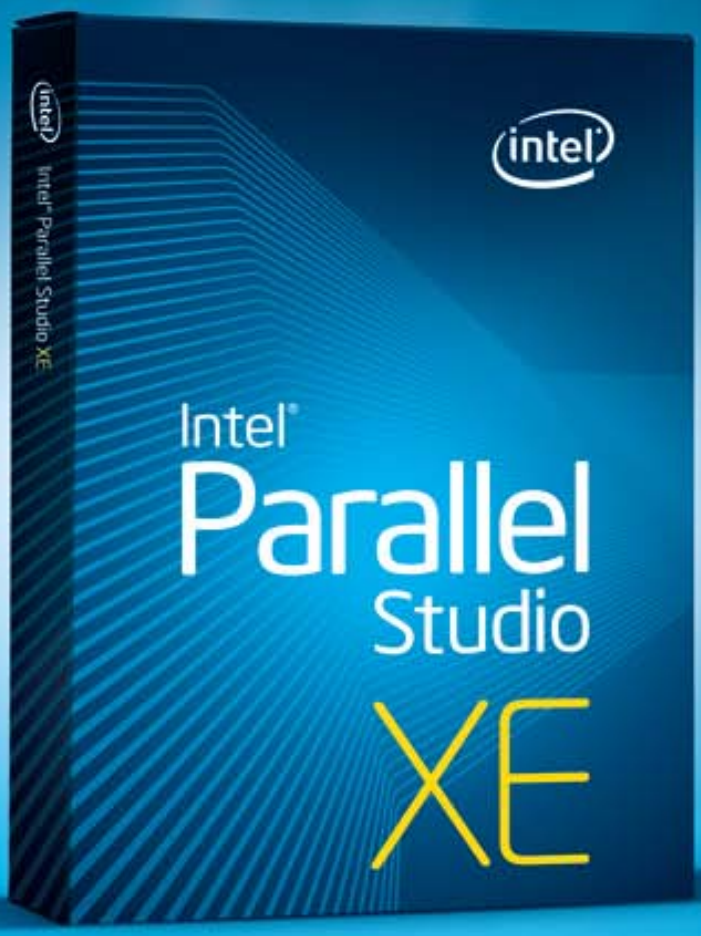
# インテル® Cilk™ Plus: 並列化への近道



# インテル® Cilk™ Plus: 並列化への近道

## 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 Composer、インテル® Parallel Studio、インテル® Composer XE、およびインテル® Parallel Studio XE の一部です。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:<br>*cilk_for*: for ループを並列化します。<br>*cilk_spawn*: 並列実行の開始を指定します。<br>*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。 |
| 要素関数 | 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 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、並列化の機会を簡潔に表現する
- 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 – 意図を表現
- 同じデータで並列操作とシリアル操作を組み合わせる

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。

# 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、インテル® Cilk™ Plus の要素関数を利用したブラックショールズ・アルゴリズムです。インテル® SSE を利用できるようスカラー関数がベクトル化され、cilk_for によりベクトル化されたコードが複数コアで並列に実行されます。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 を利用したデータ並列化 | インテル® Cilk™ Plus を利用したタスク並列化 |
|---|---|---|---|
| モンテカルロ・シミュレーション [1] | **5.8 秒** | **2.8 秒 = 1.9 倍のスピードアップ** | **0.50 秒 = 11.6 倍のスピードアップ (16 仮想コア)** |
| ブラックショールズ [2] | **2.1 秒** | **1.1 秒 = 1.9 倍のスピードアップ** | **0.14 秒 = 15 倍のスピードアップ** |

コンパイラー: インテル® Parallel Composer 2011 と Microsoft* Visual Studio* 2008。
[1] システムの仕様: インテル® Xeon® プロセッサー W5580、3.20GHz、8 コア、ハイパースレッディング有効、6GB RAM、Microsoft* Windows* XP Professional x64 Version 2003、Service Pack 2。
[2] システムの仕様: インテル® Xeon® プロセッサー E5462、2.80GHz、8 コア、8GB RAM、Microsoft* Windows Server* 2003 x64 Edition。

# 実践

このガイドでは、インテル® Parallel Studio 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
- *cilk_for* キーワードと要素関数を使用したブラックショールズ・アプリケーション
- 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

# インテル® Parallel Studio 2011 のインストール

### インテル® Parallel Studio のインストールと設定:

1. インテル® Parallel Studio 2011 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 (システムにより異なりますが、約 15-30 分かかります)。

**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® Composer XE Linux* 版のインテル® 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

