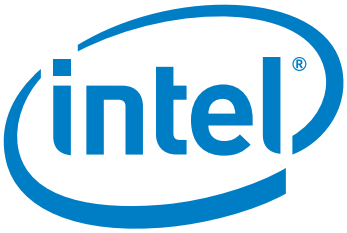


インテル® C/C++ コンパイラー OpenMP* 活用ガイド

デュアルコア / マルチコア対応アプリケーション開発 ②

このテキストは、C/C++ プログラマー向けのマルチスレッド・プログラミングの入門として、OpenMP* について解説した資料です。OpenMP* を利用してのマルチスレッド・プログラミングについて、平易に説明しています。ここで紹介しているプログラムの実行例やコンパイルオプションの説明は、全てインテル® コンパイラー バージョン 9.0 を用いています。より詳細な説明はコンパイラー・マニュアルなどをご参照ください。

1. はじめに 2
2. マルチスレッド・プログラミング 3
3. OpenMP* マルチスレッド並列プログラミング 5
3.1 プログラミングの特徴 5
3.2 OpenMP* アーキテクチャー 6
3.3 OpenMP* の適用の阻害要因とその対策 8
4. OpenMP* によるマルチスレッド・プログラミング 9
4.1 宣言子の書式 9
4.2 OpenMP* ライブラリー 9
4.3 OpenMP* 宣言子一覧 10
4.4 構造ブロック 10
5. OpenMP* 宣言子の構文説明 11
5.1 parallel 構文（並列実行領域の指定） 11
5.2 ワークシェアリング構文 14
5.3 データ環境 19
5.4 同期構文 23
5.5 その他の宣言句 27
5.6 実行時関数 / 環境変数 29
6. OpenMP* の適用について 30
7. おわりに 31

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

1. はじめに

今後は、より多くのコアが一つのプロセッサー上に実装されるデュアルコアとマルチコアが一般的なプロセッサー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。ここで問題となるのは、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。プログラムの開発者やプログラマーは、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をしたりすることによって、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。

インテル® コンパイラー バージョン 9.0 は、優れたマルチスレッド対応の開発ツールです。32bit と 64bit Linux* 及び 32bit と 64bit Windows* のオブジェクト・コードを作成し、それぞれのプラットフォームでのプログラムの性能を最大限に引き出すことを可能とします。バージョン 9.0 でサポートされているプログラムのマルチスレッド化は、大きく 2 つに分けられます。一つは自動並列化、そしてもうひとつが OpenMP* のサポートです。自動並列化では、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します。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。OpenMP* はユーザーがプログラムの並列化を指示する構文をプログラム中に記述することで、マルチスレッド並列プログラムを開発する枠組みを提供します。バージョン 9.0 では、OpenMP* 2.5 仕様の実装がなされており、最新の OpenMP* の機能が利用可能となっています。ここでは、この OpenMP* によるマルチスレッド・プログラミングの基本をご紹介致します。

2. マルチスレッド・プログラミング

OpenMP* の説明を始める前に、マルチスレッド・プログラミングについて少し説明します。スレッドとは OS による処理単位であり、OS は要求される処理タスクを分割しそれぞれをこのスレッドの単位で処理を行います。このスレッドを複数の CPU に割り当てられ処理するのが、マルチスレッドによる並列処理となります。

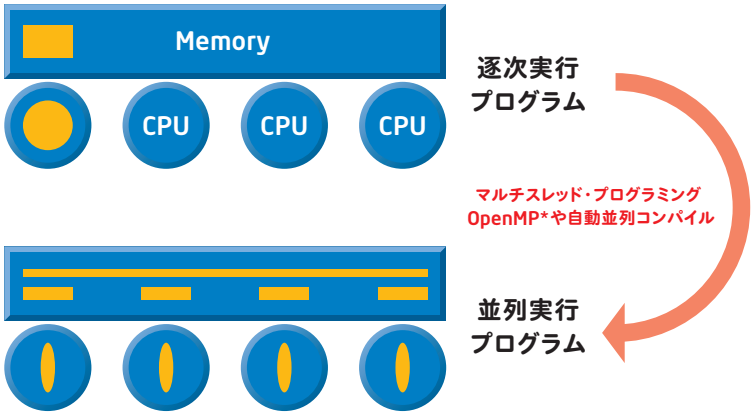


図 .1 マルチスレッドによる並列実行

マルチスレッドは、プロセス内の仮想メモリー空間といったリソースやコンテキストを共有し、固有のスタックとプログラムカウンター (PC) を個別に持つことになります。

個々のプロセスが固有の仮想空間上で動作するプロセスと異なり、このメモリー空間を共有することで、スレッドは並列処理を行う上で非常に効率的な実行を可能としています。メモリー空間の共有により、スレッド間でのデータのやり取りは直接アクセスが可能となり、OS を介した通信のようなオーバーヘッドの大きなオペレーションを必要としません。同時に、スレッドの生成や切り替えはプロセスの場合と比較して高速であり、システムのリソースへのインパクトも非常に小さくなります。このようなスレッドの特性を最大限に活用することで、効率のよい並列処理を行うのが、マルチスレッド・プログラミングです。

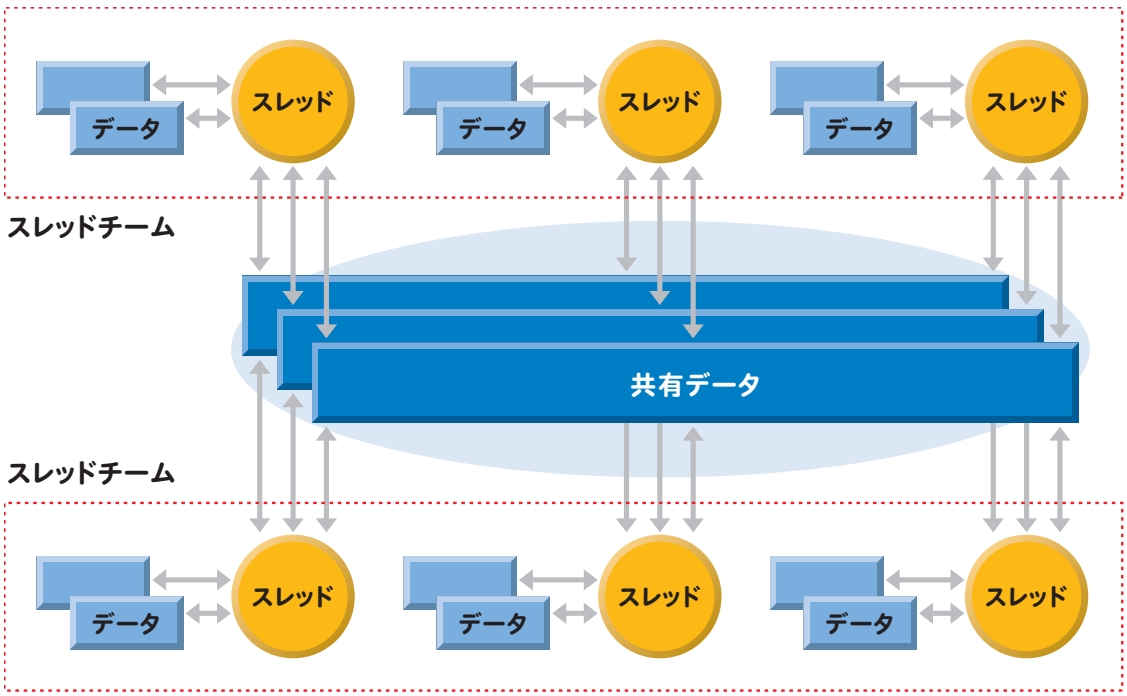


図 .2 マルチスレッドによるリソースとコンテキストの共有

3. OpenMP* マルチスレッド並列プログラミング

OpenMP* は、マルチスレッド並列プログラミングのための API（Application Programming Interface）です。OpenMP* API は 1997 年に発表され、その後継続的にバージョンアップされている業界標準規格で、多くのハードウェアおよびソフトウェア・ベンダーが参加する非営利団体（Open MP Architecture Review Board）によって管理されており、Linux*、UNIX* そして Windows* システムで利用可能です。OpenMP* は、C/C++ や Fortran と言ったコンパイラー言語ではありません。また、OpenMP* 自身はコンパイラーではなく、コンパイラーに対する並列処理の機能拡張を規定したものです。したがって、OpenMP* を利用するには、インテル® コンパイラー バージョン 9.0 のような OpenMP* をサポートするコンパイラーが必要です。OpenMP* の詳細については、OpenMP* のホームページ http://www.openmp.org/ にその歴史も含めて、詳細な情報があります。最新の OpenMP* のリリースは、2005 年 5 月の OpenMP* 2.5 であり、この仕様では初めて、C/C++ と Fortran の双方の規格の統合がなされました。

