高性能プログラミング技法の基礎（２）

東京大学情報基盤センター　准教授　片桐孝洋

２０１５年６月２日（火）　８：３０－１０：１５

東京大学情報基盤センター
Information Technology Center,The University of Tokyo

講義日程（工学部共通科目）

レポートおよびコンテスト課題
（締切：
2015年8月3日（月）24時　厳守

- ~~4月14日：ガイダンス~~

1. ~~4月21日~~
   - ~~並列数値処理の基本演算（座学）~~
2. ~~4月28日：座学のみ~~
   - ~~ソフトウエア自動チューニング~~
   - ~~非同期通信~~
3. ~~5月12日：スパコン利用開始~~
   - ~~ログイン作業、テストプログラム実行~~
4. ~~5月19日~~
   - ~~高性能演算技法1（ループアンローリング）~~
5. 6月2日（8：30−10：15）
   - 高性能演算技法2（キャッシュブロック化）
5. 6月2日（10：25−12：10）
   - 行列-ベクトル積の並列化
6. 6月9日（8：30−10：15）
   ★大演習室2
   - べき乗法の並列化
7. 6月9日（10：25−12：10）
   - 行列-行列積の並列化（1）
8. 6月16日
   - 行列－行列積の並列化（2）
9. 6月23日
   - LU分解法（1）
   - コンテスト課題発表
10. 6月30日
    - LU分解法（2）
11. 7月7日
    - LU分解法（3）

講義の流れ

1. ブロック化
2. その他の高速化技術
3. OpenMP超入門
4. サンプルプログラムの実行
   （行列-行列積のOpenMP化）
5. 演習課題
6. レポート課題

ブロック化

小さい範囲のデータ再利用

iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

ブロック化によるアクセス局所化

- キャッシュには大きさがあります。
- この大きさを超えると、たとえ連続アクセスしても、キャッシュからデータは追い出されます。
- データが連続してキャッシュから追い出されると、メモリから転送するのと同じとなり、高速なアクセス速度を誇るキャッシュの恩恵がなくなります。
- そこで、高速化のためには、以下が必要です
  1. キャッシュサイズ限界までデータを詰め込む
  2. 詰め込んだキャッシュ上のデータを、何度もアクセスして再利用する

ブロック化によるキャッシュミスヒット削減例

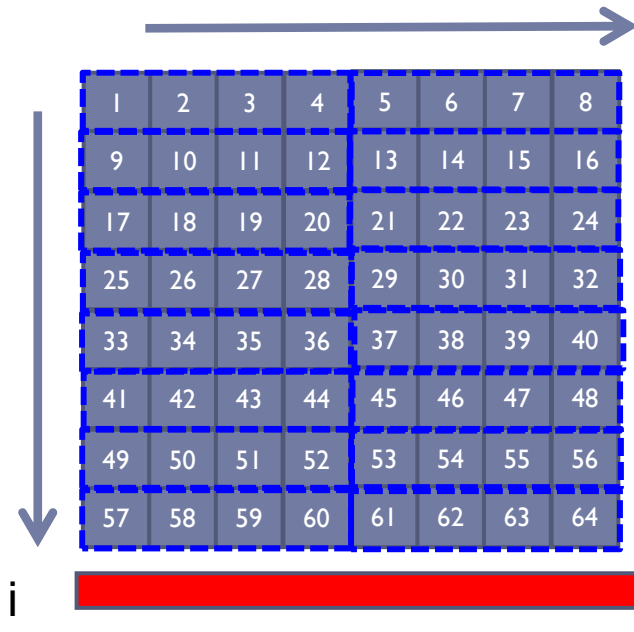
- 行列ー行列積
- 行列サイズ：８×８
  - double A[8][8];
- キャッシュラインは4つ
- １つのキャッシュラインに4つの行列要素が載る
  - キャッシュライン：4×8バイト(double)=32バイト
- 配列の連続アクセスは行方向（C言語）
- キャッシュの追い出しアルゴリズム：
  Least Recently Used (LRU)

iTC 東京大学情報基盤センター Information Technology Center,The University of Tokyo

配列とキャッシュライン構成の関係

- この前提の、＜配列構成＞と＜キャッシュライン＞の関係
  - ここでは、キャッシュライン衝突は考えません

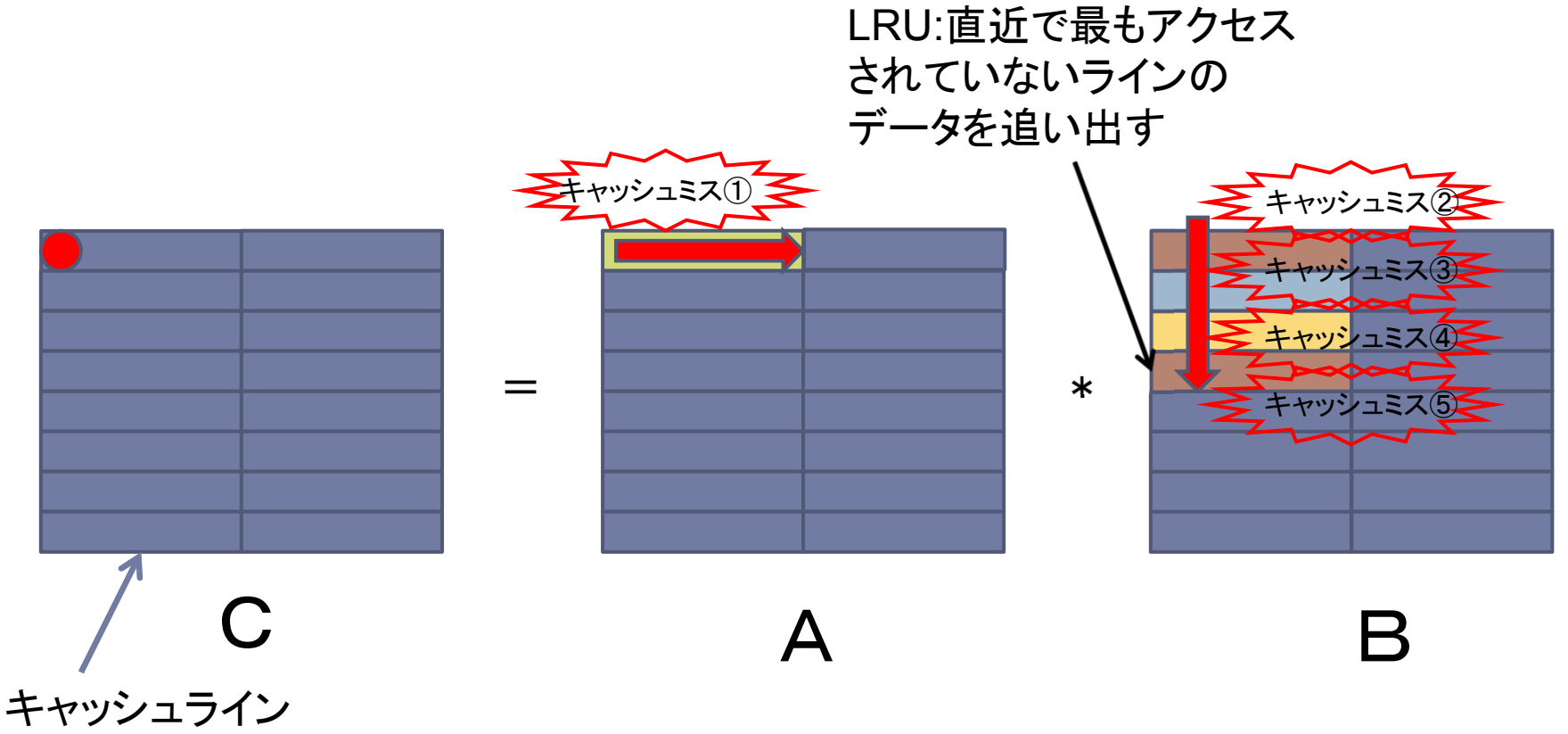
- C言語の場合
  配列A[i][j]、B[i][j]、C[i][j]



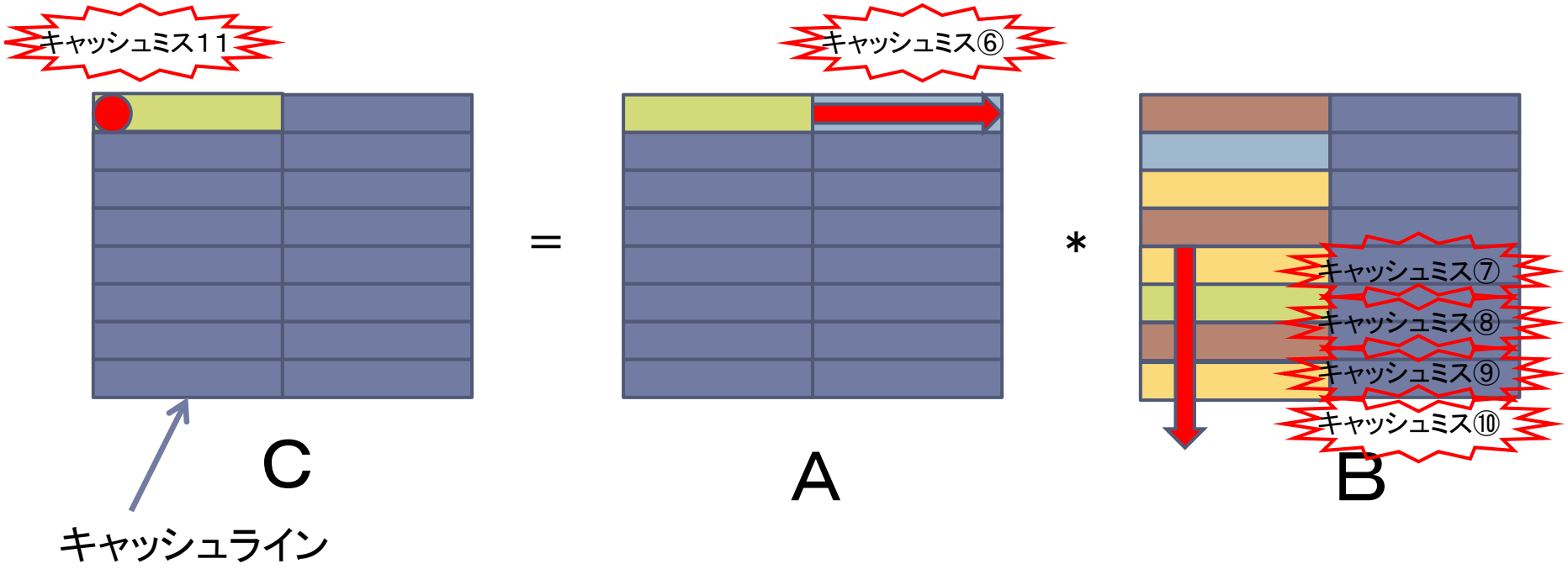
- １×４の配列要素が、キャッシュラインに乗る
- どのキャッシュラインに乗るかは、<配列アクセスパターン> と <置き換えアルゴリズム>依存で決まる

