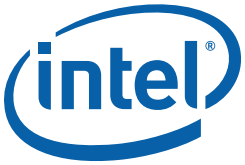

Intel(R) Threading Building Blocks

Reference Manual

Document Number 315415-001US

Revision: 1.14

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

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
315415-001	1.8	Reword description of class aligned_space. Add `atomic<bool>`. Fix definition of `concurrent_queue::empty()`. Remove partitioner as concept. Add `affinity_partitioner`. Update description of `concurrent_vector`. Add `recursive_mutex`.	2007-Dec-19
	1.9	Fix errors in task patterns with continuation passing. Update copyright to 2008. Add `const` to argument `hint` for memory allocators. Add class `task_scheduler_observer`, `tbb_thread`, and `tbb_allocator`. Update `concurrent_hash_map`. Add stack size parameter for `task_scheduler_init`. Add memory allocator argument to container classes.	2008-Mar-31

Document Number	Revision Number	Description	Revision Date
	1.10	Add `pipeline` enhancements. Clarify behavior of assignment and copy constructor for `atomic<T>`. Extend description of `parallel_scan`. Add `task_group_context`. Add new `insert` and `erase` methods to `concurrent_hash_map`. Add chapter on exceptions.	2008-Jun-3
	1.11	Condense discussion of using partitioner with `parallel_scan`. Deprecate virtualness of `~pipeline`. Add methods `clear()` and `get_allocator()` to `concurrent_queue`, and add optional allocator argument to constructor. Add section about TBB_VERSION.	2008-Sept-4
	1.12	Add `task_scheduler_init::is_active()`. Document new debugging macros and deprecate old ones. Add `filter::serial_out_of_order` and `filter::finalize()`. Describe automatic reset of cancellation state by `task::wait_for_all()`. Clarify behavior of `concurrent_vector::clear()`. Add `TBB_runtime_interface_version()`. Add new constructors to `concurrent_queue`. Consistently use `typename` keyword for template parameters.	2008-Dec-5
	1.13	Update copyright to 2009. Remove requirement to clear a `pipeline` before destroying its filters. Add `null_mutex` and `null_rw_mutex`.	2009-Feb-6
	1.14	Clarify semantics of `concurrent_hash_map` methods `insert` and `erase`.	2009-Mar-13

Contents

1 Overview

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

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

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

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

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

2 General Conventions

This section describes conventions used in this document.

2.1 Notation

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

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

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

The actual implementation might look like:

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

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

typedef internal::Foo_v3 Foo;
```

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

- `Foo` is actually a typedef to `Foo_v3`.

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

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

2.2 Terminology

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

2.2.1 Concept

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

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

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

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

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

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

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

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

[1] See Section 3.3.2 of *Concepts for C++0x* available at http://www.osl.iu.edu/publications/prints/2005/siek05:_concepts_cpp0x.pdf for further discussion of valid expressions versus pseudo-signatures.

operator<(U& x, U& y) is acceptable because C++ permits implicit addition of a const qualifier to a reference type.

2.2.2 Model

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

2.2.3 CopyConstructible

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

Table 2: CopyConstructible Requirements

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

2.3 Identifiers

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

2.3.1 Case

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

2.3.2 Reserved Identifier Prefixes

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

2.4 Namespaces

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

2.4.1 tbb Namespace

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

2.4.2 tbb::internal Namespace

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

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

2.5 Thread Safety

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

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

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

2.6 Enabling Debugging Features

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

Table 3: Debugging Macros

Macro	Default Value	Feature
TBB_USE_DEBUG	Windows* systems: 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® Threading Tools.
TBB_USE_PERFORMANCE_WARNINGS		Enable warnings about performance issues.

2.6.1 TBB_USE_ASSERT Macro

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

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

TIP: On Windows* systems, debug builds implicitly set TBB_DO_ASSERT to 1.

2.6.2 TBB_USE_THREADING_TOOLS Macro

The macro TBB_USE_THREADING_TOOLS controls support for Intel® Threading Tools:

- Intel® Thread Profiler
- Intel® Thread Checker.

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

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

2.6.3 TBB_USE_PERFORMANCE_WARNINGS Macro

The macro TBB_USE_PERFORMANCE_WARNINGS controls performance warnings. Define it to be 1 to enable the warnings. Currently, the only warnings affected are some that report poor hash functions for concurrent_hash_map. Enabling the warnings may impact performance.

2.7 Version Information

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

2.7.1 Version Macros

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

Table 4: Version Macros

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

2.7.2 TBB_VERSION Environment Variable

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

Table 5: Output from TBB_VERSION

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

	`scalable_malloc` if TBB malloc library was successuly loaded; `malloc` otherwise.

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

2.7.3 TBB_runtime_interface_version Function

Summary

Function that returns the interface version of the 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 TBB headers.

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

3 Algorithms

Most parallel algorithms provided by the library are generic. They operate on all types that model the necessary concepts. Parallel algorithms may be nested. For example, the body of a `parallel_for` can invoke another `parallel_for`.

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

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

3.1 Splittable Concept

Summary

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

Requirements

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

Table 6: Splittable Concept

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

Description

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

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

The following model types provide examples.

Model Types

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

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

3.1.1 split Class

Summary

Type for dummy argument of a splitting constructor.

Syntax

```
class split;
```

Header

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

Description

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

Members

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

3.2 Range Concept

Summary

Requirements for type representing a recursively divisible set of values.

Requirements

Table 7 lists the requirements for a Range type `R`.

Table 7: Range Concept

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

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

Description

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

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

Example

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

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

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

Model Types

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

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

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

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

3.2.1 blocked_range<Value> Template Class

Summary

Template class for a recursively divisible half-open interval.

Syntax

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

Header

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

Description

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

A blocked_range models the Range concept (3.2).

Table 8: Value Concept for blocked_range

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

A blocked_range<Value> specifies a *grainsize* of type size_t. A blocked_range is splittable into two subranges if the size of the range exceeds *grain size*. The ideal grain size depends upon the context of the blocked_range<Value>, which is typically as the range argument to the loop templates parallel_for, parallel_reduce, or

parallel_scan. A too small grainsize may cause scheduling overhead within the loop templates to swamp speedup gained from parallelism. A too large grainsize may unnecessarily limit parallelism. For example, if the grain size is so large that the range can be split only once, then the maximum possible parallelism is two.

Here is a suggested procedure for choosing *grainsize*:

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

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

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

Examples

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

Members

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

        // constructors
        blocked_range( Value begin, Value end, size_type
grainsize=1);
        blocked_range( blocked_range& r, split );

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

        // access
```

```
        size_type grainsize() const;
        bool is_divisible() const;

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

3.2.1.1 size_type

Description

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

```
const_iterator
```

Description

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

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

Requirements

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

Effects

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

Example

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

3.2.1.3 blocked_range(blocked_range& range, split)

Requirements

`is_divisible()` is true.

Effects

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

Example

Let `i` and `j` be integers that define a half-open interval `[i,j)` and let `g` specifiy a grain size. The statement `blocked_range<int> r(i,j,g)` constructs a `blocked_range<int>` that represents `[i,j)` with grain size `g`. Running the statement `blocked_range<int> s(r,split);` subsequently causes `r` to represent `[i, i +(j –i)/2)` and `s` to represent `[i +(j –i)/2, j)`, both with grain size `g`.

3.2.1.4 size_type size() const

Requirements

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

Effects

Determines size of range.

Returns

`end()–begin()`

3.2.1.5 bool empty() const

Effects

Determines if range is `empty`.

Returns

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

3.2.1.6 size_type grainsize() const

Returns

Grain size of range.

3.2.1.7 bool is_divisible() const

Requirements

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

Effects

Determines if range can be split into subranges.

Returns

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

3.2.1.8 const_iterator begin() const

Returns

Inclusive lower bound on range.

3.2.1.9 const_iterator end() const

Returns

Exclusive upper bound on range.

3.2.2 blocked_range2d Template Class

Summary

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

Syntax

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

Header

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

Description

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

Members

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

```
        typedef blocked_range<ColValue> col_range_type;

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

        // Capacity
        bool empty() const;

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

Example

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

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

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

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

```
using namespace tbb;

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

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

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

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

3.2.2.1 row_range_type

Description

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

3.2.2.2 col_range_type

Description

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

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

Effects

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

Example

The statement "`blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2 );`" constructs a two-dimensional space that contains all value pairs of the form (i, j), where i ranges from `'a'` to `'z'` with a grain size of `3`, and j ranges from `0` to `9` with a grain size of `2`.

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

Effects

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

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

Effects

Partitions `range` into two subranges. The newly constructed `blocked_range2d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. The split is either by rows or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective row and column grain sizes. For example, if the `row_grainsize` is twice `col_grainsize`, the subranges will tend towards having twice as many rows as columns.

3.2.2.6 bool empty() const

Effects

Determines if range is empty.

Returns

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

3.2.2.7 bool is_divisible() const

Effects

Determines if range can be split into subranges.

Returns

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

3.2.2.8 const row_range_type& rows() const

Returns

Range containing the rows of the value space.

3.2.2.9 const col_range_type& cols() const

Returns

Range containing the columns of the value space.

3.2.3 blocked_range3d Template Class

Summary

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

Syntax

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

Header

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

Description

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

Members

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

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

        // Capacity
        bool empty() const;

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

3.3 Partitioners

Summary

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

Description

The default behavior of the loop templates `parallel_for` (3.4), `parallel_reduce` (3.5), and `parallel_scan` (3.6) is to recursively split a range until it is no longer divisible (3.2). An optional partitioner parameter enables other behaviors to be specified, as shown in Table 9.

Table 9: Partitioners

Partitioner	Loop Behavior
`simple_partitioner` (default)	Recursively splits a range until it is no longer divisible. The `Range::is_divisible` function is wholly responsible for deciding when recursive splitting halts. When used with classes such as `blocked_range`, the selection of an appropriate grainsize is critical to enabling concurrency while limiting overheads (see the discussion in Section 3.2.1).
`auto_partitioner`	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 defdault 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.

3.3.1 simple_partitioner Class

Summary

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

Syntax

```
class simple_partitioner;
```

Header

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

Description

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

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

Members

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

3.3.1.1 simple_partitioner()

Construct a `simple_partitioner`.

3.3.1.2 ~simple_partitioner()

Destroy this `simple_partitioner`.

3.3.2 auto_partitioner Class

Summary

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

Syntax

```
class auto_partitioner;
```

Header

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

Description

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

The range subdivision is initially limited to S subranges, where S is proportional to the number of threads specified by the `task_scheduler_init` (8.2.1). Each of these subranges is not divided further unless it is stolen by an idle thread. If stolen, it is further subdivided to create additional subranges. Thus a loop template with an `auto_partitioner` creates additional subranges only when necessary to balance load.

TIP: When using `auto_partitioner` and a `blocked_range` for a parallel loop, the body may be passed a subrange larger than the `blocked_range`'s grainsize. Therefore do

not assume that the grainsize is an upper bound on the size of the subrange. Use a `simple_partitioner` if an upper bound is required.

Members

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

3.3.2.1 auto_partitioner()

Construct an `auto_partitioner`.

3.3.2.2 ~auto_partitioner()

Destroy this `auto_partitioner`.

3.3.3 affinity_partitioner

Summary

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

Syntax

```
class affinity_partitioner;
```

Header

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

Description

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

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

TIP: The affinity_partitioner generally improves performance only when:

- The computation does a few operations per data access.
- The data acted upon by the loop fits in cache.

- The loop, or a similar loop, is re-executed over the same data.
- There are more than two hardware threads available.

Members

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

Example

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

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

using namespace tbb;

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

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


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

```
    }
}
```

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

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

3.3.3.1 affinity_partitioner()

Construct an `affinity_partitioner`.

3.3.3.2 ~affinity_partitioner()

Destroy this `affinity_partitioner`.

3.4 parallel_for<Range,Body> Template Function

Summary

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

Syntax

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

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

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

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

Header

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

Description

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

Table 10: Requirements for parallel_for Body

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

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

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

When worker threads are available (8.2), `parallel_for` executes iterations is non-deterministic order. Do not rely upon any particular execution order for correctness. However, for efficiency, do expect `parallel_for` to tend towards operating on consecutive runs of values.

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

Complexity

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

Example

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

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

using namespace tbb;

struct Average {
    const float* input;
    float* output;
    void operator()( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end(); ++i )
            output[i] = (input[i-
1]+input[i]+input[i+1])*(1/3.0f);
    }
};

// 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, 1000 ), 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>()
    );
}
```

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

3.5 parallel_reduce<Range,Body> Template Function

Summary

Computes reduction over a range.

