第203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：2023年4月26日（水） 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：Wisteria/BDEC-01 (Odyssey)
- 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）

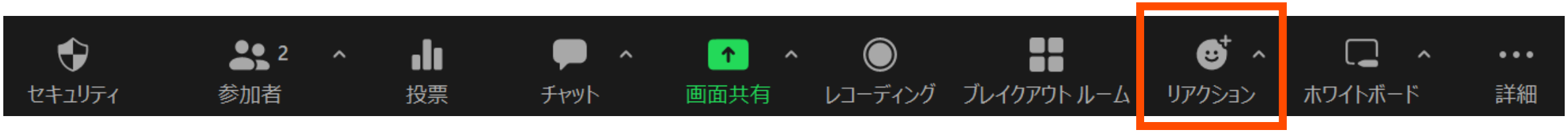
Zoom関連

- 「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.adm.u-tokyo.ac.jp/zoom/how_to_use

「手を挙げる」方法

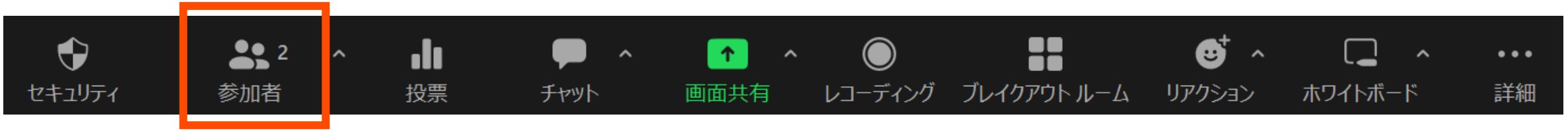
1．Zoomメニュー中の「リアクション」をクリック



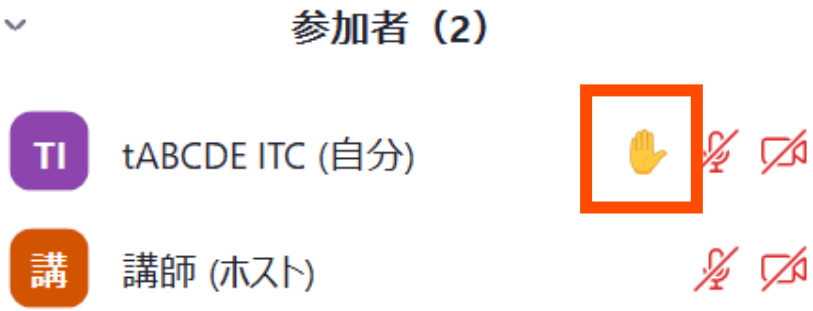
2．ポップアップで表示された「挙手」をクリック

手が挙がっていることの確認方法

1. Zoomメニュー中の「参加者」をクリックして，参加者一覧を表示



2. 表示された参加者一覧の，自分のところを見ると手が挙がっている



「手をおろす」方法

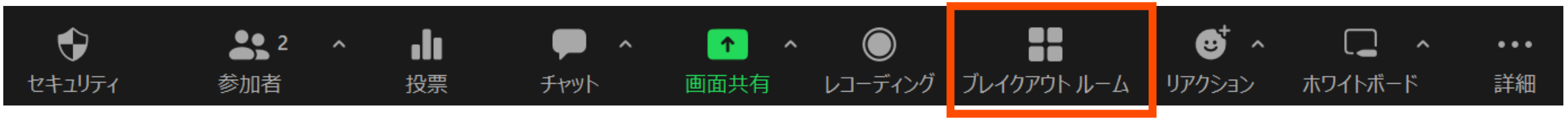
1. ポップアップで表示されている「手をおろす」をクリック

- もし「手をおろす」が表示されていなければ，「リアクション」の中から探す

ブレイクアウトルーム（1/6）

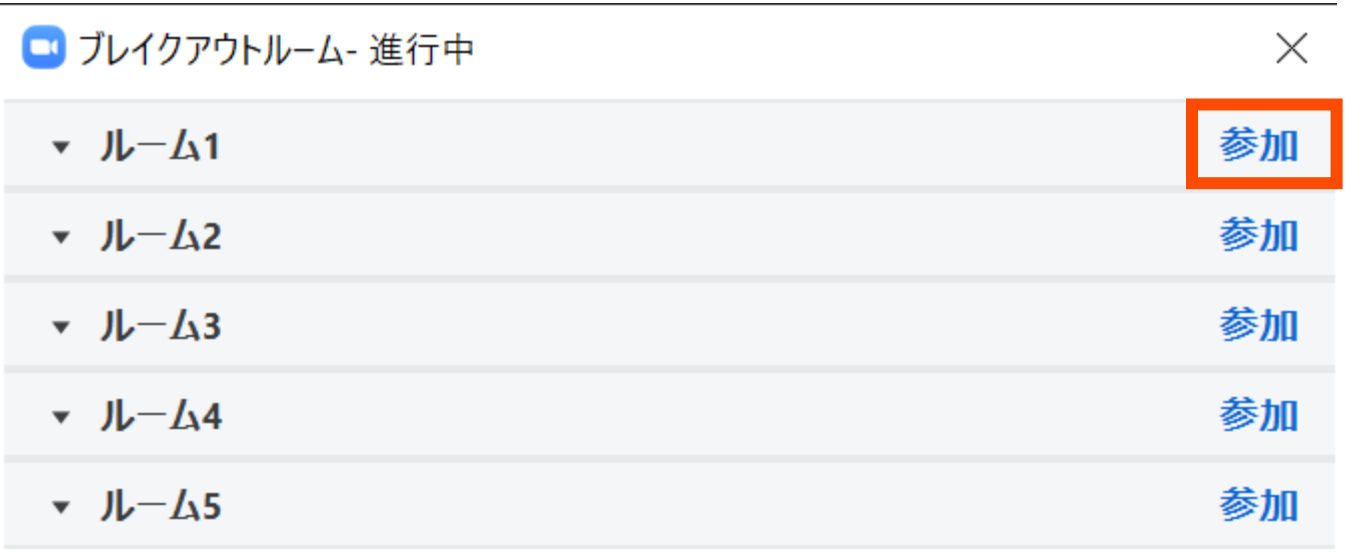
- 演習時に使用するかもしれません
- 演習中に「ヘルプを求める」ことができます
  - ホストを招待した後に「画面を共有」することで，皆さんの記述したプログラムを一緒に見ながら問題解決にあたります

- Zoomメニュー中の「ブレイクアウトルーム」をクリック



ブレイクアウトルーム（2/6）

- 進行中のブレイクアウトルームのリストが表示されるので，空いている部屋に「参加」してください
  - 左の例では5部屋がすべて空室，右の例ではルーム1のみ参加者がいる



