並列プログラミング
MPI入門

並列分散処理　講義ノート2

甲斐宗徳

MPI ＝ Message Passing Interface

- メッセージを受け渡すインタフェース
- MPIは言語ではなくて、C言語などにメッセージ送受信機能を提供するライブラリ

example.c

```
#include <stdio.h>

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

  printf("Hello, %s!!¥n“, argv[1]);

  return 0;
}
```

コンパイル　$ gcc example.c –o example
実行　　$ ./example kai
実行結果　Hello, kai!!

MPIを使ってみると

mpi-ex01.c

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

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  printf("Hello, %s!!¥n“, argv[1]);

  MPI_Finalize();
  return 0;
}
```

MPIプログラムの作成と実行まで

例：mpi-ex01.c を下記の内容とする。

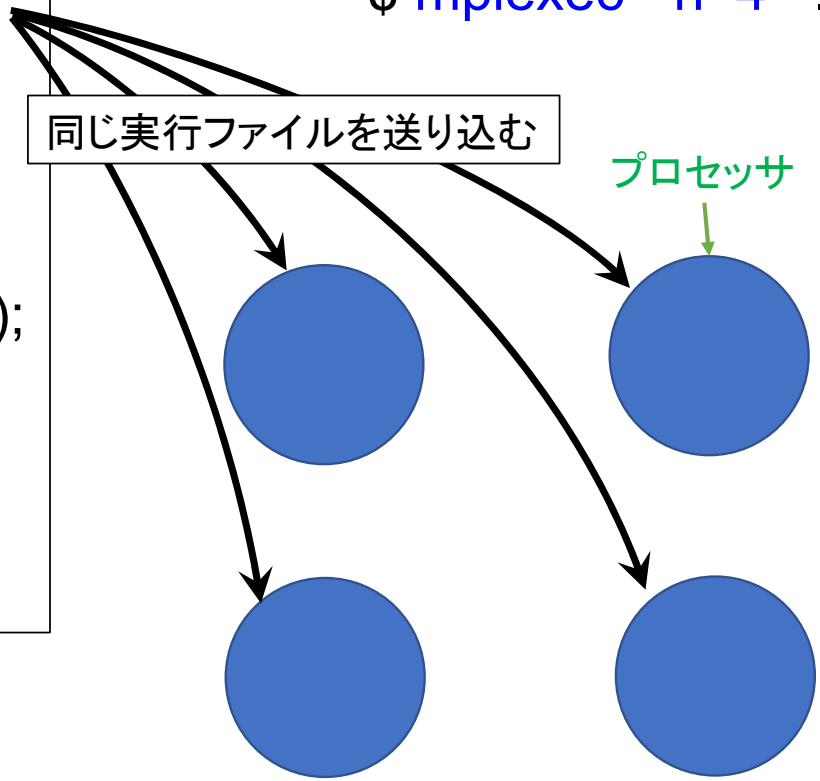
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  printf("Hello, %s!!¥n“, argv[1]);

  MPI_Finalize();
  return 0;
}
```

MPIプログラムのコンパイル方法

```
$ mpicc mpi-ex01.c  -o  mpi-ex01
```

実行の仕方

```
$ mpiexec  -n  4   ./mpi-ex01  kai
```

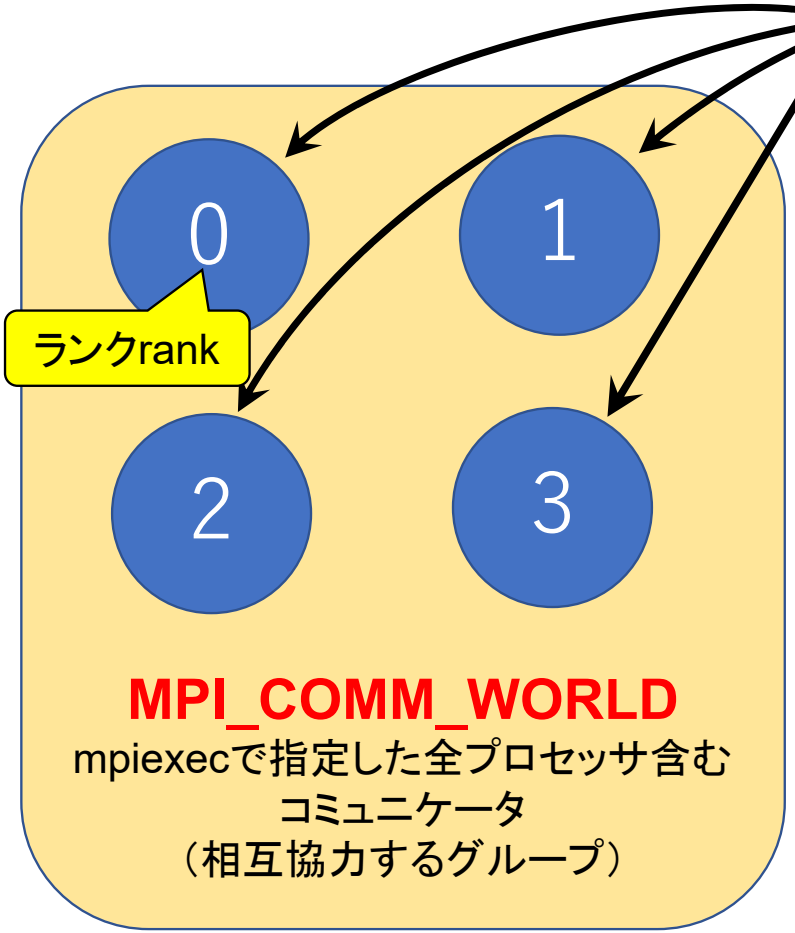


実行結果

```
Hello, kai!!
Hello, kai!!
Hello, kai!!
Hello, kai!!
```

MPI_Init()の働き

① MPI_COMM_WORLDの生成
② ランク（0～プロセッサ数-1）の割当



例：mpi-ex01.c

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

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  printf("Hello!!!¥n");

  MPI_Finalize();
  return 0;
}
```

以降のMPI関数を使うためのヘッダファイル

argc: コマンドライン上の引数の個数
argv: 各文字列の先頭アドレス用配列

MPIの初期化（MPI_COMM_WORLDの生成）

MPI_Init()によって全ランクが
MPI_COMM_WORLDというコミュニケータを
知ることとなる。

MPIの終了処理（MPI_COMM_WORLDの解散）

```
mpiexec -n 4 ./mpi-ex01 kai
```

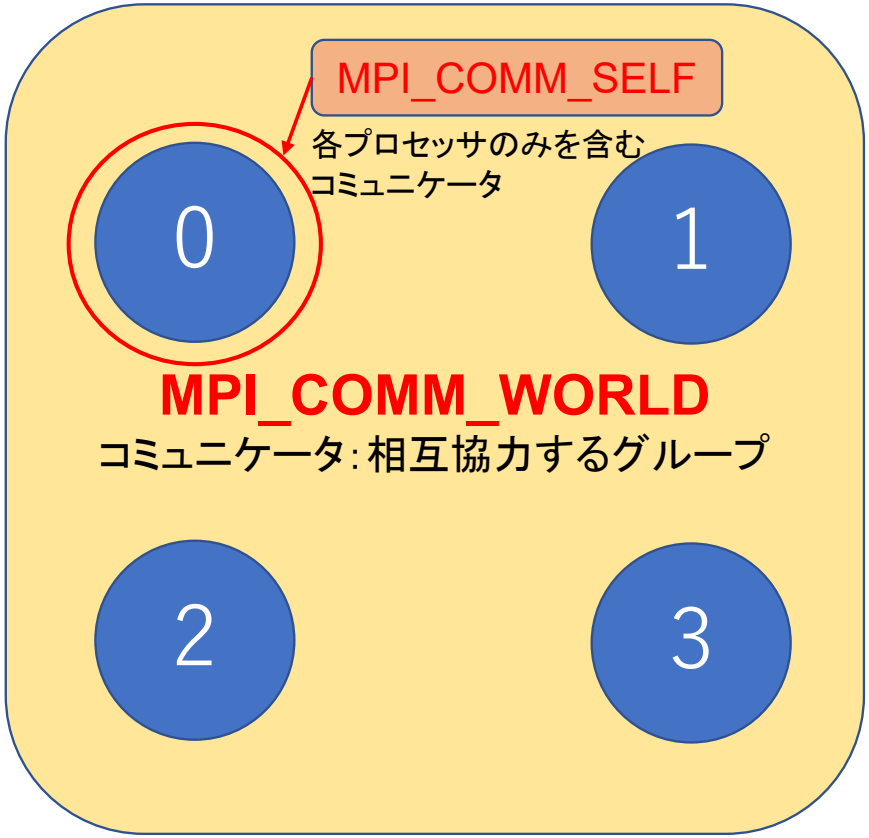
argc=5
argv[0] → “mpiexec”
argv[1] → “-n”
argv[2] → “4”
argv[3] → “./mpi-ex01”
argv[4] → “kai”

コマンドライン引数を逐次実行の
場合と揃えるための調整を行う

