

内容に関する質問は
katagiri@cc.u-tokyo.ac.jp
まで

第2回 MPIの基礎

東京大学情報基盤センター 片桐孝洋

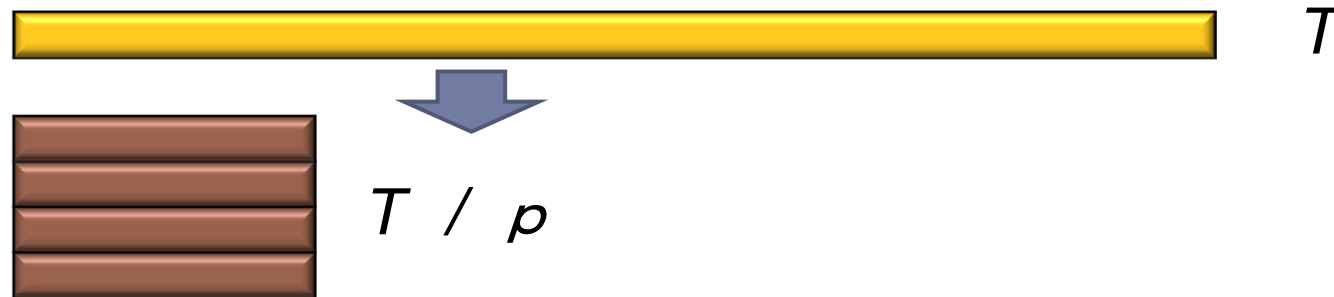
講義日程と内容について

- ▶ **2013年度 計算科学技術特論A(1学期:木曜3限)**
 - ▶ 第1回:プログラム高速化の基礎、2013年4月11日
 - ▶ イントロダクション、ループアンローリング、キャッシュブロック化、数値計算ライブラリの利用、その他
 - ▶ **第2回:MPIの基礎、2013年4月18日**
 - ▶ 並列処理の基礎、MPIインターフェース、MPI通信の種類、その他
 - ▶ 第3回:OpenMPの基礎、2013年4月25日
 - ▶ OpenMPの基礎、利用方法、その他
 - ▶ 第4回:Hybrid並列化技法(MPIとOpenMPの応用)、2013年5月9日
 - ▶ 背景、Hybrid並列化の適用事例、利用上の注意、その他
 - ▶ 第5回:プログラム高速化の応用、2013年5月16日
 - ▶ プログラムの性能ボトルネックに関する考えかた(I/O、単体性能(演算機ネック、メモリネック)、並列性能(バランス))、性能プロファイル、その他

並列プログラミングの基礎

並列プログラミングとは何か？

- ▶ 逐次実行のプログラム(実行時間 T)を、 p 台の計算機を使って、 T/p にすること.

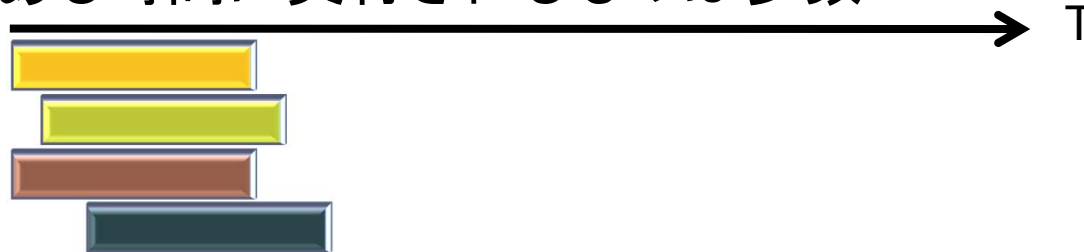


- ▶ 素人考えでは自明。
- ▶ 実際は、できるかどうかは、対象処理の内容(アルゴリズム)で **大きく** 難しさが違う
 - ▶ アルゴリズム上、絶対に並列化できない部分の存在
 - ▶ 通信のためのオーバヘッドの存在
 - ▶ 通信立ち上がり時間
 - ▶ データ転送時間

並列と並行

▶ 並列 (Parallel)

- ▶ 物理的に並列 (時間的に独立)
- ▶ ある時間に実行されるものは多数



▶ 並行 (Concurrent)

- ▶ 論理的に並列 (時間的に依存)
- ▶ ある時間に実行されるものは1つ (= 1プロセッサで実行)



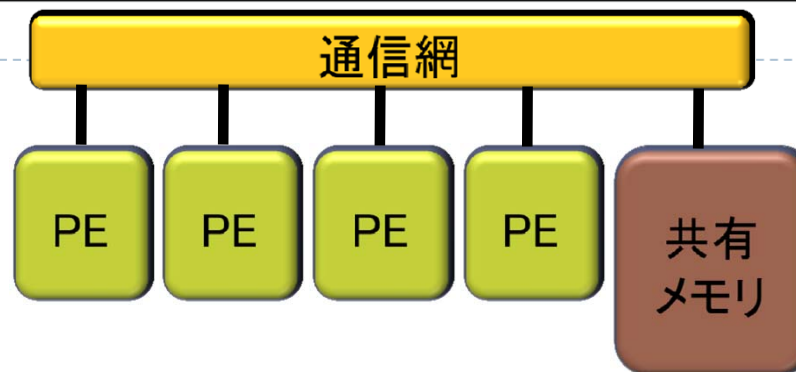
- ▶ 時分割多重、疑似並列
- ▶ OSによるプロセス実行スケジューリング (ラウンドロビン方式)

並列計算機の種類

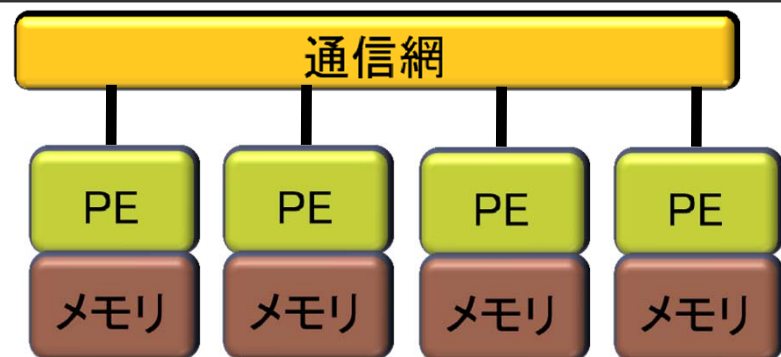
- ▶ Michael J. Flynn教授(スタンフォード大)の種類(1966)
- ▶ 単一命令・単一データ流
(SISD, Single Instruction Single Data Stream)
- ▶ 単一命令・複数データ流
(SIMD, Single Instruction Multiple Data Stream)
- ▶ 複数命令・単一データ流
(MISD, Multiple Instruction Single Data Stream)
- ▶ 複数命令・複数データ流
(MIMD, Multiple Instruction Multiple Data Stream)

並列計算機のメモリ型による分類

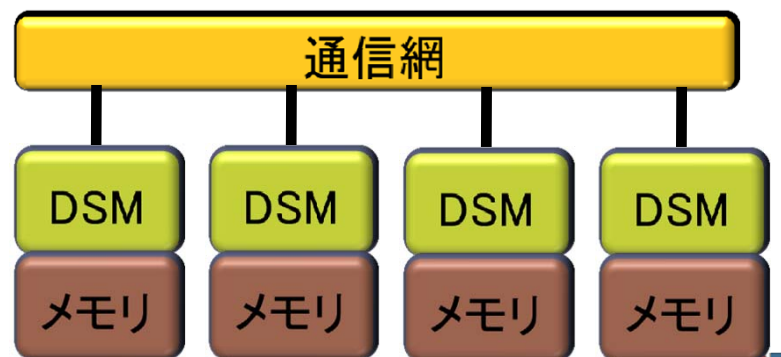
1. 共有メモリ型
(SMP、Symmetric Multiprocessor)



2. 分散メモリ型
(メッセージパッシング)

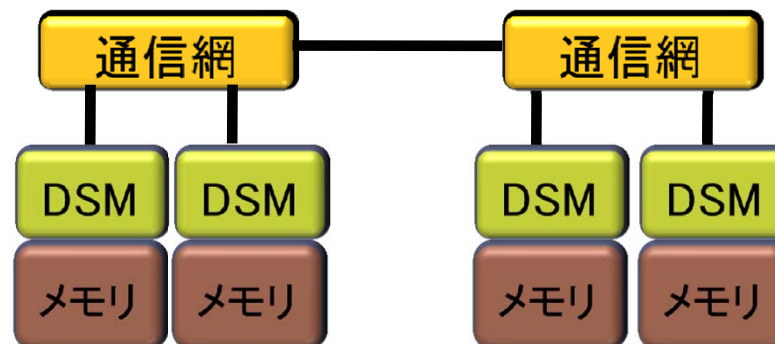


3. 分散共有メモリ型
(DSM、Distributed Shared Memory)



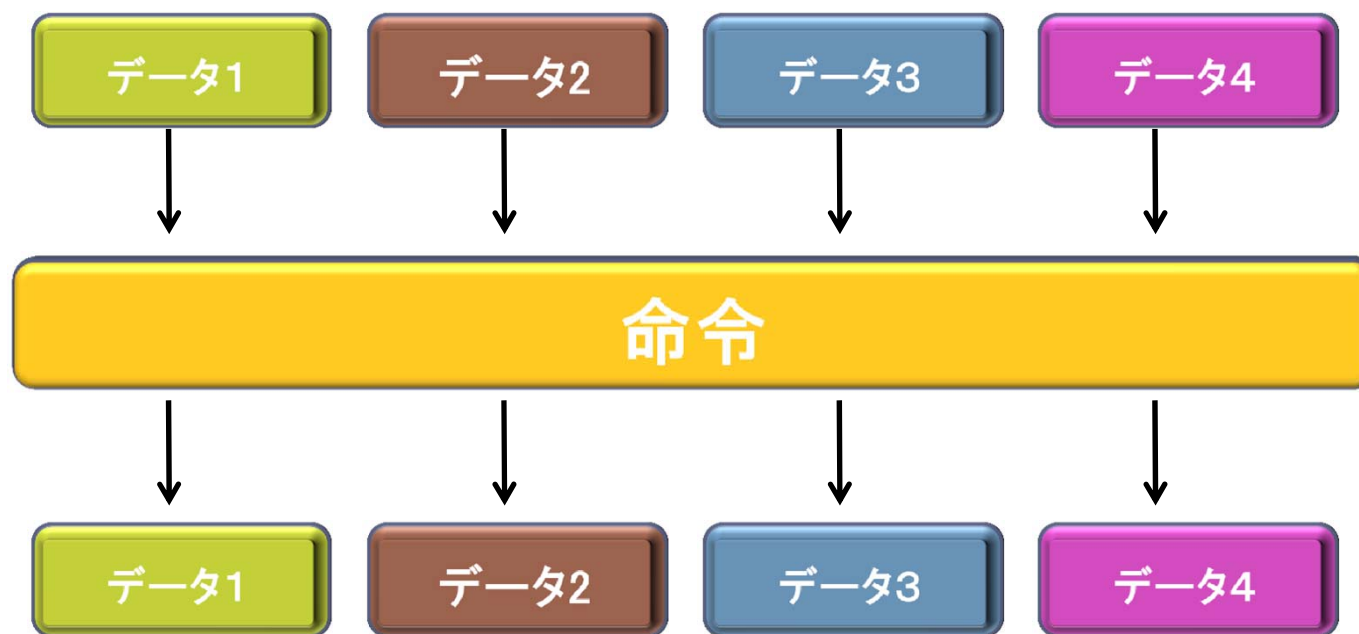
並列計算機のメモリ型による分類

4. 共有・非対称メモリ型
(ccNUMA、
Cache Coherent Non-
Uniform Memory Access)



並列プログラミングのモデル

- ▶ 実際の並列プログラムの挙動はMIMD
- ▶ アルゴリズムを考えるときは<SIMDが基本>
- ▶ 複雑な挙動は理解できないので



並列プログラミングのモデル

▶ MIMD上での並列プログラミングのモデル

1. SPMD (Single Program Multiple Data)

- ▶ 1つの共通のプログラムが、並列処理開始時に、全プロセッサ上で起動する
- ▶ MPI(バージョン1)のモデル



2. Master / Worker (Master / Slave)

- ▶ 1つのプロセス (Master) が、複数のプロセス (Worker) を管理 (生成、消去) する。

並列プログラムの種類

▶ マルチプロセス

- ▶ MPI (Message Passing Interface)
- ▶ HPF (High Performance Fortran)
 - ▶ 自動並列化Fortranコンパイラ
 - ▶ ユーザがデータ分割方法を明示的に記述

プロセスとスレッドの違い

- メモリを意識するかどうかの違い
- 別メモリは「プロセス」
- 同一メモリは「スレッド」

▶ マルチスレッド

- ▶ Pthread (POSIX スレッド)
- ▶ Solaris Thread (Sun Solaris OS用)
- ▶ NT thread (Windows NT系、Windows95以降)
 - ▶ スレッドの Fork(分離) と Join(融合) を明示的に記述
- ▶ Java
 - ▶ 言語仕様としてスレッドを規定
- ▶ Open MP
 - ▶ ユーザが並列化指示行を記述