キャッシュラインの構成

行列-行列積の場合（ブロック化しない）

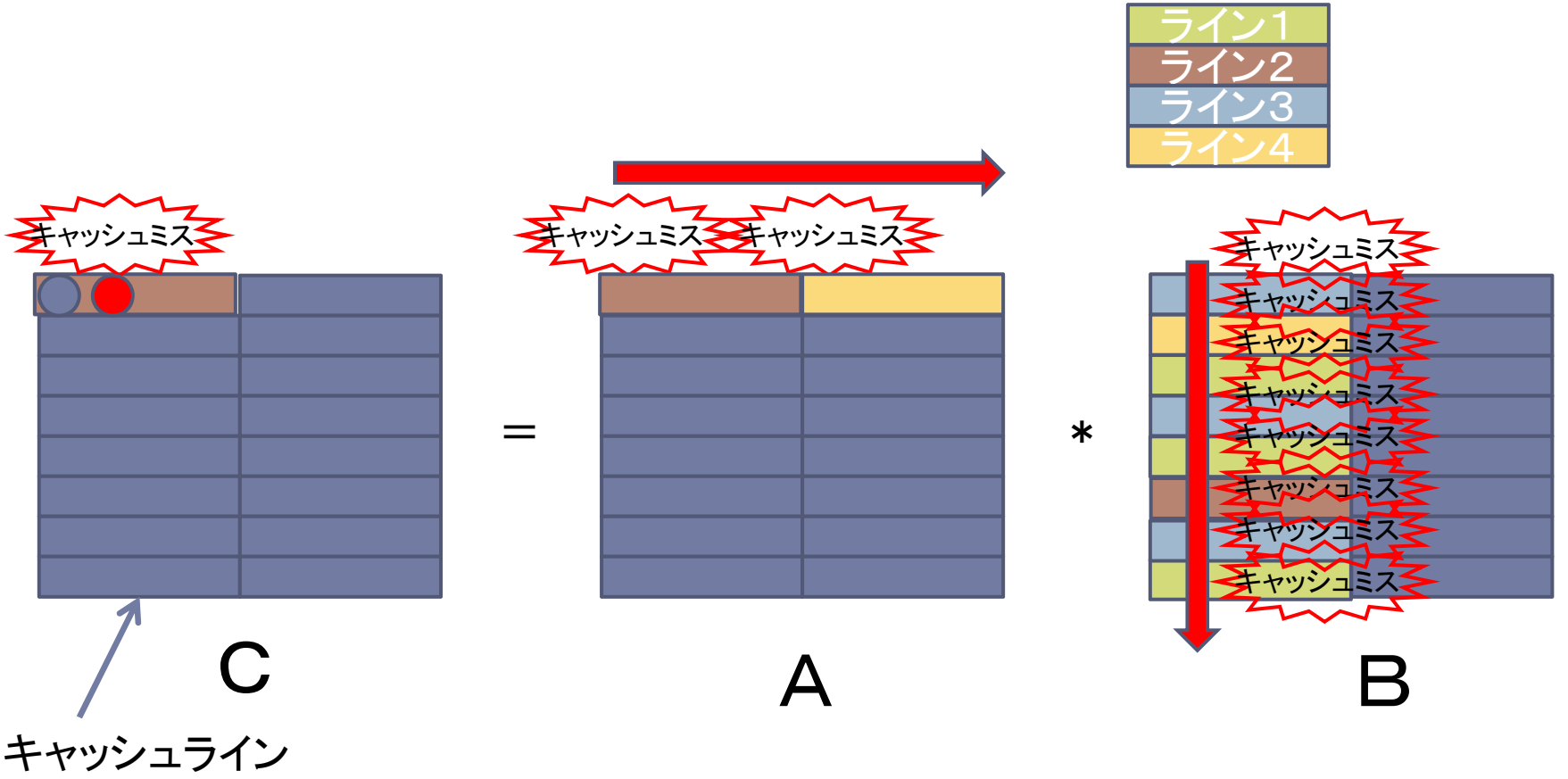


行列-行列積の場合（ブロック化しない）



※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

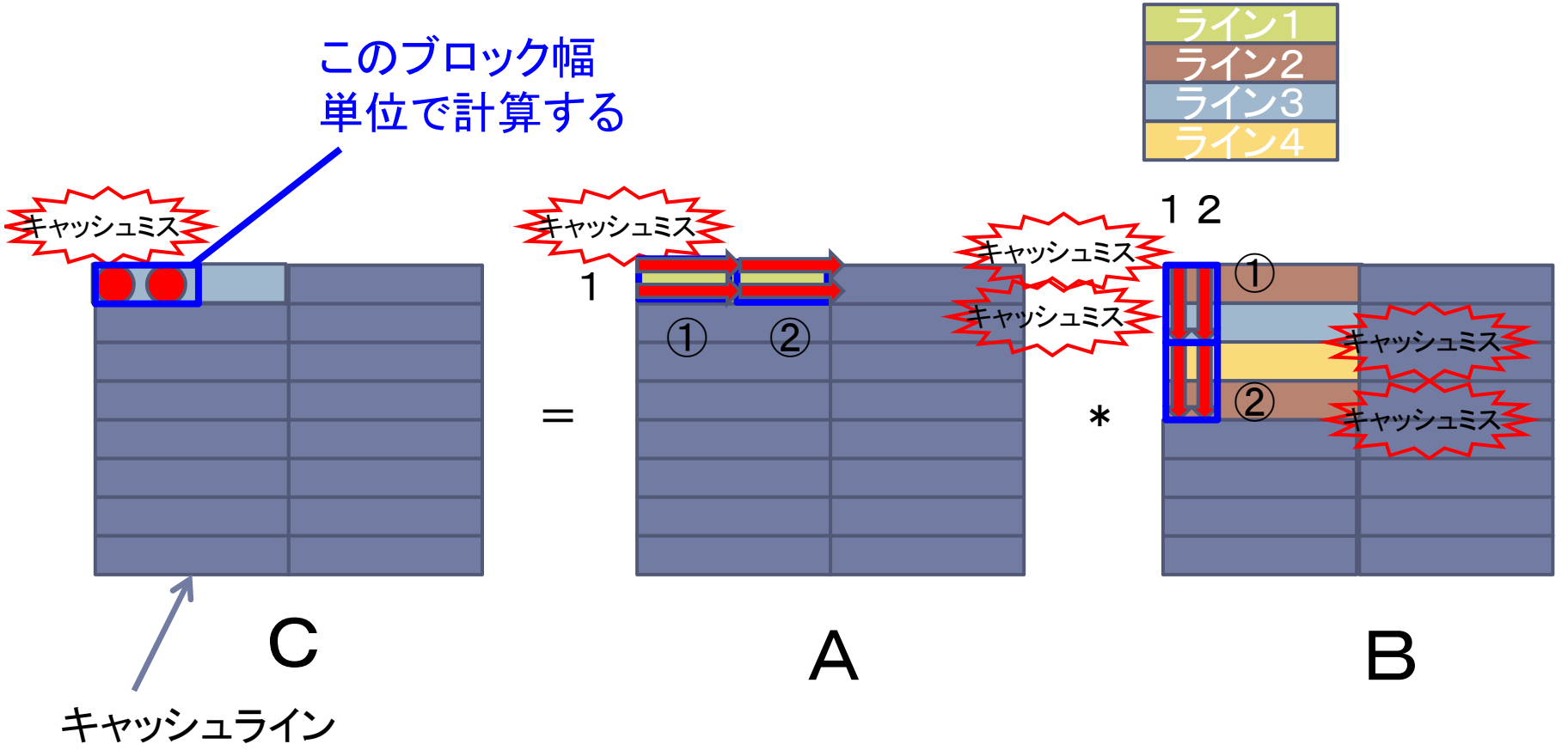
行列-行列積の場合（ブロック化しない）



※2要素計算するのに、
キャッシュミスヒット22回

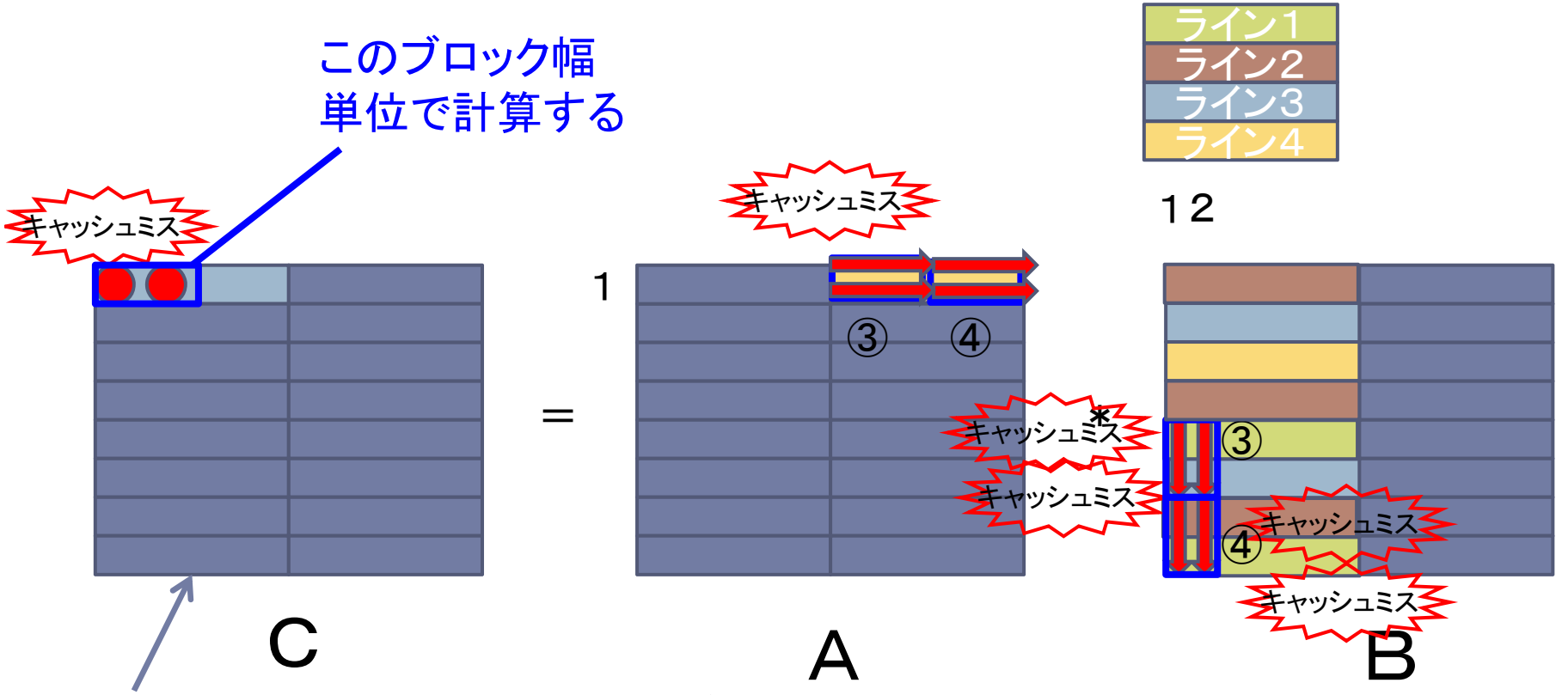
※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

行列-行列積の場合（ブロック化する：2要素）



※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

行列-行列積の場合（ブロック化する：2要素）



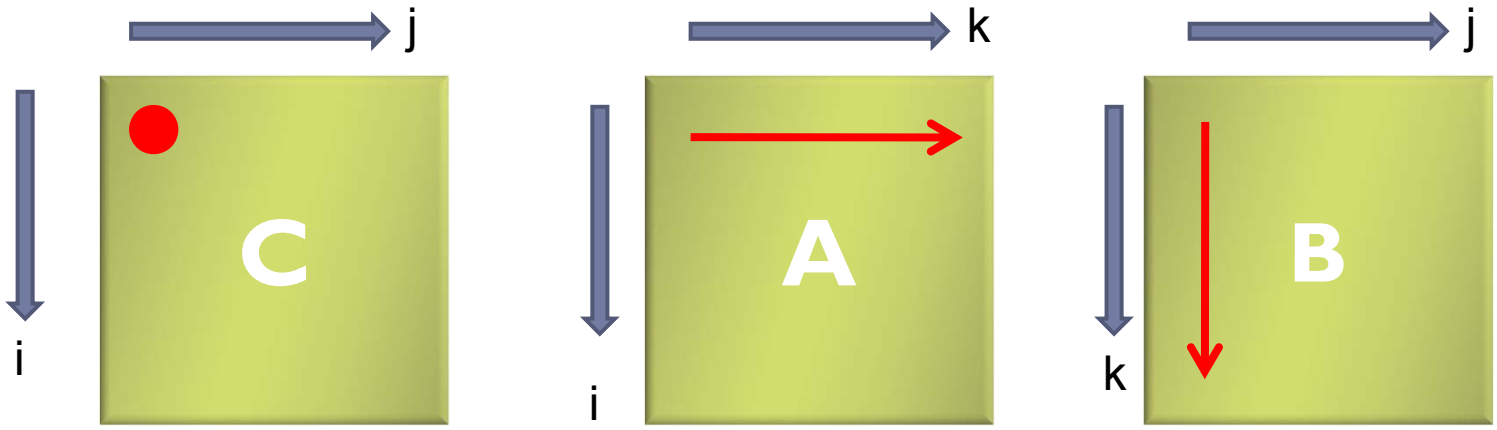
※2要素計算するのに、
キャッシュミスヒット10回

※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

行列積コード（C言語）：キャッシュブロック化なし

