

内容に関する質問は  
katagiri@cc.u-tokyo.ac.jp  
まで

# 第3回 OpenMPの基礎

東京大学情報基盤センター 片桐孝洋



# 講義日程と内容について

## ▶ 2015年度 CMSI計算科学技術特論A(1学期:木曜3限)

- ▶ 第1回:プログラム高速化の基礎、2015年4月9日
  - ▶ イントロダクション、ループアンローリング、キャッシュブロック化、数値計算ライブラリの利用、その他
- ▶ 第2回:MPIの基礎、2015年4月16日
  - ▶ 並列処理の基礎、MPIインターフェース、MPI通信の種類、その他
- ▶ 第3回:OpenMPの基礎、2015年4月23日
  - ▶ OpenMPの基礎、利用方法、その他
- ▶ 第4回:Hybrid並列化技法(MPIとOpenMPの応用)、2015年5月7日
  - ▶ 背景、Hybrid並列化の適用事例、利用上の注意、その他
- ▶ 第5回:プログラム高速化の応用、2015年5月14日
  - ▶ プログラムの性能ボトルネックに関する考えかた(I/O、単体性能(演算機ネック、メモリネック)、並列性能(バランス))、性能プロファイル、その他

## 教科書（演習書）

- ▶ 「並列プログラミング入門：  
サンプルプログラムで学ぶOpenMPとOpenACC」
  - ▶ 片桐 孝洋 著
  - ▶ 東大出版会、ISBN-10: 4130624563、ISBN-13: 978-4130624565、  
発売日：2015年5月25日（出版予定）
  - ▶ 【本書の特徴】
    - ▶ C言語、Fortran90言語で解説
    - ▶ C言語、Fortran90言語の複数のサンプルプログラムが  
入手可能（ダウンロード形式）
    - ▶ 本講義の内容を全てカバー
    - ▶ Windows PCで演習可能(Cygwin利用)。スパコンでも演習可能。
    - ▶ 内容は初級。初めて並列プログラミングを学ぶ人向けの  
入門書

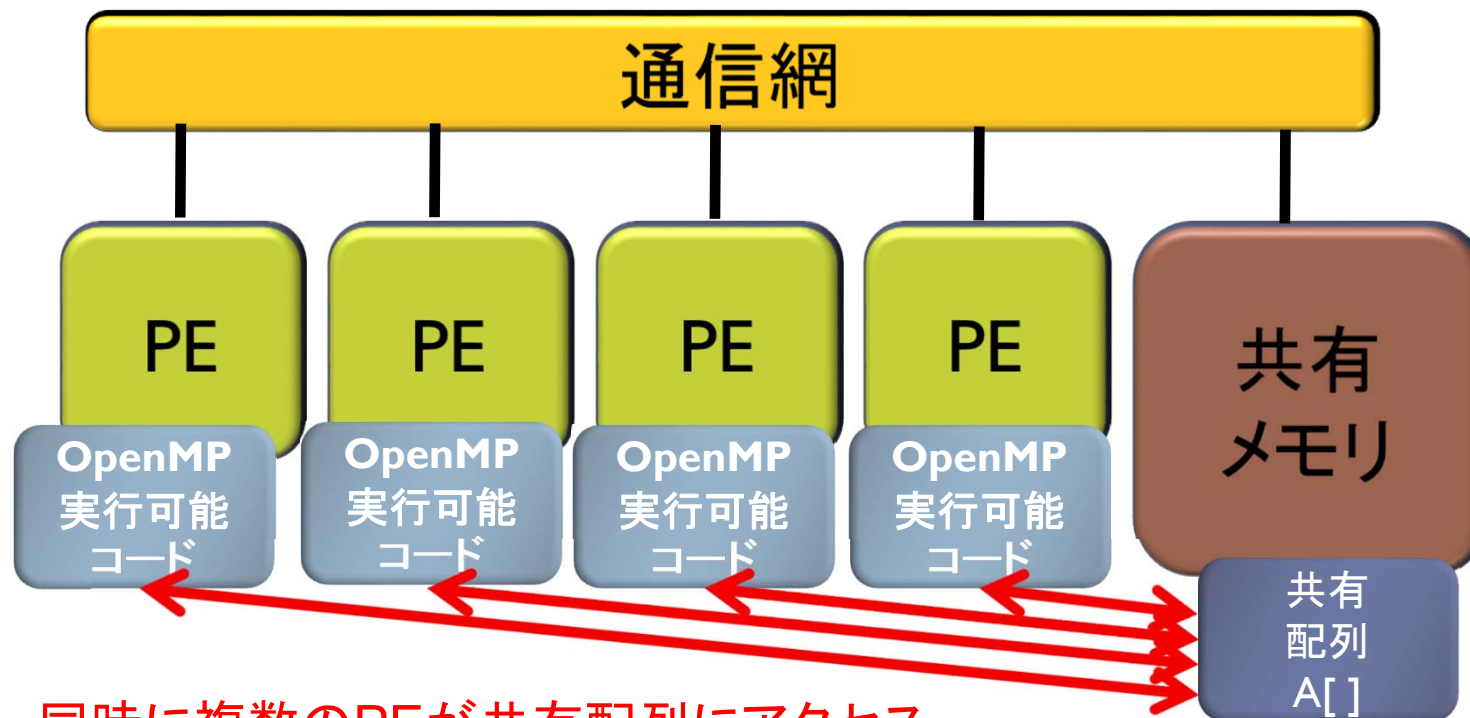
# OpenMP 超入門

指示文による簡単並列化

# OpenMPの概要

# OpenMPの対象計算機

- ▶ OpenMPは共有メモリ計算機のためのプログラム言語



同時に複数のPEが共有配列にアクセス

⇒ 並列処理で適切に制御をしないと、逐次計算の結果と一致しない

# OpenMPとは

---

- ▶ OpenMP (OpenMP C and C++ Application Program Interface Version 1.0)とは、共有メモリ型並列計算機用にプログラムを並列化する以下：
  1. 指示文
  2. ライブラリ
  3. 環境変数を規格化したものです。
- ▶ ユーザが、並列プログラムの実行させるための指示を与えるものです。  
コンパイラによる自動並列化ではありません。
- ▶ 分散メモリ型並列化(MPIなど)に比べて、データ分散の処理の手間が無い分、実装が簡単です。

# OpenMPとマルチコア計算機（その1）

- ▶ スレッド並列化を行うプログラミングモデル
- ▶ 近年のマルチコア計算機に適合
  - ▶ 経験的な性能:
    - 8スレッド並列以下の実行に向く
  - ▶ 8スレッドを超えるスレッド実行で高い並列化効率を出すには、プログラミングの工夫が必要
    1. メインメモリ - キャッシュ間のデータ転送能力が演算性能に比べ低い
    2. OpenMPで並列性を抽出できないプログラムになっている(後述)
- ▶ ノード間の並列化はOpenMPではできない
  - ▶ ノード間の並列化はMPIを用いる
  - ▶ 自動並列化コンパイラも、スレッド並列化のみ
    - ▶ HPF、XcalableMP(筑波大)などのコンパイラではノード間の並列化が可能だが、まだ普及していない



# OpenMPとマルチコア計算機（その2）

- ▶ 典型的なスレッド数
  - ▶ 16スレッド／ノード
    - ▶ FX10スーパーコンピュータシステム (Sparc64 IVfx)
  - ▶ 32～128スレッド／ノード
    - ▶ Fujitsu FX100 (Sparc64 VIfx)
    - ▶ HITACHI SR16000 (IBM Power7)
      - 32物理コア、64～128論理コア (SMT利用時)
  - ▶ 60～240スレッド／ノード
    - ▶ Intel Xeon Phi (Intel MIC (Many Integrated Core) 、Knights Conner)
      - 60物理コア、120～240論理コア (HT利用時)
- ▶ 100スレッドを超えたOpenMPによる実行形態が普及
  - ▶ 性能を出すためには、相当のプログラム上の工夫が必要

# OpenMPコードの書き方の原則

---

- ▶ C言語の場合
  - ▶ `#pragma omp`  
で始まるコメント行
- ▶ Fortran言語の場合
  - ▶ `!$omp`  
で始まるコメント行

# OpenMPのコンパイルの仕方

- ▶ 逐次コンパイラのコンパイルオプションに、OpenMP用のオプションを付ける
  - ▶ 例) 富士通Fortran90コンパイラ  
`f90 -Kfast,openmp foo.f`
  - ▶ 例) 富士通Cコンパイラ  
`fcc -Kfast,openmp foo.c`
- ▶ **注意**
  - ▶ **OpenMPの指示がないループは逐次実行**
  - ▶ コンパイラにより、自動並列化によるスレッド並列化との併用ができる場合があるが、できない場合もある
    - ▶ OpenMPの指示行がある行はOpenMPによるスレッド並列化、指示がないところはコンパイラによる自動並列化
    - ▶ 例) 富士通Fortran90コンパイラ  
`f90 -Kfast,parallel,openmp foo.f`

# OpenMPの実行可能ファイルの実行

- ▶ OpenMPのプログラムをコンパイルして生成した実行可能ファイルの実行は、そのファイルを指定することで行う
- ▶ プロセス数を、環境変数OMP\_NUM\_THREADSで指定
- ▶ 例) OpenMPによる実行可能ファイルがa.outの場合

```
$ export OMP_NUM_THREADS=16
```

```
$ ./a.out
```

- ▶ **注意**
  - ▶ 逐次コンパイルのプログラムと、OpenMPによるプログラムの実行速度が、OMP\_NUM\_THREADS=1にしても、異なることがある(後述)
    - ▶ この原因は、OpenMP化による処理の増加(オーバーヘッド)
    - ▶ 高スレッド実行で、このオーバーヘッドによる速度低下が顕著化
    - ▶ プログラミングの工夫で改善可能

