

インテル® コンパイラー 自動並列化ガイド

デュアルコア / マルチコア対応アプリケーション開発 ④

自動並列化入門

1.	はじめに.....	2
2.	自動並列化	2
3.	自動並列化の阻害要因とその対策	5
4.	自動並列化の適用について.....	9
5.	おわりに.....	11
付録.1	自動並列処理と OpenMP*	12
付録.2	依存性解析	15

注記:

『デュアルコア / マルチコア対応アプリケーション開発』は、次の 4 巻から構成されています。

- ① インテル® コンパイラー OpenMP* 入門
- ② インテル® C/C++ コンパイラー OpenMP* 活用ガイド
- ③ インテル® Fortran コンパイラー OpenMP* 活用ガイド
- ④ インテル® コンパイラー 自動並列化ガイド

本資料で言及されているインテル製品は、一般的な商業目的にのみ使用することを前提としています。特定の目的に本製品を使用する場合、適合性の評価についてはお客様の責任になります。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。

本資料のすべての情報は、現状のまま提供され、インテルは、本資料に記載表現されている情報及びその中に非明示的に記載されていると解釈される情報に対して一切の保証をいたしません。また、本資料に含まれる情報の誤りや、それによって生じるいかなるトラブル (PC パーツの破損などを含むがこれらに限られない) に対しても一切の責任と補償義務を負いません。また、本資料に掲載されている内容は、予告なく変更されることがあります。

1. はじめに

今後は、より多くのコアが一つのプロセッサ上に実装されるデュアルコアやマルチコアが一般的なプロセッサとなることは間違いありません。デュアルコアとマルチコアの持つ大きな可能性とその能力を最大限に発揮するためのキーとなるのが、マルチスレッドでのアプリケーション実行の高速化です。

このアプリケーションの高速化には、複数のスレッドが並列に処理を行うことが必要になります。ここで疑問となるのは、アプリケーション・プログラムに対して並列処理を適用する為の特別な作業やそのための開発工数が必要になるかということです。実際には、マルチスレッド化や並列化といった作業にはそれほどの時間を必要とするものではありません。マルチスレッド対応の開発ツールがあれば、これらの並列化は容易に行うことが可能です。プログラム開発者は、プログラムの本質的なロジックを記述することに専念し、並列化については、既に高度に最適化・並列化されたライブラリーを利用したり、並列化コンパイラーの支援によって、プログラムのマルチスレッド化を図ることが現在では可能になっています。

ここでは、容易なマルチスレッド・プログラミングを可能とするコンパイラーの自動並列化による自動並列化機能を紹介します。自動並列化とは、その名のとおりコンパイラーがソースコードを解析し、自動的にプログラムの構造に適したマルチスレッド実行が可能な実行モジュールを作成することを意味します。コンパイラーの自動並列化のオプションによって、アプリケーション内で複数のスレッドや拡張機能の自動作成が可能になっています。生成されるバイナリはマルチコア・プラットフォーム向けに最適化され、アプリケーション処理性能の効率化がなされています。

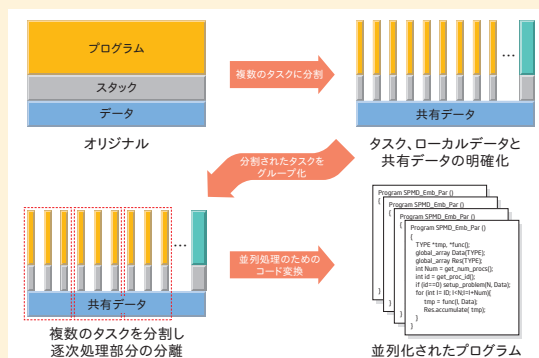
プログラム開発者や研究者がプログラムを作るのは、そのプログラムの並列化を行う為ではありません。ある処理、解析を目的にプログラムを書き、そのプログラムをプラットフォームで効率良く、高速に実行できることを目的としています。これらのコンパイルツールは、開発者が本来のプログラムの開発目的である、これらのアルゴリズムの実装やロジックの検証のための作業に専念することを可能とし、必要ではあります但し本質的ではない、並列化という手間のかかる作業を開発者の代わりに担うものです。

