インテル® Cilk™ Plus による並列化への近道

はじめに

本ガイドは、インテル® Cilk™ Plus を使用して、簡単にコードをベクトル化および並列化 (スレッド化) する方法について説明します。機能の概要を紹介し、エンドツーエンドの例を用いて使用方法を示します。インテル® Cilk™ Plus は、インテル® Parallel Studio XE スイートで利用可能なインテル® C++ コンパイラーの機能です。

コンパイラーの拡張によりタスクとデータの並列化が容易に

インテル® Cilk™ Plus は、インテル® C++ コンパイラーで実装される C/C++ 言語にシンプルな言語拡張を追加して、データとタスクの並列化を表現します。インテル® C++ コンパイラーは、インテル® Parallel Studio XE に含まれます。この強力な言語拡張は、使いやすく、幅広いアプリケーションに簡単に適用できます。インテル® Cilk™ Plus には、次の機能と利点があります。

機能	利点
単純なキーワード	単純で強力なタスク並列化表現 • cilk_for: for ループを並列化します。 • cilk_spawn: 並列実行の開始を指定します。 • cilk_sync: 並列実行の終わりを指定します。
ハイパーオブジェクト (レデューサー)	各タスクに対して自動で共有リダクション変数のビューを作成し、タスクの完了後に共有変数に戻すことによって、タスク間の共有変数の競合をなくします。
配列表記 (アレイ・ノーテーション)	配列の全体または部分とその操作のデータ並列化です。
SIMD 対応関数	配列の全体または部分に適用される、関数全体や演算のデータ並列化を有効にします。

インテル® Cilk™ Plus はオープン仕様であるため、ほかのコンパイラーでもこの画期的な新しい C/C++ 言語機能を実装できます。詳細は、www.cilkplus.org を参照してください。

インテル® Cilk™ Plus の使用

インテル® Cilk™ Plus は、次のような場合に使用します。

- 配列に対する操作の実行制御よりも、並列化の可能性を簡潔に表現する
- 本来のデータ並列性セマンティクスによって、より優れたパフォーマンスを得る – 配列表記
- マネージド・デプロイメントではなく、ネイティブ・プログラミングを使用する – 同じデータで並列操作とシリアル操作を組み合わせる意図を表現

インテル® Cilk™ Plus では、並列処理の最適化と管理にコンパイラーが関与します。次のような利点があります。

- キーワードや直感的な構文により言語に統合されるためコードが書きやすく、理解しやすい。
- コンパイラーが言語セマンティクスの実装、一貫性のチェック、プログラミング・エラーのレポートを行う。
- コンパイラー・インフラストラクチャーとの統合により、コンパイラーによる多くの最適化を並列コードに適用可能。コンパイラーは、インテル® Cilk™ Plus の 4 つの機能を理解しているため、コンパイラーによる診断、最適化、ランタイム・エラー・チェックを利用できる。

パフォーマンスとスケーリング

ここでは、実際のアプリケーションを使用した例を 2 つ紹介します。1 つ目の例は、インテル® Cilk™ Plus を利用したモンテカルロ・シミュレーションです。配列表記を用いてコンパイラーによるベクトル化を行い、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) を使用してデータ並列パフォーマンスを最大限に引き出します。さらに cilk_for を追加してシミュレーションのドライバー関数を並列化することにより、タスクレベルの並列処理のために複数のプロセッサー・コアを最大限活用します。2 つ目の例は、配列表記によりレンダリング関数をベクトル化し、cilk_for でタスクを並列化して複数のコアにワークを分配する、ビジュアル・コンピューティング・アルゴリズムです。どちらのアプリケーションでも、わずかな作業でアプリケーションのスピードが大幅に向上します。