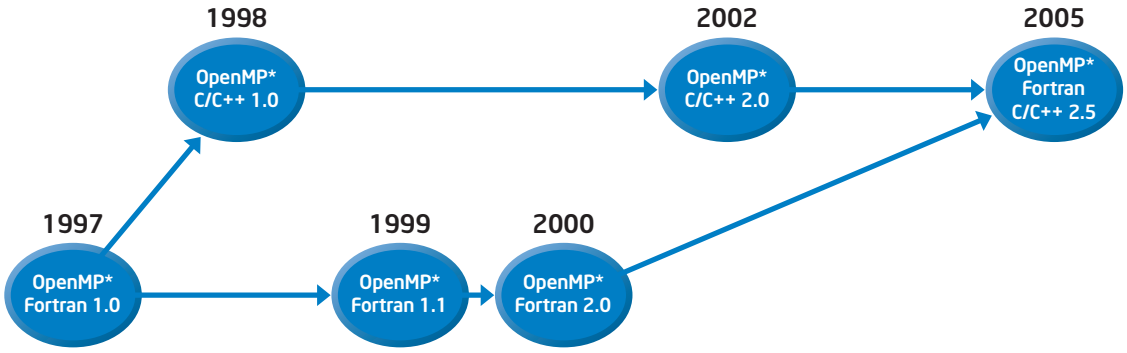


図 .3 OpenMP* リリースの歴史

3.1 プログラミングの特長

C/C++ や Fortran には並列処理のための API が無かったために、それを補うものとして規定されたのが OpenMP* であるとも言えます。OpenMP* は、C/C++ や Fortran の言語規格に準拠しているため、OpenMP* を利用してもプログラムの移植性や互換性を損なうことなく、並列処理を容易に適用することができます。

コンパイラーのサポート

OpenMP* を利用した並列プログラミングは、プログラムに対して OpenMP* で規定された宣言子を挿入し、コンパイル時にコンパイルオプションとして /Qopenmp スイッチ（Windows*）、-openmp スイッチ（Linux*）を指定することで可能となります。コンパイラーは、ソースコード中の OpenMP* 宣言子で指示された範囲を並列化したことをメッセージとして出力します。

明示的な並列化の指示

OpenMP* 宣言子は、コンパイラーに対して並列化のためのヒントを与えるのでなく、明示的に並列化を指示するものです。したがって、間違った宣言子を指定しても、コンパイラーはその指示に従って並列化を行います。また、データの依存性などがあっても、コンパイラーは警告メッセージやエラーメッセージを出したり、その指示を無視することなく忠実に並列化を行いますので、依存性があるループなどを OpenMP* で並列化した場合には、計算結果が不正になります。一方、OpenMP* を使用した場合、スレッドの生成や各スレッドの同期コントロールといった制御をユーザーが気にする必要はありません。

プログラムの段階的な並列化が可能

プログラムの開発者は、コードの設計時から OpenMP* を利用した並列処理を実装することも可能です。また、既に開発されたプログラムを、OpenMP* を利用して段階的に並列化することも可能です。OpenMP* での並列化を適用していて計算などが不正になった場合、簡単にその部分だけを逐次実行に切り替えることができるので、プログラムのデバッグが非常に容易です。

自動並列化との併用

自動並列化と OpenMP* を併用することも可能であり、プログラムの一部だけを OpenMP* で並列化し、他の部分を自動並列化することも可能です。OpenMP* と自動並列化は、ハイレベルのマルチスレッド・ライブラリーを共有することで混在が可能となっています。また、オペレーティング・システム (32bit と 64bit Linux* 及び 32bit と 64bit Windows*) に関係なく並列処理の適用が可能となっています。

疎粒度での並列化の適用

OpenMP* での並列化では、自動並列化が難しい関数やサブルーチンの呼び出しを含むタスクでの並列化(疎粒度での並列化) も可能になります。コンパイラーの自動並列化では認識できないこのような粒度の大きな並列化では、よりオーバーヘッドの小さな並列化処理が可能となります。

3.2 OpenMP* アーキテクチャー

OpenMP* のアーキテクチャーは以下のようなものになります。

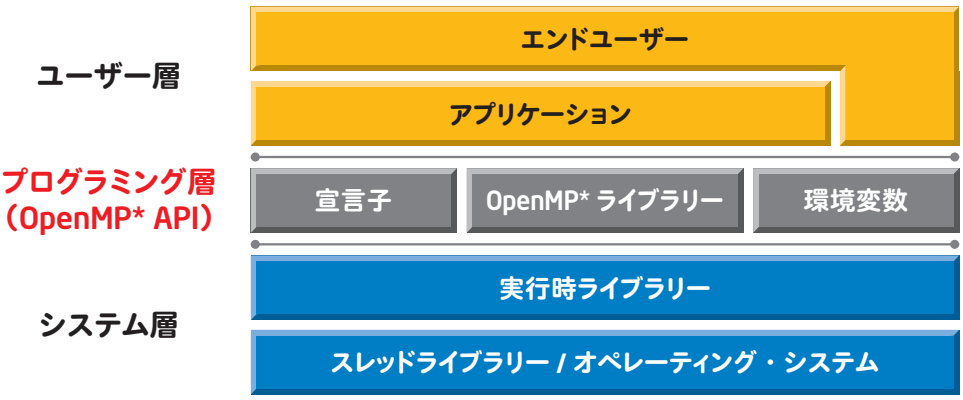


図 .4 OpenMP* のアーキテクチャー

OpenMP* API は、プログラムの並列化領域やデータ属性などを指示する宣言子と並列処理を補助するための OpenMP* ライブラリー、そして、並列処理を行う場合の実行環境を指定する環境変数から構成されます。これらは、デスクトップからスーパーコンピューターまで、統一されたひとつの API となっています。従って、OpenMP* API でプログラムの並列化を行うことで、システムのスケールアップに合わせてプログラムを書き直す必要はありません。また、OpenMP* API をサポートしているプラットフォームであれば、現在及び将来のプラットフォーム間でのプログラムの移行は非常に容易です。

OpenMP* のもっとも基本的な考え方は、プログラム中の計算負荷が大きなループに対して、その並列処理のための「指示」をコンパイラーに行うというものです。OpenMP* マルチスレッド・コンパイルをサポートするコンパイルシステムは、この指示に従って並列処理の適用をコードに対して行います。実際には OpenMP* の適用はループについてだけ行うものではありませんが、以下の例では主に for ループを並列化する事例を中心に説明を行います。

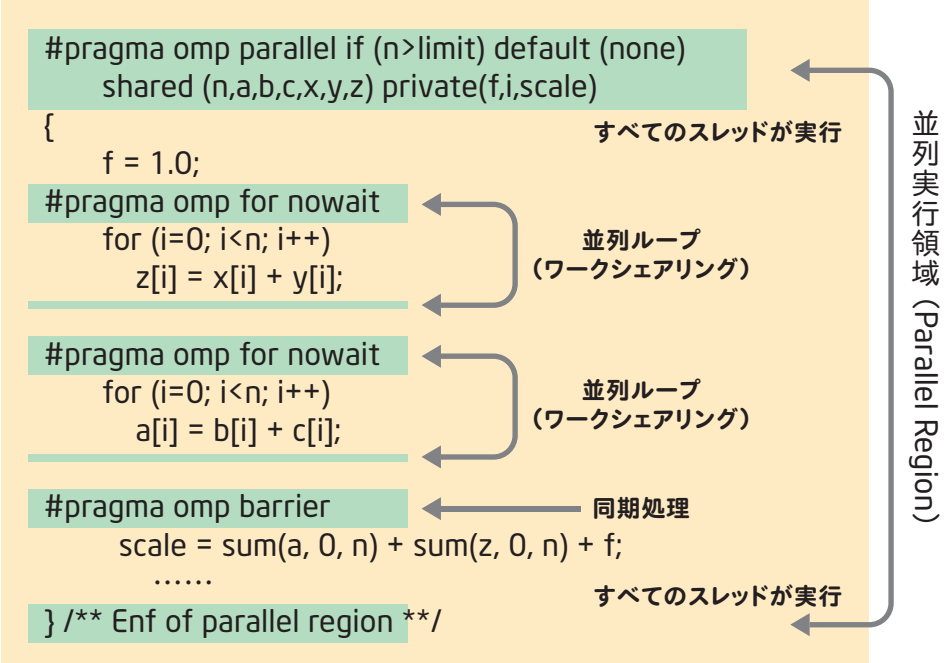


図 .5 OpenMP* プログラミングの概要

簡単なπの計算で OpenMP* プログラムを示します。このプログラムに OpenMP* 宣言子を挿入し、並列化した例は次のようになります。(Linux* でのコンパイル例と出力)

```
cat -n pi_omp.c
    1  #include <omp.h>
    2  static int num_steps = 100000; double step;
    3  void main ()
    4  {
    5     int i; double x, pi, sum = 0.0;
    6     int nthreads;
    7     step = 1.0/(double) num_steps;
    8
    9  #pragma omp parallel
   10     nthreads = omp_get_num_threads();
   11  #pragma omp parallel for reduction (+:sum) private(x)
   12     for (i=1;i<= num_steps; i++){
   13           x = (i-0.5)*step;
   14           sum = sum + 4.0/(1.0+x*x);
   15     }
   16     pi = step * sum;
   17     printf("%d Threads  PI = %f \n",nthreads,pi);
   18  }
$ icc -O3 -openmp _openmp-report2 pi_omp.c
pi_omp.c(9) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
pi_omp.c(11) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
$ setenv OMP_NUM_THREADS 2
$ time ./a.out
2 Threads  PI = 3.141593
0.021u 0.001s 0:00.10 20.0%     0+0k 0+0io 122pf+0w
```