```
./mpi-ex01 kai
```

argc=2
argv[0]→”./mpi-ex01”
argv[1]→”kai”

コミュニケータのサイズと自身のランクを知る

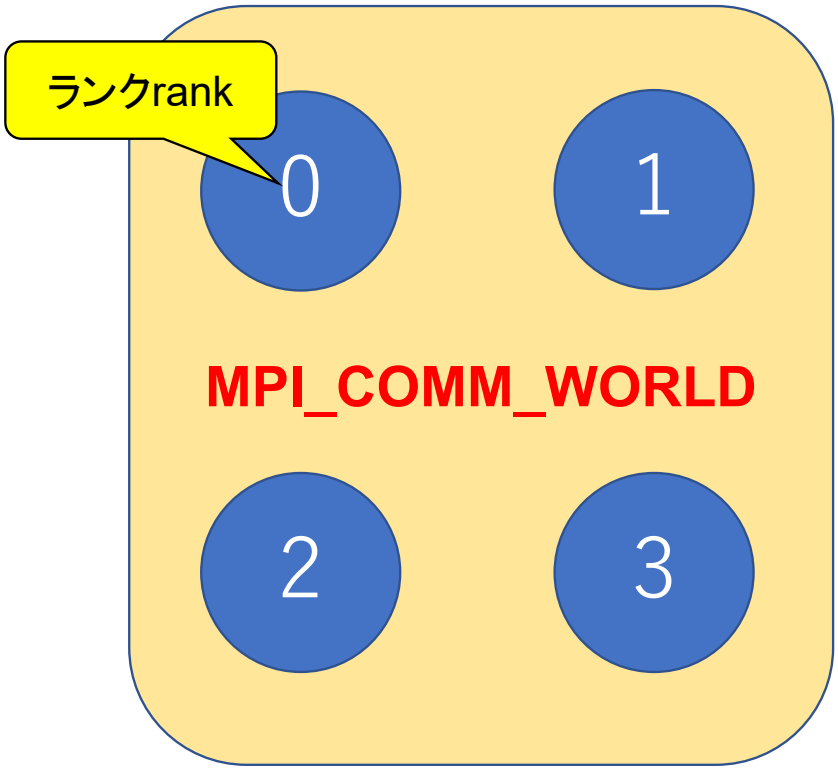



```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf("Hello!!!¥n");

  MPI_Finalize();
  return 0;
}
```

MPI_Comm_rank(MPI_COMM_WORLD, &rank)
　自分のランクを取得する

MPI_Comm_size(MPI_COMM_WORLD, &size)
　所属するコミュニケータのサイズを取得する

各ランクの処理する内容を区別するには・・・




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

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if(rank%2==0)  // 偶数rankのみが出力
     printf("Hello!! I am No.%d/%d¥n", rank, size);

  MPI_Finalize();
  return 0;
}
```

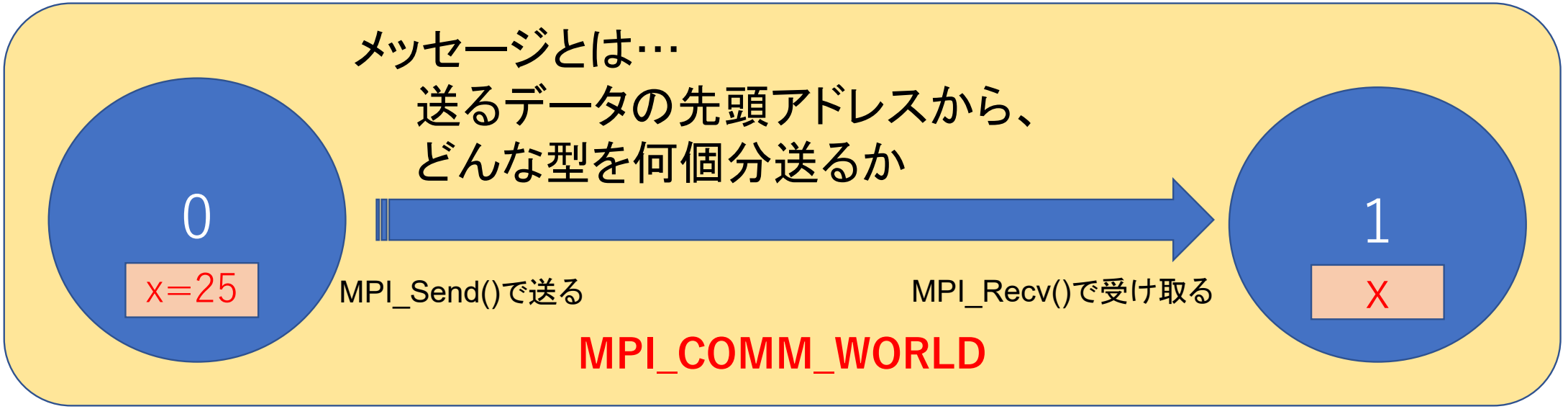
各ランクの処理する内容を区別するには・・・

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

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if(rank==0) // rank=0の処理すること
     printf("Hello!!!¥n");
  else if(rank==1){
       ・・・
  }else if(rank==2){
       ・・・
  }else{
       ・・・
  }

