インテル® TBB による効率良いスレッド化

はじめに

このガイドでは、インテル® Parallel Studio XE に含まれるインテル® TBB を使用して効率良くスレッド化する方法を説明します。インテル® TBB は広く使用されている C++ テンプレート・ライブラリーであり、安定性を備え、移植性とスケーラビリティーに優れた並列アプリケーションの作成を支援します。インテル® TBB を利用することで、さまざまな環境でマルチコアおよびメニーコア・プロセッサーの能力を最大限に活用し、パフォーマンスを引き出すことができるだけでなく、保守も容易な優れたタスクベースの並列アプリケーションを簡単に短期間で開発できます。

最初に、サンプルコードを例に、インテル® TBB のパワフルな機能を説明します。次に、インテル® Parallel Studio XE を活用した 6 つのプロセスによりアプリケーションを並列化する手順を説明します。最後のセクションには、スレッド化に役立つ重要な情報が含まれています。

インテル® TBB の parallel_for テンプレートを含む数行のコードを追加するだけで、パフォーマンスが最大 1.59 倍になります (Adding_Parallelism サンプルコードでシングルスレッドから 2 スレッドにした場合)。ただし、実際の結果は異なる可能性があります。このガイドの演習後、ご自身のコードで試してみてください。以下は、関数をシリアルから並列に変えた場合の変更前と変更後のコードです (図 1a と 1b)。

```
void change_array(){

//Instructional example - serial version
  for (int i = 0; i < list_count; i++) {
        data[i] = busyfunc (data[i]);
    }
}
```

図 1a

```
void parallel_change_array(){

//Instructional example - parallel version
parallel_for (blocked_range<int> (0, list_count),
 [=](const blocked_range<int>& r) {
   for (int i = r.begin(); i < r.end(); i++) {
         data[i] = busyfunc (data[i]);
}
});
```

図 1b

デモ: 並列処理の実装

インテル® TBB は、並列処理を実装するための「ビルディング・ブロック (積み木)」の集合です。C++ テンプレートにより、一般的なプログラミング・パターンで利用できる強力な並列化機能を提供します。例えば、インテル® TBB の parallel_for 構文は、標準的なシリアル for ループを、並列 for ループに変換できます。parallel_for は、最も簡単で頻繁に使用されるビルディング・ブロックです。これまで並列処理を実装したことがない開発者は、まずこの構文から始めてみてください。

インテル® TBB を使用する理由: 移植性、信頼性、スケーラビリティー、容易さ

- **移植性:** インテル® TBB のスレッド API は、32 ビットおよび 64 ビットの Windows*、Linux*、OS X* プラットフォームをはじめ、オープンソース版の FreeBSD*、IA Solaris*、QNX、Xbox* 360 などで利用できます。
- **オープンデザイン:** コンパイラー、オペレーティング・システム、プロセッサーに依存しません。
- **フォワード・スケーリング:** 開発したバイナリーはコードを変更/再コンパイルすることなく、利用可能なコア数に応じて自動的にスケーリングします。
- **統合されたソリューション:** 同期プリミティブ、スケーラブルなメモリー割り当て、並列アルゴリズムとタスク制御、コンカレント・コンテナーが含まれます。
- **ライセンス:** 商用およびオープンソース版があります。詳細は、以下のリンクを参照してください。

- **製品:** インテル® Parallel Studio XE に同梱されています。単一パッケージ、オープンソース版も提供されています。

詳細は、商用版またはオープンソース版の Web サイトをご覧ください。

実装例

ここでは、インテル® TBB の parallel_for を使ったサンプルを紹介します。この演習をお読みになった後で、ここで紹介する 4 つのステップと Adding_Parallelism サンプルコードを実際に試してみてください。

ステップ 1: インテル® Parallel Studio XE のインストールと設定

ソフトウェア要件: サポートしているバージョンの Microsoft* Windows* および Microsoft* Visual Studio* 2010 以上。本書の説明は、Microsoft* Visual Studio* 2010 用のものです。

推定所要時間: 15 ～ 30 分

1. インテル® Parallel Studio XE の評価版をダウンロードします。
2. インテル® Parallel Studio XE をインストールします (システムにより異なりますが、約 15 ～ 30 分かかります)。

