

計算科学演習 I

MPIを用いた並列計算 (I)

神戸大学大学院システム情報学研究科

谷口 隆晴

yaguchi@pearl.kobe-u.ac.jp

この資料は昨年度担当の横川先生の資料を参考にさせて頂いています。

講義概要

- 分散メモリ型計算機上のプログラミング
- メッセージ・パッシング・インターフェイス
(Message Passing Interface, MPI)
- MPIプログラム (M-1) : Hello, world! **コンパイル・実行方法**
- MPIプログラム (M-1) : 1対1通信関数 **基本的な考え方
(SPMD) に慣れる**
- MPIプログラム (M-3) : 集団通信関数

分散メモリ型並列計算機

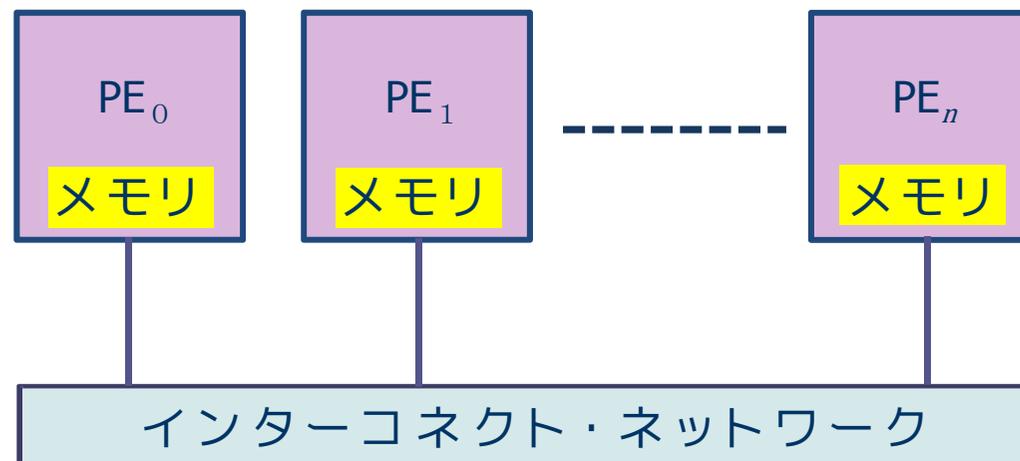
- 複数のプロセッサがネットワークで接続。
- それぞれのプロセッサ（PE）がメモリを保有。
 - 各PEが自分のメモリ領域のみアクセス可能。

特徴

数千から数万PE規模の並列システムが可能。

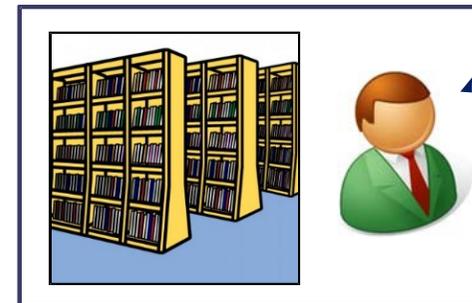
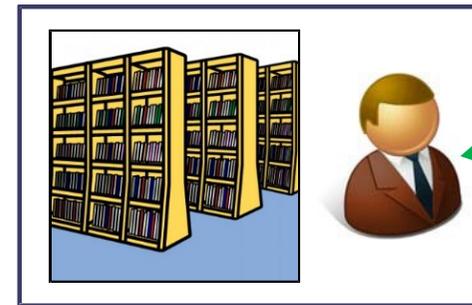
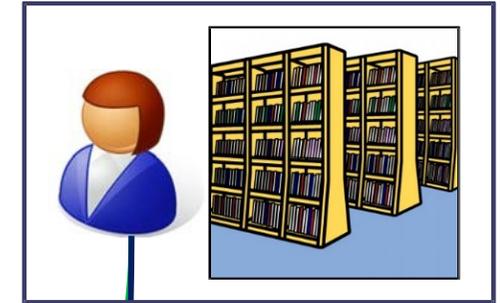
PEの間のデータ分散を意識したプログラミングが必要。