Syntax

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

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

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

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

Header

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

Description

A `parallel_reduce(`*range*`,`*body*`)` performs parallel reduction of *body* over each value in *range*. Type Range must model the Range concept (3.2). The body must model the requirements in Table 11.

Table 11: Requirements for parallel_reduce Body

Pseudo-Signature	Semantics
`Body::Body( Body&, split );`	Splitting constructor (3.1). Must be able to run concurrently with `operator()` and method `join`.
`Body::~Body()`	Destructor.
`void Body::operator()( 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 (8.2.1), `parallel_reduce` invokes the splitting constructor for the body. For each such split of the body, it invokes method `join` in order to merge the results from the bodies. Define `join` to update this to represent the accumulated result for this and rhs. The reduction operation should be associative, but does not have to be commutative. For a noncommutative operation *op*, "*left*.join(*right*)" should update *left* to be the result of *left op right*.

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

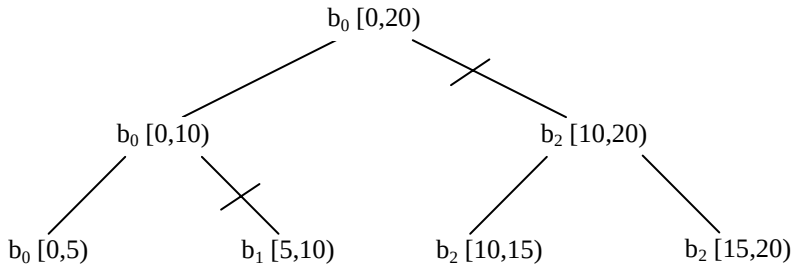

Figure 1: Example Execution of parallel_reduce Over blocked_range<int>(0,20,5)

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

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

Complexity

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

Example

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*>& range ) {
        float temp = value;
        for( float* a=range.begin(); a!=range.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, 1000),
                     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.

The block size of 1000 can be omitted if you use an `auto_partitioner` or `affinity_paritioner`. With an `auto_partitioner`, the invocation of `parallel_reduce` can be changed as shown below:

```
    parallel_reduce( blocked_range<float*>( array, array+n, 1000),
                     total, auto_partitioner() );
```

3.6 parallel_scan<Range,Body> 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
simple_partitioner& );

template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body, const
auto_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 12.

Table 12: 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`.

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

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

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

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

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

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

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

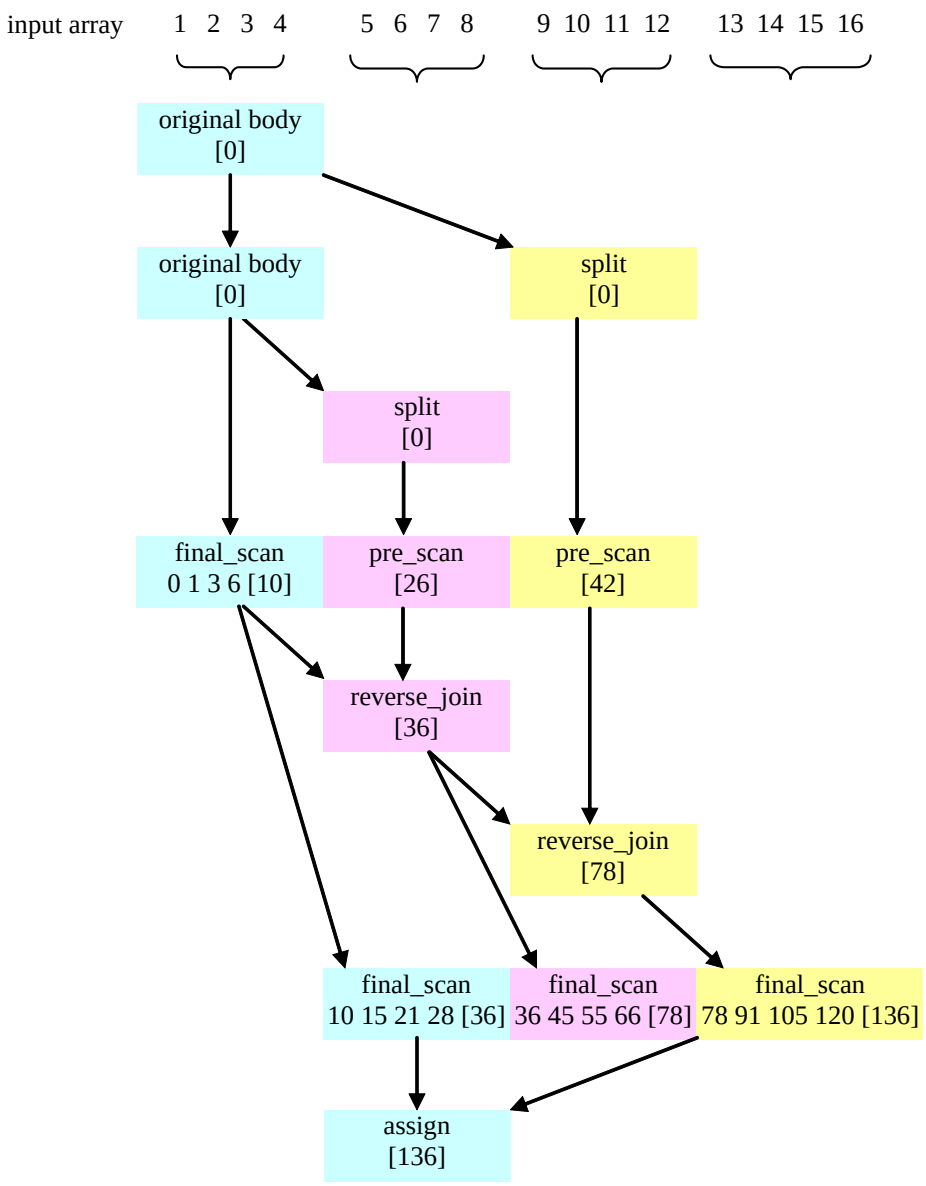

Figure 2: Example Execution of parallel_scan

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

```
using namespace tbb;

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

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

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

float DoParallelScan( T y[], const T x[], int n) {
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n,1000), 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.

The block size of 1000 can be omitted if you use an auto_partitioner. With an auto_partitioner, the invocation of parallel_scan can be changed as shown below:

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

3.6.1 pre_scan_tag and final_scan_tag Classes

Summary

Types that distinguish the phases of parallel_scan..

Syntax

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

Header

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

Description

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

Members

```
namespace tbb {

    struct pre_scan_tag {
        static bool is_final_scan();
    };

    struct final_scan_tag {
        static bool is_final_scan();
    };

}
```

3.6.1.1 bool is_final_scan()

Returns

True for a final_scan_tag, otherwise false.

3.7 parallel_do<InputIterator,Body> Template Function

Summary

Template function that processes work items in parallel.

Syntax

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

Header

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

Description

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

The requirements for input iterators are specified in Section 24.1 of the ISO C++ standard. Table 13 shows the requirements on type `Body`.

Table 13: parallel_do Requirements for Body B and its Argument Type T

Pseudo-Signature	Semantics

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

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

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

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

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

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

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

Example

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

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

3.7.1 parallel_do_feeder<Item> class

Summary

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

Syntax

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

Header

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

Description

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

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

Members

```
namespace tbb {
```

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

3.7.1.1 void add(const Item& item)

Requirements

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

Effects

Adds item to collection of work items to be processed.

3.8 pipeline Class

Summary

Class that performs pipelined execution.

Syntax

```
class pipeline;
```

Header

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

Description

A pipeline represents pipelined application of a series of filters to a stream of items. Each filter operates in a particular mode: parallel, serial in order, or serial out of order (MacDonald 2004). See class filter (3.8.6) for details.

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

1. Derive classes f_i from filter. The constructor for f_i specifies its mode as a parameter to the constructor for base class filter (3.8.6.1).
2. Override virtual method filter::operator() to perform the filter's action on the item, and return a pointer to the item to be processed by the next filter. The first filter f_0 generates the stream. It should return NULL if there are no more items in the stream. The return value for the last filter is ignored.
3. Create an instance of class pipeline.

4. Create instances of the filters f_i and add them to the pipeline, in order from first to last. An instance of a filter can be added at most once to a pipeline. A filter should never be a member of more than one pipeline at a time.
5. Call method `pipeline::run`. The parameter `max_number_of_live_tokens` puts an upper bound on the number of stages that will be run concurrently. Higher values may increase concurrency at the expense of more memory consumption from having more items in flight. See the Tutorial, in the section on class `pipeline`, for more about effective use of `max_number_of_live_tokens`.

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

Members

```
namespace tbb {
    class pipeline {
    public:
        pipeline();
        ~pipeline();[2]
        void add_filter( filter& f );
        void run( size_t max_number_of_live_tokens );
        void clear();
    };
}
```

3.8.1 pipeline()

Effects

Constructs pipeline with no filters.

3.8.2 ~pipeline()

Effects

Removes all filters from the pipeline and destroys the pipeline

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

3.8.3 void add_filter(filter& f)

Effects

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

3.8.4 void run(size_t max_number_of_live_tokens)

Effects

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

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

3.8.5 void clear()

Effects

Removes all filters from the pipeline.

3.8.6 filter Class

Summary

Abstract base class that represents a filter in a pipeline.

Syntax

```
class filter;
```

Header

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

Description

A `filter` represents a filter in a `pipeline` (3.8). There are three modes of filters:

- A **`parallel`** filter can process multiple items in parallel and in no particular order.
- A **`serial_out_of_order`** filter processes items one at a time, and in no particular order.

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

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

Class `filter` should only be used in conjunction with class `pipeline` (3.8).

CAUTION: TBB 2.0 and prior treated parallel input stages as serial. TBB 2.1 can execute a parallel input stage in parallel, so if you specify such a stage, ensure that its `operator()` is thread safe.

Members

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

Example

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

3.8.6.1 filter(mode filter_mode)

Effects

Constructs a filter of the specified `mode`.

NOTE: Earlier versions of TBB had a similar constructor with a `bool` argument `is_serial`. That constructor exists but is deprecated (Section A.2.1).

3.8.6.2 ~filter()

Effects

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

3.8.6.3 bool is_serial() const

Returns

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

3.8.6.4 bool is_ordered() const

Returns

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

3.8.6.5 virtual void* operator()(void * item)

Description

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

Returns

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

3.8.6.6 virtual void finalize(void * item)

Description

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

Effects

The default definition has no effect.

3.9 parallel_sort<RandomAccessIterator, Compare> 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 14.

Table 14: Requirements on Value Type T of RandomAccessIterator for parallel_sort

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

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

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

Complexity

`parallel_sort` is comparison sort with an average time complexity of O(N log (N)), where N is the number of elements in the sequence. When worker threads are

available (8.2.1), `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 Containers

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

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

4.1 Container Range Concept

Summary

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

Requirements

A Container Range is a Range (3.2) with the further requirements listed in Table 15.

Table 15: Requirements on a Container Range R (In Addition to Table 7)

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

Model Types

Classes `concurrent_hash_map` (4.2.4) and `concurrent_vector` (4.4.5) both have member types `range_type` and `const_range_type` that model a Container Range.

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

4.2 concurrent_hash_map<Key,T,HashCompare,Allocator> Template Class

Summary

Template class for associative container with concurrent access.

Syntax

```
template<typename Key, typename T, typename HashCompare,
         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 be other elements in flight but not in the map as described in Section 4.2.3. The interface resembles typical STL associative containers, but with some differences critical to supporting concurrent access. It meets the Container Requirements of the ISO C++ standard.

