インテル® Parallel Composer による N クイーンの並列化

サンプル・コード・ガイド

文書番号: 320506-001JA

改訂: 1.2

Web サイト: http://www.intel.com/ (英語)

著作権と商標について

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスも許諾するものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他知的財産権の侵害への保証を含む) にも一切応じないものとします。

インテルによる書面での合意がない限り、インテル製品は、その欠陥や故障によって人身事故が発生するようなアプリケーションでの使用を想定した設計は行われていません。

インテル製品は、予告なく仕様や説明が変更される場合があります。機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。この情報は予告なく変更されることがあります。この情報だけに基づいて設計を最終的なものとしないでください。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

最新の仕様をご希望の場合や製品をご注文の場合は、お近くのインテルの営業所または販売代理店にお問い合わせください。

本書で紹介されている注文番号付きのドキュメントや、インテルのその他の資料を入手するには、1-800-548-4725 (アメリカ合衆国) までご連絡いただくか、インテルの Web サイトを参照してください。

インテル® プロセッサー・ナンバーはパフォーマンスの指標ではありません。プロセッサー・ナンバーは同一プロセッサー・ファミリー内の製品の機能を区別します。異なるプロセッサー・ファミリー間の機能の区別には用いません。詳細については、http://www.intel.com/products/processor_number/ を参照してください。

Intel、インテル、Intel ロゴは、アメリカ合衆国およびその他の国における Intel Corporation の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

目次

1. はじめに

このサンプル・コード・ガイドでは、インテル® Parallel Composer で利用できる既存および新しい並列プログラミングの言語拡張、宣言子、ライブラリーについて説明します。

本ガイドは、インテル® Parallel Composer で提供される並列化の選択肢の呈示と既定のアルゴリズムへの適用方法を示すことを基本的な目的としています。既存のシリアル・アプリケーションを並列化する際に、どの程度のコーディング作業が必要なのか、またどのメカニズムが高次元の構造でスレッド化を行う利点を提供するかを重要な課題として言及します。

サンプルで示す新しい並列化手法は次のとおりです。

- OpenMP* 3.0 宣言子
- 新しい並列拡張機能: `__taskcomplete`、`__task`、`__finish`、`__par`
- インテル® スレッディング・ビルディング・ブロック (インテル® TBB) と "ラムダ" 関数言語拡張

上記の新しい手法はそれぞれ、以前のインテル® C++ コンパイラー製品では利用できなかった並列アプリケーションの開発における新たな範囲のスケーラビリティー、パフォーマンス、ユーザビリティーを C++ 開発者に提供します。

これらの新機能の使用方法とパフォーマンスを説明する例として、本ガイドでは、N クイーン問題を取り上げ、各手法を使用して並列ソリューションを実装する方法を示します。N クイーン問題とは、明確な解がある有名なプログラミング問題です。本ガイドは、各並列手法を使用して、各コードの実装を示し、その有用性とパフォーマンスを比較対照します。

本ガイドで示すすべてのコードブロックと例は、インテル® Parallel Composer に含まれている実際のコードサンプルを使用しています。

1.1. N クイーンのサンプル

N クイーンのサンプルは、Microsoft* Visual Studio* プロジェクト、ソリューション、C/C++ ソースファイルのコレクションです。このアルゴリズムには、さまざまなアプローチ方法 (例えば、速度やメモリー、あるいはその両方を最適化する方法) があります。そのため、いくつかの異なるサンプルが用意されています。サンプルでは、N クイーン問題の解を求める各種アプローチとアルゴリズムを例証します。

N クイーン問題は、世界中の Web サイトで問題の歴史やその他の解決方法が提供されています。

注: N クイーン問題は、1848 年、ドイツのチェスに関する雑誌において Max Bezzel 氏により初めて発表され、解は 1850 年に Franz Nauck 博士により発見されました。N クイーン問題の最初の一般的な解法は 1969 年に発表されています。[1]

1.1.1. サンプルの場所

インテル® Parallel Composer をインストールすると、N クイーンのサンプルは、`<install-directory>¥Samples¥NQueens¥` ディレクトリーに格納されます。

1.2. 問題

N クイーン問題とは、N × N のチェス盤に N 個のクイーンを互いに攻撃することができない位置に配置して、その解の総数を求める問題です。

2 個のクイーンは、縦横同じ列、または斜めの (2*(2*n - 1)) の位置に配置されたときに互いに攻撃することができます。

例えば、8 × 8 の盤の場合、92 個のバリエーション解があります。基本解とバリエーション解の違いは、バリエーション解では盤面の回転と反転は別個として数えられることです。8 × 8 の盤の場合、12 個の基本解があります。

また、N クイーン問題の計算量は指数関数的に増加します。現在の世界最高記録は、25 × 25 の盤の解です。この解の算出には、有名な Seti@home (セティ・アット・ホーム) プロジェクトと同様、世界中の異機種クラスターが使用されました。累積計算時間は 50 年以上にも上ります。

```
The number of solutions is: 2.207.893.435.808.352
Start date: October 8th 2004
End time: June 11th 2005
Computation duration time = 4444h 54m 52s 854 i.e. 185 days 4 hours 54
minutes 52 seconds 854
```

上記は、(http://proactive.inria.fr/index.php?page=nqueens25) からの引用です。

1.2.1. 問題の分析

最も一般的なアプローチは、BF 法 (Brute-Force、総当り方式) を使用することで、これは単純にそれぞれの盤サイズで各配置をテストするものです。

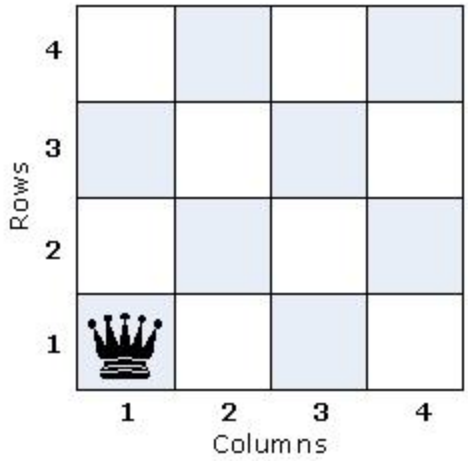
しかし、しかし、Brute-Force (総当り) アルゴリズムを使用して有効な各配置をテストする場合、必然的に莫大な数の配置 -$(n^2)!/((n^2)-n)!$- を検証することになります。4 × 4 の盤でも、配置の候補は 16!/12! =43680 通りにもなります。

代わりに、1 行につき 1 個のクイーンだけを許可するヒューリスティック・アプローチを使用すると、問題サイズを n 要素のベクトルに減少させることができます。この場合、各要素は行の位置を確保し、解の候補は nn 個に減少します。

このアプローチでは、4 × 4 の盤で 256 通りの配置になります。さらに数を減らすには、1 列につき 1 個のクイーンだけを許可します。すると、1 列につき 1 組のクイーン ($Q_1 \ldots Q_n$) となり、解の候補が n! に減少します。4 × 4 の盤では、24 個になります。

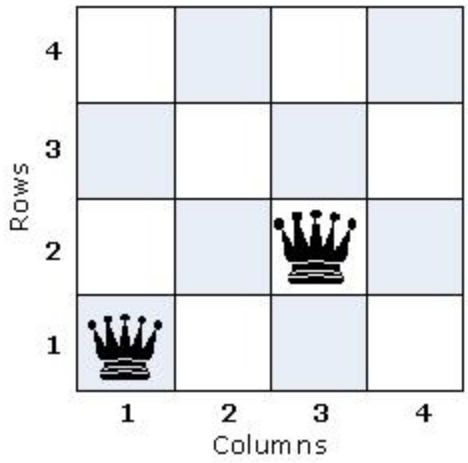
4 × 4 の盤の問題を見直して、1 つの解候補を例証します。まず最初に、図 1 のように 1 行 1 列目に最初のクイーンを置きます。

図 1



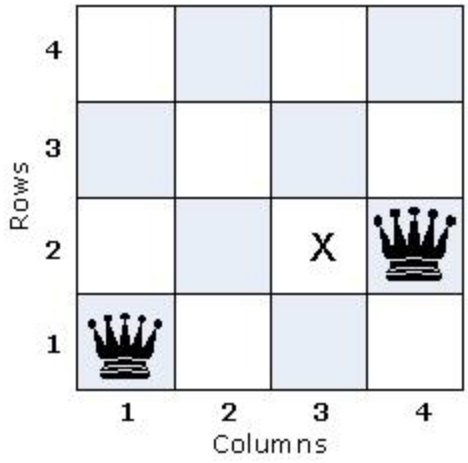
1 列目に別のクイーンを置くことはできないため、次のクイーンは 2 行目に配置する必要があります。最初の可能な配置は 2 行 3 列目 (図 2) です。

