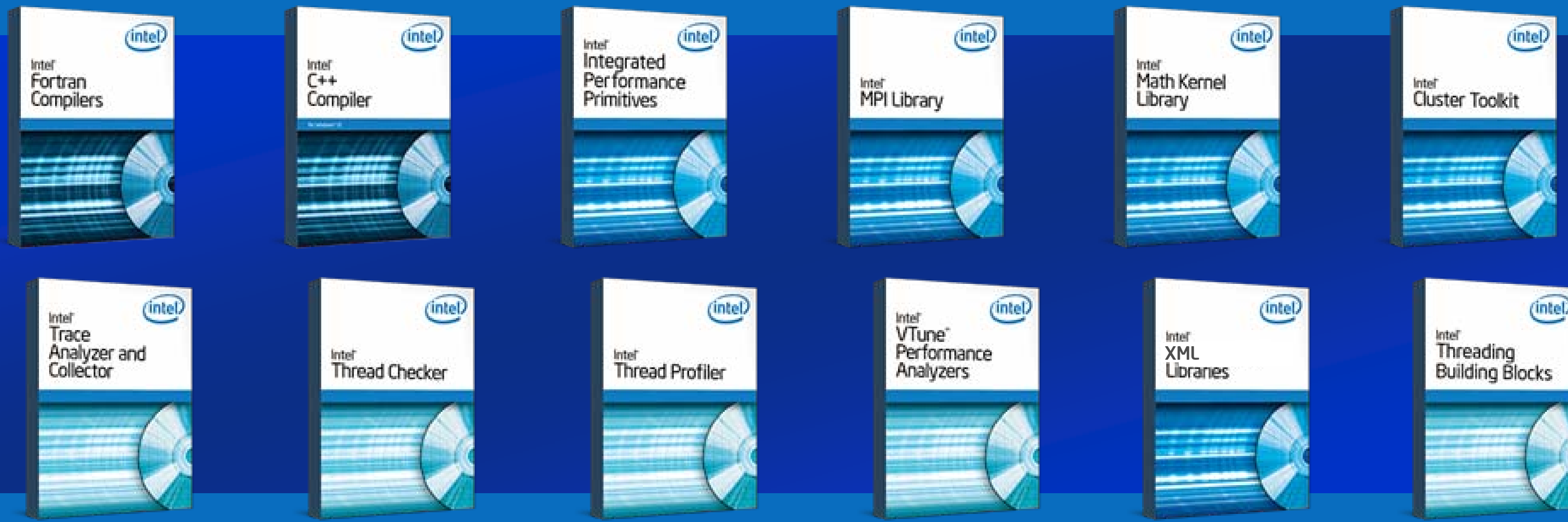
インテル® ソフトウェア開発製品

James Reinders
セールス & マーケティング・ディレクター

インテル® ソフトウェア開発製品



インテル® C++ コンパイラーは、Windows*、Linux* 、Mac OS* X で開発を行う Microsoft* と gcc (GNU) ユーザーの最も一般的なアップグレード・パス

