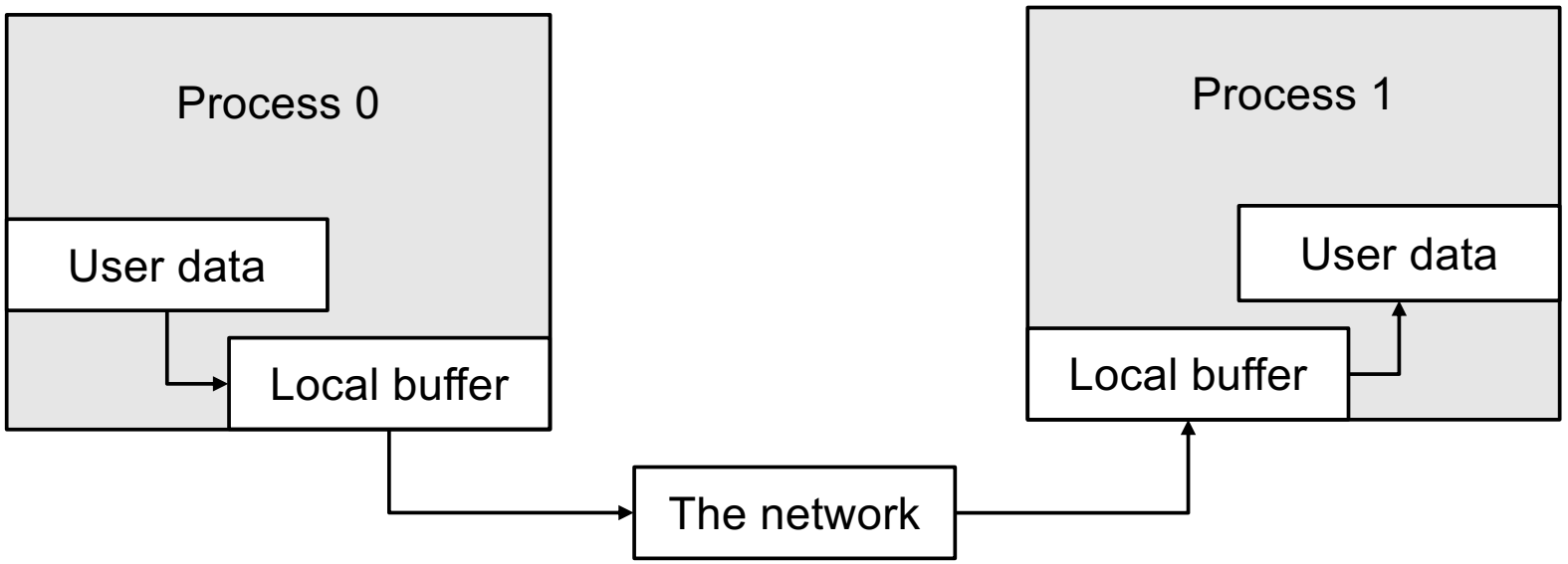
Notes on Buffering in MPI

Buffering in MPI (I)

- When you send data, where does it go? One possibility is:



Buffering in MPI (II)

- Message buffering decouples send/receive operations.
- Allows `MPI_Send` to return before data has actually been transferred.
- Additional overhead for memory-memory copying.
- Amount of buffering is application and implementation dependent:
  - Application can choose communication modes – and gain finer control (with additional hazards) over messaging behavior.
  - The standard mode is implementation dependent.
- A *properly coded* (“safe”) program will not fail if the buffer throttles back on the sends, thereby causing blocking. Program is safe, if it still works when standard sends (MPI_Send) are replaced by synchronous send (MPI_Ssend)
- An *improperly coded* program may deadlock.

Buffering in MPI (III)

- Implementation may buffer on sending process, receiving process, both, or none.
- In practice, tend to buffer “small” messages on receiving process (message = envelope + data):
  - *Eager* protocol: message is sent assuming destination can store it
  - *Short* protocol: message data sent within envelope
- Otherwise:
  - *Rendezvous* protocol: message not sent until destination sends OK after request.

Buffering in MPI (IV)

- Many MPI implementations use both the *eager* and *rendezvous* methods of message delivery.
- Switch between the two methods according to message size.
- Often the cut-over point is controllable via a environment variables, e.g.
  - `MPI_BUFFER_MAX` and `MPI_DEFAULT_SINGLE_COPY_OFF` for SGI MPT
  - `MP_EAGER_LIMIT` and `MP_USE_FLOW_CONTROL` for IBM PE.
  - `btl_tcp_eager_limit` for OpenMPI

Use of Buffered Send (I)

- For buffered send, the programmer cannot assume any pre-allocated or sufficient buffer space for buffered send.
- Programmer must explicitly attach enough buffer space for the program:

