「情報システム学特別講義３」
行列-行列積の並列化

東京大学情報基盤センター准教授　片桐孝洋

２０１４年６月２４日（火）１４：４０－１６：１０

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

講義日程
（情報システム学特別講義３）

レポートおよびコンテスト課題
（締切：
2014年8月11日（月）24時　厳守

1. ~~4月8日：ガイダンス~~
2. ~~4月15日~~
   - ~~プログラム高速化の基礎（その1）~~
3. ~~4月22日~~
   - ~~プログラム高速化の基礎（その2）~~
4. ~~5月13日~~
   - ~~MPIの基礎~~
5. ~~5月20日~~
   - ~~OpenMPの基礎~~
6. ~~5月27日~~
   - ~~Hybrid並列化技法（MPIとOpenMPの応用編）~~
7. ~~6月3日~~
   - ~~プログラム高速化の応用~~
8. ~~6月10日~~
   - ~~行列ーベクトル積の並列化~~
9. ~~6月17日~~
   - ~~べき乗法の並列化~~
10. 6月24日
    - 行列−行列積の並列化
11. 7月8日
    - LU分解の並列化
12. 7月15日
    - 非同期通信
    - 疎行列反復解法の並列化
13. 7月22日
    - ソフトウェア自動チューニング
14. 8月5日（補講日）
    - エクサフロップスコンピューティングに向けて

行列 - 行列積の演習の流れ

- 演習課題（１）
  - 本日行う
  - 簡単なもの（30分程度で並列化）
  - 通信関数が一切不要
- 演習課題（２）
  - 来週行う
  - ちょっと難しい（１時間以上で並列化）
  - １対１通信関数が必要

行列-行列積とは

実装により性能向上が見込める基本演算

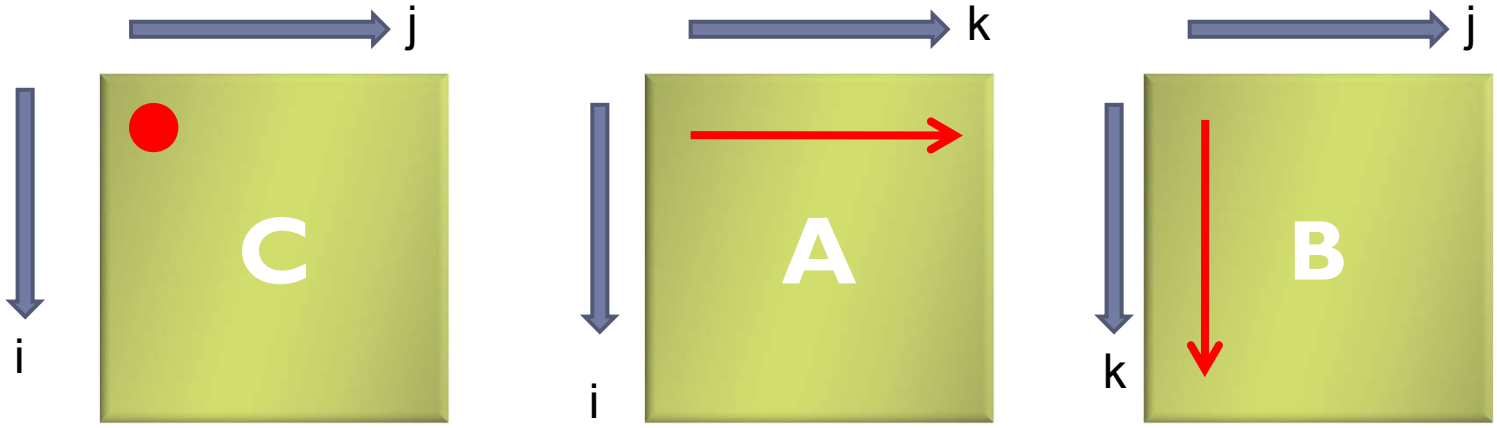
1.5　行列の積

- 行列積　C＝A・B は、コンパイラや計算機のベンチマークに使われることが多い
  - 理由1：実装方式の違いで性能に大きな差がでる
  - 理由2：手ごろな問題である（プログラムし易い）
  - 理由3：科学技術計算の特徴がよく出ている
    1. 非常に長い＜連続アクセス＞がある
    2. キャッシュに乗り切らない＜大規模なデータ＞に対する演算である
    3. メモリバンド幅を食う演算（メモリ・インテンシブ）な処理である

行列積コード例（C言語）

- 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];
```



1.5　行列の積

- 行列積　$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ (i, j = 1, 2, \ldots, n)$

  の実装法は、次の二通りが知られている：

1. ループ交換法
   - 連続アクセスの方向を変える目的で、行列-行列積を実現する3重ループの順番を交換する
2. ブロック化（タイリング）法
   - キャッシュにあるデータを再利用する目的で、あるまとまった行列の部分データを、何度もアクセスするように実装する

1.5　行列の積（C言語）

- ループ交換法
  - 行列積のコードは、以下のような3重ループになる

```
for(i=0; i<n; i++) {
  for(j=0; j<n; j++) {
    for(k=0; k<n; k++) {
      c[ i ][ j ] = c[ i ][ j ] + a[ i ][ k ] * b[ k][ j ];
    }
  }
}
```

  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

1.5　行列の積（Fortran言語）

- ループ交換法
  - 行列積のコードは、以下のような3重ループになる

```
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
```

  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

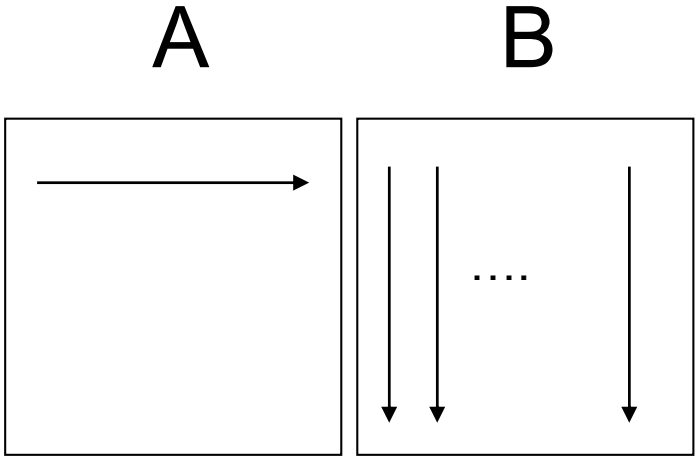
1.5　行列の積

- 行列データへのアクセスパターンから、
  以下の3種類に分類できる
  1. 内積形式 (inner-product form)
     最内ループのアクセスパタンが
     ＜ベクトルの内積＞と同等
  2. 外積形式 (outer-product form)
     最内ループのアクセスパタンが
     ＜ベクトルの外積＞と同等
  3. 中間積形式 (middle-product form)
     内積と外積の中間

1.5　行列の積（C言語）

- 内積形式 (inner-product form)
  - ijk, jikループによる実現

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    dc = 0.0;
    for (k=0; k<n; k++){
      dc = dc + A[ i ][ k ] * B[ k ][ j ];
    }
    C[ i ][ j ]= dc;
  }
}
```



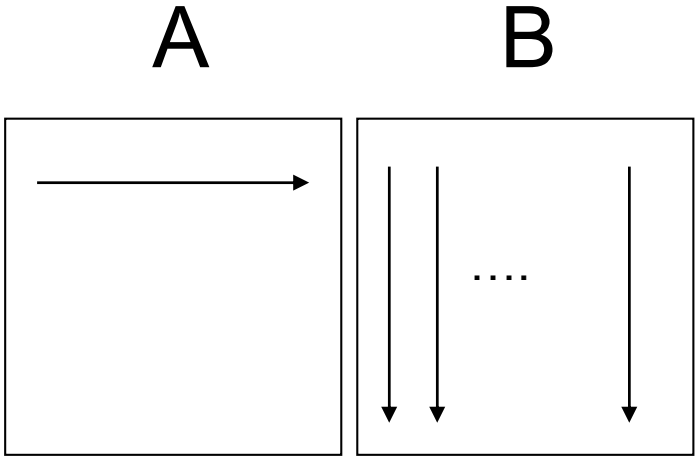
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法：
A, Bどちらか一方を転置しておく

※以降、最外のループからの変数の順番で実装法を呼ぶ。たとえば上記のコードは＜ijkループ＞。

1.5　行列の積（Fortran言語）

- 内積形式 (inner-product form)
  - ijk, jikループによる実現

```
do i=1, n
  do j=1, n
    dc = 0.0d0
    do k=1, n
      dc = dc + A( i , k ) * B( k , j )
    enddo
    C( i , j ) = dc
  enddo
enddo
```



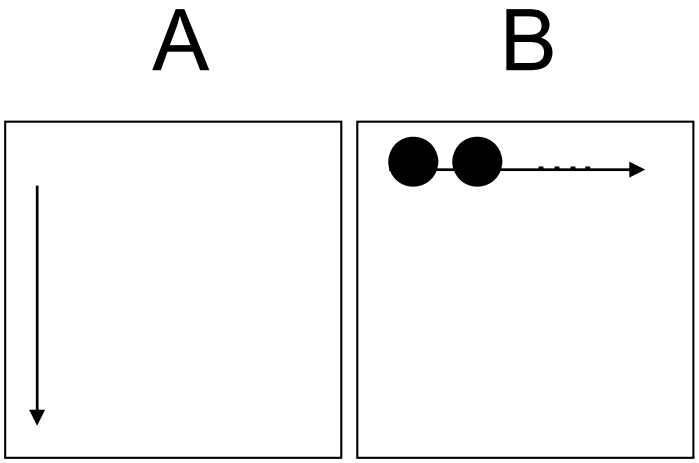
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法：
A, Bどちらか一方を転置しておく

※以降、最外のループからの変数の順番で実装法を呼ぶ。たとえば上記のコードは＜ijkループ＞。

1.5　行列の積（C言語）

- 外積形式 (outer-product form)
  - kij, kjiループによる実現

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    C[ i ][ j ] = 0.0;
  }
}
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    db = B[ k ][ j ];
    for (i=0; i<n; i++) {
      C[ i ][ j ]= C[ i ][ j ]+ A[ i ][ k ]* db;
    }
  }
}
```

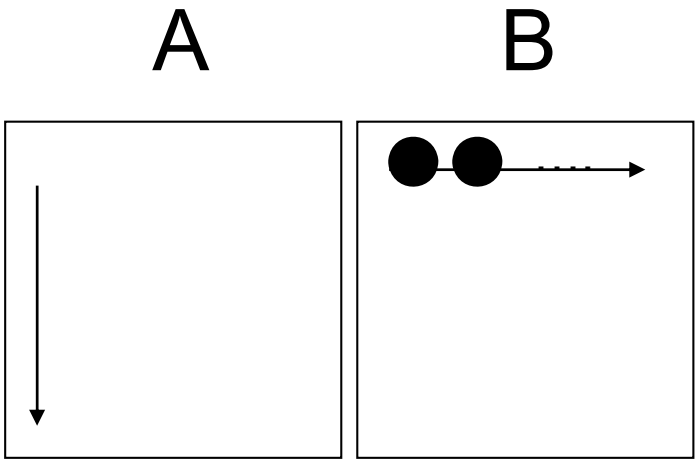


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

1.5　行列の積（Fortran言語）

- 外積形式 (outer-product form)
  - kij, kjiループによる実現

