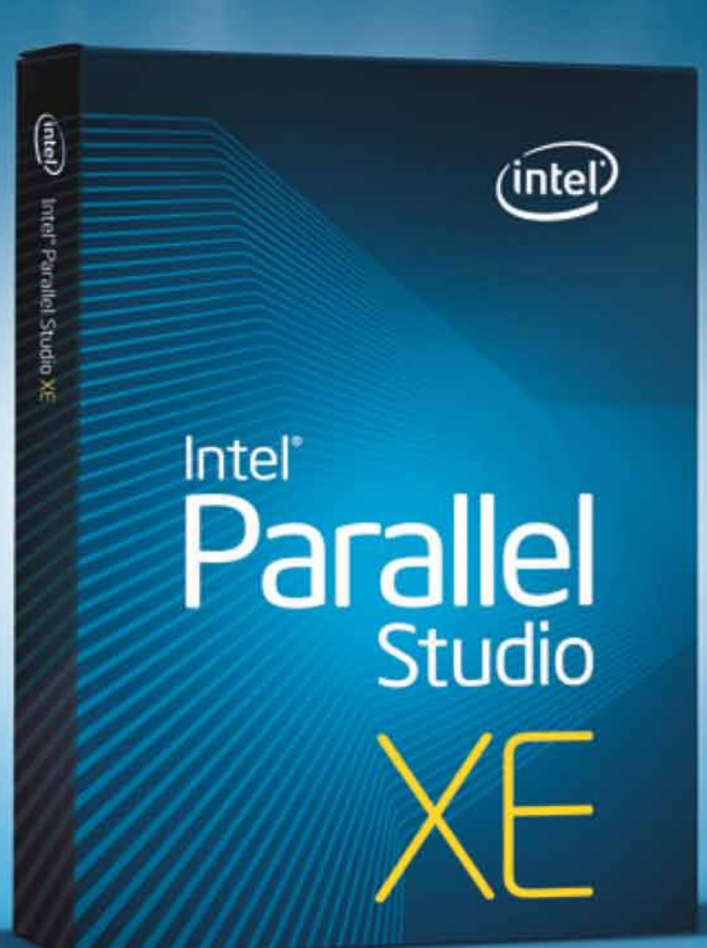
並列化による 既存プログラムの最適化

インテル® Parallel Studio XE Windows* 版



はじめに

このガイドでは、インテル® Parallel Studio XE に含まれる強力なスレッド・ライブラリーを使用して既存のアプリケーションを並列化する方法を説明します。最初に、サンプルコードを利用して、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) 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かもしれません。このガイドの演習後、自分のコードで試してみましょう。ここでは、シリアルから並列に変更する前後の関数の例を紹介します (コード例 1 と 2)。

コード例 1

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

コード例 2

```
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]);
       }
    });
}
```

インテル® Parallel Studio XE は、C++ および Fortran を利用する開発者向けに設計された統合ツールで、スケーラブルかつハイパフォーマンスなマルチコア・プロセッサー対応の並列アプリケーションの開発を、簡単なアプローチで迅速に行えます。

インテル® Composer XE 2011: 最適化コンパイラー、インテル® Parallel Building Blocks (インテル® PBB)、ハイパフォーマンス・ライブラリーが含まれています。

インテル® Inspector XE 2011: 優れたスレッド/メモリー・エラー・チェッカーです。

スタティック・セキュリティー解析 (SSA): セキュリティーの脆弱性をなくし、さまざまな不具合を排除します。

インテル® VTune™ Amplifier XE: 高度なパフォーマンス・プロファイラーです。

インテル® Parallel Building Blocks (インテル® PBB): マルチコアのパワーを十分活用できるよう支援します。既存のアプリケーションへ並列化を容易に実装する 3 つの並列プログラミング・アプローチを提供します。