ブレイクアウトルーム（3/6）

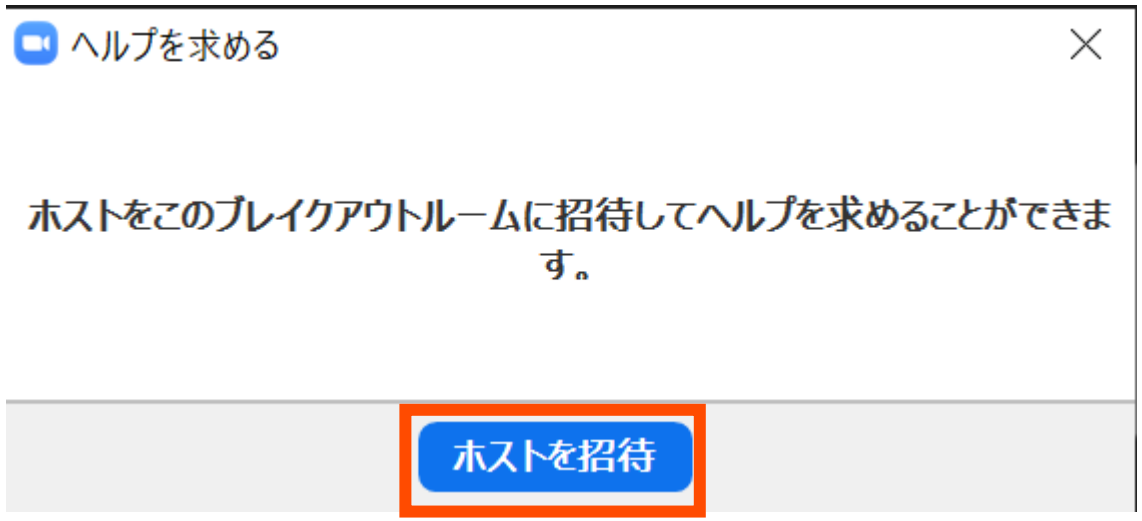
- 「参加」をクリックすると確認画面が出てくるので，「はい」を選択すると入室できます

ブレイクアウトルーム（4/6）

- 再度メニュー中の「ブレイクアウトルーム」をクリックすると，「ヘルプを求める」が増えているのでクリックしてください

ブレイクアウトルーム（5/6）

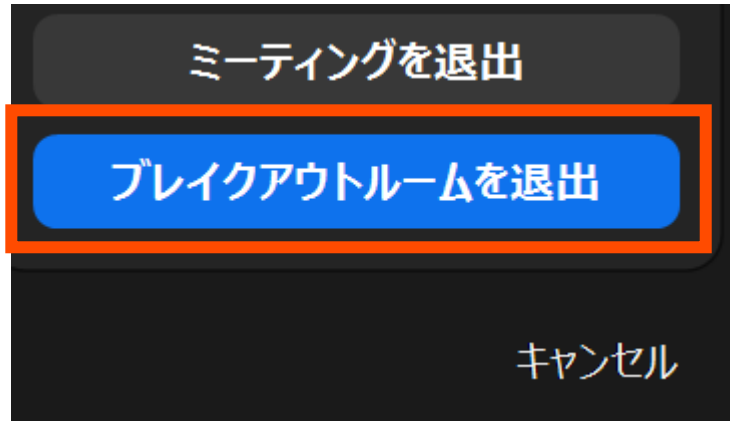
- ポップアップで出てきた「ホストを招待」をクリック
- ホスト（講師）の参加待ちに移行します（画面はそのまま）
  - 他の受講者のヘルプ中など，直ちに対応できない場合もあります



ブレイクアウトルーム（6/6）

- 問題解決後は，Zoomメニュー中の「ルームを退出する」

- 「ブレイクアウトルームを退出」が表示されるのでクリックして，元の講習会会場にお戻りください
  - 間違えて「ミーティングを退出」すると講習会から退出します



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

- 左側のメニューバーのチャンネル一覧内に「第203回-mpi基礎」があるので，クリック

- 見つからない場合
  1. 「チャンネルを追加する」をクリック
  2. 「チャンネル一覧を確認する」をクリック
  3. 「第203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

- 最下部に入力欄があるので，質問内容を記載して Ctrl+Enter
  - 入力後に右下の「メッセージを送信する」をクリックしても同じ
    （メッセージ入力前には，「メッセージを送信する」は押せない）

- コードを入力する際には，「コードブロック」がおすすめ
  - 枠が生成されるので，この中にコピペするのが簡単かつ見やすい
  - ```（JIS配列ならばShift+@を3連打）しても枠が生成される

自分が参加したスレッドのみの表示方法

- 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
  - 質問内容には，「スレッドで返信する」形式で回答するので，自分が聞いた内容のみが表示できます

Slackメッセージの保存

- 講習会slackは無償版なので，90日以上過去のメッセージは表示されなくなります
- メッセージの保存方法は複数ありますが，比較的簡単な方法を紹介
- SlatickHTML: https://sites.google.com/view/slatickhtml/
  - Chrome拡張機能なので，Slack Appの作成などが不要
    - Vivaldi，Edgeなどのブラウザからも同様に使用できる（はずですが，確認していません）
  - （添付するファイル形式をチェックしておけば）ファイル取得も可能
  - HTMLファイルとして保存するため，保存後はブラウザを用いて表示可能
  - デフォルトでは今年のメッセージしか表示されないので注意（上部のメニューで「全て表示」か該当年を選択すれば，過去のメッセージも表示されます）

ユーザアカウント

- 使用システム： Wisteria/BDEC-01 (Odyssey)
  - $ ssh USERNAME@wisteria.cc.u-tokyo.ac.jp
- 本講習会でのユーザ名（上記コマンドでの USERNAME のこと）
  - 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
  - 5/26 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
  - Slack で質問していただいても結構です
  - （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/share/mpi_basic

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/$USER
   ```
2. /work/gt00/share/mpi_basic にあるサンプルをコピー
   ```
   $ cp /work/gt00/share/mpi_basic/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-samples.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
```

余談：ファイルを固める・圧縮するコマンド
```
$ tar -acvf filename.tar.[gz bz2 xz] files
```
- 今回のサンプル作成時には --exclude-vcs もつけてバージョン管理システム（git などのこと）関連のファイルを除外して固めた

TIPS（タブ補完）

- ターミナル上では[Tab]キーを入力してタブ補完を効かせながら入力すると良い
  - キー入力数が減るのでお得（自動的にtypoも減る）
  - 自動補完できる部分だけを入力するので，とりあえず[Tab]を入力した上で補っていけば良い
    - Windowsとは違い，候補を順番に表示するようなことはない
- 先ほど入力してもらったコマンド群の場合：
  1. `$ cd /wo[Tab]/gt00/$USER`
  2. `$ cp /wo[Tab]/gt00/s[Tab]/mp[Tab]/m[Tab] .`
  3. `$ tar -xvf m[Tab]`
  4. `$ cd m[Tab]`

