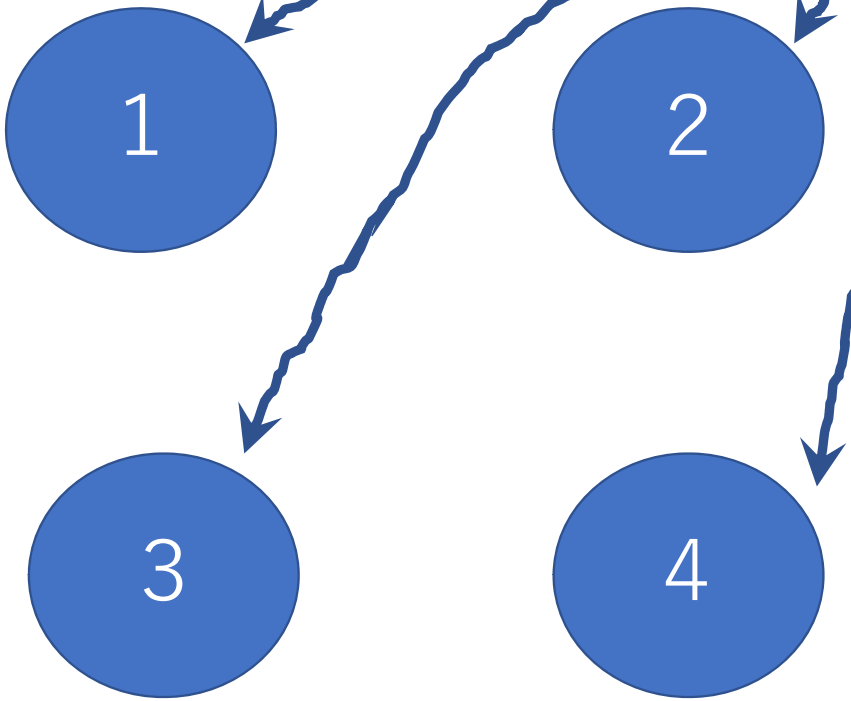
$ mpicc mpi-ex01.c –o mpi-ex01　これがコンパイル方法

$ mpiexec –n 4 ./mpi-ex01　実行の仕方




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

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

  printf("Hello!!!¥n");

