インテル® VTune™ Amplifier XE によるマルチスレッド化とタスク解析

Vladimir Tsymbal
インテル コーポレーション
ソフトウェア & サービスグループ テクニカル・コンサルティング・エンジニア

パフォーマンス向上の可能性を求める開発者は、実行のシリアルステージと並列ステージを組み合わせたパイプラインを構成できるアプリケーションのクラスを検討することがあるでしょう。その場合スレッドの管理には注意が必要です。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）を使用すると、適切なスレッド管理によって並列性が向上し、パイプラインの構築など並列アルゴリズムの設計とサポートにかかる時間や労力を軽減できます。インテル® TBB により、プログラマーは実行ステージをスレッドにマップしてスレッド間のワークバランスを取るという困難な作業をしなくて済みます。問題を実行タスクのセットとして表現するだけで、インテル® TBB は、システムで利用可能なハードウェア・スレッドへタスクを動的に割り当てます。タスク管理はアプリケーションの複雑さに応じて重要になります。インテル® VTune™ Amplifier XE のタスク解析では、問題調査向けの便利なビジュアル・インストルメントが提供されるため、プログラマーは時間をかけることなく、ユーザータスクを調整することができます。ここでは、実際の問題を単純化した例を用いて、パフォーマンス向上を実現する並列化、パイプラインの構築、タスク解析について説明します。

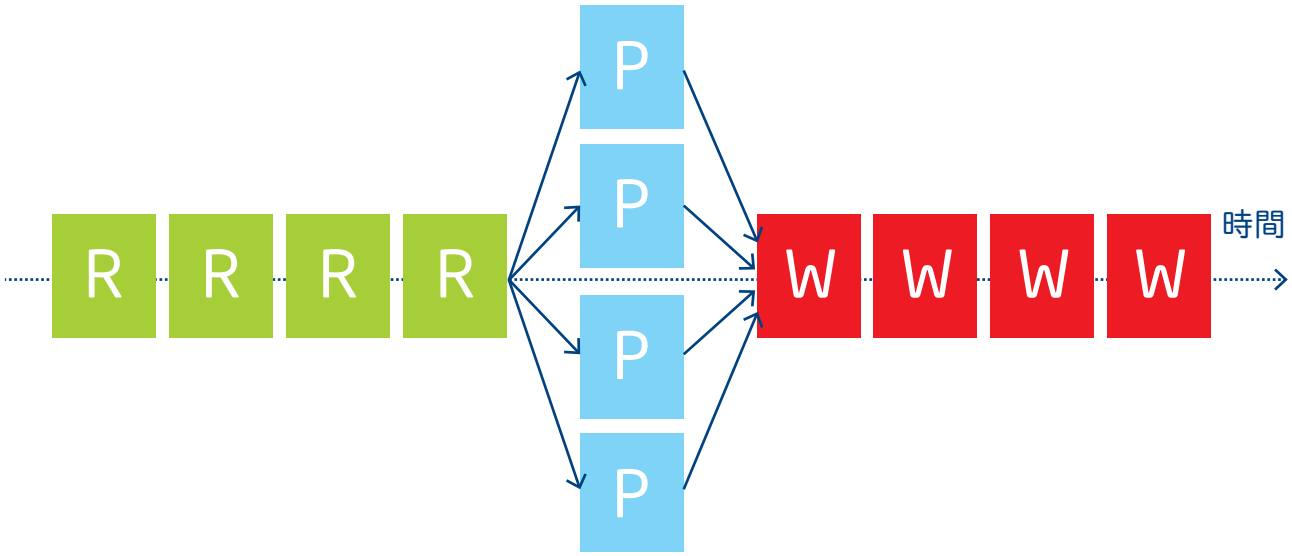
インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

実行タスクは、与えられた量のワークを一連の実行ユニットに分配することで並列化できることはよく知られています。実行問題や入力データ構造に応じて、データ分割またはタスク分割を適用します。**データ分割による並列化**とは、データ配列を、同じ処理ユニットで並列に実行するチャンクに分割します。**タスク分割による並列化**とは、複数の異なる処理ユニットが同じデータを指定された方法で並列処理します。問題の実行を分割する際の一般的な指標は、すべての処理ユニットにワークを提供することと、問題を解く間すべての処理ユニットがビジーになるようにすることです。タスクが処理ユニット間で適切に分配されないと、計算の効率が低下して、実行時間に悪影響を及ぼします。このため、パフォーマンスの最適化にはタスク実行の解析が不可欠です。

実際のタスクを単純化したケースとして、連続する入力データをデータソースから取得し、機能ユニットで処理によって、シンクに格納するタスク（**図 1**）を考えてみます。これは、コーデック、フィルター、コミュニケーション・レイヤーなどのアプリケーションで見られる一般的なデータフローです。

1

ここでは単純にするため、入力データのデータチャンク間の依存性は考慮しません。通常は、マルチメディア・デコーダー（ピクチャーブロックが隣接するセルに依存し、フレームが後続のものに依存する）のように依存性があります。ただし、このようなケースでも、独立して並列処理できるデータブロックを見つけることは可能です。そのため、ここではデータ分割モデルを適用して、独立したデータチャンクを処理ユニットに分配し、並列処理によって実行時間を短縮します。処理が完了すると、データは次のステージに書き込むユニットへ移行します（**図 2**）。

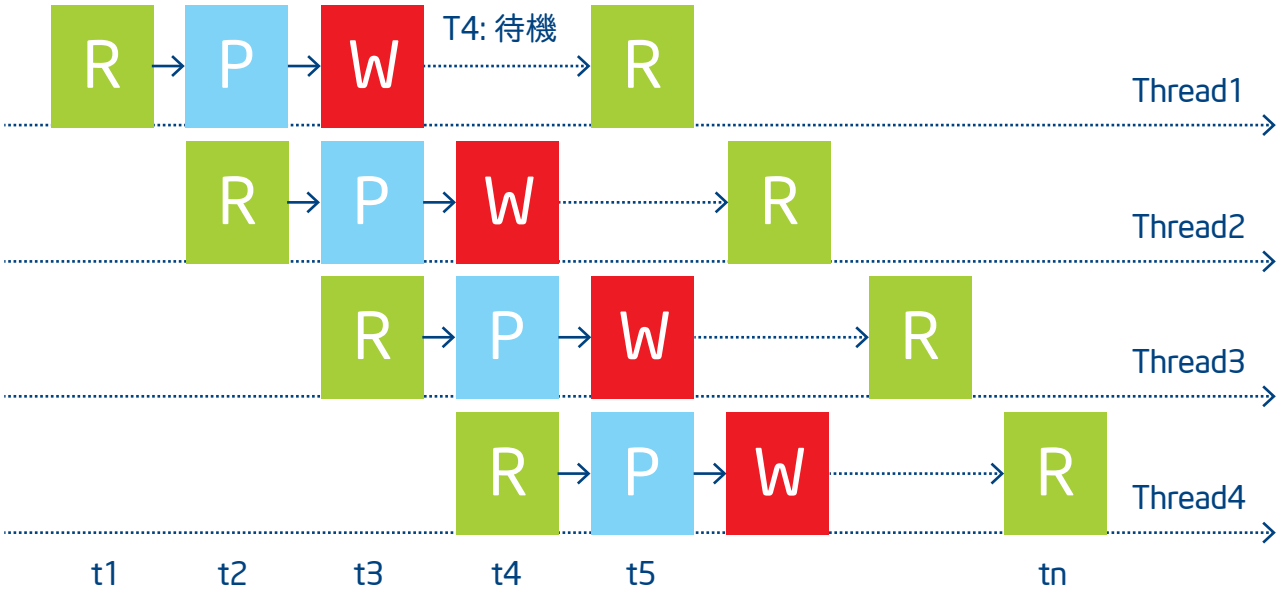


2

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

データソースが、ハードディスク、ストリーミング・デバイスのようなデバイスやネットワークの場合、データの読み取りと書き込みステージは基本的にシリアルです。また、システムに多くのハードウェア実行スレッドがあっても、実行時間を短縮できるのは処理ステージのみです。アムダールの法則によれば、最適化によるスピードアップの上限はシリアル部分の実行時間によって制限されます。読み取りおよび書き込みステージはシリアル部分であり、これらの処理を実行している間、残りのハードウェア・スレッドはスリープモードで待機します。