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. 環境設定
   ```
   $ module load fj
   ```
3. コンパイル
   ```
   $ make
   ```
4. 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-o
#PJM -L node=12
#PJM --mpi proc=576
#PJM -L elapse=00:01:00
#PJM -g gt00

module load fj fjmpi
mpiexec ./hello
```

リソースグループ名：lecture-o

利用ノード数：12ノード使用

MPIプロセス数：576 (= 48 * 12)

実行時間制限：1分

利用グループ名：gt00

環境設定

MPIジョブを576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-o”になっているので，これを“tutorial-o”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.XXXXXX.out    # 標準出力ファイル
   run.sh.XXXXXX.err    # 標準エラー出力ファイル
   ```
   ジョブ名 + . + ジョブID + .[out err] というファイル名
3. 標準出力ファイルの中身を見てみる
   ```
   $ cat run.sh.XXXXXX.out
   ```
   “Hello world!”が576 (= 48プロセス * 12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576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.XXXXXX.out | wc -l
  576 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.XXXXXX.out | sort -k4n | less
  rank: 0 から始まって，rank: 575 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

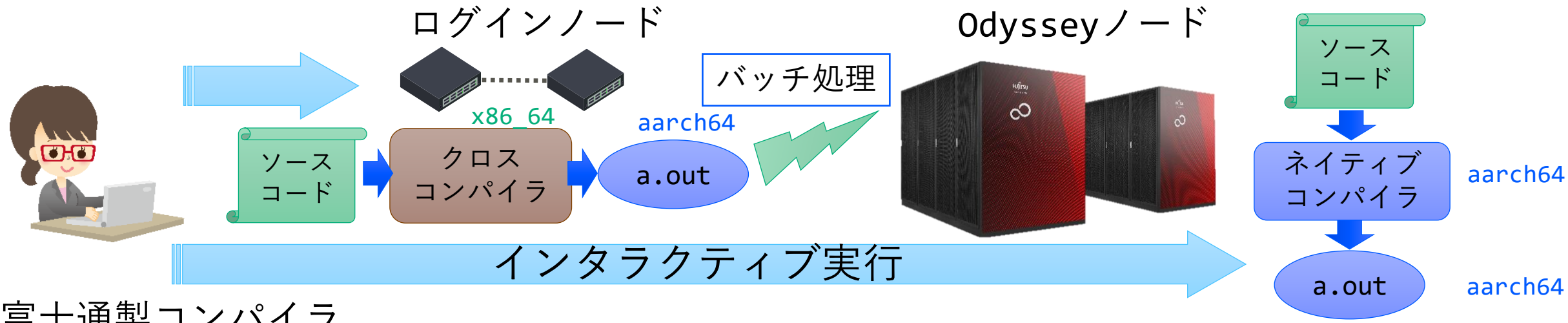
全プロセス数を取得
（全ランクで共通の値）

自分のIDを取得
（全ランクで異なる値）

MPIの終了

コンパイラの種類と実行(Odyssey)

- ログインノードとOdysseyの計算ノードとで、CPUの命令セットが大きく異なる
  - ログインノード：命令セットアーキテクチャ Intel CascadeLake＋AVX512, x86_64
  - Odyssey計算ノード： Fujitsu A64FX, 命令セットアーキテクチャ ARM v8.2 + SVE, aarch64



- 富士通製コンパイラ
  - module load fj で使用可能になる（またはodyssey）
  - ネイティブコンパイラ： ＜コンパイラの種類名＞
  - クロスコンパイラ： ＜コンパイラの種類名＞px
  - MPI: module load fjmpi が必要
    mpi+コンパイラ名（例：mpifccpx）

言語	ネイティブコンパイラ	クロスコンパイラ
C	fcc	fccpx
C++	FCC	FCCpx
Fortran	frt	frtpx

moduleの指定

- コンパイラ・ライブラリ等の環境をセットアップ
  - Odyssey向け富士通コンパイラ、MPIを使用
    ```
    $ module load fj  （fjmpiは自動的にロードされる）　または
    $ module load odyssey
    ```
  - Aquarius向けgcc、cuda, Open MPI(CUDA対応)を使用
    ```
    $ module load gcc cuda ompi-cuda  または
    $ module load aquarius cuda ompi-cuda
    ```
- 現在指定済みのmoduleを確認するには
  ```
  $ module list
  ```
- 困った時（module環境を整理したくなった時）は
  ```
  $ module purge
  ```

moduleの一覧

- 現在利用中の環境で追加できるものを確認
  ```
  $ module avail
  ```
- 使い方の確認（例はModuleNameというmoduleのヘルプを表示）
  ```
  $ module help ModuleName
  ```
- 設定されるPATHなどの確認（例はModuleNameというmoduleの場合）
  ```
  $ module show ModuleName
  ```

- 全ての環境を確認（Wisteria/BDEC-01向けに提供されるコマンド）
  ```
  $ show_module
  ```

```
ApplicationName         ModuleName          Node             BaseCompiler/MPI
-------------------------------------------------------------------------------
Archiconda              archiconda3/0.2.3   odyssey          -
Arm Forge               forge/21.0.2        aquarius         -
…
```

Wisteria/BDEC-01でのジョブ実行

- 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
  - Wisteria/BDEC-01 (Odyssey)では，最大2304ノード（110592コア），24時間まで
  - Wisteria/BDEC-01 (Aquarius)では，最大8ノード（64 GPUs），24時間まで

