MPI
コミュニケータとデータタイプ

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

MPI 上級編

- 午前：MPI における重要な概念
  - コミュニケータ
  - データタイプ
- 午後：MPI-IO
  - 集団 I/O
  - MPI-IO 演習

コミュニケータ

コミュニケータ

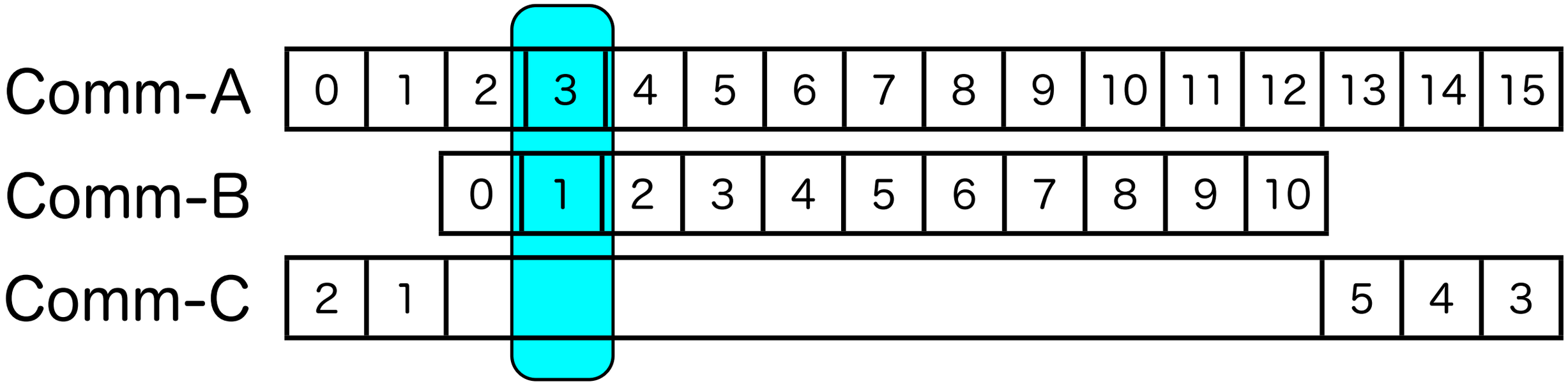
- コミュニケータは主に集団的な操作において操作の対象となる「集団」を示すもの
  - MPI における集団的な操作
    - 集団通信（Collective Communication）
      - 集団 I/O（Collective I/O Operation）
        - 非集団 I/O もあるにはあるけど…
    - Window 操作
- （現時点で）MPI 固有な概念（多分）

ランク番号（再考）

```
C:      int MPI_Comm_rank( MPI_Comm comm, int *rank )
        int MPI_Comm_size( MPI_Comm comm, int *size )
F:      MPI_COMM_RANK( communicator, rank, ierr )
        MPI_COMM_SIZE( communicator, size, ierr )
```

- MPI_COMM_RANK
  - 指定されたコミュニケータにおける自プロセスのランク番号を得る（最初はゼロ）
- MPI_COMM_SIZE
  - 指定されたコミュニケータにおけるランクの数を得る

コミュニケータの例



- プロセスは複数のコミュニケータに属することができる
  - Comm-A：Comm_rank = 3, Comm_size = 16
  - Comm-B：Comm_rank = 1, Comm_size = 11
  - Comm-C：Comm_rank = ?, Comm_size = ?

Group と Communicator

- (Process) Group
  - プロセスの順序集合
- Communicator (C言語の型：MPI_Comm)
  - 通信の対象となるプロセスグループ
  - 通信の状態を保持する
  - 送受信のマッチ
    - Source/Destination, Tag, Communicator
  - Pre-defined communicator
    - **MPI_COMM_WORLD**：全体
    - **MPI_COMM_SELF**：自分自身のプロセス

Communicator の生成と開放

```
C: MPI_Comm_dup( MPI_Comm comm,  MPI_Comm *new )
   MPI_Comm_free( MPI_Comm *comm )
F: MPI_COMM_DUP( comm, new, ierr)
   MPI_COMM_FREE( comm, ierr )
```

