第141回 お試しアカウント付き 並列プログラミング講習会 「MPI基礎：並列プログラミング入門」

東京大学 情報基盤センター
三木 洋平

講習会概略

- 開催日：2020年10月13日（火） 10:00–17:00
- 形態：ZoomおよびSlackを用いたオンライン講習会
- 使用システム：Oakforest-PACS (OFP)
- 講習会プログラム：
  - 10:00–11:20 テストプログラムの実行など（演習）
  - 11:30–12:30 並列プログラミングの基本（座学）
    (12:30–13:40昼休み)
  - 13:40–14:40 MPIプログラム実習1（演習）
  - 14:50–15:50 MPIプログラム実習2（演習）
  - 16:00–17:00 MPIプログラム実習3（演習）

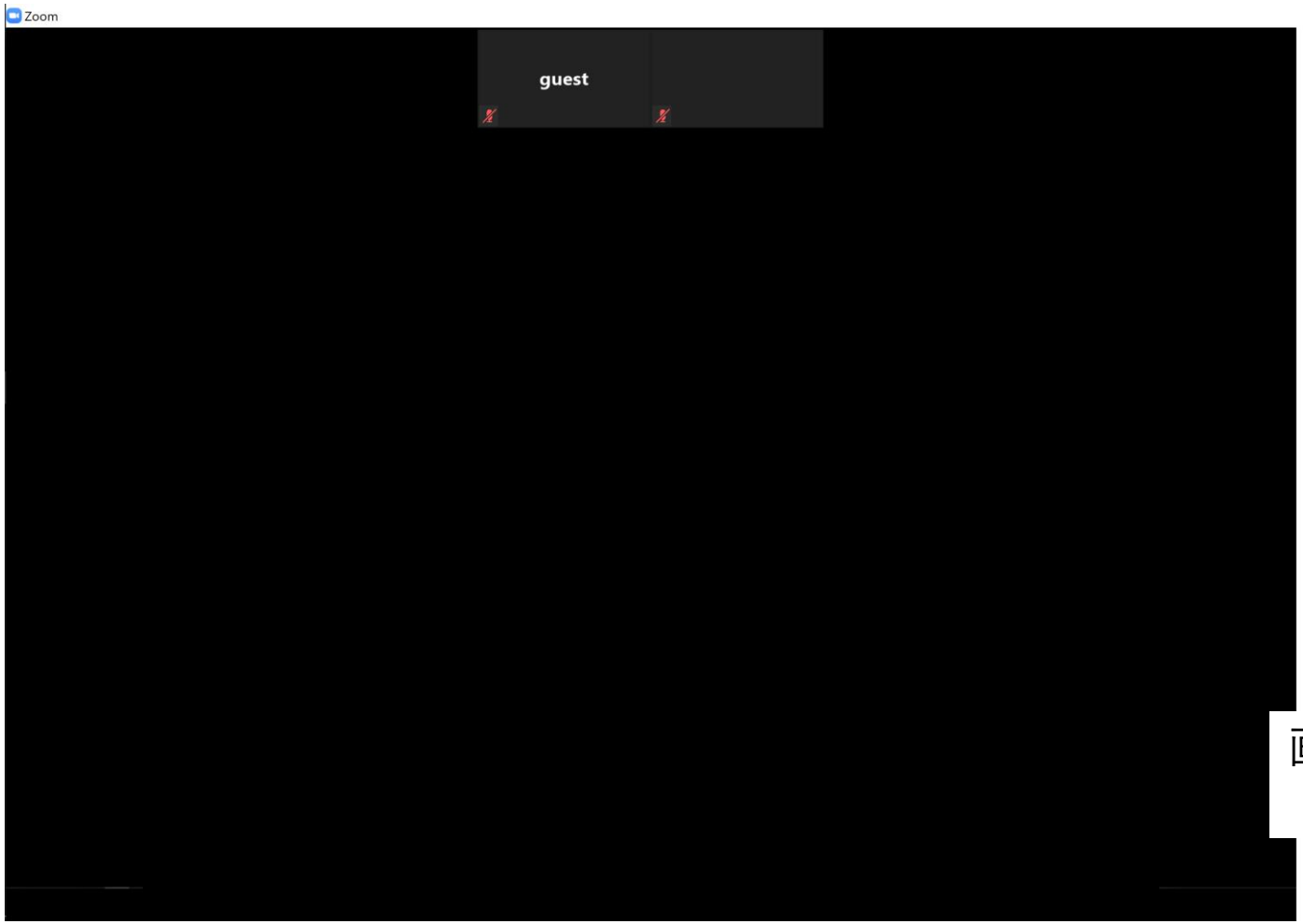
Zoom関連

- 「手をあげる」機能
  - 質問がある際，全体の状況を確認するため使用
- ブレークアウトセッション
  - 画面を共有しながらエラー対応する際に使用
  - （なるべく口頭でのやりとりやSlackで対応する予定）

- https://utelecon.github.io/zoom/how_to_use

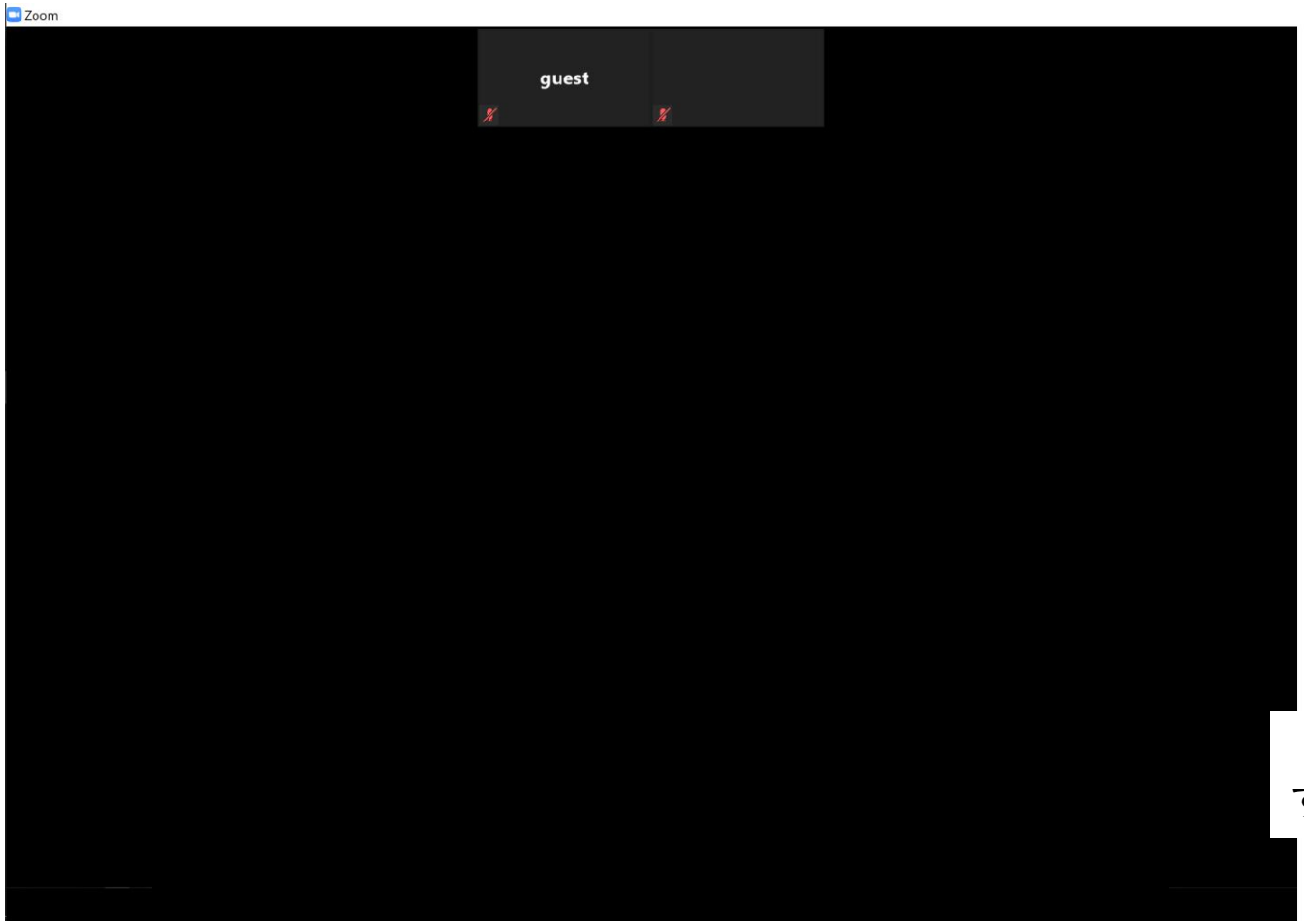
「手を挙げる」機能の使い方（1/3）

「手を挙げる」機能の使い方（2/3）



画面右下の
「手を挙げる」をクリック

「手を挙げる」機能の使い方（3/3）



自分のところに
手のアイコンが現れる

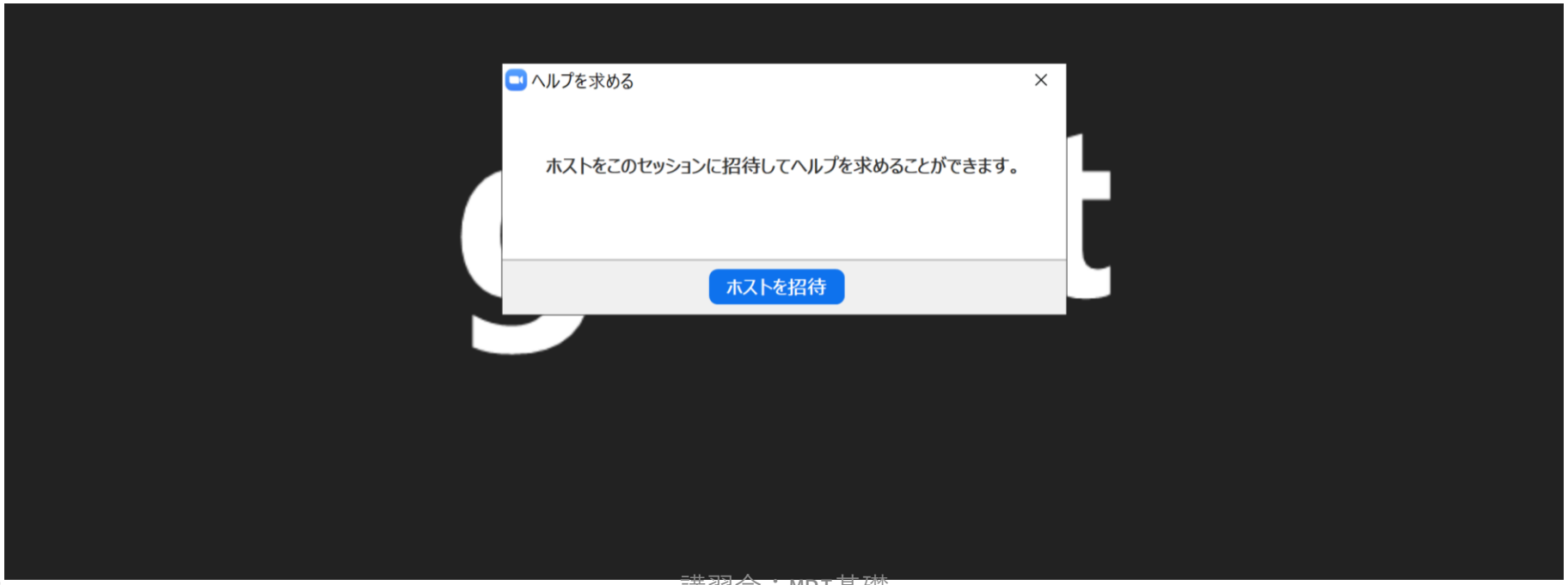
「手を降ろす」をクリックすると，元に戻る

ブレークアウトセッション（1/4）

- 演習時に使用するかもしれません
- 演習中に「ヘルプを求める」ことができます
  - ホストを招待した後に「画面を共有」することで，皆さんの記述したプログラムを一緒に見ながら問題解決にあたります

