インテル® Parallel Building Blocks: 入門チュートリアルと実践演習

概要

用語

インテル® Parallel Building Blocks - インテル® PBB

インテル® Array Building Blocks - インテル® ArBB

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

インテル® Cilk™ Plus - タスクとデータの並列拡張、Cilk、配列表記がそれぞれ含まれています。

本評価ガイドの要件:

- Microsoft* Visual Studio* 2008 または 2010
- インテル® Parallel Studio 2011 の評価版のダウンロード: http://software.intel.com/en-us/articles/intel-software-evaluation-center/ (英語)
- インテル® Array Building Blocks ベータ版のダウンロード: http://software.intel.com/en-us/articles/intel-array-building-blocks/ (英語)
- 実践演習資料のダウンロード: http://software.intel.com/en-us/articles/intel-parallel-building-blocks-getting-started-tutorial-and-hands-on-lab/

マイクロプロセッサーのパフォーマンス・ゲインの主な要因が GHz からマルチコアなどの機能に移行するのに伴い、アプリケーションを最適化して新しいプラットフォーム機能を活用することの重要性が増しています。以前は、より高いクロックスピードを備えた新しい CPU では同一アプリケーションのパフォーマンスは自動で向上していました。

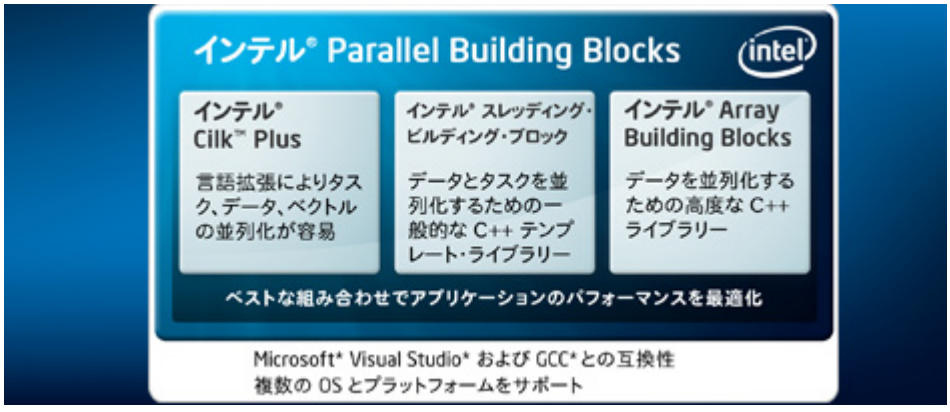
しかしながら、今日では最新の CPU を購入しても、シリアル (またはシーケンシャル) に記述されたアプリケーションではパフォーマンスの向上は見られない可能性があります。ユーザーが期待するパフォーマンスの向上を提供するには、開発者は CPU で利用できる新しい機能を理解し、それらの機能を有効活用する必要があります。

インテル® ソフトウェア・グループでは、アプリケーションの並列化に取り組む開発者を支援する幅広いツールを提供しています。本ガイドでは、汎用目的から特殊な並列化まで広範囲にわたるソリューションを提供する補完的なモデルセット、インテル® Parallel Building Blocks (インテル® PBB) を紹介します。インテル® PBB のツールを使用することで、開発者は特定の環境やニーズに合わせて、アプリケーション内でソリューションをうまく組み合わせることができます。インテル® PBB は、マルチコアの時代を享受する、簡単でありながらスケーラブルな方法を提供します。

この実践演習には、インテル® PBB を使い始めるにあたって役立つ豊富な情報が含まれています。

- 各モデルの詳細な説明
  - 特定のモデルを使用する場合の推奨事項
- それぞれのモデルの説明ビデオ
- 初級者/上級者用の 64 ビット・システム向けに設定された Microsoft* Visual Studio* のインテル® PBB 演習コーディング・プロジェクト

図 1



インテル® Parallel Building Blocks: 概要

高レベルの並列化を引き出しつつ、高レベルの抽象化を利用してコードを作成するには、どうしたら良いでしょうか。当然ながら、インテル製品をお使いのお客様はすでに開発中のプログラムをお持ちでしょう。インテル® PBB モデルでは、マルチコア・ハードウェア向けのプログラミング方法を幅広くサポートしています。インテル® PBB の並列抽象化により、多くの計算を行うコード部分を強化したり、あるいは新しい並列アプリケーションの一からの作成もより迅速に行うことができます。

インテル® PBB は、製品の名前ではありません。並列化の実装に取り組む開発者を支援するモデル (またはツール) の集合を表す用語です。インテル® PBB は、インテル® Cilk™ Plus、インテル® TBB、インテル® ArBB で構成されています。

	インテル® Parallel Building Blocks		
	インテル® Cilk™ Plus	**インテル® TBB**	**インテル® ArBB**
概要	タスクおよびベクトル並列化を簡単にする言語拡張	タスクを並列化するための一般的な C++ テンプレート・ライブラリー	ベクトル並列化のための高度な C++ テンプレート・ライブラリー
機能	• 並列化のための 3 つのシンプルなキーワードと配列表記タスクおよびベクトル並列化のサポート • シリアルコードと類似したセマンティクス	• 並列アルゴリズムとデータ構造 • スケーラブルなメモリーの割り当てとタスク・スケジューリング • 同期プリミティブ	• 将来のインテル® プラットフォームへ自動的にスケール • ランタイム・コンパイラーにより決定されるコア、スレッド、SIMD の使用
使用する理由	• コードを並列化する簡単な方法を提供 • 順序一貫性 + 低オーバーヘッド= 強力なソリューション • C および C++。Windows*、Linux*	• 汎用並列処理向けに豊富な機能セット • C++。Windows*、Linux*、Mac OS*、その他の OS	• 柔軟性のあるベクトル並列化に使用 • JIT + VM テクノロジー=柔軟性に富んだ強力なソリューション • C++。Windows*、Linux*

インテルでは、幅広いソリューションを提供するためのツールを販売しているため、プログラマーはすべての問題を決められた型にはめ込む必要はありません。本ガイドではインテル® PBB モデルについて詳細に説明しますが、ここで紹介するものだけがすべてではありません。並列化におけるほぼすべての問題は、これらのビルディング・ブロックの 1 つ、あるいは複数のビルディング・ブロックを組み合わせて表現できます。したがって、インテル® PBB の最も優れた長所は、モデルの多様性とその一貫した表現力の両方にあると言えます。

インテル® PBB モデルは使いやすいテンプレート形式のライブラリーまたは言語拡張のため、プログラマーは並列化を通じて、高い生産性を保ちつつ、優れたパフォーマンスを得られます。インテル® PBB は、次の 4 つの点で優れています。

ユーザビリティー

インテル® PBB モデルは高レベルの抽象化を利用して汎用プログラミングを表現します。

信頼性

インテル® PBB モデルは、既存のインテル® ツールと互換性があり、一般的なコーディング・エラーやスレッディング・エラーを排除します。

スケーラビリティー

インテル® PBB モデルは、より多くのプロセッサー・コアに対するフォワード・スケーリングをサポートしています。そのため、インテル® PBB を利用して並列化を導入することで、CPU コア数が増加しても引き続き利点が得られます。

オープン性

インテル® PBB モデルは、柔軟なライセンス体系により、複数のオペレーティング・システム、ハードウェア、統合開発環境 (IDE)、コンパイラー・プラットフォームに移植が可能です。

できる限りの最適化を引き出したいと願う開発者もいることでしょう。インテル® PBB は、それを実現します。このモデルはあらゆるタイプのプログラマーに並列プログラミングを開放する一方で、必要に応じて経験豊富なプログラマーが根底にある詳細まで掘り下げることも可能です。

並列プログラミングの注意事項

今日の並列プログラミングでは、スレッディングによるコア間の並列化に加えて、SIMD 命令セットを通じて、個々のコア内部の並列化も行う必要があります。スレッドによる並列化は新しいものではありませんが、Posix* や Windows* スレッドは、言語自体ではなく、オペレーティング・システムの一部として設計されています。インテルでは、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) による抽象化を通じてこのモデルを進化させました。インテル® TBB のプログラマーは個々のスレッドではなく、タスクで作業します。インテル® ソフトウェア開発製品は、インテル® Parallel Composer 2011 を中心に、さまざまなレベルのベクトル化およびスレッド化をサポートし、経験の浅いプログラマーから熟練したプログラマーまで幅広く支援します。インテル® Cilk™ はユーザビリティー、安全性の観点からインテル® TBB を補強し、インテル® ArBB はスレッディング・ランタイムとしてインテル® TBB を使用し、対象のアーキテクチャーに適合した SIMD 命令セットを生成します。