図 .6 OpenMP* プログラムのコンパイルと実行例

OpenMP* を利用した並列プログラミングは、プログラムに対して OpenMP* で規定された宣言子を挿入し、コンパイル時にコンパイル・オプションとして /Qopenmp スイッチ(Windows*)、-openmp スイッチ(Linux*)を指定することで可能となります。OpenMP* 宣言子は、コンパイラーに対して並列化のためのヒントを与えるのでなく、明示的に並列化を指示するものです。したがって、間違った宣言子を指定してもコンパイラーはその指示に従って並列化を行います。また、データの依存性などがあっても、コンパイラーは警告メッセージやエラーメッセージを出し、その指示を無視することなく忠実に並列化を行いますので、依存性があるループなどを OpenMP* で並列化した場合には計算結果が不正になります。一方、OpenMP* を使用した場合、スレッドの生成や各スレッドの同期コントロールといった制御をユーザーが気にする必要はありません。コンパイラーは OpenMP* の指示によって並列化を行ったことをメッセージとして出力します。

3.3 OpenMP* の適用の阻害要因とその対策

OpenMP* によるマルチスレッド化の適用に際しては、いくつかの条件を満たしている必要があります。逆に、このような条件に適応できないケースでは、OpenMP* による並列化はできません。

ここでは、OpenMP* によるマルチスレッドの適用の対象となるのは、C/C++ と Fortran などでの「ループ」による反復計算部分として説明します。

- 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。
- 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ことが必要です。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。
- ループ内に外部関数の呼び出しがあるような場合には、外部関数の呼び出しによる依存関係を明確にして、関数のデータのスコーピングを行う必要があります。

4. OpenMP* によるマルチスレッド・プログラミング

4.1 宣言子の書式

OpenMP* 宣言子は、#pragma として指定します。これは、OpenMP* コンパイルを無効とする場合や OpenMP* をサポートしていない処理系の場合は、コメント行とみなされます。宣言子は、次のように行の先頭に #pragma と書き、次に宣言名、宣言句 1 などの指定を行います。

```
#pragma omp construct [clause [clause]…]
```

4.2 OpenMP* ライブラリー

OpenMP* API では、並列処理の補助を行う関数として以下のような OpenMP* ライブラリーが用意されています。

環境変数	説 明	逐次実行領域での呼び出し	並列実行領域での呼び出し
call omp_set_num_threads(integer)	並列実行領域で使用するスレッド数を設定する	スレッド数の設定	呼び出し不可
integer omp_get_num_threads()	並列実行領域のスレッド数を返す	1	スレッド数
integer omp_get_max_threads()	最大のスレッド数を返す	OMP_NUM_THREADS の設定値	
integer omp_get_thread_num()	並列スレッドの番号を 0 から返します	0	各スレッド番号
integer omp_get_num_procs()	プログラムで使用可能なプロセッサー数を返す	システムの物理 CPU 数	
logical omp_in_parallel()	並列実行中であれば真を返す	.FALSE.	.TRUE.
call omp_set_dynamics (logical)	スレッド数の動的制御有効、無効の設定		呼び出し不可
logocal omp_get_dynamic()	スレッド数の動的制御の判定	有効な場合には真を返す	
call omp_set_nested(logical)	ネストされた並列実行領域の有効、無効の設定		呼び出し不可
logical omp_get_nested()	ネストされた並列実行領域の判定	有効な場合には真を返す	

表 .1 OpenMP* 補助関数（一部のみ掲載）

これらの OpenMP* 関数を利用するためのインクルード・ファイルは、

```
#include “omp.h”
```

で指定できます。

表 .1 に記述したように、これらの関数は呼び出し可能な場所が決まっています。また、呼び出された場所によって返す値などが変わりますので注意が必要です。

1. データ属性の指定などを行うもので、英文で clause と記述された部分です。日本語マニュアルでは、句や節などと訳されます。このテキストでは、宣言句という名称でこの属性指定の部分を記述しました。

4.3 OpenMP* 宣言子一覧

OpenMP* 構文構成は、以下の 5 つのカテゴリに分類可能です。

- 並列実行領域(Parallel Regions) 構文
- ワークシェアリング(Worksharing) 構文
- データ環境(Data Environment) 構文
- 同期(Synchronization) 構文
- 実行時関数 / 環境変数(Runtime functions/environment variables)

これらの構文での代表的な宣言子を以下に示します。

- 並列実行領域 (Parallel Regions) 構文
  #pragma omp parallel
- ワークシェアリング (Worksharing) 構文
  #pragma omp for
  #pragma omp sections
  #pragma omp single
- データ環境 (Data Environment) 構文
  宣言子 : threadprivate
  宣言句 : shared、private、lastprivate、reduction、copyin、copyprivate
- 同期 (Synchronization) 構文
  宣言句 : critical、barrier、atomic、flush、order、master
- 実行時関数 / 環境変数(Runtime functions/environment variables)

4.4 構造ブロック

OpenMP* 宣言子の構文は、構造ブロックと呼ばれる単位に対して適用されます。C/C++ の場合には、OpenMP* 宣言子とそれに続く一つのステートメントか、後続の {} に囲まれた範囲がこの構造ブロックになります。構造ブロックについては、この部分からの飛び出しや飛び込みは禁止されます。ただし exit() 文の実行のための飛び出しは可能となります。

```
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto more;
}
   printf(" All done ¥n");
```

構造ブロック

```
      if(go_now()) goto more;
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto done;
   ?     go to more;
}
done:     if(!really_done()) goto more;
```

非構造ブロック

図 .7 構造ブロックの定義

5. OpenMP* 宣言子の構文説明

5.1 parallel 構文（並列実行領域の指定）

OpenMP* による並列処理モデルの柱となるコンセプトは並列実行領域（Parallel Region）です。並列実行領域は、for ループに限らずプログラムの並列化を行うべきコードセグメントを指定することができます。並列実行領域ではワークシェアリング構造が設定され、この部分は全てのスレッドによって分割処理されます。また、クリティカル・セクション構造が指定された場合には、その部分は同時にただ一つのプロセスだけが処理を行います。Parallel 構造を構築できるのは、一つにプログラム単位（メインやサブルーチンなど）内で、GOTO 文などでの飛び出しや飛び込みの無い、一つ以上の C ステートメントから構成されるプログラム領域です。並列実行領域は、プログラム実行時には指定されてスレッド数のスレッドによって並列に実行されます。

parallel 構造の指定は、以下のようになります。{}（中括弧）で並列実行するプログラムブロックを囲みます。

```
#pragma omp parallel
{
並列実行するプログラムブロック
}
```

コンパイラーは宣言句（clause）に記述してある付加情報に基づき、並列化処理のためのコードを生成します。

OpenMP* の実行モデル

OpenMP* のプログラミングモデルでは、Fork-Join モデルと呼ばれるもので逐次実行部分（シングルスレッドの実行）と並列実行（マルチスレッドのマルチプロセッサー、マルチコアプロセッサー上での実行）が交互に切り替わって実行されます。

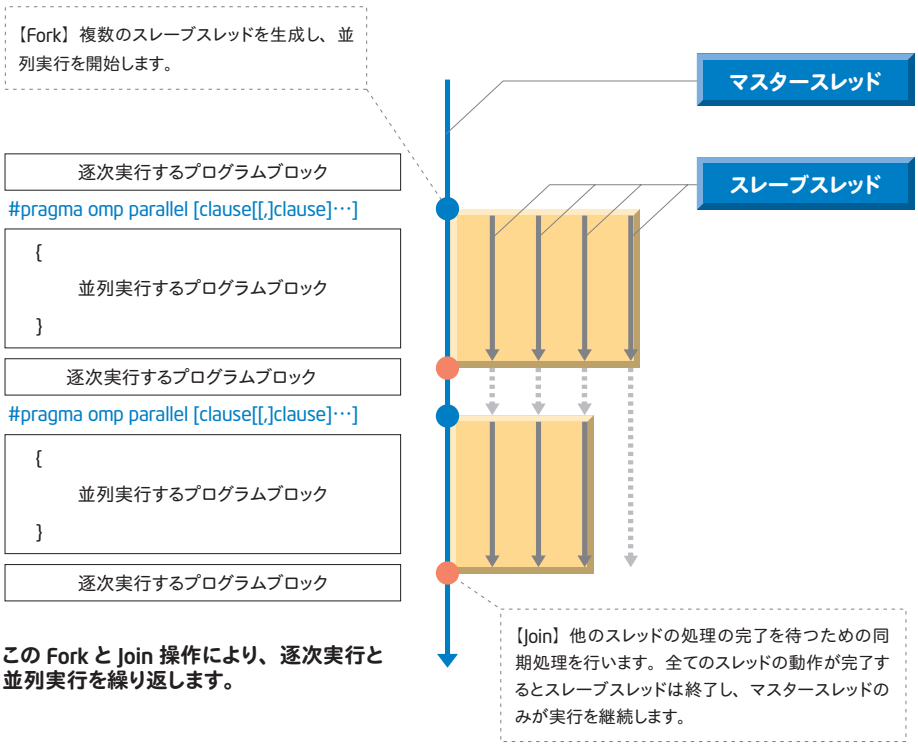


図 .8 ‘Fork-Join’ モデルでの並列処理

1) OpenMP* の処理では、最初にマスタースレッドが起動されてプログラムの実行を開始します。このマスタースレッドは、逐次的にプログラムの処理を行います。

2) マスタースレッドの実行が OpenMP* 宣言子 #pragma omp parallel の部分に到達すると、スレーブスレッドと呼ばれるスレッドを生成（スレッドを Fork する）し、分割されたプログラムのタスクを並列に処理します。

