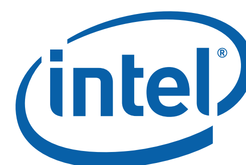


インテル® スレッディング・ビルディング・ブロック

リファレンス・マニュアル

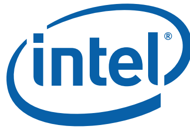
© 2007 Intel Corporation.

無断での引用、転載を禁じます。

資料番号 315415-001J

改訂番号: 1.6

Web サイト: http://www.intel.com



著作権と商標について

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスを許諾するためのものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証（特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む）にも一切応じないものとします。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。
インテル製品は、予告なく仕様や説明が変更される場合があります。

機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。

MPEG は、ビデオの圧縮/伸張に関する国際的な規格であり、ISO によって奨励されています。MPEG コーデックまたは MPEG 対応のプラットフォームを実装するには、Intel Corporation をはじめとする各種の団体からライセンスを取得しなければならない場合があります。

本資料で説明されているソフトウェアには、不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのソフトウェアの不具合については、インテルまでお問い合わせください。

本資料およびこれに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、その使用および複製はライセンス契約で定められた条件下でのみ許可されます。本資料で提供される情報は、情報供与のみを目的としたものであり、予告なく変更されることがあります。また、本資料で提供される情報は、インテルによる確約と解釈されるべきものではありません。インテルは本資料の内容およびこれに関連して提供されるソフトウェアにエラー、誤り、不正確な点が含まれていたとしても一切責任を負わないものとします。

ライセンス契約で許可されている場合を除き、インテルからの文書による承諾なく、本資料のいかなる部分も複製したり、検索システムに保持したり、他の形式や媒体によって転送したりすることは禁じられています。

機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。

Intel、インテル、Intel ロゴ、Itanium は、アメリカ合衆国およびその他の国における Intel Corporation の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

© 2007 Intel Corporation.

改訂履歴

文書番号	改訂番号	説明	改訂日付
315415-001	1.5	Partitioner コンセプトを `parallel_for`、`parallel_reduce` および `parallel_scan` のプレビュー機能として追加。`recycle_as_safe_continuation` メソッドを追加。"継続渡しスタイル" 図の誤りを訂正。	2007 年 3 月 1 日
	1.6	誤字を修正。	2007 年 4 月 11 日

目次

1 概要 1

2 一般的な規則 2

2.1 表記規則 2

2.2 用語 3

2.2.1 コンセプト 3

2.2.2 モデル 4

2.2.3 CopyConstructible 4

2.3 識別子 4

2.3.1 大文字と小文字 4

2.3.2 予約済み識別子プリフィックス 5

2.4 名前空間 5

2.4.1 tbb 名前空間 5

2.4.2 tbb::internal 名前空間 5

2.5 スレッド安全性 5

2.6 デバッグ機能の有効化 6

2.6.1 TBB_DO_ASSERT マクロ 6

2.6.2 TBB_DO_THREADING_TOOLS マクロ 6

3 アルゴリズム 7

3.1 分割可能なコンセプト 7

3.1.1 split クラス 8

3.2 Range コンセプト 8

3.2.1 blocked_range<Value> テンプレート・クラス 10

3.2.1.1 size_type 12

3.2.1.2 blocked_range(Value begin, Value end, size_t grainsize=1) 13

3.2.1.3 blocked_range(blocked_range& range, split) 13

3.2.1.4 size_type size() const 14

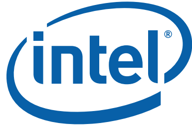
3.2.1.5 bool empty() const 14

3.2.1.6 size_type grainsize() const 14

3.2.1.7 bool is_divisible() const 14

3.2.1.8 const_iterator begin() const 15

3.2.1.9 const_iterator end() const 15



3.2.2 blocked_range2d テンプレート・クラス 15

3.2.2.1 row_range_type 17
3.2.2.2 col_range_type 17
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) 18
3.2.2.4 blocked_range2d<RowValue,ColValue> (blocked_range2d& range, split) 18
3.2.2.5 bool empty() const 18
3.2.2.6 bool is_divisible() const 19
3.2.2.7 const row_range_type& rows() const 19
3.2.2.8 const col_range_type& cols() const 19

3.3 Partitioner コンセプト 19

3.3.1 simple_partitioner クラス 21

3.3.1.1 simple_partitioner() 21
3.3.1.2 simple_partitioner(simple_partitioner &partitioner, split) 21
3.3.1.3 template<typename Range> bool should_execute_range (const Range &r, const task &t) 21

3.3.2 auto_partitioner クラス 22

3.3.2.1 auto_partitioner() 22
3.3.2.2 auto_partitioner(auto_partitioner &partitioner, split) 22
3.3.2.3 template<typename Range> bool should_execute_range (const Range &r, const task &t) 23

3.4 parallel_for<Range,Body> テンプレート関数 23

3.4.1 Partitioner 機能の使用 27

3.5 parallel_reduce<Range,Body> テンプレート関数 29

3.5.1 Partitioner 機能の使用 32

3.6 parallel_scan<Range,Body> テンプレート 関数 34

3.6.1 pre_scan_tag and final_scan_tag クラス 36

3.6.1.1 bool is_final_scan() 37

3.6.2 Partitioner 機能の使用 37

3.7 parallel_while テンプレート・クラス 39

3.7.1 parallel_while<Body>() 40
3.7.2 ~parallel_while<Body>() 40
3.7.3 テンプレート <typename Stream> void run(Stream& stream, const Body& body) 41
3.7.4 void add(const value_type& item) 41

3.8 pipeline クラス 41

3.8.1 pipeline() 43
3.8.2 ~pipeline() 43
3.8.3 void add_filter(filter& f) 43
3.8.4 void run(size_t max_number_of_live_tokens) 43
3.8.5 void clear() 44

3.8.6 filter クラス 44

3.8.6.1 filter(bool is_serial) 45
3.8.6.2 ~filter() 45
3.8.6.3 bool is_serial() const 45
3.8.6.4 virtual void* operator()(void * item) 45

3.9 parallel_sort<RandomAccessIterator, Compare> テンプレート関数 46

4 コンテナー 48

4.1 concurrent_hash_map<Key,T, HashCompare> テンプレート・クラス 48

4.1.1 テーブル全体の操作 50

4.1.1.1 concurrent_hash_map() 50
4.1.1.2 concurrent_hash_map(const concurrent_hash_map& table) 50
4.1.1.3 ~concurrent_hash_map() 50
4.1.1.4 concurrent_hash_map& operator= (concurrent_hash_map& source) 51
4.1.1.5 void clear() 51

4.1.2 同時アクセス 51

4.1.2.1 const_accessor 52
4.1.2.2 accessor 54

4.1.3 並列操作 55

4.1.3.1 bool find(const_accessor& result, const Key& key) const 55
4.1.3.2 bool find(accessor& result, const Key& key) 56
4.1.3.3 bool insert(const_accessor& result, const Key& key) 56
4.1.3.4 bool insert(accessor& result, const Key& key) 56
4.1.3.5 bool erase(const Key& key) 57

4.1.4 並列反復 57

4.1.4.1 const_range_type range(size_t grainsize) const 57
4.1.4.2 range_type range(size_t grainsize) 58

4.1.5 容量 58

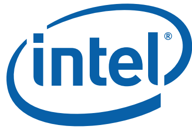
4.1.5.1 size_type size() const 58
4.1.5.2 bool empty() const 58
4.1.5.3 size_type max_size() const 58

4.1.6 イテレーター 58

4.1.6.1 iterator begin() 58
4.1.6.2 iterator end() 59
4.1.6.3 const_iterator begin() const 59
4.1.6.4 const_iterator end() const 59

4.2 concurrent_queue<T> テンプレート・ クラス 59

4.2.1 concurrent_queue() 61
4.2.2 ~concurrent_queue() 61
4.2.3 void push(const T& source) 61
4.2.4 void pop(T& destination) 62
4.2.5 bool pop_if_present(T& destination) 62
4.2.6 size_type size() const 62
4.2.7 bool empty() const 62
4.2.8 size_type capacity() const 62
4.2.9 void set_capacity(size_type capacity) 63



4.2.10 イテレーター 63
4.2.10.1 iterator begin() 63
4.2.10.2 iterator end() 64
4.2.10.3 const_iterator begin() const 64
4.2.10.4 const_iterator end() const 64
4.3 concurrent_vector 64
4.3.1 ベクトル全体の操作 66
4.3.1.1 concurrent_vector() 66
4.3.1.2 concurrent_vector(const concurrent_vector& src) 66
4.3.1.3 concurrent_vector& operator=(const concurrent_vector& src) 66
4.3.1.4 ~concurrent_vector() 66
4.3.1.5 void clear() 66
4.3.2 並列操作 67
4.3.2.1 size_type grow_by(size_type delta) 67
4.3.2.2 void grow_to_at_least(size_type n) 67
4.3.2.3 size_t push_back(const_reference value); 67
4.3.2.4 reference operator[](size_type index) 67
4.3.2.5 const_reference operator[](size_type index) const; 68
4.3.3 並列反復 68
4.3.3.1 range_type range(size_t grainsize) 68
4.3.3.2 const_range_type range(size_t grainsize) const 68
4.3.4 容量 69
4.3.4.1 size_type size() const 69
4.3.4.2 bool empty() const 69
4.3.4.3 size_type capacity() const 69
4.3.4.4 void reserve(size_type n) 69
4.3.4.5 size_type max_size() const 69
4.3.5 イテレーター 70
4.3.5.1 iterator begin() 70
4.3.5.2 iterator end() 70
4.3.5.3 const_iterator begin() const 70
4.3.5.4 const_iterator end() const 70
4.3.5.5 iterator rbegin() 70
4.3.5.6 iterator rend() 70
4.3.5.7 const_reverse_iterator rbegin() const 71
4.3.5.8 const_ reverse_iterator rend() const 71
5 メモリー割り当て 72
5.1 Allocator コンセプト 72
5.2 scalable_allocator<T> テンプレート・ クラス 73
5.2.1 スケーラブル・アロケーターの C インターフェイス 74
5.3 cache_aligned_allocator<T> テンプレート・クラス 75
5.3.1 pointer allocate(size_type n, void* hint=0) 77
5.3.2 void deallocate(pointer p, size_type n) 77
5.3.3 char* _Charalloc(size_type size) 77
5.4 aligned_space テンプレート・クラス 78

5.4.1 aligned_space() ... 78
5.4.2 ~aligned_space() ... 79
5.4.3 T* begin() ... 79
5.4.4 T* end() ... 79

6 同期 ... 80

6.1 Mutex ... 80

6.1.1 Mutex コンセプト ... 80

6.1.2 mutex クラス ... 81

6.1.3 spin_mutex クラス ... 82

6.1.4 queuing_mutex クラス ... 83

6.1.5 ReaderWriterMutex コンセプト ... 83

6.1.5.1 ReaderWriterMutex() ... 85
6.1.5.2 ~ReaderWriterMutex() ... 85
6.1.5.3 ReaderWriterMutex::scoped_lock() ... 85
6.1.5.4 ReaderWriterMutex::scoped_lock(ReaderWriterMutex& rw, bool write =true) ... 85
6.1.5.5 ReaderWriterMutex::~scoped_lock() ... 85
6.1.5.6 void ReaderWriterMutex:: scoped_lock:: acquire(ReaderWriterMutex& rw, bool write=true) ... 85
6.1.5.7 bool ReaderWriterMutex:: scoped_lock::try_acquire(ReaderWriterMutex& rw, bool write=true)86
6.1.5.8 void ReaderWriterMutex:: scoped_lock::release() ... 86
6.1.5.9 bool ReaderWriterMutex:: scoped_lock::upgrade_to_writer() ... 86
6.1.5.10 bool ReaderWriterMutex:: scoped_lock::downgrade_to_reader() ... 86

6.1.6 spin_rw_mutex クラス ... 87

6.1.7 queuing_rw_mutex クラス ... 87

6.2 atomic<T> テンプレート・クラス ... 88

6.2.1 enum memory_semantics ... 90
6.2.2 value_type fetch_and_add(value_type addend) ... 90
6.2.3 value_type fetch_and_increment() ... 91
6.2.4 value_type fetch_and_decrement() ... 91
6.2.5 value_type compare_and_swap ... 91
6.2.6 value_type fetch_and_store(value_type new_value) ... 91

7 時間計測 ... 93

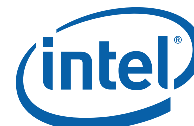
7.1 tick_count クラス ... 93

7.1.1 static tick_count tick_count::now() ... 94
7.1.2 tick_count::interval_t operator−(const tick_count& t1, const tick_count& t0) ... 94

7.1.3 tick_count::interval_t クラス ... 94

7.1.3.1 interval_t() ... 95
7.1.3.2 double seconds() const ... 95
7.1.3.3 interval_t operator+=(const interval_t& i) ... 95
7.1.3.4 interval_t operator−=(const interval_t& i) ... 96
7.1.3.5 interval_t operator+ (const interval_t& i, const interval_t& j) ... 96
7.1.3.6 interval_t operator− (const interval_t& i, const interval_t& j) ... 96

8 タスク・スケジューリング ...97
8.1 スケジューリング・アルゴリズム ...98
8.2 task_scheduler_init クラス ...99
8.2.1 task_scheduler_init(int number_of_threads=automatic) ...101
8.2.2 ~task_scheduler_init() ...101
8.2.3 void initialize(int number_of_threads=automatic) ...102
8.2.4 void terminate() ...102
8.2.5 OpenMP との混在使用 ...102
8.3 task クラス ...103
8.3.1 タスクの派生 ...106
8.3.1.1 execute() の処理 ...106
8.3.2 タスクの割り当て ...107
8.3.2.1 new(task::allocate_root()) T ...107
8.3.2.2 new(this. allocate_continuation()) T ...108
8.3.2.3 new(this. allocate_child()) T ...108
8.3.2.4 new(this.task::allocate_additional_child_of(parent)) ...109
8.3.3 明示的なタスクの派生 ...110
8.3.3.1 void destroy(task& victim) ...110
8.3.4 タスクの再利用 ...111
8.3.4.1 void recycle_as_continuation() ...111
8.3.4.2 void recycle_as_safe_continuation() ...112
8.3.4.3 void recycle_as_child_of(task& parent) ...112
8.3.4.4 void recycle _to_reexecute() ...113
8.3.5 タスクの深さ ...113
8.3.5.1 depth_type ...113
8.3.5.2 depth_type depth() const ...113
8.3.5.3 void set_depth(depth_type new_depth) ...113
8.3.5.4 void add_to_depth(int delta) ...114
8.3.6 同期 ...114
8.3.6.1 void set_ref_count(int count) ...115
8.3.6.2 void wait_for_all() ...115
8.3.6.3 void spawn(task& child) ...116
8.3.6.4 void spawn (task_list& list) ...116
8.3.6.5 void spawn_and_wait_for_all(task& child) ...117
8.3.6.6 void spawn_and_wait_for_all(task_list& list) ...117
8.3.6.7 static void spawn_root_and_wait(task& root) ...118
8.3.6.8 static void spawn_root_and_wait(task_list& root_list) ...118
8.3.7 タスクのコンテキスト ...118
8.3.7.1 static task& self() ...118
8.3.7.2 task* parent() const ...119
8.3.7.3 bool is_stolen_task() const ...119
8.3.8 タスクのデバッグ ...119
8.3.8.1 state_type state() const ...119
8.3.8.2 int ref_count() const ...121



8.4 empty_task クラス ... 122
8.5 task_list クラス ... 122
8.5.1 task_list() ... 123
8.5.2 ~task_list() ... 123
8.5.3 bool empty() const ... 124
8.5.4 push_back(task& task) ... 124
8.5.5 task& task pop_front() ... 124
8.5.6 void clear() ... 124
8.6 推奨するタスク回帰パターンのカタログ ... 124
8.6.1 ブロックスタイルと *k* の子 ... 125
8.6.2 継続渡しスタイルと *k* の子 ... 125
8.6.2.1 継続としての親の再利用 ... 125
8.6.2.2 子としての親の再利用 ... 126
9 参考文献 ... 127

1 概要

インテル® スレッディング・ビルディング・ブロックは、標準 ISO C++ コードを使用したスケーラブルな並列プログラミングをサポートするライブラリーです。特別な言語やコンパイラーは不要で、スケーラブルなデータ並列プログラミングを促進するように設計されています。また、入れ子の並列処理を完全にサポートするので、より小さな並列コンポーネントからより大きな並列コンポーネントを構築することができます。ライブラリーを使用する場合は、スレッドではなく、タスクを指定し、ライブラリーが効率的な方法でスレッド上にタスクをマップするようにします。

ライブラリー・インターフェイスの多くは、インターフェイスが特定の型ではなく型の要件によって定義される、汎用プログラミングを使用します。C++ 標準テンプレート・ライブラリー (STL) は汎用プログラミングの例です。汎用プログラミングを使用することで、インテル® スレッディング・ビルディング・ブロックは、柔軟かつ効率的なプログラミングが可能になっています。汎用インターフェイスにより、特定の用途に応じてコンポーネントをカスタマイズすることができます。

インテル® スレッディング・ビルディング・ブロックを使用することで、直接スレッドを使用するよりもはるかに便利に並列処理を指定することができ、同時にパフォーマンスも向上させることができます。

このドキュメントはリファレンス・マニュアルです。構文とセマンティクスに関する詳細な情報が参照できるように構成されています。ライブラリーを効率的に使用する方法を学習するため、このドキュメントの前に、『*インテル® スレッディング・ビルディング・ブロック 入門ガイド*』および『*インテル® スレッディング・ビルディング・ブロック・チュートリアル*』をお読みください。

ヒント: インテル® スレッディング・ビルディング・ブロックは並列処理の優れた再帰モデルと汎用アルゴリズムを使用するため、並列処理に詳しいプログラマーの方々も、このリファレンス・マニュアルを使用する前に、『*インテル® スレッディング・ビルディング・ブロック・チュートリアル*』をお読みになることを推奨します。

2 一般的な規則

このセクションでは、このドキュメントで使用されている規則について説明します。

2.1 表記規則

リテラル・プログラム・テキストは Courier フォントで表記しています。代数プレースホルダーは斜体のモノスペース・フォントで表記しています。例えば、blocked_range<*Type*> は blocked_range がリテラルで、Type がプレースホルダーであることを示します。実際のプログラムテキストでは、*Type* は、blocked_range<int> のように実際の型に置き換えられます。

クラスメンバーは、実際にどのように実装されているかではなく、クライアントに見えるようにクラスを説明する略式宣言によって要約されます。例えば、次のような Foo クラスの略式宣言があるとします。

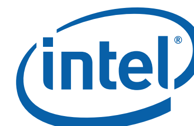
```
class Foo {
public:
    int x();
    int y;
    ~Foo();
};
```

実際の実装は次のようになります。

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

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

この例は、実際の実装が略式宣言から外れている 2 つのケースを示しています。

- x() メソッドは、protected 基本クラスから継承されている。
- デストラクターは、コンパイラーによって生成された暗黙のメソッドである。



略式宣言は、実装とは特に関係のない部分に気を散らすことなくクラスを使用するために、何を知っておく必要があるかを示します。

2.2 用語

このセクションでは、インテル® スレッディング・ビルディング・ブロック特有の用語について説明します。

2.2.1 コンセプト

コンセプト(concept) は、型の要件のセットです。要件は、構文的または意味的になります。例えば、"sortable" のコンセプトは配列のソートを可能にする要件のセットとして定義されます。型 `T` は、次の場合にソート可能です。

- `x < y` がブール値で、型 `T` の項目の全順序を表す。
- `swap(x,y)` が項目 `x` と `y` を交換する。

C++ で、ソート可能な型の配列をソートするテンプレート関数を記述することができます。

コンセプトを定義するための 2 つのアプローチは、*有効式* と *擬似署名* [1] です。ISO C++ 標準は、使用方法のパターンを示す有効式アプローチに従っていますが、このアプローチには、重要な詳細が表記規則に影響される欠点があります。このドキュメントでは、簡潔で、最初の実装にカットアンドペーストできるため、擬似署名を使用しています。

例えば、表 1 は、ソート可能な型 `T` の擬似署名を示しています。

表 1: コンセプト例 "sortable" の擬似署名

擬似署名	セマンティクス
`bool operator<(const T& x, const T& y)`	`x` と `y` を比較します。
`void swap(T& x, T& y)`	`x` と `y` を交換します。

[1] 有効式と擬似署名についての詳細は、*Concepts for C++0x* (http://www.osl.iu.edu/publications/prints/2005/siek05:_concepts_cpp0x.pdf) の 3.3.2 を参照してください。