  MPI_Finalize();
  return 0;
}
```

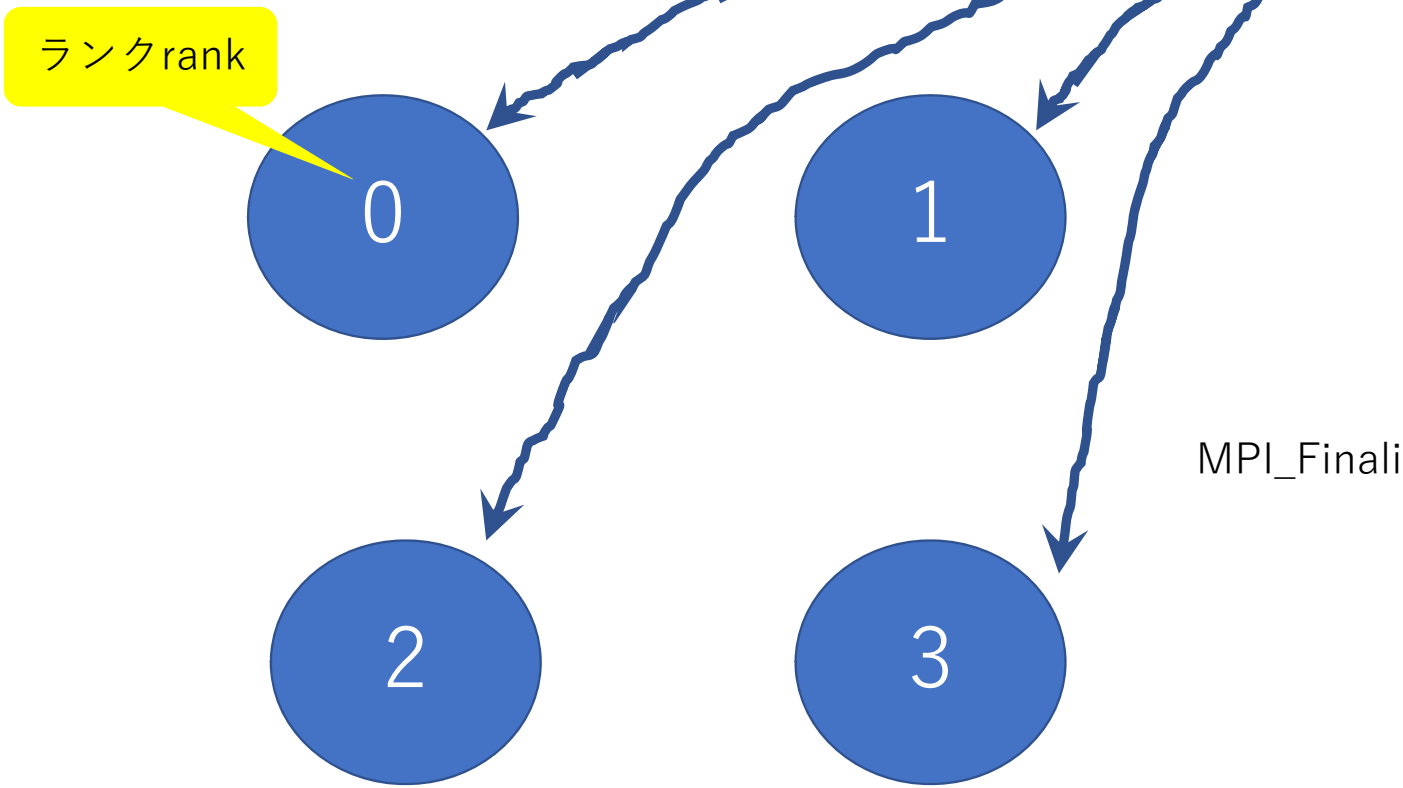
MPI_Init(&argc, &argv)：MPIの初期化
argc:コマンドライン上の引数の個数
左上の例では　argc=4
argv: 各引数の文字列の先頭アドレス
argv[0] →　“mpiexec”
argv[1] →　“-n”
argv[2] →　“4”
argv[3] → “./mpi-ex01”

mpiexec –n 4 ./mpi-ex01 → ./mpi-ex01

argc=1
argv[0]→”./mpi-ex01

$ mpicc mpi-ex01.c –o mpi-ex01　これがコンパイル方法

$ mpiexec –n 4 ./mpi-ex01　実行の仕方