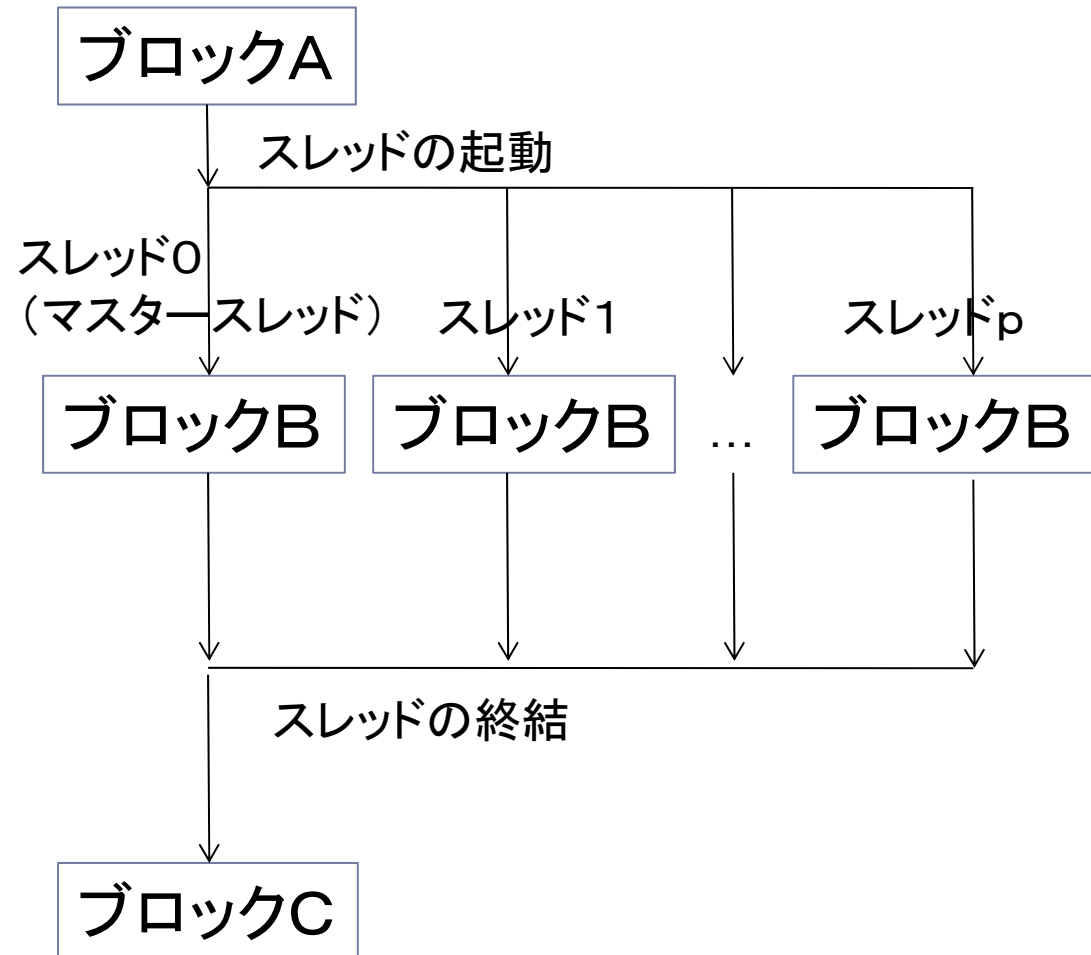
# OpenMPの実行モデル

# OpenMPの実行モデル（C言語）

## OpenMP指示文

```
ブロックA  
#pragma omp parallel  
{  
  ブロックB  
}  
ブロックC
```

※スレッド数pは、  
環境変数  
**OMP\_NUM\_THREADS**  
で指定する。

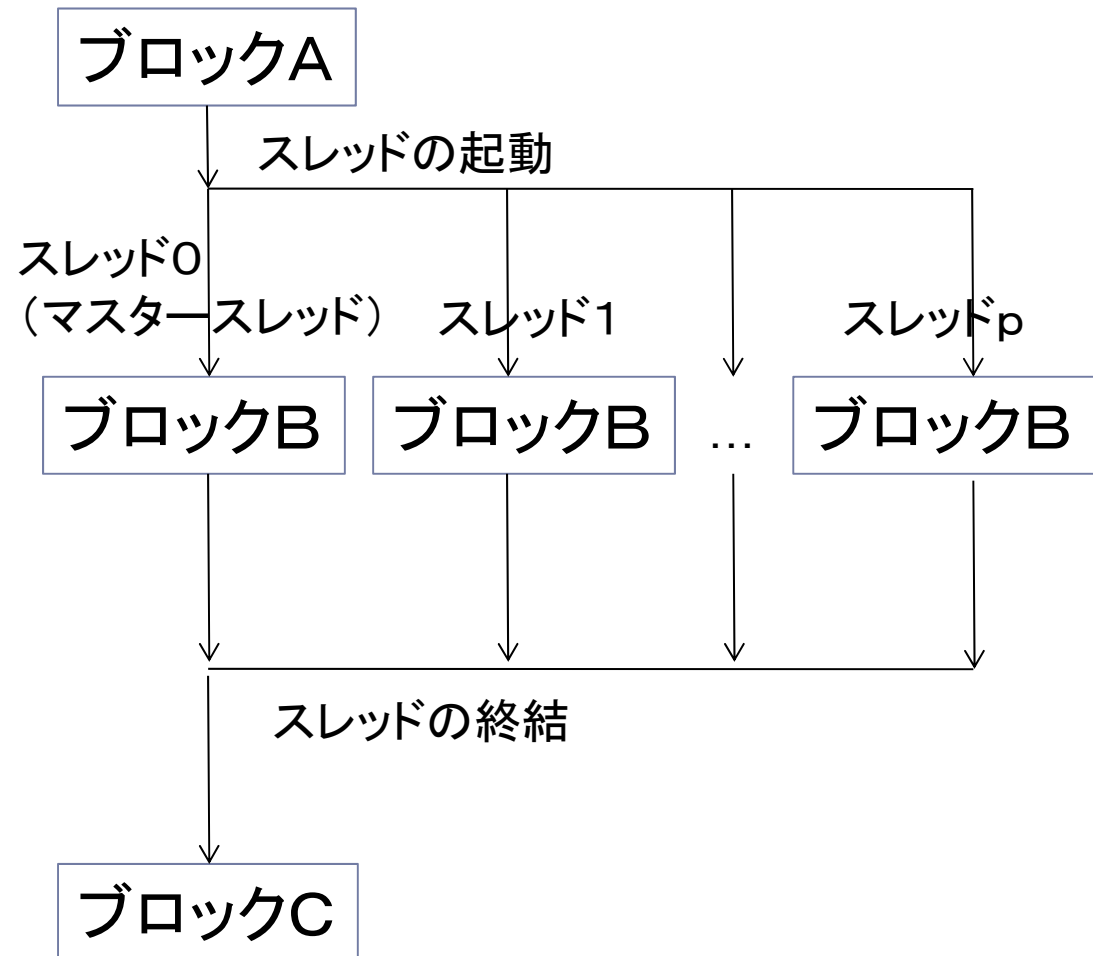


# OpenMPの実行モデル (Fortran言語)

## OpenMP指示文



※スレッド数 $p$ は、  
環境変数  
**OMP\_NUM\_THREADS**  
で指定する。



# Work sharing構文

- ▶ parallel指示文のように、複数のスレッドで実行する場合において、OpenMPで並列を記載する処理(ブロックB)の部分を**並列領域 (Parallel Region)**と呼ぶ。
- ▶ 並列領域を指定して、スレッド間で並列実行する処理を記述するOpenMPの構文を**Work Sharing構文**と呼ぶ。
- ▶ Work Sharing構文は、以下の2種がある。
  1. **並列領域内で記載するもの**
    - ▶ for構文(do構文)
    - ▶ sections構文
    - ▶ single構文(master構文)、など
  2. **parallel指示文と組み合わせるもの**
    - ▶ parallel for 構文(parallel do構文)
    - ▶ parallel sections構文、など



# 代表的な指示文

# For構文 (do構文)

```
#pragma omp parallel for  
for (i=0; i<100; i++){  
    a[i] = a[i] * b[i];  
}
```

※Fortran言語の場合は  
!\$omp parallel do  
~  
!\$omp end parallel do

上位の処理

スレッドの起動

スレッド0

```
for (i=0; i<25; i++){  
    a[i] = a[i] * b[i];  
}
```

スレッド1

```
for (i=25; i<50; i++){  
    a[i] = a[i] * b[i];  
}
```

スレッド2

```
for (i=50; i<75; i++){  
    a[i] = a[i] * b[i];  
}
```

スレッド3

```
for (i=75; i<100; i++){  
    a[i] = a[i] * b[i];  
}
```

スレッドの終結

下位の処理

※指示文を書くループが  
並列化をしても、  
正しい結果になることを  
ユーザが保障する。

## For構文の指定ができない例

```
for (i=0; i<100; i++) {  
    a[i] = a[i] +1;  
    b[i] = a[i-1]+a[i+1];  
}
```

- ループ並列化指示すると、  
逐次と結果が異なる  
(a[i-1]が更新されていない  
場合がある)

```
for (i=0; i<100; i++) {  
    a[i] = a[ ind[i] ];  
}
```

- ind[i]の内容により、  
ループ並列化できるか  
どうか決まる
- a[ind[i]]が既に更新された  
値でないとき、  
ループ並列化できる

# Sections構文

```
#pragma omp parallel sections
{
  #pragma omp section
  sub1();
  #pragma omp section
  sub2();
  #pragma omp section
  sub3();
  #pragma omp section
  sub4();
}
```

