Parallelization of Fully Distributed dense Matrix-Matrix Multiplication (1)

東京大学情報基盤中心　准教授　片桐孝洋

Takahiro Katagiri, Associate Professor,
Information Technology Center, The University of Tokyo

台大数学科学中心　科学計算冬季学校

Lessons for Parallelization of Matrix-Matrix Multiplications

- Lesson 1
  - This lesson.
  - Easy to parallelize. It needs 30 minutes or so.
  - No communication is needed.
- Lesson 2
  - Next lesson.
  - Medium level. It needs one hour or so.
  - 1-to-1 communications are used.

What is matrix-matrix multiplication?

The basic operation that can improve performance by code optimization.

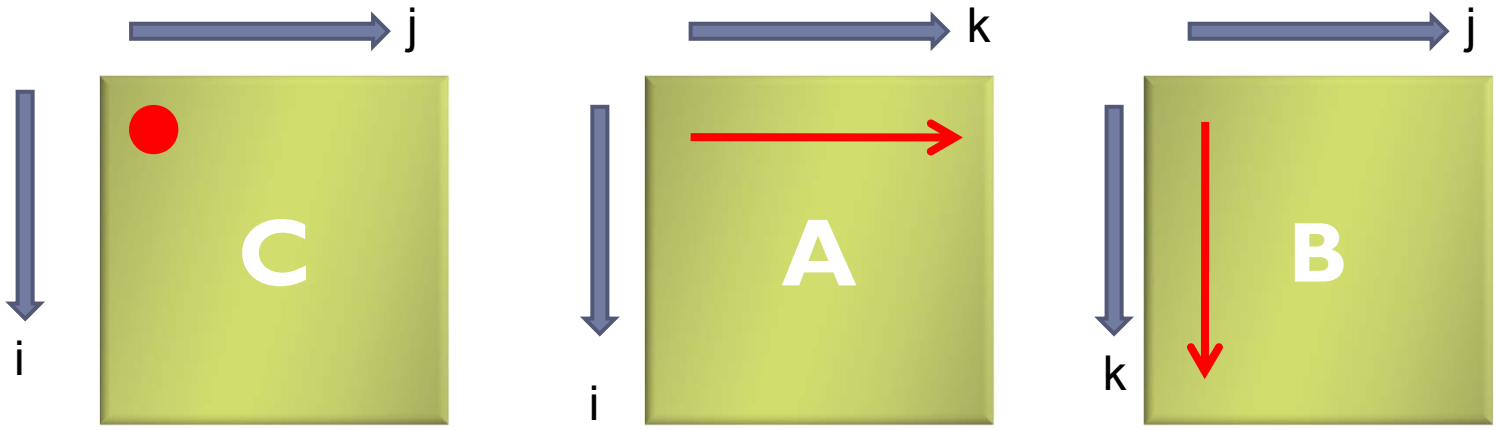
Dense Matrix-Matrix Multiplication

- A dense matrix-matrix multiplication C = A B is utilizing a benchmark for compilers and computer systems.
  - Reason 1: Big impact of performance depends on implementations.
  - Reason 2: Easy to understand. It can also implement codes easily.
  - Reason 3: It represents characteristics of scientific and technology computations.
    1. There is a large <continuous> loop.
    2. It accesses <big data> without cache memory in simple implementation.
    3. If 2, it is memory intensive computation, which accesses memory frequently.

A Simple Implementation (C Language)

- An implementation:

```
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];
```



Optimization Methods for Matrix-matrix Multiplication (MMM)

- A Matrix-matrix multiplication:

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

can be optimized by the followings:

1. Loop Exchange Method:
   - Exchange 3-nested loops of MMM to perform continuous access.
2. Blocking (Tiling) Method:
   - Implement codes to reuse of data in a partial part of matrices in cache memory.

Loop Exchange Method (C Language)

- The loop of MMM forms the following 3-nested loop:

```
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 ];
      }
   }
}
```

  - Although we exchange the outer loops, result of computation is not changed with respect to inner computation.

    → Hence we have 6 ways to exchange the loop.

Loop Exchange Method (Fortran Language)

- The loop of MMM forms the following 3-nested loop:

```
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
```

  - If we exchange the outer loops, results of computation do not change with respect to inner computation.
    → Hence we have 6 ways to exchange the loop.

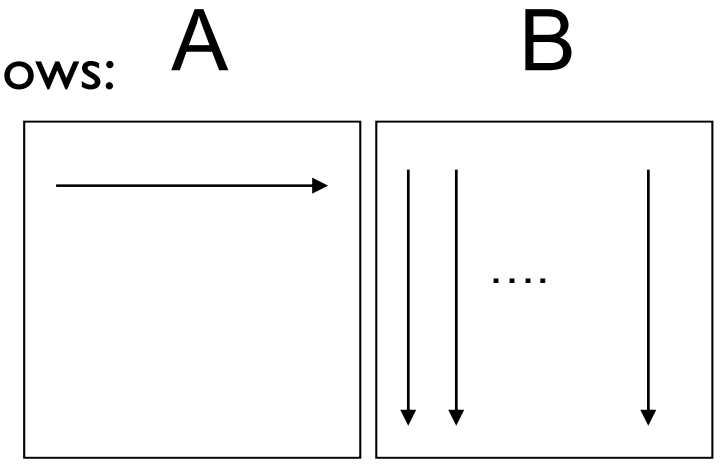
Classification of MMM

- There are three classifications for MMM according to memory access pattern.
  1. Inner-product form
     It is same as <dot products of vectors> for access pattern of the inner computation.
  2. Outer-product form
     It is same as <outer products of vectors> for access pattern of the inner products.
  3. Middle-product form
     It is hybrid form between inner-product and outer-product forms.

The inner-product form of MMM (C Language)

- Inner-product form
  - Implementation with ijk, jik loops as follows:

```
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;
  }
}
```



*Here after, we denote implementation with order of loop induction variables from the outer loop. For example, the above code is <ijk loop>.

