講義4 並列プログラミングの基本(OpenMP)

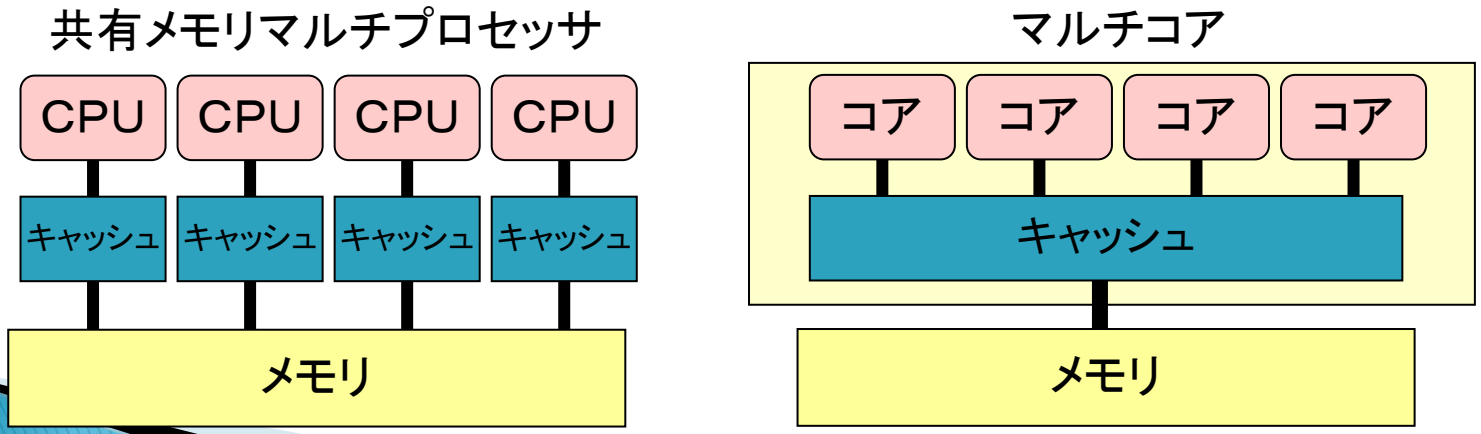
AICS 村井 均

もくじ

- 背景 — OpenMPとは
- OpenMPの基本
- OpenMPプログラミングにおける注意点
- やや高度な話題

背景

- 共有メモリマルチプロセッサシステムの普及
- 共有メモリマルチプロセッサシステムのための並列化指示文を共通化する必要性
  - 各社で仕様が異なり、移植性がない。
- そして、いまやマルチコア・プロセッサが主流となり、そのような並列化指示文の重要性はさらに増している。



従来手法: POSIXスレッド

- 「スレッド」のPOSIX標準（pthreadsライブラリ）
- 「スレッド」とは？
  - 一連のプログラムの実行を抽象化したもの。「仮想的なプロセッサ」と見なすこともできる。
  - 複数のスレッド間で資源、特に「メモリ空間」を共有する点が、「プロセス」とは異なる。
  - 通常、一つの共有メモリマルチプロセッサまたはコアに割り当てられる。
- 複数のスレッドによる並列処理を明示的に記述する。

※ 他に、コンパイラによる「自動並列化」を利用できる場合もある。

OpenMPとは（1）

- 共有メモリマルチプロセッサにおける並列プログラミングのためのモデル
  - ベース言語(Fortran/C/C++)をdirective（指示文）で並列プログラミングできるように拡張
- 米国コンパイラ関係のISVを中心に仕様を決定
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - 現在、OpenMP 3.0
- URL
  - http://www.openmp.org/
  - 日本語版の仕様書も公開されている。

OpenMPとは（2）

- 並列実行モデルへのAPI
  ⇔ 従来の指示文は並列化コンパイラのための「ヒント」
- 科学技術計算が主なターゲット（これまで）
  - 並列性が高い。
  - 95%の実行時間を占める(?)5%のコードを簡単に並列化する。
- 共有メモリマルチプロセッサシステムがターゲット
  - small-scale(～16プロセッサ）からmedium-scale (～64プロセッサ）

従来手法(POSIXスレッド)によるプログラミング

```
int main(void)
{
  for (t = 1; t < n_thd; t++)
    r = pthread_create(thd_main, t)
  thd_main(0);
  for (t =1; t < n_thd; t++)
    pthread_join();
}

int s;  /* global */
int n_thd; /* number of threads */

int thd_main(int id)
{
  int c, b, e, i, ss;
  c = 1000 / n_thd;
  b = c * id;
  e = s + c;
  ss = 0;
  for(i = b; i < e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

問題:a[0]～a[999]の総和を求める。

以下の全ての処理を明示しなければならない。

- スレッドの生成
- ループの担当部分の分割
- 足し合わせの同期

OpenMPによるプログラミング

逐次プログラムに、指示行を追加するだけ！

