計算科学入門 第7回
MPIによる並列計算

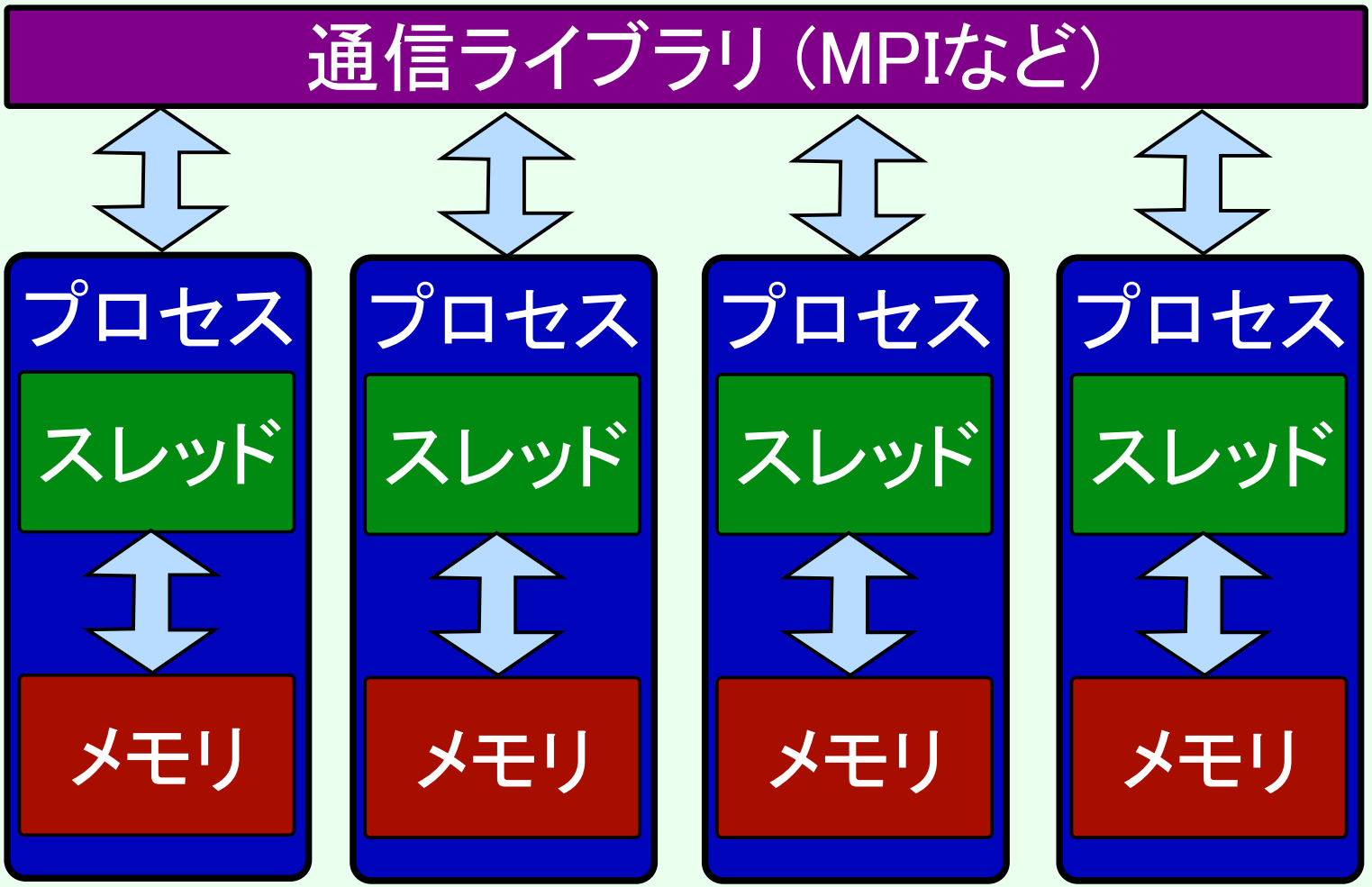
京都大学 大学院情報学研究科 数理工学専攻/高度情報教育基盤コア準備室

關戸 啓人

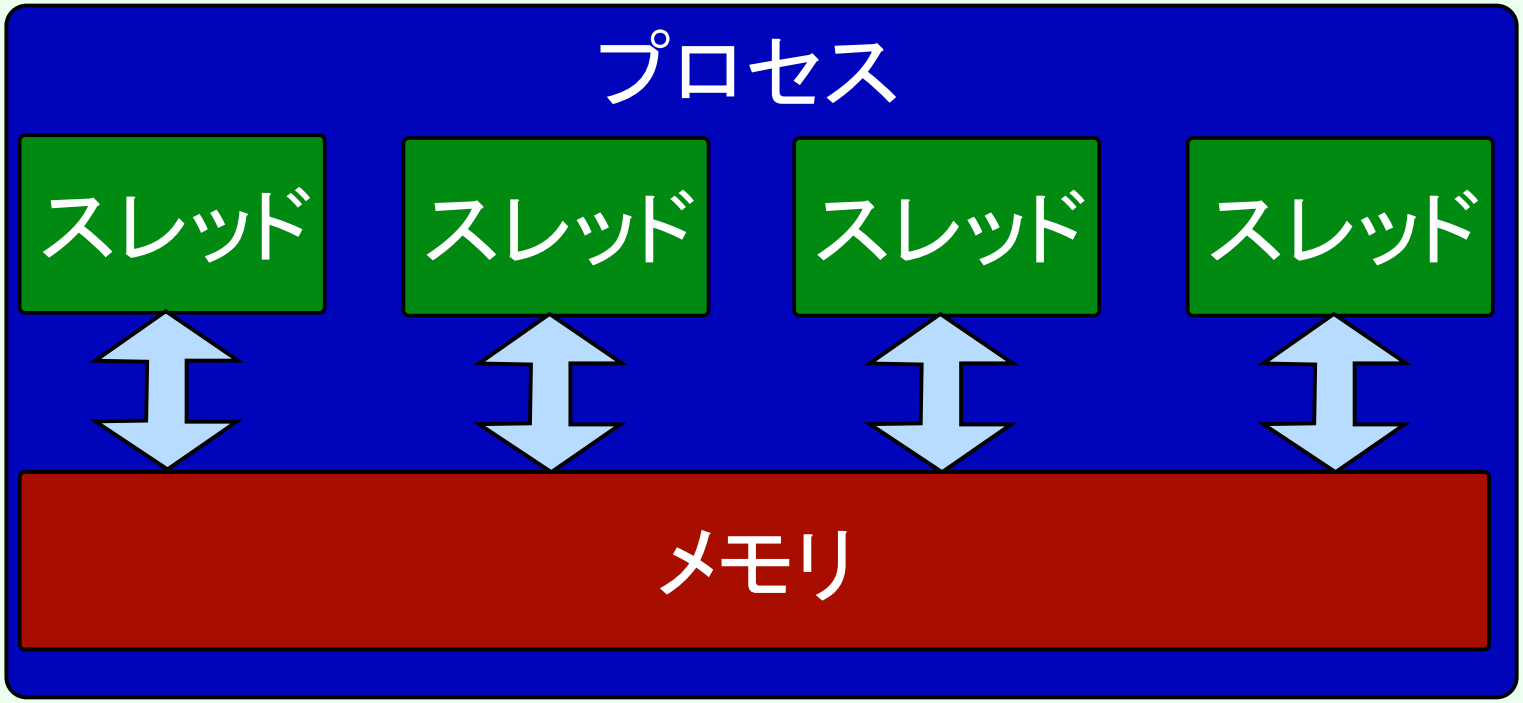
MPIとは (概要)