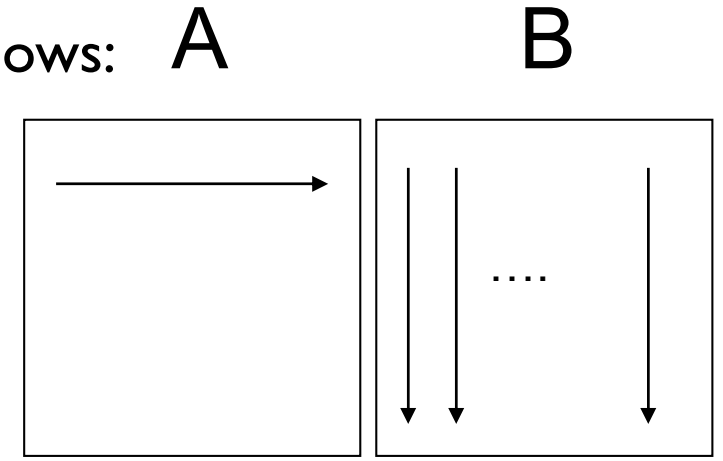
* With accesses for row-size and column-wise:
→ Performance goes down between languages that provide row-wise and column-wise allocations.
One of solutions:
Transpose array for A or B.

The inner-product form of MMM (Fortran Language)

- Inner-product form
  - Implementation with ijk, jik loops as follows:

```
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
```

*Here after, we denote implementation with order of loop induction variables from the outer loop. For example, the above code is <ijk loop>.

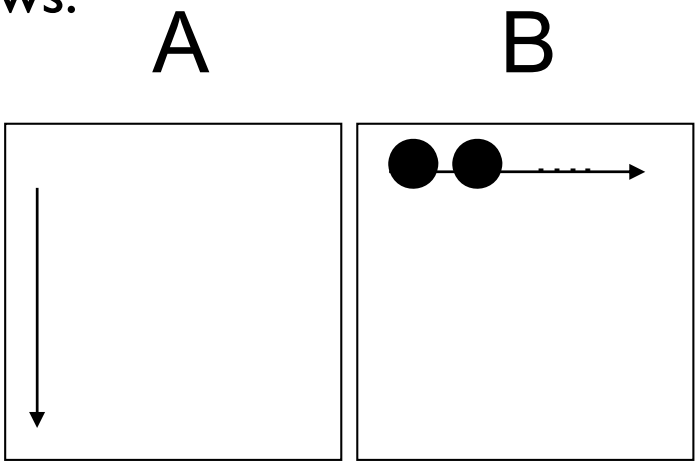


* With accesses for row-size and column-wise:
  → Performance goes down between languages that provide row-wise and column-wise allocations.
 One of solutions:
  Transpose array for A or B.

The outer-product form of MMM (C Language)

- Outer-product form
  - Implementation with kij, kji loops as follows:

```
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;
    }
  }
}
```

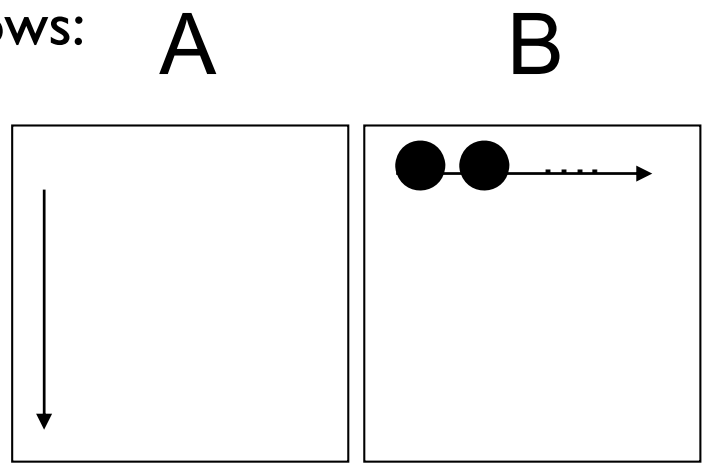


* In kji loop, main access direction is column-wise.
→ It is good for language that provides column-wise array allocation.
(Fortran)

The outer-product form of MMM (Fortran Language)

- Outer-product form
  - Implementation with kij, kji loops as follows:

```
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
```

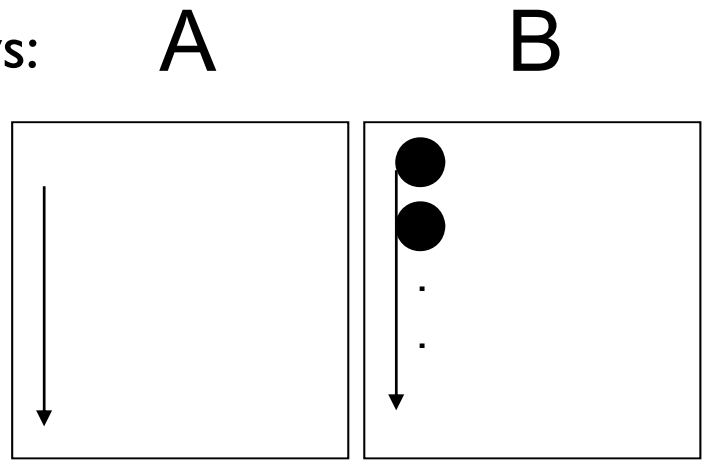


* In kji loop, main access direction is column-wise.
→ It is good for language that provides colmn-wise array allocation. (Fortran)

The middle-product form of MMM (C Language)

- Middle-product form
  - Implementation with ikj, jki loops as follows:

```
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;
    }
  }
}
```