  MPI_Finalize();
  return 0;
}
```

同一コード内に
異なるrankが行う処理を
記述する形で、
並列実行できる

MPIにおける通信の基本①

- MPI_Send( );　// メッセージを送信する
- MPI_Recv( );　// メッセージを受信する



- MPI_Send(&x, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);

先頭アドレス, 個数, 型, 宛先, タグ, コミュニケータ

MPIにおける通信の基本②

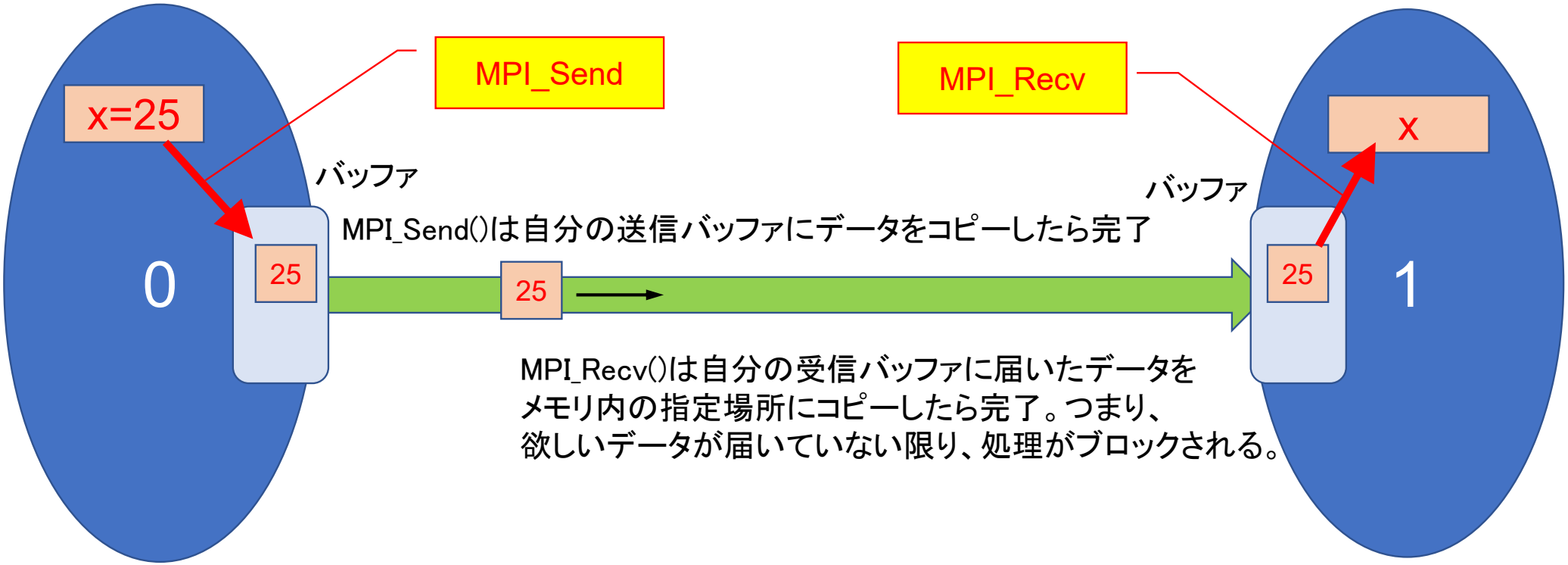
- MPI_Send( );　// メッセージを送信する
- MPI_Recv( );　// メッセージを受信する

MPI_Recv(&x,1,MPI_INT, 0, 10, MPI_COMM_WORLD, &status);

先頭アドレス,個数,型, 送り主, タグ, コミュニケータ, 受信状態

MPI_Recvでは MPI_Status status; が追加される

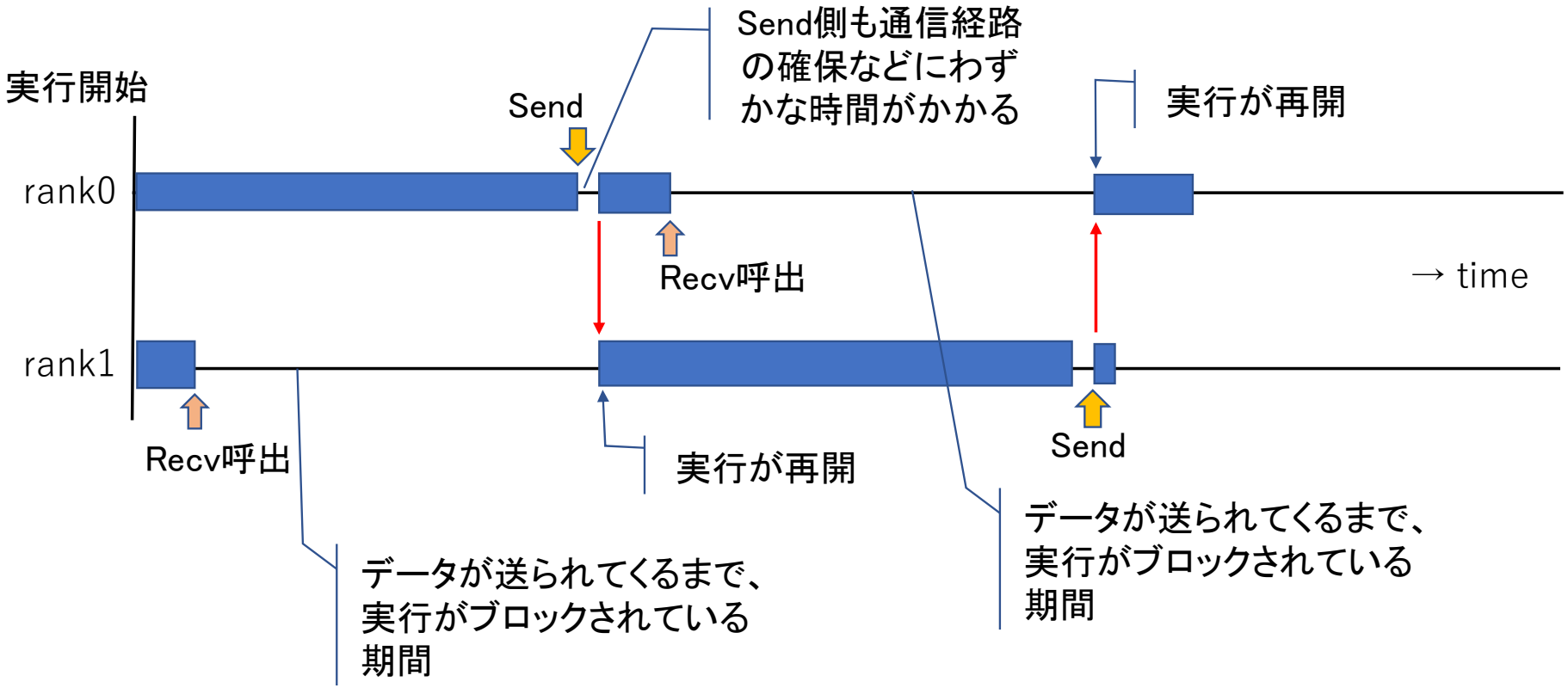
Send／Recvに使用されるバッファについて



- MPI_Send()は、受信側へのデータの到着に関係なく、比較的短時間で呼び出し(call)から戻ることができる。
- 一方、MPI_Recv()は、欲しいデータが送信され、自分のバッファに入ったのち、それを自分の指定のメモリに入るまで待たされる(ブロックされる)ことになる。
- 同期モード(Synchronous)の送信(MPI_Ssend())というのがあり、受信側が受け取るまで送信側もブロックされる。

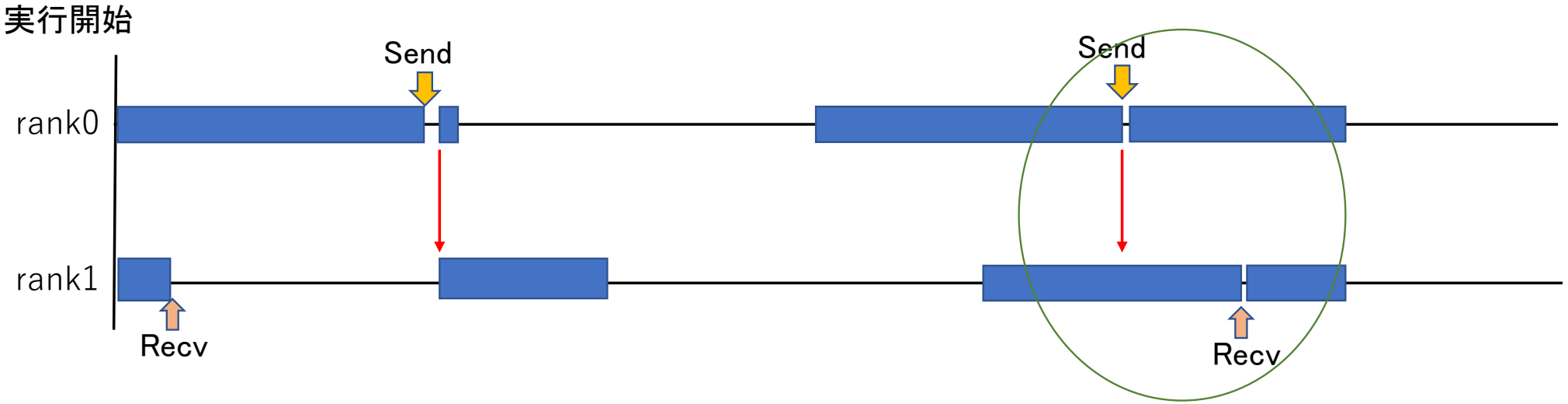
Send／Recvのタイミング①

MPI_Send()とMPI_Recv()は必ずペアになっていなければならない。



上記のような通信のタイミングでは、どちらか一方しか実行していない時間が長い→並列効率小

Send／Recvのタイミング②



相手がSendしたあとにRecvすると、ブロックされてい期間を削減できる
→ 通信と処理がオーバラップできる → 並列効率大

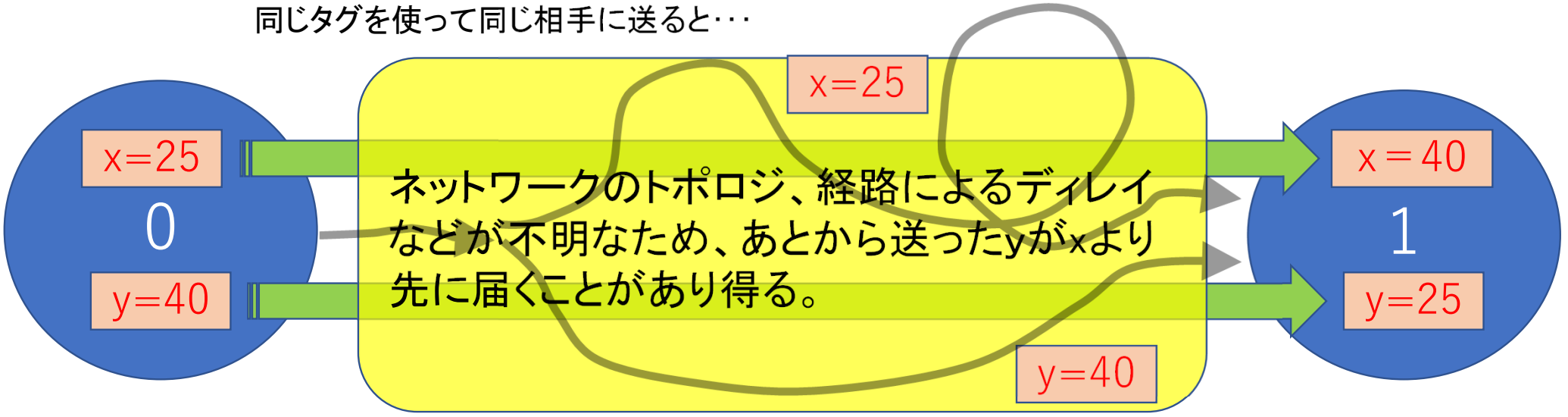
Send／Recvに指定されるタグについて

Rank0は、