	スカラーコード	インテル® Cilk™ Plus のデータ並列処理	インテル® Cilk™ Plus のタスク並列処理	インテル® Cilk™ Plus のデータおよびタスク並列処理
モンテカルロ・シミュレーション [1]	4.477 秒	1.450 秒 =3.09 倍向上	1.264 秒 =3.54 倍向上 (4 コア、ハイパースレッディング無効)	0.390 秒 = 11.48 倍向上
AOBench	3.50 fps	4.21 fps = 1.20 倍向上	12.18 fps =3.48 倍向上	14.83 fps = 4.24 倍向上

[1] システム構成: インテル® Core™ i5-3550 プロセッサー 3.30GHz、4 コア、4GB RAM、Microsoft* Windows Server* 2008 R2 Enterprise SP1 (64 ビット)、インテル® Parallel Studio XE 2015 Composer Edition for C++ Windows*、Microsoft* Visual Studio* 2012。

実践

このガイドでは、インテル® Parallel Studio XE を使用して、インテル® Cilk™ Plus をアプリケーションに追加する方法を説明します。以下の例を紹介します。

- cilk_spawn キーワードと cilk_sync キーワードを使用した簡単な quick-sort の実装
- 配列表記構文と cilk_for キーワードを使用したモンテカルロ・シミュレーションとビジュアル・コンピューティング・アルゴリズムの実装
- インテル® Cilk™ Plus SIMD ベクトル化と SIMD 対応関数の使用法を示す簡単な入門チュートリアル

インテル® Parallel Studio XE のインストール

インテル® Parallel Studio XE のインストールと設定

1. インテル® Parallel Studio XE の評価版をダウンロードしてインストールします。

注: このガイドの例では、Windows* 上で Microsoft* Visual Studio* を使用していますが、インテル® Cilk™ Plus はインテル® Parallel Studio XE for Linux* のインテル® C++ コンパイラーを使用して Linux* でも利用できます。

サンプル・アプリケーションの入手

サンプル・アプリケーションのインストール

1. インテル® Parallel Studio XE のインストール・ディレクトリーでインテル® Cilk™ Plus のサンプル Cilk.zip を探します。一般的なインストールの場合、c:\Program Files (x86)\Intel\Composer XE 2015\Samples\en_US\C++\Cilk.zip にあります。
2. MonteCarloSample.zip サンプルファイルをローカルマシンにダウンロードします。このサンプルでは、モンテカルロ・シミュレーションのシリアル/スカラーカーネル、インテル® Cilk™ Plus の cilk_for を使用したドライバー関数、インテル® Cilk™ Plus の配列表記を使用したカーネル、インテル® Cilk™ Plus の cilk_for と配列表記の両方を使用したカーネルを紹介します。
3. CilkPlus-AOBench サンプルファイルをローカルマシンにダウンロードします。このサンプルは、「アンビエント・オクルージョン」と呼ばれるアルゴリズムを用いて、3 つの動いている球体に光と影を描画します。このサンプルでは、アルゴリズムのシリアル/スカラー実装、インテル® Cilk™ Plus の配列表記でレンダリング・コードをベクトル化するデータ並列処理バージョン、インテル® Cilk™ Plus の cilk_for によりワークを複数のコアに分散するタスク並列処理バージョン (データ並列処理は行わない)、タスクとデータ並列処理の両方を行うバージョンを紹介します。バージョン (並列化) が進むにつれ、より大きなスピードアップが得られます。このプログラムは、リアルタイムでレンダリングされる複雑な照明効果を伴うアニメーションを描画します。4 つの異なるコード実装を使用してアニメーションを 4 回レンダリングし、インテル® Cilk™ Plus により実現可能なパフォーマンスの大幅な向上を示します。
4. 書き込み可能なディレクトリーまたはシステムの共有ディレクトリー (My Documents\cilksamples フォルダーなど) にそれぞれの zip ファイルを展開します。
5. すべてのサンプルの展開が終了すると、次のようなディレクトリーが作成されます。
    - Cilk – この下に複数のサンプル・ディレクトリーが作成されます。ここでは qsort を使用します。
    - MonteCarloSample
    - CilkPlus-AOBench