- MPIとはMessage Passing Interfaceの略で，プロセス間のメッセージ交換ライブラリの規格．
  - お互いにメモリに直接触れないプロセスが明示的にデータを他のプロセスに渡す，もしくは，受け取る．
  - MPIの機能は関数として提供される．
  - コンパイル時にMPIのライブラリにリンクさせる．実際には，リンクオプションを含んだコンパイラのラッパーが提供される．
- OpenMPは何だったか
  - 標準化された，コンパイラの拡張機能．
  - データの受け渡しは，メモリに直接書き込み，それを読み込む．

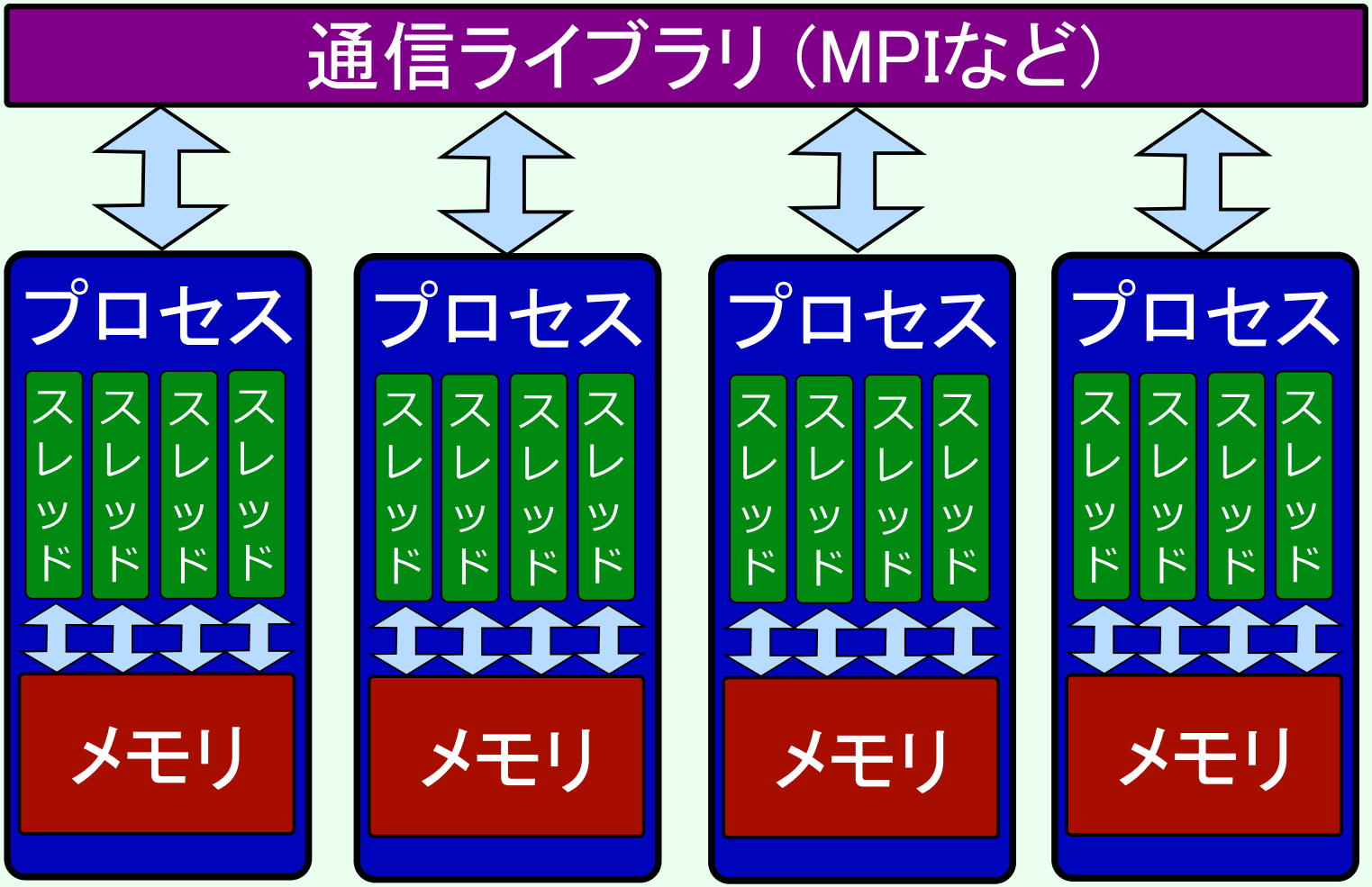
プロセス並列 (MPI)


通信ライブラリ（MPIなど）
プロセス
スレッド
メモリ
プロセス
スレッド
メモリ
プロセス
スレッド
メモリ
プロセス
スレッド
メモリ

スレッド並列 (OpenMP)



ハイブリッド並列 (MPI + OpenMP)



MPIとは (基礎的な諸注意1)

- MPIでは1つのソースコードを書き，それを複数のプロセスで実行する．
  - プロセスごとに異なった処理を行う場合は，自分のプロセス番号を調べて，それを用いる．
  - OpenMPと違い，1行pragmaを書けば自動的に並列化してくれるなどということはない．単純な`for`ループの並列化も，自分のプロセス番号や，全体のプロセス数を用いて，各プロセスが担当する範囲を指定しなければならない．
  - データ転送も明示的に書かなければならなく，OpenMPと比べて，ソースコードが長く，複雑になる傾向がある．

MPIとは (基礎的な諸注意2)

- 分散メモリ（メモリは共有ではない）
  - 例えば，OpenMPでは，変数`int n`を宣言したら，全スレッドが同じメモリ領域に`int n`を確保する．あるスレッドが`n`の値を書き換えれば，別のスレッドの変数`n`も書き換えられる．
  - 一方，MPIでは，変数`int n`を宣言したら，それぞれのプロセスが別々のメモリ領域に`int n`を確保する．あるプロセスが`n`の値を書き換えても，別のスレッドの変数`n`には影響しない．
  - 少し複雑なことをやろうと思ったときは，MPIの方が気を使うべきことが少ないと思われる．

MPIとは (歴史)

- MPI以前（～1980年代）
  - 並列計算機ベンダーが各々並列言語もしくは通信ライブラリを作成
- MPI-1（1994年6月）
  - 一対一通信
  - 集団通信
  - （派生データ型，コミュニケータ，… ）
- MPI-2（1997年7月）
- MPI-3（2012年9月）

参考：MPIの規格書は`http://www.mpi-forum.org/docs/docs.html`から閲覧可能

実行方法 (Intel MPIの場合)

- 京大のスパコン（laurel）では，C言語，C++，Fortranで利用可能
  - コンパイル方法（プログラムが`code.***`で実行ファイルとして`hoge`を作る場合）
    - Cの場合: `mpiicc -o hoge code.c`
    - C++の場合: `mpiicpc -o hoge code.cpp`
    - Fortranの場合: `mpiifort -o hoge code.f`（または`code.f90`）
  - 実行方法（8プロセスで実行する場合）
    - `mpiexec.hydra -n 8 ./hoge`

