

## 10. 並列計算性能の評価方法

（時間計測関数，バリア同期関数，sendrecv関数）

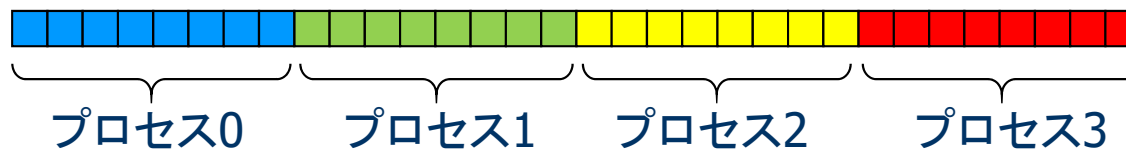
## 演習10-1 MPI\_Allreduceの応用：ベクトルの正規化

- $n$ 次元ベクトル  $x$  (`double x[N];`) の第  $i$  要素を  $i + 1$  とする.
  - ◆  $x[i] = i + 1 \quad (i = 0, \dots, n - 1)$
- $x$  を正規化したベクトル  $x/\|x\|_2$  を求めるプログラム `normalize.c` の逐次実行動作を確認せよ.
  - ◆ 変数は倍精度 (`double`) とし, 倍精度で計算する.  $\|x\|_2$  は  $x$  の各要素の2乗和の平方根である.
  - ◆ 正規化されたベクトルの要素は
$$\tilde{x}[i] = (i + 1) / \sqrt{n(n + 1)(2n + 1)/6}$$
であるので (簡単に計算できる), プログラムではこの値と比較することによって, 正しく計算ができている (`Error = 0.0`) ことを確認している.

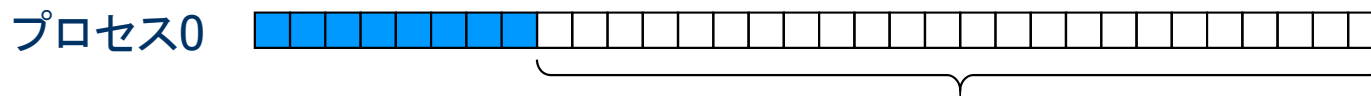
```
$ cp /tmp/Summer/M-3/normalize.c ./
$ icc -O3 normalize.c
$ ./a.out
Error = 0.000000
```

## 演習10-2 normalize.cを並列化せよ

- 各プロセスが自分の担当分の要素について、2乗和を計算する。
- 各プロセスの担当する要素 (nprocs はMPIプロセス数)
  - $i_{start} = (n/nprocs) * myrank$
  - $i_{end} = (n/nprocs) * (myrank+1) - 1$



- ベクトルの格納方法
  - ◆ 各プロセスは長さ  $n$  の配列を持ち、そのうち自分の担当部分のみを使う



プロセス0では、この部分が使われない

- 全プロセスの総和を `MPI_Allreduce`関数で求め、各プロセスはその総和を使って、自分の担当する要素について正規化を行う。
- $n=1000$  としてプロセス数を1, 2, 4, 8と変えて計算せよ。結果の確認も工夫すること。

# 時間計測関数, バリア同期関数

# 計算時間の計測

- 並列計算の目的は、計算時間の短縮にある。
  - ◆ 大規模問題を解くためにメモリ容量を増やすことも目的の一つ
- 同じ結果が得られるが、アルゴリズムや書き方が異なったプログラムのうち、どれが一番良いか？

 (正しい結果が得られるならば) 計算時間の短いものが良いはず。

- 計算時間を計測して比較する。

# 計算時間を計測する方法

```
double time0, time2;  
    .  
    .
```

```
MPI_Barrier( MPI_COMM_WORLD );  
time0 = MPI_Wtime();
```

(計測する部分)

```
MPI_Barrier( MPI_COMM_WORLD );  
time1 = MPI_Wtime();
```

(time1-time0 を出力する)