並列処理の実行形態 (1)

▶ データ並列

- ▶ データを分割することで並列化する。
- ▶ データの操作(=演算)は同一となる。
- ▶ データ並列の例: 行列-行列積

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$$

● 並列化

全CPUで共有

CPU0	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$	$=$	$\begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \end{pmatrix}$
CPU1	$\begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$			$\begin{pmatrix} 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \end{pmatrix}$
CPU2	$\begin{pmatrix} 7 & 8 & 9 \end{pmatrix}$			$\begin{pmatrix} 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$

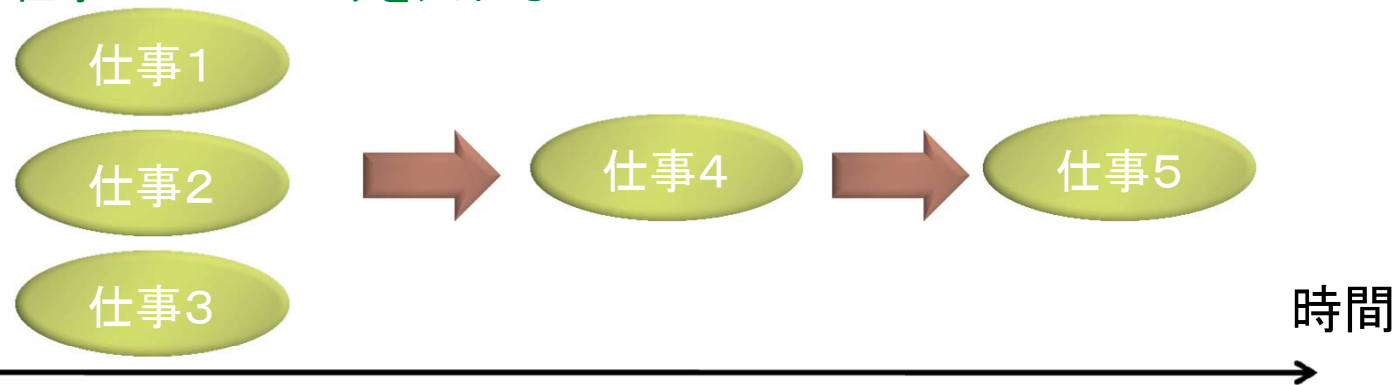
並列に計算: 初期データは異なるが演算は同一

並列処理の実行形態（2）

▶ タスク並列

- ▶ タスク(ジョブ)を分割することで並列化する。
- ▶ データの操作(=演算)は異なるかもしれない。
- ▶ タスク並列の例: **カレーを作る**
 - ▶ 仕事1: 野菜を切る
 - ▶ 仕事2: 肉を切る
 - ▶ 仕事3: 水を沸騰させる
 - ▶ 仕事4: 野菜・肉を入れて煮込む
 - ▶ 仕事5: カレールーを入れる

● 並列化



MPIの特徴

- ▶ **メッセージパッシング用のライブラリ規格の1つ**
 - ▶ メッセージパッシングのモデルである
 - ▶ コンパイラの規格、特定のソフトウェアやライブラリを指すものではない！
- ▶ **分散メモリ型並列計算機で並列実行に向く**
- ▶ **大規模計算が可能**
 - ▶ 1プロセッサにおけるメモリサイズやファイルサイズの制約を打破可能
 - ▶ プロセッサ台数の多い並列システム(MPPシステム、Massively Parallel Processingシステム)を用いる実行に向く
 - ▶ 1プロセッサ換算で膨大な実行時間の計算を、短時間で処理可能
 - ▶ 移植が容易
 - ▶ **API(Application Programming Interface)の標準化**
- ▶ **スケーラビリティ、性能が高い**
 - ▶ 通信処理をユーザが記述することによるアルゴリズムの最適化が可能
 - ▶ プログラミングが難しい(敷居が高い)

MPIの経緯 (1/2)

- ▶ MPIフォーラム (<http://www.mpi-forum.org/>) が仕様策定
 - ▶ 1994年5月1.0版 (MPI-1)
 - ▶ 1995年6月1.1版
 - ▶ 1997年7月1.2版、および 2.0版 (MPI-2)
- ▶ 米国アルゴンヌ国立研究所、およびミシシッピ州立大学で開発
- ▶ MPI-2 では、以下を強化：
 - ▶ 並列I/O
 - ▶ C++、Fortran 90用インターフェース
 - ▶ 動的プロセス生成/消滅
 - ▶ 主に、並列探索処理などの用途
 - ▶ 片方向通信、RMA (Remote Memory Access)

MPIの経緯 (2/2) MPI3.0策定 (3.1/4.0も策定中)

- ▶ 以下のページで経緯・ドキュメントを公開中
 - ▶ http://meetings.mpi-forum.org/MPI_3.0_main_page.php
 - ▶ <http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf> (MPI 3.0, Released September 21, 2012)
- ▶ 注目すべき機能
 - ▶ RMA (Remote Memory Access) サポート
 - ▶ FT (Fault Tolerant) stabilization、FT を意識したMPIプログラム作成のAPI群とセマンティクス
 - ▶ MPIT (Performance Tool)、デバッガやパフォーマンスモニタ、開発環境といったツール群とのインターフェース仕様

MPIの経緯 (2/2) MPI3.0策定 (3.1/4.0も策定中)

▶ 注目すべき機能(つづき)

- ▶ Neighborhood collectives(物理的に近いノードを対象にした専用コレクティブ通信), 粗通信モデルの為の集団通信API
- ▶ Fortran bindings – improved Fortran bindings, taking into account Fortran 2003 and 2008 features.
- ▶ Non-blocking collective I/O – extend non-blocking collective support to include MPI-I/O
- ▶ Hybrid: Shared memory communicator, Threads, and Endpoint proposals

MPIの実装

- ▶ **MPICH(エム・ピッチ)**
 - ▶ 米国アルゴンヌ国立研究所が開発
- ▶ LAM(Local Area Multicomputer)
 - ▶ ノートルダム大学が開発
- ▶ その他
 - ▶ OpenMPI (FT-MPI, LA-MPI, LAM/MPI, PACX-MPIの統合プロジェクト)
 - ▶ YAMPII(東大・石川研究室)(SCore通信機構をサポート)
 - ▶ 注意点:メーカー独自機能拡張がなされていることがある

MPIによる通信

- ▶ 郵便物の郵送と同じ
- ▶ 郵送に必要な情報:
 1. 自分の住所、送り先の住所
 2. 中に入っているものはどこにあるか
 3. 中に入っているものの分類
 4. 中に入っているものの量
 5. (荷物を複数同時に送る場合の)認識方法(タグ)
- ▶ MPIでは:
 1. 自分の認識ID、および、送り先の認識ID
 2. データ格納先のアドレス
 3. データ型
 4. データ量
 5. タグ番号

MPI関数

▶ システム関数

- ▶ MPI_Init; MPI_Comm_rank; MPI_Comm_size; MPI_Finalize;

▶ 1対1通信関数

▶ ブロッキング型

- ▶ MPI_Send; MPI_Recv;

▶ ノンブロッキング型

- ▶ MPI_Isend; MPI_Irecv;

▶ 1対全通信関数

- ▶ MPI_Bcast

▶ 集団通信関数

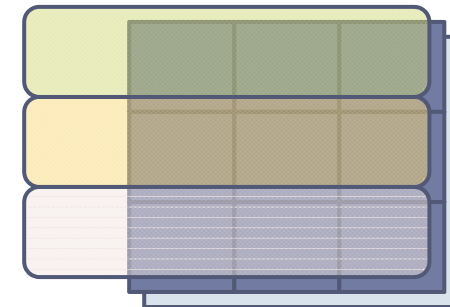
- ▶ MPI_Reduce; MPI_Allreduce; MPI_Barrier;

▶ 時間計測関数

- ▶ MPI_Wtime

コミュニケータ

- ▶ MPI_COMM_WORLDは、コミュニケータとよばれる概念を保存する変数
- ▶ コミュニケータは、操作を行う対象のプロセッサ群を定める
- ▶ 初期状態では、0番～ $\text{numprocs} - 1$ 番までのプロセッサが、1つのコミュニケータに割り当てられる
 - ▶ この名前が、“MPI_COMM_WORLD”
- ▶ プロセッサ群を分割したい場合、MPI_Comm_split 関数を利用
 - ▶ メッセージを、一部のプロセッサ群に放送するとき利用
 - ▶ “マルチキャスト”で利用



性能評価指標

並列化の尺度

性能評価指標－台数効果

▶ 台数効果

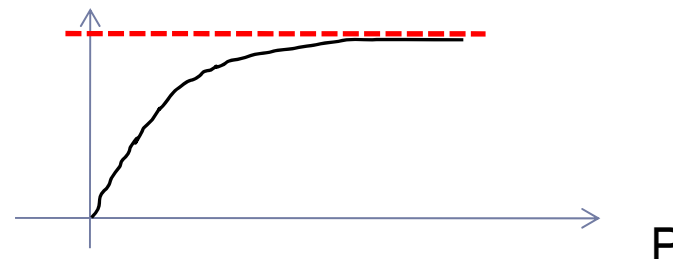
- ▶ 式: $S_p = T_S / T_p$ ($0 \leq S_p$)
- ▶ T_S : 逐次の実行時間、 T_p : P台での実行時間
- ▶ P台用いて $S_p = P$ のとき、理想的な(ideal)速度向上
- ▶ P台用いて $S_p > P$ のとき、スーパーニア・スピードアップ
 - ▶ 主な原因は、並列化により、データアクセスが局所化されて、キャッシュヒット率が向上することによる高速化

▶ 並列化効率

- ▶ 式: $E_p = S_p / P \times 100$ ($0 \leq E_p$) [%]

▶ 飽和性能

- ▶ 速度向上の限界
- ▶ Saturation、「さちる」



アムダールの法則

- ▶ 逐次実行時間を K とする。
そのうち、並列化ができる割合を α とする。
- ▶ このとき、台数効果は以下のようなになる。

$$S_p = K / (K\alpha / P + K(1-\alpha))$$
$$= 1 / (\alpha / P + (1-\alpha)) = 1 / (\alpha(1/P - 1) + 1)$$

- ▶ 上記の式から、たとえ無限大の数のプロセッサを使っても ($P \rightarrow \infty$)、台数効果は、高々 $1 / (1 - \alpha)$ である。

(アムダールの法則)

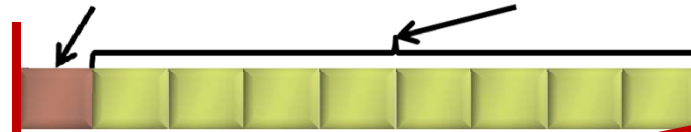
- ▶ 全体の90%が並列化できたとしても、無限大の数のプロセッサをつかっても、 $1 / (1 - 0.9) = 10$ 倍 にしかない！

→ 高性能を達成するためには、少しでも並列化効率を上げる
実装をすることがとても重要である

アムダールの法則の直観例

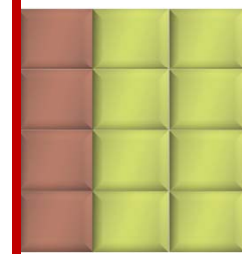
並列化できない部分(1ブロック) 並列化できる部分(8ブロック)

● 逐次実行



= 88.8%が並列化可能

● 並列実行(4並列)



$9/3=3$ 倍

● 並列実行(8並列)

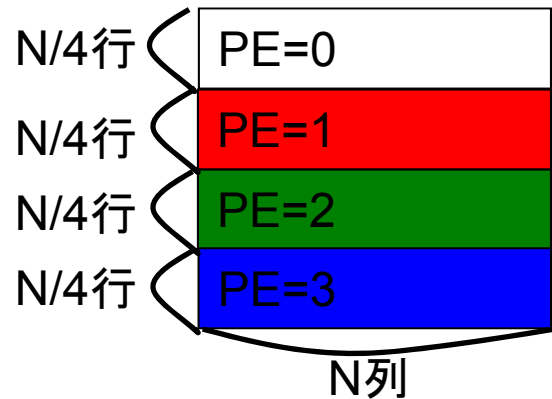


$9/2=4.5$ 倍 \neq 6倍

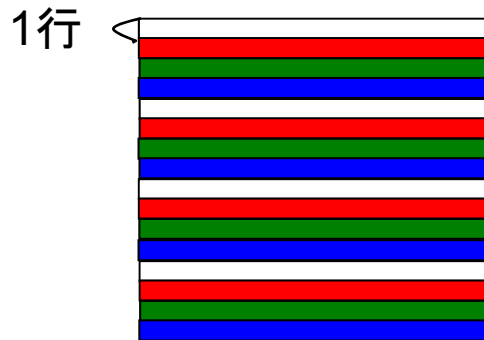
基本演算

- ▶ 逐次処理では、「データ構造」が重要
- ▶ 並列処理においては、「データ分散方法」が重要になる！
 1. 各PEの「演算負荷」を均等にする
 - ▶ ロード・バランシング： 並列処理の基本操作の一つ
 - ▶ 粒度調整
 2. 各PEの「利用メモリ量」を均等にする
 3. 演算に伴う通信時間を短縮する
 4. 各PEの「データ・アクセスパターン」を高速な方式にする
(=逐次処理におけるデータ構造と同じ)
- ▶ 行列データの分散方法
 - ▶ <次元レベル>： 1次元分散方式、2次元分散方式
 - ▶ <分割レベル>： ブロック分割方式、サイクリック(循環)分割方式