- Group の生成と Group から Communicator を生成する方法 - 省略
- MPI_Comm_dup：複製を作る
  - 同じプロセスグループだが違うコミュニケータを生成する
  - 違うコミュニケータを使うことで、通信を分離できる
- MPI_Comm_free：開放する
  - この関数を呼んだ後で、このコミュニケータを使うことはできない

Communicator の分割

C: MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *new)
F: MPI_COMM_SPLIT(comm, color, key, new, ierr)

- Communicator comm を 同じ color を持つ（複数の、オーバラップのない）communicator に分割する。分割された communicator における rank 番号は、key の値の小さい順に割り当てられる。Key の値が同じ場合は、システムが適当に rank 番号を割り当てる。

MPI_COMM_SPLITの実行例

C: MPI_Comm_split(comm, color, key, *new_comm)
F: MPI_COMM_SPLIT(COMM, color, key, new, ierr)

Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Color	1	1	1	1	2	2	2	2	7	7	7	7	4	4	4	4
Key	**2**	**2**	5	0	2	6	1	0	2	7	1	0	2	8	1	9

元Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
新Comm	comm0				comm1				comm2				comm3			
新Rank	**1**	**2**	3	0	2	3	1	0	2	3	1	0	1	2	0	3

2	**1**

となる場合もある

データタイプ

データタイプ

- Data Type（データの型）
  - Basic Data Type（基本データ型）
  - Derived Data Type（派生データ型）
    - 基本データ型の組み合わせ and/or
    - 複数の同じ基本データ型

MPIにおける基本データ型

C言語のデータ型との対応

MPI datatype	C datatype
MPI_CHAR	`char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_C_BOOL	`_Bool`
MPI_INT8_T	`int8_t`
MPI_INT16_T	`int16_t`
MPI_INT32_T	`int32_t`
MPI_INT64_T	`int64_t`
MPI_UINT8_T	`uint8_t`
MPI_UINT16_T	`uint16_t`
MPI_UINT32_T	`uint32_t`
MPI_UINT64_T	`uint64_t`
MPI_C_COMPLEX	`float _Complex`
MPI_C_FLOAT_COMPLEX (as a synonym)	`float _Complex`
MPI_C_DOUBLE_COMPLEX	`double _Complex`
MPI_C_LONG_DOUBLE_COMPLEX	`long double _Complex`
MPI_BYTE	
MPI_PACKED	

FORTRAN言語のデータ型との対応

MPI datatype	Fortran datatype
MPI_INTEGER	`INTEGER`
MPI_REAL	`REAL`
MPI_DOUBLE_PRECISION	`DOUBLE PRECISION`
MPI_COMPLEX	`COMPLEX`
MPI_LOGICAL	`LOGICAL`
MPI_CHARACTER	`CHARACTER(1)`
MPI_BYTE	
MPI_PACKED	