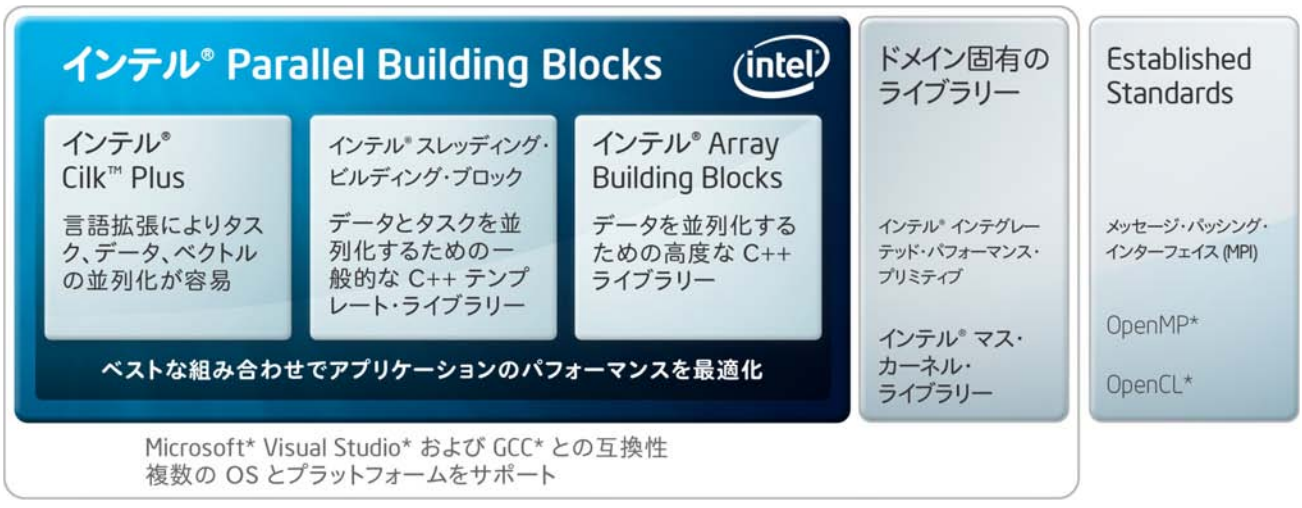
次に、抱えている問題が、データの並列化または汎用的な並列化によって解決できるかどうかを判断しなければなりません。汎用並列化では、並列アルゴリズムと組み合わせて 1 つの問題を解くタスクを管理します。データ並列化は、並列化の特殊なケースです。アプリケーションの同一コードが実行順を指定せずに複数のデータ項目にアクセスします。ベクトル並列化、ループ、タスクの組み合わせを使用して実装され、データ項目に同時アクセスできる場合や計算処理に比べてアクセス時間が無視できるほどわずかな場合に適しています。

ただし、データ並列化をベクトル化と混同しないように注意してください。ベクトル化は、汎用並列アプローチとデータ並列アプローチの両方で使用できます。

インテル® PBB は、広範囲で補完的なプログラミング構造セットを提供し、現在だけでなく将来のプログラミング・ニーズにも対応する実績のある相互運用可能な統合ソリューションをもたらします。

- 既存のアプリケーションで使用しているツールともシームレスに動作するデータ並列ソリューションと汎用並列ソリューションの両方を利用できます。
- 言語拡張とライブラリー・ソリューションの両方を利用し、業界で導入、実証されているソリューションを使用してアプリケーションの信頼性を強化します。
- インテル® PBB は、最適化された高レベルのアルゴリズムおよびカスタム・ワークロードを構築するための低レベルの構造の両方から利点を得ます。必要な制御レベルを維持しながら、アプリケーションの自動スケーリングを通じて、将来にわたってパフォーマンスを最大限に引き出します。
- 環境/アプリケーションに合うようにアプリケーション内で新しい並列モデルをうまく組み合わせることができ、プラットフォームが進化してもコーディングをあまり行わなくて済むようにします。

図 2



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)

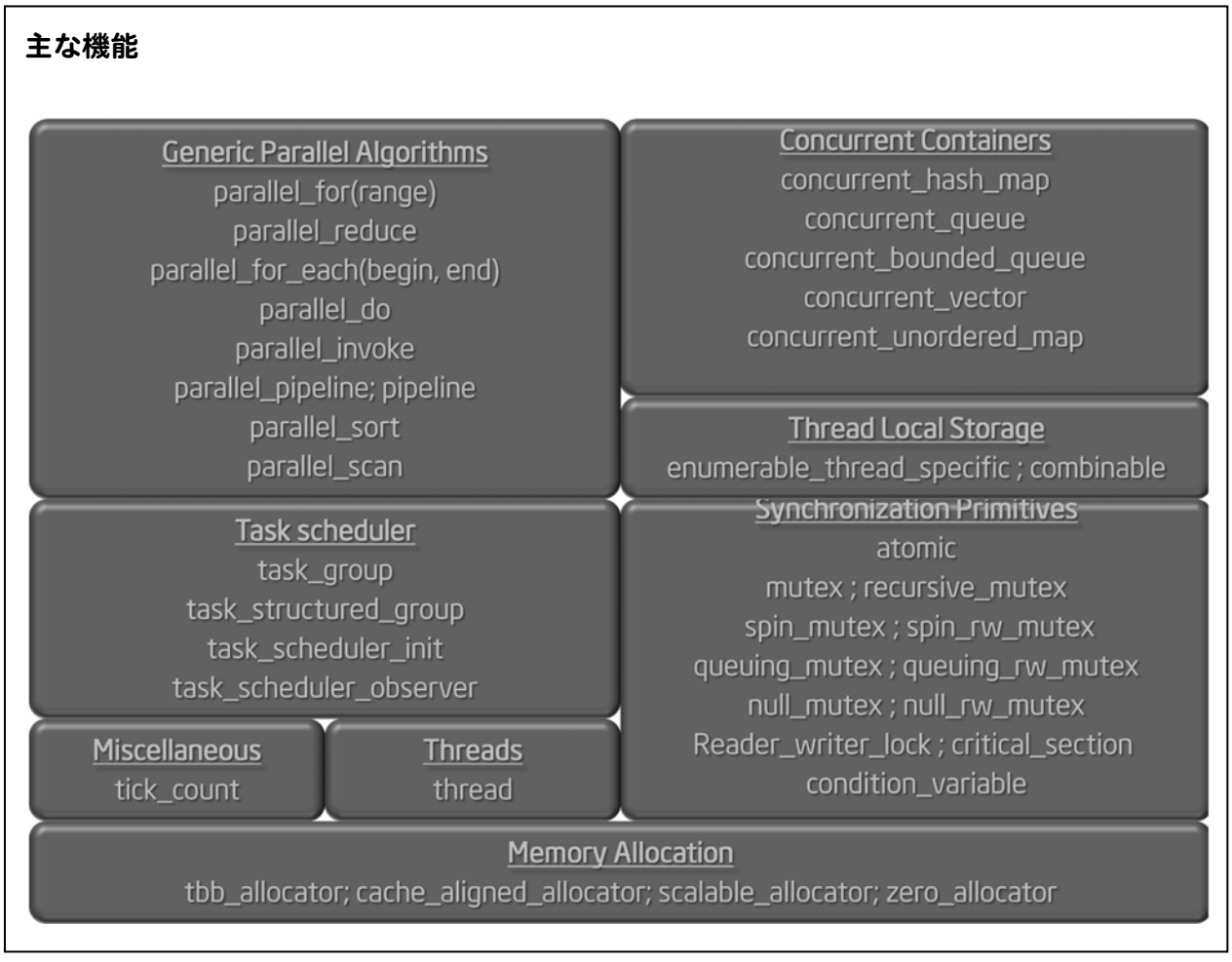
インテル® スレッディング・ビルディング・ブロックとその利点

インテル® TBB は、一般的な共有メモリーの並列化を行うためのコンポーネントを提供する C++ ライブラリーです。既存の環境へ簡単に、段階的に統合することができます。入れ子の並列化を念頭に設計されており、高レベルの汎用コンポーネントも含まれています。インテル® TBB は、共通のイディオムのテンプレート・コードを提供する一方、特定の使用ケースへのカスタマイズも可能で、より高度な制御力を必要とするエキスパートには低レベルのコンポーネントも提供します。また、カスタム・アルゴリズムや任意のタスクグラフを構築できるほどの十分な汎用性も備えています。インテル® Parallel Inspector とインテル® Parallel Amplifier は、インテル® TBB と併用できます。

インテル® TBB の使用

インテル® TBB は、データ並列化の特殊な構造が適していない場合など、汎用並列化を行う場合に使用します。また、巧妙なアルゴリズムやコンテナー (parallel sort、concurrent vectors など) が適している場合に使用します。

主な機能



演習 1: インテル® スレッディング・ビルディング・ブロックを使用したソートと集計

難易度: 初級
演習プロジェクトの設定: Microsoft* Visual Studio* 2008 (64 ビット・プロジェクト)
必要な資料:

- 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:

  http://www.threadingbuildingblocks.org/uploads/81/91/Latest Open Source Documentation/Reference.pdf (英語)

- 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:

  http://www.threadingbuildingblocks.org/uploads/81/91/Latest Open Source Documentation/Tutorial.pdf (英語)

目的: この演習では、std::vector (People 型を参照) として格納された個人情報のデータベースを操作する単純なプログラムを使用します。このプログラムは、年齢別に情報をソートしてから、それぞれの名前の発生頻度を数えます。この演習問題では、インテル® TBB を使用してソート機能とメモリー管理を最適化します。

インテル® PBB を使用する理由: 本プログラムでは、中央集約型のデータ構造を使用し、大量のメモリーの割り当てと解除を行い、並列可能な方法でベクトルを操作します。インテル® TBB の多くのパフォーマンス機能により、このコードの向上に適した選択肢が提供されます。並列化のほか、プログラムではインテル® TBB を活用して、コンカレント・コンテナーを利用したデータの格納 (演習 2)、スケーラブル・メモリー・アロケーターを使用したメモリー処理の大幅な向上を実現できます。

手順:

インテル® TBB について理解するため、この演習に取り組む前にインテル® スレッディング・ビルディング・ブロックのビデオを参照してください:
http://software.intel.com/en- us/videos/introduction-to-intel-threading-building-blocks/
(英語)