実署名は暗黙的型変換が相違に対処する方法で実装する擬似署名とは異なります。例えば、C++ では int から bool への暗黙的型変換、および U から (const U&) への暗黙的型変換が可能であるため、型 U、表 1 の operator< を実装する実署名は、int operator<(U x, U y) として表現することができます。同様に、C++ では参照型への const 修飾子の暗黙的な追加も可能であるため、実署名 bool operator<(U& x, U& y) も許容されます。

2.2.2 モデル

コンセプトの要件を満たす場合、型はコンセプトを*モデル*にします。例えば、2 つの int 値 x と y を交換する関数 swap(x,y) が存在する場合、int 型は表 1 のソート可能なコンセプトをモデルにします。ソート可能なための別の要件、特に x<y は、int 型のビルドイン operator< によってすでに満たされています。

2.2.3 CopyConstructible

まれに、型が ISO C++ 標準で定義される CopyConstructible コンセプトをモデルにしなければならないことがあります。表 2 は、CopyConstructible の要件を擬似署名形式で示しています。

表 2: CopyConstructible の要件

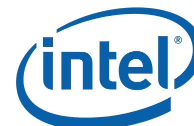
擬似署名	セマンティクス
T(const T&)	const T のコピーを構築します。
~T()	デストラクター
T* operator&()	アドレスを取得します。
const T* operator&() const	const T のアドレスを取得します。

2.3 識別子

このセクションでは、インテル® スレッディング・ビルディング・ブロックによって使用される識別子規則について説明します。

2.3.1 大文字と小文字

ライブラリーの識別子規則は、ISO C++ 標準ライブラリーのスタイルに従っています。識別子は underscore_style (下線付き)、コンセプトは PascalCase （パスカルケース）で記述されています。



2.3.2 予約済み識別子プリフィックス

ライブラリーは、ユーザーコードで直接参照しない内部識別子とマクロ用にプリフィックス __TBB を予約しています。

2.4 名前空間

このセクションでは、インテル® スレッディング・ビルディング・ブロックによって使用される予約済み名前空間について説明します。

2.4.1 tbb 名前空間

ライブラリーのすべてのパブリッククラスと関数は、名前空間 tbb に含まれています。

2.4.2 tbb::internal 名前空間

ライブラリーの内部識別子には、名前空間 tbb::internal を使用します。ユーザーコードで名前空間 tbb::internal またはその内部の識別子を直接参照しないでください。ヘッダーファイルで提供されるパブリック typedef 経由での間接参照は許可されます。

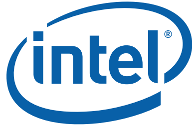
直接参照と間接参照を区別する例として、concurrent_vector<T>::iterator 型について説明します。この型は内部クラス internal::vector_iterator<Container,Value> の typedef です。ユーザーコードで、iterator typedef を使用してください。

2.5 スレッド安全性

特に明記されている場合を除いて、ライブラリーのスレッド安全性規則は次のとおりです。

- 2 つのスレッドは、異なるオブジェクトでメソッドまたは関数を同時に呼び出すことができます。
- 同じオブジェクトでメソッドまたは関数を同時に呼び出す 2 つのスレッドは安全ではありません。

クラスの説明では、この規則に沿っていない点を明記します。例えば、コンカレント・コンテナーはより自由です。コンカレント・コンテナーは、その性質により、同じコンテナー・オブジェクト上で並列操作を許可します。



2.6 デバッグ機能の有効化

ヘッダーには、特定のデバッグ機能を制御する 2 つのマクロが含まれています。一般に、これらの機能を開発コードではオン、製品コードではオフにしてコンパイルすると便利です。

2.6.1 TBB_DO_ASSERT マクロ

`TBB_DO_ASSERT` マクロは、エラーチェックをヘッダーファイルで有効にするかどうかを制御します。エラーチェックを有効にするには、`TBB_DO_ASSERT` を 1 として定義してください。

エラーが検出されると、ライブラリーはエラーメッセージを `stderr` に出力して標準 C ルーチン `abort` を呼び出します。内部エラーチェックがエラーを検出したときにプログラムを停止するには、`tbb::assertion_failure` にブレークポイントをセットしてください。

ヒント: Windows* システムでは、デバッグビルドは `TBB_DO_ASSERT` を `1` に暗黙的に設定します。

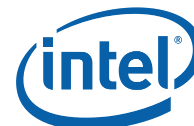
2.6.2 TBB_DO_THREADING_TOOLS マクロ

`TBB_DO_THREADING_TOOLS` マクロは、インテル® スレッド化ツールのサポートを制御します。

- インテル® スレッド・プロファイラー
- インテル® スレッドチェッカー

これらのツールのフルサポートを有効にするには、`TBB_DO_THREADING_TOOLS` を 1 として定義してください。ライブラリーのデバッグバージョンは常にフルサポートが有効になっています。

ツールの一部のサポートをオフにしてパフォーマンスを最大にするには、`TBB_DO_THREADING_TOOLS` を未定義のままにしておくか、0 として定義してください。現在の実装では、影響を受ける機能は `spin_mutex` (6.1.3) と `spin_rw_mutex` (6.1.6) のみです。



3 アルゴリズム

ライブラリーによって提供されるほとんどのアルゴリズムは汎用的です。それらのアルゴリズムは、必要なコンセプトをモデルにするすべての型で動作します。

3.1 分割可能なコンセプト

概要

インスタンスを 2 つのピースに分割可能な型の要件。

要件

表 3 は、インスタンス `x` で分割可能な型 `X` の要件を示しています。

表 3: 分割可能なコンセプト

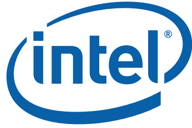
擬似署名	セマンティクス
`X::X(X& x, Split)`	`x` を `x` と新しく構築されるオブジェクトに分割します。

説明

型がインスタンスを 2 つのピースに分割することを許可する*分割コンストラクター*を含む場合、型は分割可能です。分割コンストラクターは、オリジナルのオブジェクトへの参照と、ライブラリーによって定義された `Split` 型の仮引数を引数にします。仮引数は分割コンストラクターとコピー・コンストラクターを区別します。コンストラクターを実行した後、*x* および新しく構築されるオブジェクトはオリジナルの *x* の 2 つのピースに相当します。ライブラリーは、分割コンストラクターを 2 つのコンテキストで使用します。

- *パーティション* - 範囲を同時に処理できる 2 つのサブ範囲に分割します。
- *フォーク* - ボディー (関数オブジェクト) から同時に実行できる 2 つのボディーを生成します。

次のモデル型は例を提供します。



モデル型

blocked_range (3.2.1) および blocked_range2d (3.2.2) は分割可能な範囲を表します。それぞれについて、分割は範囲を 2 つのサブ範囲に分割します。blocked_range<Value> の分割コンストラクターについては、セクション 3.2.1.3 の例を参照してください。

parallel_reduce (3.5) と parallel_scan (3.6) のボディーは分割可能でなければなりません。それぞれについて、分割により、同時に実行できる 2 つのボディーが生成されます。

3.1.1 split クラス

概要

分割コンストラクターの仮引数の型

構文

```
class split;
```

ヘッダー

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

説明

split 型の引数は、分割コンストラクターとコピー・コンストラクターを区別するために使用されます。

メンバー

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

3.2 Range コンセプト

概要

再帰的に分割可能な値のセットを表す型の要件。

要件

表 4 は、Range 型 `R` の要件を示しています。

表 4: Range コンセプト

擬似署名	セマンティクス
`R::R( const R& )`	コピー・コンストラクター
`R::~R()`	デストラクター
`bool R::empty() const`	範囲が空の場合は true です。
`bool R::is_divisible() const`	範囲を 2 つのサブ範囲に分割できる場合は true です。
`R::R( R& r, split )`	`r` を 2 つのサブ範囲に分割します。

説明

Range は 2 つの部分に再帰的に再分割することができます。分割後の作業がほぼ同等になることを推奨しますが、必須ではありません。できるだけ同等に分割することにより、一般的に、最良の並列処理が行われます。理想的には、さらに分割を進めるのではなく、直列で実行する方がより効率的となる作業を部分が表現するまで、範囲を再帰的に分割することができます。Range で表される作業の量は、一般的に、より高いレベルのコンテキストに依存します。このため、Range をモデルにする典型的な型では、分割の程度を制御する方法を提供してください。例えば、`blocked_range` (3.2.1) テンプレート・クラスには、分割不可能であると考えられる最も大きな範囲を指定する *grainsize* パラメーターがあります。

分割を実装するコンストラクターは、*分割コンストラクター*と呼ばれます。規則では、値のセットが向きの情報を含む場合、分割コンストラクターは範囲の 2 番目の部分を構築して、最初の半分になるように引数を更新します。この規則に従うと、順に実行するとき、`parallel_for` (3.4)、`parallel_reduce` (3.5)、および `parallel_scan` (3.6) アルゴリズムは通常の順次ループの典型的な増加順で範囲を介して動作します。

例

次のコードは、Range コンセプトをモデルにする `TrivialIntegerRange` 型を定義します。片方の整数まで分割可能な半分の区間 (lower,upper) を表しています。この半分の区間を半開区間と呼びます。

```
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 はデモ用です。粒度パラメーターがないため、あまり実用的ではありません。代わりに、blocked_range ライブラリー・クラスを使用してください。

モデル型

blocked_range (3.2.1) モデルは 1 次元の範囲です。

blocked_range2d (3.2.2) モデルは 2 次元の範囲です。

3.2.1 blocked_range<Value> テンプレート・クラス

概要

再帰的に分割可能な半開区間用のテンプレート・クラス。

構文

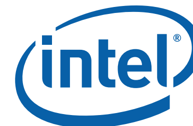
```
template<typename Value> class blocked_range;
```

ヘッダー

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

説明

blocked_range<Value> は、再帰的に分割できる半開範囲 [*i*,*j*) を表します。*i* および *j* の型は表 5 の要件をモデルにしなければなりません。要件が擬似署名であるため、暗黙的型変換によって異なる署名が許可されます。例えば、2 つの int 値は size_t に暗黙的に型変換できるため、blocked_range<int> は許可されます。Value 要件をモデルにする例は、相違が size_t に暗黙的に型変換できる整数型、ポインター、および STL ランダムアクセス・イテレーターです。



blocked_range は、Range コンセプト (3.2) をモデルにします。

表 5: blocked_range の Value コンセプト

擬似署名	セマンティクス
`Value::Value( const Value& )`	コピー・コンストラクター
`Value::~Value()`	デストラクター
`bool operator<( const Value& i, const Value& j )`	値 *i* は値 j に先行します。
`size_t operator−( const Value& i, const Value& j )`	範囲 [i,j) にある値の数です。
`Value operator+( const Value& i, size_t k )`	*i* の後の *k* 番目の値です。

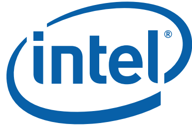
blocked_range<Value> は、size_t 型の *grainsize* を指定します。範囲のサイズが *grainsize* を超える場合、blocked_range は 2 つのサブ範囲に分割可能です。理想的な粒度は、parallel_for ループ・テンプレート、parallel_reduce ループ・テンプレート、または parallel_scan ループ・テンプレートの典型的な範囲引数である blocked_range<Value> のコンテキストに依存します。粒度が非常に小さい場合、並列処理による速度向上よりもループ・テンプレート内のオーバーヘッドに時間がかかるようになります。粒度が非常に大きい場合、不必要に並列処理を制限することがあります。例えば、粒度が非常に大きいために範囲を 1 回しか分割できない場合、最大の可能な並列処理は 2 になります。

grainsize を選択する推奨手順を次に示します。

1. 粒度パラメーターを 10,000 に設定します。この値は、すべてのループボディーでスケジューラー・オーバーヘッドの影響を考慮する必要がない高い値ですが、不必要に並列処理を制限します。
2. アルゴリズムを *1 つの* プロセッサー上で実行します。
3. 粒度パラメーターを半分にして、値の減少とともにアルゴリズムがどの程度遅くなるか確認します。

約 5-10% 遅くなる値が、汎用的に利用可能な設定です。

ヒント: blocked_range [i,j) で j<i の場合、すべてのメソッドに動作が指定されているとは限りません。しかし、j<i の場合でも、十分なメソッドに、parallel_for (3.4)、parallel_reduce (3.5)、および parallel_scan (3.6) が直列ループ (Value index=i; index<j; ++index)... と同じ反復空間上を反復する動



作が指定されています。 TBB_DO_ASSERT (2.6.1) が非ゼロの場合、動作が指定されていないメソッドではアサーション・エラーが発生します。

例

blocked_range<Value> は、通常、ループ・テンプレートの範囲引数として使用されます。parallel_for (3.4)、parallel_reduce (3.5)、および parallel_scan (3.6) の例を参照してください。

メンバー

```
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

説明

blocked_range のサイズを測定する型です。型は常に size_t です。

```
const_iterator
```

説明

範囲内の値の型です。その名前に反して、const_iterator 型は必ずしも STL イテレーターではありません。表 5 の Value 要件を満たす必要があるだけです。しかし、型が const_iterator である場合、blocked_range は読み取り専用の STL コンテナーのように動作するので、この型を const_iterator と呼ぶと便利です。

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

要件

パラメーター grainsize が正であること。この要件が満たされない場合、ライブラリーのデバッグバージョンはアサーション・エラーになります。

結果

指定された grainsize で、半開区間 (begin,end) を表す blocked_range を構築します。

例

"blocked_range<int> r(5, 14, 2);" 文は、粒度 2 で、値 5 から 13 までを含む int の範囲を構築します。その後、r.begin()==5 および r.end()==14 に設定します。

3.2.1.3 blocked_range(blocked_range& range, split)

要件

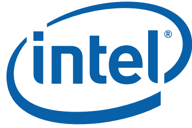
is_divisible() が true であること。

結果

range を 2 つのサブ範囲に分割します。新しく構築される blocked_range は、ほぼオリジナルの range の半分で、range は残りになるように更新されます。各サブ範囲の粒度は、オリジナルの range と同じ grainsize になります。

例

i および j を半開区間 (i,j) を定義する整数、g を粒度とします。blocked_range<int> r(i,j,g) 文は、粒度 g で、(i,j) を表す blocked_range<int> を構築します。



blocked_range<int> s(r,split); 文を実行すると、粒度 g で、r は (i, i +(j −i)/2)、s は (i +(j −i)/2, j) を表します。

3.2.1.4 size_type size() const

要件

end()<begin() が false であること。

結果

範囲のサイズを決定します。

リターン

end()−begin()

3.2.1.5 bool empty() const

結果

範囲が空かどうかを決定します。

リターン

!(begin()<end())

3.2.1.6 size_type grainsize() const

リターン

範囲の粒度。

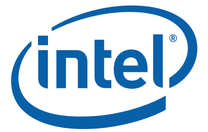
3.2.1.7 bool is_divisible() const

要件

!(end()<begin())

結果

範囲がサブ範囲に分割できるかどうかを決定します。



リターン

`size()>grainsize()` の場合は true。その他の場合は false。

3.2.1.8 const_iterator begin() const

リターン

範囲の包含的な下限。

3.2.1.9 const_iterator end() const

リターン

範囲の排他的な上限。

3.2.2 blocked_range2d テンプレート・クラス

概要

再帰的に分割可能な 2 次元の半開区間を表すテンプレート・クラス。

構文

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

ヘッダー

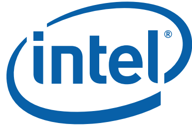
```
#include "tbb/blocked_range2d.h"
```

説明

`blocked_range2d<`*RowValue*, *ColValue*`>` は、半開 2 次元範囲 $(i_0,j_0)\times(i_1,j_1)$ を表します。範囲の各軸には、独自の分割しきい値があります。*RowValue* および *ColValue* は表 5 の要件を満たしていなければなりません。いずれかの軸が分割可能な場合、`blocked_range` は分割可能です。`blocked_range` は、Range コンセプト (3.2) をモデルにします。

メンバー

```
namespace tbb {
template<typename RowValue, typename ColValue=RowValue>
    class blocked_range2d {
```



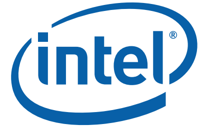
```
    public:
        // Types
        typedef blocked_range<RowValue> row_range_type;
        typedef blocked_range<ColValue> col_range_type;

        // 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;
    };
}
```

例

次のコードは、直列行列乗算と、`blocked_range2d` を使用して反復空間を指定する、対応する並列行列乗算を示しています。

```
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) );
}
```

blocked_range2d は、直列バージョンの 2 つの最外ループを並列ループにします。parallel_for は、ピースが 16x32 以下になるまで blocked_range2d を再帰的に分割して、各ピースで MatrixMultiplyBody2D::operator() を呼び出します。

3.2.2.1 row_range_type

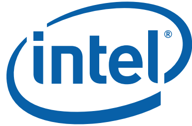
説明

blocked_range<RowValue>。つまり、行の値の型です。

3.2.2.2 col_range_type

説明

blocked_range<ColValue>。つまり、列の値の型です。



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)

結果

値の 2 次元空間を表す blocked_range2d を構築します。空間は、行と列が指定された粒度の、半開直積 (row_begin,row_end)× (col_begin,col_end) です。

例

"blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2);" 文は、形式 (i, j) のすべての値ペアを含む 2 次元の空間を構築します。ここで、i の範囲は 'a' から 'z' で粒度 3、j の範囲は 0 から 9 で粒度 2 です。

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

結果

range を 2 つのサブ範囲に分割します。新しく構築される blocked_range2d は、ほぼオリジナルの range の半分で、range は残りになるように更新されます。各サブ範囲の粒度は、オリジナルの range と同じ粒度になります。分割は行と列のいずれかで行われます。分割する軸を選択すると、分割を繰り返した後、サブ範囲は行と列粒度の比率に近くなります。例えば、row_grainsize が col_grainsize の 2 倍の場合、サブ範囲は列の 2 倍の行を持つようになります。

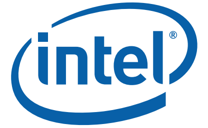
3.2.2.5 bool empty() const

結果

範囲が空かどうかを決定します。

リターン

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



3.2.2.6 bool is_divisible() const

結果

範囲がサブ範囲に分割できるかどうかを決定します。

リターン

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

3.2.2.7 const row_range_type& rows() const

リターン

値空間の行を含む範囲。

3.2.2.8 const col_range_type& cols() const

リターン

値空間の列を含む範囲。

3.3 Partitioner コンセプト

概要

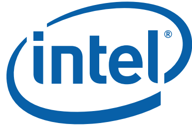
範囲 (3.2) をタスクボディーで操作するか、範囲をより分割するかを決定する型の要件。

要件

表 6 は、Partitioner 型 `P` の要件を示しています。

表 6: Partitioner コンセプト

擬似署名	セマンティクス
`P::~P()`	デストラクター
`template <typename Range>` `bool P::should_execute_range(const Range &r, const task &t)`	`r` を `t` のボディーに渡す場合は true です。`r` を分割する場合は false です。
`P::P( P& p, split )`	`p` を 2 つのパーティショナーに分割します。



説明

Partitioner コンセプトは、指定された範囲の再分割をやめても、タスクのボディーによって全体として操作するような時機を決定する規則を実装します。

`parallel_for` (3.4) アルゴリズム、`parallel_reduce` (3.5) アルゴリズムおよび `parallel_scan` (3.6) アルゴリズムのデフォルトの動作は、Range コンセプト (3.2) の関数 `is_divisible` によって決定されるように、分割可能なサブ範囲がなくなるまで、範囲を再帰的に分割します。Partitioner コンセプトは、デフォルト動作を変更できるようにして、範囲の再帰的な分割を早めに終了するための規則をモデルにします。Partitioner オブジェクトの意思決定は、分割コンストラクターと `should_execute_range` 関数の 2 つの関数を使用して行われます。

並列アルゴリズム内では、各 Range オブジェクトは Partitioner オブジェクトと関連しています。Range オブジェクトが 2 つのサブ範囲を作成するように分割コンストラクターを使用して分割されると、関連する Partitioner オブジェクトも 2 つの一致する Partitioner オブジェクトを作成するように同様に分割されます。

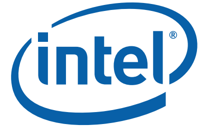
`parallel_for` アルゴリズム、`parallel_reduce` アルゴリズム、または `parallel_scan` アルゴリズムで範囲をさらに再分割するかどうかを決定する必要がある場合、アルゴリズムはその範囲に関連する Partitioner オブジェクトの `should_execute_range` 関数を呼び出します。`should_execute_range` 関数が指定された範囲とタスクに対して true を返す場合、その範囲でさらなる分割は行われず、現在のタスクがそのボディーを範囲全体に適用します。

例

次のコードは、Partitioner コンセプトをモデルにする型 `simple_partitioner` を定義します。範囲関数 `is_divisible` が false を返す場合、その関数 `should_execute_range` から true を返します。

