# MPIによる並列化実装
## ～ハイブリッド並列～

八木学
（理化学研究所 計算科学研究機構）

**謝辞**
松本洋介氏（千葉大学）

KOBE HPC Spring School 2018
2018年3月15日　神戸大学計算科学教育センター

# MPIとは

- Message Passing Interface
- 分散メモリのプロセス間の**通信規格**（API）
- SPMD（Single Program Multi Data）が基本
  - 各プロセスが「同じことをやる」が「データが違う」
- 『mpich』や『openmpi』などの実装が有名

＊OpenMPとは違います

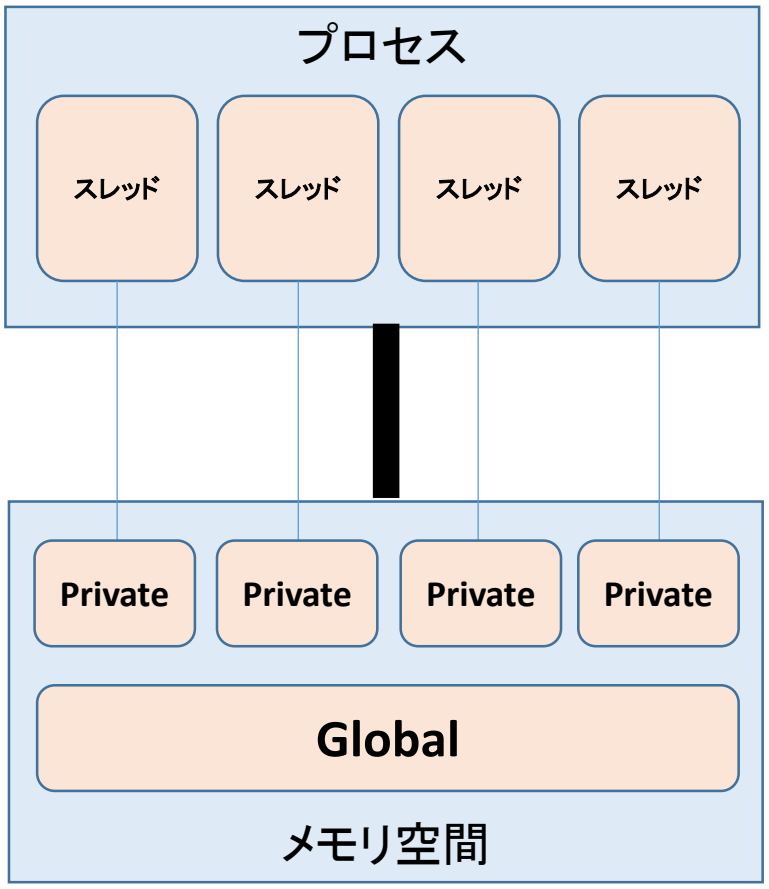
- **以前の HPC Summer School の資料も使えます**

http://www.eccse.kobe-u.ac.jp/simulation_school/

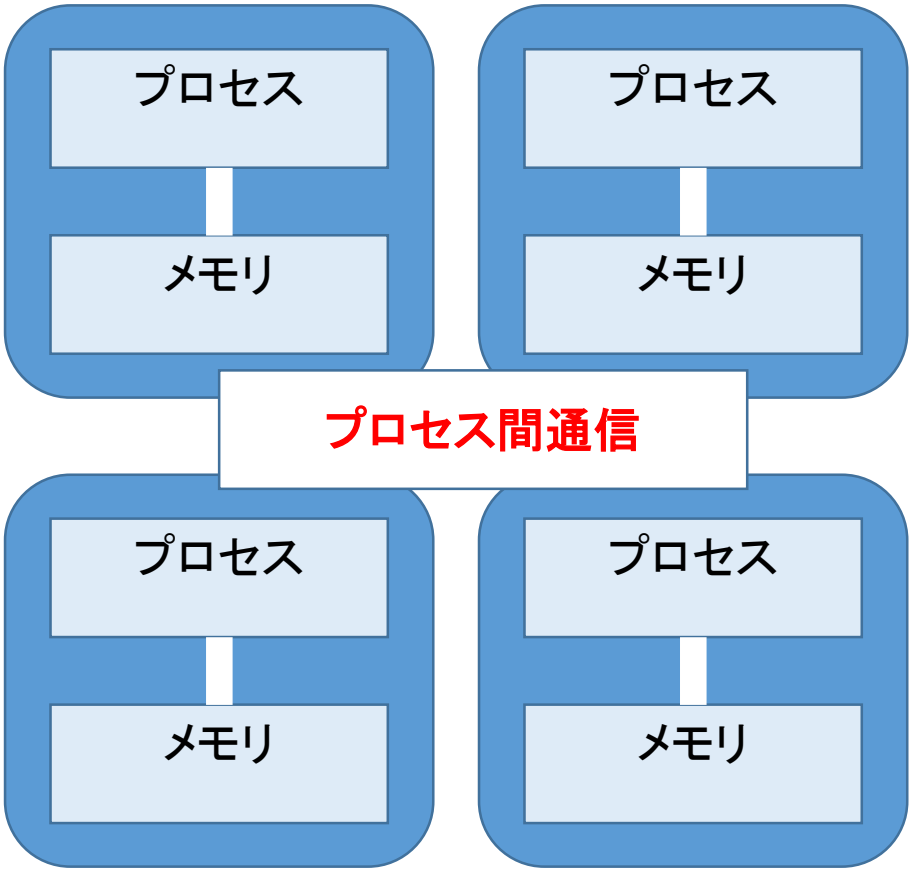
**＊本日の演習では、本当に最小限の命令しか使いません。**

# スレッド並列とプロセス並列

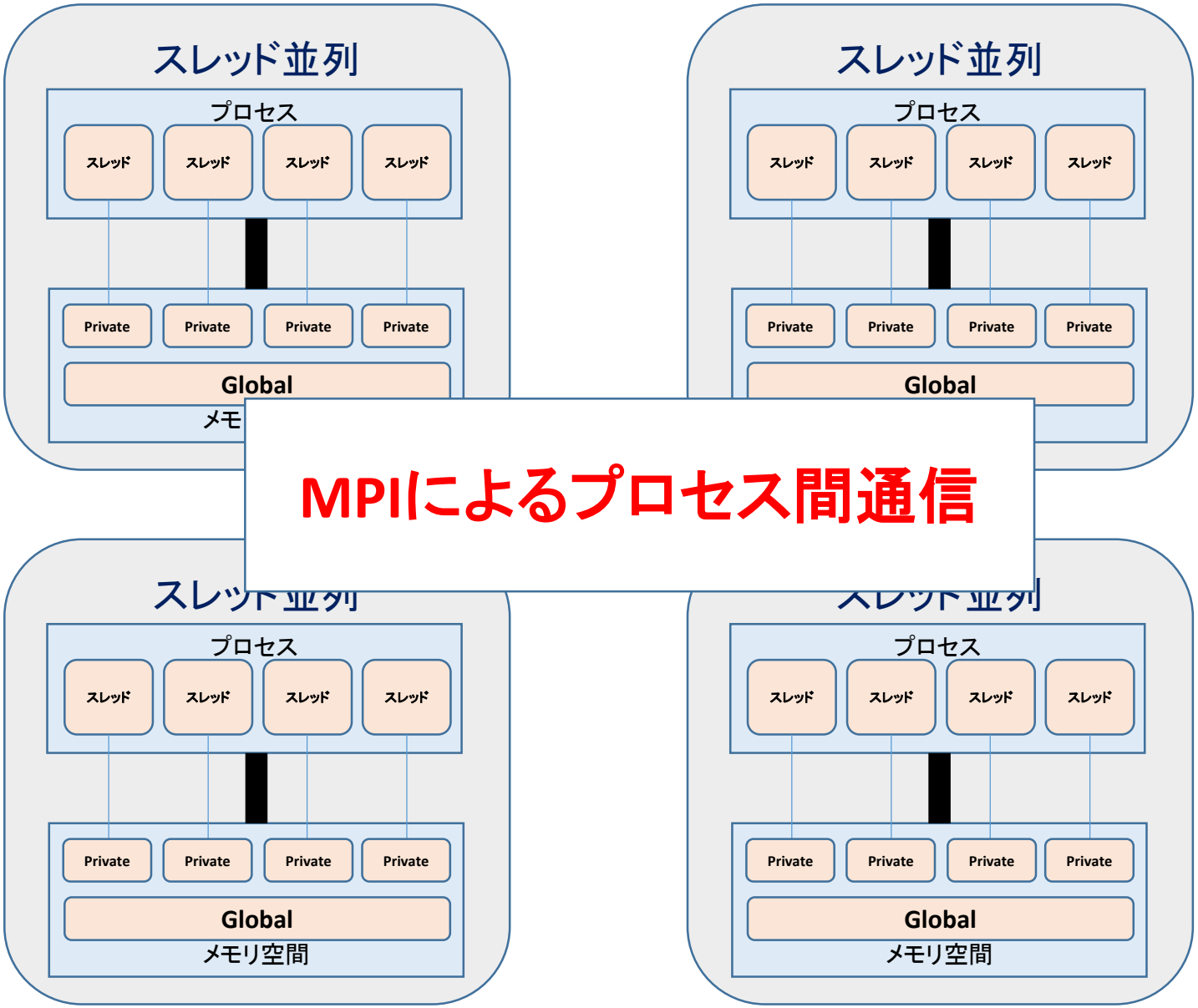
