マルチコアシステムの並列パフォーマンス向けに Fortran アプリケーションをスレッド化

この記事は、インテル® デベロッパー・ゾーンに公開されている「Threading Fortran Applications for Parallel Performance on Multi-Core Systems」の日本語参考訳です。

マルチコアシステムの並列パフォーマンス向けに Fortran アプリケーションをスレッド化

今日のほとんどのプロセッサーは複数のコアを搭載しており、今後は一度に 1 命令を処理するコアや SIMD (Single Instruction, Multiple Data) 命令で同時に複数を処理するコアの数が増加することで、パフォーマンスがさらに向上ことが期待されます。パフォーマンスが重要なアプリケーションは、この機会を逃すとすぐに取り残されてしまうでしょう。この記事では、既存のシリアル Fortran アプリケーションが、複数のコアを搭載した単一の共有メモリーシステムでこの可能性を利用する方法を示します。ここでは、データレイアウト、スレッドセーフ、パフォーマンス、デバッグなどの問題に対応します。ここで使用するインテル® Fortran コンパイラーとソフトウェア・ツールは、スケーラビリティーに優れた強固な並列アプリケーションの作成を支援します。

並列処理レベル

アプリケーションは、複数のレベルで並列処理を利用できます。最も基本的なものはプロセッサー・レベルであり、これは通常、命令レベルの並列処理 (ILP) と呼ばれます。コンパイラーとプロセッサーが連携して同時に実行できる可能性を探ります。高度なスキルを備えた開発者は、アプリケーションのパフォーマンスを最大限に引き出すため、ILP を考慮することができます。ベクトル化は特殊な ILP であり、コンパイラーがプロセッサーのハードウェア命令セットを利用して、複数のデータ値に対して同時に実行できる単一の命令を特定します。これは、SIMD 処理と呼ばれます。インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) とインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) 命令セット・ファミリーをサポートするインテル® プロセッサーは、SIMD 処理に対応しています。

並列処理は、共有メモリーを使用してスレッドレベルでも実装できます。開発者が対応するコンパイラー・オプションを指定すると、最近の Fortran コンパイラーは可能な範囲でスレッド並列処理を特定して、並列化された DO ループとされなかった DO ループを示すレポートを生成できます。さらなる並列性を引き出すため、開発者は POSIX* スレッド (Pthreads) を使用したり、ソースに OpenMP* スレッド・ディレクティブを挿入してコンパイラーに明示的に並列化を指示できます。OpenMP* ディレクティブを使用するほうが簡単です。

分散メモリー型並列処理は、単一のプログラムがコンピューター・ネットワーク上で複数の処理を同時に実行できるようにします。メッセージ・パッシング・インターフェイス (MPI) を使用するこの並列処理レベルは、自動的には実装されません。開発者が、コンピューター間のデータ移動を行うルーチンの呼び出しを追加する必要があります。Fortran 2008 標準規格の Co-Array Fortran (CAF) は、分散メモリー型並列処理をサポートしており、コンパイラーがコンピューター間のデータ移動を考慮するため、開発者がルーチンの呼び出しを追加する必要はありません。

これらの並列処理はそれぞれ独立しており、組み合わせて利用することができます。例えば、OpenMP* ディレクティブを使用するアプリケーションは、ILP だけでなく、ベクトル化によってパフォーマンスを向上できます。すべての開発者にとって夢のようなアプリケーションとは、驚異的な並列性を備えたアプリケーションです。個々のタスクを識別しやすく、タスクを実行するスレッド間の通信や調整がわずかで済みます。