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

このクラスは、範囲が再分割できなくなるまで範囲を分割する、デフォルトの動作をコード化します。



モデル型

simple_partitioner (3.3.1) は、範囲が再分割できなくなるまで範囲を分割する、デフォルトの動作をモデルにします。

auto_partitioner (3.3.2) は、task_scheduler (8) のワークスチール動作をモニターし、分割の数を減らしてモデルにします。

3.3.1 simple_partitioner クラス

概要

範囲が再分割できなくなるまで範囲を分割する、parallel_for (3.4) アルゴリズム、parallel_reduce (3.5) アルゴリズムおよび parallel_scan (3.6) アルゴリズムのデフォルト範囲分割動作をモデルにするクラス。

構文

```
class simple_partitioner;
```

ヘッダー

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

説明

simple_partitioner クラスは、parallel_for アルゴリズム、parallel_reduce アルゴリズムおよび parallel_scan アルゴリズムのデフォルトの範囲分割動作をモデルにします。

3.3.1.1 simple_partitioner()

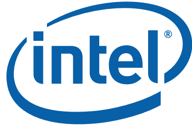
空のデフォルト・コンストラクター。

3.3.1.2 simple_partitioner(simple_partitioner &partitioner, split)

空の分割コンストラクター。

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

提供された範囲が指定されたタスクにより完了するまで実行する場合に true を返す関数。!range.is_divisible() を返します。



3.3.2 auto_partitioner クラス

概要

task_scheduler のワークスチール動作をモニターして、行う分割の数を管理する適応パーティショナーをモデルにするクラス。

構文

```
class auto_partitioner;
```

ヘッダー

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

説明

auto_partitioner クラスは、ワークスチール・イベントに反応することでロード・バランシングに必要な分割の数を制限する適応パーティショナーをモデルにします。

範囲は、最初に S_I サブ範囲に分割されます。ここで、S_I はタスク・スケジューラーによって作成されたスレッド数に比例します。これらのサブ範囲は、スチールされなければ、タスクによって完了まで実行されます。サブ範囲がアイドルスレッドによってスチールされた場合、auto_partitioner は範囲をさらに再分割して追加のサブ範囲を作成します。

スレッドが積極的に作業をスチールしている場合のみ、auto_partitioner は追加のサブ範囲を作成します。負荷のバランスが取れている場合、少数の大きな最初のサブ範囲のみを使用すると、範囲を分割または連結するときに発生するオーバーヘッドが減少します。しかし、ワークスチールによるロード・インバランスがある場合、auto_partitioner は、スチールできる追加のサブ範囲を作成して負荷のバランスを取ります。

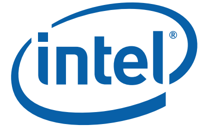
そのため、ワークスチールの十分な機会が提供されている間、auto_partitioner は範囲分割の数を最小限に抑えようとします。

3.3.2.1 auto_partitioner()

空のデフォルト・コンストラクター。

3.3.2.2 auto_partitioner(auto_partitioner &partitioner, split)

`auto_partitioner` パーティショナーを 2 つのパーティショナーに分割する分割コンストラクター。



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

提供された範囲を指定されたタスクのボディーによって全体として操作する場合に true を返す関数。この関数は、`range.is_divisible() == true` の場合でも true を返すことがありますが、`range.is_divisible() == false` の場合は常に true を返します。つまり、この関数は、`t` がさらに再分割できる `r` を処理すると決定することがあります。しかし、この関数は、`t` がさらに再分割できない `r` を処理すると常に決定します。

3.4 parallel_for<Range,Body> テンプレート関数

概要

値の範囲で並列反復を行うテンプレート関数。

構文

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

ヘッダー

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

説明

`parallel_for<Range,Body>` は、`Range` の各値について `Body` の並列実行を表します。`Range` 型は、Range コンセプト (3.2) をモデルにしなければなりません。ボディーは、表 7 の要件をモデルにしなければなりません。

表 7: parallel_for のボディーの要件

擬似署名	セマンティクス
`Body::Body( const Body& )`	コピー・コンストラクター
`Body::~Body()`	デストラクター
`void Body::operator()( Range& range ) const`	`range` にボディーを適用します。

parallel_for は、is_divisible() が各サブ範囲で false になるポイントまで、範囲をサブ範囲に再帰的に分割して、これらの各サブ範囲についてボディーのコピーを作成します。各ボディー/サブ範囲ペアについて、Body::operator() を呼び出します。空間のオーバーヘッドを最小化してキャッシュを効率的に使用するために、呼び出しは再帰的な分割を使用して交互に配置されます。

範囲とボディーのコピーの一部は、parallel_for のリターンの後に破棄されます。この後からの破棄は典型的な使用方法では問題ありませんが、複雑な副作用がある実行トレースの検索、範囲やボディー・オブジェクトの記述を行う際には注意する必要があります。

ワーカースレッドが利用可能な場合 (8.2)、parallel_for は非決定性順に反復を実行します。正確性のためには、特定の実行順に依存してはなりません。しかし、効率のためには、parallel_for が値の連続する順に実行することを予想してください。

ワーカースレッドが利用できない場合、parallel_for は、次のように左から右に反復を実行します。再帰的な分割を表すバイナリーツリーを描くことを想像してください。各ノンリーフノードは、分割コンストラクター Range(r,split()) を呼び出してサブ範囲 r を分割することを表します。左の子は、r の更新された値を表します。右の子は、新しく構築されたオブジェクトを表します。ツリーの各リーフは、個々のサブ範囲を表します。Body::operator() メソッドが各リーフのサブ範囲上で、左から右に呼び出されます。

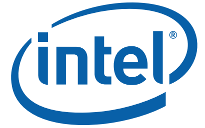
計算量

範囲とボディーが O(1) 空間を使用して範囲をほぼ等しい断片に分割する場合、空間計算量は O(P log(N)) です。ここで、N は範囲のサイズ、P はスレッド数です。

例

この例は、input[i-1]、input[i]、および input[i+1] (0≤i<n の場合) の平均を output[i] に設定する ParallelAverage ルーチンを定義します。

```
#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 );
}
```

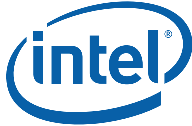
例

この例はより複雑で、STL に精通している必要があります。この例は、フラットな反復空間が及ばない `parallel_for` の能力を示します。コードは、2 つのソートされたシーケンスの並列マージを行います。コードは、ランダムアクセス・イテレーターを使用して任意のシーケンスで動作します。アルゴリズムは、次のように再帰的に動作します。

1. シーケンスが並列処理を使用するには短すぎる場合は、順次マージを行います。その他の場合は、ステップ 2-6 を行います。
2. 必要な場合、シーケンスを交換します。その結果、最初のシーケンス (begin1,end1) は少なくとも 2 番目のシーケンス (begin2,end2) と同じくらいの長さになります。
3. m1 を (begin1,end1) の中央の位置に設定します。その位置 *key* の項目を呼び出します。
4. m2 を *key* が (begin2,end2) になる位置に設定します。
5. (begin1,m1) と (begin2,m2) をマージしてマージドシーケンスの最初の部分を作成します。
6. (m,end1) と (m2,end2) をマージしてマージドシーケンスの 2 番目の部分を作成します。

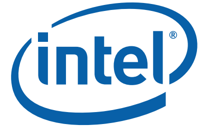
このアルゴリズムによるインテル® スレッディング・ビルディング・ブロックの実装は、範囲オブジェクトを使用してほとんどのステップを行います。`is_divisible` はステップ 1 とステップ 2 のテストを行います。分割コンストラクターはステップ 3-6 を行います。ボディー・オブジェクトは順次マージを行います。

```
#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 first sequence to be
merged
    Iterator begin2, end2;       // [begin2,end2) is first sequence to be
merged
    Iterator out;                // where to put merged sequence
```




```
    bool empty()   const {return (end1-begin1)+(end2-begin2)==0;}
    bool is_divisible() {
        if( end1-begin1 < end2-begin2 ) {
            std::swap(begin1,begin2);
            std::swap(end1,end2);
        }
        // [begin2,end2) is now at least as short as [begin1,end1)
        return end2-begin2 > grainsize;
    }
    ParallelMergeRange( ParallelMergeRange& r, split ) {
        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>() );
}
```

アルゴリズムは多くの位置を移動するため、帯域幅が制限されることがあります。速度向上は、システムによって異なります。



3.4.1 Partitioner 機能の使用

概要

Partitioner パラメーターによってガイドされた範囲の分割を使用して、値の範囲で並列反復を行うテンプレート関数。

構文

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

ヘッダー

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

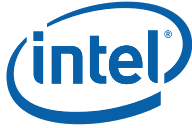
説明

`parallel_for<Range,Body,Partitioner>` は、`Range` の各値について `Body` の並列実行を表します。`Range` 型は、Range コンセプト (3.2) をモデルにしなければなりません。ボディーは、表 7 の要件をモデルにしなければなりません。`Partitioner` 型は 、Partitioner コンセプト (3.3) をモデルにしなければなりません。

例

この例は、`parallel_for` を使用した Partitioner コンセプトの単純な使用方法を示します。次のコードは、以前のサブセクションで示された単純な例を拡張したものです。 `auto_partitioner` は、範囲の分割をガイドするために使用されます。

```
#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() );
}
```

2 つの重要な変更に注意する必要があります。(1) parallel_for への呼び出しで 3 つ目の引数 auto_partitioner オブジェクトを使用している。(2) blocked_range コンストラクターで粒度パラメーターが提供されていない。

セクション 3.2.1 および 3.2.2 で説明されているコンストラクターに加えて、blocked_range テンプレート・クラスおよび blocked_range2d テンプレート・クラスは、すべての粒度パラメーターを 1 に初期化する追加のコンストラクターを定義するようになりました。これらのクラスでは、粒度は、範囲が分割可能であると考えられるサイズを指定するために使用されます。

表 8 は、simple_partitioner クラスと auto_partitioner クラスを選択するためのガイダンスを示しています。

表 8: Partitioner を選択するためのガイダンス

Partitioner の種類	説明
`simple_partitioner`	範囲が分割できなくなるまで再帰的に範囲を分割します。`Range::is_divisible` 関数は、再帰的な分割をいつ停止するかを決定することに関して全責任を負います。`blocked_range` や `blocked_range2d` のようなクラスとともに使用された場合、オーバーヘッドを制限する一方で並列化を許可するため、粒度の選択は非常に重要です (セクション 3.2.1 の説明を参照)。
`auto_partitioner`	タスク・スケジューラーのワークスチール動作に基づいて分割決定をガイドします。`blocked_range` や `blocked_range2d` のようなクラスとともに使用された場合、適切な粒度の選択はそれほど重要ではありません。ロード・インバランスが検出されなければ、粒度よりも大きなサブ範囲が使用されます。そのため、デフォルトの粒度 1 を使用するだけで許容可能なパフォーマンスがしばしば達成されます。

ヒント: auto_partitioner を使用すると、粒度よりも大きな範囲がボディーに渡されます。そのため、ボディーが (例えば、一時記憶領域の割り当てで) 粒度の値を範囲のサイズの上限として使用しないようにしてください。

3.5 parallel_reduce<Range,Body> テンプレート関数

概要

範囲のリダクションを計算します。

構文

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

ヘッダー

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

説明

parallel_reduce<Range,Body> は、Range の各値について Body の並列リダクションを行います。Range 型は、Range コンセプト (3.2) をモデルにしなければなりません。Body は、表 9 の要件をモデルにしなければなりません。

表 9: parallel_reduce **の Body の要件**

擬似署名	セマンティクス
`Body::Body( Body&, split );`	分割コンストラクター (3.1)。operator() および join メソッドと同時に実行できなければなりません。
`Body::~Body()`	デストラクター
`void Body::operator()( Range& range );`	サブ範囲の結果を累積します。
`void Body::join( Body& rhs );`	結果を連結します。rhs の結果は this の結果にマージしてください。

parallel_reduce は、is_divisible() が各サブ範囲で false になるポイントまで、範囲をサブ範囲に再帰的に分割します。parallel_reduce は、分割コンストラクターを使用して各スレッドのボディーの 1 つ以上のコピーを作成します。ボディーの operator() または join メソッドが同時に実行している間、ボディーをコピーします。ユーザーは、そのような並列化の安全性を保証する責任を負います。典型的な使用方法では、安全性のために余分な労力は必要ありません。

ワーカースレッドが利用可能な場合 (8.2.1)、parallel_reduce はボディーに対して分割コンストラクターを呼び出します。そのようなボディーの各分割について、ボディーから結果をマージするために join メソッドを呼び出します。this が this と rhs の累積された結果を表すように join を定義します。リダクション演算は連想演算に違いありませんが、可換演算である必要はありません。非可換演算 *op* では、"*left*.join(*right*)" は *left* を *left op right* の結果になるように更新します。

ボディーは範囲が分割している場合のみ分割されますが、逆は必ずしもそうではありません。図 1 は、parallel_reduce のサンプル実行を図解したものです。ルートは、半開区間 [0,20) に適用されているオリジナルボディー b0 を表します。範囲は各レベルで 2 つのサブ範囲に再帰的に分割

されます。例の粒度は 5 なので、4 つのリーフ範囲が生成されます。斜線 (/) は、どこでボディー分割コンストラクターによってボディーのコピー (b_1 および b_2) が作成されたかを示します。ボディー b_0 および b_1 は、それぞれ 1 つのリーフを評価します。ボディー b_2 は、リーフ [10,15) および [15,20) を順に評価します。ツリーを戻る途中で、`parallel_reduce` は b_0.join(b_1) と b_0.join(b_2) を呼び出してリーフの結果をマージします。

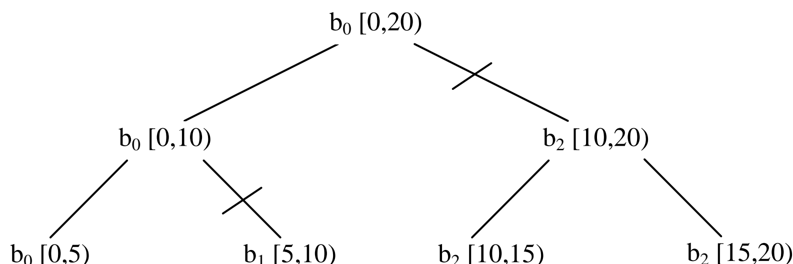


図 1: blocked_range<int>(0,20,5) **での** parallel_reduce **の実行例**

図 1 は、1 つの可能な実行のみ示しています。ほかの有効な実行には、b_2 から b_2 および b_3 に分割すること、または全く分割しないことが含まれます。分割しない場合、b_0 は、`join` を呼び出すことなく、各リーフを左から右に順に評価します。指定されたボディーは常に 1 つ以上の連続するサブ範囲を左から右に順に評価します。例えば、図 1 では、ボディー b_2 は [10,15) を [15,20) の前に評価することが保証されています。ボディーの指定されたインスタンスの左から右に連続する特性には依存してもかまいませんが、ボディー分割の特定の選択に依存してはなりません。`parallel_reduce` は、ボディーの分割を非決定性で行います。

ワーカースレッドが利用できない場合、`parallel_reduce` は `parallel_for` (3.4) と同じ方法で左から右に順に実行します。順次実行は、分割コンストラクターや `join` メソッドを呼び出しません。

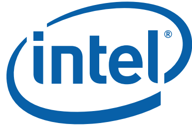
計算量

範囲とボディーが O(1) 空間を使用して範囲をほぼ等しいピースに分割する場合、空間計算量は O(P log(N)) です。ここで、N は範囲のサイズ、P はスレッド数です。

例

次のコードは、配列の値を合計します。

```
#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;
}
```

任意の連想演算 *op* でのリダクションのために、例を次のように一般化します。

- 0 を *op* の単位元に置換します。
- `+=` を *op*= またはその論理的同値に置換します。
- 名前 `Sum` を *op* でより適切な名前に変更します。

演算は非可換であってもかまいません。例えば、*op* は行列乗算であってもかまいません。

3.5.1 Partitioner 機能の使用

概要

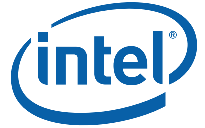
Partitioner パラメーターによってガイドされた範囲の分割を使用して、範囲のリダクションを計算します。

構文

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

ヘッダー

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



説明

parallel_reduce<Range,Body> は、Range の各値について Body の並列リダクションを行います。Range 型は、Range コンセプト (3.2) をモデルにしなければなりません。Body は、表 9 の要件をモデルにしなければなりません。Partitioner 型は、Partitioner コンセプト (3.3) をモデルにしなければなりません。

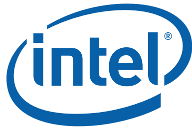
例

次のコードは、auto_partitioner を使用して以前のセクションの例を拡張します。

```
#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;
}
```

2 つの重要な変更に注意する必要があります。(1) parallel_reduce への呼び出しで 3 つ目の引数 auto_partitioner オブジェクトを使用している。(2) blocked_range コンストラクターで粒度パラメーターが提供されていない。セクション 3.4.1 で説明したように、blocked_range はデフォルトで粒度を 1 に設定する追加コンストラクターをサポートします。

表 8 は、simple_partitioner クラスと auto_partitioner クラスを選択するためのガイダンスを示しています。



3.6 parallel_scan<Range,Body> テンプレート関数

概要

並列プリフィックスを計算するテンプレート関数。

構文

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

ヘッダー

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

説明

parallel_scan<Range,Body> は、並列プリフィックス (並列スキャンとも呼ばれる) を計算します。この計算は、並列計算における高度なコンセプトで、本質的に直列に依存しているように見えるシナリオで役立つことがあります。

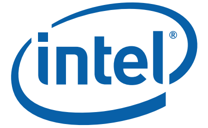
並列プリフィックスの数学的な定義は次のとおりです。⊕ を左単位元が $id_{\oplus}$ の連想演算 ⊕ にします。シーケンス $x_0, x_1, ...x_{n-1}$ の ⊕ の並列プリフィックスはシーケンス $y_0, y_1, y_2, ...y_{n-1}$ です。

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

例えば、⊕ が加算の場合、並列プリフィックスは合計と一致します。並列プリフィックスの直列実装は次のとおりです。

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

並列プリフィックスは、⊕ のアプリケーションを再結合して 2 パスを使用することで、並列にこれを行います。並列プリフィックスは、直列プリフィックス・アルゴリズムの 2 倍まで ⊕ を呼び出します。正しい粒度と十分なハードウェア・スレッドが指定されると、より多くの作業を行う場合でも、複数のハードウェア・スレッドにわたって作業を分散することができるので、直列プリフィックスより性能が優れています。



ヒント: parallel_scan は 2 つのパスが必要なため、2 つのハードウェア・スレッドしかないシステムでは速度向上はわずかです。parallel_scan は、将来のマルチコアシステム用の技術の中でも非常に有望なものの 1 つです。本質的に順次に見える問題をどのように並列化することができるかを示しているため、その技術自体も興味深いものです。

parallel_scan<Range,Body> テンプレートは、並列プリフィックスを汎用的に実装します。表 10 で説明されている署名が必要です。

表 10: parallel_scan の要件

擬似署名	セマンティクス
void Body::operator()(const Range& r, pre_scan_tag)	範囲 r の反復の前処理を行います。
void Body::operator()(const Range& r, final_scan_tag)	範囲 r の反復の最終処理を行います。
Body::Body(Body& b, split)	this と b を別々に累積できるように b を分割します。
void Body::reverse_join(Body& a)	a の前処理状態を this にマージします。ここで、a は前に b の分割コンストラクターによって b から作成されます。
void Body::assign(Body& b)	b の状態を this に代入します。

次のコードは、順次の例に似た方法で parallel_scan を使用するためにこれらの署名をどのように実装しなければならないかを説明しています。

```
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,1000), body );
    return body.get_sum();
}
```

operator() の定義は、parallel_scan を使用するときの典型的なパターンを説明します。

- 1 つのテンプレートで両方のバージョンを定義します。これは必須ではありませんが、2 つのバージョンは通常似ているため、コーディングの労力を省くことができます。ライブラリーは、バージョンを区別できるように、スタティック・メソッド is_final_scan() を定義します。
- プリスキャン可変要素は ⊕ リダクションを計算しますが、y を更新しません。プリスキャンはルックアヘッドの局部リダクションを生成するために parallel_scan によって使用されます。
- 最後のスキャン可変要素は ⊕ リダクションを計算して y を更新します。

reverse_join 操作は、引数が逆であることを除けば、parallel_reduce によって使用される join 操作に似ています。つまり、this は ⊕ の*右*の引数です。parallel_scan テンプレート関数は、並列作業を生成するかどうか、およびいつ生成するかを決定します。したがって、⊕ が連想演算で Body のメソッドがそれを正確に表現することは重要です。やや連想型の浮動小数点の追加のような演算は、parallel_scan で使用される連想に依存して結果が異なって丸められる場合があることを理解して使用する必要があります。再連想は、同じマシン上の実行の間でも異なる場合があります。しかし、利用可能なワーカースレッドがない場合、実行はこのセクションの最初で説明した直列形式と同一であると連想されます。

