C 言語による OpenMP 入門

黒 田 久 泰

東京大学情報基盤センター

1. はじめに

OpenMP は非営利団体 OpenMP Architecture Review Board（ARB）によって規定されている業界標準規格です。共有メモリ型並列計算機用のプログラムの並列化を記述するための指示文、ライブラリ関数、環境変数などが規定されています。OpenMP を利用するには、OpenMP に対応したコンパイラが必要となりますが、現在、多くのコンパイラが OpenMP に対応しています。

OpenMP の特長としては、下記のような点が挙げられます。

・並列プログラム（マルチスレッドプログラム）が簡単かつ短いコード量で記述できる。
・異なるシステム間でソースプログラムを共通化できるので移植性が高い。
・逐次プログラムから段階的に並列化していくことが可能である。
・同一のソースプログラムで並列環境と非並列環境を共存できる。
・コンパイラの自動並列化機能に比べてプログラムの高速化を達成し易い。

OpenMP は下記のようにいくつかのバージョンがあります。現在、広く普及しているのは Version 2.0 のものですので、本記事も Version 2.0 をもとに説明します。OpenMP は現在のところ Fortran 言語と C/C++言語で利用できますが、本記事は C 言語（C++言語でも使い方は同じ）を取り扱います。

1997 年 10 月	OpenMP Fortran API 1.0
1998 年 10 月	OpenMP C/C++ API 1.0
1999 年 11 月	OpenMP Fortran API 1.1
2000 年 11 月	OpenMP Fortran API 2.0
2002 年 3 月	OpenMP C/C++ API 2.0
2005 年 5 月	OpenMP Fortran C/C++ API Version 2.5
2007 年 10 月	OpenMP Fortran C/C++ API Version 3.0 Draft

OpenMP では、プラグマ・ディレクティブ（#pragma）と呼ばれるコンパイラへの命令文を用いて記述します。OpenMP をサポートしていないコンパイラでは単にコメント行とみなされます。

なお、複数ノードにまたがる並列プログラムは OpenMP では記述できません。MPI のような通信ライブラリを用いた並列プログラミングが必要となります。しかしながら、1 ノードが複数プロセッサで構成されているような並列計算機であればノード内を OpenMP で並列化を行い、ノード間を MPI で並列化するといった方法で高性能なアプリケーション開発を行うことができます。

OpenMP について詳しく知りたい方は OpenMP のホームページ（http://www.openmp.org/）をご覧ください。

2. OpenMP におけるキーワード一覧

OpenMP の全体像を理解するために、指示文、指示節、実行時ライブラリ関数、環境変数にそれぞれどうようなものがあるのかを最初に示します。各詳細については第 4 章以降で説明します。

2.1 OpenMP の指示文

OpenMP の指示文は、プログラム内で並列化を行う場所に挿入して並列化の方法を指定します。プラグマ（#pragma）によって記述され、必ず「#pragma omp ・・・」のような形をとります。

並列リージョン指示文	同期に関する指示文
#pragma omp parallel	#pragma omp single
処理分散指示文	#pragma omp master
#pragma omp for	#pragma omp critical
#pragma omp sections	#pragma omp atomic
	#pragma omp barrier
	#pragma omp ordered
	#pragma omp flush
結合指示文（並列リージョン指示文と処理分散指示文を結合したもの）	
#pragma omp parallel for	
#pragma omp parallel sections	
データ属性指示文	
#pragma omp threadprivate	

「#pragma omp sections」指示文は特別に次のような宣言子を利用します。

section 宣言子
#pragma omp section

2.2 OpenMP の指示節

OpenMP の指示節は必ず OpenMP の指示文とともに使われ、「#pragma omp ・・・ 指示節」のような形をとります。

スコープ指示節	その他の指示節
private	ordered
firstprivate	schedule
lastprivate	if
shared	num_threads
default	nowait
reduction	
copyin	
copyprivate	

2.3 実行時ライブラリ関数

OpenMP では指示文以外にも役に立つ実行時ライブラリ関数が提供されています。これらの関数を一切使わなくても並列化は行えますが、より高度な並列化を行う際に利用します。実行時ライブラリ関数を利用する場合には、プログラムの先頭部分に「#include <omp.h>」を記述して OpenMP 用のヘッダファイル omp.h を読み込む必要があります。

実行環境ルーチン	ロックルーチン
omp_set_num_threads(int)	omp_init_lock(omp_lock_t *)
omp_grt_num_threads()	omp_destroy_lock(omp_lock_t *)
omp_get_max_threads()	omp_set_lock(omp_lock_t *)
omp_get_thread_num()	omp_unset_lock(omp_lock_t *)
omp_get_num_procs()	omp_test_lock(omp_lock_t *)
omp_in_parallel()	**ネスト可能なロックルーチン**
omp_set_dynamic(int)	omp_init_nest_lock(omp_nest_lock_t *)
omp_get_dynamic()	omp_destroy_nest_lock(omp_nest_lock_t *)
omp_set_nested(int)	omp_set_nest_lock(omp_nest_lock_t *)
omp_get_nested()	omp_unset_nest_lock(omp_nest_lock_t *)
時間計測ルーチン	omp_test_nest_lock(omp_nest_lock_t *)
omp_get_wtime()	
omp_get_wtick()	