図 2



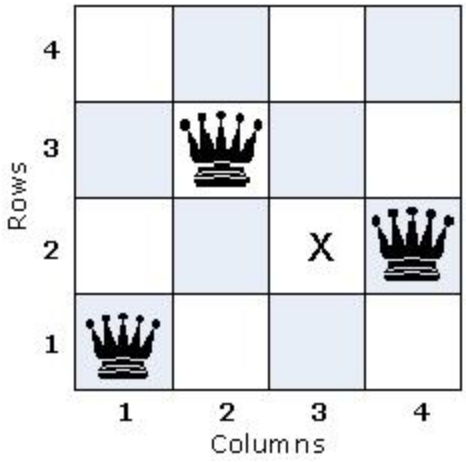
しかし、ここでは、クイーンを 3 行目に置くことができないため、この配置は実行可能な解ではありません。そこで 2 行目の別の列、この場合は図 3 で示すように 4 列目に配置してみます。

図 3



この新しい配置は有効なようです。次の配置候補は、3 行 2 列目(図 4) です。

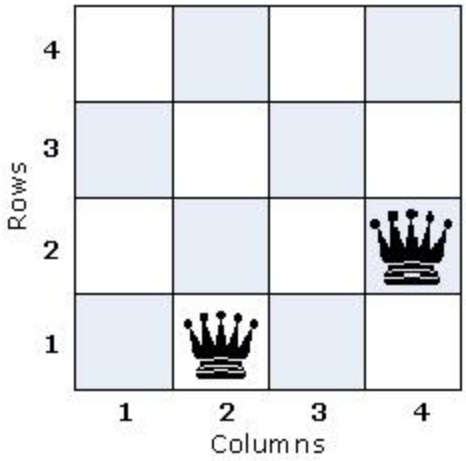
図 4



しかし、この新しい配置では 4 行目の配置候補がありません。

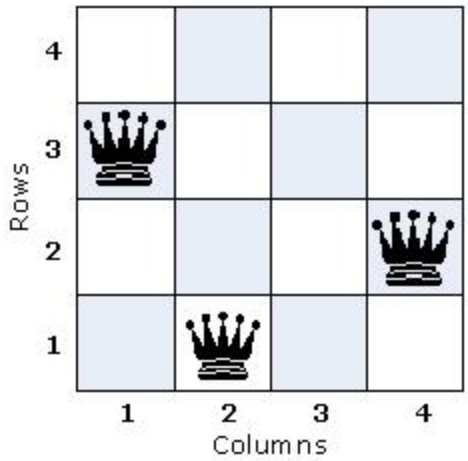
最初に戻り、クイーンを 1 行 2 列目 (図 5) に配置します。前と同じ手法を用いて次のクイーンを 2 行 4 列目に置きます。

図 5



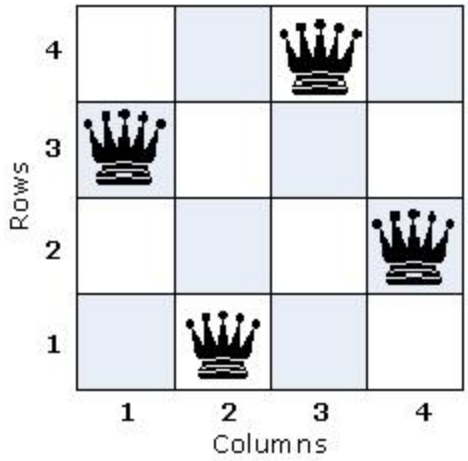
新しい配置では、3 行 1 列目 (図 6) に新しいクイーンを追加することができます。

図 6



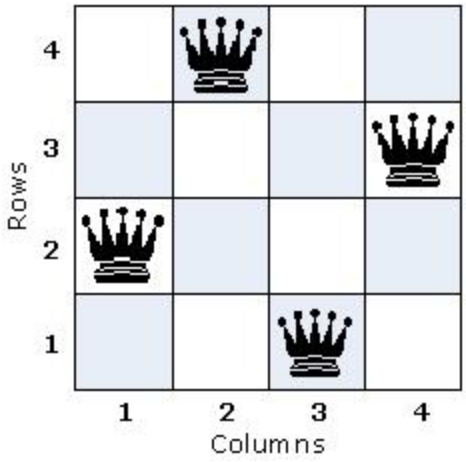
この新しい配置では次のクイーンを 4 行 3 列目 (図 7) に置くことができるため、ついに解の候補が得られたことになります。

図 7



まだ別の解があるかもしれません。1 行 3 列目から開始して、再び配置を一通り行うと、最初の解の鏡像である別の解が見つかります (図 8)。

図 8

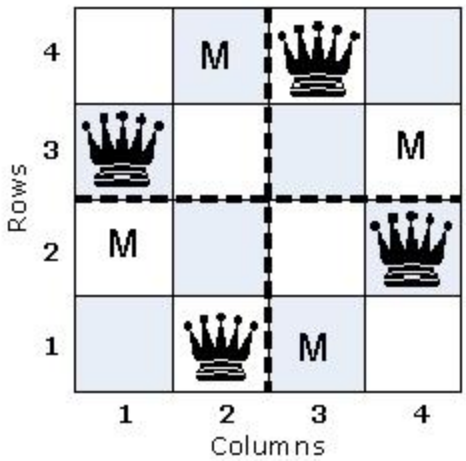


ほかに解の候補を調べてみると、存在しないことが判明します。このため、4 x 4 の盤には 2 個の解があると言えます。この解は、バリエーション解です。では、基本解はいくつあるでしょうか。

この質問に答えるには、対称を考慮する必要があります。点線で盤面を区切り、その点線を介して左右対称の鏡像を作成すると、別の解が浮かび上がります (図 9)。

別の解は対称によって得られるものであるため、基本解は 1 個です。

図 9



解は (n) 値のベクトル (v) (行) で表すことができます。すべての値にクイーンを配置できる列数が含まれています。

最初の 16 マス (4 × 4) の場合では、v[4] = [2 4 1 3] と [3 1 4 2] の解が見つかりました。

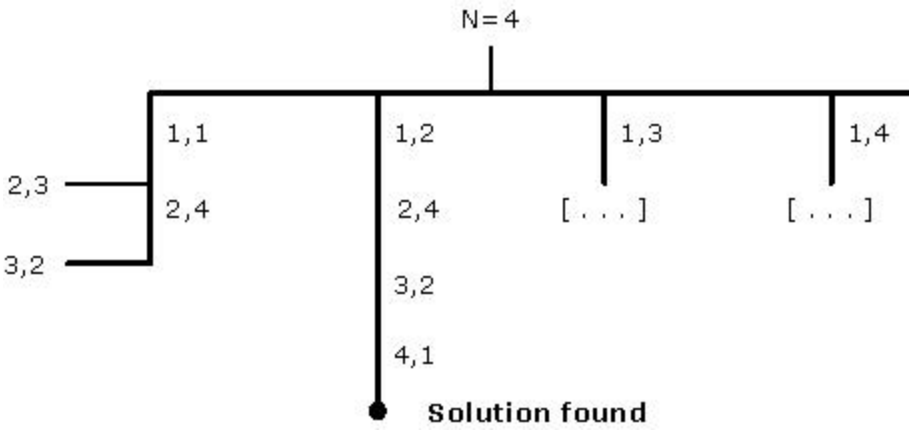
1.2.2. アルゴリズム

問題の解を求める最も明白なアプローチは、1 駒ずつ、反復的に盤面に追加してすべての解候補を見つける方法です。

各列に 1 個ずつクイーンを追加していき、1 番目のクイーンの元の配置を基に解を探すため、比較的簡単に特定することができます。それぞれの解は、ほかの列にあるクイーンから導き出される解とは独立しています。これはツリー検索の例と言えます。このアルゴリズムは、一旦クイーンが盤面に配置されると、検索ツリーの下まで繰り返し適用されるためです。