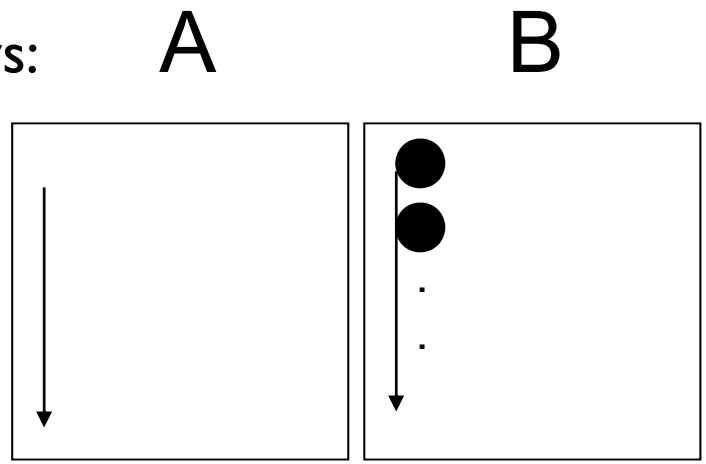


* In jki loop, all access directions are column-wise.
  → It is the best for language that provides column-wise array allocation. (Fortran)

The middle-product form of MMM (Fortran Language)

- Middle-product form
  - Implementation with ikj, jki loops as follows:

```
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
```

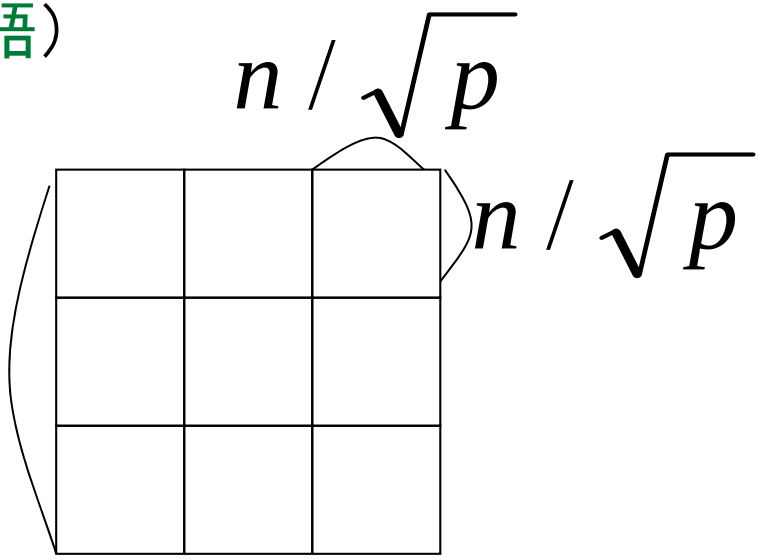


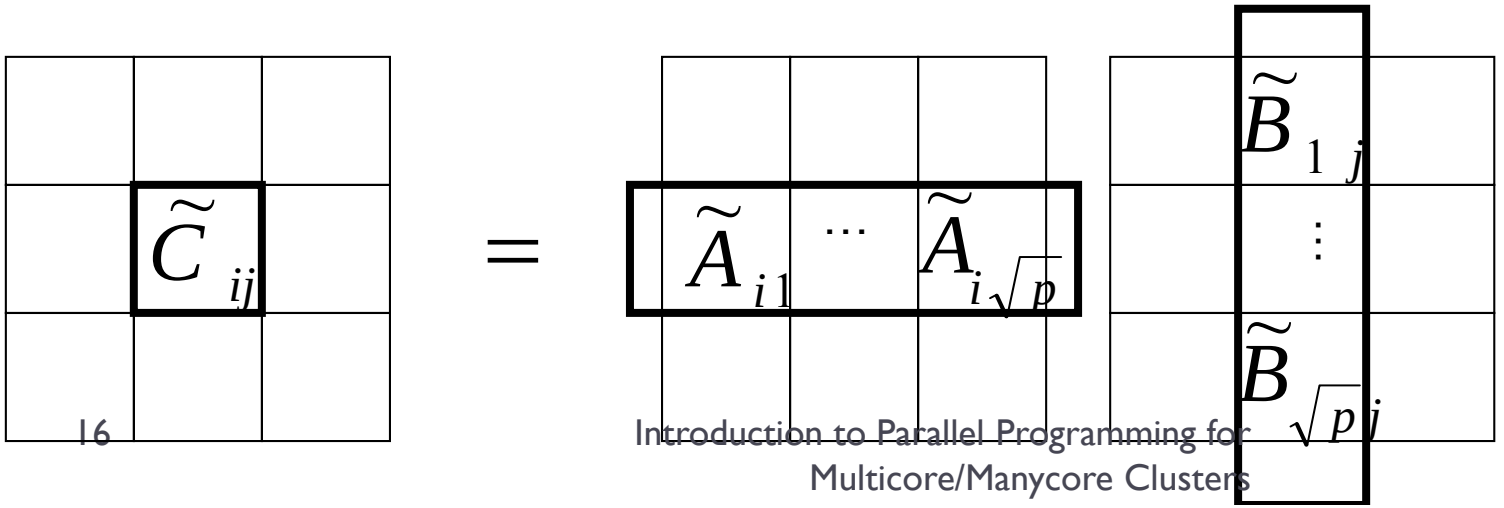
* In jki loop, all access directions are column-wise.
  → It is the best for language that provides column-wise array allocation. (Fortran)

