buffering. Therefore it always rejects `try_get` calls.

Returns

False.

6.7.10 bool try_reserve(output_type &)

Description

The `continue_node` does not contain buffering. Therefore it cannot be reserved.

Returns

False.

6.7.11 bool try_release( )

Description

The `continue_node` does not contain buffering. Therefore it cannot be reserved.

Returns

False.

6.7.12 bool try_consume( )

Description

The `continue_node` does not contain buffering. Therefore it cannot be reserved.

Returns

False.

6.8 function_node Template Class

Summary

A template class that is a `graph_node`, `receiver<Input>` and a `sender<Output>`. This node may have concurrency limits as set by the user. By default, a `function_node` has an internal FIFO buffer at its input. Messages that cannot be immediately processed due to concurrency limits are temporarily stored in this FIFO buffer. A template argument can be used to disable this internal buffer. If the FIFO buffer is disabled, incoming message will be rejected if they cannot be processed immediately while respecting the concurreny limits of the node.

Syntax

```
template < typename Input,
           typename Output = continue_msg,
           graph_buffer_policy = queueing,
           typename Allocator=cache_aligned_allocator<Input> >
class function_node;
```

Header

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

Description

A `function_node` receives messages of type `Input` at a single input port and generates a single output message of type `Output` that is broadcast to all successors. Rejection of messages by successors is handled using the protocol in Figure 4.

If `graph_buffer_policy==queueing`, an internal unbounded input buffer is maintained using memory obtained through an allocator of type `Allocator`.

A `function_node` maintains an internal constant threshold T and an internal counter C. At construction, C=0 and T is set the value passed in to the constructor. The behavior of a call to `try_put` is determined by the value of T and C as shown in Table 23.

Table 23: Behavior of a call to a function_node's try_put

Value of threshold T	Value of counter C	bool try_put(input_type v)
`T == flow::unlimited`	NA	A task is enqueued that broadcasts the result of body(v) to all successors. Returns `true`.
`T != flow::unlimited`	C < T	Increments C. A task is enqueued that broadcasts the result of `body(v)` to all successors and then decrements C. Returns `true`.
`T != flow::unlimited`	C >= T	If the template argument `graph_buffer_policy==queueing`, `v` is stored in an internal FIFO buffer until C < T. When T becomes less than C, C is incremented and a task is enqueued that broadcasts the result of `body(v)` to all successors and then decrements C. Returns `true`. If the template argument `graph_buffer_policy==rejecting` and C >= T, returns `false`.

A `function_node` has a user-settable concurrency limit. It can have `flow::unlimited` concurrency, which allows an unlimited number of invocations of the body to execute concurrently. It can have `flow::serial` concurrency, which allows only a single call of body to execute concurrently. The user can also provide a value of type `size_t` to limit concurrency to a value between 1 and `unlimited`.

A `function_node` with `graph_buffer_policy==rejecting` will maintain a predecessor set as described in Figure 4. If the `function_node` transitions from a state where C >= T to a state where C < T, it will try to get new messages from its set of predecessors until C >= T or there are no valid predecessors left in the set.

NOTE: A `function_node` can serve as a terminal node in the graph. The convention is to use an `Output` of `continue_msg` and attach no successor.

The Body concept for `function_node is shown in` Table 24.

Table 24: function_node<InputType, OutputType> Body Concept

Pseudo-Signature	Semantics
`B::B( const B& )`	Copy constructor.
`B::~B()`	Destructor.
`void`[20] `operator=( const B& )`	Assignment
`Output B::operator()(const Input &v) const`	Perform operation on v and return value of type OutputType.

CAUTION: The body object passed to a `function_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the `copy_body` function described in 6.24 can be used to obtain an updated copy.

`Input` and `Output` must be copy-constructible and assignable.

Members

```
namespace tbb {
namespace flow {

enum graph_buffer_policy {
   rejecting, reserving, queueing, tag_matching };

template < typename Input, typename Output = continue_msg,
graph_buffer_policy = queueing, typename
Allocator=cache_aligned_allocator<Input> >
class function_node :
  public graph_node, public receiver<Input>,
  public sender<Output> {
public:
    template<typename Body>
    function_node( graph &g, size_t concurrency, Body body );
    function_node( const function_node &src );
```

[20]The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored.

```
    // receiver<Input>
    typedef Input input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<Output>
    typedef Output output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.8.1 template< typename Body> function_node(graph &g, size_t concurrency, Body body)

Description

Constructs a `function_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

6.8.2 function_node(const function_node &src)

Effect

Constructs a `function_node` that has the same initial state that `src` had when it was constructed. The `function_node` that is constructed will have a reference to the same `graph` object as `src`, will have a copy of the initial body used by `src`, and have the same concurrency threshold as `src`. The predecessors and successors of `src` will not be copied.

CAUTION: The new body object is copy constructed from a copy of the original body provided to `src` at its construction. Therefore changes made to member variables in `src`'s body after the construction of `src` will not affect the body of the new `function_node`.

6.8.3 bool register_predecessor(predecessor_type & p)

Effect

Adds p to the set of predecessors.

Returns

true.

6.8.4 bool remove_predecessor(predecessor_type & p)

Effect

Removes p from the set of predecessors.

Returns

true.

6.8.5 bool try_put(const input_type &v)

Effect

See Table 23 for a description of the behavior of `try_put`.

Returns

true.

6.8.6 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.8.7 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.8.8 bool try_get(output_type &v)

Description

A `function_node` does not contain buffering of its output. Therefore it always rejects `try_get` calls.

Returns

false.

6.8.9 bool try_reserve(output_type &)

Description

A `function_node` does not contain buffering of its output. Therefore it cannot be reserved.

Returns

false.

6.8.10 bool try_release( )

Description

A `function_node` does not contain buffering of its output. Therefore it cannot be reserved.

Returns

false.

6.8.11 bool try_consume( )

Description

A `function_node` does not contain buffering of its output. Therefore it cannot be reserved.

Returns

false.

6.9 source_node Class

Summary

A template class that is both a `graph_node` and a `sender<Output>`. This node can have no predecessors. It executes a user-provided `body` function object to generate messages that are broadcast to all successors. It is a serial node and will never call its `body` concurrently. It is able to buffer a single item. If no successor accepts an item that it has generated, the message is buffered and will be provided to successors before a new item is generated.

Syntax

```
template < typename OutputType > class source_node;
```

Header

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

Description

This type of node generates messages of type `Output` by invoking the user-provided `body` and broadcasts the result to all of its successors.

`Output` must be copy-constructible and assignable.

A `source_node` is a serial node. Calls to `body` will never be made concurrently.

A `source_node` will continue to invoke `body` and broadcast messages until the `body` returns `false` or it has no valid successors. A message may be generated and then rejected by all successors. In that case, the message is buffered and will be the next message sent once a successor is added to the node or `try_get` is called. Calls to `try_get` will return a buffer message if available or will invoke `body` to attempt to generate a new message. A call to `body` is made only when the internal buffer is empty.

Rejection of messages by successors is handled using the protocol in Figure 4.

Table 25: source_node<Output> Body Concept

Pseudo-Signature	Semantics
`B::B( const B& )`	Copy constructor.
`B::~B()`	Destructor.
`void`[21] `operator=( const B& )`	Assignment
`bool B::operator()(Output &v)`	Returns `true` when it has assigned a new value to `v`. Returns `false` when no new values may be generated.

CAUTION: The body object passed to a `source_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node.

`Output` must be copy-constructible and assignable.

Members

```
namespace tbb {
namespace flow {

template < typename Output >
class source_node : public graph_node, public sender< Output > {
public:
    typedef Output output_type;
    typedef receiver< output_type > successor_type;

    template< typename Body >
    source_node( graph &g, Body body, bool is_active = true );
    source_node( const source_node &src );
    ~source_node();

    void activate();
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type &v );
    bool try_release( );
    bool try_consume( );
};
```

[21]The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored.

```
}
}
```

6.9.1 template< typename Body> source_node(graph &g, Body body, bool is_active=true)

Description

Constructs a `source_node` that will invoke `body`. By default the node is created in the active state, that is, it will begin generating messages immediately. If `is_active` is false, messages will not be generated until a call to `activate` is made.

6.9.2 source_node(const source_node &src)

Description

Constructs a `source_node` that has the same initial state that `src` had when it was constructed. The `source_node` that is constructed will have a reference to the same `graph` object as `src`, will have a copy of the initial body used by `src`, and have the same initial active state as `src`. The predecessors and successors of `src` will not be copied.

CAUTION: The new body object is copy constructed from a copy of the original body provided to `src` at its construction. Therefore changes made to member variables in `src`'s body after the construction of `src` will not affect the body of the new `source_node`.

6.9.3 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.9.4 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.9.5 bool try_get(output_type &v)

Description

Will copy the buffered message into `v` if available or will invoke `body` to attempt to generate a new message that will be copied into `v`.

Returns

`true` if a message is copied to v. `false` otherwise.

6.9.6 bool try_reserve(output_type &v)

Description

Reserves the `source_node` if possible. If a message can be buffered and the node is not already reserved, the node is reserved for the caller and the value is copied into `v`.

Returns

`true` if the node is reserved for the caller. `false` otherwise.

6.9.7 bool try_release( )

Description

Releases any reservation held on the `source_node`. The message held in the internal buffer is retained.

Returns

true

6.9.8 bool try_consume( )

Description

Releases any reservation held on the `source_node` and clears the internal buffer.

Returns

True

6.10 multifunction_node Template Class

Summary

A template class that is a receiver<InputType> and has a tuple of sender<T> outputs. This node may have concurrency limits as set by the user. When the concurrency limit allows, it executes the user-provided `body` on incoming messages. The body may create one or more output messages and broadcast them to successors..

Syntax

```
template < typename InputType, typename OutputTuple >
class multifunction_node;
```

Header

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

Description

This type is used for nodes that receive messages at a single input port and may generate one or more messages that are broadcast to successors.

A `multifunction_node` maintains an internal constant threshold T and an internal counter C. At construction, C=0 and T is set the value passed in to the constructor. The behavior of a call to `try_put` is determined by the value of T and C as shown in .

Table 26: Behavior of a call to a multioutput_function_node's try_put

Value of threshold T	Value of counter C	bool try_put(input_type v)
`T == graph::unlimited`	NA	A task is enqueued that executes body(v). Returns `true`.
`T != flow::unlimited`	C < T	Increments C. A task is enqueued that executes body(v) and then decrements C. Returns `true`.
`T != flow::unlimited`	C >= T	Returns `false`.

A `multifunction_node` has a user-settable concurrency limit. It can have `flow::unlimited` concurrency, which allows an unlimited number of copies of the node to execute concurrently. It can have `flow::serial` concurrency, which allows only a single copy of the node to execute concurrently. The user can also provide a value of type `size_t` to limit concurrency to a value between 1 and unlimited.

The Body concept for `multifunction_node` is shown in .

Table 27: multifunction_node<InputType, OutputTuple> Body Concept

Pseudo-Signature	Semantics
`B::B( const B& )`	Copy constructor.
`B::~B()`	Destructor.
`void`[22] `operator=( const B& )`	Assignment
`void B::operator()(const InputType &v, output_ports &p)`	Perform operation on v. May call try_put on zero or more output_ports. May call try_put on output_ports multiple times..

Example

The example below shows a multifunction_node that separates a stream of integers into odd and even, placing each in the appropriate output queue.

The Body method will receive as parameters a read-only reference to the input value and a reference to the tuple of output ports. The Body method may put items to one or more output ports.

The output ports of the multifunction_node can be connected to other graph nodes using the `make_edge` method or by using `register_successor`:

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

using namespace tbb::flow;

typedef multifunction_node<int, std::tuple<int,int> > multi_node;

struct MultiBody {
```

```
    void operator()(const int &i, multi_node::output_ports_type
&op) {
        if(i % 2) std::get<1>(op).put(i);  // put to odd queue
        else      std::get<0>(op).put(i);  // put to even queue
    }
};

int main() {
    graph g;
    queue_node<int> even_queue(g);
    queue_node<int> odd_queue(g);
    multi_node node1(g,unlimited,MultiBody());
    output_port<0>(node1).register_successor(even_queue);
    make_edge(output_port<1>(node1), odd_queue);

    for(int i = 0; i < 1000; ++i) {
        node1.try_put(i);
    }
    g.wait_for_all();
}
```

Members

```
namespace tbb {

template< typename InputType, typename OutputTuple,
graph_buffer_policy=queueing, A>
class multifunction_node :
  public graph_node, public receiver<InputType>,
  {
public:
typedef (input_queue<InputType>) queue_type;
    template<typename Body>
    multifunction_node( graph &g, size_t concurrency, Body body,
queue_type *q = NULL );
    multifunction_node( const multifunction_node &other,
queue_type *q = NULL);
    ~multifunction_node();

    typedef InputType input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( input_type v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    typedef (tuple of sender<T...>) output_ports_type;
```

```
    template<size_t N, typename MFN> &output_port(MFN &node);
}
```

6.10.1 template< typename Body> multifunction_node(graph &g, size_t concurrency, Body body, queue_type *q = NULL)

Description

Constructs a `multifunction_node` that will invoke `body`. At most `concurrency` calls to the body may be made concurrently.

6.10.2 template< typename Body> multifunction_node(multifunction_node const & other, queue_type *q = NULL)

Effect

Constructs a copy of a multifunction_node with an optional input queue.

6.10.3 bool register_predecessor(predecessor_type & p)

Effect

Adds p to the set of predecessors.

Returns

true.

6.10.4 bool remove_predecessor(predecessor_type & p)

Effect

Removes p from the set of predecessors.

Returns

true.

6.10.5 bool try_put(input_type v)

Effect

If fewer copies of the node exist than the allowed concurrency, a task is spawned to execute `body` on the `v`. The body may put results to one or more successors in the tuple of output ports.

Returns

true.

6.10.6 *(output port &)* output_port<N>(node)

Returns

A reference to port `N` of the `multifunction_node` node.

6.11 overwrite_node Template Class

Summary

A template class that is a `graph_node`, `receiver<Input>` and `sender<Output>`. An `overwrite_node` is a buffer of a single item that can be over-written. The value held in the buffer is initially invalid. Gets from the node are non-destructive.

Syntax

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

Header

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

Description

This type of node buffers a single item of type T. The value is initially invalid. A `try_put` will set the value of the internal buffer, and broadcast the new value to all

successors. If the internal value is valid, a `try_get` will return true and copy the buffer value to the output. If the internal value is invalid, `try_get` will return false.

Rejection of messages by successors is handled using the protocol in Figure 4.

`T` must be copy-constructible and assignable

Members

```
namespace tbb {
namespace flow {

template< typename T >
class overwrite_node :
  public graph_node, public receiver<T>,
  public sender<T> {
public:
    overwrite_node(graph &g);
    overwrite_node( const overwrite_node &src );
    ~overwrite_node();

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );

    bool is_valid();
    void clear();
};

}
}
```

6.11.1 overwrite_node(graph &g)

Effect

Constructs an object of type `overwrite_node` with an invalid internal buffer item.

6.11.2 overwrite_node(const overwrite_node &src)

Effect

Constructs an object of type `overwrite_node` that belongs to the `graph g` with an invalid internal buffer item. The buffered value and list of successors is NOT copied from `src`.

6.11.3 ~overwrite_node()

Effect

Destroys the `overwrite_node`.

6.11.4 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.11.5 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.11.6 bool try_put(const input_type &v)

Effect

Stores `v` in the internal single item buffer. Calls `try_put( v )` on all successors.

Returns

true.

6.11.7 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors. If a valid item `v` is held in the buffer, a task is enqueued to call `r.try_put(v)`.

Returns

true.

6.11.8 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.11.9 bool try_get(output_type &v)

Description

If the internal buffer is valid, assigns the value to `v`.

Returns

`true` if `v` is assigned to. `false` if `v` is not assigned to.

6.11.10 bool try_reserve(output_type &)

Description

Does not support reservations.

Returns

false.

6.11.11 bool try_release( )

Description

Does not support reservations.

Returns

false.

6.11.12 bool try_consume( )

Description

Does not support reservations.

Returns

false.

6.11.13 bool is_valid()

Returns

Returns true if the buffer holds a valid value, otherwise returns false.

6.11.14 void clear()

Effect

Invalidates the value held in the buffer.

6.12 write_once_node Template Class

Summary

A template class that is a `graph_node, receiver<Input>` and `sender<Output>`. A `write_once_node` represents a buffer of a single item that cannot be over-written. The first put to the node sets the value. The value may be cleared explicitly, after which a new value may be set. Gets from the node are non-destructive.

Rejection of messages by successors is handled using the protocol in Figure 4.

`T` must be copy-constructible and assignable

Syntax

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

Header

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

Members

```
namespace tbb {
namespace flow {

template< typename T >
class write_once_node :
  public graph_node, public receiver<T>,
  public sender<T> {
public:
    write_once_node(graph &g);
    write_once_node( const write_once_node &src );

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
```

```
    bool try_release( );
    bool try_consume( );

    bool is_valid();
    void clear();
};

}
}
```

6.12.1 write_once_node(graph &g)

Effect

Constructs an object of type `write_once_node` that belongs to the `graph g`,with an invalid internal buffer item.