プログラミング技術：メッセージ・パッシング・インターフェイス（MPI）。



プログラミングのイメージ

- 離れた場所にいる複数の人に仕事を依頼.
- それぞれの人が, 本棚に一連のノートを持.
 - ◆ それぞれの人には名前が付いている (人を区別できる)
 - ◆ ノートには同じ名前が付けられているが中身は違っている.
- 本棚のノートをそれぞれが, 読んだり書いたり...
 - ◆ 大体は同じ作業だが, 最初のノートの中身が違うので中身はそれぞれ違う.
 - ◆ ある人に, 他の人とは違う作業をさせたい場合には, 名前で作業を指示.
- 時々, 他の人のノートを見たい.
 - ◆ 相手にノートの中身を送ってあげる.
 - ◆ 送られた人は, それを違う名前のノートに中身を書き写す.

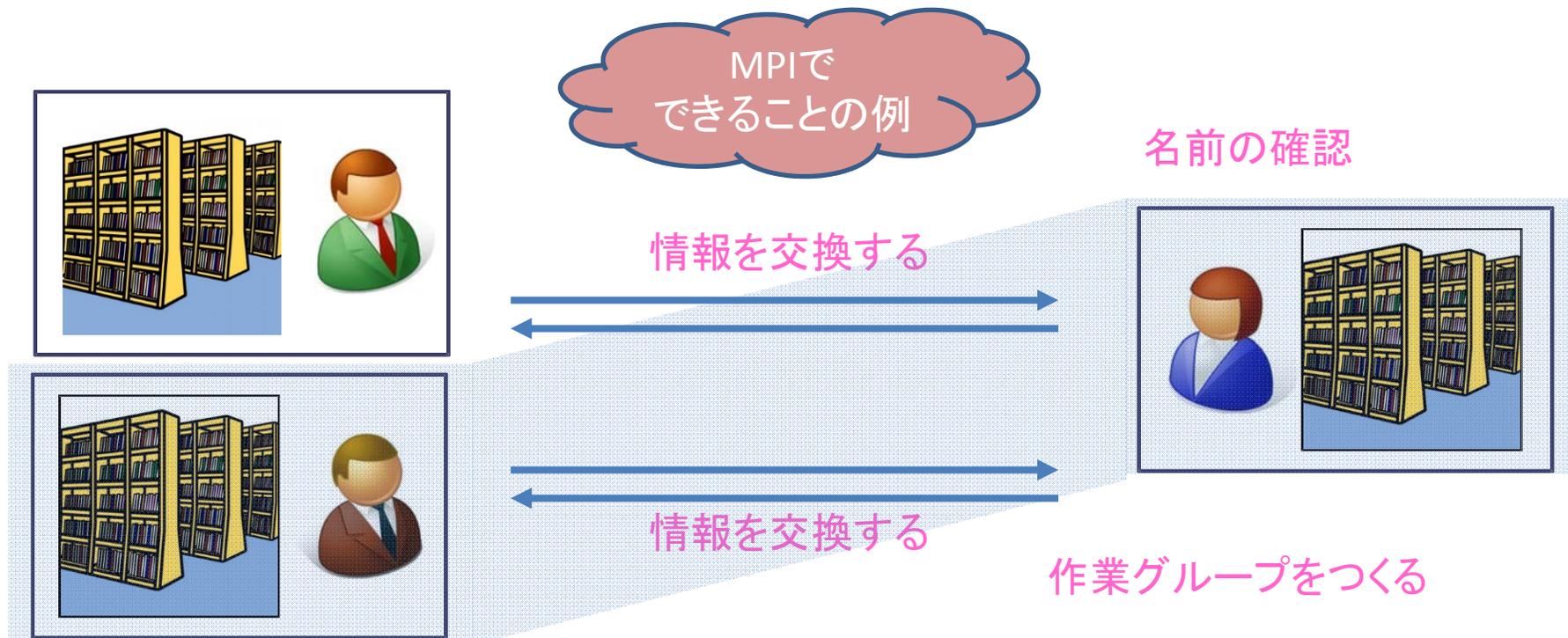


【必要な情報】

- ◆ ノートの名前
- ◆ 何冊
- ◆ 誰に