上記の検索ツリーは、次のようになります。

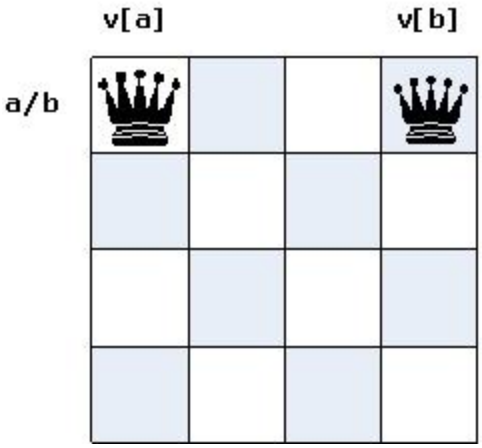
図 10



このアプローチをコーディングするには、クイーン (b) がクイーン (a) を攻撃できる条件を定義する必要があります。

まず、クイーンが同じ行または同じ列 (v[a] = v[b] の場合)、ここでは [4 x x 4] について考えます。

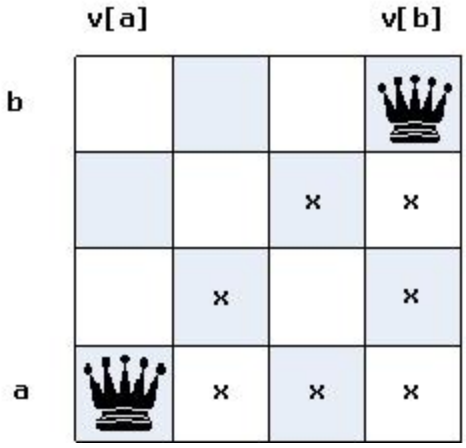
図 11



次に、クイーンが対角線上で攻撃できる条件を定義する必要があります。行カウントと列カウントの差分の絶対値は等しくなければなりません (a < b の場合)。

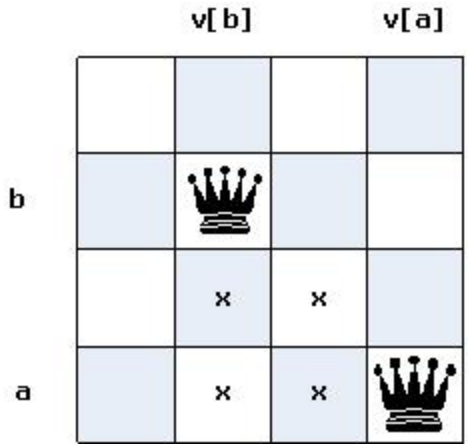
両方のクイーンは、同じ対角線上に位置するときに互いに攻撃することができます。2 つの候補があります: b-a = v[b] - v[a]

図 12



または、b-a = v[a] - v[b]

図 13



2 つの条件を合わせると、b-a = | v[a] -v[b] | となります。結果のアルゴリズムは次の擬似コードのようになります。

```
Read board size: n

Set row to 1
Set v[1] to 0
while ( column > 0) do
[
   Find out a safe place in column where to safely place a queen
   if there is a location
      put v[row] = location
      if [row = n]
         [Write solution vector v[1 ... n]]
      end
      else
      if [row < n]
         go forward one line : row = row + 1
         set v[row] to 0
      end
      else do
      one column back : column = column - 1
      end
   end
]
```

バックトラック法は、段階的に解を構築し、一部の解をそこから派正する解とともに切り捨てることにより、Brute-Force 探索法を改良したものです。

この方法は、エイトクイーン問題に適しています。例えば、何も置いていない盤面に 1 個ずつクイーンを追加していくと、クイーンを追加した時点で競合が発生し、クイーンをさらに追加しても、その競合は解消されないことがわかります。そのため、そのクイーンから派生するすべての解は、テストしなくても切り捨てることができます。

2. ソリューションの並列化

このセクションでは、主要な「演算処理」ルーチンと、その並列化の方法について説明します。基本的な並列分析とアルゴリズムの分解について考え、何を並列化できるかを検証します。その後、各並列ソリューションについて説明します。

各ソリューションの説明の終わりに、そのメソッドのユーザビリティーとパフォーマンスの観点から考えられるトレードオフを考察します。

まず、プログラム中で最も時間を消費している部分を見つける必要があります。これは、並列化を開始する手がかりとなります。プログラム中の hotspot を、プロファイラーを使用して発見します。ここでは、Whatif.intel.com Web サイト (http://whatif.intel.com/、英語) のパフォーマンス・チューニング (PTU) を使用します。ただし、VTune™ パフォーマンス アナライザーのような、ほかのプロファイラーを使用することもできます。

2.1. シリアル「演算処理」関数

サンプルには、シリアルバージョンも含まれています。1 つ目は、擬似コードを C++ へ変換したものです。STL (Standard Template Library) ベクトルを使用して、解ベクトルの情報を保持しています。このバージョンは、バックトラック法で解が切り捨てられるたびに STL ベクトルのサイズが変更されてしまうため、少し非効率的です。2 つ目は、変数ベクトルを固定配列によって置換し、パフォーマンス・ペナルティーを排除するものです。もちろん、これは別の方法でも処理できます。主な変更は、演算処理をカプセル化し、アルゴリズムを個別の関数に置くという、よりモジュール的なアプローチを使用していることです。

当然、単純なアルゴリズムが最も速いとは限りません。BCPL の父 Martin Richards 氏は、2005 年に再帰的ビットパターンのアルゴリズムを使用してクイーン問題の解を発表しています。ここで使用する C++ バージョンは、Richards 氏により発表された MCPL バージョンを直接変換したものです。これらのアルゴリズムについていくつかの補足事項があります。まず、第一にシリアルコードを向上させる方法は、いくらでもあるということです。2 つ目の実装は、スレッド化環境で最も簡単に使用でき、異なるモデルの実装方法を確認することができます。つまり、基本的な並列アルゴリズムを示し

ます。本ガイドは、可能性のあるアルゴリズムの中で最も高速なものを示す目的で提供されているわけではありません。並列モデルの各種の実装を紹介することを目的としていることに注意してください。

3 つ目のアルゴリズムは、並列化の概念を検証する前に、最も速いアルゴリズムを示す目的だけに含まれています。このアルゴリズムはより難解です。

注: バックトラック・ツリー・バージョンは、Martin Richards 氏の論文に基づいています。原文は、http://www.cl.cam.ac.uk/~mr10/ にある Martin Richards の『Backtracking Algorithms in MCPL using Bit Patterns and Recursion』を参照してください。

2.1.1. STL (Standard Template Library) ソリューション

サンプルコードは、`..¥NQueens¥nq-stl¥` 以下にあります。

このサンプルは、上記のアルゴリズムを単純に C++ で表現したもので、解を表すために STL ベクトル・オブジェクトを使用しています。

ベクトル手法の pop_back/push_back は、解ベクトルの展開と折り畳みを行うことができます。この方法で、STL ベクトルにアクセスするのは非効率的なため、次の実装ではベクトルを整数配列に置き換えています。

2.1.2. シリアル・ソリューション

サンプルコードは、`..¥NQueens¥nq-serial¥` 以下にあります。

このサンプルは、N クイーン・アルゴリズムのよりモジュール化されたバージョンです。すべてが solve 関数にラップされます。解ベクトルは単純な整数配列に置き換えられます。

`solve` 関数内で、最初の行の異なる列に対してループする `setQueen()` を呼び出します。このアプローチの長所は、計算が独立しているため、後で並列化を簡単に行えるという点です。N クイーンの演算処理関数の核となるループを以下に示します。