```
do i=1, n
  do j=1, n
    C( i , j ) = 0.0d0
  enddo
enddo
do k=1, n
  do j=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j )+ A( i , k ) * db
    enddo
  enddo
enddo
```

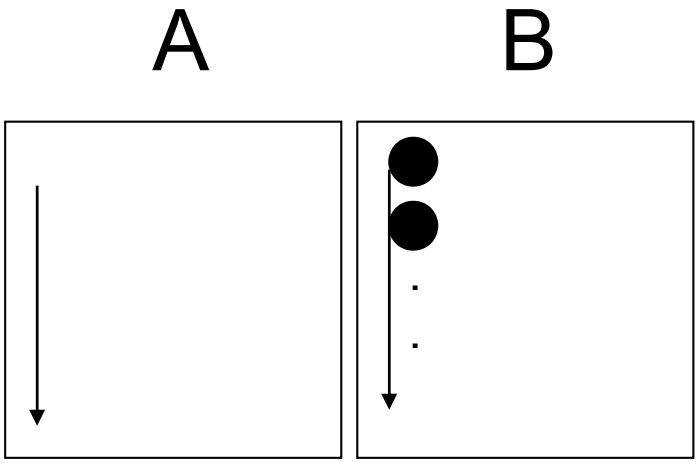


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

1.5　行列の積（C言語）

- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現

```
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    C[ i ][ j ] = 0.0;
  }
  for (k=0; k<n; k++) {
    db = B[ k ][ j ];
    for (i=0; i<n; i++) {
      C[ i ][ j ] = C[ i ][ j ] + A[ i ][ k ] * db;
    }
  }
}
```

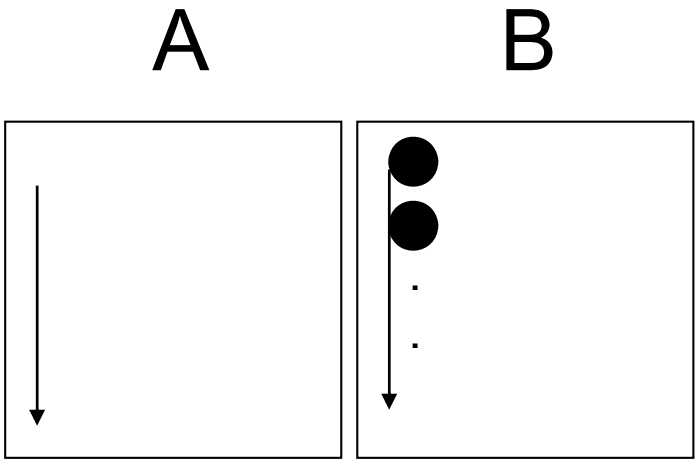


●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

1.5　行列の積（Fortran言語）

- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現