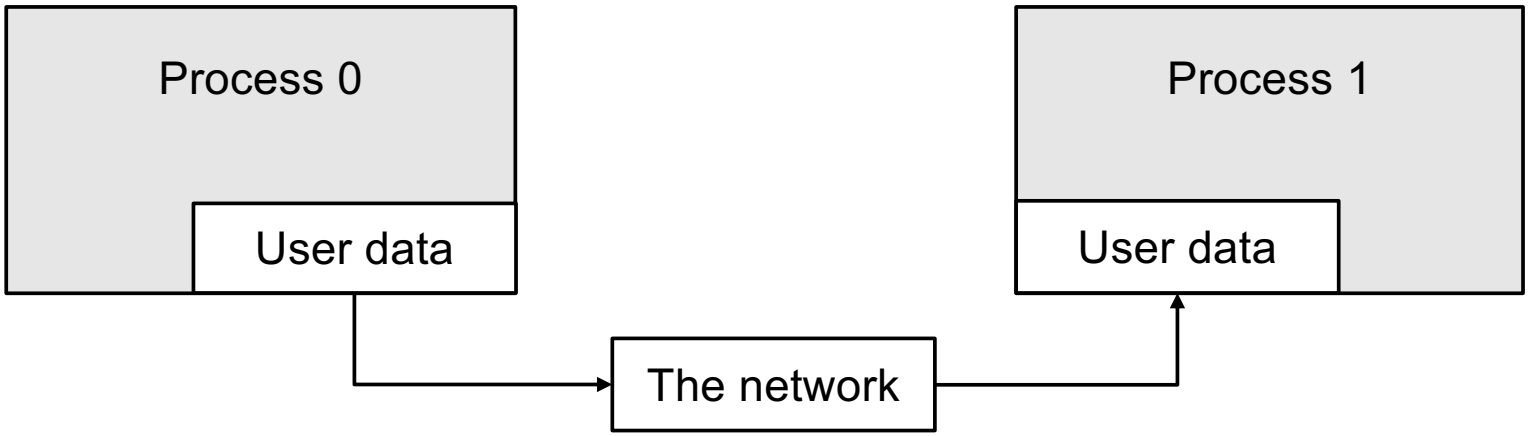
```
MPI_Buffer_attach(buf, ... )
MPI_Bsend(sendbuf, ... )
MPI_Buffer_detach(buf, ... )
```

Use of Buffered Send (II)

- *Safe* MPI programs do not rely on system buffering for success.
- Any system will eventually run out of buffer space as message buffer sizes are increased.
- Users are free to take advantage of knowledge of an implementation's buffering policy to increase performance, but they do so by relaxing the margin for safety (as well as decreasing portability, of course).

Avoid Buffering

- It is better to avoid copies:

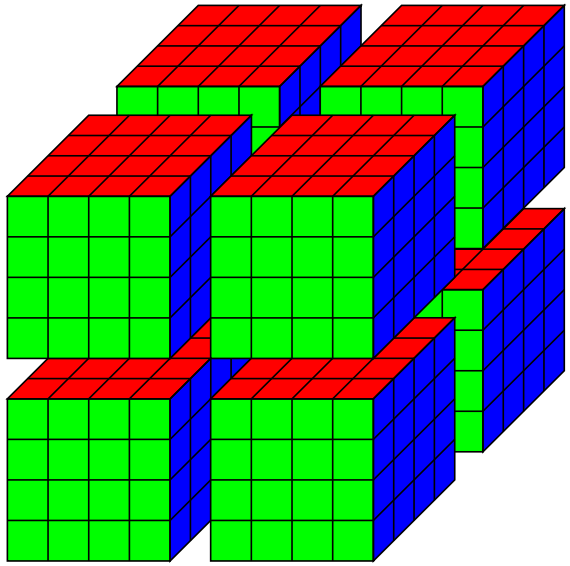


This requires that `MPI_Send` waits on delivery, or that `MPI_Isend` returns before transfer is complete and we wait later.

Derived Datatypes

Motivation for Derived Datatypes

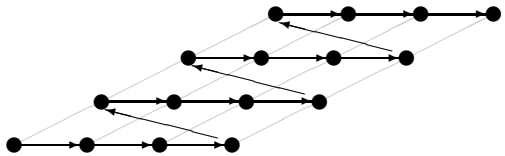
- Consider a 3D cube domain
  - distributed over 8 processes



