Intel(R) Threading Building Blocks

Reference Manual

Document Number 315415-001US

Revision: 1.6

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

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
315415-001	1.5	Add Partitioner concept as a preview feature for `parallel_for`, `parallel_reduce` and `parallel_scan`. Add method `recycle_as_safe_continuation`. Fix missing "Continuation-passing style" figure.	2007-Mar-1
	1.6	Fix typographical errors. Fix ParallelMerge example to make `is_divisible() const`.	2007-May-18

Contents

1 Overview 1

2 General Conventions 2

2.1 Notation....2
2.2 Terminology 3
2.2.1 Concept 3
2.2.2 Model 3
2.2.3 CopyConstructible 4
2.3 Identifiers 4
2.3.1 Case....4
2.3.2 Reserved Identifier Prefixes 4
2.4 Namespaces 4
2.4.1 tbb Namespace 4
2.4.2 tbb::internal Namespace 5
2.5 Thread Safety 5
2.6 Enabling Debugging Features 5
2.6.1 TBB_DO_ASSERT Macro 5
2.6.2 TBB_DO_THREADING_TOOLS Macro 5

3 Algorithms 7

3.1 Splittable Concept 7
3.1.1 split Class 8
3.2 Range concept 8
3.2.1 blocked_range<Value> Template Class 9
3.2.1.1 size_type....11
3.2.1.2 blocked_range(Value begin, Value end, size_t grainsize=1) 12
3.2.1.3 blocked_range(blocked_range& range, split)....12
3.2.1.4 size_type size() const....12
3.2.1.5 bool empty() const 13
3.2.1.6 size_type grainsize() const....13
3.2.1.7 bool is_divisible() const 13
3.2.1.8 const_iterator begin() const 13
3.2.1.9 const_iterator end() const....13
3.2.2 blocked_range2d Template Class....14
3.2.2.1 row_range_type 16
3.2.2.2 col_range_type 16
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) 16
3.2.2.4 blocked_range2d<RowValue,ColValue> (blocked_range2d& range, split) 16
3.2.2.5 bool empty() const 17
3.2.2.6 bool is_divisible() const 17
3.2.2.7 const row_range_type& rows() const....17
3.2.2.8 const col_range_type& cols() const 17

3.3 Preview Feature: Partitioner Concept....................17
3.3.1 simple_partitioner Class....................3
3.3.1.1 simple_partitioner()....................3
3.3.1.2 simple_partitioner(simple_partitioner &partitioner, split) .3
3.3.1.3 template<typename Range> bool should_execute_range (const Range &r, const task &t) 3
3.3.2 auto_partitioner Class....................3
3.3.2.1 auto_partitioner()....................4
3.3.2.2 auto_partitioner(auto_partitioner &partitioner, split).......4
3.3.2.3 template<typename Range> bool should_execute_range (const Range &r, const task &t) 4
3.4 parallel_for<Range,Body> Template Function 4
3.4.1 Using the Partitioner Preview Feature....................8
3.5 parallel_reduce<Range,Body> Template Function....................9
3.5.1 Using the Partitioner Preview Feature....................12
3.6 parallel_scan<Range,Body> Template Function....................13
3.6.1 pre_scan_tag and final_scan_tag Classes....................15
3.6.1.1 bool is_final_scan()....................16
3.6.2 Using the Partitioner Preview Feature....................16
3.7 parallel_while Template Class....................18
3.7.1 parallel_while<Body>()....................19
3.7.2 ~parallel_while<Body>() 19
3.7.3 Template <typename Stream> void run(Stream& stream, const Body& body)....................19
3.7.4 void add(const value_type& item)....................19
3.8 pipeline Class 20
3.8.1 pipeline()....................21
3.8.2 ~pipeline()....................21
3.8.3 void add_filter(filter& f)....................21
3.8.4 void run(size_t max_number_of_live_tokens)....................21
3.8.5 void clear() 21
3.8.6 filter Class....................22
3.8.6.1 filter(bool is_serial)....................22
3.8.6.2 ~filter()....................22
3.8.6.3 bool is_serial() const....................23
3.8.6.4 virtual void* operator()(void * item)....................23
3.9 parallel_sort<RandomAccessIterator, Compare> Template Function 23
4 Containers....................25
4.1 concurrent_hash_map<Key,T,HashCompare> Template Class 25
4.1.1 Whole Table Operations 27
4.1.1.1 concurrent_hash_map() 27
4.1.1.2 concurrent_hash_map(const concurrent_hash_map& table)....................27
4.1.1.3 ~concurrent_hash_map() 27
4.1.1.4 concurrent_hash_map& operator= (concurrent_hash_map& source)....................27
4.1.1.5 void clear()....................27
4.1.2 Concurrent Access....................28
4.1.2.1 const_accessor....................28
4.1.2.2 accessor....................30
4.1.3 Concurrent Operations....................30
4.1.3.1 bool find(const_accessor& result, const Key& key) const31
4.1.3.2 bool find(accessor& result, const Key& key)................31

4.1.3.3 bool insert(const_accessor& result, const Key& key) 31
4.1.3.4 bool insert(accessor& result, const Key& key) 32
4.1.3.5 bool erase(const Key& key) 32
4.1.4 Parallel Iteration 32
4.1.4.1 const_range_type range(size_t grainsize) const 32
4.1.4.2 range_type range(size_t grainsize) 33
4.1.5 Capacity 33
4.1.5.1 size_type size() const 33
4.1.5.2 bool empty() const 33
4.1.5.3 size_type max_size() const 33
4.1.6 Iterators 33
4.1.6.1 iterator begin() 33
4.1.6.2 iterator end() 33
4.1.6.3 const_iterator begin() const 34
4.1.6.4 const_iterator end() const 34
4.2 concurrent_queue<T> Template Class 34
4.2.1 concurrent_queue() 36
4.2.2 ~concurrent_queue() 36
4.2.3 void push(const T& source) 36
4.2.4 void pop(T& destination) 36
4.2.5 bool pop_if_present(T& destination) 36
4.2.6 size_type size() const 36
4.2.7 bool empty() const 37
4.2.8 size_type capacity() 37
4.2.9 void set_capacity(size_type capacity) 37
4.2.10 Iterators 37
4.2.10.1 iterator begin() 38
4.2.10.2 iterator end() 38
4.2.10.3 const_iterator begin() const 38
4.2.10.4 const_iterator end() const 38
4.3 concurrent_vector 38
4.3.1 Whole Vector Operations 39
4.3.1.1 concurrent_vector() 40
4.3.1.2 concurrent_vector(const concurrent_vector& src) 40
4.3.1.3 concurrent_vector& operator=(const concurrent_vector& src) 40
4.3.1.4 ~concurrent_vector() 40
4.3.1.5 void clear() 40
4.3.2 Concurrent Operations 40
4.3.2.1 size_type grow_by(size_type delta) 40
4.3.2.2 void grow_to_at_least(size_type n) 41
4.3.2.3 size_t push_back(const_reference value); 41
4.3.2.4 reference operator[](size_type index) 41
4.3.2.5 const_reference operator[](size_type index) const; 41
4.3.3 Parallel Iteration 41
4.3.3.1 range_type range(size_t grainsize) 42
4.3.3.2 const_range_type range(size_t grainsize) const 42
4.3.4 Capacity 42
4.3.4.1 size_type size() const 42
4.3.4.2 bool empty() const 42
4.3.4.3 size_type capacity() const 42
4.3.4.4 void reserve(size_type n) 42
4.3.4.5 size_type max_size() const 43
4.3.5 Iterators 43

4.3.5.1 iterator begin() 43
4.3.5.2 iterator end() 43
4.3.5.3 const_iterator begin() const 43
4.3.5.4 const_iterator end() const 43
4.3.5.5 iterator rbegin() 43
4.3.5.6 iterator rend() 43
4.3.5.7 const_reverse_iterator rbegin() const 44
4.3.5.8 const_ reverse_iterator rend() const 44

5 Memory Allocation 45
5.1 Allocator Concept 45
5.2 scalable_allocator<T> Template Class 46
5.3 cache_aligned_allocator<T> Template Class 47
5.3.1 pointer allocate(size_type n, void* hint=0) 49
5.3.2 void deallocate(pointer p, size_type n) 49
5.3.3 char* _Charalloc(size_type size) 49
5.4 aligned_space Template Class 50
5.4.1 aligned_space() 50
5.4.2 ~aligned_space() 50
5.4.3 T* begin() 51
5.4.4 T* end() 51