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

Type HashCompare specifies how keys are hashed and compared for equality. It must model the HashCompare concept in Table 16.

Table 16: 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`[3]	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

[3] Prior versions of this document accidentally omitted the trailing const from hash. TBB 2.1 enforces the const requirement.

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.

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

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 Allocator=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 Allocator allocator_type;

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

```
        concurrent_hash_map operator=( const concurrent_hash_map&
);
        void clear();
        allocator_type get_allocator() const;

        // concurrent access
        class const_accessor;
        class accessor;

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

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

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

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

```
            equal_range( const Key& key ) const;
    };

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

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

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

Exception Safey

The following functions must not throw exceptions:

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

The following hold true:

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

4.2.1 Whole Table Operations

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

4.2.1.1 concurrent_hash_map()

Effects

Constructs empty table.

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

Effects

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

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

Effects

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

4.2.1.4 ~concurrent_hash_map()

Effects

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

4.2.1.5 concurrent_hash_map& operator= (concurrent_hash_map& source)

Effects

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

Returns

Reference to the destination table.

4.2.1.6 void swap(concurrent_hash_map& table)

Effects

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

4.2.1.7 void clear()

Effects

Erases all key-value pairs from the table.

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

4.2.1.8 allocator_type get_allocator() const

Returns

Copy of allocator used to construct table.

4.2.2 Concurrent Access

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

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

Table 17: Differences Between const_accessor and accessor

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

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

4.2.2.1 const_accessor

Summary

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

Syntax

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

Header

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

Description

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

Members

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

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

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

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

4.2.2.1.1 bool empty() const

Returns

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

4.2.2.1.2 void release()

Effects

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

4.2.2.1.3 const value_type& operator*() const

Effects

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

Returns

Const reference to key-value pair.

4.2.2.1.4 const value_type* operator->() const

Returns

```
&operator*()
```

4.2.2.1.5 const_accessor()

Effects

Constructs const_accessor that points to nothing.

4.2.2.1.6 ~const_accessor

Effects

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

4.2.2.2 accessor

Summary

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

Syntax

```
template<typename Key, typename T, typename HashCompare,
         typename Allocator>
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 Allocator>
    class
concurrent_hash_map<Key,T,HashCompare,Allocator>::accessor:

concurrent_hash_map<Key,T,HashCompare,Allocator>::const_accessor
{
    public:
        typedef std::pair<const Key,T> value_type;
```

```
        value_type& operator*() const;
        value_type* operator->() const;
    };
}
```

4.2.2.2.1 value_type& operator*() const

Effects

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

Returns

Reference to key-value pair.

4.2.2.2.2 value_type* operator->() const

Returns

```
&operator*()
```

4.2.3 Concurrent Operations

The operations `count`, `find`, `insert`, and `erase` are the only operations that may be concurrently invoked on the same `concurrent_hash_map`. These operations search the table for a key-value pair that matches a given key. The `find` and `insert` methods each have two variants. One takes a const_accessor argument and provides read-only access to the desired key-value pair. The other takes an accessor argument and provides write access.

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

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

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

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

- To construct the resource: Obtain an `accessor` to the key in the map before constructing the resource.

- To destroy the resource: Obtain an accessor to the key, destroy the resource, and then erase the key using the accessor.

Below is a sketch of how this can be done.

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

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

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

4.2.3.1 size_type count(const Key& key) const

Returns

1 if map contains key; 0 otherwise.

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

Effects

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

Returns

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

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

Effects

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

Returns

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

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

Effects

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

Returns

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

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

Effects

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

Returns

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

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

Effects

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

Returns

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

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

Effects

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

Returns

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

4.2.3.8 bool insert(const value_type& value)

Effects

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

Returns

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

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

Effects

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

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

4.2.3.10 bool erase(const Key& key)

Effects

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

Returns

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

4.2.3.11 bool erase(const_accessor& item_accessor)

Requirements

item_accessor`.empty()==false`

Effects

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

Returns

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

4.2.3.12 bool erase(accessor& item_accessor)

Requirements

item_accessor.empty()==false

Effects

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

Returns

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

4.2.4 Parallel Iteration

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

4.2.4.1 const_range_type range(size_t grainsize=1) const

Effects

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

Returns

const_range_type object for the table.

4.2.4.2 range_type range(size_t grainsize=1)

Returns

range_type object for the table.

4.2.5 Capacity

4.2.5.1 size_type size() const

Returns

Number of key-value pairs in the table.

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

4.2.5.2 bool empty() const

Returns

`size()==0`.

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

4.2.5.3 size_type max_size() const

Returns

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

4.2.6 Iterators

Template class `concurrent_hash_map` supports forward iterators; that is, iterators that can advance only forwards across a table. Reverse iterators are not supported. Modification of a table invalidates any existing iterators that point into the table.

4.2.6.1 iterator begin()

Returns

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

4.2.6.2 iterator end()

Returns

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

4.2.6.3 const_iterator begin() const

Returns

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

4.2.6.4 const_iterator end() const

Returns

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

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

Returns

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

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

Description

See 4.2.6.5.

4.2.7 Global Functions

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

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

Returns

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

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

Returns

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

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

Effects

a`.swap(`b`)`

4.3 concurrent_queue<T,Allocator> Template Class

Summary

Template class for queue with concurrent operations.

Syntax

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

Header

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

Description

A 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.

The interface is different than for an STL std::queue because concurrent_queue is designed for concurrent operations.

Table 18: 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
size()	Returns number of items in queue	Returns number of pushes minus the number of pops. Waiting push or pop operations are included in the difference. The size() is negative if there are pops waiting for corresponding pushes.
Copy and pop item from queue q.	x=q.front(); q.pop()	*q*.pop(x)

Feature	STL std::queue	concurrent_queue

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

pop of empty queue	not allowed	Wait until item becomes available.

CAUTION: If the push or pop operations block, they block using user-space locks, which can waste processor resources when the blocking time is long. Class concurrent_queue is designed for situations where the blocking time is typically short relative to the rest of the application time.

Members

```
namespace tbb {
    template<typename T, typename
Allocator=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 Allocator& a = Allocator());
        concurrent_queue(const concurrent_queue& src,
                         const Allocator& a = Allocator());
        template<typename InputIterator>
        concurrent_queue(InputIterator first, InputIterator last,
                         const Allocator& a = Allocator());
        ~concurrent_queue();

        void push( 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 );
        Allocator get_allocator() const;

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

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

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

Effects

Constructs empty queue.

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

Effects

Constructs a copy of *src*.

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

Effects

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

4.3.4 ~concurrent_queue()

Effects

Destroys all items in the queue.

4.3.5 void push(const T& source)

Effects

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

4.3.6 void pop(T& destination)

Effects

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

4.3.7 bool pop_if_present(T& destination)

Effects

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

Returns

True if value was popped; false otherwise.

4.3.8 void clear()

Effects

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

4.3.9 size_type size() const

Returns

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

4.3.10 bool empty() const

Returns

`size()<=0`

4.3.11 size_type capacity() const

Returns

Maximum number of values that the queue can hold.

4.3.12 void set_capacity(size_type capacity)

Effects

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

4.3.13 Allocator get_allocator() const

Returns

Copy of allocator used to construct the queue.

4.3.14 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);
    for( concurrent_queue<int>::const_iterator i(queue.begin());
i!=queue.end(); ++i )
        cout << *i << " ";
    cout << endl;
    return 0;

}
```

4.3.14.1 iterator begin()

Returns

`iterator` pointing to beginning of the queue.

4.3.14.2 iterator end()

Returns

`iterator` pointing to end of the queue.

4.3.14.3 const_iterator begin() const

Returns

`const_iterator` with pointing to beginning of the queue.

4.3.14.4 const_iterator end() const

Returns

`const_iterator` pointing to end of the queue.

4.4 concurrent_vector

Summary

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

Syntax

```
template<typename T, class Allocator=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
Allocator=cache_aligned_allocator<T> >
    class concurrent_vector {
    public:
        typedef size_t size_type;
        typedef allocator-A-rebound-for-T[4] allocator_type;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T* pointer;
        typedef const T *const_pointer;
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef implementation-defined reverse_iterator;
        typedef implementation-defined const_reverse_iterator;

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

        // Constructors
        explicit concurrent_vector( const allocator_type& a =
```

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

```
                                        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
        size_type grow_by( size_type delta );
        size_type grow_by( size_type delta, const T& t );
        void grow_to_at_least( size_type n );
        size_type 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();
        const_reverse_iterator rbegin() const;
        const_reverse_iterator rend() 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.[5] 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-initialized memory.

Otherwise the program's behavior is undefined.

If an exception is thrown during a growth (4.4.3) or assignment (4.4.1) operation, the instance of the vector becomes broken unless it is stated otherwise in the method's description.

- Some items added to a broken vector may be zero-filled instead of default-constructed.
- A broken vector cannot be repaired. It is unable to grow anymore.
- Size and capacity reported by a broken vector are incorrect, and calculated as if the failed operation were successful
- Access to the added items via `operator[]`, `back()`, or iterators is unsafe for an invalid vector. Access via `at()` may cause an exception to be thrown.

If a concurrent growth operation successfully completes, all elements it added to the vector remain valid and accessible even if a subsequent growth operations fails.

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 `compact()`merges several smaller arrays into a single contiguous array, which may improve access time.

[5] 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.

4.4.1 Construction, Copy, and Assignment

Safety

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

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

Effects

Constructs empty vector using optionally specified allocator instance.

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

Effects

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

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

Effects

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

4.4.1.4 concurrent_vector(const concurrent_vector& src)

Effects

Constructs copy of *src*.

4.4.1.5 concurrent_vector& operator=(const concurrent_vector& src)

Effects

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

Returns

Reference to left hand side.

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

Assign contents of *src* to *this.

Returns

Reference to left hand side.

4.4.1.7 void assign(size_type n, const_reference t)

Assign *n* copies of *t*.

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

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

4.4.2 Whole Vector Operations

Safety

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

4.4.2.1 void reserve(size_type n)

Effects

Reserves space for at least *n* elements.

Throws

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

Safety

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

4.4.2.2 void compact()

Effects

Compacts the internal representation to reduce fragmentation.

4.4.2.3 void swap(concurrent_vector& x)

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

4.4.2.4 void clear()

Effects

Erases all elements. Afterwards, `size()==0`. Does not free internal arrays.[6]

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

4.4.2.5 ~concurrent_vector()

Effects

Erases all elements and destroys the vector.

4.4.3 Concurrent Growth

Safety

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

4.4.3.1 size_type grow_by(size_type delta, const_reference t=T())

Effects

Atomically appends *delta* copies of *t* to the end of the vector. If *t* is not specified, the new elements are default constructed.

Returns

Old size of the vector. If it returns *k*, then the new elements are at the half-open index range [*k*..*k*+*delta*).

4.4.3.2 void grow_to_at_least(size_type n)

Effects

Grows the vector until it has at least *n* elements. The new elements are default constructed.

---

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

4.4.3.3 size_t push_back(const_reference value);

Effects

Atomically appends copy of `value` to the end of the vector.

Returns

Index of the copy.

4.4.4 Access

Safety

The methods described in this section may be concurrently invoked on the same vector as methods for concurrent growth (4.4.3). However, the returned reference may be to an element that is being concurrently constructed.

4.4.4.1 reference operator[](size_type index)

Returns

Reference to element with the specified index.

4.4.4.2 const_refrence operator[](size_type index) const

Returns

Const reference to element with the specified index.

4.4.4.3 reference at(size_type index)

Returns

Reference to element at specified index.

Throws

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

4.4.4.4 const_reference at(size_type index) const

Returns

Const reference to element at specified index.

Throws

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

4.4.4.5 reference front()

Returns

`(*this)[0]`

4.4.4.6 const_reference front() const

Returns

`(*this)[0]`

4.4.4.7 reference back()

Returns

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

4.4.4.8 const_reference back() const

Returns

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

4.4.5 Parallel Iteration

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

4.4.5.1 range_type range(size_t grainsize=1)

Returns

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

4.4.5.2 const_range_type range(size_t grainsize=1) const

Returns

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

4.4.6 Capacity

4.4.6.1 size_type size() const

Returns

Number of elements in the vector. The result may include elements that are under construction by concurrent calls to any of the growth methods (4.4.3).

4.4.6.2 bool empty() const

Returns

`size()==0`

4.4.6.3 size_type capacity() const

Returns

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

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

4.4.6.4 size_type max_size() const

Returns

Highest possible size of the vector could reach.

4.4.7 Iterators

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

4.4.7.1 iterator begin()

Returns

`iterator` pointing to beginning of the vector.

4.4.7.2 const_iterator begin() const

Returns

`const_iterator` pointing to beginning of the vector.

4.4.7.3 iterator end()

Returns

`iterator` pointing to end of the vector.

4.4.7.4 const_iterator end() const

Returns

`const_iterator` pointing to end of the vector.

4.4.7.5 reverse_iterator rbegin()

Returns

`reverse iterator` pointing to beginning of reversed vector.

4.4.7.6 const_reverse_iterator rbegin() const

Returns

`const_reverse_iterator` pointing to beginning of reversed vector.

4.4.7.7 iterator rend()

Returns

`const_reverse_iterator` pointing to end of reversed vector.

4.4.7.8 const_reverse_iterator rend()

Returns

`const_reverse_iterator` pointing to end of reversed vector.

5 Memory Allocation

This section describes classes related to memory allocation.

5.1 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 19 summarizes the allocator concept. Here, A and B represent instances of the allocator class.

Table 19: 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.

Pseudo-Signature	Semantics
`void A::deallocate( T* p, size_t n )`	Deallocate n values.
`size_type A::max_size() const throw()`	Maximum plausible 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` (5.2), `scalable_allocator` (5.3), and `cached_aligned_allocator` (5.4) model the Allocator concept.

5.2 tbb_allocator<T> 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 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 TBB malloc library is being used. Details are in Section 2.7.2.

5.3 scalable_allocator<T> 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 19. 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 (5.1).

Acknowledgement

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

5.3.1 C Interface to Scalable Allocator

Summary

Low level interface for scalable memory allocation.

Syntax

