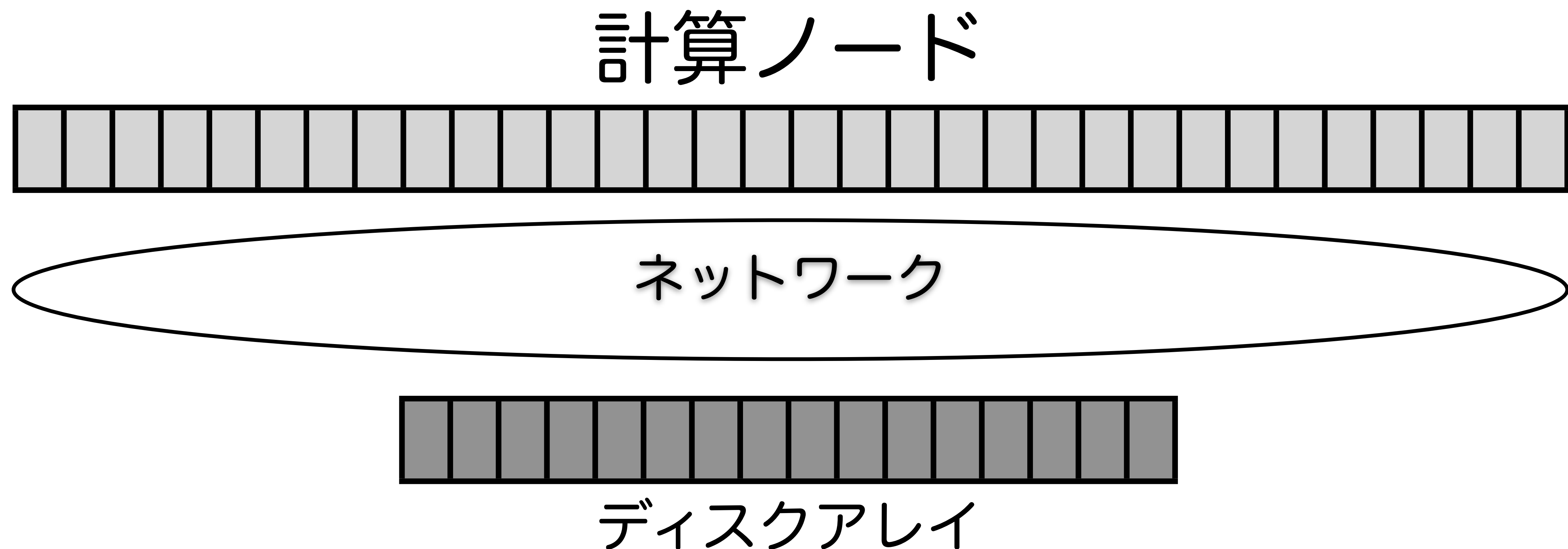
MPI-IO

理化学研究所 AICS
システムソフトウェア研究チーム
堀　敦史

RIKEN AICS HPC Spring School　　2015/3/6

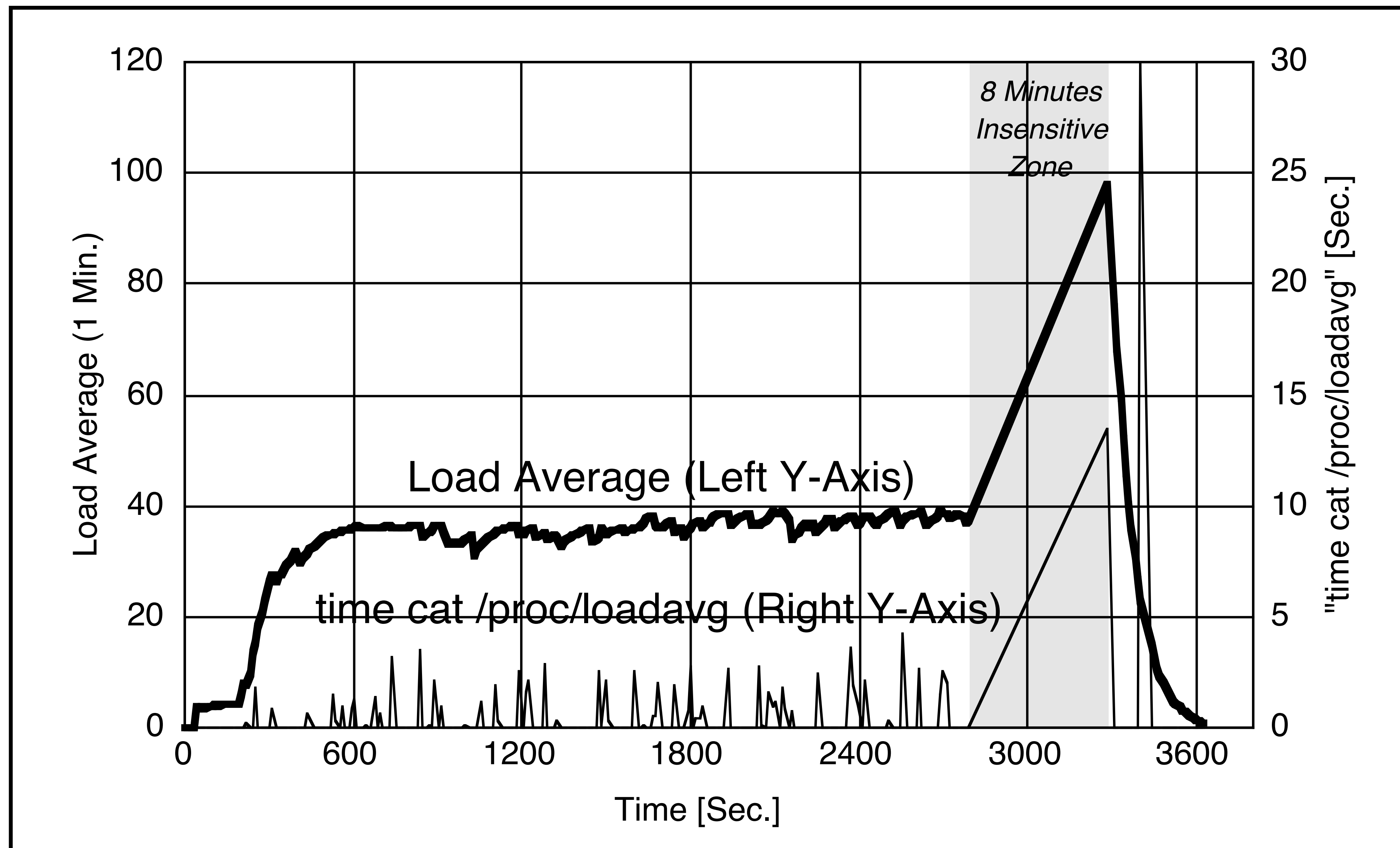
並列ファイルシステムの構成

- 並列ファイルシステムでは、ネットワークを経由してファイルにアクセスする
- 通信の特性として、小さいメッセージは遅いので、結果として、小さい単位でのファイル IO は遅くなる