```
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 1000; i++) s+= a[i];
```

OpenMPの特徴

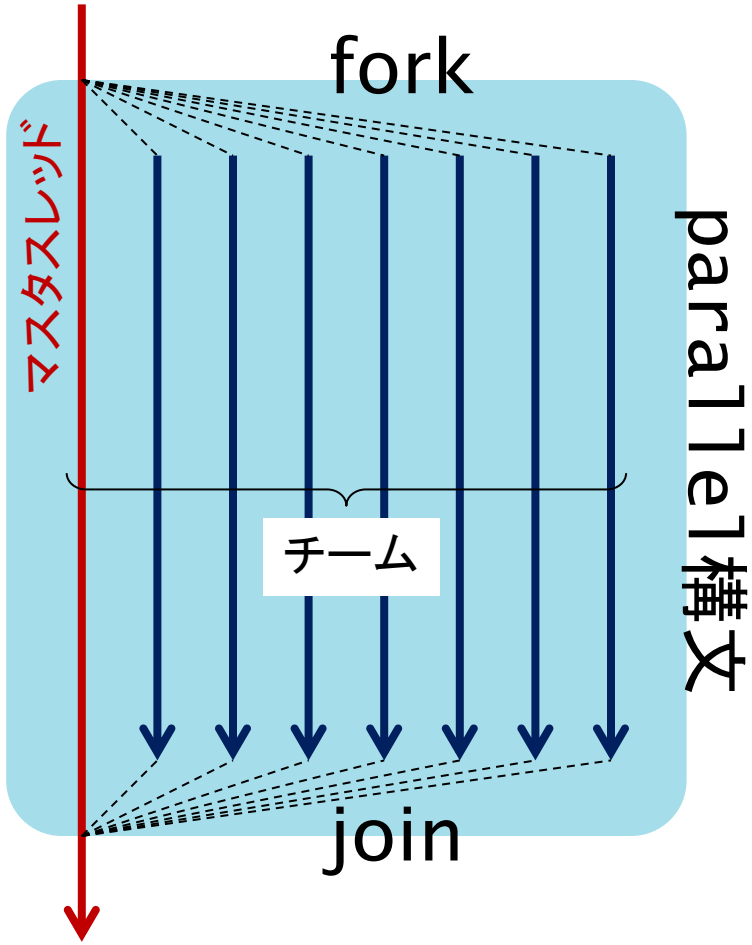
- 新しい言語ではない。
  - コンパイラ指示文、実行時ライブラリルーチン、環境変数によりベース言語を拡張。
  - ベース言語：Fortran, C/C++
- 自動並列化ではない。
  - 並列実行および同期をプログラマが明示する。
- 指示文を無視すれば、逐次プログラムとして実行可。
  - incrementalな並列化
  - プログラム開発、デバックの面から実用的
  - 逐次版と並列版を同じソースで管理ができる。

もくじ

- 背景 — OpenMPとは
- OpenMPの基本
- OpenMPプログラミングにおける注意点
- やや高度な話題

実行モデル(1)

- fork-joinモデル
  - ただ一つのスレッドが実行を開始する。
  - parallel構文の入口に遭遇したスレッド(マスタスレッド)は、n個のスレッドを生成し(fork)、チームを構成する。
  - parallel構文の出口で、マスタスレッド以外のスレッドは消滅する(join)。



実行モデル(2)

- ワークシェアリング
  - チームは、ワークシェアリング構文に遭遇すると、指定された仕事を「分担して」実行する → 「並列処理」
  - ループ(do/for), sections, single, workshare

※ ワークシェアリング構文に遭遇しない限り、チームの各スレッドは、仕事を「重複して」実行する。

例.

仕事1～100を、4スレッドで「ワークシェア」した結果、

- スレッド0は、仕事1～25、
- スレッド1は、仕事26～50、
- スレッド2は、仕事51～75、
- スレッド3は、仕事76～100、

をそれぞれ実行する。

メモリモデル

- 特に指定のない限り、全てのスレッドはメモリ空間を共有する。
  - どのスレッドも、どのデータをもアクセスできる。
  - 競合状態やコンシステンシを意識する必要がある（後述）。
- いくつかの指示文または指示節によって、あるデータのデータ共有属性を指定できる。

例.

```
int a, b;
#pragma omp threadprivate (b)
```

- 全てのスレッドは、共有変数aを読み書きできる。
- 各スレッドは、プライベート変数bを持つ。

OpenMP指示文の形式

- ベース言語Fortran
  - コメントの形式

```
!$omp  directive-name  [clause[[,] clause]...]
```

- ベース言語Fortran
  - プリプロセッサ指示の形式

```
#pragma omp  directive-name  [clause[[,] clause]...]
```

parallel構文

- 「parallelリージョン」＝「複数のスレッド（チーム）によって並列実行される部分」を指定する。
  - リージョン内の各文（関数呼び出しを含む）を、チーム内のスレッドが重複実行する。

Fortran:

```
!$OMP PARALLEL
      parallelリージョン
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel
{
      parallelリージョン
}
```

ループ構文

- ループの各繰り返しを、チーム内のスレッドが「分担して」実行することを指示する。

Fortran:

```
!$OMP DO [clause]...
DO i = 1, 100
   ...
END DO
```

C/C++:

