インテル® Parallel Studio XE 2013

ー 入門ガイド ー

エクセルソフト株式会社
www.xlsoft.com

Rev. 1.0 (2012/10/30)

－　目次　－

１．はじめに 4
２．インテル® Parallel Studio XE 2013 概要 4
３．基本使用方法 5
　３－１．基本開発工程 6
　３－２．使用サンプル・プログラム 7
　３－３．サンプル・プログラムのビルド（インテル® Composer XE 使用） 9
　３－４．マルチスレッドの設計（インテル® Advisor XE 使用） 11
　３－５．マルチスレッドの実装（インテル® Composer XE 使用） 17
　３－６．マルチスレッドの診断（インテル® Composer XE 使用） 21
　３－７．マルチスレッドの診断（インテル® Inspector XE 使用） 24
　３－８．マルチスレッドコードの修正（インテル® Composer XE 使用） 27
　３－９．マルチスレッドのチューニング（インテル® VTune Amplifier XE 使用） 30
　３－１０．更なるチューニング（インテル® VTune Amplifier XE 使用） 37
４．関連情報 42
　４－１．インテル® Composer XE 42
　　４－１－１．主要コンパイルオプション 42
　　４－１－２．静的解析機能の検出対象問題 44
　４－２．インテル® Inspector XE 45
　　４－２－１．メモリーチェック機能 45
　　４－２－２．検出対象問題の一覧 46
　　４－２－３．デバッガーでエラーをトラップする方法 46
　　４－２－４．検出オーバーヘッドの軽減方法 48
　　４－２－５．検出中にアプリケーションの状態を確認する方法 49
　　４－２－６．推奨されるコンパイルオプション 50
　　４－２－７．動的検出 vs. 静的検出 50
　　４－２－８．Tips 51
　４－３．インテル® VTune Amplifier XE 52
　　４－３－１．サンプリング・メカニズム 52
　　４－３－２．CPI 値について 53
　　４－３－３．推奨されるコンパイルオプション 54
　　４－３－４．特定箇所のプロファイル方法 54

５．追加情報 57
５－１．ドキュメントの参照方法 57
５－２．N-Queens サンプル・プログラム 58
５－３．OpenMP について 58
５－４．Intel® Cilk™ Plus について 58
５－５．Intel® TBB について 58
６．最後に 59

1．はじめに

本ドキュメントでは、インテル® Parallel Studio XE 2013（以下、本製品）の基本的な使用方法について説明します。説明には本製品とともにインストールされるサンプル・プログラム（C++）を使用して、アプリケーションの並列化を行います。本ドキュメントを通して、本製品に含まれる各ツールの基本機能および並列化の作業手順を習得することができます。またその他、各ツールの補足情報や並列化に関する情報も記載しています。

2．インテル® Parallel Studio XE 2013 概要

インテル® Parallel Studio XE 2013 は、Microsoft* Visual Studio* に統合され、アプリケーションの最適化、並列化を支援するツールであり、以下の 4 つのツールで構成されています。

- インテル® Advisor XE 2013
  - 実際にアプリケーションの並列化を実施する前に、並列化の候補となる処理やパフォーマンス向上の可能性、また並列化によって発生しえ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。
- インテル® Composer XE 2013
  - アプリケーションのパフォーマンスを引き出すインテル® コンパイラーをはじめ、マルチスレッド対応高速ライブラリーを含む最適化・並列化実装ツールです。なお、インテル® Composer XE 2013 には以下のコンポーネントが含まれています。

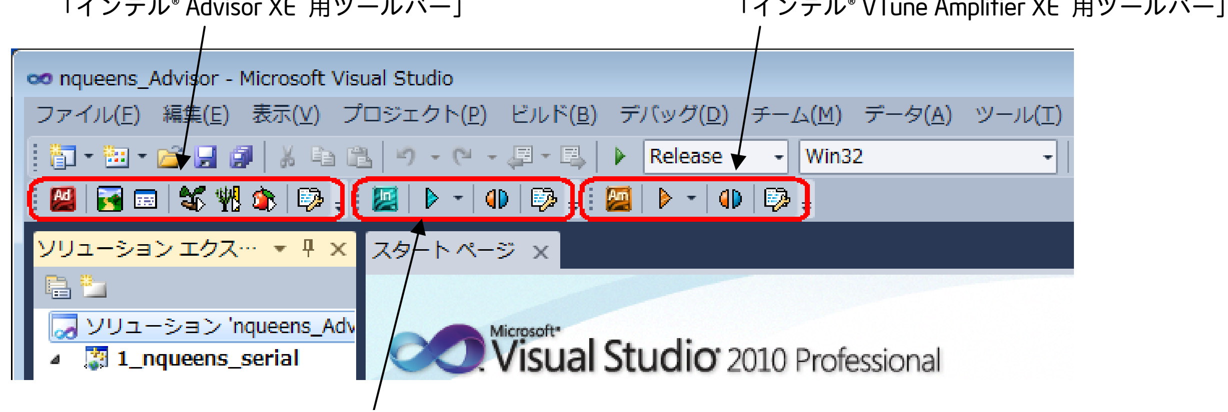
    1）インテル® C++ コンパイラー 13.0 （IA-32 および Intel® 64 対応アプリケーション向け）
    2）インテル® Fortran コンパイラー 13.0（IA-32 および Intel® 64 対応アプリケーション向け）
    3）インテル® インテグレーテッド・パフォーマンス・プリミティブ 7.1（インテル® IPP）
    4）インテル® マス・カーネル・ライブラリー 11.0（インテル® MKL）
    5）インテル® スレッディング・ビルディング・ブロック 4.1（インテル® TBB）
- インテル® Inspector XE 2013
  - プログラム内に潜む「メモリーエラー」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 2013
  - プログラム内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・レベルでのプロファイリングを実施する、分析チューニング・ツールです。

（※ 以下、ツール名のバージョン番号は省いて記載します）

3．基本使用方法

本章では、本製品の基本的な使用方法について説明します。

まず Visual Studio を起動してみましょう。下図のように新しいツールバーが追加されていることを確認してください。（※「インテル® Composer XE 用ツールバー」は、VS2008 にのみ表示されます）



「インテル® Inspector XE 用ツールバー」

本製品には、以下の 4 つのツールが含まれています。

- インテル® Advisor XE
- インテル® Composer XE
- インテル® Inspector XE
- インテル® VTune Amplifier XE

これらのツールはマルチスレッド・プログラミングを支援するためのツールであり、それぞれのツールの役割を理解してマルチスレッド開発工程における位置づけを認識することが大切です。本章では各ツールの使用方法を説明する前に、まず本製品を使用してマルチスレッド・プログラミングを行う基本的な開発工程（ワークフロー）を説明します。その後、サンプル・プログラムを使用して、このワークフローに沿った手順でそれぞれのツールの使い方を説明していきます。

なお、本章で使用するシステムの構成は以下のとおりです。

> プロセッサー：Intel(R) Core(TM) i7-2600 CPU 3.4GHz（コード名：Sandy Bridge、コア数：8）
> OS：Microsoft Windows 7 Professional (x64)
> IDE：Microsoft Visual Studio 2010 Professional（以下、VS2010）

3－1．基本開発工程

ここでは、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）　現在のシリアル・アプリケーションに対してインテル® Advisor XE を使用し、並列化導入の設計（デザイン）を行い、どのようにマルチスレッド処理を実装するのかの方針を決定します。

（２）　決定された方針に従って、実際にマルチスレッド処理を実装します。

（３）　インテル® Composer XE を使用してマルチスレッド・アプリケーションをビルドします。

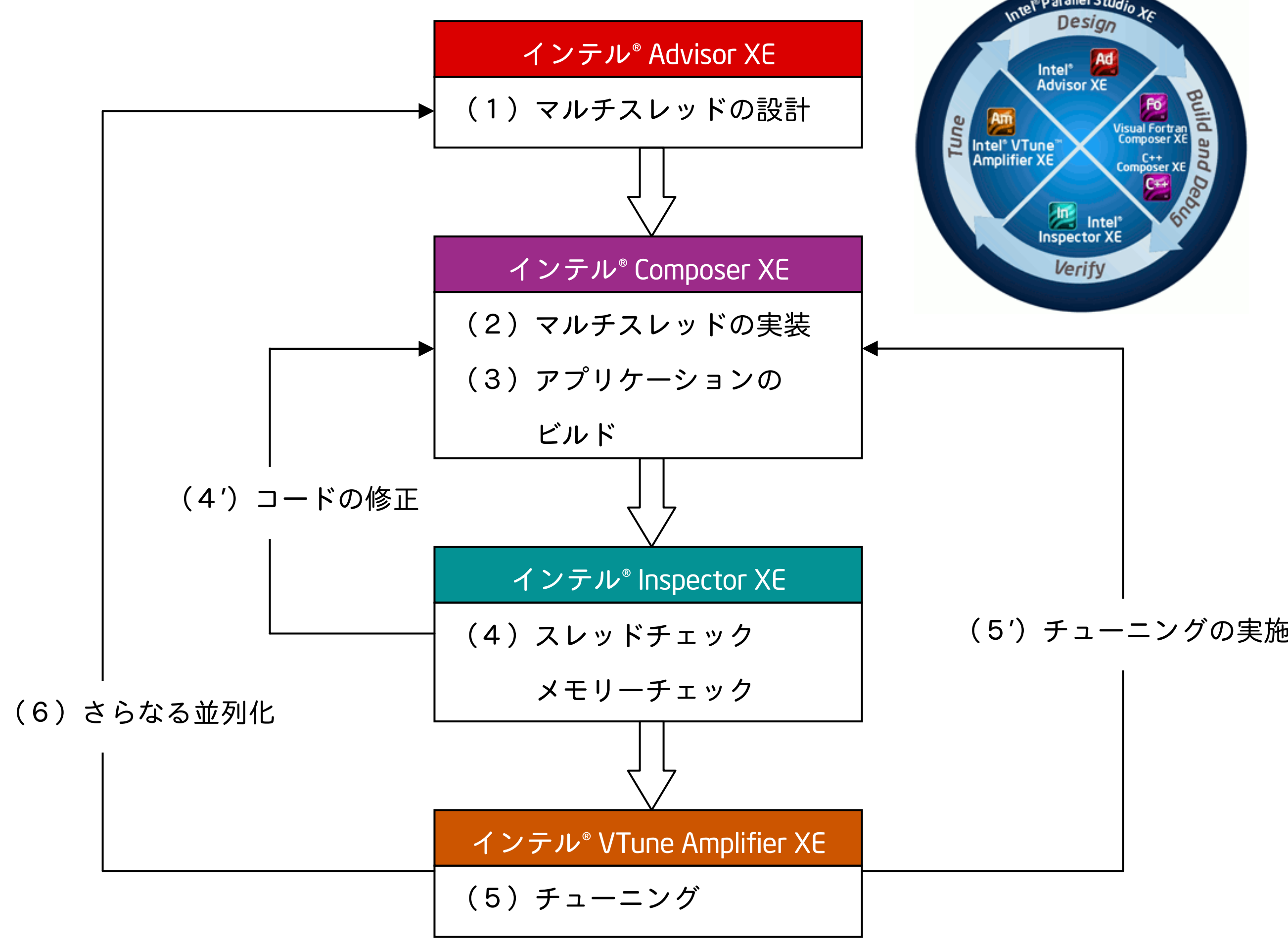
（４）　インテル® Inspector XE で、マルチスレッド処理のエラーチェックを実施します。

（４'）　エラーが存在する場合は、コードを修正して再コンパイルしエラーがないことを確認します。

（５）　インテル® VTune Amplifier XE で、実装したマルチスレッドを分析してチューニングを行います。

（５'）　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。

（６）　さらに並列化可能な処理があるか、インテル® Advisor XE を使用して検証します。



３－２．使用サンプル・プログラム

本章で使用するサンプル・プログラムは、インテル® Advisor XE に付属する N-Queens サンプル・プロジェクトです。本製品をデフォルト設定でインストールした場合は、この N-Queens サンプル・プロジェクトは以下の Zip ファイルとしてインストールされます。

C:¥Program Files (x86)¥Intel¥Advisor XE 2013¥samples¥en¥C++¥nqueens_Advisor.zip

（※ 上記の Zip ファイルは、適当な作業フォルダーにコピーしてご利用ください）

この Zip ファイルを解凍すると、複数のプロジェクトファイルが生成されます。本章で使用するサンプル・プログラムは、1_nqueens_serial プロジェクトに含まれる以下のファイルです。

¥nqueens_Advisor¥1_nqueens_serial¥nqueens_serial.cpp

この N-Queens プロジェクトは、ある特定のマス数のチェスボードにおいて、Queen の駒を安全に配置できるパターンの総数を求める問題です。チェスのルールで Queen は縦、横、斜め（将棋の飛車と角行を合わせた動き）に動けるので、この動作範囲を考慮して Queen 同士で衝突しない安全なマスに各 Queen を配置することを考えます。また求めるパターン数は、パターン間の対称性を考慮します。つまりチェスボードを回転して同じパターンになる場合や、鏡のように左右、上下対称となる場合は同一のパターンとみなします。この N-Queens 問題は古くから解かれており、1850 年に正しい答えが見つかりました。また 1969 年に初めて一般的な解法が発見されています。

インテル® Advisor XE に付属する N-Queens サンプル・プロジェクトでは、この N-Queens 問題に対して複数の並列化手法を使用したサンプルが紹介されています。本章では並列化が実装される前のベースプロジェクト（1_nqueens_serial）のソースコードを使用して、本製品の基本使用方法を前節で紹介したワークフローに沿って説明します。

使用するサンプル・ソース・コード（nqueens_serial.cpp）の内容を簡単に説明します。

次ページでサンプル・プログラムの主要関数を簡略化した内容を記載していますが、まず main() 関数の引数の値がチェスボードのサイズ （N × N）のマス数として指定されています。引数が指定されていない場合のデフォルト値は 14 となっています。また main() 関数内で N-Queens 問題を解くための関数 solve() がコールされ、この関数の計算時間を timeGetTime() 関数を使用して計っています。solve() 関数ではループ文にて計算エンジン関数 setQueen() が N マス回実行されます。このループでは、チェスボードの列の値（i）をインクリメントしており、Queen のポジション（0 行、i 列）を初期計算値として setQueen() 関数に与えられています。そして setQueen() 関数では、この初期値から Queen を安全に配置できる次の（行、列）を求めるため、 setQueen() 関数を再帰的にコールしています。そしてこの再帰処理で見つかったパターンの数が随時グローバル変数である nrOfSolutions に加算され最終的な解答が求められます。

```
int main(int argc, char*argv[]) {
    if(argc !=2) {
        cerr << "Usage: 1_nqueens_serial.exe boardSize [default is 14].¥n";
        size = 14;          // ← チェスボードのデフォルトサイズ
    } else {
        size = atoi(argv[1]);
        if (size < 4 || size > 15) {
            cerr << "Boardsize should be between 4 and 15; setting it to 14. ¥n" << endl;
            size = 14;      // ← チェスボードのデフォルトサイズ
        }
    }

    （中略）

    DWORD startTime=timeGetTime();
    solve();      // N-Queens 問題を解く関数をコール
    DWORD endTime=timeGetTime();

    （中略）

    return 0;
}
```

```
void solve() {

    int * queens = new int[size];

    for(int i=0; i<size; i++) {
        setQueen(queens, 0, i); // 計算エンジン関数をNマス回コール
    }
}
```

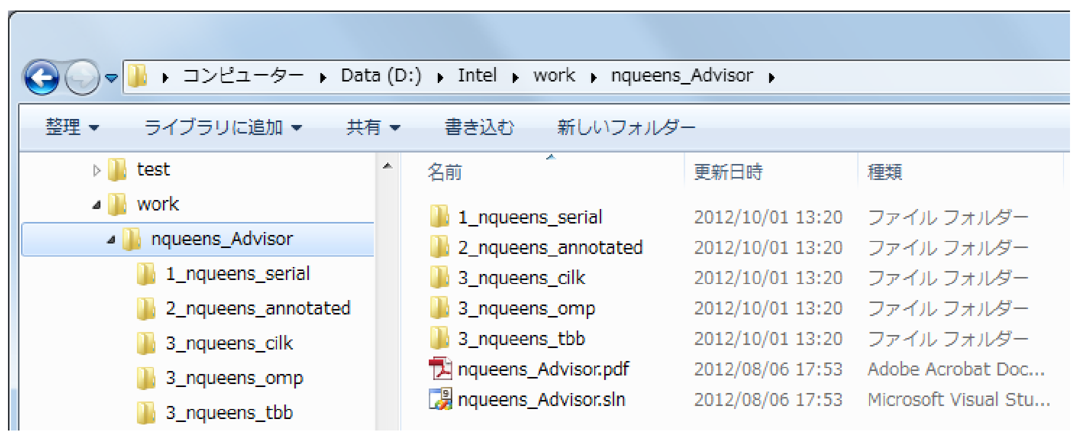
```
void setQueen(int queens[], int row, int col) {
    //check all previously placed rows for attacks
    for(int i=0; i<row; i++) {
        // vertical attacks
        if (queens[i]==col) {
            return;
        }
        // diagonal attacks
        if (abs(queens[i]-col) == (row-i) ) {
            return;
        }
    }
    // column is ok, set the queen
    queens[row]=col;

    if(row==size-1) {
        nrOfSolutions++;        //N-Queens 問題の解答を格納
    }
    else {
        // try to fill next row
        for(int i=0; i<size; i++) {
            setQueen(queens, row+1, i);     // 次の行を計算する再帰関数
        }
    }
}
```

３－３．サンプル・プログラムのビルド（インテル® Composer XE 使用）

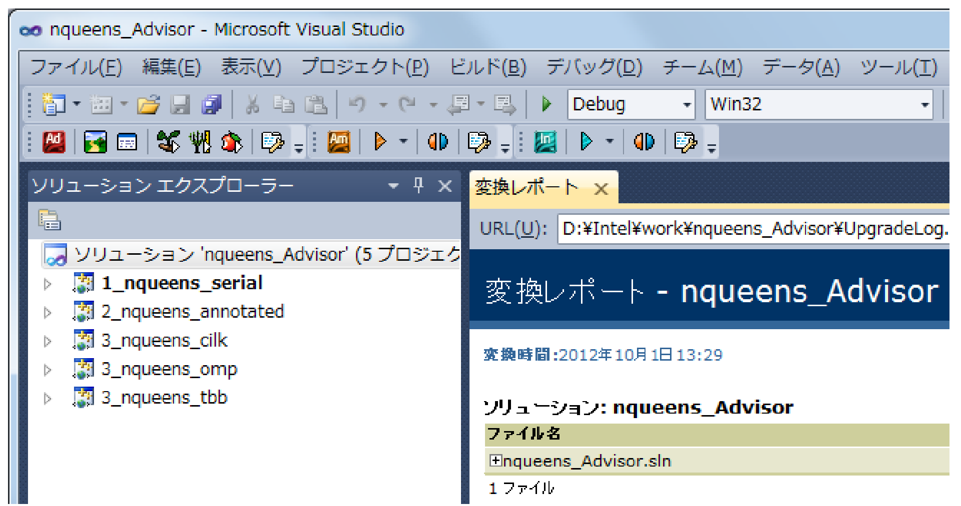
まず、サンプル・プログラムをビルドして実行ファイルを生成する必要があります。VS2010 でサンプル・プログラムのプロジェクトを開き、インテル® C++ コンパイラーを使用してビルドを実施します。

＜ビルド手順＞

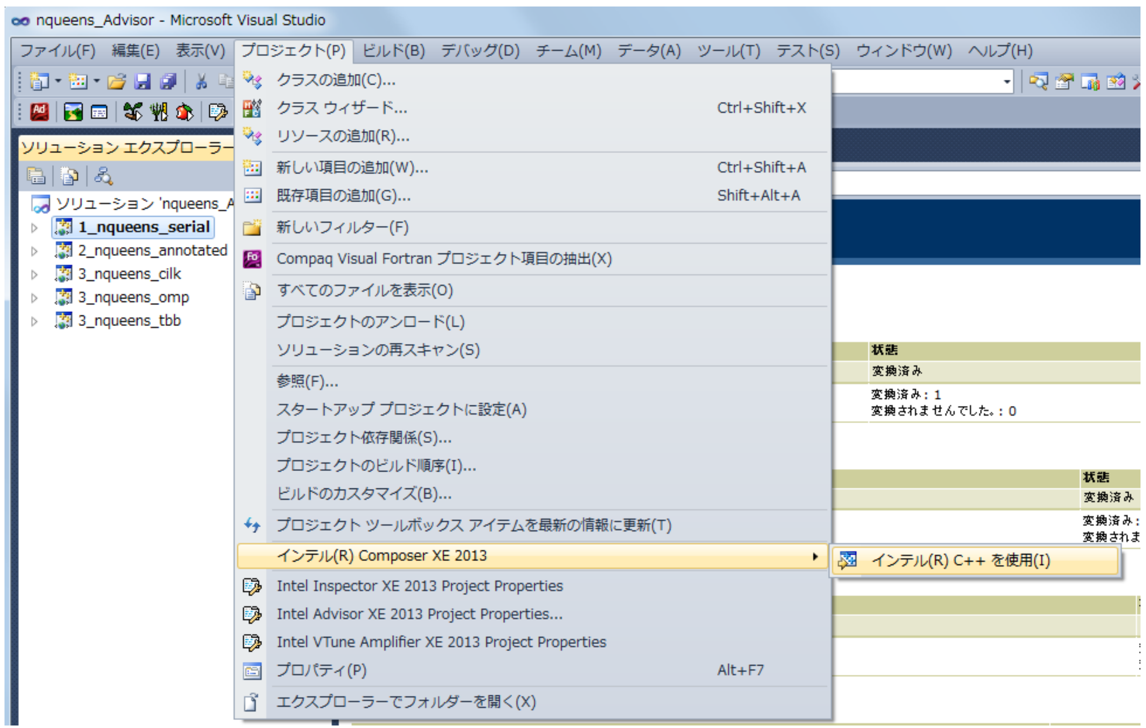
1. サンプル・プログラム（nqueens_Advisor.zip）を適当な作業フォルダーに展開します。



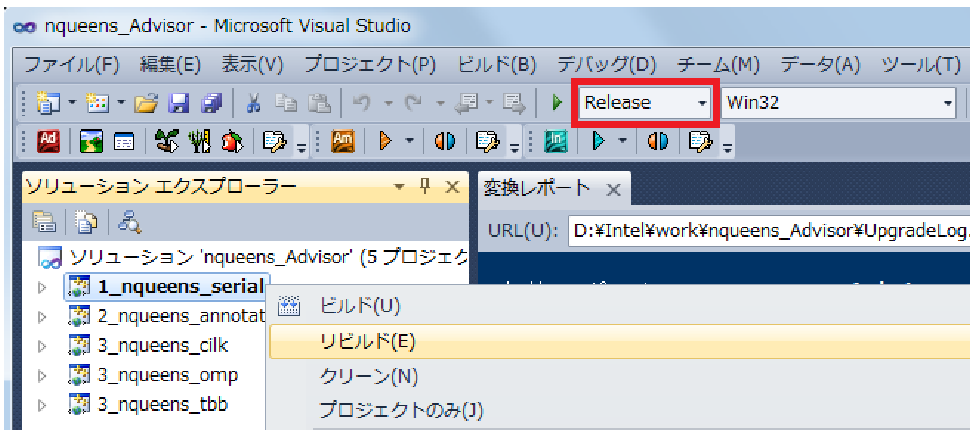
2. “nqueens_Advisor.sln”ソリューションファイルを、VS2010 で開きます。このときソリューションの変換処理が実行されます。



3. 使用コンパイラーをインテル® C++ コンパイラーに切り替えます。“1_nqueens_serial”を選択した状態で、[プロジェクト] - [インテル(R) Composer XE 2013] - [インテル(R) C++ を使用] を選択します。または、プロジェクトを右クリックして表示されるコンテキストメニューからも切り替えることができます。



4. Release モードに切り替えてビルド（リビルド）します。結果は出力ウィンドウで確認します。



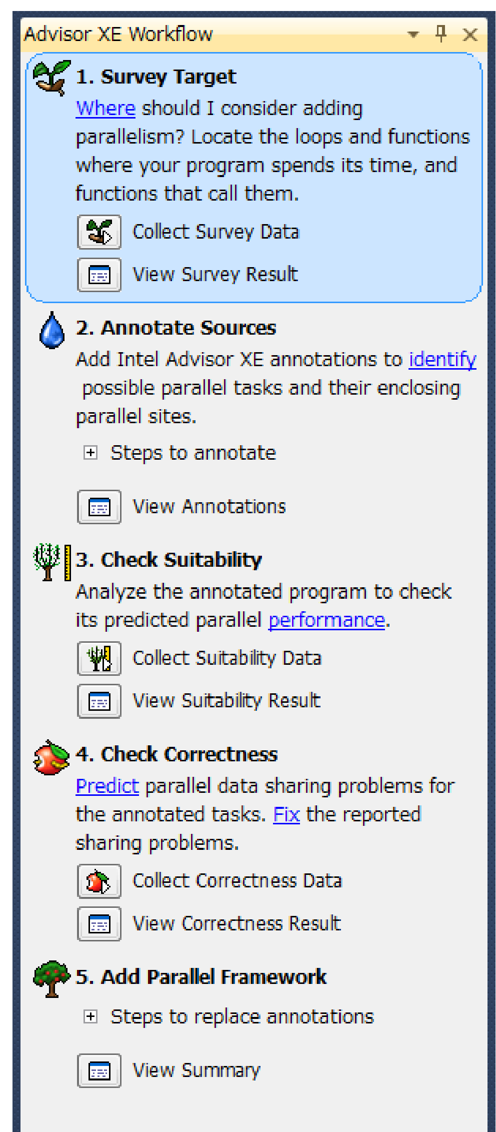
5. [デバッグ] - [デバッグなしで開始] で実行します。マスサイズ１４の時の答えが 365596 であることが分かります。また、この実行でかかった計算時間も表示されています。

C:¥Windows¥system32¥cmd.exe

```
Usage: 1_nqueens_serial.exe boardSize [default is 14].
Starting nqueens solver for size 14...
Number of solutions: 365596
Correct result!

Calculations took 5400ms.
続行するには何かキーを押してください . . .
```

３－４．マルチスレッドの設計（インテル® Advisor XE 使用）

ここでは、インテル® Advisor XE を使用して並列化の実装内容を決定します。インテル® Advisor XE は、以下のツールバーのボタンをクリックして表示される「Advisor XE Workflow」に沿って作業を進めることができます。以下に手順概要を記します。



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処理（Hotspot という）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。