第 1 部: ベースラインの作成

1. TBBLab.zip ファイルを PBB Labs/Threading Building Blocks からローカル・ディレクトリーにコピーします。
2. TBBLab.zip をローカル・ディレクトリーに展開します。
3. Microsoft* Visual Studio* 2008 で TBBLab.sln ソリューションを開きます。ソリューションは、事前に設定されているため、何も変更する必要はありません (図 4)。
4. win32 Release 版のソリューションを使用していることを確認します (図 5)。
5. [Build (ビルド)] > [Rebuild (リビルド)] を使用して、ソリューションをビルドし、正常にビルドされていることを確認します (図 6)。

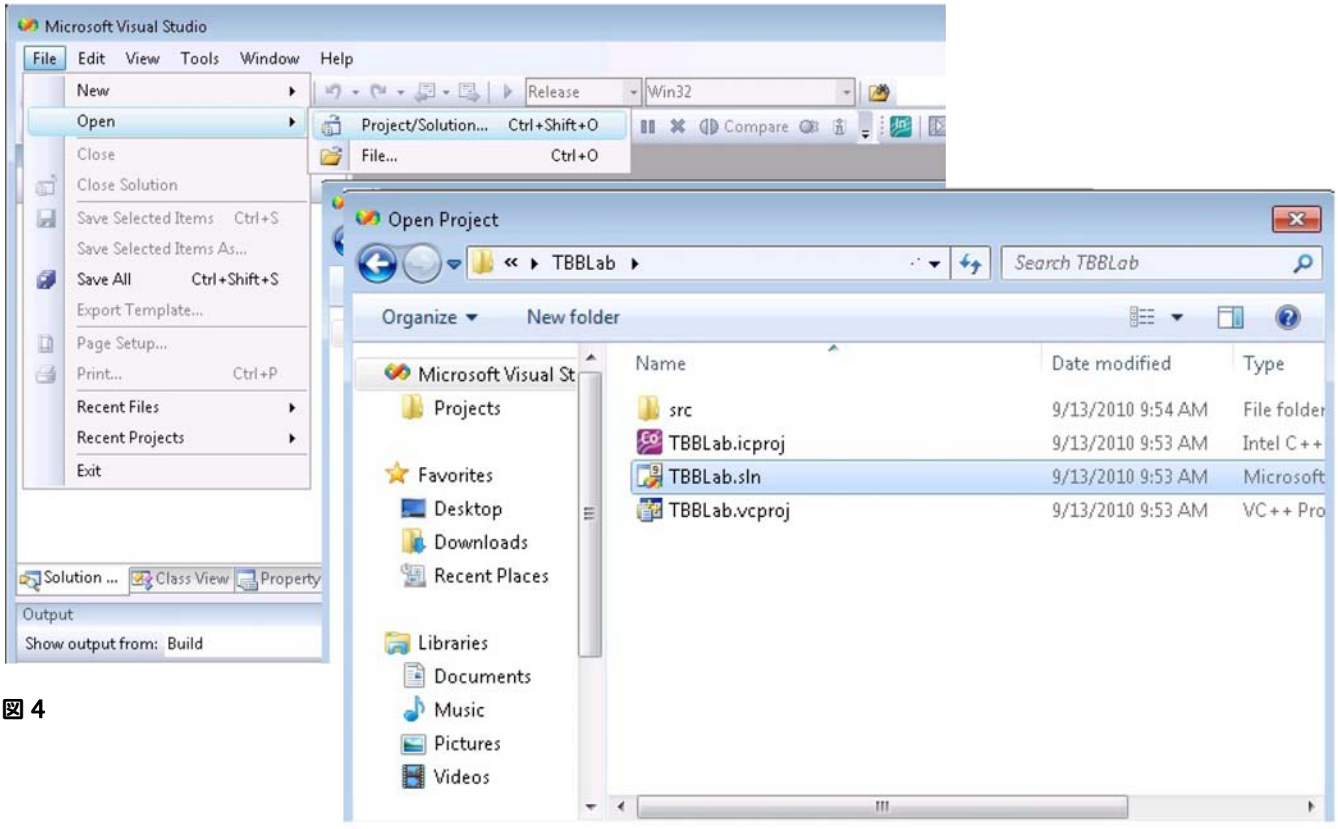


図 4

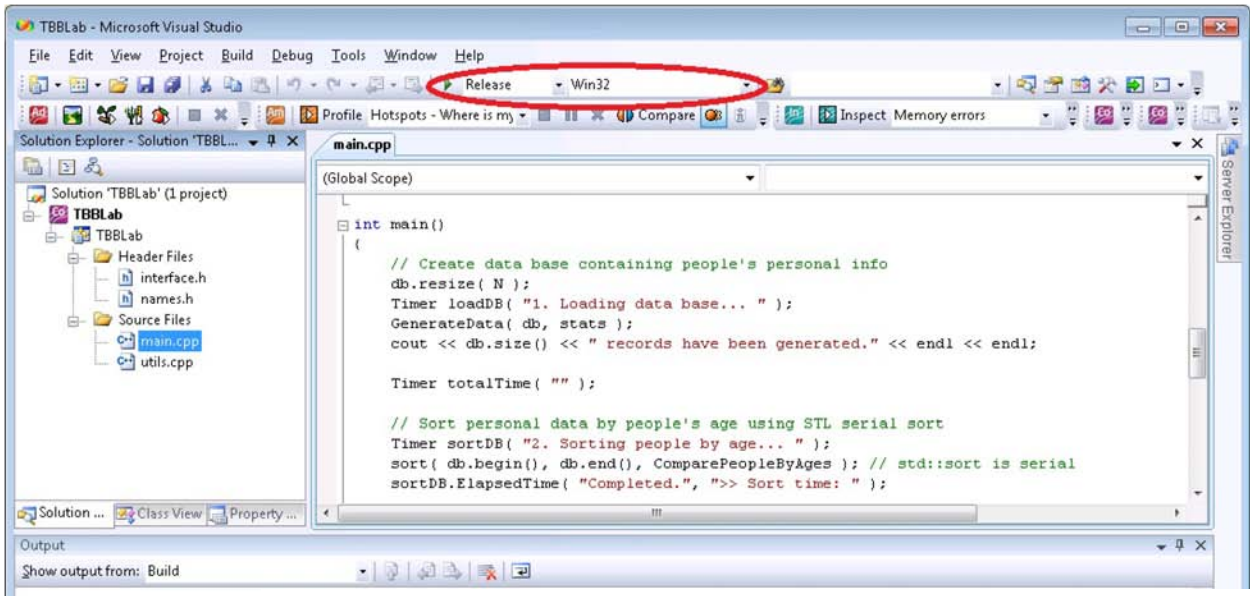


図 5

図 6

6. 演習 1 のコードは、std::vector として格納されたレコードのデータベースを使用しています。それぞれのレコードには、名字、名前、年齢が含まれています。ベクトルは年齢別にソートされてから、それぞれの名前の発生数が「stats」と呼ばれる std::map コンテナーに格納されます。本演習の焦点ではありませんが、このプログラムではインテル® TBB の parallel_for テンプレートを使用して、並列に名前をカウントします。main. cpp の CountNames 関数のコードを (任意で) 調査します。
7. [Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] を選択して、プログラムを実行します。
8. 合計実行時間とソート実行時間を記録します。パフォーマンス測定のため、Release ビルドを実行していることを確認してください。

   ソート実行時間:

   _____

   合計実行時間:

   _____

9. ソート実行時間と合計時間の両方が向上するよう、プログラムを最適化します。

第 2 部: ソートの最適化

1. 名前のソートは、main() の 44 行目で標準テンプレート・ライブラリー (STL) の sort 関数を呼び出して行っています。ソート時間 (main() から呼び出される sort 関数) を短縮するには、リファレンス・ガイドのセクション 4.11 にあるように、インテル® TBB の parallel_sort テンプレートを使用します。インテル® TBB parallel_sort は、STL sort と同じパラメーターを使用します。tbb/parallel_sort.h を忘れずにインクルードしてください。
2. [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、[Start without Debugging (デバッグなしで開始)] でプログラムを実行します。新しい合計実行時間とソート実行時間を記録します。

   ソート実行時間:

   _____

   合計実行時間:

   _____

3. シリアル sort の代わりに並列 sort を使用した結果、速度はどれぐらい向上しましたか?

第 3 部: メモリー管理の最適化

1. チュートリアルのセクション 10 に記載されているインテル® TBB のスケーラブル・メモリー・アロケーターを使用して、プログラムの実行時間をさらに向上させることができます。interface.h の 21 行目で MyString 型の最後のパラメーターとして tbb_allocator<char> を使用します。(最後のパラメーターは空白になっています。これは、STL 標準アロケーターが使用されていることを意味します。) これにより、インテル® TBB のスケラーブル・アロケーターを使用して、STL 文字列にメモリーの割り当てと解除を行うよう命令します。tbb/tbb_allocator.h を忘れずにインクルードしてください。
2. プログラムを再実行して、新しい合計実行時間とソート実行時間を記録します。

   ソート実行時間:
   _____

   合計実行時間:
   _____

3. インテル® TBB のスケラーブル・アロケーターを使用した結果、速度はどれぐらい向上しましたか?

