インテル® Parallel Studio XE 2017 Professional Edition

ー 入門ガイド ー

エクセルソフト株式会社

www.xlsoft.com

Rev 1.0.0

2. 目次

1. 目次 2

2. はじめに 4

2-1 インテル® Parallel Studio XE 製品概要 4

2-2 開発ワークフロー 4

3. マルチスレッド化の最適化チュートリアル 5

3-1 プログラムをビルドする 6

STEP1 Visual Studio にサンプルプログラムのプロジェクトをロードする 6

STEP2 インテル®コンパイラーを使用してソースコードをビルドする 7

3-2 マルチスレッド処理を実装する 9

STEP3 Visual Studio からインテル® Advisor にアクセスする 9

STEP4 プログラム内でマルチスレッド化可能な処理を見つける 10

STEP5 マルチスレッド化による性能変化をシミュレーションする 13

STEP6 OpenMP*を使用してマルチスレッド化処理を実装する 17

3-3 実行の度に異なる計算結果を改善する 21

STEP7 インテル® Inspector を使用するための準備 21

STEP8 Visual Studio からインテル® Inspector にアクセスする 22

STEP9 スレッドエラー解析を実行する 23

STEP10 計算結果が異なる原因を特定する 25

STEP11 ソースコードを変更して改善する 26

3-4 マルチスレッド化された処理の性能を検証する 30

STEP12 Visual Studio からインテル VTune™ Amplifier XE にアクセスする 30

STEP13 並列性に注目して解析する 31

STEP14 解析結果から実行状況を確認する 33

STEP15 スレッド毎の実行時間に差がる原因を確認する 38

3-5 より実行時間を短縮するための最適化 41

STEP16 インテル® VTune Amplifier XE の EBS 機能を使用する 41

STEP17 General Exploration 解析の結果から最適化できそうな箇所を判断する 41

STEP18 キャッシュミスの原因を特定して改善する 44

4. 解析ツールに関する Tips 47

4-1 インテル® Advisor 47

◇ 効果的なベクトル化を実装する 47

4-2 インテル® Inspector 48

◇ メモリエラー解析 48

◇ 外部デバッガーと連携して解析結果のエラーをデバッグする 48

4-3 インテル® VTune Amplifier XE 50

◇ TBS と EBS 50

◇ CPI 51

◇ プログラムの実行中から解析を開始する 51

5. 参考情報 52
5-1 インテル® Distribution for Python* 52
◇ ダウンロード方法 52
◇ Visual Studio での実行方法 53
5-2 参考資料 56

3. はじめに

3-1 インテル® Parallel Studio XE 製品概要

インテル® Parallel Studio XE 製品は、アプリケーションの高速化を支援するスイート製品です。本ドキュメントでは、インテル® Parallel Studio XE 製品に含まれる解析ツールを使用した最適化手順をチュートリアル形式で紹介します。

■ インテル® Advisor

実際にアプリケーションをマルチスレッド化する前に、スレッド並列の候補となる処理や、マルチスレッド化によるパフォーマンス向上の可能性、またマルチスレッド化により発生する可能性のある問題などの情報を事前に提供する並列化設計ツールです。また、アプリケーションのベクトル化情報も提供し効果的なベクトル化を支援します。

■ インテル® Inspector

プログラム内に潜む「メモリエラー」やマルチスレッド・アプリケーションで発生する「スレッドエラー」に関するチェックをランタイムで行う動的診断ツールです。

■ インテル® VTune™ Amplifier XE

プログラムのボトルネックの調査やマルチスレッド並列実行動作の分析、また CPU キャッシュミスやパイプライン・ストール状況、分岐予測ミスなどマイクロアーキテクチャ・レベルのプロファイリングを行うパフォーマンス解析ツールです。

3-2 開発ワークフロー

本製品を使用してプログラムの並列化を行う基本開発工程（ワークフロー）について説明します。

(1) プログラムに対してインテル® Advisor を使用し、効果的なマルチスレッド化が可能な処理を発見します。

(2) マルチスレッド化が可能な処理に対してマルチスレッド処理を実装します。

(3) コンパイラーを使用してマルチスレッド・プログラムをビルドします。

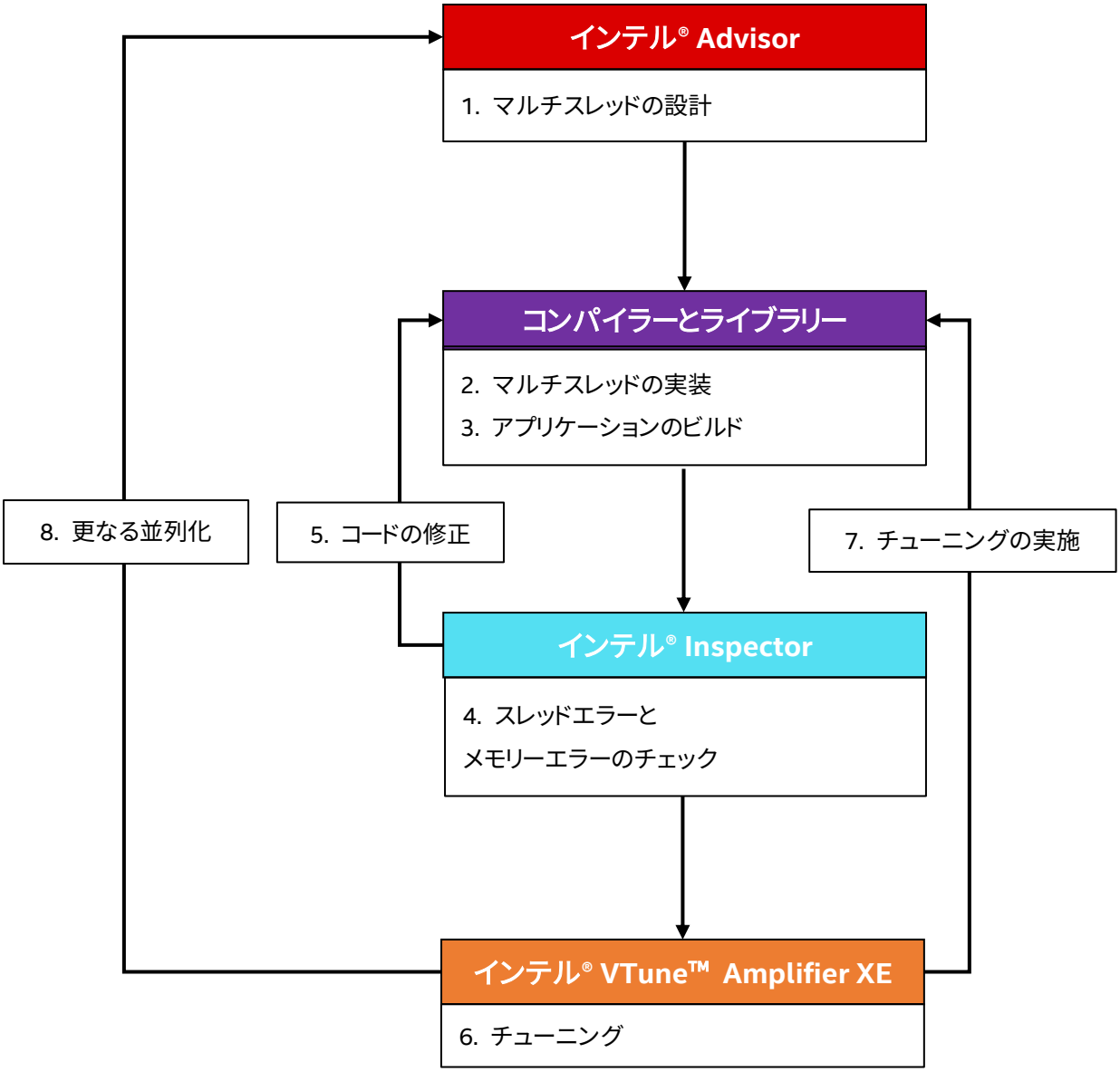
(4) インテル® Inspector で、マルチスレッド処理のエラーチェックを実施します。

(5) エラーが潜在する場合は、コードを修正してエラーを取り除きます。

(6) インテル® VTune™ Amplifier XE で、プログラムを解析します。

(7) 解析結果からチューニングの余地がある場合は、コードを改善して再コンパイルし効果を確認します。

(8) さらに効果的なマルチスレッド化が可能な処理があるか、インテル® Advisor を使用して検証します。



4. マルチスレッド化の最適化チュートリアル

このチュートリアルでは、Microsoft® Windows® OS で本製品を Microsoft® Visual Studio® 上にインストールした環境で行います。使用するサンプルプログラムに含まれる Visual Studio のプロジェクトを読み込み、Visual Studio 上から最適化チュートリアルを開始します。

インテル® Advisor 付属のサンプルプログラムを使用します。製品インストール後、下記のパスに配置されます。

C:\Program Files (x86)\IntelSWTools\samples_2017\en\Advisor\C++\ nqueens_Advisor.zip

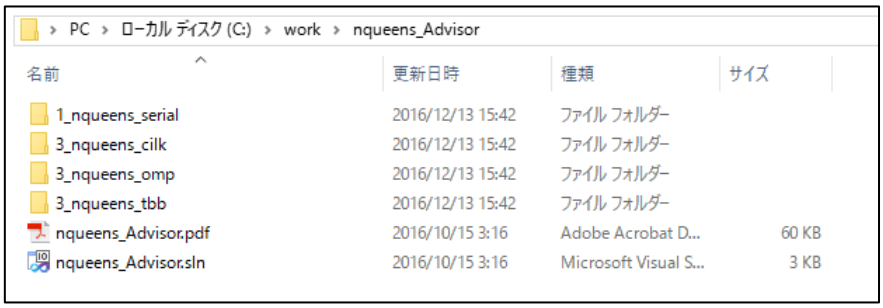
4-1 プログラムをビルドする

インテルコンパイラーを使用してサンプルプログラムをビルドして、実行結果と実行速度を確認します。

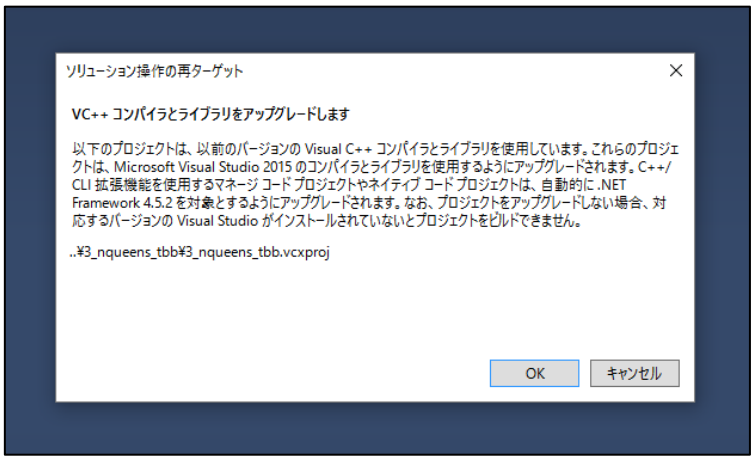
STEP1　Visual Studio にサンプルプログラムのプロジェクトをロードする

(1)　サンプルプログラムを適当な作業フォルダに展開します

※ここでは、C:¥work フォルダに展開しています。



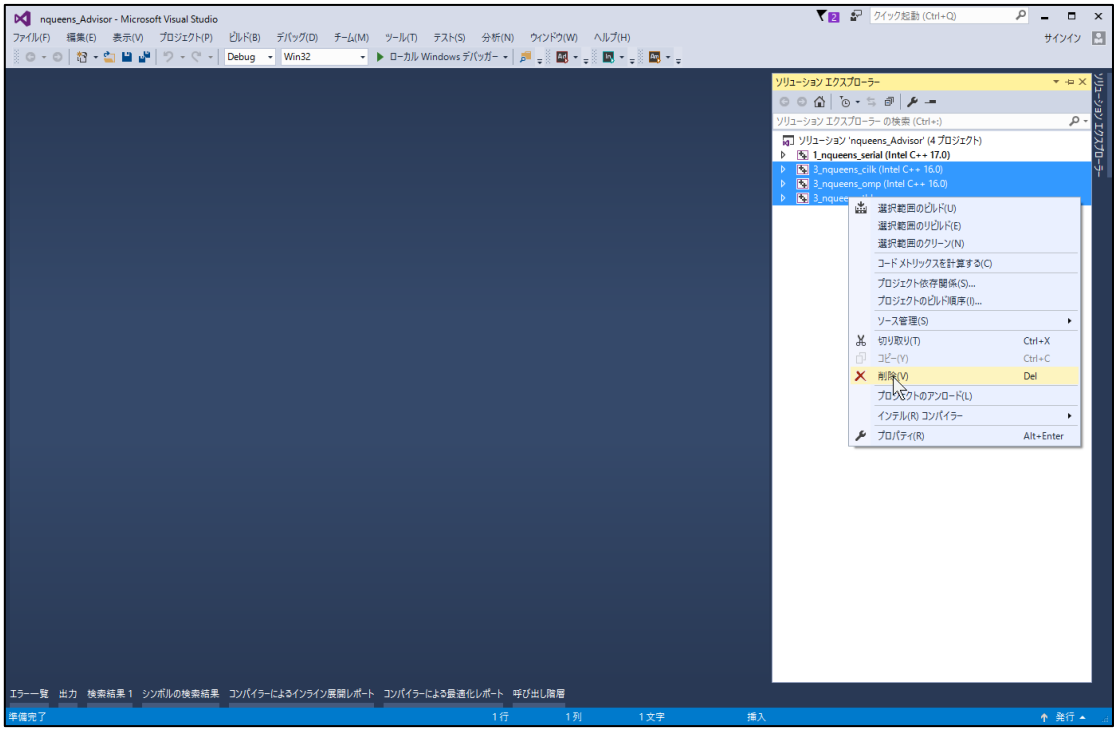
(2)　nqueens_Advisor.sln を Visual Studio で開きます



※サンプルプログラムの Visual Studio プロジェクトに対してアップグレード処理を行うので、OK を選択します。

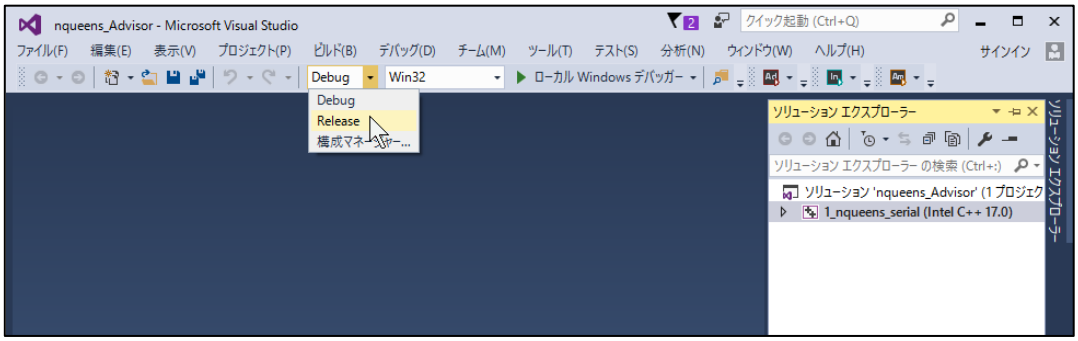
(3)　ソリューションエクスプローラーから 1_nqueens_serial 以外のプロジェクトを削除します

チュートリアルでは 1_nqueens_serial を使用します。

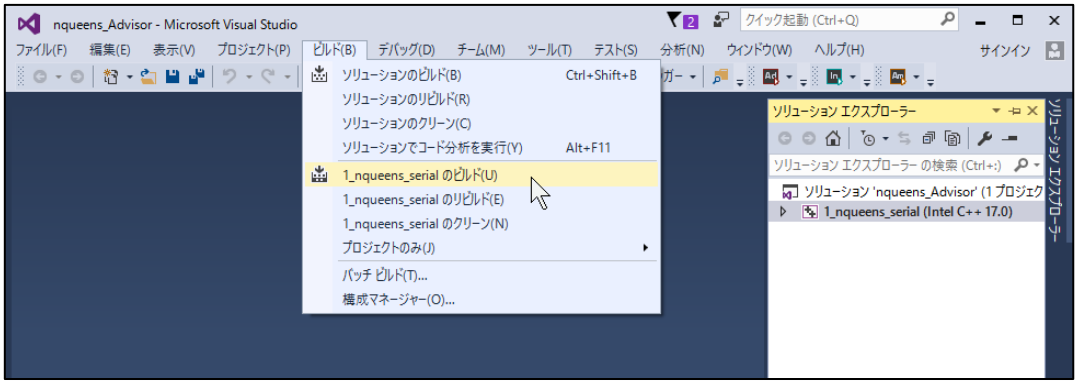


STEP2　インテル®コンパイラーを使用してソースコードをビルドする

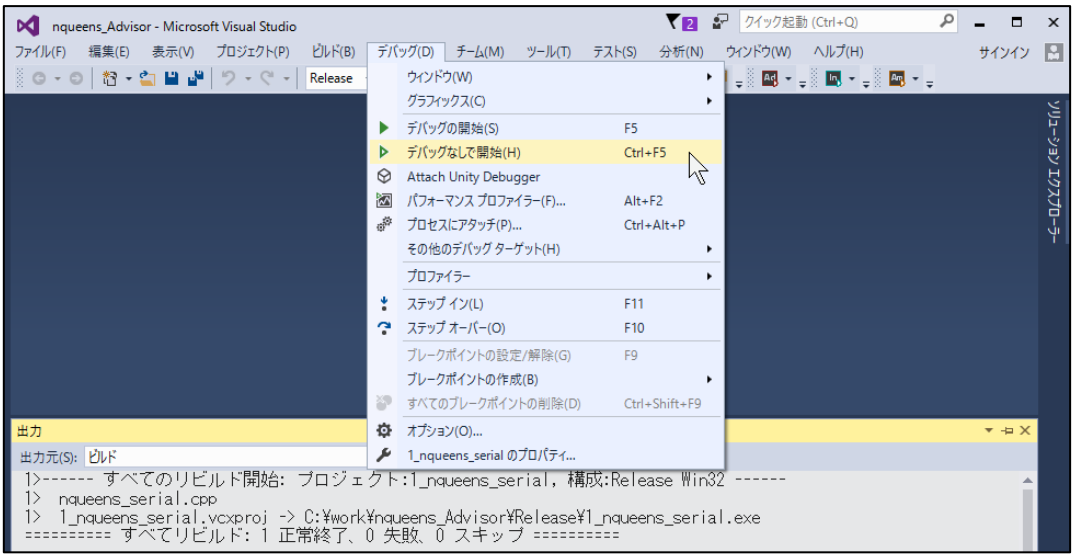
(1)　プロジェクト構成を Release モードに切り替えます



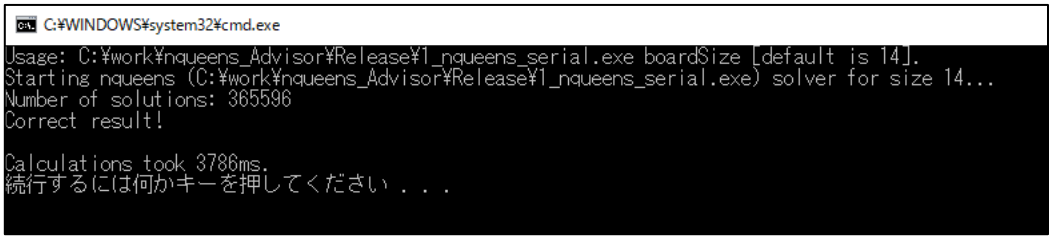
(2)　プロジェクトをビルドします



(3)　実行して結果を確認します



Calculations took に実行時間が表示されます。ここでは、サンプルプログラムの計算に **3786** ミリ秒かかっていることが確認できます。また、Number of solutions に計算の結果が **365596** であることが表示されています。

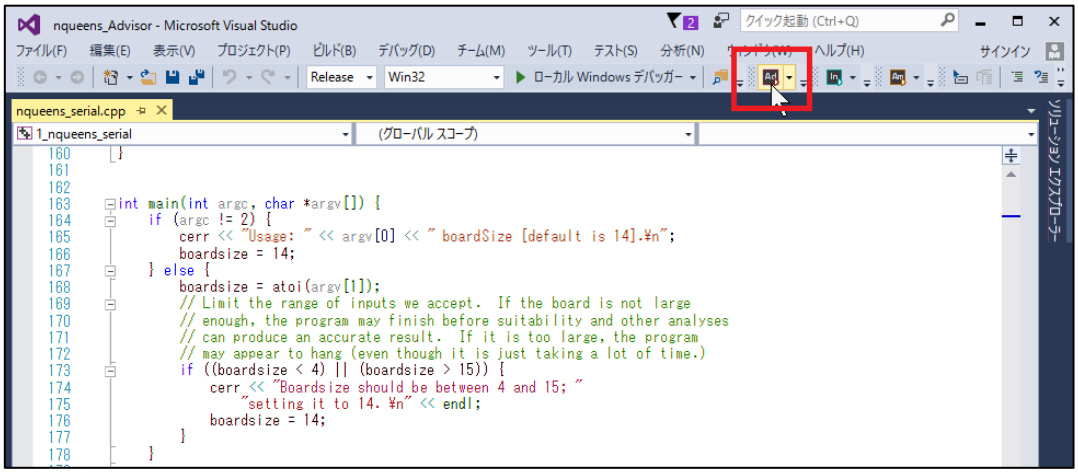


4-2 マルチスレッド処理を実装する

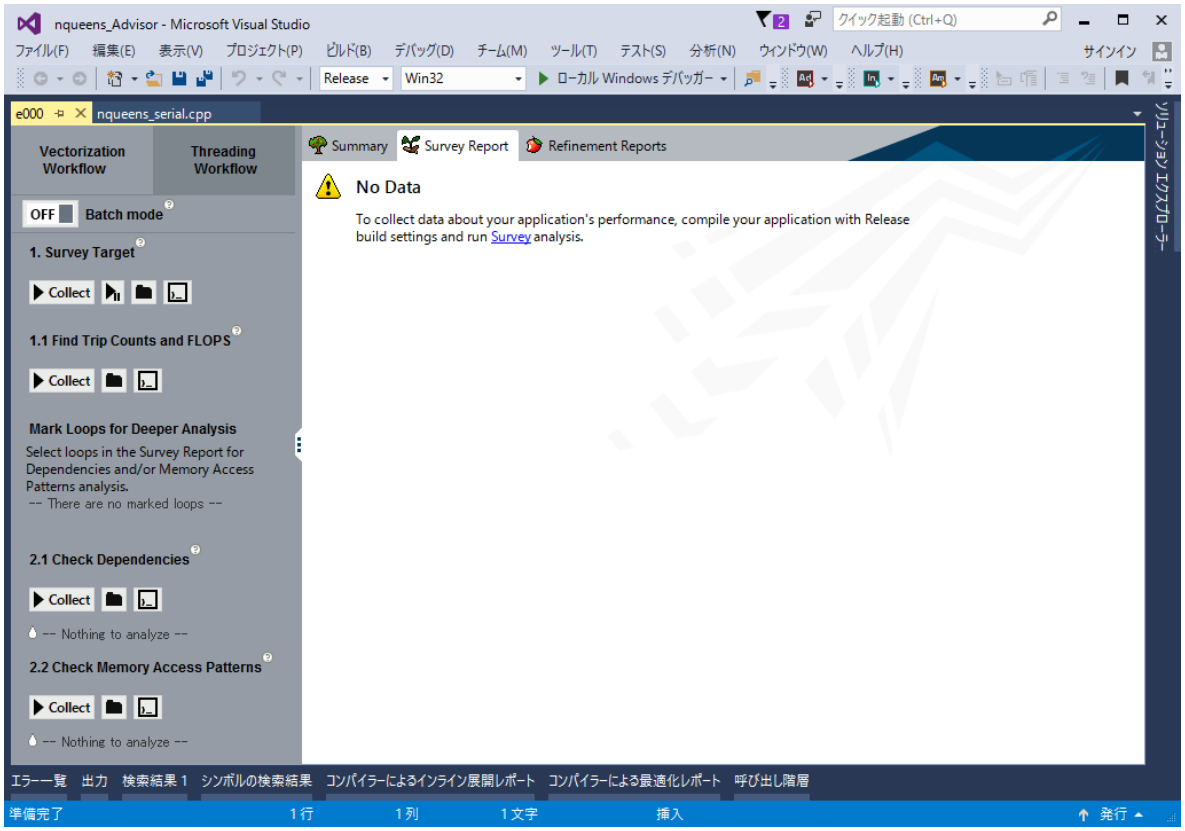
マルチスレッド化の実装にはインテル® Advisor のマルチスレッド化アドバイザーを使用することができます。マルチスレッド化を行うべき処理を特定したり、マルチスレッド化した場合の性能変化をシミュレーションしたりすることで、マルチスレッド化により最も効果的に計算時間を短縮できる処理を発見します。