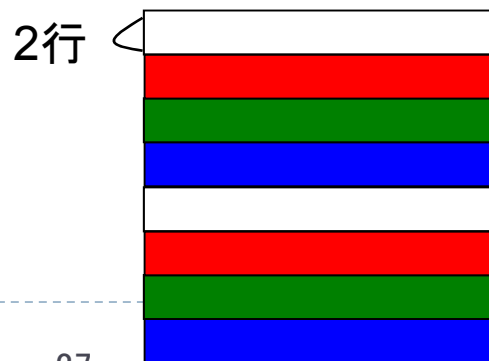
1次元分散



- (行方向) ブロック分割方式
- (Block, *) 分散方式



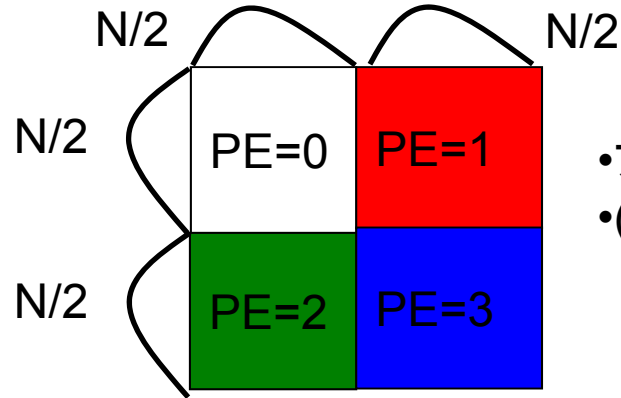
- (行方向) サイクリック分割方式
- (Cyclic, *) 分散方式



- (行方向)ブロック・サイクリック分割方式
- (Cyclic(2), *) 分散方式

この例の「2」: <ブロック幅>とよぶ

2次元分散



- ブロック・ブロック分割方式
- (Block, Block)分散方式

- サイクリック・サイクリック分割方式
- (Cyclic, Cyclic)分散方式

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3

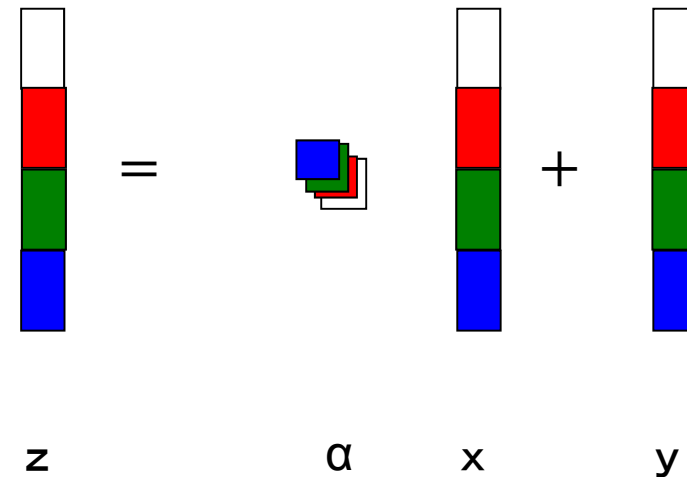
- 二次元ブロック・サイクリック分割方式
- (Cyclic(2), Cyclic(2))分散方式

ベクトルどうしの演算

- ▶ 以下の演算

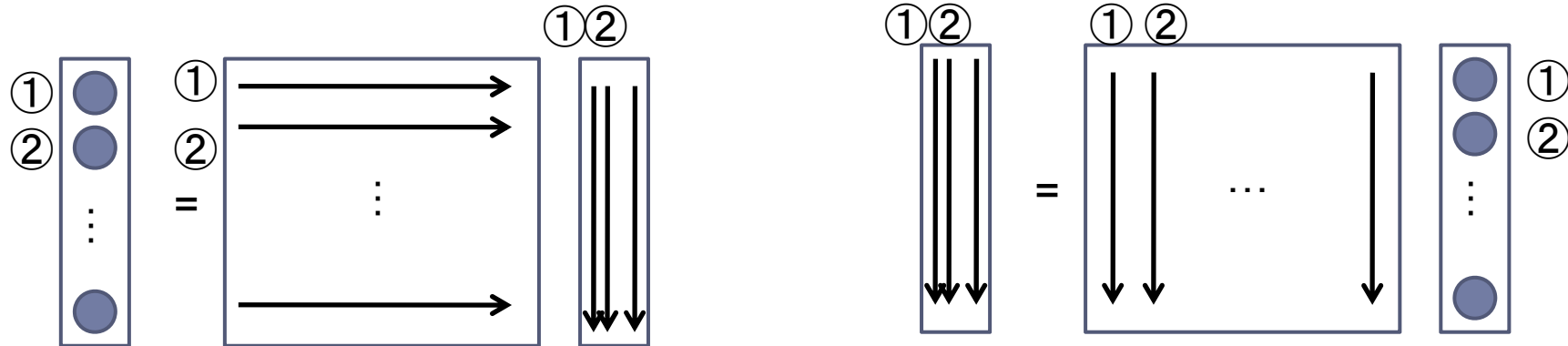
$$z = \alpha x + y$$

- ▶ ここで、 α はスカラ、 z 、 x 、 y はベクトル
- ▶ どのようなデータ分散方式でも並列処理が可能
 - ▶ ただし、スカラ α は全PEで所有する。
 - ▶ ベクトルは $O(n)$ のメモリ領域が必要なのに対し、スカラは $O(1)$ のメモリ領域で大丈夫。
→スカラメモリ領域は無視可能
- ▶ 計算量： $O(N/P)$
- ▶ あまり面白くない



行列とベクトルの積

- ▶ **<行方式>**と**<列方式>**がある。
 - ▶ **<データ分散方式>**と**<方式>**組のみ合わせがあり、少し面白い



```
for (i=0; i<n; i++) {
    y[i]=0.0;
    for (j=0; j<n; j++) {
        y[i] += a[i][j]*x[j];
    }
}
```

<行方式>： 自然な実装
C言語向き

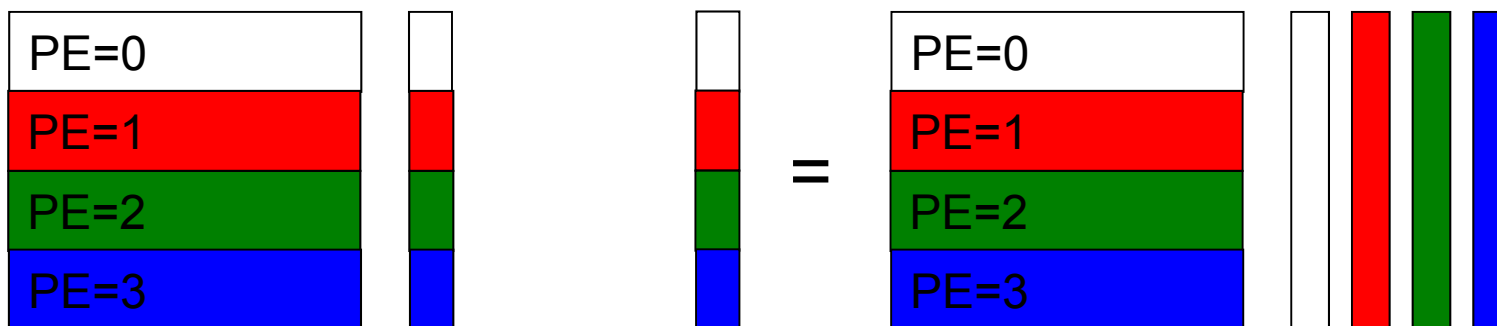
```
for (j=0; j<n; j++) y[j]=0.0;
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        y[i] += a[i][j]*x[j];
    }
}
```

<列方式>： Fortran言語向き

行列とベクトルの積

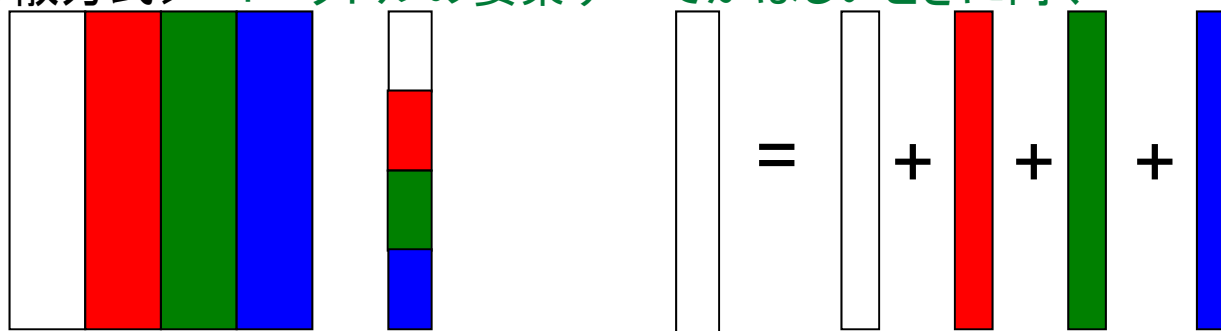
＜行方式の場合＞

＜行方向分散方式＞ : 行方式に向く分散方式



右辺ベクトルを `MPI_Allgather` 関数
を利用し、全PEで所有する
各PE内で行列ベクトル積を行う

＜列方向分散方式＞ : ベクトルの要素すべてがほしいときに向く



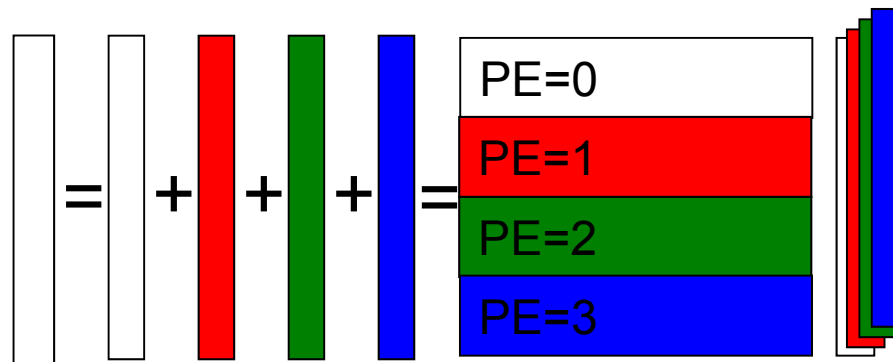
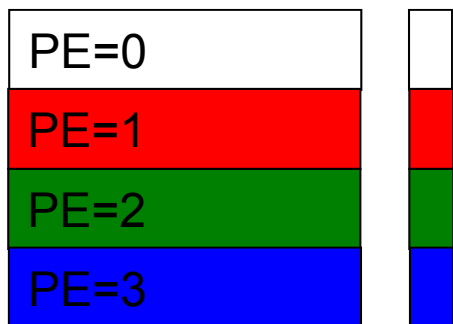
各PE内で行列-ベクトル積
を行う

`MPI_Reduce` 関数で総和を求める
(※ある1PEにベクトルすべてが集まる)

行列とベクトルの積

＜列方式の場合＞

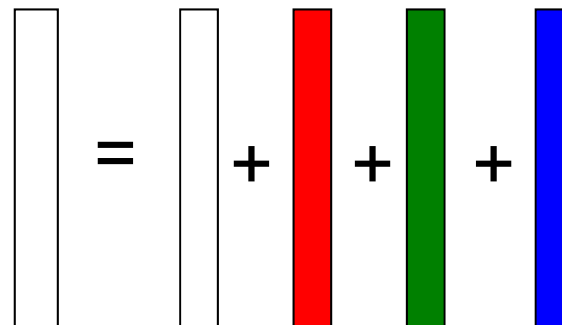
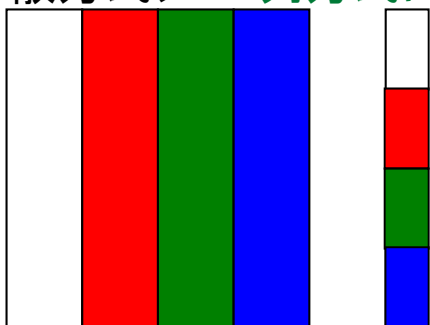
＜行方向分散方式＞ : 無駄が多く使われない



右辺ベクトルを `MPI_Allgather` 関数
を利用して、全PEで所有する

結果を `MPI_Reduce` 関数により
総和を求める

＜列方向分散方式＞ : 列方式に向く分散方式



各PE内で行列-ベクトル積
を行う

`MPI_Reduce` 関数で総和を求める
(※ある1PEにベクトルすべてが集まる)

