はじめての
MPI片側通信プログラミング

-基本的な利用方法と
粒子系シミュレーションによる演習-

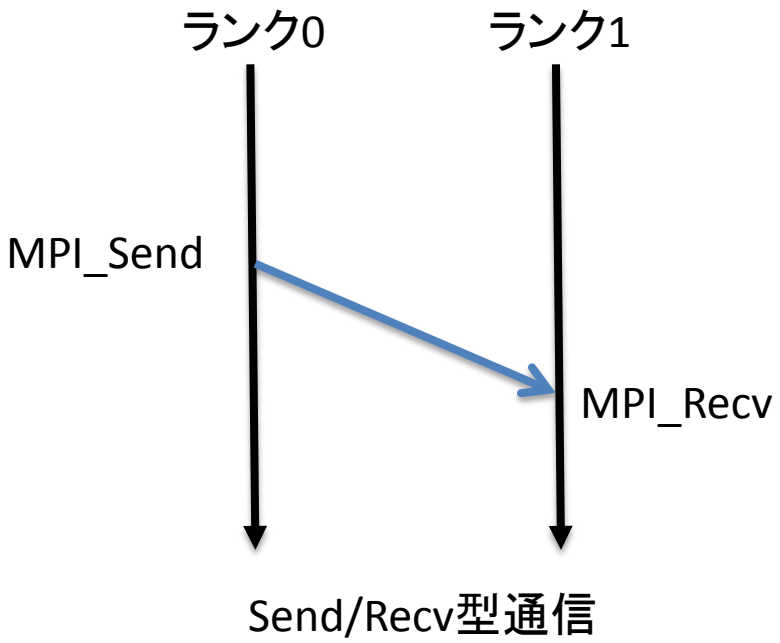
九州大学
情報基盤研究開発センター
森江　善之、薄田　竜太郎

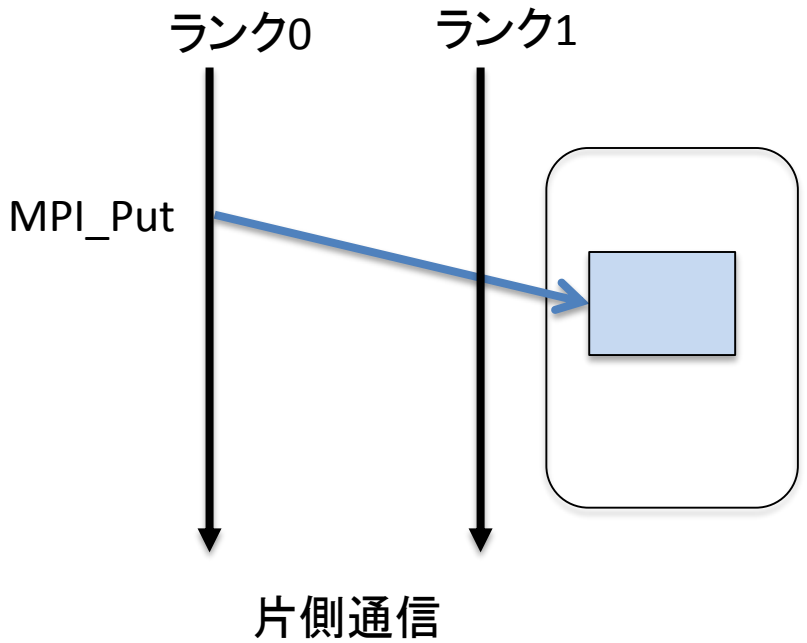
Message Passing Interface (MPI)

- Message Passing Interface (MPI)とは…
  - プロセス間のメッセージ通信のインターフェースの規格
  - Single Program Multiple Data (SPMD)
    - １個のプログラムで複数のデータを並列に処理
- 当初は、分散メモリ間の通信インターフェース(Send/Recv型通信)のみ
- MPI2.0で共有メモリの概念を導入し、片側通信インターフェースを追加

Send/Recv型通信と片側通信の違い

- Send/Recv型通信
  - 送受信の両側のプロセスで対応する関数を発行。
  - 自プロセスのメモリ領域のみへアクセス。
- 片側通信
  - 一方のプロセスのみで通信が完結。
  - 一方のプロセスが他方のメモリ領域へアクセス。



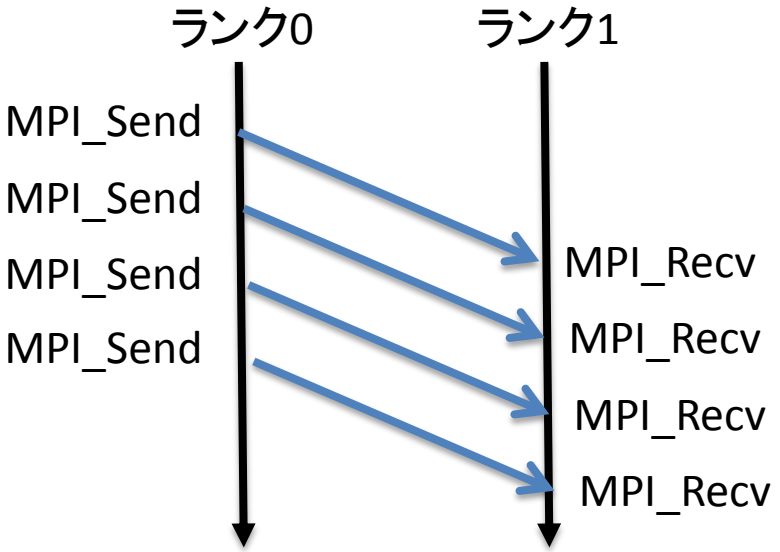


片側通信の利点

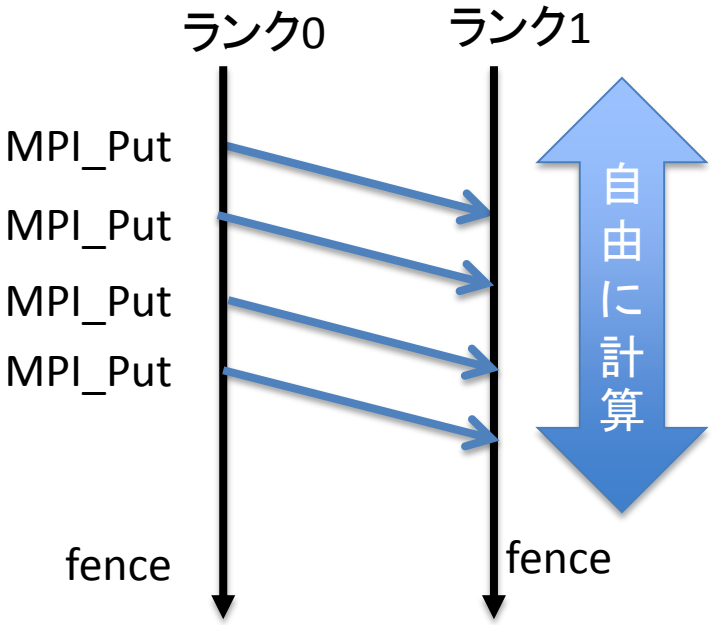
- 同期待ちの削減
- データコピーの削減
- RDMA機能との親和性
  - ハードウェア機能を直接使用
  - オーバラップ