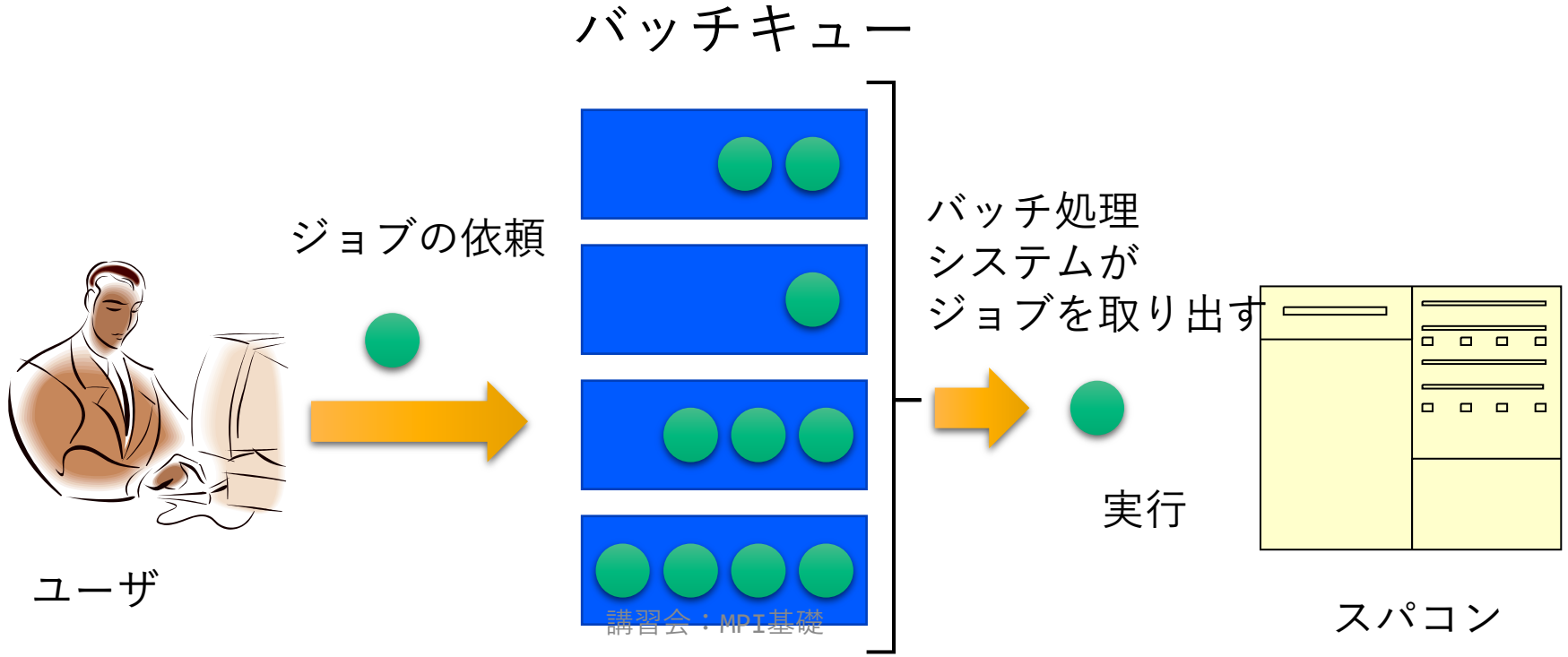
※講習会アカウントでは
バッチジョブ実行のみ，
最大12ノード15分まで

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（48コア）：30分まで
  - 12ノード（576コア）：10分まで

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

- Wisteria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 --rscuse`
  - 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`

参考：インタラクティブ実行のやり方（本講習会アカウントでは使えません）

- 1ノード実行
  ```
  $ pjsub --interact -g グループ名 -L rg=interactive-o,elapse=01:00
  ```
- 12ノード実行
  ```
  $ pjsub --interact -g グループ名 -L rg=interactive-o,node=12,elapse=01:00
  ```

- インタラクティブ用のノードが全て使われている場合，資源が空くまでログインできません
- 本講習会用のアカウントでは使えません

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-o
  - 最大15分まで
  - 最大ノード数は12ノード（576コア）まで
- 本講習会終了後のキュー名
  - lecture-o
  - 利用条件tutorial-o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
- Wisteria/BDEC-01 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`

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 1_hybrid
   ```
2. 環境設定（ログイン後に一度だけ実行すれば良い）
   ```
   $ module load fj
   ```
3. コンパイル
   ```
   $ make
   ```
4. 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-o
#PJM -L node=12
#PJM --mpi proc=12
#PJM --omp thread=48
#PJM -L elapse=00:01:00
#PJM -g gt00

module load fj fjmpi
mpiexec ./hello_omp
```

リソースグループ名：lecture-o

利用ノード数：12ノード使用

MPIプロセス数：12

OpenMPスレッド数：48

実行時間制限：1分

利用グループ名：gt00

MPIジョブを12プロセスで実行

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-o”になっているので，これを“tutorial-o”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.XXXXXX.out    # 標準出力ファイル
   run.sh.XXXXXX.err    # 標準エラー出力ファイル
   ```
   ジョブ名 + . + ジョブID + .[out err] というファイル名
3. 標準出力ファイルの中身を見てみる
   ```
   $ cat run.sh.XXXXXX.out
   ```
   “Hello world!”が576 (= 48スレッド * 12プロセス) 行あれば成功

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

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 exercise/reference (comment out
for exercise)
# REF   := ref_

# environment
CC  := mpifccpx

# option(s)
CFLAGS  := -Kfast -Koptmsg=2

# 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 exercise/reference (comment out
for exercise)
REF := ref_

# environment
CC  := mpifccpx

# option(s)
CFLAGS  := -Kfast -Koptmsg=2

# 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-o”になっているので，これを“tutorial-o”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

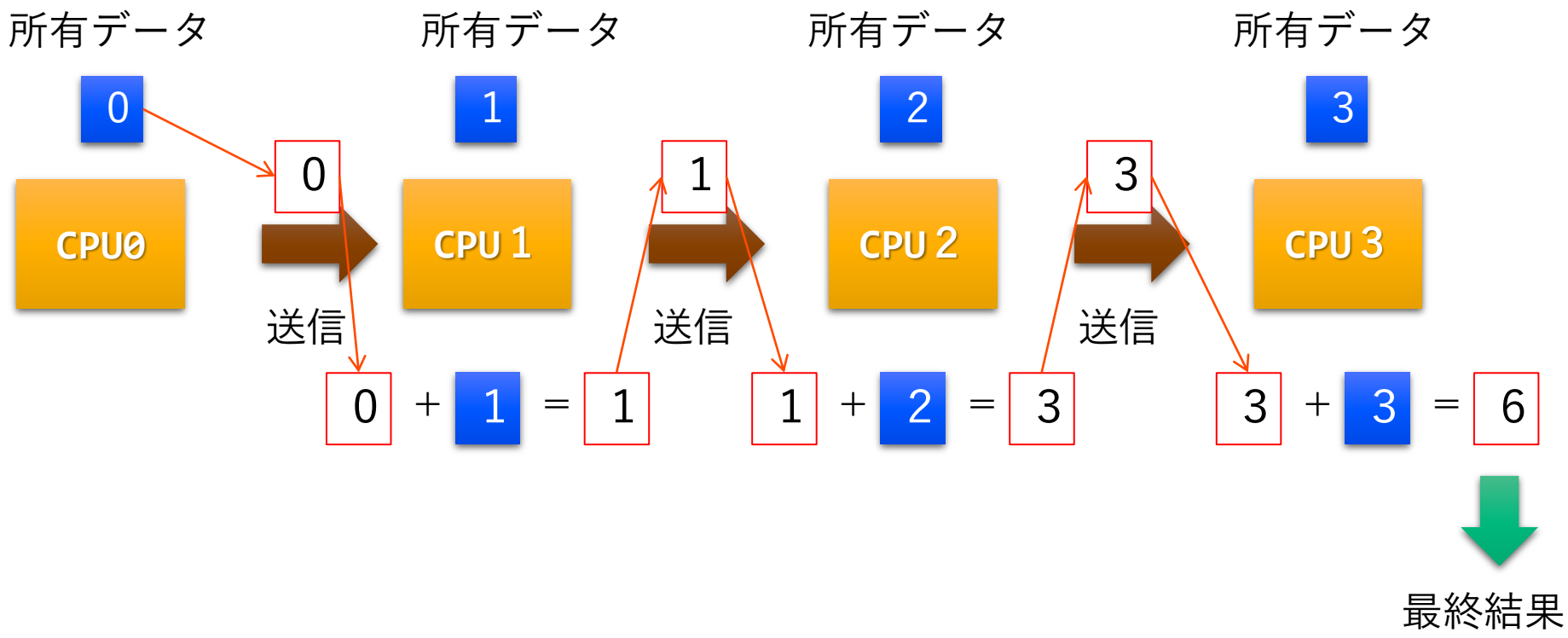
```
$ pjsub run.sh
```

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）

- 37行目，44行目のMPI関数部分が部分的に実装されているので，穴埋めして動作するように書き換えてください
  - 左隣（rank - 1）の人から値を受け取り，右隣（rank + 1）の人に送る
  - tag は送受信のペア間で値が一致するように指定