ステップ 2: Adding_Parallelism サンプル・アプリケーションのインストールと参照

サンプル・アプリケーションの準備:

1. サンプルファイル Adding_Parallelism_Exercise.zip をダウンロードします。このサンプルは、Microsoft* Visual Studio* を使用して作成された C++ コンソール・アプリケーションです。
2. Adding_Parallelism_Exercise.zip ファイルをシステムの書き込み可能なフォルダー (例えば、**マイ ドキュメント\Visual Studio 2010\Intel\samples** フォルダー) に展開します。

サンプルの表示:

3. **[ファイル] > [開く] > [プロジェクト/ソリューション]** を選択して、ソリューションを Microsoft* Visual Studio* にロードします。ファイルを展開したフォルダーにある Adding_Parallelism.sln ファイルを選択します (図 2)。

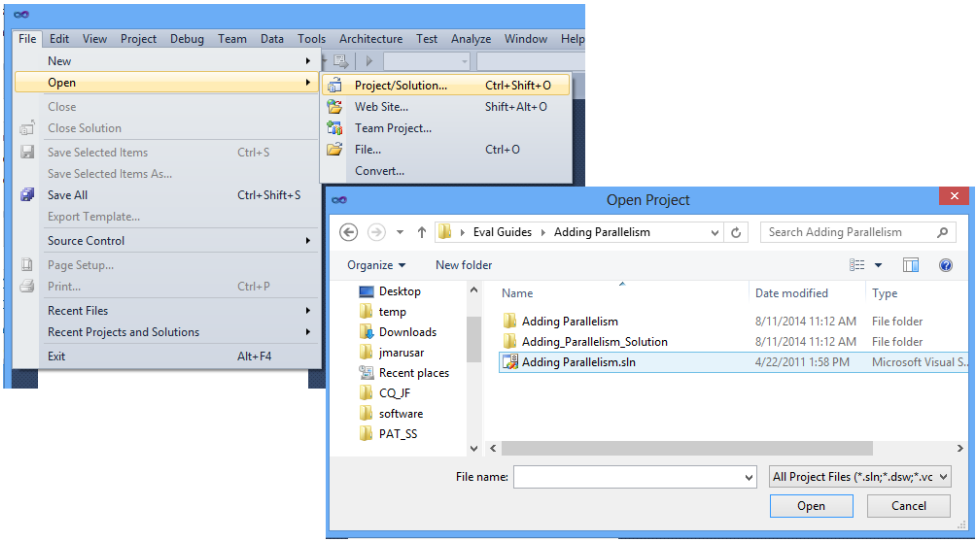

図 2

4. このソリューションには 2 つのプロジェクトが含まれています。1 つ目のプロジェクト Adding_Parallelism には、シリアルバージョンのサンプルコードとインテル® TBB の例が含まれています。2 つ目の Adding_Parallelism_Solution には、インテル® TBB 向けに変更されたサンプルコード (並列バージョン) が含まれています (図 3)。

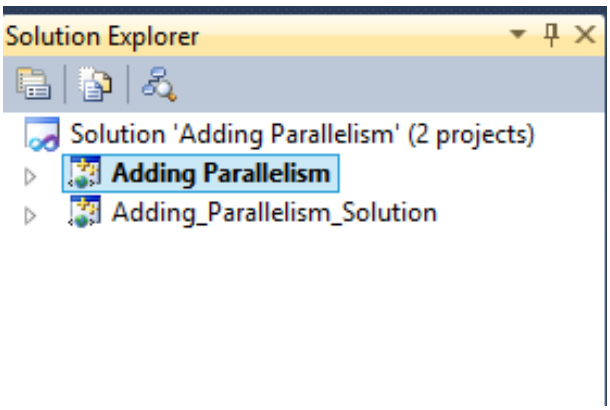

図 3

5. 必要に応じて、**[プロジェクト] > [Intel Compiler XE 15.0 (インテル(R) コンパイラー XE 15.0)] > [Use Intel C++ (インテル(R) C++ を使用)]** を選択して、Adding_Parallelism プロジェクトがインテル® C++ コンパイラーを使用するように設定します (図 4)。