```
#pragma omp for [clause]...
for (i = 0; i < 100; i++)
{
   ...
}
```

※ 対象のループは「標準形」でなければならない。

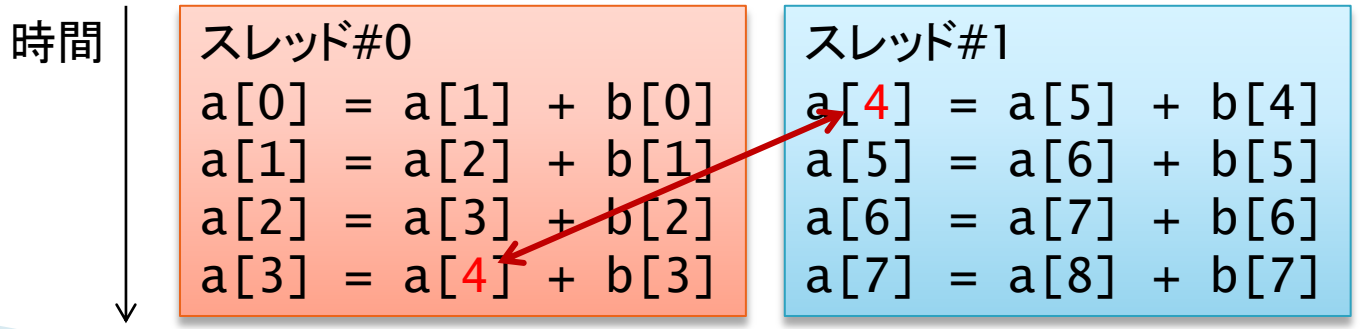
- *clause*で並列ループのスケジューリング、データ属性を指定できる。

ループ構文に関する注意

- 対象のループは、「並列化可能」でなければならない。
- 「並列化可能」＝「各繰り返しにおける、共有変数の定義引用に重なりがない」
- 並列化可能でなかった場合の結果は不定。

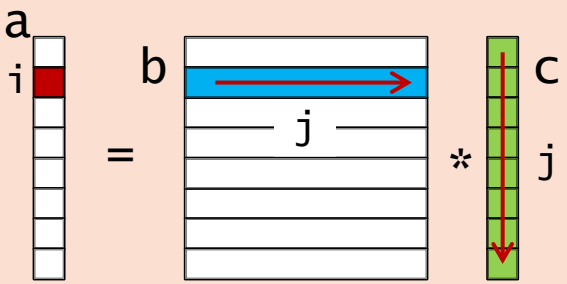
並列化できないループの例

```
for (i = 0; i < 8; i++)
    a[i] = a[i+1] + b[i];
```



ループ構文の例: 行列ベクトル積

```
#pragma omp parallel
#pragma omp for private(i,j,sum) shared(a,b,c)
for (i = 0; i < 8; i++)
{
  sum = 0.0;
  for (j = 0; j < 8; j++)
    sum += b[i][j] * c[j];
  a[i] = sum;
}
```



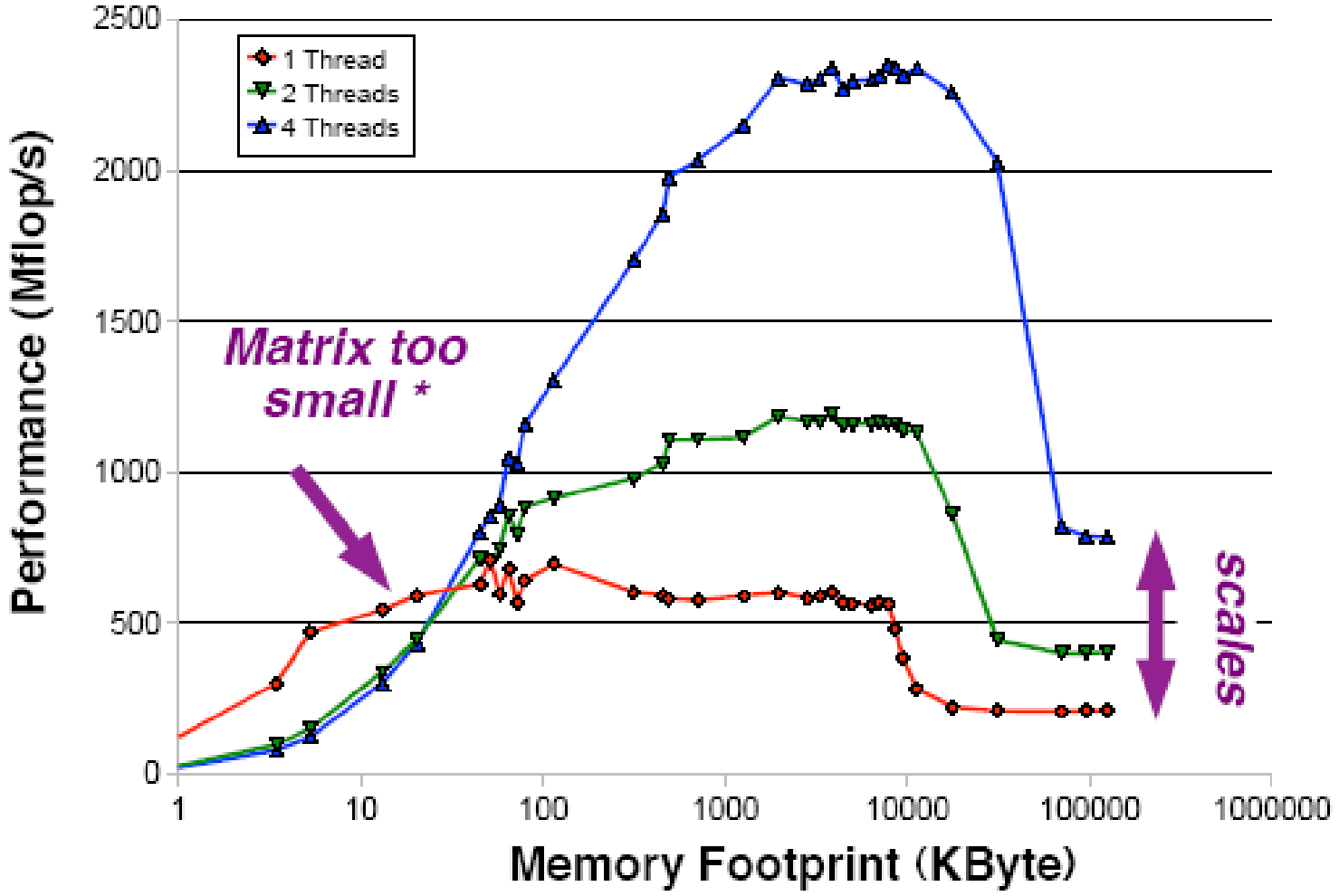
スレッド#0