参考：`http://web.kudpc.kyoto-u.ac.jp/manual/ja/library/intelmpi`

実行方法 (OpenMPIの場合)

- 以下のどれかのmoduleをloadする
  - `module load openmpi/1.6_intel-12.1`
  - `module load pgi` と `module load openmpi/1.6_pgi-12.3`
  - `module load openmpi/1.6_gnu-4.4.6`
- コンパイル
  - **Cの場合:** `mpicc -o hoge code.c`
  - **C++の場合:** `mpic++ -o hoge code.cpp`
  - **Fortran 77の場合:** `mpif77 -o hoge code.f`
  - **Fortran 90の場合:** `mpif90 -o hoge code.f90`
- 実行方法（8プロセスで実行する場合）
  - `mpiexec -n 8 ./hoge`

参考：`http://web.kudpc.kyoto-u.ac.jp/manual/ja/library/openmpi`

実行方法 (バッチ処理，Intel MPIの場合)

- 実行時間を正確に測りたい場合はバッチ処理を行う（ノードを独占的に使用して実行）

```
#!/bin/bash
#QSUB -q eb
#QSUB -W 0:30
#QSUB -A p=16:t=1:c=1:m=3840M
set -x
mpiexec.hydra ./hoge
```

のようなファイル（job.txtとする）を作成する．

- ebというキューに実行時間制限0時間30分でジョブを投入するという意味
- p=16プロセスを使う．t=1は1プロセスあたりのスレッド数．c=1は1プロセスあたりのコア数で基本的にtと同じ．m=3840Mは1プロセスあたりのメモリ使用可能上限．
  - qsub < job.txt：ジョブを投入
  - qjobs：ジョブの状態確認
  - qkill JOBID：JOBIDのジョブを停止
  - 実行が終わると結果などが書かれたファイルができる

参考:https://web.kudpc.kyoto-u.ac.jp/manual/ja/run/batchjob/systembc

C言語とFortranの違い

これ以降，主に，C言語でのMPIについて解説する．違いはあまり無いが，注意が必要と思われるときはFortranについても述べる．両方述べるときは，C言語は赤，Fortranは青で書く．主な違いは以下の通り．

- インクルードするヘッダファイル
  - C言語では，`mpi.h`
  - Fortranでは，`mpif.h`
- 関数の引数
  - Fortranでは，引数の最後にエラーコードの戻り値（`ierr`）を指定する事が多い．C言語では関数の戻り値で取得．
- `MPI_INIT`の引数は違う（後述）

C++では，時々関数の引数や戻り値が異なるが本講義では説明しない．

高速なプログラムを書くために

- 通信にも時間がかかる
- 高速化のためのテクニック
  - 通信回数を減らす
  - 通信するデータの容量を減らす
  - 非ブロック通信を使い，通信しながら計算する
  - 使えるところでは一対一通信ではなく集団通信を使う
  - 不連続なデータの通信にはデータ型を作る

!

必要最小限の関数

- 本当に必要最小限の関数
  - MPI_INIT：MPI環境の開始
  - MPI_FINALIZE：MPI環境の終了
  - MPI_COMM_SIZE：全プロセス数を調べる
  - MPI_COMM_RANK：自分のプロセス番号を調べる
- 一対一通信の必要最小限の関数
  - MPI_SEND：データを送る
  - MPI_RECV：データを受け取る

コミュニケータ

- コミュニケータは，いくつかのプロセスからなるグループである
  - 集団通信では，とあるコミュニケータに属すプロセス間でデータのやり取りをすることができる
  - 最初は，全てのプロセスは`MPI_COMM_WORLD`というコミュニケータに属す
- この授業では，使用するコミュニケータは，`MPI_COMM_WORLD`だけで十分事足りる．

MPI_INIT

MPI_INITはちょうど1度だけ呼び出されなければならない．

また，MPI_INITを呼び出す前には，Cではmpi.h，Fortranではmpif.hがincludeされていなければならない．

- int MPI_Init(int *argc, char ***argv)
  - 引数は，それぞれmain関数の引数へのポインタ．
  - 戻り値（int）はエラーコードが返される．
- MPI_INIT(IERROR)
  - INTEGER IERRORには，エラーコードが代入される．

MPI_INITが呼び出される前はargc，argvは全てのプロセスで正しくセットされているとは限らない．

MPI_FINALIZE

MPI_FINALIZEは（MPI_INITを呼び出した後で）プログラムが終了する前に1度だけ呼び出されなければならない．また，ほぼ全てのMPIの関数はMPI_INITとMPI_FINALIZEとの間でしか使用することはできない．

- `int MPI_Finalize(void)`
- `MPI_FINALIZE(IERROR)`

例 ex1.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   puts("Hi");
06   return 0;
07 }
```

実行結果

Abort（処理系によってはHiが4行表示される）

例 ex2.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   MPI_Init(&argc, &argv);
06   puts("Hi");
07   MPI_Finalize();
08   return 0;
09 }
```

実行結果

```
Hi
Hi
Hi
Hi
```

例 ex3.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   puts("Hi");
06   MPI_Finalize();
07   return 0;
08 }
```

実行結果

```
Abort
```

例 ex4.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   MPI_Init(&argc, &argv);
06   puts("Hi");
07   return 0;
08 }
```

実行結果

```
Abort
```

例 ex5.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   MPI_Init(&argc, &argv);
06   puts("Hi");
07   MPI_Finalize();
08   MPI_Init(&argc, &argv);
09   puts("Hi");
10   MPI_Finalize();
11   return 0;
12 }
```

実行結果

```
Abort
```

MPI_COMM_SIZE

MPI_COMM_SIZEはコミュニケータに属するプロセス数を求める．コミュニケータにMPI_COMM_WORLDを指定すれば，全プロセス数を求めることができる．

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - `MPI_Comm comm`は指定するコミュニケータ
  - `size`にプロセス数が代入される
- `MPI_COMM_SIZE(COMM, SIZE, IERROR)`
  - `INTEGER COMM`：Fortranでは`INTEGER`．

MPI_COMM_RANK

MPI_COMM_RANKは自分のプロセスの指定したコミュニケータ内での番号を求める．最も小さい番号は0から始まり，連続した整数で記述される．

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - `MPI_Comm comm`は指定するコミュニケータ
  - `rank`にプロセス番号が代入される
- `MPI_COMM_RANK(COMM, RANK, IERROR)`

例 ex6.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   int my_rank, num_proc;
06
07   MPI_Init(&argc, &argv);
08   MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
09   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
10
11   printf("Hi, my rank = %d, and size = %d\n", my_rank, num_proc);
12
13   MPI_Finalize();
14   return 0;
15 }
```

実行結果

```
Hi, my rank = 1, and size = 4
Hi, my rank = 2, and size = 4
Hi, my rank = 3, and size = 4
Hi, my rank = 0, and size = 4
```

例 ex6.f (4プロセスで実行)

```
01        include "mpif.h"
02        INTEGER ierr, my_rank, num_proc
03        call MPI_INIT(ierr)
04        call MPI_COMM_SIZE(MPI_COMM_WORLD, num_proc, ierr)
05        call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
06        write(*,*) 'Hi, my rank = ', my_rank, ', and size = ', num_proc
07        call MPI_FINALIZE(ierr)
08        end
```