トップに戻る

配列処理の導入方法

インテル® MKL

プログラムを並列化する前に、スレッド化ライブラリーの利用を検討すると良いでしょう。スレッド化ライブラリーを呼び出すほうが、自身でスレッド化するよりもずっと簡単です。インテル® マス・カーネル・ライブラリー (インテル® MKL) には、インテルのマルチコア・アーキテクチャーの利点を引き出すため、OpenMP* を使用して実装された多くの標準ライブラリーが含まれています。BLAS (Basic Linear Algebra Subprograms) レベル 1、2、および 3、LAPACK (Linear Algebra Package)、スパース BLAS、高速フーリエ変換 (FFT)、直接法スパースソルバー、ベクトル演算、および乱数関数などを利用できます。詳細は、こちら (英語) を参照してください。

トップに戻る

ベクトル化

ベクトル化に取り組む際に役立つ情報は、ベクトル化コードブックを参照してください。

トップに戻る

POSIX* スレッド

POSIX* スレッド (Pthreads) は言語に依存せず、主にタスクレベルの並列処理に使用されます。Pthreads は、コードの記述とデバッグが困難です。Fortran から Pthreads API への直接インターフェイスはありません。C で記述されたラッパーを Fortran で呼び出さなければならないことも複雑な要因の 1 つです。

トップに戻る

自動並列化

インテル® Fortran コンパイラーは、単純なループを自動でスレッド化できます。-parallel -O2 (Windows* では /Qparallel /O2) オプションを指定してコードをコンパイルします。各ループが並列化されたかどうかを示すレポートを生成すると非常に便利です。コンパイラーがループを並列化しなかった場合、その理由が示されます。レポートを生成するには、-qopt-report-phase=par (Windows* では /Qopt-report-phase:par) オプションを追加します。

-par-threshold[n] (Windows* では /Qpar-threshhold[[:]n]) (デフォルトは n=100) を指定することで、並列処理のコストモデルのチューニングを行うことができます。n=100 にすると、コンパイラーはパフォーマンス向上の可能性が高い場合のみスレッド化 (並列化) します。n=0 にすると、パフォーマンス予測に関係なく、安全な場合は並列化します (テストに役立ちます)。

トップに戻る

自動並列化の条件

今日のインテル® コンパイラーは、-parallel -O2 (Windows* では /Qparallel /O2) オプションが指定されると、単純なループを自動で並列化します。ループを自動並列化するための条件は、ループの自動ベクトル化の条件と似ています。

DO ループは、ループへのジャンプまたはループからのジャンプを含まない単純なループでなければなりません。ループの反復回数は、実行時に判明している必要があります。DO WHILE ループは自動並列化できません。

ループの各反復は互いに独立している必要があり、次のようなデータ依存関係の可能性を示す構文があってはなりません。

```
X(I+1) = Y(I+1) + X(I)
```

コンパイラーによってインライン展開可能な関数呼び出しを含む単純なループは自動ベクトル化できます。

配列ポインターと可能なエイリアシング (異なるポインターによる同じ配列要素へのアクセス) に注意してください。コンパイラーは、オーバーラップの可能性があるメモリー位置に対して、リスクを取らず保守的な判断を下します。

コンパイラーは、総和などのリダクション操作を認識して並列化します。浮動小数点の総和操作を並列化すると、丸めの差異によって結果が大きく異なる可能性があることに注意してください。