```
for (i = 0,1,2,3)
{
  ...
}
```

スレッド#1

```
for (i = 4,5,6,7)
{
  ...
}
```

ループ構文による並列化の効果

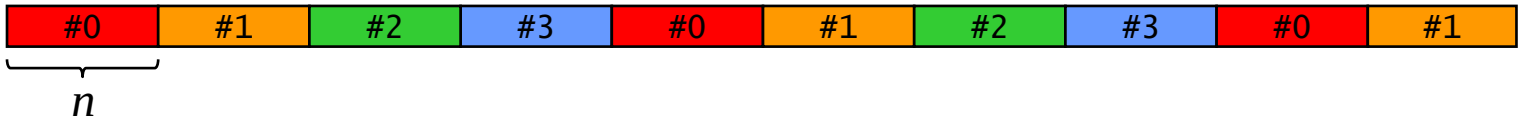


並列ループのスケジューリング

※ スレッド数4の場合

!$omp do schedule(static) ← デフォルト

!$omp do schedule(static,$n$)　コンパイル時に割り当てる（決定的）



!$omp do schedule(dynamic,$n$)　実行時に割り当てる（非決定的）

!$omp do schedule(guided,$n$)　実行時に割り当てる（だんだん短くなる）

データ共有属性

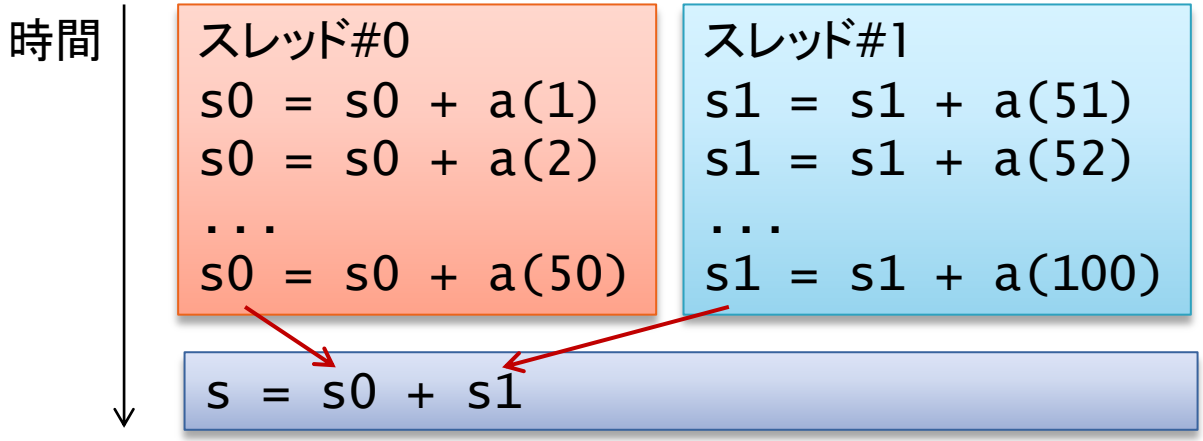
- shared(*var_list*) ← デフォルト
  - 指定された変数は共有変数である(スレッド間で共有される)。
- private(*var_list*)
  - 指定された変数はプライベート変数である(各スレッドに固有である)。
- firstprivate(*var_list*)
  - privateと同様だが、直前の値で初期化される。
- lastprivate(*var_list*)
  - privateと同様だが、ワークシェアリング構文の終了時に、逐次実行された場合の値を反映する。
- reduction(*op*:*var_list*)
  - privateと同様だが、構文の終了時に、*op*で指定された方法で各変数を「集計」した結果(e.g. 総和、最大値)を反映する。

reduction節の例

- 総和を求めるループ