インテル® Fortran コンパイラーは、現在最も使われている Fortran コンパイラー

ライブラリー、解析ツール、クラスターツールに加えて、スレッド化の助けとなる新しい製品が追加されている

インテル® スレッディング・ビルディング・ブロック (インテル® TBB) は、並列処理用の C++ 拡張ライブラリー

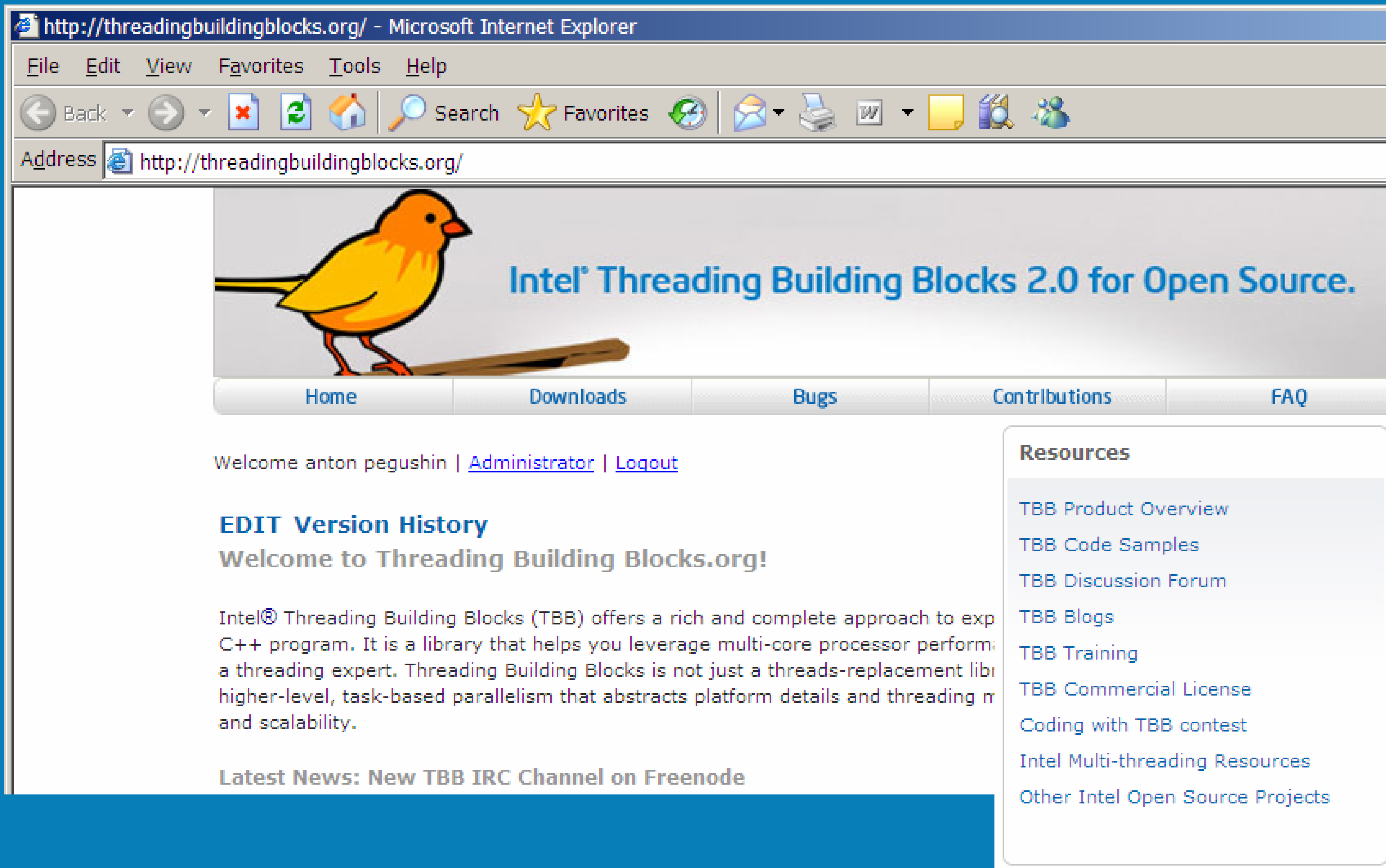
インテル® TBB の紹介

Robert Reed

テクニカル・コンサルティング・エンジニア

ソフトウェア & ソリューション・グループ

オープンソースと商用 – ライセンスを除いて同一



TBB に関する非常に活発なフォーラム、ブログ、FAQ

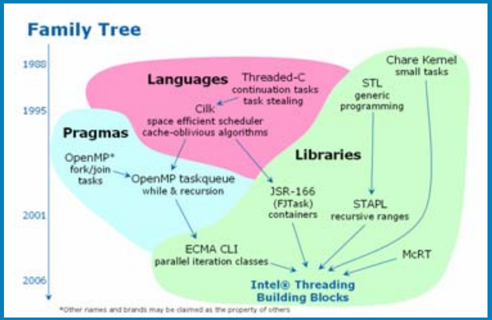
- 扱われている内容:
  - –ライセンス
  - –移植に関する問題
  - –ベンチマーク
  - –オーバーヘッドの評価
  - –ライブラリーの機能 (既存/拡張)
  - –互換性
  - –共存
  - –さまざまなケースでの利用

Debug version should do better job of checking ownership by MAD\adrobiso	by MAD\adrobiso **Yesterday, 10:19 AM**	-	17
Splitting Constructors Odd by DEADBEEF	by MAD\adrobiso **Yesterday, 10:09 AM**	3	183
What exactly is meant by "multi-core"? by masserberg	by masserberg **Yesterday, 6:36 AM**	2	31
Compiler warnings when building code by etc_26	by Alexey Kukanov **Yesterday, 1:38 AM**	2	103
Thread checker infinite loop by MAD\mkgumbel	by MAD\apegushi 08-17-2007 5:39 AM	1	69
Outfitting C++ for Multie-core Processor Parallelism by kenthompson1@gmail.com	by MAD\rdsmith1 08-11-2007 9:53 PM	2	152
PowerPC by tcmichals@yahoo.com	by tcmichals@yahoo.... 08-10-2007 5:15 AM	4	116
Sleeping Tasks And Waiting For Another Task To Wakeup by aj.guillon@gmail.com	by aj.guillon@gmail... 08-09-2007 4:37 PM	6	146
Licensing: Using TBB (open source version) in commercial apps by claus_brod@cocreate.com	by MAD\rdsmith1 08-07-2007 11:23 AM	3	263
A Debian/Ubuntu TBB package by branan@gmail.com	by branan@gmail.com 08-05-2007 9:16 PM	-	94

インテル® TBB の内容

基本的なアルゴリズム: parallel_for, parallel_reduce, parallel_scan

高度なアルゴリズム: parallel_while, pipeline, parallel_sort



コンテナー: concurrent_queue,
concurrent_vector,
concurrent_hash_map

スケーラブルなメモリー割り当て:
scalable_malloc, scalable_free,
scalable_realloc, scalable_calloc,
scalable_allocator, cache_aligned_allocator

排他制御:

- **mutex:** mutexspin_mutex, queuing_mutex,
spin_rw_mutex, queuing_rw_mutex
- **アトミック操作:** fetch_and_add, fetch_and_increment,
fetch_and_decrement, compare_and_swap,
fetch_and_store

時間計測: 移植性のあるファイングレイン (細粒度) のグローバル・タイム・スタンプ

タスク・スケジューラー: タスクの作成とアクティベーションを制御する直接アクセス

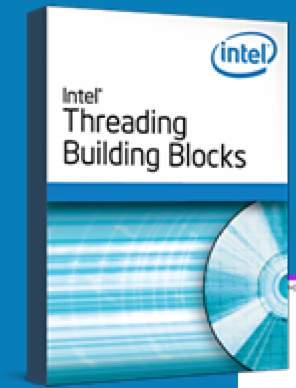
インテル® TBB 2.0
threadingbuildingblocks.org

- **多くのプラットフォームをサポート**
  すべてのコンパイラー、すべての OS、すべてのプロセッサーをサポートすることを*目標*とし、現在は主要なものをサポートしている
- **オープンソース・プロジェクト**
  http://threadingbuildingblocks.org (標準ライセンス: GPL v2、ランタイム例外 – gnu C++ が長年使用してきたものと同じ)
- **オープンソースと商用版の違い**
  継続して提供される商用版ではインテルからサポートを受けられる
- **オライリーの Nutshell 本**