- ◆ 荷物のタグなど

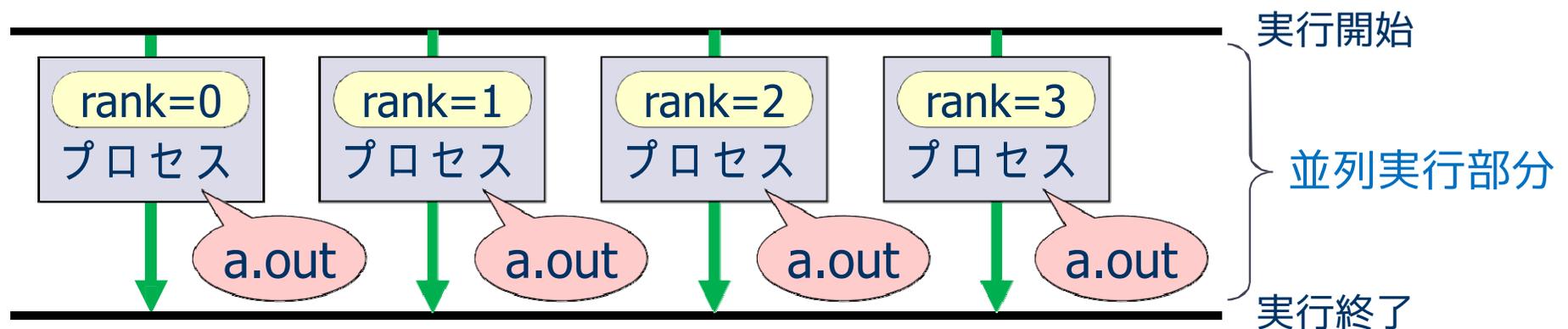
Message Passing Interface (MPI)

- 複数の独立したプロセス間で，並列処理を行うためのプロセス間メッセージ通信の標準規格.
- 1992年頃より米国の計算機メーカー，大学などを中心に標準化.
- 最新版はMPI3.1（2015年6月.）



MPIの実行モデル：SPMD (Single Program, Multiple Data)

- 複数のプロセス (=離れた所にいる人) により並列実行.
- 実行開始から終了まで,
 - 全プロセスが**同じプログラム**を実行.
 - 各プロセスは**固有の番号 (ランク番号)**を持つ.
 - ◆ P個のプロセスで実行する場合,
プロセス番号は0から(P-1)までの整数
 - ◆ **この番号によって仕事を変える** ← プログラムが頑張る.