```
extern "C" {
    void* scalable_calloc ( size_t nobj, size_t size );
    void  scalable_free( void* ptr );
    void* scalable_malloc( size_t size );
    void* scalable_realloc( void* ptr, size_t size );
}
```

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 the C standard library function *x*. Storage allocated by a scalable_*x* function should be freed or resized by a scalable_*x* function, 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.

5.4 cache_aligned_allocator<T> 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 19. 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`.

5.4.1 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.

5.4.2 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.

5.4.3 char* _Charalloc(size_type size)

NOTE: This method is provided only on 64-bit Windows* 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.

5.5 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();
    };
}
```

5.5.1 aligned_space()

Effects

None. Does not invoke constructors.

5.5.2 ~aligned_space()

Effects

None. Does not invoke destructors.

5.5.3 T* begin()

Returns

Pointer to beginning of storage.

5.5.4 T* end()

Returns

`begin()+N`

6 Synchronization

The library supports mutual exclusion and atomic operations.

6.1 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.

6.1.1 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 20 shows the requirements for the Mutex concept for a mutex type M

Table 20: Mutex Concept

Pseudo-Signature	Semantics
`M()`	Construct unlocked mutex.
`~M()`	Destroy unlocked mutex.
`typename M::scoped_lock`	Corresponding scoped-lock type.

Pseudo-Signature	Semantics
`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 reader-writer mutex; false otherwise.
`static const bool M::is_fair_mutex`	True if mutex is fair; false otherwise.

Table 21 summarizes the classes that model the Mutex concept.

Table 21: 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 rationale for null mutexes.

6.1.2 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 (6.1.1). 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 (6.1.1).

6.1.3 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` (6.1.2), 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 (6.1.1).

6.1.4 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 (6.1.1). 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 (6.1.1).

6.1.5 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 (6.1.1). 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 (6.1.1).

6.1.6 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 22 shows the requirements for a ReaderWriterMutex `RW`. They form a superset of the Mutex Concept (6.1.1).

Table 22: 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

`spin_rw_mutex` (6.1.7) and `queuing_rw_mutex` (6.1.8) model the ReaderWriterMutex concept.

6.1.6.1 ReaderWriterMutex()

Effects

Constructs unlocked `ReaderWriterMutex`.

6.1.6.2 ~ReaderWriterMutex()

Effects

Destroys unlocked `ReaderWriterMutex`. The effect of destroying a locked `ReaderWriterMutex` is undefined.

6.1.6.3 ReaderWriterMutex::scoped_lock()

Effects

Constructs a `scoped_lock` object that does not hold a lock on any mutex.

6.1.6.4 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.

6.1.6.5 ReaderWriterMutex::~scoped_lock()

Effects

If the object holds a lock on a `ReaderWriterMutex`, releases the lock.

6.1.6.6 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.

6.1.6.7 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.

6.1.6.8 void ReaderWriterMutex:: scoped_lock::release()

Effects

Releases lock. The effect is undefined if no lock is held.

6.1.6.9 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.

6.1.6.10 bool ReaderWriterMutex:: scoped_lock::downgrade_to_reader()

Effects

Changes writer lock to a reader lock. The effect is undefined if the object does not already hold a writer lock.

Returns

`false` if lock was released and reacquired; `true` otherwise.

NOTE: Intel's current implementations for `spin_rw_mutex` and `queuing_rw_mutex` always return `true`. Different implementations might sometimes return `false`.

6.1.7 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 (6.1.6). 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 (6.1.6).

6.1.8 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 (6.1.6). 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 (6.1.6).

6.1.9 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 (6.1.1) 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 (6.1.1).

6.1.10 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 (6.1.6) 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 (6.1.6).

6.2 atomic<T> 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 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. The specializations `atomic<void*>` and `atomic<bool>` do not allow arithmetic.

Some of the methods have template method variants that permit more selective memory fencing. On IA-32 and EM64T processors, they have the same effect as the non-templated variants. On 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 23 shows the fencing for the non-template form.

Table 23: 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 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();

        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 );

        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;
```

6.2.1 enum memory_semantics

Description

Defines values used to select the template variants that permit more selective control over visibility of operations (see Table 23).

6.2.2 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*.

6.2.3 value_type fetch_and_increment()

Effects

Let *x* be the value of `*this`. Atomically updates *x* = *x* + 1.

Returns

Original value of *x*.

6.2.4 value_type fetch_and_decrement()

Effects

Let *x* be the value of `*this`. Atomically updates $x = x - 1$.

Returns

Original value of *x*.

6.2.5 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*.

6.2.6 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*.

7 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.

7.1 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
```

7.1.1 static tick_count tick_count::now()

Returns

Current wall clock timestamp.

7.1.2 tick_count::interval_t operator–(const tick_count& t1, const tick_count& t0)

Returns

Relative time that t1 occurred after t0.

7.1.3 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
```

7.1.3.1 interval_t()

Effects

Constructs `interval_t` representing zero time duration.

7.1.3.2 interval_t(double sec)

Effects

Constructs interval_t representing specified number of seconds.

7.1.3.3 double seconds() const

Returns

Time interval measured in seconds.

7.1.3.4 interval_t operator+=(const interval_t& i)

Effects

```
*this = *this + i
```

Returns

Reference to `*this`.

7.1.3.5 interval_t operator–=(const interval_t& i)

Effects

```
*this = *this – i
```

Returns

Reference to `*this`.

7.1.3.6 interval_t operator+ (const interval_t& i, const interval_t& j)

Returns

Interval_t representing sum of intervals *i* and *j*.

7.1.3.7 interval_t operator– (const interval_t& i, const interval_t& j)

Returns

`Interval_t` representing difference of intervals *i* and *j*.

8 Task Scheduling

The library provides a task scheduler, which is the engine that drives the algorithm templates (Section 3). You may also call it directly. Using tasks is often simpler and more efficient than using threads, because the task scheduler takes care of a lot of details.

The tasks are quanta of computation. The scheduler maps these onto physical threads. The mapping is non-preemptive. Each thread has a method `execute()`. Once a thread starts running `execute()`, the task is bound to that thread until `execute()` returns. During that time, the thread services other tasks only when it waits on child tasks, at which time it may run the child tasks, or if there are no pending child tasks, service tasks created by other threads.

The task scheduler is intended for parallelizing computationally intensive work. Because task objects are not scheduled preemptively, they should not make calls that might block for long periods, because meanwhile that thread is precluded from servicing other tasks.

CAUTION: There is no guarantee that *potentially* parallel tasks *actually* execute in parallel, because the scheduler adjusts actual parallelism to fit available worker threads. For example, given a single worker thread, the scheduler creates no actual parallelism. For example, it is generally unsafe to use tasks in a producer consumer relationship, because there is no guarantee that the consumer runs at all while the producer is running.

Potential parallelism is typically generated by a split/join pattern. Two basic patterns of split/join are supported. The most efficient is continuation-passing form, in which the programmer constructs an explicit “continuation” task. The parent task splits child tasks and specifies a continuation task to be executed when the children complete. The continuation inherits the parent’s ancestor. The parent task then exits; i.e., it does not block on its children. The children subsequently run, and after they (or their continuations) finish, the continuation task starts running. Figure 3 shows the steps. The running tasks at each step are shaded.

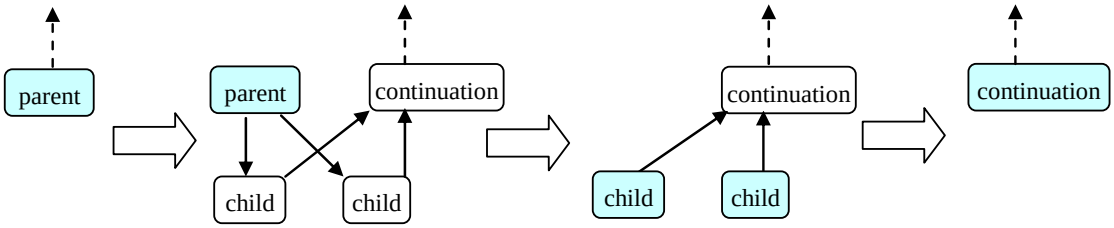

Figure 3: Continuation-passing Style

Explicit continuation passing is efficient, because it decouples the thread's stack from the tasks. However, it is more difficult to program. A second pattern is "blocking style", which uses implicit continuations. It is sometimes less efficient in performance, but more convenient to program. In this pattern, the parent task blocks until its children complete, as shown in Figure 4.

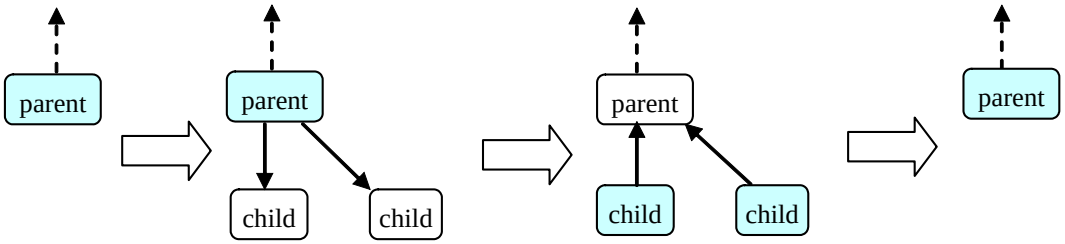

Figure 4: Blocking Style

The convenience comes with a price. Because the parent blocks, its thread's stack cannot be popped yet. The thread must be careful about what work it takes on, because continually stealing and blocking could cause the stack to grow without bound. To solve this problem, the scheduler constrains a blocked thread such that it never executes a task that is less deep than its deepest blocked task. This constraint may impact performance because it limits available parallelism, and tends to cause threads to select smaller (deeper) subtrees than they would otherwise choose.

8.1 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 an array of lists of `task`, where the list for the *ith* element corresponds to tasks at level *i* in the tree. The lists are manipulated in last-in first-out order. A task at level *i* spawns child tasks at level *i*+1. A thread chooses its next task according to the first rule below that applies:

1. The task returned by `task::execute()` that the thread invoked previously.
2. The task whose lastly completed child was completed by this thread.
3. A task from the deepest non-empty list in the array.
4. A task with affinity for the thread.
5. A task from the shallowest list in another randomly chosen thread's array.

Work stealing tends to strike a good balance between locality of reference, space efficiency, and parallelism. The work-stealing algorithm in the task scheduler is similar to that used by Cilk (Blumofe 1995). The notion of work-stealing dates back to the 1980s (Kumar 1987). The thread affinity support is more recent (Acar 2000).

8.2 task_scheduler_init Class

Summary

Class that represents thread's interest in task scheduling services.

Syntax

