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 計算科学研究センター）

KOBE HPC Spring School 2019
2019年3月14日

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

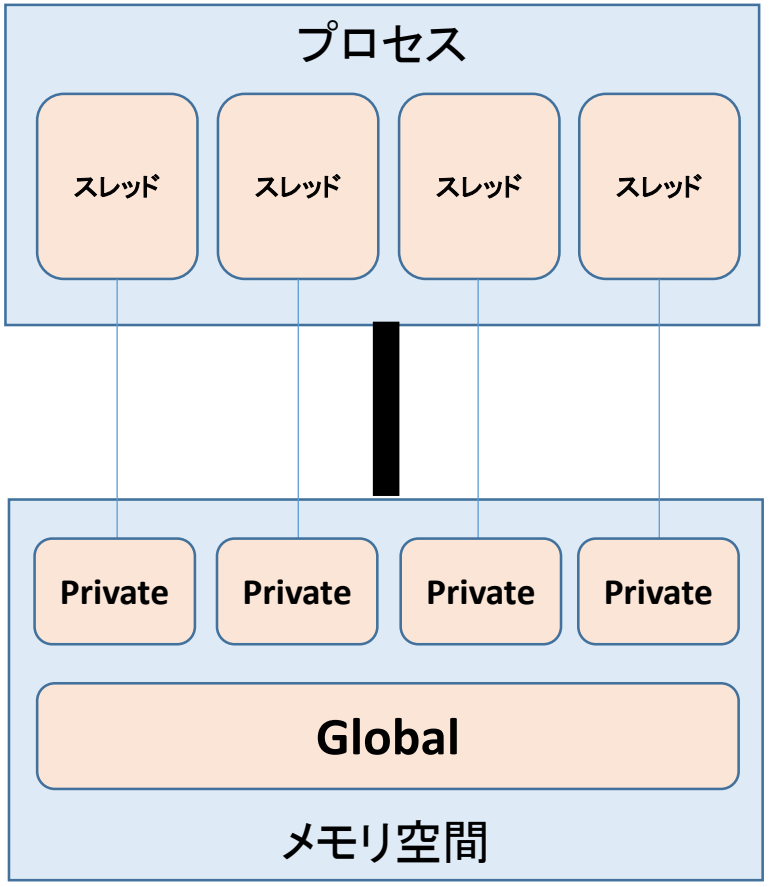
- **以前の HPC Summer School の資料も適宜参照**

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

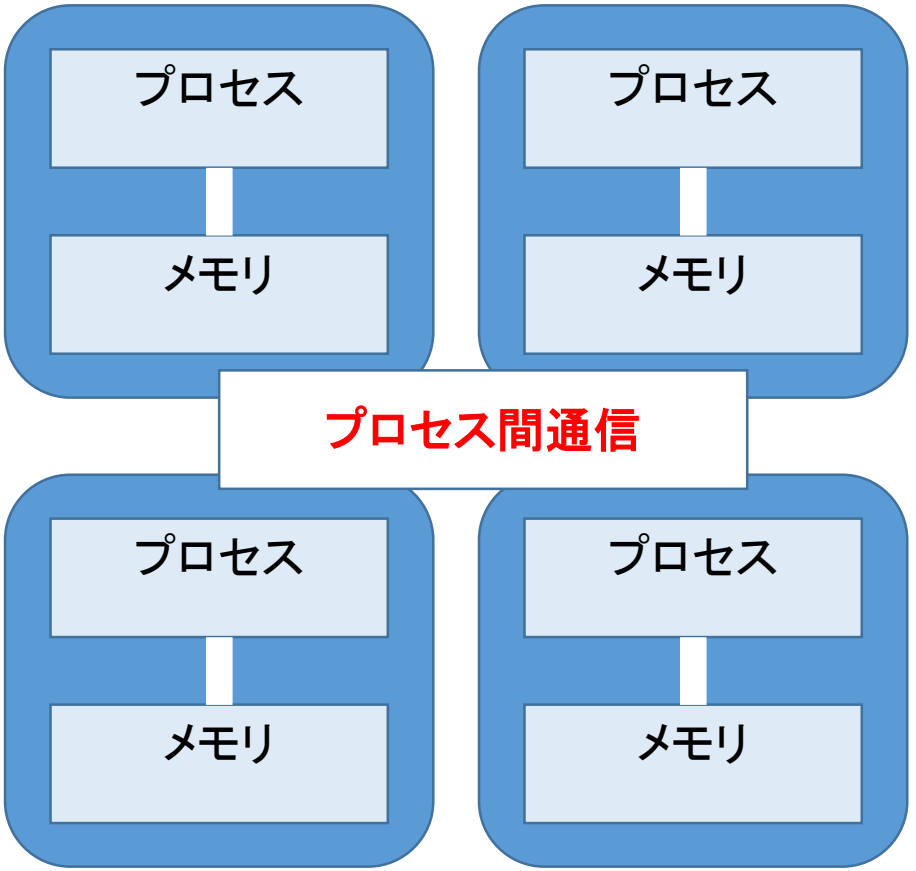
＊本日の演習では、本当に最小限の命令しか使いません。

スレッド並列とプロセス並列

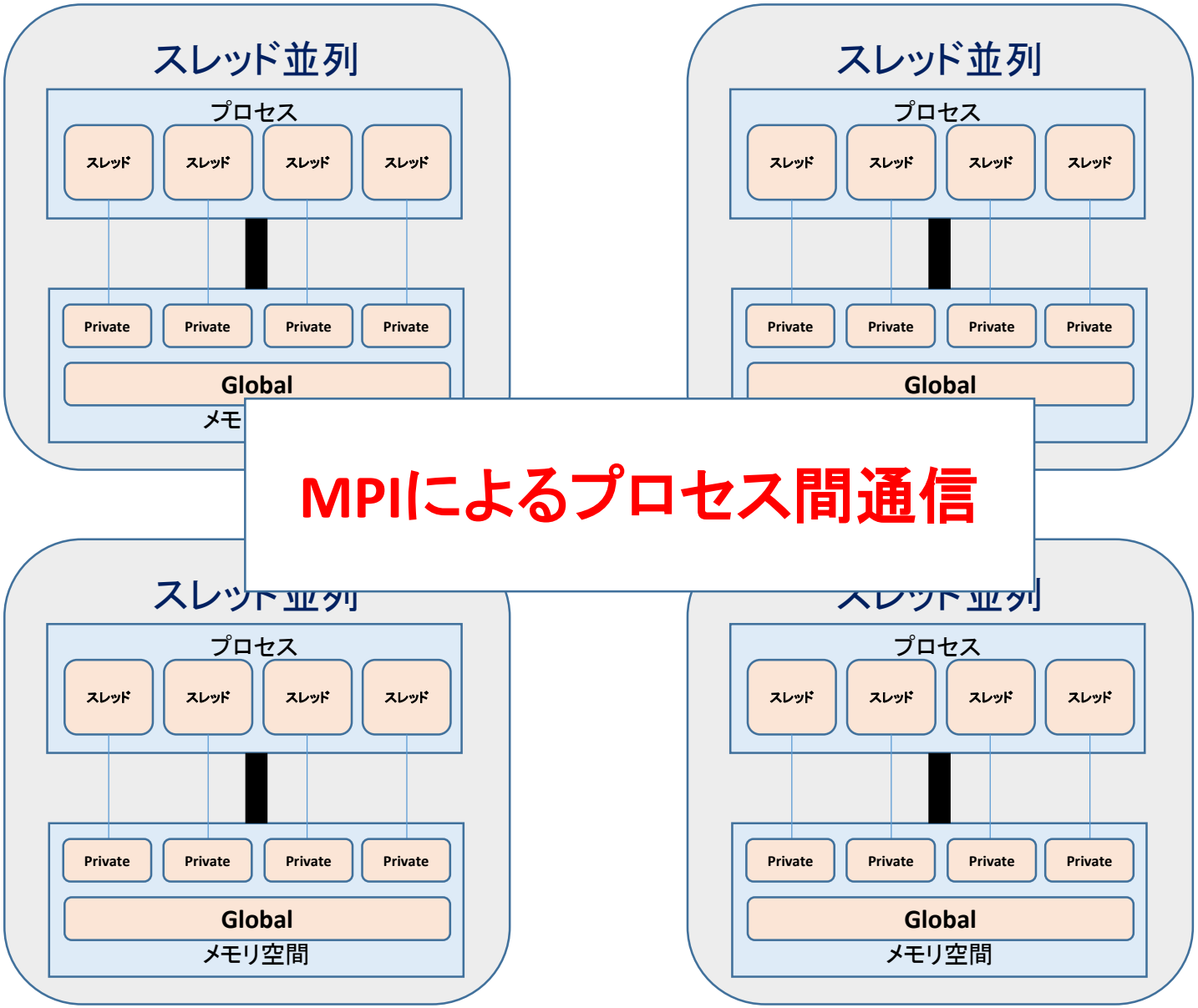
スレッド並列
OpenMP、自動並列化



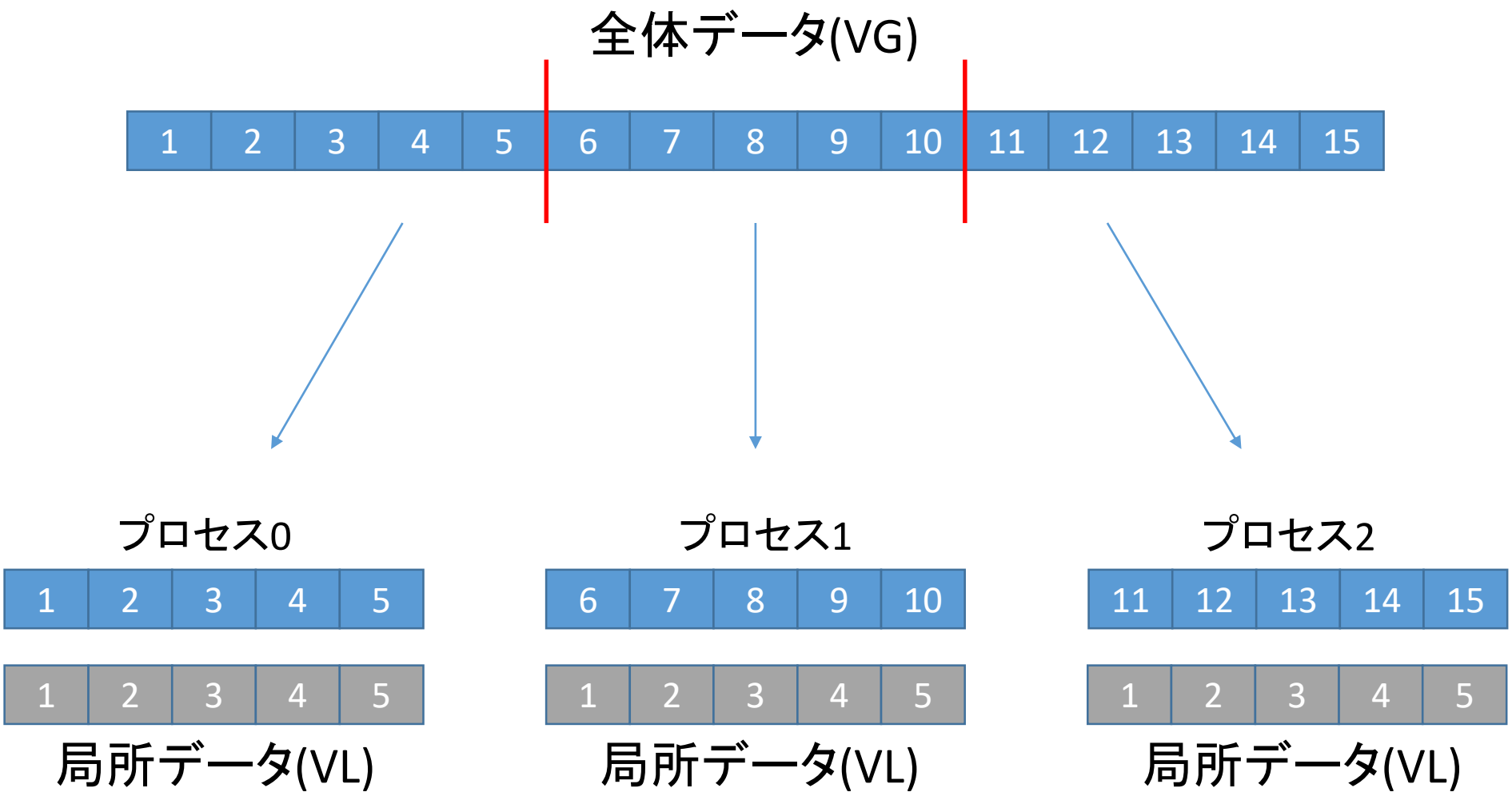
プロセス並列
MPI



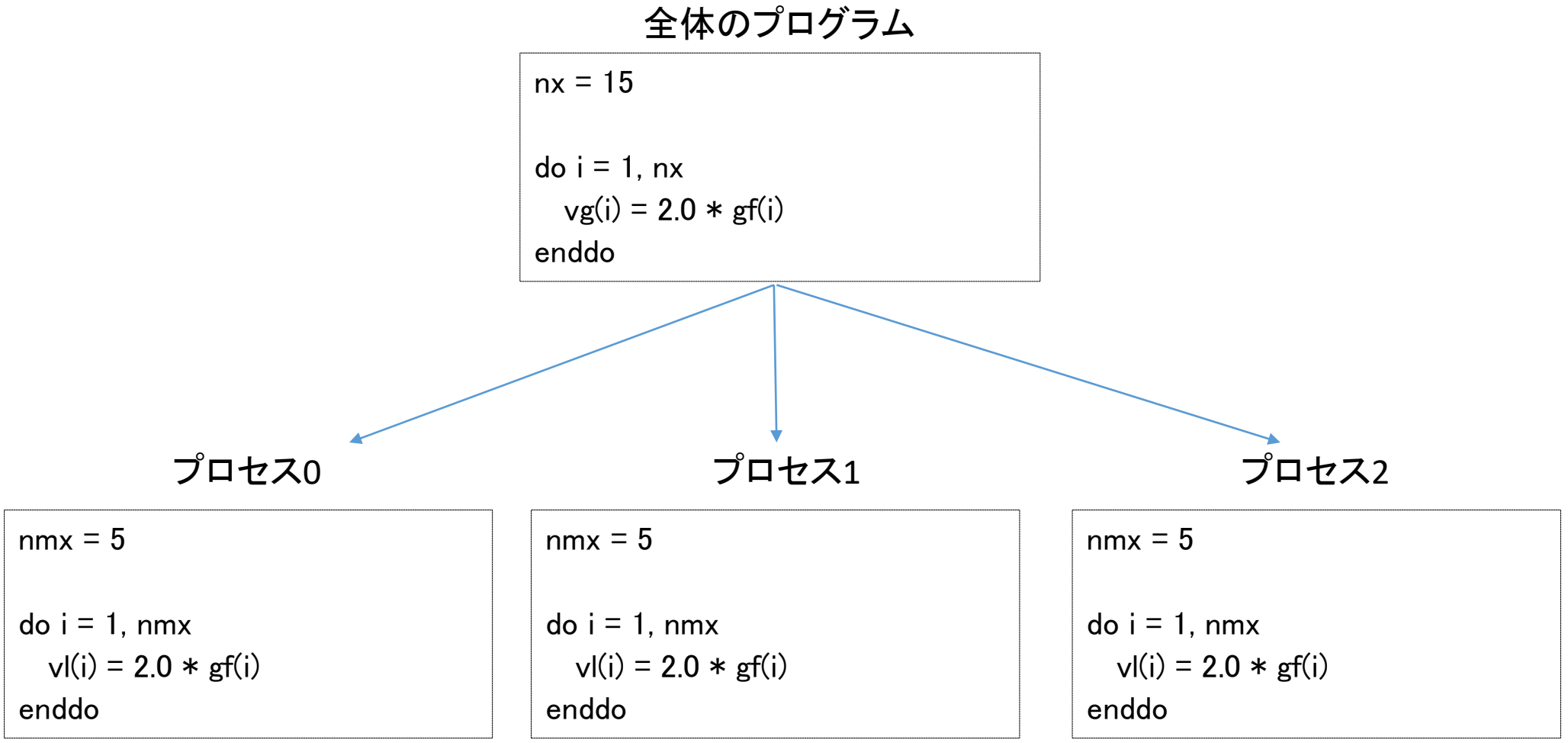
ハイブリッド並列


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



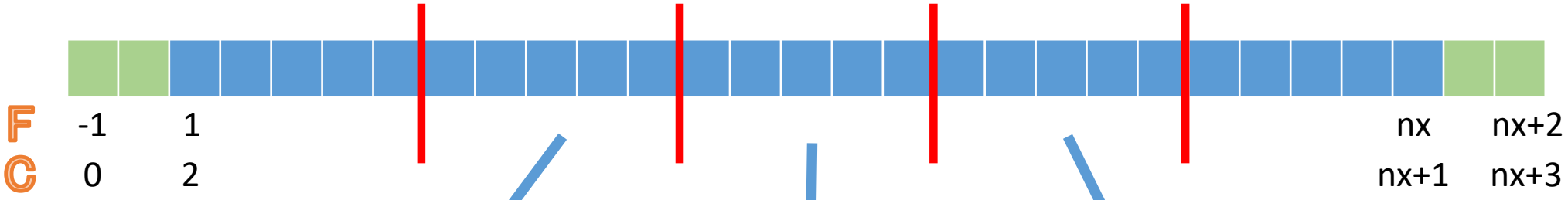
各プロセスで実行するコードは同じだが、データが異なる

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]

