

# インテル® Parallel Amplifier 入門ガイド

本ガイドでは、インテル® Parallel Amplifier を使用し、マルチコアシステム上でサンプル・アプリケーションのパフォーマンスを分析します。アプリケーションの実行中にすべてのコアが十分に利用されているかどうかを確認し、スレッド化によって恩恵が得られるコードセクションを特定する方法を説明します。

本ガイドでは、matrix アプリケーションをサンプルとして使用し、次の作業を行います。

- シリアル・アプリケーションのパフォーマンスの問題を特定する
- アプリケーションをスレッド化する
- スレッド化アプリケーションのパフォーマンスを最適化する
- スレッド化アプリケーションのパフォーマンスを比較する

---

注: インテル® Parallel Amplifier のビデオデモ ([Show Me](#)) をご利用いただけます。ビデオデモ (Show Me) を表示するには、Adobe® Flash® Player が必要です。

---

## 目次

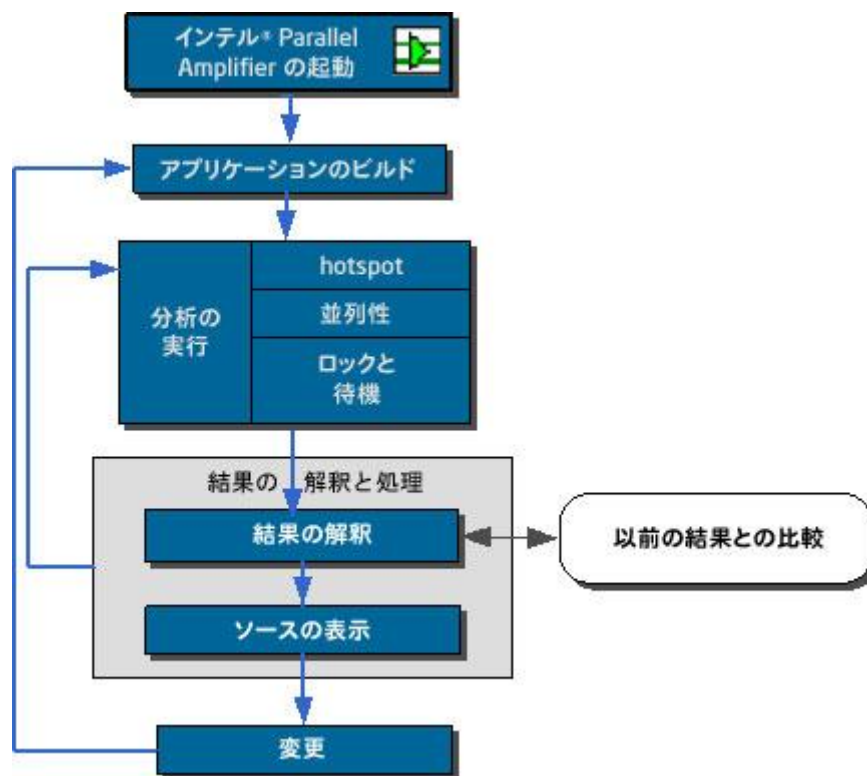
1	概要.....	2
2	アプリケーションのビルド.....	3
3	プログラム中で最も時間を消費している場所は?.....	4
4	並列性の低い場所は?.....	8
5	プログラムが待機している場所は?.....	11
6	得られた最適化の程度は?.....	14
7	ユーザー・リファレンス・ドキュメント.....	16
	著作権と商標について.....	17

# 1 概要

## 基本ワークフロー

本ガイドでは、基本的なワークフローに沿ってアプリケーションの hotspot を特定、分析する方法を紹介します。

また、このワークフローは、インテル® Parallel Amplifier ヘルプの基本編成を示します。



## ツール

本ガイドでは、以下のツールを使用します。

- Microsoft® Visual Studio® 2005 または 2008 開発環境
- インテル® Parallel Amplifier
- インテル® Parallel Amplifier に付属のサンプル・アプリケーション



---

**注:** すべてのコードサンプルは、インテル® Parallel Amplifier の機能を示すためにだけ設計されています。コードサンプルはマルチスレッド・コードを作成するための最良の事例を表しているわけではありません。分析の性質により、結果は異なることがあります。

---

## 準備

1. `<install_dir>\samples\<locale>` の `matrix.zip` ファイルを `C:\My Documents\Amplifier\Samples` などにコピーします。  
これにより、サンプル・アプリケーションが以前に実行されていても、元の状態にソースコードを戻すことができます。

---

**注:** デフォルトでは、`<install_dir>` は `C:\Program Files\Intel\Parallel Studio\Amplifier` です。インテル® 64 アーキテクチャー・システムの場合は、`C:\Program Files (x86)\Intel\Parallel Studio\Amplifier` です。

---


2. サンプルファイルを `.zip` ファイルから抽出します。

## 2 アプリケーションのビルド

---

最初に、Microsoft Visual Studio 環境でサンプル・アプリケーションをビルドする必要があります。本ガイドで使用する行列アプリケーションは、行列変換を計算します。分析を容易にし、迅速な最適化作業の見積りに役立つよう、アプリケーションにはタイマーが用意され、行列変換の計算に要した時間を出力できます。