```
class task_scheduler_init;
```

Header

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

Description

A `task_scheduler_init` is either "active" or "inactive". Each thread that uses a `task` should have one active `task_scheduler_init` object that stays active over the duration that the thread uses `task` objects. A thread may have more than one active `task_scheduler_init` at any given moment.

The default constructor for a `task_scheduler_init` activates it, and the destructor uninitializes it. To defer initialization, pass the value `task_scheduler_init::deferred` to the constructor. Such a `task_scheduler_init` may be initialized later by calling method `initialize`. Destruction of an initialized `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. The Tutorial document says more about this topic.

To minimize time overhead, it is best to have a thread 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.

Important

The template algorithms (Section 3) implicitly use class `task`. Hence creating a `task_scheduler_init` is a prerequisite to using the template algorithms. The debug version of the library reports failure to create the `task_scheduler_init`.

Example

```
#include "tbb/task_scheduler_init"

int main() {
    task_scheduler_init init;
     ... use task or template algorithms here...
    return 0;
```

```
}
```

Members

```
namespace tbb {
    typedef unsigned-integral-type stack_size_type;

    class task_scheduler_init {
    public:
        static const int automatic = implementation-defined;
        static const int deferred = implementation-defined;
        task_scheduler_init( int number_of_threads=automatic,
                             stack_size_type thread_stack_size=0
);
        ~task_scheduler_init();
        void initialize( int number_of_threads=automatic );
        void terminate();
        static int default_num_threads();
    };
} // namespace tbb
```

8.2.1 task_scheduler_init(int number_of_threads=automatic, stack_size_type thread_stack_size=0)

Requirements

The value `number_of_threads` shall be one of the values in Table 24.

Effects

If `number_of_threads==task_scheduler_init::deferred`, nothing happens, and the `task_scheduler_init` remains inactive. Otherwise, the `task_scheduler_init` is activated as follows. If the thread has no other active `task_scheduler_init` objects, the thread allocates internal thread-specific resources required for scheduling `task` objects. If there were no threads with active `task_scheduler_init` objects yet, then internal worker threads are created as described in Table 24. These workers sleep until needed by the task scheduler.

The optional parameter `thread_stack_size` specifies the stack size of each worker thread. A value of 0 specifies use of a default stack size.

Table 24: Values for number_of_threads

number_of_threads	Semantics
`task_scheduler_init::automatic`	Let library determine `number_of_threads` based on hardware configuration.

number_of_threads	Semantics
task_scheduler_init::deferred	Defer activation actions.
positive integer	If no worker threads exist yet, create number_of_threads–1 worker threads. If worker threads exist, do not change the number of worker threads.

8.2.2 ~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.

8.2.3 void initialize(int number_of_threads=automatic)

Requirements

The task_scheduler_init shall be inactive.

Effects

Similar to constructor (8.2.1).

8.2.4 void terminate()

Requirements

The task_scheduler_init shall be active.

Effects

Deactivates the task_scheduler_init without destroying it. The description of the destructor (8.2.2) specifies what deactivation entails.

8.2.5 int default_num_threads()

Returns

One more than the number of worker threads that task_scheduler_init creates by default.

8.2.6 bool is_active() const

Returns

True if `*this` is active as described in Section 8.2; false otherwise.

8.2.7 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
...
        }
    }
}
```

8.3 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 25.

Table 25: Task Attributes

Attribute	Description
owner	The worker thread that is currently in charge of the task.
parent	Either null, or a pointer to another task whose refcount field will be decremented after the present task completes. Typically, the other task is the parent or a continuation of the parent.
depth	The depth of the task in the task tree.
refcount	The number of Tasks that have this is their parent. Increments and decrement of refcount are always atomic.

TIP: Always allocate memory for `task` objects using special overloaded new operators (8.3.2) 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 5.

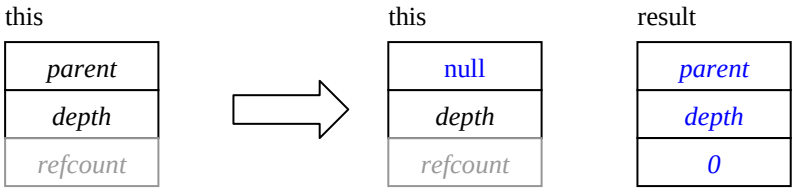

Figure 5: Example Effect Diagram

Conventions in these diagrams are as follows:

- The big arrow denotes the transition from the old state to the new state.
- Each task's state is shown as a box divided into *parent*, *depth*, and *refcount* sub-boxes.
- Gray denotes state that is ignored. Sometimes ignored state is simply left blank..
- Black denotes state that is read.
- Blue denotes state that is written.

Members

In the description below, types *proxy1*...*proxy4* are internal types. Methods returning such types should only be used in conjunction with the special overloaded new operators, as described in Section (8.3.2).

```
namespace tbb {
    class task {
    protected:
        task();

    public:
        virtual ~task() {}

        virtual task* execute() = 0;

        // task allocation and destruction
        static proxy1 allocate_root();
        proxy2 allocate_continuation();
        proxy3 allocate_child();
        proxy4 allocate_additional_child_of( task& t );

        // Explicit task destruction
        void destroy( task& victim );

        // Recycling
        void recycle_as_continuation();
        void recycle_as_child_of( task& new_parent );
        void recycle_to_reexecute();

        // task depth
        typedef implementation-defined-signed-integral-type
depth_type;
        depth_type depth() const;
        void set_depth( depth_type new_depth );
        void add_to_depth( int delta );

        // Synchronization
        void set_ref_count( int count );
        void wait_for_all();
        void spawn( task& child );
        void spawn( task_list& list );
        void spawn_and_wait_for_all( task& child );
        void spawn_and_wait_for_all( task_list& list );
        static void spawn_root_and_wait( task& root );
        static void spawn_root_and_wait( task_list& root );
```

```
        // task context
        static task& self();
        task* parent() const;
        bool is_stolen_task() const;

        // Cancellation
        bool cancel_group_execution();
        bool is_cancelled() const;

        // Affinity
        typedef implementation-defined-unsigned-type affinity_id;
        virtual void note_affinity( affinity_id id );
        void set_affinity( affinity_id id );
        affinity_id affinity() const;

        // task debugging
        enum state_type {
            executing,
            reexecute,
            ready,
            allocated,
            freed
        };
        int ref_count() const;
        state_type state() const;
    };
} // namespace tbb

void *operator new( size_t bytes, const proxy1& p );
void operator delete( void* task, const proxy1& p );
void *operator new( size_t bytes, const proxy2& p );
void operator delete( void* task, const proxy2& p );
void *operator new( size_t bytes, const proxy3& p );
void operator delete( void* task, const proxy3& p );
void *operator new( size_t bytes, proxy4& p );
void operator delete( void* task, proxy4& p );
```

8.3.1 task Derivation

Class `task` is an abstract base class. You **must** override method `task::execute`. Method `execute` should perform the necessary actions for running the task, and then return the next `task` to execute, or NULL if the scheduler should choose the next task to execute. Typically, if non-NULL, the returned task is one of the children of `this`. Unless one of the recycle/reschedule methods described in Section (8.3.4) 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 8.3.9.

8.3.1.1 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. If the task's *parent* is not null, then atomically decrements *parent->refcount*, and if becomes zero, puts the *parent* into the ready pool.
    b. Calls the task's destructor.
    c. Frees the memory of the task for reuse.
3. If the task has been marked for recycling:
    a. If marked by `recycle_to_reexecute`, puts the task back into the ready pool.
    b. Otherwise it was marked by `recycle_as_child` or `recycle_as_continuation`.

8.3.2 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.

8.3.2.1 new(task::allocate_root(task_group_context& group)) T

Allocate a `task` of type *T* with the specified cancellation group, with a depth of one more than the depth of the innermost `task` currently being executed by the current native thread. Figure 6 summarizes the state transition.

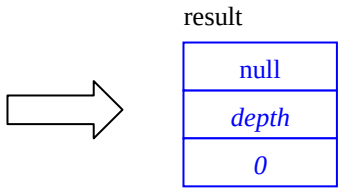

Figure 6: Effect of task::allocate_root()

Use method `spawn_root_and_wait` (8.3.6.7) to execute the `task`.

8.3.2.2 new(task::allocate_root())

Like `new(task::allocate_root(task_group_context&))` except that cancellation group is the current innermost cancellation group.

8.3.2.3 new(this. allocate_continuation()) T

Allocate and construct a task of type T at the same depth as this, and transfers the *parent* from this to the new task. No reference counts change. Figure 7 summarizes the state transition.

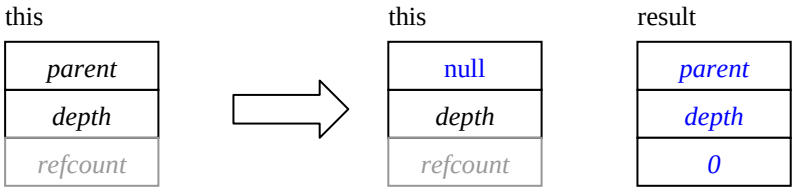

Figure 7: Effect of allocate_continuation()

8.3.2.4 new(this. allocate_child()) T

Effects

Allocates a `task` with a depth one more than this, with this as its *parent*. Figure 8 summarizes the state transition.

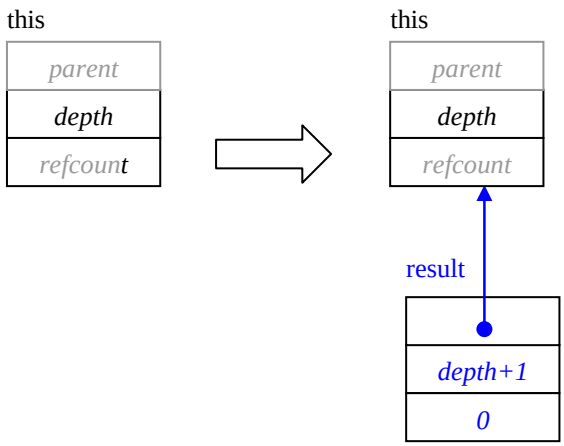

Figure 8: Effect of allocate_child()

If using explicit continuation passing, then the continuation, not the parent, should call the allocation method, so that *parent* is set correctly. The task `this` must be owned by the current thread.

If the number of tasks is not a small fixed number, consider building a `task_list` (8.5) of the children first, and spawning them with a single call to `task::spawn` (8.3.6.3). If a `task` must spawn some children before all are constructed, it should use `task::allocate_additional_child_of(*this)` instead, because that method atomically increments *refcount*, so that the additional child is properly accounted. However, if doing so, the `task` must protect against premature zeroing of *refcount* by using a blocking-style task pattern.

8.3.2.5 new(this.task::allocate_additional_child_of(parent))

Effects

Allocates a `task` as a child of another `task` *parent*. The result becomes a child of `parent`, not this. The `parent` may be owned by another thread, and may be already running or have other children running. The `task` object `this` must be owned by the current thread, and the result has the same owner as the current thread, not the parent. Figure 9 summarizes the state transition.