並列ファイルシステムの必要性

- よく使われている NFS では、まったく性能が出ない！



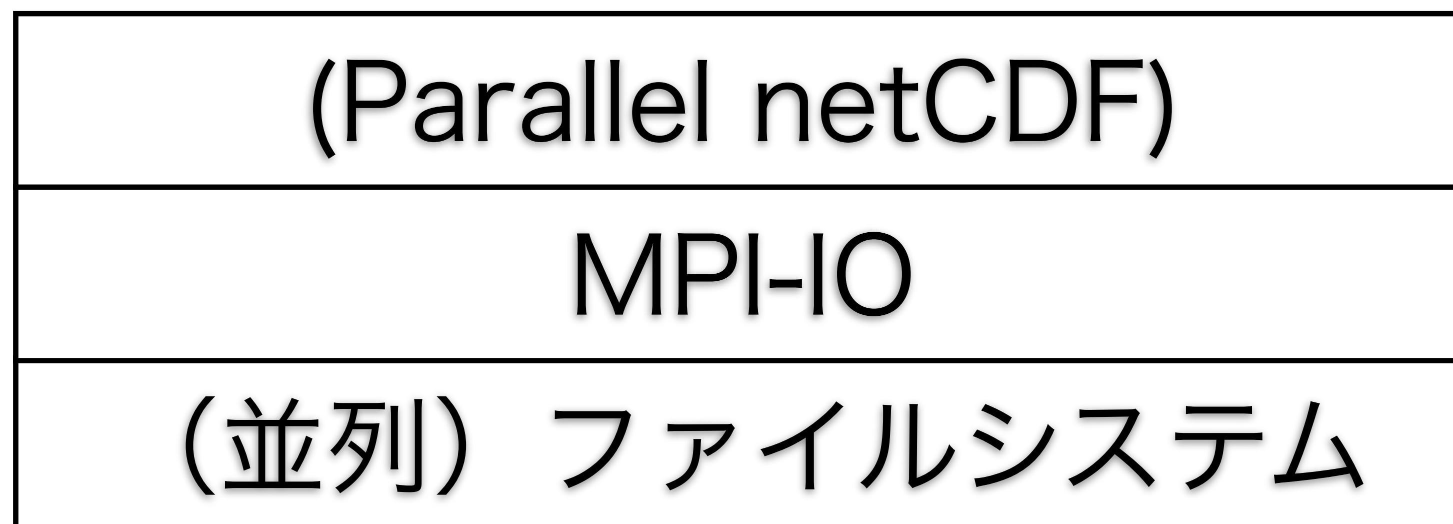
A timeline of NFS server load average when 64 processes (4 nodes) are creating 1GB file

並列ファイルシステム

- 計算ノード数に（ほぼ）比例する IO バンド幅を提供する
- 普通のディスクのバンド幅は 60 MB/s 前後
  - 普通の SSD で 100 MB/s 前後
- これより大きなバンド幅は、複数のディスクを高速ネットワークで束ねて実現している
  - **小さな単位のアクセスでは性能が全く出ない！**
- MPI-IO ではさまざまな工夫がある

MPI-IO のソフトウェア構造

- MPI-IO は（並列）ファイルシステムを MPI という枠組みで抽象化したもの
- 性能向上のためにヒント情報を渡すことができるようになっている
  - MPI-IO の実装に依存したパラメータ
  - 機種やファイルシステム依存したパラメータ
- MPI-IO を用いた並列 IO ライブラリも存在する
  - Parallel netCDF　http://trac.mcs.anl.gov/projects/parallel-netcdf/



MPI-IO - open と close

```
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 )
```

- 集団呼出し（コミュニケータ comm に属する全てのプロセスで同じように呼び出す必要がある）
- **amode** :次頁参照
- **info** : ファイルシステムへのヒント（MPI_INFO_NULL)
- **fh**：ファイルハンドル - これに対しファイルを操作する

MPI_File_open における amode

- amode - アクセスモード
  - 以下の値のビット OR

- MPI_MODE_RDONLY — read only,
- MPI_MODE_RDWR — reading and writing,
- MPI_MODE_WRONLY — write only,
- MPI_MODE_CREATE — create the file if it does not exist,
- MPI_MODE_EXCL — error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE — delete file on close,
- MPI_MODE_UNIQUE_OPEN — file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL — file will only be accessed sequentially,
- MPI_MODE_APPEND — set initial position of all file pointers to end of file.

MPI_File_set_view

```
C: MPI_File_set_view( MPI_File *fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype ftype,
                      char *datarep )
F: MPI_FILE_SET_VIEW( fh, disp, etype, ftype, datarep, ierr )
```