6 Synchronization 52
6.1 Mutexes 52
6.1.1 Mutex Concept 52
6.1.2 mutex Class 53
6.1.3 spin_mutex Class 54
6.1.4 queuing_mutex Class 54
6.1.5 ReaderWriterMutex Concept 55
6.1.5.1 ReaderWriterMutex() 56
6.1.5.2 ~ReaderWriterMutex() 56
6.1.5.3 ReaderWriterMutex::scoped_lock() 56
6.1.5.4 ReaderWriterMutex::scoped_lock(ReaderWriterMutex& rw, bool write =true) 56
6.1.5.5 ReaderWriterMutex::~scoped_lock() 56
6.1.5.6 void ReaderWriterMutex:: scoped_lock:: acquire(ReaderWriterMutex& rw, bool write=true) 56
6.1.5.7 bool ReaderWriterMutex:: scoped_lock::try_acquire(ReaderWriterMutex& rw, bool write=true) 57
6.1.5.8 void ReaderWriterMutex:: scoped_lock::release() 57
6.1.5.9 bool ReaderWriterMutex:: scoped_lock::upgrade_to_writer() 57
6.1.5.10 bool ReaderWriterMutex:: scoped_lock::downgrade_to_reader() 57
6.1.6 spin_rw_mutex Class 57
6.1.7 queuing_rw_mutex Class 58
6.2 atomic<T> Template Class 59
6.2.1 enum memory_semantics 60
6.2.2 value_type fetch_and_add(value_type addend) 61
6.2.3 value_type fetch_and_increment() 61
6.2.4 value_type fetch_and_decrement() 61
6.2.5 value_type compare_and_swap 61
6.2.6 Effect 61
6.2.7 value_type fetch_and_store(value_type new_value) 62

7 Timing......63

7.1 tick_count Class......63

7.1.1 static tick_count tick_count::now()......64

7.1.2 tick_count::interval_t operator−(const tick_count& t1, const tick_count& t0)......64

7.1.3 tick_count::interval_t Class......64

7.1.3.1 interval_t()......65

7.1.3.2 double seconds() const......65

7.1.3.3 interval_t operator+=(const interval_t& i)......65

7.1.3.4 interval_t operator−=(const interval_t& i)......65

7.1.3.5 interval_t operator+ (const interval_t& i, const interval_t& j)......65

7.1.3.6 interval_t operator− (const interval_t& i, const interval_t& j)......65

8 Task Scheduling......66

8.1 Scheduling Algorithm......67

8.2 task_scheduler_init Class......67

8.2.1 task_scheduler_init(int number_of_threads=automatic)......69

8.2.2 ~task_scheduler_init()......69

8.2.3 void initialize(int number_of_threads=automatic)......70

8.2.4 void terminate()......70

8.2.5 Mixing with OpenMP......70

8.3 task Class......71

8.3.1 task Derivation......73

8.3.1.1 Processing of execute()......74

8.3.2 task Allocation......74

8.3.2.1 new(task::allocate_root()) T......74

8.3.2.2 new(this. allocate_continuation()) T......75

8.3.2.3 new(this. allocate_child()) T......75

8.3.2.4 new(this.task::allocate_additional_child_of(parent))......76

8.3.3 Explicit task Destruction......77

8.3.3.1 void destroy(task& victim)......77

8.3.4 Recycling Tasks......78

8.3.4.1 void recycle_as_continuation()......78

8.3.4.2 Preview Feature: void recycle_as_safe_continuation()......78

8.3.4.3 void recycle_as_child_of(task& parent)......79

8.3.4.4 void recycle _to_reexecute()......79

8.3.5 task Depth......79

8.3.5.1 depth_type......79

8.3.5.2 depth_type depth() const......80

8.3.5.3 void set_depth(depth_type new_depth)......80

8.3.5.4 void add_to_depth(int delta)......80

8.3.6 Synchronization......80

8.3.6.1 void set_ref_count(int count)......81

8.3.6.2 void wait_for_all()......81

8.3.6.3 void spawn(task& child)......82

8.3.6.4 void spawn (task_list& list)......82

8.3.6.5 void spawn_and_wait_for_all(task& child)......83

8.3.6.6 void spawn_and_wait_for_all(task_list& list)......83

8.3.6.7 static void spawn_root_and_wait(task& root)......83

8.3.6.8 static void spawn_root_and_wait(task_list& root_list)......84

8.3.7 task Context......84

8.3.7.1 static task& self()......84

8.3.7.2 task* parent() const 84
8.3.7.3 bool is_stolen_task() const 84
8.3.8 task Debugging 84
8.3.8.1 state_type state() const 85
8.3.8.2 int ref_count() const 86
8.4 empty_task Class 87
8.5 task_list Class 87
8.5.1 task_list() 88
8.5.2 ~task_list() 88
8.5.3 bool empty() const 88
8.5.4 push_back(task& task) 88
8.5.5 task& task pop_front() 88
8.5.6 void clear() 89
8.6 Catalog of Recommended task Recurrence Patterns 89
8.6.1 Blocking Style With *k* Children 89
8.6.2 Continuation-Passing Style With *k* Children 90
8.6.2.1 Recycling Parent as Continuation 90
8.6.2.2 Recycling Parent as a Child 90
9 References 92

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:

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

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

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

- 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 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)`

[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.

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 concurrent operations on the same container object.

2.6 Enabling Debugging Features

The headers have two macros that control certain debugging features. In general, it is useful to compile with these features on for development code, and off for production code.

2.6.1 TBB_DO_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_DO_THREADING_TOOLS Macro

The macro `TBB_DO_THREADING_TOOLS` controls support for Intel® Threading Tools:

- Intel® Thread Profiler
- Intel® Thread Checker.

Define `TBB_DO_THREADING_TOOLS` as 1 to enable full support for these tools. The debug version of the library always has full support enabled.

Leave `TBB_DO_THREADING_TOOLS` undefined or zero to enable top performance, at the expense of turning off some support for tools. In the current implementation, the only features affected are `spin_mutex` (6.1.3) and `spin_rw_mutex` (6.1.6).

3 Algorithms

Most algorithms provided by the library are generic. They operate on all types that model the necessary concepts.

3.1 Splittable Concept

Summary

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

Requirements

Table 3 lists the requirements for a splittable type `X` with instance `x`.

Table 3: 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 two contexts:

- *Partition* 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 4 lists the requirements for a Range type `R`.

Table 4: Range Concept

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

Description

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

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

`blocked_range2d` (3.2.2) models a two-dimensional 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 5. 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 5: 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 *grain size* 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_DO_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

5. 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

Determine size of range.

Returns

`end()–begin()`

3.2.1.5 bool empty() const

Effects

Determine 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

Determine 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 5. 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( 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> (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.

3.2.2.5 bool empty() const

Effects

Determines if range is empty.

Returns

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

3.2.2.6 bool is_divisible() const

Effects

Determine if range can be split into subranges.

Returns

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

3.2.2.7 const row_range_type& rows() const

Returns

Range containing the rows of the value space.

3.2.2.8 const col_range_type& cols() const

Returns

Range containing the columns of the value space.

3.3 Preview Feature: Partitioner Concept

Summary

Requirements for a type that decides if a range (3.2) should be operated over by a task body or if the range should be further split.

Requirements

Table 6 lists the requirements for a Partitioner type `P`.

Table 6: Partitioner Concept

Pseudo-Signature	Semantics
`P::~P()`	Destructor

`template <typename Range>` `bool P::should_execute_range(const Range &r, const task &t)`	True if `r` should be passed to the body of `t`. False if `r` should instead be split.
`P::P( P& p, split )`	Split `p` into two partitioners.

Description

The Partitioner concept implements rules for deciding when a given range should no longer be subdivided, but should be operated over as a whole by a task's body.

The default behavior of the algorithms `parallel_for` (3.4), `parallel_reduce` (3.5) and `parallel_scan` (3.6) is to recursively split a range until no subrange remains that is divisible, as decided by the function `is_divisible` of the Range concept (3.2). The Partitioner concept models rules for the early termination of the recursive splitting of a range, providing ability to change the default behavior. A Partitioner object's decision making is implemented using two functions, a splitting constructor and the function `should_execute_range`.