手順２：Annotate Sources（アノテーションの挿入）

並列化候補の処理に対し、アノテーション（インテル® Advisor XE が提供するマクロ処理）を挿入します。

手順３：Check Suitability（並列化内容の分析）

アノテーションが付加されたアプリケーションを実行し、擬似的な並列動作内容を表示します。この結果を元に並列化性能の期待値を評価します。

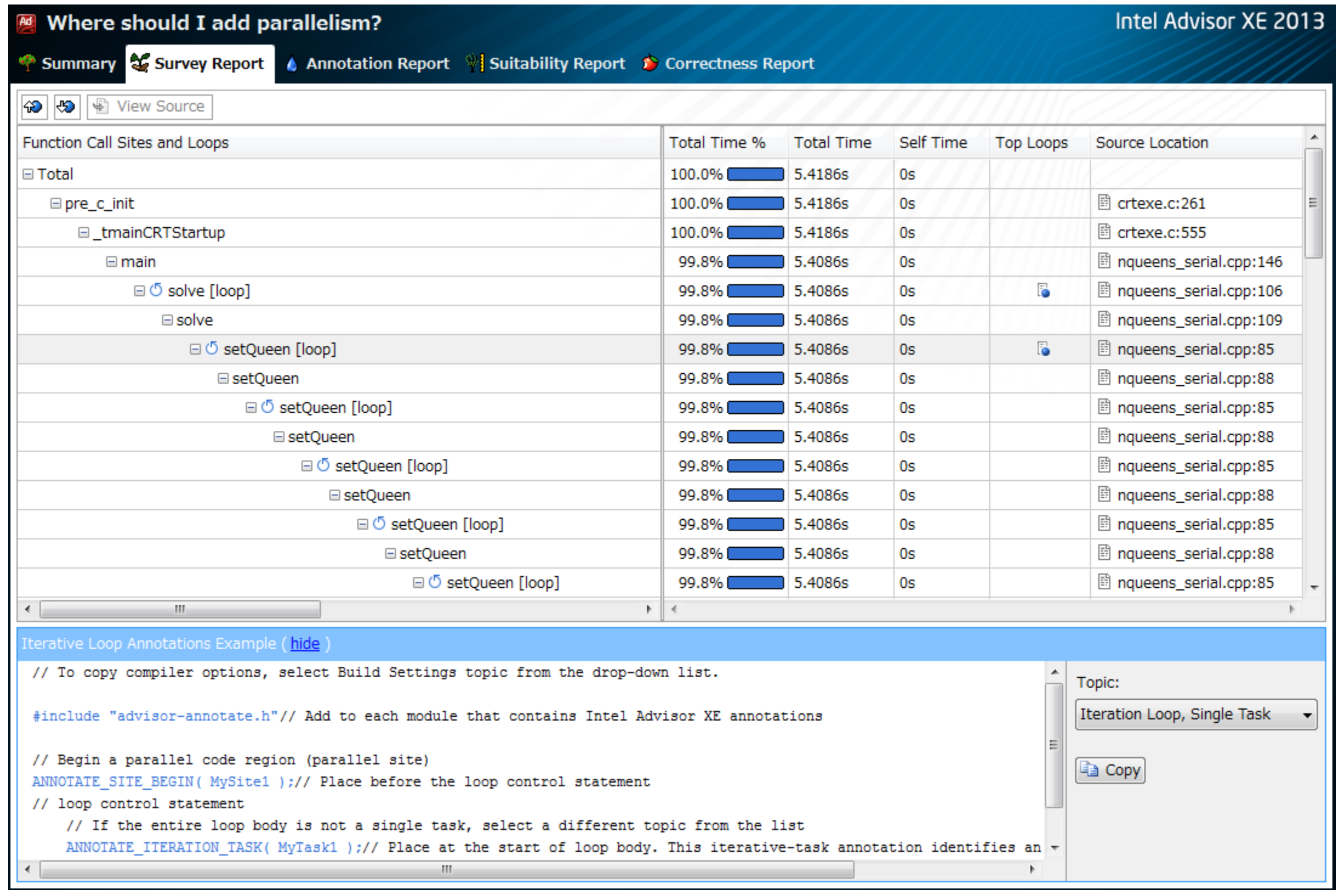
手順４：Check Correctness（並列化問題の予測）

アノテーションが付加されたアプリケーションを実行し、データ競合などの並列化問題が発生する可能性を確認します。

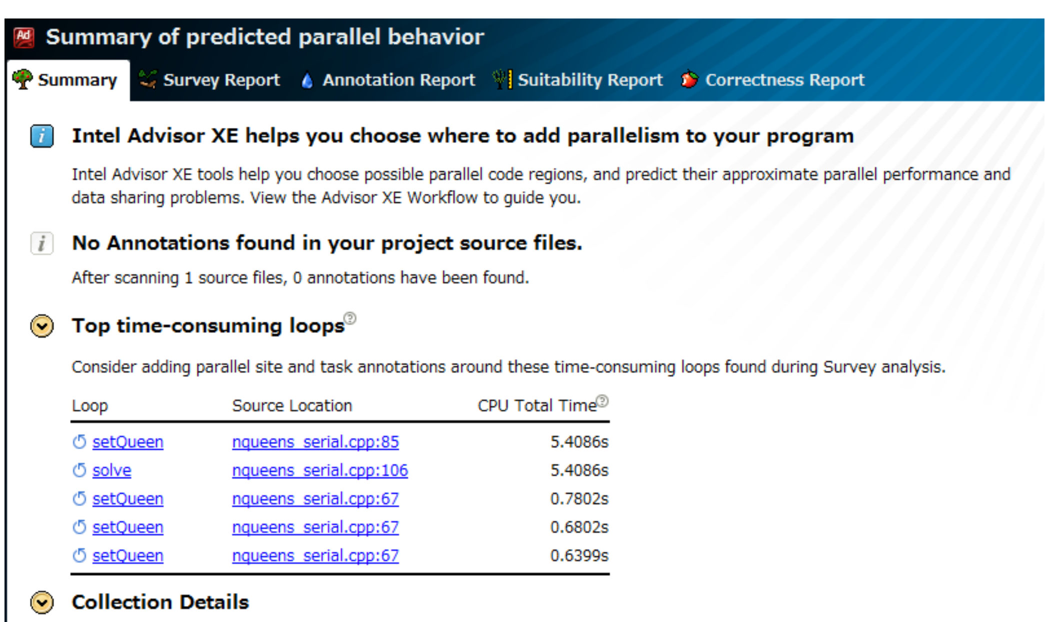
手順５：Add Parallel Framework（並列化処理の実装）

挿入したアノテーションの内容を参考に、ソースコードに実際の並列化処理を実装します。

それではまず、手順１の「Survey Target」を実行します。ボタンをクリックして調査を開始します。プログラムの実行が完了すると以下のような Survey Report 画面が表示されます。この画面では Hotspot までの関数コールトレースが表示されます。ループ処理がある箇所は のマークが表示されていることが分かります。そして Top Loops カラムにはその中でも特に Total Time が大きいループがチェックされています。



さらに Summary 画面を見ると Top Loops の情報が列挙されており、並列処理を実装する候補として参照することができます。



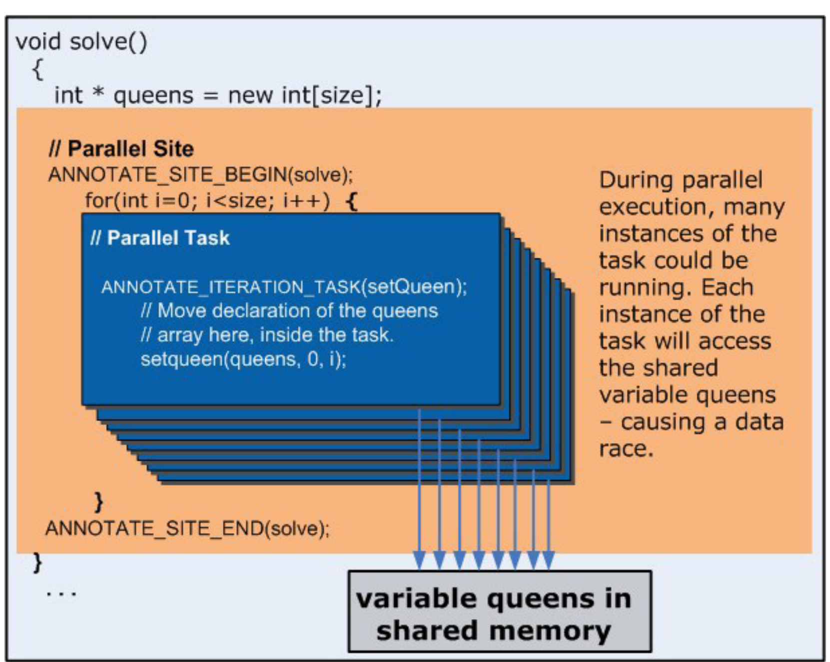
ここでは、親関数である solve 関数を並列化の対象関数とし、同関数内のループ文（nqueens_serial.cpp ファイルの 106 行目）に並列処理を導入することを検討します。

次に手順２として、solve 関数内のループ文に対して「アノテーションの挿入」を行います。

本サンプル・プログラムの solve 関数には以下のように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ます。ここでは３つのコメントを外すことで作業が完了します。

```
void solve() {
    int * queens = new int[size];
    //ANNOTATE_SITE_BEGIN(solve);  ←このコメントを外す
        for(int i=0; i<size; i++) {
            // try all positions in first row
            //ANNOTATE_ITERATION_TASK(setQueen);  ←このコメントを外す
                setQueen(queens, 0, i);
        }
    //ANNOTATE_SITE_END();  ←このコメントを外す
}
```

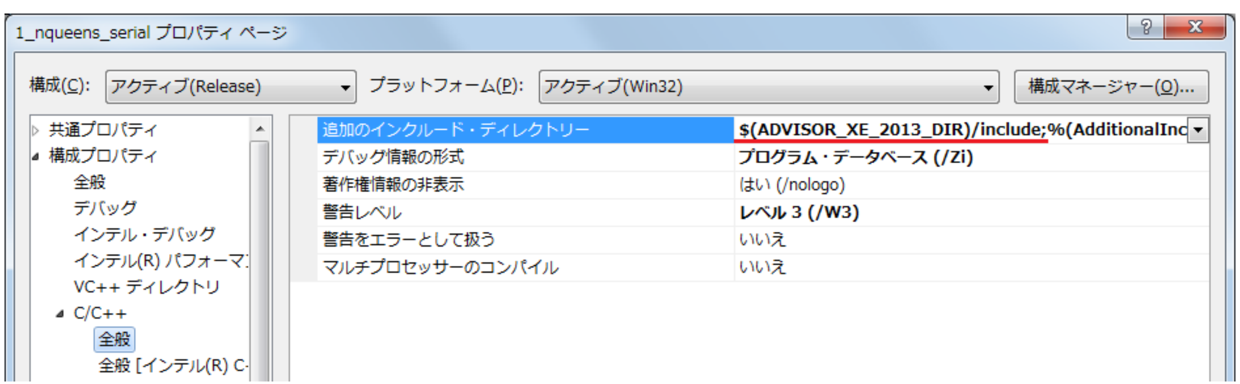
上記のようにループ文の外側を ANNOTATE_SITE_BEGIN と ANNOTATE_SITE_END で囲み、ループ内の setQueen 関数に対して ANNOTATE_ITERATION_TASK を指定することで、次の手順の「Check Suitability」実行の際に、以下の図のようにループ文が並列実行領域、そして setQueen 関数がタスクとして見なされ、タスクはループの反復回数（size）存在し、各タスクは同時実行するものと扱われます。



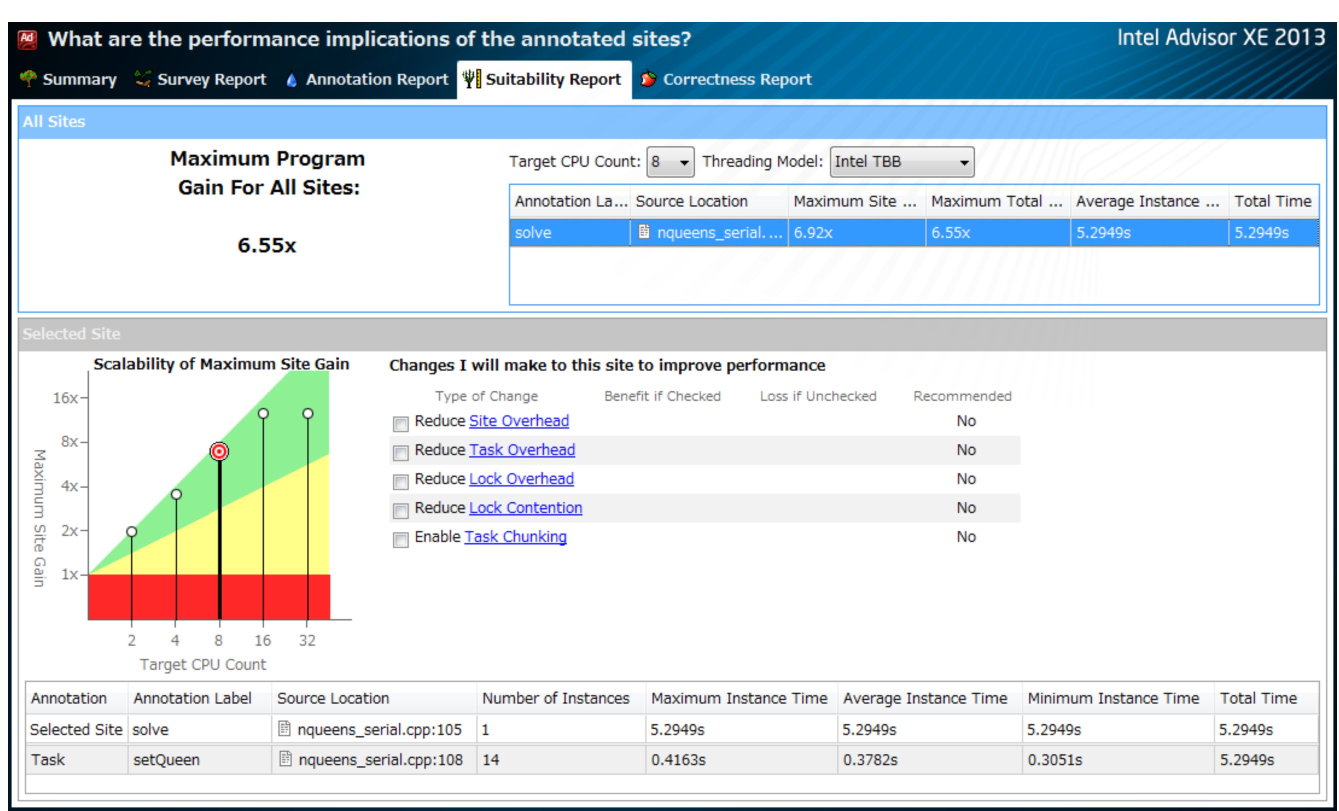
なお、このアノテーションを使用する場合は、ヘッダーファイル（advisor-annotate.h）をインクルードする必要があります。nqueens_serial.cpp ファイルの上のほうにある以下のコメントを外してください。

```
//#include <advisor-annotate.h>
```

また、以下の図のように「追加のインクルード・ディレクトリー」に “$(ADVISOR_XE_2013_DIR)/include” を追加します。設定が完了したら、ビルドを実施して正常終了していることを確認します。

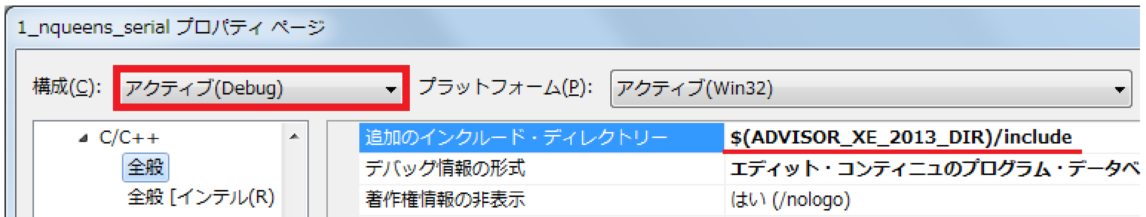


次に手順３として、アノテーションを記述したアプリケーションに対して「Check Suitability」を実行します。 ボタンをクリックします。実行が完了すると以下のような Suitability Report 画面が表示されます。

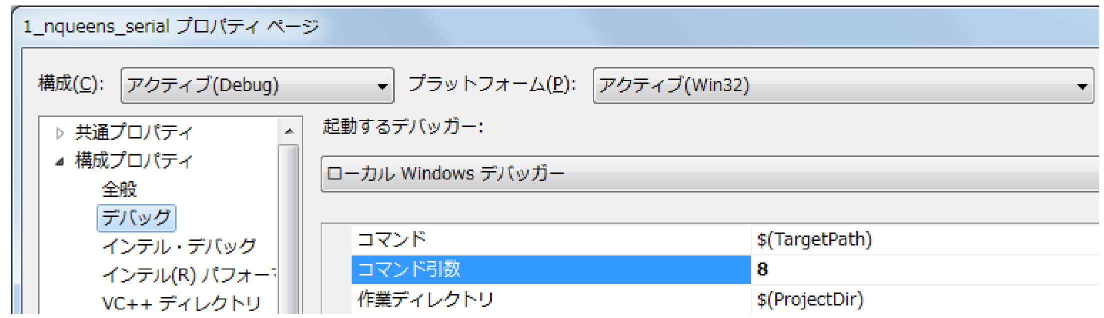


Suitability Report 画面では上下２つのペインが表示され、上側（All Sites）がすべてのサイト情報（並列実行領域）、下側（Selected Site）は All Sites ペインで選択したサイトの詳細情報が表示されます。ここではソースコード上に定義した一つのサイト情報が表示されています。All Sites ペインでは、プログラム全体のパフォーマンス・ゲインとサイト単位のパフォーマンス・ゲインが示されています。Selected Site ペインでは、タスク実行に関するデータとコア数に応じたパフォーマンス・スケール情報、またパフォーマンス向上に関するヒントが表示されています。これらの性能情報を基にこのサイトに対する並列化実装の価値を検討します。

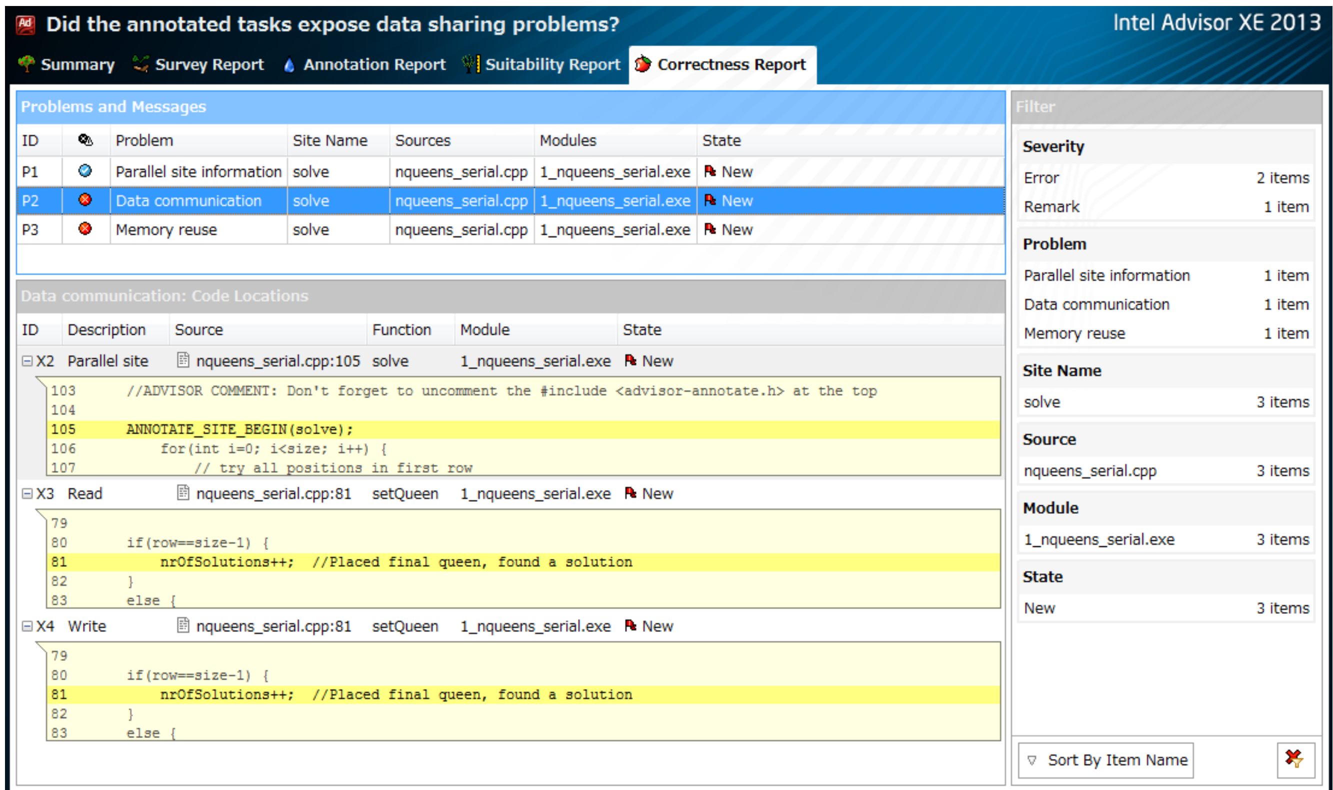
次に手順４として「Check Correctness」を実行しこのサイトで発生しえる並列化問題を検証します。Check Correctness を実行する場合は Debug モードが推奨されますので、プロジェクトを Debug モードに切り替え、プロジェクトのプロパティページからインクルード・ディレクトリーの追加を再度設定してビルドします。



また実行に際してプログラムに渡す引数（チェスボードのサイズ）を 8 とし実行処理量を軽減します。

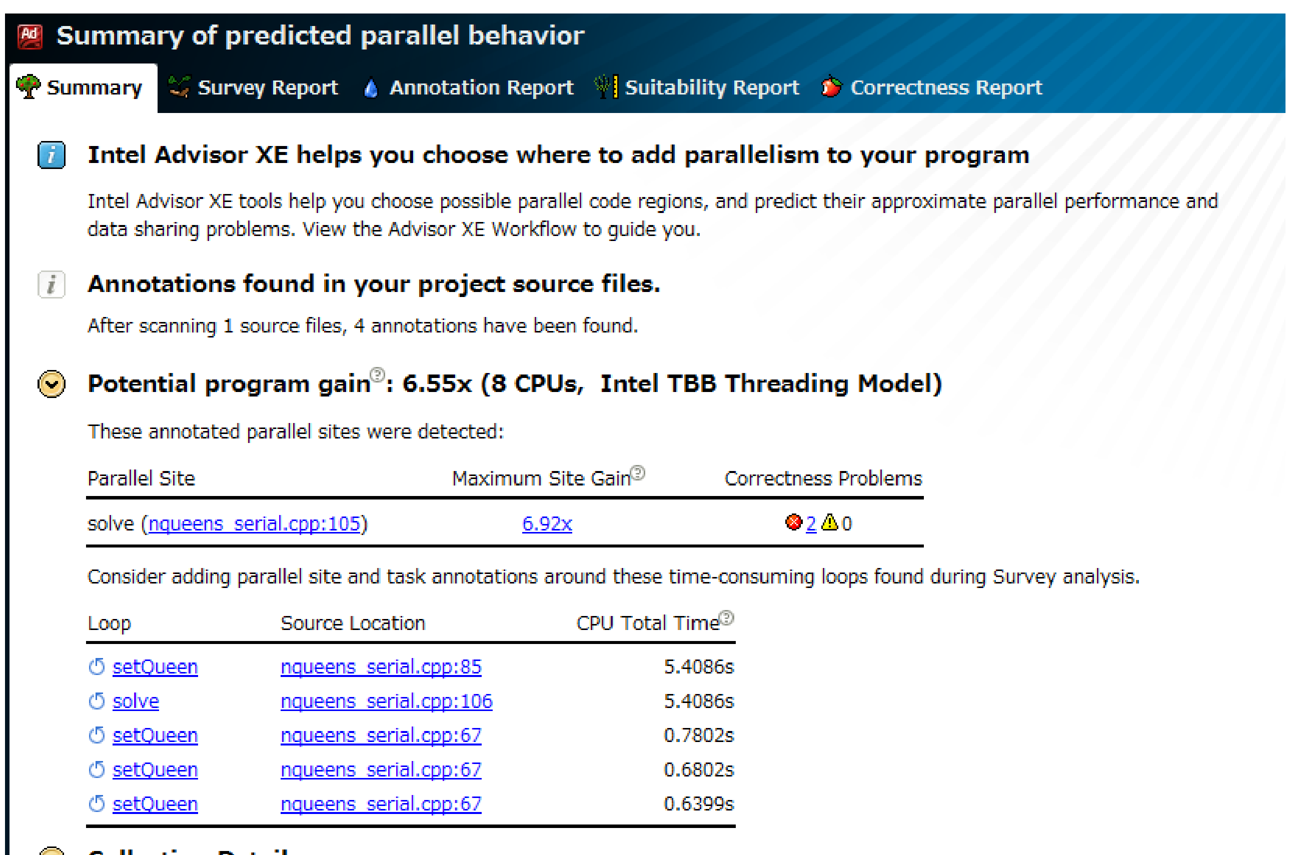


ボタンをクリックして Check Correctness を実行します。実行が完了すると以下のような結果が表示されます。



表示される「Correctness Report」画面で、「Problems and Messages」ペインには問題の件数や問題の種類などが表示され、発生しえる問題の全体像が把握できます。ここでは、P2 の“Data Communication”と P3 の“Memory reuse”の 2 つの問題が提示されています。このペインで選択された問題の詳細は、下のペインに表示されます。また、この問題をダブルクリックするとさらに詳細な画面へと切り替わります。

最後の手順として Summary ページを開き、期待されるパフォーマンス・ゲインと考慮すべき並列化問題のコストを考えて最終的な決定を行います。ここではこのサイトに対して並列化を実装します。



３－５．マルチスレッドの実装（インテル® Composer XE 使用）

インテル® Composer XE がサポートするマルチスレッド方法には大きく以下の４種類があります。

１）Win 32 スレッド API

２）OpenMP（バージョン 3.1 対応）

３）インテル® Cilk™ Plus

４）インテル® Threading Building Blocks（インテル TBB）

Win 32 スレッド API は、マイクロソフト社が提供するスレッド・ライブラリーで、CreateThread 関数や _beginthread 関数などを使用してスレッドの生成を行う最も基本的なマルチスレッド方法です。この方法を使用した場合は、スレッドの作成や管理、またスレッド間の同期処理などきめ細かな制御ができ、プログラマーにとって自由度の高いマルチスレッド実装が可能となります。しかしその反面、コーディング量の著しい増加やメンテナンス性の低下、また複雑化するプログラミングに伴う不具合の増加が発生します。一般的に Win 32 スレッド API は、他の３つのハイレベルな言語に対して、並列化のアセンブリ言語と位置づけられます。