6.12.2 write_once_node(const write_once_node &src)

Effect

Constructs an object of type `write_once_node` with an invalid internal buffer item. The buffered value and list of successors is NOT copied from `src`.

6.12.3 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.12.4 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.12.5 bool try_put(const input_type &v)

Effect

Stores `v` in the internal single item buffer if it does not already contain a valid value. If a new value is set, it calls `try_put( v )` on all successors.

Returns

true.

6.12.6 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors. If a valid item `v` is held in the buffer, a task is enqueued to call `r.try_put(v)`.

Returns

true.

6.12.7 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.12.8 bool try_get(output_type &v)

Description

If the internal buffer is valid, assigns the value to `v`.

Returns

`true` if `v` is assigned to. `false` if `v` is not assigned to.

6.12.9 bool try_reserve(output_type &)

Description

Does not support reservations.

Returns

false.

6.12.10 bool try_release( )

Description

Does not support reservations.

Returns

false.

6.12.11 bool try_consume( )

Description

Does not support reservations.

Returns

false.

6.12.12 bool is_valid()

Returns

Returns true if the buffer holds a valid value, otherwise returns false.

6.12.13 void clear()

Effect

Invalidates the value held in the buffer.

6.13 broadcast_node Template Class

Summary

A node that broadcasts incoming messages to all of its successors.

Syntax

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

Header

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

Description

A broadcast_node is a `graph_node`, `receiver<T>` and `sender<T>` that broadcasts incoming messages of type `T` to all of its successors. There is no buffering in the node, so all messages are forwarded immediately to all successors.

Rejection of messages by successors is handled using the protocol in Figure 4.

`T` must be copy-constructible and assignable

Members

```
namespace tbb {
namespace flow {

template< typename T >
class broadcast_node :
  public graph_node, public receiver<T>, public sender<T> {
public:
    broadcast_node(graph &g);
    broadcast_node( const broadcast_node &src );

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );
```

```
    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.13.1 broadcast_node(graph &g)

Effect

Constructs an object of type `broadcast_node` that belongs to the `graph g`.

6.13.2 broadcast_node(const broadcast_node &src)

Effect

Constructs an object of type `broadtcast_node`. The list of successors is NOT copied from `src`.

6.13.3 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.13.4 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.13.5 bool try_put(const input_type &v)

Effect

Broadcasts `v` to all successors.

Returns

Always returns `true`, even if it was unable to successfully forward the message to any of its successors.

6.13.6 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.13.7 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.13.8 bool try_get(output_type &)

Returns

false.

6.13.9 bool try_reserve(output_type &)

Returns

false.

6.13.10 bool try_release( )

Returns

false.

6.13.11 bool try_consume( )

Returns

false.

6.14 buffer_node Class

Summary

An unbounded buffer of messages of type `T`. Messages are forwarded in arbitrary order.

Syntax

```
template< typename T, typename A=cache_aligned_allocator<T> >
class buffer_node;
```

Header

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

Description

A `buffer_node` is a `graph_node`, `receiver<T>` and `sender<T>` that forwards messages in arbitrary order to a single successor in its successor set. Successors are tried in the

order that they were registered with the node. If a successor rejects the message, it is removed from the successor list according to the policy in Figure 4 and the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. Items that are successfully transferred to a successor are removed from the buffer.

A `buffer_node` is reservable and supports a single reservation at a time. While an item is reserved, other items may still be forwarded to successors and `try_get` calls will return other non-reserved items if available. While an item is reserved, `try_put` will still return `true` and add items to the buffer.

An allocator of type `A` is used to allocate internal memory for the `buffer_node`.

`T` must be copy-constructible and assignable

Rejection of messages by successors is handled using the protocol in Figure 4.

Members

```
namespace tbb {
namespace flow {


template< typename T, typename A=cache_aligned_allocator<T> >
class buffer_node :
  public graph_node, public receiver<T>, public sender<T> {
public:
    buffer_node( graph &g );
    buffer_node( const buffer_node &src );

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};
```

```
}
}
```

6.14.1 buffer_node(graph& g)

Effect

Constructs an empty `buffer_node` that belongs to `graph g`.

6.14.2 buffer_node(const buffer_node &src)

Effect

Constructs an empty `buffer_node` that belongs to the same `graph g` as `src`. The list of predecessors, the list of successors and the messages in the buffer are NOT copied.

6.14.3 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.14.4 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.14.5 bool try_put(const input_type &v)

Effect

Adds `v` to the buffer. If v is the only item in the buffer, a task is also enqueued to forward the item to a successor.

Returns

true.

6.14.6 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.14.7 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.14.8 bool try_get(output_type & v)

Returns

Returns `true` if an item can be removed from the buffer and assigned to `v`. Returns `false` if there is no non-reserved item currently in the buffer.

6.14.9 bool try_reserve(output_type & v)

Effect

Assigns a newly reserved item to `v` if there is no reservation currently held and there is at least one item available in the buffer. If a new reservation is made, the buffer is marked as reserved.

Returns

Returns `true` if `v` has been assigned a newly reserved item. Returns `false` otherwise.

6.14.10 bool try_release( )

Effect

Releases the reservation on the buffer. The item that was returned in the last successful call to `try_reserve` remains in the buffer.

Returns

Returns `true` if the buffer is currently reserved and `false` otherwise.

6.14.11 bool try_consume( )

Effect

Releases the reservation on the buffer. The item that was returned in the last successful call to `try_reserve` is removed from the buffer.

Returns

Returns `true` if the buffer is currently reserved and `false` otherwise.

6.15 queue_node Template Class

Summary

An unbounded buffer of messages of type T. Messages are forwarded in first-in first-out (FIFO) order.

Syntax

```
template <typename T, typename A=cache_aligned_allocator<T> >
```

```
class queue_node;
```

Header

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

Description

A `queue_node` is a `graph_node`, `receiver<T>` and `sender<T>` that forwards messages in first-in first-out (FIFO) order to a single successor in its successor set. Successors are tried in the order that they were registered with the node. If a successor rejects the message, it is removed from the successor list as described by the policy in Figure 4 and the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. Items that are successfully transferred to a successor are removed from the buffer.

A `queue_node` is reservable and supports a single reservation at a time. While the `queue_node` is reserved, no other items will be forwarded to successors and all `try_get` calls will return `false`. While reserved, `try_put` will still return `true` and add items to the `queue_node`.

An allocator of type `A` is used to allocate internal memory for the `queue_node`.

`T` must be copy-constructible and assignable.

Rejection of messages by successors is handled using the protocol in Figure 4.

Members

```
namespace tbb {
namespace flow {


template <typename T, typename A=cache_aligned_allocator<T> >
class queue_node :
  public buffer_node<T,A> {
public:
    queue_node( graph &g );
    queue_node( const queue_node &src );

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( input_type v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
```

```
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.15.1 queue_node(graph& g)

Effect

Constructs an empty `queue_node` that belongs to `graph g`.

6.15.2 queue_node(const queue_node &src)

Effect

Constructs an empty `queue_node` that belongs to the same `graph g` as `src`. The list of predecessors, the list of successors and the messages in the buffer are NOT copied.

6.15.3 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.15.4 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.15.5 bool try_put(const input_type &v)

Effect

Adds `v` to the `queue_node`. If `v` is the only item in the `queue_node`, a task is enqueued to forward the item to a successor.

Returns

true.

6.15.6 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.15.7 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.15.8 bool try_get(output_type & v)

Returns

Returns `true` if an item can be removed from the front of the `queue_node` and assigned to `v`. Returns `false` if there is no item currently in the `queue_node` or if the node is reserved.

6.15.9 bool try_reserve(output_type & v)

Effect

If the call returns `true`, the node is reserved and will forward no more messages until the reservation has been released or consumed.

Returns

Returns `true` if there is an item in the `queue_node` and the node is not currently reserved. If an item can be returned, it is assigned to `v`. Returns `false` if there is no item currently in the `queue_node` or if the node is reserved.

6.15.10 bool try_release( )

Effect

Release the reservation on the node. The item that was returned in the last successful call to `try_reserve` remains in the `queue_node`.

Returns

Returns `true` if the node is currently reserved and `false` otherwise.

6.15.11 bool try_consume( )

Effect

Releases the reservation on the `queue_node`. The item that was returned in the last successful call to `try_reserve` is popped from the front of the queue.

Returns

Returns `true` if the `queue_node` is currently reserved and `false` otherwise.

6.16 priority_queue_node Template Class

Summary

An unbounded buffer of messages of type `T`. Messages are forwarded in priority order.

Syntax

```
template< typename T,
```

```
          typename Compare = std::less<T>,
          typename A=cache_aligned_allocator<T> >
class priority_queue_node;
```

Header

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

Description

A `priority_queue_node` is a `graph_node`, `receiver<T>` and `sender<T>` that forwards messages in priority order to a single successor in its successor set. Successors are tried in the order that they were registered with the node. If a successor rejects the message, it is removed from the successor list as described by the policy in Figure 4 and the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. Items that are successfully transferred to a successor are removed from the buffer.

The next message to be forwarded has the largest priority as determined by `Compare`.

A `priority_queue_node` is reservable and supports a single reservation at a time. While the `priority_queue_node` is reserved, no other items will be forwarded to successors and all `try_get` calls will return `false`. While reserved, `try_put` will still return `true` and add items to the `priority_queue_node`.

An allocator of type `A` is used to allocate internal memory for the `priority_queue_node`.

`T` must be copy-constructible and assignable.

Rejection of messages by successors is handled using the protocol in Figure 4.

Members

```
namespace tbb {
namespace flow {

template< typename T, typename Compare = std::less<T>,
          typename A=cache_aligned_allocator<T>>
class priority_queue_node : public queue_node<T> {
public:
    typedef size_t size_type;
    priority_queue_node( graph &g );
    priority_queue_node( const priority_queue_node &src );
    ~priority_queue_node();

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
```

```
    bool try_put( input_type v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.16.1 priority_queue_node(graph& g)

Effect

Constructs an empty `priority_queue_node` that belongs to `graph g`.

6.16.2 priority_queue_node(const priority_queue_node &src)

Effect

Constructs an empty `priority_queue_node` that belongs to the same `graph g` as `src`. The list of predecessors, the list of successors and the messages in the buffer are NOT copied.

6.16.3 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.16.4 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.16.5 bool try_put(const input_type &v)

Effect

Adds `v` to the `priority_queue_node`. If `v`'s priority is the largest of all of the currently buffered messages, a task is enqueued to forward the item to a successor.

Returns

true.

6.16.6 bool register_successor(successor_type &r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.16.7 bool remove_successor(successor_type &r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.16.8 bool try_get(output_type & v)

Returns

Returns `true` if a message is available in the node and the node is not currently reserved. Otherwise returns `false`. If the node returns `true`, the message with the largest priority will have been copied to v.

6.16.9 bool try_reserve(output_type & v)

Effect

If the call returns `true`, the node is reserved and will forward no more messages until the reservation has been released or consumed.

Returns

Returns `true` if a message is available in the node and the node is not currently reserved. Otherwise returns `false`. If the node returns `true`, the message with the largest priority will have been copied to v.

6.16.10 bool try_release( )

Effect

Release the reservation on the node. The item that was returned in the last successful call to `try_reserve` remains in the `priority_queue_node`.

Returns

Returns true if the buffer is currently reserved and false otherwise.

6.16.11 bool try_consume( )

Effect

Releases the reservation on the node. The item that was returned in the last successful call to `try_reserve` is removed from the `priority_queue_node`.

Returns

Returns true if the buffer is currently reserved and false otherwise.

6.17 sequencer_node Template Class

Summary

An unbounded buffer of messages of type `T`. Messages are forwarded in sequence order.

Syntax

```
template< typename T, typename A=cache_aligned_allocator<T> >
class sequencer_node;
```

Header

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

Description

A `sequencer_node` is a `graph_node`, `receiver<T>` and `sender<T>` that forwards messages in sequence order to a single successor in its successor set. Successors are tried in the order that they were registered with the node. If a successor rejects the message, it is removed from the successor list as described by the policy in Figure 4 and the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. Items that are successfully transferred to a successor are removed from the buffer.

Each item that passes through a sequencer_node is ordered by its sequencer order number. These sequence order numbers range from 0 … N, where N is the largest integer representable by the `size_t` type. An item's sequencer order number is determined by passing the item to a user-provided function object that models the Sequencer Concept shown in Table 28.

Table 28: sequencer_node<T> Sequencer Concept

Pseudo-Signature	Semantics
`S::S( const S& )`	Copy constructor.
`S::~S()`	Destructor.
`void`[23] `operator=( const S& )`	Assignment
`size_t S::operator()( const T &v )`	Returns the sequence number for the provided message `v`.

[23]The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored.

A `sequencer_node` is reservable and supports a single reservation at a time. While a `sequencer_node` is reserved, no other items will be forwarded to successors and all `try_get` calls will return `false`. While reserved, `try_put` will still return `true` and add items to the sequencer_node.

An allocator of type `A` is used to allocate internal memory for the `sequencer_node`.

`T` must be copy-constructible and assignable.

Rejection of messages by successors is handled using the protocol in Figure 4.

Members

```
namespace tbb {
namespace flow {

template< typename T, typename A=cache_aligned_allocator<T> >
class sequencer_node :
  public queue_node<T> {
public:
    template< typename Sequencer >
    sequencer_node( graph &g, const Sequencer& s );
    sequencer_node( const sequencer_node &src );

    // receiver<T>
    typedef T input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( input_type v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.17.1 template<typename Sequencer> sequencer_node(graph& g, const Sequencer& s)

Effect

Constructs an empty `sequencer_node` that belongs to `graph g` and uses `s` to compute sequence numbers for items.

6.17.2 sequencer_node(const sequencer_node &src)

Effect

Constructs an empty `sequencer_node` that belongs to the same `graph g` as `src` and will use a copy of the Sequencer `s` used to construct `src`. The list of predecessors, the list of successors and the messages in the buffer are NOT copied.

CAUTION: The new Sequencer object is copy constructed from a copy of the original Sequencer object provided to `src` at its construction. Therefore changes made to member variables in `src's` object will not affect the Sequencer of the new `sequencer_node`.

6.17.3 bool register_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.17.4 bool remove_predecessor(predecessor_type &)

Description

Never rejects puts and therefore does not need to maintain a list of predecessors.

Returns

false.

6.17.5 bool try_put(input_type v)

Effect

Adds `v` to the `sequencer_node`. If `v`'s sequence number is the next item in the sequence, a task is enqueued to forward the item to a successor.

Returns

true.

6.17.6 bool register_successor(successor_type &r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.17.7 bool remove_successor(successor_type &r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.17.8 bool try_get(output_type & v)

Returns

Returns `true` if the next item in the sequence is available in the `sequencer_node`. If so, it is removed from the node and assigned to `v`. Returns `false` if the next item in sequencer order is not available or if the node is reserved.

6.17.9 bool try_reserve(output_type & v)

Effect

If the call returns `true`, the node is reserved and will forward no more messages until the reservation has been released or consumed.

Returns

Returns `true` if the next item in sequencer order is available in the `sequencer_node`. If so, the item is assigned to `v`, but is not removed from the `sequencer_node` Returns `false` if the next item in sequencer order is not available or if the node is reserved.

6.17.10 bool try_release( )

Effect

Releases the reservation on the node. The item that was returned in the last successful call to `try_reserve` remains in the `sequencer_node`.

Returns

Returns `true` if the buffer is currently reserved and `false` otherwise.

6.17.11 bool try_consume( )

Effect

Releases the reservation on the node. The item that was returned in the last successful call to `try_reserve` is removed from the `sequencer_node`.

Returns

Returns `true` if the buffer is currently reserved and `false` otherwise.

6.18 limiter_node Template Class

Summary

An node that counts and limits the number of messages that pass through it.

Syntax

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

Header

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

Description

A `limiter_node` is a `graph_node`, `receiver<T>` and `sender<T>` that broadcasts messages to all of its successors. It keeps a counter C of the number of broadcasts it makes and does not accept new messages once its user-specified threshold T is reached. The internal count of broadcasts C can be decremented through use of its embedded `continue_receiver decrement`.

The behavior of a call to a `limiter_node's` try_put is shown in Table 29.

Table 29: Behavior of a call to a limiter_node's try_put

Value of counter C	bool try_put(input_type v)
C < T	C is incremented and `v` is broadcast to all successors. If no successor accepts the message, C is decremented. Returns `true` if the message was successfully broadcast to at least one successor and `false` otherwise.
C == T	Returns `false`.

When `try_put` is called on the member object decrement, the `limiter_node` will try to get a message from one of its known predecessors and forward that message to all of its successors. If it cannot obtain a message from a predecessor, it will decrement C. Rejection of messages by successors and failed gets from predecessors are handled using the protocol in Figure 4.

`T` must be copy-constructible and assignable.

