# The Parallel Universe

オープンソース Python*、R、Julia ベースの

# HPC アプリケーションの高速化

intel® Software

Issue 17
2014

# 目次


**編集者からのメッセージ** 3
パフォーマンス・マスター・クラス
James Reinders

注目記事

**オープンソース Python*、R、Julia ベースの HPC アプリケーションの高速化** 4
Vipin Kumar E K
インテル® コンパイラーを使用して簡単な作業を行うだけで、ハイパフォーマンスなインテル® MKL 対応の Python*、R、Julia をビルドしインストールすることができます。

**インテル® VTune™ Amplifier XE によるマルチスレッド化とタスク解析** 13
Vladimir Tsymbal
スレッド管理を通じて並列性を向上させることにより、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かかる時間や労力を軽減します。

**スレッド・パフォーマンスをスピードアップ : インテル® C++ コンパイラーによるインテル® TSX のサポート** 29
Anoop Madhusoodhanan Prabha
インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) により、クリティカル・セクションを並行実行して、プログラムの並行性を活かすことができます。

**Tachyon レイトレーサー : インテル® Xeon Phi™ コプロセッサーへの移植** 35
Roman Lygin および Dmitry Durnov
オープンソースのレイトレーサーであり、Spec MPI* スイートに含まれる Tachyon を、インテル® Xeon Phi™ コプロセッサーやインテル® Xeon® プロセッサーに移植する方法を説明します。

**インテル® C++ コンパイラーによるネイティブ Android* アプリケーションのビルド** 46
Anoop Madhusoodhanan Prabha
インテル® C++ コンパイラーは、Android* アプリケーションに対して何を行えるのか、主要な機能と利点を紹介します。



© 2014 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Xeon、Intel Xeon Phi、Cilk、VTune は、アメリカ合衆国および / または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。

# 編集者からのメッセージ

**James Reinders**
インテル コーポレーションの並列プログラミング・エバンジェリスト兼ディレクター
Morgan Kaufmann から出版された 2 冊の新しい書籍、『Intel® Xeon Phi™ Coprocessor High Performance Programming』（2013、日本語：『インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング』）および『Structured Parallel Programming』（2012、日本語：『構造化並列プログラミング』）の共著者。このほかに、『Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism』（O' Reilly Media、2007、日本語：『インテル スレッディング・ビルディング・ブロック ーマルチコア時代の C++ 並列プログラミング』、中国語、韓国語の翻訳版があります）や『VTune™ Performance Analyzer Essentials』(Intel Press, 2005) などの文献を発表しています。

# パフォーマンス・マスター・クラス

パフォーマンスに対する需要がますます高まるにつれ、開発者は HPC レベルの能力をアプリケーションにもたらす、新しい言語や手法に取り組んでいます。この号では、アプリケーションの機能を強化したり拡大できる開発テクニックや優れた問題解決の「インサイダー」情報を紹介します。

注目記事である「オープンソース Python*、R、Julia ベースの HPC アプリケーションの高速化」では、Linux* 上でインテル® ソフトウェア・ツールを使用した（ベクトル化や OpenMP* の活用を含む）パフォーマンスの最適化について解説します。

「インテル® VTune™ Amplifier XE によるマルチスレッド化とタスク解析」では、タスク解析、スレッド化、タスク管理を通じて、実行のシリアルステージおよび並列ステージのパイプライン化を促進する過程を紹介します。

「スレッド・パフォーマンスをスピードアップ：インテル® C++ コンパイラーによるインテル® TSX のサポート」では、ハードウェア・レイヤーで実装されるトランザクショナル・メモリー・ソリューション、インテル® トランザクショナル・シンクロナイゼーション・エクステンション（インテル®TSX）について説明します。また、アプリケーションのクリティカル・セクションを並行実行することで、効率性を評価して向上する方法を調査します。

「Tachyon レイトレーサー：インテル® Xeon Phi™ コプロセッサーへの移植」では、オープンソースのレイトレーサーであり、Spec MPI* スイートに含まれる Tachyon を、インテル® Xeon Phi™ コプロセッサーやインテル® Xeon® プロセッサーに移植する手順を紹介します。

「インテル® C++ コンパイラーによるネイティブ Android* アプリケーションのビルド」では、Android* アプリケーションの能力を引き出す方法について説明します。

今すぐ取り組まなければならないと感じているのは、あなただけではありません。パフォーマンスの最適化がミッション・クリティカルの要件である、競争に不可欠である、革新が必要であるなど、いずれの場合でも、ここで紹介したツールやヒントが皆さんの知識として吸収され、ソフトウェアの高速化に役立つことを願っています。

**James Reinders**

2014 年 3 月

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。

# オープンソース Python*、R、Julia ベースの HPC アプリケーションの高速化

**Vipin Kumar E K**
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

## インテル® コンパイラーとインテル® マス・カーネル・ライブラリー（インテル® MKL）によるパフォーマンスの向上

HPC コミュニティーでは、Python*、R、新しい Julia などのオープンソース言語に対する関心が高まっています。この記事では、GNU* コンパイラーと NumPy/SciPy、R、Julia 言語に同梱されているデフォルトの算術ライブラリーの代わりに、インテル® コンパイラーとインテル® マス・カーネル・ライブラリー（インテル® MKL）を利用してこれらの言語をビルドしインストールする方法を説明します。Linux* プラットフォーム上でこれらの言語をインテル® コンパイラーからビルドすることで、ベクトル化、OpenMP*、その他のコンパイラー機能を利用できるようになります。さらに、インテル® プラットフォーム用に高度に最適化されたインテル® MKL を介して、アプリケーションのパフォーマンスは大幅に向上するでしょう。

これらの言語には、インテル® コンパイラーやインテル® MKL が持つ最適化機能が含まれていません。その 1 つの機能はベクトル化です。ベクトル化により、インテル® アドバンスト・ベクトル・エクステンション（インテル® AVX1/AVX2）、インテル® Xeon Phi™ コプロセッサーの 512 ビット幅 SIMD、リリース予定のインテル® AVX-512 レジスターと命令のような、最新の SIMD ベクトルユニットを活用できます。インテル® ソフトウェア・ツールを導入すると、プロシージャー間の最適化、キャッシュとレジスターの効率良い使用、並列化などのインテル® アーキテクチャーの機能によって、プロセッサーのコアを最大限に利用できます。

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。

インテル® Xeon Phi™ コプロセッサーがシステムに搭載されている場合、インテル® MKL の自動オフロード機能を有効にしてコプロセッサーを活用することで、パフォーマンスをさらに向上できます。GEMM、SYMM、TRSM、TRMM、LU、QR、コレスキー分解など、インテル® MKL の一部の BLAS および LAPACK 関数は、ホスト・プロセッサーとインテル® Xeon Phi™ コプロセッサー間で計算を自動的に分割できます。この機能を有効にするには、MKL_MIC_ENABLE=1 環境変数を設定します。問題サイズが大きくなるほど効果があります。

これらの言語のビルドを開始する前に、http://software.intel.com（英語）から最新のインテル® Parallel Studio XE またはインテル® Cluster Studio XE をダウンロードしてください。

インテル® ソフトウェア開発ツールのインストール・フォルダーから `compilervars.sh` を実行して、インテル® C/C++ コンパイラー、インテル® Fortran コンパイラー、インテル® MKL、インクルード・ファイル用の PATH、LD_LIBRARY_PATH、その他の環境変数をセットアップします。Linux* プラットフォームでは、インテル® コンパイラーおよびインテル® MKL のデフォルトのインストール先は /opt/intel/composer_xe_2013_sp1 です (インストールするバージョンによって異なります)。

64 ビットのインテル® アーキテクチャー（IA）向けに環境をセットアップするには、次のコマンドを実行します。

```
$source /opt/intel/composer_xe_2013_sp1/compilervars.sh intel64
```

32 ビットのインテル® アーキテクチャー（IA）向けの環境には、次のコマンドを実行します。

```
$source /opt/intel/composer_xe_2013_sp1/compilervars.sh ia32
```

## インテル® MKL とインテル® コンパイラーを使用した Python* のインストール

Python* は、移植が容易で迅速に開発できる、オープンソースのインタープリター型オブジェクト指向言語です。Python* は記述が簡単であり、NumPy や SciPy のようなライブラリーをサポートしていることから非常にポピュラーで、科学技術計算コミュニティーでは、C/C++ および Fortran に続き最も広く使用されている言語の 1 つです。Python* が優れている点は、ほかの言語で記述されたソフトウェアへ容易に統合し、既存のプログラムを実行する制御言語として利用できることです。Python* は、異なるシステムを組み合わせる言語として動作します。

### インテル® MKL とインテル® コンパイラーの最適化を使用した NumPy/SciPy のインストール

NumPy/SciPy は、科学技術計算や HPC（ハイパフォーマンス・コンピューティング）向けのオブジェクトとルーチンを提供する基本的なライブラリーです。インテル® MKL と統合することで、これらのライブラリーの BLAS および LAPACK のパフォーマンスを向上できます。インテル® Core™ i5 マシン（4 スレッド）でのテストでは、インテル® MKL で行列乗算を行った場合、オープンソースの ATLAS ライブラリー（4 スレッド）に比べてパフォーマンスが約 2 倍も向上しました。

**NumPy** は、可能であれば、ベクトルと行列の演算を BLAS および LAPACK 関数に自動的にマップします。インテル® MKL はこれらの標準インターフェイスに対応しているため、NumPy スクリプトを変更するだけでインテル® MKL の最適化機能の恩恵が得られます。

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。

NumPy (http://numpy.scipy.org（英語）) は、Python* による科学技術計算に必須のパッケージであり、次のものが含まれます。

- 強力な N 次元の配列オブジェクト
- 高度な（ブロードキャスト）関数
- C/C++ と Fortran コードを統合するツール
- 優れた線形代数、フーリエ変換、乱数機能

科学的な用途に加えて、NumPy は汎用データの効率良い多次元コンテナーとして利用することもできます。

**SciPy** (http://www.scipy.org（英語）) には、統計、最適化、積分、線形代数、フーリエ変換、信号 / 画像処理、ODE ソルバー、その他のモジュールが含まれています。SciPy ライブラリーは、便利で高速な N 次元の配列操作を提供する NumPy に依存します。また、NumPy 配列で動作するように構築されており、数値積分用のルーチンや Python* ユーザー向けの最適化など、多数の使いやすく効率良い数値計算ルーチンを提供します。

NumPy および SciPy のソースコードは、http://www.scipy.org/Download（英語）からダウンロードできます。

最新バージョンの NumPy と SciPy の tar ファイルをダウンロードしたら、ファイルを展開してソース・ディレクトリーを作成します。その後、次の操作を行います。

1. numpy のルートフォルダー **`numpy-x.x.x`** に移動し、既存のファイルから `site.cfg` を作成します。
2. `site.cfg` を次のように編集します。
   a. 64 ビット用の NumPy をビルドする場合は、トップレベルの NumPy ディレクトリーの `site.cfg` に次のように指定します（インテル® Parallel Studio XE 2013 またはインテル® Composer XE 2013 バージョンのデフォルトパスにインテル® MKL をインストールした場合）。

```
[mkl]
library_dirs = /opt/intel/composer_xe_2013_sp1/mkl/lib/intel64
include_dirs = /opt/intel/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

   b. 32 ビット用の NumPy をビルドする場合は、次のように指定します。

```
[mkl]
library_dirs = /opt/intel/composer_xe_2013_sp1/mkl/lib/ia32
include_dirs = /opt/intel/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

3. numpy/distutils/intelccompiler.py の cc_exe を次のように変更します。

```
self.cc_exe = 'icc -O3 -xavx -ipo -g -fPIC -fp-model strict -fomit-frame-pointer
-openmp -DMKL_ILP64'
```

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。

ここでは、**-O3** 最適化オプションを指定して、融合、アンロールとジャムのブロック、IF 文の折りたたみなど、より強力なループ変換を有効にしています。また、OpenMP* スレッド用の **-openmp** オプションを使用し、**-xavx** オプションでインテル® AVX 命令を生成するようにコンパイラーに指示します。プロセッサーのアーキテクチャーが不明な場合は、**-xHost** オプションを指定すると、コンパイル時にホストのプロセッサーが持つ最上位命令セットが使用されます。ILP64 インターフェイスを使用している場合は、**-DMKL_ILP64** コンパイラー・オプションを追加します。

プロセッサー固有のオプションの詳細は、`icc --help` を実行して確認できます。各種コンパイラー・オプションの詳細については、インテル® コンパイラーのドキュメントを参照してください。

インテル® Fortran コンパイラー向けのコンパイラー・オプションを使用するように、numpy-x.x.x/numpy/distutil/fcompiler/intel.py の Fortran コンパイラー設定ファイルを変更します。

32 ビットおよび 64 ビットの場合、次のオプションを指定します。

```
ifort -xhost -openmp -fp-model strict -i8 -fPIC
```

次のように、インテル® コンパイラーで NumPy をコンパイルしインストールします (32 ビット・プラットフォームの場合は "**intelem**" を "**intel**" に変更します)。

```
$python setup.py config --compiler=intelem build_clib
--compiler=intelem build_ext --compiler=intelem install
```

次のように、インテル® コンパイラーで SciPy をコンパイルしインストールします (32 ビット・プラットフォームの場合は "**intelem**" を "**intel**" に変更します)。

```
$python setup.py config --compiler=intelem --fcompiler=intelem
build_clib --compiler=intelem --fcompiler=intelem build_ext
--compiler=intelem --fcompiler=intelem install
```

次のように、**LD_LIBRARY_PATH** 環境変数をエクスポートして、インテル® MKL とインテル® コンパイラーのライブラリーのパスをセットアップします。

64 ビット・プラットフォームの場合 :

```
$export
LD_LIBRARY_PATH=/opt/intel/composer_xe_2013_sp1/mkl/lib/intel64:/opt/
intel/composer_xe_2013_sp1/lib/intel64:$LD_LIBRARY_PATH
```

32 ビット・プラットフォームの場合 :

```
$export
LD_LIBRARY_PATH=/opt/intel/composer_xe_2013_sp1/mkl/lib/ia32:/opt/intel/
composer_xe_2013_sp1/lib/ia32:$LD_LIBRARY_PATH
```

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。

インテル® MKL とインテル® Composer XE をデフォルト以外のディレクトリーにインストールした場合、**LD_LIBRARY_PATH** 変数が正しく動作しないことがあります。この問題を回避するには、**LD_RUN_PATH** 変数を設定した環境内で Python*、NumPy、SciPy をビルドします。例えば、32 ビット・プラットフォームでは、次のようにパスを設定します。

```
$export LD_RUN_PATH=/opt/intel/composer_xe_2013_sp1/lib/ia32:/opt/
intel/composer_xe_2013_sp1/mkl/lib/ia32
```

注 : パフォーマンスの観点から、および NumPy が CBLAS を使用することから、Fortran 形式 (列優先) ではなく、デフォルトの 'C' 形式 (行優先) の配列を使用することを推奨します。

## インテル® コンパイラーとインテル® MKL を使用した R のインストール

R は、統計計算、計算生物学、ビッグデータ・アナリティクスで頻繁に使用される複雑な計算を行う、高水準プログラミング言語です。R には、行列演算、数値積分、高度な統計などを実行する高レベルな関数が含まれており、さまざまな科学技術計算アプリケーションで利用できます。

ここでは、インテル® MKL の BLAS と LAPACK ライブラリーを介して R のパフォーマンスを改善する方法を説明します。ほかのインテル® MKL 関数を用いる場合は、ラッパーを利用します。インテル® Core™ i7 マシン (4 コア、SMT 有効) 上でインテル® MKL とインテル® コンパイラーの最適化を有効にして、標準 R ベンチマークの 3000x3000 行列のコレスキー分解を実行したところ、GNU* ツールの場合に比べて約 20 倍もスピードアップしました。

インテル® MKL の BLAS と LAPACK ライブラリーを使用するように R を設定するには、設定時に **-with-blas** オプションを指定します。

インテル® コンパイラーの最適化を活かすには、まず、設定プロセスで使われる AR および LD 変数をエクスポートして、インテル® コンパイラーのリンカーとライブラリー・アーカイブ・ツール用の環境変数を設定します。

```
$export AR="xiar"

$export LD="xild"
```