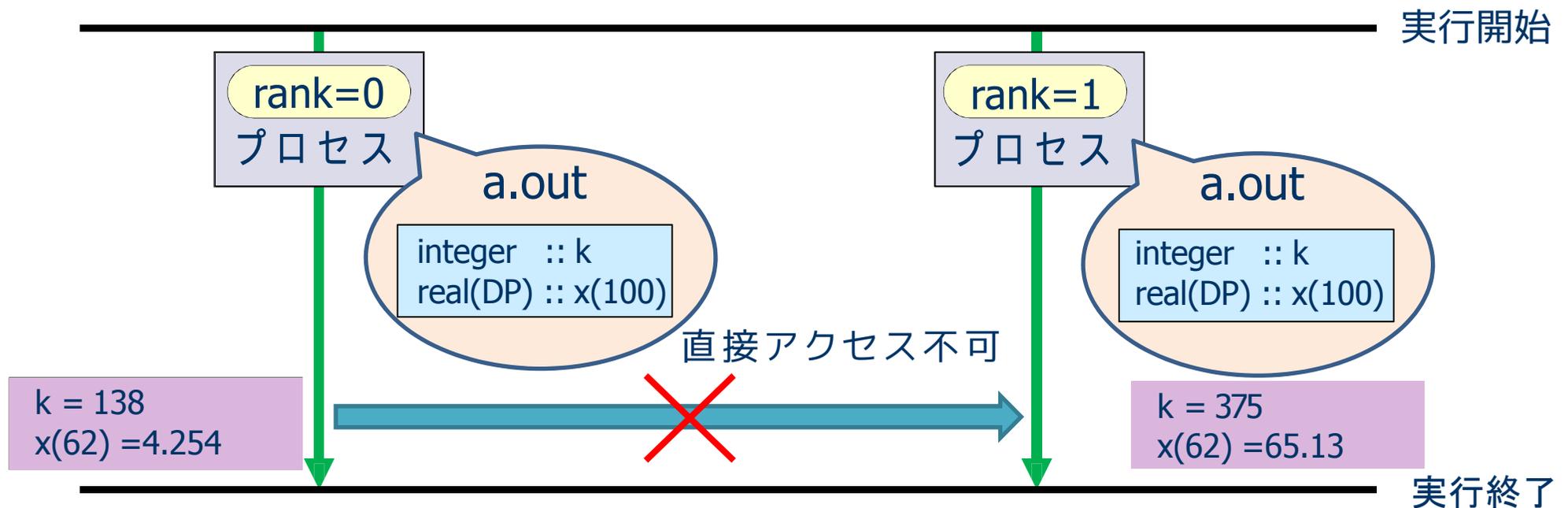
MPIでのメモリ空間とプロセス間通信

MPIでのメモリ空間：プロセスごとに**独立して保持**

プログラム中で定義された変数や配列は、
同じ名前で独立に各プロセスのメモリ上に割り当てられる。

同じ名前の変数や配列にプロセスごとに違う値を与えることが可能。
他のプロセスの持つ変数や配列には直接アクセスできない

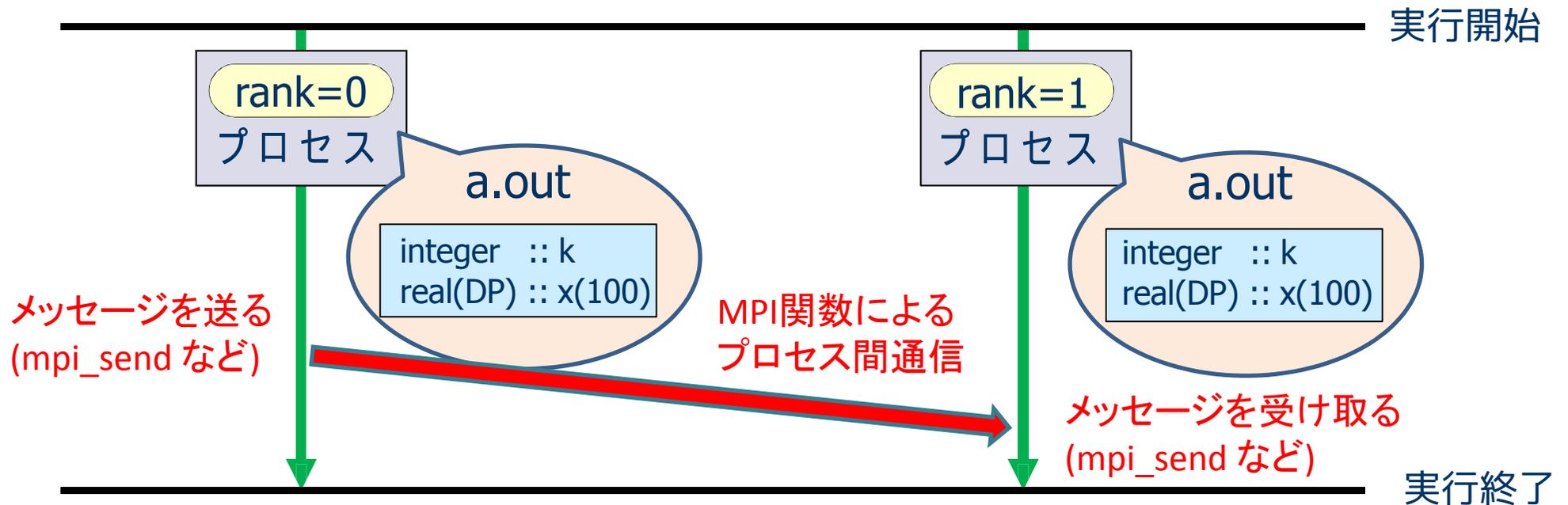
→ プロセス間通信でデータを送ってもらう。



メッセージパッシング方式

- メッセージパッシング方式：
送り手から受け手へメッセージ（データ）を伝達
（ \approx 荷物の配送，手紙）

- MPI \approx この方式によるプロセス間通信関数の集まり



MPIプログラムのスケルトン

```
program main
use mpi
implicit none
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 )
```