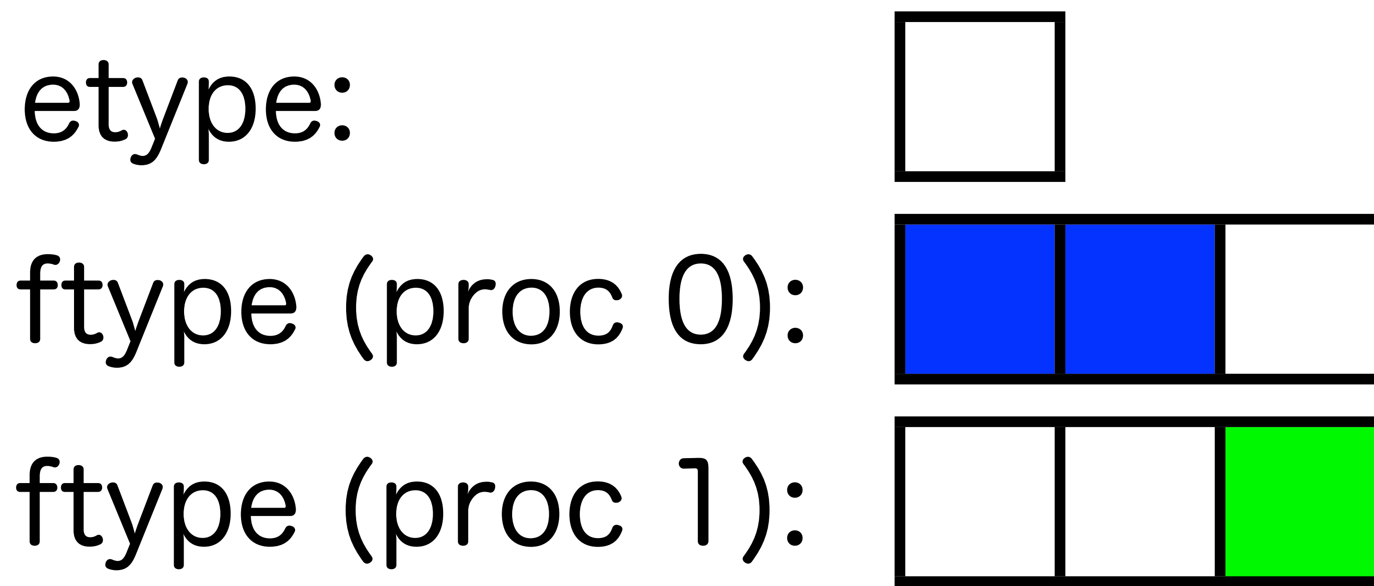
- Collective 関数
- **ftype**：etype の派生データ型で、これを呼んだプロセスがそのプロセスの **FP を用いて**アクセスするファイルの部分を示す
- **datarep**：ファイルに格納する数値データの表現方式（互換性）
  - "native"　　　マシン固有のバイナリー表現（無変換）
  - "internal"　　（native と external32 の中間）
  - "external32"　もっとも一般的で可搬性のあるバイナリー表現
- **info**：ファイルシステムに与えるヒント情報（システム依存）

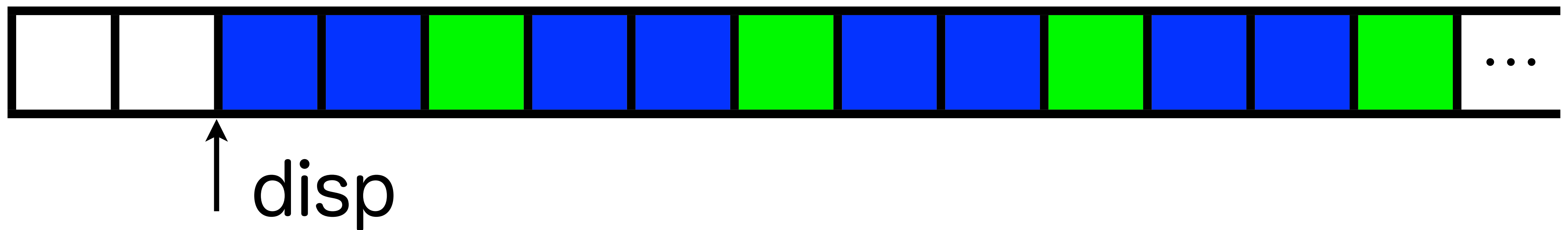
MPI-IO における Datatype

- etype (elementary type)
  - ファイルにアクセスする場所を示す基本的な型
  - READ/WRITE で指定されるデータ型と *type signature* が同じでなければならない（各要素は同じ基本データ型であること）
- ftype (file type)
  - ファイルアクセスを分割するためのテンプレート
- デフォルト
  - etype：MPI_BYTE
  - ftype：MPI_BYTE

MPI_File_open と MPI_File_set_view

- 通常 MPI_File_set_view は MPI_File_open の直後に宣言する





ファイルの IO

```
C:  MPI_File_read( MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *status)
    MPI_File_read_all( MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *status)
    MPI_File_read_at( MPI_File fh, MPI_Offset off, void *buf, int count, MPI_Datatype type,
                      MPI_Status *status)
    MPI_File_read_at_all( MPI_File fh, MPI_Offset off, void *buf, int count, MPI_Datatype type,
                      MPI_Status *status)

    MPI_File_write( MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *status)
    MPI_File_write_all( MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *status)
    MPI_File_write_at( MPI_File fh, MPI_Offset off, void *buf, int count, MPI_Datatype type,
                      MPI_Status *status)
    MPI_File_write_at_all( MPI_File fh, MPI_Offset off, void *buf, int count, MPI_Datatype type,
                      MPI_Status *status)

F:  MPI_FILE_READ( fh, buf, count, datatype, status, ierr )
    MPI_FILE_READ_ALL( fh, buf, count, datatype, status, ierr )
    MPI_FILE_READ_AT( fh, offset, buf, count, datatype, status, ierr )
    MPI_FILE_READ_AT_ALL( fh, offset, buf, count, datatype, status, ierr )

    MPI_FILE_WRITE( fh, buf, count, datatype, status, ierr )
    MPI_FILE_WRITE_ALL( fh, buf, count, datatype, status, ierr )
    MPI_FILE_WRITE_AT( fh, offset, buf, count, datatype, status, ierr )
    MPI_FILE_WRITE_AT_ALL( fh, offset, buf, count, datatype, status, ierr )
```

MPI_Status

```
C:  MPI_Get_count( MPI_Status status, MPI_Datatype type, int *count )
F:  MPI_GET_COUNT( status, type, count, ierr )
```

- MPI_Status
  - 1対1通信における受信
  - MPI-IO におけるデータの read/write
- MPI_Status に対し、MPI_GET_COUNT を呼ぶと、その status を返した操作におけるデータの個数が返る
  - 1対1通信の受信：受信したデータの個数
  - MPI-IO：I/O したデータの個数

ファイルアクセス関数の分類

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 & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

