

「情報システム学特別講義 3」 LU分解の並列化

東京大学情報基盤センター准教授 片桐孝洋

2014年7月8日(火) 14:40－16:10

情報システム学特別講義3

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・ベクトル積の並列化~~

~~9. 6月17日~~

~~● ベキ乗法の並列化~~

~~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

講義の流れ

1. LU分解法

- ▶ ガウス・ジョルダン法
- ▶ ガウス消去法
- ▶ 枢軸選択
- ▶ LU分解法
 - ▶ 外積形式、内積形式、クラウト法、ブロック形式ガウス法、縦ブロックガウス法、前進・後退代入

2. サンプルプログラムの実行

3. 並列化のヒント

4. 実習課題

5. レポート課題

LU分解法の概略

いろいろな変種があります

3 密行列に対する連立一次方程式

▶ 以下の式

$$Ax = b$$

ここで A は実数の密行列 x, b は実数のベクトルとすると、解ベクトル x を求めること。

▶ 解ベクトルを求める方法は、以下の二種類が知られている

1. 直接解法

行列操作により厳密解を求める方法

2. 反復解法

近似解を反復計算で解に収束させ求める方法

3.2 ガウス・ジョルダン法

- ▶ 基本的な消去法により解を求める

- ▶ 第1ステップ

第一行をもとに
係数を消去

$$\begin{array}{l}
 a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
 \boxed{\phantom{a_{21}x_1}} a_{22}'x_2 + \cdots + a_{2n}'x_n = b_2' \\
 \vdots \\
 \boxed{\phantom{a_{n1}x_1}} a_{n2}'x_2 + \cdots + a_{nn}'x_n = b_n'
 \end{array}$$

- ▶ 第2ステップ

第二行をもとに
係数を消去

$$\begin{array}{l}
 a_{11}x_1 + \boxed{\phantom{a_{21}x_1}} + \cdots + a_{1n}''x_n = b_1'' \\
 a_{22}'x_2 + \cdots + a_{2n}''x_n = b_2'' \\
 \vdots \\
 \boxed{\phantom{a_{n1}x_1}} + \cdots + a_{nn}''x_n = b_n''
 \end{array}$$

- ▶ 最終ステップ

$$\begin{array}{l}
 a_{11}x_1 = b_1^* \\
 a_{22}'x_2 = b_2^* \\
 \vdots \\
 a_{nn}''x_n = b_n^*
 \end{array}
 \longrightarrow \text{割り算のみで解を得る}$$

3.2 ガウス・ジョルダン法

- ▶ 右辺 b の代わりに単位行列 I を用意して同様の操作をすれば、最終ステップでは逆行列が求まる
- ▶ 各ステップでの計算量が同じなので、並列化時の負荷バランスが良い

3.3 ガウス消去法

- ▶ 対角線より上の要素をゼロにしない方法

- ▶ 第1ステップ

第一行をもとに
係数を消去

$$\begin{array}{l}
 a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
 \boxed{\phantom{a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2}} \quad a_{22}'x_2 + \cdots + a_{2n}'x_n = b_2' \\
 \vdots \\
 \boxed{\phantom{a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n}} \quad a_{n2}'x_2 + \cdots + a_{nn}'x_n = b_n'
 \end{array}$$

- ▶ 第2ステップ

第二行をもとに
係数を消去

$$\begin{array}{l}
 a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
 \phantom{a_{11}x_1 +} a_{22}''x_2 + \cdots + a_{2n}''x_n = b_2'' \\
 \vdots \\
 \boxed{\phantom{a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n}} + \cdots + a_{nn}''x_n = b_n''
 \end{array}$$

- ▶ 最終ステップ

$$\begin{array}{l}
 a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
 \phantom{a_{11}x_1 +} a_{22}''x_2 + \cdots + a_{2n}''x_n = b_2'' \\
 \phantom{a_{11}x_1 +} \vdots \\
 \phantom{a_{11}x_1 +} a_{nn}^*x_n = b_n^*
 \end{array}$$

この消去を
前進消去 (forward elimination)
とよぶ

3.3 ガウス消去法

- ▶ 前進消去後、最後の項から順に解を求めていく

$$a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n = b_1$$

$$a_{22} x_2 + \cdots + a_{2n} x_n = b_2$$

$$\vdots$$

$$a_{nn} x_n = b_n$$



$$x_n = b_n / a_{nn},$$

$$x_{n-1} = (b_{n-1} - a_{n-1,n} x_n) / a_{n-1,n-1},$$

$$\vdots$$

この代入処理を、後退代入 (backward substitution) とよぶ

3.3 ガウス消去法

- ▶ ガウス消去法は、ガウス・ジョルダン法に比べ、消去演算をする範囲が少ない
(基本行より下のみ)
 - ▶ 演算量が低下する: $n^3 \rightarrow (2/3)n^3$
- ▶ 基本行より下のみ演算するため、並列化するとガウス・ジョルダン法に比べて、負荷バランスの劣化を起こしやすい
 - ▶ 並列処理に向かないと考えた専門家がいた。
 - ▶ 現在はデータ分散の改良や通信の隠蔽技法、ハードウェア能力向上から、ガウス消去法のほうが高速である。

3.3.1 ピボットティング

- ▶ ガウス・ジョルダン法、ガウス消去法とも、基本行の係数がゼロだと、ゼロによる除算が生じ、計算が続行できない

第1行をもとに
係数を消去

$$\begin{array}{l} 0 \longrightarrow a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n = b_1 \\ a_{22}' x_2 + \cdots + a_{2n}' x_n = b_2' \\ \vdots \\ a_{n2}' x_n + \cdots + a_{nn}' x_n = b_n' \end{array}$$

- ▶ これを回避するため、消去する列から最も係数の大きなものを選択して、基本行と入れ替える
(**枢軸選択、ピボットティング、pivot selection**)

3.3.1 ピボットティング

- ▶ ピボットティングには以下の2種の方法がある

1. 完全ピボットティング

更新対象全体から最大のものを選ぶ方法

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = & b_2 \\ & \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = & b_n \end{array}$$

2. 部分ピボットティング

更新対象の列または行から最大のものを選ぶ方式

- ▶ ピボットティングの手間、経験的な数値安定性から部分ピボットティングが用いられることが多い

3.4 LU分解法

- ▶ ガウス消去法のような消去処理を行列演算として定式化
- ▶ 連立一次方程式の行列表記:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

$$\downarrow$$

$$Ax = b$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & \vdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

3.4 LU分解法

- ▶ LU分解法では、以下の3つのステップで解を計算する
- ▶ **第1ステップ**: 行列AのLU分解

$$A = LU, \quad L = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & & \\ & & \ddots & \vdots \\ & & & u_{nn} \end{bmatrix}$$

- ▶ **第2ステップ**: 前進代入

$$\begin{aligned} Ax &= b, \\ (LU)x &= b, \\ L(Ux) &= b \end{aligned} \quad \begin{cases} Lc = b, \\ c = Ux \end{cases} \quad Lc = b : \text{ベクトル } c \text{ を求める}$$

$$\begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- ▶ **第3ステップ**: 後退代入

$$Ux = c$$

: 解ベクトルxを求める

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & & \\ & & \ddots & \vdots \\ & & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

3.4 LU分解法

- ▶ 行列AのLU分解 $A = LU$ には、データアクセスの違いから以下の3種の方法が知られている
 1. 外積形式ガウス法 (outer-product form)
 - ▶ 普通の消去法から導出
 2. 内積形式ガウス法 (inner-product form)
 - ▶ LU分解がなされたとして、Lの対角要素を1に固定して導出
 3. クラウト法 (Crout method)
 - ▶ LU分解がなされたとして、Uの対角要素を1に固定して導出

3.4.1 LU分解法の種類

- ▶ 外積形式 (outer-product form) ガウス法
 - ▶ ガウス消去法と同等の操作でLU分解する
 - ▶ 第 k 列を消去したい場合、

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

...

$$\begin{array}{c} a_{kk}x_k \\ \vdots \\ a_{nk}x_k \end{array} + \cdots + a_{nn}x_n = b_n$$

係数 a_{kk} を用いて $a_{k,k+1}, a_{k,k+2}, \dots, a_{k,n}$ を消去

3.4.1 外積形式ガウス法

- ▶ すなわち列の消去は、
$$a_{ik} - a_{kk}(a_{ik} / a_{kk}), i = k+1, k+2, \dots, n$$
- ▶ これを行列表記にすると、行列Lを

$$L_k = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & l_{k+1,k} & 1 & \\ & & \vdots & & \ddots \\ & & l_{mk} & & & 1 \end{bmatrix}, \quad \begin{aligned} l_{ik} &= -(a_{ik} / a_{kk}), \\ i &= k+1, \dots, n \end{aligned}$$

とすると、この消去は

$$L_k A_k = U_{k+1}$$

3.4.1 外積形式ガウス法

- ▶ 一般的に

$$L_{n-1}L_{n-2} \cdots L_2L_1A = U$$

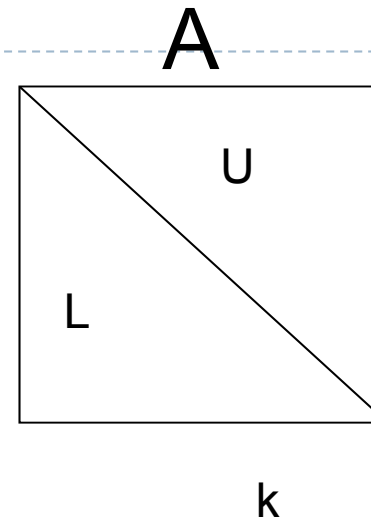
- ▶ したがってLU分解は

$$\begin{aligned} A &= (L_{n-1}L_{n-2} \cdots L_2L_1)^{-1}U \\ &= (L_1^{-1}L_2^{-1} \cdots L_{n-2}^{-1}L_{n-1}^{-1})U \\ &= LU \end{aligned}$$

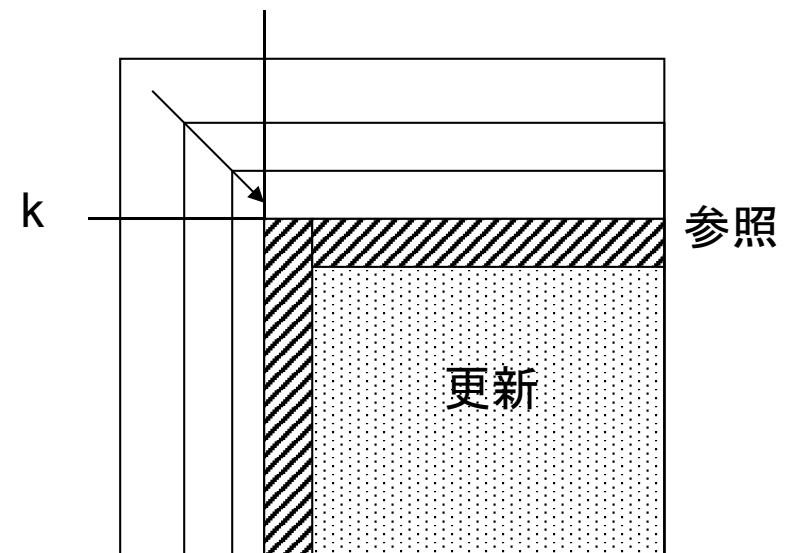
- ▶ ここで、 L_k^{-1} は L_k の要素の符号を反転させたものであり、容易に得られる
 - ▶ 消去作業が終われば行列Lが得られる

3.4.1 外積形式ガウス法 (C言語)

```
for (k=0; k<n; k++) {  
    dtmp = 1.0 / A[k][k];  
    for (i=k+1; i<n; i++) {  
        A[i][k] = A[i][k]*dtmp;  
    }  
    for (j=k+1; j<n; j++) {  
        dakj = A[k][j];  
        for (i=k+1; i<n; i++) {  
            A[i][j] = A[i][j]-A[i][k]*dakj;  
        }  
    }  
}
```