この制限を解消するには、入力データブロックが独立しておりデータの順序が決まっていない場合、あるいは後で順序を変更できる場合、ハードウェア・スレッドが常にビジーになるようにソフトウェアを再設計します。各スレッドにより実行される読み取り / 処理 / 書き込みステージから成る単純なパイプラインを構築すると良いでしょう（**図 3**）。読み取りおよび書き込みステージはシーケンシャルのままですが、実行スレッド間で分配されます。あるスレッドが読み取りまたは書き込みステージを実行しているときにほかのスレッドがそのステージを開始することはできませんが、2 つの異なるスレッドが読み取りと書き込みステージを（それぞれ）同時に実行することは可能であることに注意してください。

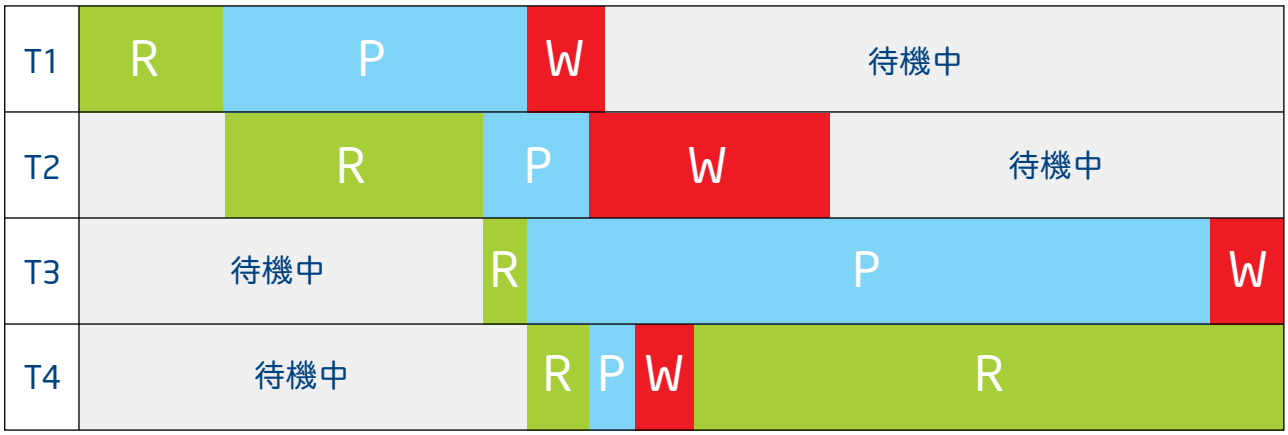


3

このように再設計して、複数のスレッドで別々のタスク（関数）を同時に実行する、並列タスク分割を導入します。この新しい設計では、同時にデータ分割も導入しています。これらの方法は、データ読み取り / 書き込みに遅延がないと仮定し、ハードウェア･スレッドを常にビジーになるようにします。また、I/O のオーバーラップをカバーする十分な量のワークが必要であり、処理ユニットは利用可能なプロセッサーで実行します。

実際のシステムやアプリケーションでは、読み取り / 書き込み操作が安定していること、プロセッサーがある瞬間にすべてビジーではないこと、データが一貫していてホモジニアスであることは保証されません。入力データの提供が遅れたり、入力バッファーをすべて満たすだけのデータが提供されないこともあります。また、プロセッサーが優先度の高いほかのタスクでビジーになっていたり、他のプロセスとリソースを共有していることもあります。データによっては、処理に多くのプロセッサー時間を要する場合もあります。例えば、コーデックは、背景や密度の低いシーンよりも詳細で密度の高いシーンに多くの処理が必要です。I/O サブシステムへの書き込み操作はメモリーにバッファーされるため通常は問題になりませんが、書き込んだデータが入力として必要になると問題が生じます。これらの問題が起こると、最初の並列実装と同じように効率の悪いパイプラン実行につながる可能性があります（**図 4**）。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



4

ワーク・インバランスにより引き起こされる問題を回避するには、スレッド間のワークロードを動的に分配します。このためには、タスクをより小さなタスクに分割して、ハードウェア・スレッドに渡せるように、ビジーでないスレッドのモニタリングとタスクの管理が必要です。スレッドとタスク管理インフラストラクチャーの実装はかなり複雑です。インテル® TBB のようなスレッディング・ライブラリーを利用することで、このようなインフラストラクチャーの開発とサポートにかかる負担を減らすことができます。この例に関しては、インテル® TBB ライブラリーの「パイプライン・クラス」と呼ばれる組込みアルゴリズムが適しているでしょう。

ここでは、インテル® TBB のパイプライン実装については詳しく説明しません（詳細は、http://threadingbuildingblocks.org/（英語）に掲載されているドキュメントやライブラリーのソースコードを参照してください）。次に、タスクの並列実行用のパイプラインを作成する方法を紹介します。このプロジェクトとソースコードは、インテル® TBB 製品のインストール・ディレクトリーに含まれています。また、Web サイトから入手することもできます。

基本的に、パイプラインは、一連の実行ステージをストリームへ適用します。実行ステージは、データストリームを処理するタスクとなります。インテル® TBB ライブラリーでは、これらのステージを *filter* クラスのインスタンスとして定義することができます。このため、一連のフィルターとしてパイプラインを構築します。一部のステージ（処理ステージなど）は異なる項目を同時（コンカレント）または並列（パラレル）に実行できるため、これらのステージは *parallel* filter クラスとして定義します。読み取りや書き込みなどのほかのステージはシリアルで順に実行しなければならないため、これらのステージは *serial_in_order* filter クラスとして定義します。インテル® TBB には、このようなフィルターの抽象クラスが含まれているため、これらのクラスから独自のクラスを派生させるだけで済みます。次に例を示します（ここでは分かりやすくするため、必要な定義をすべて含めていません）。

```
class MyReadFilter:public tbb::filter {
    FILE* input_file;
    DataItem* next_item;
    /*オーバーライド*/ void* operator()(void*);
public:
    MyReadFilter( FILE* in );
};
```

```
MyReadFilter::MyReadFilter( FILE* in ) :
    filter(serial_in_order),
    input_file(in),
    next_item(DataItem*)
{
}
```

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

```
class MyWriteFilter:public tbb::filter {
    FILE* output_file;
    /*オーバーライド*/ void* operator()(void*);
public:
    MyWriteFilter( FILE* out );
};
```

```
MyWriteFilter::MyWriteFilter( FILE* out ) :
    filter(serial_in_order),
    output_file(out),
{
}
```

データはファイルから読み取るため、クラスでファイルポインターのプライベート・メンバーを定義する必要があります。同様に、書き込みステージ用の MyWriteFilter クラスを定義し、適切なクラスメンバーで出力ファイルのポインターを割り当てます。これらのクラスは、パイプラインを通じてメモリーの割り当てとデータ項目の引渡しを担います。ステージの主な処理は、ベースクラスで定義される *operator()* メソッドで行われます。*operator()* メソッドをオーバーライドし、ファイルからデータコンテナーへのデータ項目の読み取りと、コンテナーから出力ファイルへのデータの書き込みを実装します。

```
void* MyReadFilter::operator()(void*) {
// メモリーを割り当て
// データ項目をファイルから読み取り
// データ項目をコンテナーに格納
}
```

```
void* MyWriteFilter::operator()(void*) {
// データ項目をコンテナーから取得
// データ項目をファイルに書き込み
// メモリーを解放
}
```

処理ステージは並列に実行できるため、*parallel* filter クラスとして定義し、アルゴリズムに従って *operator()* メソッドでストリーミング・データを処理します。

```
class MyProcessFilter:public tbb::filter {
public:
    MyProcessFilter();
    /*オーバーライド*/void* operator()( void* item );
};
MyProcessFilter::MyProcessFilter() :
    tbb::filter(parallel)
{}
```

```
void* MyProcessFilter::operator()( void*
item ) {
// コンテナーの現在のデータ項目を確認
// 項目を処理
}
```