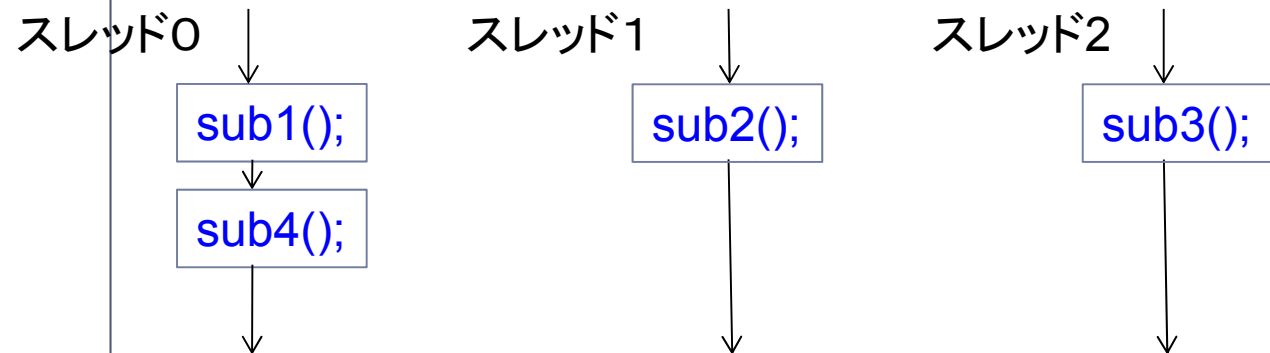
※Fortran言語の場合は

**!\$omp parallel sections**

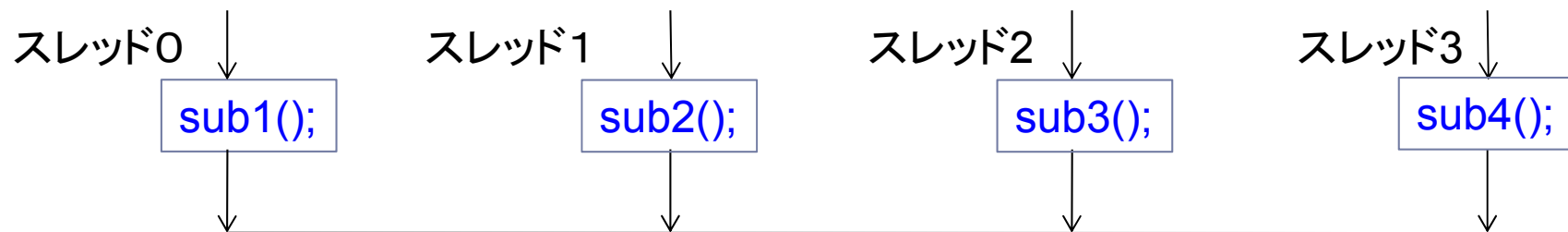
~

**!\$omp end parallel sections**

●スレッド数が3の場合



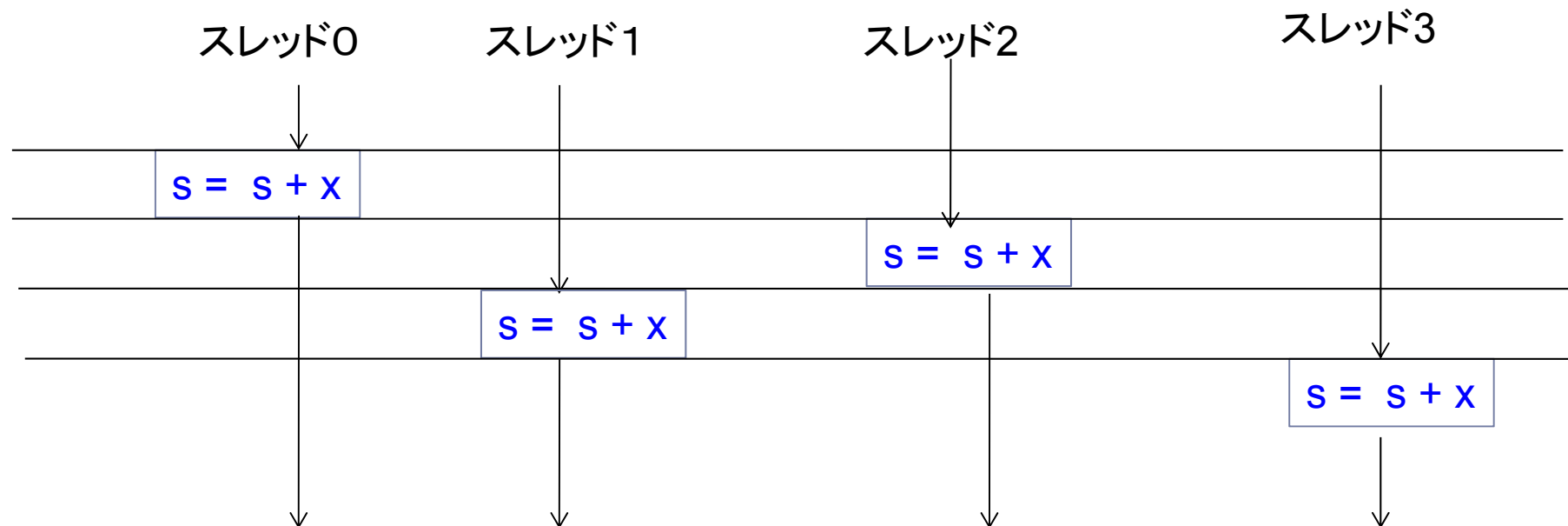
●スレッド数が4の場合



# Critical補助指示文

```
#pragma omp critical
{
    s = s + x;
}
```

※Fortran言語の場合は  
!\$omp critical  
～  
!\$omp end critical



# Private補助指示文

```
#pragma omp parallel for private(c)
for (i=0; i<100; i++){
    a[i] = a[i] + c * b[i];
}
```

ただし、c にループに入る前の値を  
代入して使う場合は **firstprivate(c)**

※変数cが各スレッドで  
別の変数を確保して実行  
→高速化される

上位の処理

スレッドの起動

スレッド0

```
for (i=0; i<25; i++){
    a[i] = a[i] + c0*b[i];
}
```

スレッド1

```
for (i=25; i<50; i++){
    a[i] = a[i] + c1*b[i];
}
```

スレッド2

```
for (i=50; i<75; i++){
    a[i] = a[i] + c2*b[i];
}
```

スレッド3

```
for (i=75; i<100; i++){
    a[i] = a[i] + c3*b[i];
}
```

スレッドの終結

下位の処理

## Private補助指示文の注意（C言語）

```
#pragma omp parallel for private( j )  
for (i=0; i<100; i++) {  
    for (j=0; j<100; j++) {  
        a[ i ] = a[ i ] + amat[ i ][ j ]* b[ j ];  
    }  
}
```

- ループ変数 **j** が、各スレッドで別の変数を確保して実行される。
- **private( j )** がいない場合、各スレッドで 共有変数の **j** のカウントを独立で行ってしまい、逐次と加算結果が異なる。  
→ 演算結果が逐次と異なり、エラーとなる。

## Private補助指示文の注意 (Fortran言語)

```
!$omp parallel do private( j )  
do i=1, 100  
  do j=1, 100  
    a( i ) = a( i ) + amat( i , j ) * b( j )  
  enddo  
enddo  
!$omp end parallel do
```

- ループ変数 **j** が、各スレッドで別の変数を確保して実行される。
- **private( j )** がない場合、各スレッドで 共有変数の **j** のカウントを独立で行ってしまい、逐次と加算結果が異なる。  
→ 演算結果が逐次と異なり、エラーとなる。



# リダクション補助指示文 (C言語)

- ▶ 内積値など、スレッド並列の結果を足しこみ、1つの結果を得たい場合に利用する
  - ▶ 上記の足しこみはスレッド毎に非同期になされる
  - ▶ reduction補助指示文が無いと、ddotは共有変数になるため、並列実行で逐次の結果と合わなくなくなる

```
#pragma omp parallel for reduction(+, ddot )  
for (i=1; i<=100; i++) {  
    ddot += a[ i ] * b[ i ]  
}
```

ddotの場所はスカラー変数のみ記載可能(配列は記載できません)

# リダクション補助指示文 (Fortran言語)

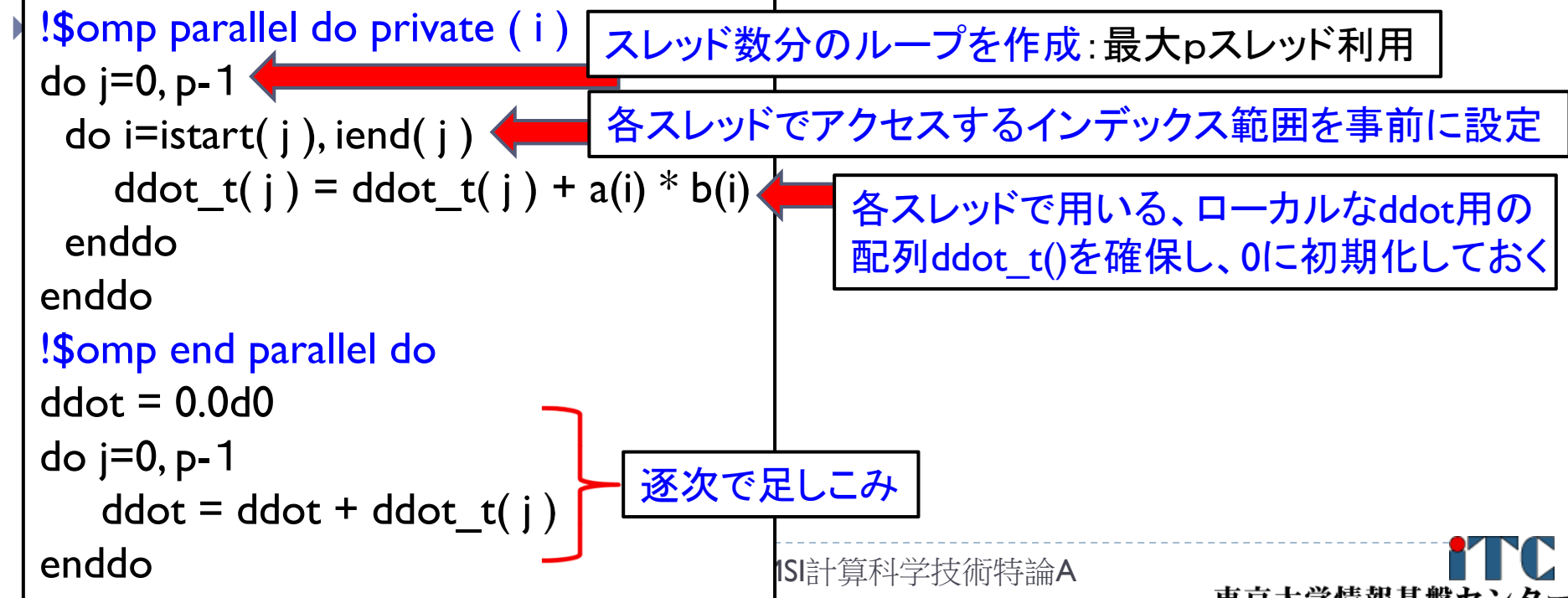
- ▶ 内積値など、スレッド並列の結果を足しこみ、1つの結果を得たい場合に利用する
  - ▶ 上記の足しこみはスレッド毎に非同期になされる
  - ▶ reduction補助指示文が無いと、ddotは共有変数になるため、並列実行で逐次の結果と合わなくなる

```
!$omp parallel do reduction(+, ddot )  
do i=1, 100  
    ddot = ddot + a(i) * b(i)  
enddo  
!$omp end parallel do
```

ddotの場所はスカラー変数のみ記載可能(配列は記載できません)

