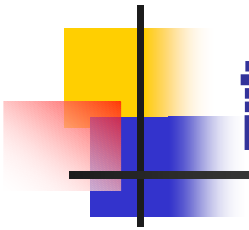
RIKEN AICS Summer School
講義6「並列アルゴリズム基礎」

2013年8月7日

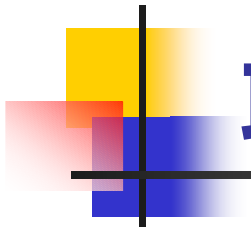
神戸大学大学院システム情報学研究科
計算科学専攻
山本有作



講義の目標

- 基本的な行列計算アルゴリズムを例題として，並列化の手法を学ぶ

- 特に，次の点に着目して説明を行う
  - アルゴリズム
  - アルゴリズムの並列性
  - データ分割方式
  - 通信パターンと利用できる MPI 関数
  - 並列化効率向上のための最適化手法



取り上げる例題と学習項目

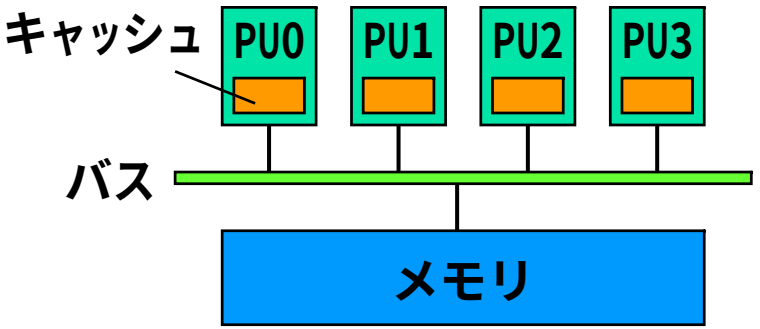
- 行列ベクトル積
  - 巡回通信
  - ベクトルに対するリダクション
- 行列どうしの積
  - MPI プログラムの性能予測
  - 2次元分割
  - Fox のアルゴリズム
  - 部分ブロードキャスト
  - mpi_sendrecv

取り上げる例題と学習項目 (続き)

- 連立1次方程式の直接解法 (LU分解)
  - サイクリック分割
  - ブロックサイクリック分割
- 疎行列ベクトル積
  - 不規則データを扱う場合の並列化
- 3次元FFT
  - 全対全通信
- 熱伝導方程式の数値解法
  - 通信量の削減
  - 通信の隠蔽

対象とする計算機アーキテクチャ (I)

- 共有メモリ型並列計算機
  - 複数のプロセッサ (PU) がバスを通してメモリを共有
  - どのPUも同じメモリ領域にアクセスできる

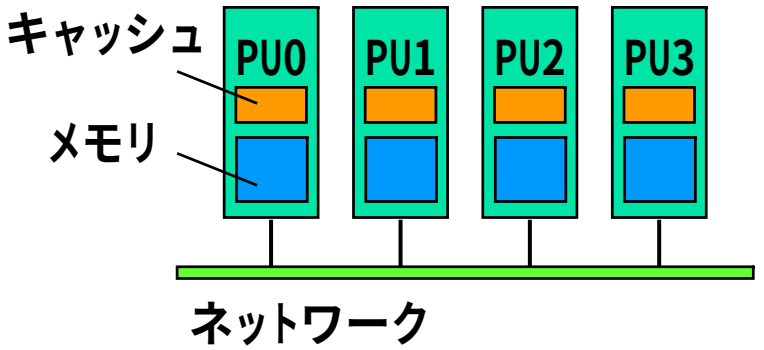


- 特徴
  - メモリ空間が単一のためプログラミングが容易
  - PUの数が多すぎると，アクセス競合により性能が低下
    → 2〜16台程度の並列が多い

- プログラミング言語
  - OpenMP (FORTRAN/C/C++ ＋ 指示文) を使用
  - コンパイラによる自動並列化も利用可能

対象とする計算機アーキテクチャ (II)

- 分散メモリ型並列計算機
  - 各々がメモリを持つ複数のPUを ネットワークで接続
  - 各PUはそれぞれ自分の持つメモリ のみにアクセス可能

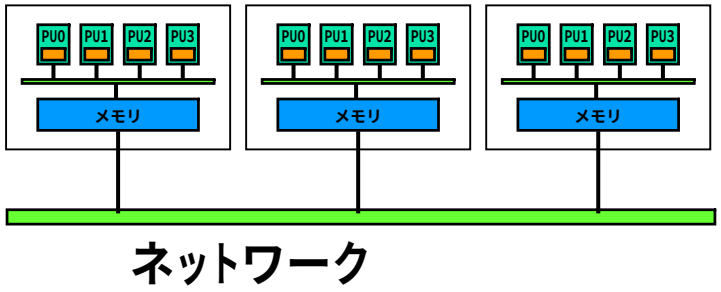


- 特徴
  - 数千～数万PU規模の並列が可能
  - PU間へのデータ分散を意識したプログラミングが必要

- プログラミング言語
  - FORTRAN/C/C++ ＋ MPI を使用

対象とする計算機アーキテクチャ (III)

- SMPクラスタ
  - 複数の共有メモリ型並列計算機 (SMP) をネットワークで接続



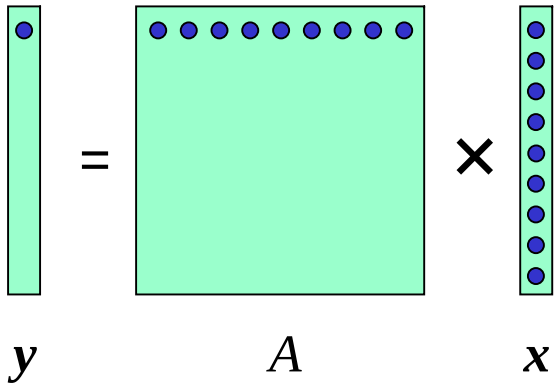
- 特徴
  - 各ノードの性能を高くできるため，比較的少ないノード数で高性能を達成できる
  - プログラミングは，ノード内部の計算では共有メモリ型並列機として，ノードをまたがる計算では分散メモリ型並列機として行う
  - 「京」をはじめとするハイエンドスパコンはほとんどがこのタイプ

- プログラミング
  - MPI と OpenMP (あるいは自動並列化) とを組み合わせて使
  - MPI のみでプログラミングすることも可能

行列ベクトル積 $y = Ax$

- アルゴリズム

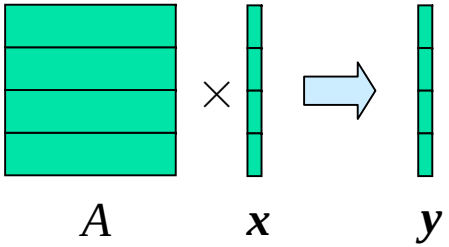
```
do i=1, n
   s = 0.0
   do j=1, n
      s = s + a(i,j) * x(j)
   end do
   y(i) = s
end do
```