```
MPI_Send( &x,1,MPI_INT, 1, 10, MPI_COMM_WORLD );
MPI_Send( &y,1,MPI_INT, 1, 10, MPI_COMM_WORLD );
```

Rank1は、

```
MPI_Recv( &x,1,MPI_INT, 0, 10, MPI_COMM_WORLD, &status );
MPI_Recv( &y,1,MPI_INT, 0, 10, MPI_COMM_WORLD, &status );
```



Send／Recvに指定されるタグについて

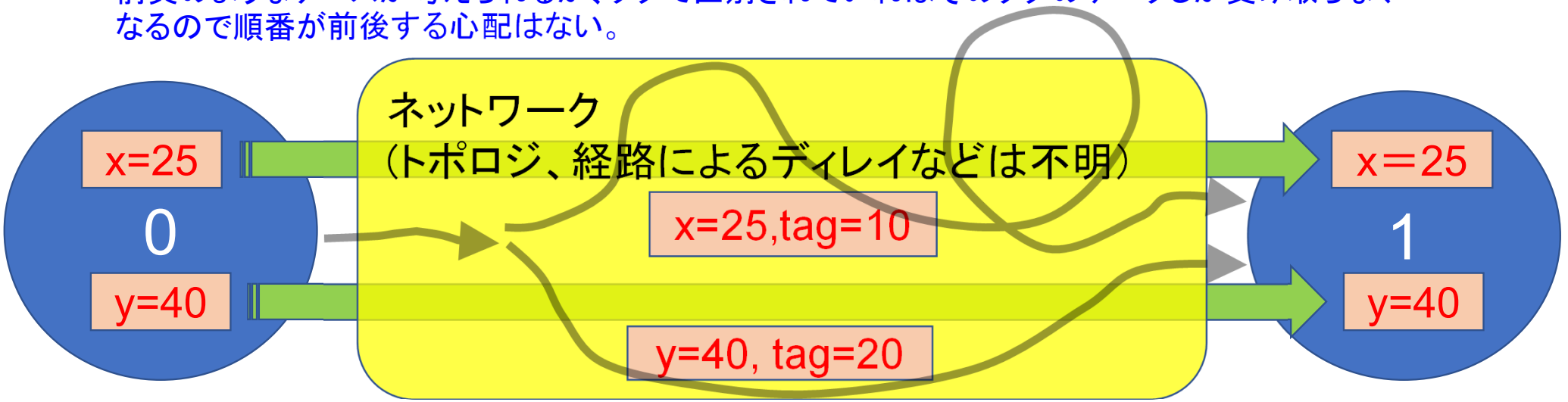
Rank0は、

```
MPI_Send( &x,1,MPI_INT, 1, 10, MPI_COMM_WORLD );
MPI_Send( &y,1,MPI_INT, 1, 20, MPI_COMM_WORLD );
```

Rank1は、

```
MPI_Recv( &x,1,MPI_INT, 0, 10, MPI_COMM_WORLD, &status );
MPI_Recv( &y,1,MPI_INT, 0, 20, MPI_COMM_WORLD, &status );
```

前頁のようなケースが考えられるが、タグで区別されていればそのタグのデータしか受け取らなくなるので順番が前後する心配はない。



MPI_Recvで送信元やタグにワイルドカードを指定可能

Rank0は、

```
MPI_Send( &x,1,MPI_INT, 1, 33, MPI_COMM_WORLD );
```

Rank1は、

```
MPI_Recv( &a,1,MPI_INT, MPI_ANY_SOURCE,  MPI_ANY_TAG,
                                          MPI_COMM_WORLD, &status );
```

Rank1は受信後に、status変数（構造体）のメンバにより、送信元をstatus.MPI_SOURCEで、
タグをstatus.MPI_TAGで知ることができる。

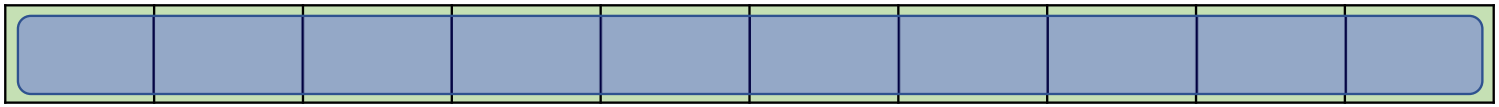
例：

```
MPI_Recv( &a,1,MPI_INT, MPI_ANY_SOURCE,  MPI_ANY_TAG,
                                          MPI_COMM_WORLD, &status );
printf("I received data from %d with tag%d.¥n", status.MPI_SOURCE,
                                                 status.MPI_TAG);
```

配列データの送受信

Rank0で・・・

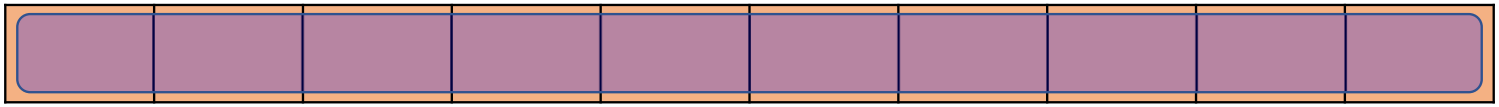
int ar[10];



MPI_Send(ar,10,MPI_INT, 1, 11, MPI_COMM_WORLD);

Rank1で・・・

int br[10];



MPI_Recv(br,10,MPI_INT, 0, 11, MPI_COMM_WORLD, &status);

Rank0で・・・

int ar[10];

MPI_Send(&ar[2],4,MPI_INT, 1, 11, MPI_COMM_WORLD);

Rank1で・・・

int br[10];

MPI_Recv(br,10,MPI_INT, 0, 11, MPI_COMM_WORLD, &status);

送信されるデータより広い領域であれば受け取ることが出来る

配列データの送受信

Rank0で・・・

int ar[10];

MPI_Send(&ar[2],4,MPI_INT, 1, 11, MPI_COMM_WORLD);

Rank1で・・・

int br[10];

MPI_Recv(br,10,MPI_INT, 0, 11, MPI_COMM_WORLD, &status);

送信されるデータより広い領域であれば受け取ることができる

実際に何個のデータを受け取ったか知りたい場合には・・・

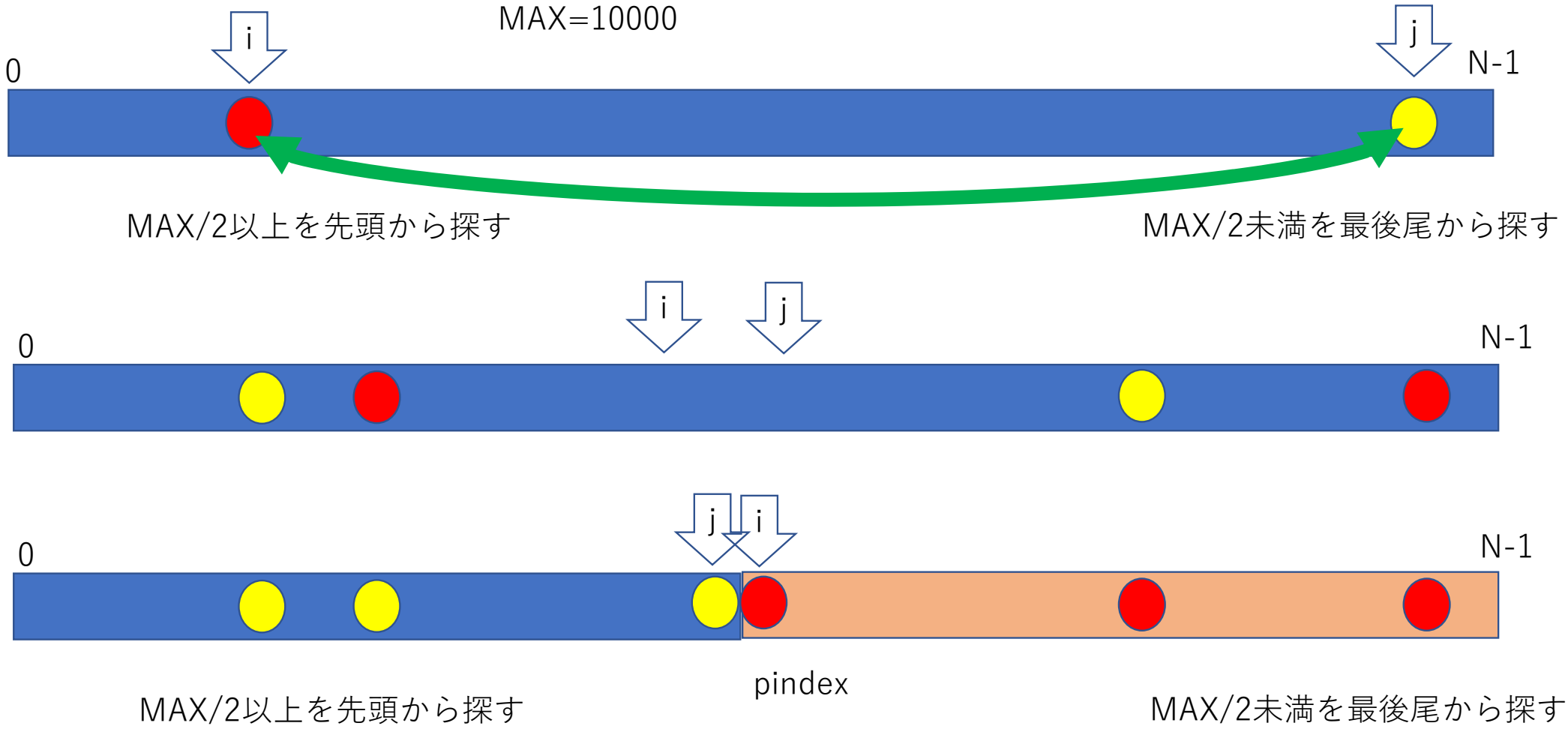
```
int count;
MPI_Get_count(&status, MPI_INT, &count);
```

上の図であれば count に 4 を得ることができる

演習

- int型の1つのデータの送受信を2つのrank間で数回繰り返してみよう。
- int型配列に格納されたデータの合計を、2つのrankで協力して求めよう。
- int型配列に格納されたデータの最大値を、任意個数のrankで協力して求めよう。
- int型配列に格納されたデータを2つのrankで協力してソートしよう。

2つのプロセスで並列ソート



並列プログラムの動作確認に役立つ関数紹介①

■スリープしていることを1秒ごとに表示する関数