インテル® MKL と OpenMP* ライブラリーのパスを設定します。

```
$MKL_LIB_PATH='/opt/intel/composer_xe_2013_sp1/mkl/lib/intel64'

$OMP_LIB_PATH='/opt/intel/lib/intel64'

$export LD_LIBRARY_PATH=${MKL_LIB_PATH}:${OMP_LIB_PATH}

$MKL=" -L${MKL_LIB_PATH} -L${OMP_LIB_PATH} -lmkl_intel_lp64 -
lmkl_intel_thread -lmkl_core -liomp5 -lpthread"
```

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。

インテル® コンパイラーとコンパイラーの最適化機能を使用するように、R.x.x.x ソースのルートフォルダーに含まれている **config.site** ファイルを変更します。

1. 次のように C/C++ および Fortran コンパイラーとオプション関連行を変更して、config.site を編集します。

```
CC='icc -std=c99'
CFLAGS='-O3 -ipo -xavx -openmp'
F77='ifort'
FFLAGS='-O3 -ipo -xavx -openmp'
CXX='icpc'
CXXFLAGS='-O3 -ipo -xavx -openmp'
$./configure --with-blas="$MKL" --with-lapack
```

   デフォルトのスレッド数はシステムの物理コアの数と同じですが、OMP_NUM_THREADS または MKL_NUM_THREADS を設定して制御できます。

2. ログファイル config.log を調べて、インテル® MKL が構成テスト中に動作していたかどうかを確認します。

```
configure:29075: checking for dgemm_ in -lmkl_intel_lp64
-lmkl_intel_thread -lmkl_core -liomp5 -lpthread

configure:29096: icc -std=c99 -o conftest -O3 -ipo -openmp -xHost
-I/usr/local/include  -L/usr/local/lib64 conftest.c -lmkl_intel_lp64
-lmkl_intel_thread -lmkl_core -liomp5 -lpthread  -lifport -lifcoremt
-limf -lsvml -lm -lipgo -liomp5 -lirc -lpthread -lirc_s -ldl -lrt -ldl -lm  >&5

conftest.c(210): warning #266: function "dgemm_" declared implicitly
  dgemm_()
  ^

configure:29096: $? = 0
configure:29103: result: yes
configure:29620: checking whether double complex BLAS can be used
configure:29691: result: yes
configure:29711: checking whether the BLAS is complete
```

3. 次に、make を実行し、インテル® コンパイラーとインテル® MKL を使用して R をビルドしインストールします。

```
$make && make install
```

4. インテル® MKL ライブラリーが動的にリンクされている場合、ldd コマンドを実行してインテル® MKL とインテル® コンパイラーのライブラリーが R にリンクされていることを確認します。インテル® コンパイラーを使用して、インテル® MKL が正常にリンクされた場合、出力は次のようになります。

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。

```
$ ldd ./bin/exec/R
        linux-vdso.so.1 =>  (0x00007fff2a1ff000)
        libmkl_intel_lp64.so =>
/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64/libmkl_intel_lp64.so
(0x00007f7d89bf9000)
        libmkl_intel_thread.so =>
/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64/libmkl_intel_thread.so
(0x00007f7d88b5e000)
        libmkl_core.so =>
/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64/libmkl_core.so
(0x00007f7d87630000)
        libiomp5.so =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libiomp5.so
(0x00007f7d87317000)
        libpthread.so.0 => /lib64/libpthread.so.0
(0x00000036ec600000)
        libifport.so.5 =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libifport.so.5
(0x00007f7d870c8000)
        libifcoremt.so.5 =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libifcoremt.so.5
(0x00007f7d86d59000)
        libimf.so =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libimf.so
(0x00007f7d8689a000)
        libsvml.so =>
/opt/intel/composer_xe_2013_sp1.0.080/compiler/lib/intel64/libsvml.so
(0x00007f7d85e3f000)

............
```

R のパフォーマンスを検証するため、クアッドコアのインテル® Core™ i7-975 プロセッサー エクストリーム・エディション（8MB LLC、3.33GHz）、メモリー 16GB、RHEL* 6 x86_64 のシステムで、R benchmarks サイト（英語）の R-benchmark-25.R 使ってテストしました。最初に R-2.15.3 をインストールし、GNU* ツールチェーンを使用してベンチマークを実行しました。

インテル® Core™ i7 3770K クアッドコア（8MB LLC、3.50 GHz）、メモリー 16GB、Ubuntu* 13.1 のシステム上でインテル® MKL を使用して標準 R ベンチマーク R-2.15.3 を実行したところ、大幅にパフォーマンスが向上しました。次の結果で、最後の項目はベンチマークのすべてのテストの合計時間を示しています（単位はすべて秒）。

| | GNU* | インテル® MKL & インテル® コンパイラー |
|---|---|---|
| 2800x2800 行列のクロス積 (b = a’ * a) | 9.4276667 | 0.2966667 |
| 3000x3000 行列のコレスキー分解 | 3.7346666 | 0.1713333 |
| 15 種類の全テストの合計時間 | 29.1816667 | 5.4736667 |

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。

# インテル® MKL を使用した Julia のインストール

最後に、HPC コミュニティーで評判が高まっている Julia 言語のビルドとインストール方法を説明します。Julia は、ハイパフォーマンスな数値計算と科学技術計算に焦点を当てた高水準の動的プログラミング言語ですが、汎用プログラミングにも効果があります。Julia もインタープリター型言語ですが、コンパイラー型言語と同等の高い機能を提供するよう設計されています。Julia の実装設計は、コア言語プラットフォームとして多重ディスパッチを含み、高度な型システム、より速いコードを生成する動的な型推測、実行時の型に応じて生成されるコードの特殊化などに対応します。インテル® Core™ i7 マシン（4 コア）でのレベル 3 BLAS パフォーマンス・テストでは、インテル® MKL を使用した場合、デフォルトのインストールに比べてパフォーマンスが 1.5 ～ 2 倍に向上しました。

Julia のソースコードは GIT hub（https://github.com/JuliaLang/julia/（英語））からダウンロードできます

インテル® MKL 環境をセットアップするには、64 ビットと 8 バイト整数（ILP64）をサポートするようにビルドします。

次のようにインテル® 64 および ilp64 用の引数を渡して `mklvars.sh` スクリプトを実行し、インテル® MKL 環境変数を設定します。

```
$source /opt/intel/mkl/bin/mklvars.sh intel64 ilp64
```

32 ビット・アーキテクチャーの場合、環境変数を設定する引数として ia32 を指定します。

```
$source /opt/intel/mkl/bin/mklvars.sh ia32
```

今回のケースでは、64 ビットと ILP64 アーキテクチャーを使用するには、インテル® MKL のインターフェイス・レイヤー変数を次のように設定する必要がありました。

```
$export MKL_INTERFACE_LAYER=ILP64
```

次に、Julia のルートフォルダーにある Make.inc を編集するか、Make.user ファイルを作成して次の変更を加えます。

`USE_MKL=0` 変数を `USE_MKL=1` に変更します。

`USE_INTEL_IMF=1` を追加します。

インテル® MKL 環境を設定してインテル® MKL を使用するように Make.inc ファイルを変更したら、次のコマンドを実行してインテル® MKL 対応の Julia をビルドします。

```
$make install
```

make に引数を渡してインテル® MKL 対応の Julia をビルドすることもできます。

```
$make USE_MKL=1 install
```

インテル® コンパイラーの最適化機能を利用するには、次のコマンドで Julia をビルドします。

```
$make -j 4 CC=icc CFLAGS="-O3 -xavx" CXX=icpc CXXFLAGS="-O3
-xavx"  FC=ifort FFLAGS="-O3 -xavx" USE_MKL=1 USE_INTEL_IMF=1
```

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。

ここでは、-j 4 を渡して 4 つのスレッドでビルドを実行していることに注意してください。この値は、システムで利用可能なスレッド数に応じて変更してかまいません。

インテル® MKL 対応の Julia ソースをリビルドするには、Julia の /deps フォルダーから OpenBLAS、ARPACK、SuiteSparse の依存関係を削除して、次のコマンドを実行します。

```
$make cleanall testall
```

インテル® ソフトウェア・ツールに対応した Julia のパフォーマンスを検証するには、Julia/test/perf/blas と Julia/test/perf/lapack フォルダーに含まれている各種パフォーマンス・テストを実行します。

## まとめ

Python*、R、Julia は、高速なプログラミングと可読性に優れたコードの開発を可能にするポピュラーな言語です。HPC、アナリティクス、科学技術計算分野の開発者は、これらの言語をインテル® ソフトウェア・ツールを使用してビルドすることで、最新のインテル® アーキテクチャーの機能を活用することができます。システム上のコアのすべての機能を利用するには、インテル® コンパイラーのベクトル化オプションと OpenMP* のベクトル化サポートにより、SIMD/AVX ベクトル命令を活用します。コンパイラーは、プロシージャー間の最適化やその他のパフォーマンス最適化機能によって、複数ファイルの最適化を行います。インテル® MKL ライブラリーは、優れたアルゴリズム、ベクトル化、OpenMP* スレッド化を通じて最新のインテル® アーキテクチャー向けに最適化された、ハイパフォーマンスな業界標準の数値演算ライブラリーです。条件付き数値再現性機能を用いると、再現可能な結果を得ることもできます。インテル® コンパイラーを使用して簡単な作業を行うだけで、ハイパフォーマンスなインテル® MKL 対応の Python*、R、Julia をビルドしインストールすることができます。システムにインテル® Xeon Phi™ コプロセッサーが搭載されている場合、インテル® MKL のホスト・プロセッサーとコプロセッサー間で計算を分割する機能が自動的に有効になり、パフォーマンスがさらに向上するでしょう。

インテル® コンパイラーを使用して簡単な作業を行うだけで、ハイパフォーマンスなインテル® マス・カーネル・ライブラリー（インテル® MKL）対応の Python*、R、Julia をビルドしインストールすることができます。

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。

# 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。

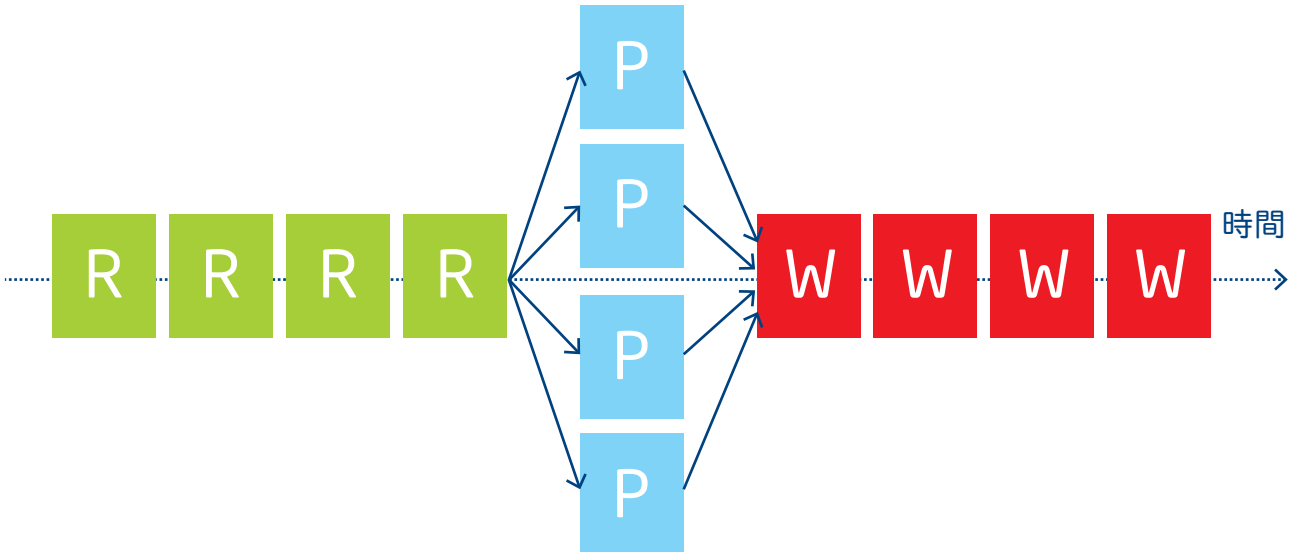
インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