# リダクション補助指示文の注意

- ▶ reduction補助指示文は、排他的に加算が行われるので、性能が悪い
  - ▶ 経験的に、8スレッド並列を超える場合、性能劣化が激しい
- ▶ 以下のように、ddot用の配列を確保して逐次で加算する方法が高速な場合もある(ただし、問題サイズ、ハードウェア依存)



## その他、よく使うOpenMPの関数

# 最大スレッド数取得関数

- ▶ 最大スレッド数取得には、`omp_get_num_threads()`関数を利用する
- ▶ 型はinteger (Fortran言語)、int (C言語)

## ● Fortran90言語の例

```
use omp_lib
Integer nthreads

nthreads = omp_get_num_threads()
```

## ● C言語の例

```
#include <omp.h>
int nthreads;

nthreads = omp_get_num_threads();
```

# 自スレッド番号取得関数

- ▶ 自スレッド番号取得には、`omp_get_thread_num()`関数を利用する
- ▶ 型はinteger (Fortran言語)、int (C言語)

## ● Fortran90言語の例

```
use omp_lib
Integer myid

myid = omp_get_thread_num()
```

## ● C言語の例

```
#include <omp.h>
int myid;

myid = omp_get_thread_num();
```

# 時間計測関数

- ▶ 時間計測には、`omp_get_wtime()`関数を利用する
- ▶ 型はdouble precision (Fortran言語)、double (C言語)

## ● Fortran90言語の例

```
use omp_lib
double precision dts, dte

dts = omp_get_wtime()
  対象の処理
dte = omp_get_wtime()
print *, "Elapse time [sec.] =", dte-dts
```

## ● C言語の例

```
#include <omp.h>
double dts, dte;

dts = omp_get_wtime();
  対象の処理
dte = omp_get_wtime();
printf("Elapse time [sec.] = %lf ¥n",
      dte-dts);
```

## その他の構文



# Single構文

- ▶ Single補助指示文で指定されたブロックを、どれか1つのスレッドに割り当てる
- ▶ **どのスレッドに割り当てられるかは予測できない**
- ▶ `nowait`補助指示文を入れない限り、同期が入る

※Fortran言語の場合は  
**!\$omp single**  
~  
**!\$omp end single**

```
#pragma omp parallel for  
{  
    ブロックA  
    #pragma omp single  
    {   ブロックB   }  
    ...  
}
```

プログラムの開始

スレッドの起動

スレッド0  
(マスタースレッド)

スレッド1

スレッドp

ブロックA

ブロックA

...

ブロックA

ブロックB

同期処理

## Master構文

---

- ▶ 使い方は、single補助指示文と同じ
- ▶ ただし、master補助指示文で指定した処理（先ほどの例の「ブロックB」の処理）は、**必ずマスタースレッドに割り当てる**
- ▶ 終了後の同期処理が入らない
  - ▶ そのため、場合により高速化される

# Flush構文

- ▶ 物理メモリとの一貫性を取る
- ▶ Flush構文で指定されている変数のみ、その場所で一貫性を取る。  
それ以外の共有変数の値は、メモリ上の値との一貫性は無い。
  - ▶ 演算結果はレジスタ上に保存されるだけ。  
メモリに計算結果を書き込んでいない
  - ▶ つまり、flush補助指定文を書かないと、スレッド間で同時に足しこんだ結果が、実行ごとに異なる。
  - ▶ barrier補助指定文、critical補助指定文の出入口、parallel構文の出口、for、sections、single構文の出口では、暗黙的にflushされている。
- ▶ Flushを使うと性能は悪くなる。できるだけ用いない。

`#pragma omp flush` (対象となる変数名の並び)

省略すると、  
全ての変数が対象

# Threadprivate構文

- ▶ スレッドごとにプライベート変数にするが、スレッド内で大域アクセスできる変数を宣言する。
- ▶ スレッドごとに異なる値をもつ大域変数の定義に向く。
  - ▶ たとえば、スレッドごとに異なるループの開始値と終了値の設定

```
#include <omp.h>
int myid, nthreds, istart, iend;
#pragma omp threadprivate (istart, iend)
...
void kernel() {
    int i;
    for (i=istart; i<iend; i++) {
        for (j=0; j<n; j++) {
            a[i] = a[i] + amat[i][j] * b[j];
        }
    }
}
```

```
...
void main() {
    ...
    #pragma omp parallel private (myid, nthreds, istart, iend) {
        nthreds = omp_num_threads();
        myid = omp_get_thread_num();
        istart = myid * (n/nthreds);
        iend = (myid+1)*(n/nthreds);
        if (myid == (nthreds-1)) {
            nend = n;
        }
        kernel();
    }
```

スレッド毎に異なる値を持つ大域変数を、Parallel構文中で定義する

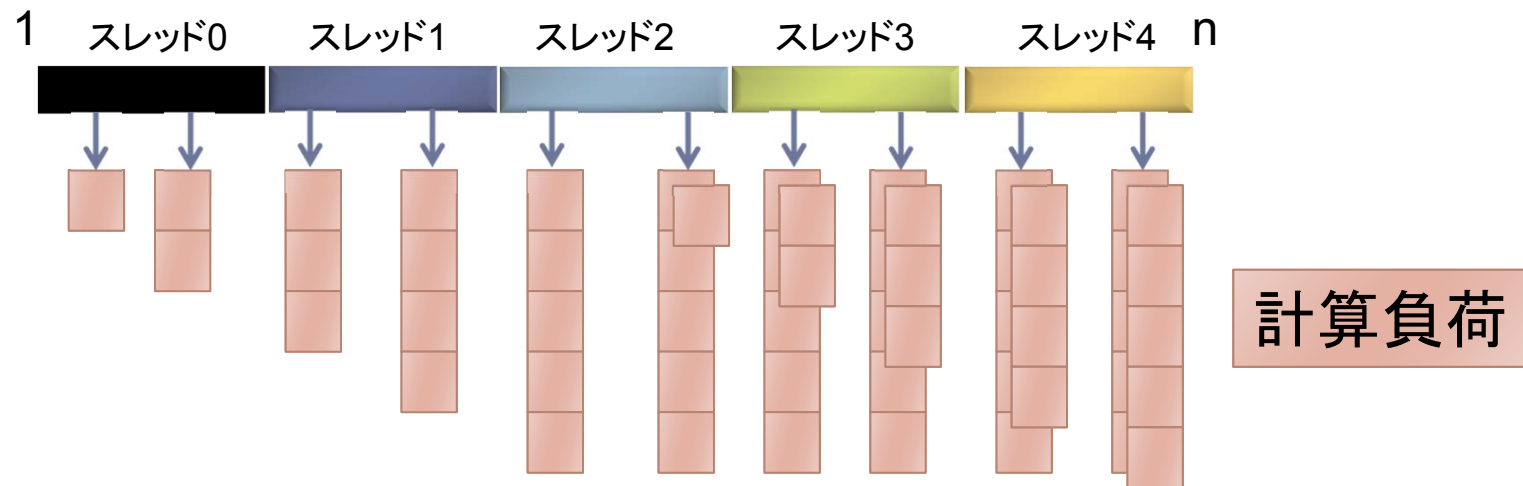
# スケジューリング

# スケジューリングとは（その1）

- ▶ Parallel do構文 (Parallel for構文) では、対象ループの範囲 (例えば1～nの長さ) を、単純にスレッド個数分に分割 (連続するように分割) して、並列処理をする。