## スレッド並列
**OpenMP、自動並列化**



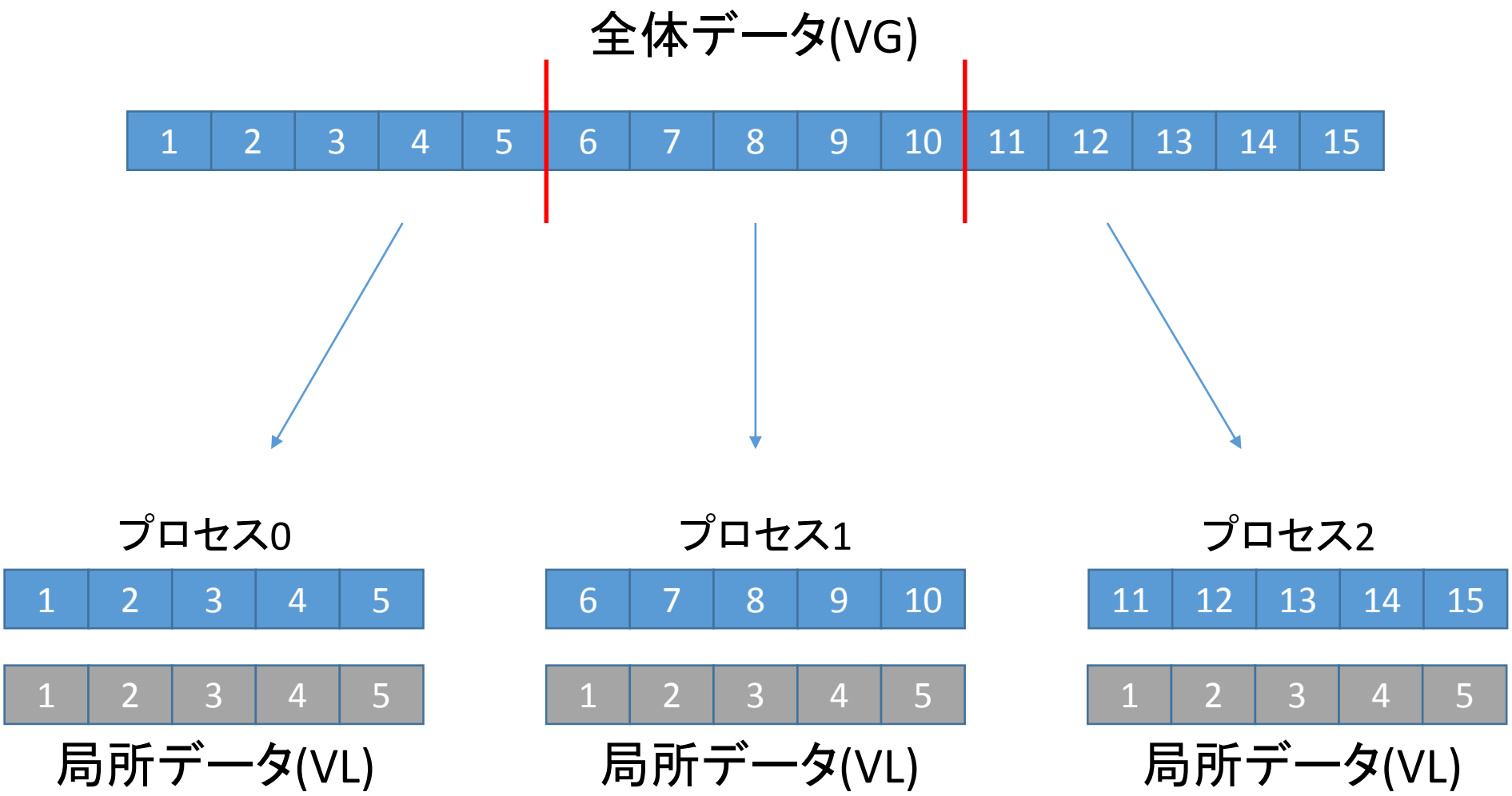

## プロセス並列
**MPI**



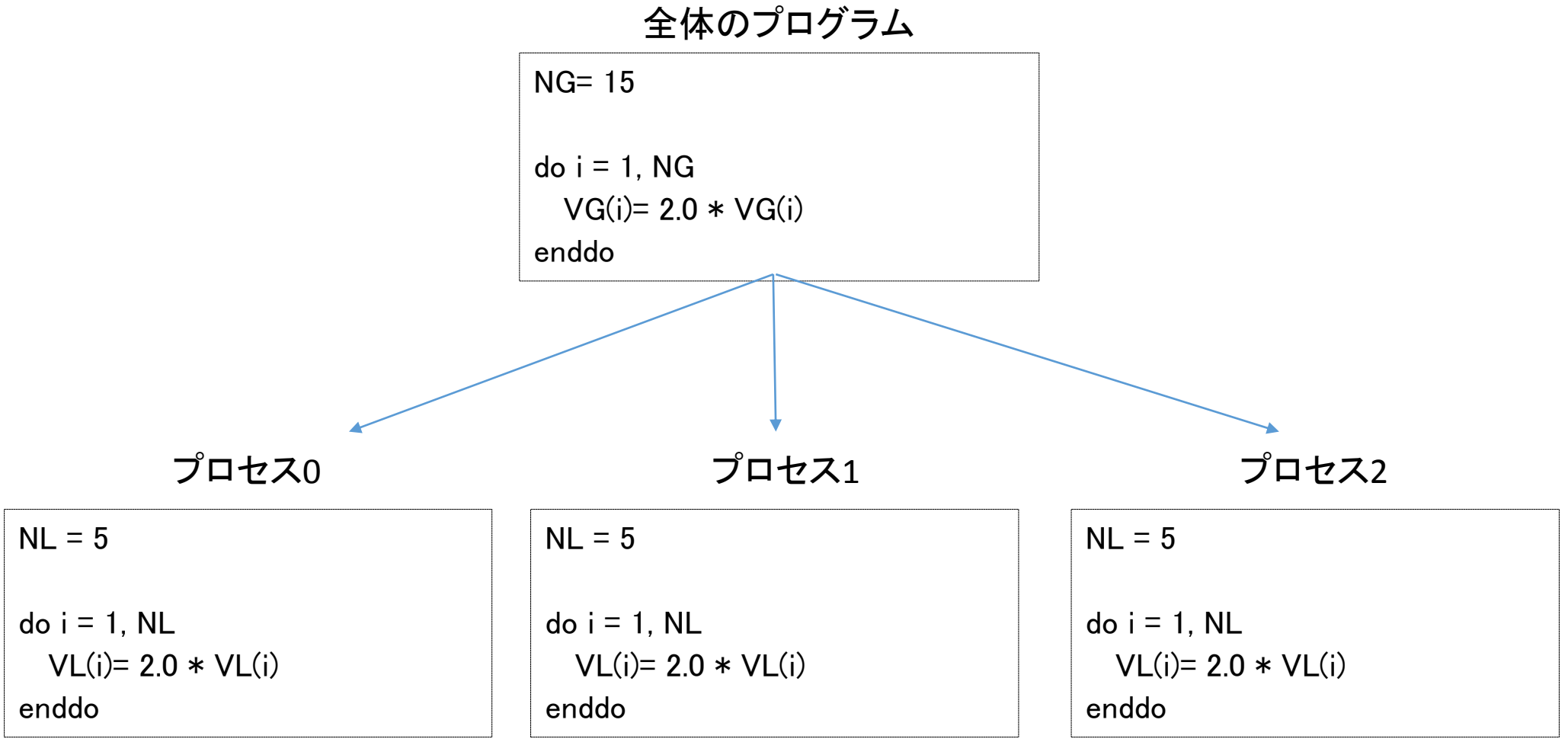
# ハイブリッド並列


スレッド並列
プロセス
スレッド
スレッド
スレッド
スレッド
Private
Private
Private
Private
Global
メモ
スレッド並列
プロセス
スレッド
スレッド
スレッド
スレッド
Private
Private
Private
Private
Global
MPIによるプロセス間通信
プロセス
スレッド
スレッド
スレッド
スレッド
Private
Private
Private
Private
Global
メモリ空間
プロセス
スレッド
スレッド
スレッド
スレッド
Private
Private
Private
Private
Global
メモリ空間

# SPMDの考え方


全体データ(VG)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
プロセス0
1 2 3 4 5
