

# 計算科学演習 I

## MPIを用いた並列計算 (III)

神戸大学大学院システム情報学研究科

横川 三津夫

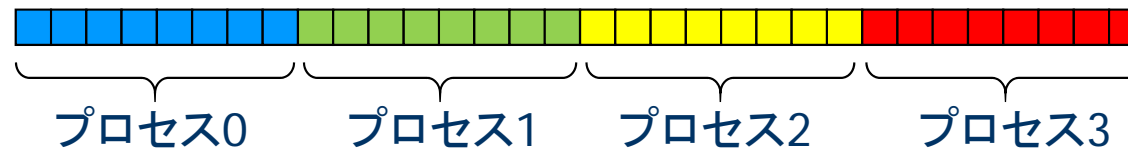
yokokawa@port.kobe-u.ac.jp

# 今週の講義の概要

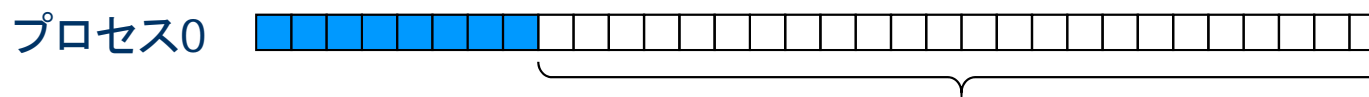
1. 前回課題の解説
2. 部分配列とローカルインデックス
3. ブロッキング関数とデッドロック
  - ◆ `mpi_sendrecv`
  - ◆ `mpi_isend`, `mpi_irecv`, `mpi_wait`

# 演習M2-2：ベクトルの正規化【再掲】

- $n$ 次元ベクトル  $x$  の第  $i$  要素を  $i$  とする ( $x(i) = i$ ) .
- このとき,  $x$  を正規化したベクトル  $x/\|x\|_2$  を求めるプログラムを作成せよ.
  - ◆  $\|x\|_2$  は  $x$  の各要素の2乗和の平方根である.
  - ◆ ベクトルは, ブロック分割で各プロセスに配置する.
- 各プロセスの担当する要素 ( $nprocs$  はMPIプロセス数)
  - ◆  $istart = (n/nprocs)*myrank + 1$
  - ◆  $iend = (n/nprocs)*(myrank+1)$



- 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

# 解答例

```
program dnorm2
use mpi
implicit none
integer, parameter :: n=1000
integer :: i,istart,iend
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
real(DP) :: sum_local, sum, error_local, error, const
real(DP) :: x(n)
integer :: nprocs,myrank,ierr
call mpi_init( ierr )
call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )
istart = (n/nprocs)*myrank + 1
iend   = (n/nprocs)*(myrank+1)
sum_local = 0.0d0
do i = istart, iend
  x(i)      = dble(i)
  sum_local = sum_local + x(i)*x(i)
end do
call mpi_allreduce( sum_local, sum, 1, MPI_REAL8, MPI_SUM, MPI_COMM_WORLD, ierr )
const = 1.0d0/sqrt(dble(n*(n+1)*(2*n+1))/6.0d0)
sum    = 1.0d0/sqrt(sum)
error_local = 0.0d0
do i = istart, iend
  x(i) = x(i)*sum
  error_local = error_local + abs( x(i) - i*const )
end do
call mpi_reduce( error_local, error, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, ierr )
if( myrank == 0 ) write(6,*) "Error = ", error
call mpi_finalize(ierr)
stop
end program
```

配列x(n)の宣言

配列xのうち、自分の担当する部分の要素をセット  
要素の2乗の部分和を計算

部分和の合計の平方根の逆数

自分の担当する要素を正規化する

# 解答に対するコメント

- `mpi_allreduce()` を使い、すべてのプロセスにおいて、ベクトルの大きさを持つことがポイント。
- 真の値との差を求めるのに、`i/sqrt(sum)`との差を計算していた。

$$i/\text{sqrt}(\text{sum}) - i/\text{sqrt}(\text{real}(n*(n+1)*(2*n+1)/6))$$

- ◆ たまたま  $x(i) = i$  としたので、これでも良いが、ベクトルの正規化を問題にしており、ベクトル  $x(i)$  はいつも決まって値ではないので、配列としてプログラムを作って欲しかった。

