MPI-IO: 並列入出力関数

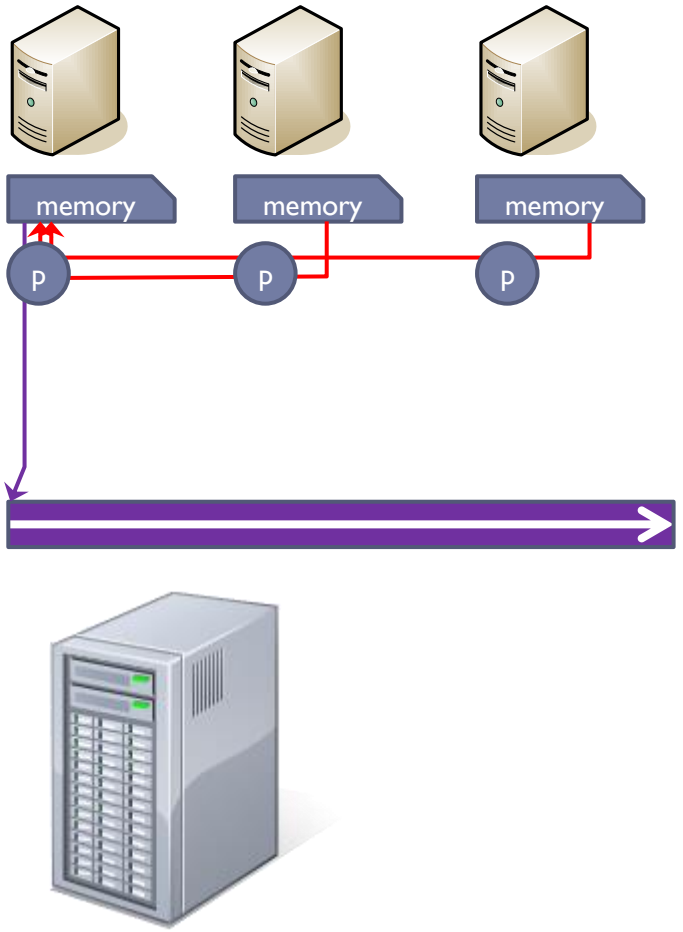
Hideyuki Jitsumoto (UTokyo)
jitumoto@cc.u-tokyo.ac.jp

MPI-IO とは

- 並列ファイルシステムと効率的にやり取り(送受信)するシステム
  - メモリ・ファイル双方における不連続域アクセス
  - 集団入出力
  - 明示的オフセットを用いた seek 発行の削減
  - プロセス間共有ファイルポインタ
  - ノンブロッキング出力
  - Etc etc…
- 以後 API は C言語での宣言を説明していくが、Fortran でも引数はほぼ同じである
  - 末尾に返り値用の引数がつく、各自リファレンスを参照のこと
- C++宣言に関してはobsoleteされたのでC宣言を利用すること

