チュートリアル: MPI/OpenMP* ハイブリッド・アプリケーションの解析

最終更新 2018 年 2 月 1 日

この記事は、2018 年 3 月 6 日時点の、インテル® デベロッパー・ゾーンに公開されている「Tutorial: Analyzing an OpenMP* and MPI Application」の日本語訳です。

インテル® Trace Analyzer & Collector

アプリケーション・パフォーマンス・スナップショット

インテル® VTune™ Amplifier for Linux*

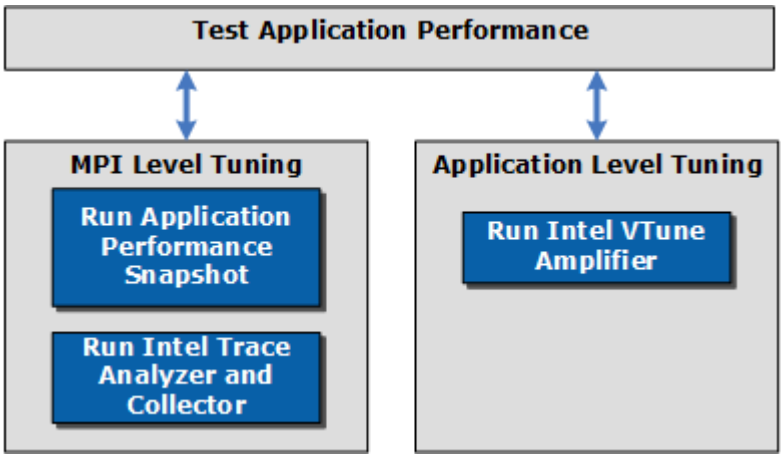
インテル® Parallel Studio XE を使用して非効率な MPI コードを確認し、スレッドのロードバランスを取ることによりハイブリッド・アプリケーションをチューニングする方法を説明します。

チュートリアルの概要	このチュートリアルは、サンプル・アプリケーション `heart_demo` を使用して、インテル® VTune™ Amplifier のアプリケーション・パフォーマンス・スナップショット、インテル® Trace Analyzer & Collector、およびインテル® VTune™ Amplifier を利用して、MPI/OpenMP* ハイブリッド・アプリケーションの非効率的なコードを解析する基本的なステップを説明します。 このチュートリアルは、インテル® Parallel Studio XE 2018 のリリース時に更新されました。解析は、インテル® Xeon Phi™ プロセッサー (開発コード名 Knights Landing) を搭載したクラスターシステム (8 クラスターノード、各 256 論理 CPU) 上で行われました。
所要時間	内容の確認: 10 分程度 サンプル・アプリケーションの実行: 60 分以上

目的	このチュートリアルでは、以下のトピックについて説明します。 • MPI ライブラリーおよびインテル® C++ コンパイラーを使用してアプリケーションをビルドする。 • アプリケーション・パフォーマンス・スナップショットを実行してパフォーマンス最適化の可能性について高レベルの概要を取得する。 • インテル® Trace Analyzer & Collector を実行して MPI 依存のコードを特定する。 • ソースコードの通信パターンを解析する。 • インテル® VTune™ Amplifier の HPC パフォーマンス特性解析を実行してサンプルコードのベクトル化問題および並列化問題を特定する。 • 最適化前と最適化後の結果を比較する。
関連情報	• インテル® Parallel Studio XE トレーニング・ページ: https://www.isus.jp/event/ • インテル® Parallel Studio XE 製品紹介ページ: https://www.isus.jp/intel-parallel-studio-xe/ • インテル® VTune™ Amplifier Web セミナー (41 分): https://www.isus.jp/products/psxe/performance_analysys_for_hpc_webinar/

MPI/OpenMP* ハイブリッド・アプリケーションのパフォーマンスの解析

インテル® Parallel Studio XE Cluster Edition を使用して、ワークフローの一連の手順を実行し、ハイブリッド・アプリケーションの非効率的なコードの原因を理解します。このチュートリアルでは、心臓の電気的活動をシミュレートするサンプル MPI/OpenMP* アプリケーション `heart_demo` を使用して、順に説明します。



ステップ 1: アプリケーションのビルドおよび設定	• `heart_demo` サンプル・アプリケーションをビルドする。 • MPI プロセスおよび OpenMP* スレッドの組み合わせをテストする。
ステップ 2: アプリケーション・パフォーマンス・スナップショットを使用したパフォーマンスの概要の取得	• アプリケーション・パフォーマンス・スナップショットを実行する。 • 結果データを解釈する。
ステップ 3: インテル® Trace Analyzer & Collector を使用した通信問題の特定	• インテル® Trace Analyzer & Collector の環境を設定する。 • インテル® Trace Analyzer & Collector を有効にしてアプリケーションを実行する。
ステップ 4: MPI 依存コードのチューニング	• メッセージ・プロファイル・グラフを確認する。 • MPI 通信問題が解決するようにアプリケーション・コードを更新する。 • 更新したアプリケーションでアプリケーション・パフォーマンス・スナップショットを実行する。 • アプリケーションのパフォーマンスをテストする。

ステップ 5: インテル® VTune™ Amplifier を使用したベクトル命令セットの解析	• `-gtool` を使用してコマンドラインからインテル® VTune™ Amplifier の HPC パフォーマンス特性解析を実行する。 • 解析データを確認して従来の命令セットを使用している場所を特定する。 • ベクトル命令セットを修正する。 • アプリケーションのパフォーマンスをテストする。
ステップ 6: インテル® VTune™ Amplifier を使用したシリアルコードおよび並列コードの効率の解析	• HPC パフォーマンス特性解析を実行する。 • スレッドレベルの並列化が有効な関数を特定する。 • 並列化された関数を使用するようにアプリケーション・コードを更新する。 • アプリケーションのパフォーマンスをテストする。

アプリケーションのビルドおよび設定

ターゲット・アプリケーションのパフォーマンスの解析を開始する前に、次の操作を行います。

1. ソフトウェア・ツールを入手する
2. アプリケーションをビルドする
3. アプリケーションを実行して最適な MPI/OpenMP* プロセスとスレッド構成を判断する

ソフトウェア・ツールを入手する

`heart_demo` サンプル・アプリケーションを使用してチュートリアルを実行するには、次のツールが必要です。

- インテル® Parallel Studio XE Cluster Edition (インテル® C++ コンパイラー、インテル® MPI ライブラリー、インテル® Trace Analyzer & Collector、インテル® VTune™ Amplifier を含む)。
- `heart_demo` サンプル・アプリケーション (GitHub* https://github.com/CardiacDemo/Cardiac_demo (英語) からダウンロードできます)。

アプリケーションをビルドする

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. アプリケーションの GitHub* リポジトリーのクローンをローカルシステムに作成します。

   ```
   $ git clone https://github.com/CardiacDemo/Cardiac_demo.git
   ```

2. インテル® C++ コンパイラーの環境を設定します。

   ```
   $ source <compiler_installdir>/bin/compilervars.sh intel64
   ```

   デフォルトでは、*<compiler_installdir>* は `/opt/intel/compilers_and_libraries_<version>.<update>.<package#>/linux` です。

3. サンプルパッケージのルートレベルに `build` ディレクトリーを作成し、作成したディレクトリーに移動します。

   ```
   $ mkdir build
   ```

   ```
   $ cd build
   ```

4. 次のコマンドを使用してアプリケーションをビルドします。

   ```
   $ mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo
   -O3 -std=c++11 -qopenmp -parallel-source-info=2
   ```