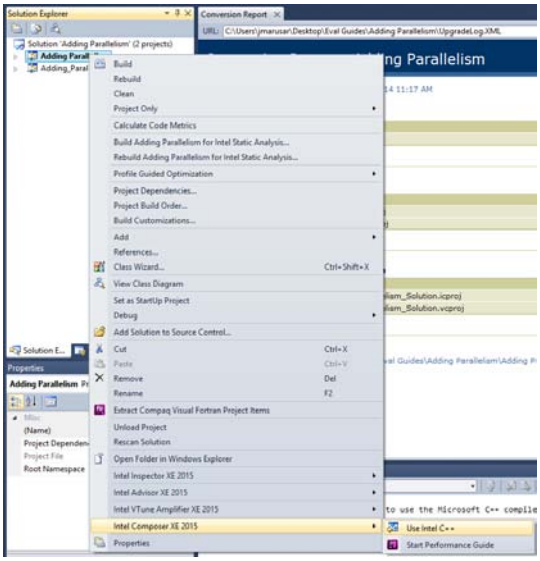

図 4

6. **[プロジェクト] > [プロパティ] > [構成プロパティ] > [Intel Performance Libraries (インテル(R) パフォーマンス・ライブラリー)] > [Use TBB (インテル(R) TBB を使用]** で **[Yes (はい)]** を選択して、Adding_Parallelism プロジェクトがインテル® TBB を使用するように設定します (図 5)。

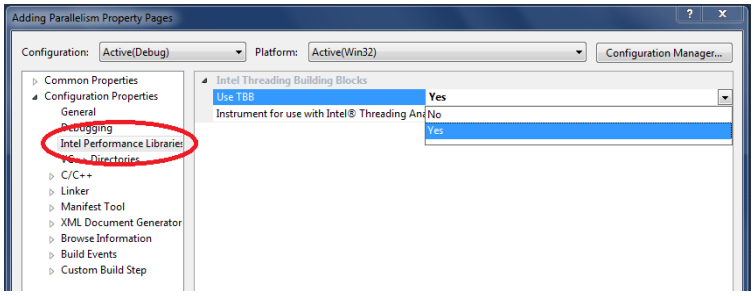

図 5

7. **[プロジェクト] > [プロパティ] > [C/C++] > [Language [Intel C++] (言語 [インテル(R) C++])] > [Enable C++11x Support (C++11x サポートを有効にする)]** で **[Yes (はい)]** を選択して、Adding_Parallelism プロジェクトがラムダ式を使用するように設定します (図 6)。

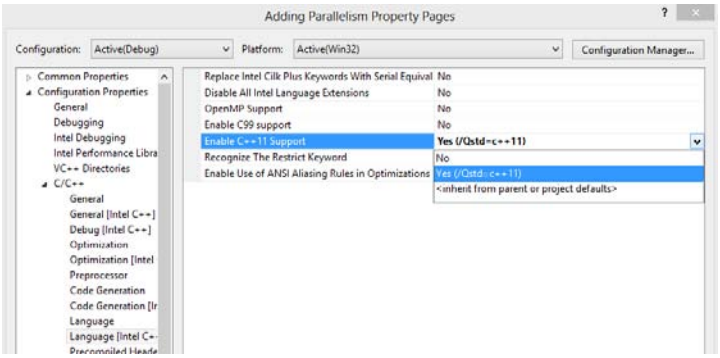

図 6

ステップ 3: インテル® TBB の parallel_for を使用した find_primes 関数の並列化

1. ヘッダーファイルがすでにインクルードされていることに注意してください。インテル® TBB の parallel_for を使用するには、tbb/parallel_for.h と tbb/blocked_range.h をインクルードする必要があります。
2. find_primes 関数のコピーを作成して、名前を parallel_find_primes に変更します。関数の戻り型や引数リストを変更する必要はありません。図 7 にオリジナル (シリアルバージョン) の find_primes 関数を示します。

```
void find_primes(int* &my_array, int *&prime_array) {
    int prime, factor, limit;
    for (int list=0; list < list_count; list++) {
      prime = 1;
      if ((my_array[list] % 2) ==1) {
        limit = (int) sqrt((float)my_array[list]) + 1;
        factor = 3;
        while (prime && (factor <=limit)) {
            if (my_array[list] % factor == 0) prime = 0;
            factor += 2;
        }
      } else prime = 0;
      if (prime) {
        prime_array[list] = 1;
      }
      else
        prime_array[list] = 0;
    }
}
```