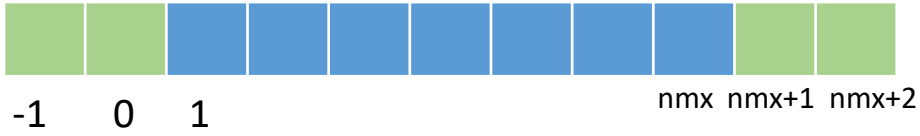
非並列計算の場合、計算領域は 1 ~ nx
差分計算に左右2グリッドずつ必要なため、-1 ~ nx+2 の間で領域を確保



プロセス番号 : P-1

プロセス番号 : P

プロセス番号 : P+1



各プロセスの配列

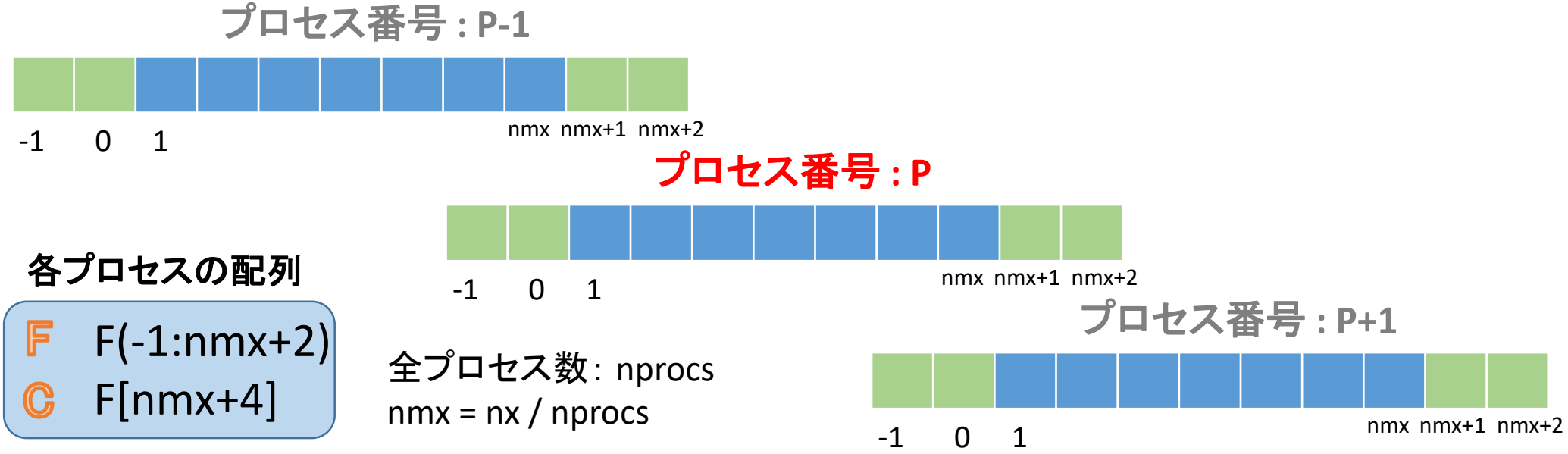
F F(-1:nmx+2)
C F[nmx+4]

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）

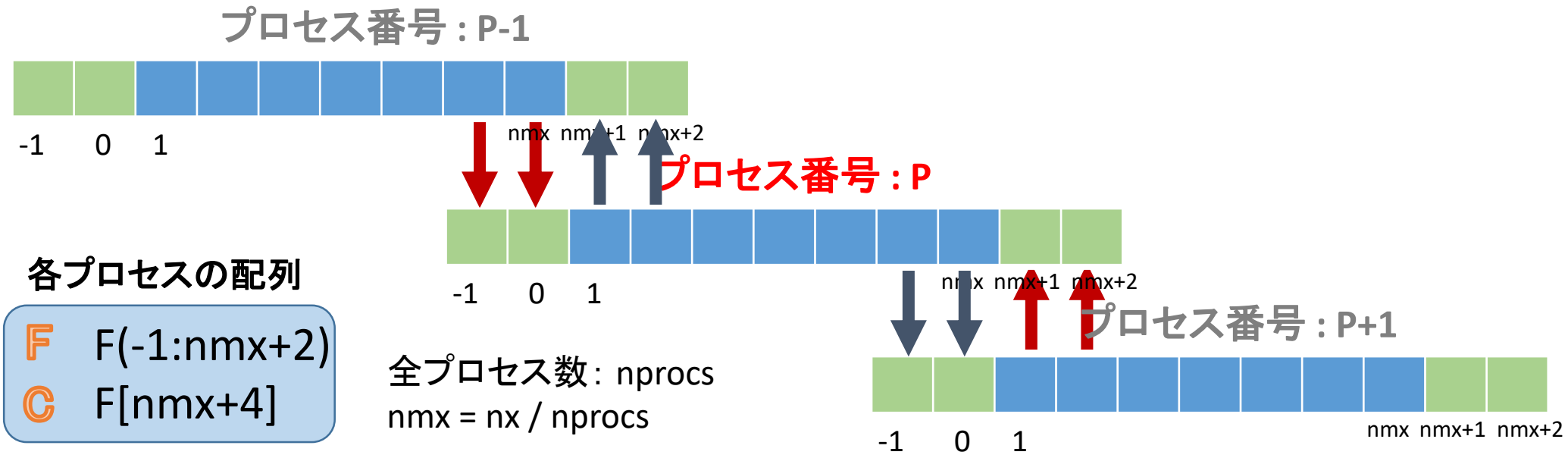
・プロセス P の F(-1), F(0), F(nmx+1), F(nmx+2) は、プログラムにおいて計算はしないが差分計算の上で値は必要。



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

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)

MPIの基本的な機能

F

```
include 'mpif.h'
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)

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

call MPI_FINALIZE(ierr)

stop
end
```

C

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

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  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 :: 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)

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

call MPI_FINALIZE(ierr)

stop
end
```

F

run.sh

```
#!/bin/bash
#PBS -q S
#PBS -l select=1:ncpus=4:mpiprocs=4
#PBS -N HybridJob
#PBS -o output
#PBS -j oe
source /etc/profile.d/modules.sh
module load intel
module load intelmpi

cd ${PBS_O_WORKDIR}
mpirun dplace ./a.out
```

CPU, プロセス数

- この例では4つのプロセスが立ち上がる（”mpiprocs=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 :: 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)

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

call MPI_FINALIZE(ierr)

stop
end
```