並列ループの開始と終了にはオーバーヘッドが伴うため、並列化によってもたらされる利点がそれを上回るように十分なワーク量が必要です。

DO ループは、自動並列化および自動ベクトル化される可能性があります。以下の行列乗算サブルーチンが良い例です。コンパイラーが行ったことがコメントで示されています。詳細は、最適化レポートで確認できます。

```
subroutine matmul(a,b,c,n)
real(8) a(n,n),b(n,n),c(n,n)
c=0.d0
do i=1,n          ! Outer loop is parallelized.
   do j=1,n       ! inner loops are interchanged
      do k=1,n    ! new inner loop is vectorized
         c(j,i)=c(j,i)+a(k,i)*b(j,k)
      enddo
   enddo
enddo
end

$ ifort -O2 -parallel -qopt-report -c matmul.f90
… レポートの抜粋 …
ループの開始 matmul.f90(4,1)
   remark #25444: ループの入れ子の交換: ( 1 2 3 ) --> ( 1 3 2 )
   remark #17109: ループが自動並列化されました。
   remark #15542: ループはベクトル化されませんでした: 内部ループがすでにベクトル化されています。

   ループの開始 matmul.f90(6,7)
      remark #15542: ループはベクトル化されませんでした: 内部ループがすでにベクトル化されています

      LOOP BEGIN at matmul.f90(5,4)
      <ベクトル化のピールループ>
      LOOP END

      LOOP BEGIN at matmul.f90(5,4)
         remark #15301: 変換されたループがベクトル化されました。
      LOOP END

      LOOP BEGIN at matmul.f90(5,4)
      <代替アライメントでベクトル化されたループ>
      LOOP END

      LOOP BEGIN at matmul.f90(5,4)
      <ベクトル化の剰余ループ>
         remark #15335: 剰余ループ はベクトル化されませんでした: ベクトル化は可能ですが非効率です。オーバーライドするには vector always ディレクティブまたは -vec-threshold0 を使用してください。
      ループの終了
   ループの終了
```

ループの終了
…レポートの抜粋…

トップに戻る

OpenMP*

さらに並列性を引き出すため、プログラムに OpenMP* ディレクティブを追加できます。OpenMP* は、SIMD を含むデータ並列処理を容易にし、タスク並列処理もサポートします。OpenMP* は標準 API であるため高い移植性を持ちます。インテルは、OpenMP* ARB (Architecture Review Board) (英語) の最新の標準規格を実装するように努めています。OpenMP* ディレクティブを利用すると、シーケンシャル・プログラムを実行可能な状態で残すことができるため、プログラムの 2 つのバージョンを保持する必要がありません。プログラムのパフォーマンスに重大な影響を与える部分を OpenMP* ディレクティブで囲んで、インクリメンタルな並列化が可能です。

OpenMP* は、fork-join 並列処理モデルで動作します。必要に応じて、マスタースレッドがスレッドをスポーンします (つまり、シーケンシャル・プログラムが並列プログラムになります)。タスクが完了すると、スレッドはスレッドプールに戻され、プログラムの次の並列実行領域で利用できます。

並列化を行う場合、最初に、プログラムが最も時間を費やしている場所を特定します。インテル® Advisor やインテル® Vtune™ Amplifier を使用して必要な情報を得ることができます。期待されるスピードアップにはアムダールの法則が適用されます。つまり、プログラムのシリアル領域によってスピードアップが制限されます。

最高のパフォーマンスを達成するため、粗粒度の並列処理について高レベルで調査します。つまり、入れ子構造のループの外側のループ、最も遅い可変グリッド座標、高レベルのドライバールーチンなどを確認します。高レベルの並列処理は、スレッドの開始と終了のオーバーヘッドを軽減し、データの局所性と各スレッドによるデータの再利用を向上します。反復間にデータ依存関係があるループは、並列化の候補には適していません。

データ並列処理の領域を調査します。スレッド間で計算のバランスを取り、より多くのコアを使用するほうが簡単です。

以下に示す電荷の均一な二乗分布による平面内の一連の点における静電ポテンシャルを計算する、数値積分アルゴリズムの基本構造を調査します。これは、本質的に二乗分布上での 2 次元 (2D) 積分です。

```
DO Square_charge loops over points
   DO TwoD_int integrate over y
      DO Trap_int integrate over x
         sum = sum + Func(calculates 1/r potential)
```

概念的に、このアルゴリズムを並列化する 1 つの方法は、Func() をインライン展開してから、x のループをベクトル化します。そして、y のループまたは外側の points のループをスレッド化します。