3) このマスタースレッドとスレーブスレッドの処理は、プログラム中での並列実行領域の終了点に到達すると終了します。この終了時には、全スレッドが各自の処理を終了するまで、先に終了したスレッドも待つことになり同期処理を必要とします。

4) 全スレッドが完了した時点（スレッドを join する）で、プログラムの実行処理は再びマスタースレッドだけが行うことになります。

このような処理を繰り返してプログラムの実行を行うので、“Fork-Join”モデルと呼ばれています。

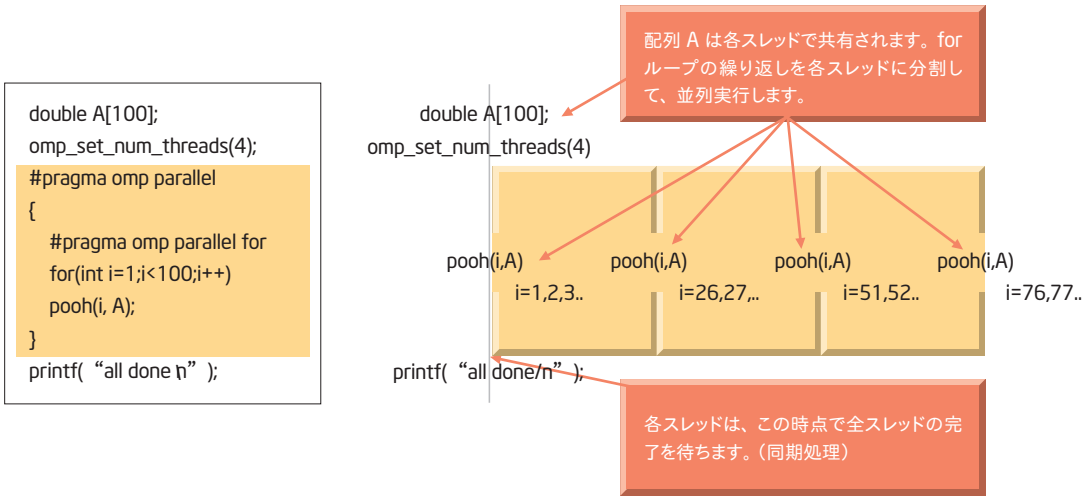


図 .9 並列実行領域の設定と並列処理

OpenMP* プログラムでは、サブルーチンなどの手続きに、parallel 構造の設定なしで OpenMP* 宣言子の指定が可能です。このような宣言子を親無し宣言子と呼んでいます。このような親無し宣言子を含むサブルーチンは、逐次実行領域からも並列実行領域からも呼び出すことが可能であり、並列実行領域から呼び出された場合だけ並列処理を行います。

次のプログラムのように並列実行領域からのサブルーチン呼び出しを含む場合を、並列実行領域の動的範囲と呼んでいます。

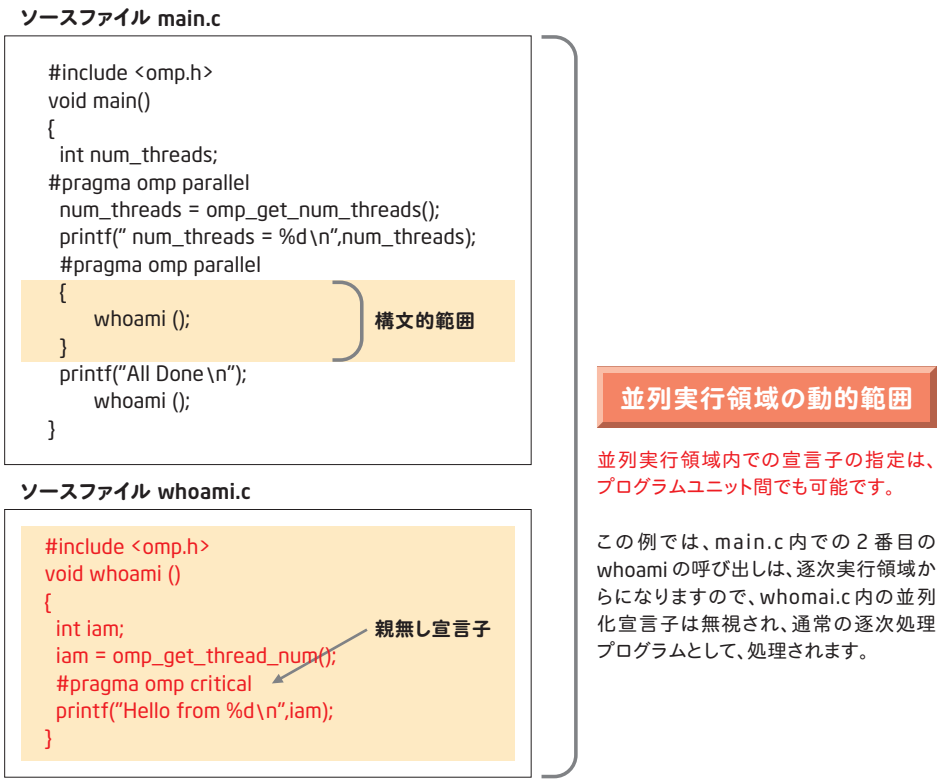


図 .10 並列実行領域の動的範囲

ここでは並列実行領域から関数を呼び出し、呼び出された関数内で OpenMP* の構文指定が行われる例を示しています。このように、並列実行内での宣言子の指定はプログラムユニット間でも可能です。ここでは、main() 内での 2 番目の whoami() の呼び出しは、逐次実行領域からの呼び出しになりますので、whoami() 内での並列化宣言子は無視され、通常の逐次処理プログラムとして処理されます。

5.2 ワークシェアリング構文

並列実行領域内の C プログラムは、全スレッドで実行されます。従って、並列処理可能な領域を parallel 構造で指定しても並列処理による計算時間の短縮はできず、並列実行領域での「ワークロード」を各スレッドに分担させることが必要になります。このような「ワークロード」の分散は、プログラム内で明示的に行なうことも可能です。OpenMP* API では、各スレッドにつけられた番号などを取得する関数が用意されています。これらの関数を使えば、例えば、次のような形でプログラムの処理の分散も可能です。

(1)逐次実行コード

```
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

(2)OpenMP* での並列実行領域の指定

```
#pragma omp parallel private(I,id,istart,iend)
{
   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   istart = id * N / Nthrds;
   iend = (id+1) * N / Nthrds;
   for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

(3)OpenMP* での並列実行領域の指定とワークシェアリング構文

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```

```
#pragma omp parallel for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```

図 .11 OpenMP* によるワークシェアリングの実装

(1) のような for ループの逐次処理は、並列実行領域内でのスレッド数でループを分割することで、個々のスレッドが分担するループの範囲 (ループ長) を決定し、それに基づいて個々のスレッドで (2) に示すような形で並列処理することが可能となります。しかし、実際には、このような明示的なプログラムは煩雑ですし、また実装も容易ではありません。OpenMP* のワークシェアリング構文では、このようなワークロードのスレッドへの分担を指示する宣言子が用意されています。この例を (3) に示してあります。

for 構文

for 構文は直後の for ループの処理を各スレッドに分割して並列実行します。形式は次のようになっています。