MPI-IO におけるファイルのアクセス

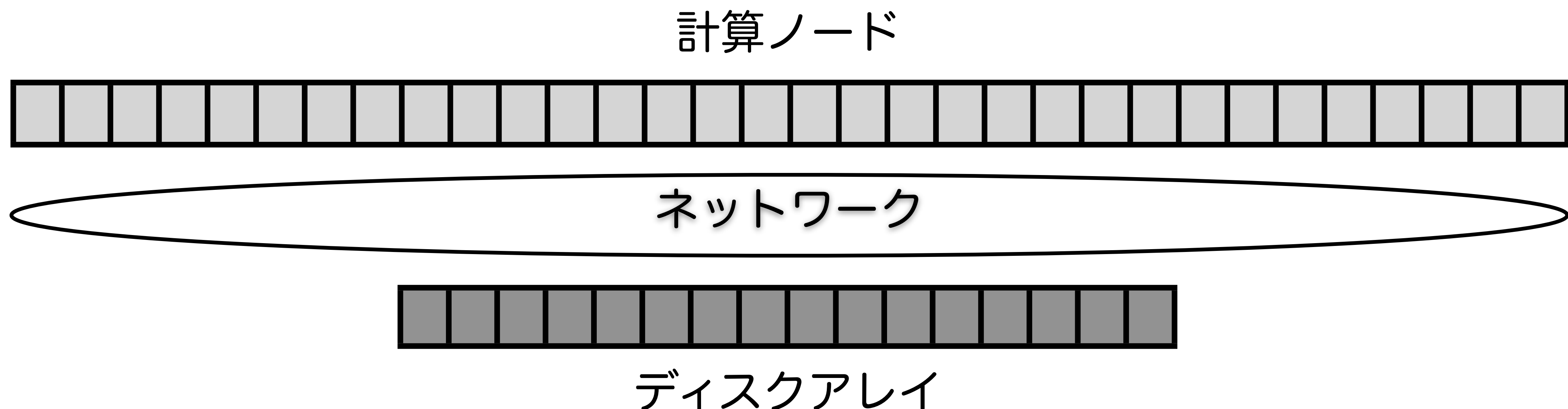
- Collective IO
  - 集団 IO
- File Pointer
  - アクセスする場所の指定
  - File View
    - 個々のプロセスへの割り付け
- (Non) Blocking IO
  - IO と計算のオーバーラップ

ファイルアクセス 関数

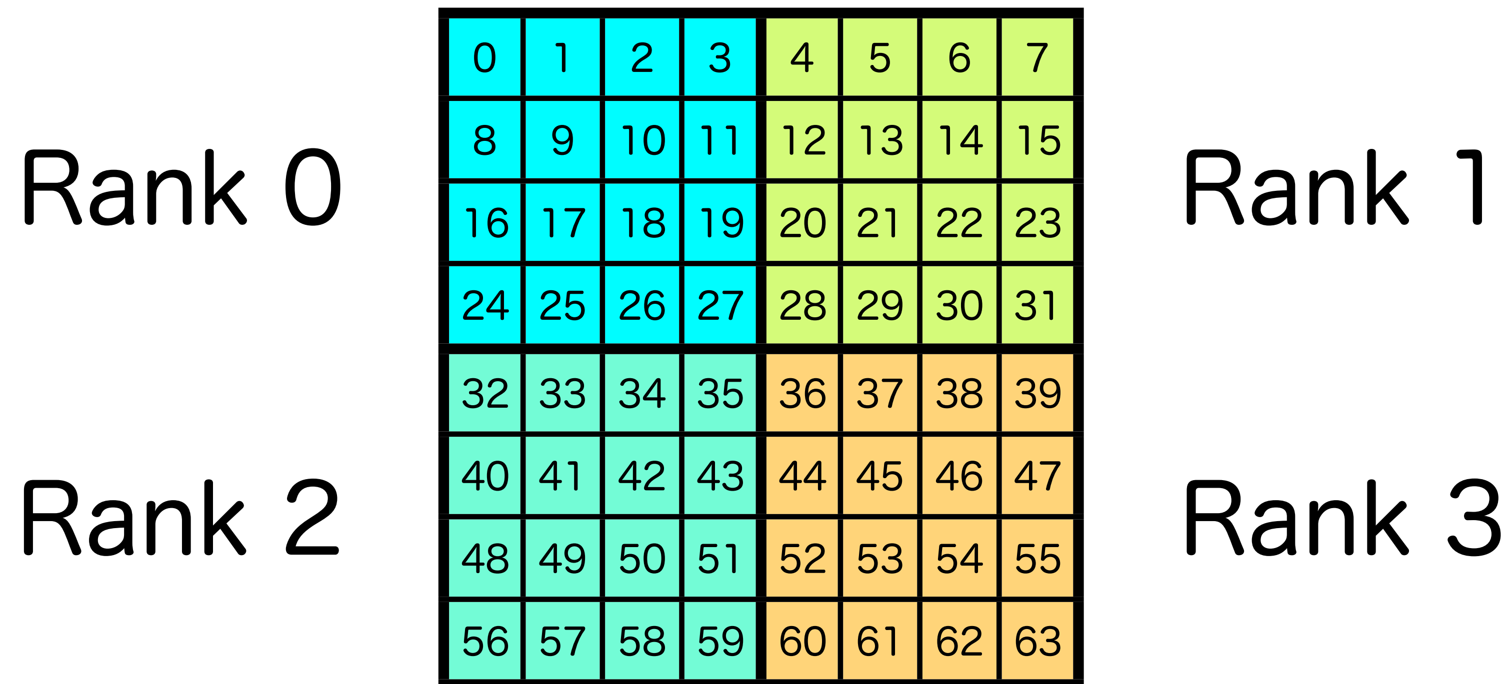
positioning	**synchronism**	**coordination**	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