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**）。

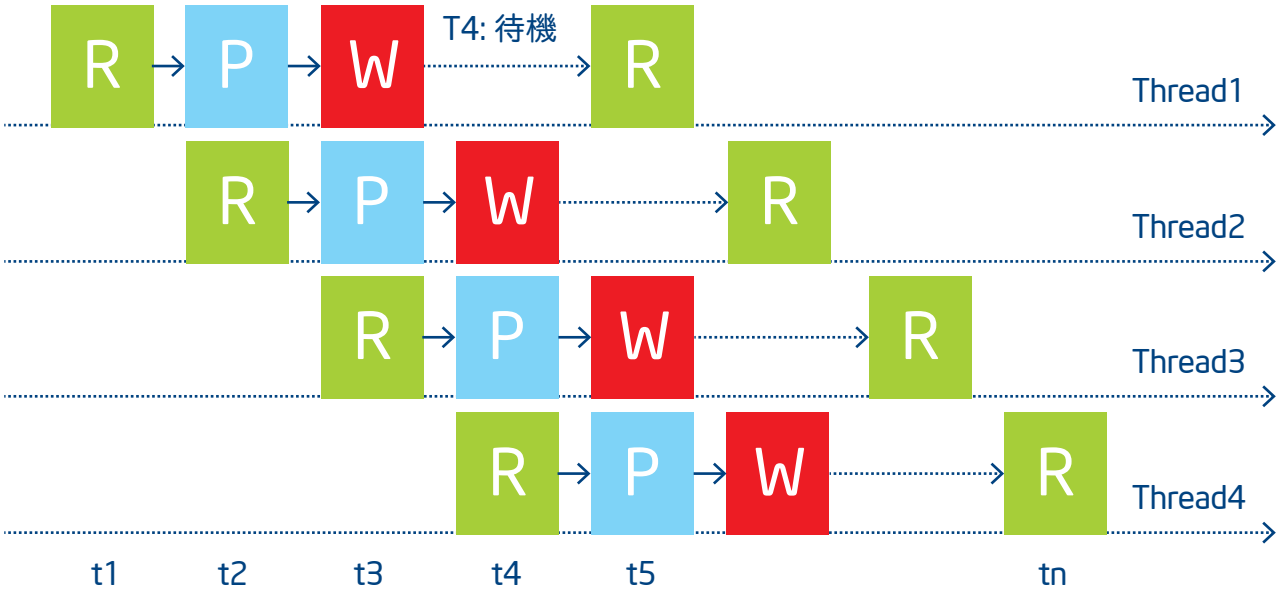



2

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。

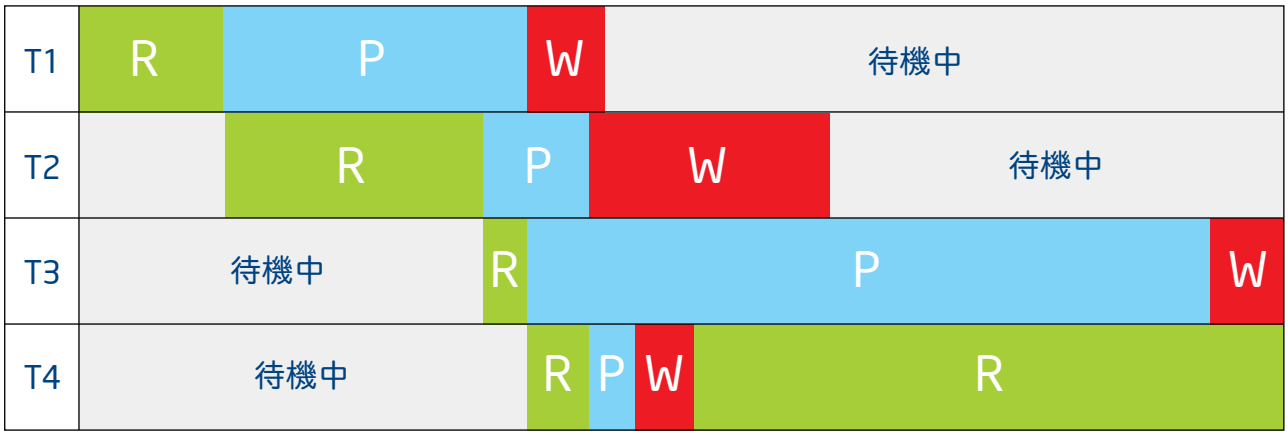



3

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)
```

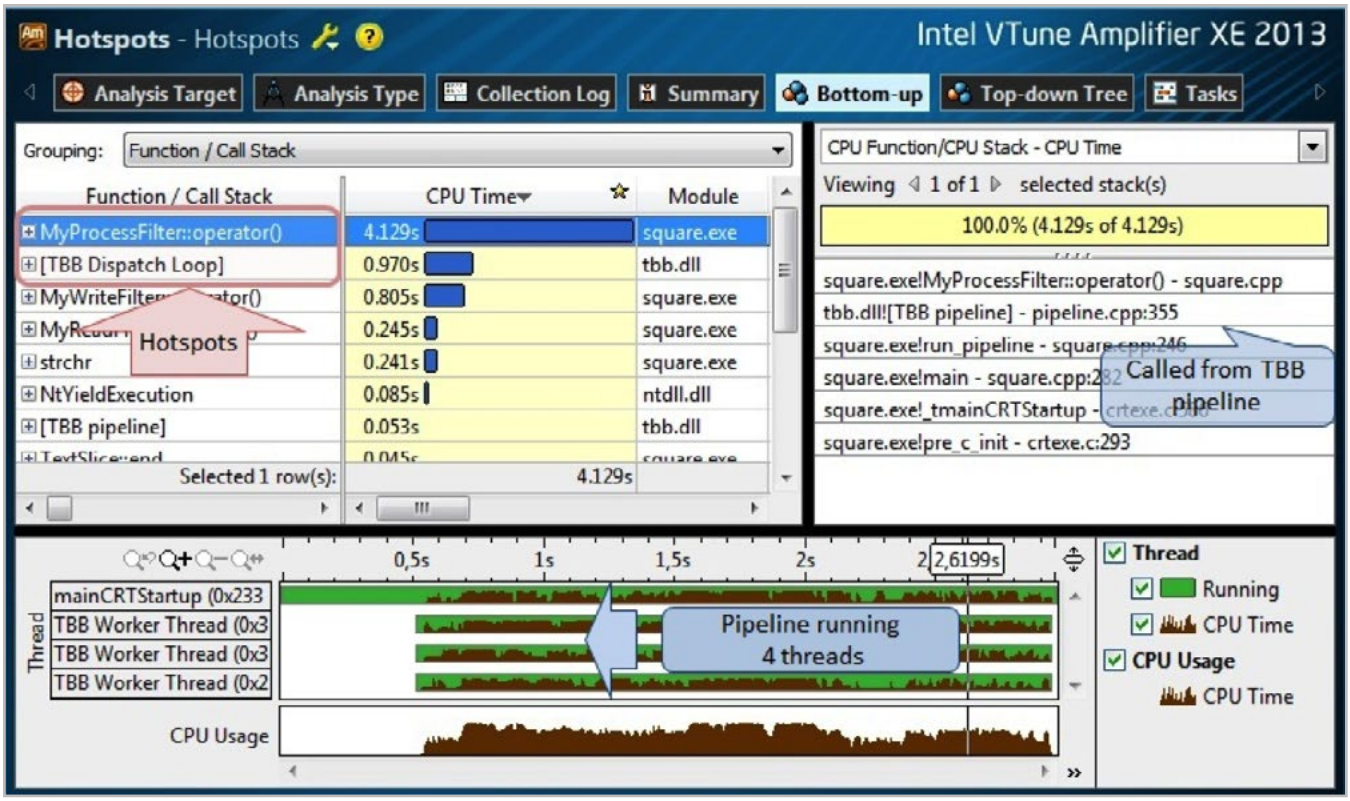
これで、プロセッサー・リソースの使用率を最大限にするように、インテル® TBB ライブラリーによって処理される 3 つのパイプライン化されたタスクを作成できました。処理ステージは並列化され、残りのステージとともに、ハードウェア・スレッドの状況に応じて異なるハードウェア・スレッドに動的に再割り当てされます。スレッド管理について考慮する必要はありません。注意が必要なのは、パイプラインのタスクを適切に作成してリンクすることだけです。

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。



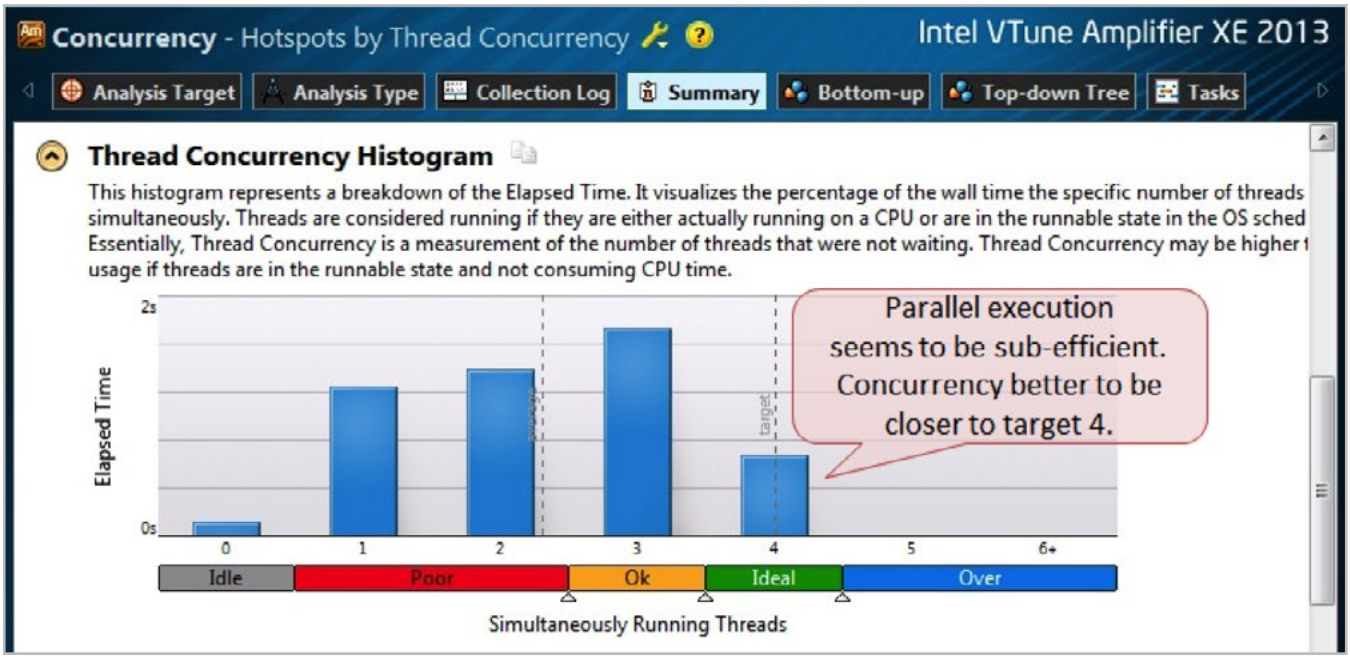

5

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。

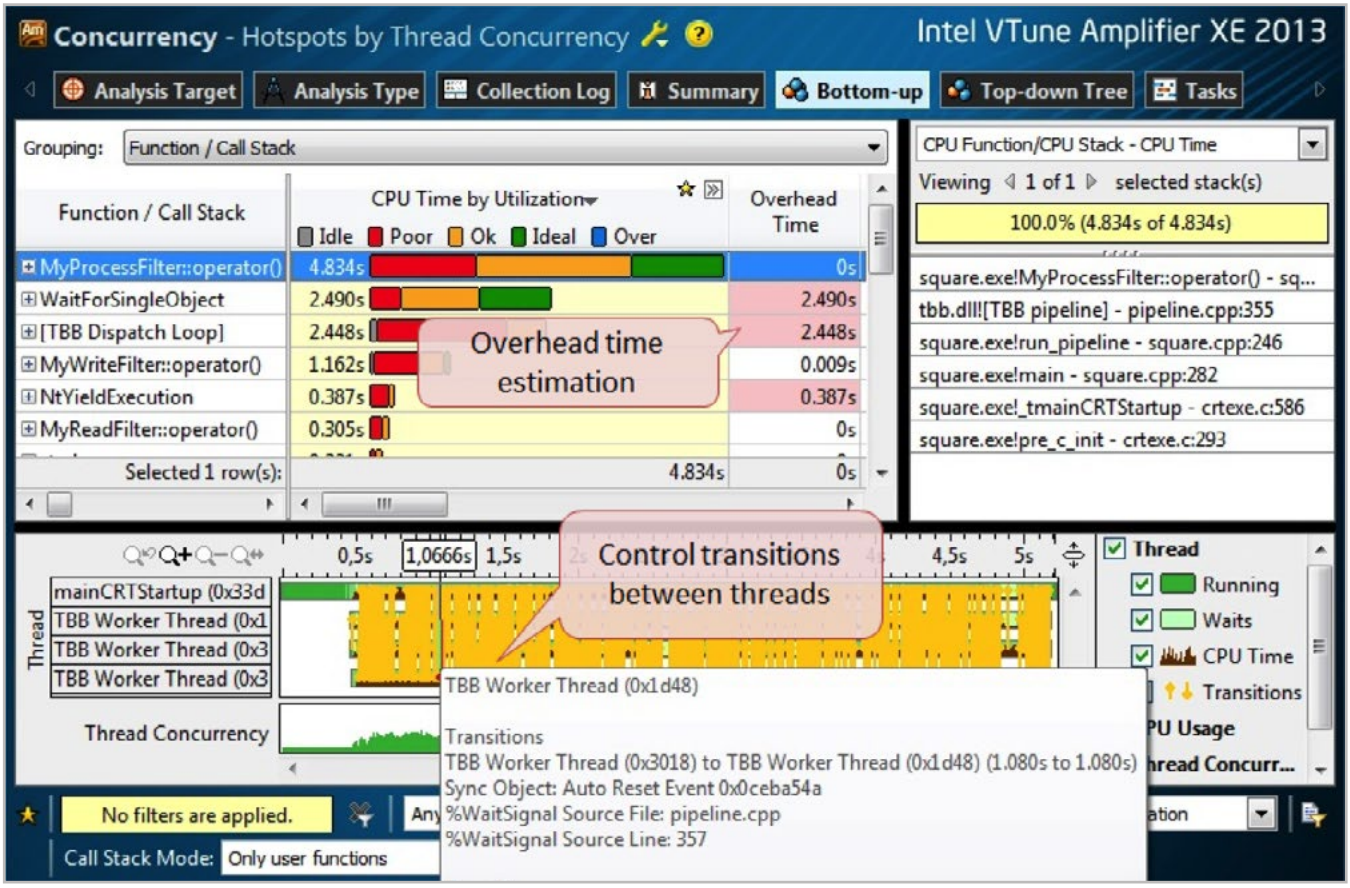
コンカレンシー解析の結果から、この並列実行は効率的ではなく、実行時間のほとんどで 4 スレッドが同時に実行されていないことが分かりました (**図 6**)。(注 : 青いコンカレンシー・レベルのバーは、スレッドが並列に実行された時間を示しています。コンカレンシー・レベル 4 にのみバーが表示される、つまり、4 つのコアの 4 つのスレッドでアプリケーション実行時間がすべて実行されることが理想的です。)

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。

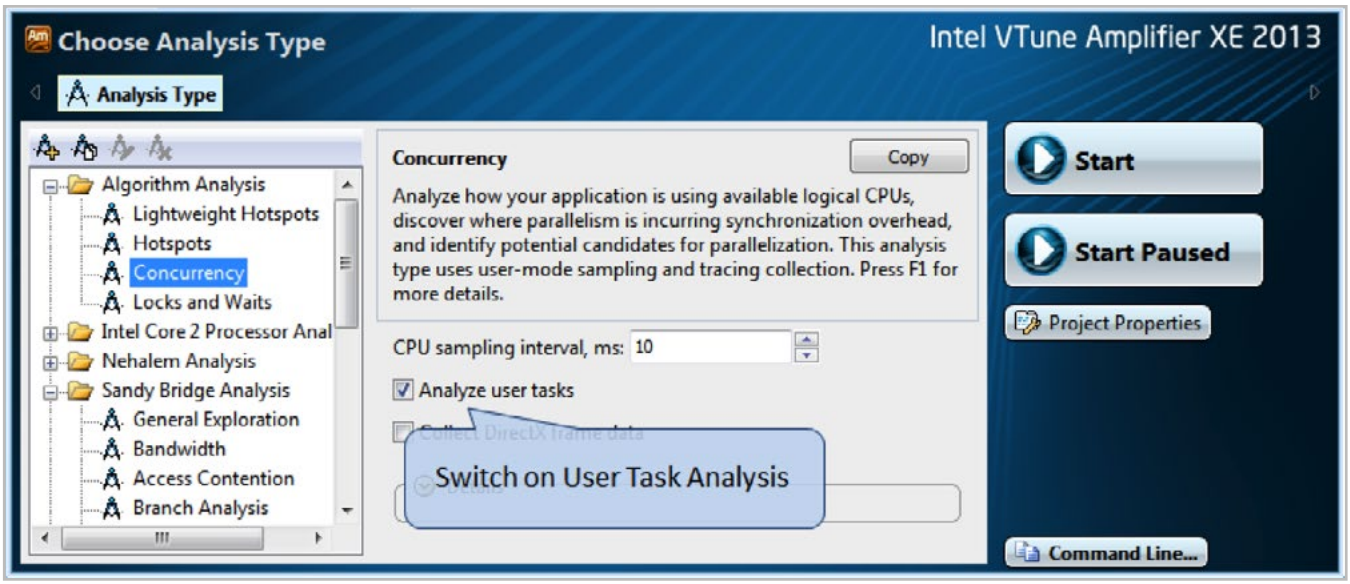
処理ステージは次のようになります (API 呼び出しに関する詳細は、製品のドキュメントを参照してください)。