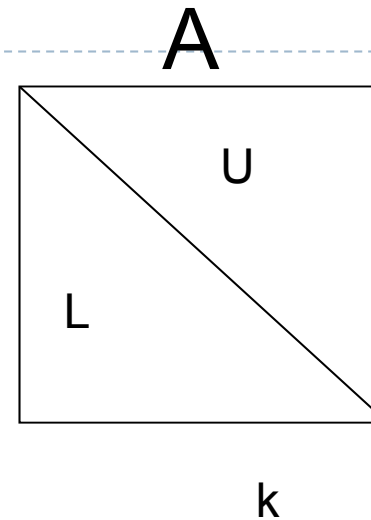


注意:
Lの対角要素は
1であることを仮定
(計算しない)
→Uの対角要素を
入れる

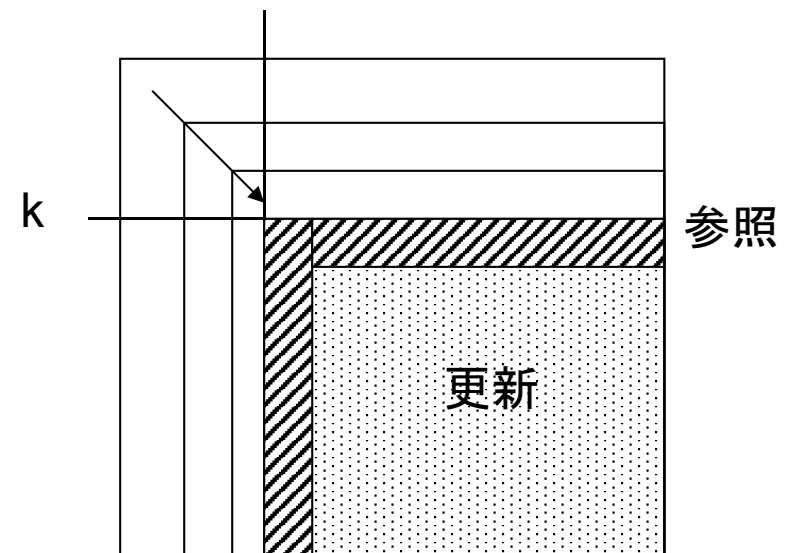


3.4.1 外積形式ガウス法 (Fortran言語)

```
do k=1, n
  dtmp = 1.0d0 / A(k, k)
  do i=k+1, n
    A(i, k) = A(i, k) * dtmp
  enddo
  do j=k+1, n
    dakj = A(k, j)
    do i=k+1, n
      A(i, j) = A(i, j) - A(i, k)*dakj
    enddo
  enddo
enddo
```



注意:
Lの対角要素は
1であることを仮定
(計算しない)
→Uの対角要素を
入れる



3.4.1 外積形式ガウス法

- ▶ 外積形式ガウス法では分解列の右側の領域が更新される
 - ▶ right-lookingアルゴリズムと呼ばれる
- ▶ 外積形式ガウス法は並列化に向く
 - ▶ 処理の中心の更新領域が多い
 - ▶ 負荷バランスよくデータ分散できる
 - ▶ 更新処理が、分解行と分解列という少ないデータを所有するだけで、要素ごとに独立して行える

3.4.1 内積形式ガウス法

▶ 内積形式 (inner-product form) ガウス法

- ▶ LU分解がなされたと仮定した上で、行列Lの対角要素を1として導出した方法

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & 0 \\ \vdots & & \ddots & \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & & \\ & & \ddots & \\ & 0 & & u_{nn} \end{bmatrix}$$

$$a_{11} = u_{11}, \leftarrow u_{11} \text{ が求まる}$$

$$l_{21}u_{11} = a_{21}, l_{31}u_{11} = a_{31}, \dots, l_{n1}u_{11} = a_{n1}$$

$$\nwarrow l_{21} \text{ が求まる}$$

3.4.1 内積形式ガウス法

- ▶ この導出作業を一般化すると、以下の二部分に分かれる

- ▶ (I) u の導出部

- ▶ (II) (I)で得られた値を元に、 L の導出部

- ▶ まとめると

- ▶ (I) $u_{1k} = a_{1k}$

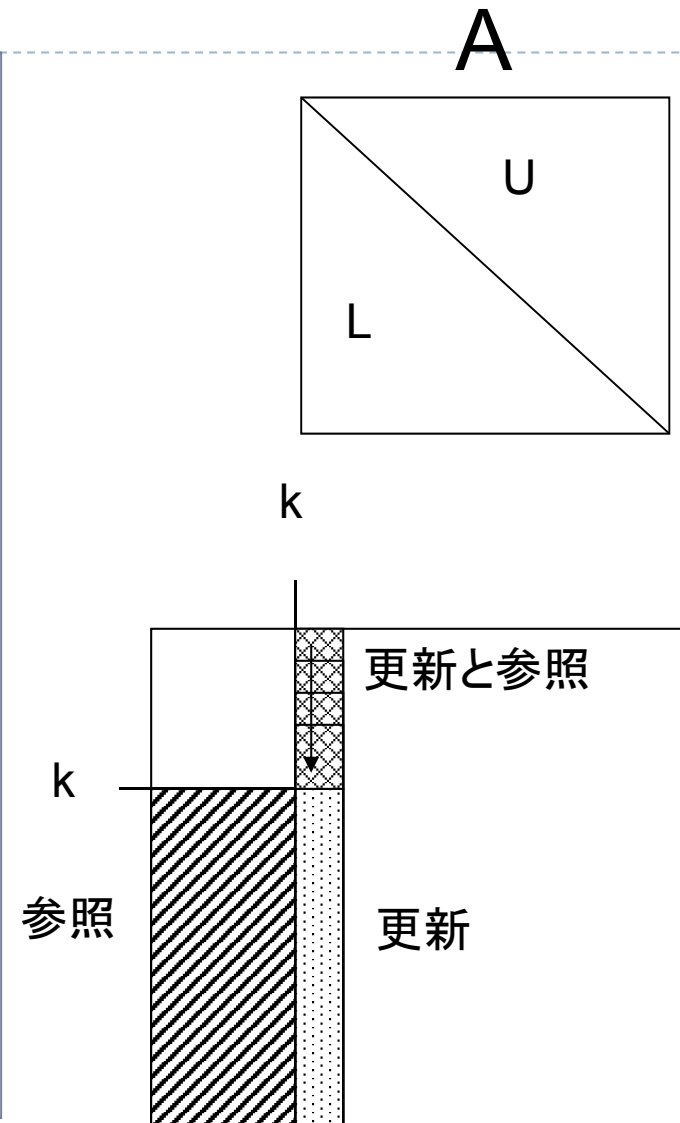
$$u_{ik} = a_{1k} - \sum_{j=1}^{i-1} l_{ij} u_{jk}, (i = 2, 3, \dots, k)$$

- ▶ (II)

$$l_{ik} = (a_{ik} - \sum_{j=1}^{k-1} l_{ij} u_{jk}) / u_{kk}, (i = k + 1, k + 2, \dots, n)$$

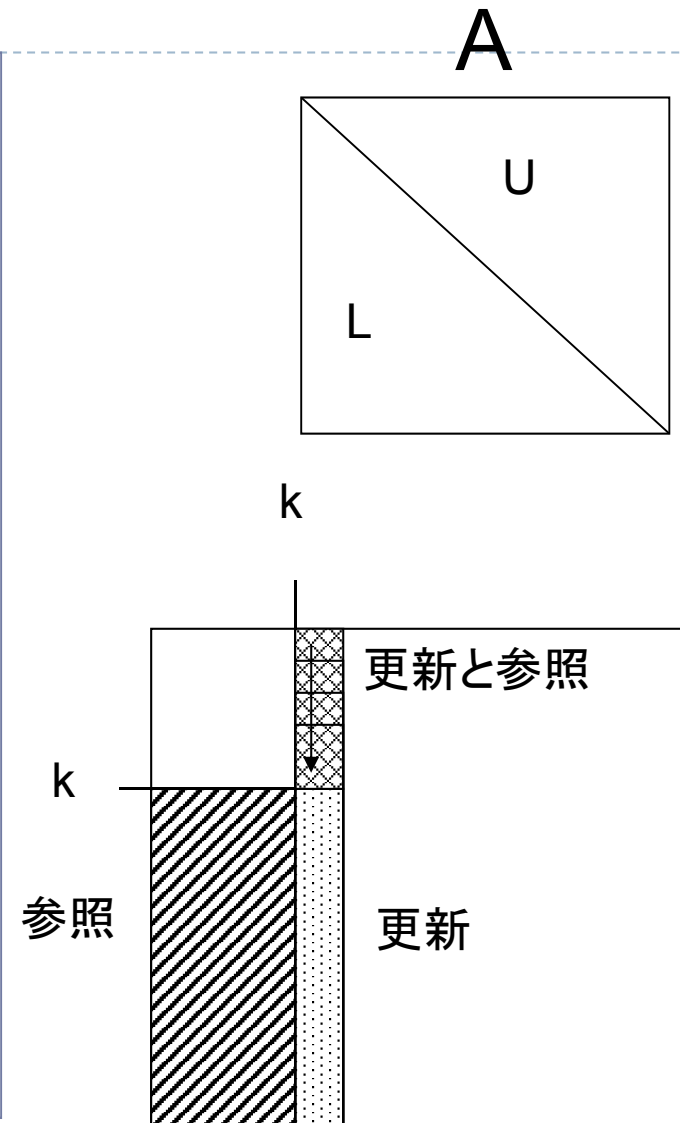
3.4.1 内積形式ガウス法 (C言語)

```
for (k=0; k<n; k++) {  
    for (j=0; j<k; j++) {  
        dajk = A[j][k];  
        for (i=j+1; i<n; i++) {  
            A[i][k] = A[i][k] - A[i][j]*dajk;  
        }  
    }  
    A[k][k] = 1.0 / A[k][k];  
    for (i=k+1; i<n; i++) {  
        A[i][k] = A[i][k]*A[k][k];  
    }  
}
```



3.4.1 内積形式ガウス法 (Fortran言語)

```
do k=1, n
  do j=1, k
    dajk = A(j, k)
    do i=j+1, n
      A(i, k) = A(i, k) - A(i, j) * dajk;
    enddo
  enddo
  A(k, k) = 1.0d0 / A(k, k)
  do i=k+1, n
    A(i, k) = A(i, k) * A(k, k)
  enddo
enddo
```



3.4.1 内積形式ガウス法

- ▶ 内積形式ガウス法では、分解列の左側の領域が主に参照される
 - ▶ left-lookingアルゴリズムと呼ばれる
- ▶ 内積形式ガウス法の並列化
 - ▶ 行列Aを列方向分散(*, Cyclic)
 - ▶ 参照領域のデータがないので、通信多発（ベクトルリダクションが毎回必要）
 - ▶ 行列Aを行方向分散(Cyclic, *)
 - ▶ 上三角行列Uの要素（データ数が少ない）を所有すれば、独立して計算可能

3.4.1 クラウト法

▶ クラウト法 (Clout Method)

- ▶ LU分解がなされたと仮定した上で、行列Uの対角要素を1として導出した方法 (cf. 内積形式ガウス法)

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & 0 \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & \cdots & u_{1n} \\ & 1 & & \\ & 0 & \ddots & \vdots \\ & & & 1 \end{bmatrix}$$

$$l_{11} = a_{11}, \quad l_{21} = a_{21}, \quad l_{n1} = a_{n1} \quad \leftarrow \begin{array}{l} \text{Iの第1列が} \\ \text{求まる} \end{array}$$

$$l_{11}u_{12} = a_{12}, \quad l_{11}u_{13} = a_{13}, \dots, \quad l_{11}u_{1n} = a_{1n}$$

3.4.1 クラウト法

- ▶ この計算を一般化すると、
 - ▶ Lの第k列を求める場合

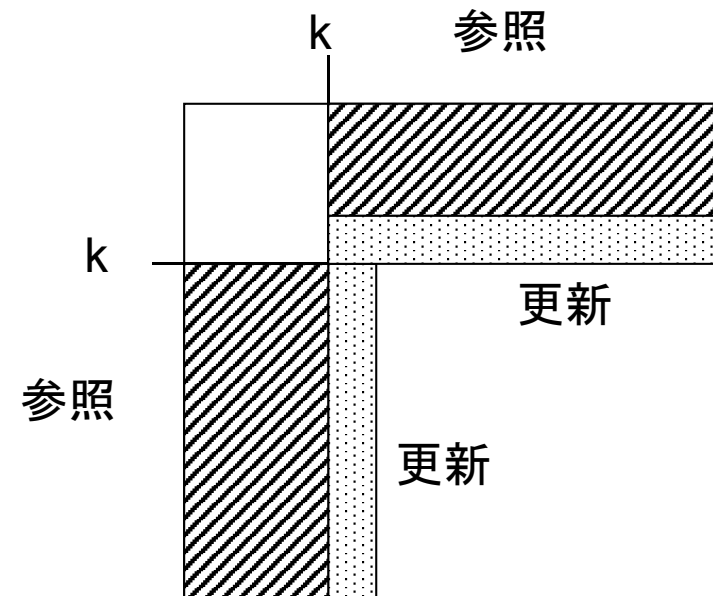
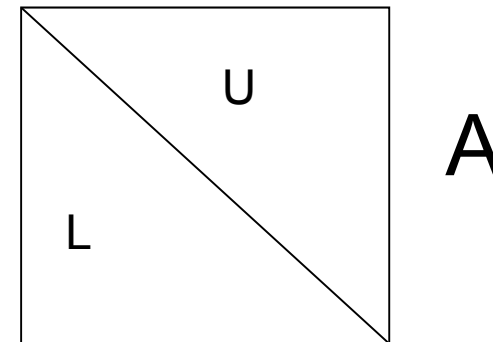
$$l_{ik} = a_{ik} - \sum_{j=1}^{k-1} l_{ij} u_{jk}, \quad (i = k, k+1, \dots, n)$$