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

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



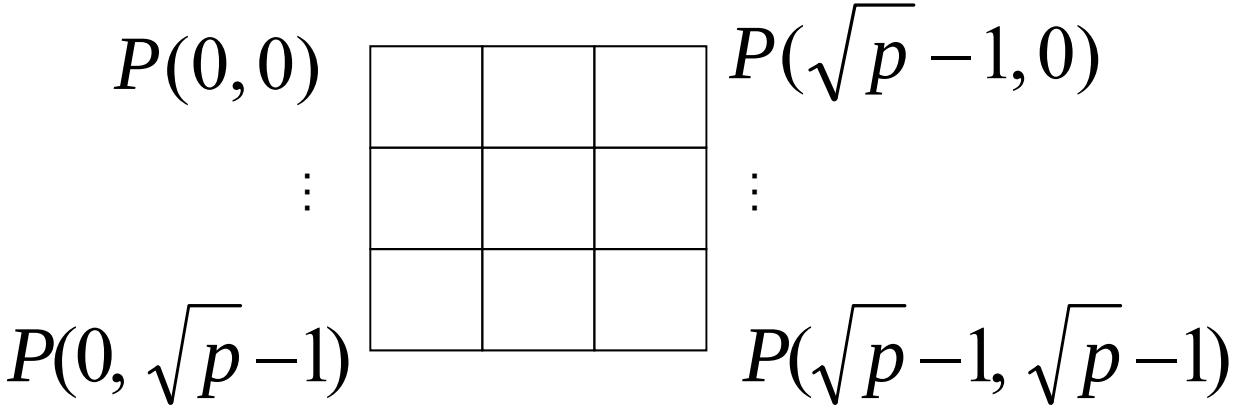


1.5　行列の積

- 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



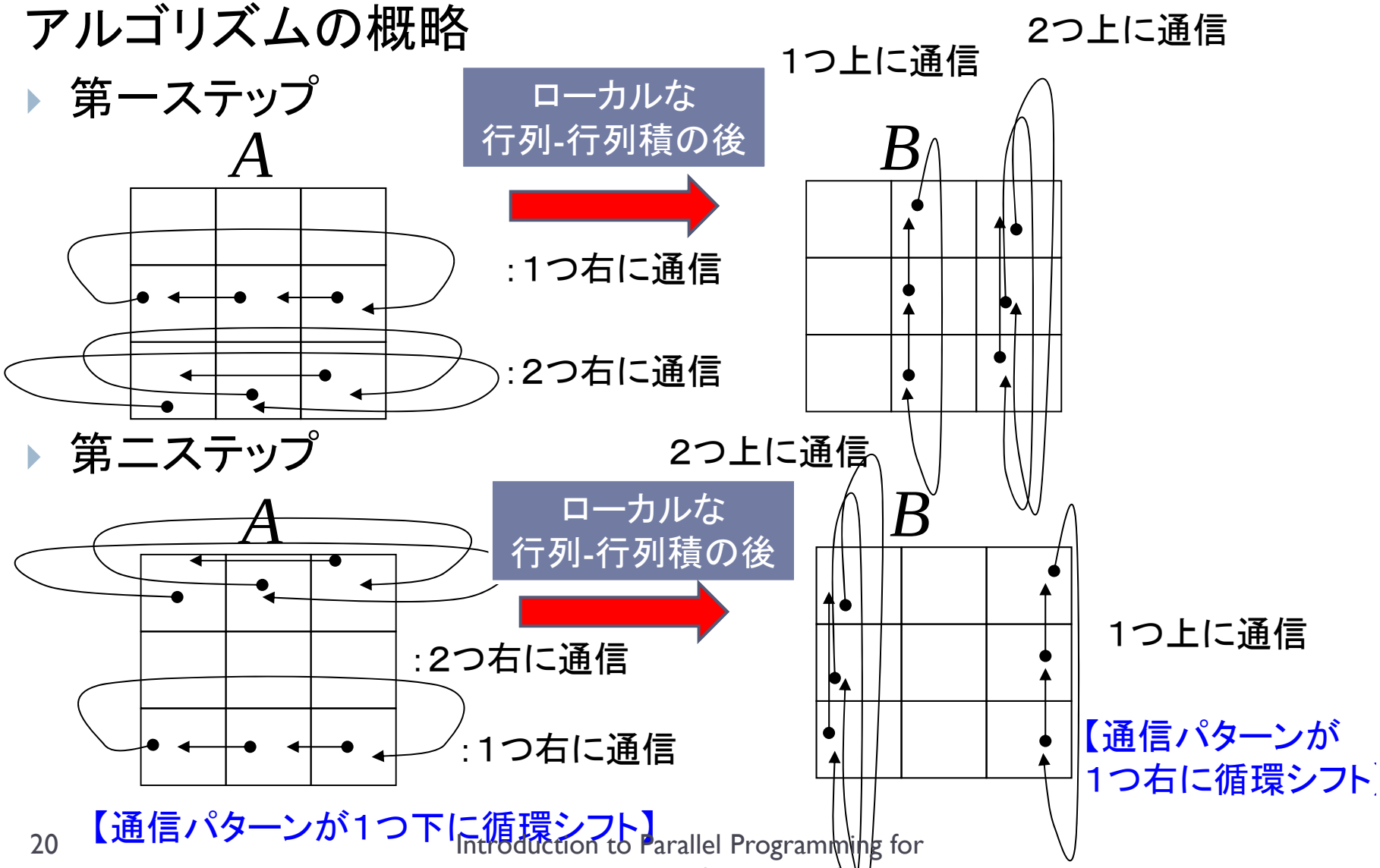
  - PE数が、2のべき乗数しかとれない
  - 各PEは、行列A、B、Cの対応する各小行列を、1つづつ所有
  - 行列A、Bの小行列と同じ大きさの作業領域を所有