OpenMP のデータ型（OpenMP で定義されているデータ型は下記の 2 つです）

omp_lock_t　　ロック情報を格納する型（ロックルーチンで使用）

omp_nest_lock_t ロック情報を格納する型（ネスト可能なロックルーチンで使用）

2.4 環境変数

プログラムを実行する際に OpenMP で定義されている環境変数を設定することで、プログラムで使用する際のスレッド数などを指定することができます。

実行時の動作環境に関するもの

OMP_NUM_THREADS

OMP_SCHEDULE

OMP_DYNAMIC

OMP_NESTED

OMP_WAIT_POLICY（OpenMP 3.0 の機能）

OMP_STACK_SIZE　（OpenMP 3.0 の機能）

3. コンパイルと実行方法

3.1 コンパイラとコンパイルオプション

OpenMP に対応しているコンパイラとよく使われるコンパイルオプションを示します。プログラムの実行がうまくいかない場合には、最適化のレベルを下げる必要があります。

コンパイラ	コマンド名	オプションの説明	推奨する最適化
gcc	gcc (*1)	man gcc	-O3
日立コンパイラ	cc	man cc	-Os +Op
Intel コンパイラ	icc	icc -help	-fast
PGI コンパイラ	pgcc	pgcc -help	-fastsse -O4 (*2)
Sun Studio	cc	cc -flags	-fast
Visual Studio	cl.exe	cl.exe /help	/Ox
IBM XLC	xlc	xlc -help	-O5
PathScale	pathcc	pathcc -help	-O3

(*1) gcc はバージョン 4.1 から OpenMP に対応。コマンド名が gcc41 や gcc42 の場合もある。
(*2) PGI コンパイラで-fastsse だけだと-O2 が設定されるため、-O3 や-O4 を追加で設定する。

OpenMP の規格ではコンパイルオプションの記述方法までは規定されていません。そのためコンパイラごとに OpenMP を有効にしてコンパイルする方法が異なります。OpenMP が使われているプログラムを、OpenMP 並列化のみ、OpenMP を無効にして自動並列化のみ、OpenMP 並列化と自動並列化の両方を行う、のそれぞれの場合のコンパイルオプションを示します。

コンパイラ	OpenMP 並列化のみ	自動並列化のみ	OpenMP＋自動並列化
gcc	-fopenmp -lgomp	なし	なし
日立コンパイラ	-parallel -omp	-parallel	なし
Intel コンパイラ	-openmp	-parallel	-openmp -parallel
PGI コンパイラ	-mp	-Mconcur	-mp -Mconcur
Sun Studio	-xopenmp=parallel	-xautopar	-xopenmp=parallel -xautopar
Visual Studio	/openmp	なし	なし
IBM XLC	-qsmp=omp	なし(*1)	-qsmp
PathScale	-mp	-apo	-mp -apo

(*1) -qsmp=noomp だと OpenMP と自動並列化の両方を有効にします。OpenMP だけを無効にするオプションはありません。

詳細については、各コンパイラのマニュアルをご覧ください。

3.2 実行方法

プログラムを実行する前に、環境変数 OMP_NUM_THREADS に並列スレッド数を設定します。並列スレッド数を 16 に指定する場合は下記のようにします。あとは普通に実行するだけです。

シェル	環境変数の設定方法
sh	OMP_NUM_THREADS=16 export OMP_NUM_THREADS
csh と tcsh	setenv OMP_NUM_THREADS 16
bash と Linux や FreeBSD 上の sh	export OMP_NUM_THREADS=16
Windows	set OMP_NUM_THREADS=16

4　OpenMP の指示文

4.1　#pragma omp parallel

「#pragma omp parallel」の次の 1 文またはブロック（｛から｝の部分）が並列に実行されます。並列に実行される区間を並列リージョンと呼びます。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello World!¥n");
  }
}
```

上記のプログラムの場合、「Hello World!」が実行スレッドの数だけ表示されます。OpenMP ならこれだけのコード量で並列化が実現できます。

4.2　#pragma omp for

for 文を並列化します。並列リージョン内で指定する必要があります。

int 型の配列 a[100]の全要素を 0 に初期化するプログラムを並列化するには下記のようにします。

```
int main(void)
{
  int i, a[100];
  #pragma omp parallel
  {
    #pragma omp for
    for(i=0;i<100;i++){
      a[i]=0;
    }
  }
}
```