2. 自動並列化

自動並列化はループとプログラム構造をコンパイラーが解析し、プログラム中で並列実行可能な領域を見つける高度なプログラム解析機能に基づいた並列処理です。コンパイラーによって認識された並列実行領域は、その領域内でのマルチスレッド並列プログラムとして実行されます。ループの展開やデータのアライメントの共有、スレッドのスケジューリング、同期化といった低レベルの細かい作業をユーザが行わなくても並列化が行われるようになり、マルチプロセッサ・システムやハイパースレッディング・テクノロジー対応システムが持つスレッド機能を十分に生かすことができます。

自動並列化は、コンパイル時にコンパイル・オプションとして、/Qparallel スイッチ (Windows*)、-parallel スイッチ (Linux*) を指定することで可能となります。これらのオプションが指定されると、コンパイラーは入力されたファイルに対してその並列処理の可能性の解析と並列化のための最適化を行います。

では、実際には、どのように変換されるのでしょうか？



並列化コンパイルフロー

自動並列化コンパイラーでは、プログラム内でループとして形成される一連の処理におけるその実行タスクとデータフローを解析します。実際には、プログラム中で利用される配列や変数がループ中でどのようにアクセスされるかを解析します。このデータアクセスのパターンにおいて、ループの実行時にそのデータ参照においてコンフリクトが無い事を確認する依存性解析を行います。データアクセスのパターン解析では、プログラム内の配列や変数が、並列化した場合に各スレッドが“Private”として個々に利用するものであるか、“Shared”としてスレッド間で共有されるかを判定します。また自動並列化では、ループ処理のタスクで並列に実行できる部分と逐次処理部分も分離して、並列化処理のためのコードへの変換がなされます。

自動並列化において、コンパイラーは並列可能なループを見つけ、そのループ内のデータ参照パターンを自動で認識し、それらのデータについての属性を判断し、並列化のためのコードの書き換え、同期処理の追加、実行時のスケジューリングの適用など非常に多くの解析と作業を行います。

スカラー最適化やループ最適化（レジスター内でのベクトル化やメモリー最適化）といった最適化機能と並列化機能の2つは緊密に統合されて、キャッシュの局所性を高め、スレッドレベル以外での並列性を効率的に利用しています。自動並列化では、入れ子になった多重ループの場合、できるだけ外側のループを並列化の対象とするようになっています。これは、できるだけ並列処理のオーバーヘッドを減らすことに寄与します。たとえば、二重にネストされたループの場合、最内側ループについては、キャッシュの局所性を高め、レジスター利用の最適化を最大限に図り、計算回数とメモリー参照を最小化するような最適化の適用を目指し、外側ループでマルチスレッド・コードを生成するような最適化を行います。自動並列化での並列処理は、一般には、「細粒度」での並列処理となります。

例えば、次のようなプログラムの自動並列化を考えてみましょう。

```
for (i=1; i<100; i++)  
{  
    a[i] = a[i] + b[i] * c[i];  
}
```

このプログラムは、コンパイラーによる自動並列化により、自動的に次のように各スレッドのタスクに分割されます。

```
// Thread 1  
for (i=1; i<50; i++)  
{  
    a[i] = a[i] + b[i] * c[i];  
}  
// Thread 2  
for (i=50; i<100; i++)  
{  
    a[i] = a[i] + b[i] * c[i];  
}
```

プログラムの開発者は、この例で示したようなループの分割、分割したループのマルチスレッドでの実行の制御などに対して気を使う必要はありません。普通のループで必要な処理を記述すれば、コンパイラーが並列処理をサポートします。またこの例では、ループ回数は陽的に記述しましたが、実際にはループ回数はプログラムの実行時に始めて明らかになるケースがほとんどです。コンパイラーは、並列化したループの実行前に各スレッドが実行するループの反復回数を決定して実行します。また、反復回数が少ない場合には、並列化処理を行わず逐次処理を行うような条件判断を行うことも可能です。

簡単な π の計算での自動並列化コンパイルを示します。 π の値を計算するには、以下のようなプログラムが可能です。

```
1 #define num_steps 1000000  
2 double step;  
3 main ()  
4 { int i; double x, pi, sum = 0.0;  
5  
6     step = 1.0/(double) num_steps;  
7  
8     for (i=1;i<= num_steps; i++){  
9         x = (i-0.5)*step;  
10        sum = sum + 4.0/(1.0+x*x);  
11    }  
12    pi = step * sum;  
13 }
```