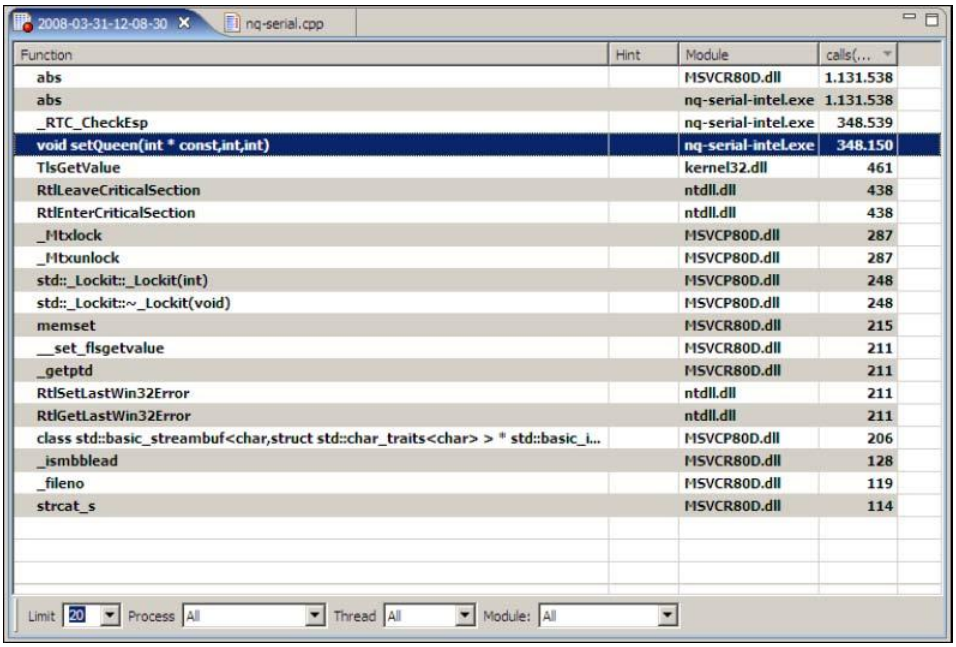
```
void setQueen(int queens[], int row, int col) {
   for(int i=0;i<100;i++) {
      if (queens[i]==col) // 縦の攻撃
         return;
      if (abs(queens[i]-col) == (row-i) ) // 対角線の攻撃
         return;
   }
   queens[row]=col; // 列は OK、クイーンをセットします
   if(row==size-1) {
      nrOfSolutions++;
   }
```

```
    else {
        for(int i=0; i<size; i++) { // 次の行を試行します
            setQueen(queens, row+1, i);
        }
    }
}
```

各列 (i) で、ドライバー関数 `solve()` から再帰的バックトラック・アルゴリズムを開始します。

```
void solve(int queens[]) {
    for(int i=0; i<size; i++) {
    // 最初の行のすべての位置を試行します
    // 各再帰で個別の配列を作成します
        setQueen(queens, 0, i);
    }
}
```

プロファイラーを使用すると、プログラム中で最も時間を消費している場所を見つけることができます。



ほとんどの時間が `abs()` 関数で費やされていることは意外なことではありません。この関数は、2 つのクイーンが同じ対角線上にある場合の計算を行います (アルゴリズムの説明を参照してください)。

コアの `solve` 関数によって、348150 回も呼び出されている `setQueen()` で `abs()` は最もホットな関数です。

2.2. 「演算処理」の並列化

この例は、インテルが提供する並列モデルの深い研究に取って代わるものではありませんが、手始めとして十分な情報が得られるでしょう。シリアルサンプルを本ガイドの手順に従って進めていくと、hotspot を特定することができます。この例は、まさにパレート原理に沿っており、コードの 20% の部分でパフォーマンスの 80% が費やされている多くのアプリケーションに当てはまります。通常、これだけで、並列化の労力全体を抑えることができます。次に、できるだけソリューションを一般化できるよう最も高い抽象化レベルで並列化を開始します。この手法は、アプローチを今後再利用したり、リファクターリング可能にするのに役立ちます。

ここでは、3 つのモジュール (`solve()`、`setQueens()`、`abs()`) を検討します。`solve()` 関数の抽象化カウントが最も高く、abs() 関数は最も低いため、`solve()` 関数を使用します。先ほどのディシジョン・ツリーを見てみると、枝が互いに独立していることがわかります。これは並列化ストラテジーにとっては重要かつ不可欠な条件です。

2.2.1. Windows* 32 スレッド化 API

サンプルコードは、`..¥NQueens¥nq-serial¥` 以下にあります。

N クイーン・アルゴリズムの「クラシック」な並列実装では、Windows 32 API のようなネイティブスレッドを利用しています。API アプローチの主な長所は、C++ を使用することで、その他の言語に比べて、スレッド化をより柔軟により細かく制御できる点です。しかし、ほかのアプローチではランタイムによって処理されるものも含め、このソリューションではプログラマーがすべてのスレッド処理を実装しなければならず、必要なコード量がより多くなります。このようなことから、このアプローチはスレッド化のアセンブリー言語と呼ばれることがあります。アセンブリー言語は柔軟性がありますが、高水準言語のほうがより早く目的を達成し、良い結果、または少なくとも十分な結果が得られます。さらに、作成されるスレッド数に影響を与えるコア数を特定する必要があります。この作業は、プラットフォームに依存しないソリューションを念頭に置いている場合は簡単なことではありません。紙面の都合上、本ガイドではスレッド化 API については言及しませんが、根底にある手法の着想を簡単に紹介します。まず最初に、スレッド・ローカル・ストレージを作成し、ループのどの部分を作業するかなど、スレッド固有の情報を格納します。これは、`struct _thr_params` で行います。

```
// スレッド・ローカル・データ構造には、反復範囲と出力するメッセージが含まれています
struct _thr_params {
    size_t start;
    size_t end;
    int *queens;
    std::string msg;
};
typedef struct _thr_params thr_params;
```

次に、スレッド化するコード部分を特定します。ここでも `setQueens()` モジュールで作業するため、ドライバー関数を実装して、複数のスレッド作業を調整する必要があります。これは、`run_threaded_loop()` で行います。

```
void run_threaded_loop (int num_thr, size_t size, int _queens[]) {
    HANDLE* threads = new HANDLE[num_thr];
    thr_params* params = new thr_params[num_thr];

    for (int i = 0; i < num_thr; ++i) {
        // 各スレッドに等しく行数を与えます
        params[i].start = i * (size/num_thr);
        params[i].end = params[i].start + (size/num_thr);
        params[i].queens = _queens;

        // データ競合を回避するために引数ポインターを
        // 各スレッドのパラメーターのメモリーに渡します
        threads[i] = CreateThread (NULL, 0, run_solve,
        static_cast<void *> (&params[i]), 0, NULL);
    }

    // スレッドを結合します。すべてのスレッドが完了するまで待機します
    WaitForMultipleObjects (num_thr, threads, true, INFINITE);

    // メモリーを解放します
    delete[] params;
    delete[] threads;
}
```

並列で実行したいコードを `run_solve()` という新しい関数にカプセル化します。`run_solve` の各インスタンスは、ループ・インデックスの開始地点と終了地点をそれぞれのローカルに持っています。

```
// ワーカースレッド関数 (各スレッドで並列に実行)
DWORD WINAPI run_solve (void* param)
{
    // このスレッドに渡された引数を取得します。param はポインターです
    // void、static_cast はこれを thr_params へのポインターに変換します
    thr params* params = static cast<thr params*> (param);

    // メッセージを出力します
    for (size_t i = params->start; i < params->end; ++i) {
        // 各再帰で個別の配列を作成します
        setQueen(new int[size], 0, i);
    }
    return 0;
}
```

当然、競合状態を回避するため、正しいプログラミング構造にする必要があります。これは、OpenMP やインテル® TBB のアプローチに非常によく似ています。

```
CRITICAL_SECTION lock;
CRITICAL_SECTION plock;
```

純粋なソース行の数でソリューションを比較すると、スレッド化 API アプローチが最も複雑で、陥りやすい過ちやエラーの可能性がより多く含まれていることが明らかです。ただし、このアプローチは、柔軟性が高く実装における制御力が最もあります。さらに、適切なスレッド数を選択して、コア数を自動で特定する巧みな方法を実装するのも良いでしょう。インテル® TBB と OpenMP では、この作業が自動で行われます。ここで使用したスレッド化 API アプローチは、別のマシンに実行ファイルを移動すると、すぐにオーバーサブスクリプション (コア数よりも多いスレッド数の使用) やアンダーサブスクリプション (コア数よりも少ないスレッド数の使用) を引き起こす可能性があります。コア数よりも多いスレッドが使用されている場合、インテル® スレッド・プロファイラーでは、青いバーで示されます。この青いバーは、オーバーサブスクリプションを意味し、パフォーマンスを低下させます。これは、黄色の同期バーを取り除く作業とともに、ユーザーの課題となります。

2.2.2. __task と __taskcomplete による並列化

サンプルコードは、..¥NQueens¥nq-parexp-intel¥ 以下にあります。

インテル® コンパイラーは、新しく追加された C/C++ 言語拡張を使用して、並列プログラミングを容易にします。このバージョンのコンパイラーには、__taskcomplete、__task、__par、__critical の 4 つのキーワード

が用意されています。これらのキーワードは、文のプリフィックスとして使用されます。キーワードを使用してアプリケーションが並列化の利点を享受するには、コンパイル時に /Qopenmp コンパイラー・オプションを使用する必要があります。コンパイラーは、適切なランタイム・サポート・ライブラリーにリンクします。実際の並列化のレベルは、ランタイムシステムによって管理されます。並列拡張機能は、OpenMP 3.0 ランタイム・ライブラリーを利用しますが、OpenMP プラグマと宣言子を使用することなく、より自然な C または C++ コードを保つことができます。