さまざまな設定でアプリケーションを実行する

MPI/OpenMP* ハイブリッド・アプリケーションを実行する場合、プロセスおよびスレッドを実行する最適な組み合わせを見つけることが重要です。異なる組み合わせにより結果が大きく変わることがあるため、さまざまな組み合わ

せを試すことはアプリケーション・パフォーマンスの最適化における重要なステップです。いくつかのシナリオを実行して結果を記録し、アプリケーション・パフォーマンス最適化のベースラインを設定します。

最適な組み合わせの選択は、各ノードのコア数にも依存します。heart_demo アプリケーションでは、利用可能な論理 CPU の半分のロードが最適であることが実験的に判明しています。

例えば、このチュートリアルでは、インテル® Xeon Phi™ プロセッサー (開発コード名 Knights Landing) を搭載したクラスターシステム (8 クラスターノード、各 256 論理 CPU) を使用してアプリケーションを起動しました。そのため、各ノードの 128 コアに MPI プロセスまたは OpenMP* スレッドをロードしました。純粋な計算時間および経過時間を測定するため、次の 3 つの組み合わせを評価しました。

- 128 MPI プロセス、1 OpenMP* スレッド
- 32 MPI プロセス、4 OpenMP* スレッド
- 2 MPI プロセス、64 OpenMP* スレッド

時間を測定するには、次の操作を行います。

1. インテル® MPI ライブラリーの環境を設定します。

   ```
   $ source <impi_installdir>/intel64/bin/mpivars.sh
   ```

   ここで、<impi_installdir> はインテル® MPI ライブラリーのインストール・ディレクトリーです (デフォルトの場所は /opt/intel/compilers_and_libraries_<version>.<update>.<package#>/linux/mpi です)。

2. すべてのクラスターノードをリストしたホストファイルを作成します。

   ```
   node1
   node2
   ...
   node8
   ```

3. 作成したファイルを hosts.txt として build ディレクトリーに保存します。

4. build ディレクトリーで、3 つの組み合わせを使用してアプリケーションを実行します。time ユーティリティーを使用してアプリケーションの経過時間を測定します。計算時間はアプリケーションにより内部的に計算されます。

   ```
   # 128/1
   $ cat > run_ppn128_omp1.sh
   export OMP_NUM_THREADS=1
   mpirun -n 1024 -ppn 128 -f hosts.txt ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t
   50
   $ time ./run_ppn128_omp1.sh
   # 32/4
   $ cat > run_ppn32_omp4.sh
   ```

```
export OMP_NUM_THREADS=4
mpirun -n 256 -ppn 32 -f hosts.txt ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 50
$ time ./run_ppn32_omp4.sh
# 2/64
$ cat > run_ppn2_omp64.sh
export OMP_NUM_THREADS=64
mpirun -n 16 -ppn 2 -f hosts.txt ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 50
       $ time ./run_ppn2_omp64.sh
```

5. 計算時間および経過時間の値を確認して保存します。計算時間および経過時間の値はアプリケーションの出力の最後の行に含まれています。

```
...
wall time: <value>
real <value>
...
```

各実験の結果を次に示します。各自の結果も同様の値になるはずです。

組み合わせ (MPI/OpenMP*)	計算時間	経過時間
128/1	521.57	613.53
32/4	172.60	202.29
2/64	57.15	73.56

結果から次のことが分かります。

- 最初の組み合わせは MPI 並列処理のみ使用しているため、MPI と OpenMP* の両方を使用した場合よりもパフォーマンスは低くなります。さらに調査する必要はありません。
- 2 つ目の組み合わせは中間点です。パフォーマンスは大幅に向上していますが、最高ではありません。アプリケーションの MPI 通信パターンが最適化されていないことが原因であると考えられます。
- 3 つ目の組み合わせは最高のパフォーマンスを示しています。さらに最適化を行うには、この結果に注目することが妥当と言えます。

重要なポイント

- 並列処理の 1 つの手法のみ使用するのは非効率的です。MPI および OpenMP* 並列処理を同時に使用するとパフォーマンスは大幅に向上します。

- ハイブリッド・アプリケーションで MPI プロセスと OpenMP* スレッドのさまざまな組み合わせをテストします。同じアプリケーションでも組み合わせが異なるとパフォーマンス結果が大きく異なることがあります。

最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。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® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、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。

注意事項の改訂 #20110804

アプリケーション・パフォーマンス・スナップショットを使用したパフォーマンスの概要の取得

MPI/OpenMP* ハイブリッド・アプリケーションの解析の最初のステップは、アプリケーション・パフォーマンスの概要を取得することです。アプリケーション・パフォーマンス・スナップショットは、アプリケーションに関する全般的なパフォーマンス情報を提供するツールです。MPI/OpenMP* 時間およびロードバランス情報、メモリーおよびディスク使用状況に関する情報、最も使用されている MPI 操作、その他の情報が含まれます。

アプリケーション・パフォーマンス・スナップショット解析を実行する

アプリケーション・パフォーマンス・スナップショットはインテル® VTune™ Amplifier の一部として配布され、インテル® MPI ライブラリーと統合されます。

`heart_demo` アプリケーションを解析するには、次の操作を行います。

1. MPS の環境を設定します。

   ```
   $ source <parallel_studio_installdir>/intel64/bin/mpsvars.sh
   ```

   ```
   $ source <parallel_studio_installdir>/performance_snapshots/apsvars.sh
   ```

   ここで、*<parallel_studio_installdir>* はインテル® Parallel Studio XE のインストール・ディレクトリーです (デフォルトの場所は `/opt/intel` です)。

2. アプリケーション・パフォーマンス・スナップショット解析を有効にして `heart_demo` アプリケーションを実行します。ノードあたり 2 MPI プロセス、プロセスあたり 64 OpenMP* スレッド (最適な組み合わせ) を使用します。

   ```
   $ export OMP_NUM_THREADS=64 # OpenMP* スレッド数を設定
   ```

   ```
   $ export MPS_STAT_DIR_POSTFIX=_initial # MPS 結果ディレクトリーの postfix を設定
   ```

   ```
   $ mpirun -n 16 -ppn 2 -f hosts.txt aps ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 50
   ```