実行結果

```
 Hi, my rank =            3 , and size =            4
 Hi, my rank =            0 , and size =            4
 Hi, my rank =            1 , and size =            4
 Hi, my rank =            2 , and size =            4
```

例 ex6.f90 (4プロセスで実行)

```
01 program main
02   include "mpif.h"
03   INTEGER my_rank, num_proc, ierr
04   call MPI_INIT(ierr)
05   call MPI_COMM_SIZE(MPI_COMM_WORLD, num_proc, ierr)
06   call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
07   print *, 'Hi, my rank = ', my_rank, ', and size = ', num_proc
08   call MPI_FINALIZE(ierr)
09 end program main
```

実行結果

```
 Hi, my rank =            3 , and size =            4
 Hi, my rank =            1 , and size =            4
 Hi, my rank =            0 , and size =            4
 Hi, my rank =            2 , and size =            4
```

これらの例 (ex6.*) がすべての基本．出力の順番は実行の度に変わる．

MPI_SEND

MPI_SENDは一対一通信でデータを送信する関数である．送信先でデータが受信されるまで，この関数は終わらないことがある．

- `int MPI_Send(void *buf, int count,`
  `MPI_Datatype type, int dest, int tag, MPI_Comm comm)`
- `MPI_SEND(BUF, COUNT, TYPE, DEST, TAG, COMM, IERROR)`
  - `INTEGER TYPE`はFortranでは整数
  - `void *buf`は送信するデータの最初のアドレスを指定する
  - `int count`は送信するデータの個数を指定する
  - `MPI_Datatype type`は送信するデータの型（後述）を指定する
  - `int dest`は受け取るプロセスの番号を指定する
  - `int tag`はデータを識別するための番号．受信の時（`MPI_RECV`）に送信時につけた`tag`の番号を指定する．
  - `MPI_Comm comm`はコミュニケータを指定する．

MPI_RECV

MPI_RECVは一対一通信でデータを受信する関数である．データを受信するまで，この関数は終わらない．RECVはRecieveの意味．

- `int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `MPI_RECV(BUF, COUNT, TYPE, SOURCE, TAG, COMM, STATUS, IERROR)`
  - `INTEGER STATUS(MPI_STATUS_SIZE)`はFortranでは整数の配列
  - `void *buf`は受信するデータを格納する最初のアドレスを指定する
  - `int count`は受信するデータの個数を指定する
  - `MPI_Datatype type`は送信するデータの型（後述）を指定する
  - `int source`は送信するプロセスの番号を指定する
  - `int tag`はデータを識別するための番号．送信の時（MPI_SEND等）に指定したtagと同じ番号を指定する．
  - `MPI_Comm comm`はコミュニケータを指定する．
  - `MPI_Status *status`は送信に関する情報が代入される．

*bufについての補足

C言語のポインタ，配列に詳しくない人は以下を参考にせよ．

1つの変数`int a`を送信，受信する場合

```
MPI_Send(&a, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

配列`int a[10]`の最初の5要素（`a[0]`〜`a[4]`）を送信，受信する場合

```
MPI_Send(a, 5, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

または，

```
MPI_Send(&a[0], 5, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

配列`int a[10]`の6要素（`a[3]`〜`a[8]`）を送信，受信する場合

```
MPI_Send(a+3, 6, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

または，

```
MPI_Send(&a[3], 6, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

MPI Datatypes (C言語)

MPI_SEND，MPI_RECV等で用いるMPI_Datatype typeとC言語の型との対応関係は以下の通り．

- char：MPI_CHAR
- short：MPI_SHORT
- int：MPI_INT
- long：MPI_LONG
- float：MPI_FLOAT
- double：MPI_DOUBLE
- unsigned char：MPI_UNSIGNED_CHAR
- unsigned short：MPI_UNSIGNED_SHORT
- unsigned：MPI_UNSIGNED
- unsigned long：MPI_UNSIGNED_LONG
- long double：MPI_LONG_DOUBLE
- 等

MPI Datatypes (Fortran)

MPI_SEND，MPI_RECV等で用いるINTEGER TYPEとFortranの型との対応関係は以下の通り．

- INTEGER：MPI_INTEGER
- REAL：MPI_REAL
- REAL*8：MPI_REAL8
- DOUBLE PRECISION：MPI_DOUBLE_PRECISION
- COMPLEX：MPI_COMPLEX
- LOGICAL：MPI_LOGICAL
- CHARACTER：MPI_CHARACTER
- 等

（補足）MPI_Status

MPI_RECVで，MPI_Status *statusには，送信元のrankや，tagなどが代入される．MPI_WAIT_ALLなどではエラー情報も代入されることもあるが，通常は関数がエラー情報を返すので，MPI_Status *statusのエラー情報は更新されないことも多い．そのような情報が必要ない場合は，MPI_Status *statusにMPI_STATUS_IGNOREを指定すれば良い．

MPI_SENDで指定する送信データの個数int countは，MPI_RECVで指定する受信データの個数int countより小さくても良い．実際に受信したデータの個数を取得するには，MPI_GET_COUNTを用いる．これにも，MPI_Status *statusが必要．

（補足）MPI_GET_COUNT

MPI_GET_COUNTは受信したデータの個数（バイト数ではない）を取得する関数である．

- int MPI_Get_count(MPI_Status *status, MPI_Datatype type,
  int *count)
- MPI_RECV(STATUS, TYPE, COUNT, IERROR)
  - MPI_Status *statusは受信の時に代入されたステータスのポインタを指定する．
  - int countには受信したデータの個数が代入される．

（補足）MPI_GET_COUNTの例 ex12.c (2プロセス)

```
13    if(my_rank == 0){
14      char send[] = "How are you?";
15      MPI_Send(send, strlen(send)+1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
16    } else if(my_rank == 1){
17      char recv[100]; int cnt;
18      MPI_Recv(recv, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
19      printf("I recieved: %s\n", recv);
20      MPI_Get_count(&status, MPI_CHAR, &cnt);
21      printf("length = %d\n",cnt);
22    }
```

実行結果

```
I recieved: How are you?
length = 13
```

（補足）MPI_GET_COUNTの例 ex13.c (2プロセス)

```
13   if(my_rank == 0){
14     char send[] = "I send looooooooooooooooooong message. How are you?";
15     MPI_Send(send, strlen(send)+1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
16   } else if(my_rank == 1){
17     char recv[100]; int cnt;
18     MPI_Recv(recv, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
19     printf("I recieved: %s\n", recv);
20     MPI_Get_count(&status, MPI_CHAR, &cnt);
21     printf("length = %d\n",cnt);
22   }
```

実行結果

```
Abort
```

例 ex7.c (一部抜粋，2プロセスで実行)

```
04 #define N 3
08   int i, send[N], recv[N], source, dest;
09   MPI_Status status;
14
15   send[0] = 1;
16   for(i=1;i<N;i++) send[i] = (send[i-1] * 3 + my_rank) % 10007;
17
18   source = dest = 1 - my_rank;
19   MPI_Send(send, N, MPI_INT, dest, 0, MPI_COMM_WORLD);
20   MPI_Recv(recv, N, MPI_INT, source, 0, MPI_COMM_WORLD, &status);
```

実行結果
正常終了（出力は無し）

例 ex8.c (一部抜粋，2プロセスで実行)