計測のための変数を倍精度実数型で宣言する。

MPI\_Barrier関数で、計測開始の足並みを揃える。  
mpi\_wtime関数で開始時刻をtime0に設定

全プロセスで終了の足並みを揃える。  
mpi\_wtime関数で終了時刻をtime1に設定

time1-time0が計測した部分の計算時間となる。

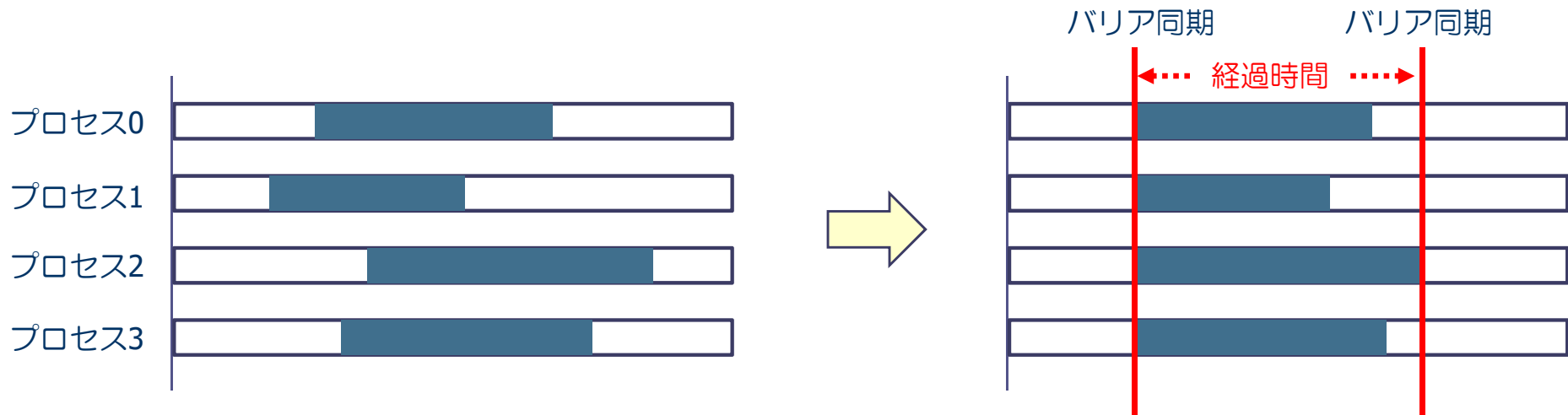
MPI\_Barrier(comm) : バリア同期関数

◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)

var = MPI\_Wtime() : 倍精度実数を返す関数 (double var;)

# 時間計測のイメージ

- 各プロセスでの計算時間の測定関数
  - ◆ MPI\_Wtime()
    - ある時点を基準とした経過秒数を**倍精度実数型**で返す関数
- プログラムのある区間の計算時間の測定
  - ◆ プログラムの実行は各プロセスで独立なので、開始時間や終了時間が異なる。
  - ◆ ある部分の計算時間の計測では、**バリア同期** (MPI\_Barrier) により測定開始と測定終了の**足並みを揃えて**、計測する。



## 演習10-3 並列化normalize.c の実行時間計測

- 並列化したベクトルの正規化プログラムの実行時間を計ってみる.
  - ◆ 計測範囲は、各プロセスが二乗和を求めるところの直前から、正規化した直後まで、とする。
- $n = 50,000, 100,000, 500,000$  に対して、並列数を1, 2, 4, 8, 16 と変化させて時間を計測せよ【最後に. . .】
  - ◆ 1プロセスの実行時間を $T(1)$ 、 $m$ プロセスの実行時間を  $T(m)$  とするとき、速度向上率  $T(1)/T(m)$  を求めよ（次の表参照）。
  - ◆ また、速度向上率のグラフを描け（例えば gnuplotを使う）。
    - gnuplotは、Linuxで動作するグラフ描画ソフトウェア。



# 表：ベクトルサイズと並列数の関係

並列数 (m)	n=50,000		n=100,000		n=500,000	
	計算時間 (秒) T(m)	速度向上率 T(1)/T(m)	計算時間 (秒) T(m)	速度向上率 T(1)/T(m)	計算時間 (秒) T(m)	速度向上率 T(1)/T(m)
1		1.000		1.000		1.000
2						
4						
8						
16						

# MPI\_Sendrecv関数

## 演習10-4 【準備】 関数 MPI\_Sendrecv

- プログラム sr.c は、2つのプロセスで互いにデータを送りあうプログラムである。
  - ◆ プログラムをコピー，実行し，動作を確かめる。

```
$ cp /tmp/Summer/M-3/sr.c ./
```

```
$ icc sr.c -lmpi
```

```
$ qsub ... ← 2プロセスのMPIのバッチジョブ（これまでのモノを利用）
```

```
$ cat xxxxx.onnnnnn ← バッチジョブの実行結果を確認
```

```
Before exchange... Rank: 0, a0= 1.0, a1= -99.0
```