   統計データを含む `aps_result_<date>` ディレクトリーが作成されます。

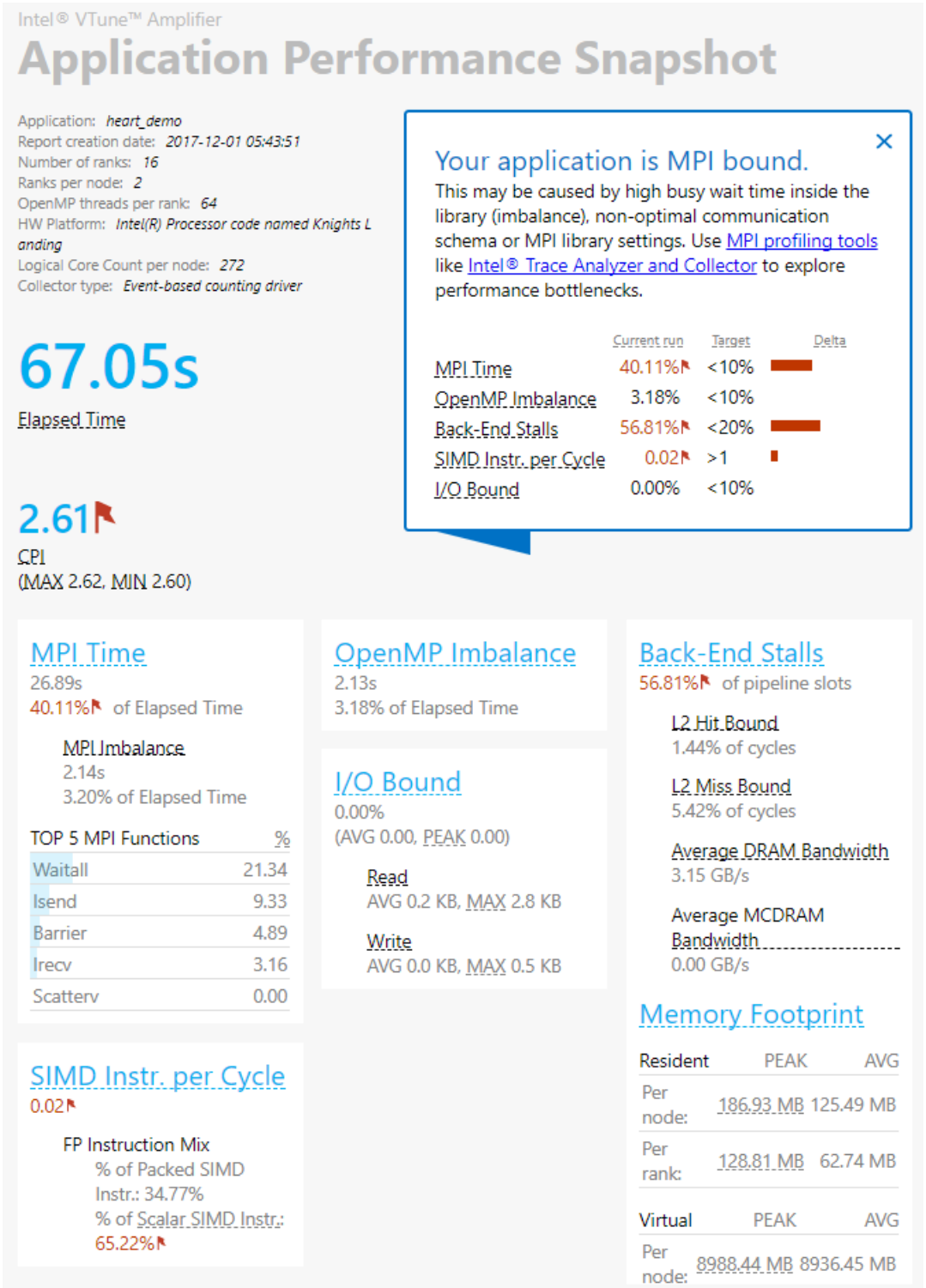
3. アプリケーション・パフォーマンス・スナップショットで統計データを解析し、HTML レポートを生成します。

   ```
   $ aps-report aps_result_<date> -O report_initial.html
   ```

4. 生成された `report_initial.html` ファイルを開いてアプリケーション・パフォーマンスの問題点を調べます。

アプリケーション・パフォーマンス・スナップショットの結果データを解釈する

以下の図は生成された解析レポートを示しています。



アプリケーションが MPI 呼び出しでかなりの時間を費やしていることが分かります。一般的な推奨事項は、MPI 時間をできるだけ減らしてアプリケーションが計算に利用できる時間を増やすことです。このケースでは、MPI 時間が長くアプリケーションが MPI 依存であるため、詳細な調査を行います。より詳細な MPI 解析には、インテル® Trace Analyzer & Collector を使用します。このツールでアプリケーションの通信パターンを明らかにすることにより、問題点を簡単に特定できます。

重要なポイント

アプリケーション・パフォーマンス・スナップショットの HTML レポートを使用して MPI/OpenMP* ハイブリッド・アプリケーションのパフォーマンス問題を特定します。

インテル® Trace Analyzer & Collector を使用した通信問題の特定

アプリケーションが MPI 依存になる主な原因は 3 つあります。

- MPI ライブラリー内部の待機時間が長い。この問題はプロセスがほかのプロセスのデータを待つ場合に発生します。MPI インバランス・インジケーターの値が高いことが特徴です。
- 活発に通信が行われている。
- ライブラリーの最適化設定が適切でない。

最初の 2 つの問題は、インテル® Trace Analyzer & Collector を使用して対処できます。起動コマンドに `-trace` オプションを追加するだけで、アプリケーション・パフォーマンス・スナップショットと同様に簡単にプロファイルを行うことができます。

インテル® Trace Analyzer & Collector 解析を実行する

解析を実行するには、インテル® MPI ライブラリーの環境を設定した状態で、次の操作を行います。

1. インテル® Trace Analyzer & Collector の環境を設定します。

   ```
   $ source <itac_installdir>/bin/itacvars.sh
   ```

   ここで、*<itac_installdir>* はインテル® Trace Analyzer & Collector のインストール・ディレクトリーです (デフォルトの場所は `/opt/intel` です)。

2. `-trace` オプションを指定して `heart_demo` アプリケーションを実行します。前のステップで作成したホストファイルを使用し、同じプロセスとスレッド構成を使用します。

   ```
   $ export OMP_NUM_THREADS=64
   ```

   ```
   $ mpirun -genv VT_LOGFILE_FORMAT=SINGLESTF -trace -n 16 -ppn 2 -f hosts.txt ./heart_demo
   -m ../mesh_mid -s ../setup_mid.txt -t 50
   ```

   この起動コマンドでは、`-genv VT_LOGFILE_FORMAT=SINGLESTF` オプションを指定して、トレースファイルをファイルのセット (デフォルト) ではなく単一ファイルとして生成しています。

   `heart_demo.single.stf` ファイルが作成されます。

3. トレースファイルを開いてアプリケーションを解析します。

   ```
   $ traceanalyzer ./heart_demo.single.stf &
   ```

インテル® Trace Analyzer & Collector の結果データを解釈する

インテル® Trace Analyzer のすべてのグラフの中で、`heart_demo` アプリケーションで最も役に立つグラフはメッセージ・プロファイルです。このグラフは、センダー/レシーバーの各ペアについて、ポイントツーポイント通信の強度を示します。

メッセージ・プロファイル・グラフを開くには、**[Charts (グラフ)]** > **[Message Profile (メッセージ・プロファイル)]** を選択するか、Ctrl + Alt + M キーを押します。`heart_demo` アプリケーションのグラフは次のようになります。