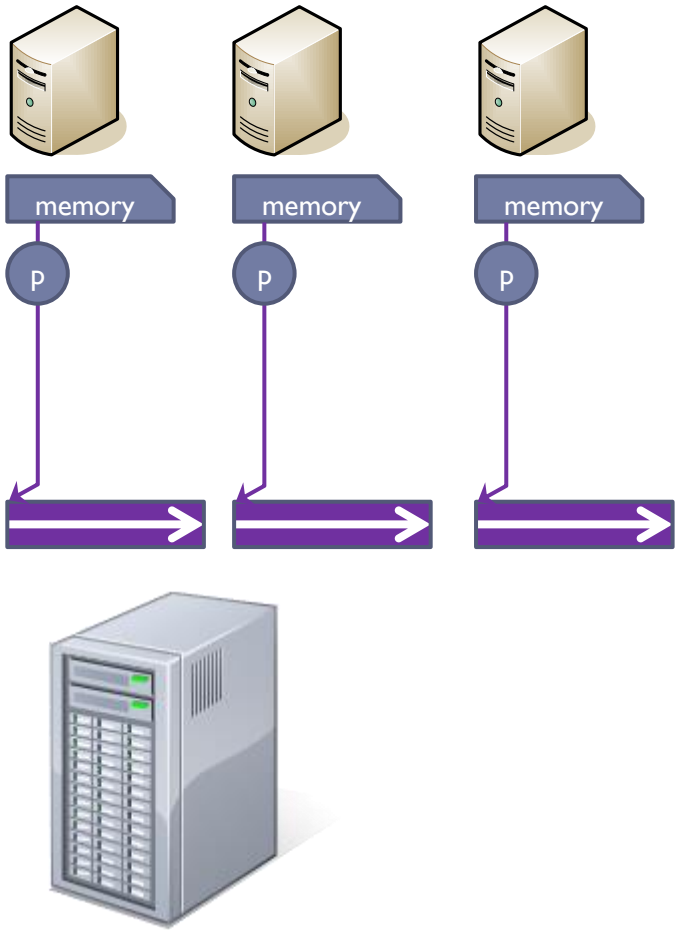
非MPI-IO：
よくある並列アプリ上ファイルI/O

- 逐次入出力
  - 利点
    - 単一ファイルによる優秀な取り回し
    - 書き込み操作回数の削減(トレードオフ)
  - 欠点
    - 書き込み時間の増加(トレードオフ)
    - 並列入出力をサポートしたファイルシステムの性能を生かせない



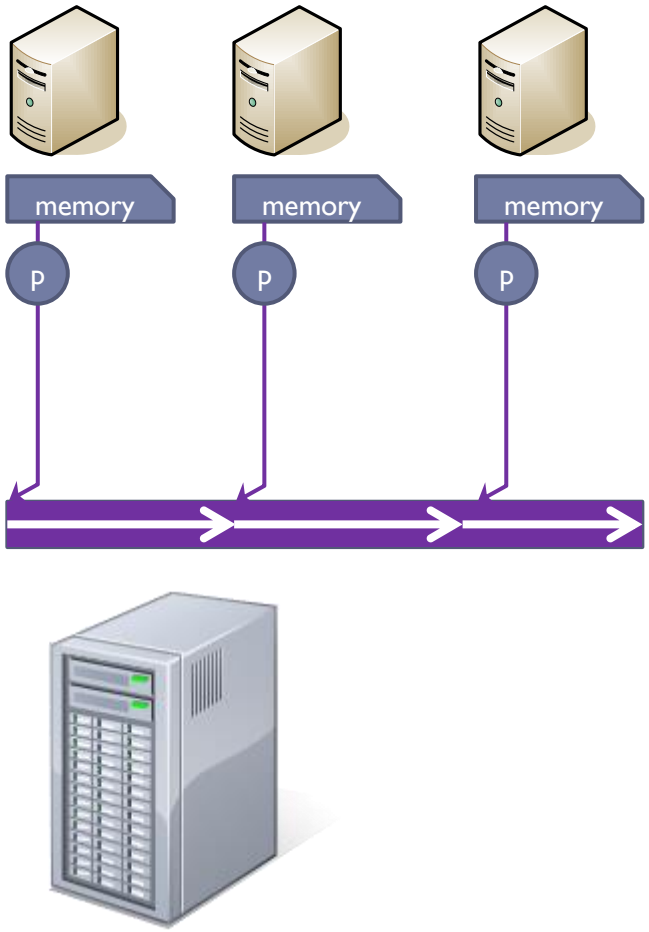
非MPI-IO：
よくある並列アプリ上ファイルI/O

- 並列入出力
  - 利点
    - ファイルシステムの並列入出力サポートを生かせる
    - 書き込み時間の削減（トレードオフ）
  - 欠点
    - 入出力回数の増大（トレードオフ）
      - 多数の小ファイル書き込み
    - 複数ファイルによる劣悪な取り回し