同期待ちの削減

- Send/Recv型ではターゲットランクでMPI_RecvやMPI_Waitが必要。
- 片側通信ではターゲットランクは自由に計算が可能。



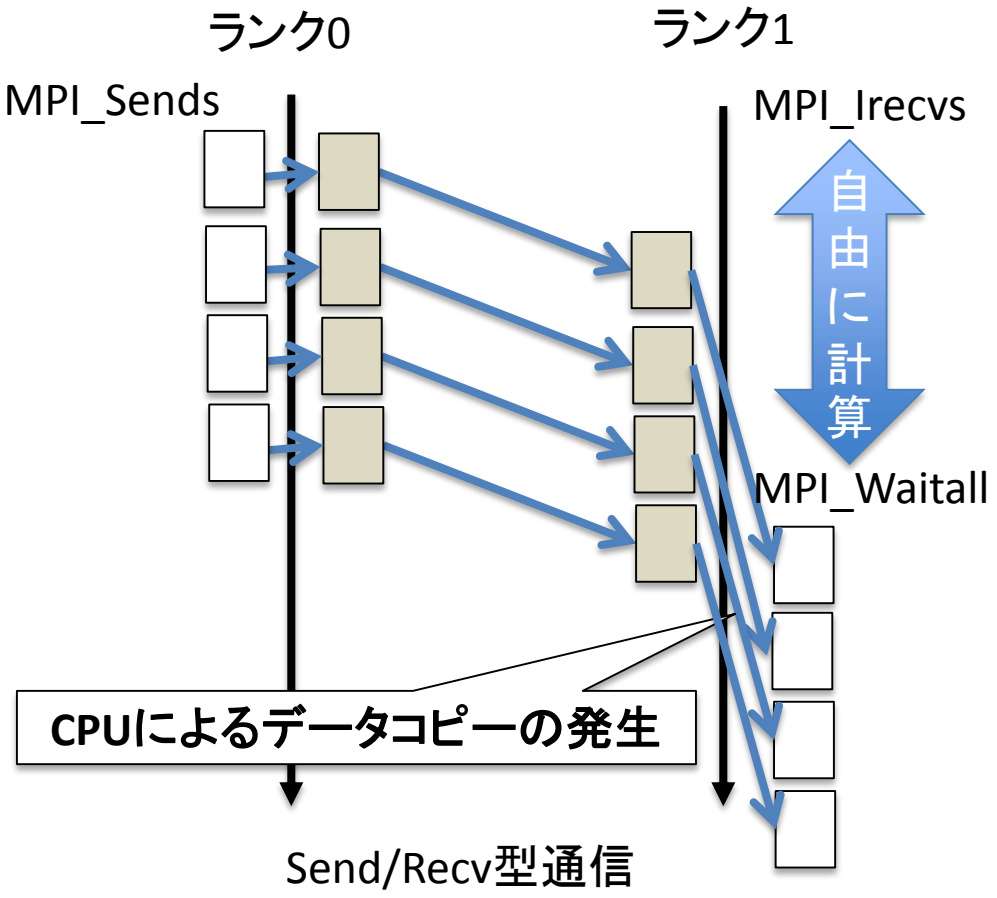
Send/Recv型通信

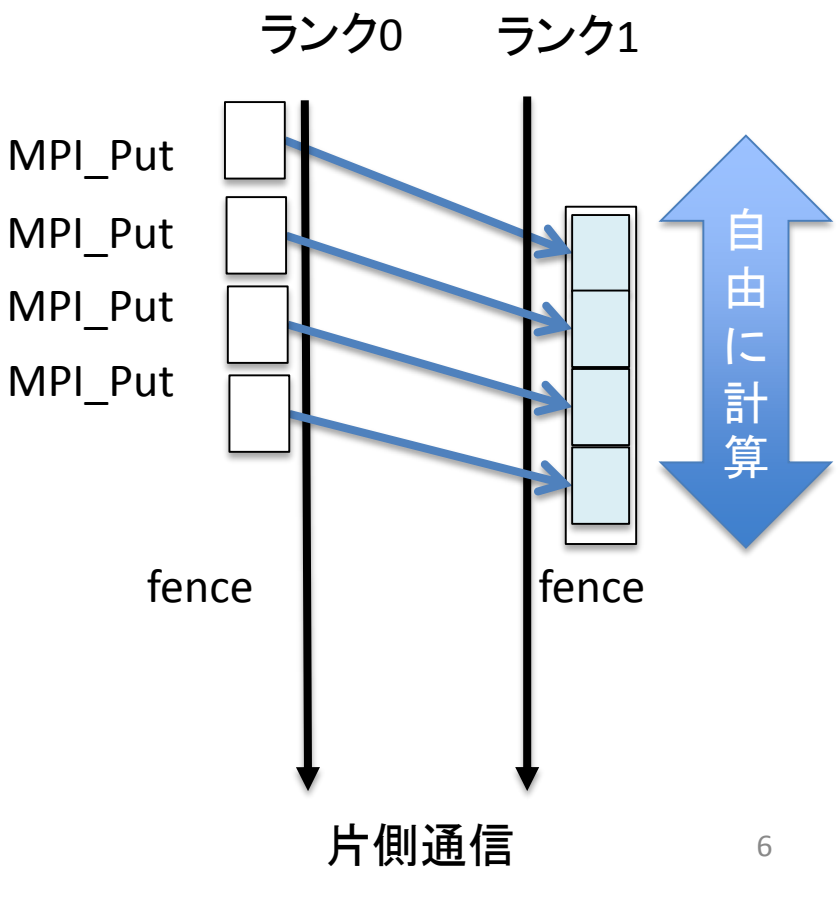


片側通信

データコピーの削減

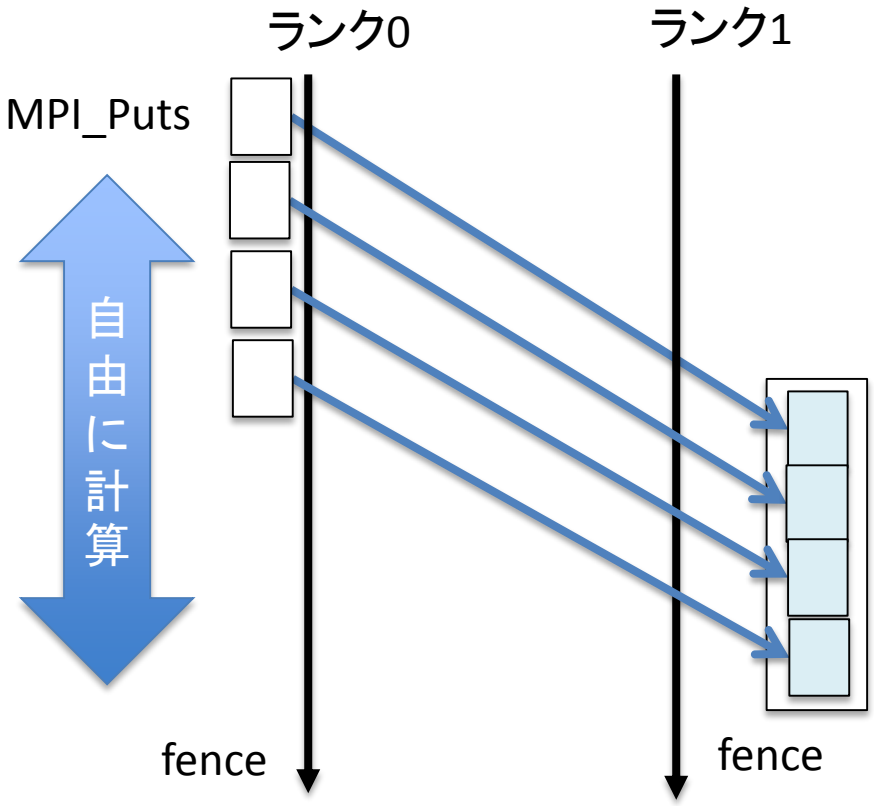
- Send/Recv型通信では、通信バッファを用いるため、**データコピーが発生**。
- 片側通信では**直接ユーザメモリ領域にコピー**。
- リモートプロセスでは自由に計算が可能。





RDMAと親和性

- 片側通信は**インターフェースが<u>RDMAと同等</u>**
  - 余計な処理がない → 高性能
- 通信デバイスに**<u>データコピー</u>**を任せることが可能 → オーバラップ



本講習について

- MPIの片側通信は当初利点がありながら性能面に問題があり、あまり普及しなかった。

- MPI片側通信の資料が少ない。
  - 特にアプリケーションに関する資料

- **MPI片側通信プログラミングの資料提供**

- 本講習会では…
  - 前半は、MPI片側通信インターフェースの説明
  - 後半は、片側通信を用いた粒子系シミュレーション演習

MPI片側通信インターフェース

MPI RMA

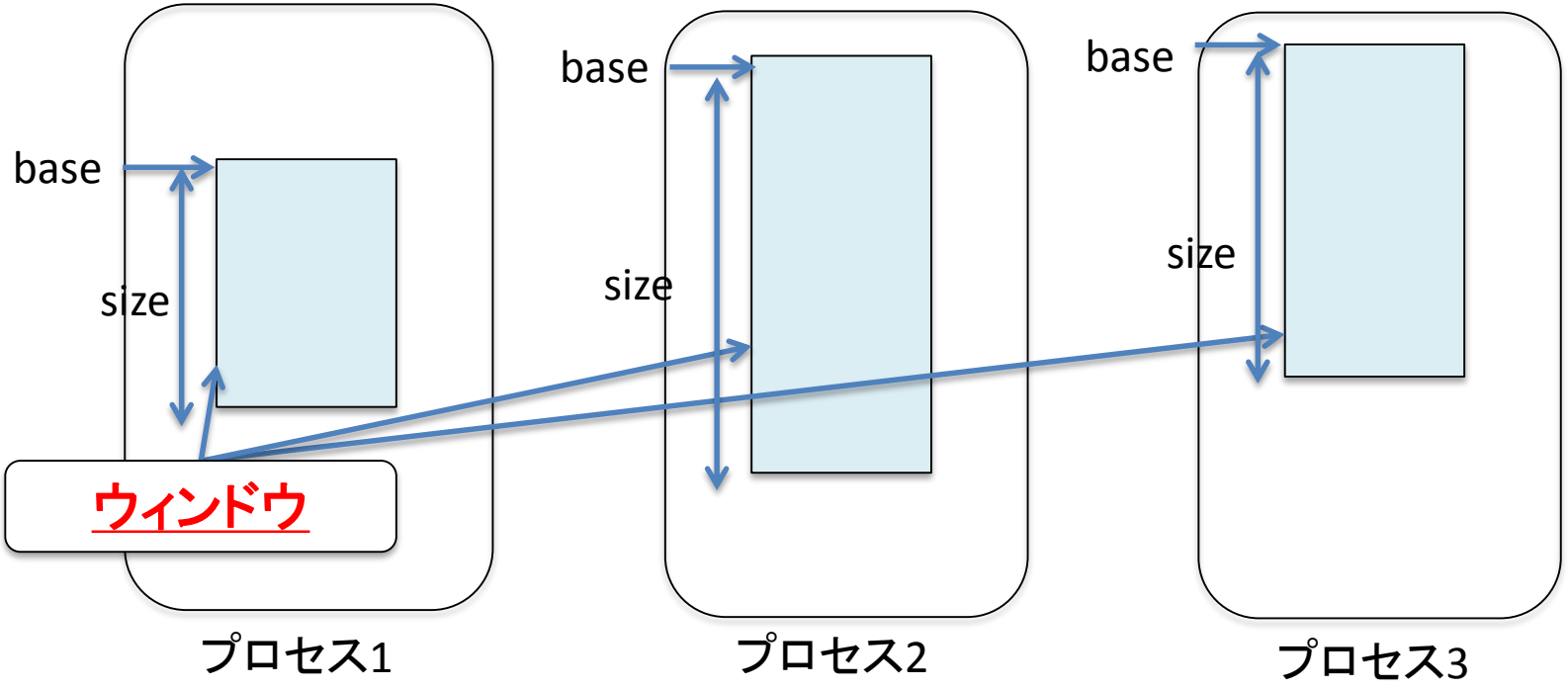
- MPIでは、MPI RMA (Remote Memory Access)として片側通信インターフェース群を提供

- RMAでは、一般にリモートプロセスへのメモリアクセスは許可が必要。

- MPI RMAでは、アクセス許可されたメモリ領域「ウィンドウ」を生成。

ウィンドウの指定

- 各プロセスでウィンドウを指定。



- ウィンドウオブジェクトを生成。
  - 各プロセスのウィンドウへアクセスするための情報体。
  - ウィンドウオブジェクトの生成は集団通信。

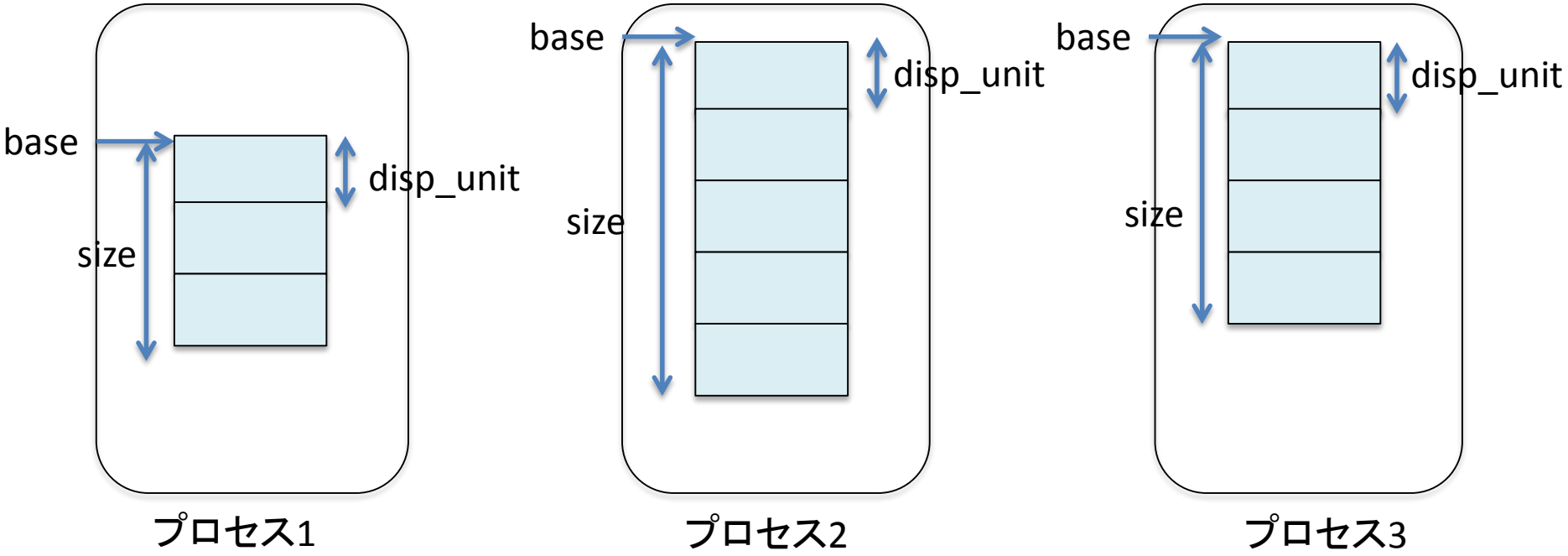
ウィンドウオブジェクトの生成関数: MPI_Win_create

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_info info
MPI_Comm comm, MPI_Win *win)
```

- 概要
  - RMA操作のためのウィンドウオブジェクトを生成。

- base: メモリ領域の先頭アドレスを指定。
- size: メモリ領域のサイズ。
- disp_unit: Put/Getで指定されるデータの単位。
- info: 最適化のための情報←後半で説明。
- comm: コミュニケータ。
- win: ウィンドウオブジェクト。

MPI_Win_createのdisp_unit



- disp_unitはPutやGetの単位データとなる。
  - ターゲットランクのアクセス領域の先頭アドレスの計算に使う。

ウィンドウオブジェクトの解放関数