- インテル® Cilk™ Plus: インテル® C/C++ コンパイラー固有の並列処理の実装です。インテル® Cilk™ Plus は、単純なループとタスクを使用して並列アプリケーションを作成する C++ 開発者向けです。ベクトル化機能と、高度なループベースのデータ並列処理およびタスク処理を組み合わせることで、優れた機能を提供します。
- 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): 汎用ループとタスクを使用して並列アプリケーションを作成するための C++ テンプレート・ライブラリーです。スケーラブルなメモリー割り当て、負荷分散、ワークスチール・タスク・スケジューリング、スレッドセーフなパイプラインとコンカレント・コンテナー、高度な並列アルゴリズム、さまざまな同期プリミティブが含まれます。
- インテル® Array Building Blocks (インテル® ArBB): 特定の並列メカニズムやハードウェア・アーキテクチャーに依存しない、ベクトル並列プログラミングの汎用ソリューションを提供します。ベクトル並列数値計算アルゴリズムを記述する C++ 開発者向けです。(2011 年 6 月時点 ベータ版)

本評価ガイドでは、インテル® TBB について主に説明します。

デモ：並列処理の実装

インテル® TBB は、並列化を実装するための「ビルディング・ブロック (積み木)」の集合です。C++ のテンプレートを利用することで、共通のプログラミング様式によ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 構文を使用すると、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、インテル® TBB で最も容易かつ頻繁に使用されるビルディング・ブロックです。並列処理の実装をこれまで行ったことがない開発者は、まずこの構文を使用することから始めてください。

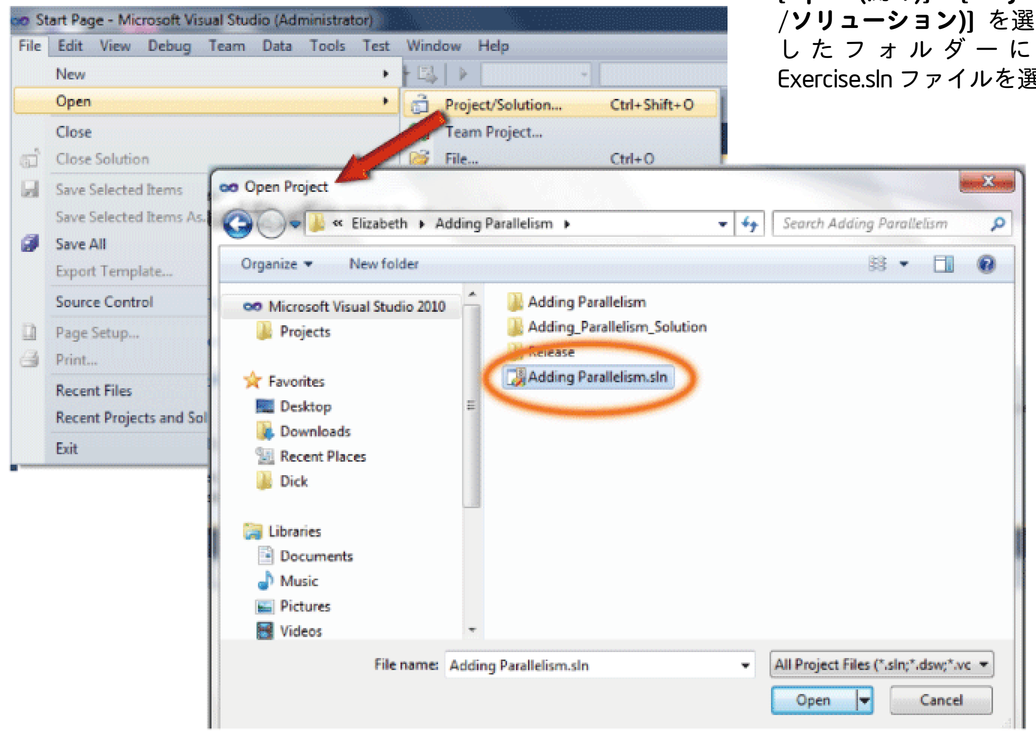
並列化による既存プログラムの最適化

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理由:
移植性、信頼性、スケーラブル、容易

- **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 Windows、Linux*、Mac 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:** プリミティブ、スレッド、スケーラブルなメモリー割り当てとタスク制御、並列アルゴリズム、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利用可能です。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、インテル® Parallel Studio XE およびインテル® コンパイラー・プロフェッショナル・エディション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。

図 1



実装例

ここでは、インテル® TBB の parallel_for を使用したサンプルを紹介します。ここで紹介する 4 つのステップとサンプルコード Adding_Parallelism を使用して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

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. parallel_studio_xe_2011_setup.exe をクリックしてインテル® 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: 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インストール:

