第176回 お試しアカウント付き 並列プログラミング講習会 「MPI基礎：並列プログラミング入門」

東京大学 情報基盤センター
三木 洋平

講習会概略

- 開催日：2022年4月26日（火） 10:00–17:00
- 形態：ZoomおよびSlackを用いたオンライン講習会
- 使用システム：Wisteria/BDEC-01 (Odyssey)
- 講習会プログラム：
  - 10:00–11:20 テストプログラムの実行など（演習）
  - 11:30–12:30 並列プログラミングの基本（座学）
    (12:30–13:40昼休み)
  - 13:40–14:40 MPIプログラム実習1（演習）
  - 14:50–15:50 MPIプログラム実習2（演習）
  - 16:00–17:00 MPIプログラム実習3（演習）

並列プログラミングの基礎（座学）

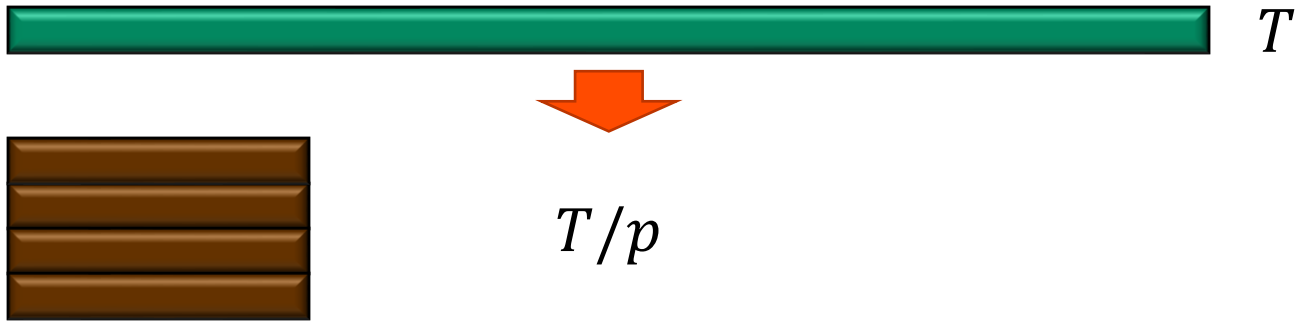
- 並列プログラミングの基礎
- 性能評価指標
- MPI (Message Passing Interface)
- 基礎的なMPI関数
- データ分散方式

並列プログラミングの基礎（座学）

- 並列プログラミングの基礎
- 性能評価指標
- MPI (Message Passing Interface)
- 基礎的なMPI関数
- データ分散方式

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）の実行時間を，p台の計算機を用いてT/pにすること



- 自明に思えるが，実際にできるかどうかは，対象処理の内容（アルゴリズムなど）で難易度が異なる
  - 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ーヘッド（通信立ち上がり時間，データ転送時間）

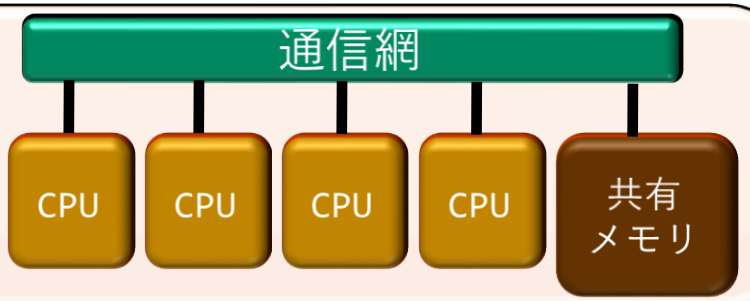
並列計算機の分類

- Michael J. Flynn教授（スタンフォード大）の分類（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（1/2）

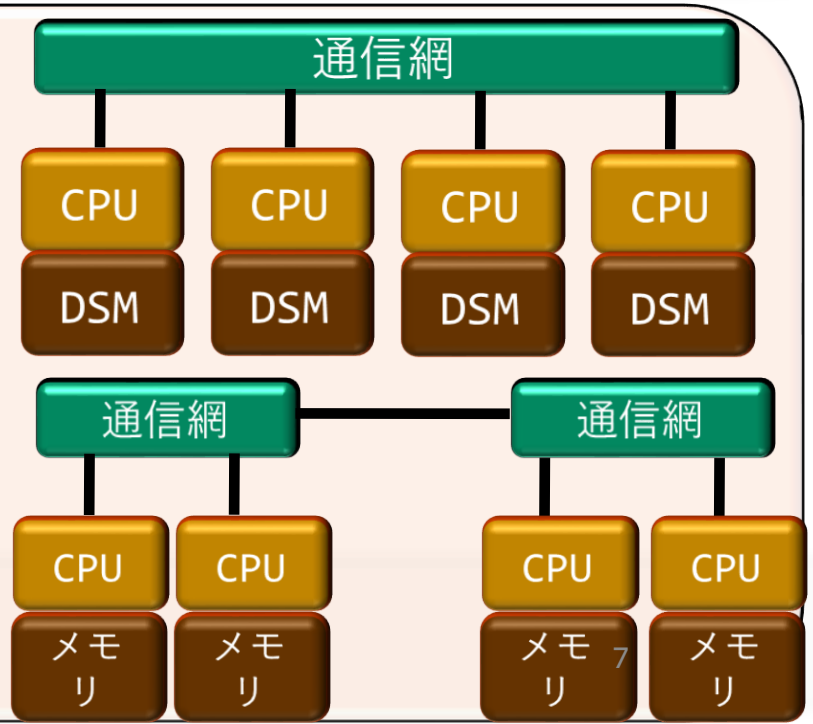
- A) メモリアドレスを共有している：互いのメモリがアクセス可能

1. 共有メモリ型
   (SMP: Symmetric Multiprocessor,
   UMA: Uniform Memory Access)



2. 分散共有メモリ型
   (DSM: Distributed Shared Memory)

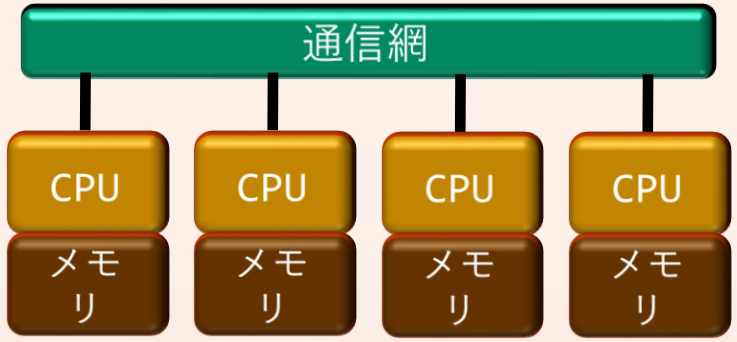
   共有・非対称メモリ型
   (ccNUMA: Cache Coherent
   Non-Uniform Memory Access)



並列計算機のメモリ型による分類(2/2)

- B) メモリアドレスは独立：互いのメモリはアクセス不可

3．分散メモリ型
（メッセージパッシング）



プログラミング手法から見た分類

1．マルチスレッド
- Pthreads，…

2．データ並列
- OpenMP
- （最近の）Fortran
- PGAS (Partitioned Global Address Space)言語： XcalableMP, UPC, Chapel, X10, Co-array Fortran, …

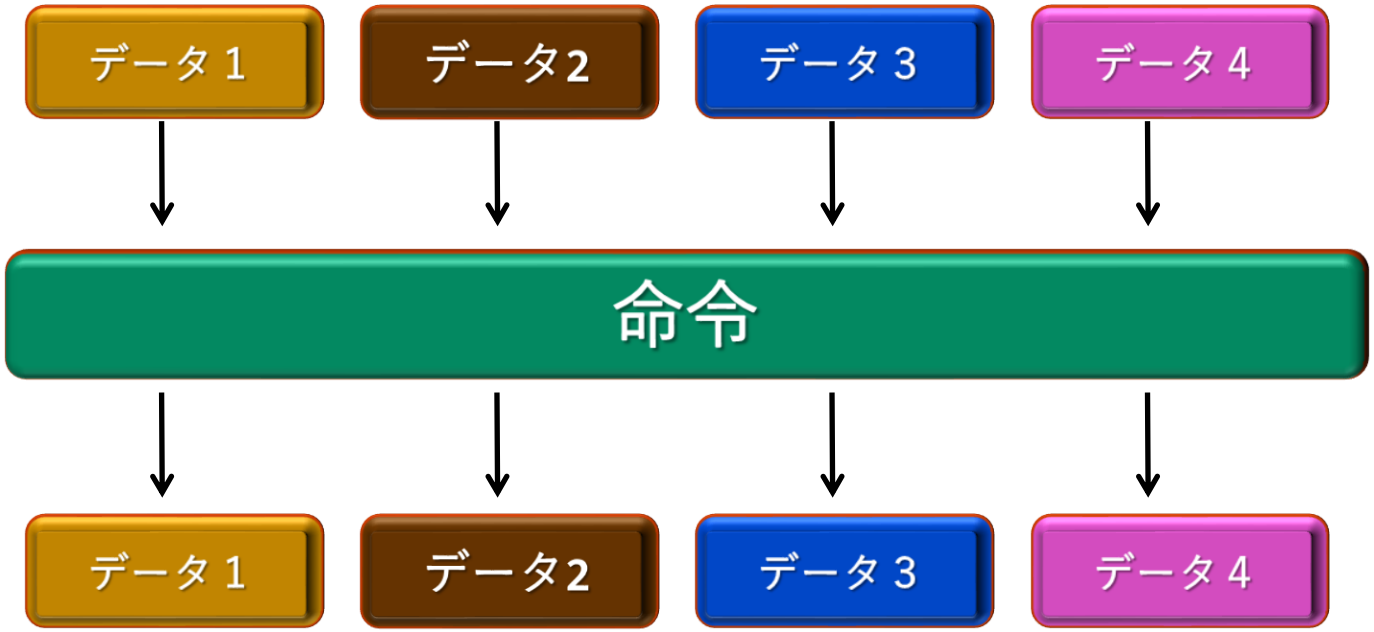
3．タスク並列
- Cilk (Cilk plus), Thread Building Block (TBB), StackThreads, MassiveThreads, …

4．メッセージ並列
- MPI (Message Passing Interface)

複数ノードにまたがる並列化に使える
他はメモリを共有していること
（共有メモリ）が前提

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にはSIMDが基本
  - いきなり複雑な挙動を考えるのは難しいので，なるべくシンプルに



並列プログラミングのモデル

- 多くの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
- MPI（バージョン1）のモデル

2. Master/Worker

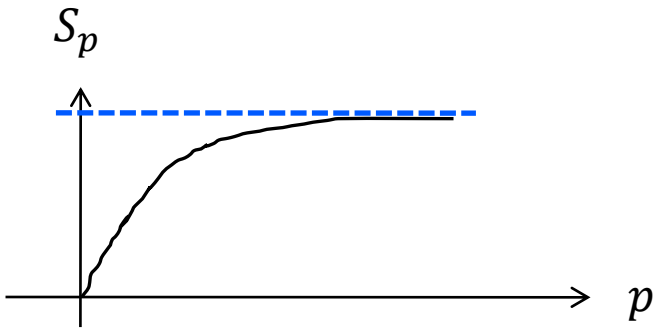
- 1つのプロセス（Master）が，複数のプロセス（Worker）を管理（生成，消去）