```
04 #define N 3333
08   int i, send[N], recv[N], source, dest;
09   MPI_Status status;
14
15   send[0] = 1;
16   for(i=1;i<N;i++) send[i] = (send[i-1] * 3 + my_rank) % 10007;
17
18   source = dest = 1 - my_rank;
19   MPI_Send(send, N, MPI_INT, dest, 0, MPI_COMM_WORLD);
20   MPI_Recv(recv, N, MPI_INT, source, 0, MPI_COMM_WORLD, &status);
```

実行結果
終了しない.（このコードを実行するときは`Ctrl-C`などで強制終了させよ）

デッドロック

先程の例では，

- プロセス0は，
  - プロセス1にデータを送信してから，
  - プロセス1からデータを受信する．
- プロセス1は，
  - プロセス0にデータを送信してから，
  - プロセス0からデータを受信する．

この場合，データを送信するとき，互いが互いの受信待ちをする場合がある．

このように，全てのプロセスが，他のプロセスを待つ状態に入り，プログラムが進まなくなることを**デッドロック**と言う．

デッドロック (解決策の例)

デッドロックを回避するには，例えば，

- プロセス0は，
  - プロセス1にデータを送信してから，
  - プロセス1からデータを受信する．
- プロセス1は，
  - プロセス0からデータを受信してから．
  - プロセス0にデータを送信する

とすれば，デッドロックに陥らない．または，例えば，後に紹介する

MPI_SENDRECVや非ブロック通信を使っても良い．

MPI_SENDRECV

MPI_SENDRECVは一対一通信でデータを送るのと受信するのと両方を1回のMPI関数の呼び出しで行う関数である．

- ```
  int MPI_Sendrecv(void *sendbuf, int sendcount,
          MPI_Datatype sendtype, int dest, int sendtag,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,
          int source, int recvtag, MPI_Comm comm,
          MPI_Status *status)
  ```
- ```
  MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,
          RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM,
          STATUS, IERROR)
  ```

送ったデータの領域を受け取ったデータで上書きするMPI_SENDRECV_REPLACEもある．

- ```
  int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype type,
          int dest, int sendtag, int source,
          int recvtag, MPI_Comm comm, MPI_Status *status)
  ```
  ```

例 ex23.c (4プロセスで実行)

```
06    int my_val, recv_val;
11
12    my_val = my_rank * my_rank;
13    MPI_Sendrecv(&my_val, 1, MPI_INT, (my_rank+1)%num_proc, 0,
14                 &recv_val, 1, MPI_INT, (my_rank+num_proc-1)%num_proc, 0,
15                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
16
17    printf("my_rank %d, my_val %d, recv_val %d\n", my_rank, my_val, recv_val);
```

実行結果

```
my_rank 3, my_val 9, recv_val 4
my_rank 2, my_val 4, recv_val 1
my_rank 0, my_val 0, recv_val 9
my_rank 1, my_val 1, recv_val 0
```

非ブロック通信

非ブロック通信で送受信する関数は（MPI_ISEND，MPI_IRECV等）データを送信・受信できたかどうかに関わらず，すぐ終わる．データ通信の間に計算を進めることができ，高速な計算が可能だが，コードは複雑になりやすい．まだ送信していないデータを書き換えない，まだ受信していないデータを使わないように注意．

- 送受信
  - MPI_ISEND：一対一の非ブロック通信で，データの送信
  - MPI_IRECV：一対一の非ブロック通信で，データの受信
- 送受信を終了待ちするための関数
  - MPI_WAIT：指定した非ブロック通信が終わるまで待つ

MPI_ISEND

MPI_ISENDは一対一の非ブロック通信でデータを送信する関数である．送信先でデータが受信されかどうかに関わらず，この関数は終わる．ISENDのIはImmediateの意味．

- `int MPI_Isend(void *buf, int count, MPI_Datatype type,`
  `int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `MPI_ISEND(BUF, COUNT, TYPE, DEST, TAG, COMM, REQUEST, IERROR)`
  - `INTEGER REQUEST`はFortranでは整数
  - `MPI_Request *request`はこの送信（一般的にはこの関数呼び出し）を識別するためのデータが代入される．

MPI_IRECV

MPI_IRECVは一対一の非ブロック通信でデータを送信する関数である．データを受信しなくても，この関数は終わる．

- int MPI_Irecv(void *buf, int count, MPI_Datatype type,
  int source, int tag, MPI_Comm comm, MPI_Request *request)
- MPI_IRECV(BUF, COUNT, TYPE, SOURCE, TAG, COMM, REQUEST, IERROR)

MPI_WAIT

MPI_WAIT は指定した非ブロック通信が終了するまで待つ関数である．

- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
- `MPI_WAIT(REQUEST, STATUS, IERROR)`
  - `MPI_Request *request` は待ちたい送受信の際に得られた値を指定する．
  - `MPI_Status *status` はステータスが代入される．

例 ex9.c (一部抜粋，2プロセスで実行)

```
06    int i, source=0, dest=1, tag;
07    double a = 500.0, b = 0.05, c = 0.0;
08    MPI_Request request;
14    if(my_rank == 0){
15      tag = 1;
16      MPI_Isend(&a, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &request);
17      tag = 0;
18      MPI_Isend(&b, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &request);
19    } else if(my_rank == 1){
20      tag = 0;
21      MPI_Irecv(&c, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &request);
22      printf("recieved %f\n", c);
23      tag = 1;
24      MPI_Irecv(&c, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &request);
25      printf("recieved %f\n", c);
26    }
```

実行結果

```
recieved 0.000000
recieved 0.000000
```

例 ex10.c (一部抜粋，2プロセスで実行)

```
06    int i, source=0, dest=1, tag;
07    double a = 500.0, b = 0.05, c = 0.0;
08    MPI_Request request;
09    MPI_Status status;
15    if(my_rank == 0){
16      tag = 1;
17      MPI_Isend(&a, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &request);
18      tag = 0;
19      MPI_Isend(&b, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &request);
20    } else if(my_rank == 1){
21      tag = 0;
22      MPI_Irecv(&c, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &request);
23      MPI_Wait(&request, &status);
24      printf("recieved %f\n", c);
25      tag = 1;
26      MPI_Irecv(&c, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &request);
27      MPI_Wait(&request, &status);
28      printf("recieved %f\n", c);
```

実行結果

```
recieved 0.050000
recieved 500.000000
```

集団通信 (1)

集団通信関数は，あるコミュニケータに属す全てのプロセスが係る通信を1回の呼び出しで行うことのできる関数である．処理ごとに最適化されており，一対一通信で愚直に書くより速いことが多い．また，一対一通信で書くよりもデッドロックなども起きにくい．

集団通信では，基本的に，通信に参加する全てのプロセスが関数を呼び出す．各集団通信はIのついた（MPI_IBCAST等）非ブロック通信もある．

- 一対多通信
  - MPI_BCAST：あるrankの持つデータを全てのプロセスに送受信する
  - MPI_SCATTER：あるrankの持つデータを，等分して，各々のプロセスに配分する

集団通信 (2)

- 多対一通信
  - MPI_GATHER：MPI_SCATTERの逆．各々のプロセスのデータを連結して1つのプロセスに集める．
  - MPI_REDUCE:MPI_GATHERと演算を組み合わせたもの．例えば，各々のプロセスのデータの和，積，最大値，最小値などを求め，1つのプロセスに渡す．
- 多対多通信
  - MPI_ALLTOALL：各プロセスの配列を各プロセスに配分する
  - MPI_ALLGATHER：MPI_GATHERの結果を全プロセスに渡す．
  - MPI_ALLREDUCE：MPI_REDUCEの結果を全プロセスに渡す．