```
#pragma omp parallel
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

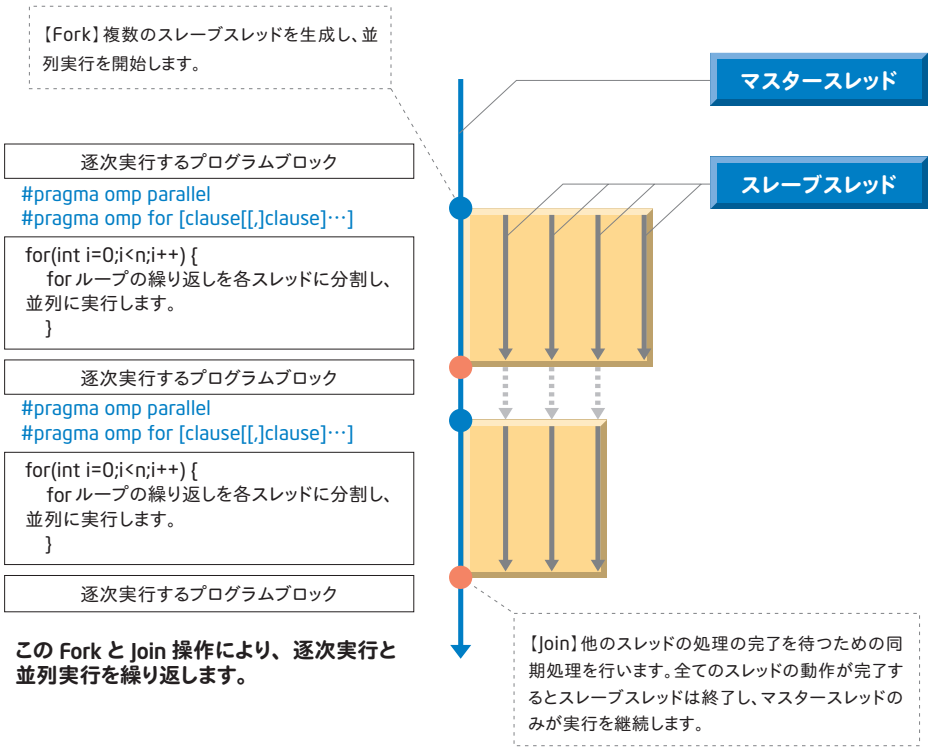


図 .12 for ループに対する OpenMP* での並列処理の適用

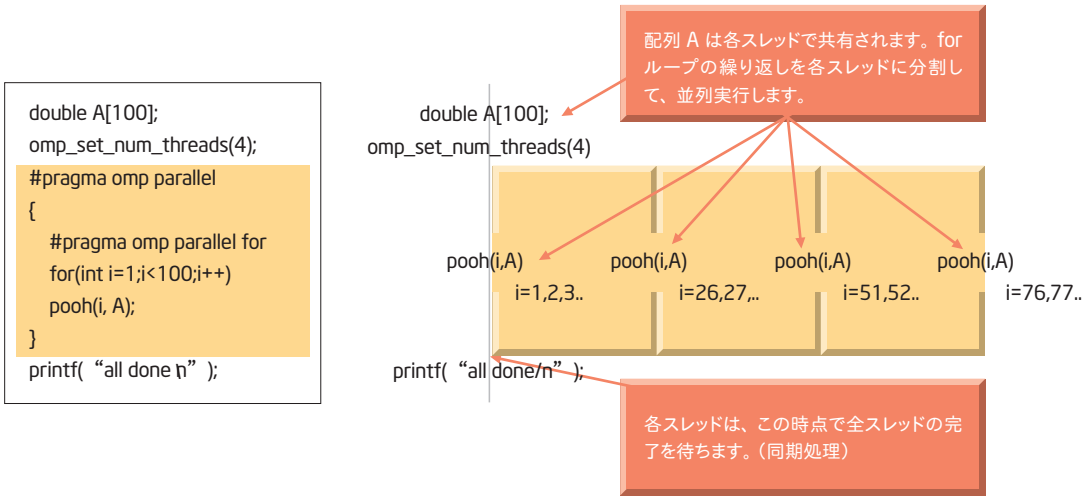


図 .13 for ループに対するワークロードとデータの分散並列処理

ループの分割については、オプションで分割方法を指定することもできます。for 構文に schedule 句を指定してこの分割方法を指示します。

```
#pragma omp parallel
#pragma omp for schedule(type[,chunk]) [clause [[,] clause]..]
     for (I=0;I<N;I++){
            NEAT_STUFF(I);
     }
```

TYPE	説 明
STATIC	繰り返し数は、chunk で指定したサイズに分割されます。分割された部分は、スレッドの番号順でラウンドロビン形式（全てのスレッドに対して、平等に巡回的に割り当てる）で各スレッドに静的に割り当てられます。
DYNAMIC	繰り返し数は、オプションで指定可能な chunk で指定したサイズに分割され、各スレッドは割り当てられた繰り返し部分を終了すると繰り返し数の次のセットが動的に割り当てられます。オプションの chunk の指定が無い場合には、1 が設定されます。
GUIDED	GUIDED を指定した場合、chunk のサイズは繰り返し数を各スレッドに割り当てた毎に指数関数的に減少します。オプションの chunk の指定には、割り当てごとの最小繰り返しの設定が可能です。オプションの chunk の指定が無い場合には、1 が設定されます。
RUNTIME	スケジュールの指定を実行時に指定します。オプションの type 及び chunk は環境変数 OMP_SCHEDULE の設定によって実行時に決定されます。環境変数 OMP_SCHEDULE がない場合には、SCHEDULE(STATIC) が設定されます。

表 .2 ループ分割のスケジューリング・オプション

各パラメータの指定時のループの分割を模式的に記述すると次のようになります。ここでは、4 プロセッサーでの並列処理における各スケジューリングの各プロセッサーへの処理の分配を示しています。

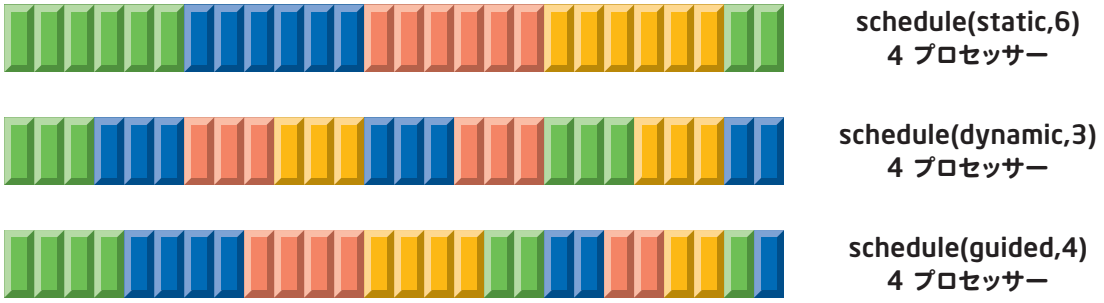


図 .14 for ループの並列処理におけるスケジューリング例

do while ループなど繰り返し数を持たないループは、指定できません。nowait パラメーターを記述した場合、個々のプロセスは do 構文の実行後、ただちに後に続く処理を行います。nowait を指定しない場合は、プロセスは他の全てのプロセスが do 構文を終了するまで待ち状態になります。

sections 構文

sections 構文では、section 宣言子で囲まれた個々のコードブロックがひとつのスレッドに割り当てられて処理を行います。

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
      並列実行するコードブロック
#pragma omp section
      並列実行するコードブロック
#pragma omp section
      並列実行するコードブロック
}
```

宣言句 nowait を記述した場合、個々のプロセスは sections 構文の実行後、ただちに後に続く処理を行います。nowait を指定しない場合は、プロセスは他の全てのプロセスが sections 構文を終了するまで待ち状態になります。section 宣言子は sections 宣言子の外に記述することはできません。section から、他の section へ分岐、移動することはできません。

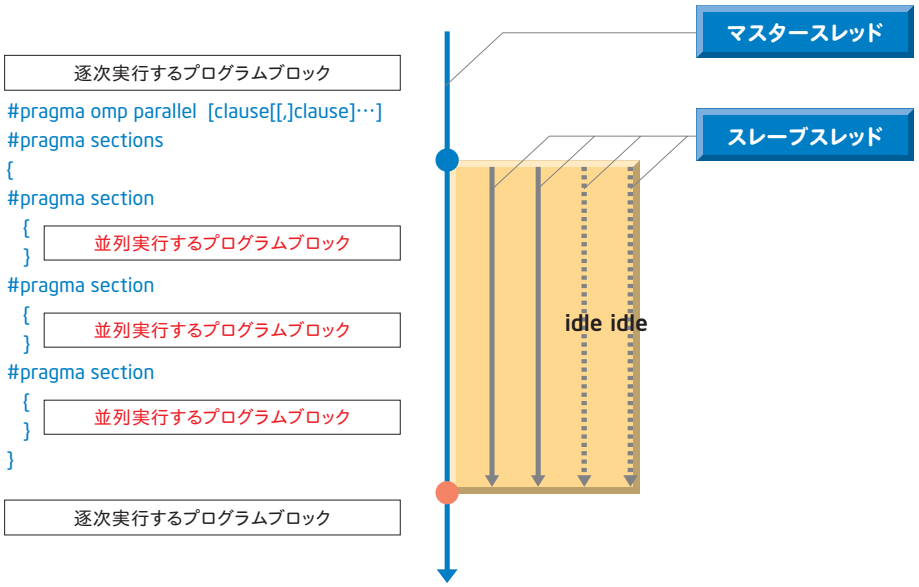


図 .15 sections 構文の実行

実行を指定したスレッド数に対して、section 宣言子で囲まれるブロックが少ない場合には、仕事が割り当てられないスレッドができます。逆に、section 宣言子で囲まれるブロックが多い場合には、先の仕事を終えたスレッドが残りのブロックを分担して実行します。

single 構文

single 構文は並列実行領域の中でのみ記述できます。single 構文で指定されたコードブロックは、ただ一つのプロセスでのみ実行されます。nowait が指定されていない場合は、並列実行領域終了点のところで同期します。シンタックスは以下の通りです。