```
  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) */
  if(rank > 0)
    err = MPI_Recv(&recv, 1, MPI_INT, int source, int tag, MPI_COMM_WORLD, &status);

  int send = rank + recv;
  /* send current sum "send" to the right process (= rank + 1) if the right process exists
(i.e. rank + 1 <= size - 1) */
  if(rank < size - 1)
    err = MPI_Send(&send, 1, MPI_INT, int dest, int tag, MPI_COMM_WORLD);
```

総和演算プログラムの演習（Fortran）

- 41行目，51行目のMPI関数部分が部分的に実装されているので，穴埋めして動作するように書き換えてください
  - 左隣（rank - 1）の人から値を受け取り，右隣（rank + 1）の人に送る
  - tag は送受信のペア間で値が一致するように指定

```
  recv = 0
!!$  receive partial sum from the left process
  if(rank > 0) then
    call MPI_Recv(recv, 1, MPI_INTEGER, integer :: source, integer :: tag, MPI_COMM_WORLD, stat
us, err)
  end if

  send = rank + recv
!!$  send current sum "send" to the right process if the right process exists
  if(rank < size - 1) then
    call MPI_Send(send, 1, MPI_INTEGER, integer :: dest, integer :: tag, MPI_COMM_WORLD, err)
  end if
```

MPI_Send

- err = MPI_Send(*buf, count, datatype, dest, tag, comm);
  - buf: 送信領域の先頭アドレスを指定
  - count: 整数型．送信領域のデータ要素数を指定
  - datatype: MPI_Datatype型．送信領域のデータ型を指定
    - MPI_INT（整数型），MPI_FLOAT（単精度実数型），MPI_DOUBLE（倍精度実数型）など
  - dest: 整数型．送信先の（comm内での）プロセスランクを指定
  - tag: 整数型．メッセージにつけるタグの値を指定
  - comm: MPI_Comm型．コミュニケータを指定
    - 通常はMPI_COMM_WORLDを指定すればよい
  - err（戻り値）: 整数型．エラーコードが入る

MPI_Recv (1/2)

- `err = MPI_Recv(*buf, count, datatype, source, tag, comm, *status);`
  - `buf`: 受信領域の先頭アドレスを指定
  - `count`: 整数型．受信領域のデータ要素数を指定
  - `datatype`: MPI_Datatype型．受信領域のデータ型を指定
  - `source`: 整数型．メッセージの送信元のランクを指定
    - 任意のランクから受信したいときは，`MPI_ANY_SOURCE`を指定
  - `tag`: 整数型．受信したいメッセージについているタグを指定
    - 任意のタグ値のメッセージを受信したいときは，`MPI_ANY_TAG`を指定

MPI_Recv (2/2)

- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- status: MPI_Status型．受信状況に関する情報が入る
  - 必ず専用の型宣言をした配列を確保する
  - C言語： MPI_Status status;
  - Fortran: integer :: status(MPI_STATUS_SIZE)
  - Fortran 2008: type(MPI_Status) :: status
  - 要素数がMPI_STATUS_SIZEの整数配列が確保される
  - 受信メッセージの送信元のランクがstatus[MPI_SOURCE]，タグがstatus[MPI_TAG]に代入される
- err: 整数型．エラーコードが入る

総和演算プログラムの実装例（C言語）

- 左のプロセス（rank - 1）から値を受信
- 右のプロセス（rank + 1）へと値を送信
- 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);
```