Members

```
namespace tbb {
namespace flow {

template< typename T >
class limiter_node : public graph_node, public receiver<T>,
  public sender<T> {
public:
    limiter_node( graph &g, size_t threshold,
                  int number_of_decrement_predecessors = 0 );
    limiter_node( const limiter_node &src );

    // a continue_receiver
    implementation-dependent-type decrement;

    // receiver<T>
    typedef T input_type;
```

```
    typedef sender<input_type> predecessor_type;
    bool try_put( const input_type &v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );

    // sender<T>
    typedef T output_type;
    typedef receiver<output_type> successor_type;
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

6.18.1 limiter_node(graph &g, size_t threshold, int number_of_decrement_predecessors)

Description

Constructs a `limiter_node` that allows up to `threshold` items to pass through before rejecting `try_puts`. Optionally a `number_of_decrement_predecessors` value can be supplied. This value is passed on to the `continue_receiver decrement's` constructor.

6.18.2 limiter_node(const limiter_node &src)

Description

Constructs a `limiter_node` that has the same initial state that `src` had at its construction. The new `limiter_node` will belong to the same `graph g` as `src`, have the same `threshold`, and have the same initial `number_of_decrement_predecessors`. The list of predecessors, the list of successors and the current count of broadcasts, C, are NOT copied from `src`.

6.18.3 bool register_predecessor(predecessor_type& p)

Description

Adds a predecessor that can be pulled from once the broadcast count falls below the threshold.

Effect

Adds `p` to the set of predecessors.

Returns

true.

6.18.4 bool remove_predecessor(predecessor_type & r)

Effect

Removes `p` to the set of predecessors.

Returns

true.

6.18.5 bool try_put(input_type &v)

Effect

If the broadcast count is below the threshold, `v` is broadcast to all successors. For each successor `s`, if `s.try_put( v ) == false && s.register_predecessor( *this ) == true`, then `s` is removed from the set of succesors. Otherwise, `s` will remain in the set of successors.

Returns

`true` if `v` is broadcast. `false` if `v` is not broadcast because the threshold has been reached.

6.18.6 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.18.7 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.18.8 bool try_get(output_type &)

Description

Does not contain buffering and therefore cannot be pulled from.

Returns

false.

6.18.9 bool try_reserve(output_type &)

Description

Does not support reservations.

Returns

false.

6.18.10 bool try_release( )

Description

Does not support reservations.

Returns

false.

6.18.11 bool try_consume( )

Description

Does not support reservations.

Returns

false.

6.19 join_node Template Class

Summary

A node that creates a tuple<T0,T1,…> from a set of messages received at its input ports and broadcasts the tuple to all of its successors. The `class join_node` supports three buffering policies at its input ports: `reserving`, `queueing` and `tag_matching`. By default, `join_node` input ports use the `queueing` policy.

Syntax

```
template<typename OutputTuple, graph_buffer_policy JP=queueing>
class join_node;
```

Header

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

Description

A join_node is a `graph_node` and a `sender< std::tuple< T0, T1, … >`. It contains a tuple of input ports, each of which is a `receiver<Ti>` for each of the `T0` .. `TN` in `OutputTuple`. It supports multiple input receivers with distinct types and broadcasts a tuple of received messages to all of its successors. All input ports of a `join_node` must use the same buffering policy. The behavior of a `join_node` based on its buffering policy is shown in Table 30.

Table 30: Behavior of a join_node based on the buffering policy of its input ports.

Buffering Policy	Behavior
queueing	As each input port is put to, the incoming message is added to an unbounded first-in first-out queue in the port. When there is at least one message at each input port, the `join_node` broadcasts a tuple containing the head of each queue to all successors. If at least one successor accepts the tuple, the head of each input port's queue is removed, otherwise the messages remain in their respective input port queues.
reserving	As each input port is put to, the `join_node` marks that an input may be available at that port and returns `false`. When all ports have been marked as possibly available, the `join_node` will try to reserve a message at each port from their known predecessors. If it is unable to reserve a message at a port, it un-marks that port, and releases all previously acquired reservations. If it is able to reserve a message at all ports, it broadcasts a tuple containing these messages to all successors. If at least one successor accepts the tuple, the reservations are consumed; otherwise, they are released.
tag_matching	As each input port is put to, a user-provided function object is applied to the message to obtain its tag. The message is then added to a hash table at the input port, using the tag as the key. When there is message at each input port for a given tag, the `join_node` broadcasts a tuple containing the matching messages to all successors. If at least one successor accepts the tuple, the messages are removed from each input port's hash table; otherwise, the messages remain in their respective input ports. If an input's tag matches one already stored in a join node's input port, the try_put() will fail and return false.

Rejection of messages by successors of the `join_node` and failed gets from predecessors of the input ports are handled using the protocol in Figure 4.

The function template `input_port` described in 6.21 simplifies the syntax for getting a reference to a specific input port.

`OutputTuple` must be a `std::tuple<T0,T1,…>` where each element is copy-constructible and assignable.

Example

```
#include<cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

int main() {
   graph g;
```

```
    function_node<int,int> f1( g, unlimited,
        [](const int &i) { return 2*i; } );
    function_node<float,float> f2( g, unlimited,
        [](const float &f) { return f/2; } );

    join_node< std::tuple<int,float> > j(g);

    function_node< std::tuple<int,float> >
        f3( g, unlimited,
        []( const std::tuple<int,float> &t ) {
            printf("Result is %f\n",
                std::get<0>(t) + std::get<1>(t));});

    make_edge( f1, input_port<0>(j) );
    make_edge( f2, input_port<1>(j) );
    make_edge( j, f3 );

    f1.try_put( 3 );
    f2.try_put( 3 );
    g.wait_for_all();
    return 0;
}
```

In the example above, three `function_node` objects are created: `f1` multiplies an `int i` by 2, `f2` divides a `float f` by 2, and f3 receives a `std::tuple<int,float> t`, adds its elements together and prints the result. The `join_node j` combines the output of `f1` and `f2` and forwards the resulting tuple to `f3`. This example is purely a syntactic demonstration since there is very little work in the nodes.

Members

```
namespace tbb {
namespace flow {

enum graph_buffer_policy {
    rejecting, reserving, queueing, tag_matching };

template<typename OutputTuple, graph_buffer_policy JP=queueing>
class join_node :
    public graph_node, public sender< OutputTuple > {

public:
    typedef OutputTuple output_type;
    typedef receiver<output_type> successor_type;
    implementation-dependent-tuple input_ports_type;
```

```
    join_node(graph &g);
    join_node(const join_node &src);
    input_ports_type &inputs();
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );

};

//
// Specialization for tag_matching
//

template<typename OutputTuple>
class join_node<OutputTuple, tag_matching> :
    public graph_node, public sender< OutputTuple > {

public:

    // Has the same methdods as previous join_node,
    // but has constructors to specify the tag_matching
    // function objects

    template<typename B0, typename B1>
    join_node(graph &g, B0 b0, B1 b1);

    // Constructors are defined similarly for
    // 3 through 10 elements …
};

}
}
```

6.19.1 join_node(graph &g)

Effect

Creates a `join_node` that will enqueue tasks using the root task in `g`.

6.19.2 template < typename B0, typename B1, ... > join_node(graph &g, B0 b0, B1 b1, ...)

Description

A constructor only available in the `tag_matching` specialization of `join_node`.

Effect

Creates a `join_node` that uses the function objects b0, b1, …, bN to determine that tags for the input ports 0 through N. It will enqueue tasks using the root task in `g`.

6.19.3 join_node(const join_node &src)

Effect

Creates a `join_node` that has the same initial state that `src` had at its construction. The list of predecessors, messages in the input ports, and successors are NOT copied.

6.19.4 input_ports_type& input_ports()

Returns

A `std::tuple` of receivers. Each element inherits from `tbb::receiver<T>` where `T` is the type of message expected at that input. Each tuple element can be used like any other `flow::receiver<T>`. The behavior of the ports based on the selected `join_node` policy is shown in Table 30.

6.19.5 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

6.19.6 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

6.19.7 bool try_get(output_type &v)

Description

Attempts to generate a tuple based on the buffering policy of the `join_node`.

Returns

If it can successully generate a tuple, it copies it to v and returns `true`. Otherwise it returns `false`.

6.19.8 bool try_reserve(T &)

Description

A `join_node` cannot be reserved.

Returns

false.

6.19.9 bool try_release( )

Description

A `join_node` cannot be reserved.

Returns

false.

6.19.10 bool try_consume( )

Description

A `join_node` cannot be reserved.

Returns

false.

6.19.11 template<size_t N, typename JNT> typename std::tuple_element<N, typename JNT::input_ports_type>::type &input_port(JNT &jn)

Description

Equivalent to calling std::get<N>(jn.input_ports())

Returns

Returns the N^{th} input port for `join_node jn`.

6.20 split_node Template Class

Summary

A template class that is a `receiver<InputTuple>` and has a tuple of `sender<T>` outputs. A `split_node` is a `multifunction_node` with a body that sends each element of the incoming tuple to the output port that matches the element's index in the incoming tuple. This node has unlimited concurrency.

Syntax

```
template < typename InputType >
class split_node;
```

Header

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

Description

This type is used for nodes that receive tuples at a single input port and generate a message from each element of the tuple, passing each to its corresponding output port.

A `split_node` has unlimited concurrency, no buffering, and behaves as a `broadcast_node` with multiple output ports.

Example

The example below shows a `split_node` that separates a stream of tuples of integers, placing each element of the tuple in the appropriate output queue.

The output ports of the `split_node` can be connected to other graph nodes using the `make_edge` method or by using `register_successor`:

```
#define TBB_PREVIEW_GRAPH_NODES 1
#include "tbb/flow_graph.h"

using namespace tbb::flow;

typedef split_node< std::tuple<int,int> > s_node;

int main() {
    typedef std::tuple<int,int> int_tuple_type;
    graph g;
    queue_node<int> first_queue(g);
    queue_node<int> second_queue(g);
    s_node node1(g);
    output_port<0>(node1).register_successor(first_queue);
    make_edge(output_port<1>(node1), second_queue);

    for(int i = 0; i < 1000; ++i) {
        node1.try_put(int_tuple_type(2*i,2*i+1));
    }
    g.wait_for_all();
}
```

Members

```
namespace tbb {

template< typename InputType, A >
class split_node :
  public multifunction_node<InputType,InputType,rejecting,A>
  {
public:

    split_node( graph &g);
    split_node( const split_node &other);
    ~split_node();
```

```
    typedef InputType input_type;
    typedef sender<input_type> predecessor_type;
    bool try_put( input_type v );
    bool register_predecessor( predecessor_type &p );
    bool remove_predecessor( predecessor_type &p );


    typedef (tuple of sender<T...>) output_ports_type;

    template<size_t N, typename MFN> &output_port(MFN &node);
}
```

6.20.1 split_node(graph &g)

Description

Constructs a `split_node`.

6.20.2 split_node(split_node const & other)

Effect

Constructs a copy of a `split_node`.

6.20.3 bool register_predecessor(predecessor_type & p)

Effect

Adds `p` to the set of predecessors.

Returns

true.

6.20.4 bool remove_predecessor(predecessor_type & p)

Effect

Removes `p` from the set of predecessors.

Returns

true.

6.20.5 bool try_put(input_type v)

Effect

Forwards each element of the input tuple v to the corresponding output port.

Returns

true.

6.20.6 *(output port &)* output_port<N>(node)

Returns

A reference to port `N` of the `split_node`.

6.21 input_port Template Function

Summary

A template function that given a `join_node` or `or_node` returns a reference to a specific input port.

Syntax

```
template<size_t N, typename NT>
typename std::tuple_element<N,
                   typename NT::input_ports_type>::type&
input_port(NT &n);
```

Header

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

6.22 make_edge Template Function

Summary

A template function that adds an edge between a `sender<T>` and a `receiver<T>`.

Syntax

```
template< typename T >
inline void make_edge( sender<T> &p, receiver<T> &s );
```

Header

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

6.23 remove_edge Template Function

Summary

A template function that removes an edge between a `sender<T>` and a `receiver<T>`.

Syntax

```
template< typename T >
void remove_edge( sender<T> &p, receiver<T> &s );
```

Header

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

6.24 copy_body Template Function

Summary

A template function that returns a copy of the body function object from a `continue_node` or `function_node`.

Syntax

```
template< typename Body, typename Node >
Body copy_body( Node &n );
```

Header

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

7 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.

7.1 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;
```

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);
    };
}
```

7.1.1 combinable()

Effects

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

7.1.2 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.

7.1.3 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.

7.1.4 ~combinable()

Effects

Destroy all thread-local elements in `*this`.

7.1.5 combinable& operator=(const combinable& other)

Effects

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

7.1.6 void clear()

Effects

Remove all elements from `*this`.

7.1.7 T& local()

Effects

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

Returns

Reference to thread-local element.

7.1.8 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.

7.1.9 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.

7.1.10 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`.

7.2 enumerable_thread_specific Template Class

Summary

Template class for thread local storage.

Syntax

```
enum ets_key_usage_type {
    ets_key_per_instance,
    ets_no_key
};

template <typename T,
```

```
          typename Allocator=cache_aligned_allocator<T>,
          ets_key_usage_type ETS_key_type=ets_no_key>
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_key` 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_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_no_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 &other
        );
        template<typename U, typename Alloc,
            ets_key_usage_type Cachetype>
        enumerable_thread_specific(
            const enumerable_thread_specific<U, Alloc,
                                             Cachetype>& other );
        template <typename Finit>
        enumerable_thread_specific( Finit finit );
        enumerable_thread_specific(const T &exemplar);
        ~enumerable_thread_specific();
        enumerable_thread_specific&
        operator=(const enumerable_thread_specific& other);
        template<typename U, typename Alloc,
                 ets_key_usage_type Cachetype>
        enumerable_thread_specific&
        operator=(
            const enumerable_thread_specific<U, Alloc, Cachetype>&
                other
        );
        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;
    };
}
```

7.2.1 Whole Container Operations

Safety

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

7.2.1.1 enumerable_thread_specific()

Effects

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

7.2.1.2 enumerable_thread_specific(const enumerable_thread_specific &other)

Effects

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

7.2.1.3 template<typename U, typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific(const enumerable_thread_specific<U, Alloc, Cachetype>& other)

Effects

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

7.2.1.4 template< typename Finit> enumerable_thread_specific(Finit finit)

Effects

Constructs `enumerable_thread_specific` 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.

7.2.1.5 enumerable_thread_specific(const &exemplar)

Effects

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

7.2.1.6 ~enumerable_thread_specific()

Effects

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

7.2.1.7 enumerable_thread_specific& operator=(const enumerable_thread_specific& other);

Effects

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

7.2.1.8 template< typename U, typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific& operator=(const enumerable_thread_specific<U, Alloc, Cachetype>& other);

Effects

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

NOTE: The allocator and key usage specialization is unchanged by this call.

7.2.1.9 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`.

7.2.2 Concurrent Operations

7.2.2.1 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.

7.2.2.2 reference 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.

7.2.2.3 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.

7.2.2.4 bool empty() const

Returns

`size()==0`

7.2.3 Combining

The methods in this section iterate across the entire container.

7.2.3.1 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.

7.2.3.2 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`.

7.2.4 Parallel Iteration

Types `const_range_type` and `range_type` model the Container Range concept (5.1). 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.

7.2.4.1 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.

7.2.4.2 range_type range(size_t grainsize=1)

Returns

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

7.2.5 Iterators

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

7.2.5.1 iterator begin()

Returns

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

7.2.5.2 iterator end()

Returns

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

7.2.5.3 const_iterator begin() const

Returns

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

7.2.5.4 const_iterator end() const

Returns

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

7.3 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);
}
```

7.3.1 Whole Container Operations

Safety

These operations must not be invoked concurrently on the same `flattend2d`.

7.3.1.1 flattened2d(const Container& c)

Effects

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

7.3.1.2 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.

7.3.2 Concurrent Operations

Safety

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

7.3.2.1 size_type size() const

Returns

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

7.3.3 Iterators

Template class `flattened2d` supports foward iterators only.

7.3.3.1 iterator begin()

Returns

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

7.3.3.2 iterator end()

Returns

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

7.3.3.3 const_iterator begin() const

Returns

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

7.3.3.4 const_iterator end() const

Returns

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

7.3.4 Utility Functions

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.

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.

8 Memory Allocation

This section describes classes related to memory allocation.

8.1 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 31 summarizes the allocator concept. Here, A and B represent instances of the allocator class.

Table 31: 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` (8.2), `scalable_allocator` (8.3), and `cached_aligned_allocator` (8.4), and `zero_allocator` (8.5) model the Allocator concept.

8.2 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 3.1.2.

8.3 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 31. 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 (8.1).

Acknowledgement

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

8.3.1 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 32. 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 32: 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*[24]
2	`scalable_aligned_malloc`	scalable_aligned_free	Microsoft* C run-time library
	`scalable_aligned_free`		

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

	scalable_aligned_realloc		

8.3.1.1 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.

8.4 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 31. 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`.

8.4.1 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.

8.4.2 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.

8.4.3 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.

8.5 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);
    };
}
```

8.6 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();
```

```
    };
}
```

8.6.1 aligned_space()

Effects

None. Does not invoke constructors.

8.6.2 ~aligned_space()

Effects

None. Does not invoke destructors.

8.6.3 T* begin()

Returns

Pointer to beginning of storage.

8.6.4 T* end()

Returns

`begin()+N`

9 Synchronization

The library supports mutual exclusion and atomic operations.

9.1 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.

9.1.1 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 33 shows the requirements for the Mutex concept for a mutex type M

Table 33: 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 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 34 summarizes the classes that model the Mutex concept.