```
#include <unistd.h>
void zzz(int sec, int rnk) {  // 休止秒数とランクを受け取る
  for(int i=0; i<sec; i++){
    sleep(1);
    printf("#%d is sleeping...¥n", rnk);
  }
}
```

■乱数を生成する

```
#include <stdlib.h>
int rand(void);   // 0～INT最大値までの整数乱数を発生
void srand(unsigned int seed); // seedを乱数生成の種に設定する
```

サイコロの目を生成するには・・・　rand()%6+1　　これで1～6をランダムに生成

並列プログラムの動作確認に役立つ関数紹介②

■プログラムの指定した2点間の経過時間を求める

```
#include <sys/time.h>
double gettime()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return((double)(tv.tv_sec)+(double)(tv.tv_usec)*1e-6);
}
```

```
double sttime,endtime;
sttime=gettime();
  // 経過時間を計りたいコード部分
endtime=gettime();
printf("Elapsed time:%lfsec¥n",endtime-sttime);
```

■上記で利用するint gettimeofday(struct timeval *tv, struct timezone *tz);

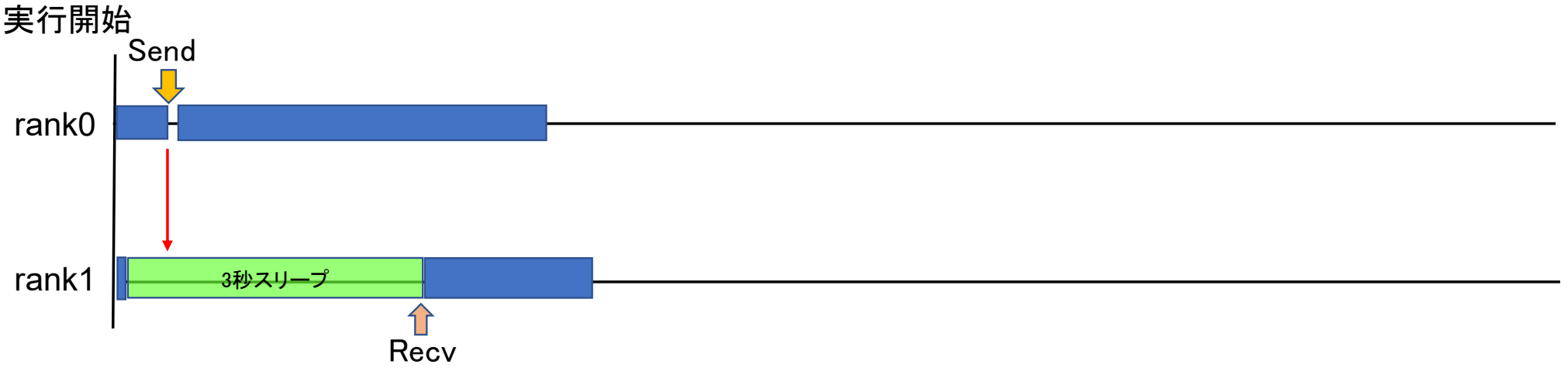
```
struct timeval {
    time_t      tv_sec;     /* 秒 */
    suseconds_t tv_usec;    /* マイクロ秒 */
};
struct timezone {
      int tz_minuteswest;     /* グリニッジ標準時との差 (西方に分単位) */
      int tz_dsttime;         /* 夏時間調整の型 */
};
```

MPI_Send()と同期モードのMPI_Ssend()の比較

普通のMPI_Send()とMPI_Recv()の動作の場合

```
MPI_Send(&x,1,MPI_INT, 1,11,MPI_COMM_WORLD);
printf("Rank0 has sent data.¥n");
```

MPI_Send()はすぐに終わるのですぐ次のprintf()に進む

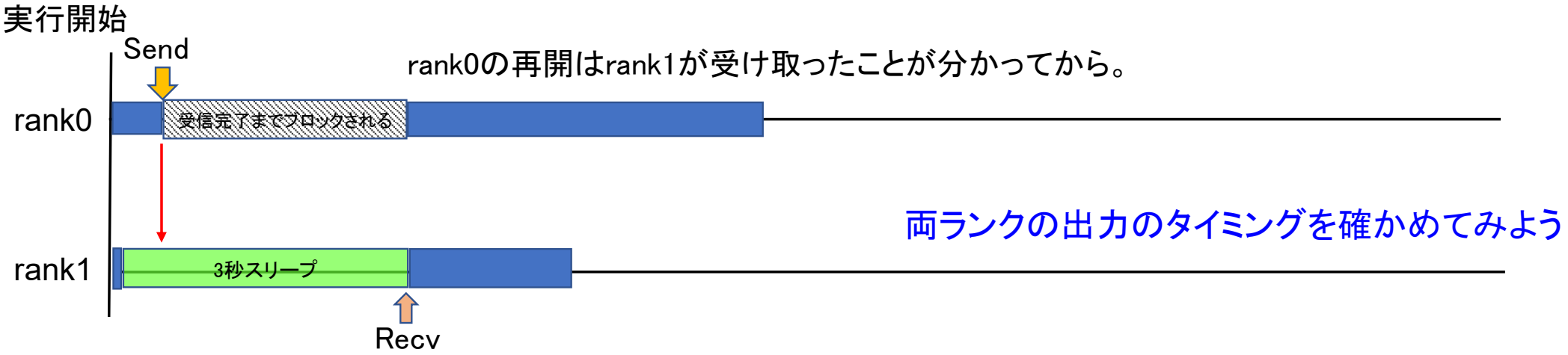