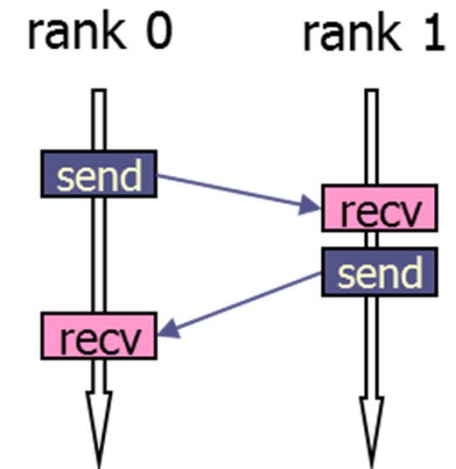
```
Before exchange... Rank: 1, a0= 2.0, a1= -99.0
```

```
After exchange... Rank: 0, a0= 1.0, a1= 2.0
```

```
After exchange... Rank: 1, a0= 2.0, a1= 1.0
```

# プログラム sr.c の説明

```
#include <mpi.h>                                include行
int main( int argc, char **argv )
{
    double a0, a1;
    int nprocs, myrank;
    MPI_Status status;
    【省略】
    if( myrank == 0 ) {
        a0 = 1.0 ;                                ← rank 0では, a0=1.0, a1=0.0
        a1 = -99.0 ;
    } else {
        a0 = 2.0 ;                                ← rank 1では, a0=0.0, a1=2.0
        a1 = -99.0 ;
    };
    【省略】
    if( myrank == 0 ) {
        MPI_Send( &a0, 1, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD );
        MPI_Recv( &a1, 1, MPI_DOUBLE, 1, 200, MPI_COMM_WORLD, &status );
    } else {
        MPI_Recv( &a1, 1, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &status );
        MPI_Send( &a0, 1, MPI_DOUBLE, 0, 200, MPI_COMM_WORLD );
    };
    【省略】
}
```



※ 交換部分：各プロセスでのsend, recvの**実行順序に注意**

rank 0の a0 の値を rank 1へ送る。  
rank 1からの値を a1 で受け取る。

rank 0 からの値を a1 で受け取る。  
rank 1の a0 の値を rank 0へ送る。

※キーワード：**ブロッキング関数**

# 双方向通信 : MPI\_Sendrecv関数

```
MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest,  
              int sendtag,  
              void *recvbuf, int recvcount, MPI_Datatype recvtype, int source,  
              int recvtag,  
              MPI_Comm comm, MPI_Status *status )
```

- ◆ sendbuf: 送信するデータのための変数名 (先頭アドレス)
- ◆ sendcount: 送信するデータの数
- ◆ sendtype: 送信するデータの型
  - MPI\_INT, MPI\_DOUBLE, MPI\_CHAR など
- ◆ dest: 送信する相手のプロセス番号 (destination)
- ◆ sendtag: メッセージ識別番号. 送るデータを区別するための番号
- ◆ recvbuf: 受信するデータのための変数名 (先頭アドレス)
- ◆ recvcount: 受信するデータの数 (整数型)
- ◆ recvtype: 受信するデータの型
- ◆ source: 送信してくる相手のプロセス番号
- ◆ recvtag: メッセージ識別番号. 送られて来たデータを区別するための番号
- ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)
- ◆ status: 受信の状態を格納する変数

## 演習12-4 : プログラムsr.c の書き換え

- プログラム sr.c のデータの交換部分を MPI\_Sendrecv関数で書き換えよ.

```
if( myrank == 0 ) {  
    MPI_Send( &a0, 1, MPI_DOUBLE, 1, 100, MPI_COMM_WORLD );  
    MPI_Recv( &a1, 1, MPI_DOUBLE, 1, 200, MPI_COMM_WORLD, &status );  
} else {  
    MPI_Recv( &a1, 1, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &status );  
    MPI_Send( &a0, 1, MPI_DOUBLE, 0, 200, MPI_COMM_WORLD );  
};
```

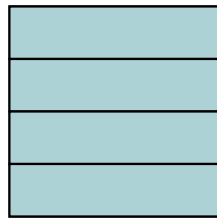


MPI\_Sendrecv( . . . ); rank 0 では, rank 1 に送り, rank 1 から受け取る.  
rank 1 では, rank 0 に送り, rank 0 から受け取る.