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

最後のステップは、filter クラス・オブジェクトと pipeline クラス・オブジェクトを作成し、ステージをリンクしてパイプライン全体を構築することです。パイプラインを構築した後、pipeline クラスの *run()* メソッドを呼び出して、トークンの最大数を指定します。このケースでは、トークンの数は、パイプラインで同時に処理されるデータ項目の数です。この数の選択については、議論の余地がありそうですが、ここでは触れません。この例では、インテル® TBB のガイドラインに従って、利用可能なスレッド数の 2 倍にしました。その結果、各ステージで処理するデータ項目があることと、最初のステージが次のステージよりも高速にデータを処理できるため、データ項目キューが大きくならないことが保証されます。

```
tbb::pipeline pipeline;
 MyReadFilter r_filter( input_file );
 pipeline.add_filter( r_filter );

 MyProcessFilter p_filter;
 pipeline.add_filter(p_filter );

 MyWriteFilter w_filter( output_file );
 pipeline.add_filter( w_filter );

 pipeline.run(2*nthreads)
```

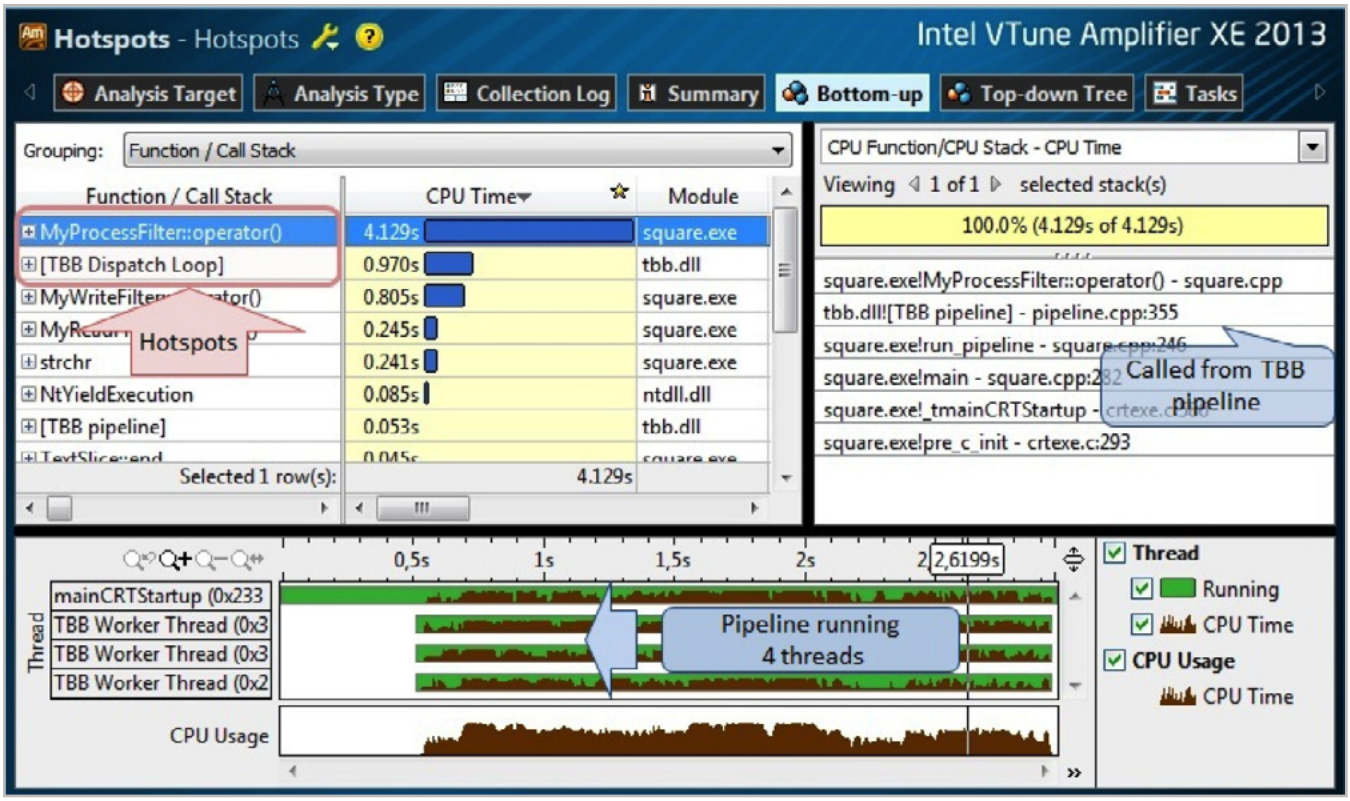
これで、プロセッサー・リソースの使用率を最大限にするように、インテル® TBB ライブラリーによって処理される 3 つのパイプライン化されたタスクを作成できました。処理ステージは並列化され、残りのステージとともに、ハードウェア・スレッドの状況に応じて異なるハードウェア・スレッドに動的に再割り当てされます。スレッド管理について考慮する必要はありません。注意が必要なのは、パイプラインのタスクを適切に作成してリンクすることだけです。

ここで次のような疑問が湧くかもしれません。「構築したパイプラインの効率は ?」「ステージはパイプラインでどのように実行されているか ?」「ライブラリーのオーバーヘッドやステージ間の実行ギャップはあるか ?」

これらの疑問に答えるには、パイプラインの実行を解析する必要があります。インテル® VTune™ Amplifier XE でデータ制御プロセスをプロファイルすることにより、プロセッサー・コアの使用効率を推定し、インテル® TBB ライブラリーのタスク管理の詳細を理解して、並列化、つまりパフォーマンス向上を妨げる落とし穴を見つけます。

ここでは一般的なプロファイル手順に従って、インテル® VTune™ Amplifier XE の hotspot プロファイルで解析をスタートします。10 進数を含むテキスト形式ファイルを読み取り [読み取り]、数を二乗に変更して [処理]、新しいファイルにデータを書き込む [書き込み] サンプルプログラムを解析します。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



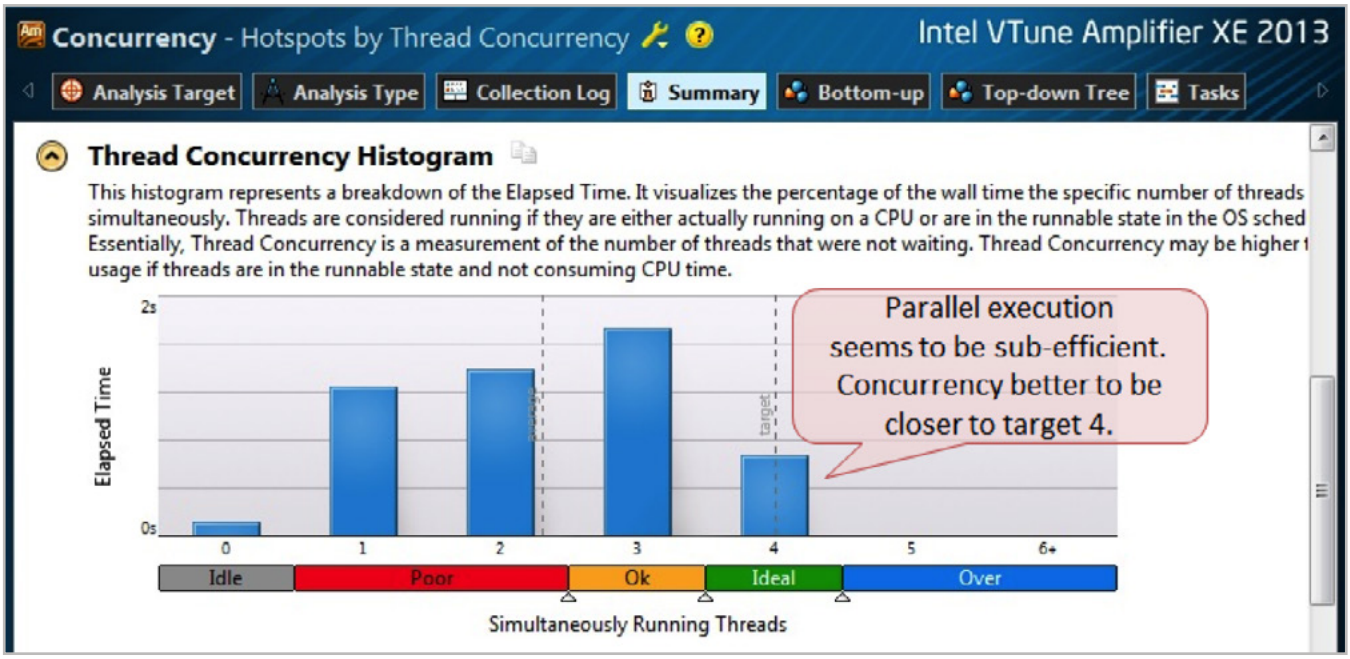
5

hotspot 解析の結果は、予想どおりでした (**図 5**)。テストはシステムで利用可能な 4 つのスレッドで実行されました。インテル® TBB パイプラインから呼び出される *MyProcessClass::operator()* メソッドは、テキストから整数への変換を行い、二乗を計算して、結果をテキストに戻しているため、最もホットな関数でした。hotspot リストの [TBB Dispatch Loop] は、実行中に発生したインテル® TBB のタスク・ディスパッチャーのオーバーヘッドであると考えられます。次に、コンカレンシー解析を行って、アプリケーションの並列実行の効率を測定します。