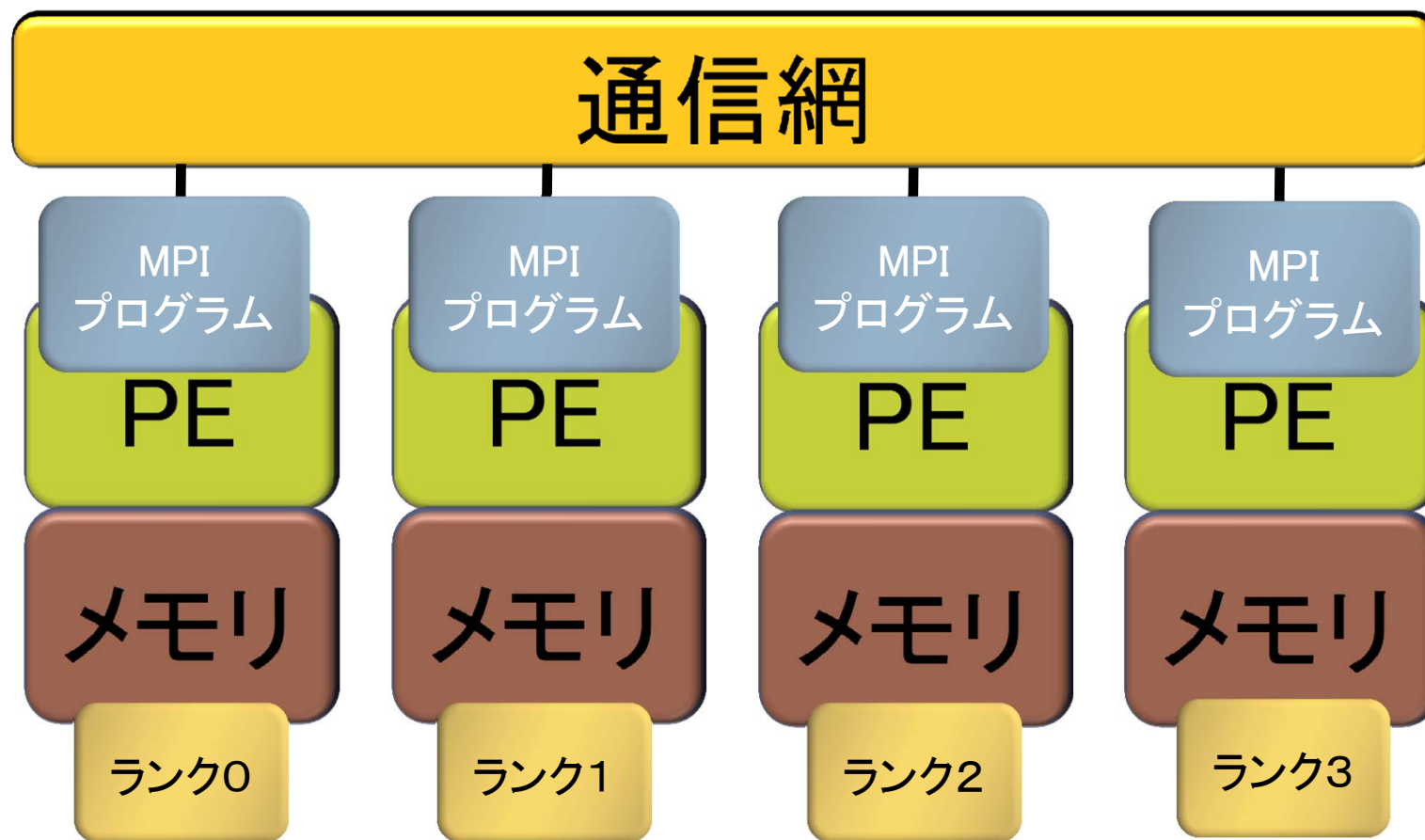
基本的なMPI関数

送信、受信のためのインタフェース

略語とMPI用語

- ▶ MPIは「プロセス」間の通信を行います。プロセスは(普通は)「プロセッサ」(もしくは、コア)に一対一で割り当てられます。
- ▶ 今後、「MPIプロセス」と書くのは長いので、ここではPE (Processer Elementsの略)と書きます。
 - ▶ ただし用語として「PE」は現在はあまり使われていません。
- ▶ **ランク(Rank)**
 - ▶ 各「MPIプロセス」の「識別番号」のこと。
 - ▶ 通常MPIでは、MPI_Comm_rank関数で設定される変数(サンプルプログラムではmyid)に、0～全PE数-1 の数値が入る
 - ▶ 世の中の全MPIプロセス数を知るには、MPI_Comm_size関数を使う。
(サンプルプログラムでは、numprocs に、この数値が入る)

ランクの説明図



C言語インターフェースと Fortranインターフェースの違い

- ▶ C版は、 整数変数*ierr* が戻り値

```
ierr = MPI_Xxxx(...);
```

- ▶ Fortran版は、最後に整数変数*ierr*が引数

```
call MPI_XXXX(..., ierr)
```

- ▶ システム用配列の確保の仕方

- ▶ C言語

```
MPI_Status istatus;
```

- ▶ Fortran言語

```
integer istatus(MPI_STATUS_SIZE)
```

C言語インターフェースと Fortranインターフェースの違い

▶ MPIにおける、データ型の指定

□ C言語

MPI_CHAR (文字型)、**MPI_INT** (整数型)、**MPI_FLOAT** (実数型)、**MPI_DOUBLE**(倍精度実数型)

□ Fortran言語

MPI_CHARACTER (文字型)、**MPI_INTEGER** (整数型)、**MPI_REAL** (実数型)、**MPI_DOUBLE_PRECISION**(倍精度実数型)、**MPI_COMPLEX**(複素数型)

▶ 以降は、C言語インターフェースで説明する

基礎的なMPI関数—MPI_Recv (1 / 2)

```
▶ ierr = MPI_Recv(recvbuf, icount, idatatype,  source,
                 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)

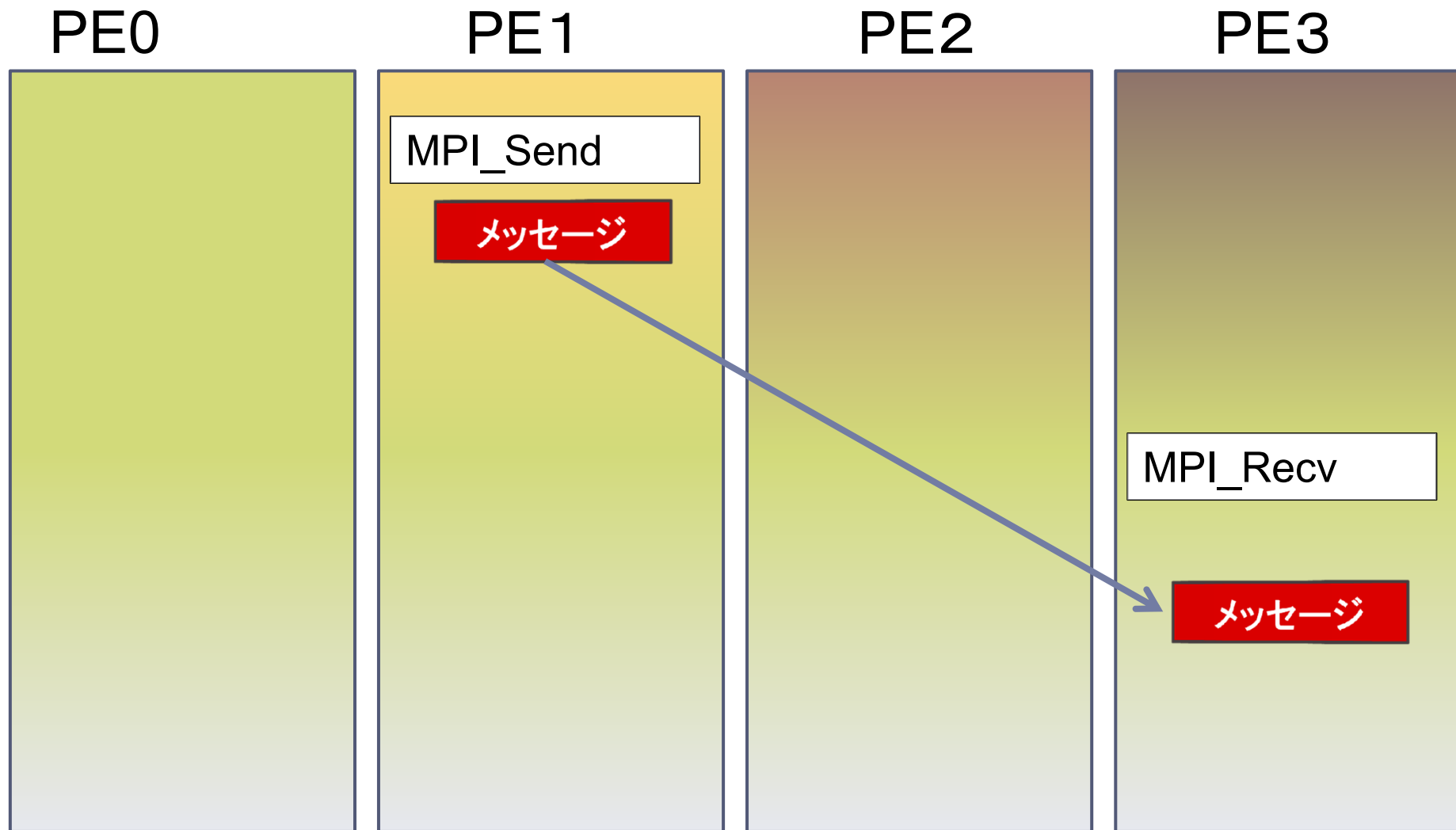
- ▶ **itag** : 整数型。受信したいメッセージに付いているタグの値を指定する。
 - ▶ 任意のタグ値のメッセージを受信したいときは、**MPI_ANY_TAG** を指定する。
- ▶ **icomm** : 整数型。PE集団を認識する番号であるコミュニケータを指定する。
 - ▶ 通常では**MPI_COMM_WORLD** を指定すればよい。
- ▶ **istatus** : MPI_Status型(整数型の配列)。受信状況に関する情報が入る。**かならず専用の型宣言をした配列を確保すること。**
 - ▶ 要素数が**MPI_STATUS_SIZE**の整数配列が宣言される。
 - ▶ 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。

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。

Send-Recvの概念 (1対1通信)

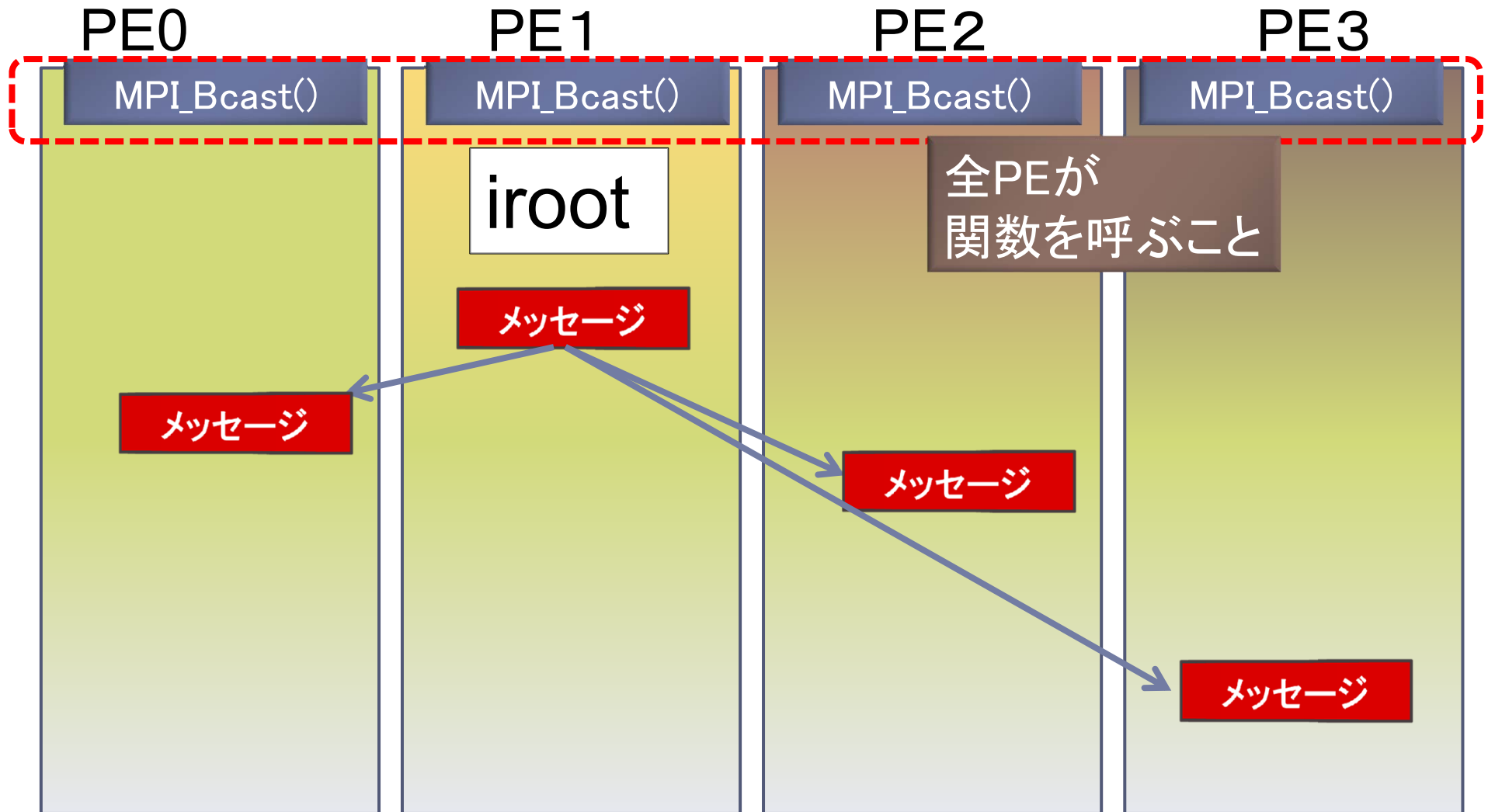


基礎的なMPI関数—MPI_Bcast