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P0		31.7246	29.9264	29.3301	28.0471	27.667	26.9317	26.2456	24.0593	23.8244	23.2323	22.5	22.3944	22.426	21.7984	21.5232
P1	30.8038		29.4393	28.8912	27.6483	27.2928	26.5862	25.9433	23.863	23.8996	23.22	22.4644	22.3953	22.488	21.8862	21.6183
P2	31.0017	29.474		28.9794	27.9728	27.7438	26.9907	26.3162	24.155	24.114	23.5783	22.6101	22.4698	22.5294	21.9331	21.6607
P3	29.7342	27.9606	27.1749		26.2735	26.0674	25.4494	24.8739	22.7934	22.8303	22.3614	21.5518	21.2142	21.2045	20.5999	20.3098
P4	31.4655	29.6966	28.8357	28.3527		27.638	27.0077	26.4046	24.3333	24.3688	23.9055	23.1854	23.1178	22.9618	22.2834	21.9833
P5	32.0563	30.2991	29.4274	28.8507	27.7607		27.4396	26.8475	24.7515	24.7727	24.3211	23.6136	23.5536	23.6299	22.7637	22.4136
P6	31.7759	30.1136	29.2335	28.6466	27.5934	27.388		26.7187	24.6802	24.7377	24.3018	23.6034	23.5403	23.6087	23.0116	22.4703
P7	32.5673	30.8842	30.0668	29.483	28.4151	28.1946	27.5866		25.4373	25.4985	25.0533	24.3698	24.2991	24.3956	23.8028	23.5443
P8	36.705	35.6248	34.317	33.7145	32.5894	32.3301	31.7187	31.1393		29.6221	29.1775	28.4998	28.423	28.5004	27.917	27.6558
P9	36.186	34.6406	33.9068	33.2568	32.1097	31.8944	31.3261	30.7945	28.7884		28.8957	28.2312	28.187	28.287	27.6992	27.447
P10	35.7396	35.3056	34.6798	34.0654	32.8628	32.6285	31.9999	31.4437	29.3903	29.4856		28.8413	28.799	28.9114	28.3318	28.0917
P11	35.6488	34.3419	33.7796	33.3818	32.2312	31.9911	31.3973	30.8575	28.8292	28.9345	28.5345		28.2416	28.1352	27.3858	26.974
P12	33.7705	32.2796	31.5875	31.1689	30.1597	29.7399	29.0809	28.4988	26.4515	26.526	26.1117	25.4457		25.695	24.9503	24.5371
P13	33.3292	31.911	31.1742	30.7381	29.7333	29.5551	28.7071	28.0545	25.9975	26.0766	25.6642	25.0018	24.711		24.4893	24.0874
P14	33.746	32.3459	31.6046	31.1667	30.172	30.0095	29.4396	28.6606	26.5594	26.6334	26.2287	25.5594	25.2971	25.2407		24.6819
P15	33.6317	32.1992	31.5318	31.1233	30.1216	29.9511	29.3808	28.8228	26.5279	26.51	26.0818	25.4007	25.1275	25.0477	24.3271	

このグラフで、垂直のプロセスバーはセンダーのランクを表し、水平のプロセスバーはレシーバーのランクを表しています。グラフから分かるように、各ランクはほかのランクと通信を行い、ランク 0 はほかのランクよりも少し多いメッセージを受け取っています。

これは、1 つの「マスター」プロセス (ここではプロセス 0) がほかのプロセス間のワークロードの分散および計算結果の収集を行う通信パターンの典型的なケースです。

重要なポイント

インテル® Trace Analyzer & Collector を使用してアプリケーションを調べたときに問題の場所が常に明らかになるとは限りません。アプリケーションおよびすべての利用可能なグラフを調べて問題を見つけます。アプリケーションが MPI 依存の場合、最初に問題の原因 (MPI ライブラリーの待機時間が長い、活発に通信が行われている、最適化設定が適切でない) を特定します。

MPI 依存コードのチューニング

heart_demo アプリケーションのパフォーマンスを向上するには、通信パターンを変更する必要があります。

- サンプルコードを更新する
- インテル® Trace Analyzer & Collector で変更点を確認する
- アプリケーション・パフォーマンス・スナップショットで変更点を確認する
- 異なる設定でパフォーマンスの違いを確認する

サンプルコードを更新する

heart_demo アプリケーションでは、計算を実行する前にメッシュのレンダリングにカットヒル・マキー法を適用することによりアプリケーションのパフォーマンスを向上できます。対応するコードはアプリケーションのソースにすでに含まれています。このアルゴリズムを有効にするには、-i オプションを指定します。この順序変更の効果を確認するため、アプリケーションを再トレースします。

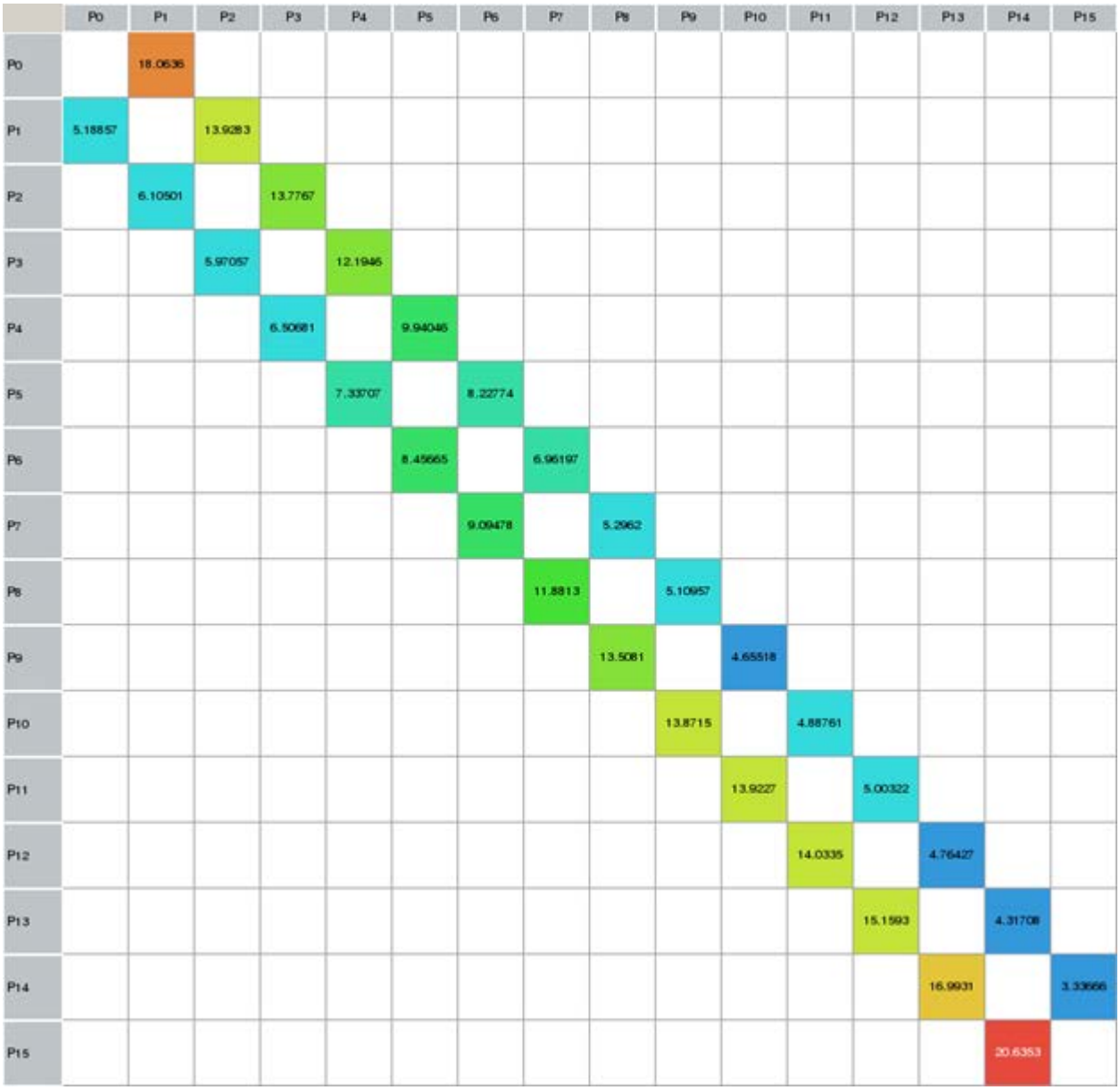
```
$ mpirun -genv VT_LOGFILE_FORMAT=SINGLESTF -genv
VT_LOGFILE_NAME=heart_demo_improved.single.stf -trace -n 16 -ppn 2 -f hosts.txt ./heart_demo
-m ../mesh_mid -s ../setup_mid.txt -t 50 -i

$ traceanalyzer ./heart_demo_improved.single.stf &
```