```
int MPI_Win_free(MPI_Win *win)
```

- 概要
  - 生成したウィンドウオブジェクトを開放する。
- win: 生成したウィンドウオブジェクト。

データ書き込み関数：MPI_Put

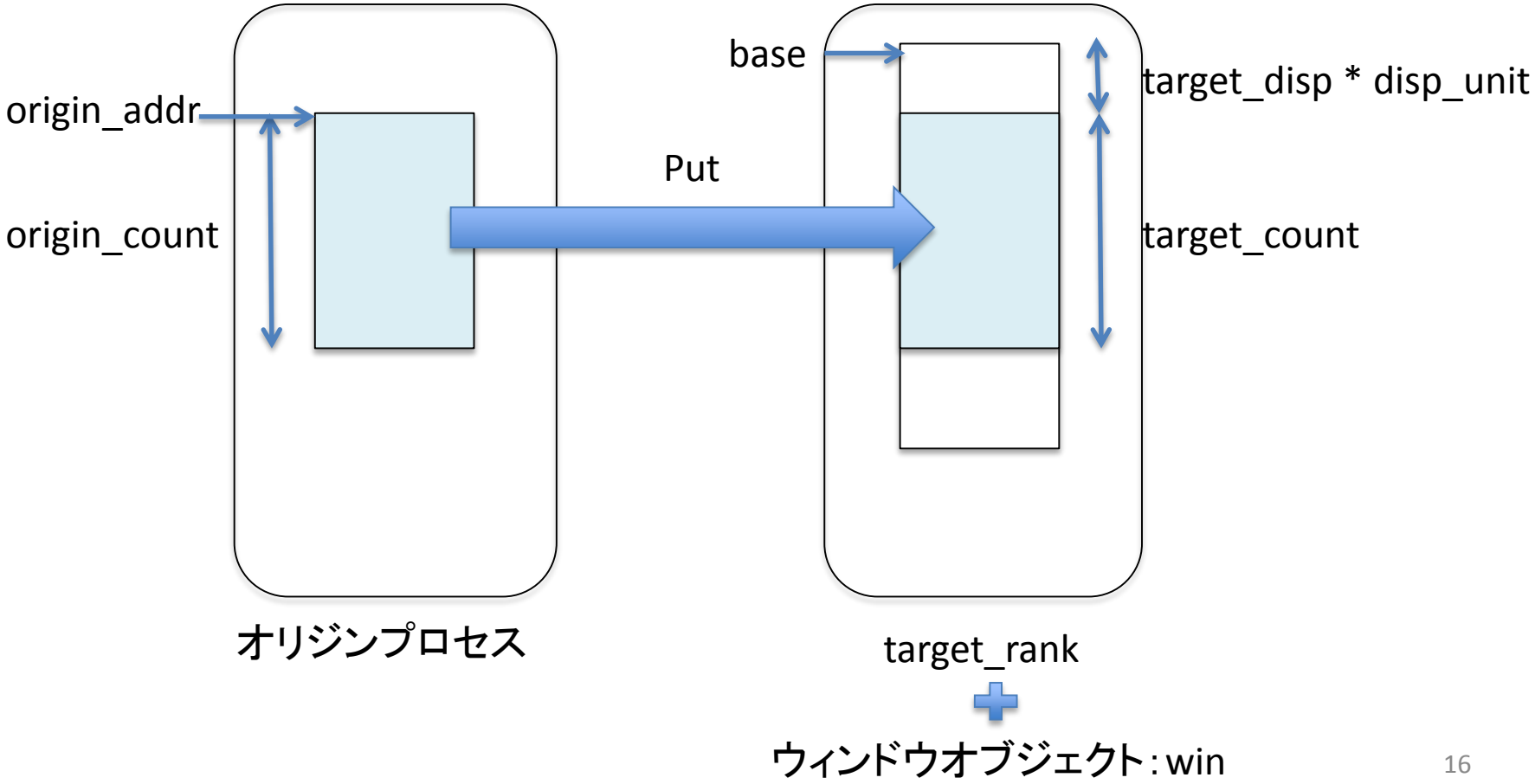
```
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win
            win)
```

- 概要
  - ターゲットプロセスへのデータ書き込みを行う。

- origin_addr ： 自プロセスのデータ元の先頭アドレス。
- origin_count ： データの個数。
- origin_datatype ： データの型。
- targert_rank ： ターゲットプロセスのランク。
- target_disp ： ウィンドウの先頭からのずれ。
- target_count： データの個数。
- target_datatype： データの型。
- win： ウィンドウオブジェクト。

MPI_Put

- MPI_Putの引数の意味。



データ読み込み関数: MPI_Get

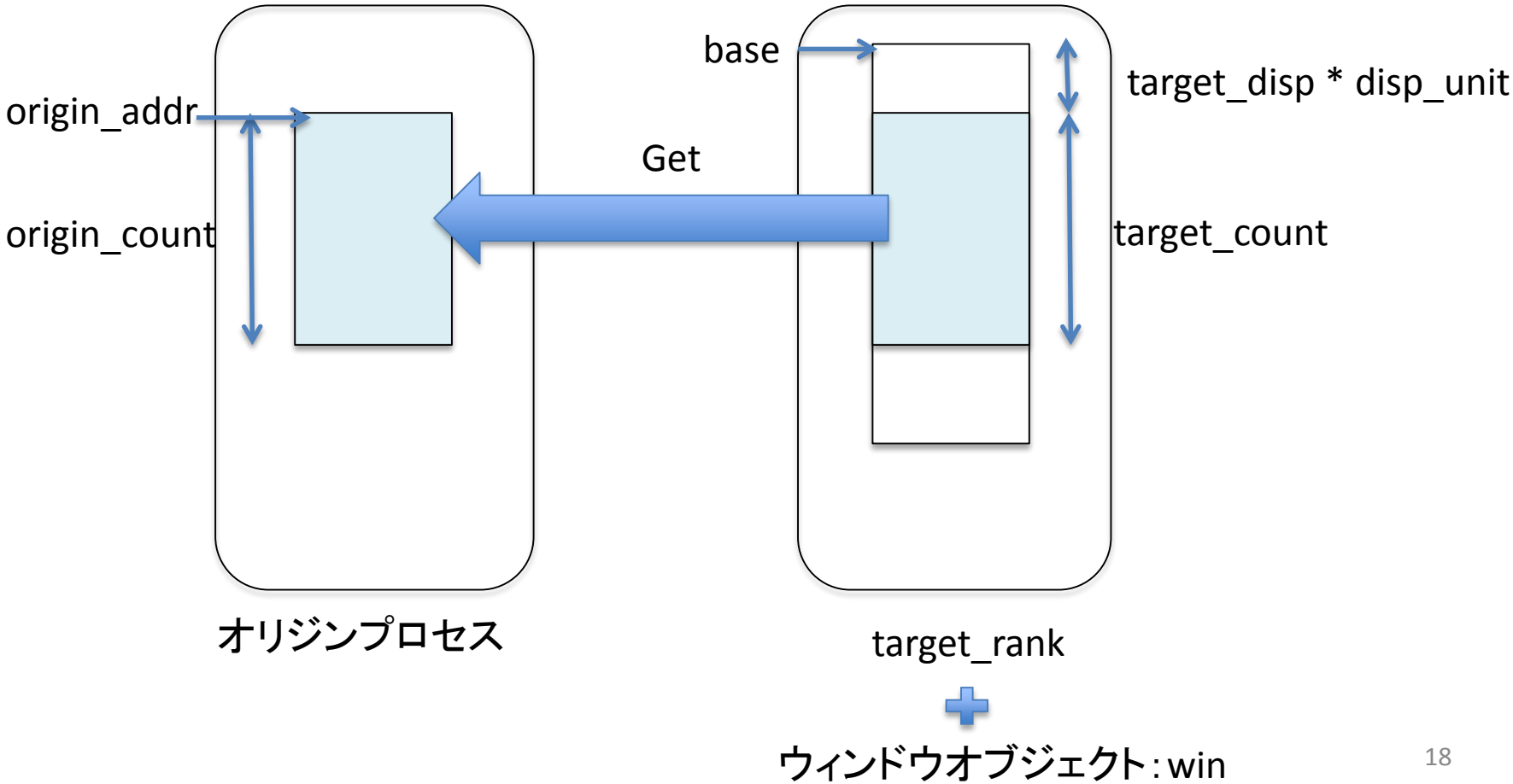
```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, MPI_Aint target_disp,
            int target_count, MPI_Datatype target_datatype, MPI_Win
            win)
```

- 概要
  - ターゲットプロセスからの読み込みを行う。

- origin_addr： 自プロセスのデータの格納先の先頭アドレス。
- origin_count： データの個数。
- origin_datatype： データの型。
- targert_rank： ターゲットプロセスのランク。
- target_disp： ウィンドウの先頭からのずれ。
- target_count: データの個数。
- target_datatype: データの型。
- win: ウィンドウオブジェクト。

MPI_Get

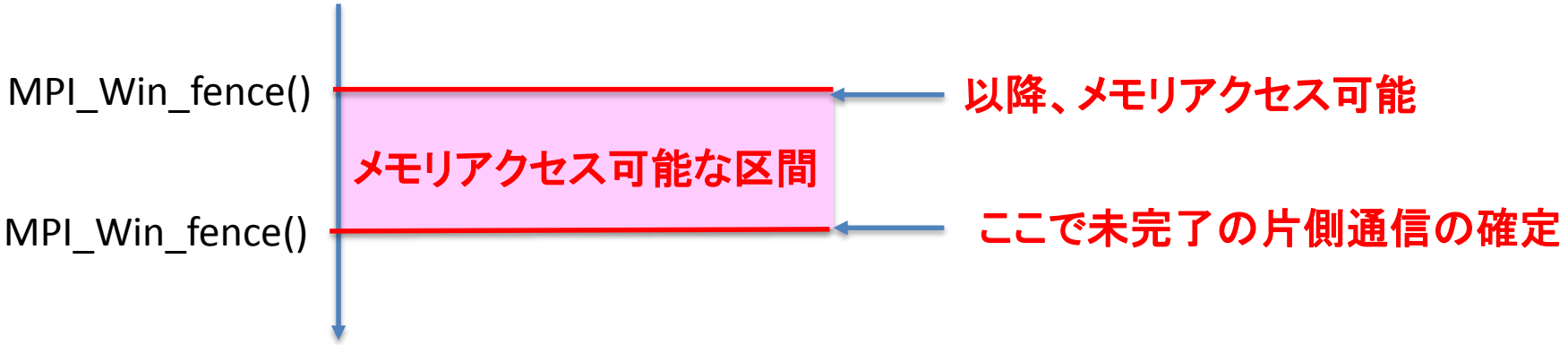
- MPI_Getの引数の意味。



メモリアクセス許可区間

- MPI片側通信は、「エポック」というメモリアクセス可能な区間でのみ機能する。

- エポックは同期関数により設定する。



- 同期関数は2種類存在する。
  - アクティブターゲットとパッシブターゲット。

片側通信の同期の種類

- アクティブターゲット
  - **両方のプロセス**が明示的に通信に関与。
    - MPI_Win_fence
    - MPI_Win_start/MPI_Win_complete/MPI_Win_post/MPI_Win_complete←今回は省略
- パッシブターゲット
  - **オリジンプロセスのみ**が明示的に通信に関与。
    - MPI_Win_lock/MPI_Win_unlock

アクティブターゲットの同期関数

```
int MPI_Win_fence(int assert, MPI_Win win)
```

- 概要
  - 指定されたwinオブジェクトに関する同期をとる。関数終了後、以前に発行された片側通信が完了する。

- assert: ウィンドウの状態を指定。
- win: ウィンドウオブジェクトの指定。

アクティブターゲットの疑似コード

```
MPI_Win_create( …, &win );

If ( rank == 0 )
    データの設定

 MPI_Win_fence ( 0, win );

If ( rank == 0 ) {
   MPI_Put( …, win );
}

MPI_Win_fence ( 0, win );

If ( rank == 1 )
    値の出力

MPI_Win_free( &win );
```

winに関連するプロセス全体で同期

パッシブターゲットの同期関数

```
int MPI_Win_lock (int lock_type, int rank, int assert, MPI_Win win)
```

- 概要
  - ターゲットプロセスのウィンドウへのエポックを開始する。

- lock_type： MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED
- rank: ターゲットランク。
- assert: ウィンドウの状態を指定。
- win: ウィンドウオブジェクトの指定。

```
Int MPI_Win_unlock(int rank, MPI_Win win)
```

- 概要
  - ターゲットプロセスのウィンドウへのエポックを完了する。

- rank: ターゲットランク。
- win: ウィンドウオブジェクトの指定。

片側通信のデータフラッシュ関数

```
int MPI_Win_flush(int rank, MPI_Win win)
```

- 概要
  - 指定されたウィンドウに対する呼び出し元が発行したすべての未完了の片側通信を完了する

- rank： ターゲットランク
- win： ウィンドウオブジェクト

パッシブターゲットの疑似コード

```
MPI_Win_create( …, &win );

If ( rank == 0 )
    データの設定

If ( rank == 0 ) {
   MPI_Win_lock( MPI_WIN_EXCLUSIVE, 1, 0, win );
   MPI_Put( …, win );
   MPI_Win_flush( 1, win );
   MPI_Win_unlock( 1, win );
}
```

同期を取っていない