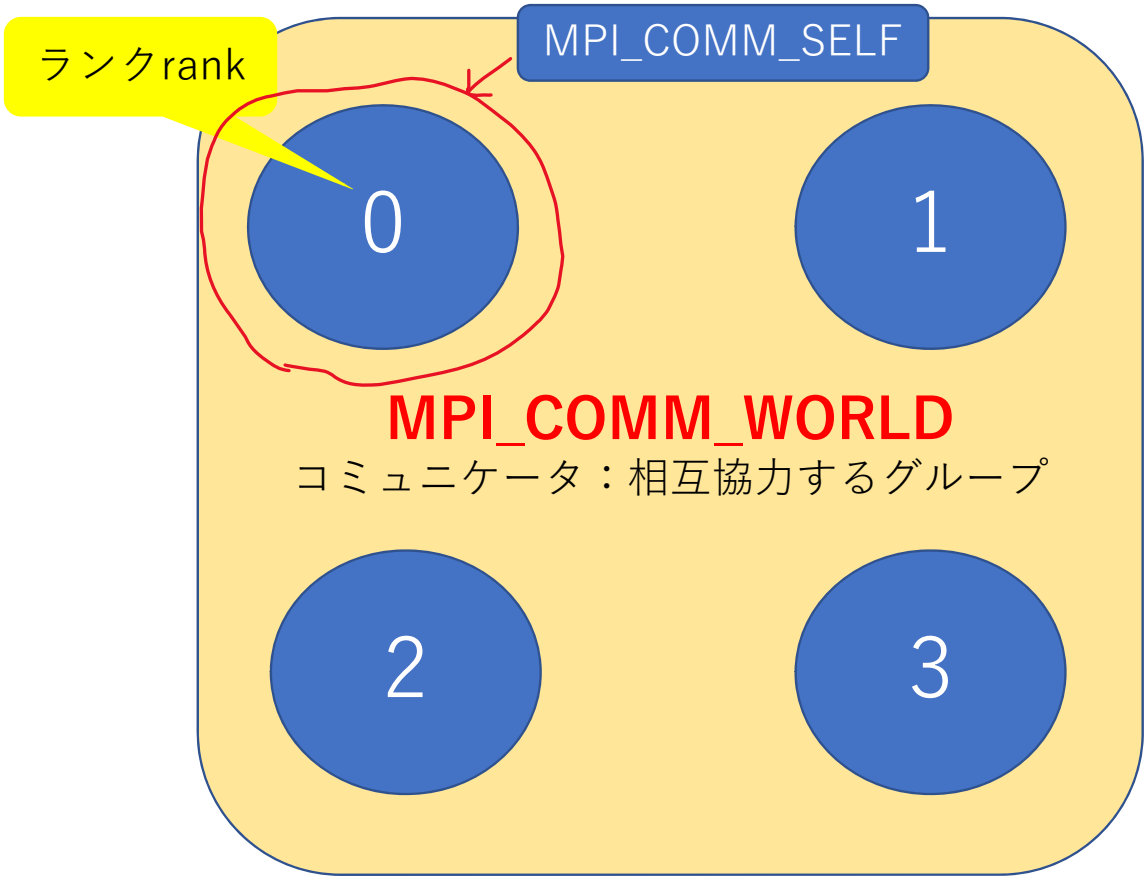



```
#include <stdio.h>
#include <mpi.h>

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

  printf("Hello!!!¥n");

  MPI_Finalize();
  return 0;
}
```

MPI_Finalize( )：MPIの終了処理（解散）

$ mpicc mpi-ex01.c –o mpi-ex01　これがコンパイル方法

$ mpiexec –n 4 ./mpi-ex01　実行の仕方



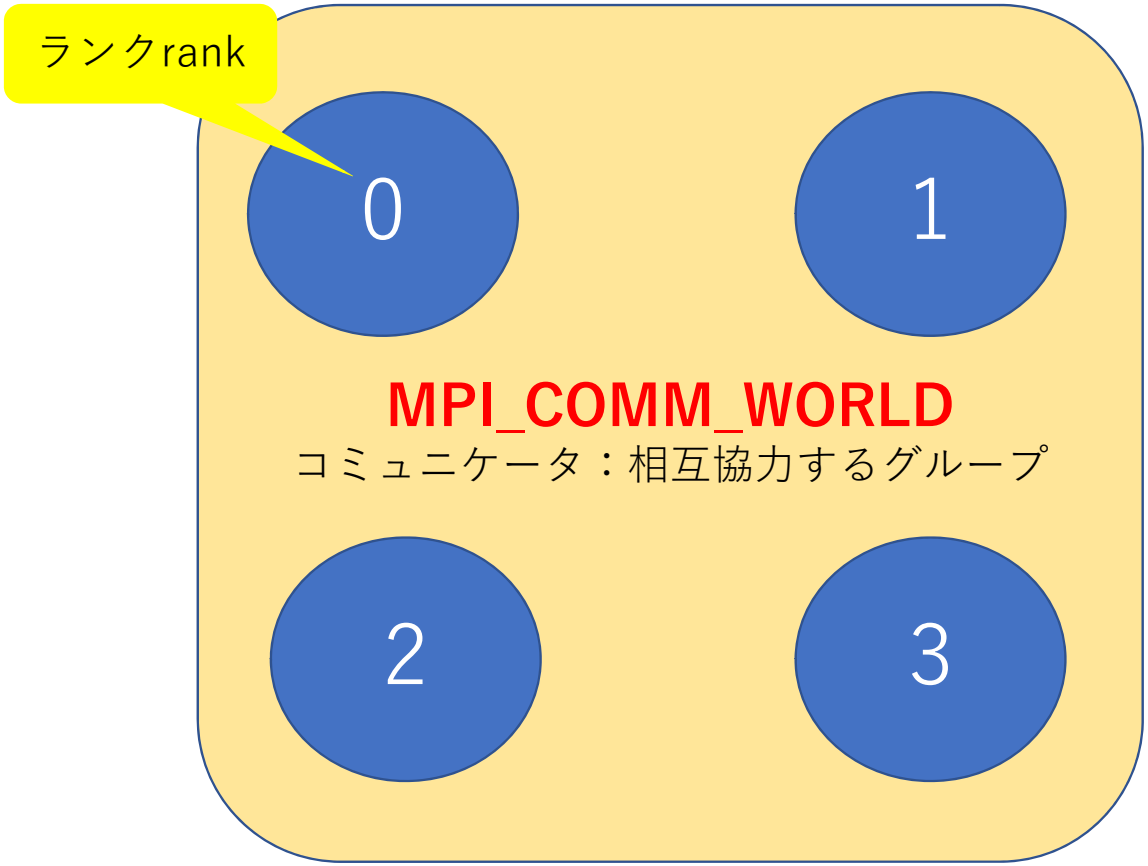