行列アプリケーションをビルドするには、次の操作を行います。

1. Visual Studio から、**[ファイル]** > **[開く]** > **[プロジェクト/ソリューション]** をクリックし、`matrix.zip` ファイルが解凍された場所から `matrix.sln` ファイルを選択します。  
プロジェクトが Visual Studio に追加され、**[ソリューション エクスプローラ]** が表示されます。
2. **[ビルド]** > **[構成マネージャ]** をクリックして、ダイアログボックスを開き、**[Release]** モードを選択します。
3. システム・ライブラリーのデバッグ情報をダウンロードするようにします。
  - a. **[ツール]** > **[オプション...]** をクリックします。**[オプション]** ダイアログボックスが開きます。
  - b. 左ペインから、**[デバッグ]** > **[シンボル]** を選択します。
  - c. **[シンボル ファイル (.pdb) の場所]** フィールドで、 ボタンをクリックし、  
`http://msdl.microsoft.com/download/symbols/` を指定します。
  - d. 追加したアドレスのチェックボックスがオンになっていることを確認します。



- e. [シンボル サーバーからシンボルをキャッシュするディレクトリ] フィールドで、ダウンロードされたシンボルファイルを格納する場所を指定します。
  - f. Microsoft Visual Studio 2005 の場合は、[このダイアログを閉じるときに更新された設定を使用してシンボルを読み込む] チェックボックスをオンにします。
  - g. [OK] をクリックします。
4. バイナリーファイルのデバッグ情報を生成するようにします。
    - a. [プロジェクト] > [プロパティ] をクリックします。
    - b. [matrix プロパティ ページ] ダイアログボックスで、[構成プロパティ] > [全般] をクリックし、[構成] (ダイアログの上部) で [アクティブ (リリース)] が選択されていることを確認します。
    - c. [matrix プロパティ ページ] ダイアログボックスで、[C/C++] > [全般] > [デバッグ情報の形式] > [プログラム データベース (/ZI)] を選択します。
    - d. [matrix プロパティ ページ] ダイアログボックスで、[リンカ] > [デバッグ] > [デバッグ情報の生成] > [はい (/DEBUG)] を選択します。
  5. [ビルド] > [ソリューションのビルド] をクリックします。  
matrix.exe アプリケーションがビルドされます。

本ガイドの行列アプリケーションは、パフォーマンス解析に推奨される最適化をすべて使用してリリースモードでビルドされます。デバッグシンボルが生成され、インテル® Parallel Amplifier がソースコードを開くことができるようになります。

---

注: 詳細は、Microsoft Visual Studio の [ヘルプ] メニューにあるインテル® Parallel Amplifier ヘルプを参照してください。

---

## 3 プログラム中で最も時間を消費している場所は?

---

アプリケーションをビルドした後、パフォーマンスを分析することができます。インテル® Parallel Amplifier は、さまざまなタイプのパフォーマンス・データを収集し、数種類の分析を提供します。このステップでは、hotspot 分析を実行してデータを収集し、結果を表示して、特定のソースコードにある問題箇所をクローズアップします。hotspot 分析は、アプリケーション中で時間を費やしている場所を発見し、最も時間が消費されている関数を特定し、それらの関数がどのように呼び出されたかを示します。ボトルネックを排除できる hotspot もあれば、必然的でその性質のために実行に長時間かかる hotspot もあります。

hotspot 分析は、シリアル・アプリケーションと並列アプリケーションの両方のパフォーマンスを分析するのに役立ちます。

---

注: アプリケーションのパフォーマンスを再現可能にするため、その他のソフトウェアの実行を最小限に抑えた同一システムですべてのチューニング処理を行ってください。

---

## パフォーマンス・ベースラインの確立

パフォーマンスの指標となるベースラインを作成します。

1. Visual Studio メニューから、**[デバッグ] > [デバッグなしで開始]** をクリックします。  
Visual Studio で行列プロジェクトをコピーしたディレクトリーから行列アプリケーションが実行されます。

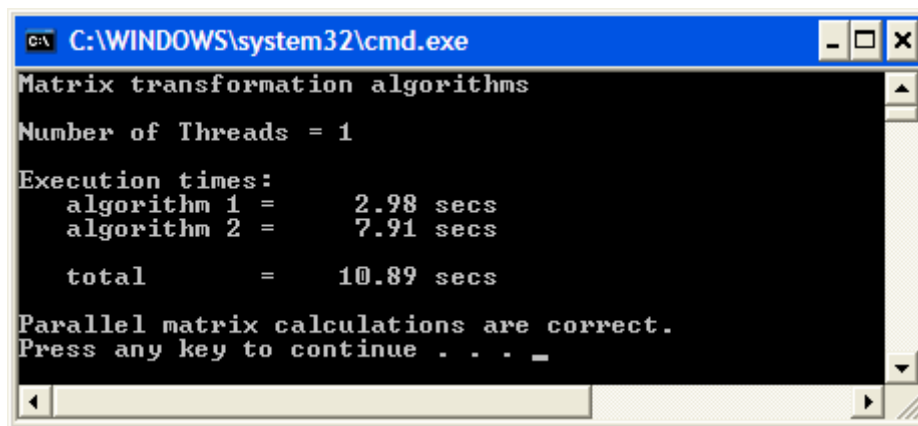
---

**注:** アプリケーションを開始する前に、より正確な結果が得られるよう、システムで実行されているソフトウェアの数を最小限にしてください。

---

ベースラインは、今後のリビジョンの比較の目安となるように、測定可能で再現可能でなければなりません。

2. アプリケーションの実行後、出力結果の実行時間に注目してください。



```
C:\WINDOWS\system32\cmd.exe
Matrix transformation algorithms
Number of Threads = 1
Execution times:
algorithm 1 = 2.98 secs
algorithm 2 = 7.91 secs
total = 10.89 secs
Parallel matrix calculations are correct.
Press any key to continue . . . _
```