STEP3　Visual Studio からインテル® Advisor にアクセスする

(1)　Visual Studio の上部メニューより、画像赤枠の Ad と表示されるアイコンを選択します

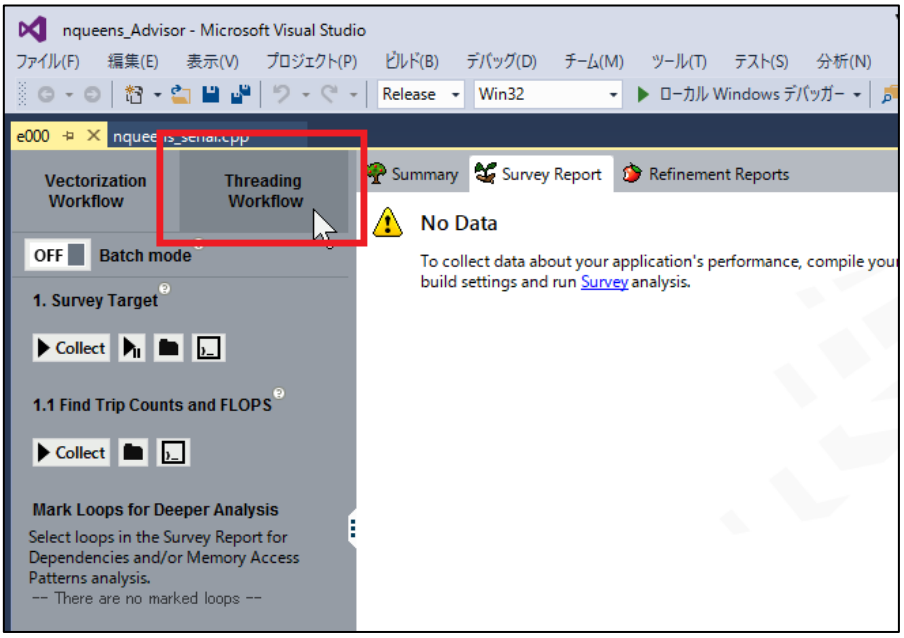


インテル® Advisor が起動します。



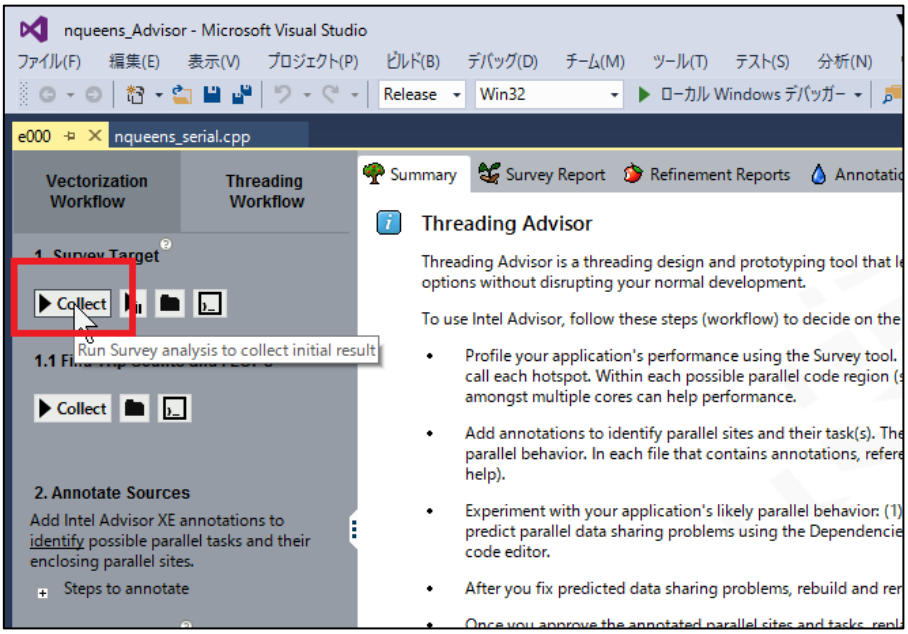
STEP4　プログラム内でマルチスレッド化可能な処理を見つける

(1)　操作を Threading Workflow に切り替えます



起動時は Vectorization Workflow が設定されています。

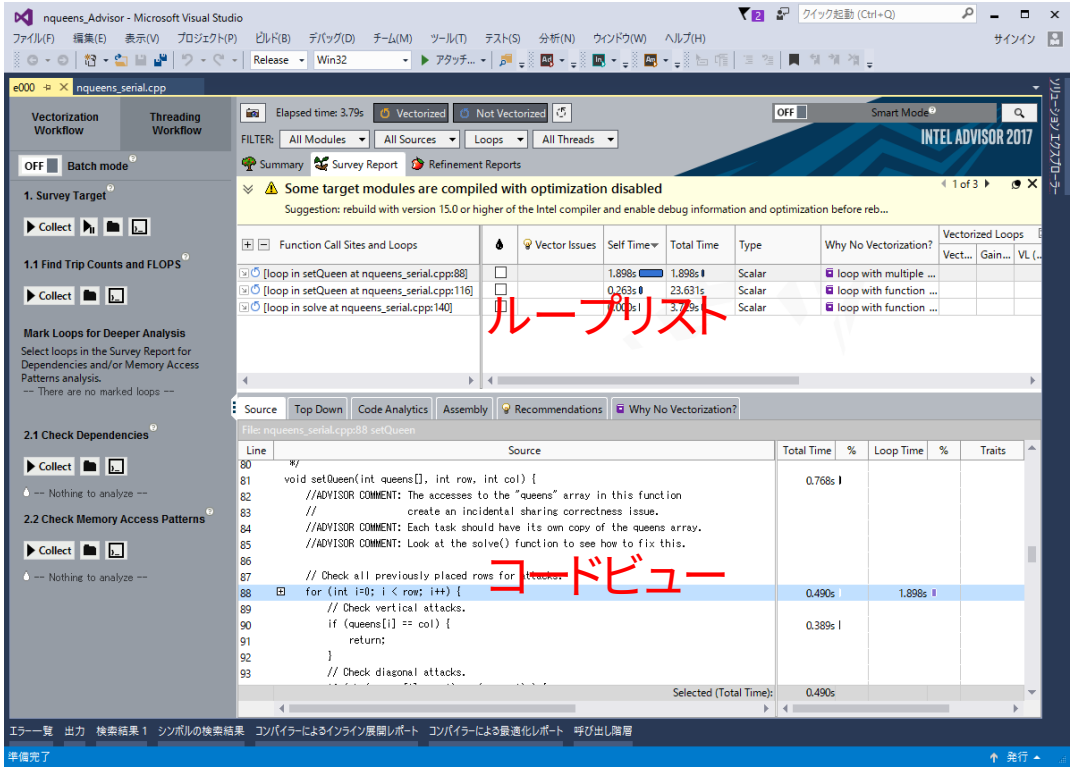
(2)　Survey Target を実行します



インテル® Advisor が指定されたプログラムを解析して、Survey Report が生成されます。

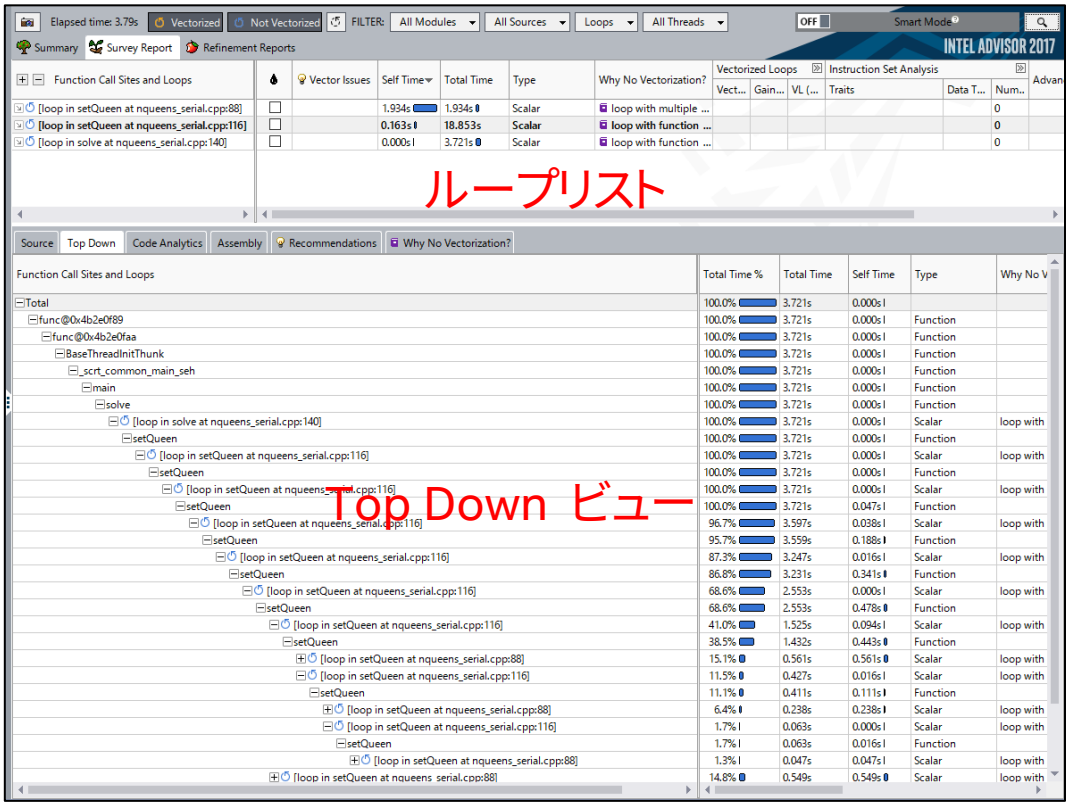
Survey Report の役割 アプリケーションを実行して実行時間が多い処理（Hotspot）を検出し、その Hotspot までの関数コールトレースを表示します。またループ処理が存在する場合はその箇所を別途表示し、並列化箇所の候補として列挙します。

(3) Survey Report を確認します



プログラムが終了すると Survey Target により実行された内容をもとにループ処理や関数に関わる情報をリスト化して Survey Report に表示します。Survey Report にはベクトル化に関する情報もあわせて表示します。

(4) マルチスレッド化を実装すると効果的な処理を予測します



より見やすくするために、Survey Report の表示を大きくしています。マルチスレッド化は一般的にループ処理に実装すると効

果的なため、ループ処理のリストとループ処理であることを示す が表示されている処理に注目します。

ここでは、Threading Workflow のメニューを閉じ、Top Down タブを開いています。

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops		
						Vect...	Gain...	VL (...
[loop in setQueen at nqueens_serial.cpp:88]		1.934s	1.934s	Scalar	loop with multiple ...			
[loop in setQueen at nqueens_serial.cpp:116]		0.163s	18.853s	Scalar	loop with function ...			
[loop in solve at nqueens_serial.cpp:140]		0.000s	3.721s	Scalar	loop with function ...			

func@0x4b2e0f89	100.0%	3.721s
func@0x4b2e0faa	100.0%	3.721s
BaseThreadInitThunk	100.0%	3.721s
_scrt_common_main_seh	100.0%	3.721s
main	100.0%	3.721s
solve	100.0%	3.721s
[loop in solve at nqueens_serial.cpp:140]	100.0%	3.721s
setQueen	100.0%	3.721s
[loop in setQueen at nqueens_serial.cpp:116]	100.0%	3.721s
setQueen	100.0%	3.721s
[loop in setQueen at nqueens_serial.cpp:116]	100.0%	3.721s
setQueen	100.0%	3.721s
[loop in setQueen at nqueens_serial.cpp:116]	96.7%	3.597s
setQueen	95.7%	3.559s
[loop in setQueen at nqueens_serial.cpp:116]	87.3%	3.247s
setQueen	86.8%	3.231s
[loop in setQueen at nqueens_serial.cpp:116]	68.6%	2.553s
setQueen	68.6%	2.553s
[loop in setQueen at nqueens_serial.cpp:116]	41.0%	1.525s
setQueen	38.5%	1.432s
[loop in setQueen at nqueens_serial.cpp:88]	15.1%	0.561s
[loop in setQueen at nqueens_serial.cpp:116]	11.5%	0.427s
setQueen	11.1%	0.411s
[loop in setQueen at nqueens_serial.cpp:88]	6.4%	0.238s
[loop in setQueen at nqueens_serial.cpp:116]	1.7%	0.063s
setQueen	1.7%	0.063s
[loop in setQueen at nqueens_serial.cpp:88]	1.3%	0.047s

マルチスレッド化を実装するにあたり注目するべきは、Self Time が少なく Total Time が大きなループ処理です。

Self Time では対象ループ内に実装された処理を完了させるまでにかかった時間を示しています。ループ処理に関数が含まれる場合、その関数内での処理は Self Time に計上されません。Total Time では対象のループ処理を抜けるまでに消費した時間を示しています。これらの時間はマルチスレッド化を検討する上で重要な項目になります。

また、マルチスレッド化は可能な限り大きな処理に対して行うことも重要です。Top Down 画面から、より上位に位置するループ処理を対象にすると並列処理領域が大きくなるので、マルチスレッド化により高速化されやすいです。

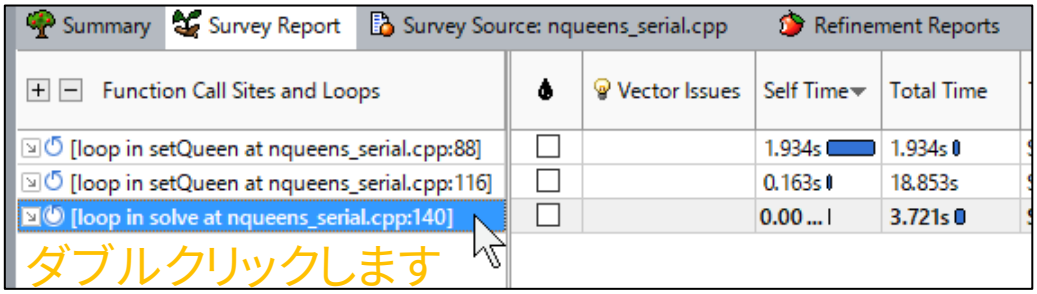
多重ループ処理が複数ある場合にマルチスレッド化すると効果的と予測されるループ処理の主な判断要素

- Self Time が小さい
- Total Time が大きい
- プログラムの上位に位置する
- 呼び出し回数が少ない

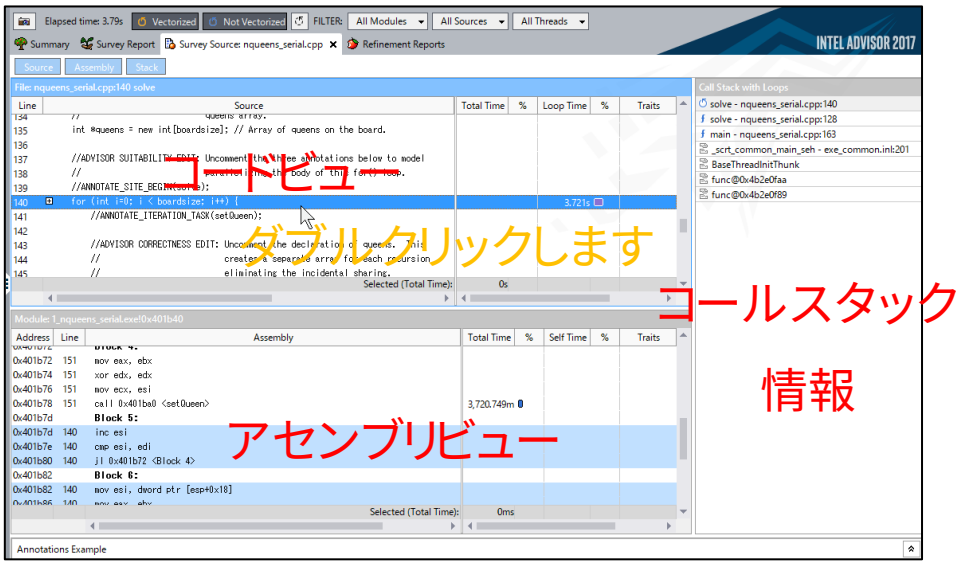
解析結果を確認すると、nqueen_serial.cpp の solve 関数内にあるループ処理(140 行目)がプログラム内で最も上位に位置するループ処理であり、かつ Self Time が少なく Total Time も実行時間に対してある程度大きいことが Survey Report によって確認できます。そのため、140 行目のループ処理を並列処理の対象として進めます。

STEP5　マルチスレッド化による性能変化をシミュレーションする

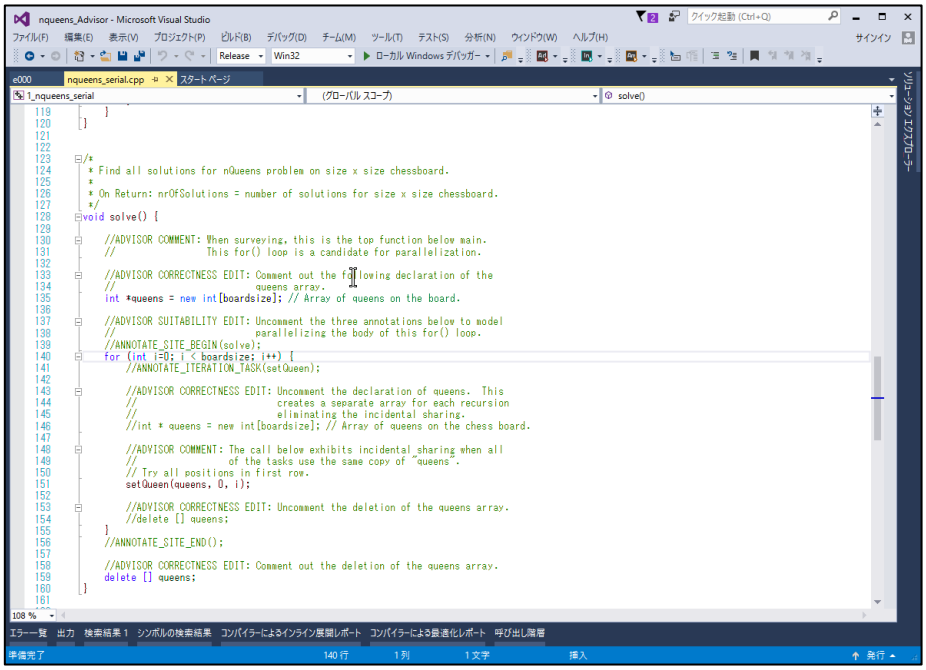
(1)　インテル® Advisor からテキストエディタに移動します



対象のループ処理をダブルクリックすることで、ループ処理に関する情報画面に遷移します。



コードビュー内でダブルクリックすると Visual Studio のテキストエディタに遷移します。



(2) annotation を記述します

```
void solve() {

    //ADVISOR COMMENT: When surveying, this is the top function below main.
    //                 This for() loop is a candidate for parallelization.

    //ADVISOR CORRECTNESS EDIT: Comment out the following declaration of the
    //                          queens array.
    int *queens = new int[boardsize]; // Array of queens on the board.

    //ADVISOR SUITABILITY EDIT: Uncomment the three annotations below to model
    //                          parallelizing the body of this for() loop.
    ANNOTATE_SITE_BEGIN(solve);
    for (int i=0; i < boardsize; i++) {
        ANNOTATE_ITERATION_TASK(setQueen);

        //ADVISOR CORRECTNESS EDIT: Uncomment the declaration of queens.  This
        //                          creates a separate array for each recursion
        //                          eliminating the incidental sharing.
        //int * queens = new int[boardsize]; // Array of queens on the chess board.

        //ADVISOR COMMENT: The call below exhibits incidental sharing when all
        //                 of the tasks use the same copy of "queens".
        // Try all positions in first row.
        setQueen(queens, 0, i);

        //ADVISOR CORRECTNESS EDIT: Uncomment the deletion of the queens array.
        //delete [] queens;
    }
    ANNOTATE_SITE_END();

    //ADVISOR CORRECTNESS EDIT: Comment out the deletion of the queens array.
    delete [] queens;
}
```

annotation の設定

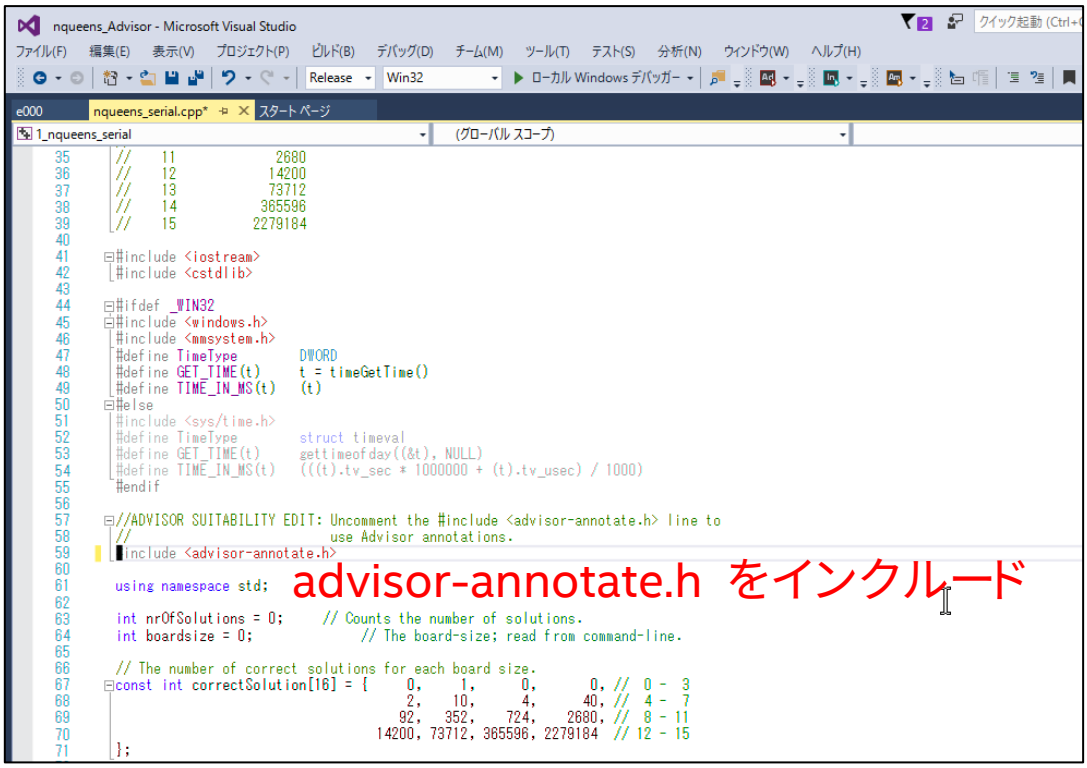
サンプルプログラムの solve 関数には、あらかじめ必要なアノテーションの内容がコメントとして記載されているため、このサンプルプログラムでは3つのコメントを外すことで作業が完了します。

これによりループ処理を並列実行領域、setQueen 関数をタスクとして認識します。この場合、タスクはループの反復回数(boardsize)分存在し、各タスクは同時実行するものと扱われます。

基本的な annotation の設定(単純なループの並列化)

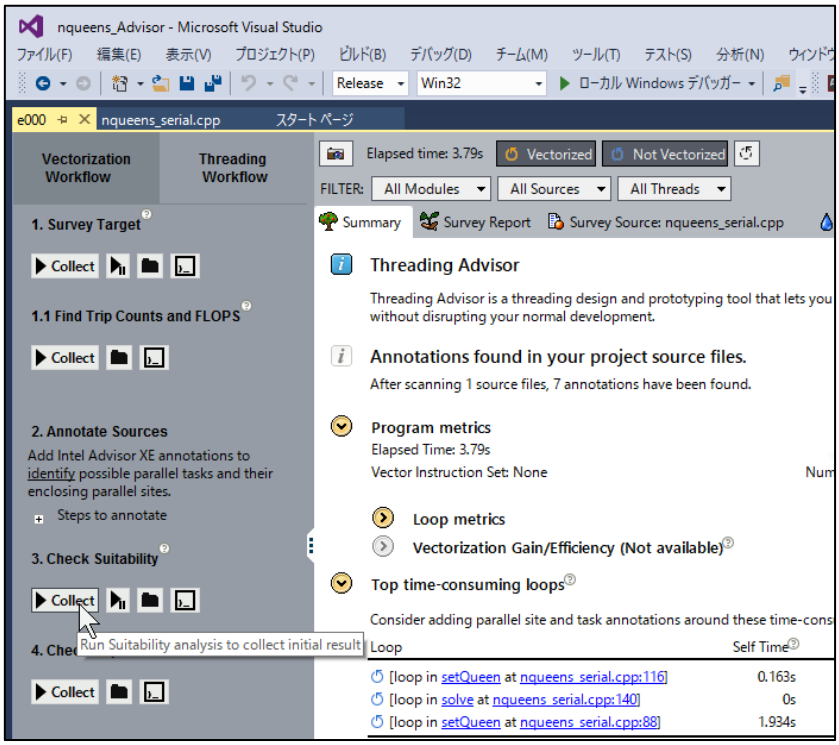
- マルチスレッド化対象のループ文の外側を ANNOTATE_SITE_BEGIN(任意の名前) と ANNOTATE_SITE_END() で囲み、並列実行領域を指定する。
- ループの内側で、その先頭に ANNOTATE_ITERATION_TASK(任意の名前)を付け、ループごとの計算処理をマルチスレッドで分担して同時実行すると仮定する。

(3) annotation を含むコードのコンパイルに必要なヘッダーをインクルードします



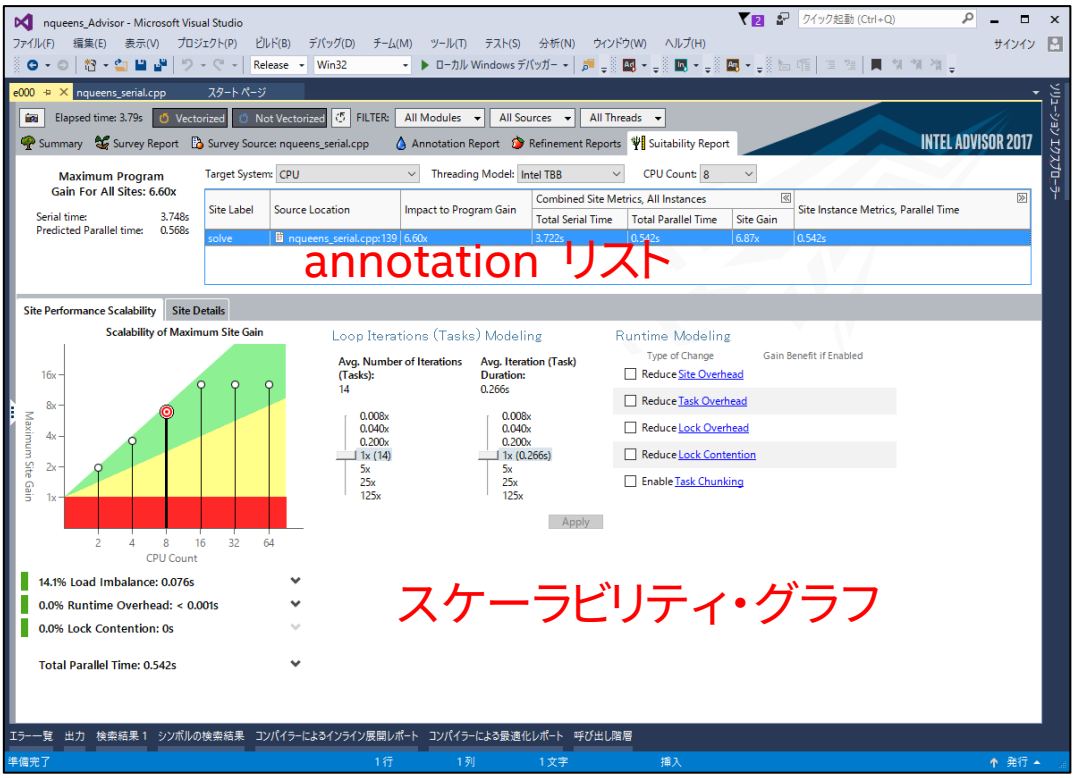
annotation が記述されたコードでは、advisor-annotate.h をインクルードする必要があります。nqueens_serial.cpp ではコメントアウトされているので、コメントを解除します。ビルドが正常に完了するか確認してください。

(4) Check Suitability を実行します

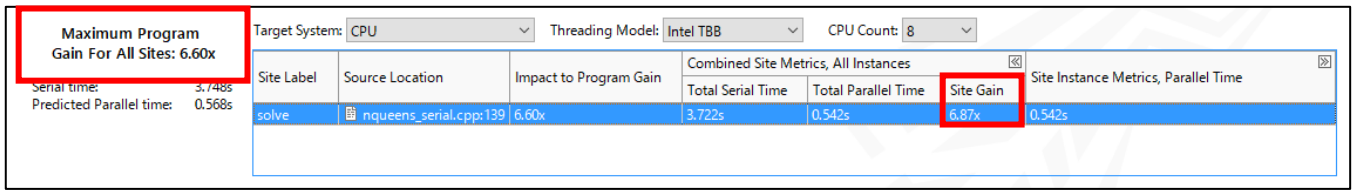


annotation が設定された箇所をマルチスレッド化して処理した場合をシミュレートして、Suitability Report が生成されます。

(5) Suitability Report を確認します



Suitability Report では設定した annotation ごとのパフォーマンス情報(annotation リスト)やコア数の増加に対する性能向上割合を示すスケーラビリティ・グラフについての情報が表示されます。この情報をもとに annotation を設定した処理がマルチスレッド化された場合の性能変化を確認します。



サンプルプログラムではプログラム全体の性能向上を確認すると 6.60 倍になることが予測されます。また、annotation を設定したループ処理自体の性能は 6.87 倍になることが予測されています。この情報から solve 関数内にあるループ処理を並列化するだけで、実行時間をかなり短縮できることが考えられます。

STEP6　OpenMP*を使用してマルチスレッド処理を実装する

インテル® Advisor を使用してマルチスレッド化により性能向上が得られるであろう処理を特定したので、実際にマルチスレッド化を OpenMP* を使用して実装します。

OpenMP*を使用したマルチスレッド化について

OpenMP* を使用するには、まず以下のように宣言子を使用して並列実行領域を定義します。

```
#pragma omp parallel
{
   /* 並列実行領域 */
   <処理>
}
```

並列実行領域に記述されたコードは複数のスレッドによって実行される処理となります。例えば、並列実行領域に printf() がある場合、生成された各スレッドはそれぞれ printf() を実行し、複数の printf 出力が表示されます。デフォルトの設定で生成されるスレッド数は OS が認識するコア数と同じ数です。

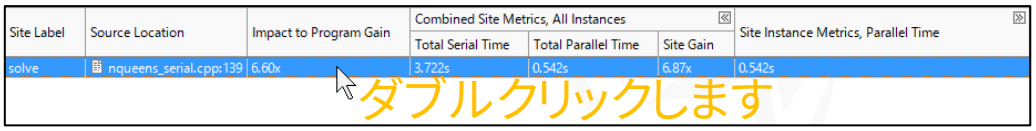
ループ処理を OpenMP により複数のスレッドに分担させるには、さらにワークシェアリング構文を使用します。ワークシェアリング構文は、並列実行領域内で実行される分担可能な処理をスレッド間で分担させます。分担方法については、特に指定がない限りループ回数を自動的に分配します。ここでは C/C++ の for ループに対応する for 宣言子を使用します。

```
#pragma omp for        // 並列実行領域内で実行されていると
for(i=0; i<N; i++)    // 0~N までの計算をスレッド間で自動分配する
{
      <処理>
}
```