並列プログラミングの基礎（座学）

- 並列プログラミングの基礎
- 性能評価指標
- MPI (Message Passing Interface)
- 基礎的なMPI関数
- データ分散方式

性能評価指標 --台数効果--

- 台数効果
  - 式： $S_p = \frac{T_s}{T_p}$，ただしT_sは逐次の，T_pは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，スーパーリニア・スピードアップ
    - 主な原因は，並列化によってデータアクセスが局所化され，キャッシュヒット率が向上したことによる高速化
- 並列化効率
  - 式： $E_p = \frac{S_p}{p} \times 100\ [\%]$
- 飽和性能
  - 速度向上の限界



Weak ScalingとStrong Scaling

- 並列処理においてシステム規模を大きくする方法
- Weak Scaling: プロセッサあたりの問題サイズを変えず並列度をあげる
  - 全体の問題サイズが（並列数に比例して）大きくなる
  - 通信のオーバーヘッドはあまり変わらないか，やや増加
  - 今まで解けなかった規模の問題が解けるようになる
- Strong Scaling: 全体の問題サイズを変えずに並列度をあげる
  - プロセッサあたりの問題サイズは（並列数に反比例して）小さくなる
  - 通信のオーバーヘッドは相対的に大きくなる
  - 同じ問題規模でも，より短時間に結果が得られる（Weakよりも難しい）

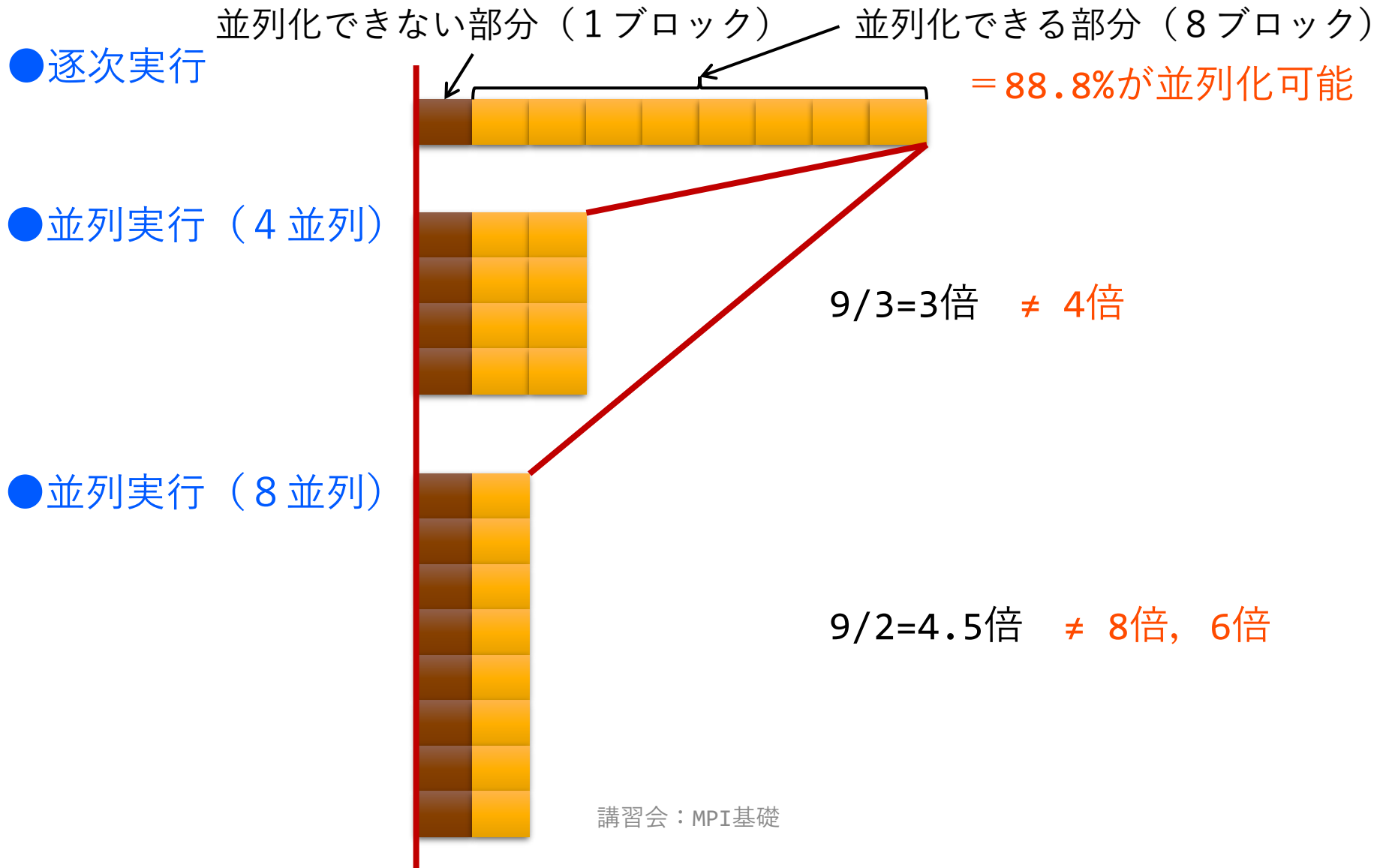
アムダールの法則

- 逐次実行時間をT_sとし，このうち並列化できる割合をαとする
- 台数効果は以下のように計算できる：

$$S_p = \frac{T_s}{\frac{\alpha T_s}{p} + (1-\alpha)T_s} = \frac{1}{\frac{\alpha}{p} + 1 - \alpha} = \frac{1}{1 + \alpha\left(\frac{1}{p} - 1\right)}.$$

- 無限大の数のプロセッサを使っても，台数効果は$1/(1-\alpha)$が上限（アムダールの法則）
  - 全体の90%が並列化できたとしても，台数効果は最大で$\frac{1}{1-0.9} = 10$倍
  - →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重要

アムダールの法則の直感例



Byte/Flop,B/F値

1．プログラム中で要求する演算あたりのメモリアクセスの割合

```
double A[N][N];
...
A[i][j] = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j] + A[i][j+1] + A[i+1][j]);
```

- メモリアクセス：8byteを5回ロード，8byteを1回ストア = 48byte
- 演算： 加算4回，乗算1回 = 5Flops ➔ B/F = 9.6

2．メモリシステムがデータを演算コアに供給する能力，供給できた割合

- 通常は0.1以下，良くても0.5未満
  - B/F=0.5のシステムで上記の計算をすると，メモリ性能の不足により96回分の計算ができたはずの時間で5回しか動かない（i.e.，演算性能の5%しか使えない）
- ➔B/F値の不足をキャッシュによって補う

- どちらのコンテキストで話しているか注意しないと混乱しやすい

並列プログラミングの基礎（座学）

- 並列プログラミングの基礎
- 性能評価指標
- MPI (Message Passing Interface)
- 基礎的なMPI関数
- データ分散方式

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
  - 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の並列実行に向く
- 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（Massively Parallel Processing (MPP)システム）を用いる実行に向く
  - 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って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：これまで

- MPIフォーラム（https://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）
  - 2008年5月 1.3版，2008年6月 2.1版
  - 2009年9月 2.2版
    - 日本語版 https://www.pccluster.org/ja/mpi.html
- MPI-2では，以下を強化：
  - 並列I/O
  - C++，Fortran 90用インタフェース
  - 動的プロセス生成・消滅
    - 主に，並列探索処理などの用途

MPIの経緯：MPI-3.1

- 2012年9月 MPI-3.0
- 2015年6月 MPI-3.1
- 以下のページで現状・ドキュメントを公開中
  - https://www.mpi-forum.org/docs/
- 注目すべき機能
  - ノンブロッキング集団通信機能（MPI_Iallreduceなど）
  - 高性能な片方向通信（RMA: Remote Memory Access）
  - Fortran 2008 対応など

MPIの経緯：MPI-4.0策定（2021/6/9）

- https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
- 新しい機能（pp.1046--1049，34項目のアップデート）
- 主な新機能
  - カウンタ値，バッファサイズの拡張（int32の制限緩和） MPI_{}_c()
  - MPI_Isendrecv()
  - 永続的（Persistent）コレクティブ MPI_{Allgather, …}_init()
  - ハイブリッドプログラミングへの対応（partitioned comm.） MPI_Psend_init()
  - 性能アサーションとヒント
  - セッションモデル
  - RMA/One sided通信
- 見送り
  - MPIアプリケーションの耐故障性（FT: Fault Tolerance）

MPI-4.0 完全準拠の実装がいつ使えるようになるかは不明

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
- MVAPICH（エムヴァピッチ）
  - 米国オハイオ州立大学で開発，MPICHをベース
  - InfiniBand向けの優れた実装
- OpenMPI
  - オープンソース
- ベンダMPI
  - たいてい，上記がベース（例：Intel MPIはMPICH/MVAPICHベース）
  - 注意点：メーカ独自の機能拡張がなされていることがある

MPIによる通信：郵便物の郵送と同様

- 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

代表的なMPI関数

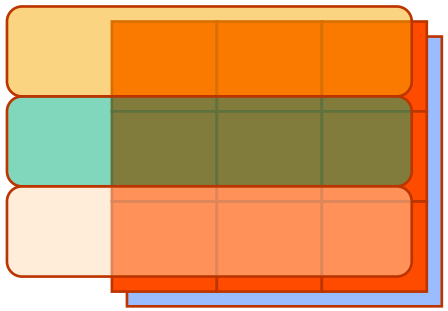
- システム関数
  - MPI_Init(), MPI_Comm_rank(), MPI_Comm_size(), MPI_Finalize()
- 1対1通信関数
  - ブロッキング型: MPI_Send(), MPI_Recv()
  - ノンブロッキング型: MPI_Isend(), MPI_Irecv()
- 1対全通信関数
  - MPI_Bcast()
- 集団通信関数
  - MPI_Reduce(), MPI_Allreduce(), MPI_Barrier()
- 時間計測関数
  - MPI_Wtime()

頻出するMPI用語

- MPIは「プロセス」間の通信を行う
  - MPIプロセス，あるいは単にプロセスと呼ぶ
  - プロセスは（普通は）プロセッサ（あるいはコア）に1対1で割り当てられる
- ランク（Rank）
  - 各MPIプロセスの認識番号
  - 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rank）に，$0 \sim N_p - 1$の数値が入る
    （注意： Fortranでもランクは0から始まる）
  - 全MPIプロセス数N_pを知るには，MPI_Comm_size()関数を使う（サンプルプログラムではsize）

コミュニケータ

- 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，$0 \sim N_p - 1$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
  - MPI_COMM_WORLD という名前のコミュニケータ
- MPI_Comm_split()関数によってコミュニケータを分割可能
  - メッセージを一部のプロセッサ群のみに放送するときに利用
  - “マルチキャスト”で利用