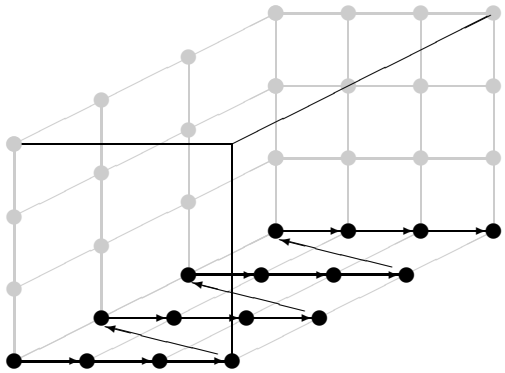
Motivation for Derived Datatypes

Storage sequence of multi-dimensional arrays

Fortran

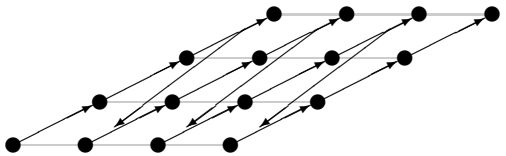


```
real(8) :: v(Nx, Ny)
```

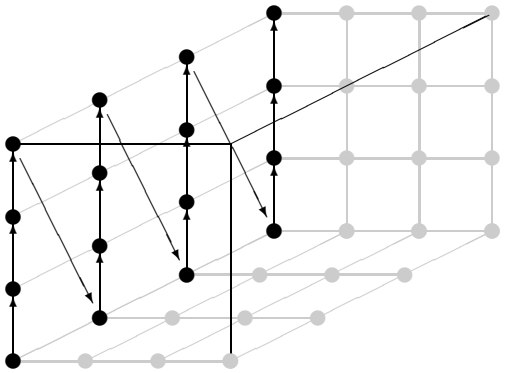


```
real(8) :: v(Nx, Ny, Nz)
```

C



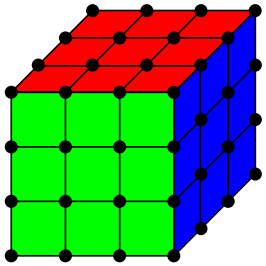
```
double v[Nx][Ny];
```

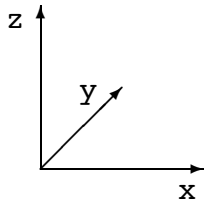


```
double v[Nx][Ny][Nz];
```

Motivation for Derived Datatypes

- Exchange of boundary data
- 3D example: surface of a cube

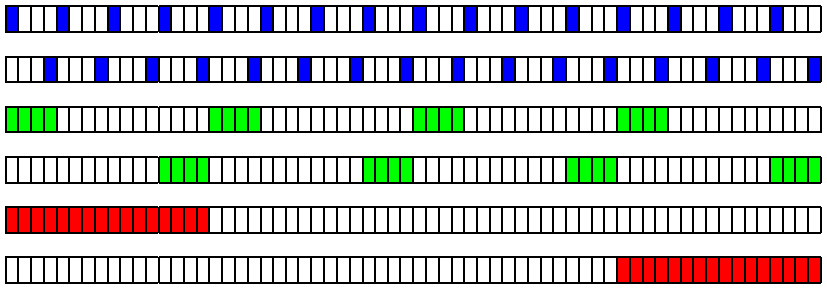





```
x = 0 .. Nx-1
y = 0 .. Ny-1
z = 0 .. Nz-1
```

- Storage in memory

Surface



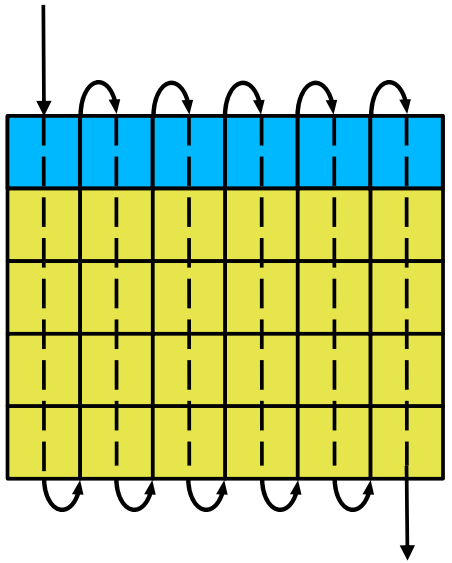
	count	*block*	*stride*	*offset*
x ≡ xMin	Ny*Nz	1	Nx	0
x ≡ xMax	Ny*Nz	1	Nx	Nx-1
y ≡ yMin	Nz	Nx	Nx*Ny	0
y ≡ yMax	Nz	Nx	Nx*Ny	Nx*(Ny-1)
z ≡ zMin	1	Nx*Ny	Nx*Ny*Nz	0
z ≡ zMax	1	Nx*Ny	Nx*Ny*Nz	Nx*Ny*(Nz-1)

Motivation for Derived Datatypes

- Optimal message construction for mixed data types
  - Examples so far:
    - uniform type
    - contiguous in memory
  - real world?
- Sending messages of different types separately?
  - requires considerable overhead (esp. for small messages)
  - leading to inefficient message passing