Table 34: 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.

9.1.1.1 C++ 200x Compatibility

Classes `mutex`, `recursive_mutex`, `spin_mutex`, and `spin_rw_mutex` support the C++ 200x interfaces described in Table 35.

Table 35: C++ 200x 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 lock_guard<M>`	See Section 9.4
`class unique_lock<M>`	

Classes mutex and recursive mutex also provide the C++ 200x idiom for accessing their underlying OS handles, as described in Table 36.

Table 36: Native handle interface (M is `mutex` or `recursive_mutex`).

Pseudo-Signature	Semantics
`M::native_handle_type`	Native handle type. Operating system: Windows* operating system — Native handle type: `LPCRITICAL_SECTION` Operating system: Other operationing systems — Native handle type: `(pthread_mutex*)`
`native_handle_type M::native_handle()`	Get underlying native handle of mutex M.

As an extension to C++ 200x, class `spin_rw_mutex` also has methods `read_lock()` and `try_read_lock()` for corresponding operations that acquire reader locks.

9.1.2 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 (9.1.1). 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 (9.1.1).

9.1.3 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` (9.1.2), 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 (9.1.1).

9.1.4 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 (9.1.1). 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 (9.1.1).

9.1.5 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 (9.1.1). 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 (9.1.1).

9.1.6 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 37 shows the requirements for a ReaderWriterMutex `RW`. They form a superset of the Mutex Concept (9.1.1).

Table 37: 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.
`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

Classes `spin_rw_mutex` (9.1.7) and `queuing_rw_mutex` (9.1.8) model the ReaderWriterMutex concept.

9.1.6.1 ReaderWriterMutex()

Effects

Constructs unlocked `ReaderWriterMutex`.

9.1.6.2 ~ReaderWriterMutex()

Effects

Destroys unlocked `ReaderWriterMutex`. The effect of destroying a locked `ReaderWriterMutex` is undefined.

9.1.6.3 ReaderWriterMutex::scoped_lock()

Effects

Constructs a `scoped_lock` object that does not hold a lock on any mutex.

9.1.6.4 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.

9.1.6.5 ReaderWriterMutex::~scoped_lock()

Effects

If the object holds a lock on a `ReaderWriterMutex`, releases the lock.

9.1.6.6 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.

9.1.6.7 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.

9.1.6.8 void ReaderWriterMutex:: scoped_lock::release()

Effects

Releases lock. The effect is undefined if no lock is held.

9.1.6.9 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.

9.1.6.10 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.

Intel's current implementations for `spin_rw_mutex` and `queuing_rw_mutex` always return `true`. Different implementations might sometimes return `false`.

9.1.7 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 (9.1.6). 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 (9.1.6).

9.1.8 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 (9.1.6). 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 (9.1.6).

9.1.9 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 (9.1.1) 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 (9.1.1).

9.1.10 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 (9.1.6) 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 (9.1.6).

9.2 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 38 shows the fencing for the non-template form.

Table 38: 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;
```

9.2.1 memory_semantics Enum

Description

Defines values used to select the template variants that permit more selective control over visibility of operations (see Table 38).

9.2.2 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*.

9.2.3 value_type fetch_and_increment()

Effects

Let *x* be the value of `*this`. Atomically updates $x = x + 1$.

Returns

Original value of *x*.

9.2.4 value_type fetch_and_decrement()

Effects

Let *x* be the value of `*this`. Atomically updates $x = x - 1$.

Returns

Original value of *x*.

9.2.5 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*.

9.2.6 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*.

9.3 PPL Compatibility

Classes `critical_section` and `reader_writer_lock` exist for compatibility with the Microsoft Parallel Patterns Library (PPL). They do not follow all of the conventions of other mutexes in Intel® Threading Building Blocks.

9.3.1 critical_section

Summary

A PPL-compatible mutex.

Syntax

```
class critical_section;
```

Header

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

Description

A critical_section implements a PPL `critical_section`. Its functionality is a subset of the functionality of a `tbb::mutex`.

Members

```
namespace tbb {
    class critical_section {
    public:
        critical_section();
        ~critical_section();
        void lock();
        bool try_lock();
        void unlock();

        class scoped_lock {
        public:
            scoped_lock( critical_section& mutex );
            ~scoped_lock();
        };
   };
```

```
}
```

9.3.2 reader_writer_lock Class

Summary

A PPL-compatible reader-writer mutex that is scalable and gives preference to writers.

Syntax

```
class reader_writer_lock;
```

Header

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

Description

A `reader_writer_lock` implements a PPL-compatible reader-writer mutex. A `reader_writer_lock` is scalable and nonrecursive. The implementation handles lock requests on a first-come first-serve basis except that writers have preference over readers. Waiting threads busy wait, which can degrade system performance if the wait is long. However, if the wait is typically short, a `reader_writer_lock` can provide performance competitive with other mutexes.

A `reader_writer_lock` models part of the ReaderWriterMutex Concept (9.1.6) and part of the C++ 200x compatibility interface (9.1.1.1). The major differences are:

- The scoped interfaces support only strictly scoped locks. For example, the method `scoped_lock::release()` is not supported.
- Reader locking has a separate interface. For example, there is separate scoped interface scoped_lock_read for reader locking, instead of a flag to distinguish the reader cases as in the ReaderWriterMutex Concept.

Members

```
namespace tbb {
    class reader_writer_lock {
    public:
        reader_writer_lock();
        ~reader_writer_lock();
        void lock();
        void lock_read();
        bool try_lock();
        bool try_lock_read();
        void unlock();

        class scoped_lock {
        public:
```

```
            scoped_lock( reader_writer_lock& mutex );
            ~scoped_lock();
        };
        class scoped_lock_read {
        public:
            scoped_lock_read( reader_writer_lock& mutex );
            ~scoped_lock_read();
        };
    };
}
```

Table 39 summarizes the semantics.

Table 39: `reader_writer_lock` Members Summary

Member	Semantics
`reader_writer_lock()`	Construct unlocked mutex.
`~reader_writer_lock()`	Destroy unlocked mutex.
`void reader_writer_lock::lock()`	Acquire write lock on mutex.
`void reader_writer_lock::lock_read()`	Acquire read lock on mutex.
`bool reader_writer_lock::try_lock()`	Try to acquire write lock on mutex. Returns *true* if lock acquired, *false* otherwise.
`bool reader_writer_lock::try_lock_read()`	Try to acquire read lock on mutex. Returns *true* if lock acquired, *false* otherwise.
`reader_writer_lock::unlock()`	Release lock.
`reader_writer_lock::scoped_lock (reader_writer_lock& m)`	Acquire write lock on mutex m.
`reader_writer_lock::~scoped_lock()`	Release write lock (if acquired).
`reader_writer_lock::scoped_lock_read (reader_writer_lock& m)`	Acquire read lock on mutex m.
`reader_writer_lock::~scoped_lock_read()`	Release read lock (if acquired).

9.4 C++ 200x Synchronization

Intel® TBB approximates a portion of C++ 200x interfaces for condition variables and scoped locking. The approximation is based on the C++0x working draft N3000. The major differences are:

- The implementation uses the `tbb::tick_count` interface instead of the C++ 200x <chrono> interface.

- The implementation throws `std::runtime_error` instead of a C++ 200x `std::system_error`.
- The implementation omits or approximates features requiring C++ 200x language support such as `constexpr` or `explicit` operators.
- The implementation works in conjunction with `tbb::mutex` wherever the C++ 200x specification calls for a `std::mutex`. See 9.1.1.1 for more about C++ 200x mutex support in Intel® TBB.

See the working draft N3000 for a detailed descriptions of the members.

CAUTION: Implementations may change if the C++ 200x specification changes.

CAUTION: When support for `std::system_error` becomes available, implementations may throw `std::system_error` instead of `std::runtime_error`.

The library defines the C++ 200x interfaces in namespace `std`, not namespace `tbb`, as explained in Section 2.4.7.

Header

```
#include "tbb/compat/condition_variable"
```

Members

```
namespace std {
    struct defer_lock_t { };
    struct try_to_lock_t { };
    struct adopt_lock_t { };
    const defer_lock_t defer_lock = {};
    const try_to_lock_t try_to_lock = {};
    const adopt_lock_t adopt_lock = {};

    template<typename M>
    class lock_guard {
    public:
        typedef M mutex_type;
        explicit lock_guard(mutex_type& m);
        lock_guard(mutex_type& m, adopt_lock_t);
        ~lock_guard();
    };

    template<typename M>
    class unique_lock: no_copy {
    public:
        typedef M mutex_type;

        unique_lock();
        explicit unique_lock(mutex_type& m);
```

```
        unique_lock(mutex_type& m, defer_lock_t);
        unique_lock(mutex_type& m, try_to_lock_t));
        unique_lock(mutex_type& m, adopt_lock_t);
        unique_lock(mutex_type& m, const tick_count::interval_t
&i);
        ~unique_lock();

        void lock();

        bool try_lock();
        bool try_lock_for( const tick_count::interval_t &i );

        void unlock();

        void swap(unique_lock& u);

        mutex_type* release();

        bool owns_lock() const;
        operator bool() const;
        mutex_type* mutex() const;
    };

    template<typename M>
    void swap(unique_lock<M>& x, unique_lock<M>& y);

    enum cv_status {no_timeout, timeout};

    class condition_variable : no_copy {
    public:
        condition_variable();
        ~condition_variable();

        void notify_one();
        void notify_all();

        void wait(unique_lock<mutex>& lock);

        template <class Predicate>
        void wait(unique_lock<mutex>& lock, Predicate pred);

        cv_status wait_for(unique_lock<mutex>& lock,
                           const tick_count::interval_t& i);

        template<typename Predicate>
```

```
        bool wait_for(unique_lock<mutex>& lock,
                      const tick_count::interval_t &i,
                      Predicate pred);

        typedef implementation-defined native_handle_type;
        native_handle_type native_handle();
    };
} // namespace std
```

10 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.

10.1 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
```

10.1.1 static tick_count tick_count::now()

Returns

Current wall clock timestamp.

CAUTION: On Microsoft Windows* operating systems, the current implementation uses the function `QueryPerformanceCounter`. 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.

10.1.2 tick_count::interval_t operator–(const tick_count& t1, const tick_count& t0)

Returns

Relative time that t1 occurred after t0.

10.1.3 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
```

10.1.3.1 interval_t()

Effects

Constructs `interval_t` representing zero time duration.

10.1.3.2 interval_t(double sec)

Effects

Constructs interval_t representing specified number of seconds.

10.1.3.3 double seconds() const

Returns

Time interval measured in seconds.

10.1.3.4 interval_t operator+=(const interval_t& i)

Effects

```
*this = *this + i
```

Returns

Reference to `*this`.

10.1.3.5 interval_t operator–=(const interval_t& i)

Effects

```
*this = *this – i
```

Returns

Reference to `*this`.

10.1.3.6 interval_t operator+ (const interval_t& i, const interval_t& j)

Returns

Interval_t representing sum of intervals *i* and *j*.

10.1.3.7 interval_t operator– (const interval_t& i, const interval_t& j)

Returns

`Interval_t` representing difference of intervals *i* and *j*.

11 Task Groups

This chapter covers the high-level interface to the task scheduler. Chapter 12 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 40.

Table 40: Requirements on functor arguments

Pseudo-Signature	Semantics
`Func::Func (const Func&)`	Copy constructor.
`Func::~Func ()`	Destructor.
`void Func::operator()() const;`	Evaluate functor.

11.1 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` (4.4) or `parallel_invoke` (4.12) 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();
    }
}
```

11.1.1 task_group()

Constructs an empty `task group`.

11.1.2 ~task_group()

Requires

Method `wait` must be called before destroying a `task_group`, otherwise the destructor throws an exception.

11.1.3 template<typename Func> void run(const Func& f)

Effects

Spawn a task that computes `f()` and return immediately.

11.1.4 template<typename Func> void run (task_handle<Func>& handle);

Effects

Spawn a task that computes `handle()` and return immediately.

11.1.5 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`.

11.1.6 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`.

11.1.7 task_group_status wait()

Effects

Wait for all tasks in the group to complete or be cancelled.

11.1.8 bool is_canceling()

Returns

True if this task group is cancelling its tasks.

11.1.9 void cancel()

Effects

Cancel all tasks in this `task_group`.

11.2 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
    };
}
```

11.3 task_handle Template Class

Summary

Template class used to wrap a function object in conjunction with class `structured_task_group`.

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;
};
```

11.4 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`*`)`

11.5 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();
    };
}
```

11.6 is_current_task_group_canceling Function

Returns

True if innermost task group executing on this thread is cancelling its tasks.

12 Task Scheduler

Intel Threading Building Blocks (Intel® TBB) provides a task scheduler, which is the engine that drives the algorithm templates (Section 4) and task groups (Section 11). 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 its predecessor tasks, at which time it may run the predecessor tasks, or if there are no pending predecessor tasks, the thread may 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 generally avoid making 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 creates 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; it does not block on its children. The children subsequently run, and after they (or their continuations) finish, the continuation task starts running. Figure 7 shows the steps. The running tasks at each step are shaded.

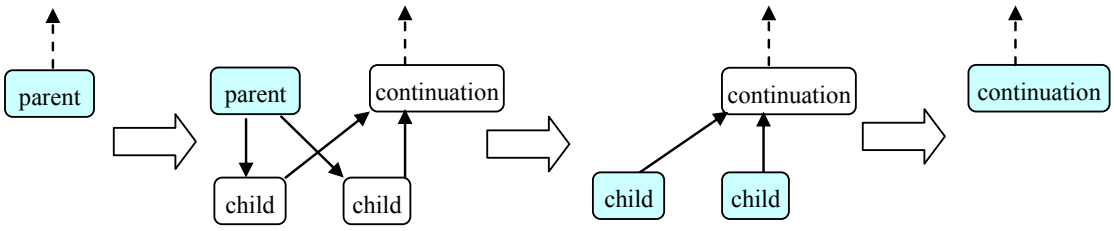

Figure 7: Continuation-passing Style

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 8.

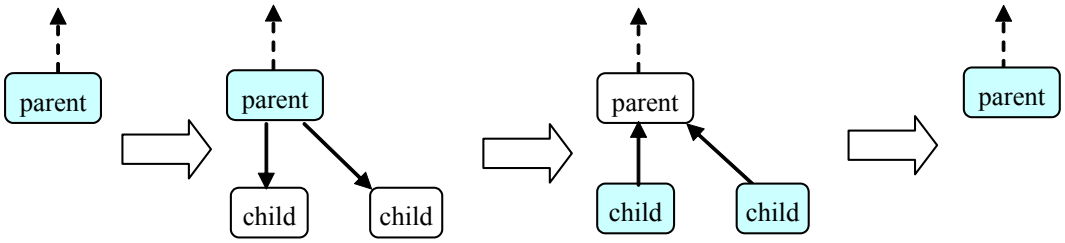

Figure 8: Blocking Style

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.

12.1 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 (double-ended queue) of `task` objects that were *spawned.* Additionally, there is a shared queue of task objects that were *enqueued.* The distinction between spawning a task and enqueuing a task affects when the scheduler runs the task.

After completing a task *t,* a thread chooses its next task according to the first applicable rule below:

1. The task returned by `t.execute()`
2. The successor of *t* if *t* was its last completed predecessor.
3. A task popped from the end of the thread's own deque.
4. A task with affinity for the thread.
5. A task popped from approximately the beginning of the shared queue.
6. A task popped from 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 (6) gets the least recently spawned task of another thread.

When a thread *enqueues* a task, it pushes it onto the end of the shared queue. Hence rule (5) gets one of the less recently enqueued tasks, and has no preference for tasks that are enqueued. This is in contrast to spawned tasks, where by rule (3) a thread prefers its own most recently spawned task.

Note the "approximately" in rule (5). For scalability reasons, the shared queue does **not** guarantee precise first-in first-out behavior. If strict first-in first-out behavior is desired, put the real work in a separate queue, and create tasks that pull work from that queue. The chapter "Non-Preemptive Priorities" in the Intel® TBB Design Patterns manual explains the technique.

It is important to understand the implications of spawning versus enqueuing for nested parallelism.

- Spawned tasks emphasize locality. Enqueued tasks emphasize fairness.
- For nested parallelism, spawned tasks tend towards depth-first execution, whereas enqueued tasks cause breadth-first execution. Because the space demands of breadth-first execution can be exponentially higher than depth-first execution, enqueued tasks should be used with care.
- A spawned task might never be executed until a thread explicitly waits on the task to complete. An enqueued tasks will eventually run if all previously enqueued tasks complete. In the case where there would ordinarily be no other worker thread to execute an enqueued task, the scheduler creates an extra worker.

In general, used spawned tasks unless there is a clear reason to use an enqueued task. Spawned tasks yield the best balance between locality of reference, space efficiency, and parallelism. The algorithm for spawned tasks is similar to the work-stealing algorithm used by Cilk (Blumofe 1995). The notion of work-stealing dates back to the 1980s (Burton 1981). The thread affinity support is more recent (Acar 2000).

12.2 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.h"

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 max_threads=automatic,
                             stack_size_type thread_stack_size=0
);
        ~task_scheduler_init();
        void initialize( int max_threads=automatic );
        void terminate();
        static int default_num_threads();
        bool is_active() const;
    };
} // namespace tbb
```

12.2.1 task_scheduler_init(int max_threads=automatic, stack_size_type thread_stack_size=0)

Requirements

The value `max_threads` shall be one of the values in Table 41.