このプログラムに対して、自動並列化コンパイルのためのコンパイラー・オプションを指定してコンパイルすることで自動並列化コンパイルが可能です。(Linux* でのコンパイル例)

```
$ icc -parallel -par-report3 -par-threshold0 -O3 sample.c
procedure: main
sample.c(9) : (col. 11) remark: LOOP WAS AUTO-PARALLELIZED
```

```
parallel loop: line 9
  shared      : { }
  private     : {"i", "x"}
  first priv.: {"step"}
  reductions  : {"sum"}
```

ここでは、自動並列化に関するレポートを出力させているので、どのループが並列化されたかを示すメッセージが出力されています。

最初に示したように、自動並列化解析とはコンパイラーによる並列実行可能なタスクの認識とそのタスク内でのデータの利用に関する解析です。コンパイラーは、プログラム内のループの構造をチェックしてその並列実行の可能性をチェックします。そのループを複数のタスクに分割できると判断した後で、並列処理が可能であるかどうかはそれらのループ内でのデータの依存性の判断になります。例えば、次のような例では、各ループの実行はその前のループの反復計算の結果を必要としますので依存性があることになります。従って、このようなループの自動並列化はできません。コンパイラーは、依存性解析を行った結果をメッセージとしても出力しますので、そのメッセージなどを参考に並列化のためのプログラム変更などの検討も可能となります。

```
$ cat -n sample.c
1  #define N 1000
2  main ()
3  { int i;   double a[N], b[N], c[N];
4    for (i=1; i<= N; i++){
5        a[i] = a[i-1] + b[i] * c[i];
6    }
7  }

$ icc -parallel -par-report3 -par-threshold0 sample.c
procedure: main
serial loop: line 5
      flow data dependence from line 5 to line 5      flow data dependence
from stmt 2 to stmt 2, due to "a"
```

コンパイラーが生成するコードは、ハイレベルのマルチスレッド・ライブラリーを参照します。このマルチスレッド・ライブラリーは、OpenMP* でも共通に利用されます。これによって、自動並列化と OpenMP* の混在が可能となり、またオペレーティング・システムに関係なく、自動並列化の適用が可能になります。

3. 自動並列化の阻害要因とその対策

コンパイラーが自動並列化によるコードの並列化を行う際には、いくつかの条件を満たしている必要があります。逆にいうと、このような条件に適応できないケースでは、コンパイラーによる自動並列化はできません。

自動並列化の対象となるのは、C/C++ と Fortran などでの「ループ」による反復計算部分です。このループの反復計算を、マルチスレッド化によって高速化を図ることを目的としています。コンパイラーによるループの並列化に際して、ループ内のデータフローやその構造などが解析されます。その解析段階で以下のような条件が並列化には必要となります。

- ループの反復回数がループの実行を開始される時に明らかになっている必要があります。従って、while ループなどの並列化は通常はできません。
- ループの並列化に際して、十分な計算負荷がそのループにあることが必要となります。この並列化の可否の判断は、コンパイラーがループ内の演算量を推定し設定されたしきい値によって決定されます。/Qpar-threshold[n] (Windows*) 及び -par-threshold[n] (Linux*) によって、このしきい値のパラメータは設定できます。n の値の範囲は 0 ~ 100 であり、0 はこのしきい値の判定なしで並列化可能なループは全て並列化することを指示します。100 を指定した場合、性能向上の可能性が高いループのみの並列化を行います。(省略時は、75 の設定)

ループに十分な仕事量が無いとコンパイラーが判断した場合、次のようなメッセージを出力します。

```
procedure: copy  
serial loop: line 30: not a parallel candidate due to insufficient work
```

- ループ内の演算は、相互に独立である必要があります。言い換えれば、各反復計算の実行順序が計算の整合性に影響を与えないことが必須です。(この場合、計算の丸め誤差などでの計算順序の差異については、自動並列化では考慮はなされません) ループの各反復が他の反復計算の結果を参照したりする場合には並列化はできません。(依存関係)

```
for (i=1; i<= n; i++) {  
    a[インデックス計算式 1] = . . . . .  
    . . . . . = a[インデックス計算式 2] . . . . . ;  
}
```