図 7

3. parallel_find_primes 内部で parallel_for を呼び出します。parallel_change_array 関数の呼び出しが参考になるでしょう。ここで提供されるコードや Adding_Parallelism_Solution プロジェクトで提供されるコードを使用してもかまいません。parallel_for は、シリアル for ループを並列に実行する複数のスレッドに分配します。parallel_for には、2 つの引数があります (以下のステップ 4 と 5 で説明します)。parallel_find_primes 関数は図 8 のようになります。

```
void parallel_find_primes (int *&my_array, int *& prime_array){
      parallel_for (
```

図 8

parallel_for の仕組み

parallel_for は、インテル® TBB で最も簡単で一般的に使用されるテンプレートです。シリアル for ループの作業を複数のタスクに分割した後、実行時に利用可能なすべてのプロセッサー・コアに対してタスクを分配します。parallel_for を使うことで、スレッドの細かな制御についてではなく、アプリケーションのアルゴリズムに注力できます。必要なことは、シリアル for ループの反復が独立していることを保証するだけです。独立している場合、parallel_for を使用できます。

インテル® TBB は、利用可能なプロセッサー・コアの数に応じて適切な大きさのスレッドプールで、スレッドの生成、終了、ロードバランスを管理します。タスクはスレッドに分配されます。この実装モデルは、オーバーヘッドを減らし、将来も利用できるスケーラビリティーを保証します。インテル® TBB は利用可能なプロセッサー・コアを最大限に活用するようにスレッドプールを作成します。

このガイドではデフォルトの設定で parallel_for を使用していますが、アプリケーションが最良のパフォーマンスを得られるように、いくつかの調整可能なパラメーターが用意されています。また、ここではラムダ式形式で表現されていますが、C++11x 標準をサポートしないコンパイラーを利用する場合は別の形式で表現できます。

4. blocked_range を第 1 引数として渡します。blocked_range は、インテル® TBB に含まれている型で、for ループの範囲を指定します。parallel_for を呼び出すと、blocked_range の範囲はオリジナルのシリアルループと同じになります (この例では 0 から list_count)。parallel_for は多くのタスクを作成し、それぞれのタスクは指定された範囲の一部を処理します。インテル® TBB のスケジューラーは、これらのタスクに個別のより小さな blocked_range を割り当てます。parallel_find_primes 関数は図 9 のようになります。

```
void parallel_find_primes (int *&my_array, int *& prime_array){
  parallel_for (blocked_range<int>(0,list_count),
```

図 9

5. for ループの本体をラムダ式で記述し、第 2 引数として渡します。この引数は、各タスクの処理を指定します。for ループは分割され、複数のタスクによって実行されるため、for ループの範囲を各タスクに割り当てる範囲 (<range>.begin() および <range>.end()) に変更する必要があります。

   また、このラムダ式では各タスクの処理も定義する必要があります。ラムダ式を使用すると、コンパイラーは、インテル® TBB のテンプレート関数で使用可能な関数オブジェクトを作成できます。ラムダ式は、コードで動的に指定できる関数です (lisp のラムダ関数、あるいは .NET の匿名関数の概念に似ています)。

   下記のコードでは、[=] によりラムダ式が有効になります。[&] の代わりに [=] を使用すると、ラムダ式の外部で宣言される変数 list_count と my_array は、関数オブジェクト内のフィールドとして**値渡し**されなければなりません。[=] の後に、生成される関数オブジェクトの operator() のパラメーター・リストと宣言を指定します。変更後の parallel_find_primes 関数は図 10 のようになります。

```
void parallel_find_primes(int *&my_array, int *& prime_array) {
    parallel_for (blocked_range<int>(0, list_count),
        [=](const blocked_range<int>& r) {
            int prime, factor, limit;
            for (int list=r.begin(); list < r.end(); list++) {
                prime = 1;
                if ((my_array[list] % 2) ==1) {
                    limit = (int)
                    sqrt((float)my_array[list]) + 1;
                    factor = 3;
                    while (prime && (factor <=limit)) {
                        if (my_array[list] % factor == 0) prime = 0;
                        factor += 2;
                    }
                }
                else prime = 0;
            if (prime)
                prime_array[list] = 1;
            else
                prime_array[list] = 0;
            }
    });
}
```