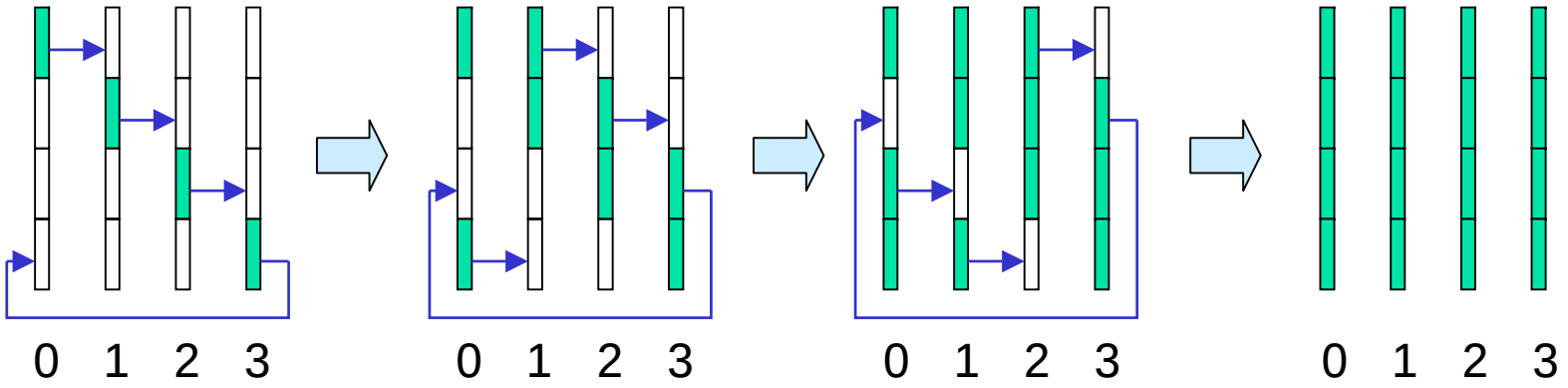
- 並列化方法
  - 各 i についての計算は独立なので，i のループを並列化可能
    - ベクトル y の各要素は，他とは独立に計算できる
  - j のループは総和なので，これを並列化することも可能
    - 部分和を計算し，後で足し合わせる

並列化方法 (I)：i に関する並列化

- データ分割
  - 右図のように，A はブロック行分割，x はブロック分割されているとする

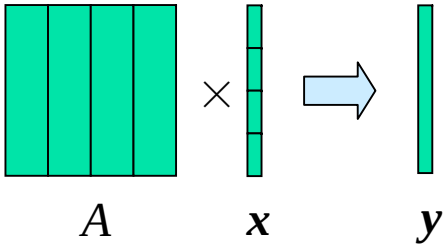


- 計算方法
  - A の行ベクトルとの内積を計算するため，各PUは x の全体が必要
  - 巡回通信により，各PU がそれぞれ x の全体を持つようにする
  - その後，各PUは担当する y の要素を独立に計算

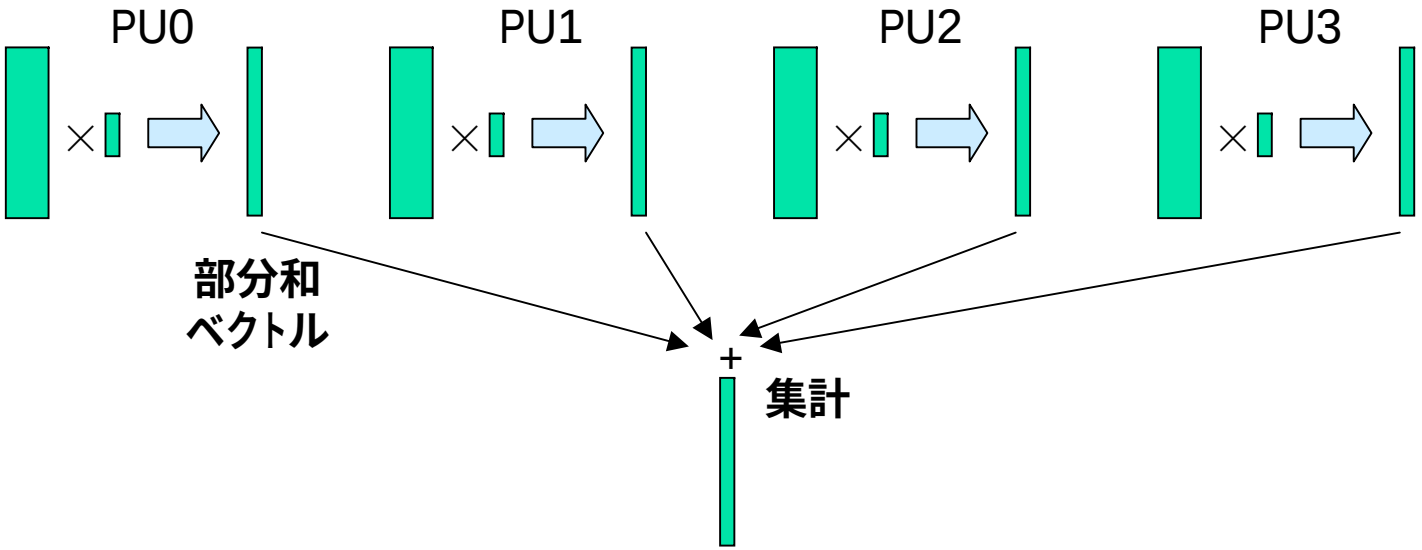


並列化方法 (II)：j に関する並列化

- **データ分割**
  - 右図のように，A はブロック列分割，x はブロック分割されているとする



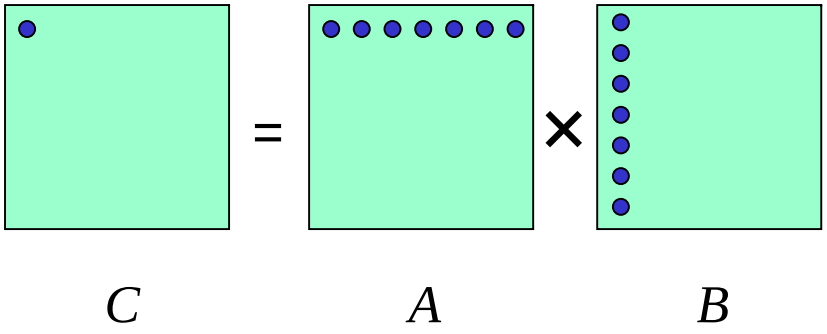
- **計算方法**
  - 各PUが自分の持つ A, x の要素のみを使い，部分和ベクトルを計算
  - 部分和ベクトルを mpi_reduce でPU0に集計し，y を計算



行列どうしの積 $C = AB$

- アルゴリズム

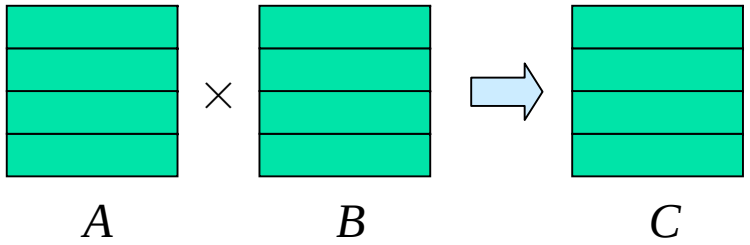
```
do i=1, n
   do j=1, n
      s = 0.0
      do k=1, n
         s = s + a(i,k) * b(k,j)
      end do
      c(i,j) = s
   end do
end do
```



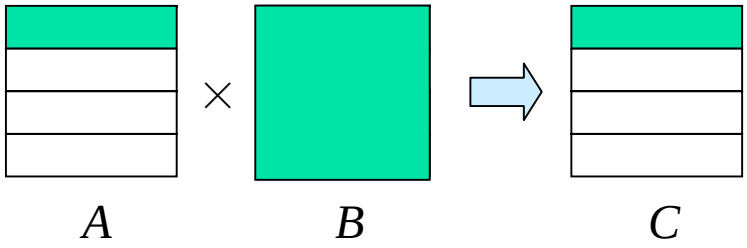
- 並列化方法
  - 各 i についての計算は独立なので, i のループを並列化可能
  - 各 j についての計算は独立なので, j のループも並列化可能
  - k のループは総和なので, これを並列化することも可能