この合計実行時間が、今後のアプリケーション実行の比較で使用するベースラインになります。

---

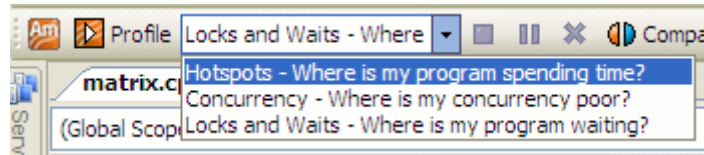
**注:** アプリケーションを数回実行し、各実行時間をメモして、平均値を算出します。これにより、一時的なシステム・アクティビティーによる不均衡を最小限に抑えることができます。

---

## hotspot の発見

hotspot 分析を実行して、*hotspot* (実行に長時間を要する関数/コードセクション) を特定します。

1. Amplifier ツールバーから、**[Hotspots - Where is my program spending time running? (hotspot - プログラム中で最も時間を消費している場所は?)]** を選択します。



2. **[Profile (プロファイル)]** ボタンをクリックします。  
行列変換を計算して終了する行列アプリケーションが起動します。

データ収集が完了すると、**[Bottom-up (ボトムアップ)]** ウィンドウが開き、関数が下から上 (子関数とその親関数の上に直接配置される) の順で表示されます。

注: 本ガイドで使用されているスクリーンショットと実行時間データは、2 CPU コアのシステムで収集されたものです。実際のデータは、使用するシステムの CPU コア数と種類により異なります。

Function	CPU Time:Self	Module
algorithm_2	7.765s	matrix.exe
do_xform ← BaseThreadStart	7.765s	matrix.exe
algorithm_1	2.917s	matrix.exe
main	0.021s	matrix.exe
Selected:	7.765s	

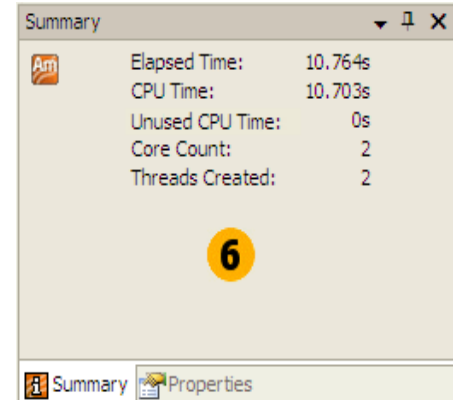
Call Stack (CPU time (User))

```

1 selected stack(s), Viewing 1 of 1
★ Stack viewed is 100.0% of selection
100.0% (7.765s of 7.765s)
-----
matrix.exe!algorithm_2(int) - matrix.cpp
matrix.exe!do_xform(void *) - matrix.cpp:142
kernel32.dll!BaseThreadStart+0x36
  
```



<b>1</b>	hotspot 分析からの結果。結果が収集されるたびに番号 (000) が上がります。
<b>2</b>	<b>[Function - Caller Function Tree (関数 - 呼び出し関数ツリー)]</b> は、hotspot データのデフォルトのグループレベルです。矢印ボタンをクリックして、グループレベルを変更します。
<b>3</b>	関数名の前にあるプラス記号をクリックすると、選択された関数のコールスタックを表示できます。選択された関数の呼び出し元が表示されると、次に最初の呼び出し元の呼び出し元、のように順に表示されます。
<b>4</b>	CPU 時間は、論理プロセッサで関数を実行するのにかかる時間です。複数のスレッドの場合は CPU 時間が合計されます。これは、hotspot 分析結果の <b>[Data of Interest (特定のデータ)]</b> 列です。
<b>5</b>	選択された関数のスタック情報全体がグリッドに表示されます。黄色のバーは、hotspot 関数の CPU 時間に対する選択されたスタックの割合を示しています。
<b>6</b>	分析実行のサマリーデータ: 1) <b>[Elapsed Time (経過時間)]</b> は、アプリケーションの開始から終了までの実行時間です。 2) <b>[CPU Time (CPU 時間)]</b> は、すべてのスレッドの CPU 時間です。 3) <b>[Unused CPU Time (未使用の CPU 時間)]</b> は、待機中またはアプリケーションで有効利用されていない各コアの合計時間です。 4) <b>[Core Count (コア数)]</b> は、マシンの論理 CPU 数です。 5) <b>[Threads Created (作成されたスレッド数)]</b> は、アプリケーション実行中にシステムにより作成されたスレッドの数です。