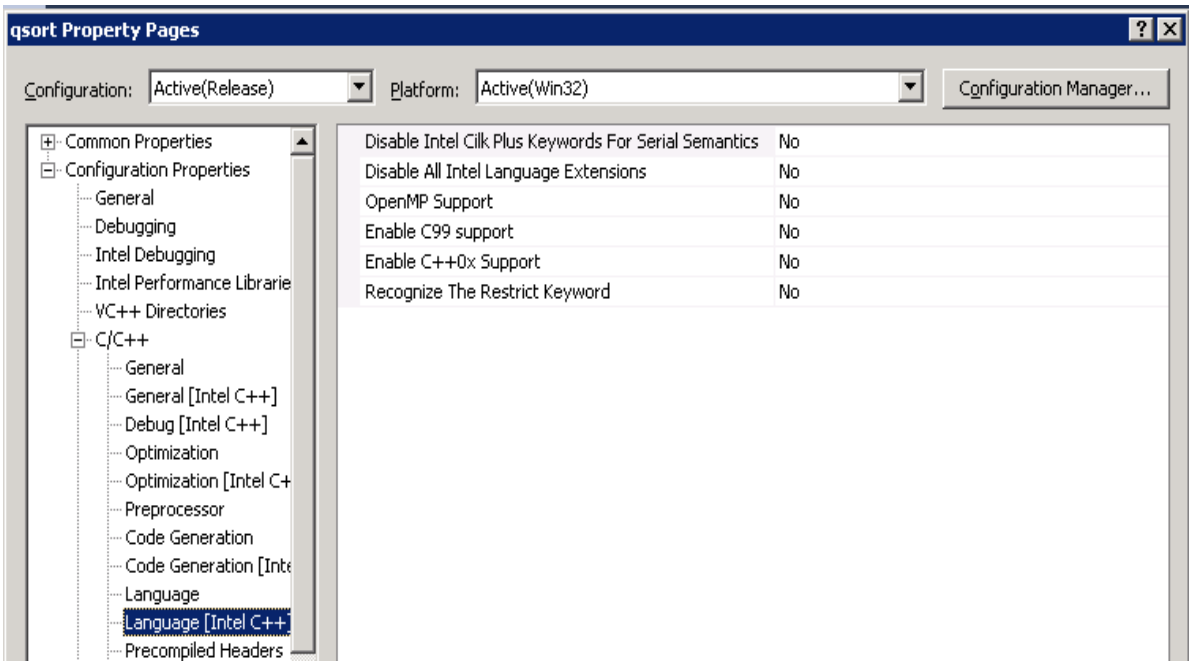
### サンプル・アプリケーションのインストール

1. BlackScholesSample.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、まずインテル® C++ コンパイラーで主要な関数をベクトル化する方法、および SIMD (Single instruction Multiple Data) の並列化バージョンを生成する方法を紹介します。その後、cilk_for を使用してベクトル化された関数を並列化します。
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を紹介します。
3. インテル® Parallel Composer 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\Intel\Parallel Studio 2011\Composer\Samples\en_US\C++\Cilk.zip にあります。
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\Visual Studio 200x\Intel\samples 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。
   - BlackScholesSample
   - MonteCarloSample

### サンプルのビルド:

各サンプルには、Visual Studio* 2005、2008、2010 で使用できる Microsoft* Visual Studio* 2005 ソリューション・ファイル (.sln) があります。

1) インテル® C++ コンパイラーの Release (最適化) 構成設定を使用して、インテル® Parallel Composer 2011 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) **[Disable Intel Cilk Plus Keywords for Serial Semantics (シリアル・セマンティクスでインテル(R) Cilk Plus キーワードを無効にする)]** 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。次に qsort の [構成プロパティ] を示します。




3) Microsoft* Visual Studio* からアプリケーションを実行します。**[Debug (デバッグ)] > [Start Without Debugging (デバッグなしで開始)]** を選択します。