結論: インテル® TBB を使用することで、スレッド化の詳細を管理することなく、大幅なスケーラビリティーが得られました。インテル® TBB は、自動で実行対象のシステム上にあるすべてのプロセッシング・コアを利用しようとします (デフォルト)。

演習 2: インテル® スレッディング・ビルディング・ブロックを使用したソートと集計

難易度: 中級
演習プロジェクトの設定: Microsoft* Visual Studio* 2008 (64 ビット・プロジェクト)
必要な資料:

- 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: http://www.threadingbuildingblocks.org/uploads/81/91/Latest Open Source Documentation/Reference.pdf (英語)
- 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: http://www.threadingbuildingblocks.org/uploads/81/91/Latest Open Source Documentation/Tutorial.pdf (英語)

目的: この演習では、インテル® TBB の演習 1 と同じプログラムを使用します (std::vector として格納された個人情報のデータベースの集計とソートを行う簡単なアプリケーション)。この演習では、インテル® TBB を使用して、名前の集計をさらに最適化します。

インテル® PBB を使用する理由: 本プログラムでは、中央集約型のデータ構造を使用し、大量のメモリーの割り当てと解除を行い、並列可能な方法でベクトルを操作します。インテル® TBB の多くのパフォーマンス機能により、このコードの向上に適した選択肢が提供されます。並列化のほか、プログラムではインテル® TBB を活用して、コンカレント・コンテナーを利用したデータの格納、アトミック操作による低いオーバーヘッドでのスレッドの安全性を確保、スケーラブル・メモリー・アロケーターを使用したメモリー処理の大幅な向上が実現できます。

手順:

第 1 部: ベースラインの作成

1. この演習では、演習 1 で最適化したインテル® TBB の演習プロジェクトを引き続き使用します。コードには、名前の計算にインテル® TBB の parallel_for テンプレートが使用されています。名前のデータベースを処理する際にスレッドは並列で動作します。STL マップコンテナーにある名前を 1 つずつ検索します。

   マップには、名前とその発生頻度が格納されています。データベースで名前が見つかると、マップの発生頻度が更新されます。main.cpp の CountNames 関数のコードを参照してください。
2. [Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] を選択して、プログラムを実行します。
3. 合計実行時間とソート実行時間を記録します。

   ソート実行時間:
   _____

   合計実行時間:
   _____

4. プログラムを最適化して、(合計実行時間の一部である) 集計時間を向上させます

第 2 部: 集計を最適化します。

1. main. cpp の CountNames 関数のコードを再度確認してみましょう。2 つのスレッドが同時にマップにアクセスしないように Windows* のクリティカル・セクション・オブジェクトが使用されています。(クリティカル・セクションの詳細については、Microsoft* Visual Studio* 2010 ヘルプを参照してください。) インテル® TBB の parallel_for のようなテンプレートを使用する際は、開発者が責任を持って複数のスレッドがアクセスする可能性のあるデータの保護を行う必要があります。ただし、並列プログラムでグローバルロックを使用すると、パフォーマンスやスケーラビリティーが制限されることに注意してください。グローバルロックによりスレッドセーフなアクセスを保証するデータコンテナーを使用する代わりに、インテル® TBB のコンカレント・コンテナーのような、並行処理に対応するコンテナーを使用すると、同期を最小限に抑えたり、回避できます。
2. interface.h の 23 行目で宣言されている Stats 型の STL マップの代わりにインテル® TBB の concurrent_unordered_map コンテナーを使用します。concurrent_unordered_map コンテナーは同時挿入と走査をサポートしています。つまり、データを破損することなく、複数のスレッドが新しいエントリーの追加

と値の検索を行うことができます。詳細は、インテル® TBB リファレンス・ガイドのセクション 5.2 を参照してください。tbb/concurrent_ unordered_map.h を忘れずにインクルードしてください。

3. コンカレント・コンテナーを使用することで、マップへのアクセスでグローバルロックによる保護が必要なくなります。しかし、マップに格納されているカウンター値の更新には、まだ保護が必要です。保護を施さないと、別のスレッドが書き込むのと同時にカウンター値を読み込んでしまう可能性があるからです。更新処理は単純な整数の加算なので、安全性を確保するのに最も簡単で最も軽量な方法はアトミック操作を使用することです。マップの 2 番目のパラメーター (現時点では int) を atomic<int> に替えて、インテル® TBB の concurrent_unordered_map の値の型とし、インテル® TBB のアトミック操作を使用します(interface.h の 23 行目の Stats 型を参照)。 詳細は、インテル® TBB チュートリアルのセクション 8 を参照してください。tbb/atomic.h を忘れずにインクルードしてください。
4. コンカレント・コンテナーとアトミック更新を使用することで、クリティカル・セクションは必要なくなりました。main() (main.cpp) の CountNames 関数にあるクリティカル・セクションの呼び出しを忘れずに削除してください。
5. [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。そして、[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] を選択して、プログラムを実行します。
6. プログラムが正しく動作することを確認してください。ソートと集計が正しく行われると、最後に "Correct!" と出力されます。
7. 合計実行時間とソート実行時間を記録します。

   ソート実行時間:
   _____

   合計実行時間:
   _____

8. クリティカル・セクションのオーバーヘッドを取り除いた結果、速度はどれぐらい向上しましたか? 第 1 部のステップ 3 と比べてみてください。

結論: インテル® TBB を使用することで、スレッド化の詳細を管理することなく、大幅なスケーラビリティーが得られました。インテル® TBB のビルディング・ブロックは、開発者の並列化ニーズに対応するように事前にコーディングされたコンポーネントを提供します。

インテル® Cilk™ Plus

インテル® Cilk™ Plus とその利点

インテル® Cilk™ Plus には、キーワードとレデューサー、配列表記、要素関数、ユーザー指示によるベクトル化の 5 つのコンポーネントがあります。これらは、インテル® Parallel Composer 2011 に統合されています。インテル® Cilk™ Plus は、シリアルプログラムを並列化する最も簡単な方法です。単純な構文のため、分かりやすく、使いやすい機能です。

- 厳密な fork-join モデルにより、明瞭なセマンティクスを利用できます。
- インテル® Cilk™ Plus は、プログラムの並列制御フローを理解するのに最も簡単な方法を提供します。
- ワークスチールによる自動ロード・バランシングに加えて、親のスチールによりシリアル実行時と同等の結果が得られます。
- オーバーヘッドの低いタスクスポーンにより、小さなタスクを多数作成できます。多数の小さなタスクから成るプログラムでは、コア数が増えるとともに、タスク・スケジューラーがロード・バランシングとフォワード・スケーリングの両方を活用できる機会が増えます。

インテル® Cilk™ Plus のキーワードとレデューサーの使用

次のような場合に cilk_spawn/cilk_sync/cilk_for/reducers を使用します。

- インテル® Composer 2011 を使用して、シリアルプログラムを簡単かつ迅速に、依存性なく並列化したい場合。単純な構文なので、分かりやすく、簡単に使えます。
- 親のスチールによりシリアル実行時と同等の結果 (毎回、正しい解を得るなど) を迅速に保証したい場合。
- データ並列構造があまり役に立たず、また、作成しているコードがインテル® TBB で提供される既存のアルゴリズム実装にマップしない場合。

主な機能

機能	例	セマンティクス
関数呼び出しのスポーン	x = cilk_spawn func(g(y),h(z));	func は非同期的に実行。
同期文	cilk_sync;	現在の関数でスポーンされたすべての子が終了するのを待機。

Parallel_for ループ	cilk_for (int i = 0; i < N; i++) {
	statement;
	}
		ループの繰り返しを並列で実行。

配列表記

配列表記

配列表記は、既存のデータ型の配列で既存の言語の操作が可能な言語拡張です。

- 配列セクションを指定しやすい構文です。
- 配列要素に関連した操作を定義します。
- 要素関数を介して、同時にスカラー関数を複数の配列要素にマップします。
- 配列表記は、強固なリダクション演算も提供します。

主な機能

配列セクション

既存のデータ型の配列で既存の言語の操作ができる言語拡張:

<配列ベース> [<下限> :<長さ> [:<ストライド>]]
[<下限> :<長さ> [:<ストライド>]].....

セクション指定子は、Fortran 形式 [上限:下限] とは異なり、[下限:長さ] のペア (memcpy 形式) です。

```
A[:]// ベクトル A のすべて
B[2:6] // ベクトル B の要素 2 から 7
C[:][5] // 行列 C の列 5
D[0:3:2] // ベクトル D の要素 0、2、4
```

演算子マップ

ほとんどの C/C++ 算術演算子と論理演算子を配列セクションで利用できます。

+、-、*、/、%、<、==、>、<=、!=、>=、++、--、|、&、^、&&、||、!、-(単項)、+(単項)、+=、-=、*=、/=、*(ポインター逆参照)

演算子は暗黙的に部分配列オペランドの全要素にマップされます。

```
a[:]* b[:]// 要素ごとの乗算
a[3:2][3:2] + b[5:2][5:2] // 2x2 行列の加算
```

異なる要素の演算も実行順序の制限なしに、並列に実行可能です。

配列オペランドのランクとサイズは同じでなければなりません。

```
a[0:4][1:2] + b[1:2][0:4] // ランクサイズの不一致エラー
```

スカラーオペランドは、セクション全体をフィルするよう自動的に展開されます。

```
a[0:4][1:2] + b[0][1] // OK、スカラー b[0][1] を加算
```

代入マップ

代入演算子は、左辺 (LHS) の部分配列の各要素に並列に適用されます。代入の LHS は右辺 (RHS) が評価される配列コンテキストを定義します。RHS 部分配列のランクは LHS と同じでなければなりません。各ランクの長さは、対応する LHS ランクと一致しなければなりません。スカラーは自動的に展開されます。

```
a[:][:]= b[:][2][:]+ c;
e[:]= d;
e[:]= b[:][1][:];// ランクの不一致エラー
a[:][:]= e[:];// ランクの不一致エラー
```