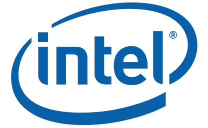
3.6.1 pre_scan_tag and final_scan_tag クラス

概要

parallel_scan のフェーズを識別する型。

構文

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



ヘッダー

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

説明

pre_scan_tag 型と final_scan_tag 型は、parallel_scan とともに使用される仮の型です。これらの型が operator() の署名でどのように使用されるかについては、セクション 3.6 の例を参照してください。

メンバー

```
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()

リターン

final_scan_tag の場合は true、その他の場合は false。

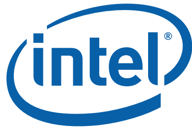
3.6.2 Partitioner 機能の使用

概要

Partitioner パラメーターによってガイドされた範囲の分割を使用して、並列プリフィックスを計算するテンプレート関数。

構文

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



ヘッダー

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

説明

parallel_scan<Range,Body,Partitioner> は、並列プリフィックス (並列スキャンとも呼ばれる) を計算します (並列プリフィックスの一般的な説明については、セクション 3.6 を参照)。

例

次のコードは、auto_partitioner を使用して以前のセクションの例を拡張します。

```
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();
}
```

2 つの重要な変更に注意する必要があります。(1) parallel_scan への呼び出しで 3 つ目の引数 auto_partitioner オブジェクトを使用している。(2) blocked_range コンストラクターで粒度パラメーターが提供されていない。セクション 3.4.1 で説明したように、blocked_range はデフォルトで粒度を 1 に設定する追加コンストラクターをサポートします。

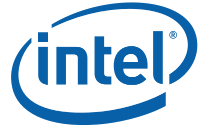


表 8 は、simple_partitioner クラスと auto_partitioner クラスを選択するためのガイダンスを示しています。

3.7 parallel_while テンプレート・クラス

概要

作業項目を処理するテンプレート・クラス。

構文

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

ヘッダー

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

説明

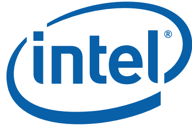
parallel_while<Body> は、項目の並列反復を行います。各項目で行う処理は、Body 型の関数オブジェクトによって定義されます。項目は 2 つの方法で指定されます。

1. 項目のストリーム。
2. ストリームの処理中に追加される追加項目。

表 11 は、ストリームとボディーの要件を示しています。

表 11: ストリーム S とボディー B の parallel_while の要件

擬似署名	セマンティクス
`bool S::pop_if_present( B::argument_type& item )`	次のストリーム項目を取得します。parallel_while は、同じ this のメソッドを同時に呼び出しません。
`B::operator()( B::argument_type& item ) const`	item を処理します。parallel_while は、同じ this で item が異なる operator を同時に呼び出します。
`B::argument_type()`	デフォルト・コンストラクター



`B::argument_type( const B::argument_type& )`	コピー・コンストラクター
`~B::argument_type()`	デストラクター

例えば、C++ 標準のセクション 20.3 で定義されている単項の関数オブジェクトは、B の要件をモデルにします。`concurrent_queue` (4.2) は、S の要件をモデルにします。

ヒント: 速度を向上するには、`B::operator()` の粒度は少なくともほぼ 10,000 である必要があります。粒度が小さい場合、`parallel_while` の内部オーバーヘッドが大きくなります。すべての項目が入力ストリームからの項目である場合、`parallel_while` の並列処理はスケーラブルではありません。スケーリングを達成するには、`add` メソッドが作業の断片を 2 つ以上追加するようにアルゴリズムを設計してください。

メンバー

```
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>()

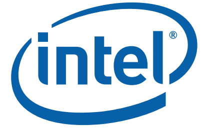
結果

まだ実行していない `parallel_while` を構築します。

3.7.2 ~parallel_while<Body>()

結果

`parallel_while` を破棄します。



3.7.3 テンプレート <typename Stream> void run(Stream& stream, const Body& body)

結果

body を *stream* の各項目と `add` メソッドによって追加されたほかの項目に適用します。次の条件が両方とも真の場合に終了します。

1. `stream.pop_if_present` が false を返した
2. `body(x)` がストリームまたは add メソッドから生成したすべての項目 *x* について返された。

3.7.4 void add(const value_type& item)

要件

`parallel_while` によって作成された *body*.`operator()` への呼び出しから呼び出されること。その他の場合、`run` メソッドの終了セマンティクスは未定義です。

結果

処理する項目のコレクションに項目を追加します。

3.8 pipeline クラス

概要

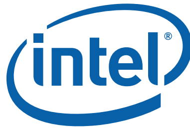
パイプライン実行を行う抽象的な基本クラス。

構文

```
class pipeline;
```

ヘッダー

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



説明

`pipeline` は、項目のストリームへのフィルターを含む、パイプライン化された一連のアプリケーションを表します。各フィルターは並列または直列です。詳細は、`filter` クラス (3.8.6) を参照してください。

パイプラインは、ここで f_i として示された 1 つ以上のフィルターを含みます。*i* は、パイプラインにおけるフィルターの位置を示します。パイプラインは、f_0 フィルターで開始し、f_1、f_2、... と続きます。次のステップで、クラス・パイプラインの使用方法を説明します。

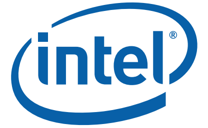
1. `filter` から f_i クラスを派生します。f_i のコンストラクターは、基本クラス `filter` (3.8.6.1) のコンストラクターへのブール・パラメーターにより直列かどうかを指定します。
2. 項目でフィルターのアクションを行い、次の filter によって処理される項目へのポインターを返すように仮想メソッド `filter::operator()` をオーバーライドします。最初のフィルター f_0 はストリームを生成します。ストリームに項目がそれ以上ない場合、NULL を返します。最後のフィルターのリターン値は無視されます。
3. `pipeline` クラスのインスタンスを作成します。
4. f_i フィルターのインスタンスを作成して、最初から最後の順にパイプラインに追加します。フィルターのインスタンスは、パイプラインに多くて一度追加することができます。フィルターは、同時に 2 つ以上のパイプラインのメンバーであってはなりません。
5. `pipeline::run` メソッドを呼び出します。`max_number_of_live_tokens` パラメーターは、同時に実行するステージ数の上限を設定します。高い値を設定すると、同時に実行できる数は増加しますが、処理する項目が多くなるためメモリーの消費量がより多くなります。`max_number_of_live_tokens` の効果的な使用方法に関する詳細は、チュートリアルの `pipeline` クラスのセクションを参照してください。

十分なプロセッサーとトークンが提供された場合、パイプラインの処理能力は最も遅い直列フィルターの処理能力に制限されます。

`filter` は、破棄する前に `pipeline` から削除しなければなりません。最初に `pipeline` を破棄するか、`pipeline::clear()` を呼び出してください。

メンバー

```
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()

結果

フィルターなしのパイプラインを構築します。

3.8.2 ~pipeline()

結果

パイプラインからすべてのフィルターを削除してパイプラインを破棄します。

3.8.3 void add_filter(filter& f)

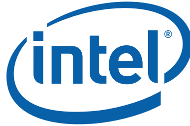
結果

パイプラインのフィルターのシーケンスにフィルター *f* を付加します。フィルター *f* は、まだパイプラインにあってはなりません。

3.8.4 void run(size_t max_number_of_live_tokens)

結果

最初のフィルターが NULL を返し、後のフィルターがそれぞれ、その前からの項目をすべて処理するまで、パイプラインを実行します。並列に処理される項目の数は、パイプラインの構造と利用可能なスレッド数に依存します。大部分の `max_number_of_live_tokens` は、指定された時間で達することはありません。



3.8.5 void clear()

結果

パイプラインからすべてのフィルターを削除します。

3.8.6 filter クラス

概要

パイプライン中のフィルターを表す抽象的な基本クラス。

構文

```
class filter;
```

ヘッダー

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

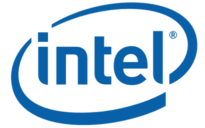
説明

filter は、pipeline (3.8) 中のフィルターを表します。フィルターは並列または直列です。並列フィルターは、並列にアウトオブオーダーで複数の項目を処理できます。直列フィルターは、オリジナルのストリーム順で一度に 1 つのみ項目を処理します。並列の速度向上が見込めるため、通常は並列フィルターを使用します。フィルターが直列なのか並列なのかは、コンストラクターの引数で指定されます。

filter クラスは、pipeline クラス (3.8) とのみ、ともに使用してください。

メンバー

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



例

チュートリアル (doc/Tutorial.pdf) のフィルターの例 MyInputFilter、MyTransformFilter、および MyOutputFilter を参照してください。

3.8.6.1 filter(bool is_serial)

結果

is_serial が true の場合、直列フィルターを構築します。is_serial が false の場合、並列フィルターを構築します。

3.8.6.2 ~filter()

結果

フィルターを破棄します。フィルターは pipeline にあってはなりません。ある場合は、メモリーエラーになります。ライブラリーのデバッグバージョンは、フィルターが pipeline にある場合、アサーション・エラーになります。常に pipeline を含むフィルターを最初に消去または破棄してください。ポイントは、pipeline はフィルターのコンテナーのように働き、C++ コンテナーは項目がコンテナーにある間は項目を破棄することを通常は許可しないということです。

3.8.6.3 bool is_serial() const

リターン

フィルターが直列の場合は true。フィルターが並列の場合は false。

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

結果

派生したフィルターは、項目を処理して、次の filter によって処理される項目へのポインターを返すようにこのメソッドをオーバーライドします。パイプラインの最初のフィルターの項目パラメーターは NULL です。

リターン

処理する項目がない場合、pipeline の最初のフィルターは NULL を返します。pipeline の最後のフィルターの結果は無視されます。

3.9 parallel_sort<RandomAccessIterator, Compare> テンプレート関数

概要

シーケンスをソートします。

構文

```
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 );
```

ヘッダー

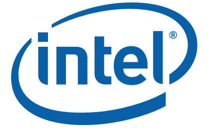
```
#include "tbb/parallel_sort.h"
```

説明

シーケンス [*begin1*, *end1*) の *unstable* ソートを行います。unstable ソートは、等しいキーの相対的な順序を保存しません。ソートは決定性です。同じシーケンスのソートは毎回同じ結果になります。イテレーターとシーケンスの要件は、std::sort と同じです。具体的には、RandomAccessIterator はランダムアクセス・イテレーターであり、その値の型 *T* は表 12 の要件をモデルにします。

表 12: RandomAccessIterator の値の型 T の要件

擬似署名	セマンティクス
`void swap( T& x, T& y )`	x と y を交換します。
`bool Compare::operator()( const T& x, const T& y )`	x が y の前に現れる場合は true。その他の場合は false。



呼び出し parallel_sort(i,j,comp) は、相対的な順序を決定するために 2 番目の引数 comp を使用して、シーケンス [i,j) をソートします。comp(x,y) が true を返す場合、x はソートされたシーケンスで y の前に現れます。

parallel_sort(i,j) 呼び出しは、parallel_sort(i,j,std::less<T>) と等価です。

計算量

parallel_sort は、平均時間計算量 O(N log (N)) の比較ソートです。ここで、N はシーケンスの要素の数です。ワーカースレッドが利用可能な場合 (8.2.1)、parallel_sort は実行時間が改善されるように、同時に実行するサブタスクを作成します。

例

次の例は、2 つのソートを示しています。配列 a のソートは、昇順にソートするデフォルト比較を使用します。配列 b のソートは、比較に std::greater<float> を使用して降順にソートします。

```
#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 コンテナー

コンテナークラスは、複数のスレッドが同じコンテナーの特定のメソッドを同時に呼び出すことを可能にします。

STL と異なり、インテル® スレッディング・ビルディング・ブロックのコンテナーは `allocator` 引数に関してテンプレート化されません。ライブラリーは、メモリー割り当てに対する制御を保存します。

4.1 concurrent_hash_map<Key,T, HashCompare> テンプレート・クラス

概要

同時アクセスを含む連想コンテナー用のテンプレート・クラス。

構文

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

ヘッダー

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

説明

`concurrent_hash_map` は、複数のスレッドが同時に値にアクセスすることを許可する方法で、キーを値にマップします。キーは順序付けされません。インターフェイスは、典型的な STL の連想コンテナーに似ていますが、同時アクセスのサポートで重大ないくつかの相違点があります。

`Key` 型および `T` 型は、CopyConstructible コンセプト (2.2.3) をモデルにしなければなりません。

`HashCompare` 型は、同等にするため、キーがどのようにハッシュされ比較されるか指定します。表 13 の HashCompare コンセプトをモデルにしなければなりません。

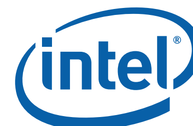


表 13: HashCompare コンセプト

擬似署名	セマンティクス
`HashCompare::HashCompare( const HashCompare & )`	コピー・コンストラクター
`HashCompare::~HashCompare ()`	デストラクター
`bool HashCompare::equal( const Key& j, const Key& k ) const`	キーが等しい場合は true
`size_t HashCompare::hash( const Key& k )`	キーのハッシュコード

注意: ほとんどのハッシュテーブルでは、2 つのキーが等しい場合、同じハッシュコードにハッシュしなければなりません。つまり、指定された HashCompare `h` と任意の 2 つのキー `j` および `k` について、次のアサーションは "`!h.equal(j,k) || h.hash(j)==h.hash(k)`" を保持しなければなりません。このプロパティーの重要性は、`concurrent_hash_map` がキーを同等にして、ハッシュが別々のオブジェクトの代わりに単一オブジェクトでともに移動するという点にあります。

メンバー

```
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 テーブル全体の操作

これらの操作はテーブル全体に影響します。同じテーブルでこれらの操作を同時に呼び出さないでください。

4.1.1.1 concurrent_hash_map()

結果

空のテーブルを構築します。

4.1.1.2 concurrent_hash_map(const concurrent_hash_map& table)

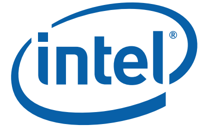
結果

テーブルをコピーします。コピーするテーブルには、その上で同時に実行するマップ操作が含まれます。

4.1.1.3 ~concurrent_hash_map()

結果

テーブルからすべての項目を削除して破棄します。このメソッドを、同じ concurrent_hash_map のほかのメソッドと同時に実行することは安全ではありません。



4.1.1.4 concurrent_hash_map& operator= (concurrent_hash_map& source)

結果

ソースおよびデスティネーション (`this`) テーブルが別の場合、デスティネーション・テーブルを消去して、ソーステーブルからデスティネーション・テーブルにすべてのキー/値ペアをコピーします。その他の場合は、何もしません。

リターン

デスティネーション・テーブルへの参照。

4.1.1.5 void clear()

結果

テーブルからすべてのキー/値ペアを消去します。

4.1.2 同時アクセス

`const_accessor` メンバークラスと `accessor` メンバークラスは、*アクセサー*と呼ばれます。アクセサーは、複数のスレッドが共有の `concurrent_hash_map` のペアに同時アクセスを許可します。アクセサーは、`concurrent_hash_map` のペアへのスマートポインターとして動作します。インスタンスが破棄されるか、`release` メソッドがアクセサーで呼び出されるまで、ペアの暗黙的なロックを保持します。

`const_accessor` クラスと `accessor` クラスは、許可するアクセスの種類において異なります。

表 14: const_accessor と accessor の違い

クラス	value_type	pair の暗黙的なロック
const_accessor	const std::pair<const Key,T>	リーダーロック - ほかのリーダーで共有アクセスを許可します。
accessor	std::pair<const Key,T>	ライターロック - ほかのスレッドによるアクセスをブロックします。

割り当てまたはコピー構築を許可するとロック・セマンティクスが非常に複雑になるため、アクセサーは割り当てまたはコピー構築できません。

4.1.2.1 const_accessor

概要

concurrent_hash_map のキー/値ペアへの読み取り専用アクセスを提供します。

構文

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

ヘッダー

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

説明

const_accessor は、concurrent_hash_map のキー/値ペアへの読み取り専用アクセスを許可します。

メンバー

```
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

リターン

インスタンスが何も指さない場合は ture。インスタンスがキー/値ペアを指す場合は false。

4.1.2.1.2 void release()

結果

`!empty()` の場合、ペアの暗黙的なロックをリリースして、インスタンスが何も指さないように設定します。その他の場合は、何もしません。

4.1.2.1.3 const value_type& operator*() const

結果

empty() と TBB_DO_ASSERT (2.6.1) が非ゼロとして定義された場合、アサーション・エラーが発生します。

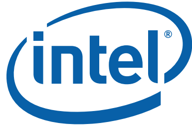
リターン

キー/値ペアへの const 参照。

4.1.2.1.4 const value_type* operator->() const

リターン

```
&operator*()
```



4.1.2.1.5 const_accessor()

結果

何も指さない const_accessor を構築します。

4.1.2.1.6 ~const_accessor

結果

キー/値ペアを指す場合は、ペアの暗黙的なロックを解除します。

4.1.2.2 accessor

概要

concurrent_hash_map のペアへの読み取り/書き込みアクセスを提供するクラス。

構文

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

ヘッダー

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

説明

accessor は、concurrent_hash_map のキー/値ペアへの読み取り/書き込みアクセスを許可します。const_accessor からの派生なので、const_accessor に暗黙的にキャストできます。

メンバー

```
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

結果

`empty()` と `TBB_DO_ASSERT` (2.6.1) が非ゼロとして定義された場合、アサーション・エラーが発生します。

リターン

キー/値ペアへの参照。

4.1.2.2.2 value_type* operator->() const

リターン

```
&operator*()
```

4.1.3 並列操作

`find` 操作、`insert` 操作、`erase` 操作のみ、同じ `concurrent_hash_map` で同時に呼び出すことができます。これらの操作は指定されたキーと一致する値/キーペアについてテーブルを検索します。find メソッドと insert メソッドはそれぞれ、2 つの可変要素を持ちます。1 つは、const_accessor 引数を使用して目的のキー/値ペアへの読み取りアクセスを提供します。もう 1 つは、accessor 引数を使用して書き込みアクセスを提供します。

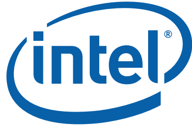
ヒント: キーの検索で `nonconst` 可変要素が成功すると、その結果生じる書き込みアクセスは、アクセサー・オブジェクトが破棄されるまで、ほかのスレッドがキーをアクセスしないようにブロックします。可能であれば、並列化を向上するために const 可変要素を使用してください。

操作に成功した場合、マップ操作の結果は true です。

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

結果

指定されたキーを含むペアがあるかテーブルを検索します。キーが見つかった場合、一致するペアへの読み取り専用アクセスを提供するように結果を設定します。



リターン

キーが見つかった場合は ture。キーが見つからなかった場合は false。

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

結果

指定されたキーを含むペアがあるかテーブルを検索します。キーが見つかった場合、一致するペアへの書き込みアクセスを提供するように結果を設定します。

リターン

キーが見つかった場合は ture。キーが見つからなかった場合は false。

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

結果

指定されたキーを含むペアがあるかテーブルを検索します。ペアがない場合、テーブルに新しいペアを挿入します。新しいペアは `pair(key,T())` で初期化されます。一致するペアへの読み取り専用アクセスを提供するように結果を設定します。

リターン

新しいペアが挿入された場合は true。キーがすでにマップにある場合は false。

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

結果

指定されたキーを含むペアがあるかテーブルを検索します。ペアがない場合、テーブルに新しいペアを挿入します。新しいペアは `pair(key,T())` で初期化されます。一致するペアへの書き込みアクセスを提供するように結果を設定します。

リターン

新しいペアが挿入された場合は true。キーがすでにマップにある場合は false。

4.1.3.5 bool erase(const Key& key)

結果

指定されたキーを含むペアがあるかテーブルを検索します。ペアがある場合、一致するペアを削除します。

リターン

ペアが削除された場合は true。キーがマップになかった場合は false。

4.1.4 並列反復

`const_range_type` 型と `range_type` 型は Range コンセプト (3.2) をモデルにして、表 15 に示す範囲の境界にアクセスするメソッドを提供します。型は、const_range_type の境界が const_iterator 型で、range_type の境界が iterator 型である点のみ異なります。

`concurrent_hash_map` のペアを反復する `parallel_for` (3.4)、`parallel_reduce` (3.5)、および `parallel_scan` (3.6) と関連する範囲型を使用してください。

表 15: concurrent_hash_map **Range** R **のコンセプト (表 4 以外)**

擬似署名	セマンティクス
`R::iterator R::begin() const`	範囲の最初の項目
`R::iterator R::end() const`	範囲の過去の最後の項目

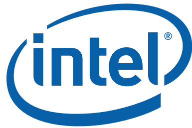
4.1.4.1 const_range_type range(size_t grainsize) const

結果

テーブルのすべてのキーを表す const_range_type を構築します。`grainsize` パラメーターは、ハッシュ・テーブル・スロットのユニット数です。各スロットには通常、平均にほぼ 1 つのキー/値ペアが含まれます。

リターン

テーブルの `const_range_type` オブジェクト。



4.1.4.2 range_type range(size_t grainsize)

リターン

テーブルの `range_type` オブジェクト。

4.1.5 容量

4.1.5.1 size_type size() const

リターン

テーブルのキー/値ペアの数。

メモ: このメソッドは大部分の STL コンテナー用よりも時間がかかります。

4.1.5.2 bool empty() const

リターン