**注:** マウスを移動するか、または F1 を押して、ユーザー・インターフェイスの状況依存ヘルプを表示できます。

## 結果の分析

**[Bottom-up (ボトムアップ)]** ウィンドウにリストされている最初の関数で、最も時間が消費されているのは、`algorithm_2` です。この関数に焦点をあてて、パフォーマンスを向上させる方法を探します。

`algorithm_2` 関数をダブルクリックして、ソースコードを表示します。行番号 182 で CPU 時間が最も消費されていることに注目してください。

Line	Source	CPU Time:Self
173	// Enter Critical Section to protect inner loops from multithreaded access (Needed?)	
174	// EnterCriticalSection(&initialization_section);	
175	for (int j = 0; j < N; ++j) {	
176	int ij = i*N + j;	
177		
178	c2[ij] = 0.0;	0.011s
179	for (int k = 0; k < N; ++k) {	
180	int ik = i*N + k;	
181	int kj = k*N + j;	
182	c2[ij] += a[ik]*b[kj];	7.754s
183	}	
184	}	
185		
186	// Exit from Critical Section	
187	// LeaveCriticalSection(&initialization_section);	
188	}	
Selected:		7.754s

下の表は、hotspot 分析データを表示した際に [Source (ソース)] ペインで利用できるいくつかの機能を説明しています。

1	編集不可でアプリケーションのソースコードが表示されます。関数のシンボル情報が利用できる場合に開きます。CPU 時間を最も消費しているコード行がハイライトされます。
2	プロセッサ時間は特定のコード行に起因します。hotspot がシステム関数の場合は、デフォルトでは、このシステム関数を呼び出したユーザー関数に起因します。
3	hotspot ナビゲーション・ボタンを使用して実行に時間のかかるコード行を切り替えます。
4	Visual Studio エディターでコードを開いて編集することのできるソース・ファイル・エディター・ボタンです。

シリアルコード中の hotspot を特定したら、コードを変更し、hotspot の実行速度を向上させることは可能です。しかし、このドキュメントでは、並列最適化手法に焦点を置いています。このサンプルコードを並列化するには、スレッドをアプリケーションに追加し、マルチコア・プロセッサで効率良く実行できるようにします。そのため、複数のスレッドに分割するのに一番良い場所をアプリケーションで特定しなければなりません。

## 4 並列性の低い場所は? [Show Me](#)

このステップでは、並列処理分析を実行して、利用可能なすべてのコアがアプリケーションで効率的に利用されているか、また並列化に最も適したシリアルコードはどこかを特定します。

理想的なプロセッサ利用率とは、実行スレッド数が利用可能なコア数と等しいときに達成されます。プロセッサ利用率の低い hotspot に重点を置き、すべてのプロセッサ・コアを利用できるようにします。コアが停止状態の hotspot がある場合は、並列化を追加して、競合の不均衡を是正または減少することを検討してください。

## 並列性の検証

並列性分析を行うには、Amplifier ツールバーから **[Concurrency - Where is my concurrency poor? (並列性 - 並列性の低い場所は?)]** を選択して、**[Profile (プロファイル)]** をクリックします。行列アプリケーションが起動し、Amplifier でデータ収集が開始されます。出力ペインに表示される、PDF ファイルが見つからないという旨の警告は無視します。

行列アプリケーションが終了すると、Amplifier は結果を処理し、**[Concurrency (並列性)]** ウィンドウを開いて関数がボトムアップ順に配置されたコールスタックを表示します。

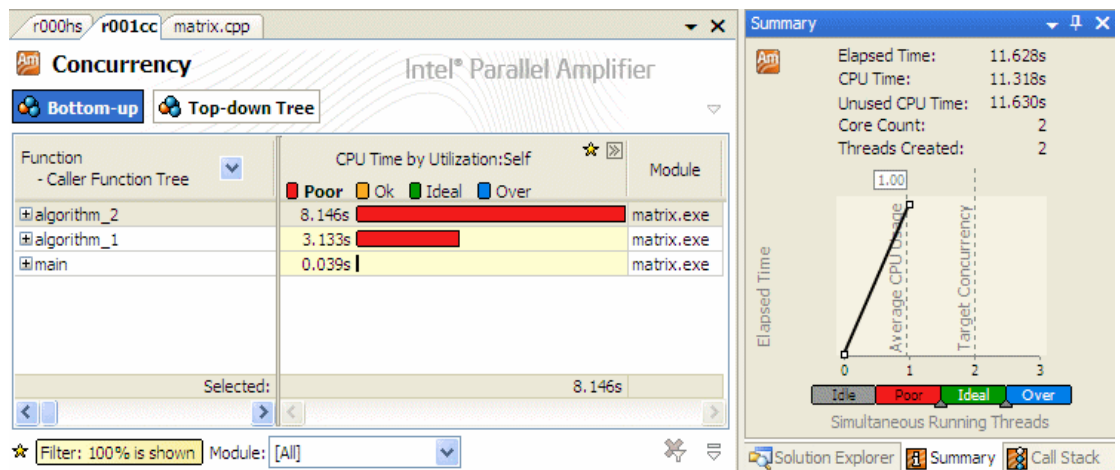
### CPU 利用率

**[Poor (低)]** - CPU コアは十分に利用されていません。

**[OK (中)]** - CPU コアは許容可能範囲で利用されています。

**[Ideal (高)]** - CPU コアは十分に利用されています。

**[Over (超過)]** - CPU コアはコア数を超えています。



**[Concurrency (並列性)]** ウィンドウと **[Summary (サマリー)]** タブの両方で、行列アプリケーション全体がシリアルであることが示されています。**[Time by Utilization (利用率別時間)]** 列の CPU 時間の赤いバーは、利用率の低いプロセッサを表します。**[Summary (サマリー)]** タブは、0 または 1 つの実行スレッドの CPU 時間のみを示します。