```
If ( rank == 1 )
    値の出力

MPI_Win_free( &win );
```

ソースコードおよび使用法

1. ファイルのコピー
```
cp /home/mpirma/mpirma.tar.gz .
```
2. ファイルの展開
```
tar -xzvf mpirma.tar.gz
```

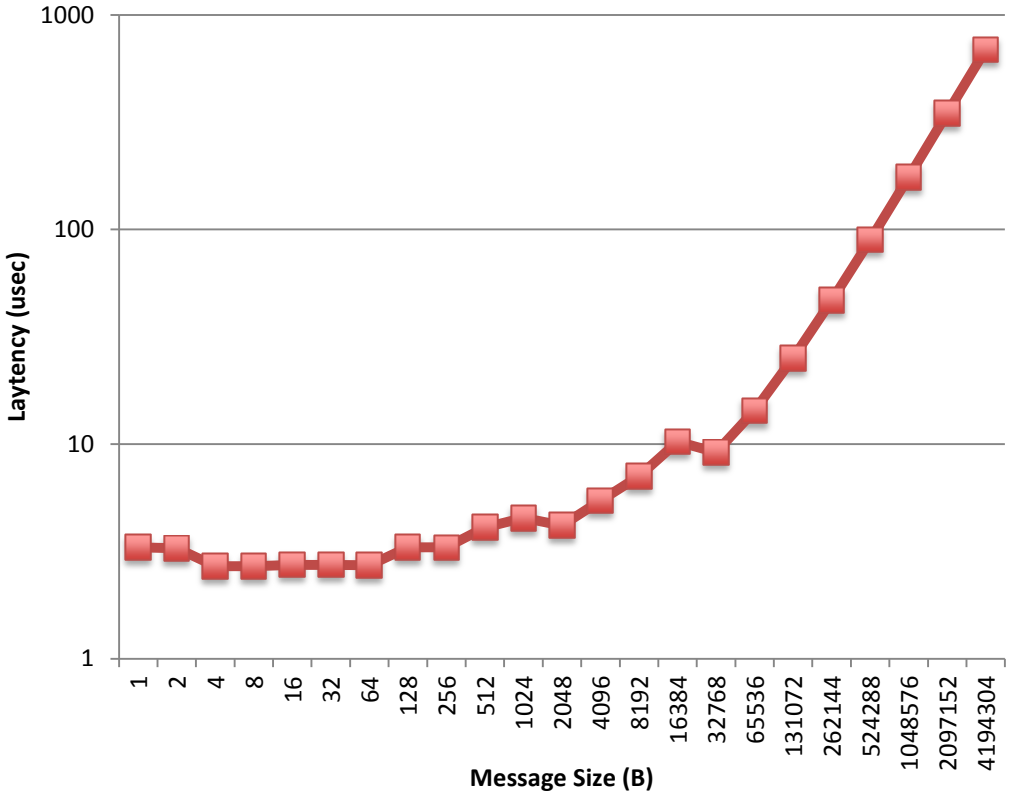
ファイル名	内容
mpi_active.c, mpi_passive.c, mpi_timing.c	ソースコード
mpi_active, mpi_passive, mpi_timing	実行オブジェクト
go_active.sh, go_passive.sh, go_timing.sh	ジョブスクリプト
Makefile	実行オブジェクトを生成

アクティブターゲットの例
1. ファイルの表示
```
less mpi_active.c
```
2. ジョブの投入
```
pjsub go_active.sh
```
3. 結果の表示
```
less mpi_active.out
```

パッシブターゲットの例
1. ファイルの表示
```
less mpi_passive.c
```
2. ジョブの投入
```
pjsub go_passive.sh
```
3. 結果の表示
```
less mpi_passive.out
```

MPI 片側通信の性能と動作

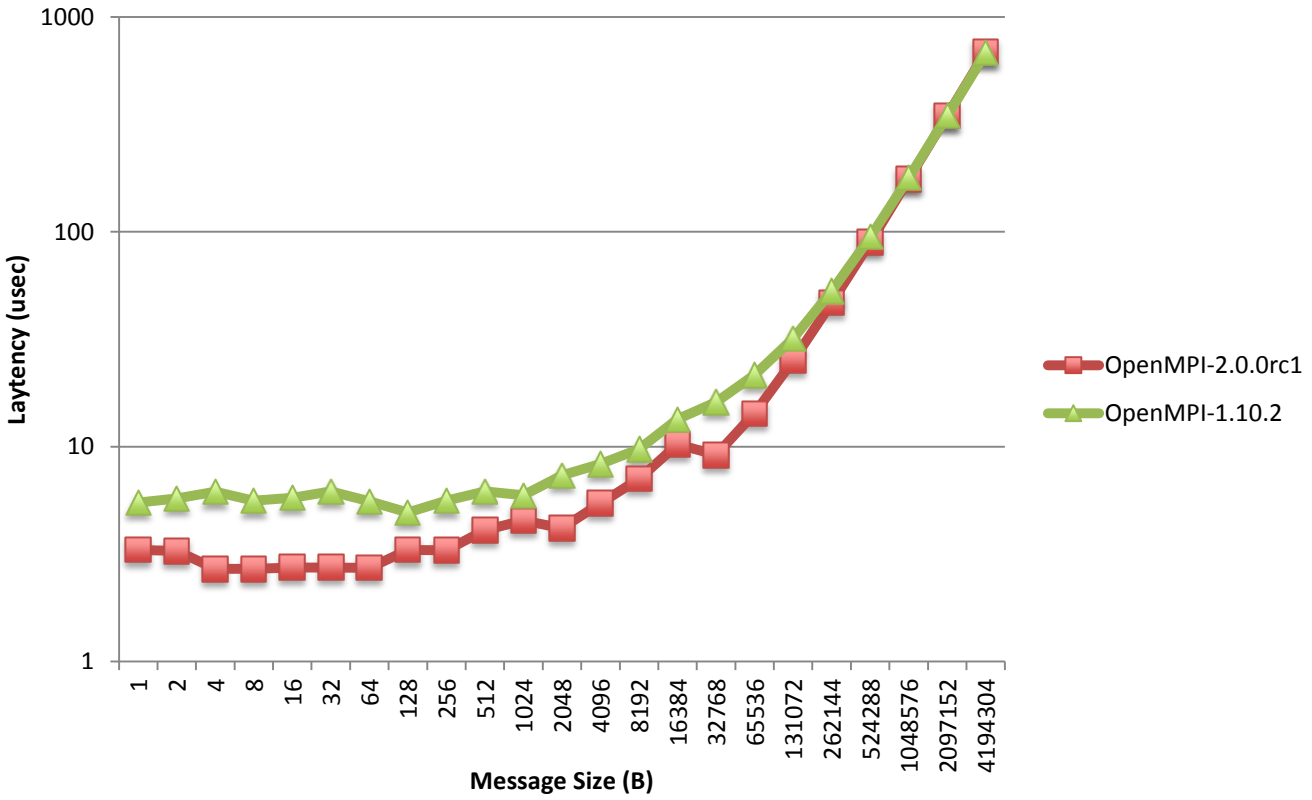
MPI 片側通信の通信性能



InfiniBand FDR
レイテンシ1.0 usec程度

- 小メッセージで 2.5 usec 程度。
- 小メッセージを単独で送るには向かない。
- Ex. 小メッセージをたくさん送る or 大メッセージを送る。

実装による通信性能の違い



- Open MPI-2.0.0rc1は、Open MPI-1.10.2より性能が高い。
  - Open MPI-2.0.0rc1はRMAインターフェースを使用。
  - Open MPI-1.10.2はSend/Recv型インターフェースを使用。

片側通信の動作の違い

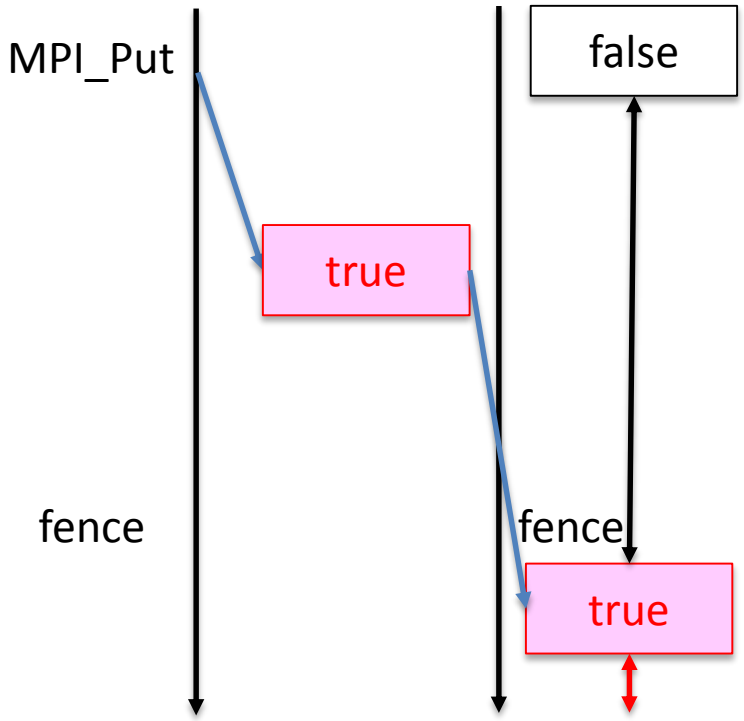
- 片側通信の処理を実行する主体により異なる。

実行主体	ターゲットプロセスでのデータ更新	通信性能	オーバラップ
ターゲットプロセス	MPI関数発行時	×	×
通信スレッド	**MPI関数の発行に非依存**	×	○
通信デバイス	**MPI関数の発行に非依存**	○	○

- 一般に通信デバイスによるRDMA機能を用いた場合に型が通信に期待される動作となる。

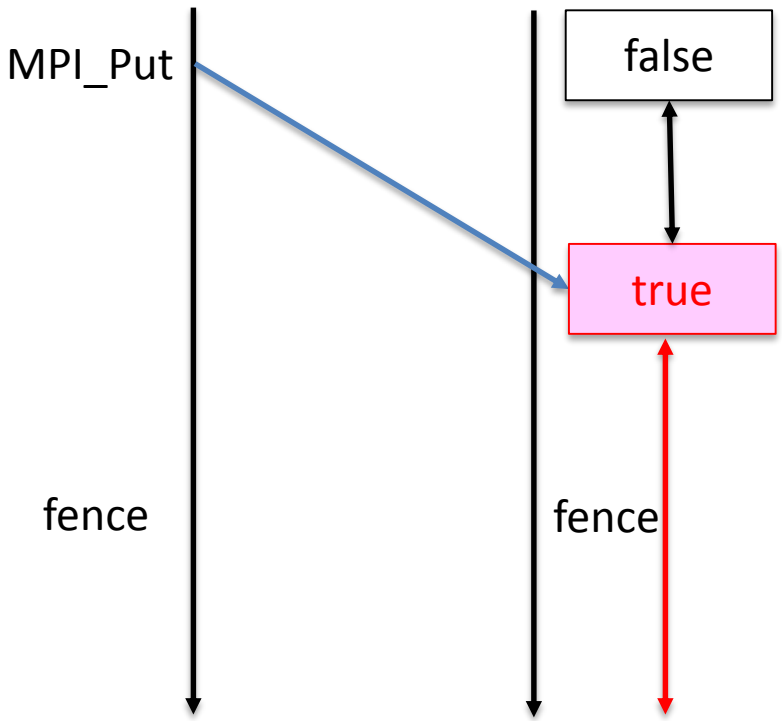
データの更新タイミング
- ターゲットプロセスでのデータの変更の検出 -

ノード内通信（共有メモリ）
```
pjsub go_timing1.sh