MPIモジュールの取り込み (おまじない1)

MPIで使う変数の宣言

MPIの初期化 (おまじない2)

MPIで使うプロセス数を nprocs に取得
自分のプロセス番号を myrank に取得

(この部分に並列実行するプログラムを書く)

```
call mpi_finalize( ierr )
```

MPIの終了処理 (おまじない3)

```
end program main
```

🔗 それぞれのプロセスに何をさせるかは myrank の値で場合分けし、うまく仕事を割り振る。

関数の説明

- `call mpi_init(ierr)`
MPIの初期化を行う。MPIプログラムの最初に必ず書く。
- `call mpi_comm_size(MPI_COMM_WORLD, nprocs, ierr)`
MPIの全プロセス数を2番目の引数 `nprocs` (整数型) に取得。
`MPI_COMM_WORLD`はコミュニケータと呼ばれるものの一つで、
「最初に割り当てられる全てのプロセスの集合」を表す。
- `call mpi_comm_rank(MPI_COMM_WORLD, myrank, ierr)`
自分のプロセス番号 (0から`nprocs-1`のどれか) を、
2番目の引数 `myrank` (整数型) に取得。
- `call mpi_finalize(ierr)`
MPIの終了処理をする。MPIプログラムの最後に必ず書く。

π-コンピュータでのプログラムの作業手順

① エディタを用いてプログラム (xxx.f90) を作成 (修正) .

② プログラムのコンパイル : `mpifrtpx xxx.f90` ← これまでと違う. 注意!

③ ジョブスクリプトを作成 : エディタを用いて `job.sh` を作成.

④ ジョブを投入 : `pjsub (ジョブスクリプトファイル名)`
[INFO] PJM xxxx pjsub JOB nnnnn submitted.
【注意】 nnnnn がジョブ番号 (ジョブに与えられたシステム内で唯一の番号)

⑤ ジョブ状態の確認 : `pjstat`

⑥ 結果の確認 : `jobname.onnnnn` を確認.

ジョブスクリプト例（2プロセスで実行する場合）

<pre>#!/bin/bash #PJM -N "jobname" #PJM -L "rscgrp=small" #PJM -L "node=2" #PJM -L "elapsed=00:02:00" #PJM -j mpirexec -n 2 ./a.out</pre>	<p>シェル名</p> <p>ジョブ名（任意の名前）を指定 投入先のキュー 使用ノード数 最大実行時間 stderrをstdoutにマージ</p> <p>並列数を指定してプログラムを実行</p>
---	---

4プロセスで実行する場合は 2 を 4 に変更。

MPIプログラム (M-1) : Hello, world!

```
program hello_by_mpi

use mpi
implicit none

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 )

print *, 'Hello, world! My rank number and nprocs are', myrank, ',', nprocs

call mpi_finalize( ierr )

end program hello_by_mpi
```

演習M1-1 : Hello, world! を並列に出力する.

MPI版 "Hello, world!" を **2, 及び4プロセス** で実行し, 結果を確認せよ (以下の通りに実行する)

<pre>\$ mkdir cpmpi \$ cd cpmpi</pre>	今日の演習用のディレクトリを作成.
<pre>\$ mkdir M-1 \$ cd M-1</pre>	演習M-1用のディレクトリを作成.
<pre>\$ cp /tmp/cpmpi/M-1/hello_mpi.f90 ./</pre>	ソースプログラム hello_mpi.f90 をカレントディレクトリにコピー. ※ 中身を見て確認すること.
<pre>\$ mpifrtpx hello_mpi.f90</pre>	ソースプログラムをコンパイル.
<pre>\$ cp /tmp/cpmpi/M-1/go.sh ./</pre>	ジョブスクリプト go.sh をコピー. (プロセス数の指定など, 必要な部分を編集)
<pre>\$ pjsub go.sh [INFO] PJM 0000 pjsub Job nnnnn submitted.</pre>	ジョブを投入し, 実行結果を確認.
<pre>\$ cat hello.onnnnn</pre>	※ hello は, go.sh 内で指定されている jobname