- ▶ Uの第k行を求める場合

$$u_{kj} = (a_{kj} - \sum_{i=1}^{k-1} l_{ki} u_{ij}) / l_{kk}, \quad (j = k, k+1, \dots, n)$$

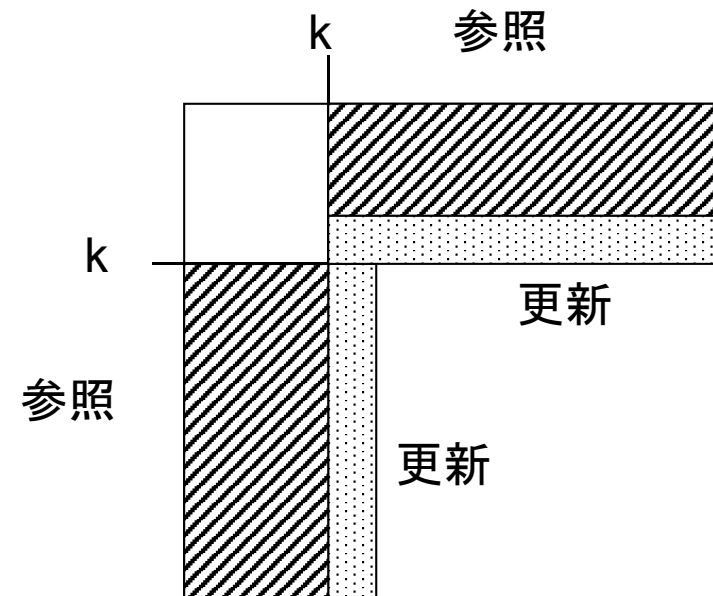
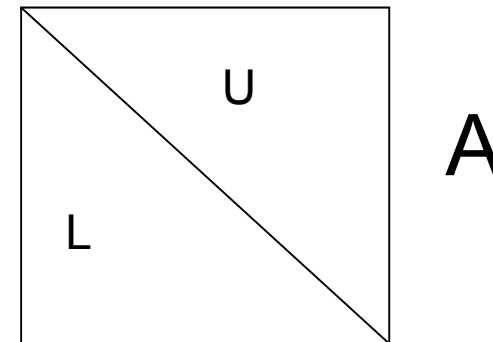
3.4.1 クラウト法 (C言語)

```
A[0][0]=1.0/A[0][0];
for (j=1; j<n; j++) {
    A[0][j]=A[0][j]*A[0][0]; }
for (k=0; k<n; k++) {
    for (j=0; j<k; j++) {
        dajk=A[j][k];
        for (i=k; i<n; i++) {
            A[i][k]=A[i][k]-A[i][j]*dajk;
        } }
    A[k][k]=1.0/A[k][k];
    for (i=0; i<k; i++) {
        daki=A[k][i];
        for (j=k+1; j<n; j++) {
            A[k][j]=A[k][j]-daki*A[i][j];
        } }
    for (j=k+1; j<n; j++) {
        A[k][j]=A[k][j]*A[k][k]; }
    }
```



3.4.1 クラウト法 (Fortran言語)

```
A(1,1)=1.0d0/A(1,1)
do j=2, n
  A(1,j) =A(1,j) * A(1, 1) enddo
do k=1, n
  do j=1, k
    dajk=A(j, k)
    do i=k, n
      A(i, k)=A(i, k) - A(i, j) * dajk
    enddo; enddo
  A(k, k) =1.0d0 / A(k, k)
  do i=1, k
    daki=A(k, i)
    do j=k+1, n
      A(k, j)=A(k, j) - daki * A(i, j)
    enddo; enddo
  do j=k+1, n
    A(k, j)=A(k, j) * A(k, k) enddo
  enddo
```



3.4.1 クラウト法

- ▶ クラウト法では、最内ループの交換ができる
 - ▶ 長さ $(1 \sim k-1)$ のループ、長さ $(k-n)$ のループの内、最も長いループを最内に移動可
 - ▶ ベクトル計算機で実行性能が良い
- ▶ 分解列および分解行の外側に2つの参照領域
 - ▶ 分散メモリ型並列計算機での実装が困難
 - ∴どのようにデータ分割しても大量通信発生
- ▶ 共有メモリ型並列計算機では並列化可能
 - ∴参照領域があれば分解列と分解行は独立に計算可能

3.4.1 ブロック形式ガウス法

- ▶ 行列Aを小行列に分解し、その小行列単位でLU分解する方法。**LU分解と行列-行列積で実現できる。**
- ▶ 具体的には（各小行列を各PEが所有）

$$\begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} & \tilde{A}_{13} \\ \tilde{A}_{21} & \tilde{A}_{22} & \tilde{A}_{23} \\ \tilde{A}_{31} & \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix} = \begin{bmatrix} \tilde{L}_{11} & & 0 \\ \tilde{L}_{21} & \tilde{L}_{22} & \\ \tilde{L}_{31} & \tilde{L}_{32} & \tilde{L}_{33} \end{bmatrix} \begin{bmatrix} \tilde{U}_{11} & \tilde{U}_{12} & \tilde{U}_{13} \\ & \tilde{U}_{22} & \tilde{U}_{23} \\ 0 & & \tilde{U}_{33} \end{bmatrix}$$

とすると、右辺は

$$\tilde{A}_{11} = \tilde{L}_{11} \tilde{U}_{11}, \tilde{A}_{12} = \tilde{L}_{11} \tilde{U}_{12}, \quad \tilde{A}_{13} = \tilde{L}_{11} \tilde{U}_{13},$$

$$\tilde{A}_{21} = \tilde{L}_{21} \tilde{U}_{11}, \tilde{A}_{22} = \tilde{L}_{21} \tilde{U}_{12} + \tilde{L}_{22} \tilde{U}_{22}, \tilde{A}_{23} = \tilde{L}_{21} \tilde{U}_{13} + \tilde{L}_{22} \tilde{U}_{23},$$

$$\tilde{A}_{31} = \tilde{L}_{31} \tilde{U}_{11}, \tilde{A}_{32} = \tilde{L}_{31} \tilde{U}_{12} + \tilde{L}_{32} \tilde{U}_{22}, \tilde{A}_{33} = \tilde{L}_{31} \tilde{U}_{13} + \tilde{L}_{32} \tilde{U}_{23} + \tilde{L}_{33} \tilde{U}_{33}$$

3.4.1 ブロック形式ガウス法

▶ 第1ステップ LU分解

$$\begin{aligned} \tilde{A}_{11} &= \tilde{L}_{11} \tilde{U}_{11}, \tilde{A}_{12} = \tilde{L}_{11} \tilde{U}_{12}, & \tilde{A}_{13} &= \tilde{L}_{11} \tilde{U}_{13}, \\ \tilde{A}_{21} &= \tilde{L}_{21} \tilde{U}_{11}, \tilde{A}_{22} = \tilde{L}_{21} \tilde{U}_{12} + \tilde{L}_{22} \tilde{U}_{22}, \tilde{A}_{23} = \tilde{L}_{21} \tilde{U}_{13} + \tilde{L}_{22} \tilde{U}_{23}, \\ \tilde{A}_{31} &= \tilde{L}_{31} \tilde{U}_{11}, \tilde{A}_{32} = \tilde{L}_{31} \tilde{U}_{12} + \tilde{L}_{32} \tilde{U}_{22}, \tilde{A}_{33} = \tilde{L}_{31} \tilde{U}_{13} + \tilde{L}_{32} \tilde{U}_{23} + \tilde{L}_{33} \tilde{U}_{33} \end{aligned}$$

▶ 第2ステップ

$$\begin{aligned} \tilde{A}_{11} &= \tilde{L}_{11} \tilde{U}_{11}, \tilde{A}_{12} = \tilde{L}_{11} \tilde{U}_{12}, & \tilde{A}_{13} &= \tilde{L}_{11} \tilde{U}_{13}, \\ \tilde{A}_{21} &= \tilde{L}_{21} \tilde{U}_{11}, \tilde{A}_{22} = \tilde{L}_{21} \tilde{U}_{12} + \tilde{L}_{22} \tilde{U}_{22}, \tilde{A}_{23} = \tilde{L}_{21} \tilde{U}_{13} + \tilde{L}_{22} \tilde{U}_{23}, \\ \tilde{A}_{31} &= \tilde{L}_{31} \tilde{U}_{11}, \tilde{A}_{32} = \tilde{L}_{31} \tilde{U}_{12} + \tilde{L}_{32} \tilde{U}_{22}, \tilde{A}_{33} = \tilde{L}_{31} \tilde{U}_{13} + \tilde{L}_{32} \tilde{U}_{23} + \tilde{L}_{33} \tilde{U}_{33} \end{aligned}$$

→ L_{11} を転送、 U_{1*} を計算

→ U_{11} を転送、 L_{*1} を計算

▶ 第3ステップ

$$\begin{aligned} \tilde{A}_{11} &= \tilde{L}_{11} \tilde{U}_{11}, \tilde{A}_{12} = \tilde{L}_{11} \tilde{U}_{12}, & \tilde{A}_{13} &= \tilde{L}_{11} \tilde{U}_{13}, \\ \tilde{A}_{21} &= \tilde{L}_{21} \tilde{U}_{11}, \tilde{A}_{22} = \tilde{L}_{21} \tilde{U}_{12} + \tilde{L}_{22} \tilde{U}_{22}, \tilde{A}_{23} = \tilde{L}_{21} \tilde{U}_{13} + \tilde{L}_{22} \tilde{U}_{23}, \\ \tilde{A}_{31} &= \tilde{L}_{31} \tilde{U}_{11}, \tilde{A}_{32} = \tilde{L}_{31} \tilde{U}_{12} + \tilde{L}_{32} \tilde{U}_{22}, \tilde{A}_{33} = \tilde{L}_{31} \tilde{U}_{13} + \tilde{L}_{32} \tilde{U}_{23} + \tilde{L}_{33} \tilde{U}_{33} \end{aligned}$$

