第190回 お試しアカウント付き 並列プログラミング講習会 「MPI上級編」

東京大学 情報基盤センター
塙　敏博

内容に関するご質問は
hanawa ＠ cc.u-tokyo.ac.jp
まで、お願いします。

講習会概略

- 開催日：　2022年10月12日（水）　10:00 – 17:30
- 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, Aquarius)
- 講習会プログラム：
  - 10:00 – 11:20　MPI概要、Wisteria/BDEC-01で使えるMPI実装
  - 11:30 - 12:30　ノンブロッキング通信、演習
  - (12:30 – 13:30　お昼休憩)
  - 13:30 – 14:30　派生データ型、MPI-IO、演習
  - 14:40 – 16:10　コミュニケータ、マルチスレッド、演習
  - 16:20 – 17:30　片側通信、演習

MPI
(Message Passing Interface)

おさらいも兼ねて

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、特定のソフトウエ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（Massively Parallel Processing (MPP)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（これまで）

- 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）
  - 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 http://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

MPIの経緯 MPI-3.1

実質的に現時点で使える最新版

- MPI-3.0 2012年9月
- MPI-3.1 2015年6月
- 以下のページで現状・ドキュメントを公開中
  - http://mpi-forum.org/docs/docs.html
  - https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
    - https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm (unofficial HTML版)
- 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）
  - Fortran2008 対応、など

MPIの経緯 MPI-4.0標準 (2021/6/9)

https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

- 新しい機能 (p.1046~1049, 34項目のアップデート)
- 主なもの
  - 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（Fault Tolerance, FT）

完全準拠の実装がいつ
使えるようになるか不明

MPIの実装

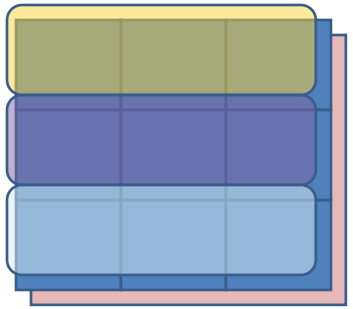
- MPICH（エム・ピッチ）
  - 米国アルゴンヌ国立研究所が開発
- MVAPICH (エムヴァピッチ)
  - 米国オハイオ州立大学で開発、MPICHをベース
  - InfiniBand向けの優れた実装
- OpenMPI
  - オープンソース
- ベンダMPI
  - 大抵、上のどれかがベースになっている
    例: 富士通「富岳」、Odyssey用のMPI: Open-MPIベース
    Intel MPI: MPICH、MVAPICHベース
  - 注意点:メーカ独自機能拡張がなされていることが多い

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。
- 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に、0～全プロセス数-1　の数値が入る
  - コミュニケータ中のMPIプロセス数を知るために，MPI_Comm_size関数を使う。

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～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に利用
  - “マルチキャスト”で利用
- 他にも様々な作成方法がある ➔ 後述



MPIに含まれるもの

MPI4.0の目次に相当 (4.0からのものは赤字)

- 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；
- Partitioned １対１通信関数
- 派生データ型
  - MPI_Type_
- 集団通信関数
  - MPI_Bcast ；
    MPI_Reduce； MPI_Allreduce；
    MPI_Barrier；
- グループ、コミュニケータ
  - MPI_Comm_dup ; MPI_Comm_split ;
- プロセストポロジ
  - MPI_Cart_create ;
- 環境の管理や表示
  - MPI_Init； MPI_Comm_rank；
    MPI_Comm_size； MPI_Finalize；
  - 時間計測関数
    - MPI_Wtime
- Infoオブジェクト
- プロセス生成・管理
- 片側通信
  - MPI_Put ; MPI_Get ;
- 外部インタフェース
- 並列ファイルIO (MPI-IO)
  - MPI_File_open,
- FortranとCのbinding
- ツールサポート

基本的なMPI関数

送信、受信のためのインタフェース

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)
[f2008~] TYPE(MPI_Status) :: istatus
```

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言語インタフェースで説明する

MPI関数を使うためのヘッダファイル等

- C言語
  ```
  #include <mpi.h>
  ```
- Fortran77
  ```
  include ’mpif.h’
  ```
- Fortran90,95,2003
  ```
  use mpi
  ```
- Fortran2008以降
  ```
  use mpi_f08
  ```

但し、引数が間違っていても
チェックされないので全くお勧めしません

引数の型がC言語と似たものに変わります

基礎的なMPI関数: 1対1通信—MPI_Recv

```
• ierr = MPI_Recv(recvbuf, count, datatype, source, tag,
                  comm, status);
```

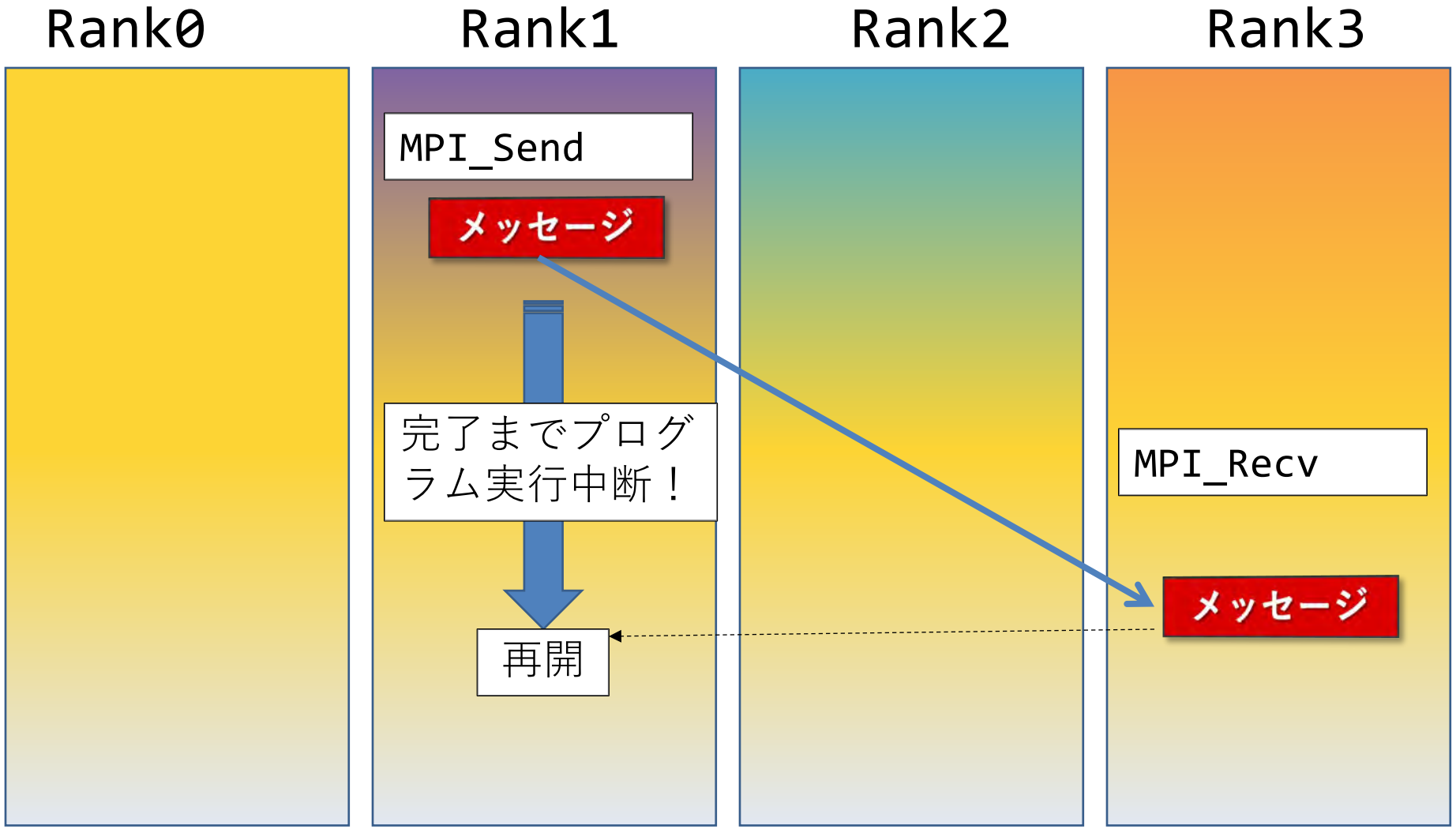
- void * recvbuf (OUT): 受信領域の先頭番地を指定する。
- int count (IN): 受信領域のデータ要素数を指定する。
- int datatype (IN): 受信領域のデータの型を指定する。
  - MPI_CHAR (文字型), MPI_INT (整数型),
    MPI_FLOAT (実数型), MPI_DOUBLE(倍精度実数型), など
- int source (IN): 受信したいメッセージを送信するプロセスのランク番号を指定する。
  - 任意のランクから受信したいときは、MPI_ANY_SOURCE を指定
- int tag (IN): 受信したいメッセージに付いているタグの値を指定。
  - 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_Comm comm (IN): コミュニケータを指定。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- MPI_Status status (OUT): 受信ステータス

基礎的なMPI関数: 1対1通信—MPI_Send

```
• ierr = MPI_Send(sendbuf, count, datatype,  dest,
                  tag, comm);
```

- void * sendbuf (IN): 送信領域の先頭番地を指定
- int count (IN): 送信領域のデータ要素数を指定
- int datatype (IN): 送信領域のデータの型を指定
- int dest (IN): 送信したい相手のランクを指定
  - 任意のランクから受信したいときは、MPI_ANY_SOURCE を指定
- int tag (IN): 受信したいメッセージに付いているタグの値を指定。
  - 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_Comm comm (IN): コミュニケータを指定。
  - 通常ではMPI_COMM_WORLD を指定すればよい。

Send－Recvの概念（１対１通信）



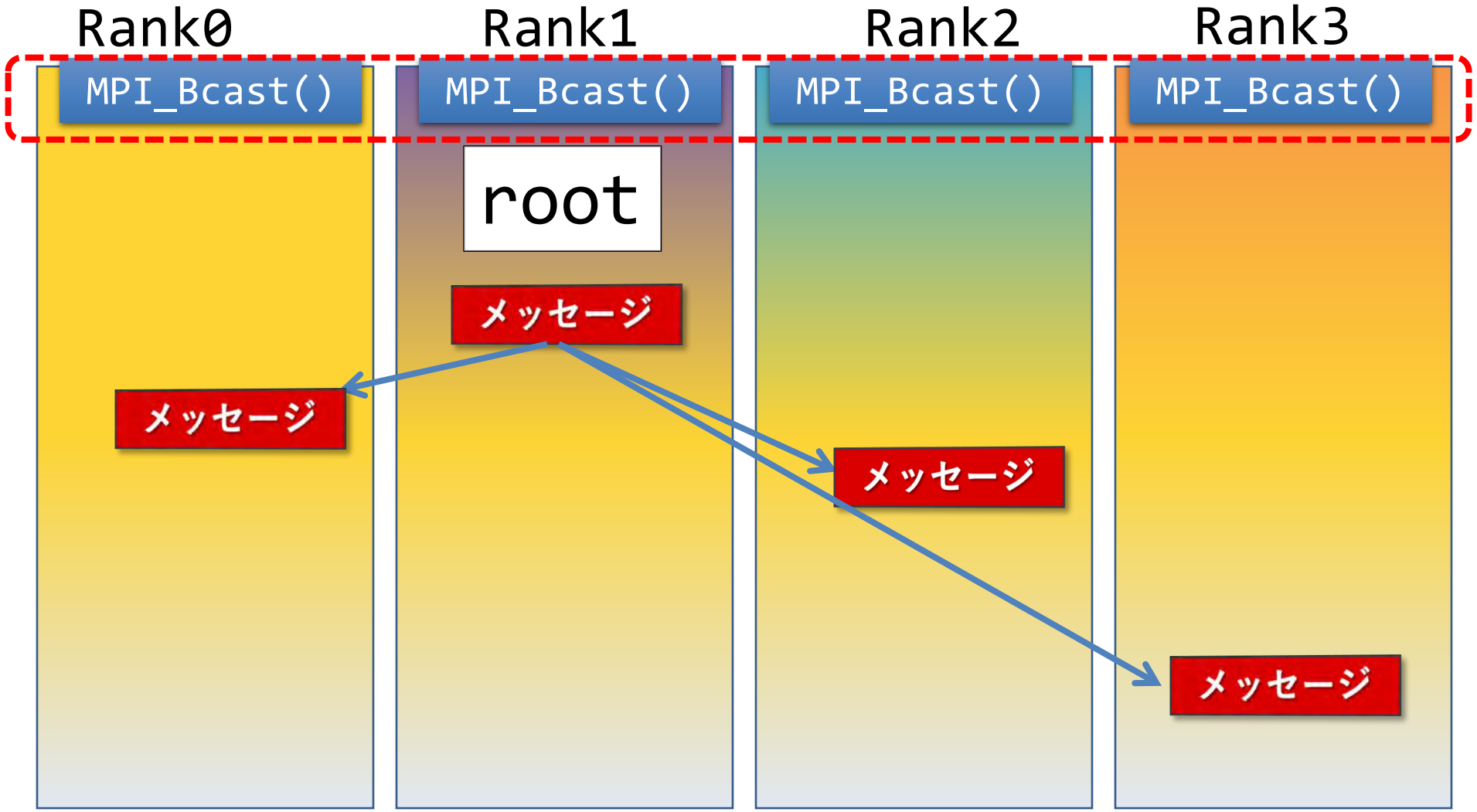
基礎的なMPI関数：集団通信—MPI_Bcast