サンプルのビルド

1. 各サンプルには、Visual Studio* 2010、2012、2013 で使用できる Microsoft* Visual Studio* ソリューション・ファイル (.sln) があります。インテル® Parallel Studio XE 2015 に含まれるインテル® C++ コンパイラーの Release (最適化) 構成設定でソリューションをビルドします。
2. 各ソリューション・ファイルでインテル® Cilk™ Plus 言語拡張が有効になります。
   a. ソリューションを右クリックして、**[Properties (プロパティ)] > [Configuration Properties (構成プロパティ)] > [C/C++] > [Language [Intel C++] (言語 [インテル(R) C++])]** を選択します。
   b. **[Replace Intel Cilk Plus Keywords With Serial Equivalents (インテル(R) Cilk(TM) Plus キーワードをシリアル対応に置き換える)]** を **[No (いいえ)]** に設定します。
   c. **[Disable All Intel Language Extensions (すべてのインテル(R) 言語拡張を無効にする)]** を **[No (いいえ)]** に設定します。図 1 に qsort の **[Configuration Properties (構成プロパティ)]** を示します。

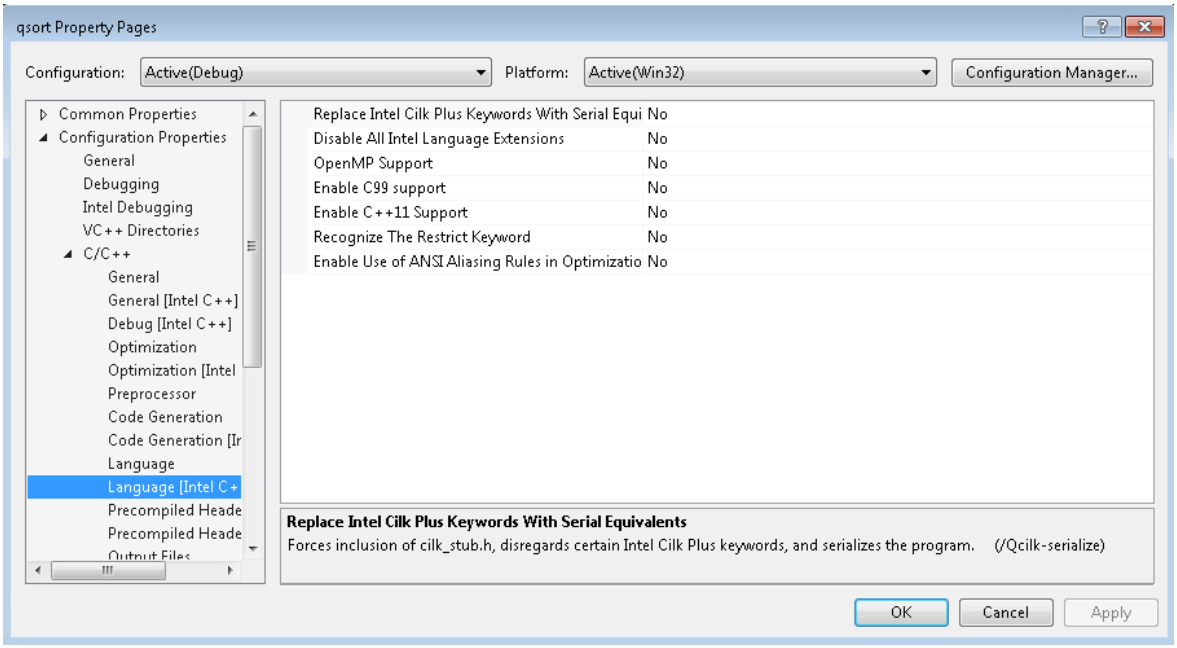

図 1

3. 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。

インテル® Cilk™ Plus のキーワードを使用した並列化の実装

cilk_spawn キーワードと cilk_sync キーワードを使用して、タスクを素早く並列化します。

1. Microsoft* Visual Studio* で qsort ソリューションを開きます。
2. qsort.cpp ファイルを開き、sample_qsort() ルーチンを表示します。

```
void sample_qsort_cilk(int * begin, int * end)
{
    if (begin != end) {
        --end;  // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
                          std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);    // move pivot to middle
        cilk_spawn sample_qsort_cilk(begin, middle);
        sample_qsort_cilk(++middle, ++end); // Exclude pivot and restore end
        cilk_sync;
    }
}
```

図 2