```
#pragma omp single [clauses]
        構造ブロック
```

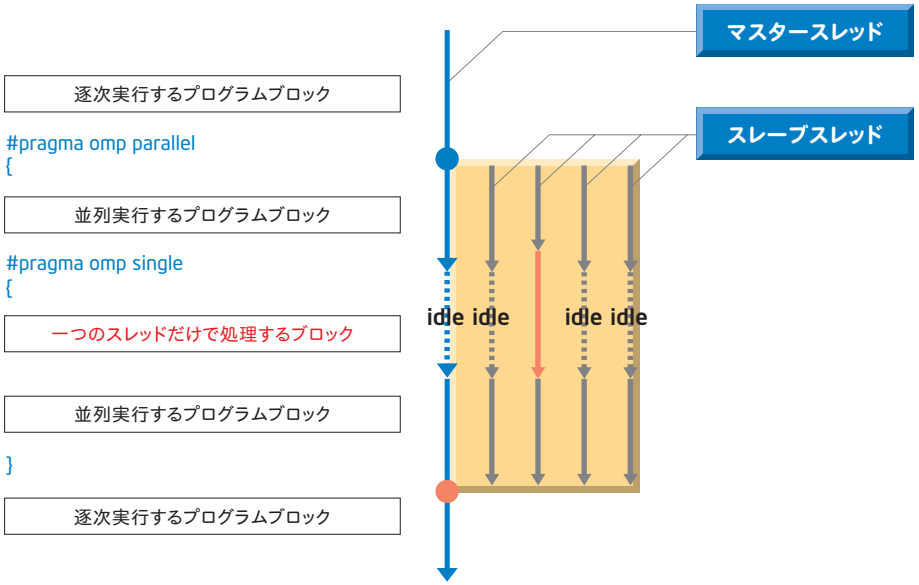


図 .16 single 構文の実行

この single 構文は、sections 構文で section 宣言子の指定がひとつしかない場合と同等です。

結合構文

OpenMP* では、parallel 構文とワークシェアリング構文を結合した次のようなショートカット構文が定義されています。

parallel for 構文

parallel sections 構文

parallel workshare 構文

これらは parallel 宣言子の直後にそれぞれのワークシェアリング宣言子を指定した場合と同じ意味を持ちます。

5.3 データ環境

いままで紹介した OpenMP* 宣言子には、宣言句（clause）という表示がありました。この宣言句は、OpenMP* 構文機能に対する付加情報を与えるものです。この与えられた付加情報によって、コンパイラーは並列実行のための並列化を行ないます。OpenMP* 宣言子と指定可能な宣言句の関係は、以下のようになります。

	parallel	do	sections	workshare	single	parallel do	parallel sections	parallel workshape
if	●					●	●	●
schedule		●				●		
private	●	●	●	●	●	●	●	●
share	●					●	●	●
default	●					●	●	●
firstprivate	●	●	●	●	●	●	●	●
lastprivate		●	●			●	●	
copyin	●					●	●	●
copyprivate					●			
reduction	●	●	●			●	●	●
ordered		●				●		
nowait		●	●	●				
num_threads	●					●	●	●

表 .3 OpenMP* 宣言子と宣言句の指定

ここで、データスコープ属性を指定する宣言句は、並列実行領域での変数の取り扱いについての記述を行なうものです。データスコープとは、並列プログラム内での変数がスレッド毎に独立の記憶域を持ち、それぞれ異なった値を持つものとなるか、プログラム内で一つの記憶域で管理されるかを決定するものです。

		SHARED	PRIVATE
		全スレッドがアクセス可能	スレッド毎に独立にデータを割り当て
グローバル（大域）変数	全てのプログラムユニット内でアクセスが可能な変数	▪ static 変数 ▪ ファイルスコープ変数	▪ OpenMP* 宣言子での threadprivate 指定
ローカル（局所）変数	プログラムユニット内だけでアクセスが可能な変数	▪ OpenMP* でのデフォルト ▪ OpenMP* 宣言子での shared 指定	▪ OpenMP* 宣言子での private 指定 ▪ for ループの反復変数 ▪ stack 変数

表 .4 データスコープ : 並列実行領域内での変数の取り扱い

```
#include <omp.h>
float a[100],b[100];
int stride;
main()
{
int i,s,iam,nthreads;
#pragma omp parallel
  nthreads = omp_get_num_threads();
#pragma omp master
  stride = 100 / nthreads;
#pragma omp parallel for private (iam)
  for ( i = 0; i < nthreads; i++) {
    iam = omp_get_thread_num(); s= iam * stride;
    f(s);
  }
}

f(int from)
{
  int i;
  float tmp;
  for( i=from; i<from+stride; i++ ) {
     tmp = a[i];
     a[i] = b[i];
     b[i] = tmp;
  }
}
```

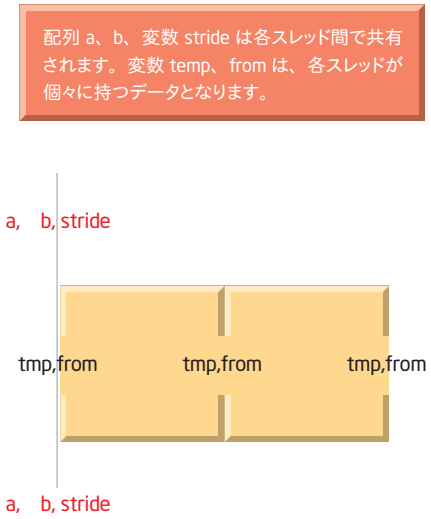


図 .17 並列実行領域での変数の取り扱い

ここでは、並列領域を実行する際のデータ環境の制御について説明します。

threadprivate 宣言子

この宣言子はスレッド間でのデータ環境を定義するための宣言文であり、グローバル変数をスレッド間では独立とし、同じスレッド間では共有する枠組みを提供します。

この宣言子はプログラムユニットの宣言部分のセクションに記述しなければなりません。個々のスレッドは指定されたグローバル変数のコピーを保持します。したがって、それぞれのグローバル変数に行われたデータの更新は他のスレッドのデータには反映されません。プログラムの逐次実行領域、および master 宣言子指定の領域では threadprivate で指定されたグローバル変数へのアクセスはマスタースレッドのグローバル変数が使用されます。

最初の並列実行領域を実行する際には、threadprivate のデータの内容は宣言句 copyin が parallel 宣言子に指定されていない場合未定となります。

```
#pragma omp parallel (list)
```

データスコープ宣言句

```
private (list)
```

マルチスレッド・プログラミングでは、for ループの制御変数のように、スレッド毎に異なる値をもつ変数が必要になります。private 宣言句の list に指定された変数は、各スレッドに独立に領域が確保されます。これらの変数は、スレッド内だけでアクセスされる変数となります。private 宣言句に指定された変数は、並列実行領域でスレッドが Fork さ

れる毎に作成されます。従って、これらの変数の値は並列実行領域の開始時点では、全て値が未定義になっています。for ループ制御変数は、OpenMP* で、全て private 属性の変数と定義されます。

```
shared (list)
```

OpenMP* でのデータスコープは指定が無い場合には、プログラム内で一つの記憶域として管理される shared 属性となります。

```
default (shared | private | none)
```

default 宣言句は、並列領域内にある全ての変数のデータ属性に対して、省略時（変数の定義がない場合）の取り扱いを private 属性とするか、shared 属性とするかを指定します。

default (private) を指定した場合、threadprivate で指定されたグローバル変数を除き、並列領域にある全ての変数をスレッドにプライベートなコピーを作成します。これは一つ一つの変数を private(list) に記載した場合と同じになります。

default (shared) を指定した場合、並列実効領域にある全ての変数は全てのスレッドによって共有されます。明示的に default の指定をしていない場合は、default (shared) となります。

default (none) を指定した場合、private 属性と shared 属性のスコーピングに関しては何ら規定値を持たないことを指示します。この場合、並列領域にある全ての変数について、private、shared、firstprivate、lastprivate、あるいは reduction のスコーピング属性を与える必要があります。

変数は private、shared、firstprivate、lastprivate、あるいは reduction の指定によって、デフォルトの指定から除外することも可能です。したがって、次のような指定方法も問題ありません。

```
#pragma omp default (private)
#pragma firstprivate(i) shared(x)
#pragma shared(s) lastprivate(i)
```

```
firstprivate ( list )
```

firstprivate は private のスーパーセットです。list に記述された変数は private の取り扱いと同じになりますが、その値は並列実行領域の開始時に逐次実行部分の変数の持つ値を各スレッドの private 変数に値がコピーされます。

```
     b = 23.0;
      . . . . .
#pragma omp parallel firstprivate(b), private(i,myid)
   {
      myid = omp_get_thread_num();
      for (i=0; i<n; i++){
         b += c[myid][i];
      }
      c[myid][n] = b;
   }
```

```
lastprivate ( list )
```

並列実行領域を終了した時点では、並列実行領域の private 変数の値は不定になっています。lastprivate は private のスーパーセットです。list に記述された変数は private と同様な取り扱いとなりますが、その値は並列実行の際の最後のイタレーションで保持された値を持ちます。section 構文の指定の場合には、最後の section における値がコピーされます。