プログラム M-1の実行結果の確認

2 プロセスでの実行結果

```
Hello, world! My rank number and nprocs are 0 , 2  
Hello, world! My rank number and nprocs are 1 , 2
```

4 プロセスでの実行結果

```
Hello, world! My rank number and nprocs are 2 , 4  
Hello, world! My rank number and nprocs are 0 , 4  
Hello, world! My rank number and nprocs are 3 , 4  
Hello, world! My rank number and nprocs are 1 , 4
```

(注意) 出力はランク順に並ぶとは限らない。
また、実行ごとに出力の順番が異なることが多い。

ポイント

- 各プロセスが同じプログラムを実行。
- 各プロセスのランク番号(myrankの値) は異なっている。

プログラムM-1の説明（2プロセスの場合）

どのように実行されたのか？

```
program hello_by_mpi
use mpi
implicit none
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 )

print *, 'Hello, world! My rank number and nprocs are', myrank, ',', nprocs

call mpi_finalize( ierr )
end program hello_by_mpi
```

1つ目のプロセスでは0
2つ目のプロセスでは1

1つ目のプロセスが実行する文

```
print *, 'Hello, world! My rank number and nprocs are', 0, ',', nprocs
```

2つ目のプロセスが実行する文

```
print *, 'Hello, world! My rank number and nprocs are', 1, ',', nprocs
```

myrank の値が変わる
→ これを使って役割分担！

MPIプログラム (M-2) : 1対1通信関数

【問題】

1から100までの整数の和を2並列で求めなさい。

プログラム作成の方針

- プロセス0: 1から50までの和を計算。
プロセス1: 51から100までの和を計算。
- プロセス1の結果をプロセス0に転送。
- プロセス0が自分の結果と転送された結果を足して出力。

MPIプログラム M-2

```
program sum100_by_mpi
use mpi
implicit none
integer :: i, istart, iend, isum_local, isum_tmp
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 )
istart = myrank*50 + 1
iend   = (myrank+1)*50
isum_local = 0
do i = istart, iend
    isum_local = isum_local + i
enddo
if( myrank == 1 ) then
    call mpi_send( isum_local, 1, MPI_INTEGER, 0, 100, MPI_COMM_WORLD, ierr )
else
    call mpi_recv( isum_tmp, 1, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, istat, ierr )
end if
if( myrank == 0 ) print *, 'sum =', isum_local+isum_tmp
call mpi_finalize( ierr )
end program sum100_by_mpi
```

MPIプログラム M-2の説明

```
program sum100_by_mpi
use mpi
implicit none
integer :: i, istart, iend, isum_local, isum_tmp
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 )
istart = myrank*50 + 1
iend = (myrank+1)*50
isum_local = 0
do i = istart, iend
  isum_local = isum_local + i
enddo
if( myrank == 1 ) then
  call mpi_send( isum_local, 1, MPI_INTEGER, 0, 100, MPI_COMM_WORLD, ierr )
else
  call mpi_recv( isum_tmp, 1, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, istat, ierr )
end if
if( myrank == 0 ) print *, 'sum =', isum_local+isum_tmp
call mpi_finalize( ierr )
end program sum100_by_mpi
```

青：MPIプログラムのおまじない（既出）
緑：プロセス番号（ランク）に応じた処理
赤：MPI関数によるプロセス間通信

ランクの値から自分の計算範囲を求める

各プロセスが
部分和を計算

プロセス1はプロセス0に
自分の部分和を送信

プロセス0が総和を出力

プロセス0はプロセス1
から部分和を受信
（変数名に注意）

1対1通信 – 送信関数 mpi_send (送り出し側)

```
mpi_send( buff, count, datatype, dest, tag, comm, 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)
- ◆ ierr: 戻りコード (整数型)

1対1通信 – 受信関数 mpi_recv (受け取り側)