cilk_spawn と *cilk_sync* の使用法を見てみましょう。プログラマーに代わって、*cilk_spawn* がタスクの作成とスレッドへのスケジューリングを行い、quick-sort アルゴリズムを並列化しています。一方、*cilk_sync* は並列領域の終わり、つまりタスクが完了し、シリアル実行が再開されるポイントを示しています。この例では、cilk_spawn と cilk_sync の間の 2 つの sample_qsort() の呼び出しがインテル® Cilk™ Plus のランタイムで利用可能なリソースに応じて、並列に実行されます。

本ガイドの執筆時点で、インテル® Cilk™ Plus はいくつかのオープンソース・コンパイラーでも利用できます。詳細は、http://www.cilkplus.org/which-license を参照してください。また、他のツールベンダーでの採用を促進するために仕様を公開しています (「関連情報」を参照)。インテル® Cilk™ Plus キーワードの重要な特性は、キーワードを無効にした場合でもシリアル・セマンティクスによって、コードを変更することなくアプリケーションがシリアルモードで正しく実行される点です。**[Properties Pages (プロパティ ページ)]** でキーワードを有効/無効に設定することにより (前のセクションを参照)、シリアルおよび並列のランタイム・パフォーマンスと安定性を簡単に確認することができます (**[Replace Intel Cilk Plus Keywords With Serial Equivalents (インテル(R) Cilk(TM) Plus キーワードをシリアル対応に置き換える)]** を **[Yes (はい)]** に設定すると、並列実行が無効になります)。また、インテル® Cilk™ Plus のキーワードとレデューサー・ハイパーオブジェクトをサポートしていない別のコンパイラーでこれらを追加したファイルをコンパイルする場合は、キーワードが使用されているファイルの先頭に図 3 のコードを追加します。

```
#ifndef __cilk
#include <cilk/cilk_stub.h>
#endif
#include <cilk/cilk.h>
```

図 3

cilk_stub.h ヘッダーファイルは、ソースコードを変更することなく別のコンパイラーでファイルをコンパイルできるように、キーワードをコメントアウトします。レデューサー・ハイパーオブジェクトについては、『インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド』の「インテル® Cilk™ Plus」セクションとインテル® Cilk™ Plus のサンプル・ディレクトリーにあるその他のサンプル、そしてインテル® C++ コンパイラー・コード・サンプルにあるその他のサンプルを参照してください。インテル® Cilk™ Plus のキーワードとレデューサー・ハイパーオブジェクトを利用することで、バリアや同期コードを必要とすることなく、タスク間の共有変数を確実かつ容易に保護できます。前述の手順に従って、シリアルモードにしリビルドして、シリアルコードのパフォーマンスと先ほど記録した並列コードのパフォーマンスを比較してみてください。

インテル® Cilk™ Plus の SIMD 対応関数と cilk_for によるパフォーマンス - モンテカルロ・シミュレーション

次に、より複雑な例を見てみましょう。ここでは、モンテカルロ・シミュレーションでインテル® Cilk™ Plus の *cilk_for* キーワードを使用してメインのドイラバールーブを並列化し、配列表記を使用してシミュレーション・カーネルをベクトル化する方法を紹介します。