```
int main(void)
{
  int i, a[100];
  #pragma omp parallel for
  for(i=0;i<100;i++){
    a[i]=0;
  }
}
```

上記の 2 つのプログラムは基本的に同じ意味です（コンパイラによっては異なるバイナリコードを吐き出すものもあります）。並列リージョンの中に「#pragma omp for」が 1 つしかない場合には、右のプログラムのように結合指示文「#pragma omp parallel for」を使って書くと行数が短くなります。

上記のプログラムの場合、例えば実行スレッド数を 4 にして実行すると、4 つのスレッドが下記のように for ループの処理を分担して実行します。

スレッド 0 番：　for(i= 0;i< 25;i++) a[i]=0;
スレッド 1 番：　for(i=25;i< 50;i++) a[i]=0;
スレッド 2 番：　for(i=50;i< 75;i++) a[i]=0;
スレッド 3 番：　for(i=75;i<100;i++) a[i]=0;

実行スレッド数は動的に決定されるため、このような分担をプログラム実行時に行うような実行コードがコンパイラによって生成されています。

4.3 #pragma omp sections

「#pragma omp sections」指示文ではブロック内を並列に処理します。必ず宣言子「#pragma omp section」とともに使われます。その宣言子「#pragma omp section」の次の 1 文またはブロックを 1 つのスレッドに割り当てて並列に実行します。なお、「#pragma omp section」の後の 1 文またはブロックをセクションと呼びます。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      printf("Hello 1¥n");
      #pragma omp section
      printf("Hello 2¥n");
      #pragma omp section
      {
        printf("Hello 3¥n");
        printf("Hello 3¥n");
      }
    }
  }
}
```

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel sections
  {
    #pragma omp section
    printf("Hello 1¥n");
    #pragma omp section
    printf("Hello 2¥n");
    #pragma omp section
    {
      printf("Hello 3¥n");
      printf("Hello 3¥n");
    }
  }
}
```

あるスレッドが「Hello 1」と表示、別のスレッドが「Hello 2」と表示、さらに別のスレッドが「Hello 3」を 2 回表示します。実行スレッド数が「#pragma omp section」宣言子で指定したセクションの数より少ない場合には、早く処理の終わったスレッドがまだ実行が開始されていないセクションを実行していきます。逆に実行スレッド数が「#pragma omp section」宣言子で指定したセクションの数より多い場合には、仕事を一切行わないスレッドが出てくることになります。

最初のセクションの始まりの「#pragma omp section」宣言子は記述を省略することができます。

「#pragma omp section」宣言子には指示節を付けることはできません。

「#pragma omp sections」指示文で指定したブロックの出口では暗黙のバリア（全てのスレッドがその場所に到達するまで待機する）があります。そのため、ブロック以降の処理を開始する段階で、全てのセクションの実行が終えていることが保証されています。ただし、これは「#pragma omp sections」指示文に nowait 指示節が指定されていない場合に限ります。

並列リージョンの中に指示文が「#pragma omp sections」指示文の 1 つしかない場合には、右のプログラムのように結合指示文「#pragma omp parallel sections」を使って書くと行数が短くなります。

セクション部分では、当然ながら関数呼び出しを行うことも可能です。そうすると、各スレッドで全く独立した処理を実行するようなことも可能となります。

4.4 #pragma omp single

1 スレッドだけが実行するブロックであることを指定します。どのスレッドが実行するかは決まっていません。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello 1¥n");
    #pragma omp single
    {
       printf("Hello 2¥n");
    }
    printf("Hello 3¥n");
  }
}
```

「Hello 1」と「Hello 3」は実行スレッドの数だけ表示されますが、「Hello 2」は 1 回だけ表示されます。

「#pragma omp single」指示文の出口では暗黙のバリアがあります。そのため、「Hello 1」と「Hello 2」の表示が全て行われたあとに、「Hello 3」が表示されます。逆に入口では暗黙のバリアがないため、「Hello 1」が全て表示される前に「Hello 2」が表示されることがあります。

4.5 #pragma omp master

マスタースレッド(0 番スレッド)だけが実行することを指定します。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello 1¥n");
    #pragma omp master
    {
       printf("I am a master¥n");
    }
    printf("Hello 2¥n");
  }
}
```

「Hello 1」と「Hello 2」は実行スレッドの数だけ表示されます。「I am a master」はマスタースレッドだけが表示します。一般には、マスタースレッドに限定するよりは「#pragma omp single」指示文を使ってどのスレッドが実行してもいいようにした方が効率は良くなります。

「#pragma omp master」指示文では、指定されたブロックの入口と出口で暗黙のバリアは存在しません。「I am a master」が表示される前に「Hello 2」が表示されることがあります。その点は「#pragma omp single」指示文とは異なっているので注意が必要です。

4.6 #pragma omp critical

直後の 1 文またはブロックの実行を一度に 1 つのスレッドに制限します。このような領域をクリティカル領域といいます。全てのスレッドが実行を行うという点に注意してください。

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
  #pragma omp parallel
  {
    #pragma omp critical (name)
    {
      sleep(1);
      printf("sleep end¥n");
    }
  }
}
```

上記のプログラムでは、各スレッドが sleep(1)で 1 秒待機した後、「sleep end」と表示します。一度に 1 つのスレッドに制限しているので、結果的に、1 秒おきに「sleep end」が表示されます（実行スレッドの数だけ)。

オプションの name は、クリティカル領域を識別するのに使用します。スレッドは、同じ名前の クリティカル領域を他のどのスレッドも実行していない状態になるまで、クリティカル領域の入り口で待機します。名前のないクリティカル領域は全て同一のクリティカル領域として扱われます。