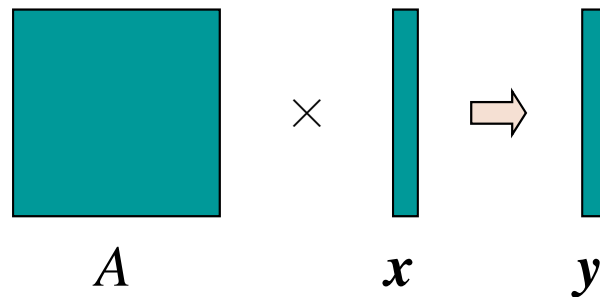
$$x(i)/\text{sqrt}(\text{sum}) - i/\text{sqrt}(\text{real}(n*(n+1)*(2*n+1)/6))$$

- プログラムが正しいかどうかは、今回のケースでは  $x(i)$  が計算できるので、真値との差が0.0であることを確認する。

# リダクション演算の応用：行列ベクトル積

## ■ 問題

- ◆  $n \times n$  行列  $A$  の第  $(i, j)$  要素を  $i + j$  とする ( $a_{i,j} = i + j$ ) .
- ◆  $n$  次元ベクトル  $x$  の第  $i$  要素を  $i$  とする ( $x_i = i$ ) .
- ◆ このとき,  $y = Ax$  を計算する.



## ■ プログラムのカーネル（主要な部分）

- ◆ ベクトル  $y$  の要素を1個ずつ計算する.
- ◆  $y$  の第  $i$  要素は,  $A$  の第  $i$  行とベクトル  $x$  との内積

$$y_i = \sum_j a_{i,j} x_j$$

# 逐次プログラム M-5 (mv\_s.f90)

```
program mv
implicit none
integer, parameter :: n=100
integer :: i, j
real(kind=8), dimension(n,n) :: a
real(kind=8), dimension(n) :: x, y
real(kind=8) :: error, ans
do i = 1, n
  x(i) = i
end do
do i = 1, n
  do j = 1, n
    a(i,j) = dble(i+j)
  end do
end do
do i = 1, n
  y(i) = 0.0d0
  do j = 1, n
    y(i) = y(i) + a(i,j)*x(j)
  end do
end do
error = 0.0d0
do i = 1, n
  ans = dble( i*n*(n+1)/2 + n*(n+1)*(2*n+1)/6)
  error = error + abs( y(i) - ans )
end do
print *, 'error =', error
end program mv
```

x(i), a(i,j) の初期設定

y=Axの計算

結果の確認

abs() は絶対値関数

## 演習M3-1 : M-5プログラムの実行

- mv\_s.f90 をコンパイルして, 実行せよ

```
$ cp /tmp/cpmpi/M-5/mv_s.f90 ./
$ f95 -o mv_s mv_s.f90
$ ./mv_s
```

- 結果が正しいことを確認せよ.

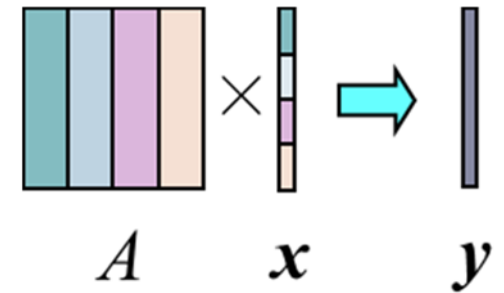
```
$ ./mv_s
error = 0.000000000000000000
```