配列表記により、コンパイラーによって認識され、最適化、ベクトル化、あるいは場合によっては並列化が行われる構文を用いて、配列セクションの操作を行えます。基本の構文:

[<lower bound> : <length> : <stride>]

<下限>、*<長さ>*、*<ストライド>* はオプションで、それぞれ整数型です。配列宣言自体は、C/C++ の配列定義構文と変わりません。図 4 に配列操作と代入の例を示します。

```
Operations:
a[:] * b[:] // element-wise multiplication
a[3:2][3:2] + b[5:2][5:2] // matrix addition of 2x2 subarrays within a and b starting at
a[3][3] and b[5][5]
a[0:4][1:2] + b[0][1] // adds a scalar b[0][1] to each element of the array section in a

Assignments:
a[:][:] = b[:][2][:] + c;
e[:] = d; // scalar variable d is broadcast to all elements of array e
```

図 4

MonteCarloSample アプリケーションを見てみましょう。

1. Microsoft* Visual Studio* で MonteCarloSample ソリューションを開きます。
2. mc01.c ファイルを開き、Pathcalc_Portfolio_Scalar_Kernel 関数を表示します。このスカラーカーネル関数のスカラー配列宣言は以下のとおりです。

   __declspec(align(64)) float B[nmat], S[nmat], L[n];

   配列操作を利用するため、関数の操作を変更して、単一要素を処理する代わりに「ストライド」要素を処理します。まず、B、S、L を一次元配列から二次元配列に変更します。

   __declspec(align(64)) float B[nmat][vlen], S[nmat][vlen], L[n][vlen];

   サイズ (nmat または n) と長さ (vlen) を指定します。また、カーネル内の計算ループにある多数のスカラー累積と代入を処理するため、いくつかの配列も宣言する必要があります。

   変更後のコードは、配列セクションの指定子を除き、スカラーバージョンと非常によく似ています。カーネル内のループの 1 つを比較してみましょう (図 5 と図 6)。

   スカラーバージョン:

```
for (j=nmat; j<n; j++) {
    b = b/(1.0+delta*L[j]);
    s = s + delta*b;
    B[j-nmat] = b;
    S[j-nmat] = s;
}
```

図 5

   配列表記バージョン:

```
for (j=nmat; j<n; j++) {
    b[:] = b[:]/(1.0+delta*L[j][:]);
    s[:] = s[:] + delta*b[:];
    B[j-nmat][:] = b[:];
    S[j-nmat][:] = s[:];
}
```

図 6

   この簡単な変更により、一度に複数の要素を処理する配列操作を実装することで、コンパイラーが SIMD コードを使って、シリアル/スカラーカーネル実装と比べて大幅なスピードアップを達成できるようになりました。

3. 次に *cilk_for* を使用してループの呼び出しを並列化してみましょう。図 7 の Pathcalc_Portfolio_Scalar() 関数について考えてみます。ここで必要な処理は、*for* を *cilk_for* に置換するだけです。下記の Pathcalc_Portfolio_CilkArray() 関数では、この置換がすでに行われています。このコードをそのまま使用できます。

```
void Pathcalc_Portfolio_CilkArray(FPPREC *restrict z,
                                  FPPREC *restrict v,
                                  FPPREC *restrict L0,
                                  FPPREC * restrict lambda)
{
    int   stride = SIMDVLEN, path;
    DWORD startTime = timeGetTime();

    cilk_for (path=0; path<npath; path+=stride) {

              Pathcalc_Portfolio_Array_Kernel(stride,
                                              L0,
                                              &z[path*nmat],
                                              lambda,
                                              &v[path]);
    }

    perf_cilk_array = timeGetTime()-startTime;
}
```