```
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);
}
```

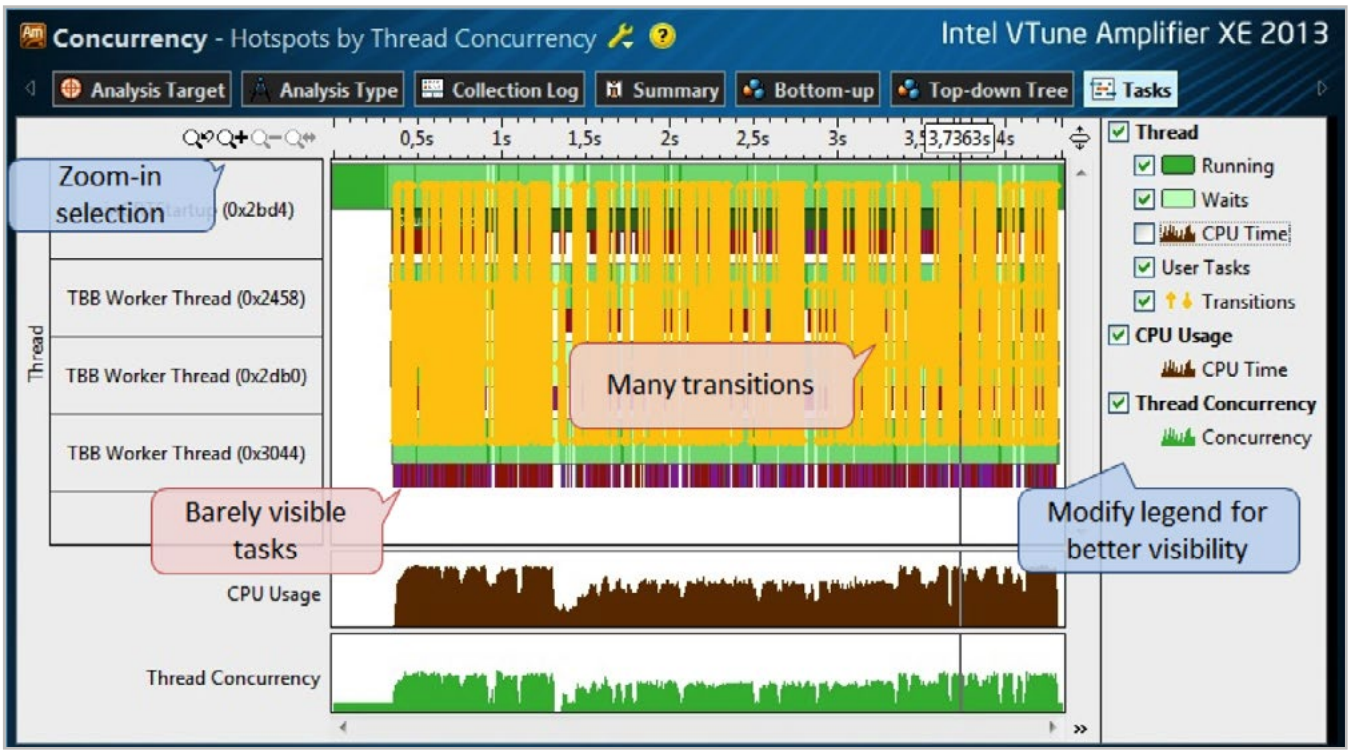
5. プロジェクトにインテル® VTune™ Amplifier XE ヘッダーのパスを追加します。
   `$(VTUNE_AMPLIFIER_XE_2013_DIR)`
6. システムの環境に応じて、パス `$(VTUNE_AMPLIFIER_XE_2013_DIR)lib[32|64]` にある *libittnotify.lib* を静的にリンクします。

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

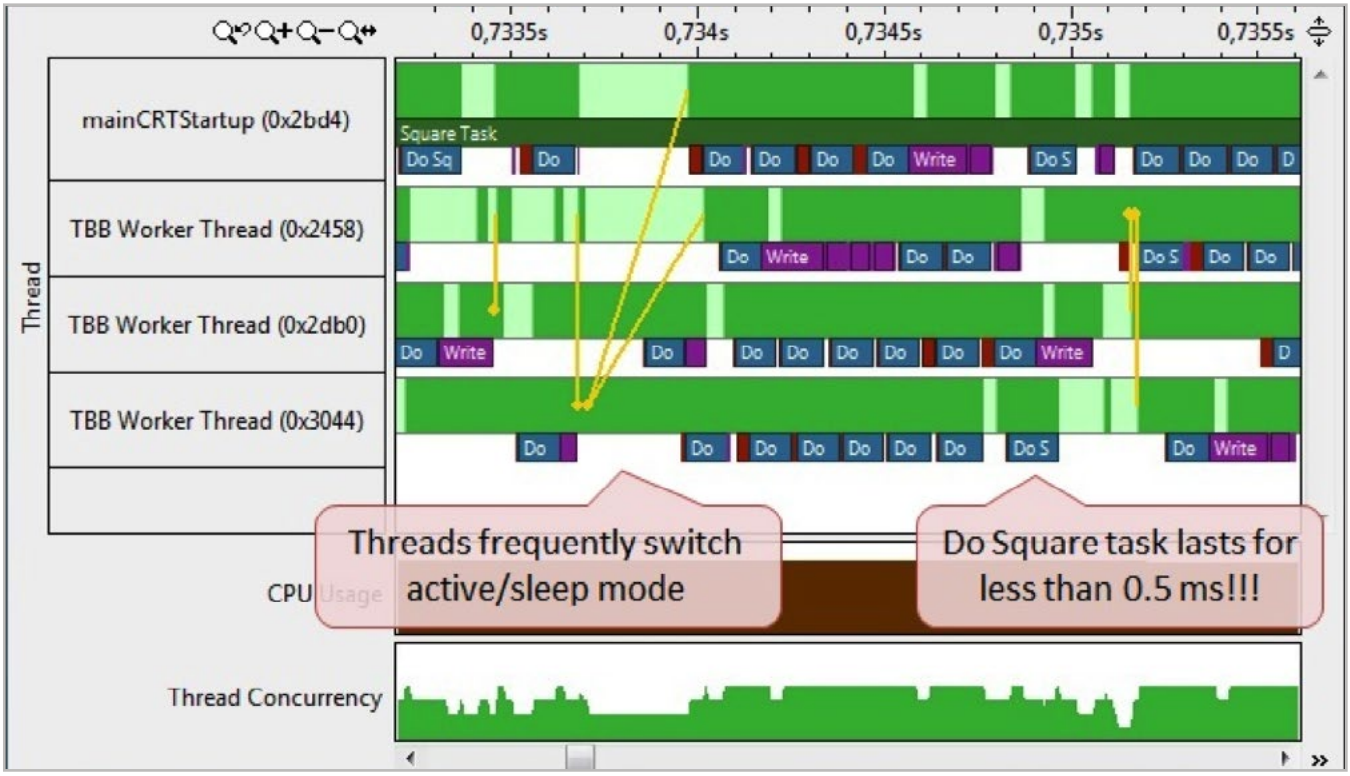



9

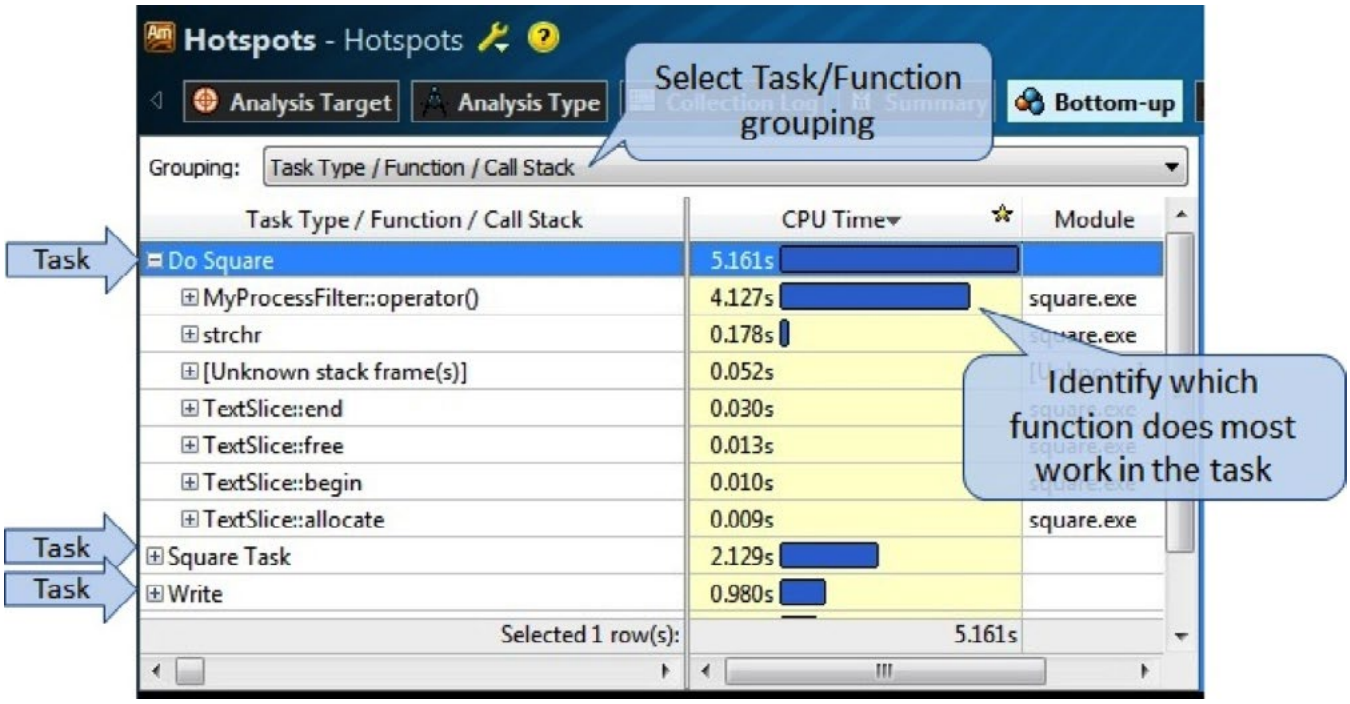
タイムラインを拡大したところ、2 つの問題が明らかになりました（**図 10**）。1 つ目は、タスク間に大きなギャップがあり、その間スレッドはアクティブではない（同期のために待機している）ことです。2 つ目は、タスク時間で、「Do Square」タスクは約 0.1 ミリ秒しか実行されていないことです。この期間はタスクがインテル® TBB のタスク・スケジューラーで制御されることを考慮すると非常に短く、スケジューラーが頻繁にタスクを制御しなければならないため、オーバーヘッドが大きくなります。解決方法は、各タスクのワーク量を増やしてタスク管理のオーバーヘッドを減らすことです。

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。



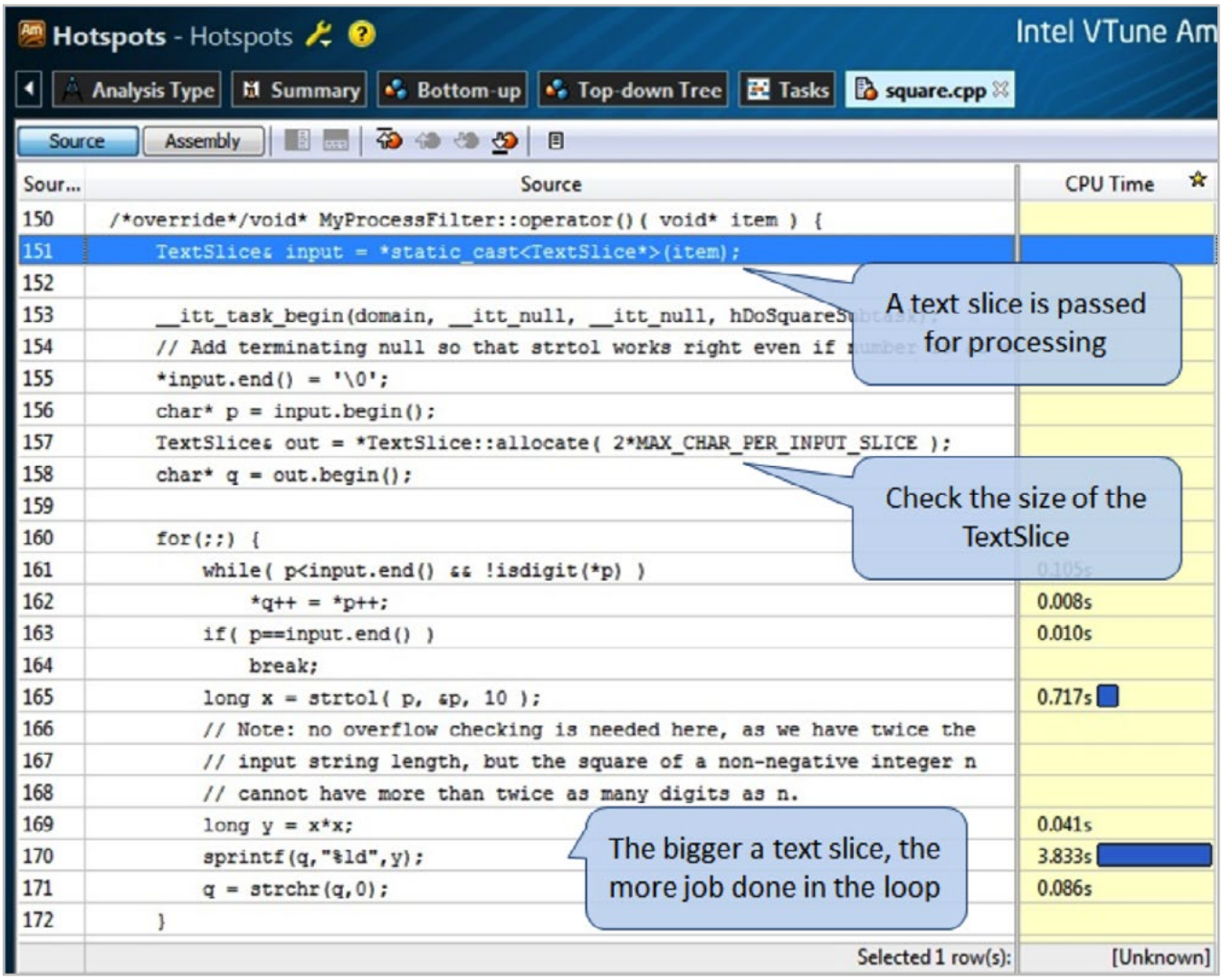

10




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。

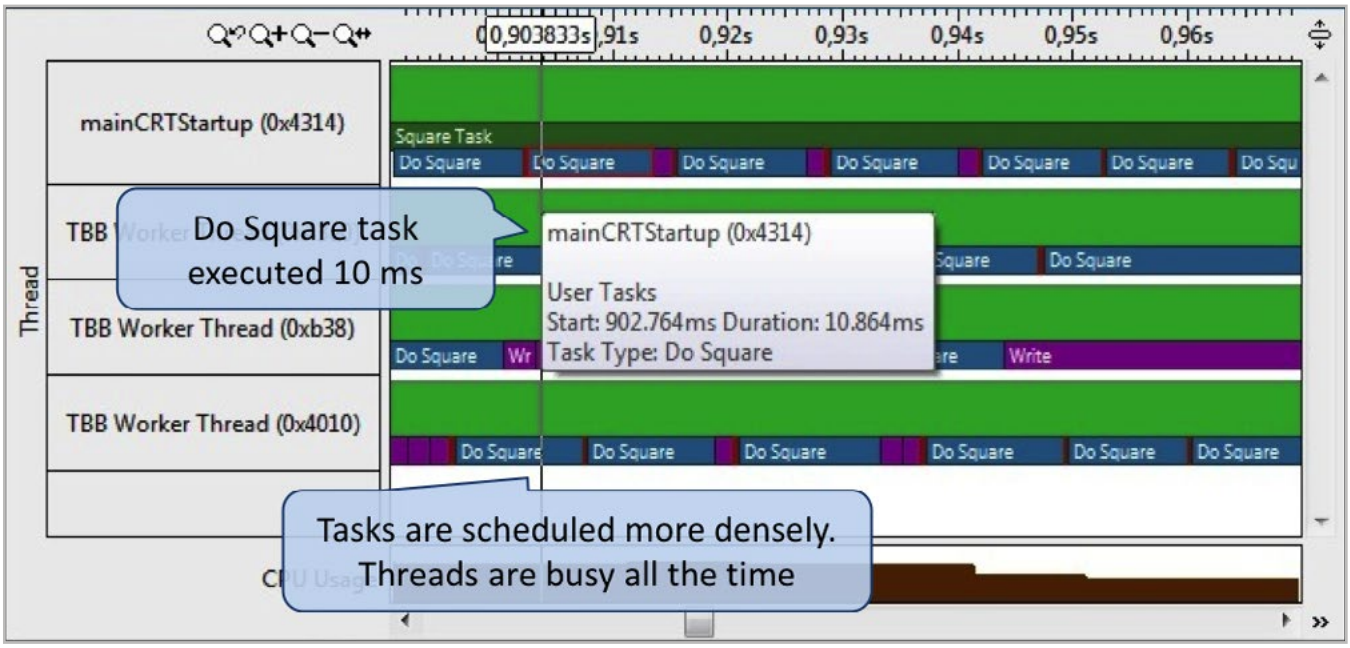
次に、*MyProcessFilter::operator()* 関数のソースコードにドリルダウンすると、関数にテキストスライス項目が渡されていることが分かります (**図 12**)。このテキストデータを反復して、各文字列を long 型の値に変換し、値を乗算して二乗の値を取得した後、値を文字列に変換しています。この関数のワークロードを増やす最も簡単な方法は、テキストスライスのサイズを増やして、処理操作の数を増やすことです。ここでは、新しいサイズの制限として、(タスク実行タイミングに基づいて) `MAX_CHAR_PER_INPUT_SLICE` にオリジナルの 100 倍の値を選択しました。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

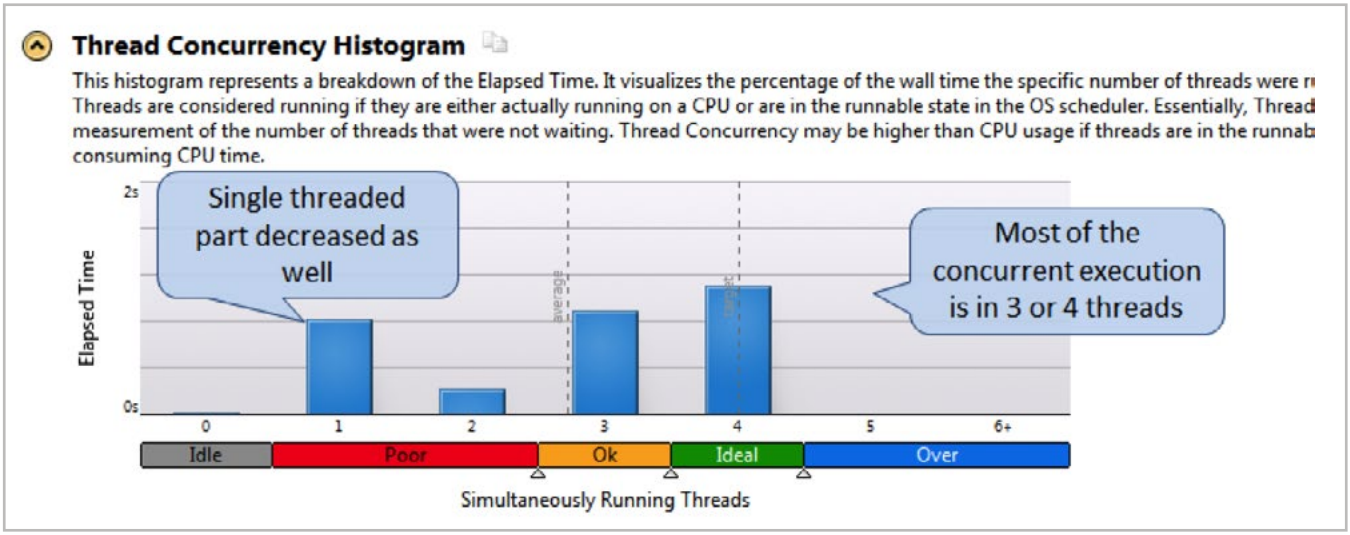



13

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。

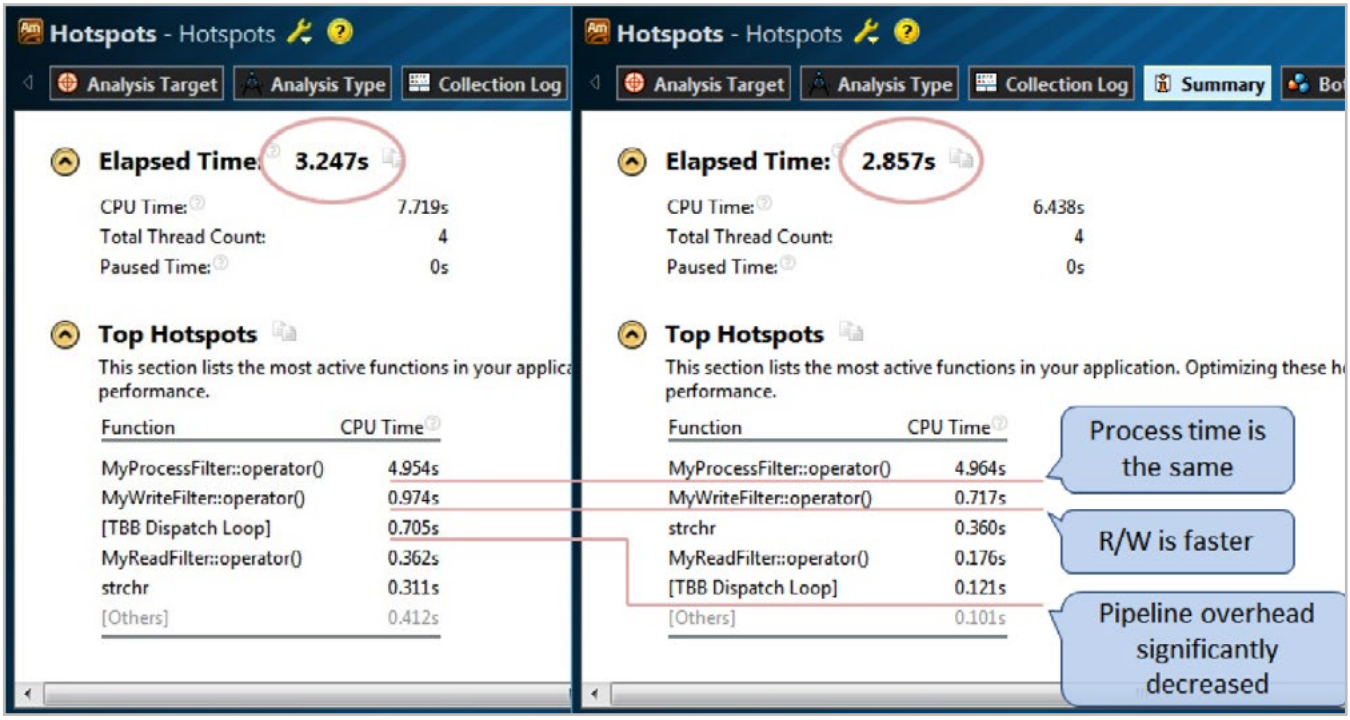
単一スレッドで実行されている部分もまだ残っていますが、オリジナルの値よりもかなり少なくなりました。つまり、プログラムのシリアル実行部分はまだ存在しますが、パイプラインにより並列性が向上したことになります（タイムライン・ビューの始めで確認できます）。シリアル実行部分を選択して表示すると、アプリケーションのメインスレッドの初期化フェーズであることが分かります。アプリケーションの初期化を行うコード部分が明確に識別できる場合は、ポーズ状態で開始する API を用いて、アプリケーションが実際のワークを行う時点でレジューム API を使用して収集を再開すると良いでしょう。

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"+
```