RHS は LHS の要素がストアされる前に評価されます。コンパイラーは、必要に応じて一時配列を挿入します。RHS のオペランドが LHS の L 値とエイリアスする場合でも、コンパイラーは RHS の演算をベクトル化することができます。

```
a[1:s] = a[0:s] + 1; // a[1:s-1] の古い値を使用
```

リダクション

リダクションは部分配列の要素を組み合わせ、スカラー結果を生成します。

```
Int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // 合計 10
```

基本的な C のデータ型をサポートしている 9 つのビルトインのリダクション関数があります。

```
__sec_reduce_add__sec_reduce_mul
__sec_reduce_all_zero__sec_reduce_all_nonzero
__sec_reduce_any_nonzero
```

```
__sec_reduce_max__sec_reduce_min
__sec_reduce_max_ind__sec_reduce_min_ind
```

ユーザー定義のリダクション関数を作成することもできます。

```
typefn(typein1, type in2); // スカラー・リダクション関数
out = __sec_reduce(fn, identity_value, in[x:y:z]);
```

複数のランクのリダクションは実行コンテキストに基づきます。

```
sum = __sec_reduce_add(a[:][:]); // 配列 a 全体の合計を計算
sum_col[:]= __sec_reduce_add(a[:][:]); // 列の合計を計算
```

インテル® Cilk™ Plus の配列表記コンポーネントの使用

配列表記は次のような場合に使用します。

- 配列で演算を実行したい場合
- 本質的なデータ並列性セマンティクスにより実現可能な、より高いパフォーマンスに関心がある場合
- 同じデータで並列演算とシリアル演算を組み合わせたい場合
- 実行の制御よりも、並列実行を意図的に表現することに関心がある場合
- コンパイラーにコア、スレッド、および SIMD 実行リソースの最適な使用の判断を委ねる場合。慎重に使用することで、構文による制御も可能です。

演習 3: インテル® Cilk™ Plus を使用した行列転置と平方根の合計

難易度: 初級
演習プロジェクトの設定: Microsoft* Visual Studio* 2008 (64 ビット・プロジェクト)
必要な資料:

- インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド:
  - C:\Program Files (x86)\Intel\ Parallel Studio 2011\Composer\Documentation\ en_US\compiler_c\cl\index.htm (インテル® Parallel Studio 2011 がインストールされていると想定)

目的: この実践演習では、インテル® Cilk™ Plus のキーワード、cilk_for を使用して複数のプロセッサー・コアで並列化を達成する方法を紹介します。この演習では、インテル® Cilk™ Plus を使用して、100x100 の行列転置を行うループのパフォーマンスを向上させます。

インテル® PBB を使用する理由: インテル® Cilk™ Plus には、for ループや分割統治アルゴリズムが利用できる場合に、2 種類の異なるパフォーマンス・ゲインを得るための簡単な API が備わっています。複数のプロセッサー・コアにまたがる並列化は、cilk_for などのインテル® Cilk™ Plus の並列化キーワードで達成し (演習 1)、シングルコアでのデータ並列化は、プロセッサー上でハードウェアによるベクトル化を有効にするインテル® Cilk™ Plus の配列表記または要素関数 (演習 2) を使用して達成します。

手順:

インテル® Cilk™ Plus について理解するため、この演習に取り組む前に Cilk_Plus_5min_Intro ビデオを参照してください:http://software.intel.com/en-us/videos/introduction-to-intel-cilk-plus/ (英語)

第 1 部: ベースラインの作成

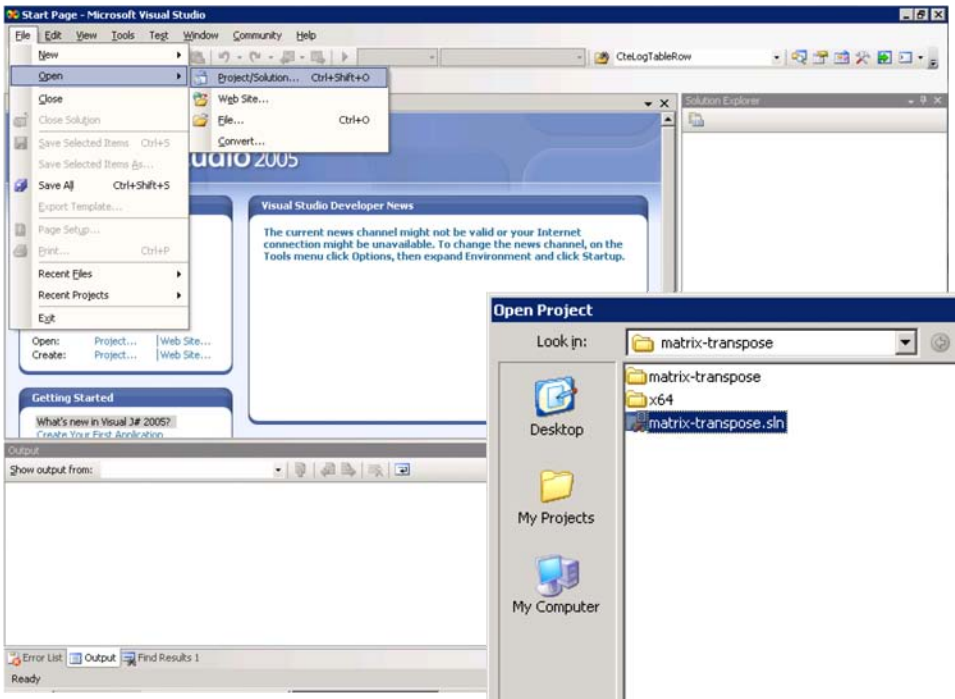
2 つの行列転置を行う C++ コードがあります。1 つは倍精度の配列に対するもので、もう 1 つは正当性チェックの一部として最初の転置の結果に対するものです。このコードは、matrix-transpose.cpp で定義されています。2 つの倍精度の交換を行う転置コードに、平方根の合計を求める関数が入れ子されています。その後、計算した合計を分解して、その正当性の検証を行います。このコードは、sum_squareroot.cpp、sum_squareroot.h、subtract_squareroot.cpp、subtract_squareroot.h にあります。

1. C:/IDF LAB/Cilk Plus から matrix-transpose.zip をローカル・ディレクトリーにコピーします。
2. matrix-transpose.zip をローカル・ディレクトリーに展開します。
3. Microsoft* Visual Studio* 2008 で matrix-transpose.sln ソリューションを開きます。[File (ファイル)] メニューの [Open (開く)] > [Project/ Solution (プロジェクト/ソリューション)] で開くことができます。ソリューションは、事前に設定されているため、何も変更する必要はありません (図7 と図 8)。

   プロジェクトが Release x64 モードであることと、3 つのヘッダーファイル (cilktime.h、subtract_squareroot.h、sum_squareroot.h) が表示されていることを確認してください。matrix-transpose.cpp、sum_squareroot.cpp、subtract_squareroot.cpp を含む 3 つのソースファイルがあることも確認してください (図 9)。
4. [Build (ビルド)] > [Rebuild Solution (ソリューションのリビルド)] でソリューションをリビルドします。エラーがないことを確認してください (図 10)。
5. 作成された matrix-transpose. exe を実行します。Microsoft* Visual Studio* IDE で Ctrl+F5 を押すか、または [Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] を選択します。ランタイム・エラーがないことを確認し、 実行にかかった時間を記録します。

   1 番目の転置時間:_____

   2 番目の転置時間:_____


Start Page - Microsoft Visual Studio
File Edit View Tools Test Window Community Help
New
Open
Close
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Export Template...
Page Setup...
Print... Ctrl+P
Recent Files
Recent Projects
Exit
Project/Solution... Ctrl+Shift+O
Web Site...
File... Ctrl+O
Convert...
Open: Project... Web Site...
Create: Project... Web Site...
Visual Studio Developer News
The current news channel might not be valid or your Internet connection might be unavailable. To change the news channel, on the Tools menu click Options, then expand Environment and click Startup.
Getting Started
What's new in Visual J# 2005?
Create Your First Application
Output
Show output from:
Error List
Output
Find Results 1
Ready

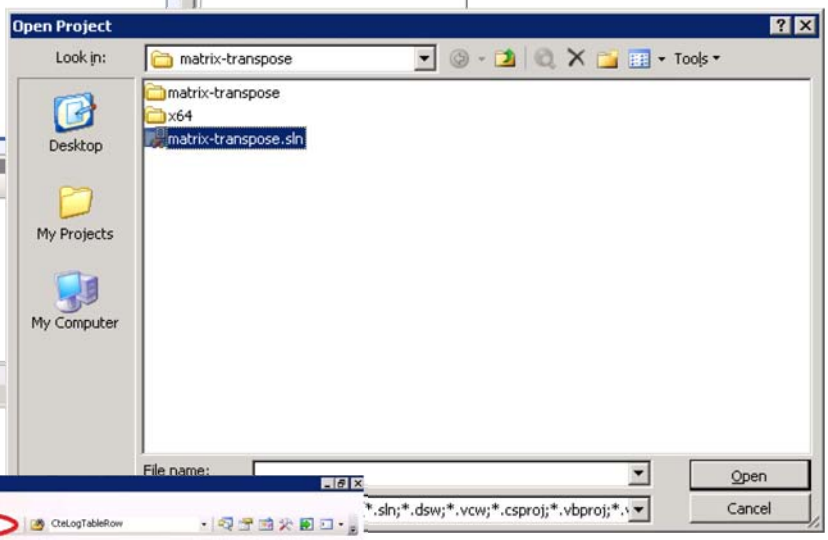

Open Project
Look in:
matrix-transpose
Tools
Desktop
My Projects
My Computer
matrix-transpose
x64
matrix-transpose.sln
File name:
Open
Cancel