処理系によってはMPI_DOUBLE_COMPLEX等もサポートしている場合もある

C言語とFORTRAN言語両方に対応するデータ型

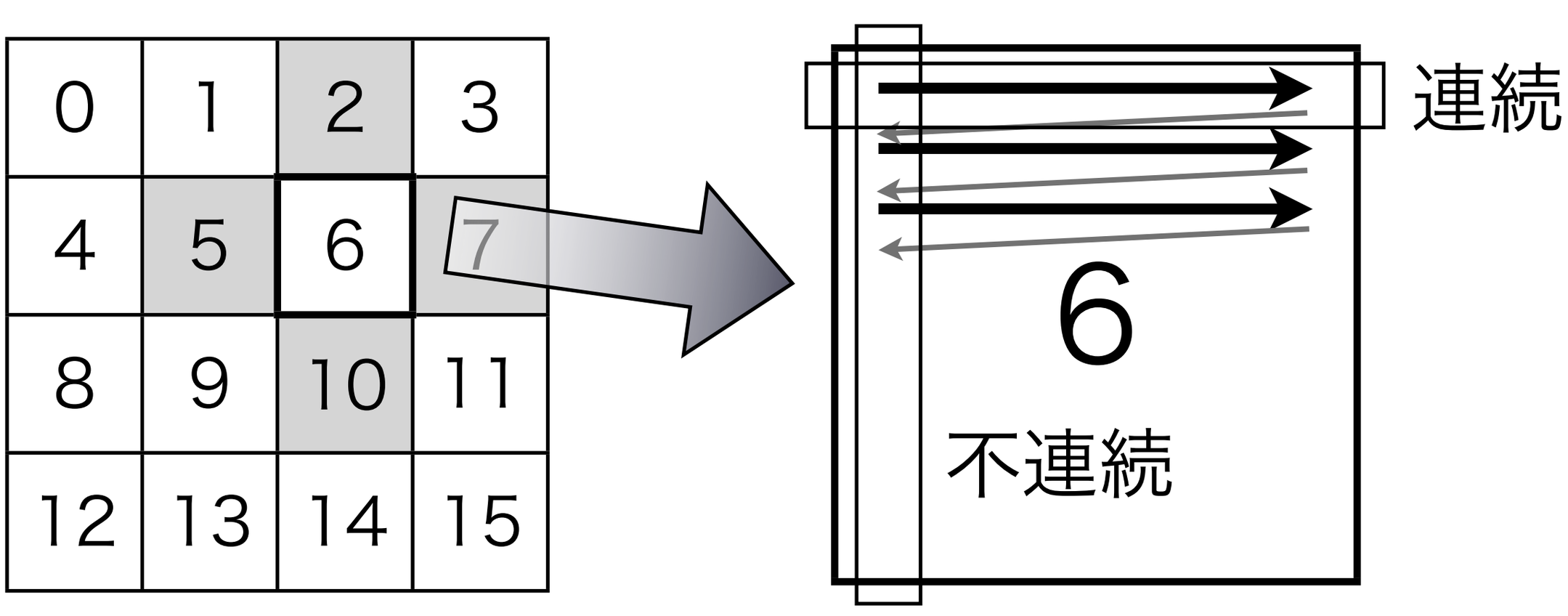
MPI datatype	C datatype	Fortran datatype
MPI_AINT	`MPI_Aint`	`INTEGER (KIND=MPI_ADDRESS_KIND)`
MPI_OFFSET	`MPI_Offset`	`INTEGER (KIND=MPI_OFFSET_KIND)`

Derived Datatype

- 派生データ型（Derived Datatype）
- 不連続なデータを、ひとまとめに表現することで、不連続なデータの通信等の高速化への対処（MPI の実装）を可能とする
- Basic (Predefined) Datatype と（配列内の）オフセットの並びで表現される

例：多次元配列のブロック分割

- 多次元配列の場合、あるひとつの次元のみデータの並びが連続である ⇒ 他の次元の並びは不連続
- 例えば、ステンシル計算において隣のブロックと halo 領域を交換しようとすると、不連続なデータの通信が発生する