  オライリーから出版された TBB の Nutshell 本

  初心者から熟練者まで、すべての C / C++ プログラマーに役立つ

  日本語訳版は近日出版予定



TBB が... C++ 標準ライブラリーの並列性における基礎となったのは喜ばしいことです。
Alexander Stepanov 氏
アドビシステムズ

TBB は C++ における新しい並列プログラミング方法を提供します。
Marc Snir 氏
イリノイ大学

インテルの TBB と並列プログラミングの関係は、単純な C++ と STL のようなものです。
Lawrence Rauchwerger 氏
テキサス A&M 大学

インテル® TBB はさまざまな機能をサポート

タスク・イニシエーター

parallel_for　parallel_while
parallel_reduce　parallel_scan
pipeline　parallel_sort

タスク・スケジューラー

スレッドセーフ・コンテナー

concurrent_vector
concurrent_queue
concurrent_hash_map

時間計測関数

メモリー割り当て

scalable_allocator　cache_aligned_allocator

同期プリミティブ

ネイティブ mutex を含むスコープロック、
スピンまたはキュー、単純な型または
読み取り/書き込み型、アトミック型

インテル® TBB の設計目標

- プロセッサー数の増加に伴う並列プログラミングの負担を軽減
- 簡単な説明でパフォーマンスを向上する先進の機能を開発
- アクセス可能な手法を構築して採用を促進

インテル® TBB の概念

- ワーカー (スレッド) ではなく作業 (タスク) に注目
- 再帰的な並列処理を利用する機会を最大限にする
  – キャッシュに収まるように範囲を分割し、ワークスチールを促進
- 汎用プログラミングを使用して新しいデータ型とアプリケーションで動作する効率的なコードを抽象化

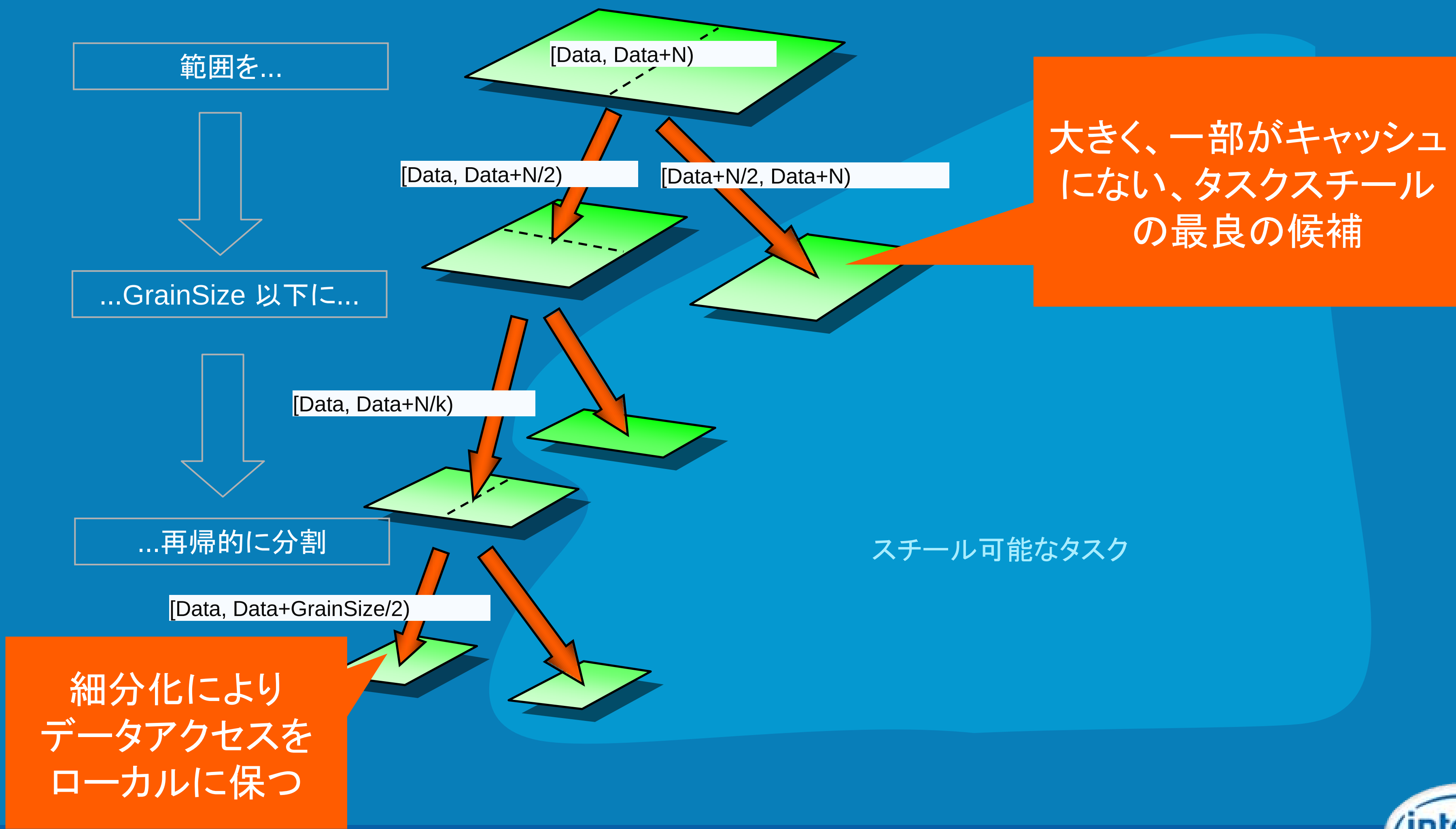
メモリーはフリーではない – レイテンシー

- CPU / メモリーの速度差は徐々に拡大している
- 最近のアーキテクチャーには 3〜4 のメモリーレイヤーが存在
- キャッシュの最適化がパフォーマンス向上の鍵
- インテル® TBB は、分割可能な範囲の概念を使用

