MPI-IO

辻田　祐一
（理研AICS）

並列ファイルシステム

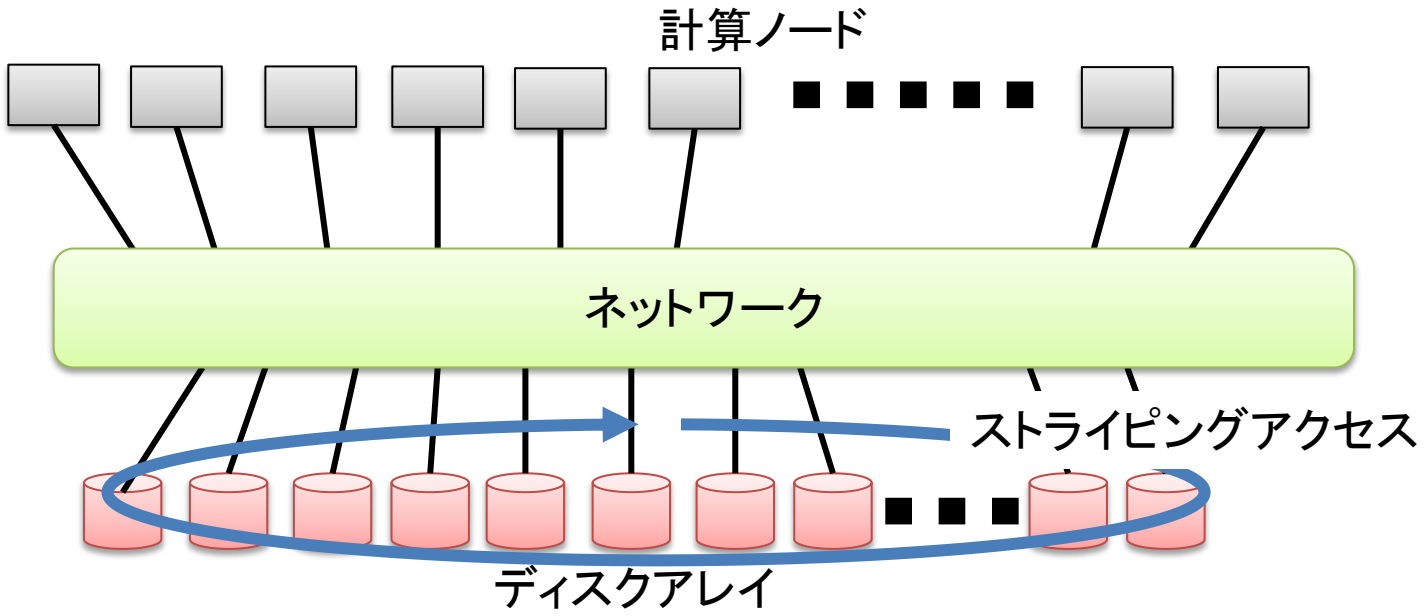
- 並列ファイルシステムが有する機能・特徴の例
  - データブロックの分散配置
  - メタデータ管理手法の最適化
  - データの信頼性と回復可能性
  - キャッシュ一貫性
  - 高い利用可能性
  - スケーラブルな容量と性能

ローカルファイルシステムと同じ使用イメージを提供しつつ、
高いI/O性能を実現

並列ファイルシステムの構成

- ネットワークを介してディスクアレイの複数のディスクにストライピングアクセスすることで高いI/O性能を実現。

- ネットワーク通信では、小さいサイズの通信は遅くなるため、ネットワークを介してアクセスする並列ファイルシステムでは、分割されるファイルサイズを小さくすると性能が低下する。よってある程度大きな単位で分割させる方が高い性能を得やすい。



並列ファイルシステムの必要性

- 広く利用されている分散ファイルシステムであるNFS
  - I/O性能が出ない
  - 大規模データのI/Oに不向き

- 並列ファイルシステムの利用
  - 計算ノードの数に比例するI/Oバンド幅
    - 複数のディスク間でストライピングすることで高い性能を実現
  - MPI-IOを用いた高速並列I/Oも利用可能

（参考）神戸大のFX10の並列ファイルシステム

- FEFS（Fujitsu Exabyte File System)というLustreをベースとした並列ファイルシステム
  - 基本機能はLustreと同じ
  - 扱えるファイル容量の増加や性能向上へのチューニングなど富士通が独自に改良を加えている。