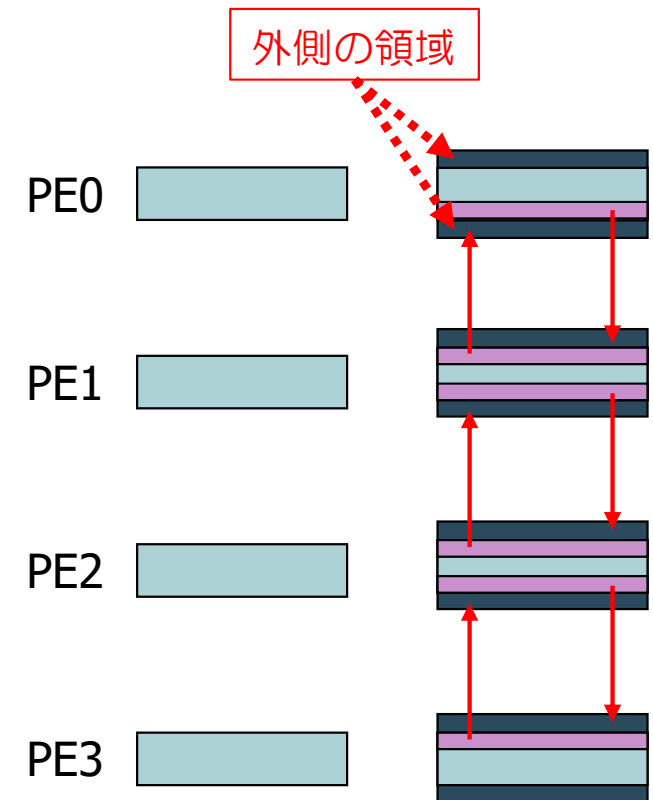
# 【発展】 MPI\_Sendrecvの行列への適用

- 2次元配列 ( $u[N+2][N+2]$ ) をブロック行分割する.

2次元配列  
(ブロック行分割)



- このとき、自分の上下の1行の要素を、隣接するプロセスの持つ領域の外側に受信用の領域を確保し、その領域に各プロセスが転送するプログラムを作る.



上下のプロセスから1行を受信  
(受信用の領域を確保しておく)

# MPI\_Sendrecvの発展（続き）

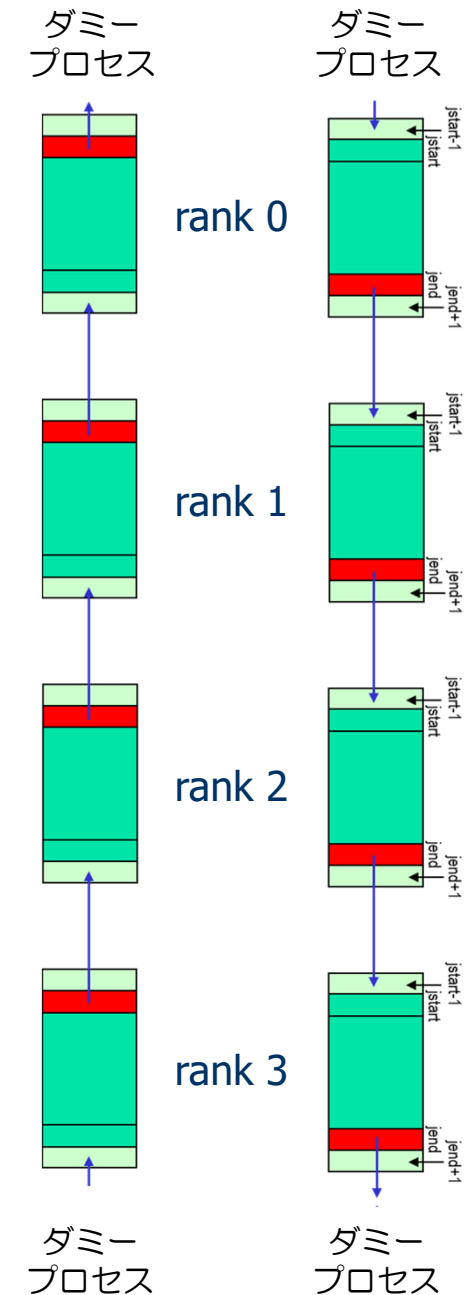
第1の sendrecv    第2の sendrecv

## ■ 自分の担当範囲を計算

- ◆ 変数myrankを用いて、担当範囲  $js \sim je$  行を計算
- ◆ 受信領域を考慮し、 $js-1$ ,  $je+1$  行の領域を扱う。
  - メモリ節約の観点では、自分の領域分のメモリがあれば良いが、C言語のポインタ、malloc関数を使ってやる必要があるため、本スクールではやらない（上級）。