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* 2005 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 20xx\Intel\samples** フォルダー) に展開します。

サンプルの表示:

1. Microsoft* Visual Studio* で、**[File (ファイル)]** > **[Open (開く)]** > **[Project/Solution (プロジェクト/ソリューション)]** を選択します。ファイルを展開したフォルダーにある Adding_Parallelism_Exercise.sln ファイルを選択します。図 1

2. 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。最初のプロジェクト Adding_Parallelism には、シリアル・サンプルコードとインテル® TBB の例が含まれています。2 つめの **Adding_Parallelism_Solution** には、インテル® TBB を使用するために変更されたサンプルコードが含まれています。図 2

3. どちらのプロジェクトも、インテル® C++ Composer XE を使用するように構成されています。この設定は、プロジェクト名を右クリックして **[Properties (プロパティ)]** を選択し、[Configuration Properties (構成プロパティ)] > [General (全般)] で確認できます。図 3

4. Adding_Parallelism.cpp のコードを確認するか、下記の説明を読んでください。

サンプルコードには、for ループを使用する 4 つの関数が含まれています。最初の 2 つは、change_array と並列バージョンの parallel_change_array です。この関数と並列バージョンは、parallel_for の単純な使用例で、このガイドでも紹介します。次の 2 つの関数はどちらもシリアルで、乱数の配列から素数を検索します。最初のバージョンは、素数が見つかった場所のコンパニオン配列に 1 を入れます。2 つめのバージョンは、素数が見つかると、カウンターの値をインクリメントして値を返します。

このガイドでは、find_primes の最初のバージョンを並列に変換します。2 つめの関数は変換が少し複雑で、ここでは説明しませんが、試してみてください。実装例が Solution として両方の関数のサンプルに含まれて

図 2

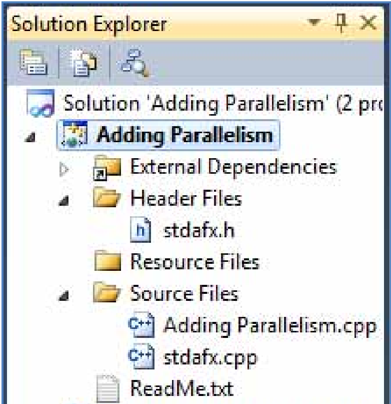
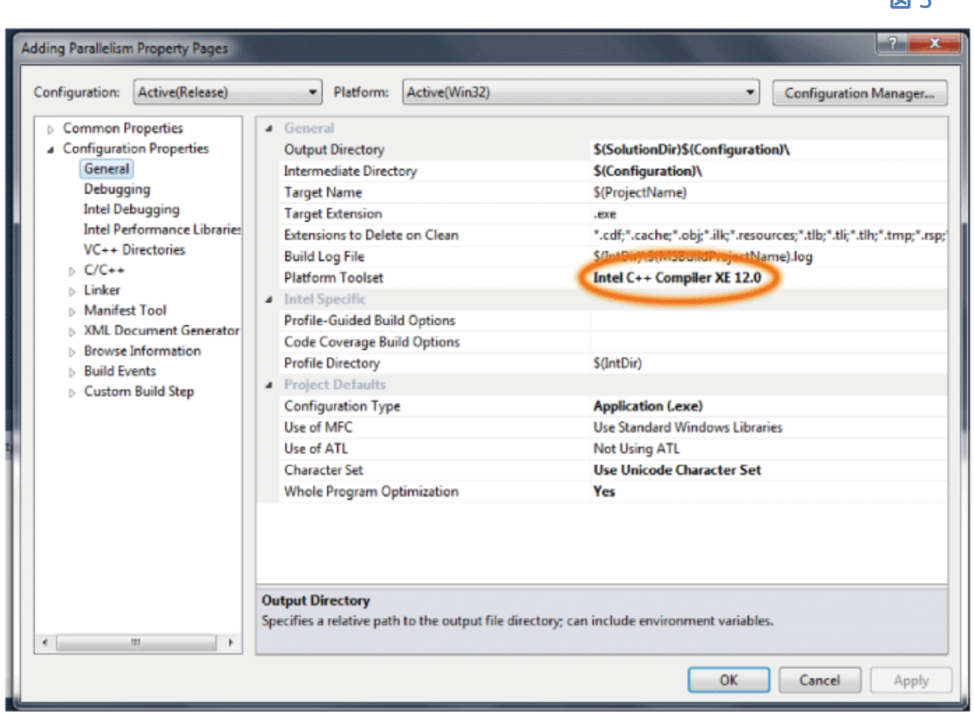