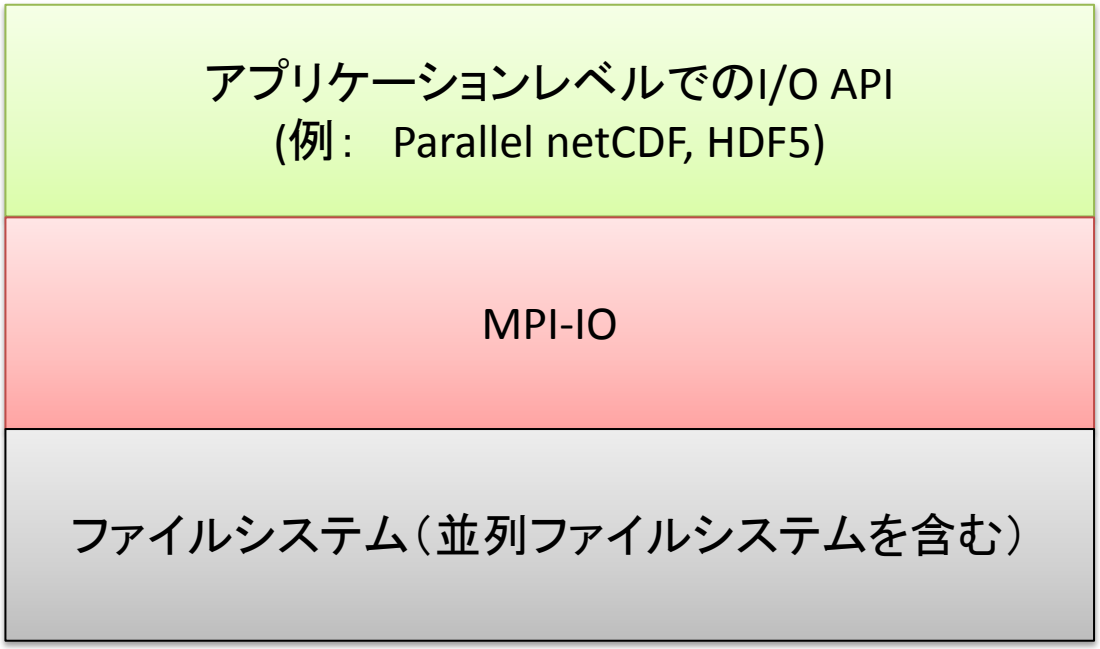
例えば、lfs df　でファイルシステムの構成（MDTおよびOSTの構成）が分かる。

```
[tsujita@pi NPB3.3-MPI_ext_pi]$ lfs df
UUID                 1K-blocks       Used   Available Use% Mounted on
home-MDT0000_UUID     553577888   50718552   474285640   9% /home[MDT:0]
home-OST0000_UUID   15270269408 2148166252 12357848348  14% /home[OST:0]
home-OST0001_UUID   15270269408 2104385764 12401628724  13% /home[OST:1]
home-OST0002_UUID   15270269408 2169024088 12336990540  14% /home[OST:2]
home-OST0003_UUID   15270269408 2125603924 12380410744  13% /home[OST:3]
home-OST0004_UUID   15270269408 2139166088 12366848564  14% /home[OST:4]
home-OST0005_UUID   15270269408 2093410696 12412603812  13% /home[OST:5]
home-OST0006_UUID   15270269408 2090395876 12415618744  13% /home[OST:6]
home-OST0007_UUID   15270269408 2155104296 12350910332  14% /home[OST:7]
home-OST0008_UUID   15270269408 2098221712 12407792800  13% /home[OST:8]
home-OST0009_UUID   15270269408 2156946792 12349067864  14% /home[OST:9]
home-OST000a_UUID   15270269408 2131663116 12374351392  13% /home[OST:10]
home-OST000b_UUID   15270269408 2160262008 12345752632  14% /home[OST:11]
home-OST000c_UUID   15270269408 2126495752 12379518896  13% /home[OST:12]
home-OST000d_UUID   15270269408 2113170988 12392843516  13% /home[OST:13]
home-OST000e_UUID   15270269408 2166148192 12339866480  14% /home[OST:14]
home-OST000f_UUID   15270269408 2099631916 12406382076  13% /home[OST:15]

filesystem summary:  244324310528 34077797460 198018435464  13% /home
```

MDTが1個、OSTが16個あるのが分かる。

MPI-IO

- MPI-IO
  - MPI Standardにおける並列I/Oを含むI/Oインタフェース群
  - 実装としてROMIOやOMPIOが広く利用されている。
  - HDF5やParallel netCDFなどのアプリ向けI/Oライブラリで並列I/O機能の基盤システムとして利用



MPI-IOによるオープンおよびクローズ

```
C:  MPI_File_open(MPI_Comm comm, char *filename,
              int amode, MPI_Info info, MPI_File *fh);
    MPI_File_close(MPI_File *fh);

F:  MPI_FILE_OPEN(comm, filename, amode, info, fh, ierr)
    MPI_FILE_CLOSE(fh, ierr)
```

- 集団操作のため、引数に与えるコミュニケータに属するプロセスで同じ関数を呼び出す必要あり。
- amode: アクセスモード（詳細は次のスライド）
- info: MPI-IOに関するヒント
- fh: ファイルハンドル（これを用いてファイル操作を行う。）

＊オープンしたら必ずクローズすること。

アクセスモード

- 以下の定義済みのビットのORでアクセスモードを定義する。

  - MPI_MODE_RDONLY — 読込のみ可能
  - MPI_MODE_RDWR — 読込みと書き込みの両方可能
  - MPI_MODE_WRONLY — 書込みのみ可能
  - MPI_MODE_CREATE — ファイルが無い場合、新規作成
  - MPI_MODE_EXCL — 既にファイルがある場合にエラーを返す
  - MPI_MODE_DELETE_ON_CLOSE — ファイルを閉じる際に消去
  - MPI_MODE_UNIQUE_OPEN — 同時にファイルをオープンしない
  - MPI_MODE_SEQUENTIAL —逐次的なファイルのオープン
  - MPI_MODE_APPEND — 全てのファイルポインタをファイル終端にセット

ファイル情報の設定

```
C:  MPI_File_set_info (MPI_File fh, MPI_info info);
    MPI_File_get_info(MPI_File fh, MPI_Info *info_used);

F:  MPI_FILE_SET_INFO (fh, info, ierr)
    MPI_FILE_GET_INFO (fh, info_used, ierr)
```

- MPI_File_set_info: ファイルI/Oに関する情報をkey,value対で設定
- MPI_File_get_info: 設定済みのファイルI/Oに関する情報を取得

＜MPI-IOに関連するkey,value対の例（他にも様々なkey,value対があります。）＞

key	Value（デフォルト値）	意味
cb_buffer_size	4194304 （対象ファイルシステムによって変わる可能性あり。） （例： LustreやFEFS → ストライプサイズに設定される。）	集団型I/Oの内部でI/O処理で使う一時バッファの大きさ
cb_nodes	ノード数	集団型I/OでI/O処理を行うプロセス数

- MPI_Info_setによりkey,value対が設定されたinfoをMPI_File_set_infoに与える。

ファイルビューの定義

```
C:  MPI_File_set_view (MPI_File fh, MPI_Offset disp,
                MPI_Datatype etype, MPI_Datatype ftype,
                char *datarep, MPI_Info info);

F:  MPI_FILE_SET_VIEW (fh, disp, etype, ftype,datarep, info, ierr)
```

- 集団操作： 全プロセスで呼び出しする。
- fh： 当該ファイルのファイルハンドル
- disp： オフセット値
- etype： ファイルビューを表すftypeを生成する基になるデータ型
- ftype： ファイルビューを表すデータ型
- datarep： 以下の3種類の内のどれかを指定
  - native： メモリ中のバイナリデータ表現と同じ表現
  - internal： 同一システム内で互換するデータ表現
  - external32: 異なるシステム間でも互換性のあるデータ表現
- info： MPI_Infoオブジェクト（key,valueペアで指定されたパラメタ群）

