

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 2017

2017年3月14日 神戸大学計算科学教育センター

MPIとは

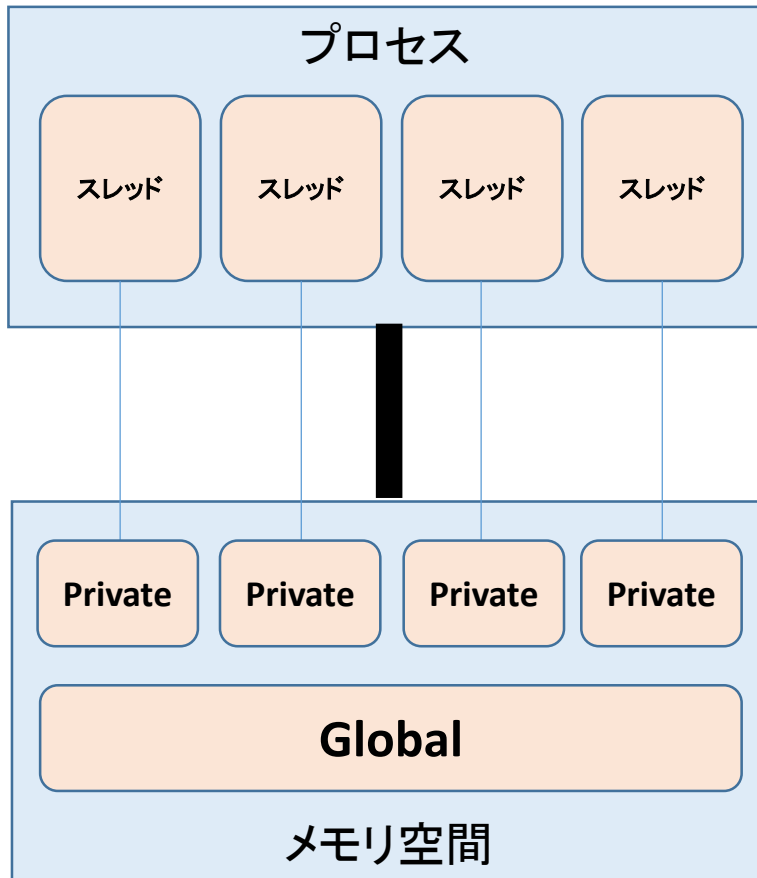
- Message Passing Interface
- 分散メモリのプロセス間の**通信規格** (API)
- SPMD (Single Program Multi Data) が基本
 - 各プロセスが「同じことをやる」が「データが違う」
- 『mpich』や『openmpi』などの実装が有名
 - * OpenMPとは違います
- 詳しくは **HPC Summer School 2016 の資料参照**