- コード例

```
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-行列積のブロック化のコード（C言語）

- nがブロック幅（ibl=16）で割り切れるとき、以下のような6重ループのコードになる

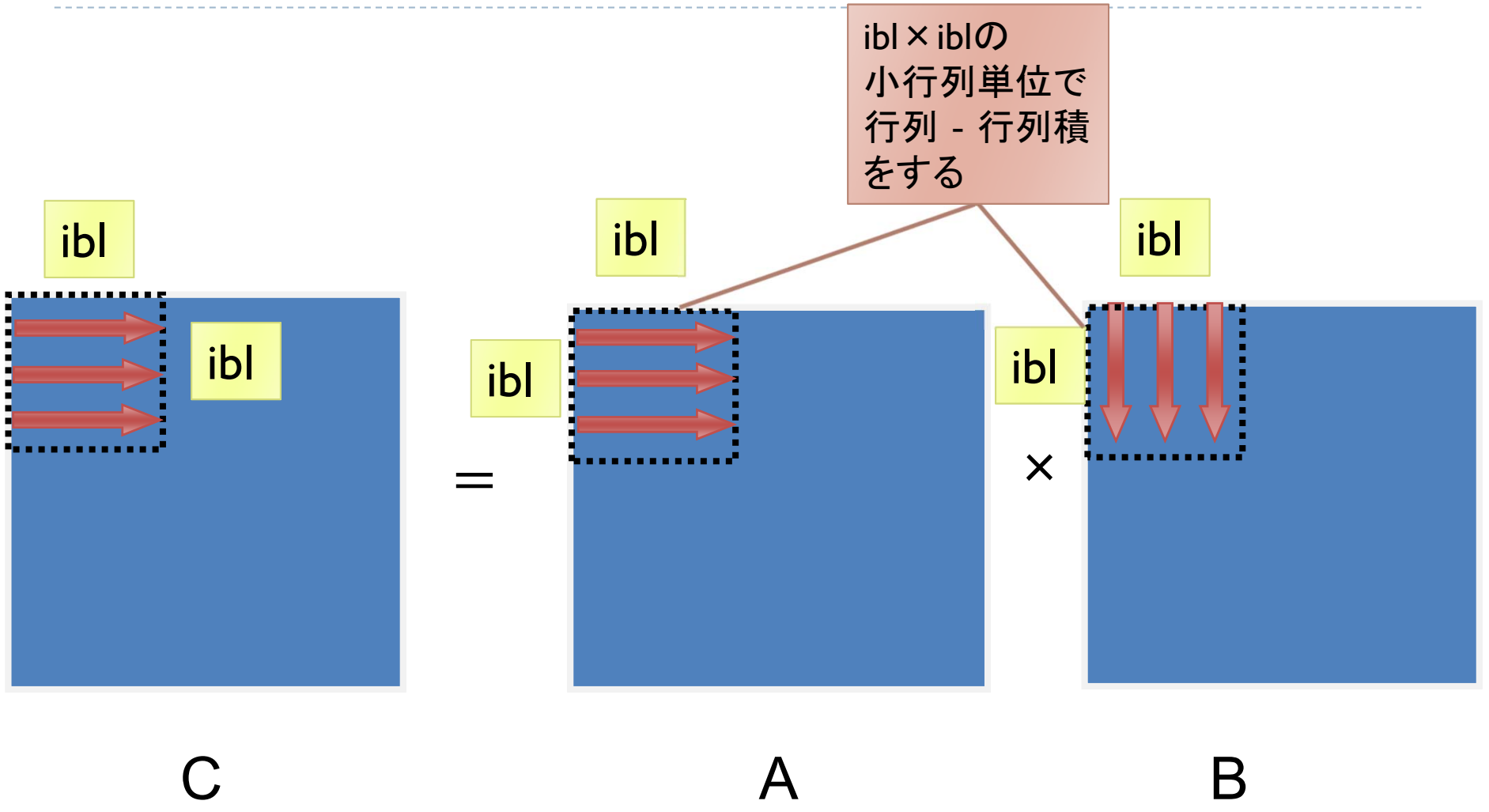
```
ibl = 16;
for ( ib=0; ib<n; ib+=ibl ) {
  for ( jb=0; jb<n; jb+=ibl ) {
    for ( kb=0; kb<n; kb+=ibl ) {
      for ( i=ib; i<ib+ibl; i++ ) {
        for ( j=jb;  j<jb+ibl; j++ ) {
          for ( k=kb; k<kb+ibl; k++ ) {
            C[i][j]  +=  A[i][k]  *  B[k][j];
          } } } } } }
```

行列-行列積のブロック化のコード（Fortran言語）

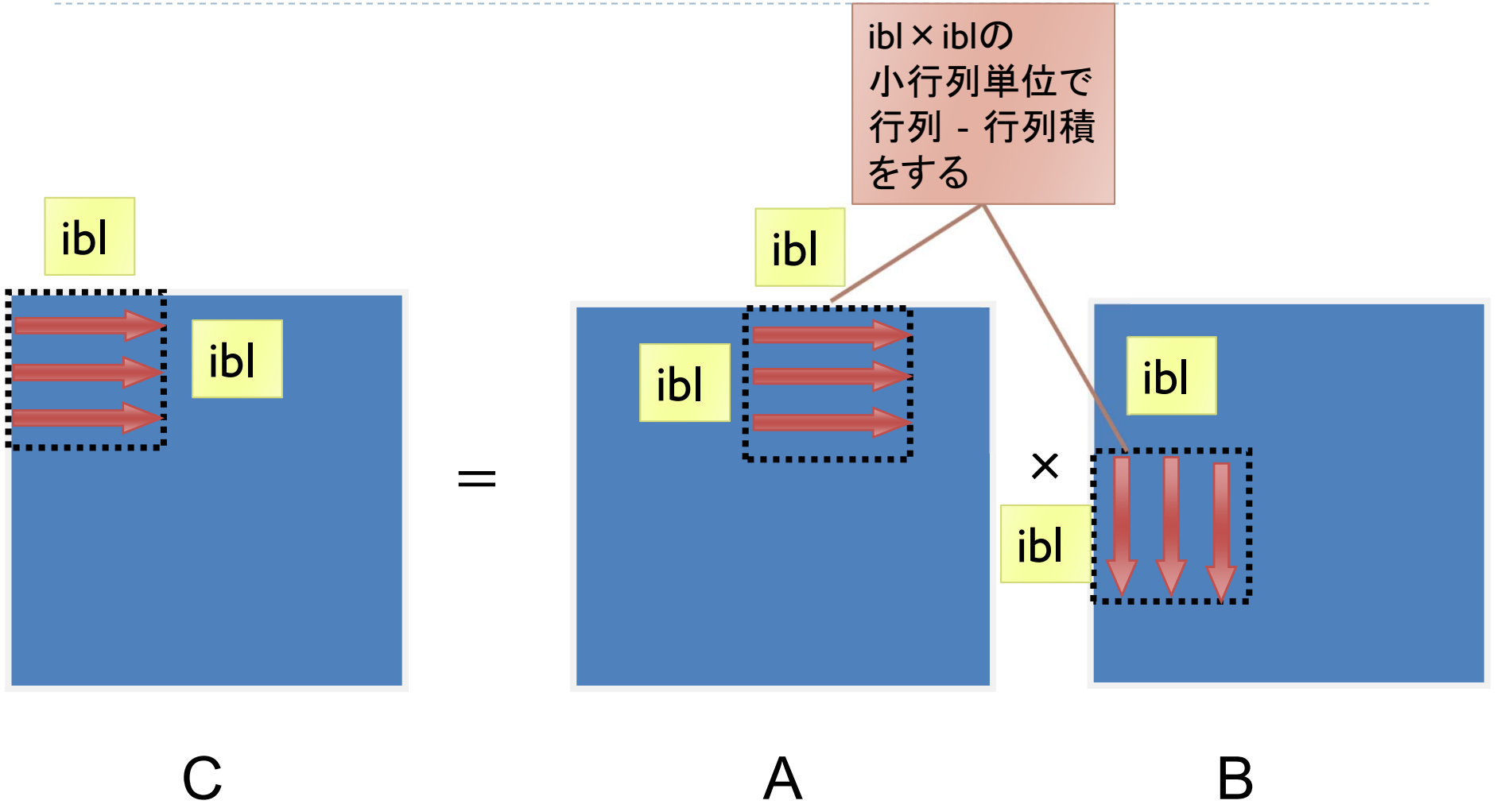
- nがブロック幅（ibl=16）で割り切れるとき、以下のような6重ループのコードになる