1 2 3 4 5
局所データ(VL)
プロセス1
6 7 8 9 10
1 2 3 4 5
局所データ(VL)
プロセス2
11 12 13 14 15
1 2 3 4 5
局所データ(VL)

# SPMDの考え方




各プロセスで実行するコードは同じだが、データが異なる

# Fortran/Cの違い

**・基本的にインタフェースはほとんど同じ**

　– Cの場合，「MPI_Comm_size」のように「MPI」は大文字、「MPI_」のあとの最初の文字は大文字、以下小文字

・Fortranはエラーコード（ierr）の戻り値を引数の最後に指定する必要がある

・Cは変数の特殊な型がある.

　– MPI_Comm, MPI_Datatype, MPI_Op etc.

**・最初に呼ぶ「MPI_Init」だけは違う**

　– <Fortran> call MPI_INIT (ierr)

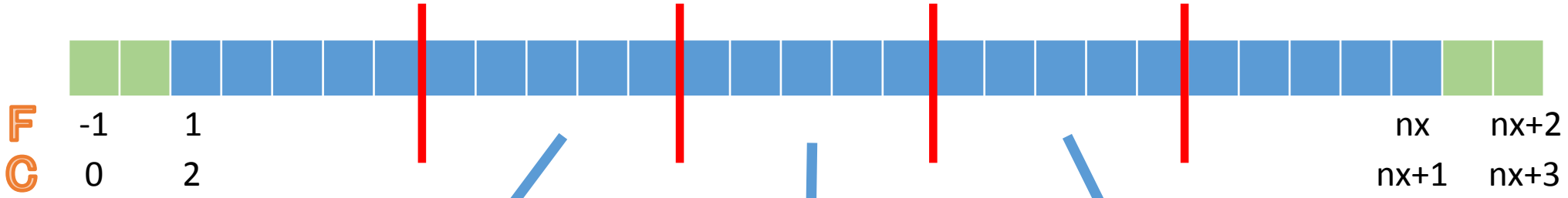
　– <C> MPI_Init (int *argc, char ***argv)

# 配列計算（1次元）のプロセス分割

**全体の配列**

F　F(-1:nx+2)
C　F[nx+4]