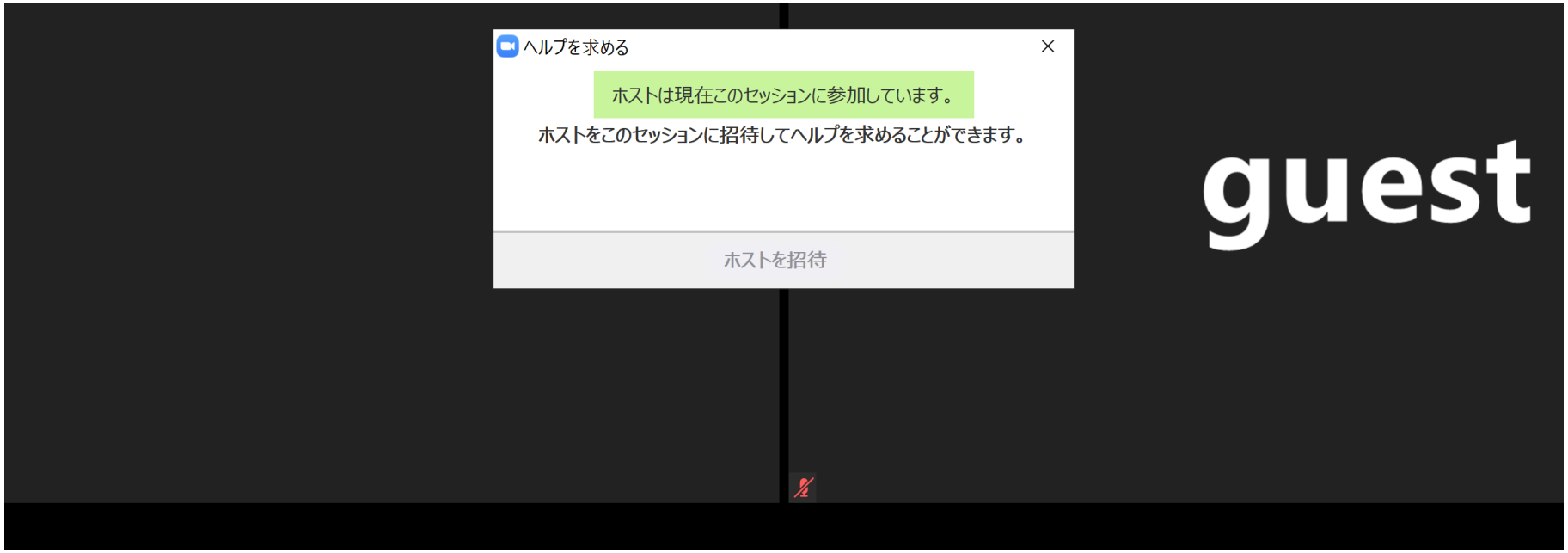
ブレークアウトセッション（2/4）

- この表示が出たら，「ホストを招待」をクリック
- ホストの承認待ちに移行
  - 他の受講者のヘルプ中など，直ちに対応できない場合もあります

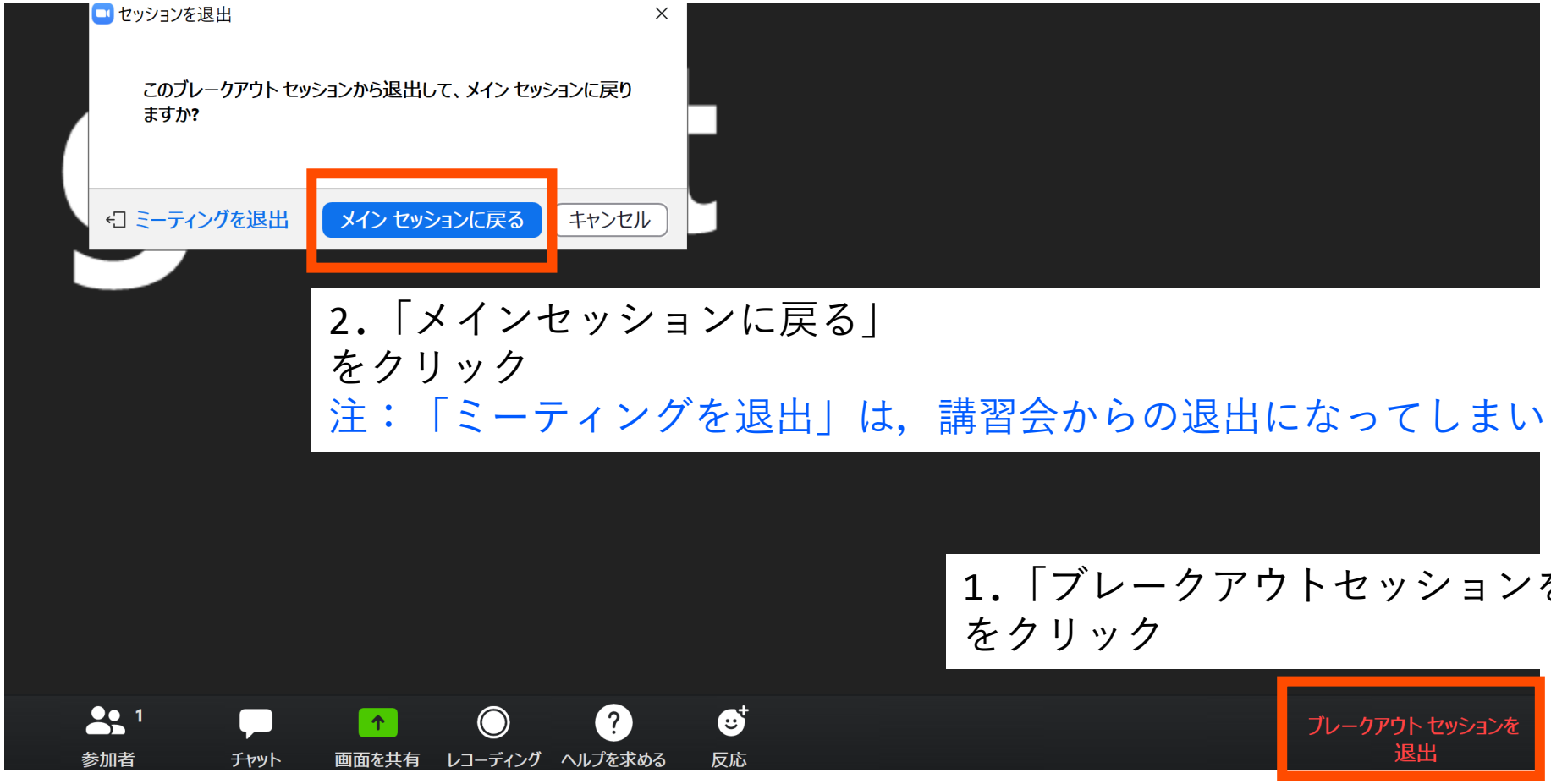


ブレークアウトセッション（3/4）

- この状態になると，ホスト（講師）と会話可能
  - マイクの有効化，画面の共有などをしながら相談



ブレークアウトセッション（4/4）



Slack関連

- ブラウザ上で使う場合には：
  - https://w1590055008-bgo338004.slack.com/
  - 注：ログインには，事前にお配りしたリンクからの登録が必要です
- 質問対応に使用
- コードの貼り付け方
- スレッドの確認方法

- 以下，ブラウザ版で説明しますがアプリ版でも操作は同じです

質疑応答チャンネルへの移動

左側の「チャンネル」の中に
「#第141回-mpi基礎」があるので，クリック

「#第141回-mpi基礎」が見つからない場合は，
「チャンネル」の右側の「＋」をクリックし，
さらに「チャンネル一覧」をクリック．
チャンネル一覧の中に「#第141回-mpi基礎」
があるので，「参加する」をクリック

メッセージの入力方法


Slack | 第141回-mpi基礎 | 東大ス
閉じる
レット (2)
その他のブックマーク
東大スパコン
スレッド
すべての DM
その他
チャンネル
general
random
第133回-gpuプログラミ...
第141回-mpi基礎
ダイレクトメッセージ
#第141回-mpi基礎
9月30日、あなたがこのチャンネルを作成しました。#第141回-mpi基礎 チャンネルをどんどん活用していきましょう！ 説明： 第141回 お試しアカウント付き並列プログラミング講習会「MPI基礎：並列プログラミング入門」の質疑応答 (編集)
アプリを追加する
メンバーを追加する
9月30日 (水)
Yohei MIKI 11:14
#第141回-mpi基礎 に参加しました。
Yohei MIKI 11:15
チャンネルの説明を設定しました : 第141回 お試しアカウント付き並列プログラミング講習会「MPI基礎：並列プログラミング入門」の質疑応答
最下部に入力欄があるので，質問内容を記載して
Ctrl+Enter する
（右下の「メッセージを送信する」でも同じ）
#第141回-mpi基礎 へのメッセージ
講習会：MPI基礎
2020/10/13
App

コードの貼り付け方

「コードブロック」とあるものをクリックすると枠が生成されるので，この中にコピペするのがやりやすい
Ctrl+Alt+Shift+C でも良いが，これはやりづらい
```（Shift+@を3連打）しても良い
```

スレッドの確認方法


スレッド
すべての DM
その他
チャンネル
general
random
第133回-gpuプログラミ...
第141回-mpi基礎
ダイレクトメッセージ
App
2020/10/13
左上の「スレッド」をクリックすると，自分が
参加しているスレッドの一覧が表示されます
#第141回-mpi基礎
9月30日、あなたがこのチャンネルを作成しました。#第141回-mpi基礎 チャンネルをどんどん活用していきましょう！ 説明： 第141回 お試しアカウント付き並列プログラミング講習会「MPI基礎：並列プログラミング入門」の質疑応答 (編集)
アプリを追加する
メンバーを追加する
9月30日 (水)
Yohei MIKI 11:14
#第141回-mpi基礎 に参加しました。
Yohei MIKI 11:15
チャンネルの説明を設定しました : 第141回 お試しアカウント付き並列プログラミング講習会「MPI基礎：並列プログラミング入門」の質疑応答
昨日
講習会：MPI基礎

ユーザアカウント

- 使用システム： Oakforest-PACS (OFP)
  - $ ssh USERNAME@ofp.jcahpc.jp
- 本講習会でのユーザ名
  - 利用者番号： tABCDE（ABCDEは，適宜書き換えてください）
  - 利用グループ： gt00
- 利用期限
  - 11/13 9:00まで有効

- 注:本講習会関連の質問はymiki[at]cc.u-tokyo.ac.jpまで
  - （講習会アカウントでは）公式の相談対応システムは使わないでください

テストプログラムの概要

- C言語版・Fortran版共通ファイル：
  mpi-samples.tar.gz
- tar で展開後，それぞれの言語用のディレクトリが作られる
  - c/ ： C言語用
  - fortran/ ： Fortran 95用
- 上記ファイルの置き場所：
  /work/gt00/z30118/MPI

サンプルプログラムの取得（1/2）

- 実行してもらうコマンドは $ 以降に青字で記載しています
  - ターミナルへの入力が終わったら 「Enter」 キーを押してください

1. Lustreファイルシステムに移動
   ```
   $ cd /work/gt00/tABCDE          # 下線部は自分のIDに変更
   ```
2. /work/gt00/z30118/MPI にあるサンプルファイルをコピー
   ```
   $ cp /work/gt00/z30118/MPI/mpi-samples.tar.gz .
   ```
   mpi-samples.tar.gz と .（ドット）の間に半角スペース
3. サンプルファイルを展開
   ```
   $ tar xvf mpi-sample.tar.gz
   ```

サンプルプログラムの取得（2/2）

4. mpi-samples ディレクトリに入る
```
$ cd mpi-samples
```
5. 自分の使いたい言語のディレクトリに入る
```
$ cd c              # C言語を使用する場合
$ cd fortran        # Fortranを使用する場合
```
6. サンプルプログラム（0番から5番まで）があることを確認
```
$ ls
```

サンプルプログラム

- 0_hello
- 1_hybrid
- 2_sum_relay
- 3_sum_binary
- 4_sum_reduce
- 5_diffusion

サンプルプログラム

- 0_hello
- 1_hybrid
- 2_sum_relay
- 3_sum_binary
- 4_sum_reduce
- 5_diffusion

並列版Helloプログラムをコンパイル

1. 0_hello ディレクトリに入る
   ```
   $ cd 0_hello
   ```
2. コンパイル
   ```
   $ make
   ```
3. 実行ファイル（hello）ができていることを確認
   ```
   $ ls
   ```

ジョブスクリプトの説明（フラットMPI）

- 内容はC言語，Fortranで共通

```
#!/bin/bash
#PJM -L rscgrp=lecture-flat
#PJM -L node=16
#PJM --mpi proc=1088
#PJM -L elapse=00:01:00
#PJM -g gt00

mpiexec.hydra
-n ${PJM_MPI_PROC} ./hello
```

リソースグループ名：lecture-flat

利用ノード数：16ノード使用

MPIプロセス数：1088 (= 68 * 16)