```
• ierr = MPI_Bcast(sendbuf,  count,  datatype,
                   root,  comm);
```

- void * sendbuf (IN/OUT): 送信(root)/受信(root以外)領域の先頭番地
- int count (IN):　　送信領域のデータ要素数
- MPI_Datatype datatype (IN) : 送信領域のデータ型
- int root (IN):　送信プロセスのランク番号
- MPI_Comm comm (IN):　コミュニケータ

全ランクが
同じように関数を呼ぶこと!!

MPI_Bcastの概念（集団通信）



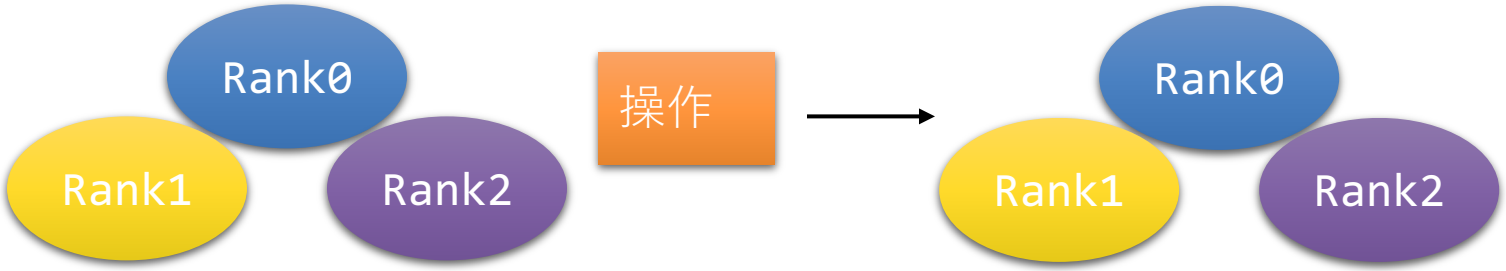
リダクション演算

- ＜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）　→　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ランクの違い
  - MPI_Reduce関数
    - リダクション演算の結果を、ある一つのランクに所有させる

  - MPI_Allreduce関数
    - リダクション演算の結果を、全てのランクに所有させる



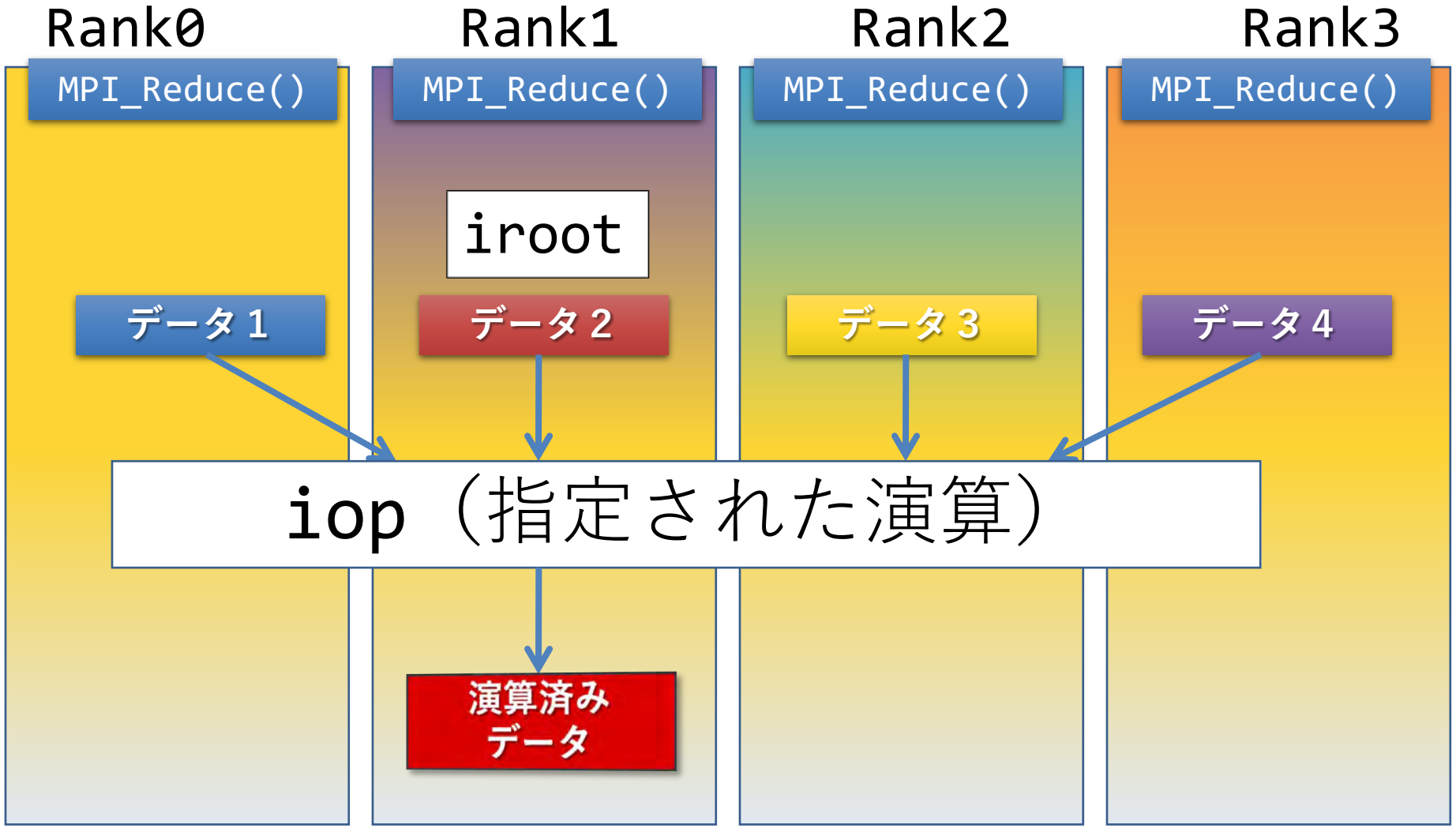
基礎的なMPI関数：集団通信—MPI_Reduce

- `ierr = MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);`

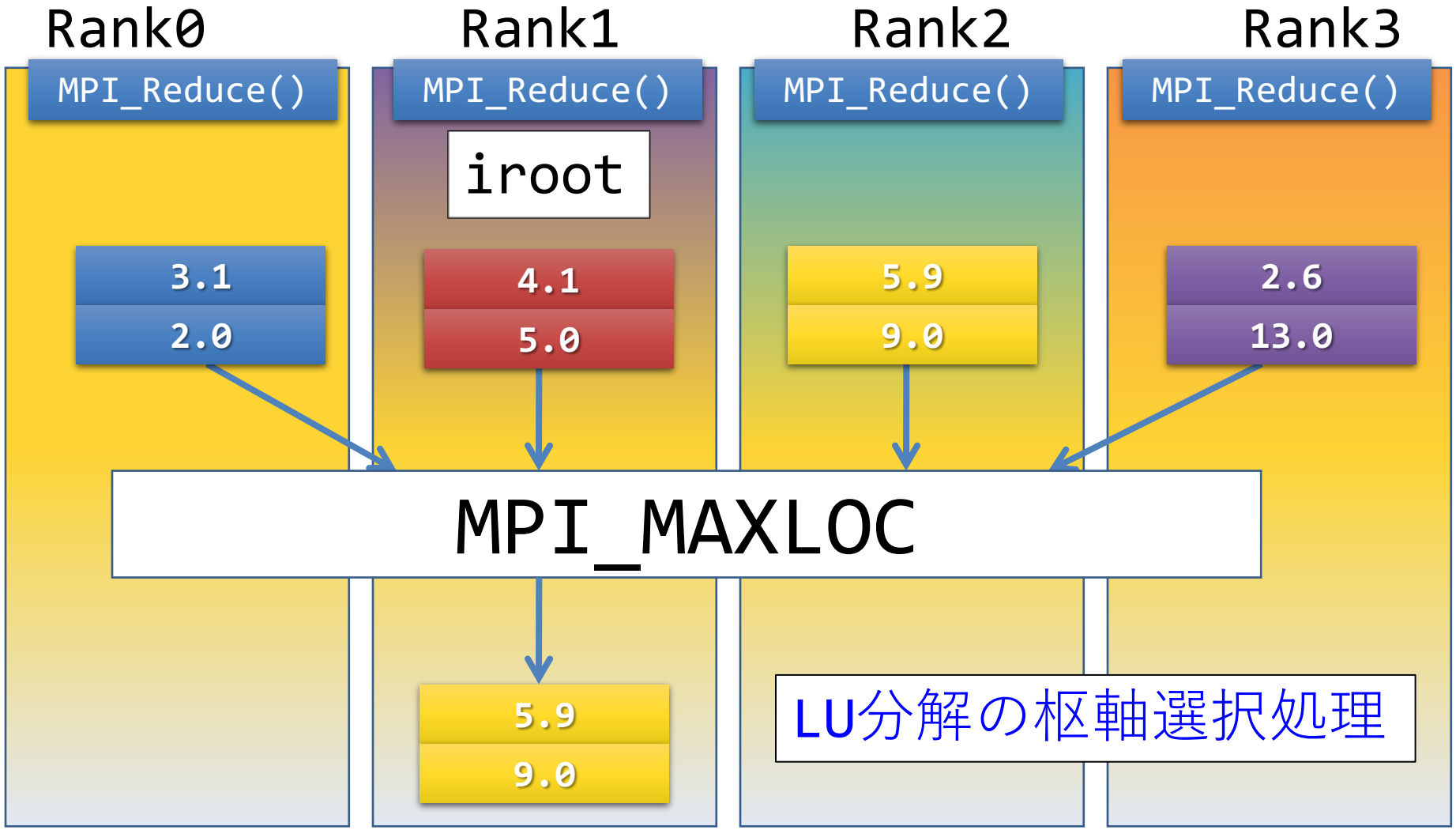
実は回避策はある（後述）

- void * sendbuf (IN) : 送信領域の先頭番地
- void * recvbuf (OUT) : 受信領域の先頭番地
  - 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、異なる配列を確保する必要がある。
- int count (IN) : 送信領域のデータ要素数
- MPI_Datatype datatype (IN) : 送信領域のデータ型
- MPI_Op op (IN): 演算の種類
  - MPI_SUM (総和), MPI_PROD (積), MPI_MAX (最大), MPI_MIN (最小), MPI_MAXLOC (最大とその位置), MPI_MINLOC (最小とその位置) など。
- int root (IN): 送信プロセスのランク番号
- MPI_Comm comm (IN): コミュニケータ

MPI_Reduceの概念（集団通信）



MPI_Reduceによる２リスト処理例
（MPI_2DOUBLE_PRECISION と MPI_MAXLOC）

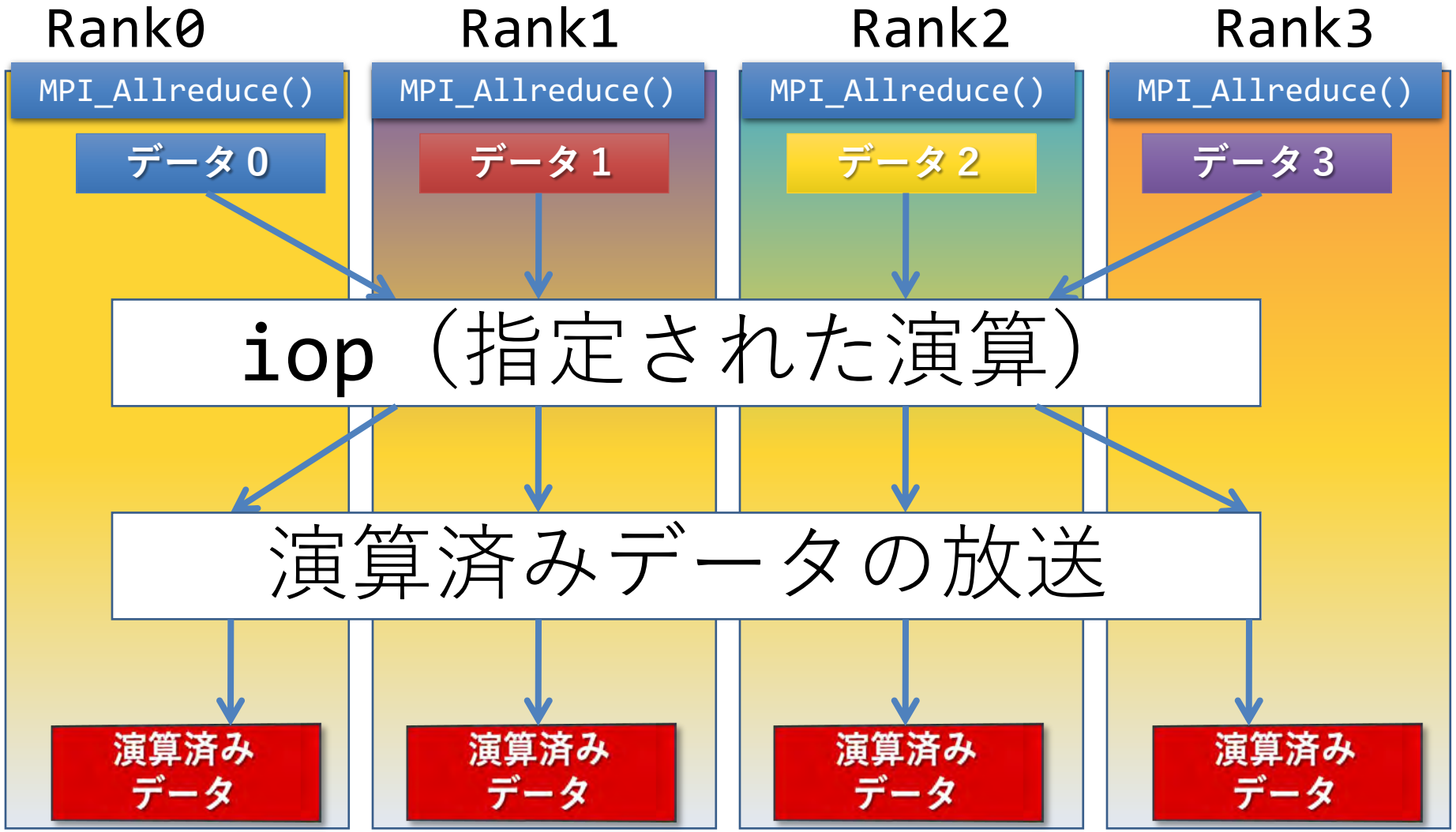


基礎的なMPI関数：集団通信―MPI_Allreduce

```
• ierr = MPI_Allreduce(sendbuf, recvbuf, count,
       datatype, op, comm);
```

- void * sendbuf (IN) : 送信領域の先頭番地
- void * recvbuf (OUT) : 受信領域の先頭番地
  - 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、異なる配列を確保する必要がある。
- int count (IN) : 送信領域のデータ要素数
- MPI_Datatype datatype (IN) : 送信領域のデータ型
- MPI_Op op (IN): 演算の種類
  - MPI_SUM (総和), MPI_PROD (積), MPI_MAX (最大), MPI_MIN (最小), MPI_MAXLOC (最大とその位置), MPI_MINLOC (最小とその位置) など。
- MPI_Comm comm (IN): コミュニケータ

MPI_Allreduceの概念（集団通信）



基礎的なMPI関数—MPI_Gather

```
• ierr = MPI_Gather ( sendbuf,  sendcount,  sendtype,
         recvbuf,  recvcount,  recvtype,  root,  comm);
```

- void * sendbuf (IN) :　送信領域の先頭番地
- int sendcount (IN) :　送信領域のデータ要素数
- MPI_Datatype sendtype (IN) : 送信領域のデータ型
- void * recvbuf (OUT) :　受信領域の先頭番地
  - 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、異なる配列を確保する必要がある。
- int recvcount (IN):　受信領域のデータ要素数
- MPI_Datatype recvtype (IN) : 受信領域のデータ型
  - root で指定したランクのみ有効 (recvbuf, recvcount, recvtype)
- int root (IN):　受信プロセスのランク番号
- MPI_Comm comm (IN):　コミュニケータ

```
sendcount * size
= recvcount
```

基礎的なMPI関数—MPI_Scatter

```
ierr = MPI_Scatter ( sendbuf,  sendcount,  sendtype,
       recvbuf,  recvcount,  recvtype,  root,  comm);
```

- void * sendbuf (IN) : 送信領域の先頭番地
- int sendcount (IN) : 送信領域のデータ要素数
- MPI_Datatype sendtype (IN) : 送信領域のデータ型
  root で指定したランクのみ有効 (sendbuf, sendcount, sendtype)
- void * recvbuf (OUT) : 受信領域の先頭番地
  - 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、異なる配列を確保する必要がある。
- int recvcount (IN): 受信領域のデータ要素数
- MPI_Datatype recvtype (IN) : 受信領域のデータ型
- int root (IN): 送信プロセスのランク番号
- MPI_Comm comm (IN): コミュニケータ

```
sendcount
= recvcount * size
```

ブロッキング、ノンブロッキング

1. ブロッキング

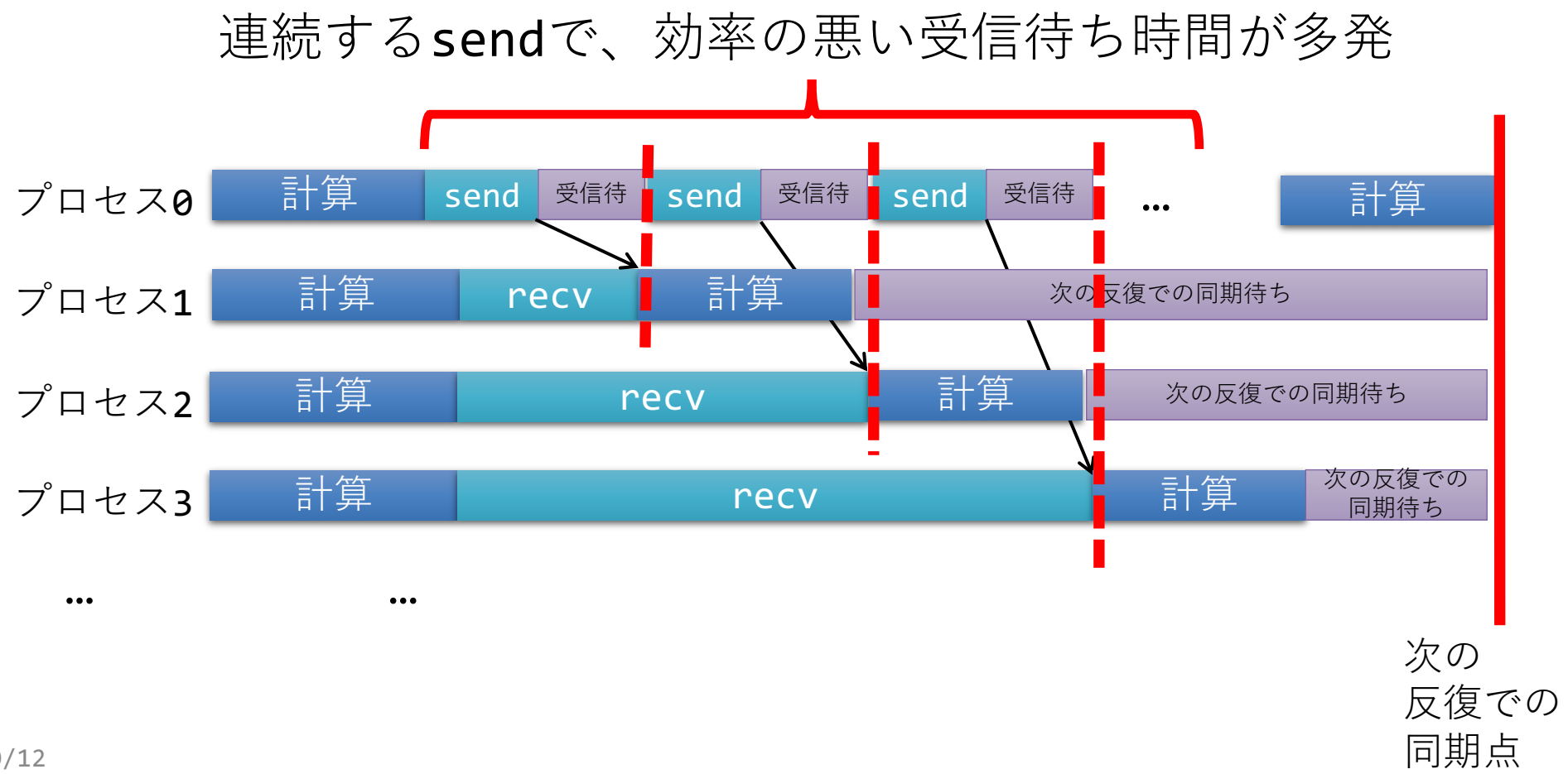
- 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない
- バッファ領域上のデータの一貫性を保障
- `MPI_Send`, `MPI_Bcast`など

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
- バッファ領域上のデータの一貫性を保障せず
  - 一貫性の保証はユーザの責任

ブロッキング通信で効率の悪い例

- プロセス0が必要なデータを持っている場合



ノンブロッキング通信関数— MPI_Isend