```
!$omp do reduction(+:s)
do i = 1, 100
   s = s + a(i)
end do
```

- 各スレッドは「部分和」を求める。
- ループの終了後に、各「部分和」を足し合わせて「総和」を求める。



※ 総和の他に、論理積、論理和、最大値、最小値などを指定できる。

その他のワークシェアリング構文

- single
  - 直後のブロックを、1つのスレッドだけが実行する。
- sections
  - 複数のブロックを、チーム内の各スレッドが分担して実行する。
- workshare
  - Fortranの配列構文を、チーム内の各スレッドが分担して実行する。

並列ワークシェアリング構文

- parallel構文とワークシェアリング構文をまとめて指定するための「ショートカット」

```
!$OMP PARALLEL
!$OMP DO [clause]...
DO i = 1, 100
   ...
END DO
!$OMP END PARALLEL
```

||

```
!$OMP PARALLEL DO [clause]...
DO i = 1, 100
   ...
END DO
```

もくじ

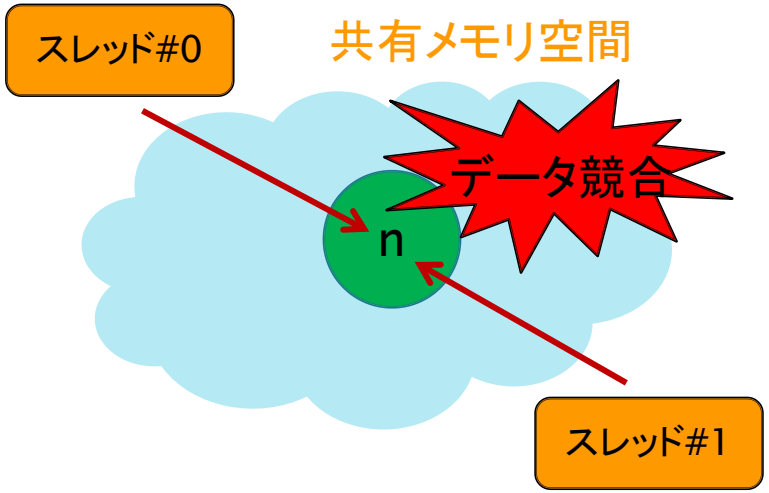
- 背景 — OpenMPとは
- OpenMPの基本
- OpenMPプログラミングにおける注意点
- やや高度な話題

「共有メモリ」に関する注意（1）

- OpenMPの共有メモリモデルでは、
  - 複数のスレッドが一つの共有変数を同時に書き換える＝「データ競合」
  - 複数のスレッドが一つの共有変数を同時に（順不同で）読み書きする。

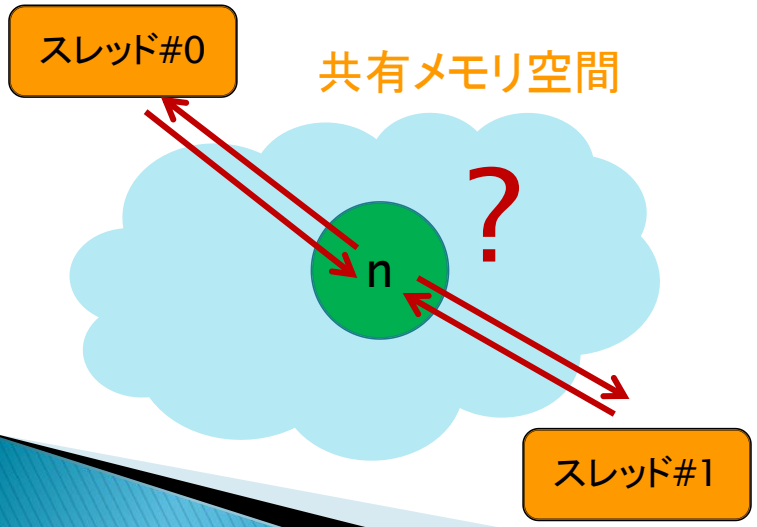
  という状況が起こり得る。
- その場合の結果は不定である → バグの温床

「共有メモリ」に関する注意(2)