MPI_BCAST

MPI_BCASTは集団通信で，あるrankの持つデータを全てのプロセスに送受信する関数である．送受信に関わる全てのプロセスがMPI_BCASTを呼びださなければならなく，root，commは一致しなければならない．count，typeも普通は一致する．BCASTはBroadcastの意味．

- `int MPI_Bcast(void *buf, int count, MPI_Datatype type, int root, MPI_Comm comm)`
- `MPI_BCAST(BUF, COUNT, TYPE, ROOT, COMM, IERROR)`
  - `void *buf`は，送信する，もしくは，受信する，データの最初のアドレス．
  - `int root`は，データの送信主のプロセス番号を指定する．

MPI_BCAST

プロセス0	a_0	a_1	a_2	a_3	…
プロセス1	b_0	b_1	b_2	b_3	…
プロセス2	c_0	c_1	c_2	c_3	…
プロセス3	d_0	d_1	d_2	d_3	…

↓

プロセス0	a_0	a_1	a_2	a_3	…
プロセス1	a_0	a_1	b_2	b_3	…
プロセス2	a_0	a_1	c_2	c_3	…
プロセス3	a_0	a_1	d_2	d_3	…

例 ex15.c (一部抜粋，4プロセスで実行)

```
06    int i, arr[10], sum;
12    if(my_rank==0){
13      for(i=0;i<10;i++) arr[i] = i;
14    }
16    MPI_Bcast(arr, 10, MPI_INT, 0, MPI_COMM_WORLD);
18    sum = 0;
19    for(i=0;i<10;i++) sum += arr[i];
21    printf("my_rank = %d, sum = %d\n", my_rank, sum);
```

実行結果

```
my_rank = 0, sum = 45
my_rank = 3, sum = 45
my_rank = 1, sum = 45
my_rank = 2, sum = 45
```

MPI_SCATTER

MPI_SCATTERはあるrankの持つデータを，等分して，各々のプロセスに配分する関数である．

- int MPI_Scatter(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm Comm)
- MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
  RECVTYPE, ROOT, COMM, IERROR)
  - void *sendbufは送信するデータの最初のアドレスを指定する．int rootで指定されたプロセス以外では何を指定しても良い．
  - int sendcountは，1プロセスあたりに送信するデータの数を指定する．通常，int recvcountと同じになる．
  - void *recvbufは受信するデータの最初のアドレスを指定する．rankがrootのプロセスはvoid *sendbufと領域が重なってはいけない．rankがrootのプロセスで，自分には送受信したくない場合はMPI_IN_PLACE（MPI-2の機能）を指定する．
  - int rootは，データの送信主のプロセス番号を指定する．

MPI_SCATTER

プロセス0 (送信)	z_0	z_1	z_2	z_3	…
プロセス0 (受信)	a_0	a_1	a_2	a_3	…
プロセス1	b_0	b_1	b_2	b_3	…
プロセス2	c_0	c_1	c_2	c_3	…
プロセス3	d_0	d_1	d_2	d_3	…

↓

プロセス0 (送信)	z_0	z_1	z_2	z_3	…
プロセス0 (受信)	z_0	a_1	a_2	a_3	…
プロセス1	z_1	a_1	b_2	b_3	…
プロセス2	z_2	a_1	c_2	c_3	…
プロセス3	z_3	a_1	d_2	d_3	…

例 ex16.c (一部抜粋，4プロセスで実行)

```
06    int i, arr[12], myarr[3], sum;
12    if(my_rank==0){
13      for(i=0;i<3*num_proc;i++) arr[i] = i;
14    }
16    MPI_Scatter(arr, 3, MPI_INT, myarr, 3, MPI_INT, 0, MPI_COMM_WORLD);
18    sum = 0;
19    for(i=0;i<3;i++) sum += myarr[i];
21    printf("my_rank = %d, sum = %d\n", my_rank, sum);
```

実行結果

```
my_rank = 3, sum = 30
my_rank = 0, sum = 3
my_rank = 1, sum = 12
my_rank = 2, sum = 21
```

例 ex17.c (一部抜粋，4プロセスで実行)

```
06    int i, arr[12], sum;
12    if(my_rank==0){
13      for(i=0;i<3*num_proc;i++) arr[i] = i;
14    }
15
16    if(my_rank == 0)
17      MPI_Scatter(arr, 3, MPI_INT, MPI_IN_PLACE, 3, MPI_INT, 0,
                    MPI_COMM_WORLD);
18    else
19      MPI_Scatter(arr, 3, MPI_INT, arr, 3, MPI_INT, 0, MPI_COMM_WORLD);
20
21    sum = 0;
22    for(i=0;i<3;i++) sum += arr[i];
23
24    printf("my_rank = %d, sum = %d\n", my_rank, sum);
```

実行結果

```
my_rank = 2, sum = 21
my_rank = 3, sum = 30
my_rank = 0, sum = 3
my_rank = 1, sum = 12
```

MPI_GATHER

MPI_GATHERは各プロセスの持つ同じ長さのデータを結合して，あるプロセスに渡す関数である．MPI_SCATTERと逆の動作をする．

- int MPI_Gather(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm Comm)
- MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
  RECVTYPE, ROOT, COMM, IERROR)
  - void *recvbufは受信するデータの最初のアドレスを指定する．int rootで指定されたプロセス以外では何を指定しても良い．
  - int recvcountは，1プロセスあたりの送信するデータの数を指定する．通常，int sendcountと同じになる．
  - void *sendbufはMPI_IN_PLACEを指定可能．
  - int rootは，データを受信するプロセス番号を指定する．

MPI_GATHER

プロセス0 (受信)	z_0	z_1	z_2	z_3	…
プロセス0 (送信)	a_0	a_1	a_2	a_3	…
プロセス1	b_0	b_1	b_2	b_3	…
プロセス2	c_0	c_1	c_2	c_3	…
プロセス3	d_0	d_1	d_2	d_3	…

↓

プロセス0 (受信)	a_0	b_0	c_0	d_0	…
プロセス0 (送信)	a_0	a_1	a_2	a_3	…
プロセス1	b_0	b_1	b_2	b_3	…
プロセス2	c_0	c_1	c_2	c_3	…
プロセス3	d_0	d_1	d_2	d_3	…

MPI_REDUCE

MPI_REDUCEは，各プロセスの持つ同じ長さのデータを何らかの演算をして，その結果をあるプロセスに渡す関数である．

- `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)`
- `MPI_REDUCE(SENDBUF, RECVBUF, COUNT, TYPE, OP, ROOT, COMM, IERROR)`
  - `INTEGER OP`はFortranでは整数である．
  - `void *sendbuf`は`MPI_IN_PLACE`を指定可能．
  - `void *recvbuf`は受信するデータの最初のアドレスを指定する．`int root`で指定されたプロセス以外では何を指定しても良い．
  - `MPI_Op op`は演算の種類を指定する（後述）
  - `int root`は，データを受信するプロセス番号を指定する．

MPI_REDUCE

プロセス0 (受信)	z_0	z_1	z_2	z_3	…
プロセス0 (送信)	1	5	a_2	a_3	…
プロセス1	2	6	b_2	b_3	…
プロセス2	3	7	c_2	c_3	…
プロセス3	4	8	d_2	d_3	…

↓

プロセス0 (受信)	10	26	z_2	z_3	…
プロセス0 (送信)	1	5	a_2	a_3	…
プロセス1	2	6	b_2	b_3	…
プロセス2	3	7	c_2	c_3	…
プロセス3	4	8	d_2	d_3	…

MPIオペランド

MPI_REDUCE，MPI_ALLREDUCEで用いることのできる演算MPI_Op op.