```
sleep(3);   // #include <unistd.h>   が必要
MPI_Recv(&x,1,MPI_INT, 0,11,MPI_COMM_WORLD, &status);
```

MPI_Send()と同期モードのMPI_Ssend()の比較

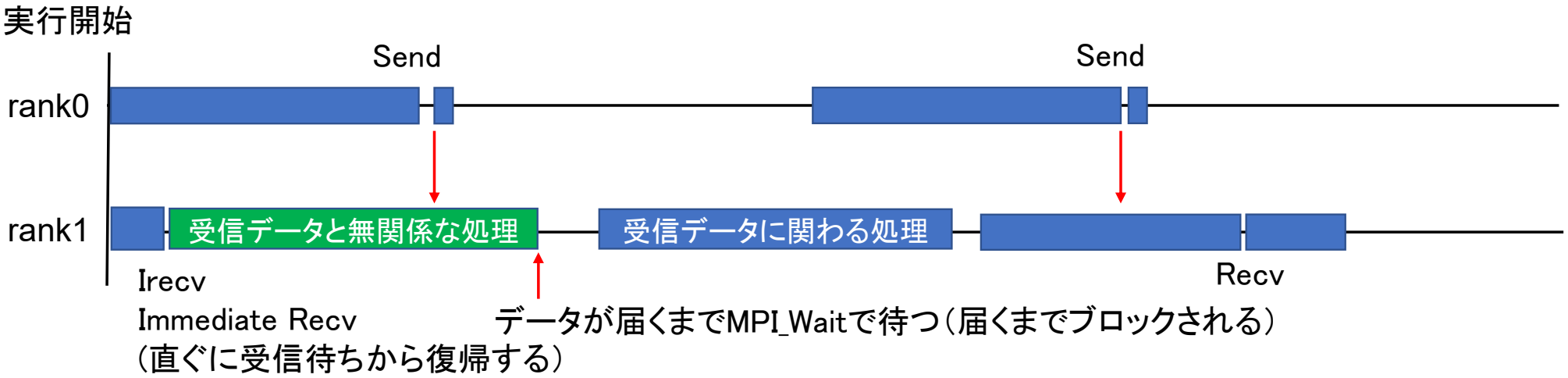
同期モード（Synchronous）の送信（MPI_Ssend()）というのがある。
　受信側が受け取るまで、送信側もブロックされる命令。

```
MPI_Ssend(&x,1,MPI_INT, 0,11,MPI_COMM_WORLD);
printf(“Rank0 has sent data.¥n”);
```




```
sleep(3);   // #include <unistd.h>   が必要
MPI_Recv(&x,1,MPI_INT, 0,11,MPI_COMM_WORLD, &status);
printf(“Rank1 has gotten data.¥n”);
```

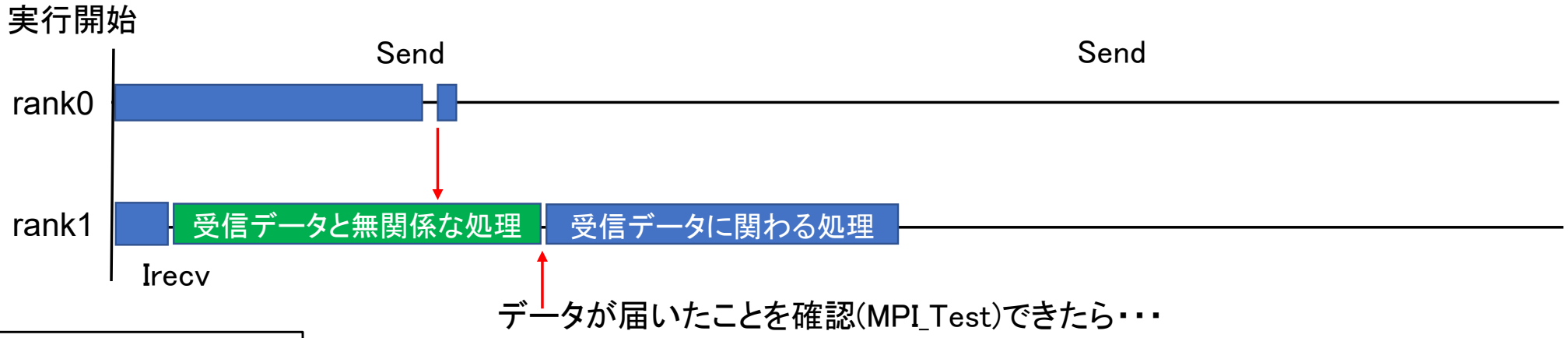
ノンブロッキング通信



MPI_Irecvの使用例1

```
MPI_Status status;  // 受信状態用変数
MPI_Request req;  // ノンブロッキング通信のリクエスト用変数
    ・・・
MPI_Irecv(&x,1,MPI_INT, 0,11,MPI_COMM_WORLD, &req);  // 受信のリクエストを行う
//  データを待つ間に先に処理できる内容をMPI_Irecv()とMPI_Wait()の間に記述する
MPI_Wait(&req, &status);  // 出しておいたリクエストを待つ。届くとstatusに受信情報が入る。
//  受信したデータの処理を記述
```

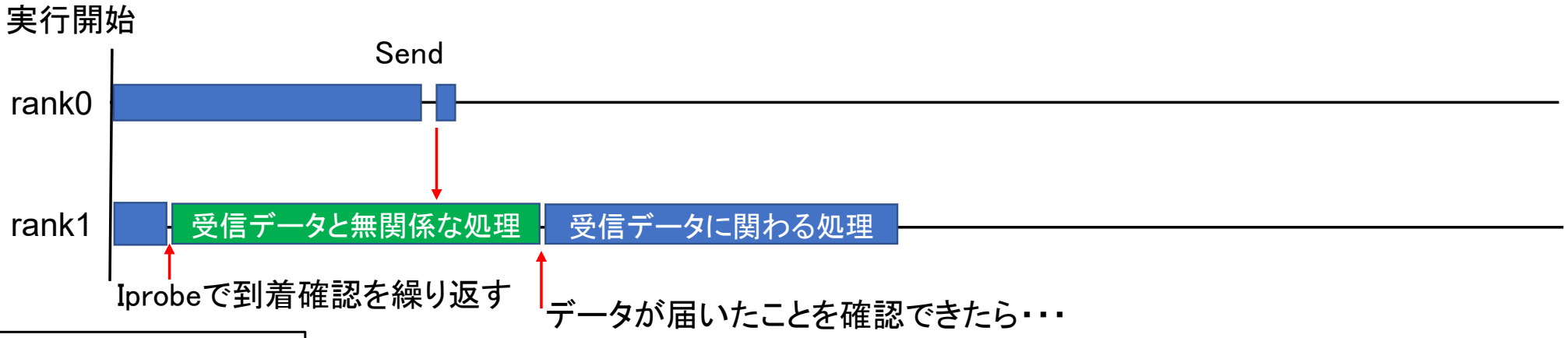
ノンブロッキング通信



MPI_Irecvの使用例2

```
MPI_Status status;  // 受信状態用変数
MPI_Request req;  // ノンブロッキング通信のリクエスト用変数
    ・・・
int flag;
MPI_Irecv(&x,1,MPI_INT, 0,11,MPI_COMM_WORLD, &req);  // 受信のリクエストを行う
do{
   //  データを待つ間に処理できる繰り返し内容を記述する
   MPI_Test(&req, &flag, &status);  // 1回のイタレーションごとに受信データの到着を確認する
}while(!flag);  // MPI_Testの結果、flagが真になるとdoループを抜けられる
// 受信データの処理を記述
```

ノンブロッキング通信



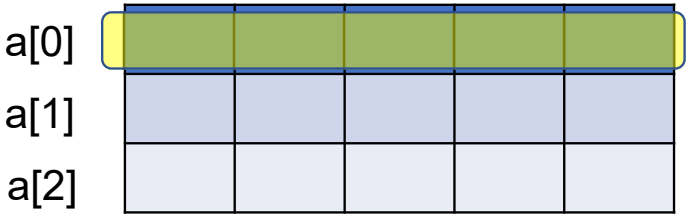
MPI_Iprobeの使用例