# 行列ベクトル積の並列化

## ■ 行列とベクトルの分割

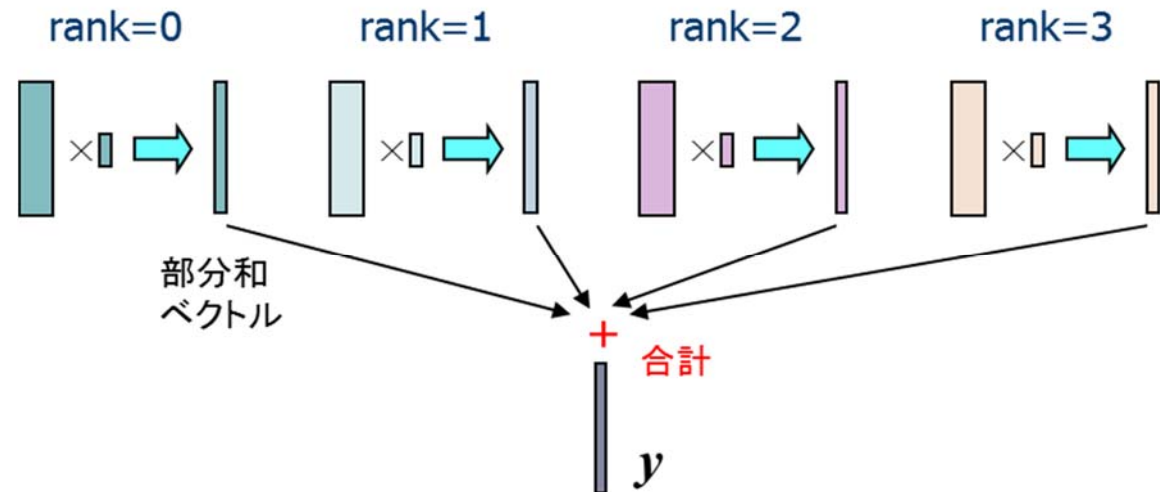
- ◆  $A$  は列方向に短冊形に分割（ブロック列分割）する。
- ◆  $x$  は行方向にブロック分割する。



## ■ プログラムの方針

- ◆ 各プロセスが持つ自分の要素のみを使って、部分和ベクトルを計算
- ◆ 部分和ベクトルを `mpi_reduce`関数によりプロセス0で集計する。

$$\begin{aligned} y_i &= \sum_{\text{rank}=0} a_{i,j} x_j + \sum_{\text{rank}=1} a_{i,j} x_j \\ &+ \sum_{\text{rank}=2} a_{i,j} x_j + \sum_{\text{rank}=3} a_{i,j} x_j \end{aligned}$$



# 演習M3-2：M-5 (mv\_s.f90) の並列化 (M-6)

## ■ プログラム書き換えの方針

◆ MPIの定義, 初期化, 終了処理を忘れないこと.

◆ 各プロセスの計算範囲を求める

- $istart = (n/nprocs) * myrank + 1$
- $iend = (n/nprocs) * (myrank + 1)$

◆  $A$ ,  $x$  について, 各プロセスが担当する部分のみ初期化する.

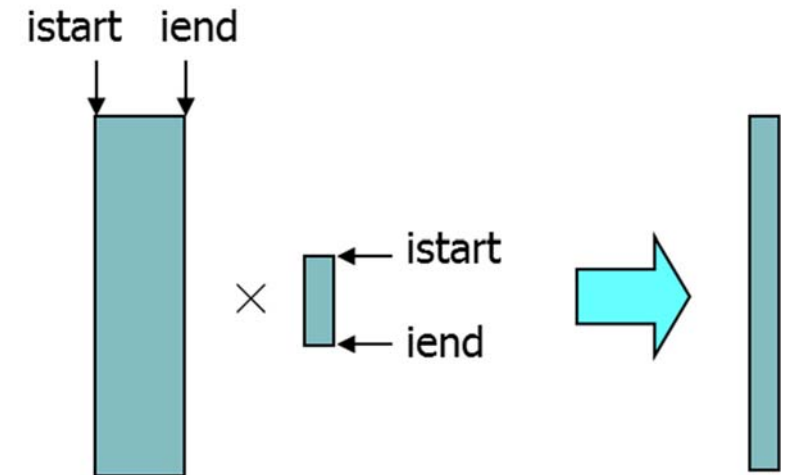
- $A$  : 第  $istart$  列 ~ 第  $iend$  列
- $x$  : 第  $istart$  要素 ~ 第  $iend$  要素

◆ 部分和ベクトルは, 各プロセスの持つ要素のみを使って計算

- 部分和ベクトルは, 別の配列 (例えば  $y\_tmp$ ) を用いる.