並列プログラミングの基礎（座学）

- 並列プログラミングの基礎
- 性能評価指標
- MPI (Message Passing Interface)
- 基礎的なMPI関数
- データ分散方式

C言語インタフェースとFortranインタフェースの違い（1/2）

- C言語版は，整数変数errが戻り値
  ```
  err = MPI_Xxxx(…);
  ```
- Fortran版は，整数変数errが最後の引数
  ```
  call MPI_Xxxx(…, err)
  ```
- システム用配列の確保方法
  - C言語
    ```
    MPI_Status status;
    ```
  - Fortran
    ```
    integer :: status(MPI_STATUS_SIZE)
    ```
  - Fortran 2008
    ```
    type(MPI_Status) :: status
    ```

C言語インタフェースとFortranインタフェースの違い（2/2）

- データ型の指定方法
  - C言語
    MPI_CHAR（文字型），MPI_INT（整数型），MPI_FLOAT（単精度の実数型），MPI_DOUBLE（倍精度の実数型）
  - Fortran
    MPI_CHARACTER（文字型），MPI_INTEGER（整数型），MPI_REAL（単精度の実数型），MPI_DOUBLE_PRECISION(=MPI_REAL8)（倍精度実数型），MPI_COMPLEX（複素数型）

- 以降は，C言語インタフェースで説明する

MPI_Send

- err = MPI_Send(*buf, count, datatype, dest, tag, comm);
  - buf: 送信領域の先頭アドレスを指定
  - count: 整数型．送信領域のデータ要素数を指定
  - datatype: MPI_Datatype型．送信領域のデータ型を指定
    - MPI_INT（整数型），MPI_FLOAT（単精度実数型），MPI_DOUBLE（倍精度実数型）など
  - dest: 整数型．送信先の（comm内での）プロセスランクを指定
  - tag: 整数型．メッセージにつけるタグの値を指定
  - comm: MPI_Comm型．コミュニケータを指定
    - 通常はMPI_COMM_WORLDを指定すればよい
  - err（戻り値）: 整数型．エラーコードが入る

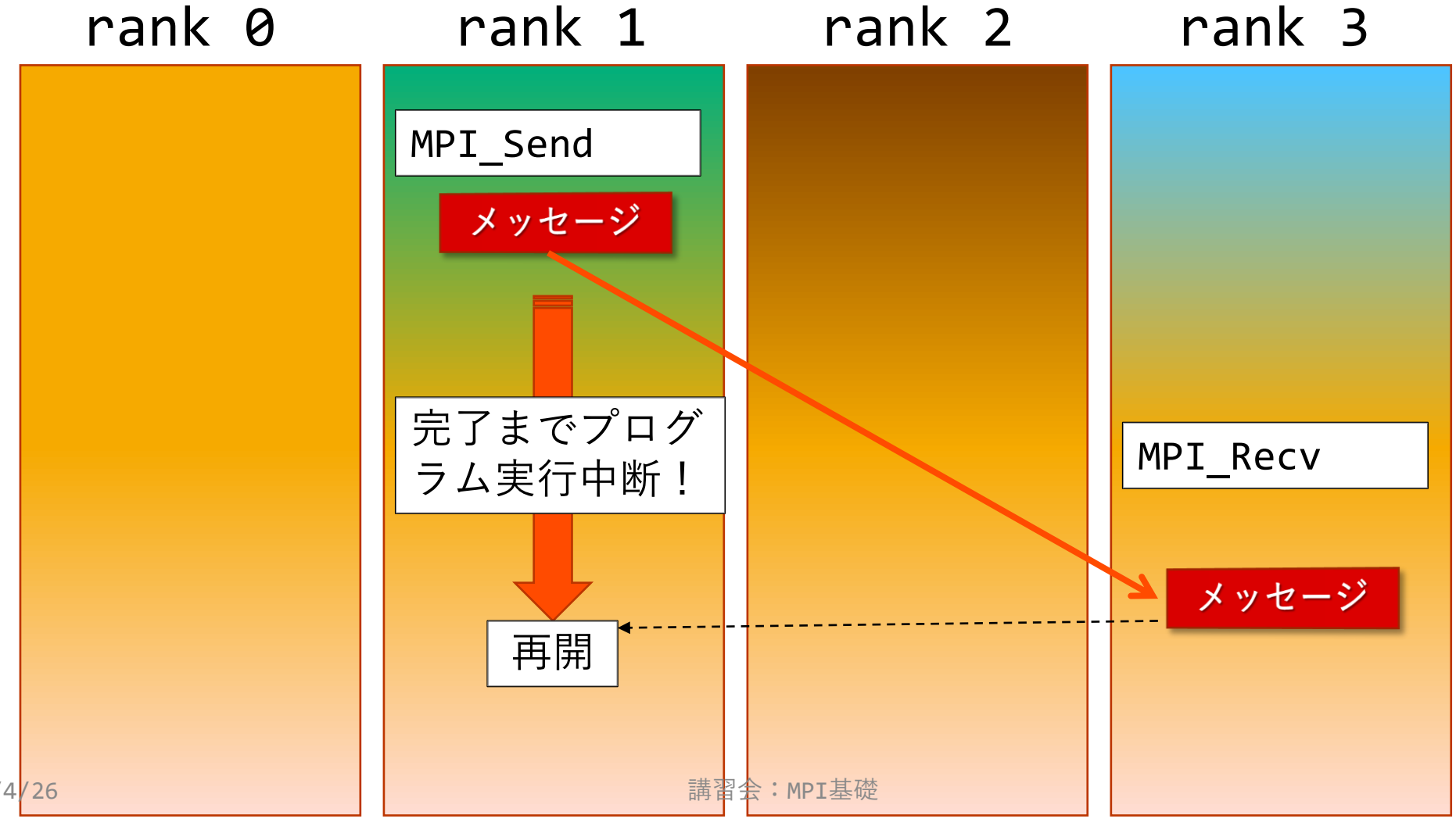
MPI_Recv (1/2)

- `err = MPI_Recv(*buf, count, datatype, source, tag, comm, *status);`
  - `buf`: 受信領域の先頭アドレスを指定
  - `count`: 整数型．受信領域のデータ要素数を指定
  - `datatype`: MPI_Datatype型．受信領域のデータ型を指定
  - `source`: 整数型．メッセージの送信元のランクを指定
    - 任意のランクから受信したいときは，`MPI_ANY_SOURCE`を指定
  - `tag`: 整数型．受信したいメッセージについているタグを指定
    - 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

MPI_Recv (2/2)

- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- status: MPI_Status型．受信状況に関する情報が入る
  - 必ず専用の型宣言をした配列を確保する
  - C言語： MPI_Status status;
  - Fortran: integer :: status(MPI_STATUS_SIZE)
  - Fortran 2008: type(MPI_Status) :: status
  - 要素数がMPI_STATUS_SIZEの整数配列が確保される
  - 受信メッセージの送信元のランクがstatus[MPI_SOURCE]，タグがstatus[MPI_TAG]に代入される
- err: 整数型．エラーコードが入る

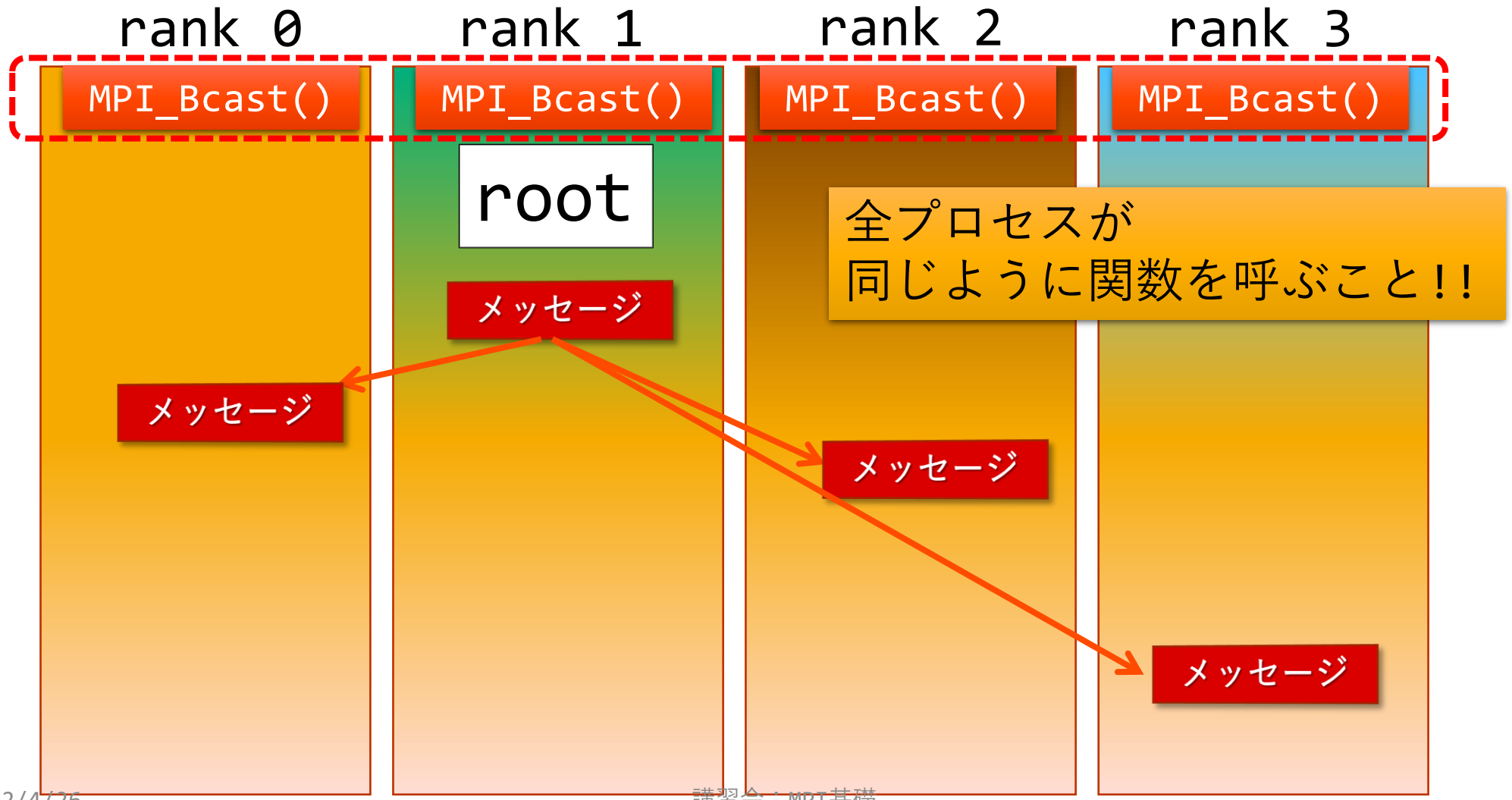
MPI_Send()--MPI_Recv()の動作（1対1通信）



MPI_Bcast

- err = MPI_Bcast(*buffer, count, datatype, root, comm);
  - buffer: 送信および受信領域の先頭アドレスを指定
  - count: 整数型．buffer上のデータ要素数を指定
  - datatype: MPI_Datatype型．bufferのデータ型を指定
  - root: 整数型．メッセージを送信するプロセスのIDを指定
    - 全プロセスが同じ値を指定する
  - comm: MPI_Comm型．通信に関与するコミュニケータを指定
  - err: 整数型．エラーコードが入る