Within the parallel algorithms, each Range object is now associated with a Partitioner object. Whenever a Range object is split using its splitting constructor to create two subranges, the associated Partitioner object is likewise split to create two matching Partitioner objects.

When a `parallel_for`, `parallel_reduce` or `parallel_scan` algorithm needs to decide whether to further subdivide a range, it invokes the function `should_execute_range` for the Partitioner object associated with the range. If the function `should_execute_range` returns true for the given range and task, then no further splits are performed on the range and the current task applies its body over the entire range.

Example

The following code defines a type `simple_partitioner` that models the Partitioner concept. It returns true from its function `should_execute_range` if the range function `is_divisible` returns false.

```
class simple_partitioner {
public:
  simple_partitioner() {}
  simple_partitioner(simple_partitioner &partitioner,
                         split) {}

  template <typename Range>
  inline bool should_execute_range(const Range &r, const task &t) {
    return ( !r.is_divisible() );
  }
};
```

This class encodes the default behavior, where a range is recursively split until it cannot be further subdivided.

Model Types

`simple_partitioner` (3.3.1) models the default behavior of splitting a range until it cannot be further subdivided.

`auto_partitioner` (3.3.2) models an adaptive behavior that monitors the work-stealing actions of the `task_scheduler` (8) to reduce the number of splits performed.

3.3.1 simple_partitioner Class

Summary

A class that models that default range splitting behavior of the `parallel_for` (3.4), `parallel_reduce` (3.5) and `parallel_scan` (3.6) algorithms, where a range is recursively split until it cannot be further subdivided.

Syntax

```
class simple_partitioner;
```

Header

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

Description

The class `simple_partitioner` models the default range splitting behavior of the `parallel_for`, `parallel_reduce` and `parallel_scan` algorithms.

3.3.1.1 simple_partitioner()

An empty default constructor.

3.3.1.2 simple_partitioner(simple_partitioner &partitioner, split)

An empty splitting constructor.

3.3.1.3 template<typename Range> bool should_execute_range (const Range &r, const task &t)

A function that returns true when the provided range should be executed to completion by the given task. It returns `!range.is_divisible()`.

3.3.2 auto_partitioner Class

Summary

A class that models an adaptive partitioner that monitors the work-stealing actions of the task_scheduler to manage the number of splits performed.

Syntax

```
class auto_partitioner;
```

Header

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

Description

The class auto_partitioner models an adaptive partitioner that limits the number of splits needed for load balancing by reacting to work-stealing events.

The range is first divided into S_I subranges, where S_I is proportional to the number of threads created by the task scheduler. These subranges are executed to completion by tasks unless they are stolen. If a subrange is stolen by an idle thread, the auto_partitioner further subdivides the range to create additional subranges.

The auto_partitioner creates additional subranges only if threads are actively stealing work. If the load is well balanced, the use of only a few large initial subranges reduces the overheads incurred when splitting and joining ranges. However, if there is a load imbalance that results in work-stealing, the auto_partitioner creates additional subranges that can be stolen to more finely balance the load.

The auto_partitioner therefore attempts to minimize the number of range splits, while providing ample opportunities for work-stealing.

3.3.2.1 auto_partitioner()

An empty default constructor.

3.3.2.2 auto_partitioner(auto_partitioner &partitioner, split)

A splitting constructor that divides the `auto_partitioner partitioner` into two partitioners.

3.3.2.3 template<typename Range> bool should_execute_range (const Range &r, const task &t)

A function that returns true when the provided range should be operated over as a whole by the given task's body. This function may return true even if `range.is_divisible() == true` and always returns true if `range.is_divisible() == false`. That is, this function may decide that `t` should process an `r` that can be further subdivided, but it always decides that `t` should process an `r` that cannot be further subdivided.

3.4 parallel_for<Range,Body> Template Function

Summary

Template function performs parallel iteration over a range of values.

Syntax

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

Header

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

Description

A `parallel_for<Range,Body>` represents parallel execution of `Body` over each value in `Range`. Type `Range` must model the Range concept (3.2). The body must model the requirements in Table 7.

Table 7: 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 0≤`i`<`n`..

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

using namespace tbb;

struct Average {
    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: The input must be padded such that input[-1] and input[n]
// can be used to calculate the first and last output values.
void ParallelAverage( float* output, float* input, size_t n ) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 0, 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.

3.4.1 Using the Partitioner Preview Feature

Summary

Template function performs parallel iteration over a range of values, with the splitting of the range guided by the Partitioner parameter.

Syntax

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

Header

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

Description

A `parallel_for<Range,Body,Partitioner>` represents parallel execution of `Body` over each value in `Range`. Type `Range` must model the Range concept (3.2). The body must model the requirements in Table 7. Type `Partitioner` must model the Partitioner concept (3.3).

Example

This example shows a simple use of the Partitioner concept with a `parallel_for`. The code shown below is an extension of the simple example presented in the previous subsection. An `auto_partitioner` is used to guide the splitting of the range.

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

using namespace tbb;

struct Average {
    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: The input must be padded such that input[-1] and input[n]
// can be used to calculate the first and last output values.
void ParallelAverage( float* output, float* input, size_t n ) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 0, n ), avg, auto_partitioner() );
}
```

Two important changes should be noted: (1) the call to `parallel_for` takes a third argument, an `auto_partitioner` object, and (2) the `blocked_range` constructor is not provided with a grainsize parameter.

In addition to the constructors described in Sections 3.2.1 and 3.2.2, the `blocked_range` and `blocked_range2d` template classes now define additional constructors that initialize all grainsize parameters to 1. In both of these classes, the grainsize is used to designate a size above which a range is considered to be divisible.

Table 8 provides guidance for selecting between the `simple_partitioner` and `auto_partitioner` classes.

Table 8: Guidance for Selecting a Partitioner

Partitioner Type	Discussion
`simple_partitioner`	Recursively splits a range until it is no longer divisible. The `Range::is_divisible` function is wholly responsible for deciding when recursive splitting halts. When used with classes such as `blocked_range` and `blocked_range2d`, the selection of an appropriate grainsize is therefore critical to allow concurrency while limiting overheads (see the discussion in Section 3.2.1).
`auto_partitioner`	Guides splitting decisions based on the work stealing behavior of the task scheduler. When used with classes such as `blocked_range` and `blocked_range2d`, the selection of an appropriate grainsize is less important. Subranges that are larger than the grain size are used unless load imbalances are detected. Therefore acceptable performance may often be achieved by simply using the default grain size of 1.

TIP: Ranges larger than grain size may be passed to the body when using an auto_partitioner. The body should not therefore use the value of grain size as an upper bound on the size of the range (for allocating temporary storage for example).

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, Body& body );
```

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 9.

Table 9: 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.

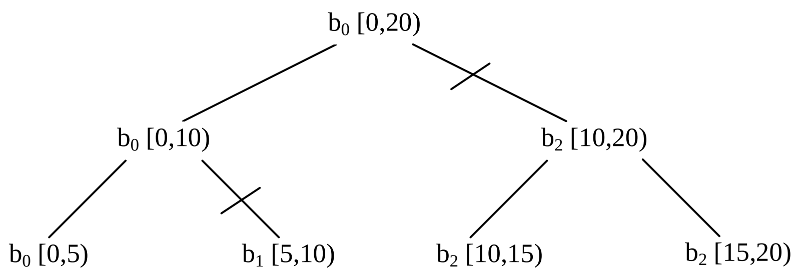

Figure 1: Example execution of parallel_reduce over blocked_range<int>(0,20,5)

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.

3.5.1 Using the Partitioner Preview Feature

Summary

Computes reduction over a range, with the splitting of the range guided by the Partitioner parameter.

Syntax

```
template<typename Range, typename Body, typename Partitioner>
        void parallel_reduce( const Range& range, Body& body,
                              Partitioner &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 9. Type `Partitioner` must model the Partitioner concept (3.3).

Example

The following code extends the example the previous section by using an `auto_partitioner`.

```
#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 ),
                     total, auto_partitioner() );
    return total.value;
}
```

Two important changes should be noted: (1) the call to `parallel_reduce` takes a third argument, an `auto_partitioner` object, and (2) the `blocked_range` constructor is not provided with a grainsize parameter. As discussed in Section 3.4.1, the `blocked_range` supports an additional constructor that sets the grainsize to 1 by default.

Table 8 provides guidance for selecting between the `simple_partitioner` and `auto_partitioner` classes.

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 );
```

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 ⊕ is addition, the parallel prefix corresponds a running sum. A serial implementation of parallel prefix is:

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

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

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

The template `parallel_scan<Range,Body>` implements parallel prefix generically. It requires the signatures described in Table 10.