```
do j=1, n
  do i=1, n
    C( i , j ) = 0.0d0
  enddo
  do k=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j ) + A( i , k ) * db
    enddo
  enddo
enddo
```

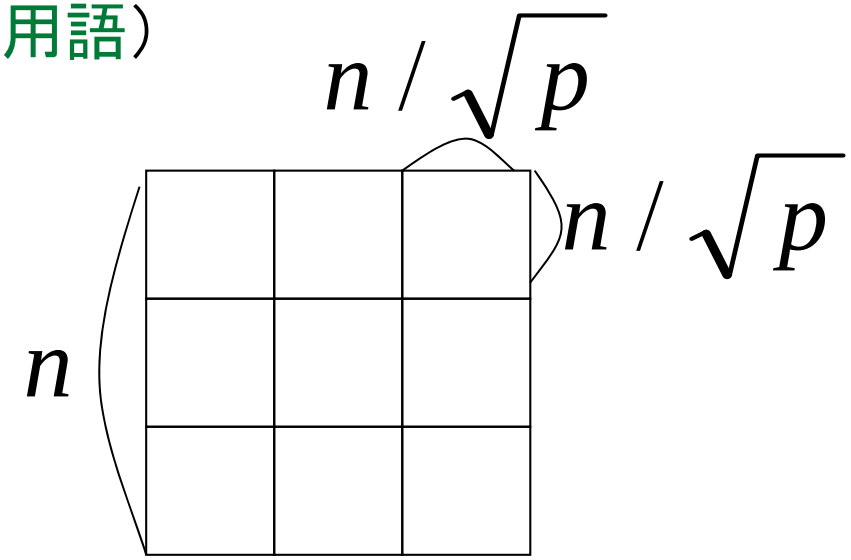


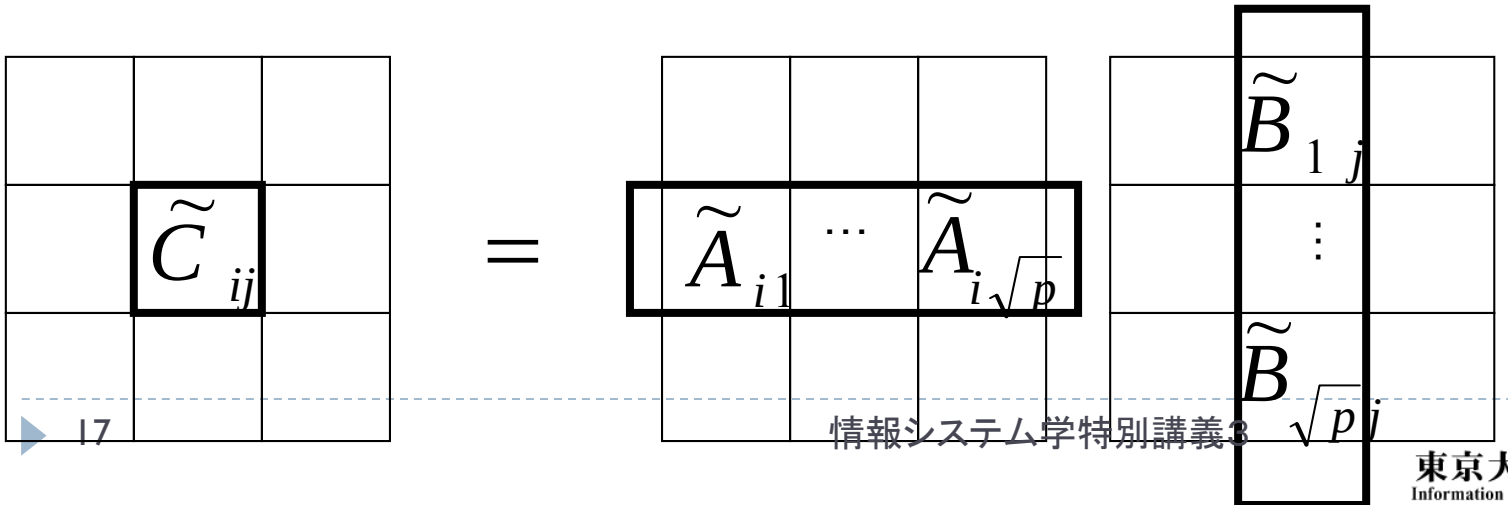
●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

1.5　行列の積

- 小行列ごとの計算に分けて（配列を用意し）計算
  （ブロック化、タイリング：　コンパイラ用語）
- 以下の計算

$$\tilde{C}_{ij} = \sum_{k=1}^{\sqrt{n}} \tilde{A}_{ik} \tilde{B}_{kj}$$



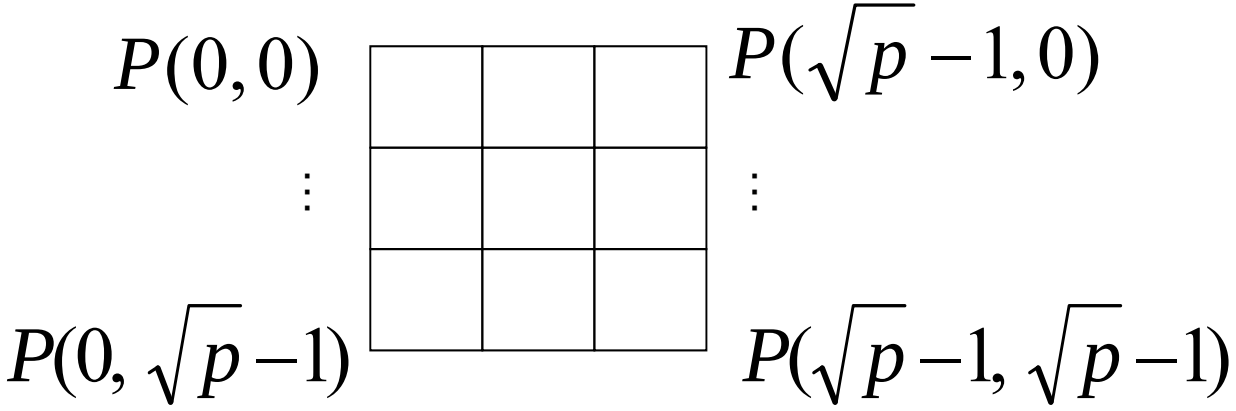


1.5　行列の積

- 各小行列をキャッシュに収まるサイズにする。
  1. ブロック単位で高速な演算が行える
  2. 並列アルゴリズムの変種が構築できる
- 並列行列積アルゴリズムは、データ通信の形態から、以下の2種に分類可能：
  1. セミ・シストリック方式
     - 行列A、Bの小行列の一部をデータ移動
       （Cannonのアルゴリズム）
  2. フル・シストリック方式
     - 行列A、Bの小行列のすべてをデータ移動
       （Foxのアルゴリズム）

1.5.1 Cannonのアルゴリズム

- データ分散方式の仮定
  - プロセッサ・グリッドは　二次元正方



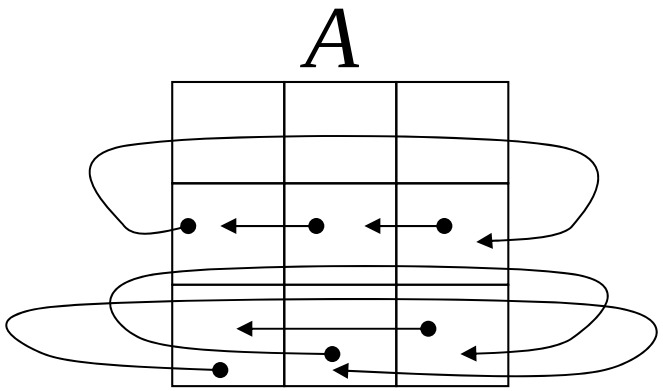
  - PE数が、2のべき乗数しかとれない
  - 各PEは、行列A、B、Cの対応する各小行列を、1つづつ所有
  - 行列A、Bの小行列と同じ大きさの作業領域を所有

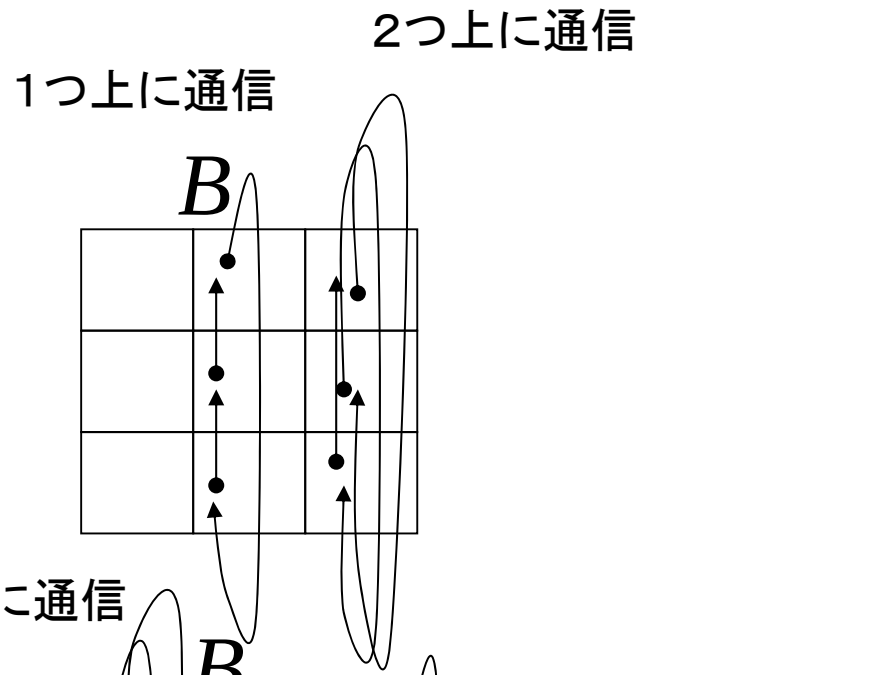
言葉の定義－放送と通信

- 通信
  - ＜通信＞とは、1つのメッセージを1つのPEに送ることである
  - MPI_Send関数、MPI_Recv関数で記述できる処理のこと
  - 1対1通信ともいう
- 放送
  - ＜放送＞とは、同一メッセージを複数のPEに（同時に）通信することである
  - MPI_Bcast関数で記述できる処理のこと
  - 1対多通信ともいう
  - 通信の特殊な場合と考えられる

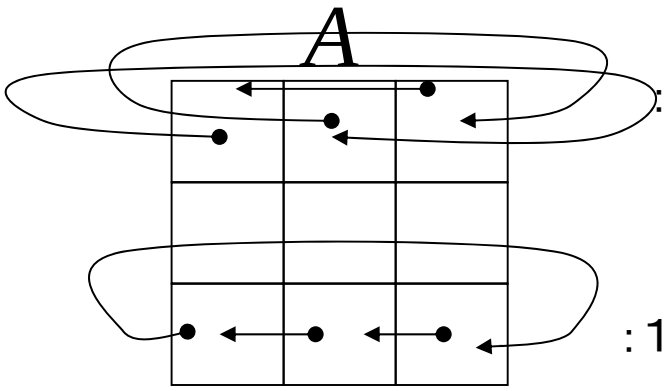
1.5.1 Cannonのアルゴリズム

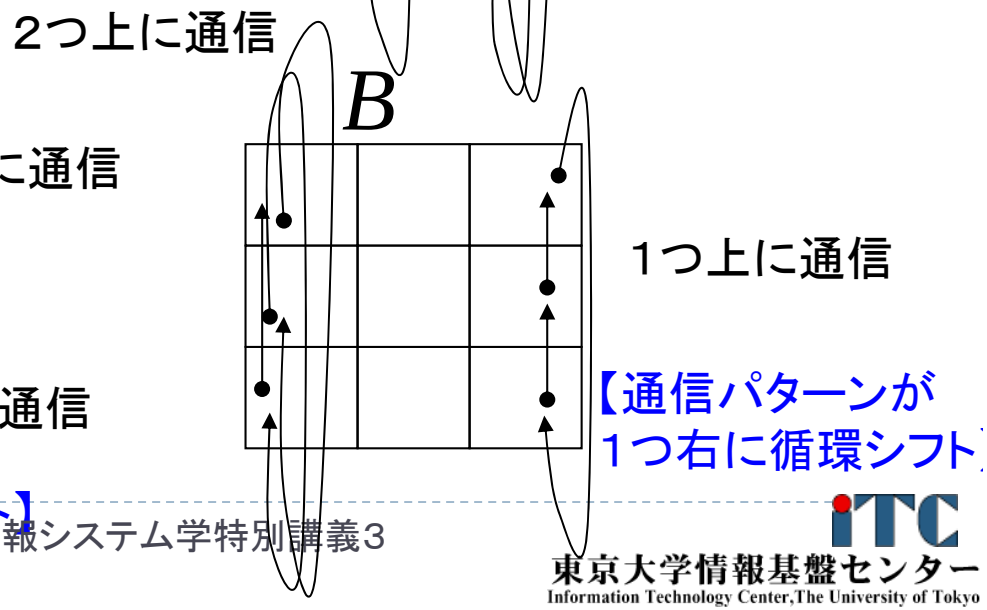
- アルゴリズムの概略
  - 第一ステップ





  - 第二ステップ





【通信パターンが1つ下に循環シフト】

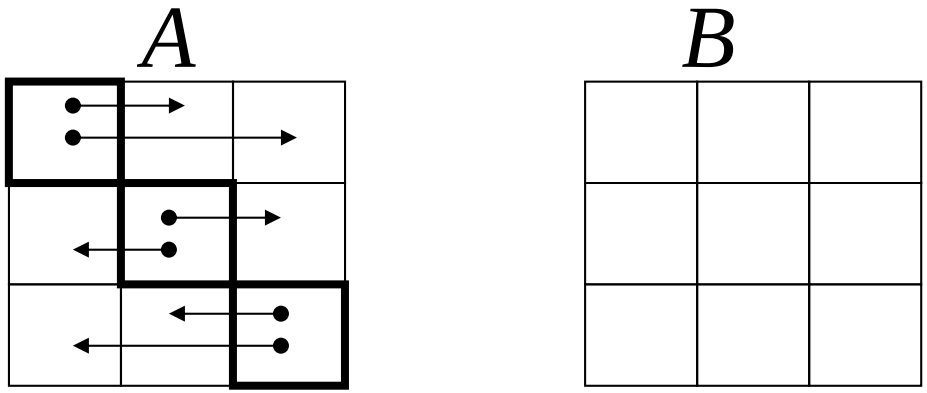
【通信パターンが
1つ右に循環シフト】

1.5.1　Cannonのアルゴリズム

- まとめ
  - ＜循環シフト通信＞のみで実現可能
  - １対１通信（隣接通信）のみで実現可能
  - 物理的に隣接PEにしかつながっていないネットワーク（例えば二次元メッシュ）向き
  - 放送処理がハードウエアでできるネットワークをもつ計算機では、遅くなることも

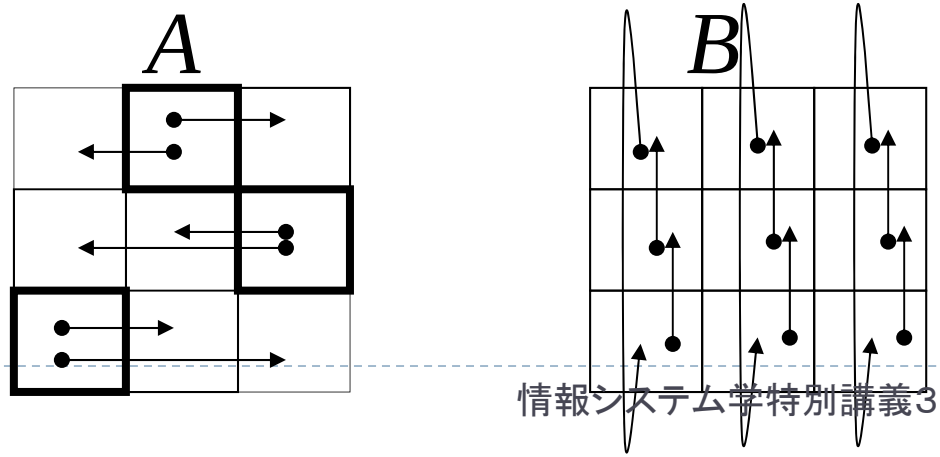
1.5.2　Foxのアルゴリズム

- アルゴリズムの概要
  - 第一ステップ



  - 第二ステップ

【放送PEが
1つ右に
循環シフト】



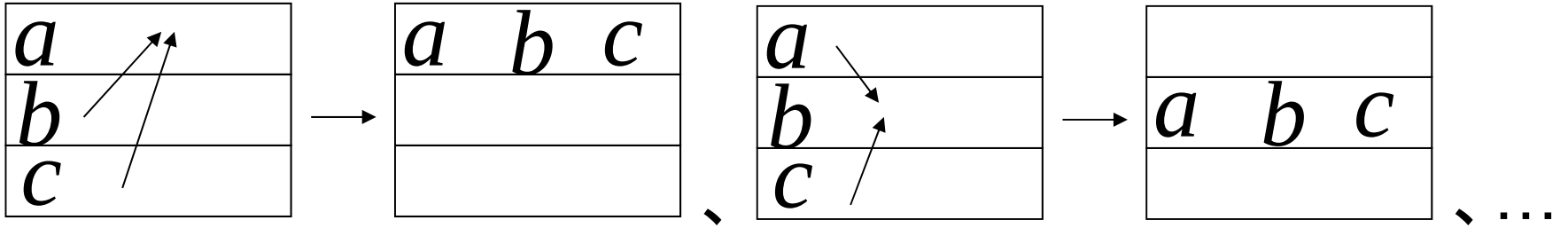
1つ上に通信

1.5.2　Foxのアルゴリズム

- まとめ
  - ＜同時放送（マルチキャスト）＞が必要
  - 物理的に隣接PEにしかつながっていないネットワーク（例えば二次元メッシュ）で性能が悪い（通信衝突が多発）
  - 同時放送がハードウエアでできるネットワークをもつ計算機では、Cannonのアルゴリズムに比べ高速となる

1.5.3　転置を行った後での行列積

- 仮定
  1. データ分散方式：
     行列A、B、C：行方向ブロック分散方式（Block，＊）
  2. メモリに十分な余裕があること：
     分散された行列Bを各PEに全部収集できること
- どうやって、行列Bを収集するか？
  - 2.4節の行列転置の操作をプロセッサ台数回実行



1.5.3　転置を行った後での行列積

- 特徴
  - 一度、行列Bの転置行列が得られれば、*一切通信は不要*
  - 行列Bの転置行列が得られているので、たとえば行方向連続アクセスのみで行列積が実現できる（行列転置の処理が不要）

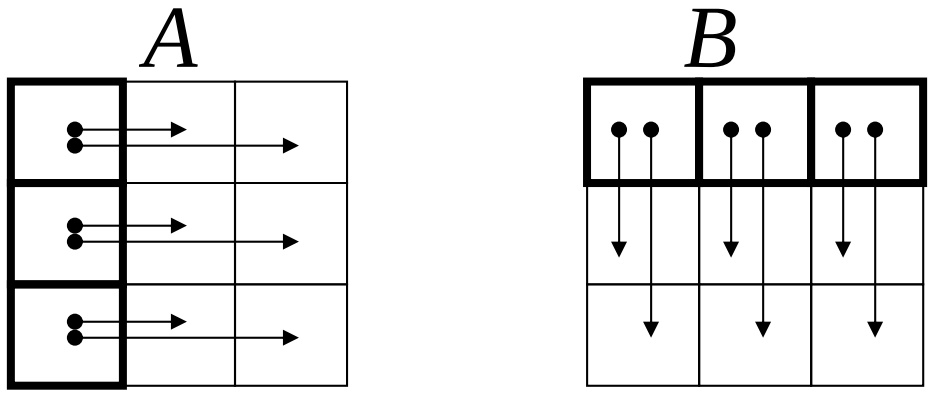
1.5.4　SUMMA、PUMMA

- 近年提案された並列アルゴリズム

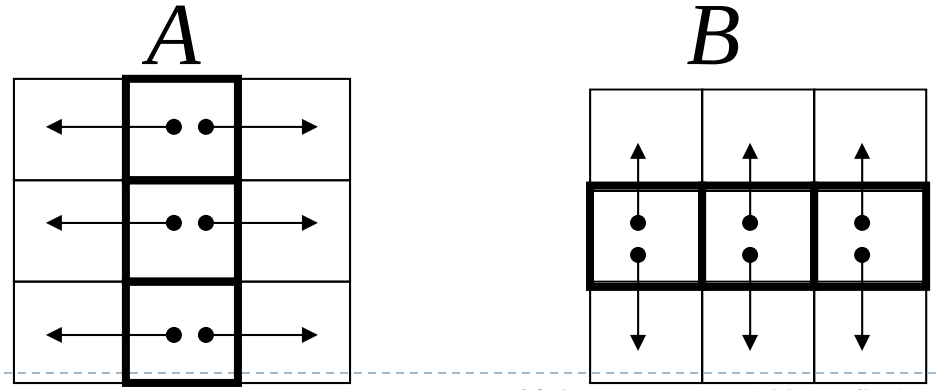
1. SUMMA（Scalable Universal Matrix Multiplication Algorithm）
   - R.Van de Geijinほか、1997年
   - 同時放送（マルチキャスト）のみで実現
2. PUMMA（Parallel Universal Matrix Multiplication Algorithms）
   - Choiほか、1994年
   - 二次元ブロックサイクリック分散方式むきのFoxのアルゴリズム

1.5.4 SUMMA

- アルゴリズムの概略
  - 第一ステップ

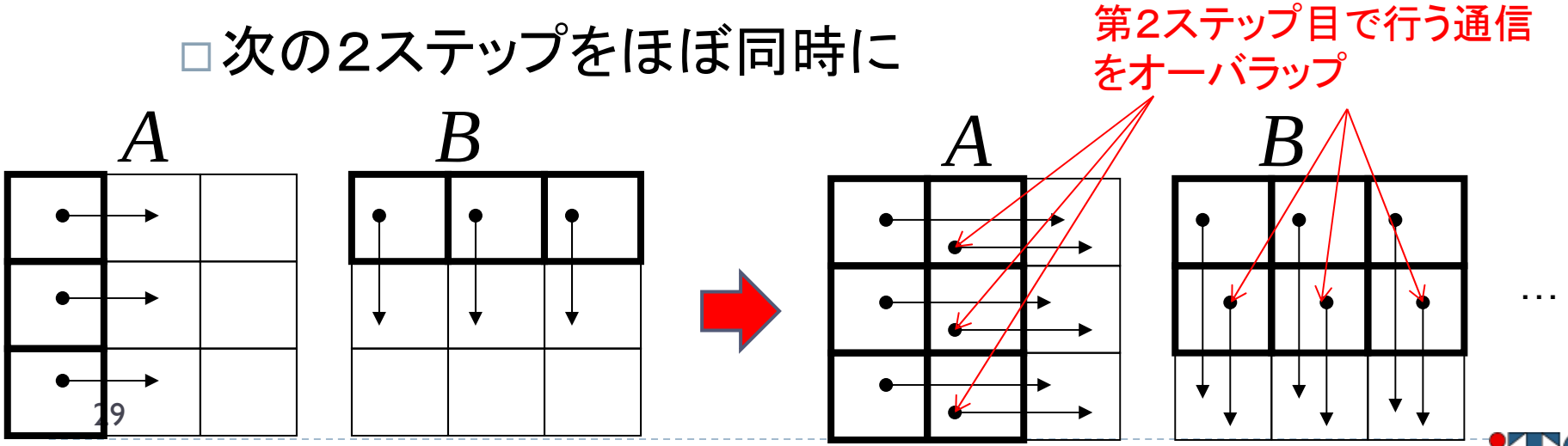


  - 第二ステップ



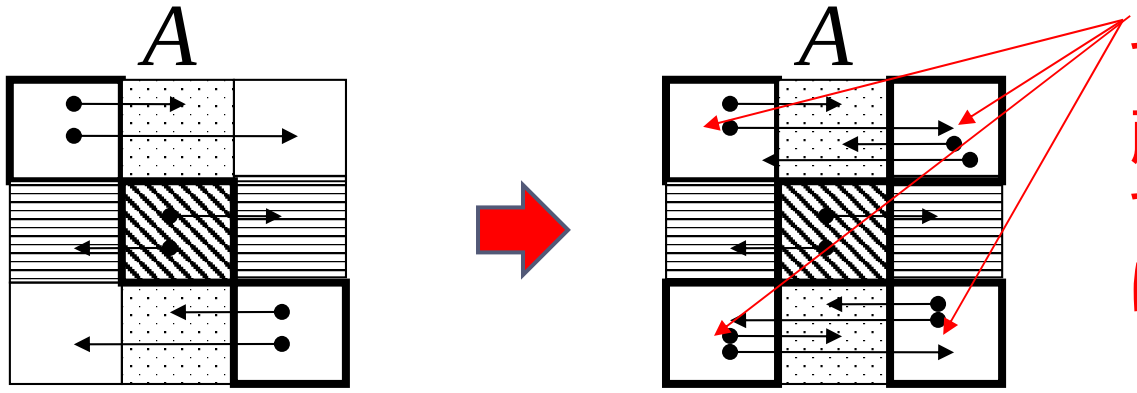
1.5.4 SUMMA

- 特徴
  - 同時放送をブロッキング関数（例. MPI_Bcast）で実装すると、同期回数が多くなり性能低下の要因になる
  - SUMMAにおけるマルチキャストは、非同期通信の1対1通信（例. MPI_Isend）で実装することで、通信と計算のオーバラップ（ ***通信隠蔽*** ）可能
    - 次の2ステップをほぼ同時に



1.5.4 PUMMA

- 概略
  - 二次元ブロックサイクリック分散方式用のFoxアルゴリズム
  - ScaLAPACKが二次元ブロックサイクリック分散を採用していることから開発された
  - 例：



＜同じPE＞が所有しているデータだから、所有データをまとめて＜同一宛先PE＞に一度に送る

1.5.5　Strassenのアルゴリズム

- 素朴な行列積：n^3 の乗算と $(n-1)^3$ の加算
- Strassenのアルゴリズムでは $n^{\log 7}$ の演算
- アイデア：　＜分割統治法＞
  - 行列を小行列に分割して、計算を分割
- 実際の性能
  - 再帰処理や行列のコピーが必要
  - 素朴な実装法より遅くなることがある
  - 再帰の打ち切り、再帰処理展開などの工夫をすれば、（nが大きい範囲で）効率の良い実装が可能

1.5.5　Strassenのアルゴリズム

- 並列化の注意
  - アルゴリズムを単純に分散メモリ型並列計算機に実装すると通信が多発
    - 性能がでない
- PE内の行列積をStrassenで行い、PE間をSUMMAなどで実装すると効率的な実装が可能
- ところが通信量は、アルゴリズムの性質から、通常の行列 - 行列積アルゴリズムに対して減少する。この性質を利用して、近年、Strassenを用いた通信回避アルゴリズムが研究されている。

サンプルプログラムの実行（行列-行列積）

行列-行列積のサンプルプログラムの注意点

- C言語版/Fortran言語版の共通ファイル名

  Mat-Mat-fx.tar

- ジョブスクリプトファイルmat-mat.bash
- pjsub　してください。
  - lecture : 実習時間外のキュー

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-fx.tar ./
$ tar xvf Mat-Mat-fx.tar
$ cd Mat-Mat
```