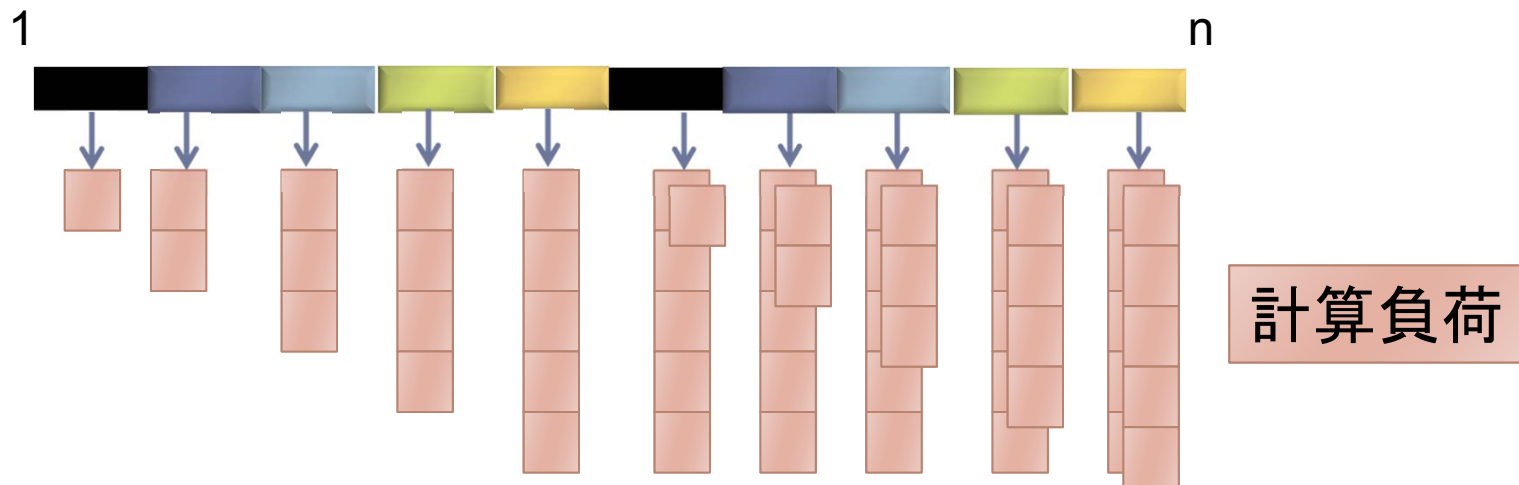


- ▶ このとき、各スレッドで担当したループに対する計算負荷が均等でないと、スレッド実行時の台数効果が悪くなる



## スケジューリングとは（その2）

- ▶ 負荷分散を改善するには、割り当て間隔を短くし、かつ、循環するように割り当てればよい。



- ▶ 最適な、割り当て間隔(**チャンクサイズ**とよぶ)は、計算機ハードウェアと、対象となる処理に依存する。
- ▶ 以上の割り当てを行う補助指示文が用意されている。

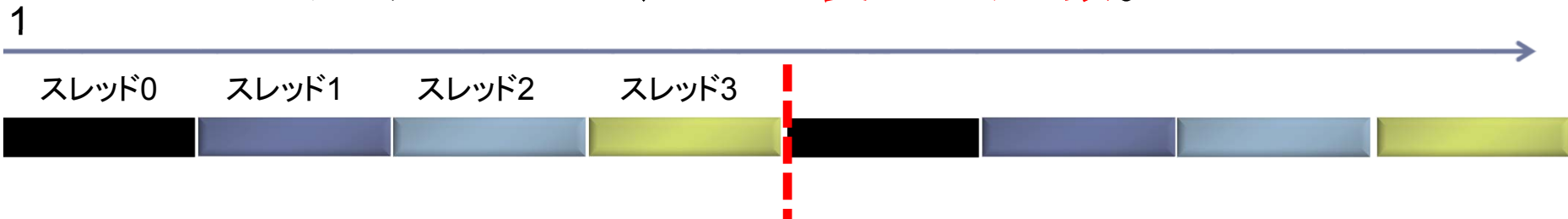
# ループスケジューリングの補助指定文 (その1)

## ▶ `schedule (static, n)`

- ▶ ループ長をチャンクサイズで分割し、スレッド0番から順番に（スレッド0、スレッド1、・・・というように、ラウンドロビン方式と呼ぶ）、循環するように割り当てる。

**nにチャンクサイズを指定できる。**

- ▶ Schedule補助指定文を記載しないときのデフォルトはstaticで、かつチャンクサイズは、**ループ長/スレッド数**。

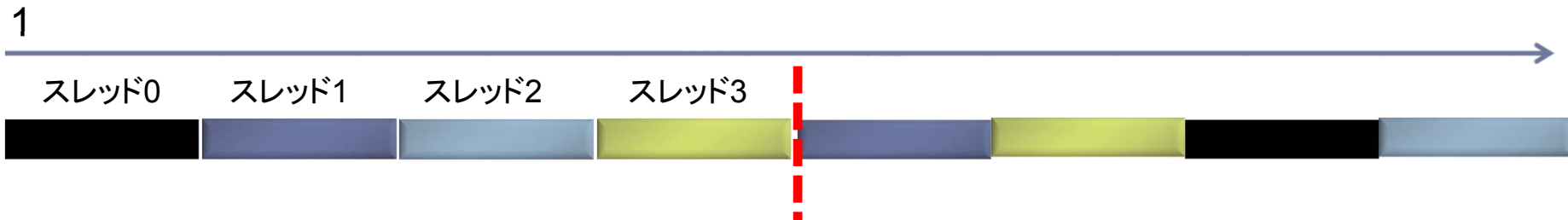




# ループスケジューリングの補助指定文 (その2)

## ▶ `schedule(dynamic, n)`

- ▶ ループ長をチャンクサイズで分割し、**処理が終了したスレッドから早い者勝ちで**、処理を割り当てる。nにチャンクサイズを指定できる。



# ループスケジューリングの補助指定文 (その3)

## ▶ `schedule(guided, n)`

- ▶ ループ長をチャンクサイズで分割し、徐々にチャンクサイズを小さくしながら、処理が終了したスレッドから早い者勝ちで、処理を割り当てる。nにチャンクサイズを指定できる。
- ▶ チャンクサイズの指定が1の場合、残りの反復処理をスレッド数で割ったおおよその値が各チャンクのサイズになる。
- ▶ チャンクサイズは 1 に向かって指数的に小さくなる。
- ▶ チャンクサイズに 1 より大きい k を指定した場合、チャンク サイズは指数的に k まで小さくなるが、最後のチャンクは k より小さくなる場合がある。
- ▶ チャンクサイズが指定されていない場合、デフォルトは 1 になる。

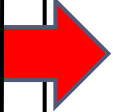


# ループスケジューリングの補助指示文の使い方

## ● Fortran90言語の例

j-ループの反復回数が  
間接参照により決まるので、  
i-ループの計算負荷が均等  
であるか不明。

実行時にしか、計算負荷の  
状況がわからないため、  
dynamicスケジューリング  
を適用



```
!$omp parallel do private( j, k ) schedule(dynamic,10)
do i=1, n
  do j=indj(i), indj (i+1)-1
    y( i ) = amat( j ) * x( indx( j ) )
  enddo
enddo
!$omp end parallel do
```

## ● C言語の例

```
#pragma omp parallel for private( j, k ) schedule(dynamic,10)
for (i=0; i<n; i++) {
  for ( j=indj(i); j<indj (i+1); j++) {
    y[ i ] = amat[ j ] * x[ indx[ j ] ];
  }
}
```

# ループスケジューリングにおけるプログラミング上の注意

- ▶ **dynamic、guidedのチャンクサイズは性能に大きく影響**
  - ▶ チャンクサイズが小さすぎると負荷バランスは良くなるが反面、システムのオーバヘッドが大きくなる。
  - ▶ 一方、チャンクサイズが大きすぎると負荷バランスが悪くなる半面、システムのオーバヘッドが小さくなる。
  - ▶ **両者のトレードオフがある。**
  - ▶ 実行時のチャンクサイズのチューニングが必須で、チューニングコストが増える。
- ▶ **staticのみで高速実装ができる（場合がある）**
  - ▶ dynamicなどの実行時スケジューリングは、システムのオーバーヘッドが入るが、staticはオーバーヘッドは（ほとんど）無い。
  - ▶ **事前に負荷分散が均衡となるループ範囲を調べた上で、staticスケジューリングを使うと、最も効率が良い可能性がある。**
  - ▶ ただし、プログラミングのコストは増大する

# Staticスケジューリングのみで負荷バランスを均衡化させる実装例

## ▶ 疎行列 - ベクトル積へ適用した例（詳細は後述）

```
!$omp parallel do private(S, J_PTR,I)
DO K=1, NUM_SMP
DO I=KBORDER(K-1)+1, KBORDER(K)
S=0.0D0
DO J_PTR=IRP(I), IRP(I+1)-1
S=S + VAL( J_PTR ) * X(ICOL( J_PTR))
END DO
Y(I)=S
END DO
END DO
!$omp end parallel do
```

スレッド個数文のループ  
(スレッドごとのループ担当範囲を知るために必要)

事前に調べて設定しておいた、  
負荷分散が均衡となる  
スレッドごとのループ範囲  
(各スレッドは、連続しているが、  
不均衡なループ範囲を設定)

実行前に、各スレッドが担当するループ範囲について、  
連続する割り当てで、かつ、それで負荷が均衡する  
問題に適用できる。

※実行時に負荷が動的に変わっていく場合は適用できない

# OpenMPのプログラミング上の注意 (全般)

# OpenMPによるプログラミング上の注意点

---

- ▶ OpenMP並列化は、

**parallel構文を用いた単純なforループ並列化**

が主になることが多い。

- ▶ 複雑なOpenMP並列化はプログラミングコストがかかるので、OpenMPのプログラミング上の利点が失われる
- ▶ parallel構文による並列化は

**private補助指示文の正しい使い方**

を理解しないと、バグが生じる！

# Private補助指示文に関する注意（その1）

- ▶ OpenMPでは、対象となる直近のループ変数以外は、private変数で指定しない限り、全て**共有変数**になる。
  - ▶ デフォルトの変数は、スレッド間で個別に確保した変数でない

## ● ループ変数に関する共有変数の例

```
!$omp parallel do
```

```
do i=1, 100
```

```
do j=1, 100
```

```
tmp = b(i) + c(i)
```

```
a(i) = a(i) + tmp
```

```
enddo
```

```
enddo
```

```
!$omp end parallel do
```

宣言なしにプライベート変数として確保されるのは、このi-ループ変数のみ

このj-ループ変数は、private宣言なしでは共有変数になる  
←スレッド間で早い者勝ちで加算 ←並列実行時にバグ

この変数tmpは、private宣言なしでは共有変数になる  
←スレッド間で早い者勝ちで値が代入 ←並列実行時にバグ



## Private補助指示文に関する注意（その2）

- ▶ Private補助指示文に記載する変数を減らすため、  
対象部分を関数化し、かつ、その関数の引数を増やすと、  
関数呼び出し時間が増加し、スレッド並列化の効果を  
相殺することがある