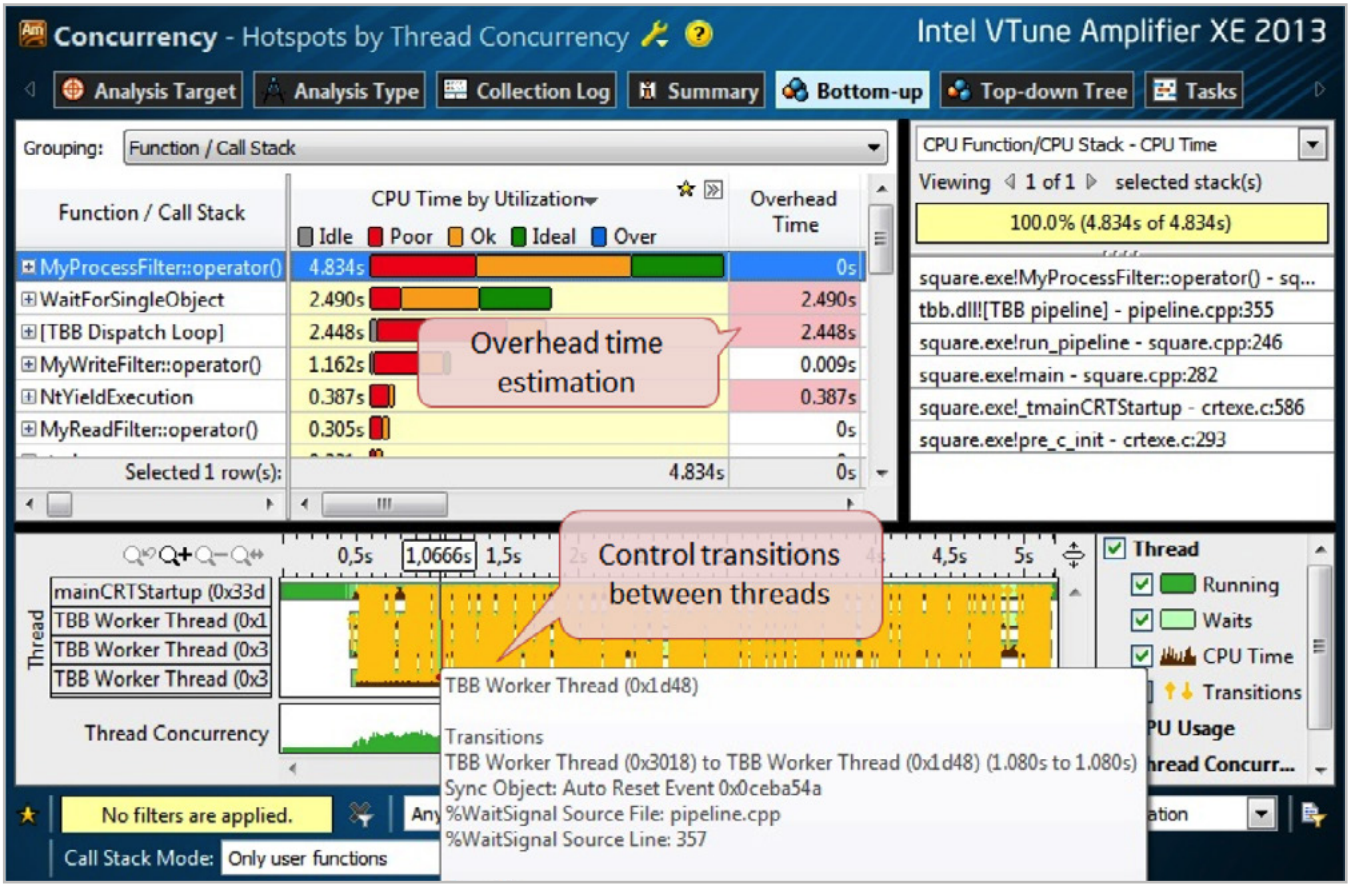
コンカレンシー解析の結果から、この並列実行は効率的ではなく、実行時間のほとんどで 4 スレッドが同時に実行されていないことが分かりました (**図 6**)。(注 : 青いコンカレンシー・レベルのバーは、スレッドが並列に実行された時間を示しています。コンカレンシー・レベル 4 にのみバーが表示される、つまり、4 つのコアの 4 つのスレッドでアプリケーション実行時間がすべて実行されることが理想的です。)

ボトムアップ・ビューでは、インテル® TBB パイプラインの同期オーバーヘッドが大きいことが分かります (**図 7**)。黄色の遷移行にマウスのカーソルを移動すると、この同期のソースを調べられます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



6



7

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

この時点で、非効率な並列処理の原因を探ることを諦め、インテル® TBB のパイプライン実装のせいにしたくなるかもしれません。しかし、タスク実行を確認しなければ調査が完了したとは言えません。タスクをトレースすると、タスクのシーケンスとタイミングに関してより詳しい情報が得られるでしょう。ソースコードにおけるタスクのカスタム・インストルメンテーションは、トレースを収集する最も簡単な方法です。ただし、トレース制御と解析表示には労力が要ります。インテル® VTune™ Amplifier XE は、トレースを収集してグラフィカルに表現する、強力なタスク解析インストルメントを提供するため、迅速に検証できます。

タスク解析を行うには、ツールの特定のタスク API を使用してタスク実行コードをインストルメントする必要があります。次に手順を示します。

1. ソースファイルに API ヘッダーをインクルードします。

```
#include "ittnotify.h"
```

2. プログラムでタスクドメインを定義します。プロジェクトで使用するさまざまなスレッディング・ドメインを区別できるようにしておくと便利です。

```
__itt_domain* domain = __itt_domain_create("PipelineTaskDomain");
```

3. ステージに対応するタスクハンドルを定義します。

```
__itt_string_handle* hFileReadSubtask = __itt_string_handle_create("Read");
__itt_string_handle* hFileWriteSubtask = __itt_string_handle_create("Write");
__itt_string_handle* hDoSquareSubtask = __itt_string_handle_create("Do Square");
```

4. 各ステージの実行ソースコードを *__itt_task_begin* と *__itt_task_end* インストルメンテーション呼び出しでラップします。例えば、読み取りステージと書き込みステージは次のようになります。

```
void* MyReadFilter::operator()(void*) {
__itt_task_begin(domain, __itt_null, __itt_null, hFileReadSubtask);
// メモリーを割り当て
// データ項目をファイルから読み取り
// データ項目をコンテナーに格納
__itt_task_end(domain);
}
```

```
void* MyWriteFilter::operator()(void*) {
__itt_task_begin(domain, __itt_null, __itt_null, hFileWriteSubtask);
// データ項目をコンテナーから取得
// データ項目をファイルに書き込み
// メモリーを解放
__itt_task_end(domain);
}
```