オリジナルのトレースファイルと区別するため、この起動コマンドでは、-genv VT_LOGFILE_NAME=heart_demo_improved.single.stf オプションを指定して、生成されるトレースファイルの名前を指定しています。

インテル® Trace Analyzer & Collector で変更点を確認する

最適化された通信パターンは、メッセージ・プロファイル・グラフで次のようになります。



通信マトリクスが対角線になり、プロセス間の通信時間が大幅に向上しました。

アプリケーション・パフォーマンス・スナップショットで変更点を確認する

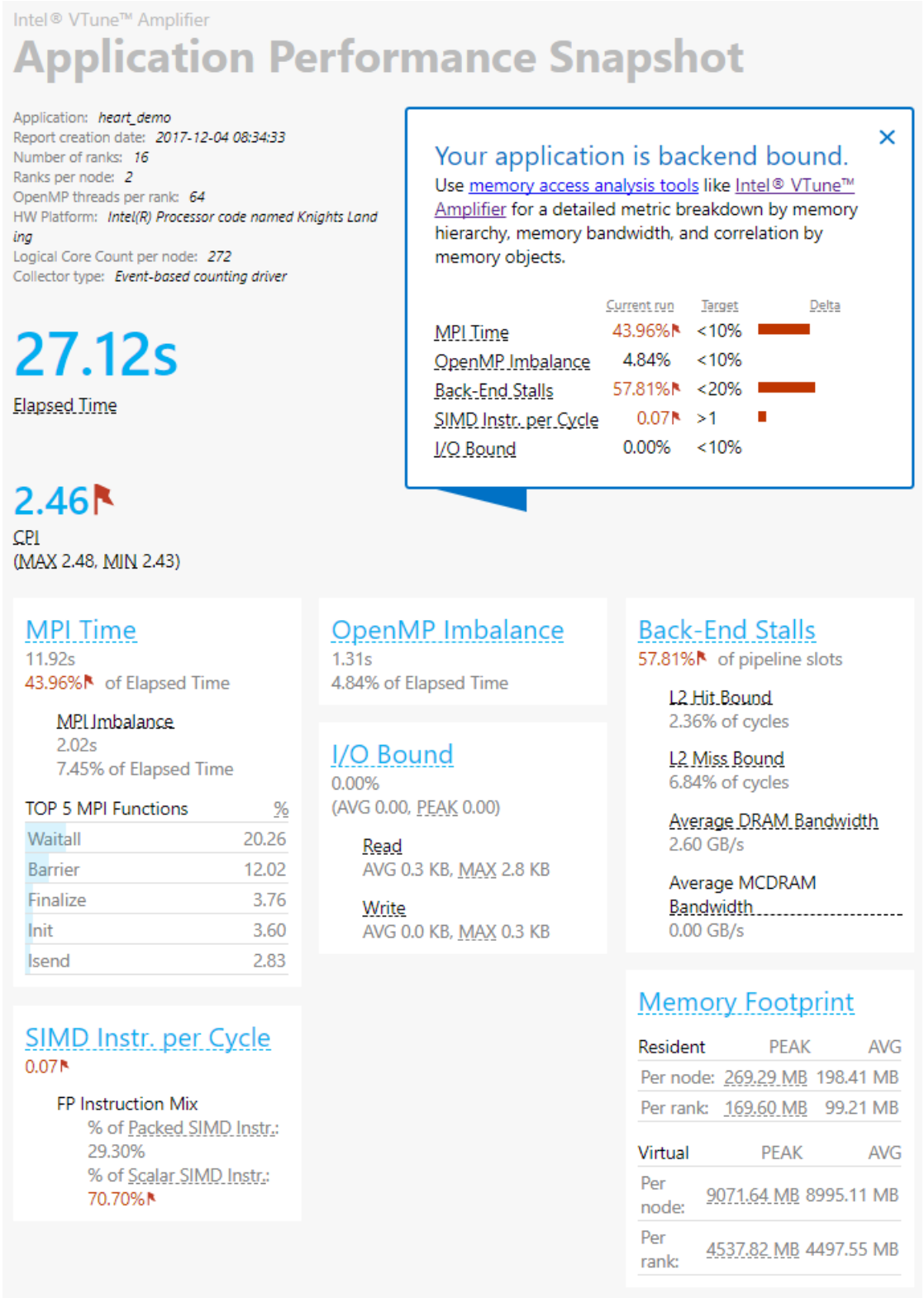
アプリケーション・パフォーマンス全体の影響を確認するため、アプリケーション・パフォーマンス・スナップショットで再度解析を行います。

```
$ export MPS_STAT_DIR_POSTFIX=_second

$ mpirun -aps -n 16 -ppn 2 -f hosts.txt ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 50 -i

$ aps-report stat_second -O report_second.html
```

アプリケーション・パフォーマンス・スナップショットも、アプリケーションのウォールクロック時間 (67.05 秒から 27.12 秒) および MPI 時間 (26.89 秒から 11.92 秒) の大幅な向上を示しています。結果は、まだ MPI 時間が長いことを示しているため、アプリケーションのスレッド化およびベクトル化を検討します。



異なる設定でパフォーマンスの違いを確認する

通信パターンを最適化した後、2 つの MPI/OpenMP* の組み合わせのパフォーマンスを再度確認します。

```
# 2/64
$ cat > run_ppn2_omp64.sh
export OMP_NUM_THREADS=64
mpirun -n 16 -ppn 2 -f hosts.txt ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 50 -i
$ time ./run_ppn2_omp64.sh
```

```
# 32/4
$ cat > run_ppn32_omp4.sh
export OMP_NUM_THREADS=4
mpirun -n 256 -ppn 32 -f hosts.txt ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 50 -i
$ time ./run_ppn32_omp4.sh
```

上記のステップを実行した結果を次に示します。

組み合わせ (MPI/OpenMP*)	計算時間	経過時間
2/64	19.43	35.66
32/4	15.92	45.65

通信の向上の後、2 つ目の組み合わせの計算時間は (経過時間は長いにもかかわらず) 1 つ目の組み合わせよりも短くなりました。これは、プロセス数が多いと MPI 環境の配備に時間がかかるためと考えられます。

2 つ目の組み合わせの計算時間が短いことは、1 つ目の組み合わせでも同様にパフォーマンスが向上する可能性があることを示しています。次のステップでは、インテル® VTune™ Amplifier を使用してこの可能性を調べます。

重要なポイント

最適化が完了した後、MPI プロセスと OpenMP* スレッドの組み合わせのパフォーマンスを再度確認して、違いがないか調べます。正確な経過時間を計測するため、解析ソフトウェアなしでアプリケーションを実行します。

インテル® VTune™ Amplifier を使用したベクトル命令セットの解析

インテル® VTune™ Amplifier を使用して、2/64 の組み合わせの計算時間が (経過時間は短いにもかかわらず) 32/4 の組み合わせよりも長い原因を調べます。MPI 配備のオーバーヘッドにより 32/4 の組み合わせの経過時間を短縮することは不可能です。そのため、2/64 の組み合わせの計算時間を短縮することにします。

インテル® VTune™ Amplifier を使用してアプリケーションのパフォーマンスを解析するには、次の操作を行います。

- 解析を設定する: 解析を実行するプロセスを判断します。
- 収集を実行する: `mpirun` 実行のコマンドラインで `-gtool` オプションを指定してインテル® VTune™ Amplifier の HPC パフォーマンス特性解析タイプを実行します。
- 結果を表示して解析する: インテル® VTune™ Amplifier GUI で結果ファイルを開いてアプリケーション固有の問題を識別します。
- アプリケーションをリビルドする: 新しいベクトル命令セットを使用してアプリケーションをリビルドします。
- アプリケーションのパフォーマンスを確認する: 両方の設定オプションを使用してアプリケーションを再度実行し、パフォーマンスを確認します。