実行時間制限：1分

利用グループ名：gt00

MPIジョブを1088プロセスで実行

並列版Helloプログラムの実行

- ジョブスクリプト名は run.sh です
- 配布したサンプルではキュー名が“lecture-flat”になっているので，これを“tutorial-flat”に変更してください

```
$ emacs -nw run.sh    # emacs で編集する場合
$ vim run.sh          # vim   で編集する場合
$ nano run.sh         # nano  で編集する場合
```

- ジョブを投入

```
$ pjsub run.sh
```

並列版Helloプログラムの結果確認（1/2）

1. 自分が投入したジョブの状態を確認
   ```
   $ pjstat
   ```
2. ジョブの実行が終了すると，以下のファイルが生成される
   ```
   run.sh.oXXXXXXX    # 標準出力ファイル
   run.sh.eXXXXXXX    # 標準エラー出力ファイル
   ```
   ジョブ名 + .[o e] + ジョブID というファイル名になっています
3. 標準出力ファイルの中身を見てみる
   ```
   $ cat run.sh.oXXXXXXX
   ```
   “Hello world!”が1088 (= 68プロセス * 16ノード) 行あれば成功

並列版Helloプログラムの結果確認（2/2）

- 出力が多すぎるため，本当に1088個出力されているか確認したい
  →Hello worldの個数を数え上げる
  ```
  $ grep Hello run.sh.oXXXXXXX | wc -l
  ```
  1088 と表示されればO.K.
  - ｜（パイプ）は，Shift + ¥ （英字キーボードではバックスラッシュ）
- 出力がばらばらなので，きちんと連番になっているか確認したい
  →出力をソートして確認する
  ```
  $ grep Hello run.sh.oXXXX | sort -k 4 -n | less
  ```
  rank: 0 から始まって，rank: 1087 で終わっていればO.K.
  - less の表示を終了するには，qを入力して Enter する（quitのq）

並列版Helloプログラムの説明（C言語）

全プロセスがこのプログラムを起動

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  int err = MPI_Init(&argc, &argv);
  int size, rank;
  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello world! rank: %d¥n", rank);

  err = MPI_Finalize();

  return (0);
}
```

MPIの初期化

全プロセス数を取得
（全ランクで共通の値）

自分のIDを取得
（全ランクで異なる値）

MPIの終了

並列版Helloプログラムの説明（Fortran）

全プロセスがこのプログラムを起動

```
program main
  use mpi
  implicit none
  integer :: err
  integer :: size, rank

  call MPI_Init(err)
  call MPI_Comm_size(MPI_COMM_WORLD, size, err)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)

  print *, "Hello world! rank:", rank

  call MPI_Finalize(err)

  stop
end program main
```

MPIの初期化

全プロセス数を取得
（全ランクで共通の値）

自分のIDを取得
（全ランクで異なる値）

MPIの終了

Oakforest-PACSでのジョブ実行（1/2）

- 以下の2通りの実行形態があります

1．バッチジョブ実行

- バッチジョブシステムに処理を依頼して実行
- 実行したい処理をファイル（ジョブスクリプト）で指示
- スパコン環境で一般的
- 大規模実行用
  - OFPでは，最大2048ノード（139264コア），24時間まで

2．インタラクティブジョブ実行

- PCでの実行のように，コマンドを入力して実行
- スパコン環境では一般的ではない
- デバッグ用，大規模実行はできない
  - 1ノード（68コア）：2時間まで
  - 16ノード（1088コア）：10分まで

※講習会アカウントでは
バッチジョブ実行のみ，
最大16ノード15分まで

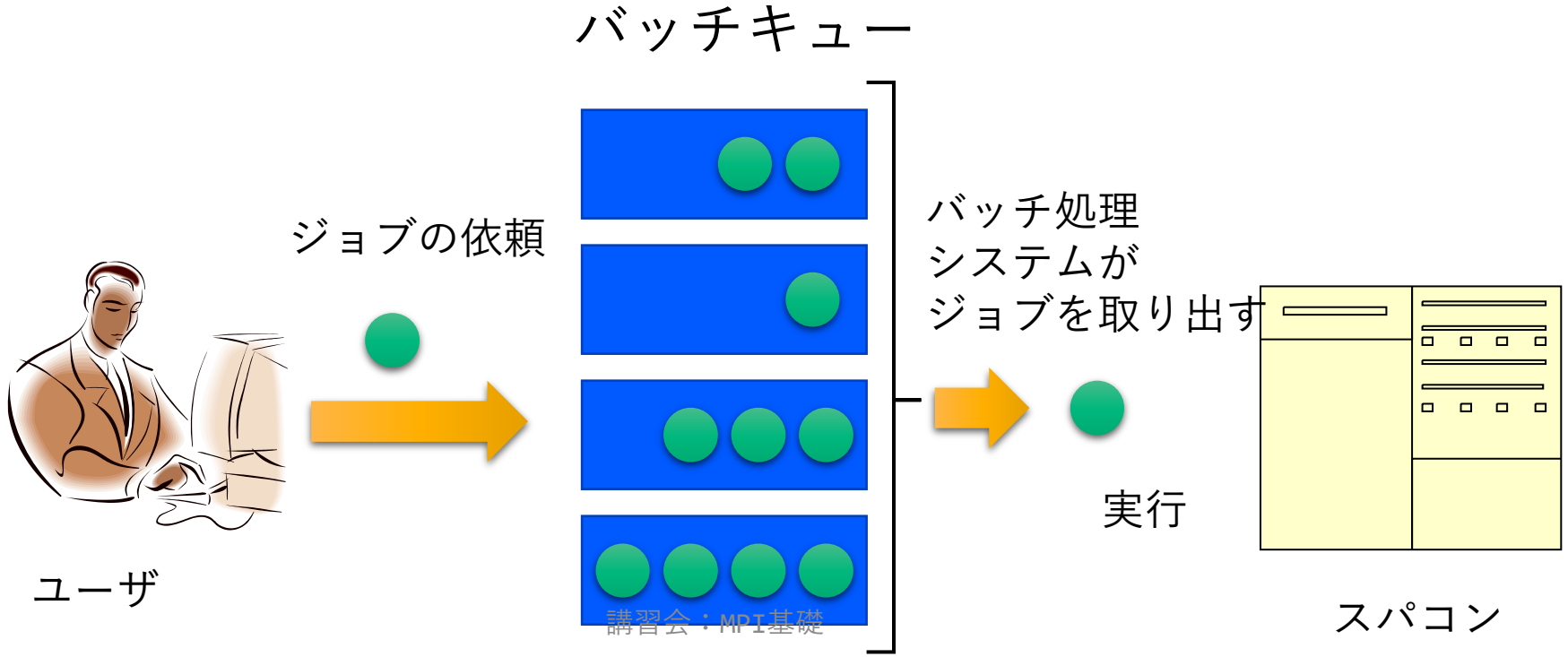
Oakforest-PACSでのジョブ実行（2/2）

- 2つの異なるメモリモードを用意
  1. Flatモード: MCDRAMとDDR4メモリを個別にアクセス可能
  2. Cacheモード: MCDRAMとDDR4メモリのキャッシュとして働く

- 各ジョブキューには，`-flat`，`-cache` をそれぞれ用意
  - 講習会アカウントでは，Flatモードだけが使えます

バッチ処理とは

- スパコン環境では，通常は，インタラクティブ実行（コマンドラインで実行すること）はできません
- ジョブはバッチ処理で実行します



バッチキューの設定方法

- OFPでのバッチ処理は，富士通のバッチシステムで管理
- 主要コマンド：
  - ジョブの投入: `pjsub <ジョブスクリプト名>`
  - 自分が投入したジョブの状況確認: `pjstat`
  - 投入ジョブの削除: `pjdel <ジョブID>`
  - 計算ノードの込み具合を見る: `pjstat --nodeuse`
  - バッチキューの状態を見る: `pjstat --rsc`
  - バッチキューの詳細構成を見る: `pjstat --rsc -x`
  - 投げられているジョブ数を見る: `pjstat --rsc -b`
  - 過去の投入履歴を見る: `pjstat -H`
  - 同時に投入できる数・実行できる数を見る: `pjstat --limit`

本お試し講習会でのキュー・グループ名

- 本講習会中のキュー名
  - tutorial-flat
  - 最大15分まで
  - 最大ノード数は16ノード（1088コア）まで
- 本講習会終了後のキュー名
  - lecture-flat
  - 利用条件tutorial-flatと同様
- グループ名: gt00

依存関係のあるジョブの投げ方（ステップジョブ，チェーンジョブ）

- ジョブスクリプト go0.sh の後に go1.sh，go2.sh，と投げたい
  - ステップジョブ（またはチェーンジョブ）という
- Oakforest-PACS におけるステップジョブの投げ方

1. `$ pjsub --step go0.sh`
   `[INFO] PJM 0000 pjsub Job 800967_0 submitted.`
2. 上記のジョブID（800967）を用いて，以下のように投入
   `$ pjsub --step --sparam jid=800967 go1.sh`
   `[INFO] PJM 0000 pjsub Job 800967_1 submitted`
3. 以降は同様
   `$ pjsub --step --sparam jid=800967 go2.sh`
   `[INFO] PJM 0000 pjsub Job 800967_2 submitted`

MPIプログラム実習1

- 完成しているプログラムを動かしてみる
- ほぼ完成しているプログラムにMPI関数を実装
  - MPI_Send()
  - MPI_Recv()
  - MPI_Reduce()

サンプルプログラム

- 0_hello
- 1_hybrid
- 2_sum_relay
- 3_sum_binary
- 4_sum_reduce
- 5_diffusion

ハイブリッド並列版プログラムをコンパイル

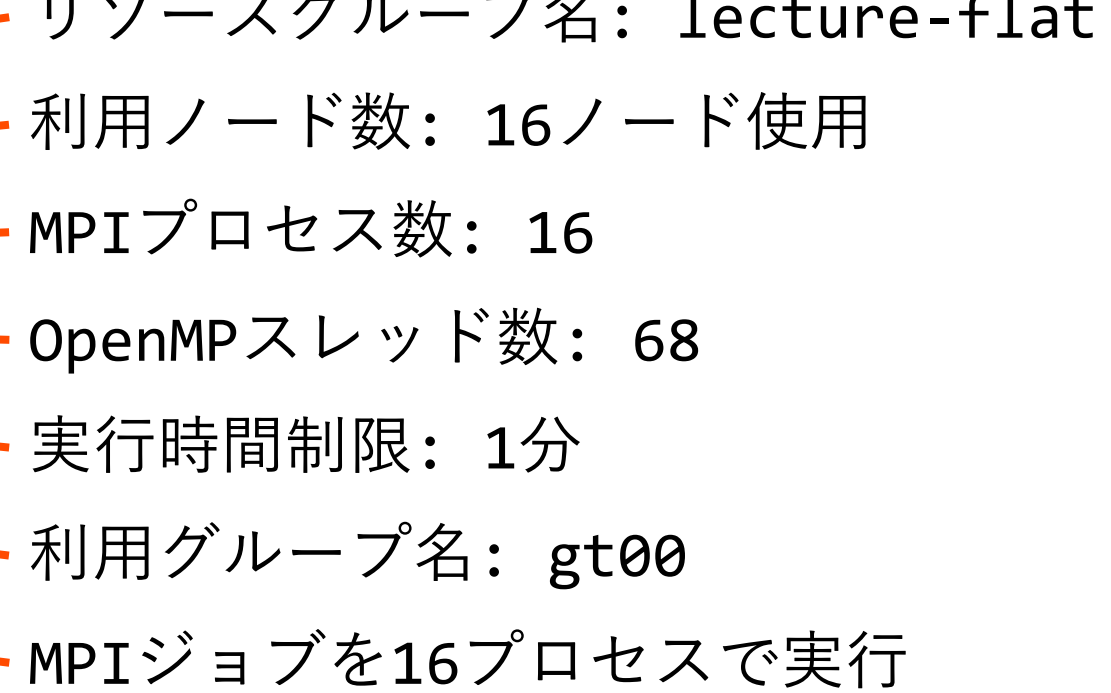
1. 1_hybrid ディレクトリに入る
   ```
   $ cd 0_hybrid
   ```
2. コンパイル
   ```
   $ make
   ```
3. 実行ファイル（hello_omp）ができていることを確認
   ```
   $ ls
   ```

ジョブスクリプトの説明（OpenMP/MPIハイブリッド版）

- 内容はC言語，Fortranで共通

```
#!/bin/bash
#PJM -L rscgrp=lecture-flat
#PJM -L node=16
#PJM --mpi proc=16
#PJM --omp thread=68
#PJM -L elapse=00:01:00
#PJM -g gt00

mpiexec.hydra
-n ${PJM_MPI_PROC} ./hello_omp
```



ハイブリッド並列版Helloプログラムの実行

- ジョブスクリプト名は run.sh です
- 配布したサンプルではキュー名が“lecture-flat”になっているので，これを“tutorial-flat”に変更してください