並列ファイルシステムの特性と並列プログラムの特性

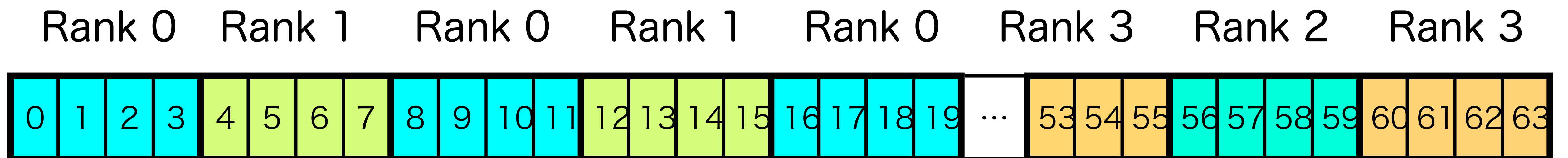
- 並列ファイルシステムの特性
  - 複数のディスクを同時に使うことでバンド幅を稼ぐ
  - 複数のディスクを同時に使うほど、大きな単位でアクセスしないと、バンド幅がでない
- 並列プログラムの特性
  - **Weak Scaling**　ノード数に比例して問題サイズが大きくなる
  - **Strong Scaling**　ノード数に依らず問題サイズが一定
    - ノード数が大きくなるとノードあたりのデータは小さくなる
    - IO の性能が悪くなる