非並列計算の場合、計算領域は 1 ~ nx
差分計算に左右2グリッドずつ必要なため、-1 ~ nx+2 の間で領域を確保







**各プロセスの配列**

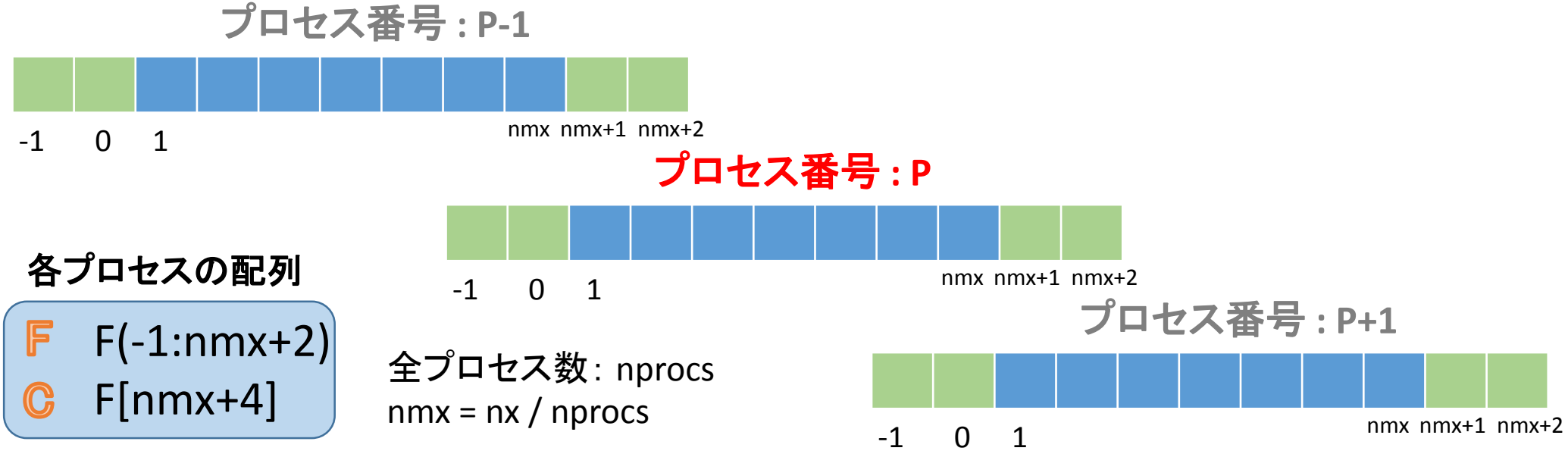
F　F(-1:nmx+2)
C　F[nmx+4]

全プロセス数：nprocs
nmx = nx / nprocs

# 配列計算（1次元）のプロセス分割

・各プロセスの変数は基本的に独立であり、他のプロセスの持つ変数を参照することはできない。（プロセス間通信が必要）

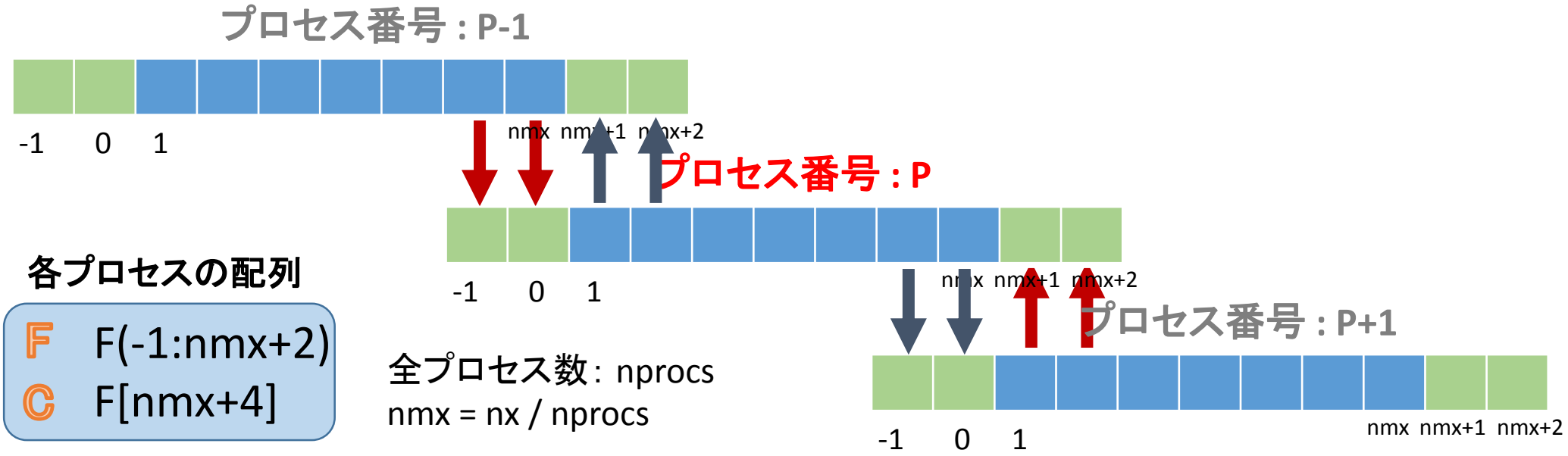
・プロセス P の F(-1), F(0), F(nmx+1), F(nmx+2) は、プログラム中で計算はしていないが差分計算の上で値は必要。



# 配列計算（1次元）のプロセス分割

-- 左隣のプロセス (P-1) の持つ F(nmx-1) と F(nmx) の値を受け取り、F(-1) と F(0) に格納
-- 右隣のプロセス (P+1) の持つ F(1) と F(2) の値を受け取り、F(nmx-1) と F(nmx) に格納

-- 右側のプロセス (P+1) に F(nmx-1) と F(nmx) の値を送信
-- 左側のプロセス (P-1) に F(1) と F(1) の値を送信




**各プロセスの配列**

F　F(-1:nmx+2)
C　F[nmx+4]

全プロセス数 : nprocs
nmx = nx / nprocs

# 物理的な境界条件

## 周期境界

--プロセス番号が 0（左端のプロセス）の場合、左のプロセスを nprocs-1 として扱う
-- プロセス番号が nprocs-1（右端のプロセス）の場合、右のプロセスを 0 として扱う

## 自由端、固定端

-- プロセス番号が 0（左端のプロセス）の場合、F(-1)とF(0)に処理を入れる
-- プロセス番号が nprocs-1（右端のプロセス）の場合、F(nmx+1)とF(nmx+2)に処理を入れる

```
If (myrank == 0 ) then
  F(-1) = F(1)
  F(0)  = F(1)
else if (myrank == nprocs-1) then
  F(nmx+1) = F(nmx)
  F(nmx+2) = F(nmx)
endif
```

自由端の場合

```
iup = myrank+1
idown = myrank - 1

if (myrank == 0 ) then
  idown = nprocs - 1
else if (myrank == nprocs-1) then
  iup = 0
endif
```