```
• ierr = MPI_Isend(sendbuf,  icount,  datatype,
     idest,  itag,  icomm,  irequest);
```

- 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
- datatype ： 整数型。送信領域のデータの型を指定する
- idest ： 整数型。送信したいランクの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。ランク集団を認識する番号であるコミュニケータを指定
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- irequest : MPI_Request型（整数型の配列）。送信を要求したメッセージにつけられた識別子が戻る

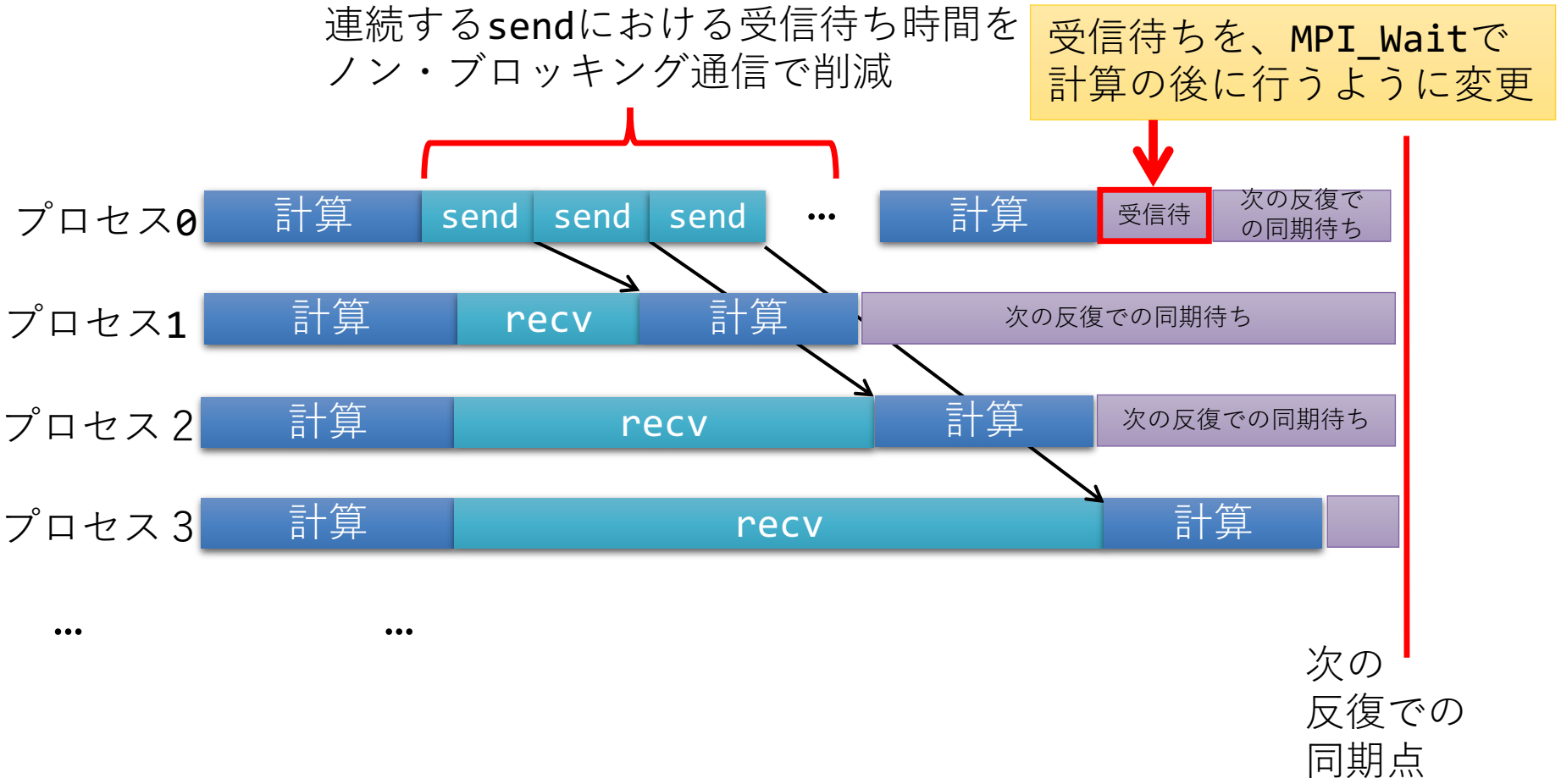
同期待ち関数

- `ierr = MPI_Wait(irequest, istatus);`

  - `irequest` : MPI_Request型（整数型配列）。
    送信を要求したメッセージにつけられた識別子。
  - `istatus` : MPI_Status型（整数型配列）。
    受信状況に関する情報が入る。
    - 要素数が`MPI_STATUS_SIZE`の整数配列を宣言して指定する。
    - 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。
  - 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が必要なデータを持っている場合



複数の同期待ち関数— MPI_Waitall

- `ierr = MPI_Waitall(icount, irequests, istatuses);`

  - `icount` : 同期待ちをする個数
  - `irequests[]` : MPI_Request型の配列（Fは整数型配列）。送信を要求したメッセージにつけられた識別子。
  - `istatuses[]` : MPI_Status型の配列（Fは整数型配列）。受信状況に関する情報が入る。

前ページの例で、MPI_Wait()をループするより MPI_Waitall()の方が効率が良い

注意点

- 以下のように解釈してください：
  - MPI_Send関数
    - 関数中にMPI_Wait関数が入っている；
  - 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戻る；

ノンブロッキング集団通信

- MPI-3.1で定義
- 集団通信のノンブロッキング版，MPI_Ixxx

分類	ブロッキング	ノンブロッキング	MPI_IN_PLACE	Intercomm
One-to-All	MPI_Bcast	MPI_Ibcast		✓
	MPI_Scatter{,v}	MPI_Iscatter{,v}	recvbuf@root	✓
All-to-One	MPI_Gather{,v}	MPI_Igather{,v}	sendbuf@root	✓
	MPI_Reduce	MPI_Ireduce	sendbuf@root	✓
All-to-All	MPI_Allgather{,v}	MPI_Iallgather{,v}	sendbuf	✓
	MPI_Alltoall{,v,w}	MPI_Ialltoall{,v,w}	sendbuf	✓
	MPI_Allreduce	MPI_Iallreduce	sendbuf	✓
	MPI_Reduce_scatter{,_block}	MPI_Ireduce_scatter{,_block}	sendbuf	✓
	MPI_Barrier	MPI_Ibarrier		✓
Others	MPI_Scan, Exscan	MPI_Iscan, Iexscan	sendbuf	

特別な変数

- 送受信で同じ配列を指定したい場合
  - collective通信で単に同じ配列をsendbuf, recvbufに指定するとエラーになる
    => 代わりにMPI_IN_PLACEを使う
    - 関数によって，sendbuf側かrecvbuf側に指定：前ページの表を参照

- 変数の指定を省略したい場合
  - MPI_STATUS_IGNORE
    - MPI_Recvなどで，MPI_Status の返り値が不要な場合
  - MPI_STATUSES_IGNORE
    - MPI_Waitallなどで MPI_Status []の返り値が不要な場合
  - MPI_ERRCODES_IGNORE
    - Fortranでierrの返り値が不要な場合

参考文献

1. Message Passing Interface Forum
   （ http://www.mpi-forum.org/ ）
2. MPI並列プログラミング、P.パチェコ 著 ／ 秋葉 博 訳
3. 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://accc.riken.jp/HPC/training/text.html ）
4. MPI-Jメーリングリスト
   （ http://phase.hpcc.jp/phase/mpi-j/ml/ ）
5. 講習会資料ページ (RIST)
   http://www.hpci-office.jp/pages/seminar_text

MPI実装と Wisteria/BDEC-01での利用方法

Open MPI, 富士通MPI

- LAM/MPIから発展
  - LAM: Ohio State Univ. => Univ. of Nortre Dam => Indiana Univ.
- オープンコミュニティー
- 最新版は 4.1.4
  - サポート継続　4.0.7
  - 3.X以前は非推奨
  - 次は 5.0 の予定
- NVIDIA独自にリリース➔NVIDIA HPC SDKに同梱 (22.5: 3.1.5, 4.0.5, 4.1.4 ベース)
- 富士通MPI: 1.2.35 (OMPI 4.0.1ベース)
- 参考情報
  - https://www.open-mpi.org/doc/
    - はっきり言ってこれだけではよくわからない
  - https://www.open-mpi.org/faq/
    - MCAオプションを指定することで機能を切り替える

Intel MPI

- Intel OneAPI HPC Toolkit (旧Parallel Studio XE cluster edition)にて，C, Fortranコンパイラと一緒に提供
  - MPICH, MVAPICH2をベースに開発されているが，独自機能も多数入っている
- 最新版は2022.3
- Wisteria/BDEC-01 Aquariusで利用可能なバージョン
  - 2021.2.0, 2022.1.2
  - GPUと一緒に使うにはお勧めしない
- 参考資料
  - Webポータル=>ドキュメント閲覧
  - https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mpi-for-linux/top.html
  - https://software.intel.com/en-us/mpi-developer-guide-linux
  - https://software.intel.com/en-us/mpi-developer-reference-linux

MVAPICH2

- オハイオ州立大で Prof. D.K. Pandaのグループが開発
  - Argonne National Lab.等で開発されているMPICHがベース (MPICH4)
- （主に）InfiniBand向け，最先端機能をいち早く実装
- 現時点での最新版： 2.3.7
- Wisteria/BDEC-01 Aquariusで利用可能なバージョン:
  - 未インストール (Open MPI推奨のため)
  - ビルドすれば動くはずだが未確認
    - 昔はMVAPICH2の方が明らかに高速だったが、今は差がないか、OpenMPIの方が速いことも多い（はず）
- 参考情報
  - http://mvapich.cse.ohio-state.edu
  - http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.3.7-userguide.html

参考：moduleコマンドの使い方

- 様々なコンパイラ，MPI環境などを切り替えるためのコマンド
- パスや環境変数など必要な設定が自動的に変更される
- ジョブ実行時にもコンパイル時と同じmoduleをloadすること

- 使用可能なモジュールの一覧を表示：module avail
- 使用中のモジュールを確認：module list
- モジュールのload: module load モジュール名
- モジュールのunload: module unload モジュール
- モジュールの切り替え：module switch 旧モジュール 新モジュール
- モジュールを全てクリア: module purge

Wisteria/BDEC-01で使用可能なMPI実装

- W-Odyssey
  - 富士通 MPI
- W-Aquarius
  - Open MPI
    - CUDA向け各種
  - NVIDIA版Open MPI
  - NVIDIA HPC-X
  - Intel MPI
  - (MVAPICH2自分でコンパイル)
- 参考

  $ show_module

module load odyssey

1.2.35		module load fj/1.2.35 fjmpi/1.2.35
Open MPI	4.1.1	module load gcc/8.3.1 ompi/4.1.1
		module load intel/2021.2.0 ompi/4.1.1
		module load nvidia/{21.3,22.2,22.5} ompi/4.1.1
Open MPI +CUDA	4.1.1 (一部 4.1.4)	module load gcc/8.3.1 cuda/{10.1,10.2,11.0,11.1,11.2,11.3,11.4} ompi-cuda/4.1.1-{…,11.4}
		module load intel/2021.2.0 cuda/11.2 ompi-cuda/4.1.1-11.2
		module load nvidia/{21.3,22.2,22.5} cuda{…,11.4} ompi-cuda/{4.1.1-11.2,4.1.4-11.4}
Intel MPI	2021.2.0, 2021.5.1	module load gcc/8.3.1 impi/2021.{2.0,5.1}
		module load intel/{2021.2.0,2022.1.2} impi/2021.{2.0,5.1}
NVIDIA MPI (OpenACC)	21.3,22.2,22.5 (3.1.5)	module load nvidia/{21.3,22.2,22.5} nvmpi/{21.3,22.2,22.5}
HPC-X	2.11	module use /work/opt/local/x86_64/cores/nvidia/22.5/Linux_x86_64/22.5/comm_libs/hpcx/hpcx-2.11/modulefiles module load hpcx nvidia

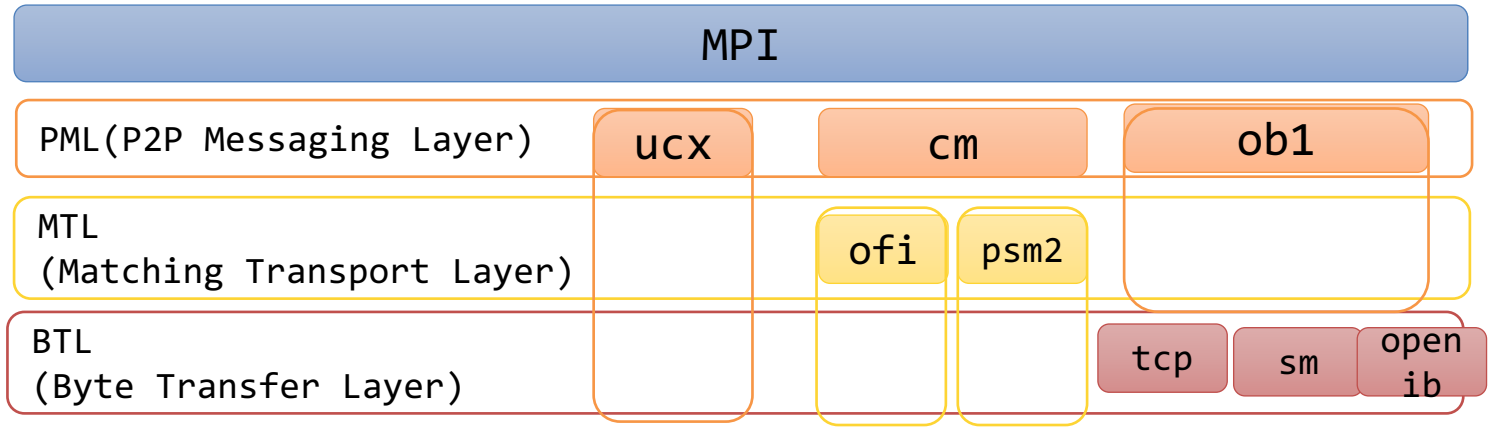
パラメータの最適化

- MPI実装のデフォルトパラメータが最適とは限らない
  - 同じハードウェアで開発・調整しているわけではない

プロトコルスタック選択（Aquarius）

- Open MPIは歴史的に様々なスタックを使ってきた
  - InfiniBand向けだけで
    - openib (btl), (yalla (pml)), ofi(mtl), ucx(pml)
  - 現在はUCXがデフォルト
    - 軽量、高速
  - 他はUCXに問題があるときなどに選ぶ程度
- 参考：

