インテル® Parallel Studio 導入ガイド

並列化による既存プログラムの最適化

並列化による既存プログラムの最適化

はじめに

このガイドでは、インテル® Parallel Studio 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 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。

コード例 1. 直列バージョン (インテル® TBB の parallel_for 適用前)

```
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. 並列バージョン (インテル® TBB の parallel_for 適用後)

```
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® 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 1.99 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)。ここでは、直列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)。

インテル® Parallel Studio は、Microsoft* Visual Studio* C++ を利用する開発者向けに設計されたツール群で、スケーラブルで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より迅速に行えます。インテル® Parallel Studio には、以下のツールが含まれています。

- **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):** C++ テンプレート・ライブラリーであり、スレッドをタスクに抽象化し、信頼性と移植性に富んだスケーラブルな並列アプリケーションの開発を支援します。
- **インテル® Parallel Composer:** Windows* ベースのクライアント・アプリケーションの並列化を支援する、インテル® C++ コンパイラー、ライブラリー、デバッグ機能を含むツールです。
- **インテル® Parallel Amplifier:** アプリケーションのボトルネックを容易かつ迅速に特定する、パフォーマンス・アナライザーおよびチューニング・ソリューションを提供します。プロセッサーのアーキテクチャーやアセンブリー・コードの知識は必要ありません。
- **インテル® Parallel Inspector:** 直列コードおよび並列コードの動的なメモリー / スレッド分析ツールです。ソフトウェアのエラーや欠陥を容易かつ迅速に発見します。

並列化による既存プログラムの最適化

並列処理の実装

インテル® TBB を使用すると、直列アプリケーションの並列化プロセスを簡略化できます。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、標準的な直列 "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。

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* 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。
- 製品 : インテル® Parallel Studio 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、次のサイトを参照してください : **インテル® TBB** (英語)

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 のインストールと設定

1. インテル® Parallel Studio の評価版をダウンロードします。
2. parallel_studio_setup.exe をクリックしてインテル® Parallel Studio をインストールします。

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、My Documents\Visual Studio 200x\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、**[ファイル] > [開く] > [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 PiSolver.sln ファイルを選択します。図 1 を参照してください。