hotspot 分析では、algorithm\_2 が CPU 時間を最も消費する関数でした。並列性分析でも、すべての CPU コアが使用されていないために、この関数がシリアルコード中の hotspot としてハイライトされています。**[Concurrency (並列性)]** の **[Bottom-up (ボトムアップ)]** ウィンドウでは、algorithm\_2 関数に最も長い非並列時間が含まれていることが示されています。このモジュールは、最適な並列化候補である可能性があります。algorithm\_2 をダブルクリックして、ソースコードを表示し、最も長いシリアル時間の行を特定します。

## アプリケーションのリビルド

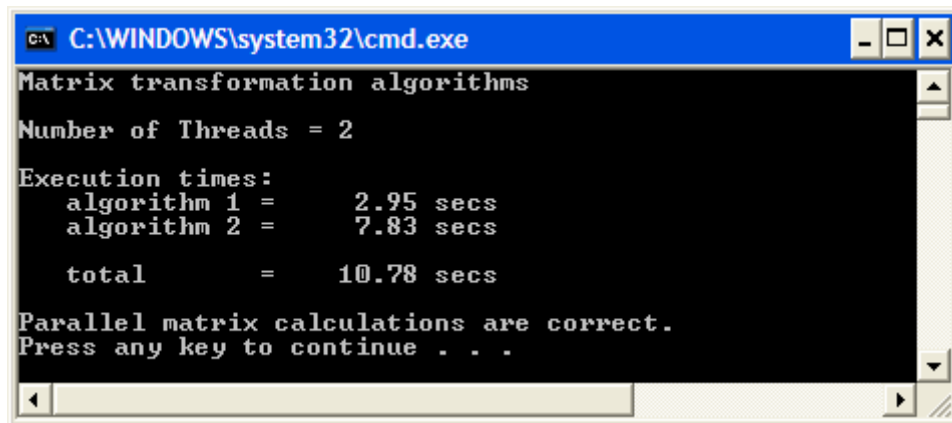
このステップでは、行列アプリケーションを修正し、リビルドして並列化します。

1. Visual Studio から、matrix.cpp を開きます。
2. 24 行目で、USE\_MULTIPLE\_THREADS が TRUE と定義されているマクロをアンコメントします。
3. 25 行目で、USE\_MULTIPLE\_THREADS が FALSE と設定されているマクロをコメントアウトします。

4. algorithm\_2 プロシーチャーの 174 行目と 187 行目の EnterCriticalSection コールと LeaveCriticalSection コールのコードをアンコメントします。これは、(意図的に) ロック・パフォーマンスの問題を引き起こします。
5. ソリューションをリビルドします。  
Visual Studio の出力ペインでエラーが 0、警告が 0 であることを確認してください。

## パフォーマンスの比較

新しくビルドされたアプリケーションをコマンドウィンドウで再度実行します。



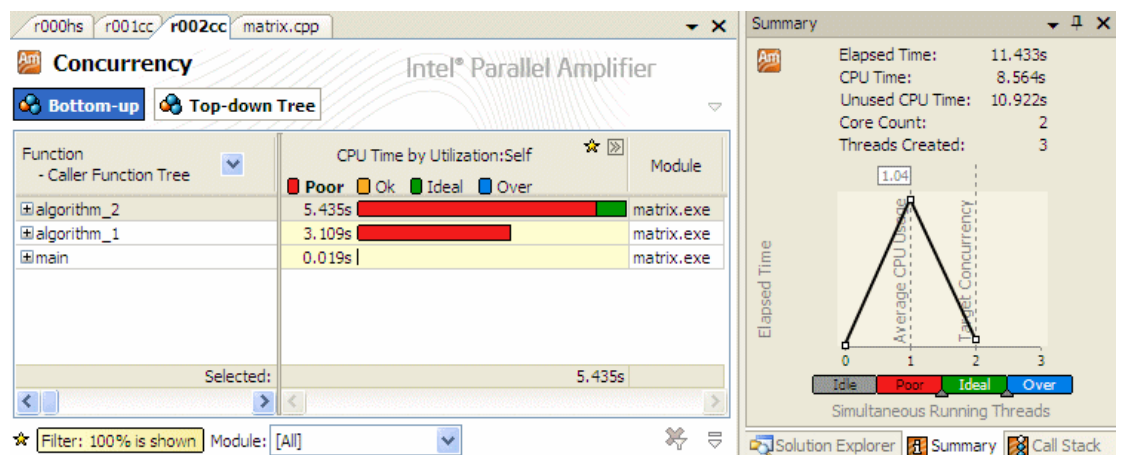
```

C:\WINDOWS\system32\cmd.exe
Matrix transformation algorithms
Number of Threads = 2
Execution times:
  algorithm 1 = 2.95 secs
  algorithm 2 = 7.83 secs
  total       = 10.78 secs
Parallel matrix calculations are correct.
Press any key to continue . . .
  