→ L_{21} を転送

→ U_{12} を転送 → U_{13} を転送 → L_{31} を転送

3.4.1 ブロック形式ガウス法

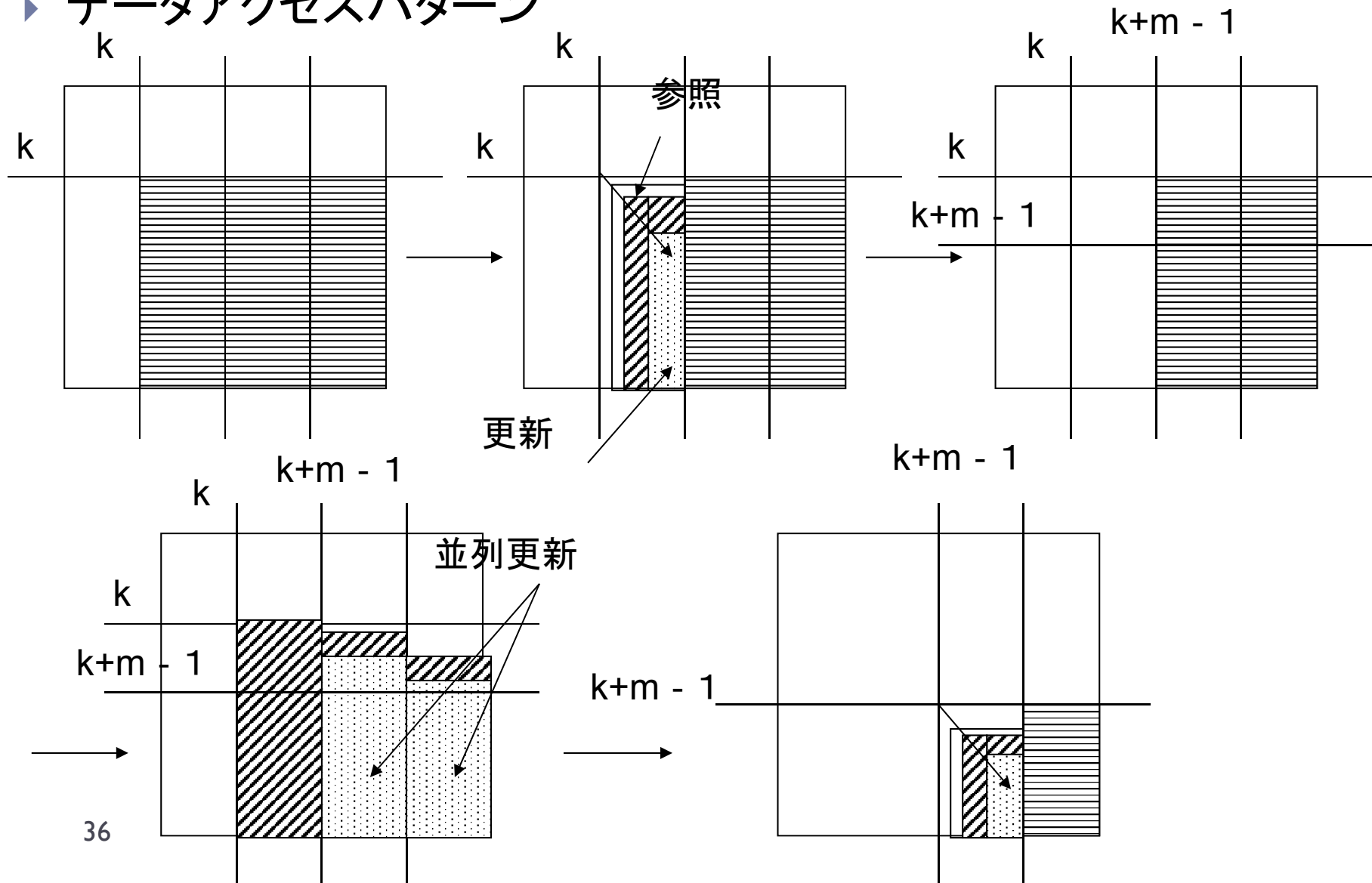
- ▶ 対角要素がLU分解して、行方向、列方向に部分的なLU分解を転送する。
- ▶ ブロック形式ガウス法の実現法は二通りある
 1. 実際に小行列L、Uの逆行列を求める方法
例) $L_{21} = A_{21} U_{11}^{-1}$
 2. 逆行列を求めず、LU分解を用いる方法
例) $A_{21} = L_{21} U_{11}$
- ▶ 1の実装の場合、行列-行列積が主演算となる
 - ▶ 高効率で実装可能

3.4.2 縦ブロックガウス法

- ▶ 縦ブロックガウス法は、列方向のみデータを分割する方法 (cf. ブロック形式ガウス法)
- ▶ 並列化した場合、PE内に列データを全て所有しているため、ピボット処理が実装しやすい
 - ▶ ブロック形式ガウス法は実装が難しい
- ▶ 外積形式ガウス法の並列化に比べ
 1. 通信回数の削減
 2. ループアンローリングによる性能向上が期待できる

3.4.2 縦ブロックガウス法

▶ データアクセスパターン



3.4.2 縦ブロックガウス法

- ▶ 縦ブロックガウス法は、ある幅ごとにLU分解を行う
 - ▶ この幅のことをブロック幅とよぶ
 - ▶ ブロック幅を用いて設計されたアルゴリズムを一般的にブロック化アルゴリズムとよぶ
- ▶ ブロック化をすることで、演算カーネルが2重ループ(レベル2 BLAS)から、3重ループ(レベル3 BLAS3演算)になる
 - ▶ 実装による性能向上が得られやすい

3.4.2 縦ブロックガウス法 (C言語)

▶ 実際のカーネル部分

```
▶ for (jm=k; jm<k+m; jm++) {  
    for (j=k+m; j<n; j++) {  
        dakj = A[jm][j];  
        for (i=jm+1; i<n; i++) {  
            A[i][j]=A[i][j] - A[i][jm]*dakj;  
        }  
    }  
}
```

▶ ループ jm, j, i についてループの展開 (ループアンローリング) 可能

3.4.2 縦ブロックガウス法 (C言語)

▶ jmについて2段のアンローリング

```
▶ for (jm=k; jm<k+m; jm+=2) {  
    for (j=k+jm; j<n; j++) {  
        dakj0 = A[jm][j];  
        dakj1 = A[jm+1][j];  
        for (i=jm+1; i<n; i++) {  
            A[i][j]=A[i][j] - A[i][jm]*dakj0  
                        - A[i][jm+1]*dakj1;  
        }  
    }  
}
```

3.4.2 縦ブロックガウス法 (C言語)

▶ さらにjについても、2段のアンローリング

```
▶ for (jm=k; jm<k+m; jm+=2) {  
    for (j=k+m; j<n; j+=2) {  
        dakj00 = A[jm][j];  
        dakj10 = A[jm+1][j];  
        dakj01 = A[jm][j+1];  
        dakj11 = A[jm+1][j+1];  
        for (i=jm+1; i<n; i++) {  
            A[i][j] = A[i][j] - A[i][jm]*dakj00  
                      - A[i][jm+1]*dakj10;  
            A[i][j+1] = A[i][j+1] - A[i][jm]*dakj01  
                          - A[i][jm+1]*dakj11;  
        }  
    }  
}
```

- ▶ この処理は、ループ内で2段2列分の消去を同時にしているともみなせる (多段多列同時消去法)

3.4.2 縦ブロックガウス法 (Fortran言語)

▶ 実際のカーネル部分

```
▶ do jm=k, k+m  
    do j=k+m+1, n  
        dakj = A(jm, j)  
        do i=jm + 1, n  
            A (i, j) = A(i, j) - A(i, jm) * dakj  
        enddo  
    enddo  
enddo
```

▶ ループ jm, j, i についてループの展開 (ループアンローリング)可能

3.4.2 縦ブロックガウス法 (Fortran言語)

▶ jmについて2段のアンローリング

```
▶ do jm=k, k+m-1, 2
    do j=k+m, n
        dakj0 = A(jm, j)
        dakj1 = A(jm+1, j)
        do i=jm+1, n
            A(i, j) = A(i, j) - A(i, jm) * dakj0
            &          - A(i, jm+1) * dakj1
        enddo
    enddo
enddo
```

3.4.2 縦ブロックガウス法 (Fortran言語)

- ▶ さらにjについても、2段のアンローリング

```
▶ do jm=k, k+m-1, 2
  do j=k+m, n, 2
    dakj00 = A(jm, j)
    dakj10 = A(jm+1, j)
    dakj01 = A(jm, j+1)
    dakj11 = A(jm+1, j+1)
    do i=jm+1, n
      A(i, j) = A(i, j) - A(i, jm) * dakj00
      &          - A(i, jm+1) * dakj10
      A(i, j+1) = A(i, j+1) - A(i, jm) * dakj01
      &          - A(i, jm+1) * dakj11
    enddo; enddo; enddo
```

- ▶ この処理は、ループ内で2段2列分の消去を同時にしているとみなせる (多段多列同時消去法)

3.4.2 縦ブロックガウス法

- ▶ ブロック化するとできる通信隠蔽
- ▶ 縦ブロックガウス法において、データを列方向ブロックサイクリック分散 ($*$, Cyclic(m))するだけで実現可能

- ▶ **LU分解が必要なブロックを所有するPE**

1. 優先してLU分解を行い結果を放送
2. その他の行列更新を行う

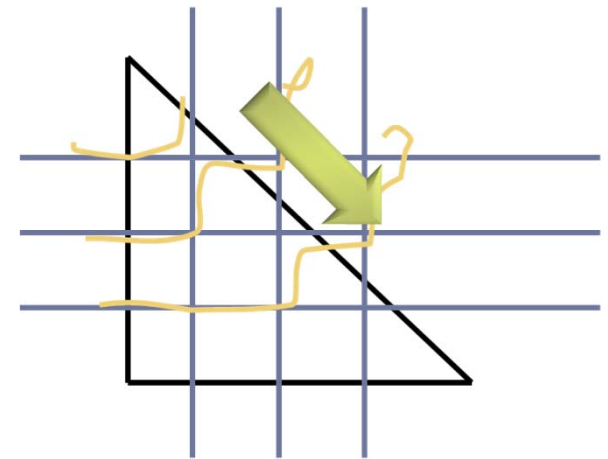
- ▶ **その他のPE**

1. LU分解データ受信待ち
2. 行列更新

通信と計算の
オーバーラップ
→通信時間隠蔽

3.4.3 代入計算

- ▶ 行列Aを固定、右辺bを変えて計算する場合は前進代入、後退代入を並列化する必要がある
- ▶ **結論**: データ分散により、処理パターンは異なるが並列化可能
- ▶ **列方向分散方式** (* , Block) など
 - ▶ ウェーブフロント処理で並列化
- ▶ **行方向分散方式** (Block , *) など
 - ▶ 列単位で並列性 (放送処理が必要)



サンプルプログラムの実行 (LU分解法)

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

LU分解法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/LU-fx.tar ./
$ tar xvf LU-fx.tar
$ cd LU
```
- ▶ 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 lu.bash
```
- ▶ 実行が終了したら、以下を実行する

```
$ cat lu.bash.oXXXXXXX
```

LU分解法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 = 192

LU solve time = 0.004611 [sec.]

1051.432427 [MFLOPS]

Pass value: 3.017485e-07

Calculated value: 2.232057e-10

OK! Test is passed.

LU分解法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 = 192

LU solve time[sec.] = 4.647028981707990E-03

MFLOPS = 1043.219661224964

Pass value: 3.017485141754150E-07

Calculated value: 1.742616051458867E-10

OK! Test is passed.

Fortran言語のサンプルプログラムの注意

- ▶ 行列サイズN(および、プロセッサ数 NPROCS)の宣言は、以下のファイルにあります。

lu.inc

- ▶ 行列サイズ変数が、NNとなっています。

integer NN

parameter (NN=192)

サンプルプログラムの説明

▶ #define N 192

- ▶ 数字を変更すると、行列サイズが変更できます

▶ #define MATRIX 1

- ▶ 生成行列の種類の指定です
- ▶ 「1」にすると、枢軸選択なしでも解ける行列を設定します
- ▶ 「1以外」にすると、乱数で行列を設定します。

この行列を解くには、枢軸選択処理が必要です。
(サンプルプログラムでは解けません)

▶ 解の検査方法

- ▶ 解ベクトル x が1ベクトルとなるように、 $Ax=b$ の右辺 b を計算して設定しています。
- ▶ 残差ベクトルの2ノルムが、 $|A|*N$ より大きくなるとエラーです。

サンプルプログラムの説明

▶ MyLUSolve関数の仕様

- ▶ double型の密行列Aと、右辺ベクトルbを入力とします。
- ▶ LU分解を用いて $Ax=b$ を求解し、解ベクトルxを出力します。
- ▶ LU分解のアルゴリズムは外積形式(right-looking)です。

