

# べき乗法

---

東京大学情報基盤センター 准教授 塙 敏博

2020年6月2日(火) 10:25-12:10

レポートおよびコンテスト課題  
(締切:  
2020年8月3日(月)24時 厳守

# 講義日程(工学部共通科目)

~~1. 4月14日 ガイダンス~~

~~2. 4月21日~~

- ~~● 並列数値処理の基本演算(座学)~~

~~3. 4月28日:スパコン利用開始~~

- ~~● ログイン作業、テストプログラム実行~~

~~4. 5月12日~~

- ~~● 高性能プログラミング技法の基礎1  
(階層メモリ、ループアンローリング)~~

~~5. 5月19日~~

- ~~● 高性能プログラミング技法の基礎2  
(キャッシュブロック化)~~

~~6. 5月26日~~

- ~~● 行列ベクトル積の並列化~~

7. 6月2日

- べき乗法の並列化

8. 6月9日

- 行列-行列積の並列化(1)

9. 6月16日

- 行列-行列積の並列化(2)

10. 6月23日

- LU分解法(1)
- コンテスト課題発表

11. 6月30日

- LU分解法(2)

12. 7月7日

- LU分解法(3)、非同期通信

13. 7月14日

- RB-Hお試し、研究紹介他

# 講義の流れ

1. べき乗法とは
2. べき乗法のサンプルプログラムの実行
3. サンプルプログラムの説明
4. 並列化実習
5. レポート課題

# べき乗法とは

---

簡単な数値アルゴリズム

# べき乗法とは

- べき乗法は、標準固有値問題の〈最大固有値〉と、それに付随する〈固有ベクトル〉を計算できます。
  - 標準固有値問題:  $Ax = \lambda x$
  - 固有値:  $\lambda$  固有ベクトル:  $x$
- いま、行列Aを  $n \times n$  の正方行列とします。
- 行列Aの固有値を、絶対値の大きい方から整列し、かつ重複していないものを  $\lambda_1, \lambda_2, \dots, \lambda_n$  とします。
- 正規直交なベクトルを  $x_1, x_2, \dots, x_n$  とします。
- このとき、任意のベクトルは、以下の線形結合で表わされます。

$$u = c_1 x_1 + c_2 x_2 + \dots + c_n x_n$$

# べき乗法とは

- Aを左辺に作用させると

$$Au = A(c_1x_1 + c_2x_2 + \dots + c_nx_n)$$

- さらに標準固有値問題の等式を考慮すると

$$Au = c_1\lambda_1x_1 + c_2\lambda_2x_2 + \dots + c_n\lambda_nx_n$$

$$= \lambda_1 \left[ c_1x_1 + c_2 \frac{\lambda_2}{\lambda_1} x_2 + \dots + c_n \frac{\lambda_n}{\lambda_1} x_n \right]$$

# べき乗法とは

- $Au$ の積を、 $k$ 回行くと

$$A^k u = \lambda_1^k \left[ c_1 x_1 + c_2 \left[ \frac{\lambda_2}{\lambda_1} \right]^k x_2 + \dots + c_n \left[ \frac{\lambda_n}{\lambda_1} \right]^k x_n \right]$$

- すなわち、 $k$ が増えていくと、段々  $x_1$  以外のベクトルの係数が小さくなっていく。  
→ 最大固有値、および、それに付随する固有ベクトルに収束する

# べき乗法とは

- 内積を  $(x, y)$  と記載する。このとき、以下の計算を考える。

$$\frac{(A^{k+1}u, A^{k+1}u)}{(A^{k+1}u, A^k u)} = \frac{\sum_{i=1}^n \sum_{j=1}^n c_i c_j \lambda_i^{k+1} \lambda_j^{k+1} (x_i, x_j)}{\sum_{i=1}^n \sum_{j=1}^n c_i c_j \lambda_i^{k+1} \lambda_j^k (x_i, x_j)}$$

$$= \frac{\lambda_1^{2k+2} \left[ c_1^2 |x_1|^2 + \sum_{i=2}^n c_i^2 \left[ \frac{\lambda_i}{\lambda_1} \right]^{2k+2} |x_i|^2 \right]}{\lambda_1^{2k+1} \left[ c_1^2 |x_1|^2 + \sum_{i=2}^n c_i^2 \left[ \frac{\lambda_i}{\lambda_1} \right]^{2k+1} |x_i|^2 \right]} \approx \lambda_1 \quad (k \rightarrow \infty)$$