ここでは、インデックス計算式 1 の値は、各反復時に異なった値となる必要があります。また、インデックス計算式 1 とインデックス計算式 2 の値が異なる場合には、参照関係に依存性が生じます。

- 配列の総和を計算するような場合には、実際には各反復計算の結果は相互依存します。しかし、ループ構造が明確な場合などは、コンパイラーがソースコードを変換し、この見かけの依存性を排除することも可能となります。
- ループ内での演算の独立性の確認のためには、ループ内で使われている変数、配列の参照関係が明確である必要があります。C/C++ でのポインターでの配列参照は、同じメモリー上を複数のポインターが指す場合もあり、並列化の阻害要因となります。このようなループ中でのポインターの配列参照については、コンパイルオプション /Qfno-alias (Windows*) 及び -fno-alias (Linux*) などの指定で、コンパイル時に明示的にコンパイラーにそのポインターの参照が安全であることを指示することで、並列化が可能となる場合もあります。
- ループ内に外部関数の呼び出しがあるような場合には、コンパイラーは自動並列化を行うことはできません。これは外部関数の呼び出しによる依存関係をコンパイラーが判断できないためです。この場合、/Qipo (Windows*) 及び -ipo (Linux*) の指定を行うことでプロシージャー間の最適化を図り、依存関係を明確にすることで自動並列化を可能とすることもできます。また、実際のアプリケーションで高い並列性能を発揮するには、できるだけ並列実行領域を大きく確保することが必要となります。そのためには、関数やサブルーチンを含む領域を並列実行の対象とすることは非常に効率の良い並列化を可能とします。(付録参照)

以下、具体的なプログラム例による自動並列化の適用について示します。

```
void add (int n, int k, double *a, double *b, double *c)
{
    int i;
    for (i=1; i<= n; i++){
        a[i] = a[i+k] + b[i] * c[i];
    }
}
```

このコードを自動並列化コンパイルした場合、コンパイラーは以下のようなメッセージを出力します。ここでは、配列 a の参照に依存性があることを示しています。

```
procedure: add
serial loop: line 20
    anti data dependence assumed from line 20 to line 20      anti data
dependence assumed from stmt 2 to stmt 2, due to "a"
    flow data dependence assumed from line 20 to line 20      flow data
dependence assumed from stmt 2 to stmt 2, due to "a"
    flow data dependence assumed from line 20 to line 20      flow data
dependence assumed from stmt 2 to stmt 2, due to "a"
```

コンパイラーは、k の値を知らないため、例えば k = -1 の場合のようにループ内の反復に依存性がある場合も想定し、このループの自動並列化を禁止します。しかし、k の値が明確で、例えば k の値がゼロまたは 1000 以上の数値であれば、このループは並列化可能です。

このループの呼び出しで、k の値が設定されている場合、次のようなケースの並列化を考えます。

```
k=0;
add (n, k, a, b, c);
```

この場合、k の値が 0 ですので実際には並列実行が可能です。ここで、コンパイラーにプロシージャー間の最適化を行う、/Qipo (Windows*) 及び -ipo (Linux*) の指定を行った場合、コンパイラーはこのループをインライン展開し並列化します。

```
procedure: main
sample.c(10) : (col. 3) remark: LOOP WAS AUTO-PARALLELIZED.
parallel loop: line 10
    shared      : {"a", "b", "c"}
    private    : {"i"}
    first priv.: { }
    reductions : { }
```

次のような配列インデックスが間接的に参照される場合も自動並列処理は阻害されます。

```
{
  int i;
  for(i=0; i<n; i++) a[indx[i]] += s*b[i];
}
```

このようなループの自動並列化は通常はできません。もし、このプログラムで配列 index での配列 a の参照に依存性が無いとわかっているのであれば、OpenMP* 宣言子などを挿入することで並列化可能です。

```
{
  int i;
  #pragma omp parallel for
  for(i=0; i<n; i++) a[indx[i]] += s*b[i];
}
```