図 7

図 8

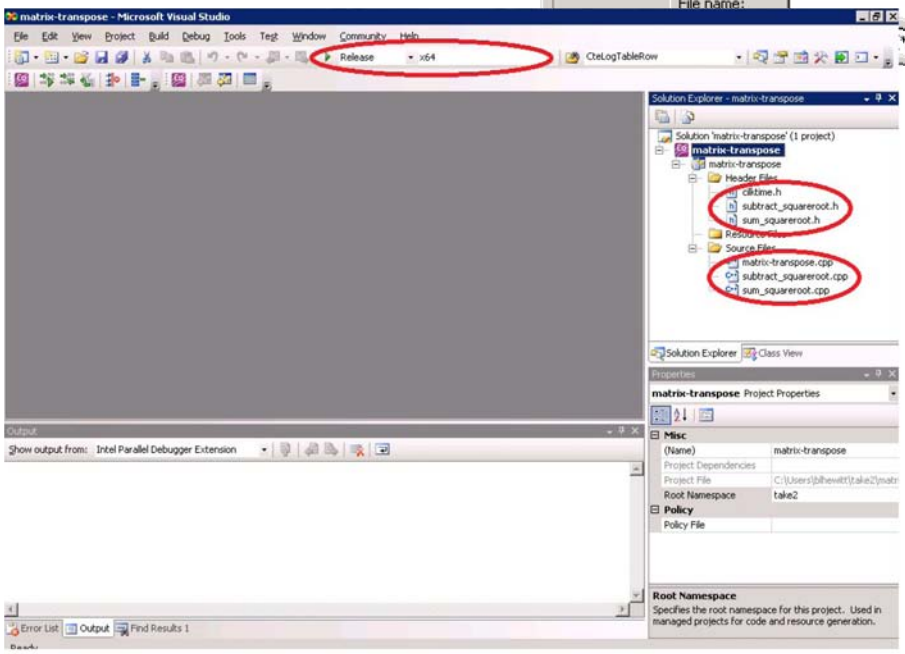

matrix-transpose - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test Window Community Help
Release
x64
CteLogTableRow
Solution Explorer - matrix-transpose
Solution 'matrix-transpose' (1 project)
matrix-transpose
matrix-transpose
Header Files
clktime.h
subtract_squareroot.h
sum_squareroot.h
Source Files
matrix-transpose.cpp
subtract_squareroot.cpp
sum_squareroot.cpp
Solution Explorer
Class View
Properties
matrix-transpose Project Properties
Misc
(Name) matrix-transpose
Project Dependencies
Project File
Root Namespace take2
Policy
Policy File
Root Namespace
Specifies the root namespace for this project. Used in managed projects for code and resource generation.
Output
Show output from: Intel Parallel Debugger Extension
Error List
Output
Find Results 1

図 9

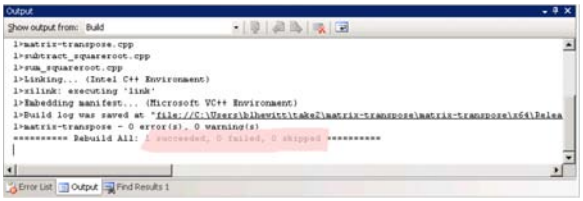


図 10

第 2 部: メインの転置ループの最適化

ここでは、複数のコアにまたがる並列化の機会を有効利用するためにコードを最適化します。第 1 部で行ったパフォーマンス解析から、このコードの主な hotspot は sum_squareroot.cpp の sum_squareroot() 関数であることが分かりました。パフォーマンスを向上するには、matrix-transpose.cpp で定義されている matrix_transpose() 関数で繰り返し呼び出している sum_squareroot() のワークを複数の CPU コアで分散する必要があります。ここでは、タスクの並列化を利用することで、パフォーマンスが向上します。

1. この演習の資料セクションにあるリンクからインテル® Composer ユーザー・リファレンス・ガイドを開きます。ユーザー・リファレンス・ガイドの「並列アプリケーションの作成」 > 「インテル® Cilk™ Plus の使用」 > 「インテル® Cilk™ Plus キーワード」でキーワードの説明を参照してください。また、ユーザー・リファレンス・ガイドの「並列アプリケーションの作成」 > 「インテル® Cilk™ Plus の使用」 > 「インテル® Cilk™ Plus キーワード」 > 「cilk_for」で cilk_for の説明も参照してください。
2. matrix_transpose() は、sum_squareroot() を繰り返し呼び出し、それから倍精度配列の 2 つの項目を入れ替えて行列転置を行う入れ子された for ループで構成されています。外側の for ループはデータに依存しないため (つまり反復の結果が前の反復に依存しないため、潜在的なデータ競合の可能性がないので)、並列で実行できます。#include <cilk/cilk.h> を追加して、matrix_transpose() の外側の for ループを cilk_for に置換して、個々の反復を複数の CPU コアで並列に実行します。
3. [Build (ビルド)] > [Build Solution (ソリューションのビルド)] でアプリケーションをビルドしてから、[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] または Ctrl+F5 でアプリケーションを再度実行します。[タスク マネージャ] を実行し (画面下の [タスク] バーを右クリックし、[タスク マネージャ] をクリックします)、[パフォーマンス] タブを調べて複数のコアが使用されていることを確認します。作成された実行ファイルの正当性を確認して、パフォーマンスの向上を記録します。

   1 番目の転置時間:

   _____

   2 番目の転置時間:

   _____
4. メインの転置ループを変換した結果、速度はどれぐらい向上しましたか? (第 1 部のステップ 6 と比べてみてください)

結論: インテル® Cilk™ Plus は、わずかな時間と労力でパフォーマンスとスケーリングにおいて大きな効果を発揮しました。1 行のコードを変更しただけで、ほぼ直線的なスケーリングを達成していることが分かります。

演習 4: インテル® Cilk™ Plus を使用した行列転置と平方根の合計

難易度: 上級
演習プロジェクトの設定: Microsoft* Visual Studio* 2008 (64 ビット・プロジェクト)
必要な資料:

- インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド:
  - C:\Program Files (x86)\Intel\Parallel Studio2011\Composer\Documentation\en_US\ compiler_c\cl\index.htm (インテル® Parallel Studio 2011 がインストールされていると想定)

目的: この演習では、インテル® Cilk™ Plus の配列表記と要素関数 __declspec(vector) を使用して、インテル® コンパイラーでプログラムに最も効率の良いコードを作成する方法を紹介します。この演習の最後に、インテル® Cilk™ Plus を使用して、100x100 の行列転置を行うループのパフォーマンスを向上させます。

インテル® PBB を使用する理由: for loop や分割統治アルゴリズムがある場合、インテル® Cilk™ Plus により簡単な API が提供され、2 種類のパフォーマンス・ゲインを達成できます。プロセッサー・コアにまたがる並列化は、cilk_for などのインテル® Cilk™ Plus の並列化キーワードで達成し (演習 1)、シングルコアでのデータ並列化は、プロセッサー上のベクトル化ハードウェアの使用を有効にするインテル® Cilk™ Plus 配列表記または要素関数 (演習 2) を使用して達成します。

手順:

第 1 部: 合計処理の最適化

ここでは、複数のコアにまたがる並列化の機会を有効利用するためにコードを最適化します。前に行ったパフォーマンス解析から、このコードの主な hotspot は sum_squareroot.cpp の sum_squareroot() 関数であることが分かりました。インテル® Cilk™ Plus の配列表記と要素関数を使用して、SIMD プロセッサー命令でこれらの演算を最適化する必要があります。

1. この実践演習では、演習 1 で最適化した matrix-transpose プロジェクトを引き続き使用します。matrix- transpose.exe (**[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)]**) を実行し、実行にかかった時間を記録します。

1 番目の転置時間:

_____

2 番目の転置時間:

_____

2. この演習の資料セクションにあるリンクからインテル® Composer ユーザー・リファレンス・ガイドを開きます。『インテル® C++ Compiler 12.0 ユーザー・リファレンス・ガイド』の「並列アプリケーションの作成」 > 「インテル® Cilk™ Plus の使用」 > 「配列表記 (アレイ・ノーテーション) の拡張」 > 「配列表記 (アレイ・ノーテーション) の C/C++ 拡張プログラミング・モデル」でインテル® Cilk™ Plus の配列表記の説明を参照してください。特に、最初の例の構文に注目してください。また、さらに理解を深めるために「並列アプリケーションの作成」 > 「インテル® Cilk™ Plus の使用」 > 「配列表記 (アレイ・ノーテーション) の拡張」 > 「配列表記 (アレイ・ノーテーション) の C/C++ 拡張の概要」も参照してください。
3. sum_squareroot() の最初の for ループは、sqrt() 算術関数の呼び出しと一連の配列へのアクセスを行います。インテル® Cilk™ Plus の配列表記により、この for ループに対してコンパイラーでより効率的なデータ並列コードを生成することができます。内側の for ループ (インデックス j のループ) を配列全体の += 増分処理に置換し、下限 0、長さ、ストライド 1 で代入します。
4. アプリケーションを [Build (ビルド)] > [Build Solution (ソリューションのビルド)] でビルドし [Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] で実行して、作成された実行ファイルの正当性を検証して、パフォーマンスの向上を記録します。

   1 番目の転置時間:

   _____

   2 番目の転置時間:

   _____

5. 配列表記を使用して、コンパイラーのベクトル化を有効にした結果、速度はどれぐらい向上しましたか? (第 1 部のステップ 1 と比べてみてください)