C

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

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  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 :: 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)

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

call MPI_FINALIZE(ierr)

stop
end
```

F

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

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  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 :: 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)

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

call MPI_FINALIZE(ierr)

stop
end
```

F

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

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

C

・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

F

```
include 'mpif.h'
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)

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

call MPI_FINALIZE(ierr)

stop
end
```

C

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

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  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 :: 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)

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, nprocs, i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  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_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_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_Wait(all) : ノンブロッキング通信の確認

MPI_Sendrecv : MPI_Send+MPI_Recv

ブロッキング通信:
通信が終了するまで次の処理を行わない（プログラムが止まる）
→場合によってはデッドロックが起こる

ノンブロッキング通信:
通信中も他の処理を行う（プログラムが次へ行く）

MPI_SEND / MPI_Send

call MPI_SEND
(sendbuf, count, datatype, dest, tag, comm, ierr)

F

MPI_Send
(sendbuf, count, datatype, dest, tag, comm)

C

・ブロッキング通信

・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。

– 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	※	I	コミュニケータ

※ Fortranの場合は整数型、Cの場合はMPI_Comm型

MPI_RECV / MPI_Recv

call MPI_RECV
(recvbuf, count, datatype, dest, tag, comm, status, ierr) F

MPI_Recv
(recvbuf, count, datatype, dest, tag, comm, status) C

・ブロッキング通信
・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。

– 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	※	I	コミュニケータ
– status	※	O	状況オブジェクト配列 サイズ(MPI_STATUS_SIZE, count)

※ Fortranの場合は整数型、Cの場合、commはMPI_Comm、statusはMPI_Status型

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

・ノンブロッキング通信
・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_Wait(all)」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	※	I	コミュニケータ
– request	※	O	通信識別子、MPI_WAITALLで使用する。 (配列:サイズは同期する必要のある「MPI_ISEND」呼び出し数)

※ Fortranの場合は整数型、Cの場合、commはMPI_Comm、requestはMPI_Request型

MPI_IRECV / MPI_Irecv

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

F

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

C

・ノンブロッキング通信
・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_Wait(all)」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	※	I	コミュニケータ
– request	※	O	通信識別子、MPI_WAITALLで使用する (配列:サイズは同期する必要のある「MPI_RECV」呼び出し数)

※ Fortranの場合は整数型、Cの場合、commはMPI_Comm、requestはMPI_Request型

MPI_WAIT / MPI_Wait

call MPI_WAIT
(request, status, ierr)

F

MPI_Waitall
(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_WAIT」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_WAIT」を呼ぶ前に受信バッファの内容を利用してはならない。

– request	※	I/O	通信識別子。「MPI_ISEND」「MPI_IRECV」で利用した識別子名に対応。
– status	※	O	状況オブジェクト配列 サイズ（MPI_STATUS_SIZE）
– ierr	整数	O	エラーコード

※ Fortranの場合は整数型、Cの場合、requestはMPI_Request型、statusはMPI_Status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	※	I/O	通信識別子。「MPI_ISEND」「MPI_IRECV」で利用した識別子名に対応。
– status	※	O	状況オブジェクト配列 サイズ（MPI_STATUS_SIZE, count）
– ierr	整数	O	エラーコード

※ Fortranの場合は整数型、Cの場合、requestはMPI_Request型、statusはMPI_Status型

MPI_SENDRECV / MPI_Sendrecv

```
call MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag,
 recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierr)
```

F

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
 recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierr)
```

C

『MPI_Send』+『MPI_Recv』

– sendbuf	任意	I	送信バッファの先頭アドレス
– sendcount	整数	I	送信メッセージのサイズ
– sendtype	整数	I	送信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)
– sendtag	整数	I	送信用メッセージタグ。同じタグ番号同士で通信。
– recvbuf	任意	I	受信バッファの先頭アドレス
– recvcount	整数	I	受信メッセージのサイズ
– recvtype	整数	I	受信メッセージのデータタイプ
– dest	整数	I	受信プロセスのアドレス(ランク)
– sendtag	整数	I	受信用メッセージタグ。同じタグ番号同士で通信。
– comm	※	O	コミュニケータ
– status	※	O	状況オブジェクト配列 サイズ(MPI_STATUS_SIZE)
– ierr	整数	O	完了コード

request, statusの型について

通信識別子（request）、状況オブジェクト配列(status)は、C言語とFortranで変数の型などが異なる。

C

```
#include "mpi.h"
MPI_Status status[4];
MPI_Request request[4];

MPI_Isend(....., &request[0])
MPI_Irecv(....., &request[1])
.....
MPI_Irecv(....., &request[N-1])

MPI_Waitall(N, request, status)
```

F

```
use mpi
integer :: status(MPI_STATUS_SIZE, N)
integer :: request(N)

call mpi_isend(....., request(1), ierr)
call mpi_irecv(....., request(2), ierr)
.....
call mpi_irecv(....., request(N), ierr)