```c
#include <stdio.h>
#include <mpi.h>

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

$ mpicc mpi-ex01.c –o mpi-ex01　　これがコンパイル方法

$ mpiexec –n 4 ./mpi-ex01　　実行の仕方




```
#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!!!¥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

```
#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
指示する形で、並列実行ができる

MPIにおける通信

- 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

- 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_Status status; が追加

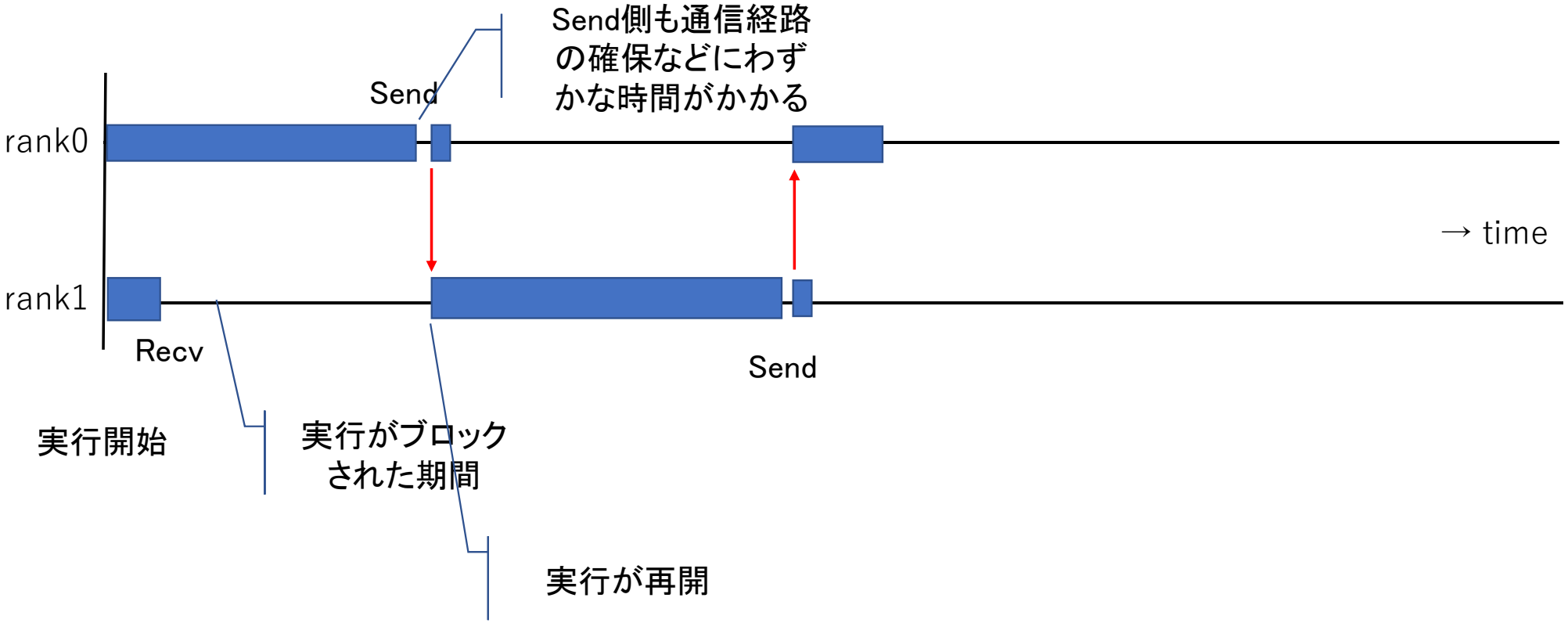
- MPI_Recv(&x,1,MPI_INT, 0, 10, MPI_COMM_WORLD, &status);