第 2 部: 減算処理の最適化

1. ユーザー・リファレンス・ガイドの**「並列アプリケーションの作成」 > 「インテル® Cilk™ Plus の使用」 > 「要素関数」**でインテル® Cilk™ Plus 要素関数と __declspec(vector) の説明を参照してください。
2. sum_squareroot.cpp の 2 番目の for ループは、単純なスカラー関数である、外部関数 subtract_squareroot() を呼び出します。この関数をインテル® Cilk™ Plus の要素関数として定義して、コンパイラーが複数の配列要素にこの関数を同時に適用し、コードをベクトル化するようにできます。これは、subtract_squareroot の宣言と定義の両方で __declspec(vector) 表記を使用することで行えます。必要に応じてコードを変更します (subtract_squareroot.cpp と subtract_squareroot.h)。
3. アプリケーションを [Build (ビルド)] > [Build Solution (ソリューションのビルド)] でビルドし、[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] で実行して、作成された実行ファイルの正当性を検証して、パフォーマンスの向上を記録します。

   1 番目の転置時間:

   _____

   2 番目の転置時間:

   _____

4. 要素関数を使用して、コンパイラーが subtract_squareroot を複数の配列要素に対して並列に適用できるようにした結果、速度はどれぐらい向上しましたか? (第 1 部のステップ 4 と比べてみてください)

結論: インテル® Cilk™ Plus は、わずかな時間と労力でパフォーマンスとスケーリングにおいて大きな効果を発揮しました。1 行のコードを削除し、3 行のコードを変更しただけで、パフォーマンスが大幅に向上していることが分かります。

インテル® Array Building Blocks (インテル® ArBB)

インテル® ArBB は、何通りにも定義できます。ライブラリーに裏付けられた API であり、特別なプリプロセッサーを必要としません。インテル® ArBB は、プログラミング言語拡張です (つまり、ホスト言語を必要とする「補足言語」です)。

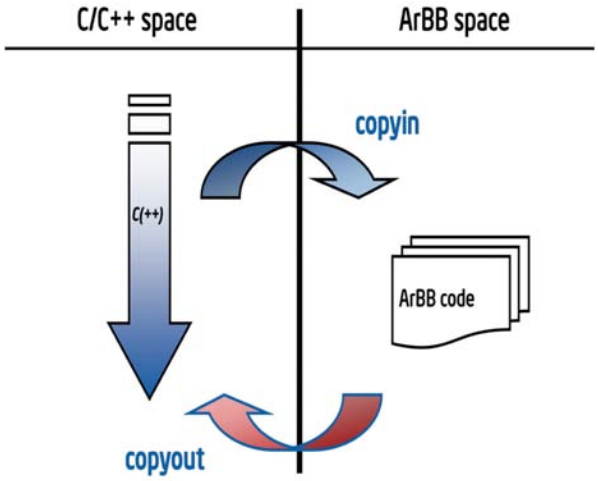
インテル® ArBB は不規則な行列や疎行列などの複雑なデータ並列化に対応するよう C++ を拡張します。次のような特性があります。

- 移植可能な並列開発プラットフォーム
- ハードウェアに依存しない並列計算
- シーケンシャル・セマンティクス、優れたデータ局所性
- デフォルトでの安全性: デッドロックなし、データ競合なし

インテル® ArBB は、計算を多用するデータ並列アプリケーション(ベクトル算術演算などがしばしば関わる) に最適です。

この API は、汎用データ並列プログラミング・ソリューションを形成します。アプリケーション開発者は、特定のハードウェア・アーキテクチャーへの依存から解放され、既存の C++ 開発ツールと統合し、並列アルゴリズムを高レベルで指定できます。

図 11



インテル® ArBB は、モジュール化によるオーバーヘッドを取り除くことができる動的コンパイルを基にしています。計算処理の高レベルな記述を効率良い並列実装に変換することで、SIMD とスレッドレベルの並列化の双方をうまく利用することができます。

主な機能

インテル® ArBB プログラムでは、2 つのコンパイル処理が行われます (図 11)。1 つはインテル® アーキテクチャー (IA) のバイナリー配布用の C++ コンパイルです。2 つ目はハイパフォーマンス実行のための動的コンパイルで、インテル® ArBB の動的エンジンによって行われます。データが複数のコア向けに最適化されるよう、データを C++ 空間からインテル® ArBB 空間にコピーする必要があります。データは、隔離されたデータ空間に保たれ、コレクション・クラスによって管理されます。

図 12は、プラットフォーム固有の規則的なデータ構造と不規則なデータ構造を表したものです。インテル® ArBB のコレクション型、スカラー型に対する演算を使用して、このような構造の計算処理を C++ 関数として表現します。

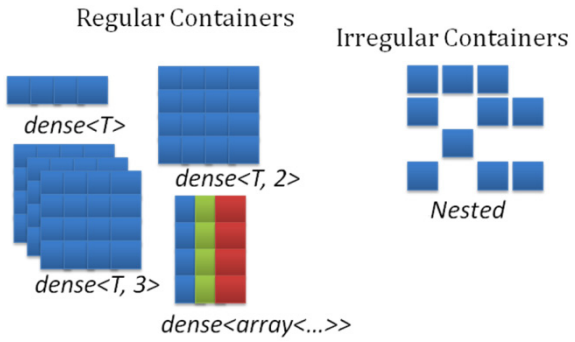


図 12

演習 5: インテル® Array Building Blocks を使用した大規模な配列の Modular 計算

難易度: 初級
演習プロジェクトの設定: Microsoft* Visual Studio* 2010 (64 ビット・プロジェクト)

目的: この実践演習では、インテル® Array Building Blocks (インテル® ArBB) の dense コンテナーと関数の使用法を紹介します。プログラムでは、dense コンテナーを使用して、1024 要素の 2 つの配列を格納します。この演習問題では、これら 2 つの配列の 2 乗残差 (SSD) を計算する関数を記述し、インテル® ArBB を使用して、プロセッサーのベクトル化機能と SIMD 機能を活用できるようその関数を並列に呼び出します。

インテル® PBB を使用する理由: 従来のプログラミング手法を使用して記述した場合、このプログラムでは 2 つの大きな配列が作成され、これらの配列の計算にはシリアル実行されるループが使用されます。しかし、この計算は、並列化、ベクトル化が可能なため、インテル® ArBB を使用して、わずかなコードで、より優れたパフォーマンスを引き出すプログラムを一から作成することができます。インテル® ArBB の高レベルの抽象化により、開発者は個々の要素の操作ではなく、配列全体の観点で考えることができます。インテル® ArBB のランタイムを使用することで、ハードウェアで利用可能な並列化やベクトル化が自動で得られます。

手順:

インテル® ArBB について理解するため、この演習に取り組む前に ArBB_5min_Intro ビデオを参照してください: http://software.intel.com/en-us/videos/introduction-to-intel-array-building-blocks/ (英語)

第 1 部: ソリューションの設定

1. ArBBLab.zip ファイルを C:/IDF LAB/Array Building Blocks からローカル・ディレクトリーにコピーします。
2. ArBBLab.zip をローカル・ディレクトリーに展開します。
3. ArBBLab フォルダーの中の tutorial.bat ファイルを実行して Microsoft* Visual Studio* 2010 でソリューションを開きます。
4. 4. Microsoft* Visual Studio* ソリューションが表示されます。このソリューションは、すでに ArBB を使用できるように設定されていますが、コンパイルは行われていません。
5. ソース・コード・エディターで tutorial.cpp ファイルを開きます。インテル® ArBB を使用するためのヘッダーと名前空間宣言がすでに含まれていることに注意してください。
6. tutorial.cpp の main 関数のコードとコメントを確認します。インテル® ArBB は、配列や配列操作についてより直感的な別の考え方を必要とする新しいプログラミング・モデルです。インテル® ArBB を使用するには、3 つの点が必要です。

- インテル® ArBB のメモリー空間に格納されるデータ構造を作成する。これはすでに作成されています。main で 2 つのインテル® ArBB dense コンテナー a と b を作成し、それらにサンプルデータをセットします。
- これらのコンテナーで行う操作を記述する。インテル® ArBB のランタイム・コンポーネントは、それらの操作を効率的な並列化およびベクトル化が施されたコードにコンパイルして、何度も実行できるようにそのコードをキャッシュします。これは、第 2 部で行います。
- インテル® ArBB コンテナーの操作を適切に呼び出す。これは、第 3 部で行います。

第 2 部: dense コンテナーで行う演算の記述

1. インテル® ArBB のガイドラインに沿った標準の C++ 関数を使用して、dense コンテナー a と b で行う演算を定義する必要があります。この演習で使用する関数は、sum_of_squared_differences で、24 行目で宣言されています。関数の宣言から、この関数の戻り型は void (インテル® ArBB の要件) で、結果 (result と呼ばれる) を保持する変数は参照によって渡されていることが分かります。

2. コンテナーで行う演算を記述します。インテル® ArBB では、個々の要素ではなくコンテナー全体の観点で考えることができます。そのため、演算はコンテナー a と b の観点から記述し、別のコンテナーを作成して結果を格納することができます。ここでは、インテル® ArBB の紹介ビデオにあるように 2 乗残差 (SSD) を計算します。tutorial.cpp の 32 行目に移動し、インテル® ArBB dense コンテナーの 2 乗残差 (SSD) を計算するコードを記述して、結果を "temp" という 3 つ目の dense コンテナーに格納します。26-31 行に役立つコメントがあります。また、ビデオ (3:59 分のあたり) も参考にしてください。