トップに戻る

OpenMP*: スレッドの相互作用

OpenMP* は共有メモリーモデルであり、スレッドは変数を共有することで通信します。スレッド間で共有される変数とスレッドごとに個別のコピーが必要な変数を識別することが重要です。Fortran 開発者向けの保守を容易にするコーディングのヒントは、共有データを MODULES または COMMON ブロックのグローバル変数として明示的に宣言して、スレッド・プライベート・データをローカルまたは自動変数にすることです。動的に割り当てられるデータ (malloc または ALLOCATE) を使用できます。共有データはアプリケーションのシリアル領域で割り当てます。並列領域内で割り当てられたデータはそのスレッドでのみ利用できます。データは THREADPRIVATE として宣言することもできます。

スレッド間の意図しないデータ共有は、スレッドのスケジュールが異なるとプログラムの結果が変わったり、競合状態を引き起こす可能性があります。競合状態は同期によって排除できますが、パフォーマンス上のペナルティーを伴います。競合状態を排除する最良の方法は、データのアクセス方法を変更することです。各スレッドには個別のプライベート・スタックがありますが、ヒープはすべてのスレッドによって共有されます。ヒープ上のデータをスレッドセーフに保つには、ロックが必要となりコストがかかります。

明示的な並列領域は、自己文書化コードです。!$OMP PARALLEL と !$OMP END PARALLEL で囲まれているか、!$OMP PARALLEL DO のようなワークシェア構造の場合は対応する ENDDO で終わります。並列領域内から呼び出される関数とサブルーチンは、OpenMP* ディレクティブを含んでいないことがあるため識別が困難ですが、スレッドセーフにする必要があります。

OpenMP* 標準で定義されているように、並列領域内では、ループ・インデックスを除くすべてのデータはデフォルトで共有になります。ローカルでのみ使用され共有されないデータは、PRIVATE として宣言する必要があります。グローバルデータは、THREADPRIVATE として宣言することができます。別のコーディングのヒントとして、共有領域では共有またはプライベートを指定して各変数を宣言することで、コードが自己文書化されます。

通常、並列領域内の初期値は未定義です。FIRSTPRIVATE を使用して変数をループに入った時点の値に初期化します。グローバル変数の場合は、COPYIN を使用して必要なデータを設定します。

逆に、並列領域の終了後、最終値は未定義です。LASTPRIVATE を使用すると、並列領域の最後の値になります。

トップに戻る

スレッドセーフ

スレッドセーフな関数は、複数のスレッドから同時に呼び出されても正しい結果を生成できます。潜在的な競合は、保護 (同期) するか、回避 (プライベート化によって) する必要があります。

先に進む前に、データに関する 2 つの概念を定義する必要があります。

1. 「スタティック・ローカル・データ」はすべてのスレッドで共有されます。つまり、すべてのスレッドが同じデータ位置にアクセスする可能性があります。これは潜在的に危険です。
2. 「自動データ」は各スレッド間で独立しています。スレッドごとに個別のスタックのコピーを保持します。

インテル® Fortran コンパイラーでコンパイルしてシリアルに実行する場合、デフォルトではローカルスカラー変数は自動になり、ローカル配列はスタティックになります。ただし、-qopenmp を指定してコンパイルすると、ローカル変数はデフォルトで自動になります。これは、-auto を指定してコンパイルした場合と同じです。この場合、最大スタックサイズを増やす必要があるかもしれません。スタックサイズが小さすぎると、実行時にセグメンテーション・フォルトが発生します。

トップに戻る

関数またはサブルーチンをスレッドセーフにする

関数をスレッドセーフにするには 2 つのオプションがあります: コマンドラインにコンパイラー・オプションを追加する、またはソースコードに分類を追加する。

コマンドラインを使用する場合、-qopenmp または -auto (Windows* では /Qopenmp または /auto) オプションを追加します。-auto のほうがシリアル処理の最適化への影響が少ない可能性があります。

ソースで必要な変数定義に AUTOMATIC キーワードを追加します。ただし、これはコンパイラーにより生成される一時変数には適用されません。コマンドラインを使用せずにコードをスレッドセーフにする最良の方法は、関数またはサブルーチン全体を RECURSIVE として宣言することです。これにより、コンパイラーにより生成されるコードを含む関数全体がカバーされます。