```
$ emacs -nw run.sh    # emacs で編集する場合
$ vim run.sh          # vim   で編集する場合
$ nano run.sh         # nano  で編集する場合
```

- ジョブを投入

```
$ pjsub run.sh
```

ハイブリッド並列版Helloプログラムの確認

1. 自分が投入したジョブの状態を確認
   ```
   $ pjstat
   ```
2. ジョブの実行が終了すると，以下のファイルが生成される
   ```
   run.sh.oXXXXXXX    # 標準出力ファイル
   run.sh.eXXXXXXX    # 標準エラー出力ファイル
   ```
   ジョブ名 + .[o e] + ジョブID というファイル名になっています
3. 標準出力ファイルの中身を見てみる
   ```
   $ cat run.sh.oXXXXXXX
   ```
   “Hello world!”が1088 (= 68スレッド * 16プロセス) 行あれば成功

サンプルプログラム

- 0_hello
- 1_hybrid
- 2_sum_relay
- 3_sum_binary
- 4_sum_reduce
- 5_diffusion

演習課題：MPI関数を用いて並列化してみる

- プログラムの一部を書いて，並列化を完了させてください
  - sum.[c f90] が演習用ファイル，ref_sum.[c f90] が実装例です
- 総和演算プログラム（逐次転送方式）
  - MPI_Send(), MPI_Recv() の使用（2_sum_relay）
- 総和演算プログラム（二分木通信方式）
  - MPI_Send(), MPI_Recv() の使用（3_sum_binary）
- 総和演算プログラム（MPI_Reduce使用）
  - MPI_Recv() の使用（4_sum_reduce）

演習ファイルと実装例の切り替え方法

- 演習ファイルをコンパイル
  Makefile冒頭の REF の行を
  コメントアウトし，make

```
# switch of excercise/reference (comment out
 for excercise)
# REF   := ref_

# environment
CC  := mpiicc

# option(s)
CFLAGS  := -O2

# source(s)
SRC := $(REF)sum.c
```

- 実装例をコンパイル
  Makefile冒頭の REF の行を
  有効にしたまま make

```
# switch of excercise/reference (comment out
 for excercise)
REF := ref_

# environment
CC  := mpiicc

# option(s)
CFLAGS  := -O2

# source(s)
SRC := $(REF)sum.c
```

総和演算プログラムの実行（2,3,4共通）

- ジョブスクリプト名は run.sh です
- 配布したサンプルではキュー名が“lecture-flat”になっているので，これを“tutorial-flat”に変更してください

```
$ emacs -nw run.sh    # emacs で編集する場合
$ vim run.sh          # vim   で編集する場合
$ nano run.sh         # nano  で編集する場合
```

- コンパイル

```
$ make
```

- ジョブを投入

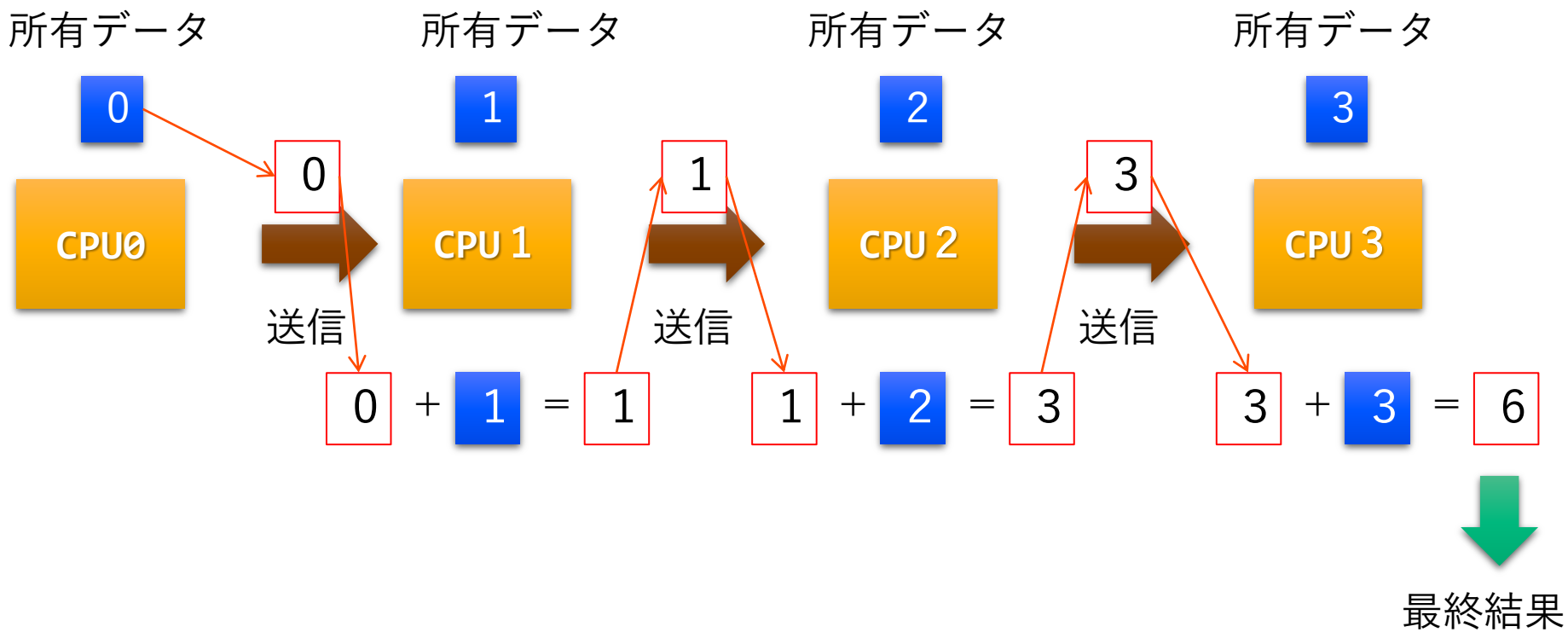
```
$ pjsub run.sh
```

総和演算プログラム（逐次転送方式）

- 各プロセスが所有するデータを，全プロセスで加算し，ある代表プロセス1つが結果を所有する演算を考える
- 素朴な方法（逐次転送方式）

1．（先頭プロセスでなければ）左隣のプロセスからデータを受信
2．【自分のデータ】と【受信データ】を加算
3．（末尾プロセスでなければ）右隣のプロセスに加算後データを送信
4．処理を終了

逐次転送方式（バケツリレー方式）による加算



総和演算プログラムの演習（C言語）

- write_program_recv; write_program_send; となっている部分を実装してください
  - 左隣（rank - 1）の人から値を受け取り，右隣（rank + 1）の人に送る
  - それぞれMPI_Recv(), MPI_Send() を用いればよいです

```
  /* receive partial sum */
  MPI_Status status;
  int recv = 0;
  /* receive partial sum from the left process (= rank -
 1) and put it to "recv" if the left process exists (i.e. rank - 1 >= 0) */
  write_program_recv;

  /* send sum */
  int send = rank + recv;
  /* send current sum "send" to the right process (= rank + 1) if the right process exist
s (i.e. rank + 1 <= size - 1) */
  write_program_send;
```

総和演算プログラムの演習（Fortran）

- write_program_recv, write_program_send となっている部分を実装してください
  - 左隣（rank - 1）の人から値を受け取り，右隣（rank + 1）の人に送る
  - それぞれMPI_Recv(), MPI_Send() を用いればよいです

```
!!$  receive partial sum
  recv = 0
!!$  receive partial sum from the left process (= rank -
 1) and put it to "recv" if the left process exists (i.e. rank - 1 >= 0)
 write_program_recv

!!$  send sum
  send = rank + recv
!!$  send current sum "send" to the right process (= rank + 1) if the right process exi
sts (i.e. rank + 1 <= size - 1)
  write_program_send
```

総和演算プログラムの実装例（C言語）

- 左端（rank = 0：始点）と右端（rank = size - 1：終点）のプロセスについては特別扱いが必要
- タグの値は任意（実装例では送信プロセスのランク）

```
/* receive partial sum */
MPI_Status status;
int recv = 0;
if(rank > 0)
  err = MPI_Recv(&recv, 1, MPI_INT, rank - 1, rank - 1, MPI_COMM_WORLD, &status);

/* send sum */
int send = rank + recv;
if(rank < size - 1)
  err = MPI_Send(&send, 1, MPI_INT, rank + 1, rank, MPI_COMM_WORLD);
```

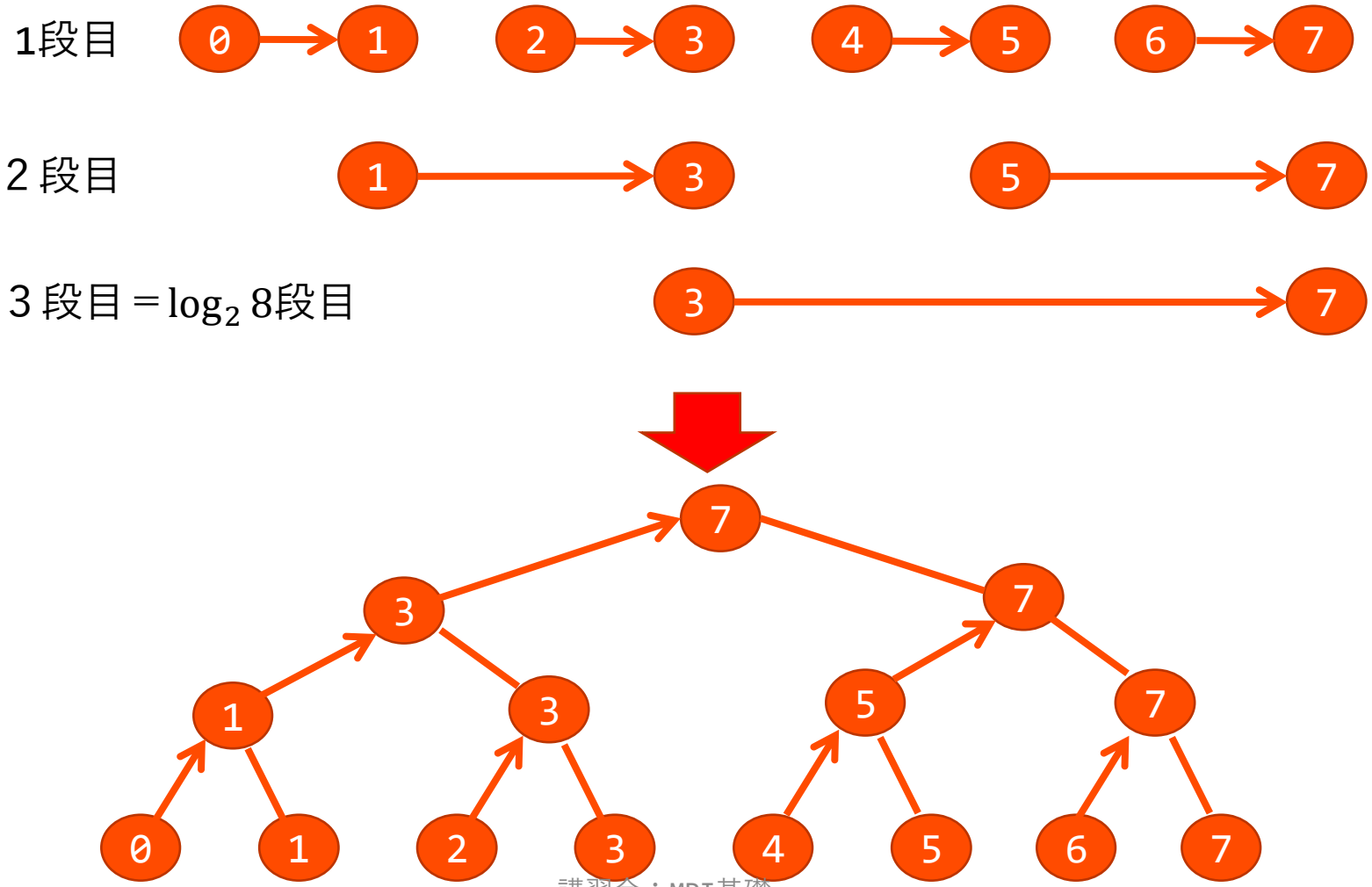
総和演算プログラムの実装例（Fortran）

- 左端（rank = 0：始点）と右端（rank = size - 1：終点）のプロセスについては特別扱いが必要
- タグの値は任意（実装例では送信プロセスのランク）