```
#pragma omp parallel shared(n)
{
  n = omp_get_thread_num();
}
```

※ omp_num_thread_numは、自スレッドのIDを返す関数



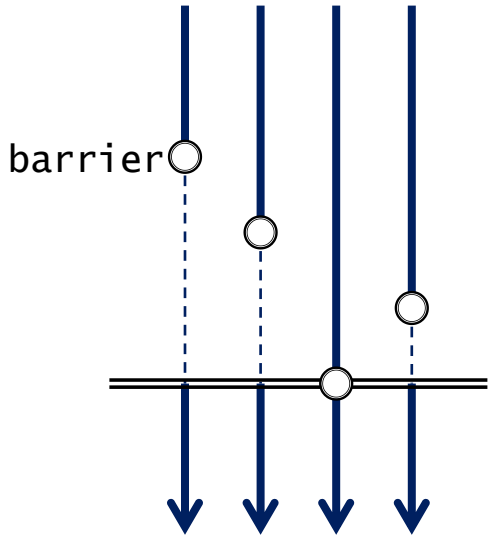

```
#pragma omp parallel shared(n)
{
  n = ...;
  ... = n ...
}
```

barrier構文

- バリア同期を行う。
  - チーム内の全スレッドがbarrier構文に達するまで待つ。
  - それまでのメモリ書き込みもflushする。
  - parallel構文とワークシェアリング構文の終わりでは、暗黙的にバリア同期が行われる。

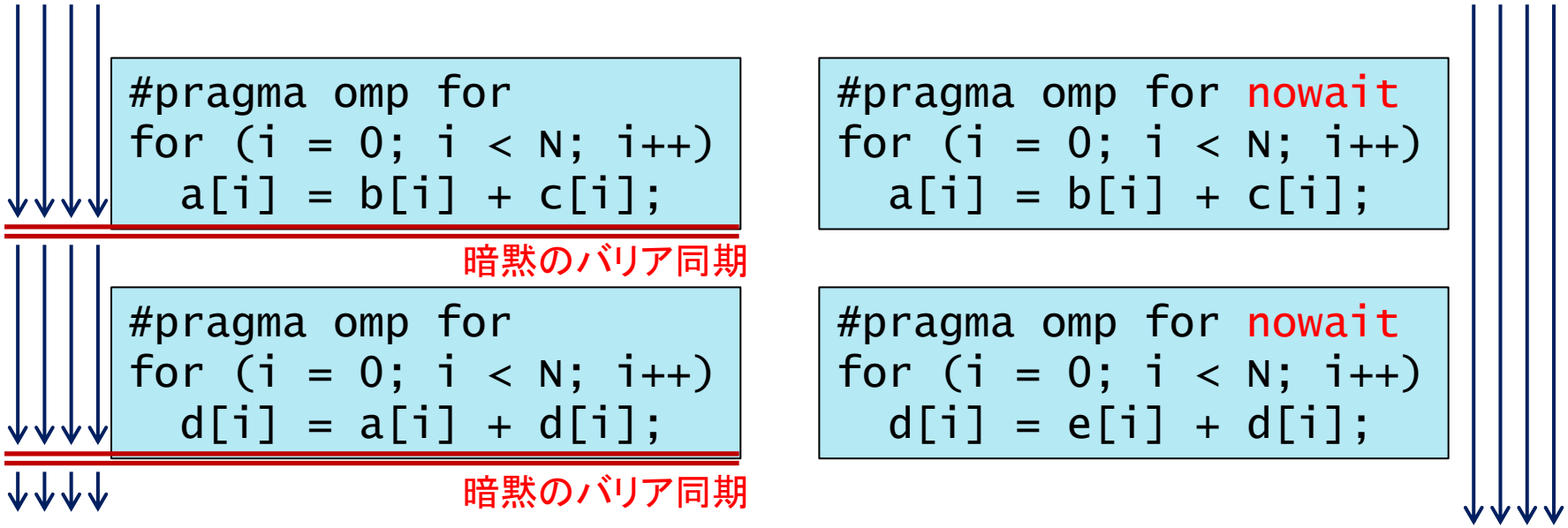
```
S1;
#pragma omp barrier
S2;
```

S2の実行時に、S1の処理が完了していることを保証する → S1とS2の定義引用が重なっていてもOK



nowait指示節

- parallel構文とワークシェアリング構文に付随する暗黙のバリア同期を除去することにより、性能向上につながる場合がある。



その他の構文/指示文

- master
  - 直後のブロックを、マスタ・スレッドだけが実行する。
- critical
  - クリティカルセクション（同時に実行できない部分）
- flush
  - メモリのフラッシュ
- threadprivate
  - スレッドプライベート変数を宣言する。

実行時ライブラリルーチン&環境変数

- 代表的な実行時ライブラリルーチン:

`int omp_get_num_threads(void)`	チーム内のスレッドの数を返す。
`int omp_get_thread_num(void)`	自スレッドのIDを返す。
`void omp_set_lock(omp_lock_t *lock)`	ロック変数*lock*が解放されるまで待つ。
`void omp_unset_lock(omp_lock_t *lock)`	ロック変数*lock*を解放する。

- 代表的な環境変数:

`OMP_NUM_THREADS`	parallelリージョンを実行するスレッドの数の既定値

もくじ

- 背景 — OpenMPとは
- OpenMPの基本
- OpenMPプログラミングにおける注意点
- やや高度な話題

タスク並列

- OpenMP 3.0で、「タスク並列処理」のための機能が導入された。
  - それまでのOpenMPは、基本的にループを並列処理するための仕様だった。
- 基本的な考え方:
  - あるスレッドがtask構文に遭遇すると、そのコードブロックが「タスク」として登録される。
  - 登録されたタスクは、チーム内のいずれかのスレッドによって、いずれかのタイミングで実行される。
  - taskwait構文またはbarrier構文は、登録された全てのタスクの完了を待つ。

task構文の例

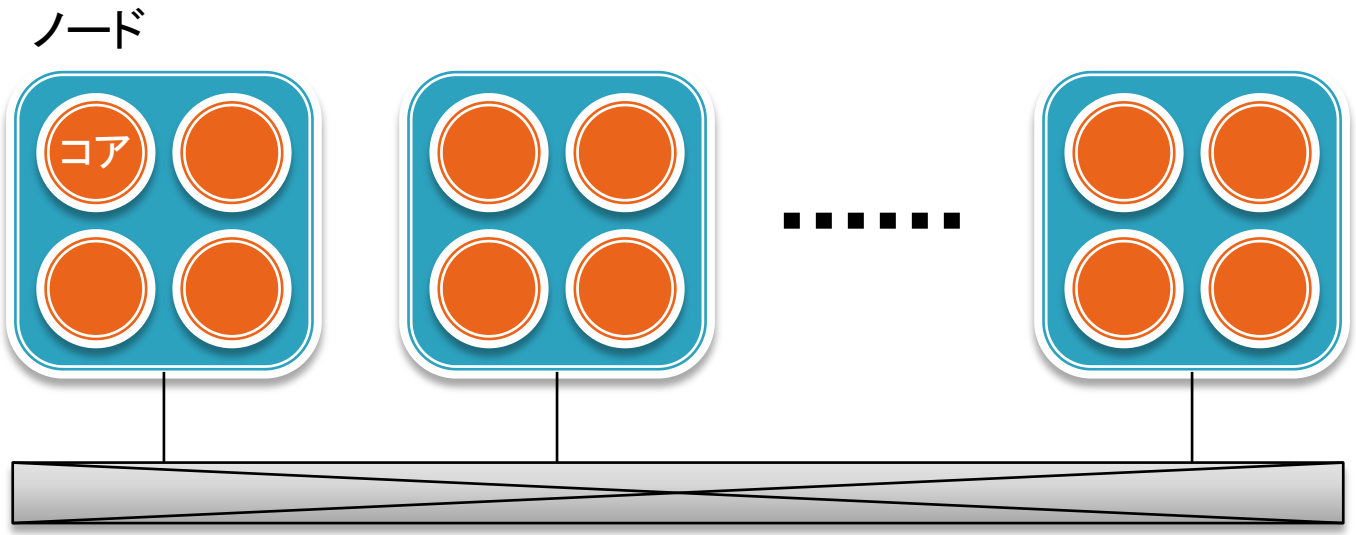
線形リストの処理