MPI-IOによる並列アプリ上ファイルI/O例

- 単一ファイル並列書き込み
  - 書き込み時間と書き込み回数のバランスを MPI-IOが考える
    - 実際には並列ファイルシステムに押しつけることがおおい
  - ファイルを1つにして取り回しの良さを維持



講習で用いるMPI-IOモデル

- プロセス毎 seek & read/write
  - プロセス毎のファイルポインタを利用し、読み書きを行う
  - MPI_File_seek, MPI_File_read, MPI_File_write
- 明示的オフセット
  - プロセス毎のファイルポインタを変更せず、読み書き開始位置を指定しながら書き込む
  - MPI_File_read_at, MPI_File_write_at
- 不連続アクセス（上級）
  - プロセスの読み書き可能位置を変更し、1ファイル内を不連続にアクセスする
  - MPI_File_set_view
- 共有ファイルポインタ（上級/参考）
  - プロセス全体で共有されるファイルポインタを用いた協調入出力
  - MPI_File_read/write_share

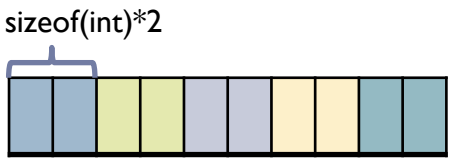
ファイルの open と close

- すべてのモデルについて利用前にファイルをopen、利用後に close する必要がある
- int MPI_File_open(
  MPI_Comm comm, //コミュニケータ
  char *filename, //操作ファイル名
  int amode, //アクセスモード（読専、書専等）
  MPI_Info info, //実装へのユーザからのヒント
  MPI_File *fh //ファイルハンドラ
)
- int MPI_File_close(
  MPI_File *fh //ファイルハンドラ
)

プロセス毎 seek & read/write

- プロセス毎のファイルポインタを利用し、読み書きを行う
  - MPI_File_open でハンドラを取得
  - MPI_File_seek で読み書きすべき位置に移動（プロセス毎に違う）
  - MPI_File_read/write で読み書き
  - MPI_File_close でファイルを閉じる

```
…
#define COUNT  2
MPI_File fh;
MPI_Status st;
int buf[COUNT];
int bufsize = sizeof(int)*COUNT;
MPI_File_open(MPI_COMM_WORLD, “datafile”,
                    MPI_MODE_RDONLY, MPI_INFO_NULL,
                    &fh);
MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, COUNT, MPI_INT, &st);
MPI_File_close(&fh);
…
```



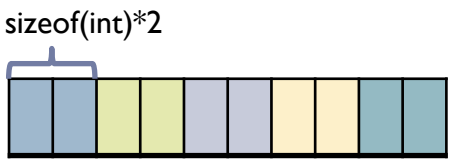
MPI_File_seek/read/write

- int MPI_File_seek(
  MPI_File fh, //ファイルハンドラ
  MPI_Offset offset, //オフセット位置(バイト)
  int whence //指定手法（セット／増加／終端等）
)
- int MPI_File_read(
  MPI_File fh, //ファイルハンドラ
  void *buf, //読み込みバッファ先頭アドレス
  int count, //データの「個数」
  MPI_Datatype dt, //データの型
  MPI_Status *st //終了状態の返値
)
- MPI_File_write は read と同じ引数、読み込みバッファ部分に書き込みバッファの先頭アドレスを示す

明示的オフセット

- 読み書き開始位置を指定しながら書き込む
  - MPI_File_open でハンドラを取得
  - MPI_File_read/write_at で読み書き
  - MPI_File_close でファイルを閉じる
- ファイルポインタの変更を行わない
  - スレッド利用のハイブリッドプログラムに最適
  - ファイルポインタはプロセス毎に一つ→クリティカルセクション

```
…
#define COUNT  2
MPI_File fh;
MPI_Status st;
int buf[COUNT];
int bufsize = sizeof(int)*COUNT;
MPI_File_open(MPI_COMM_WORLD, "datafile",
                    MPI_MODE_RDONLY, MPI_INFO_NULL,
                    &fh);
MPI_File_read_at(fh, rank*bufsize, buf, COUNT, MPI_INT, &st);
MPI_File_close(&fh);
…
```