less mpi_timing1.out
```



ターゲットプロセス自身でデータ更新

ノード間通信（InfiniBand）
```
pjsub go_timing2.sh
less mpi_timing2.out
```



通信デバイス/通信スレッドがデータ更新

ここまで質問ありますでしょうか？

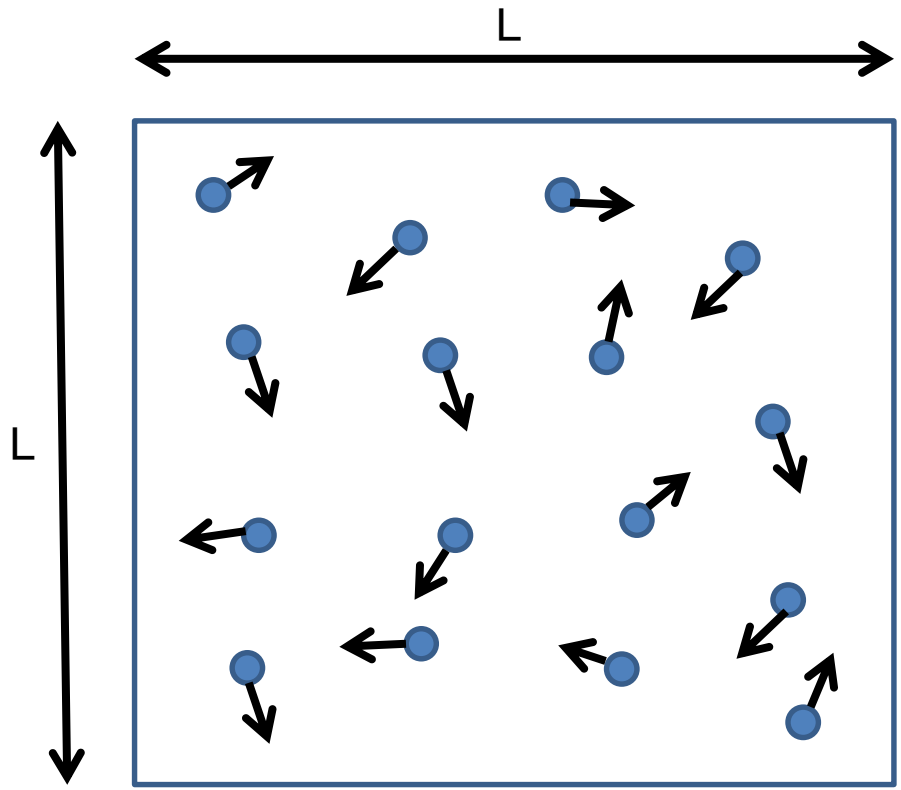
片側通信による
粒子系シミュレーション

話の流れ

- 粒子系シミュレーション
- 片側通信によるデータ移動
  - ウィンドウオブジェクトの作成
  - 送信バッファへデータをコピー
  - 移動先アドレスの取得
  - データ移動
- Send/Recv型通信を用いた場合
- 片側通信のチューニング
- MPI規格のバージョン

粒子系シミュレーション

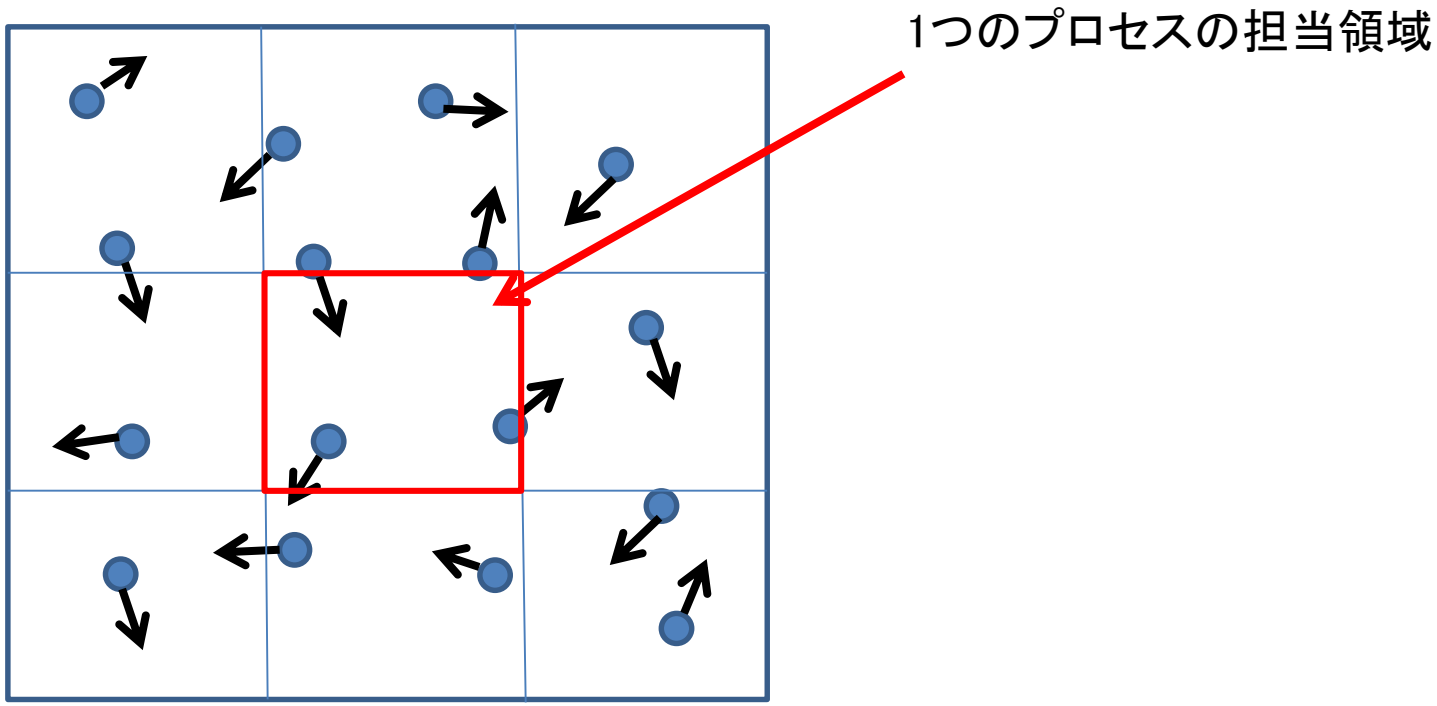
粒子系シミュレーション



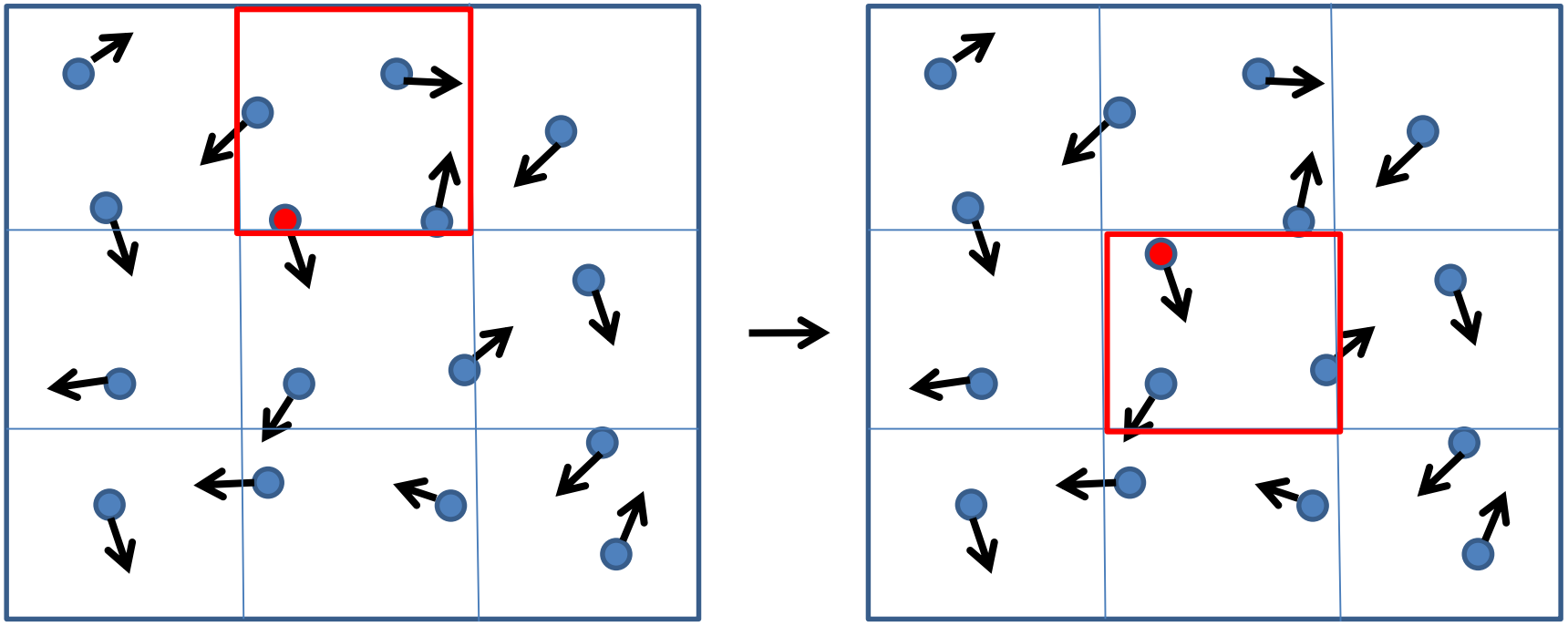
多数の粒子 N 個から構成される系のシミュレーション

各粒子は時間とともに移動する

領域分割による並列化



粒子データの移動



粒子が移動して、担当プロセスが変わる場合がある

元の担当プロセスから新しい担当プロセスへ粒子データを移動させる

片側通信によるデータ移動

ウィンドウオブジェクトの作成、送信バッファへデータをコピー、移動先アドレスの取得、データ移動

片側通信で粒子データを移動してみましょう

シミュレーションのソースコード、および、ジョブスクリプトを以下の方法で取得してください

1. ファイルのコピー
   ```
   cp /home/sim/sim.tgz .
   ```
2. ファイルの展開
   ```
   tar xvf sim.tgz
   ```

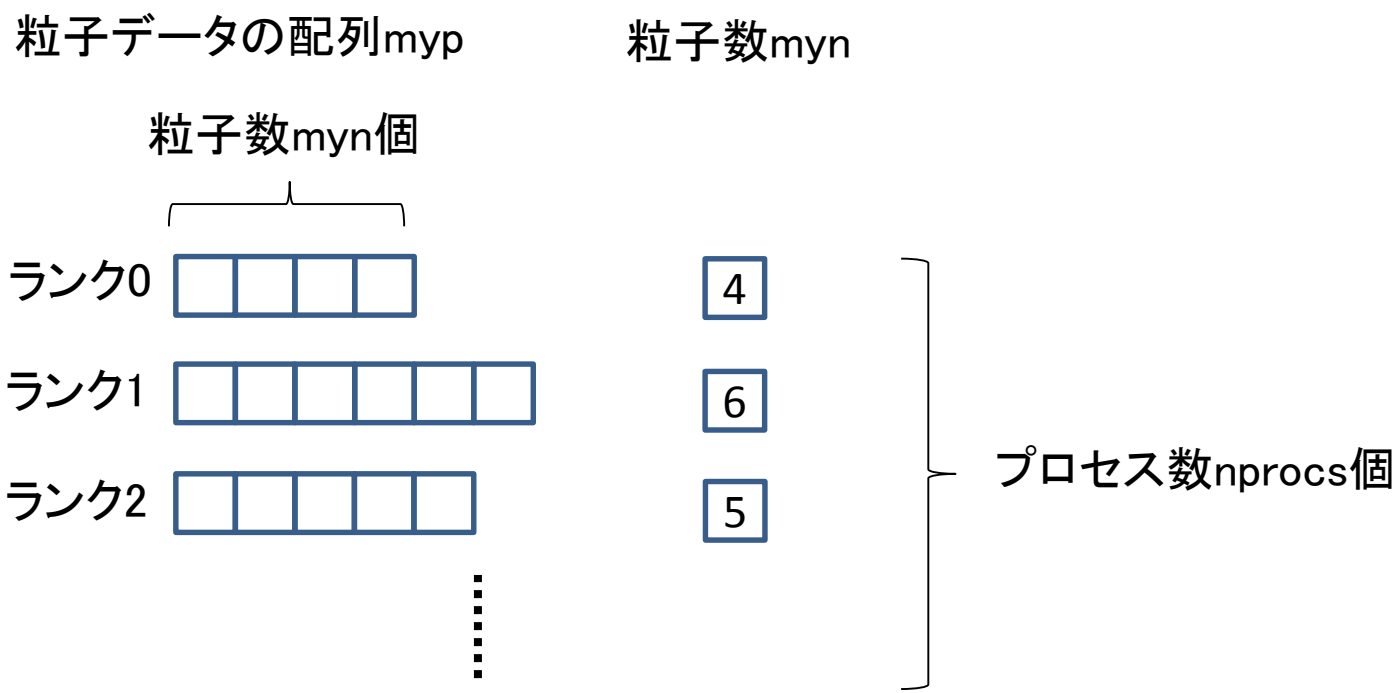
展開されたディレクトリsimには、以下のファイルが含まれています

ファイル名	内容
sim.c, sim.h, sim-mpi.c, win-create.c, one-sided.c, win-create-sample.c, one-sided-sample.c, win-create-tune.c, one-sided-tine.c, p2p.c	シミュレーションのソースコード(後述)
sim, sim-one-sided1, sim-one-sided2, sim-tune1, sim-tune2, sim-p2p	実行オブジェクト
go-sh, go-one-sided1.sh, go-one-sided2.sh, go-tune1.sh, go-tune2.sh, go-p2p.sh	sim*関連のジョブスクリプト
compile.sh	コンパイルスクリプト

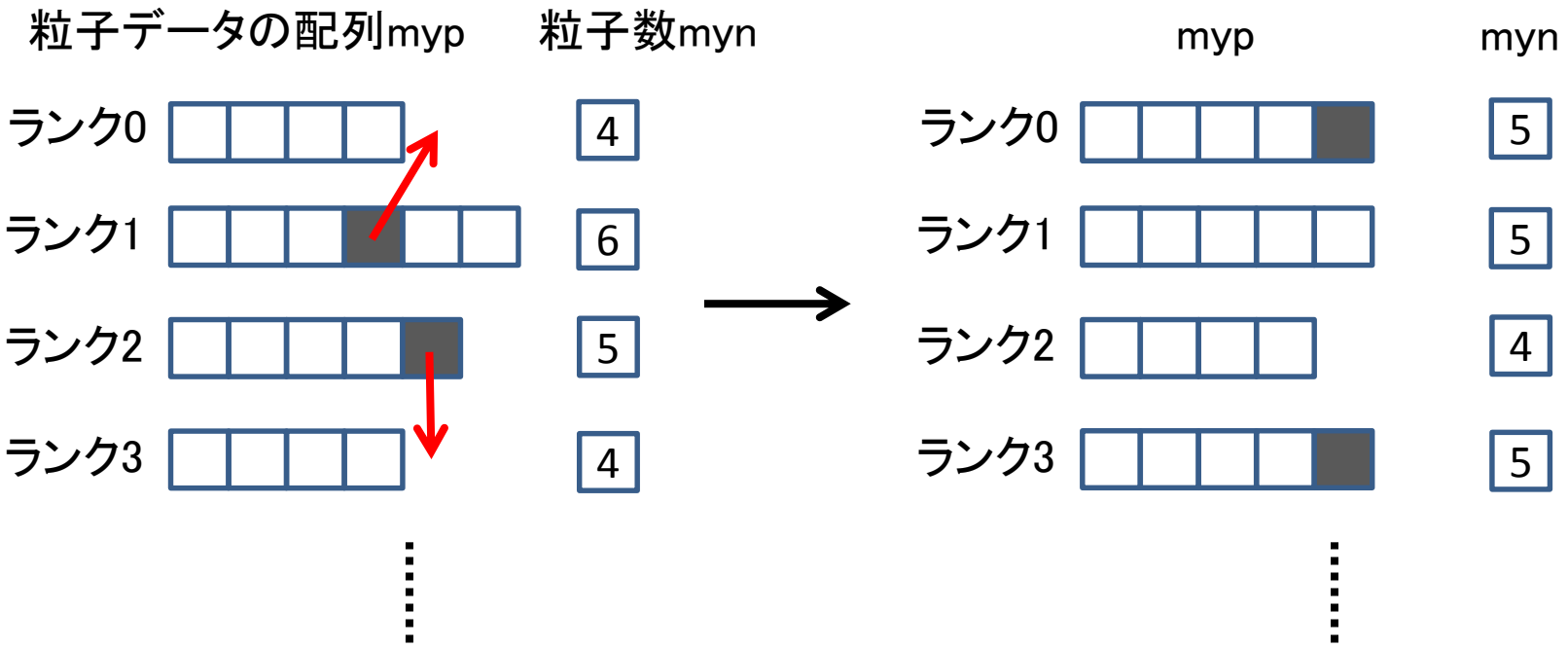
粒子データ