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。

# スレッド・パフォーマンスをスピードアップ：インテル® C++ コンパイラーによるインテル® TSX のサポート

**Anoop Madhusoodhanan Prabha**
インテル コーポレーション
ソフトウェア・エンジニア

## はじめに

プロセッサーは元々シングルコアでしたが、この 10 年の間に、インテルや他のプロセッサー・メーカーによってマルチコア（複数の処理ユニット）を搭載するプロセッサーが開発されました。このシングルコアからマルチコアへプロセッサーの設計トレンドが移行したことにより、マルチスレッド・プログラミングはますます重要になりました。アプリケーションはワークロードを複数のスレッドに分散することで、ターゲットマシン上のすべてのコアを活用することができます。

アプリケーションは、複数のスレッド間でデータセットを互いに排他的な方法で分割できますが、単純なリダクション変数やヒストグラム（配列またはハッシュテーブル）など、少なくとも 1 つのリソースはアプリケーションの全スレッドで共有されます。この共有リソース（クリティカル・リソース）は、データ競合を回避するため、クリティカル・セクション内で保護しなければなりません。これまでクリティカル・セクションのコードは、ロックによりアクセスを制御することで、シリアルに実行されてきました。クリティカル・セクションを実行するスレッドは、ロックを取得する必要があり、取得にはロック・オブジェクトに対する読み取りおよび書き込み操作が伴います。例えば、スレッド A がロックを取得している場合、そのロックで保護されているクリティカル・セクションを実行したい別のスレッドは、ロックが解放されるまで待つ必要があります。このアプローチは、共有リソースのどの場所にアクセスする場合でも、1 つのロックによってその共有リソース全体へのアクセスが保護されるため、粗粒度のロックと呼ばれます。

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。

別のアプローチとしては、細粒度のロックがあります。このアプローチでは、共有リソースがより小さなチャンクに分けられ、それぞれのチャンクへのアクセスは個別のロックによって保護されます。粗粒度のロックと異なり、このアプローチには 2 つのスレッドが共有リソースの別々のチャンクを同時に更新できるという利点があります。ただし、更新する共有リソースのチャンクに基づいて、取得するロックを特定するコードロジックを追加しなければなりません。

さらにほかのソリューションとして、インテル® トランザクショナル・シンクロナイゼーション・エクステンション（インテル® TSX）があります。インテル® TSX は、ハードウェア・レイヤー（L1 キャッシュ）で実装されるトランザクショナル・メモリー・ソリューションです。プログラムでこのハードウェア機能を利用できるように、2 つのソフトウェア・インターフェイスが用意されています。このソフトウェア・インターフェイスは、インテル® C++ Composer XE パッケージの一部として提供されており、構文は粗粒度のロックと同じですが、細粒度のロックのように振る舞います（チャンクサイズは L1 キャッシュラインのサイズと同じになります）。

## インテル® TSX

インテル® TSX は、ロックベースのプログラミング・モデルを維持しつつ、ロックで保護されたクリティカル・セクションのパフォーマンスを向上させることを目的としています。プロセッサーは、クリティカル・セクションへのアクセスをシリアル化する必要があるかどうかを動的に判断できます。インテル® TSX では、ロックを取得する代わりに無効化します（ロック・オブジェクトに対して読み取りおよび監視操作のみを行い、書き込み操作は行いません）。ロックが取得されないため、ほかのスレッドが同じクリティカル・セクションを実行できるようになり、並行性がもたらされます。クリティカル・セクションは、トランザクション領域として定義されます。トランザクション領域内のメモリー操作は、トランザクション実行が成功した場合のみコミットされます。

## トランザクション実行の成功と失敗

クリティカル・セクションのロックを無効化する 2 つ以上のスレッドが、同じ L1 キャッシュライン内の共有リソースを更新しない場合、トランザクション実行は成功します。各読み取り / 書き込みメモリーアドレスは、インテル® TSX によって、キャッシュラインの粒度で読み取りセットおよび書き込みセットに保持されます。トランザクション実行に失敗すると、トランザクション・アボートになります。

トランザクション・アボートはさまざま原因により生じます。

- 別の論理プロセッサーがトランザクション領域の書き込みセットに含まれる場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットに含まれる場所で書き込みを行うと、データ競合が発生します。インテル® TSX はキャッシュライン単位でデータ競合を検出するため、同じキャッシュラインに配置されたデータは無関係であっても競合として認識されます。
- トランザクション・アボートは、トランザクション・リソースの制限により発生することもあります。例えば、共有リソースが大きすぎて L1 キャッシュに収まらない場合、L1 データ・キャッシュ・ミスになります。
- トランザクション領域内での CPUID（プロセッサー識別命令）の使用は、シリアル化によるトランザクション・アボートを引き起こします。
- トランザクション実行中に生じる割り込みは、トランザクション・アボートを引き起こします。

トランザクション・アボートの詳細と、トランザクション・アボートを最小限に抑える方法については、『**インテル® 64 アーキテクチャーおよび IA-32 アーキテクチャー最適化リファレンス・マニュアル**』の 12.2.3 節を参照してください。

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。

# インテル® TSX のソフトウェア・インターフェイス

次の 2 つのソフトウェア・インターフェイスでインテル® TSX 機能を使用することができます。

- **Hardware Lock Elision (HLE):** 従来のプロセッサーと互換性のある命令セット拡張で、命令プリフィクスとして XACQUIRE と XRELEASE を使用します。このインターフェイスは、従来のハードウェアでインテル® TSX を利用したコードを実行する際に使います。トランザクション実行と非トランザクション実行のコードパスは同じです。インテル® TSX モードでは、プログラムは最初にトランザクション実行を試みて、成功した場合はメモリー操作をアトミックにコミットします。トランザクションがアボートした場合は、バッファーですべてのメモリー操作がロールバックされ、そのトランザクション領域は非トランザクション・モードで（ロックを無効化する代わりに取得して）実行されます。インテル® C++ コンパイラーの **HLE 組込み関数**を用いて、スレッド・ライブラリーでロックの無効化ロジックを実装できます。
- **Restricted Transactional Memory (RTM):** 第 4 世代インテル® Core™ プロセッサーでサポートされた新しい命令セットです。プロセッサーがこの命令セットをサポートしているかどうかは、http://ark.intel.com（英語）で確認できます（**図 1**）。

  この命令セットには、XBEGIN、XEND、XABORT、XTEST 命令があります。HLE インターフェイスと違い、RTM インターフェイスではトランザクション領域と非トランザクション領域（フォールバック・パス）が異なります。トランザクションがアボートすると、バッファーですべてのメモリー操作がロールバックされ、実行がフォールバック・パスに移り、非トランザクション・モードで実行されます。インテル® C++ コンパイラーの **RTM 組込み関数**を用いて、プログラムでインテル® TSX を有効にすることができます。

| | |
|---|---|
| Intel® TSX-NI | Yes |

1

インテル® TSX のソフトウェア・インターフェイスについては、『Intel® Architecture Instruction Set Extensions Programming Reference』（英語）の第 12 章で詳しく説明されています。

# インテル® TSX 対応ロック

- **GNU* C ライブラリーの pthread* ライブラリー：**ロックの無効化コードパスでは、glibc POSIX* スレッドのミューテックスと rwlock を使用することができます。pthread* ロックを使うアプリケーションは、glibc の新しいバージョン（glibcrtm-2.17 branch）とリンクして環境変数を設定すると、自動的にインテル® TSX 対応になります。インテル® TSX を有効にし、アプリケーションで利用する方法については、「**GNU* C ライブラリーでのロックの無効化**」（英語）で詳しく説明されています。
- **インテル® C++ コンパイラーに含まれる OpenMP* ライブラリー：**これまで、OpenMP* プログラムでクリティカル・セクションを保護するには、#pragma critical { } ブロックまたは OpenMP* ロック（omp_set_lock()/omp_unset_lock()）が使われていました。OpenMP* ロックもインテル® TSX のコードパスで利用できます。OpenMP* ロックを使用するアプリケーションは、環境変数 **KMP_LOCK_KIND=adaptive** を設定することで、コードを変更しなくてもトランザクション・モードで実行することができます。

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。

## RTM 組込み命令によるインテル® TSX の有効化

簡単なヒストグラムを例に、RTM 組込み関数を用いてインテル® TSX を有効にする方法を説明します。このプログラムは、.bmp（RGB）形式の入力イメージを読み取り、0 ～ 255 の範囲内の青の画素値の分布を追跡するヒストグラムを作成します。データセット（イメージペイロードまたはデータ）は、複数の OpenMP* スレッド間で互いに排他的な方法で分割されますが、ヒストグラム（histogramblue）はトランザクション領域で使用される共有リソースです。**図 2** のサンプルコードで、"TSX" マクロはインテル® TSX コードパスを有効にします。トランザクション領域は、_xbegin() コンパイラー組込み関数の呼び出しで始まり、_xend() コンパイラー組込み関数の呼び出しで終わります。_xbegin() と _xend() で囲まれたコードがトランザクション・モードで実行されます。この例では、トランザクション領域でヒストグラムの場所にアクセスし、1 だけインクリメントします。_xbegin() の呼び出しが _XBEGIN_STARTED を返した場合は、実行がトランザクション・モードに移ります。そうでない場合は、フォールバック・パスに移り、（トランザクション・アボートによりロックが取得されると）非トランザクション・モードで実行されます。

**ここで紹介したインテル® TSX を有効にするサンプルコードは、こちらからダウンロード**できます。

```
104 #pragma omp parallel for shared(in, histogramblue) schedule(dynamic, 5000)
105     for(int i = 0; i < numofdatapoints; i+=3)
106     {
107         int j = i+2;
108 #ifdef TSX
109         unsigned int status = _xbegin();
110         if(status == _XBEGIN_STARTED){
111             histogramblue[in[j]]++;
112             _xend();
113         } else {
114             #pragma omp atomic
115             histogramblue[in[j]]++;
116         }
117 #else
118 #pragma omp atomic
119             histogramblue[in[j]]++;
120 #endif
```

2

# インテル® C++ コンパイラーに含まれる OpenMP* ライブラリーによるインテル® TSX の有効化

同じヒストグラム・プログラムを、インテル® TSX 対応の OpenMP* ロックを使って記述することができます。インテル® C++ コンパイラー 14.0 以降には、インテル® TSX 対応の OpenMP* ライブラリーが含まれています。操作手順は次のとおりです。

1. 以下のようにコードを変更します。

   次のコードは、OpenMP* ロックの定義と初期化を行います。

```
48    omp_lock_t writelock;
49    omp_init_lock(&writelock);
```