Table 10: parallel_scan Requirements

Pseudo-Signature	Semantics
`void Body::operator()( const Range& r, pre_scan_tag )`	Preprocess iterations for range `r`.
`void Body::operator()( const Range& r, final_scan_tag )`	Do final processing for iterations of range `r`.
`Body::Body( Body& b, split )`	Split `b` so that `this` and `b` can accumulate separately.
`void Body::reverse_join( Body& a )`	Merge preprocessing state of `a` into `this`, where `a` was created earlier from `b` by `b`'s splitting constructor.
`void Body::assign( Body& b )`	Assign state of `b` to `this`.

The following code demonstrates how these signatures must be implemented to use `parallel_scan` in a way similar to the sequential example.

```
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.

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.

3.6.2 Using the Partitioner Preview Feature

Summary

Template function that computes parallel prefix, with the splitting of the range guided by the Partitioner parameter.

Syntax

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

Header

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

Description

A `parallel_scan<Range,Body,Partitioner>` computes a parallel prefix, also known as parallel scan (see Section 3.6 for a general description of parallel prefix).

Example

The following code extends the example the previous section by using an `auto_partitioner`.

```
using namespace tbb;

class Body {
    T sum;
    T* const y;
    const T* const x;
public:
    Body( T y_[], const T x_[] ) : sum(0), x(x_), y(y_) {}
    T get_sum() const {return sum;}

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

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

Two important changes should be noted: (1) the call to `parallel_scan` takes a third argument, an `auto_partitioner` object, and (2) the `blocked_range` constructor is not provided with a grainsize parameter. As discussed in Section 3.4.1, the `blocked_range` supports an additional constructor that sets the grainsize to 1 by default.

Table 8 provides guidance for selecting between the `simple_partitioner` and `auto_partitioner` classes.

3.7 parallel_while Template Class

Summary

Template class that processes work items.

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:

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

Table 11 shows the requirements on the stream and body.

Table 11: 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`.
`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` (4.2) models the requirements for S.

TIP: To achieve speedup, the grain size 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 );
    };
}
```

3.7.1 parallel_while<Body>()

Effects

Construct a `parallel_while` that is not yet running.

3.7.2 ~parallel_while<Body>()

Effects

Destroy a `parallel_while`.

3.7.3 Template <typename Stream> void run(Stream& stream, const Body& body)

Effects

Apply *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:

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

3.7.4 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

Add item to collection of items to be processed.

3.8 pipeline Class

Summary

Abstract base 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 is parallel or serial. 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 whether it is serial or not via the boolean 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.

A `filter` must be removed from the `pipeline` before destroying it. You can accomplish this by destroying the `pipeline` first, or calling `pipeline::clear()`.

Members

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

Remove all filters from the pipeline and destroy the pipeline

3.8.3 void add_filter(filter& f)

Effects

Append 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

Run 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.

3.8.5 void clear()

Effects

Remove 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). A filter is parallel or serial. A parallel filter can process multiple items in parallel and possibly out of order. A serial filter processes items one at a time in the original stream order. Parallel filters are preferred when practical because they permit parallel speedup. Whether the filter is serial or parallel is specified by an argument to the constructor.

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

Members

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

Example

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

3.8.6.1 filter(bool is_serial)

Effects

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

3.8.6.2 ~filter()

Effects

Destroys the filter. The filter must not be in a `pipeline`, otherwise memory might be corrupted. The debug version of the library raises an assertion failure if the filter is in a pipeline. Always clear or destroy the containing pipeline first. A way to remember this is that a pipeline acts like a container of Filters, and a C++ container usually does not allow destroying an item while it is in the container.

3.8.6.3 bool is_serial() const

Returns

True if filter is serial; false if filter is parallel.

3.8.6.4 virtual void* operator()(void * item)

Effects

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

Returns

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

3.9 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 12.

Table 12: Requirements on value type T of RandomAccessIterator for parallel_sort

Pseudo-Signature	Semantics
`void swap( T& x, T& y )`	Swaps `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.

Unlike STL, the Intel® Threading Building Blocks containers are not templated with respect to an `allocator` argument. The library retains control over memory allocation.

4.1 concurrent_hash_map<Key,T,HashCompare> Template Class

Summary

Template class for associative container with concurrent access.

Syntax

```
template<typename Key, typename T, typename HashCompare> 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. The interface resembles typical STL associative containers, but with some differences critical to supporting concurrent access.

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 13.

Table 13: 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 )`	Hashcode for key

CAUTION: As for most hash tables, if two keys are equal, they must hash to the same hash code. That is for a given HashCompare `h` and any two keys `j` and `k`, the following assertion must hold: "`!h.equal(j,k) || h.hash(j)==h.hash(k)`". The importance of this property is the reason that `concurrent_hash_map` makes key equality and hashing travel together in a single object instead of being separate objects.

Members

```
namespace tbb {
    template<typename Key, typename T, typename HashCompare>
    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;

        // whole-table operations
        concurrent_hash_map();
        concurrent_hash_map( const concurrent_hash_map& );
        ~concurrent_hash_map();
        concurrent_hash_map operator=( const concurrent_hash_map& );
        void clear();

        // 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 erase( const Key& key );

        // parallel iteration
        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;

        // 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;
    };
}
```

4.1.1 Whole Table Operations

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

4.1.1.1 concurrent_hash_map()

Effects

Construct empty table.

4.1.1.2 concurrent_hash_map(const concurrent_hash_map& table)

Effects

Copy a table. The table being copied may have map operations running on it concurrently.

4.1.1.3 ~concurrent_hash_map()

Effects

Remove all items from the table and destroy it. This method is not safe to execute concurrently with other methods on the same concurrent_hash_map.

4.1.1.4 concurrent_hash_map& operator= (concurrent_hash_map& source)

Effects

If source and destination (`this`) table are distinct, clear the destination table and copy all key-value pairs from the source table to the destination table. Otherwise, do nothing.

Returns

Reference to the destination table.

4.1.1.5 void clear()

Effects

Erase all key-value pairs from the table.

4.1.2 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 14: 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 – blocks access by other threads

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

4.1.2.1 const_accessor

Summary

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

Syntax

```
 template<typename Key, typename T, typename HashCompare> class
concurrent_hash_map<Key,T,HashCompare>::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>
    class concurrent_hash_map<Key,T,HashCompare>::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();
    };
}
```

4.1.2.1.1 bool empty() const

Returns

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

4.1.2.1.2 void release()

Effects

If `!empty()`, release the implied lock on the pair, and set instance to point to nothing. Otherwise do nothing.

4.1.2.1.3 const value_type& operator*() const

Effects

Raise assertion failure if empty() and TBB_DO_ASSERT (2.6.1) is defined as nonzero.

Returns

Const reference to key-value pair.

4.1.2.1.4 const value_type* operator->() const

Returns

```
&operator*()
```

4.1.2.1.5 const_accessor()

Effects

Construct const_accessor that points to nothing.