# べき乗法のアルゴリズム

- 以下の手順を、収束するまで行う

1. 適当なベクトル $x$ を作り、正規化;
2.  $\lambda_0 = 0.0$ ;  $i=1$ ;
3. 行列積  $y = Ax$ ;
4. 近似固有値  $\lambda_i = (y, y) / (y, x)$  を計算;
5.  $|\lambda_i - \lambda_{i-1}|$  が十分小さいとき:
  - 収束したとみなし終了;
6. そうでないなら:
  - $y$ を正規化して  $x = y$ ;
  - $i = i + 1$ ; 3.へ戻る;

# サンプルプログラムの実行 (べき乗法)

---

はじめての数値アルゴリズムの並列化

# べき乗法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  
**PowM-ofp.tar.gz**
- ジョブスクリプトファイル**powm.bash** 中の  
キュー名を  
**lecture-flat** から  
**lecture5-flat** (工学部共通科目)  
に変更し、**pjsub** してください。
  - **lecture-flat** : 実習時間外のキュー
  - **lecture5-flat**: 実習時間内のキュー
- **グループも gt45**に変える

# べき乗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

```
$ cd /work/gt45/t45xxx
$ cp /work/gt45/z30105/PowM-ofp.tar.gz ./
$ tar xvfz PowM-ofp.tar.gz
$ cd PowM
```
- 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
```
- ジョブスクリプトを修正したら

```
$ pjsub powm.bash
```
- 実行が終了したら、以下を実行する

```
$ cat powm.bash.oXXXXXX
```

# べき乗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 = 4000

Power Method time = 1.209419 [sec.]

Eigenvalue = 2.000342e+03

Iteration Number: 11

Residual 2-Norm  $\|Ax - \lambda x\|_2 = 6.288935e-13$

N = 4000

Power Method time = 1.109105 [sec.]

Eigenvalue = 2.000342e+03

Iteration Number: 11

Residual 2-Norm  $\|Ax - \lambda x\|_2 = 6.288935e-13$

N = 4000

Power Method time = 0.264855 [sec.]

Eigenvalue = 2.000342e+03

Iteration Number: 11

Residual 2-Norm  $\|Ax - \lambda x\|_2 = 6.288935e-13$

# べき乗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

N = 4000

Power Method time[sec.] = 0.949132919311523

Eigenvalue = 1999.85535461023

Iteration Number: 7

Residual 2-Norm  $\|A x - \lambda x\|_2 = 7.495077627870977E-009$

N = 4000

Power Method time[sec.] = 1.05101490020752

Eigenvalue = 1999.77828254775

Iteration Number: 9

Residual 2-Norm  $\|A x - \lambda x\|_2 = 3.636552038660712E-012$

N = 4000

Power Method time[sec.] = 0.335184812545776

Eigenvalue = 2000.24938906580

Iteration Number: 10

Residual 2-Norm  $\|A x - \lambda x\|_2 = 3.146643327947337E-012$

# サンプルプログラムの説明

- `#define N 4000`  
の、数字を変更すると、行列サイズが変更できます
- **PowM関数の仕様**
  - 戻り値は、最大固有値 (Double型)
  - Double型の配列xに、最大固有値に付随する固有ベクトルが格納される。
  - 引数n\_iterに収束するまでの反復回数が入る。
    - “-1”が戻る場合、反復回数の上限MAX\_ITERまでに収束しなかったことを意味する。

# Fortran言語のサンプルプログラムの注意

- 行列サイズ変数が、NNとなっています。  
`integer, parameter :: NN=4000`



# サンプルプログラムの概略 (PowM関数内)

```

/* Normization of x */
d_tmp1 = 0.0;
for(i=0; i<n; i++) {
    d_tmp1 += x[i] * x[i];
}
d_tmp1 = 1.0 / sqrt(d_tmp1);
for(i=0; i<n; i++) {
    x[i] = x[i] * d_tmp1;
}

```

ベクトルx  
正規化部分