```
ibl = 16
do ib=1, n, ibl
   do jb=1, n, ibl
      do kb=1, n, ibl
         do i=ib, ib+ibl-1
            do j=jb, jb+ibl-1
               do k=kb, kb+ibl-1
                  C(i, j) = C(i, j) + A(i, k)  *  B(k, j)
enddo; enddo; enddo; enddo; enddo; enddo;
```

キャッシュブロック化時のデータ・アクセスパターン



キャッシュブロック化時のデータ・アクセスパターン



行列-行列積のブロック化のコードのアンローリング（C言語）

- 行列-行列積の6重ループのコードに加え、さらに各6重ループにアンローリングを施すことができる。
- i-ループ、およびj-ループ2段アンローリングは、以下のようなコードになる。（ブロック幅iblが2で割り切れる場合）

```
ibl = 16;
for (ib=0; ib<n; ib+=ibl) {
  for (jb=0; jb<n; jb+=ibl) {
    for (kb=0; kb<n; kb+=ibl) {
      for (i=ib; i<ib+ibl; i+=2) {
        for (j=jb; j<jb+ibl; j+=2) {
          for (k=kb; k<kb+ibl; k++) {
            C[i  ][j  ] += A[i  ][k] * B[k][j  ];
            C[i+1][j  ] += A[i+1][k] * B[k][j  ];
            C[i  ][j+1] += A[i  ][k] * B[k][j+1];
            C[i+1][j+1] += A[i+1][k] * B[k][j+1];
          } } } } } }
```


iTC 東京大学情報基盤センター Information Technology Center,The University of Tokyo

行列-行列積のブロック化のコードのアンローリング（Fortran言語）

- 行列-行列積の6重ループのコードに加え、
さらに各6重ループにアンローリングを施すことができる。
- i-ループ、およびj-ループ2段アンローリングは、以下のようなコードになる。（ブロック幅iblが2で割り切れる場合）

```
ibl = 16
do ib=1, n, ibl
  do jb=1, n, ibl
    do kb=1, n, ibl
      do i=ib, ib+ibl, 2
        do j=jb, jb+ibl, 2
          do k=kb, kb+ibl
            C(i  , j  ) = C(i  , j  ) + A(i  , k) * B(k, j  )
            C(i+1, j  ) = C(i+1, j  ) + A(i+1, k) * B(k, j  )
            C(i  , j+1) = C(i  , j+1) + A(i  , k) * B(k, j+1)
            C(i+1, j+1) = C(i+1, j+1) + A(i+1, k) * B(k, j+1)
enddo; enddo; enddo; enddo; enddo; enddo;
```

その他の高速化技術

共通部分式の削除（１）

- 以下のプログラムは、冗長な部分がある。

```
d = a + b + c;
f = d + a + b;
```

- コンパイラがやる場合もあるが、以下のように書く方が無難である。

```
temp = a + b;
d = temp + c;
f = d + temp;
```

共通部分式の削除（２）

- 配列のアクセスも、冗長な書き方をしないほうがよい。

```
for (i=0;  i<n;  i++) {
  xold[i] = x[i];
  x[i] = x[i] + y[i];
}
```

- 以下のように書く。

```
for (i=0;  i<n;  i++) {
  dtemp = x[i];
  xold[i] = dtemp;
  x[i] = dtemp + y[i];
}
```

コードの移動

- 割り算は演算時間がかかる。ループ中に書かない。

```
for (i=0;  i<n;  i++) {
   a[i] = a[i] / sqrt(dnorm);
}
```

- 上記の例では、掛け算化して書く。

```
dtemp = 1.0d0 / sqrt(dnorm);
for (i=0;  i<n;  i++) {
   a[i] = a[i] *dtemp;
}
```

ループ中のＩＦ文

- なるべく、ループ中にIF文を書かない。

```
for (i=0;  i<n;  i++) {
  for (j=0;  j<n;  j++) {
    if ( i != j )  A[i][j] = B[i][j];
    else   A[i][j] = 1.0d0;
  } }
```

- 以下のように書く。

```
for (i=0;  i<n;  i++) {
  for (j=0;  j<n;  j++) {
    A[i][j] = B[i][j];
  } }
for (i=0;  i<n;  i++)  A[i][i] = 1.0d0;
```

ソフトウェア・パイプライニングの強化

- 基のコード
  （2段のアンローリング）

定義ー参照の距離が近い
→ソフトウェア的には
何もできない

```
for (i=0; i<n; i+=2) {
   dtmpb0 = b[i];
   dtmpc0 = c[i];
   dtmpa0 = dtmpb0 + dtmpc0;
   a[i] = dtmpa0;
   dtmpb1 = b[i+1];
   dtmpc1 = c[i+1];
   dtmpa1 = dtmpb1 + dtmpc1;
   a[i+1] = dtmpa1;
}
```

- ソフトウェアパイプライニング
  を強化したコード
  （2段のアンローリング）

定義ー参照の距離が遠い
→ソフトウェアパイプライニング
が適用できる機会が増加！