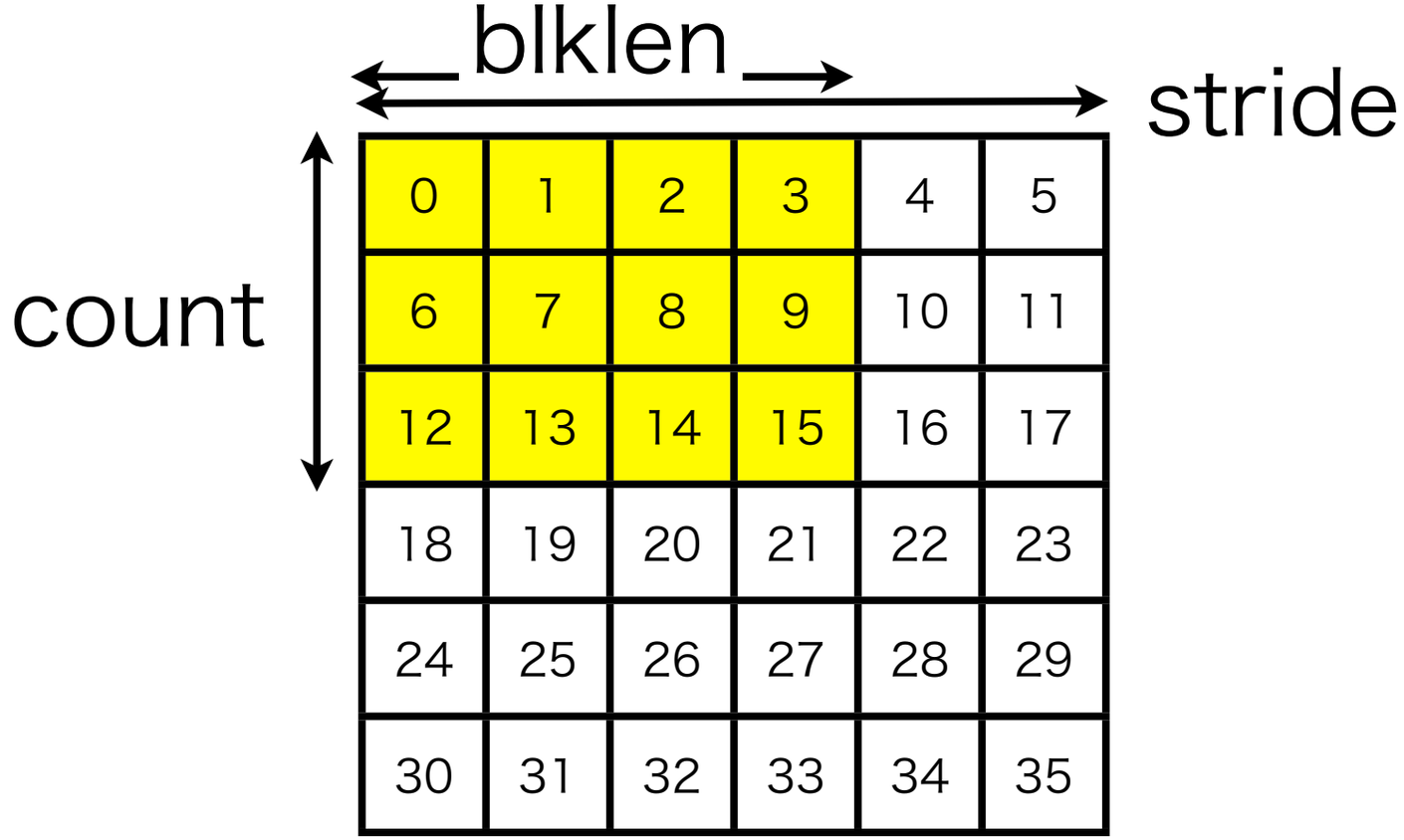
Derived Datatype の定義

- 以下の関数が用意されている
  - MPI_Type_contiguous
  - MPI_Type_vector
  - MPI_Type_indexed
  - MPI_Type_create_indexed_block
  - MPI_Type_create_subarray
  - MPI_Type_create_darray など
- 定義された「派生データ型」は commit して初めて使うことができる
  - **MPI_Type_commit <<<< 忘れないように！**

MPI_Type_vector

C: MPI_Type_vector(int count, int blklen, in stride,
MPI_Datatype otype, MPI_Datatype *ntype)
F: MPI_TYPE_VECTOR(count, blklen, stride,
otype, ntype, ierr)

MPI_Type_vector(3, 4, 6, MPI_DOUBLE, &newtype)



MPI_Type_create_subarray (1)

C: MPI_Type_create_subarray(int ndims, int array_of_sizes[], in array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype otype, MPI_Datatype *ntype)

F: MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, otype, ntype, ierr)