4.7 #pragma omp atomic

直後の 1 文をアトミック命令（複数のスレッドが衝突せずに安全に共有変数の値を更新する）として実行することを指定します。ブロックを指定することはできません。

```
#include <stdio.h>
int main(void)
{
  int i=0;
  #pragma omp parallel
  {
    #pragma omp atomic
    i++;
    printf("i=%d¥n",i);
  }
}
```

「#pragma omp atomic」宣言文の後に続く 1 文は下記のものだけに限定されています。x はスカラ変数、値には x を参照しない一般の式を記述できます。

```
x++     ++x     x--     --x     x+=値   x-=値
x*=値   x/=値   x&=値   x^=値   x|=値   x<<=値   x>>=値
```

「#pragma omp atomic」指示文はいつでも「#pragma omp critical」指示文に置き換えることが可能です。いかし、「#pragma omp atomic」指示文を用いると一般にハードウェア命令による値の更新を行うため「#pragma omp critical」指示文を使うよりも効率が良くなります。

4.8 #pragma omp barrier

同時に実行されている全てのスレッドの同期を取ります。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello 1¥n");
    #pragma omp barrier
    printf("Hello 2¥n");
  }
}
```

上記の場合、全てのスレッドが「Hello 1」を表示し終わってから、「Hello 2」が表示されます。

共有変数の値の参照・更新、プログラム内部での時間計測、出力を順番通りに行いたい場合などに利用します。また、プログラムのデバッグの際にもよく使われます。ただし 「#pragma omp barrier」指示文をたくさん入れてしまうと速度低下の一因になりますので、利用は最小限に留めましょう。

4.9 #pragma omp ordered

直後の 1 文またはブロックを for ループが逐次実行された場合の順番で実行します。これは、「#pragma omp for」指示文または「#pragma omp parallel for」指示文のブロック内で指定する必要があります。また、それらの指示文には ordered 指示節を付ける必要があります。

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
  int i, a[100];
  #pragma omp parallel for ordered
  for(i=0;i<100;i++){
    a[i]=0;
    #pragma omp ordered
    printf(" i=%d thread_num=%d¥n",i,omp_get_thread_num());
  }
}
```

上のプログラムを実行スレッド数 4 で実行すると下記のように表示されます。

i=0 thread_num=0
i=1 thread_num=0
・・・
i=24 thread_num=0
i=25 thread_num=1
・・・
i=49 thread_num=1
i=50 thread_num=2
・・・
i=74 thread_num=2
i=75 thread_num=3
・・・
i=99 thread_num=3

この場合、i=0～24、i=25～49、i=50～74、i=75～99 の各ループは並列に実行されていないため、全くマルチプロセッサを有効に利用できておらず速度向上にはなりません。通常は、5.8 節で説明する schedule 指示節を指定して、for ループをサイクリックに並列処理するなどの指定が必要となります。

4.10 #pragma omp flush

実行中のスレッド間で共有変数や配列要素などの値の一貫性は通常保証されていません。これは、最適化などによりレジスタ内で値が保持されている間はメモリに書き込まれないことがあるからです。「#pragma omp flush」指示文を使うことで、明示的にレジスタに保持されている値をメモリに書き出すことができます。

ただし、下記の場所では、自動的に flush が行われ、共有変数や配列要素などのメモリの一貫性が保たれます。

#pragma omp parallel の入口と出口
#pragma omp for と #pragma omp parallel for の出口
#pragma omp sections と #pragma omp parallel sections の出口
#pragma omp single の出口
#pragma omp critical の入口と出口
#pragma omp barrier
#pragma omp ordered の入口と出口

ただし、nowait 指示節を入れると flush は行われません。

逆に、下記の場所では、flush が行われないので注意が必要です。

#pragma omp for と#pragma omp parallel for の入口
#pragma omp sections と #pragma omp parallel sections の入口
#pragma omp single の入口
#pragma omp master の入口と出口

4.11 #pragma omp threadprivate(list)

スレッドごとにプライベートで、スレッド内ではグローバルにアクセスできる変数（threadprivate 変数）を宣言します。通常はプログラムのグローバル領域（関数の外）で宣言します。複数の変数を指定する場合には、カンマ（,）で区切ります。

スレッドは他のスレッドの threadprivate 変数を参照することはできません。プログラムの逐次実行領域では、マスタースレッドの持つ値を参考することになります。

threadprivate 変数の初期値は、宣言した変数の初期値と等しくなります。

「#pragma omp parallel」指示文に copyin 指示節を指定することでマスタースレッドの値が全てのスレッドの値としてコピーされます。

threadprivate 変数は、copyin 指示節、 schedule 指示節、if 指示節に指定することができますが、private 指示節、firstprivate 指示節、 lastprivate 指示節、shared 指示節、reduction 指示節で指定することはできません。また、threadprivate 変数については default 指示節は適用されません。