周期境界の場合

myrank: 自分のプロセス番号
iup : 右のプロセス番号
idown : 左のプロセス番号

# 配列計算（2次元）のプロセス並列

**全体の配列**

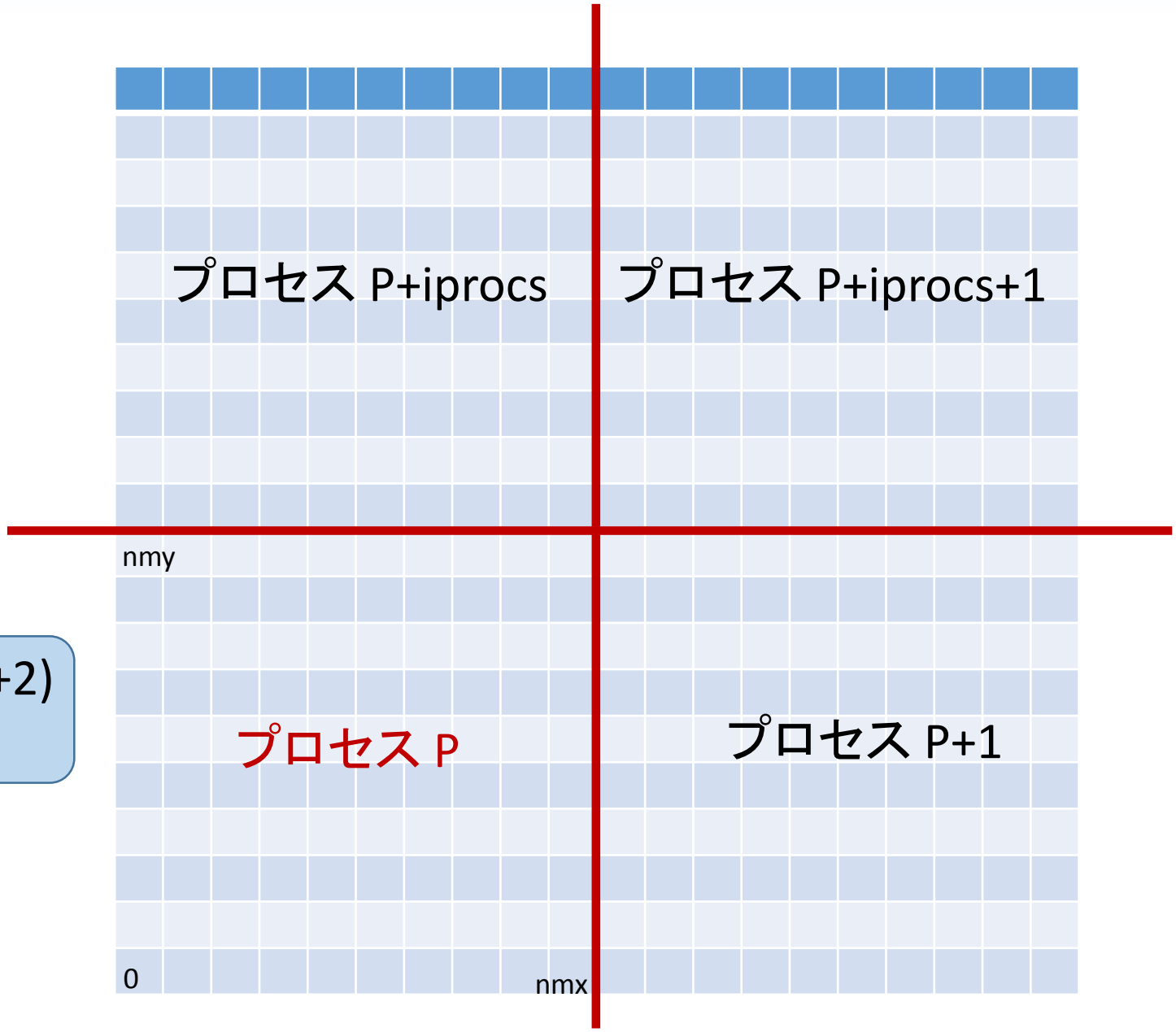
F　F(-1:nx+2, -1:ny+2)
C　F[ny+4][nx+4]

X方向の分割数：iprocs
Y方向の分割数：jprocs
全体のプロセス：
nprocs = iprocs * jprocs

**各プロセスの配列**

F　F(-1:nmx+2, -1:nmy+2)
C　F[nmy+4][nmx+4]

nmx = nx/iprocs
nmy = ny/jprocs



# 配列計算（2次元）のプロセス並列

**全体の配列**

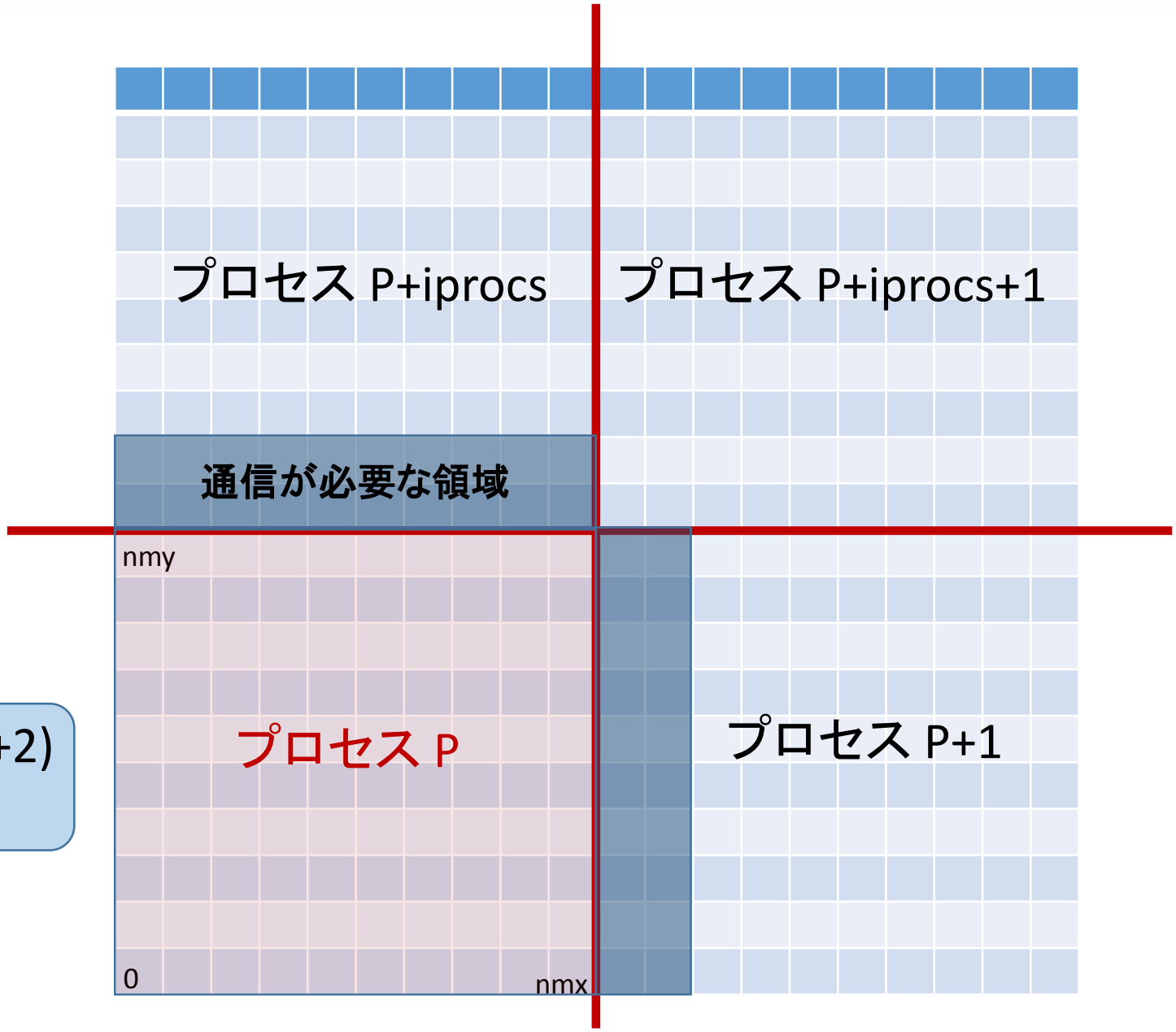
F　F(-1:nx+2, -1:ny+2)
C　F[ny+4][nx+4]