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

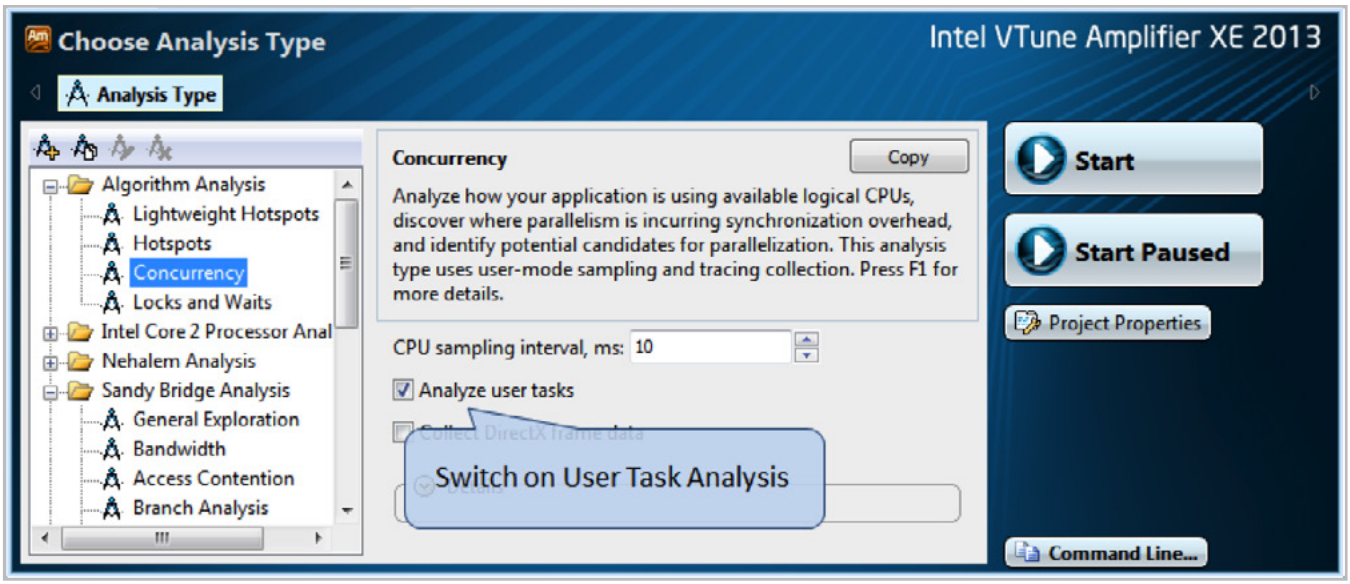
処理ステージは次のようになります (API 呼び出しに関する詳細は、製品のドキュメントを参照してください)。

```
void* MyProcessFilter::operator()( void* item ) {
__itt_task_begin(domain, __itt_null, __itt_null, hDoSquareSubtask);
// コンテナーの現在のデータ項目を確認
// 項目を処理
__itt_task_end(domain);
}
```

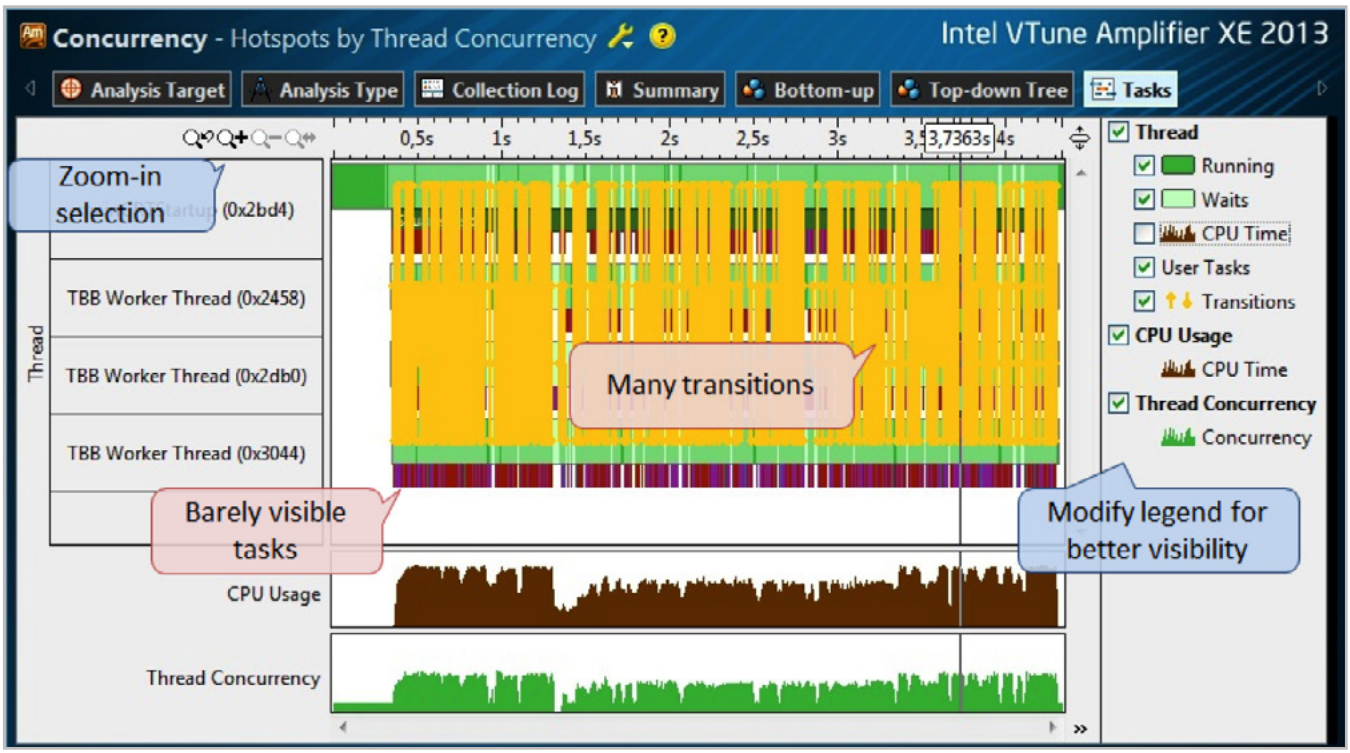
5. プロジェクトにインテル® VTune™ Amplifier XE ヘッダーのパスを追加します。
   `$(VTUNE_AMPLIFIER_XE_2013_DIR)`
6. システムの環境に応じて、パス `$(VTUNE_AMPLIFIER_XE_2013_DIR)lib[32|64]` にある *libittnotify.lib* を静的にリンクします。

最後に、インテル® VTune™ Amplifier XE の解析設定ウィンドウでユーザータスクの解析をオンにします (**図 8**)。これで、任意の解析タイプを実行して、タスクにマップできる結果が得られます。

コンカレンシー解析の収集を実行した後、タスクビューを表示します (**図 9**)。タイムライン全体に黄色の遷移行が表示されます。これらの行は、右ペインのオプションをオフにすると非表示になります。同様に、茶色の CPU 時間グラフの表示も設定できます。しかし、カラーバーは非常に細く識別が難しいため、タイムラインでタスクを確認するには、時間範囲を選択してコンテキスト・メニューやズームツールで時間範囲を拡大すると良いでしょう。