```
copyin (list)
```

copyin は、threadprivate 宣言子で宣言されたグローバル変数に対して、firstprivate 宣言句に指定された変数と同様にマスタースレッド（逐次実行部分）の値をコピーします。

```
copyprivate (list)
```

copyprivate は、指定された private 属性をもつ変数を他のスレッドにブロードキャストします。この宣言句は、データの読み込み時などにそのデータの各スレッドにブロードキャストするのに便利な機能を提供します。

5.4 同期構文

同期構文には、master、critical、barrier、atomic、flush、ordered の 6 つの宣言子があります。

master 宣言子

master 宣言子で指定されたブロックはマスタースレッドのみで実行されます。

```
#pragma omp master
  実行するコードブロック
```

マスター以外のスレッドはこのコードブロックをスキップし、次の実行に移ります。master 宣言子エントリ、または出口には同期ポイントは設けられません。

critical 宣言子

クリティカル・セクションは、指定したコードブロックを同時に1つのプロセスだけが実行することを指示するものです。同コードブロックを実行する他のプロセスはクリティカル・セクションを実行しているプロセスが終了するのを待たなければなりません。

critical 宣言子のシンタックスは以下の通りです。

```
#pragma omp critical   [(name)]
          実行するコードブロック
```

クリティカル・セクションに名前を付けることができます。同じ名前を持つクリティカル・セクションは、同一のものとしてクリティカル・セクションにまたがっての排他制御が可能となります。省略した場合には、全て同じものとしてマッピングされます。

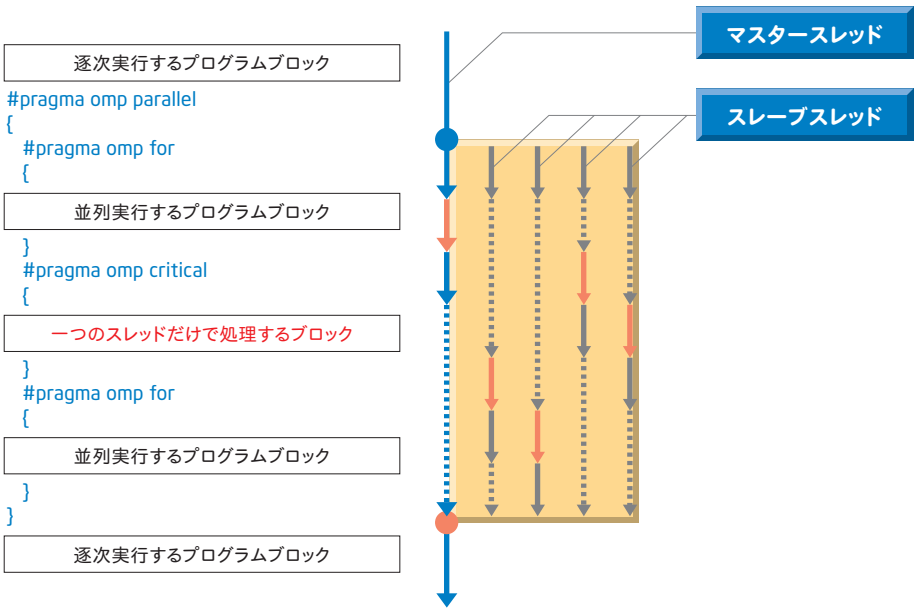


図 .18 クリティカル・セクションの実行

barrier 宣言子

バリアーは、全てのプロセスを同期させる為の指示を行います。個々のスレッドは、全てのスレッドがバリアポイントに到着するまでその後の実行を開始しません。全スレッドがバリアポイントに到達することにより同期がとれます。シンタックスは以下の通りです。

```
#pragma omp barrier
```

atomic 宣言子

atomic 宣言子は、同時に複数のスレッドからアップデートされる可能性のあるメモリーエリアの更新を排他的に行います。

```
#pramga omp atomic
  Expression-statement
```

このディレクティブは、ディレクティブに続く 1 行のステートメントだけに適用されます。ステートメントは以下のいずれかのフォームである必要があります。

x = x op expr,　x = expr op x,　x = intr (x, expr) ,　x = intr(expr, x)

x は組み込み関数で利用可能なデータ形式を持つスカラー変数で、expr は x を参照しない演算式となり、intr は max、min、iand、ior、ieor のいずれかの組み込み関数、op は +,*,-,/,.and.,.or.,.eqv.,.neqv. のいずれかの演算となります。

xに対するロード、ストアが atomic の対象となります。演算自身が atomic の対象となるわけではありません。critical 構文でもこの atomic 構文と同じ処理が可能です。実際には、critical 構文の方が柔軟な排他制御が可能となります。

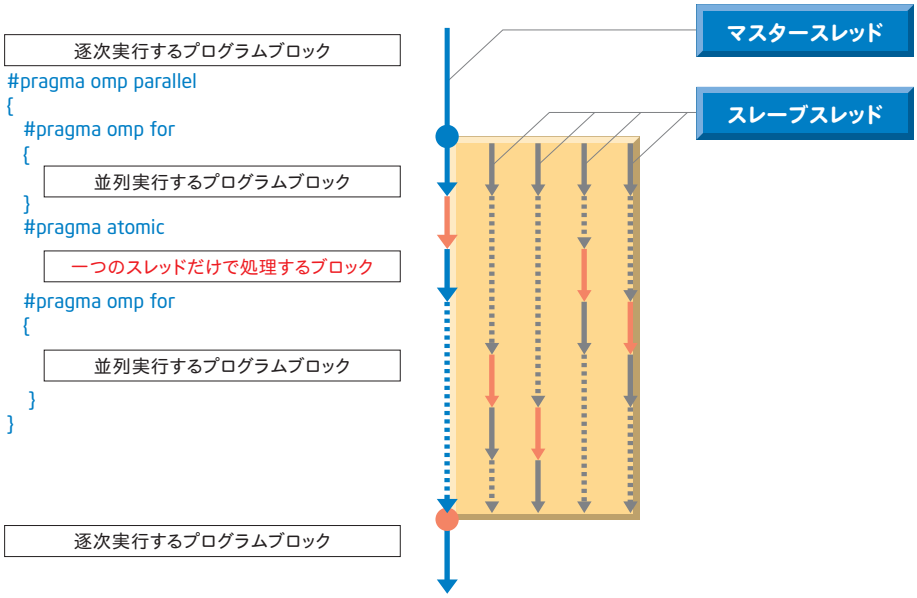


図 .19 atomic アップデートの指定

flush 宣言子

コンパイラーは、flush が指定されたポイントで、スレッドの変数に関連する一貫性の保持のための共有変数に関する同期処理を指示します。

```
    #pragma omp flush(list)]
```

ここでの list には、変数名をカンマ（,）で区切って指定します。省略すると以下の全ての共有変数について同期が取られます。

グローバル変数

save 属性のないローカル変数で、そのアドレスが他のサブプログラムへ引き渡されるなど、アドレスが参照されるもの

ローカル変数で、サブプログラムにあるパラレル領域で shared と宣言されるもの

ダミーの引数

全てのポインター参照

list には必要な変数をコンマ (,) で区切って記述します。list の指定がない場合は全ての変数がフラッシュされます。

order 宣言子

order で囲まれたコードブロックは、そのコードを逐次実行した場合と同様に実行されます。

```
#pragma omp order
実行コードブロック
```

order 宣言子は for 指示構文または parallel for 指示構文中にだけに記述できます。また、その時、for 宣言子には、order 宣言句も指定されている必要があります。

reduction 宣言句

reduction 宣言句は、for 構文に指定され reduction 演算を並列処理するものです。

```
reduction( {op| intr }: list )
```

list で指定された変数は、組み込み関数で利用可能なデータ形式を持つスカラー変数と配列です。また、その変数のスコーピング属性は shared 属性である必要があります。l ist で指定された変数はそれぞれのスレッドのためにプライベートのコピーが作成され、その値はオペレーターの種類によって特定の値で初期化されます。intr は max、min、iand、ior、ieor のいずれかの組み込み関数が指定可能で、op は +,*,-,/,.and.,.or.,.eqv.,.neqv. のいずれかの演算となります。

reduction を行ったループの後で共有のリダクション変数は、そのオリジナルの値とそれぞれのスレッドが持つプライベートのリダクション変数の最終値を使用して記述された op の演算を行い、アップデートされます。リダクション演算は引き算を除き結合可能であるため、コンパイラーは最終値の計算について自由に再構成することができます。

共有変数の値は最初のスレッドが演算に到達すると未定義となり、リダクション演算が終了するまでその状態となります。通常、演算処理は reduction 構文の最後で完了します。ただし、nowait が指定された reduction 構文では、全てのスレッドがリダクション演算を終了したことを確認するために barrier 同期を行うまでは、共有変数の値は未定義のままとなります。

reduction 宣言句は通常、次のようなワークシェアリング構文で使用されることを意図しています。(最初に示したπの計算プログラムをもう一度示します)

```
#include <omp.h>
static long num_steps = 100000;
double step;
void main ()
{       int i;          double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=1;i<= num_steps; i++){
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