```
▶ ierr = MPI_Bcast(sendbuf, icount, idatatype,  
    iroot,  icomm);
```

- ▶ **sendbuf** : 送信および受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。
- ▶ **iroot** : 整数型。送信したいメッセージがあるPEの番号を指定する。全PEで同じ値を指定する必要がある。
- ▶ **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。
- ▶ **ierr (戻り値)** : 整数型。エラーコードが入る。

MPI_Bcastの概念 (集団通信)



リダクション演算

- ▶ <操作>によって<次元>を減少 (リダクション)させる処理
 - ▶ 例: 内積演算
ベクトル(n 次元空間) \rightarrow スカラ(1次元空間)
- ▶ リダクション演算は、通信と計算を必要とする
 - ▶ 集団通信演算 (collective communication operation) と呼ばれる
- ▶ 演算結果の持ち方の違いで、2種のインターフェースが存在する

リダクション演算

▶ 演算結果に対する所有PEの違い

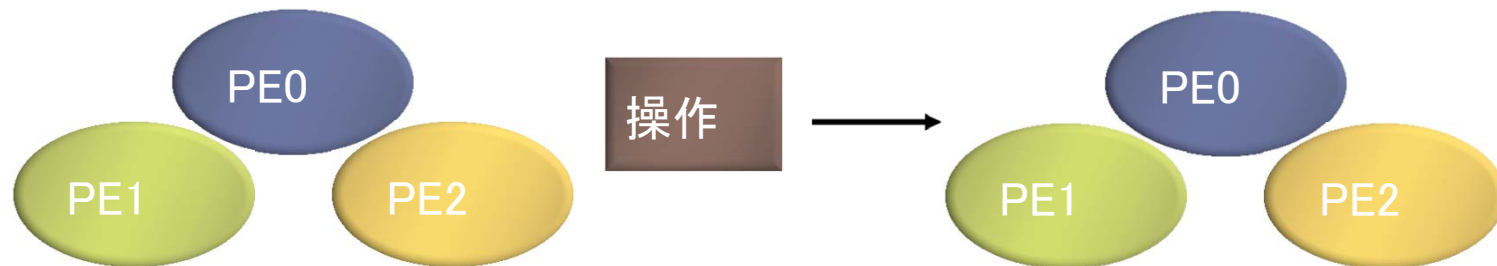
▶ MPI_Reduce関数

- ▶ リダクション演算の結果を、ある一つのPEに所有させる



▶ MPI_Allreduce関数

- ▶ リダクション演算の結果を、全てのPEに所有させる



基礎的なMPI関数—MPI_Reduce

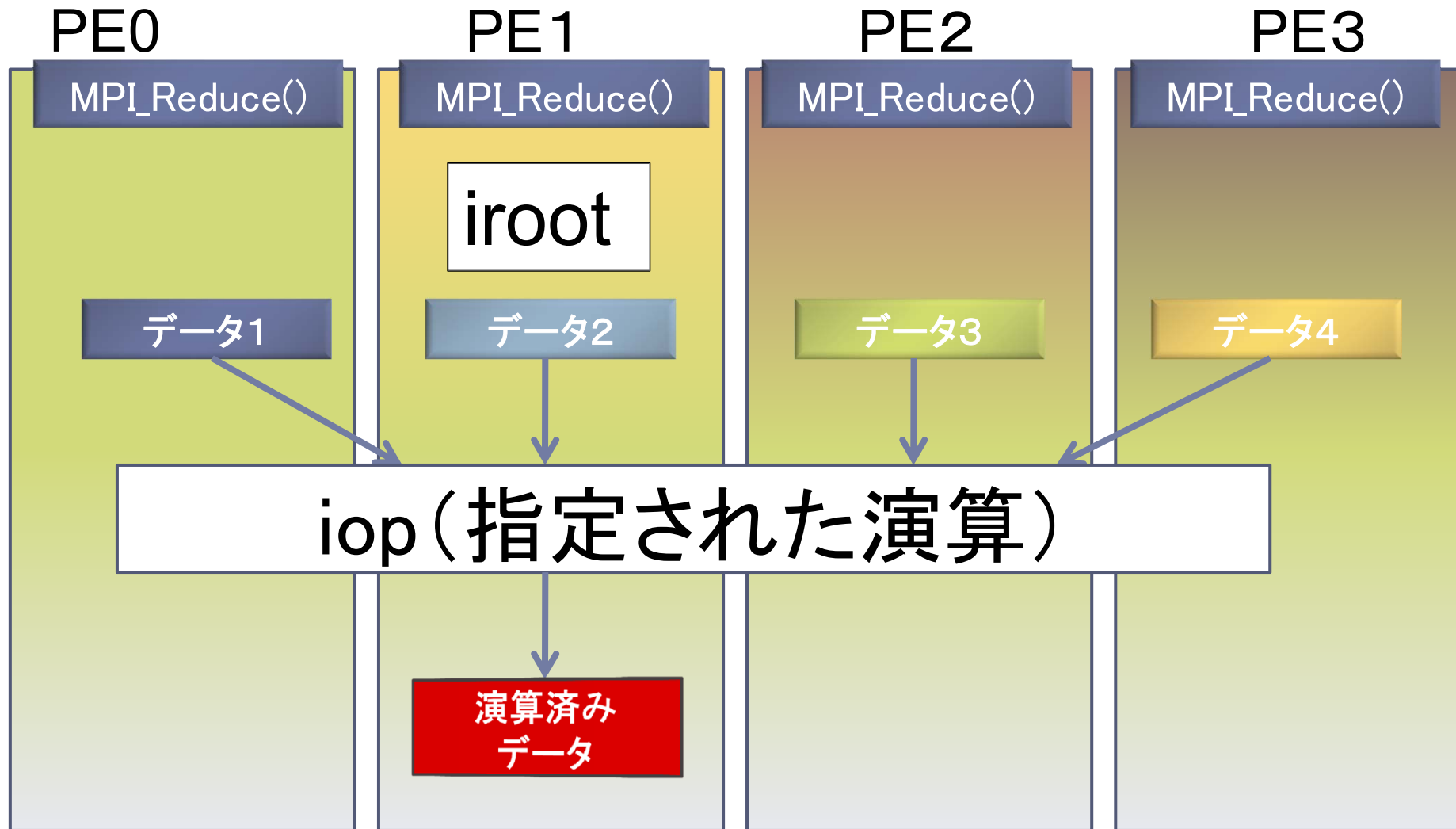
```
▶ ierr = MPI_Reduce(sendbuf, recvbuf, icount,  
                    idatatype, iop, iroot, icomm);
```

- ▶ **sendbuf** : 送信領域の先頭番地を指定する。
- ▶ **recvbuf** : 受信領域の先頭番地を指定する。iroot で指定したPEのみで書き込みがなされる。
送信領域と受信領域は、同一であってはならない。
すなわち、異なる配列を確保しなくてはならない。
- ▶ **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。
 - ▶ <最小／最大値と位置>を返す演算を指定する場合は、**MPI_2INT**(整数型)、**MPI_2FLOAT**(単精度型)、**MPI_2DOUBLE**(倍精度型)、を指定する。

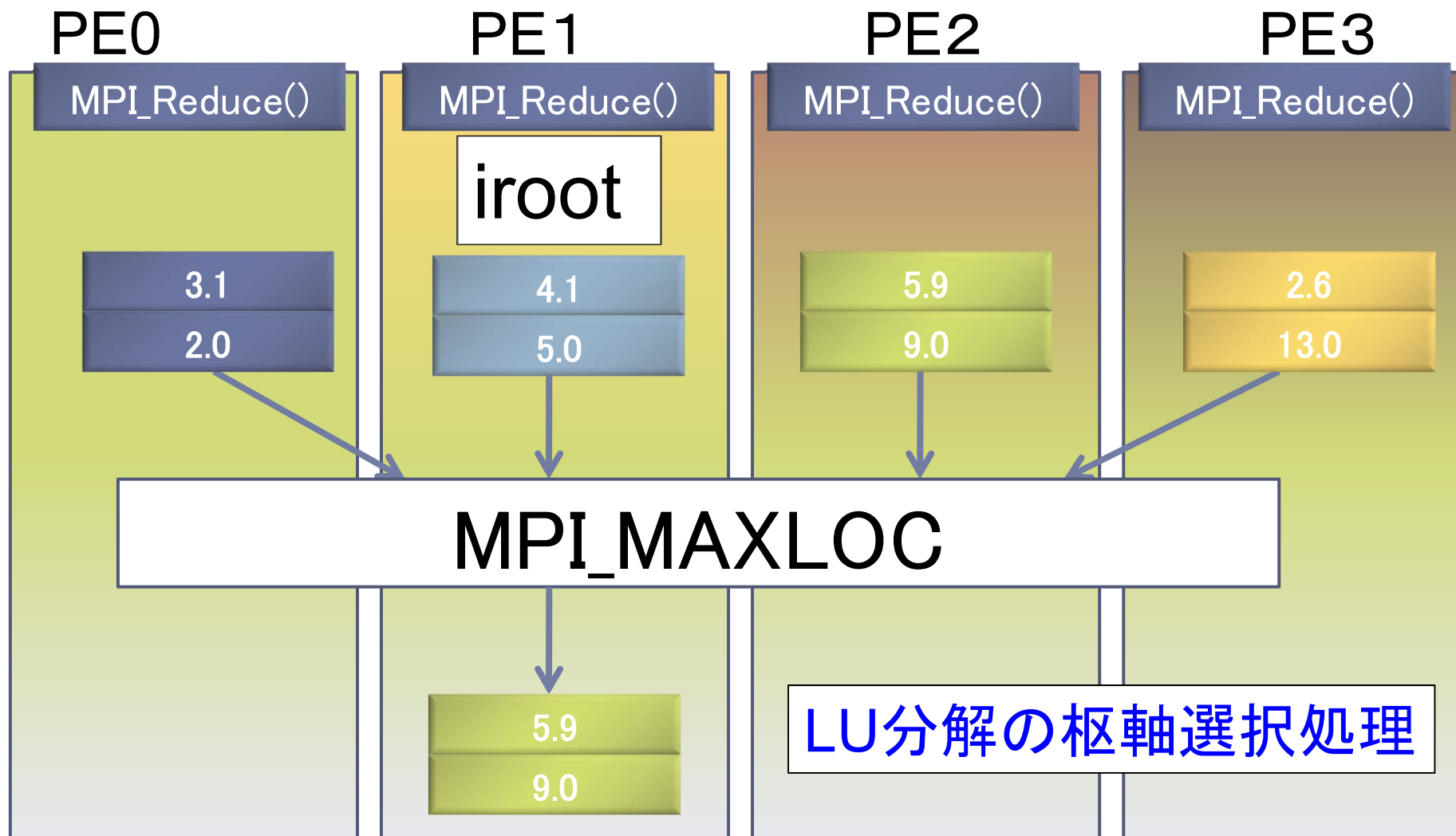
基礎的なMPI関数—MPI_Reduce

- ▶ **iop** : 整数型。演算の種類を指定する。
 - ▶ **MPI_SUM** (総和)、**MPI_PROD** (積)、**MPI_MAX** (最大)、**MPI_MIN** (最小)、**MPI_MAXLOC** (最大と位置)、**MPI_MINLOC** (最小と位置) など。
- ▶ **iroot** : 整数型。結果を受け取るPEのicomm 内のランクを指定する。全てのicomm 内のPEで同じ値を指定する必要がある。
- ▶ **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。
- ▶ **ierr** : 整数型。エラーコードが入る。

MPI_Reduceの概念 (集団通信)



MPI_Reduceによる2リスト処理例 (MPI_2DOUBLEとMPI_MAXLOC)



基礎的なMPI関数—MPI_Allreduce

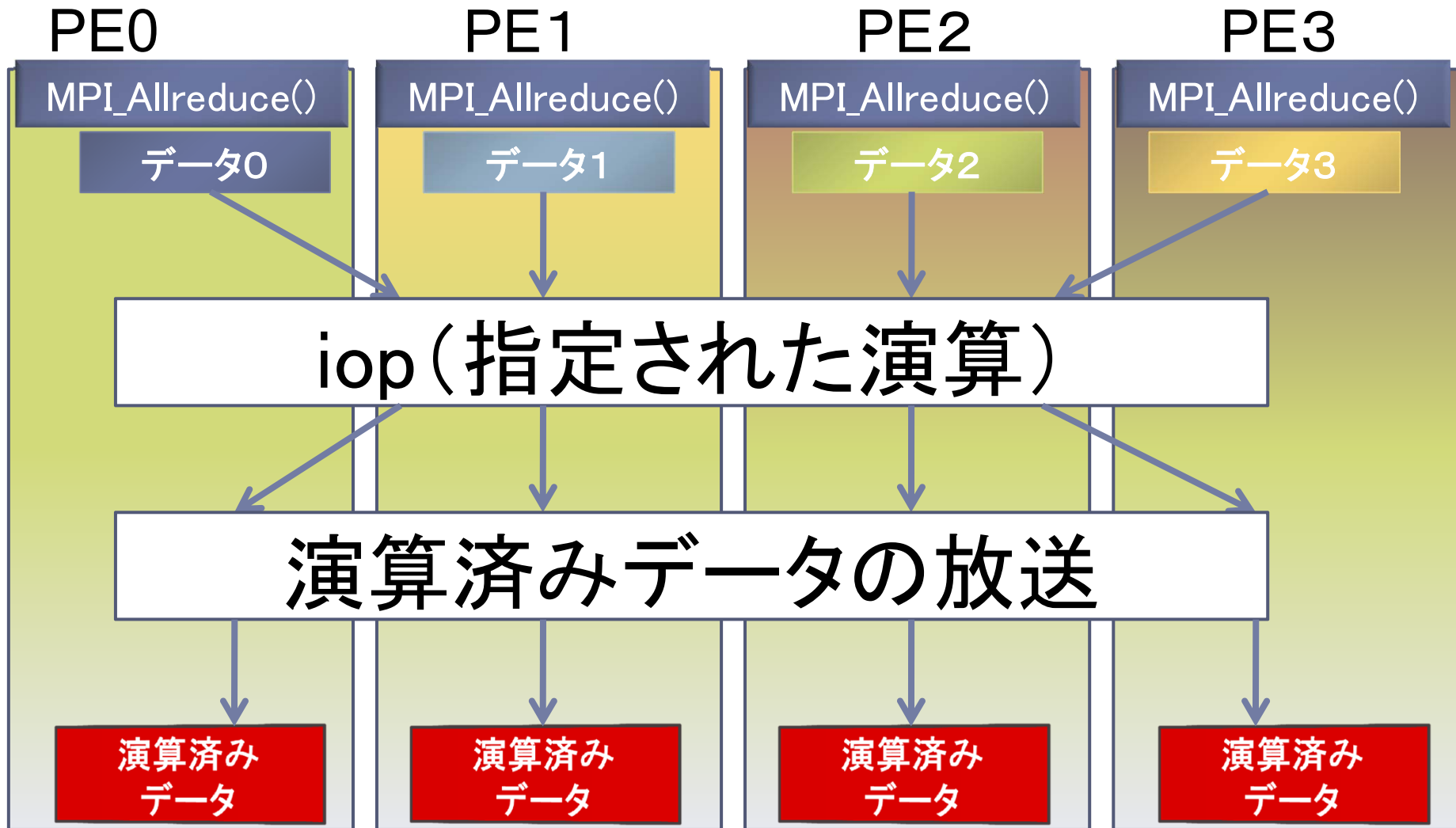
```
▶ ierr = MPI_Allreduce(sendbuf, recvbuf, icount,  
    idatatype, iop, icommm);
```

- ▶ **sendbuf** : 送信領域の先頭番地を指定する。
- ▶ **recvbuf** : 受信領域の先頭番地を指定する。iroot で指定したPEのみで書き込みがなされる。
送信領域と受信領域は、同一であってはならない。
すなわち、異なる配列を確保しなくてはならない。
- ▶ **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。
 - ▶ 最小値や最大値と位置を返す演算を指定する場合は、**MPI_2INT**(整数型)、**MPI_2FLOAT**(単精度型)、**MPI_2DOUBLE**(倍精度型)を指定する。

基礎的なMPI関数—MPI_Allreduce

- ▶ **iop** : 整数型。演算の種類を指定する。
 - ▶ **MPI_SUM** (総和)、**MPI_PROD** (積)、**MPI_MAX** (最大)、**MPI_MIN** (最小)、**MPI_MAXLOC** (最大と位置)、**MPI_MINLOC** (最小と位置) など。
- ▶ **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。
- ▶ **ierr** : 整数型。エラーコードが入る。

MPI_Allreduceの概念 (集団通信)



リダクション演算

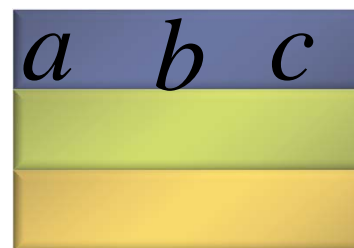