   図 2 の OpenMP* の parallel for ブロックを以下のコードに置き換えます。

```
104 #pragma omp parallel for shared(in, histogramblue) schedule(dynamic, 5000)
105     for(int i = 0; i < numofdatapoints; i+=3)
106     {
107         int j = i+2;
108         omp_set_lock(&writelock);
109         histogramblue[in[j]]++;
110         omp_unset_lock(&writelock);
111     }
```

   最後に、次のように OpenMP* ロックを破棄します。

```
122     omp_destroy_lock(&writelock);
```

2. 次に示すように、環境変数 KMP_LOCK_KIND=adaptive を設定します。

```
KMP_LOCK_KIND    adaptive
```

3. インテル® C++ コンパイラーに含まれる OpenMP* ライブラリーにリンクします。

プログラムを通常モード（非トランザクション・モード）で実行するには、環境変数 KMP_LOCK_KIND をリセットするだけです。コードの変更は必要ありません。

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。

## 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）によるインテル® TSX の有効化

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）4.2 では、HLE 命令によりインテル® TSX を有効にする speculative_spin_mutex が追加されました。インテル® TBB の将来のリリースでは、RTM 命令とともに speculative_rw_spin_mutex がコミュニティー・プレビュー機能として追加される予定です。インテル® TBB 4.2 は、インテル® C++ Composer XE 2013 SP1 Update 2 パッケージの一部として、あるいはインテル® TBB 公式 Web サイト（英語）からスタンドアロン・ライブラリーとして利用できます。

## まとめ

インテル® TSX は、ハードウェアでトランザクション・メモリーを実装します。このインテル® TSX をサポートする 2 つのソフトウェア・インターフェイスが用意されています。HLE は従来のプロセッサー向けで、RTM は第 4 世代インテル® Core™ プロセッサー以降向けです。インテル® TSX を利用することで、実行時に競合が検出されない限り、クリティカル・セクションを並行実行し、プログラムの並行性を活かすことができます。

インテル® TSX を利用するアプリケーションの効率は、インテル® PCM ツールと Linux* perf で評価できます。これらのツールは、成功したトランザクション実行の数やトランザクション・アボートの発生回数に関する詳細情報を提供します。インテル® C++ Composer XE の評価版をダウンロードして、インテル® TSX をぜひお試しください。

インテル® TSX は、ハードウェアでトランザクション・メモリーを実装します。このインテル® TSX をサポートする 2 つのソフトウェア・インターフェイスが用意されています。HLE は従来のプロセッサー向けで、RTM は第 4 世代インテル® Core™ プロセッサー以降向けです。インテル® TSX を利用することで、実行時に競合が検出されない限り、クリティカル・セクションを並行実行し、プログラムの並行性を活かすことができます。

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。

# Tachyon レイトレーサー：インテル® Xeon Phi™ コプロセッサーへの移植

**Roman Lygin** インテル コーポレーション シニア・テクニカル・サポート・エンジニア
**Dmitry Durnov** インテル コーポレーション ソフトウェア開発エンジニア

## はじめに

インテル® Xeon Phi™ コプロセッサーの登場は、開発コミュニティーにおいて強い関心を集めました。しかし、その計算能力を引き出すためには、多くの場合アプリケーションの調整が必要になります。インテル® Xeon Phi™ コプロセッサー・アーキテクチャーの大きな利点は、既存のプログラミング・モデルとツールを使用して、CPU とコプロセッサーで 1 つのソースベースを維持できることです。

この記事では、オープンソースのレイトレーサー[1] であり、Spec MPI* スイートに含まれる Tachyon を、インテル® Xeon Phi™ コプロセッサーへ移植する実用的なケーススタディーを紹介します。この移植プロジェクトの目的は、最良のパフォーマンスを引き出すことではなく、開発ツール、動機、コードの最適化に重点を置いてワークフローの概要を説明することです。実際のプロジェクトにはさまざまな制約があり、特に時間と労力に関する制約はつきものです。ここでは、実際のプロジェクトを想定して、大幅なコードの書き直しを避け、できるだけ労力を費やさないようにしました。

開発コミュニティーでは、すでにほかのレイトレーサーをインテル® Xeon Phi™ プロセッサーへ移植する試みがいくつか行われています。[2,3] また、インテル コーポレーションの研究部門であるインテル・ラボでは、インテル® Xeon Phi™ コプロセッサーを含むインテル® アーキテクチャー向けに高度に最適化されたレイトレーシング用の計算カーネル群 Embree[4] の開発が進められています。

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。

## Tachyon

Tachyon は、ファイルからシーンの定義を読み取ります。シーンには光源と 3D モデルが含まれており、3D モデルは、幾何プリミティブ（三角形、球体など）と光反射の特性を決定する関連マテリアルのコレクションとして定義されています。Tachyon は、シーンを動き回るカメラのパスも読み取ります。

データを読み取ると、Tachyon アルゴリズムは、カメラ位置ごとに RGB 形式の NxM ピクセルのイメージとしてフレームを計算します。各ピクセルの色は、光線と 3D モデルの最初の交点を使用し（向きはピクセル座標とカメラ位置に依存します）、さらにパラメーター化されたシェーダーで反射光線を処理して計算されます。

1 サンプル 3D モデル（ティーポット）で計算された 1 フレーム

**図 1** は、サンプル 3D モデル（ティーポット）で計算された 1 フレームです。

アルゴリズムのパフォーマンスは FPS（frames per second）で測定されます。Tachyon は、即時レンダリングとスループット計算（生成されるイメージをファイルに出力する場合）のどちらにも使用できるため、このケーススタディーでは FPS を向上させ、妥当なインタラクティブ FPS のしきい値を超えるようにしました。

Tachyon は、MPI と OpenMP* によるハイブリッド並列処理をサポートしています。各フレームごとに P 個の MPI ランク間でワークが等しく分配され、各 MPI ランクはフレームの 1/P を計算します（厳密には、各ランクは M/P フレーム行をラウンドロビン方式で計算します。M はフレームの高さを表します）。マスターランク 0 は、ほかのランクと同じ回数の計算を実行し、ワーカーから計算済みのチャンクを収集して、バッファーの後処理（画面への表示やファイル出力など）を行います。MPI ランクに委任されたフレームチャンク行を処理する並列ループの実行には、OpenMP* スレッドが使用されます。

各フレームの前に同期処理（MPI_BARRIER など）はありませんが、頻繁な通信によって通信チャネルが飽和し、ワーカーは次のフレームへ進む前にマスターを待機するようになります。そのため、このアルゴリズムではスケーラビリティーが制限される傾向にあります。マスターがボトルネックとなり（ワーカーと同じ計算量に加えて、その他の処理も行うため）、同期は暗黙に行われます。

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。

## テスト環境

テストには、4 ノードのクラスターを使用しました。各ノードは、2 つのインテル® Xeon® プロセッサー E5-2680 と 1 つのインテル® Xeon Phi™ コプロセッサー 7120 で構成されています。Tachyon は、最新バージョン 0.99 beta 6 に多少手を加えて、各計算済みフレームチャンクを 1 行ずつ送信する代わりに、各ワーカーからマスターへ 1 つのバッファーを送信するように変更しました。Tachyon コードの事前解析に基づいたこの変更により、MPI ランク間の通信回数が減り、スケーラビリティーが大幅に向上しました。

座標と法線には、単精度浮動小数点モードを使いました。テスト・ワークロードには、**図 1** の teapot.dat という名前のモデル（よく知られているユタ・ティーポット・モデル）と、カメラファイル teapot.cam を 10 回連結して十分なワークロードにしたものを使用しました。

パフォーマンスは、計算済みフレーム数を合計計算時間で割って平均 FPS を算出する、アプリケーションのメカニズムを利用して測定されました。

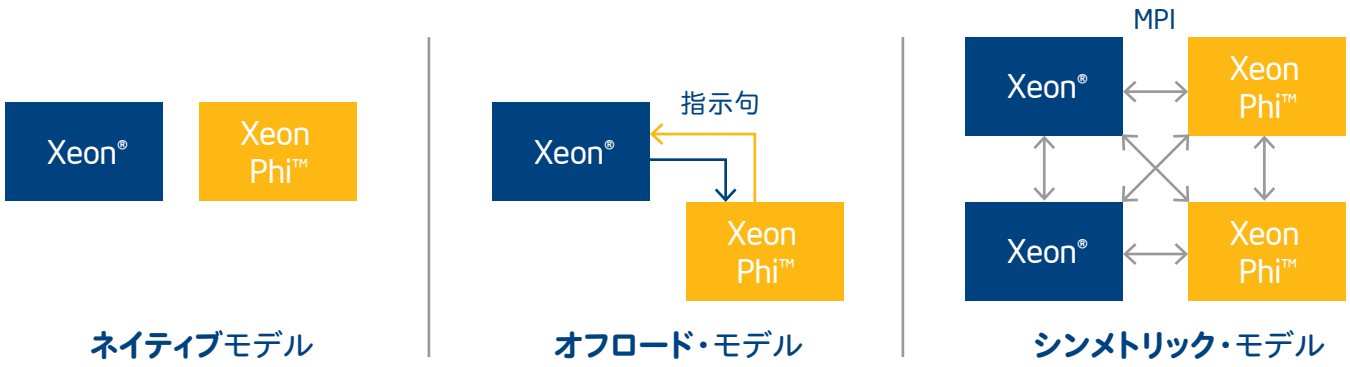
コンパイル、実行、パフォーマンス解析には、2013 年 9 月にリリースされたインテル® Cluster Studio XE 2013 SP1[5] を使用しています。インテル® Cluster Studio XE は、並列アプリケーションを効率良く作成し解析するソフトウェア・スイートです。インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー [6] での開発を支援するコンパイラー、プログラミング・ライブラリー、ツールが含まれています。

テスト環境の詳細は以下のとおりです。

- RHEL* 6.2; MPSS* 3.1.1
- インテル® Cluster Studio XE 2013 SP1
- 4 ノード（各ノードは 2 つのインテル® Xeon® プロセッサー E5-2680 と 1 つのインテル® Xeon Phi™ コプロセッサー 7120 で構成されている）

また、複雑なほかのモデルもいくつかテストしました。ここでは、3D モデルを表す最も一般的な方法として三角形を使用しましたが、ここで紹介する概念はほかのプリミティブ（例えば、球体など）にも適用できます。

インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーのヘテロジニアス実行では、事前のコード変更が不要なシンメトリック・モデル（**図 2**）を使用しました。




**2** インテル® Xeon Phi™ コプロセッサーでサポートされている実行モデル

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。

## ベースライン

Tachyon アプリケーションはインテル® Xeon Phi™ コプロセッサーへ簡単に移植できます。インテル® コンパイラーのコマンドラインに -mmic と -fp-model fast=2 オプションを追加するだけです。1 つ目のオプションは、インテル® Xeon Phi™ コプロセッサーをターゲット・プラットフォームに指定し、2 つ目のオプションは、精度よりもパフォーマンスを重視する浮動小数点計算モデル（レイトレーシングを含むさまざまなアプリケーションでは許容範囲内）を指定します。

**表 1** にベースライン・パフォーマンス（FPS）を示します。

| | **ベースライン** |
|---|---|
| インテル® Xeon® プロセッサーのみ | 141.8 |
| インテル® Xeon Phi™ コプロセッサーのみ | 38 |
| インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー | 39 |

1 初期パフォーマンス（FPS）

ヘテロジニアス実行では、インテル® Xeon® プロセッサーのみで実行した場合と比べてパフォーマンスが低下しています。この原因を詳しく解析してみましょう。

## パフォーマンス解析

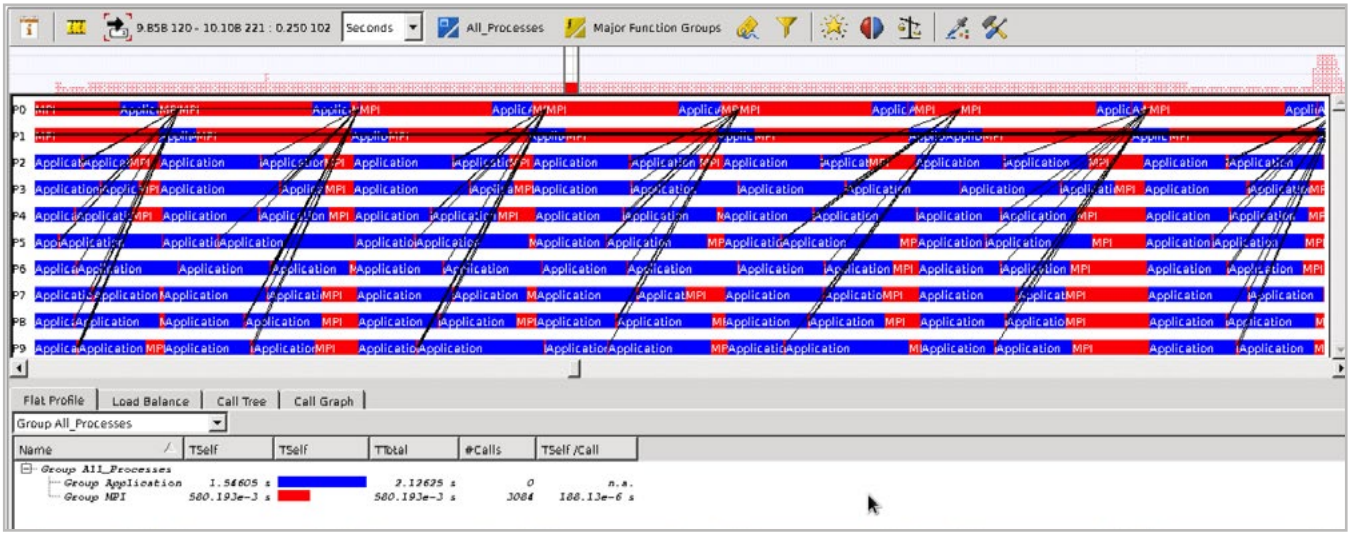
インテル® Cluster Studio XE には、共有メモリー型と分散メモリー型のどちらの並列処理にも対応するパフォーマンス解析ツールが含まれています。

- インテル® Trace Analyzer & Collector (ITAC): MPI ベースのアプリケーション用のパフォーマンス・プロファイラーと正当性チェッカー
- インテル® VTune™ Amplifier XE: シングルノード・アプリケーション用のパフォーマンス・プロファイラー

これらのツールを併用することで、ハイブリッド・アプリケーションを含む並列アプリケーションのパフォーマンス問題を正確に把握することができます。

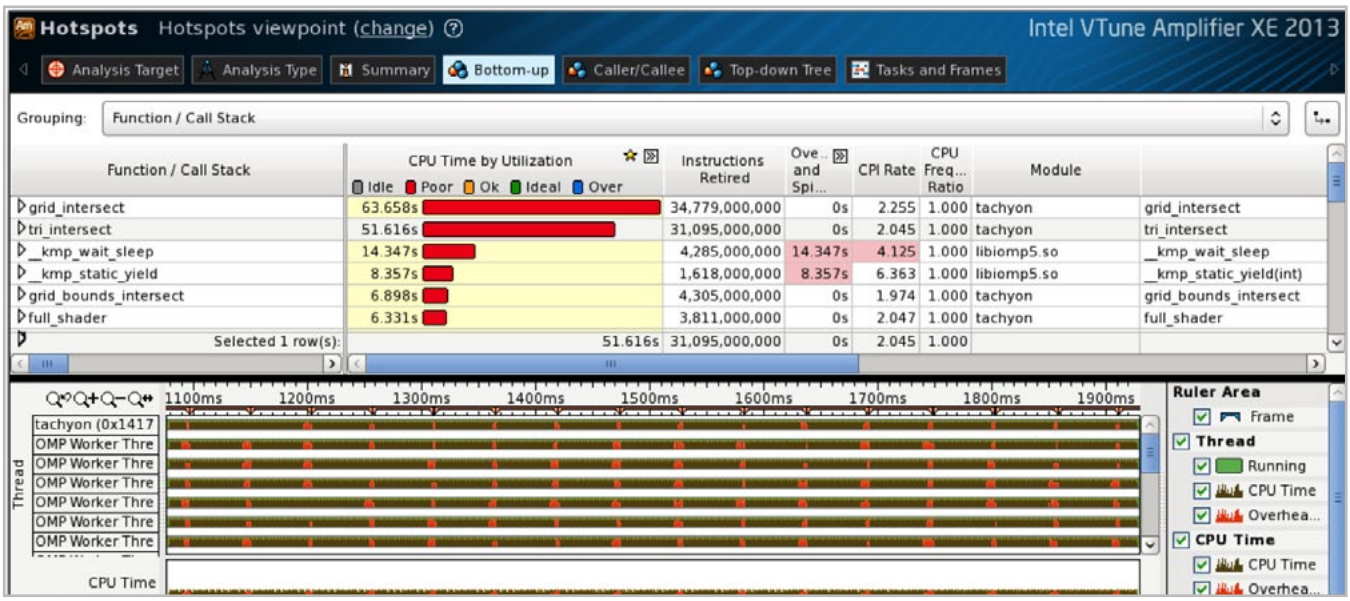
ITAC による解析から、MPI ランクが多くの時間を待機に費やしていることが明らかになりました。**図 3** に示すように、ペイロードと比べて、MPI 関数で費やされている時間（赤い部分）が非常に大きいことが分かります。これは、アルゴリズムの頻繁な通信と暗黙の同期によるもので、ワークのインバランス（同じ大きさに分割された、複雑性が異なるフレームチャンク）、およびインテル® Xeon プロセッサーとインテル® Xeon Phi™ コプロセッサーの能力の違いが原因です。ITAC のタイムライン・ビューから、インテル® Xeon® プロセッサー（一番上）のほうがこの影響を被っており、計算時間よりも待機時間のほうが多いことが分かります（各フレームの最後の赤い部分）。