並列化方法 (I)：i に関する並列化

- **データ分割**
  - 右図のように，A も B もブロック行分割されているとする



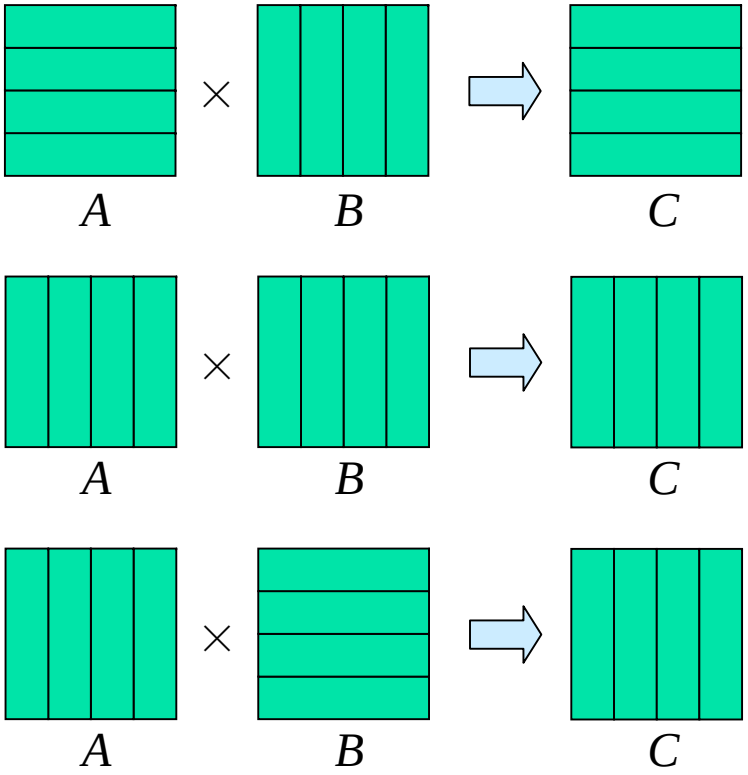
- **計算方法**
  - A の行ベクトルとの内積を計算するため，各PUは B の全体が必要
  - 巡回通信により，各PU がそれぞれ B の全体を持つようにする
  - その後，各PUは担当する C の要素を独立に計算



巡回通信後に PU0 が持つ部分行列と演算パターン

並列化方法 (II)：他の1次元分割

- データ分割の自由度
  - 行列乗算では，A, B, C の分割方式に大きな自由度がある
  - 分割方式に応じて，通信パターンを決定する必要がある

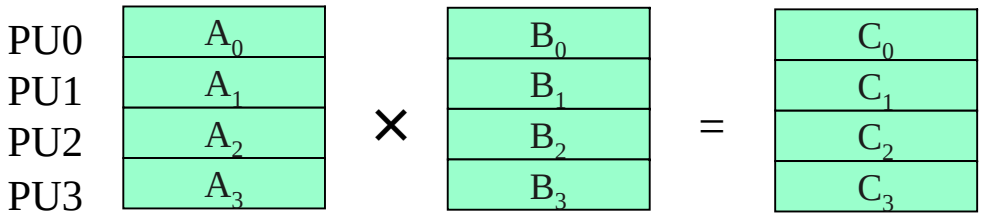


MPIプログラムの実行時間予測

- **演算時間**
  - T_{comp} ＝ (**演算量**) ／ (**演算速度**)
    - もっとも単純なモデル化
    - より精密には，キャッシュの影響などを考慮する必要あり
- **通信時間**
  - T_{comm} ＝ (**通信回数**) × (**ｾｯﾄｱｯﾌﾟ時間**) ＋ (**通信量**) ／ (**通信速度**)
- **待ち時間**
  - T_{idle}
    - PU間で負荷の不均衡がある場合などに発生
- **全実行時間**
  - $T_{exec} = T_{comp} + T_{comm} + T_{idle}$

行列乗算（ブロック行分割）の例

- **計算方法**
  - **巡回通信によって全PUが行列 B を共有した後，乗算を行う**



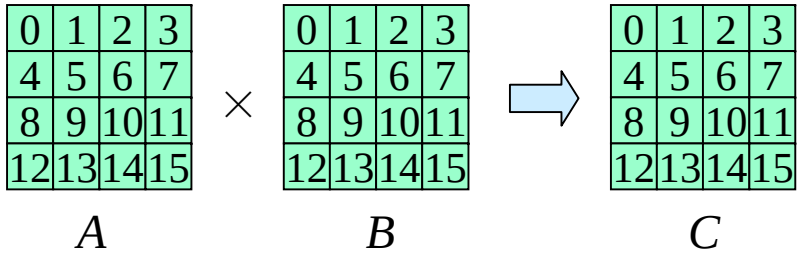
- **実行時間の予測**
  - $T_{comp} = 2N^3 / P / s$ ：P に反比例して減少
  - $T_{comm} = (P - 1) * (T_{setup} + (N / P) * N * 8 / b)$
    $= (P - 1) * T_{setup} + 8 (1 - 1/P) N^2 / b)$ ：P とともに減少しない
  - $T_{idle} = 0$ ：負荷は完全に均等

⇨ **P を増やすと並列化効率が低下**

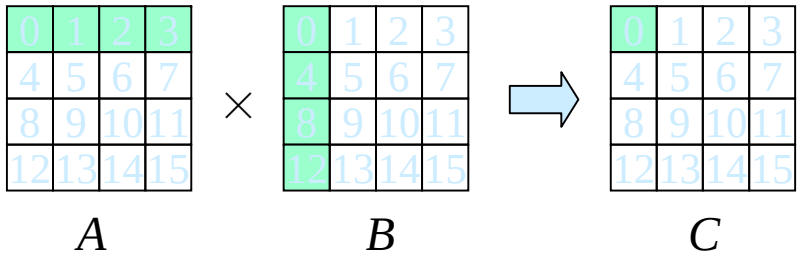
⇨ **より効率のよい並列化方法は？**

並列化方法 (III)：2次元分割

- データ分割
  - A, B が行･列の**両方向**に**ブロック分割されているとする**
  - **このとき，C も同じ分割形式で求めるとする**



- 基本的な考え方
  - **自PUが持つ のブロックを計算する**
  - **そのために必要な A, B のブロックを他PUから受け取る**
  - **A のブロックは同じ行のPU，B のブロックは同じ列のPUからもらう**



PU0 **が受け取る A, B の部分行列と演算パターン**

Fox のアルゴリズム (I)

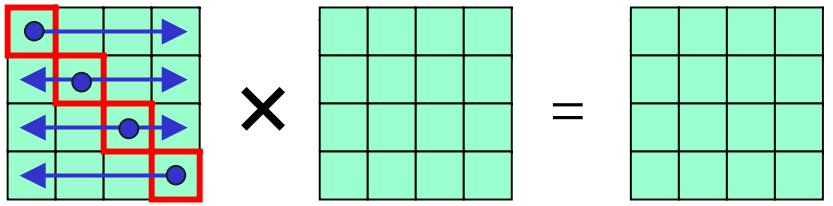
- PUと行列のブロックの番号付け
  - I行J列にあるPU／ブロックを (I,J) で表す
  - PU (I,J) は，第 K ステップで

$$C_{IJ} \mathrel{+}= A_{I,\ MOD(I+K-1,\sqrt{P})}\ B_{MOD(I+K-1,\sqrt{P}),\ J}$$

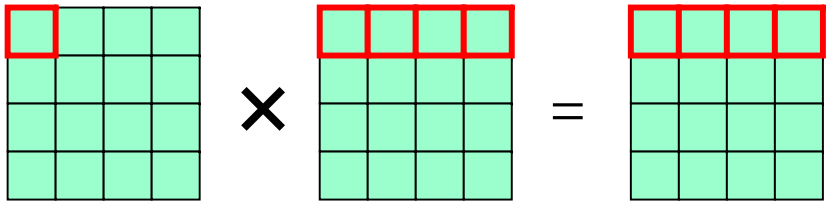
を計算

- 第1ステップ：Aの対角ブロックとの乗算

$A_{I,I}$ の横方向
ブロードキャスト



$C_{I,J} \mathrel{+}= A_{I,I} B_{I,J}$
の計算

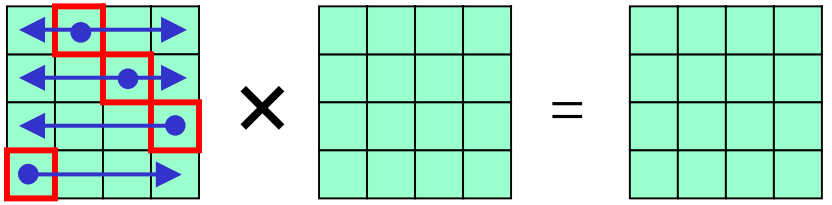


A B C

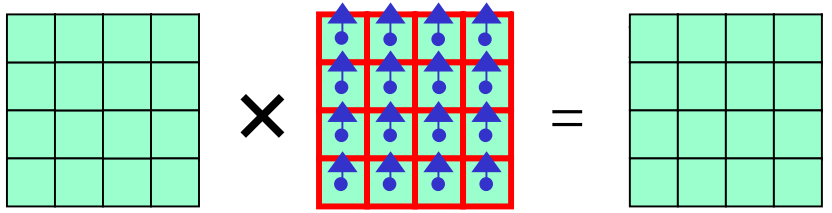
Fox のアルゴリズム (II)