3. 次に、もう 1 つ演算を追加します。インテル® ArBB ビデオの例では、2 つのインテル® ArBB dense コンテナーの 2 乗残差を計算し、結果を 3 つ目のインテル® ArBB dense コンテナーに格納しています。標準的なプログラミングでは、dense コンテナーは配列であることを思い出してください。上記のステップ 2 が終わった時点およびビデオの例では、配列 a の各要素と対応する b の要素の 2 乗残差を保持している dense コンテナーがあります。この演習のプログラムでは、a と b の 2 乗残差の合計を計算します。これを行うには、2 乗残差を格納している dense コンテナー (tutorial.cpp の "temp") に格納されている 2 乗残差の値を調べ、これらの値を合計する必要があります。

配列の合計処理は、リダクションの一種です。つまり、値セットに繰り返し演算を適用して、1 つの値に「まとめ」ます。これは、add_reduce() というインテル® ArBB 関数で行うことができます。従来の方法を使用して temp の値を 1 つずつ確認し、合計する代わりに、add_reduce を呼び出します。add_reduce は利用可能なすべてのハードウェア・リソースを使用して、この処理を行います。add_reduce のパラメーターは 1 つだけで、合計する dense コンテナーを受け取り、同じ型の結果を返します。

37 行目で dense コンテナー temp を使用して add_reduce を呼び出し、結果を result パラメーターに格納します。34-36 行目に役立つコメントがあります。

第 3 部: インテル® Array Building Blocks 関数の呼び出し

1. これまでの手順で、main でインテル® ArBB dense コンテナーを作成し、sum_ of_squared_differences 関数でコンテナーの演算を定義しました。次に、インテル® ArBB ランタイムを適切に呼び出して、パフォーマンスを最大限に引き出し、利用可能なハードウェア・リソースを使用しながら、dense コンテナーの演算を行います。インテル® ArBB ランタイムは、次のように call 関数を使用して呼び出します。call(& ユーザー定義関数)(関数に渡すパラメーターのリスト);

   ここでは、ユーザー定義の関数は sum_of_squared_differences で、パラメーターは a、b、result です。これはコード行の 78 行目です。71-77 行目に役立つコメントがあります。また、ビデオ (3:49-4:25 のあたり) で使用されている例のコードの最後の行も参考にしてください。

2. 実行時に、インテル® ArBB は、ハードウェアのベクトル化と SIMD 機能を使用するよう、関数の just-in-time コンパイルを行います。一般にパラメーターは何百、何千という要素からなる dense コンテナーであるため、関数のコンパイルはキャッシュされ、指定したパラメーターに対して並列に呼び出されます。main には、画面に結果を出力するコードが含まれています。[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、エラーがある場合は修正します。

3. [Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] でプログラムを実行して、結果を参照します。解は 1024 になります。

結論: インテル® ArBB 関数では、配列に対する処理をどのように行うかではなく、配列に対して何を行うかを指定するカスタムのアルゴリズムを作成することができます。インテル® ArBB は、対象となるアーキテクチャー向けにベクトル化およびスレッド化されたコードを生成します。

演習 6: インテル® Array Building Blocks のスコープタイマーとランタイム・コンパイラーの動作

難易度: 中級
演習プロジェクトの設定: Microsoft* Visual Studio* 2010 (64 ビット・プロジェクト)
目的:この演習では、インテル® ArBB の dense コンテナーと関数についてより実践的な内容を学習します。また、ランタイム・コンパイラーでスコープタイマー (Scoped Timer) を使用してテストする方法も習得します。

インテル® PBB を使用する理由: このプログラムでは、インテル® ArBB 関数を使用して、2 つの大規模な配列の基本的な計算 (ここでは減算/乗算と加算-リダクション) をキャプチャします。通常、loop での計算中に配列の個々の要素を 1 つずつ確認する必要はありません。すべてのデータ要素に対して同じ処理が行われるため、ランタイム・コンパイラーは利用可能なすべてのリソースを活用して、並列に計算を実行します。この演習では、インテル® ArBB 関数の実行時間の測定方法とランタイムコンパイルが行われる正確なタイミングを学びます。インテル® ArBB 関数では、配列に対する処理をどのように行うかではなく、配列に対して何を行うかを指定するカスタムのアルゴリズムを作成することができます。インテル® ArBB は、対象となるアーキテクチャー向けにベクトル化およびスレッド化されたコードを生成します。

手順:

第 1 部: dense コンテナーのサイズと内容の変更

1. この演習では、演習 1 で記述した配列計算プログラムを引き続き使用します。第 1 部では、コードの変更を通して、インテル® ArBB dense コンテナーの扱いにより慣れるようにします。まず、dense コンテナーのサイズを変更することから始めます。(メモリーを考慮して) 1024 × 200 を超えない任意のサイズを選択し、49 行目のサイズの値を変更します。

2. 次に、dense コンテナーの要素を変更します。インテル® ArBB の dense コンテナーでは、一度にデータ構造全体を変更することも、read_write_range を使用して従来の配列のように 1 つずつ変更することもできます。ここでは、59-60 行目で 各 dense コンテナーに対して read_write_range が作成されています。

3. 演習 1 では、63 行目と 64 行目で dense コンテナーに 1s (a の場合) と 2s (b の場合) が代入されています。この演習では、従来の配列と同様に、read_write_range を使用してコンテナーの各要素に値を代入します。まず、63 行目と 64 行目を削除します。(これらの行には 2 つの std::fill 呼び出しを配置します。)

4. 63 行目の周辺に dense コンテナー a と b の各要素に初期値を代入するためのコードを追加します。任意の値を代入できます。ただし、dense コンテナーはインテル® ArBB の f32 型なので、32 ビットの浮動小数点値にする必要があります。以下は、コンテナーの範囲内にある要素に繰り返し値を代入する方法の例です。以下の値をそのまま使用することも、独自の値を使用することもできます。

```
const double multiplier = 2.0;
for (std::size_t i = 0; i != size; ++i) {
    range_a[i] = static_cast<float>(multiplier * i);
    range_b[i] = static_cast<float>(multiplier * i * 2.0);
}
```

第 2 部: タイミング実験の実施

1. ここでは、main のコードの一部を変更します。始めに、演習 1 で使用したインテル® ArBB 関数を実行し、結果を画面に出力するコードを main から削除します。f64 型の result 変数を作成した後から catch 節の前までのすべてのコードが対象です。main で "**** " とマークされたコメントを探して、その間にあるコードをすべて削除します。

2. インテル® ArBB 関数を複数回実行する下記のコードを確認して、実行時間を測定します。その後、このコー

ドを main の演習 1 のコードを削除した場所にコピーします。(sum_of_squared_ differences 関数の呼び出しを行うようにコメントされている行に追加してください。)

```
double time = std::numeric_limits<double>::max();

//必要に応じて反復回数を変更します。
//(for ループでは初め 10 に設定されています。)
for (int i = 0; i < 10; i++)
{//for ループ

double time_i;

{ // scoped_timer を使用してセクションをプロファイルします。
const scoped_timer timer(time_i);

//<追加 - ArBB 関数の sum_of_squared_differences を呼び出します。
//演習 1 第 3 部を参照>

}//スコープタイマー

time = std::min(time, time_i);
std::cout << "Time " << (i + 1) << ":" << time_i << " ms\n";
}//for ループ

std::cout << "\n\n";
std::cout << "Time :" << time << " ms\n";
std::cout << "Result:" << value(result) << '\n';
}//try
```

3. インテル® ArBB 関数の呼び出しを追加したら、**[Build (ビルド)] > [Build Solution (ソリューションのビルド)]** でコードをビルドし、エラーがある場合は修正します。
4. [Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)] でコードを実行します。結果はいかがでしたか?

   インテル® ArBB 関数の呼び出し数:
   _____
   実行最短時間:
   _____
   最初の呼び出しの実行時間:
   _____

5. インテル® ArBB ランタイム・コンパイラーは効率的なインテル® ArBB 関数のコンパイルバージョンを作成し、今後の使用のためにキャッシュします。関数を繰り返し呼び出すことで、このランタイム・コンパイラーの動作の効果を確認することができます。インテル® ArBB 関数内部の配列で実行する処理、実行回数、データサイズ、を変更して、いろいろな組み合わせを試してみてください。また、インテル® コンパイラーと Microsoft* Visual Studio* コンパイラーはいつでも切り替えが可能です。様々なテストパターンを試してみて、実行時間に差が起こる原因を検証してみてください。

結論: インテル® ArBB は、データを構築して、そのデータで行う計算を記述する各種の方法を提供します。インテル® ArBB のランタイム・ライブラリーは、インテル® ArBB 関数を並列で最も効率の良いバージョンにコンパイルして、それをキャッシュし、パフォーマンスを最大限に引き出します。

インテル® PBB についてのまとめ

次世代の並列プログラミング・モデルであるインテル® PBB

インテル® TBB は、人気があり幅広く使用されている実証済みのモデルで、業界トップの会社からも推奨されています。インテル® ArBB は、現在ベータ版が提供されている新しいテクノロジーで、スレッド化にはインテル® TBB ランタイムを、アーキテクチャー向けに調整された SIMD 命令セットの生成には独自のランタイムを使用してビルドされます。インテル® Cilk™ Plus は、インテル® コンパイラーに組み込まれた革新的なテクノロジーで、できるだけ多くの並列化の機会を容易に利用できるようにします。ほかのモデルが使用されていない新しい開発プロジェクトでインテル® PBB を是非使用してみてください。

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を行わ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 (インテル® 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 #20101101