```
mpi_recv( buff, count, datatype, source, tag, comm, status, 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．

- ◆ 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)
- ◆ 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) ↑ send と違うところ

関数の引数に関する注意（共通）

■ buff

- ◆ 送信するデータは領域は、**メモリ上で連続アドレス**でなければならない。
“先頭アドレスから $x \times$ バイトを送れ” という関数なので。
- ◆ 他の通信関数でも同じ。
- ◆ したがって、メモリ上で離れた変数を同時に送りたい場合は、別の変数に連続してパック（pack）させてから送る必要がある。

■ datatype

- ◆ **予約語**がある。MPI_INTEGER（整数型）、MPI_REAL（単精度実数型）、MPI_DOUBLE_PRECISION または MPI_REAL8（倍精度実数型）など。
- ◆ バイト数を計算するために必要（c言語のsizeof()のようなもの）

■ tag

- ◆ 同じプロセスに複数回メッセージを送るとき、メッセージを受取ったプロセスが、どのメッセージかを区別するために使用。
- ◆ 受取側の mpi_recv では、メッセージに対応したtagで受け取る。
- ◆ 複数回のメッセージでも、送受信の順番などを区別できる場合は、同じtagを使っても良い。

演習M1-2：まずは実行してみる.

1 から 100 までの整数の和を2並列で求める
プログラムを2プロセスで実行し, 結果を確認せよ.

【手順】

- ① `/tmp/cpmpi/M-2/sum100_mpi.f90` を適切なディレクトリにコピーする.
- ② `/tmp/cpmpi/M-2/go.sh` をコピーして, ジョブを実行.
- ③ 結果 (`sum100.onnnnn`) を確認する.

出力に正しい答え (`sum = 5050`) が出力されているか?
(`myrank=0`のプロセスだけが出力していることに注意.)

演習M1-3 (提出課題1)

1 から 100 までの整数の和を求めるプログラムを,
4 並列で実行できるように修正し, 4 プロセスで実行せよ.

※ `mpi_send`, `mpi_recv` 関数を使うこと.

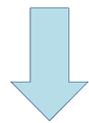
プログラム修正のポイント

- ◆ 各プロセスの部分和を計算する際, その範囲を `myrank` うまく使って求める (`myrank` は 0-3 の整数).
- ◆ `myrank ≠ 0` 以外のプロセスから `myrank=0` のプロセスに部分和を送信する.
- ◆ プロセス0 (`myrank=0`) は, 他の3つのプロセスから送られた部分和を受信し, 全体の和を計算する.

集団通信関数

1対1通信関数は複雑になりやすい

特にプロセス数が多くなると非常に複雑
→ バグも入りやすい。



もっと楽にできないの？

集団通信関数

◆ mpi_bcast

あるプロセスから全てのプロセスに値を一斉に配布する関数。

◆ mpi_reduce

すべてのプロセスから、あるプロセスに値を集めて、
何らかの演算（+，x，max，min）をする関数。

MPIプログラム M-3 (集団通信関数を使う)

```
program sum_by_reduction
use mpi
implicit none
integer :: n, i, istart, iend, isum_local, isum
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 )
if( myrank==0) n=10000
call mpi_bcast( n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )
istart = (n/nprocs)*myrank + 1
iend   = (n/nprocs)*(myrank+1)
isum_local = 0
do i = istart, iend
    isum_local = isum_local + i
enddo
call mpi_reduce( isum_local, isum, 1, MPI_INTEGER, MPI_SUM, 0, &
                MPI_COMM_WORLD, ierr )
if( myrank == 0 ) print *, 'sum (by reduction function) =', isum

call mpi_finalize( ierr )
end program sum_by_reduction
```