配列の block 分割の場合

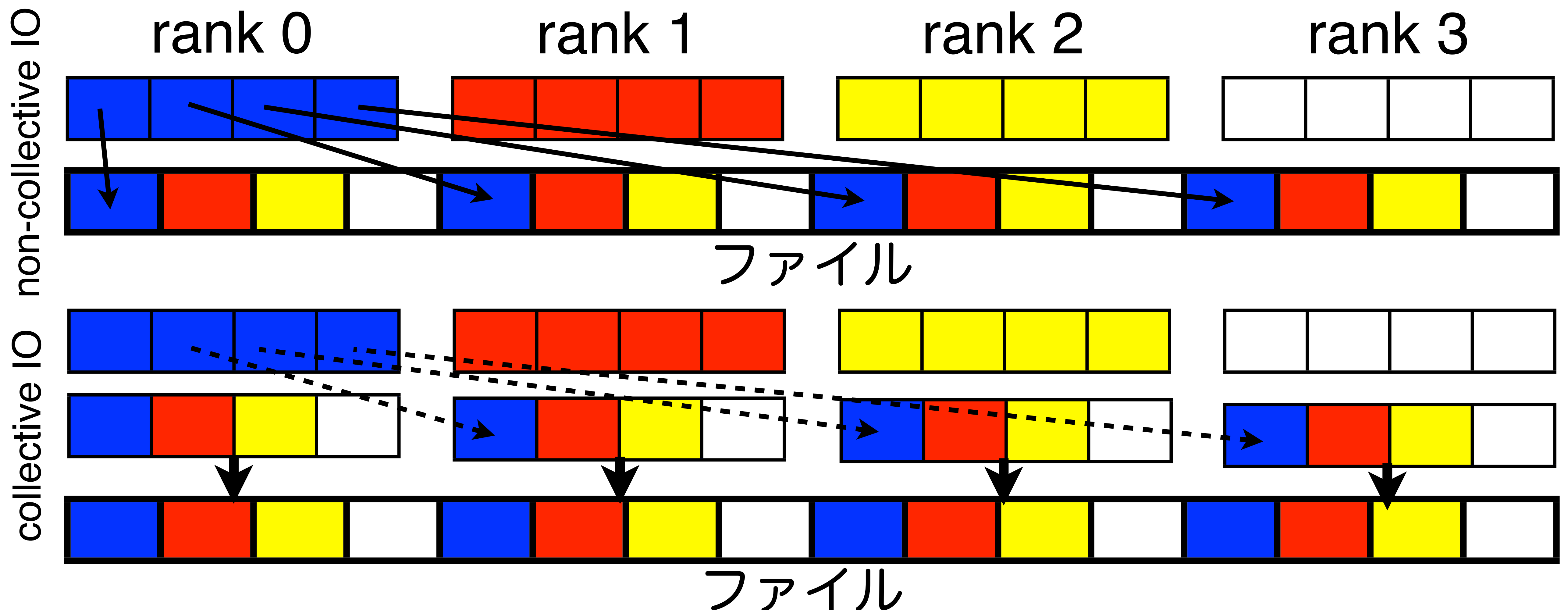


Rank数が大きくなる程、配列の次元数が大きくなる程、細かい IO になる



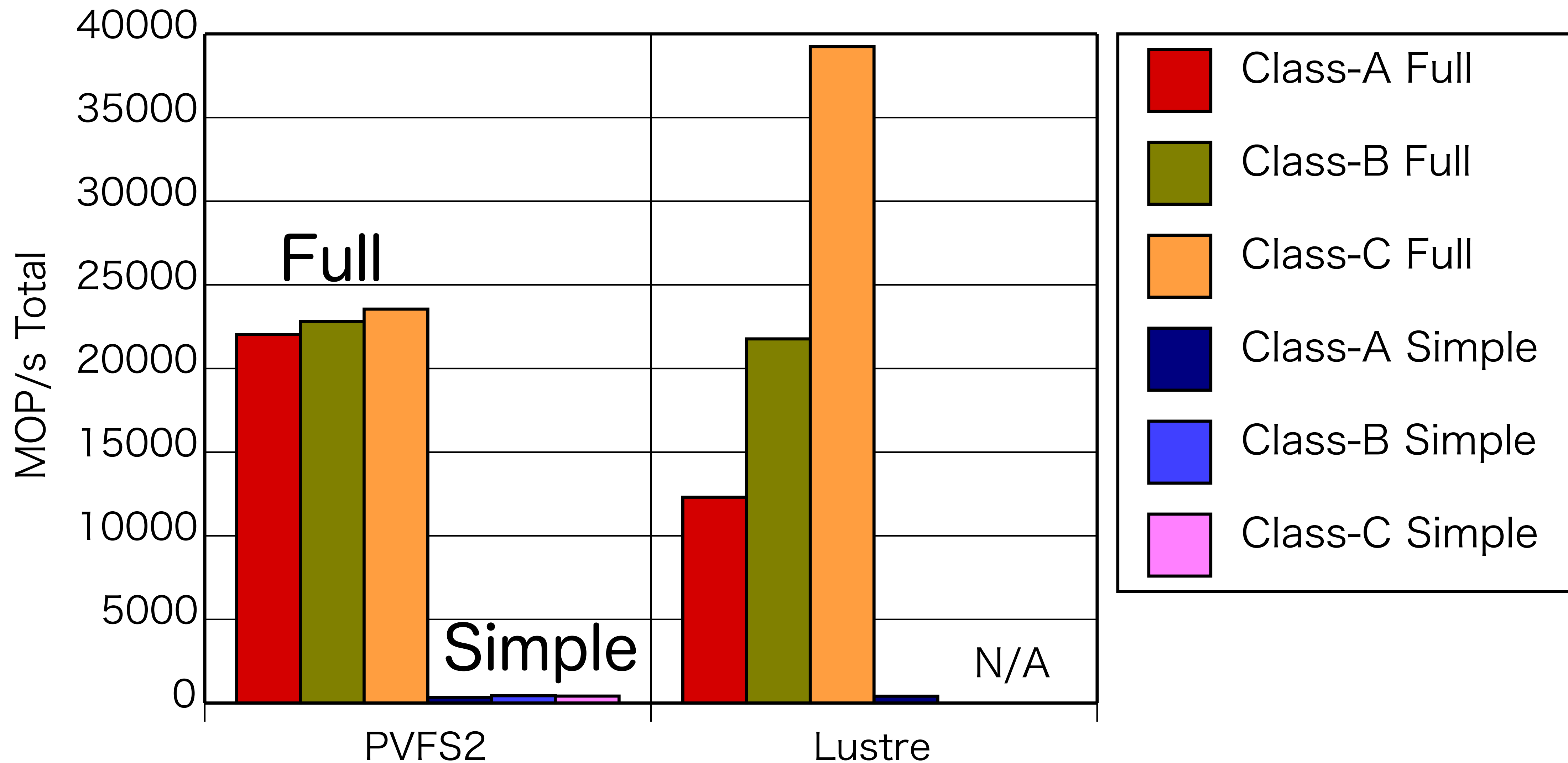
Collective IO

- 全てのノードが同時に IO することで、細かいデータをとりまとめることができ（MPI の通信）、結果として IO のデータ量を大きくすることができる。Derived Datatype も同時に使い、1 回の IO 呼出でより多くの範囲のデータを指定することが大事
- 同時に、ファイルにアクセスする場所を連続にすることが可能
- 結果として、IO 性能が大きく向上する



MPI-IO ベンチマーク

- BT-IO： MPI-IO ベンチマークプログラム
  - Class： 問題の大きさ（A < B < C）
  - Full： Colletive IO
  - Simple： Non-Collective IO



Collective I/O の問題点

BT-IO Full

```
      subroutine setup_btio

      include 'header.h'
      include 'mpinpb.h'

      integer ierr
      integer mstatus(MPI_STATUS_SIZE)
      integer sizes(4), starts(4), subsizes(4)
      integer cell_btype(maxcells), cell_ftype(maxcells)
      integer cell_blength(maxcells)
      integer info
      character*20 cb_nodes, cb_size
      integer c, m
      integer cell_disp(maxcells)

       call mpi_bcast(collbuf_nodes, 1, MPI_INTEGER,
     >                root, comm_setup, ierr)

       call mpi_bcast(collbuf_size, 1, MPI_INTEGER,
     >                root, comm_setup, ierr)

       if (collbuf_nodes .eq. 0) then
          info = MPI_INFO_NULL
       else
          write (cb_nodes,*) collbuf_nodes
          write (cb_size,*) collbuf_size
          call MPI_Info_create(info, ierr)
          call MPI_Info_set(info, 'cb_nodes', cb_nodes, ierr)
          call MPI_Info_set(info, 'cb_buffer_size', cb_size,
ierr)
          call MPI_Info_set(info, 'collective_buffering', 'true',
ierr)
       endif

       call MPI_Type_contiguous(5,
MPI_DOUBLE_PRECISION,
     $                          element, ierr)
       call MPI_Type_commit(element, ierr)
       call MPI_Type_extent(element, eltext, ierr)

       do  c = 1, ncells
c
c Outer array dimensions ar same for every cell
c
          sizes(1) = IMAX+4
          sizes(2) = JMAX+4
          sizes(3) = KMAX+4
c
c 4th dimension is cell number, total of maxcells cells
c
          sizes(4) = maxcells
c
c Internal dimensions of cells can differ slightly between
cells
c
          subsizes(1) = cell_size(1, c)
          subsizes(2) = cell_size(2, c)
          subsizes(3) = cell_size(3, c)
c
c Cell is 4th dimension, 1 cell per cell type to handle
varying
c cell sub-array sizes
c
          subsizes(4) = 1
c
c type constructors use 0-based start addresses
c
          starts(1) = 2
          starts(2) = 2
          starts(3) = 2
          starts(4) = c-1

c
c Create buftype for a cell
c
          call MPI_Type_create_subarray(4, sizes,
subsizes,
     $          starts, MPI_ORDER_FORTRAN, element,
     $          cell_btype(c), ierr)
c
c block length and displacement for joining cells -
c 1 cell buftype per block, cell buftypes have own
displacment
c generated from cell number (4th array dimension)
c
          cell_blength(c) = 1
          cell_disp(c) = 0

       enddo
c
c Create combined buftype for all cells
c
       call MPI_Type_struct(ncells, cell_blength, cell_disp,
     $            cell_btype, combined_btype, ierr)
       call MPI_Type_commit(combined_btype, ierr)

       do  c = 1, ncells
c
c Entire array size
c
          sizes(1) = PROBLEM_SIZE
          sizes(2) = PROBLEM_SIZE
          sizes(3) = PROBLEM_SIZE
c
c Size of c'th cell
c
          subsizes(1) = cell_size(1, c)
          subsizes(2) = cell_size(2, c)
          subsizes(3) = cell_size(3, c)

c
c Starting point in full array of c'th cell
c
          starts(1) = cell_low(1,c)
          starts(2) = cell_low(2,c)
          starts(3) = cell_low(3,c)

          call MPI_Type_create_subarray(3, sizes,
subsizes,
     $          starts, MPI_ORDER_FORTRAN,
     $          element, cell_ftype(c), ierr)
          cell_blength(c) = 1
          cell_disp(c) = 0
       enddo

       call MPI_Type_struct(ncells, cell_blength, cell_disp,
     $            cell_ftype, combined_ftype, ierr)
       call MPI_Type_commit(combined_ftype, ierr)

       iseek=0
       if (node .eq. root) then
          call MPI_File_delete(filenm, MPI_INFO_NULL,
ierr)
       endif

      call MPI_Barrier(comm_solve, ierr)

       call MPI_File_open(comm_solve,
     $          filenm,
     $          MPI_MODE_RDWR+MPI_MODE_CREATE,
     $          MPI_INFO_NULL, fp, ierr)

       if (ierr .ne. MPI_SUCCESS) then
              print *, 'Error opening file'
              stop
       endif

        call MPI_File_set_view(fp, iseek, element,
     $          combined_ftype, 'native', info, ierr)

       if (ierr .ne. MPI_SUCCESS) then
              print *, 'Error setting file view'
              stop
       endif

      do m = 1, 5
         xce_sub(m) = 0.d0
      end do

      idump_sub = 0

      return
      end
```