▶ その他

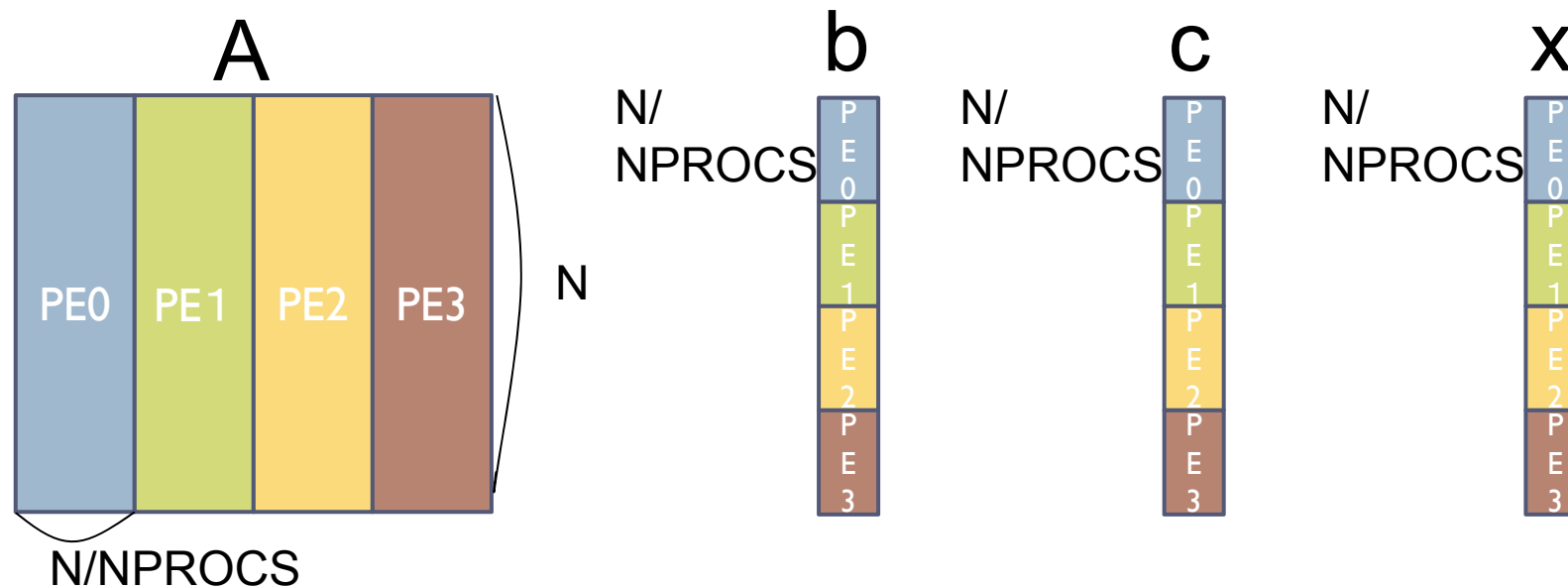
- ▶ N=192の時の、LU分解後の行列Aの値、およびベクトルcの値(C言語のもの)が、ファイル luAc.dat にあります。
デバックに活用してください。

演習課題

- **MyLUSolve関数を並列化してください。**
 - ▶ 中級以上のレベルであり、簡単ではありません。
 - ▶ とりあえず $N=192$ で並列化してください。
 - ▶ できれば $N=192$ 以上の大きな値にして実行してください。
 - ▶ $N=192$ で動いても、 $N=384$ で動かなくなることがあります。
これは、おそらく、前進代入か、前進消去部分が間違っています。
 - ▶ 何が問題か分からなくなった時は、
 1. LU分解後のAの値を表示、OKなら
 2. ベクトルcの値を表示、OKなら
 3. ベクトルxの値を表示
- というように、段階を経て部分を特定し、地道にデバックしてください。
これは、並列プログラミングの鉄則です。

並列化のヒント：データ分散方式

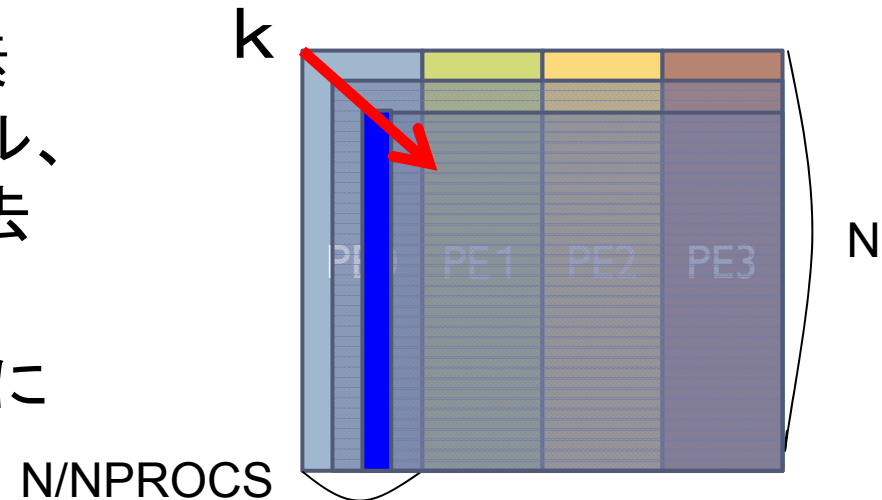
- ▶ 行列A、およびベクトルb、c、xの計算担当領域は以下のようになります。（それぞれ各PEで重複して持ちます）
（ただし以下は4PEの場合で、実習環境は192PEです。）



- ▶ 1対1通信関数(MPI_Send, MPI_Recv)のみで実装できます。
- ▶ 受信用バッファ(buf[N])が必要です。

並列化のヒント：LU分解部分

- ▶ LU分解部分は、行列Aに関して、最外のk-ループが1つつ変動し消去部分が1つつ小さくなっていきます。
- ▶ 現在のkにおいて、対角要素から1行(右図の青いベクトル、**枢軸ベクトル**と呼ぶ)は、消去に必要な情報です。
- ▶ 枢軸ベクトルなしでは、並列に消去できません。
- ▶ 以上から、並列化する際、以下を考慮する必要があります。
 1. 対角要素を持っているPE番号をどう計算するか
 2. 対角要素を持っているPEは、**担当範囲が1つ小さくなる**
 3. 対角要素を持っているPEは、枢軸ベクトルを放送する。
(その他のPEは受け取る。)

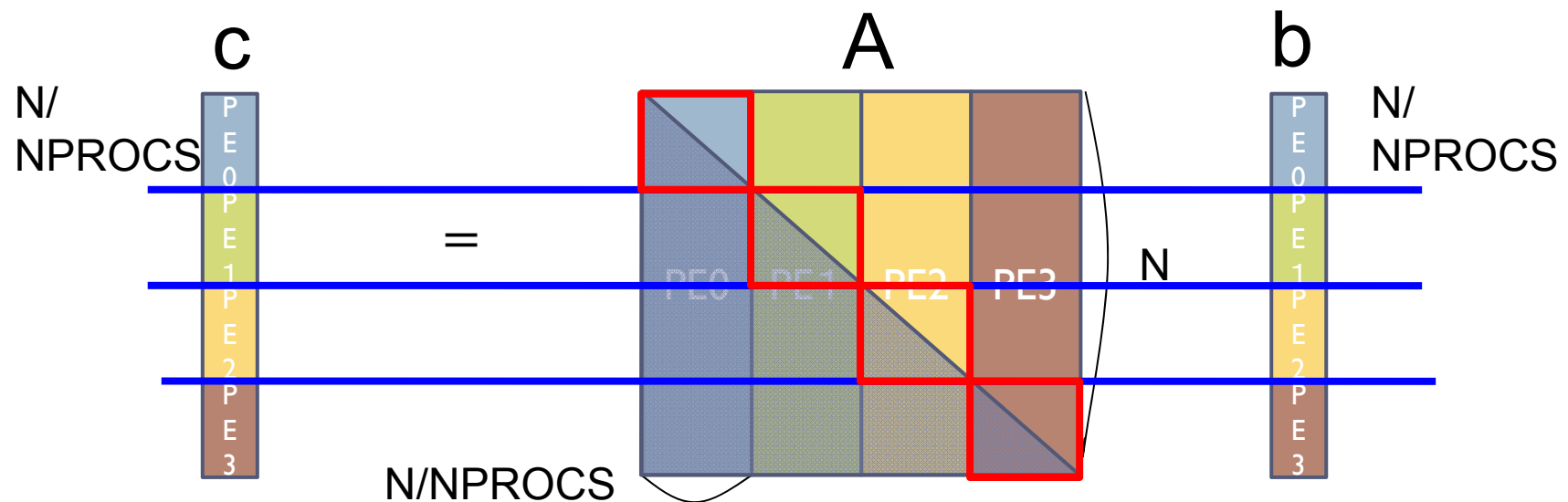


並列化の道具

- ▶ 対角要素を持っているPE番号は、 $(*, \text{BLOCK})$ 分割方式の場合で、かつ k -ループ(k 行目)の場合、以下のようになる.
 - ▶ k / ib ,
ここで, $ib = n / \text{numprocs}$;
- ▶ 枢軸ベクトルを放送する相手は、自分のPE番号より大きく、 $\text{numprocs} - 1$ 番までのPEである。

並列化のヒント：前進代入部分

- ▶ 前進代入部分は、**このデータ分散方法では**、対角ブロック部分に相当するベクトル c の要素すべて決定し、その後、対角ブロックに相当するベクトル c が各PEで参照されます。
- ▶ 対角ブロック部分の値が決定しないと、次の処理に進めません。