```

/* Main iteration loop ----- */
for(i_loop=1; i_loop<MAX_ITER; i_loop++) {

```

```

/* Matrix Vector Product */
MyMatVec(y, A, x, n);

```

行列-ベクトル積  
部分

```

/* inner products */
d_tmp1 = 0.0;
d_tmp2 = 0.0;
for (i=0; i<n; i++) {
    d_tmp1 += y[i] * y[i];
    d_tmp2 += y[i] * x[i];
}

```

行列xとyの  
内積部分

```

/* current approximately eigenvalue */
dlambda = d_tmp1 / d_tmp2;

```

```

/* Convergence test*/
if (fabs(d_before-dlambda) < EPS ) {
    *n_iter = i_loop;
    return dlambda;
}

```

```

/* keep current value */
d_before = dlambda;

```

```

/* Normalization and set new x */
d_tmp1 = 1.0 / sqrt(d_tmp1);
for(i=0; i<n; i++)
    x[i] = y[i] * d_tmp1;

```

正規化と  
新しいx  
設定部分

```

} /* end of i_loop ----- */

```

# 演習課題

- PowM関数(手続き)を並列化してください。
  - デバック時は、`#define N 2176` としてください。
  - 前回演習の並列行列-ベクトル積ルーチンを利用してください。
- サンプルプログラムでは、残差ベクトル $Ax - \lambda x$ の2-ノルムを計算しています。デバックに活用してください。
  - つまり、この値が十分小さくないとバグっています。
  - 固有ベクトル $x$ の分散方式により、残差ベクトル計算部分の並列化が必要になります。注意してください。
- 並列化すると、反復回数(=実行時間)や残差の2ノルム値が変化することがあります。

# 並列化の注意

- 以下のようなプログラムを書くと、美しくない&コードマネージが大変になる。

同じプログラム