8

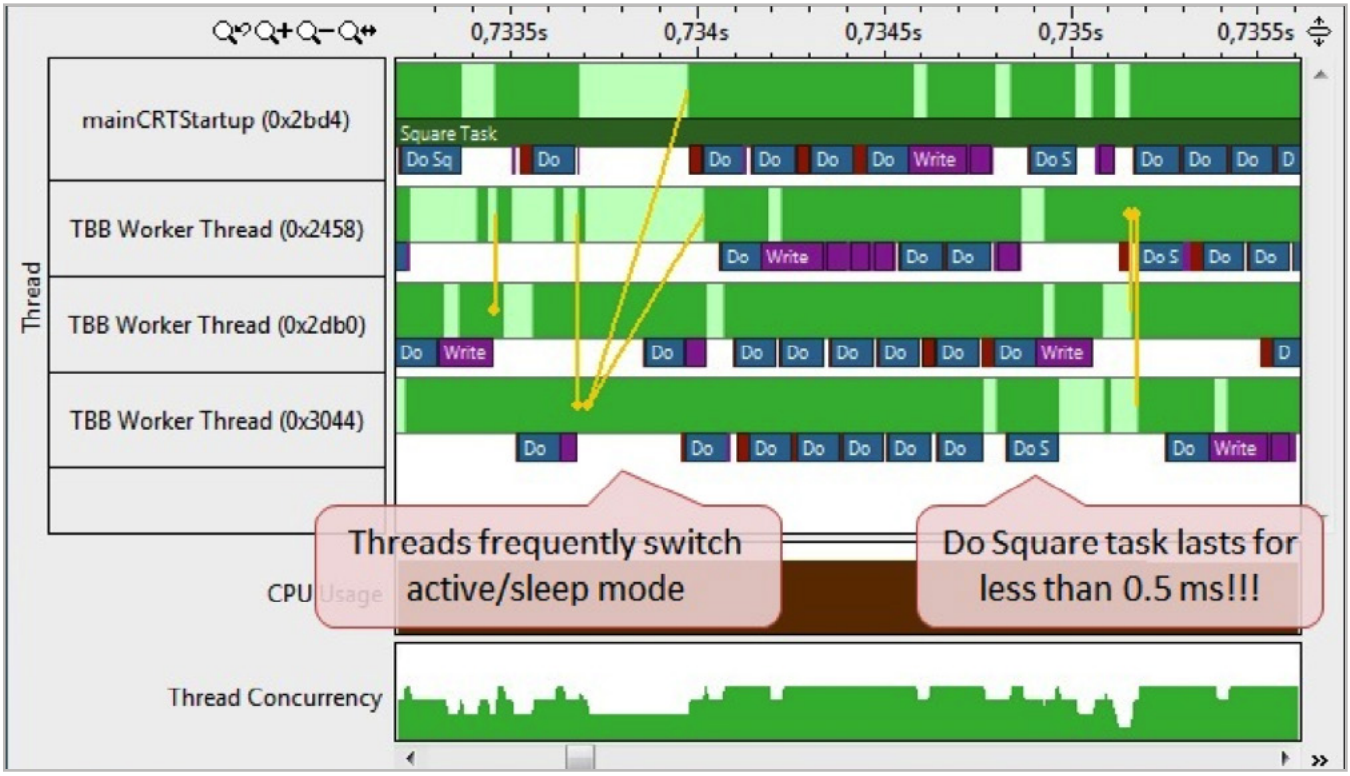


9

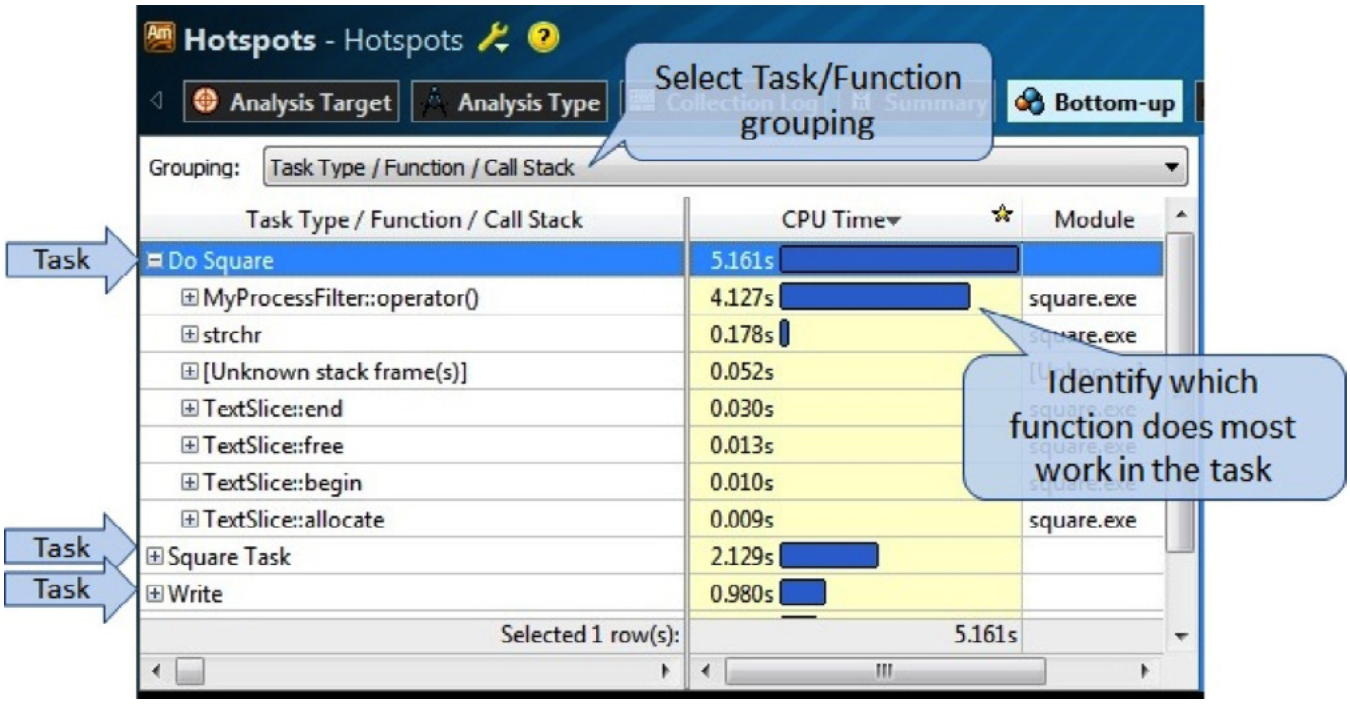
タイムラインを拡大したところ、2 つの問題が明らかになりました（**図 10**）。1 つ目は、タスク間に大きなギャップがあり、その間スレッドはアクティブではない（同期のために待機している）ことです。2 つ目は、タスク時間で、「Do Square」タスクは約 0.1 ミリ秒しか実行されていないことです。この期間はタスクがインテル® TBB のタスク・スケジューラーで制御されることを考慮すると非常に短く、スケジューラーが頻繁にタスクを制御しなければならないため、オーバーヘッドが大きくなります。解決方法は、各タスクのワーク量を増やしてタスク管理のオーバーヘッドを減らすことです。

この簡単な例では、どの関数がどのタスクを実行しているかが明白ですが、実際のアプリケーションでは、タスクは多くの関数で実行されます。関数を識別してタスクのワーク量を増やす場所を判断するには、コンカレンシー解析あるいは hotspot 解析の結果をボトムアップ･ビューで確認すると良いでしょう。タスクタイプ / 関数にグループを変更して、グリッドのタスクリストを調べます。このグループのタスクを展開すると、関数ツリーと各関数の CPU タスク時間が表示されます（**図 11**）。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