- 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
```

- 以下共通

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

- 実行が終了したら、以下を実行する

```
$ cat mat-mat.bash.oXXXXXX
```

行列-行列積のサンプルプログラムの実行（C言語）

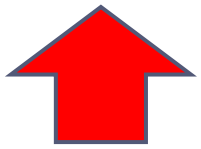
- 以下のような結果が見えれば成功

N = 1000

Mat-Mat time = 0.209609 [sec.]

9541.570931 [MFLOPS]

OK!



1コアのみで、9.5GFLOPSの性能

行列-行列積のサンプルプログラムの実行（Fortran言語）

- 以下のような結果が見えれば成功

NN = 1000

Mat-Mat time[sec.] = 0.2047346729959827

MFLOPS = 9768.741003580422

OK!

1コアのみで、9.7GFLOPSの性能

サンプルプログラムの説明

- #define　N　　1000
  の、数字を変更すると、行列サイズが変更できます
- #define　DEBUG　0
  の「0」を「1」にすると、行列-行列積の演算結果が検証できます。
- MyMatMat関数の仕様
  - Double型N×N行列AとBの行列積をおこない、Double型N×N行列Cにその結果が入ります

Fortran言語のサンプルプログラムの注意

- 行列サイズNの宣言は、以下のファイルにあります。

  mat-mat.inc

- 行列サイズ変数が、NNとなっています。