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

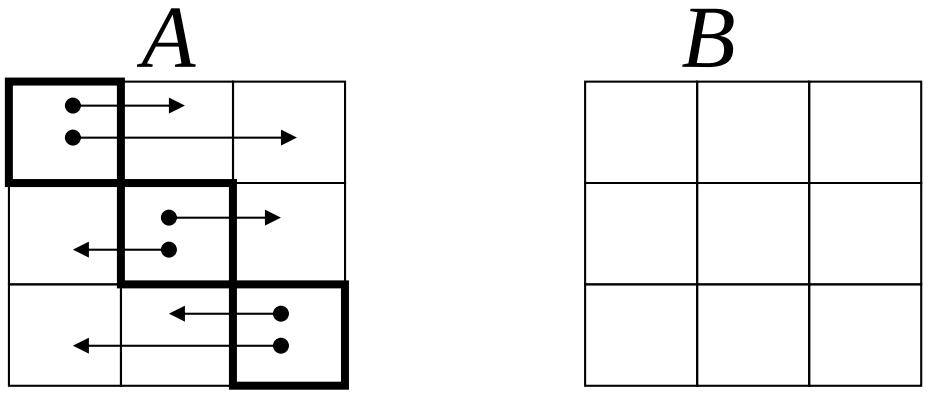
【通信パターンが1つ下に循環シフト】

1.5.1　Cannonのアルゴリズム

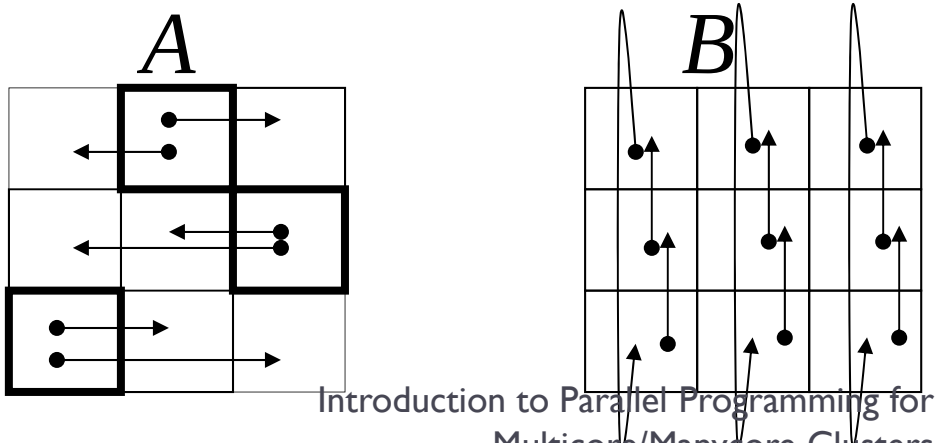
- まとめ
  - ＜循環シフト通信＞のみで実現可能
  - １対１通信（隣接通信）のみで実現可能
  - 物理的に隣接PEにしかつながっていないネットワーク（例えば二次元メッシュ）向き
  - 放送処理がハードウエアでできるネットワークをもつ計算機では、遅くなることも

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】

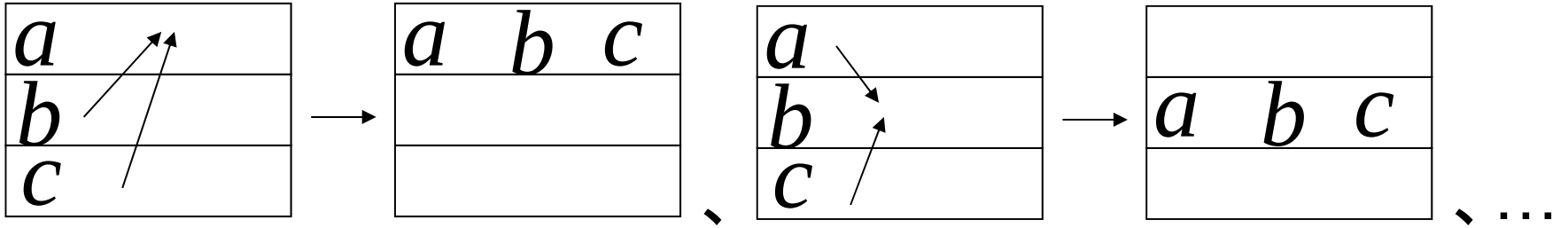
1つ上に通信

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）

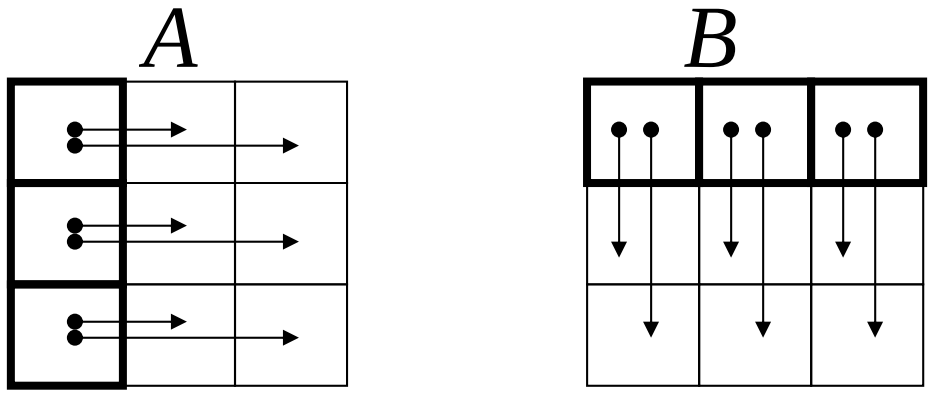
1.5.4 SUMMA、PUMMA

- 近年提案された並列アルゴリズム

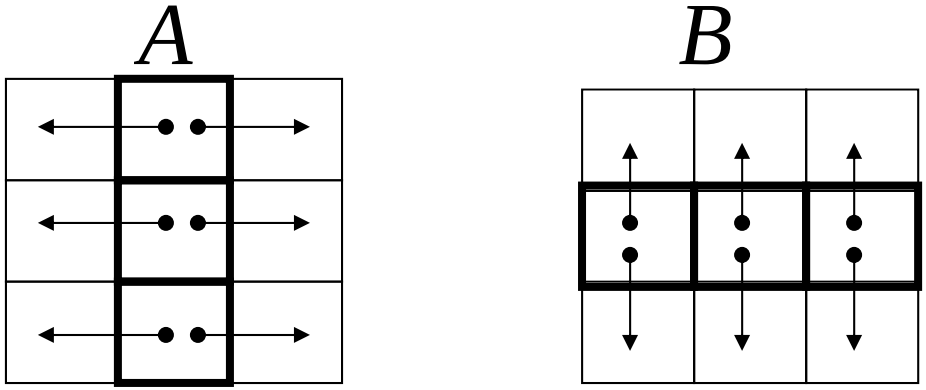
1. SUMMA（Scalable Universal Matrix Multiplication Algorithm）
   - R.Van de Geijinほか、1997年
   - 同時放送（マルチキャスト）のみで実現
2. PUMMA（Parallel Universal Matrix Multiplication Algorithms）
   - Choiほか、1994年
   - 二次元ブロックサイクリック分散方式むきのFoxのアルゴリズム