MPI_Bcastの動作（集団通信）



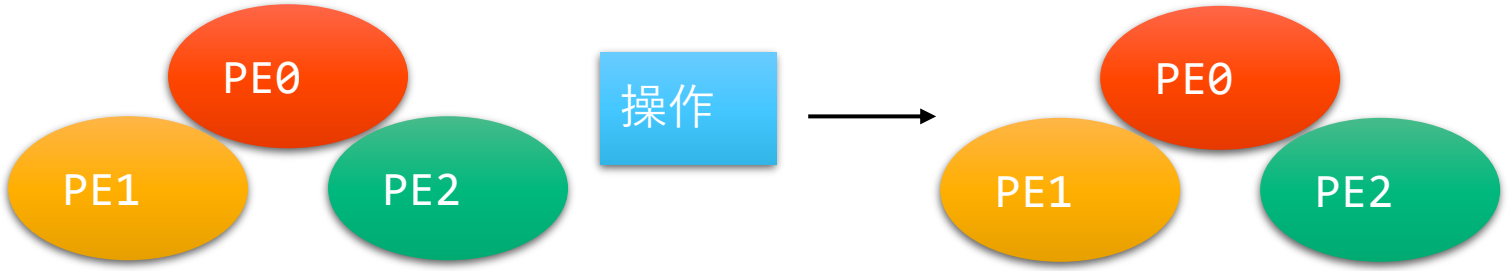
リダクション演算

- 「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）➔ 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）と呼ぶ
- 演算結果の持ち方の違いで，2種のインタフェースが存在
  - MPI_Reduce()
  - MPI_Allreduce()

リダクション演算の2種のインタフェース

- MPI_Reduce
  - リダクション演算の結果を，ある1つの代表プロセスに所有させる

- MPI_Allreduce
  - リダクション演算の結果を，全プロセスに所有させる



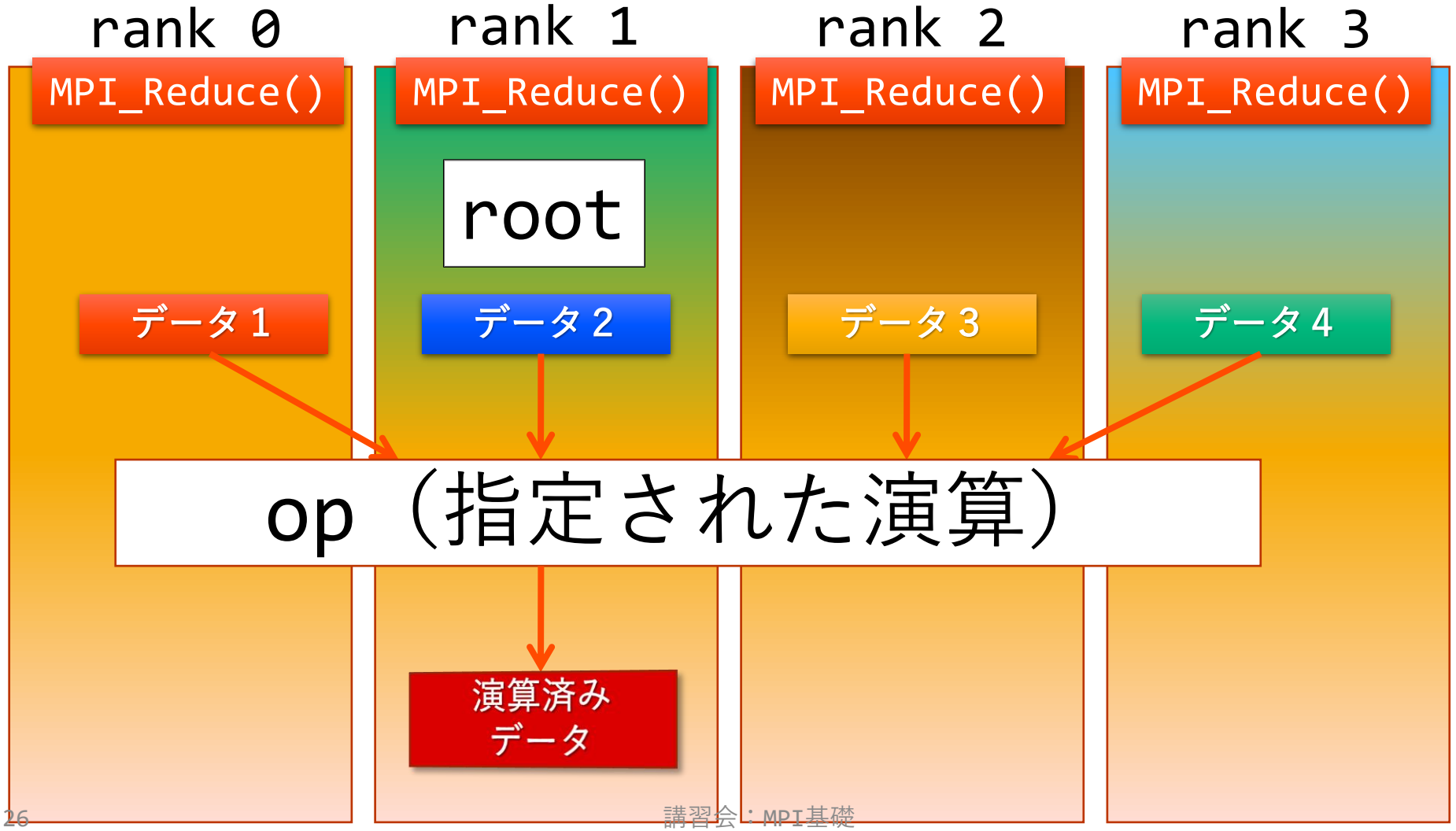
MPI_Reduce（1/2）

- err = MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op, root, comm);
  - sendbuf: 送信領域の先頭アドレスを指定
  - recvbuf: 受信領域の先頭アドレスを指定
    - rootで指定したプロセスのみで書き込まれる
    - 送信領域と受信領域は同一であってはならない（異なる配列を確保）
    - ↑ rootのsendbufとしてMPI_IN_PLACEを指定することで同一の領域を指定可能
  - count: 整数型．送信領域のデータ要素数を指定
  - datatype: MPI_Datatype型．送信領域のデータ型を指定
    - Fortranの場合：<最小・最大値と位置>を返す演算（MPI_MINLOCなど）を指定する場合は，MPI_2INTEGER（整数型），MPI_2REAL（単精度型），MPI_2DOUBLE_PRECISION(=MPI_2REAL8)（倍精度型）を指定

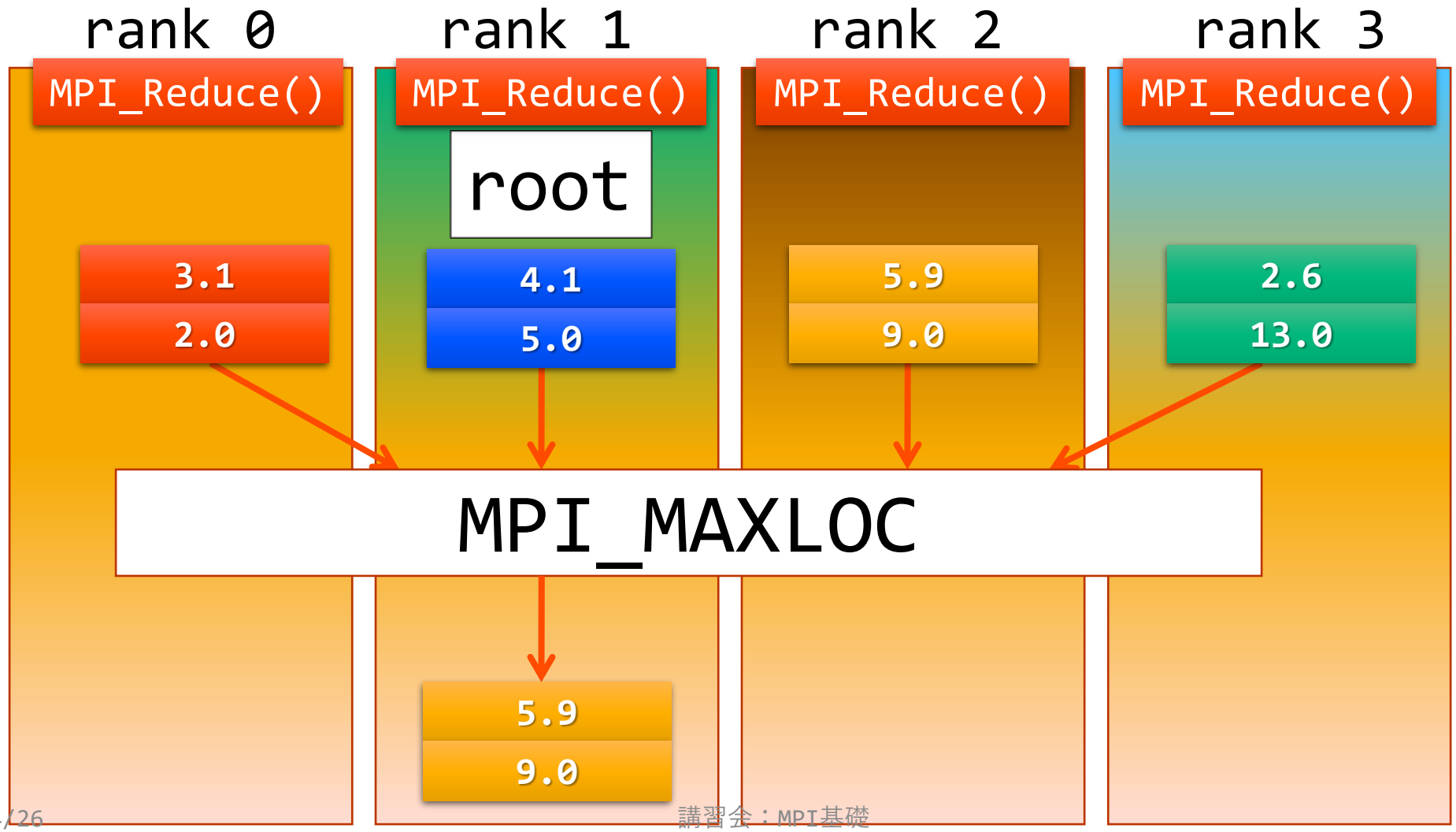
MPI_Reduce（2/2）

- op: MPI_Op型．演算の種類を指定
  - MPI_SUM: 総和
  - MPI_PROD: 積
  - MPI_MAX: 最大値
  - MPI_MIN: 最小値
  - MPI_MAXLOC: 最大値とその位置
  - MPI_MINLOC: 最小値とその位置
- root: 整数型．結果を受け取るプロセスのランクを指定
  - comm内の全プロセスが同じ値を指定する必要がある
- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- err: 整数型．エラーコードが入る