- 第2ステップ：Aの対角より1個右のブロックとの乗算

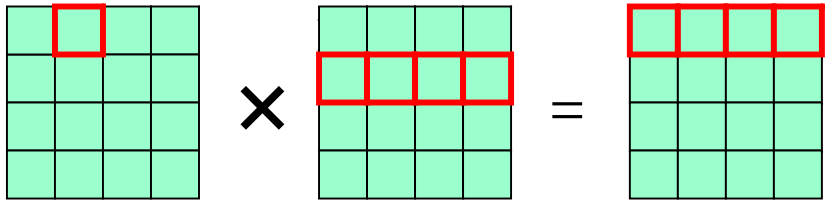
$A_{I,I+1}$ の横方向
ブロードキャスト



$B_{I,J}$ の縦方向
サイクリック転送



$C_{I,J} \mathrel{+}= A_{I,I+1} B_{I+1,J}$
の計算



A　　B　　C

- 以下同様に第3，第4ステップを行う

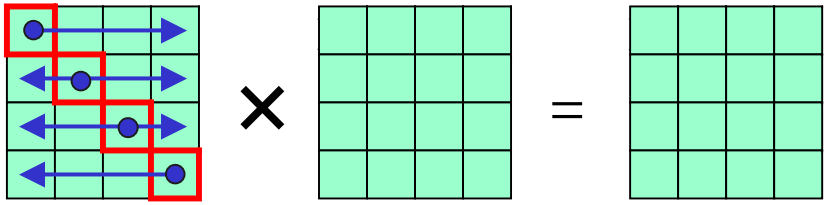
Fox のアルゴリズム (III)

- アルゴリズム

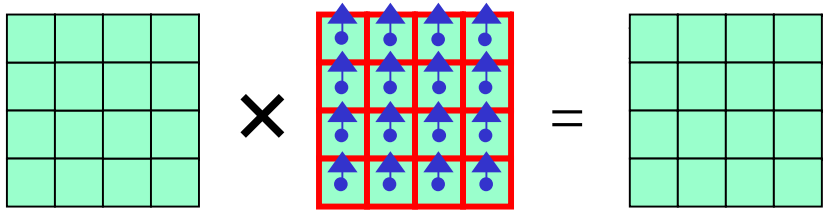
```
program fox
(MPIの初期化と環境情報取得)
(配列の確保: A,B,C,A_TMP,B_TMP)          自分の担当する部分行列を入れる配列
(自分の行番号, 列番号をそれぞれI_num,J_numとする)    0 ≦ I_num, J_num < √P
(配列A,Bにデータを入れる。配列Cを0に初期化)
do K = 1, √P
   IF (J_num = MOD(I_num+K-1,√P)) (配列AをA_TMPにコピー)
   (配列A_TMPを行方向のプロセス間でブロードキャスト)
   IF (K > 1) THEN
      (同じ列の1個上のプロセスにB_TMPを送る)
      (同じ列の1個下のプロセスからBにデータを受け取る)
   END IF
   (Bを配列B_TMPにコピー)
   C += A_TMP×B_TMP
end do
(配列Cの出力)
(MPIの終了)
stop
end
```

MPI 通信関数による実装

- A_{TMP} の横方向ブロードキャスト
  - 同じI_{num}を持つプロセスに対してコミュニケータ (プロセスのグループ) を定義
  - コミュニケータを用い，mpi_bcast により部分ブロードキャストを行う



- B_{TMP} の縦方向への巡回型通信
  - mpi_sendrecv を用いて，1つ上の PU へのデータ送信と，1つ下の PU からのデータ受信とを同時に行う



Fox のアルゴリズムの実行時間予測

- **仮定**
  - **P個のPUへの（部分）ブロードキャストは，1対1通信の $\log_2(P)$ 倍の時間で行える**
  - **mpi_sendrecv による同時送受信は，1対1通信（一方が送信して他方が受信）と同じ時間で行える**

- **実行時間の予測**
  - $T_{comp} = 2N^3 / P / s$ **：P に反比例して減少**
  - $T_{comm} = \sqrt{P} * \log_2(\sqrt{P}) * (T_{setup} + (N^2/P) * 8 / b)$ **：A_{TMP} の転送**
    $+ (\sqrt{P} - 1) * (T_{setup} + (N^2/P) * 8 / b)$ **：B_{TMP} の転送**
  - $T_{idle} = 0$ **：負荷は完全に均等**

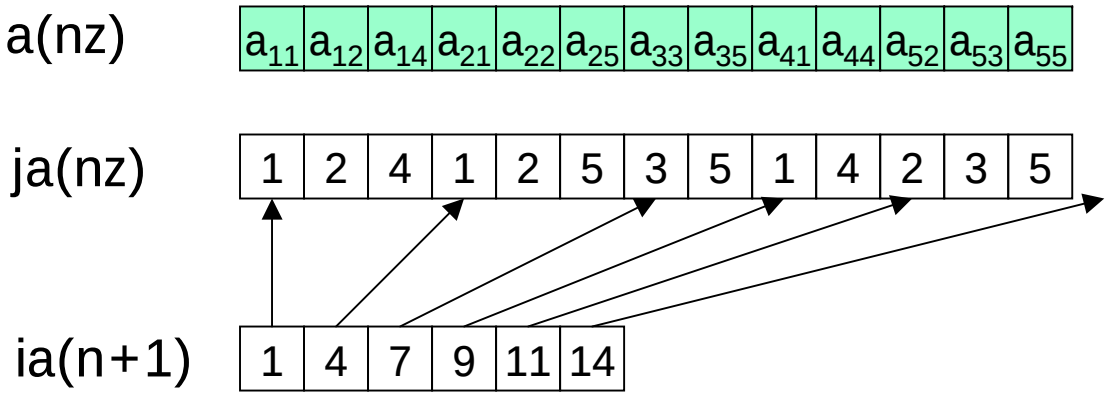
⇒ **通信時間のうち，通信量に比例する部分は P とともに減少**
1方向分割に比べて，より高い並列化効率が期待できる

疎行列ベクトル積 $y = Ax$ (I)

- **疎行列データの格納**
  - A を $n \times n$ の疎行列とし，非ゼロ要素数が nz 個であるとする
  - 3本の配列で A の非ゼロ要素を格納する CRS 形式がよく使われる
    - 非ゼロ要素の値を格納する実数型配列 a(nz)
    - 非ゼロ要素の行番号を格納する整数型配列 ja(nz)
    - 配列 c 中で各列の最初の要素を指すポインタ配列 ia(n+1)

a_{11}	a_{12}		a_{14}	
a_{21}	a_{22}			a_{25}
		a_{33}		a_{35}
a_{41}			a_{44}	
	a_{52}	a_{53}		a_{55}

疎行列 A

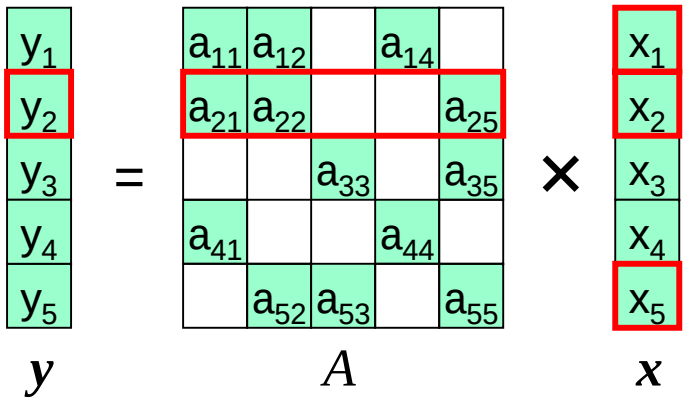


CRS形式による格納

疎行列ベクトル積 $y = Ax$ (II)