```
for (i=0; i<n; i+=2) {
   dtmpb0 = b[i];
   dtmpb1 = b[i+1];
   dtmpc0 = c[i];
   dtmpc1 = c[i+1];
   dtmpa0 = dtmpb0 + dtmpc0;
   dtmpa1 = dtmpb1 + dtmpc1;
   a[i] = dtmpa0;
   a[i+1] = dtmpa1;
}
```

OpenMP 超入門

指示文による簡単並列化

教科書（演習書）

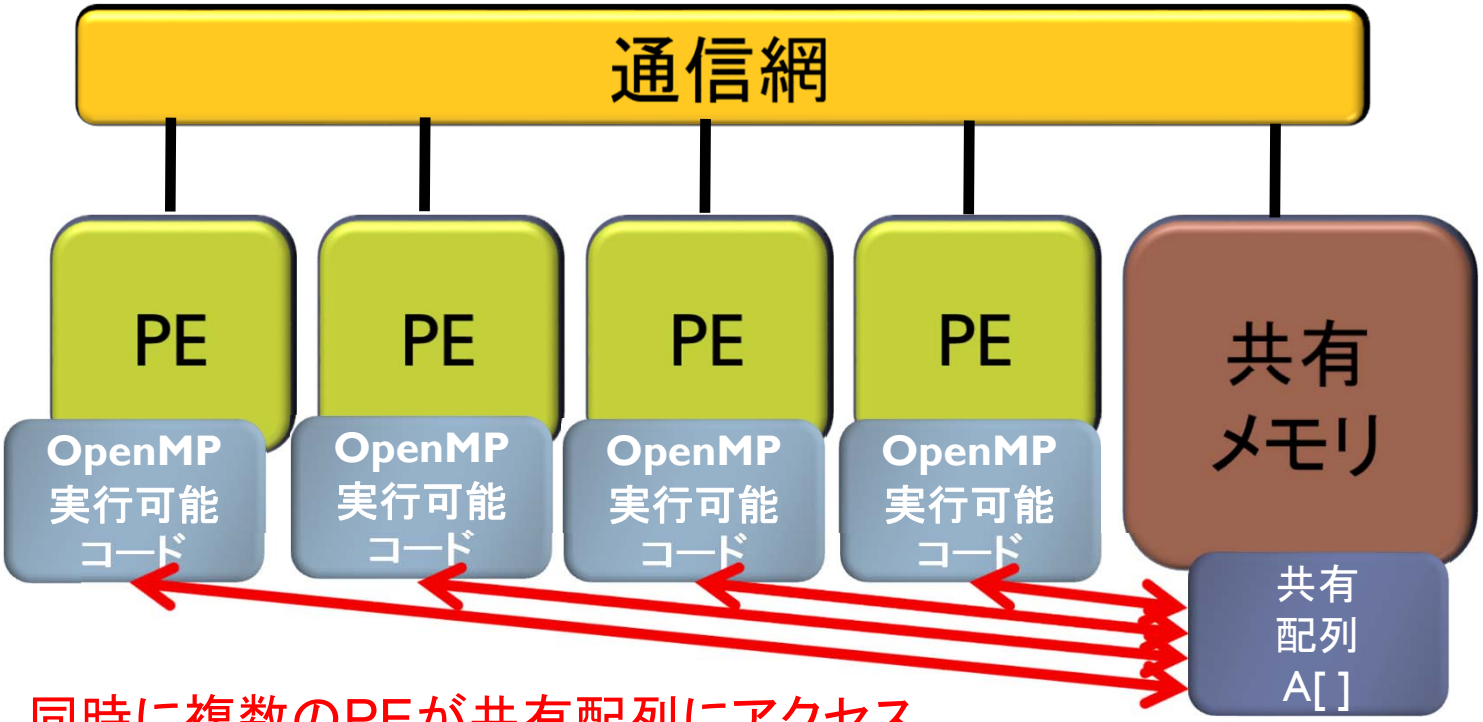
- 「並列プログラミング入門：
  サンプルプログラムで学ぶOpenMPとOpenACC」（仮題）
  - 片桐 孝洋　著
  - 東大出版会、ISBN-10: 4130624563、
    ISBN-13: 978-4130624565、発売日： 2015年5月25日
  - 【本書の特徴】
    - C言語、Fortran90言語で解説
    - C言語、Fortran90言語の複数のサンプルプログラムが入手可能
      （ダウンロード形式）
    - 本講義の内容を全てカバー
    - Windows PC演習可能(Cygwin利用)。スパコンでも演習可能。
    - 内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

OpenMPの概要

iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

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確保するには、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
    - T2Kオープンスパコン（AMD Quad Core Opteron(Barcelona) 、4ソケット）、FX10スーパコンピュータシステム（Sparc64 IVfx）
  - 32～128スレッド／ノード
    - 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）
- 近い将来（２～３年後）には、100スレッドを超えたOpenMPによる実行形態が普及すると予想
  - 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Fotran90コンパイラ
    frt –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
      frt –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
- スレッド数を、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

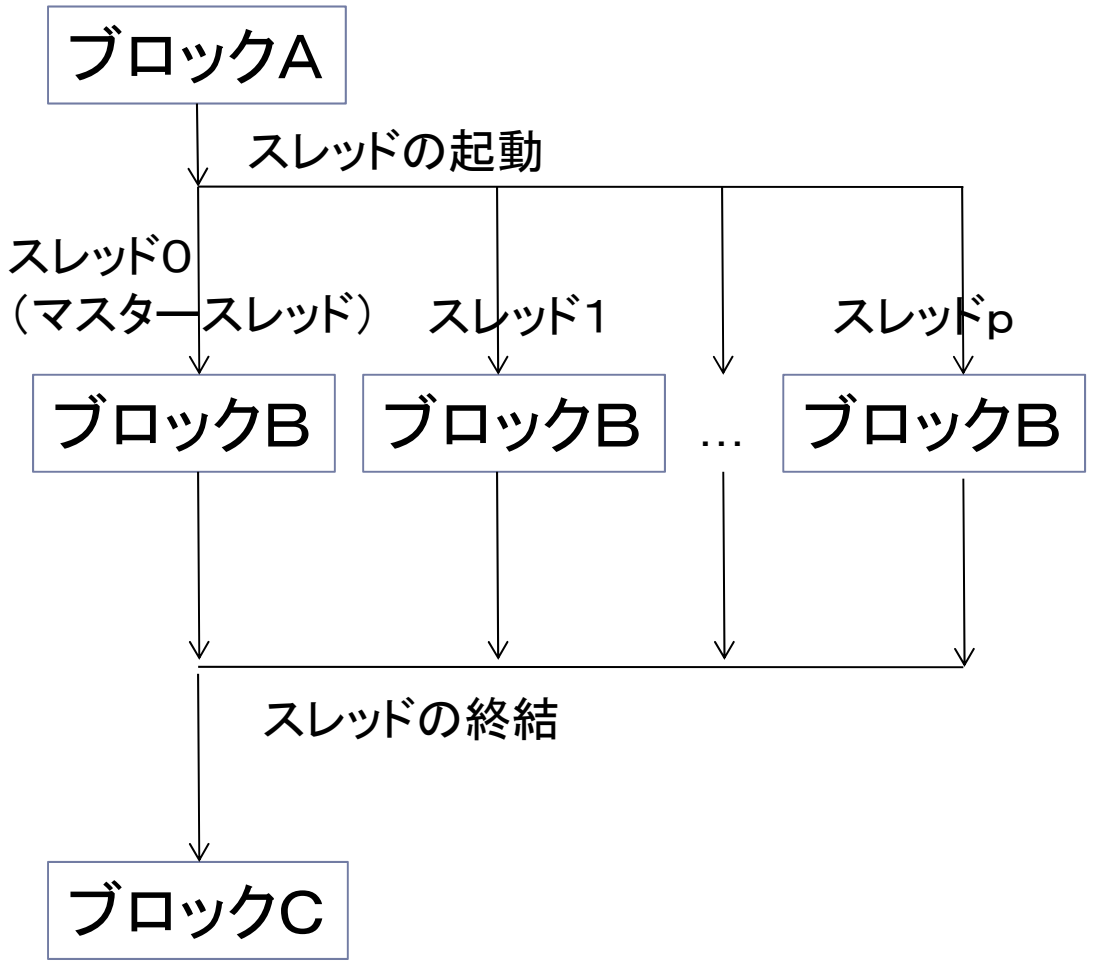
OpenMPの実行モデル

iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

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
```

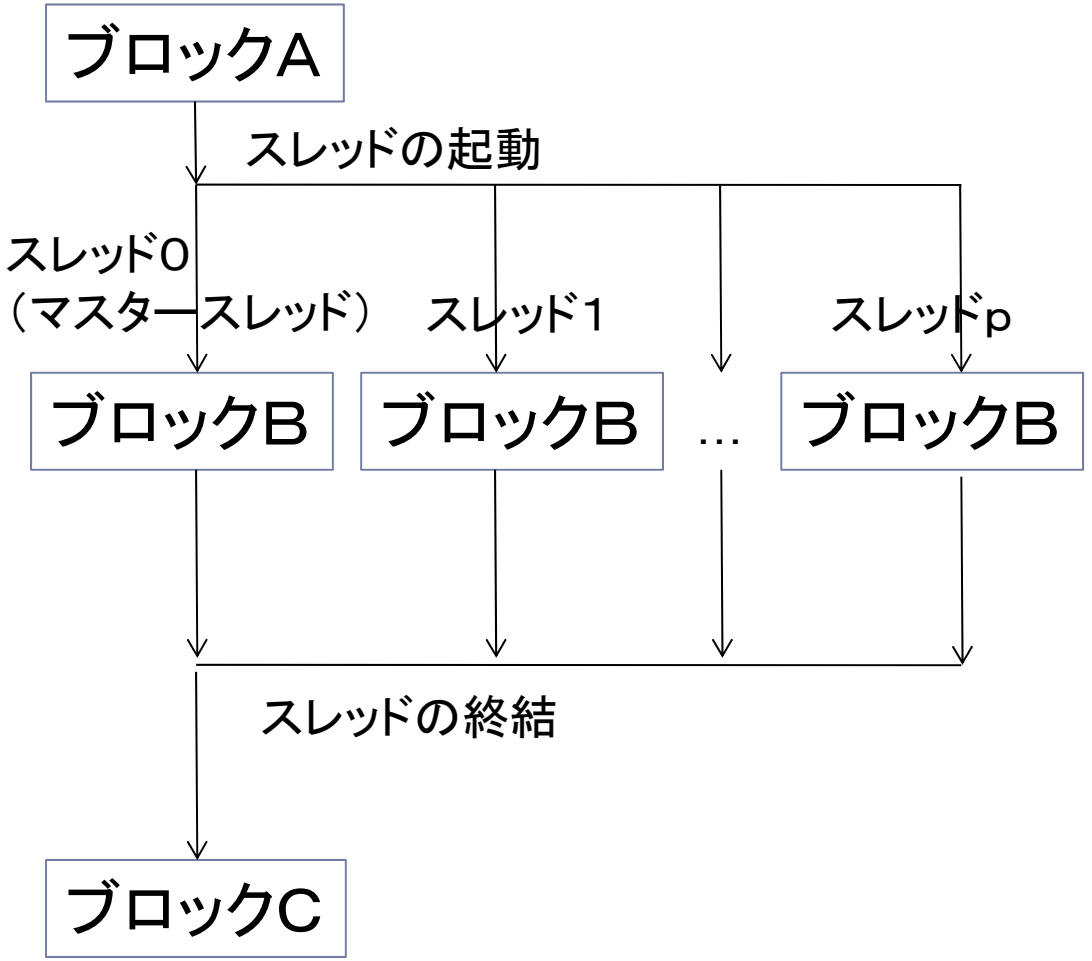


※スレッド数pは、
環境変数
OMP_NUM_THREADS
で指定する。

OpenMPの実行モデル（Fortran言語）

OpenMP指示文

```
    ブロックA
!$omp  parallel
    ブロックB
!$omp  end parallel
    ブロックC
```