1.5.4 SUMMA

- アルゴリズムの概略
  - 第一ステップ

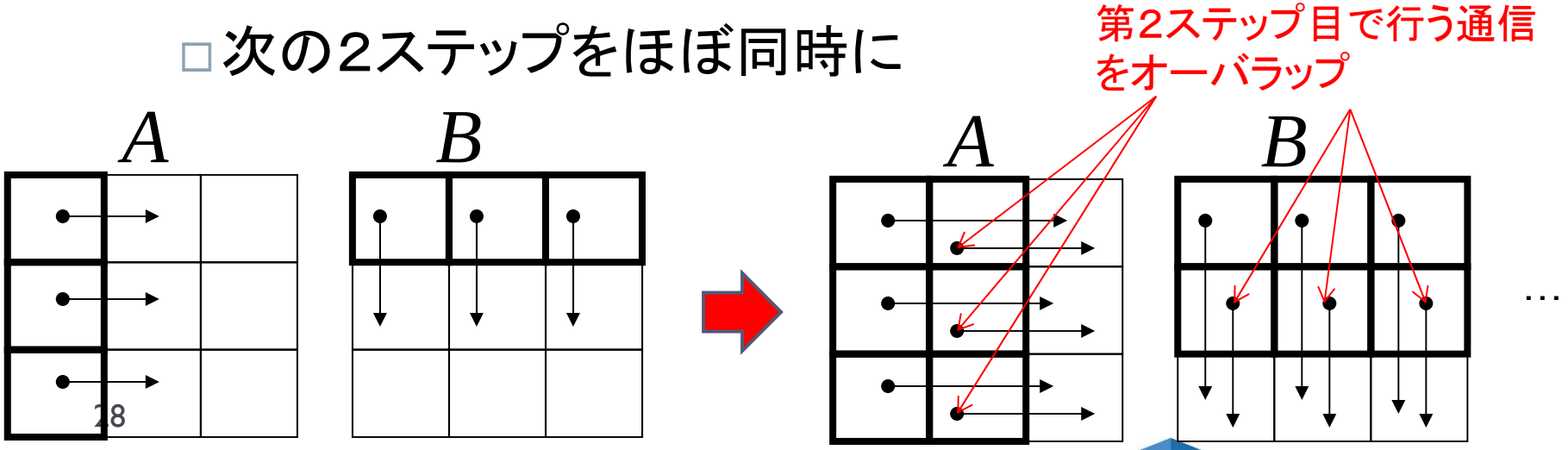


  - 第二ステップ



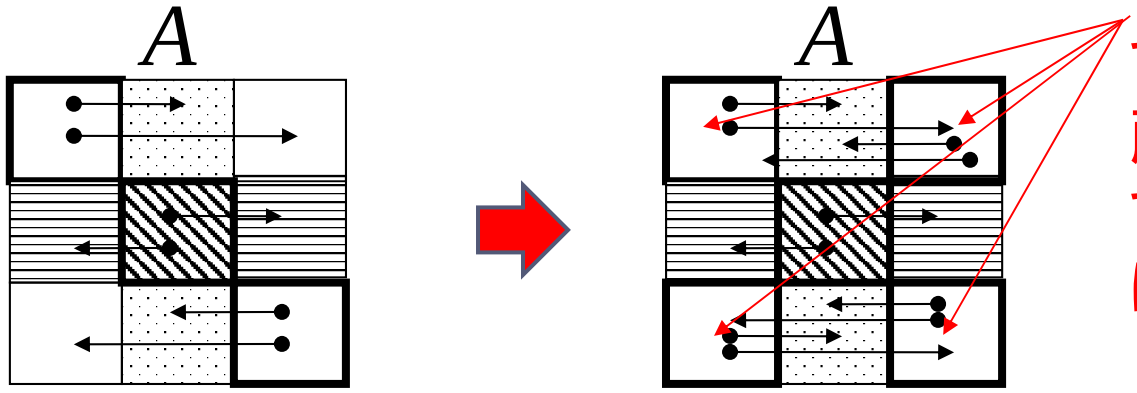
1.5.4 SUMMA

- 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

Execution of sample program (Dense matrix-matrix multiplication)

Note: sample program of dense matrix-matrix multiplication

- Common file name of C/Fortran languages:

  Mat-Mat-fx.tar

- Modify queue name from lecture to lecture6 in job script file mat-mat.bash. Then type "pjsub".
  - lecture : Queue in out of time of this lecture.
  - lecture6 Queue in time of this lecture.

Execute sample program of dense matrix-matrix multiplication

- Type followings in command line:

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

- Choose the follows:

```
$ cd C  : For C language.
$ cd F  : For Fortran language.
```

- The follows are common:

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

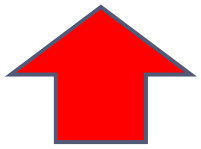
- After finishing the job, type the follow:

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

Output of sample program of dense matrix-matrix multiplication (C Language)

- If the run is successfully ended, you can see the follows:

```
N  = 1000
Mat-Mat time  = 0.209609 [sec.]
 9541.570931 [MFLOPS]
 OK!
```



It is established 9.5GFLOPS with one core.