X方向の分割数：iprocs
Y方向の分割数：jprocs
全体のプロセス：
nprocs = iprocs * jprocs

**各プロセスの配列**

F　F(-1:nmx+2, -1:nmy+2)
C　F[nmy+4][nmx+4]

nmx = nx/iprocs
nmy = ny/jprocs



# 配列計算（2次元）のプロセス並列

## 2次元以上の配列を扱う上での注意点

-- MPIの仕様上、データを通信する場合はメモリの先頭アドレスと
幅を指定する必要がある（後述）

-- 配列の縦方向は、一見連続に見えてもメモリの上では不連続

-- メモリ上不連続なデータを送信する場合、新たに配列を定義し、
連続な配列に置き換えて送信する

# MPIの基本的な機能

F

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

C

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int n, myrank, numprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  printf ("Hello World %d¥n", myrank);
  MPI_Finalize();
}
```

**mpif.h / mpi.h**

環境変数デフォルト値

Fortran90ではuse mpiでも可

**MPI_INIT**

MPIプロセス開始

**MPI_COMM_SIZE**

プロセス数取得

mpiexec -np XX <prog>

**MPI_COMM_RANK**

自分のプロセス番号を取得

**MPI_FINALIZE**

MPIプロセス終了

# MPIの基本的な機能

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

```
#!/bin/sh
#PJM =L "node=1"                    ノード数
#PJM =L "elapse=00:00:30"           実行時間
#PJM =L "rscgrp=small"実行キュー名
#PJM -j
#PJM -o "hello.lst"                 標準出力ファイル名
#PJM --mpi "proc=4"                 MPIプロセス数

mpiexec ./a.out                     実行ファイル名
```

- この例では4つのプロセスが立ち上がる（"proc=4"）
  - 同じプログラムが4つ流れる
  - データの値（myrank）を書き出す
- 4つのプロセスは同じことをやっているが、データ として取得したプロセスID （myrank）は異なる
- 結果として各プロセスは異なった出力をしていることになる

SPMD（Single Program Multi Data）

# mpif.h ／ mpi.h

F

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

C

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int n, myrank, numprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  printf ("Hello World %d¥n", myrank);
  MPI_Finalize();
}
```

・MPIに関連した様々なパラメータおよび初期値を記述

・変数名は「MPI_」で始まっている

・ここで定められている変数はMPIサブルーチンの引数として使用する以外は陽に値を変更してはいけない

・ユーザーは「MPI_」で始まる変数を独自に設定しないのが無難

# MPI_INIT / MPI_Init

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

F

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int n, myrank, numprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  printf ("Hello World %d¥n", myrank);
  MPI_Finalize();
}
```

C

・MPIを起動する。他のMPIサブルーチンより前にコールする必要がある（必須）

・全実行文の前に置くことを勧める

＜Fortran＞
call MPI_INIT (ierr)
 - ierr : エラーコード（integer）

＜C＞
MPI_Init (&argc, &argv)

# MPI_FINALIZE / MPI_Finalize

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int n, myrank, numprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  printf ("Hello World %d¥n", myrank);
  MPI_Finalize();
}
```

・MPIを終了する。他の全てのMPIサブルーチンより後にコールする必要がある（必須）

・全実行文の後に置くことを勧める

・これを忘れると大変なことになる

＜Fortran＞
call MPI_FINALIZE (ierr)
- ierr : エラーコード（integer）

＜C＞
MPI_Finalize()

# MPI_COMM_SIZE / MPI_Comm_Size

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

F

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int n, myrank, numprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  printf ("Hello World %d¥n", myrank);
  MPI_Finalize();
}
```

C

・コミュニケーター「comm」で指定されたグループに含まれるプロセス数の合計が「size」に返る。

＜Fortran＞

**call MPI_COMM_SIZE (comm, size, ierr)**

- comm : コミュニケータ
- size : commのプロセス数の合計
- ierr : エラーコード

＜C＞

**MPI_Comm_size (comm, size)**

- comm : コミュニケータ
- size : commのプロセス数の合計

# MPI_COMM_RANK / MPI_Comm_Rank

F

```
include 'mpif.h'
integer :: PETOT, myrank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, PETOT, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

write (*,'(a,2i8)') 'Hello World', myrank, PETOT

call MPI_FINALIZE(ierr)

stop
end
```

C

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int n, myrank, numprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  printf ("Hello World %d¥n", myrank);
  MPI_Finalize();
}
```

・コミュニケーター「comm」で指定されたグループにおけるプロセスIDが「rank」に返る。

＜Fortran＞

**call MPI_COMM_RANK (comm, rank, ierr)**

- comm : コミュニケータ
- size : commにおけるプロセスID
- ierr : エラーコード

＜C＞

**MPI_Comm_RANK (comm, size)**

- comm : コミュニケータを指定
- size : commのプロセス数の合計

# MPIの基本命令

MPI_ISEND：データを送信（ノンブロッキング通信）

MPI_IRECV：データを受信（ノンブロッキング通信）

MPI_WAITALL：ノンブロッキング通信の確認

MPI_SEND：データを送信（ブロッキング通信）

MPI_RECV：データを受信（ブロッキング通信）

MPI_SENDRECV：MPI_SEND+MPI_RECV

# MPI_ISEND / MPI_Isend

call MPI_ISEND
(sendbuf, count, datatype, dest, tag, comm, request, ierr)

F

MPI_Isend
(sendbuf, count, datatype, dest, tag, comm, request)

C

・送信バッファ「sendbuf」内の、連続した「count」個の送信メッセージをタグ「tag」 を付けて、コミュニケータ内の「dest」に送信する。「MPI_Waitall」を呼ぶまで、送信バッファの内容を更新してはならない。

| | | | |
|---|---|---|---|
| – sendbuf | 任意 | I | 送信バッファの先頭アドレス |
| – count | 整数 | I | メッセージのサイズ |
| – datatype | 整数 | I | メッセージのデータタイプ |
| – dest | 整数 | I | 宛先プロセスのアドレス(ランク) |
| – tag | 整数 | I | メッセージタグ。同じタグ番号同士で通信。 |
| – comm | 整数* | O | コミュニケータ |
| – request | 整数* | O | 通信識別子、MPI_WAITALLで使用する。<br>(配列:サイズは同期する必要のある「MPI_ISEND」呼び出し数(隣接プロセス数など)) |

*Cの場合、commはMPI_Comm、requestはMPI_Request型

# MPI_IRECV / MPI_Irecv

call MPI_ISEND
(recvbuf, count, datatype, dest, tag, comm, request, ierr)

F

MPI_Isend
(recvbuf, count, datatype, dest, tag, comm, request)