- MPI_MAX：最大値
- MPI_MIN：最小値
- MPI_SUM：和
- MPI_PROD：積
- MPI_LAND：論理積
- MPI_BAND：ビットごとに論理積を取る演算
- MPI_LOR：論理和
- MPI_BOR：ビットごとに論理和を取る演算
- MPI_LXOR：排他的論理和
- MPI_BXOR：ビットごとに排他的論理和を取る演算
- MPI_MAXLOC：最大値とその位置
- MPI_MINLOC：最小値とその位置

例 ex11.c (4プロセスで実行)

```
01 #include<stdio.h>
02 #include<mpi.h>
03
04 int main(int argc, char **argv){
05   int my_rank, num_proc;
06   int i, my[2], sum[2];
07
08   MPI_Init(&argc, &argv);
09   MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
10   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
11
12   for(i=0;i<2;i++) my[i] = i*4 + my_rank + 1;
13   MPI_Reduce(my, sum, 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
14   if(my_rank == 0) printf("%d %d\n", sum[0], sum[1]);
15
16   MPI_Finalize();
17   return 0;
18 }
```

実行結果

```
10 26
```

MPI_ALLTOALL

MPI_ALLTOALL は各プロセス持つデータを，一定数ごとに各プロセスに分配していき，分配されたデータを結合する関数である．次のページの例を参照．

- int MPI_Alltoall(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, MPI_Comm comm)
- MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
  RECVTYPE, COMM, IERROR)
  - void *sendbuf は送信するデータの最初のアドレスを指定する．
  - int sendcount は，各プロセスが，1プロセスあたりに送信するデータの数を指定する．通常，int recvcount と同じになる．
  - void *recvbuf は受信するデータの最初のアドレスを指定する．
  - int recvcount は，各プロセスが，1プロセスあたりから受信するデータの数を指定する．通常，int sendcount と同じになる．

MPI_ALLTOALL

プロセス0 (送信)	a_0	a_1	a_2	a_3	…
プロセス1 (送信)	b_0	b_1	b_2	b_3	…
プロセス2 (送信)	c_0	c_1	c_2	c_3	…
プロセス3 (送信)	d_0	d_1	d_2	d_3	…

↓

プロセス0 (受信)	a_0	b_0	c_0	d_0	…
プロセス1 (受信)	a_1	b_1	c_1	d_1	…
プロセス2 (受信)	a_2	b_2	c_2	d_2	…
プロセス3 (受信)	a_3	b_3	c_3	d_3	…

MPI_ALLGATHER

MPI_ALLGATHERはMPI_GATHERの結果を全てのプロセスに渡す関数である．引数はMPI_GATHERとの違いはrootが無くなっただけである．

- int MPI_Allgather(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, MPI_Comm Comm)
- MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
  RECVTYPE, COMM, IERROR)

MPI_ALLGATHER

プロセス0 (送信)	a_0	a_1	a_2	a_3	…
プロセス1 (送信)	b_0	b_1	b_2	b_3	…
プロセス2 (送信)	c_0	c_1	c_2	c_3	…
プロセス3 (送信)	d_0	d_1	d_2	d_3	…

↓

プロセス0 (受信)	a_0	b_0	c_0	d_0	…
プロセス1 (受信)	a_0	b_0	c_0	d_0	…
プロセス2 (受信)	a_0	b_0	c_0	d_0	…
プロセス3 (受信)	a_0	b_0	c_0	d_0	…

MPI_ALLREDUCE

MPI_ALLREDUCE はMPI_REDUCE の結果を全てのプロセスに渡す関数である．引数はMPI_REDUCE との違いはroot が無くなっただけである．

- int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, MPI_Comm comm)
- MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, TYPE, OP, COMM, IERROR)

MPI_ALLREDUCE

プロセス0 (送信)	1	5	a_2	a_3	...
プロセス1 (送信)	2	6	b_2	b_3	...
プロセス2 (送信)	3	7	c_2	c_3	...
プロセス3 (送信)	4	8	d_2	d_3	...

↓

プロセス0 (受信)	10	26	x_2	x_3	...
プロセス1 (受信)	10	26	y_2	y_3	...
プロセス2 (受信)	10	26	z_2	z_3	...
プロセス3 (受信)	10	26	u_2	u_3	...

補足

以降は今まで紹介しなかった関数などの紹介．

MPI_BARRIER

MPI_BARRIERは全プロセスが呼び出すまで待ち続ける関数である．

- `int MPI_Barrier(MPI_Comm comm)`
- `MPI_BARRIER(COMM)`

MPI_WAITALL

MPI_WAITALLは指定した複数の非ブロック通信が全て終了するまで待つ関数である．

- `int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)`
- `MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)`
  - `int count`は待ちたい非ブロック通信の数を指定する．
  - `MPI_Request *array_of_requests`は待ちたい送受信の際に得られた値の配列（要素数は`int count`）を指定する．
  - `MPI_Status *array_of_statuses`はステータスが代入される．必要ないなら，`MPI_STATUSES_IGNORE`を指定すれば良い．(`MPI_STATUS_IGNORE`ではない)

指定した複数の非ブロック通信の1つ以上が終了するまで待ち，終了した通信を1つ教えてくれる`MPI_WAITANY`，指定した複数の非ブロック通信の1つ以上が終了するまで待ち，終了した通信を全部教えてくれる`MPI_WAITSOME`もある．

MPI_TEST

MPI_TESTは指定した非ブロック通信が終了しているかどうかチェックする関数である．非ブロック版のMPI_WAITと思うことができる．MPI_TESTALL，MPI_TESTANY，MPI_TESTSOMEもある．

- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
- `MPI_MPI_TEST(REQUEST, FLAG, STATUS, IERROR)`
  - `LOGICAL FLAG`はFortranでは`LOGICAL`型．
  - `int flag`は指定した通信が終わっていれば`true`な値，終わっていなければ`false`な値（0）が代入される．

MPI_WTIME

MPI_WTIMEは，とある時点からの経過秒数を返す関数である．プログラムの実行中は，とある時点は変更しないことが保証されており，2回呼び出し，差を取れば実行にかかった時間を計測することができる．

- `double MPI_Wtime(void)`
- `DOUBLE PRECISION MPI_WTIME()`

例 ex14.c (4プロセスで実行)

```
07    double start_time, end_time;
08    static double arr[10000000]; /* around 80MB */
14    start_time = MPI_Wtime();
15
16    if(my_rank == 0){
17      MPI_Send(arr, 10000000, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
18    } else if(my_rank == 1){
19      MPI_Recv(arr, 10000000, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
20               MPI_STATUS_IGNORE);
21    } else if(my_rank == 2){
22      int i;
23      for(i=0;i<10000000;i++) arr[i] = 10000.0 / i;
24    }
25
26    end_time = MPI_Wtime();
27    printf("rank = %d, elapsed = %f = %f - %f\n",
28           my_rank, end_time-start_time, end_time, start_time);
29
```