```
size()==0。
```

メモ: このメソッドは大部分の STL コンテナー用よりも時間がかかります。

4.1.5.3 size_type max_size() const

リターン

テーブルが保持できるキー/値ペアの数の包含的な上限。

4.1.6 イテレーター

`concurrent_hash_map` テンプレート・クラスは、前方イテレーター (テーブルを前方にのみ進むことができるイテレーター) をサポートします。逆イテレーターはサポートされません。

4.1.6.1 iterator begin()

リターン

キー/値シーケンスの最初を指す `iterator`。

4.1.6.2 iterator end()

リターン

キー/値シーケンスの最後を指す `iterator`。

4.1.6.3 const_iterator begin() const

リターン

キー/値シーケンスの最初を指す `const_iterator`。

4.1.6.4 const_iterator end() const

リターン

キー/値シーケンスの最後を指す `const_iterator`。

4.2 concurrent_queue<T> テンプレート・クラス

概要

同時アクセスを含むキュー用のテンプレート・クラス。

構文

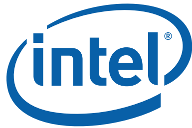
```
template<typename T> class concurrent_queue;
```

ヘッダー

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

説明

`concurrent_queue` は、複数のスレッドが項目のプッシュとポップを同時に行うことができる境界付きの FIFO (先入れ先出し) データ構造です。デフォルトの境界はキューを実際に無制限にできるほど十分な大きさで、対象のマシン上のメモリー制限に従います。



concurrent_queue は並列操作用に設計されているため、インターフェイスは STL `std::queue` 用のものとは異なります。

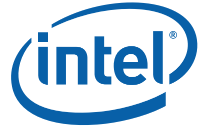
表 16: STL のキューとインテル® スレッディング・ビルディング・ブロックの concurrent_queue の違い

機能	STL `std::queue`	`concurrent_queue`
front および back へのアクセス	`front` メソッドと `back` メソッド	提供されません。並列操作が行われている間は安全ではありません。
`size_type`	符号なし整数型	*符号付き* 整数型
`size()`	キューの項目の数を返します。	プッシュの数からポップの数を引いた数を返します。待機中のプッシュまたはポップ操作は差に含まれます。対応するプッシュを待っているポップがある場合、`size()` は負です。
キュー `q` から項目をコピーしてポップする	`x=q.front(); q.pop()`	*q*.`pop(x)`
キュー `q` が空でなければ、項目をコピーしてポップする	`bool b=!q.empty(); if(b) { x=q.front(); q.pop(); }`	bool b = *q*.`pop_if_present(x)`
空のキューのポップ	許可されていません。	項目が利用可能になるまで待ちます。

注意: プッシュまたはポップ操作をブロックする場合、ユーザー空間ロックを使用してブロックします。このため、ブロック時間が長い場合、プロセッサーのリソースが浪費されます。`concurrent_queue` クラスは、ブロック時間がアプリケーション時間の残りに比べて短い状況用に設計されています。

メンバー

```
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()

結果

空のキューを構築します。

4.2.2 ~concurrent_queue()

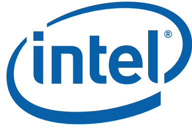
結果

キューのすべての項目を破棄します。

4.2.3 void push(const T& source)

結果

size()<capacity になるまで待ってから、`source` のコピーをキューの後ろにプッシュします。



4.2.4 void pop(T& destination)

結果

値が利用可能になるまで待ち、値をキューからポップします。値を `destination` に代入します。オリジナルの値を破棄します。

4.2.5 bool pop_if_present(T& destination)

結果

値が利用可能な場合、値をキューからポップして、値を destination に代入し、オリジナルの値を破棄します。その他の場合は、何もしません。

リターン

値がポップされた場合は true。その他の場合は false。

4.2.6 size_type size() const

リターン

プッシュの数からポップの数を引いた数。対応するプッシュを待っているポップ操作がある場合、結果は負です。

4.2.7 bool empty() const

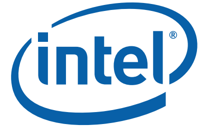
リターン

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

4.2.8 size_type capacity() const

リターン

キューが保持できる値の最大数。



4.2.9 void set_capacity(size_type capacity)

結果

キューが保持できる値の最大数を設定します。

4.2.10 イテレーター

concurrent_queue は、単独でプログラマーがデバッグ中にキューを検査できるように制限付きのイテレーター・サポートを提供します。iterator 型および const_iterator 型を提供します。どちらも、前方イテレーター用の通常の STL 規則に従います。反復順は、以前プッシュされたものから最近プッシュされたものの順になります。修正すると、concurrent_queue を参照するすべてのイテレーターが無効になります。

注意: イテレーターは比較的遅いため、デバッグのみに使用してください。

例

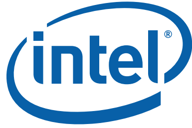
次のプログラムは、整数 0..9 でキューを構築した後、キューを標準出力にダンプします。その結果、0 1 2 3 4 5 6 7 8 9 が印刷されます。

```
#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()

リターン

キューの最初を指す iterator。



4.2.10.2 iterator end()

リターン

キューの最後を指す iterator。

4.2.10.3 const_iterator begin() const

リターン

キューの最後を指す const_iterator。

4.2.10.4 const_iterator end() const

リターン

キューの最後を指す const_iterator。

4.3 concurrent_vector

概要

同時に拡張してアクセスできるベクトル用のテンプレート・クラス。

構文

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

ヘッダー

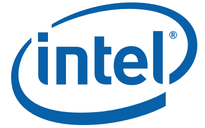
```
#include "tbb/concurrent_vector.h"
```

説明

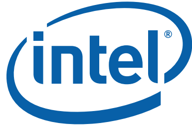
concurrent_vector は、拡張可能な 動的配列で、拡張しているときにベクトルの要素に同時に安全にアクセスできます。最初の要素のインデックスは 0 です。

メンバー

```
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 ベクトル全体の操作

これらの操作は同じインスタンス上ではスレッドセーフでは*ありません*。

4.3.1.1 concurrent_vector()

結果

空のベクトルを構築します。

4.3.1.2 concurrent_vector(const concurrent_vector& src)

結果

`src` のコピーを構築します。

4.3.1.3 concurrent_vector& operator=(const concurrent_vector& src)

結果

src の内容を *this に代入します。

リターン

左辺への参照。

4.3.1.4 ~concurrent_vector()

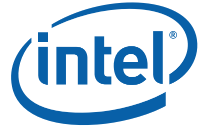
結果

すべての要素を消去してベクトルを破棄します。

4.3.1.5 void clear()

結果

すべての要素を消去します。その後、`size()==0` にします。



4.3.2 並列操作

このセクションで説明されているメソッドは、`concurrent_vector<T>` の同じインスタンス上で安全に実行されます。

4.3.2.1 size_type grow_by(size_type delta)

結果

ベクトルの最後に *delta* 要素を自動的に付加します。新しい要素は T() で初期化されます。ここで、T はベクトルの `value_type` です。

リターン

ベクトルの古いサイズ。*k* を返す場合、新しい要素は半開インデックス範囲 (*k*..*k*+*delta*) にあります。

4.3.2.2 void grow_to_at_least(size_type n)

結果

少なくとも *n* 要素になるまでベクトルを拡張させます。新しい要素は `T()` で初期化されます。ここで、`T` はベクトルの `value_type` です。

4.3.2.3 size_t push_back(const_reference value);

結果

ベクトルの最後に *value* のコピーを自動的に付加します。

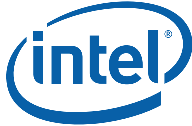
リターン

コピーのインデックス。

4.3.2.4 reference operator[](size_type index)

リターン

指定したインデックスの要素への参照。



4.3.2.5 const_reference operator[](size_type index) const;

リターン

指定したインデックスの要素への const 参照。

4.3.3 並列反復

const_range_type 型と range_type 型は、Range コンセプト (3.2) をモデルにして、表 15 に示す範囲の境界にアクセスするメソッドを提供します。型は、const_range_type の境界が const_iterator 型で、range_type の境界が iterator 型である点のみ異なります。

concurrent_vector のペアを反復する parallel_for (3.4)、parallel_reduce (3.5)、および parallel_scan (3.6) と関連する範囲型を使用してください。

表 17: concurrent_vector Range R のコンセプト

擬似署名	セマンティクス
`R::iterator R::begin() const`	範囲の最初の項目
`R::iterator R::end() const`	範囲の過去の最後の項目

4.3.3.1 range_type range(size_t grainsize)

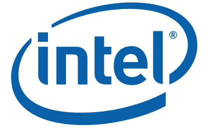
リターン

読み取り/書き込みアクセスを許可する concurrent_vector 全体の範囲。

4.3.3.2 const_range_type range(size_t grainsize) const

リターン

読み取り専用アクセスを許可する concurrent_vector 全体の範囲。



4.3.4 容量

4.3.4.1 size_type size() const

リターン

ベクトルの要素の数。結果には、`grow_by` メソッド (4.3.2.1) または `grow_to_at_least` (4.3.2.2) への同時呼び出しによって構築中の要素も含まれます。

4.3.4.2 bool empty() const

リターン

```
size()==0。
```

4.3.4.3 size_type capacity() const

リターン

ベクトルがより多くのメモリーを割り当てることなく拡張できる最大サイズ。

メモ: STL ベクトルと異なり、`concurrent_vector` は、より多くのメモリーを割り当てる必要がある場合に既存の要素を移動しません。

4.3.4.4 void reserve(size_type n)

リターン

少なくとも *n* 要素の予約空間。

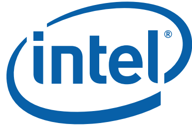
スロー

`n>max_size()` の場合、`std::length_error`。

4.3.4.5 size_type max_size() const

リターン

表現可能な最高サイズのベクトル。



4.3.5 イテレーター

concurrent_vector<T> テンプレート・クラスは、ISO C++ 標準のセクション 24.1.4 で定義されているランダムアクセス・イテレーターをサポートします。std::vector とは異なり、イテレーターは生のポインターではありません。concurrent_vector<T> は、ISO C++ 標準の表 66 の可逆コンテナーの要件を満たします。

4.3.5.1 iterator begin()

リターン

ベクトルの最初を指す iterator。

4.3.5.2 iterator end()

リターン

ベクトルの最後を指す iterator。

4.3.5.3 const_iterator begin() const

リターン

ベクトルの最初を指す const_iterator。

4.3.5.4 const_iterator end() const

リターン

キューの最後を指す const_iterator。

4.3.5.5 iterator rbegin()

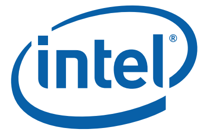
リターン

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

4.3.5.6 iterator rend()

リターン

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



4.3.5.7 const_reverse_iterator rbegin() const

リターン

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

4.3.5.8 const_ reverse_iterator rend() const

リターン

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

5 メモリー割り当て

このセクションでは、メモリー割り当てに関連するクラスを説明します。

5.1 Allocator コンセプト

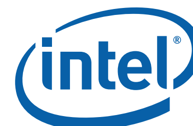
インテル® スレッディング・ビルディング・ブロックのアロケーターの Allocator コンセプトは ISO C++ 標準の表 32 の "Allocator 要件" に似ていますが、ISO C++ コンテナーの使用には ISO C++ 標準 (セクション 20.1.5、段落 4) のさらなる保証が必要です。表 18 は、Allocator コンセプトを要約したものです。ここで、A と B はアロケーター・クラスのインスタンスを表します。

表 18: Allocator コンセプト

擬似署名	セマンティクス
`typedef T* A::pointer`	*T* へのポインター
`typedef const T* A::const_pointer`	const *T* へのポインター
`typedef T& A::reference`	*T* への参照
`typedef const T& A::const_reference`	const *T* への参照
`typedef T A::value_type`	割り当てる値の型
`typedef size_t A::size_type`	値の数を表す型
`typedef ptrdiff_t A::difference_type`	ポインターの差を表す型

`template<typename U> struct rebind {`
`    typedef A<U> A::other;`
`};`
	異なる型 *U* に再バインドします。

`A() throw()`	デフォルト・コンストラクター
`A( const A& ) throw()`	コピー・コンストラクター
`template<typename U> A( const A& )`	再バインド・コンストラクター
`~A() throw()`	デストラクター
`T* A::address( T& x ) const`	アドレスを取得します。
`const T* A::const_address( const T& x )`	const アドレスを取得します。



const	
T* A::allocate(size_type n, void* hint=0)	n 値用の空間を割り当てます。
void A::deallocate(T* p, size_t n)	n 値の割り当てを解除します。
size_type A::max_size() const throw()	allocate メソッドに対する最大限の妥当な引数
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&)	true を返します。
bool operator!=(const A&, const B&)	false を返します。

モデル型

scalable_allocator テンプレート・クラス (5.2) と cached_aligned_allocator テンプレート・クラス (5.3) は、Allocator コンセプトをモデルにします。

5.2 scalable_allocator<T> テンプレート・クラス

概要

スケーラブルなメモリー割り当て用のテンプレート・クラス。

構文

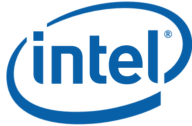
```
template<typename T> class scalable_allocator;
```

ヘッダー

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

説明

scalable_allocator は、プロセッサーの数でスケールする方法でメモリーの割り当てと解放を行います。scalable_allocator は、表 18 で説明されている allocator 要件をモデルにします。std::allocator の代わりに scalable_allocator を使用すると、プログラムのパフォーマ



ンスが向上する場合があります。scalable_allocator によって割り当てられたメモリーは、std::allocator ではなく scalable_allocator で解放してください。

メンバー

Allocator コンセプト (5.1) を参照してください。

謝辞

スケーラブル・メモリー・アロケーターは、インテルの PSL CTG チームによって開発された McRT テクノロジーを採用しています。

5.2.1 スケーラブル・アロケーターの C インターフェイス

概要

スケーラブルなメモリー割り当て用の低水準インターフェイス。

構文

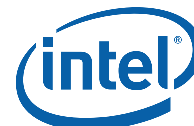
```
extern "C" {
    void* scalable_calloc ( size_t nobj, size_t size );
    void  scalable_free( void* ptr );
    void* scalable_malloc( size_t size );
    void* scalable_realloc( void* ptr, size_t size );
}
```

ヘッダー

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

説明

これらの関数は、スケーラブルなアロケーターに C 水準のインターフェイスを提供します。scalable_*x* ルーチンはそれぞれ、C 標準ライブラリー関数 *x* に似た動作を行います。scalable_*x* 関数によって割り当てられた記憶領域は、C 標準ライブラリー関数ではなく、scalable_*x* 関数によって解放またはサイズ変更してください。同様に、C 標準ライブラリー関数によって割り当てられた記憶領域は、scalable_*x* 関数によって解放またはサイズ変更しないでください。



5.3 cache_aligned_allocator<T> テンプレート・クラス

概要

フォールス・シェアリングを回避する方法でのメモリー割り当て用のテンプレート・クラス。

構文

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

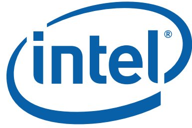
ヘッダー

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

説明

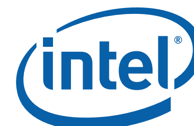
`cache_aligned_allocator` は、フォールス・シェアリングを回避するため、キャッシュライン境界上にメモリーを割り当てます。フォールス・シェアリングは、論理的に別の項目が同じキャッシュラインにアクセスするときに発生します。複数のスレッドが異なる項目に同時にアクセスすると、パフォーマンスが低下します。項目が論理的には別であっても、プロセッサーのハードウェアはそれらの項目が場所を共有しているようにプロセッサー間のキャッシュラインを転送します。最終的には、論理的に別の項目が異なるキャッシュライン上にある場合よりも、さらに多くのメモリー・トラフィックが発生することがあります。

`cache_aligned_allocator` は、表 18 で説明されている allocator 要件をモデルにします。`std::allocator` を置換するために使用できます。注意して `cache_aligned_allocator` を使用すると、フォールス・シェアリングが減り、パフォーマンスが向上します。しかし、キャッシュライン上の割り当ての恩恵は `cache_aligned_allocator` が暗黙的に追加するパディング メモリーの代償によるものなので、置換が適切でない場合もあります。パディングは通常 128 バイトです。`cache_aligned_allocator` では多くの小さなオブジェクトが割り当てられるため、メモリーの使用量は増加します。



メンバー

```
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>& );

}
```

簡潔に説明するため、次のサブセクションでは `std::allocator` の対応するメソッドと大幅に異なるメソッドについてのみ説明します。

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

結果

キャッシュライン境界上で *size* バイトのメモリーを割り当てます。割り当てには追加の隠しパディングが含まれます。

リターン

割り当てたメモリーへのポインター。

5.3.2 void deallocate(pointer p, size_type n)

要件

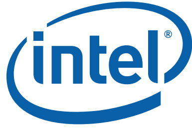
ポインター *p* が、`allocate(n)` メソッドの結果であること。メモリーがすでに割り当て解除されていないこと。

結果

p が指しているメモリーの割り当てを解除します。また、追加の隠しパディングの割り当ても解除します。

5.3.3 char* _Charalloc(size_type size)

メモ: このメソッドは 64 ビット Windows* プラットフォーム上でのみ提供されます。このメソッドは、このメソッドが必要な Windows のコンテナーのバージョンとの後方互換性のために存在する非 ISO メソッドです。直接使用しないでください。



5.4 aligned_space テンプレート・クラス

概要

初期化されていないメモリー空間。

構文

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

ヘッダー

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

説明

aligned_space は、配列 *T[N]* を保持するために十分なメモリーを占有します。クライアントは、オブジェクトの初期化や破棄に関する責任を負います。aligned_space は、通常、固定長の初期化されていないメモリーのブロックが必要なシナリオで、ローカル変数またはフィールドとして使用されます。

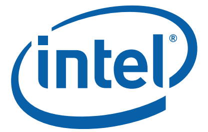
メンバー

```
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()

結果

なし。コンストラクターを呼び出しません。



5.4.2 ~aligned_space()

結果

なし。デストラクターを呼び出しません。

5.4.3 T* begin()

リターン

記憶領域の最初へのポインター。

5.4.4 T* end()

リターン

```
begin()+N
```

6 同期

ライブラリーは、排他制御とアトミック操作をサポートします。

6.1 Mutex

Mutex は、コードのセクションからのスレッドの排他制御 (MUTual EXclusion) を行います。

一般に、明示的なロックの使用は直列のボトルネックとなるため、使用を最小限に抑えるように設計してください。明示的なロックが必要な場合、複数のスレッドが同じ mutex のロックで競合しないように、ロックを広げてください。

6.1.1 Mutex コンセプト

ここの mutex とロックのインターフェイスは、ハイ・パフォーマンス用に設計された比較的厳格なものです。インターフェイスは、次の理由により、C++ ライブラリーで広く使用されている*スコープ・ロック・パターン*を使用します。

1. プログラマーがロックをリリースすることを覚えておく必要がない。
2. 例外がロックで保護されている排他制御領域からスローされた場合、ロックをリリースする。

パターン (*mutex* オブジェクト) には、mutex のロックを取得する *lock* オブジェクトの構築とロックをリリースする *lock* オブジェクトの破棄の 2 つの部分があります。次に例を示します。