__par を使用して、solve() 関数の並列化を行うことができます。このプリフィックスを使用することにより、関数を並列処理用に変更します。引数にオーバーラップがないと仮定し、solve() 関数に __par キーワードを追加します。関数を呼び出す方法には変更がなく、計算処理が並列化されます。これは、次のようになります。

```
void solve() {
    __par for(int i=0; i<size; i++) {
        // 最初の行のすべての位置を試行します
        // 各再帰で個別の配列を作成します
        // ここから開始です
        setQueen(new int[size], 0, i);
    }
}
```

サンプルでは、マクロをループの前に配置するだけで、__par 宣言子によってループが並列化されています。プログラムを数回実行すると、それぞれで結果が異なり、__par の使用によってある種のエラーがプログラムで発生していることがわかります。これは、並列プログラミングにおける最も一般的な問題の 1 つで、競合状態と呼ばれます。このような状況では、特定の 1 つのコードブロックで複数のスレッドがそれぞれ動作していて、実行結果は、最初にコードブロックに到達したスレッドに依存します。この例では、2 つのプロセスが有効な解を見つけると、それぞれが nrOfSolutions の値を 1 ずつ増やします。

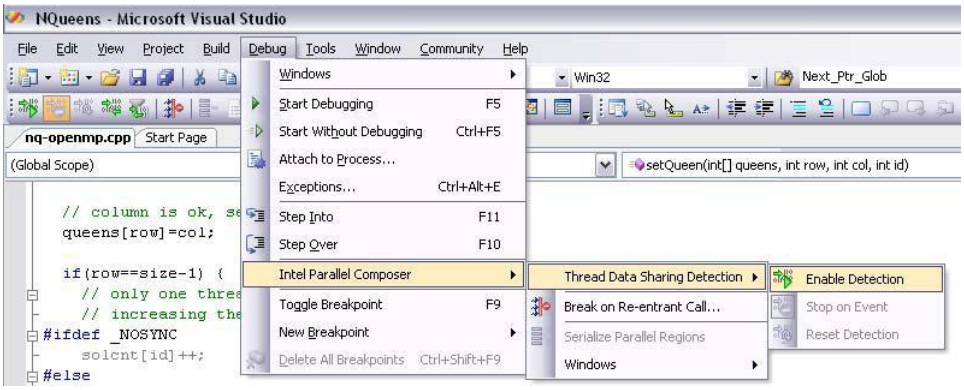
これは、3 つの個別の操作で構成されています。

- nrOfSolutions の値をレジスターにロードする。
- レジスターの値に 1 を追加する。
- レジスターの内容を変数 nrOfSolutions に書き込む。

2 つのプロセスがそれぞれ交互に動作すると、両方のプロセスの終了時点で nrOfSolutions の最終値は初期値に 2 を足したものと等しくなることは明らかです。別の可能性としては、最初のプロセスが 1 番目と 2 番目の操作を終了した後で、最初のプロセスが 2 つ目のプロセスに先を越されることも考えられます。例えば、2 つ目のプロセスが nrOfSolutions の値

を 6 として読み取ると、その値は 7 となり、変数 nrOfSolutions に書き込まれます。最初のプロセスが再開すると、中断したところから操作が開始し、nrOfSolutions の値を 6 と書き込みます。このように、最初の例では、コードは nrOfSolutions の最終値を 7 と生成し、2 つ目の例では、同じコードでも nrOfSolutions の最終値を X と生成します。特定の実行によってどちらの結果が生成されるかは予想できません。この影響は、シリアル・プログラム・フローから並列プログラム・フローに移行したときに顕著に現れます。並列プログラムではすべてのスレッディング・アプローチが共有メモリーを介して通信するからです。

コードの検証によって競合状態を検出することは、複雑なコードでは現実的に不可能です。そこで、この作業に役立つツールを使用します。1 つは、コードをインストルメントして、テストデータを実行し、競合する可能性のある状態を記録するツールです (例: インテル® スレッド・チェッカー)。別のアプローチとしては、ある 1 つのコードパスのデバッグ中にそのような状態をフラグすることです。インテル® Parallel Composer には、Microsoft* Visual Studio デバッガーの拡張版として、スレッド化コード特有の問題をトラッキングできるインテル® Parallel Debugger Extension が含まれています。



この問題に対する解決策としては、**_critical** のような相互排他的実行構造によりソリューションのカウンターを保護することで、アルゴリズムの同時発生的動作によって引き起こされた競合状態を阻止することです。プログラムフローをシリアル化するため、実行速度は多少落ちますが、正しい解を得ることができます。最終的なコードは次のようになります。

```
__critical
{
    // ソリューション・カウンターはアトミックには増えません
    nrOfSolutions++;
}
```

2.2.3. OpenMP 3.0 の使用

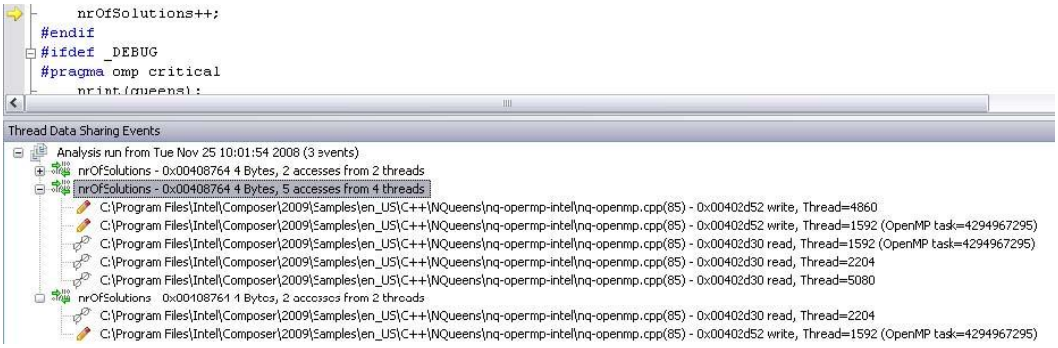
サンプルコードは、..¥NQueens¥nq-openmp-intel¥ 以下にあります。

OpenMP は、移植性を確保したマルチスレッド・アプリケーション開発のための業界スタンダードです。このアプローチは細粒度 (ループレベル) および粗粒度 (関数レベル) のスレッド化に効果的です。OpenMP 宣言子は、シリアル・アプリケーションを並列アプリケーションに変換する簡単で強力な方法を提供し、マルチコアシステムおよび対称型マルチプロセッサー・システムの並列実行から、潜在的な大きなパフォーマンス・ゲインを引き出します。元のソースは変更されずにコンパイルされます。宣言子がコードに挿入されますが、それらについてアクションがない場合は (例えば、共有メモリー並列モードでアプリケーションが実行していないなど)、プログラムは変更されません。共有メモリー並列コンピューターでは、シリアル実行と並列実行の単純比較ができます。宣言子のみがコードに挿入されるため、インクリメンタルにコードを変更することができます。インクリメンタルなコードの変更は、シリアルの一貫性の維持に役立ちます。コードを 1 つのプロセッサーで実行すると、変更されていないソースコードを実行したときと同じ結果が得られます。OpenMP は、複数のプラットフォームとオペレーティング・システムをサポートするシングル・ソースコード・ソリューションです。また、OpenMP ランタイムにより適切なコア数が選択されるため、コア数を特定する必要もありません。

OpenMP が最もよく使用されるループレベルの並列化に加え、最新の OpenMP バージョン 3.0 には新しくタスクレベルの並列化構造が含まれ、関数の並列化が簡単になりました。この例では、前述の並列試行バージョンと同じアプローチを使用しています。違いは、__par マクロが、まったく同じ機能を持つ OpenMP プラグマに置き換わることだけです。

```
void solve() {
    #pragma omp parallel for
    for(int i=0; i<size; i++) {
        // 最初の行のすべての位置を試行します
        // 各再帰で個別の配列を作成します
        // ここから開始です
        setQueen(new int[size], 0, i);
    }
}
```

このプログラムの重要な関数は、solve 関数です。solve は、解が独立しているので (検索ツリーについて参照してください) 並列化が容易です。サンプルでは、OpenMP の #pragma parallel for を使用して、重要なループを並列化しています。インテル® Parallel Debugger Extension [2] を使用することで、ここでも同じ競合状態が発生していることがわかります。