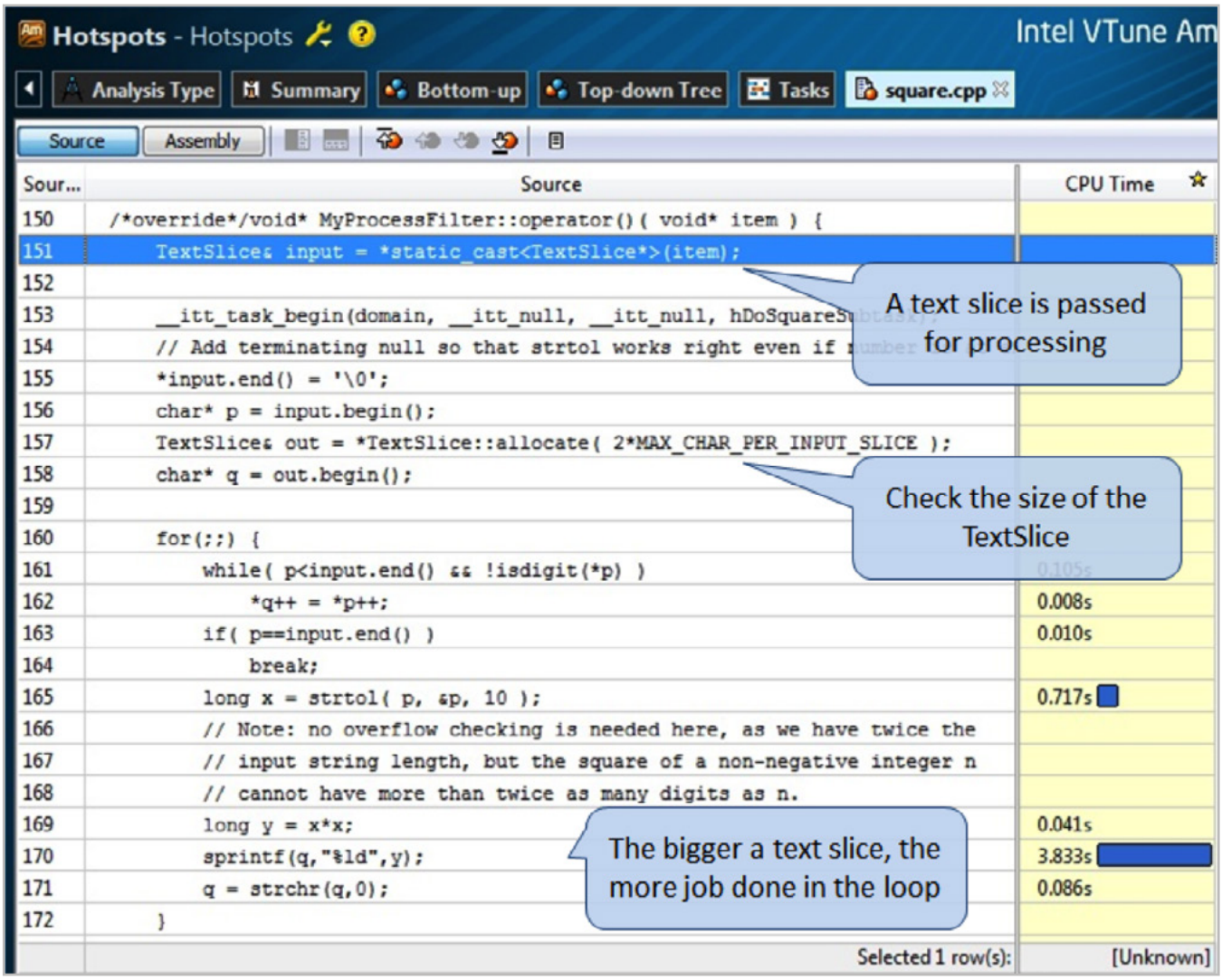
10



11

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

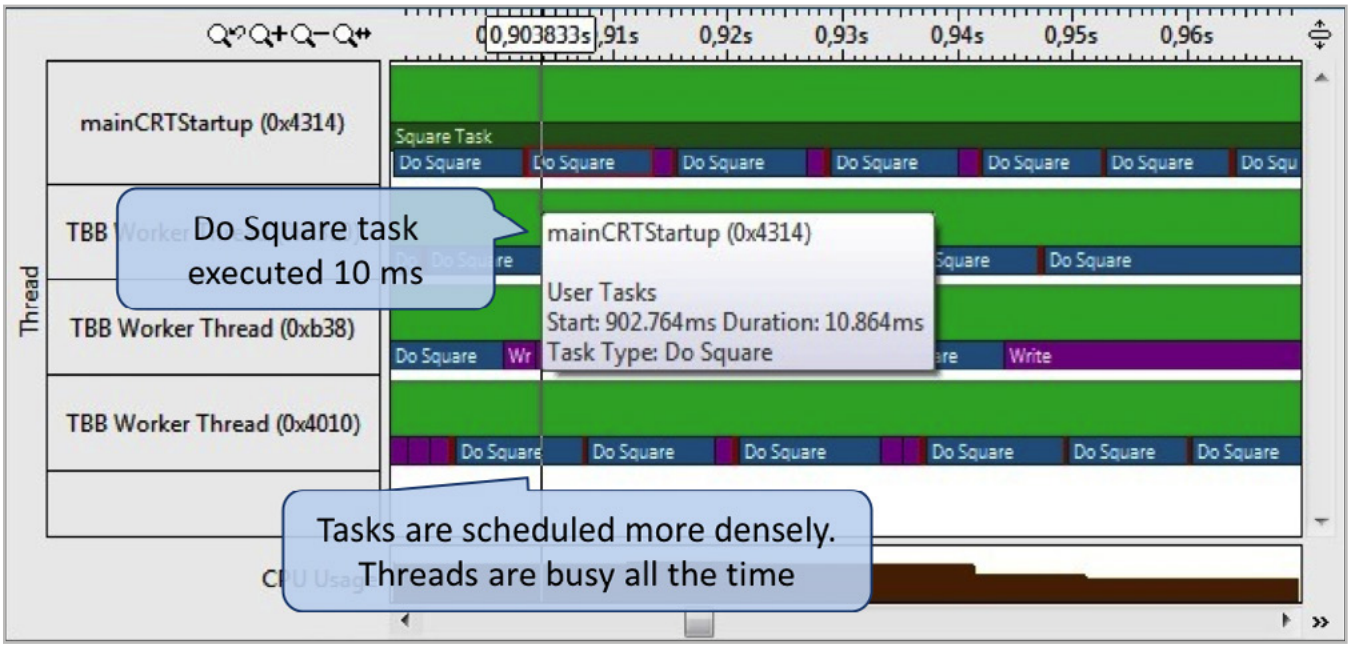
次に、*MyProcessFilter::operator()* 関数のソースコードにドリルダウンすると、関数にテキストスライス項目が渡されていることが分かります (**図 12**)。このテキストデータを反復して、各文字列を long 型の値に変換し、値を乗算して二乗の値を取得した後、値を文字列に変換しています。この関数のワークロードを増やす最も簡単な方法は、テキストスライスのサイズを増やして、処理操作の数を増やすことです。ここでは、新しいサイズの制限として、(タスク実行タイミングに基づいて) `MAX_CHAR_PER_INPUT_SLICE` にオリジナルの 100 倍の値を選択しました。また、読み取り操作と書き込み操作でも、スライスを大きくすることでメリットが得られると考えました。



12

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

再コンパイルしたアプリケーションでは、少なくとも画面上は、タスク解析の結果が向上しました（**図 13**）。「Do Square」タスクの実行時間は約 10 ミリ秒になりました（タスクブロックの上にマウスのカーソルを移動すると、注釈に情報が表示されます）。タスク間のギャップはほとんどなくなり、スレッドはほぼビジーになっています。全体的なスケジューリングと切り替えのオーバーヘッドを最小化してスレッドをビジーに保つため、パイプラインが書き込みタスクのような同様のタスクをスレッドの 1 つのスタックにスケジューリングしていることも分かります。

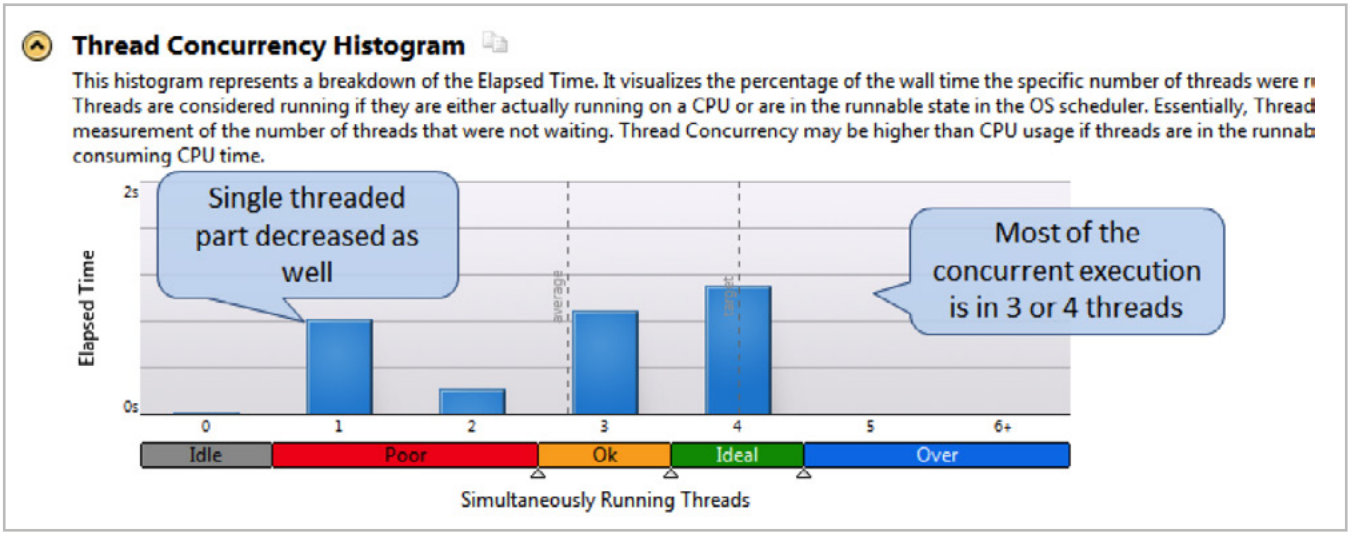


13

スレッド・コンカレンシー・ヒストグラムを見て、全体的なコンカレンシーが向上しているかを確認します。画面（**図 14**）から分かるように、並列実行のほとんどは 3 つまたは 4 つのスレッドで行われています。最初の解析のヒストグラムと比較すると、コンカレンシーが大幅に向上していることが分かります。平均のコンカレンシー数を比較することもできますが、この値は実行のシリアル部分も考慮して計算されています。