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
bool R::is_divisible() const	範囲を分割できる場合は true
R::R (R& r, split)	*r* を 2 つのサブ範囲に分割

- 範囲分割とスレッドの割り当てはライブラリーによって制御
- 現在サポートしているクラス: blocked_range、blocked_range2d (まもなく 3d)
- サブ範囲の深さを粒度で制御

分割範囲をローカルに保つ



コードへの適用方法

- 単純なコード例から始める

```
long d[10000];
fill_data(&d);
long sum = 0, sumsq = 0;
for (int i = 0; i < 10000; ++i) {
    sum += d[i];
    sumsq += d[i] * d[i];
}
```

コードへの適用方法

```
class Body {
    Body() {}
    operator() (const blocked_range<int> &r) const {
        for (int i = r.begin(); i != r.end(); ++i) {
            sum += d[i];
            sumsq += d[i] * d[i]; }
    }
};
long d[10000];
fill_data(&d);
long sum = 0, sumsq = 0;
parallel_for (blocked_range<int>(0, 10000), Body body, auto_partitioner());
```

平均と標準偏差の計算

- 配列 d は blocked_range と parallel_for を使用してタスクのセットに細分化できる
- for の範囲を blocked_range に変更する
- ループ本体は個別の関数とクラスに含める必要がある
- 最後に、parallel_for (と他のコード) を追加する
- または、スタティック・パーティショナーを使用して粒度を調整する

```
parallel_for (blocked_range<int>(0, 10000, 200), Body body);
```

コードの確認

```
tbb::atomic<float>  sum, sumsq;
class Body {
    Body() {}
    operator() (const blocked_range<int> &r) {
        for (int i = r.begin(); i != r.end(); ++i) {
            sum += d[i];
            sumsq += d[i] * d[i];
}   }   };
long d[10000];
fill_data(&d);
long sum = 0, sumsq = 0;
parallel_for (blocked_range<int>(0, 10000), Body body, auto_partitioner);
```

- インテル® スレッドチェッカーを実行すると、ここに競合状態があることがわかる
  - アクセスをアトミックにする: 最も簡単だが非効率
    - 注: atomic は整数型でのみ動作 (float や double 型では動作しない)

コードの確認

```
class Body {
    Body() {}
    operator() (const blocked_range<int> &r) {
        for (int i = r.begin(); i != r.end(); ++i) {
            SummingMutexType::scoped_lock lock(SummingMutex);
            sum += d[i];
            sumsq += d[i] * d[i];
}   }   };
long d[10000];
fill_data(&d);
long sum = 0, sumsq = 0;
parallel_for (blocked_range<int>(0, 10000), Body body, auto_partitioner());
```

- インテル® スレッドチェッカーを実行すると、ここに競合状態があることがわかる
  - アクセスをアトミックにする: 最も簡単だが非効率
    - ＞注: atomic は整数型でのみ動作 (float や double 型では動作しない)
  - スコープロックを使用する: 簡単だが効率はあまり良くない

コードの確認

```
class Body {
    long m_sum, m_sumsq;
    Body() : m_sum(0), m_sumsq(0) {}
    operator() (const blocked_range<int> &r) {
        for (int i = r.begin(); i != r.end(); ++i) {
            m_sum += d[i];
            m_sumsq += d[i] * d[i]; } }
    void join(Body &rhs) { m_sum += rhs.m_sum; m_sumsq += rhs.m_sumsq; }
}   };
long d[10000];
fill_data(&d);
Body body;
parallel_reduce (blocked_range<int>(0, 10000), body, auto_partitioner());
```

- インテル® スレッドチェッカーを実行すると、ここに競合状態があることがわかる
  - アクセスをアトミックにする: 最も簡単だが非効率
    - >注: atomic は整数型でのみ動作 (float や double 型では動作しない)
  - スコープロックを使用する: 簡単だが効率はあまり良くない
  - parallel_reduce に変更する: この例では最良のソリューション

はじめに – 初期化が必要

- 最初の並列コードの前に、初期化を行う

```
main (int argc, char **argv)
{
    task_scheduler_init init;
    …プログラムの残りの部分
}
```

- このオブジェクトは、スレッドプールの作成を含む状態を定義する
  - スレッドカウントはオブジェクトの存在期間によって決まる
  - デフォルトは利用可能な処理要素あたり 1 つのスレッド
  - 少なくとも 1 つの初期化が必要 - 追加の宣言は参照カウント
  - スレッドカウントは明示的に保留または設定できる

    task_scheduler_init init(*thread-count* | task_scheduler_init::automatic|deferred);
  - スレッド化環境を併用する場合、オーバーサブスクリプションを回避する必要がある
  - init オブジェクトはダイナミックにスレッドカウントを変更できる

    ＞(init オブジェクトの存在期間を制御)

インクルード・ファイル

- すべての機能には独自のインクルード・ファイルがある

  task_scheduler_init?#include <tbb/task_scheduler_init.h>
  blocked_range?#include <tbb/blocked_range.h>
  parallel_for?#include <tbb/parallel_for.h>
  spin read-write mutex?#include <tbb/spin_rw_mutex.h>

- 多くの TBB の動作はインライン展開をサポートするため、ヘッダーでエンコードされる
  – ヘッダーをよく見ると多くのことがわかる

```
template<typename RandomAccessIterator, typename Compare>
struct quick_sort_range {
    static const size_t grainsize = 500;
```

- 2、3 のデバッグ宣言もある

  #define TBB_DO_ASSERT 1 // ヘッダーファイルにエラーチェックを追加

  – Windows システムでは自動的にアサートされ、デバッグ・ライブラリーとともに使用される
  – tbb::assertion_failure にデバッガーのブレークポイントを設定するとエラーになる