「#pragma omp threadprivate」指示文の使い方の例を示します。

```
 1: #include <stdio.h>
 2: #include <omp.h>
 3:
 4: int i=100;
 5: #pragma omp threadprivate(i)
 6:
 7: int main()
 8: {
 9:   i=200;
10:   #pragma omp parallel
11:   printf("thread_num=%d i=%d¥n",omp_get_thread_num(),i);
12:
13:   i=1000;
14:   #pragma omp parallel copyin(i)
15:   {
16:     i+=omp_get_thread_num();
17:     printf("thread_num=%d i=%d¥n",omp_get_thread_num(),i);
18:   }
19: }
```

5 行目で int 変数である i を threadprivate 変数にしています。4 行目に i の初期値として 100 を代入しているため各スレッドの参照する i の値も全て 100 が初期値として代入されます。9 行目に i=200 としておりこれはマスタースレッドの保持する i のみが 200 になります。マスタースレッド以外のスレッドの保持している値は 100 のままです。したがって、11 行目の printf()で表示される結果は次のようになります。

```
thread_num=0 i=200
thread_num=1 i=100
thread_num=2 i=100
thread_num=3 i=100
```

次に、13 行目で i=1000 とし、14 行目では copyin(i)を指定しているためマスタースレッドの保持する i の値である 1000 が全てのスレッドの threadprivate 変数の i にコピーされます。16 行目で各スレッドの番号を threadprivate 変数の i に加えています。その結果、17 行目の printf()で出力される結果は次のようになります。

```
thread_num=0 i=1000
thread_num=1 i=1001
thread_num=2 i=1002
thread_num=3 i=1003
```

5　OpenMP の指示節

5.1　private(list)

list に指定された変数が各スレッドでプライベートであることを宣言します。複数の変数を指定する場合には、カンマ（,）で区切ります。「#pragma omp for」指示文の対象となる for ループのインデックス変数は自動的にプライベートになります。そのためこのインデックス変数については記述を省略できます。default 指示節を指定しない場合、変数のデフォルトの属性は共有変数（shared）になっていますので、プライベート変数がある場合にはこの private 指示節を使って宣言する必要があります。

5.2　firstprivate(list)

list に指定された変数は private 指示節と同様、プライベートであることを宣言します。private 指示節との違いは、変数の初期値として並列リージョン開始時の変数の値が各スレッドのプライベート変数にコピーされるという点です。並列リージョン内でプライベート変数の初期値を設定する場合に firstprivate 指示節を用いると効率が悪くなります。

5.3　lastprivate(list)

list に指定された変数は private 指示節と同様、プライベートであることを宣言します。private 指示節との違いは次の点です。「#pragma omp for」指示文で指定された場合には、for ループの最後の繰り返しを実行したスレッドの持つプライベート変数の値が元の変数の値に代入されます。「#pragma omp sections」指示文で指定された場合には、最後のセクションを実行したスレッドの持つプライベート変数の値が元の変数の値に代入されます。

5.4　shared(list)

list に指定された変数が各スレッドで共有変数であることを宣言します。default 指示節を指定しない場合、デフォルトの属性が共有変数（shared）ですので、shared 指示節を使う必要はありません。

5.5　default(shared|none)

並列リージョン内の全ての変数に対してデフォルトの属性を与えます。

default(shared)を指定した場合、デフォルトを共有変数とします。default(none)を指定すると、デフォルトの属性を与えません。この場合、変数は private、shared、firstprivate、lastprivate、reduction のどれかの指示節で指定されていなくてはなりません。なお、default 指示節を指定しなかった場合には、default(shared)が設定されているとみなします。

5.6 reduction(operator:list)

list で指定した変数に対して演算子 operator のリダクション演算を行うことを宣言します。リダクション演算とは総和を求めるような計算のことです。operator には、次のいずれかを指定します。

+	*	-	&	^	|	&&	||

list には複数の変数を指定することできますが、この場合、カンマ（,）で区切ります。

「#pragma omp atomic」指示文や「#pragma omp critical」指示文でリダクション演算を記述することもできますが、この reduction 指示節を使える場合には使った方が高速になります。

double 型配列 a[]の n 個の要素の和を求める関数であれば、次のようになります。

```
double sum(double a[], int n)
{
   int i;
   double sum=0.0;
   #pragma omp parallel for reduction(+:sum)
   for(i=0; i<n; i++) {
      sum += a[i];
   }
   return sum;
}
```

一見、「reduction(+:sum)」の部分がなくても正しく動作するように見えるかもしれませんが、複数スレッドで実行すると共有変数 sum には正しい結果が入る保証はありません。共有変数 sum が各スレッド内ではレジスタで処理されたり更新の際にメモリ競合が起こったりするためです。reduction 指示節を用いて、共有変数 sum を宣言しておくと、正しい結果が格納されるようになります。

reduction 指示節は次のように「#pragma omp parallel sections」指示文においても指定できます。

```
double sum(double a[], int n)
{
   int i;
   double sum=0.0;
   #pragma omp parallel sections private(i) reduction(+:sum)
   {
      #pragma omp section
      for(i=0; i<n/2; i++) sum += a[i];
      #pragma omp section
      for(i=n/2;i<n;i++) sum += a[i];
   }
   return sum;
}
```