また、コンパイラーにこのループには依存性がないことを指示するための宣言子 (#pragma ivdep) を加え、コンパイル時に /Qivdep_parallel (Windows*) 及び -ivdep_parallel (Linux*) の指定を行った場合には、自動並列化も可能となります。

```
$ cat -n sample.c
 1 #define N 100000
 2 main ()
.....
28 void update_ivdep(int n, double *a, double *b,
29                   int *indx, double s)
30 {
31   int i;
32   #pragma ivdep
33   for(i=0; i<n; i++) a[indx[i]] += s*b[i];
34 }
$ icc -O3 -parallel -ivdep_parallel -par-report3 sample.c
.....
sample.c(33) : (col. 23) remark: LOOP WAS AUTO-PARALLELIZED.
  parallel loop: line 33
    shared      : {"indx", "b", "a", "n"}
    private    : {"_2$4_2_$spill_t0"}
    first priv.: {"s"}
    reductions : { }
```

自動並列化では、コンパイラーは可能な限り外側ループでの並列化を行います。ただ、実際のループの演算回数が明示的にプログラム中に明記されていないようなケースやプロシージャ間でそのデータが引き渡されるケースでは、効率の悪いループを並列化の対象とするケースもあります。

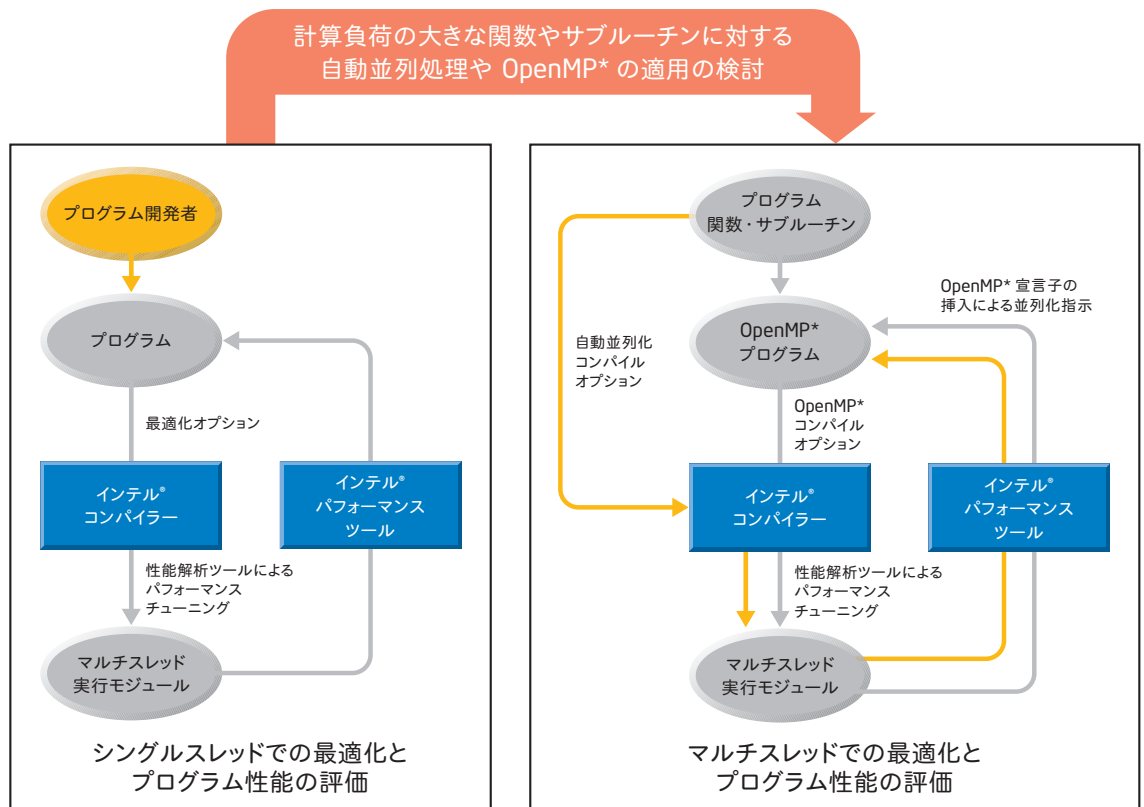
```
1      subroutine copy (imx,jmx,kmx,imp2,jmp2,kmp2,w,ws)
2      real*8 ws(5,imx,jmx,kmx),w(2,imx,jmx,kmx)
3      do nv=1,5
4          do k = 1,kmx
5              do j = 1,jmx
6                  do i = 1,imx
7                      ws(i,j,k,nv) = w(i,j,k,nv)
8                  end do
9              end do
10         end do
11     end do
12     return
13     end
```