※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

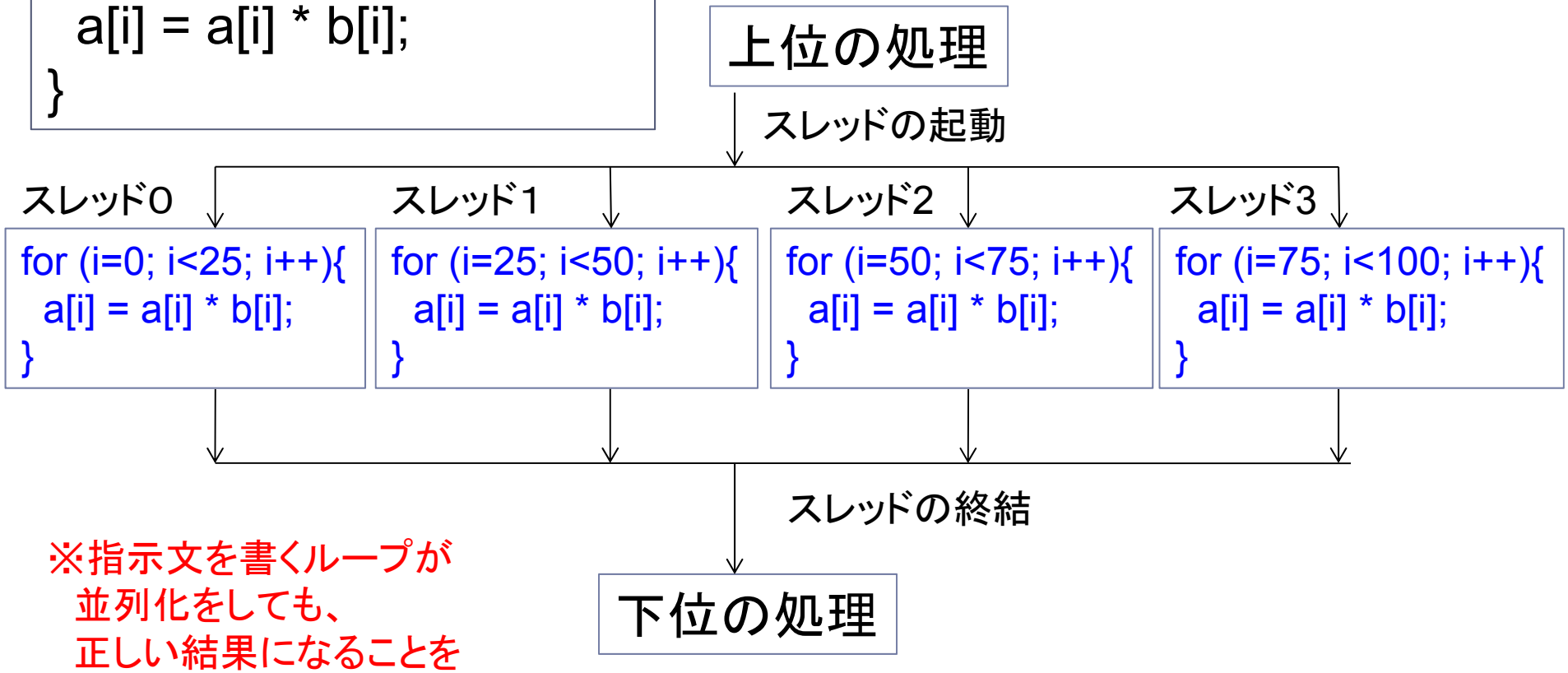
代表的な指示文

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
```



※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

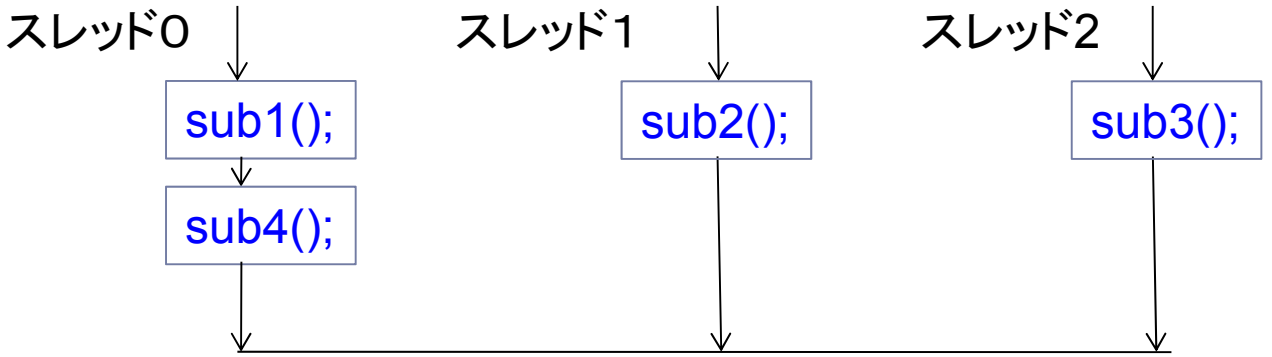
Sections構文

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

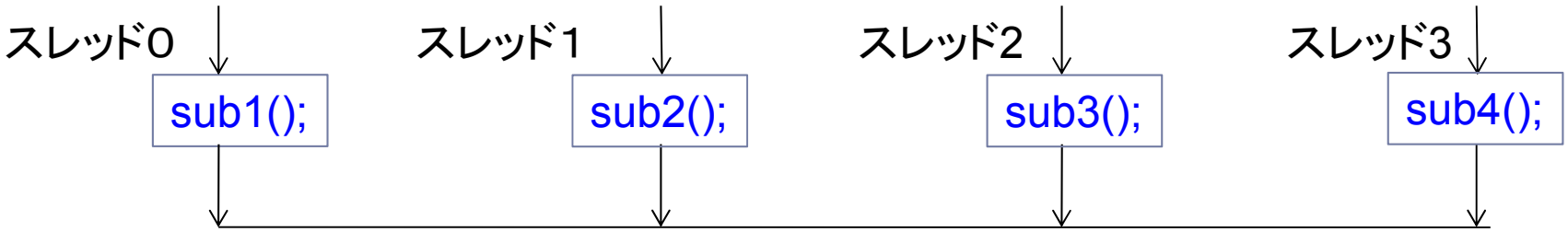
※Fortran言語の場合は

```
!$omp sections
～
!$omp end sections
```

●スレッド数が3の場合



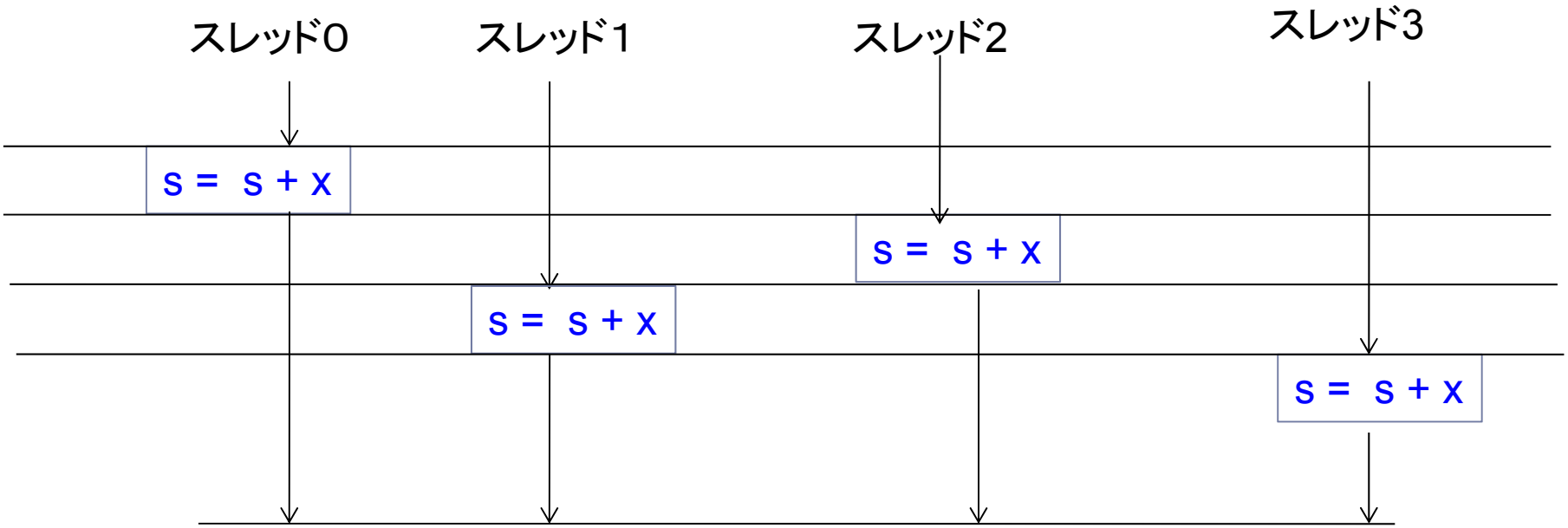
●スレッド数が4の場合



Critical補助指示文

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

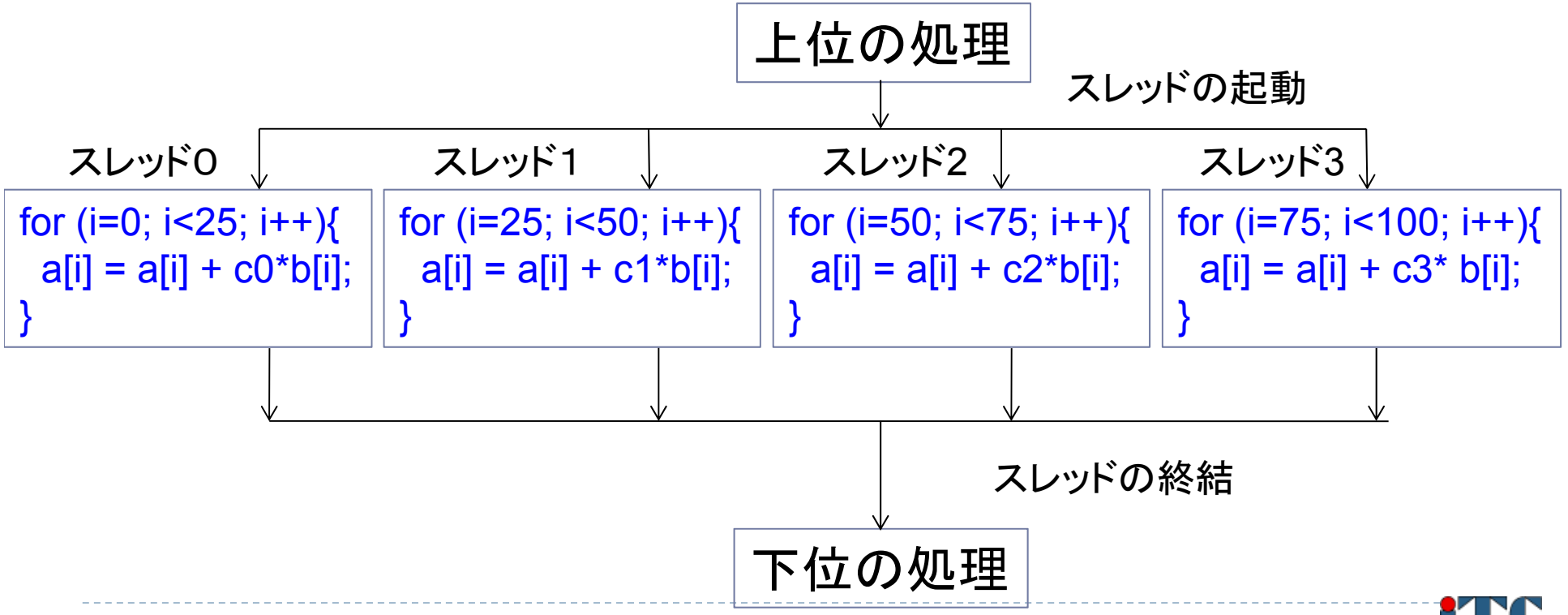
```
#pragma omp critical
{
  s =  s +  x;
}
```



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];
}
```