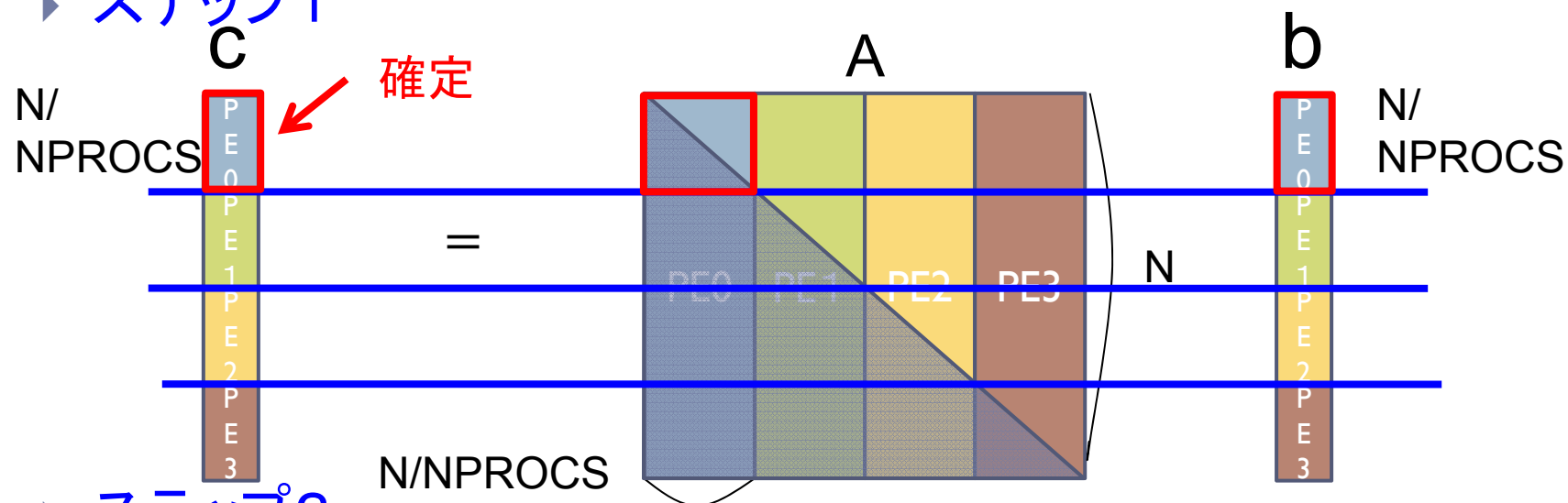


並列化のヒント：前進代入部分

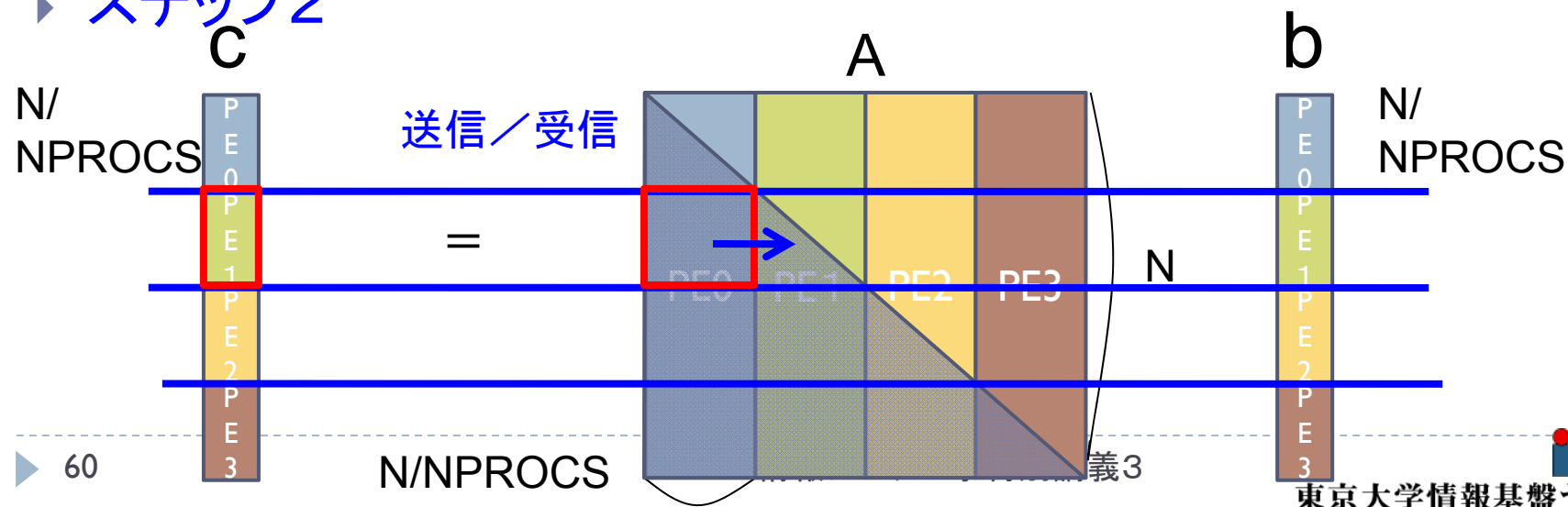
- ▶ 以上をまとめると：
 1. 最外ループ k は、ブロック幅 ib ごとに進みます
 2. 対角ブロックを持っているPEは、**対角ブロック用の計算**(←注意)をして、対応する c の要素を確定します。
 - ▶ 対角ブロックを持っているPEの判定方法は、LU分解の場合と同じです。
 3. 対角ブロックをもつPEは、 $myid - 1$ から計算している c の部分を受け取り、計算後、 $myid + 1$ に結果を送る。
 - ▶ PE0は受け取らない、PE $numprocs - 1$ は送らない
 4. 対角ブロック担当PEは、計算結果を送らない。

前進代入部分：処理の流れ

▶ ステップ1

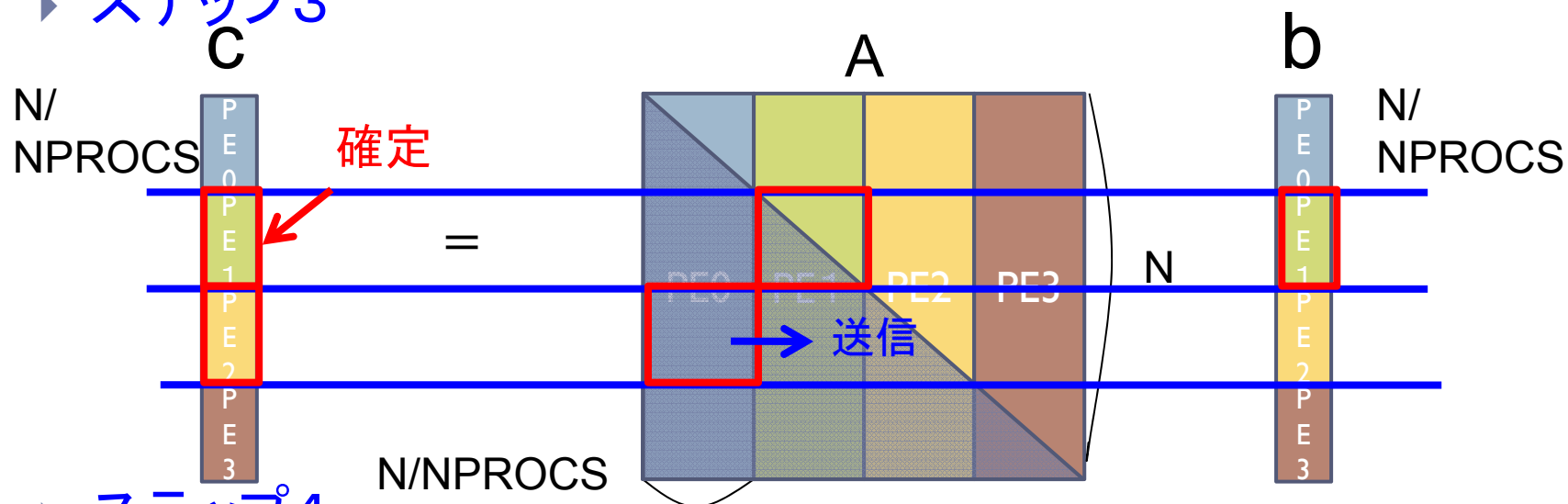


▶ ステップ2

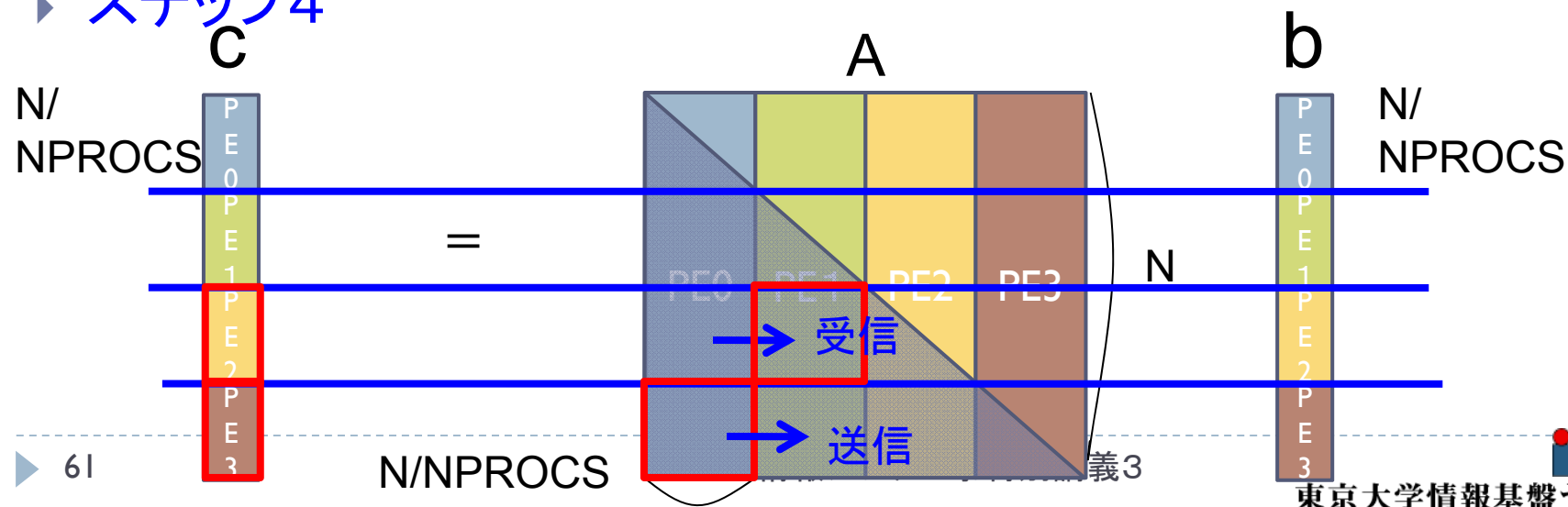


前進代入部分：処理の流れ

▶ ステップ3



▶ ステップ4

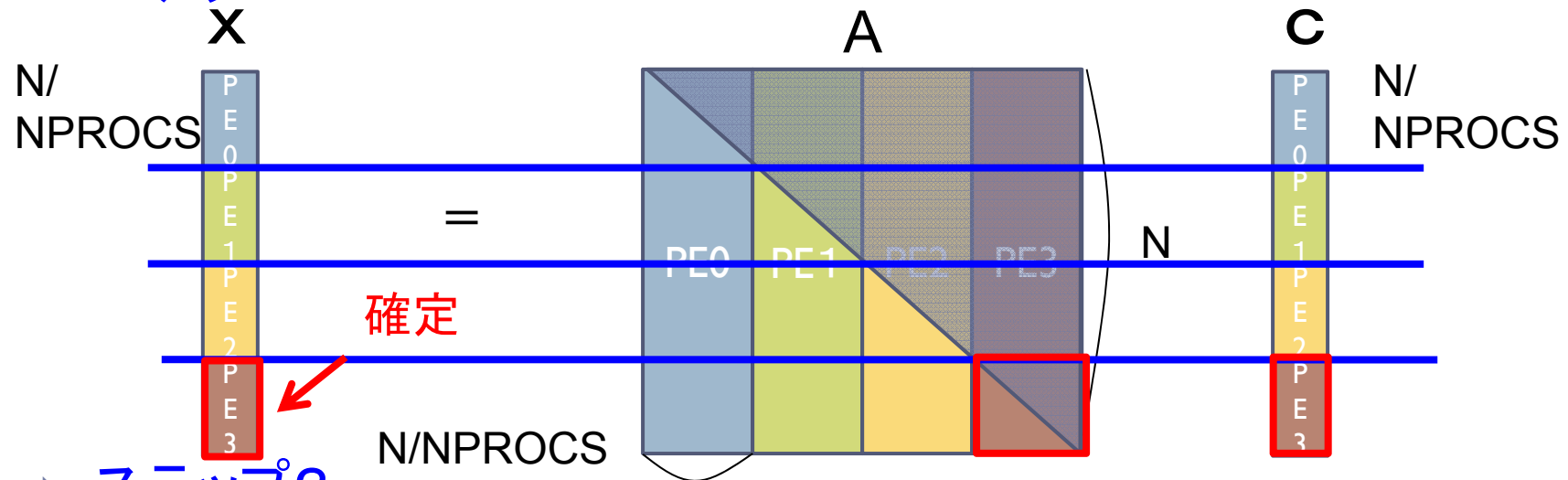


後退代入部分

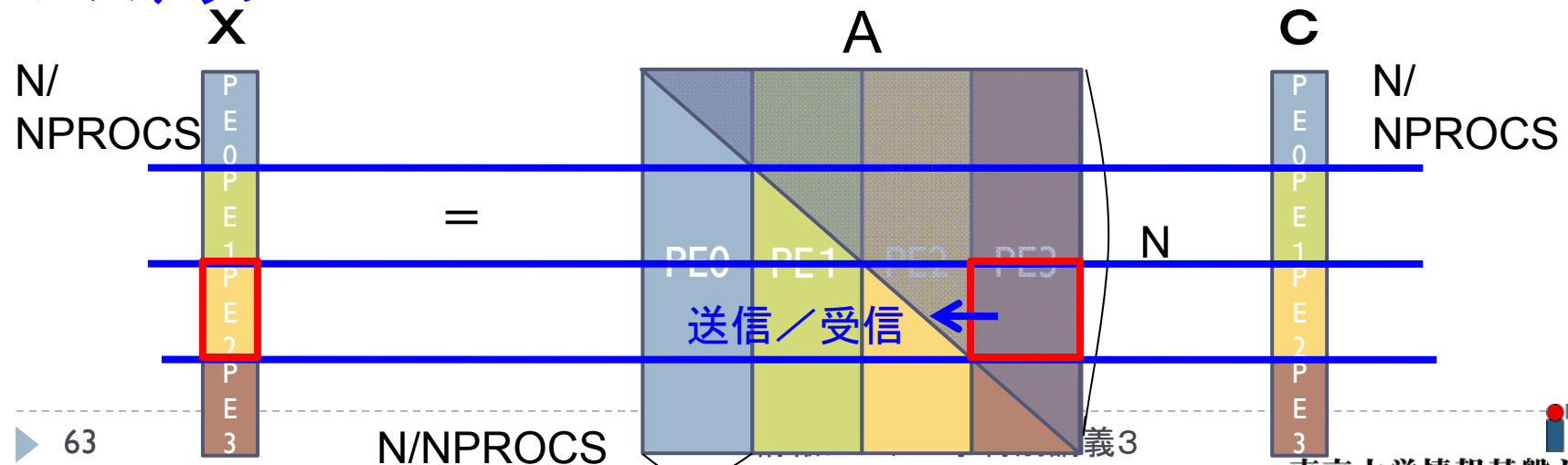
- ▶ 前進代入と同様な処理をします。
- ▶ ただし後退代入は前進代入に比べ、以下の違いがあります。
 1. 後ろから処理が始まります
 2. 対角ブロックでの、**行列Aの対角要素の割り算**が必要です