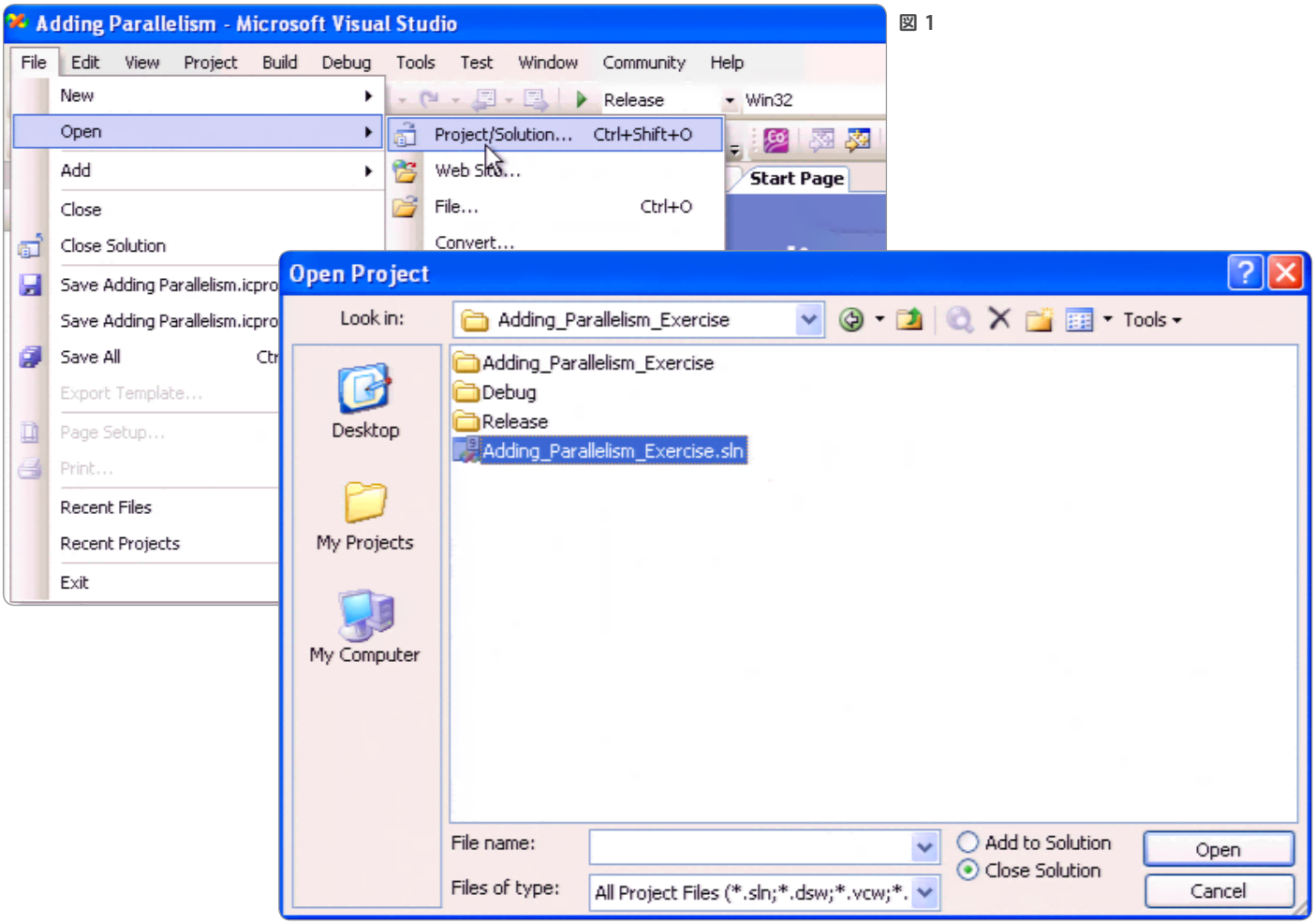


図 1

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、直列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した直列サンプルコードが含まれています。どちらのプロジェクトも、インテル® Parallel Composer を使用するように構成されています。図 2

3. 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、**[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 **[プロパティ]** を選択します。適切な変更を下記に示します。詳細は、Adding_Parallelism.cpp の先頭のコメントを参照してください。図 3