```
MPI_Status status;  // 受信状態用変数
MPI_Request req;  // ノンブロッキング通信のリクエスト用変数
    ・・・
int flag;
while(MPI_Iprobe(0,1,MPI_COMM_WORLD, &flag, &status), !flag){   // 到着済みかを確認
    //  データを待つ間に処理できる繰り返し内容を記述する
}
MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD, &status);  // 到着確認済みのデータを受信
// 受信データの処理を記述
```

MPI_Iprobe自体は受信は行わず（送り主とタグを指定）、データの到着を確認するだけ

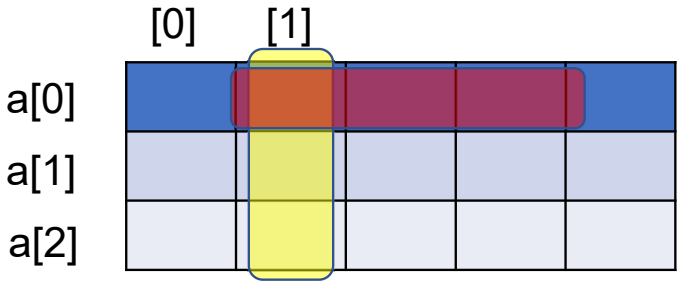
ストライド・ベクトル・データタイプ

int a[3][5];



1行目を送信したい時
MPI_Send(a[0],5,MPI_INT,…);

全行を送信したい時
MPI_Send(a[0],15,MPI_INT,…);
配列はメモリ内で行優先で一次元化されるから

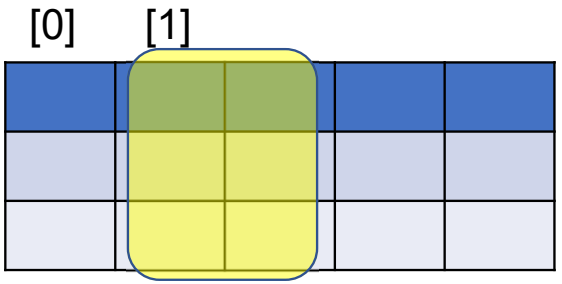


では、列方向のデータを送るにはどうするか？

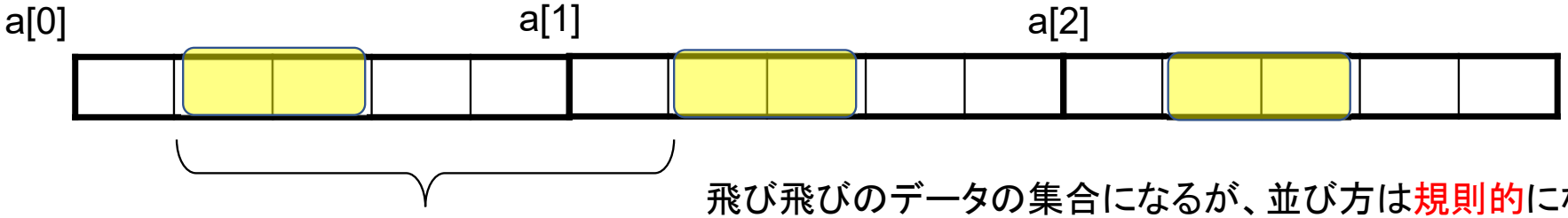
MPI_Send(&a[0][1],3,MPI_INT,　これはできない
（図の赤い部分の送信になってしまう）

ストライド・ベクトル・データタイプ

int a[3][5];



では、列方向のデータを送るにはどうするか？



飛び飛びのデータの集合になるが、並び方は規則的になっている。

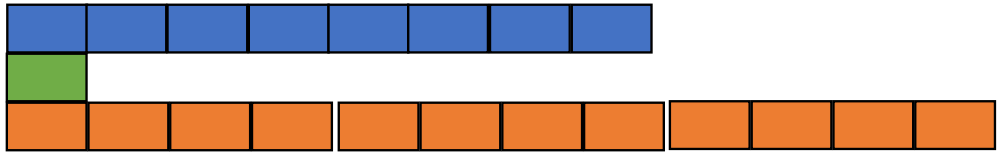
同じ間隔（ストライド：歩幅）

ストライド・ベクトル・データ型

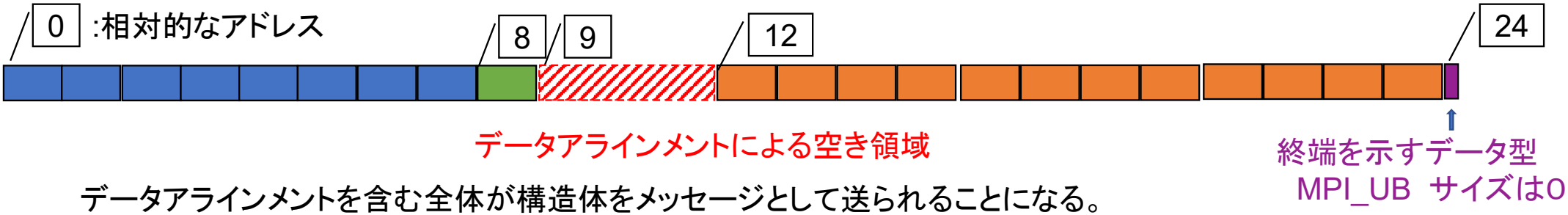
```
MPI_Datatype newtype;
MPI_Type_vector(3, 2, 5, MPI_INT, &newtype);   // これは型の構造を定義しただけ
                        引数：個数、1ヶ所の個数、ストライド、1個分の型、新しく定義する型名
MPI_Type_commit(&newtype);    // これでnewtypeをMPI_Send関数でデータ型として使用可能になる
      MPI_Send(&a[0][1],1,newtype,…);
```

構造体データタイプ

```
struct cell{
    double      energy;
    char        flags;
    float       coord[3];
};
```



構造体メンバもメモリ内では一次元化される



データアラインメントを含む全体が構造体をメッセージとして送られることになる。

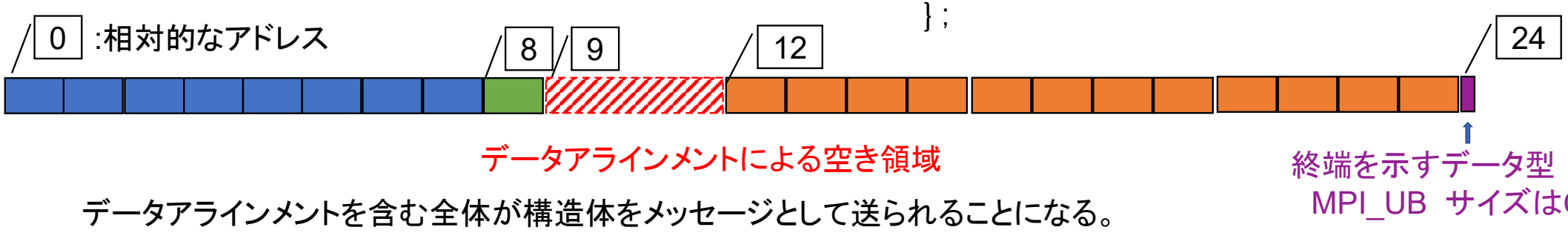
MPI_Type_struct(int count, int blocklengths[], MPI_Aint displacements[], MPI_Datatype dtypes[],
MPI_Datatype *newtype);

引数:メンバ数, 各ブロックの長さ, 各ブロックの相対位置, 各ブロックのデータ型,
定義される新しいデータ型

構造体データタイプ

```
struct cell{
    double      energy;
    char        flags;
    float       coord[3];
};
```

構造体メンバもメモリ内では一次元化される



データアラインメントを含む全体が構造体をメッセージとして送られることになる。

MPI_Type_struct(int count, int blocklengths[], MPI_Aint displacements[], MPI_Datatype dtypes[],
MPI_Datatype *newtype);

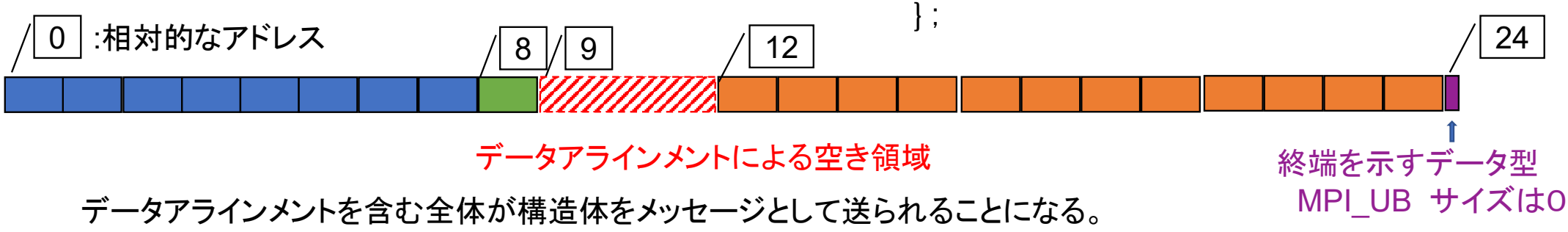
引数：メンバ数, 各ブロックの長さ, 各ブロックの相対位置, 各ブロックのデータ型,
定義される新しいデータ型