```
!!$  receive partial sum
  recv = 0
  if(rank > 0) then
     call MPI_Recv(recv, 1, MPI_INTEGER, rank - 1, rank - 1, MPI_COMM_WORLD, status, err)
  end if

!!$  send sum
  send = rank + recv
  if(rank < size - 1) then
     call MPI_Send(send, 1, MPI_INTEGER, rank + 1, rank, MPI_COMM_WORLD, err)
  end if
```

総和演算プログラム（二分木通信方式）



二分木通信方式実装上の工夫

- （以下の内容はサンプルプログラム3_sum_binary/では実装済み）
- プロセス番号の2進数表記の情報を利用
- 第 i 段において受信するプロセスの条件：
  rank & disp が disp と一致
  - ただし，disp = 2^(i - 1)
  - プロセス番号の2進数表記で，右からi番目のビットが立っているプロセス
  - データの送信元は rank - disp のプロセス
- 通信が成立するプロセス番号の間隔: disp = 2^(i - 1)
- 送信プロセスの条件についても同様に考えればよい
  - データ送信は1回のみ

総和演算プログラムの演習（C言語）

- write_program_recv; write_program_send; となっている部分を実装してください
- rank - disp のプロセスから値を受信, rank + disp へと送信
- MPI_Recv(), MPI_Send() を使って実装できます

```
/* summation based on binary tree manner */
MPI_Status status;
int recv = 0;
int send = rank;
int disp = 1;
for(int ii = 0; ii < log2p; ii++){
  if((rank & disp) == disp){
    /* receive partial sum from the pair process (= rank -
 disp) and put it to "recv" */
    write_program_recv;
    send += recv;
    disp <<= 1;
  }
  else{
    /* send current sum "send" to the target process (= rank +
 disp) */
    write_program_send;
    break;
  }
}
```

総和演算プログラムの演習（Fortran）

- write_program_recv, write_program_send となっている部分を実装してください
- rank - disp のプロセスから値を受信, rank + disp へと送信
- MPI_Recv(), MPI_Send() を使って実装できます

```
!!$  summation based on binary tree manner
  recv = 0
  send = rank
  disp = 1
  do ii = 0, log2p - 1
     if(iand(rank, disp) == disp) then
!!$        receive partial sum from the pair process (= rank
- disp) and put it to "recv"
        write_program_recv
        send = send + recv
        disp = disp * 2
     else
!!$        send current sum "send" to the target process (= r
ank + disp)
        write_program_send
        exit
     end if
  end do
```

総和演算プログラムの実装例（C言語）

- タグについては，ステージごとに異なる値になるように工夫

```
/* summation based on binary tree manner */
MPI_Status status;
int recv = 0;
int send = rank;
int disp = 1;
for(int ii = 0; ii < log2p; ii++){
  if((rank & disp) == disp){
    err = MPI_Recv(&recv, 1, MPI_INT, rank - disp, rank -
 disp + ii * size, MPI_COMM_WORLD, &status);
    send += recv;
    disp <<= 1;
  }
  else{
    err = MPI_Send(&send, 1, MPI_INT, rank + disp, rank + ii * size, MPI_COMM_WORLD);
    break;
  }}
```

総和演算プログラムの実装例（Fortran）

- タグについてはステージごとに異なる値になるように工夫

```
!!$  summation based on binary tree manner
  recv = 0
  send = rank
  disp = 1
  do ii = 0, log2p - 1
     if(iand(rank, disp) == disp) then
        call MPI_Recv(recv, 1, MPI_INTEGER, rank - disp, rank -
 disp + ii * size, MPI_COMM_WORLD, status, err)
        send = send + recv
        disp = disp * 2
     else
        call MPI_Send(send, 1, MPI_INTEGER, rank + disp, rank + ii * size, MPI_COMM_WORLD, err)
        exit
     end if
  end do
```

総和演算プログラムのアルゴリズム比較

- 逐次転送方式：$N_{\mathrm{p}}-1$ 回だけ通信する
- 二分木通信方式：
  - 仮定：各段での通信は完全に並列実行される（通信の衝突は発生しない）
  - 段数 = $\log_2(N_{\mathrm{p}})$ 回が通信回数となる
- 通信回数の比較
  - プロセス数が増えると，通信回数の差（～実行時間の差）が増大
  - $N_p = 1024\ (= 2^{10})$の場合には，1023回 対 10回
  - ただし，必ず二分木通信方式が良いという保証はない
    （通信衝突が多発する可能性）

総和演算プログラム（MPI_Reduce使用）

- MPI_SUM を指定すれば総和を計算できるので，MPI_Reduce() を用いて実装するのが一番簡単
  - ライブラリ側で最適なアルゴリズムを選択するため，こちらの方が速いと期待される

- 今まで自分で実装していた部分を，MPI_Reduce() を用いて実装してみてください
  - ファイルは 4_sum_reduce/ にあります

総和演算プログラムの演習（C言語）

- write_program; となっている部分を実装してください
- MPI_Reduce() を使ってください
- rank = 0 のプロセスが結果を保持するようにしてください
- send の総和を recv に格納してください

```
 /* calculate the total sum by using MPI_Reduce */
 int send = rank;
 int recv = 0;
 /* calculate total sum of "send" and put the answer to "recv", only the root process
(rank = 0) receives the result */
 write_program;
```

総和演算プログラムの演習（Fortran）

- write_program となっている部分を実装してください
- MPI_Reduce() を使ってください
- rank = 0 のプロセスが結果を保持するようにしてください
- send の総和を recv に格納してください

```
!!$  calculate the total sum by using MPI_Reduce
  send = rank
  recv = 0
!!$  calculate total sum of "send" and put the answer to "recv", only the root proce
ss (rank = 0) receives the result
  write_program
```

総和演算プログラムの実装例（C言語）

- データ型が int なので，MPI_INT を指定
- 総和を求めるので，MPI_SUM を指定
- ランク0に結果を返すので，0を指定

```
/* calculate the total sum by using MPI_Reduce */
int send = rank;
int recv = 0;
err = MPI_Reduce(&send, &recv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

総和演算プログラムの実装例（Fortran）

- データ型が integer なので，MPI_INTEGER を指定
- 総和を求めるので，MPI_SUM を指定
- ランク0に結果を返すので，0を指定

```
!!$  calculate the total sum by using MPI_Reduce
  send = rank
  recv = 0
  call MPI_Reduce(send, recv, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD, err)
```

おまけ：実行時間の測定方法（C言語）

1. 測定開始前に，全プロセスの同期をとる（MPI_Barrier）
2. 現在時刻を取得（MPI_Wtime以外の関数でも良い）し，処理開始
3. 測定対象の処理が終わったタイミングで，時刻を再取得
4. 経過時間の最大値（= 全体の実行時間）をMPI_Reduceで取得

```
  err = MPI_Barrier(MPI_COMM_WORLD);
  double t_ini = MPI_Wtime();

  測定対象の処理を実行

  double t_fin = MPI_Wtime();
  double elapsed = t_fin - t_ini;
  err = MPI_Reduce((rank > 0) ? &elapsed : MPI_IN_PLACE,
&elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

おまけ：実行時間の測定方法（Fortran）

- やっていることはC言語版と同じだが，3項演算子を使わない実装

```
  call MPI_Barrier(MPI_COMM_WORLD, err)
  t_ini = MPI_Wtime()

  測定対象の処理を実行

  t_fin = MPI_Wtime()
  elapsed = t_fin - t_ini
  if(rank /= 0) then
     call MPI_Reduce(     elapsed, elapsed, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0, MPI_
COMM_WORLD, err)
  else
     call MPI_Reduce(MPI_IN_PLACE, elapsed, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0, MPI_
COMM_WORLD, err)
  end if
```

総和演算プログラムのまとめ

- 計算結果が正しいことを確認してください
  - 計算結果と正解が標準出力（run.sh.oXXXXXXX）に書かれています
- 3通りの方法で実装した総和演算の実行時間を比較してください
  - 総和演算部分の実行時間は標準出力に書かれています
  - お渡ししたスクリプトでは，プログラムを5回連続実行します
  - 一番速かった場合を代表値とする場合，中央値を代表値とする場合などがあります（どういうデータを取りたいかに応じて適切に選択）

MPI実行時のリダイレクト

- Oakforest-PACSスーパーコンピュータシステムでは, MPI実行時の入出力のリダイレクトができます
- 例： `mpiexec.hydra ./a.out < in.txt > out.txt`

- 次節 `5_diffusion` の `gen.sh`, `run.sh` では入力をリダイレクトしています

性能プロファイラ

- Oakforest-PACS
  - Intel Vtune Amplifier
  - PAPI (Performance API)
  - Oakforest-PACS PAライブラリ
- 詳細はWebポータルから「ドキュメント閲覧」➔
  Oakforest-PACS システム利用手引書
  7.1. パフォーマンス分析ツール
  または
  Oakforest-PACS PAライブラリ利用ガイド
  を参照してください

MPIプログラム実習2,3

- 2次元拡散方程式
  - MPI_Bcast()
  - MPI_Send()
  - MPI_Recv()
  - MPI_Scatter()
  - MPI_Gather()

サンプルプログラム

- 0_hello
- 1_hybrid
- 2_sum_relay
- 3_sum_binary
- 4_sum_reduce
- 5_diffusion

2 次元拡散方程式

- 支配方程式（物理量u，$a > 0$は拡散係数）：

$$\frac{\partial u}{\partial t} = a\nabla^2 u$$

- 一番シンプルな差分化を適用（簡単のため$h = \Delta x = \Delta y$とする）：

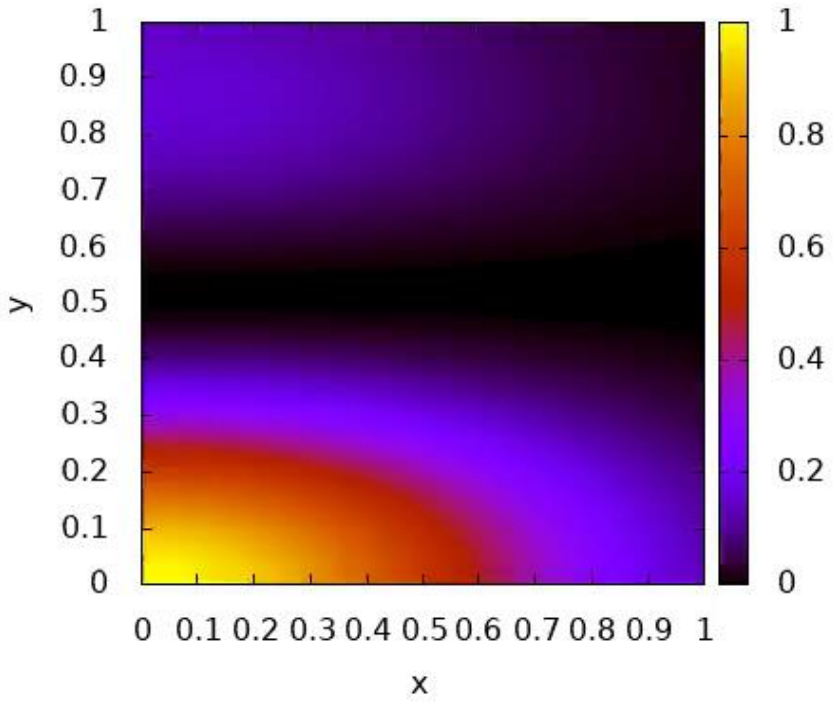
$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{a\,\Delta t}{h^2}\left(u_{i,j-1}^{n} + u_{i,j+1}^{n} + u_{i-1,j}^{n} + u_{i+1,j}^{n} - 4u_{i,j}^{n}\right)$$

- 安定性条件：

$$\nu \equiv \frac{a\,\Delta t}{(\Delta x)^2} + \frac{a\,\Delta t}{(\Delta y)^2} \leq \frac{1}{2}$$

ムービー作成例

- `$ ffmpeg -r 30 -i fig/map%03d.png -vf scale="trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -profile:v high -x264-params slower -pix_fmt yuv420p -g 30 map.mov`
  - この例ではフレームレートを30 fps とした
  - ここまでオプションを渡さなくてもムービーは作れるがTeXを使ってPDFに埋め込むならば上記の例を推奨
- 右の例は空間分解能，時間分解能ともに初期設定から変更しているので，完全に同じムービーを作るにはパラメータを再設定する必要あり
  - 設定ファイルは global.cfg
  - 1024*1024メッシュ（プロセス数も変更する）
  - スナップショット間隔は0.00390625



演習課題

- 2次元拡散方程式のシミュレーションコードを並列化してください
  - 元になるプログラムは `5_diffusion/src` の中に入っています
  - ソースコードは機能ごとに分割されています
- いきなり全体を並列化というのは非常に大変なので…
  - 同じフォルダの `ref_` から始まるファイルは並列化済みサンプルです
  - `Makefile` は `ref_` つきファイルを使ってコンパイルするようになっているので，編集中のファイルに対応する場所だけ `Makefile` を書き換えれば，少しずつ並列化していくことが可能です
  - （`ref_` つきファイルは並列化したサンプルコードなので，並列化方法が分からないときにはヒントとして使ってください）

サンプルプログラムの動かし方

- `$ cd 5_diffusion`
- `$ make dir` `# 初めに一度だけ実行`
- `$ make` `# bin/diffusion, bin/gen_ic を生成`
- `$ pjsub gen.sh` `# 初期条件(dat/snp000.dat)の生成`
- `$ pjsub run.sh` `# dat/snp008.dat までが出力される`
- `$ pjsub plt.sh` `# 計算結果を可視化し，fig/ 以下に出力`

設定ファイル（global.cfg）の書式

- # 以降はコメントなので，実際のファイルには書かない