後退代入部分

▶ ステップ1



▶ ステップ2



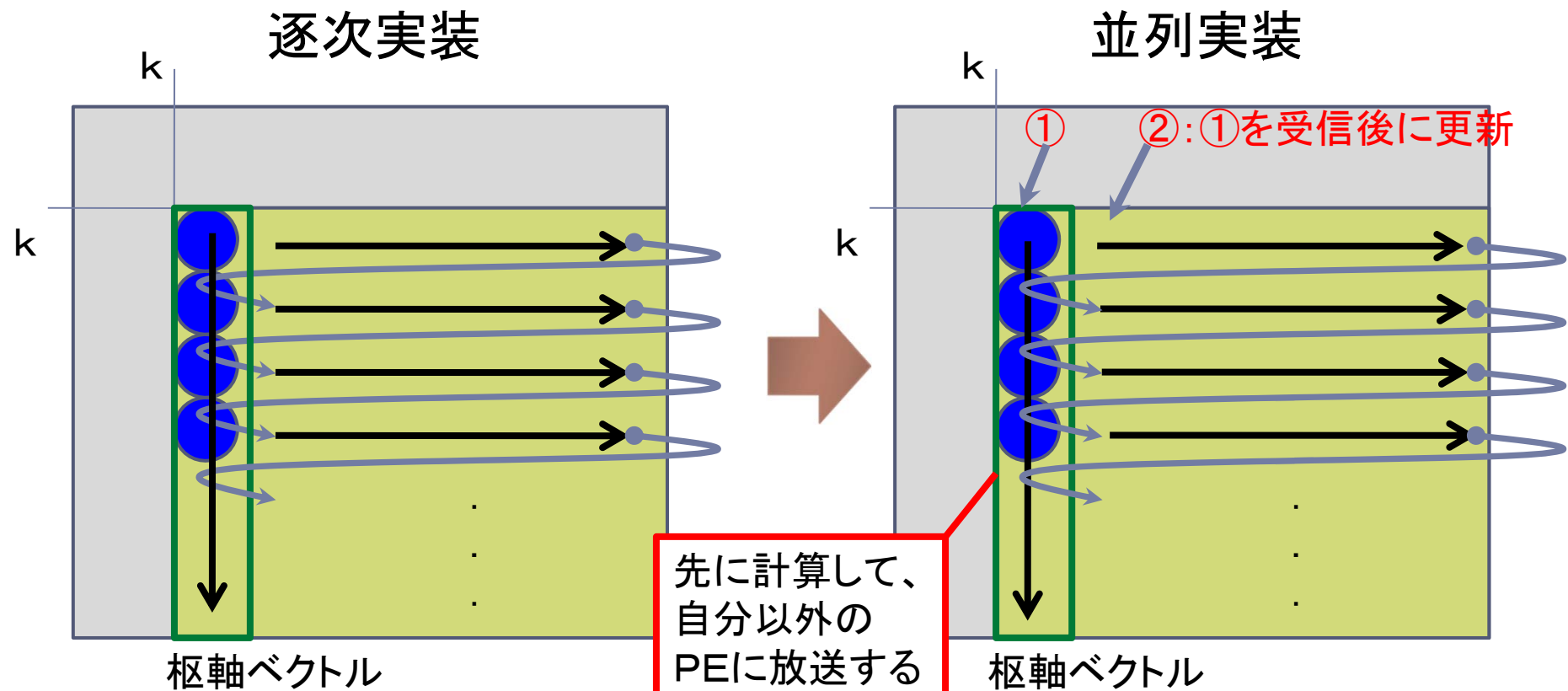
講義の流れ

1. LU分解法の
逐次アルゴリズム解説
2. 並列化実習のつづき

LU分解並列化のヒント（２）

LU分解部分並列化の方針（C言語）

- ▶ LU分解部分では、枢軸ベクトルをもつPEが先に計算し（図の①）、それをその他のPEに放送する必要があります。



LU分解部分のプログラム解説 (C言語)

```
for (k=0; k<n; k++) {
```

基本行(k行)の移動ループ

```
    dtemp = 1.0 / A[k][k];  
    for (i=k+1; i<n; i++) {  
        A[i][k] = A[i][k]*dtemp;  
    }
```

基本行からの係数を計算し、
枢軸ベクトルを求めている
部分(①)

```
    for (j=k+1; j<n; j++) {  
        dtemp = A[j][k];  
        for (i=k+1; i<n; i++) {  
            A[j][i] = A[j][i] - A[k][i]*dtemp;  
        }  
    }  
}
```

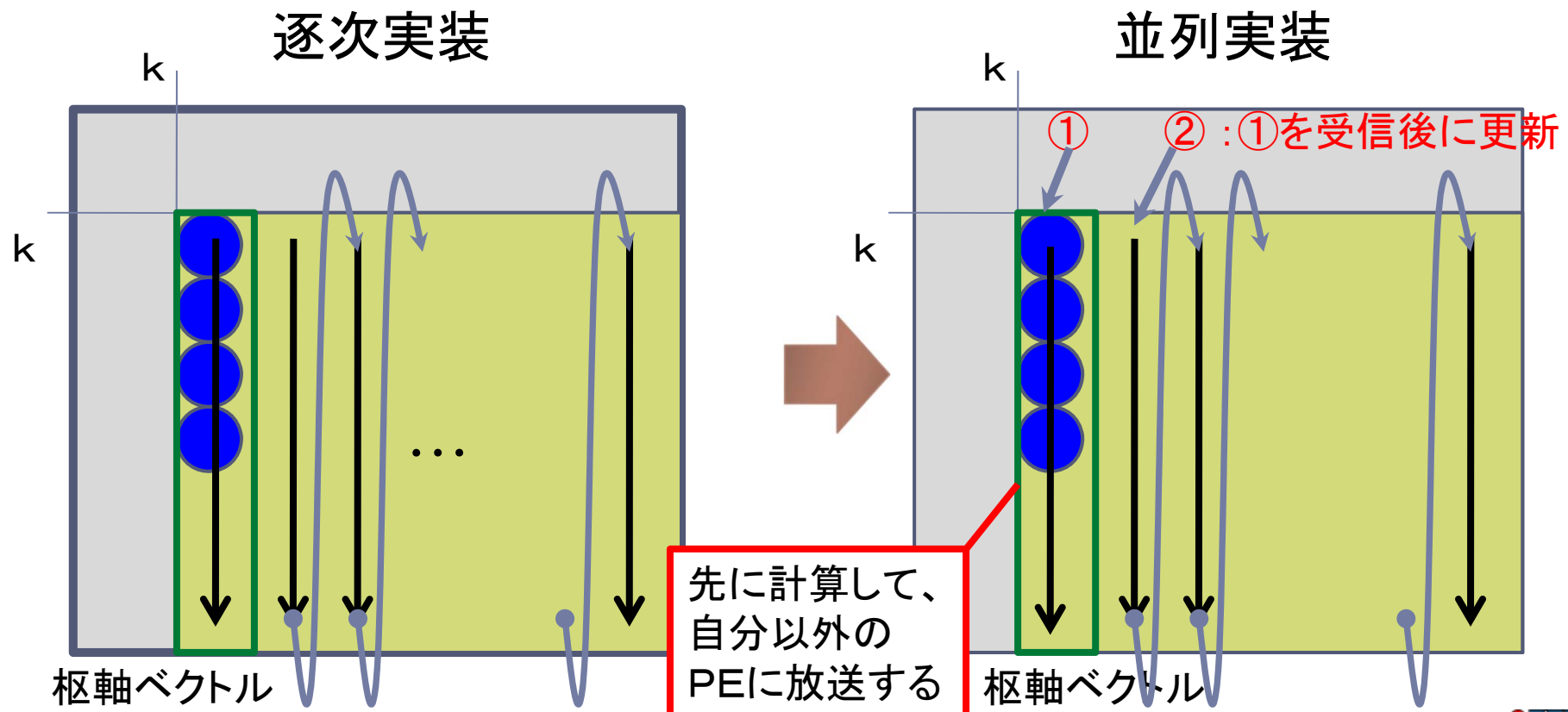
枢軸ベクトルを参照

枢軸ベクトルを参照しつつ、
消去を行っている部分(②)

基本行を参照

LU分解部分並列化の方針（Fortran言語）

- ▶ LU分解部分では、枢軸ベクトルをもつPEが先に計算し（図の①）、それをその他のPEに放送する必要があります。



LU分解部分のプログラム解説 (Fortran言語)

```
do k=1, n
```

基本行(k行)の移動ループ

```
  dtemp = 1.0d0 / A(k, k)
```

```
  do i=k+1, n
```

```
    A(i, k) = A(i, k)*dtemp
```

```
  enddo
```

基本行からの係数を計算し、
枢軸ベクトルを求めている
部分(①)

```
  do j=k+1, n
```

```
    dtemp = A(k, j)
```

```
    do i=k+1, n
```

```
      A(i, j) = A(i, j) - dtemp * A(i, k)
```

```
    enddo
```

```
  enddo
```

基本行を参照

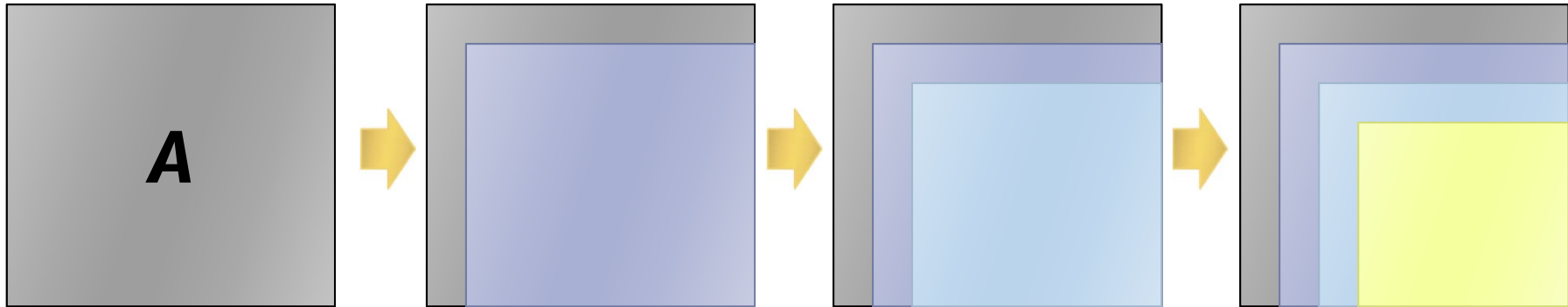
枢軸ベクトルを参照しつつ、
消去を行っている部分(②)

枢軸ベクトルを参照

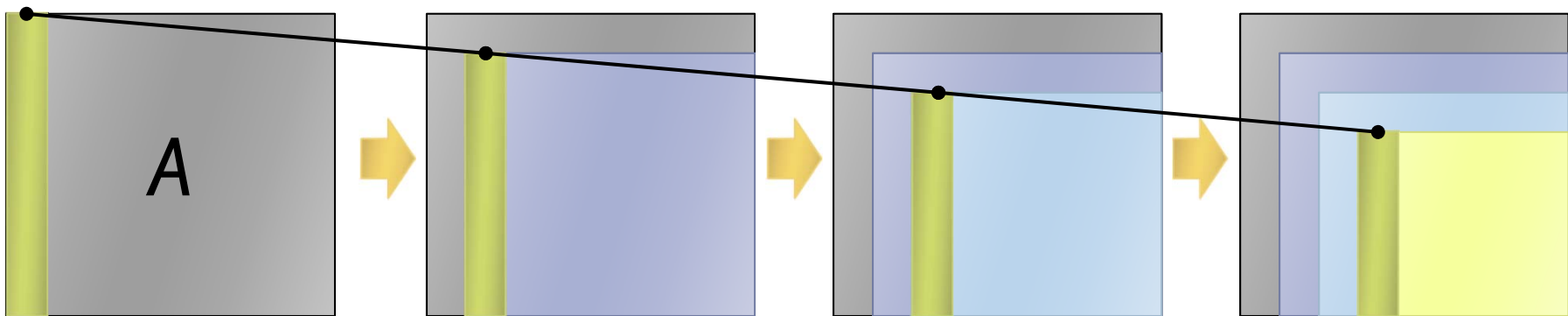
```
enddo
```

LU分解のアルゴリズムの特徴

- ▶ LU分解は、更新範囲が1つづ小さくなっていく



- ▶ 枢軸ベクトルも、1つづ小さくなっていく
- ▶ 送信するメッセージサイズも、1つづ小さくなっていく



前進代入部分のプログラム解説（C言語）

```
for (k=0; k<n; k++) {
```

```
    c[k] = b[k];
```

```
    for (j=0; j<k; j++) {
```