BT-IO Simple

```
      subroutine setup_btio
      include 'header.h'
      include 'mpinpb.h'

      integer m, ierr

      iseek=0

      if (node .eq. root) then
          call MPI_File_delete(filenm, MPI_INFO_NULL, ierr)
      endif

      call MPI_Barrier(comm_solve, ierr)

      call MPI_File_open(comm_solve,
     $          filenm,
     $          MPI_MODE_RDWR + MPI_MODE_CREATE,
     $          MPI_INFO_NULL,
     $          fp,
     $          ierr)

      call MPI_File_set_view(fp,
     $          iseek, MPI_DOUBLE_PRECISION, MPI_DOUBLE_PRECISION,
     $          'native', MPI_INFO_NULL, ierr)

      if (ierr .ne. MPI_SUCCESS) then
          print *, 'Error opening file'
          stop
      endif

      do m = 1, 5
         xce_sub(m) = 0.d0
      end do

      idump_sub = 0

      return
      end
```

コミュニケータを指定しない Collective な操作

- 片方向通信の Window や MPI-IO のファイルハンドル
- それらの生成時にコミュニケータを指定しているから
  - 生成時に指定されたコミュニケータを MPI_COMM_DUP して内部に持っている

ファイルアクセス 関数

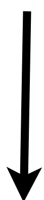
positioning	**synchronism**	**coordination**	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

File Pointer 問題

- 逐次プログラムの IO では、file pointer（アクセスする位置）がひとつだけ（暗黙に）ある
- 並列プログラムでは、複数のプロセスが同時に書込むために、file pointer の扱いに注意する必要がある

逐次

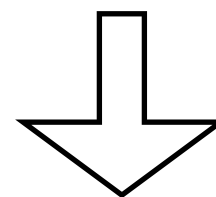
write(“abc”)
write(“def”)



a	b	c	d	e	f		

並列

Rank 0	Rank 1	Rank 2	Rank 3
write(“abc”) write(“def”)	write(“ghi”) write(“jkl”)	write(“mno”) write(“pqr”)	write(“stu”) write(“vwx”)



a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	

アクセスする場所の指定の仕方（3種類）

1. FP を使わないで、IO 時に常に場所を指定する
2. プロセス毎に FP が存在する
3. ファイルにひとつだけ FP が存在する
   ⇒ **遅い！！**

	Rank 0	Rank 1
1	write_at(“abc”, 0) write_at(“def”, 3)	write_at(“ghi”, 6) write_at(“jkl”, 9)
2	seek(0) write(“abc”) write(“def”)	seek(6) write(“ghi”) write(“jkl”)
3	seek(0) / barrier() barrier() / write(“def”) write(“abc”) / barrier()	barrier() / write(“ghi”) barrier() / write(“jkl”) barrier()

Resulting file (same for all three methods):

0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	c	d	e	f	g	h	i	j	k	l		

2種類の File Pointer

```
C: MPI_File_seek( MPI_File fh, MPI_Offset off, int whence )
   MPI_File_seek_shared( MPI_File fh, MPI_Offset off, int whence )
   MPI_File_get_position( MPI_File fh, MPI_Offset *off )
   MPI_File_get_position_shared (MPI_File *fh, MPI_Offset *off )
F: MPI_FILE_SEEK( fh, offset, whence, ierr )
   MPI_FILE_SEEK_SHARED( fh, off, whence, ierr )
   MPI_FILE_GET_POSITION( fh, off, ierr )
   MPI_FILE_GET_POSITION_SHARED( fh, off, ierr )
```

- MPI_FILE_SEEK
  - プロセス固有の FP を設定する
- MPI_FILE_SEEK_SHARED
  - 共有 FP を設定する
- MPI_FILE_GET_POSITION
  - プロセス固有の FP の値を得る
- MPI_FILE_GET_POSITION_SHARED
  - 共有 FP の値を得る

Write_at_all と Read_at_all

```
C:  MPI_File_write_at_all( MPI_File fh, MPI_Offset off, void *data,
        int count, MPI_Datatype type, MPI_Status status )
    MPI_File_read_at_all( MPI_File fh, MPI_Offset off, void *data,
        int count, MPI_Datatype type, MPI_Status status )
F:  MPI_FILE_WRITE_AT_ALL( fh, offset, data, count, type,
        status, ierr )
    MPI_FILE_READ_AT_ALL( fh, offset, data, count, type,
        status, ierr )
```

- 指定された Offset から、count 個数の type で指定されたデータ型の data を書込む／読込む。
- これらの関数では file pointer を使わない。

ファイルの操作

```
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
  - ファイルの削除
- MPI_FILE_PREALLOCATE
  - ファイルサイズを実際に書込む前に確保する
- MPI_FILE_GET_SIZE
  - ファイルの大きさを得る

MPI-IO のまとめ

- 3種類のアクセスする場所の指定方法
  - 陽に指定する MPI_File_***_at()
  - プロセス固有の File Pointer
    - File View の指定が可能
  - ファイルハンドルで共有される File Pointer
    - 遅いのでできるだけ使わないように
- Collective IO
  - Derived Datatype と Collective IO を組み合わせて使うと速くなることが多い

時間の計測