```
$ ompi_info -all
```



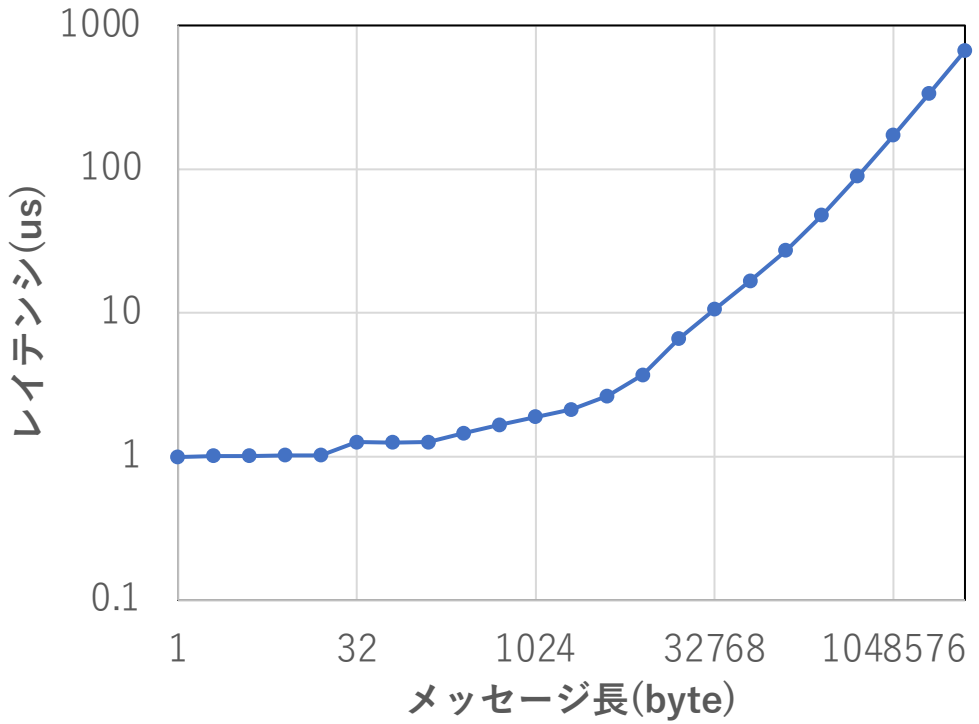
EagerとRendezvous

- Eagerプロトコル
  - 送信側はとにかく送信する
  - 受信側は
    - マッチするものがあれば受信処理
    - マッチしなければ，受信バッファに保持 => バッファへのコピーが発生
  - メッセージサイズ小のときに適している
- Rendezvousプロトコル
  - 送受信の前に，受信相手がいるかどうかを確認してから送る
    => ゼロコピーで通信が可能
  - メッセージサイズ大のときに適している

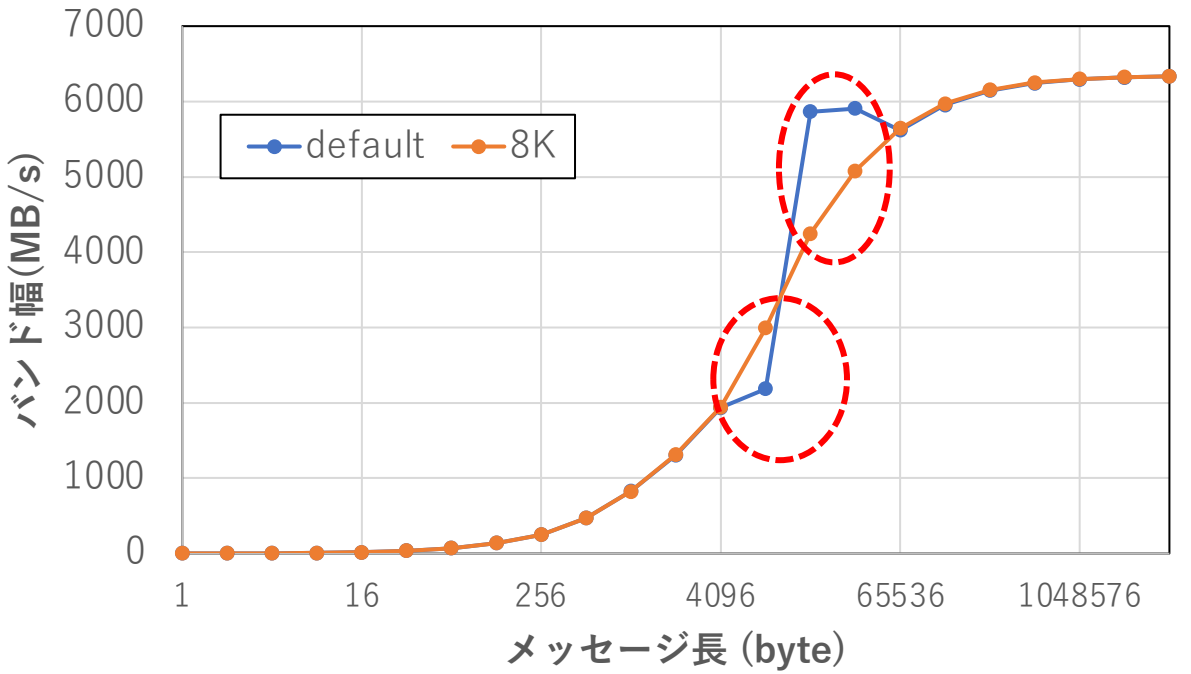
- 通常，メッセージサイズに応じて自動的に切り替わるが指定も可能
  - 富士通 MPI:
    export OMPI_MCA_btl_tofu_eager_limit=メッセージ長
  - Open MPI(UCX):
    export UCX_RNDV_THRESH=メッセージ長
  - Intel MPI:
    export I_MPI_EAGER_THRESHOLD=メッセージ長
  - MVAPICH2:
    export MV2_IBA_EAGER_THRESHOLD=メッセージ長

MPI性能 (Odyssey, pt2pt)

- レイテンシ：osu_latency
  - 最小 **0.9us**

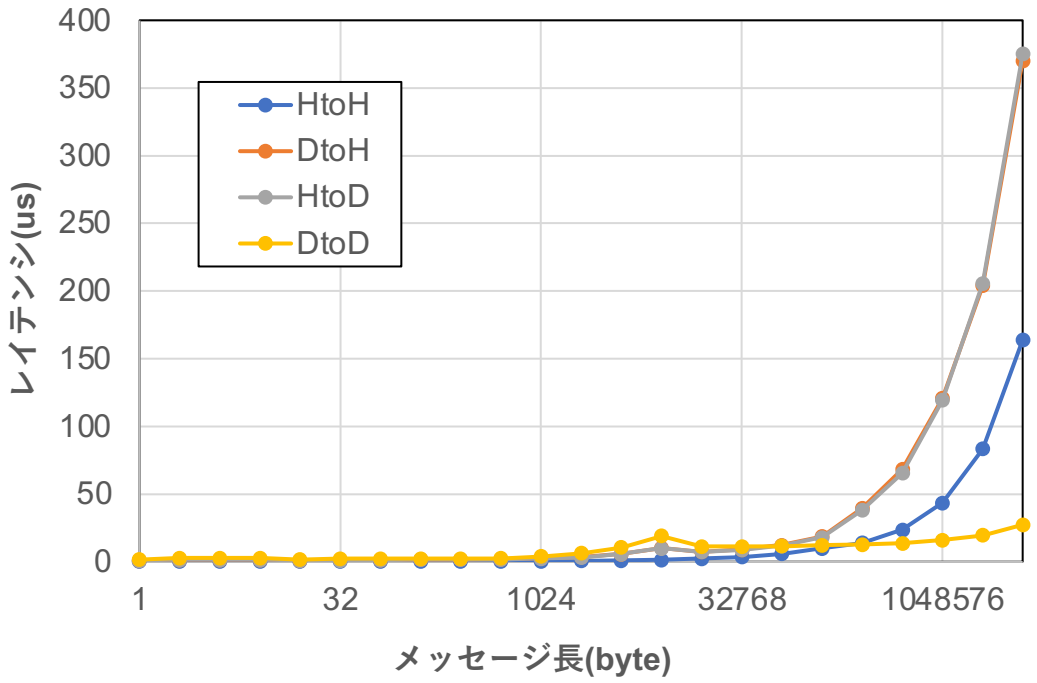


- バンド幅: osu_bw
  - 最大 **6.3 GB/s**
  - 8Kを底上げ➔Eager limit=8K
    - 2.2G➔3.3G@8K / 5.9G➔4.2G@16K

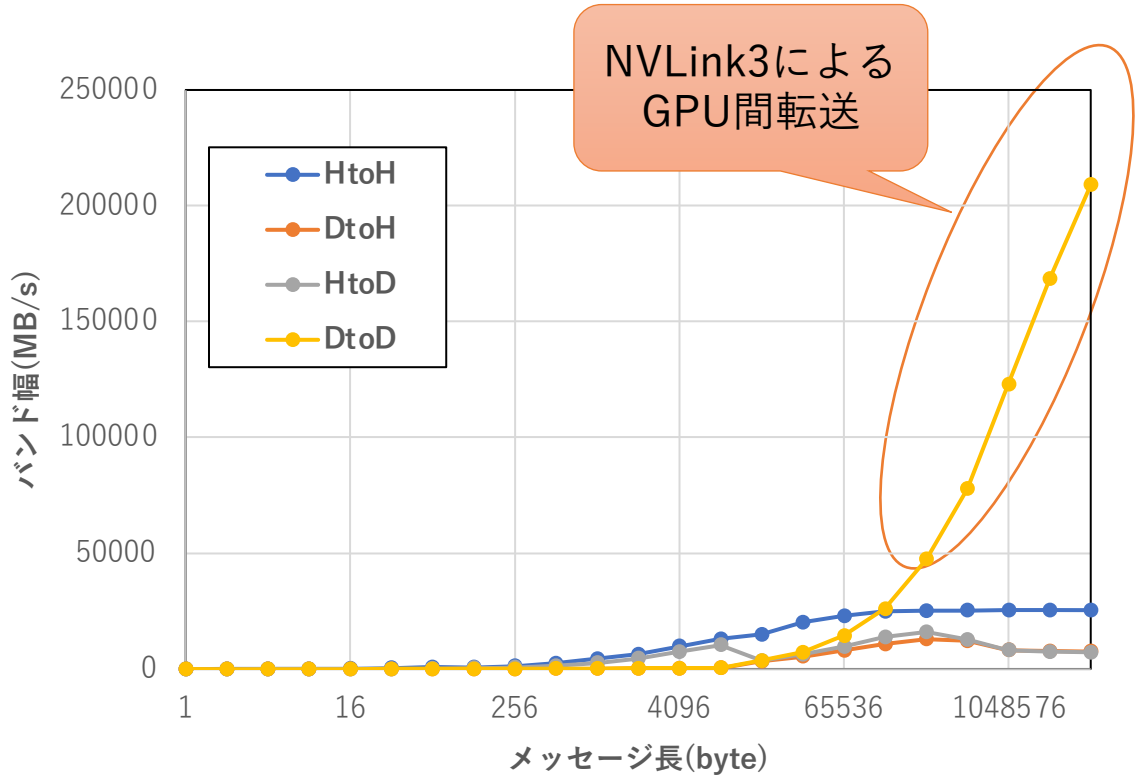


MPI性能 (Aquarius, OpenMPI, ノード内)

- osu_latency
  - DtoH, HtoDはほぼ同じ
  - 512KB以上は DtoDが圧倒的速さ

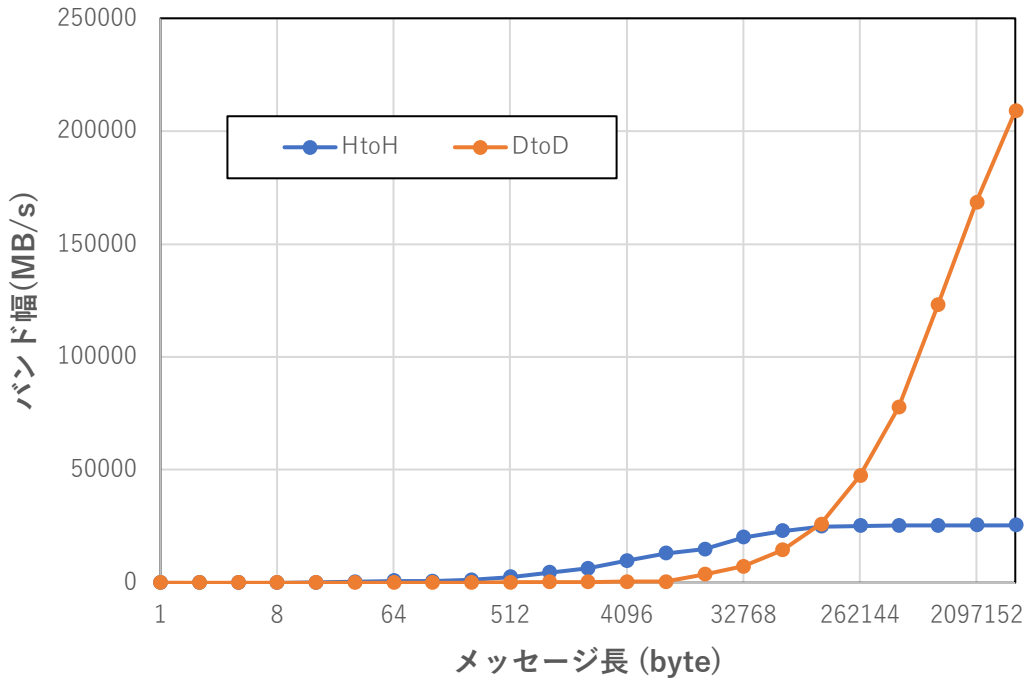


- osu_bw
  - DtoH, HtoDはほぼ同じ

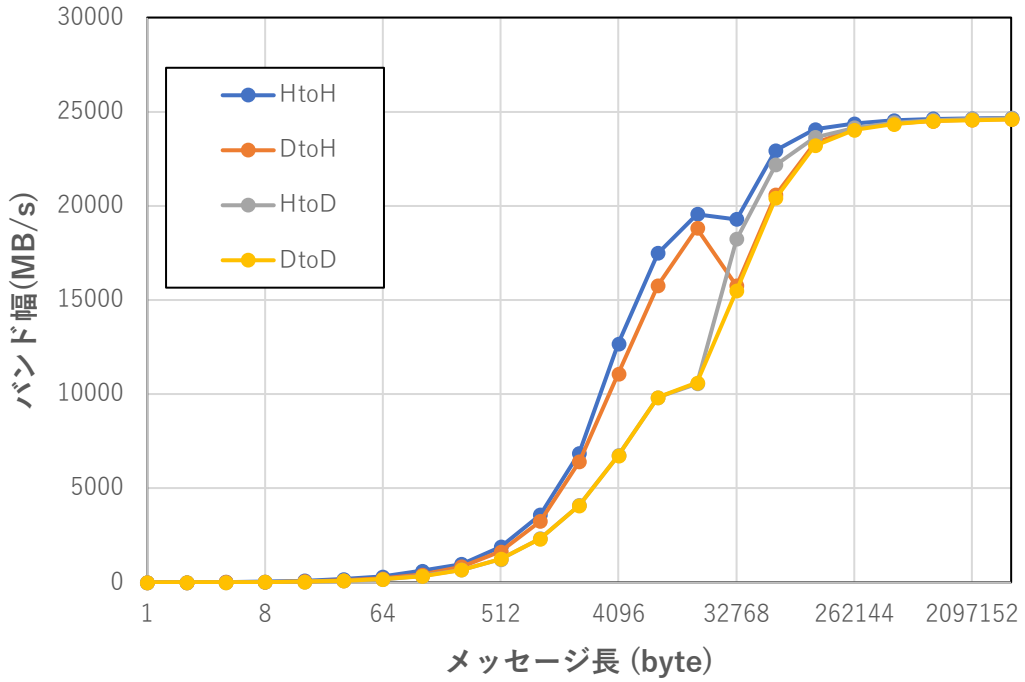


参考：MPI性能（Aquarius, OpenMPI, ノード間）

- ノード内バンド幅: osu_bw
  - Aquarius: DtoD 最大 **209.0GB/s**
    - NVLink3 300GB/s
  - HtoH 最大 **24.5GB/s**



- ノード間バンド幅: osu_bw
  - **1リンク**
  - Aquarius: 最大 **24.6 GB/s**
- 最小レイテンシ **1.58us (HtoH), 2.27us (DtoD)**
- UCX: UCX_RNDV_THRESH=32K



アルゴリズムの選択 (富士通 MPI)

- 集団通信には複数のアルゴリズムが実装されている
  - デフォルトでは、ランク数やメッセージサイズに合わせて動的に変化
- 例：MPI_Allreduce (Tofu専用)
  1. basic_linear
  2. nonoverlapping
  3. recursive_doubling
  4. ring
  5. segmented_ring
  6. rdbc
  7. trinaryx3 (trix3)
  8. trinaryx6 (trix6)
- export OMPI_MCA_coll_select_allreduce_algorithm=rdbc
  - rdbcアルゴリズムを強制
- 詳細は「MPI使用手引書」8.4 アルゴリズム選択

アルゴリズムの選択 (Intel MPI)

- 集団通信には複数のアルゴリズムが実装されている
- 例：MPI_Allreduce
  1. Recursive doubling
  2. Rabenseifner’s
  3. Reduce + Bcast
  4. Topology aware Reduce + Bcast
  5. Binomial gather + scatter
  6. Topology aware binominal gather + scatter
  7. Shumilin's ring
  8. Ring
  9. Knomial
  10. Topology aware SHM-based flat
  11. Topology aware SHM-based Knomial
  12. Topology aware SHM-based Knary
- export I_MPI_ADJUST_ALLREDUCE=8:1024-4096@16-32;…
  - 1024~4096 Byteかつ16~32ノードのときRingアルゴリズムを使う

演習

- 各種MPIライブラリ実装の性能を確認してみよう

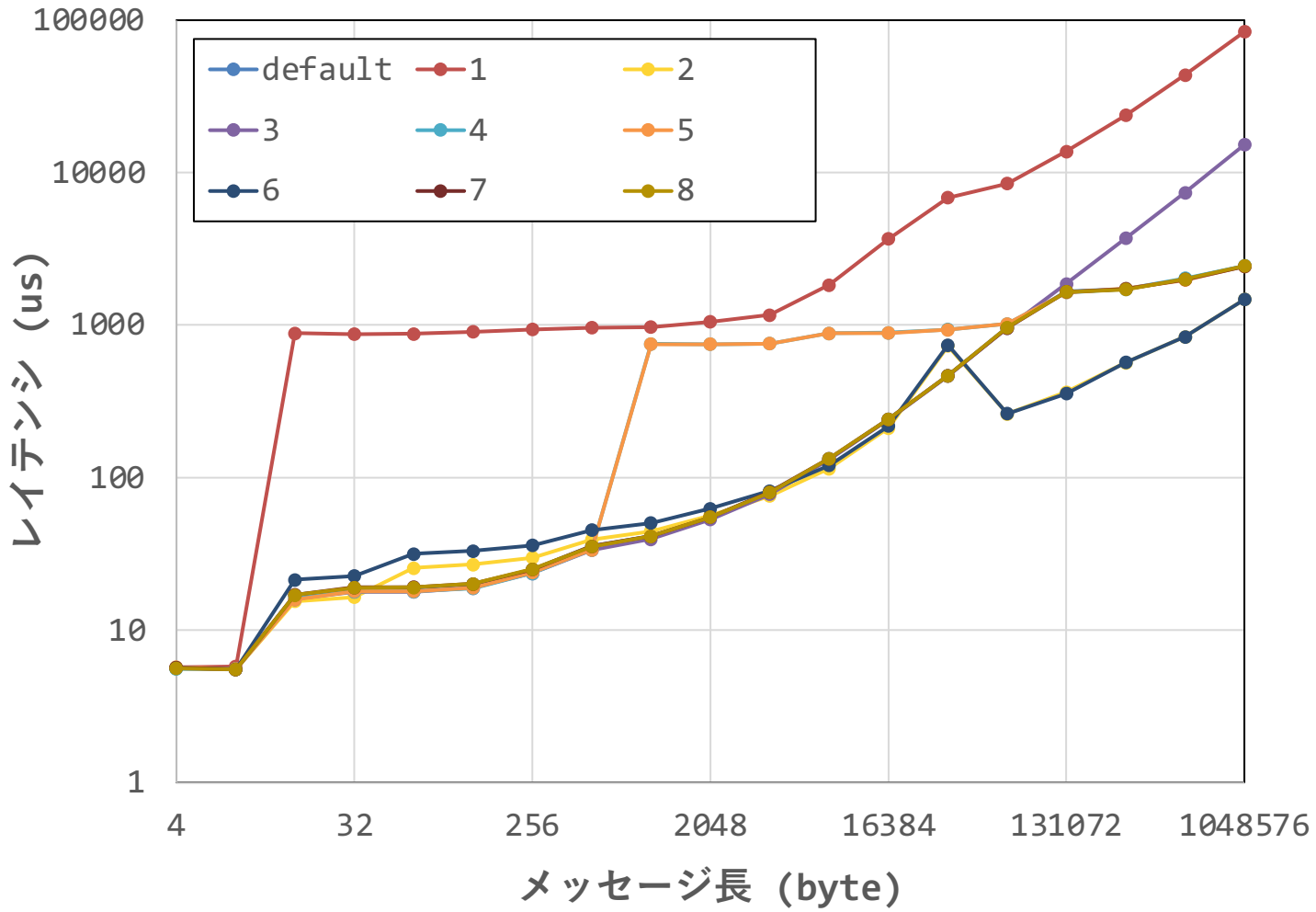
```
$ cd MPIbench
$ ./make-omb-all.sh
$ cd coll
$ pjsub omb-coll-fjmpi-allreduce.sh
```

(他のバージョンのものも適宜お試しください)

```
$ ./pick-allreduce.sh omb-coll-fjmpi-allreduce.sh.XXXX.out
```

Allreduceのアルゴリズムによる性能差 (Odyssey)

- Odyssey 8ノード
- 256プロセス
- 富士通MPI 1.2.32
- サイズによって最適アルゴリズムが変わる
  - defaultも悪くはないが、よりいいパターンがありそう

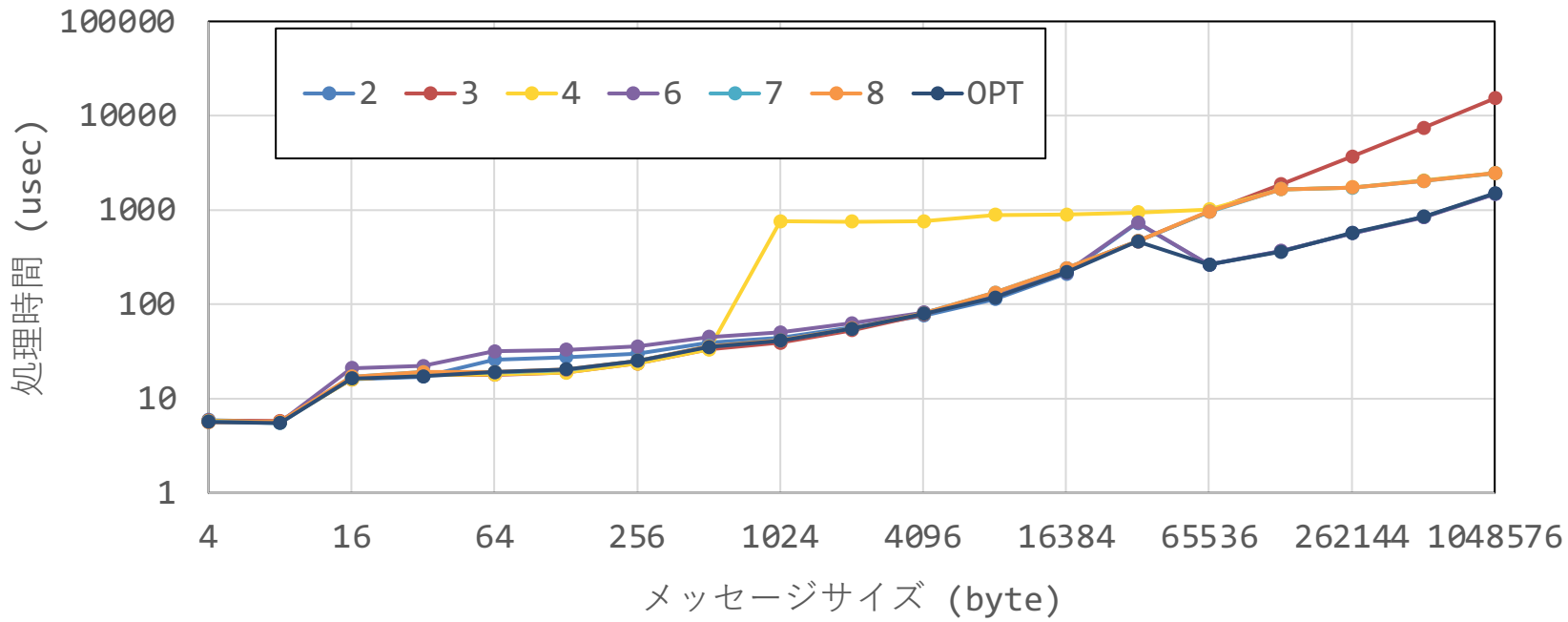


Allreduceの性能改善 (8ノードx32, 富士通MPI 1.2.32の場合)

- メッセージ長ごとに最適なアルゴリズムを組み合わせ、設定ファイル作成