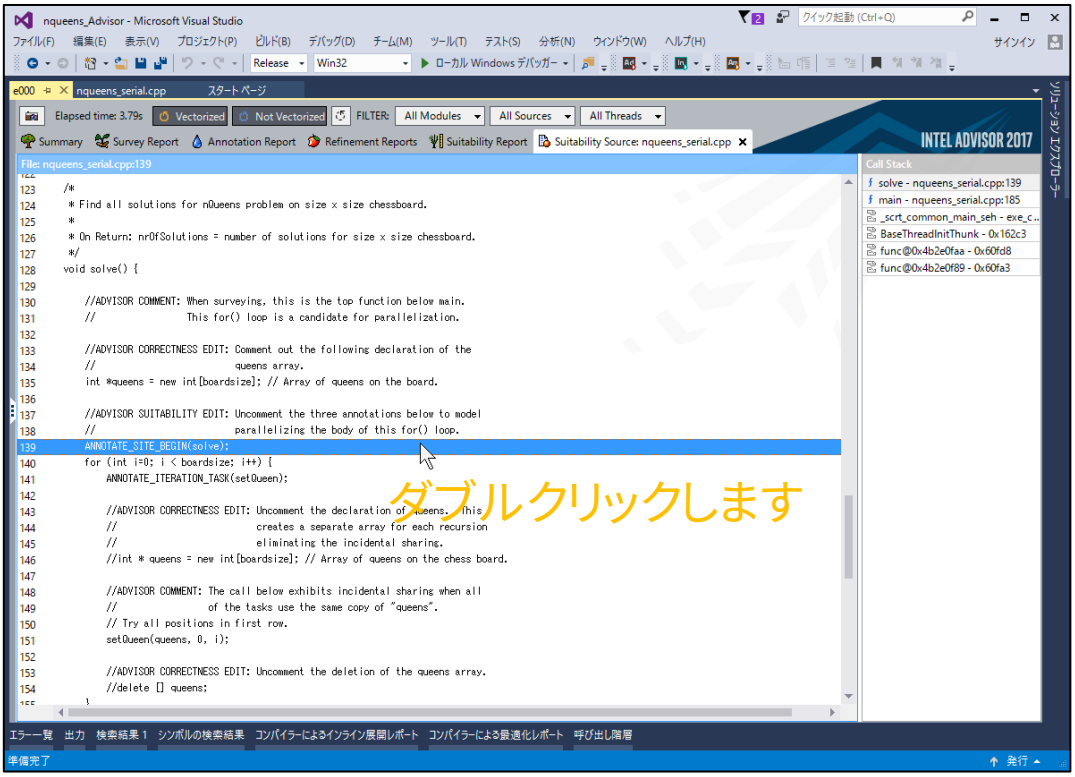
2つの宣言子は組み合わせて指定できます。

```
#pragma omp parallel for
for(i=0; i<N; i++)    // 0~N までの計算をスレッド間で自動分配し、処理内容を並列に実行
{
      <処理>
}
```

(1)　マルチスレッド化対象の処理に移動します



Suitability Report の annotation リストをダブルクリックします。



annotation が設定されているコードビューに移動できるので、コードビュー内でダブルクリックします。

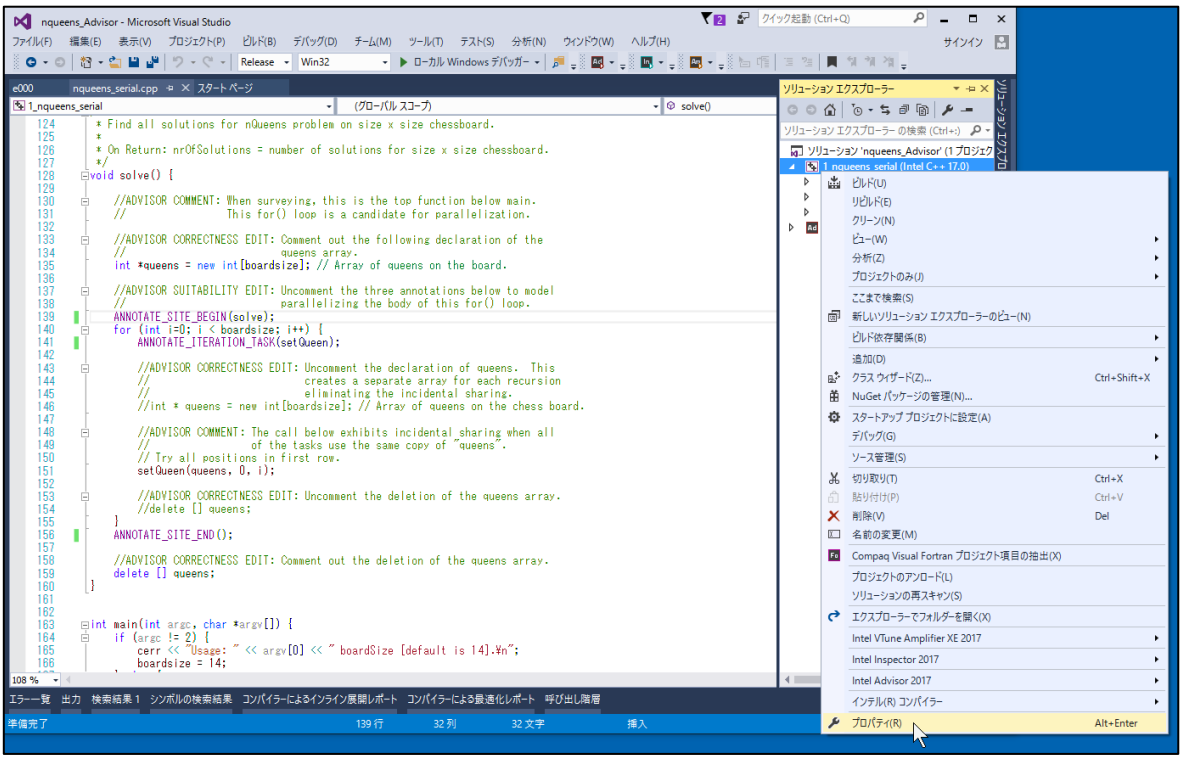
(2)　マルチスレッド処理を実装します

```
void solve() {
        int * queens = new int[boardsize];

#pragma omp parallel for
        for(int i=0; i<boardsize; i++) {
                // try all positions in first row
                setQueen(queens, 0, i);
        }
}
```

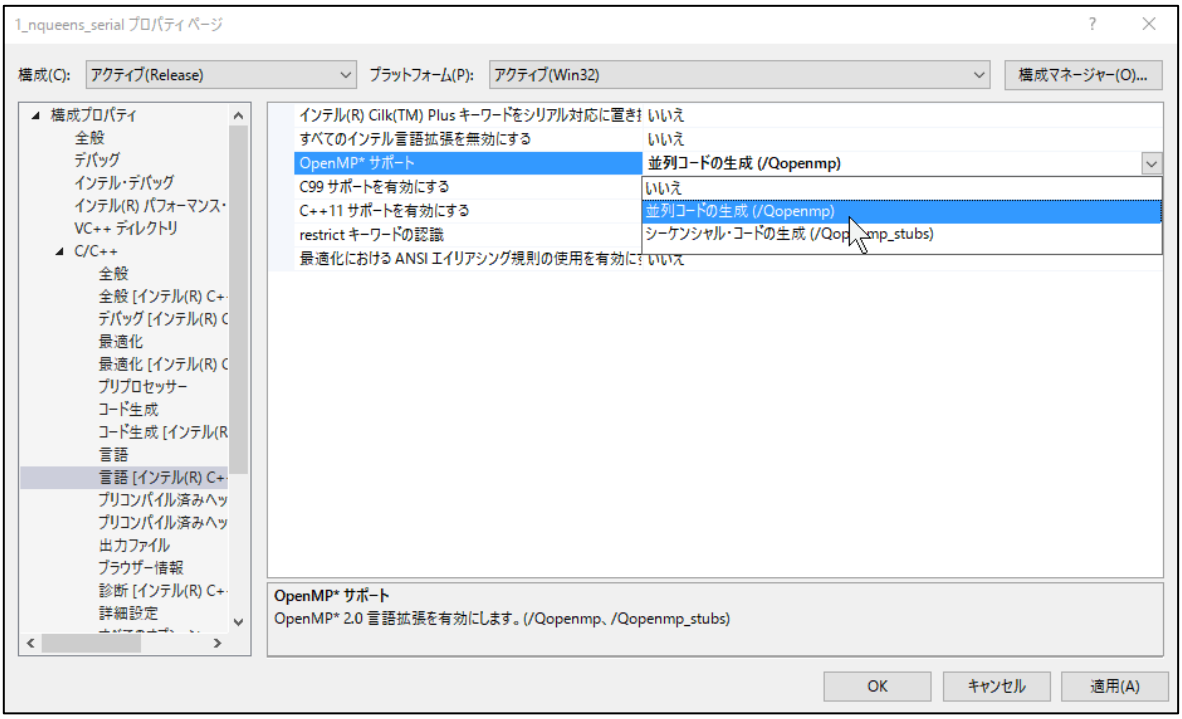
140 行目のループ処理をマルチスレッド化します。

(3) OpenMP を認識させるためのプロパティを設定します

インテル®コンパイラーのデフォルト設定では OpenMP を認識しません。Visual Studio のプロジェクト・プロパティから OpenMP を使用するためにコンパイラーオプションを設定します。

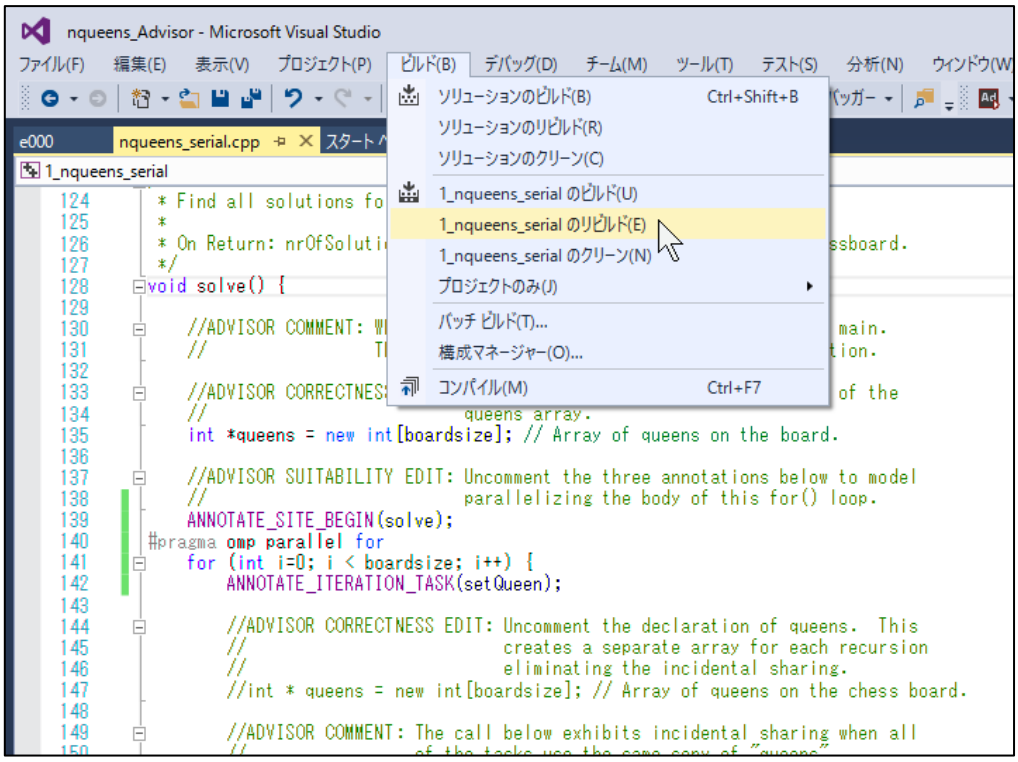


1_nqueens_serial を右クリックしてプロパティ(R)を選択します。



[構成プロパティ] → [C/C++] → [言語[インテル(R) C++]] → [OpenMP*サポート]の項目を並列コードの生成 (/Qopenmp) に変更します。

(4) プログラムを実行します



リビルドして実行します。

サンプルプログラムの計算結果は 365596 でしたが、マルチスレッド化を実装したことで異なる計算結果が表示されました。さらに、このプログラムは実行するたびに異なる計算結果を出力します。

```
Number of solutions: 1167631
!!Incorrect result!! Number of solutions should be 365596
```

```
Number of solutions: 858763
!!Incorrect result!! Number of solutions should be 365596
```

```
Number of solutions: 1294112
!!Incorrect result!! Number of solutions should be 365596
```

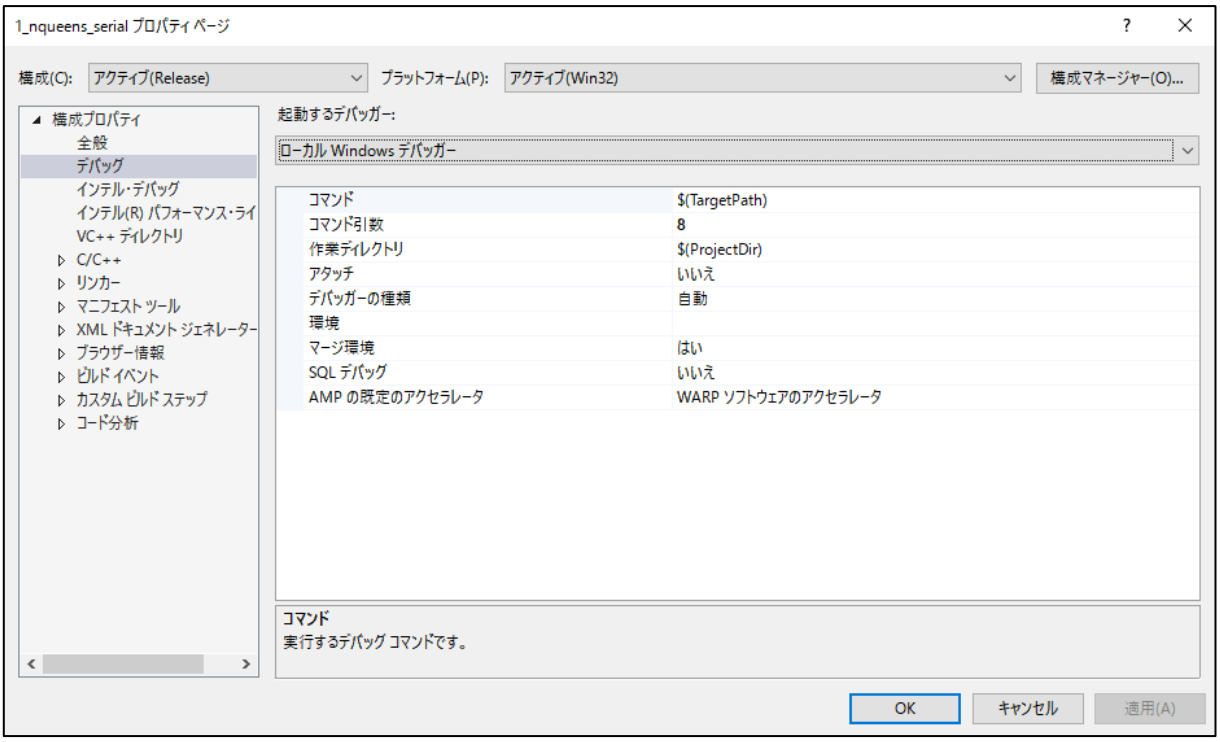
4-3 実行の度に異なる計算結果を改善する

マルチスレッド化による計算時間の短縮はできましたが、計算結果が実行する毎に異なる現象が発生しました。これはマルチスレッド化した際に発生する特有の現象の一つです。インテル®Inspector はマルチスレッド化によって発生した、プログラム内に潜在する問題を改善するための情報を提供します。

STEP7　インテル® Inspector を使用するための準備

(1)　与えるデータ数を少なくします

インテル® Inspector を使用した解析には、対象アプリケーションの実行時間の数十倍から数百倍の大きなオーバーヘッドがかかります。サンプルプログラムでは実行時に 14 の値を指定して計算させていますが、解析にかかる時間を抑えるため、一時的に 8 を指定します。

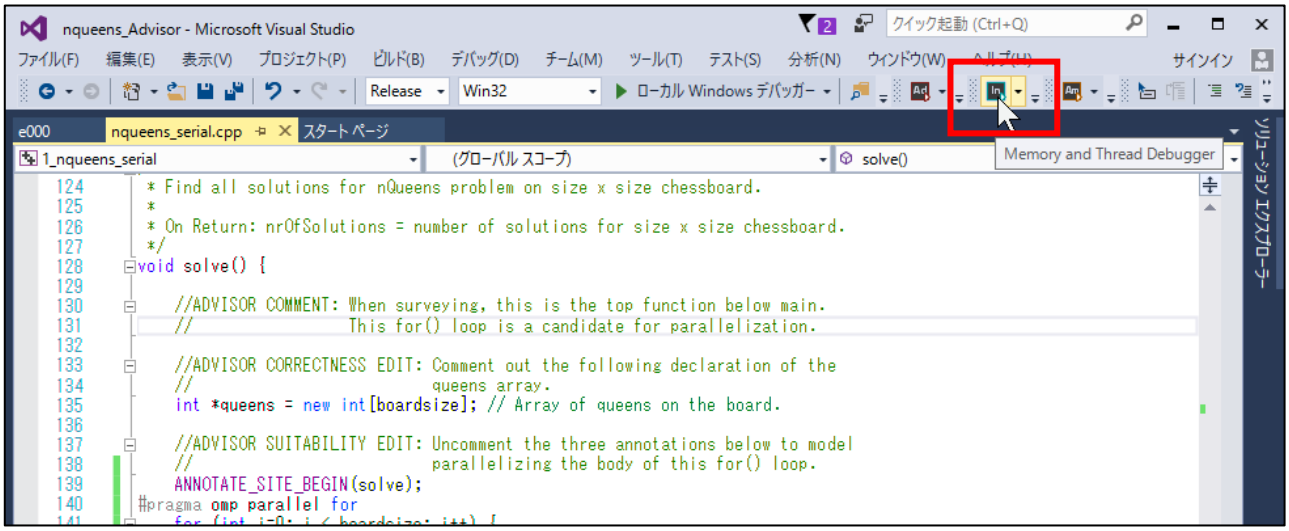


インテル® Inspector で解析する場合に効果的なアプローチ

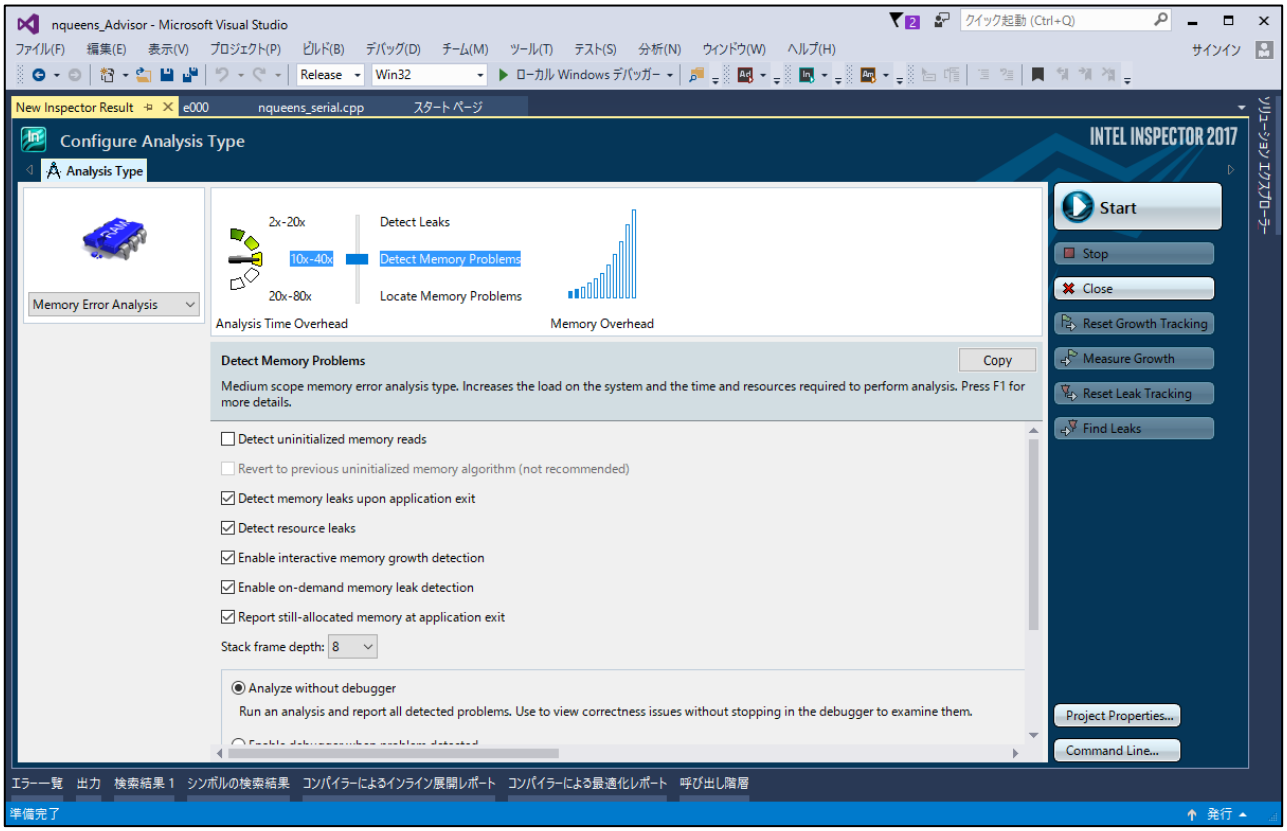
- マルチスレッド化された処理を切り離して個別に解析する
- 計算に使用するデータ数を少なくする