◆ 部分和ベクトルの合計

- `mpi_reduce` 関数により, ランク0のプロセスで, 配列  $y\_tmp$  の合計を配列  $y$  に入れる.
- `mpi_reduce` 関数の第3引数 (`count`) に注意 (前回資料 29ページ)
- 結果は,



## 演習M3-2：続き

- $n=1000$ として、プロセス数1, 2, 4, 及び8と変化させて実行させ、結果が正しいことを確認せよ。
- そのときの計算時間の変化を調べよ。
  - ◆ 初期設定、結果の確認部分は、計測範囲に含めないこと。
  - ◆ プロセス数 ( $n$ ) , 計算時間 ( $T_n$ ) , 加速率 ( $S_n = T_1/T_n$ ) をまとめる。

$n$	$T_n$	$S_n$
1	XXXXXXXX	1.000
2	XXXXXXXX	XXXXXXXX
4	XXXXXXXX	XXXXXXXX
8	XXXXXXXX	XXXXXXXX

# プログラムの問題点：メモリの無駄

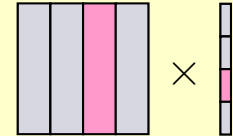
```
istart = (n/nprocs)*myrank + 1
iend   = (n/nprocs)*(myrank+1)
do j = istart, iend
  x(j) = j
end do
do i = 1, n
  do j = istart, iend
    a(i,j) = dble(i+j)
  end do
end do
do i = 1, n
  yp(i) = 0.0d0
  do j = istart, iend
    yp(i) = yp(i) + a(i,j)*x(j)
  end do
end do
call mpi_reduce(yp, y, n, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
if( myrank == 0 ) then
  err = 0.0d0
  do i = 1, n
    ans = dble(i*n*(n+1)/2+n*(n+1)*(2*n+1)/6)
    err = err + abs( y(i) - ans )
  end do
  print *, 'error =', err
end if

call mpi_finalize(ierr)
end program mv
```

自プロセスの担当する範囲を計算

A, x のうち、自プロセスの担当する範囲のみを初期化

例えば、rank=2では、ピンクの部分だけしか使っていないので、メモリがもったいない。



部分和ベクトル yp の計算

yp を合計して y を得る

プロセス0で結果をチェック

# 部分配列とローカルインデックス

## ■ 部分配列の利用

- ◆ プログラムM-5を並列化したプログラムM-6では、各プロセスが  $A$ ,  $x$  全体を格納できる配列を確保し、そのうち自分の担当部分のみに値を入れて使用している。
- ◆ 実際に使用する範囲のみを確保すれば、メモリを節約できる。
  - $A$  : 第 istart 列 ~ 第 iend 列
  - $x$  : 第 istart 要素 ~ 第 iend 要素
- ◆ これを実現するには、**allocatable 配列**を利用すればよい

## ■ ローカルインデックス

- ◆ Fortranでは、allocate 文により、 $x$ のインデックスを istart から始まるようにできる。
  - C言語の malloc() と、メモリの動的確保という点では、同等の関数
- ◆ これにより、プログラムをほとんど変えずに部分配列を利用可能
- ◆ サイクリック分割等の場合は、やや複雑なインデックス変換が必要

## 演習M3-3: 部分配列とローカルインデックス (提出課題)

- allocate文を使って，メモリを節約するようにM-6を書き換え，実行し，結果を確認せよ。

### ■ ヒント

```
real(DP), dimension(:,:), allocatable :: a
real(DP), dimension(:), allocatable :: x

allocate( a(n,istart:iend) )
allocate( x(istart:iend) )
```

# MPIプログラム M-7 : デッドロック

```
program deadlock
use mpi
implicit none

integer, parameter :: n=10
double precision :: a0(n), a1(n)
integer :: nprocs, myrank, ierr
integer :: istat(MPI_STATUS_SIZE)
call mpi_init( ierr )
call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )
if( myrank == 0 ) then
  a0 = 1.0
else
  a1 = 2.0
endif

if( myrank == 0 ) then
  call mpi_send( a0, n, MPI_REAL8, 1, 100, MPI_COMM_WORLD, ierr )
  call mpi_recv( a1, n, MPI_REAL8, 1, 200, MPI_COMM_WORLD, istat, ierr )
else
  call mpi_send( a1, n, MPI_REAL8, 0, 200, MPI_COMM_WORLD, ierr )
  call mpi_recv( a0, n, MPI_REAL8, 0, 100, MPI_COMM_WORLD, istat, ierr )
end if

call mpi_finalize( ierr )
end program deadlock
```