図 10

6. parallel_find_primes 関数の時間を測定するように main 関数を変更します。インテル® TBB の tick_count オブジェクトを用いて時間を測定します。tick_count は、スレッドセーフかつスレッドアウェアなタイマーです。parallel_find_primes を呼び出して時間を測定するコードを下記に示します。main コードでこのほかの変更は必要ありません (図 11)。

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

parallel_find_primes(data, isprime);

tick_count parallel_prime_end=tick_count::now();

cout << "Time to find primes in parallel for " << list_count << " numbers: " <<
(parallel_prime_end - parallel_prime_start).seconds()
  << " seconds."<< endl;
```

図 11

ステップ 4: 並列バージョンのビルドと速度向上の確認

1. **[Build (ビルド)] > [Build Solution (ソリューションのビルド)]** でプログラムをビルドします (図 12)。

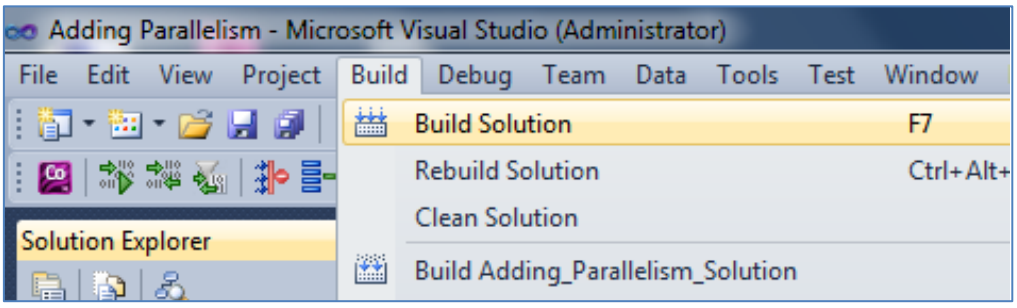

図 12

2. Microsoft* Visual Studio* で **[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)]** を選択して、アプリケーションを実行します (図 13)。

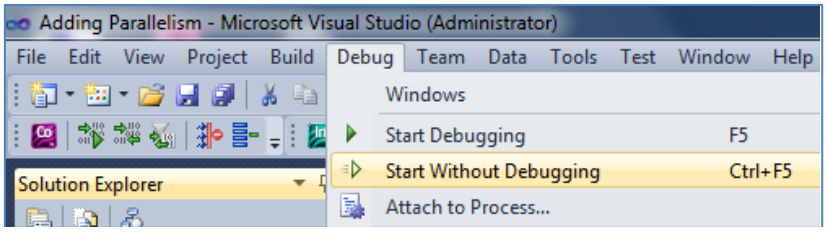

図 13

3. マルチコアシステムで実行している場合、大幅に高速化されるでしょう。正確に時間を測定するため、シリアルバージョンと並列バージョンを別々に実行します。

   注: パフォーマンス測定のため、Release 構成で Adding_Parallelism プロジェクトを設定、ビルド、実行してください (図 14 (シリアル) と図 15 (並列) を参照)。

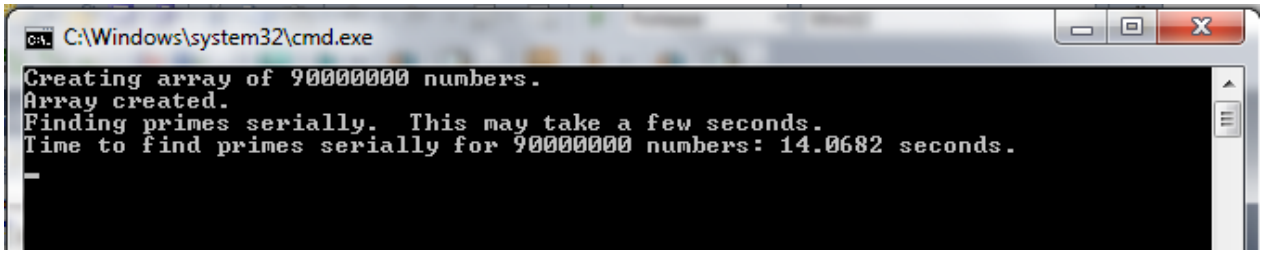

図 14

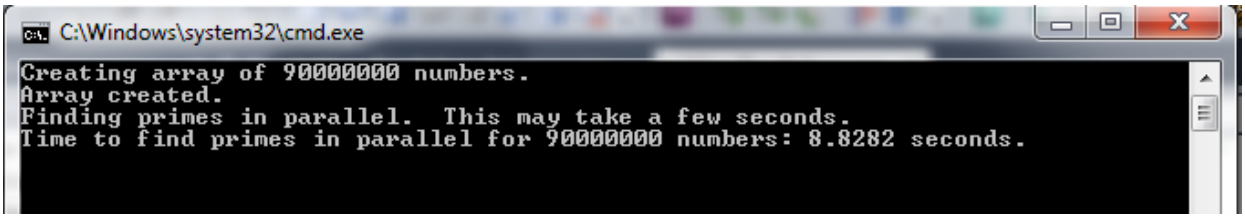

図 15

結果

この例では、for ループを parallel_for に変更して (その他のチューニングを行うことなく) 大幅にパフォーマンスを向上させる方法を説明しました。この例のスケーラビリティーはほぼ完璧ですが、一般に、parallel_for による速度の向上は、使用しているアルゴリズムとデータ構造に依存します。多くの場合、インテル® VTune™ Amplifier XE でチューニングすることで、スケーラビリティーがさらに向上します。

以下の表は、デュアルソケットのインテル® Core™ i7 プロセッサー (1.6 GHz、4 コア、4GB RAM)、Microsoft* Windows* 7、インテル® Parallel Studio XE Update 1、Microsoft* Visual Studio* 2010 を搭載したラップトップで、90,000,000 の数列から素数を検索した場合の結果です (図 16)。

Number of cores	Run time (1st version of find primes only)	Speedup over serial
1	11.95s	--
2	6.02s	1.99x