```
int  blocklengths[4]={1, 1, 3, 1}; // double1個、char1個、float3個、MPI_UB 1個
MPI_Aint displacements[4]={0, 8, 12, 24};    // 各ブロックの相対アドレスを求めるのが困難な場合がある
MPI_Datatype dtypes[4]={MPI_DOUBLE, MPI_CHAR, MPI_FLOAT, MPI_UB};
MPI_Type_struct(4, blocklengths, displacements, dtypes, &celltype);
MPI_Type_commit(&celltype);
```

構造体データタイプ

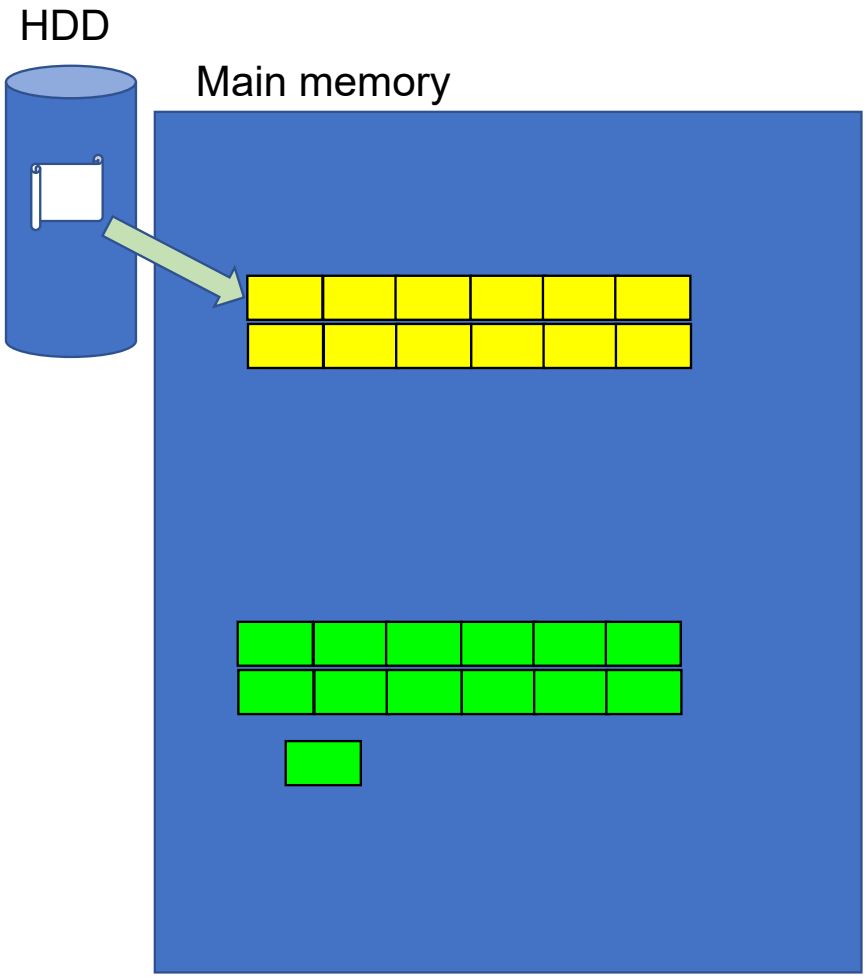
```
struct cell{
    double      energy;
    char        flags;
    float       coord[3];
};
```

構造体メンバもメモリ内では一次元化される



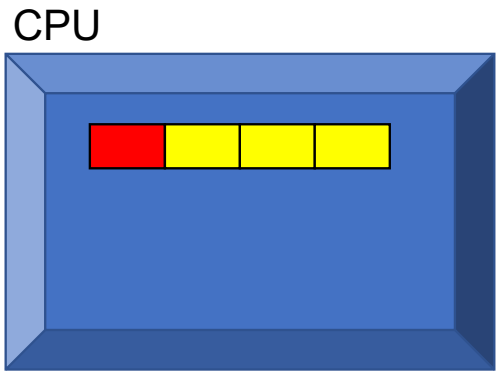
データアラインメントを含む全体が構造体をメッセージとして送られることになる。

```
int  blocklengths[4]={1, 1, 3, 1}; // double1個、char1個、float3個、MPI_UB 1個
MPI_Aint displacements[4];    // 各ブロックの相対アドレスを求めるのが困難な場合がある
MPI_Aint base;
struct cell cloud[2];
MPI_Address(&cloud[0].energy, &displacements[0]); // displacements[0]=&cloud[0].energyの意味
MPI_Address(&cloud[0].flags,    &displacements[1]);
MPI_Address(cloud[0].coord,      &displacements[2]);
MPI_Address(&cloud[1].energy, &displacements[3]);
base=displacements[0];
for(int i=0; i<4; i++) displacements[i]-=base;
MPI_Datatype dtypes[4]={MPI_DOUBLE, MPI_CHAR, MPI_FLOAT, MPI_UB};
MPI_Type_struct(4, blocklengths, displacements, dtypes, &celltype);
MPI_Type_commit(&celltype);
```



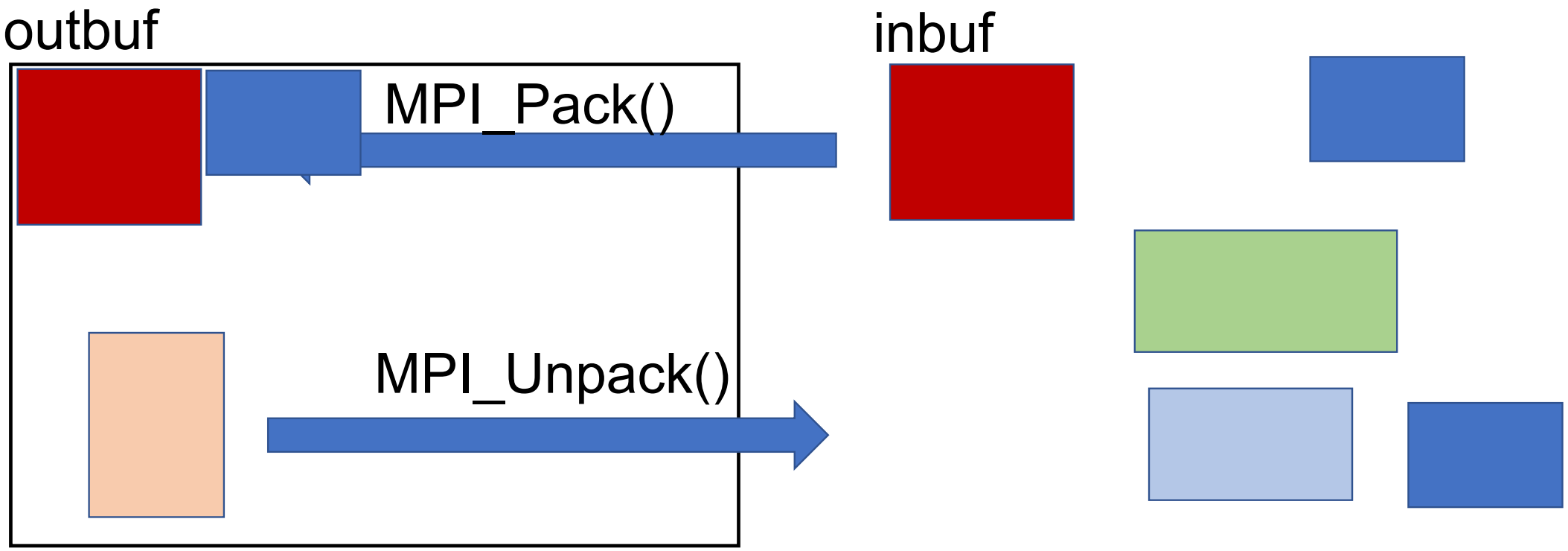

```
int main()
{
   int a,b;
   for(int i=0; i<10; i++){


   }
```

$ ./a.out　　実行プログラムのメモリへのロード



命令のCPUへのフェッチ

色々なデータをパックしてメッセージにする



MPI_Send(buffer, position, **MPI_PACKED**, …);