```

合計実行時間が 10.89 秒から 10.78 秒に短縮されたことに注目してください。

## 並列性の再検証

このステップでは、修正された行列アプリケーションで再度、並列性データの収集を行います。



ここで、algorithm\_2 について、CPU 低利用率インジゲーター (赤いバー) が減少していることに注目してください。利用可能な 2 CPU のうち 1.04 (または 52.2%) が利用されています。



しかし、赤いバーは最適化の余地のあるシリアル時間がまだ含まれているコードがあることを示しています。

## 5 プログラムが待機している場所は? Show Me

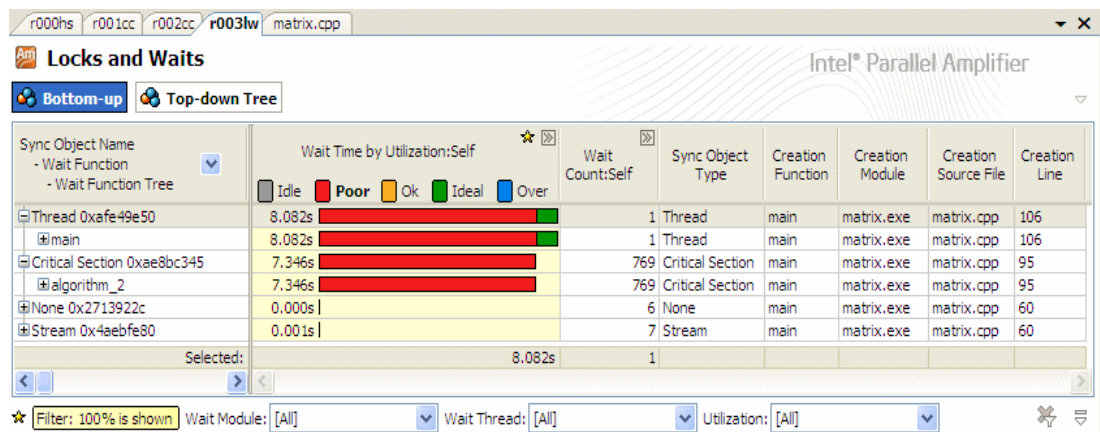
このステップでは、ロックと待機の分析を実行して、algorithm\_2 の CPU シリアル時間の原因を理解します。

この分析では、非効率的な並列アプリケーション (プロセッサ・コアが十分利用されずにスレッドが同期オブジェクト (ロック) で長時間待機している) における最も一般的な理由の 1 つを特定します。システムが十分に活用されずに長時間の待機が発生しているオブジェクトを重点的にチューニングします。

### ロックと待機の分析

ロックと待機の分析を実行するには、Amplifier ツールバーから、**[Locks and Waits - Where is my application waiting? (ロックと待機 - アプリケーションの待機場所)]** を選択して、**[Profile (プロファイル)]** をクリックします。

**[Locks and Waits (ロックと待機)]** の **[Bottom-up (ボトムアップ)]** ウィンドウが開き、アプリケーション中の同期オブジェクトで費やされている CPU 時間が表示されます。



最も待機時間の長い同期オブジェクトがスレッドであることに注意してください。このスレッドをダブルクリックして、待機ソースコードに移動します。

Line	Source	Wait Time by Utilization:Self					Wait Count:Self
		Idle	Poor	Ok	Ideal	Over	
111	}						
112	}						
113							
114	// Wait for all algorithm_2 threads to finish						
115	int done = WaitForMultipleObjects(NumThreads, h, TRUE, INFINITE);	8.082s					1
116							
117	time2 = GetSeconds() - time2;						
118							
119	printf("Execution times:\n");	0.000s					2
120	printf(" algorithm 1 = %.2f secs\n", time1);	0.000s					2
121	printf(" algorithm 2 = %.2f secs\n", time2);	0.000s					2
122	printf("\n total = %.2f secs\n", time1+time2);	0.000s					2
123							
124	if(checkResults())						
125	printf("\nParallel matrix calculations are correct.\n");						
126	else						
127	printf("\n*** Parallel matrix calculations have failed! ***\n");	0.000s					2
128							
Selected:		8.082s					1

このスレッドが行列変換スレッドの終了を待機しているメインスレッドであることがわかります。行列変換スレッドが計算中で、メインスレッドは計算が終了するのを待機しているだけなので、これは問題ではありません。

**[Bottom-up (ボトムアップ)]** ウィンドウの2番目の項目を見てみます。シリアルだけの時間が表示され、待機が発生しているクリティカル・セクションです。このクリティカル・セクションをダブルクリックして、待機関数のソースコードを表示します。

Line	Source	Wait Time by Utilization:Self					Wait Count:Self
		Idle	Poor	Ok	Ideal	Over	
166							
167	// Matrix transformation #2						
168	// This method is an example of a function that has some serialization of threads and						
169	// synchronization overhead.						
170	void algorithm_2(int myid) {						
171	for (int i = myid; i < N2; i += NumThreads){						
172							
173	// Enter Critical Section to protect inner loops from multithreaded access (Needed?)						
174	EnterCriticalSection(&initialization_section);	7.346s					769
175	for (int j = 0; j < N; ++j) {						
176	int ij = i*N + j;						
177							
178	c2[ij] = 0.0;						
179	for (int k = 0; k < N; ++k) {						
180	int ik = i*N + k;						
181	int kj = k*N + j;						
182	c2[ij] += a[ik]*b[kj];						
183	}						
Selected:		7.346s					769