STEP8　Visual Studio からインテル® Inspector にアクセスする

(1)　Visual Studio の上部メニューより赤枠の In と表示されるアイコンを選択します

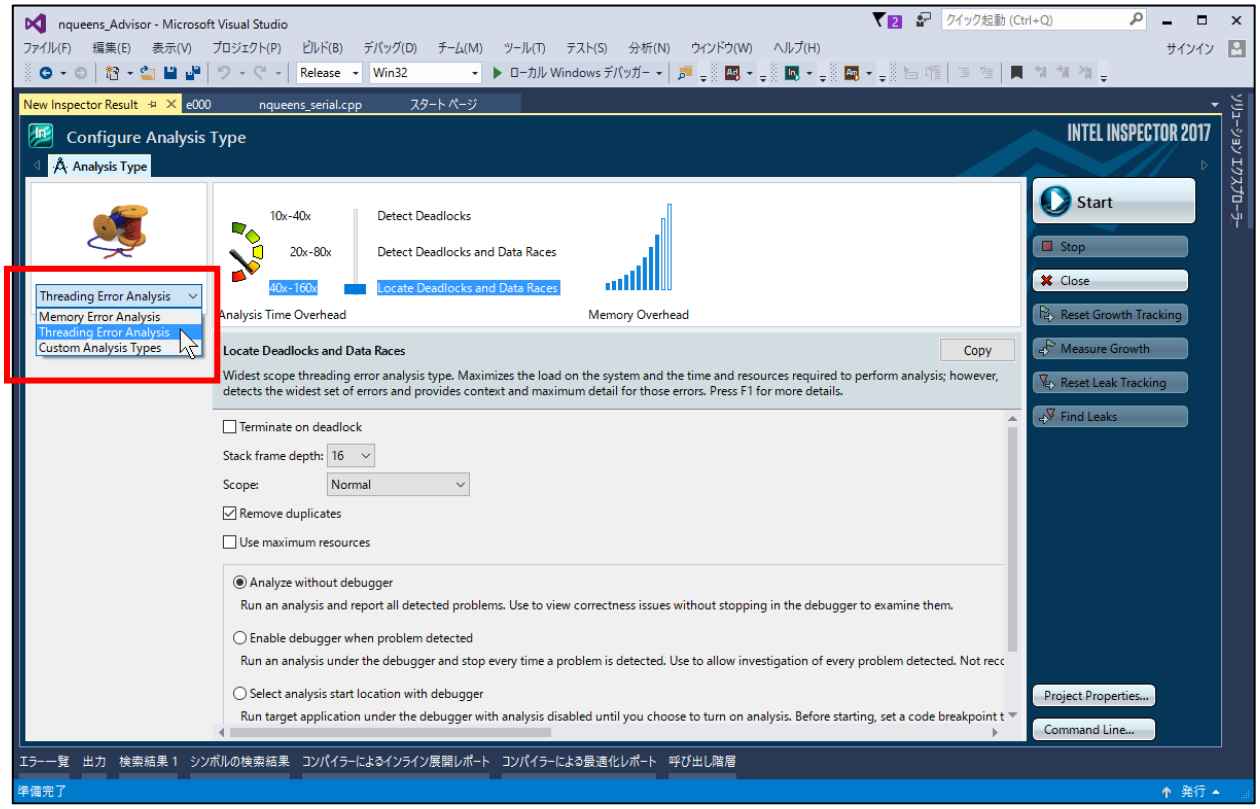


インテル® Inspector が起動します。

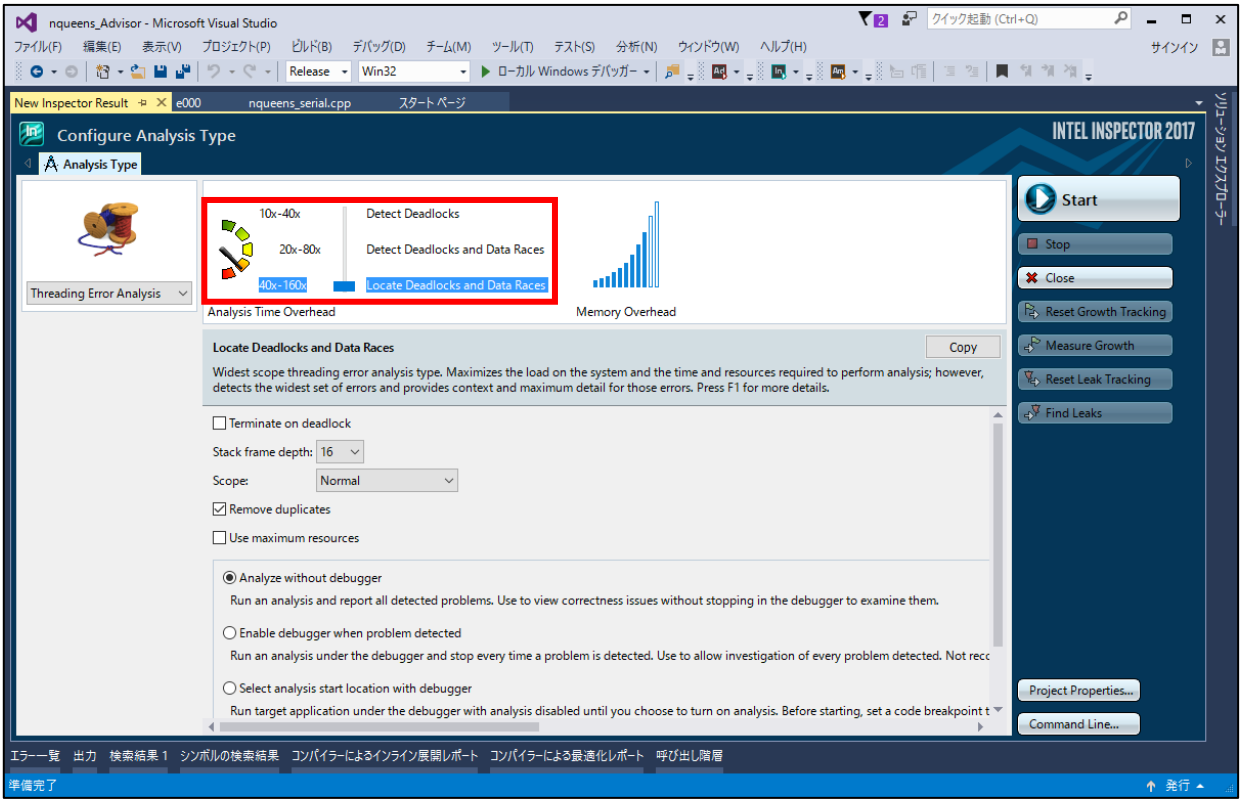


STEP9　スレッドエラー解析を実行する

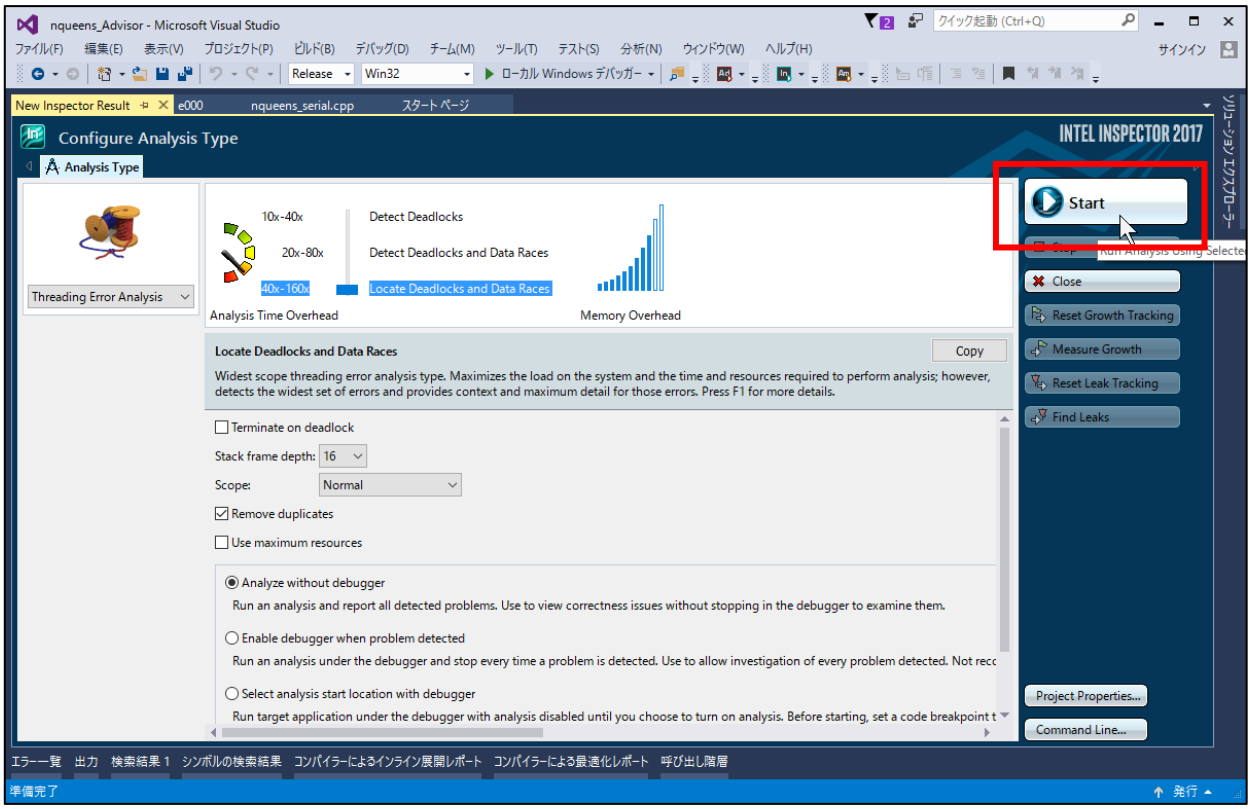
(1)　Threading Error Analysis に切り替えます



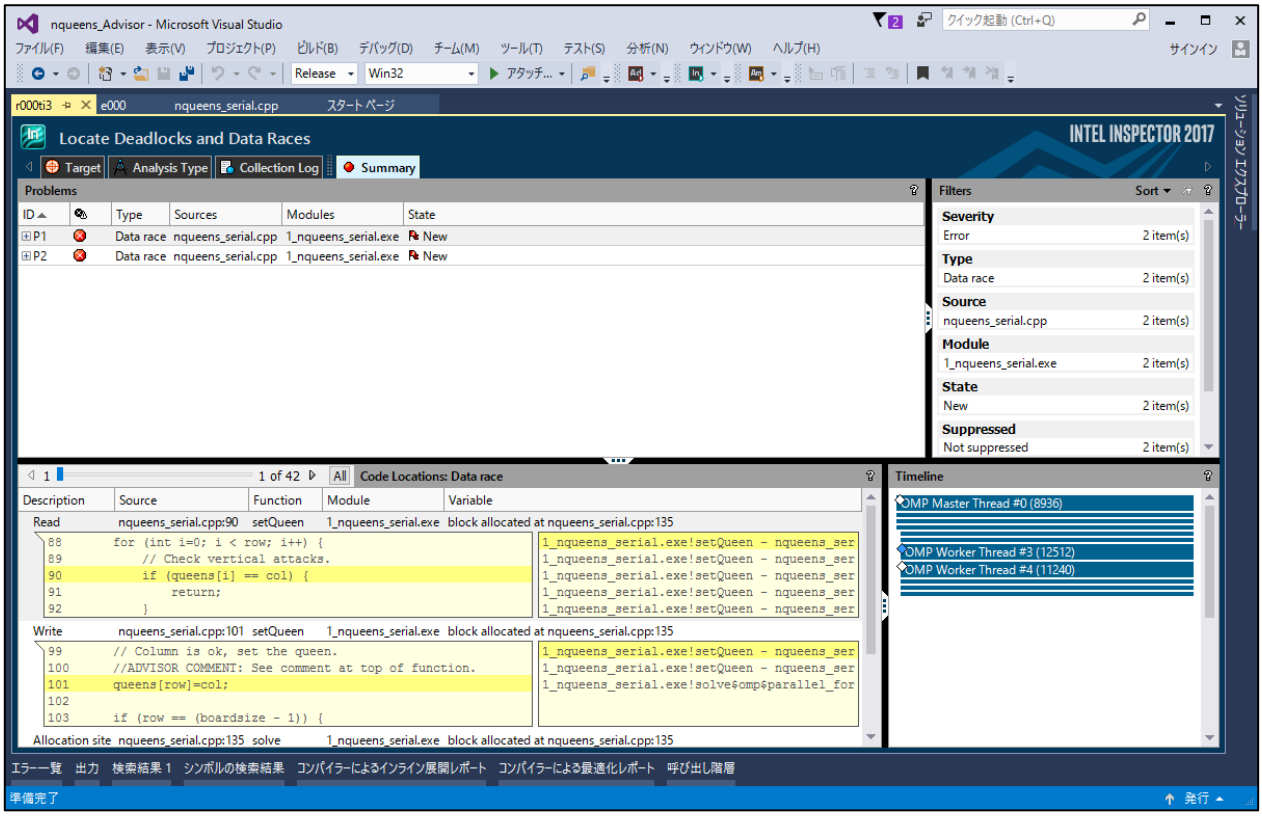
(2)　Analysis Time Overhead を Locate Deadlocks and Data Races に設定します



(3) 解析を開始します

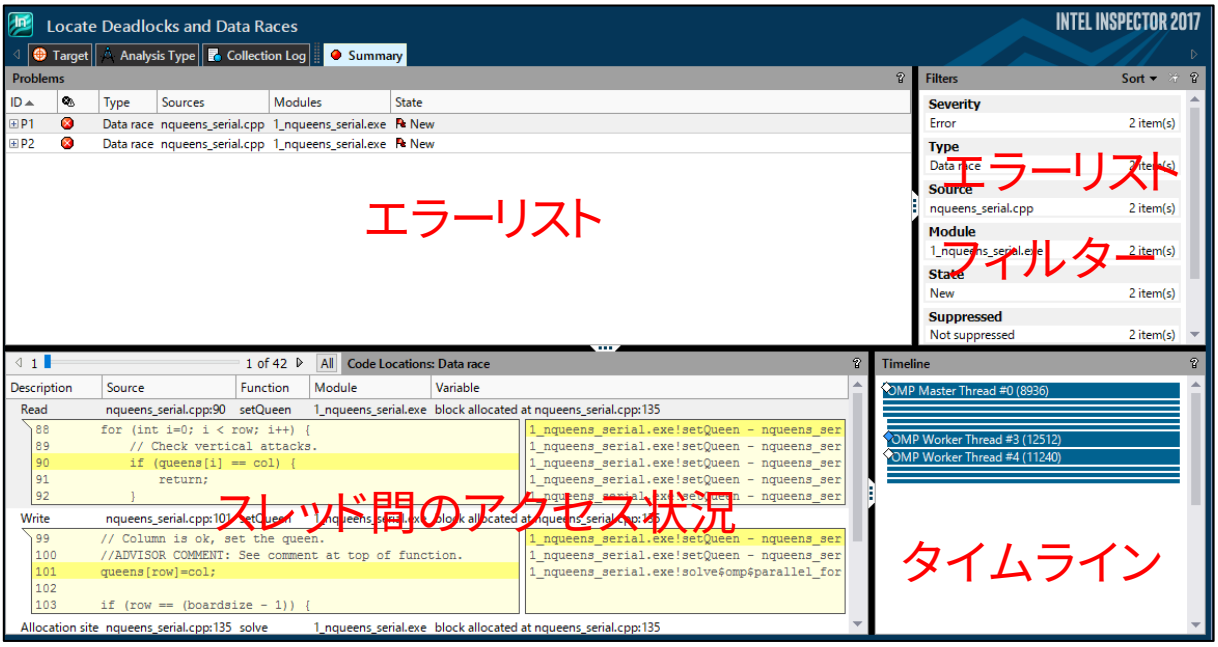


インテル® Inspector が指定されたプログラムを実行して Summary を表示します。



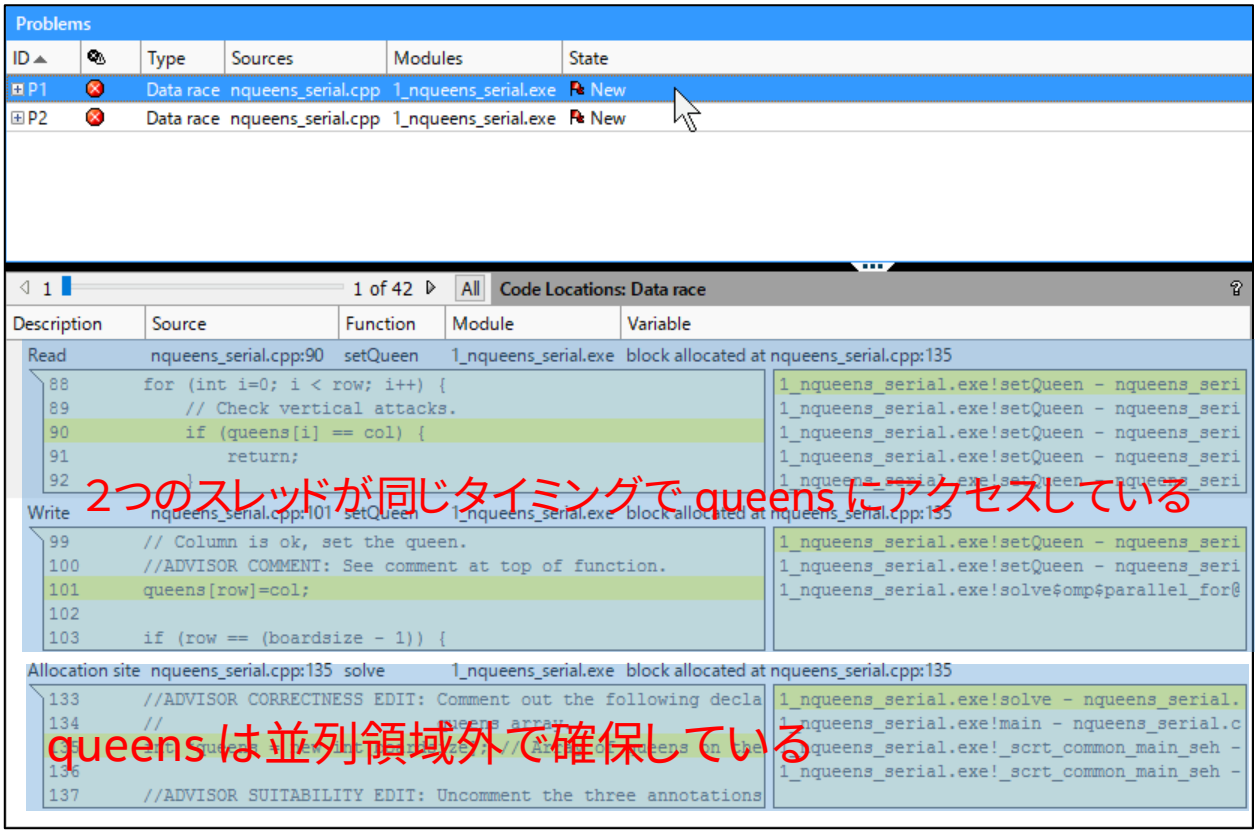
STEP10 計算結果が異なる原因を特定する

(1) インテル® Inspector が指摘する問題を確認する



インテル® Inspector の Summary には問題と推定されるエラーをリストとして表示します。エラーリストの Type 項目には、P1 と P2 の Data race (データ競合)が発生していると表示されており、この2つのエラーに注目して計算結果が異なる問題の原因を特定します。

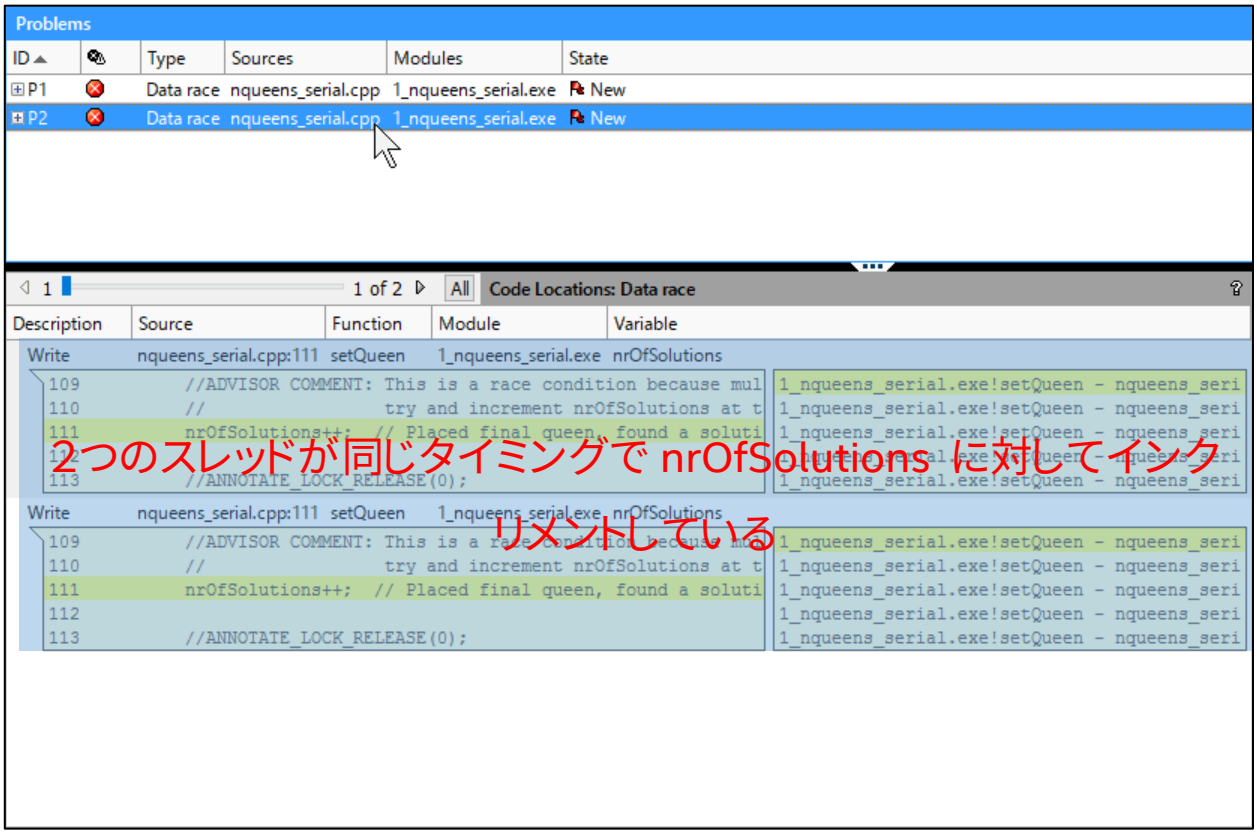
(2) Data race (データ競合)が発生している原因を特定する



P1 のエラーについてスレッド間のアクセス状況を確認します。スレッドのアクセス状況を見ると、queens に対してハイライトされており、setQueen 関数内で 2 つのスレッドが同じタイミングで queens に対して Read (読み込み)と Write (書き込み)を行っていることが確認できます。

サンプルプログラムでは queens は計算結果を求めるための一時的な作業領域として使用されています。Summary にはさらに queens のメモリ確保は並列領域外で行われていることが表示されており、P1 の原因として全てのスレッドが一か所の queens に対して読み込み/書き込みが行える状態であることを示しています。

続いて P2 の問題について原因を特定します。



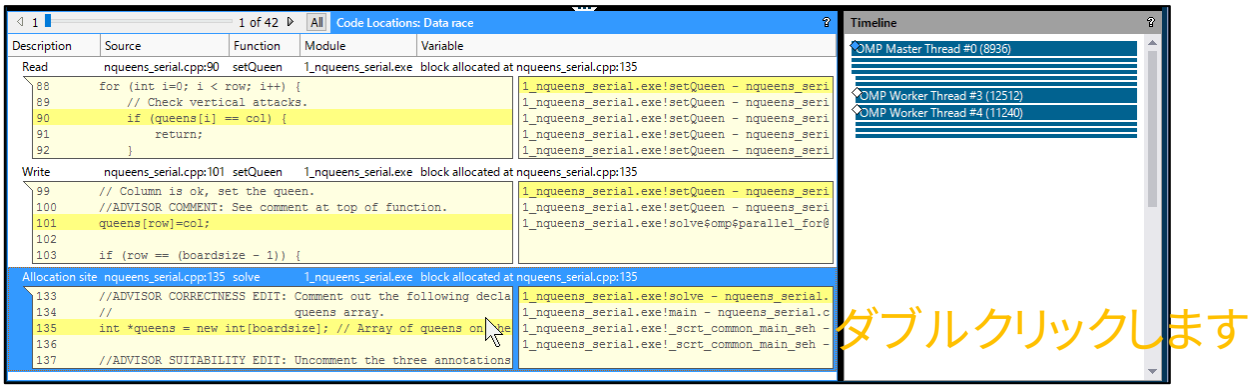
P2 のスレッド間のアクセス状況を確認します。ここでは nrOfSolutions に対して2つのスレッドが同時にインクリメントを処理していることを示しています。

サンプルプログラムでは nrOfSolutions は setQueen() 関数で使用され、計算結果を格納するための変数です。計算しながら値を更新していく必要がありますが、複数のスレッドによって同時に書き込み処理が行われると、計算結果は正しく反映されません。

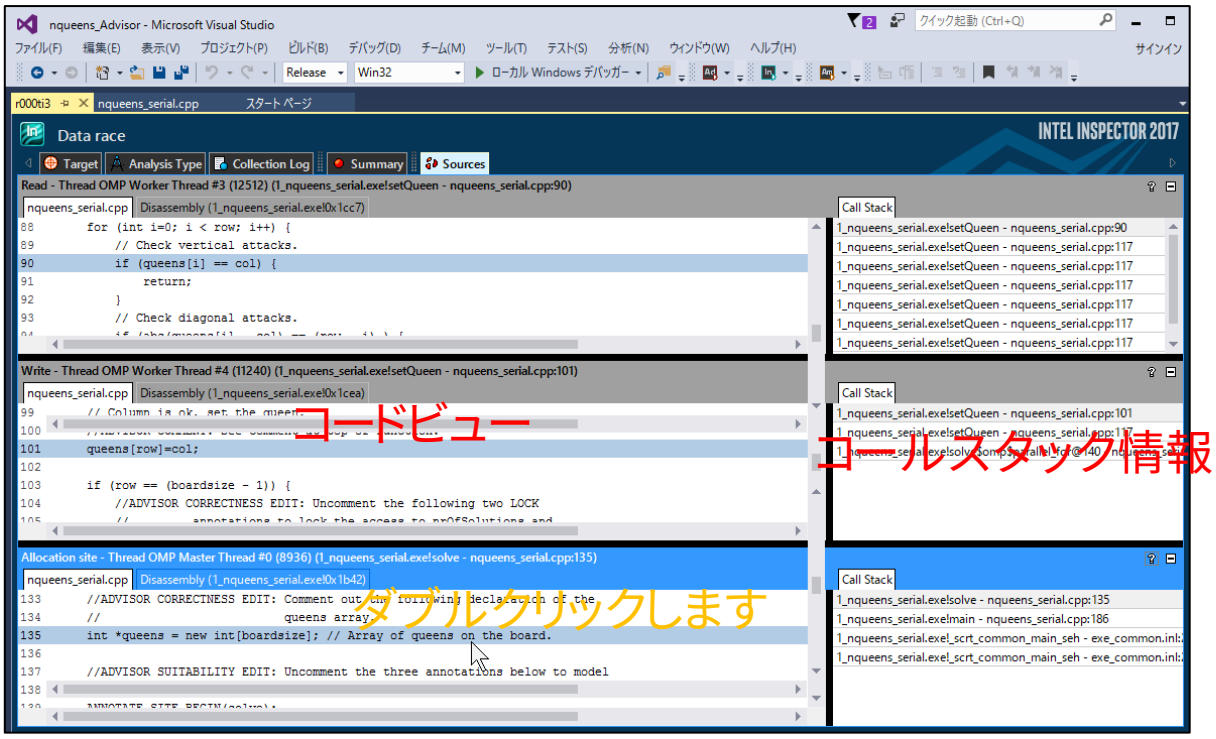
STEP11 ソースコードを変更して改善する

インテル® Inspector によって計算結果が実行毎に異なる原因を特定したので、ソースコードを修正します。

(1)　インテル® Inspector からテキストエディタに移動します



Summary に表示されている queens を確保している処理をダブルクリックします。



さらに queens を確保する処理をダブルクリックして Visual Studio のテキストエディタが開きます。

(2) queens を各スレッドで確保するように変更する

```
136
137     //ADVISOR SUITABILITY EDIT: Uncomment the three annotations below to model
138     //                          parallelizing the body of this for() loop.
139     ANNOTATE_SITE_BEGIN(solve);
140   #pragma omp parallel for
141     for (int i=0; i < boardsize; i++) {
142         ANNOTATE_ITERATION_TASK(setQueen);
143
144         //ADVISOR CORRECTNESS EDIT: Uncomment the declaration of queens.  This
145         //                          creates a separate array for each recursion
146         //                          eliminating the incidental sharing.
147         int * queens = new int[boardsize]; // Array of queens on the chess board.
148
149         //ADVISOR COMMENT: The call below exhibits incidental sharing when all
150         //                 of the tasks use the same copy of "queens".
151         // Try all positions in first row.
152         setQueen(queens, 0, i);
153
154         //ADVISOR CORRECTNESS EDIT: Uncomment the deletion of the queens array.
155         delete [] queens;
156     }
157     ANNOTATE_SITE_END();
158
159     //ADVISOR CORRECTNESS EDIT: Comment out the deletion of the queens array.
160     // delete [] queens;
161 }
```

メモリの確保/解放を

並列領域内に記述します

queens はスレッド単位で独立したポインタ領域（スレッドローカルな変数）にすればデータの競合は発生しません。solve() 関数内の for ループ文の内側に定義することで、ループごとに queens を確保して setQueen 関数の入力引数として渡します。

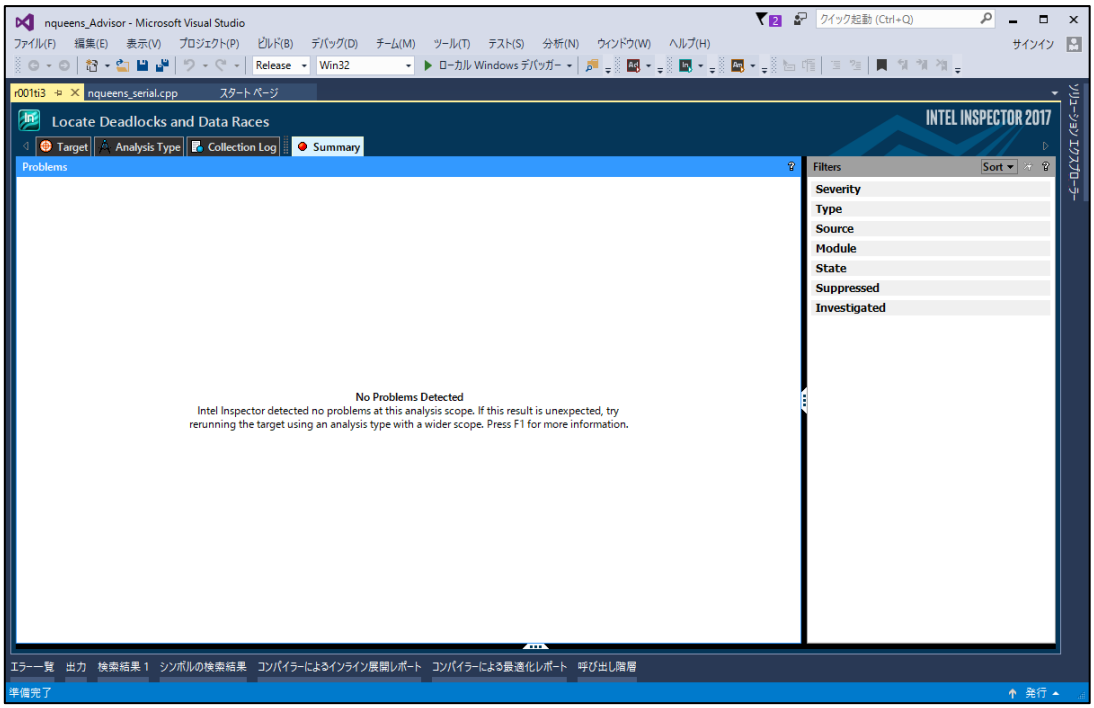
(3) nrOfSolutions を排他アクセスに設定する

```
 96         }
 97     }
 98
 99     // Column is ok, set the queen.
100     //ADVISOR COMMENT: See comment at top of function.
101     queens[row]=col;
102
103     if (row == (boardsize - 1)) {
104         //ADVISOR CORRECTNESS EDIT: Uncomment the following two LOCK
105         //         annotations to lock the access to nrOfSolutions and
106         //         eliminate the race condition.
107         //ANNOTATE_LOCK_ACQUIRE(0);
108   #pragma omp atomic
109         //ADVISOR COMMENT: This is a race condition because multiple tasks may
110         //                 try and increment nrOfSolutions at the same time.
111         nrOfSolutions++;  // Placed final queen, found a solution!
112
113         //ANNOTATE_LOCK_RELEASE(0);
114     } else {
115         // Try to fill next row.
116         for (int i=0; i < boardsize; i++) {
117             setQueen(queens, row+1, i);
118         }
119     }
120 }
121
122
```

nrOfSolutions に排他処理を行う

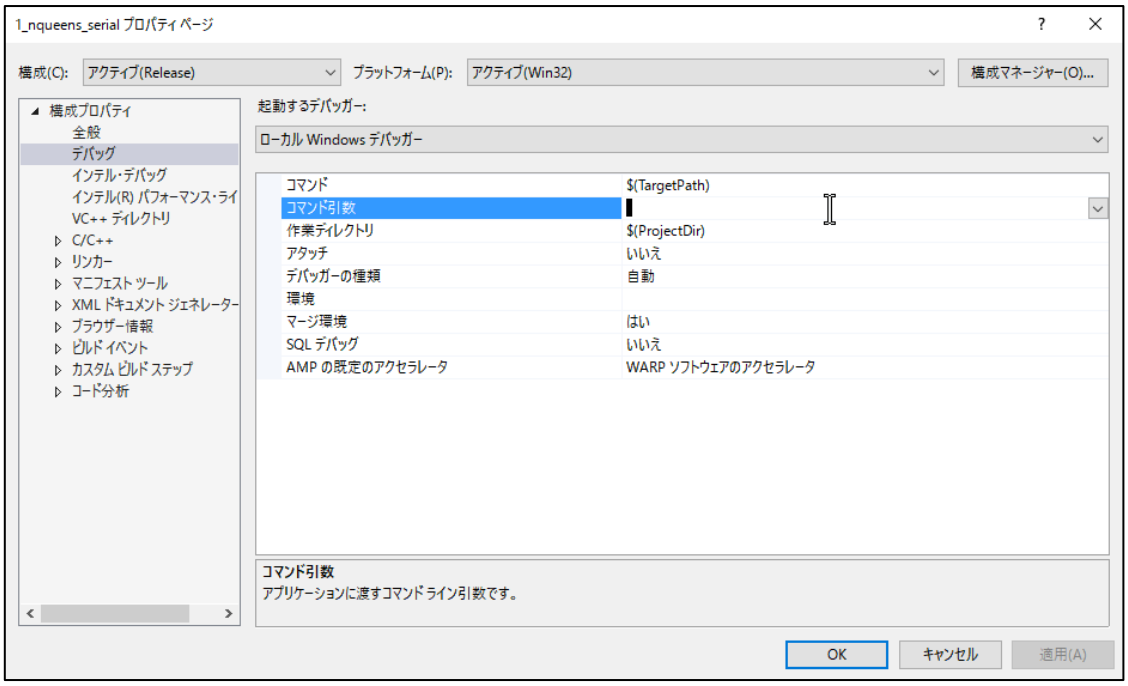
排他処理では常に単一のスレッドのみが対象の処理を実行できます。これにより、実行中のスレッドが nrOfSolutions のインクリメント処理を完了するまで、他のスレッドは nrOfSolutions をインクリメントできません。OpenMP が提供する排他処理構文にはいくつかありますが、ここでは atomic 宣言子を適用します。