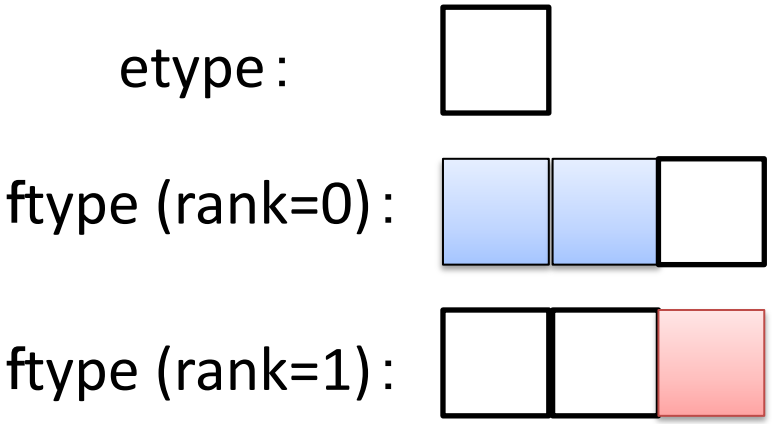
  （特に設定するものが無い場合、MPI_INFO_NULLを引数に与える。）

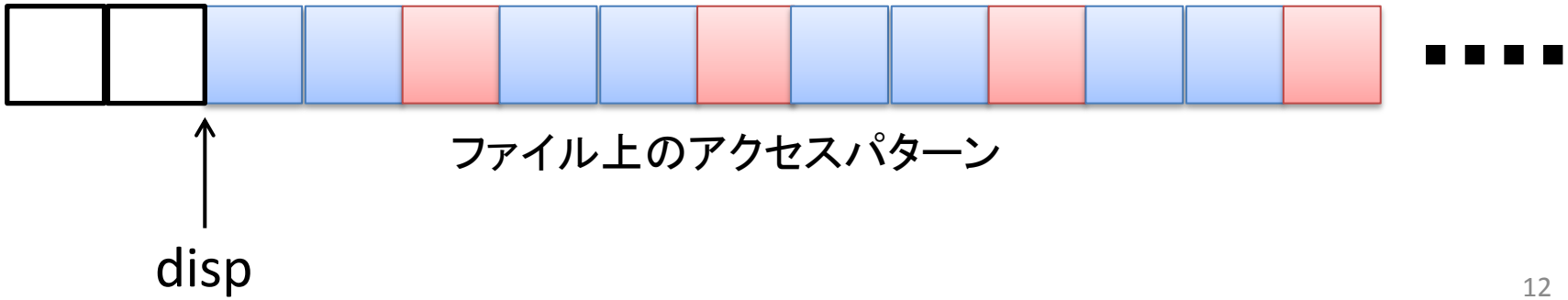
ファイルビューにおけるデータ型に関して

- etype
  - ファイルアクセスの単位となるデータ型
  - データ型のベースとなるデータ型がI/O処理で使われるデータ型と同じである必要あり。
- ftype
  - ファイルビューを表すデータ型
- datarep： 以下の3種類の内のどれかを指定
  - native：
    - メモリ中のバイナリデータ表現と同じ
    - 同一計算機内での利用を想定したデータ表現
  - internal：
    - 同一システム内で互換するデータ表現
  - external32:
    - 異なるシステム間でも互換性のあるデータ表現

MPI_File_openとMPI_File_set_view

- MPI_File_set_viewによるファイルビュー生成はMPI_File_openでファイルハンドルを取得後に行う。





通信・I/Oにおけるデータ型に関する操作

```
C: MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count);
   MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype,
        int *count);

F:  MPI_GET_COUNT (status, datatype, count, ierr)
    MPI_GET_ELEMENTS(status, datataype, count, ierr)
```

- 引数に与えるstatus： 通信やI/OでのstatusMPI_Get_countあるいはMPI_Get_elementが実行される。
  - 与えられたstatusに関連する通信あるいはI/Oに関して
- datatype： 通信あるいはI/Oに用いられる派生データ型
- count：
  - 受信あるいはI/Oを行ったデータの個数（MPI_Get_count）
  - 受信あるいはI/Oを行ったデータの基本データ型の個数（MPI_Get_elements）

使用例

```
…

MPI_Type_create_subarray(2, gsizes, lsizes, lstarts, MPI_ORDER_C,
            MPI_DOUBLE, &ftype);
MPI_Type_commit(&ftype);

MPI_File_open(MPI_COMM_WORLD, "./example.dat",
            MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_DOUBLE, ftype, "native", MPI_INFO_NULL);

MPI_File_write_all(fh, &(buf[0][0]), local_size, MPI_DOUBLE, &status);

MPI_Get_count(status, ftype, &d_count);
MPI_Get_elements(status, ftype, &d_element);

…
```

MPI_File_write_allで書き込んだデータにおける派生データ型ftypeに関する情報を取得

MPI-IO関数の分類

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	MPI_File_read_at, MPI_File_write_at	MPI_File_read_at_all, MPI_File_write_at_all
	nonblocking	MPI_File_iread_at, MPI_File_iwrite_at	MPI_File_iread_at_all, MPI_File_iwrite_at_all
	split collective	N/A	MPI_File_read_at_all_begin/end, MPI_File_write_at_all_begin/end
individual file pointers	blocking	MPI_File_read, MPI_File_write	MPI_File_read_all, MPI_File_write_all
	nonblocking	MPI_File_iread, MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	split collective	N/A	MPI_File_read_all_begin/end, MPI_File_write_all_begin/end
shared file pointer	blocking	MPI_File_read_shared, MPI_File_write_shared	MPI_File_read_ordered, MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared, MPI_File_iwrite_shared	N/A
	split collective	N/A	MPI_File_read_ordered_begin/end, MPI_File_write_ordered_begin/end