どちらの方法を使用する場合であっても、-save コンパイラー・オプションや SAVE キーワードは使用しないでください。これらはすべての変数をスタティックにします。また、スレッドセーフ・ルーチンのデータは、Fortran の DATA 文で初期化しないでください。変数がスタティックになります。

書き込みが同期されていない限り、グローバル変数への書き込みは行わないようにします。

OpenMP* には、並列に実行されると安全ではない可能性がある操作を保護するため、さまざまな同期構造が用意されています。一度に 1 つのスレッドのみがコードブロックを通過できるように、必要に応じてソースに次のコードを追加します。

```
!$OMP CRITICAL
- または -
!$OMP SINGLE
```

1 つの文のみを非同期に実行する場合は、!$OMP ATOMIC ディレクティブが役立ちます。

もう 1 つの良く使用される OpenMP 節は REDUCTION です、開発者は、いくつかの演算子を使用できますが、REDUCTION 節の変数は共有でなければなりません。コンパイラーが、スレッドと必要な同期の管理を行います。

トップに戻る

スレッドセーフ・ライブラリー

インテル® マス・カーネル・ライブラリー (インテル® MKL) のシーケンシャル・バージョンとマルチスレッド・バージョンは、どちらもスレッドセーフです。マルチスレッド・バージョンは並列領域から呼び出すことができます

が、使用するスレッド数に注意が必要です。スレッド数が多すぎると、プロセッサーがオーバーサブスクリプションになり、パフォーマンスが低下します。

Fortran メインプログラムを -qopenmp (Windows* では /Qopenmp) でコンパイルすると、自動的にスレッドセーフ・バージョンがリンクされます。

Fortran ルーチンを使用する C/C++ メインプログラムの場合は、-qopenmp (Windows* では /Qopenmp) でコンパイルして、FOR_RTL_INIT ルーチンを呼び出して Fortran ランタイム環境を初期化します。

トップに戻る

パフォーマンスに関する考察

最高のアプリケーション・パフォーマンスを達成するには、関数とサブルーチンのインライン展開や内部ループのベクトル化などによって最適化されたシリアルコードを使用します。

解析ツールを使用して時間が費やされている場所 (ホットスポット) を確認します。そして、ホットスポットに十分な並列ワークがあるかどうかを判断します。

読み取り専用でない限り、スレッド間のデータ共有を最小限に抑えます。スレッドの同期にかかる時間は、パフォーマンスを低下させる可能性があります。

キャッシュラインのフォルス・シェアリングはパフォーマンスを低下させます。次のコード例について考えてみます。

```
!$OMP parallel do
  do i=1,nthreads
     do j=1,1000
       A(i,j) = A(i,j) + …
  enddo
```

各スレッドは、それぞれの A(i,j) のコピーが無効になった可能性があると考えます。A のインデックスを逆にすることで、各スレッドのデータの局所性が改善されます。また、連続したメモリーアクセスにより内部ループのベクトル化が向上する可能性があります。

OpenMP* のパフォーマンスを向上するため考慮すべきもう 1 つの項目は、ワークロードがループの反復全体に均等に分散されているかどうかです。均等に分散されていない場合は、DYNAMIC や GUIDED などのほかのスケジュール・オプションを検討します。デフォルトは、オーバーヘッドが最も低い STATIC です。

トップに戻る

マルチスレッド・アプリケーション向けのタイマー

CPU 時間には注意が必要です。すべてのタイマーは同じではありません。

Fortran 標準タイマー CPU_TIME() は、「プロセッサー時間」を返します。これは、Linux* の "time" コマンドと同じで、すべてのスレッドの合計時間です。そのため、マルチスレッド・アプリケーションはシリアル・アプリケーションよりも速く実行されないように見えます。