Effects

If `max_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 41. These workers sleep until needed by the task scheduler. Each worker created by the scheduler has an implicit active `task_scheduler_init` object.

NOTE: As of TBB 3.0, it is meaningful for the parameter `max_threads` to differ for different calling threads. For example, if thread A specifies `max_threads`=3 and thread B specifies `max_threads=7`, then A is limited to having 2 workers, but B can have up to 6 workers. Since workers may be shared between A and B, the total number of worker threads created by the scheduler could be 6.

NOTE: Some implementations create more workers than necessary. However, the excess workers remain asleep unless needed.

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. The first active `task_scheduler_init` establishes the stack size for all worker threads.

Table 41: Values for max_threads

`max_threads`	Semantics
`task_scheduler_init::automatic`	Let library determine `max_threads` based on hardware configuration.
`task_scheduler_init::deferred`	Defer activation actions.
positive integer	Request that up to `max_threads`–1 worker threads work on behalf of the calling thread at any one time.

12.2.2 ~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.

12.2.3 void initialize(int max_threads=automatic)

Requirements

The `task_scheduler_init` shall be inactive.

Effects

Similar to constructor (12.2.1).

12.2.4 void terminate()

Requirements

The `task_scheduler_init` shall be active.

Effects

Deactivates the `task_scheduler_init` without destroying it. The description of the destructor (12.2.2) specifies what deactivation entails.

12.2.5 int default_num_threads()

Returns

One more than the number of worker threads that `task_scheduler_init` creates by default.

12.2.6 bool is_active() const

Returns

True if `*this` is active as described in Section 12.2; false otherwise.

12.2.7 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
...
        }
    }
}
```

12.3 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 42.[25]

[25] The depth attribute in Intel® TBB 2.1 no longer exists (A.6).

Table 42: Task Attributes[26]

Attribute	Description
successor	Either null, or a pointer to another task whose refcount field will be decremented after the present task completes. Typically, the successor is the task that allocated the present task, or a task allocated as the continuation of that task. Methods of class `task` call the successor "parent" and its preceding task the "child", because this was a common use case. But the library has evolved such that a child-parent relationship is no longer required between the predecessor and successor.
refcount	The number of Tasks that have this as their parent. Increments and decrement of refcount are always atomic.

TIP: Always allocate memory for `task` objects using special overloaded new operators (12.3.2) 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 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 9.

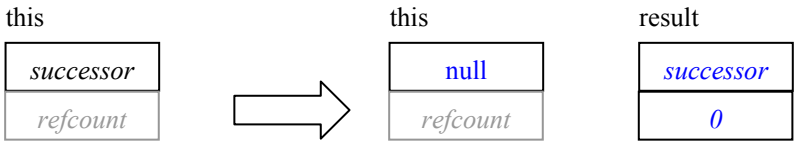

Figure 9: Example Effect Diagram

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 left blank.
- Black denotes state that is read.
- Blue denotes state that is written.

[26] The ownership attribute and restrictions in Intel® TBB 2.1 no longer exist.

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 (12.3.2).

```
namespace tbb {
    class task {
    protected:
        task();

    public:
        virtual ~task() {}

        virtual task* execute() = 0;

        // Allocation
        static proxy1 allocate_root();
        static proxy2 allocate_root( task_group_context& );
        proxy3 allocate_continuation();
        proxy4 allocate_child();
        static proxy5 allocate_additional_child_of( task& );

        // Explicit destruction
        static void destroy( task& victim );

        // Recycling
        void recycle_as_continuation();
        void recycle_as_safe_continuation();
        void recycle_as_child_of( task& new_parent );

        // Synchronization
        void set_ref_count( int count );
        void increment_ref_count();
        int decrement_ref_count();
        void wait_for_all();
        static void spawn( task& t );
        static void spawn( task_list& list );
        void spawn_and_wait_for_all( task& t );
        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 );
        static void enqueue( task& );

        // Task context
```

```
        static task& self();
        task* parent() const;
        void set_parent(task *p);
        bool is_stolen_task() const;
        task_group_context* group();
        void change_group( task_group_context& ctx );

        // 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;

        // 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 );
```

NOTE: Prior to Intel® TBB 3.0, methods `allocate_additional_child_of`, `destroy`, and `spawn` were non-static. Evolution of the library made the `this` argument superfluous for these calls. The change preserves source compatibility except in cases where the address of the method was taken. Executables compiled with the older headers that

had the non-static form will continue to work when linked against the current Intel® TBB 3.0 run-time libraries.

12.3.1 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 predecessor tasks of `this`. Unless one of the recycle/reschedule methods described in Section (12.3.4) 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 12.3.8.

12.3.1.1 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 *successor->refcount*, and if becomes zero, puts the *successor* 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` (deprecated), puts the task back into the ready pool.
   b. Otherwise it was marked by `recycle_as_child` or `recycle_as_continuation`.

12.3.2 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.

12.3.2.1 new(task::allocate_root(task_group_context& group)) *T*

Allocate a `task` of type *T* with the specified cancellation group. Figure 10 summarizes the state transition.

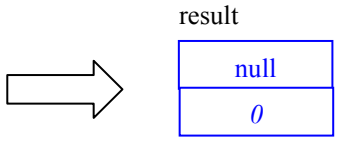

Figure 10: Effect of task::allocate_root()

Use method `spawn_root_and_wait` (12.3.5.9) to execute the `task`.

12.3.2.2 new(task::allocate_root()) *T*

Like `new(task::allocate_root(task_group_context&))` except that cancellation group is the current innermost cancellation group.

12.3.2.3 new(x.allocate_continuation()) *T*

Allocates and constructs a task of type *T*, and transfers the *successor* from *x* to the new task. No reference counts change. Figure 11 summarizes the state transition.

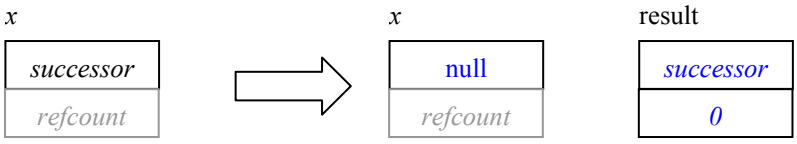

Figure 11: Effect of allocate_continuation()

12.3.2.4 new(*x*.allocate_child()) *T*

Effects

Allocates a `task` with `this` as its *successor*. Figure 12 summarizes the state transition.

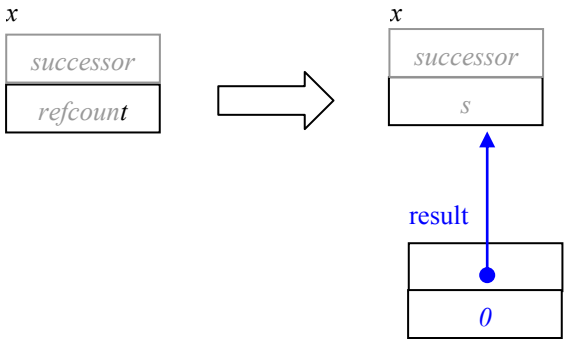

Figure 12: Effect of allocate_child()

If using explicit continuation passing, then the continuation, not the successor, should call the allocation method, so that *successor* is set correctly.

If the number of tasks is not a small fixed number, consider building a `task_list` (12.5) of the predecessors first, and spawning them with a single call to `task::spawn` (12.3.5.5). If a `task` must spawn some predecessors before all are constructed, it should use `task::allocate_additional_child_of(*this)` instead, because that method atomically increments *refcount*, so that the additional predecessor is properly accounted. However, if doing so, the `task` must protect against premature zeroing of *refcount* by using a blocking-style task pattern.

12.3.2.5 new(task::allocate_additional_child_of(*y*)) T

Effects

Allocates a `task` as a predecessor of another `task` `y`. Task `y` may be already running or have other predecessors running. Figure 13 summarizes the state transition.

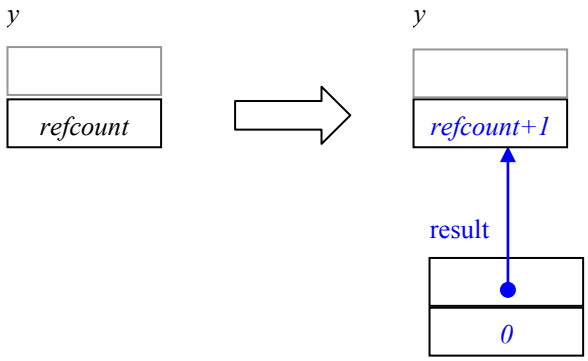

Figure 13: Effect of allocate_additional_child_of(successor)

Because `y` may already have running predecessors, the increment of y.*refcount* is atomic (unlike the other allocation methods, where the increment is not atomic). When adding a predecessor to a task with other predecessors running, it is up to the

programmer to ensure that the successor's *refcount* does not prematurely reach 0 and trigger execution of the successor before the new predecessor is added.

12.3.3 Explicit task Destruction

Usually, a `task` is automatically destroyed by the scheduler after its method `execute` returns. But sometimes `task` objects are used idiomatically (such as for reference counting) without ever running `execute`. Such tasks should be disposed with method `destroy`.

12.3.3.1 static void destroy (task& victim)

Requirements

The *refcount* of `victim` must be zero. This requirement is checked in the debug version of the library.

Effects

Calls destructor and deallocates memory for `victim`. If `victim`.*parent* is not null, atomically decrements `victim`.*parent*->*refcount*. The parent is **not** put into the ready pool if its *refcount* becomes zero. Figure 14 summarizes the state transition.

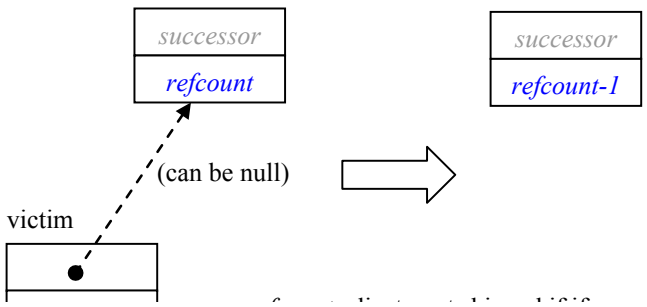

Figure 14: Effect of destroy(victim).

12.3.4 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 predecessors.

CAUTION: **Overlap rule:** A recycled task *t* must not be put in jeopardy of having *t*`.execute()` rerun while the previous invocation of *t*`.execute()` is still running. The debug version of the library detects some violations of this rule.

For example, *t.*execute() should never spawn *t* directly after recycling it. Instead, *t.*execute() should return a pointer to *t*, so that *t* is spawned after *t.*execute() completes.

12.3.4.1 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 predecessors of the continuation task.

CAUTION: The caller must guarantee that the task's *refcount* does not become zero until after method `execute()` returns, otherwise the overlap rule is broken. If the guarantee is not possible, use method `recycle_as_safe_continuation()` instead, and set the *refcount* to *n*+1.

The race can occur for a task *t* when:

t`.execute()` recycles *t* as a continuation.

The continuation has predecessors that all complete before *t*`.execute()` returns.

Hence the recycled *t* will be implicitly respawned with the original *t*`.execute()`still running, which breaks the overlap rule.

Patterns that use `recycle_as_continuation()` typically avoid the race by making *t*`.execute()` return a pointer to one of the predecessors instead of explicitly spawning that predecessor. The scheduler implicitly spawns that predecessor after *t*`.execute()` returns, thus guaranteeing that the recycled *t* does not rerun prematurely.

Effects

Causes `this` to not be destroyed when method `execute()` returns.

12.3.4.2 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 predecessors 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 the race discussed for `recycle_as_continuation` because the additional +1 in the *refcount* prevents the continuation from executing until the original invocation of `execute()` completes.

12.3.4.3 void recycle_as_child_of(task& new_successor)

Requirements

Must be called while method `execute()` is running.

Effects

Causes `this` to become a predecessor of *new_successor*, and not be destroyed when method `execute()` returns.

12.3.5 Synchronization

Spawning a task *t* either causes the calling thread to invoke *t*.`execute()`, or causes *t* to be put into the ready pool. Any thread participating in task scheduling may then acquire the task and invoke *t*.`execute()`. Section 12.1 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`.

Some calls distinguish between spawning root tasks and non-root tasks. A root task is one that was created using method `allocate_root`.

Important

A `task` should not spawn any predecessor task until it has called method `set_ref_count` to indicate both the number of predecessors and whether it intends to use one of the "wait_for_all" methods.

12.3.5.1 void set_ref_count(int count)

Requirements

`count`≥0.[27] If the intent is to subsequently spawn *n* predecessors and wait, then *count* should be *n*+1. Otherwise *count* should be *n.*

Effects

Sets the *refcount* attribute to `count`.

12.3.5.2 void increment_ref_count();

Effects

Atomically increments *refcount* attribute.

12.3.5.3 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 11.6 of the Tutorial ("General Acyclic Graphs of Tasks") explains more.

12.3.5.4 void wait_for_all()

Requirements

refcount=*n*+1, where *n* is the number of predecessors that are still running.

Effects

Executes tasks in ready pool until *refcount* is 1. Afterwards, leaves *refcount*=1 if the task's `task_group_context` specifies `concurrent_wait`, otherwise sets *refcount* to 0.[28] Figure 15 summarizes the state transitions.

---

[27] Intel® TBB 2.1 had the stronger requirement *count*>0.

Also, `wait_for_all()` automatically resets the cancellation state of the `task_group_context` implicitly associated with the task (12.6), 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.

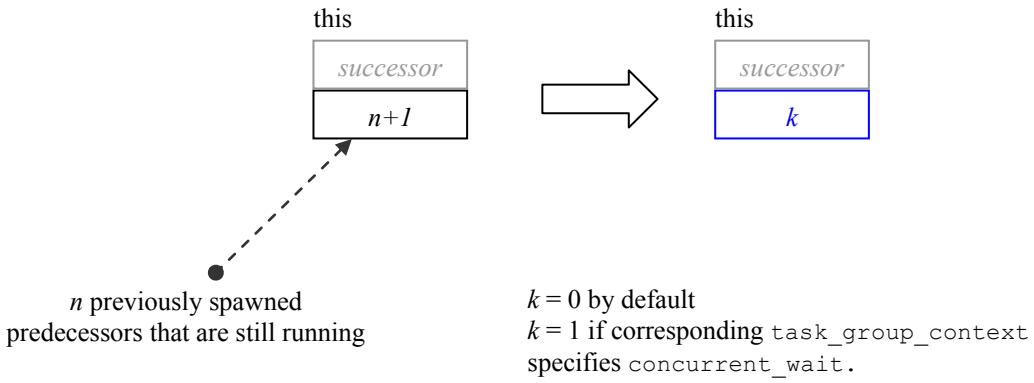

Figure 15: Effect of wait_for_all

12.3.5.5 static void spawn(task& t)

Effects

Puts task `t` into the ready pool and immediately returns.

If the *successor* of t is not null, then `set_ref_count` must be called on that *successor* before spawning any child tasks, because once the child tasks commence, their completion will cause *successor.refcount* to be decremented asynchronously. The debug version of the library often detects when a required call to `set_ref_count` is not made, or is made too late.

---

[28] For sake of backwards compatibility, the default for `task_group_context` is *not* `concurrent_wait`, and hence to set *refcount*=0.

12.3.5.6 static void spawn (task_list& list)

Effects

Equivalent to executing `spawn` on each task in *`list`* and clearing *`list`*, but may be more efficient. If *`list`* is empty, there is no effect.

NOTE: Spawning a long linear list of tasks can introduce a bottleneck, because tasks are stolen individually. Instead, consider using a recursive pattern or a parallel loop template to create many pieces of independent work.

12.3.5.7 void spawn_and_wait_for_all(task& t)

Requirements

Any other predecessors of `this` must already be spawned. The `task` *`t`* must have a non-null attribute *successor*. There must be a chain of *successor* links from *`t`* to the calling `task`. Typically, this chain contains a single link. That is, *`t`* is typically an immediate predecessor of `this`.

Effects

Similar to `{spawn(`*`t`*`); 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 16 illustrates the state transitions. It is similar to Figure 15, with task `t` being the *n*th task.

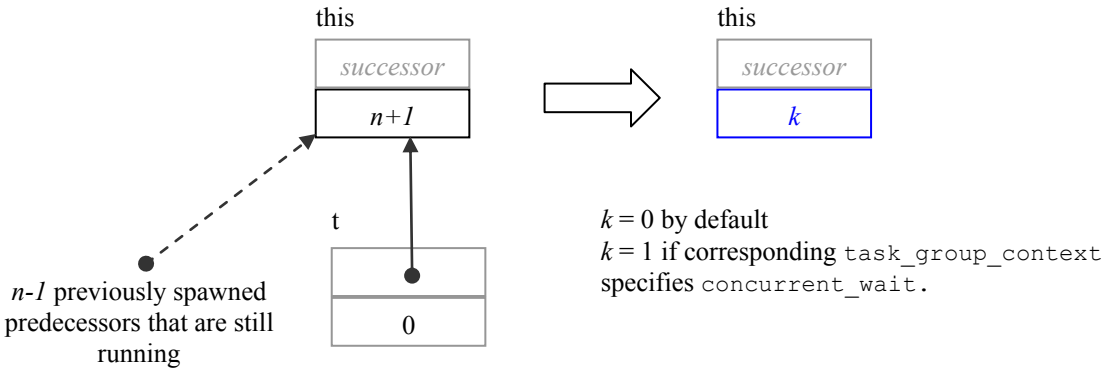

Figure 16: Effect of spawn_and_wait_for_all

12.3.5.8 void spawn_and_wait_for_all(task_list& list)

Effects

Similar to `{spawn(`*`list`*`); wait_for_all();}`, but often more efficient.

12.3.5.9 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 12.3.1.1. Destroys *root* afterwards unless *root* was recycled.

12.3.5.10 static void spawn_root_and_wait(task_list& root_list)

Requirements

Each `task` object *t* in *root_list* must meet the requirements in Section 12.3.5.9.

Effects

For each `task` object *t* in *root_list*, performs `spawn_root_and_wait`(*t*), possibly in parallel. Section 12.3.5.9 describes the actions of `spawn_root_and_wait`(*t*).

12.3.5.11 static void enqueue (task&)

Effects

The task is scheduled for eventual execution by a worker thread even if no thread ever explicitly waits for the task to complete. If the total number of worker threads is zero, a special additional worker thread is created to execute enqueued tasks.

Enqueued tasks are processed in roughly, but not precisely, first-come first-serve order.

CAUTION: Using enqueued tasks for recursive parallelism can cause high memory usage, because the recursion will expand in a breadth-first manner. Use ordinary spawning for recursive parallelism.

CAUTION: Explicitly waiting on an enqueued task should be avoided, because other enqueued tasks from unrelated parts of the program might have to be processed first. The recommended pattern for using an enqueued task is to have it asynchronously signal its completion, for example, by posting a message back to the thread that enqueued it. See the Intel® Threading Building Blocks *Design Patterns* manual for such an example.

12.3.6 task Context

These methods expose relationships between `task` objects, and between `task` objects and the underlying physical threads.

12.3.6.1 static task& self()

Returns

Reference to innermost `task` that the 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.

12.3.6.2 task* parent() const

Returns

Value of the attribute *successor*. 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`.

12.3.6.3 void set_parent(task* p)

Requirements

Both tasks must be in the same task group. For example, for `task t`, `t.group() == p->group()`.

Effects

Sets parent task pointer to specified value p.

12.3.6.4 bool is_stolen_task() const

Returns

`true` if task is running on a thread different than the thread that spawned it.

NOTE: Tasks enqueued with `task::enqueue()` are never reported as stolen.

12.3.6.5 task_group_context* group()

Returns

Descriptor of the task group, which this task belongs to.

12.3.6.6 void change_group(task_group_context& ctx)

Effects

Moves the task from its current task group int the one specified by the `ctx` argument.

12.3.7 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 12.6.

12.3.7.1 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.

12.3.7.2 bool is_cancelled() const

Returns

True if task's group has received a cancellation request; false otherwise.

12.3.8 Priorities

Priority levels can be assigned to individual tasks or task groups. The library supports three levels {low, normal, high} and two kinds of priority:

- Static priority for enqueued tasks.
- Dynamic priority for task groups.

The former is specified by an optional argument of the `task::enqueue()` method, affects a specific task only, and cannot be changed afterwards. Tasks with higher priority are dequeued before tasks with lower priorities.

The latter affects all the tasks in a group and can be changed at any time either via the associated `task_group_context` object or via any task belonging to the group. The priority-related methods in task_group_context are described in Section 12.6.

The task scheduler tracks the highest priority of ready tasks (both enqueued and spawned), and postpones execution of tasks with lower priority until all higher priority task are executed. By default all tasks and task groups are created with normal priority.

NOTE: Priority changes may not come into effect immediately in all threads. So it is possible that lower priority tasks are still being executed for some time even in the presence of higher priority ones.

When several user threads (masters) concurrently execute parallel algorithms, the pool of worker threads is partitioned between them proportionally to the requested concurrency levels. In the presence of tasks with different priorities, the pool of worker threads is proportionally divided among the masters with the highest priority first. Only after fully satisfying the requests of these higher priority masters, will the remaining threads be provided to the other masters.

Though masters with lower priority tasks may be left without workers, the master threads are never stalled themselves. Task priorities also do not affect and are not affected by OS thread priority settings.

NOTE: Worker thread migration from one master thread to another may not happen immediately.

Related constants and methods