- アルゴリズム
  - 第 i 行に関する内積では,
    - 配列 a の非ゼロ要素のみを連続にアクセス
    - 対応する配列 x の要素を不連続にアクセス

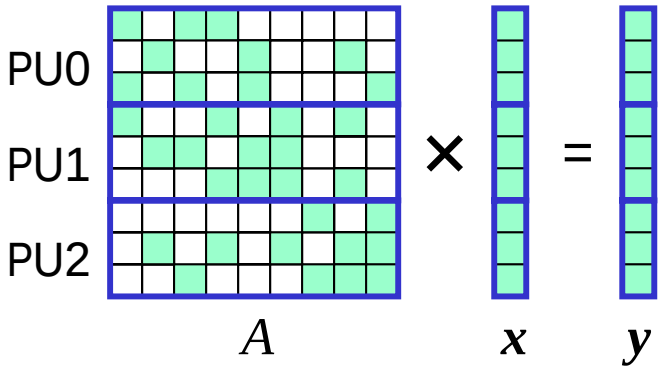
```
do i=1, n
   s = 0.0
   do j=ia(i), ia(i+1)-1
      s = s + a(j) * x(ja(j))
   end do
   y(i) = s
end do
```



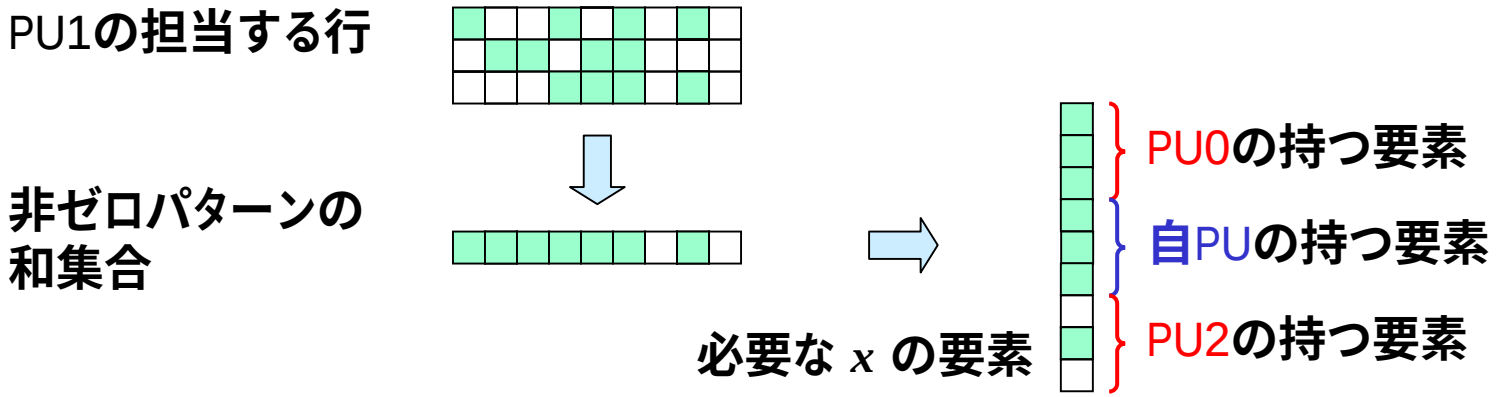
- 並列化方法
  - 各 i についての計算は独立なので, i のループを並列化する

疎行列ベクトル積の並列化 (I)

- **データ分割**
  - 右図のように，A はブロック行分割，x はブロック分割されているとする



- **各PUでの計算に必要な x の要素**
  - 各行の計算では，A の非ゼロ要素に対応する x の要素が必要
  - 担当する行の非ゼロ構造の和集合から，必要な x の要素が定まる



疎行列ベクトル積の並列化 (II)

- 計算方法
  - 自分の担当行の計算に必要な x の要素を求める
  - 他の各PUから，x のどの要素をもらってくればよいかを調べる
  - 必要な x の要素の送信を，他の各PUに依頼
    同時に，他の各PUからの送信要求を受け取る
  - 他の各PUに，要求された x の要素を送る
    同時に，他の各PUから送ってもらった要素を受け取る
  - 自分の担当行について $y = Ax$ を計算

2段階の送受信

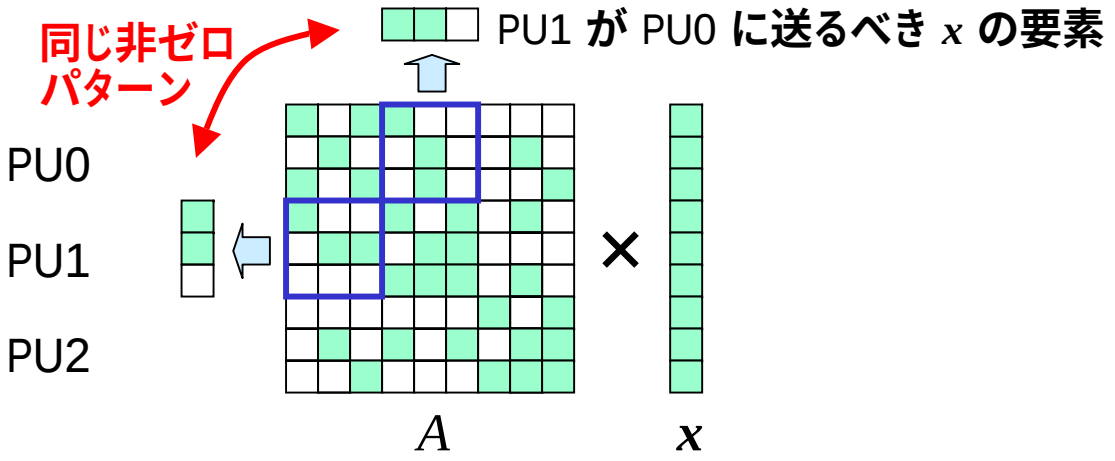
一般に，不規則データを扱う並列化では，
・どのデータを送信すべきか，受信元に教えてもらう
・実際にデータを送信する
という2段階の通信が必要となる

最適化

- **非ゼロパターンが対称の場合**
  - PU1 が PU0 に送るべき x の要素は，PU0 の第2ブロックからわかる
  - 非ゼロパターンが対称の場合は，

PU0 の第2ブロックの非ゼロパターン
=
PU1 の第1ブロックの非ゼロパターンの転置

  - したがって，PU1 は**第1段階の通信を行わずに**，自分の非ゼロパターンを調べるだけで，PU0 に送るべき x の要素がわかる



連立1次方程式の直接解法

- n 元連立1次方程式 $Ax = b$

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

- 直接解法（ガウスの消去法）
  - 変数を順番に消去していくことで，連立1次方程式を解く
  - 第1ステップ：変数の消去
    - 変数を1個ずつ消去し，1ずつ元数の小さい方程式に直していく
    - 最終的に，変数1個（x_n）のみの1次方程式が得られる
  - 第2ステップ：変数の代入
    - 1次方程式を解いて x_n を求める
    - 代入計算により，x_{n-1}, x_{n-2}, …, x_1 を順に求める

変数の消去 (第1ステップ)

- x_1 の消去
  - $a_{11} \neq 0$ と仮定
  - 第1行の a_{i1}/a_{11} 倍を第 i 行から引き，x_1 を消去
  - 更新された行列要素，右辺ベクトル要素を $a_{ij}^{(2)}$，$b_i^{(2)}$ で表す
  - 第2行目以降の式が，$n-1$ 元の連立1次方程式となる

$$
\begin{cases}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\
a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + \cdots + a_{2n}^{(2)}x_n = b_2^{(2)} \\
a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + \cdots + a_{3n}^{(2)}x_n = b_3^{(2)} \\
\vdots \\
a_{n2}^{(2)}x_2 + a_{n3}^{(2)}x_3 + \cdots + a_{nn}^{(2)}x_n = b_n^{(2)}
\end{cases}
$$