ndims: 元となる配列の次元数

array_of_sizes: それぞれの次元の大きさ

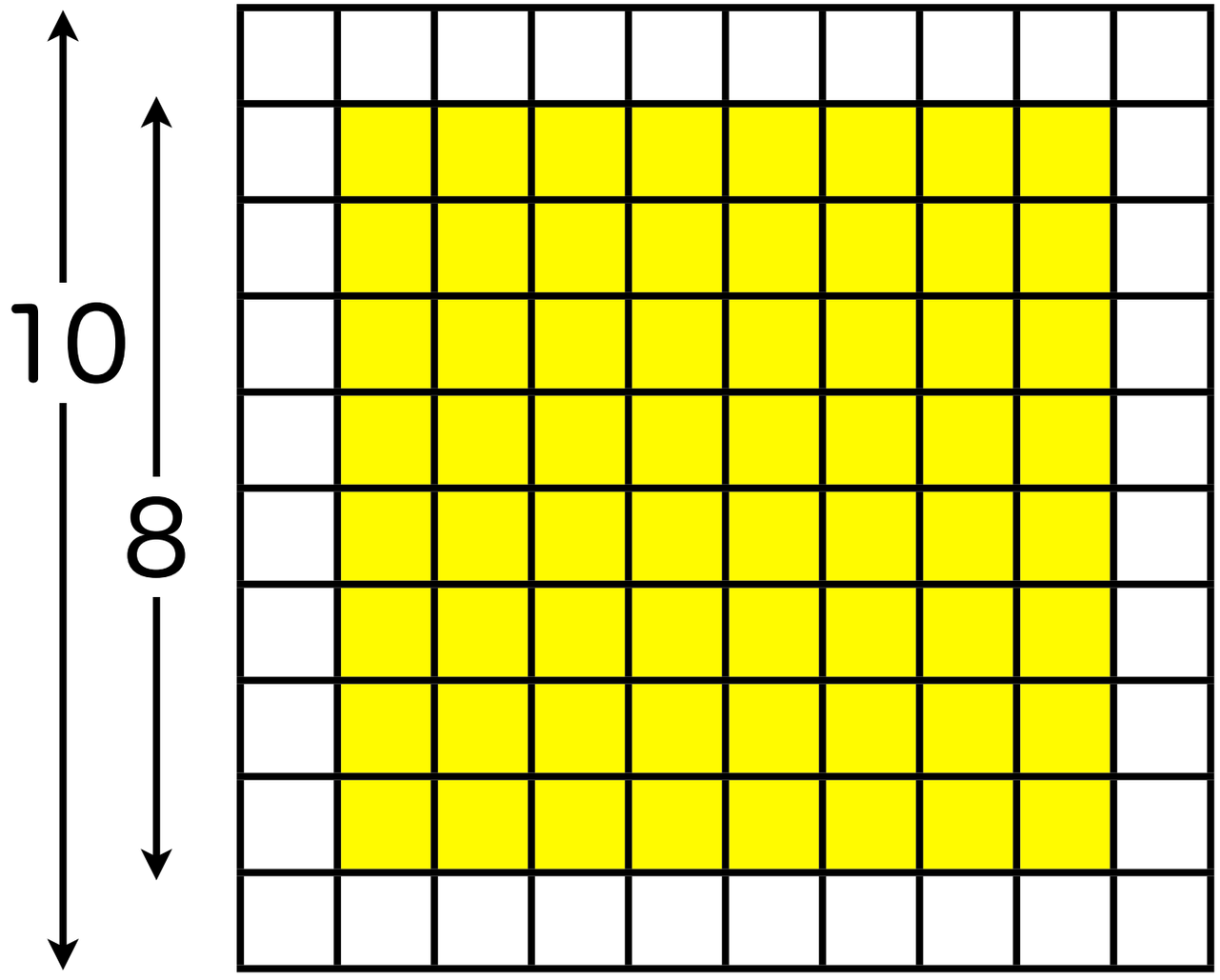
array_of_subsizes: 部分配列の大きさ

array_of_starts: 部分配列の始まりFORTRANでも0から始める

order: MPI_ORDER_C又はMPI_ORDER_FORTRAN

MPI_Type_create_subarray (2)

MPI_Type_create_subarray(2, [10,10], [8,8], [1,1], MPI_ORDER_C, MPI_DOUBLE, &ntype)