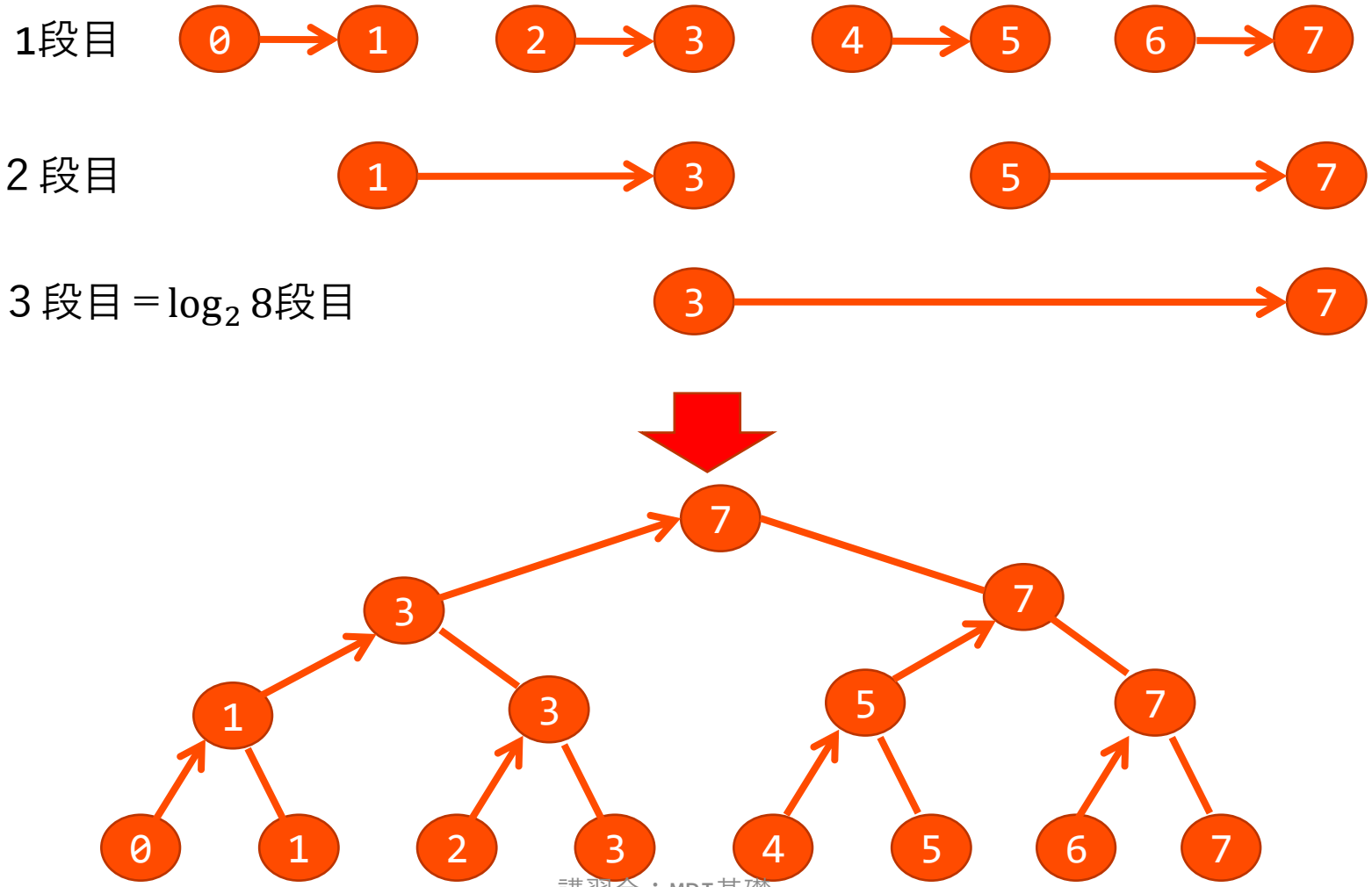
総和演算プログラムの実装例（Fortran）

- 左のプロセス（rank - 1）から値を受信
- 右のプロセス（rank + 1）へと値を送信
- 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）

- 50行目，57行目のMPI関数部分が部分的に実装されているので，穴埋めして動作するように書き換えてください
- rank - disp のプロセスから値を受信，rank + disp へと送信
- tag は送受信のペア間で値が一致するように指定

```
  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 */
      err = MPI_Recv(&recv, 1, MPI_INT, int source, int tag, M
PI_COMM_WORLD, &status);
      send += recv;
      disp <<= 1;
    }
    else{
      /* send current sum "send" to the target process*/
      err = MPI_Send(&send, 1, MPI_INT, int dest, int tag, MPI
_COMM_WORLD);
      break;
    }
  }
```

総和演算プログラムの演習（Fortran）

- 54行目，62行目のMPI関数部分が部分的に実装されているので，穴埋めして動作するように書き換えてください
- rank - disp のプロセスから値を受信，rank + disp へと送信
- tag は送受信のペア間で値が一致するように指定

```
!!$  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
      call MPI_Recv(recv, 1, MPI_INTEGER, integer :: source,
integer :: tag, MPI_COMM_WORLD, status, err)
      send = send + recv
      disp = disp * 2
    else
!!$         send current sum "send" to the target process
      call MPI_Send(send, 1, MPI_INTEGER, integer :: dest, in
teger :: tag, MPI_COMM_WORLD, err)
      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 ＝ $\log_2\left(N_{\mathrm{p}}\right)$ 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
  - $N_p = 512\ \left(= 2^9\right)$の場合には，511回 対 9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）

- 38行目のMPI関数部分が部分的に実装されているので，穴埋めして動作するように書き換えてください
- send の総和を recv に格納してください
- 総和（MPI_SUM）を計算してください

```
  /* 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 p
rocess (rank = 0) receives the result */
  err = MPI_Reduce(const void *sendbuf, void *recvbuf, 1, MPI_INT, MPI_Op op, 0,
 MPI_COMM_WORLD);
```

総和演算プログラムの演習（Fortran）

- 39行目のMPI関数部分が部分的に実装されているので，穴埋めして動作するように書き換えてください
- send の総和を recv に格納してください
- 総和（MPI_SUM）を計算してください

```
!!$  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
 process (rank = 0) receives the result
  call MPI_Reduce(<type> :: sendbuf(*), <type> :: recvbuf(*), 1, MPI_INTEGER,
integer :: op, 0, MPI_COMM_WORLD, err)
```

MPI_Reduce（1/2）

- `err = MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op, root, comm);`
  - `sendbuf`: 送信領域の先頭アドレスを指定
  - `recvbuf`: 受信領域の先頭アドレスを指定
    - rootで指定したプロセスのみで書き込まれる
    - 送信領域と受信領域は同一であってはならない（異なる配列を確保）
    - ↑ rootのsendbufとして`MPI_IN_PLACE`を指定することで同一の領域を指定可能
  - `count`: 整数型．送信領域のデータ要素数を指定
  - `datatype`: MPI_Datatype型．送信領域のデータ型を指定
    - Fortranの場合：<最小・最大値と位置>を返す演算（MPI_MINLOCなど）を指定する場合は，MPI_2INTEGER（整数型），MPI_2REAL（単精度型），MPI_2DOUBLE_PRECISION(=MPI_2REAL8)（倍精度型）を指定

MPI_Reduce（2/2）

- op: MPI_Op型．演算の種類を指定
  - MPI_SUM: 総和
  - MPI_PROD: 積
  - MPI_MAX: 最大値
  - MPI_MIN: 最小値
  - MPI_MAXLOC: 最大値とその位置
  - MPI_MINLOC: 最小値とその位置
- root: 整数型．結果を受け取るプロセスのランクを指定
  - comm内の全プロセスが同じ値を指定する必要がある
- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- err: 整数型．エラーコードが入る

総和演算プログラムの実装例（C言語）

- 送信バッファは &send を指定
- 受信バッファは &recv を指定
- 総和を求めるので，MPI_Op には MPI_SUM を指定

```c
/* 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）

- 送信バッファは send を指定
- 受信バッファは recv を指定
- 総和を求めるので，MPI_Op には MPI_SUM 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.XXXXXX.out）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）

性能プロファイラ

- 詳細はWebポータルから「ドキュメント閲覧」➔
  Wisteria/BDEC-01 システム利用手引書
  6. 開発ツール
  または
  プロファイラ利用手引書
  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）：