```
if (myid == numprocs-1) {
  for (j=myid*ib; j<n; j++) {
    for (...) {
      ...
    }
  }
} else {
  for (j=myid*ib; j<(myid+1)*ib; j++) {
    for (...) {
      ...
    }
  }
}
```

- 並列化の対象のループは1つにし、ループ制御変数を工夫し、並列化するように心がける。

# 並列化のヒント

- 前回示した方針のように、すべてのPEで重複して、行列Aを $N \times N$ 、ベクトル $x, y$ を $N$ のサイズで確保すると、実装が簡単です。
- 以下の分散方式を仮定します。  
(先週の行列-ベクトル積の演習と同じ)
  - 行列A:  
1次元行方向ブロック分割方式
  - ベクトル $x$ :  
全PEで、 $N$ 次元ベクトルを重複所有
  - ベクトル $y$ :  
ブロック分割方式

# 並列化のヒント( 1.「行列-ベクトル積」のみ)

- 以下の2通りの＜並列化方針＞があります
  - 方法1: 「行列-ベクトル積」のみ並列化
  - 方法2: すべてを並列化
- **最も簡単な方法は方法1。**以下はその手順:

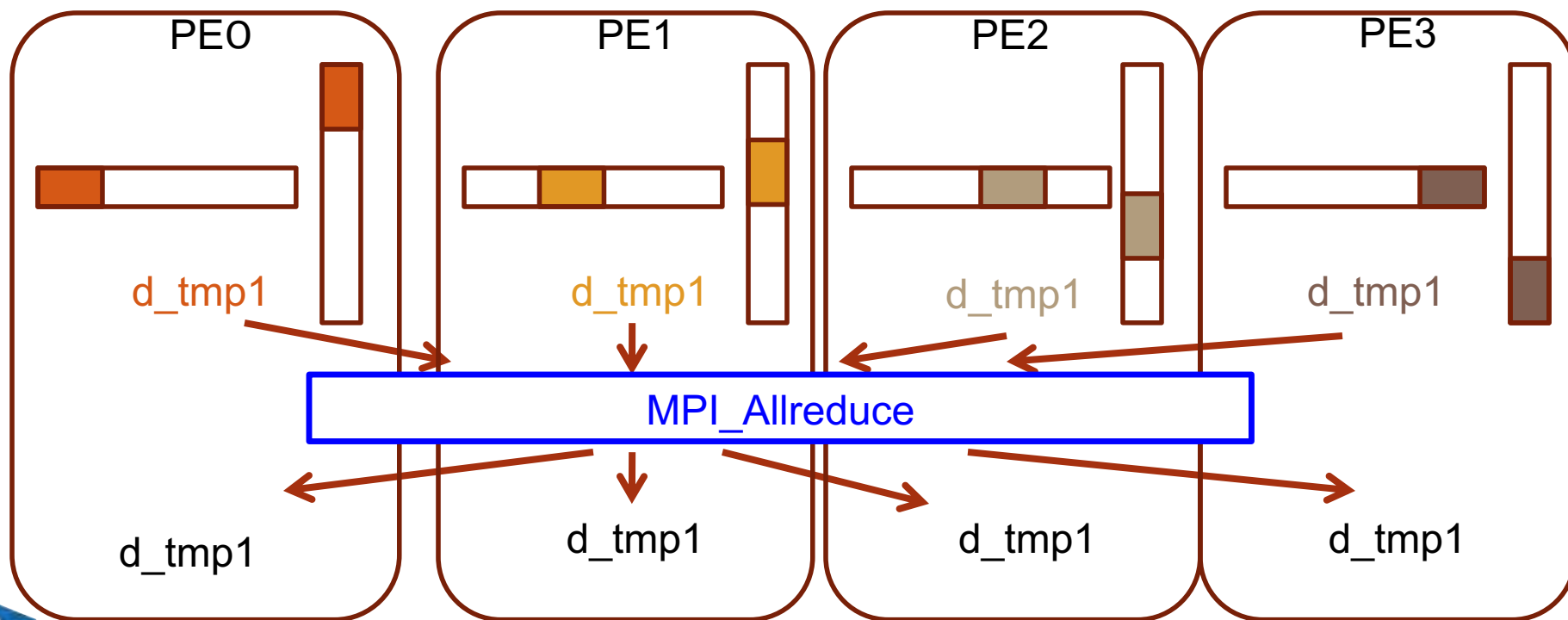
1. 開発した「並列行列-ベクトル積」コードを使う