```
#pragma omp parallel
{
  #pragma omp single
  {
    node *p = head;
    while (p) {
      #pragma omp task
      process(p);
      p = p->next;
    }
  }
}
```

- while文の中で、ある一つのスレッドが、リストの各アイテムに対する処理の「タスク」を次々に生成。
- parallelリージョンの出口の暗黙のバリア同期において、全てのタスクの完了を待つ。

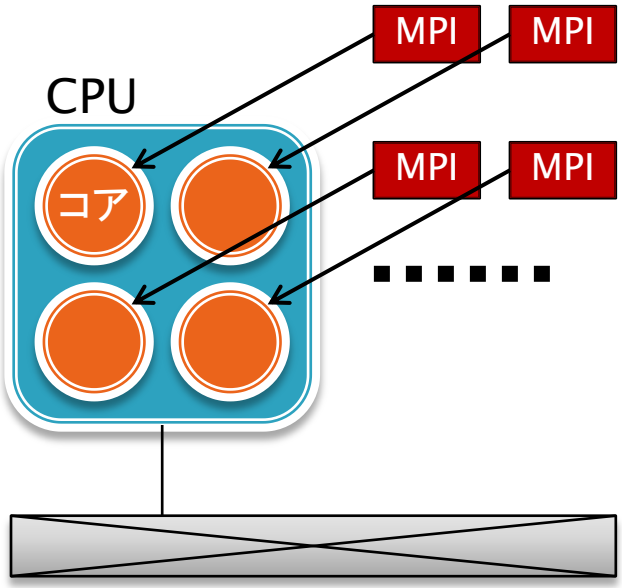
マルチコアクラスタにおける並列化（1）

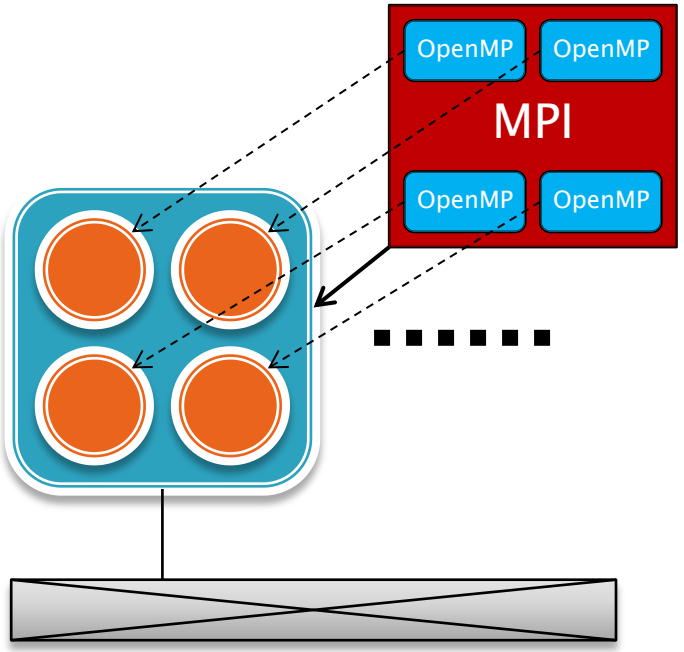
- 「マルチコアクラスタ」とは？
  - 各ノード（CPU）がマルチコアプロセッサであるクラスタ。現在のスーパーコンピュータの主流。
  - ノード間、コア間の2種類の（階層的な）並列性を持つ。



マルチコアクラスタにおける並列化（2）

- フラット並列化
  各コアにMPIプロセスを割り当てる。
- ハイブリッド並列化
  各CPUにMPIプロセスを割り当て、各コアにOpenMPスレッドを割り当てる。





マルチコアクラスタにおける並列化（3）

- ハイブリッド並列化の例

外側ループをMPI並列化