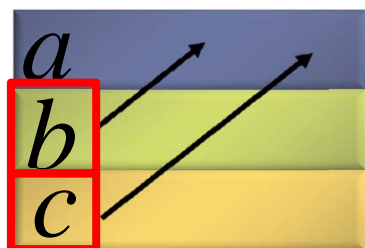
▶ 性能について

- ▶ リダクション演算は、1対1通信に比べ遅い
 - ▶ プログラム中で多用すべきでない！
- ▶ MPI_Allreduce は MPI_Reduce に比べ遅い
 - ▶ MPI_Allreduce は、放送処理が入る。
 - ▶ なるべく、MPI_Reduce を使う。

行列の転置

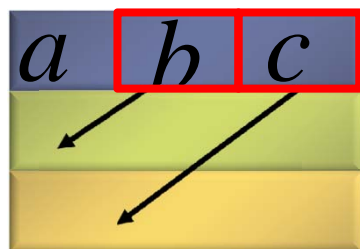
- ▶ 行列 A が (Block, *) 分散されているとする。
- ▶ 行列 A の転置行列 A^T を作るには、MPIでは次の2通りの関数を用いる

- ▶ MPI_Gather関数



集めるメッセージ
サイズが各PEで
均一のとき使う

- ▶ MPI_Scatter関数



集めるサイズが
各PEで均一で
ないときは
MPI_GatherV関数
MPI_ScatterV関数

基礎的なMPI関数—MPI_Gather

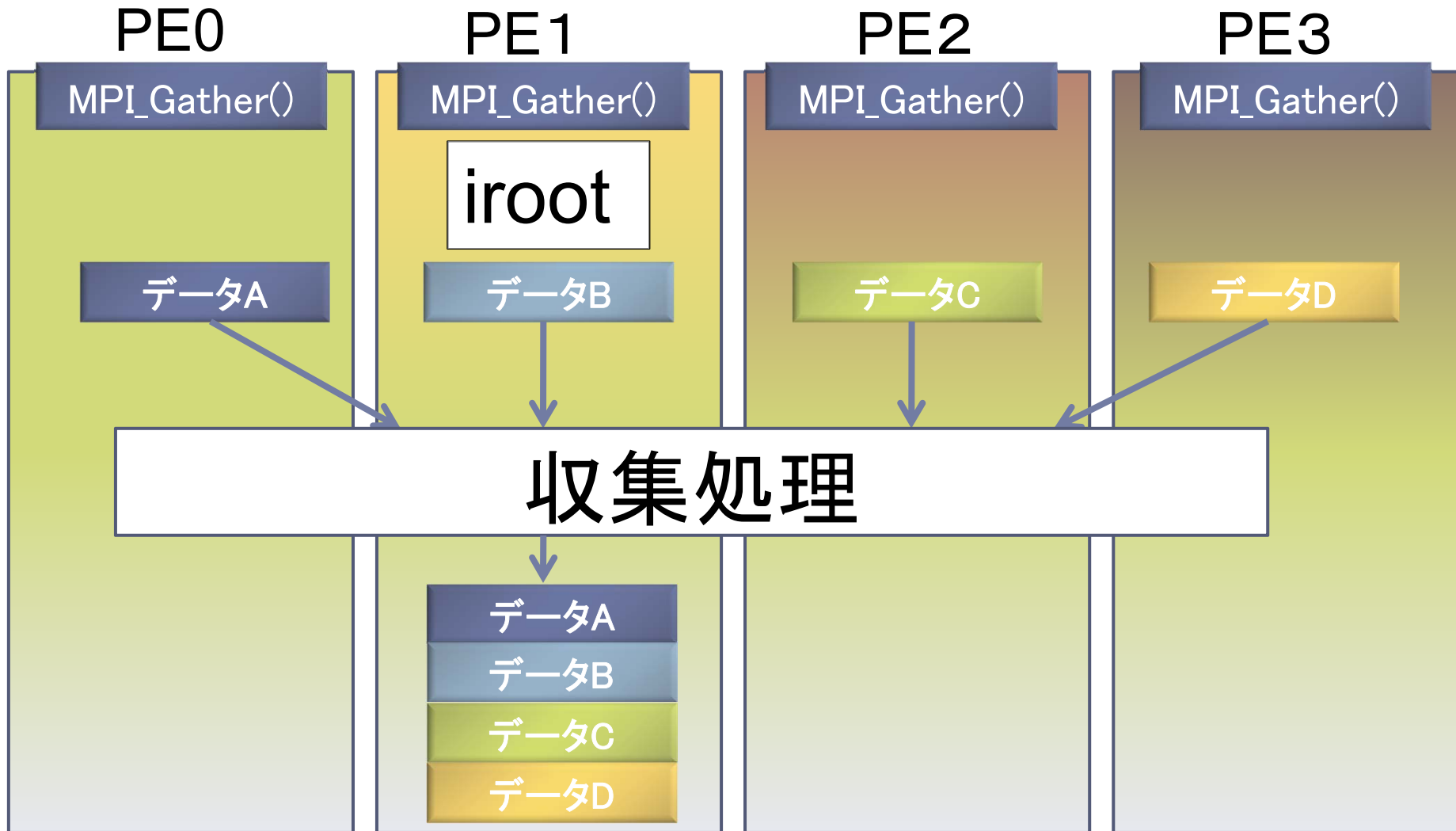
```
▶ ierr = MPI_Gather (sendbuf, isendcount, isendtype,  
                    recvbuf, irecvcount, irecvtype, iroot,  icomm);
```

- ▶ **sendbuf** : 送信領域の先頭番地を指定する。
- ▶ **isendcount**: 整数型。送信領域のデータ要素数を指定する。
- ▶ **isendtype** : 整数型。送信領域のデータの型を指定する。
- ▶ **recvbuf** : 受信領域の先頭番地を指定する。iroot で指定したPEのみで書き込みがなされる。
 - ▶ なお原則として、送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。
- ▶ **irecvcount**: 整数型。受信領域のデータ要素数を指定する。
 - ▶ この要素数は、1PE当たりの送信データ数を指定すること。
 - ▶ MPI_Gather 関数では各PEで異なる数のデータを収集することはできないので、同じ値を指定すること。

基礎的なMPI関数—MPI_Gather

- ▶ **irecvtype** : 整数型。受信領域のデータ型を指定する。
- ▶ **irroot** : 整数型。収集データを受け取るPEの `icomm` 内でのランクを指定する。
 - ▶ 全ての `icomm` 内のPEで同じ値を指定する必要がある。
- ▶ **icomm** : 整数型。PE集団を認識する番号である `icomm` を指定する。
- ▶ **ierr** : 整数型。エラーコードが入る。

MPI_Gatherの概念 (集団通信)



基礎的なMPI関数—MPI_Scatter

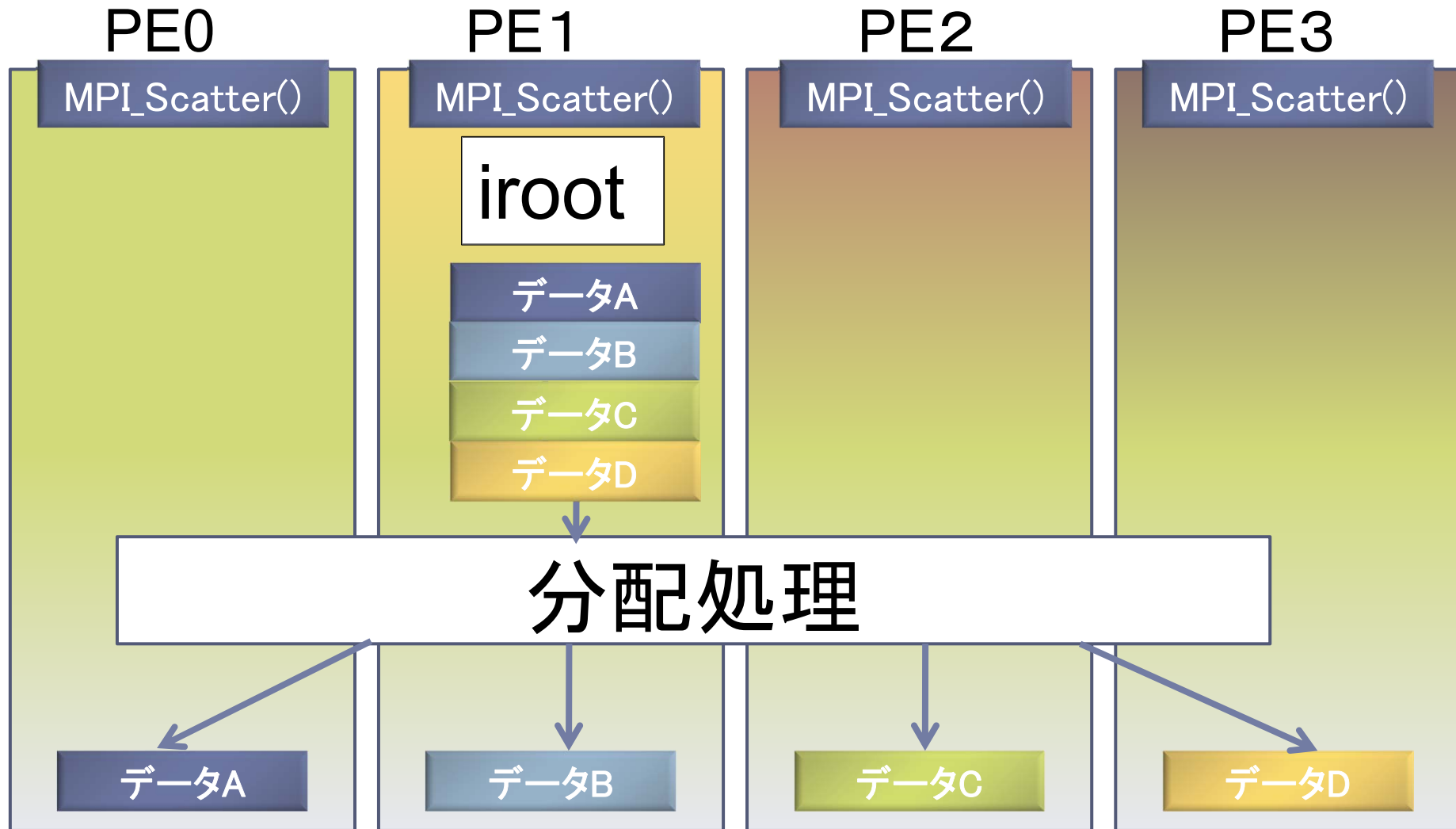
```
▶ ierr = MPI_Scatter ( sendbuf, isendcount, isendtype,  
                    recvbuf, irecvcount, irecvtype, iroot,  icomm);
```

- ▶ **sendbuf** : 送信領域の先頭番地を指定する。
- ▶ **isendcount**: 整数型。送信領域のデータ要素数を指定する。
 - ▶ この要素数は、1PEあたりに送られる送信データ数を指定すること。
 - ▶ MPI_Scatter 関数では各PEで異なる数のデータを分散することはできないので、同じ値を指定すること。
- ▶ **isendtype** : 整数型。送信領域のデータの型を指定する。
iroot で指定したPEのみ有効となる。
- ▶ **recvbuf** : 受信領域の先頭番地を指定する。
 - ▶ なお原則として、送信領域と受信領域は、同一であってはならない。
すなわち、異なる配列を確保しなくてはならない。
- ▶ **irecvcount**: 整数型。受信領域のデータ要素数を指定する。

基礎的なMPI関数—MPI_Scatter

- ▶ **irecvtype** : 整数型。受信領域のデータ型を指定する。
- ▶ **irroot** : 整数型。収集データを受け取るPEの **icomm** 内でのランクを指定する。
 - ▶ 全ての **icomm** 内のPEで同じ値を指定する必要がある。
- ▶ **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。
- ▶ **ierr** : 整数型。エラーコードが入る。

MPI_Scatterの概念 (集団通信)



MPIプログラム実例

MPIの起動

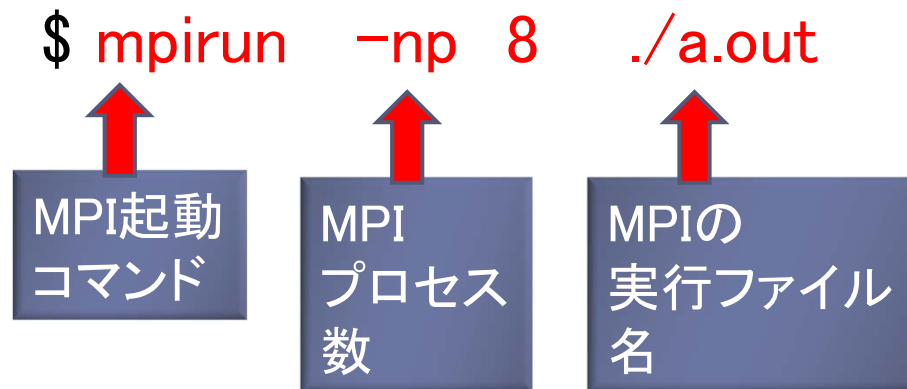
▶ MPIを起動するには

1. MPIをコンパイルできるコンパイラでコンパイル

- ▶ 実行ファイルは a.out とする(任意の名前を付けられます)

2. 以下のコマンドを実行

- ▶ インタラクティブ実行では、以下のコマンドを直接入力
- ▶ バッチジョブ実行では、ジョブスクリプトファイル中に記載

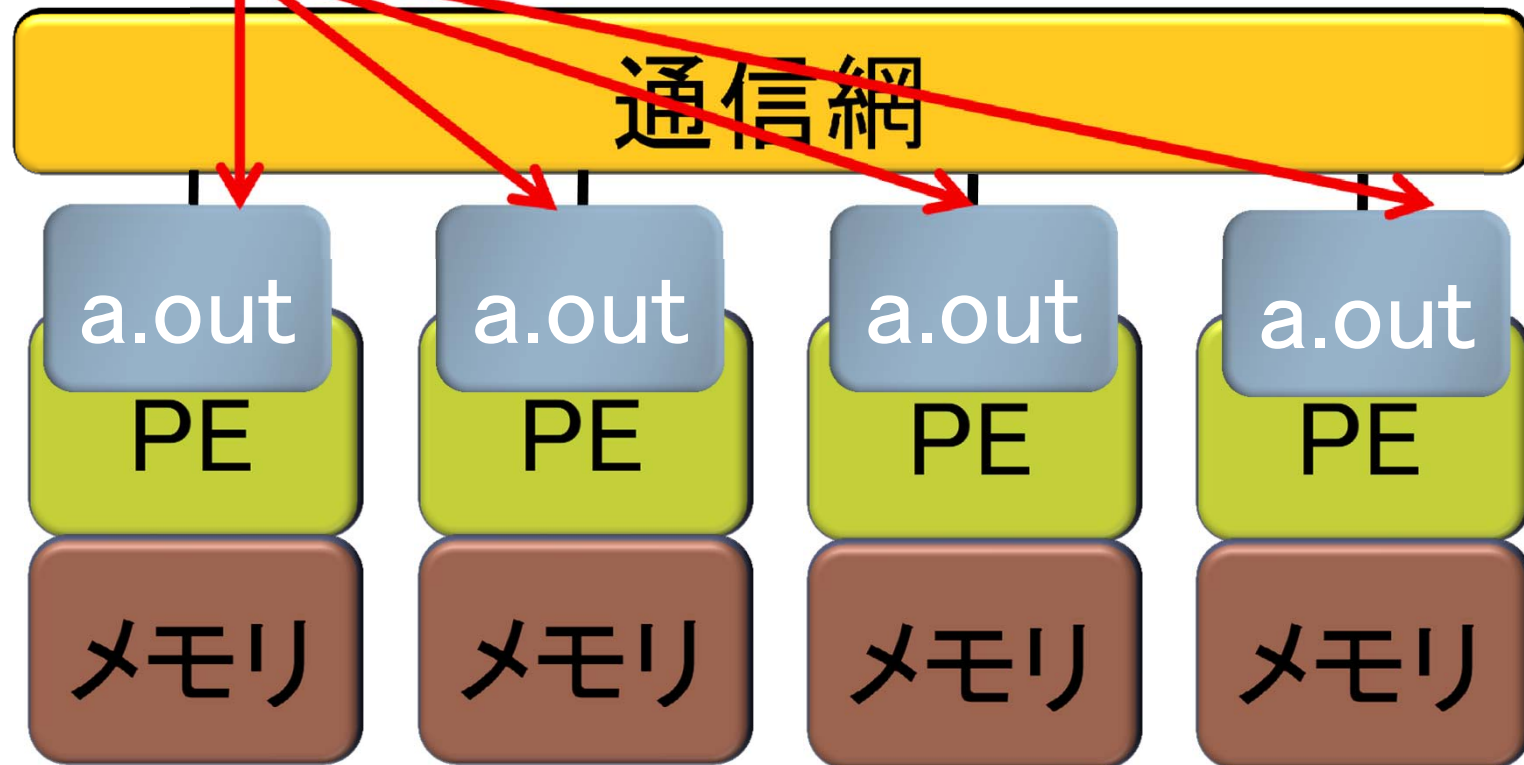


※スパコンのバッチジョブ実行では、MPIプロセス数は専用の指示文で指定する場合があります。その場合は以下になることがあります。