MPI_Reduceの動作（集団通信演算）



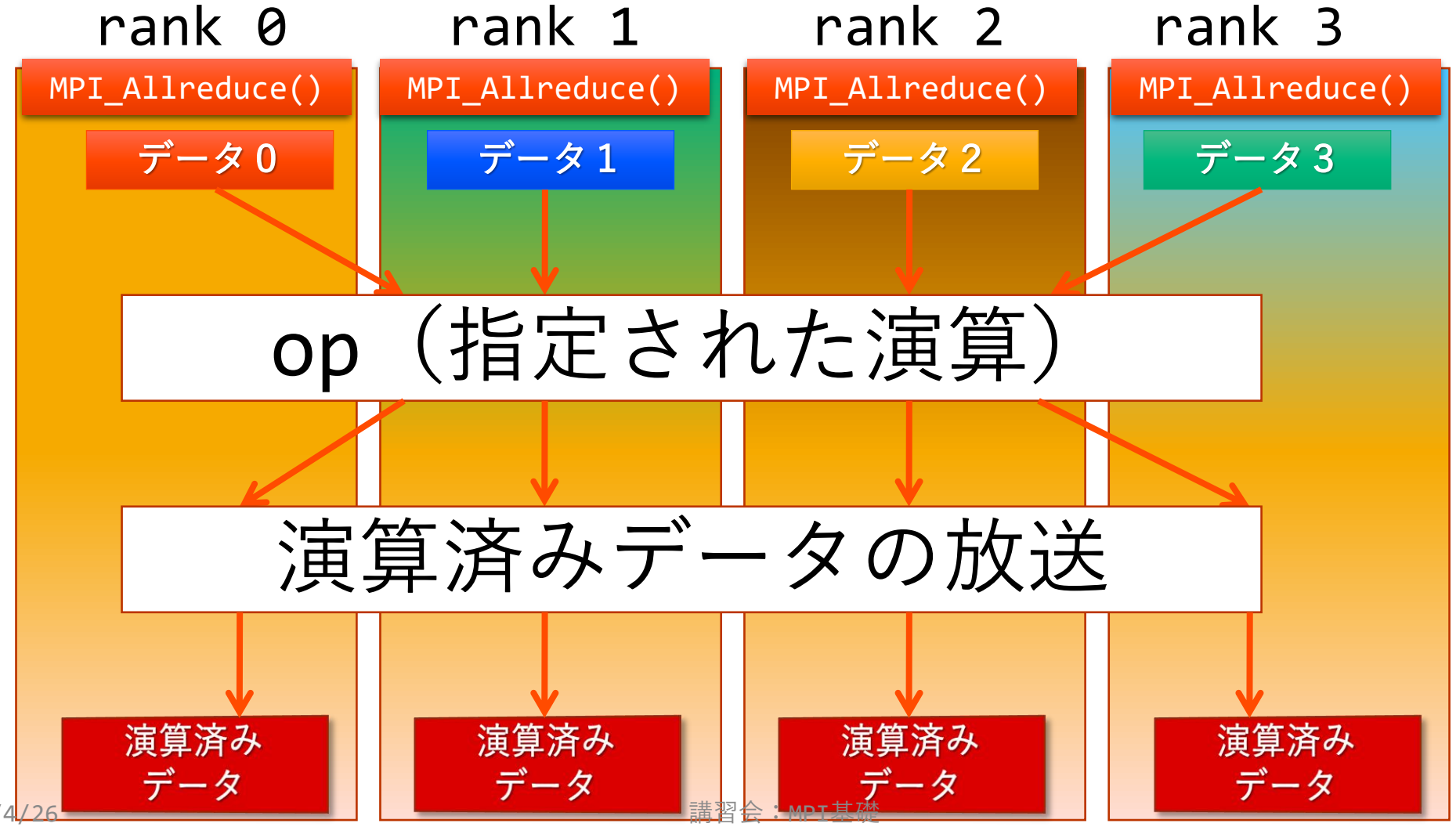
MPI_Reduceによる2リスト処理例



MPI_Allreduce

- err = MPI_Allreduce(*sendbuf, *recvbuf, count, datatype, op, comm);
  - sendbuf: 送信領域の先頭アドレスを指定
  - recvbuf: 受信領域の先頭アドレスを指定
    - 送信領域と受信領域は同一であってはならない（異なる配列を確保）
    - ↑ sendbufとしてMPI_IN_PLACEを指定することで同一の領域を指定可能
  - count: 整数型．送信領域のデータ要素数を指定
  - datatype: MPI_Datatype型．送信領域のデータ型を指定
  - op: MPI_Op型．演算の種類を指定
  - comm: MPI_Comm型．通信に関与するコミュニケータを指定
  - err: 整数型．エラーコードが入る

MPI_Allreduceの動作（集団通信演算）



リダクション演算の性能について

- リダクション演算は，1対1通信に比べて遅い
  - プログラム中で多用すべきではない
- MPI_Allreduce()はMPI_Reduce()に比べて遅い
  - MPI_Allreduce()は，MPI_Reduce()+MPI_Bcast()に相当
    - MPI_Reduce()+MPI_Bcast()を自分で書くよりは高速
  - なるべく，MPI_Reduce()を使う
    - 注：結果を全プロセスに放送する処理をなるべく減らすようにという意図であり，MPI_Allreduce()をMPI_Reduce()+MPI_Bcast()でばらして実装するように，という意図ではない

MPI_Gather（1/2）

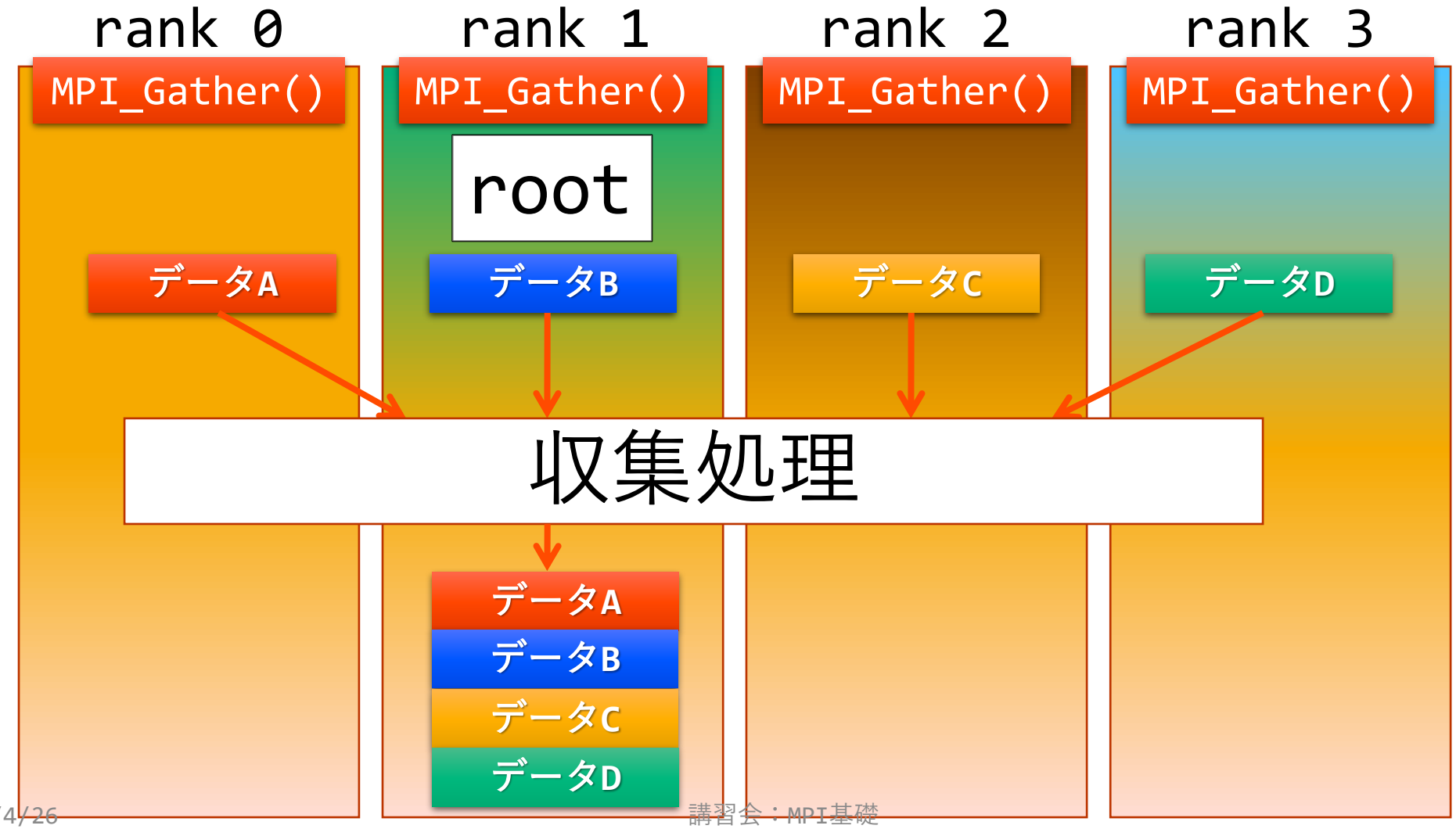
- err = MPI_Gather(*sendbuf, sendcount, sendtype, *recvbuf, recvcount, recvtype, root, comm);
  - sendbuf: 送信領域の先頭アドレスを指定
  - sendcount: 整数型．送信領域のデータ要素数を指定
  - sendtype: MPI_Datatype型．送信領域のデータ型を指定
  - recvbuf: 受信領域の先頭アドレスを指定
    - （原則として）送信領域と受信領域は同一であってはならない
    - ↑ rootのsendbufとしてMPI_IN_PLACEを指定することで同一の領域を指定可能
  - recvcount: 整数型．受信領域のデータ要素数を指定
    - 1プロセスから受信する要素数
    - MPI_Gatherでは全プロセスのメッセージサイズは同じでないといけない

MPI_Gather（2/2）

- recvtype: MPI_Datatype型．受信領域のデータ型を指定
- root: 整数型．結果を受け取るプロセスのランクを指定
  - comm内の全プロセスが同じ値を指定する必要がある
- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- err: 整数型．エラーコードが入る

- recvbuf, recvcount, recvtypeの指定
  - rootでのみ意味を持つ（他のプロセスでの指定は無視される）
  - （実用上は）全プロセスがrootでの指定値を書いておけばよい