  ```
  export OMPI_MCA_coll_select_dectree_file=./wo-allreduce-opt.conf
  ```

- MPI使用手引書J2UL-2480-02Z0(02) 「8.3.1.5 外部入力ファイルによるアルゴリズム選択」参照



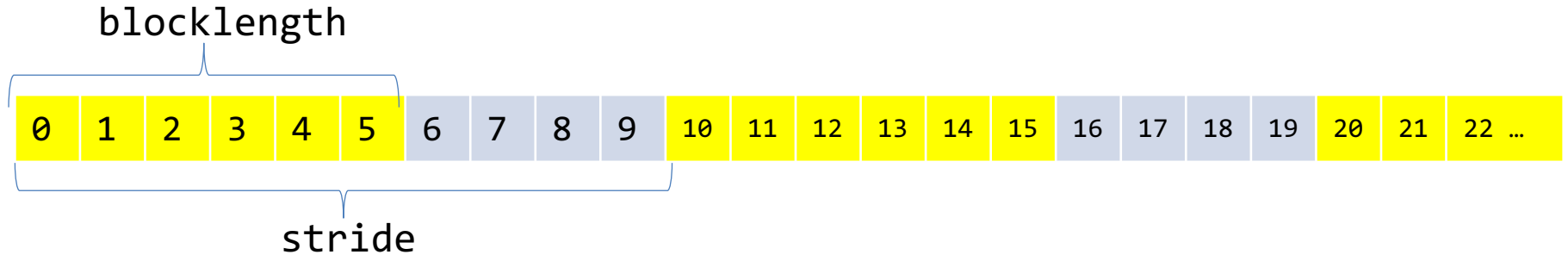
派生データ型：
新しいデータ型の定義

ブロックストライド転送（同一型，規則的）

- `ierr = MPI_Type_vector(count, blocklength, stride, oldtype, newtype)`

  - `int count (IN):` 繰り返し回数（strideの）
  - `int blocklength (IN):` ブロック数（連続データの個数）
  - `int stride (IN):` ブロック開始の間隔
  - `MPI_Type oldtype (IN):` 元のデータ型
  - `MPI_Type * newtype (OUT):` 新しいデータ型

- 利用する前に　`MPI_Type_commit`を忘れずに!!



同一型，不規則なデータ配置

```
• ierr = MPI_Type_indexed(count, array_of_blocklengths,
                          array_of_displacements, oldtype, newtype);
```

- int count (IN): ブロックの数
- int array_of_blocklengths[] (IN): 各ブロックの要素数
- int array_of_displacements[] (IN): 次のブロックまでの間隔
- MPI_Datatype oldtype (IN): 元のデータ型
- MPI_Datatype *newtype (OUT): 新しいデータ型

同一型，不規則なデータ配置(2)

```
• ierr = MPI_Type_create_hindexed(count, array_of_blocklengths,
                                  array_of_displacements, oldtype, newtype);
```

- int count (IN): ブロックの数
- int array_of_blocklengths[] (IN): 各ブロックの要素数
- MPI_Aint array_of_displacements[] (IN): 次のブロックまでの間隔（アドレス値）
- MPI_Datatype oldtype (IN): 元のデータ型
- MPI_Datatype *newtype (OUT): 新しいデータ型

- MPI_Type_indexedとはarray_of_displacementのみが異なる
- MPI-2.0以前はMPI_Type_hindexedだった

異なる型，不規則なデータ配置

```
• ierr = MPI_Type_create_struct(count, array_of_blocklengths,
       array_of_displacements, array_of_types, newtype);
```

- int count (IN): メンバの数
- int array_of_blocklengths[] (IN): 各メンバの個数
- MPI_Aint array_of_displacements[] (IN):次のメンバまでの間隔（アドレス値）
- MPI_Datatype array_of_types[] (IN): 各メンバのデータ型
- MPI_Datatype *newtype (OUT): 新しいデータ型

- MPI-2.0以前はMPI_Type_structだった

ファイルシステムと MPI-IO

Wisteria/BDEC-01で利用可能なファイルシステム

PATH	種類	備考
/home/ログイン名	NFS	ログインノードからのみ利用可能 容量が小さい ログインに必要なもの・各種設定ファイルなど、最低限のものだけ置くこと
/work/グループ名/ログイン名	並列 (FEFS)	ログインノードからも計算ノードからも利用可能，一般的な用途に使える バッチジョブの投入はここから行う
/data/{perm,scratch}/グループ名/ログイン名	高速 (FEFS)	ログインノードからも計算ノードからも利用可能，一般的な用途に使える scratchは一定期間で削除
/tmp	Ramdisk	非常に容量が小さい 使用を推奨しない
/dev/shm		

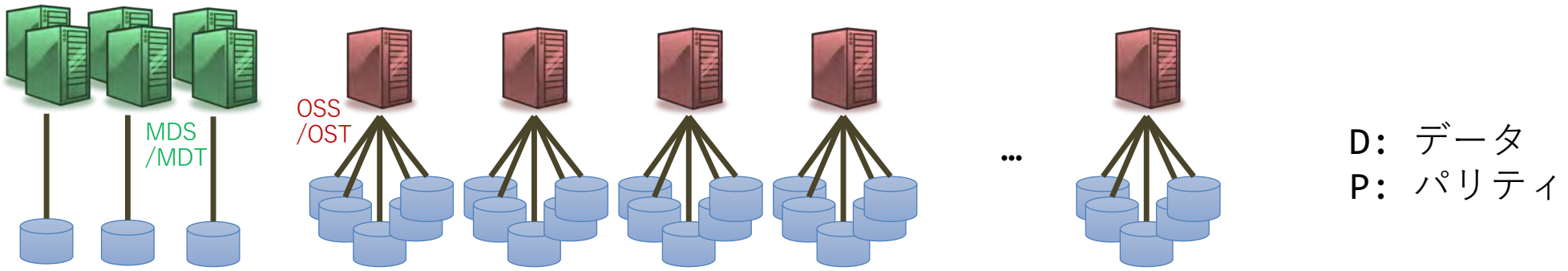
Wisteria/BDEC-01の並列ファイルシステム

- 富士通FEFS (Fujitsu Exabyte File System)
  - 大規模ファイル入出力、メタデータ操作の両方で高性能なファイルシステム
  - データの分散方法をファイルごとに指定可能(後述)
  - Lustreファイルシステムの上位互換, Oakbridge-CX等
  - 「富岳」も同じファイルシステム

FEFSの物理構成とデータ配置

- メタデータを格納するMDS/MDT (Meta Data Server/Target)
  - ファイル管理情報（日付、サイズ等）、OSS/OSTへのデータ格納情報
- データを格納するOSS/OST (Object Storage Server/Target)
  - ファイルデータそのもの
- Wisteria/BDEC-01の構成
  - MDT：5 領域(/work), 2領域(/data)　(DNE: Distributed Namespace)
  - OSS：「174 (RAID6(8D2P)相当) + 2 スペア」　を16セット => 26 PB (/work)
    「22 (RAID6(8D2P)相当) + 1 スペア」　を16セット => 1 PB (/data)

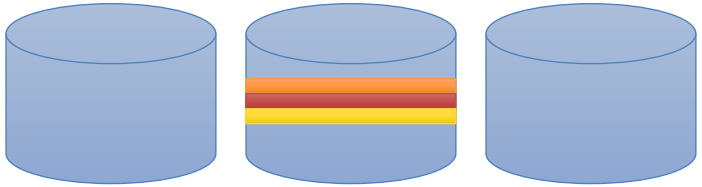
  いずれも 128 OSTで管理



FEFSのデータ配置の指定

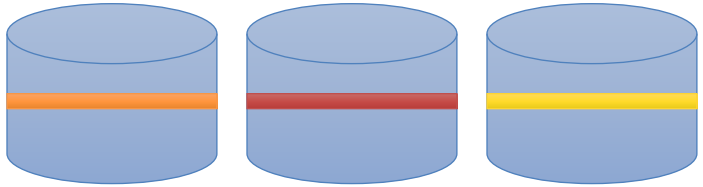
- データ配置の指定
  - １ファイルのデータをひとつのOSTに配置するか、複数のOSTに分散して配置するかはユーザが指定できる
    - デフォルトでは１ファイルあたりひとつのOSTに配置、ファイル単位で使用するOSTが決められる
    - `lfs getstripe / lfs setstripe`コマンドで参照・変更可能
  - 複数プロセスから単一のファイルに対する処理を高速化したい場合には複数のOSTを使うような指定が必要

ひとつのOSTに配置



どんなにがんばっても最大で10ST分の
読み書き性能しか得られない

複数のOSTに配置



複数OST分の読み書き性能が得られる
可能性がある

FEFSのデータ配置の指定の方法

- `lfs setstripe [OPTIONS] <ディレクトリ名|ファイル名>`
  - 主なオプション：`-s size -c count`
    - ストライプサイズ `size` 毎に`count` 個のOSTに渡ってデータを分散配置する、という設定にした空のファイルを作成する
    - 既存ディレクトリに対して行うとその後に作るファイルに適用される
    - countに`-1`を指定すると全OSTを使用
  - 使用例

```
$ rm /path/to/data.dat
$ lfs setstripe -s 1M -c 50 /path/to/data.dat
$ dd if=/dev/zero of=/path/to/data.dat bs=1M count=4096
```

※ファイルを作成する際に指定する必要があるため、最初に削除している

- `lfs getstripe <ディレクトリ名|ファイル名>`
  - 設定情報を確認する
- `lfs df`
  - MDT/OST構成情報を確認する

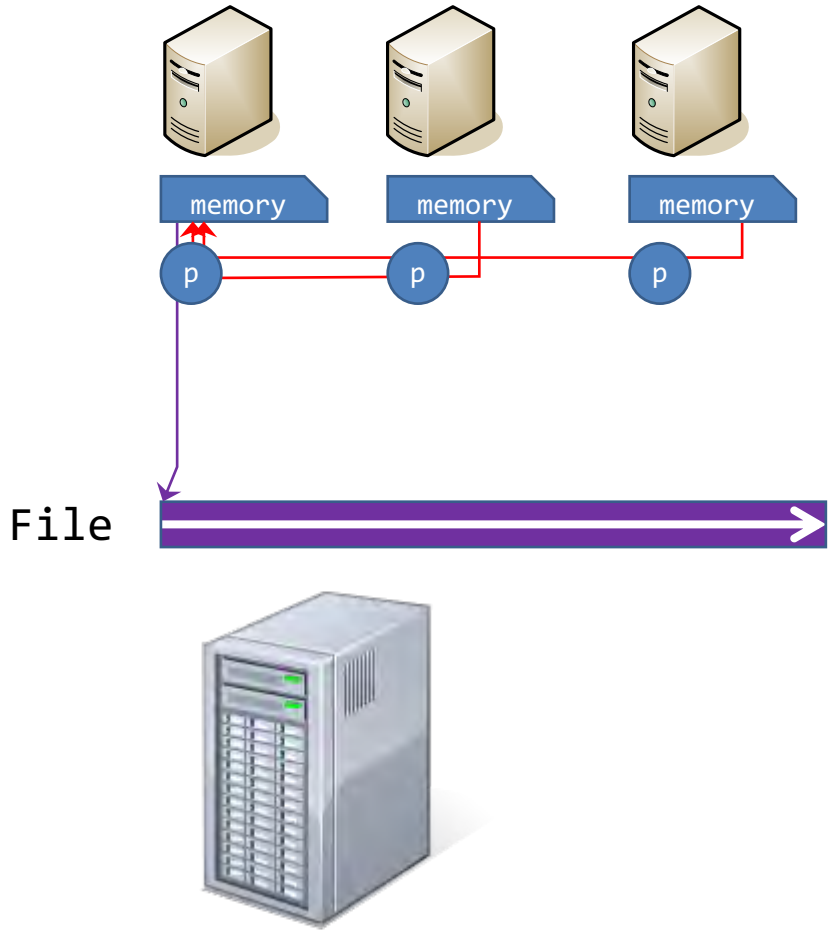
MPI-IO とは

- 並列ファイルシステムをMPIの枠組みで効率的に利用するための仕組み
  - ある程度抽象化を持たせた記述をすることで、（利用者が気にすることなく）最適な実装が利用可能になる（ことが期待される）
  - 例：利用者はファイル上のデータが配列上のどこに配置されて欲しいかだけを指定
    → MPI-IOにより「まとめて読み込んでMPI通信で分配配置」されて高速に処理が行われる、かも知れない

- 以後 API は C言語での宣言や利用例を説明するが、Fortranでも同名の関数が利用できる
  - 具体的な引数の違いなどはリファレンスを参照のこと
  - C++宣言はもう使われていない、C宣言を利用する

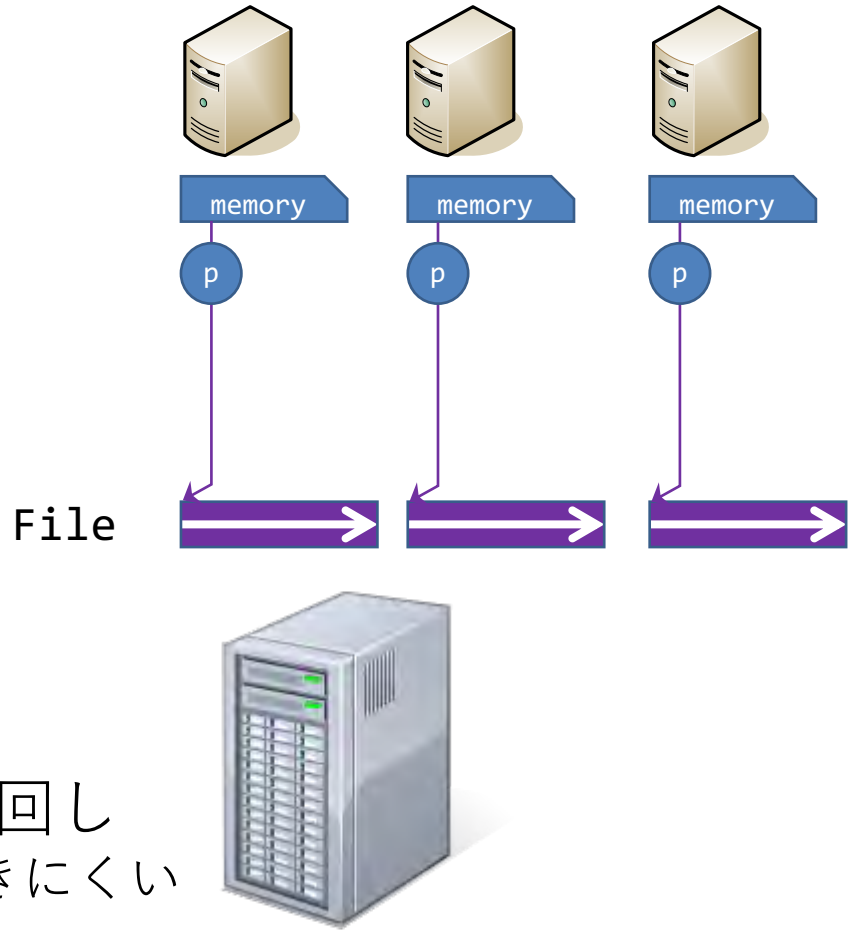
非MPI-IO：並列アプリによるファイルI/Oの例1

- 逐次入出力
  - 1プロセスのみがI/Oを行う、(MPI)通信によりデータを分散・集約する
    - MPI_Gather + fwrite
    - fread + MPI_Scatter
  - 利点
    - 単一ファイルによる優秀な取り回し
    - 読み書き操作回数の削減
  - 欠点
    - スケーラビリティの制限
    - 並列入出力をサポートしたファイルシステムの性能を活かせない



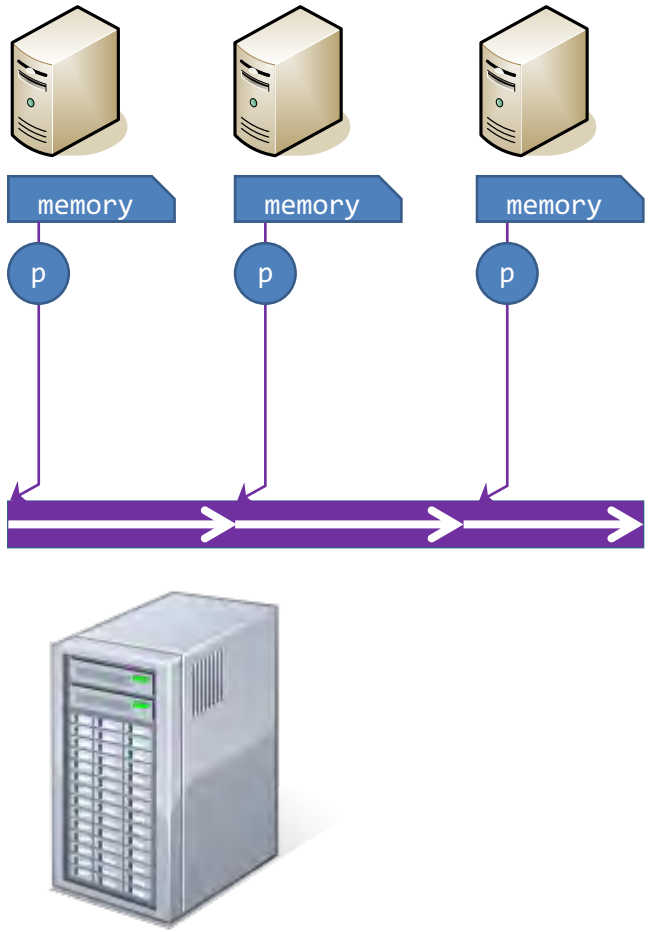
非MPI-IO：並列アプリによるファイルI/Oの例2

- 並列入出力
  - 各プロセスが個別にI/Oを行う
  - 利点
    - ファイルシステムの並列入出力サポートを生かせる
    - スケーラビリティ向上
  - 欠点
    - 入出力回数の増大
      - 多数の小ファイルアクセス
    - 複数ファイルによる劣悪な取り回し
      - プログラム（ソース）も読み/書きにくい



MPI-IOによる並列アプリ上ファイルI/Oの一例

- 単一ファイルに対する並列入出力
  - スケーラビリティ向上
  - プログラム（ソース）もわかりやすい



想定するシナリオ

- MPI-IO関数を使うことで、複数プロセスによる単一ファイルへの並列入出力を簡単に行うことを考える
  - 今回は利用方法の習得と利便性の理解を目的とし、入出力性能についてはこだわらない
  - 単一ファイルに対する操作のため、入出力の性能も上げたい場合には`lfs setstripe`の設定も必要

MPI-IOによる並列ファイル入出力

- 基本的な処理の流れ

1. MPI_File_open 関数によりファイルをオープン
2. 読み書きの処理を実行（様々な関数が用意されている）
3. MPI_File_close 関数によりファイルをクローズ

- ファイルのオープンとクローズ