## ■ MPI\_Sendrecv による送受信

- ◆ まず、上側に  $js$  行を送り、下側から送られてくるデータを  $(je+1)$  行で受信
- ◆ 次に、下側に  $je$  行を送り、上側から送られてくるデータを  $(js-1)$  行に受信
- 最上下端のプロセスは、ダミープロセス (MPI\_PROC\_NULL) と送受信するようにする。
  - MPI\_Sendrecv の source, dest に使うことができる。ダミープロセスを指定すると、実際にはどこからにもデータを送らないし、どこからもデータを受け取らない。





# プログラム sr\_matrix.c (続く)

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

#define N 20

int main( int argc, char **argv )
{
    double u[N+2][N+2];
    int i, j, is, ie;

    int nprocs, myrank, upper, lower, srtag;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

    is = (N/nprocs)* myrank + 1;
    ie = (N/nprocs)*(myrank+1);

    upper = myrank-1;
    if( myrank == 0 ) upper = MPI_PROC_NULL;

    lower = myrank+1 ;
    if( myrank == nprocs-1 ) lower = MPI_PROC_NULL;
```

N = 20 とする

(N+2)次の行列として宣言する。後で分かる。

Recvで必要な配列変数の宣言

各プロセスの担当する行の範囲を計算

上側プロセスのプロセス番号を設定  
(存在しない場合は MPI\_PROC\_NULL とする)

下側プロセスのプロセス番号を設定

# プログラム sr\_matrix.c (続き)

```
for( i=is; i<=ie; i++ ) {
    for( j=0; j<N+2; j++ ) {
        u[i][j] = (myrank+1)*10.0 ;
    };
};

/* from the lower process to the upper one */
MPI_Sendrecv(      );

/* from the upper process to the lower one */
MPI_Sendrecv(      );

printf("Rank = %d %d %d\n", myrank, is, ie ) ;
for( i=is-1; i<=ie+1; i++ ) {
    printf("%6.2f", u[i][N/2] ) ;
};
printf("\n") ;

MPI_Finalize();
return 0;
}
```

/\* 下側のプロセスから上側へ \*/  
MPI\_Sendrecv による送受信

/\* 上側のプロセスから下側へ \*/  
MPI\_Sendrecv による送受信

N/2 の列のみプリントして確認している。

# 演習10-5 : プログラムの完成と実行

- `sr_matrix.c` は**未完成**である。
- `MPI_Sendrecv`の部分を完成させ、コンパイルして、2, 4 プロセスで実行し、データの送受信が正しくできていることを確かめよ。  
`$ cp /tmp/Summer/M-3/sr_matrix.c ./`
- 実行結果（出力順は、以下とは異なる）。

```
Rank= 0, is:ie=  1:  5
  0.00 10.00 10.00 10.00 10.00 10.00 10.00 20.00
Rank= 1, is:ie=  6: 10
10.00 20.00 20.00 20.00 20.00 20.00 30.00
Rank= 2, is:ie= 11: 15
20.00 30.00 30.00 30.00 30.00 30.00 40.00
Rank= 3, is:ie= 16: 20
30.00 40.00 40.00 40.00 40.00 40.00 0.00
```

上のプロセスのie行が、下のプロセスの(is-1) 行にコピー

下のプロセスのis行が、上のプロセスの(ie+1)行にコピー

されていることを確認する。

# 参考：Fortran版

# 計算時間を計測する方法

```
real(DP) :: time0, time2
```

```
·  
·
```

```
call mpi_barrier( MPI_COMM_WORLD, ierr )  
time0 = mpi_wtime()
```

(計測する部分)