MPI_Gatherの動作（集団通信）



MPI_Scatter（1/2）

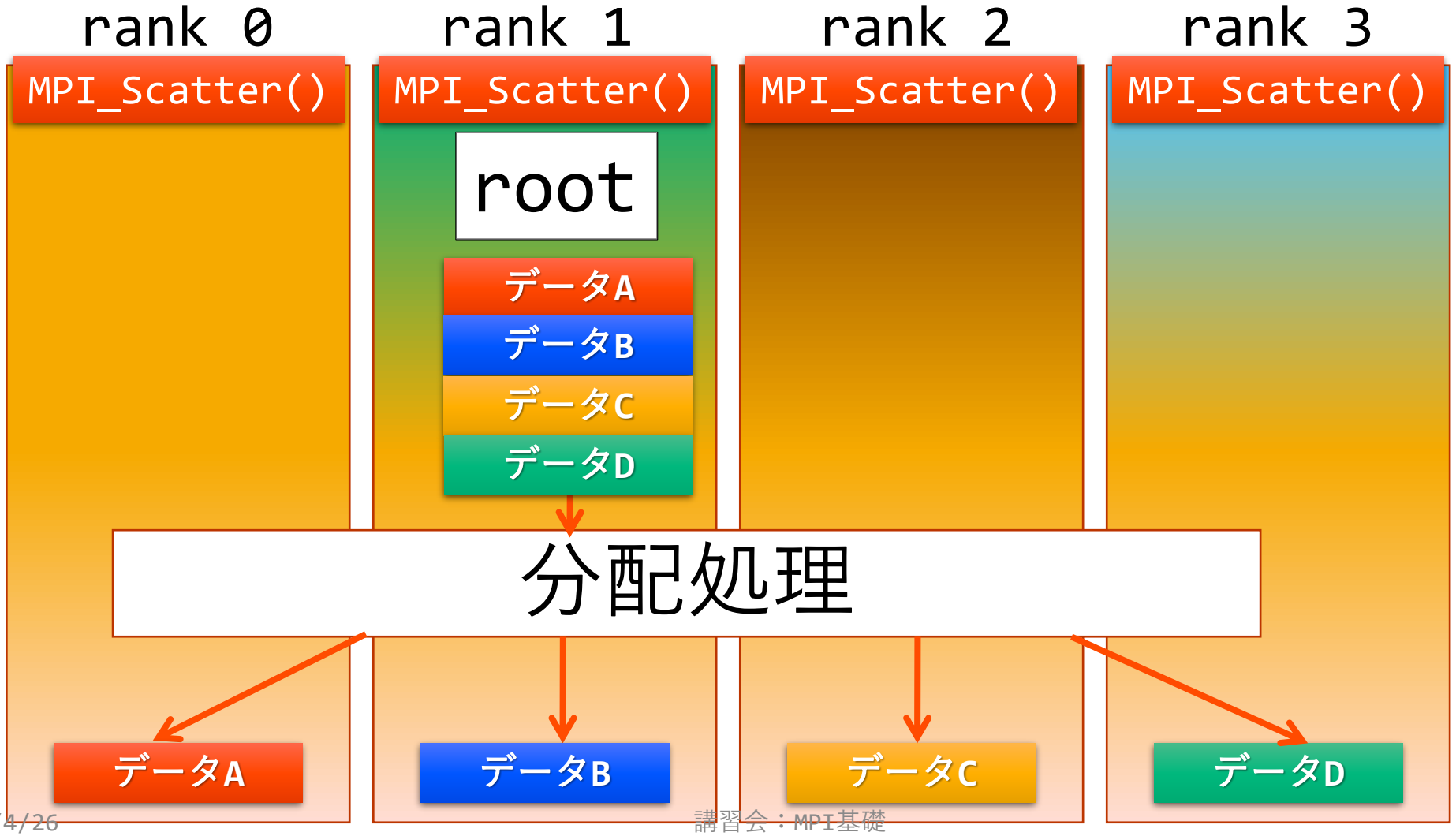
- `err = MPI_Scatter(*sendbuf, sendcount, sendtype, *recvbuf, recvcount, recvtype, root, comm);`
  - `sendbuf`: 送信領域の先頭アドレスを指定
  - `sendcount`: 整数型．送信領域のデータ要素数を指定
    - 1プロセス宛てに送信する要素数
    - MPI_Scatterでは全プロセスに配るメッセージサイズは同じでないといけない
  - `sendtype`: MPI_Datatype型．送信領域のデータ型を指定
  - `recvbuf`: 受信領域の先頭アドレスを指定
    - （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
    - ↑ rootのrecvbufとして`MPI_IN_PLACE`を指定することで同一の領域を指定可能
  - `recvcount`: 整数型．受信領域のデータ要素数を指定

MPI_Scatter（2/2）

- recvtype: MPI_Datatype型．受信領域のデータ型を指定
- root: 整数型．結果を受け取るプロセスのランクを指定
  - comm内の全プロセスが同じ値を指定する必要がある
- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- err: 整数型．エラーコードが入る

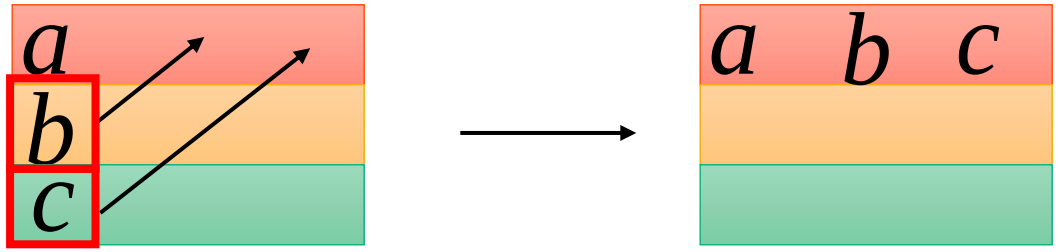
- sendbuf, sendcount, sendtypeの指定
  - rootでのみ意味を持つ（他のプロセスでの指定は無視される）
  - （実用上は）全プロセスがrootでの指定値を書いておけばよい

MPI_Scatterの動作（集団通信）

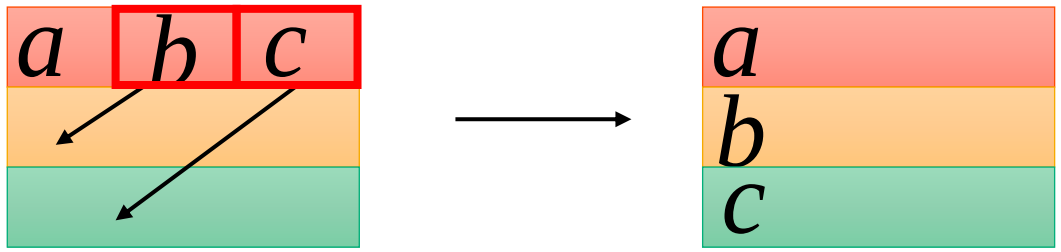


使用例：行列の転置

- 行列Aが（Block, *）分散されている場合を考える
- MPIで行列Aの転置行列A^Tを作る方法：
  - MPI_Gather()



  - MPI_Scatter()



集めるメッセージサイズが
全プロセスで均一のとき使う

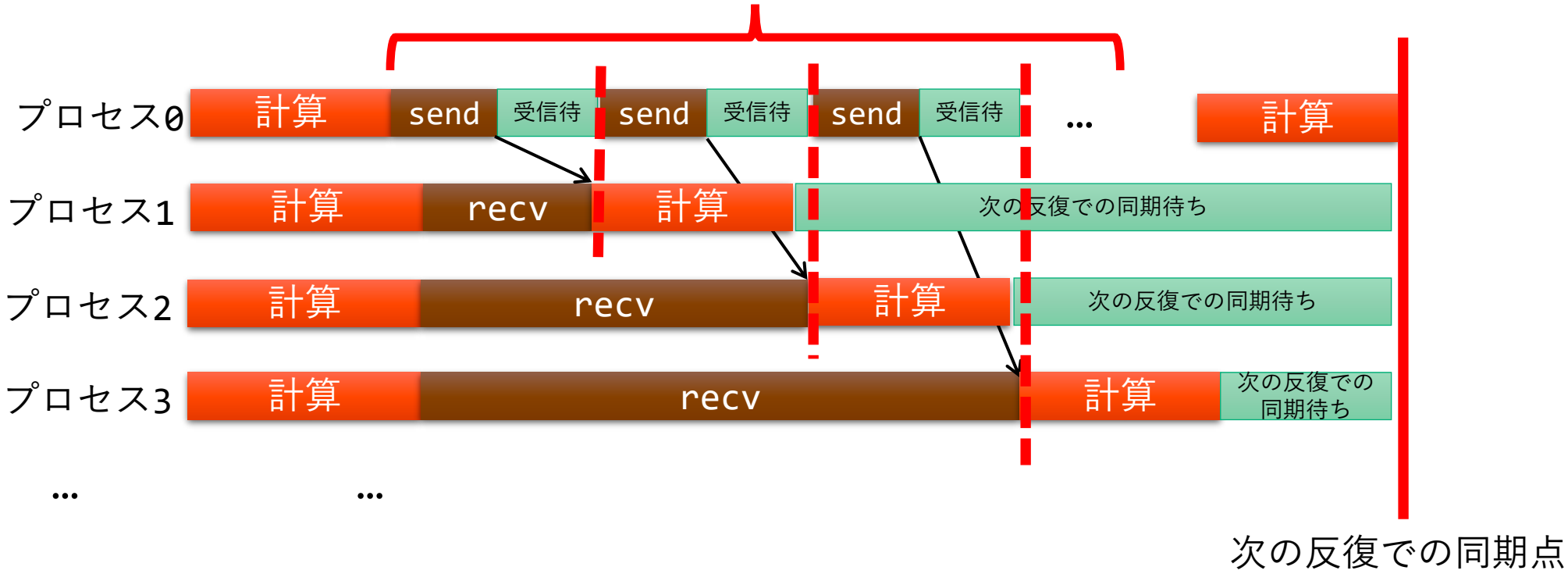
集めるサイズが全プロセスで均一でないとき：
MPI_GatherV関数
MPI_ScatterV関数

ブロッキングとノンブロッキング

1. ブロッキング
   - 送信・受信側のバッファ領域にメッセージが格納され，受信・送信側のバッファ領域が自由にアクセス・上書きできるまで呼び出しが戻らない
   - バッファ領域上のデータの一貫性を（MPI側で）保障
   - MPI_Send, MPI_Bcastなど
2. ノンブロッキング
   - 送信・受信側のバッファ領域のデータを保障せず，すぐに呼び出しが戻る
   - （MPI側では）バッファ領域上のデータの一貫性を保障しない
     - 一貫性の保証はユーザの責任
   - MPI_Isend, MPI_Ibcastなど

ブロッキング通信では効率が悪い例

- プロセス0だけが必要なデータを持っている場合
  - 連続するsendで，効率の悪い受信待ち時間が多発



MPI_Isend（1/2）

- ノンブロッキング通信関数の例として紹介
- `err = MPI_Isend(*buf, count, datatype, dest, tag, comm, *request);`
  - `buf`: 送信領域の先頭アドレスを指定
  - `count`: 整数型．送信領域のデータ要素数を指定
  - `datatype`: MPI_Datatype型．送信領域のデータ型を指定
  - `dest`: 整数型．送信先の（comm内での）プロセスランクを指定
  - `tag`: 整数型．メッセージにつけるタグの値を指定
  - `comm`: MPI_Comm型．コミュニケータを指定
    - 通常は`MPI_COMM_WORLD`を指定すればよい