4. Adding_Parallelism.cpp のコードを確認するか、下記の概要を参照してください。

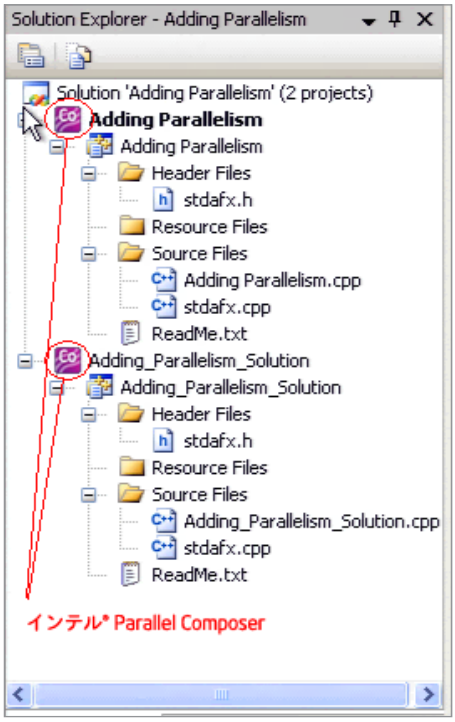


図 2

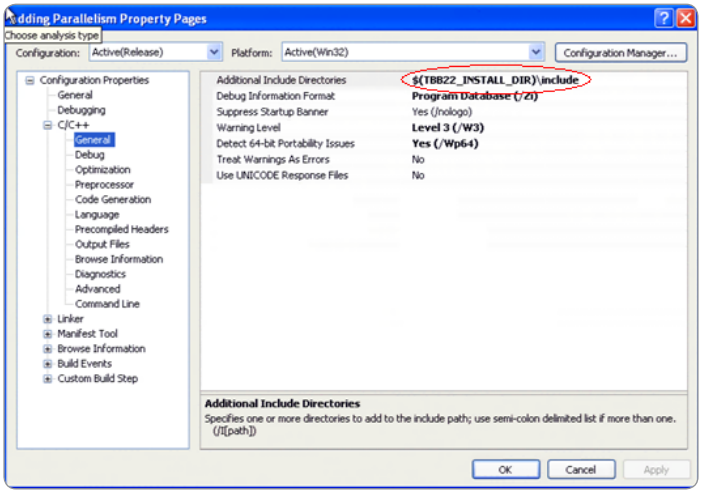
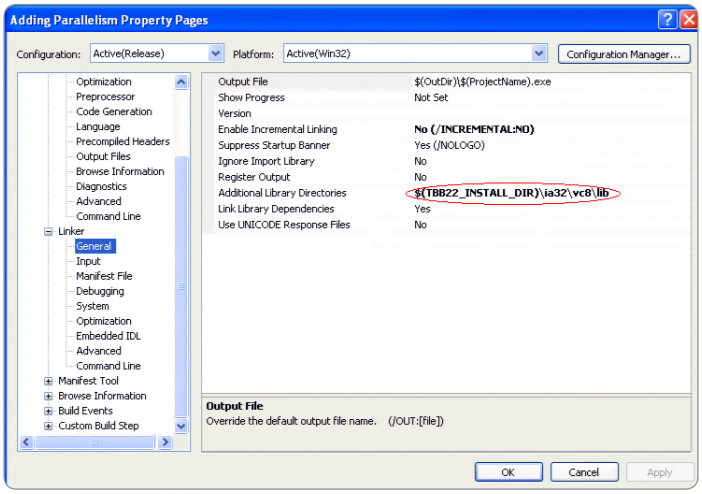


図 3



Adding_Parallelism サンプルコードは何を行っているか

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直列で、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います。

並列化による既存プログラムの最適化

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 (直列) の find_primes 関数を示します : **コード例 3**

3. parallel_find_primes 内部で、parallel_for を呼び出します。parallel_change_array 関数の呼び出しが参考になります。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、直列 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。直列 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、直列 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直列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**

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」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() の引数リストと宣言です。完全な parallel_find_primes 関数は下記のようになります : **コード例 6**

コード例 5

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

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. **[ビルド] > [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 4**
2. **[デバッグ] > [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 5**

図 4

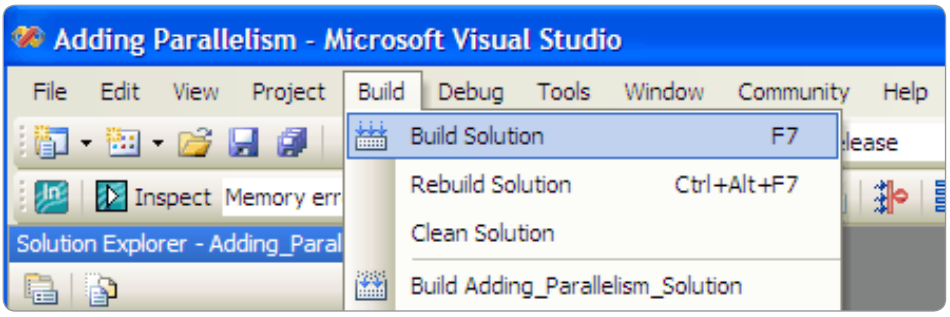
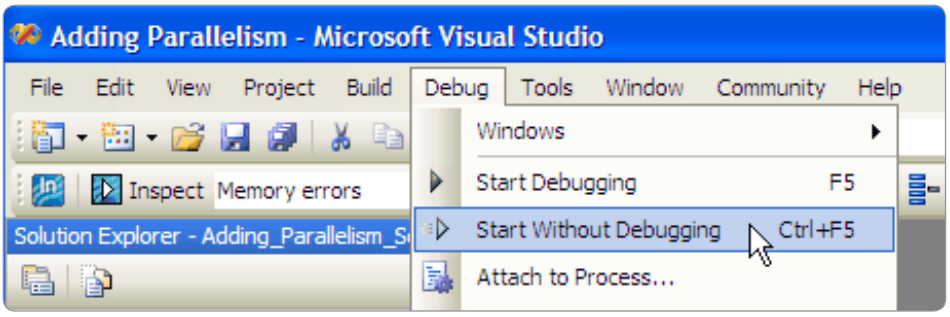


図 5



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、直列バージョンと並列バージョンを別々に実行します。図 6