```
call mpi_barrier( MPI_COMM_WORLD, ierr )  
time1 = mpi_wtime()
```

(time1-time0 を出力する)

計測のための変数を**倍精度実数型**で宣言する。

MPI\_barrier関数で、計測開始の足並みを揃える。  
mpi\_wtime関数で開始時刻をtime0に設定

全プロセスで終了の足並みを揃える。  
mpi\_wtime関数で終了時刻をtime1に設定

time1-time0が計測した部分の計算時間となる。

mpi\_barrier(comm, ierr) : **バリア同期関数**

- ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)
- ◆ ierr: 戻りコード (整数型)

var = mpi\_wtime() : **倍精度実数**を返す関数

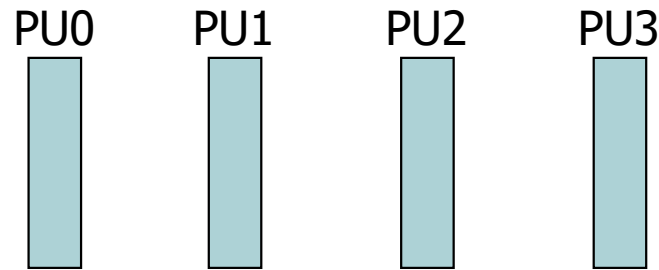
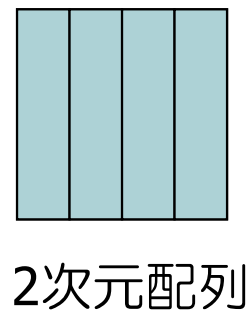
# 双方向通信：mpi\_sendrecv関数

```
mpi_sendrecv( sendbuff, sendcount, sendtype, dest, sendtag,  
recvbuff, recvcount, recvtype, source, recvtag,  
comm, status, ierr )
```

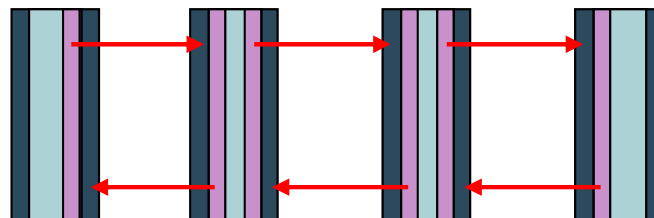
- ◆ sendbuff: 送信するデータのための変数名（先頭アドレス）
- ◆ sendcount: 送信するデータの数（整数型）
- ◆ sendtype: 送信するデータの型（MPI\_REAL, MPI\_INTEGERなど）
- ◆ dest: 送信する相手プロセスのランク番号
- ◆ sendtag : メッセージ識別番号. 送くるデータを区別するための番号
- ◆ recvbuff: 受信するデータのための変数名（先頭アドレス）
- ◆ recvcount: 受信するデータの数（整数型）
- ◆ recvtype: 受信するデータの型（MPI\_REAL, MPI\_INTEGERなど）
- ◆ source: 送信してくる相手プロセスのランク番号
- ◆ recvtag: メッセージ識別番号. 送られて来たデータを区別するための番号
- ◆ comm: コミュニケータ（例えば, MPI\_COMM\_WORLD）
- ◆ status: 受信の状態を格納するサイズMPI\_STATUS\_SIZEの配列（整数型）
- ◆ ierr: 戻りコード（整数型）

# 【発展】 MPI\_Sendrecvの行列への適用

- 2次元配列がブロック列分割されている。
- このとき、自分の両端の1列の要素を、隣接するプロセスの持つ領域の外側に受信用の領域を確保し、その領域にそれぞれ転送するプログラムを作る。



ブロック列分割



両隣のプロセスから1列を受信  
(受信用の領域を確保しておく)

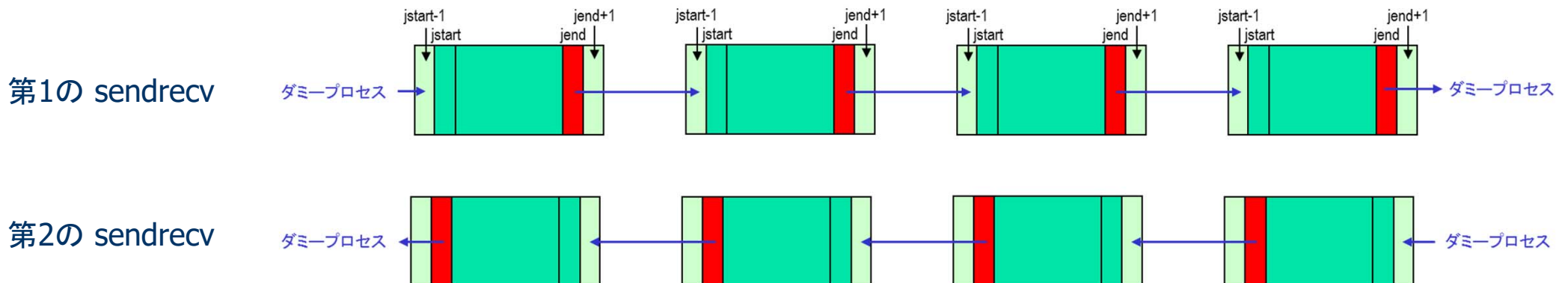
# MPI\_sendrecvの応用（続き）

## ■ 自プロセスの担当範囲を計算

- ◆ 自プロセスの担当範囲は  $js$  列  $\sim$   $je$  列

## ■ mpi\_sendrecv による送受信

- ◆ まず、右隣に  $je$  列を送り、左隣から  $js-1$  列に受信
- ◆ 次に、左隣に  $js$  列を送り、右隣から  $je+1$  列に受信
- ◆ 両端のプロセスは、ダミープロセス (MPI\_PROC\_NULL) と送受信するようになる。
  - MPI\_Sendrecv の source, dest に使うことが出来る。ダミープロセスを指定すると、実際にはどこからにもデータを送らないし、どこからもデータを受け取らない。





# プログラム sr\_matrix.f90 (続く)