```
struct particle{
    double r[3];
    double v[3];
};

struct particle myp[NMAX];
int myn;
```



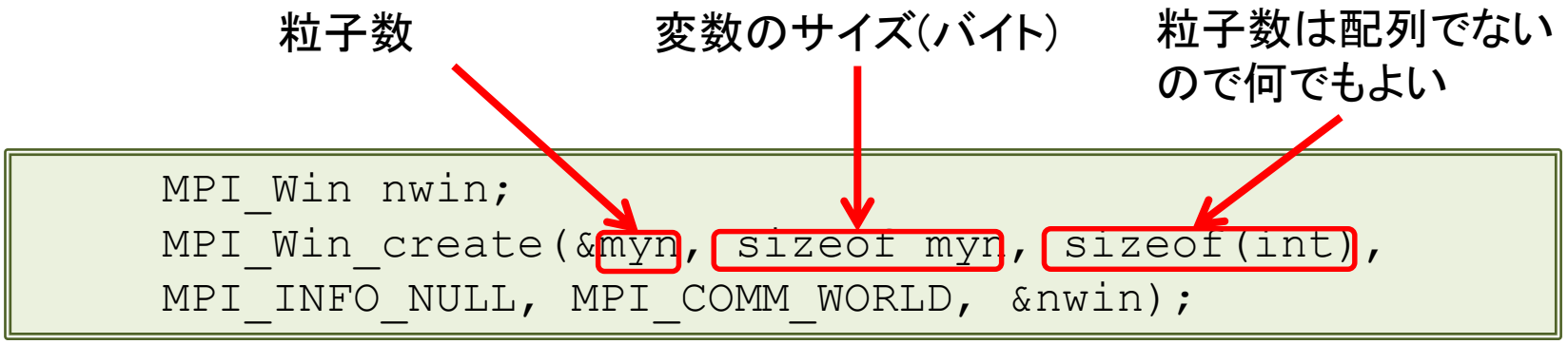
粒子データの移動


粒子データの配列myp
粒子数myn
ランク0
ランク1
ランク2
ランク3
4
6
5
4
myp
myn
ランク0
ランク1
ランク2
ランク3
5
5
4
5

粒子データの移動の手順

- ウィンドウオブジェクトの作成
- 移動する粒子データを送信バッファへコピー
- 移動先アドレスの取得&粒子数の更新
- 粒子データの移動

粒子数に対するウィンドウオブジェクトの作成

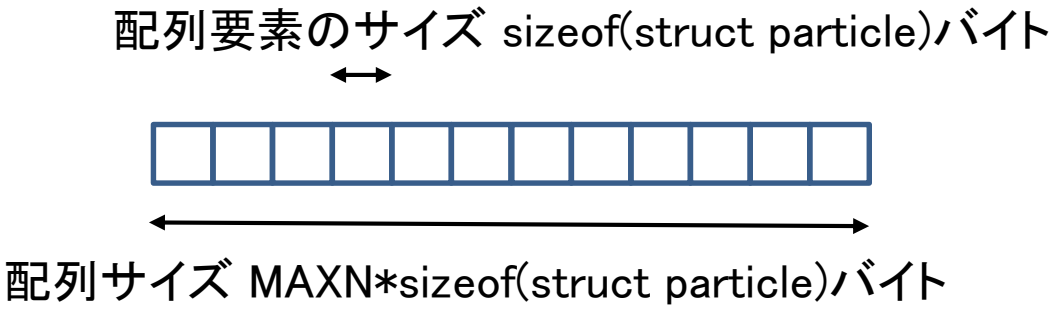


ウィンドウオブジェクトはシミュレーションの最初に1回だけ作成すればよい

粒子データに対するウィンドウオブジェクトの作成