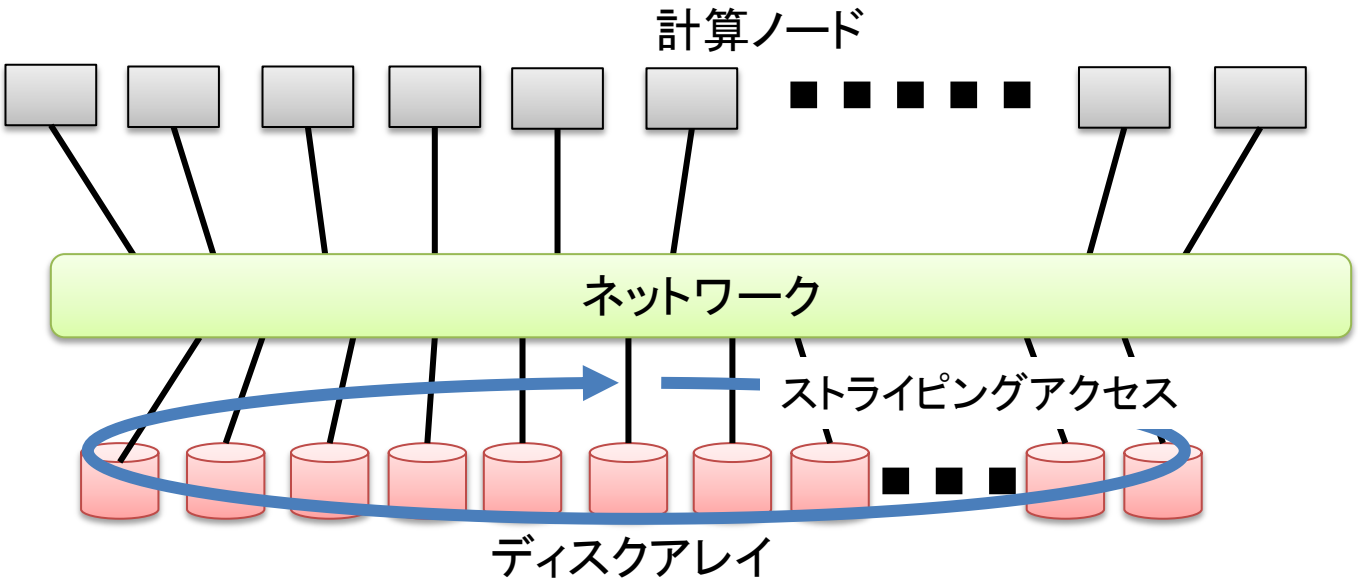
* MPI-3.1から追加： 実装によってはまだサポートされていない。

MPI-IO関数の選択

- Noncollective / Collective
- File view
- File pointer (independent / shared)
- Blocking / nonblocking

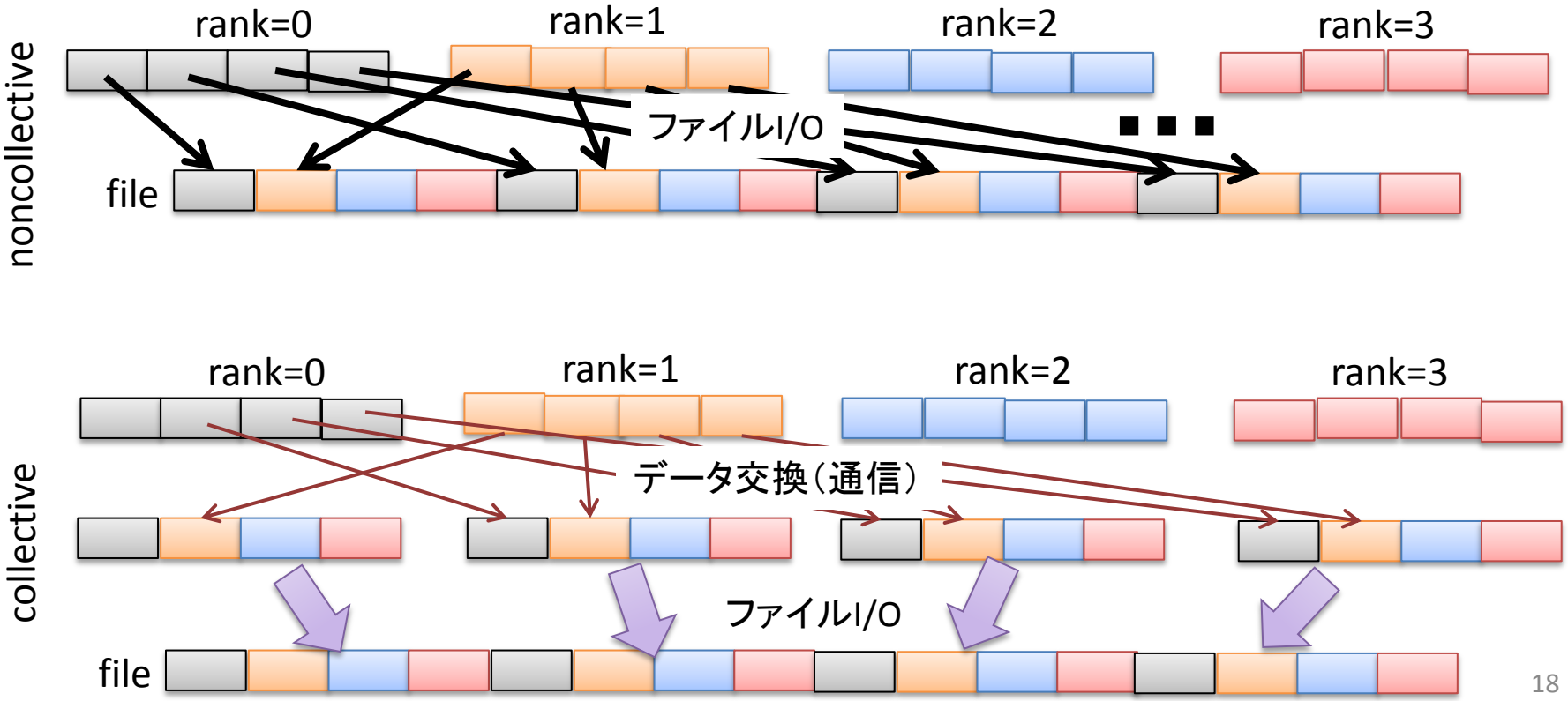
並列ファイルシステムと並列プログラムの特性

- 並列ファイルシステム
  - ディスクの数を増やすことで、高バンド幅、大容量のI/Oを実現
  - I/Oサイズを大きくしないと高い性能が望めない。
- 並列プログラム
  - Weak scaling： プロセスあたりのデータサイズは一定。プロセス数に比例して全体で扱うデータサイズが増大
  - Strong scaling： プロセス全体で扱うデータサイズが一定。プロセス数の増加に伴い、プロセスあたりのデータサイズが縮小
    - プロセス数が増える程、並列I/Oにおける性能向上が望めなくなる。