Fortran 組込みサブルーチン SYSTEM_CLOCK() は、リアルタイム・クロックからのデータを返します。また、すべてのスレッドの時間を合計しません。Linux* の "time" コマンドのように経過時間を報告します。SYSTEM_CLOCK を使用してコードの一部の時間を測定し、スレッド数の異なる複数のタイミング情報からスピードアップを計算できます。

```
call system_clock(count1, count_rate)
   -- compute good stuff --
call system_clock(count2, count_rate)
time = (count2 - count1) / count_rate
```

現在のプロセス開始からの経過時間を秒単位で返す、インテル固有の DCLOCK() 関数を使用することもできます。これは特に 24 時間を超える時間間隔の場合に便利です。SECNDS() も経過時間を測定しますが、24 時間未満の時間間隔のコードで使用します。

トップに戻る

スレッド・アフィニティー・インターフェイス

OpenMP* ディレクティブを使用して並列化したアプリケーションのパフォーマンスに関する考慮事項として、物理プロセッサーへのスレッドの配置(スレッド・アフィニティー) があります。メモリー配置とキャッシュ利用は、迅速なソリューションの鍵となる可能性があります。

OMP_PROC_BIND 環境変数は、次のように設定できます。

- *close*: 共有キャッシュの利点を得るため、スレッドを同じソケット上の連続するコアに割り当てます。
- *spread*: すべてのコアに少なくとも 1 つのスレッドが割り当てられるまで、スレッドを別々のコアに配置します。

このほかにも、スレッドの配置をより細かく制御するため、KMP_AFFINITY、OMP_PLACES、KMP_HW_SUBSET などの環境変数があります。詳細は、『インテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス』(英語) を参照してください。

スレッドを注意深く配置することで、特にハイパースレッディングが有効な場合に、パフォーマンスを向上できます。

トップに戻る

NUMA に関する考察

複数のマルチコア・プロセッサーを搭載したコンピューターは非一様メモリーアクセス (NUMA) です。つまり、メモリーアクセス時間は、プロセスが実行しているプロセッサーに対するプロセスの物理メモリー位置に依存します。最適なパフォーマンスが得られるように、プロセスによって割り当てられるメモリーは使用される場所の近くに配置すべきです。

Linux* オペレーティング・システムがメモリー管理に使用する「ファーストタッチ」ポリシーは、プロセスによって割り当てられるメモリーを、プロセスが実行しているプロセッサー上またはその近くの物理メモリーに配置します。これにより、後続のアクセスに必要な時間を軽減します。

特に大規模な配列では、OpenMP* ループ内のデータを初期化します。

```
!$OMP parallel do
do i=1,n
  do j=1,m
    A(j,i) = 0.0
  enddo
enddo

!$OMP parallel do
do i=1,n
  do j=1,m
    dowork(A(j,i))
  enddo
enddo
```

また、OMP_PROC_BIND やその他の関連する環境変数を適切に設定することを忘れないでください。

トップに戻る

セグメンテーション・フォルト - OpenMP* アプリケーションで一般的なランタイム問題

これは、OpenMP* に関して最もよく報告される問題です。

症状: アプリケーションがほとんど実行せずに、セグメンテーション・フォルトで失敗します。

原因: スタックサイズが小さすぎます。

解決策: プログラム全体と各スレッドのスタックサイズを増やします。

Linux* シェルと OpenMP* のスタックサイズを増やします。

最初に、Linux* シェルの制限を調整します。一般的な OS のデフォルトは、プログラム全体 (共有データとローカルデータ) で約 8MB です。Linux* でこれを増やすには、シェルの制限を増やします。bash では "ulimit -s unlimited" を使用します。csh では、"limit stacksize unlimited" を使用します。Windows* では、/F:100000000 を指定してプログラムを再リンクすると、プログラム全体のスタックサイズが 100,000,000 バイト (100MB) に設定されます。