# 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. qsort ソリューションを Microsoft* Visual Studio* にロードします。
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(int * begin, int * end)
{
   if (begin != end) {
      --end;  // partition から最後の要素 (pivot) を除外します。
      int * middle = std::partition(begin, end,
                  std::bind2nd(std::less<int>(), *end));
      using std::swap;
      swap(*end, *middle);   // pivot を中央に移動します。
      cilk_spawn sample_qsort(begin, middle);
      sample_qsort(++middle, ++end); // pivot を除外し、end を元に戻します。
      cilk_sync;
   }
}
```

*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、並列で実行されます。

現時点では、インテル® Cilk™ Plus のキーワードはインテル® C++ コンパイラーに固有です。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 (下記の関連情報を参照してください)。インテル® Cilk™ Plus キーワードの鍵となる特性は、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 (**[Disable Intel Cilk Plus Keywords for Serial Semantics (シリアル・セマンティクスでインテル(R) Cilk Plus キーワードを無効にする)] 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以下のコードを追加します。

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

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 12.0 ユーザー・リファレンス・ガイド』の「インテル® Cilk™ Plus の使用」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® Parallel Studio 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使用することで、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簡単に確実に保護できます。

## インテル® Cilk™ Plus の要素関数と *cilk_for* によるパフォーマンス

このサンプルでは、インテル® Cilk™ Plus の要素関数の実装方法とインテル® コンパイラーの機能を使用して関数呼び出しをベクトル化する方法を紹介します。

要素関数を記述する場合、コードでベクトルバージョンの生成が必要なことをコンパイラーに示す必要があります。次のように、*__declspec(vector)* 構文を使用します。

```
__declspec(vector) float saxpy_elemental(float a, float x, float y)
{
    return(a * x + y);
}

以下は、配列表記を使用して、ベクトル化された要素関数を呼び出す方法です。

z[:]= saxpy_elemental(a, b[:], c[:]);
```

コンパイラーは、算術ライブラリーで標準関数のベクトル実装を提供しています。特別な方法でそれらを宣言する必要はありません。コンパイラーは、下記のように、配列表記の呼び出しから使用すべきバージョンを判断し、ベクトルバージョンを呼び出します。

```
a[:]= sin(b[:]);              // ベクトル・マス・ライブラリー関数
a[:]= pow(b[:], c);           // ベクトル・マス・ライブラリー関数による b[:]**c
```

(インテルの算術ライブラリーのルーチンの多くは、互換マイクロプロセッサーと比べてインテル製マイクロプロセッサー向けにより高度に最適化されています。)

次に要素関数を使用したブラックショールズの例を見てみましょう。

1. CilkPlus-BlackScholesSample ソリューションを Microsoft* Visual Studio* にロードします。
2. ベクトル化されていない元のバージョンまたはシリアルバージョンのコードを含む BlackScholes-Serial.cpp ファイルを開きます。

test_option_price_call_and_put_black_scholes_Serial() ルーチンは、元のバージョンまたはシリアルバージョンのルーチンを使用してコールオプションとプットオプションの計算を行います。

```
double option_price_call_black_scholes(...) {...}
double option_price_put_black_scholes(...) {...}

void test_option_price_call_and_put_black_scholes_Serial(
                double S[NUM_OPTIONS], double K[NUM_OPTIONS],
                double r, double sigma,double time[NUM_OPTIONS],
                double call_Serial[NUM_OPTIONS], double put_Serial[NUM_OPTIONS])
{
  for (int i=0; i < NUM_OPTIONS; i++) {
    // コールオプションを計算する関数を呼び出します。
    call_Serial[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);

    // プットオプションを計算する関数を呼び出します。
    put_Serial[i] = option_price_put_black_scholes(S[i], K[i], r, sigma, time[i]);
  }
}
```

次に BlackScholes-ArrayNotationsOnly.cpp ファイルを開き、配列表記で要素関数を使用したインテル® Cilk™ Plus バージョンを見てみましょう。

```
__declspec(vector) double option_price_call_black_scholes(...) {...}
__declspec(vector) double option_price_put_black_scholes(...) {...}

void test_option_price_call_and_put_black_scholes_ArrayNotations(
                double S[NUM_OPTIONS], double K[NUM_OPTIONS],
                double r, double sigma,double time[NUM_OPTIONS],
                double call_ArrayNotations[NUM_OPTIONS], double put_ArrayNotations[NUM_OPTIONS])
{
  // 計算を実行するコアで利用可能な SIMD 向けに最適化されたベクトル化バージョンの call 関数を呼び出します。
  call_ArrayNotations[:]= option_price_call_black_scholes_ArrayNotations(S[:], K[:], r, sigma, time[:]);