```
namespace tbb {
    enum priority_t {
        priority_normal = implementation-defined,
        priority_low = implementation-defined,
        priority_high = implementation-defined
    };

    class task {
        // . . .
        static void enqueue( task&, priority_t );
        void set_group_priority ( priority_t );
        priority_t group_priority () const;
        // . . .
    };
}
```

12.3.8.1 void enqueue (task& t, priority_t p) const

Effects

Enqueues task `t` at the priority level `p`.

NOTE: Priority of an enqueued task does not affect priority of the task group, from the scope of which `task::enqueue()` is invoked (i.e. the group, which the task returned by `task::self()` method belongs to).

12.3.8.2 void set_group_priority (priority_t)

Effects

Changes priority of the task group, which this task belongs to.

12.3.8.3 priority_t group_priority () const

Returns

Priority of the task group, which this task belongs to.

12.3.9 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.

12.3.9.1 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.

12.3.9.2 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.

12.3.9.3 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`.

12.3.9.4 affinity_id affinity() const

Returns

Affinity of this task as set by `set_affinity`.

12.3.10 task Debugging

Methods in this subsection are useful for debugging. They may change in future implementations.

12.3.10.1 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 43 describes valid states. Any other value is the result of memory corruption, such as using a `task` whose memory has been deallocated.

Table 43: 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 17 summarizes possible state transitions for a `task`.

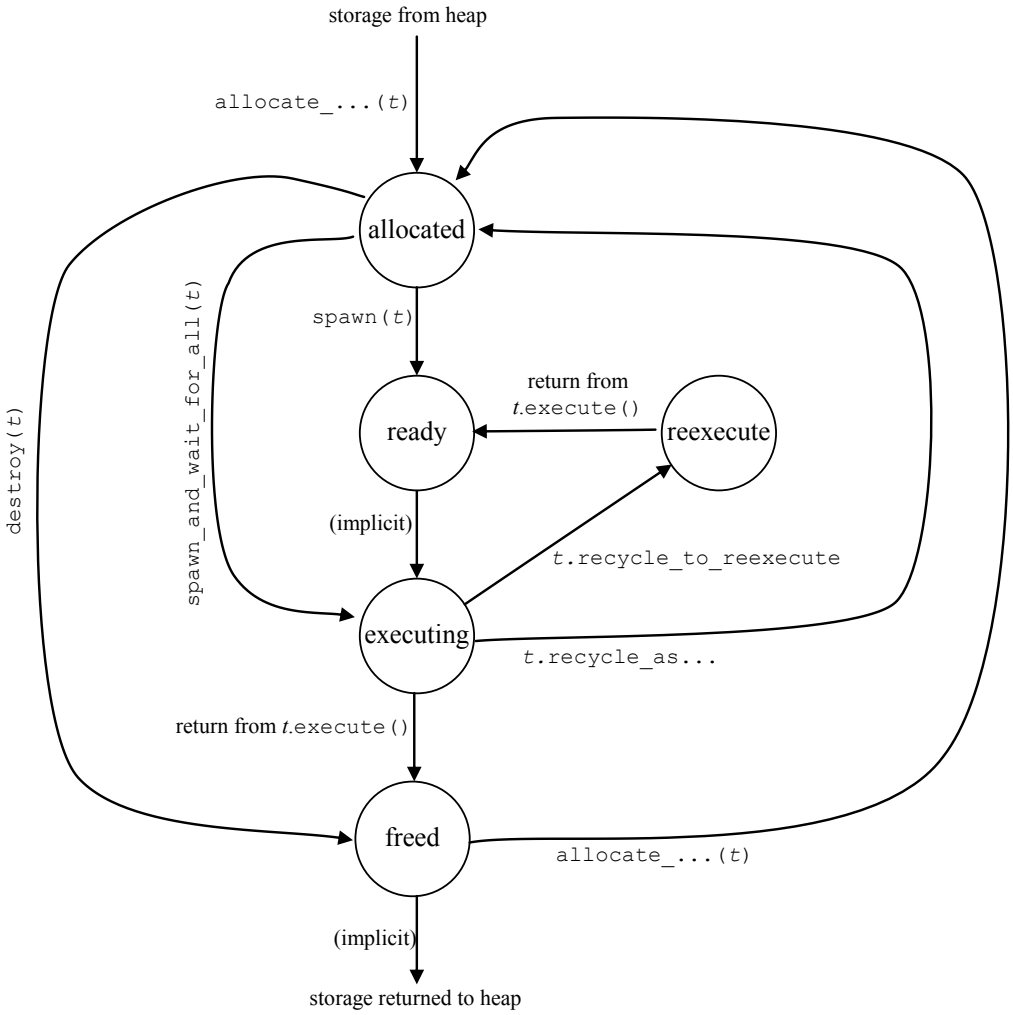

Figure 17: Typical task::state() Transitions

12.3.10.2 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*.

12.4 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 predecessors to complete.

Members

```
namespace tbb {
    class empty_task: public task {
        /*override*/ task* execute() {return NULL;}
    };
}
```

12.5 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 tasks and spawn them all at once via the method `task::spawn(task_list&)`, as described in 12.3.5.6.

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();
    };
}
```

12.5.1 task_list()

Effects

Constructs an empty list.

12.5.2 ~task_list()

Effects

Destroys the list. Does not destroy the `task` objects.

12.5.3 bool empty() const

Returns

True if list is empty; false otherwise.

12.5.4 push_back(task& task)

Effects

Inserts a reference to *task* at back of the list.

12.5.5 task& task pop_front()

Effects

Removes a `task` reference from front of list.

Returns

The reference that was removed.

12.5.6 void clear()

Effects

Removes all `task` references from the list. Does not destroy the `task` objects.

12.6 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 or have their priority level set together. All tasks belong to some group. A task can be a member of only one group at any moment.

A root task is associated with a group by passing task_group_context object into task::allocate_root() call. A child task automatically joins its parent task's group. A task can be moved into other group using `task::change_group()` method.

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. Canceling a `task_group_context` causes the entire subtree rooted at it to be cancelled.

The priorities for all the tasks in a group can be changed at any time either via the associated `task_group_context` object, or via any task belonging to the group. Priority changes propagate into the child task groups similarly to cancelation. The effect of priorities on task execution is described in Section 12.3.8.

Each user thread that creates a `task_scheduler_init` (12.2) 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 (12.3.6.1).

Members

```
namespace tbb {
    class task_group_context {
    public:
        enum kind_t {
            isolated = implementation-defined,
            bound = implementation-defined
        };

        enum traits_type {
            exact_exception = implementation-defined,
            concurrent_wait = implementation-defined,
#if TBB_USE_CAPTURED_EXCEPTION
            default_traits = 0
#else
            default_traits = exact_exception
#endif /* !TBB_USE_CAPTURED_EXCEPTION */
        };
        task_group_context( kind_t relation_with_parent = bound,
                            uintptr_t traits = default_traits );
        ~task_group_context();
        void reset();
        bool cancel_group_execution();
        bool is_group_execution_cancelled() const;
        void set_priority ( priority_t );
        priority_t priority () const;
    };
}
```

12.6.1 task_group_context(kind_t relation_to_parent=bound, uintptr_t traits=default_traits)

Effects

Constructs an empty `task_group_context`. If *`relation_to_parent`* is `bound`, the `task_group_context` will become a child of the innermost running task's group when it is first passed into the call to `task::allocate_root(task_group_context&)`. If this call is made directly from the user thread, the effect will be as if *`relation_to_parent`* were `isolated`. If *`relation_to_parent`* is `isolated`, it has no parent `task_group_context`.

The `traits` argument should be the bitwise OR of `traits_type` values. The flag `exact_exception` controls how precisely exceptions are transferred between threads. See Section 13 for details. The flag `concurrent_wait` controls the reference-counting behavior of methods `task::wait_for_all` and `task::spawn_and_wait_for_all`.

12.6.2 ~task_group_context()

Effects

Destroys an empty task_group_context. It is a programmer error if there are still extant tasks in the group.

12.6.3 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.

12.6.4 bool is_group_execution_cancelled() const

Returns

True if group has received cancellation.

12.6.5 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.

12.6.6 void set_priority (priority_t)

Effects

Changes priority of the task group.

12.6.7 priority_t priority () const

Returns

Priority of the task group.

12.7 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 ) {}
    };
}
```

12.7.1 task_scheduler_observer()

Effects

Constructs instance with observing disabled.

12.7.2 ~task_scheduler_observer()

Effects

Disables observing. Waits for extant invocations of `on_scheduler_entry` or `on_scheduler_exit` to complete.

12.7.3 void observe(bool state=true)

Effects

Enables observing if state is true; disables observing if state is false.

12.7.4 bool is_observing() const

Returns

True if observing is enabled; false otherwise.

12.7.5 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.

12.7.6 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.

12.8 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.

12.8.1 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`.

12.8.2 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.

12.8.2.1 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& t_k   = *new(allocate_child()) T(...); spawn(t_k);
        task& t_k-1 = *new(allocate_child()) T(...); spawn(t_k-1);
        ...
        // Return pointer to first child instead of spawning it,
        // to remove unnecessary overhead.
        task& t_1 = *new(allocate_child()) T(...);
        return &t_1;
    }
}
```

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 12.8.1.
- 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.

12.8.2.2 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` (12.4).

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
```

```
    } else {
        // Construct continuation
        C& c = allocate_continuation();
        c.set_ref_count(k);
        // Recycle self as first child
        task& t_k  = *new(c.allocate_child()) T(...); spawn(t_k);
        task& t_k-1 = *new(c.allocate_child()) T(...); spawn(t_k-1);
        ...
        task& t_2 = *new(c.allocate_child()) T(...);  spawn(t_2);
        // task t_1 is our recycled self.
        recycle_as_child_of(c);
        update fields of *this to subproblem to be solved by t_1
        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 12.8.1.
- 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.

12.8.3 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` (12.4).

```
task* dummy = new( task::allocate_root() ) empty_task;
dummy->set_ref_count(k+1);
task& t_k = *new( dummy->allocate_child() ) T;  dummy->spawn(t_k);
task& t_k-1= *new( dummy->allocate_child() ) T;  dummy->spawn(t_k-1);
...
task& t_1 = *new( dummy->allocate_child() ) T;  dummy->spawn(t_1);
...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.

13 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 or an approximation of it is captured.
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.

The exact exception is captured when both of the following conditions are true:

- The task's `task_group_context` was created in a translation unit compiled with `TBB_USE_CAPTURED_EXCEPTION=0`.
- The Intel® TBB library was built with a compiler that supports the `std::exception_ptr` feature of C++ 200x.

Otherwise an appoximation of the original exception `x` is captured as follows:

1. If `x` is a `tbb_exception`, it is captured by `x.move()`.
2. If `x` is a `std::exception`, it is captured as a `tbb::captured_exception(typeid(x).name(),x.what())`.
3. Otherwise `x` is captured as a `tbb::captured exception` with implementation-specified value for `name()` and `what()`.

13.1 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.

13.2 captured_exception

Summary

Class used by Intel® TBB to capture an approximation of an exception.

Syntax

```
class captured_exception;
```

Header

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

Description

When a task throws an exception, sometimes Intel® TBB converts the exception to a `captured_exception` before propagating it. The conditions for conversion are described in Section 13.

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 13.1 describes the rest of the interface.

13.2.1 captured_exception(const char* name, const char* info)

Effects

Constructs a `captured_exception` with the specified *`name`* and *`info`*.

13.3 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 13.1 describes the rest of the interface.

13.3.1 movable_exception(const ExceptionData& src)

Effects

Construct `movable_exception` containing copy of src.

13.3.2 ExceptionData& data() throw()

Returns

Reference to contained data.

13.3.3 const ExceptionData& data() const throw()

Returns

Const reference to contained data.

13.4 Specific Exceptions

Summary

Exceptions thrown by other library components.

Syntax

```
class bad_last_alloc;
class improper_lock;
class invalid_multiple_scheduling;
class missing_wait;
class user_abort;
```

Header

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

Description

Table 44 describes when the exceptions are thrown.

Table 44: Classes for Specific Exceptions.

Exception	Thrown when...
bad_last_alloc	• A `pop` operation on a `concurrent_queue` or `concurrent_bounded_queue` correrspondsto a `push` that threw an exception. • An operation on a `concurrent_vector` cannot be performed because a prior operation threw an exception.
improper_lock	A thread attempts to lock a `critical_section` or `reader_writer_lock` that it it has already locked.
invalid_multiple_scheduling	A `task_group` or `structured_task_group` attempts to run a `task_handle` twice.

missing_wait	A `task_group` or `structured_task_group` is destroyed before method `wait()` is invoked.
user_abort	A `push` or `pop` operation on a `concurrent_bounded_queue` was aborted by the user.

Members

```
namespace tbb {
    class bad_last_alloc: public std::bad_alloc {
    public:
        const char* what() const throw();
    };
    class improper_lock: public std::exception {
    public:
        const char* what() const throw();
    };
    class invalid_multiple_scheduler: public std::exception {
       const char* what() const throw();
    };
    class missing_wait: public std::exception {
    public:
        const char* what() const throw();
    };
    class user_abort : public std::exception {
    public:
        const char* what() const throw();
    };
}
```

14 Threads

Intel® Threading Building Blocks (Intel® TBB) provides a wrapper around the platform's native threads, based upon the N3000 working draft 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 library defines the wrapper in namespace `std`, not namespace `tbb`, as explained in Section 2.4.7.[29]

The significant departures from N3000 are shown in Table 45.

Table 45: Differences Between N3000 and Intel® TBB Thread Class

N3000	Intel® TBB
`template<class Rep, class Period> std::this_thread::sleep_for( const chrono::duration<Rep, Period>& rel_time)`	`std::this_thread::sleep_for( 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 `std::thread` takes 0-3 arguments.

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 entities on most systems, and running too many threads on a system can seriously degrade performance. Consider using a task based solution instead if practical.

[29] In Intel® TBB 2.2, the class was `tbb::tbb_thread`. Appendix A.7 explains the changes.

14.1 thread Class

Summary

Represents a thread of execution.

Syntax

```
class thread;
```

Header

```
#include "tbb/compat/thread"
```

Description

Class `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 std {
    class thread {
    public:
#if _WIN32||_WIN64
        typedef HANDLE native_handle_type;
#else
        typedef pthread_t native_handle_type;
#endif // _WIN32||_WIN64

        class id;

        thread();
        template <typename F> explicit thread(F f);
        template <typename F, typename X> thread(F f, X x);
        template <typename F, typename X, typename Y>
            thread (F f, X x, Y y);
        thread& operator=( thread& x);
        ~thread();

        bool joinable() const;
        void join();
        void detach();
        id get_id() const;
        native_handle_type native_handle();
```

```
        static unsigned hardware_concurrency();
    };
}
```

14.1.1 thread()

Effects

Constructs a `thread` that does not represent a thread of execution, with `get_id()==id()`.