集団型I/O

- プロセス全体でI/O処理を行う。
- プロセス数の増加により、より大きなデータのI/Oが可能。
- 派生データ型を用いて不連続なデータの並びに対し、一度に多くのデータを扱うことが可能。
- I/Oを行うデータをファイル上で連続に並べ替えることで高いI/O性能が期待できる。（ファイルI/O性能向上が通信コスト増を大幅に上回る）



派生データ型による集団型I/O

- BTIOベンチマーク：NAS Parallel Benchmarks(NPB)のベンチマーク群の1つ
  - Class: ベンチマークの問題サイズ： A < B < C < D
  - Subtype
    - Simple: Noncollective
    - Full: Collective
- ベンチマーク結果の例（16プロセス@4ノードPCクラスタでの評価）
  - Intel Xeon E3-1280 V2（1ソケット/ノード）
  - メモリ： 32GiB
  - Interconnect: InfiniBand FDRx4（1 HCA/ノード）
  - Lustre File system
    - 1 MDS, 4 OSTs (2 OSTs/OSS)

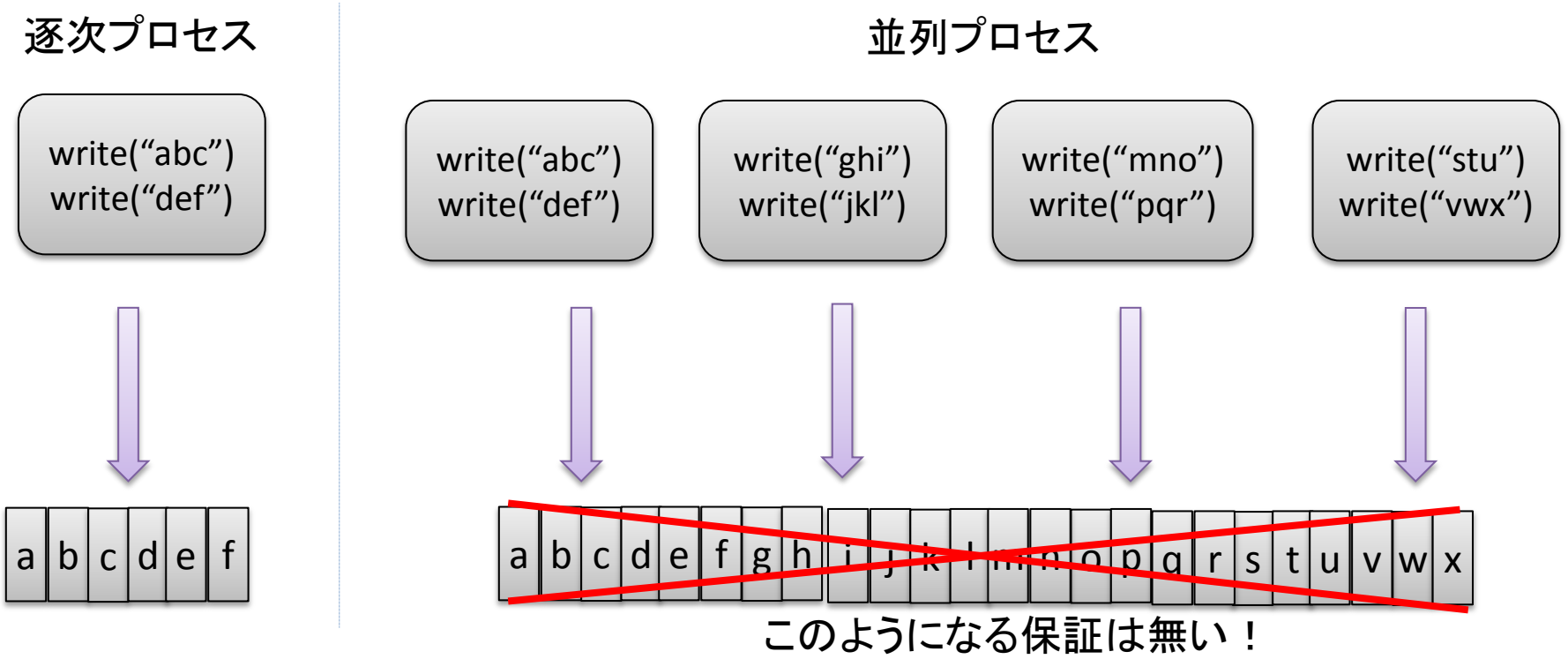
I/Oパターン	Class-B	Class-C
noncollective	16485.00	17133.45
collective	32200.14	38271.68

Collectiveの方が高い性能を実現（表内の数字の単位はMop/s）

- 集団型I/Oの問題点： 集団型I/Oの方が非集団型に比べてコードが複雑かつ長くなりやすい。
  - アクセス領域のプロセス間調整
  - 派生データ型の事前準備、等々

ファイルポインタに関する注意点

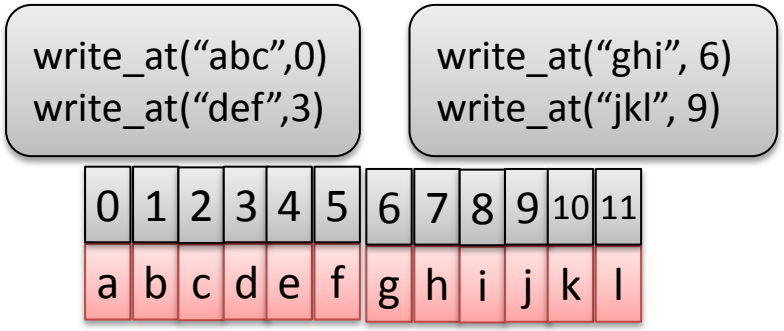
- 逐次プロセスの場合： ファイルポインタは1個（プロセス内で一意的）で、扱いは容易。
- 並列プロセスの場合： プロセス間でファイルポインタの動きを把握する必要あり。