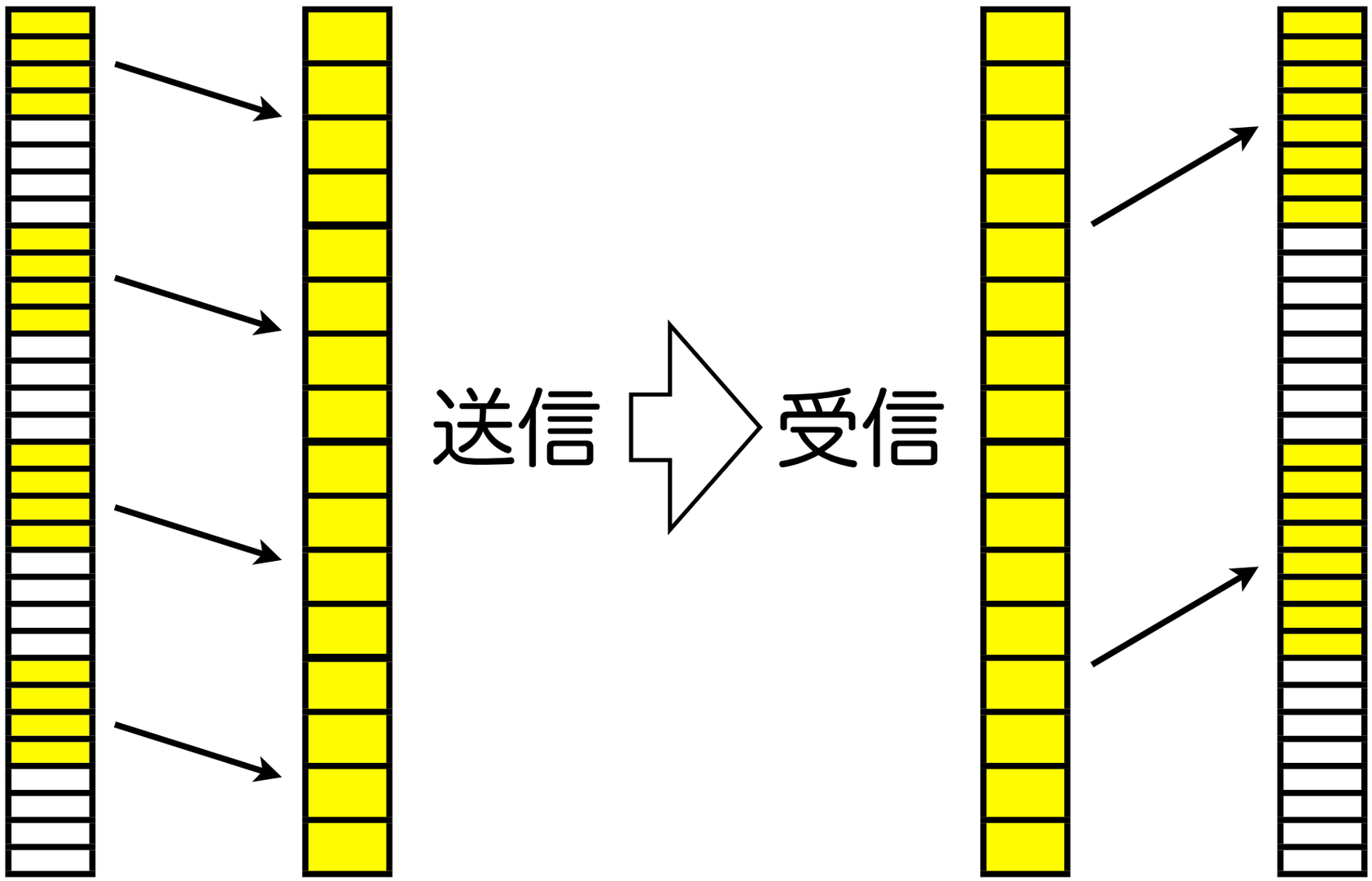
MPI_Type_commit

C: MPI_Type_commit(MPI_Datatype type)
F: MPI_TYPE_COMMIT(type, ierr)

作成した DataType をコミットする。これにより、以後、通信等でこの DataType を使うことができる。MPI_DOUBLE 等は予めコミットされている。