14.1.2 template<typename F> thread(F f)

Effects

Construct a `thread` that evaluates f()

14.1.3 template<typename F, typename X> thread(F f, X x)

Effects

Constructs a `thread` that evaluates f(x).

14.1.4 template<typename F, typename X, typename Y> thread(F f, X x, Y y)

Effects

Constructs `thread` that evaluates f(x,y).

14.1.5 thread& operator=(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.

14.1.6 ~thread

Effects

`if( joinable() ) detach()`.

14.1.7 bool joinable() const

Returns

`get_id()!=id()`

14.1.8 void join()

Requirements

`joinable()==true`

Effects

Wait for thread to complete. Afterwards, `joinable()==false`.

14.1.9 void detach()

Requirements

`joinable()==true`

Effects

Sets `*this` to default constructed state and returns without blocking. The thread represented by `*this` continues execution.

14.1.10 id get_id() const

Returns

id of the thread, or a default-constructed id if *this does not represent a thread.

14.1.11 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`.

14.1.12 static unsigned hardware_concurrency()

Returns

The number of hardware threads. For example, 4 on a system with a single Intel® Core™2 Quad processor.

14.2 thread::id

Summary

Unique identifier for a thread.

Syntax

```
class thread::id;
```

Header

```
#include "tbb/compat/thread"
```

Description

A `thread::id` is an identifier value for a thread that remains unique over the thread's lifetime. A special value `thread::id()` represents no thread of execution. The instances are totally ordered.

Members

```
namespace tbb {
    class thread::id {
    public:
        id();
    };
    template<typename charT, typename traits>
    std::basic_ostream<charT, traits>&
        operator<< (std::basic_ostream<charT, traits> &out,
                    thread::id id)
```

```
    bool operator==(thread::id x, thread::id y);
    bool operator!=(thread::id x, thread::id y);
    bool operator<(thread::id x, thread::id y);
    bool operator<=(thread::id x, thread::id y);
    bool operator>(thread::id x, thread::id y);
    bool operator>=(thread::id x, thread::id y);
} // namespace tbb
```

14.3 this_thread Namespace

Description

Namespace this_thread contains global functions related to threading.

Members

```
namepace tbb {
    namespace this_thread {
        thread::id get_id();
        void yield();
        void sleep( const tick_count::interval_t );
    }
}
```

14.3.1 thread::id get_id()

Returns

Id of the current thread.

14.3.2 void yield()

Effects

Offers to suspend current thread so that another thread may run.

14.3.3 void sleep_for(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_thread::sleep_for( tick_count::interval_t(30) );
}
```

15 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.

Working Draft, Standard for Programming Language C++. WG21 document N3000. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n3000.pdf>

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.

W.F. Burton and R.M. Sleep. Executing functional programs on a virtual tree of processors. *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (October 1981):187-194.

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 (4.7) 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 46 shows the requirements on the stream and body.

Table 46: 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 )`	Process `item`. `parallel_while`

Pseudo-Signature	Semantics
`const`	may concurrently invoke the operator for the same `this` but 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` (5.5) 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 4.9.6.1.

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 47. 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 3.2.

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 47: 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 48.

Table 48: 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 49: Change in Return Types

Method	Deprecated Return Type	New Return Type
`grow_by` (5.8.3.1)	`size_type`	`iterator`
`grow_to_at_least` (5.8.3.2)	`void`	`iterator`
`push_back` (5.8.3.3)	`size_type`	`iterator`

A.5.1 void compact()

Effects

Same as `shrink_to_fit() (5.8.2.2).`

A.6 Interface for class task

Some methods of class `task` are deprecated because they have obsolete or redundant functionality.

Deprecated Members of class task

```
namespace tbb {
    class task {
    public:
        ...
        void recycle_to_reexecute();
        // 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 ){}
        ...
    };
}
```

A.6.1 void recycle _to_reexecute()

Intel® TBB 3.0 deprecated method `recycle_to_reexecute` because it is redundant. Replace a call *t*`->recycle_to_reexecute()` with the following sequence:

```
    t->set_refcount(1);
```

```
    t->recycle_as_safe_continuation();
```

A.6.2 Depth interface for class task

Intel® TBB 2.2 eliminated 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.

A.7 tbb_thread Class

Intel® TBB 3.0 introduces a header `tbb/compat/thread` that defines class `std::thread`. Prior versions had a header `tbb/tbb_thread.h` that defined class `tbb_thread`. The old header and names are still available, but deprecated in favor of the replacements shown inTable 50.

Table 50: Replacements for Deprecated Names

Entity	Deprecated	Replacement
Header	`tbb/tbb_thread.h`	`tbb/compat/thread`
Identifiers	`tbb::tbb_thread`	`std::thread`
	`tbb::this_tbb_thread`	`std::this_thread`
	`tbb::this_tbb_thread::sleep`	`std::this_tbb_thread::sleep_for`

Most of the changes reflect a change in the way that the library implements C++ 200x features (2.4.7). The change from `sleep` to `sleep_for` reflects a change in the C++ 200x working draft.

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 51 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 51: PPL Compatibility Features

Section	Feature
4.4	parallel_for(*first*,*last*, *f*)
4.4	parallel_for(*first*,*last*,*step*,*f*)
4.8	parallel_for_each
4.12	parallel_invoke
9.3.1	critical_section
9.3.2	reader_writer_lock
11.3	task_handle
11.2	task_group_status
11.1.1	task_group
11.4	make_task
11.5	structured_task_group
11.6	is_current_task_group_cancelling
13.4	improper_lock
13.4	invalid_multiple_scheduling
13.4	missing_wait

For `parallel_for`, only the variants listed in the table are 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.

Appendix D Community Preview Features

This section provides documentation for Community Preview (CP) features.

What is a Community Preview Feature?

A Community Preview feature is a component of Intel® Threading Building Blocks (Intel® TBB) that is being introduced to gain early feedback from developers. Comments, questions and suggestions related to Community Preview features are encouraged and should be submitted to the forums at www.threadingbuildingblocks.org.

The key properties of a CP feature are:

- It must be explicitly enabled. It is off by default.
- It is intended to have a high quality implementation.
- There is no guarantee of future existence or compatibility.
- It may have limited or no support in tools such as correctness analyzers, profilers and debuggers.

CAUTION: A CP feature is subject to change in the future. It may be removed or radically altered in future releases of the library. Changes to a CP feature do NOT require the usual deprecation and deletion process. Using a CP feature in a production code base is therefore strongly discouraged.

Enabling a Community Preview Feature

A Community Preview feature may be defined completely in header files or it may require some additional support defined in a library.

For a CP feature that is contained completely in header files, a feature-specific macro must be defined before inclusion of the header files.

Example

```
#define TBB_PREVIEW_FOO 1
#include "tbb/foo.h"
```

If a CP feature requires support from a library, then an additional library must be linked with the application.

The use of separate headers, feature-specific macros and separate libraries mitigates the impact of Community Preview features on other product features.

NOTE: Unless a CP feature is explicitly enabled using the above mechanisms, it will have no impact on the application.

D.1 Flow Graph

This section describes Flow Graph nodes that are available as Community Preview features.

D.1.1 or_node Template Class

Summary

A node that broadcasts messages received at its input ports to all of its successors. Each input port p_i is a `receiver<T_i>`. The messages are broadcast individually as they are received at each port. The output message types is a struct that contains an index number that identifies the port on which the message arrived and a tuple of the input types where the value is stored.

Syntax

```
template<typename InputTuple>
class or_node;
```

Header

```
#define TBB_PREVIEW_GRAPH_NODES 1
#include "tbb/flow_graph.h"
```

Description

An or_node is a `graph_node` and `sender< or_node<InputTyple>::output_type >`. It contains a tuple of input ports, each of which is a `receiver<Ti>` for each of the `T0` .. `TN` in `InputTuple`. It supports multiple input receivers with distinct types and broadcasts each received message to all of its successors. Unlike a `join_node`, each message is broadcast individually to all successors of the `or_node` as it arrives at an input port. The incoming messages are wrapped in a struct that contains the index of the port number on which the message arrived and a tuple of the input types where the received value is stored.

The function template `input_port` described in 6.21 simplifies the syntax for getting a reference to a specific input port.

Rejection of messages by successors of the `or_node` is handled using the protocol in Figure 4. The input ports never reject incoming messages.

`InputTuple` must be a `std::tuple<T0,T1,…>` where each element is copy-constructible and assignable.

Example

```
#include<cstdio>
#define TBB_PREVIEW_GRAPH_NODES 1
```

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

using namespace tbb::flow;

int main() {
   graph g;
   function_node<int,int> f1( g, unlimited,
       [](const int &i) { return 2*i; } );
   function_node<float,float> f2( g, unlimited,
       [](const float &f) { return f/2; } );

   typedef or_node< std::tuple<int,float> > my_or_type;
   my_or_type o(g);

   function_node< my_or_type::output_type >
       f3( g, unlimited,
           []( const my_or_type::output_type &v ) {
               if (v.indx == 0) {
                   printf("Received an int %d\n",
std::get<0>(v.result));
               } else {
                   printf("Received a float %f\n",
std::get<1>(v.result));
               }
       }

   );
   make_edge( f1, input_port<0>(o) );
   make_edge( f2, input_port<1>(o) );
   make_edge( o, f3 );
   f1.try_put( 3 );
   f2.try_put( 3 );
   g.wait_for_all();

   return 0;
}
```

In the example above, three `function_node` objects are created: `f1` multiplies an `int i` by 2, `f2` divides a `float f` by 2, and f3 prints the values from `f1` and `f2` as they arrive. The `or_node j` wraps the output of `f1` and `f2` and forwards each result to `f3`. This example is purely a syntactic demonstration since there is very little work in the nodes.

Members

```
namespace tbb {
namespace flow {
```

```
template<typename InputTuple>
class or_node : public graph_node,
                public sender< impl-dependent-output-type > {
public:
    typedef struct {  size_t indx;
                      InputTuple result;
                   } output_type;    typedef receiver<output_type>
successor_type;
    implementation-dependent-tuple input_ports_type;

    or_node(graph &g);
    or_node(const or_node &src);
    input_ports_type &input_ports();
    bool register_successor( successor_type &r );
    bool remove_successor( successor_type &r );
    bool try_get( output_type &v );
    bool try_reserve( output_type & );
    bool try_release( );
    bool try_consume( );
};

}
}
```

D.1.1.1 or_node(graph &g)

Effect

Constructs an `or_node` that belongs to the `graph g`.

D.1.1.2 or_node(const or_node &src)

Effect

Constructs an `or_node`. The list of predecessors, messages in the input ports, and successors are NOT copied.

D.1.1.3 input_ports_type& input_ports()

Returns

A `std::tuple` of receivers. Each element inherits from `tbb::receiver<T>` where `T` is the type of message expected at that input. Each tuple element can be used like any other `flow::receiver<T>`.

D.1.1.4 bool register_successor(successor_type & r)

Effect

Adds `r` to the set of successors.

Returns

true.

D.1.1.5 bool remove_successor(successor_type & r)

Effect

Removes `r` from the set of successors.

Returns

true.

D.1.1.6 bool try_get(output_type &v)

Description

An `or_node` contains no buffering and therefore does not support gets.

Returns

false.

D.1.1.7 bool try_reserve(T &)

Description

An `or_node` contains no buffering and therefore cannot be reserved.

Returns

false.

D.1.1.8 bool try_release( )

Description

An `or_node` contains no buffering and therefore cannot be reserved.

Returns

false.

D.1.1.9 bool try_consume( )

Description

An `or_node` contains no buffering and therefore cannot be reserved.

Returns

false.

D.2 Run-time loader

Summary

The run-time loader is a mechanism that provides additional run-time control over the version of the Intel ® Threading Buidling Blocks (Intel® TBB) dynamic library used by an application, plug-in, or another library.

Header

```
#define TBB_PREVIEW_RUNTIME_LOADER 1
#include "tbb/runtime_loader.h"
```

Library

OS	Release build	Debug build
Windows	tbbproxy.lib	tbbproxy_debug.lib

Description

The run-time loader consists of a class and a static library that can be linked with an application, library, or plug-in to provide better run-time control over the version of Intel® TBB used. The class allows loading a desired version of the dynamic library at run time with explicit list of directories for library search. The static library provides stubs for functions and methods to resolve link-time dependencies, which are then dynamically substituted with the proper functions and methods from a loaded Intel® TBB library.

All instances of `class runtime loader` in the same module (i.e. exe or dll) share certain global state. The most noticeable piece of this state is the loaded Intel® TBB library. The implications of that are:

Only one Intel® TBB library per module can be loaded.

If one `runtime_loader` instance has already loaded a library, another one created by the same module will not load another one. If the loaded library is suitable for the second instance, both will use it cooperatively, otherwise an error will be reported (details below).

If different versions of the library are requested by different modules, those can be loaded, but may result in processor oversubscription.

`runtime_loader` objects are not thread-safe and may work incorrectly if used concurrently.

NOTE: If an application or a library uses `runtime_loader`, it should be linked with one of the above specified libraries instead of a normal Intel® TBB library.

Example

```
#define TBB_PREVIEW_RUNTIME_LOADER 1
#include "tbb/runtime_loader.h"
#include "tbb/parallel_for.h"
#include <iostream>

char const * path[] = { "c:\\myapp\\lib\\ia32", NULL };

int main() {
    tbb::runtime_loader loader( path );
    if( loader.status()!=tbb::runtime_loader::ec_ok )
        return -1;

    // The loader does not impact how TBB is used
    tbb::parallel_for(0, 10, ParallelForBody());

    return 0;
}
```

In this example, the Intel® Threading Building Blocks (Intel®) library will be loaded from the c:\myapp\lib\ia32 directory. No explicit requirements for a version are specified, so the minimal suitable version is the version used to compile the example, and any higher version is suitable as well. If the library is successfully loaded, it can be used in the normal way.

D.2.1 runtime_loader Class

Summary

Class for run time control over the loading of an Intel® Threading Building Blocks dynamic library.

Syntax

```
class runtime_loader;
```

Members

```
namespace tbb {
    class runtime_loader {
        // Error codes.
        enum error_code {
            ec_ok,        // No errors.
            ec_bad_call,  // Invalid function call.
            ec_bad_arg,   // Invalid argument passed.
            ec_bad_lib,   // Invalid library found.
            ec_bad_ver,   // The library found is not suitable.
            ec_no_lib     // No library found.
        };
        // Error mode constants.
        enum error_mode {
            em_status, // Save status of operation and continue.
            em_throw,  // Throw an exception of error_code type.
            em_abort   // Print message to stderr, and abort().
        };

        runtime_loader( error_mode mode = em_abort );
        runtime_loader(
            char const *path[],
                     // List of directories to search in.
            int min_ver = TBB_INTERFACE_VERSION,
                     // Minimal suitable version
            int max_ver = INT_MAX,
                     // Maximal suitable version
            error_mode mode = em_abort
                     // Error mode for this instance.
        );
        ~runtime_loader();
        error_code load(
            char const * path[],
            int min_ver = TBB_INTERFACE_VERSION,
            int max_ver = INT_MAX
```

```
        );
        error_code status();
    };
}
```

D.2.1.1 runtime_loader(error_mode mode = em_abort)

Effects

Initialize `runtime_loader` but do not load a library.