4	3.02s	3.96x
8	1.52s	7.86x

図 16

まとめ: コードを並列化するための 6 つのステップ

並列化はパフォーマンスを大幅に向上させる可能性があります。特に計算負荷の高いアプリケーションではその可能性は広がります。しかし、商用ソフトウェアの並列化は演習サンプルのように単純に行えるものではありません。インテル® Parallel Studio XE のコンポーネントは、通常のアプリケーションのスレッド化、デバッグ、チューニングの複雑さを減らすことを目的に設計されています。parallel_for を自身のコードで使用するには、その場所を最初に決定する必要があります。下記の手順を参考にしてください。

1. **hotspot を特定する**
   インテル® VTune™ Amplifier XE で hotspot 解析を実行して、アプリケーションで最も時間を費やしている関数を確認します。
2. **計算負荷の高い for ループを調べる**
   最も時間を費やしている関数をダブルクリックしてコードを表示し、ループを調べます。
3. **選択したループの依存性を確認して切り離す**
   ループの反復を少なくとも 3 回逆順に実行してみます。問題なく動作する場合は、ループ反復間のデータ依存はないと考えられます。
4. **インテル® TBB の parallel_for に変換する**
   外側のループを変更して (入れ子の場合)、parallel_for を (可能であればラムダ式で) 実装します。
5. **インテル® Inspector XE で正当性を検証する**
   インテル® Inspector XE でスレッドエラー解析を実行して、並列化したコードにデータ競合がないことを確認します。
6. **パフォーマンスを測定する**
   シリアル実行と並列実行を比較して、並列化による速度向上を計算します。

parallel_for による最適化に適さないケース

コードに計算負荷の高いループが含まれていない場合は、前述のステップ 2、3、4 でインテル® TBB の別のオプションを利用できます。Adding_Parallelism サンプルコードには、parallel_reduce に変換できる関数も含まれています。parallel_reduce は parallel_for に似たテンプレートで、ループから値 (最小、最大、合計、見つかったインデックスなど) を返すことができます。インテル® TBB は、ソート、パイプライン化、再帰、フローグラフのような複雑なアルゴリズムもサポートしています。

並列処理の主な概念に関する情報

インテルでは、開発者が現在および将来のプロセッサー処理能力を活用する、正確で高性能なコードを記述できるように、並列処理に関するさまざまな情報を提供しています。インテル® Parallel Studio XE およびその他の関連項目についてインテル社のエキスパートが提供している情報をご活用ください。

関連情報

theadingbuildingblocks.org – インテル® TBB オープンソース Web サイト

ラーニングラボ – テクニカルビデオ、ホワイトペーパー、Webinar の再生など

インテル® Parallel Studio XE 製品ページ – HOW TO ビデオ、入門ガイド、ドキュメント、製品の詳細情報、サポートなど

評価ガイド – さまざまな機能の使用法を紹介する評価ガイド

30 日間の評価版のダウンロード

購入方法: 言語別のスイート

インテル® Parallel Studio XE には、開発のニーズに応じて 3 つのエディションがあります。Composer Edition と Professional Edition では、C++ または Fortran のいずれかの言語で利用できます。