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

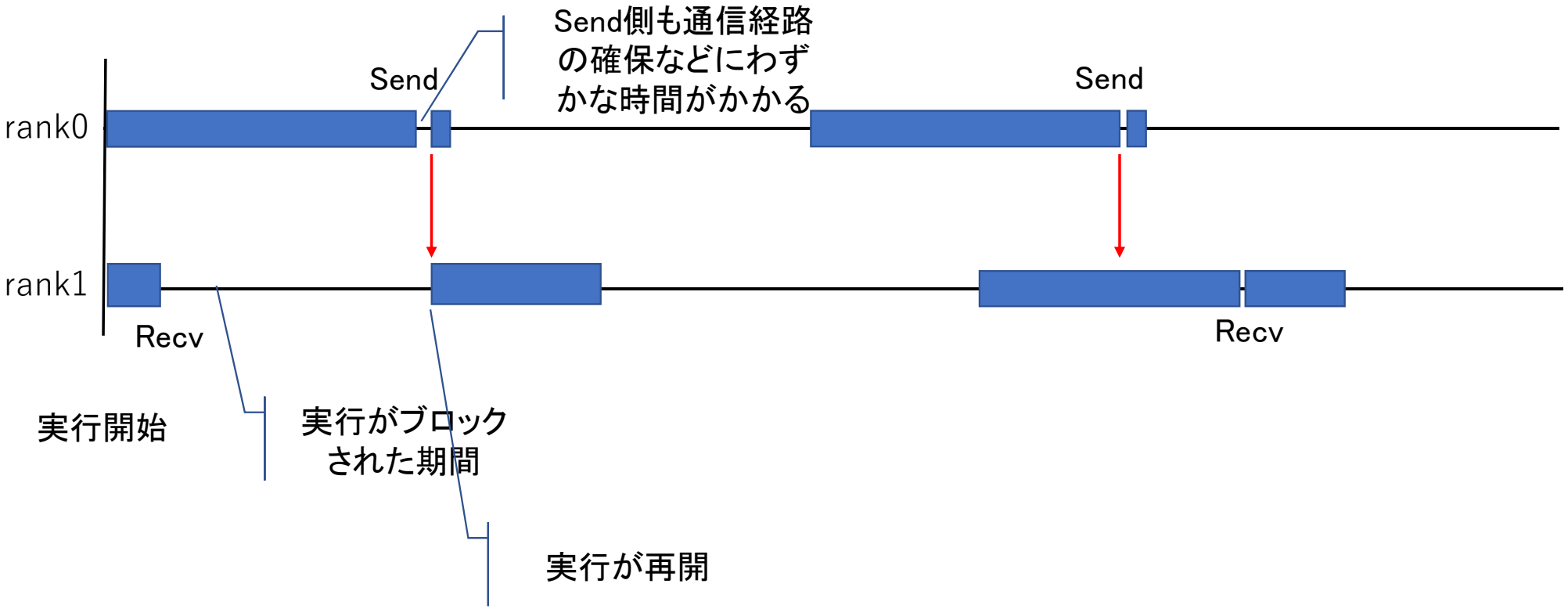
Send／Recvのタイミング

133.220.114.90　グローバルIPアドレス

192.168.1.1　ローカルIPアドレス



Send／Recvのタイミング



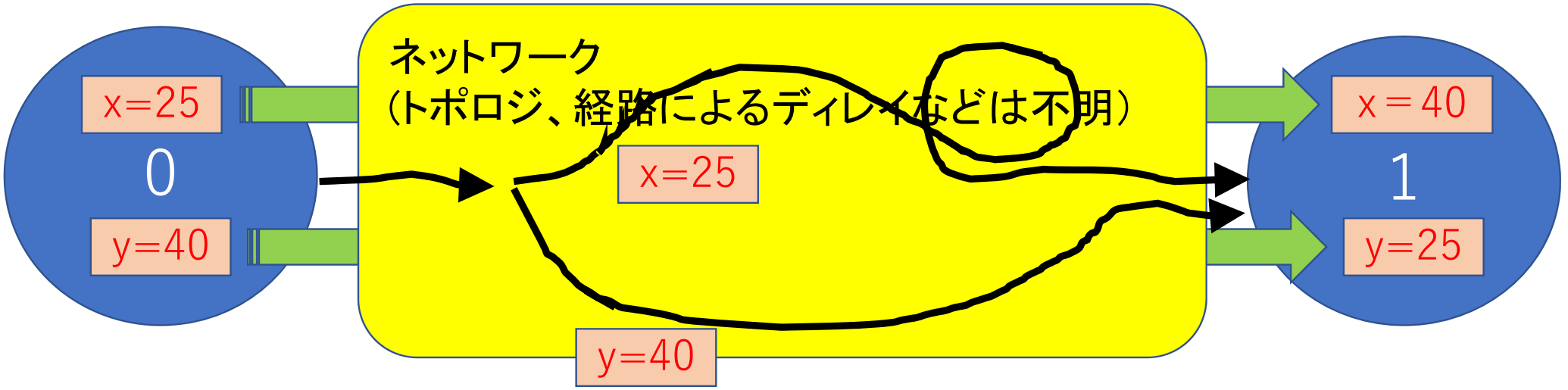
Send／Recvに指定されるタグについて

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 );
```



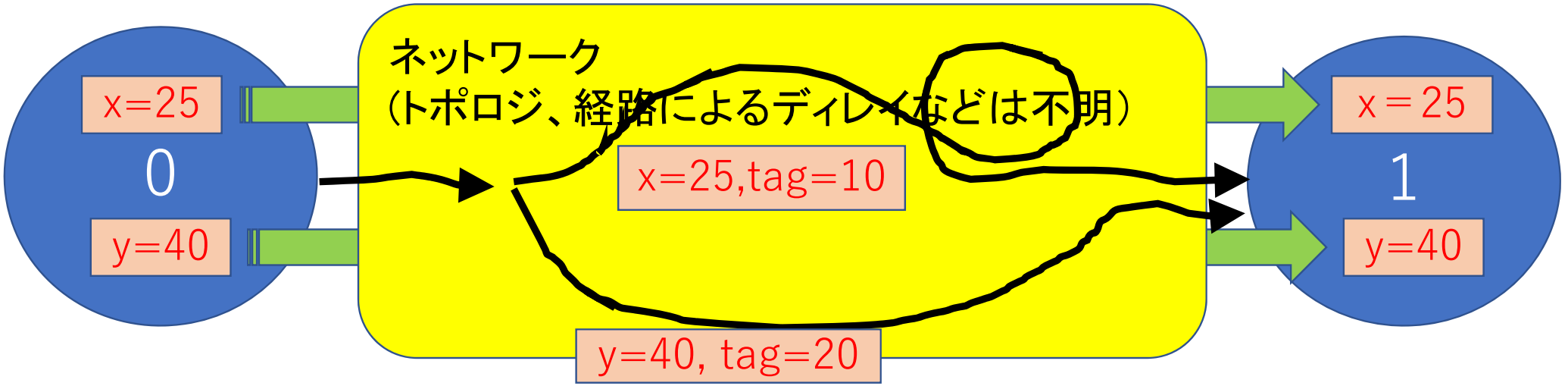
Send／Recvに指定されるタグについて

Rank0は、

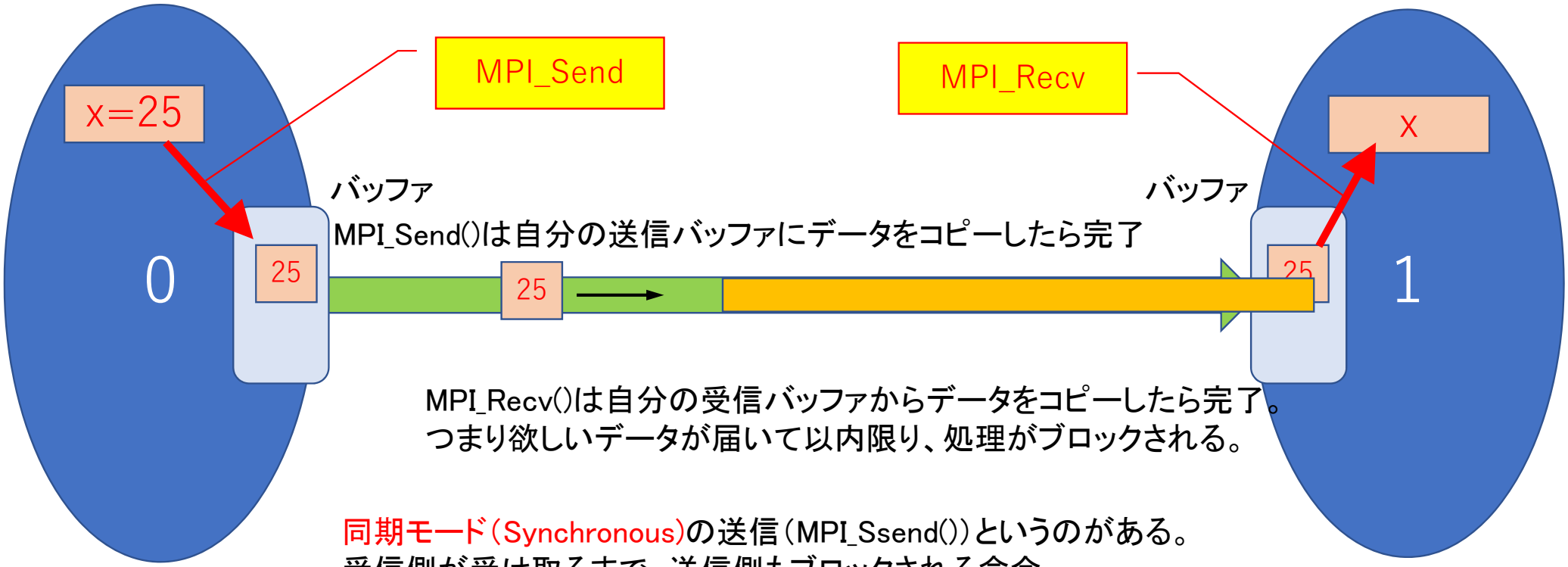
```
MPI_Send( &x,1,MPI_INT, 1, 10, MPI_COMM_WORLD );
MPI_Send( &y,1,MPI_INT, 1, 20, MPI_COMM_WORLD );
```

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 );
```