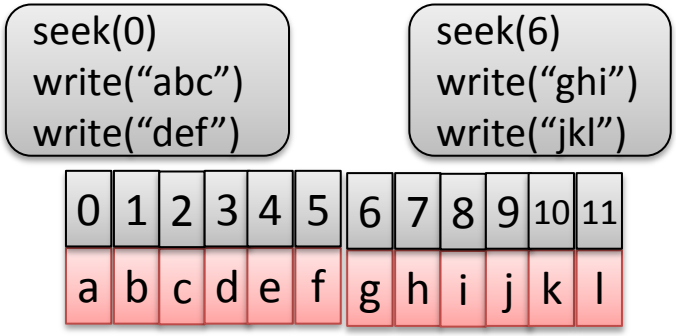
ファイルポインタの種類によって適切な対応を行う必要あり

MPI-IOにおけるオフセットの取り扱い

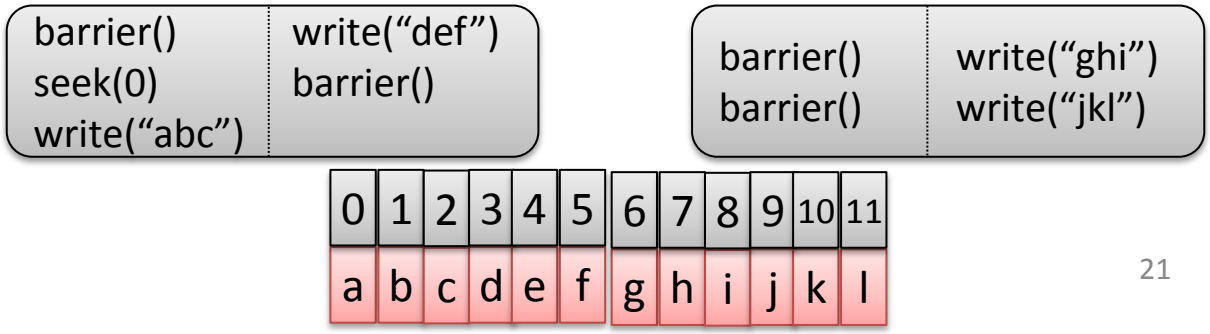
- 直接オフセットを指定してI/Oを行う
  - 例：MPI_File_write_at, MPI_File_read_at



- プロセス個々にファイルポインタを保持： 独立ファイルポインタ（Independent file pointer）



- プロセス間で一つのファイルポインタを共有
  - 例： MPI_File_write_shared, MPI_File_read_shared



ファイルポインタに対する操作

```
C: MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence);
   MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence);
   MPI_File_get_position(MPI_File fh, MPI_Offset *offset);
   MPI_File_get_position_shared(MPI_File_get_position(MPI_File fh,
         MPI_Offset *offset);

F:  MPI_FILE_SEEK (fh, offset, whence, ierr)
   MPI_FILE_SEEK_SHARED(fh, offset, whence, ierr)
   MPI_FILE_GET_POSITION(fh, offset, ierr)
   MPI_FILE_GET_POSITION_SHARED(fh, offset, ierr)
```

- 2種類のファイルポインタ（独立および共有）のそれぞれに対応する操作を行う関数が用意されている。（共有ファイルポインタ向けには関数名にsharedがある。）

- whenceに与える引数
  - MPI_SEEK_SET： offsetが指す位置にポインタを移動
  - MPI_SEEK_CUR： ポインタが現在指す位置＋offsetにポインタを移動
  - MPI_SEEK_END： ファイルの終端＋offsetの位置にポインタを移動

オフセット付I/O処理

C: MPI_File_write_at_all (MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status);
MPI_File_read_at_all (MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status); など

F: MPI_FILE_WRITE_AT_ALL (fh, offset, buf, count, datatype, status, ierr)
MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status, ierr) など

- オフセット付きでI/Oを行う関数の場合、引数にファイルハンドルを与えるが、I/O処理ではファイルポインタを使わない。
- 指定されたオフセット値のところから、datatypeの型をcount分だけI/Oを行う。

ファイル操作

```
C: MPI_File_delete (char *filename, MPI_Info info);
   MPI_File_preallocate (MPI_File fh, MPI_Offset size);
   MPI_File_get_size(MPI_File fh, MPI_Offset *size);

F:  MPI_FILE_DELETE (filename, info, ierr)
    MPI_FILE_PREALLOCATE (fh, size, ierr)
    MPI_FILE_GET_SIZE(fh, size, ierr)
```

- MPI_File_delete:　指定したファイルの削除
- MPI_File_preallocate:　指定したファイルに対し、引数で与えたサイズ分の領域をI/O処理を行う前に確保する。
- MPI_File_get_size:　指定したファイルのサイズを取得する。

MPI-IOのまとめ

- 集団型I/O
  - 全プロセスでI/O処理を実施
  - 派生データ型を使った集団型I/Oで高速化が期待できる。

- 3種類のファイルアクセス場所の指定方法
  - オフセット付きアクセス（ファイルポインタは使わない）
  - 独立ファイルポインタによるアクセス
    - File viewの指定が可能
  - 共有ファイルポインタによるアクセス
    - 逐次的に処理されるので、高い性能は望めない。

タイマ関数

```
C: double MPI_Wtime (void);

F:  DOUBLE PRECISION  MPI_WTIME ( )
```

- 秒単位での現時刻値を倍精度実数型の値で返す。

➢ 使用例

```
…
double   time;
…
time = MPI_Wtime ();
計算や通信・I/O処理など
time = MPI_Wtime() – time;
```

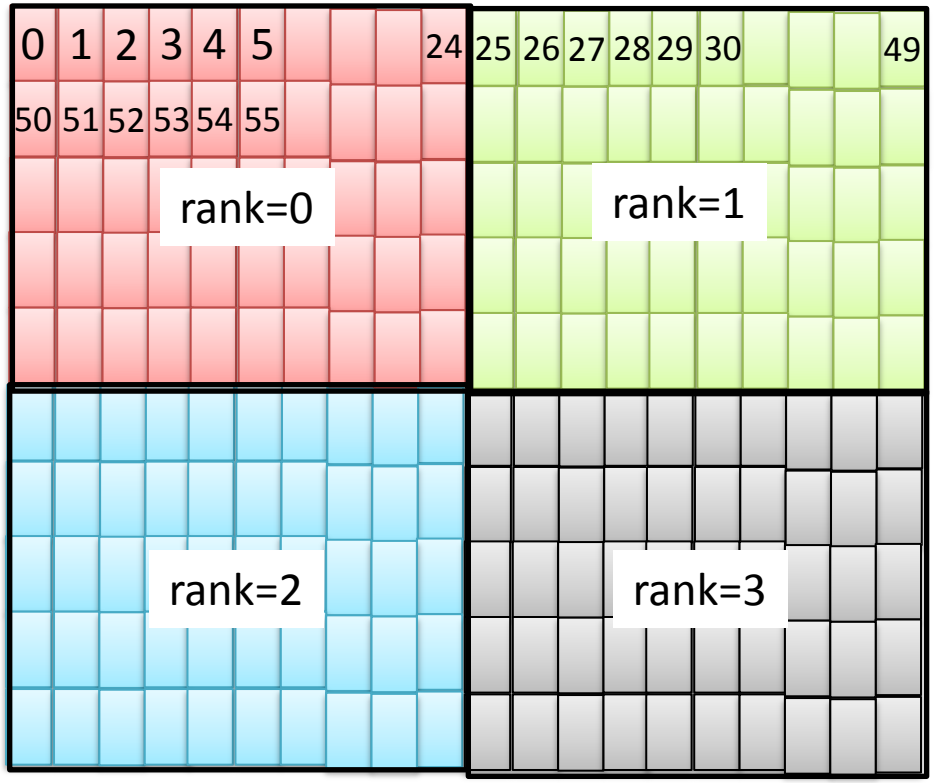
- 計測対象をMPI_Wtime()で挟み込み、取得した時間情報の差分を取ることで、経過時間を計測できる。