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

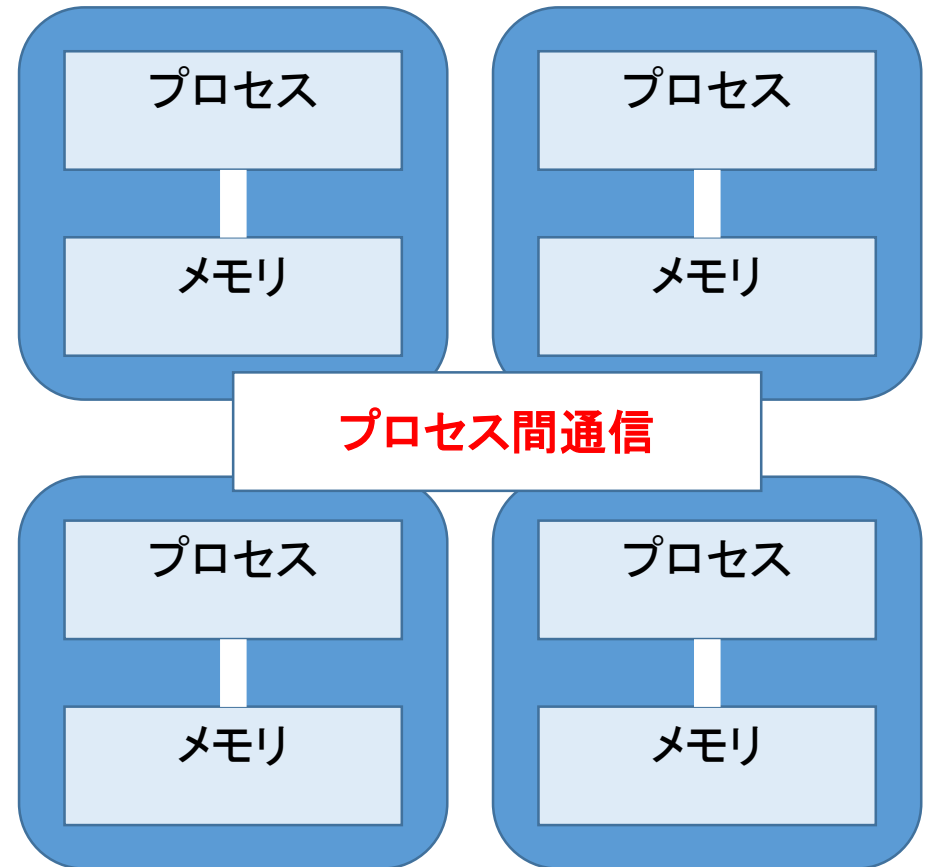
* 本日の演習では、本当に最小限の命令しか使いません。

スレッド並列とプロセス並列

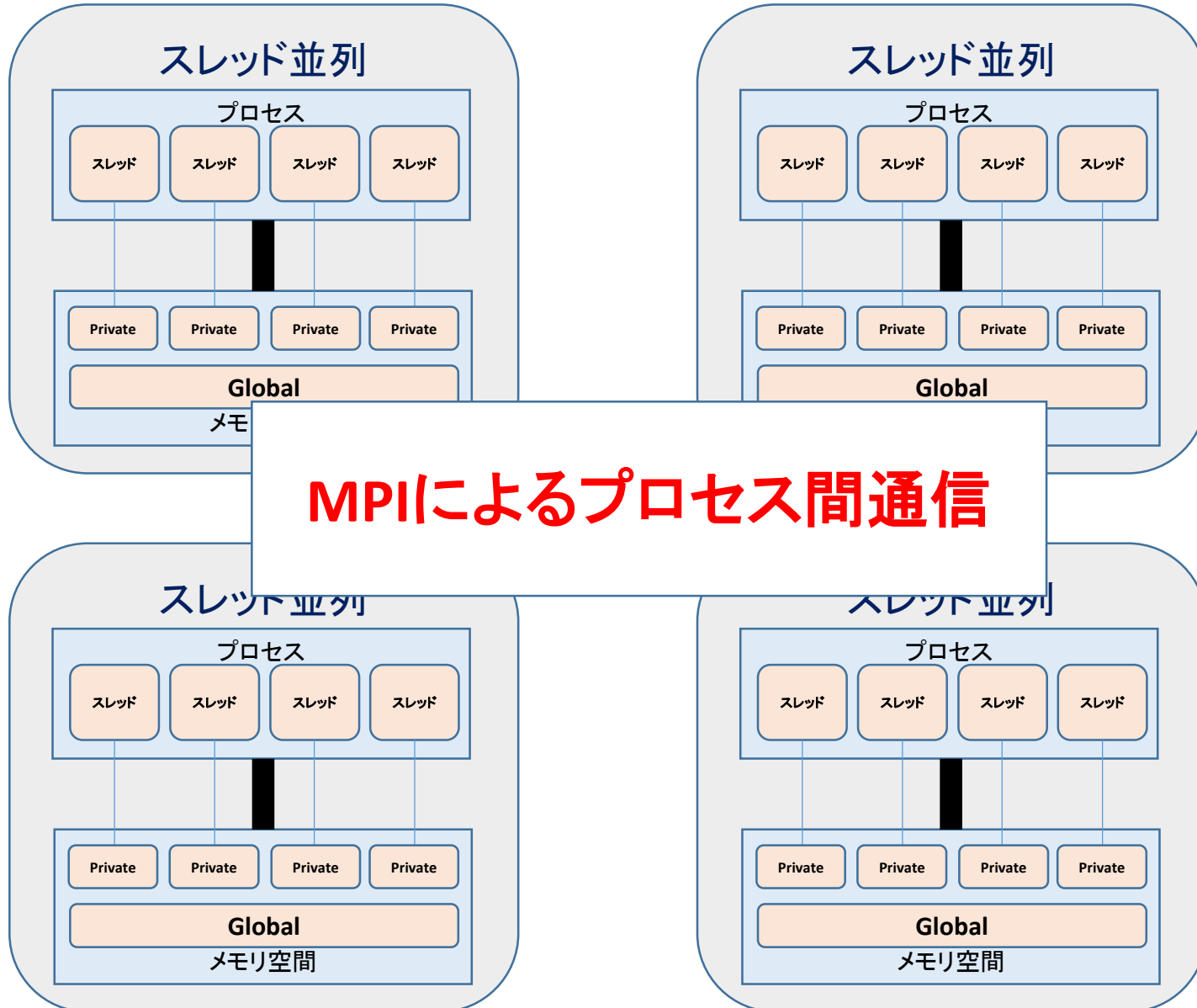