```
{
    // 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
}
```

アクションが例外をスローすると、ロックは自動的にリリースされ、ブロックは終了します。

表 19 は、mutex 型 M の Mutex コンセプトの要件を示しています。

表 19: Mutex コンセプト

擬似署名	セマンティクス
`M()`	ロックしていない mutex を構築します。
`~M()`	ロックしていない mutex を破棄します。
`typename M::scoped_lock`	対応するスコープロックの種類
`M::scoped_lock()`	mutex を取得しないでロックを構築します。
`M::scoped_lock(M&)`	ロックを構築して mutex のロックを取得します。
`M::~scoped_lock()`	ロックをリリースします (取得している場合)。
`M::scoped_lock::acquire(M&)`	mutex のロックを取得します。
`bool M::scoped_lock::try_acquire(M&)`	mutex のロックを取得しようとします。ロックを取得した場合は true、取得しなかった場合は false を返します。
`M::scoped_lock::release()`	ロックをリリースします。

表 20 は、Mutex コンセプトをモデルにするクラスを要約したものです。

表 20: Mutex コンセプトをモデルにする mutex

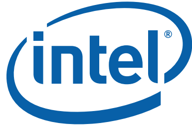
	スケーラブル	フェア	リエントラント	スリープ	サイズ
`mutex`	OS 依存	OS 依存	×	○	≥ 3 ワード
`spin_mutex`	×	×	×	×	1 バイト
`queuing_mutex`	✓	✓	×	×	1 ワード
`spin_rw_mutex`	×	×	×	×	1 ワード
`queuing_rw_mutex`	✓	✓	×	×	1 ワード

mutex プロパティーの説明は、チュートリアルを参照してください。

6.1.2 mutex クラス

概要

根本的な OS ロックを使用して Mutex コンセプトをモデルにするクラス。



構文

```
class mutex;
```

ヘッダー

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

説明

`mutex` は、Mutex コンセプト (6.1.1) をモデルにします。mutex は、排他制御を行う OS 呼び出しのラッパーです。OS 呼び出しの代わりに `mutex` を使用する利点は次のとおりです。

- インテル® スレッディング・ビルディング・ブロックでサポートされているすべてのオペレーティング・システムにわたって可搬性がある。
- 例外がコードの保護領域からスローされた場合、ロックをリリースする。

メンバー

Mutex コンセプト (6.1.1) を参照してください。

6.1.3 spin_mutex クラス

概要

スピンロックを使用して Mutex コンセプトをモデルにするクラス。

構文

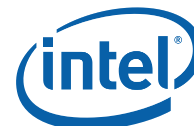
```
class spin_mutex;
```

ヘッダー

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

説明

`spin_mutex` は、Mutex コンセプト (6.1.1) をモデルにします。`spin_mutex` は、スケーラブル、フェア、またはリエントラントではありません。ロックがわずかに競合していて、少数の機械語命令のために保持されている場合に理想的です。スレッドが `spin_mutex` の取得を待つ必要がある場合、ビジーウェイトを行うため、ウェイトが長くなるとシステムのパフォーマンスに影響し



ます。しかし、ウェイトが短い場合、`spin_mutex` は、ほかの mutex よりも大幅にパフォーマンスを向上させます。

メンバー

Mutex コンセプト (6.1.1) を参照してください。

6.1.4 queuing_mutex クラス

概要

フェアで、スケーラブルな Mutex コンセプトをモデルにするクラス。

構文

```
class queuing_mutex;
```

ヘッダー

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

説明

`queuing_mutex` は、Mutex コンセプト (6.1.1) をモデルにします。スレッドが mutex の取得を待つ必要がある場合、自身のローカル・キャッシュ・ライン上でスピンするという意味で、`queuing_mutex` はスケーラブルです。`queuing_mutex` は、フェアです。スレッドは、要求する順に queuing_mutex のロックを取得します。`queuing_mutex` は、リエントラントではありません。

現在の実装はビジーウェイトを行うため、ウェイトが長い場合、`queuing_mutex` の使用はシステムのパフォーマンスに影響します。

メンバー

Mutex コンセプト (6.1.1) を参照してください。

6.1.5 ReaderWriterMutex コンセプト

ReaderWriterMutex コンセプトは、リーダーロックとライターロックの概念を含むように Mutex コンセプトを拡張します。このコンセプトは、ライターロック (`write` =true) とリーダーロック

(`write` =false) が要求されているかどうかを指定するブール・パラメーター `write` を採用しています。ライターロックがない場合、複数のリーダーロックを ReaderWriterMutex で同時に保持することができます。ReaderWriterMutex のライターロックは、mutex のロックを同時に保持することからほかのすべてのロックを除外します。

表 21 は、ReaderWriterMutex `RW` の要件を示しています。

表 21: ReaderWriterMutex コンセプト

擬似署名	セマンティクス
`RW()`	ロックしていない mutex を構築します。
`~RW()`	ロックしていない mutex を破棄します。
`typename RW::scoped_lock`	対応するスコープロックの種類
`RW::scoped_lock()`	mutex を取得しないでロックを構築します。
`RW::scoped_lock(RW&, bool write=true)`	ロックを構築して mutex のロックを取得します。
`RW::~scoped_lock()`	ロックをリリースします (取得している場合)。
`RW::scoped_lock::acquire(RW&, bool write=true)`	mutex のロックを取得します。
`bool RW::scoped_lock::try_acquire(RW&, bool write=true)`	mutex のロックを取得しようとします。ロックを取得した場合は `true`、取得しなかった場合は `false` を返します。
`RW::scoped_lock::release()`	ロックをリリースします。
`bool RW::scoped_lock::upgrade_to_writer()`	リーダーロックをライターロックに変更します。
`bool RW::scoped_lock::downgrade_to_reader()`	ライターロックをリーダーロックに変更します。

次のサブセクションは、ReaderWriterMutex コンセプトの詳細なセマンティクスを説明しています。

モデル型

`spin_rw_mutex` (6.1.6) および `queuing_rw_mutex` (6.1.7) は、ReaderWriterMutex コンセプトをモデルにします。

6.1.5.1 ReaderWriterMutex()

結果

ロックしていない ReaderWriterMutex を構築します。

6.1.5.2 ~ReaderWriterMutex()

結果

ロックしていない ReaderWriterMutex を破棄します。ロックしている ReaderWriterMutex を破棄した場合の結果は未定義です。

6.1.5.3 ReaderWriterMutex::scoped_lock()

結果

mutex のロックを保持しない scoped_lock オブジェクトを構築します。

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

結果

mutex *rw* のロックを取得する scoped_lock オブジェクトを構築します。ロックは *write* が true の場合はライターロックで、その他の場合はリーダーロックです。

6.1.5.5 ReaderWriterMutex::~scoped_lock()

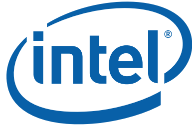
結果

オブジェクトが ReaderWriterMutex のロックを保持している場合、ロックをリリースします。

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

結果

mutex *rw* のロックを取得します。ロックは *write* が true の場合はライターロックで、その他の場合はリーダーロックです。



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

結果

mutex *rw* のロックを取得しようとします。ロックは *write* が true の場合はライターロックで、その他の場合はリーダーロックです。

リターン

ロックを取得した場合は `true`、取得しなかった場合は `false`。

6.1.5.8 void ReaderWriterMutex:: scoped_lock::release()

結果

ロックをリリースします。ロックが保持されていない場合の結果は未定義です。

6.1.5.9 bool ReaderWriterMutex:: scoped_lock::upgrade_to_writer()

結果

リーダーロックをライターロックに変更します。オブジェクトがすでにリーダーロックを保持していない場合の結果は未定義です。

リターン

ロックがリリースまたは再取得された場合は `false`。その他の場合は `true`。

6.1.5.10 bool ReaderWriterMutex:: scoped_lock::downgrade_to_reader()

結果

ライターロックをリーダーロックに変更します。オブジェクトがすでにライターロックを保持していない場合の結果は未定義です。

リターン

ロックがリリースまたは再取得された場合は `false`。その他の場合は `true`。

メモ: インテルの現在の実装では、`spin_rw_mutex` および `queuing_rw_mutex` は常に `true` を返します。異なる実装では、たまに `false` を返します。

6.1.6 spin_rw_mutex クラス

概要

アンフェアで、スケーラブルではない ReaderWriterMutex コンセプトをモデルにするクラス。

構文

```
class spin_rw_mutex;
```

ヘッダー

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

説明

`spin_rw_mutex` は、ReaderWriterMutex コンセプト (6.1.1) をモデルにします。`spin_rw_mutex` は、スケーラブル、フェア、またはリエントラントではありません。ロックがわずかに競合していて、少数の機械語命令のために保持されている場合に理想的です。スレッドが `spin_rw_mutex` の取得を待つ必要がある場合、ビジーウェイトを行うため、ウェイトが長くなるとシステムのパフォーマンスに影響します。しかし、ウェイトが短い場合、`spin_rw_mutex` は、ほかの mutex よりも大幅にパフォーマンスを向上させます。

メンバー

ReaderWriterMutex コンセプト (6.1.5) を参照してください。

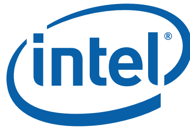
6.1.7 queuing_rw_mutex クラス

概要

フェアで、スケーラブルな ReaderWriterMutex コンセプトをモデルにするクラス。

構文

```
class queuing_rw_mutex;
```



ヘッダー

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

説明

`queuing_rw_mutex` は、ReaderWriterMutex コンセプト (6.1.1) をモデルにします。スレッドが mutex の取得を待つ必要がある場合、自身のローカル・キャッシュ・ライン上でスピンするという意味で、`queuing_rw_mutex` はスケーラブルです。`queuing_rw_mutex` は、フェアです。スレッドは、要求する順に `queuing_rw_mutex` のロックを取得します。`queuing_rw_mutex` は、リエントラントではありません。

メンバー

ReaderWriterMutex コンセプト (6.1.5) を参照してください。

6.2 atomic<T> テンプレート・クラス

概要

アトミック操作用のテンプレート・クラス。

構文

```
template<typename T> atomic;
```

ヘッダー

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

説明

`atomic<T>` は、アトミック read、write、fetch-and-add、fetch-and-store、および compare-and-swap をサポートします。型 T は、整数型またはポインター型です。T がポインター型の場合、算術演算はポインター算術として解釈されます。例えば、*x* に型 atomic<float*> が含まれ float が 4 バイトの場合、++x は 4 バイトずつ x を進めます。特殊な `atomic<void*>` では、ポインター算術は使用できません。

一部のメソッドには、より選択的なメモリーフェンスが可能なテンプレート・メソッド可変要素が含まれます。IA-32 およびインテル® 64 アーキテクチャー対応のプロセッサーでは、非テンプ

レート可変要素と同じ効果になります。Itanium® プロセッサーでは、メモリー・サブシステムに読み取りと書き込み順でより自由な裁量を与えることで、パフォーマンスが向上します。表 22 は、非テンプレート形式のメモリーフェンスを示しています。

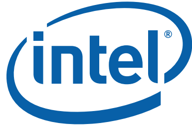
表 22: 非テンプレート・メソッドによって暗黙的に定義されるメモリーフェンス

種類	説明	デフォルト
acquire	フェンスの後の操作は上に移動しません。	read
release	フェンスの前の操作は上に移動しません。	write
full	両側の操作は上に移動しません。	`fetch_and_store`, `fetch_and_add`, `compare_and_swap`

ヒント: 非自明なコンストラクターが含まれていると `atomic<T>` の目的と異なるようにコンパイラーが動作する可能性があるため、`atomic<T>` テンプレート・クラスには非自明なコンストラクターは含まれていません。特定の値で `atomic<T>` を作成するには、デフォルトで構築してから、後で値を代入してください。

メンバー

```
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

説明

より選択的なメモリーフェンス (表 22 を参照) が可能なテンプレート・メソッド可変要素を選択するために使用する値を定義します。

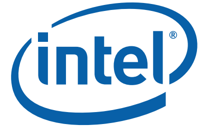
6.2.2 value_type fetch_and_add(value_type addend)

結果

x を `*this` の値にします。*x* = *x* + addend をアトミックに更新します。

リターン

x のオリジナルの値。



6.2.3 value_type fetch_and_increment()

結果

x を `*this` の値にします。$x = x + 1$ をアトミックに更新します。

リターン

x のオリジナルの値。

6.2.4 value_type fetch_and_decrement()

結果

x を `*this` の値にします。$x = x - 1$ をアトミックに更新します。

リターン

x のオリジナルの値。

6.2.5 value_type compare_and_swap

```
value_type compare_and_swap( value_type new_value, value_type comparand )
```

結果

x を `*this` の値にします。*x* と comparand をアトミックに比較して、等しい場合、*x*=new_value に設定します。

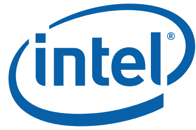
リターン

x のオリジナルの値。

6.2.6 value_type fetch_and_store(value_type new_value)

結果

x を `*this` の値にします。*x* の古い値を new_value でアトミックに交換します。



リターン

x のオリジナルの値。

7 時間計測

並列プログラミングとは、プログラムを実行する実際の時間である*ウォールクロック*時間を高速化することです。残念なことに、オペレーティング・システムによって提供される一部のウォールクロック時間計測ルーチンは、ハードウェア・クロックが同期していないために、必ずしも複数のスレッドにわたって確実に動作するとは限りません。ライブラリーは、複数のスレッドにわたる時間計測をサポートします。ルーチンは、スレッドにわたって使用することが安全であると確認したオペレーティング・サービスのラッパーです。

7.1 tick_count クラス

概要

ウォールクロック時間計算用のクラス。

構文

```
class tick_count;
```

ヘッダー

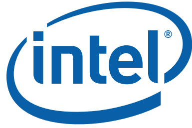
```
#include "tbb/tick_count.h"
```

説明

`tick_count` は、絶対タイムスタンプです。2 つの `tick_count` オブジェクトが、相対時間 `tick_count`::interval_t を計算するために引かれます。この時間は秒に変換することができます。

例

```
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() );
}
```



メンバー

```
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()

リターン

現在のウォールクロック・タイムスタンプ。

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

リターン

t1 が t0 の後に発生した相対時間。

7.1.3 tick_count::interval_t クラス

概要

相対ウォールクロック時間用のクラス。

構文

```
class tick_count::interval_t;
```

ヘッダー

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

説明

tick_count::interval_t は相対ウォールクロック時間または期間を表します。

メンバー

```
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()

結果

ゼロ時間期間を表す interval_t を構築します。

7.1.3.2 double seconds() const

リターン

秒単位で測定された時間間隔。

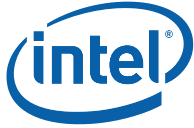
7.1.3.3 interval_t operator+=(const interval_t& i)

結果

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

リターン

*this への参照。



7.1.3.4 interval_t operator−=(const interval_t& i)

結果

```
*this = *this − i
```

リターン

`*this` への参照。

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

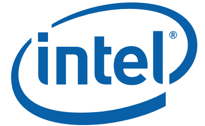
リターン

間隔 *i* と *j* の合計を表す interval_t。

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

リターン

間隔 *i* and *j* の差を表す `interval_t`。



8 タスク・スケジューリング

ライブラリーは、アルゴリズム・テンプレート (3) を作動するエンジンであるタスク・スケジューラーを提供します。直接呼び出すこともできます。タスク・スケジューラーが多くの処理を行うため、タスクの使用は多くの場合スレッドの使用よりも単純で効率的です。

タスクは計算の論理ユニットです。スケジューラーは、物理スレッドにこれらをマップします。マッピングはノン・プリエンプティブです。各スレッドには `execute()` メソッドが含まれます。いったんスレッドが `execute()` の実行を開始すると、`execute()` がリターンするまで、タスクはそのスレッドにバインドされます。その時間の間、スレッドは子タスクを待っている場合のみほかのタスクを処理します。子タスクを実行するか、保留中の子タスクがない場合は、ほかのスレッドによって作成されたサービスタスクを実行します。

タスク・スケジューラーは、計算集約的な作業を並列化するように意図されています。タスク・オブジェクトはプリエンプティブにスケジュールされないため (その間、スレッドはほかのタスクを処理できないため)、タスク・オブジェクトはサービスを長期間ブロックする呼び出しを行ってはなりません。

注意: スケジューラーは利用可能なワーカースレッドに収まるように実際の並列処理を調整するため、*潜在的に* 並列なタスクが *実際に* 並列で実行されるという保証はありません。例えば、単一のワーカースレッドが提供された場合、スケジューラーは実際の並列処理を作成しません。例えば、生産者が実行している間、消費者がそのすべてで実行する保証がないため、生産者消費者関係でタスクを使用することは一般的に安全ではありません。

潜在的な並列処理は通常、分割/連結パターンによって生成されます。分割/連結の 2 つの基本パターンがサポートされています。最も効率的なのは、プログラマーが明示的に ”continuation (継続)” タスクを構築する継続渡し形式です。親は子タスクを分割して、子が完成するときに実行する継続タスクを指定します。継続は親の先祖を継承します。その後、親タスクは終了します。つまり、子をブロックしません。続いて子が実行され、子 (またはその継続) が終了した後、継続タスクが実行を開始します。図 2 は、このステップを示しています。各ステップの実行タスクは色付きで表示されています。

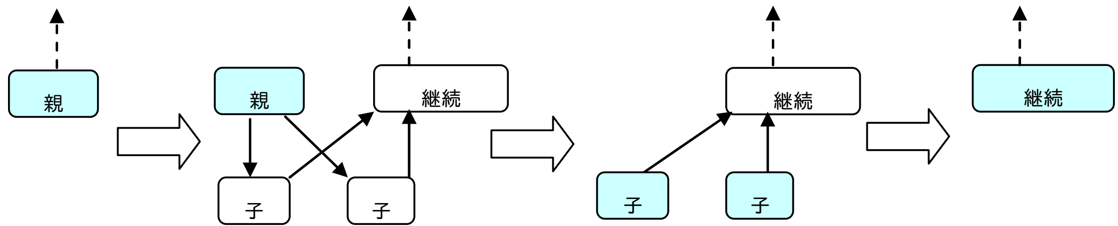
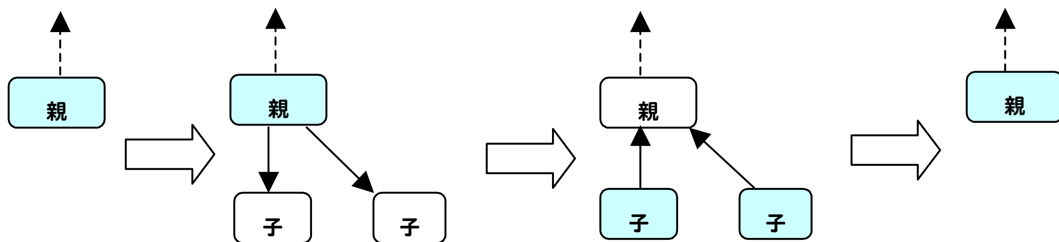


図 2: 継続渡しスタイル

明示的な継続渡しは、タスクからスレッドのスタックを分離するので効率的です。しかし、プログラムはより難しくなります。2 番目のパターンは、暗黙的な継続を使用する "ブロックスタイル" です。このパターンはパフォーマンスの点では効率が落ちますが、プログラムはより簡単です。



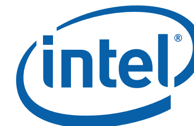
このパターンでは、図 3 に示すように、親タスクはその子が完了するまでブロックします。

図 3: ブロックスタイル

便利さには代償がつきものです。親がブロックしているため、そのスレッドのスタックはまだポップできません。連続したスチールとブロックはスタックを境界なしで拡張させるため、スレッドがどの作業を受け持つかを注意する必要があります。この問題を解決するため、スケジューラーは最も深くブロックされたタスクほど深くないタスクを実行しないブロックされたスレッドを制約します。この制約により、利用可能な並列処理が制限され、スレッドがほかの方法で選択するサブツリーよりも小さな (深い) サブツリーを選択するため、パフォーマンスに影響します。

8.1 スケジューリング・アルゴリズム

スケジューラーは、タスクスチールを使用します。各スレッドは、実行する準備ができているタスクの "レディプール" を管理します。レディプールは、`task` のリストの配列として構築されます。



ここで、*i 番目* の要素のリストがツリーのレベル *i* のタスクに相当します。リストは、LIFO (後入れ先出し) 順に処理されます。レベル *i* のタスクは、レベル *i*+1 で子タスクを生成します。スレッドは、配列で最も深い非空のリストからタスクを取り出します。非空のリストがない場合、スレッドはランダムに別のスレッドの最も浅いリストからタスクをスチールします。また、最後の子が完了した場合も、スレッドは暗黙的にスチールして、子で待機していたタスクの実行を開始します。

タスク・スケジューラーは、参照の局所性、空間効率、および並列処理がうまくバランスされるようにします。スケジューリングの手法は、Cilk (Blumofe 1995) で使用されるものに似ています。