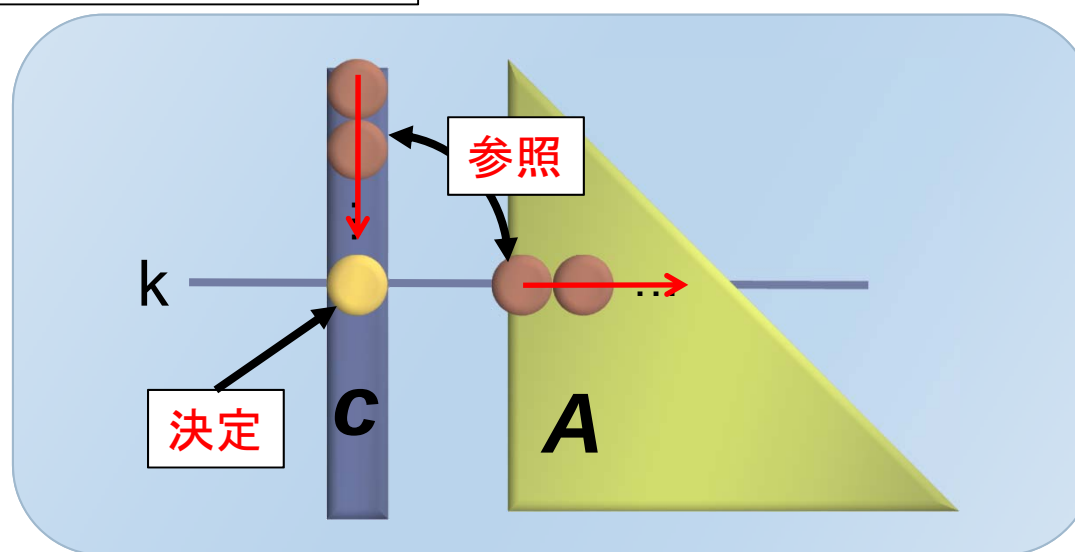
```
        c[k] -= A[k][j]*c[j];
```

```
    }
```

```
}
```

ベクトルcの値を決定する
要素(k要素)の移動ループ

k要素より前のベクトルcの要素
を参照して、k要素の値を決定

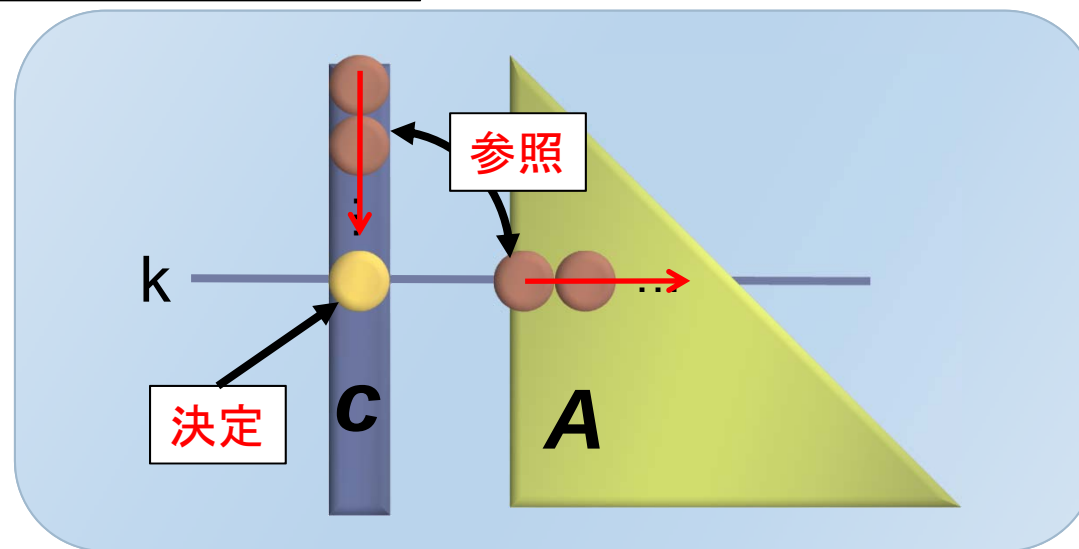


前進代入部分のプログラム解説 (Fortran言語)

```
do k=1, n  
  c(k) = b(k)  
  do j=1, k-1  
    c(k) = c(k) - A(k, j)*c(j)  
  enddo  
enddo
```

ベクトルcの値を決定する
要素(k要素)の移動ループ

k要素より前のベクトルcの要素
を参照して、k要素の値を決定



LU分解の並列化方法の確認（再掲）

1. LU分解部分のみ並列化する
2. 行列Aを表示し、逐次の答え(LuAc.dat)と一致しているか確認する
3. 前進代入部分を並列化する
4. 行列cを表示し、逐次の答え(LuAc.dat)と一致しているか確認する
5. 後退代入部分を並列化する
6. 行列xを表示し、逐次の答え(すべて1)と一致しているか確認する

**鉄則：一度にすべて並列化しても、まず動かない。
地道に並列化していくのが完成への早道。**

LU分解並列化のヒント（２） C言語版

ほぼ解答が載っています

LU分解部分(1)

```
▶ ib = n/numprocs;
  istart = myid * ib;
  iend = (myid+1)* ib;

  /* LU decomposition ----- */
  for (k=0; k<iend; k++) {
    iddiagPE = k / ib;
    if (iddiagPE == myid) { /* 枢軸列をもつPE */
      dtemp = 1.0 / A[k][k];
      枢軸列の計算と、buf[ ]へ枢軸列をコピー;
      for (i=myid+1; i<numprocs; i++) { /* 枢軸列の転送 */
        MPI_Send(&buf[...], ... , MPI_DOUBLE, i, k, MPI_COMM_WORLD);
      }
      istart = k+1; /* 担当範囲の縮小 */
    } else { /* 枢軸列を持たないPE */
      MPI_Recv(&buf[...], ... , MPI_DOUBLE, iddiagPE, k, MPI_COMM_WORLD, &istatus);
    }
  }
```

LU分解部分(2)

```
/* 共通消去部分 */
```

```
for (j=k+1; j<n; j++) {  
    dtemp = buf[j];  
    for (i=istart; i<iend; i++) {  
        A[j][i] = A[j][i] - A[k][i]*dtemp;  
    }  
}
```

```
} /* End of k-loop ----- */
```

```
/* 前進消去にメッセージがかぶらないように同期 ----- */
```

```
MPI_Barrier(MPI_COMM_WORLD);
```

前進代入部分(1)

```
▶ istart = myid * ib; iend = (myid+1) * ib; /* 担当範囲の初期化 */
/* Forward substitution ----- */
for (k=0; k<n; k++)
    c[k] = 0.0; /* cの初期化 */

for (k=0; k<n; k+=ib) { /* 対角ブロック判定用ループ */
    if (k >= istart) { /* 担当するブロックがある */
        iddiagPE = k / ib;
        if (myid != 0)
            /* 左隣りPEからデータを受け取る */
            MPI_Recv(&c[k], ib, MPI_DOUBLE, myid-1, k, MPI_COMM_WORLD, &istatus);
        if (myid == iddiagPE) { /* 対角ブロックをもつPE */
            /* 対角ブロックだけ先行計算し値を確定させる */
            for (kk=0; kk<ib; kk++) {
                c[k+kk] = b[k+kk] + c[k+kk]; /* 途中結果が送られてくるため必要な変更点 */
                for (j=istart; j<istart+kk; j++)
                    c[k+kk] -= A[k+kk][j] * c[j];
            }
        }
    }
}
```

前進代入部分(2)

```
    } else { /* 対角ブロックを持たないPE */
        /* 自分の所有範囲のデータのみ計算(まだ最終結果ではない) */
        for (kk=0; kk<ib; kk++)
            for (j=istart; j<iend; j++)
                c[k+kk] -= A[k+kk][j]*c[j];

        /* 右隣のPEに、自分の担当範囲のデータを用いた演算結果を送る */
        if (myid != numprocs-1)
            MPI_Send(&c[k], ib, MPI_DOUBLE, myid+1, k, MPI_COMM_WORLD);
    }

} /* End of if(担当するブロックがある) ----- */
} /* End of k-loop ----- */
```

LU分解並列化のヒント（2） Fortran言語版

ほぼ解答が載っています

LU分解部分(1)

```
▶  ib = n/numprocs
    istart = myid * ib + 1
    iend = (myid+1)* ib
c  --- LU decomposition -----
    do k=1, iend
        iddiagPE = (k-1) / ib
c  --- 枢軸列をもつPE
        if (iddiagPE .eq. myid) then
            dtemp = 1.0 / A(k, k)
            枢軸列の計算
c  --- 枢軸列の転送
            do i=myid+1, numprocs - 1
                call MPI_Send(A(k,k)), ..., MPI_DOUBLE_PRECISION, i, k, MPI_COMM_WORLD, ierr )
            enddo
c  --- 担当範囲の縮小
            istart = k + 1
        else
c  --- 枢軸列を持たないPE
            call MPI_Recv(A(k,k)), ..., MPI_DOUBLE_PRECISION iddiagPE, k, MPI_COMM_WORLD, istatus, ierr)
        endif
    enddo
```

LU分解部分(2)

```
c      --- 共通消去部分
      do j=istart, iend
        dtemp = A( k, j )
        do i=k+1, n
          A(i, j) = A(i, j) - A(i, k) * dtemp
        enddo
      enddo

      enddo

c      --- End of k-loop -----

c      --- 前進消去にメッセージがかぶらないように同期 -----
call MPI_Barrier(MPI_COMM_WORLD, ierr)
```

前進代入部分(1)

```
c --- 担当範囲の初期化
istart = myid * ib + 1
iend = (myid+1) * ib
c --- Forward substitution -----
c --- c の初期化
do k=1, n
    c[k] = 0.0    enddo
c ---対角ブロック判定用ループ
do k=1, n, ib
    if (k .le. istart) then
        iddiagPE = (k-1) / ib
c --- 担当するブロックがある
        if (myid .ne. 0) then
c --- 左隣りPEからデータを受け取る
            call MPI_Recv(c(k), ib,
&                MPI_DOUBLE_PRECISION,
&                myid-1, k, MPI_COMM_WORLD,
&                istatus, ierr)
```

```
        if (myid .eq. iddiagPE) then
c --- 対角ブロックをもつPE
            do kk=1, ib
c --- 途中結果が送られてくるため必要な変更点
                c(k+kk-1) = b(k+kk-1) + c(k+kk-1)
c ---対角ブロックだけ先行計算し値を確定させる
                do j=istart, istart+kk-2
                    c(k+kk-1) = c(k+kk-1) - A(k+kk-1, j) * c(j)
                enddo
            enddo
```

前進代入部分(2)

```
else
c    --- 対角ブロックを持たないPE
    do kk=1, ib
        do j=istart, iend-1
            c(k+kk-1) = c(k+kk-1) - A(k+kk-1, j) * c(j)
        enddo
    enddo
c    --- 自分の所有範囲のデータのみ計算(まだ最終結果ではない)
    if (myid .ne. numprocs-1) then
c        --- 右隣のPEに、自分の担当範囲のデータを用いた演算結果を送る
        call MPI_Send(c(k), ib, MPI_DOUBLE_PRECISION, myid+1,
&                k, MPI_COMM_WORLD, ierr)
        endif
    endif
endif
c    --- End of if 担当するブロックがある -----
enddo
c    --- End of k-loop -----
```

レポート課題

1. [L20] MyLUSolve関数を並列化せよ。各PEで行列Aについて、すべての範囲を確保してよい。
2. [L25] MyLUSolve関数を並列化せよ。各PEで行列Aについて、最低限の範囲を確保せよ。
3. [L30] MyLUSolve関数を並列化せよ。枢軸選択処理を実装せよ。

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

4. [L30] MyLUSolve関数を、同時多段多列消去法を用いて並列化せよ。また、同時多段多列の個数(ブロック幅)をチューニングして、性能を評価せよ。
5. [L35] 4. に加え、各ループにアンローリングを施し、性能をチューニングせよ。
6. [L40] 5. に加え、ノンブロッキング通信を用いて通信処理を高速化せよ。LU分解、前進代入、後退代入処理において、通信と計算がオーバーラップするようなアルゴリズムを採用せよ。ここで前進代入、後退代入処理においては、ウェーブフロント処理を考慮すること。