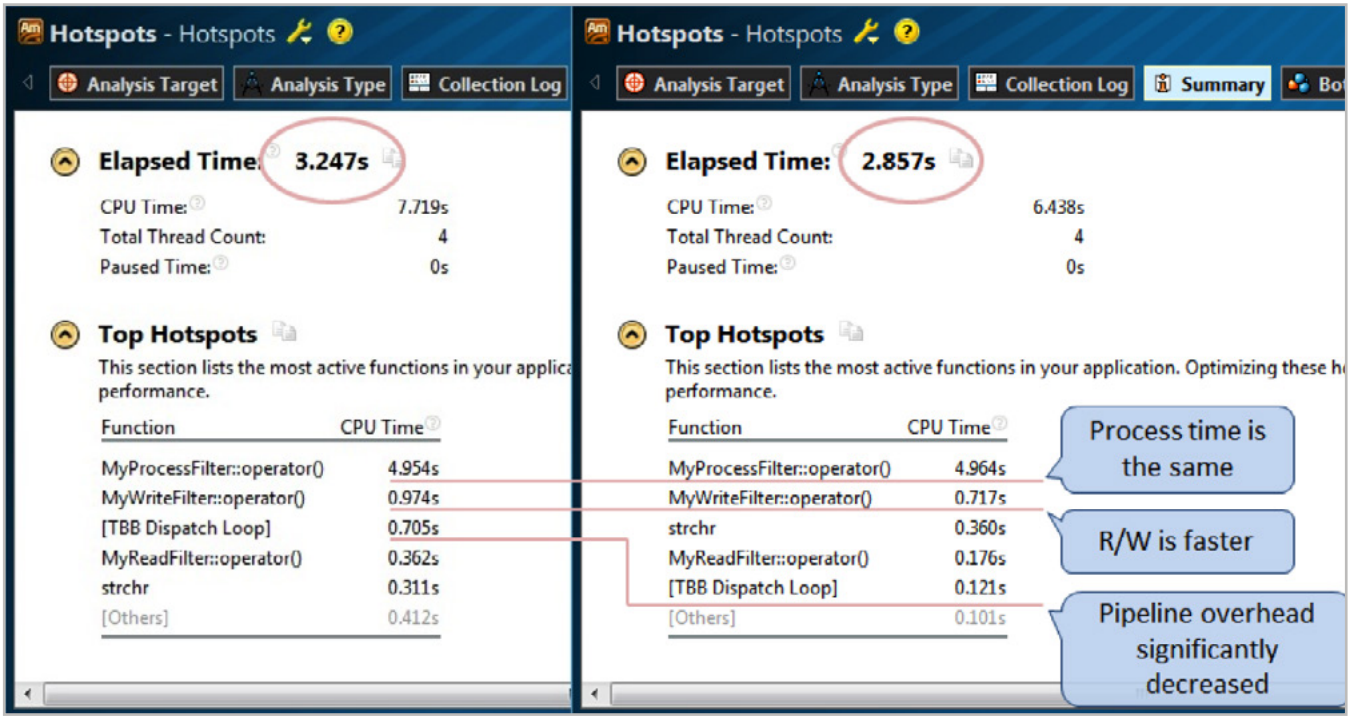
単一スレッドで実行されている部分もまだ残っていますが、オリジナルの値よりもかなり少なくなりました。つまり、プログラムのシリアル実行部分はまだ存在しますが、パイプラインにより並列性が向上したことになります（タイムライン・ビューの始めで確認できます）。シリアル実行部分を選択して表示すると、アプリケーションのメインスレッドの初期化フェーズであることが分かります。アプリケーションの初期化を行うコード部分が明確に識別できる場合は、ポーズ状態で開始する API を用いて、アプリケーションが実際のワークを行う時点でレジューム API を使用して収集を再開すると良いでしょう。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



14

向上した部分を数字で確認する場合は、ツールの比較機能を使用します。この機能は任意のビューで利用可能です。例えば、変更前と変更後の hotspot 解析の対応するサマリーペインを比較できます (**図 15**)。経過時間は短縮されましたが、ワークの量は変更していないため、処理ステージ *MyProcessFilter::operator()* の期間は変わりません。しかし、インテル® TBB のタスク・ディスパッチのオーバーヘッドは大幅に低下しています。データスライスがより大きくなったことで、読み取りステージと書き込みステージの時間も同様に減りました。



15

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

まとめ

データおよびタスクレベルの分割は、アルゴリズムの効率良い並列化を達成するために広く使用されています。多くのアプリケーションは、実行のシリアルステージと並列ステージを組み合わせたパイプラインから恩恵を得られるでしょう。インテル® TBB は、より優れた並列化のためのスレッド管理を行うことで、パイプラインのような並列アルゴリズムの設計とサポートに費やす時間や労力を大幅に軽減できます。インテル® TBB を利用することで、プログラマーはスレッドの割り当てと実行を管理する必要がありません。問題を実行タスクのセットとして表現するだけで、インテル® TBB は、システムで利用可能な実行スレッドへタスクを動的に割り当てます。タスク作成はアプリケーションの複雑さに応じて重要になります。インテル® VTune™ Amplifier XE のタスク解析では、問題調査のためのビジュアルで便利な方法が提供されるため、プログラマーは時間をかけることなく、ユーザータスクの強力なパフォーマンス・トレースを実施できます。

BLOG HIGHLIGHTS

インテル® Xeon Phi™ コプロセッサーの電力管理設定：micsmc コマンドライン・インターフェイスの使用

TAYLOR KIDD »

ここでは、各種 micsmc コマンドライン・オプションで取得可能なセンサーデータの目的と解釈について紹介します。コプロセッサーにはパッシブ冷却とアクティブ冷却の 2 つのタイプの SKU が用意されており、インレット、アウトレット、ファンに関連するセンサーがあります。これらのセンサーは両方のタイプのコプロセッサーに存在しますが、ここで説明する内容はアクティブ版向けです。パッシブ版のセンサーの意義と有用性は、コプロセッサーを含むホストのハウジングで提供される冷却システムに依存します。

コマンドラインの使い方：電力の測定

コマンドラインでは、GUI と同じ操作に加えて、さらに多くの操作が可能です。コマンドのリストは、「Table FULL」を参照してください。「Table POWER」は、電力管理に最も関連するオプションを示しています。

この記事の続きはこちら（英語）でご覧になれます。 ›

```
twkidd@knightscorner1:~> micsmc --help

Intel(R) Xeon Phi(TM) Coprocessor Platform Status Panel
VERSION: 3.1-0.1.build0
Developed by Intel Corporation. Intel, Xeon, and Intel Xeon Phi are trademarks
of Intel Corporation in the U.S and/or other countries.

This application monitors device performance, including driver info,
temperatures, core usage, etc.

This program is based in part on the work of the Qwt project
(http://qwt.sf.net).

The Status Panel User Guide is available in all supported languages, in PDF and
HTML formats, at:

   "/opt/intel/mic/sysmgmt/docs"+
```

Parallel Universe 17 号の全文はこちらでご覧いただけます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。