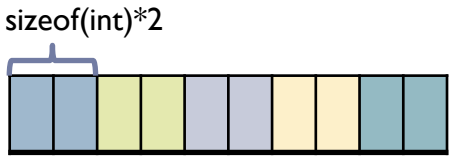


MPI_File_read/write_at

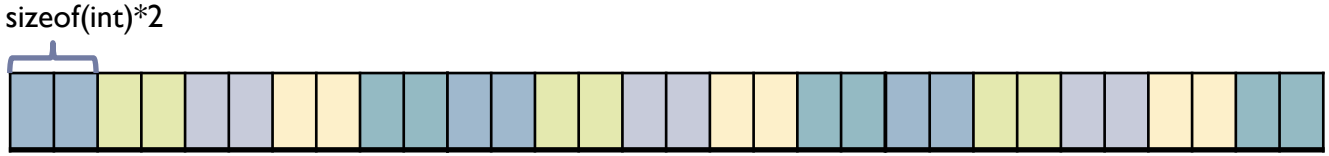
- int MPI_File_read_at(
  MPI_File fh, //ファイルハンドラ
  MPI_Offset ofs, //オフセット（バイト）
  void *buf, //読み込みバッファ先頭アドレス
  int count, //データの「個数」
  MPI_Datatype dt, //データの型
  MPI_Status *st //終了状態の返値
)
- MPI_File_write は read と同じ引数、読み込みバッファ部分に書き込みバッファの先頭アドレスを示す

不連続アクセス

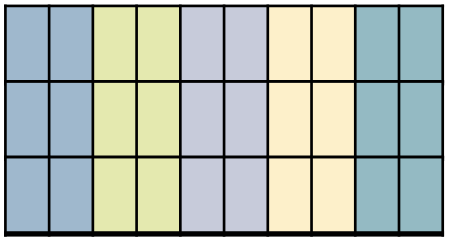
- 今までの話



- 実際にはこうなることも多い



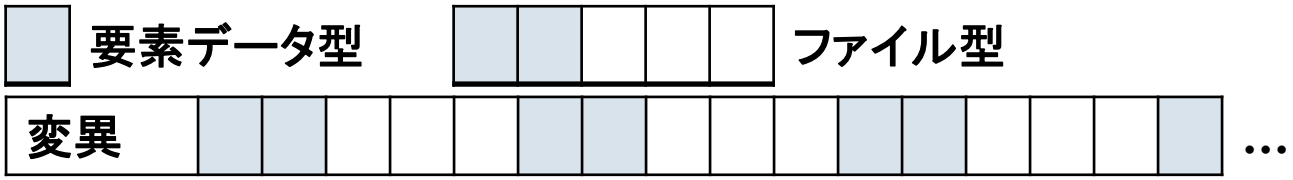
  - 大きな配列ファイルから部分配列を各プロセス毎に読み込んだり・・・
    - 10*3の配列データを2*3の部分配列毎にプロセスにそれぞれ読ませる



  - プロセス毎のアクセスパターンは
    - int を2個読んで、8個飛ばして、また2個読んで・・・

不連続アクセス：前提知識

- ファイルビュー
  - ファイル内で実際に操作できるウィンドウ
  - 以下の要素で指定（デフォルトは 0, MPI_BYTE, MPI_BYTE）
    - 変異：ファイルの先頭から読み飛ばすサイズ（バイト）
    - 要素データ型：データアクセスの基本単位（MPI_Datatype）
      - ファイルのオフセット単位になる
      - ファイルポインタも基本単位分スライドする
    - ファイル型：ファイルのどの部分がどのデータ型で見えるのか
      - 要素データ型のみから構成されるデータ型な必要がある
      - 操作不可能領域を作るために余白やextent（実データ長）を設定する
        - MPI_Type_vector のストライドや、MPI_Type_create_resizeのlb, extent を使う
        - ※後者だけ教えます（おまじないレベル？）