  #define TBB_DO_THREADING_TOOLS 1 // フックを追加して解析を改善

  – コスト・パフォーマンスを調整して解析を改善する
  – 現在の実装で影響を受ける機能: *spin_mutex/spin_rw_mutex*

parallel_for – 多少フォーマル

- 2 つの形式がある

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

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

- 前者は、単純な分割機能の使用を意味する
  - 範囲のサイズが粒度より大きい間は細分化が続行される
  - 粒度は、Range コンストラクターで指定する

    parallel_for(blocked_range<*type*>(*start, end, grain size*), *body*);

  - 3D になると、チューニングが難しくなる

    blocked_range3d<type>(start1, end1, grain1, start2, end2, grain2, start3, end3, grain3);

- 自動パーティショナーは、スレッド数に基づいて粒度を設定する
  - パフォーマンスが最適でない場合もゴールは信用できる

blocked_range<*type*>

- ブロック範囲は、ワークカーネルの内部で連続する区間を抽象化する

```
for (type i = begin; i != end; ++i)
は
for (type i = r.begin(); i != r.end(); ++i)
```

に変換される

- 2d blocked_range はループのペアを入れ子にする

```
for (type i = begini; i != endi; ++i) {
    for (type j = beginj; j != endj; ++j) {
は次のように変換される
for (type i = r.rows().begin(); i != r.rows().end(); ++i) {
    for (type j = r.cols().begin(); j != r.cols().end(); ++j) {
```

コードをまとめる

- parallel_<something> の一般的な形式は次のようになる

```
class BodyClass {
    int m_thing1, m_thing2;  // ループに利用可能なオブジェクトとメンバーのローカルコピー
  public:
    BodyClass(int thing1, int thing2 …) :
                      m_thing1(thing1), m_thing2(thing2) … {}  // ローカルデータを初期化
    operator() (const blocked_range<int> &r) {
        for (int i = r.begin(); i != r.end(); ++i) {
            ….
        }
    }
};
```

- 並列コードを呼び出すには次のように記述する

```
BodyClass body(thing1, thing2 … );
parallel_for( blocked_range<int>(0, limit), body, auto_partitioner());
```

parallel_reduce

- parallel_for に累積プロセス (ベクトルからスカラー) を追加する

```
class BodyClass {
    int m_thing1, m_thing2;  // ループに利用可能なオブジェクトとメンバーのローカルコピー
    int m_sum;  // ローカル累積変数
  public:
    BodyClass(int thing1, int thing2 …) : m_thing1(thing1), m_thing2(thing2), m_sum(0) … {}
    operator() (const blocked_range<int> &r) {
        for (int i = r.begin(); i != r.end(); ++i) {
            m_sum += <something> ;
            ….
    BodyClass (BodyClass &b, split) : m_thing1(b.m_thing1), m_thing2(b.m_thing2), m_sum(0) … {}
    join( const BodyClass &b) { m_sum += b.m_sum; }
};
```

- リダクション演算は任意の結合演算またはプロセスになる
  $(a \oplus b) \oplus c == a \oplus (b \oplus c)$
  – 減算では動作しない! ☺
  – 浮動小数点演算 (内積?) でも同様に動作しない

時間計測

- パフォーマンスを計測するタイミングは tick_count を使用すると簡単

```
tick_count t0 = tick_count::now();

// コード

tick_count t1 = tick_count::now();
double duration = (t1-t0).seconds();
```

tick_count::interval_t は、加算と減算、秒への変換をサポート

– スレッド間の時間の比較は安全であることが保証される
  > 異なるスレッドで記録された tick の差分を求めることができる
– 分解能はスレッド間で提供されるサービスの最も高い分解能に対応
  > しかし、プラットフォーム間の一貫性は保証されない

同期プリミティブ

- 並列タスクは共有データを更新する
  - データの更新が重なる場合、競合状態を回避するために**排他制御**を使用する
- TBB のすべての排他制御領域は**スコープロック**によって保護されている
  - ロックの範囲はロックの期間によって決定される
  - ロックの範囲から出るにはデストラクターを呼び出し、例外時も安定させる
  - ロックを短くして競合状態を防ぐ

```
operator() (…) {
    // 他のカーネル作業
    { // ロックの範囲を制限する内側の範囲
        SummingMutexType::scoped_lock lock(SummingMutex);
        // 競合状態が発生しやすい操作
    }
    // 他のカーネル作業
}
```

同期プリミティブ

- 並列タスクは共有データを更新する
  - データの更新が重なる場合、競合状態を回避するために排他制御を使用する
- TBB のすべての排他制御領域はスコープロックによって保護されている
  - ロックの範囲はロックの期間によって決定される
  - ロックの範囲から出るにはデストラクターを呼び出し、例外時も安定させる
  - ロックを短くして競合状態を防ぐ
- さまざまな mutex の動作が利用可能
  - スピン vs. キュー (非スケーラブル、アンフェア vs. スケーラブル、フェア)
    - > spin_mutex, queuing_mutex
  - ライター vs. リーダー/ライター (複数のリーダーと単一のライターをサポート)
    - > spin_rw_mutex, queuing_rw_mutex
  - ネイティブな排他制御関数のスコープラッパー
    - > mutex (Windows: CRITICAL_SECTION, Linux: pthreads mutex)

同期プリミティブ

- mutex の特徴
  - スケーラブル: mutex アクセスの直列化よりも優れている
  - “フェア”: スレッド要求の順序を保存して何もしない状態を防ぐ
  - リエントラント: ロックはスタック可能で、提供されている動作ではサポートされていない
  - ウェイトメソッド: スレッドがロックを待つ方法

	スケーラブル	“フェア”	ウェイト
mutex	OS 依存	OS 依存	スリープ
spin_mutex	×	×	スピン
queuing_mutex	○	○	スピン
spin_rw_mutex	×	×	スピン
queuing_rw_mutex	○	○	スピン