解析を設定する

アプリケーション全体のパフォーマンス・データを収集するのではなく、MPI 時間が最も短いプロセスのデータを収集します。MPI 時間が最も短い (計算時間が最も長い) プロセスがターゲットになります。

1. アプリケーション・パフォーマンス・スナップショットで `-t` オプションを指定してランクデータごとの MPI 時間を表示します。

   ```
   $ aps-report stat_second -t
   ```

2. MPI 時間の値が最も低いランクを見つけます。この例ではプロセス 7 です。

収集を実行する

1. インテル® VTune™ Amplifier の環境を設定します。

   ```
   $ source <vtune_installdir>/amplxe-vars.sh
   ```

   ここで、*<vtune_installdir>* はインテル® VTune™ Amplifier のインストール・ディレクトリーです (デフォルトの場所は `/opt/intel/vtune_amplifier_<version>` です)。

2. インテル® VTune™ Amplifier および適切なランク値を使用してアプリケーションを起動します。

   ```
   $ export OMP_NUM_THREADS=64
   ```

```
$ mpirun -n 16 -ppn 2 -f hosts.txt -gtool "amplxe-cl -collect hpc-performance -data-limit=0
-r result_init:7" ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -i -t 50
```

> 注
> 2 つ目のコマンドのランク値を前のセクションで識別したランク値と置換します。この例では 7 です。

コマンドには次のオプションが含まれています。

- `-gtool` オプションは、指定したランクでインテル® VTune™ Amplifier (`amplxe-cl`) のようなツールを起動します。オプションに関する詳細は、『インテル® MPI ライブラリー for Linux* デベロッパー・リファレンス』(英語) を参照してください。
- `amplxe-cl` は、インテル® VTune™ Amplifier コマンドライン・インターフェイスです。次のオプションを使用して解析を実行します。
  - `-collect` オプションはアプリケーションで実行する解析タイプを指定します。オプションに関する詳細は、「インテル® VTune™ Amplifier ヘルプ」(英語) を参照してください。
  - `-data-limit` オプションは結果ファイルのサイズ制限を指定します (0 は制限なし)。
  - `-r` オプションは結果ファイルの名前と場所を指定します。

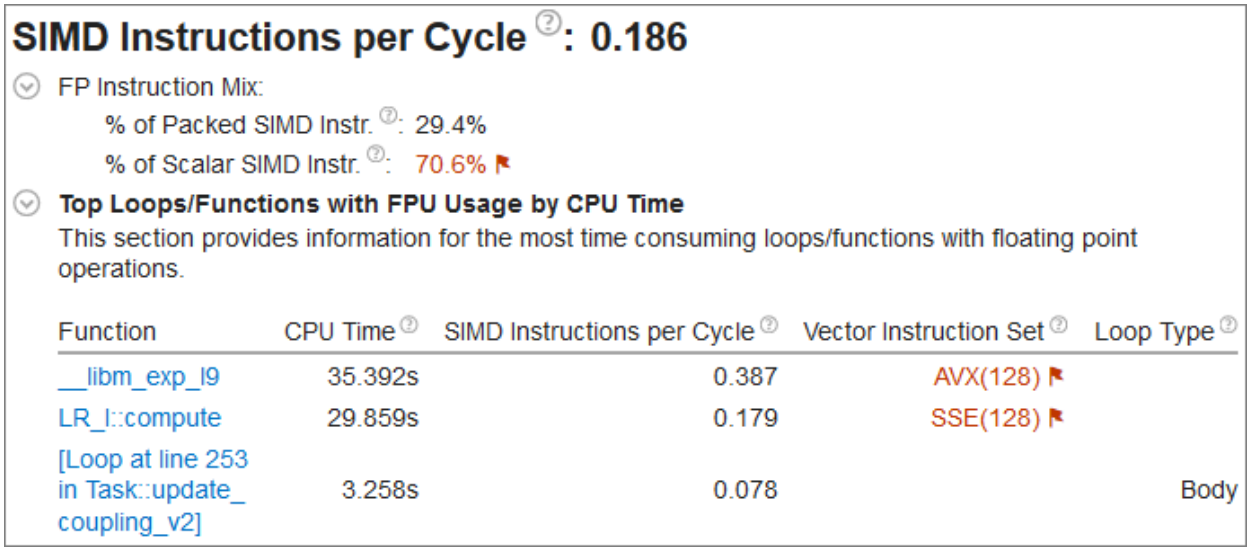
アプリケーションが起動し、パフォーマンス・データの収集が始まります。アプリケーションが完了するとデータの収集が停止し、収集したデータが結果ファイルに保存されます。

結果を表示して解析する

1. パフォーマンス解析を実行した後、次のコマンドを使用してインテル® VTune™ Amplifier GUI で結果ファイルを開きます。

   ```
   $ amplxe-gui result_init.<host>/result_init.<host>.amplxe &
   ```

2. **[Summary (サマリー)]** ウィンドウで解析を開始します。? マークアイコン の上にカーソルを移動してポップアップ・ヘルプを表示し、パフォーマンス・メトリックの意味を確認します。



3. **[SIMD Instructions per Cycle (サイクルあたりの SIMD 命令)]** セクションは、アプリケーションのベクトル化に改良の余地があることを示しています。**[Vector Instruction Set (ベクトル命令セット)]** カラムは、ベクトル命令セットの値 (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® AVX)、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)) が古いことを示しています。同じ情報は **[Bottom-up (ボトムアップ)]** ウィンドウにも表示されます。

新しい命令セットでアプリケーションをリビルドする

アプリケーションは従来の命令セット (インテル® SSE、インテル® AVX、インテル® SSE2) を使用しています。既存のビルドスクリプトに `-xMIC-AVX512` オプションを追加してインテル® AVX-512 命令セットを使用するようにします。新しい命令セットを使用してアプリケーションをリビルドします。

```
$ mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo -O3
-xMIC-AVX512 -std=c++11 -qopenmp -parallel-source-info=2
```

アプリケーションのパフォーマンスを確認する

解析ソフトウェアなしでアプリケーションを実行して、計算時間および経過時間の向上を確認します。

```
$ time run_ppn2_omp64.sh
```

```
$ time run_ppn32_omp4.sh
```

新しい結果を次に示します。

組み合わせ (MPI/OpenMP*)	計算時間	経過時間
2/64	16.32	32.54
32/4	15.37	45.17

ノードあたり 2 プロセス、プロセスあたり 64 OpenMP* スレッドの計算時間は、19.43 秒から 16.32 秒に短縮されました。経過時間も約 3 秒短縮されました。次のステップでは、更新したコードの効率を確認します。

重要なポイント

従来のベクトル命令セットではアプリケーションのパフォーマンスを引き出すことができません。アプリケーションで最新のベクトル命令セットを使用していることを確認します。

インテル® VTune™ Amplifier を使用したシリアルコードおよび並列コードの効率の解析

ベクトル命令セットを更新した後、インテル® VTune™ Amplifier を使用してパフォーマンス・データを再度収集し、ほかの最適化の可能性を調べます。

- パフォーマンス・データを収集する: HPC パフォーマンス特性データを再度収集します。
- 結果を解釈する: 結果を確認して問題のある関数を見つけます。
- コードを変更してアプリケーションをリビルドする: 並列性が向上するようにアプリケーションを更新します。
- アプリケーションのパフォーマンスを確認する: アプリケーションのパフォーマンスをチュートリアルの最初に収集した初期パフォーマンス結果と比較します。