この例の場合、kmx やjmの数值にも依存しますが、もしkmx が十分に大きな数值であれば、最外側の行番号3のループでの並列化は非効率です。

4. 自動並列化の適用について

自動並列化は、プログラム開発者から並列処理を行うための様々な作業を解放します。しかし、自動並列化では、その自動並列化を阻害する要因も多々あります。

以下の図に示すのは、簡単なプログラム開発、最適化、並列化の流れです。プログラムを自動並列化し性能評価ツールなどを使い、必要に応じて OpenMP* 宣言子による並列処理の適用を行うというのが実際の開発の流れになります。



プログラムの開発には、多くの試行錯誤とプログラムの実行と検証、デバッグといった作業が必要になります。またプログラムについては、その実行性能の向上を図るためのソースコードの書き換えなどを行う作業を、プログラムの開発中も開発後も行う必要があります。このような作業を効率良く、また短時間で行うには、優れた開発環境が必要です。インテルのソフトウェア製品は、マルチスレッド・プログラミングに対応したインテル® コンパイラーなどのプログラミングを支援する豊富なツール群が用意されています。それらのツールを活用することで、プログラム開発サイクルは、より充実したものになります。

自動並列化適用のためのステップとしては、次のような段階的な検討が必要です。

1. パフォーマンス・ツールを使いプログラムの動作の詳細な解析を行います。ホットスポットを見つけることが並列処理では必要になります。プログラム全体を /Qparallel スイッチ (Windows*)、-parallel スイッチ (Linux*) でコンパイルして並列化することも可能ですが、この場合、不要な部分の並列処理によるオーバーヘッドの発生の危険性などもあります。
2. /Qparallel スイッチ (Windows*)、-parallel スイッチ (Linux*) でホットスポットを含む関数、サブルーチンをコンパイルします。/Qpar_report3 (Windows*) 及び -par_report3 (Linux*) オプションを指定して、自動並列化についてのレポート機能を有効にします。これによって、どのループが並列化され、どのループが並列化できなかったかを確認します。

3. 並列化できなかったループについては、その構造の検討や依存関係の解消などについての検討を行います。ポインターなどによる参照が問題の場合には、コンパイラー・オプションの指定やポインター宣言での restrict 構文の指定なども検討する必要があります。
4. コンパイラーがループを並列化するために必要な変換や依存関係の解消のために行うプログラムの変更が、他のハイレベルの最適化手法（ソフトウェア・パイプラインやベクトル化）などに影響を与えることがあります。この並列化による他のハイレベルの最適化の阻害は避ける必要があります。そのためには、並列化の適用時と非適用時の性能を比較検討する必要もあります。多くのプログラム言語は逐次処理を前提として開発されているため、逐次処理ではプログラムの生産性を高め、より容易なプログラミングを可能とする機能が、並列処理においては非効率になる場合もあります。
5. 計算コアの部分について、もし可能であれば既にマルチスレッド向けに高度に最適化されているインテル MKL (Math Kernel Library) などを積極的に利用することが、並列処理の効率化を図る有効な手段です。これらのライブラリーのマルチスレッド化には OpenMP* が使用されているので、容易に自動並列化に組み込むことが可能です。
6. 自動並列化の適用が困難でも OpenMP* 宣言子の挿入などで並列化可能なループについては、OpenMP* 宣言子を挿入して並列化を試みることも可能です。