- 存在期間がある mutex では、デッドロックまたはコンボイが発生する
  - ロックの期間を減らすか、atomic 操作を使用してこれらの問題を回避する
    - > atomic<T> は、整数型の読み取り-修正-書き込みサイクルのハードウェア・ロックをサポート
    - > セマフォー、参照カウントのような単一のネイティブ型の更新に役立つ

スコープロックの構文

- mutex の種類を選択して mutex を作成する
  - mutex の存在期間は mutex を使用するロックの期間以上にする
  - ファイルの範囲で mutex を定義するのに最も安全
- 同期が必要な場所で mutex を使用してロックする

```
{
    typedef spin_mutex MyMutexType;
    MyMutexType MyMutex;
    {
        MyMutexType::scoped_lock lock(MyMutex);
        // MyMutex 保護操作
    }
}
```

- スコープロックは、必要であれば他の操作を提示する

```
MyMutexType::scoped_lock lock();
lock.acquire(MyMutex);                    lock.release();
lock.try_acquire(MyMutex);
```

parallel_sort

- このテンプレートも 2 つの形式がある

```
template <typename RandomAccessIterator> inline void
parallel_sort(RandomAccessIterator begin, RandomAccessIterator end);

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

  – 前者は std::less<RandomAccessIterator 値の型> を使用する
- parallel_sort 分割コンストラクターはクイックソート分割を行う
  – 分割の粒度のしきい値は 500 エントリー
  – 500 未満のエントリーのチャンクは std:sort にそのまま送られる
  – 分割コンストラクターは範囲中のすべてのデータも処理する
    ＞キャッシュライン処理の優れた例ではない

parallel_scan

- 呼び出された*並列プリフィックス* は、次の回帰パターンに基づいてシーケンスを別のシーケンスに変換する

  $y_0 = I_\oplus \oplus x_0$ $\quad x_{0,}\ \ x_0 \oplus x_{1,}\ \ x_0 \oplus x_1 \oplus x_{2, \ldots}$

  $y_i = y_{i-1} \oplus X_i$

- 演算子 ⊕ は結合で、parallel_reduce のように浮動小数点を処理する場合は特に注意が必要
- TBB は 2 つのパスで実装されている
  - 1 つめのパスは、リダクションのように、部分的な結果を累積する
  - 結果は 2 つめのパスで分配される
  - スレッドの分割は、操作の数が増加しても実行時間を減らす

コンカレント・コンテナー

- インテル® TBB には、コンカレント・コンテナーが用意されている
  - STL コンテナーは並列操作では安全ではない
    - > 複数の修正を同時に行うと問題が発生する
  - 標準的手法: STL コンテナーのアクセスをロックでラップする
    - >アクセサーが一度に 1 つのみ操作するように制限する
- TBB は、競合が少ない場合に効率的なファイングレイン(細粒度)・ロックを提供
  - シングルスレッドのパフォーマンスは低下するが、スケーラビリティーは向上する
  - TBB、OpenMP*、またはネイティブスレッドとともに使用可能
  - スピンロック・ベースで、長い期間待つことは想定していない

コンカレント･インターフェイスの要件

- いくつかの STL インターフェイスは並列性をサポートするように設計されていない
  - –例えば、2 つのスレッドがそれぞれ別のものを実行すると仮定する

```
extern std::queue q;
if(!q.empty()) {
    item=q.front();
    q.pop();
}
```

この時点で、別のスレッドが最後の要素をポップする可能性がある

- ソリューション: インテル® TBB の concurrent_queue で pop_if_present() メソッドを使用する

concurrent_vector<T>

- *T* のダイナミックに拡張可能な配列
  - grow_by(n)
  - grow_to_at_least(n)
- クリアされるまで要素を移動しない
  - 同時にアクセスして拡張可能
  - clear() メソッドは、アクセス/サイズ変更に関してスレッドセーフではない
  - Range コンストラクターは parallel_for などに並列ベクトルを接続する

例

```
// スレッドセーフな方法でシーケンス [begin,end) を x に追加

template<typename T>
void Append(concurrent_vector<T> &x, const T *begin, const T *end ) {
    std::copy(begin, end, x.begin() + x.grow_by(end-begin) )
}
```

concurrent_queue<T>

- ローカルの要素の順序を保存
  - あるスレッドがプッシュし、別のスレッドが 2 つの値をポップすると、プッシュした順序でポップされる
- 2 種類のポップ
  - ブロック pop()
  - 非ブロック pop_if_present()
- size() メソッドは *符号付き* 整数を返す: (push – pop)
  - size() が –*n* を返した場合、*n* ポップが対応するプッシュを待つことを意味する
- 注意: キューはキャッシュを冷却する

concurrent_hash<Key,T,HashCompare>

- ハッシュテーブルは読み取りと更新の同時アクセスを許可
  - bool insert(accessor &result, const Key &key) 追加または編集
  - bool find(accessor &result, const Key &key) 編集
  - bool find(const_accessor &result, const Key &key) 調査
  - bool erase(const Key &key) 削除
- リーダーロックは共存可能 – ライターロックは排他的

例: 文字列から整数へのマップ

```
struct MyHashCompare {
    static long hash( const char* x ) {
        long h = 0;
        for( const char* s = x; *s; s++ )
            h = (h*157)^*s;
        return h;
    }

    static bool equal( const char* x, const char* y ) {
        return strcmp(x,y)==0;
    }
};

typedef concurrent_hash_map<const char*,int,MyHashCompare> StringTable;

StringTable MyTable;

void MyUpdateCount( const char* x ) {
    StringTable::accessor a;
    MyTable.insert( a, x );
    a->second += 1;
}
```

複数のスレッドが同時にエントリーを挿入または更新できる

アクセサー・オブジェクトは、スマートポインターとライターロックとして動作する

parallel_while

- 項目のストリームを制御するために提供されている