これは、algorithm\_2 をスレッド化したときに作成されたクリティカル・セクションです。このクリティカル・セクションで、深刻な待機時間が発生しています。コードをより注意深く検証すると、クリティカル・セクションが必要ではないことがわかります。c2 配列の要素に書き込まれる代入文は、i がセットされるその上の for ループによって、複数のスレッドからアクセスされないように既に保護されています。for ループの帰納変数 i は、for ループの初期値 myid と反復子 i+=NumThreads により、各スレッドに異なってセットされます。そのため、クリティカル・セクション参照を削除して、アプリケーションを再実行します。



## 最終アプリケーションのリビルド

行列アプリケーションを再度 Visual Studio から開き、EnterCriticalSection コールと LeaveCriticalSection コールの行番号 174 と 187 をコメントアウトして、アプリケーションをリビルドします。

## 最終アプリケーションの実行

新しくビルドされた matrix.exe をコマンドウィンドウで再び実行します。

```
C:\WINDOWS\system32\cmd.exe
Matrix transformation algorithms
Number of Threads = 2
Execution times:
algorithm 1 = 3.00 secs
algorithm 2 = 4.14 secs
total = 7.14 secs
Parallel matrix calculations are correct.
Press any key to continue . . . _
```

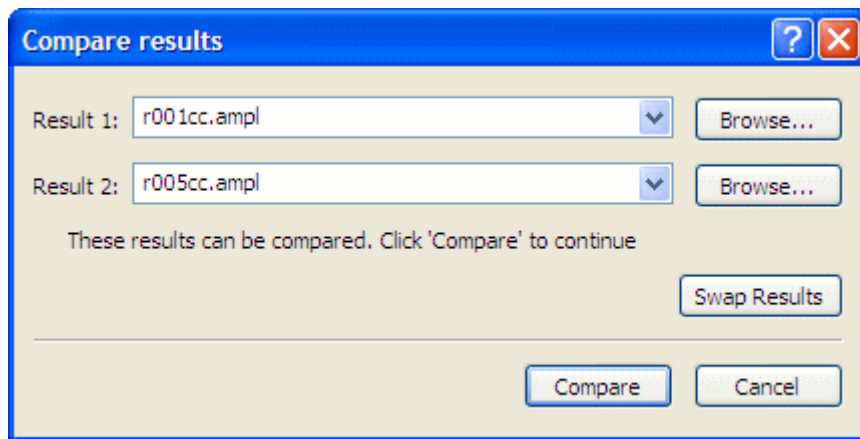
アプリケーションの合計実行時間が 10.78 秒から 7.14 秒に短縮されています。

## 6 得られた最適化の程度は? Show Me

このステップでは、並列分析結果を比較します。パフォーマンスの遷移を関数別に表示することができます。最適化の前後の結果を比較することで、行った変更がどのようにアプリケーション・パフォーマンスに効果をもたらしたかを推定できます。

並列性の結果を比較するには、次の操作を行います。

1. ロックと待機分析の後で、変更したコードの並列性分析を行います。
2. Amplifier ツールバーの **Compare** ボタンをクリックします。  
**[Compare Results (結果の比較)]** ダイアログボックスが開きます。
3. 比較する並列性結果を指定します。

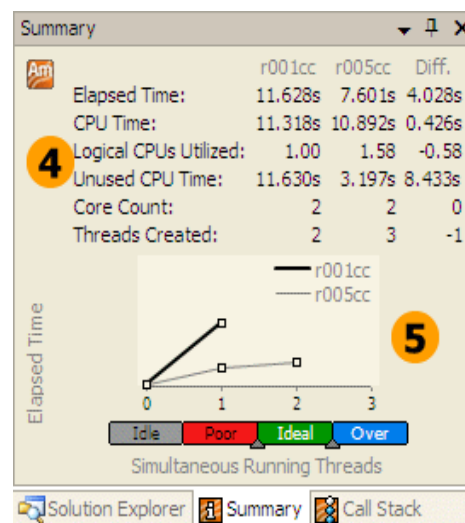


**[Concurrency (並列性)]** の **[Bottom-up (ボトムアップ)]** ウィンドウが開き、2つの結果のCPU利用率が横並びで表示されます。

Function	Utilization:Self: r001cc				Utilization:Self: r005cc				Utilization:Self: Difference				Module
	Poor	Ok	Ideal	Over	Poor	Ok	Ideal	Over	Poor	Ok	Ideal	Over	
algorithm_2													matrix.exe
algorithm_1													matrix.exe
main													matrix.exe
Selected:	8.146s				7.984s				8.036s 0s -7.874s 0s				