OpenMP は、マルチスレッドによる並列処理を実装するための業界標準規格です。この OpenMP では、既存のシリアルコードに #pragma omp 指示キーワードを使用してマルチスレッドを実現します。この方法を利用した場合の利点は、何といってもコードの修正量が極めて少ないということです。Win 32 スレッド API での細かなスレッド制御は OpenMP で規定された動作で処理されるのでプログラマーの負担が軽減されると同時にプログラマーはシリアルコードからパラレルコードへのタスク制御処理に集中することが出来ます。また OpenMP はランタイムでプロセッサー･コア数などを自動で検出しますので実行環境に適したパフォーマンスを発揮します。またコンパイラーレベルで認識する #pragma 指示キーワードを使用するスタイルのため、コンパイラーオプションによってマルチスレッド実装の制御が可能となり、たとえば開発中におけるデバッグの際は #pragma 指示キーワードを無視するようにコンパイルを行い、シリアルコードでの処理の確認が出来るようになります。さらに、システムアーキテクチャーや OS 間での移植も容易になるという利点もあります。OpenMP では、並列実行領域内でワーカースレッドの生成、並列処理の実施、処理完了の待ち合わせが暗黙的に行われており、“Fork-Join”モデルのマルチスレッド方法とも呼ばれています。

インテル® Cilk™ Plus は、インテル® C++ コンパイラーでサポートされる並列化のための言語拡張です。インテル® Cilk™ Plus では並列化をシンプルに構築でき、しかもパフォーマンスに優れた実装が可能となります。並列化を実装するためのキーワードはわずか３つしかなく、cilk_spawn、cilk_sync、cilk_for で構成されます。この３つのキーワードに加えレデューサーという機能が用意され、データ競合が発生する変数に対して適用する仕組みが整っています。シンプル性を重視した設計となっているためあまり細かい動作設定はできませんが、ロードバランスを自動制御するスチール機能などを有し、ユーザーは手軽に高い性能を持った並列化を実装することができます。

インテル® TBB は、インテル社が提供する C++ プログラマー向けのマルチスレッド・ライブラリーです。このインテル® TBB は、STL のようにテンプレート･ライブラリーとして提供されます。インテル® TBB では多数の並列化用テンプレート・クラスが定義されており、プログラマーはこれらのテンプレート・クラスを適用することにより、スレッド制御を特に意識することなくハイパフォーマンスなマルチスレッドを実装することが可能となります。また、インテル® TBB は、タスクベースな並列化概念で設計されており、OpenMP と比較すると、さらに詳細な設定を行うことができます。C++ のオブジェクト指向言語を使用するプログラマーは、この方法を使用してマルチスレッドを実装することになります。

ここでは、前節のマルチスレッドの設計で得た考察を基に、サンプル・プログラムの “solve”関数に対して OpenMP を使用したマルチスレッドの実装方法を説明します。
OpenMP では、たくさんの構文や宣言子などが定義されています。OpenMP を使用するには、まず以下の宣言子を使用して並列実行領域を明確化する必要があります。

```
#pragma omp parallel
{
    ～並列実行領域～
}
```

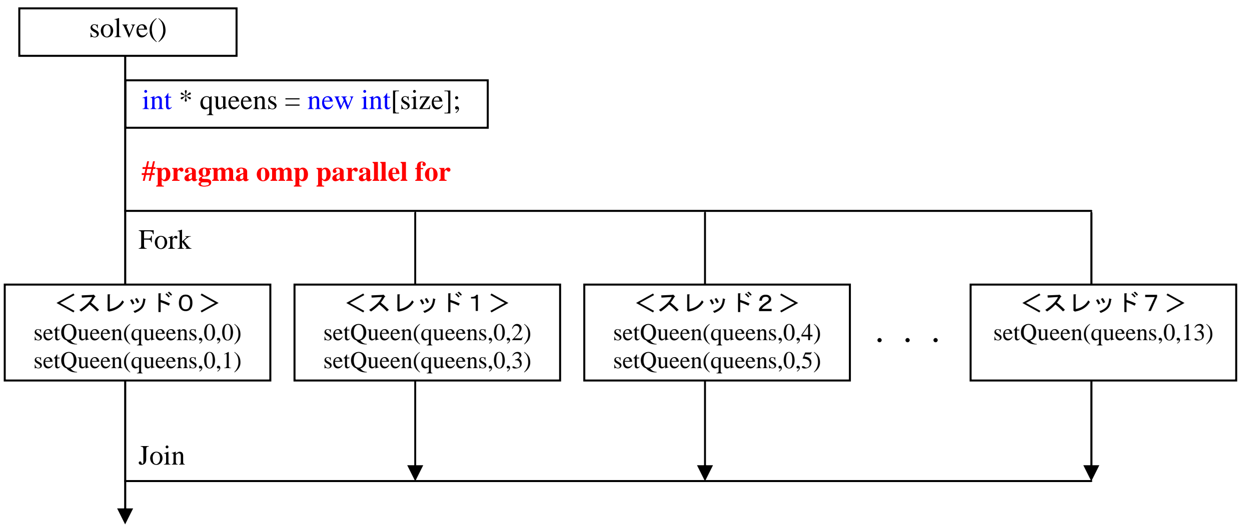
この領域内に記述されるコードは基本的に複数のスレッドによって実行される処理となります。たとえばこの領域に printf 文があった場合、生成される各スレッドで処理が実行されるので複数の printf 出力が表示されます。この生成されるスレッド数はデフォルトでは OS にて認識されるプロセッサー・コア数になります。
並列実行領域構文のほかに、ここではワークシェアリング構文を使用する必要があります。ワークシェアリング構文では、並列実行領域内でワークロード（処理されるべき仕事量）をスレッド間で分担します。この分担方法（スケジュール）は特に指定がない限り自動で配分されます。複数のワークシェアリング構文が存在しますが、ここでは以下の for 宣言子を使用します。

```
#pragma omp for
    For( i=0; i<N; i++ ) // 0 ～ N までのワークロードをスレッド間で自動分配
    {
        ・・・仕事内容・・・
    }
```

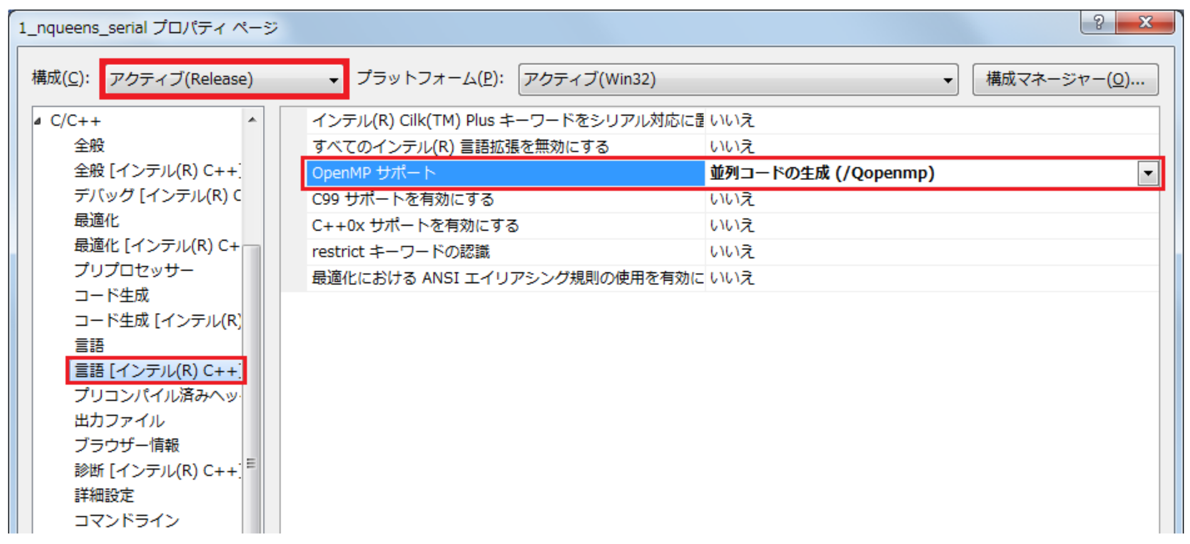
本サンプルでは、上記で説明した並列実行領域構文とワークシェアリング構文を同時に宣言して以下のような記述方法で並列化を行います。（※ インテル® Advisor XE で使用したアノテーションは外してあります）

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

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

上記のように記述した場合 OpenMP はデフォルトでコア数のスレッドを生成し for ループ文の反復回数（size）をスレッド間で分担して仕事（setQueen）を並列実行することになります。つまり、“size”の値が 14（i=0～13）で、生成されるスレッド数が８の場合（本ドキュメントでは Intel(R) Core(TM) i7-2600 を使用）、スレッド０が i= 0～1 を担当、スレッド１が 2～3、スレッド２が 4～5 と続いてスレッド６が 12、最後のスレッド７が 13 を担当するように分配されて８本のスレッドが同時に独立して setQueen の実行をすることになります。次の図は、この並列実行ロジックのイメージを示しています。



それでは、上記のように “#pragma omp parallel for”構文を for ループ文の直前に記述して“Release”構成でビルドしてみましょう。OpenMP を使用したコードをコンパイルする場合は、コンパイラーに “#pragma omp”の指示キーワードを認識させる必要があります。インテル® C++ コンパイラーでは /Qopenmp コンパイルオプションを指定することによりコンパイルが可能となります。 “1_nqueens_serial”プロジェクトのプロパティページを開いて下図のように OpenMP による並列化を有効にしてください。



オプション設定を保存してビルドを行い、正常終了を確認して実行します。以下のような結果が表示されます。

```
C:¥Windows¥system32¥cmd.exe
Usage: 1_nqueens_serial.exe boardSize [default is 14].
Starting nqueens solver for size 14...
Number of solutions: 1209935
!!Incorrect result!! Number of solutions should be 365596


Calculations took 2805ms.
続行するには何かキーを押してください . . .
```

N-Queens のサイズ 14 の答えは、365596 であるため、ここで得られた答えは間違っています。なお、得られる答えは実行する度に異なった値となります。並列化を実装したことで何か問題が発生しているようです。この問題は、すでにインテル® Advisor XE の「Check Correctness」である程度検討がついていますが、ここでは製品の機能紹介も含めて、次節以降に次の 2 つの方法による問題検出方法を説明します。

- インテル® Composer XE による静的解析
- インテル® Inspector XE による動的解析

「静的解析」は、アプリケーションを実行させないで、ソースコード・レベルで問題を検証する手法であり、**「動的解析」**は、アプリケーションを実行させながら、問題を検証する手法です。

3－6．マルチスレッドの診断（インテル® Composer XE 使用）

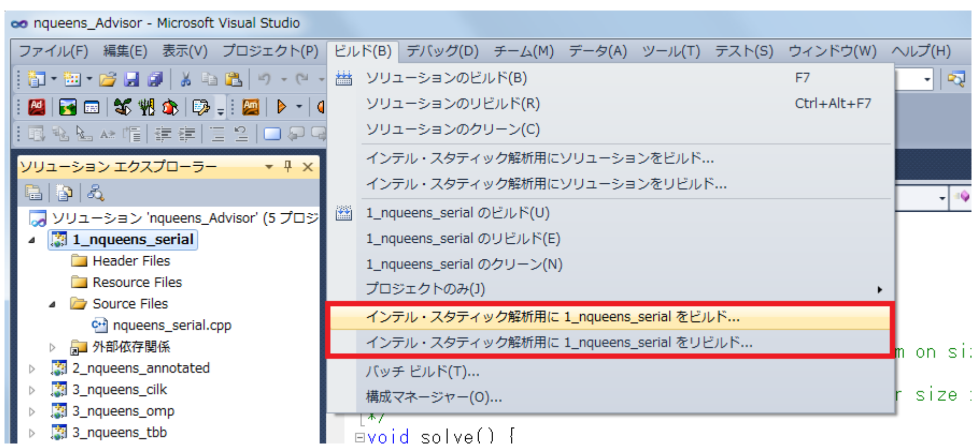
ここでは、インテル® Composer XE の静的解析を実施します。

この静的解析は、インテル® コンパイラーに含まれる機能でソフトウェアの脆弱性となりうる問題、たとえばバッファー・オーバーフローやメモリー違反、ポインタや動的メモリー領域の不正使用、メモリーリーク、言語に関する問題、OpenMP* やIntel® Cilk™ Plusの並列プログラミングに関する問題などを検出します。この機能は、インテル® コンパイラーで解析を行い、結果はインテル® Inspector XE のGUIを利用して表示します。インテルコンパイラーはビルド工程を利用して本機能を実施しますが、バイナリーは生成しません。

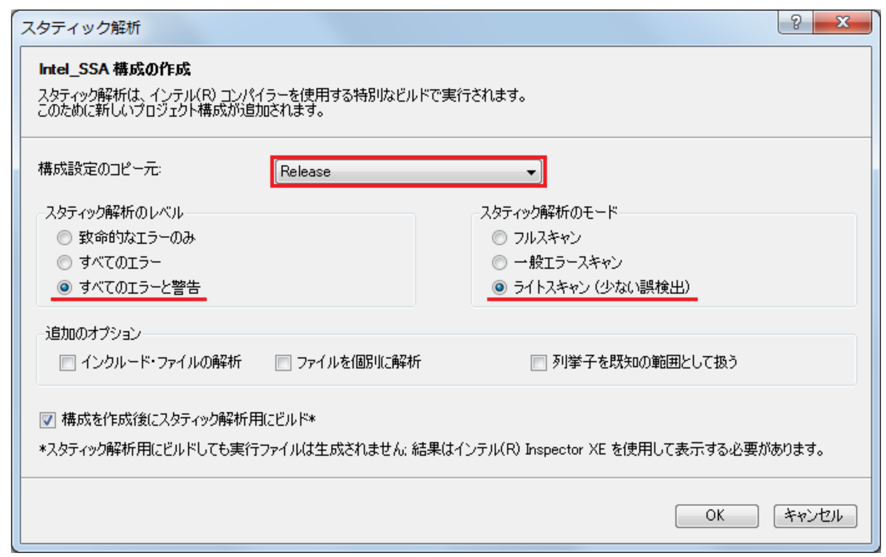
（※本機能は、プロジェクトをインテルコンパイラー用に切り替えなくても利用することが可能です）

それでは、以下に静的解析の手順を示します。

1. まず、プロジェクト“1_nqueens_serial”を選択して、[ビルド] メニューから以下の図のインテル・スタティック解析の項目を選択します。



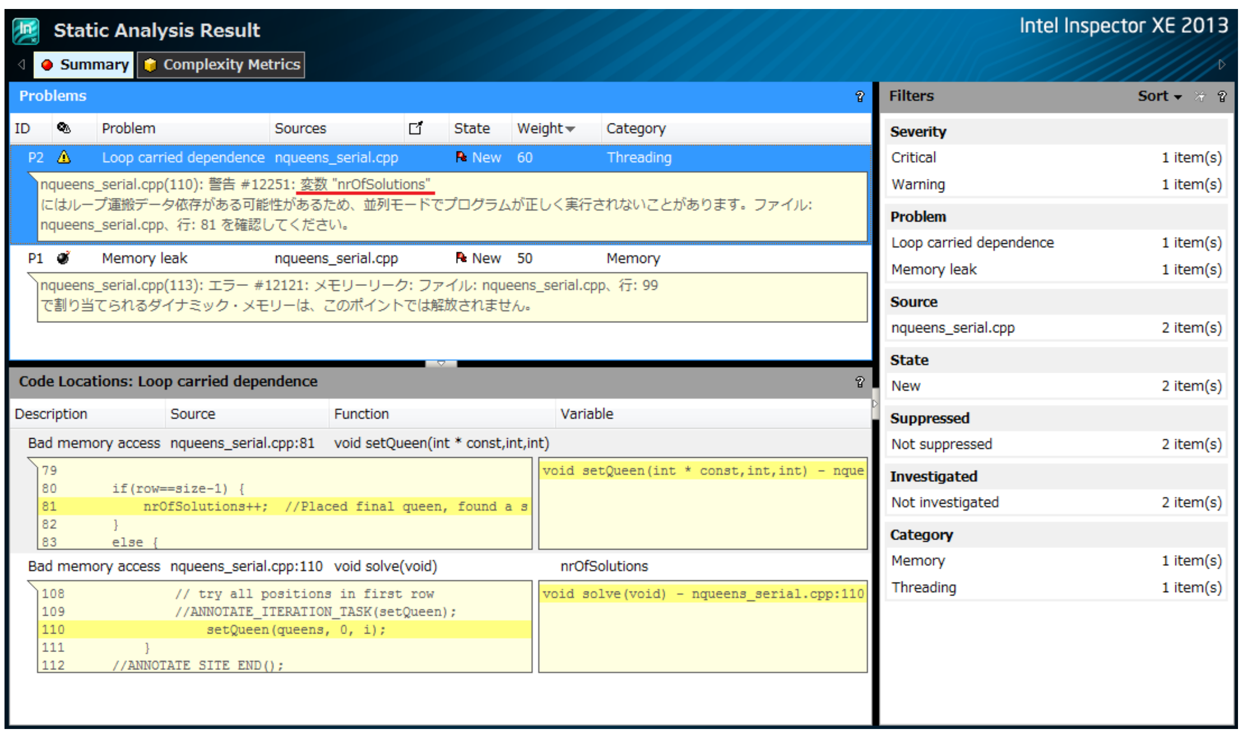
2. 「スタティック解析」ダイアログが表示されます。「構成設定のコピー元」を“Release”にします。



「スタティック解析」ダイアログには、大きく以下の 2 種類の設定が用意されています。

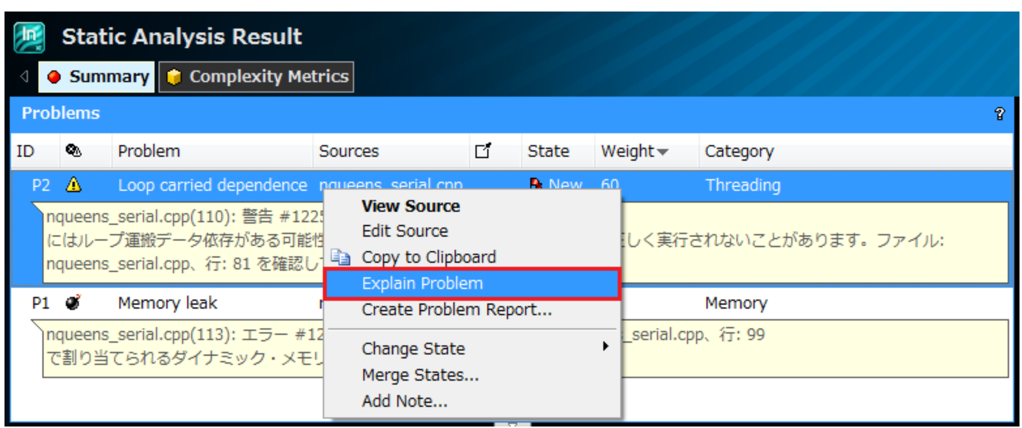
- ◆ 「スタティック解析のレベル」… 検出問題の重要度レベルを設定します。
  - ➢ “致命的なエラーのみ” ー Critical 問題のみ検出（Error、Warning は未検出）
  - ➢ “すべてのエラー” ー Critical および Error 問題の検出（Warning は未検出）
  - ➢ “すべてのエラーと警告” ー Critical、Error および Warning 問題の検出
- ◆ 「スタティック解析のモード」… False Positive／False Negative の調整。つまり誤検出のレベルを設定します。
  - ➢ “フルスキャン” ー 誤検出の可能性は無視し、すべての問題を検出します。
  - ➢ “一般エラースキャン” ー 誤検出の可能性を考慮した検出を実施します。
  - ➢ “ライトスキャン” ー 誤検出の可能性を極力考慮し、確実な問題のみを検出します。

ここでは「スタティック解析のレベル」を“すべてのエラーと警告”、「スタティック解析のモード」を“ライトスキャン”に設定して解析を実施します。準備ができたら [OK] ボタンをクリックして解析を開始します。解析が完了するとインテル® Inspector XE が自動起動されて結果が以下のように表示されます。

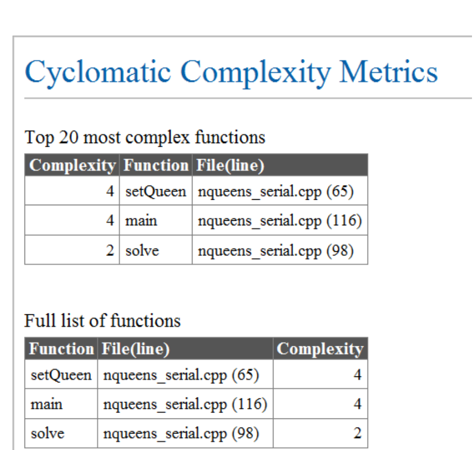


本解析の結果では、Critical が 1 件、Warning が 1 件、計 2 件の問題が検出されています。「Problems」ペインでは、Weight（重要性）が大きい順に表示されます。まず変数“nrOfSolutions”においてループ伝搬依存問題が検出されています。これはマルチスレッド間でのデータの競合を意味しています。またメモリーリーク問題も検出されています。この問題をクリックして選択すると下のペインに問題の箇所がコードレベルで表示され、solve() 関数内で確保したヒープメモリー領域が開放可能な範囲内

で開放されていない状況が示されます。また検出された問題を右クリックして、問題の種類の説明を表示することもできます。



また、上の図で「Complexity Metrics」画面を開くと、Cyclomatic complexity（循環的複雑度）レポートが表示されます。この複雑度数は、分岐命令によるプログラムの経路数でありプログラムの複雑度を現す指標となります。一般的にこの複雑度が上がると問題も発生しやすくなります。このレポートでは各関数の複雑度が表示されます。この情報を基に、複雑度の減少を検討したり、テスト工数の目安を確認することができます。



Cyclomatic Complexity Metrics

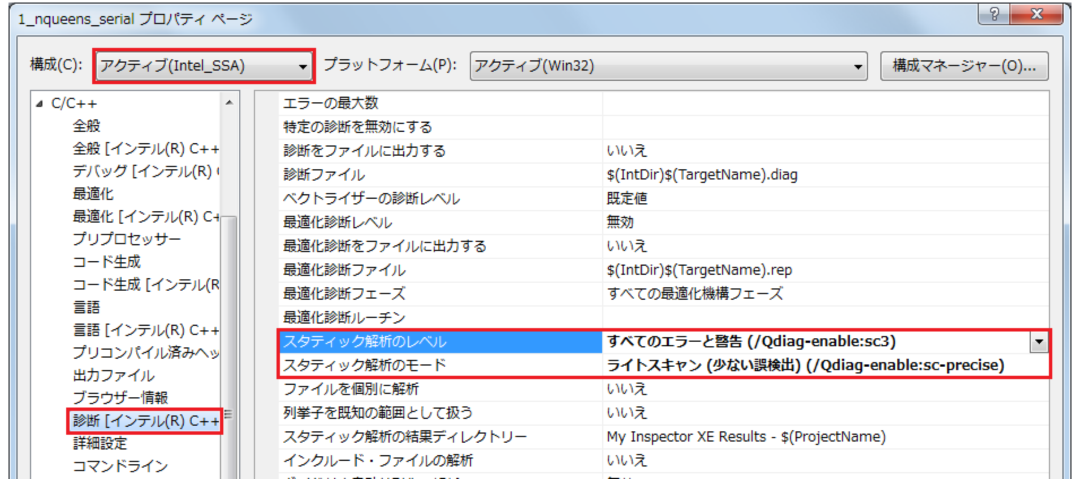
Top 20 most complex functions

Complexity	Function	File(line)
4	setQueen	nqueens_serial.cpp (65)
4	main	nqueens_serial.cpp (116)
2	solve	nqueens_serial.cpp (98)

Full list of functions

Function	File(line)	Complexity
setQueen	nqueens_serial.cpp (65)	4
main	nqueens_serial.cpp (116)	4
solve	nqueens_serial.cpp (98)	2

本節で説明したインテル® コンパイラーによる静的解析の操作を実施すると、「Intel_SSA」という名称で新しい構成が追加されます。静的解析用のコンパイルはこの構成内容で行われます。コンパイルオプションとして以下の図のように「スタティック解析のレベル」と「スタティック解析のモード」が設定されています。この設定を直接変更して静的解析機能を手動で実施することもできます。



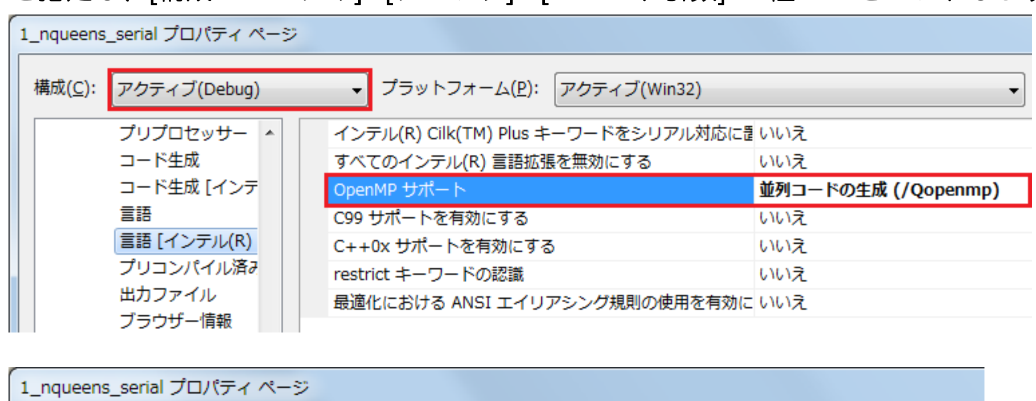
３－７．マルチスレッドの診断（インテル® Inspector XE 使用）

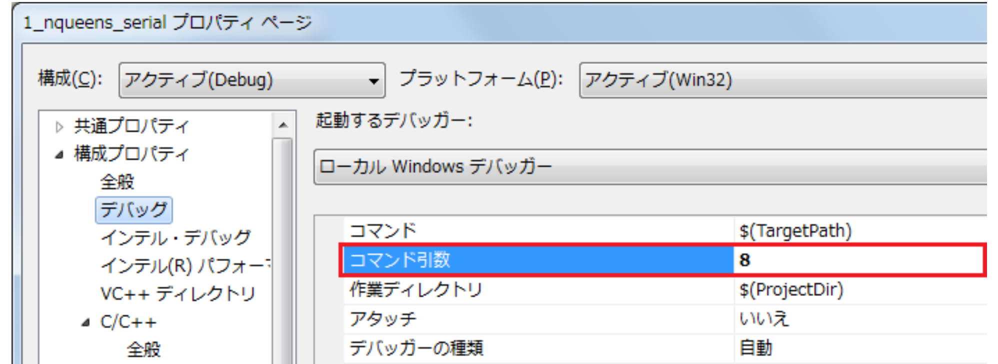
本節ではインテル® Inspector XE を使用して問題の動的解析を実施します。インテル® Inspector XE はメモリー関連問題とスレッド関連問題を検出する機能が提供されています。インテル® Inspector XE で問題検出を行う場合、Debug モードでプログラムをビルドすることが推奨されています。また、インテル® Inspector XE は検出オーバーヘッドがかなりかかりますので実行サイズは比較的小さくすることがこのツールを上手に使うコツとなります。