(4)　再度インテル® Inspector のスレッドエラー解析を行い潜在するエラーがあるか確認します

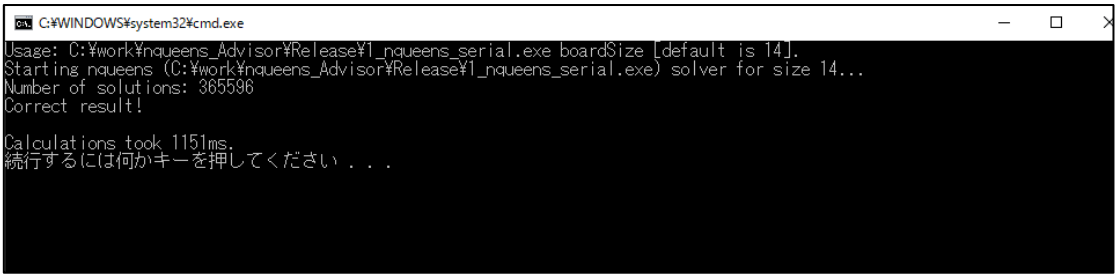


最初の解析結果で表示されていた P1 と P2 の問題が解決されインテル®Inspector からスレッドに関するエラーがないことがわかりました。

(5)　コマンド引数の値を削除して実行します



指定していた 8 を削除します。



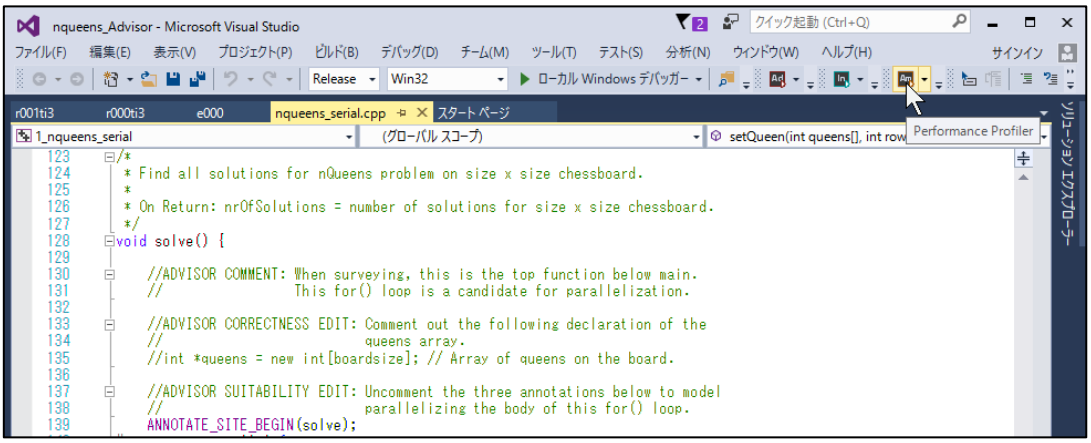
マルチスレッド化を行う前と同じ計算結果の 365596 が得られました。また、マルチスレッド化を行う前と比較して計算時間が短縮されています。

4-4 マルチスレッド化された処理の性能を検証する

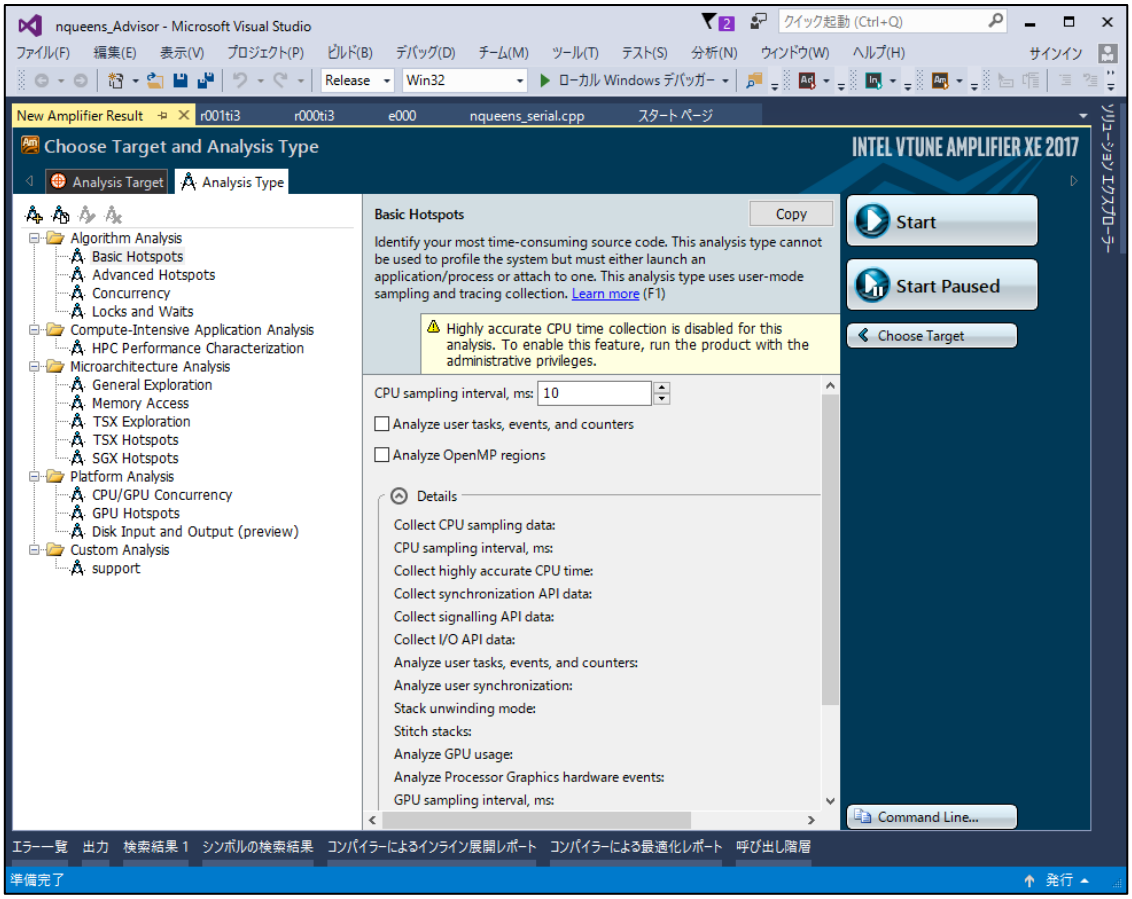
効果的なマルチスレッド処理が行われているかどうかインテル® VTune™ Amplifier XE を使用して確認します。CPU が持つ複数のコアでプログラムのスレッドが効率よく計算できているのか確認することで、さらに計算時間を短縮できるかどうか判断します。

STEP12 Visual Studio からインテル® VTune™ Amplifier XE にアクセスする

(1) Visual Studio の上部メニューより赤枠の Am と表示されるアイコンを選択します

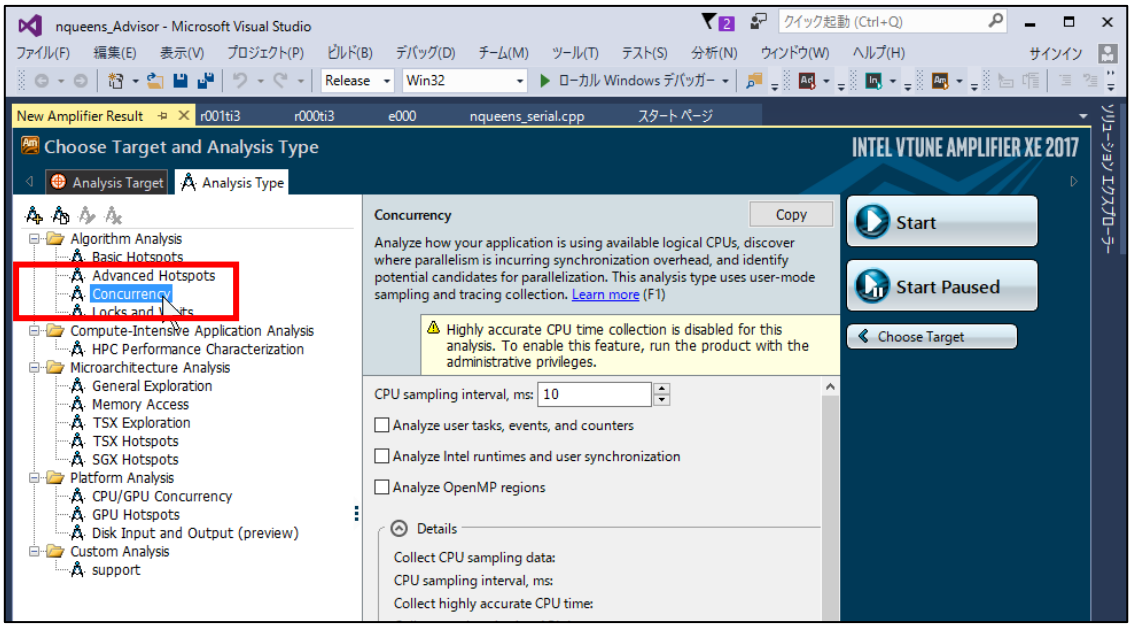


インテル® VTune™ Amplifier XE が起動します。



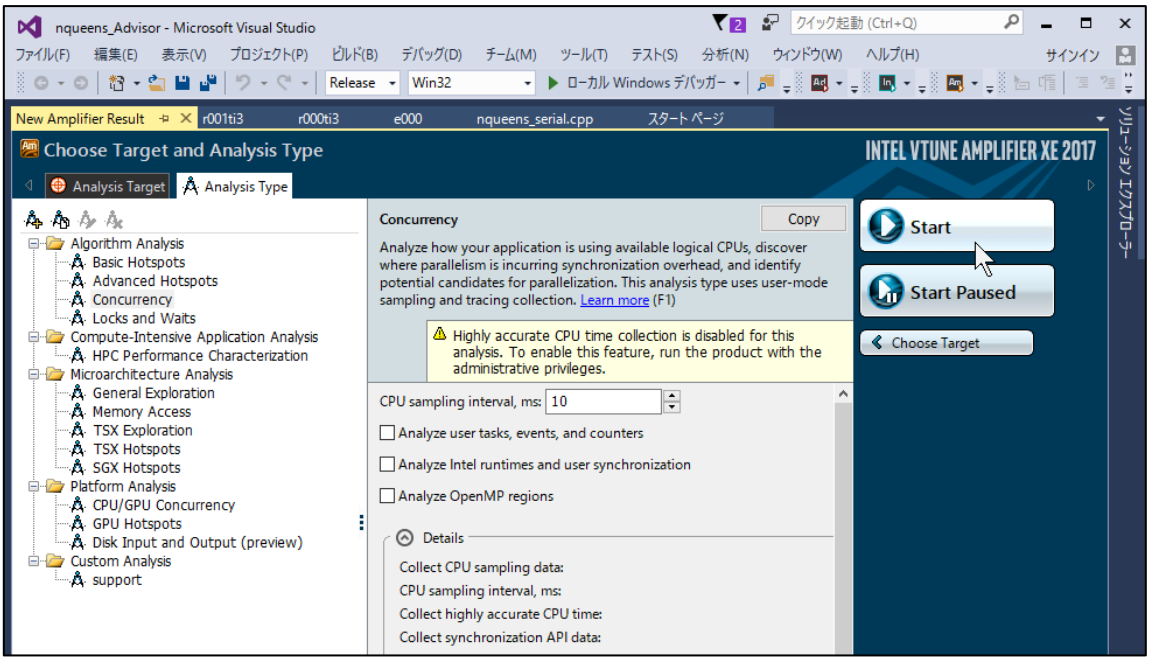
STEP13 並列性に注目して解析する

(1) 解析プリセットから Concurrency 解析を選択します

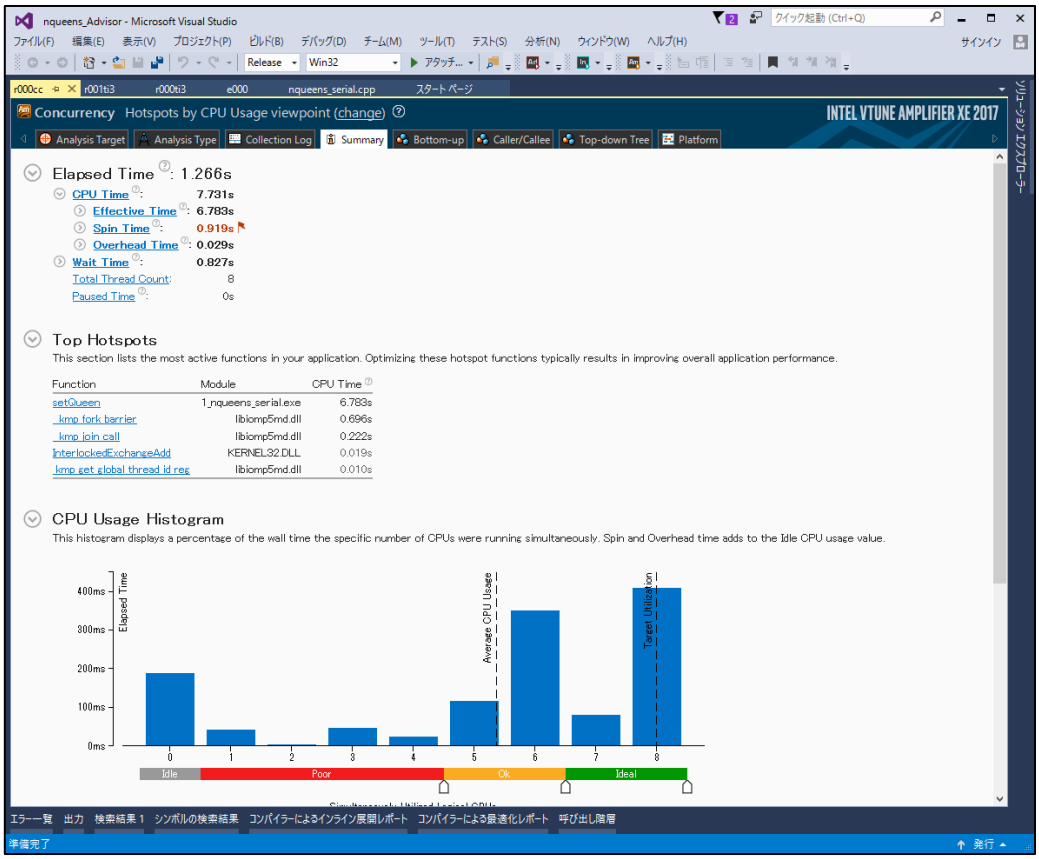


このサンプルプログラムのマルチスレッド化による動作状況を確認するために、解析プリセットの一覧から Concurrency 解析を選択します。

(2) Concurrency 解析を実行します

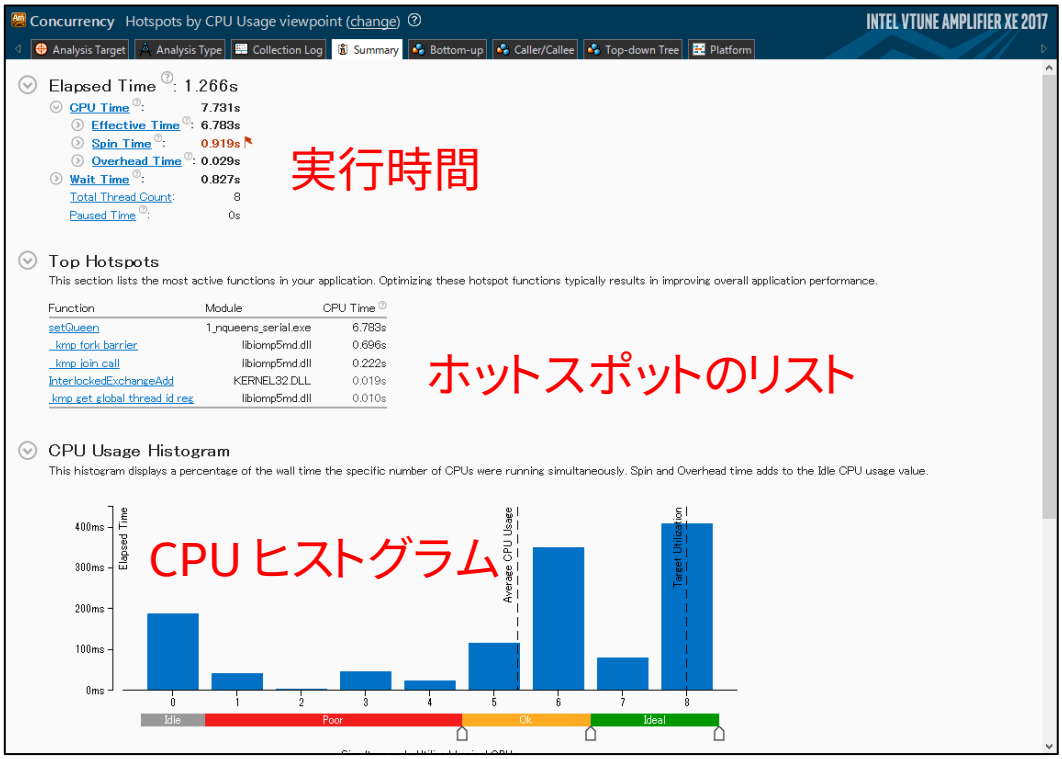


Start をクリックして解析を開始します。対象のサンプルプログラムが実行され、インテル® VTune™ Amplifier XE がサンプリングを行います。サンプルプログラムの終了後、解析結果が表示されます。



STEP14 解析結果から実行状況を確認する

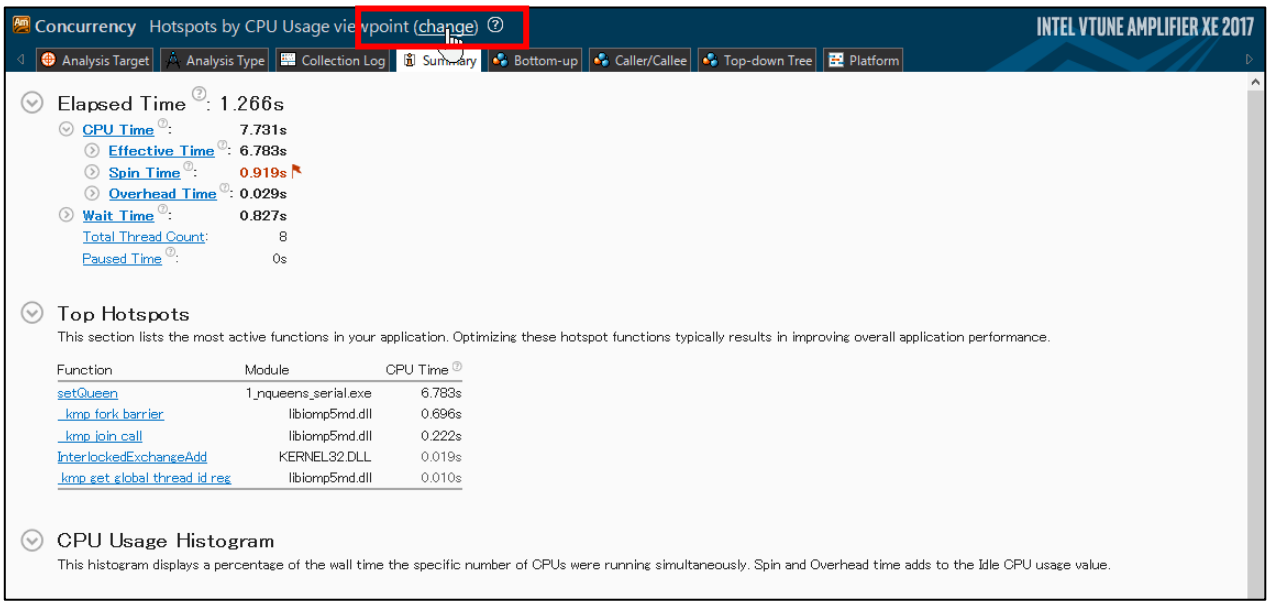
(1) Summary を確認します



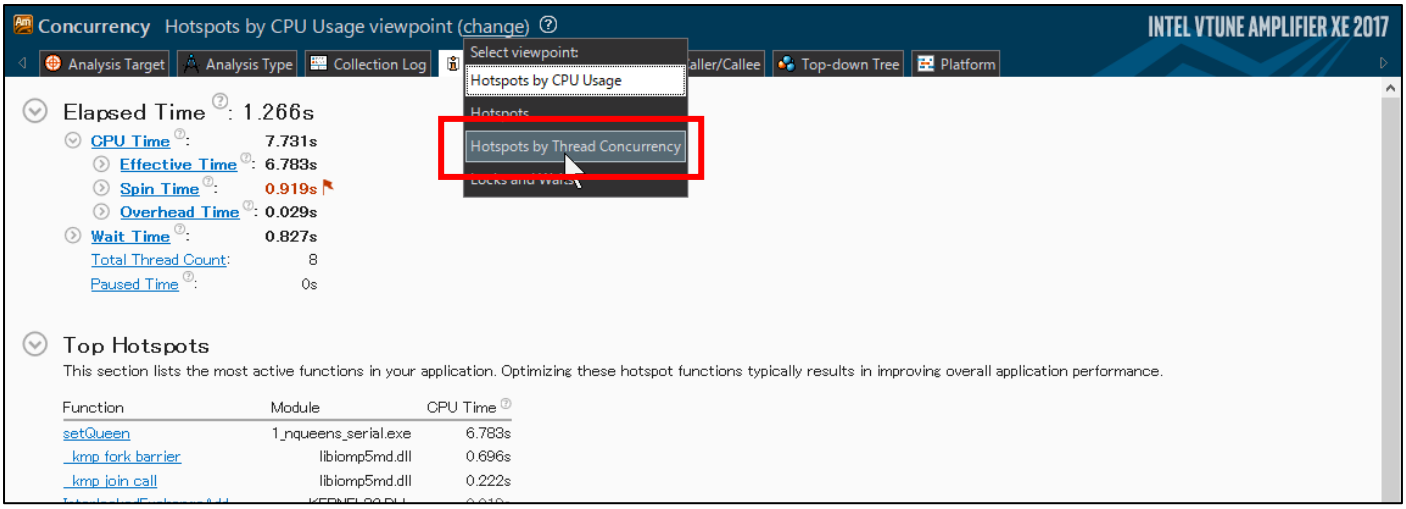
まずプログラムの実行時間や CPU 時間、Hotspot 関数を確認するための Summary が表示されます。

(2) viewpoint を変更します

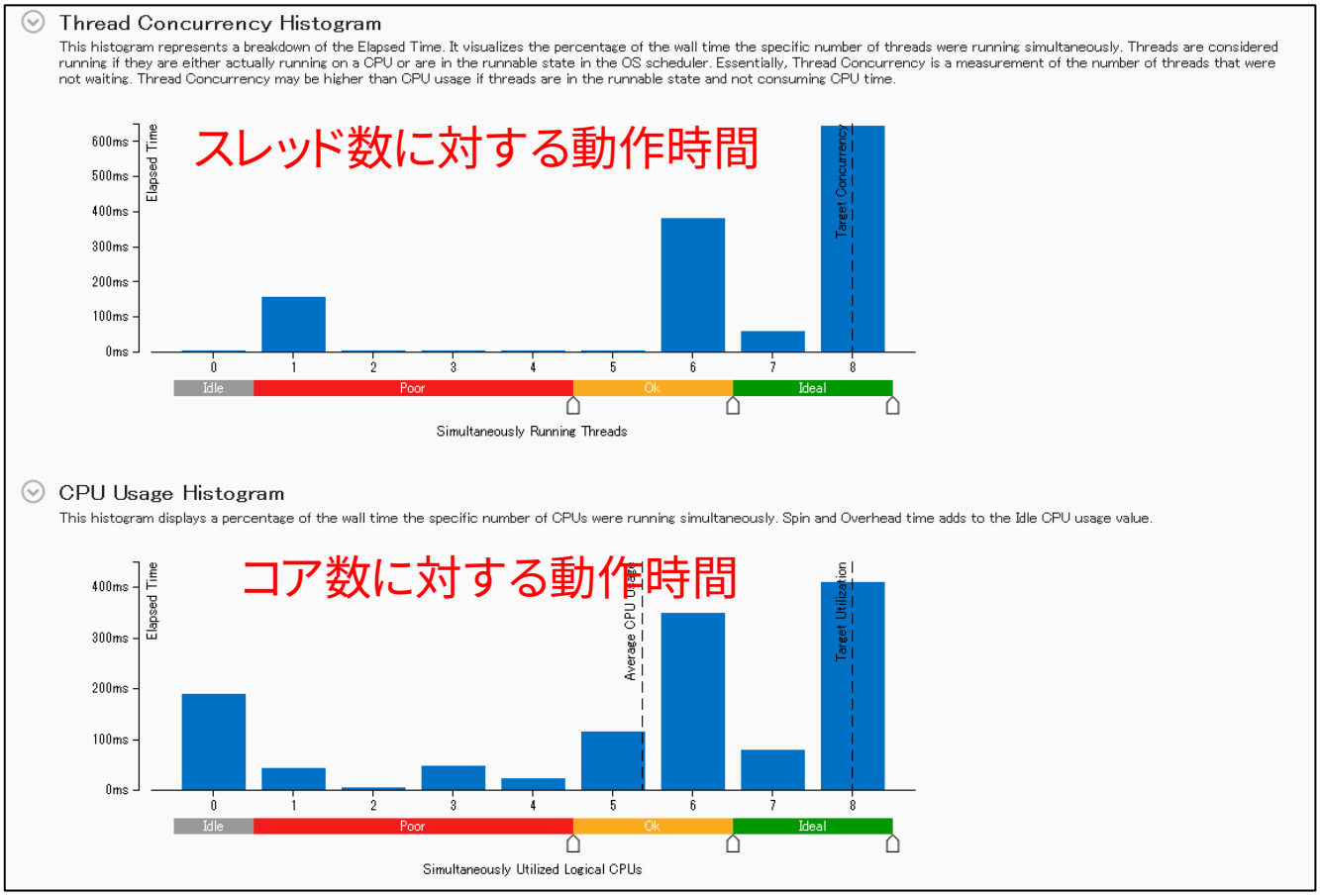
スレッドの実行状況を確認するために表示形式を変更します。



Hotspots by CPU Usage viewpoint を Hotspots by Thread Concurrency に変更します。これにより CPU の動作状況ベースではなく、スレッドの実行状況をベースにして結果を確認できます。



(3) CPU の利用率とスレッドの動作状況を確認します



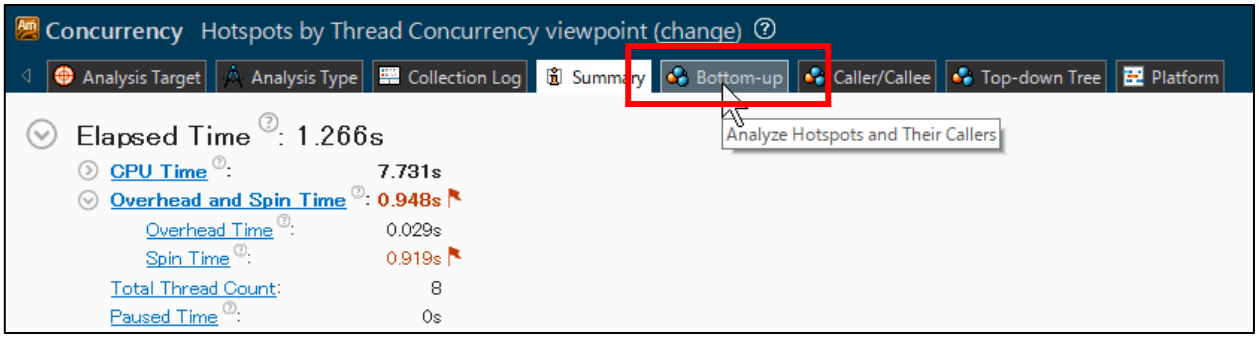
2つの Histogram が表示されており、サンプルプログラムがマルチスレッドで動作していることが確認できます。両方のヒストグラムで若干のバラつきが見られますが、8 本のスレッドが 8 コア上で実行できている時間が一番多いことが確認できます。

Thread Concurrency Histogram：
横軸はスレッド数、縦軸は経過時間を示しており、アプリケーションの実行において、同時実行されたスレッドの本数とその経過時間の分布を表しています。また同時実行スレッド数によって色分けされ、システムが搭載するコア数近辺は“Ideal”つまり理想的な実行と見なされ、逆にスレッド数が少ないエリアは“Poor”つまり不出来な状態を意味します。
CPU Usage Histogram：