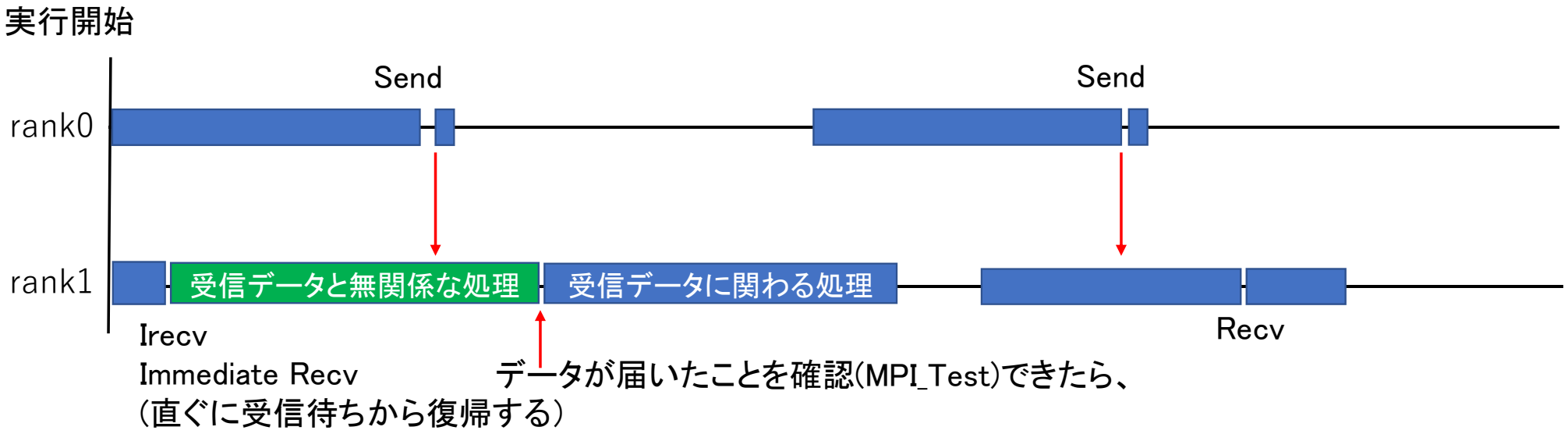
Send／Recvに使用されるバッファについて



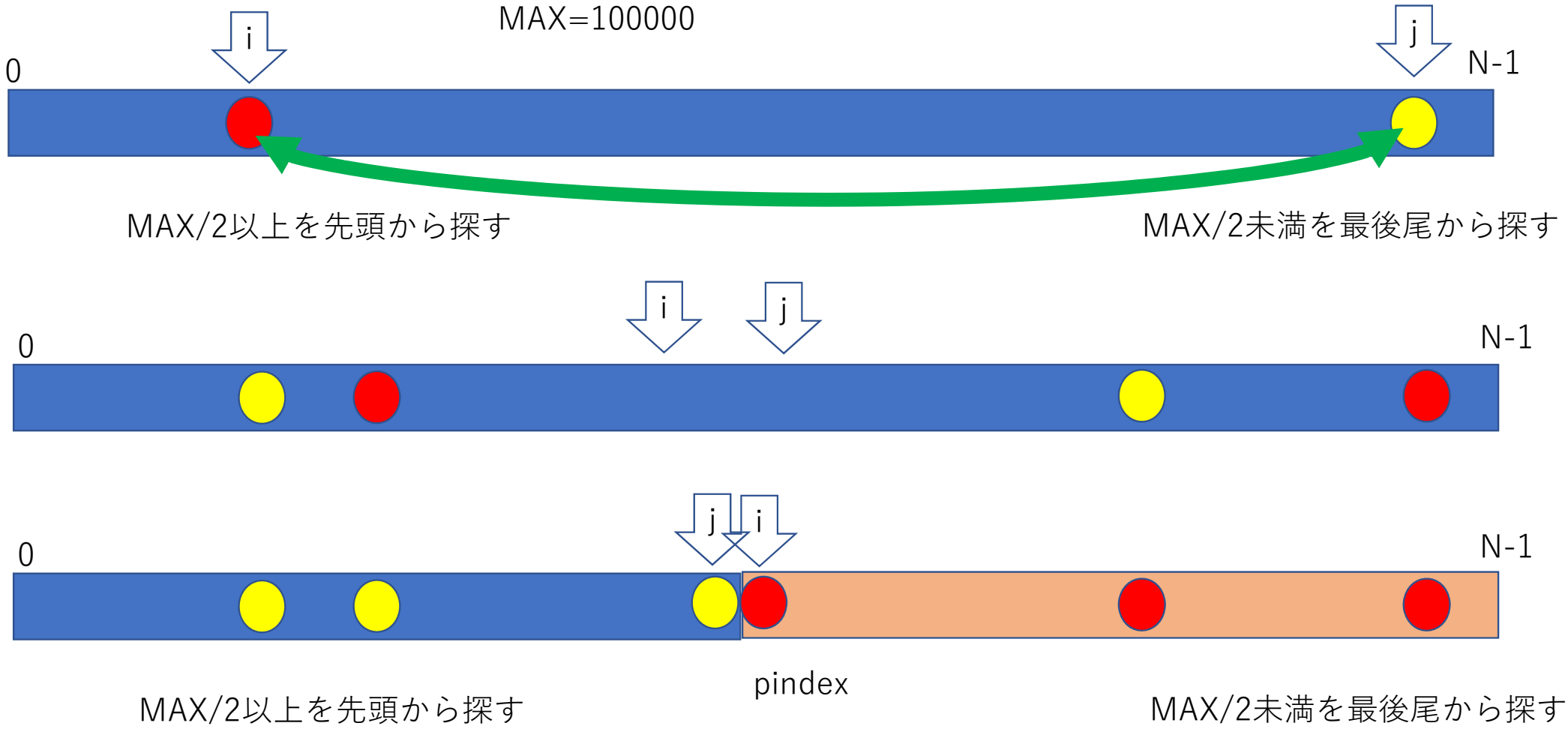
MPI_Recv()は自分の受信バッファからデータをコピーしたら完了。
つまり欲しいデータが届いて以内限り、処理がブロックされる。

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。

ノンブロッキング通信

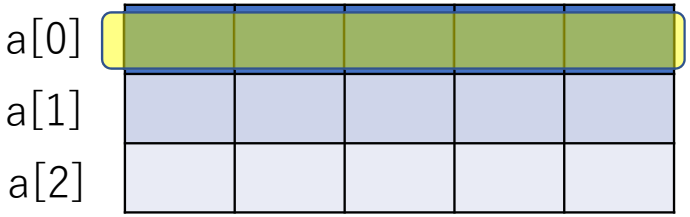

実行開始
Send
Send
rank0
rank1
受信データと無関係な処理
受信データに関わる処理
Irecv
Immediate Recv
(直ぐに受信待ちから復帰する)
Recv
データが届いたことを確認(MPI_Test)できたら、