call mpi_waitall(N, request, status, ierr)
```

サンプルプログラム（ex3）の場合、*N*は4

データ型（datatype）に関して

・MPIの基本データ型はC言語とFortranで記述が異なる

[Fortran]
MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER, など

[C言語]
MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, など

・本日の演習に関しては以上を覚えておけばよい

・詳細は明日の『コミュニケータとデータタイプ』の講義にて

MPI_SCATTER / MPI_Scatter

call MPI_SCATTER
(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)

F

MPI_Scatter
(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

C

・コミュニケータ内の一つの送信元プロセス「root」の送信バッファ「sendbuf」から各プロセスに先頭から「sendcount」ずつのサイズのメッセージを送信し、その他全てのプロセスの受信バッファ「recvbuf」にサイズ「recvcount」のメッセージを格納。

– sendbuf	任意	I	送信バッファの先頭アドレス
– sendcount	整数	I	送信メッセージのサイズ
– sendtype	整数	I	送信メッセージのデータタイプ
– recvbuf	任意	I	受信バッファの先頭アドレス
– recvcount	整数	I	受信メッセージのサイズ
– recvtype	整数	I	受信メッセージのデータタイプ
– root	整数	I	受信プロセスのアドレス（ランク）
– comm	※	O	コミュニケータ

通常は sendcount = recvcount かつ sendtype=recvtype

MPI_GATHER / MPI_Gather

call MPI_GATHER
(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)

F

MPI_Gather
(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

C

・MPI_Scatterの逆

– sendbuf	任意	I	送信バッファの先頭アドレス
– sendcount	整数	I	送信メッセージのサイズ
– sendtype	整数	I	送信メッセージのデータタイプ
– recvbuf	任意	I	受信バッファの先頭アドレス
– recvcount	整数	I	受信メッセージのサイズ
– recvtype	整数	I	受信メッセージのデータタイプ
– root	整数	I	受信プロセスのアドレス（ランク）
– comm	※	O	コミュニケータ

受信バッファrecvbufの値はroot番のプロセスに集められる

MPI_BCAST / MPI_Bcast

call MPI_BCAST
(buffer, count, datatype, root, comm)

F

MPI_Bcast
(buffer, count, datatype, root, comm)

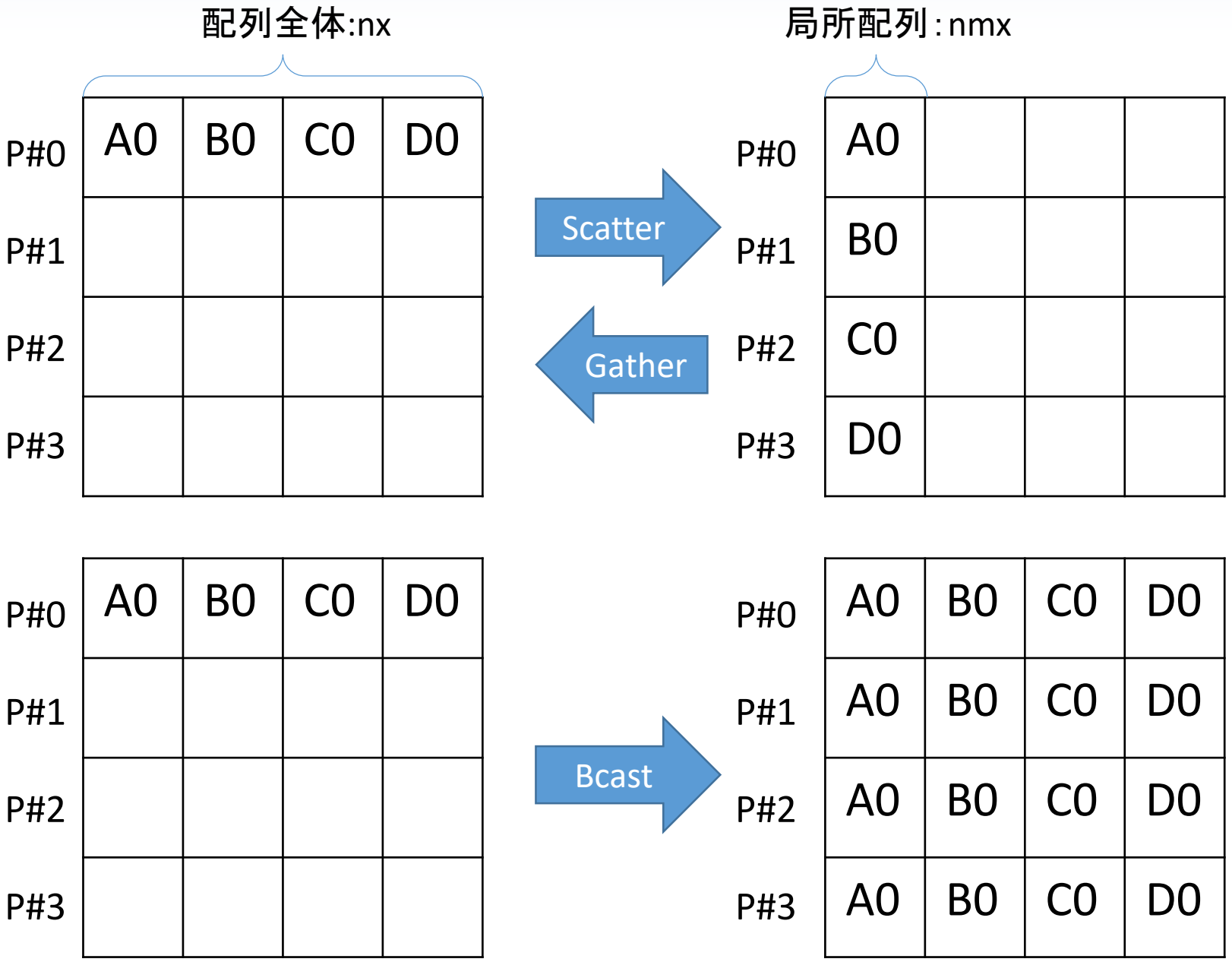
C

・コミュニケータ「comm」内の１つの送信元プロセス「root」のバッファ「buffer」から、その他全てのプロセスのバッファ「buffer」にメッセージを送信

– buffer	任意	I	バッファの先頭アドレス
– count	整数	I	メッセージのサイズ
– data	整数	I	メッセージのデータタイプ
– root	整数	I	送信プロセスのアドレス（ランク）
– comm	※	O	コミュニケータ

受信バッファrecvbufの値はroot番のプロセスに集められる

Gather/Scatterのイメージ



演習問題3−1

MPIサンプルコード ex1, ex2, ex3 を実行せよ。
また、ex3をプロセス数を変えて実行し、動作を確認せよ。

[Fortran]
2019spring/code/f90/hybrid/ex1.f90
2019spring/code/f90/hybrid/ex2.f90
2019spring/code/f90/hybrid/ex3.f90

[C言語]
2019spring/code/c/hybrid/ex1.c
2019spring/code/c/hybrid/ex2.c
2019spring/code/c/hybrid/ex3.c

ex1: 全プロセス数と自分のプロセス番号を書き出すプログラム
ex2: ランク0のデータをランク1に送信するプログラム
ex3: 配列の境界部分を、両端のプロセスと相互通信するプログラム

補足：マシン上の操作

コンパイル（MPI：INTEL MPIを使用）

> module load intel
> module load intelmpi
> **mpiicc** -qopenmp 1d_adv_hb.c　　C言語
> **mpiifort** -qopenmp 1d_adv_hb.f90　　Fortran
> qsub run.sh

コンパイル（MPI：SGI MPTを使用）

> module load intel
> module load mpt
> **icc** -qopenmp 1d_adv_hb.c -lmpi　　C言語
> **ifort** -qopenmp 1d_adv_hb.f90 -lmpi　　Fortran
> qsub run.sh

補足：マシン上の操作

ジョブスクリプト（MPI：INTEL MPIを使用）

```
#!/bin/bash
#PBS -q S
#PBS -l select=4:ncpus=16:mpiprocs=2
#PBS -N HybridJob
#PBS -o output
#PBS -j oe
source /etc/profile.d/modules.sh
module load intel
module load intelmpi
export   OMP_NUM_THREADS=8
export   KMP_AFFINITY=disabled

cd ${PBS_O_WORKDIR}
mpirun dplace ./a.out
```

ノード数、CPU数、プロセス数
ジョブ名
標準出力の出力先ファイル

コンパイラ環境のロード
MPI環境をロード

補足：マシン上の操作

ジョブスクリプト（MPI：SGI MPTを使用）

```
#!/bin/bash
#PBS -q S
#PBS -l select=4:ncpus=16:mpiprocs=2          ノード数、CPU数、プロセス数
#PBS -N HybridJob                             ジョブ名
#PBS -o output                                標準出力の出力先ファイル
#PBS -j oe
source /etc/profile.d/modules.sh
module load intel                             コンパイラ環境のロード
module load mpt                               MPI環境のロード
export   OMP_NUM_THREADS=8
export   KMP_AFFINITY=disabled

cd ${PBS_O_WORKDIR}
mpiexec_mpt omplace -nt ${OMP_NUM_THREADS} ./a.out
```

MPI実装例

[C言語] ex3.c

```
#include <stdio.h>
#include "mpi.h"
#define nmx 5

int main(int argc, char **argv){
  int nprocs,myrank;
  int iup,idown;
  int i;
  MPI_Status istat[4];
  MPI_Request ireq[4];
  double ff[nmx+4];
  char str[255];

  // MPI params
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
```

各種変数宣言

MPI開始、プロセス番号取得など

MPI実装例

[C言語] ex3.c

```
// define rank up/down
iup   = myrank + 1;
idown = myrank - 1;

// periodic
if (myrank == nprocs-1) {
  iup = 0;
} else if (myrank == 0) {
  idown = nprocs-1;
}

// initial condition
for (i=0; i<nmx+4; i++) {
  ff[i] = (double)(myrank*100+i);
}
```

iup=右隣りのプロセス
idown=左隣のプロセス

プロセス番号を
周期境界にする

配列にデータ代入

MPI実装例

[C言語] ex3.c

```
// use MPI function
  MPI_Isend(&ff[nmx]  ,2,MPI_DOUBLE,iup   ,0,MPI_COMM_WORLD,&ireq[0]);
  MPI_Irecv(&ff[0]    ,2,MPI_DOUBLE,idown,0,MPI_COMM_WORLD,&ireq[1]);
  MPI_Isend(&ff[2]    ,2,MPI_DOUBLE,idown,1,MPI_COMM_WORLD,&ireq[2]);
  MPI_Irecv(&ff[nmx+2],2,MPI_DOUBLE,iup   ,1,MPI_COMM_WORLD,&ireq[3]);


  MPI_Waitall(4, ireq, istat);