```
parallel_while<ApplyProcess> w;
ItemStream stream;
ApplyProcess body;
w.run(stream, body);
```

  – ストリームアクセスはスケーラビリティーを制限する – フェッチは直列に行われる

```
class ApplyProcess {    public:
    void operator() (Item *item) const {
        // process item->data
        w.add(item->data->left); w.add(item->data->right);
    }
    typedef Item *argument_type;
}
```

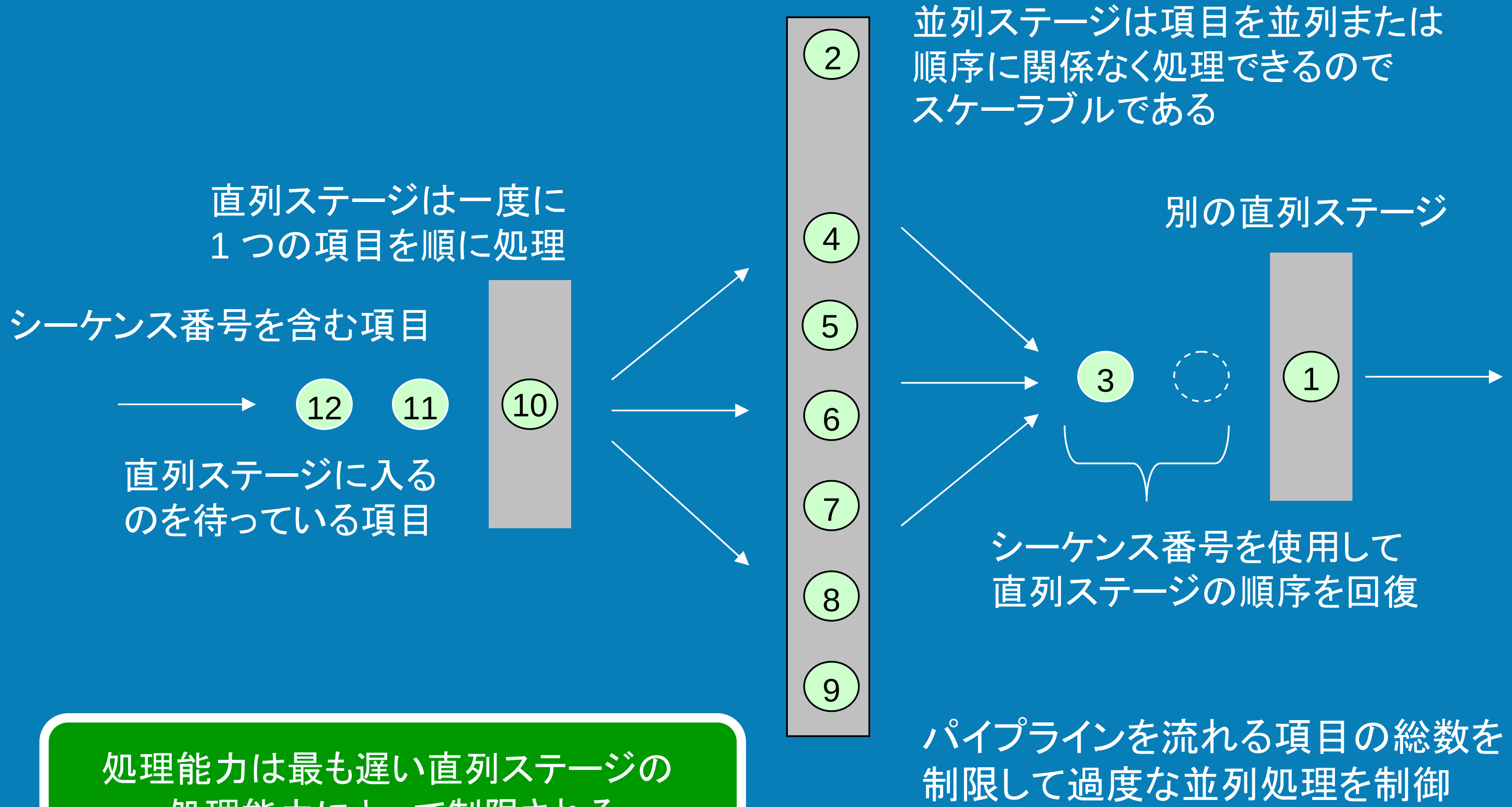
  – ストリームのボトルネックは、add メソッドを使用してタスク実行の前の拡張に再帰を追加することで回避できる

並列パイプライン

- ステージ化された線形パイプライン
  - 処理できる項目の最大値を指定
  - 線形パイプラインにマップして任意の DAG を制御
- 各ステージは*直列*または*並列*
  - 直列ステージは一度に 1 つの項目を順に処理
  - 並列ステージは一度に複数の項目を順序に関係なく処理
- キャッシュを効率的に使用
  - 各スレッドはできるだけ多くのステージを通って項目を処理
  - 新しい項目に移る前に古い項目を終了するようにバイアスがかけられる
  - コードがコアからコアへ移動する際にデータが変わらないようにする

DAG = 有向非循環グラフ (Directed Acyclic Graph)

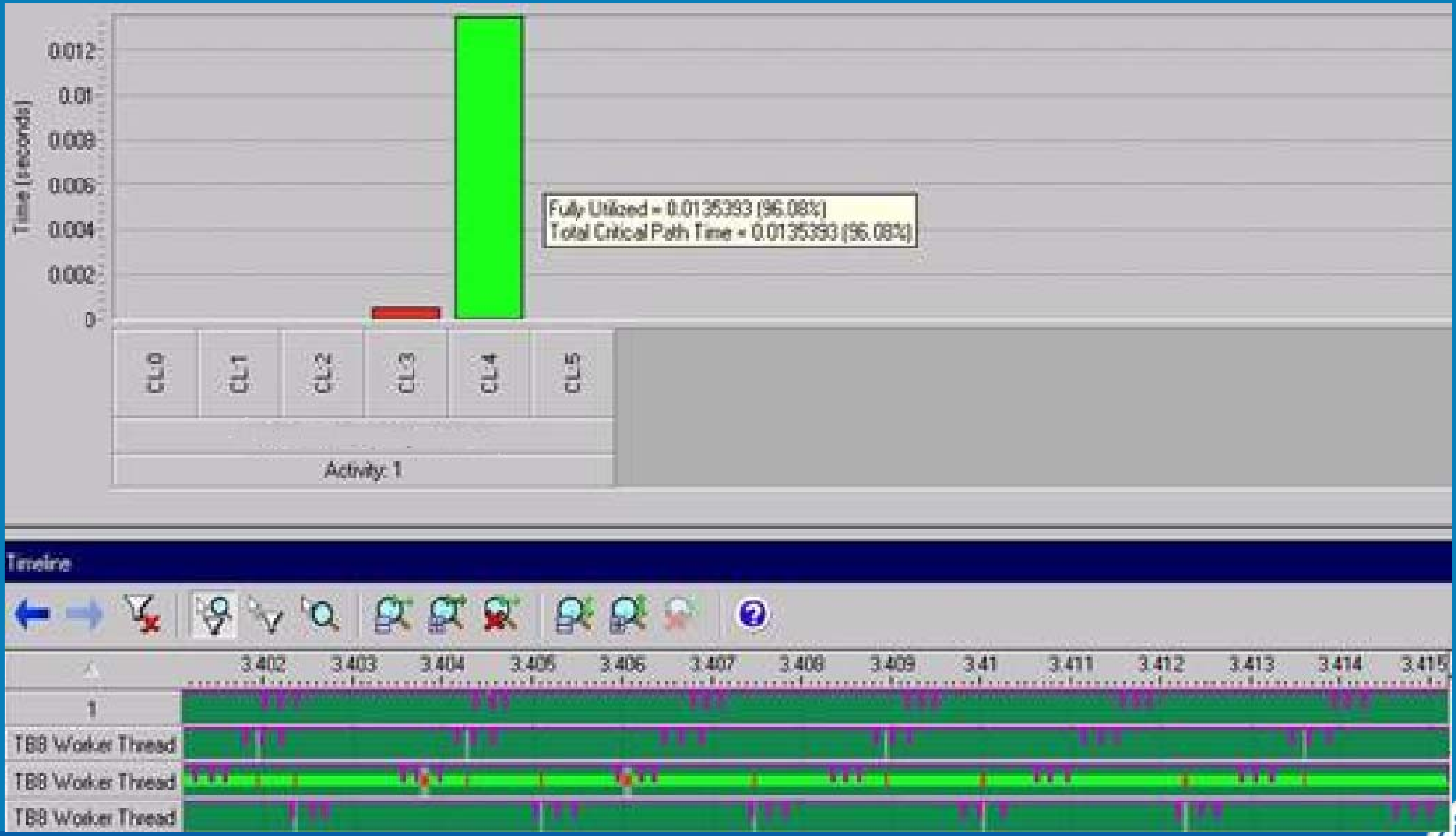
並列パイプライン



処理能力は最も遅い直列ステージの
処理能力によって制限される

パイプラインは効率的

- サンプルのインテル® TBB パイプラインはオーバーラップした I/O 処理を行う



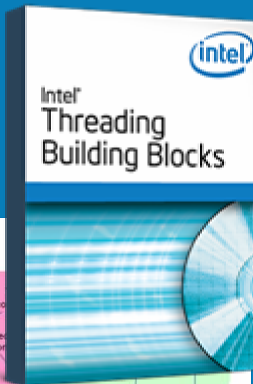