```
128 136     # x，y-方向のメッシュ数（N_x, N_y）
32 34       # x，y-方向のMPIプロセス数（p_x, p_y）
0.0625      # 拡散係数（a > 0）の値
0.25        # クーラン数（ν ≤ 0.5）の値
0.5 0.0625 # シミュレーション終了時刻とスナップショット出力間隔
0           # 初期条件とするスナップショットのファイル番号
```

  - N_xはp_xの，N_yはp_yの倍数とする（領域分割の設定を簡単にするため）
  - p_xとp_yの積は全プロセス数と一致させる（この場合には1088）

参考：画像の表示方法

- Oakforest-PACS にログインする際に -Y つきでログイン
  ```
  $ ssh -Y username@ofp.jcahpc.jp
  $ cd /work/gt00/username/…/fig
  $ eog map000.png &
  ```
  注：Cygwin からはこの方法では"cannot open display"と言われて画像が表示できません
- 手元のPCに画像ファイルをコピーして表示
  ```
  $ rsync -av username@ofp.jcahpc.jp:/work/.../fig .
  ```
  Windowsの方は，WinSCPを使ってダウンロードしても良いです
  - rsync以外にはsftp，scpなど（scpはOpenSSH的に非推奨とのこと）

Makefile の編集例

- 元々の Makefile

```
# source(s)
SRC_EXE := $(REF)diffusion.c
SRC_EXE += $(REF)topology.c
SRC_EXE += $(REF)boundary.c
SRC_EXE += $(REF)scatter.c
SRC_EXE += $(REF)gather.c
SRC_EXE += io.c
```

  - はじめはほとんどのファイルの前に $(REF) が入っている

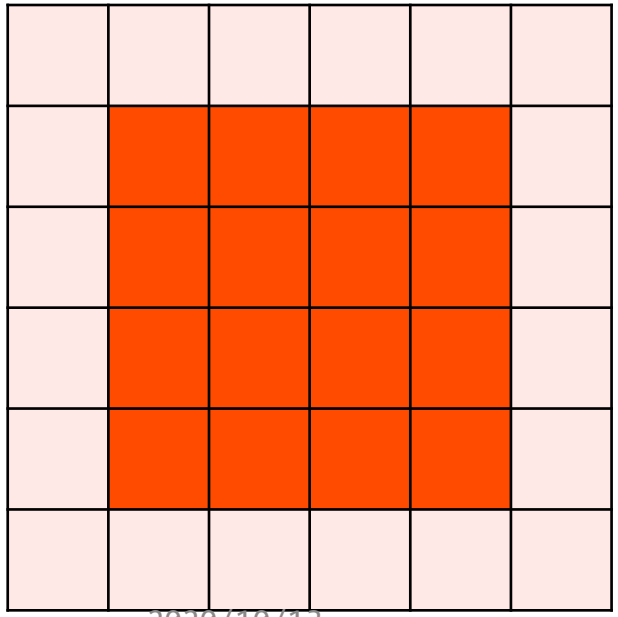
- topology.c を編集する場合

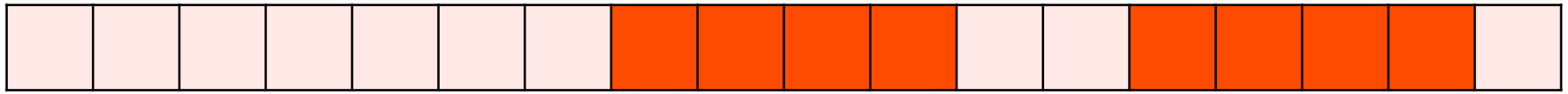
```
# source(s)
SRC_EXE := $(REF)diffusion.c
SRC_EXE += topology.c
SRC_EXE += $(REF)boundary.c
SRC_EXE += $(REF)scatter.c
SRC_EXE += $(REF)gather.c
SRC_EXE += io.c
```

  - topology.c の前にあった $(REF) を削除する
  - 全ての $(REF) が取れれば完了

2次元配列データの設定

- 担当領域のデータと境界条件のデータを1つの配列に格納
  - 境界条件に対応する（他プロセスが担当する領域の）データを別配列に格納すると，時間発展を計算する関数の実装が面倒になるため
  - 今回はNSLEEVE（= １）列分のデータを保持する領域を上下左右に追加





実装にあたって

- NSLEEVEの値は1として設定済みです
  - C言語では src/macro.h で define しています
  - Fortran では src/macro.f90 で宣言しています
- C言語で多次元配列を動的に確保（したように見せかける）のは多少面倒なので，1次元配列を多次元配列のように見なしてアクセス
  - src/macro.h で INDEX2D(nx, ny, i, j) というマクロを定義
    以下の2パターンの実装が等価

```
static int array2d[nx][ny];
for(int ii = 0; ii < nx; ii++)
    for(int jj = 0; jj < ny; jj++)
        array2d[ii][jj] = ii * jj;
```

```
static int array1d[nx * ny];
for(int ii = 0; ii < nx; ii++)
    for(int jj = 0; jj < ny; jj++)
        array1d[INDEX2D(nx, ny, ii, jj)] = ii * jj;
```

状況設定の共有

- シミュレーションの設定については，rootのみが読み込む実装
- 全プロセスが知っておくべき情報はMPI通信を用いて共有する
  - 全メッシュ数，領域分割の設定，拡散係数，クーラン数など
- やること：
  1. MPI_Bcast() を用いてrootから全プロセスに対してデータを放送

状況設定の共有（C言語）

- diffusion.c 中の write_program; とある部分を編集
  - 共有すべき変数:
    nx_tot, ny_tot, px, py, diff_coeff, courant, final, snapshot_interval, prev

```
  /* broadcast configuration of the simulation to all processes */
  /* broadcast nx_tot, ny_tot, px, py, diff_coeff, courant, final,
snapshot_interval, and prev from the root process (rank = 0) to all
 processes */
  /* int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm) */
  write_program;
```

状況設定の共有（Fortran）

- diffusion.f90 中の write_program とある部分を編集
  - 共有すべき変数:
    nx_tot, ny_tot, px, py, diff_coeff, courant, fin, snapshot_interval, prev

```
!!$  broadcast configuration of the simulation to all processes
!!$  broadcast nx_tot, ny_tot, px, py, diff_coeff, courant, fin,
snapshot_interval, and prev from the root process (rank = 0) to a
ll processes
!!$  MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
!!$            <type>    BUFFER(*)
!!$            INTEGER   COUNT, DATATYPE, ROOT, COMM, IERROR
  write_program
```

通信相手の登録

- 本サンプルコードでは，計算領域を2次元的に分割する
- 上下左右で接するプロセスとの通信が発生するため，対応するプロセスランクをあらかじめ覚えておく
- 周期的境界条件を採用していることに注意
  - 例：右端のプロセスのペアは左端のプロセス
- やること：
  1．自分のプロセスランクのx方向，y方向におけるIDを計算
  2．自分と接するプロセスランクを取得

通信相手の登録（C言語）

- topology.c 中の set_process_topology() を編集
  - 注： 単にrx-1と実装すると，rx=0の場合に意図しない挙動になる

```
void set_process_topology(const int rank, const int px, const int py,
 int *rank_l, int *rank_r, int *rank_b, int *rank_t)
{
  write_program;
  const int rx = ;/**< rx ¥in [0, px) */
  const int ry = ;/**< ry ¥in [0, py) */
  *rank_l = ;/**< (rx - 1, ry), rx - 1 ¥in [0, px) */
  *rank_r = ;/**< (rx + 1, ry), rx + 1 ¥in [0, px) */
  *rank_b = ;/**< (rx, ry - 1), ry - 1 ¥in [0, py) */
  *rank_t = ;/**< (rx, ry + 1), ry + 1 ¥in [0, py) */
}
```

通信相手の登録（Fortran）

- topology.f90 中の set_process_topology() を編集
  - 注： 単にrx-1と実装すると，rx=0の場合に意図しない挙動になる

```
subroutine set_process_topology(rank, px, py, rank_l, rank_r, rank_b, rank_t)
    implicit none

    write_program
    rx = !!< rx ¥in [0, px)
    ry = !!< ry ¥in [0, py)
    rank_l = !!< (rx - 1, ry), rx - 1 ¥in [0, px)
    rank_r = !!< (rx + 1, ry), rx + 1 ¥in [0, px)
    rank_b = !!< (rx, ry - 1), ry - 1 ¥in [0, py)
    rank_t = !!< (rx, ry + 1), ry + 1 ¥in [0, py)
  end subroutine set_process_topology
```

境界条件の設定（袖領域の交換）

- 上下左右の領域を担当するプロセスに対して，データを送受信
- やること：
  1. 送信データを準備（メモリ空間上で連続になるように置きなおす）
  2. MPI関数を用いてデータを送受信（MPI_Send/MPI_Recvの組み合わせ，MPI_Sendrecvの使用，MPI_Isend/MPI_Irecvの使用など）
  3. 受信した（連続）データを時間発展計算用の配列に置きなおす

境界条件の設定（C言語）

- boundary.c 中の set_periodic_boundaries() を編集

```
void set_periodic_boundaries(const int nx, const int ny, float *dat, float *buf,
           const int rank, const int rank_l, const int rank_r, const int rank_b,
 const int rank_t, const int py)
{
  /* assign send/receive buffers */
  write_program;
  /* prepare send buffer */
   write_program;
  /* exchange sleeve regions */
  write_program;
  /* copy from receive buffer */
  write_program;
}
```

境界条件の設定（Fortran）

- boundary.f90 中のset_periodic_boundaries()を編集

```
  subroutine set_periodic_boundaries(nx, ny, dat, buf, rank, rank_l,
rank_r, rank_b, rank_t, px)

!!$    assign send/receive buffers
    write_program
!!$    prepare send buffer
    write_program
!!$    exchange sleeve regions
    write_program
!!$    copy from receive buffer
    write_program

  end subroutine set_periodic_boundaries
```

計算データの配布

- 初期条件を root が代表して読み取る実装
- 各プロセスが計算を進めるためには，自分が担当する領域のデータを取得する必要がある
- 2次元領域のデータなので，メモリ上でのデータの並び方にも留意
- やること：
  1. root プロセスが MPI_Scatter() 用にデータを並べなおす
     （rank 0用のデータ，rank 1用のデータ，…，rank n - 1用のデータ）
  2. MPI_Scatter() を用いてデータを配布
  3. 受け取ったデータを，計算用の配列に格納

計算データの配布（C言語）

- scatter.c 中の scatter_map() を編集

```
void scatter_map(const int nx_tot, const int ny_tot, float *map_ful, float *buf_ful,
     const int nx, const int ny, float *map_loc, float *buf_loc,
     const int py, const int rank, const int size)
{
  /* prepare send buffer (only root process) */
  write_program;

  /* scatter the data */
  write_program;

  /* copy from receive buffer */
  write_program;
}
```

計算データの配布（Fortran）

- scatter.f90 中の scatter_map() を編集

```
  subroutine scatter_map(nx_tot, ny_tot, map_ful, buf_ful, nx, ny,
 map_loc, buf_loc, px, rank, size)
    implicit none