※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



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）

```
!$omp parallel do private ( i )
do j=0, p-1
  do i=istart( j ), iend( j )
     ddot_t( j ) = ddot_t( j ) + a(i) * b(i)
  enddo
enddo
!$omp end parallel do
ddot = 0.0d0
do j=0, p-1
     ddot = ddot + ddot_t( j )
enddo
```

スレッド数分のループを作成：最大pスレッド利用

各スレッドでアクセスするインデックス範囲を事前に設定

各スレッドで用いる、ローカルなddot用の配列ddot_t()を確保し、0に初期化しておく

逐次で足しこみ

その他、よく使うOpenMPの関数

iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

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

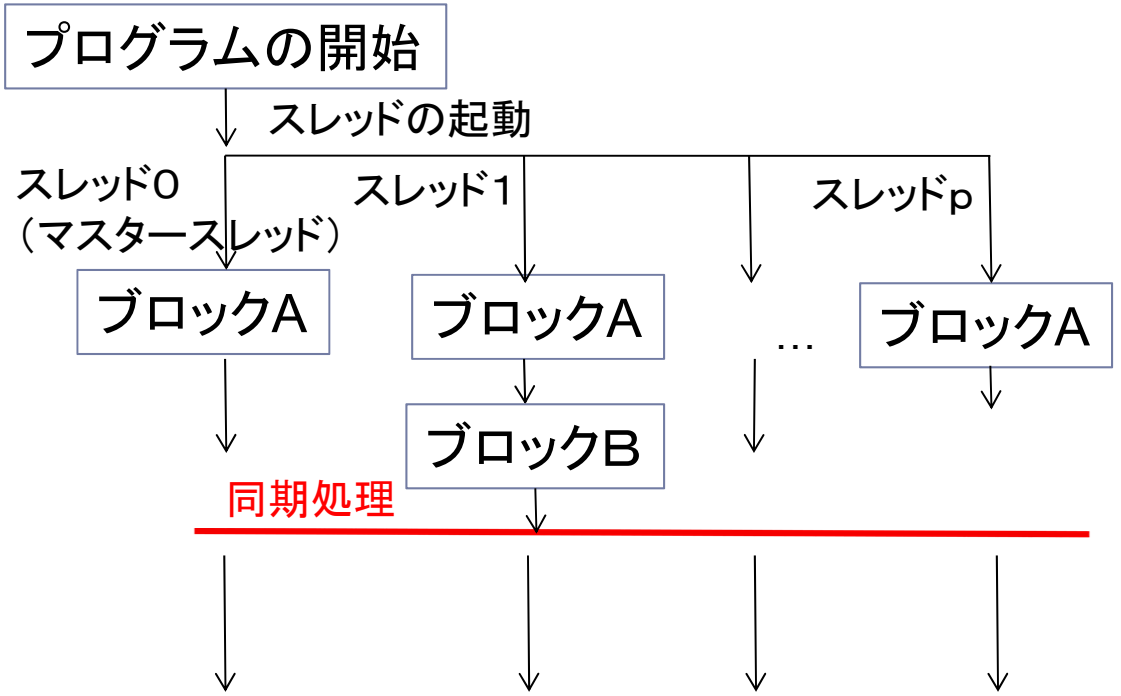
iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

Single構文

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

- 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

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



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)