変数の消去 (第2ステップ)

- x_2 の消去
  - $a_{22}^{(2)} \neq 0$ と仮定
  - 第2行の $a_{i2}^{(2)}/a_{22}^{(2)}$ 倍を第 i 行から引き，x_2 を消去
  - 更新された行列要素，右辺ベクトル要素を $a_{ij}^{(3)}$，$b_i^{(3)}$ で表す
  - 第3行目以降の式が，$n-2$ 元の連立1次方程式となる

$$
\begin{cases}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\
a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + \cdots + a_{2n}^{(2)}x_n = b_2^{(2)} \\
a_{33}^{(3)}x_3 + \cdots + a_{3n}^{(3)}x_n = b_3^{(3)} \\
\vdots \\
a_{n3}^{(3)}x_3 + \cdots + a_{nn}^{(3)}x_n = b_n^{(3)}
\end{cases}
$$

変数の消去 (第kステップ)

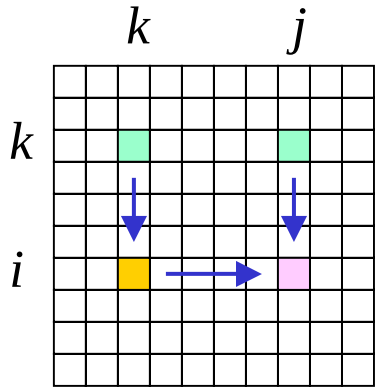
- x_k の消去
  - $a_{kk}^{(k)} \neq 0$ と仮定
  - 第k行の $a_{ik}^{(k)}/a_{kk}^{(k)}$ 倍を第 i 行から引き，x_k を消去
  - 更新された行列要素，右辺ベクトル要素を $a_{ij}^{(k+1)}$，$b_i^{(k+1)}$ で表す
  - 第 $k+1$ 行目以降の式が，$n-k$ 元の連立1次方程式となる

$$
\begin{cases}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\
a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + \cdots + a_{2n}^{(2)}x_n = b_2^{(2)} \\
a_{33}^{(3)}x_3 + \cdots + a_{3n}^{(3)}x_n = b_3^{(3)} \\
\ddots \\
\boxed{a_{nn}^{(n)}x_n = b_n^{(n)}}
\end{cases}
$$

第 $n-1$ ステップまで繰り返すことで，係数行列が上三角化される

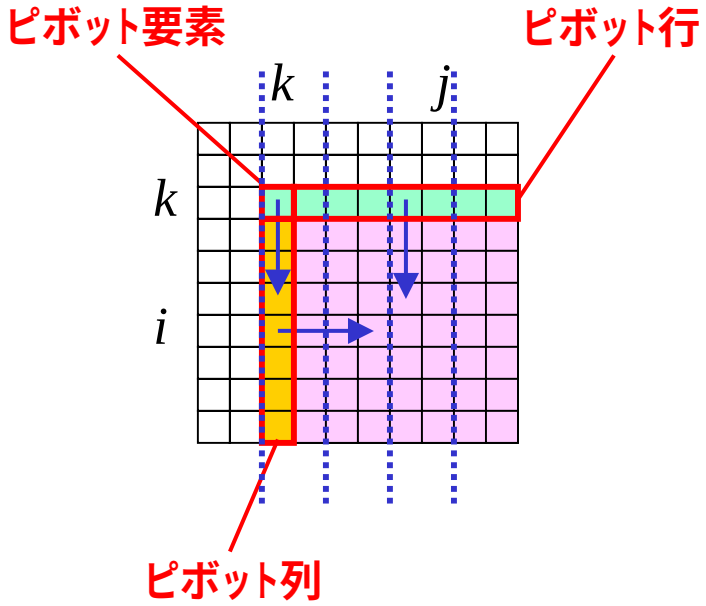
第 k ステップでの計算

- **行列要素 $a_{ij}^{(k)}$ の更新**
  - **乗数の計算**：$c_i^{(k)} = a_{ik}^{(k)} / a_{kk}^{(k)}$
  - **第 i 行から第 k 行の $c_i^{(k)}$ 倍を引く**：$a_{ij}^{(k+1)} = a_{ij}^{(k)} - c_i^{(k)} a_{kj}^{(k)}$



第 k ステップでの計算 (続き)

- **行列要素 $a_{ij}^{(k)}$ の更新**
  - **乗数の計算**: $c_i^{(k)} = a_{ik}^{(k)} / a_{kk}^{(k)}$
  - **第 i 行から第 k 行の $c_i^{(k)}$ 倍を引く**: $a_{ij}^{(k+1)} = a_{ij}^{(k)} - c_i^{(k)} a_{kj}^{(k)}$



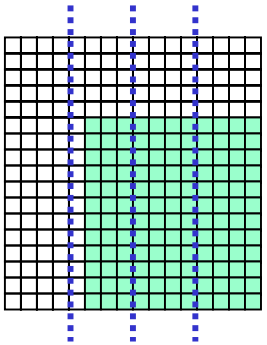
＜列分割での並列化＞

- 第k列担当のプロセスがピボット列 (乗数) を計算
- ピボット列を他のプロセスにブロードキャスト
- 全プロセスが，受け取ったピボット列と自分の持つピボット行を使い，自分の持つ要素を更新

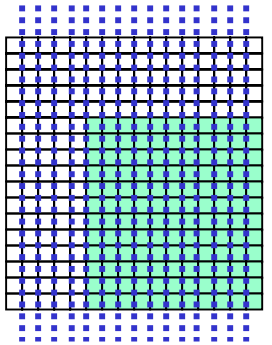
データ分割

- サイクリック分割の利用
  - 上三角化では，k が増えるごとに新範囲が減少
  - ブロック分割では，大きな負荷不均衡が生じる
  - サイクリック分割により，不均衡を軽減できる

ブロック分割
PU0はアイドル



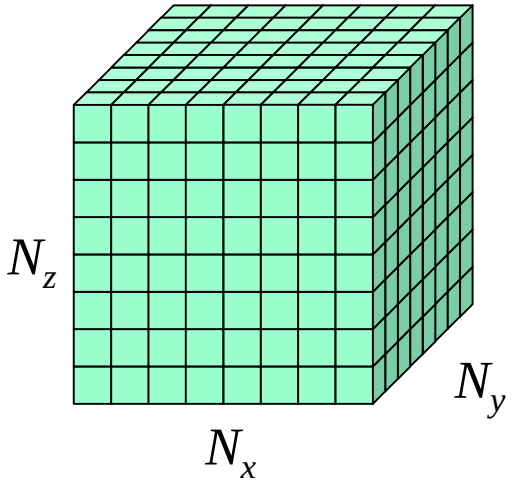
サイクリック分割
全PUが稼動



- ブロックサイクリック分割
  - ブロック単位のサイクリック分割にすることで，通信回数を削減できる
  - 通信回数と負荷均衡のトレードオフから最適ブロックサイズが決まる

3次元FFT (I)

- **入力と出力**
  - **入力**：$N_x \times N_y \times N_z$ **の3次元データ** (a_{jkl})
  - **出力**：$N_x \times N_y \times N_z$ **の3次元データ** (c_{jkl})



- **アルゴリズム**
  - **各方向ごとのFFTを3回実行**
  - $\omega_x = \exp(-2\pi i/N_x)$ **などとおくと，**

$$a^{(1)}_{pkl} = \sum_{i=1}^{Nx} a_{jkl}\,\omega_x^{jp} \qquad (k=0, \ldots, N_y,\ l=0, \ldots, N_z)$$

$$a^{(2)}_{pql} = \sum_{i=1}^{Ny} a^{(1)}_{pkl}\,\omega_y^{kq} \qquad (p=0, \ldots, N_x,\ l=0, \ldots, N_z)$$

$$c_{pqr} = \sum_{i=1}^{Nz} a^{(2)}_{pql}\,\omega_z^{lr} \qquad (p=0, \ldots, N_x,\ q=0, \ldots, N_y)$$