### ● 呼び出し関数の引数が多い例

```
!$omp parallel do  
do i=1, 100  
  call foo(i, arg1, arg2, arg3,  
           arg4, arg5, ....., arg100)  
enddo  
!$omp end parallel do
```

関数引数は自動的にプライベート変数になるため、private補助指示文に記載する変数を削減できる

← しかし、関数呼び出し時のオーバーヘッドが増加する

← スレッド実行時においても、関数呼び出しのオーバーヘッドが無視できなくなり、台数効果が制限される

※解決法: 大域変数で引き渡して引数を削減

# Private補助指示文に関する注意のまとめ

---

- ▶ OpenMPでは、宣言せずに利用する変数は、すべて共有変数 (shared variable) になる
- ▶ C言語の大域変数、Fortran90言語のcommon変数は、そのままでは共有変数になる
  - ▶ プライベート変数にしたい場合は、Threadprivate宣言が必要
- ▶ 外側ループをParallel構文などで並列化する場合
  - ▶ ループ内で呼ばれる関数(手続き)内で宣言される変数はPrivateになる
  - ▶ C言語で、ループ内で明示的に宣言される変数 (例: `int a;`) は Privateになる。

# Parallel構文の入れ子に関する注意（その1）

- ▶ Parallel構文は、do補助指示文で分離して記載できる
- ▶ **1ループが対象の場合**、分離するとdo補助指示文の場所でループごとにforkするコードを生成するコンパイラがあり、速度が低下する場合がある
  - ▶ **この逆の場合もある。双方を確認する必要あり。**

```
!$omp parallel
!$omp do private(j,tmp)
do i=1, 100
  do j=1, 100
    tmp = b(j) + c(j)
    a(i) = a(i) + tmp
  enddo
enddo
!$omp end do
!$omp end parallel
```

Parallel構文の  
対象が1ループ  
なら parallel do  
で指定

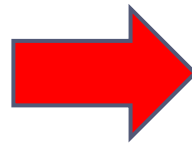


```
!$omp parallel do private(j,tmp)
do i=1, 100
  do j=1, 100
    tmp = b(j) + c(j)
    a(i) = a(i) + tmp
  enddo
enddo
!$omp end parallel do
```

## Parallel構文の入れ子に関する注意（その2）

- ▶ Parallel構文は、do補助指示文で分離して記載できる
- ▶ 複数ループの内側を並列化したい場合は、分離した方が高速になる
  - ▶ ただし、外側ループを並列化できる時はその方が性能が良い
  - ▶ 外側ループにデータ依存があり、並列化できない場合

```
do i=1, n
!$omp parallel do
  do j=1, n
    <並列化できる式>
  enddo
!$omp end parallel do
enddo
```



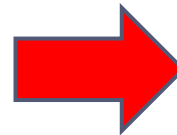
```
!$omp parallel
do i=1, n
!$omp do
  do j=1, n
    <並列化できる式>
  enddo
!$omp end do
enddo
!$omp end parallel
```

# データ依存関係を壊しバグになる例

- ▶ 間接参照があるインデックスに対して加算する例
  - ▶ 間接参照のパターン、および、スレッド実行のタイミング次第で、逐次処理と結果が一致し、正常動作だと勘違いする場合がある
    - ▶ 理論的には間違っている
  - ▶ OpenMPの共有変数は、データ一貫性の保証はしない
    - ▶ データの一貫性の保証には、critical補助指定文などによる相互排除が必要

## ● バグになるプログラム例

```
!$omp parallel do private( j )
do i=1, n
  j = indx( i )
  a( j ) = a( j ) + 1
enddo
!$omp end parallel do
```



```
!$omp parallel do private( j )
do i=1, n
  j = indx( i )
  !$omp critical
    a( j ) = a( j ) + 1
  !$omp end critical
enddo
!$omp end parallel do
```

# Critical補助指示文による速度低下

- ▶ 先述のように、critical補助指示文を入れないといけない場合、特に、**高スレッド数での実行で性能が低下する**
  - ▶ ハードウェアサポートがある場合、atomic節による実装のほうが高速であることが多いが、スレッド数が増えると性能が低下する
- ▶ 高性能化するには、基本的にはアルゴリズムを変更するしかない。
- ▶ この場合、以下の3つのアプローチがある。
  1. **スレッド内アクセスのみに限定し、critical補助指示文をはずす**
    - ▶ 間接参照されるデータについて、理論的に、割り当てられたスレッド内のデータしかアクセスしないように、アルゴリズムを変更する
  2. **スレッド間アクセスを最小化**
    - ▶ criticalの並列領域に同時に入るスレッド数が減るように、間接参照するデータを事前に調べ、間接参照するデータの順番を変更する。
  3. **スレッド間アクセス部分をループから分離し、逐次処理にする**
    - ▶ 例) 内積演算におけるリダクション補助指定文

# OpenMPを用いた並列化の欠点 (その1)

- ▶ OpenMPは単純なループを並列化することに向く
- ▶ 実用アプリケーションにおける複雑なループは、そのままではOpenMP化に向いていないことがある。
  - I. private補助指示文中に書かれる変数名の数が膨大になる
    - ▶ 外側ループからOpenMP並列化する場合、内部で使っている変数の数が多いことがある
    - ▶ private変数リストに変数を書き忘れても、コンパイラによるエラーは出ない。(並列化の責任はユーザにあるため)
    - ▶ 実行すると、タイミングに依存し計算結果が逐次と異なる。どこが間違っているかわからないので、デバックが大変になる。
    - ▶ 解決策:コンパイラによっては、最適化情報を出力することができる。その情報から、ちゃんとprivate化されているか確認する。

# OpenMPを用いた並列化の欠点 (その2)

- 2. 高スレッド実行時に性能が出ない場合のチューニングが困難
  - ▶ 一般に、8スレッド未満では性能が出るが、8スレッド以上で性能が劣化する。
    - 1. 近年のハードウェアはメモリアクセスの性能が低い
    - 2. ループそのものに並列性がない(ループ長が短い)
  - ▶ 解決するには、アルゴリズムの変更、実装の変更、が必要になり、OpenMPの利点である容易なプログラミングを損なう
- 3. 複雑なスレッドプログラミングには向かない
  - ▶ 単純な数値計算のカーネルループを、parallel for構文で記載する方針で仕様が作られている(と思われる)
  - ▶ 複雑な処理は、PthreadなどのnativeなスレッドAPIで書くほうがやりやすい



# プログラム実例

# 行列-行列積のコードのOpenMP化の例 (C言語)

- ▶ 以下のようなコードになる

```
#pragma omp parallel for private (j, k)
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        for(k=0; k<n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

# 行列-行列積のコードのOpenMP化の例 (Fortran言語)

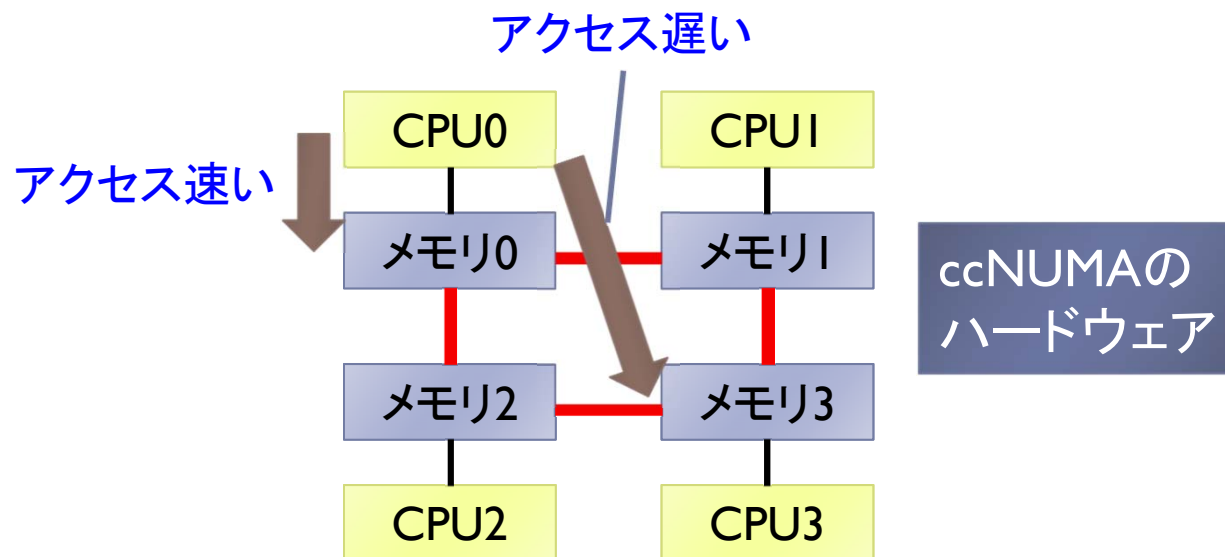
- ▶ 以下のようなコードになる

```
!$omp parallel do private (j, k)
do i=1, n
  do j=1, n
    do k=1, n
       $C(i, j) = C(i, j) + A(i, k) * B(k, j)$ 
    enddo
  enddo
enddo
!$omp end parallel do
```

# OpenMPの高速化技法： ファーストタッチ

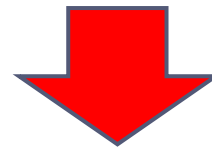
# ファーストタッチとは

- ▶ **ファーストタッチ**とは、マルチコア計算機の中でも、**ccNUMA (Cache Coherent Non-Uniform Memory Access)**のハードウェア向けの、メモリ最適化の方法
- ▶ OpenMPによる並列プログラミングでも重要な技法
- ▶ ccNUMAのメモリ構造の特性を利用する



# ファーストタッチの原理

- ▶ ccNUMA型のハードウェアでは、確保した配列は、各コアで、その配列に初めてアクセスした時、各コアに最も近いメモリに配列が置かれる
- ▶ この原理を利用し、本計算と同じデータ・アクセスパターン（＝ループ構造）で、プログラム上最も先に、OpenMP指示文を用いて配列を初期化すると、CPUに近いメモリに配列データがセットされる



- ▶ 本計算と同じループ構造で、確保した配列の初期化（例えば0クリア、もしくは、データのセット）をするだけで、ファーストタッチが実現できる

# ファーストタッチの例 (C言語の例)