D.2.1.2 runtime_loader(char const * path[], int min_ver = TBB_INTERFACE_VERSION, int max_ver = INT_MAX, error_mode mode = em_abort)

Requirements

The last element of `path[]` must be `NULL`.

Effects

Initialize `runtime_loader` and load Intel® TBB (see `load()` for details). If `error mode` equals to `em_status`, the method `status()` can be used to check whether the library was loaded or not. If `error mode` equals to `em_throw`, in case of a failure an exception of type `error_code` will be thrown. If `error mode` equals to `em_abort`, in case of a failure a message will be printed to stderr, and execution aborted.

D.2.1.3 error_code load(char const * path[],int min_ver = TBB_INTERFACE_VERSION, int max_ver = INT_MAX)

Requirements

The last element of `path[]` must be `NULL`.

Effects

Load a suitable version of an Intel® TBB dynamic library from one of the specified directories.

TIP: The method searches for a library in directories specified in the `path[]` array. When a library is found, it is loaded and its interface version (as returned by `TBB_runtime_interface_version()`) is checked. If the version does not meet the requirements specified by `min_ver` and `max_ver`, the library is unloaded. The search continues in the next specified path, until a suitable version of the Intel® TBB library is found or the array of paths ends with `NULL`. It is recommended to use default values for `min_ver` and `max_ver`.

CAUTION: For security reasons, avoid using relative directory names such as current ("."), parent ("..") or any other relative directory (like "lib") when searching for a library. Use only absolute directory names (as shown in the example above); if necessary, construct absolute names at run time. Neglecting these rules may cause your program to execute 3-rd party malicious code. (See http://www.microsoft.com/technet/security/advisory/2269637.mspx for details.)

Returns

`ec_ok` – a suitable version was successfully loaded.

`ec_bad_call` - this `runtime_loader` instance has already been used to load a library.

`ec_bad_lib` – A library was found but it appears invalid.

ec_bad_arg - min_ver and/or max_ver is negative or zero, or min_ver > max_ver.

`ec_bad_ver` - unsuitable version has already been loaded by another instance.

`ec_no_lib` - No suitable version was found.

D.2.1.4 error_code status()

Returns

If error mode is `em_status`, the function returns status of the last operation.

D.3 parallel_ deterministic _reduce Template Function

Summary

Computes reduction over a range, with deterministic split/join behavior.

Syntax

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

template<typename Range, typename Body>
void parallel_deterministic_reduce( const Range& range,
```

```
                        const Body& body
                        [, task_group_context& group] );
```

Header

```
#define TBB_PREVIEW_DETERMINISTIC_REDUCE 1
#include "tbb/parallel_reduce.h"
```

Description

The `parallel_deterministic_reduce` template is very similar to the `parallel_reduce` template. It also has the functional and imperative forms and has similar requirements for Func and Reduction (Table 12) and Body (Table 13).

Unlike `parallel_reduce`, `parallel_deterministic_reduce` has deterministic behavior with regard to splits of both Body and Range and joins of the bodies. For the functional form, it means Func is applied to a deterministic set of Ranges, and Reduction merges partial results in a deterministic order. To achieve that,

`parallel_deterministic_reduce` always uses `simple_partitioner` because other partitioners may react on random work stealing behaviour (see 4.3.1). So the template declaration does not have a partitioner argument.

`parallel_deterministic_reduce` always invokes Body splitting constructor for each range splitting.

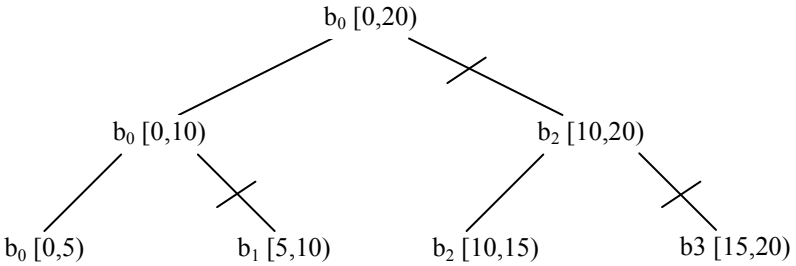

Figure 18: Execution of parallel_deterministic_reduce over blocked_range<int>(0,20,5)

As a result, `parallel_deterministic_reduce` recursively splits a range until it is no longer divisible, and creates a new body (by calling Body splitting constructor) for each new subrange. Likewise `parallel_reduce`, for each body split the method `join` is invoked in order to merge the results from the bodies. Figure 18 shows the execution of `parallel_deterministic_reduce` over a sample range, with the slash marks (/) denoting where new instances of the body were created.

Therefore for given arguments `parallel_ deterministic_reduce` executes the same set of split and join operations no matter how many threads participate in execution and how tasks are mapped to the threads. If the user-provided functions are also deterministic (i.e. different runs with the same input result in the same output), then multiple calls to `parallel_deterministic_reduce` will produce the same result. Note

however that the result might differ from that obtained with an equivalent sequential (linear) algorithm.

CAUTION: Since `simple_partitioner` is always used, be careful to specify an appropriate grainsize (see `simple_partitioner` class).

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

The example from `parallel_reduce` section can be easily modified to use `parallel_deterministic_reduce`. It is sufficient to define TBB_PREVIEW_DETERMINISTIC_REDUCE macro and rename `parallel_reduce` to `parallel_deterministic_reduce`; a partitioner, if any, should be removed to prevent compilation error. A grain size may need to be specified for blocked_range if performance suffered.

```
#define TBB_PREVIEW_DETERMINISTIC_REDUCE 1
#include <numeric>
#include <functional>
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

float ParallelSum( float array[], size_t n ) {
    size_t grain_size = 1000;
    return parallel_deterministic_reduce(
        blocked_range<float*>( array, array+n, grain_size ),
        0.f,
        [](const blocked_range<float*>& r, float value)->float {
            return std::accumulate(r.begin(),r.end(),value);
        },
        std::plus<float>()
    );

}
```

D.4 Scalable Memory Pools

Memory pools allocate and free memory from a specified region or underlying allocator providing thread-safe, scalable operations. Table 52 summarizes the memory pool concept. Here, P represents an instance of the memory pool class.

Table 52: Memory Pool Concept

Pseudo-Signature	Semantics
`~P() throw();`	Destructor. Frees all the memory of allocated objects.
`void P::recycle();`	Frees all the memory of allocated objects.
`void* P::malloc(size_t n);`	Returns pointer to *n* bytes allocated from memory pool.
`void P::free(void* ptr);`	Frees memory object specified via *ptr* pointer.
`void* P::realloc(void* ptr, size_t n);`	Reallocates memory object pointed by *ptr* to *n* bytes.

Model Types

Template class `memory_pool` (D.4.1) and class `fixed_pool` (D.4.2) model the Memory Pool concept.

D.4.1 memory_pool Template Class

Summary

Template class for scalable memory allocation from memory blocks provided by an underlying allocator.

CAUTION: If the underlying allocator refers to another scalable memory pool, the inner pool (or pools) must be destroyed before the outer pool is destroyed or recycled.

Syntax

```
template <typename Alloc> class memory_pool;
```

Header

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "tbb/memory_pool.h"
```

Description

A `memory_pool` allocates and frees memory in a way that scales with the number of processors. The memory is obtained as big chunks from an underlying allocator specified by the template argument. The latter must satisfy the subset of requirements described in Table 31 with `allocate`, `deallocate`, and `value_type` valid for `sizeof(value_type)>0`. A `memory_pool` models the Memory Pool concept described in Table 52.

Example

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "tbb/memory_pool.h"
...
tbb::memory_pool<std::allocator<char> > my_pool;
void* my_ptr = my_pool.malloc(10);
my_pool.free(my_ptr);
```

The code above provides a simple example of allocation from an extensible memory pool.

Members

```
namespace tbb {
template <typename Alloc>
class memory_pool : no_copy {
public:
    memory_pool(const Alloc &src = Alloc()) throw(std::bad_alloc);
    ~memory_pool();
    void recycle();
    void *malloc(size_t size);
    void free(void* ptr);
    void *realloc(void* ptr, size_t size);
};
}
```

D.4.1.1 memory_pool(const Alloc &src = Alloc())

Effects

Constructs memory pool with an instance of underlying memory allocator of type `Alloc` copied from `src`. Throws `bad_alloc` exception if runtime fails to construct an instance of the class.

D.4.2 fixed_pool Class

Summary

Template class for scalable memory allocation from a buffer of fixed size.

Syntax

```
class fixed_pool;
```

Header

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include “tbb/memory_pool.h”
```

Description

A `fixed_pool` allocates and frees memory in a way that scales with the number of processors. All the memory available for the allocation is initially passed through arguments of the constructor. A `fixed_pool` models the Memory Pool concept described in Table 52.

Example

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "tbb/memory_pool.h"
...
char buf[1024*1024];
tbb::fixed_pool my_pool(buf, 1024*1024);
void* my_ptr = my_pool.malloc(10);
my_pool.free(my_ptr);}
```

The code above provides a simple example of allocation from a fixed pool.

Members

```
namespace tbb {
class fixed_pool : no_copy {
public:
    fixed_pool(void *buffer, size_t size) throw(std::bad_alloc);
    ~fixed_pool();
    void recycle();
    void *malloc(size_t size);
    void free(void* ptr);
    void *realloc(void* ptr, size_t size);
};
}
```

D.4.2.1 fixed_pool(void *buffer, size_t size)

Effects

Constructs memory pool to manage the memory pointed by `buffer` and of `size`. Throws `bad_alloc` exception if runtime fails to construct an instance of the class.

D.4.3 memory_pool_allocator Template Class

Summary

Template class that provides the C++ allocator interface for memory pools.

Syntax

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

Header

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include “tbb/memory_pool.h”
```

Description

A `memory_pool_allocator` models the allocator requirements described in Table 31 except for default constructor which is excluded from the class. Instead, it provides a constructor, which links with an instance of `memory_pool` or `fixed_pool` classes, that actually allocates and deallocates memory. The class is mainly intended to enable memory pools within STL containers.

Example

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include "tbb/memory_pool.h"
...
typedef tbb::memory_pool_allocator<int>
    pool_allocator_t;
std::list<int, pool_allocator_t>
    my_list(pool_allocator_t( my_pool ));
```

The code above provides a simple example of cnostruction of a container that uses a memory pool.

Members

```
namespace tbb {
template<typename T>
class memory_pool_allocator {
```

```
public:
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    template<typename U> struct rebind {
        typedef memory_pool_allocator<U> other;
    };

    memory_pool_allocator(memory_pool &pool) throw();
    memory_pool_allocator(fixed_pool &pool) throw();
    memory_pool_allocator(const memory_pool_allocator& src)
throw();
    template<typename U>
    memory_pool_allocator(const memory_pool_allocator<U,P>& src)
throw();

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

    memory_pool_allocator(memory_pool &pool) throw();
    memory_pool_allocator(fixed_pool &pool) throw();
    memory_pool_allocator(const memory_pool_allocator& src)
throw();
    template<typename U>
```

```
    memory_pool_allocator(const memory_pool_allocator<U>& src)
throw();

};
template<typename T, typename U>
inline bool operator==( const memory_pool_allocator<T>& a,
                        const memory_pool_allocator<U>& b);
template<typename T, typename U>
inline bool operator!=( const memory_pool_allocator<T>& a,
                        const memory_pool_allocator<U>& b);
}
```

D.4.3.1 memory_pool_allocator(memory_pool &pool)

Effects

Constructs memory pool allocator serviced by `memory_pool` instance `pool`.

D.4.3.2 memory_pool_allocator(fixed_pool &pool)

Effects

Constructs memory pool allocator serviced by `fixed_pool` instance `pool`.

D.5 Serial subset

Summary

A subset of the parallel algorithms is provided for modeling serial execution. Currently only a serial version of `tbb::parallel_for()` is available.

D.5.1 tbb::serial::parallel_for()

Header

```
#define TBB_PREVIEW_SERIAL_SUBSET 1
#include "tbb/ parallel_for.h"
```

Motivation

Sometimes it is useful, for example while debugging, to execute certain `parallel_for()` invocations serially while having other invocations of `parallel_for()` executed in parallel.

Description

The `tbb::serial::parallel_for` function implements the `tbb::parallel_for` API using a serial implementation underneath. Users who want sequential execution of a certain `parallel_for()` invocation will need to define the TBB_PREVIEW_SERIAL_SUBSET macro before `parallel_for.h` and prefix the selected `parallel_for()` with `tbb::serial::`. Internally, the serial implementation uses the same principle of recursive decomposition, but instead of spawning tasks, it does recursion "for real", i.e. the body function calls itself twice with two halves of its original range.

Example

```
#define TBB_PREVIEW_SERIAL_SUBSET 1
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

Foo()
{
      // . . .
      tbb::serial::parallel_for(  . . .  );
      tbb::parallel_for(  . . .  );
      // . . .
}
```

D.6 concurrent_lru_cache Template Class

Summary

Template class for Least Recently Used cache with concurrent operations.

Syntax

```
template <typename key_type, typename value_type, typename
value_functor_type = value_type (*)(key_type) >
class concurrent_lru_cache;
```

Header

```
#define TBB_PREVIEW_CONCURRENT_LRU_CACHE 1
#include "tbb/concurrent_lru_cache.h"
```

Description

A `concurrent_lru_cache` container maps keys to values with the ability to limit the number of stored unused objects. There is at most one element in the container for each key.

The container permits multiple threads to concurrently retrieve items from it.

The container tracks the lifetime of retrieved items by returning a proxy object instead of a real value.

The container stores all the items that are currently in use and a limited number of unused items. Extra items are removed in a least recently used manner.

When no item is found for a key, the container calls the user provided function object to get a needed value and inserts it. The functor object must be thread safe.

Members

```
namespace tbb {
  template <typename key_type,
            typename value_type,
            typename value_functor_type = value_type (*)(key_type)
>
  class concurrent_lru_cache{
  private:
    class handle_object;

  public:
    typedef handle_object handle;

  public:
    concurrent_lru_cache(value_functor_type f, std::size_t
number_of_lru_history_items);

    handle_object operator()(key_type k);

private:
    struct handle_move_t;
    class handle_object {
    public:
        handle_object(handle_move_t m);
        operator handle_move_t();
        value_type& value();
        ~handle_object();
        friend handle_move_t move(handle_object& h);
    private:
        void operator=(handle_object&);
        handle_object(handle_object &);
    };
 };
 }
```

D.6.1 concurrent_lru_cache(value_function_type f, std::size_t number_of_lru_history_items);

Effects

Constructs an empty cache with a `number_of_lru_history_items` maximum number of stored unused objects, `and f` function object returning new values.

D.6.2 handle_object operator[](key_type k)

Effects

Search the container for a pair with given key. If the pair is not found, the user provided function object is called to get the value and insert it into the container.

Returns

`handle_object` pointing to the matching value.

D.6.3 ~ concurrent_lru_cache ()

Effects

Destroys all items in the container, and the container itself, so that it can no longer be used.

D.6.4 handle_object class

Summary

Class that provides read and write access to value in a `concurrent_lru_cache`.

Syntax

```
  template <typename key_type,
            typename value_type,
            typename value_functor_type = value_type (*)(key_type)
>
  class concurrent_lru_cache::handle_object {
```

Header

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

Description

A `handle_object` is a (smart handle) proxy object returned by the cache container allowing getting reference to the value.

Live object of this type prevents the container from erasing values while they are being used.

The `handle_object` does not have copy semantics; instead it only allows transfer of ownership i.e. it semantics is similar to one of std::auto_ptr or move semantics from C++0x.

Members and free standing functions

```
namespace tbb {
  template <typename key_type,
            typename value_type,
            typename value_functor_type = value_type (*)(key_type)
>
  class concurrent_lru_cache::handle_object {
    public:
        handle_object(handle_move_t m);
        operator handle_move_t();
        value_type& value();
        ~handle_object();
    private:
        void operator=(handle_object&);
        handle_object(handle_object &);
    };
 };

 handle_move_t move(handle_object& h);
 }
```

D.6.4.1 handle_object(handle_move_t m)

Effects

Constructs an `handle_object` object from a pointer or from another `handle_object` (through implicit conversion to `handle_move_t` object).

Since handle_object objects owns a reference to a value object of LRU cache, when a new `handle_object` is constructed from another `handle_object`, the former owner releases the reference ownership (i.e. no longer refers to any value) .

D.6.4.2 operator handle_move_t()

Description

This method should not be called directly, instead use free standing `move` function.

Effects

Transfer reference ownership from `handle_object` objects to temporary `handle_move_t` object.

Returns

`handle_move_t` object pointing to the same value object of LRU cache

D.6.4.3 value_type& value()

Effects

Return a reference to value object of LRU cache container.

Returns

Reference to `value_type` object inside the LRU cache container.

D.6.4.4 ~handle_object()

Effects

Release a reference to value object of LRU cache container. It it was the last reference to the value object the container is allowed to evict the value.

D.6.5 handle_move_t class

This is an instrumental class to allow certain conversions that allow ownership transfer between instances of `handle_object` objects. As well it allows `handle_object` objects to be passed to and returned from functions. The class has no members other than holding a reference to value object in LRU cache container and a pointer to the container itself.