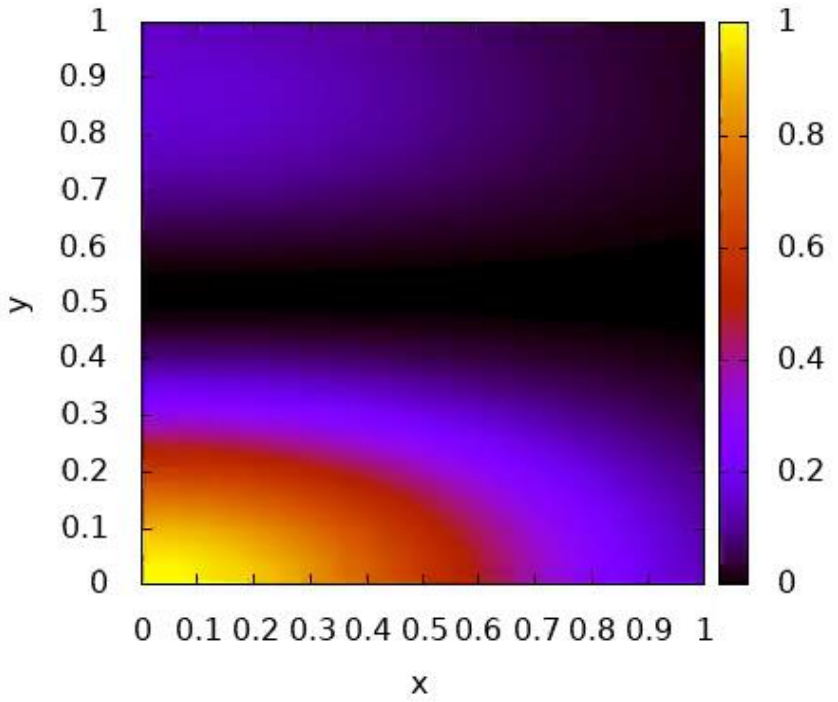
$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{a\,\Delta t}{h^2}\left(u_{i,j-1}^{n} + u_{i,j+1}^{n} + u_{i-1,j}^{n} + u_{i+1,j}^{n} - 4u_{i,j}^{n}\right)$$

- 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

```
192 192     # x，y-方向のメッシュ数（N_x, N_y）
24 24       # 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576）

参考：画像の表示方法

- Wisteria/BDEC-01 にログインする際に -Y つきでログイン
  ```
  $ ssh -Y username@wisteria.cc.u-tokyo.ac.jp
  $ cd /work/gt00/$USER/…/fig
  $ eog map000.png &
  ```
  注：Cygwin からは"cannot open display"となり表示できません
- 手元のPCに画像ファイルをコピーして表示
  ```
  $ rsync -av username@wisteria.cc.u-
  tokyo.ac.jp:/work/.../fig .
  ```
  - rsync以外にはsftp，scpなど（ただしscpはOpenSSH的に非推奨とのこと）
- Windowsの方は，WinSCPを使ってダウンロードしても良いです
- エディタの機能を利用して表示
  - EmacsのTRAMP機能や，Visual Studio CodeのRemote Windowなど

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

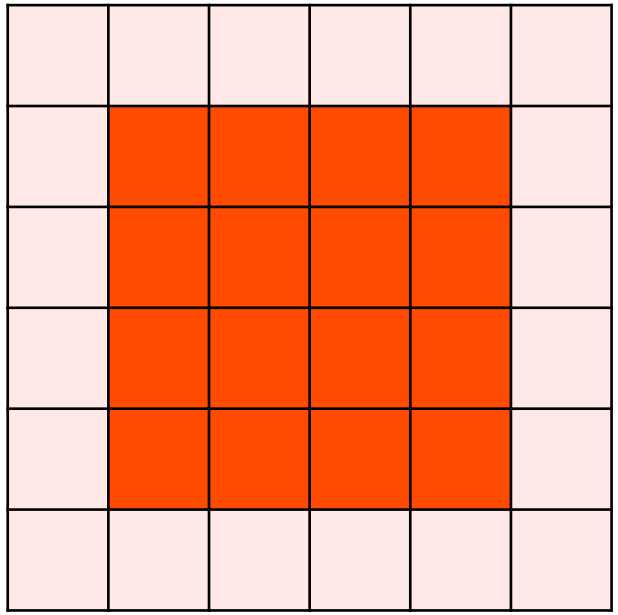
- topology.c を編集する場合

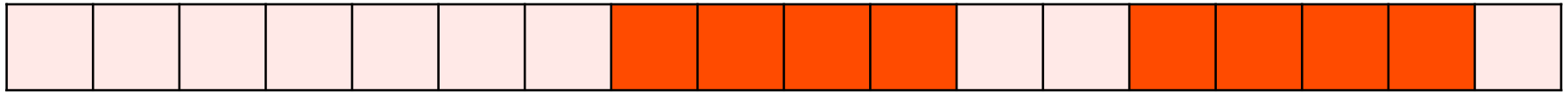
```
# source(s)
SRC_EXE := $(REF)diffusion.c
SRC_EXE += topology.c
SRC_EXE += $(REF)boundary.c
SRC_EXE += $(REF)scatter.c
SRC_EXE += $(REF)gather.c
SRC_EXE += io.c
```

  - 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

MPI_Bcast

- err = MPI_Bcast(*buffer, count, datatype, root, comm);
  - buffer: 送信および受信領域の先頭アドレスを指定
  - count: 整数型．buffer上のデータ要素数を指定
  - datatype: MPI_Datatype型．bufferのデータ型を指定
  - root: 整数型．メッセージを送信するプロセスのIDを指定
    - 全プロセスが同じ値を指定する
  - comm: MPI_Comm型．通信に関与するコミュニケータを指定
  - err: 整数型．エラーコードが入る

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
  - 参考として，nx_tot だけ実装済み

```
  /* 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) */
  err = MPI_Bcast(&nx_tot, 1, MPI_INT, 0, MPI_COMM_WORLD);
  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
  - 参考として，nx_tot だけ実装済み

```
!!$  broadcast configuration of the simulation to all processes
!!$  broadcast nx_tot, ny_tot, px, py, diff_coeff, courant, fin, snapshot_interval, an
d prev from the root process (rank = 0) to all processes
!!$  MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
!!$            <type>    BUFFER(*)
!!$            INTEGER   COUNT, DATATYPE, ROOT, COMM, IERROR
  call MPI_Bcast(nx_tot, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, err)
  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）
     （この部分は実装済み）
  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
     （この部分は実装済み）

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)
{
  実装済み部分（省略）

  /* exchange sleeve regions */
  update_program;

  実装済み部分（省略）
}
```

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)

  実装済み部分（省略）

!!$    exchange sleeve regions
    update_program

  実装済み部分（省略）

  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）
     （この部分は実装済み）
  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（この部分は実装済み）