8.2 task_scheduler_init クラス

概要

タスク・スケジューリング・サービスにおけるスレッドの状態を表すクラス。

構文

```
class task_scheduler_init;
```

ヘッダー

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

説明

`task_scheduler_init` は、"アクティブ" または "インアクティブ" のいずれかです。`task` を使用する各スレッドには、スレッドが `task` オブジェクトを使用する間アクティブである、1 つのアクティブな `task_scheduler_init` オブジェクトが必要です。指定した時期に 1 つ以上のアクティブな `task_scheduler_init` オブジェクトがスレッドにある場合もあります。

`task_scheduler_init` のデフォルト・コンストラクターはオブジェクトをアクティベートし、デストラクターはオブジェクトを初期化前の状態に戻します。初期化を遅らせるには、コンストラクターに `task_scheduler_init::deferred` 値を渡します。この場合、`task_scheduler_init` は `initialize` メソッドを呼び出して、後で初期化されます。初期化された `task_scheduler_init` の破棄は、オブジェクトを暗黙的にデアクティベートします。オブジェクトをより早くデアクティベートするには、`terminate` メソッドを呼び出します。

コンストラクターのオプション・パラメーターおよび initialize メソッドで task 実行に使用されるスレッド数を指定することができます。このパラメーターは開発中にスケーリングを調査するために役立ちますが、製品には設定しないでください。詳細は、『チュートリアル』ドキュメントを参照してください。

時間のオーバーヘッドを最小限にするには、スレッドがそのアクティベーション期間のすべてでライブラリーのタスク・スケジューラーを使用する単一の task_scheduler_init オブジェクトを作成することがベストです。task_scheduler_init は、割り当てまたはコピー構築できません。

重要

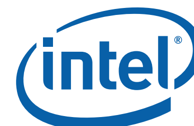
テンプレート・アルゴリズム (3) は、task クラスを暗黙的に使用します。このため、task_scheduler_init を作成することがテンプレート・アルゴリズムを使用するための前提条件です。ライブラリーのデバッグバージョンは、task_scheduler_init の作成に失敗します。

例

```
#include "tbb/task_scheduler_init"

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

メンバー

```
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)

要件

number_of_threads 値が表 23 の値の 1 つであること。

結果

number_of_threads==task_scheduler_init::deferred の場合は何も起きず、task_scheduler_init はインアクティブのままです。その他の場合、task_scheduler_init は次のようにアクティベートされます。スレッドにほかのアクティブな task_scheduler_init オブジェクトがない場合、スレッドは task オブジェクトのスケジューリングに必要な内部スレッド固有のリソースを割り当てます。アクティブな task_scheduler_init オブジェクトがまだない場合、表 23 で説明されているように内部ワーカースレッドが作成されます。これらのワーカースレッドはタスク・スケジューラーで必要になるまでスリープしています。

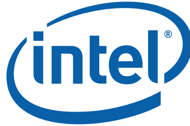
表 23: number_of_threads の値

number_of_threads	セマンティクス
task_scheduler_init::automatic	ライブラリーがハードウェア構成に基づいて number_of_threads を決定するようにします。
task_scheduler_init::deferred	アクティベーション・アクションを遅らせます。
正の整数	ワーカースレッドがまだ存在しない場合、number_of_threads–1 ワーカースレッドを作成します。ワーカースレッドが存在する場合、ワーカースレッド数を変更しません。

8.2.2 ~task_scheduler_init()

結果

task_scheduler_init がインアクティブの場合は何も起きません。その他の場合、task_scheduler_init は次のようにデアクティベートされます。スレッドにほかのアクティブ



な task_scheduler_init オブジェクトがない場合、スレッドは task オブジェクトのスケジューリングに必要な内部スレッド固有のリソースの割り当てを解除します。既存のスレッドにアクティブな task_scheduler_init オブジェクトがない場合、内部ワーカースレッドは終了します。

8.2.3 void initialize(int number_of_threads=automatic)

要件

task_scheduler_init がインアクティブであること。

結果

コンストラクター (8.2.1) を参照してください。

8.2.4 void terminate()

要件

task_scheduler_init がアクティブであること。

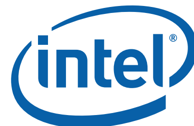
結果

task_scheduler_init を破棄しないでデアクティベートします。デアクティベートの結果は、デストラクター (8.2.2) を参照してください。

8.2.5 OpenMP との混在使用

インテル® スレッディング・ビルディング・ブロックは、OpenMP との混在使用をサポートしています。2 つの並列処理が入れ子になっている場合、OpenMP またはインテル® スレッディング・ビルディング・ブロックを単体で使用したときよりもパフォーマンスは多少低くなります。

task スケジューラーを使用する OpenMP 並列領域はインテル® スレッディング・ビルディング・ブロックにとって未知の新しいスレッドを作成するため、並列領域の内部で task_scheduler_init を作成する必要があります。これらの新しい OpenMP スレッドはそれぞれ、ネイティブスレッドのように、インテル® スレッディング・ビルディング・ブロックのアルゴ



リズムを使用する前に task_scheduler_init オブジェクトを作成しなければなりません。次に例を示します。

```
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 クラス

概要

タスク用の基本クラス。

構文

```
class task;
```

ヘッダー

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

説明

task クラスは、タスク用の基本クラスです。プログラマーは、task からクラスを派生させて、仮想メソッド task* task::execute() をオーバーライドすると予測されます。

task の各インスタンスに関連する属性は、直接見ることはできませんが、task オブジェクトがどのように使用されるかを完全に把握するために理解する必要があります。属性は表 24 で説明しています。

表 24: タスクの属性

属性	説明
owner	タスクを現在担当しているワーカースレッド。
parent	null、またはこのタスクを割り当てた親/継続タスクのいずれか。
depth	タスクツリーにおけるタスクの深さ。
refcount	親が持つタスクの数。refcount のインクリメントおよびデクリメントは常にアトミックです。

ヒント: 常にライブラリーによって提供される特別にオーバーロードされた new 演算子 (8.3.2) を使用して `task` オブジェクトにメモリーを割り当てます。その他の場合、結果は未定義です。`task` の破棄は通常は暗黙的です。タスクのコピー・コンストラクターと代入演算子はアクセスできません。この制限は、タスクが偶然コピーされ、内部データ構造が不正確になったり壊れたりすることを防ぎます。

表記規則

いくつかのメンバーの説明では、図 4 のようなダイアグラムで結果が示されます。

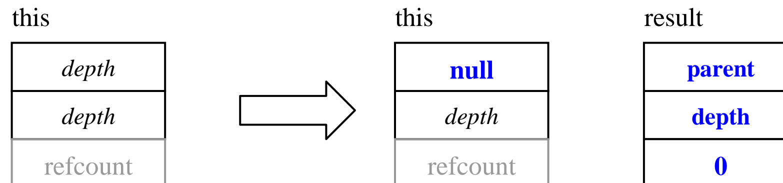
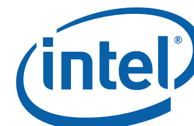


図 4: 結果ダイアグラムの例

これらのダイアグラムの規則は次のとおりです。

- 大きな矢印は古い状態から新しい状態への遷移を示します。
- 各タスクの状態は、*parent*、*depth*、および *refcount* サブボックスに分割されたボックスとして示されます。
- 灰色は、無視される状態を示します。無視される状態は単に空白で示される場合もあります。
- 黒（イタリック） は、読み取られる状態を示します。
- 青（太字）は、書き込まれる状態を示します。



メンバー

次の説明で、型 *proxy1...proxy4* は内部の型です。これらの型を返すメソッドは、セクション (8.3.2) で説明されているように、特別にオーバーロードされた new 演算子とともにのみ使用してください。

```
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 タスクの派生

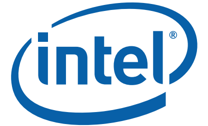
`task` クラスは、抽象的な基本クラスです。`task::execute` メソッドを **必ず**オーバーライドしなければなりません。`execute` メソッドはタスクを実行するために必要なアクションを行ってから、実行する次の `task` を返すか、スケジューラーが実行する次のタスクを選択する場合は NULL を返します。通常は、NULL でない場合、返されるタスクは `this` の子の 1 つです。セクション (8.3.4) で説明されている再利用/再スケジュール・メソッドの 1 つが `execute()` メソッドの実行中に呼び出されなければ、`this` オブジェクトは `execute` メソッドがリターンした後に暗黙的に破棄されます。

コンストラクターによって割り当てられたリソースをリリースする必要がある場合、派生したクラスは仮想デストラクターをオーバーライドします。

8.3.1.1 execute() の処理

スレッドが *task* の実行を開始することをスケジューラーが決定すると、次のステップを行います。

1. `execute()` を呼び出してリターンするのを待ちます。
2. タスクが `recycle_*` メソッドによってマークされていない場合:
    a. タスクの *parent* が null でない場合、*parent->refcount* をアトミックにデクリメントします。ゼロになる場合、レディプールに *parent* を入れます。
    b. タスクのデストラクターを呼び出します。



   c. 再使用のためにタスクのメモリーを解放します。

3. タスクが再利用するようにマークされている場合:

   a. `recycle_to_reexecute` によってマークされている場合は、レディプールにタスクを戻します。

   b. その他の場合、`recycle_as_child` または `recycle_as_continuation` によってマークされています。

8.3.2 タスクの割り当て

常に特別にオーバーロードされた new 演算子の 1 つを使用して `task` オブジェクトにメモリーを割り当てます。割り当てメソッドは、`task` を構築しません。代わりに、ライブラリーによって提供される new 演算子のオーバーロード・バージョンへの引数として使用できるプロキシー・オブジェクトを返します。

一般的に、割り当てメソッドは任意の割り当てられたタスクが生成される前に呼び出さなければなりません。この規則の例外は、`allocate_additional_child_of(t)` です。`task` *t* がすでに実行している場合でも呼び出すことができます。プロキシーの型は実装によって定義されます。唯一の保証は、フレーズ "`new(proxy) T(...)`" が型 *T* のタスクを割り当てて構築するということです。これらのメソッドは慣用句的に使用されるため、サブセクションのヘッディングは宣言ではなく慣用句を示しています。引数 `this` は通常は暗黙的ですが、インスタンス・メソッドとスタティック・メソッドを区別するために、ヘッディングでは明示的に示されています。

ヒント: 216 バイトよりも大きなタスクを割り当てると、より小さなタスクを割り当てる場合よりも大幅に遅くなります。一般に、タスク・オブジェクトは小さな軽いエンティティーにしてください。

8.3.2.1 new(task::allocate_root()) T

現在のネイティブスレッドによって現在実行されている最内 `task` の深さよりも 1 つ深い深さで型 *T* の `task` を割り当てます。図 5 は、状態遷移を要約したものです。

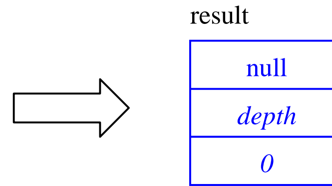


図 5: task::allocate_root() の結果

`spawn_root_and_wait` メソッド (8.3.6.7)を使用して `task` を実行します。

8.3.2.2 new(this. allocate_continuation()) T

this と同じ深さで型 T のタスクを割り当てて構築し、this から新しいタスクに *parent* を転送します。参照カウントは変更されません。図 6 は、状態遷移を要約したものです。

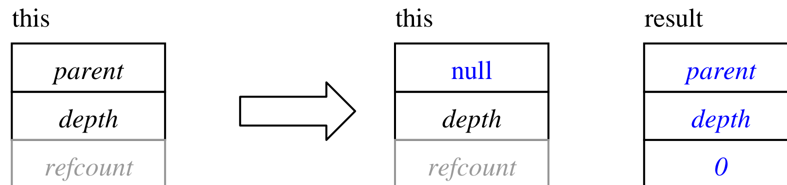


図 6: allocate_continuation() の結果

8.3.2.3 new(this. allocate_child()) T

結果

this より 1 つ深い深さで、this をその *parent* として `task` を割り当てます。図 7 は、状態遷移を要約したものです。

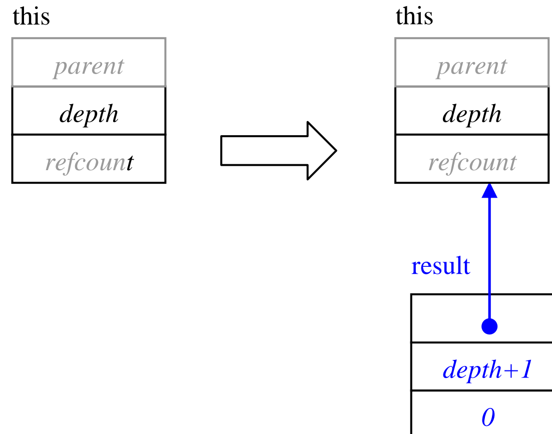


図 7: allocate_child() の結果

明示的な継続渡しを使用する場合、親ではなく継続が割り当てメソッドを呼び出します。その結果、*parent* が正確に設定されます。タスク `this` は、現在のスレッドによって所有されていなければなりません。

タスクの数が小さな定数でない場合は、子の `task_list` (8.5) を最初に構築して、`task::spawn` (8.3.6.3) への単一呼び出しで子を生成してください。すべてが構築される前に `task` が一部の子を生成する必要がある場合、そのメソッドは *refcount* をアトミックにインクリメントするため、代わりに `task::allocate_additional_child_of(*this)` を使用してください。その結果、追加の子が正しく説明されます。しかし、その場合、`task` はブロックスタイルのタスクパターンの使用によって *refcount* が早まってゼロにされないように保護しなければなりません。

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

結果

`task` を別の `task` *parent* の子として割り当てます。結果は this ではなく `parent` の子になります。`parent` は、別のスレッドによって所有されているか、すでに実行中であるか、ほかの子が実行中です。`task` オブジェクト `this` は、現在のスレッドによって所有され、結果には親ではなく現在のスレッドと同じ所有者が含まれなければなりません。図 8 は、状態遷移を要約したものです。

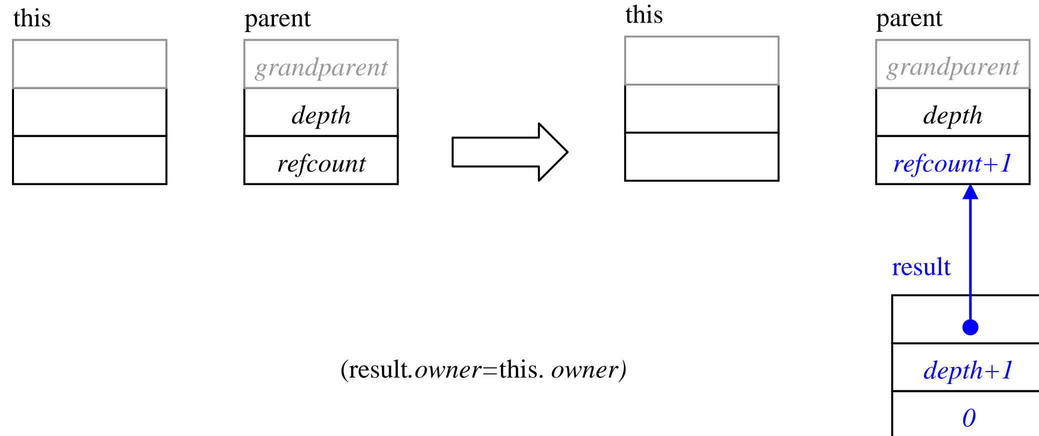


図 8: allocate_additional_child_of(parent) の結果

parent はすでに子を実行しているので、parent.*refcount* のインクリメントは (インクリメントがスレッドセーフではないほかの割り当てメソッドとは異なり) スレッドセーフです。ほかの子を実行している親に子を追加する場合、子が追加される前に親の *refcount* が早まって 0 にならないことと親の実行をトリガーしないことを保証することはプログラマーの責任です。

8.3.3 明示的なタスクの派生

通常、task は、その execute メソッドがリターンした後にスケジューラーによって自動的に破棄されます。しかし、たまに task オブジェクトが execute を実行しないで慣用句的 (例えば、参照カウント) に使用されます。そのようなタスクは、destroy メソッドを使用して処理してください。

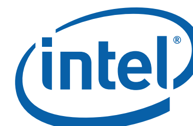
8.3.3.1 void destroy(task& victim)

要件

victim の参照カウントが 0 であること。この要件はライブラリーのデバッグバージョンでチェックされます。呼び出しスレッドが this を所有していること。

結果

デストラクターを呼び出して *victim* のメモリーの割り当てを解除します。this に null でない *parent* が含まれる場合、*parent->refcount* をアトミックにデクリメントします。*parent->refcount*



がゼロになる場合、*parent* は レディプールに**格納されません**。図 9 は、状態遷移を要約したものです。

暗黙的な引数 `this` は内部的に使用され、目に見える影響は受けません。`task` は自身を破棄することができます。例えば、“`this->destroy(*this)`” は、`task` が生成されているが、`execute` メソッドがまだ完了していない場合を除いて許可されます。

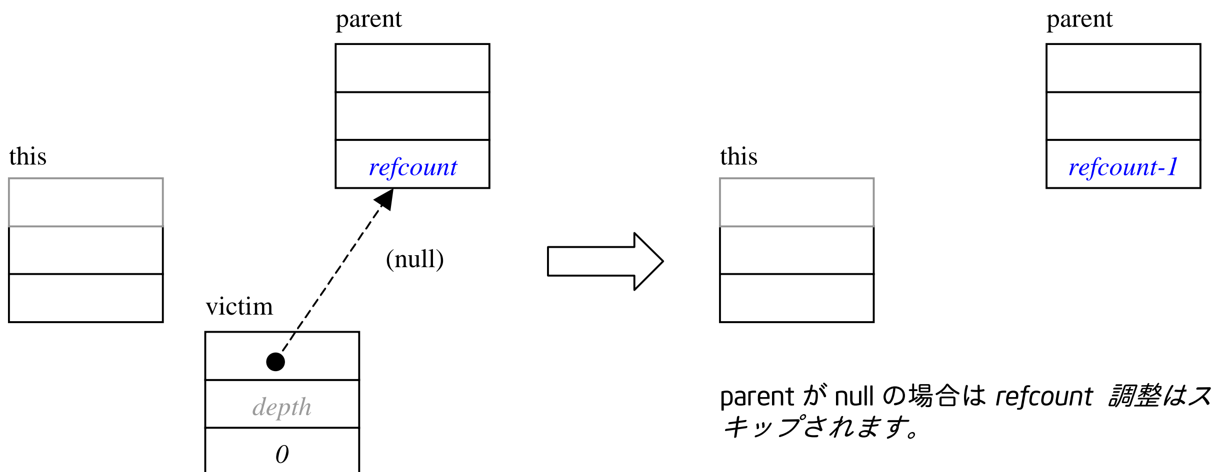


図 9: destroy(victim) の結果

8.3.4 タスクの再利用

最初から再割り当てを行うよりも、`task` オブジェクトを再利用するほうが多くの場合効率的です。しばしば、親は継続、または子の 1 つになることができます。

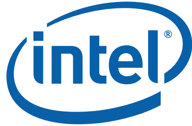
8.3.4.1 void recycle_as_continuation()

要件

`execute()` メソッドが実行している間に呼び出されること。

再利用するタスクの *refcount* は n に設定する必要があります。ここで、n は継続タスクの子の数です。

メモ: 呼び出し元は、`execute()` メソッドがリターンするまでタスクの *refcount* がゼロにならないことを保証する必要があります。保証できない場合、代わりに



`recycle_as_safe_continuation()` メソッドを使用して、refcount を n+1 に設定してください。

結果

`execute()` メソッドがリターンしたときに `this` を破棄しません。

8.3.4.2 void recycle_as_safe_continuation()

要件

`execute()` メソッドが実行している間に呼び出されること。

再利用するタスクの *refcount* は n+1 に設定する必要があります。ここで、n は継続タスクの子の数です。追加の +1 は、タスクが再利用されることを表します。

結果

`execute()` メソッドがリターンしたときに `this` を破棄しません。

このメソッドは、`recycle_as_continuation` メソッドの使用によって発生する競合状態を回避します。競合は、次の場合に発生します。

1. `execute()` メソッドが `this` を継続として再利用します。
2. 継続が子を作成します。
3. `execute()` メソッドが完了する前にすべての子が完了します。つまり、スケジューラーが `this` の実行を完了する前に継続が実行するため、スケジューラーが破損します。

refcount の追加の 1 によって、タスクが完了するまで継続が実行されないため、`recycle_as_safe_continuation` メソッドはこの競合状態を回避します。

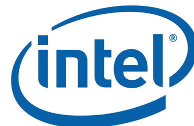
8.3.4.3 void recycle_as_child_of(task& parent)

要件

`execute()` メソッドが実行している間に呼び出されること。

結果

`this` を *parent* の子として、`execute()` メソッドがリターンしたときに破棄しません。



8.3.4.4 void recycle _to_reexecute()