3次元FFT (II)

- **並列性**
  - x 方向の変換では，N_x点のFFTを同時に $N_y \times N_z$ 組行う
    ⇨ $N_y \times N_z$ の並列性
  - y, z 方向の変換についても同様

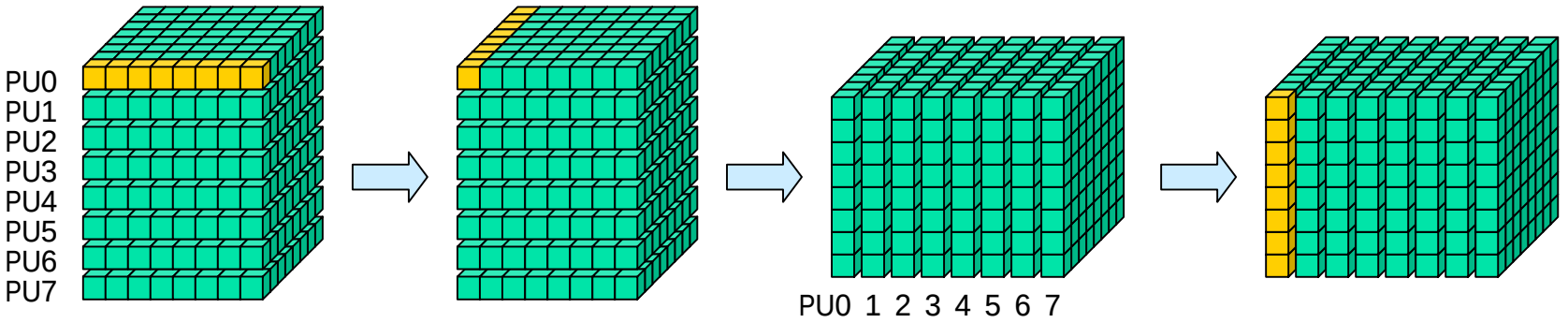
- **データ分割**
  - 各ステップにおいて，変換を行わない方向について分割すれば，変換途中での通信は不要

変換方向	分割方向
x	y または z
y	x または z
z	x または y

  - 以下，x, y, z 方向の変換でそれぞれ z, z, x方向に分割するとする

3次元FFTの並列化

- 計算方法
  - 次の4ステップで計算を行う

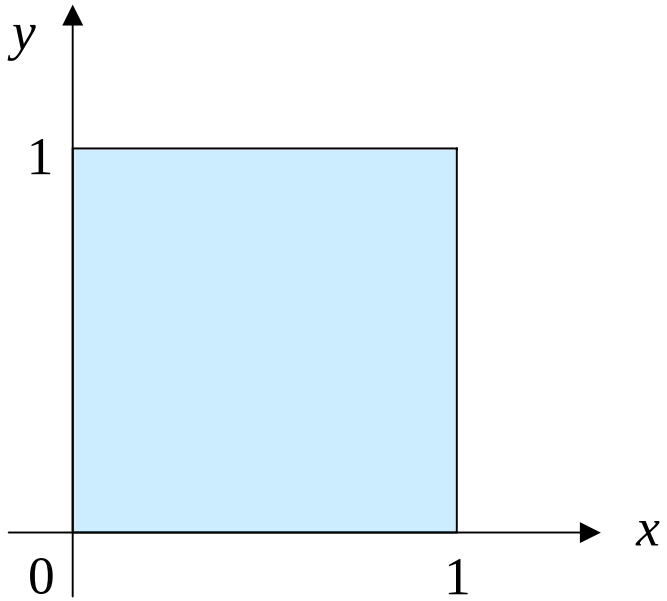


- MPI 通信関数による実装
  - データの転置では全対全通信が必要なため，mpi_alltoall を使う

熱伝導方程式の数値解法

- **問題**
  - **2次元正方形領域** $[0,1]\times[0,1]$ **での熱伝導を考える**
  - **境界をすべて0℃に固定**

$$u(0, y) = 0$$
$$u(1, y) = 0$$
$$u(x, 0) = 0$$
$$u(x, 1) = 0$$



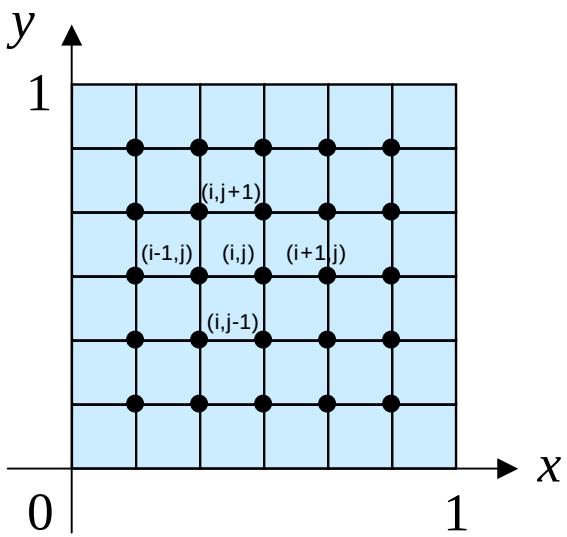
  - **領域全体に一定の熱を加える**

このとき，十分な時間が経った後での温度分布はどうなるか？

熱伝導方程式の数値解法（続き）

- **熱伝導方程式**
  - $\partial u / \partial t = \nabla^2 u + f$　（u: 温度, f: 熱源の効果）
- **離散化**
  - 領域内を格子に区切り，格子点上での温度のみを考える
  - さらに，離散的な時間ステップでの温度のみを考える
- **離散版の熱伝導方程式**
  - 時間ステップ n での格子点 (i, j) の温度を $u_{ij}^{(n)}$ とすると，

$$u_{ij}^{(n+1)} = (u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)}) / 4 + f_{ij}$$