5.7 ordered

for ループ中に「#pragma omp ordered」指示文が含まれていることを宣言します。

5.8 schedule

for ループにおいてループ反復をどのようにスレッドに割り当てるかを宣言します。schedule(type)または schedule(type,chunk)の形で使用します。type には static、dynamic、guided、runtime のうちのいずれかが入ります。ここで、チャンクとは 1 つのスレッドが処理を行う最小単位であるループの反復回数のことをいいます。

static では静的すなわち反復開始前に各スレッドに対してチャンクが割り当てられます。chunk を指定しなかった場合には、チャンクは総反復回数を実行スレッド数で割った値となります。schedule 指示節を指定しなかった場合のデフォルトは static でチャンク指定なしと同じです。

dynamic では、チャンクは遊んでいるスレッドに対して動的に割り当てられます。すなわち、処理を終了した順に次のチャンク分の反復の処理がスレッドに割り当てられます。chunk を省略すると chunk=1 とみなされます。

guided でも遊んでいるスレッドに対して動的に割り当てられますが、チャンクの値は自動的に決定

されます。この場合、chunk には、分割の際に最小となる反復回数を指定します。chunk を省略すると chunk=1 とみなされます。

runtime だけは他と異なり、環境変数 OMP_SCHEDULE の値を利用することを指定します。この場合、chunk は指定できません。

5.9 copyin

copyin 指示節は threadprivate 指示文で指定された変数に適用できます。copyin 節で指定された変数は、並列リージョンの開始時にマスタースレッド上の threadprivate 変数の値を、各スレッドの threadprivate 変数にコピーします。

5.10 copyprivate(list)

「#pragma omp single」指示文だけで利用できる指示節です。list に指定された変数を他のスレッドにコピーします。

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main()
{
  int i;
  #pragma omp parallel private(i)
  {
    #pragma omp single copyprivate(i)
    i=random();

    printf("thread_num=%d i=%d¥n",omp_get_thread_num(),i);
  }
}
```

1 スレッドだけが random()を実行します。そして、その結果を全てのスレッドのプライベート変数 i にコピーします。

5.11 if

「#pragma omp parallel」指示文を並列実行する場合の条件を記述します。条件が成立しない場合には逐次実行されます。次の場合であれば、変数 n の値が 100 以上の場合に並列実行を行います。

```
#pragma omp parallel for if(n>=100)
{
  ・・・
```

5.12 num_threads

実行スレッド数を指定します。環境変数 OMP_NUM_THREADS よりも優先されます。ただし、システムで上限が決められている場合にはそれを超える値を指定すると実行時にエラーになります。

```
#pragma omp parallel num_threads(4)
{
  ・・・
```

5.13 nowait

処理が終了したスレッドは他のスレッドの状況に関係なく、次の処理に移ってもよいことを宣言します。

5.14 指示文と指示節の可能な組み合わせ

指示文と指示節の可能な組み合わせをまとめると次の表のようになります。

	parallel 指示文	for 指示文	sections 指示文	single 指示文	parallel for 指示文	parallel sections 指示文
private	●	●	●	●	●	●
firstprivate	●	●	●	●	●	●
lastprivate		●	●		●	●
shared	●				●	●
default	●				●	●
reduction	●	●	●		●	●
ordered		●			●	
schedule		●			●	
copyin	●				●	●
copyprivate				●		
if	●				●	●
num_threads	●				●	●
nowait		●	●	●		

6 OpenMP の実行時ライブラリ関数

6.1 ヘッダファイルについて

OpenMP の実行時ライブラリ関数を利用するには、ヘッダファイル omp.h をインクルードします。