派生データ型(MPI_Sendとかにも使えます)

- MPI_INT などの MPI_Datatype をユーザ定義可能
  - 型作成→コミットという手順をとる
- MPI_Type_contiguous
  - 同じ基本型を複数並べる
- MPI_Type_create_resized
  - 型の実サイズを再定義
  - 型の前後に余白を作る

- 注意！(右では無視している)
  - ファイル可搬性のためにはバイト指定部分に言語の基本データ型を使わないほうがよい
    - 環境によってMPI_INTの実サイズが違うかもしれない
    - ファイル書き込み時の表現形式を使う
    - MPI_File_get_type_extent で書き込み時のMPI_Datatypeのextentを取得できる（リファレンスを参照）

←作成する型

```
…
MPI_Datatype filetype, temptype;

MPI_Type_contiguous(2, MPI_INT, &temptype);
// MPI_INT 2個のデータ型 temptype の作成

MPI_Type_create_resized(
    temptype, //拡張元の型
    0, //前方のデータ開始位置（最初からtemptypeを詰める）
    5*sizeof(int), //後方型終了位置（前後ともバイト単位）
    &filetype // 新しい型の格納先
);
// temptype の後ろに3個のint 分の余白を作成

MPI_Type_commit(&filetype);
// Datatype のコミット
…
```

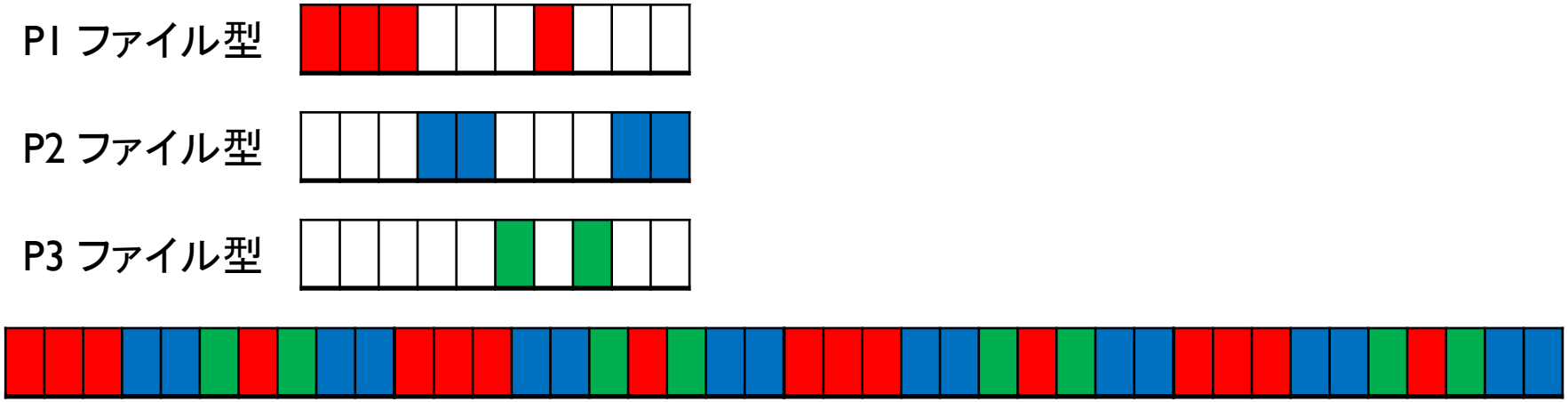
不連続アクセス

- プロセスの読み書き可能位置を変更し、1ファイル内を不連続にアクセスする
  - 派生型の作成（前頁）
  - MPI_File_open
  - MPI_File_set_view でファイルビューの作成
  - MPI_File_read/write で読み書き
  - MPI_File_close
- この例では1000個のint を buf に読み込む

```
…
MPI_Datatype filetype, temptype;
MPI_Type_contiguous(2, MPI_INT, &temptype);
MPI_Type_create_resized(temptype, 0, 5*sizeof(int), &filetype);
MPI_Type_commit(&filetype);
int count = 1000;
int disp = 0;
MPI_File_open(MPI_COMM_WORLD, "datafile",
                    MPI_MODE_RDONLY, MPI_INFO_NULL,
                    &fh);

MPI_File_set_view(
          fh, disp, MPI_INT, filetype, "native", MPI_INFO_NULL);

MPI_File_read(fh, buf, count, MPI_INT, &st);

MPI_File_close(&fh);

…
```