以下にインテル® Inspector XE による本サンプル・プログラムの問題検出手順を説明します。

１）まず、プロジェクトを Debug モードに切り替えます。

２）プロジェクトのプロパティページで [C/C++] - [インテル(R) C++] - [OpenMP サポート] に /Qopenmp を指定し、[構成プロパティ] - [デバッグ] - [コマンド引数] の値に 8 をセットします。

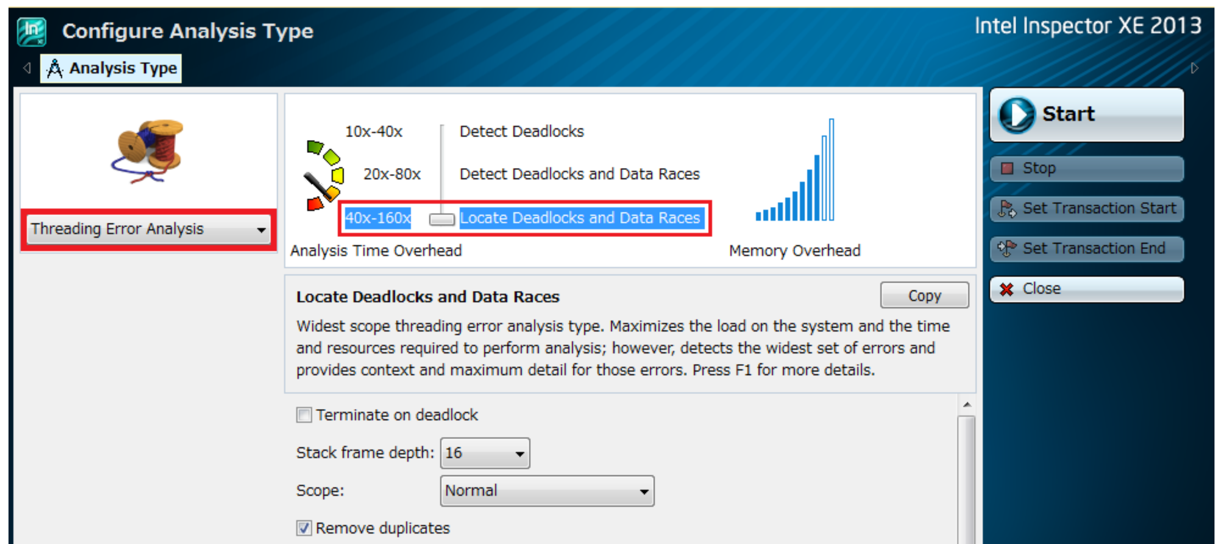




３）プロパティページの設定が完了したらプロジェクトをリビルドし正常終了を確認します。

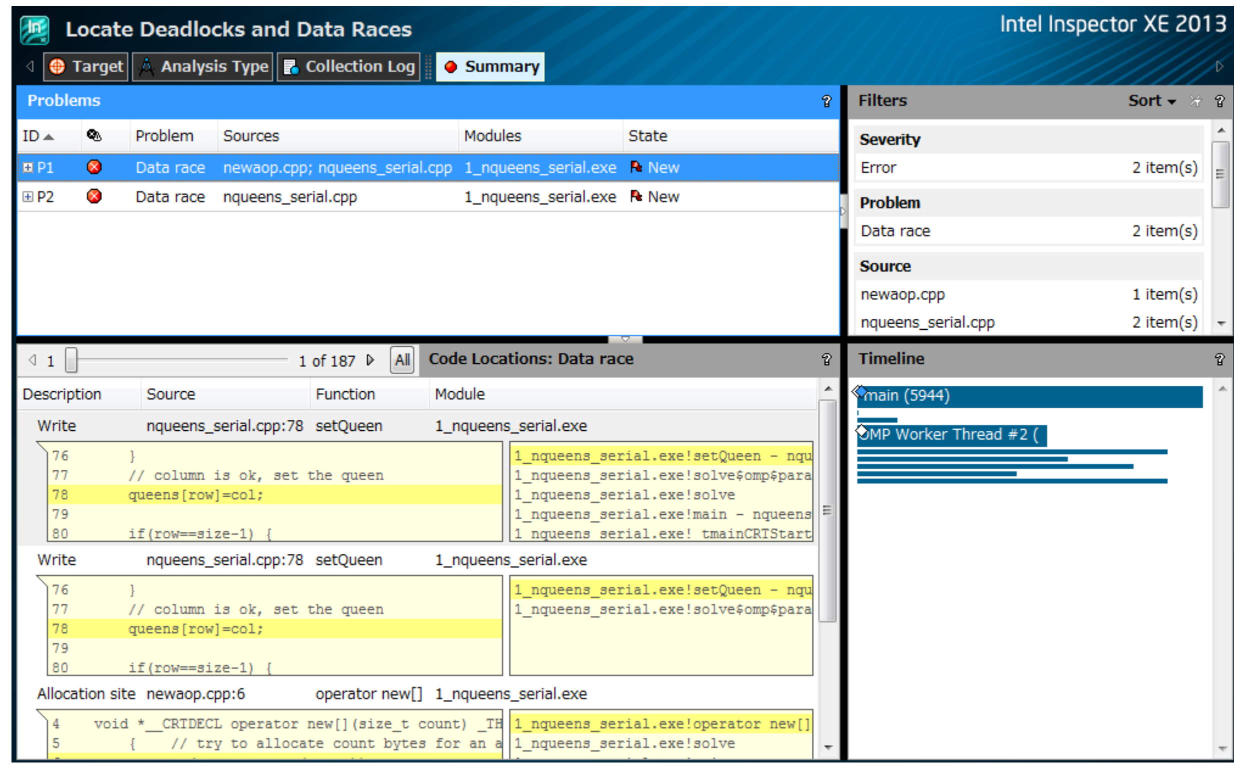
４）インテル® Inspector XE 用ツールバーから New Analysis ボタン ( ) をクリックします。

５）「Configure Analysis Type」画面が表示されるので、次の図のように解析タイプを “Threading Error Analysis” にセットします。また、解析レベルを設定するゲージがあります。解析レベルは 3 種類用意され、一番下側がもっとも高い解析レベルとなります。解析レベルが上がるとより詳細な解析が可能になりますが、解析オーバーヘッドもその分多くかかります。ここでは一番高い解析レベル「Locate Deadlocks and Data Races」に設定します。なお同画面の「Stack frame depth」の値は結果に表示する関数コールスタックの深さを意味し、「Scope」を “Normal” から “Extremely Thorough” に変更すると、スレッドスタック領域におけるデータ競合もチェックするようになります。



6）次に [Start] ボタンをクリックしてサンプル・プログラムのスレッドエラー検出を開始します。

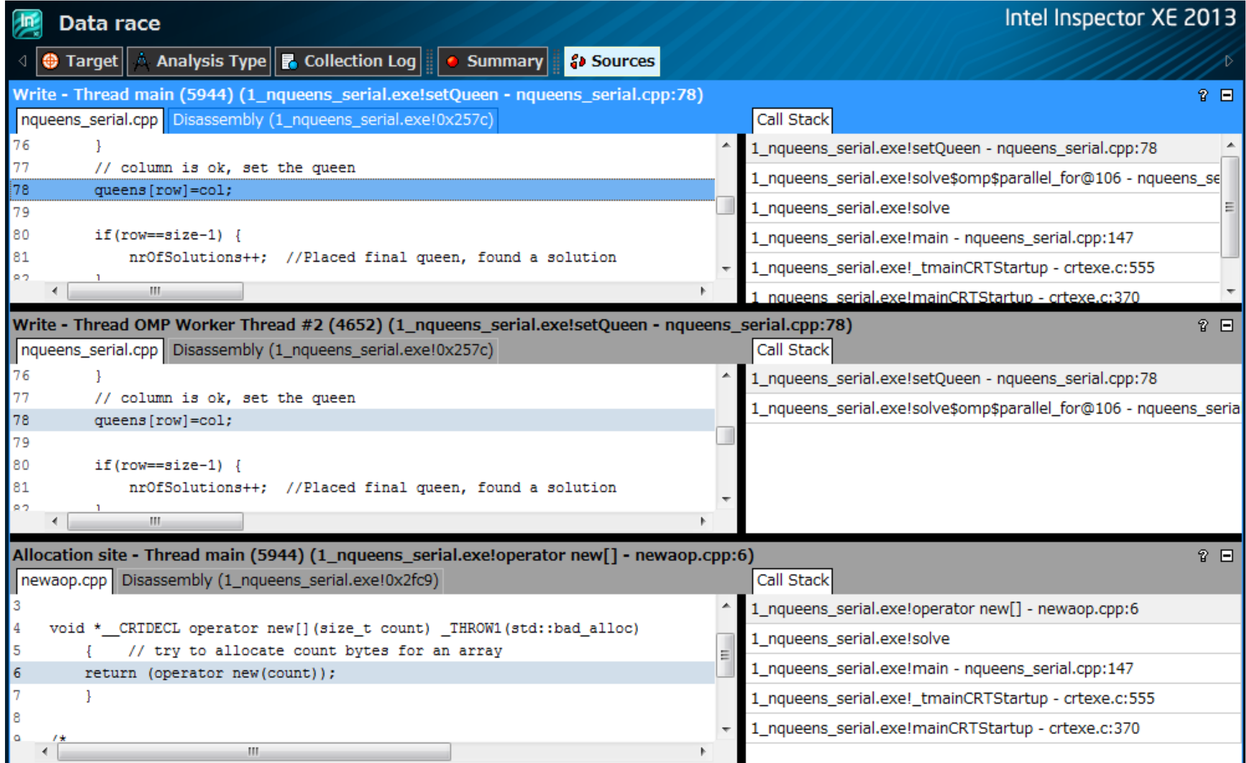
7）検出処理が完了すると下図のような結果画面（Summary 画面）が表示されます。



表示される Summary 画面の「Problems」ペインには、検出された問題一覧が表示されます。ここでは “Data race” 、つまりデータ競合問題が 2 件（P1 と P2）検出されています。またその下のペインでは、「Problems」ペインで選択されている問題の発生場所がソースレベルで表示されます。問題の内容を確認すると、P1 の問題は queens ポインタにおけるデータ競合であり、queens ポインタの同じ領域に複数のスレッドからの“Write”処理が発生していることが分かります。また右下の「Timeline」ペインには問題に関与したスレッド名と発生タイミングが示されています。

同様に P2 の問題を確認すると、nrOfSolutions 変数に対するデータ競合であることが分かります。

表示されている問題をダブルクリックすると、さらに詳細な情報をソースレベルで表示します。

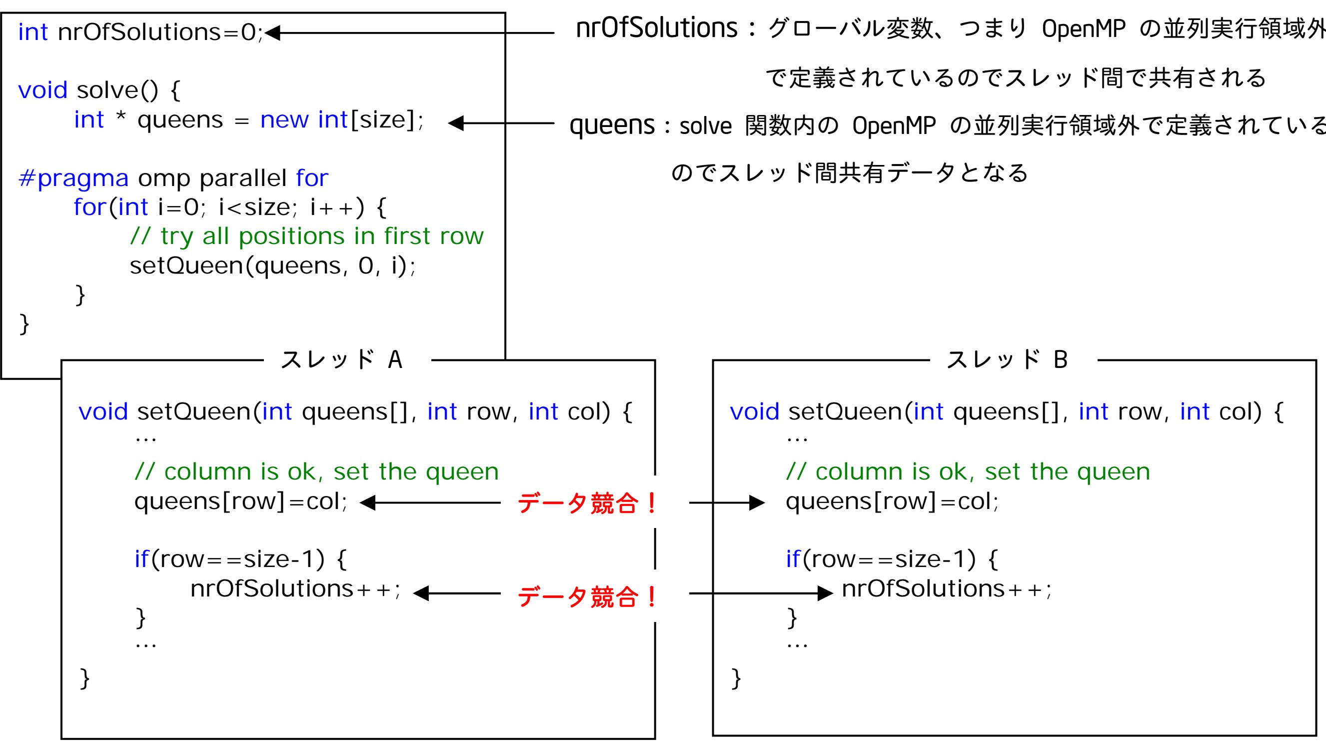


このように、インテル® Inspector XE を使用して動的に解析することにより、queens ポインタと nrOfSolutions 変数に対するデータ競合問題が明らかになりました。

次節ではこれらの問題の修正作業に入ります。

３－８．マルチスレッドコードの修正（インテル® Composer XE 使用）

本サンプル・プログラムでは、queens ポインタおよび nrOfSolutions 変数においてスレッド間でデータの競合が発生していることがわかりましたので、これらのデータに関するソースコードの修正を行う必要があります。修正に当たって本サンプル・プログラムでの OpenMP におけるデータの有効範囲を考える必要があります。OpenMP では、基本的に並列実行領域外で定義されたデータは、スレッド間で共有データとして扱われます。つまり以下に示すように、queens ポインタ、および nrOfSolutions 変数は共有データとなり、スレッド間で同じタイミングでの書き込み処理が発生する可能性があるデータとなります。



では問題の対処法ですが、まず queens ポインタは setQueen() 関数内で使用され、安全に Queen の駒を配置できるマス目（行、列）を格納するメモリー領域として使用されています。ここで重要な点は、このポインタ領域は最終的な答えを求めるための一時的な作業領域として使用されていることです。よってこのポインタ領域をマルチスレッドで使用する場合は、solve() 関数内でコールされる setQueen() 関数単位で独立したものを用意するか、または少なくとも生成される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領域）にすればデータの競合は発生しません。ここでは setQueen() 関数単位で独立した queens ポインタ領域として定義します。つまり solve() 関数内の for ループ文の内側で定義し、イテレーション毎に独立した queens ポインタ領域を生成して setQueen 関数の入力引数として渡します。

それから nrOfSolutions 変数に関しては、この変数も setQueen() 関数で使用され、安全に Queen の駒を配置できるパターン（つまり答え）を格納する変数として使用されています。ただしこの変数は最終的な答えを格

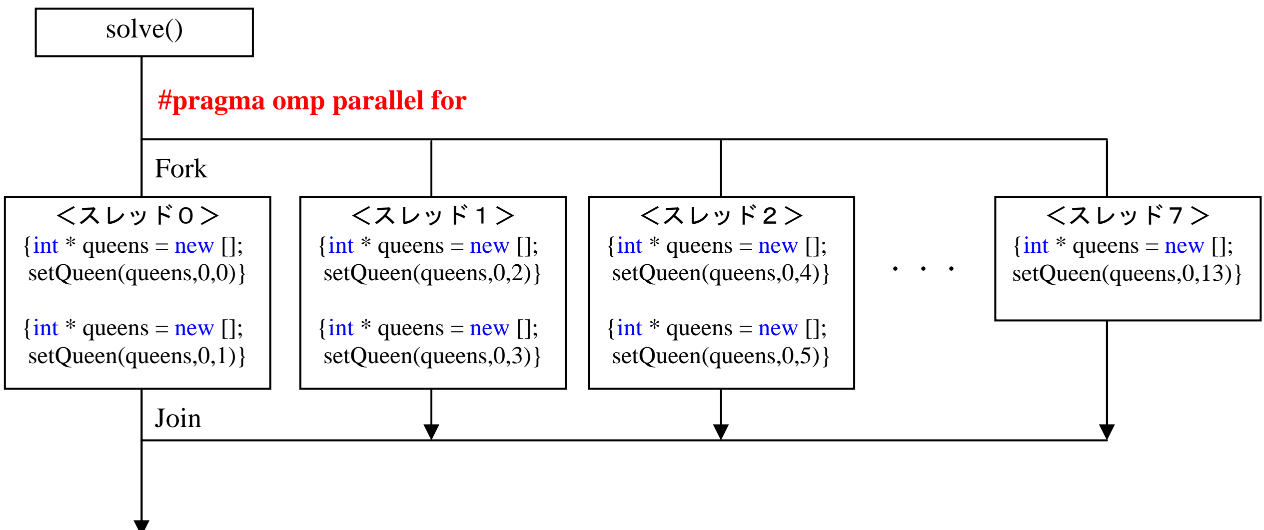
納するため計算しながら値をインクリメントしていく必要があります。このため各スレッド間で共有すべき変数となります。しかし複数のスレッドによって同時に書き込み処理が行われるとデータの値が正しく反映されなくなります。このデータの不整合を回避するため“nrOfSolutions++”の処理に対してスレッド間の同期処理を施す必要があります。この同期処理を行った場合は、常に単一のスレッドのみが対象の処理を実行できるようになり、他のスレッドは現在実行中のスレッドがこの処理を終了するまで待ち状態となります。OpenMP が提供する同期処理構文にはいくつかありますが、ここでは一般的な critical 宣言子を適用します。なお、通常このように排他制御が必要な領域をクリティカル・セッションといい、本サンプル・プログラムでは、“nrOfSolutions++”をクリティカル・セッションとして扱います。

上記までの修正内容をサンプル・プログラムに反映したものを以下に記します。

```
int nrOfSolutions=0;

void solve() {
      //int * queens = new int[size];          ← コメントアウト

#pragma omp parallel for
      for(int i=0; i<size; i++) {
            int * queens = new int[size];      ← ループ文の内側で定義
            // try all positions in first row
            setQueen(queens, 0, i);
      }
}
```

```
void setQueen(int queens[], int row, int col) {

      …

      // column is ok, set the queen
      queens[row]=col;

      if(row==size-1) {          ← 共有データ nrOfSolutions へのアクセス競合を防ぐため
#pragma omp critical                 スレッド間の同期処理を追加する
            nrOfSolutions++;   //N-Queens 問題の解答を格納
      }
      else {
            // try to fill next row
            for(int i=0; i<size; i++) {
                  setQueen(queens, row+1, i);
            }
      }
}
```

次の図は、本修正を施したコードの並列実行ロジックのイメージを示しています。solve() 関数でコールされるそれぞれの setQueen() 関数には、個別の queens ポインタ領域が渡されます。これによってスレッド間でのデータの競合はなくなります。また setQueen() 関数内のクリティカル・セッションはスレッド間で同期が取られ、同時実行されることはありません。



サンプル・プログラムの修正が完了したら、プロジェクトを“Debug”構成でリビルドして、再度インテル® Inspector XE のスレッドチェックを試してみてください。今度はエラーが表示されないはずです。

次に、プロジェクトを“Release”構成に変更してリビルドを行い、並列化の効果を確認してみましょう。ビルドの際はプロジェクトのプロパティページで以下の項目を確認してください。

- [構成プロパティ] - [C/C++] - [Language] - [OpenMP Support] → /Qopenmp

ビルドが完了したら、「デバッグなしで開始」を実行してください。結果が以下のように表示されます。

```
C:¥Windows¥system32¥cmd.exe
Usage: 1_nqueens_serial.exe boardSize [default is 14].
Starting nqueens solver for size 14...
Number of solutions: 365596
Correct result!

Calculations took 1667ms.
続行するには何かキーを押してください . . .
```

並列化する前の以下のシリアルコードの結果と比較すると、ここでは３倍強のパフォーマンス向上があることが分かります。

```
C:¥Windows¥system32¥cmd.exe
Usage: 1_nqueens_serial.exe boardSize [default is 14].
Starting nqueens solver for size 14...
Number of solutions: 365596
Correct result!

Calculations took 5400ms.
続行するには何かキーを押してください . . .
```

では次にインテル® VTune AmplifierXE を使用して、この並列化したサンプル・プログラムの並列性をチェックしてチューニングに挑戦してみましょう。

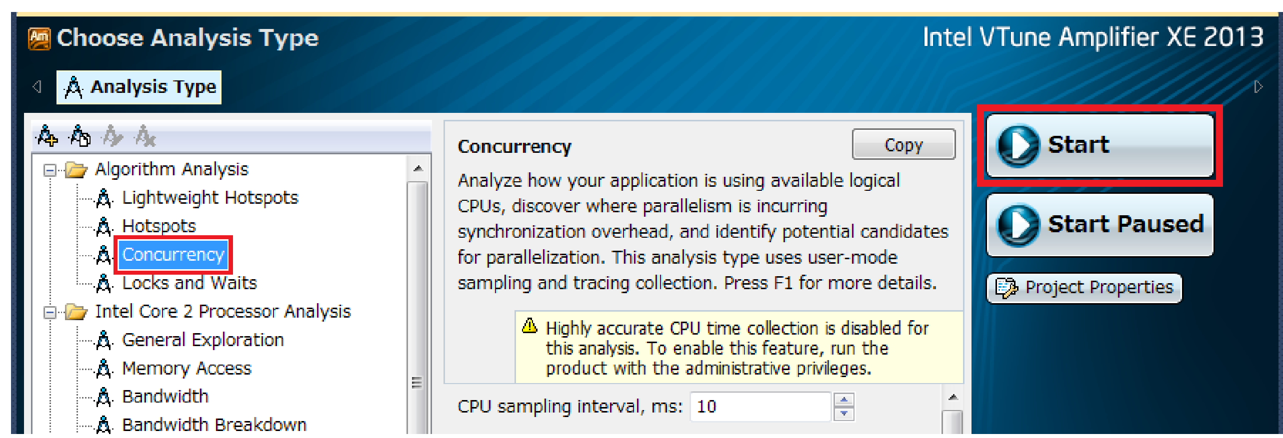
３－９．マルチスレッドのチューニング（インテル® VTune Amplifier XE 使用）

ここでは、並列化したサンプル・プログラムに対してさらにチューニングを行ってパフォーマンスの改善に望みます。チューニングを行う前に現在の並列化プログラムの並列性をチェックします。この並列性を調べることにより対象のアプリケーションが、システムに搭載される CPU リソースをどれだけ効果的に利用できているかを知ることができます。一般的にこの並列性の数値が高いほどパフォーマンス性能がよいと言えます。本チューニングでは、この並列性を高めることを目的とします。

それでは、現在のサンプル・プログラムの並列性の測定方法を説明しますが、前節までのサンプル・プログラムの修正と“Release”構成でのビルドが完了していることを事前に確認してください。

並列性の測定にはインテル® VTune Amplifier XE を使用します。

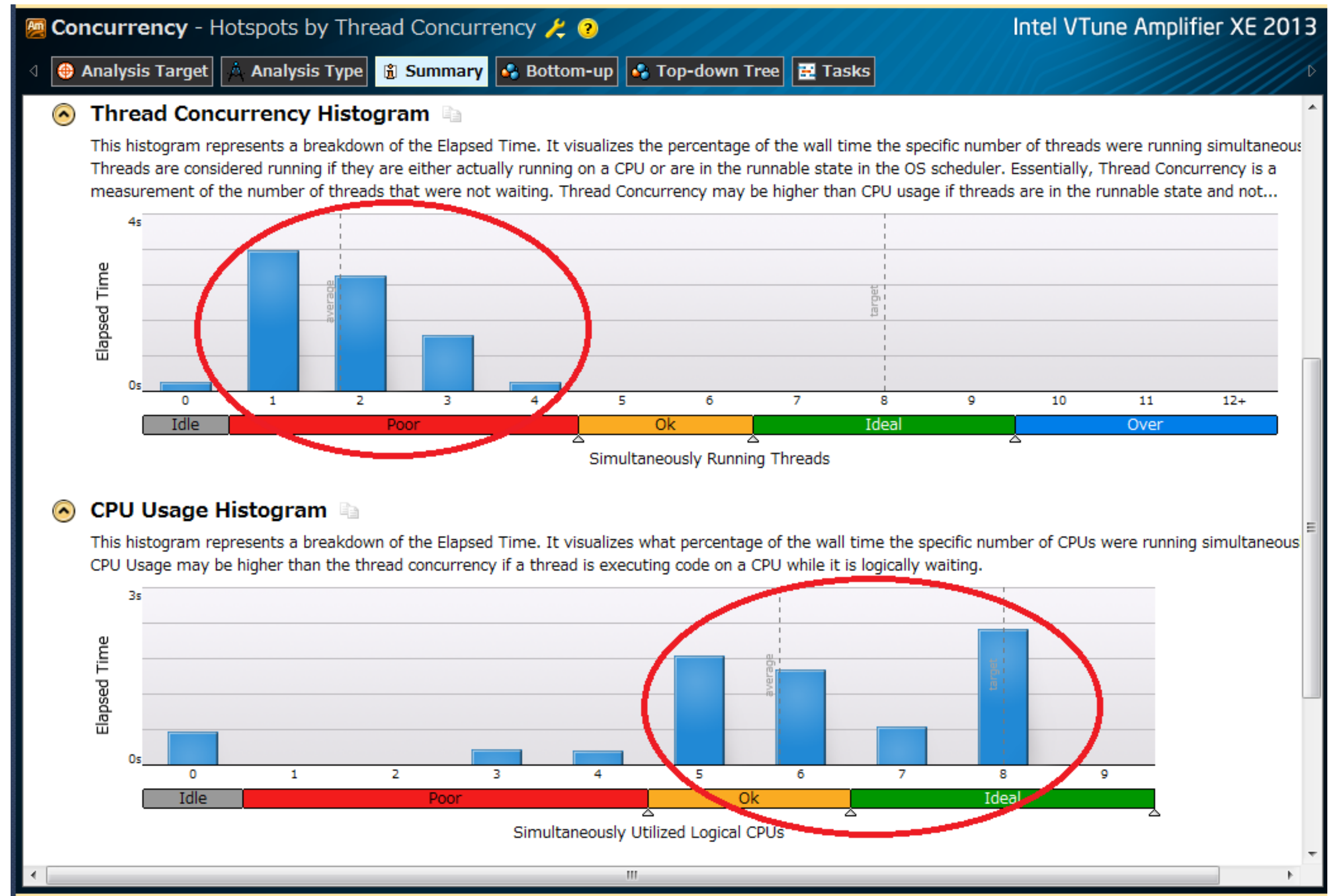
まず、インテル® VTune Amplifier XE 用ツールバー（）から [New Analysis] をクリックして [Analysis Type] 画面を表示します。[Analysis Type] 画面の [Concurrency] を選択して [Start] ボタンをクリックしてプロファイルの実行を開始します。