4.1.2.1.6 ~const_accessor

Effects

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

4.1.2.2 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>
class concurrent_hash_map<Key,T,HashCompare>::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>
    class concurrent_hash_map<Key,T,HashCompare>::accessor:
        concurrent_hash_map<Key,T,HashCompare>::const_accessor {
    public:
        typedef std::pair<const Key,T> value_type;
        value_type& operator*() const;
        value_type* operator->() const;
    };
}
```

4.1.2.2.1 value_type& operator*() const

Effects

Raise assertion failure if `empty()` and `TBB_DO_ASSERT` (2.6.1) is defined as nonzero.

Returns

Reference to key-value pair.

4.1.2.2.2 value_type* operator->() const

Returns

```
&operator*()
```

4.1.3 Concurrent Operations

The operations `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.

The result of the map operations is true if the operation succeeds.

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

Effects

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

Returns

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

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

Effects

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

Returns

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

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

Effects

Search table for pair with given key. If not present, insert new pair into table. The new pair is initialized with `pair(key,T())`. Set result to provide read-only access to the matching pair.

Returns

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

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

Effects

Search table for pair with given key. If not present, insert new pair into table. The new pair is initialized with `pair(key,T())`. Set result to provide write access to the matching pair.

Returns

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

4.1.3.5 bool erase(const Key& key)

Effects

Search table for pair with given key. Remove the matching pair if it exists.

Returns

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

4.1.4 Parallel Iteration

Types `const_range_type` and `range_type` model the Range concept (3.2) and provide methods to access the bounds of the range as shown in Table 15. 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.

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

Table 15: Concept for concurrent_hash_map Range R (In Addition to Table 4)

Pseudo-Signature	Semantics
`R::iterator R::begin() const`	First item in range
`R::iterator R::end() const`	One past last item in range

4.1.4.1 const_range_type range(size_t grainsize) const

Effects

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

Returns

`const_range_type` object for the table.

4.1.4.2 range_type range(size_t grainsize)

Returns

`range_type` object for the table.

4.1.5 Capacity

4.1.5.1 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.

4.1.5.2 bool empty() const

Returns

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

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

4.1.5.3 size_type max_size() const

Returns

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

4.1.6 Iterators

Template class `concurrent_hash_map` supports forward iterators; that is, iterators that can advance only forwards across the table. Reverse iterators are not supported.

4.1.6.1 iterator begin()

Returns

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

4.1.6.2 iterator end()

Returns

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

4.1.6.3 const_iterator begin() const

Returns

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

4.1.6.4 const_iterator end() const

Returns

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

4.2 concurrent_queue<T> 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 16: 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)`
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	waits 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>
    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() {}
        ~concurrent_queue();

        void push( const T& source );
        void pop( T& destination );
        bool pop_if_present( T& destination );
        size_type size() const {return internal_size();}
        bool empty() const;
        size_t capacity() const;
        void set_capacity( size_type capacity );

        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;
    };
}
```

4.2.1 concurrent_queue()

Effects

Construct empty queue.

4.2.2 ~concurrent_queue()

Effects

Destroy all items in the queue.

4.2.3 void push(const T& source)

Effects

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

4.2.4 void pop(T& destination)

Effects

Wait until a value becomes available and pop it from the queue. Assign it to `destination`. Destroy the original value.

4.2.5 bool pop_if_present(T& destination)

Effects

If value is available, pop it from the queue, assign it to destination, and destroy the original value. Otherwise do nothing.

Returns

True if value was popped; false otherwise.

4.2.6 size_type size() const

Returns

Number pushes minus number of pops. The result is negative if there are pop operations waiting for corresponding pushes.

4.2.7 bool empty() const

Returns

`size()==0`

4.2.8 size_type capacity() const

Returns

Maximum number of values that the queue can hold.

4.2.9 void set_capacity(size_type capacity)

Effects

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

4.2.10 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;

}
```

4.2.10.1 iterator begin()

Returns

`iterator` pointing to beginning of the queue.

4.2.10.2 iterator end()

Returns

`iterator` pointing to end of the queue.

4.2.10.3 const_iterator begin() const

Returns

`const_iterator` with pointing to beginning of the queue.

4.2.10.4 const_iterator end() const

Returns

`const_iterator` pointing to end of the queue.

4.3 concurrent_vector

Summary

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

Syntax

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

Header

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

Description

A `concurrent_vector` is a dynamically growable array for which it is safe to simultaneously access elements in the vector while growing it. The index of the first element is 0.

Members

```
namespace tbb {
    template<typename T>
    class concurrent_vector {
    public:
        typedef size_t size_type;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T& reference;
        typedef const T& const_reference;

        // whole vector operations
        concurrent_vector() {}
        concurrent_vector( const concurrent_vector& );
        concurrent_vector& operator=( const concurrent_vector&);
        ~concurrent_vector();
        void clear();

        // concurrent operations
        size_type grow_by( size_type delta );
        void grow_to_at_least( size_type new_size );
        size_type push_back( const_reference value );
        reference operator[]( size_type index );
        const_reference operator[]( size_type index ) const;

        // parallel iteration
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef generic_range_type<iterator> range_type;
        typedef generic_range_type<const_iterator> const_range_type;

        range_type range( size_t grainsize );
        const_range_type range( size_t grainsize ) const;

        // capacity
        size_type size() const;
        bool empty() const;
        size_type capacity() const;
        void reserve( size_type n );
        size_type max_size() const;

        // STL support
        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;

        typedef implementation-defined reverse_iterator;
        typedef implementation-defined const_reverse_iterator;
        iterator rbegin();
        iterator rend();
        const_iterator rbegin() const;
        const_iterator rend() const;
    };
}
```

4.3.1 Whole Vector Operations

These operations are *not* thread safe on the same instance.

4.3.1.1 concurrent_vector()

Effects

Construct empty vector.

4.3.1.2 concurrent_vector(const concurrent_vector& src)

Effects

Construct copy of *src.*

4.3.1.3 concurrent_vector& operator=(const concurrent_vector& src)

Effects

Assign contents of *src* to *this.

Returns

Reference to left hand side.

4.3.1.4 ~concurrent_vector()

Effects

Erase all elements and destroy the vector.

4.3.1.5 void clear()

Effects

Erase all elements. Afterwards, `size()==0`.

4.3.2 Concurrent Operations

The methods described in this section safely execute on the same instance of a `concurrent_vector<T>`.

4.3.2.1 size_type grow_by(size_type delta)

Effects

Atomically append *delta* elements to the end of the vector. The new elements are initialized with T(), where T is the `value_type` of the vector.

Returns

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

4.3.2.2 void grow_to_at_least(size_type n)

Effects

Grow the vector until it has at least *n* elements. The new elements are initialized with `T()`, where `T` is the `value_type` of the vector.

4.3.2.3 size_t push_back(const_reference value);

Effects

Atomically append copy of *value* to the end of the vector.

Returns

Index of the copy.

4.3.2.4 reference operator[](size_type index)

Returns

Reference to element with the specified index.

4.3.2.5 const_reference operator[](size_type index) const;

Returns

Const reference to element with the specified index.

4.3.3 Parallel Iteration

Types `const_range_type` and `range_type` model the Range concept (3.2) and provide methods to access the bounds of the range as shown in Table 15. 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.

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

Table 17: Concept for concurrent_vector Range R

Pseudo-Signature	Semantics
`R::iterator R::begin() const`	First item in range
`R::iterator R::end() const`	One past last item in range

4.3.3.1 range_type range(size_t grainsize)

Returns

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

4.3.3.2 const_range_type range(size_t grainsize) const

Returns

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

4.3.4 Capacity

4.3.4.1 size_type size() const

Returns

Number of elements in the vector. The result may include elements that are under construction by concurrent calls to methods `grow_by` (4.3.2.1) or `grow_to_at_least` (4.3.2.2).

4.3.4.2 bool empty() const

Returns

`size()==0.`

4.3.4.3 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 has to allocate more memory.

4.3.4.4 void reserve(size_type n)

Returns

Reserve space for at least *n* elements.

Throws

`std::length_error` if `n>max_size().`

4.3.4.5 size_type max_size() const

Returns

Highest size vector that might be representable.

4.3.5 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.

4.3.5.1 iterator begin()

Returns

`iterator` pointing to beginning of the vector.

4.3.5.2 iterator end()

Returns

`iterator` pointing to end of the vector.

4.3.5.3 const_iterator begin() const

Returns

`const_iterator` with pointing to beginning of the vector.

4.3.5.4 const_iterator end() const

Returns

`const_iterator` pointing to end of the vector.

4.3.5.5 iterator rbegin()

Returns

```
const_reverse_iterator(end())
```

4.3.5.6 iterator rend()

Returns

```
const_reverse_iterator(begin())
```

4.3.5.7 const_reverse_iterator rbegin() const

Returns

```
const_reverse_iterator(end())
```

4.3.5.8 const_ reverse_iterator rend() const

Returns

```
const_reverse_iterator(begin())
```

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

Table 18: 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, void* hint=0 )`	Allocate space for `n` values
`void A::deallocate( T* p, size_t  n )`	Deallocate n values

`size_type A::max_size() const throw()`	Maximum plausible 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 `scalable_allocator` (5.2) and `cached_aligned_allocator` (5.3) model the Allocator concept.

5.2 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 18. 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`.

Members

See Allocator concept (5.1).

Acknowledgement

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

5.2.1 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.

5.3 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 18. 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 NFS_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, 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`.

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

Effects

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

Returns

Pointer to the allocated memory.

5.3.2 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

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

5.3.3 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.

5.4 aligned_space Template Class

Summary

Uninitialized memory space.

Syntax

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

Header

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

Description

An `aligned_space` occupies enough memory 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();
    };
}
```

5.4.1 aligned_space()

Effects

None. Does not invoke constructors.