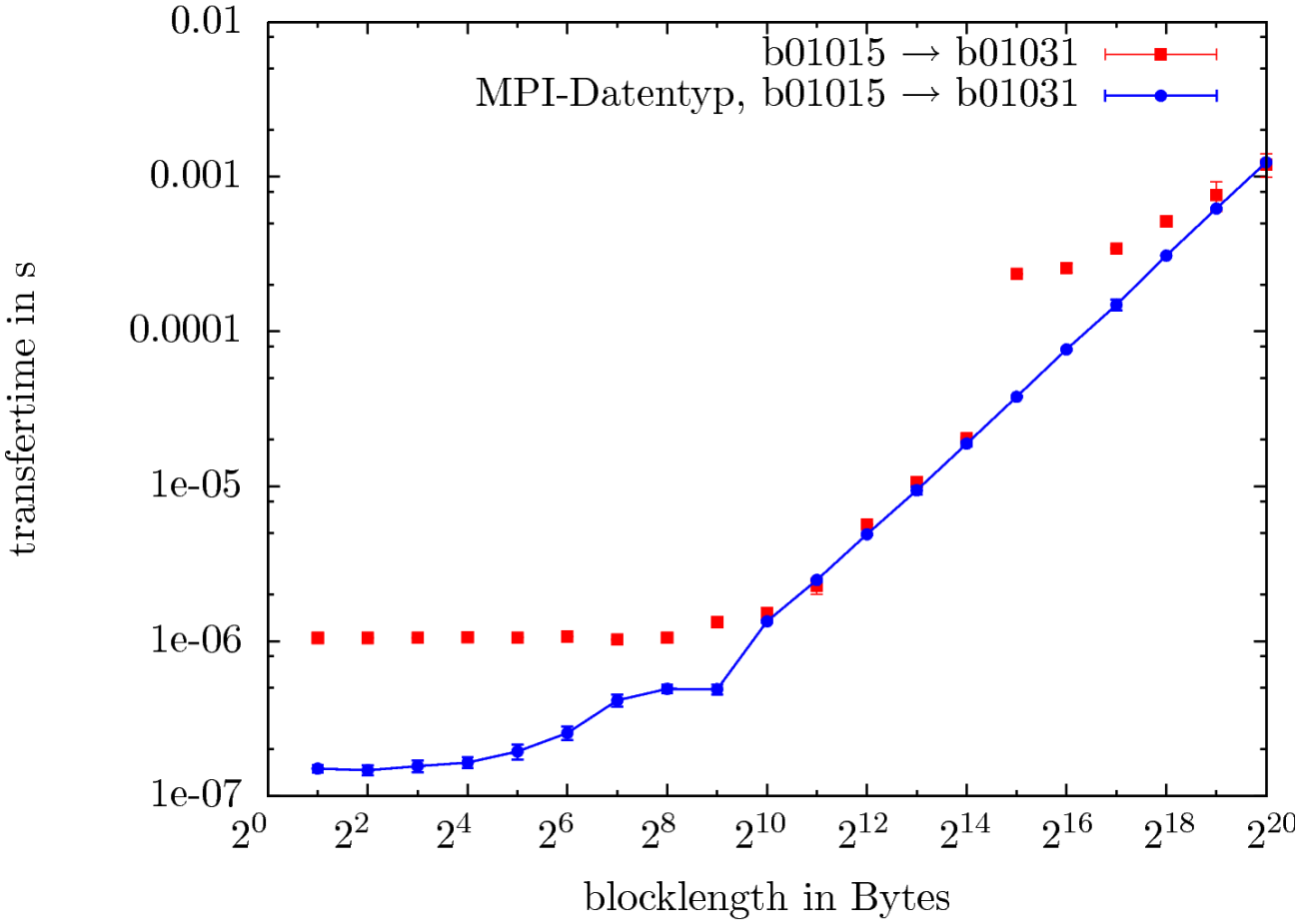
Derived Datatypes

- Examples



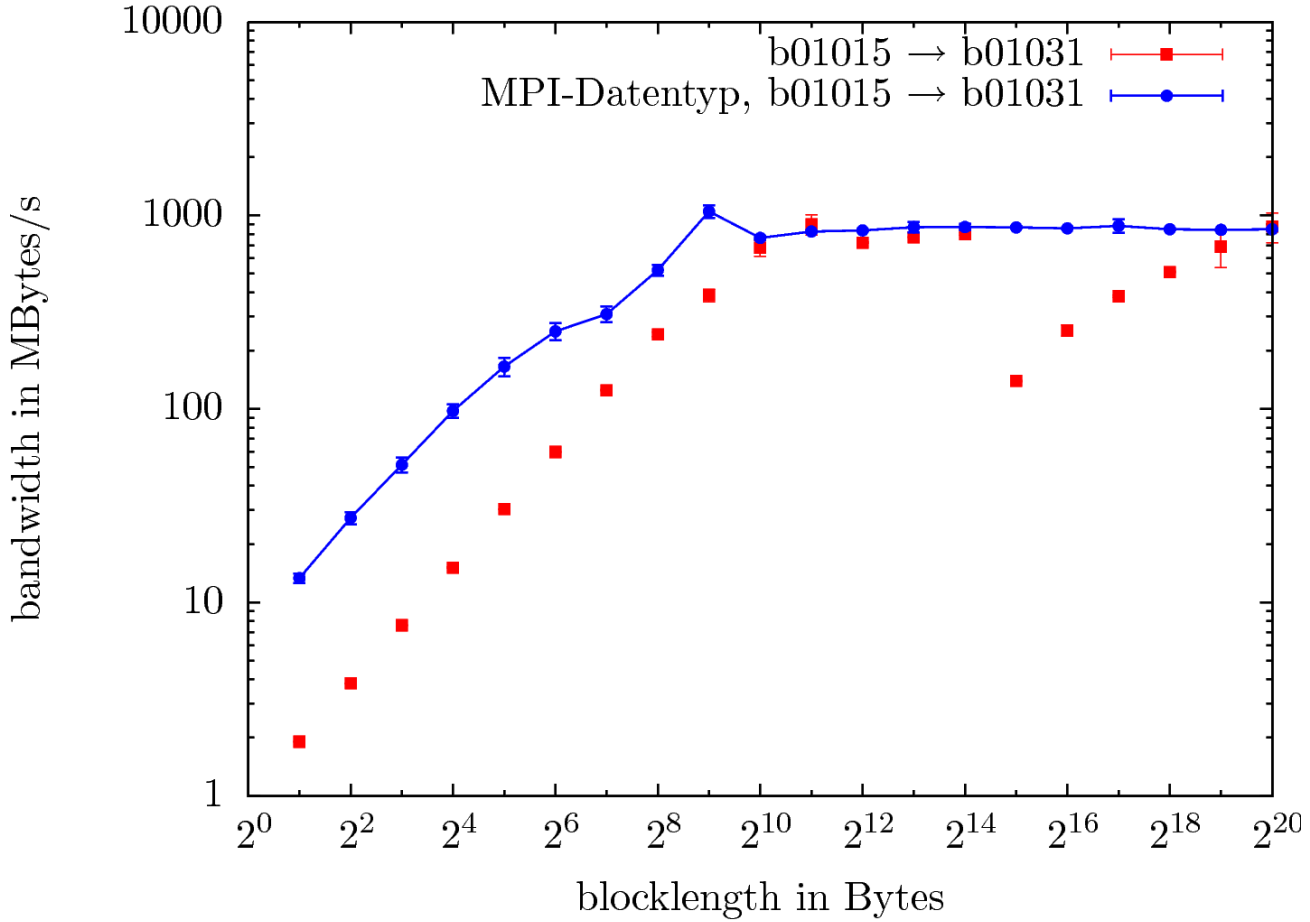
MPI Datatype Performance Example (I)