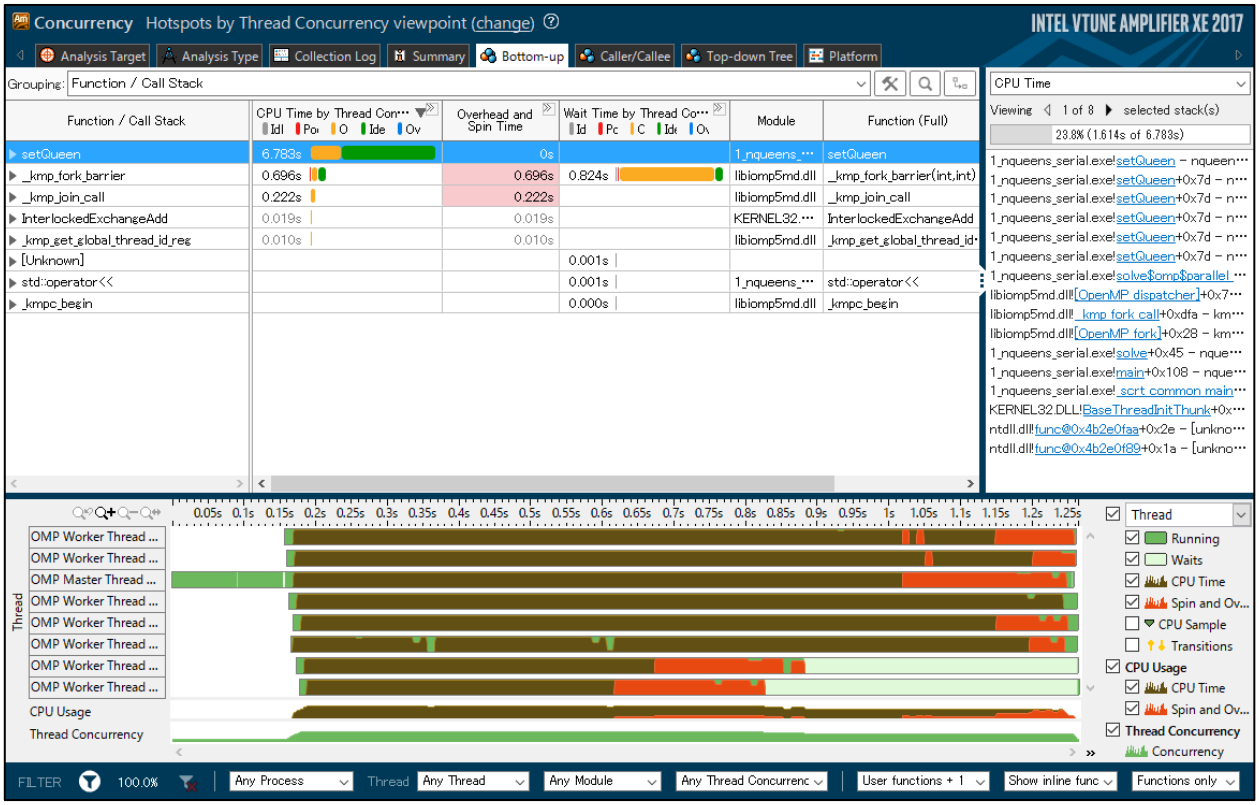
横軸は CPU コア数、縦軸は経過時間を示しており、同時に使用された CPU コアの数とその経過時間の分布を表しています。こちらも性能別に色分けされ、最大コア数近辺での実行は“Ideal”であり低いコア数のエリアは“Poor”となります。

(4) Bottom-up に移動します

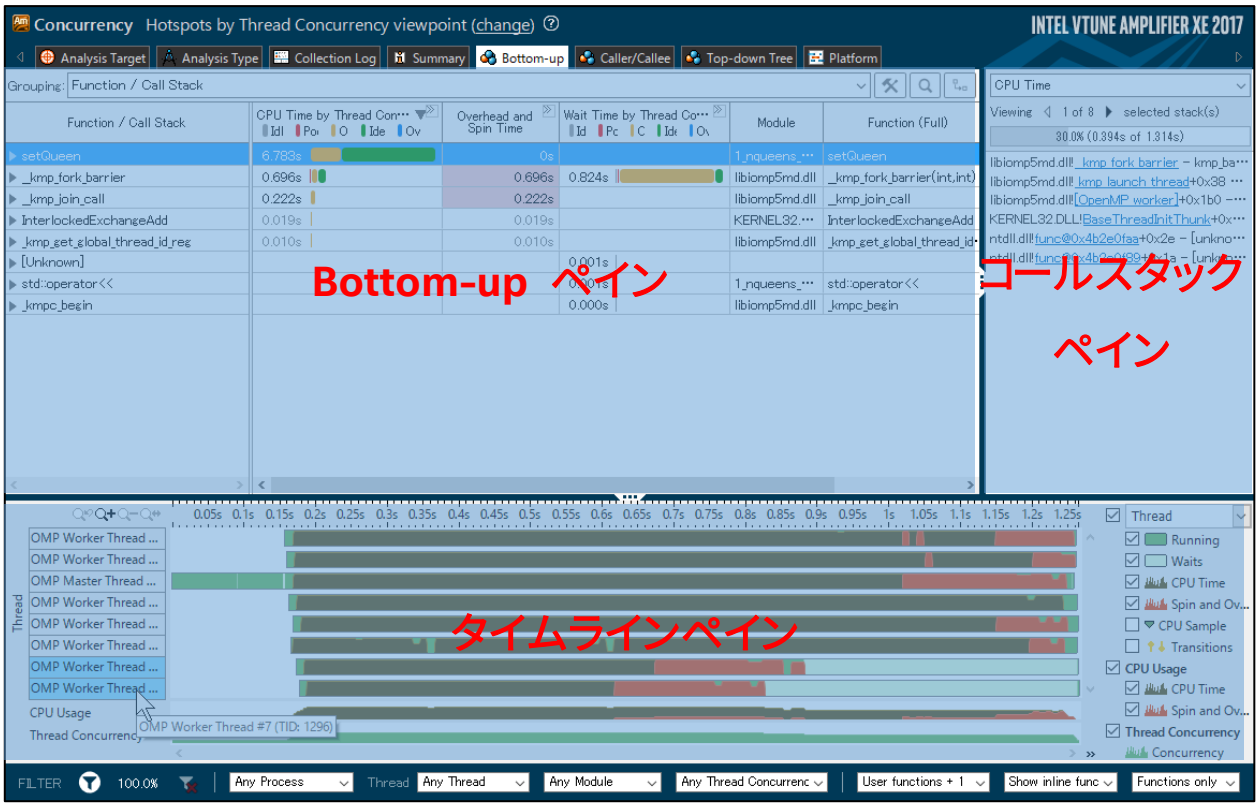
より具体的な情報を確認するために解析画面を移動します。



Bottom-up タブをクリックします。



(5) Bottom-up を確認します



Bottom-up・ペイン

Hostspot 関数や消費 CPU 時間などを表示

コールスタック・ペイン

関数のコールスタック(呼び出し関係)を表示

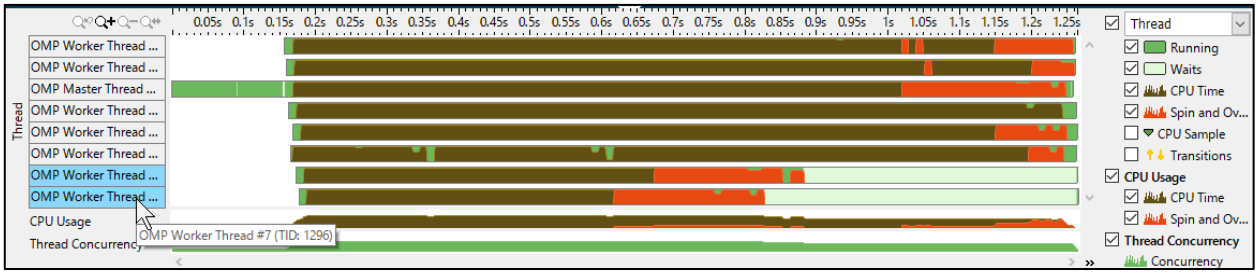
タイムライン・ペイン

スレッド単位の実行状態を時系列に表示

Bottom-up ペインには setQueen が一番計算に時間がかかっている関数として表示されています。setQueen に続いて openmp ライブラリーで実装されているスレッド関連の関数(_kmp_...)が確認できます。CPU Time by Thread Concurrency から setQueen が CPU を 6.783(秒)使用しており、サンプルプログラム内のほとんどの時間を消費していることが分かります。また CPU 時間の多くは緑色のバーで示されていることからこの関数の同時実行状況は“Ideal”であることも分かります。

(6) 計算時間が少ないスレッドを確認します

タイムライン・ペインからスレッド毎の動作状況を確認します。

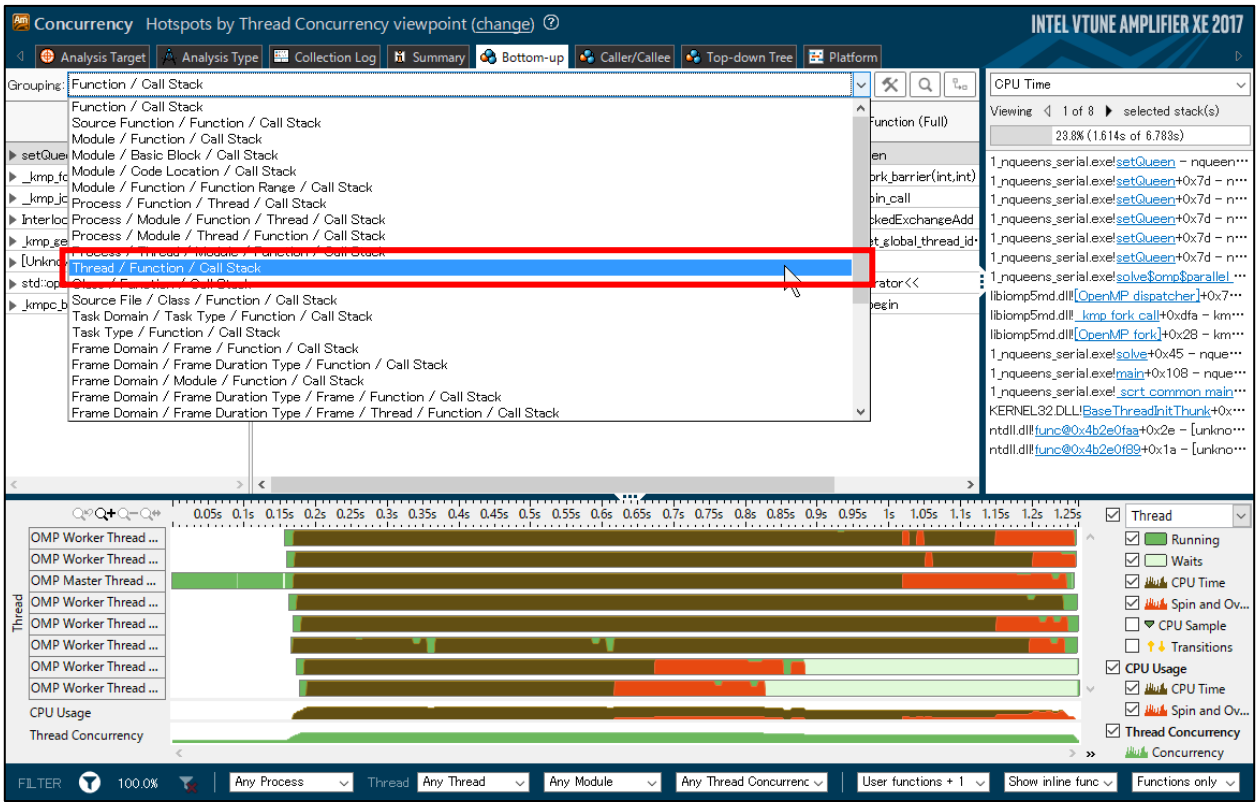


Timeline ペインを見ると、茶色のバーと赤色のバーが表示されています。これはそれぞれ、CPU time (茶)と Spin and Overhead (赤)を示しており、スレッドに対する CPU の実行状態を示しています。CPU Time では そのスレッドに対してアプリケーションの計算が処理されているのに対して、Spin and Overhead ではサンプルプログラムの実行に付随して発生する余分な処理になり、演算に直接関係ない処理を行っています。

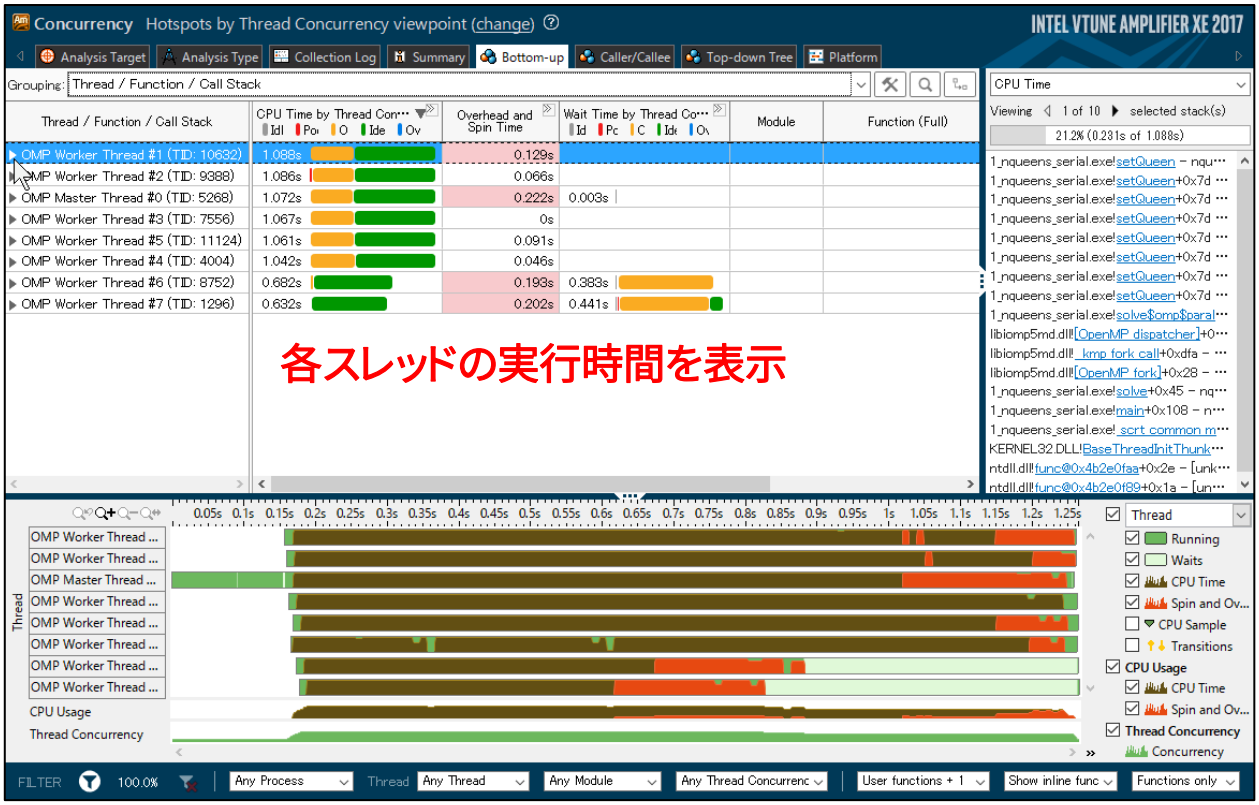
タイムライン・ペインでは、2 本のスレッドの動作が他のスレッドと大きく異なっていることが確認できます。

(7) Grouping を Thread/Function/Call Stack に変更します

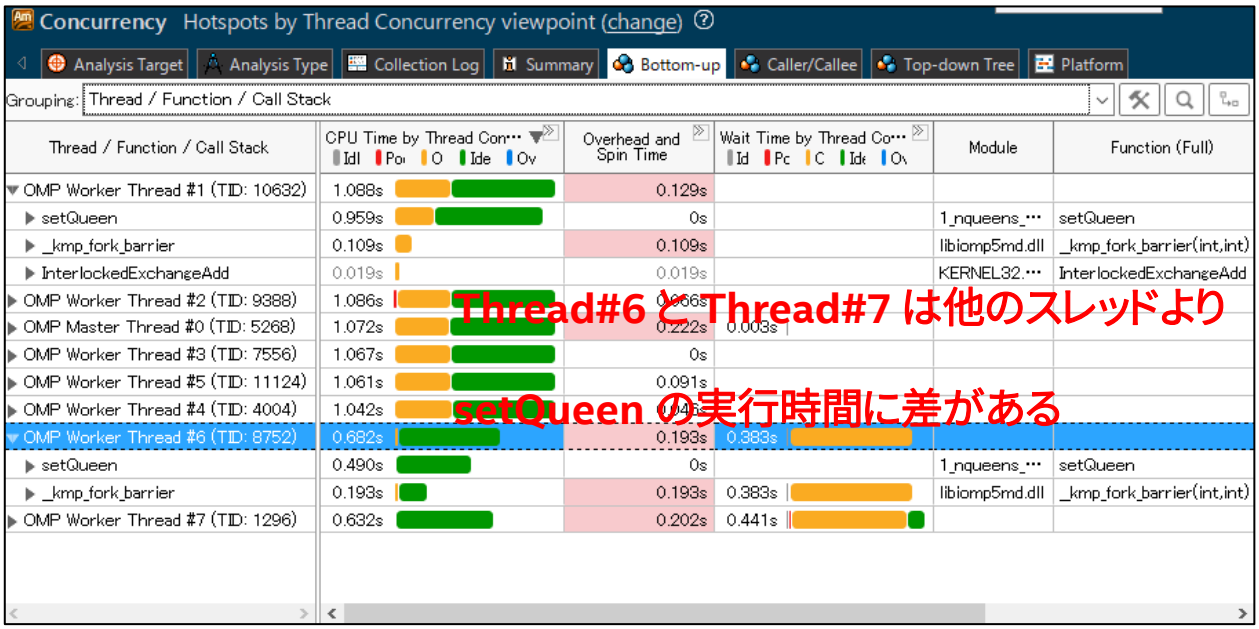
各スレッド内で実行された関数や、実行時間を確認します。



Grouping を変更することで、Bottom-up ペインに表示される情報をソートすることができます。



(8) 計算時間が少ないスレッドの処理内容を確認します



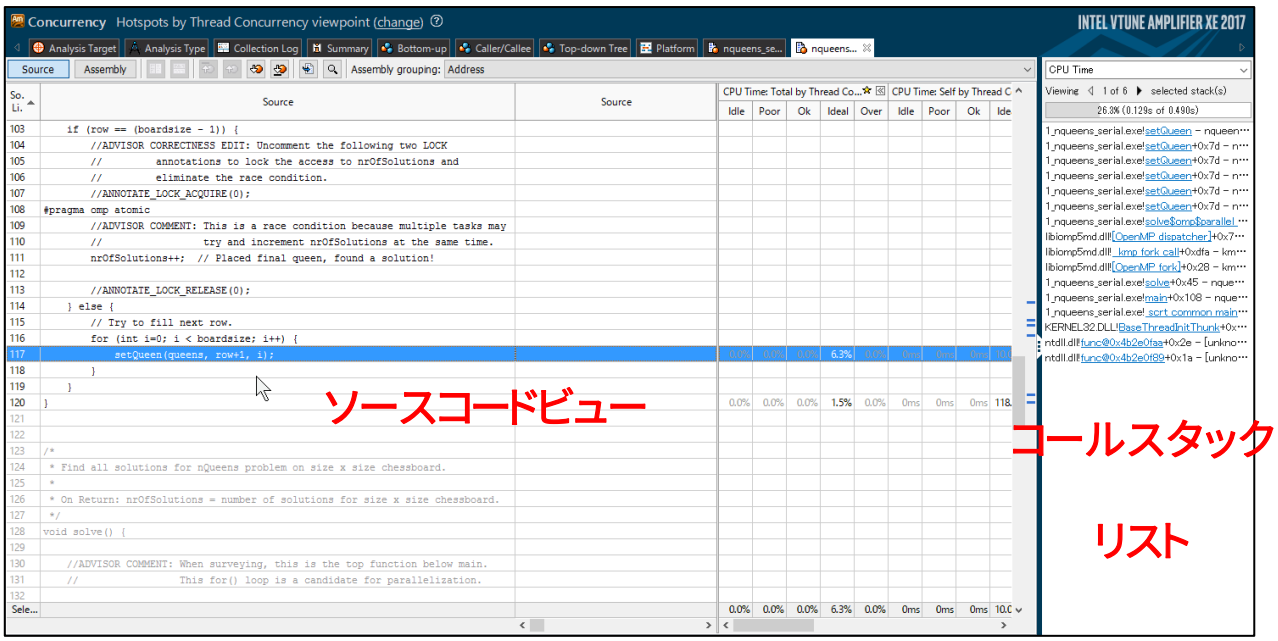
画面から OMP Worker Threads #6 と OMP Worker Thread #7 は他のスレッドと比べて setQueen の実行時間に約2倍の差があることが確認できます。

STEP15 スレッド毎の実行時間に差がある原因を確認する

一部のスレッドの実行時間が他と大きく異なる場合、マルチスレッド化した処理に注目します。マルチスレッドで処理される仕事量にばらつきがあるかどうか確認します。

(1) setQueen のソースコードを確認します

Bottom-up から Thread#6 が実行した setQueen をダブルクリックします。



(2) ハイライトされている処理を確認します

So. Li.	Source
103	`    if (row == (boardsize - 1)) {`
104	`        //ADVISOR CORRECTNESS EDIT: Uncomment the following two LOCK`
105	`        //         annotations to lock the access to nrOfSolutions and`
106	`        //         eliminate the race condition.`
107	`        //ANNOTATE_LOCK_ACQUIRE(0);`
108	`#pragma omp atomic`
109	`        //ADVISOR COMMENT: This is a race condition because multiple tasks may`
110	`        //                 try and increment nrOfSolutions at the same time.`
111	`        nrOfSolutions++;  // Placed final queen, found a solution!`
112	
113	`        //ANNOTATE_LOCK_RELEASE(0);`
114	`    } else {`
115	`        // Try to fill next row.`
116	`        for (int i=0; i < boardsize; i++) {`
117	`            setQueen(queens, row+1, i);`
118	`        }`
119	`    }`
120	`}`
121	
122	
123	`/*`
124	`  * Find all solutions for nQueens problem on size x size chessboard.`
125	`  *`
126	`  * On Return: nrOfSolutions = number of solutions for size x size chessboard.`

ここでは、再帰的に処理しているため、setQueen で呼び出している setQueen が表示されていますが、ループ条件に設定されている boardsize に注目します。

マルチスレッド化により並列に実行できる処理はスレッド数で均等に割られて実行されます。ここでは総スレッド数 8 に対して、ループ回数が 14 (boardsize) に設定されているため、インテル® VTune™ Amplifier XE で確認したスレッド OMP Worker Threads #6 と OMP Worker Thread #7 が1 boardsize 分の計算しか行っていないと推測できます。

ただし、この実行時間の差を解決するためには、さらにループ回数を多くとるように計算アルゴリズムを大幅に変更する必要があり、高速化するための現実的な方法ではありません。この場合、Concurrency レベルの解析ではなく、より詳細な解析によってチューニング可能な箇所を見つけます。

4-5 より実行時間を短縮するための最適化

マルチスレッドによる並列化を実装し複数のコアを使用して同時に計算を行うことで、CPU の使用率が上がり実行時間を短縮しました。ここでは、さらに実行時間を短縮するためにサンプルプログラムに最適化が可能かどうか確認します。

STEP16 インテル® VTune™ Amplifier XE の EBS 機能を使用する

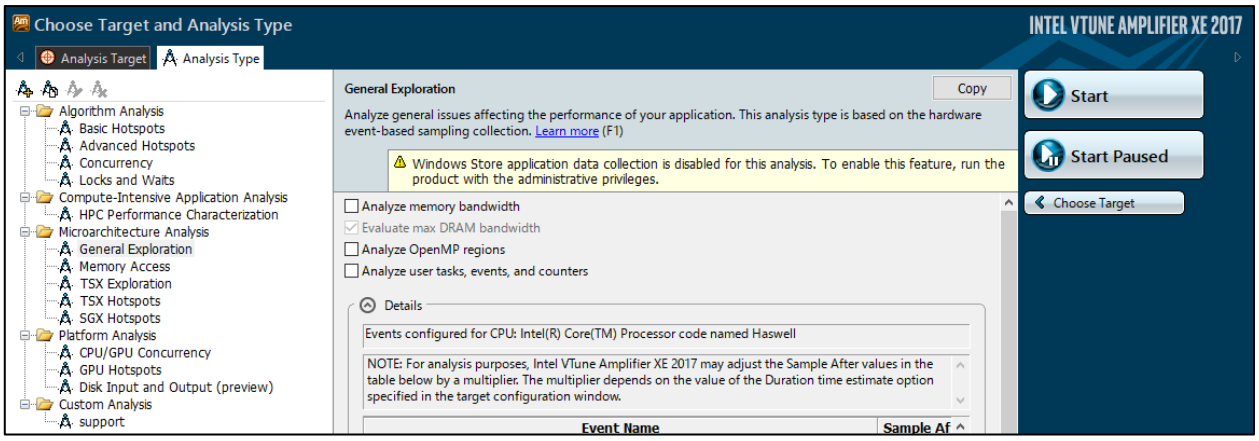
インテル® VTune™ Amplifier XE のイベント・ベース・サンプリング（EBS）機能を使用して、CPU 内部で発生するさまざまな処理（イベント）情報を収集することができます。

EBS で取得可能な情報について

イベントは CPU アーキテクチャーによって使用できるイベントの種類、イベント数、イベント名が異なり、例えば、下記のイベントが収集できます。

- CPU クロック数
- リタイア命令数
- 分岐予測ミス
- L1 / L2 / LLC キャッシュミス
- キャッシュ スヌープ処理
- ITLB / DTLB ミス
- 各種命令（FP / MMX / SIMD / LOAD / STORE など）のカウント
- 実行ポート単位のμOP（マイクロオペレーション）数
- パイプライン・ストール
- オフコアイベント

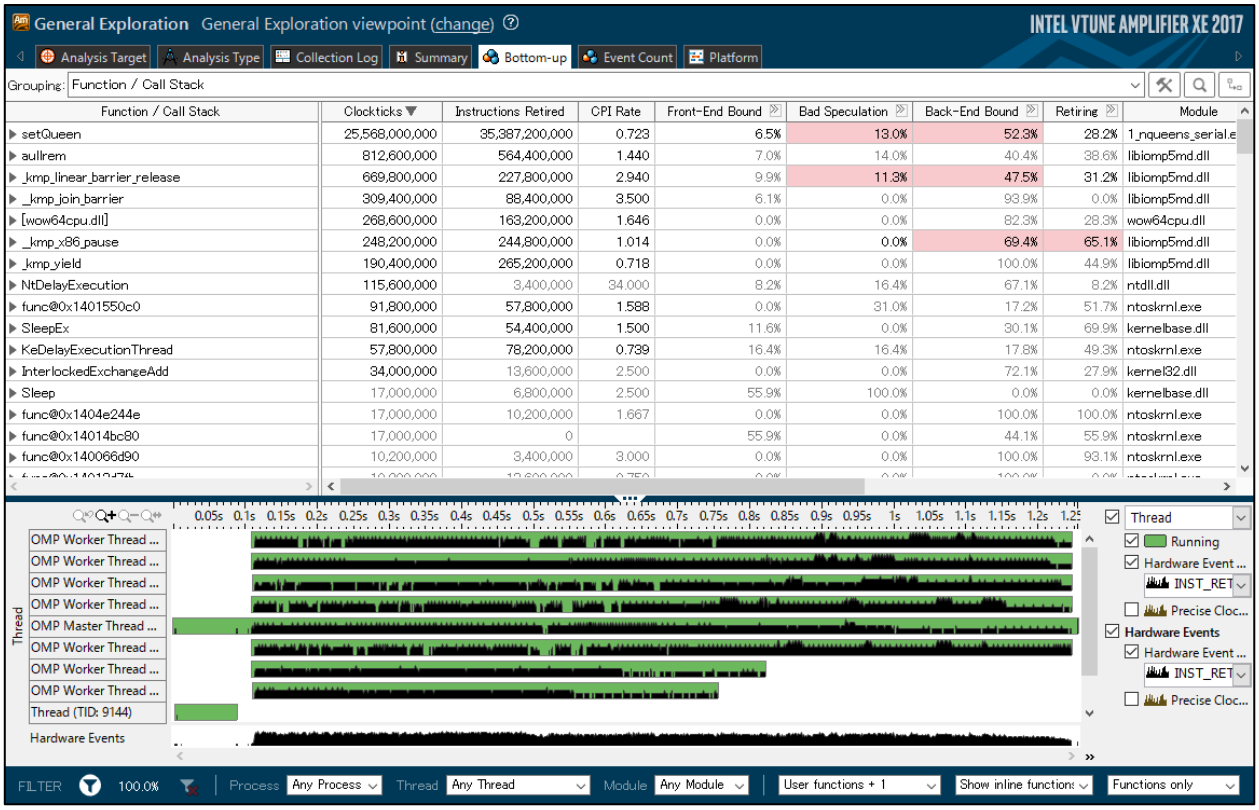
(1) General Exploration 解析を実行します



STEP17 General Exploration 解析の結果から最適化できそうな箇所を判断する

(1) Bottom-up からサンプルプログラムの実行状況を調査します

解析終了後、Summary から Bottom-up に移動します



Bottom-up 画面を確認すると、setQueen 関数にて、Bad Speculation と Back-End Bound のセルがピンク色でマークされています。これはサンプルプログラムの動作に対して、対象の値の性能が悪化している可能性があることを示しています。