MPI_Isend（2/2）

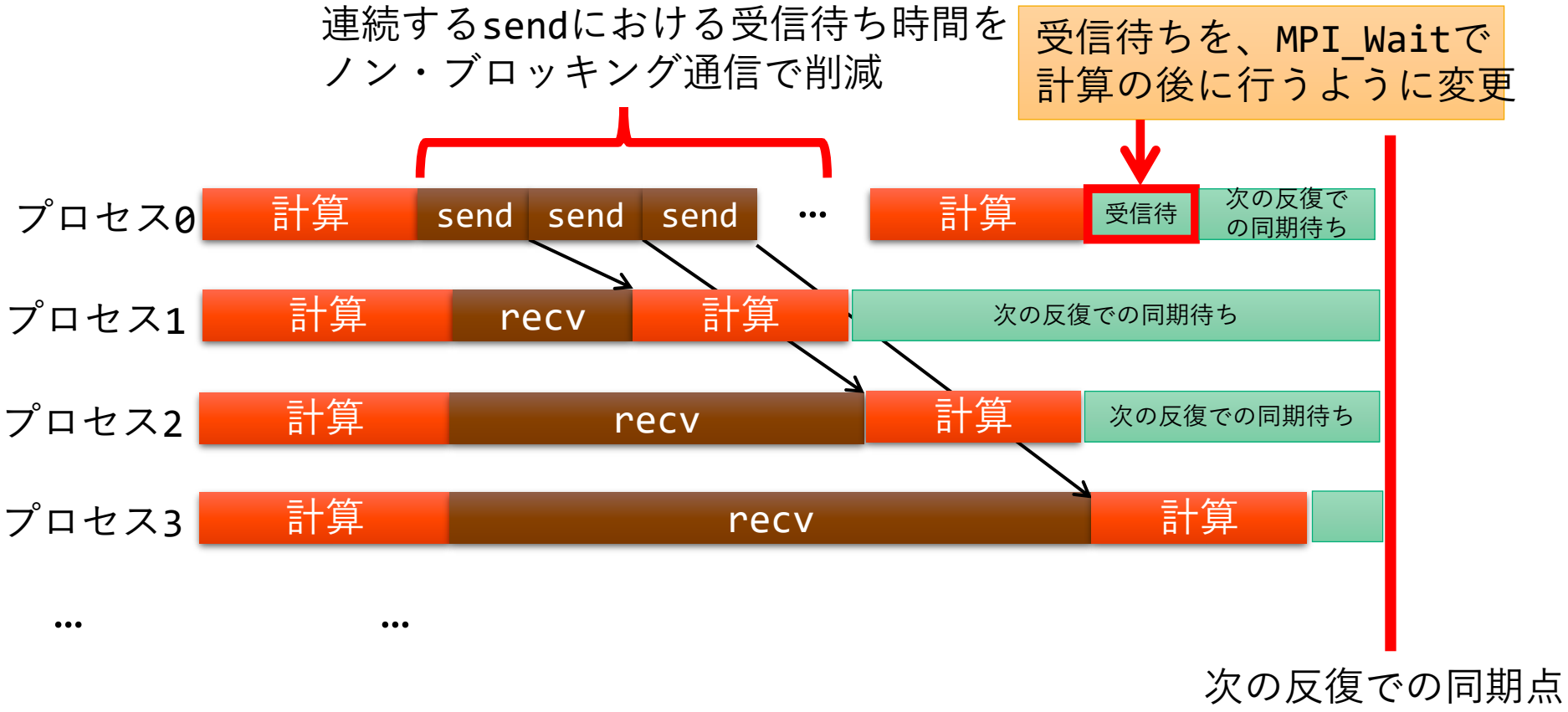
- request: MPI_Request型．送信を要求したメッセージにつけられた識別子が戻る
- err（戻り値）：整数型．エラーコードが入る

MPI_Wait

- MPI_Isend/MPI_Irecvだけではバッファ領域上のデータの一貫性を保障できないため，同期待ち関数が必要
- `err = MPI_Wait(*request, *status);`
  - `request`: MPI_Request型．送信を要求したメッセージにつけられた識別子
  - `status`: MPI_Status型．受信状況に関する情報が入る
  - 送信データを変更する前，受信データを読み出す前には必ず呼ぶこと

ノンブロッキング通信による改善

- プロセス0だけが必要なデータを持っている場合



MPI_Send()とMPI_Isend()の違い

- MPI_Send()関数
  - 関数中にMPI_Wait()関数が入っている
  - したがって，ユーザがMPI_Wait()を発行する必要はない
- MPI_Isend()関数
  - 関数中にMPI_Wait()関数が入っていない
  - かつ，すぐにユーザプログラムに戻る

MPIに関する資料

- 「MPI上級編」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/142/
  - 講義パートの動画が公開されています
- MPI仕様書（3.1）
  - https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
- MPI仕様書（4.0）
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
- Intel MPI
  - ガイド，一部サンプルもあり https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-linux/top.html
  - リファレンス，環境変数など https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top.html

並列プログラミングの基礎（座学）

- 並列プログラミングの基礎
- 性能評価指標
- MPI (Message Passing Interface)
- 基礎的なMPI関数
- データ分散方式

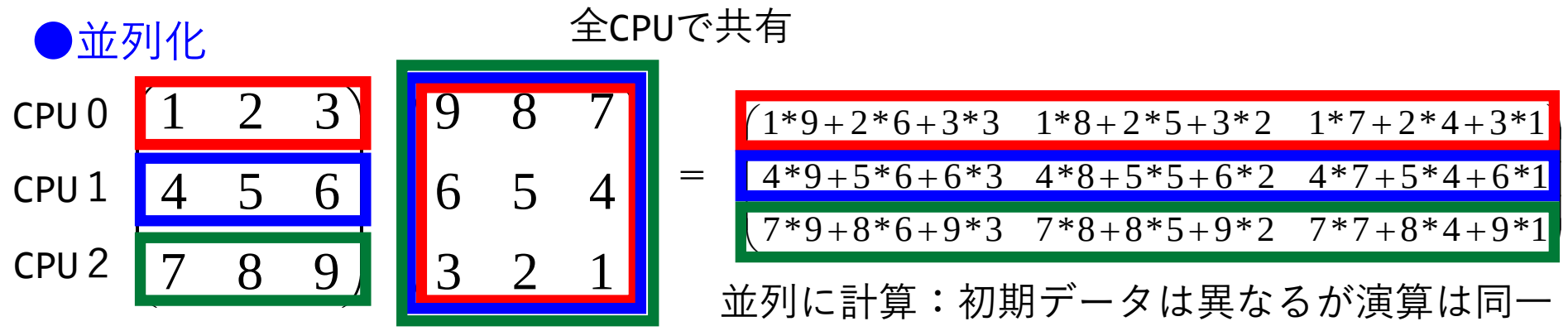
データ分散方式

- 逐次処理においては，「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
  1．各MPIプロセスの「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1つ
     - 粒度調整
  2．各MPIプロセスの「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MPIプロセスの「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）
- 例：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

並列化の考え方

- 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）は全プロセスで同一
  - データ並列の例：行列・行列積

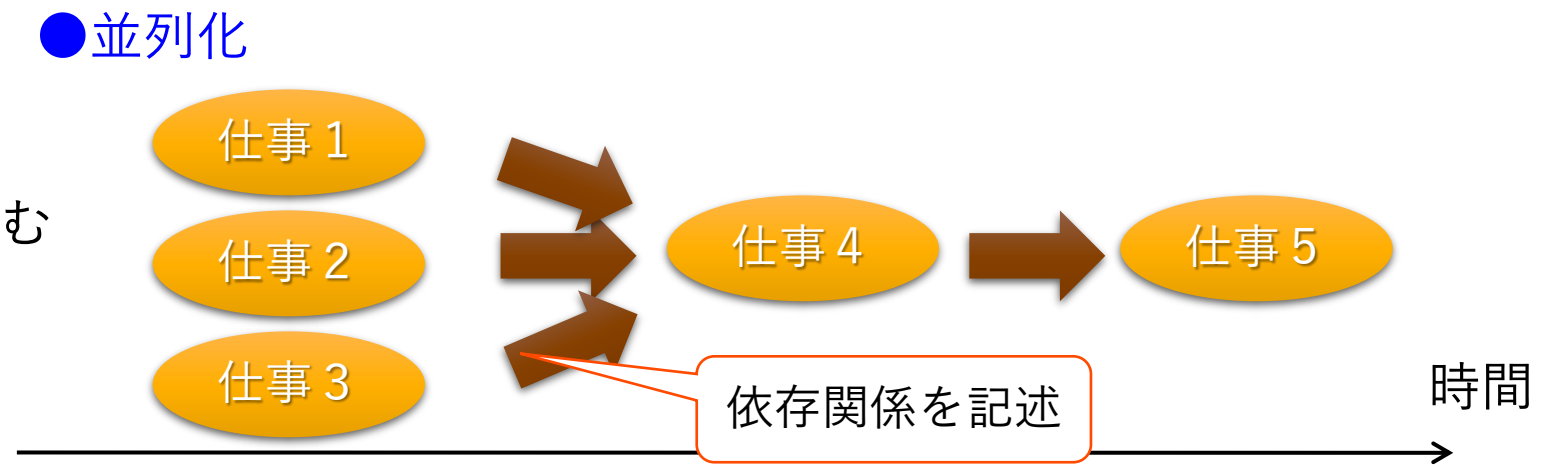
SIMDの
考え方と同じ



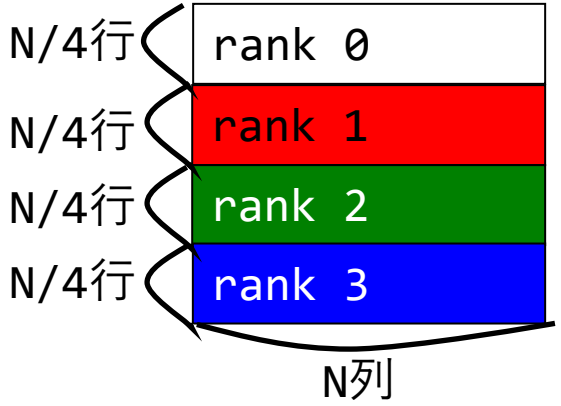
並列に計算：初期データは異なるが演算は同一

その他の並列化手法

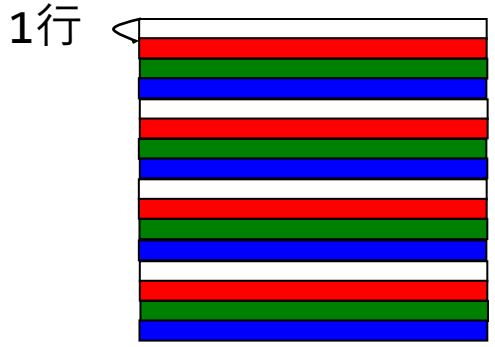
- タスク並列
  - タスク（ジョブ）を分割することで並列化
  - データの操作（=演算）はプロセスごとに異なるかもしれない
  - タスク並列の例：カレーを作る
    - 仕事1: 野菜を切る
    - 仕事2: 肉を切る
    - 仕事3: 水を沸騰させる
    - 仕事4: 野菜・肉を入れて煮込む
    - 仕事5: カレールウを入れる