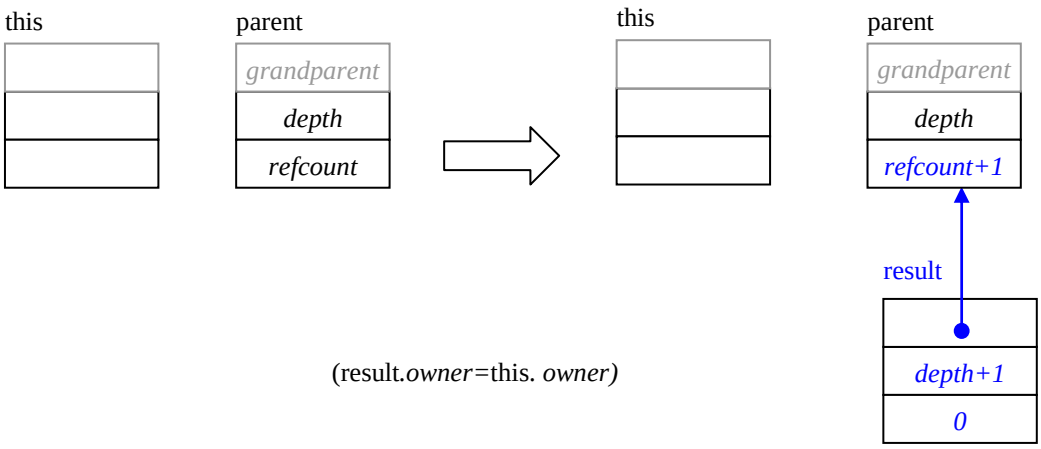

Figure 9: Effect of allocate_additional_child_of(parent)

Because `parent` may already have running children, the increment of parent.*refcount* is thread safe (unlike the other allocation methods, where the increment is not thread safe). When adding a child to a parent with other children running, it is up to the programmer to ensure that the parent's *refcount* does not prematurely reach 0 and trigger execution of the parent before the child is added.

8.3.3 Explicit task Destruction

Usually, a `task` is automatically destroyed by the scheduler after its method `execute` returns. But sometimes `task` objects are used idiomatically (e.g. for reference counting) without ever running `execute`. Such tasks should be disposed of with method `destroy`.

8.3.3.1 void destroy(task& victim)

Requirements

The reference count of *victim* should be 0. This requirement is checked in the debug version of the library. The calling thread must own `this`.

Effects

Calls destructor and deallocates memory for *victim*. If this has non-null *parent*, atomically decrements *parent->refcount*. The *parent* is **not** put into the ready pool if *parent->refcount* becomes zero. Figure 10 summarizes the state transition.

The implicit argument `this` is used internally, but not visibly affected. A `task` is allowed to destroy itself; e.g., "`this->destroy(*this)`" is permitted unless the `task` has been spawned but has not yet completed method `execute`.

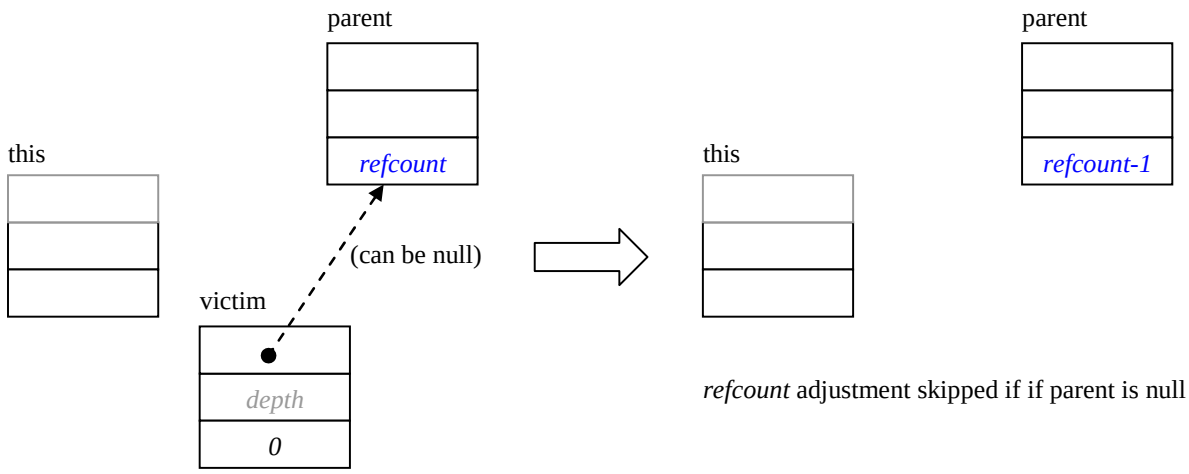

Figure 10: Effect of destroy(victim)

8.3.4 Recycling Tasks

It is often more efficient to recycle a `task` object rather than reallocate one from scratch. Often the parent can become the continuation, or one of the children.

8.3.4.1 void recycle_as_continuation()

Requirements

Must be called while method `execute()` is running.

The *refcount* for the recycled task should be set to `n`, where `n` is the number of children of the continuation task.

NOTE: The caller must guarantee that the task's *refcount* does not become zero until after the method `execute()` returns. If this is not possible, use the method `recycle_as_safe_continuation()` instead, and set refcount to `n+1`.

Effects

Causes `this` to not be destroyed when method `execute()` returns.

8.3.4.2 void recycle_as_safe_continuation()

Requirements

Must be called while method `execute()` is running.

The *refcount* for the recycled task should be set to `n+1`, where `n` is the number of children of the continuation task. The additional `+1` represents the task to be recycled.

Effects

Causes `this` to not be destroyed when method `execute()` returns.

This method avoids race conditions that can arise from using the method `recycle_as_continuation`. The race occurs when:

The method `execute()` recycles `this` as a continuation.

The continuation creates children.

All the children finish before method `execute()` completes, so the continuation executes before the scheduler is done running `this`, which corrupts the scheduler.

Method `recycle_as_safe_continuation` avoids this race because the additional `+1` in the *refcount* prevents the continuation from executing until the task completes.

8.3.4.3 void recycle_as_child_of(task& new_parent)

Requirements

Must be called while method `execute()` is running.

Effects

Causes `this` to become a child of *new_parent*, and not be destroyed when method `execute()` returns.

8.3.4.4 void recycle _to_reexecute()

Requirements

Only valid to call while method `execute()` is running. When method `execute()` returns, it must return a pointer to another `task`.

Effects

Causes `this` to be automatically spawned after `execute()` returns.

8.3.5 task Depth

For general fork-join parallelism, there is no need to explicitly set the depth of a task. However, in specialized task patterns that do not follow the fork-join pattern, it may be useful to explicitly set or adjust the depth of a task.

8.3.5.1 depth_type

The type `task::depth_type` is an implementation-defined signed integral type.

8.3.5.2 depth_type depth() const

Returns

Current *depth* attribute for the task.

8.3.5.3 void set_depth(depth_type new_depth)

Requirements

The value *new_depth* must be non-negative.

Effects

Sets the depth attribute of the task to *new_depth*. Figure 11 shows the update.

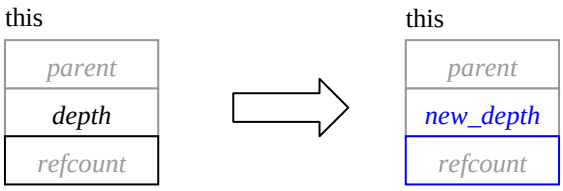

Figure 11: Effect of set_depth

8.3.5.4 void add_to_depth(int delta)

Requirements

The task must not be in the ready pool. The sum *depth*+*delta* must be non-negative.

Effects

Sets the depth attribute of the task to depth+*delta*. Figure 12 illustrates the effect. The update is not atomic.

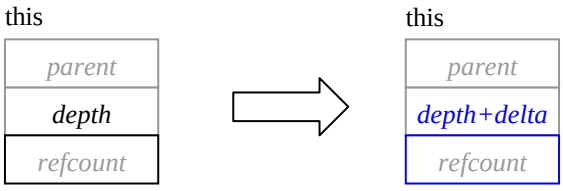

Figure 12: Effect of add_to_depth(delta)

8.3.6 Synchronization

Spawning a task *task* either causes the calling thread to invoke *task*.execute(), or causes *task* to be put into the ready pool. Any thread participating in task scheduling may then acquire the task and invoke *task*.execute(). Section 8.1 describes the structure of the ready pool.

The calls that spawn come in two forms:

- Spawn a single `task`.
- Spawn multiple `task` objects specified by a `task_list` and clear `task_list`.

The calls distinguish between spawning root tasks and child tasks. A root task is one that was created using method `allocate_root`.

Important

A `task` should not spawn any child until it has called method `set_ref_count` to indicate both the number of children and whether it intends to use one of the "wait_for_all" methods.

8.3.6.1 void set_ref_count(int count)

Requirements

count>0. If the intent is to subsequently spawn *n* children and wait, then *count* should be *n*+1. Otherwise *count* should be *n.*

Effects

Sets the *refcount* attribute to *count*.

8.3.6.2 void wait_for_all()

Requirements

refcount=n+1, where *n* is the number of children who are still running.

Effects

Executes tasks in ready pool until *refcount* is 1. Afterwards sets *refcount* to 0. Figure 13 summarizes the state transitions.

Also, `wait_for_all()`automatically resets the cancellation state of the `task_group_context` implicitly associated with the task (8.6), when all of the following conditions hold:

- The task was allocated without specifying a context.
- The calling thread is a user-created thread, not a 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.

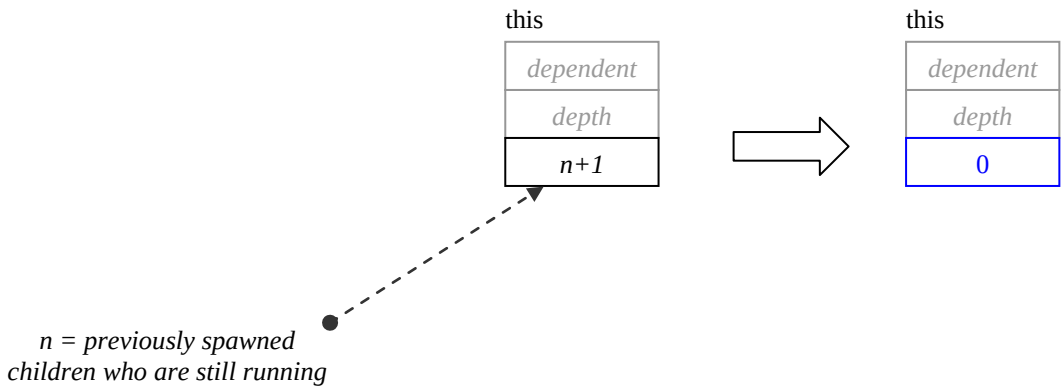

Figure 13: Effect of wait_for_all

8.3.6.3 void spawn(task& child)

Requirements

`child.refcount>0`

The calling thread must own `this` and *`child`*.

Effects

Puts the task into the ready pool and immediately returns. The `this` task that does the spawning must be owned by the caller thread. A `task` may spawn itself if it is owned by the caller thread. If no convenient `task` owned by the current thread is handy, use `task::self().spawn(task)` to spawn `task`.

The parent must call `set_ref_count` before spawning any child tasks, because once the child tasks are going, their completion will cause *refcount* to be decremented asynchronously. The debug version of the library detects when a required call to `set_ref_count` is not made, or is made too late.

8.3.6.4 void spawn (task_list& list)

Requirements

For each task in *list*, *refcount*>0. The calling thread must own `this` and each task in *list.* Each task in *list* must be the same value for its *depth* attribute.

Effects

Equivalent to executing `spawn` on each task in *list* and clearing *list*, but more efficient. If *list* is empty, there is no effect.

8.3.6.5 void spawn_and_wait_for_all(task& child)

Requirements

Any other children of `this` must already be spawned. The `task` *child* must have a non-null attribute *parent*. There must be a chain of *parent* links from the child to the calling `task`. Typically, this chain contains a single link. That is, *child* is typically a child of `this`.