Bad Speculation

CPU 内部のパイプライン上で効率よく命令(uOps: micro Operation)が実行できなかったことを示しています。

Back-End Bound

実行に必要な命令が CPU に供給されていないことを示しています。

解析結果では Back-End Bound が 52.3%の割合で性能に影響していることを示しており、キャッシュのヒット率が悪く、CPU 側に命令処理能力に対して命令の供給が追い付いていないため、不要な待機時間が発生している可能性があると考えられます。

(2)　キャッシュミスが原因かどうか確認します

Bottom-up に表示された Back-End-Bound の項目を分け、キャッシュミスの発生を確認します。

クリックします

Grouping: Function / Call Stack

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring	Module
setQueen	25,568,000,000	35,387,200,000	0.723	6.5%	13.0%	52.3%	28.2%	1_nqueens_serial.exe
aullrem	812,600,000	564,400,000	1.440	7.0%	14.0%	40.4%	38.6%	libiomp5md.dll
_kmp_linear_barrier_release	669,800,000	227,800,000	2.940	9.9%	11.3%	47.5%	31.2%	libiomp5md.dll
_kmp_join_barrier	309,400,000	88,400,000	3.500	6.1%	0.0%	93.9%	0.0%	libiomp5md.dll
[wow64cpu.dll]	268,600,000	163,200,000	1.646	0.0%	0.0%	82.3%	28.3%	wow64cpu.dll
_kmp_x86_pause	248,200,000	244,800,000	1.014	0.0%	0.0%	69.4%	65.1%	libiomp5md.dll
_kmp_yield	190,400,000	265,200,000	0.718	0.0%	0.0%	100.0%	44.9%	libiomp5md.dll

クリックします

Grouping: Function / Call Stack

Function / Call Stack	ion	Back-End Bound						Retir
		Memory Bound	Core Bound					
			Divider	Port Utilization				
				Cycles of 0 Ports Utilized	Cycles of 1 Port Utilized	Cycles of 2 Ports Utilized	Cycles of 3+ Ports Utilized	
setQueen	13.0%	21.6%	0.0%	22.3%	7.6%	12.0%	35.1%	
aullrem	14.0%	3.0%	0.0%	100.0%	88.9%	84.2%	100.0%	
_kmp_linear_barrier_release	11.3%	0.0%	0.0%	0.0%	100.0%	22.7%	100.0%	
_kmp_join_barrier	0.0%	57.6%	0.0%	100.0%	0.0%	0.0%	0.0%	
[wow64cpu.dll]	0.0%	5.9%	0.0%	99.0%	99.0%	99.0%	99.0%	

Memory Bound の項目を開きます。

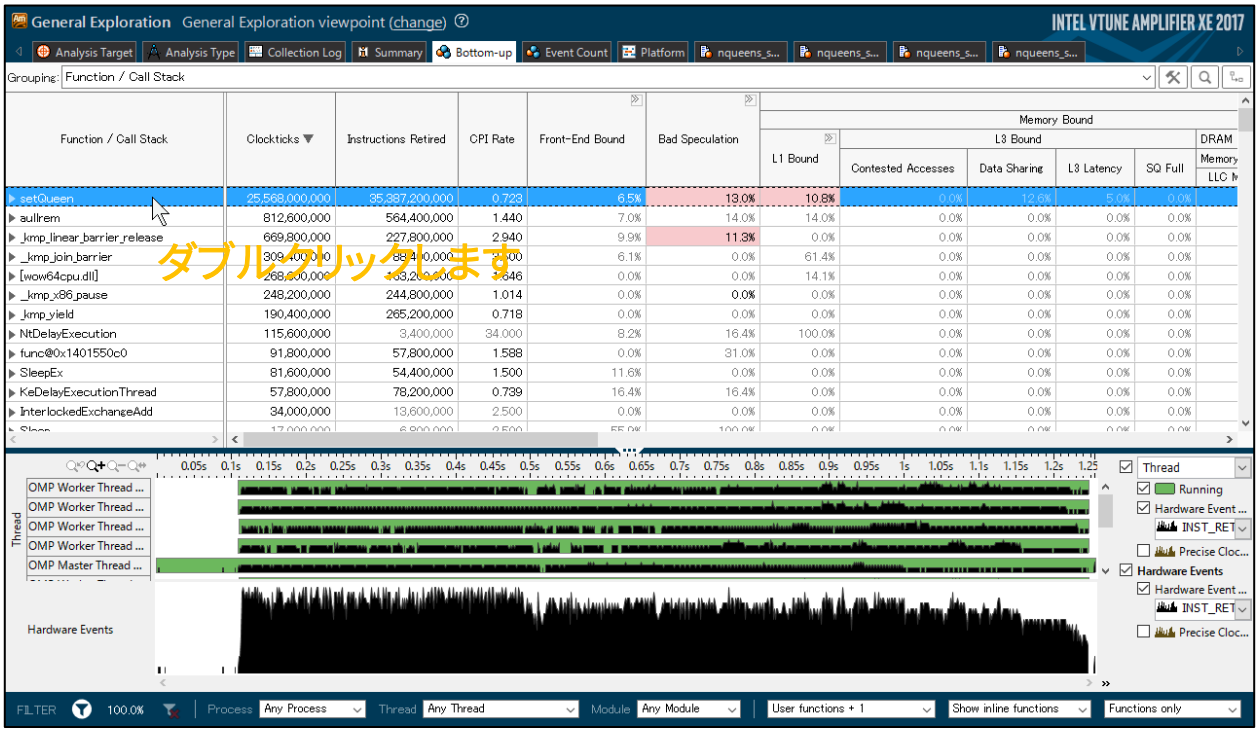
Grouping: Function / Call Stack

Function / Call Stack	Front-End Bound	Bad Speculation	Back								
			Memory Bound								
			L1 Bound	L3 Bound				DRAM Bo…	Store Bound		
				Contested Accesses	Data Sharing	L3 Latency	SQ Full	Memory L… LLC Miss			
setQueen	6.5%	13.0%	10.8%	0.0%	12.6%	5.0%	0.0%	0.0%	0.0%		
aullrem	7.0%	14.0%	14.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		
_kmp_linear_barrier_release	9.9%	11.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		
_kmp_join_barrier	6.1%	0.0%	61.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		

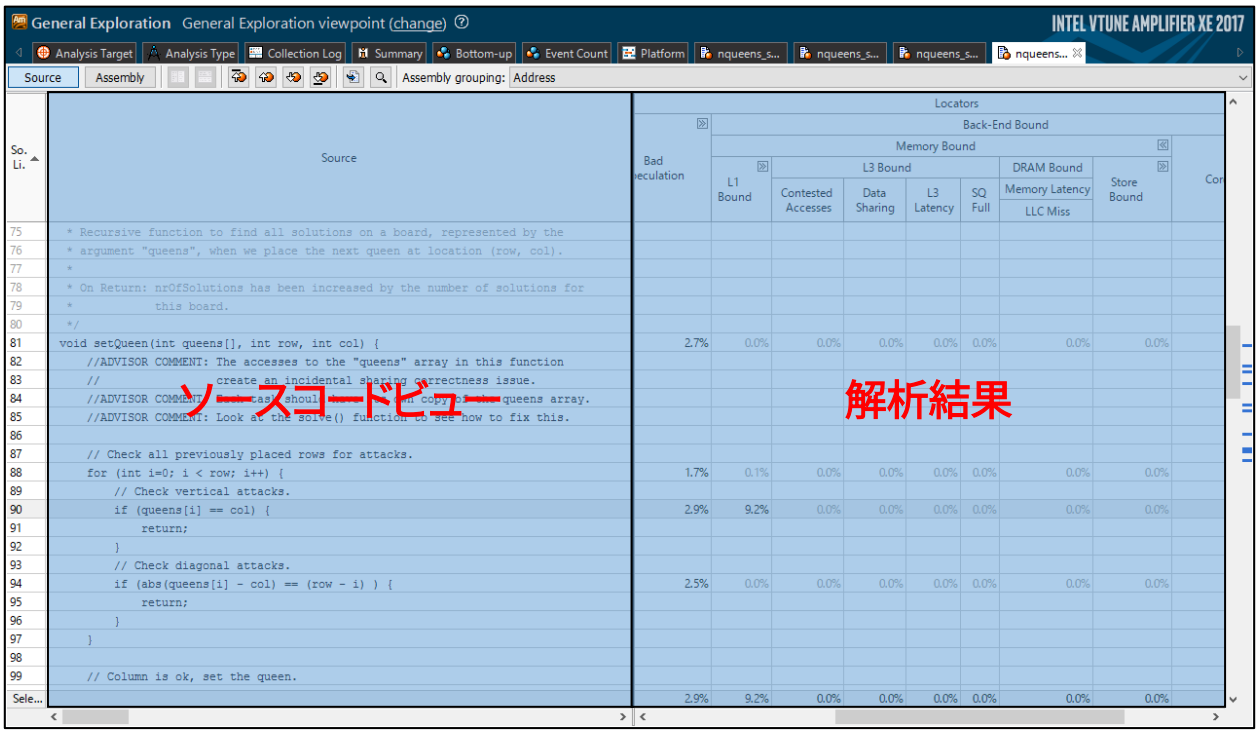
L1 Bound では L1 レベルのキャッシュミスが発生していることを示しており、このセルがピンク色になっていることが確認できます。サンプルプログラム内でキャッシュミスを発生させる原因を特定して改善することで、さらに実行時間を短縮できる可能性があります。

STEP18 キャッシュミスの原因を特定して改善する

(1) ソースコード内でキャッシュミスが発生している処理を特定する



setQueen をダブルクリックして対してソースコードレベルで結果を確認します

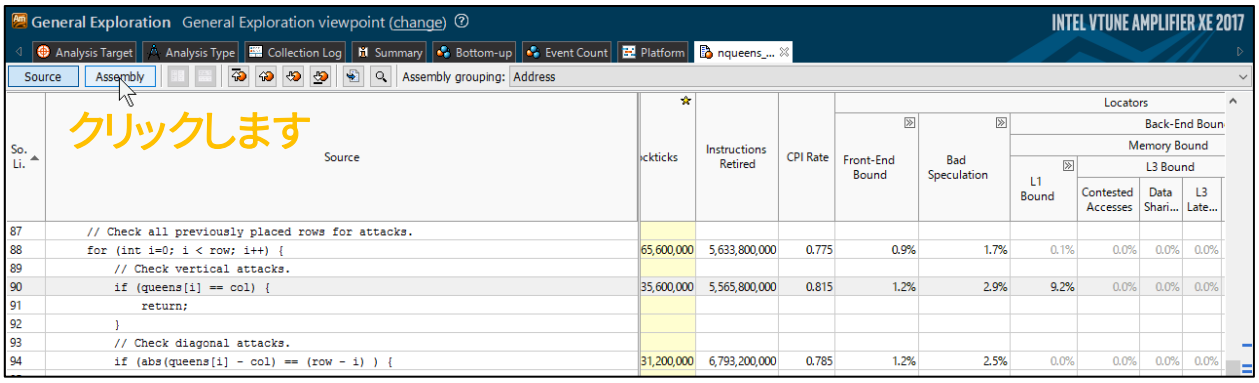


setQueen のソースコードとインテル® VTune™ Amplifier XE の解析結果が紐づけされソースコードレベルの解析結果が表示されます。

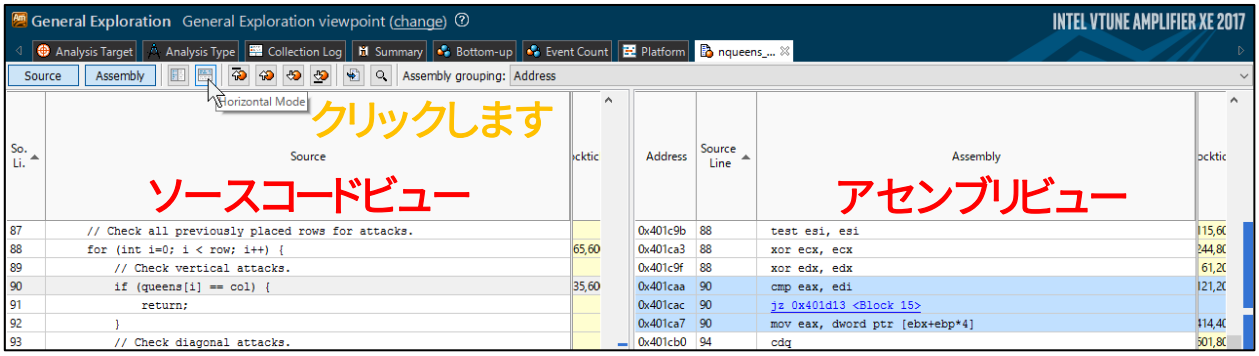
87	// Check all previously placed rows for attacks.									
88	for (int i=0; i < row; i++) {	1.7%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	
89	// Check vertical attacks.									
90	if (queens[i] == col) {	2.9%	9.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	
91	return;									
92	}									
93	// Check diagonal attacks.									
94	if (abs(queens[i] - col) == (row - i)) {	2.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	
95	return;									
96	}									
97	}									

ソースコードレベルで見ると L1 Bound を発生している処理が 90 行目にあることを確認できます。

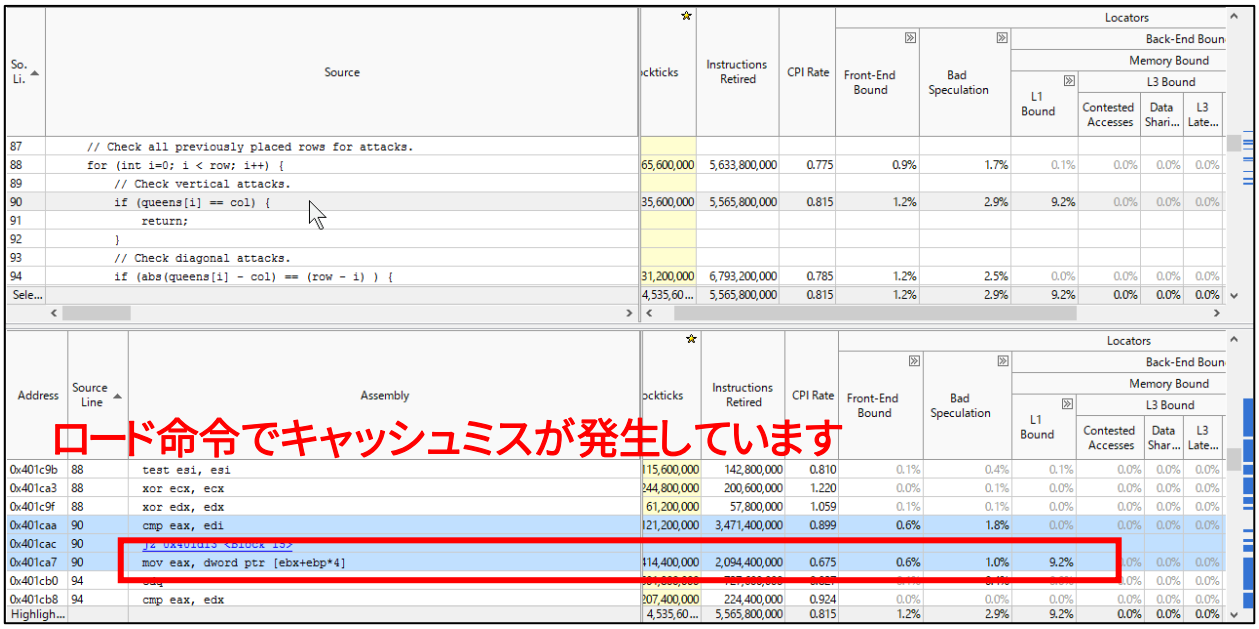
(2) アセンブラからキャッシュミスの原因を特定する



Assembly をクリックして 90 行目に対応したアセンブラを表示します。



可読性を上げるためにコードビューとアセンブリビューを水平に表示します。



各イベントの値を確認すると、queens の値をレジスタにロードする命令（mov）で L1 Bound が発生しています。このロード命令は並列実行領域内で実行されているため、フォルス・シェアリングが発生している可能性があります。

> フォルス・シェアリング：
> 同じキャッシュラインに対して、複数のコアがアクセスを行うことでキャッシュミスが発生する問題

(3) フォルス・シェアリングを解決する

フォルス・シェアリングはアクセス対象の配列がキャッシュラインをまたがないようにアライメントすることで解決します。

```
139         ANNOTATE_SITE_BEGIN(solve);
140     #pragma omp parallel for
141         for (int i=0; i < boardsize; i++) {
142             ANNOTATE_ITERATION_TASK(setQueen);
143
144             //ADVISOR CORRECTNESS EDIT: Uncomment the declaration of queens.  This
145             //                          creates a separate array for each recursion
146             //                          eliminating the incidental sharing.
147             //int * queens = new int[boardsize]; // Array of queens on the chess board.
148             int * queens = (int *)_mm_malloc(sizeof(int)*boardsize, 64);
149
150             //ADVISOR COMMENT: The call below exhibits incidental sharing when all
151             //                 of the tasks use the same copy of "queens".
152             // Try all positions in first row.
153             setQueen(queens, 0, i);
154
155             //ADVISOR CORRECTNESS EDIT: Uncomment the deletion of the queens array.
156             //delete [] queens;
157             _mm_free(queens);
158         }
159         ANNOTATE_SITE_END();
160
161         //ADVISOR CORRECTNESS EDIT: Comment out the deletion of the queens array.
162        // delete [] queens;
163     }
```

queens は solve() の OpenMP の並列領域内で定義され、for ループの回数分のメモリ領域を取得します。メモリ領域を64バイト境界に余裕を持って確保することで、スレッド間のキャッシュラインの共有を防ぐことができます。

ここでは、アライメント付きの動的メモリ取得関数（_mm_malloc）と解放関数（_mm_free）を使用します。

(4) サンプルプログラムを実行します

C:¥WINDOWS¥system32¥cmd.exe

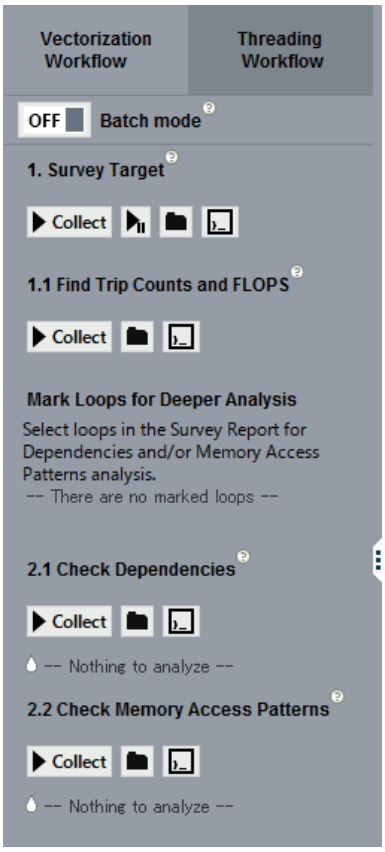
```
Usage: C:¥work¥nqueens_Advisor¥Release¥1_nqueens_serial.exe boardSize [default is 14].
Starting nqueens (C:¥work¥nqueens_Advisor¥Release¥1_nqueens_serial.exe) solver for size 14...
Number of solutions: 365596
Correct result!

Calculations took 968ms.
続行するには何かキーを押してください . . .
```

queens の確保処理を変更することで、さらに実行時間を短縮することができました。

5. 解析ツールに関する Tips

5-1 インテル® Advisor

◇ 効果的なベクトル化を実装する



手順 1：Survey Target (ターゲットアプリの調査)
まず、アプリケーションを実行して実行時間が多いループ処理を表示し、その中から問題を含むループ処理に対してアノテーションを設定します。

手順 1.1：Find Trip Counts (ループ回数の調査)
ループ処理の反復回数を追加で検出し、レポートに表示します。

手順 2.1：Check Dependencies（依存性の調査）
アノテーションが設定されたループに対して、データ競合などのベクトル化の実装を妨げている問題を確認します。

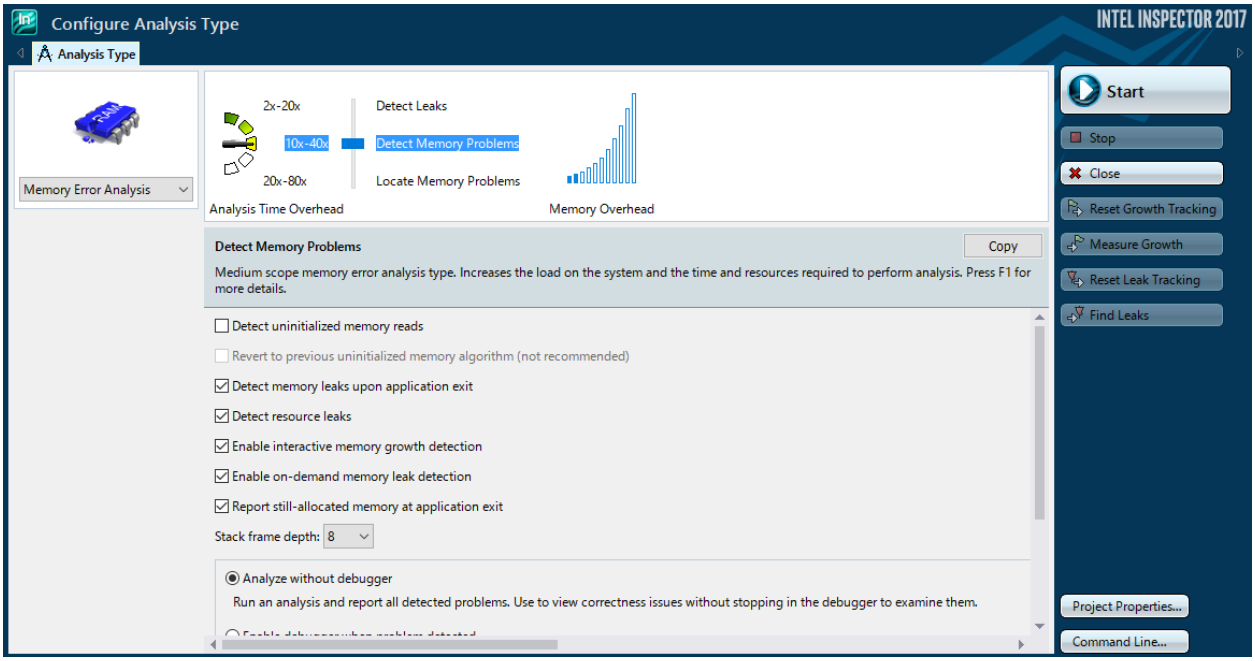
手順 2.2：Check Memory Access Patterns（メモリーアクセスの調査）
アノテーションが設定されたループ処理に対して、メモリーアクセスに関わる依存性をチェックし、メモリージャンプの可能性があるかどうか確認します。

ベクトル化アドバイザーを使用することで、対象のプログラムに実装されているベクトル状況を把握することができます。また、ベクトル化できない処理に対して、ベクトル化できない原因とその回避策を提示することで、既存のプログラムに対して効果的なベクトル化が可能かどうか判断できます。

> ベクトル化アドバイザーを利用する際は、インテル® コンパイラーの最適化レポートオプションである /Qopt-report:5 を追加してプログラムをビルドしておくことを推奨しています。また日本語環境におけるレポート関連の問題を回避するには /Qdiag-message-catalog- オプションを追加してください。

5-2 インテル® Inspector

◇ メモリエラー解析



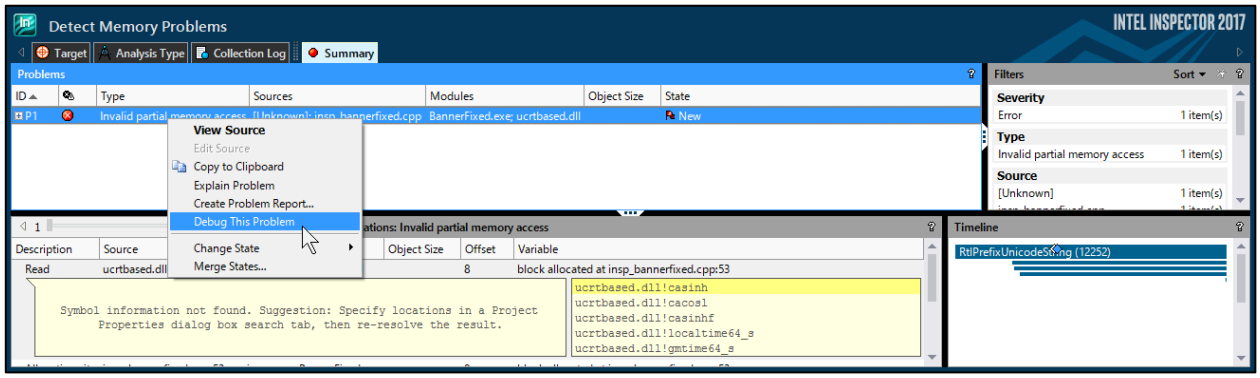
インテル® Inspector には、スレッドエラーのほかに、メモリエラーをチェックする機能があります。この機能は、シングルスレッドおよびマルチスレッド・プログラムにおいて潜在するメモリエラーを検出します。“Memory Error Analysis”を選択し、検出レベルを設定し、Start ボタンをクリックしてメモリーチェックを開始します。

◇ 外部デバッガーと連携して解析結果のエラーをデバッグする

3 つの方法から、デバッガーと連携してプログラムを解析することが可能です。

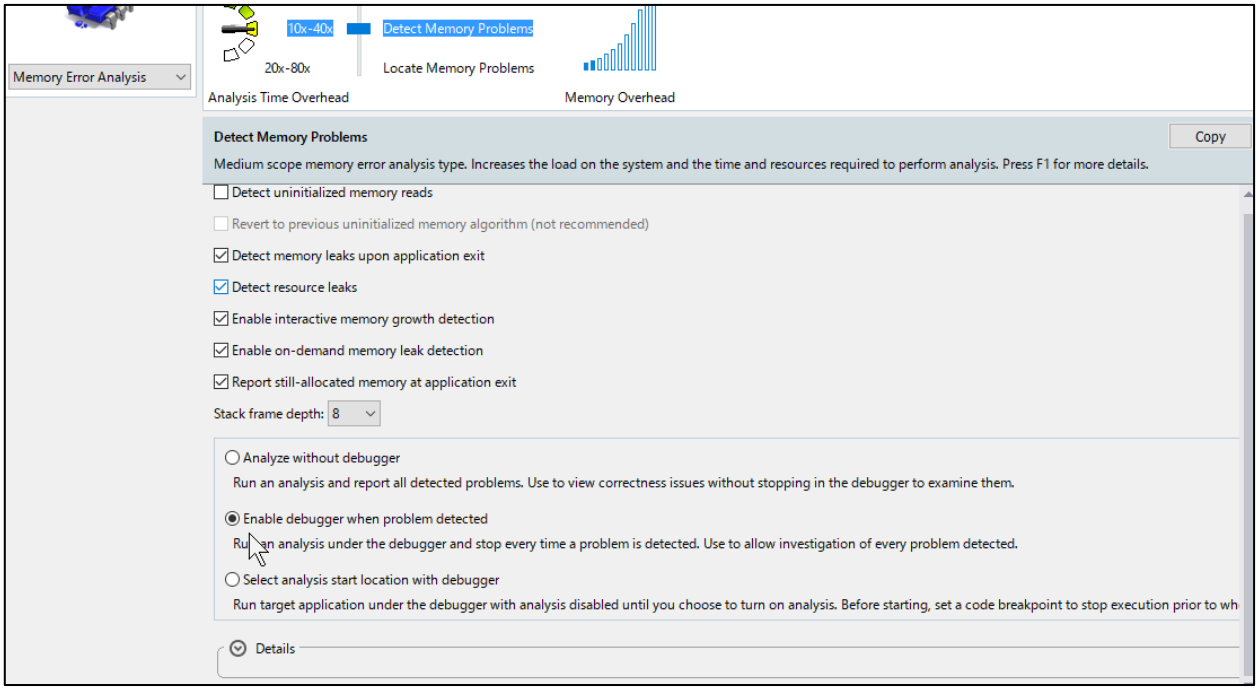
■ 表示されたエラーについてデバッグする

解析結果から表示されたエラーを右クリックしてメニューから Debug This Problem を選択します。



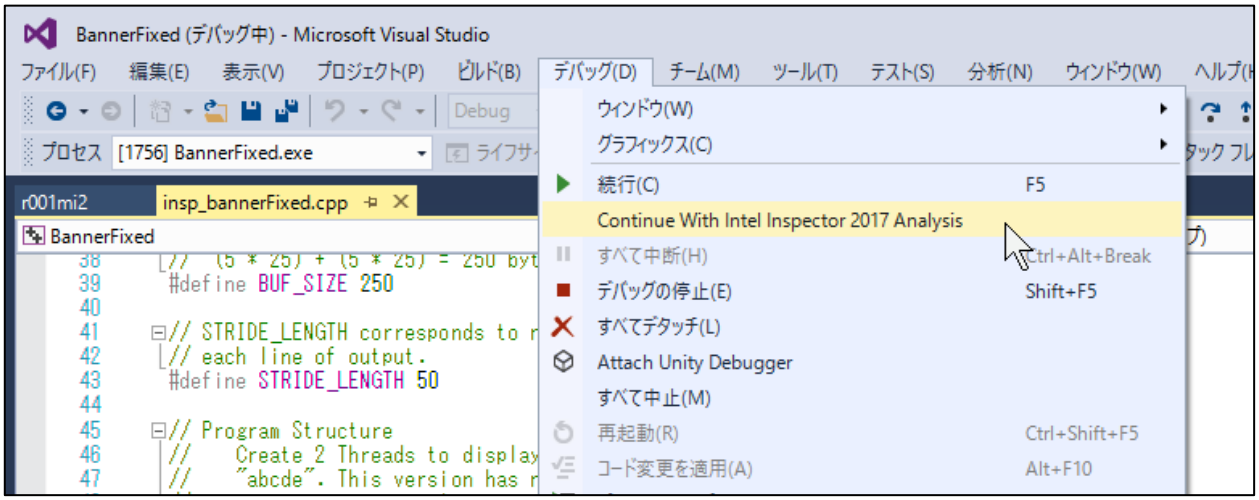
■ 任意のエラーをデバッグする

インテル® Inspector の解析前の設定を Enable debugger when problem detected に変更して解析を開始します。



■ ブレークポイントからエラーのデバッグを行う

インテル® Inspector の解析前の設定を Select analysis start location with debugger に設定して解析を開始します。プログラムがブレークポイントまで実行された後、[デバッグ]から Continue With Intel Inspector XXXX Analysis を選択します。