C

・受信バッファ「recvbuf」内の、連続した「count」個の送信メッセージをタグ「tag」 を付けて、コミュニケータ内の「dest」から受信する。「MPI_Waitall」を呼ぶまで、受信バッファの内容を利用した処理を実施してはならない。

| | | | |
|---|---|---|---|
| – recvbuf | 任意 | I | 受信バッファの先頭アドレス |
| – count | 整数 | I | メッセージのサイズ |
| – datatype | 整数 | I | メッセージのデータタイプ |
| – dest | 整数 | I | 宛先プロセスのアドレス(ランク) |
| – tag | 整数 | I | メッセージタグ。同じタグ番号同士で通信。 |
| – comm | 整数* | I | コミュニケータ |
| – request | 整数* | O | 通信識別子、MPI_WAITALLで使用する<br>(配列:サイズは同期する必要のある「MPI_RECV」呼び出し数(隣接プロセス数など)) |

*Cの場合、commはMPI_Comm、requestはMPI_Request型

# MPI_WAITALL / MPI_WAITALL

call MPI_WAITALL
(count, request, status, ierr)

F

MPI_Waitall
(count, request, status)

C

・『MPI_ISEND』と『MPI_IRECV』を使用した場合に、プロセスの同期をとる。
・「MPI_WAITALL」を呼ぶ前に送信バッファの内容を変更してはならない。
・「MPI_WAITALL」を呼ぶ前に受信バッファの内容を利用してはならない。
・整合性がとれていれば ISEND/IRECV を同時に同期してもよい。

| | | | |
|---|---|---|---|
| – count | 整数 | I | 同期する必要のある「MPI_ISEND」「MPI_RECV」呼び出し数 |
| – request | 整数 | I/O | 通信識別子。「MPI_ISEND」「MPI_IRECV」で利用した識別子名に対応。 |
| – status | 整数 | O | 状況オブジェクト配列<br>サイズ（MPI_STATUS_SIZE, count） |
| – ierr | 整数 | O | エラーコード |

# MPI実装例

sample.f90

```
include 'mpif.h'
integer :: ierr,nprocs,myrank
integer :: reqsend(1),reqrecv(1)
integer :: statsend(MPI_STATUS_SIZE,1), statrecv(MPI_STATUS_SIZE,1)
integer :: iup, idown
integer :: i
integer, parameter :: nx = 100
real(8) :: ff(-1:nx+2)

call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myrank,ierr)
```

各種変数宣言

MPI開始、プロセス番号取得など

# MPI実装例

## sample.f90　続き

```
do i=-1,nx+2
  ff(i) = dble(i+myrank*200)
enddo

iup   = myrank + 1
idown = myrank -1

if (myrank == nprocs-1) then
  iup = 0
elseif (myrank == 0) then
  idown = nprocs-1
endif

call mpi_isend(ff(nx-1),2,mpi_double_precision,iup,0,mpi_comm_world,reqsend(1),ierr)
call mpi_irecv(ff(-1),2,mpi_double_precision,idown,0,mpi_comm_world,reqrecv(1),ierr)
call mpi_waitall(1, reqsend, statsend, ierr)
call mpi_waitall(1, reqrecv, statrecv, ierr)

call mpi_finalize(ierr)

end
```

配列にデータ代入

iup=右隣りのプロセス
idown=左隣のプロセス

周期境界条件

ff(nx-1)とff(nx)を、右隣のプロセスの
ff(-1)とff(0)に代入する

# OpenMP + MPI（ハイブリッド並列）

```
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myrank,ierr)
.....
        右隣のプロセス（iup）、左隣のプロセス（idown）の定義など
.....
do it = 1, 1000
  call mpi_isend(ff(nx),1,mpi_double_precision,iup,0,mpi_comm_world,reqsend(1),ierr)
  call mpi_irecv(ff(0),1,mpi_double_precision,idown,0,mpi_comm_world,reqrecv(1),ierr)
  call mpi_isend(ff(1),1,mpi_double_precision,iup,1,mpi_comm_world,reqsend(2),ierr)
  call mpi_irecv(ff(nx+1),1,mpi_double_precision,idown,1,mpi_comm_world,reqrecv(2),ierr)
  !$OMP PARALLEL DO
  do i=2,nx-1
    ff(i) = ..........
  enddo
  !$OMP END PARALLEL DO
  call mpi_waitall(2, reqsend, statsend, ierr)
  call mpi_waitall(2, reqrecv, statrecv, ierr)
  ff(1) = ...
  ff(nx) = ...
enddo

call mpi_finalize(ierr)
```

差分計算で必要な「のり代」のデータ交換

**メインの計算部分**
MPIで局所化したデータを、OpenMPでスレッド並列

# 演習問題3

演習問題２でOpenMP並列化したプログラムを、
MPIを用いて並列化せよ

[Fortran]

| | |
|---|---|
| **2018spring/code/f90/openmp/1d_adv_omp.f90** | 移流方程式 |
| 2018spring/code/f90/openmp/1d_fluid_rk_omp.f90 | 流体１次元 |
| 2018spring/code/f90/openmp/2d_fluid_omp.f90 | 流体２次元 |

[C言語]

| | |
|---|---|
| **2018spring/code/c/openmp/1d_adv_omp.c** | 移流方程式 |
| 2018spring/code/c/openmp/1d_fluid_rk_omp.c | 流体１次元 |
| 2018spring/code/c/openmp/2d_fluid_omp.c | 流体２次元 |

をベースとして使ってもよい

# 補足：πコンピュータ上の操作

## コンパイル・ジョブの投入（Fortran）

[xxxx@pi]$ mpifrtpx **-Kopenmp** 1d_adv_hb.f90
[xxxx@pi]$ pjsub run.sh

run.sh

```
#!/bin/sh
#PJM -L "node=8"                    ノード数
#PJM -L "elapse=00:01:00"           実行時間
#PJM =L "rscgrp=small"              実行キュー
#PJM -j
#PJM -o "output.lst"                標準出力ファイル名
#PJM --mpi "proc=8"                 起動するプロセス数

export OMP_NUM_THREADS=16           スレッド数を指定
mpiexec ./a.out                     実行ファイル
```

# 補足：πコンピュータ上の操作

## コンパイル・ジョブの投入（C言語）

[xxxx@pi]$ mpifccpx **-Kopenmp** 1d_adv_hb.c
[xxxx@pi]$ pjsub run.sh

run.sh

```
#!/bin/sh
#PJM -L "node=8"                    ノード数
#PJM -L "elapse=00:01:00"           実行時間
#PJM =L "rscgrp=small"              実行キュー
#PJM -j
#PJM -o "output.lst"                標準出力ファイル名
#PJM --mpi "proc=8"                 起動するプロセス数

export OMP_NUM_THREADS=16           スレッド数を指定
mpiexec ./a.out                     実行ファイル
```

# 演習問題3

## 解答例

[Fortran]

| | |
|---|---|
| 2018spring/code/f90/hybrid/1d_adv_hb.f90 | 移流方程式 |
| 2018spring/code/f90/hybrid/1d_fluid_rk_hb.f90 | 流体１次元 |
| 2018spring/code/f90/hybrid/2d_fluid_hb.f90 | 流体２次元 |

[C言語]

| | |
|---|---|
| 2018spring/code/c/hybrid/1d_adv_hb.c | 移流方程式 |
| 2018spring/code/c/hybrid/1d_fluid_rk_hb.c | 流体１次元 |
| 2018spring/code/c/hybrid/2d_fluid_hb.c | 流体２次元 |