```
#pragma omp parallel for private( j )  
for (i=0; i<100; i++) {  
    for (j=0; j<100; j++) {  
        a[ i ] = 0.0;  
        amat[ i ][ j ] = 0.0;  
    }  
    ....  
}
```

ファーストタッチの  
ための初期化  
(プログラムの  
一番最初に  
実行すること)

```
#pragma omp parallel for private( j )  
for (i=0; i<100; i++) {  
    for (j=0; j<100; j++) {  
        a[ i ] = a[ i ] + amat[ i ][ j ] * b[ j ];  
    }  
}
```

ファーストタッチ  
データを利用した  
本計算

# ファーストタッチの例 (Fortran言語の例)

```
!$omp parallel do private( j )  
do i=1, 100  
  do j=1, 100  
    a( i ) = 0.0d0  
    amat( i , j ) =0.0d0  
  enddo  
enddo  
!$omp end parallel do
```

ファーストタッチの  
ための初期化  
(プログラムの  
一番最初に  
実行すること)

```
....  
!$omp parallel do private( j )  
do i=1, 100  
  do j=1, 100  
    a( i ) = a( i ) + amat( i , j ) * b( j )  
  enddo  
enddo  
!$omp end parallel do
```

ファーストタッチ  
データを利用した  
本計算



# ファーストタッチの効果の例

- ▶ T2Kオープンスパコン(1ノード16スレッド)
- ▶ AMD Quad Core Opteron (Barcelona)
  - ▶ 4ソケット、1ソケットあたり4コア、合計16コアのccNUMA型計算機
- ▶ 疎行列 - ベクトル積の演算(数値計算ライブラリXabclibの実装例)

```
!$omp parallel do private(S,J_PTR,I)
```

```
DO K=1, NUM_SMP
```

```
DO I=KBORDER(K-1)+1,KBORDER(K)
```

```
S=0.0D0
```

```
DO J_PTR=IRP(I),IRP(I+1)-1
```

```
S=S+VAL( J_PTR ) * X(ICOL( J_PTR ))
```

```
END DO
```

```
Y(I)=S
```

```
END DO
```

```
END DO
```

```
!$omp end parallel do
```

各スレッドが担当する疎行列の  
行インデックス

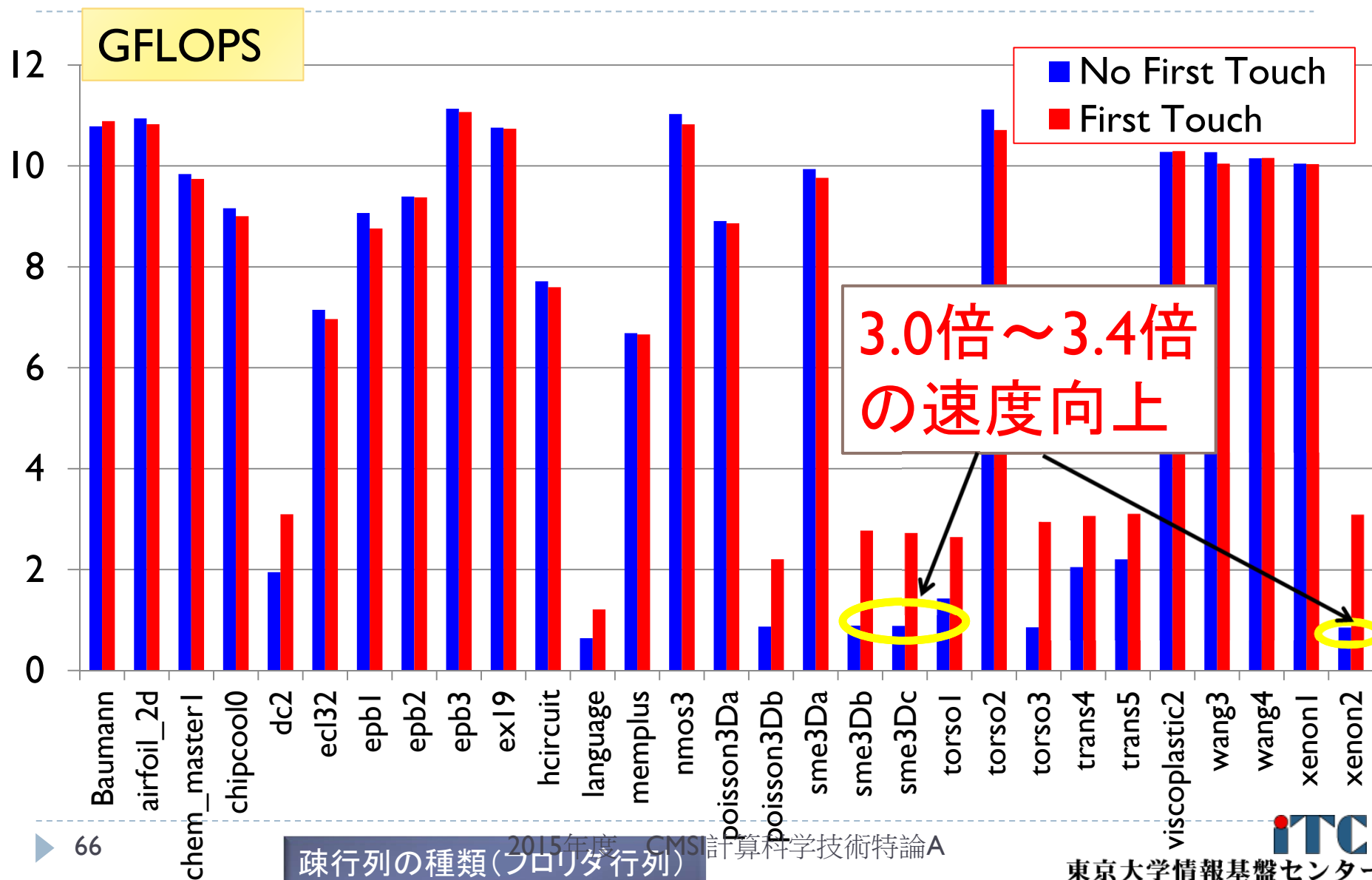
計算中の行の非ゼロ要素へのアクセス

疎行列 - ベクトル積の演算

右辺  $b$  のインデックス(間接参照)

疎行列格納形式  
CRS (Compressed Row Storage)

# 疎行列-ベクトル積でのファーストタッチ の効果 (AMD Quad Core Opteron, 16スレッド)



# ファーストタッチの効果が大きい行列

## ▶ sme3Da

- ▶ <http://www.cise.ufl.edu/research/sparse/matrices/FEMLAB/sme3Da.html>
- ▶ 非ゼロ要素分布は、行列の全体に広がっている
- ▶ number of rows: 12,504
- ▶ 行列サイズが小さい

←行列Aが最適配置、かつ、右辺 $b$ 全体がキャッシュに入る

## ▶ xenon2

- ▶ <http://www.cise.ufl.edu/research/sparse/matrices/Ronis/xenon2.html>
- ▶ 形状は、「三重対角行列」に近い

三重対角行列  
←ccNUMAで行列Aと右辺 $b$ の最適配置が可能

# ファーストタッチの実装上の注意

- ▶ **ccNUMAのアーキテクチャでないと効果がない**
  - ▶ 京コンピュータ、FX10、はccNUMAではないため、効果がない
- ▶ **対象となる配列を自ら確保し、演算も自ら行う「手製の」プログラムでないと効果がない**
- ▶ **数値計算ライブラリを使う場合**
  - ▶ 配列データはユーザが用意する。
  - ▶ 一般的に、配列データの値を設定するプログラムが先に動いて、その後、数値計算ライブラリを呼ぶ。
  - ▶ このとき、数値計算ライブラリ内でのアクセスパターンがわからないので、配列データを設定するプログラムのアクセスパターンが数値計算ライブラリ内のデータアクセスパターンと異なる。
  - ▶ 以上の理由から、ファーストタッチできない。

## 参考文献

---

- ▶ 佐藤三久 著:「OpenMP 並列プログラミング入門」  
<http://www.ccs.tsukuba.ac.jp/workshop/HPCseminar/2012/material/2012-05-openmp.pdf>
- ▶ 黒田久泰 著:「C言語によるOpenMP入門」  
[http://www.cc.u-tokyo.ac.jp/support/kosyu/03/kosyu-openmp\\_c.pdf](http://www.cc.u-tokyo.ac.jp/support/kosyu/03/kosyu-openmp_c.pdf)
- ▶ 南里豪志、天野浩文、渡部善隆 著:  
「OpenMP入門(1)～(3)」  
<http://www2.cc.kyushu-u.ac.jp/scp/system/library/OpenMP/openmp0109.pdf>  
<http://www2.cc.kyushu-u.ac.jp/scp/system/library/OpenMP/openmp0201.pdf>  
<http://www2.cc.kyushu-u.ac.jp/scp/system/library/OpenMP/openmp0209.pdf>

# OpenMP 4.0

# OpenMP 4.0

---

- ▶ 2013年7月仕様公開
  - ▶ <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>
- ▶ デバイス(GPU等)へのOpenMP演算のオフロード指定
  - ▶ Target構文
- ▶ 複数の並列デバイスを指定
  - ▶ Terms構文
- ▶ SIMD指定
  - ▶ SIMD構文
- ▶ スレッドとコアへの割り当て指定(NUMAアフィニティ)
  - ▶ Proc\_bind節
- ▶ GPU利用について、後述のOpenACCと同等の機能

# OpenACCへの展開

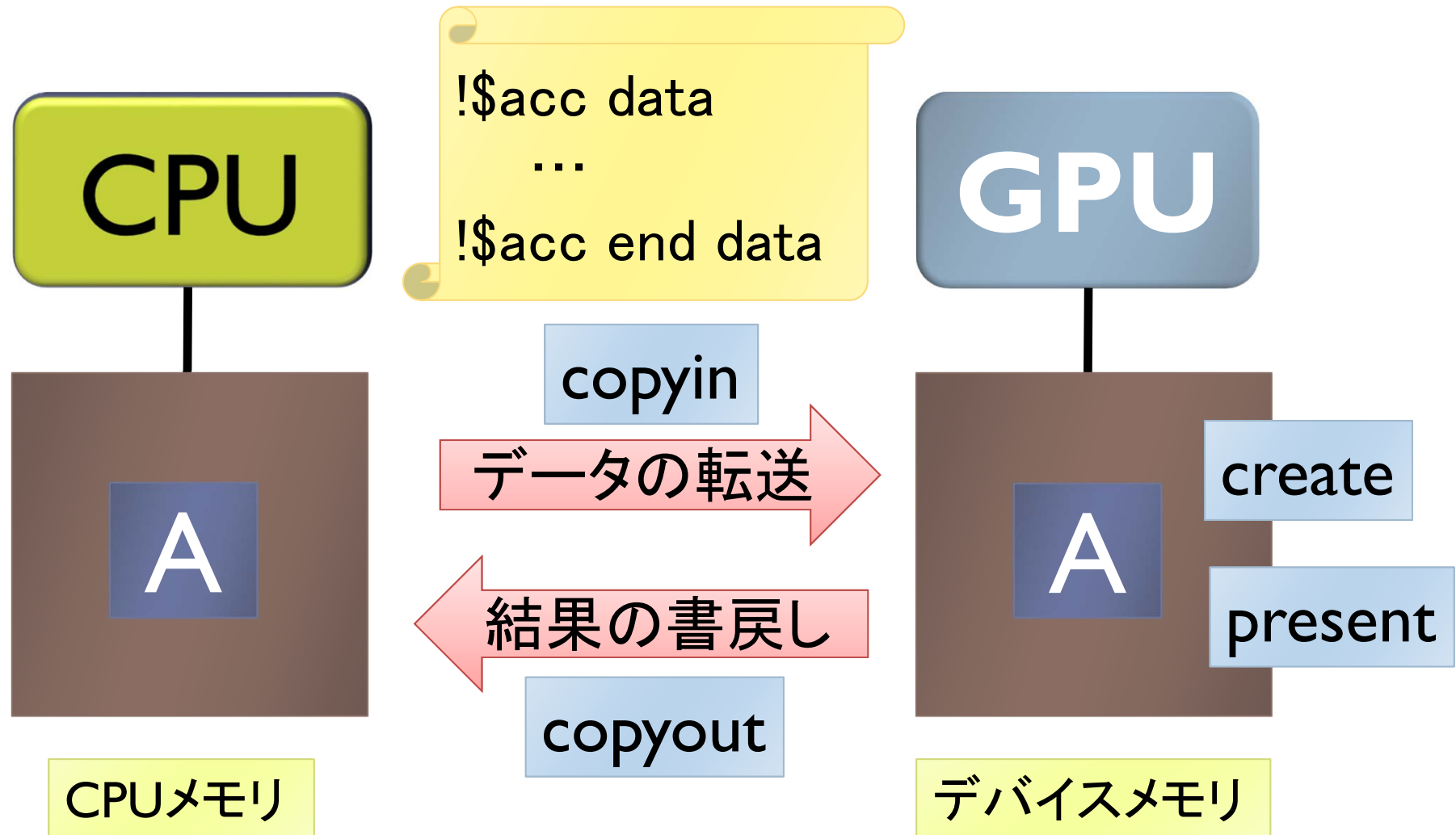


# OpenACCへの展開

---

- ▶ GPUを、OpenMPのように、ディレクティブで指定して使う  
OpenACCが普及しつつある
  - ▶ OpenMP 4.0でもGPUを扱えるようになるため、どちらが普及するかわからない
- ▶ OpenMP化されたプログラムは、比較的簡単に、OpenACCに変換できる
  - ▶ OpenMP の Parallel構文  
→ OpenACC の Kernel構文 か Parallel構文 に書き換え
- ▶ 注意する点は:
  - ▶ CPU→GPU、および、GPU→CPUのデータ移動の最小化
  - ▶ データ転送の対象となる配列を指定するData構文が重要

# Data構文の節