`#pragma omp critical` または `#pragma omp atomic` などの相互排他的実行構造でソリューションのカウンターを保護することで、この共有アクセスを回避します。

```
#pragma omp atomic
   nrOfSolutions++; // ソリューション・カウンターはアトミックには増えません
```

OpenMP には、Win32 スレッドなどの典型的な Windows スレッディング・アプローチにはない、独特な長所があります。このプラグマベースの手法では、並列化にインクリメンタルなアプローチを用いることができます。つまり、コードを最後まで追って必要に応じてスレッド化するのではなく、どの部分、そしてどの hotspot を並列化するかを決定することができます。もう 1 つの長所は、シリアルコードは壊されず、完全な状態が維持されることです。さらに、OpenMP ランタイムが可能な場合はいつでも自動でスレッドプールを行うため、高度なスレッド処理を気にする必要がありません。典型的な Windows スレッディングでは、スレッドプールを使用しないため、各スレッドでスレッド起動時のオーバーヘッドに直面します。これは、多くのアプリケーションで障害になっています。OpenMP は、アプリケーションをどのように、そしてどこをスレッド化するかを明示的にコンパイラーに指示する柔軟で簡単なプラグマ、関数コール、環境変数のセットです。OpenMP 機能を活用すると、シングル・スレッド・プログラミングと比べても、並列プログラミングはそれほど難しいことではありません。

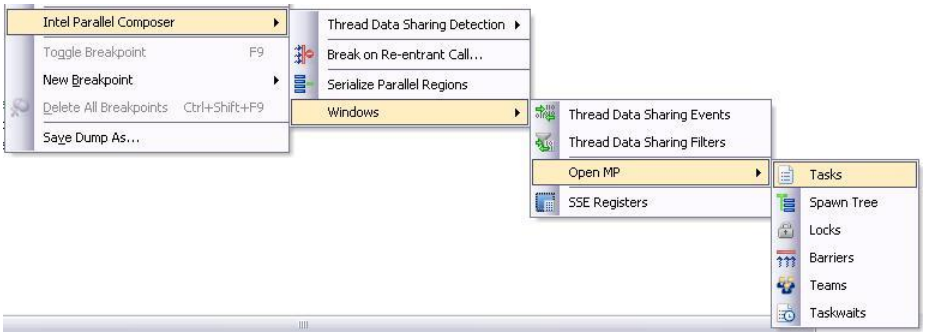
2.2.4. OpenMP 3.0 のタスク・キューイング

サンプルコードは、..¥NQueens¥nq-openmp-taskq¥ 以下にあります。

不規則なパターンの動的なデータ構造体、または再帰など複雑な制御構造体を持つプログラムでは、効率的に並列化するのは困難です。ワークキューイング・モデルを使用すると、OpenMP 2.0 や 2.5 ではできなかった不規則な並列化が可能になります。task プラグマは、囲まれた作業 (タスク) 単位が実行される環境を指定します。タスクブロック内に task プラグマが存在すると、タスクブロックの内側のコードは、タスクに関連付けられている概念キューにキューイングされます。シーケンシャルなセマンティクスを保持するために、タスク完了時に暗黙的なバリアがあります。ユーザーは、タスクブロック間、あるいはタスクブロック内のコードとタスクブロック外のタスクブロック内のコード間で、依存性が存在しないこと、または適切に同期されることを確認する必要があります。このガイドの例では、このコードは次のようになります。

```
#pragma omp parallel
#pragma omp single
{
    for(int i=0; i<size; i++) {
        // 最初の行のすべての位置を試行します
        // 各再帰で個別の配列を作成します
        // ここから開始です
#pragma omp task
        setQueen(new int[size], 0, i);
    }
}
```

このガイドの例では、1 つのタスクキューしか必要ないため、キューを 1 つのスレッドのみで設定する必要があります (omp single)。setQueens は、それぞれが独立しているため、タスクの概念に非常によく合致します。Visual Studio で動作するインテル® Parallel Debugger Extension では、OpenMP のタスク、チーム、ロック、バリア、タスク待機の状態を専用のウィンドウで簡単に調べることができます。コードのスレッド化により発生する問題を無視する場合は、[Serialize parallel regions (並列領域をシリアル化)] をインテル固有のデバッグメニューから選択して、並列領域の実行を再コンパイルせずにシリアル化することもできます。



プログラムや環境で *num_threads* が設定されていたとしても、プログラム内の後続の並列領域は 1 つのスレッドのみで実行されます。インテル® Parallel Debugger Extension [2] についての詳細は、Visual Studio から利用可能なオンラインヘルプを参照してください。

2.2.5. インテル® スレッディング・ビルディング・ブロック (インテル®TBB) による並列化

サンプルコードは、`..¥NQueens¥nq-tbb-intel¥` 以下にあります。

インテル® スレッディング・ビルディング・ブロック (インテル® TBB) は、C++ プログラムの並列化を記述する機能豊富な手法を提供します。インテル® TBB は、マルチコア・プロセッサーのパフォーマンスの活用に役立つライブラリーです。プラットフォームの詳細を抽象化する、より高いレベルのタスクベースの並列化と、パフォーマンスとスケーラビリティーのためのスレッド化メカニズムを示します。また、オブジェクト指向や C++ の汎用フレームワークによく合致します。インテル® TBB は、ランタイムベースのプログラミング・モデルを使用し、ユーザーに標準テンプレート・ライブラリー (STL) と同じようなテンプレート・ライブラリーをベースとした汎用並列アルゴリズムを提供します。インテル® TBB のタスク・スケジューラーがプログラマーに代わって負荷分散を担います。スレッドベースのプログラミングの場合、負荷分散を正しく処理することは簡単ではないため、プログラマーの手を煩わせることが多々あります。プログラムを小さなタスクに分割することによって、インテル® TBB スケジューラーは作業が均等に分散されるようにタスクをスレッドに割り当てます。

テンプレートの扱いに慣れていないプログラマーのために、ここではインテル® TBB の使用について簡単に紹介します。テンプレートは、より汎用的でデータ型に依存しないプログラミングが可能なプログラミング構造を提供します。この汎用アプローチでは、演算子の多重定義と継承を組み合わせ、非常に高いレベルの抽象化を達成することができ、強力な並列設計概念を可能にします。テンプレートは、1 つまたは複数のパラメーター化された型を持つ架構式構造で、関数 (function テンプレート) やクラス (class テンプレート)、そしてテンプレート自体 (template テンプレート) を作成するのに使用できます。マクロと比べてテンプレートの大きな長所は、型セキュリティーであることです。コンパイル時に検証されるため、実行時エラーを防

ぐのに役立ちます。ここでは、2 つの要素の最大値を求める簡単な関数テンプレートの例を紹介します。

```
template <typename T>
T max(T x, T y)
{
   if (x < y)
      return y;
   else
      return x;
}
```

このテンプレートは、正規関数などと呼ばれます (例: max(3, 7))。コンパイラーは、パラメーターを見て型を判断します。そのため、整数だけでなく、浮動小数点、文字列、さらにクラスにも有効です。唯一必要な条件は、比較演算子とコピー・コンストラクターは使用されるデータ型に対して定義される点です。また、インテル® TBB を使用する上で知っておくべきことは、ファンクターがどうのように動作するかということです。C++ のファンクターは、指定されたデータ構造の要素で動作します。ファンクターとは関数として使用されるクラスです。クラスは、関数コール演算子を多重定義することにより定義されるため、operator() メンバー関数が定義されます。簡単なファンクターは次のようになります。

```
#include <list>
#include <algorithm>
#include <iostream>

struct Sum {
   double sum;
   Sum():sum(0.0) {}
   void operator()(const double & d) { sum += d; }
};

std::list<double> myList;
…
Sum mySum = std::for_each(myList.begin(), myList.end(), Sum());
std::cout << "Summe " << mySum.sum << std::endl;
```

クラス sum は、和を求めます。コンストラクターは 0 に初期化されます。operator() は、別の値を追加するように定義されます。クラスは、for_each 文とともに使用され、MyList の各要素に対して operator() を呼び出し、結果は for_each によりが返され、mySum.sum に格納されます。この手法は、既に C++ で使用されており、インテル® TBB によって拡張され、簡単な並列化概念を実現しています。インテル® TBB は、parallel_for、parallel_while、parallel_reduce、pipeline、parallel_sort、parallel_scan のようないくつかの関数テンプレート