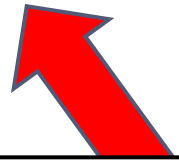
省略すると、
全ての変数が対象

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_threds();
  myid = omp_get_thread_num();
  istart = myid * (n/nthreads);
  iend = (myid+1)*(n/nthreads);
  if (myid == (nthreads-1)) {
    nend = n;
  }
  kernel();
}
```



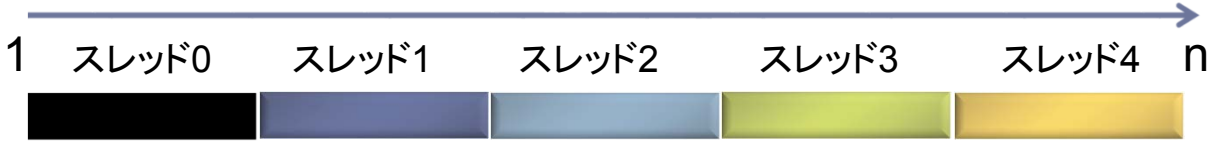
スレッド毎に異なる値を持つ大域変数を、parallel構文中で定義する

スケジューリング

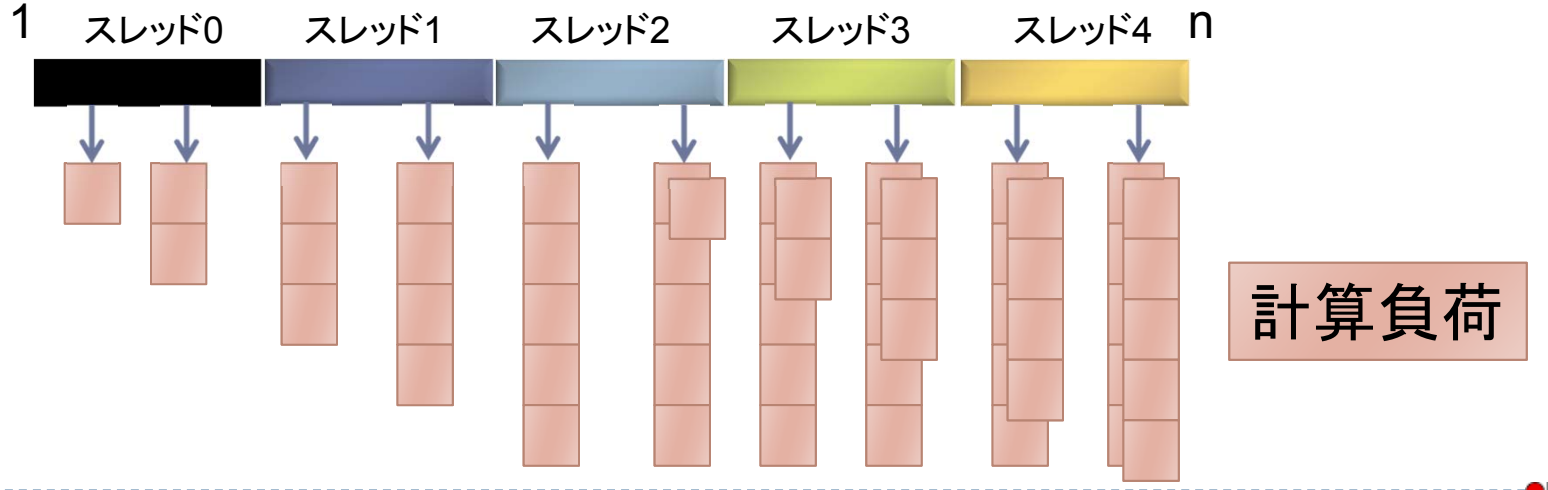
iTC
東京大学情報基盤センター
Information Technology Center,The University of Tokyo

スケジューリングとは（その１）

- Parallel do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。

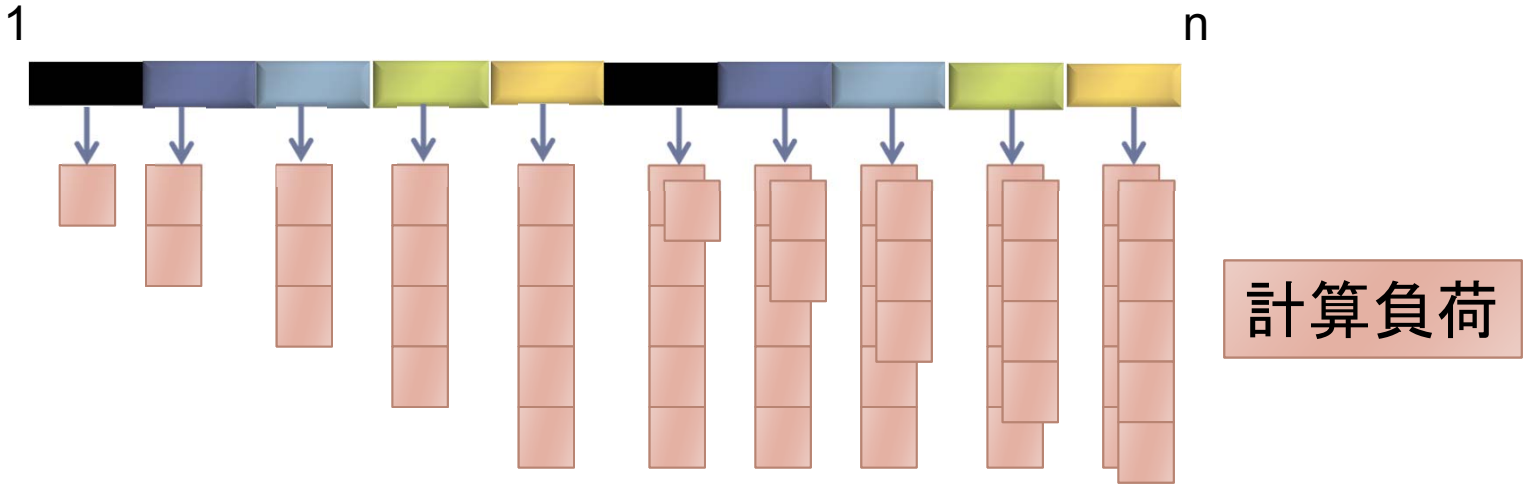


- このとき、各スレッドで担当したループに対する計算負荷が均等でないと、スレッド実行時の台数効果が悪くなる



スケジューリングとは（その２）

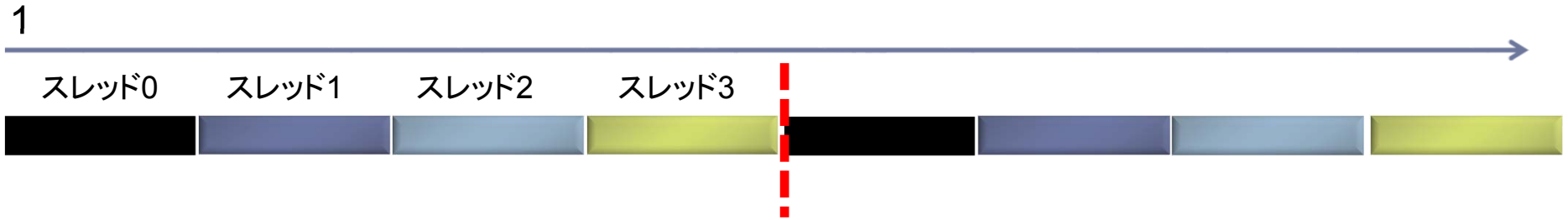
- 負荷分散を改善するには、割り当て間隔を短くし、かつ、循環するように割り当てればよい。



- 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。

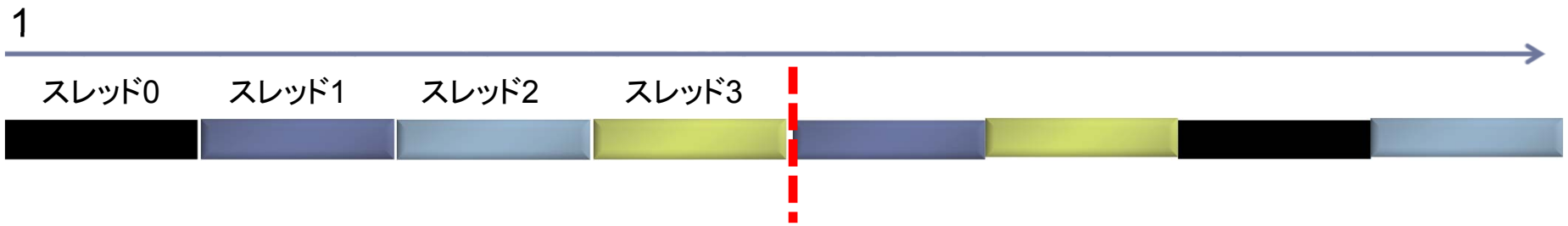
ループスケジューリングの補助指定文（その１）

- 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。



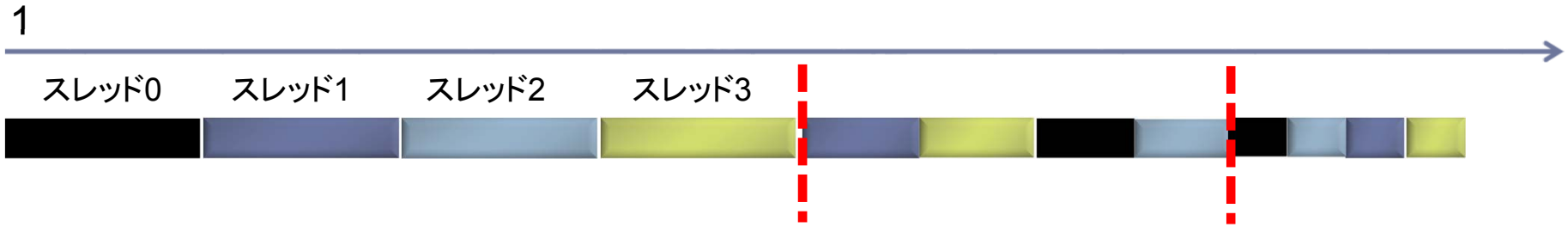
ループスケジューリングの補助指定文（その２）

- 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、処理待ち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と負荷バランスが悪くなる半面、処理待ちのオーバヘッドが小さくなる。
  - 上記の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）

iTC 東京大学情報基盤センター
Information Technology Center,The University of Tokyo

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、Fotran90言語のcommon変数、module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
- parallel構文で関数呼び出ししている場合、その関数内でローカルに宣言している変数も、共有変数になる
  - そのままでは、並列処理で正常動作しない
  - これを防ぐには、以下のコードの変更が必要
    - 上記のローカル変数を引数にした関数呼び出しを作る
    - 上記のローカル変数を大域変数にして、Threadprivate宣言する

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

```
!$omp parallel
!$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
```

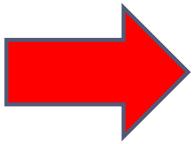
Parallel構文の対象が1ループなら parallel do で指定

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



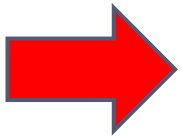
```
!$omp parallel
do i=1, n
!$omp do
  do j=1, n
    <並列化できる式>
  enddo
!$omp end do
enddo
!$omp end parallel
```

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、データー貫性の保証はしない
    - データー貫性の保証には、critical補助指定文などの指定が必要

● 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
- 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

サンプルプログラムの実行（行列-行列積OpenMP）

iTC 東京大学情報基盤センター
Information Technology Center,The University of Tokyo

行列-行列積のサンプルプログラム(OpenMP版)の注意点

- C言語版およびFortran言語版のファイル名
  Mat-Mat-openmp-fx.tar
- ジョブスクリプトファイルmat-mat-openmp.bash
  中のキュー名を　lecture から
  lecture7　(工学部共通科目)
  に変更し、pjsub　してください。
  - lecture : 実習時間外のキュー
  - lecture7: 実習時間内のキュー

行列-行列積のサンプルプログラムの実行

- 以下のコマンドを実行する

```
$ cp /home/z30082/Mat-Mat-openmp-fx.tar ./
$ tar xvf Mat-Mat-openmp-fx.tar
$ cd Mat-Mat-openmp
```

- 以下のどちらかを実行

```
$ cd C :C言語を使う人
$ cd F :Fortran言語を使う人
```

- 以下は共通

```
$ make
$ pjsub mat-mat-openmp.bash
```

- 実行が終了したら、以下を実行する

```
$ cat mat-mat-openmp.bash.oXXXXX
```

行列-行列積のサンプルプログラムの実行

- 以下のような結果が見えれば成功(C言語)
  (ただし、OpenMP化されていません)

N = 1000

Mat-Mat time = 0.181551 [sec.]

11016.210378 [MFLOPS]

OK!

行列-行列積のサンプルプログラムの実行

- 以下のような結果が見えれば成功（Fortran言語）
  (ただし、OpenMP化されていません)

```
N  =  1000
Mat-Mat time[sec.] =  0.18022635299712 42
MFLOPS =  11097.15622433085
OK!
```

演習課題

1. MyMatMat関数をブロック化により高速化してください。
   - 前回のサンプルプログラム(mat-mat-noopt-fx.tar)を使ってください。OpenMP版ではありません！
   - コンパイラの最適化レベルを0にしてあります
   - コンパイラによる最適化を行わないでください。
     手によるアンローリングの効果がなくなります。
2. MyMatMat関数を、OpenMP化して高速化してください。
   - 本日のサンプルプログラム(mat-mat-openmp-fx.tar)を使ってください。
   - さらにアンローリング、ブロック化などのチューニングをしてください。

行列-行列積のOpenMP化の解答

東京大学情報基盤センター
Information Technology Center,The University of Tokyo

行列-行列積のコードのOpenMP化の解答（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];
        }
      }
    }
```


東京大学情報基盤センター
Information Technology Center,The University of Tokyo

行列-行列積のコードのOpenMP化の解答（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
```

レポート課題

1. [L15]　ブロック化を行った行列-行列積のコードに対し、アンローリングを各ループについて施し性能を調査せよ。行列の大きさ(N)を変化させ、各Nに対して、適切なアンローリング段数を調査せよ。
2. [L15]　OpenMP化した行列-行列積のコードに対し、ブロック化とアンローリングを施し性能を調査せよ。

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。

レポート課題

3. [L20]　セクタキャッシュを制御するコマンドにより性能チューニングを行え。
（前回の、行列 - 行列積コードに対し）

参考文献

- 寒川光、「RISC超高速化プログラミング技法」、共立出版、ISBN4－320－02750－7、3,500円
- Kevin Dowd著、久良知真子訳、「ハイ・パフォーマンス・コンピューティング：RISCワークステーションで最高の性能を引き出すための方法」、インターナショナル・トムソン・パブリッシング・ジャパン、ISBN4－900718－03－3、4,400円

来週へつづく

行列 - ベクトル積

東京大学情報基盤センター
Information Technology Center,The University of Tokyo