スレッド並列 OpenMP、自動並列化



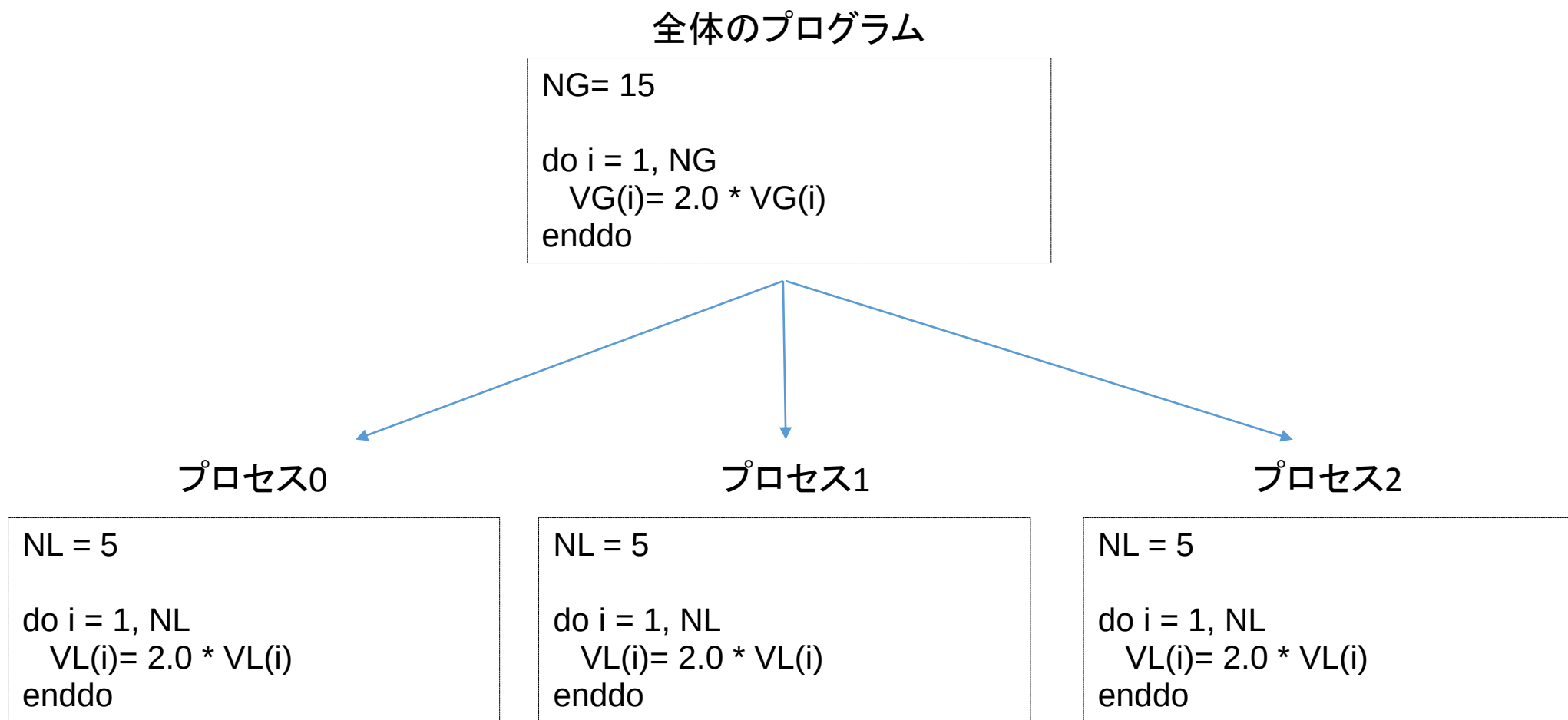
プロセス並列 MPI



ハイブリッド並列



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

SPMDの考え方

全体データ(VG)