MPIプログラム M-3の説明

```
program sum_by_reduction
use mpi
implicit none
integer :: n, i, istart, iend, isum_local, isum
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 )
if( myrank==0 ) n=10000
call mpi_bcast( n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )
istart = (n/nprocs)*myrank + 1
iend   = (n/nprocs)*(myrank+1)
isum_local = 0
do i = istart, iend
    isum_local = isum_local + i
enddo
call mpi_reduce( isum_local, isum, 1, MPI_INTEGER, MPI_SUM, 0, &
                MPI_COMM_WORLD, ierr )
if( myrank == 0 ) print *, 'sum (by reduction function) =', isum

call mpi_finalize( ierr )
end program sum_by_reduction
```

青：MPIプログラムのおまじない（既出）
緑：プロセス番号（ランク）に応じた処理
赤：MPI関数によるプロセス間通信

プロセス0がnの値をセット

nの値を放送

ランクの値から自分の計算範囲を計算

各プロセスが
部分和を計算

部分和の総和を計算
(プロセス0に集約)

プロセス0だけが結果を出力

集団通信 – broadcast

```
mpi_bcast( buff, count, datatype, root, comm, ierr )
```

※ ランク番号`root`のプロセスが持つ`buff`の値を, `comm`で指定されたグループに属する, 他のすべてのプロセスの`buff`に配布する.

- ◆ `buff`: 送り主 (`root`) が送信するデータの変数名 (先頭アドレス)
他のMPIプロセスは, 同じ変数名でデータを受け取る.
- ◆ `count`: データの個数 (整数型) .
- ◆ `datatype`: 送信するデータの型.
MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION など.
- ◆ `root`: 送り主のMPIプロセス番号.
- ◆ `comm`: コミュニケータ (例えば, MPI_COMM_WORLD) .
- ◆ `ierr`: 戻りコード (整数型) .

集団通信 – reduction

```
mpi_reduce( sendbuff, recvbuff, count, datatype, op, root, comm, ierr )
```

※ commで指定されたすべてのプロセスからデータを，ランク番号 root のプロセスに集め，演算（op）を適用した結果を recvbuff に設定する。

- ◆ sendbuff: 送信するデータの変数名（先頭アドレス）
- ◆ recvbuff: 受信するデータの変数名（先頭アドレス）
- ◆ count: データの個数（整数型）
- ◆ datatype: 送信するデータの型
MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISIONなど
- ◆ op: 集まってきたデータに適用する演算の種類
MPI_SUM(総和), MPI_PROD(掛け算), MPI_MAX(最大値)など
- ◆ root: データを集めるMPIプロセス番号
- ◆ comm: コミュニケータ（例えば，MPI_COMM_WORLD）
- ◆ ierr: 戻りコード（整数型）

演習M1-4 プログラムM-3を動かしてみる

1からnまでの整数の和を並列で求めるプログラム M-3を
2, 4 及び 8 プロセスで実行し, 結果を確認せよ.

※ n=10000 (8で割り切れる値) は変更しないこと.

【手順】

- ① /tmp/cpmmpi/M-3/summ3.f90 を適切なディレクトリにコピー.
- ② /tmp/cpmmpi/M-3/go.sh をコピーして, ジョブを実行.
- ③ 結果を確認する.

出力に正しい答え (50005000) が出力されているか?

演習M1-5（提出課題2）

プログラム M-3を以下のように書き換え，
8プロセスで実行せよ．

- ◆ 変数 `isum_local`, `isum`を倍精度実数 `sum_local`, `sum`に変更
- ◆ `mpi_reduce` でのリダクション演算も倍精度で行うように変更

修正の方針

- ◆ 倍精度実数型変数の宣言の仕方を思い出す．
- ◆ `mpi_reduce`内の`datatype` を適切に変更する．

課題の提出方法と提出期限

■ 演習M1-3, M1-5 の提出方法

- ① 課題ごとに修正した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

```
$ nkf -Lu report-xx.txt | mail -s "1-n:tnnnnnnn" kobeuniv.compra1@gmail.com
```

Note) **tnnnnnnn** は自分のログインID(学籍番号)
番号1-nは、演習課題番号(M1-3なら、1-3など)

■ 締切

6月28日(火) ※ うまく動かない場合でも、途中結果を提出せよ。