図 3



います。

5. これらのプロジェクトは、ラムダ式のサポートを含むインテル® TBB を使用するように構成されています。設定を確認するには、[Solution Explorer (ソリューション エクスプローラ)] の各プロジェクトを右クリックして [Properties (プロパティ)] を選択します (図 4 と 5)。詳細は、Adding_Parallelism.cpp の先頭のコメントを参照してください。

図 4

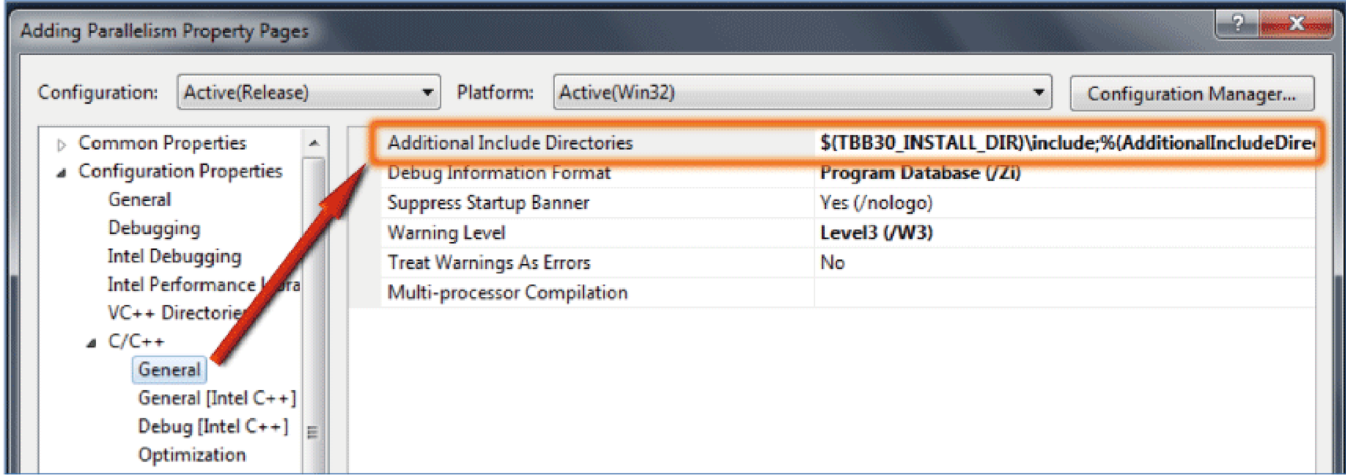
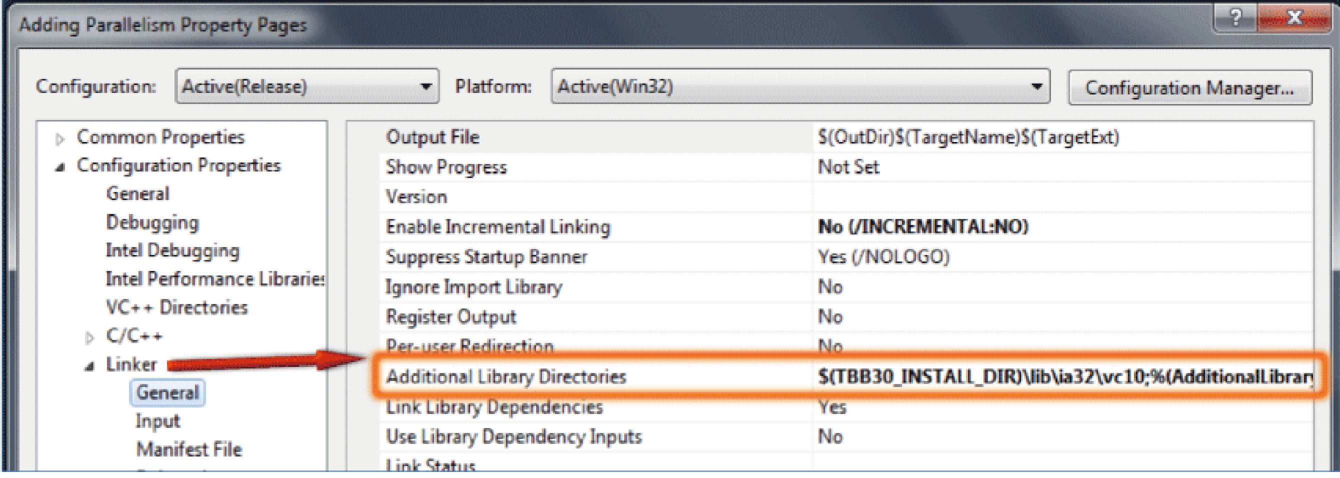