  MPI_Finalize();
  return 0;
}
```

プロセス iup に ff[nmx], ff[nmx+1] を送信
プロセス idown からのデータを f[0], f[1]に格納

プロセス idown に ff[2], ff[3] を送信
プロセス iup からのデータを f[nmx+2], f[nmx+3]に格納

4つの通信に関してプロセス間の同期をとる

MPIの終了処理

MPI実装例

[Fortran] ex3.f90

```
program main
use mpi
implicit none
integer, parameter ::  nmx = 5
integer :: ierr,nprocs,myrank
integer :: iup, idown
integer :: i
integer :: ireq(4)
integer :: istat(MPI_STATUS_SIZE,4)
real(8) :: ff(-1:nmx+2)

!! MPI params
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

[Fortran] ex3.f90

```
!! define rank up/down
iup   = myrank + 1
idown = myrank - 1

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

!! initial condition
do i=-1,nmx+2
  ff(i) = dble(myrank*100+i)
enddo

write(*,'(a,i2,a,14f12.4)') ' (init) myrank =', myrank, ' ::', ff
```

iup=右隣りのプロセス
idown=左隣のプロセス

プロセス番号を
周期境界にする

配列にデータ代入

MPI実装例

[Fortran] ex3.f90

```
!! use MPI function
call mpi_isend(ff(nmx-1),2,mpi_double_precision,iup,0,mpi_comm_world,ireq(1),ierr)
call mpi_irecv(ff(-1)   ,2,mpi_double_precision,idown,0,mpi_comm_world,ireq(2),ierr)
call mpi_isend(ff(1)    ,2,mpi_double_precision,idown,1,mpi_comm_world,ireq(3),ierr)
call mpi_irecv(ff(nmx+1),2,mpi_double_precision,iup,1,mpi_comm_world,ireq(4),ierr)


call mpi_waitall(4, ireq, istat, ierr)

call mpi_finalize(ierr)

end
```

プロセス iup に ff(nmx-1), ff(nmx) を送信
プロセス idown からのデータを f(-1), f(0)に格納

プロセス idown に ff(1), ff(2) を送信
プロセス iup からのデータを f(nmx+1), f(nmx+2)に格納

4つの通信に関してプロセス間の同期をとる

MPIの終了処理

ハイブリッド並列（MPI+OpenMP）のイメージ

```
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)
  call mpi_waitall(2, reqsend, statsend, ierr)
  call mpi_waitall(2, reqrecv, statrecv, ierr)
  !$OMP PARALLEL DO
  do i = 1, nmx
    ff(i) = ...........
  enddo
  !$OMP END PARALLEL DO
enddo

call mpi_finalize(ierr)
```

差分計算で必要な境界部分のデータ交換

メインの計算部分
MPIで局所化したデータ 1~nmx を、さらにOpenMPでスレッド並列化する

演習問題3−2

MPIを用いて次のハイブリッド並列コードを完成させよ

[Fortran]

2019spring/code/f90/hybrid/1d_adv_hb.f90 **移流方程式**

2019spring/code/f90/hybrid/1d_fluid_rk_hb.f90 流体1次元

[C言語]

2019spring/code/c/hybrid/1d_adv_hb.c **移流方程式**

2019spring/code/c/hybrid/1d_fluid_rk_omp.c 流体1次元

プロセス並列(1次元)の基本方針

移流方程式:1d_adv_hb

方針

1. 全プロセス数、自身のプロセス番号の取得(myrank, nprocs)
2. 左右のプロセスのプロセス番号を定義(iup, idown)
3. 毎ステップ、移流計算の直前に境界部分の通信を行う
(MPI_Isend/MPI_RecvもしくはMPI_Sendrecv)

nx : 全体の計算範囲
nmx : 各プロセスの計算範囲

すなわち、プロセス数をnprocsとすると
nmx = nx/nprocs
となる

プロセス並列(1次元)の基本方針

流体方程式(1次元):1d_fluid_rk

方針

基本的には1次元の移流と方針は同じであるが、流体ということで扱う変数が複数存在している。

書式:

Fortran : ff(X, ieq)

C : ff[ieq][X]

ただし

ieq=1:密度,

ieq=2:圧力,

ieq=3:速度 である

プロセス並列(1次元)の基本方針

流体方程式(1次元):1d_fluid_rk

左のノード(idown)に送りたいのは、左端2グリッド分のデータ

密度:
圧力:
速度:

F	C
ff(-1, 1), ff(0, 1)	ff[0][0], f[0][1]
ff(-1, 2), ff(0, 2)	ff[1][0], f[1][1]
ff(-1, 3), ff(0, 3)	ff[2][0], f[2][1]

である。各パラメータはメモリ上では連続ではないため、このままではsend/recvを3回行わなければならない。

そこで、通信したデータを1まとめにパックした変数を新たに定義する。

プロセス並列(1次元)の基本方針

流体方程式(1次元):1d_fluid_rk

すなわち

2グリッド×3変数

```
real(8) :: fl_send(2,3)
fl_send(1,1) = ff(-1,1)
fl_send(2,1) = ff( 0,1)
fl_send(1,2) = ff(-1,2)
fl_send(2,2) = ...
fl_send(1,3) = ...
...
```

F

2グリッド×3変数

```
double fl_send[3][2]
fl_send[0][0] = ff[0][0]
fl_send[0][1] = ff[0][1]
fl_send[1][0] = ff[1][0]
fl_send[1][1] = ...
fl_send[2][0] = ...
...
```

C

として fl_send を送信すれば1回の通信で済ませることができる。

call mpi_isend(fl_send, 2*3, mpi_double_precision, idown, ...) F

MPI_Isend(fl_send, 2*3, MPI_DOUBLE, iup, ...) C

演習問題3−2

解答例

[Fortran]
2019spring/code/f90/hybrid/sample/1d_adv_hb_sample.f90
2019spring/code/f90/hybrid/sample/1d_fluid_rk_hb_sample.f90

[C言語]
2019spring/code/c/hybrid/sample/1d_adv_hb_sample.c
2019spring/code/c/hybrid/sample/1d_fluid_rk_hb_sample.c