セグメンテーション・フォルトが解決されない場合は、OMP_STACKSIZE 環境変数にプライベート・スタックのサイズをキロバイトで指定して、各スレッドのローカルデータのスタックサイズを増やします。これはスレッドごとに設定され、実際にメモリーが割り当てられることに注意してください。

トップに戻る

OpenMP* アプリケーションのデバッグのヒント

最初に、スレッド数を 1 に設定 (OMP_NUM_THREADS 環境変数を 1 に設定) して、実行します。動作する場合は、インテル® Inspector でマルチスレッド・コードを調査して、競合状態やその他の同期問題などのスレッドエラーをチェックします。

問題が解決しない場合は、-qopenmp-stubs -auto (Windows* では /Qopenmp-stubs /auto) コンパイラー・オプションを指定してビルドし、実行します。-qopenmp-stubs を指定すると、ランタイム・ライブラリー (RTL) 呼び出しは解決されますが、マルチスレッド・コードは生成されません。-auto は、ローカル配列を必ずスタック上に配置するようにコンパイラーに指示します。-qopenmp を指定すると、デフォルトでこのオプションが有効になります。

これで動作する場合は、不足している FIRSTPRIVATE 節や LASTPRIVATE 節がないかチェックします。FIRSTPRIVATE 節は、並列領域内の変数の値を並列領域の前の最後の値に設定します。並列領域終了時に、LASTPRIVATE は指定された変数の値を並列領域終了時の値に設定します。

これでも問題が解決しない場合は、マルチスレッド・コードの生成が原因ではありません。

-auto を省略してもアプリケーションが正常に実行する場合は、変更されたメモリーモデルが関係しています。おそらく、スタックサイズが不十分であるか、連続した呼び出しで値が保持されてない可能性があります。

PRINT 文を使用してデバッグする場合のヒントは、内部 I/O バッファーはスレッドセーフですが (アプリケーションが -qopenmp (Windows* では /Qopenmp) でコンパイルされている場合)、異なるスレッドの PRINT 文が出力される順序は不定です。デバッグ情報とともにスレッド番号を出力すると便利です。OpenMP* ランタイム・ライブラリー・ルーチンに USE OMP_LIB を追加して、スレッド番号を取得して整数結果を出力するため、OMP_GET_THREAD_NUM() を呼び出します。

-O0 -qopenmp を指定してデバッグします。ほかのほとんどの最適化とは異なり、OpenMP* のスレッド化は -O0 で無効になりません。

トップに戻る

浮動小数点再現性と OpenMP*

同じ実行ファイルを異なるスレッド数で実行すると、異なるワーク分割により演算の丸めにわずかな差が生じ、結果がわずかに異なることがあります。ほとんどのケースでは、-fp-model consistent を指定してコンパイルすることでこの問題を回避できます。「インテル® コンパイラーの浮動小数点演算における結果の一貫性」は、再現性のさまざまな側面について述べています。

浮動小数点リダクション操作は、OpenMP* において同じスレッド数の場合であっても厳密な再現性をまだ達成できていません。異なるスレッドからの結果が加算される順序は、実行ごとに異なる可能性があります。KMP_DETERMINISTIC_REDUCTION=true を設定して、固定数のスレッドでスタティック・スケジュールを使用します。これで異なる実行でも再現性が得られるはずです。

トップに戻る

インテル固有の環境変数

インテルは、OpenMP* ランタイム環境で認識される、インテル固有の多数の環境変数を提供しています。以下は、その中でも特に役立つものです。

- KMP_SETTINGS = 0 ¦ 1 - 実行時に環境変数またはデフォルト設定を出力します。
- KMP_VERSION = off ¦ on - ランタイム・ライブラリー・バージョンを出力します。