```
do iter = 1, MAX_ITER
```

```
!$acc kernels
```

```
do i=1, n
```

```
do j=1, n
```

```
b(i) = A(i, j) * ...
```

```
enddo
```

```
enddo
```

```
!$acc end kernels
```

```
...
```

```
!$acc kernels
```

```
do i=1, n
```

```
do j=1, n
```

```
b(i) = b(i) + A(i, j) * ...
```

```
enddo
```

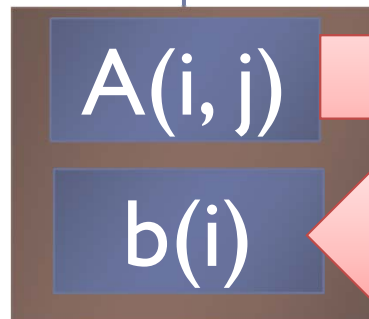
```
enddo
```

```
!$acc end kernels
```

```
...
```

```
enddo
```

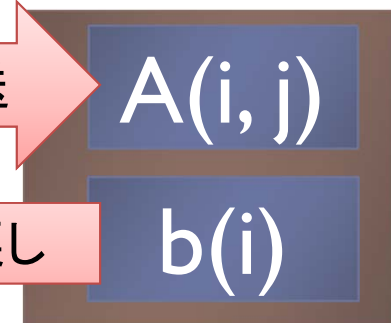
CPUメモリ



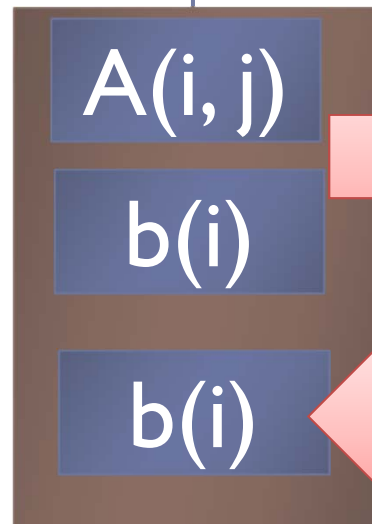
データの転送

結果の書戻し

デバイスメモリ



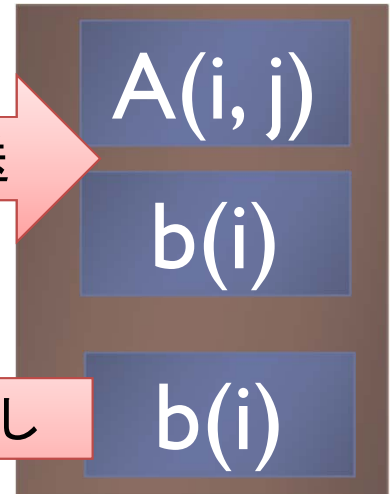
CPUメモリ



データの転送

結果の書戻し

デバイスメモリ



```

!$acc data copyin(A) create(b)
  do iter = 1, MAX_ITER
!$acc data present(A, b)
!$acc kernels
  do i=1, n
    do j=1, n
      b(i) = A(i, j) * ...
    enddo
  enddo
!$acc end kernels
!$acc end data
  ...
!$acc data present(A, b)
!$acc kernels
  do i=1, n
    do j=1, n
      b(i) = b(i) + A(i, j) * ...
    enddo
  enddo
!$acc end kernels
!$acc end data
  ...
enddo
!$acc end data

```

CPUメモリ

デバイスメモリ

A(i, j)

データの転送

A(i, j)

b(i)

デバイスメモリ

A(i, j)

b(i)

デバイスメモリ上にあるデータのみで演算  
(CPUメモリからの転送、および、  
CPUメモリへの書き戻しが無い)

# レポート課題（その1）

## ▶ 問題レベルを以下に設定

問題のレベルに関する記述:

- L00: きわめて簡単な問題。
- L10: ちょっと考えればわかる問題。
- L20: 標準的な問題。
- L30: 数時間程度必要とする問題。
- L40: 数週間程度必要とする問題。複雑な実装を必要とする。
- L50: 数か月程度必要とする問題。未解決問題を含む。

※L40以上は、論文を出版するに値する問題。

## ▶ 教科書のサンプルプログラムは以下が利用可能 (ただし、MPIの部分をコメントアウトする必要あり)

- ▶ [Mat-Mat-noopt-fx.tar](#)
- ▶ [Mat-Vec-fx.tar](#)

## レポート課題（その2）

---

1. [L10] 行列 - 行列積のコードをOpenMPで並列化せよ。  
また、1スレッド実行に対する台数効果を測定せよ。
2. [L10] 行列 - 行列積のコードについて、ファーストタッチを実装し、性能を評価せよ。
3. [L20] 疎行列 - 行列積のコードについて、OpenMPで並列化せよ。また、1スレッド実行に対する台数効果を測定せよ。

## レポート課題（その3）

---

4. [L10] データスコープ属性とは何か調べよ。また、firstprivate, lastprivate補助指示文の機能は何かを調べよ。
5. [L10] Barrier指示文、Nowait補助指示文について調べよ。またどのように利用するか例を記載して説明せよ。
6. [L10] 本講義で取り上げていない、OpenMPの実行時ライブラリ関数を調べ、その機能と利用方法を記せ。
7. [L10] OMP\_NUM\_THREADS以外のOpenMPで定義された環境変数を調べ、その機能を説明せよ。

## レポート課題（その4）

---

8. [L10] スケジューラの補助指示構文runtimeの機能調べよ。  
また、OpenMPの環境変数との関係を説明せよ。
9. [L15] OpenMP version 3.0、もしくは、4.0の仕様を調べよ。
10. [L15] OpenACC version 1.0、もしくは2.0 の仕様を調べよ。
11. [L10~] 自分の持っている逐次コードを、OpenMPで並列化せよ。スレッド数を変化させて、台数効果を調べよ。