Effects

Similar to `{spawn(`*task*`); wait_for_all();}`, but often more efficient. Furthermore, it guarantees that *task* is executed by the current thread. This constraint can sometimes simplify synchronization. Figure 14 illustrates the state transitions.

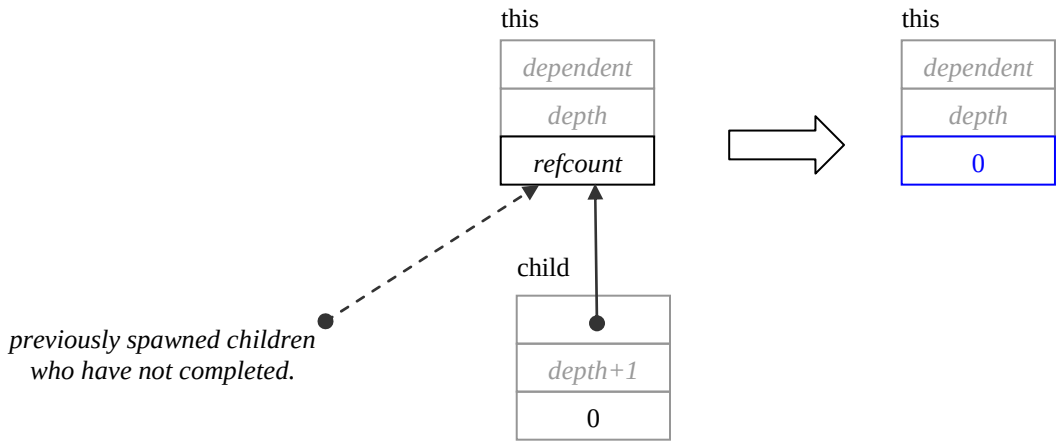

Figure 14: Effect of spawn_and_wait_for_all

8.3.6.6 void spawn_and_wait_for_all(task_list& list)

Effects

Similar to `{spawn(`*list*`); wait_for_all();}`, but often more efficient.

8.3.6.7 static void spawn_root_and_wait(task& root)

Requirements

The memory for task *root* was allocated by `task::allocate_root()`. The calling thread must own `root`.

Effects

Sets *parent* attribute of *root* to an undefined value and execute *root* as described in Section 8.3.1.1. Destroys *root* afterwards unless *root* was recycled.

8.3.6.8 static void spawn_root_and_wait(task_list& root_list)

Requirements

each `task` object *t* in *root_list* must meet the requirements in Section 8.3.6.7.

Effects

For each `task` object *t* in *root_list*, performs `spawn_root_and_wait(`*t*`)`, possibly in parallel. Section 8.3.6.7 describes the actions of `spawn_root_and_wait(`*t*`)`.

8.3.7 task Context

These methods expose relationships between `task` objects, and between `task` objects and the underlying physical threads.

8.3.7.1 static task& self()

Returns

Reference to innermost `task` that calling thread is running. A task is considered "running" if its methods `execute()`, `note_affinity()`, or destructor are running. If the calling thread is a user-created thread that is not running any task, `self()` returns a reference to an implicit dummy task associated with the thread.

8.3.7.2 task* parent() const

Returns

Value of the attribute *parent*. The result is an undefined value if the task was allocated by `allocate_root` and is currently running under control of `spawn_root_and_wait`.

8.3.7.3 bool is_stolen_task() const

Requirements

The attribute *parent* is not null and `this.execute()` is running. The calling task must not have been allocated with allocate_root.

Returns

true if the attribute *owner* of `this` is unequal to *owner* of *parent*.

8.3.8 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 8.6.

8.3.8.1 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.

8.3.8.2 bool is_cancelled() const

Returns

True if task's group has received a cancellation request; false otherwise.

8.3.9 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.

8.3.9.1 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 TBB implementation.

8.3.9.2 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.

8.3.9.3 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`.

8.3.9.4 affinity_id affinity() const

Returns

Affinity of this task as set by `set_affinity`.

8.3.10 task Debugging

Methods in this subsection are useful for debugging. They may change in future implementations.

8.3.10.1 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 26 describes valid states. Any other value is the result of memory corruption, such as using a `task` whose memory has been deallocated.

Table 26: 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 15 summarizes possible state transitions for a `task`.

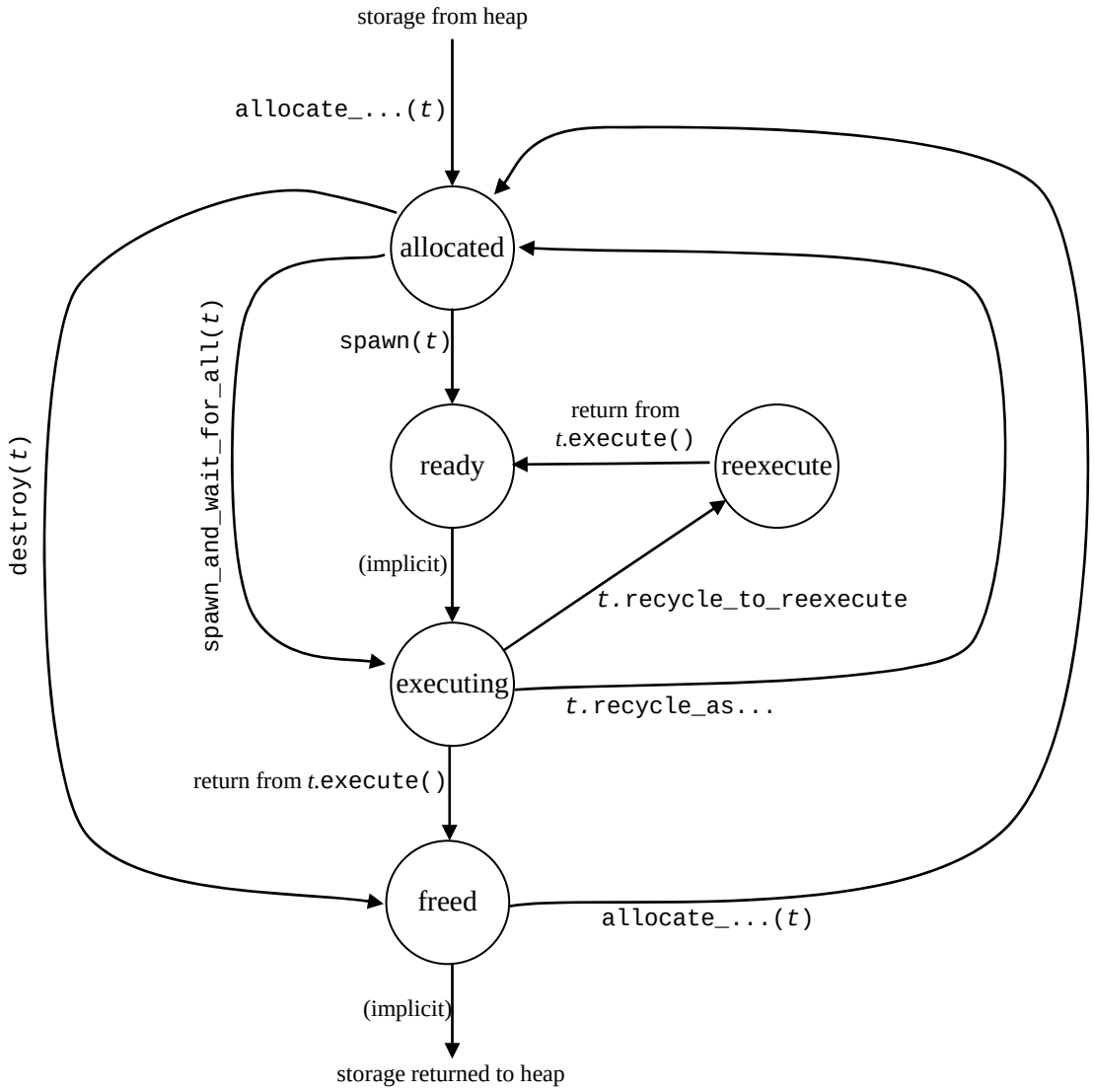

Figure 15: Typical task::state() Transitions

8.3.10.2 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*.

8.4 empty_task Class

Summary

Subclass of `task` that represents doing nothing.

Syntax

```
class empty_task;
```

Header

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

Description

An `empty_task` is a task that does nothing. It is useful as a continuation of a parent task when the continuation should do nothing except wait for its children to complete.

Members

```
namespace tbb {
    class empty_task: public task {
        /*override*/ task* execute() {return NULL;}
    };
}
```

8.5 task_list Class

Summary

List of `task` objects.

Syntax

```
class task_list;
```

Header

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

Description

A `task_list` is a list of references to `task` objects. The purpose of `task_list` is to allow a `task` to create a list of child tasks and spawn them all at once via the method `task::spawn(task_list&)`, as described in 8.3.6.4.

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();
    };
}
```

8.5.1 task_list()

Effects

Constructs an empty list.

8.5.2 ~task_list()

Effects

Destroys the list. Does not destroy the task objects.

8.5.3 bool empty() const

Returns

True if list is empty; false otherwise.

8.5.4 push_back(task& task)

Effects

Inserts a reference to *task* at back of the list.

8.5.5 task& task pop_front()

Effects

Removes a task reference from front of list.

Returns

The reference that was removed.

8.5.6 void clear()

Effects

Removes all task references from the list. Does not destroy the task objects.

8.6 task_group_context

Summary

A cancellable group of tasks.

Syntax

```
class task_group_context;
```

Header

```
#include “tbb/task.h”
```

Description

A task_group_context represents a group of tasks that can be cancelled together. The task_group_context objects form a forest of trees. Each tree’s root is a task_group_context constructed as isolated.

A task_group_context is cancelled explicitly by request, or implicitly when an exception is thrown out of a task. Cancelling a task_group_context causes the entire subtree rooted at it to be cancelled.

Each user thread that creates a task_scheduler_init (8.2) 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 (8.3.7.1).

Members

```
namespace tbb {
```

```
    class task_group_context {
    public:
        enum kind_t {
            isolated = implementation-defined,
            bound = implementation-defined
        };
        task_group_context( kind_t relation_to_parent = bound );
        ~task_group_context();
        void reset();
        bool cancel_group_execution();
        bool is_group_execution_cancelled() const;
    };
}
```

8.6.1 task_group_context(kind_t relation_to_parent=bound)

Effects

Constructs an empty `task_group_context`. If *`relation_to_parent`* is bound, the `task_group_context` becomes a child of the current innermost `task_group_context`. If *`relation_to_parent`* is `isolated`, it has no parent `task_group_context`.

8.6.2 ~task_group_context()

Effects

Destroys an empty `task_group_context`. It is a programmer error if there are still extant tasks in the group.

8.6.3 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.

8.6.4 bool is_group_execution_cancelled() const

Returns

True if group has received cancellation.

8.6.5 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.

8.7 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`.

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 } {}
    };
}
```

8.7.1 task_scheduler_observer()

Effects

Constructs instance with observing disabled.

8.7.2 ~task_scheduler_observer()

Effects

Disables observing. Waits for extant invocations of `on_scheduler_entry` or `on_scheduler_exit` to complete.

8.7.3 void observe(bool state=true)

Effects

Enables observing if state is true; disables observing if state is false.

8.7.4 bool is_observing() const

Returns

True if observing is enabled; false otherwise.

8.7.5 virtual void on_scheduler_entry(bool is_worker)

Description

The task scheduler invokes this method when a thread starts participating in task scheduling. If the instance of `task_scheduler_observer` is created 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 TBB scheduler; false otherwise.

NOTE: If a thread creates a `task_scheduler_observer` 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.

8.7.6 virtual void on_scheduler_exit(bool is_worker)

Description

The task scheduler invokes this method when a thread stops participating in task scheduling.

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.

Effects

The default behavior does nothing.

8.8 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 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.