図 7

4. CTRL + F5 キーを押して、プロジェクトをリビルドし実行します。

配列表記を追加して SIMD によるデータ並列化を有効にすることで、4 コアのシステム (詳細は 3 ページの表を参照) では、スカラーバージョンに比べてパフォーマンスが 3 倍向上しました。さらに、*cilk_for* キーワードを追加してカーネルの呼び出しを並列化することで優れたスケーリングが得られ、同一システムで 3.5 倍近くのスピードアップが達成されました。このことから、インテル® Cilk™ Plus がいかに簡単で強力なツールであるかが分かります。

インテル® Cilk™ Plus の SIMD 対応関数と cilk_for によるパフォーマンス - アンビエント・オクルージョン

このサンプルでは、ビジュアル・コンピューティングでインテル® Cilk™ Plus を使用する方法を紹介します。インテル® Cilk™ Plus の 配列表記と cilk_for キーワードを使用してレンダリング・コードをベクトル化し、複数の CPU コアにわたってワークを並列化します。

1. Visual Studio* 2012 を起動し、aobench_sdl\aobench_sdl.sln ソリューション・ファイルをロードします。
2. [Solution Explorer (ソリューション エクスプローラ)] の [Source Files (ソース ファイル)] フォルダーで、「parallel.cpp」をダブルクリックして開きます (図 8)。

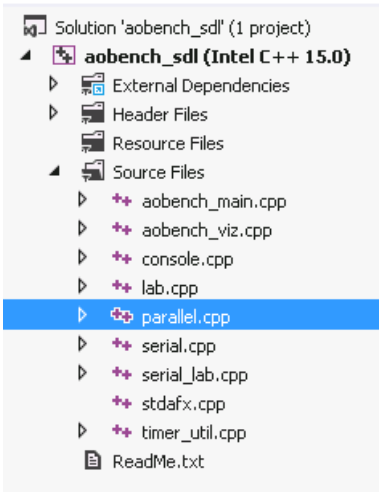

図 8

Ctrl + F キーを押して [Find and Replace (検索と置換)] ウィンドウを開き、render_vector_cilk を検索します (図 9)。ESC キーを押して、[Find and Replace (検索と置換)] ウィンドウを閉じます。

```
// Calls the vector code to render one line, and uses Cilk to render multiple lines in parallel.
void render_vector_cilk(float *fimg, int w, int h, int nsubsamples, SDL_Surface *surface, int offset_row, int offset_col)
{
    cilk_for (int y = 0; y < h; y++)
    {
        render_vector_x(fimg, w, h, nsubsamples, y);
        draw_line(surface, offset_row, offset_col, fimg, y, w);
    }
    draw_buffer(surface, offset_row, offset_col, fimg, w, h);
```

図 9

3. この関数には、1 つのアニメーション・フレームの各行をレンダリングするループがあります (「y」は各行の y 値で、0 からフレームの高さの範囲です)。ここでは、通常の「for」の代わりに、「_Cilk_for」キーワードを使用しています。これは、複数のタスクにより、すべてのループの反復を並列に実行できることを示しています。各タスクは、別々のコアにマップすることができます。「for」を「_Cilk_for」に変更するだけでタスク並列処理を実装し、フレームの各行を複数のコアにわたって並列にレンダリングすることができます。

データ並列処理により得られる利点の検証

4. **[Solution Explorer (ソリューション エクスプローラ)]** で「serial.cpp」を開きます。ファイルの先頭に移動し (Ctrl + Home キー)、関数 ambient_occlusion() にあるアンビエント・オクルージョン・コードのシリアルバージョンを確認します。この関数は、イメージの 1 行を構成する個々のピクセルをレンダリングします。ここでは、アルゴリズムがどのように結果を計算してているかではなく、シリアルバージョンとデータ並列バージョンの違いに注目します。
5. 「parallel.cpp」に戻り、ファイルの先頭にある関数 ambient_occlusion_arr_notation() のコードを確認します (図 10)。これは、最初の関数のデータ並列バージョンです。

```
int    i, j;
int    ntheta = NAO_SAMPLES;
int    nphi   = NAO_SAMPLES;
float eps = 0.0001f;
vec p;
vec basis[3];
float occlusion = 0.0;

// rand1, rand2: buffer arrays to store random results
// theta, phi, x, y, z, rx, ry, rz: array versions of scalar loop variables from serial version
// v: array version of scalar from manually inlined ray_plane_intersect()
// Added the memory alignment explicitly
__declspec(align(64)) float rand1[NAO_SAMPLES],rand2[NAO_SAMPLES],theta[NAO_SAMPLES],phi[NAO_SAMPLES];
__declspec(align(64)) float x[NAO_SAMPLES], y[NAO_SAMPLES], z[NAO_SAMPLES],rx[NAO_SAMPLES], ry[NAO_SAMPLES], rz[NAO_SAMPLES],v[NAO_SAMPLES];
```