サンプル・プログラムが自動で実行されて並列性の測定が開始され、しばらくすると次のような [Summary] 画面が表示されます。この画面の中ほどに２つの Histogram が表示されています。それぞれのヒストグラムの意味合いは以下のようになります。

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。



本サンプル・プログラムの場合は、スレッドの同時実行数は低く、CPU コアの同時使用率は高いという結果となっています。これは一体どういうことでしょうか。それでは [Bottom-up] 画面を開いてみましょう。

[Bottom-up] 画面では、大きく 3 つのペインに分かれており、まず左上には Hotspot 関数や消費 CPU 時間などが表示されたメインペインがあり、右上には関数のコールスタック情報などを表示するペインがあり、下側にはスレッド単位の時系列実行状態を示すタイムラインビューがあります。

メインペインには、“setQueen” 関数が Hotspot 関数としてリストされており CPU 時間のほとんどを消費していることが分かります。また赤色で示されていることからこの関数の同時実行分布は “Poor” であることも分かります。それからタイムラインビューを見るとほとんど黄色い縦線で埋め尽くされています。このペインの右側にはタイムラインビューで使用されるマークの意味と表示有無のチェックボックスがあります。黄色い縦線は “Transitions” を意味しており、つまりスレッド間の同期処理の遷移状態を示しています。よって本サンプル・プログラムの実行では同期処理が多数発生していることになります。この “Transition” のチェックを外して同期処理の表示を非表示にすると “CPU Time” の状態が見えてきます。多くのスレッドが CPU コアを使用している様子が分かります。タイムラインビューの左下に “CPU Usage” と “Thread Concurrency” というエントリーがあり時系列に CPU の使用率とスレッドの並列実行数の状態が表示されています。それぞれの表示内容の上をマウスポインターでなぞってみるとその刻々の数値が表示されます。[Summary] 画面でも確認したように CPU 使用率は高いがスレッド同時実行数は低いことが分かります。

Concurrency - Hotspots by Thread Concurrency　　Intel VTune Amplifier XE 2013

Analysis Target | Analysis Type | Summary | Bottom-up | Top-down Tree | Tasks

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Over... Time	Wait Time by Utilization (Idle / Poor / Ok / Ideal / Over)	
setQueen	43.585s	2.149s	31.800s	1_r
RtlIntegerToUnicodeSt	0.132s	0s		ntd
__kmp_end_split_barri	0.126s	0s		libic
NtDelayExecution	0.101s	0s		ntd
solve	0.082s	0s	2.288s	1_r
__kmp_x86_pause	0.050s	0s		libic
IsNLSDefinedString	0.020s	0s		KEF
__kmp_execute_tasks	0.020s	0s		libic
[Import thunk _kmpc_	0.020s	0s		1_r
Selected 1 row(s):	43.585s	2.149s	31.800s	

CPU Function/CPU Stack - CPU Time

Viewing 1 of 18 selected stack(s)

71.5% (31.169s of 43.585s)

1_nqueens_serial.exe!setQueen - n...

Thread: mainCRTStartup (0x, __kmp_launch_moni, OMP Worker Thread, OMP Worker Thread, OMP Worker Thread

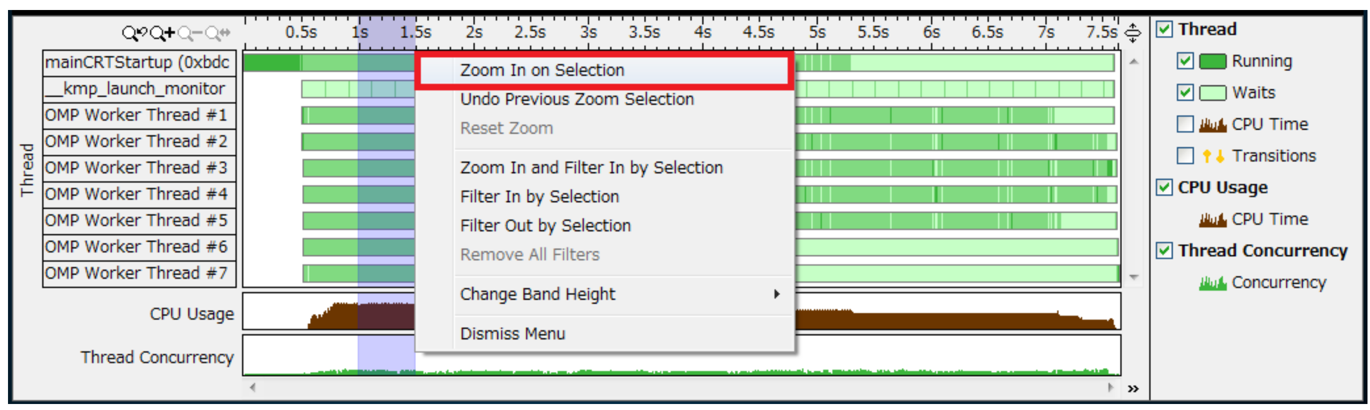
CPU Usage / Thread Concurrency

Thread: Running, Waits, CPU Time, Transitions / CPU Usage: CPU Time / Thread Concurre...: Concurrency

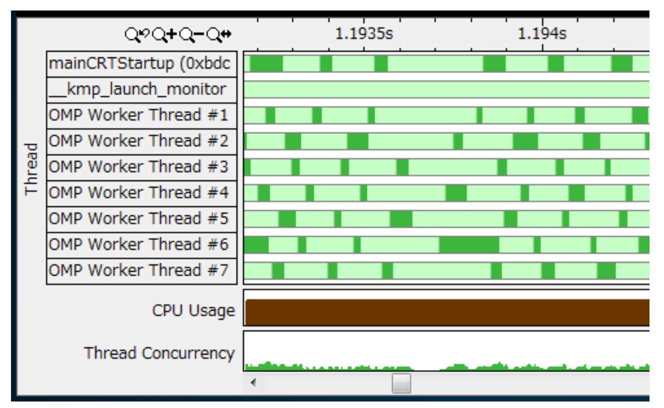
No filters are applied. | Any Process | Any Thread | Any Module | Utilization: Any Utilization

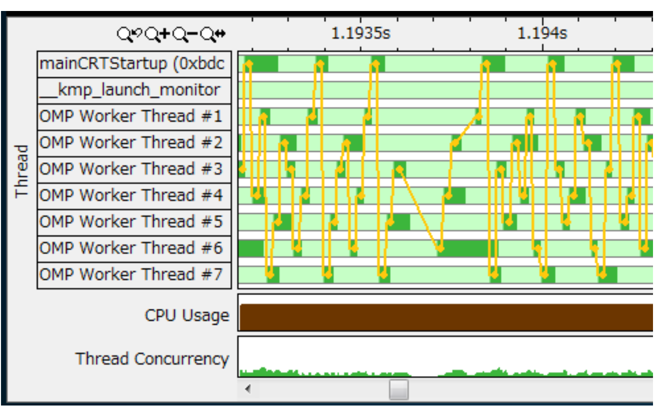
Call Stack Mode: Only user functions | Inline Mode: on

では今度は“CPU Time”のチェックも外してください。そしてタイムラインビュー上のある適当な時間帯をマウスでドラッグし表示されるコンテキストメニューから [Zoom In on Selection] を選択します（または、左上のボタン からも操作できます）。この動作を繰り返して時間の目盛が 1ms 程度のスケールになるまで繰り返します。するとスレッドの“Running”状態と“Waits”状態の詳細が見えてきます。このサンプル・プログラムのスレッドは実行状態より待機状態のほうが多いことが分かります。

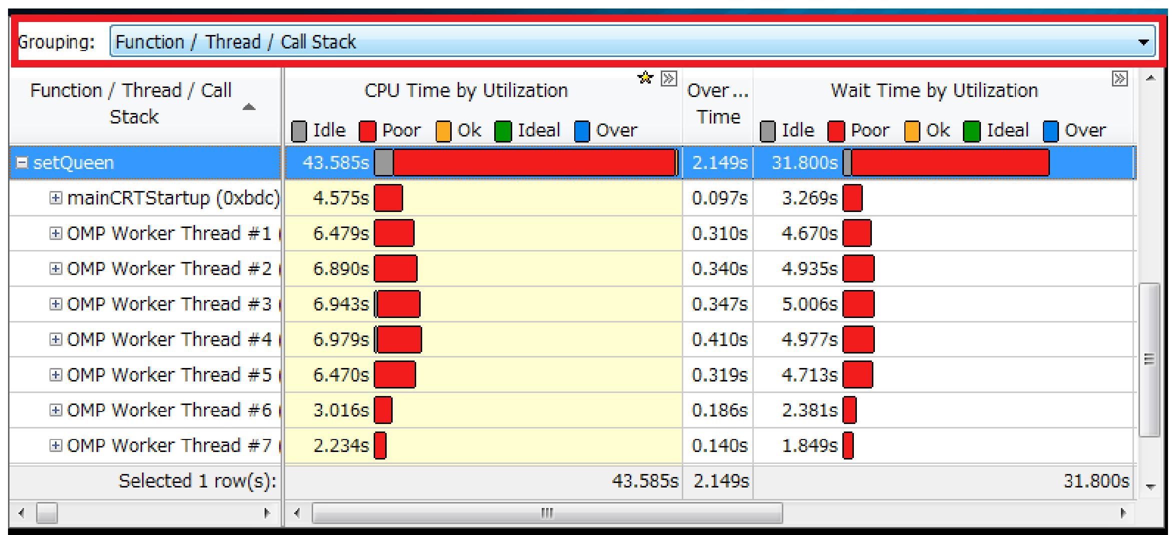


このタイムラインビューに“Transitions”のチェックを ON にするとスレッド間の同期の遷移状態が表示されそれぞれのスレッドが待機状態から同期処理を介して実行状態に遷移している様子を見ることができます。



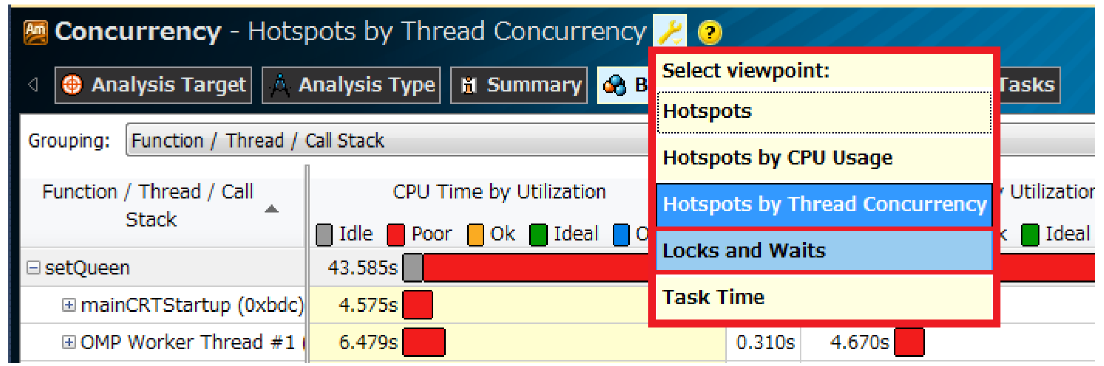


また、それぞれのスレッドの待機状態の時間も表示することができます。メインペイン上の [Grouping] を "Function / Thread / Call Stack" に切り替えて "setQueen" 関数の左に表示されている "+" 記号をクリックすると、'setQueen' 関数を実行したスレッド一覧が表示されスレッド単位の CPU 時間と待機時間が表示されます。本サンプルでは CPU 時間に対する待機時間の割合が非常に高いことが分かります。(またこのビューからスレッド単位のワークロードも観察できロードバランスの調査も可能となります)


Grouping: Function / Thread / Call Stack

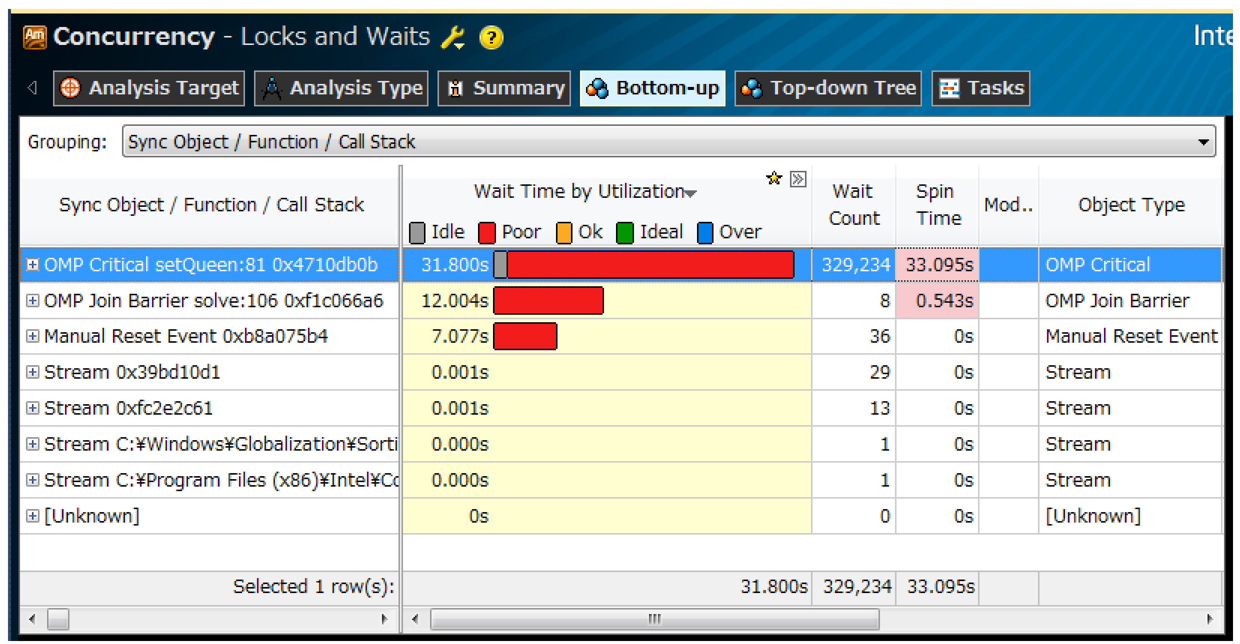
Function / Thread / Call Stack	CPU Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Over... Time	Wait Time by Utilization (Idle / Poor / Ok / Ideal / Over)
setQueen	43.585s	2.149s	31.800s
mainCRTStartup (0xbdc)	4.575s	0.097s	3.269s
OMP Worker Thread #1	6.479s	0.310s	4.670s
OMP Worker Thread #2	6.890s	0.340s	4.935s
OMP Worker Thread #3	6.943s	0.347s	5.006s
OMP Worker Thread #4	6.979s	0.410s	4.977s
OMP Worker Thread #5	6.470s	0.319s	4.713s
OMP Worker Thread #6	3.016s	0.186s	2.381s
OMP Worker Thread #7	2.234s	0.140s	1.849s
Selected 1 row(s):	43.585s	2.149s	31.800s

それでは今度は、ビューポイントを変えてみましょう。 ボタンをクリックして [Locks and Waits] を選択します。ビューポイントを切り替えることで表示する項目内容が変更され、別な視点からパフォーマンス問題を検証することができるようになります。

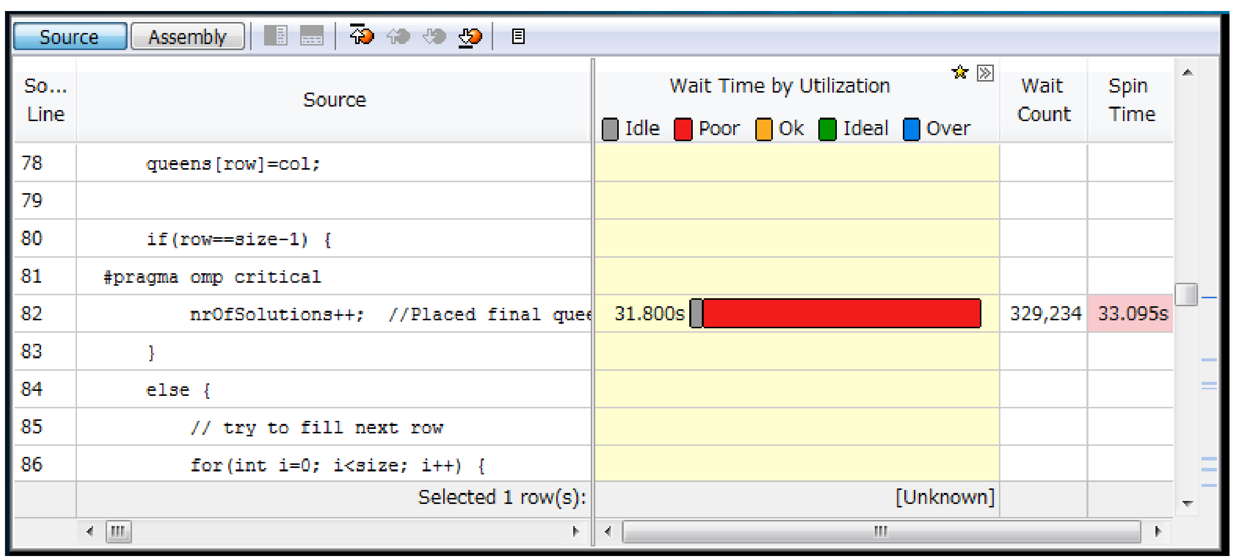


切り替えが完了したら、[Bottom-up] 画面を開きます。Grouping が "Sync Object / Function / Call Stack" であ

ることを確認して内容を見てみます。Sync Object リストの一番上に“OMP Critical setQueen:#”があり、その Object の待機時間、待機回数、スピンタイムなどの情報が表示されています。この行をダブルクリックするとソースコードにドリルダウンすることができます。



ソースコードを見るとクリティカル･セッションとして定義した処理に待機時間が属していることが分かります。つまりこの処理を実行する際にスレッドが待たされるケースが多く発生しており、この問題が並列性を低下させる原因であると考えられます。



このスレッドの待機状態をなくすためにはこの変数をスレッド間で共有するのではなく独立した変数としてスレッド単位で用意する必要があります。そうすれば各スレッドは他のスレッドに影響されることなく完全に独立した状態で（非同期に）処理を実行することができるようになります。それぞれのスレッドでの仕事が完了した後で、各スレッドの合計値を求めることで最終的な解答を得ることができます。では、このロジックをソースコードに反映してみましょう。本チューニングに当たって、OpenMP 規格で定義される、各スレッドの

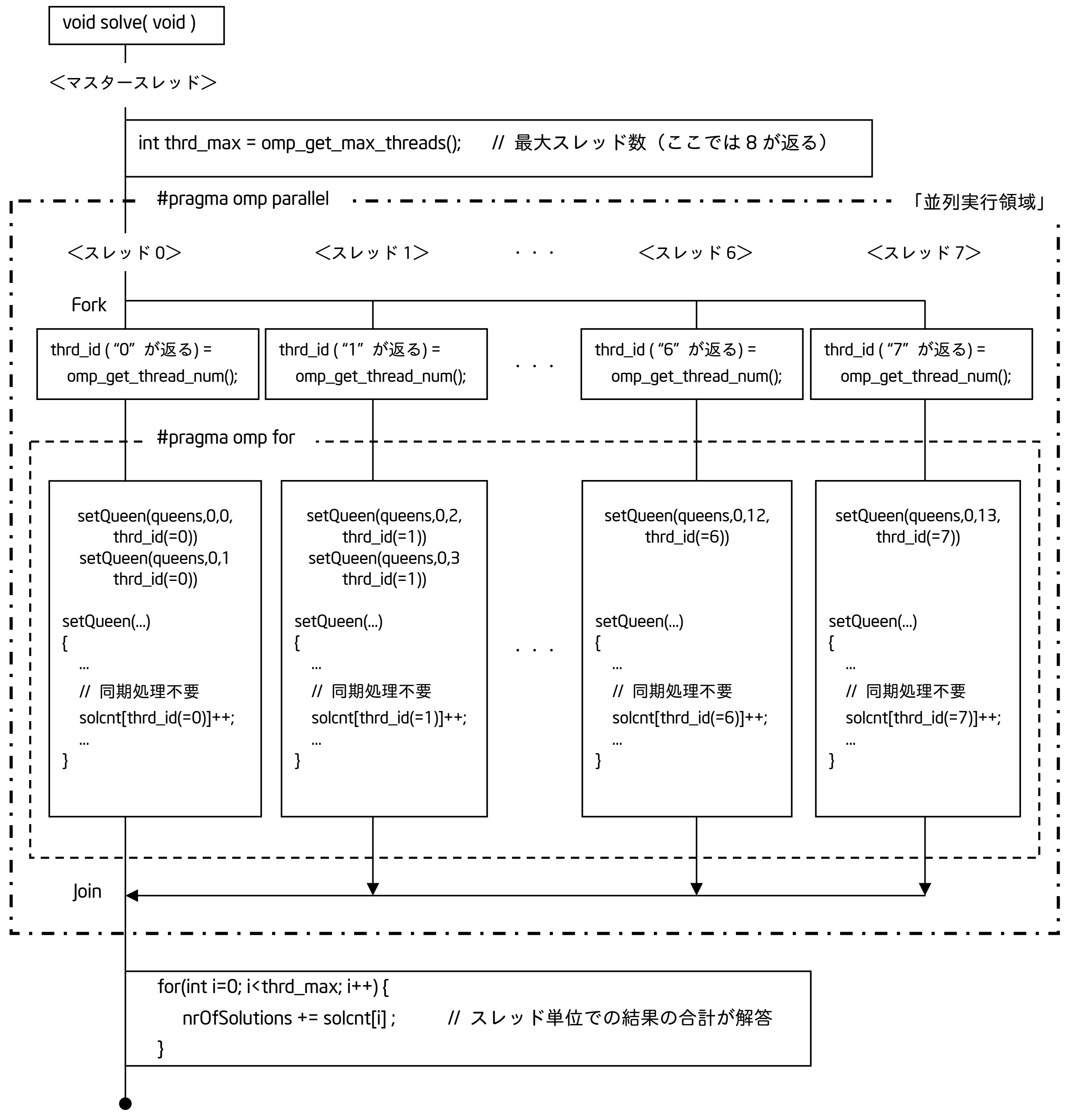
ID を取得する関数とスレッド数を取得する関数を利用します。またスレッド単位で結果を格納する方法には幾つかありますが、ここでは解答変数を新たに定義します。修正が必要な関数は、solve() 関数と setQueen() 関数です。以下に修正内容を記します。

```
#include <iostream>
#include <windows.h>
#include <mmsystem.h>

#include "omp.h"        // OpenMP 関数を使用するためのヘッダー

using namespace std;

int solcnt[32];         // スレッド単位の解答を格納する配列（最大32スレッド）
```

```
void solve() {

    int thrd_max = omp_get_max_threads();      // 利用可能なスレッド数の最大値の取得

#pragma omp parallel
    {
        int thrd_id = omp_get_thread_num();    // 本関数を実行するスレッドIDの取得

#pragma omp for
        for(int i=0; i<size; i++) {
            int * queens = new int[size];
            // try all positions in first row
            setQueen(queens, 0, i, thrd_id);   // スレッドID を引数に追加
        }

    }   // pragma omp parallel（Join）

    for(int i=0; i<thrd_max; i++) {
        nrOfSolutions += solcnt[i] ;           // スレッド単位での結果の合計が解答
    }

}
```

```
void setQueen(int queens[], int row, int col, int thrd_id) {   // スレッドIDを引数に追加

    ...

    // column is ok, set the queen
    queens[row]=col;

    if(row==size-1) {
        solcnt[thrd_id]++;      // スレッド単位に独立した変数（同期処理不要）
    }
    else {
        // try to fill next row
        for(int i=0; i<size; i++) {
            setQueen(queens, row+1, i, thrd_id);     // スレッドIDを引数に追加
        }
    }
}
```

以下は、本チューニング・ロジックのイメージ図です。



修正が完了したら、"Release" 構成のままでプロジェクトをリビルドします。

ビルドが完了したら、再度インテル® VTune Amplifier XE を使用して待機状態と並列性を測定してみてください。

本チューニングで、setQueen() 関数における待機状態は解消され、並列性も向上しているはずです。

３－１０．更なるチューニング（インテル® VTune Amplifier XE 使用）

ここでは更なるチューニングに挑戦します。

前節までは、マルチスレッドによる並列化を実装しマルチコアをフル活用することでパフォーマンスの向上を図りました。その結果、CPU 使用率も上がり並列性も向上しました。しかし、これはアプリケーション・レベルでは一見そのように見えますが、CPU のアーキテクチャー・レベルで見た場合、つまり CPU 内部で処理される命令レベルで効率性を考えた場合、本当に効率がよい動作となっているかは不明です。インテル® VTune Amplifier XE には、「イベント・ベース・サンプリング（EBS)」と呼ばれるデータ収集方法があり、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）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。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ます。以下に、イベントの例を挙げます。

- 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

このレベルのチューニングを行うには、インテル CPU アーキテクチャーの知識がある程度必要になります。本節ではインテル® VTune Amplifier XE の EBS を使用してマルチスレッド特有の問題となりうるフォルス･シェアリングを考えます。フォルス・シェアリングは同一の CPU キャッシュラインに対して複数コアからアクセスすることによって物理メモリー転送が発生する問題です。この問題に関する注意点を以下にまとめます。