```
int MPI_File_open(
  MPI_Comm comm,   // コミュニケータ
  char *filename,  // 操作ファイル名
  int amode,       // アクセスモード（読み書き、作成など）
  MPI_Info info,   // 実装へのユーザからのヒント
  MPI_File *fh     // ファイルハンドラ
)
```

```
int MPI_File_close(
  MPI_File *fh     // ファイルハンドラ
)
```

読み書きする場所・パターンの指定

- いくつかの指定方法がある
  - `_at` 系の`read/write`関数でその都度指定する
  - `MPI_File_seek` であらかじめ指定しておく

```
int MPI_File_seek(
  MPI_File fh,         // ファイルハンドラ
  MPI_Offset offset,   // オフセット（バイト数）
  int whence           // 指定方法のバリエーション ※
)
```

※即値、現在位置からのオフセット、ファイル末尾からのオフセット

  - `MPI_File_set_view` であらかじめ指定しておく

```
int MPI_File_set_view(
  MPI_File fh,         // ファイルハンドラ
  MPI_Offset disp,     // オフセット（バイト数）
  MPI_Datatype dtype,  // 要素データ型
  MPI_Datatype filetype, // ファイル型 ※
  char* datarep,       // データ表現形式
  MPI_Info info        // 実装へのユーザからのヒント
)
```

※要素データ型と同じ基本型または要素データ型で構成される派生型

読み書き方法のバリエーション

- 「view」を用いた書き込みだけでも様々なバリエーションが存在
  - ブロッキング・非集団出力
    - MPI_File_write
  - 非ブロッキング・非集団出力
    - MPI_File_iwrite / MPI_Wait
  - ブロッキング・集団出力
    - MPI_File_write_all
  - 非ブロッキング・集団出力
    - MPI_File_write_all_begin / MPI_File_write_all_end
- 用途に合わせて使い分ける
  - 非ブロッキング：読み書きが終わらなくても次の処理を実行できる
  - 集団出力：同一コミュニケータの全プロセスが行わねばならない、まとめて行われるため読み書き処理の時間自体は高速

並列出力の例：MPI_File_set_view/writeの場合

- プロセス番号（MPIランク）に基づいて1つずつ整数を書き出すだけの単純な例

```
MPI_File mfh;
MPI_Status st;
int disp;
int data;
const char filename[] = "data1.dat";

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

disp = sizeof(int)*rank; // 各プロセスが書き込む場所を設定
data = 1+rank; // 書き込みたいデータを設定（MPIランク+1）

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDWR|MPI_MODE_CREATE, MPI_INFO_NULL, &mfh);

MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write(mfh, &data, 1, MPI_INTEGER, &st);
MPI_File_close(&mfh);

MPI_Finalize();
```

出力結果の例（OFP 16プロセス）

```
$hexdump data1.dat
0000000 0001 0000 0002 0000 0003 0000 0004 0000
0000010 0005 0000 0006 0000 0007 0000 0008 0000
0000020 0009 0000 000a 0000 000b 0000 000c 0000
0000030 000d 0000 000e 0000 000f 0000 0010 0000
0000040
```

利点
- データの集約処理は不要
- 書き込み結果が１ファイルにまとまる

並列出力の例：その他の関数の場合

- MPI_File_iwrite / MPI_Wait

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_iwrite(mfh, &data, 1, MPI_INTEGER, &req);
MPI_Wait(req, &st);
```

- MPI_File_write_all

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write_all(mfh, &data, 1, MPI_INTEGER, &st);
```

- MPI_File_write_all_begin / MPI_File_write_all_end

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write_all_begin(mfh, &data, 1, MPI_INTEGER);
MPI_File_write_all_end(mfh, &data, &st);
```

- MPI_File_write_at

```
MPI_File_write_at(mfh, disp, &data, 1, MPI_INTEGER, &st);
```

いずれも前ページのMPI_File_openと
MPI_File_closeの間を置き換えて使う

互換性・可搬性について

HDF5やnetCDFといった
データ管理ライブラリの利用もお勧め

- MPI_File_set_viewのdatarep引数によるデータ表現形式によって可搬性を向上させることができる
  - “native”: メモリ上と同じ姿での表現（何も変換しない）
  - “internal”: 同じMPI実装を利用するとき齟齬がない程度の変換
  - “external32”: MPIを利用する限り齟齬がないように変換
  - その他のオプションはMPI仕様書を参照
- 「View」を使わない入出力処理は可搬性が保証されない
  - MPI_File_open → MPI_File_write_at → MPI_File_close は記述量が少なく簡単だが、可搬性が低い
- 具体的な例
  - Oakforest-PACSと京コンピュータで同じプログラム（ソースコード）を使いたい
    - nativeでは京コンピュータとOakforest-PACSの出力結果が一致しない
    - external32では出力結果が一致する（京コンピュータの結果に揃う）
  - Oakforest-PACS同士でもMPI処理系を変更すると結果が変わるかも知れない

共有ファイルポインタ

- ファイルポインタを共有することもできる
  - MPI_File_read/write_sharedで共有ファイルポインタを用いて入出力
  - 読み書き動作が他プロセスの読み書きにも影響する、「カーソル」の位置が共有される
  - 複数プロセスで順番に（到着順に）１ファイルを読み書きするような際に使う
    - 例：ログファイルへの追記
- 共有ファイルポインタを用いた集団入出力もある
  - MPI_File_read/write_ordered
  - 順序が保証される
    - ランク順に処理される
    - 並列ではない

```
#define COUNT  2
MPI_File fh;
MPI_Status st;
int buf[COUNT];
MPI_File_open
  (MPI_COMM_WORLD, “datafile”,
  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_read_shared
  (fh, buf, COUNT, MPI_INT, &st);
MPI_File_close(&fh);
```

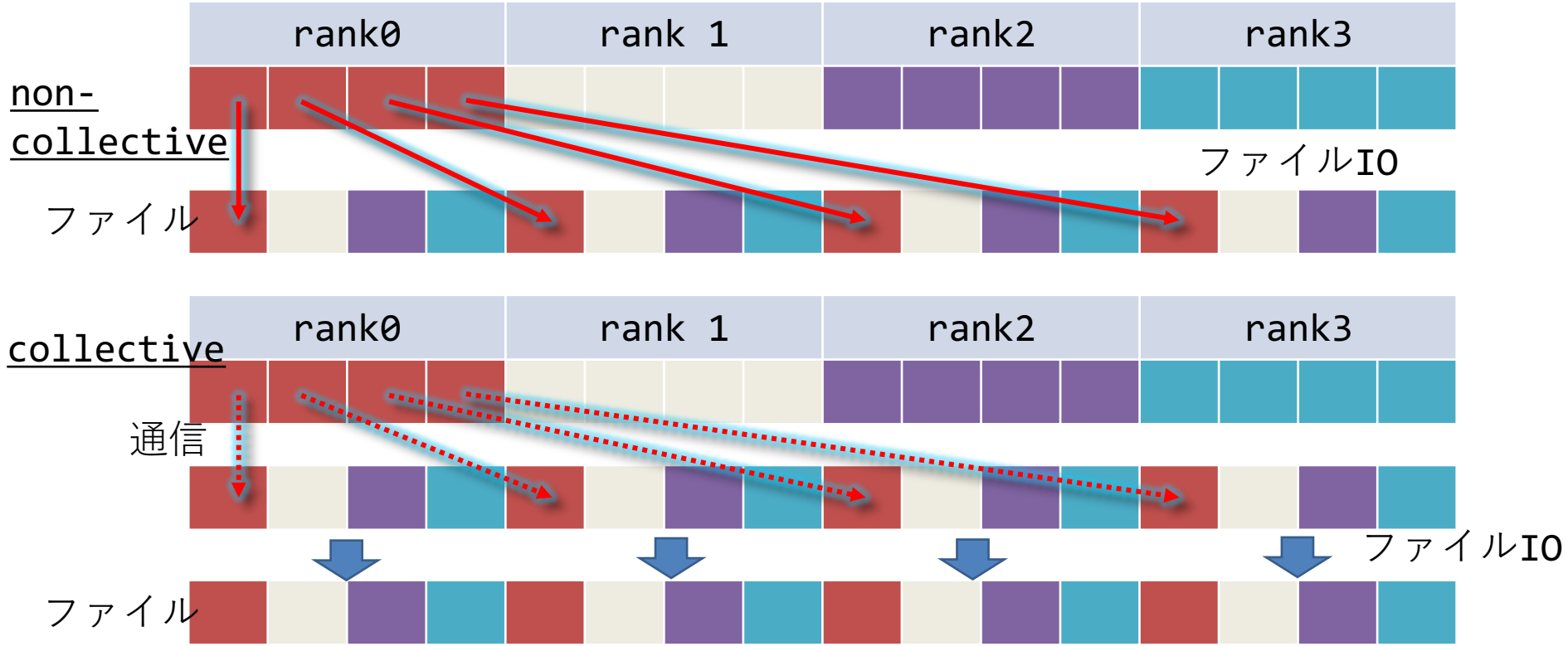
ファイル入出力関数のまとめ

位置	同期	non-collective	collective
明示的オフセット	blocking	MPI_FILE_{READ,WRITE}_AT	MPI_FILE_{READ,WRITE}_AT_ALL
	nonblocking	MPI_FILE_I{READ,WRITE}_AT	MPI_FILE_I{READ,WRITE}_AT_ALL
	split collective	N/A	MPI_FILE_{READ,WRITE}_AT_ALL_{BEGIN,END}
個別ファイルポインタ	blocking	MPI_FILE_{READ,WRITE}	MPI_FILE_{READ,WRITE}_ALL
	nonblocking	MPI_FILE_I{READ,WRITE}	MPI_FILE_I{READ,WRITE}_ALL
	split collective	N/A	MPI_FILE_{READ,WRITE}_ALL_{BEGIN,END}
共有ファイルポインタ	blocking	MPI_FILE_{READ,WRITE}_SHARED	MPI_FILE_{READ,WRITE}_ORDERED
	nonblocking	MPI_FILE_I{READ,WRITE}_SHARED	N/A
	split collective	N/A	MPI_FILE_{READE,WRITE}_ORDERED_{BEGIN,END}

MPI_FILE_WRITE_SHARED はサポートしないシステムもある（他にも共有ポインタは要確認）

Collective IOの利点

- MPIプロセス数が増えると細かなIOが増加
  =>Collective IOによってIOをまとめる効果



演習

- 実際のファイルIOを測定してみよう

- MPIIO
  - File Per Process (FPP): mpiio.c
  - Single Shared File (SSF)
    - block書き出し: mpiio-single.c
      - ファイルをストライピングしているとどう変わるか
    - stripe or cyclic書き出し: mpiio-single-stripe.c
      - collective IOを使うと改善される
  - /workと/data/scratchで比較

コミュニケータ
グループ
トポロジ

コミュニケータ

- MPIにおいて，操作の対象になるMPIプロセスの集合を表すもの
- 定義済みのコミュニケータ
  - **MPI_COMM_WORLD** ： MPI実行中の全プロセス
  - **MPI_COMM_SELF**: 自分自身のプロセス
- コミュニケータの種類
  - **イントラコミュニケータ (Intra-communicator)**
    - お互いに通信が可能，集団通信操作が可能
    - 通常使うのはこちら
  - インターコミュニケータ (Inter-communicator)
    - 異なるイントラコミュニケータに属するプロセス間での通信に用いる

コミュニケータの作成・開放

1. 既存のコミュニケータを利用して作る
   - MPI_Comm_dup, MPI_Comm_idup : 既存のコミュニケータを複製
   - MPI_Comm_split : 既存のコミュニケータを元に分割

2. グループを使って新たにコミュニケータを作る
   - MPI_Comm_create

- コミュニケータの削除
   - MPI_Comm_free

コミュニケータの分割

- プロセッサ群を分割したい場合、MPI_Comm_split 関数を利用
- ierr = MPI_Comm_split (comm, color, key, newcomm);
  - MPI_Comm comm (IN) : 元になるコミュニケータ
  - int color (IN): 同じcolorを持つランクが同じコミュニケータに入る
  - int key (IN): keyの小さいプロセスから，0から順にランクを割り当てる
  - MPI_Comm* newcomm (OUT): 新しいコミュニケータ
- commから，同じcolorを持つ複数のコミュニケータに分割
- 各newcommにはプロセスの重複はない
- **comm中の全プロセスが必ず実行すること**

MPI_Comm_splitの例

ランク	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
color	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
key	7	8	3	2	1	5	4	2	9	1	2	3	0	0	4	2

MPI_Comm_split

new-comm	(comm0)				comm1				comm2				comm3			
new-commにおけるランク	2	3	1	0	0	3	2	1	3	0	1	2	0	1	3	2
													1	0	3	2

グループからコミュニケータを生成(1)

- グループは，まずコミュニケータから生成する必要がある

```
• ierr = MPI_Comm_group(comm, group);
```

  - MPI_Comm comm (IN):　　元になるコミュニケータ
  - MPI_Group* group (OUT):得られたグループ

グループからコミュニケータを生成(2)

- 新しいグループを生成，ここでは部分集合を作成
- `ierr = MPI_Group_incl(group, n, ranks, newgroup);`
  - `MPI_Group group (IN):` 元になるグループ
  - `int n (IN):` 新しいグループに含まれるメンバ数
  - `int ranks[] (IN):` グループに含まれるランクのリスト（配列）
  - `MPI_Group* newgroup (OUT):` 得られた新しいグループ
- 他にもグループを編集する手段はいろいろある
  - `MPI_Group_union, MPI_Group_intersection, MPI_Group_difference`
  - `MPI_Group_excl, MPI_Group_range_{incl,excl},…`

グループからコミュニケータを生成(3)

- グループからコミュニケータを生成

```
• ierr = MPI_Comm_create(comm, group, newcomm);
```

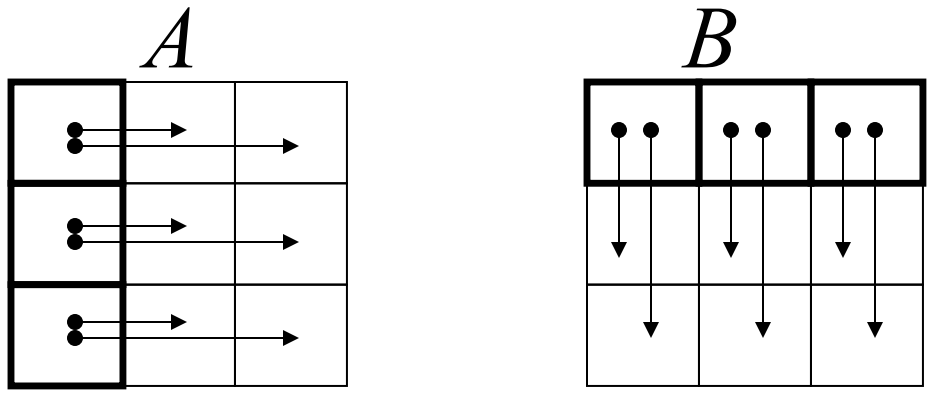
  - MPI_Comm comm (IN): 元になるコミュニケータ
  - MPI_Group group (IN): 部分集合グループ，commに含まれている必要がある
  - MPI_Comm* newcomm (OUT): 得られたコミュニケータ
- **commの中の全プロセスが必ず実行すること**

実例：SUMMAによる行列積

- SUMMA（Scalable Universal Matrix Multiplication Algorithm）
  - R.Van de Geijinほか、1997年
  - 同時放送（マルチキャスト）のみで実現