!!$    prepare send buffer
    write_program

!!$    scatter the data
    write_program

!!$    copy from receive buffer
    write_program

  end subroutine scatter_map
```

計算データの収集

- スナップショットは root が代表して出力する実装
- 全プロセスが計算したデータをrootプロセスに集める必要がある
- 2次元領域のデータなので，メモリ上でのデータの並び方にも留意
- やること：
  1. MPI_Gather() 用にデータを並べなおす
  2. MPI_Gather() を用いてデータを root に集める
  3. rootプロセスが受け取ったデータを，並べなおす
     （受信データは rank 0用のデータ，rank 1用のデータ，…，
     rank n - 1用のデータ という風に並んでいる）

計算データの収集（C言語）

- gather.c 中の gather_map() を編集

```
void gather_map(const int nx_tot, const int ny_tot, float *map_ful, float *buf_ful,
    const int nx, const int ny, float *map_loc, float *buf_loc,
    const int py, const int rank, const int size)
{
  /* prepare send buffer */
  write_program;

  /* gather the data */
  write_program;

  /* copy from receive buffer */
  write_program;
}
```

計算データの収集（Fortran）

- gather.f90 中の gather_map() を編集

```
  subroutine gather_map(nx_tot, ny_tot, map_ful, buf_ful, nx, ny,
 map_loc, buf_loc, px, rank, size)
    implicit none

!!$    prepare send buffer
    write_program

!!$    gather the data
    write_program

!!$    copy from receive buffer
    write_program

  end subroutine gather_map
```

実装例の解説

- ここから先は実装例（ref_つきのファイル）の解説です
- 演習が終わった，あるいは行き詰ってしまってどうしようもないという場合にご参照ください

状況設定の共有（C言語）

- ref_diffusion.c の中身
- MPI_Bcastではバッファの先頭アドレスを指定するので，&をつける
- intデータについてはMPI_INT，floatデータについてはMPI_FLOAT
- 変数の値を読み込んだのは rank = 0 のプロセス

```
/* broadcast configuration of the simulation to all processes */
err = MPI_Bcast(&nx_tot, 1, MPI_INT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&ny_tot, 1, MPI_INT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&px, 1, MPI_INT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&py, 1, MPI_INT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&diff_coeff, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&courant, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&final, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&snapshot_interval, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
err = MPI_Bcast(&prev, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

状況設定の共有（Fortran）

- ref_diffusion.f90 の中身
- integerデータについてはMPI_INTETER,
  realデータについてはMPI_REAL
- 変数の値を読み込んだのは rank = 0 のプロセス

```
!!$  broadcast configuration of the simulation to all processes
  call MPI_Bcast(nx_tot, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(ny_tot, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(px, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(py, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(diff_coeff, 1, MPI_REAL, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(courant, 1, MPI_REAL, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(fin, 1, MPI_REAL, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(snapshot_interval, 1, MPI_REAL, 0, MPI_COMM_WORLD, err)
  call MPI_Bcast(prev, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
```

通信相手の登録（C言語）

- ref_topology.c の中身
- MPIプロセスを2次元的に配置した時の上下左右のプロセスを計算
- 左側，下側については単に -1 するのではなく，px - 1 を加えた後にpxで割った余りを求めることで，負の値が混入することを回避

```
void set_process_topology(const int rank, const int px, const int py, int *rank_l, in
t *rank_r, int *rank_b, int *rank_t)
{
  const int rx = rank / py;
  const int ry = rank % py;
  *rank_l = INDEX2D(px, py, (rx + px - 1) % px, ry);
  *rank_r = INDEX2D(px, py, (rx      + 1) % px, ry);
  *rank_b = INDEX2D(px, py, rx, (ry + py - 1) % py);
  *rank_t = INDEX2D(px, py, rx, (ry      + 1) % py);
}
```

通信相手の登録（Fortran）

- ref_topology.f90 の中身
- MPIプロセスを2次元的に配置した時の上下左右のプロセスを計算
- 左側，下側については単に -1 するのではなく，px - 1 を加えた後にpxで割った余りを求めることで，負の値が混入することを回避

```
subroutine set_process_topology(rank, px, py, rank_l, rank_r, rank_b, rank_t)
  implicit none
  integer, intent(in) :: rank, px, py
  integer, intent(out) :: rank_l, rank_r, rank_b, rank_t
  integer :: rx, ry
  rx = mod(rank, px)
  ry = rank / px
  rank_l = mod(rx + px - 1, px) + px * ry
  rank_r = mod(rx      + 1, px) + px * ry
  rank_b = rx + px * mod(ry + py - 1, py)
  rank_t = rx + px * mod(ry      + 1, py)
end subroutine set_process_topology
```

境界条件の設定（袖領域の交換：C:1/5）

- ref_boundary.c の中身
- 送受信バッファの領域を（衝突しないように）設定

```
void set_periodic_boundaries(const int nx, const int ny, float *dat, float *buf,
           const int rank, const int rank_l, const int rank_r, const int rank_b,
 const int rank_t, const int py)
{
  /* assign send/receive buffers */
  const int send_l = 0;
  const int send_r = send_l + NSLEEVE * ny;
  const int send_b = send_r + NSLEEVE * ny;
  const int send_t = send_b + NSLEEVE * nx;
  const int recv_l = send_t + NSLEEVE * nx;
  const int recv_r = recv_l + NSLEEVE * ny;
  const int recv_b = recv_r + NSLEEVE * ny;
  const int recv_t = recv_b + NSLEEVE * nx;
```

次ページへ続く

境界条件の設定（袖領域の交換：C:2/5）

- 送信バッファ上にデータが連続に並ぶようにデータをコピー
- 派生データ型をうまく使うと，この部分は省略できる

前ページからの続き

```
  /* prepare send buffer */
  for(int ii = 0; ii < NSLEEVE; ii++)
    for(int jj = 0; jj < ny; jj++){
      buf[send_l + INDEX2D(NSLEEVE, ny, ii, jj)] = dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEV
E + ii, NSLEEVE + jj)];
      buf[send_r + INDEX2D(NSLEEVE, ny, ii, jj)] = dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE,      n
x + ii, NSLEEVE + jj)];}

  for(int ii = 0; ii < nx; ii++)
    for(int jj = 0; jj < NSLEEVE; jj++){
      buf[send_b + INDEX2D(nx, NSLEEVE, ii, jj)] = dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEV
E + ii, NSLEEVE + jj)];
      buf[send_t + INDEX2D(nx, NSLEEVE, ii, jj)] = dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEV
E + ii,      ny + jj)];}
```

次ページへ続く

境界条件の設定（袖領域の交換：C:3/5）

- 水平方向の両隣との通信
- デッドロックにならないように送受信の順番を工夫
- MPI_Sendrecv()や非同期通信を使っても良い

前ページからの続き

```
  /* exchange sleeve regions */
  int err;
  MPI_Status status;
  /* horizontal exchanging */
  if((rank / py) & 1){
    MPI_Recv(&buf[recv_r], NSLEEVE * ny, MPI_FLOAT, rank_r, rank_r, MPI_COMM_WORLD, &
status);
    MPI_Send(&buf[send_l], NSLEEVE * ny, MPI_FLOAT, rank_l, rank  , MPI_COMM_WORLD);
    MPI_Recv(&buf[recv_l], NSLEEVE * ny, MPI_FLOAT, rank_l, rank_l, MPI_COMM_WORLD, &
status);
    MPI_Send(&buf[send_r], NSLEEVE * ny, MPI_FLOAT, rank_r, rank  , MPI_COMM_WORLD);
  }
  else{
    MPI_Send(&buf[send_l], NSLEEVE * ny, MPI_FLOAT, rank_l, rank  , MPI_COMM_WORLD);
    MPI_Recv(&buf[recv_r], NSLEEVE * ny, MPI_FLOAT, rank_r, rank_r, MPI_COMM_WORLD, &
status);
    MPI_Send(&buf[send_r], NSLEEVE * ny, MPI_FLOAT, rank_r, rank  , MPI_COMM_WORLD);
    MPI_Recv(&buf[recv_l], NSLEEVE * ny, MPI_FLOAT, rank_l, rank_l, MPI_COMM_WORLD, &
status);
  }
```

次ページへ続く

境界条件の設定（袖領域の交換：C:4/5）

- 上下方向のプロセスと通信
- 実装については水平方向と同じ

```
前ページからの続き
  /* vertical exchanging */
  if((rank % py) & 1){
    MPI_Recv(&buf[recv_t], NSLEEVE * nx, MPI_FLOAT, rank_t, rank_t, MPI_COMM_WORLD, &
status);
    MPI_Send(&buf[send_b], NSLEEVE * nx, MPI_FLOAT, rank_b, rank  , MPI_COMM_WORLD);
    MPI_Recv(&buf[recv_b], NSLEEVE * nx, MPI_FLOAT, rank_b, rank_b, MPI_COMM_WORLD, &
status);
    MPI_Send(&buf[send_t], NSLEEVE * nx, MPI_FLOAT, rank_t, rank  , MPI_COMM_WORLD);
  }
  else{
    MPI_Send(&buf[send_b], NSLEEVE * nx, MPI_FLOAT, rank_b, rank  , MPI_COMM_WORLD);
    MPI_Recv(&buf[recv_t], NSLEEVE * nx, MPI_FLOAT, rank_t, rank_t, MPI_COMM_WORLD, &
status);
    MPI_Send(&buf[send_t], NSLEEVE * nx, MPI_FLOAT, rank_t, rank  , MPI_COMM_WORLD);
    MPI_Recv(&buf[recv_b], NSLEEVE * nx, MPI_FLOAT, rank_b, rank_b, MPI_COMM_WORLD, &
status);
  }
次ページへ続く
```

境界条件の設定（袖領域の交換：C:5/5 ）

- 受信したデータを，計算データを格納する配列にコピー
  - 受信データは連続
  - 計算配列の境界条件部分に格納
- 派生データ型をうまく使うと，この部分を省略できる

前ページからの続き

```
  /* copy from receive buffer */
  for(int ii = 0; ii < NSLEEVE; ii++)
    for(int jj = 0; jj < ny; jj++){
      dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE,
  ii, NSLEEVE + jj)]
      = buf[recv_l + INDEX2D(NSLEEVE, ny, ii, jj)];
      dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEVE + nx + ii,
NSLEEVE + jj)]
      = buf[recv_r + INDEX2D(NSLEEVE, ny, ii, jj)];
    }
  for(int ii = 0; ii < nx; ii++)
    for(int jj = 0; jj < NSLEEVE; jj++){
      dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEVE + ii,
                  jj)]
      = buf[recv_b + INDEX2D(nx, NSLEEVE, ii, jj)];
      dat[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEVE + ii, NSLEE
VE + ny + jj)]
      = buf[recv_t + INDEX2D(nx, NSLEEVE, ii, jj)];
    }
}
```

境界条件の設定（袖領域の交換：F:1/6）

- ref_boundary.f90 の中身
- 変数の宣言部分
- 計算データを格納する配列 dat は2次元配列
- 送受信用のバッファとして使用する buf は1次元配列として使用

```
  subroutine set_periodic_boundaries(nx, ny, dat, buf, rank, rank_l, rank_r, r
ank_b, rank_t, px)
    implicit none

    integer, intent(in) :: nx, ny
    real, intent(inout) :: dat(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE)
    real, intent(out) :: buf((nx + 2 * NSLEEVE) * (ny + 2 * NSLEEVE))
    integer, intent(in) :: rank, rank_l, rank_r, rank_b, rank_t, px

    integer :: send_l, send_r, send_b, send_t
    integer :: recv_l, recv_r, recv_b, recv_t

    integer :: ii, jj

    integer :: err
    integer :: status(MPI_STATUS_SIZE)
次ページへ続く
```

境界条件の設定（袖領域の交換：F:2/6）

- 送受信バッファの領域を（衝突しないように）設定

```
前ページからの続き
!!$    assign send/receive buffers
    send_l = 1
    send_r = send_l + NSLEEVE * ny
    send_b = send_r + NSLEEVE * ny
    send_t = send_b + NSLEEVE * nx
    recv_l = send_t + NSLEEVE * nx
    recv_r = recv_l + NSLEEVE * ny
    recv_b = recv_r + NSLEEVE * ny
    recv_t = recv_b + NSLEEVE * nx
次ページへ続く
```

境界条件の設定（袖領域の交換：F:3/6）

- 送信バッファ上にデータが連続に並ぶようにデータをコピー
- 派生データ型をうまく使うと，この部分は省略できる

前ページからの続き

```
!!$    prepare send buffer
    do jj = 0, ny - 1
       do ii = 0, NSLEEVE - 1
          buf(send_l + ii + NSLEEVE * jj) = dat(1 + NSLEEVE + ii, 1 + NSLEEVE + jj)
          buf(send_r + ii + NSLEEVE * jj) = dat(1 +      nx + ii, 1 + NSLEEVE + jj)
       end do
    end do
    do jj = 0, NSLEEVE - 1
       do ii = 0, nx - 1
          buf(send_b + ii + nx * jj) = dat(1 + NSLEEVE + ii, 1 + NSLEEVE + jj)
          buf(send_t + ii + nx * jj) = dat(1 + NSLEEVE + ii, 1 +      ny + jj)
       end do
    end do
```