2.  $y = Ax$  の  $y$  が分散されて戻るため、以降の計算が逐次の結果と合わない。逐次結果と一致させるため、
  - MPI関数を **PowM関数中の MyMatVec()** が呼ばれる直後に入れて、分散された  $y$  の要素すべてを収集する。
  - 最も簡単な実装は、MPI\_Allreduce() を用いる実装
3. **MPI\_Allreduce()**を利用するため、配列の初期化(ゼロクリア)を実装する。(後述の方式)

# 並列化のヒント(2. すべてを並列化時)

- PowM関数中の処理を、以下の方針で並列化

1. ベクトルxの正規化部分

- ① ブロック分割の内積計算を計算後、MPI\_Allreduce関数(下図)を呼ぶ
- ② MPI\_Allreduce関数を使って、部分的に計算された計算結果を、全PEが全ベクトル要素を所有するようにする(後述)



# 並列化のヒント(2.すべてを並列化時)

- 以下のようなプログラムになる

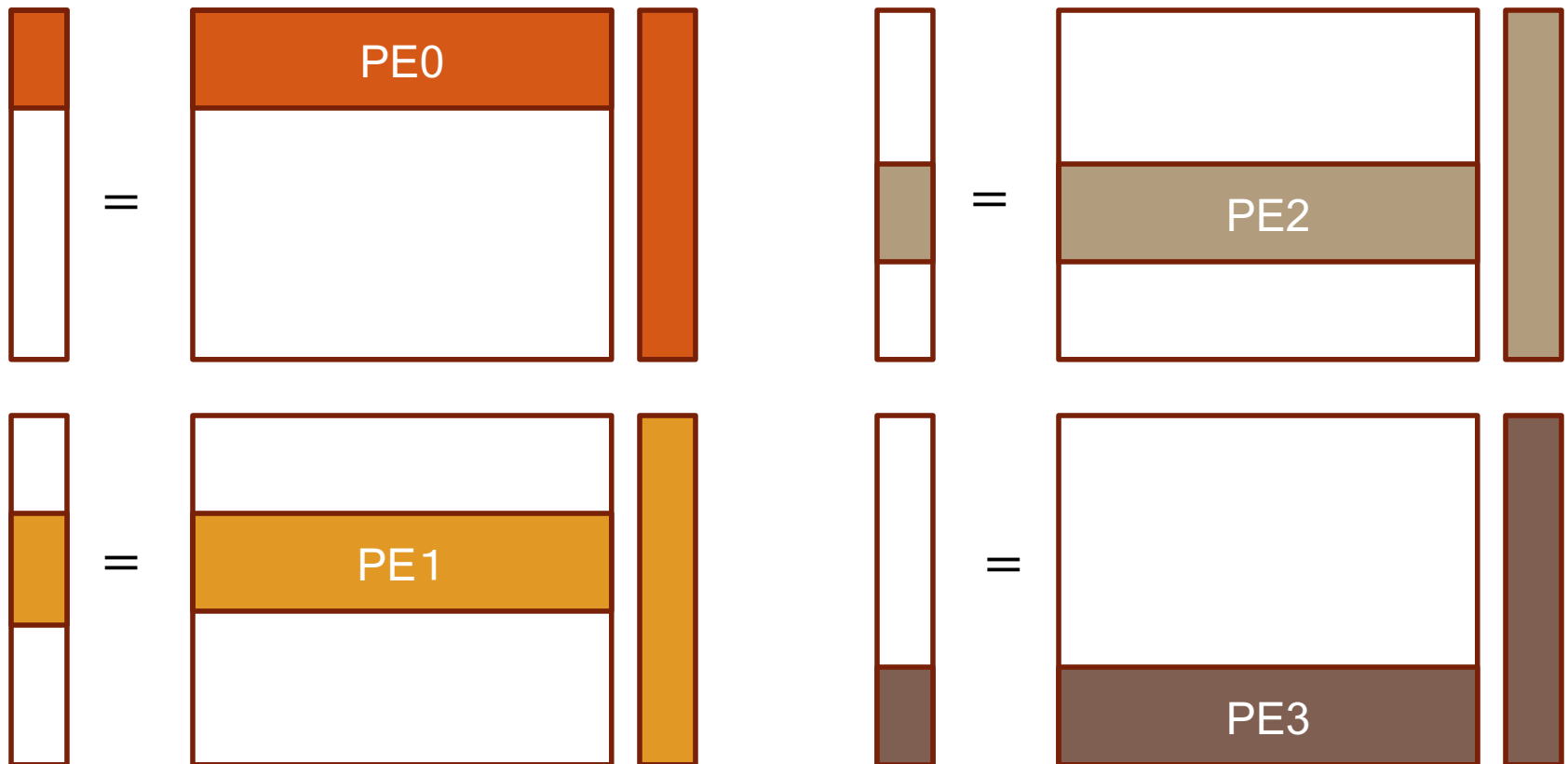
```
/* Normalization of x */
...
d_tmp1_t = 0.0;
for(i=myid*ib; i<i_end; i++) {
    d_tmp1_t += x[i] * x[i];
}
MPI_Allreduce(&d_tmp1_t, &d_tmp1, 1, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);