アプリケーションのパフォーマンス・データを収集して確認する

HPC パフォーマンス特性パフォーマンス・データを収集します。

1. インテル® VTune™ Amplifier および適切なランク値を使用してアプリケーションを起動します。

   ```
   $ export OMP_NUM_THREADS=64
   ```

   ```
   $ mpirun -n 16 -ppn 2 -f hosts.txt -gtool "amplxe-cl -collect hpc-performance -data-limit=0 -r result_second:7" ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -i -t 50
   ```

   > 注
   > 2 つ目のコマンドのランク値を前のセクションで識別したランク値と置換します。この例では 7 です。

2. インテル® VTune™ Amplifier GUI で結果を開いて **[Summary (サマリー)]** ウィンドウを表示します。

   ```
   $ amplxe-gui result_second.<host>/result_second.<host>.amplxe &
   ```

結果を解釈する

1. **[CPU Utilization (CPU 使用率)]** セクションで、**[Serial Time (outside parallel regions) (シリアル時間 (並列領域外))]** を展開して **[Top Serial Hotspots (outside parallel regions) (上位のシリアル hotspot (並列領域外))]** リストを表示します。

CPU Utilization : 0.6%

Average CPU Usage : 1.626 Out of 272 logical CPUs

Serial Time (outside parallel regions) : 16.347s (24.4%)

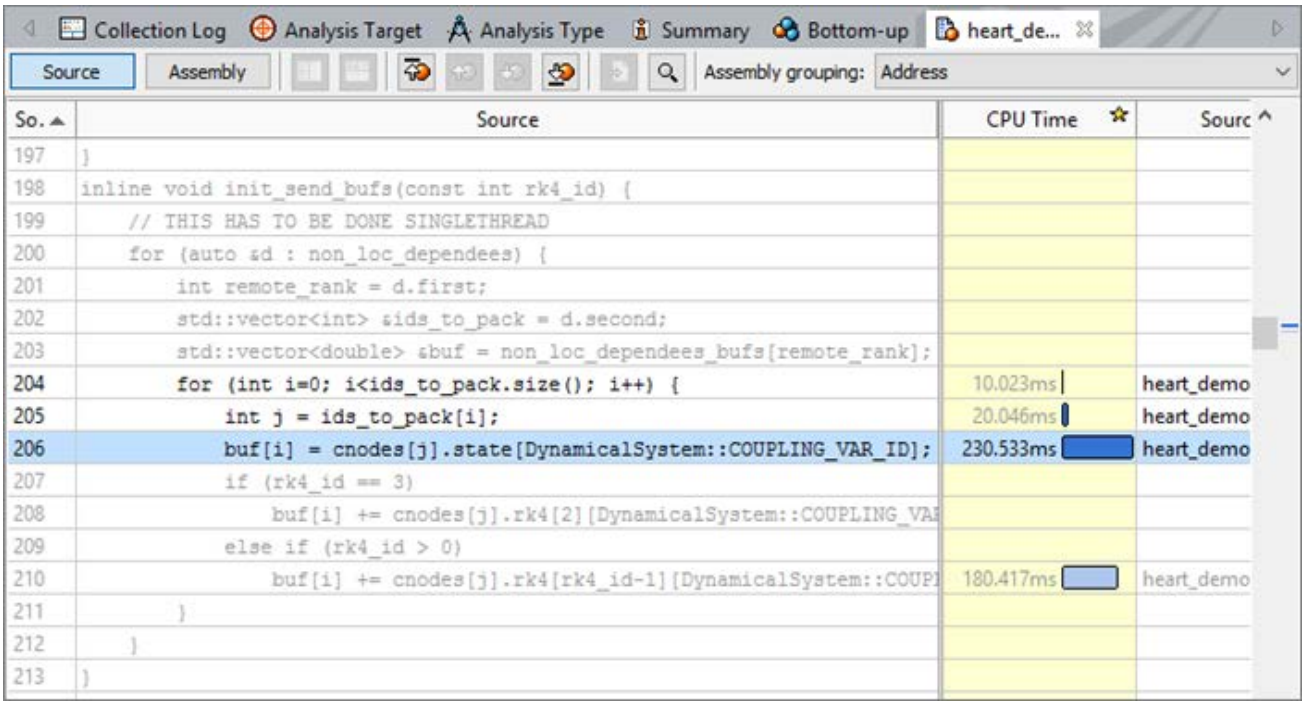
Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

Function	Module	Serial CPU Time
[Loop@0x11580b in func@0x115660]	libmpi.so.12.0	4.891s
[Loop at line 204 in Task::init_send_bufs]	heart_demo	0.441s
[Loop at line 204 in Task::init_send_bufs]	heart_demo	0.381s
[Loop at line 1801 in __kmp_fork_call]	libiomp5.so	0.291s
[Loop@0x14de4a in MPIDI_CH3I_Progress]	libmpi.so.12.0	0.241s
[Others]		9.793s

表の最初の関数は、アプリケーションにより呼び出された MPI ライブラリーの一部です。`heart_demo` アプリケーションの一部ではないため、この関数は最適化の対象ではありません。代わりに、コンパイラーの最適化により表に 2 回表示されている `init_send_bufs` 関数から始めます。

2. **[Bottom-up (ボトムアップ)]** タブを選択して、グループを **[OpenMP Region / Thread / Function / Call Stack (OpenMP* 領域/スレッド/関数/コールスタック)]** に設定し、ウィンドウ下部のフィルターを適用して **[Functions only (関数のみ)]** を表示します。ツリーを展開して、`init_send_bufs` 関数がスレッド 0 以外から呼び出されていないことを確認します。この行をダブルクリックしてソースコード・ビューを開きます。



ソースコード・ビューを使用して、コードがランク間で分割されていて CPU 時間が長いにもかかわらず、スレッドレベルで並列化されていないことを確認します。この関数の内側のループは、ループの前に別の OpenMP* プラグマ `#pragma omp parallel for` を追加して並列化できます (外部ループはシングルスレッドのまま変更しません)。

[Bottom-up (ボトムアップ)] タブに戻り、CPU 時間の長いほかの関数を確認します。アプリケーションは、`_kmp_join_barrier` 関数でも時間を費やしています。原因は、各 `#pragma omp parallel for` 構文でバリアの同期によるオーバーヘッドが発生しているためです。`heart_demo` アプリケーションにはこの構文が複数含まれています。1 つの `#pragma omp parallel` 構文のみを使用し、その内側で複数の `#pragma omp for` 構文を使用するようにコードを変更して、コストのかかる `#pragma omp parallel` 構文のバリアの同期を減らします。

コードを変更してアプリケーションをリビルドする

コード変更後のサンプル・アプリケーションが `heart_demo_opt.cpp` ファイルです。変更点は、`heart_demo.cpp` ファイルと `heart_demo_opt.cpp` ファイルを比較して確認できます。

次のコマンドを使用してアプリケーションをリビルドします。

```
$ mpiicpc ../heart_demo_opt.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo
-O3 -xMIC-AVX512 -std=c++11 -qopenmp -parallel-source-info=2
```

アプリケーションのパフォーマンスを確認して比較する

2 つの MPI/OpenMP* の組み合わせのパフォーマンスを調べて、アプリケーションの全体的なパフォーマンスの向上を確認します。次のコマンドを実行してパフォーマンスを確認します。

```
$ time run_ppn2_omp64.sh
```

```
$ time run_ppn32_omp4.sh
```