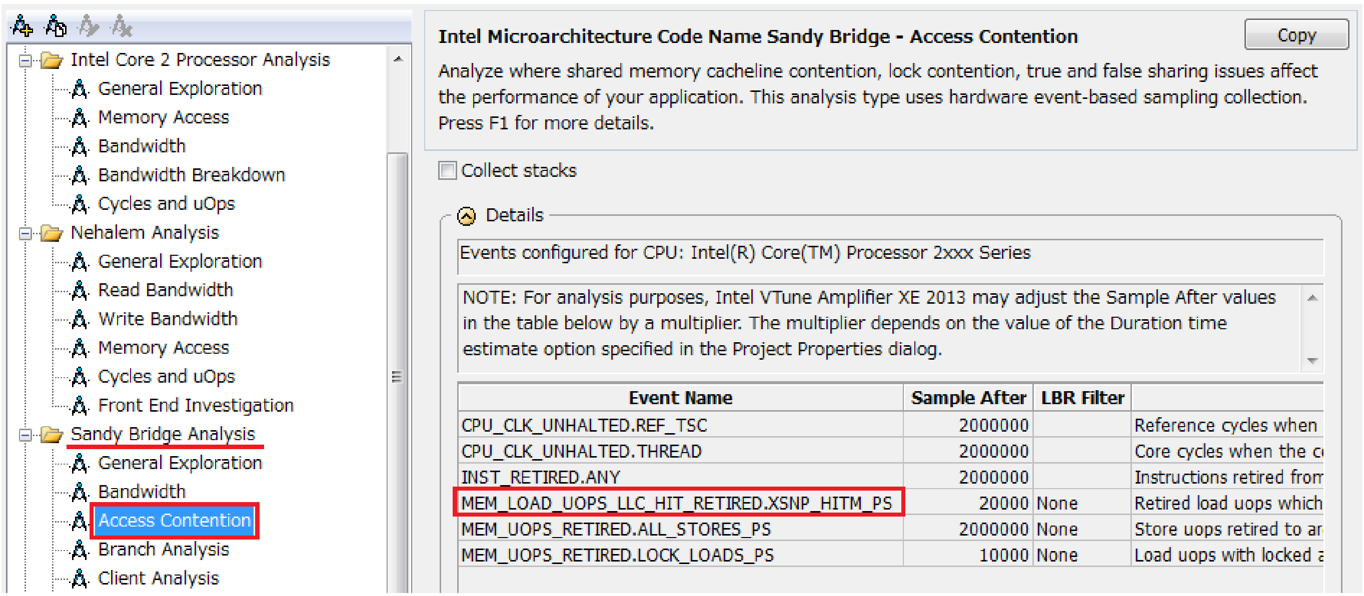
- 一般的に、シングルスレッドでは発生しない問題
- CPU キャッシュに比べ物理メモリー転送は非常に遅い
- 通常、インテル CPU のキャッシュラインは 64 バイトおよび 64 バイト境界
- キャッシュ転送はキャッシュライン単位で行われる
- 複数コアからの同一キャッシュラインへのアクセスは、読み込みだけなら問題ない
- あるコアで書き込みを行い、他のコアでアクセス（読み書き）した時、物理メモリー転送が発生する

（※ フォルス・シェアリングの詳細は、他の参考資料などを適宜ご参照ください。）

それでは、フォルス・シェアリング問題について検証を行い、更なるチューニングを考えてみます。

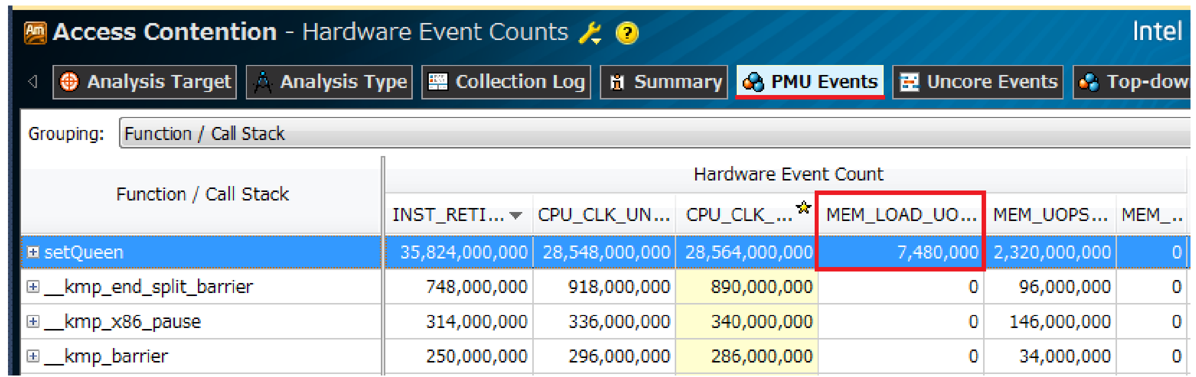
本章で使用している CPU は Intel(R) Core(TM) i7-2600 でありコード名が“Sandy Bridge”と言われるプロセッサーです。ここでは Sandy Bridge に対するチューニングとなります。

まず [Analysis Type] 画面から [Sandy Bridge Analysis] – [Access Contention] を選択します。この分析タイプには収集するイベントが 6 個選択されておりその中で“MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS”がフォルス・シェアリングを検出するイベントとなります。



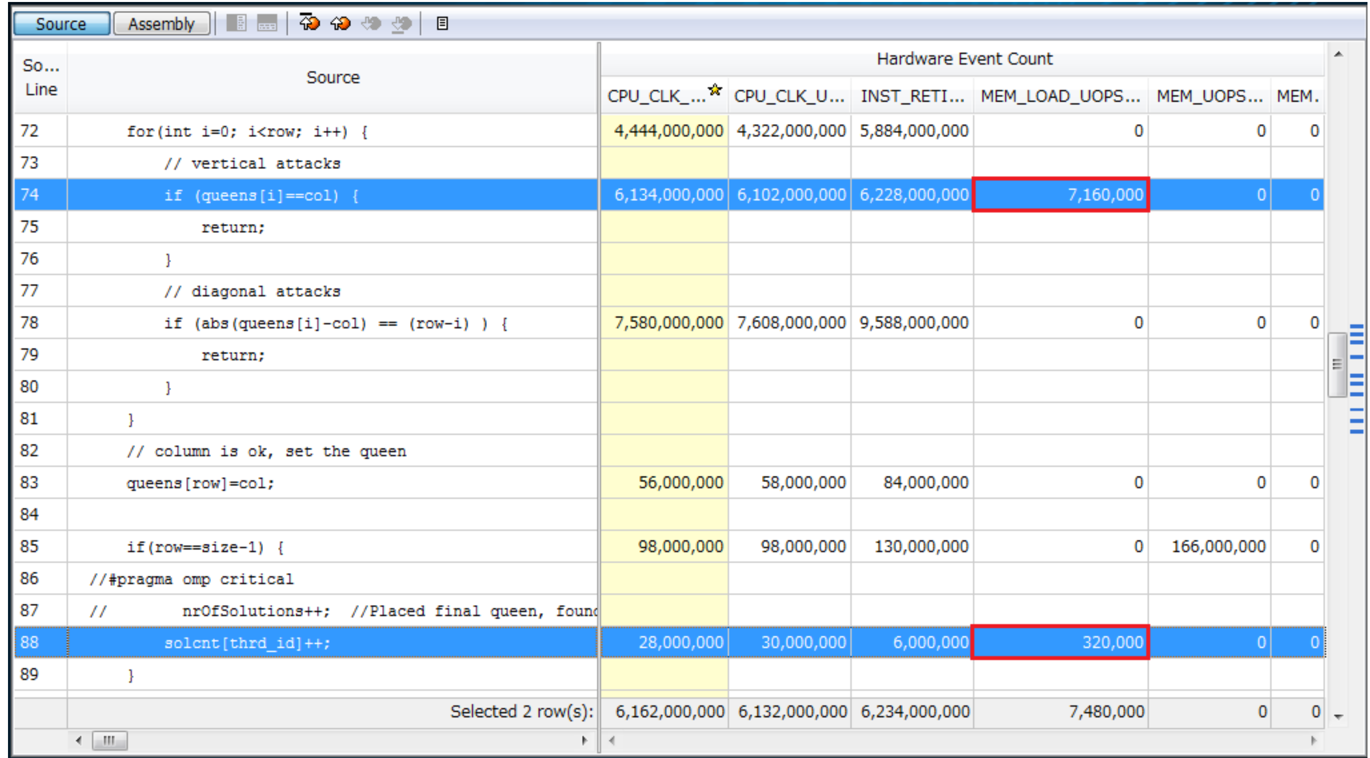
Event Name	Sample After	LBR Filter	
CPU_CLK_UNHALTED.REF_TSC	2000000		Reference cycles when
CPU_CLK_UNHALTED.THREAD	2000000		Core cycles when the c
INST_RETIRED.ANY	2000000		Instructions retired from
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS	20000	None	Retired load uops which
MEM_UOPS_RETIRED.ALL_STORES_PS	2000000	None	Store uops retired to ar
MEM_UOPS_RETIRED.LOCK_LOADS_PS	10000	None	Load uops with locked a

それでは、[Start] ボタンをクリックしてプロファイルを開始してください。
結果が表示されたら [PMU Events] 画面を開いて目的のイベント数をチェックします。フォルス・シェアリングが“setQueen”関数で発生していることがわかります。



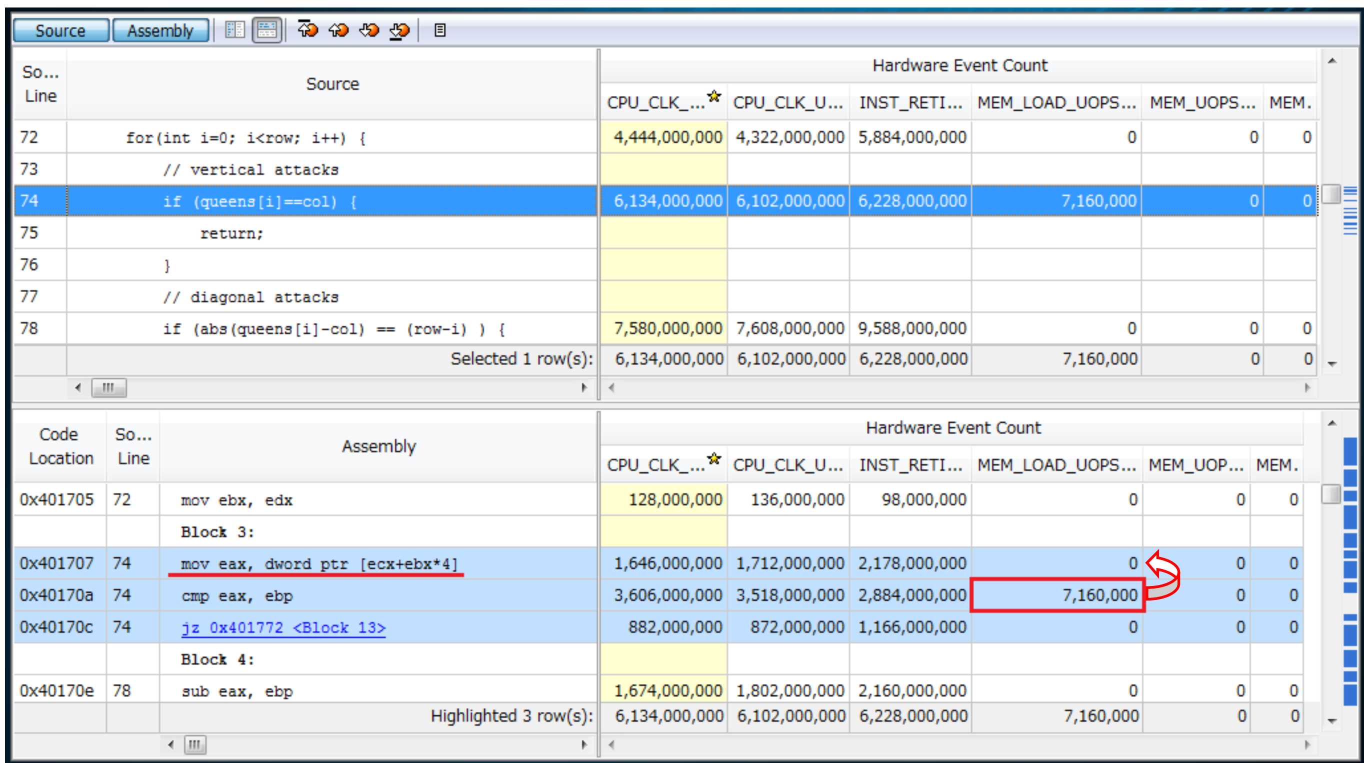
Function / Call Stack	INST_RETI...	CPU_CLK_UN...	CPU_CLK_...	MEM_LOAD_UO...	MEM_UOPS...	MEM_..
setQueen	35,824,000,000	28,548,000,000	28,564,000,000	7,480,000	2,320,000,000	0
__kmp_end_split_barrier	748,000,000	918,000,000	890,000,000	0	96,000,000	0
__kmp_x86_pause	314,000,000	336,000,000	340,000,000	0	146,000,000	0
__kmp_barrier	250,000,000	296,000,000	286,000,000	0	34,000,000	0

次に、その行をダブルクリックしてソースコードを開きます。
内容を確認すると queens ポインタ内の値と col 変数の値を比較している処理と solcnt 配列の値をインクリメントしている処理の 2 箇所でイベント値が大きくなっています。これらの処理をアセンブリで表示して詳細を見ます。アセンブリを表示する場合は、左上にある Assembly トグルボタンをクリックします。表示が見えにくい場合は、 ボタンをクリックして上下表示に切り替えます。



So... Line	Source	CPU_CLK_...☆	CPU_CLK_U...	INST_RETI...	MEM_LOAD_UOPS...	MEM_UOPS...	MEM.
72	for(int i=0; i<row; i++) {	4,444,000,000	4,322,000,000	5,884,000,000	0	0	0
73	// vertical attacks						
74	if (queens[i]==col) {	6,134,000,000	6,102,000,000	6,228,000,000	7,160,000	0	0
75	return;						
76	}						
77	// diagonal attacks						
78	if (abs(queens[i]-col) == (row-i)) {	7,580,000,000	7,608,000,000	9,588,000,000	0	0	0
79	return;						
80	}						
81	}						
82	// column is ok, set the queen						
83	queens[row]=col;	56,000,000	58,000,000	84,000,000	0	0	0
84							
85	if(row==size-1) {	98,000,000	98,000,000	130,000,000	0	166,000,000	0
86	//#pragma omp critical						
87	// nrOfSolutions++; //Placed final queen, founc						
88	solcnt[thrd_id]++;	28,000,000	30,000,000	6,000,000	320,000	0	0
89	}						
	Selected 2 row(s):	6,162,000,000	6,132,000,000	6,234,000,000	7,480,000	0	0

まず queens ポインタに関するアセンブリの内容を見ると、比較命令（cmp）のラインにイベントが表示されています。しかし EBS の結果をアセンブリレベルで読む場合は実際のイベント命令と表示される命令がずれる性質があることを考慮しなければなりません。この場合は表示より一つ前に実行された命令に置き換える必要があります。そうするとこのイベントの対象命令は queens ポインタ内の値をレジスタにロードする命令（mov）となります。



So... Line	Source	CPU_CLK_...☆	CPU_CLK_U...	INST_RETI...	MEM_LOAD_UOPS...	MEM_UOPS...	MEM.
72	for(int i=0; i<row; i++) {	4,444,000,000	4,322,000,000	5,884,000,000	0	0	0
73	// vertical attacks						
74	if (queens[i]==col) {	6,134,000,000	6,102,000,000	6,228,000,000	7,160,000	0	0
75	return;						
76	}						
77	// diagonal attacks						
78	if (abs(queens[i]-col) == (row-i)) {	7,580,000,000	7,608,000,000	9,588,000,000	0	0	0
	Selected 1 row(s):	6,134,000,000	6,102,000,000	6,228,000,000	7,160,000	0	0

Code Location	So... Line	Assembly	CPU_CLK_...☆	CPU_CLK_U...	INST_RETI...	MEM_LOAD_UOPS...	MEM_UOP...	MEM.
0x401705	72	mov ebx, edx	128,000,000	136,000,000	98,000,000	0	0	0
		Block 3:						
0x401707	74	mov eax, dword ptr [ecx+ebx*4]	1,646,000,000	1,712,000,000	2,178,000,000	0	0	0
0x40170a	74	cmp eax, ebp	3,606,000,000	3,518,000,000	2,884,000,000	7,160,000	0	0
0x40170c	74	jz 0x401772 <Block 13>	882,000,000	872,000,000	1,166,000,000	0	0	0
		Block 4:						
0x40170e	78	sub eax, ebp	1,674,000,000	1,802,000,000	2,160,000,000	0	0	0
		Highlighted 3 row(s):	6,134,000,000	6,102,000,000	6,228,000,000	7,160,000	0	0

このロード命令は他のスレッドからも並列実行されますので､別な場所で queens への書き込み処理があることを考えるとフォルス・シェアリングが発生しえるケースとなります。

それから solcnt 配列のインクリメント処理に対するアセンブリを確認します。こちらも同様な読み方をすると solcnt 配列の値をインクリメントする命令（inc）にイベント値が相当します。solcnt 配列はスレッド間で共有している変数ですのでフォルス・シェアリングの対象となりえます。

Source | Assembly

So... Line	Source	Hardware Event Count CPU_CLK_...☆	CPU_CLK_U...	INST_RETI...	MEM_LOAD_UOPS...	MEM_UOPS...	MEM.
85	`if(row==size-1) {`	98,000,000	98,000,000	130,000,000	0	166,000,000	0
86	`//#pragma omp critical`						
87	`//        nrOfSolutions++;  //Placed final queen, found`						
88	`solcnt[thrd_id]++;`	28,000,000	30,000,000	6,000,000	320,000	0	0
89	`}`						
90	`else {`						
91	`// try to fill next row`						
	Selected 1 row(s):	28,000,000	30,000,000	6,000,000	320,000	0	0

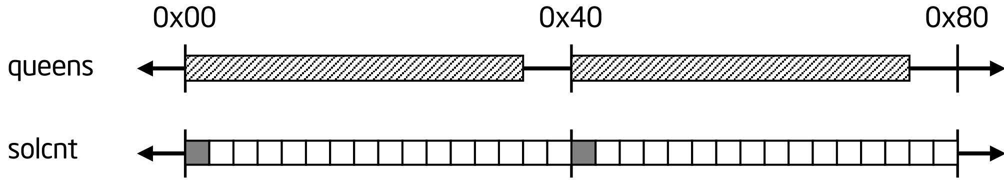
Code Location	So... Line	Assembly	Hardware Event Count CPU_CLK_...☆	CPU_CLK_U...	INST_RETI...	MEM_LOAD_UOPS...	MEM_UOP...	MEM.
		`Block 8:`						
0x401736	88	`mov eax, dword ptr [esp+0x34]`	6,000,000	2,000,000	0	0	0	0
0x40173a	88	`inc dword ptr [eax*4+0x407c40]`	0	8,000,000	2,000,000	0	0	0
0x401741	88	`add esp, 0x14`	22,000,000	20,000,000	4,000,000	320,000	0	0
0x401744	88	`pop ebp`						
0x401745	88	`pop ebx`						
0x401746	88	`pop edi`						
		Highlighted 8 row(s):	28,000,000	30,000,000	6,000,000	320,000	0	0

それではこれらの考察をソースコードに反映してみます。

まず、queens ポインタは solve() 関数の OpenMP ワークシェア構文内で定義され、for ループの回数分ヒープからメモリー領域を取得します。この取得するメモリー領域を６４バイト境界とすることでスレッド間でのキャッシュラインの共有を防ぐことができます。(なお、取得するメモリーサイズは (size(int) * 14) となり 32 ビット環境では 56 バイトです。) ここでは、アライメント付きの動的メモリー取得関数（_mm_malloc）と開放関数（_mm_free）を使用します。

それから solcnt 配列は int 型の配列でそれぞれの配列要素を各スレッドで使用していますので、キャッシュラインを共有しないために６４バイトの間隔を作ります。また、こちらも６４バイト境界として定義します。

以下に、queens と solcnt の 64 バイト境界による論理メモリー上のイメージを示します。



それでは、次にこれらのチューニングを施したプログラムの内容を記します。

```
#include <iostream>
#include <windows.h>
#include <mmsystem.h>

#include "omp.h"

using namespace std;

//int solcnt[32];   // コメントアウト（キャッシュラインの共有を避ける）
__declspec(align(64)) int solcnt[32][16];      // 64バイトの要素間隔と64バイト境界で宣言
```

```
void solve() {

	int thrd_max = omp_get_max_threads();

#pragma omp parallel
	{
		int thrd_id = omp_get_thread_num();

#pragma omp for
		for(int i=0; i<size; i++) {
			//int * queens = new int[size];  // コメントアウト
			int * queens = (int *) _mm_malloc(sizeof(int)*size, 64);    // 64バイト境界
			// try all positions in first row
			setQueen(queens, 0, i, thrd_id);
			_mm_free(queens);   // メモリー解放
		}
	}

	for(int i=0; i<thrd_max; i++) {
		nrOfSolutions += solcnt[i][0] ;  // 解答(配列宣言に合わせて変更)
	}

}
```

```
void setQueen(int queens[], int row, int col, int thrd_id) {

	...

	// column is ok, set the queen
	queens[row]=col;

	if(row==size-1) {
		solcnt[thrd_id][0]++;      // キャッシュラインの共有を避ける
	}
	else {
		// try to fill next row
		for(int i=0; i<size; i++) {
			setQueen(queens, row+1, i, thrd_id);
		}
	}
}
```

修正を加えたコードをビルドして実行し、パフォーマンスを確認してください。また、再度インテル® VTune Amplifier XE の EBS を利用してフォルス・シェアリングをチェックしてください。

（※ CPU のアーキテクチャーによって効果は異なる可能性があります。）

4．関連情報

本章では、その他の関連情報を紹介します。

4－1．インテル® Composer XE

4－1－1．主要コンパイルオプション

オプション	内容
・高度な最適化オプション	
/O3 IDE プロパティページ： [C/C++] - [最適化] - [最適化]	デフォルトの /O2 オプションに加えて、更にループやメモリーアクセスに関する最適化を行う。特に、ループの中で浮動小数点を多用している場合に効果的になります。 それ以外の場合は、/O2 よりも遅くなる可能性もあります。 （例）> icl /O3 main.cpp
・プロシージャー間の最適化（IPO）	
/Qipo IDE プロパティページ： [C/C++] - [最適化[インテル(R) C++]] - [プロシージャー間の最適化]	プログラム全体の最適化。関数のインライン展開などを行う。関数インライン展開を行うことにより、関数コールのオーバーヘッドの他に、他の最適化オプションの機会を広げることに貢献します。本オプションは "Release" 構成の場合、デフォルトで設定されています。
・ベクトル化オプション	
/Qx{SSE2|SSE3|SSSE3|SSE4.1|SSE4.2|AVX|CORE-AVX2|CORE-AVX-I|Host} IDE プロパティページ： [C/C++] - [コード生成[インテル(R) C++]] - [指定された命令セットの専用コード生成]	インテル・プロセッサーに特化したベクトル化オプション。指定した SSE 命令セットでコードを生成する。ただし、指定された SSE 命令セットを搭載しないプロセッサー上では動作しないので注意が必要。 （例１）AVX 命令セットを使用したベクトル化コードを生成。AVX 命令セットを搭載しない CPU では動作しない。 > icl /O2 /QxAVX main.cpp （例２）AVX2 命令セットを使用したベクトル化コードを生成。 AVX2 命令セットを搭載しない CPU では動作しない。 > icl /O2 /QxCORE-AVX2 main.cpp （例３）/QxHost を指定した場合は、コンパイラーが開発システム（Host）のプロセッサーを自動検出し、搭載されている最新の SSE 命令セットを使用して main.exe を生成する。これは、開発システムが実行環境である場合に、便利なオプションとなる。 > icl /O2 /QxHost main.cpp

/Qax{SSE2|SSE3|SSSE3|SSE4.1|SSE4.2|AVX|CORE-AVX2|CORE-AVX-I} IDE プロパティページ： [C/C++] - [コード生成[インテル(R) C++]] - [指定された命令セットの専用および汎用コード生成]	インテル・プロセッサーに特化したコード、および汎用コード（デフォルトでSSE2 レベル）を生成するベクトル化オプション。このオプションは、/Qx 系のオプションとは対照的に汎用コードも生成し、実行環境によって動的に実行パスを切り替える（自動ディスパッチする）コードを生成する。 （例１）AVX 命令セットを使用したベクトル化コードおよび一般 SSE2 命令セットを使用したベクトル化コードを生成する。実行パスは自動ディスパッチャーによって実行時に決定される。 > icl /O2 /QaxAVX main.cpp （例２）AVX 命令セットを使用したベクトル化コードおよび SSE4.2 命令セットを使用したベクトル化コードを生成する。実行パスは自動ディスパッチャーによって実行時に決定される。 > icl /O2 /QaxAVX /QxSSE4.2 main.cpp （例３）SSE4.2 命令セットを使用したベクトル化コードおよび IA-32 命令（x86/x87 命令）を使用したコードを生成する。実行パスは自動ディスパッチャーによって実行時に決定される。 > icl /O2 /QaxSSE4.2 /arch:IA32 main.cpp
/arch:{SSE2|SSE3|SSSE3|SSE4.1|SSE4.2|AVX|IA-32} IDE プロパティページ： [C/C++] - [コード生成] - [拡張命令セットを有効にする]	インテル・プロセッサーおよびインテル・プロセッサー以外の CPU で動作するベクトル化オプション。 （例）一般 SSE2 命令セットを使用したベクトル化コードを生成する。このオプションは /O2 以上を指定した場合は暗黙的に設定される。たとえば "Release" 構成では、デフォルト設定となる。 > icl /O2 /arch:SSE2 main.cpp
・自動並列化オプション	
/Qparallel IDE プロパティページ： [C/C++] - [最適化[インテル(R) C++]] - [並列化]	インテルコンパイラーによる自動マルチスレッドコード生成オプション。このオプションは、コンパイラーがコンパイル時に、ソースコードのループ文に対して並列化を試み、「安全」に並列化できる場合、および並列化することによって「効率」が得られる場合に限り並列化の実装を行います。ループ間に依存関係がある場合や、処理データの量が少ない場合などは、この機能は適用されません。 （例）> icl /Qparallel main.cpp
/Qpar-threshold[n]　(n = 0～100) IDE プロパティページ：なし [C/C++] - [コマンドライン] - [追加オプション]	自動並列化の効率性の閾値をコントロールするオプション。 この閾値を下げることによってコンパイラーによる効率性のチェックレベルを下げることができ、自動並列化の可能性を高めることができる。 なお、デフォルトの閾値は 100 に設定されている。 （例）> icl /Qparallel /Qpar-threshold:90 main.cpp