参考文献

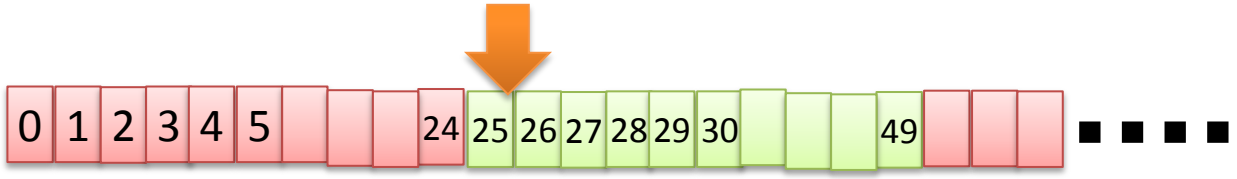
- MPI Forum
  http://www.mpi-forum.org/
  MPI仕様策定をしているコミュニティのページ
  ここから様々な仕様書等が取得可能
  最新の仕様書の版はMPI-3.1（2016年2月現在）
- MPI-2.1仕様書（日本語訳）
  http://www.pccluster.org/ja/mpi.html
  公開されている日本語訳では、現段階でこれが最新
- W. Gropp, E. Lusk, A. Skjellum, “Using MPI” (MIT Press)
- W. Gropp, E. Lusk, R. Thakur,“Using MPI-2” (MIT Press)
- W. Gropp, T. Hoefler, R. Thakur, E. Lusk, “Using Advanced MPI,” (MIT Press)
- P. Pacheco著, 秋葉博 訳, ”MPI並列プログラミング”（培風館）

演習課題1

- 4プロセス（ランク）の各々が、下の図にあるような部分配列を持っているとする。各プロセスが持つデータを図のファイル上の並び（図の番号順）になるようにMPI-IOで書き込みを行うプログラムを作成せよ。



配列全体の大きさは50×50でデータ型はMPI_DOUBLEとする。



<ファイル上の並び>

演習課題1（続き）

- プログラム作成にあたっては、以下のMPI関数を用いること。
  - MPI_Type_create_subarray
  - MPI_File_set_view
  - MPI_File_write_at_all

- また、部分配列の各要素の値は以下のようにランクごとに設定しておくこと。
  - rank=0　11.0
  - rank=1　13.0
  - rank=2　15.0
  - rank=3　17.0

- 出力されたファイルの中身をodコマンドで確認し、派生データ型の通りに書き込まれていることを確認せよ。
  （例えば、od –xv ファイル名 | less　で見て、先頭から正しい値が書き込まれていることを確認できるはず。）

演習課題2

- 4プロセスでストライプサイズを16MB、ストライプカウントが4となるファイル（ファイル名は適当に決めて良い。）を作成せよ。なおファイル生成のみでデータのI/Oは行わなくて良い。
- プログラムを実行し、ファイルが作成されたら、以下のようにしてストライプカウントとストライプサイズを確認し、正しく設定されていることを確認すること。（lfs getstripe ファイル名　で情報が表示されます。）

- ヒント：　ストライプサイズおよびストライプカウントの設定にはMPI_Info_setでinfoオブジェクトに設定し、ファイルのオープン時にこれを反映させる必要があります。なおストライプサイズおよびストライプカウントのkeyは以下の通りです。

  - ストライプサイズ：　striping_unit
  - ストライプカウント：striping_factor

補足：　神戸大のFX10では、デフォルトで以下のようになっているようです。
- ストライプサイズ：　2GB (2147483648)
- ストライプカウント：16

演習課題1: プログラム例

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

#define LEN  25

int alloc_2d_array_double(int, int, double ***);
void  free_2d_array_double(double ***);

int main(int argc, char *argv[]) {
  int  i, j, my_rank, my_size;
  int nsizes[2], nssizes[2], starts[2];
  double  val, **buf;
  MPI_Datatype  ftype;
  MPI_File  fh;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &my_size);
  nsizes[0] = nsizes[1] = 2*LEN;
  nssizes[0] = nssizes[1] = LEN;

  alloc_2d_array_double(LEN, LEN, &buf);

  switch(my_rank) {
    case 0:
      val = 11.0;
      starts[0] = starts[1] = 0;
      break;
    case 1:
      val = 13.0;
      starts[0] = 0;
      starts[1] = LEN;
      break;
    case 2:
      val = 15.0;
      starts[0] = LEN;
      starts[1] = 0;
      break;
    case 3:
      val = 17.0;
      starts[0] = starts[1] = LEN;
      break;
  }

  for(i = 0; i < LEN; i++) {
    for(j = 0; j < LEN; j++) {
       buf[i][j] = val;
    }
  }
  MPI_Type_create_subarray(2, nsizes, nssizes, starts,
      MPI_ORDER_C, MPI_DOUBLE, &ftype);
  MPI_Type_commit(&ftype);
  MPI_File_open(MPI_COMM_WORLD, "./example.dat",
    MPI_MODE_CREATE|MPI_MODE_RDWR,
    MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, 0, MPI_DOUBLE, ftype, "native",
     MPI_INFO_NULL);
  MPI_File_write_at_all(fh, 0, &(buf[0][0]), LEN*LEN, MPI_DOUBLE,
     &status);
  MPI_File_close(&fh);
  MPI_Type_free(&ftype);

  free_2d_array_double(&buf);