5.4.2 ~aligned_space()

Effects

None. Does not invoke destructors.

5.4.3 T* begin()

Returns

Pointer to beginning of storage.

5.4.4 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 19 shows the requirements for the Mutex concept for a mutex type M

Table 19: Mutex Concept

Pseudo-Signature	Semantics
`M()`	Construct unlocked mutex
`~M()`	Destroy unlocked mutex.
`typename M::scoped_lock`	Corresponding scoped-lock type
`M::scoped_lock()`	Construct lock without acquiring mutex
`M::scoped_lock(M&)`	Construct lock and acquire lock on mutex
`M::~scoped_lock()`	Release lock (if acquired)
`M::scoped_lock::acquire(M&)`	Acquire lock on mutex
`bool M::scoped_lock::try_acquire(M&)`	Try to acquire lock on mutex. Return true if lock acquired, false otherwise.
`M::scoped_lock::release()`	Release lock

Table 20 summarizes the classes that model the Mutex concept.

Table 20: Mutexes that model the Mutex concept

	Scalable	Fair	Reentrant	Sleeps	Size
`mutex`	OS dependent	OS dependent	no	yes	≥ 3 words
`spin_mutex`	no	no	no	no	1 byte
`queuing_mutex`	✓	✓	no	no	1 word
`spin_rw_mutex`	no	no	no	no	1 word
`queuing_rw_mutex`	✓	✓	no	no	1 word

See the tutorial for a discussion of the mutex properties.

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 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 reentrant. 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).

6.1.4 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 reentrant.

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).

6.1.5 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 21 shows the requirements for ReaderWriterMutex `RW`.

Table 21: 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

The following subsections explain the semantics of the ReaderWriterMutex concept in detail.

Model Types

`spin_rw_mutex` (6.1.6) and `queuing_rw_mutex` (6.1.7) model the ReaderWriterMutex concept.

6.1.5.1 ReaderWriterMutex()

Effect

Construct unlocked `ReaderWriterMutex`.

6.1.5.2 ~ReaderWriterMutex()

Effect

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

6.1.5.3 ReaderWriterMutex::scoped_lock()

Effect

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

6.1.5.4 ReaderWriterMutex::scoped_lock(ReaderWriterMutex& rw, bool write =true)

Effect

Construct 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.

6.1.5.5 ReaderWriterMutex::~scoped_lock()

Effect

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

6.1.5.6 void ReaderWriterMutex:: scoped_lock:: acquire(ReaderWriterMutex& rw, bool write=true)

Effect

Acquires a lock on mutex *rw*. The lock is a writer lock if *write* is true; a reader lock otherwise.

6.1.5.7 bool ReaderWriterMutex:: scoped_lock::try_acquire(ReaderWriterMutex& rw, bool write=true)

Effect

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.

6.1.5.8 void ReaderWriterMutex:: scoped_lock::release()

Effect

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

6.1.5.9 bool ReaderWriterMutex:: scoped_lock::upgrade_to_writer()

Effect

Change 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 and reacquired; `true` otherwise.

6.1.5.10 bool ReaderWriterMutex:: scoped_lock::downgrade_to_reader()

Effect

Change 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`.

6.1.6 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 (6.1.1). A `spin_rw_mutex` is not scalable, fair, or reentrant. 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 (6.1.5).

6.1.7 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 (6.1.1). 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 reentrant.

Members

See ReaderWriterMutex concept (6.1.5).

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 specialization `atomic<void*>` does not allow pointer 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 22 shows the fencing for the non-template form.

Table 22: Memory Fences Implied by Non-Template Methods

Kind	Description	Default For
acquire	Operations after the fence never move over it.	read
release	Operations before the fence never move over it.	write
full	Operations on either side never move over it.	`fetch_and_store`, `fetch_and_add`, `compare_and_swap`

TIP: Template class `atomic<T>` does not have any non-trivial constructors, because such constructors could lead to accidental introduction of compiler temporaries that would subvert the purpose of `atomic<T>`. 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 );

        value_type operator+=(value_type);
        value_type operator-=(value_type);
        value_type operator++();
        value_type operator++(int);
        value_type operator--();
        value_type operator--(int);
    };
}
```

6.2.1 enum memory_semantics

Description

Defines values used to select the template variants that permit more selective memory fencing (see Table 22).

6.2.2 value_type fetch_and_add(value_type addend)

Effect

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()

Effect

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()

Effect

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 )
```

6.2.6 Effect

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*.

6.2.7 value_type fetch_and_store(value_type new_value)

Effect

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 time or duration.

Members

```
namespace tbb {

    class tick_count::interval_t {
    public:
        interval_t();
        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 );

} // tbb
```

7.1.3.1 interval_t()

Effects

Construct `interval_t` representing zero time duration.

7.1.3.2 double seconds() const

Returns

Time interval measured in seconds.

7.1.3.3 interval_t operator+=(const interval_t& i)

Effects

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

Returns

Reference to `*this`.

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, const interval_t& j)

Returns

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

7.1.3.6 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 (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 logical units 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 2 shows the steps. The running tasks at each step are shaded.

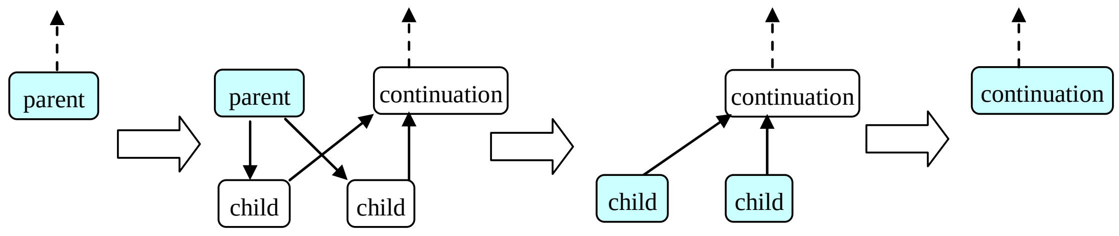

Figure 2: Continuation-passing style

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 3.

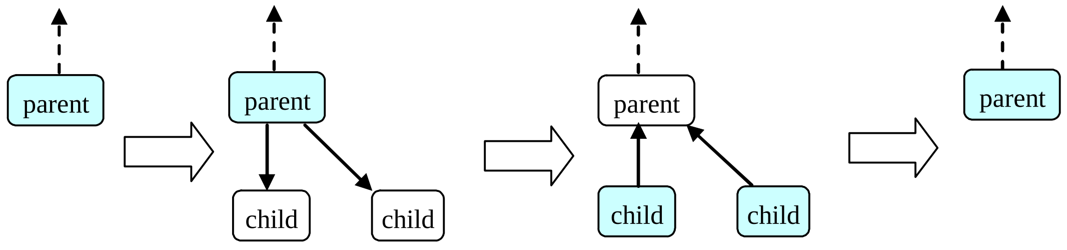

Figure 3: Blocking style

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 task 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 pulls tasks from the deepest non-empty list in the array. If there are no non-empty lists, the thread randomly steals a task from the shallowest list of another thread. A thread also implicitly steals if it completes the last child, in which case it starts executing the task that was waiting on the children.

The task scheduler tends to strike a good balance between locality of reference, space efficiency, and parallelism. The scheduling technique is similar to that used by Cilk (Blumofe 1995).

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 (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 {

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

8.2.1 task_scheduler_init(int number_of_threads=automatic)

Requirements

The value `number_of_threads` shall be one of the values in Table 23.

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 23. These workers sleep until needed by the task scheduler.

Table 23: Values for number_of_threads

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

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 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. Programmers are expected to derive classes from `task`, and 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 24.

Table 24: 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 4.