<b>1</b>	プロセッサが有効に利用されていないシングルスレッドの matrix.exe アプリケーションの CPU 時間。
<b>2</b>	プロセッサが理想的に有効利用されている最適化されたマルチスレッド matrix.exe の CPU 時間。
<b>3</b>	2つの結果の利用率レベルごとの CPU 時間の差異が、 $\langle \text{CPU 時間の差異} \rangle = \langle \text{結果 1 CPU 時間} \rangle - \langle \text{結果 2 CPU 時間} \rangle$ 形式で示されます。
<b>4</b>	2つの結果のサマリーデータの比較とその差異は、 $r001cc - r005cc = \text{Diff}$ で算出されます。  [Logical CPUs Utilized (利用されている論理 CPU)] は、アプリケーション実行中のすべてのコアの平均利用率です。
<b>5</b>	比較結果の並列グラフ



マルチスレッド・バージョンの matrix.exe アプリケーション (r005cc) では、algorithm\_2 の hotspot 関数に対して2つのスレッドを実行させることによりすべての利用可能なコアが効率良く稼働し、アプリケーションの経過時間で4秒の速度向上が得られたことがこの並列性比較結果でわかります。

定期的に分析結果を比較し、リグレッションに注意して、コードの段階的な変更によるパフォーマンスの変化をチェックしてください。

**注:** 次のステップとして、『[インテル® Parallel Amplifier サンプル・コード・ガイド](#)』(英語)を参照してください。このガイドでは、追加のコードサンプルを使用して、OpenMP\* やインテル® スレッディング・ビルディング・ブロック (インテル® TBB) を使用したコードのパフォーマンス分析結果についての解釈方法が説明されています。

## 7 ユーザー・リファレンス・ドキュメント

本ガイドは、インテル® Parallel Amplifier の基本機能を紹介しています。より詳細な機能については、次の資料を参照してください。

資料	説明
インテル® Parallel Amplifier ドキュメント	<p>この HTML ページから、その他のインテル® Parallel Amplifier のドキュメントへリンクされています。Windows* の [スタート] メニューから、<b>[Intel Parallel Studio (インテル(R) Parallel Studio)] &gt; [Parallel Studio Documentation (インテル(R) Parallel Studio ドキュメント)] &gt; [Amplifier Documentation (インテル(R) Parallel Amplifier ドキュメント)]</b> をクリックして表示します。</p>
サンプルコード	<p>&lt;install_dir&gt;\samples\&lt;locale&gt;\ の zip ファイルに含まれたサンプルコードを使用して、OpenMP やインテル® スレディング・ビルディング・ブロック (インテル® TBB) を使用したコードのパフォーマンス分析結果についての解釈方法を学習します。</p> <p>&lt;install_dir&gt;\documentation\&lt;locale&gt;\ のサンプル・コード・ガイド (英語) を参照してください。</p>
インテル® Parallel Studio の資料	<p>インテル® Parallel Studio は、インテル® Parallel Amplifier、インテル® Parallel Composer、インテル® Parallel Inspector が含まれた並列化のための包括的なツールセットを提供します。</p> <p>インテル® Parallel Studio の全コンポーネントの概要は、『インテル® Parallel Studio 入門ガイド』を参照してください。</p> <p>Windows の [スタート] メニューから、<b>[Intel Parallel Studio (インテル(R) Parallel Studio)] &gt; [Getting Started (入門ガイド)] &gt; [Parallel Studio Getting Started Guide (インテル(R) Parallel Studio 入門ガイド)]</b> をクリックして表示します。</p> <p>インストールされている各インテル® Parallel Studio ツールのドキュメントを開くには、Windows の [スタート] メニューから、<b>[Intel Parallel Studio (インテル(R) Parallel Studio)] &gt; [Parallel Studio Documentation (インテル(R) Parallel Studio ドキュメント)] &gt; [ツール名 ドキュメント]</b> を選択します。</p> <p>インテル® Parallel Studio に関するその他の情報は、<a href="http://www.intel.com/software/products/">http://www.intel.com/software/products/</a> を参照してください。</p>



## 著作権と商標について

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスも許諾するものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証(特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他知的財産権の侵害への保証を含む)にも一切応じないものとします。

インテルによる書面での合意がない限り、インテル製品は、その欠陥や故障によって人身事故が発生するようなアプリケーションでの使用を想定した設計は行われていません。

インテル製品は、予告なく仕様や説明が変更される場合があります。機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。この情報は予告なく変更されることがあります。この情報だけに基づいて設計を最終的なものとししないでください。

本書で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。

最新の仕様をご希望の場合や製品をご注文の場合は、お近くのインテルの営業所または販売代理店にお問い合わせください。

本書で紹介されている注文番号付きのドキュメントや、インテルのその他の資料を入手するには、1-800-548-4725 (アメリカ合衆国)までご連絡いただくか、インテルの Web サイトを参照してください。

インテル® プロセッサ・ナンバーはパフォーマンスの指標ではありません。プロセッサ・ナンバーは同一プロセッサ・ファミリー内の製品の機能を区別します。異なるプロセッサ・ファミリー間の機能の区別には用いません。詳細については、[http://www.intel.co.jp/products/processor\\_number/](http://www.intel.co.jp/products/processor_number/) を参照してください。

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。

© 2009 Intel Corporation. 無断での引用、転載を禁じます。

Microsoft 製品のスクリーンショットは、Microsoft Corporation の許可を得て使用しています。