```
integer  NN
parameter (NN=1000)
```

演習課題（１）

- MyMatMat関数を並列化してください。
  - #define　N　　192
  - #define　DEBUG　1

  として、デバッグをしてください。
- 行列A、B、Cは、各PEで重複して、かつ全部（N×N）所有してよいです。

演習課題（１）

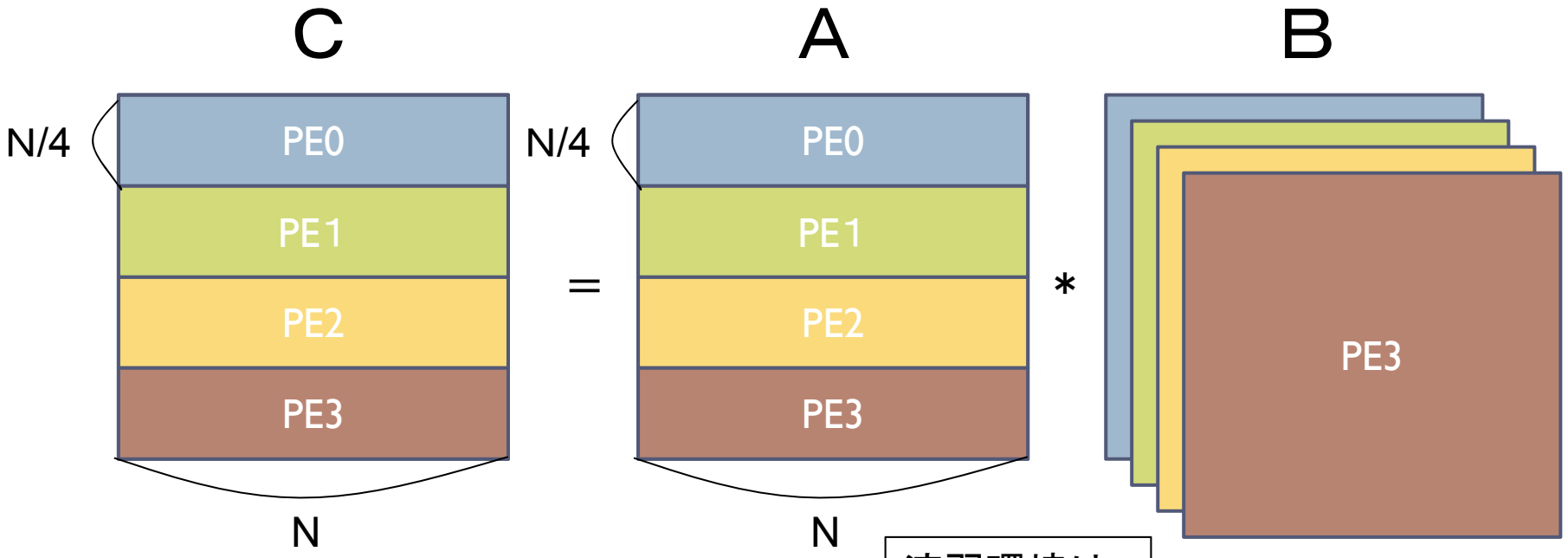
- サンプルプログラムでは、行列A、Bの要素を全部1として、行列-行列積の結果をもつ行列Cの全要素がNであるか調べ、結果検証しています。デバックに活用してください。
  - 行列Cの分散方式により、

演算結果チェックルーチンの並列化が必要

になります。注意してください。

並列化のヒント

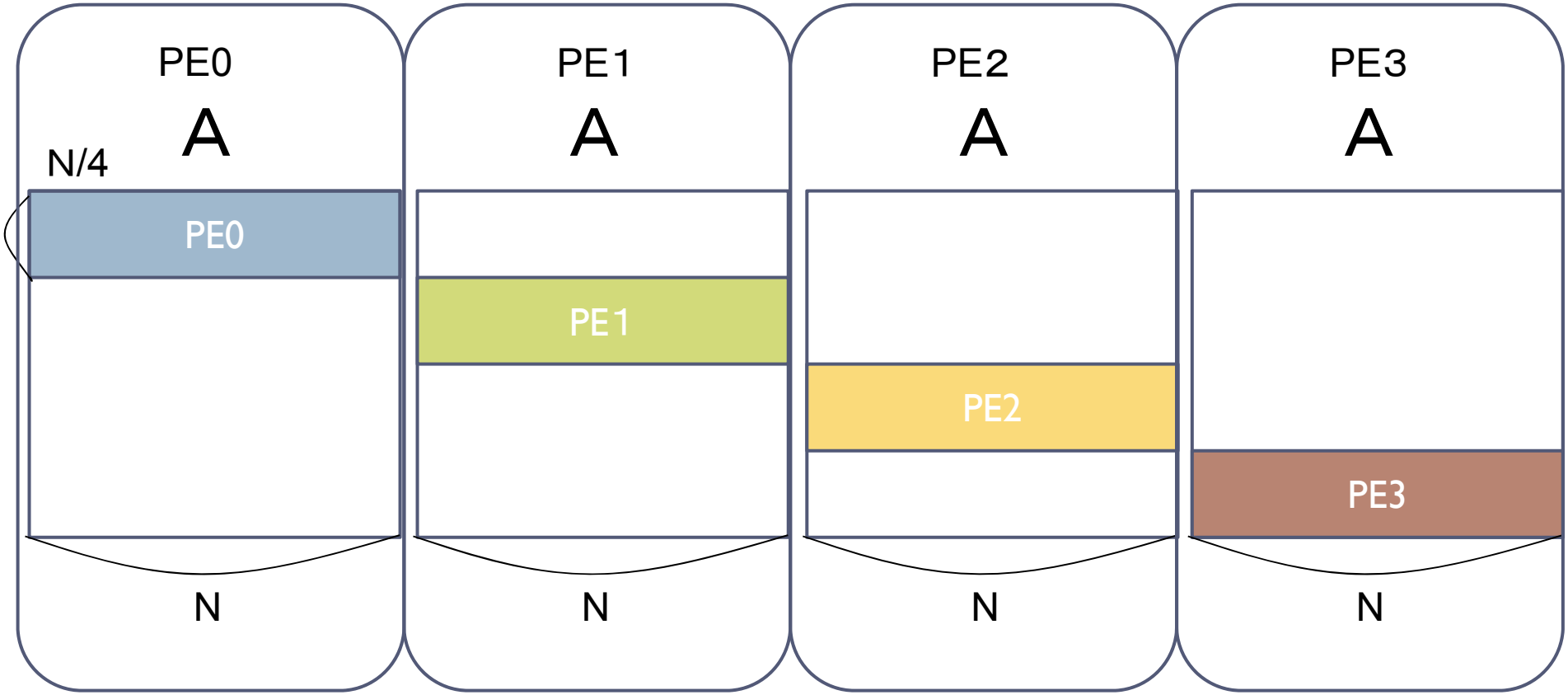
- 以下のようなデータ分割にすると、とても簡単です。



- 通信関数は一切不要です。
- 行列-ベクトル積の演習と同じ方法で並列化できます。

各PEでの配列の確保状況

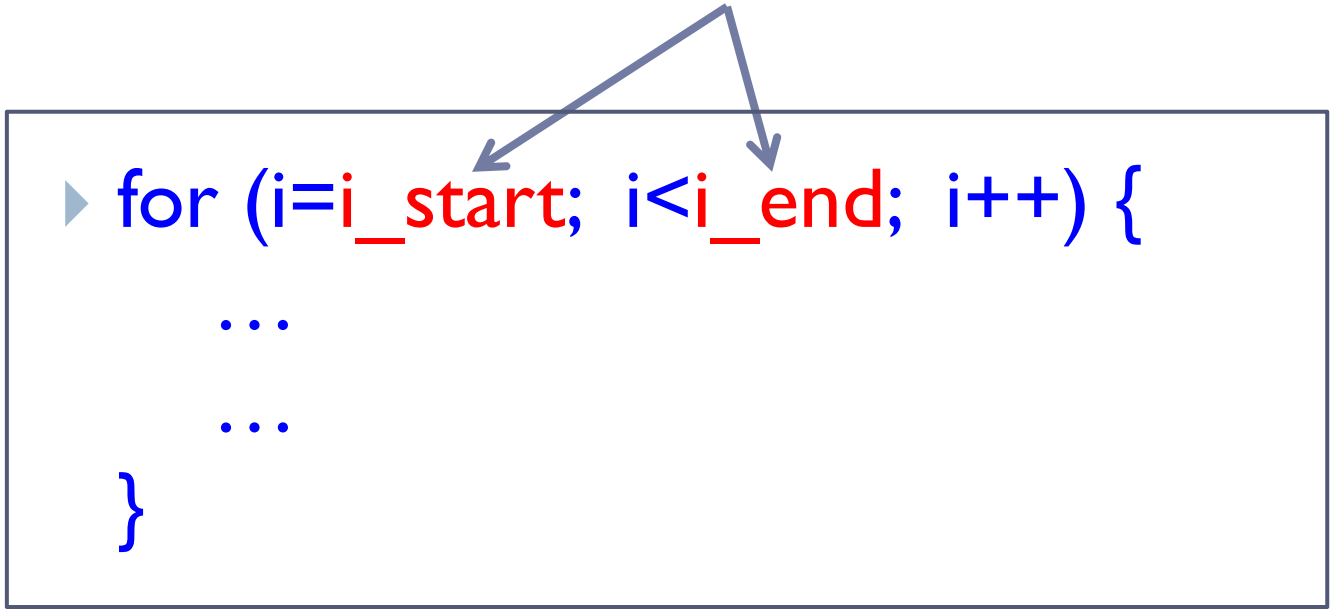
- 実際は、以下のように配列が確保されていて、
  部分的に使うだけになります



実装上の注意

- ループ変数をグローバル変数にすると、性能が出ません。
  必ずローカル変数か、定数（　2　など）にしてください。

ローカル変数にすること



模範解答

行列－行列積のピュアMPI並列化の例（C言語）

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
ierr = MPI_Comm_size(MPI_COMM_WORLD,
&numprocs);
…
ib = n/numprocs;
istart = myid * ib;
iend = (myid+1) * ib;
if ( myid == numprocs-1)  jend=n;

for(i=istart; i<iend; i++) {
    for(j=0; j<n; j++) {
        for(k=0; k<n; k++) {
          C[i][j] += A[i][k] * B[k][j];
        }
    }
 }
```

ブロック分散を仮定した担当ループ範囲の定義

MPIプロセスの担当ごとに縮小したループの構成

行列－行列積のピュアMPI並列化の例（Fortran言語）

```
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs,
ierr)
…
ib = n/numprocs
istart = 1 + myid * ib
iend = (myid+1) * ib
if ( myid .eq. numprocs-1)  jend = n

do i=istart, iend
   do j=1, n
      do k=1, n
         C(i, j) = C(i, j) + A(i, k) * B(k, j)
      enddo
   enddo
enddo
```

ブロック分散を仮定した担当ループ範囲の定義

MPIプロセスの担当ごとに縮小したループの構成

レポート課題

1. [L10]　行列-行列積を並列化せよ。
   ここで、行列A、B、Cについての初期状態は
   各PEで重複したデータをもってよい。
2. [L15]　行列-行列積を並列化せよ。ここで、行列A、B、Cの初期状態は各PEで重複したデータをもってはならない。（来週の演習課題（2））

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. [L25]　Cannonのアルゴリズムを実装せよ。
4. [L25]　Foxのアルゴリズムを実装せよ。
5. [L35]　SUMMAのアルゴリズムを実装せよ。
   ここで放送処理はマルチキャストを用いよ。
6. [L35]　PUMMAのアルゴリズムを実装せよ。
   ここで放送処理はマルチキャストを用いよ。
7. [L40]　１対１通信関数を用いて通信の
   オーバラップを行ったSUMMAのアルゴリ
   ズムを実装せよ。また、マルチキャスト版
   SUMMAと、性能を比較せよ。

レポート課題

8. [L20]　並列化したコードについて、
ピュアMPI実行とハイブリッドMPI実行を行い、
演習環境を駆使して性能評価を行え。また、
ピュアMPI実行が高速となる条件を算出し、
妥当性を実験結果から検証せよ。

講義の流れ

1. 行列-行列積（2）のサンプルプログラムの実行
2. サンプルプログラムの説明
3. 演習課題（2）：ちょっと難しい完全分散版
4. 並列化のヒント

行列 - 行列積の演習の流れ

- 演習課題（1）
  - 簡単なもの（30分程度で並列化）
  - 通信関数が一切不要
- **演習課題（2）**
  - ちょっと難しい（1時間以上で並列化）
  - 1対1通信関数が必要

サンプルプログラムの実行（行列-行列積（２））

行列-行列積のサンプルプログラムの注意点

- C言語版/Fortran言語版のファイル名

  Mat-Mat-d-fx.tar

- ジョブスクリプトファイルmat-mat-d.bash
- qsub してください。
  - lecture : 実習時間外のキュー

行列-行列積(2)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-d-fx.tar ./
  $ tar xvf Mat-Mat-d-fx.tar
  $ cd Mat-Mat-d

- 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

- 以下共通

  $ make
  $ pjsub mat-mat-d.bash

- 実行が終了したら、以下を実行する

  $ cat mat-mat-d.bash.oXXXXXX

行列-行列積のサンプルプログラムの実行（C言語版）

- 以下のような結果が見えれば成功

```
N  = 384
Mat-Mat time  = 0.000135 [sec.]
 841973.194818 [MFLOPS]
 Error! in ( 0 , 2 )-th argument in PE 0
 Error! in ( 0 , 2 )-th argument in PE 61
 Error! in ( 0 , 2 )-th argument in PE 51
 Error! in ( 0 , 2 )-th argument in PE 59
 Error! in ( 0 , 2 )-th argument in PE 50
 Error! in ( 0 , 2 )-th argument in PE 58
....
```