次ページへ続く

境界条件の設定（袖領域の交換：F:4/6）

- 水平方向の両隣との通信
- デッドロックにならないよう送受信の順番を工夫
- MPI_Sendrecv()や非同期通信でも良い

前ページからの続き

```
    if(iand(mod(rank, px), 1) == 1) then
       call MPI_Recv(buf(recv_r), NSLEEVE * ny, MPI_REAL, rank_r, rank_r, MPI_COMM_WORLD, status, err)
       call MPI_Send(buf(send_l), NSLEEVE * ny, MPI_REAL, rank_l, rank  , MPI_COMM_WORLD, err)
       call MPI_Recv(buf(recv_l), NSLEEVE * ny, MPI_REAL, rank_l, rank_l, MPI_COMM_WORLD, status, err)
       call MPI_Send(buf(send_r), NSLEEVE * ny, MPI_REAL, rank_r, rank  , MPI_COMM_WORLD, err)
    else
       call MPI_Send(buf(send_l), NSLEEVE * ny, MPI_REAL, rank_l, rank  , MPI_COMM_WORLD, err)
       call MPI_Recv(buf(recv_r), NSLEEVE * ny, MPI_REAL, rank_r, rank_r, MPI_COMM_WORLD, status, err)
       call MPI_Send(buf(send_r), NSLEEVE * ny, MPI_REAL, rank_r, rank  , MPI_COMM_WORLD, err)
       call MPI_Recv(buf(recv_l), NSLEEVE * ny, MPI_REAL, rank_l, rank_l, MPI_COMM_WORLD, status, err)
    end if
```

次ページへ続く

境界条件の設定（袖領域の交換：F:5/6）

- 上下方向のプロセスと通信
- 実装については水平方向と同じ

```
前ページからの続き
!!$    vertical exchanging
   if(iand(rank / px, 1) == 1) then
      call MPI_Recv(buf(recv_t), NSLEEVE * nx, MPI_REAL, rank_t, rank_t, MPI_COMM_WORLD,
 status, err)
      call MPI_Send(buf(send_b), NSLEEVE * nx, MPI_REAL, rank_b, rank  , MPI_COMM_WORLD,
 err)
      call MPI_Recv(buf(recv_b), NSLEEVE * nx, MPI_REAL, rank_b, rank_b, MPI_COMM_WORLD,
 status, err)
      call MPI_Send(buf(send_t), NSLEEVE * nx, MPI_REAL, rank_t, rank  , MPI_COMM_WORLD,
 err)
   else
      call MPI_Send(buf(send_b), NSLEEVE * nx, MPI_REAL, rank_b, rank  , MPI_COMM_WORLD,
 err)
      call MPI_Recv(buf(recv_t), NSLEEVE * nx, MPI_REAL, rank_t, rank_t, MPI_COMM_WORLD,
 status, err)
      call MPI_Send(buf(send_t), NSLEEVE * nx, MPI_REAL, rank_t, rank  , MPI_COMM_WORLD,
 err)
      call MPI_Recv(buf(recv_b), NSLEEVE * nx, MPI_REAL, rank_b, rank_b, MPI_COMM_WORLD,
 status, err)
   end if
次ページへ続く
```

境界条件の設定（袖領域の交換：F:6/6）

- 受信したデータを，計算データを格納する配列にコピー
  - 受信データは連続
  - 計算配列の境界条件部分に格納
- 派生データ型をうまく使うと，この部分を省略できる

```
前ページからの続き
!!$    copy from receive buffer
    do jj = 0, ny - 1
       do ii = 0, NSLEEVE - 1
          dat(1                + ii, 1 + NSLEEVE + jj) = buf(recv_
l + ii + NSLEEVE * jj)
          dat(1 + NSLEEVE + nx + ii, 1 + NSLEEVE + jj) = buf(recv_
r + ii + NSLEEVE * jj)
       end do
    end do
    do jj = 0, NSLEEVE - 1
       do ii = 0, nx - 1
          dat(1 + NSLEEVE + ii, 1                + jj) = buf(recv_
b + ii + nx * jj)
          dat(1 + NSLEEVE + ii, 1 + NSLEEVE + ny + jj) = buf(recv_
t + ii + nx * jj)
       end do
    end do
  end subroutine set_periodic_boundaries
```

計算データの配布（C言語:1/2）

- ref_scatter.c の中身
- 各プロセス宛てのデータが連続に並ぶようにデータをコピー

```
void scatter_map(const int nx_tot, const int ny_tot, float *map_ful, float *buf_ful,
     const int nx, const int ny, float *map_loc, float *buf_loc,
const int py, const int rank, const int size)
{
  /* prepare send buffer */
  if(rank == 0){
    for(int dst = 0; dst < size; dst++){
      const int rx = dst / py;
      const int ry = dst % py;
      for(int ii = 0; ii < nx; ii++)
        for(int jj = 0; jj < ny; jj++)
          buf_ful[dst * nx * ny + INDEX2D(nx, ny, ii, jj)] = map_ful[INDEX2D(nx_tot, ny_tot,
 rx * nx + ii, ry * ny + jj)];
    }
  }
```

次ページへ続く

計算データの配布（C言語:2/2）

- MPI_Scatter() を用いてデータを配布
- 受信データを，（境界条件用のゴーストセルを含んだ）計算配列へとコピー

```
前ページからの続き
  /* scatter the data */
  int err = MPI_Scatter(buf_ful, nx * ny, MPI_FLOAT, buf_loc, nx * ny, MPI_FLOAT, 0, MPI
_COMM_WORLD);

  /* copy from receive buffer */
  for(int ii = 0; ii < nx; ii++)
    for(int jj = 0; jj < ny; jj++)
      map_loc[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEVE + ii, NSLEEVE + jj)] =
 buf_loc[INDEX2D(nx, ny, ii, jj)];
}
```

計算データの配布（Fortran:1/3）

- ref_scatter.c の中身
- 変数の宣言

```
  subroutine scatter_map(nx_tot, ny_tot, map_ful, buf_ful, nx, ny, map_lo
c, buf_loc, px, rank, size)
    implicit none

    integer, intent(in) :: nx_tot, ny_tot, nx, ny, px, rank, size
    real, intent(in) :: map_ful(nx_tot, ny_tot)
    real, intent(out) :: map_loc(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE)
    real, intent(out) :: buf_loc((nx + 2 * NSLEEVE) * (ny + 2 * NSLEEVE))
, buf_ful(nx_tot * ny_tot)

    integer :: rx, ry, dst
    integer :: ii, jj
    integer :: err
```

次ページへ続く

計算データの配布（Fortran:2/3）

- 各プロセス宛てのデータが連続に並ぶようにデータをコピー

```
前ページからの続き
!!$    prepare send buffer
    if(rank == 0) then
       do dst = 0, size - 1
          rx = mod(dst, px)
          ry = dst / px
          do jj = 0, ny - 1
             do ii = 0, nx - 1
                buf_ful(1 + dst * nx * ny + (ii + nx * jj))
 = map_ful(1 + rx * nx + ii, 1 + ry * ny + jj)
             end do
          end do
       end do
    end if
次ページへ続く
```

計算データの配布（Fortran:3/3）

- MPI_Scatter() を用いてデータを配布
- 受信データを，（境界条件用のゴーストセルを含んだ）計算配列へとコピー

```
前ページからの続き
!!$    scatter the data
    call MPI_Scatter(buf_ful, nx * ny, MPI_REAL, buf_loc, nx * ny, MPI_REAL, 0, MP
I_COMM_WORLD, err)

!!$    copy from receive buffer
    do jj = 0, ny - 1
       do ii = 0, nx - 1
          map_loc(1 + NSLEEVE + ii, 1 + NSLEEVE + jj) = buf_loc(1 + ii + nx * jj)
       end do
    end do
  end subroutine scatter_map
```

計算データの収集（C言語:1/2）

- ref_gather.c の中身
- 送信データを準備した後に，MPI_Gather() で rank = 0 へと送信

```
void gather_map(const int nx_tot, const int ny_tot, float *map_ful, float *buf_ful,
    const int nx, const int ny, float *map_loc, float *buf_loc,
const int py, const int rank, const int size)
{
  /* prepare send buffer */
  for(int ii = 0; ii < nx; ii++)
    for(int jj = 0; jj < ny; jj++)
      buf_loc[INDEX2D(nx, ny, ii, jj)] = map_loc[INDEX2D(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE, NSLEEVE
 + ii, NSLEEVE + jj)];

  /* gather the data */
  int err = MPI_Gather(buf_loc, nx * ny, MPI_FLOAT, buf_ful, nx * ny, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

次ページへ続く

計算データの収集（C言語:2/2）

- rank = 0 は，受信したデータをファイル出力用の配列へとコピー
  - データを適切に並べなおす必要もある

```
前ページからの続き
  /* copy from receive buffer */
  if(rank == 0){
    for(int dst = 0; dst < size; dst++){
      const int rx = dst / py;
      const int ry = dst % py;
      for(int ii = 0; ii < nx; ii++)
        for(int jj = 0; jj < ny; jj++)
          map_ful[INDEX2D(nx_tot, ny_tot, rx * nx + ii, ry * ny + jj)]
 = buf_ful[dst * nx * ny + INDEX2D(nx, ny, ii, jj)];
    }
  }
}
```

計算データの収集（Fortran:1/3）

- ref_gather.f90 の中身
- 変数の宣言

```
  subroutine gather_map(nx_tot, ny_tot, map_ful, buf_ful, nx, ny, map_loc
, buf_loc, px, rank, size)
    implicit none

    integer, intent(in) :: nx_tot, ny_tot, nx, ny, px, rank, size
    real, intent(out) :: map_ful(nx_tot, ny_tot)
    real, intent(in) :: map_loc(nx + 2 * NSLEEVE, ny + 2 * NSLEEVE)
    real, intent(out) :: buf_loc((nx + 2 * NSLEEVE) * (ny + 2 * NSLEEVE))
, buf_ful(nx_tot * ny_tot)

    integer :: rx, ry, dst
    integer :: ii, jj
    integer :: err
```

次ページへ続く

計算データの収集（Fortran:2/3）

- 送信データを準備した後に，MPI_Gather() で rank = 0 へと送信

```
前ページからの続き
!!$    prepare send buffer
    do jj = 0, ny - 1
       do ii = 0, nx - 1
          buf_loc(1 + ii + nx * jj) = map_loc(1 + NSLEEVE + ii, 1 + NSLEEVE + jj)
       end do
    end do

!!$    gather the data
    call MPI_Gather(buf_loc, nx * ny, MPI_REAL, buf_ful, nx * ny, MPI_REAL, 0, MP
I_COMM_WORLD, err)
次ページへ続く
```

計算データの収集（Fortran:3/3）

- rank = 0 は，受信したデータをファイル出力用の配列へとコピー
  - データを適切に並べなおす必要もある

```
前ページからの続き
!!$    copy from receive buffer
    if(rank == 0) then
       do dst = 0, size - 1
          rx = mod(dst, px)
          ry = dst / px
          do jj = 0, ny - 1
             do ii = 0, nx - 1
                map_ful(1 + rx * nx + ii, 1 + ry * ny + jj)
 = buf_ful(1 + dst * nx * ny + (ii + nx * jj))
             end do
          end do
       end do
    end if
  end subroutine gather_map
```

発展的な話題

- もうちょっと楽に書けないものか？　と感じた方もいるはず
  - MPI通信前後にデータを並べなおす処理は自動化できないか？（実装自体が面倒，速度低下の原因，バグの元にもなる）
  - rootプロセスにデータを集めずに並列ファイルアクセスできないか？（今の実装では，rootプロセスは計算領域全体を格納できるメモリを確保しておく必要があるため大規模化の障害となる．とはいえ，全プロセスがばらばらにファイル出力するとファイル数が膨大になって大変）
- どちらの内容についても，より簡単に実装することは可能です
  - 「基礎」からは外れる内容なので，今回の講習会には含めていません
  - それぞれ，「派生データ型」や「MPI-IO」を使うことで実現可能

最後に

- アンケートの回答をお願いします
  - 講習会の改善のために有用な情報なので，ご協力お願いします

- 本講習会アカウントは，11/13（金） 9:00まで使えます
  - キュー名はlecture-flatです
    （tutorial-flatは10/13の17:00以降使えなくなります）
  - 最大15分まで
  - 最大ノード数は16ノードまで