・OpenMP 関連オプション	
/Qopenmp IDE プロパティページ： [C/C++] – [言語[インテル(R) C++]] – [OpenMP サポート]	ソースコード内の OpenMP 記述子（#pragma omp）を認識し、マルチスレッドコードを生成する。プログラムの実行にはインテルが提供する OpenMP 動的リンクライブラリー（libiomp5md.dll）が必要となる。 （例）> icl /Qopenmp main-omp.cpp
/Qopenmp-stubs IDE プロパティページ： [C/C++] – [言語[インテル(R) C++]] – [OpenMP サポート]	ソースコード内の OpenMP 記述子を無視し、また OpenMP ランタイム関数については空の関数ライブラリーをリンクする。OpenMP のソースコードに対し、シングルスレッドのコードを生成する場合に利用するオプション。 （例）> icl /Qopenmp-stubs main-omp.cpp
・レポート関連オプション	
/Qopt-report{0-3} IDE プロパティページ： [C/C++] – [診断[インテル(R) C++]] – [最適化診断レベル]	最適化に関する診断情報をレポートする。 診断情報のレベルを 0 から 3 まで指定できる。 （例）> icl /O3 /Qipo /Qopt-report2 main.cpp
/Qvec-report{0-6} IDE プロパティページ： [C/C++] – [診断[インテル(R) C++]] – [ベクトライザーの診断レベル]	ベクトル化に関する診断情報をレポートする。 診断情報のレベルを 0 から 6 まで指定できる。 （例）> icl /O2 /QxAVX /Qvec-report3 main.cpp > icl /O2 /Qvec-report2 main.cpp (デフォルトオプション /arch:SSE2)
/Qpar-report{0-3} IDE プロパティページ：なし [C/C++] – [コマンドライン] – [追加オプション]	自動並列化に関する診断情報をレポートする。 診断情報のレベルを 0 から 3 まで指定できる。 （例）> icl /Qparallel /Qpar-report3 main.cpp
/Qopenmp-report{0-2} IDE プロパティページ：なし [C/C++] – [コマンドライン] – [追加オプション]	OpenMP による並列化に関する診断情報をレポートする。 診断情報のレベルを 0 から 2 まで指定できる。 （例）> icl /Qopenmp /Qopenmp-report2 main-omp.cpp

４－１－２．静的解析機能の検出対象問題

インテル® コンパイラーの静的解析機能によって検出される問題の詳細は、以下のインストールされるドキュメントに記載されています。

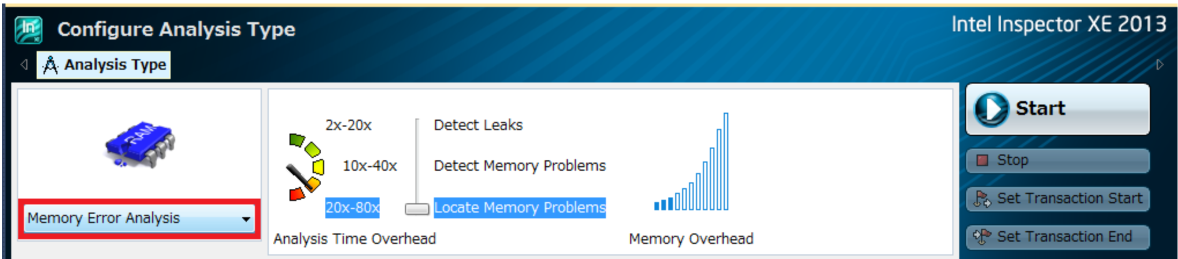
<製品インストールフォルダー>¥Composer XE 2013¥
Documentation¥en_US¥ssadiag_docs¥problem_type_reference.chm

4－2．インテル® Inspector XE

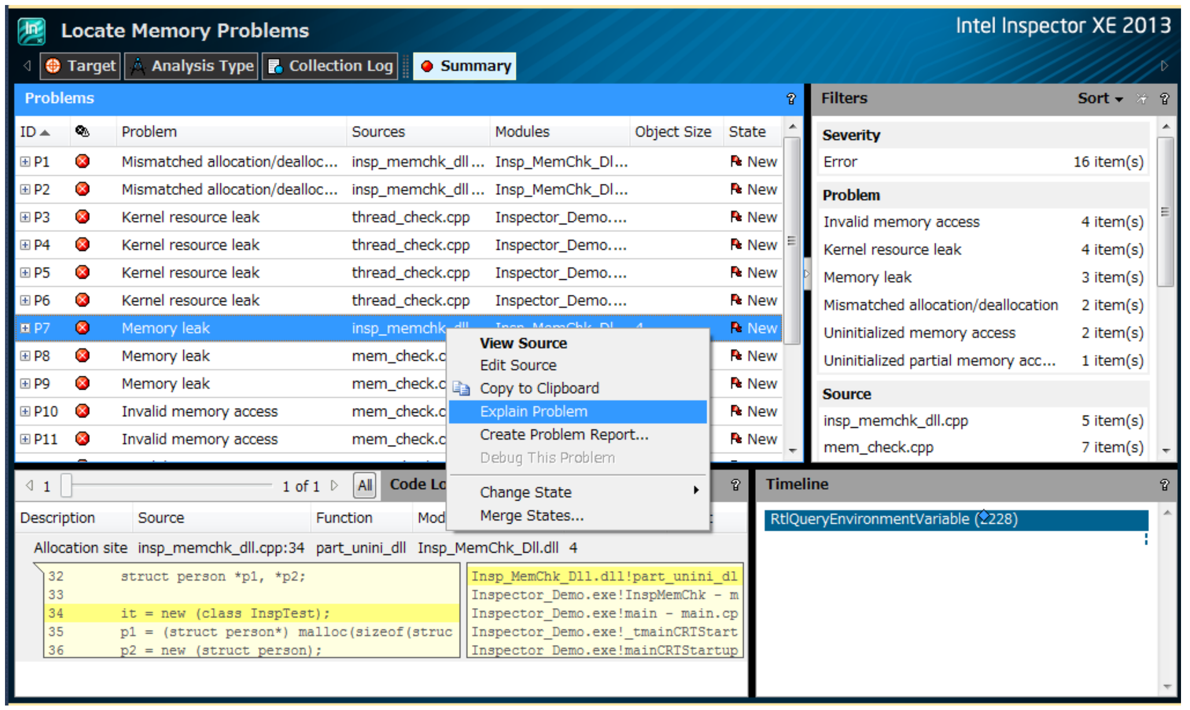
4－2－1．メモリーチェック機能

インテル® Inspector XE には、前章で説明したスレッドエラーのほかに、メモリー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ーのエラーを検出します。

使用方法はスレッドエラーの検出手順と同様に、Analysis Type 画面から“Memory Error Analysis”を選択して検出レベルを設定し最後に 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。



結果は、たとえば下図のように表示されます。



表示内容は、検出された問題の一覧、問題の種類と場所、ソースコードレベルでの確認画面などが表示されます。問題の種類の内容を知りたい場合は、検出された問題を右クリックして表示されるコンテキストメニューから [Explain Problem] を選択することで確認できます。

４－２－２．検出対象問題の一覧

以下に、インテル® Inspector XE が検出する問題（メモリーおよびスレッド）の一覧を記します。各問題の詳細は、製品ドキュメントの「Intel® Inspector XE 2013」-「Problem Type Reference」の章を参照してください。

- ◆ Cross-thread Stack Access（スレッド間のスレッド領域へのアクセス）
- ◆ Data Race（データの競合）
- ◆ Deadlock（デッドロック）
- ◆ GDI Resource Leak（GDI リソースリーク）
- ◆ Incorrect memcpy Call（不適切な memcpy コール（Linux のみ））
- ◆ Invalid Deallocation（無効なメモリー解放処理）
- ◆ Invalid Memory Access（無効なメモリーアクセス）
- ◆ Invalid Partial Memory Access（無効な部分的メモリーアクセス）
- ◆ Kernel Resource Leak（カーネル・リソースリーク）
- ◆ Lock Hierarchy Violation（ロック処理階層違反）
- ◆ Memory Growth（ある処理区間におけるメモリーの増加）
- ◆ Memory Leak（メモリーリーク：解放不可能）
- ◆ Memory Not Deallocated（メモリーリーク：解放可能）
- ◆ Mismatched Allocation/Deallocation（メモリー取得関数と解放関数の不一致）
- ◆ Missing Allocation（不適切なメモリー解放領域）
- ◆ Thread Start Information（スレッド生成情報）
- ◆ Unhandled Application Exception（処理されない例外）
- ◆ Uninitialized Memory Access（未初期化領域からの読み込み処理）
- ◆ Uninitialized Partial Memory Access（部分的未初期化領域からの読み込み処理）

４－２－３．デバッガーでエラーをトラップする方法

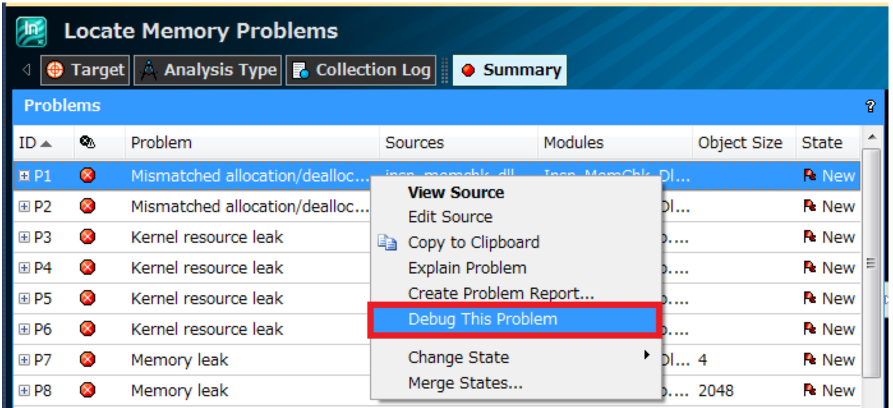
インテル® Inspector XE では、エラー検出箇所でデバッガー表示させる機能があります。この時アプリケーションの実行は停止され、これ以降は通常のデバッガー操作でエラー検出を続行することができます。このエラー検出方法によって問題の発生原因を一つずつ調査しながら作業することができます。

> **ご注意：**ただし、**メモリーリーク**と**リソースリーク**問題についてはこの方法を利用することができません。これは、これらの問題がアプリケーションの終了時に検出可能となるためです。

それでは以下に、このエラー検出を実施する 3 種類の方法をご紹介します。

＜方法１＞

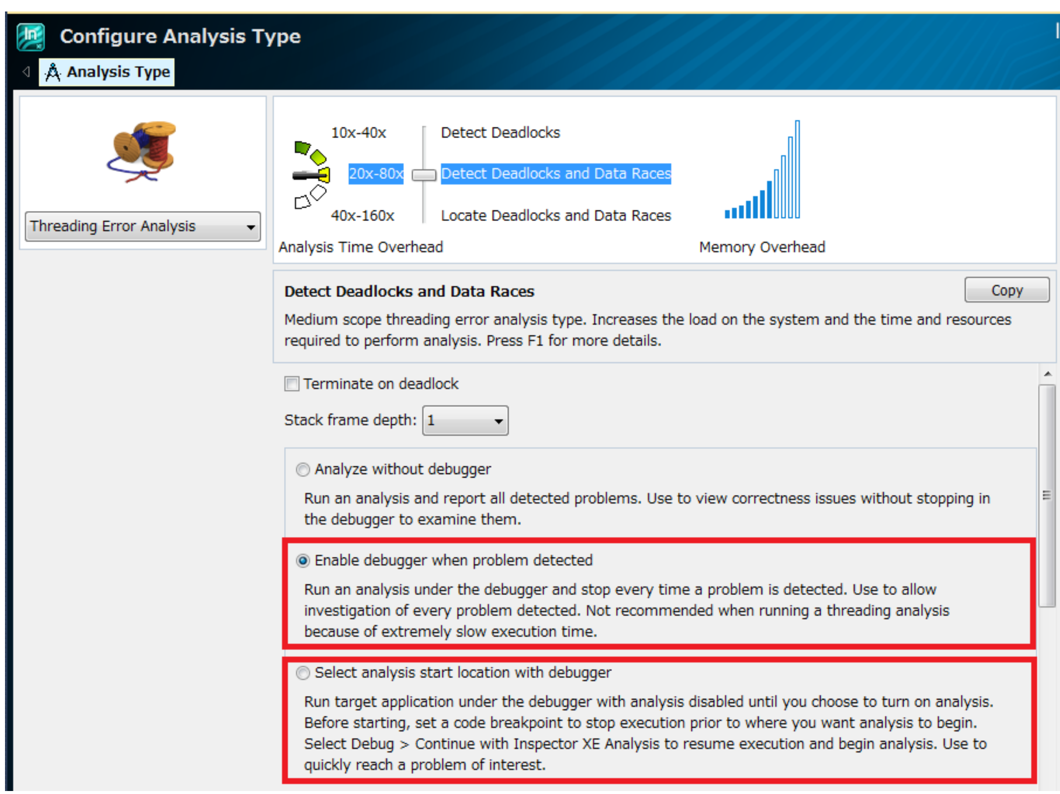
一度エラー検出を行った結果一覧から、特定の問題を選択してそれを右クリックし、表示されるコンテキストメニューから［Debug This Problem］を選択する。この場合、選択した問題についてのみデバッグ表示されます。



＜方法２＞

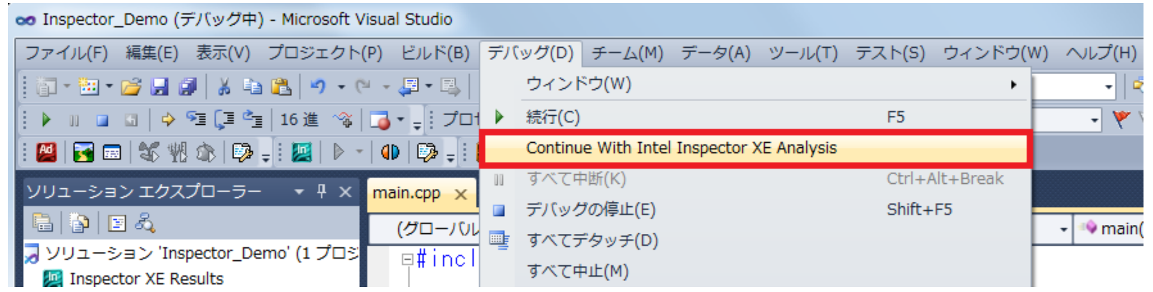
Analysis Type 画面の［Enable debugger when problem detected］を選択してエラー検出を開始する。

この場合、アプリケーションの問題検出が開始され、問題が検出され次第デバッガーに随時表示されます。

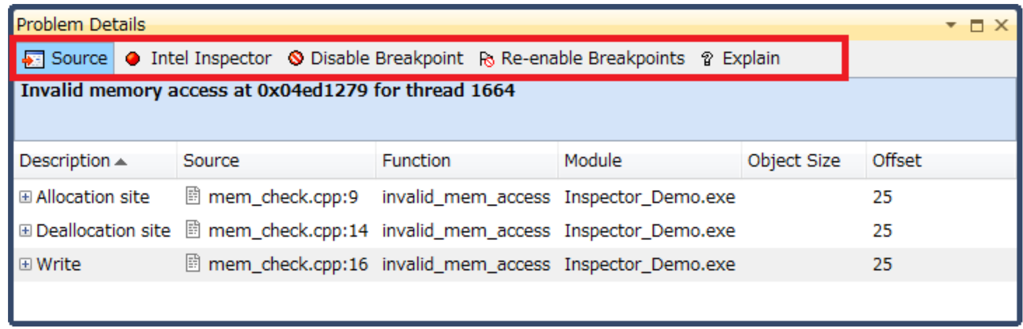


＜方法３＞

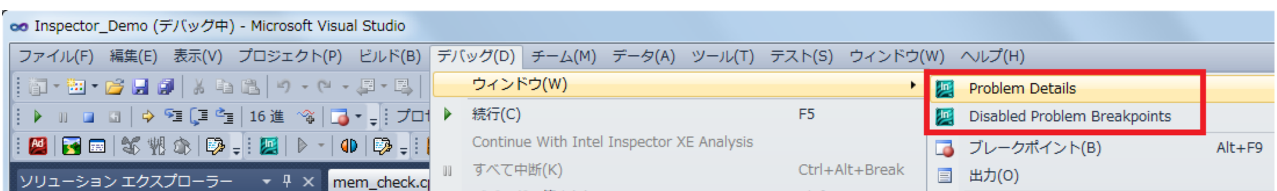
Analysis Type 画面の［Select analysis start location with debugger］を選択してエラー検出を開始する。この方法では、検出開始場所を指定することができます。本エラー検出を開始する前に、検出を開始したい場所にブレークポイントを設定しておきます。アプリケーションがブレークポイントまで実行されたら、Visual Studio のメニューから［デバッグ］-［Continue With Intel Inspector XE Analysis］を選択して検出を開始します。



これらの方法を使用して問題が検出された場合、アプリケーションが停止してデバッガーが表示されると共に、[Problem Details] ウィンドウも表示されます。このウィンドウには検出された問題の内容が表示されます。また、この問題に対する操作を行う複数のボタンも用意されています。たとえば [Disable Breakpoint] をクリックすると、次回同じ問題が発生した際に、その検出を無視することができます。また [Re-enable Breakpoints] をクリックすると、これまでに無効にした問題内容を再度有効にすることができます。



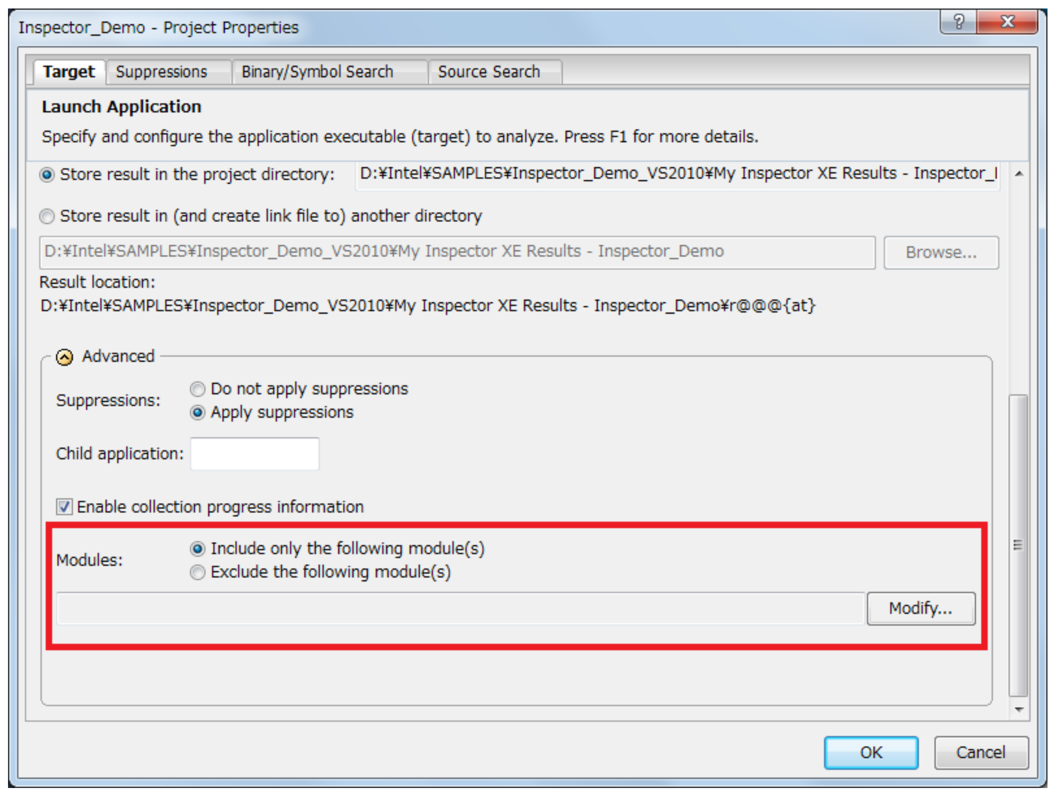
最後に、本機能で使用する２つのウィンドウの表示方法を以下の図に示します。



４－２－４．検出オーバーヘッドの軽減方法

一般的に、動的なエラー検出にはかなりのオーバーヘッドを伴います。このオーバーヘッドを小さくするためにも、まずできるだけ小さな入力値を選択したり、実行処理量が小さくなるように工夫する必要があります。ここでは、検出対象モジュールを調整してオーバーヘッドを軽減する方法をご紹介します。

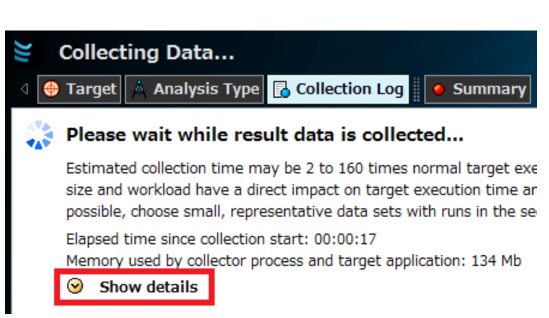
まず、このボタン（ ）をクリックしてインテル® Inspector XE のプロジェクトプロパティーを開きます。モジュールの指定は、一番下にある「Advanced」グループ内の 「Modules:」項目の [Modify...] ボタンで指定します。指定したモジュールに対して「Include only the following module(s)」にチェックした場合は指定したモジュールのみが検出対象となり、「Exclude the following module(s)」にチェックした場合は指定したモジュールは検出対象から外されます。この方法によってオーバーヘッドが軽減される場合があります。

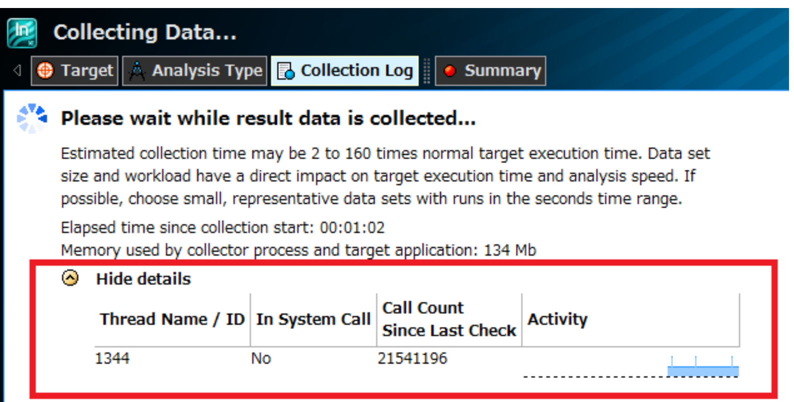


４－２－５．検出中にアプリケーションの状態を確認する方法

アプリケーションが大きくなると検出オーバーヘッドも大きくなる傾向にあり、場合によってはアプリケーションが動作しているのかハングしているのか分からないケースがあります。インテル® Inspector XE ではアプリケーションの動作状況を表示させる機能があります。以下にその手順を記します。

まず、インテル® Inspector XE のプロジェクトプロパティーページを開いて「Advanced」グループ内にある「Enable Collection progress information」にチェックがあることを確認します（前節の図を参照）。次にエラー検出を開始してしばらくすると下左図のように“Show details”という表記が表示されます。この表記の左側にある印をクリックすると下右図のような画面が表示され、アプリケーションのスレッド動作内容が周期的に更新されます。この更新が行われている場合はアプリケーションは動作中ということになります。また表示内容には使用メモリー量も表示されています。





４－２－６．推奨されるコンパイルオプション

インテル® Inspector XE の利用において、影響を与えるコンパイラーオプションを説明します。

オプション	内容
・推奨オプション	
/Zi または /ZI	ソースコードを参照するために、シンボル情報が必要となります。
/debug（リンカーオプション）	関数などのシンボル情報が必要となります。
・影響を与えるオプション	
/Od	検出されたエラーをソースコード上で正しく表示することができます。最適化されたバイナリーではソースコードを正しく表示できない場合があります。しかし最適化ありの Release 構成でビルドされたバイナリーに対しても同様に診断を行うことが必要です。
/MDd または /MD	特にスレッドエラーチェックの検出精度に影響を与えます。
・推奨されないオプション	
/RTC[su1]	メモリーエラーチェックの動作に影響を与えます。 このオプションは、インテル® Inspector XE の動作と似た内部処理を行うため、"false positives" や "false negatives" のご検出を招く可能性があります。 またこのオプションは未初期化メモリーを初期化してしまうのでインテル® Inspector XE によるメモリー検出を阻害する場合もあります。

４－２－７．動的検出 vs. 静的検出

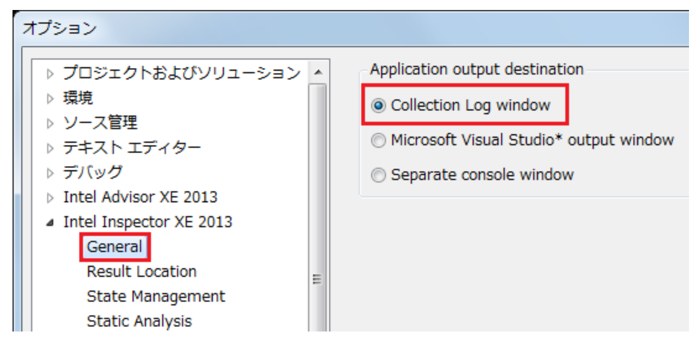
インテル® Parallel Studio XE では、メモリー／スレッドエラーの**動的検出**および**静的検出**をサポートします。
動的検出は、プログラムを実行させながら検出処理を行う手法でインテル® Inspector XE の機能を使用します。静的検出は、プログラムを実行せずに検出処理を行う手法でインテル® コンパイラーに含まれる静的解析の機能を使用します。両者の検出方法には、長所と短所がそれぞれ存在します。そのためより確実なエラー検出を行うために両方の検出処理を実施しそれぞれの短所を補うことが望まれます。
動的検出の長所としては、実行しないと分からない問題や静的検出では複雑すぎて検出不可能なケースでも検出することができる点にあります。一方、静的検出の長所としては、コンパイル対象となるすべての処理が検出対象となる点です。動的検出のように実際に実行されるパスだけでなく通常実行されないパスも対象となります。この点は脆弱性に対する攻撃防止にも役立ちます。