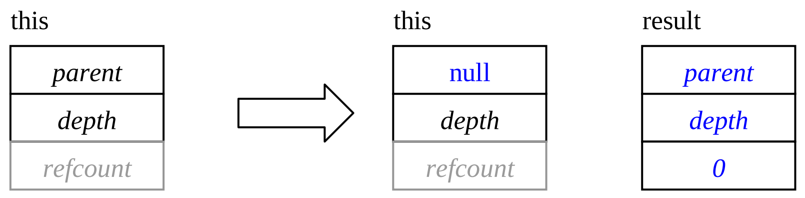

Figure 4: Example Effect Diagram

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& 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;

        // 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.

The derived class should override the virtual destructor if necessary to release resources allocated by the constructor.

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. Invoke `execute()` and wait 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 decrement *parent->refcount*, and if becomes zero, put the *parent* into the ready pool.
   b. Call the task's destructor
   c. Free the memory for task for reuse.
3. If the task has been marked for recycling:
   a. If marked by `recycle_to_reexecute`, put 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()) T

Allocates a `task` of type *T* with a depth of one more than the depth of the innermost `task` currently being executed by the current native thread. Figure 5 summarizes the state transition.

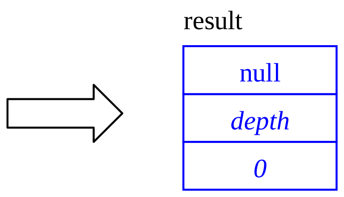

Figure 5: Effect of task::allocate_root()

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

8.3.2.2 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 6 summarizes the state transition.

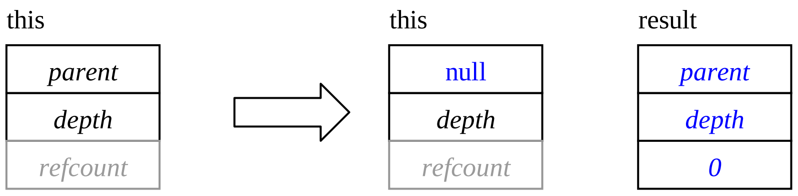

Figure 6: Effect of allocate_continuation()

8.3.2.3 new(this. allocate_child()) T

Effect

Allocates a `task` with a depth one more than this, with this as its *parent*. Figure 7 summarizes the state transition.

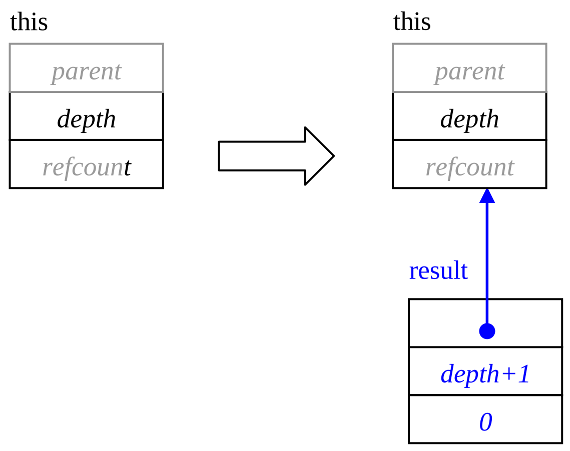

Figure 7: Effect of allocate_child()

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` `(530H8.3.6.3)`. 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.

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

Effect

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 8 summarizes the state transition.

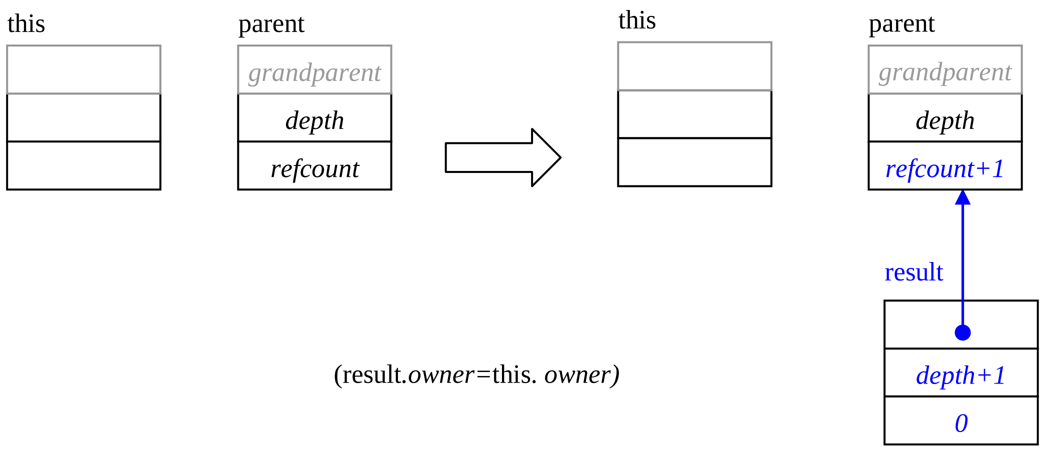

Figure 8: Effect of allocate_additional_child_of(parent)

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 9 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`.

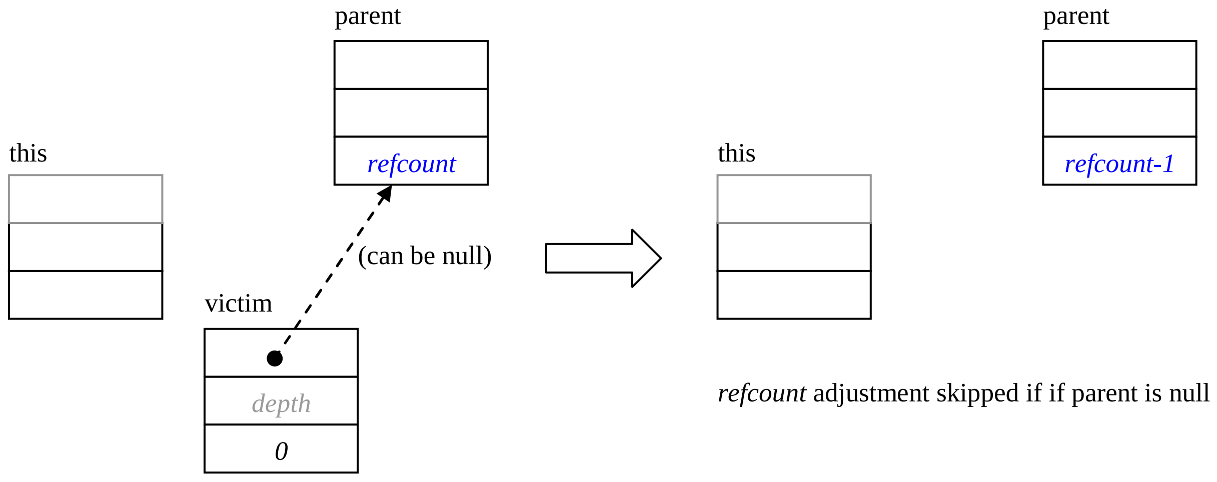

Figure 9: Effect of destroy(victim)

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 Preview Feature: 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:

1. The method `execute()` recycles `this` as a continuation.
2. The continuation creates children.
3. 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& parent)

Requirements

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

Effects

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

8.3.4.4 void recycle _to_reexecute()

Requirements

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

Set the depth attribute of the task to *new_depth.* Figure 10 shows the update.

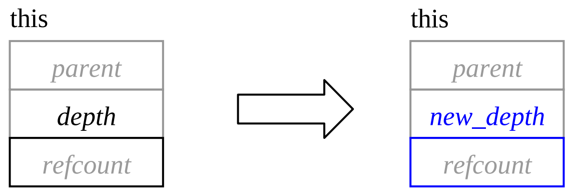

Figure 10: Effect of set_depth

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

Set the depth attribute of the task to depth+*delta.* Figure 11 illustrates the effect. The update is not atomic.