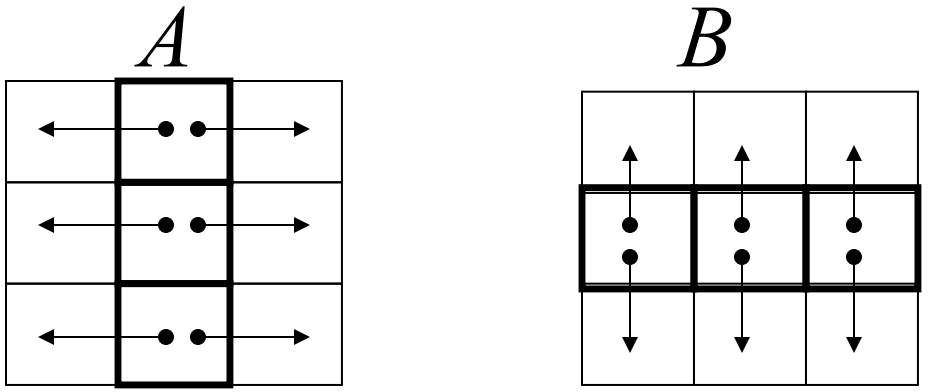
SUMMAアルゴリズムの概略

- 第一ステップ



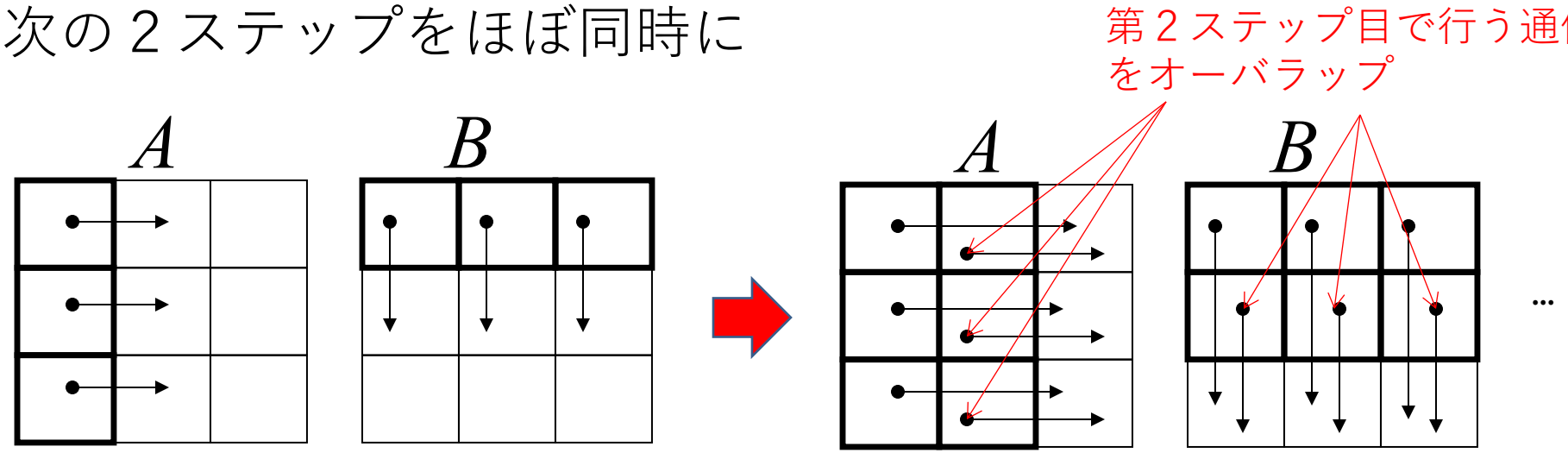
これを n/p

- 第二ステップ



SUMMA

- 特徴
  - 同時放送をブロッキング関数（例. `MPI_Bcast`）で実装すると、同期回数が多くなり性能低下の要因になる
  - SUMMAにおけるマルチキャストは、非同期通信の１対１通信（例. `MPI_Isend`）で実装することで、通信と計算のオーバラップ（***通信隠蔽***）可能
    - 次の２ステップをほぼ同時に



トポロジ

- MPIプロセス同士の接続情報
- 二次元以上の直交座標系に対してプロセスマッピングする際に使用すると便利
  - 各次元へのプロセス割り当て数を自動決定
  - 送受信先ランクの自動決定

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

プロセス数の割り当て

- 次元数を与えると適切なプロセス数を決定し、コミュニケータを割り当て

```
ierr = MPI_Dims_create(nnodes, ndims, dims);
```

  - int nnodes (IN): 全プロセス数
  - int ndims (IN): 座標系の次元
  - int dims[] (INOUT): 各次元ごとのプロセス数

- 各次元ごとのプロセス数に非零数があったら尊重する

次元情報を持つコミュニケータ作成

```
ierr = MPI_Cart_create(comm_old, ndims, dims,
                       periods,reorder, comm_cart);
```

- MPI_Comm comm_old (IN): 元にするコミュニケータ
- int ndims (IN): 座標系の次元
- int dims[] (IN): 各次元の次数 (ndims次元分)
- int periods[] (IN): 周期境界かどうか(ndims次元分)
- int reorder (IN): ランクを並べ替えるかどうか
- MPI_Comm *comm_cart (OUT):新しいコミュニケータ

- comm_oldに属する全プロセスが呼ぶ必要がある
- dimsにはMPI_Dims_create()で得られたものをそのまま渡せば良い

座標を得る

```
ierr = MPI_Cart_coords(comm, rank, maxdims,coords);
```

- MPI_Comm comm (IN):　　コミュニケータ
- int rank (IN):　　ランク番号 (comm内の)
- int maxdims (IN):　　次元数
- int coords[] (OUT):　　座標(の配列)

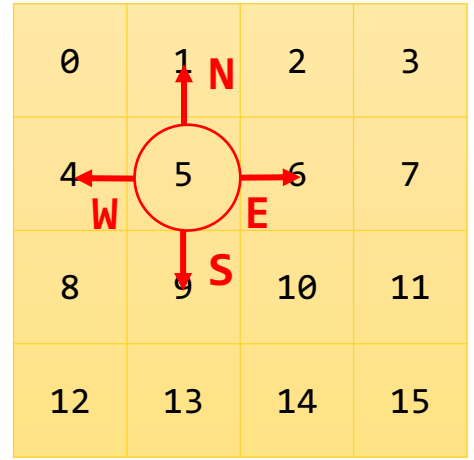
- commはMPI_Cart_createにおいて作成したコミュニケータ
  - ランク番号は(create時にreorder=1なら)以前と変わる可能性があることに注意 => MPI_Comm_rankで改めてランク番号を取得

通信相手ランクを得る

```
ierr = MPI_Cart_shift(comm, direction, displ, source, dest);
```

- MPI_Comm comm (IN): コミュニケータ
- int direction (IN): 座標の次元番号
- int displ (IN): 相手までの距離
- int * source (OUT): (自分にとっての)送信元ランク
- int * dest (OUT): 送信先ランク

- 二次元の場合、以下のようにすれば隣接ノードのランクが得られる
  - MPI_Cart_shift(comm, 0, 1, &north, &south);
  - MPI_Cart_shift(comm, 1, 1, &west, &east);
  - 次元順の取り方はFortran的



演習

- コミュニケータを作る
  - Comm/
    - comm-split: MPI_Comm_split()
    - comm-group: MPI_Comm_group(), MPI_Group_...
    - cart: MPI_Cart_...
- SUMMAへの適用
  - Mat-Mat-summa/

Hybrid並列

MPIのマルチスレッド対応

- アプリケーション中では1スレッドしか許さない
  - MPI_THREAD_SINGLE
- マルチスレッド対応
  - MPI_THREAD_FUNNELED
    - マスタースレッドのみがMPIを呼ぶ
  - MPI_THREAD_SERIALIZED
    - 誰でもMPIを呼べるが，内部では逐次化される
  - MPI_THREAD_MULTIPLE
    - 完全なマルチスレッド動作

各モードにおける記述

- SingleではParallelリージョンから出なければいけない
- Funneledではmasterだけが呼べる
- Serialized, Mutipleは誰が呼んでもいい
  - タグで通信を区別する必要

Single

```
#pragma omp parallel
{
....
}
   MPI_Send( ... );
#pragma omp parallel
 {
....
}
```

Funneled

```
#pragma omp parallel
{
  ....
#pragma omp master
  MPI_Send( ... );
  ....
}
```

Master onlyと似ているが
parallel節を閉じなくていい

Serialized, Multiple

```
#pragma omp parallel
{
  ....
  MPI_Send( ... );
  ....
}
```

マルチスレッドMPIの初期化

- MPI_Init()の代わりに、MPI_Init_thread()を使用

- C言語:

```
int provided;
MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED,
&provided);
```

- Fortran

```
integer required, provided
required=MPI_THREAD_FUNNELED
call MPI_Init_thread(required, provided, ierr)
```

Probe / Iprobe

- 実際に受信する前に受信データをチェックしてステータスのみを取得

```
ierr = MPI_Probe (source, tag, comm, status)
ierr = MPI_Iprobe (source, tag, comm, flag, status)
```

  - int source (IN), int tag (IN), MPI_Comm comm (IN), MPI_Status *status (OUT): MPI_Recvと同様
  - int *flag (OUT): flag=TRUEのときstatusを利用可能

- メッセージサイズを知って受信バッファサイズを確保
- nonblocking通信の加速
  - 実装によっては，MPI_Waitするまで通信が起こらない
  - MPI_Probeによってバックグラウンドの処理が進む

MPI_Probeの利用例

```
MPI_Status status;
int count;
int *rbuf;


MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
MPI_Get_count( &status, MPI_INT, &count );
// statusから受信データ数を取得
MPI_Alloc_mem( count*sizeof(int), MPI_INFO_NULL, rbuf );
// メモリ確保
MPI_Recv( rbuf, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
//  実際の受信, Fortranでは代わりに status(MPI_SOURCE), status(MPI_TAG)
```

マルチスレッドにおけるprobe

- MPI_Probe, MPI_Iprobeでは，スレッド間で取りちがえる可能性
- MPI_Mprobe, MPI_Improbeおよび MPI_Mrecv, MPI_Imrecvを使う

```
ierr = MPI_Mprobe(source, tag, comm, message, status)
ierr = MPI_Improbe(source, tag, comm, flag, message, status)
ierr = MPI_Mrecv (buf, count, datatype, message, status)
ierr = MPI_Imrecv(buf, count, datatype, message, request)
```

- MPI_Message *message : Mrecv/Imrecvのためのメッセージハンドル

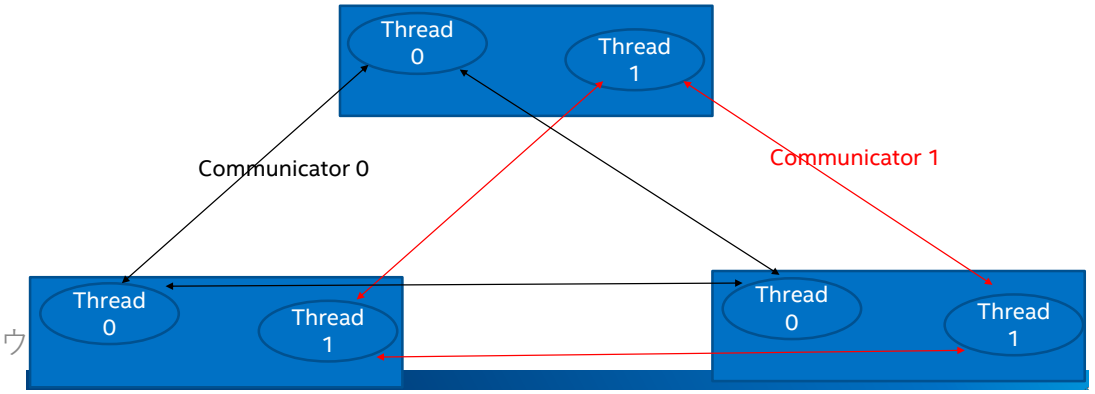
MPI_THREAD_MULTIPLEでの注意

- デフォルトでサポートしていない実装系もある
  - ビルド時に明示的に有効にする : Open MPI
  - MPI_Init_thread の providedで確認
- 性能を出すのが難しい面がある
- 集団通信には使えない
  - MPI_Comm_dup でスレッドごとに別のコミュニケータを使う
- 各スレッドで明示的にタグやコミュニケータを使い分けることで通信を識別する必要がある

Multiple Endpoint拡張

- Intelによる拡張、条件を制限することで高速化
  - 2019以降で利用可能(?)
- export I_MPI_THREAD_SPLIT=1
- export I_MPI_THREAD_RUNTIME=openmp
- export PSM2_MULTI_EP=1
- export I_MPI_THREAD_MAX=<通信に使う最大スレッド数>
  - プログラム用の指定は別途必要

Rule: Thread N in rank A has to communicate with Thread N in rank B in the same communicator.



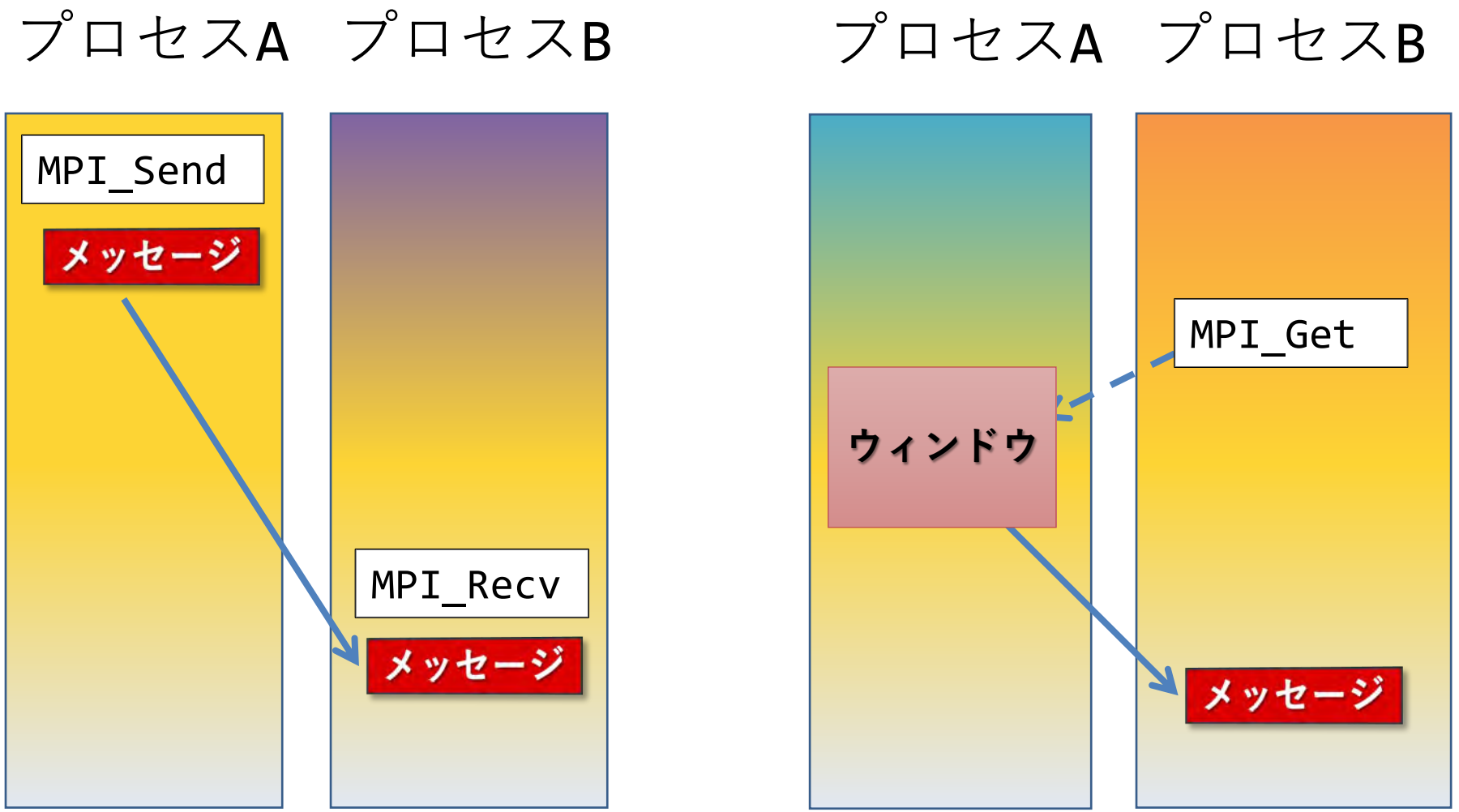
片側通信 /
One sided communication

片側通信

- 一方のプロセスのみで通信が完結
  - 通信相手の状態には無関係に記述できる

- 片側通信の利点
  - ハードウェア通信機構 Remote DMA (RDMA)との親和性
  - データコピーの削減
  - 同期コストの削減

１対１通信　vs　片側通信



基本的な使い方の流れ

1. Windowオブジェクトの生成，割り当て
2. エポックの開始
3. ターゲットへのアクセス
4. 同期
5. エポックの終了
6. Windowオブジェクトの解放

Windowオブジェクトの生成

- リモートメモリアクセス可能なWindowを定義
- すでに配列がある場合

```
ierr = MPI_Win_create(base, size, disp_unit, info, comm, win);
```

  - void *base (IN):　　ウィンドウにしたいメモリの先頭アドレス
  - MPI_Aint size (IN):　　サイズ（バイト数）
  - int disp_unit (IN):　　Put, Get等で指定されるデータのサイズ
  - MPI_Info info (IN):　　ヒント情報（後述）
  - MPI_Comm comm (IN):　　コミュニケータ
  - MPI_Win *win (OUT): 得られるWindowオブジェクト

- コミュニケータの全メンバーが呼ぶこと

Windowオブジェクトの割り当て

- リモートメモリアクセス可能なWindowを定義
- 新たに配列を確保する場合

```
ierr = MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win);
```

  - MPI_Aint size (IN): サイズ（バイト数）
  - int disp_unit (IN): Put, Get等で指定されるデータのサイズ
  - MPI_Info info (IN): ヒント情報（後述）
  - MPI_Comm comm (IN): コミュニケータ
  - void *baseptr (OUT): ウィンドウになるメモリの先頭アドレス
    Fortranの場合、type(C_PTR)でないといけない
    （演習の解答例を参照）
  - MPI_Win *win (OUT): 得られるWindowオブジェクト
- コミュニケータの全メンバーが呼ぶこと

Windowオブジェクトの解放

```
ierr = MPI_Win_free(win);
```

- MPI_Win *win (INOUT): 得られるWindowオブジェクト

MPI_Put: リモートメモリ書き込み

```
ierr = MPI_Put(origin_addr, origin_count, origin_datatype,
               target_rank, target_disp, target_count, target_datatype, win );
```

- void *origin_addr (IN):　自プロセスの先頭アドレス
  書き込みたいデータ
- int origin_count (IN):　自プロセスのデータ個数
- MPI_Datatype origin_datatype (IN):自プロセスのデータ型
- int target_rank (IN):　ターゲットのランク
- MPI_Aint target_disp (IN):　ターゲットのウィンドウからの位置
- int target_count (IN):　ターゲットに書き込むデータ個数
- MPI_Datatype target_datatype (IN):ターゲットのデータ型
- MPI_Win win (IN):　ウィンドウオブジェクト