```
program sr_matrix
  use mpi
  implicit none

  integer, parameter :: N=20
  real(8), dimension(0:N+1,0:N+1) :: u
  integer :: i, j, js, je

  integer :: nprocs, myrank, left, right, srtag, ierr
  integer, dimension(MPI_STATUS_SIZE) :: status

  call mpi_init( ierr )
  call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
  call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )

  js = (N/nprocs)* myrank + 1
  je = (N/nprocs)*(myrank+1)

  left = myrank-1
  if( myrank == 0 ) left = MPI_PROC_NULL
  right = myrank+1
  if( myrank == nprocs-1 ) right = MPI_PROC_NULL

  do i = 0, N+1
    do j = 0, N+1
      u(i,j) = 0.0
    end do
  end do
```

Recvで必要な配列変数の宣言

各プロセスの担当する列の範囲を計算

左右のプロセスのプロセス番号を計算  
(存在しない場合は MPI\_PROC\_NULL とする)

# プログラム sr\_matrix.f90 (続き)

```
do j = js, je
  do i = 0, n+1
    u(i,j) = (myrank+1)*10.0
  end do
end do
```

```
! From the left process to the right one
call mpi_sendrecv( ... )
```

```
! From the right process to the left one
call mpi_sendrecv( ... )
```

```
print '( "Rank=",i2," js:je=",i4,":",i4 )', myrank, js, je
print '(10f6.2)', (u(N/2,j),j=js-1,je+1)
```

```
call mpi_finalize( ierr )
```

```
end program sr_matrix
```

mpi\_sendrecv による送受信

正しく受信できたことを確認

# 演習10-5F : プログラムの完成と実行

- `sr_matrix.f90` は**未完成**である.
- `MPI_Sendrecv`の部分を完成させ, コンパイルして, 2, 4 プロセスで実行し, データの送受信が正しくできていることを確かめよ.  
`$ cp /tmp/Summer/M-3/sr_matrix.f90 ./`
- 実行結果 (出力順は, 以下とは異なる) .

```
Rank= 0, js:je=  1:  5
  0.00 10.00 10.00 10.00 10.00 10.00 10.00 20.00
Rank= 1, js:je=  6: 10
 10.00 20.00 20.00 20.00 20.00 20.00 30.00
Rank= 2, js:je= 11: 15
 20.00 30.00 30.00 30.00 30.00 30.00 40.00
Rank= 3, js:je= 16: 20
 30.00 40.00 40.00 40.00 40.00 40.00 0.00
```

左のプロセスのie列が, 右のプロセスの(is-1) 列にコピー

右のプロセスのis列が, 下のプロセスの(ie+1)列にコピー

されていることを確認する.