- KMP_LIBRARY = *turnaround* ¦ *throughput* ¦ *serial*
  - *turnaround* - アイドル状態のスレッドは、ほかのプロセスのワークに使用することはできません。
  - *throughput* - アイドル状態のスレッドは、ほかのプロセスのワークを手伝います。KMP_BLOCKTIME ミリ秒で指定されたスリープ状態に移行するまでのスレッド待機時間 (デフォルトは 200 ミリ秒) 後にスリープし別のワークに使用することができます。
- KMP_AFFINITY - 詳細は、『インテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス』を参照してください。
- KMP_CPUINFO_FILE - Linux* で /proc/cpuinfo の代わりに、マシントポロジーのファイルを使用します。
- KMP_DETERMINISTIC_REDUCTION -「浮動小数点再現性と OpenMP*」を参照してください。

トップに戻る

OpenMP* アプリケーションのデバッグツール

インテルによって拡張された GNU* プロジェクト・デバッガー (GDB) のほかに、アプリケーションのランタイム問題を検出するため、次のインテル® ソフトウェア・ツールを使用できます。

インテル® Advisor

- アプリケーションに並列処理を追加する作業を支援
- 期待されるスピードアップを予測するための実験が可能
- アプリケーションをスレッドセーフにする作業を支援
- インテル® Parallel Studio XE に含まれる

インテル® Inspector

マルチスレッド・アプリケーションにおいて発見が困難なエラーをピンポイントで特定する統合ツールセットです。

- データ競合
- デッドロック
- メモリーの不具合
- セキュリティー問題
- ランタイム解析を実行
- インテル® Parallel Studio XE に含まれる
- GUI で結果を表示

インテル® VTune™ Amplifier

コンカレンシー解析およびロックと待機の解析を実行

その他の機能

- コアの使用率を保証するためアプリケーションのコンカレンシー・レベルを表示
- スレッドと同期に関連するオーバーヘッドがパフォーマンスに影響している場所を特定

- パフォーマンスに影響するオブジェクトを特定
- スレッド間のワーク分割を視覚化
- スレッドのアクティブ/非アクティブ状態を表示
- ネイティブスレッド、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)、または OpenMP* アプリケーションをサポート
- 便利な GUI 表示
- ローカルまたはリモートデータ収集に対応
- インテル® Parallel Studio XE に含まれる

トップに戻る

まとめ

インテル® ソフトウェア・ツールは、マルチスレッド・アプリケーションがマルチコア・アーキテクチャーの利点を活用できるように広範なサポートを提供しています。Fortran アプリケーションをスレッド化する際に発生する、さまざまな問題に関するアドバイスや関連情報を利用できます。

トップに戻る

参考文献

インテル® ソフトウェア開発製品の情報、評価版、活発なユーザーフォーラムについては、http://software.intel.com (英語) を参照してください。

『インテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス』とこの記事で紹介したその他のツールのドキュメントは、ドキュメント (英語) で利用できます。

インテル® Advisor
インテル® VTune™ Amplifier
Code Modernization - コード・モダニゼーション (現代化) とは?
インテル® Fortran コンパイラー for Linux* で Co-Array Fortran の分散メモリー機能を使用するための基本ガイド (英語)
Windows* 用 Co-Array チュートリアル
ベクトル化コードブック
インテル® コンパイラーによる自動並列化
OpenMP* 標準規格 (英語)
命令レベルの並列性 (ILP) - Wikipedia*
自動ベクトル化 - Wikipedia*
アムダールの法則 - Wikipedia*

性能に関するテストや評価は、特定のコンピューター、システム、コンポーネント、またはそれらを組み合わせて行ったものであり、このテストによるインテル製品の性能の概算の値を表しているものです。システム、ハードウェア、ソフトウェアの設計、構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。システムやコンポーネントの購入を検討される場合は、ほかの情報も参考にして、パフォーマンスを総合的に評価することをお勧めします。インテル製品の性能評価およびインテル製品のパフォーマンスについてさらに詳しい情報をお知りになりたい場合は、www.intel.com/software/products (英語) を参照してください。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。