並列化が完成していないのでエラーが出ます。ですが、これは正しい動作です

行列-行列積のサンプルプログラムの実行（Fortran言語）

- 以下のような結果が見えれば成功

```
NN  =  384
Mat-Mat time  =  1.29550899146124 7E-03
MFLOPS =  87414.45135502046
 Error! in ( 1 , 3 )-th argument in PE  0
 Error! in ( 1 , 3 )-th argument in PE  61
 Error! in ( 1 , 3 )-th argument in PE  51
 Error! in ( 1 , 3 )-th argument in PE  58
 Error! in ( 1 , 3 )-th argument in PE  55
 Error! in ( 1 , 3 )-th argument in PE  63
 Error! in ( 1 , 3 )-th argument in PE  60
```

並列化が
完成して
いないので
エラーが出ます。
ですが、
これは正しい
動作です。

サンプルプログラムの説明

- #define N 384
  - 数字を変更すると、行列サイズが変更できます
- #define DEBUG 1
  - 「0」を「1」にすると、行列-行列積の演算結果が検証できます。
- MyMatMat関数の仕様
  - Double型の行列A((N/NPROCS)×N行列)と B((N×(N/NPROCS)行列))の行列積をおこない、Double型の(N/NPROCS)×N行列Cに、その結果が入ります。

Fortran言語のサンプルプログラムの注意

- 行列サイズNの宣言は、以下のファイルにあります。

  mat-mat-d.inc

- 行列サイズ変数が、NNとなっています。

  integer NN

  parameter (NN=384)