参考：ファイルビューはプロセス毎に定義可能

- 各プロセス毎のファイル型を変えられる
- プロセス毎に操作するデータの数が変わる
  - 下の例では実際の意味はないけど何かに使えるかも

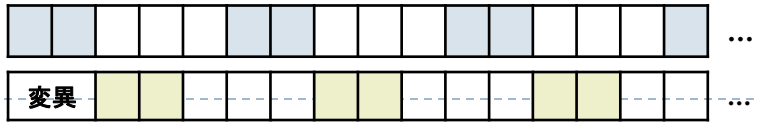


MPI_File_set_view

- int MPI_File_set_view(
  MPI_File fh, //ファイルハンドラ
  MPI_Offset disp, //変異（バイト数）
  MPI_Datatype dt, //要素データ型
  MPI_Datatype filetype, //ファイル型
  char* dtrep, //データ表現形式
  MPI_Info info //実装へのユーザからのヒント
)
- データ表現形式による可搬性向上
  - “native”: メモリ上と同じ姿での表現（何も変換しない）
  - “internal”: 同じMPI実装を利用するとき齟齬がない程度の変換
  - “external32”: MPIを利用する限り齟齬がない変換
  - そのほかにもある→MPI仕様書を参照

変異の操作と集団入出力関数

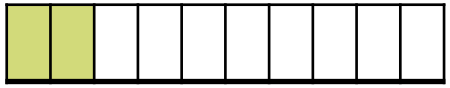
- ファイルビューの変異をランク毎に変えて呼び出すことで配列分割読み込みが可能
  - MPI-IO利用では本質的に集団入出力モデルが多い
- 集団入出力の宣言
  - 単独入出力とファイルへの結果は変わらない
  - MPI実装に、同時入出力を行っているというヒントを与え、最適化ができるかもしれない



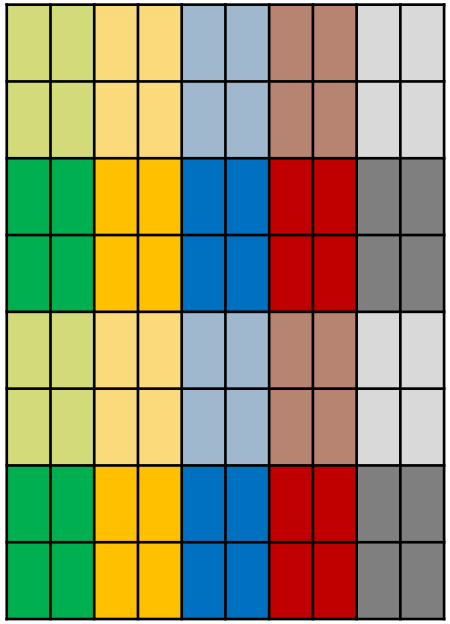

```
…
MPI_Datatype filetype, temptype;
MPI_Type_contiguous(2, MPI_INT, &temptype);
MPI_Type_create_resized(temptype, 0, 5*sizeof(int), &filetype);
MPI_Type_commit(&filetype);
int count = 1000;
int disp = rank*sizeof(int)*2;
MPI_File_open(MPI_COMM_WORLD, "datafile",
                    MPI_MODE_RDONLY, MPI_INFO_NULL,
                    &fh);

MPI_File_set_view(
          fh, disp, MPI_INT, filetype, "native", MPI_INFO_NULL);

MPI_File_read_all(fh, buf, count, MPI_INT, &st);

MPI_File_close(&fh);

…
```