図 5



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。オリジナル (シリアル) の find_primes 関数は下記のようになります: **コード例 3**

3. parallel_find_primes 内部で parallel_for を呼び出します。parallel_ change_array 関数の呼び出しは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 4 と 5 で説明します)。parallel_find_primes 関数は下記のようになります: **コード例 4**

コード例 3

```
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;
   }
}
```

コード例 4

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

parallel_for はどのように動作するか

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 を使用することで、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・コアの数に応じた適切な大きさのスレッドプールを使用して、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。

ここで紹介する parallel_for の例はデフォルトの設定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るために開発者が使用できる、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。このサンプルはラムダ式形式で表示されています。C++ 0x 標準をサポートしないコンパイラーを利用する場合、別の形式で記述できます。

4. blocked_range を最初の引数として渡します。blocked_range は、for ループの範囲を指定するインテル® TBB に含まれている型です。parallel_for を呼び出すと、blocked_range の境界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 の実装により、指定した範囲の一部を処理する多くのタスクが作成されます。インテル® TBB のスケジューラーは、これらのタスクに異なる範囲のより小さな blocked_range を割り当てます。parallel_find_primes 関数は下記のようになります: **コード例 5**

コード例 5

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

5. for ループの本体をラムダ式で記述し、2 つめの引数として渡します。この引数は、各タスクへの処理を指定します。for ループがタスク別に実行されるようになったため、各タスクに割り当てる範囲 (<range>.begin() および <range>.end()) を変更する必要があります。

   オリジナルの for ループ本体をラムダ式で記述して、各タスクの処理を定義する必要もあります。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できる関数オブジェクトを作成できるようになります。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 を関数オブジェクト内部の*値で*「キャプチャー」する必要があります。[=] の後は、生成される関数オブジェクトの operator() のパラメーター・リストと宣言です。完全な parallel_find_primes 関数は コード例 6 のようになります。

コード例 6

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

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 オブジェクトを使用して時間を測定しています。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。インテル® TBB を使用するために他の main コードを変更する必要はありません。コード例 7

コード例 7

```
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;
```

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)] を選択してソリューションをビルドします。図 6
2. **[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)]** を選択して、Microsoft* Visual Studio* からアプリケーションを実行します。図 7

図 6

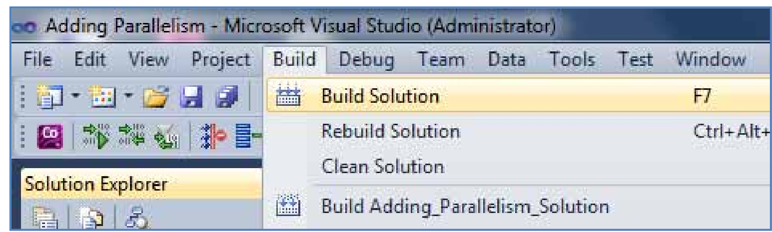
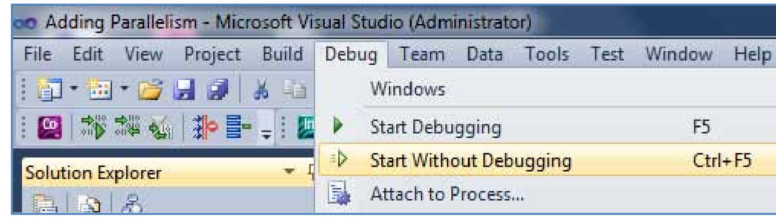