- Domain decomposition boundary exchange transfer time



MPI Datatype Performance Example (I)

- Domain decomposition boundary exchange Transfer Bandwidth

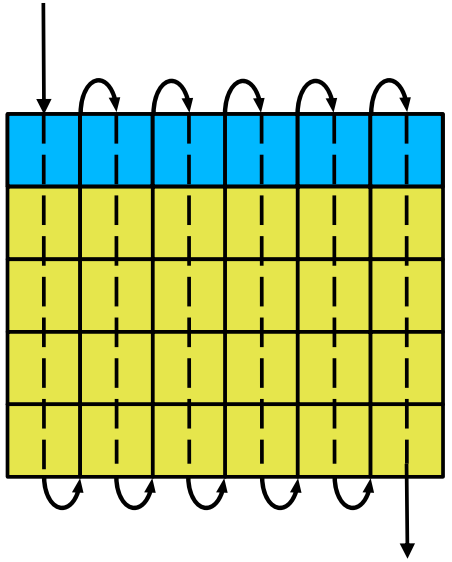


MPI Datatypes (I)

- Since all data is labeled by a datatype, an MPI implementation can support heterogeneous communication.
  - heterogeneous: between processes on machines with different memory representations and lengths of elementary data types.
- Specifying application-oriented layout of data in *memory*
  - can reduce mem-to-mem copies in the implementation
  - allows use of special hardware (gather/scatter) if available
- Specifying application-oriented layout of data on a *file*
  - can reduce system calls and physical disk I/O

MPI Datatypes (II)

- MPI provides functions to construct custom datatypes, such as an array of (int, float), or a row of a matrix stored column-wise.



MPI Datatypes (III)

- The data in a message to be sent or received is described by a triple (address, count, datatype).
- An MPI *datatype* is recursively defined as
  - predefined, corresponding to a datatype from the programming language (e.g. `MPI_INT, MPI_REAL`)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes

Creating a Derived Datatype

2 steps:

1. Construct datatype
2. Commit datatype

When no longer needed:

3. free datatype

Derived Datatypes - *Type Maps* (I)

- **Type map** = template for the derived datatype, description of layout in memory; consists of
  - *primitive types* and
  - integer (byte) *displacements*

$$\text{typemap} = \{(\text{type}_0, \text{disp}_0), (\text{type}_1, \text{disp}_1), \ldots, (\text{type}_{N-1}, \text{disp}_{N-1})\}$$

- **Displacement** = offset from the start of the communication buffer **in bytes** (**not** necessarily positive, or ordered)
- **Type signature**: sequence of primitive types:

$$\text{typesig} = \{\text{type}_0, \text{type}_1, \ldots, \text{type}_{N-1}\}$$

Derived Datatypes - *Type Maps* (II)

- Note: predefined MPI datatypes are just special cases of a derived datatype.
  - `MPI_FLOAT` is a predefined handle to a datatype with typemap?

typemap = {(float, 0)}

Contiguous Data

- The simplest derived datatype consists of a number of contiguous items of the same datatype
- C:

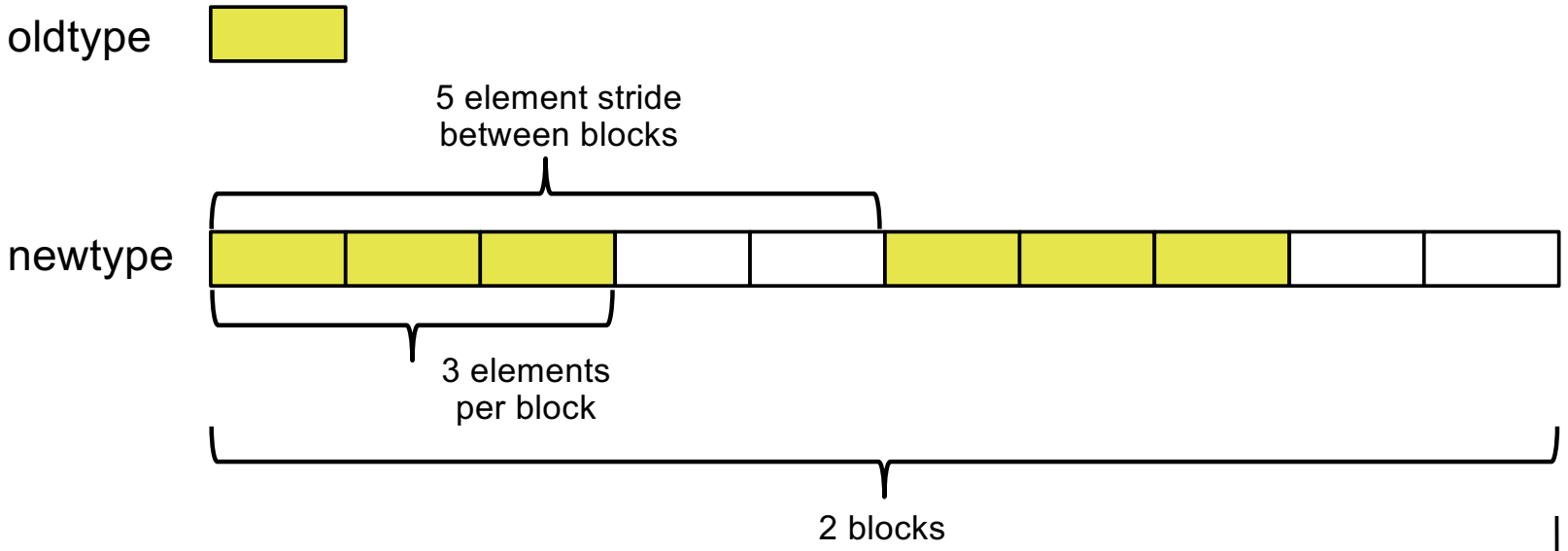
```
int MPI_Type_contiguous (int count,
      MPI_Datatype oldtype,
      MPI_Datatype *newtype)
```

- Fortran:

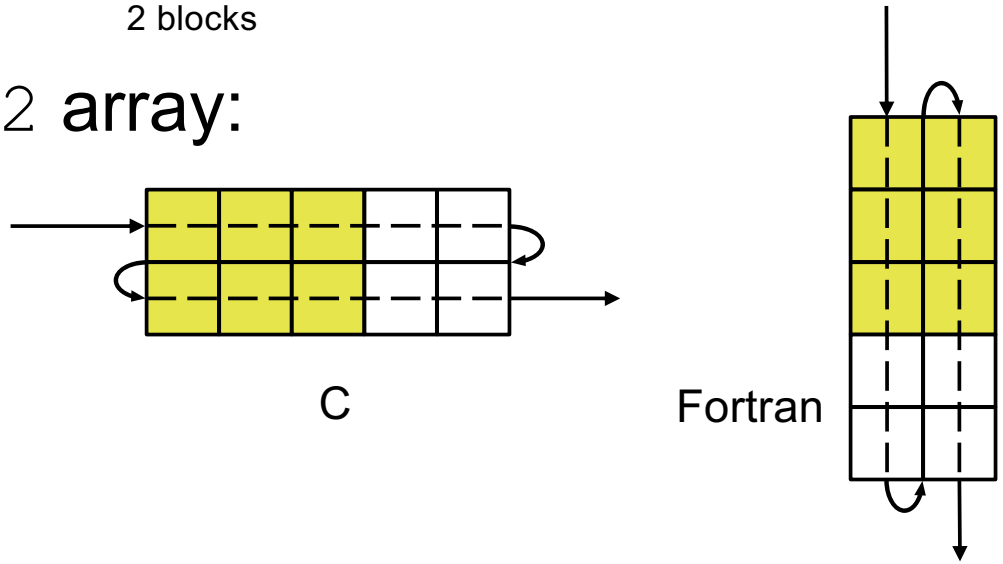
```
MPI_TYPE_CONTIGUOUS (COUNT, OLDTYPE,
     NEWTYPE, IERROR)
  INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

Type map for 5 floats?

Vector Datatype Example



- `3x2` sub-array of an `5x2` array:
  - count = 2
  - stride = 5
  - blocklength = 3



Constructing a Vector Datatype

- C:

```
int MPI_Type_vector (int count,
                     int blocklength, int stride,
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```

- Fortran:

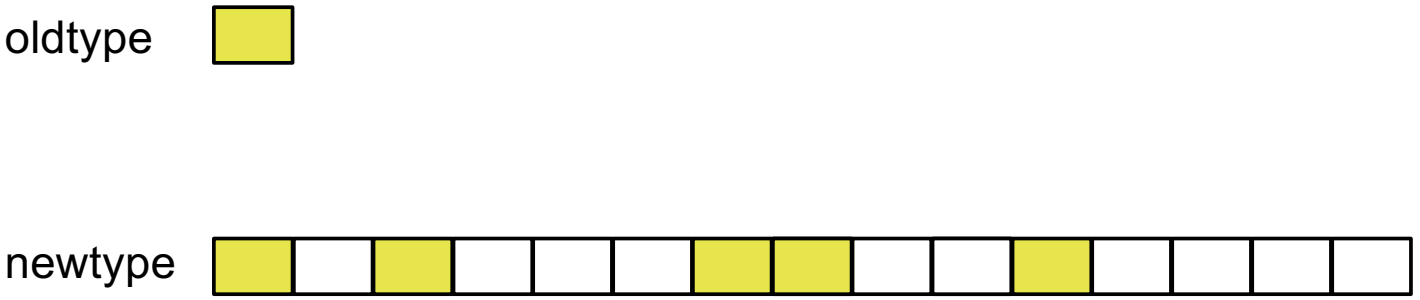
```
MPI_TYPE_VECTOR (COUNT, BLOCKLENGTH,
                 STRIDE, OLDTYPE, NEWTYPE,
                 IERROR)
```