```
#include <omp.h>
```

このままだと OpenMP 非対応コンパイラや OpenMP を無効にしてコンパイルしようとするとインクルードファイルがないというエラーになってしまいます。それを避けるには_OPENMP がマクロ定義されているかどうかを利用します。OpenMP の規格では、OpenMP を有効にしてコンパイルすると、OpenMP Version 1.0 の仕様に基づいている場合には「_OPENMP =199810」が OpenMP Version 2.0 の仕様に基づいている場合には「_OPENMP =200505」がマクロ定義されています。

同一のソースプログラムで OpenMP 対応と非対応の両方を記述するには、次のプログラムのように _OPENMP がマクロ定義されているかどうかを利用します。

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

6.2 実行環境ルーチン

実行環境の設定、現在の実行状態を参照するためのルーチンには下記のものがあります。

void **omp_set_num_threads(int);**

次の並列リージョン開始時に起動されるスレッド数を指定します。システムで上限が決められている場合にはそれを超える値を指定すると実行時にエラーになります。

int **omp_get_num_threads();**

現在、起動しているスレッドの数を返します。

int **omp_get_max_threads();**

スレッドの最大生成数を返します。

int **omp_get_thread_num();**

自分のスレッド番号を得ます。スレッド番号は 0 から p-1（スレッド数を p とする）の値になります。

int **omp_get_num_procs();**

プログラムで使用可能なプロセッサの数を返します。

int **omp_in_parallel();**

並列起動が行われている場合は 0 以外の値を返します。そうでない場合は 0 を返します。

void **omp_set_dynamic(int);**

スレッド数の動的調整機能を有効にする場合は 0 以外の値、無効にする場合は 0 を指定します。

int **omp_get_dynamic();**

スレッド数の動的調整機能が有効である場合は 0 以外の値、無効である場合には 0 を返します。

void **omp_set_nested(int);**

並列のネストを有効にする場合は 0 以外の値、無効にする場合は 0 を指定します。

int **omp_get_nested();**

並列のネストが有効である場合は 0 以外の値、無効である場合には 0 を返します。

6.3 ロックルーチン

OpenMP ではロック機構として単純ロックとネスト可能なロックの 2 種類が用意されています。これらを利用するために使うルーチンです。

void **omp_init_lock(&lock);**

ロック変数を初期化します。lock は「omp_lock_t lock」のように宣言された変数です。

void **omp_destory_lock(&lock);**

ロック変数を破棄します。

void **omp_set_lock(&lock);**

ロックの所有権を得るまで待機し、ロックの所有権を得ると処理が戻ります。ロックの所有権を得たスレッド自身が omp_unset_lock()で明示的にロックの所有権を解放するまで他のスレッドはロックの所有権を得ることはできません。

void **omp_unset_lock(&lock);**

ロックの所有権を解放します。これにより他のスレッドがロックの所有権を得ることができるよう

になります。

int **omp_test_lock(&lock);**

ロックの所有権を得ることを試みます。ロックの所有権を得ると 1 を返し、そうでない場合には 0 を返します。ロックの所有権が得られるまで待機しないという点が omp_set_lock()との違いです。

void **omp_init_nest_lock(&lock);**

ネスト可能なロック変数を初期化します。ネストカウンタ（所有権を持っているスレッドがロックを行った回数のこと）も 0 に設定されます。lock は「omp_nest_lock_t lock」のように宣言された変数です。ネスト可能なロックでは、同じスレッドによって複数回ロックすることができます。

void **omp_destory_nest_lock(&lock);**

ネスト可能なロック変数を破棄します。

void **omp_set_nest_lock(&lock);**

すでに同じスレッドによって所有権を獲得している場合にはネストカウンタを 1 増やして処理が戻ります。そうでない場合には、ネスト可能なロックの所有権を得るまで待機し、ロックの所有権を得ると処理が戻ります。

void **omp_unset_nest_lock(&lock);**

ネスト可能なロックのネストカウンタを 1 だけ減らします。その結果ネストカウンタが 0 になった場合にはロックの所有権を解放します。これにより他のスレッドがロックの所有権を得ることができるようになります。

int **omp_test_nest_lock(&lock);**

すでに同じスレッドによって所有権を獲得している場合にはネストカウンタを 1 増やしてその値を返します。そうでない場合には、ネスト可能なロックの所有権を得ることを試みます。ロックの所有権を得ると 1 を返し、そうでない場合には 0 を返します。ロックの所有権が得られるまで待機しないという点が omp_set_nest_lock()との違いです。

6.4 時間計測ルーチン

double omp_get_wtime();

1970 年 1 月 1 日午前 0 時からの経過秒数、あるいは、システムを起動してからの経過秒数を倍精度実数で返します。経過時間の測定では、計測開始時点と終了時点の 2 個所でこの関数を呼び出し、その差を経過時間とします。

double omp_get_wtick();

omp_get_wtime()の返す値の刻み幅を倍精度実数で返します。

7　OpenMP の環境変数

環境変数は全て大文字ですが、環境変数に設定する値については大文字と小文字の区別はありませんので、どちらで指定しても構いません。

7.1　OMP_NUM_THREADS

使用するスレッド数を指定します。

通常は、物理プロセッサの数を超える値を指定することができますが、システムによっては上限が決められていることもあります。また omp_set_num_threads() ライブラリルーチンを呼び出すか、「#pragma omp parallel」指示文で num_threads 指示節を使って明示的にスレッド数を指定している場合には、この変更後の値が優先されます。

7.2　OMP_SCHEDULE

「#pragma omp for」指示文と「#pragma omp parallel for」指示文において、schedule(runtime) 指示節を指定した場合のスケジューリング方法を指定します。type または type,chunk という値を指定します。type は STATIC、DYNAMIC、GUIDED のいずれかであり、chunk はチャンクの大きさを指定します。

例：setenv OMP_SCHEDULE STATIC,5
　　setenv OMP_SCHEDULE DYNAMIC,5
　　setenv OMP_SCHEDULE GUIDED

7.3　OMP_DYNAMIC

スレッド数の動的調整機能（システムの負荷によって実行スレッド数を変更する機能）を有効にする場合は TRUE、無効にする場合は FALSE を指定します。デフォルト値は実装依存とされているため、システムによって異なります。決まった数のスレッドを必要とする場合には、TRUE に設定する必要があります。PGI などの一部のコンパイラでは利用できません。

7.4　OMP_NESTED

並列のネストを有効にする場合は TRUE、無効にする場合は FALSE を指定します。デフォルト値は FALSE になっています。PGI などの一部のコンパイラでは利用できません。

7.5　OMP_WAIT_POLICY　（OpenMP 3.0 の機能）

スレッドの待機中の挙動を指定します。スピンループして待機する場合には ACTIVE（デフォルト）、スリープして待機する場合には PASSIVE を指定します。

7.6　OMP_STACK_SIZE（OpenMP 3.0 の機能）

各スレッドが生成されるときのスタックサイズを指定します。

例：setenv OMP_STACK_SIZE 2K
　　setenv OMP_STACK_SIZE 2M
　　setenv OMP_STACK_SIZE 2G
　　setenv OMP_STACK_SIZE 256B

K はキロバイト、M はメガバイト、G はギガバイト、B はバイトを示します。これらを省略した場合には、K が指定されたものとみなします。

8　サンプルプログラム

8.1 マルチ ping プログラム

192.168.0.1～192.168.0.254 のように連続した 254 個の IP アドレスに対して一斉に ping コマンドを実行するプログラムです。ping コマンドで -t オプションでタイムアウトの時間を指定できますが、タイムアウトを 1 秒に設定したとしても最大で 254 秒かかることになります。OpenMP で並列化して、実行スレッド数を 254 に設定すると 2 秒ほどで終わります。

使い方は、192.168.0.1～192.168.0.254 を調査したい場合には、次のように実行します。