図 10

6. シリアルバージョンの単一の変数 (x、y、z、theta など) は、並列バージョンでは配列 (x[NAO_SAMPLES]、y[NAO_SAMPLES] など) に置換され、一度にすべての配列要素に対して処理を実行するようにコンパイラーに指示しています。つまり、値を 1 つずつ処理する代わりに、同時に複数の値を処理します。NAO_SAMPLES は一度に処理する値の数 (ベクトル長) です。シリアルコードをデータ並列コードに変換するため、インテル® Cilk™ Plus で次のことを行います。
   - スカラー変数を配列変数に拡張する
   - 部分配列を使って配列変数に対する処理をデータ並列処理に変換するコードを下にスクロールすると、データ処理を並列化するため多数の [:] 構文が使用されているのが分かります。例えば、次のような構文が使用されています。
   - x[:] = cos(phi[:]) * theta[:];
   - この構文は、「配列 *phi* のすべての要素の余弦と配列 *theta* の要素を掛けて、結果を配列 *x* に格納する」ことを意味します。
7. Ctrl + Alt + F7 キーを押して (または **[Build (ビルド)]** メニューから **[Rebuild Solution (ソリューションのリビルド)]** を選択して) プロジェクトをリビルドし、F5 キーで実際にこのデモを実行することができます。このコードは、提供されているオリジナルバージョンと全く同じ動作になります。

(C/C++ 開発者向け): 自分でコードを並列化してみましょう!

1. **[Solution Explorer (ソリューション エクスプローラ)]** で「lab.cpp」を開きます。このファイルには、わずかな変更で並列化できるコードが含まれています。
2. ファイルの先頭に移動し (Ctrl + Home キー)、is_lab_enabled 変数の宣言 (「bool is_lab_enabled = false;」行) で初期値を「true」に変更します。Ctrl + Alt + F7 キーを押してソリューションをリビルドし、F5 キーで実行します。
3. デモが変更され、最初にシリアルバージョンを実行した後、あなたのバージョンが実行されます。この時点では、どちらのバージョンも動作は同じです。
4. まず、タスクを並列化します。
5. Ctrl + End キーを押してファイルの最後に移動し、render_lab 関数を見つけます。コメントの指示に従って、「for」を「_Cilk_for」に変更します。
6. この変更により、for ループの各反復を並列に実行できることをコンパイラーに知らせます。インテル® Cilk™ Plus ランタイムは、利点が得られる場合、これらの反復を並列に実行します。
7. プロジェクトをリビルドして再度実行します。パフォーマンスが大幅に向上したことが分かります。
8. さらにパフォーマンスを向上させるため、データ処理を並列化します。
9. Ctrl + F キーを押して **[Find and Replace (検索と置換)]** ウィンドウを開き、START OF CODE を検索します。
10. START OF CODE は、データ並列処理を実装するコード領域の開始位置を示しています。