5. おわりに

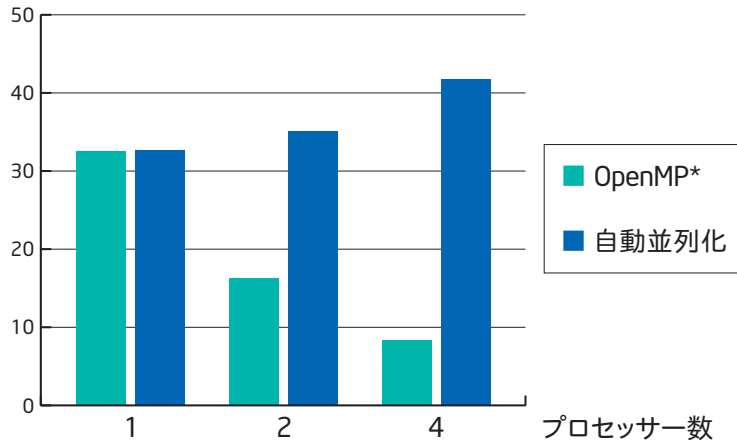
自動並列化はコンパイラーがユーザーの介在なしでマルチスレッド化を行うツールであり、もっとも簡便な並列化をユーザーに提供するものです。SMP システムやデュアルコア / マルチコア・プロセッサ・システムでは、データの分散配置については本質的に検討する必要がなく、プログラム自身の構造が明白であるようなケースでは、自動並列化は十分な並列化効率を発揮できます。しかし、自動並列化のサンプルで示されるような単一ファイル、単一アルゴリズムから構成されるプログラムであれば、自動並列化による並列処理の効果を示すことは比較的容易だとしても、フルスケールのアプリケーションに自動並列化を適用した場合には、数多くの自動並列化の阻害要因に直面することも事実です。

自動並列化の適用には現在でも限界があり、その適用範囲は必ずしも広いわけではありません。しかし、今後のマイクロプロセッサのデュアルコア、マルチコア化によって、非常に廉価にマルチプロセッサの実装が可能になります。このような状況において、自動並列化は簡便な並列処理ツールとして重要な位置を占めることになります。自動並列化はある意味、OpenMP* のようなコンパイラーに対して並列処理を指示するツールと共に使うことで、更にその価値を高めることが可能になります。

付録.1 自動並列処理と OpenMP*

次の性能データは、OpenMP* ホームページにある OpenMP* のサンプルプログラム md.f を OpenMP* で並列化したものと自動で並列化したものを比較したものです。

経過時間



SGI Altixシステム / インテル® Itanium® 2 プロセッサ 1.6GHz での計測結果 (出典: <http://www.openmp.org/>)

このケースでは自動並列化の効果は見られません。

実際のコンパイルでは、以下のように自動並列化によって、多くのループは並列化されています。

```
$ ifort -O3 -parallel -par_threshold0 md.f
md.f(35) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(98) : (col. 8) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(161) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(181) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(212) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
```

このコードでもっとも計算時間がかかるのは、以下のループです。このループの自動並列化は、サブルーチンの呼び出しなどもあり阻害されています。実際には、このループから呼び出されるサブルーチン dist 内のループなどは自動並列化されています。しかし、多重ループからの並列実行領域の呼び出しとなるため並列効率を上げることは困難であり、逆にオーバーヘッドが増えています。

```
96      do i=1,np
97          ! compute potential energy and forces
98          f(1:nd,i) = 0.0
99          do j=1,np
100             if (i .ne. j) then
101                 call dist(nd,box,pos(1,i),pos(1,j),rij,d)
102                 ! attribute half of the potential energy to particle 'j'
103                 pot = pot + 0.5*v(d)
104                 do k=1,nd
105                     f(k,i) = f(k,i) - rij(k)*dv(d)/d
106                 enddo
107             endif
108         enddo
109         ! compute kinetic energy
110         kin = kin + dotr8(nd,vel(1,i),vel(1,i))
111     enddo
```

```
148      subroutine dist(nd,box,r1,r2,dr,d)
.....
160      d = 0.0
161      do i=1,nd
162          dr(i) = r1(i) - r2(i)
163          d = d + dr(i)**2.
164      enddo
165      d = sqrt(d)
166
167      return
168      end
```

このコードについては、以下のような OpenMP* 宣言子の指定が可能です。この指定によってサブルーチンの呼び出しを含めて、疎粒度での並列処理が可能となります。

```
!$omp parallel do
!$omp& default(shared)
!$omp& private(i,j,k,rij,d)
!$omp& reduction(+ : pot, kin)
    do i=1,np
        ! compute potential energy and forces
        f(1:nd,i) = 0.0
        do j=1,np
            if (i .ne. j) then
                call dist(nd,box,pos(1,i),pos(1,j),rij,d)
                ! attribute half of the potential energy to particle 'j'
                pot = pot + 0.5*v(d)
                do k=1,nd
                    f(k,i) = f(k,i) - rij(k)*dv(d)/d
                enddo
            endif
        enddo
        ! compute kinetic energy
        kin = kin + dotr8(nd,vel(1,i),vel(1,i))
    enddo
!$omp end parallel do
    kin = kin*0.5*mass
```