MPI_Scatter（1/2）

- err = MPI_Scatter(*sendbuf, sendcount, sendtype, *recvbuf, recvcount, recvtype, root, comm);
  - sendbuf: 送信領域の先頭アドレスを指定
  - sendcount: 整数型．送信領域のデータ要素数を指定
    - 1プロセス宛てに送信する要素数
    - MPI_Scatterでは全プロセスに配るメッセージサイズは同じでないといけない
  - sendtype: MPI_Datatype型．送信領域のデータ型を指定
  - recvbuf: 受信領域の先頭アドレスを指定
    - （原則として）送信領域と受信領域は同一であってはならない
    - ↑ rootのrecvbufとしてMPI_IN_PLACEを指定することで同一の領域を指定可能
  - recvcount: 整数型．受信領域のデータ要素数を指定

MPI_Scatter（2/2）

- recvtype: MPI_Datatype型．受信領域のデータ型を指定
- root: 整数型．結果を受け取るプロセスのランクを指定
  - comm内の全プロセスが同じ値を指定する必要がある
- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- err: 整数型．エラーコードが入る

- sendbuf, sendcount, sendtypeの指定
  - rootでのみ意味を持つ（他のプロセスでの指定は無視される）
  - （実用上は）全プロセスがrootでの指定値を書いておけばよい

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)
{

  実装済み部分（省略）

  /* scatter the data */
  update_program;

  実装済み部分（省略）

}
```

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

  実装済み部分（省略）

!!$    scatter the data
    update_program

  実装済み部分（省略）

  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（この部分は実装済み）
  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）
     （この部分は実装済み）

MPI_Gather（1/2）

- err = MPI_Gather(*sendbuf, sendcount, sendtype, *recvbuf, recvcount, recvtype, root, comm);
  - sendbuf: 送信領域の先頭アドレスを指定
  - sendcount: 整数型．送信領域のデータ要素数を指定
  - sendtype: MPI_Datatype型．送信領域のデータ型を指定
  - recvbuf: 受信領域の先頭アドレスを指定
    - （原則として）送信領域と受信領域は同一であってはならない
    - ↑ rootのsendbufとしてMPI_IN_PLACEを指定することで同一の領域を指定可能
  - recvcount: 整数型．受信領域のデータ要素数を指定
    - 1プロセスから受信する要素数
    - MPI_Gatherでは全プロセスのメッセージサイズは同じでないといけない

MPI_Gather（2/2）

- recvtype: MPI_Datatype型．受信領域のデータ型を指定
- root: 整数型．結果を受け取るプロセスのランクを指定
  - comm内の全プロセスが同じ値を指定する必要がある
- comm: MPI_Comm型．通信に関与するコミュニケータを指定
- err: 整数型．エラーコードが入る

- recvbuf, recvcount, recvtypeの指定
  - rootでのみ意味を持つ（他のプロセスでの指定は無視される）
  - （実用上は）全プロセスがrootでの指定値を書いておけばよい

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)
{

  実装済み部分（省略）

  /* gather the data */
  write_program;

  実装済み部分（省略）

}
```

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:1/2）

- ref_boundary.c の中身
- 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

set_periodic_boundaries() の中身

```
  /* 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:2/2）

- 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);
  }
```

境界条件の設定（袖領域の交換：F:1/2）

- ref_boundary.f90 の中身
- 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

set_periodic_boundaries()の中身

```
    if(iand(mod(rank, px), 1) == 1) then
       call MPI_Recv(buf(recv_r), NSLEEVE * ny, MPI_REAL, rank_r, rank_r, MPI_COMM_WORL
D, status, err)
       call MPI_Send(buf(send_l), NSLEEVE * ny, MPI_REAL, rank_l, rank  , MPI_COMM_WORL
D, err)
       call MPI_Recv(buf(recv_l), NSLEEVE * ny, MPI_REAL, rank_l, rank_l, MPI_COMM_WORL
D, status, err)
       call MPI_Send(buf(send_r), NSLEEVE * ny, MPI_REAL, rank_r, rank  , MPI_COMM_WORL
D, err)
    else
       call MPI_Send(buf(send_l), NSLEEVE * ny, MPI_REAL, rank_l, rank  , MPI_COMM_WORL
D, err)
       call MPI_Recv(buf(recv_r), NSLEEVE * ny, MPI_REAL, rank_r, rank_r, MPI_COMM_WORL
D, status, err)
       call MPI_Send(buf(send_r), NSLEEVE * ny, MPI_REAL, rank_r, rank  , MPI_COMM_WORL
D, err)
       call MPI_Recv(buf(recv_l), NSLEEVE * ny, MPI_REAL, rank_l, rank_l, MPI_COMM_WORL
D, 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:2/2）

- 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
```

計算データの配布（C言語）

- ref_scatter.c の中身
- MPI_Scatter() を用いてデータを配布

```
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)
{
  実装済み部分（省略）

  /* scatter the data */
  int err = MPI_Scatter(buf_ful, nx * ny, MPI_FLOAT, buf_loc, nx * ny, MPI_FLOAT, 0, MPI_COM
M_WORLD);

  実装済み部分（省略）
}
```

計算データの配布（Fortran）

- ref_scatter.c の中身
- MPI_Scatter() を用いてデータを配布

```
  subroutine scatter_map(nx_tot, ny_tot, map_ful, buf_ful, nx, ny, map_lo
c, buf_loc, px, rank, size)

  実装済み部分（省略）

!!$    scatter the data
    call MPI_Scatter(buf_ful, nx * ny, MPI_REAL, buf_loc, nx * ny, MPI_RE
AL, 0, MPI_COMM_WORLD, err)

  実装済み部分（省略）
```

計算データの収集（C言語）

- ref_gather.c の中身
- 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)
{
 実装済み部分（省略）

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

 実装済み部分（省略）
```

計算データの収集（Fortran）

- ref_gather.f90 の中身
- MPI_Gather() で rank = 0 へと送信

```
  subroutine gather_map(nx_tot, ny_tot, map_ful, buf_ful, nx, ny, map_loc
, buf_loc, px, rank, size)

  実装済み部分（省略）

!!$    gather the data
    call MPI_Gather(buf_loc, nx * ny, MPI_REAL, buf_ful, nx * ny, MPI_REA
L, 0, MPI_COMM_WORLD, err)

  実装済み部分（省略）
```

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，より簡単に実装できます
  - 「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

より高度な内容に興味のある方へ

- 以下の講習会資料を参考にしてください
  https://www.cc.u-tokyo.ac.jp/events/lectures/
- 例えば
  - 「MPI上級編」
    https://www.cc.u-tokyo.ac.jp/events/lectures/190/
  - 「Wisteria実践」
    https://www.cc.u-tokyo.ac.jp/events/lectures/199/
  - 「GPUプログラミング入門」
    https://www.cc.u-tokyo.ac.jp/events/lectures/188/
  - 「OpenACCとMPIによるマルチGPUプログラミング入門」
    https://www.cc.u-tokyo.ac.jp/events/lectures/195/

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，5/26（金） 9:00まで使えます
  - キュー名はlecture-oです
    （tutorial-oは4/26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12ノードまで