要件

`execute()` メソッドが実行している間に呼び出されること。`execute()` メソッドが別の `task` へのポインターを返すこと。

結果

`this` は `execute()` メソッドがリターンした後自動的に生成されます。

8.3.5 タスクの深さ

一般的な fork-join 並列処理では、タスクの深さを明示的に設定する必要はありません。しかし、fork-join パターンに従わない特殊なタスクパターンでは、タスクの深さを明示的に設定または調整することが役立つ場合があります。

8.3.5.1 depth_type

`task::depth_type` 型は、処理系定義の符号付き整数型です。

8.3.5.2 depth_type depth() const

リターン

タスクの現在の *depth* 属性。

8.3.5.3 void set_depth(depth_type new_depth)

要件

new_depth 値が負でないこと。

結果

タスクの depth 属性を *new_depth* に設定します。図 10 は、効果を示しています。

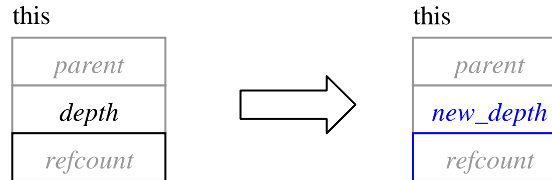


図 10: set_depth の結果

8.3.5.4 void add_to_depth(int delta)

要件

タスクがレディプールにないこと。*depth+delta* が負でないこと。

結果

タスクの depth 属性を depth+*delta* に設定します。図 11 は、効果を示しています。更新はアトミックではありません。

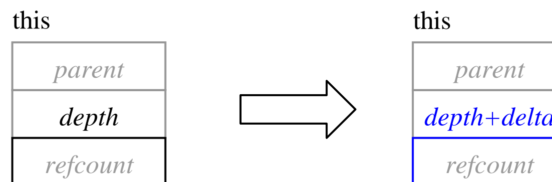


図 11: add_to_depth(delta) の結果

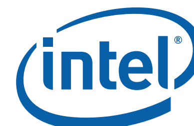
8.3.6 同期

task を生成すると、*task*.`execute()` を呼び出すスレッドが呼び出されるか、*task* がレディプールに格納されます。その後、タスク・スケジューリングに関与しているスレッドがタスクを取得して、*task*.`execute()` を呼び出します。セクション 8.1 は、レディプールの構造を示しています。

タスクを生成する呼び出しには 2 つの形式があります。

1. 単一の `task` を生成する。
2. `task_list` で指定された複数の `task` オブジェクトを生成して `task_list` を消去する。

呼び出しは、生成されるルートタスクと子タスクを識別します。ルートタスクは、`allocate_root` メソッドを使用して作成されたものです。



重要

`set_ref_count` メソッドを呼び出して、子の数と "wait_for_all" メソッドの 1 つを使用するかどうかの両方を示すまで、`task` は子を生成しません。

8.3.6.1 void set_ref_count(int count)

要件

count>0。連続で *n* の子を生成して待つ場合、*count* が *n*+1 であること。その他の場合、*count* が *n* であること。

結果

`refcount` 属性を *`count`* に設定します。

8.3.6.2 void wait_for_all()

要件

`refcount``=n+1`。ここで、*n* はまだ実行している子の数。

結果

refcount が 1 になるまでレディプールのタスクを実行します。その後、*refcount* を 0 に設定します。図 12 は、状態遷移を要約したものです。

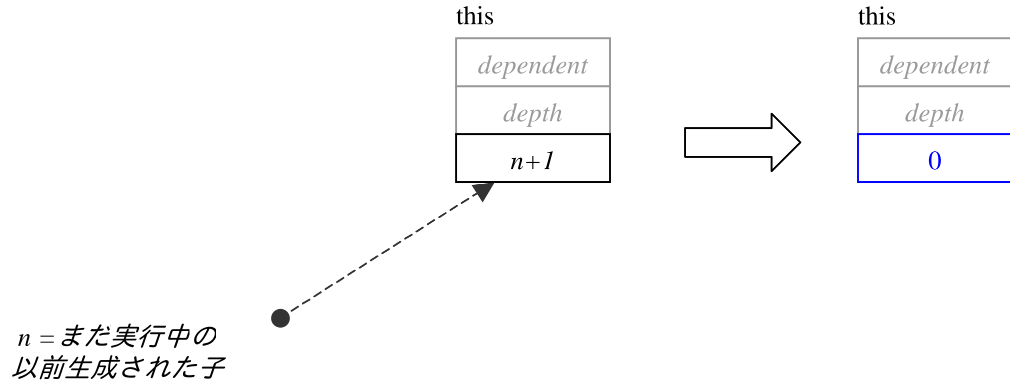


図 12: wait_for_all の結果

8.3.6.3 void spawn(task& child)

要件

`child.refcount>0`

呼び出しスレッドが `this` および *`child`* を所有していること。

結果

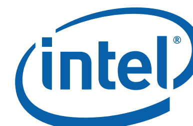
タスクをレディプールに入れて直ちにリターンします。生成を行う `this` タスクは、呼び出し元スレッドによって所有されます。呼び出し元スレッドによって所有される場合、`task` は自身を生成します。現在のスレッドによって所有される `task` がすぐ使用できない場合、`task::self().spawn(task)` を使用して `task` を生成します。

子タスクが完了すると *refcount* が非同期にデクリメントされるため、親は子タスクが生成される前に `set_ref_count` を呼び出す必要があります。ライブラリーのデバッグバージョンは、`set_ref_count` への必要な呼び出しが行われなかった場合、または遅すぎた場合を検出します。

8.3.6.4 void spawn (task_list& list)

要件

list の各タスクで *refount*>0 であること。呼び出しスレッドが `this` および *list* の各タスクを所有していること。*list* の各タスクで *depth* 属性の値が同じであること。



結果

list の各タスクで `spawn` を実行して *list* を消去するのと等価ですが、より効率的です。*list* が空の場合、何も起こりません。

8.3.6.5 void spawn_and_wait_for_all(task& child)

要件

`this` のほかの子がすでに生成されていること。`task` *child* が非 null 属性の *parent* を含むこと。子から呼び出し `task` まで *parent* リンクのチェーンがあること。通常、このチェーンは単一のリンクを含みます。つまり、*child* は `this` の子です。

結果

`{spawn(`*task*`); wait_for_all();}` に似ていますが、多くの場合より効率的です。さらに、*task* が現在のスレッドによって実行されることを保証します。この制約は、たまに同期を単純化します。図 13 は、状態遷移を要約したものです。

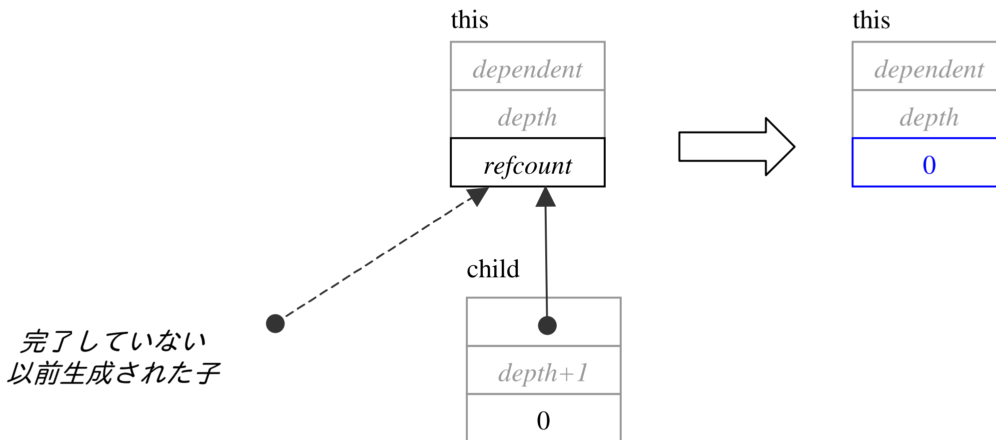
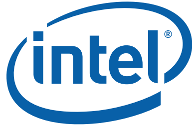


図 13: spawn_and_wait_for_all の結果

8.3.6.6 void spawn_and_wait_for_all(task_list& list)

結果

`{spawn(`*list*`); wait_for_all();}` に似ていますが、多くの場合より効率的です。



8.3.6.7 static void spawn_root_and_wait(task& root)

要件

root タスクのメモリーが `task::allocate_root()` によって割り当てられていること。呼び出しスレッドが `root` を所有していること。

結果

root の *parent* 属性に未定義の値を設定して、セクション 8.3.1.1 で説明されているように *root* を実行します。*root* が再利用されていなければ、その後で *root* を破棄します。

8.3.6.8 static void spawn_root_and_wait(task_list& root_list)

要件

root_list の各 `task` オブジェクト *t* が 8.3.6.7 の要件を満たしていること。

結果

root_list の各 `task` オブジェクト *t* について、`spawn_root_and_wait`(*t*) を並列で実行します。セクション 8.3.6.7 は、`spawn_root_and_wait`(*t*) のアクションを説明しています。

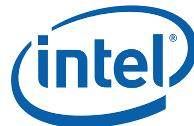
8.3.7 タスクのコンテキスト

これらのメソッドは、`task` オブジェクト間、および `task` オブジェクトと基礎を成す物理スレッド間の関係を示します。

8.3.7.1 static task& self()

リターン

呼び出しスレッドが実行している最内 `task` への参照。



8.3.7.2 task* parent() const

リターン

parent 属性の値。タスクが `allocate_root` によって割り当てられ `spawn_root_and_wait` の制御のもとに現在実行している場合、結果は未定義の値です。

8.3.7.3 bool is_stolen_task() const

要件

parent 属性が null ではなく、`this.execute()` が実行していること。呼び出しタスクが allocate_root で割り当てられていないこと。

リターン

`this` の属性 *owner* が *parent* の *owner* と等しくない場合は true。

8.3.8 タスクのデバッグ

このサブセクションのメソッドは、デバッグに役立ちます。将来の実装では変更される場合があります。

8.3.8.1 state_type state() const

注意: このメソッドはデバッグ専用です。動作またはパフォーマンスは、将来の実装では変更される場合があります。`task::state_type` の定義は、将来の実装では変更される場合があります。この情報は、デバッグ中の問題診断に役立つように提供されています。

リターン

タスクの現在の状態。表 25 に、有効な状態を示します。ほかの値の場合、メモリーの割り当てを解除した `task` を使用するなどして、メモリーが破損していることを意味します。

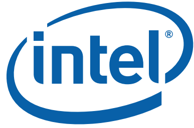


表 25: task::state() が返す値

値	説明
`allocated`	タスクは新たに割り当てられたか、または再利用されました。
`ready`	タスクはレディプールにあるか、レディプールとの間で転送中です。
`executing`	タスクは実行中で、`execute()` メソッドがリターンした後に破棄されます。
`freed`	タスクは内部フリーリストにあるか、フリーリストとの間で転送中です。
`reexecute`	タスクは実行中で、`execute()` メソッドがリターンした後に再生成されます。

図 14 は、task の可能な状態遷移を要約したものです。

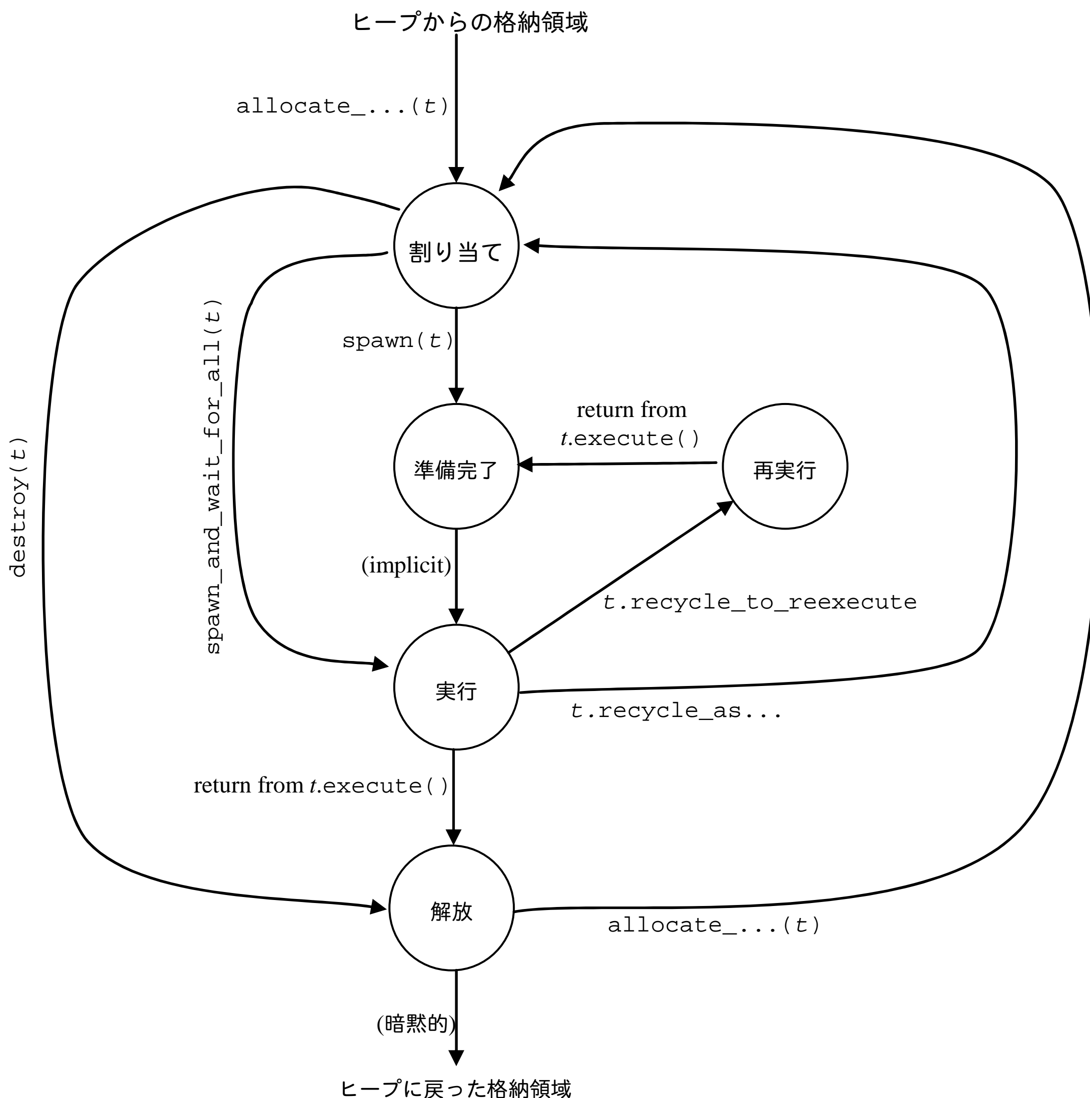
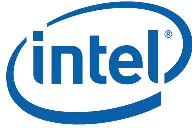


図 14: 典型的な task::state() の遷移

8.3.8.2 int ref_count() const

注意: このメソッドはデバッグ専用です。動作またはパフォーマンスは、将来の実装では変更される場合があります。



リターン

属性 *refcount* の値。

8.4 empty_task クラス

概要

何もしないことを表す task のサブクラス。

構文

```
class empty_task;
```

ヘッダー

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

説明

empty_task は、何も行わないタスクです。継続が子の完了を待つ以外に何もするべきではない場合、親タスクの継続として役立ちます。

メンバー

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

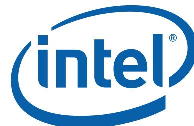
8.5 task_list クラス

概要

task オブジェクトのリスト。

構文

```
class task_list;
```



ヘッダー

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

説明

task_list は、task オブジェクトへの参照のリストです。task_list の目的は、8.3.6.4 で説明されているように、task が子タスクのリストを作成し、task::spawn(task_list&) メソッドを使用してすべての子タスクを一度に生成できるようにすることです。

task は一度に多くても 1 つの task_list に属し、その task_list に多くて一回含まれます。生成されたが実行を開始しなかった task は、task_list に属してはなりません。task_list はコピー構築または割り当てできません。

メンバー

```
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()

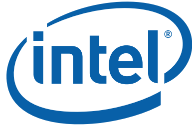
結果

空のリストを構築します。

8.5.2 ~task_list()

結果

リストを破棄します。task オブジェクトは破棄しません。



8.5.3 bool empty() const

リターン

リストが空の場合は true。その他の場合は false。

8.5.4 push_back(task& task)

結果

リストの後ろに *task* への参照を挿入します。

8.5.5 task& task pop_front()

結果

リストの前から `task` 参照を削除します。

リターン

削除された参照。

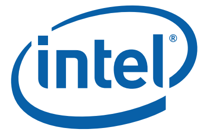
8.5.6 void clear()

結果

リストからすべての `task` 参照を削除します。`task` オブジェクトは破棄しません。

8.6 推奨するタスク回帰パターンのカタログ

子のセクションは、3 つの推奨するタスク回帰パターンをカタログに載せます。各パターンで、`T` クラスは `task` クラスから派生すると仮定されます。サブタスクは `t1`、`t2`、… `tk` と表記します。添字は、並列処理が利用可能でない場合にサブタスクを実行する順序を示します。並列処理が利用可能な場合、サブタスクの実行順は、t_1 が生成されるスレッドによって実行されることを保証される以外は非決定性です。



8.6.1 ブロックスタイルと *k* の子

次に、各レベルが *k* の子を生成する型 *T* の再帰タスク用の推奨スタイルを示します。

```
task* T::execute() {
    if( not recursing any further ) {
        ...
    } else {
        set_ref_count(k+1);
        task& tk = new( allocate_child() ) T(...);  tk.spawn();
        task& tk-1= new( allocate_child() ) T(...);  tk-1.spawn();
        ...
        task& t1 = new( allocate_child() ) T(...);   t1.spawn_and_wait(t1);
    }
    return NULL;
}
```

タスクの生成前にタスクが構築される限り、子の構築と生成の順序は変更できます。

パターンのキーポイントは次のとおりです。

- `set_ref_count` への呼び出しはその引数として *k*+1 を使用します。追加の 1 は重要です。
- タスクはそれぞれ `allocate_child` によって割り当てられます。

8.6.2 継続渡しスタイルと *k* の子

2 つの推奨するスタイルがあります。これらのスタイルは、継続として親を再利用するほうが便利なのか、子として親を再利用するほうが便利なのかという点で異なります。その決定は、継続と子のどちらがより親のように働くかどうかに基づいて行います。

8.6.2.1 継続としての親の再利用

継続が親の状態の多くを継承する必要があり、子が状態を必要としない場合、このスタイルが役立ちます。継続は親と同じ型を持っていなければなりません。

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
    } else {
        set_ref_count(k);
        recycle_as_continuation();
        task& tk = new( allocate_child() ) T(...);  tk.spawn();
        task& tk-1 = new( allocate_child() ) T(...); tk-1.spawn();
        ...
        task& t1 = new( c.allocate_child() ) T(...);  t1.spawn();
        return &t1;
    }
}
```

パターンのキーポイントは次のとおりです。

- set_ref_count への呼び出しはその引数として *k* を使用します。セクション 8.6.1 で説明したブロックスタイルにあった追加の 1 はありません。
- 子タスクはそれぞれ allocate_child によって割り当てられます。
- 継続は親から再利用され、コピー操作を行わずに親の状態を取得します。

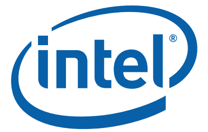
8.6.2.2 子としての親の再利用

子が親からのその状態の多くを継承し、継続で親の状態を必要としない場合、このスタイルが役に立ちます。子は親と同じ型を持っていなければなりません。例では、C は継続の型で、task クラスから派生します。C がすべての子が完了するのを待つ以外に何もしない場合、C は empty_task クラス (8.4) です。

```
task* T::execute() {
    if( not recursing any further ) {
        ...
        return NULL;
    } else {
        set_ref_count(k);
        // Construct continuation
        C& c = allocate_continuation();
        // Recycle self as first child
        task& tk  = new( c.allocate_child() ) T(...); tk.spawn();
        task& tk-1 = new( c.allocate_child() ) T(...); tk-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;
    }
}
```

パターンのキーポイントは次のとおりです。

- set_ref_count への呼び出しはその引数として *k* を使用します。セクション 8.6.1 で説明したブロックスタイルにあった追加の 1 はありません。
- t_1 を除く子タスクはそれぞれ c.allocate_child によって割り当てられます。(*this).allocate_child ではなく c.allocate_child を使用することが重要です。そうしないと、タスクグラフが間違ってしまいます。
- タスク t_1 は親から再利用され、コピー操作を行わずに親の状態を取得します。子を表すように状態を更新することを忘れないでください。そうしないと、無限の再帰が発生します。



9 参考文献

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.