自動並列化は非常に容易なマルチスレッド・プログラミングのツールですが、ループレベルでの並列化の認識には限界があることも事実です。自動並列化の内容を理解し、その上で OpenMP* をコンパイラーに対する並列化指示のためのツールとして使うことで、さらに効率の良い並列処理が可能となります。

付録.2 依存性解析

自動並列化の実施に際しては、その依存性解析が重要となります。自動並列化の適用時にコンパイラーが出力するメッセージには、この依存性解析の結果が含まれます。プログラムの並列化には、この依存性の排除が必要となります。

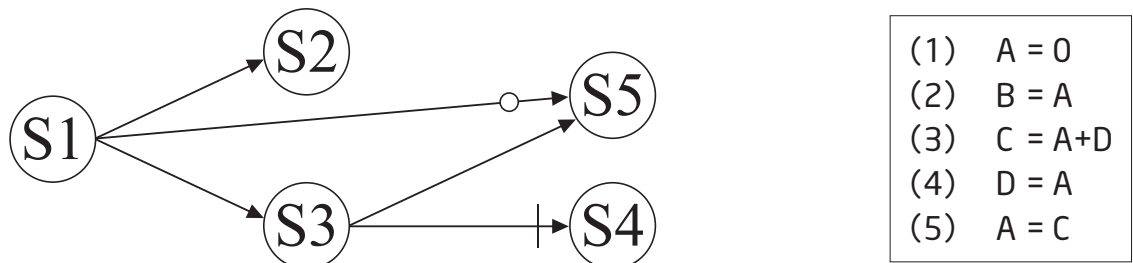
```
void add (int n, int k, double *a, double *b, double *c)
{
    int i;
    for (i=1;i<= n; i++){
        a[i] = a[i+k] + b[i] * c[i];
    }
}
```

このコードを自動並列化コンパイルした場合、コンパイラーは以下のようなメッセージを出力します。ここでは、配列 a の参照に依存性があることを示しています。

```
procedure: add
serial loop: line 20
    anti data dependence assumed from line 20 to line 20      anti data
dependence assumed from stmt 2 to stmt 2, due to "a"
    flow data dependence assumed from line 20 to line 20      flow data
dependence assumed from stmt 2 to stmt 2, due to "a"
```

ここで示される“anti data dependence...”や“flow data dependence...”の意味を理解することも必要になります。

ここでは以下のような計算式を考えてみます。下の図の S1 から S5 がそれぞれの式になります。



flow data dependence (フロー依存)

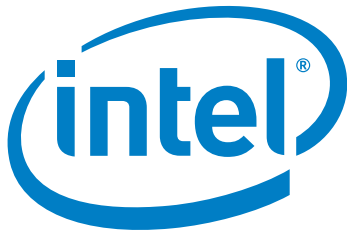
これは、データを定義されたり代入されたりした変数が、後続の式で使われることを意味しています。(例: 変数 A は、式 (1) で定義され、式 (2) で使われています)

anti data dependence (アンチ依存)

これは、一つの式で使われた変数が、後続の式で代入や定義されることを意味しています。(例: 変数 D は、式 (3) で参照されて使われ、式 (4) でデータの代入がなされています)

output dependence (出力依存)

これは、一つの式で定義された変数に対して、後続の式で再度データの定義、代入がなされていることを意味しています。(例: 変数 A は、式 (1) で定義され、再度 式 (5) で定義、代入がなされています。)



本資料で言及されているインテル製品は、一般的な商業目的にのみ使用することを前提としています。特定の目的に本製品を使用する場合、適合性の評価についてはお客様の責任になります。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。

分散並列処理、スーパーコンピュータなどに関する各種情報は、インテル HPC リソース・センターをご参照ください。 <http://www.intel.co.jp/jp/go/hpc/>

インテル株式会社

〒 300-2635 茨城県つくば市東光台 5-6

<http://www.intel.co.jp/>

Intel、インテル、Intel ロゴ、Pentium、Xeon、Itanium、VTune は、アメリカ合衆国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です。

* 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。

©2006 Intel Corporation. 無断での引用、転載を禁じます。

2006 年 8 月

528J-001

JPN/0608/PDF/SE/DEG/KS