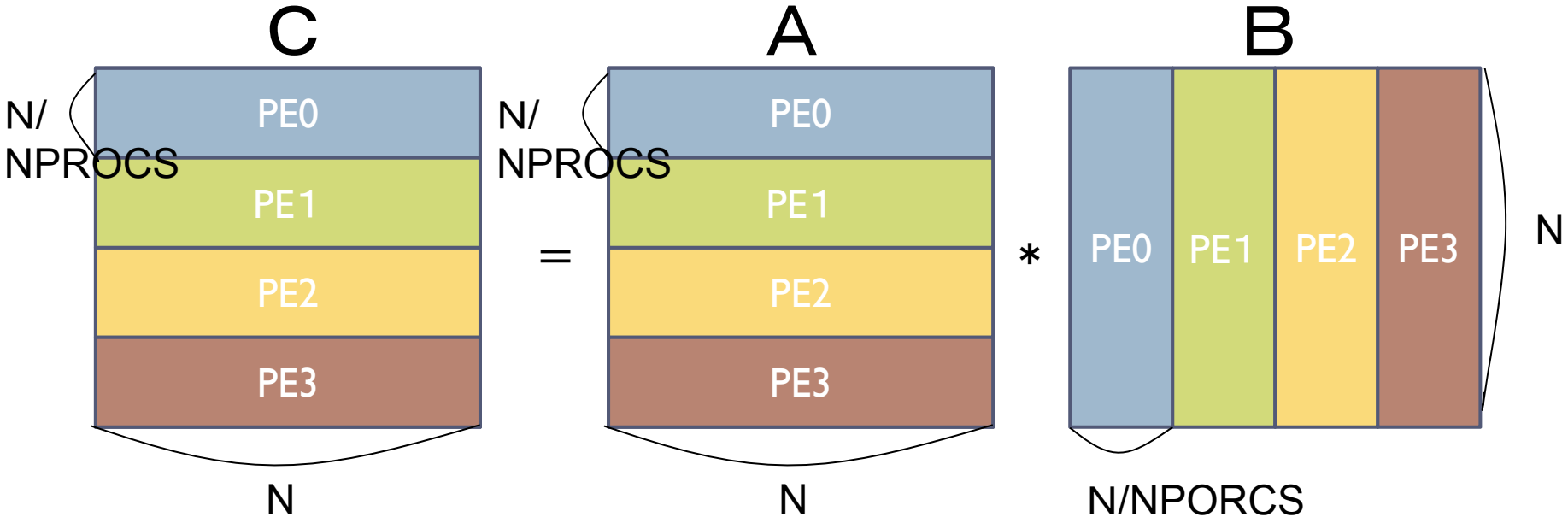
演習課題（１）

- MyMatMat関数（手続き）を並列化してください。
  - デバック時は
    #define　N　384
    としてください。

- 行列A、B、Cの初期配置（データ分散）を、十分に考慮してください。

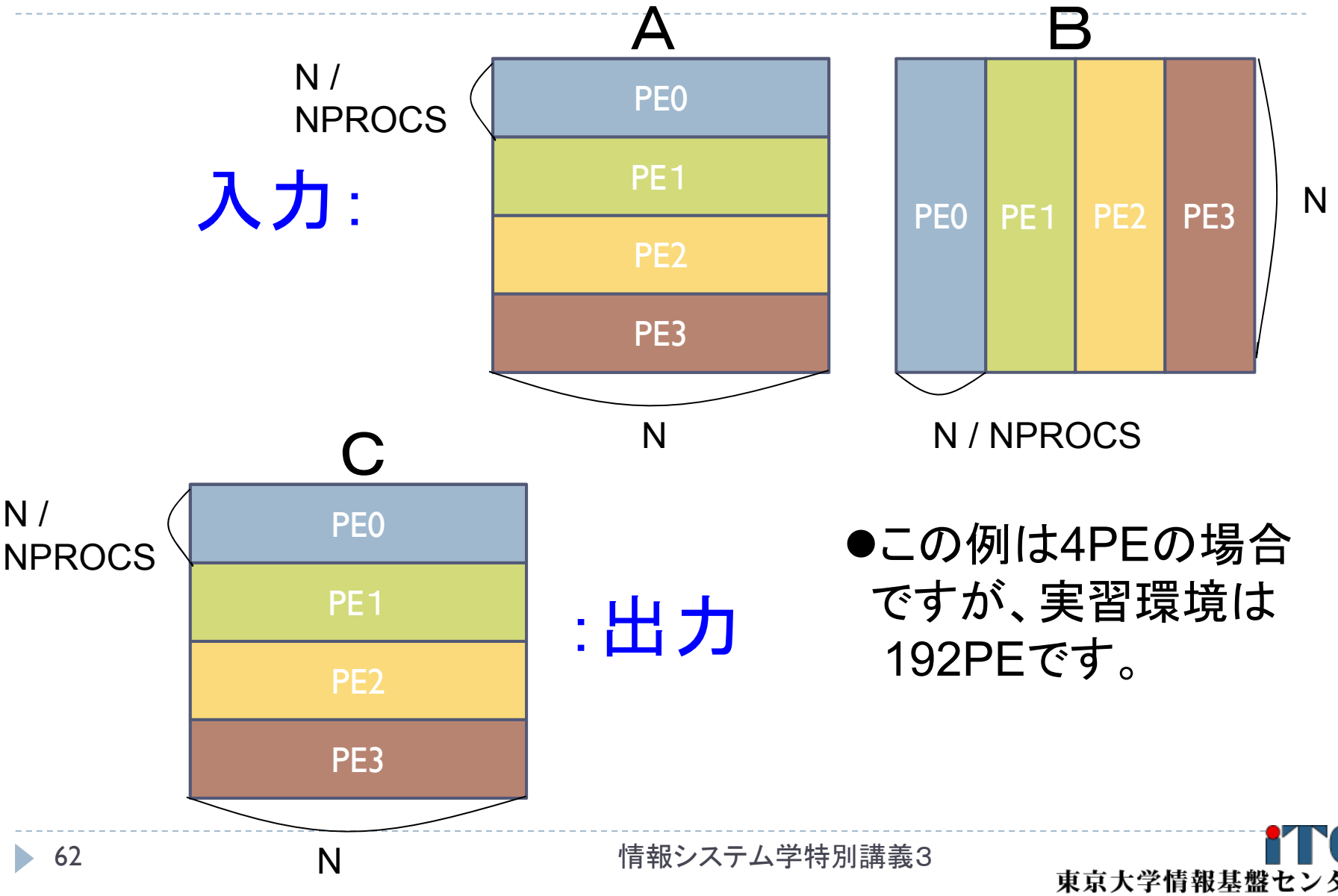
行列A、B、Cの初期配置

- 行列A、B、Cの配置は以下のようになっています。
  （ただし以下は4PEの場合で、実習環境は192PEです。）



- １対１通信関数が必要です。
- 行列A、B、Cの配列のほかに、受信用バッファの配列が必要です。

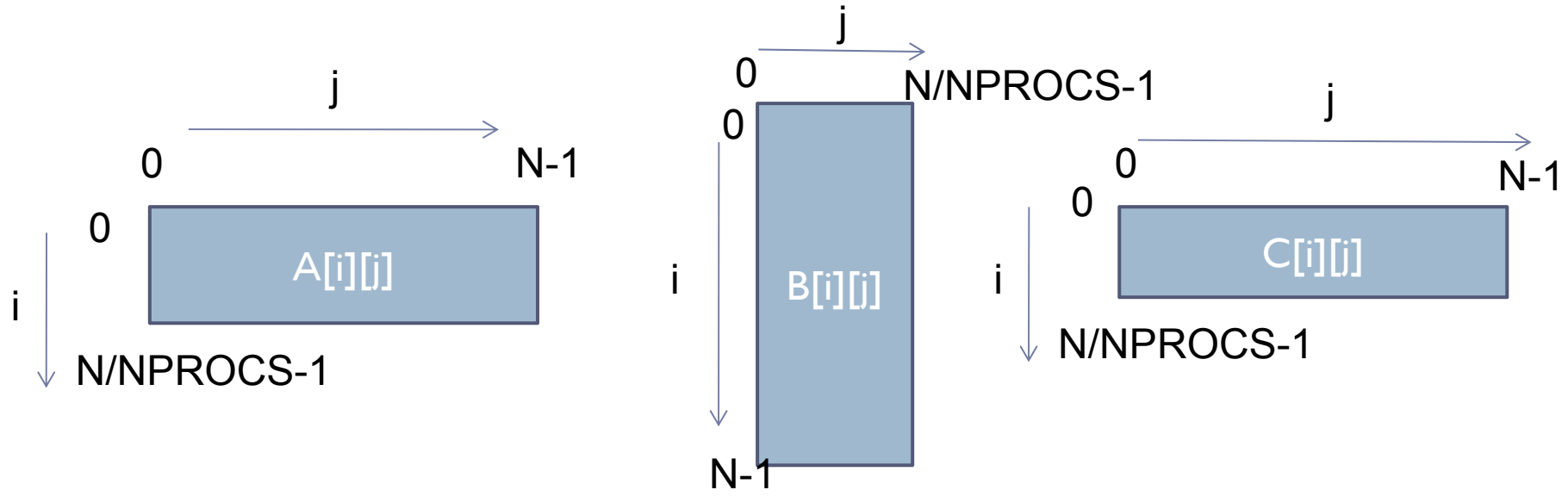
入力と出力仕様



●この例は4PEの場合ですが、実習環境は192PEです。

並列化の注意（C言語）

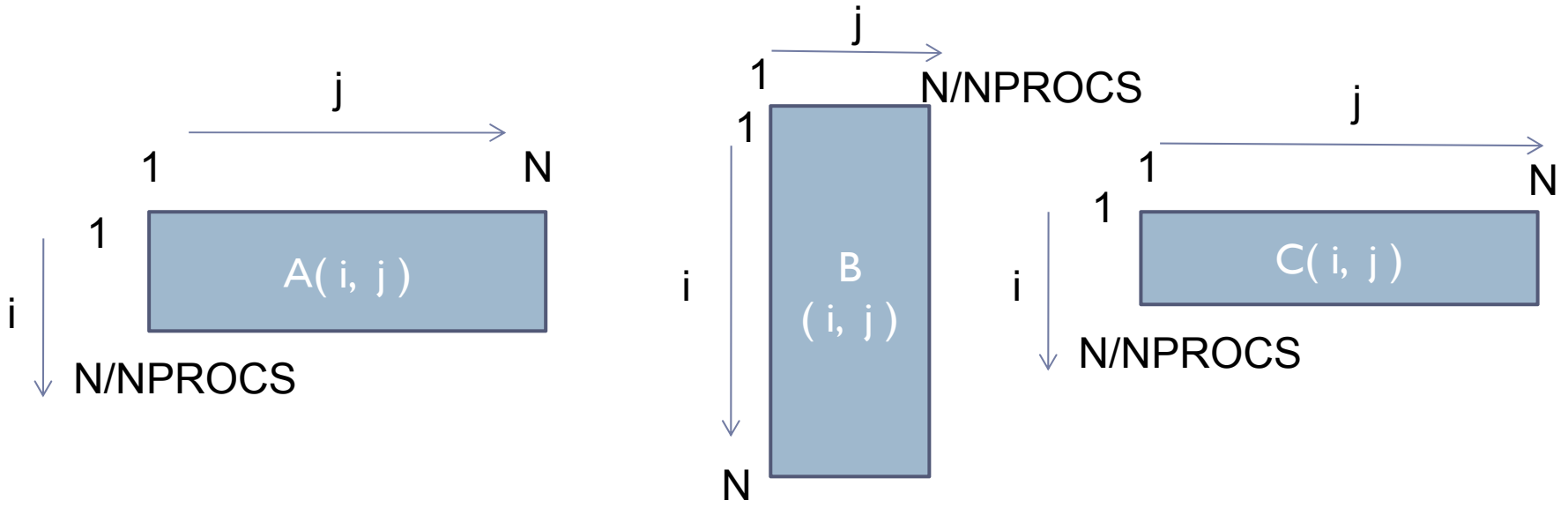
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。



- 各PEで行う、ローカルな行列-行列積演算時のインデックス指定に注意してください。

並列化の注意（Fortran言語）

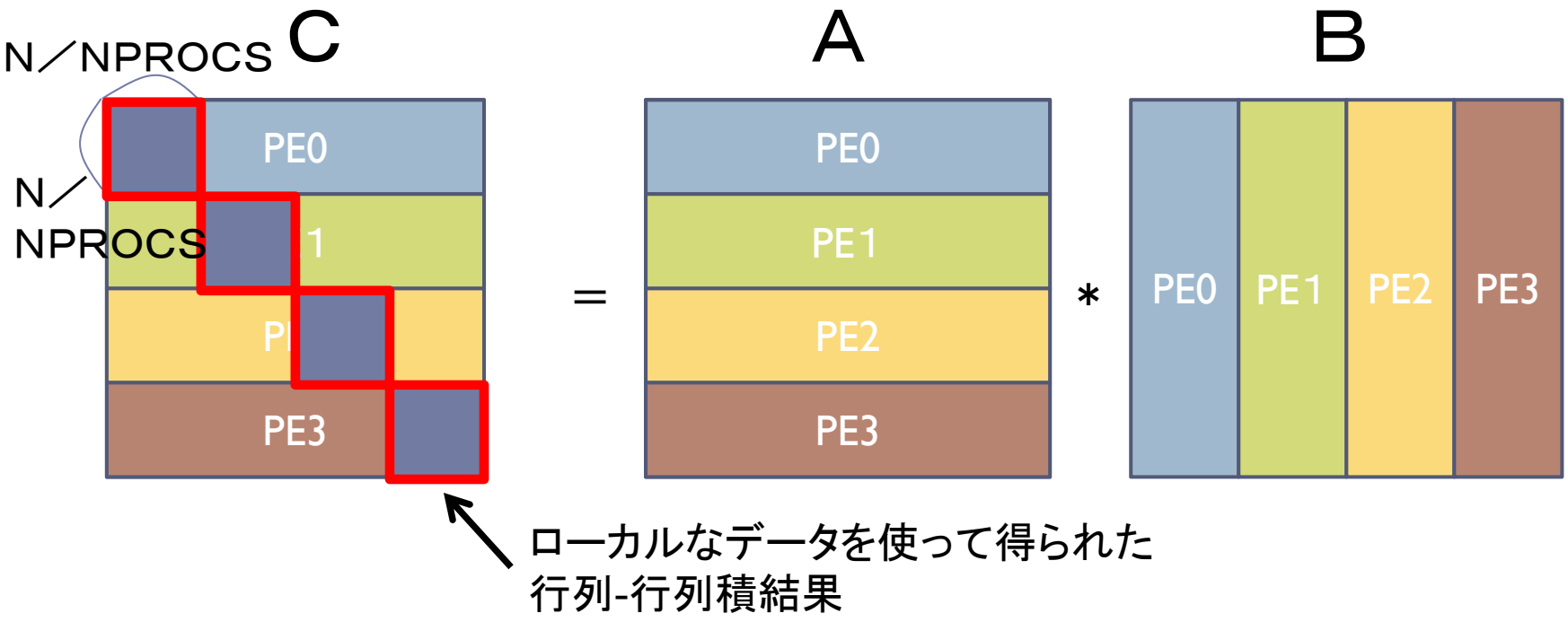
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。



- 各PEで行う、ローカルな行列-行列積演算時のインデックス指定に注意してください。

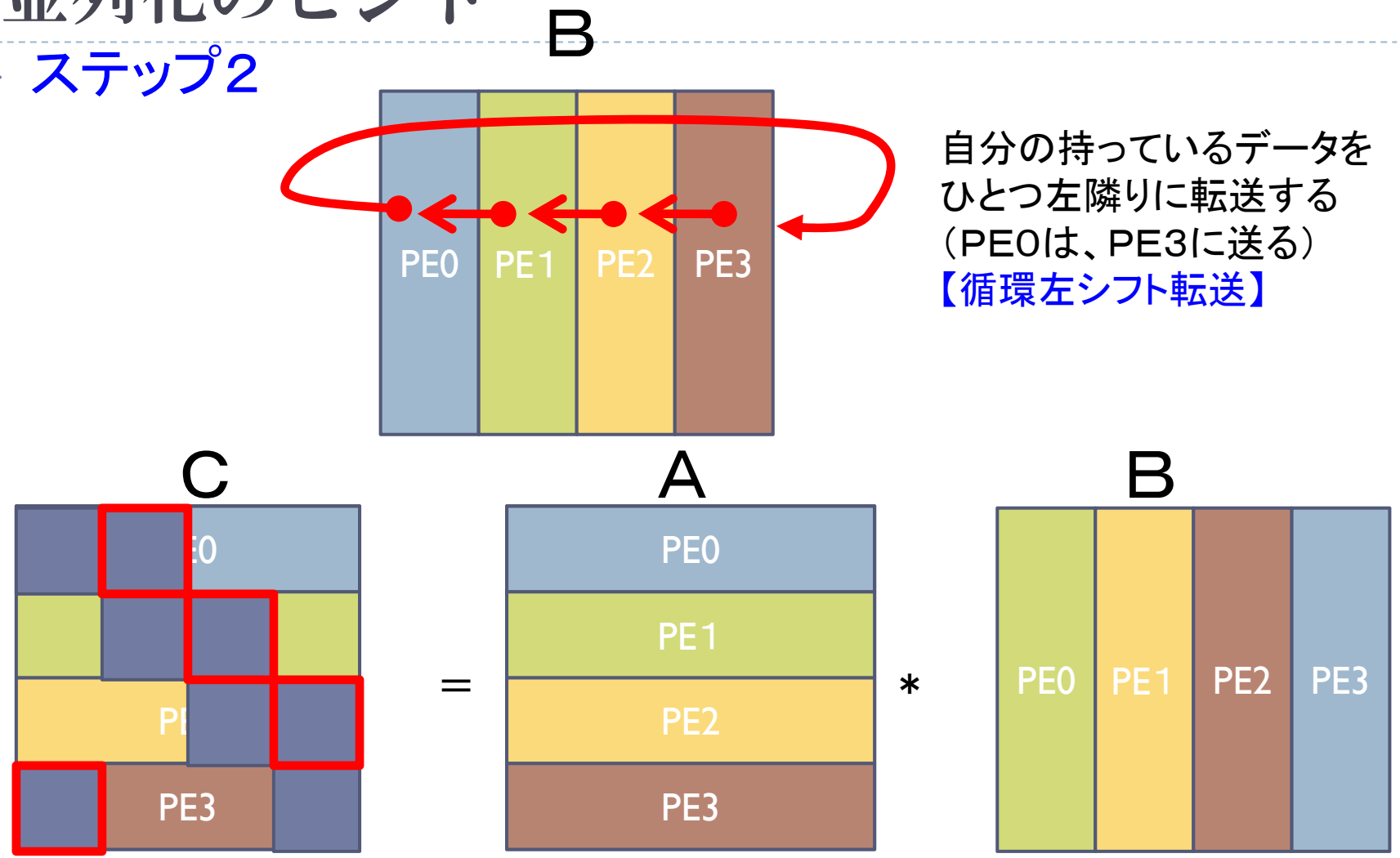
並列化のヒント

- 行列積を計算するには、各PEで完全な行列Bのデータがないので、行列Bのデータについて通信が必要です。
- たとえば、以下のように計算する方法があります。
- ステップ1