演習問題3-3

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

#PBS -l select=4:ncpus=16:mpiprocs=2

export OMP_NUM_THREADS=8

select：使用ノード数

ncpus：1ノード辺りのCPUコア数

mpiprocs：1ノード辺りのプロセス数

OMP_NUM_THREADS：1プロセス辺りのスレッド数

この例の場合、4ノード使用、各ノード16コアを使用しつつ2プロセスx8スレッドで動作

（計4*16=64コア使用）

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
```

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

-- 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

-- 上下左右の計4変数を用意してパックする必要あり

演習問題3−4

MPIを用いて以下の2次元流体コードのプロセス並列化を完成させよ

[Fortran]
2019spring/code/f90/hybrid/2d_fluid_hb.f90

[C言語]
2019spring/code/c/hybrid/2d_fluid_hb.c

多次元分割に関して

全体の配列

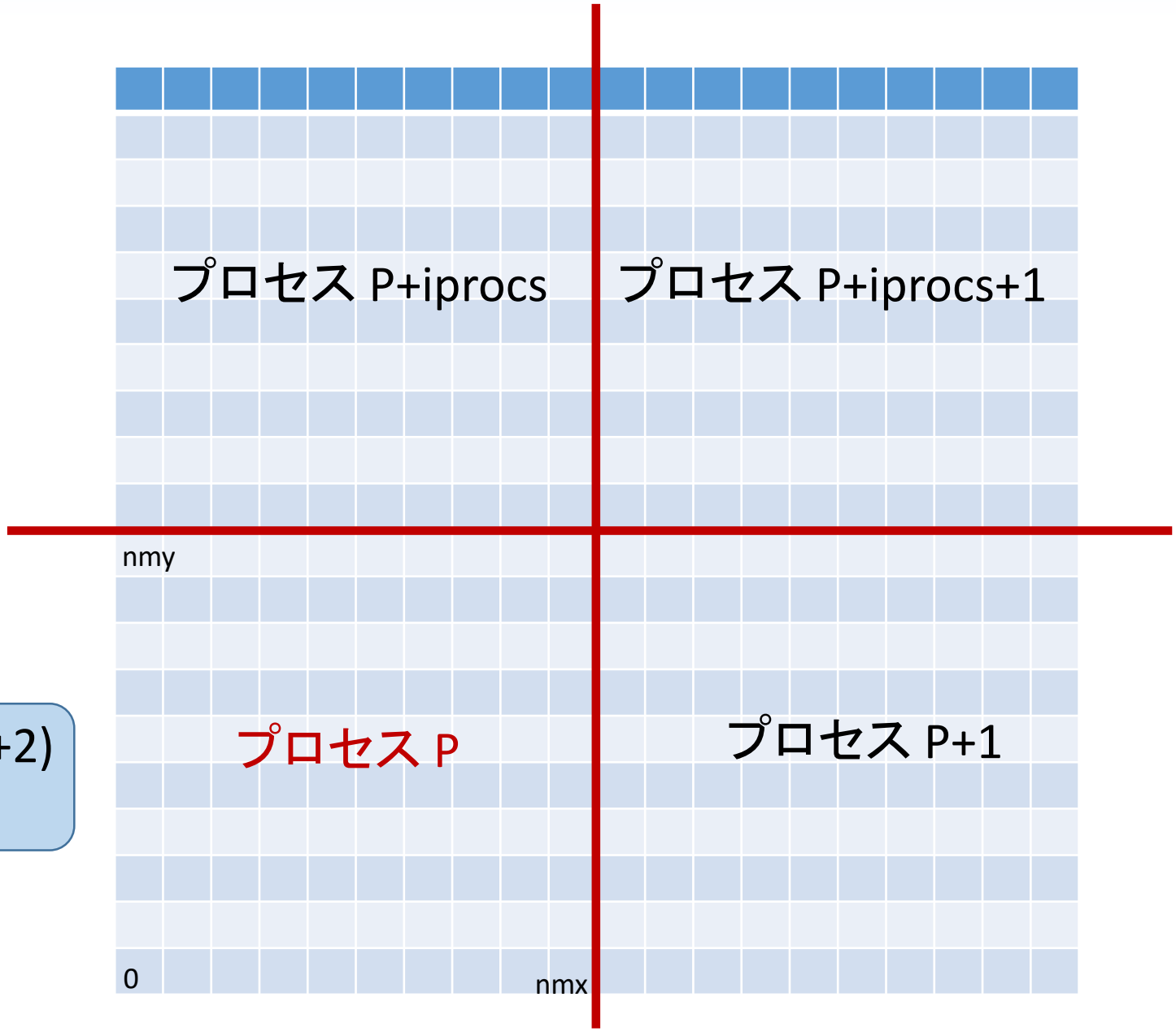
F　F(-1:nx+2, -1:ny+2)
C　F[ny+4][nx+4]

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



多次元分割に関して

全体の配列

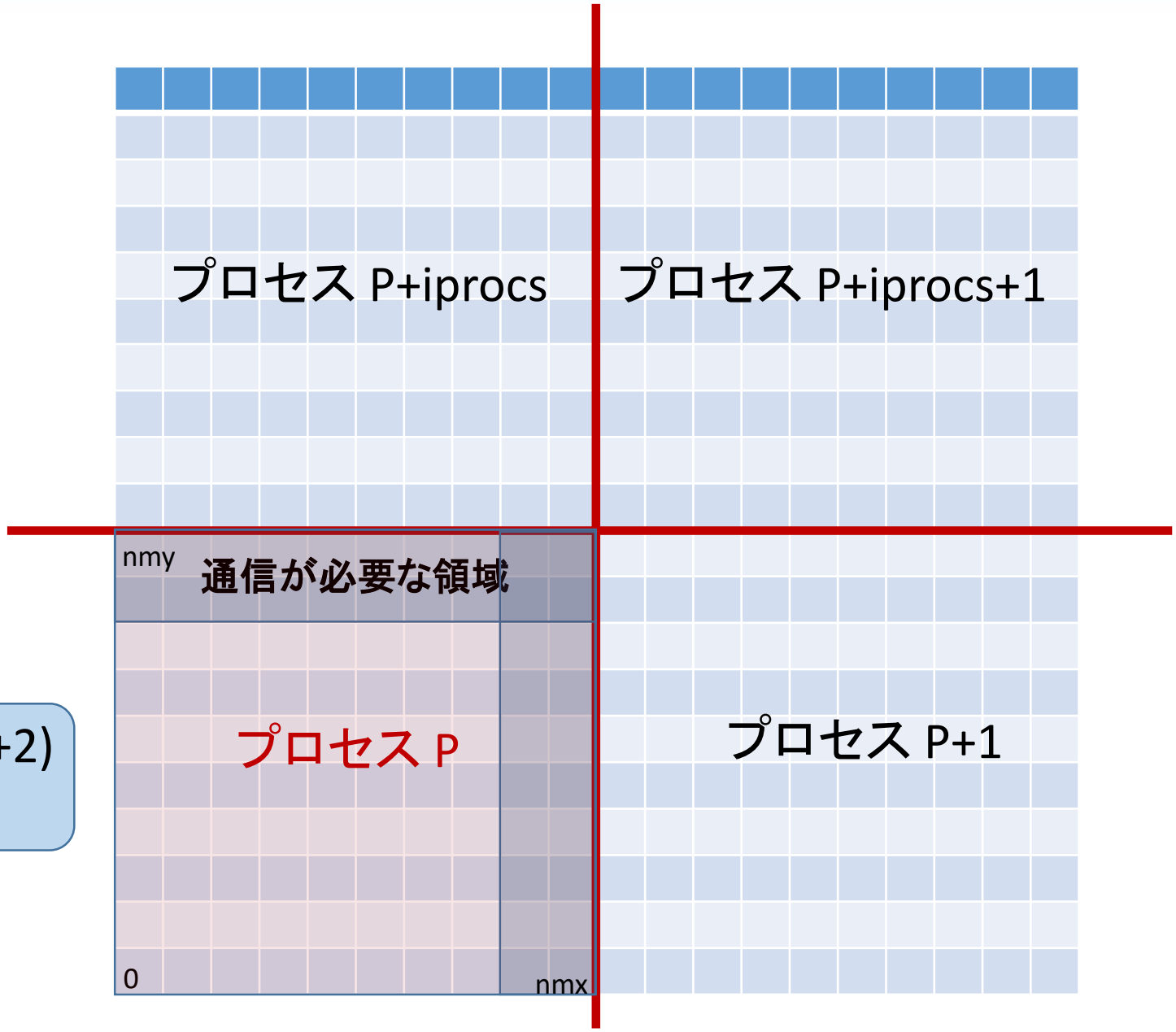
F　F(-1:nx+2, -1:ny+2)
C　F[ny+4][nx+4]

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



多次元分割に関して

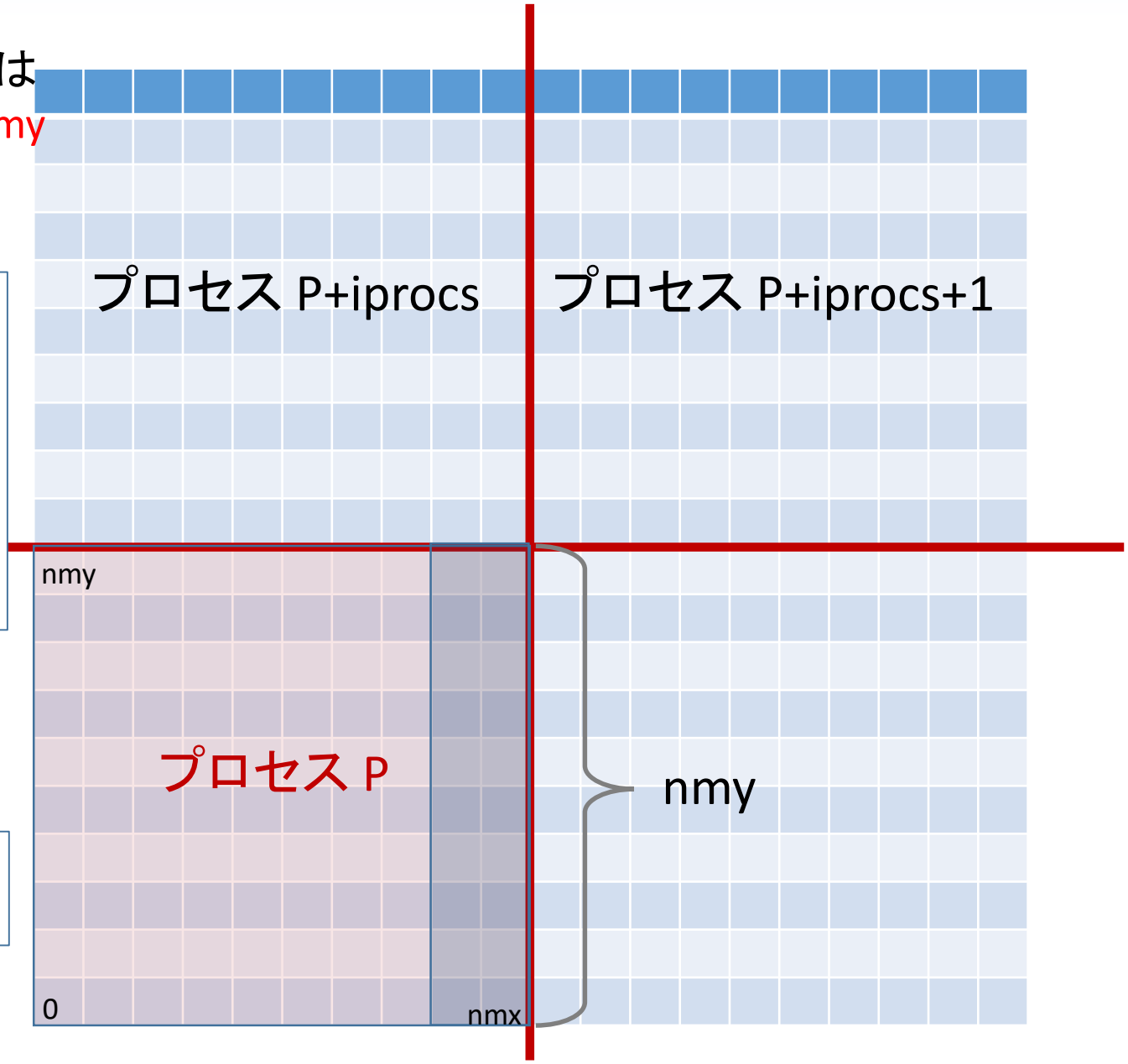
右隣のノードに送りたいデータ数は
nmy * 2[grids] * 4 [params] = 8*nmy

1つの連続な配列に置き換える