例 ex14.c (4プロセスで実行)

実行結果

```
rank = 3, elapsed = 0.000000 = 1369794662.705628 - 1369794662.705628
rank = 0, elapsed = 0.044439 = 1369794662.750048 - 1369794662.705609
rank = 1, elapsed = 0.044445 = 1369794662.750048 - 1369794662.705603
rank = 2, elapsed = 0.062007 = 1369794662.767653 - 1369794662.705646
```

MPI_BSEND, MPI_SSEND, MPI_RSEND

MPI_SENDは，送信サイズなどに応じて，以下の3つの関数を使い分けている（使い分け方は処理系依存）．以下の関数は引数などは，MPI_SENDと同じ．受信はすべてMPI_RECVで良い．

- MPI_BSEND：受信側の準備ができていれば送信し，そうでなければ，送信バッファに転送データをコピーして戻る．Bは Buffered の意味．
- MPI_SSEND：データを送信したら戻る．Sは Synchronous の意味．
- MPI_RSEND：受信側の準備ができている場合のみ使用できる．Rは Ready の意味．

ex7.cの例ではMPI_BSEND，ex8.cの例ではMPI_SSENDが用いられたと思われる．

これらの非ブロック通信版はMPI_IBSEND，MPI_ISSEND，MPI_IRSEND．引数などはMPI_ISENDと同じ．

MPI_ANY_SOURCE，MPI_ANY_TAG

MPI_RECVでは，通常，sourceで指定したIDのプロセスがtagで指定したタグと同じtagで送ってきた場合のみ受信する．任意のプロセスから受信する場合はsourceにMPI_ANY_SOURCEを指定する．任意のタグを受信する場合はtagにMPI_ANY_TAGを指定する．

MPI_PROC_NULL

MPI_SENDにおいて，送信先destにMPI_PROC_NULLを指定すると何も起こらない．MPI_RECVにおいて，送信元sourceにMPI_PROC_NULLを指定すると何も起こらない．

派生データ型

基本型（MPI_INT，MPI_DOUBLE）などを組み合わせたり，不連続な領域を表すためのデータ型を作成することができる．

- MPI_TYPE_COMMIT：
  作ったデータ型を使用可能にする
- MPI_TYPE_CONTIGUOUS：
  同じデータ型を連続的に何回か続けた新しいデータ型を作る
- MPI_TYPE_VECTOR：
  同じデータ型を等間隔に飛ばし飛ばし（間に空白があって良い）で何回か繰り返す新しいデータ型を作る
- MPI_TYPE_INDEXED：
  同じデータ型を不等間隔に飛ばし飛ばしで何回か繰り返す新しいデータ型を作る
- MPI_TYPE_CREATE_SUBARRAY：
  n次元配列のある直方体の部分のみからなるデータ型を作る
- MPI_TYPE_CREATE_STRUCT：
  異なるデータ型を不等間隔に並べた新しいデータ型を作る

MPI_TYPE_COMMIT

MPI_TYPE_COMMITは作成したデータ型を実際に使用可能な状態にする．作成した同じデータ型を何回も使いまわす場合も，最初の使用の前に1度だけMPI_TYPE_COMMITを呼び出せば良い．ただし，複数のデータ型を使用した場合は，各データ型について呼びださなければいけない．

- `int MPI_Type_commit(MPI_Datatype *type)`
- `MPI_TYPE_COMMIT(TYPE, IERROR)`
  - `MPI_Datatype *type`は使用可能にするデータ型へのポインタ．
  - `TYPE`はFortranではINTEGER．

MPI_TYPE_VECTOR

MPI_TYPE_VECTORは同じデータ型oldtypeがblen個連続しているのを1塊とし，count個の塊からなる新しいデータ型を作りnewtypeに返す．隣り合う塊の最初はoldtypeがstride個分離れているとする．

- int MPI_Type_vector(int count, int blen, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
- MPI_TYPE_VECTOR(COUNT, BLEN, STRIDE, OLDTYPE, NEWTYPE, IERROR)
  - NEWTYPEはFortranではINTEGER．

例 ex18.c (2プロセスで実行)

```
06   int i, arr[30];
07   MPI_Datatype my_type;
13   if(my_rank==0){
14     for(i=0;i<30;i++) arr[i] = i;
15   } else if(my_rank == 1){
16     for(i=0;i<30;i++) arr[i] = -1;
17   }
18
19   MPI_Type_vector(4, 2, 3, MPI_INT, &my_type);
20   MPI_Type_commit(&my_type);
21
22   if(my_rank == 0)
23     MPI_Send(arr, 1, my_type, 1, 0, MPI_COMM_WORLD);
24   else if(my_rank == 1)
25     MPI_Recv(arr, 1, my_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26
27   if(my_rank == 1){
28     for(i=0;i<30;i++) printf("%2d%c", arr[i], i%10==9?'\n':' ');
29   }
```

例 ex18.c (2プロセスで実行)

実行結果

```
 0  1 -1  3  4 -1  6  7 -1  9
10 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

例 ex19.c (2プロセスで実行)

```
06   int i, arr[30];
07   MPI_Datatype my_type;
13   if(my_rank==0){
14     for(i=0;i<30;i++) arr[i] = i;
15   } else if(my_rank == 1){
16     for(i=0;i<30;i++) arr[i] = -1;
17   }
18
19   MPI_Type_vector(4, 2, 3, MPI_INT, &my_type);
20   MPI_Type_commit(&my_type);
21
22   if(my_rank == 0)
23     MPI_Send(arr, 2, my_type, 1, 0, MPI_COMM_WORLD);
24   else if(my_rank == 1)
25     MPI_Recv(arr, 2, my_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26
27   if(my_rank == 1){
28     for(i=0;i<30;i++) printf("%2d%c", arr[i], i%10==9?'\n':' ');
29   }
```

例 ex19.c (2プロセスで実行)

実行結果

```
 0  1 -1  3  4 -1  6  7 -1  9
10 11 12 -1 14 15 -1 17 18 -1
20 21 -1 -1 -1 -1 -1 -1 -1 -1
```

例 ex20.c (2プロセスで実行)

```
06   int i, arr[30];
07   MPI_Datatype my_type;
12
13   if(my_rank==0){
14     for(i=0;i<30;i++) arr[i] = i;
15     MPI_Type_vector(4, 2, 3, MPI_INT, &my_type);
16   } else if(my_rank == 1){
17     for(i=0;i<30;i++) arr[i] = -1;
18     MPI_Type_vector(2, 4, 10, MPI_INT, &my_type);
19   }
20   MPI_Type_commit(&my_type);
21
22   if(my_rank == 0)
23     MPI_Send(arr, 1, my_type, 1, 0, MPI_COMM_WORLD);
24   else if(my_rank == 1)
25     MPI_Recv(arr, 1, my_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26
27   if(my_rank == 1){
28     for(i=0;i<30;i++) printf("%2d%c", arr[i], i%10==9?'\n':' ');
29   }
```

例 ex20.c (2プロセスで実行)

実行結果

```
 0  1  3  4 -1 -1 -1 -1 -1 -1
 6  7  9 10 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

送るデータのサイズが同じであれば，受信と送信で違うデータ型を指定しても良い

プログラムのサンプル

- 21.c：0から$N-1$までの和を求める
- 22.c：0から$N-1$までの和を求める（別方法）