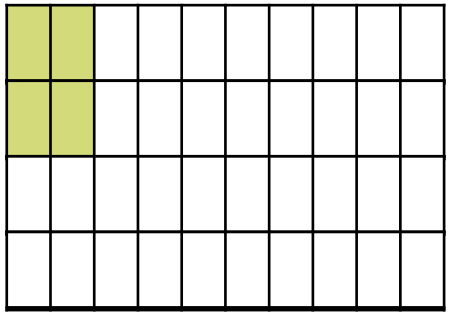
参考：二次元サイクリック分割

- 複数の型作成を組み合わせる
  - 10*8を10プロセスで2*2ずつ担当
- 4列でサイクリックになっている
  - まず最初の1列を作る
    - MPI_Type_contiguous(2, MPI_INT, &t1)
    - MPI_Type_create_resized (t1, 0, sizeof(int)*10, &t2)



  - 2列繰り返して、余白を20個つける
    - MPI_Type_contiguous(2, t2, &t3)
    - MPI_Type_create_resized (t3, 0, sizeof(int)*40, &filetype)
  - コミットする
- ファイルビューとして定義
  - 変異を工夫・・・あるいはもっと良いファイル型





参考：ソース例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char** argv){
int rank, size;
int disp;
char buf[5]; // ¥0が必要

MPI_Datatype t1, t2, t3, ft;
MPI_File fh;
MPI_Status st;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_Size(MPI_COMM_WORLD, &size);

MPI_Type_contiguous( 2, MPI_CHAR, &t1);
MPI_Type_create_resized(t1, 0, sizeof(char)*10, &t2);
MPI_Type_contiguous( 2, t2, &t3);
MPI_Type_create_resized(t3, 0, sizeof(char)*40, &ft);
MPI_Type_commit(&ft);
```

```
disp = rank*sizeof(char)*2+ (int)(rank/5)*sizeof(char)*10;
buf[4] = '¥0';

MPI_File_open(MPI_COMM_WORLD, "data.txt", MPI_MODE_RDONLY,
MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_CHAR, ft, "native", MPI_INFO_NULL);
MPI_File_read_all(fh, buf, 4, MPI_CHAR, &st);
MPI_File_close(&fh);
printf("%d:%s¥n", rank, buf);

MPI_Finalize();
}
```

data.txt

```
00112233440011223344556677889955667788 99
```

output

```
8:8888
7:7777
2:2222
5:5555
6:6666
4:4444
3:3333
9:9999
1:1111
```

共有ファイルポインタ

- 1プロセスの読み書き動作が他プロセスの読み書きに波及する
  - MPI_File_open
  - MPI_File_read/write_sharedで共有ファイルポインタを用いた入出力をする
  - MPI_File_close
  - プロセスがAPIを発行した順に処理される（つまり順不同）
- 集団入出力
  - MPI_File_read/write_ordered
  - ランク順に読む

```
…
#define COUNT  2
MPI_File fh;
MPI_Status st;
int buf[COUNT];
MPI_File_open(MPI_COMM_WORLD, “datafile”,
                    MPI_MODE_RDONLY, MPI_INFO_NULL,
                    &fh);
MPI_File_read_shared(fh, buf, COUNT, MPI_INT, &st);
MPI_File_close(&fh);
…
```

便利な機能へのポインタ集

- ノンブロッキング入出力
  - MPI_File_iXX ⇔ MPI_Wait
  - MPI_File_XX_begin ⇔ MPI_File_XX_end
  - (XX: read, read_at, read_all, write, write_at …)
- さまざまな派生型を用いた応用
  - MPI_Type_create_darray, MPI_Type_create_subarray
    - 配列型（X[x][y][z] など、講習ではX[x*a+y*b+c]で扱っていた)
    - ファイルビューを用いたのりしろ込みのアクセス
  - MPI_Type_create_indexed_box
    - 不規則ファイルアクセス