```
MPI_Win pwin;
MPI_Win_create(???, ???, ???,
               MPI_INFO_NULL, MPI_COMM_WORLD, &pwin);
```

粒子データの配列myp



win-create.cを編集

コンパイル `./compile.sh`

ジョブ投入 `pjsub go-one-sided1.sh`

実行結果の確認 `cat one-sided1.out`

実行結果(one-sided1.out)

粒子数が変わって
いないことを確認

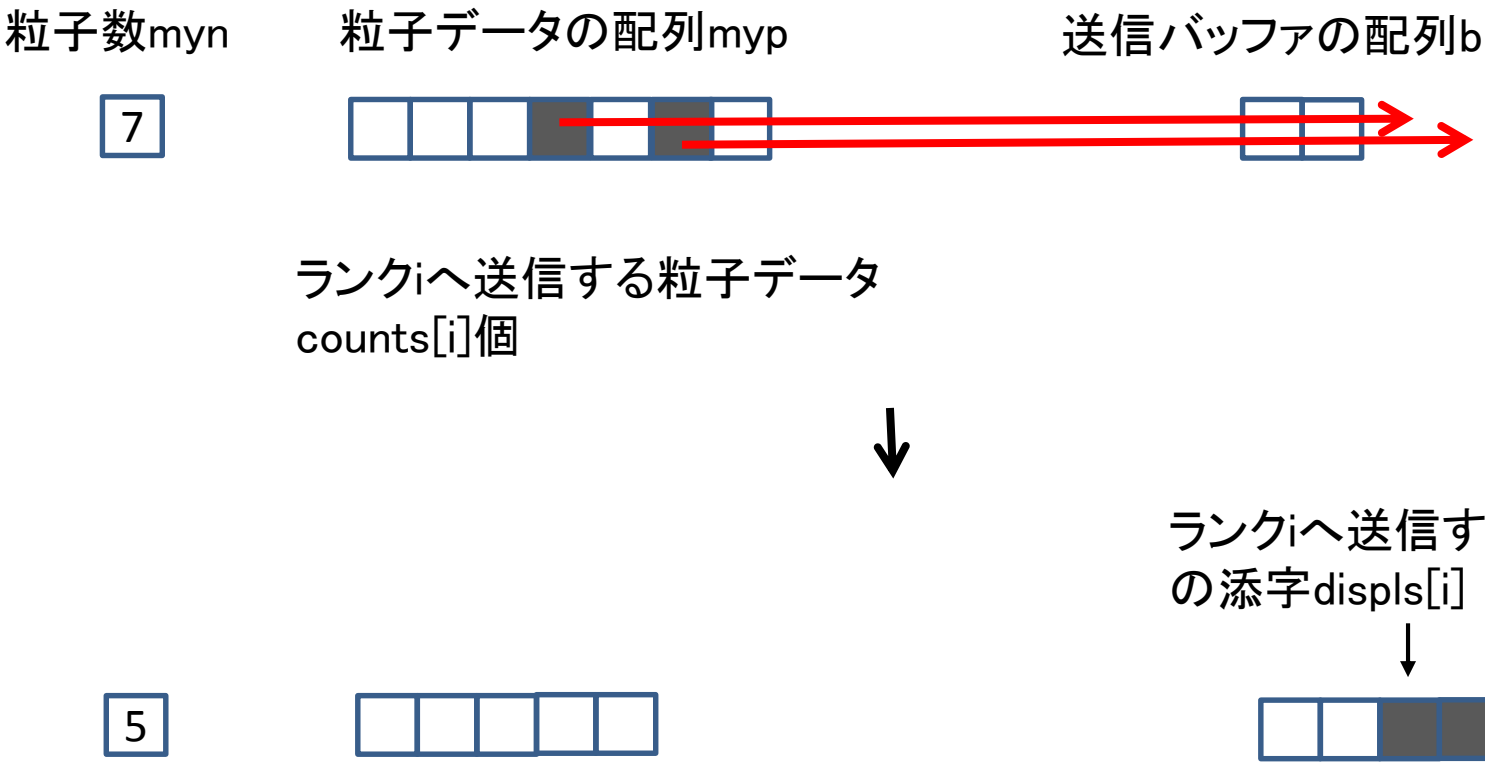
```
Time 0.000000: # of particles 10000  Total energy = 20187.665287
Time 0.001000: # of particles 10000  Total energy = 20187.838167
Move 0.000098 seconds
```

1ステップあたり粒子移動に要した時間

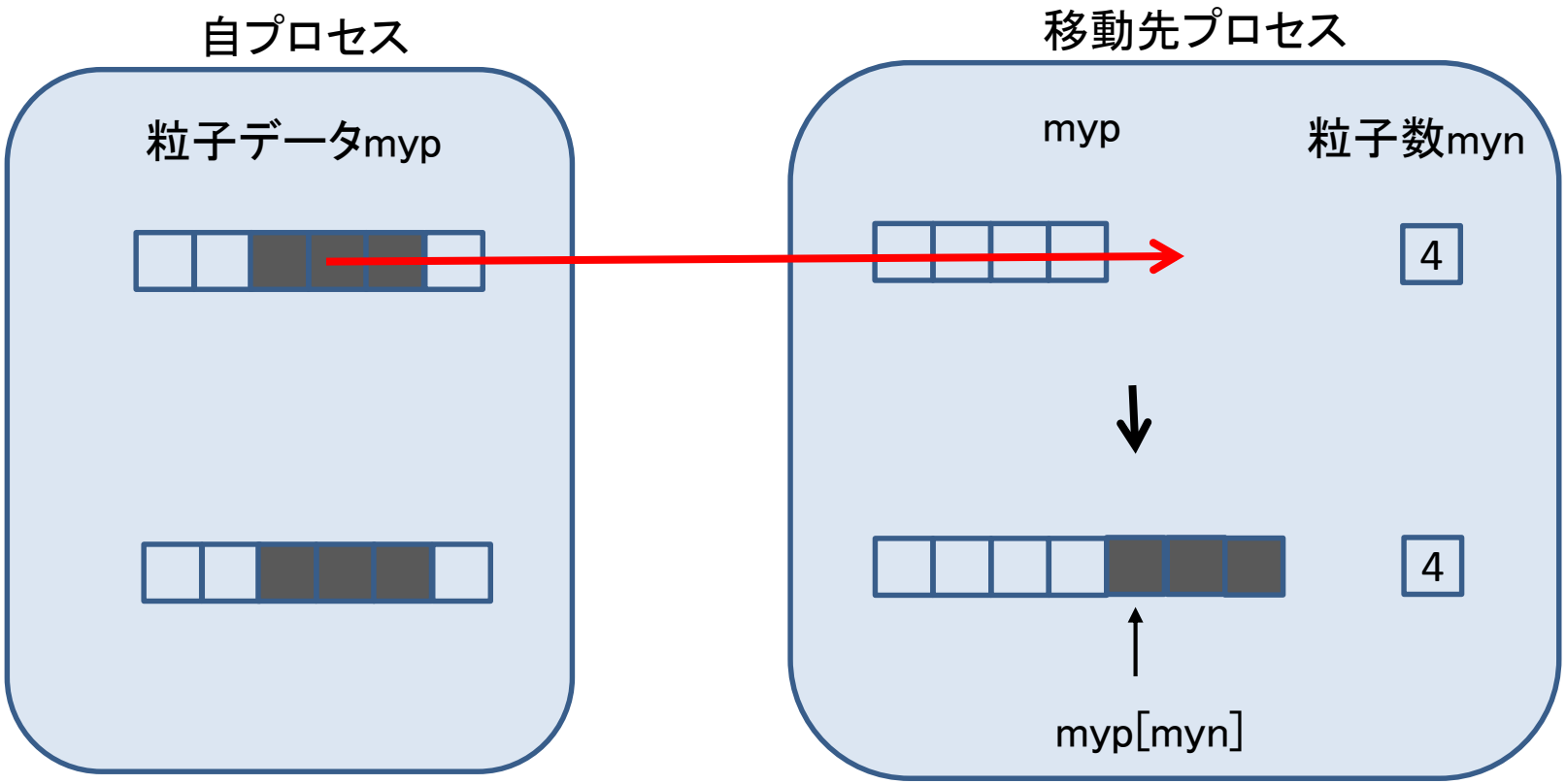
粒子系のエネルギーが
ほとんど変わっていない
ことを確認

送信する粒子データを送信バッファへコピー

通常、通信できるのは配列の連続領域なので、
同じプロセスへ送る粒子データをまとめて送信バッファへコピー



移動先アドレス



移動先アドレスは移動先プロセスの粒子数から求められる

MPI_Get_accumulate関数

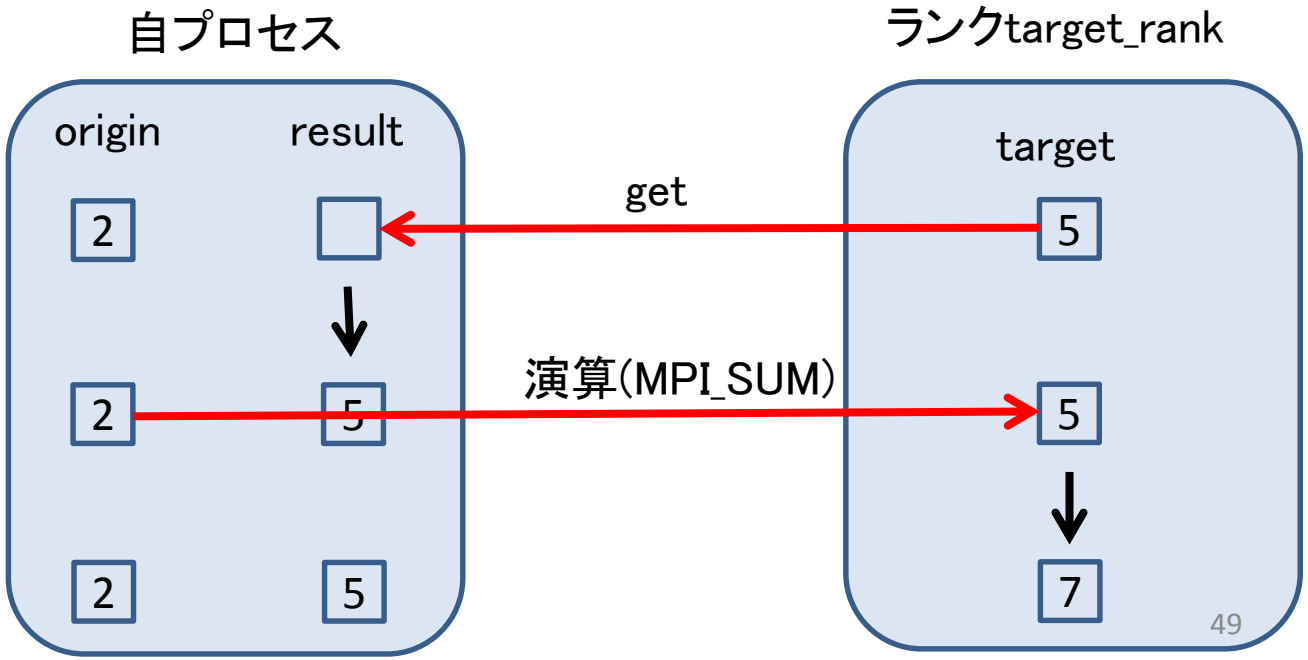
MPI_Getと演算を行う関数

MPI_Getと同じ

```
MPI_Get_accumulate(
    origin_addr, origin_count, origin_datatype,
    result_addr, result_count, result_datatype,
    target_rank, target_disp, target_count, target_datatype,
    op, win);
```

op	説明
MPI_SUM	和
MPI_MAX	最大値
MPI_MIN	最小値

この他、MPI_Reduceと同じ演算が指定可能



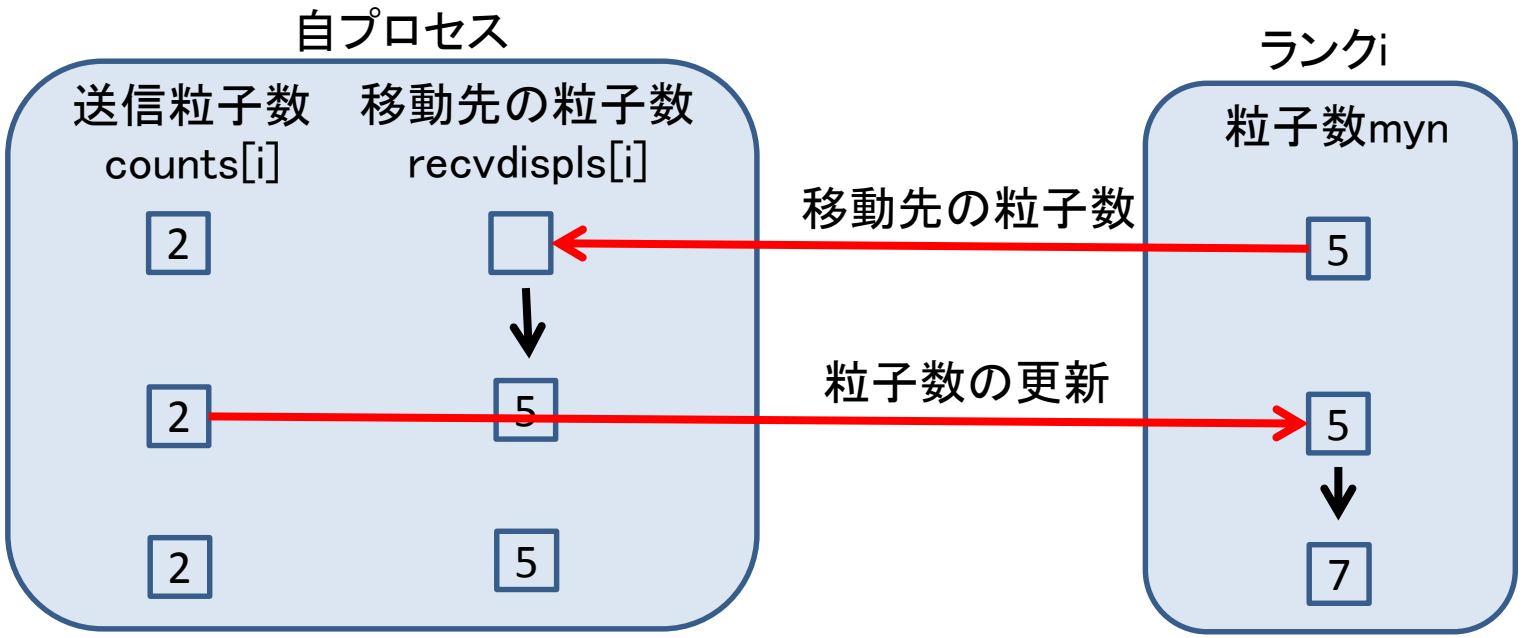
移動先の粒子数取得&粒子数の更新

```
MPI_Win_fence(0, nwin);
for(i = 0; i < nprocs; i ++){
     if(counts[i] > 0){
          MPI_Get_accumulate(&counts[i], 1, MPI_INT,
                             &recvdispls[i], 1, MPI_INT,
                             i, 0, 1, MPI_INT
                             MPI_SUM, nwin);
     }
}
MPI_Win_fence(0, nwin);
```

送信粒子数を指定

移動先の粒子数を指定

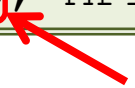
ここに移動先の粒子数を格納



構造体の配列の通信

整数の配列を通信する場合