- target_addr = window_base+target_disp×disp_unit (Window生成時)

MPI_Get: リモートメモリ読み出し

```
ierr = MPI_Get(origin_addr, origin_count, origin_datatype,
                target_rank, target_disp, target_count, target_datatype, win );
```

- void *origin_addr (OUT): 自プロセスの先頭アドレス, 読み出したデータが入る
- int origin_count (IN): 自プロセスのデータ個数
- MPI_Datatype origin_datatype (IN):自プロセスのデータ型
- int target_rank (IN): ターゲットのランク
- MPI_Aint target_disp (IN): ターゲットのウィンドウからの位置
- int target_count (IN): ターゲットに書き込むデータ個数
- MPI_Datatype target_datatype (IN):ターゲットのデータ型
- MPI_Win win (IN): ウィンドウオブジェクト

- target_addr = window_base+target_disp×disp_unit (Window生成時)

その他の関数

- MPI_Accumurate
- MPI_Get_accumurate
  - リダクション演算に似ている
- MPI_Compare_and_swap
- MPI_Fetch_and_op
  - Atomic演算

同期

- エポック：メモリアクセスを許可する期間
- 同期関連の操作と関連

- アクティブターゲット：両方のプロセスが関与
  1. MPI_Win_fence
  2. MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait (PSCW)
- パッシブターゲット：オリジン側のプロセスだけ，ターゲット側は何もしない
  - MPI_Win_lock/MPI_Win_unlock, MPI_Win_flush

Fence

- バリア同期に類似，winに属するプロセスが全て呼び出し
- 様々な組み合わせ，高い頻度で通信がある場合

```
ierr = MPI_Win_fence( assert, win);
```

- int assert (IN): 通常は0
- MPI_Win win (IN): ウィンドウオブジェクト

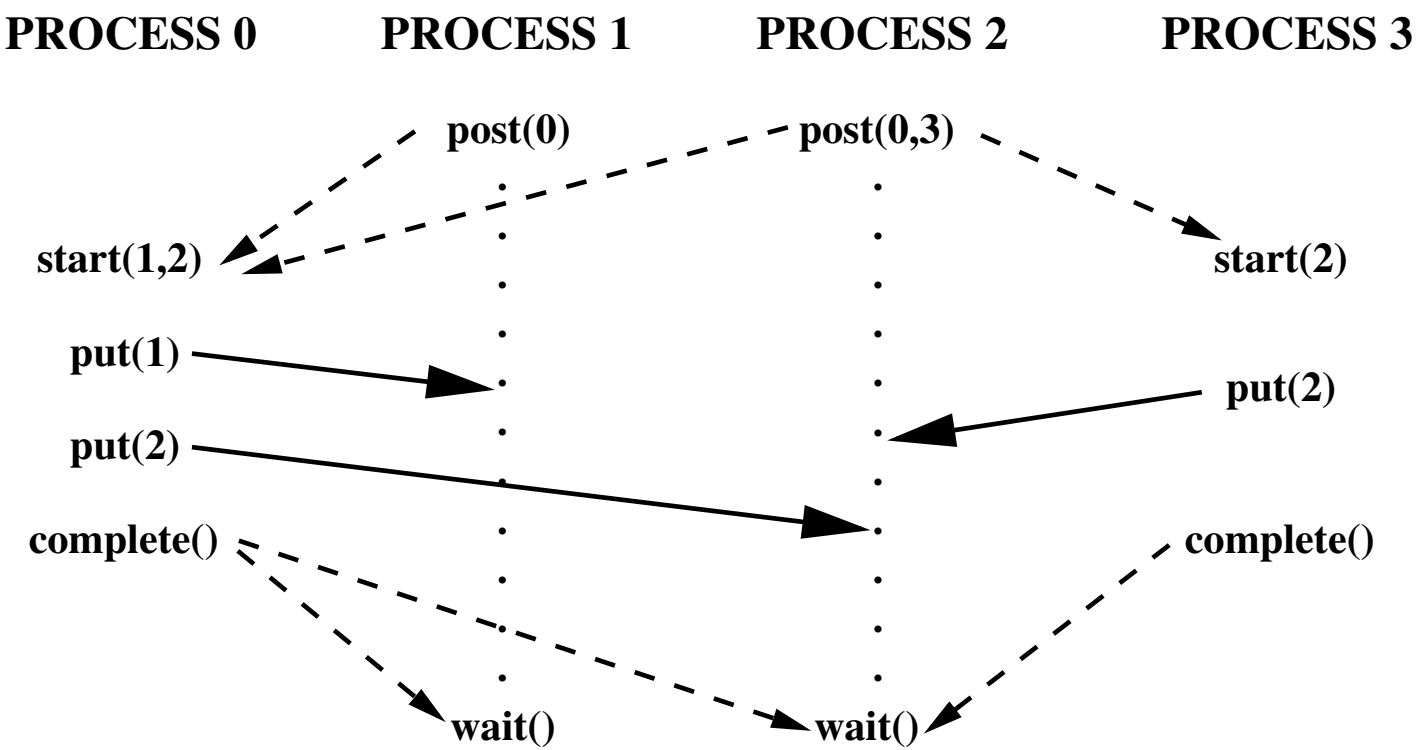
```
MPI_Win_fence(0, win);
if(rank == 0) MPI_Put( … , win);

MPI_Win_fence(0,win);
```

この間アクセス可能（エポック）

同期

PSCW同期

- 細かな同期の制御が可能
  - post-wait: ターゲット, start-complete: オリジン



Post, Start, Complete, Wait

```
ierr = MPI_Win_post(group, assert, win);
```

```
ierr = MPI_Win_start(group, assert, win);
```

```
ierr = MPI_Win_complete(win);
```

```
ierr = MPI_Win_wait(win);
```

- MPI_Group group (IN): 公開する相手のグループ
- int assert (IN): 通常は0
- MPI_Win win (IN): ウィンドウオブジェクト

- win内のgroupメンバーで必要なものだけが呼べば良い

Lock/Unlock

- パッシブターゲットのエポックを定義

```
ierr = MPI_Win_lock(lock_type, rank, assert, win);
```

- int lock_type (IN): MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED
- int rank (IN):　ターゲットのランク番号
- int assert (IN):　通常は0
- MPI_Win win (IN): ウィンドウオブジェクト

```
ierr = MPI_Win_unlock(rank, win);
```

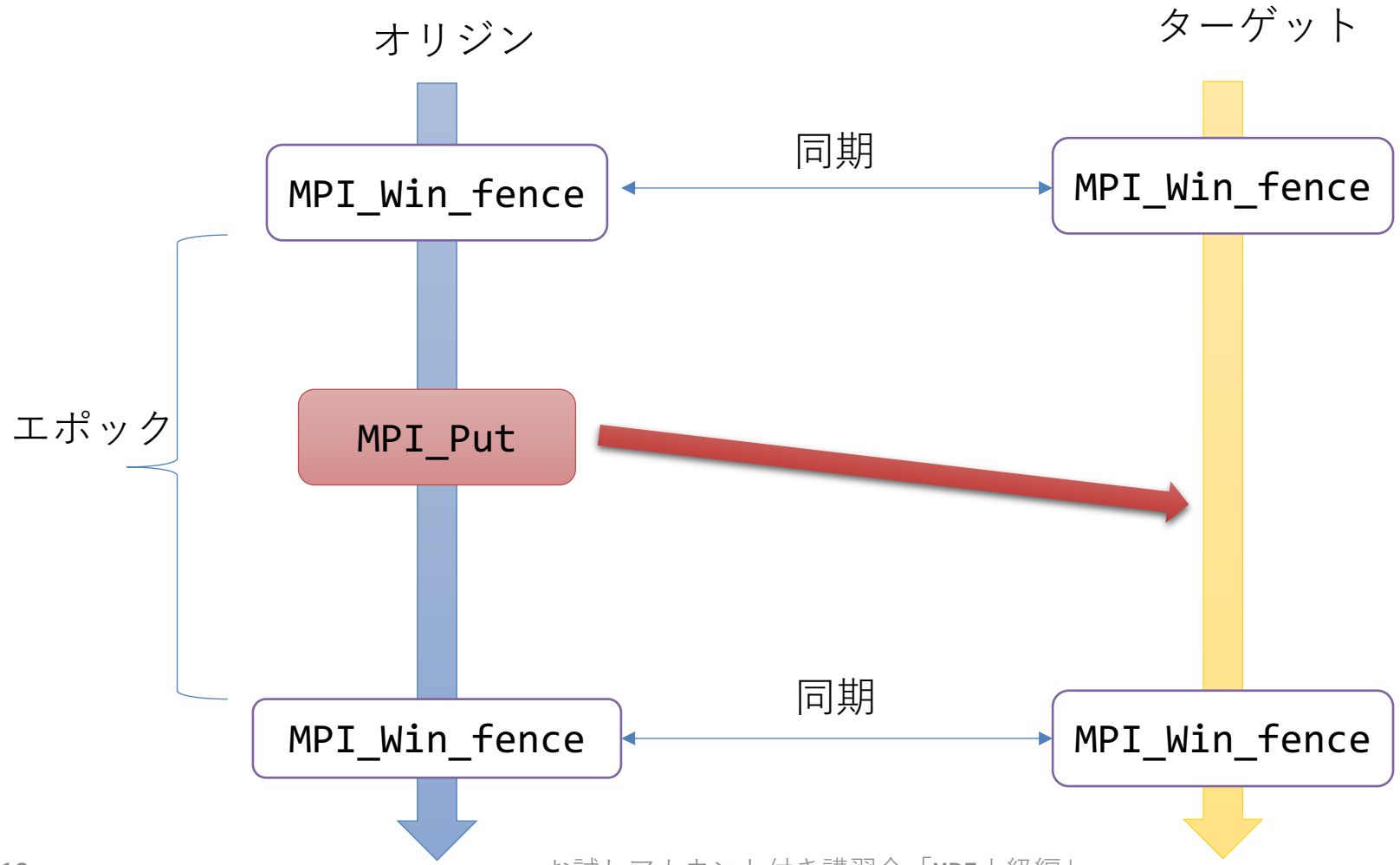
Flush

- パッシブターゲットのエポック内で利用可能
- 実行中の処理を完了させる

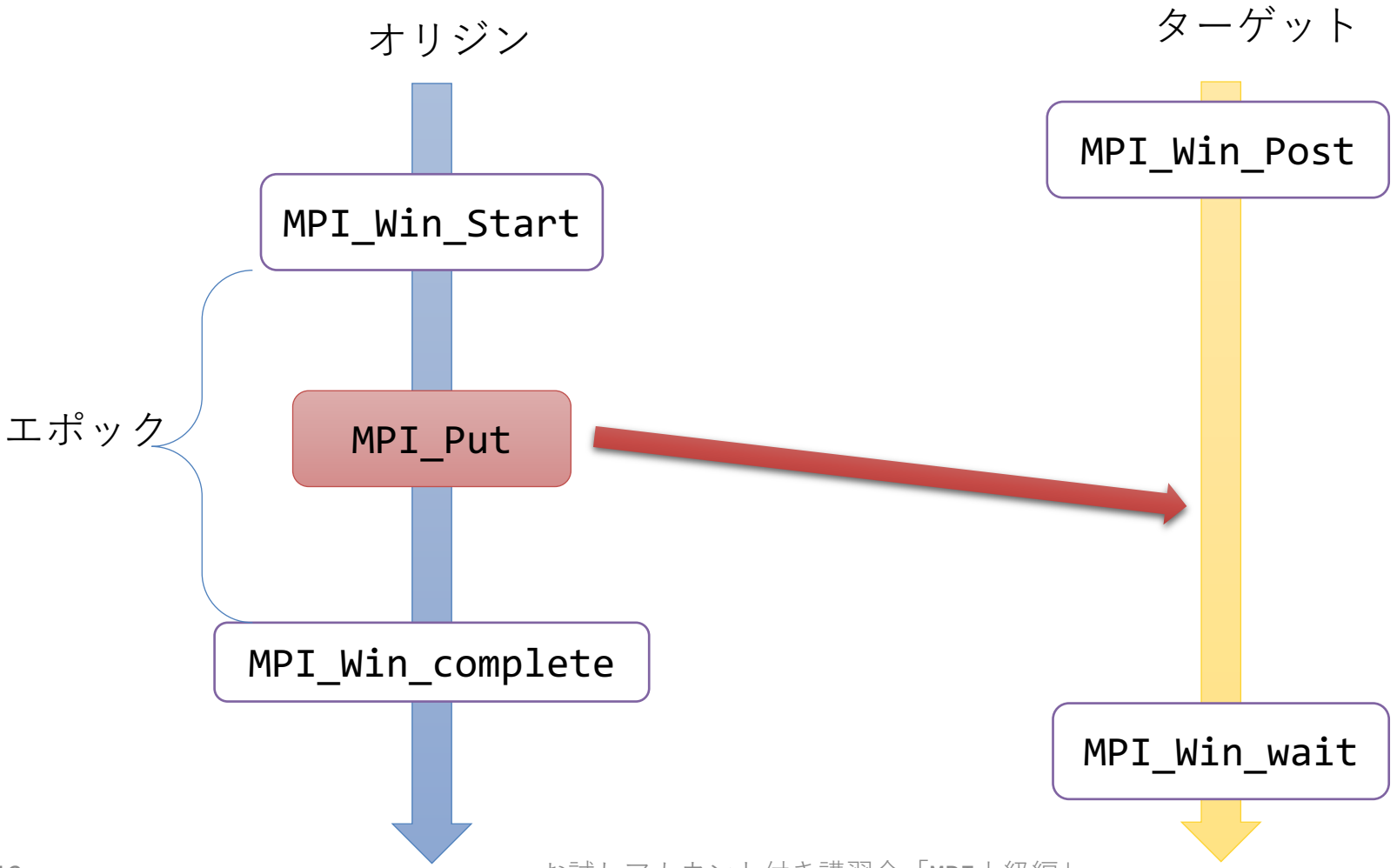
```
ierr = MPI_Win_flush(rank, win);
```

  - int rank (IN):　　ターゲットのランク番号
  - MPI_Win win (IN): ウィンドウオブジェクト

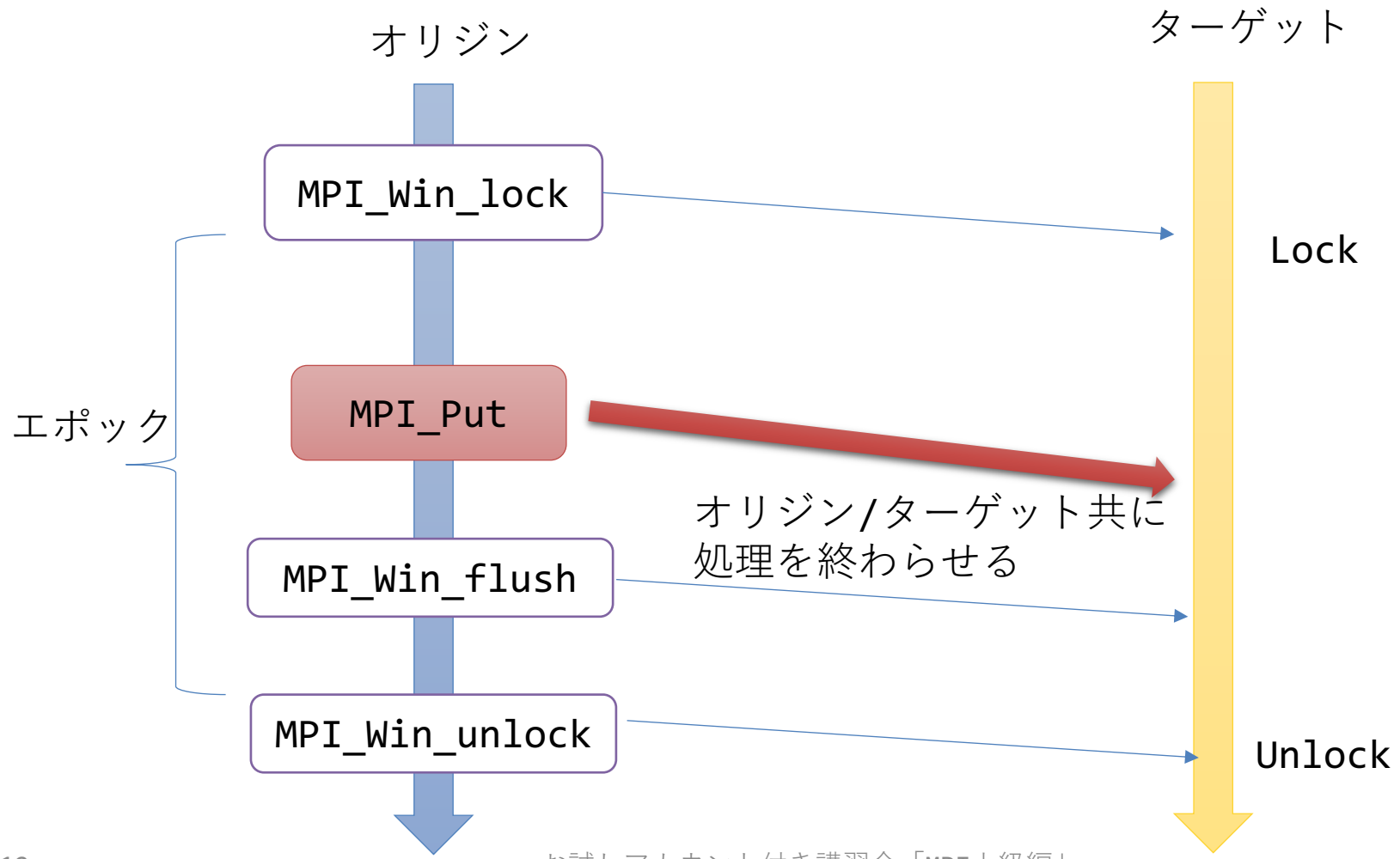
アクティブターゲット (Fence)



アクティブターゲット (PSCW)



パッシブターゲット (Lock)



演習

- 2つのプロセス間でデータ交換
  - Lock
  - Fence
  - PSCW
- 配列サイズ2N
- 前半のN個を相手の後半のN個分に上書き
  - MPI-advanced/onesided/
- 性能測定
  - MPI-advanced/MPIbench/onesided

お疲れ様でした！
アンケートにご協力お願いします

- https://forms.office.com/r/rWRk7gB8MV