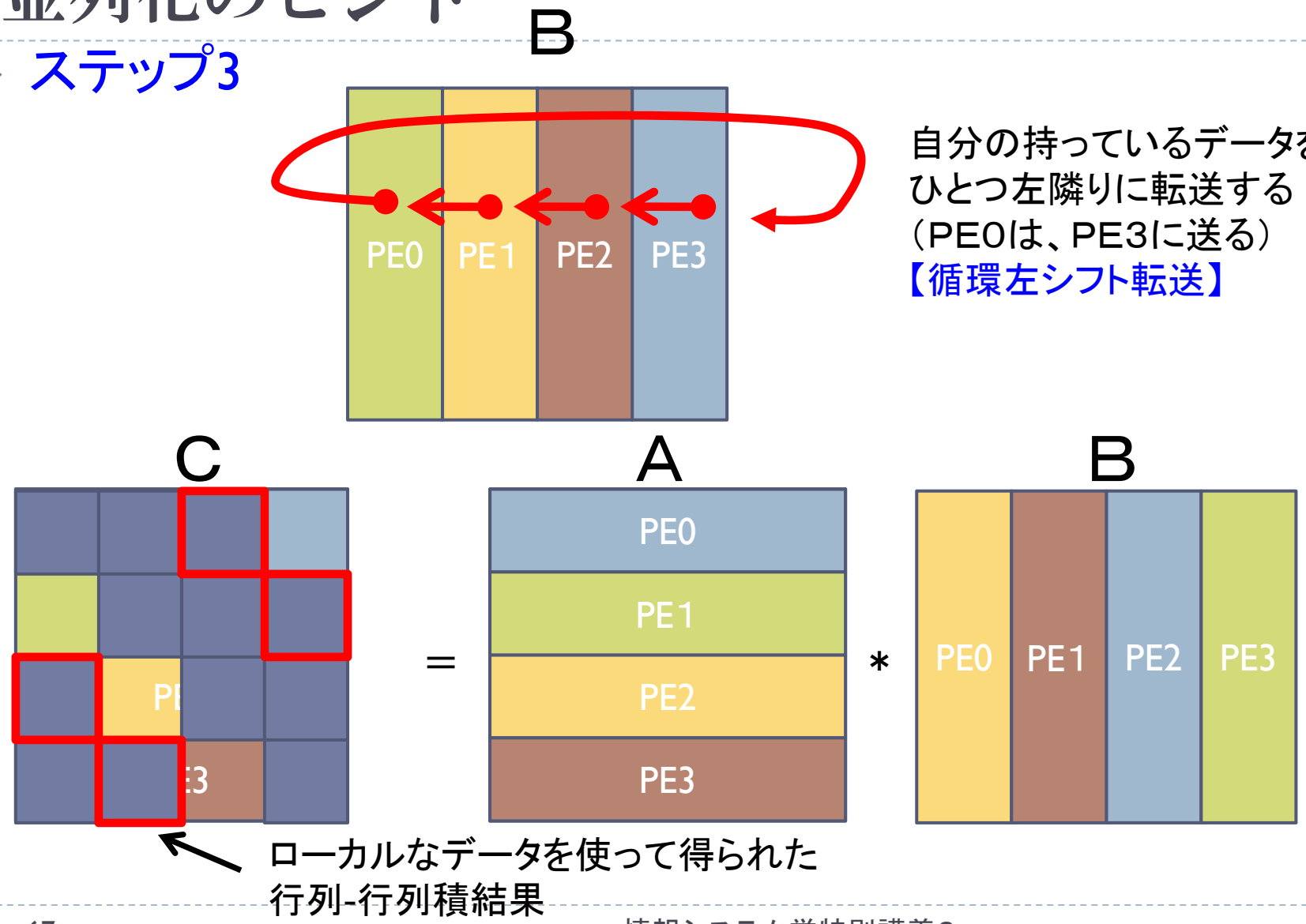
並列化のヒント

- ステップ2



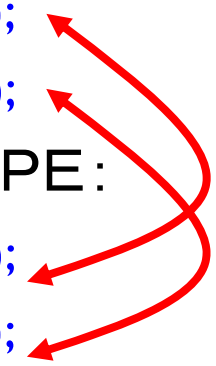
並列化のヒント

- ステップ3



並列化の注意

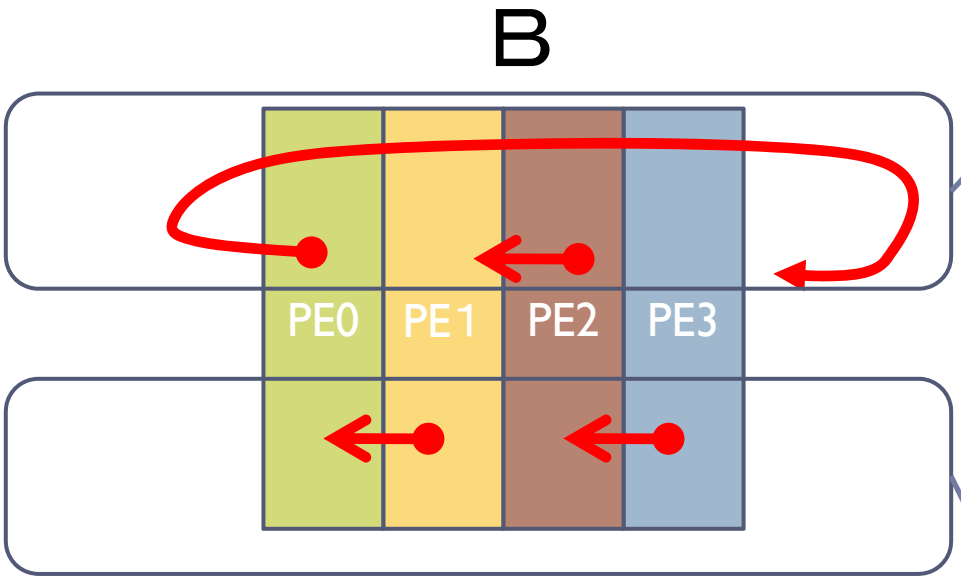
- 循環左シフト転送を実装する際、全員がMPI_Sendを先に発行すると、その場所で処理が止まる。
  （正確には、動いたり、動かなかったり、する）
  - MPI_Sendの処理中で、場合により、バッファ領域がなくなる。
  - バッファ領域が空くまで待つ（スピンウェイトする）。
  - しかし、バッファ領域不足から、永遠に空かない。
- これを回避するため、以下の実装を行う。
  - PE番号が2で割り切れるPE:
    - MPI_Send();
    - MPI_Recv();
  - それ以外のPE:
    - MPI_Recv();
    - MPI_Send();



それぞれに対応

並列化の注意

- つまり、以下の2ステップで、循環左シフト通信をする



ステップ1：
2で割り切れるPEが
データを送る

ステップ2：
2で割り切れないPEが
データを送る

基礎的なMPI関数—MPI_Send

```
ierr = MPI_Send(sendbuf, icount, idatatype, idest,
       itag, icomm);
```

- sendbuf : 送信領域の先頭番地を指定する
- icount : 整数型。送信領域のデータ要素数を指定する
- idatatype : 整数型。送信領域のデータの型を指定する
- idest : 整数型。送信したいPEのicomm内でのランクを指定する。
- itag : 整数型。受信したいメッセージに付けられたタグの値を指定する。
- icomm : 整数型。プロセッサー集団を認識する番号であるコミュニケータを指定する。
- ierr (戻り値) : 整数型。エラーコードが入る。

基礎的なMPI関数—MPI_Recv（１／２）

- ierr = MPI_Recv(recvbuf, icount, idatatype, isource, itag, icomm, istatus);

  - recvbuf : 受信領域の先頭番地を指定する。
  - icount : 整数型。受信領域のデータ要素数を指定する。
  - idatatype : 整数型。受信領域のデータの型を指定する。
    - MPI_CHAR (文字型) 、MPI_INT (整数型)、MPI_FLOAT (実数型)、 MPI_DOUBLE(倍精度実数型)
  - isource : 整数型。受信したいメッセージを送信するPEのランクを指定する。
    - 任意のPEから受信したいときは、MPI_ANY_SOURCE を指定する。

基礎的なMPI関数—MPI_Recv（２／２）

- itag： 整数型。受信したいメッセージに付いているタグの値を指定する。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定する。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- istatus： MPI_Status型（整数型の配列）。受信状況に関する情報が入る。
  - 要素数がMPI_STATUS_SIZEの整数配列が宣言される。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE]、タグが istatus[MPI_TAG] に代入される。
- ierr(戻り値)： 整数型。エラーコードが入る。

実装上の注意

- タグ(itag)について
  - MPI_Send(), MPI_Recv()で現れるタグ（itag）は、任意のint型の数字を指定してよいです。
  - ただし、同じ値（0など）を指定すると、どの通信に対応するかわからなくなり、誤った通信が行われるかもしれません。
  - 循環左シフト通信では、MPI_Send()と MPI_Recv()の対が、2つでてきます。これらを別のタグにした方が、より安全です。
  - たとえば、一方は最外ループの値iloopとして、もう一方をiloop+NPROCSとすれば、全ループ中でタグがぶつかることがなく、安全です。

さらなる並列化のヒント

以降、本当にわからない人のための資料です。
ほぼ回答が載っています。

並列化のヒント

1. 循環左シフトは、PE総数-1回 必要
2. 行列Bのデータを受け取るため、行列B[][]に関するバッファ行列B_T[][]が必要
3. 受け取ったB_T[][] を、ローカルな行列-行列積で使うため、B[][]へコピーする。
4. ローカルな行列-行列積をする場合の、対角ブロックの初期値：ブロック幅*myid。
   ループ毎にブロック幅だけ増やしていくが、
   Nを超えたら0に戻さなくてはいけない。

並列化のヒント（ほぼ回答、C言語）

- 以下のようなコードになる。

```
ib = n/numprocs;
for (iloop=0;  iloop<NPROCS;  iloop++ )  {
    ローカルな行列-行列積  C  =  A * B;
    if (iloop != (numprocs-1) ) {
      if (myid % 2 == 0 ) {
          MPI_Send(B, ib*n, MPI_DOUBLE, isendPE,
                iloop, MPI_COMM_WORLD);
          MPI_Recv(B_T, ib*n, MPI_DOUBLE, irecvPE,
                iloop+numprocs, MPI_COMM_WORLD, &istatus);
      } else {
          MPI_Recv(B_T, ib*n, MPI_DOUBLE, irecvPE,
                iloop, MPI_COMM_WORLD, &istatus);
          MPI_Send(B, ib*n, MPI_DOUBLE, isendPE,
                iloop+numprocs, MPI_COMM_WORLD);
      }
      B[ ][ ]  へ  B_T[ ][ ]  をコピーする;
    }
  }
```

並列化のヒント（ほぼ回答、C言語）

- ローカルな行列-行列積は、以下のようなコードになる。

```
jstart=ib*( (myid+iloop)%NPROCS );
for (i=0;  i<ib;  i++) {
   for(j=0;  j<ib;  j++) {
      for(k=0;  k<n;  k++) {
         C[ i ][ jstart + j ] += A[ i ][ k ] * B[ k ][ j ];
      }
   }
}
```

並列化のヒント（ほぼ回答，Fortran言語）

- 以下のようなコードになる。

```
ib = n/numprocs
do iloop=0,  NPROCS-1
    ローカルな行列-行列積  C  =  A * B
    if (iloop .ne. (numprocs-1) ) then
      if (mod(myid, 2)  .eq. 0 ) then
         call MPI_SEND(B, ib*n, MPI_DOUBLE_PRECISION,  isendPE,
&                iloop, MPI_COMM_WORLD,  ierr)
         call MPI_RECV(B_T,  ib*n,  MPI_DOUBLE_PRECISION,  irecvPE,
&                iloop+numprocs, MPI_COMM_WORLD,  istatus,  ierr)
      else
         call MPI_RECV(B_T,  ib*n,  MPI_DOUBLE_PRECISION,  irecvPE,
&                iloop,  MPI_COMM_WORLD,  istatus,  ierr)
         call MPI_SEND(B,  ib*n,  MPI_DOUBLE_PRECISION,  isendPE,
&                iloop+numprocs,  MPI_COMM_WORLD,  ierr)
      endif
      B  へ  B_T  をコピーする
    endif
  enddo
```

並列化のヒント（ほぼ回答，Fortran言語）

- ローカルな行列-行列積は、以下のようなコードになる。

```
imod = mod( (myid+iloop), NPROCS )
jstart = ib* imod
do i=1, ib
  do j=1, ib
    do k=1, n
      C( i , jstart + j ) = C( i , jstart + j ) +  A( i , k ) * B( k , j )
    enddo
  enddo
enddo
```