# 演習M3-4 デッドロックを確認せよ

## ■ プログラム M-7をコピーし，以下のことを確認せよ。

- /tmp/cpmi/M-7/deadlock.f90

◆ プログラム5行目の nを，10，100としたときに，結果がどうなるか確認せよ。プロセス数は2として実行する。

- 注意) ジョブスクリプトの #PJM -L "elapsed=00:00:xx" の xx は大きくしない。

◆ M-7において，send, recvの順番を次のように変えて実行し，結果がどうなるか確認せよ。

### 【変更1】

```
if( myrank == 0 ) then
  call mpi_recv( )
  call mpi_send( )
else
  call mpi_recv( )
  call mpi_send( )
end if
```

### 【変更2】

```
if( myrank == 0 ) then
  call mpi_send( )
  call mpi_recv( )
else
  call mpi_recv( )
  call mpi_send( )
end if
```



# 実行結果は. . .

- 次のシステム・メッセージが出るケースがある.

```
jwe0017i-u The program was terminated with signal number SIGXCPU.
```

⇒ CPUの時間制限を越えた.

⇒ ジョブが指定した時間内に終わらなかった.

- ジョブが終了するケースと、そうでないケースがある.

→ 何故か？

# ブロッキング関数とデッドロック

- `mpi_send()`, `mpi_recv()` は**ブロッキング関数**
- **ブロッキング関数の動作（実装による）**
  - ◆ 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，呼出し元に制御が戻らない。
    - `mpi_send`の場合，すべてのメッセージがMPI送信バッファに書き込みが終わった段階で，呼出し元に制御が戻る場合もある（後は，下位レイヤの通信プログラムに制御を任せてしまう）。
    - `mpi_recv`は，すべてのメッセージを受信するまで，呼出し元に制御が戻らない。
  - ◆ 次の行に制御が移らない。
- **ブロッキング関数は，その関数の処理が終了するまで，次の処理に進まない。**

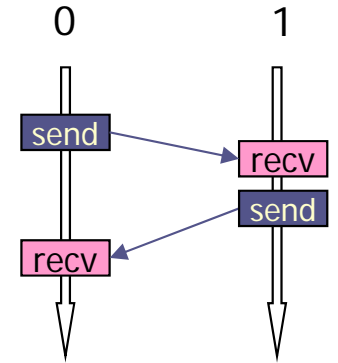
# 演習M3-4の解説

- ケース1 : send-recv : send-recv かつ  $n=10$ 
  - ◆ `mpi_send`で送るメッセージのバイト数が小さいため、システムのバッファにすべて書き込めたので、制御が戻り、次の行が実行された、と考えられる。
  - ◆ `mpi`ライブラリの実装に依る。
- ケース2 : send-recv : send-recv かつ  $n=100$ 
  - ◆ `mpi_send`で送るメッセージのバイト数が大きく、すべてのメッセージがMPI通信バッファに書き込めず、相手の`recv`の開始を待っているが、相手も`mpi_send`を実行していて、受取ってくれないので、`deadlock`となった。
- ケース3 : recv-send: recv-send
  - ◆ どちらのプロセスも`mpi_recv`関数を実行し、データの到着を待っているが、お互い`mpi_send`が実行できないので、そこで待っている間にCPUの制限時間に達した。
- ケース4 : send-recv: recv-send
  - ◆ 送受信の順番が、シリアライズされたため、上手く実行できた。