d_tmp1 = 1.0 / sqrt(d_tmp1);
for(i=myid*ib; i<i_end; i++) {
    x_t[i] = x[i] * d_tmp1;
}
(x_t[]は、ゼロに初期化をしているか要確認)
MPI_Allreduce(x_t, x, n, MPI_DOUBLE, MPI_SUM,
    MPI_COMM_WORLD);
....
```

# 並列化のヒント(方法1および方法2)

## 2. 行列-ベクトル積部分(MyMatVec関数)

- 前回演習の並列ルーチンを使う





# 並列化のヒント(方法1および方法2)

## 3. ベクトル $x$ と $y$ の内積部分

- ブロック分散されているとして計算する
- 正しい内積結果を得るため、`MPI_Allreduce`関数を使うことを忘れずに

# 並列化のヒント(2.すべてを並列化時)

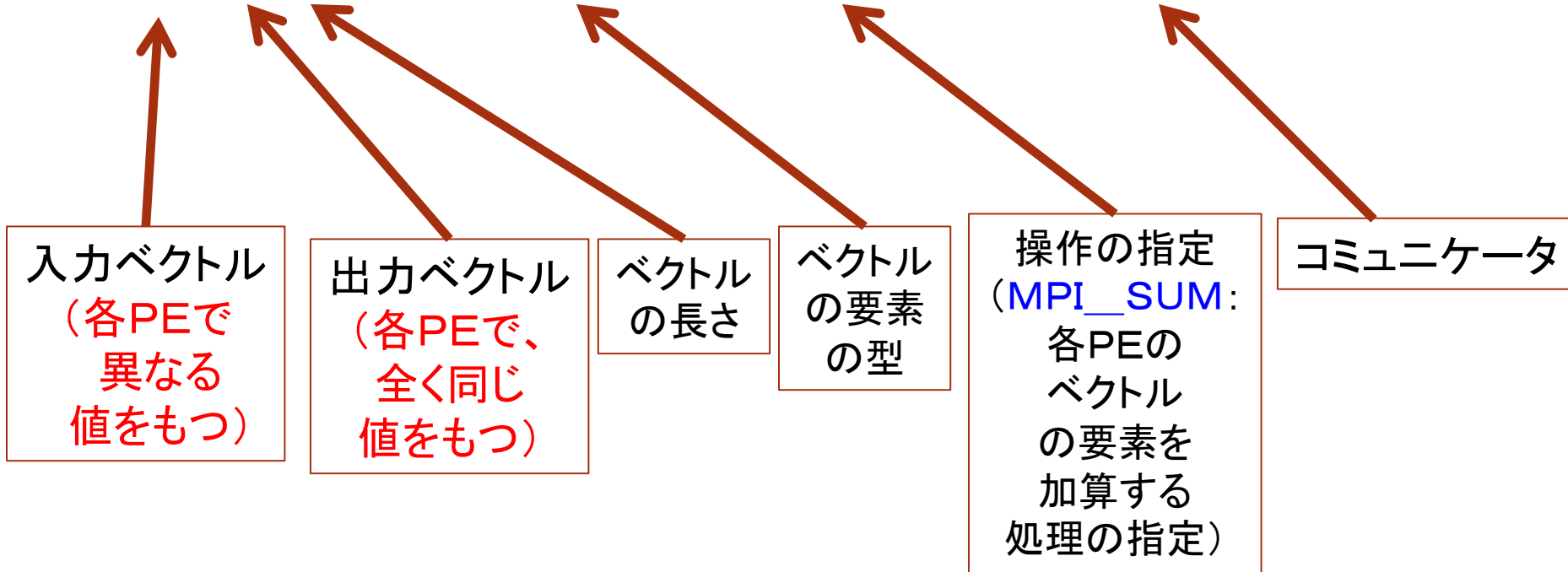
## 4. 正規化と新しいxの設定部分

- x: 全PEで同じN次元ベクトルを所有; y: ブロック分散
- 正規化はブロック分散部分のみを行い、xに結果を代入
- xは、各PEで計算結果が分散されている。  
(xは計算結果がブロック分散)
- xは、全PEで全要素を重複して所有していないと、次の並列行列-ベクトル積が実行できない。
- 各PEに分散されているベクトルxのデータを集めるため、**MPI\_Allreduce** 関数を使って集める (後述)。
  - **MPI\_Allreduce**関数を使うため、xの計算結果部分以外に0を代入したバッファx\_tを用意。  
`MPI_Allreduce( x_t, x, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );`

# MPI\_Allreduce関数の復習(C言語)

## • MPI\_Allreduce