```
C: double MPI_Wtime( void )
F: DOUBLE PRECISION MPI_WTIME()
```

- 秒単位の時刻を返す

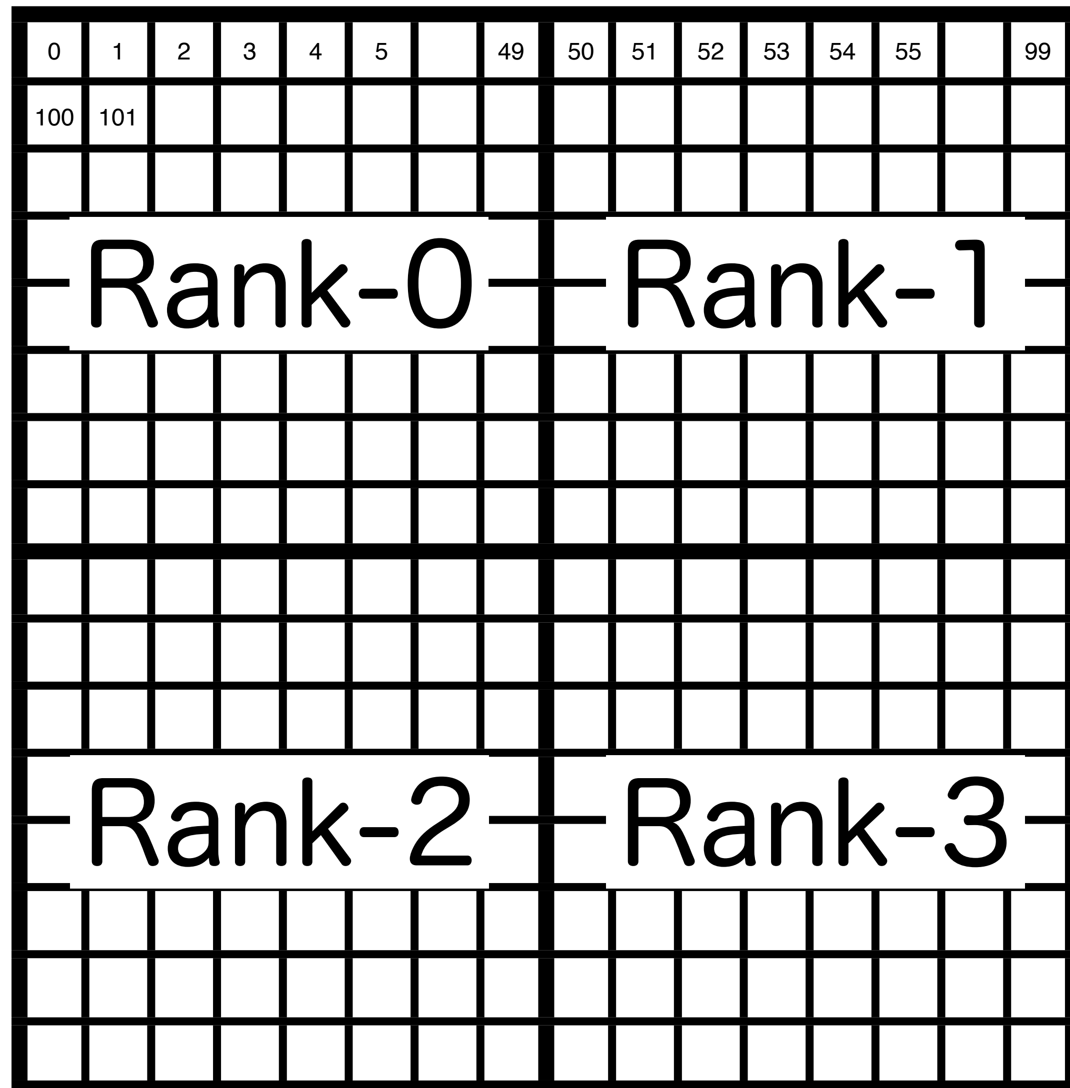
```
double dt, ts, te;

MPI_Barrier( MPI_COMM_WORLD );
ts = MPI_Wtime();
...
MPI_Barrier( MPI_COMM_WORLD );
te = MPI_Wtime();
dt = te - ts;
```

質問？

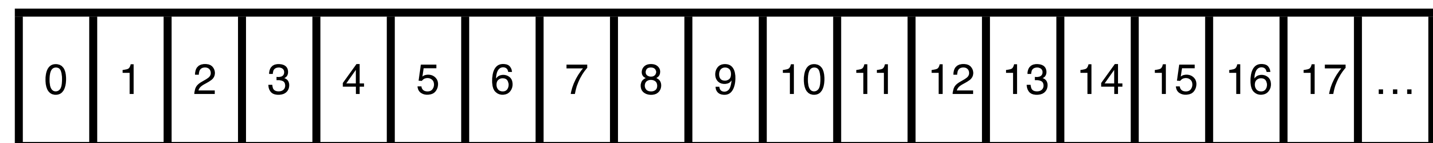
演習課題

- 4プロセス（ランク）のそれぞれが、下図に示す部分配列を持っていた時、MPI-IO を用いて出力するプログラムを作成せよ。



配列全体の大きさは100x100とする

ファイルの並びは以下の通り



演習課題

- 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
- 部分配列の要素の値は以下のようにする
  - Rank-0　11
  - Rank-1　13
  - Rank-2　15
  - Rank-3　17

余力のある人は

- 先ほど作ったファイルを、16プロセスで読込むプログラムを作れ。

プログラム例

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

nsizes[0] = LEN*2;
nsizes[1] = LEN*2;
nssizes[0] = LEN;
nssizes[1] = LEN;

switch( rank ) {
case 0:
  val = 11.0;
  starts[0] = 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] = LEN;
  starts[1] = LEN;
  break;
}

for( i=0; i<LEN; i++ ) {
  for( j=0; j<LEN; j++ ) {
    a[i][j] = val;
  }
}

MPI_Type_create_subarray( 2, nsizes, nssizes, starts,
                          MPI_ORDER_C, MPI_DOUBLE, &type );
MPI_Type_commit( &type );
MPI_Type_size( type, &tsz );
MPI_File_open( MPI_COMM_WORLD,
               "example.dat",
               MPI_MODE_CREATEIMPI_MODE_RDWR,
               MPI_INFO_NULL,
               &fh );
MPI_File_set_view( fh, 0, MPI_DOUBLE, type, "native",
          MPI_INFO_NULL );
MPI_File_write_at_all( fh, 0, a, LEN*LEN, MPI_DOUBLE,
          &status );
MPI_File_close( &fh );
```