を並列コンテナーとともに提供しています。この例では、parallel_for テンプレートを使用しています。このファンクターは、恣意的に選択可能な粒度サイズまでの並列タスクにおける、データの再帰分布を分割するのに使用されています。インターバルの処理には、インテル® TBB クラスの tbb::block_range が使用されています。これは、数値、ポインター、ランダム・アクセス・イタレーターによってインスタンス化することができます。ファンクターは、単一要素ではなく、範囲全体で動作する必要があります。solve ループでは、サンプルは、parallel_for テンプレート関数を実装します。呼び出しの最初のパラメーターは、反復空間について記述する blocked_range オブジェクトです。コンストラクターは、範囲の上下の境界と <grainsize> の 3 つのパラメーターを取ります。parallel_for は、範囲を約 <grainsize> 要素のサブ範囲に分けます。parallel_for 関数の 2 番目のパラメーターは、各サブ範囲に適用される関数オブジェクトです。そのため、これは次のようになります。

```
class SetQueens {
public:
void operator()( const blocked_range<size_t>& r ) const {
    for( size_t i=r.begin(); i!=r.end(); ++i ) {
    // 最初の行のすべての位置を試行します
    // 各再帰で個別の配列を作成します
    // ここから開始です
    setQueen(new int[size], 0, (int)i);
    }
}

void solve() {
    parallel_for(blocked_range<size_t>(0, size, 1), SetQueens() );
}
```

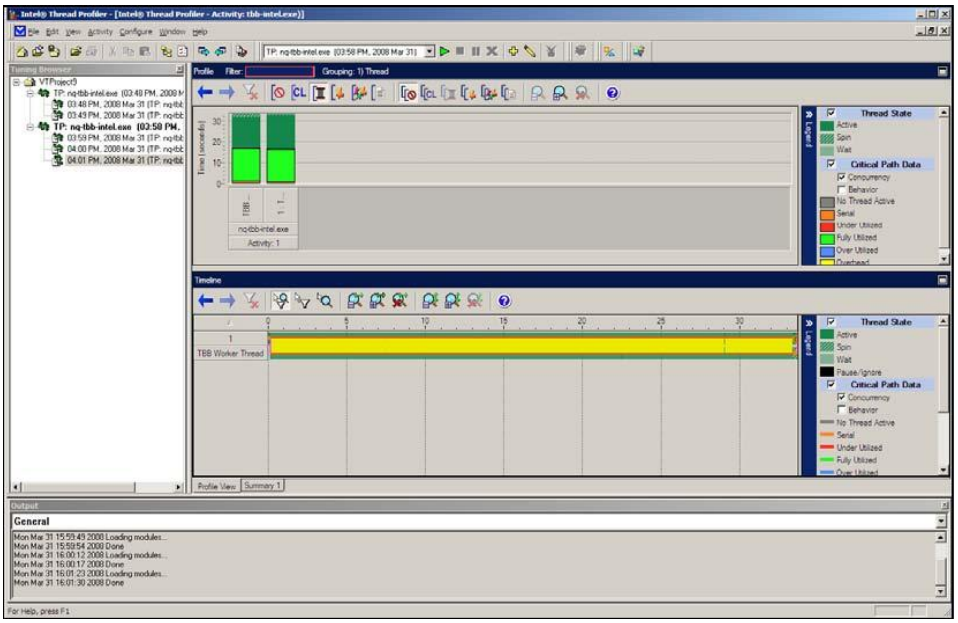
OpenMP サンプルにあるように、このサンプルは相互排他的実行構造の NrOfSolutionsMutexType によって、ソリューション・カウンターを保護しようとします。

```
if(row==size-1) {
{
    // インクリメントは、アトミックではありません。
    // そのため、ここでロックは次のように設定する必要があります。
    NrOfSolutionsMutexType::scoped_lock mylock(NrOfSolutionsMutex);
    nrOfSolutions++;
    }

// ロックを解放するクロージング・ブロックは scoped_lock デコンストラクターを起動します
}
```

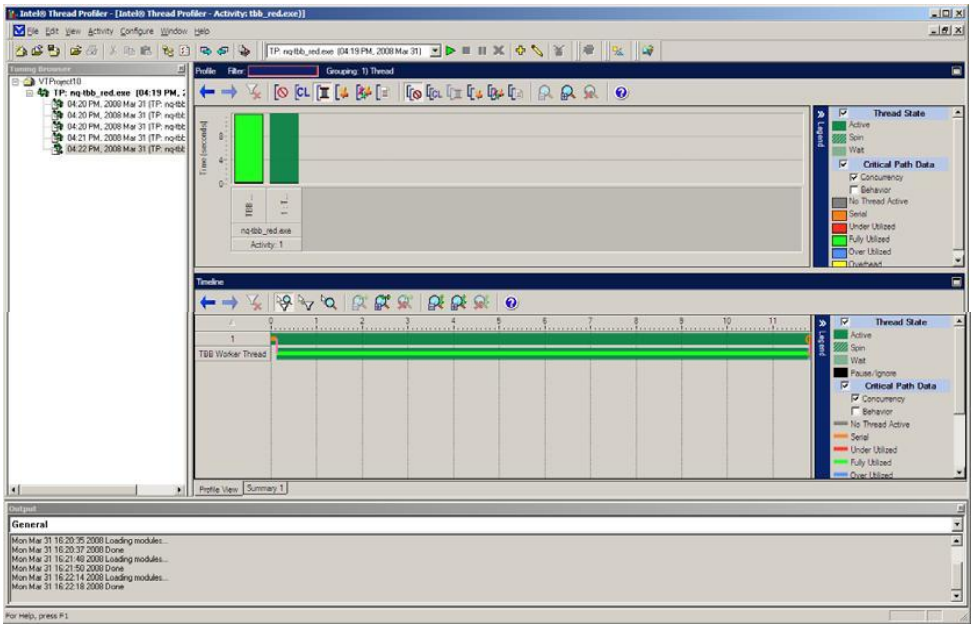
インテル® TBB では、これをより効率的に実装する方法がたくさんあります。例えば、`reduce` 操作を使用する方法ですが、ここでは説明を省きます。ここでの疑問点は、並列化が実際に効率的でスケーラブルであるかどうかということです。1 つの方法として、ランタイムを検証することがあげられます。それをスレッドに分割して、シリアルバージョンの場合と比較して速度向上を測ります。

このアプローチの短所は、並列問題に対する洞察を得られないことです。インテル® スレッド・プロファイラーでは、並列フローグラフの実際の並列実行に対して興味深い洞察を得ることができます。このツールは、取り上げたすべてのアプローチに有効です。インテル® TBB ランタイムのビューを次に示します。



2 つの緑色のバーは、ソリューションの均整はとれているものの、ソリューションの数をカウントするグローバル変数へのアクセスの同期により、多くの同期オーバーヘッドが生じていることを示しています。すべての並列化アプローチを評価し、シリアル化が引き起こす速度の低下を回避することによって、使用したアプローチでパフォーマンスが最大限に引き出されているかを確認する必要があります。(プログラムのシリアル部分が並列パフォーマンスに与える影響については、アムダールの法則を参照してください。) このサンプルガイドの例では、不要な同期化を取り除く非常にわかりやすい方法を示します。

ソリューション・カウンターに数が加算されるたびに、このアクセスを同期化しなければなりません。これは、すべてのスレッドのグローバル変数であるためです。この同期化を回避するために、ローカル・ソリューション変数を各スレッドに実装する必要があります。プログラムの最後にローカル・ソリューション変数を合計してグローバル変数にします。結果のコード実行は、次のようになります。



2.2.6. インテル® TBB でのラムダ 関数の使用

サンプルコードは、..¥NQueens¥nq-tbb-lambda¥ 以下にあります。

ラムダ関数は、C++ の次期標準、C++0x の改訂案に含まれています。この追加により、STL/インテル® TBB アルゴリズムは、かつてないほど便利な指標として使用できます。インテル® コンパイラーは、ラムダ関数を実装する最初の C++ コンパイラーです。ラムダ抽象は一般的に無名関数で、名前のないコードの集まりです。ラムダ関数をクロージャーとペアで使用し、コードとスコープを組み合わせることで、強力な概念を提供します。この構造の特性は、スコープをコードブロックと結びつけて渡すことができ、スコープは結び付けられている特定のブロックによって変更されないことです。標準では、(auto の) 「キャプチャー」という用語が使用され、auto は値や参照でキャプチャーすることができます。クロージャーは、関数定義を原文どおり含むバインディング形式によって確立されるバインディングの値を参照、変更することのできる関数です。ラムダ構造は、C++ の関数オブジェクトあるいは C の関数ポインターとほぼ同じです。クロージャーは、関数のポインターおよび名前/変数のバインディング・セットへのポインターとみなすこともできます。最終的には、ラムダ関数をクロージャーとともに関数オブジェクトと関数ポインター周りの構文糖とみなすことができます。この構文糖は、関数オブジェクトやラムダを使いたいときにすぐに記述できる便利な方法です。