    ここで、0 から NAO_SAMPLES の範囲のインデックス変数 *k* を持つ for ループを削除し、要素ごとに行われている配列操作を配列全体に対するデータ並列処理に変更します。
11. for 文とループの最後にある対応する閉じ括弧 (「END OF CODE SECTION」というコメントがある行の 1 行上) をコメントアウトします。
12. Ctrl + F キーを押して **[Find and Replace (検索と置換)]** ウィンドウを開き、**[Quick Replace (クイック置換)]** をクリックします。図 11 のように、**[Find what (検索する文字列)]** に「[k]」を指定し、**[Replace with (検索後の文字列)]** に「[:]」を指定します。

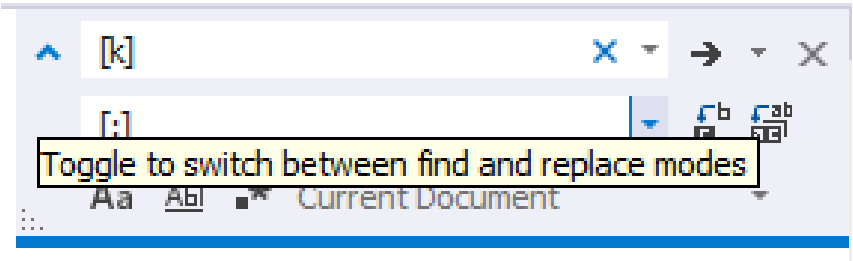

図 11

13. **[Replace All (すべて置換)]** ボタンをクリックします。これで、x[k]、y[k] などの単一要素の配列参照がすべて、x[:]、y[:] のような配列全体の参照に置換されます。
14. プロジェクトをリビルドして再度実行します。データ並列処理により、パフォーマンスがさらに向上したことが分かります。
15. これで、インテル® Cilk™ Plus によるさまざまな並列化のチュートリアルは終了です。

インテル® Cilk™ Plus の SIMD ベクトル化と SIMD 対応関数によるパフォーマンス

SIMD ベクトル化と SIMD 対応関数は、インテル® C++ コンパイラーによりサポートされるインテル® Cilk™ Plus 機能の一部で、ループとユーザー定義関数をベクトル化する方法を提供します。インテル® コンパイラーには、ベクトル化を有効にする独特な機能が用意されています。開発者は、単純なプログラミング・スタイルとベクトル化を支援するコンパイラー機能を利用して、より多くのループがベクトル化されるようにコンパイラーに指示できます。「インテル® Cilk™ Plus の SIMD ベクトル化と SIMD 対応関数」の記事では、インテル® Cilk™ Plus のベクトル SIMD 対応関数と SIMD 宣言子 (#pragma simd) を使って、コンパイラーに C/C++ コードのベクトル化を指示し、パフォーマンスを向上する方法を説明しています。

まとめ

インテル® Cilk™ Plus のキーワード、レデューサー・ハイパーオブジェクト、配列表記、SIMD 対応関数を使用することで、C/C++ アプリケーションを簡単に並列化して、並列コードの開発と保守に必要な労力を抑えながら、プロセッサーの SIMD ベクトル機能とマルチコアの両方を十分に活用することができます。

関連情報

構文とセマンティクスに関する詳細は、インテル® Cilk™ Plus ドキュメントを参照してください。

- インテル® Parallel Studio XE ドキュメント - 特に『インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド』の「インテル® Cilk™ Plus」セクションが役立ちます。
- インテル® Cilk™ Plus チュートリアル – インテル® Parallel Studio XE とともにインストールされます。また、上記のインテル® Parallel Studio XE ドキュメントとともにオンラインでも利用可能です。

インテル® Cilk™ Plus のオープン仕様も確認してみてください。ご意見・ご要望は、インテル® Cilk™ Plus ユーザーフォーラムからお寄せください。

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

インテル® C++ コンパイラー・コード・サンプル – インテル® Cilk™ Plus を利用する各種 C++ サンプル

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

インテル® ソフトウェア・ネットワーク・フォーラム – デベロッパー・コミュニティー

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 simple-path2parallelism-intel-cilk-plus_studioxe-evalguide/Rev-082014