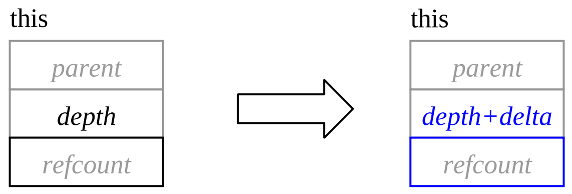

Figure 11: Effect of add_to_depth(delta)

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:

1. Spawn a single `task`
2. 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 12 summarizes the state transitions.

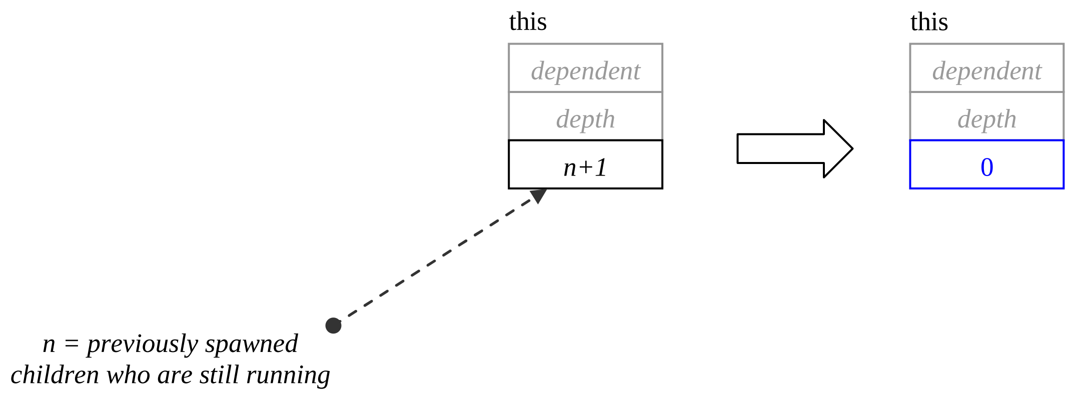

Figure 12: Effect of wait_for_all

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 13 illustrates the state transitions.

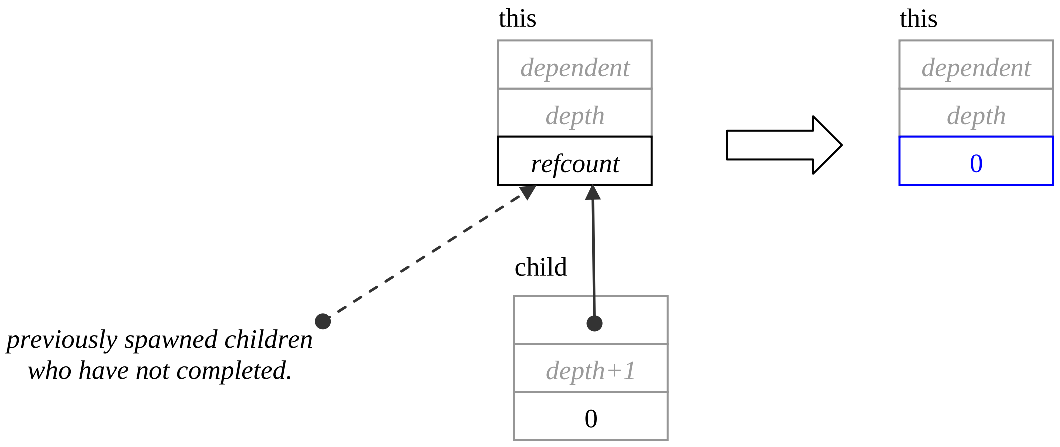

Figure 13: Effect of spawn_and_wait_for_all

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 executing.

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 task Debugging

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

8.3.8.1 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 25 describes valid states. Any other value is the result of memory corruption, such as using a `task` whose memory has been deallocated.

Table 25: 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 14 summarizes possible state transitions for a `task`.

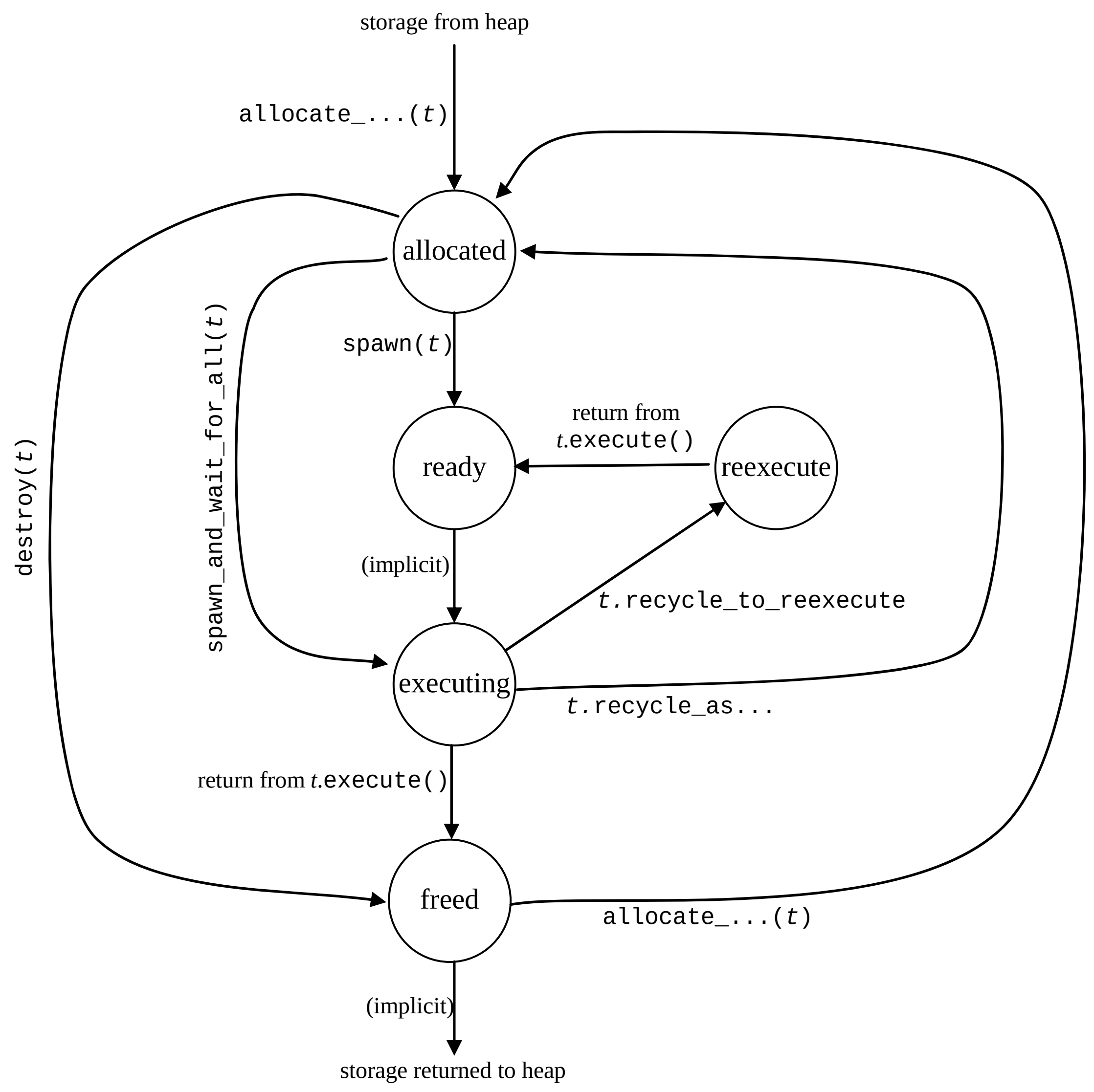

Figure 14: Typical task::state() transitions

8.3.8.2 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 Catalog of Recommended task Recurrence Patterns

This section catalogues three recommended task recurrence patterns. In each pattern, class `T` is assumed to derive from class `task`. Subtasks are labeled `t1, t2, ... tk`. 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.

8.6.1 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(...);  t_k.spawn();
        task& t_k-1= new( allocate_child() ) T(...);  t_k-1.spawn();
        ...
        task& t_1 = new( allocate_child() ) T(...);   t_1.spawn_and_wait(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`.

8.6.2 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.

8.6.2.1 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(...); t_k.spawn();
        task& t_{k-1} = new( allocate_child() ) T(...); t_{k-1}.spawn();
        ...
        task& t_1 = new( c.allocate_child() ) T(...);  t_1.spawn();
        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 8.6.1.
- 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.

8.6.2.2 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(...); t_k.spawn();
        task& t_{k-1} = new( c.allocate_child() ) T(...); t_{k-1}.spawn();
        ...
        task& t2 = new( c.allocate_child() ) T(...);  t2.spawn();
        // task t1 is our recycled self.
        recycle_as_child_of(c);
        ... update fields of *this to state subproblem to be solved by t1
        return this;
    }
}
```

The key points of the pattern are:

- The call to `set_ref_count` uses *k* as its argument. There is no extra 1 as there is in blocking style discussed in Section 8.6.1.
- 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.

9 References

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.

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.