サンプルの全体的なパフォーマンスの向上を次に示します。

組み合わせ (MPI/OpenMP*)	計算時間	経過時間
2/64	11.91	28.13
32/4	16.39	46.14

最終的に、2/64 の組み合わせの計算時間と経過時間は 32/4 の組み合わせよりも短くなりました。並列化されていないコードが並列化され、複数のスレッドで実行されるようになりました。また、各 OpenMP* 構文のバリアを削除して、アプリケーションの待機時間を減らしました。

計算時間の全体的なパフォーマンスの向上を次に示します。

組み合わせ (MPI/OpenMP*)	2/64	32/4
オリジナルの計算時間	57.15	172.60
MPI チューニング後の計算時間	19.43	15.92
ベクトル命令セット変更後の計算時間	16.32	15.37
最終的な計算時間	11.91	16.39

経過時間の全体的なパフォーマンスの向上を次に示します。

組み合わせ (MPI/OpenMP*)	2/64	32/4
オリジナルの経過時間	73.56	202.29
MPI チューニング後の経過時間	35.66	45.65
ベクトル命令セット変更後の経過時間	32.54	45.17
最終的な経過時間	28.13	46.14

重要なポイント

インテル® VTune™ Amplifier の [Bottom-up (ボトムアップ)] タブを確認して、問題のある関数を特定し、並列化が有効なアプリケーションのセクションを見つけます。

まとめ

このチュートリアルでは、アプリケーション・パフォーマンス・スナップショット、インテル® Trace Analyzer & Collector、およびインテル® VTune™ Amplifier を利用して MPI/OpenMP* ハイブリッド・アプリケーションを解析し、アプリケーションのパフォーマンスを最適化しました。各自のハイブリッド・アプリケーションを最適化するときは、次の点に注意してください。

ステップ	チュートリアルの概要	チュートリアルの重要なポイント
1. アプリケーションのビルドおよび設定	必要なツールがすべてインストールされていることを確認しました。アプリケーションをビルドして、最適化の可能性を判断するため、さまざまなプロセス数/スレッド数の組み合わせでアプリケーションを実行しました。	ハイブリッド・アプリケーションで MPI プロセスと OpenMP* スレッドのさまざまな組み合わせをテストします。同じアプリケーションでも組み合わせが異なるとパフォーマンス結果が大きく異なることがあります。
2. アプリケーション・パフォーマンス・スナップショットを使用したパフォーマンスの概要の取得	`-aps` オプションを指定して `heart_demo` アプリケーションを実行し、ロードバランス情報、メモリーおよびディスク使用状況の情報、その他のメトリックを収集しました。	アプリケーション・パフォーマンス・スナップショットの HTML レポートを使用してアプリケーションの効率が悪い部分を確認し、次に使用するツールを決定します。
3. インテル® Trace Analyzer & Collector を使用した通信問題の特定	`-trace` オプションを指定してアプリケーションを実行し、MPI ライブラリーの待機時間と通信パターンを理解しました。メッセージ・プロファイル・グラフを使用して結果を確認し、通信問題を特定しました。	• アプリケーションは、さまざまな原因 (MPI ライブラリーの待機時間が長い、活発に通信が行われている、最適化設定が適切でない) により MPI 依存になります。 • インテル® Trace Analyzer & Collector を使用してアプリケーションを調べたときに問題の場所が常に明らかになるとは限りません。アプリケーションおよびすべての利用可能なグラフを調べて問題を見つけます。 • メッセージ・プロファイル・グラフを使用して、センダー/レシーバーの各ペアについて、ポイントツーポイント通信の強度を表示します。

4. MPI 依存コードのチューニング	計算を実行する前にメッシュのレンダリングにカットヒル・マキー法を適用することによりアプリケーションを最適化しました。インテル® Trace Analyzer & Collector とアプリケーション・パフォーマンス・スナップショットを使用してパフォーマンスの向上を確認しました。	最適化が完了した後、MPI プロセスと OpenMP* スレッドの組み合わせのパフォーマンスを再度確認して、違いがないか調べます。正確な経過時間を計測するため、解析ソフトウェアなしでアプリケーションを実行します。
5. インテル® VTune™ Amplifier を使用したベクトル命令セットの解析	アプリケーション・パフォーマンス・スナップショットのレポートで指摘されたスレッドについて、インテル® VTune™ Amplifier を使用して `heart_demo` アプリケーションのパフォーマンス解析を実行しました。最新のベクトル命令セットを使用するようにオプションを変更しました。	従来のベクトル命令セットではアプリケーションのパフォーマンスを引き出すことができません。アプリケーションで最新のベクトル命令セットを使用していることを確認します。
6. インテル® VTune™ Amplifier を使用したシリアルコードおよび並列コードの効率の解析	インテル® VTune™ Amplifier を使用して並列化の問題点を確認しました。問題点を修正するようにサンプルコードを更新しました。さまざまなプロセス数/スレッド数の組み合わせを調べて効率の向上を確認しました。	インテル® VTune™ Amplifier の [Bottom-up (ボトムアップ)] タブを確認して、スレッド化が有効なアプリケーションのセクションを特定し、スレッド化されたコードの効率を調査します。

重要な用語

ベースライン: 最適化前と最適化後のアプリケーションのバージョンを比較するベースとして使用するパフォーマンス・メトリック。ベースラインは測定可能で再現可能なものにします。

計算時間: 追加のオーバーヘッド (初期化時間、ファイナライズ処理時間など) を含まないアプリケーションの実行時間。計算時間は経過時間に含まれます。

CPU 使用率: インテル® VTune™ Amplifier で、プロセッサー使用率のスケールの識別、ターゲット CPU 使用率の計算、プロセッサー・コアの数に応じたデフォルトの使用率の定義を行うときのパフォーマンス・メトリック。

使用率の種類	デフォルトの色	説明
Idle (アイドル)		すべての CPU が待機中です。実行中のスレッドはありません。
Poor (不十分)		不十分な使用率です。デフォルトでは、同時に実行している CPU の数がターゲット CPU 使用率の 50% 以下の状態です。
OK		許容可能な使用率です。デフォルトでは、同時に実行している CPU の数がターゲット CPU 使用率の 51-85% の状態です。
Ideal (理想的)		理想的な使用率です。デフォルトでは、同時に実行している CPU の数がターゲット CPU 使用率の 86-100% の状態です。

経過時間: アプリケーションの実行時間の合計で、**アプリケーション終了時のウォールクロック時間 - アプリケーション開始時のウォールクロック時間**で計算されます。経過時間には、アプリケーションが動作している時間 (計算時間)、システムがアプリケーションを初期化する時間、および解析をファイナライズする時間が含まれます。

著作権と商標について

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。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、知的財産権の非侵害性への保証、およびインテル製品の性能、取引、使用から生じるいかなる保証を含みますが、これらに限定されるものではありません。

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。本資料に含まれる情報は予告なく変更されることがあります。最新の予測、スケジュール、仕様、ロードマップについては、インテルの担当者までお問い合わせください。

本資料で説明されている製品およびサービスには、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。現在確認済みのエラッタについては、インテルまでお問い合わせください。

インテル・プロセッサー・ナンバーはパフォーマンスの指標ではありません。プロセッサー・ナンバーは同一プロセッサー・ファミリー内の製品の機能を区別します。異なるプロセッサー・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.com/content/www/jp/ja/processors/processor-numbers.html を参照してください。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。

Intel、インテル、Intel ロゴ、Xeon、Intel Xeon Phi、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。