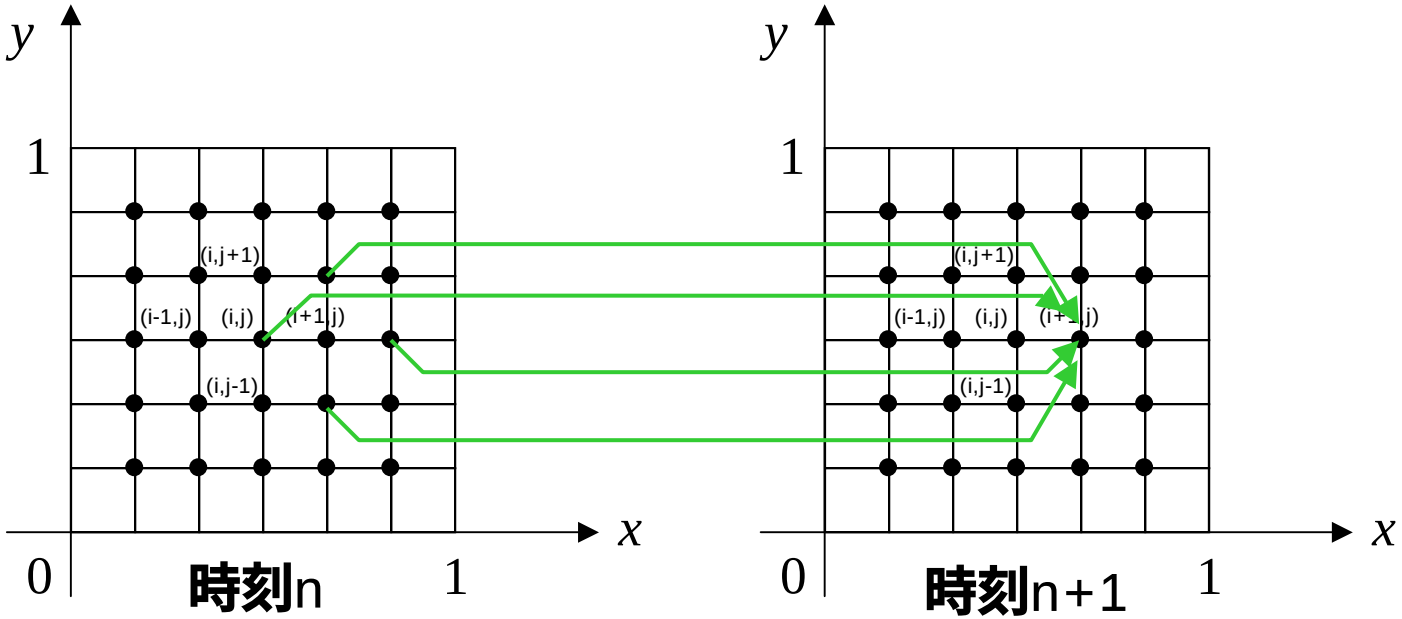


アルゴリズムと並列性

- **時間発展のアルゴリズム (ヤコビ法)**

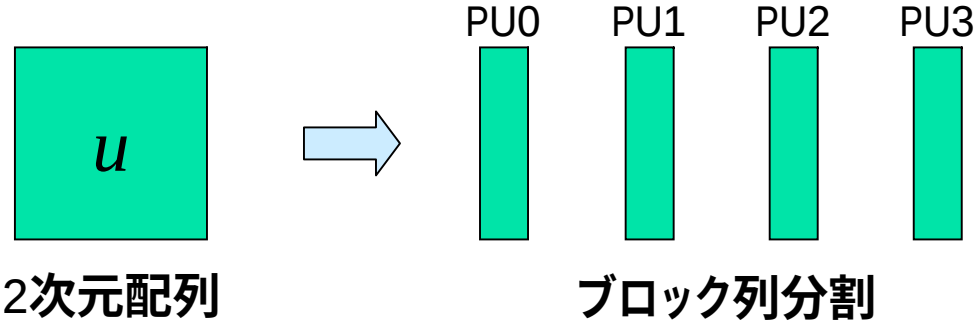
```
do j=1, m
   do i=1, m
      u_ij^(n+1) = (u_i-1,j^(n) + u_i+1,j^(n) + u_i,j-1^(n) + u_i,j+1^(n) ) / 4 + f_ij
   end do
end do
```

i, j について並列

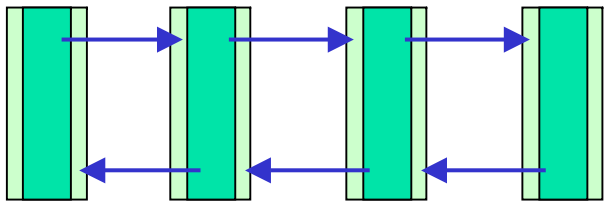


並列化方法

- データ分割
  - ブロック列分割とする



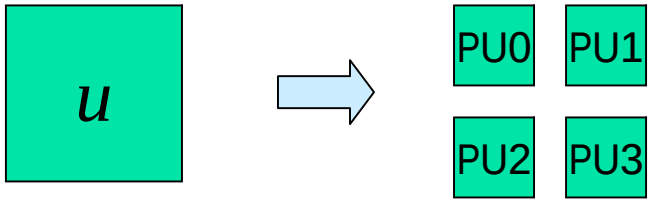
- 計算方法
  - ある点の計算には，1ステップ前における隣の4点での値が必要
  - PU境界の点の計算のため，各ステップの最初に，両隣からデータを送信してもらう



両隣のプロセスから1列を受信
（受信用の領域を確保しておく）

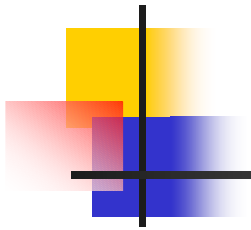
最適化

- 通信量の最小化
  - 通信量は，各PUの担当領域の周囲の長さに比例する
  - 2次元分割を行うことで，通信量を削減できる



  - 3次元の問題では，通信量削減がさらに重要

- 通信の隠蔽
  - まず境界点の値を隣接PUに送信してから，内点について時間発展の計算をし，最後に境界点の値を受信して計算を行うことで，通信時間をある程度隠蔽できる
  - ノンブロッキング型通信の mpi_isend，mpi_irecv を使う

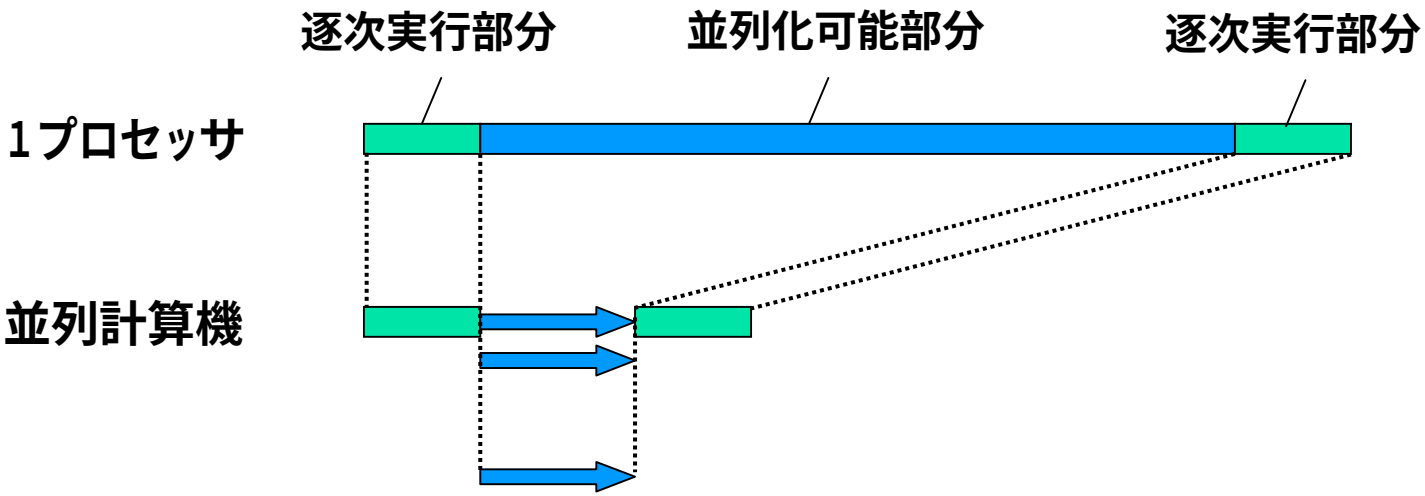


4. 並列性能の向上のために

- 並列処理による高速化
- アムダールの法則
- 並列性能の向上のために

並列処理による高速化

- 並列化可能部分と逐次実行部分
  - プログラム中で独立に実行可能な部分（並列化可能部分）を複数のプロセッサで分担することにより，実行時間を短縮
  - 並列化可能部分の割合が大きいほど，並列化の効果が大きい
  - 逐次実行部分の割合が大きいと，並列化の効果は小さい



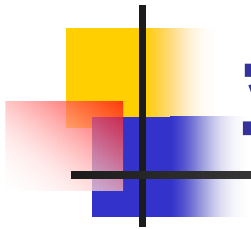
アムダールの法則

- 加速率と並列化効率
  - プロセッサ数が1，P のときの実行時間をそれぞれ T_1，T_P とするとき，T_1 / T_P を**加速率**と呼ぶ
  - $T_1 / (P * T_P)$ を**並列化効率**と呼ぶ

- アムダールの法則
  - プログラムの実行時間のうち，並列化可能部分が占める割合を q とすると，

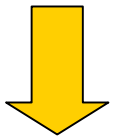
$$\text{加速率の上限} = 1/((1-q) + q/P)$$

  - 実際は，負荷不均衡やスレッド起動コストなどで，より低い値
  - 逐次実行部分が10%あると，10プロセッサを使っても最大5倍程度の加速率しか得られない



並列性能の向上のために

- 並列計算機の性能を引き出すプログラムを作るには，以下の点に関する考慮が重要
  - プロセッサ間の負荷分散
  - 並列粒度と同期回数
  - プロセッサ間通信の回数と量



- これらについては，以下の演習の中で順に説明してゆく