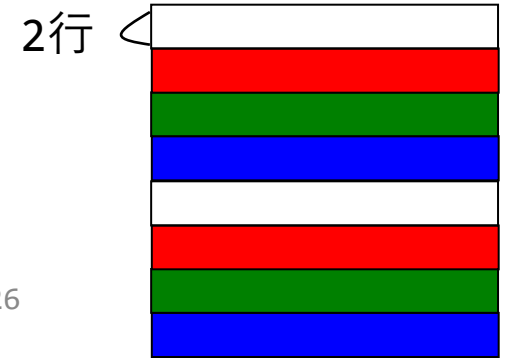
1次元分散



- （行方向） ブロック分割方式
- (Block, *) 分散方式



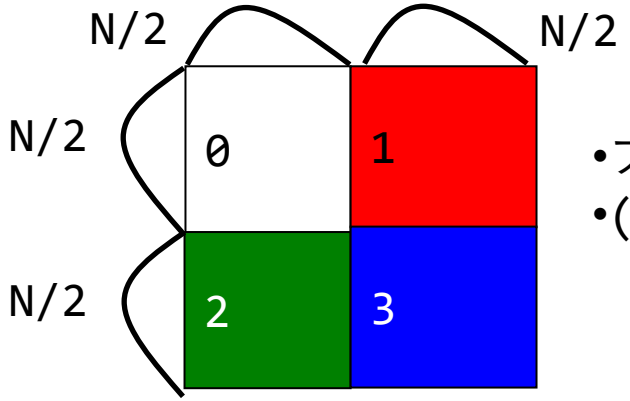
- (行方向) サイクリック分割方式
- (Cyclic, *) 分散方式



- (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

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

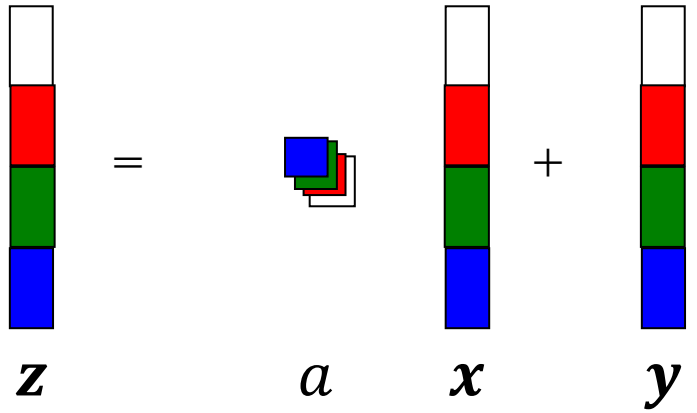
- サイクリック・サイクリック分割方式
- (Cyclic, Cyclic)分散方式

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

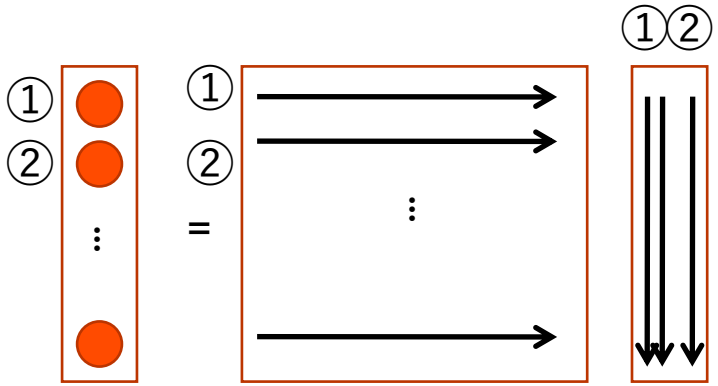
ベクトルどうしの演算

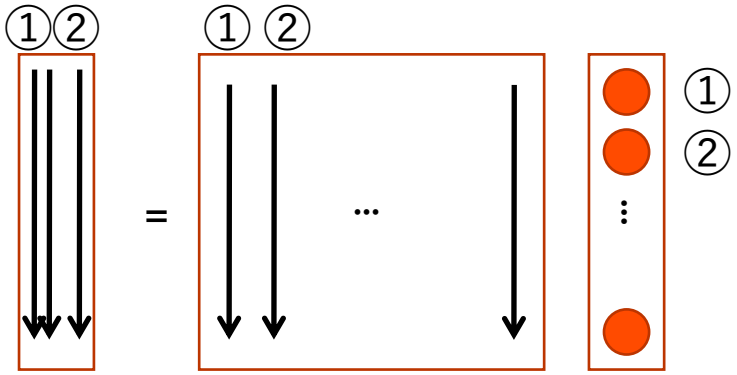
- 演算$\boldsymbol{z} = a\boldsymbol{x} + \boldsymbol{y}$を考える
  - ここで，aはスカラ，$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$はベクトル
- どのようなデータ分散方式でも並列処理が可能
  - ただし，スカラaは全プロセスで所有
  - ベクトルは$\mathcal{O}(N)$のメモリ領域が必要なのに対し，スカラは$\mathcal{O}(1)$のメモリ領域で大丈夫
    →スカラメモリ領域は無視できる
  - 計算量：$\mathcal{O}\left(\frac{N}{p}\right)$



行列とベクトルの積（1/3）

- 計算方式として，行方式と列方式がある
  - データ分散方式とメモリレイアウトの組み合わせによって性能が変化






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

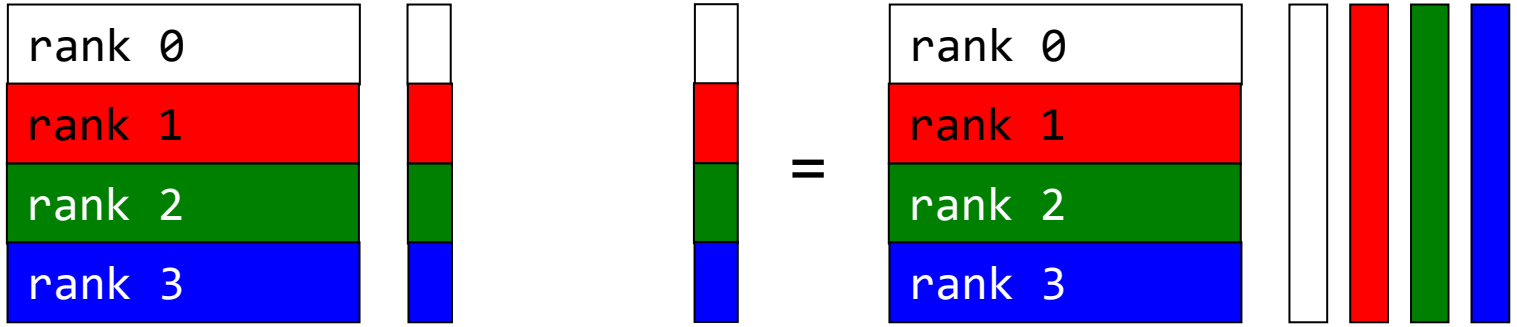
＜行方式＞：　C言語向き

```
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向き

行列とベクトルの積（2/3）：行方式の場合

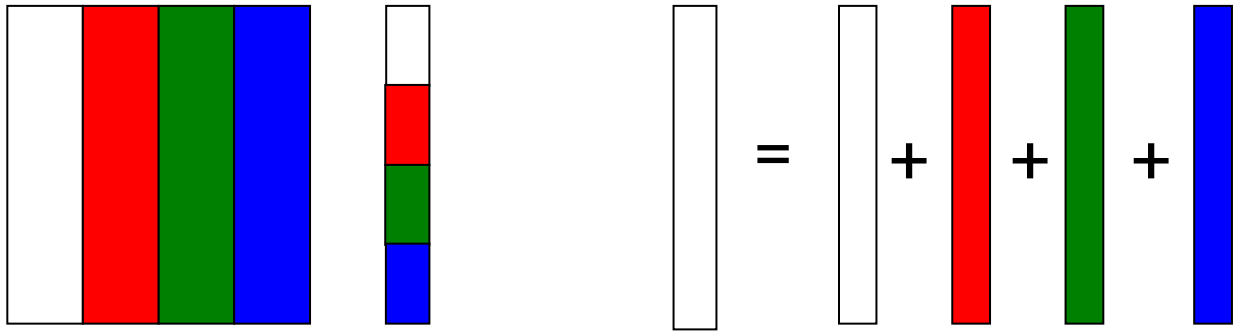
＜行方向分散方式＞　：行方式に向く分散方式



右辺ベクトルを MPI_Allgather 関数を利用し，全プロセスで所有

各プロセス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

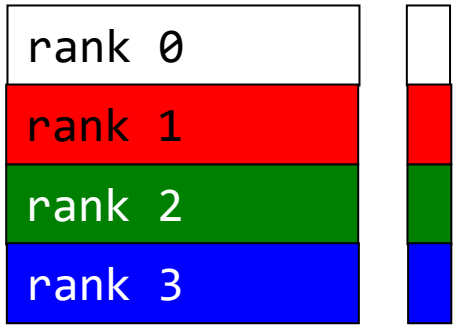


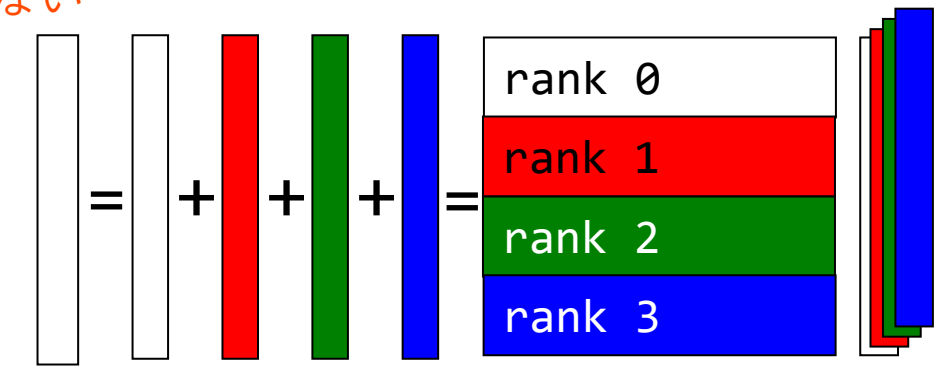
各プロセス内で行列-ベクトル積を行う

MPI_Reduce関数で総和を求める
（※あるプロセスにベクトルすべてが集まる）

行列とベクトルの積（3/3）：列方式の場合

＜行方向分散方式＞　：無駄が多く使われない

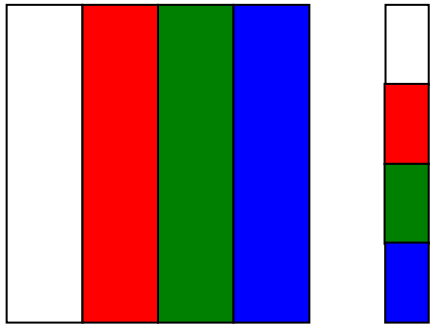


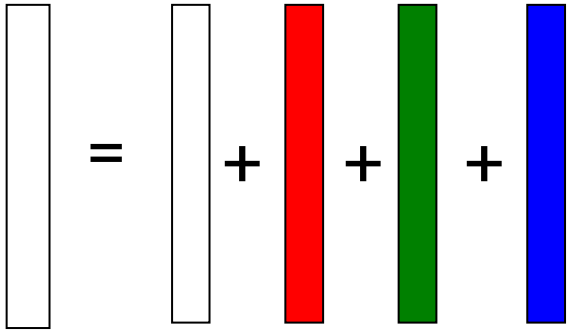


右辺ベクトルを MPI_Allgather 関数を利用し，全プロセスで所有

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





各プロセス内で行列-ベクトル積を行う

MPI_Reduce関数で総和を求める（※あるプロセスにベクトルすべてが集まる）