  MPI_Finalize();
  return 0;
}
```

<次のページに続く>

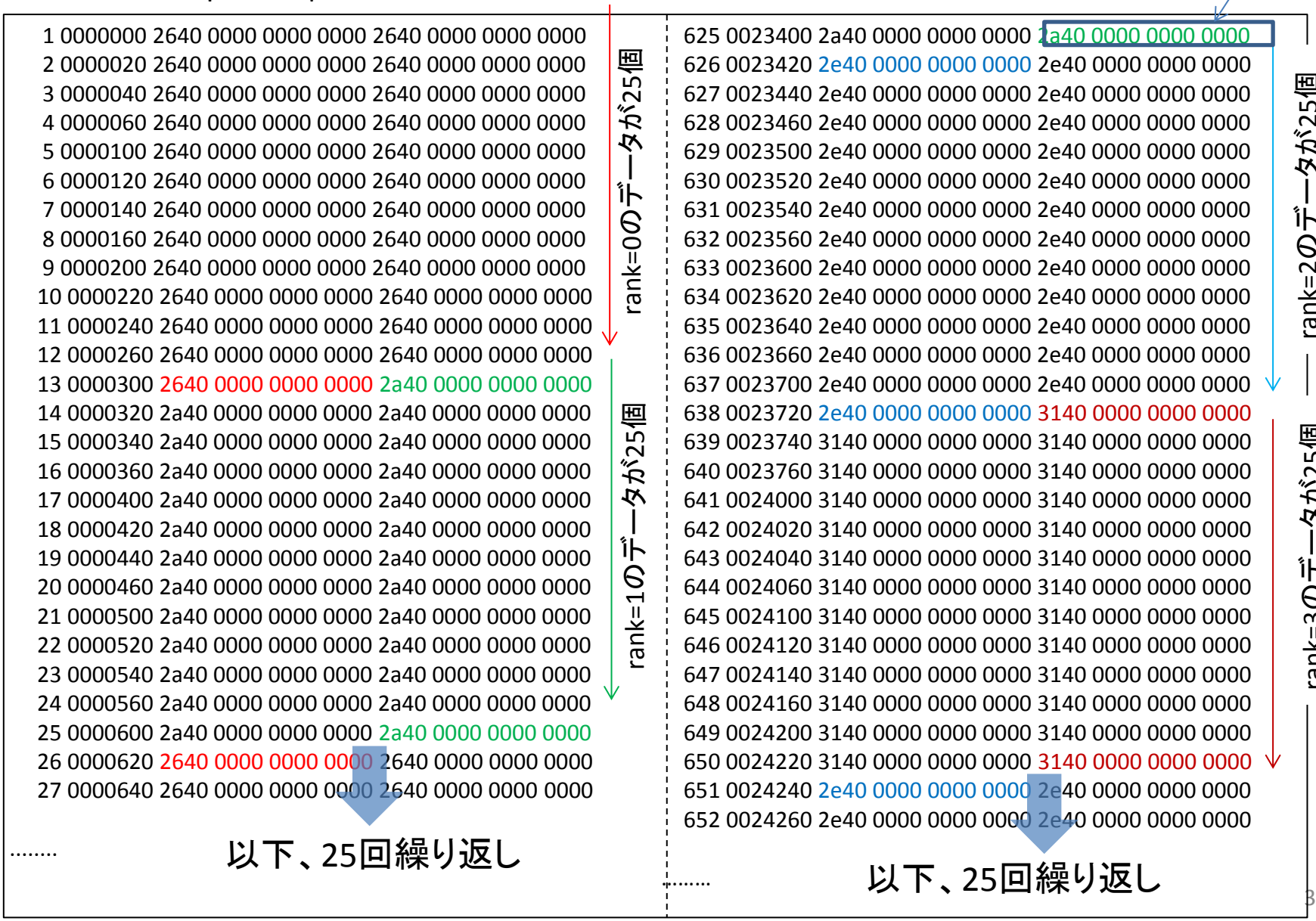
演習課題1：プログラム例（続き）

```
int alloc_2d_array_double(int dimx, int dimy, double ***p)
{
  int   I;
  double *storage;

  storage = (int *) malloc(dimx * dimy * sizeof(double));
  if (storage == NULL) {
    fprintf(stderr, “Error in malloc¥n”);
    return -1;
  }
  *p = (double **) malloc(dimx * sizeof(double *));
  if (*p == NULL) {
     fprintf(stderr, “Error in malloc¥n”);
     return -1;
  }
  for(i = 0; i < dimx; i++) {
     (*p)[i] = &storage[i*dimy];
   }
   return 0;
}

void free_2d_array_double(double ***p) {
  if (*p != NULL) {
    free((*p)[0]);
  }
  free(*p);
}
```

＊この例では、静的に配列を確保せずに、意図的に動的に配列用のメモリ領域を確保しています。
よってプログラム終了時にメモリ領域を解放しています。

演習課題1：出力データ確認の例

od –xv ./example.dat | less –N　で確認した結果（-Nオプションで行番号を付けています。）



演習課題2:プログラム例

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

#define FILENAME  "./test-io.dat"

int main(int argc, char** argv)
{
   int     i, rank, size;
   int     slen = 0;
   char    pname[MPI_MAX_PROCESSOR_NAME];
   MPI_File    fh;
   MPI_Info    myinfo;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Get_processor_name(pname, &slen);

   MPI_Info_create(&myinfo);

   MPI_Info_set(myinfo, "striping_unit", "16777216");
   MPI_Info_set(myinfo, "striping_factor", "4");
   /* start MPI-IO operations */
   if(MPI_SUCCESS != MPI_File_open(MPI_COMM_WORLD,
                     FILENAME,
                     MPI_MODE_WRONLY|MPI_MODE_CREATE,
                     myinfo, &fh)){
      MPI_Finalize();
      printf("error in MPI_File_open¥n");
      fflush(stdout);
      return -1;
   }
   if(MPI_SUCCESS != MPI_File_close(&fh)){
      fprintf(stderr, "MPI_File_close failed.¥n");
      MPI_Finalize();
      return -1;
   }

  /* free MPI_Info object */
   MPI_Info_free(&myinfo);

   if(MPI_SUCCESS != MPI_Finalize()){
      fprintf(stderr, "error in MPI_Finalize¥n");
      return 1;
   }

   return 0;
}
```