# デッドロックの回避方法

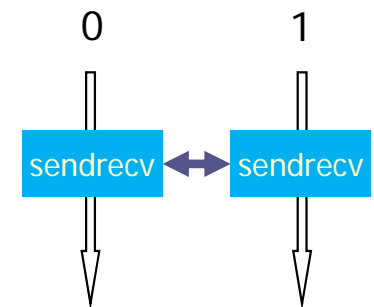
## ① 送受信の順序のシリアライズ（ケース4）

- ◆ プロセス0： 送信してから受信
- ◆ プロセス1： 受信してから送信



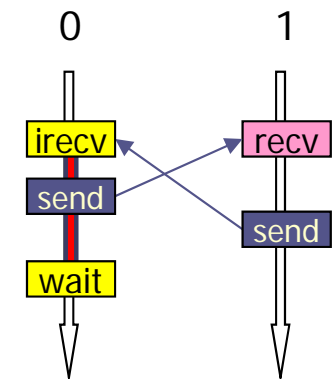
## ② mpi\_sendrecv の利用

- ◆ mpi\_send と mpi\_recv をまとめて行うルーチン
- ◆ デッドロックは生じない
- ◆ 1回の送受信の時間で済む
- ◆ 送信相手と受信相手が異なってもよい



## ③ ノンブロッキング関数の利用

- ◆ mpi\_isend
- ◆ mpi\_irecv
- ◆ ノンブロッキング関数では、制御が呼出し元にすぐに戻るため、転送する変数に関係ない他の作業をすることが出来る。
  - 特に、通信と計算が同時に動作する
- ◆ mpi\_wait で、関数の終了を確認する必要がある。



# 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)

# ノンブロッキング送信関数 mpi\_isend

```
mpi_isend( buff, count, datatype, dest, tag, comm, request, ierr )
```

※ ランク番号destのプロセスに、変数buffの値を送信する。

- ◆ buff: 送信するデータの変数名（先頭アドレス）
- ◆ count: 送信するデータの数（整数型）
- ◆ datatype: 送信するデータの型
  - MPI\_INTEGER, MPI\_REAL, MPI\_DOUBLE\_PRECISIONなど
- ◆ dest: 送信先プロセスのランク番号
- ◆ tag: メッセージ識別番号。送る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）
- ◆ request: リクエスト識別番号（整数型）
- ◆ ierr: 戻りコード（整数型）

# ノンブロッキング受信関数 `mpi_irecv`

```
mpi_irecv( buff, count, datatype, source, tag, comm, request, ierr )
```

※ ランク番号`source`のプロセスから送られたデータを, 変数`buff`に格納する.

- ◆ `buff`: 受信するデータのための変数名 (先頭アドレス)
- ◆ `count`: 受信するデータの数 (整数型)
- ◆ `datatype`: 受信するデータの型
  - `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`など
- ◆ `source`: 送信してくる相手プロセスのランク番号
- ◆ `tag`: 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`)
- ◆ `request`: リクエスト識別変数 (整数型)
- ◆ `ierr`: 戻りコード (整数型)

# 待ちの関数 `mpi_wait`

```
mpi_wait( request, status, ierr )
```

※ リクエスト識別変数`request`に対応した通信関数の終了を確認する。ブロッキング関数

- ◆ **request:** リクエスト識別変数 (整数型)
  - ◆ 対応する`mpi_isend`, または`mpi_irecv`のリクエスト識別番号と一致させる
- ◆ **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)



## 演習M3-5, M3-6 (提出課題)

- プログラム M-7を, 次の2つの方法で, deadlockしないプログラムにせよ.
  - ◆ `mpi_irecv`, `mpi_wait`を使う. (M3-5)
    - 20ページの③のとおり.
  - ◆ `mpi_sendrecv`を使う. (M3-6)
    - プロセス0, プロセス1は, それぞれ送る変数が違うことに注意.
- データがきちんと転送されていることを確認すること.

# 課題の提出方法と提出期限

## ■ 演習M3-3, M3-5, M3-6の提出方法

① それぞれプログラムと実行結果を一つのファイルにまとめる.

```
$ cat program.f90 > report-xx.txt
```

```
$ cat xxxxx.onnnnn >> report-xx.txt
```

② 以下の方法で, メールにより提出する.

```
$ cat report-xx.txt | mail -s "3-n:アカウント" yokokawa@port.kobe-u.ac.jp
```

Note) **アカウント**は自分のログインID

番号 (3-n) は, 演習課題番号

■ 期限 : **6月23日 (火) 午後5時**