5-3 インテル® VTune™ Amplifier XE

◇ TBS と EBS

インテル® VTune™ Amplifier XE では、以下の2種類のサンプリング手法が使用されます。

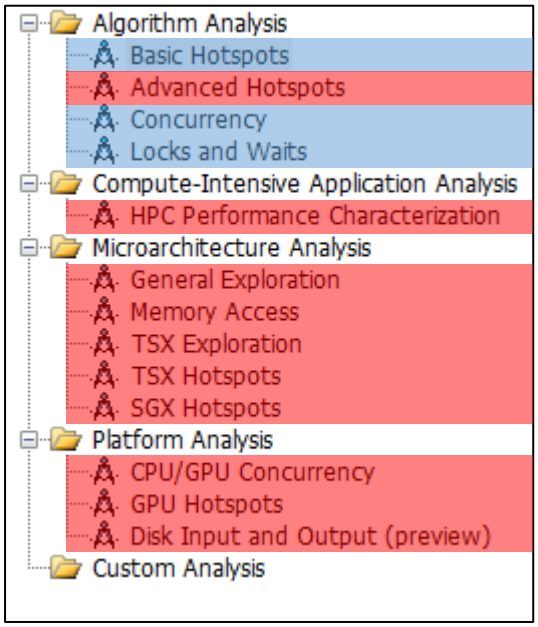
■ Time-Based Sampling (TBS)

プロセッサーに周期的に割り込みを入れて、その割り込み処理の中で各種情報（インストラクション・ポインターやスレッド情報など）を収集します。デフォルトでは 10ms に一回の割合で割り込みを入れてデータを収集します。その際のオーバーヘッドは約 5% 程度です。

■ Event-Based Sampling (EBS)

インテル®プロセッサーに搭載される PMU（パフォーマンス・モニタリング・ユニット）のイベントカウンター・オーバーフローによる割り込みを利用してプロファイル情報を収集します。この割り込みの発生回数がサンプリング回数となります。また割り込みの発生頻度は、PMU のオーバーフロー値を決定する“Sample After Value”を調整することで制御できます。例えば 1ms 単位で割り込みが発生した場合、それによるオーバーヘッドは約 2% 程度となります。

解析プリセット毎に TBS と EBS のどちらを使用するか決まっています。



Time-Based Sampling (TBS)

Event-Based Sampling (EBS)

EBS 解析の条件

いくつかの EBS を使用した解析はプロセッサーに搭載される PMU の機能に制限されます。プロセッサー世代毎にイベント情報が異なる場合がありますので注意してください。TBS の場合はプロセッサーの制限を受けません。

◇ CPI

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ setQueen	25,568,000,000	35,387,200,000	0.723
▶ aullrem	812,600,000	564,400,000	1.440
▶ _kmp_linear_barrier_release	669,800,000	227,800,000	2.940
▶ _kmp_join_barrier	309,400,000	88,400,000	3.500
▶ [wow64cpu.dll]	268,600,000	163,200,000	1.646
▶ _kmp_x86_pause	248,200,000	244,800,000	1.014
▶ _kmp_yield	190,400,000	265,200,000	0.718
▶ NtDelayExecution	115,600,000	3,400,000	34.000

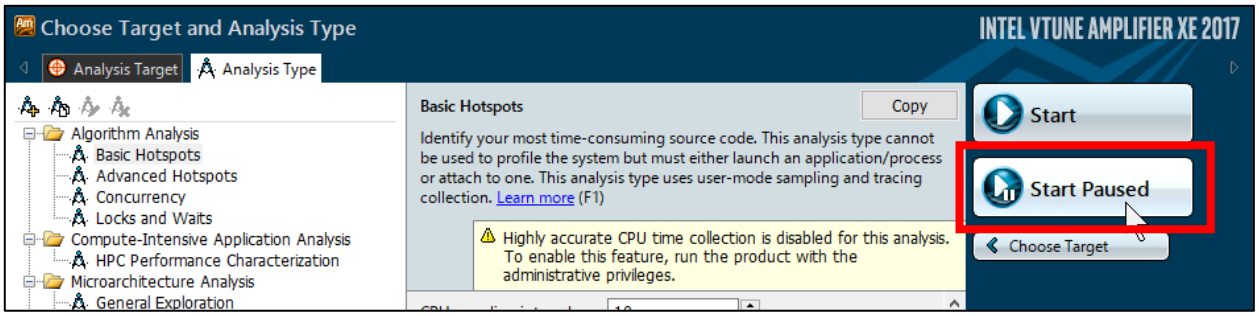
EBS を使用した解析では Clock Ticks Per Instructions retired (CPI) を取得することができます。CPI は実行された1命令を完了するために消費した CPU クロック数を示しており、命令実行の効率性を決定する基本的な指標です。

現在のインテル®プロセッサーでは、1 クロックで最大4つの命令を完了させることができます。つまり CPI の最高理論値は“0.25”(=1/4)となります。これは理論値になるため、一般的なプログラムの CPI 値が 0.25 になることはほとんどありません。

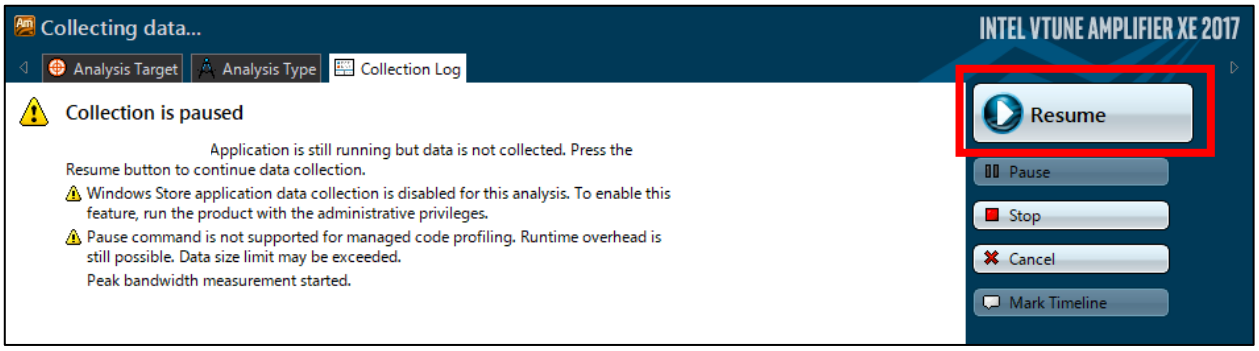
◇ プログラムの実行中から解析を開始する

特定の処理だけを解析したい場合には、ユーザーが手動でインテル® VTune™ Amplifier XE の情報収集を制御する方法と、API をソースコードに記述する方法があります。

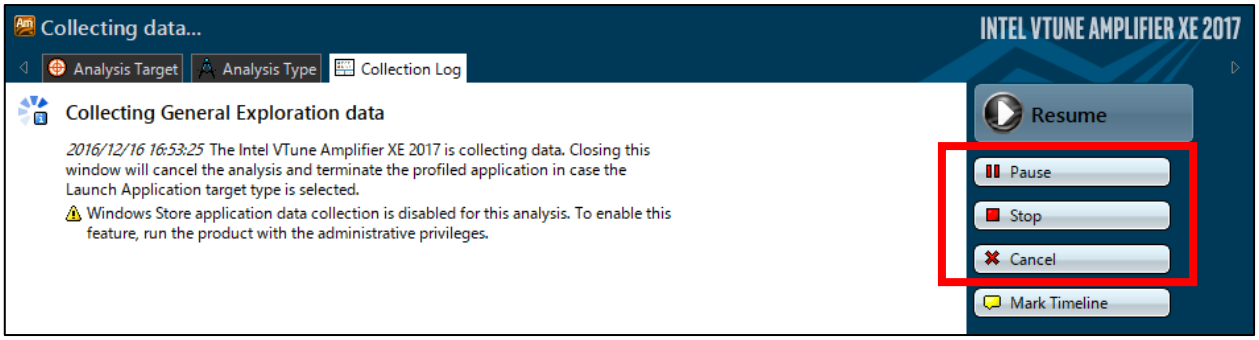
■ GUI から制御する



解析を行う際に Start Paused を選択して開始することで、実行開始時に情報収集を行いません。



プログラムの実行中に GUI から Resume をクリックすることで情報収集を開始します。



Pause や Stop、Cancel をクリックすることで、任意のタイミングで情報収集を中断することが可能です。

■ API を使用する

プロファイルを行いたい特定の処理に Resume 関数（__itt_resume）と Pause 関数（__itt_pause）を追記します。

```
#include "ittnotify.h"
...
Myfunc() {
  ...
  __itt_resume();
  <プロファイルされる処理>
  __itt_pause();
  ...
}
```

また、API を使用するには ittnotify.h をインクルードします。

インクルードパス ： C:\Program Files (x86)\IntelSWTools\VTune Amplifier XE 2017\include

あわせて、libittnotify.lib をリンクします。

ライブラリーパス（32bit）： C:\Program Files (x86)\IntelSWTools\VTune Amplifier XE 2017\lib32

ライブラリーパス（64bit）： C:\Program Files (x86)\IntelSWTools\VTune Amplifier XE 2017\lib64

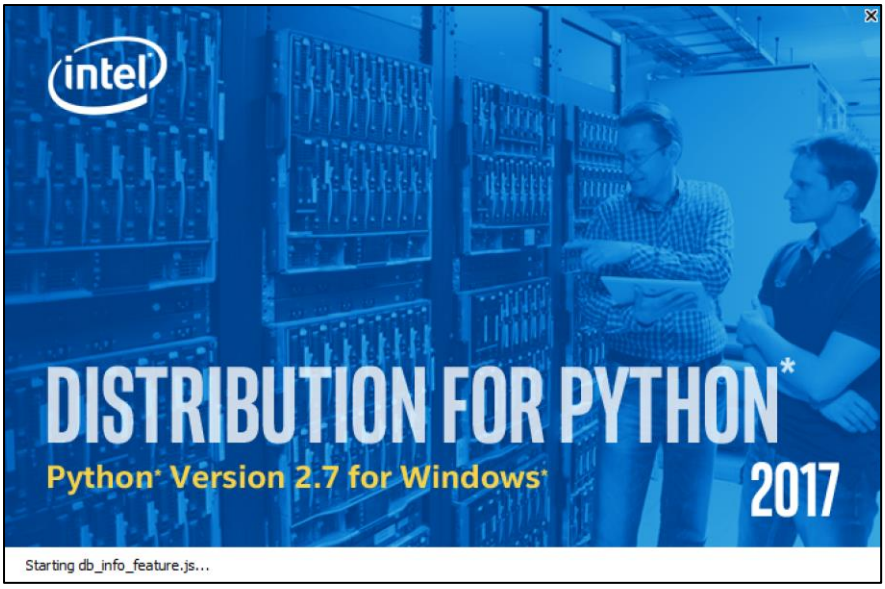
6. 参考情報

6-1 インテル® Distribution for Python*

インテル® Distribution for Python（インテル® Python）とは、既存の Python コードを修正することなくアプリケーションのパフォーマンスを向上させることができる実行環境です。インテル® MKL を始めとして、インテル® DAAL、インテル® MPI などのライブラリーも含まれています。

◇ ダウンロード方法

インテル® Python は、Parallel Studio をインストールするとあわせてインストールされるわけではありません。別途、インテル® Python のインストールが必要です。インテル® Python のインストーラーは、インテルレジストレーションセンターからダウンロードすることができます。（参照 インテル® レジストレーション・センター操作マニュアル）

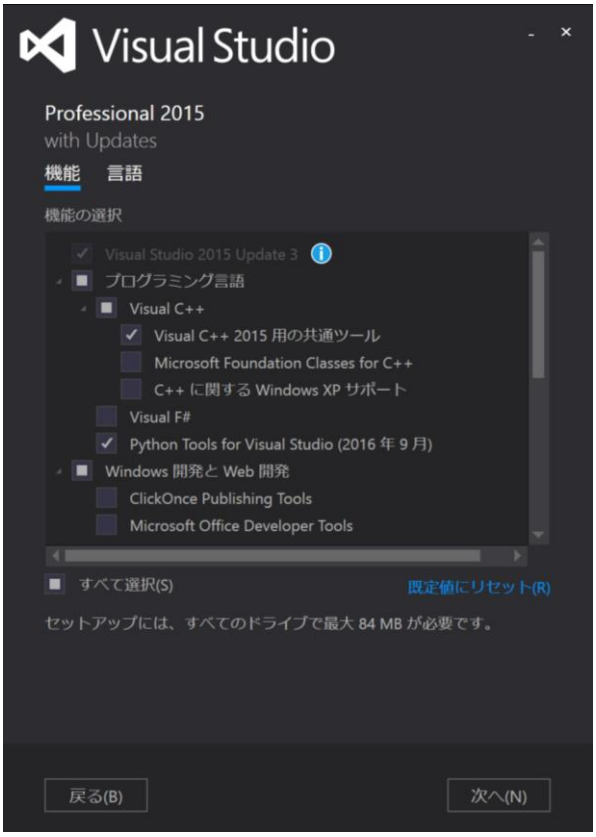


◇ Visual Studio での実行方法

インテル® Python を Visual Studio で使用する方法について説明します。インテル® Python は、Parallel Studio とは異なり自動では統合されないため、インストール後に Visual Studio で使用できるように設定を行う必要があります。

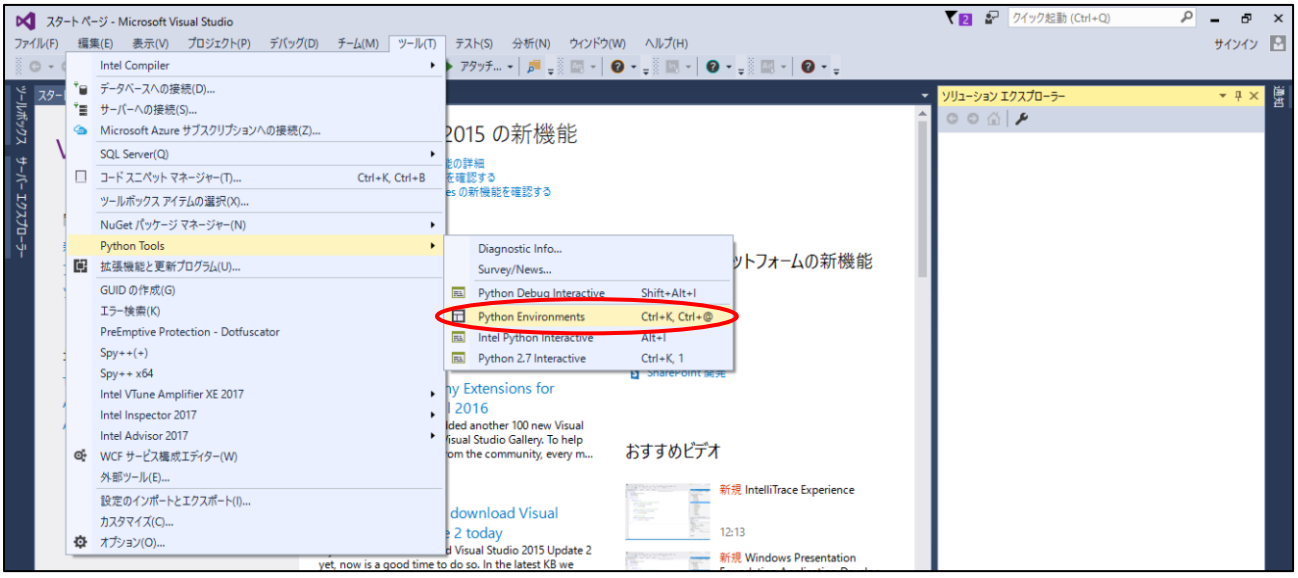
(1) Visual Studio に Python Tools for Visual Studio がインストールされているか確認します

インストールされていない場合は、[Python Tools for Visual Studio] にチェックを入れてインストールを行ってください。



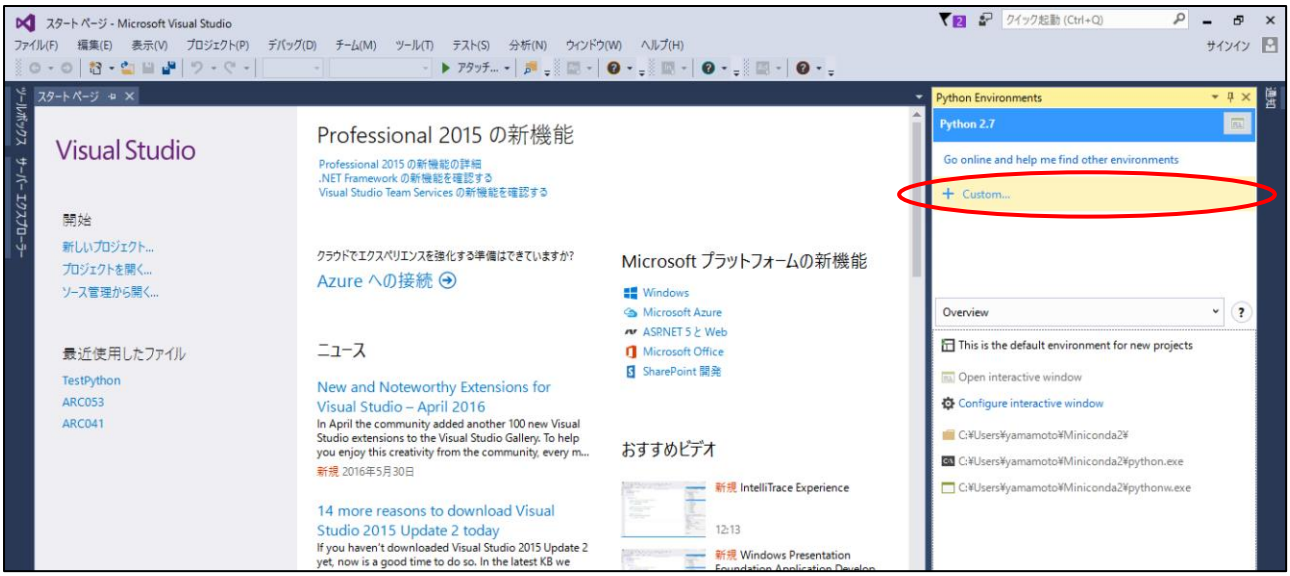
(2) Python の環境設定ページに移動します

[編集] – [Python Tools] – [Python Environments] を選択します。



(3) インテル® Python 環境の登録を行います

[+ Custom...] を選択して、Python 実行環境の新規登録画面に移動します。

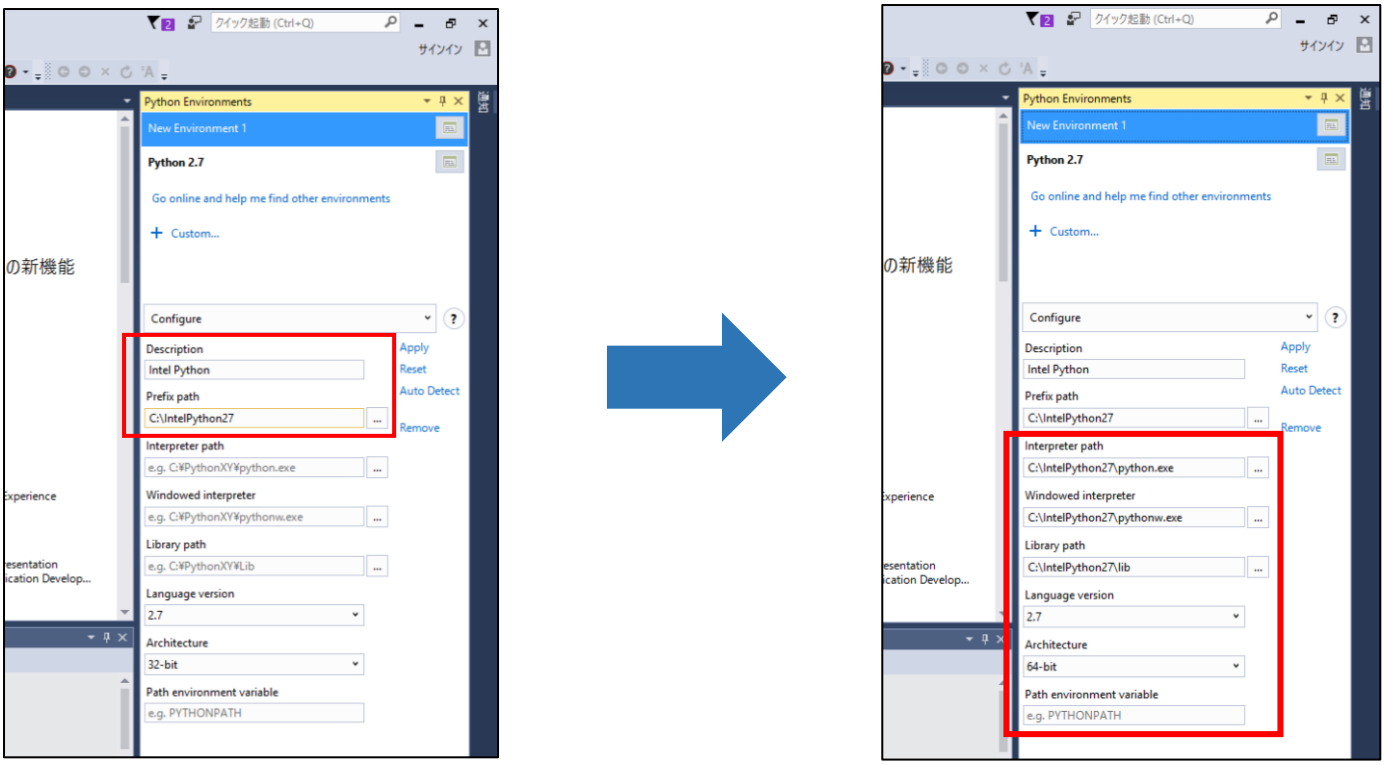


実行環境の登録画面の項目に、以下の情報を書き込みます。

・Description : Intel Python

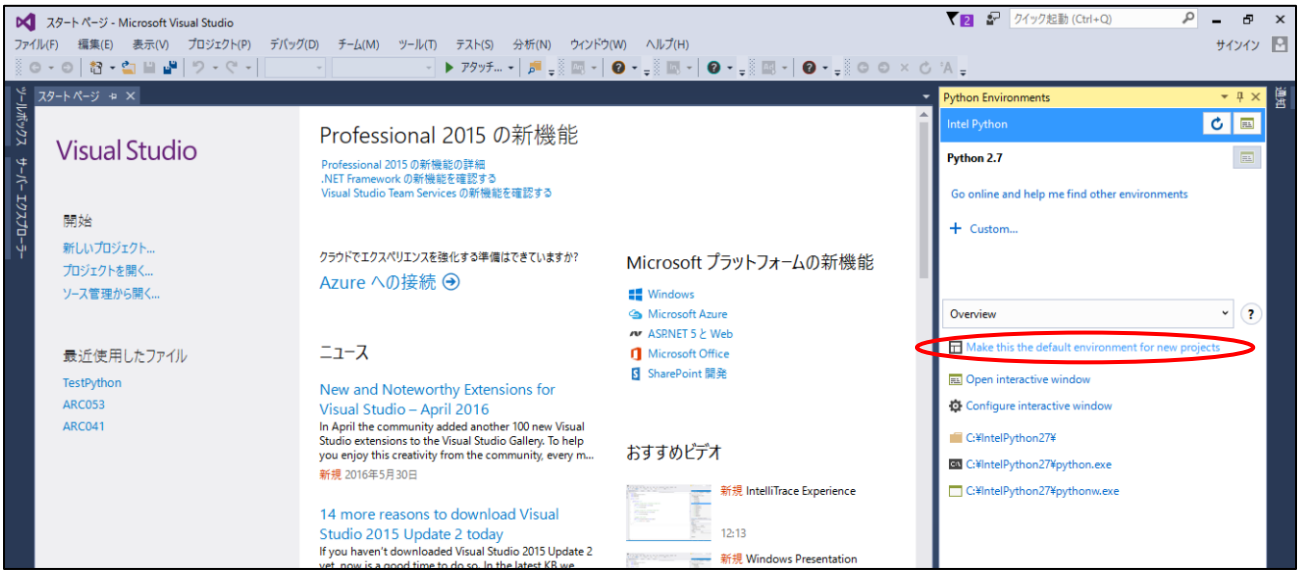
・Profile path : C:¥IntelPython27 または C:¥IntelPython35

その後、[Auto Detect] をクリックすると、その他の項目が自動的に入力されます。自動的に入力されたことを確認したら、[Apply] ボタンをクリックします。



(4) 使用する Python 実行環境を切り替えます

[Intel Python] を選択した状態で [Make this the default environment for new projects] をクリックして、インテル® Python をデフォルトに設定することにより Visual Studio で、インテル® Python 環境を使用してプログラムを実行することができるようになります。



6-2 参考資料

インテル® Advisor 2017 for Windows® スレッド化アドバイザー 入門ガイド
https://www.xlsoft.com/jp/products/intel/advisor/2017/threading_start_guide_win/index.html

インテル® Advisor 2017 for Windows® ベクトル化アドバイザー 入門ガイド
https://www.xlsoft.com/jp/products/intel/advisor/2017/vectorization_start_guide_win/index.html

インテル® VTune™ Amplifier XE / インテル® Advisor XE トレーニングビデオ
https://www.xlsoft.com/jp/products/intel/tech/online-contents.html

インテル® C++/Fortran コンパイラー 17.0 最適化クイック・リファレンス・ガイド
https://jp.xlsoft.com/documents/intel/compiler/17/Quick-Reference-Guide-Intel-Compilers-v17.pdf

The Parallel Universe
https://www.xlsoft.com/jp/products/intel/tech/documents.html#tab2

パフォーマンス解析基本用語 & インテル® VTune™ Amplifier XE 2017 入門ガイド
https://jp.xlsoft.com/documents/intel/vtune/2017/PerformanceAnalysys_GettingStarted_with_IntelVTuneAmplifierXE2017.pdf

インテル® VTune™ Amplifier XE 2017 for Windows® 入門ガイド
https://www.xlsoft.com/jp/products/intel/vtune/2017/start_guide_win/index.html

最適化・並列化の技術情報発信サイト：
http://www.isus.jp/

インテル® VTune™ Amplifier XE を利用した Python* コードの高速化
http://www.isus.jp/products/psxe/pu25-02-run-python-faster-with-vtune/?d=xl