```
int a[100];
MPI_...(a, 100, MPI_INT,...);
```



要素数

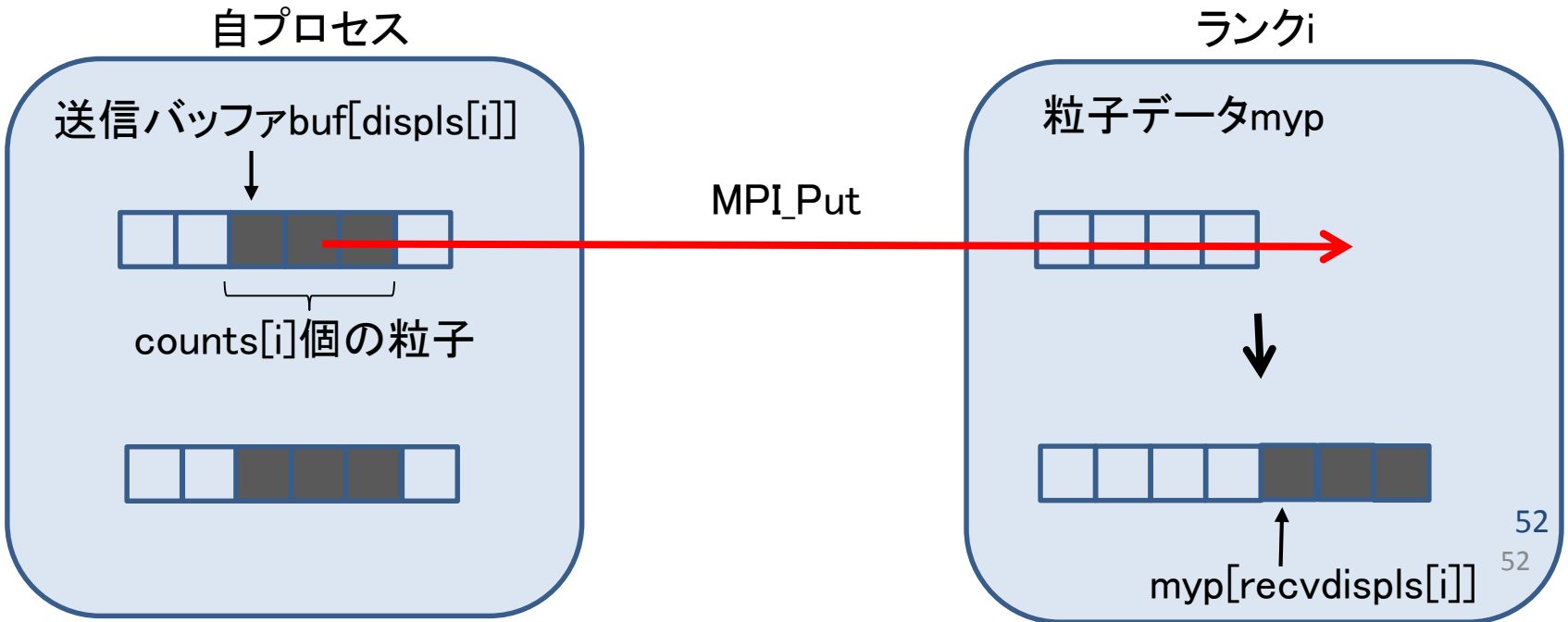
構造体の配列を通信する場合

```
struct xyz a[100];
MPI_...(a, 100 * sizeof(struct xyz), MPI_BYTE,...);
```

配列サイズ(バイト)

粒子データの移動

```
MPI_Win_fence(0, pwin);
for(i = 0; i < nprocs; i ++){
    if(counts[i] > 0){
        MPI_Put(???, ???, ???,
                ???, ???, ???, ???,
                pwin);
    }
}
MPI_Win_fence(0, pwin);
```



52

粒子データの移動

```
MPI_Win_fence(0, pwin);
for(i = 0; i < nprocs; i ++){
     if(counts[i] > 0){
          MPI_Put(???, ???, ???,
                  ???, ???, ???, ???,
                  pwin);
     }
}
MPI_Win_fence(0, pwin);
```

one-sided.cを編集

コンパイル `./compile.sh`

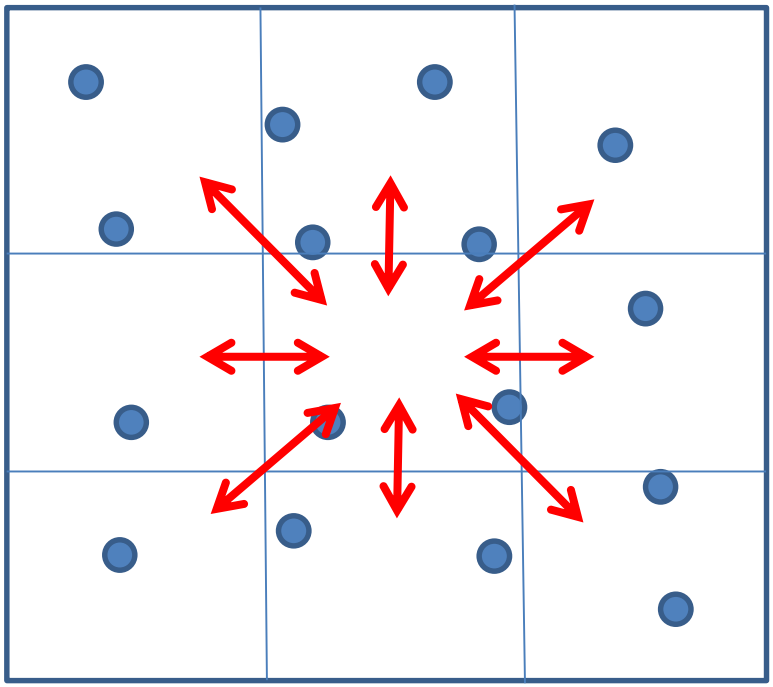
ジョブ投入 `pjsub go-one-sided2.sh`

実行結果の確認 `cat one-sided2.out`

Send/Recv型通信を用いた場合

すべての隣接プロセスと通信

どの隣接プロセスから粒子データが送信されてくるかわからなので、
すべての隣接プロセスと通信します
移動する粒子データがない場合は0バイトのデータを通信します



隣接プロセスの数だけ通信が必要

3x3x3 − 1 = 26回

ジョブ投入 `pjsub go-p2p.sh`

実行結果の確認 `cat p2p.out`

片側通信のチューニング

Infoへキーと値を設定

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, key, value);
MPI_Win_create(base, size, disp_unit, info, comm, &win)
```

キー	値	説明
“no_locks”	”true”	このウィンドウオブジェクトに対してMPI_Win_lockを用いないことを宣言
“accumulate_ordering”	“none”	同じtargetに対する演算はMPIライブラリが実行順序を入れ替えてもよいことを宣言
“accumulate_ops”	“same_op”	同じtargetに対する演算はすべて同じであることを宣言

MPI_Win_createに対する指定（win-create-tune.c）

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "no_locks", "true");
MPI_Info_set(info, "accumulate_ordering", "none");
MPI_Info_set(info, "accumulate_ops", "same_op");
MPI_Win_create(base, size, disp_unit, info, comm, &win)
```

ジョブ投入　`pjsub go-tune1.sh`

実行結果の確認　`cat tune1.out`

実際に効果があるかどうかはプログラムと実行環境に依存します

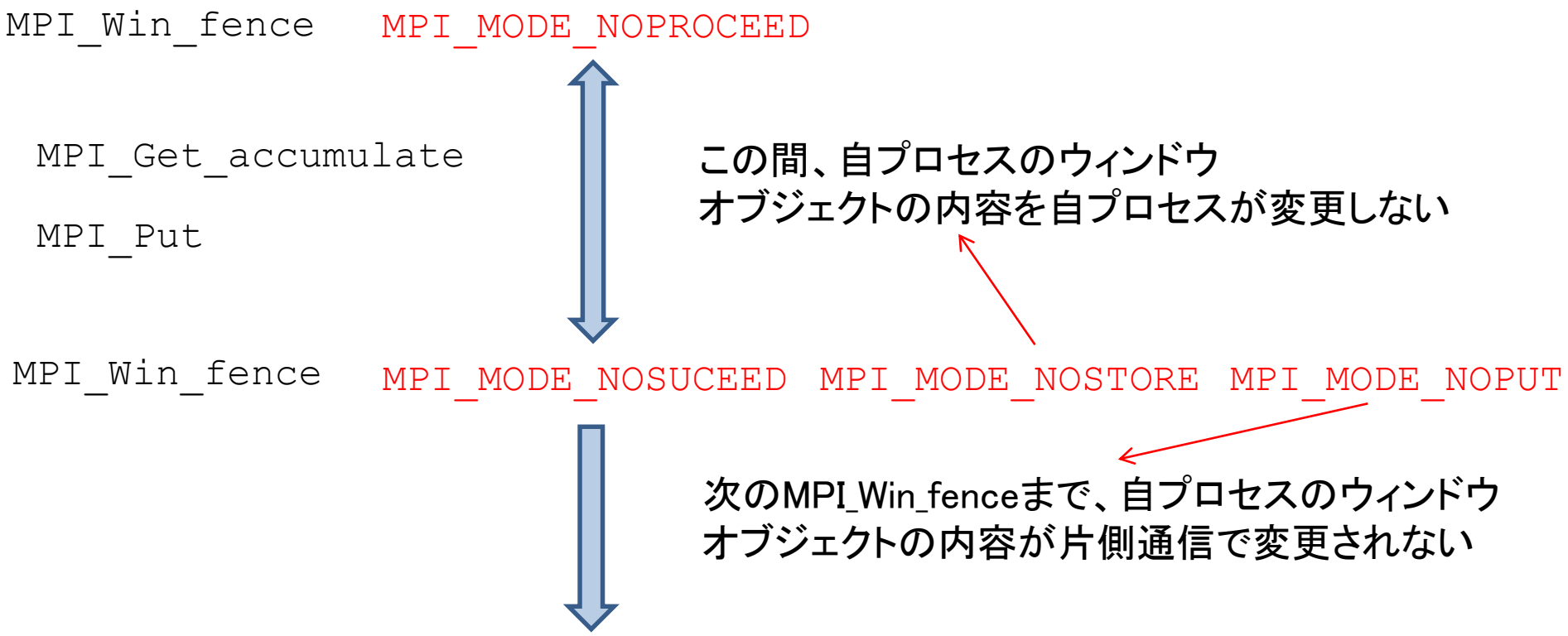
MPI_Win_fenceに対する指定

```
MPI_Win_fence(assert, win)
```

assert	説明
0	何も指定しない
MPI_MODE_NOPRECEDE	このMPI_Win_fenceから片側通信を開始
MPI_MODE_NOSUCEEDE	このMPI_Win_fenceは片側通信を完了
MPI_MODE_NOSTORE	このMPI_Win_fenceで終了する片側通信区間ではこのプロセスは自分のウィンドウオブジェクトを変更していない
MPI_MODE_NOPUT	このMPI_Win_fenceで終了する片側通信区間以降、次の片側通信区間まで、他のプロセスが片側通信でこのプロセスのウィンドウオブジェクトの内容を変更しない

複数の値を指定するときはMPI_MODE_NOSTORE | MPI_MODE_NOPUTのように縦線(ビットOR演算子)でつなぎます

assertに指定できる値



MPI_Win_fenceに対する指定（one-sided-tune.c）

```
    MPI_Win_fence(MPI_MODE_NOPRECEDE, nwin);
    for(i = 0; i < nprocs; i ++){
        if(counts[i] > 0){
            MPI_Get_accumulate(...);
        }
    }
    MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT |
                  MPI_MODE_NOSUCCEED, nwin);
```

MPI_Put前後のMPI_Win_fenceも同様

ジョブ投入　`pjsub go-tune2.sh`

実行結果の確認　`cat tune2.out`

実際に効果があるかどうかはプログラムと実行環境に依存します

MPI規格のバージョン

MPI規格のバージョン

MPIライブラリ	最新版	バージョン
OpenMPI	2.0.0	MPI-3
MPICH	3.2	MPI-3

2016年7月現在

システム	言語処理系	バージョン
高性能演算サーバ tatara PRIMERGY CX400	富士通コンパイラ	MPI-3
	Intel コンパイラ 2016.2.062	MPI-3
	Intel コンパイラ 2013.1.046	MPI-2
	Intel コンパイラ 2012.0.032	MPI-2
スーパーコンピュータ hayaka PRIMEHPC FX10	富士通コンパイラ	MPI-2
高性能アプリケーションサーバ hakozaki HA8000-tc/HT210	Intel コンパイラ	MPI-3
高性能アプリケーションサーバ mikasa SR16000 VM1	IBM XL コンパイラ	MPI-2

MPI-3: 今回の片側通信によるシミュレーションが実行できるバージョン

ご静聴ありがとうございました。