このセクションでは、C++ ラムダ式の使用について説明します。インテルが提供するラムダ式を使用するには、コマンドライン・オプションの /Qstd で c++0x を要求する必要があります。つまり、/Qstd=c++0X と指定します。コンパイラーは、ラムダ式の評価に基づいて匿名関数 (名前のない関数) オブジェクトを作成します。ラムダ式によって作成されるこの関数オブジェク

トは、オブジェクトが作成されたブロックよりも長く持続することがあります。関数オブジェクトが使用される前に破棄されてしまう変数を使用していないかを確認しなければなりません。

次の例は、N クイーンのサンプルで、ラムダ式によって作成された関数オブジェクトがどのように見えるかを示しています。C++ とインテル® TBB の統合をより密にすることで、ラムダ関数とクロージャーを使用して、コードをパラメーターとして渡し、ファンクター `operator()` の概念を簡単に表現することができます。

```
void solve() {
    parallel_for(blocked_range<size_t>(0, size, 1),
        [](const blocked range<int> &r){
        for (int i = r.begin(); i != r.end(); ++i) {
            setQueen(new int[size], 0, (int)i);
        };
    });
}
```

3. 各種手法の比較

最後に、API ベースのスレッド化手法 (Windows OS 上の Win32 マルチスレッディング API および Linux* OS 上の Pthreads ライブラリー)、OpenMP、インテル® TBB の各種の並列化アプローチの要約を示します。これらのアプローチは、抽象化の手法、制御の手法、簡潔さの手法により特徴付けることができます。このサンプルガイドの例では、柔軟性が多少損なわれる代わりに、コードへの影響をより少なくしてアプリケーションに並列化を導入する方法を示してきました。この方法により、生産性を高めることができます。Windows 32 API は、並列化のアセンブリー言語です。一方、OpenMP とインテル® TBB は、並列化の高水準言語です。必要に応じて、アプリケーションに最良の手法を決定してください。次に、各手法の主な特性を説明します。

API ベースのモデルは非常に一般的であり、プログラマーは手動で並列タスクをスレッドにマップする必要があります。スレッド間には明示的な親子関係はありません。すべてのスレッドが対等です。このようなモデルは、スレッドの作成、管理、同期におけるすべての低レベルの局面で制御能力をプログラマーに与えます。この柔軟性が、ライブラリー・ベースのスレッド化手法の鍵となる長所です。柔軟性の代償は、著しいコードの変更と明らかに大量のコーディングです。並列タスクは、スレッドにマップできる関数にカプセル化しなければなりません。また、重要な点は、ほとんどのスレッド化 API が難解な呼び出し規則を使用しており、1 つの引数しか受け付けないことです。そのため、関数のプロトタイプとデータ構造の変更がしばしば必要になり、これにより、プログラム設計における抽象化が損なわれるので、オブジェクト指向の C++ アプローチよりも C に適しているでしょう。

OpenMP、コンパイラー・ベースのスレッド化手法は、根底のスレッド・ライブラリーとの高レベルなインターフェイスを提供します。OpenMP では、プログラマーはプラグマ (Fortran の場合は宣言子) を使用して、コンパイラーに並列化を指示します。これによってコンパイラーが詳細な処理を行い、明示的なスレッド化手法における煩雑な作業を排除することができます。並列化に対してインクリメンタルなアプローチをとるため、アプリケーションのシリアル構造は完全な状態が維持され、著しいソースコードの変更は必要ありません。OpenMP 非対応コンパイラーは、単純にプラグマを無視して、根底のシリアルコードをそのまま残します。ただし、スレッドに対して細かく調整する制御力は損なわれます。また、OpenMP は、スレッドの優先度の設定やイベントベースまたはプロセス間の同期化を実行する方法も提供します。OpenMP は、スレッド間の明示的なマスターワーカーの関係に基づく fork-join スレッド化モデルです。このため、OpenMP が適合する問題の範囲は限定されます。一般に、OpenMP はデータ並列化の表現に最適で、明示的なスレッド化 API 手法は機能分割に最適です。OpenMP は、ループ構造や C コードで最適に動作することはよく知られていますが、C++ に対しては特定のサポートがありません。タスク構造が含まれた OpenMP 3.0 では、ループや再帰構造、そして、大きな向上である関数レベルの並列化などの不規則な構造に対するサポートが追加され、OpenMP が拡張されています。

インテル® スレッディング・ビルディング・ブロック・ライブラリーは STL (Standard Template Library) のような標準 C++ コードを使用して、汎用のスケーラブルな並列プログラミングをサポートしています。特定の言語やコンパイラーは不要です。抽象的でさらに汎用的なオブジェクト指向アプローチに適合する柔軟で高レベルな並列化アプローチが必要な場合は、インテル® TBB が適切かつ唯一の選択肢です。インテル® TBB は共通の並列反復パターンにテンプレートを使用し、入れ子の並列化によりスケーラブルなデータ並列プログラミングをサポートします。API アプローチと比較すると、スレッドではなくタスクを指定し、インテル® TBB ランタイムを使用して、効率的な方法でライブラリーによりタスクをスレッドにマップします。インテル® TBB スケジューラーは、スケジューリングに対してはシングルの自動的な分割統治法を優先します。スケジューラーは、ロードされたコアから休止状態のコアにタスクを移動するタスクスチールを実装します。OpenMP と比較すると、インテル® TBB に実装されている汎用アプローチでは、ユーザーはビルトイン型に限定されない、ユーザー定義の並列化構造で作業することが可能です。最後に、インテル® TBB とラムダ関数を組み合わせることで、必要なコードの再配置の量を抑えることができ、並列アプローチに新たな抽象化レベルを追加することができます。

`__task`/`__taskcomplete` 拡張子は、すぐにスレッド化を行いたい初心者ユーザーでも C および C++ で使用することができるプロトタイプです。OpenMP には、この単純なタスク機能以外にも多くの機能があり、計算集約型のアルゴリズム志向です。これらの新しいタスク機能は、すばやく、そして簡単な非同期プログラミングの入口となることでしょう。

4. まとめ

インテル® Parallel Composer は、C++ アプリケーションで並列化を表現するいくつかの新しい一意な方法を提供します。それぞれの方法には、使用目的に応じた長所があります。本ガイドで検証しているように、簡単な変更を加えて OpenMP や並列の開始/終了、インテル® TBB を実装するだけで、マルチスレッディングを通して大幅なパフォーマンス・ゲインやスケーリングを実現することができます。

多くのアルゴリズムには、シリアル実行から得られる最適化が含まれていますが、並列化を阻害する依存性を発生させてしまいます。そのような依存性は、前述のアプローチを活用し、簡単な変換によって排除できることが多々あります。スレッド作成によって発生するオーバーヘッドを最小限に抑えるため、適切なスレッド数を選択することが重要です。スレッドを作成し過ぎると、システムのオーバーヘッドの増加、粒度の低下、ロック競合の増加など、多くの理由によりパフォーマンスに悪影響を与えます。スレッド化アプリケーションの実行中の競合状態を回避するには、1 つのスレッドだけが共有リソースにアクセスし、状態を変更できるよう共有リソースに対する相互排他が必要です。共有リソースとは、データ構造またはアドレス空間のメモリーなどです。同期化のオーバーヘッドを最小限にすることは、アプリケーション・パフォーマンスには必要不可欠です。

インテル® Parallel Composer で提供されているサンプルコードを使用して、検証してみてください。詳細は、製品ドキュメントまたは製品の Web サイトを参照してください: http://www.intel.com/software/products/ (英語)

4.1. 参考資料

[1] Hoffman, E.J., Loessi, J.C. and Moore, R.C. (1969): Constructions for the Solution of the m Queens Problem, Mathematics Magazine, p. 66-72.

[2] Robert Mueller-Albrecht (2008): Intel® Parallel Debugger Extension http://software.intel.com/en-us/articles/parallel-debugger-extension.