```
$mpirun ./a.out
```

MPIの起動

```
mpirun -np 4 ./a.out
```



並列版Helloプログラムの説明（C言語）

このプログラムは、全PEで起動される

```
#include <stdio.h>
#include <mpi.h>
```

```
void main(int argc, char* argv[]) {
```

```
    int  myid, numprocs;
    int  ierr, rc;
```

```
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

```
    printf("Hello parallel world! Myid:%d ¥n", myid);
```

```
    rc = MPI_Finalize();
```

```
    exit(0);
```

```
}
```

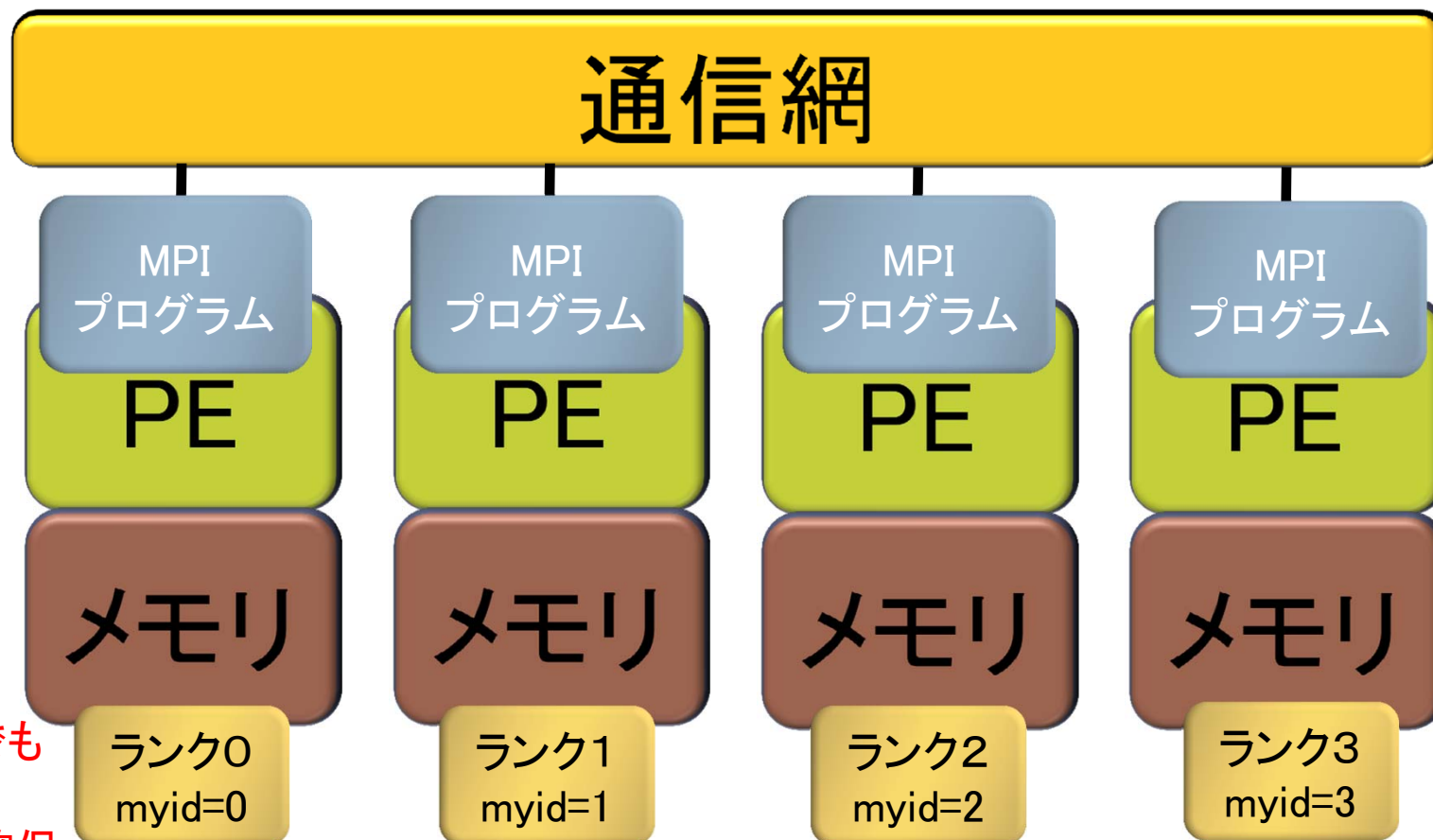
MPIの初期化

自分のID番号を取得
:各PEで値は異なる

全体のプロセッサ台数
を取得
:各PEで値は同じ

MPIの終了

変数myidの説明図



同じ変数名でも
別メモリ上
に別変数で確保

並列版Helloプログラムの説明 (Fortran言語)