Output of sample program of dense matrix-matrix multiplication (Fortran Language)

- If the run is successfully ended, you can see the follows:

```
NN  =  1000
Mat-Mat time[sec.] =  0.204734672995982 7
MFLOPS =  9768.741003580422
 OK!
```

It is established 9.7GFLOPS with one core.

Explanation of sample program (C Language)

- You can change size of matrix by the number:

  #define N 1000

- By setting 1 in the follow "0", result of matrix-matrix multiplication is verified:

  #define DEBUG 0

- Specification of MyMatMat function
  - Return result of A times B with size of [N][N] of **double** by setting C with size of [N][N] of **double**.

Explanation of sample program (Fortran Language)

- You can find declaration of size of dimension N in the following file:
  mat-mat.inc

- Variable of the size of dimension is NN, such as:

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

Homework 4

- Parallelize MyMatMat function.
  You can use the following parameter for debugging.
  - #define N 192
  - #define DEBUG 1

- Whole elements of matrices A, B, and C, that are size of N × N, can be allocated in each PE redundantly. (c.f. Strategy of parallelization)

Note: Parallelization

- In this sample program, we use a test matrix with that all elements are set to "1" for A and B. Then we compare theoretical result, that is: all elements of C are N. Please use function of verification for your debug.

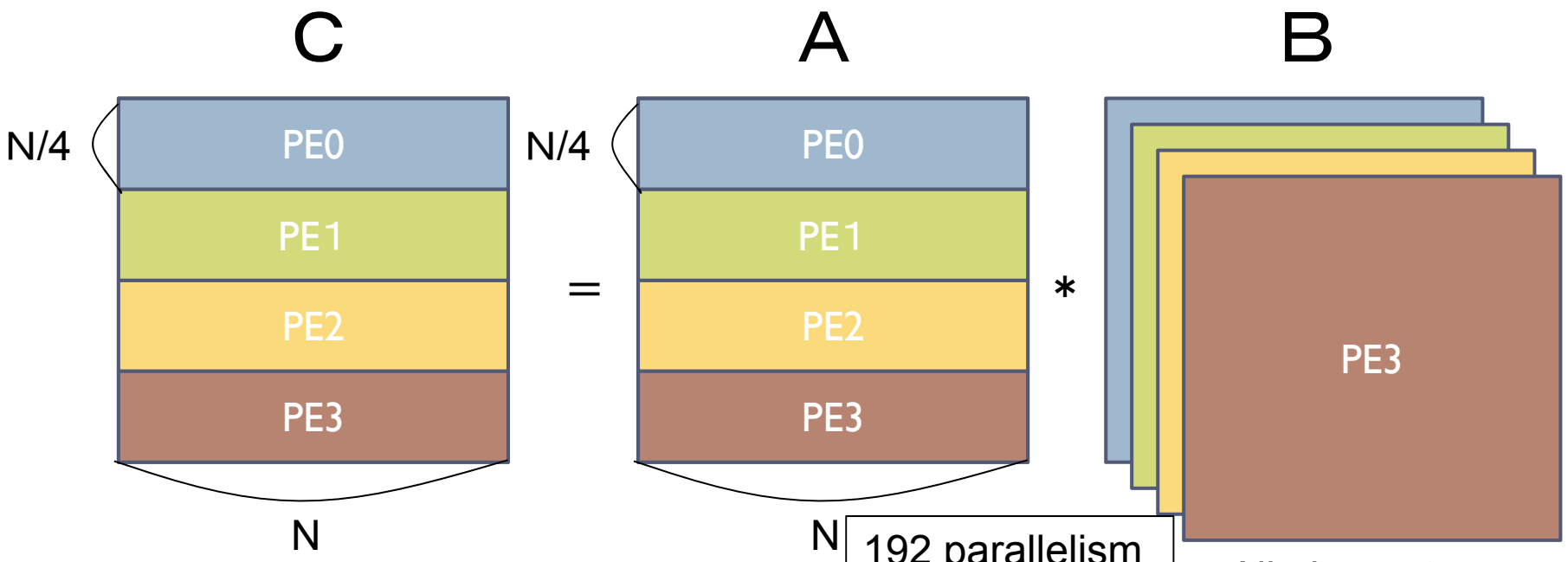
  Note:

  You need also parallelization for the verification routine.

  (c.f. Sample program of matrix-vector multiplication.)