図 7



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はずです。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。(図 8 (シリアル) と 図 9 (並列) を参照)

図 8 – シリアル実行時間 = 14.07 秒

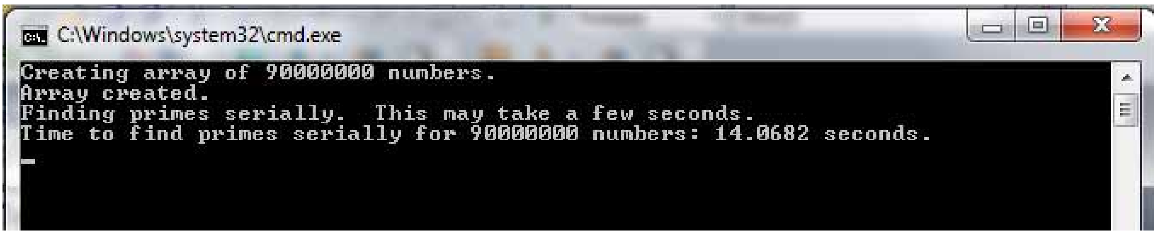


図 9 – 並列実行時間 = 8.83 秒

```
C:\Windows\system32\cmd.exe
Creating array of 90000000 numbers.
Array created.
Finding primes in parallel.  This may take a few seconds.
Time to find primes in parallel for 90000000 numbers: 8.8282 seconds.
```

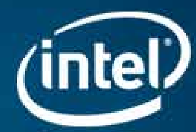
結果

この例では、for ループを parallel_for に変更して (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 を使用してチューニングを行うことで、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 プロセッサーのラップトップ (1.6 GHz、4 コア・プロセッサー、4GB RAM) で、Microsoft* Windows* 7、インテル® Parallel Studio XE Update 1、Microsoft* Visual Studio* 2010 を搭載するシステムで、90,000,000 の数列から素数を検索した場合の結果です。図 10

図 10

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



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、使用する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	少なくともループ反復を 3 回逆順にトレースします。動作している場合、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 を使用して正当性を検証する	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検証します。
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。

最適化が必要なコードに parallel_for が適していない場合

コードに計算負荷の高いループが含まれていない場合、インテル® TBB は左のプロセスのステップ 2、3、4 に別のオプションを用意しています。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。

並列処理に関する情報

重要な概念

小さく代表的なデータセットを選択する – 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、正当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

並列化による既存プログラムの最適化

最適化に関する注意事項

インテル® コンパイラー、関連ライブラリーおよび関連開発ツールには、インテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能な命令セット (SIMD 命令セットなど) 向けの最適化オプションが含まれているか、あるいはオプションを利用している可能性がありますが、両者では結果が異なります。また、インテル® コンパイラー用の特定のコンパイラー・オプション (インテル® マイクロアーキテクチャーに非固有のオプションを含む) 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。これらのコンパイラー・オプションと関連する命令セットおよび特定のマイクロプロセッサーの詳細は、『インテル® コンパイラー・ユーザー・リファレンス・ガイド』の「コンパイラー・オプション」を参照してください。インテル® コンパイラー製品のライブラリー・ルーチンの多くは、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより高度に最適化されます。インテル® コンパイラー製品のライブラリー・ルーチンの多くは、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより高度に最適化されます。インテル® コンパイラー製品のコンパイラーとライブラリーは、選択されたオプション、コード、およびその他の要因に基づいてインテル製マイクロプロセッサーおよび互換マイクロプロセッサー向けに最適化されますが、インテル製マイクロプロセッサーにおいてより優れたパフォーマンスが得られる傾向にあります。

インテル® コンパイラー、関連ライブラリーおよび関連開発ツールは、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。

インテルでは、インテル® コンパイラーおよびライブラリーがインテル製マイクロプロセッサーおよび互換マイクロプロセッサーにおいて、優れたパフォーマンスを引き出すのに役立つ選択肢であると信じておりますが、お客様の要件に最適なコンパイラーを選択いただくよう、他のコンパイラーの評価を行うことを推奨しています。インテルでは、あらゆるコンパイラーやライブラリーで優れたパフォーマンスが引き出され、お客様のビジネスの成功のお役に立ちたいと願っております。お気づきの点がございましたら、お知らせください。

改訂 #20110307

© 2011 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴは、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。