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。




3 ITAC による解析結果から、フレームの計算が頻繁に同期されており（黒で示されているのがワーカーとマスター間のメッセージの送受信で、複数のフレームがほぼ同時に開始していることが見て取れます）、ワーク・インバランスによってプロセスが多くの時間を待機（赤い部分）に費やしていることが分かります。

また、インテル® VTune™ Amplifier XE の Basic Hotspots 解析から、（各 MPI プロセスの）各 OpenMP* 並列領域でも多くの時間が無駄になっていることが分かります。インテル® VTune™ Amplifier XE は、自動的にこの問題をハイライトし、グリッドビューではピンク、タイムラインでは赤で表示します（**図 4**）。インテル® VTune™ Amplifier XE は、OpenMP* 並列領域を認識し[7]、それらをタイムラインで示すことで、解析を分かりやすくしています。



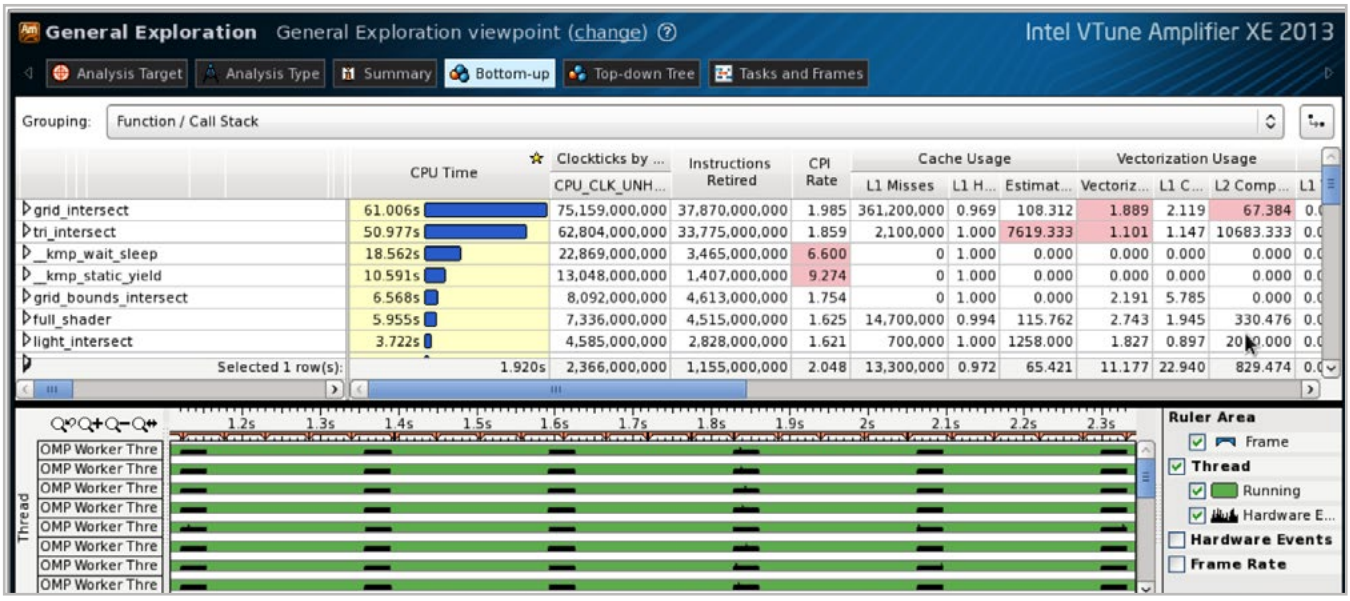

4 インテル® VTune™ Amplifier XE は OpenMP* 並列領域の無駄な時間を検出します。

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。

各 OpenMP* 並列領域の終わりにある無駄な時間も、ワーク・インバランスの問題によるものです。ワーク・インバランスは、各 MPI ランクで OpenMP* スレッドによって処理される各フレームチャンク行の複雑さが異なることが原因であると考えられます。実際、動的にバランスをとり、最小粒度を 1（1 フレームチャンク行、つまり OMP_SCHEDULE=dynamic,1）にすると最良のパフォーマンスが得られ、静的スケジューリングあるいはより大きな粒度に設定すると、ワーク・インバランスがさらに悪化しました。

ここで示す結果は、インテル® Xeon Phi™ コプロセッサー上で 61 スレッド（利用可能な 244 スレッドの 1/4）を使って、4 つの MPI プロセスを実行したものです。MPI プロセスの数を減らすと OpenMP* スレッドの数が増え、各 OpenMP* スレッドのワーク量が減るため、パフォーマンスは低下します。利用可能な並列ワーク項目の数は、フレームの行数と等しいため、これは明白です。フレームサイズが 512 x 512 で、OpenMP* スレッドの数が 61 の場合、各スレッドには平均 8 ～ 9 行 (=512/61) が割り当てられます。これは、ワークバランスを取るには十分な並列処理量ではありません。

インテル® Xeon Phi™ コプロセッサーで実行するアプリケーションを詳しく解析するため、インテル® VTune™ Amplifier XE は、コマンドライン・モードと特別なオプションを提供しています。エキスパートのアドバイスに従って [8]、このケーススタディーでは -vpu-metrics=true. オプションを使用しました。このオプションは、hotspot の関数によるベクトルレジスターの効率悪い使用を特定するのに役立ちます（**図 5**）。




5 インテル® VTune™ Amplifier XE は hotspot の関数によるベクトルレジスターの効率悪い使用を示します（ここでは 16 のうち係数 1.9 と 1.1 がハイライトされています）。

3 つの解析結果から、次のことが分かりました。

- ベクトルユニットが十分に活用されていない。
- アプリケーションに十分なスレッド並列性がないため、244 スレッドすべてが活用されていない。
- 同じワークロードの場合、MPI プロセスはインテル® Xeon® プロセッサー上よりも、インテル® Xeon Phi™ コプロセッサー上のほうが遅くなる。通常の同期では、先に完了するインテル® Xeon® プロセッサーのプロセスが、より時間のかかるインテル® Xeon Phi™ コプロセッサーのプロセスを待機しなければならず、全体のパフォーマンス低下につながる。

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。

これらの問題から、次に取るべき行動が明らかになりました。

1. MPI ランクの動的スケジューリングにより MPI プロセス間の同期を減らす。
2. 十分なスレッド並列性 (OpenMP*) を引き出す。
3. ベクトル化を利用する。

次のセクションで、これらの変更について詳しく見てみましょう。

# コードの変更

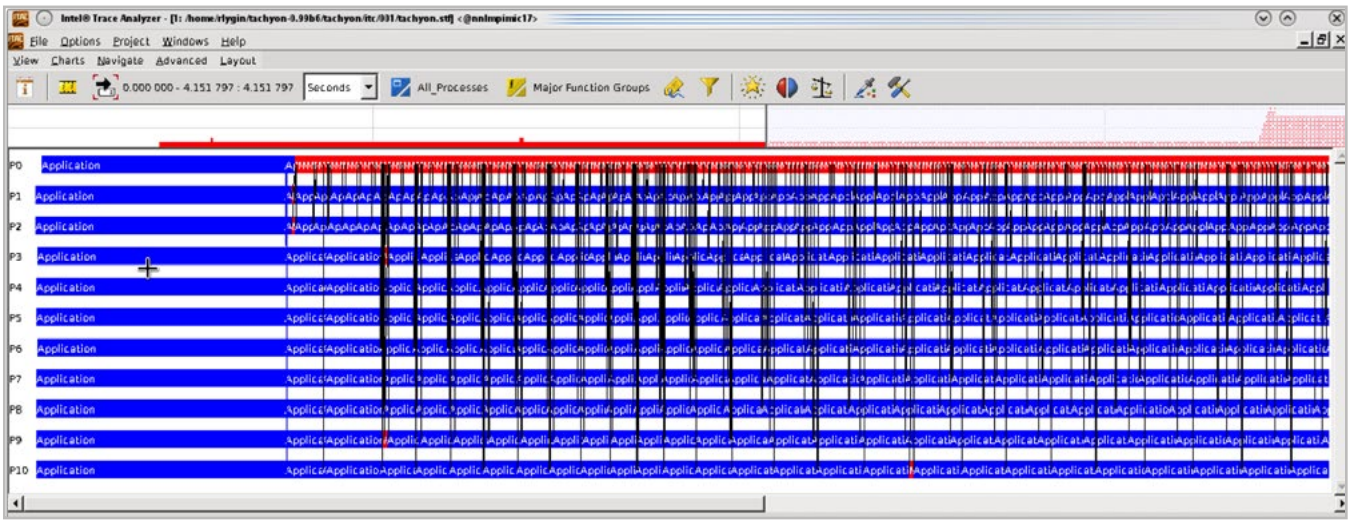
### 動的に MPI のバランスを取る

MPI ランク間の同期を減らすため、MPI ランクで 1 つずつ処理されるフレームの計算を見直し、各 MPI ランクでフレーム全体を処理するようにしました。MPI ランクは、マスターランクからフレーム番号のみを受け取り、計算を行って、計算済みのフレームを返します。そして、次に計算するフレーム番号を待機します。このスキームでは、計算済みのフレームを (画面またはファイルへ) 出力する際に、マスターによって順序が保証されるため、フレームを順番どおりに計算する必要がありません。同時に処理可能なフレームの数は、メモリー容量によって制限されます。

このスキームの実装には、約 250 行のコードの変更が必要でした。新しいスキームでは、マスターはワーカーと同じ計算ワークロードを処理せず、計算と出力の調整のみを行います。

この新しいアプローチにより、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーのプロセスの実行速度の違いをカバーすることができました。さらに、より大きなランク数でもスケーラビリティーが向上しました (インテル® Xeon® プロセッサーのプロセスのみの場合を含む)。これは、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーの両方に利点をもたらす最適化の良い例と言えるでしょう。

ITAC で再度解析を行ったところ、より構造化されてバランスの良い実行になったことを確認できました (**図 6**)。




6 新しい計算スキームでは MPI ランク間のグローバルな同期が排除され、ワークバランスが改善していることが分かります。

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。

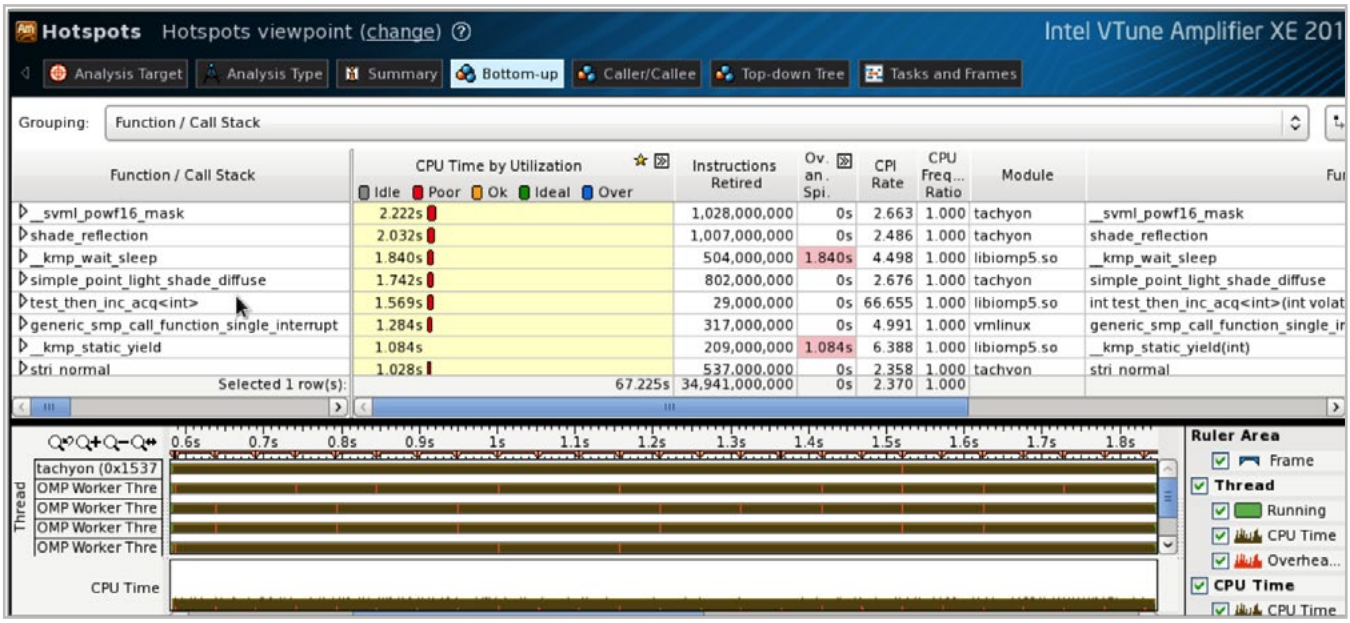
予想どおり、マスターはほとんどの時間をワーカーからの結果を待機するのに費やしています（赤い部分）。出力は、マスターの別のスレッドで行われるため、計算の妨げになることはありません。

## より多くのスレッド並列性を引き出す

次に、OpenMP* スレッドに十分な並列処理を分配できるように、粒度を小さくしました。前述のように、フレーム行の粒度はすでに最小であるため、行を分割しなければなりません。そのためには、Y 座標を使用せず、グローバル・ピクセル・インデックスを使用するように、ループを変更する必要がありました。**図 7** は変更後のコードです。この変更は、6 行のコードを変えるだけで済みました。

```
#pragma omp for schedule(runtime) nowait
#if defined(SINGLE_VAR_LOOP)
    for (p = 0; p < total_pixel; p += grain_size) {
      for (pp = 0; pp < grain_size; pp++) {
        int tp = p + pp;
        y = starty + (tp / xcount) * yinc;
        x = startx + (tp % xcount) * xinc;
        addr = hsize * (y - 1) + (3 * (x - 1));
#else /* SINGLE_VAR_LOOP */
    for (y=starty; y<=stopy; y+=yinc) {
      addr = hsize * (y - 1) + (3 * (startx - 1));
      for (x=startx; x<=stopx; x+=xinc,addr+=hskip) {
#endif
        primary.frng = cachefrng; /* each pixel uses
        col=scene->camera.cam_ray(&primary, x, y);
```

**7** OpenMP* for ループにより粒度を小さくするコード変更




**8** インテル® VTune™ Amplifier XE でも OpenMP* スレッドのワークバランスが大幅に改善されたことを確認できます。

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。

並列処理に伴うオーバーヘッドに比べて各ピクセルの計算回数は非常に大きいため、粒度を小さくすることでパフォーマンスが向上しました。「フォルス・シェアリング」（複数のスレッドが同じキャッシュラインへ書き込みを行うこと）を回避するため、最小値は 8 にしました。

インテル® VTune™ Amplifier XE を再度実行したところ、タイムライン・ビューとグリッドビューでワークバランスが向上し、OpenMP* オーバーヘッドが減少したことが確認できました（**図 8**）。