2つのプロセスで並列ソート



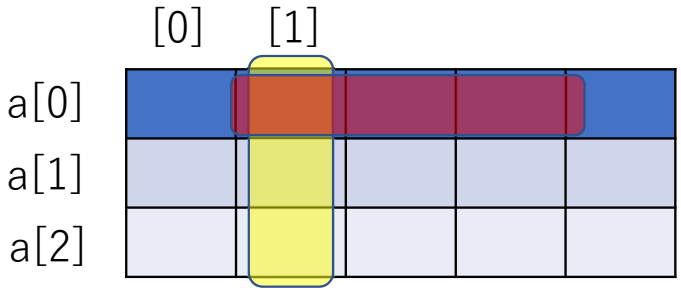
ストライド・ベクトル・データタイプ

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

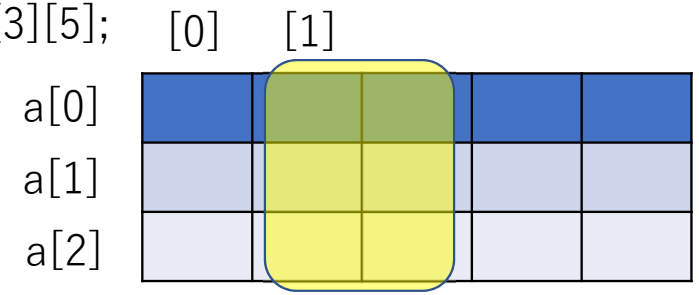


では、列方向のデータを送るにはどうするか？

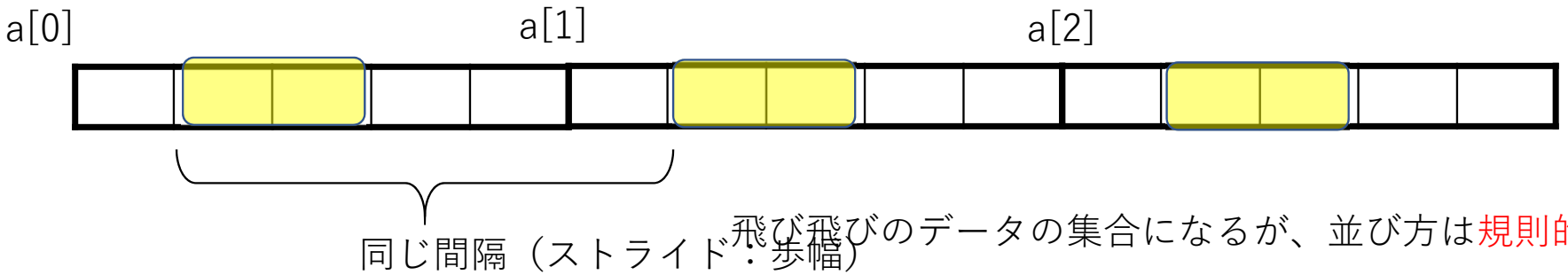
MPI_Send(&a[0][1],3,MPI_INT,　これはできない
（図の赤い部分の送信になってしまう）

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？



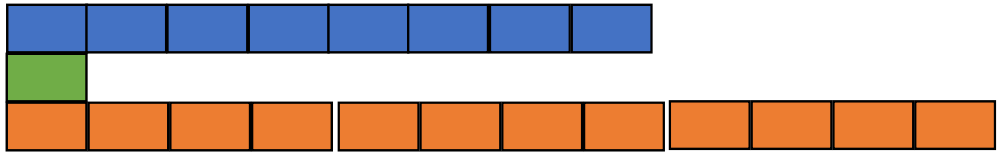
飛び飛びのデータの集合になるが、並び方は規則的になっている。

ストライド・ベクトル・データ型

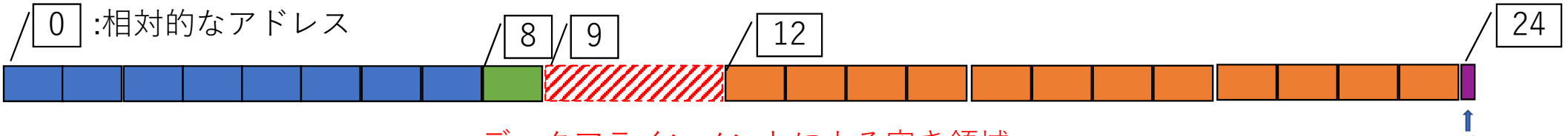
```
MPI_Datatype newtype;
MPI_Type_vector(3, 2, 5, MPI_INT, &newtype);  // これは型の構造を定義しただけ
                    引数：個数、１ヶ所の個数、ストライド、1個分の型、新しく定義する型名
MPI_Type_commit(&newtype);   // これでnewtypeをMPI_Send関数でデータ型として使用可能になる
        MPI_Send(&a[0][1],1,newtype,…);
```

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。

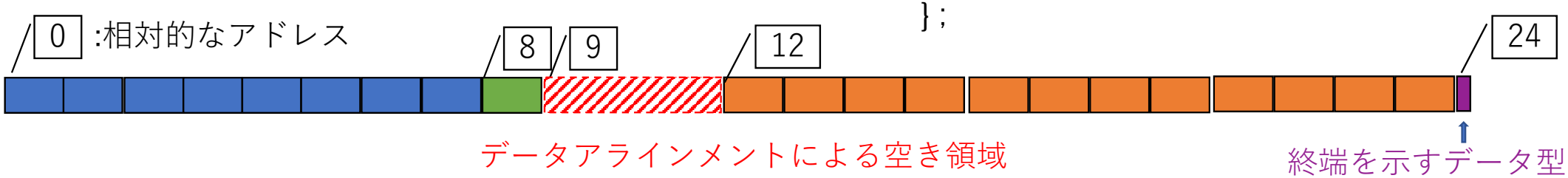
```
MPI_Type_struct(int count, int blocklengths[], MPI_Aint displacements[], MPI_Datatype dtypes[],
                                                                    MPI_Datatype *newtype);
```

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);