Hints of parallelization

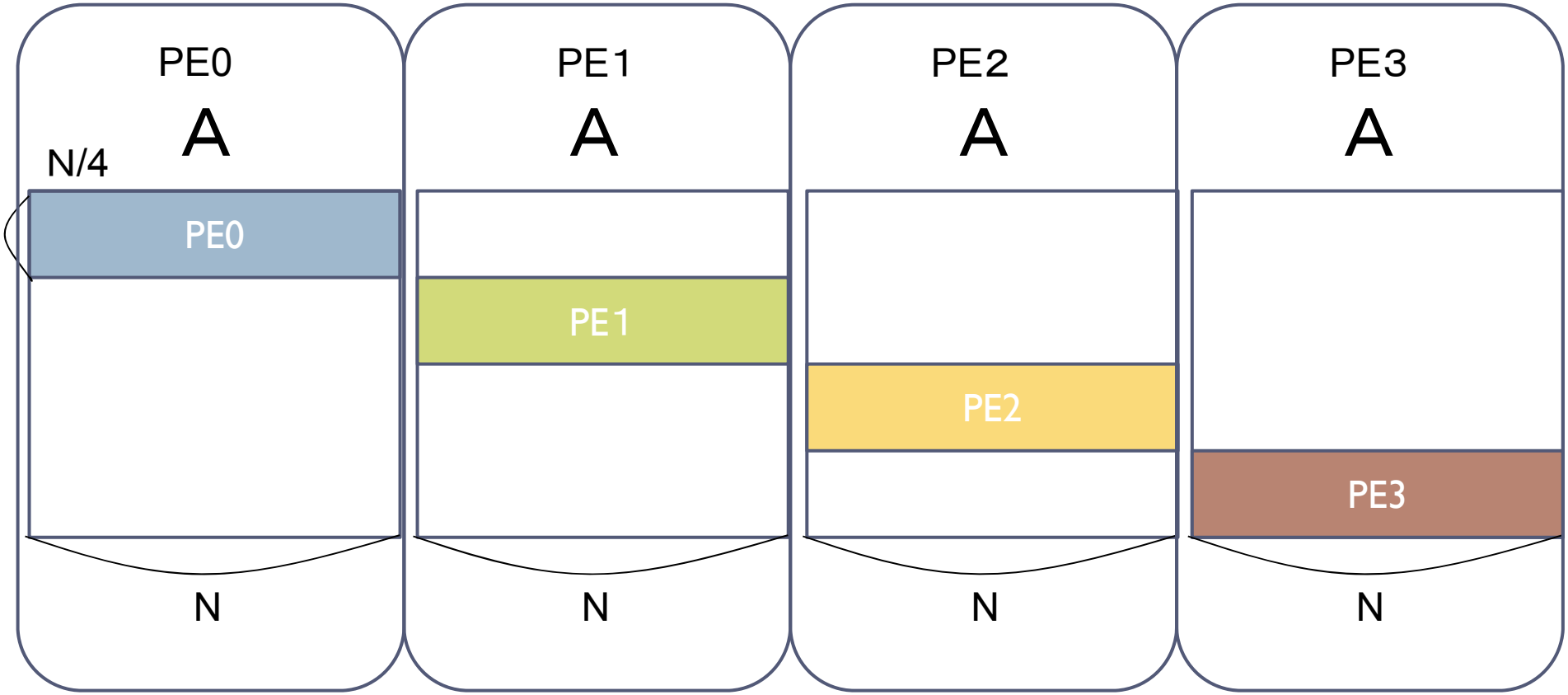
- Use the following data distribution to do easy implementation:



- No communication is needed.
- It can be parallelized as same as matrix-vector multiplication.

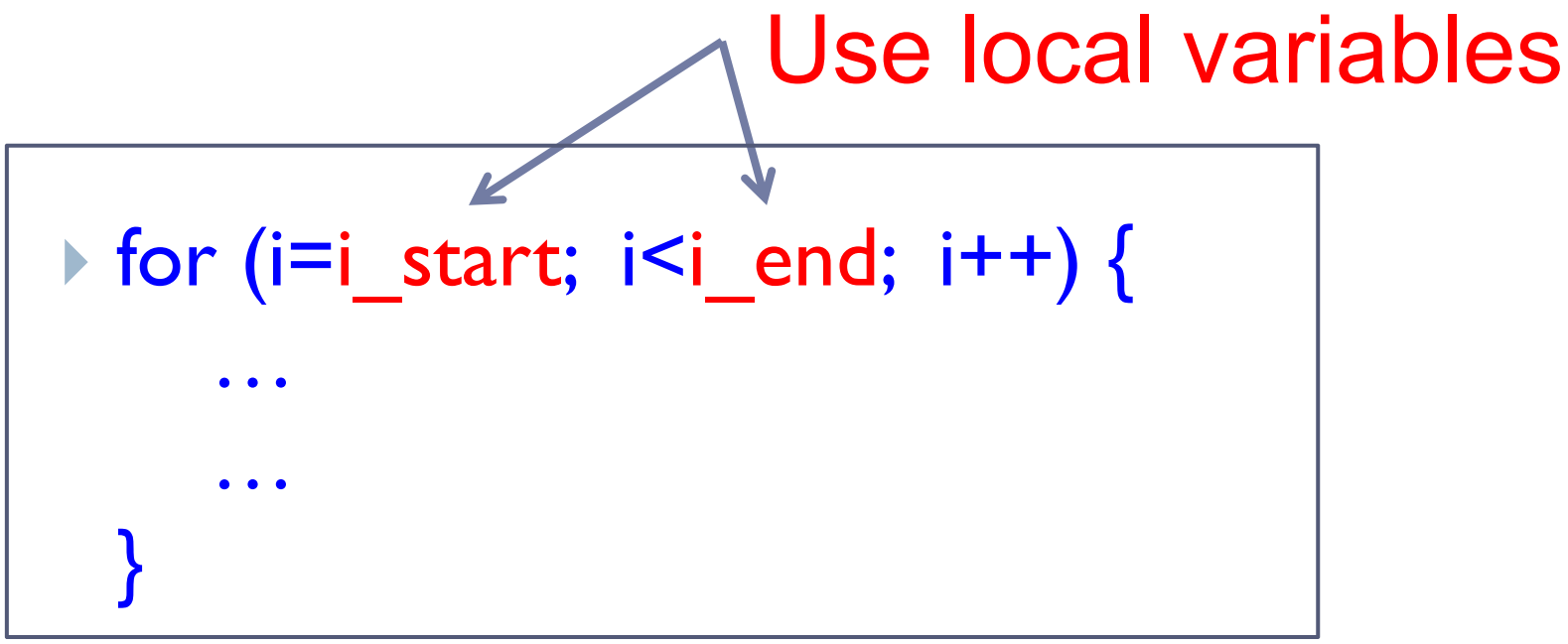
Confirmation: Allocation of array in viewpoint of each PE

- Use "partial" part of arrays in each PE
  although it allocates whole of size of arrays [N][N].



Note: performance issue by implementation

- If you use global variables for loop induction variables, you may obtain poor performance.
- Use it by local variables, or literal values, such as 100.



Homework and Lessons

1. [Homework4] Parallelize sample code of dense matrix-matrix multiplication. You can use redundant allocation of arrays for matrix A, B, and C for initial data distribution.
2. Make a hybrid MPI/OpenMP code, then evaluate its performance by using several executions with respect to MPI processes and OpenMP threads in environments of lecture. Find condition that pure MPI is the fastest by using results of the evaluation.

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。