1 つ目の変更と同様に、この変更もインテル® Xeon® プロセッサーにおいてスケーラビリティーの向上を達成しました。ただし、利用可能なスレッド数が大きく異なるため（インテル® Xeon Phi™ コプロセッサーの 61 スレッドに対して、インテル® Xeon® プロセッサーでは 16 スレッドのみ）、向上幅は小さくなっています。

## ベクトル化を利用する

ベクトル化候補を見つけるのは、次の理由から非常に困難です。

- 光線と三角形の交点を計算する hotspot の関数 tri_intersect() には、コンパイラーによってベクトル化可能なループがありません。
- この hotspot の関数はリンクリスト（2 番目の hotspot の関数 grid_intersect()）に適用されるため、配列要素に適用可能な要素関数を使用できません。

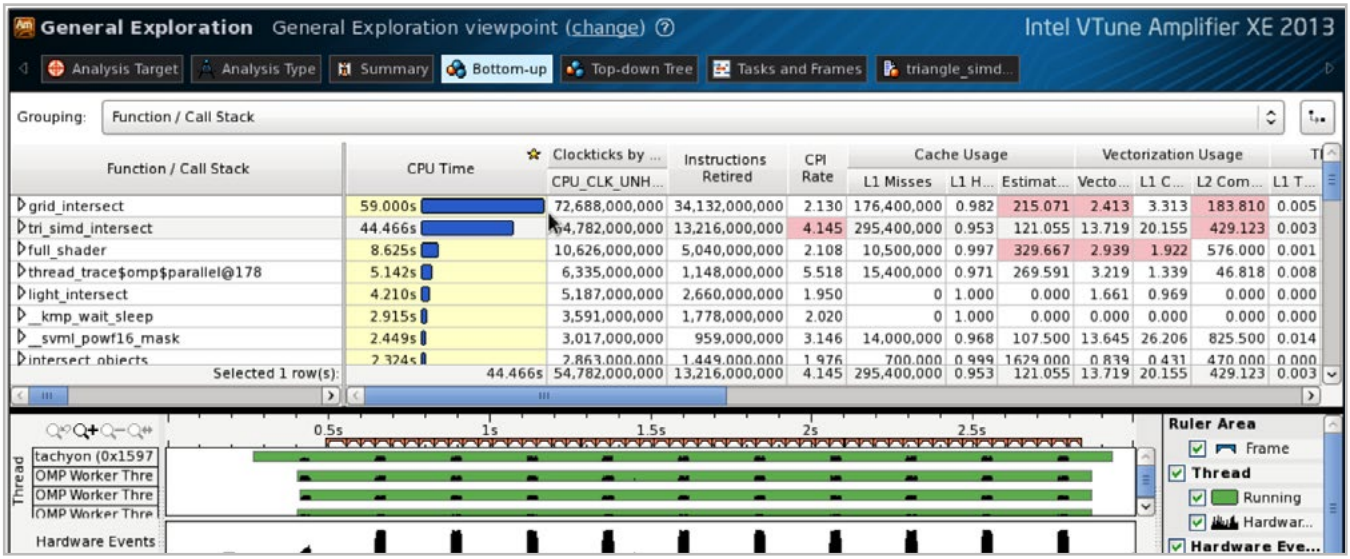
関数 tri_intersect() をベクトル化するため、ここではデータレイアウトを変更しました。16 個の三角形を格納する配列を定義し、ベクトル算術演算を使って、光線と 16 個の三角形の交点を一度に計算する tri_simd_intersection() 関数を実装しました。インテル® Xeon Phi™ コプロセッサー上のベクトルレジスターは、512 ビット幅で 16 個の単精度浮動小数点を格納できるため、これに合わせてここでは 16 個にしています。

同じように、インテル® Xeon® プロセッサー上でベクトル化する場合は、SSE レジスターならば 4 個の三角形、AVX レジスターならば 8 個の三角形を使用します。コードの変更を最小限に抑え、各アーキテクチャー用の tri_simd_intersect() バージョンで重複を避けるため、ここでは C++ テンプレートを用いて tri_simd_intersect() を実装しました。データ構造をテンプレート・パラメーターとして使用し、各データ構造で算術演算（+、-、ドット積、外積など）を多重定義しました。多重定義した演算が確実にベクトル化されるように、組込み関数（ターゲット・アーキテクチャーに応じて、プリフィクス _mm512、_mm256、または _mm を持つ）を使いました。SSE および AVX 向けの実装は、Embree[4] アーキテクチャー固有のデータ構造に基づいています。インテル® Xeon Phi™ コプロセッサー向けの実装も同様の概念です。別の選択肢として、単純なループを実装し、コンパイラーによる自動ベクトル化（あるいは #pragma simd や同様のヒント）に任せることもできるでしょう。

これらの複合オブジェクトの作成は、このアルゴリズムではシーンのロード時に一度だけ行います。この追加コストは、計算時間に比べるとわずかなものです。三角形の数は、常に 16、8、4 の倍数になるとは限らないため、ビットマスクにより実際に配列に格納されている三角形を識別しています。多くのベクトル組込み関数では、追加パラメーターとしてビットマスクを指定できます。

ベクトル化によってベクトル命令が増え、その結果 tri_intersect() が 2 番目の hotspot になりました（**図 9**）。

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。




9 インテル® VTune™ Amplifier XE でもベクトル命令の効率が向上したことを確認できます。

ベクトル化の効果は、3D モデルに大きく依存し、特に三角形をより効率良く配列に格納できる複雑なモデルでは効果が高まります。

### パフォーマンス結果

**表 2** に、3 つのコードを変更した後の Tachyon バージョンで収集した結果と、ベースラインおよびスピードアップを示します。

| | ベースライン | 最終バージョン | スピードアップ |
|---|---|---|---|
| インテル® Xeon® プロセッサーのみ | 141.8 | 218.3 | 1.5 倍 |
| インテル® Xeon Phi™ コプロセッサーのみ | 38 | 173.2 | 4.6 倍 |
| インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー（シンメトリック・モード） | 39 | 378.7 | 9.7 倍 |

2 パフォーマンス（FPS）、スピードアップ、およびベースライン

## まとめ

この記事では、（インテル® Xeon® プロセッサーと比べて）インテル® Xeon Phi™ コプロセッサーでパフォーマンスの悪いアプリケーションを、構造化ワークフローを用いて解析し、パフォーマンスの問題を解決しました。このケーススタディーは、プログラミング・プラットフォームとしてのインテル® Xeon Phi™ コプロセッサーの主要な利点を示しています。

- 同じプログラミング・モデルとツールを使用できる（MPI、OpenMP*、パフォーマンス・プロファイラー）
- 1 つのプラットフォーム（インテル® Xeon Phi™ コプロセッサーまたはインテル® Xeon® プロセッサー）向けに最適化することで、両方のプラットフォームで利点が得られる

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。

両方のプラットフォームを組み合わせた最終バージョンは、インテル® Xeon® プロセッサー向けに最適化されたバージョンと比べてパフォーマンスが 1.73 倍向上し（インテル® Xeon® プロセッサーの 218 FPS に対して 378 FPS）、ベースラインと比べて 9.7 倍もスピードアップしました。この結果を達成するにはコードの変更が必要でしたが、それは限定的なものでした。コードの変更には約 5 週間かかりましたが、投資対効果を考えれば、これは実際のプロジェクトでは許容範囲と言えるでしょう。

ここで紹介した、既存のアプリケーションをインテル® Xeon Phi™ コプロセッサーに移植する例が皆さんのソフトウェア開発に役立つことを願っています。

## 謝辞

Tachyon アプリケーションの設計と暫定バージョンの提供に協力してくれた John E. Stone に感謝します。インテルの同僚である Andrew Tananakin は Tachyon の初期バージョンの解析と OpenMP* 関連の問題の収集に、Embree チームはベクトル化可能なデータ構造の実装に、そして Hans Pabst はこのケーススタディーに意見を寄せ、ユーザーイベントでレビューを実施してくれました。彼らにも感謝しています。

## 参考文献

1. J. Stone, *Tachyon Ray Tracing System*
   http://jedi.ks.uiuc.edu/~johns/raytracer/（英語）
2. http://www.slideshare.net/NakataMaho/some-experiences-for-porting-application-to-intel-xeon-phi（英語）
3. http://www.theismus.de/HPCBlog/?page_id=114（英語）
4. *Embree* Ray Tracing Kernels*
   http://embree.github.io/（英語）
5. インテル® Cluster Studio XE: http://software.intel.com/en-us/intel-cluster-studio-xe（英語）
6. インテル® Xeon Phi™ コプロセッサーに関する開発者リソース：
   http://www.isus.jp/article/idz/mic-developer/
7. K. Rogozhin,「インテル® VTune™ Amplifier XE で OpenMP* アプリケーションをプロファイルする」
   http://www.isus.jp/article/idz/parallel/profiling-openmp-applications-with-intel-vtune-amplifier-xe
8. S. Cepeda,「インテル® Xeon Phi™ コプロセッサー向けの最適化とパフォーマンス・チューニング - パート 2: ハードウェア・イベントの理解と使用」
   http://www.isus.jp/article/mic-article/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding

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。

# インテル® C++ コンパイラーによるネイティブ Android* アプリケーションのビルド

**Anoop Madhusoodhanan Prabha**
インテル コーポレーション
ソフトウェア・エンジニア

インテル® Atom™ プロセッサー・ベースの Android* デバイスは、スマートフォンとタブレットにおいて急速な広がりを見せています。

ほとんどの Android* アプリケーションは Java* で記述されていますが、多くの開発者は C/C++ を使用しており、Android* NDK を利用して共有ライブラリーと JNI (Java* Native Interface) をビルドしています。Java* ではなく、ネイティブ・プログラミングを使う主な理由の 1 つは、スムーズなユーザー・エクスペリエンスをもたらすアプリケーションのパフォーマンスにあります。

インテル® C++ コンパイラー Android* 向けは、Android* NDK の gcc* コンパイラーと互換性があります。パフォーマンスを向上したいと考えるアプリケーション開発者は、インテル® コンパイラーを使用することをお勧めします。インテル® コンパイラーは互換性に優れているため、ほとんどの既存の C/C++ プログラムをコンパイルして、共有ライブラリー (.so) を Android* apk ファイルとともに配布することができます。つまり、開発者は現在使用している NDK ツールと同様に、インテル® C++ コンパイラーでアプリケーションをビルドおよびパッケージ化して、Google Play* などのストアで配布することができます。

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。

インテル® C++ コンパイラー 14.0 Android* 向けの主な機能は、次のとおりです。

1. **マルチ開発プラットフォームのサポートと Android* NDK への自動統合 :** インテル® C++ コンパイラー 14.0 は、Linux*、Windows*、OS X* での開発をサポートしています。インストーラーは、ユーザーが指定した NDK パスに基づいて Android* NDK にコンパイラーを統合します。この統合により、ndk-build スクリプトでインテルのツールを利用できます。ビルドは、icc/icpc を使って直接行うことも、Eclipse* IDE から行うこともできます。詳細は、**導入ガイド (「ドキュメントとサポート」タブ)** を参照してください。
2. **GNU との互換性 :** Android* NDK の GNU* C++ コンパイラーと互換性があり、マルチアーキテクチャーをサポートしています。**インテル® C++ コンパイラー 14.0 Android* 向け**の Web ページで詳しいベンチマーク情報をご覧いただけます。
3. **インテル® Atom™ プロセッサー向けの最適化 :** Saltwell (開発コード名) ベースのインテル® Atom™ プロセッサー向け最適化オプション -xatom_ssse3 と、Silvermont (開発コード名) ベースのインテル® Atom™ プロセッサー向け最適化オプション -xatom_sse4.2 が用意されています。
4. **プロシージャー間の最適化 (IPO) とプロファイルに基づく最適化 (PGO) :** インテル® C++ コンパイラー Android* 向けは、インテル® C++ Composer XE をベースにしているため、プロシージャー間の最適化 (IPO) やプロファイルに基づく最適化 (PGO) など、多数の高度な最適化を Android* ターゲットに利用できます。
5. **インテル® Cilk™ Plus:** インテル® Cilk™ Plus は、タスク並列性とデータ並列性を簡単に実装できるプログラミング・モデルです。詳細は、**「Android* アプリケーション向けにインテル® C++ コンパイラーの高度な機能を利用する」**を参照してください。

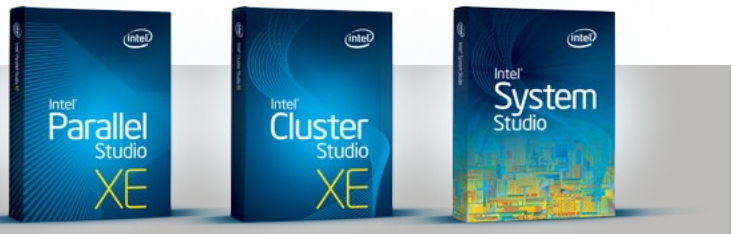
一般に、Android* アプリケーションのライフサイクルは、開発から始まり、apk ファイルへのビルド / パッケージ化を経て、物理デバイスやエミュレーターでのテストに移ります。物理デバイスが利用できない場合は、エミュレーターで代用できます。Android* SDK には、X86 エミュレーターが含まれています。Android* SDK Manager を使用して x86 エミュレーターをインストールする方法は、**「Installing the Intel® Atom™ x86 System Image for Android Emulator* Add-on from the Android SDK Manager*」**(英語) を参照してください。

x86 ベースのエミュレーターのユーザー・エクスペリエンスとパフォーマンスを向上するため、次の 2 つの操作を行うことを推奨します。

1. BIOS で インテル® バーチャライゼーション・テクノロジー (インテル® VT) を有効にします。
2. **インテル® Hardware Accelerated Execution Manager (インテル® HAXM)** をインストールします。Linux* でのインテル® HAXM の使用方法については、**「How to Start Intel® Hardware-Assisted Virtualization (hypervisor) on Linux* to Speedup Intel Android x86 Gingerbread* Emulator」**(英語) を参照してください。

最先端のツールを試そう

30 日間評価版のダウンロード




最新のインテル® Parallel Studio XE およびインテル® Cluster Studio XE (英語) >

組込みおよびモバイルシステム開発者向けインテル® System Studio (英語) >

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の詳細は、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

## BLOG HIGHLIGHTS

# インテル® Xeon Phi™ コプロセッサーの電力管理設定：micsmc GUI インターフェイスの使用

TAYLOR KIDD »

### コプロセッサーの電力管理の設定方法

ワークロード、データセンターの規模と構成、その他の要因に応じて、デフォルトとは異なる電力管理 (PM) の設定が必要になることがあります。詳しくは、このシリーズの以前のブログを参照してください。電力管理 (PM) の設定は、起動時あるいは実行時に micsmc アプリケーションを通じて行います。

micsmc（インテル® MIC System Management and Configuration）アプリケーションを使用すると、装着されているコプロセッサーの監視と設定をホスト側で行えます。GUI だけでなく、コマンドライン・モードも用意されています。

電力状態をグラフィカルに表示し、装着されているコプロセッサーの電力管理ステートを有効 / 無効にするには、FVWM* や GNOME* などのグラフィカル・ウィンドウ・マネージャーを使用して、ホストシステムにログインする必要があります。root としてログインし、通常どおり "**compilervars.sh intel64**" を実行してから、micsmc を起動します。micsmc はユーザーアカウントでも実行できますが、その場合、コプロセッサーを監視することは可能ですが、PM ステートを変更することはできません。PM ステートを変更するには、スーパーユーザー権限が必要です。

micsmc を起動すると、監視用のコンソールが表示されます（次の GUI の図を参照）。[Advanced] > [Settings] を選択すると [Advanced Settings] ウィンドウが表示されます。このウィンドウでコプロセッサーの PM の動作を変更したり、コプロセッサーのリセットや再起動を行います。

```
# source /opt/intel/composer_xe_2013_sp1/bin/compilervars.sh intel64
# micsmc &
#
```

**この記事の続きはこちら（英語）でご覧になれます。 ›**

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。

intel®

Software

The Parallel Universe

© 2014 Intel Corporation. 無断での引用、転載を禁じます。Intel、Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Xeon、Intel Xeon Phi、Cilk、VTune は、アメリカ合衆国および / または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。