- **Composer Edition:** 高速な並列コードを構築するためのコンパイラー、パフォーマンス・ライブラリー、並列モデルが含まれています。
- **Professional Edition:** Composer Edition の機能に加えて、高速な並列コードの設計、ビルド、デバッグ、チューニング用にパフォーマンス・プロファイラー、スレッド設計/プロトタイピング・ツール、メモリー/スレッドデバッガーが含まれています。
- **Cluster Edition:** Professional Edition の機能に加えて、MPI を含む高速な並列コードの設計、ビルド、デバッグ、チューニング用に MPI クラスター通信ライブラリー (MPI エラーチェックおよびチューニング・ユーティリティー付き) が含まれています。

	インテル® Parallel Studio XE Composer Edition[1]	インテル® Parallel Studio XE Professional Edition[1]	インテル® Parallel Studio XE Cluster Edition
インテル® C++ コンパイラー	√	√	√
インテル® Fortran コンパイラー	√	√	√
インテル® TBB (C++ のみ)	√	√	√
インテル® IPP (C++ のみ)	√	√	√
インテル® MKL	√	√	√
インテル® Cilk™ Plus (C++ のみ)	√	√	√
インテルによる OpenMP* 実装	√	√	√
ローグウェーブ IMSL* ライブラリー[2] (Fortran のみ)	バンドルおよびアドオン	アドオン	アドオン
インテル® Advisor XE		√	√
インテル® Inspector XE		√	√
インテル® VTune™ Amplifier XE[3]		√	√
インテル® MPI ライブラリー[3]			√
インテル® Trace Analyzer & Collector			√
オペレーティング・システム (開発環境)	Windows* (Visual Studio*) Linux* (GNU*) OS X*[4] (XCode*)	Windows* (Visual Studio*) Linux* (GNU*)	Windows* (Visual Studio*) Linux* (GNU*)

注:
1. C++ または Fortran のいずれか、あるいは両言語で利用できます。
2. Windows* Fortran スイートのアドオンまたは Composer Edition のバンドルとして利用できます。
3. スイートのバンドルまたはスタンドアロンとして利用できます。
4. OS X* の言語スイートとして利用できます。

インテル® Parallel Studio XE の詳細:

- 以下の Web サイトをご覧ください。 http://intel.ly/parallel-studio-xe
- あるいは、左の QR コードをスキャンしてください。

30 日間の評価版:

- http://intel.ly/sw-tools-eval の Web サイトで、「Product Suites」をクリックしてください。

著作権と商標について

最適化に関する注意事項

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項の適用対象である特定の命令セットに関する詳細は、該当する製品のユーザー・リファレンス・ガイドを参照してください。改訂 #20110804

JPN/1501/PDF/XL/SSG/KS introduce-parallelism-intel-tbb_studioxe-evalguide/Rev-081714