次のテーブルは、リダクションに指定可能な演算子および組み込み関数の一覧とその初期値を示しています。

演算子	初期値
+	0
*	1
-	0
.and.	全て 1

演算子	初期値
.or.	0
max	1
min	0
//	全て 1

表 .5 reduction 演算

同じ種類の reduction 宣言句には複数の変数を記述できます。異なった種類の reduction 宣言句を指定する場合は、以下のように記述します。

```
#pragma omp for reduction (+:a,y) reduction (.or.:am)
```

5.5 その他の宣言句

if 宣言句

if 句は parallel 構文に対して、その構文を実行時に有効とするか無効とするかを指定するためのものです。if 宣言句の条件式が TRUE の場合のみ並列実行構文が有効になり、並列実行がなされます。

nm_threads 宣言句

nm_threads 宣言句によって、parallel 構文を処理するスレッド数を指定することが可能です。

nowait 宣言句

OpenMP* の指示構文である do 構文、sections 構文、single 構文の終了時に、並列実行領域のスレッド全ての実行が終了するまで各スレッドは待ち状態になります。nowait 宣言句を指定した場合には、各スレッドは待ち状態にならず次の処理を開始します。

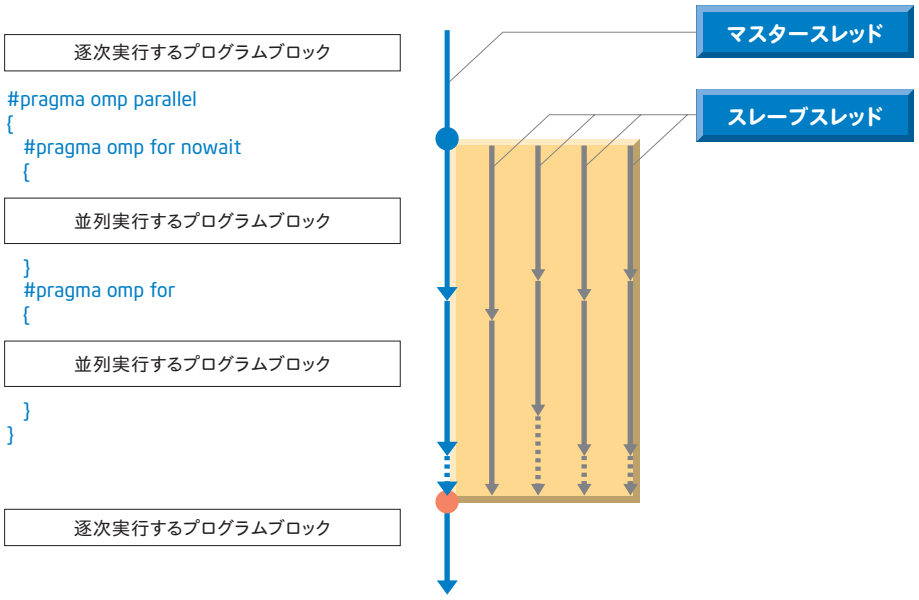


図 .19 nowait 指定時のスレッド動作

nowait は次の例のようにループ間の実行に依存性（ループ間でのデータの相互参照）がない場合に指定することで、並列実行の効率化を図ることが可能です。

```
#pragma omp parallel
{
#pragma omp for nowait
   for( i=from; i<from+stride; i++ ) {
      a[i] = c * b[i];
   }
#pragma omp for nowait
   for( i=from; i<from+stride; i++ ) {
     x[i] = y[i] * y[i];
   }
}
```

5.6 実行時関数 / 環境変数

OpenMP* によるマルチスレッド・プログラムの実行を行なう場合、環境変数によってプログラムの実行を制御することが可能です。

環境変数	説 明
OMP_NUM_THREADS	スレッド数の動的調整が有効になっている場合、環境変数の値は、使用するスレッドの数の上限として解釈されます。 例：setenv OMP_NUM_THREADS 4（Linux） set OMP_NUM_THREADS=4（Windows）
OMP_SCHEDULE	環境変数のデフォルト値は、処理系に依存します。指定では、type[,chunk]の形式で指定し、type には、STATIC/DYNAMIC/GUIDED が指定可能です。chunk の設定は、オプションです。chunk が設定されていない場合には、STATIC スケジュールの場合を除き、値 1 が使用されます。STATIC スケジュールでは、デフォルトの chunk は、ループカウントを、そのループに適用されるスレッドの数で割った値に設定されます。 例：setenv OMP_SCHEDULE "dynamic"（Linux） set OMP_SCHEDULE=dynamic（Windows）
OMP_DYNAMIC	値が TRUE に設定されている場合、並列実行領域の実行に使用されるスレッドの数は、システムのロードなどによって、システムが実行時に調整します。値がFALSE に設定されていると、この動的調整は無効になります。 例：setenv OMP_DYNAMIC TRUE（Linux） set OMP_DYNAMIC=TRUE（Windows）
OMP_NESTED	値が TRUE に設定されていると、ネストされた並列実行は有効になり、値がFALSE に設定されていると、ネストされた並列実行は無効になります。デフォルト値は FALSE です。 例：setenv OMP_NESTED TRUE（Linux） set OMP_NESTED=TRUE（Windows）

表 .6 OpenMP* 環境変数

6. OpenMP* の適用について

OpenMP* によるマルチスレッド化は、プログラマーから並列処理を行うための様々な作業を解放します。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、必要なところに 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。

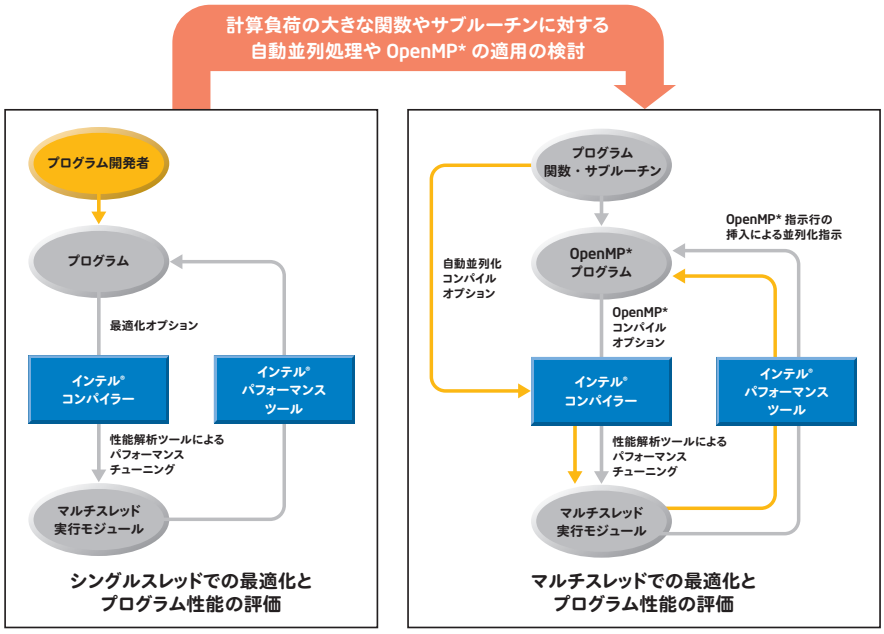


図 .21 OpenMP* プログラムの開発フロー

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、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。

OpenMP* 適用のためのステップは次のようになります。

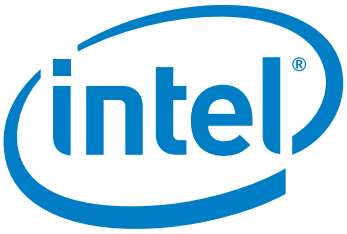
1. インテル® VTune™パフォーマンス・アナライザーを使いプログラムの動作の詳細な解析を行います。ホットスポットを見つけることが並列処理では必要になります。
2. ホットスポットに対して OpenMP* 宣言子の適用などを検討します。データの依存関係などのために並列化できない部分などについては、依存関係の解消のために行うプログラムの変更を行います。この時、他のハイレベルの最適化手法（ソフトウェア・パイプラインやベクトル化）などに影響を与えるときがあります。この並列化による他のハイレベルの最適化の阻害は避ける必要があります。そのためには、並列化の適用時と非適用時の性能を比較検討する必要もあります。
3. 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® マス・カーネル・ライブラリー（MKL）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。

7. おわりに

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、必要ではありますが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。

パフォーマンスに対する高度の要求に答える形で、プロセッサーは高速化の一途をたどってきました。しかし、現在プロセッサーとメモリーの性能格差が広がるにつれて、様々なアプリケーションにおいてメモリー・レイテンシーがパフォーマンスの面でのボトルネックになっています。また、プロセッサーの消費電力と発熱量も大きな問題です。このような状況を打破するためにも、デュアルコアやマルチコアといった最新のプロセッサーの実装技術が求められています。OpenMP* での並列化は、このような時代の要求に応えるものです。コンパイラーツールの更なる進化によって、今後、これらの機能はさらに強化されていきます。

様々なレベルでの並列処理において、それらの並列処理技術を緊密に結合することで、アプリケーションの性能を大幅に向上させることが可能です。



分散並列処理、スーパーコンピューター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/

2006 年 3 月

526J-001
JPN/0603/PDF/SE/DEG/KS