Derived Datatype の内部処理

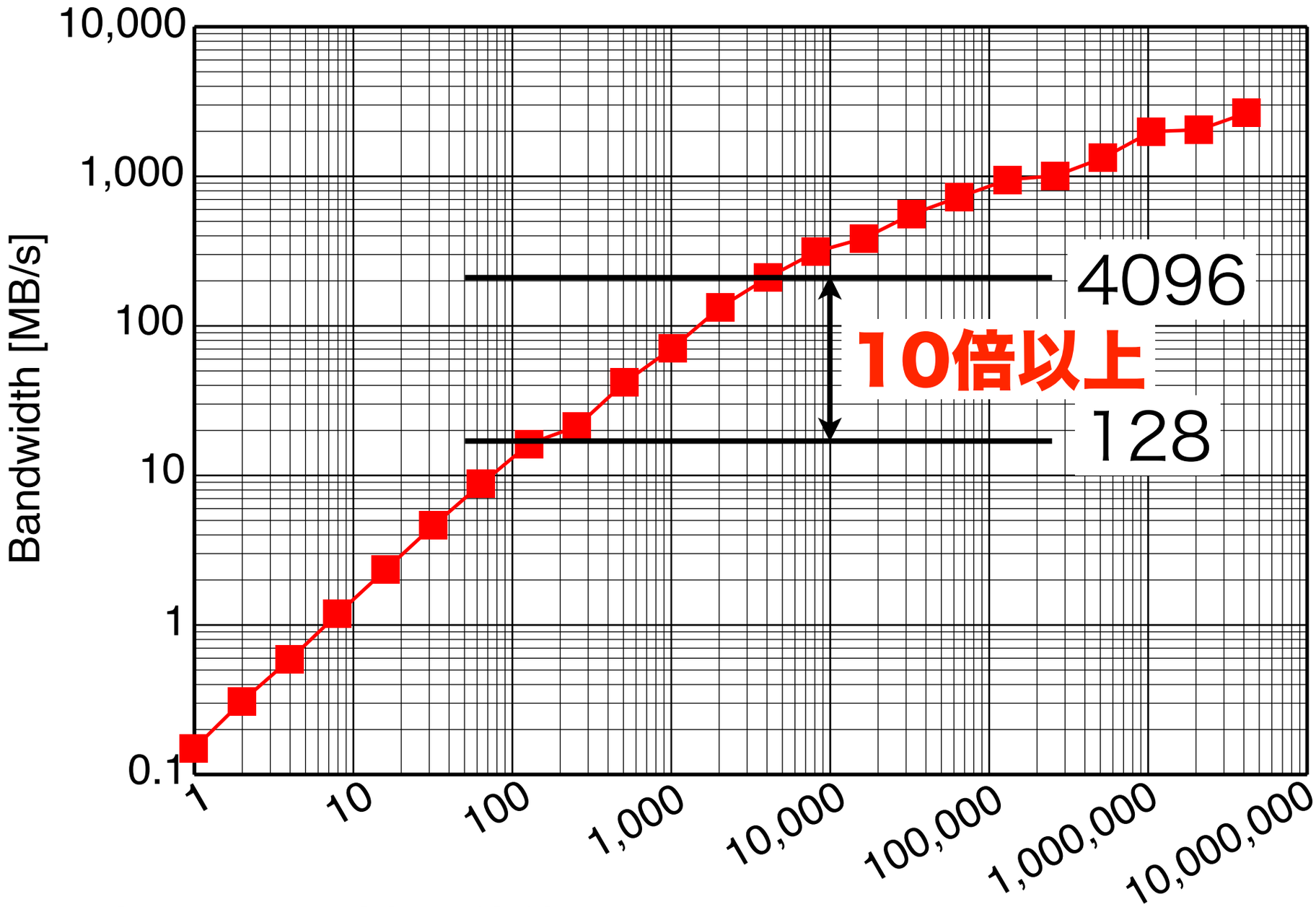
- 不連続なメモリ領域をいったん連続領域に “pack” して、それを送信し、受信側では連続領域をバラバラ（“unpack”）にする。



メッセージ通信の特性

- 128バイトを10回送る（1,280バイト）より
  4,096バイトを１回で送った方が速い

Intel Nehalem
(2.67 GHz)
Infiniband QDR
MPICH-SCore



Derived Datatype のまとめと注意

- 基本データ型から派生データ型を生成
- 生成した後で commit すること
- 作られたデータ型は、MPI の中の通信や IO 等で使うことができる
- 不連続なメモリ領域をひとまとめに処理できる
  - **ただし、これで実際に「通信が高速」になるかどうかは MPI の実装や、機種に依存する**
  - **一方、MPI-IO の多くの場合は、派生データ型を用いることで高速化が可能**

ただし「京」の場合

- 「京」では一般的に **Hybrid MPI** が用いられている
  - ノード内の8コアは、コンパイラの自動並列化あるいは OpenMP で並列化される
  - 逆に MPI の呼出は逐次処理になる
- Derived Datatype の pack/unpack は並列化されないので1コアでしか動作しない
- Pack/unpack するプログラムを**陽に**（例えば OpenMP）書いてノード内並列化した方が速い（場合が多い）

質問？