```
MPI_Comm_size(MPI_COMM_WORLD, &size);

X = N1 / size;
for (i = 0; i < N1; i++)
{
  #pragma omp parallel do
  for (j = 0; j < N2; j++)
    ...
```

内側ループをOpenMP並列化

マルチコアクラスタにおける並列化（4）

- ハイブリッド並列化の長所
  - データを共有できるため、メモリを節約できる
  - より多くの（異なるレベルの）並列性を利用できる。
- ハイブリッド並列化の短所
  - プログラミングが難しい。
  - 必ずしも速くない。
- ノード（CPU）が非常に多くなると、長所が短所を上回る？

  cf. 京速コンピュータ「京」では、ハイブリッド並列化を推奨している。

OpenMPの性能(1)

- OpenMPの「わかりやすさ(高い抽象性)」は諸刃の剣。
  - 悪いプログラムも簡単に書けてしまう。
  - 性能の最後の一滴まで絞りつくすようなプログラムを書くのは難しい。
  - 特に、NUMA環境における不均一なメモリアクセスと、false sharingの発生による性能低下には注意を要する(が、発見も対処も簡単ではない)。

OpenMPの性能(2)

- プラットフォームや問題規模による。
- 特に、問題規模は重要。
  - 並列化のオーバヘッドと並列化のgainとのトレードオフ
- コア数(スレッド数)がいくら増えても、メモリの性能はあまり変わらないため、性能には限界がある。
- ……とはいえ、16並列くらいまでなら、多くの場合で特に問題なく使える。

まとめ

- これからの高速化には、並列化は必須
  - １６並列ぐらいまでなら、OpenMP
  - 特にマルチコアプロセッサでは、OpenMPが必須
- １６並列以上になれば、MPI（との併用）が必須
  - ただし、プログラミングのコストと実行時間のトレードオフ
  - 長期的には、MPIに変わるプログラミング言語が待たれる

single構文

```
#include <stdio.h>

void work1() {}
void work2() {}

void a12()
{
  #pragma omp parallel
  {
    #pragma omp single
      printf("Beginning work1.¥n");

    work1();

    #pragma omp single
      printf("Finishing work1.¥n");

    #pragma omp single nowait
      printf("Finished work1 and beginning work2.¥n");

    work2();
  }
}
```

parallel sections構文

```
      SUBROUTINE A11()

!$OMP PARALLEL SECTIONS

!$OMP SECTION
        CALL XAXIS()
!$OMP SECTION
        CALL YAXIS()
!$OMP SECTION
        CALL ZAXIS()

!$OMP END PARALLEL SECTIONS

      END SUBROUTINE A11
```

workshare構文

```
      SUBROUTINE A14_1(AA, BB, CC, DD, EE, FF, N)
      INTEGER N
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N),
     +     EE(N,N), FF(N,N)

!$OMP PARALLEL
!$OMP   WORKSHARE
          AA = BB
          CC = DD
          EE = FF
!$OMP   END WORKSHARE
!$OMP END PARALLEL

      END SUBROUTINE A14_1
```

lastprivate節

```
void a34 (int n, float *a, float *b)
{
  int i;

  #pragma omp parallel
  {
    #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];
    }

    a[i]=b[i]; /* i == n-1 here */
}
```

collapse節

```
      subroutine sub()

!$omp do collapse(2) private(i,j,k)
      do k = kl, ku, ks
        do j = jl, ju, js
          do i = il, iu, is
            call bar(a,i,j,k)
          end do
        end do
      end do
!$omp end do

end subroutine
```