  // 計算を実行するコアで利用可能な SIMD 向けに最適化されたベクトル化バージョンの put 関数を呼び出します。
  put_ArrayNotations[:]= option_price_put_black_scholes_ArrayNotations(S[:], K[:], r, sigma, time[:]);
}
```

このコードでは、コールオプションとプットオプションを計算するそれぞれの関数の宣言に *__declspec(vector)* を追加して、コンパイラーにこれらの関数のベクトル化バージョンを生成するように指示しています。また、for ループの代わりに配列表記構文を使用して対応する配列セクションを指定し、ベクトル化バージョンの関数を呼び出しています。call_ArrayNotations[:]、put_ArrayNotations[:]、およびその引数で使用されている配列表記構文は、演算をそれぞれの配列要素に適用するように指示しています。コンパイラーは、ベクトルバージョンの関数、つまり要素関数を呼び出し、ベクトル化したループに適したループ境界を自動で見つけ出します。

このコードでは、元のアルゴリズムはまったく変更されていません。この簡単な変更により、シングルコアのシステムで 2 倍近くのスピードアップが達成されています (システム構成は、3 ページの 2 番目の注釈を参照してください)。インテル® Cilk™ Plus の要素関数はデータ並列化を行う関数をコンパイラーに対して表現できるため、簡潔さに優れています。

ベクトル化は、インテル製マイクロプロセッサーと互換マイクロプロセッサーの両方において、デフォルトの最適化レベルで有効になりますが、ベクトル化で使用されるライブラリー・ルーチンは、インテル製マイクロプロセッサー上において、より高いパフォーマンスが得られる場合があります。また、ベクトル化は /arch や /Qx (Windows*)、-m や -x (Linux* および Mac OS* X) などの特定のコンパイラー・オプションにも影響されます。

3. ここでは、ベクトル化されたシーケンシャル・コードを並列化するためにスレッディングを導入し、マルチコア・システムを活用していきます。BlackScholes-CilkPlus.cpp ファイルを開いて、元のコードにある for ループを *cilk_for* に置換するだけでコードを並列化できます。

```
void test_option_price_call_and_put_black_scholes_CilkPlus(
                double S[NUM_OPTIONS], double K[NUM_OPTIONS],
                double r, double sigma,double time[NUM_OPTIONS],
                double call_CilkPlus[NUM_OPTIONS], double put_CilkPlus[NUM_OPTIONS])
{
  // 計算を実行するコアで利用可能な SIMD 向けに最適化されたベクトル化バージョンの call 関数と put 関数を呼び出し、
  // マルチコア・パフォーマンスを最大限に引き出すよう cilk_for を使用して複数のコアに分散します。
  cilk_for (int i=0; i<NUM_OPTIONS; i++) {
    call_CilkPlus[i] = option_price_call_black_scholes_ArrayNotations(S[i], K[i], r, sigma, time[i]);
    put_CilkPlus[i] = option_price_put_black_scholes_ArrayNotations(S[i], K[i], r, sigma, time[i]);
  }
}
```

この簡単で強力な変更により、ループが自動で並列化され、前のステップで最適化されたシーケンシャル・コードを複数のコアで実行できるようになります。インテル® Cilk™ Plus のランタイムにより、システムで利用可能なコア数が特定され、それに応じてロードが分散されます。3 ページの表にあるように、この例では 8 コアのシステムで 8 倍近くのスピードアップが達成されています。

# パフォーマンスと並列化

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:

**[<下限> :<長さ> :<ストライド>]**

<*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。配列操作と代入の例を次に示します。

```
操作:
a[:]* b[:]// 要素単位の乗算
a[3:2][3:2] + b[5:2][5:2] // a[3][3]、b[5][5] から始まる a と b の 2x2 部分配列の行列の加算
a[0:4][1:2] + b[0][1] // スカラー b[0][1] を a の配列セクションの各要素に加算

代入:
a[:][:]= b[:][2][:]+ c;
e[:]= d; // スカラー変数 d が配列 e のすべての要素に代入される
```

MonteCarloSample アプリケーションを見てみましょう。

1. MonteCarloSample ソリューションを Microsoft* Visual Studio* にロードします。
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。

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;
}
```

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[:];
}
```

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 コードを使用して、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。Pathcalc_Portfolio_Scalar() 関数を表示します。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;
}
```

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、ハイパースレッディング・テクノロジー対応の 8 コアのシステム (詳細は 3 ページの表を参照) において、スカラーバージョンと比べて 2 倍近くパフォーマンスが向上しました。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 8 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。

## 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、要素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

インテル® Cilk™ Plus のユーザーフォーラムを参照してください。

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 ドキュメント - 特に『インテル® C++ コンパイラー 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」の「インテル® Cilk™ Plus の使用」が役立ちます。
- インテル® Cilk™ Plus チュートリアル – インテル® Parallel Composer 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 ドキュメントとともにオンラインでも利用可能です。

最後に、インテル® Cilk™ Plus のオープン仕様も確認してみてください。メールまたは上記のインテル® Cilk™ Plus ユーザーフォーラムから、ご意見をお聞かせください。

# インテル® Cilk™ Plus: 並列化への近道

| 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® スレッディング・ビルディング・ブロック Web サイト | |
| 並列化ブログ、論文、ビデオ | |