```
real(8) :: fr_send(2*nmy,4)
do ieq=1,4
  do j=1,nmy
    fr_send(j,ieq) = f(nmx-1,j,ieq)
    fr_send(nmy+j,ieq) = f(nmx,j,ieq)
  enddo
enddo
```

右隣りのプロセス iup に
fr_sendを送信する

```
call mpi_isend(fr_send, 8*nmy,
mpi_double_precision, iup, ...)
```



多次元分割に関して

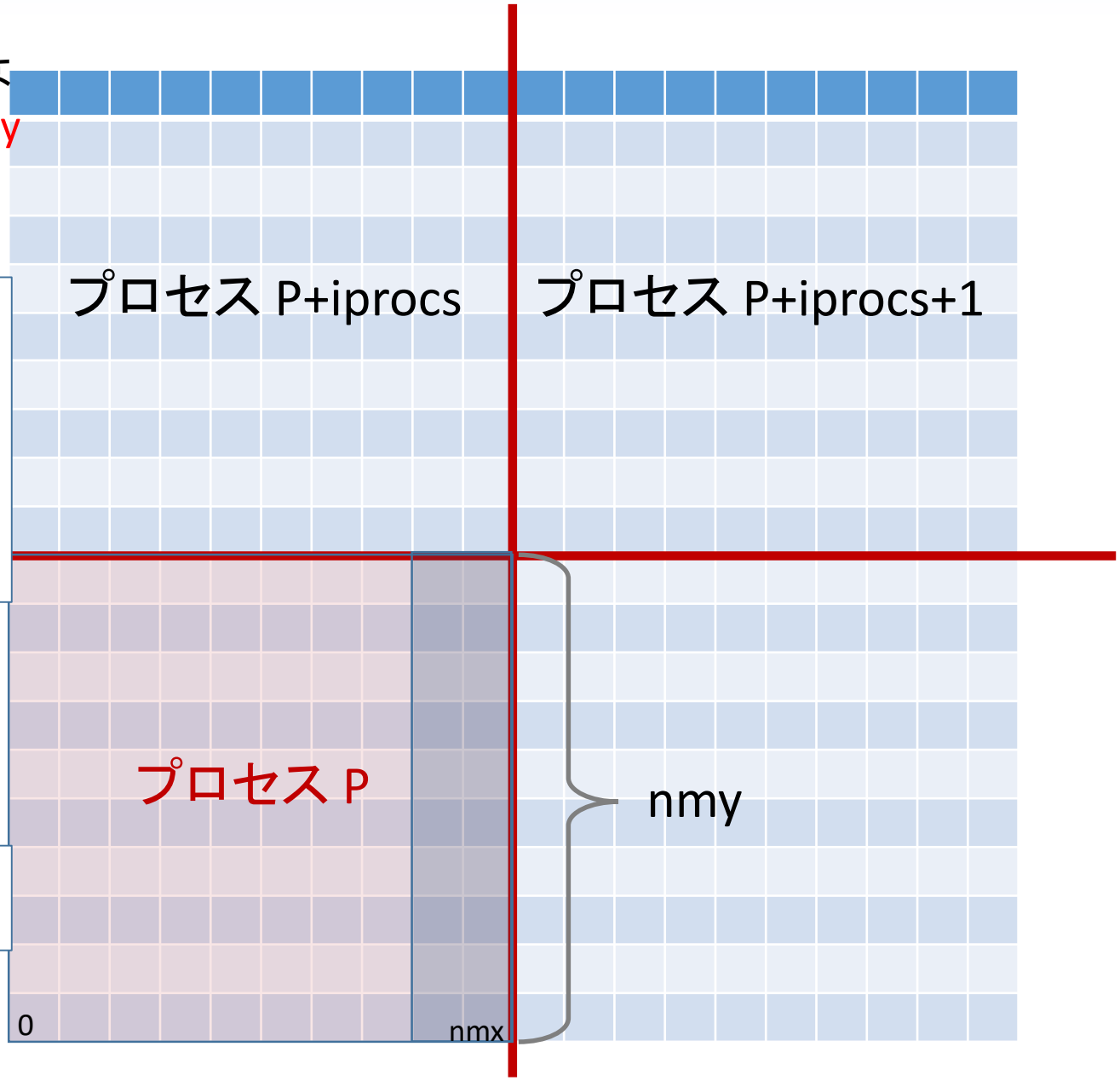
右隣のノードに送りたいデータ数は
nmy * 2[grids] * 4 [params] = 8*nmy

1つの連続な配列に置き換える

```c
double fr_send[4][2*nmy]
for (ieq=0; ieq<4; ieq++){
  for (j=0; j<nmy; j++){
    fr_send[ieq][j] = f[ieq][j+2][nmx];
    fr_send[ieq][nmy+j] = f[ieq][j+2][nmx+1];
  }
}
```

右隣りのプロセス iup に
fr_sendを送信する

```c
MPI_Isend(fr_send, 8*nmy
,MPI_DOUBLE, iup, ...)
```



演習問題3−4

解答例

[Fortran]

2019spring/code/f90/hybrid/sample/2d_fluid_hb_sample.f90

[C言語]

2019spring/code/c/hybrid/sample/2d_fluid_hb_sample.c