マルチコア・プロセッサー時代をリードするインテル® ソフトウェア開発ツール

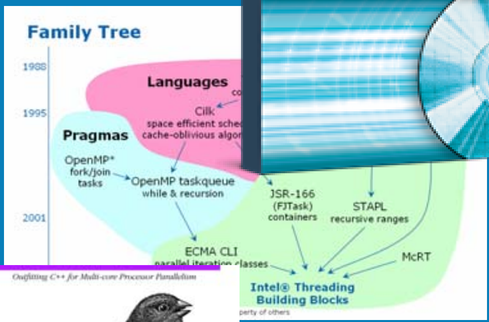
intel®

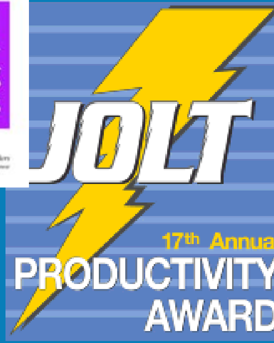
さあ、その先へ。™

インテル® TBB

基本的なアルゴリズム:	parallel_for, parallel_reduce, parallel_scan
高度なアルゴリズム:	parallel_while, pipeline, parallel_sort
コンテナー:	concurrent_queue, concurrent_vector, concurrent_hash_map
スケーラブルなメモリー割り当て:	scalable_malloc, scalable_free, scalable_realloc, scalable_calloc, scalable_allocator, cache_aligned_allocator
排他制御:	
mutex:	mutexspin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex
アトミック操作:	fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store
時間計測:	移植性のあるファイングレイン (細粒度) のグローバル・ タイム・スタンプ
タスク・スケジューラー:	タスクの作成とアクティベーションを制御する直接アクセス







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。
製品に付属の売買契約書『Intel's Terms and conditions of Sales』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) に関しても一切責任を負わないものとします。

インテル製品は、予告なく仕様が変更されることがあります。

* 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。