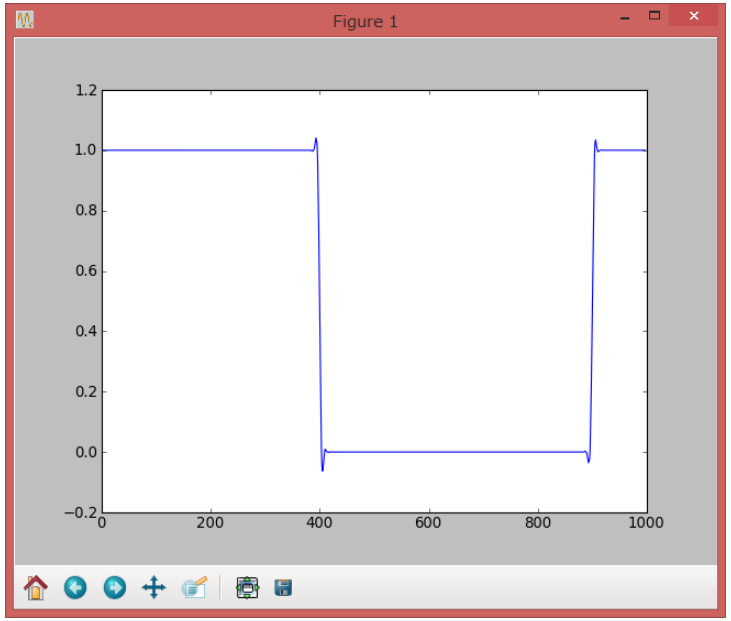
# 補足：πコンピュータ上の操作

## データ出力の確認（Pythonプログラム）

MPI版

```
[xxxx@pi]$ cd ~/2018spring/code/f90/hybrid/dat_1d_adv
[xxxx@pi dat_1d_fluid]$ plot1dmpi.py 8 1000
```

プロセス数　ステップ番号




プロット結果

出力ファイルとしては、
000000-rank0000.dat, 000000-rank0001.dat, ....
001000-rank0000.dat, 001000-rank0001.dat, ....

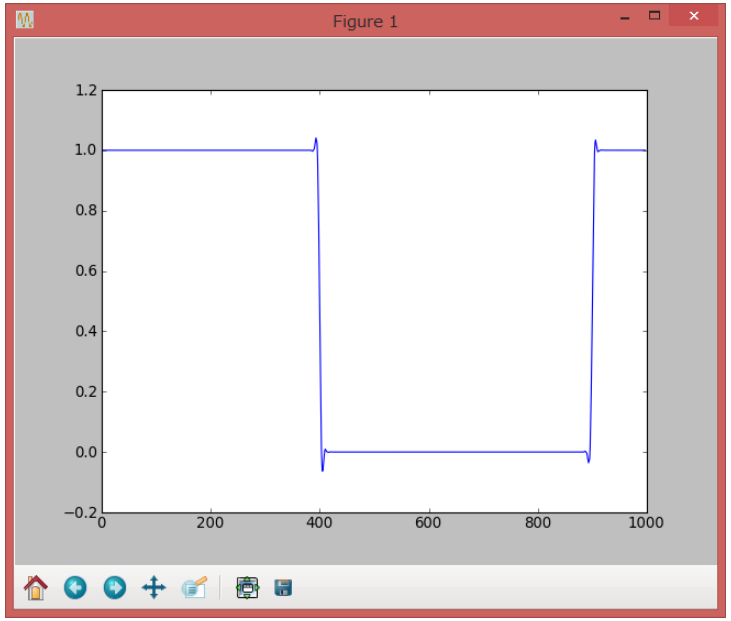
# 補足：πコンピュータ上の操作

## データ出力の確認（Pythonプログラム）

MPI版

```
[xxxx@pi]$ cd ~/2018spring/code/f90/hybrid/dat_1d_fluid
[xxxx@pi dat_1d_fluid]$ plot1dmpi.py 8 1000 pre
```

プロセス数　ステップ番号　パラメータ名




プロット結果

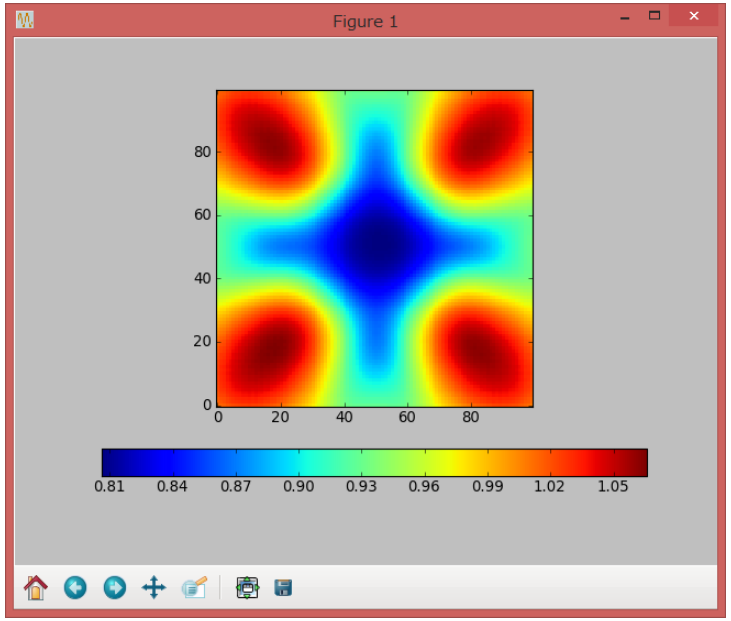
出力ファイルとしては、
pre_000000-rank0000.dat, pre_000000-rank0001.dat,
pre_001000-rank0000.dat, pre_001000-rank0001.dat,
…

# 補足：πコンピュータ上の操作

## データ出力の確認（Pythonプログラム）　2次元MPI版

```
[xxxx@pi]$ cd ~/2018spring/code/f90/hybrid/dat_2d_fluid
[xxxx@pi dat_1d_fluid]$ plot1dmpi.py 2 4 1000 pre
```

プロセス数 (x方向)　プロセス数 (y方向)　ステップ番号　パラメータ




出力ファイルとしては、
pre_000000-rank0000.dat, pre_000000-rank0001.dat,
pre_001000-rank0000.dat, pre_001000-rank0001.dat,
…

プロット結果

# 演習問題3-2

ハイブリッド並列化したプログラムを、プロセス数、スレッド数などを変えて実行し処理時間を計測せよ

各種パラメータはジョブスクリプト run.sh で指定する

#PJM -L ″node=4”

#PJM -mpi “proc=4″

OMP_NUM_THREADS=16

＊ただし実行キュー「small」の最大ノード数は１２

# 補足：時間計測（MPI）

## MPIの関数を用いる：MPI_WTIME

```
real(8) :: stime, etime

stime = MPI_WTIME()

・・・処理・・・

etime = MPI_WTIME()

print*, etime-stime
```

## 実行時にコマンドを使う：time

```
time mpiexec a.out
```