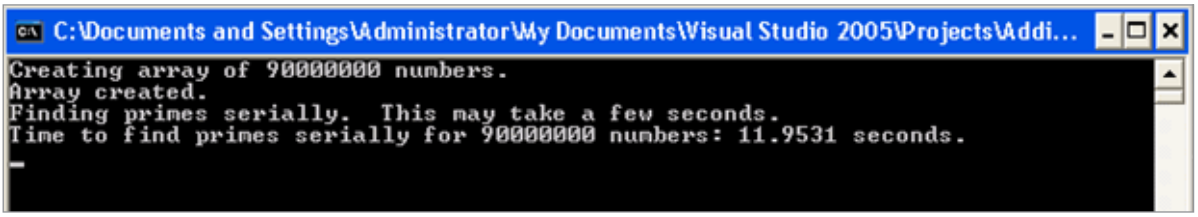
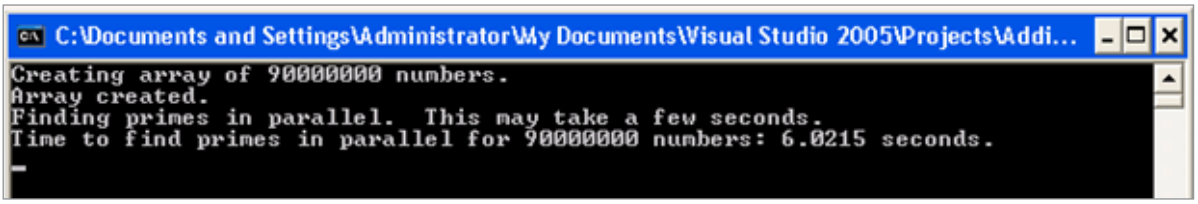


図 12



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® Parallel Amplifier 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® Xeon® サーバーを使用して、90,000,000 の数列から素数を検索した場合の結果です。図 13

コア数	実行時間 (素数検索の最初のバージョン)	直列に対する速度向上
1	11.95 秒	——
2	6.02 秒	1.99 倍
4	3.02 秒	3.96 倍
8	1.52 秒	7.86 倍

図 13

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 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® Parallel Amplifier で hotspot 分析を実行して、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® Parallel Inspector を使用して正当性を検証する	インテル® Parallel Inspector でスレッドエラー分析を実行して、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	直列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。

並列化による既存プログラムの最適化

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 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。

関連リンク (英語)

- インテル® ソフトウェア・ネットワーク・フォーラム
- インテル® ソフトウェア開発製品ナレッジベース
- インテル® ソフトウェア・ネットワーク・ブログ
- インテル® Parallel Studio Web サイト
- インテル® TBB Web サイト
- Go Parallel — 並列化に関するブログ、記事、ビデオ
- 開発者向け無料 Web セミナー (オンデマンド)

その他の導入ガイド

- hotspot の特定と最適化
- メモリーエラーの排除とプログラムの安定性の向上

インテル® Parallel Studio に対するユーザーの声

「インテル® TBB の malloc は、弊社のマルチスレッド・アプリケーションで目標とする並列処理の速度を達成するための重要なツールでした。C 標準ライブラリーのメモリー・アロケーターと単純に置換できる点も重要でした。」

DreamWorks Animation
R&D FX マネージャー、Ron Henderson 氏

「インテル® TBB に興味を持った理由は 2 つあります。既存の直列コードに統合できる相対的な容易さ、そして、プログラマーの習得に必要な時間が短いことでした。インテル® TBB を実装してから数日の間に、パフォーマンスが目に見えて向上するようになりました。」

Creative Assembly
シニア・エンジン・コーダー、Yuri O’Donnell 氏

「インテル® Parallel Studio を使用して SpeedTree を再コンパイルしたところ、パフォーマンスが劇的に向上しました。最初は測定結果が間違いだと思ったほどです。それ以来、インテル® Parallel Composer とインテル® Parallel Amplifier を使用することで、SpeedTree ランタイムの CPU クリティカルなセクションで 35% の速度向上が達成できています。いまではインテル® コンパイラーの虜になっています。弊社の開発環境で継続して利用することになったのは言うまでもありません。」

IDV
CEO、Chris King 氏

JPN/1006/PDF/XL/SSG/TT 323858-001JA