8.8.1 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& t_k = *new(allocate_child()) T(...);  spawn(t_k);
        task& t_k-1= *new(allocate_child()) T(...);  spawn(t_k-1);
        ...
        task& t_1 = *new(allocate_child()) T(...);
        spawn_and_wait_for_all(t_1);
    }
    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`.

8.8.2 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.

8.8.2.1 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 8.8.1.
- 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.

8.8.2.2 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` (8.4).

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
    } else {
        set_ref_count(k);
        // Construct continuation
        C& c = allocate_continuation();
        // Recycle self as first child
        task& 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 8.8.1.

- 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.

8.8.3 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` (8.4).

```
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.

9 Exceptions

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 TBB run-time and handled as follows:

1. If the cancellation group for the task has already been cancelled, the exception is ignored.
2. Otherwise the exception is captured as follows:
    a. If it is a tbb_exception *x*, it is captured by x.move()
    b. If it is a std::exception x, it is captured as a `captured_exception(typeid(`*x*`).name(),x.what())`.
    c. Otherwise it is captured as a captured exception with implementation-specified value for `name()` and `what()`.
3. The captured exception is rethrown from the root of the cancellation group after all tasks in the group have completed or have been successfully cancelled.

9.1 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.

9.2 captured_exception

Summary

Class used by TBB to propagate exception that is not a `tbb_exception`.

Syntax

```
class captured_exception;
```

Header

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

Description

When a task throws an exception and the exception is not a `tbb_exception`, TBB converts the exception to a `captured_exception` before propagating it. Conversion is necessary so that the exception can be propagated across threads.

Members

```
namespace tbb {
    class captured_exception: public tbb_exception {
        captured_exception( const captured_exception& src );
        captured_exception( const char* name, const char* info );
        ~captured_exception() throw();
        captured_exception& operator=( const captured_exception&
src );
```

```
        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 9.1 describes the rest of the interface.

9.2.1 captured_exception(const char* name, const char* info)

Effects

Constructs a captured_exception with the specified *name* and *info*.

9.3 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

```
template<typename ExceptionData>
namespace tbb {
    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 9.1 describes the rest of the interface.

9.3.1 movable_exception(const ExceptionData& src)

Effects

Construct `movable_exception` containing copy of src.

9.3.2 ExceptionData& data() throw()

Returns

Reference to contained data.

9.3.3 const ExceptionData& data() const throw()

Returns

Const reference to contained data.

10 Threads

TBB provides a wrapper around the platform's native threads, based upon proposal N2497 for C++ 200x. Using this wrapper has two benefits:

- It makes threaded code portable across platforms.
- It eases later migration to ISO C++ 200x threads.

The significant departures from N2497 are shown in Table 27.

Table 27: Differences Between N2497 and TBB Thread Class

N2497	TBB
`std::thread`	`tbb::tbb_thread`
`std::this_thread`	`tbb::tbb_this_thread`
`std::this_thread::sleep ( system_time );`	`tbb::tbb_this_thread::sleep(tick_count::interval_t )`
rvalue reference parameters	Parameter changed to plain value, or function removed, as appropriate.
constructor for `std::thread` takes arbitrary number of arguments.	constructor for `tbb::tbb_thread` takes 0-3 arguments.

The name changes prevent identifier collisions when `using` directives are employed. The other changes are for compatibility with the current C++ standard or TBB. For example, constructors that have an arbitrary number of arguments require the variadic template features of C++ 200x.

CAUTION: Threads are heavy weight on most systems, and running too many threads on a system can seriously degrade performance. Consider using a task based solution instead if practical.

10.1 tbb_thread Class

Summary

Represents a thread of execution.

Syntax

```
class tbb_thread;
```

Header

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

Description

Cass `tbb_thread` provides a platform independent interface to native threads. An instance represents a thread. A platform-specific thread handle can be obtained via method `native_handle()`.

Members

```
namespace tbb {
    class tbb_thread {
    public:
#if _WIN32||_WIN64
        typedef HANDLE native_handle_type;
#else
        typedef pthread_t native_handle_type;
#endif // _WIN32||_WIN64

        class id;

        tbb_thread();
        template <typename F> explicit tbb_thread(F f);
        template <typename F, typename X> tbb_thread(F f, X x);
        template <typename F, typename X, typename Y>
            tbb_thread (F f, X x, Y y);
        ~tbb_thread();

        bool joinable() const;
        void join();
        void detach();
        id get_id() const;
        native_handle_type native_handle();

        static unsigned hardware_concurrency();
    };
}
```

10.1.1 tbb_thread()

Effects

Constructs `tbb_thread` that does not represent a thread of execution, with `get_id()==id()`.

template<typename F> tbb_thread(F f)

Effects

Construct tbb_thread that evaluates f()

10.1.2 template<typename F, typename X> tbb_thread(F f, X x)

Effects

Constructs tbb_thread that evaluates f(x).

10.1.3 template<typename F, typename X, typename Y> tbb_thread(F f, X x, Y y)

Effects

Constructs tbb_thread that evaluates f(x,y).

10.1.4 ~tbb_thread

Effects

`if( joinable() ) detach()`.

10.1.5 bool joinable() const

Returns

`get_id()!=id()`

10.1.6 void join()

Requirements

`joinable()==true`

Effects

Wait for thread to complete. Afterwards, `joinable()==false`.

10.1.7 void detach()

Requirements

`joinable()==true`

Effects

Sets *this to default constructed state and returns without blocking. The thread represented by *this continues execution.

10.1.8 id get_id() const

Returns

id of the thread, or a default-constructed id if *this does not represent a thread.

10.1.9 native_handle_type native_handle()

Returns

Native thread handle. The handle is a `HANDLE` on Windows* systems and a `pthread_t` on Linux* Systems and Mac OS* X Systems. For these systems, `native_handle()` returns 0 if `joinable()==false`.

10.1.10 static unsigned hardware_concurrency()

Returns

The number of hardware threads. For example, 4 on a system with a single Intel® Core™2 Quad processor.

10.2 tbb_thread:: id

Summary

Unique identifier for a thread.

Syntax

```
class tbb_thread::id;
```

Header

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

Description

A tbb_thread::id is an identifier value for a thread that remains unique over the thread's lifetime. A special value tbb_thread::id() represents no thread of execution. The instances are totally ordered.

Members

```
namespace tbb {
    class tbb_thread::id {
    public:
        id();
    };
    template<typename charT, typename traits>
    std::basic_ostream<charT, traits>&
        operator<< (std::basic_ostream<charT, traits> &out,
                    tbb_thread::id id)

    bool operator==(tbb_thread::id x, tbb_thread::id y);
    bool operator!=(tbb_thread::id x, tbb_thread::id y);
    bool operator<(tbb_thread::id x, tbb_thread::id y);
    bool operator<=(tbb_thread::id x, tbb_thread::id y);
    bool operator>(tbb_thread::id x, tbb_thread::id y);
    bool operator>=(tbb_thread::id x, tbb_thread::id y);
} // namespace tbb
```

10.3 this_tbb_thread Namespace

Description

Namespace this_tbb_thread contains global functions related to threading.

Members

```
namepace tbb {
    namespace this_tbb_thread {
        tbb_thread::id get_id();
        void yield();
        void sleep( const tick_count::interval_t );
    }
}
```

10.3.1 tbb_thread::id get_id()

Returns

Id of the current thread.

10.3.2 void yield()

Effects

Offers to suspend current thread so that another thread may run.

10.3.3 void sleep(const tick_count::interval_t & i)

Effects

Current thread blocks for at least time interval *i*.

Example

```
using namespace tbb;

void Foo() {
    // Sleep 30 seconds
    this_tbb_thread::sleep( tick_count::interval_t(30) );
}
```

11 References

Umut A. Acar, Guy E. Blelloch, Robert D. Blumofe, The Data Locality of Work Stealing. *ACM Symposium on Parallel Algorithms and Architectures* (2000):1-12.

Robert D.Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (July 1995):207–216.

H. Hinnant, L. Crowl, Beman Dawes, A. WIlliams, J. Garland, Multi-threading Library for Standard C++ (Revision 1), ISO/IEC JTC1 SC22 WG21 N2497.

Steve MacDonald, Duane Szafron, and Jonathan Schaeffer. Rethinking the Pipeline as Object-Oriented States with Transformations. *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (April 2004):12-21.

V. Kumar and V. N. Rao, "Parallel Depth First Search. Part II. Analysis", *International Journal of Parallel Programming* (December 1987): 501-519.

ISO/IEC 14882, *Programming Languages* – C++

Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. *Workshop on Language and Compilers for Parallel Computing* (LCPC 2001), Cumberland Falls, Kentucky Aug 2001. Lecture Notes in Computer Science 2624 (2003): 193-208.

S. G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts", *IEEE Transactions on Computers*, Vol. C-36 No. 11, Nov. 1987.

Appendix A Compatibility Features

This appendix describes features of TBB that remain for compatibility with previous versions. These features are deprecated and may disappear in future versions of TBB.

A.1 parallel_while Template Class

Summary

Template class that processes work items.

TIP: This class is deprecated. Use parallel_do (3.7) instead.

Syntax

```
template<typename Body>
class parallel_while;
```

Header

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

Description

A `parallel_while<Body>` performs parallel iteration over items. The processing to be performed on each item is defined by a function object of type `Body`. The items are specified in two ways:

- A stream of items.
- Additional items that are added while the stream is being processed.

Table 28 shows the requirements on the stream and body.

Table 28: parallel_while Requirements for Stream S and Body B

Pseudo-Signature	Semantics
`bool S::pop_if_present( B::argument_type& item )`	Get next stream item. `parallel_while` does not concurrently invoke the method on the same `this`.
`B::operator()( B::argument_type& item ) const`	Process `item`. `parallel_while` may concurrently invoke the operator for the same `this` but different `item`.

Pseudo-Signature	Semantics
`B::argument_type()`	Default constructor.
`B::argument_type( const B::argument_type& )`	Copy constructor.
`~B::argument_type()`	Destructor.

For example, a unary function object, as defined in Section 20.3 of the C++ standard, models the requirements for B. A `concurrent_queue` (4.3) models the requirements for S.

TIP: To achieve speedup, the grainsize of `B::operator()` needs to be on the order of at least ~10,000 instructions. Otherwise, the internal overheads of `parallel_while` swamp the useful work. The parallelism in `parallel_while` is not scalable if all the items come from the input stream. To achieve scaling, design your algorithm such that method add often adds more than one piece of work.

Members

```
namespace tbb {
    template<typename Body>
    class parallel_while {
    public:
        parallel_while();
        ~parallel_while();

        typedef typename Body::argument_type value_type;

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

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

A.1.1 parallel_while<Body>()

Effects

Constructs a `parallel_while` that is not yet running.

A.1.2 ~parallel_while<Body>()

Effects

Destroys a `parallel_while`.

A.1.3 Template <typename Stream> void run(Stream& stream, const Body& body)

Effects

Applies *body* to each item in *stream* and any other items that are added by method add. Terminates when both of the following conditions become true:

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

A.1.4 void add(const value_type& item)

Requirements

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

Effects

Adds item to collection of items to be processed.

A.2 Interface for constructing a pipeline filter

The interface for constructing a filter evolved over several releases of TBB. The two following subsections describe obsolete aspects of the interface.

A.2.1 filter::filter(bool is_serial)

Effects

Constructs a serial in order filter if is_serial is true, or a parallel filter if is_serial is false. This deprecated constructor is superseded by the constructor filter(filter::mode) described in Section 3.8.6.1.

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 29. If you define the old macros, TBB sets each undefined new macro in a way that duplicates the behavior the old macro settings.

The old TBB_DO_ASSERT enabled assertions, full support for Intel® Threading Tools, and performance warnings. These three distinct capabilities are now controlled by three separate macros as described in Section 2.6.

TIP: To enable all three capabilities with a single macro, define `TBB_USE_DEBUG` to be 1. If you had code under "`#if TBB_DO_ASSERT`" that should be conditionally included only when assertions are enabled, use "`#if TBB_USE_ASSERT`" instead.

Table 29: 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`