```
(x_t, x, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

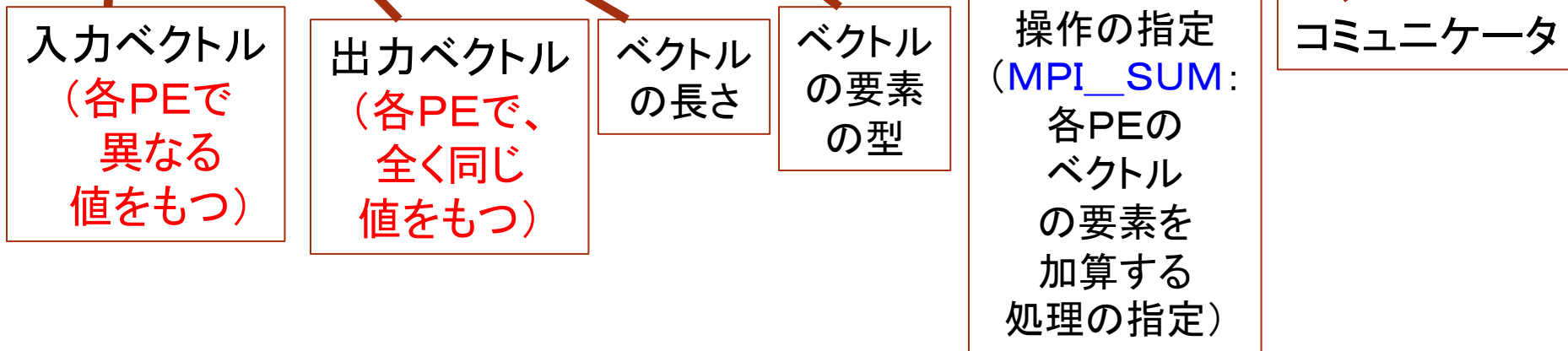


# MPI\_Allreduce関数の復習 (Fortran言語)

## • MPI\_ALLREDUCE

(x\_t, x, n, MPI\_DOUBLE\_PRECISION, MPI\_SUM, MPI\_COMM\_WORLD, ierr)

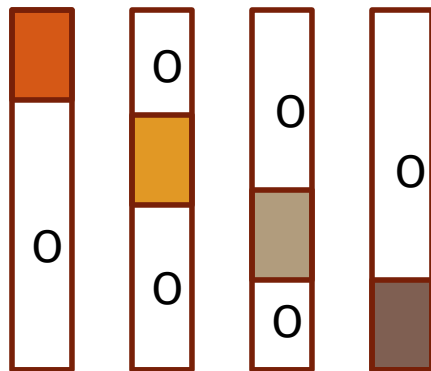
- MPI\_DOUBLE\_PRECISIONの代わりに MPI\_REAL8 でもよい



# MPIの技(MPI\_Allreduceでベクトル収集)

- `MPI_Allreduce`関数で、各PEに分散されたデータを収集し、全PEに演算結果を<重複して>所有させる方法
  - `iop` に、`MPI_SUM` を指定する
  - 自分が所有するデータ以外の箇所は、0に初期化されている。
  - そのうえで、以下のような処理を考える

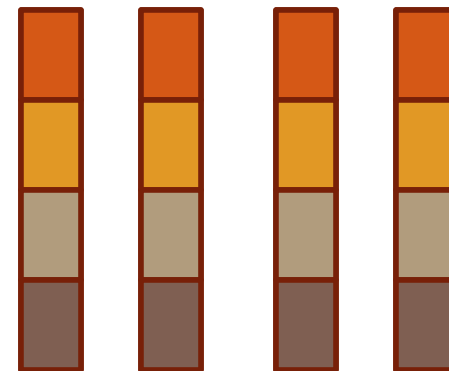
初期状態



PE0 PE1 PE2 PE3

終了状態

`MPI_Allreduce`  
`(..,MPI_SUM,..);`



PE0 PE1 PE2 PE3

※`MPI_Allgather`関数を使っても実装できます。

# 実習の手順

- はじめに、簡単な  
方法1:「行列-ベクトル積」のみ並列化  
を先に行ってください。
- それが終わったら、  
方法2:すべてを並列化  
を行ってください。

# レポート課題

1. [L15] サンプルプログラムを並列化せよ。このとき、行列Aおよびベクトルx、yのデータは、全PEで $N \times N$ のサイズを確保してよい。なお、いろいろな問題サイズ(Nの大きさ)について性能評価し、その結果の考察を行え。
2. [L20] サンプルプログラムを並列化し、性能評価と考察を行え。このとき、行列Aおよびベクトルx、yは、初期状態では各PEに割り当てられた分の領域しか確保してはいけない。また、1と同様な性能評価と考察を行え。特に、  
1. と2. で実行時間の違いはあるか評価・考察せよ。

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(続き)

4. [L5] コンパイラによる最適化により、実行時間がどのように変化するか調査せよ。コンパイラの最適化方式により、反復回数が増えることがある。そこで、1反復あたりの実行時間を計算した上で、性能評価を行い考察せよ。
5. [L20] サンプルプログラムの並列化プログラムについて、通信処理をノンブロッキングにするなどの改良して高速化を行え。いろいろな問題サイズについて性能評価を行い、高速化する前のプログラムに対して考察をせよ。
6. [L10~L20] 並列化したプログラムに対し、ピュアMPI実行、および、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。(L10)  
また、どういう条件でピュアMPI実行が高速になるか、条件を導出し、性能評価結果と検証を行え。(L20)



# 次回へつづく

---

行列-行列積の並列化