```
% ./multiping 192.168.0
```

ping の応答があった場合には、「192.168.0.1 is up. time=0.980 ms」のように表示されます。

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: int main(int argc,char *argv[])
 5: {
 6:   int i;
 7:   if( argc!=2 ){
 8:     printf("usage : multiping 192.168.0¥n");
 9:     return 1;
10:   }
11:
12:   #pragma omp parallel for schedule(dynamic) ordered
13:   for(i=1;i<255;i++){
14:     char command[64],output[1024];
15:     FILE *in_pipe;
16:     int outputsize;
17:     sprintf(command,"ping -t 1 %s.%d",argv[1],i);
18:     in_pipe=popen(command,"r");
19:     outputsize=fread(output,1,1024,in_pipe);
20:     output[outputsize]='¥0';
21:     pclose(in_pipe);
22:
23:     #pragma omp ordered   // IP アドレス順に表示するために挿入
24:     {
25:       int k=0;
26:       char *tmp;
27:       if((tmp=strstr(output,"time="))!=NULL){
28:         while(tmp[k]!='¥n') k++;
29:         tmp[k]='¥0';
30:         printf("%s.%d is up. %s¥n",argv[1],i,tmp);
31:       }else{
32:         printf("%s.%d is down.¥n",argv[1],i);
33:       }
34:     }
35:   }
36:   return 0;
37: }
```

上記のプログラムでは、プロセッサの個数が 1 個でも実行スレッド数を増やすことで実行時間が短くなります。このようにプロセッサの個数が 1 個であっても高速化が実現できることもあります。

8.2 ファイルコピープログラム

2つのスレッドを用いてファイルコピーを行うプログラムです。1つのスレッドはファイルの読み込みを担当し、もう1つのスレッドは書き込みを担当します。

使い方は、次のようになります。

```
% ./filecopy 元のファイル名 コピー先のファイル名
```

同じディスク上でファイルをコピーする場合には、かえって遅くなることがあります。

```
 1: #include <stdio.h>
 2: #include <fcntl.h>
 3: #include <sys/stat.h>
 4: #include <sys/types.h>
 5: #include <sys/uio.h>
 6: #include <unistd.h>
 7: #define BUF_SIZE 4096*1024
 8: char buf1[BUF_SIZE];
 9: char buf2[BUF_SIZE];
10:
11: int main(int argc,char *argv[])
12: {
13:   int file_src,file_dst;
14:   int size1,size2;
15:   if( argc!=3 ){
16:     printf("usage : filecopy source-file dest-file¥n");
17:     return 1;
18:   }
19:   file_src=open(argv[1],O_RDONLY);
20:   file_dst=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,S_IREAD|S_IWRITE);
21:   if( file_src==-1 || file_dst==-1 ){
22:     printf("file read/write error¥n");
23:     return 1;
24:   }
25:   size1=read(file_src,buf1,BUF_SIZE);
26:
27:   #pragma omp parallel sections num_threads(2)
28:   {
29:     #pragma omp section
30:     while(1){
31:       size2=read(file_src,buf2,BUF_SIZE);
32:       #pragma omp barrier
33:       if( size2<=0 ) break;
34:       size1=read(file_src,buf1,BUF_SIZE);
35:       #pragma omp barrier
36:       if( size1<=0 ) break;
37:     }
38:     #pragma omp section
39:     while(1){
40:       write(file_dst,buf1,size1);
41:       #pragma omp barrier
42:       if( size2<=0 ) break;
43:       write(file_dst,buf2,size2);
44:       #pragma omp barrier
45:       if( size1<=0 ) break;
46:     }
47:   }
48:   close(file_src);
49:   close(file_dst);
50:   return 0;
51: }
```