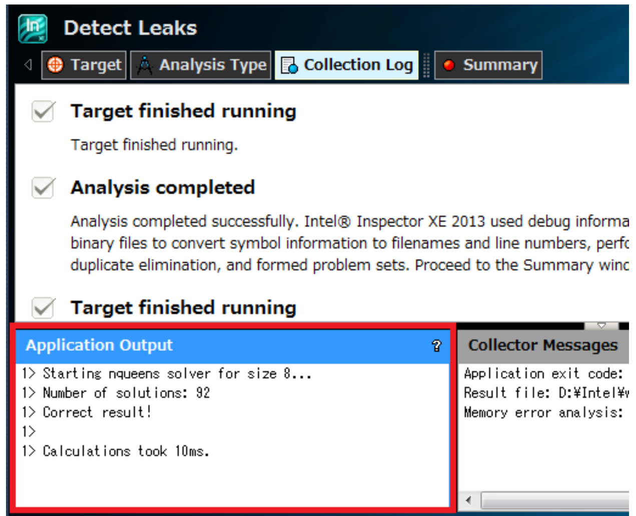
4－2－8．Tips

ここでは、知っていると便利なマイナー機能についていくつかご紹介します。

① コンソール画面の出力先変更

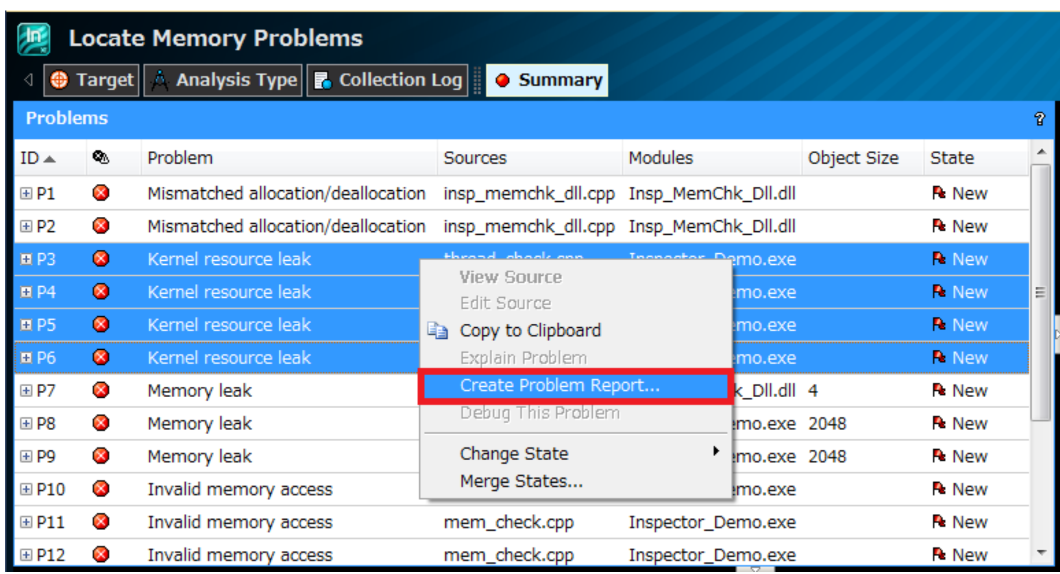
Visual Studio のメニューから [ツール] - [オプション] を選択して、下図のように "Collection Log window" をチェックすると、コンソールアプリケーションの出力内容を通常のコマンドウィンドウからインテル® Inspector XE の [Application Output] 画面に表示させることができます。





② 検出結果レポートの作成

検出結果内容を基に、レポートを作成することができます。下図のように Summary 画面からレポート作成したい問題を選択して右クリックし、表示メニューから「Create Problem Report...」をクリックするとレポート内容が表示されます。レポート内容をファイル保存することができます。



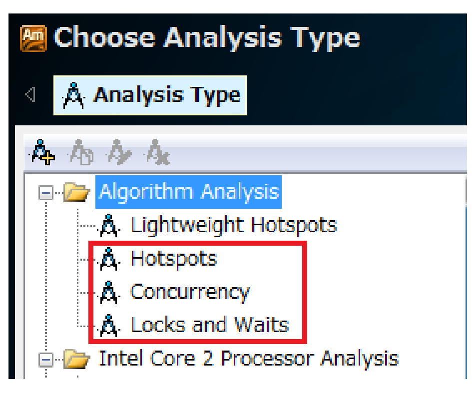
４－３．インテル® VTune Amplifier XE

４－３－１．サンプリング・メカニズム

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。

① **「ユーザーモードサンプリングおよびトレースコレクション」**

このサンプリング手法は、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。



本サンプリング手法を使用した解析タイプは、左図に示すとおり以下の 3 種類のみとなります。

- Hotspots … CPU 時間を多く消費する処理を特定する。
- Concurrency … マルチスレッド・アプリケーションの並列性能を解析する。
- Lock and Waits … スレッドをアイドル状態にする原因を特定する。

② **「イベントベース・サンプリングコレクション」**

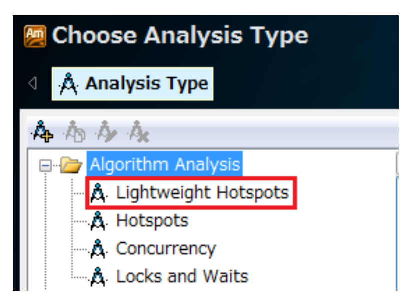
このサンプリング手法は、インテルプロセッサーに搭載される PMU（パフォーマンス・モニタリング・ユニット）のイベントカウンター・オーバーフローによる割り込みを利用してプロファイル情報を収集します。この割り込みの発生回数がサンプリング回数となります。また割り込みの発生頻度は、PMU のオーバーフロー値を決定する “Sample After Value” を調整することで制御できます。例えば 1ms 単位で割り込みが発生した場合、それによるオーバーヘッドは約 2% 程度となります。

> **ご注意：**「イベントベース・サンプリングコレクション」は PMU を使用するため、このサンプリング手法を利用できるプロセッサーが限定されています。通常インテル® Core Microarchitecture 以上であれば動作します。詳細は製品リリースノートを参照してください。
> 一方、「ユーザーモードサンプリングおよびトレースコレクション」の場合はプロセッサーの制限を受けません。

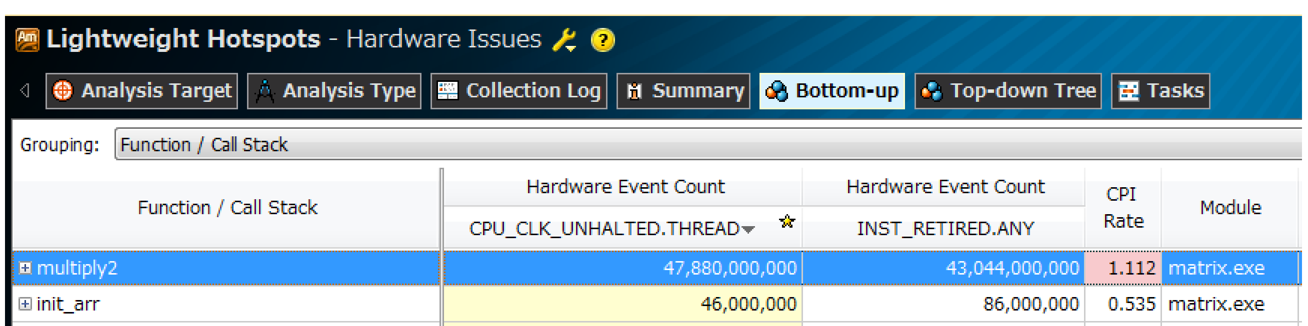
本サンプリング手法の解析タイプは、上記で示した「ユーザーモードサンプリングおよびトレースコレクション」の解析タイプ（Hotspots／Concurrency／Locks and Waits）以外のすべてとなります。また、PMU はプロセッサー・アーキテクチャー単位でそれぞれ異なるため、アーキテクチャー別に解析タイプが用意されています。

４－３－２．CPI 値について

「イベントベース・サンプリングコレクション」の基本解析タイプとして“Lightweight Hotspots”があります。



この解析タイプでは、主に CPI 値という指標データが表示されます。
CPI とは Clockticks per Instructions Retired の略で、実行完了命令数に対する消費 CPU クロック数を表し、命令実行の効率性を決定するもっとも基本的な指標です。もっと平たく言えば、1 命令の実行にどのくらいの CPU クロックを消費したのか、という値です。当然、命令の実行に消費するクロック数は小さいほうがよいので、CPI 値は小さいほうが効率がよいといえます（CPI = クロック数／命令数）。
現在のインテル CPU（Core Microarchitecture 以上）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。ただし、実際のアプリケーションでは CPI 値が 0.25 になることは通常ありません。
インテル® VTune Amplifier XE では、CPI 値が 1.00 より大きく、CPU 時間が全体の 5%より大きい関数に対して赤色でマーキングし、チューニングの余地があることを示します。



Function / Call Stack	Hardware Event Count CPU_CLK_UNHALTED.THREAD	Hardware Event Count INST_RETIRED.ANY	CPI Rate	Module
multiply2	47,880,000,000	43,044,000,000	1.112	matrix.exe
init_arr	46,000,000	86,000,000	0.535	matrix.exe

上図の例では、“multiply2”関数の CPI 値が 1.112 であり赤色でマーキングされています。この CPI 値は、“Lightweight Hotspots”解析で収集されるイベントの結果から計算されて求められます。ここでは以下の２つのイベントが使用されています。

① CPU_CLK_UNHALTED.THREAD … CPU クロック数

② INST_RETIRED.ANY … 実行完了命令数

- CPI 値 ＝ CPU_CLK_UNHALTED.THREAD ／ INST_RETIRED.ANY

このように CPI 値は、アプリケーション実行の効率性を測る基本指標となります。しかし重要なことは CPI 値はあくまで参考数値であり目的は処理速度の向上（処理時間の減少）にあります。つまり CPU クロックのイベントである CPU_CLK_UNHALTED.THREAD の値を下げることが最終目標となります。

４－３－３．推奨されるコンパイルオプション

インテル® VTune Amplifier XE は、基本的に全てのネイティブバイナリーに対してプロファイリングを行うことができますが、ここでは、プロファイリングに影響を与えるコンパイラーオプションを説明します。

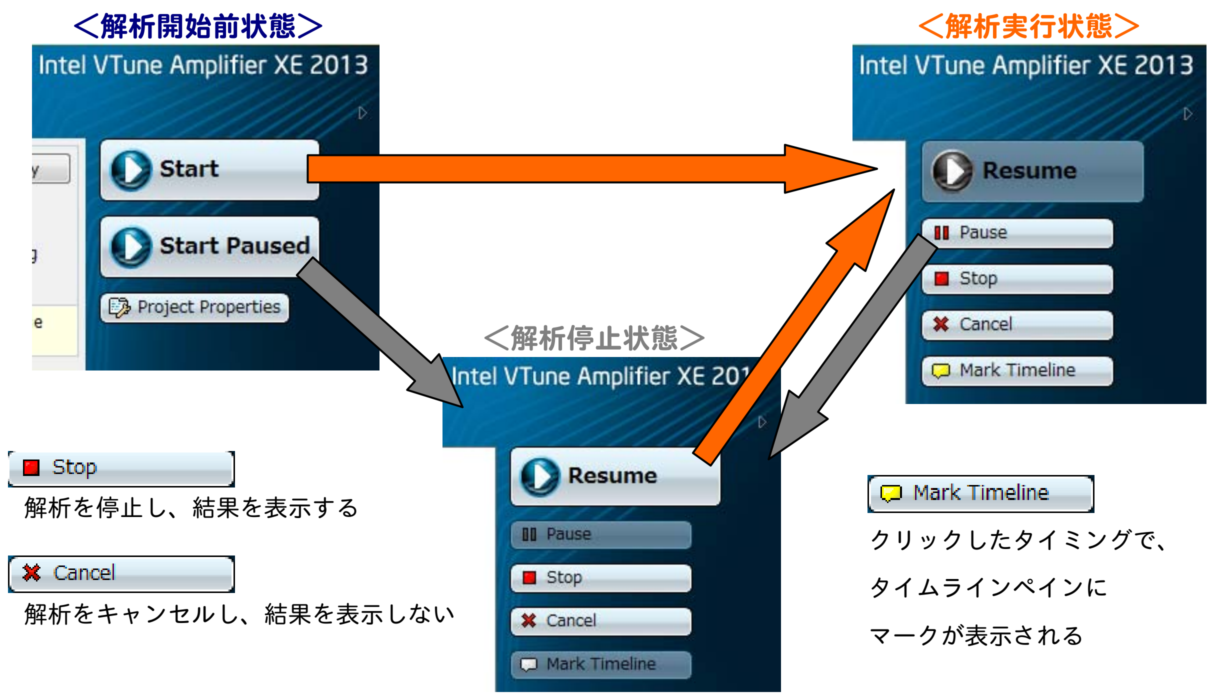
オプション	内容
・推奨オプション	
/Zi	ソースコードを参照するために、シンボル情報が必要となります。
/debug（リンカーオプション）	関数などのシンボル情報が必要となります。
/O2 など（“Release” ビルド）	より実践的なプロファイリングを行うことができます。
/MD または /MDd	C ランタイム関数をユーザ・アプリケーションと区別することができます。
・TBB アプリケーションに推奨されるオプション	
/D"TBB_USE_THREADING_TOOLS"	インテル® VTune Amplifier XE が TBB 関数を正しく認識することができます。
・その他のオプション	
/debug:inline-debug-info（インテルコンパイラー）	インテル® VTune Amplifier XE の inline mode を有効にします。
/Ob0	関数のインライン展開を無効にします。

４－３－４．特定箇所のプロファイル方法

インテル® VTune Amplifier XE は、基本的に Hotspot 関数を特定するプロファイラーです。プロファイル結果は CPU 時間が多い順に表示されます。このためアプリケーションによっては、目的の関数が他の Hotspot 関数によって表示されないケースが発生します。この場合、Pause（ポーズ）機能と、Resume（再開）機能を使用して、プロファイルする処理範囲を特定することで回避できます。また本機能を使用することで評価したい処理に関する情報のみ表示させることが可能となり、結果内容を分析しやすくなります。
この機能の使用方法には、手動でプロファイル範囲を指定する方法と、インテル® VTune Amplifier XE が提供する API を使用して明確にプロファイル範囲を指定する方法があります。

まず、以下に手動による手順を説明します。

手動でこの機能を使用する場合は、ボタンのクリック操作で作業します。
以下の図に示すように、各種ボタンをクリックすることでサンプリングを開始･停止することができます。例えば「解析開始前状態」から [Start] ボタンをクリックすると解析が実行され、また [Start Paused] ボタンをクリックした場合は、解析は実行せずにアプリケーションを起動します。「解析停止状態」から [Resume] ボタンで解析を再開させることができます。



次に、API を使用する手順について説明します。

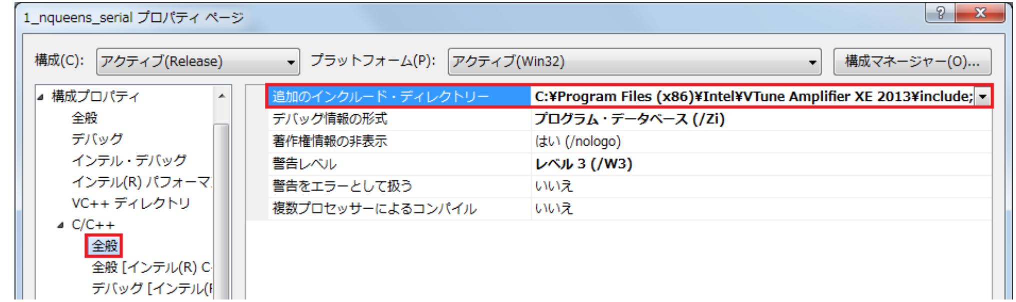
1）ソースコードを編集します。プロファイルを行いたい処理に対してインテル® VTune Amplifier XE が提供する Resume 関数（__itt_resume）と Pause 関数（__itt_pause）を追記します。例えば以下のようにプロファイルする処理を Resume 関数と Pause 関数で挟みます。また API 用ヘッダーを宣言します。

```
#include "ittnotify.h"      // API 関数用ヘッダー
…
Func() {
   前処理              // プロファイルされない処理
   __itt_resume();     // プロファイル開始
   処理 A              // プロファイルされる処理
   __itt_pause();      // プロファイル停止
   中間処理            // プロファイルされない処理
   __itt_resume();     // プロファイル開始
   処理 B              // プロファイルされる処理
   __itt_pause();      // プロファイル停止
   後処理              // プロファイルされない処理
}
```

2）プロジェクトのプロパティページにヘッダーファイル（ittnotify.h）のディレクトリーパスを設定します。設定する内容は、デフォルトインストールを行った場合は以下のパスになります。

設定パス（x86 システム）： C:¥Program Files¥Intel¥VTune Amplifier XE 2013¥include

設定パス（x64 システム）： C:¥Program Files (x86)¥Intel¥VTune Amplifier XE 2013¥include



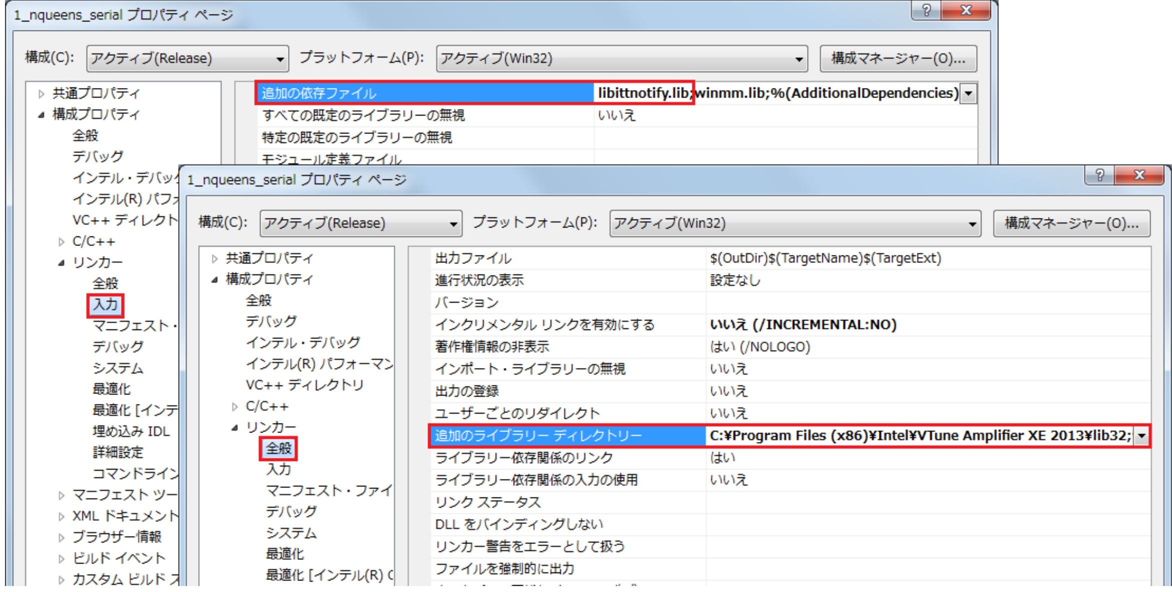
3）続いて、ライブラリーの設定を行います。指定するリンク情報は、デフォルトインストールを行った場合は以下の内容となります。

設定パス（x86 システム）：C:¥Program Files¥Intel¥VTune Amplifier XE 2013¥lib32

設定パス（x64 システム）：C:¥Program Files (x86)¥Intel¥VTune Amplifier XE 2013¥lib32

（※ Intel64 アプリケーションの場合は¥lib32 ではなく¥lib64 を指定してください）

指定ライブラリー：libittnotify.lib



4）設定が完了したら、プロジェクトをビルドし直します。

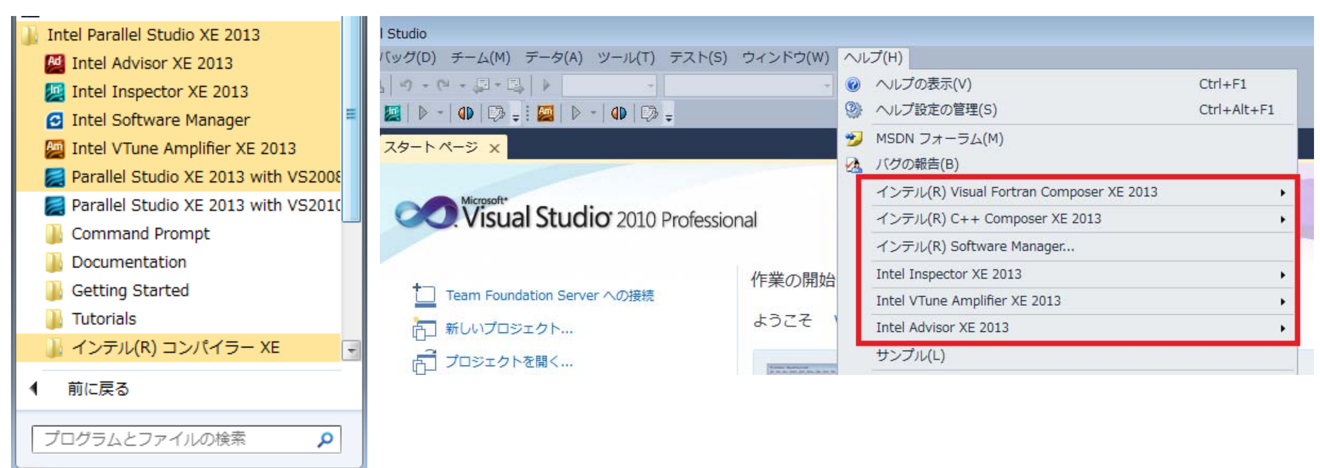
5）ビルドが正常に完了したら、[Start Paused] ボタンをクリックして解析停止状態でアプリケーションを実行します。アプリケーションが Resume 関数を実行したタイミングでプロファイルが開始されます。その後は、Pause 関数、Resume 関数の配置によってプロファイルが行われます。

5．追加情報

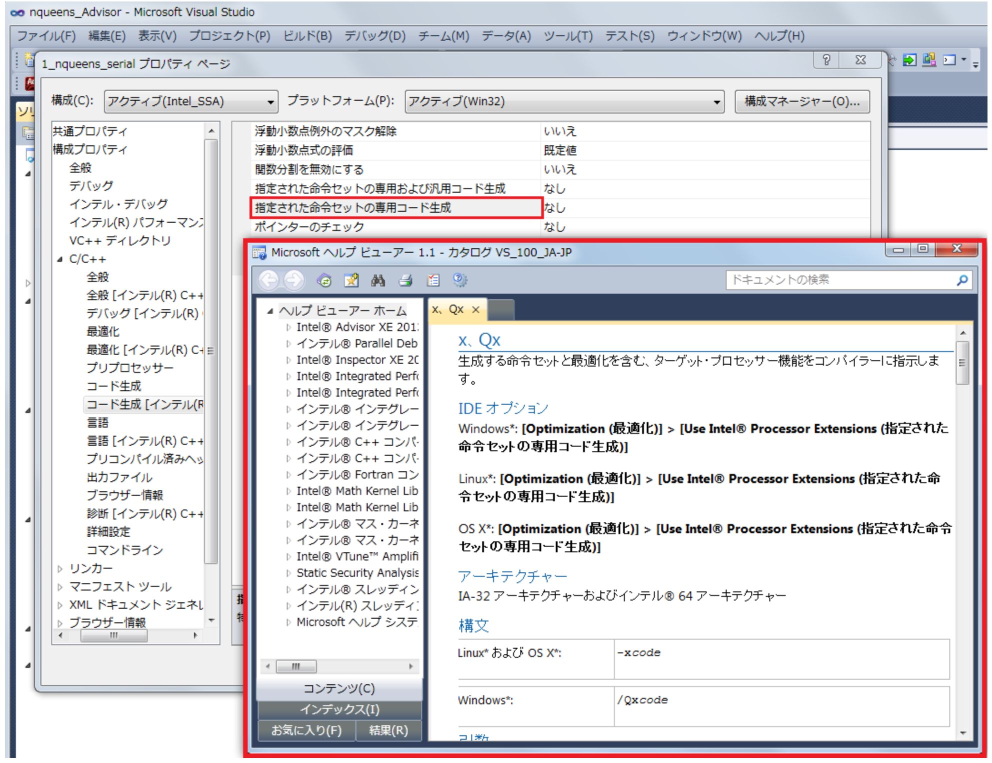
ここでは、本製品に関するその他の情報をいくつかご紹介します。

5－1．ドキュメントの参照方法

製品ドキュメントへのアクセスは Windows の「スタート」メニューから、または Visual Studio の [ヘルプ] メニューより参照することが出来ます。



また、Visual Studio* IDE からは、 [プロパティ ページ] 内の項目に対して“F1” キーに押すことにより、その項目のヘルプが表示されます。



５－２．N-Queens サンプル・プログラム

本ドキュメントでは、N-Queens サンプル・プロジェクト内の 1_nqeens_serial プロジェクトを使用し OpenMP によるマルチスレッド方法を説明しましたが、N-Queens サンプル・プロジェクトにはその他のマルチスレッド手法で記述されたサンプル・プロジェクトが含まれています。

以下に、N-Queens サンプル・プロジェクトに含まれる問題解決方法を記します。

① シンプル・シリアル・ソリューション

② OpenMP 3.0 を使用したパラレル・ソリューション

③ Intel® Cilk Plus を使用したパラレル・ソリューション

④ Intel® TBB を使用したパラレル・ソリューション

５－３．OpenMP について

OpenMP に関する参考資料および参考サイトをご紹介します。

『インテル® コンパイラー OpenMP* 入門』

http://jp.xlsoft.com/documents/intel/compiler/525J-001.pdf

『インテル® コンパイラー OpenMP* 活用ガイド』

http://jp.xlsoft.com/documents/intel/compiler/526J-001.pdf（C言語ユーザー用）

http://jp.xlsoft.com/documents/intel/compiler/527J-001.pdf（Fortran 言語ユーザー用）

・OpenMP ホームページ

http://openmp.org/wp/

５－４．Intel® Cilk™ Plus について

Intel® Cilk™ Plus に関する参考資料および参考サイトをご紹介します。

・Intel® Cilk™ Plus ホームページ

http://software.intel.com/en-us/intel-cilk-plus

５－５．Intel® TBB について

TBB に関する参考資料および参考サイトをご紹介します。

・TBB ホームページ

http://threadingbuildingblocks.org/

・TBB 製品紹介サイト

http://www.xlsoft.com/jp/products/intel/perflib/tbb/index.html
http://software.intel.com/en-us/intel-tbb

６．最後に

本製品に関するその他の情報は以下のサイトをご参照ください。
http://xlsoft.com/jp/products/intel/studio_xe/index.html

本製品に関してご不明な点がありましたら、下記お問い合わせ窓口より弊社サポートまでご連絡ください。
https://www.xlsoft.com/jp/services/xlsoft_form.html

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