プロセス0



プロセス1



プロセス2



局所データ(VL)



局所データ(VL)



局所データ(VL)

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

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

F

```
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;
```

C

```
    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
```



- 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やっていることになる

- まさにSPMD

#!/bin/sh	
#PJM =L "node=1"	ノード数
#PJM =L "elapsed=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

mpif.h / mpi.h

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

- ・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

#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_INIT / MPI_Init

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

F

```
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
```

- ・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

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

C

```
    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();  
}
```

<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
```

F

- ・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.

```
#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

<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

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

```
#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_COMM_RANK / MPI_Comm_Rank

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

F

```
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;
```

C

```
    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 : コミュニケーター
- 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 : コミュニケーターを指定
- 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。 (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 (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 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)
```

各種変数宣言

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

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
```

配列にデータ代入

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

iup=右隣りのプロセス
idown=左隣のプロセス

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

周期境界条件

```
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)
```

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

```
call mpi_finalize(ierr)
```

```
end
```

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)
```

差分計算に必要な「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

2017spring/code/openmp/1d_v102omp.f90 移流のみ
2017spring/code/openmp/1d_v204omp.f90 流体1次元
2017spring/code/openmp/2d_v101omp.f90 流体2次元
など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

ジョブスクリプトの投入

```
[xxxx@pi]$ mpifrtpx -Kopenmp 1d_v102hb.f90
```

```
[xxxx@pi]$ pjsub run.sh
```

```
run.sh
```

```
#!/bin/sh
```

```
#PJM -L "node=8"
```

```
#PJM -L "elapse=00:01:00"
```

```
#PJM =L "rscgrp=small"
```

```
#PJM -j
```

```
#PJM -o "output.lst"
```

```
#PJM --mpi "proc=8"
```

```
export OMP_NUM_THREADS=16
```

```
mpiexec ./a.out
```

ノード数

実行時間

実行キュー

標準出力ファイル名

起動するプロセス数

スレッド数を指定

実行ファイル

補足： π コンピュータ上の操作

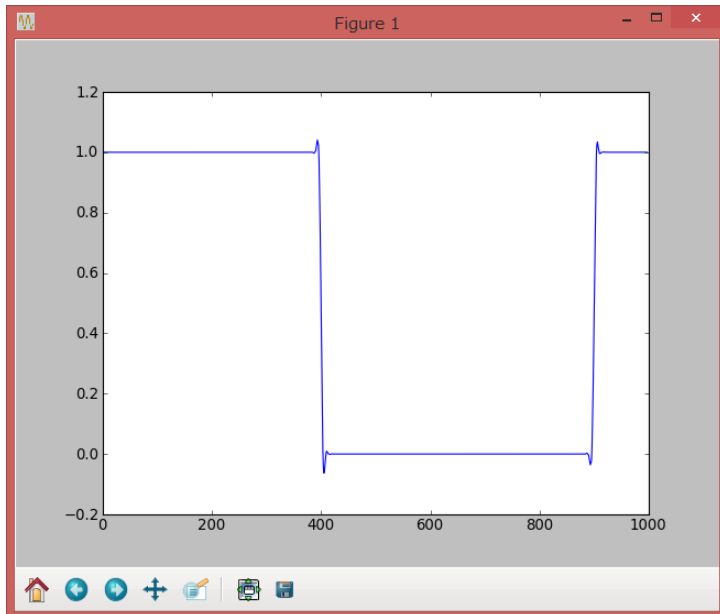
データ出力の確認(Pythonプログラム)

MPI版

```
[xxxx@pi]$ cd ~/2017spring/code/hybrid/dat_1d_v1  
[xxxx@pi dat_1d_v1]$ plot1dmpi.py 010000 8
```

ステップ番号

プロセス数



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

```
#PJM --mpi "proc=yy"
```

```
OMP_NUM_THREADS=zz
```

* ただし実行キュー「small」の最大ノード数は12