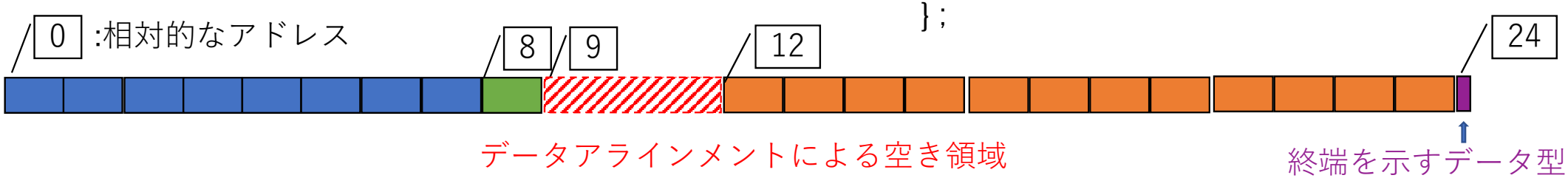
引数：メンバ数, 各ブロックの長さ, 各ブロックの相対位置, 各ブロックのデータ型,
定義される新しいデータ型

```
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

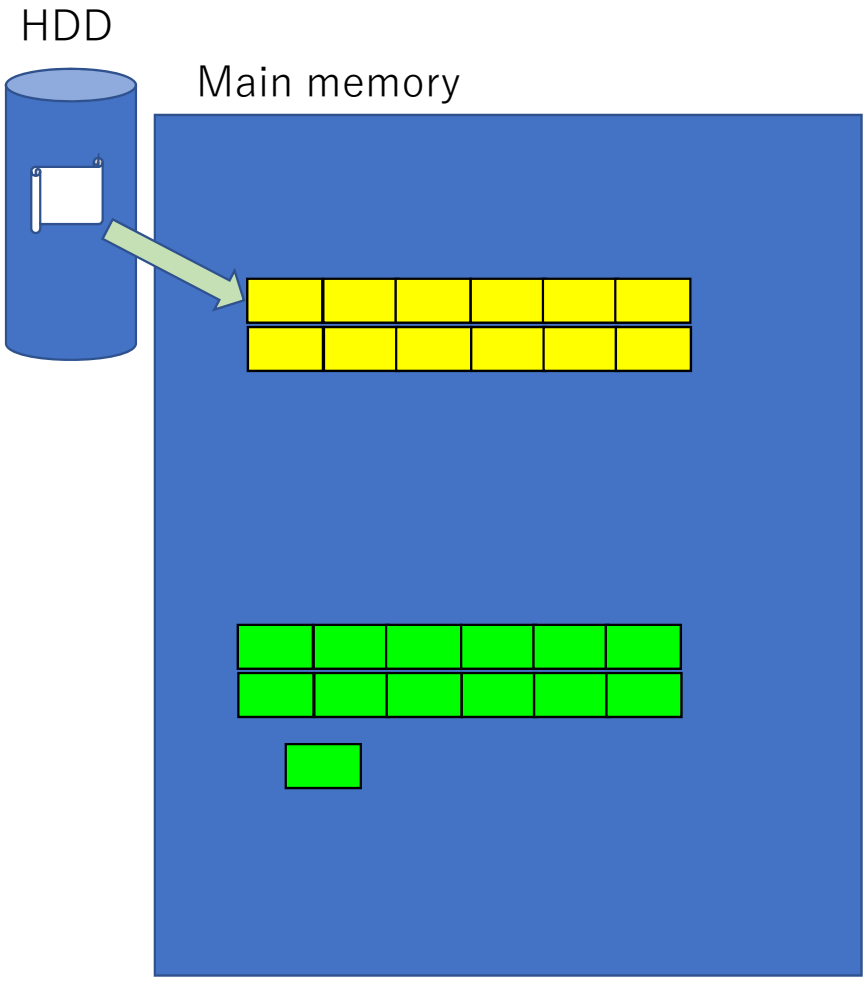
```
struct cell{
    double      energy;
    char        flags;
    float       coord[3];
};
```

構造体メンバもメモリ内では一次元化される



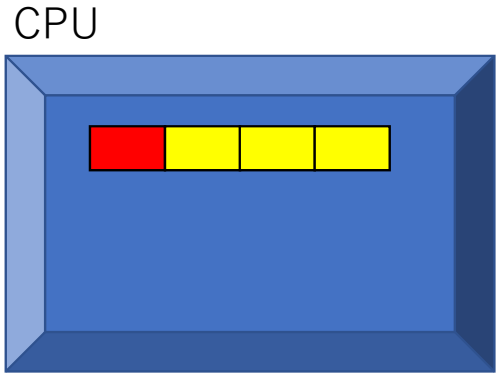
データアラインメントを含む全体が構造体をメッセージとして送られることになる。

```
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);
```



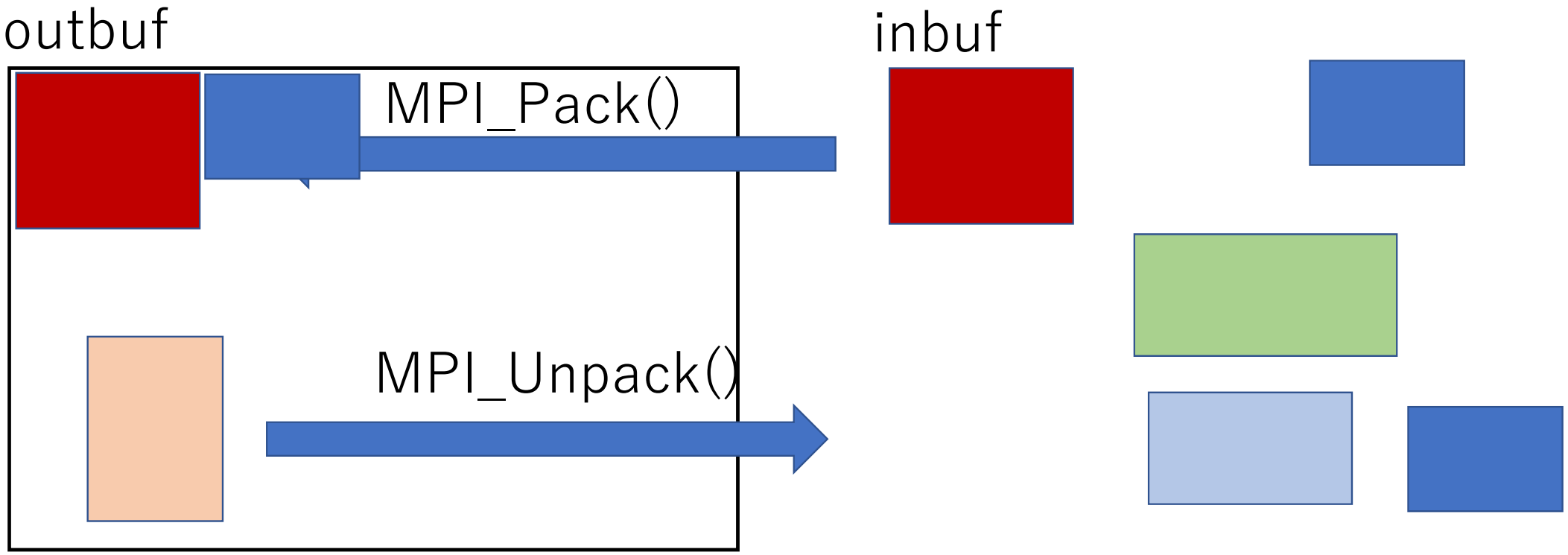

```
int main()
{
   int a,b;
   for(int i=0; i<10; i++){


   }
```

$ ./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**, …);