このプログラムは、全PEで起動される

```
program main  
include 'mpif.h'  
common /mpienv/myid,numprocs
```

```
integer myid, numprocs  
integer ierr
```

```
call MPI_INIT(ierr)  
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)  
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
```

```
print *, "Hello parallel world! Myid:", myid
```

```
call MPI_FINALIZE(ierr)
```

```
stop  
end
```

MPIの初期化

自分のID番号を取得
:各PEで値は異なる

全体のプロセッサ台数
を取得
:各PEで値は同じ

MPIの終了

プログラム出力例

▶ 4プロセス実行の出力例

Hello parallel world! Myid:0

Hello parallel world! Myid:3

Hello parallel world! Myid:1

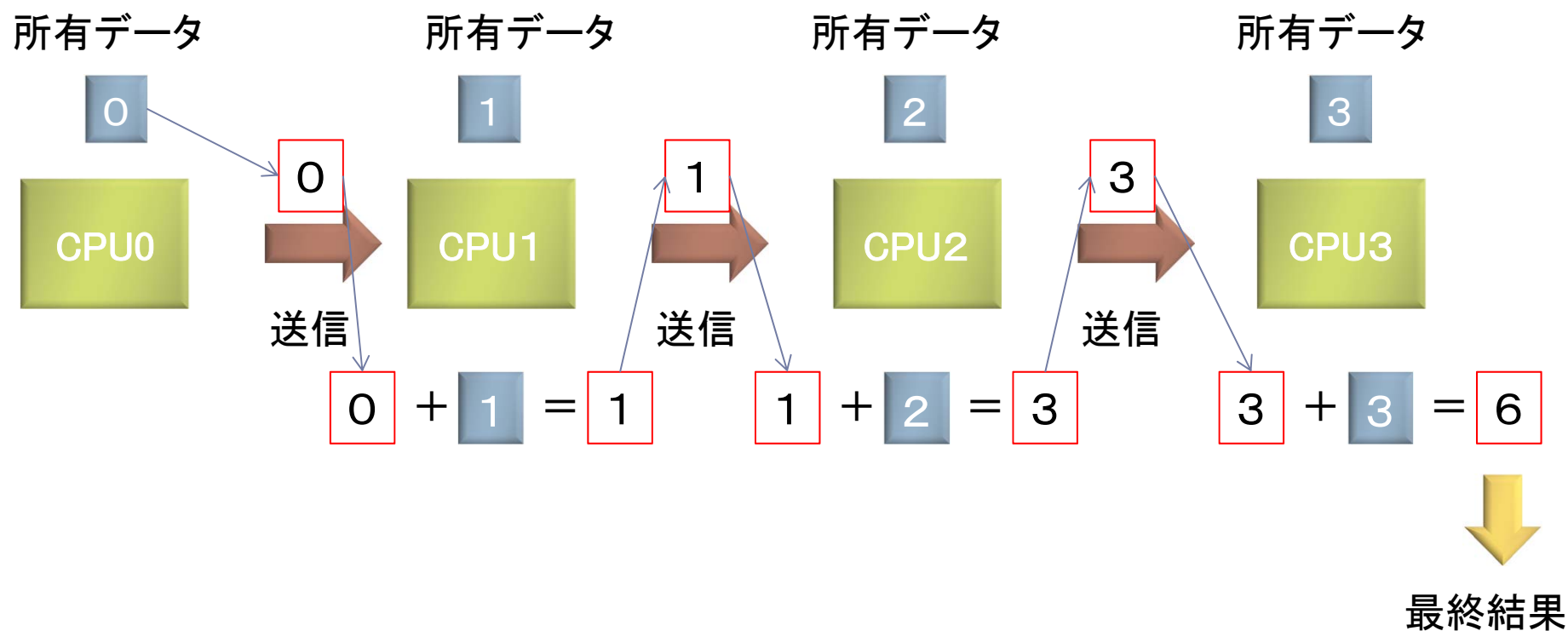
Hello parallel world! Myid:2

- 4プロセスなので、表示が4個である
(1000プロセスなら1000個出力ができる)
- myid番号が表示される。全体で重複した番号は無い。
- 必ずしも、myidが0から3まで、連続して出ない
 - 各行は同期して実行されていない
 - 実行ごとに結果は異なる

総和演算プログラム（逐次転送方式）

- ▶ 各プロセスが所有するデータを、全プロセスで加算し、あるプロセス1つが結果を所有する演算を考える。
- ▶ **素朴な方法（逐次転送方式）**
 1. (0番でなければ)左隣のプロセスからデータを受信する;
 2. 左隣のプロセスからデータが来ていたら;
 1. 受信する;
 2. **<自分のデータ>**と**<受信データ>**を加算する;
 3. (63番でなければ)右隣のプロセスに**<2の加算した結果を>**送信する;
 4. 処理を終了する;
- ▶ **実装上の注意**
 - ▶ 左隣りとは、(myid-1)のIDをもつプロセス
 - ▶ 右隣りとは、(myid+1)のIDをもつプロセス
 - ▶ myid=0のプロセスは、左隣りはないので、受信しない
 - ▶ myid=p-1のプロセスは、右隣りはないので、送信しない

バケツリレー方式による加算



1 対 1 通信利用例 (逐次転送方式、C言語)

```
void main(int argc, char* argv[]) {
  MPI_Status istatus;
  ....
  dsendbuf = myid;
  drecvbuf = 0.0;
  if (myid != 0) {
    ierr = MPI_Recv(&drecvbuf, 1, MPI_DOUBLE, myid-1, 0,
                   MPI_COMM_WORLD, &istatus);
  }
  dsendbuf = dsendbuf + drecvbuf;
  if (myid != nprocs-1) {
    ierr = MPI_Send(&dsendbuf, 1, MPI_DOUBLE, myid+1, 0,
                   MPI_COMM_WORLD);
  }
  if (myid == nprocs-1) printf ("Total = %4.2lf ¥n", dsendbuf);
  ....
}
```

受信システム配列の確保

自分より一つ少ない
ID番号(myid-1)から、
double型データ一つを
受信しdrecvbuf変数に
代入

自分より一つ多い
ID番号(myid+1)に、
dsendbuf変数に入っ
ているdouble型データ
一つを送信

1 対 1 通信利用例 (逐次転送方式、Fortran言語)

```
program main
integer istatus(MPI_STATUS_SIZE)
....
dsendbuf = myid
drecvbuf = 0.0
if (myid .ne. 0) then
  call MPI_RECV(drecvbuf, 1, MPI_DOUBLE_PRECISION,
&             myid-1, 0, MPI_COMM_WORLD, istatus, ierr)
endif
dsendbuf = dsendbuf + drecvbuf
if (myid .ne. numprocs-1) then
  call MPI_SEND(dsendbuf, 1, MPI_DOUBLE_PRECISION,
&             myid+1, 0, MPI_COMM_WORLD, ierr)
endif
if (myid .eq. numprocs-1) then
  print *, "Total = ", dsendbuf
endif
....
stop
end
```

受信システム配列の確保

自分より一つ少ない
ID番号(myid-1)から、
double型データ一つを
受信しdrecvbuf変数に
代入

自分より一つ多い
ID番号(myid+1)に、
dsendbuf変数に
入っているdouble型
データ一つを送信

総和演算プログラム（二分木通信方式）

▶ 二分木通信方式

1. $k = 1;$
2. for ($i=0; i < \log_2(\text{nprocs}); i++$)
3. if (($\text{myid} \& k$) == k)
 - ▶ ($\text{myid} - k$)番 プロセス からデータを受信;
 - ▶ 自分のデータと、受信データを加算する;
 - ▶ $k = k * 2;$
4. else
 - ▶ ($\text{myid} + k$)番 プロセス に、データを転送する;
 - ▶ 処理を終了する;

総和演算プログラム（二分木通信方式）

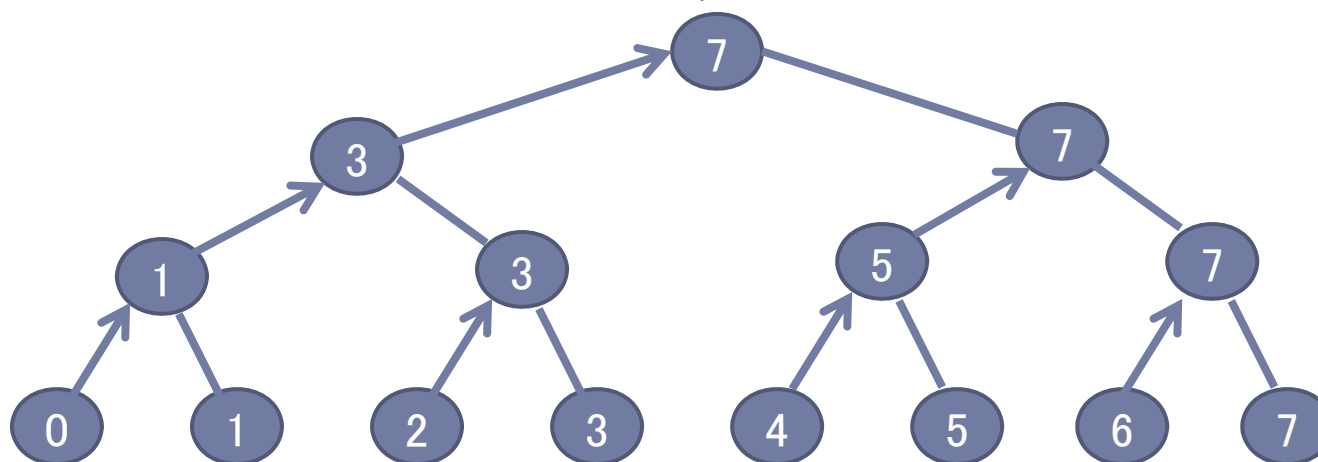
3段目 = $\log_2(8)$ 段目



2段目



1段目



総和演算プログラム（二分木通信方式）

▶ 実装上の工夫

- ▶ **要点:** プロセス番号の2進数表記の情報を利用する
- ▶ 第*i*段において、受信するプロセスの条件は、以下で書ける:
 $myid \& k$ が k と一致
 - ▶ ここで、 $k = 2^{(i-1)}$ 。
 - ▶ つまり、プロセス番号の2進数表記で右から*i*番目のビットが立っているプロセスが、送信することにする
- ▶ また、送信元のプロセス番号は、以下で書ける:
 $myid + k$
 - ▶ つまり、通信が成立するPE番号の間隔は $2^{(i-1)}$ ←二分木なので
- ▶ 送信プロセスについては、上記の逆が成り立つ。

総和演算プログラム（二分木通信方式）

- ▶ 逐次転送方式の通信回数
 - ▶ 明らかに、 $nprocs - 1$ 回
- ▶ 二分木通信方式の通信回数
 - ▶ 見積もりの前提
 - ▶ 各段で行われる通信は、完全に並列で行われる（通信の衝突は発生しない）
 - ▶ 段数の分の通信回数となる
 - ▶ つまり、 $\log_2(nprocs)$ 回
- ▶ 両者の通信回数の比較
 - ▶ プロセッサ台数が増すと、通信回数の差（＝実行時間）がとても大きくなる
 - ▶ 1024構成では、1023回 対 10回！
 - ▶ でも、必ずしも二分木通信方式がよいとは限らない（通信衝突の多発）

参考文献

1. MPI並列プログラミング、P.パチェコ 著 / 秋葉博 訳
2. 並列プログラミング虎の巻MPI版、青山幸也 著、
理化学研究所情報基盤センタ
(<http://acc.riken.jp/HPC/training/text.html>)
3. Message Passing Interface Forum
(<http://www.mpi-forum.org/>)
4. MPI-Jマーキングリスト
(<http://phase.hpcc.jp/phase/mpi-j/ml/>)
5. 並列コンピュータ工学、富田眞治著、昭晃堂(1996)

レポート課題（その1）

▶ 問題レベルを以下に設定

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。

▶ 教科書のサンプルプログラムは以下が利用可能

- ▶ [Sample-fx.tar](#)

レポート課題（その2）

1. [L05] MPIとは何か説明せよ。
2. [L10] 逐次転送方式、二分木通信方式の実行時間を計測し、どの方式が何台のプロセッサ台数で有効となるかを明らかにせよ。また、その理由について、考察せよ。
3. [L15] 二分木通信方式について、プロセッサ台数が2のべき乗でないときにも動作するように、プログラムを改良せよ。