Note:

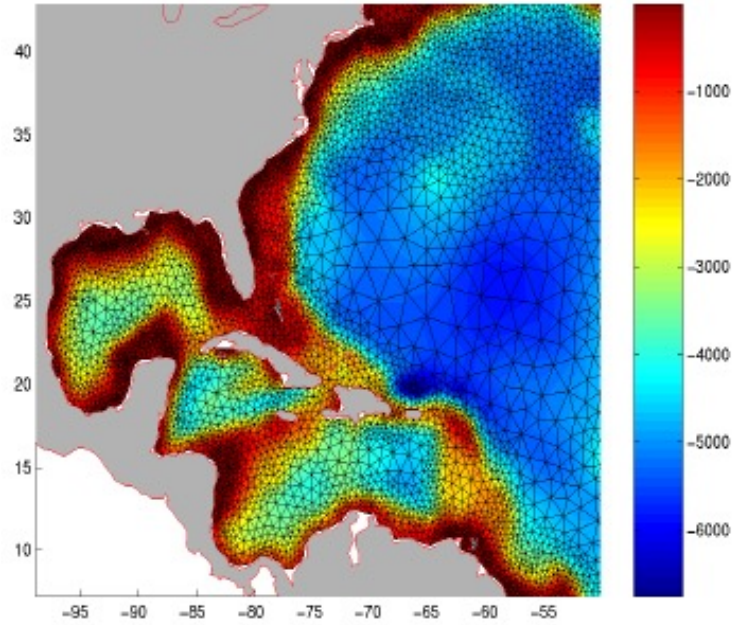
`MPI_Type_contiguous = MPI_Type_vector(count,1,1,...)`

Type map?

Indexed Datatype Example



- Example:
  Boundary in unstructured grids



Committing a datatype

- Once a datatype has been constructed, it needs to be committed before it is used.
- This is done using `MPI_Type_commit`
- C:

```
int MPI_Type_commit (MPI_Datatype *datatype)
```

- Fortran:

```
MPI_TYPE_COMMIT (DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Freeing a datatype

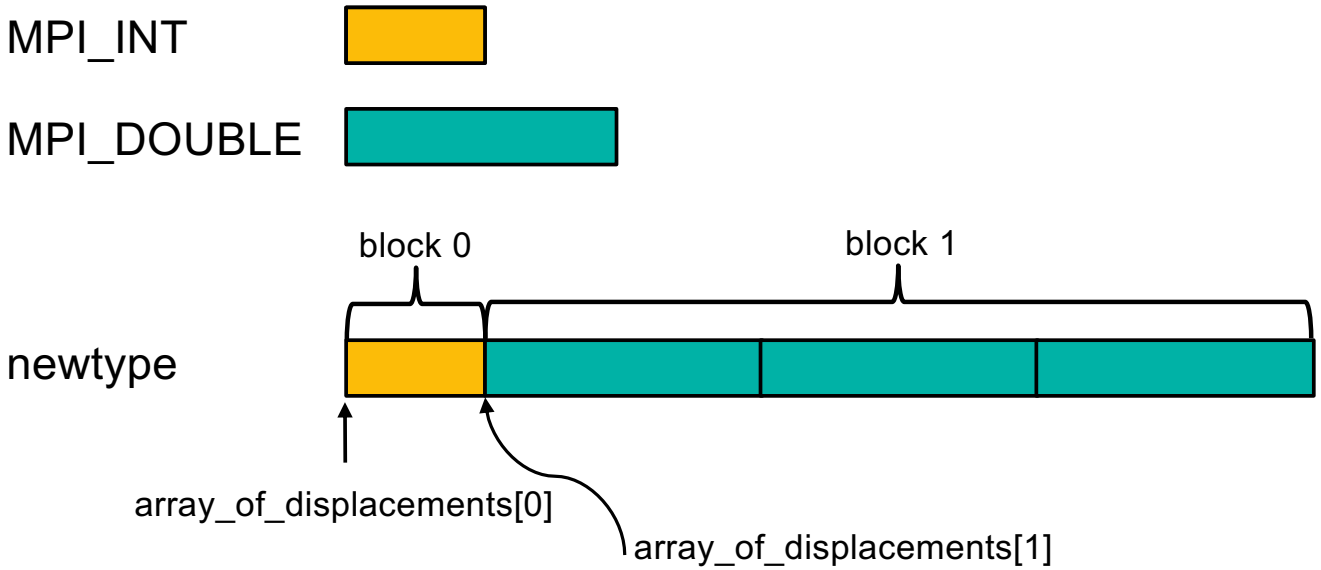
- Deallocate a datatype when no longer used
- C:

```
int MPI_Type_free (MPI_Datatype *datatype)
```

- Fortran:

```
MPI_TYPE_FREE (DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Struct Datatype Example



- `count = 2`
- `array_of_blocklengths[0] = 1`
- `array_of_types[0] = MPI_INT`
- `array_of_blocklengths[1] = 3`
- `array_of_types[1] = MPI_DOUBLE`

Constructing a Struct Datatype

- C:

```
int MPI_Type_create_struct (int count,
        int *array_of_blocklengths,
        MPI_Aint *array_of_displacements,
        MPI_Datatype *array_of_types,
        MPI_Datatype *newtype)
```

- Fortran:

```
MPI_TYPE_CREATE_STRUCT (COUNT,
  ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
  ARRAY_OF_TYPES, NEWTYPE, IERROR)

INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS
```

  - Displacements are in bytes

Extent of a Datatype

- Collect information to compute displacements
- How many bytes is a datatype long (“sizeof”)?
- C:

```
MPI_Type_get_extent (MPI_Datatype datatype,
                     MPI_Aint* lbound, MPI_Aint* extent)
```

- Fortran:

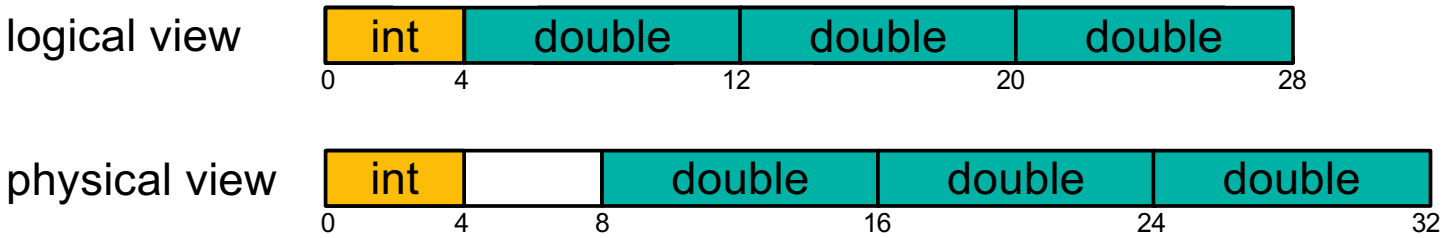
```
MPI_TYPE_GET_EXTENT( DATATYPE, LBOUND,
     EXTENT, IERROR)

  INTEGER DATATYPE, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) LBOUND, EXTENT
```

- `lbound`: lowest displacement of data type in bytes (might not start at zero if a derived data type)

Address Calculation

Why do we need address calculation?

- Storage in memory usually aligned to word boundaries
- On 64-bit systems: one word = 8 Byte



- `double` has to be aligned at word boundary; `int, float` not always (odd number)
  - ➢ Possible gaps in memory between different data items
- `Extent` may not be reliable (e.g. 4 bytes for single `int`, but alignment at 8 bytes if combined with `double`)

➢ Compute displacements based on absolute addresses (`MPI_Get_address`)

Addresses

- Retrieve an absolute address
- Difference of addresses provides displacement
- C:

```
MPI_Get_address( void *location,
                 MPI_Aint *address )
```

Do not use `&location` instead of `MPI_Get_address`!
(`&` is a pointer, not necessarily absolute address)

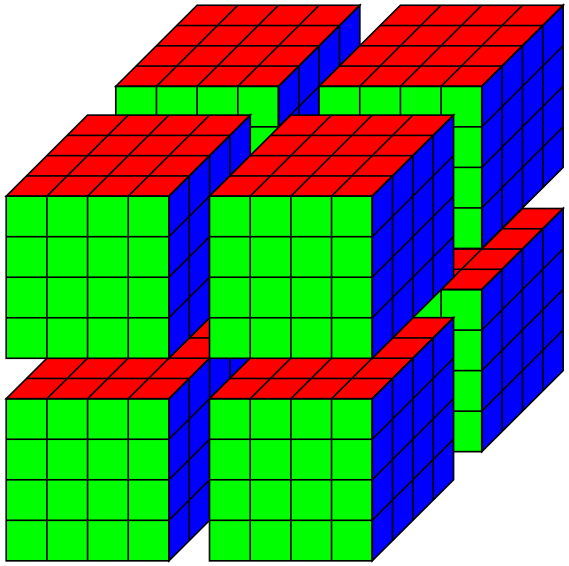
- Fortran:

```
MPI_GET_ADDRESS( LOCATION, ADDRESS, IERROR )

   <type> LOCATION (*)
   INTEGER (KIND=MPI_ADDRESS_KIND) ADDRESS
   INTEGER IERROR
```

Virtual Topologies

Virtual Topologies

- Example: The 3D distribute cubes



- Tell MPI that have have such a distributed cube
- Use MPI to compute process neighbors
- Use MPI to take care of possible periodicity

Virtual Topologies

- Convenient process naming
- Naming scheme to fit the communication pattern.
  - Many applications can benefit from a 2D or 3D topological communication pattern.
- Simplifies writing of code
- Can allow MPI to optimize communications by mapping of runtime processes to available hardware
- Machine independent

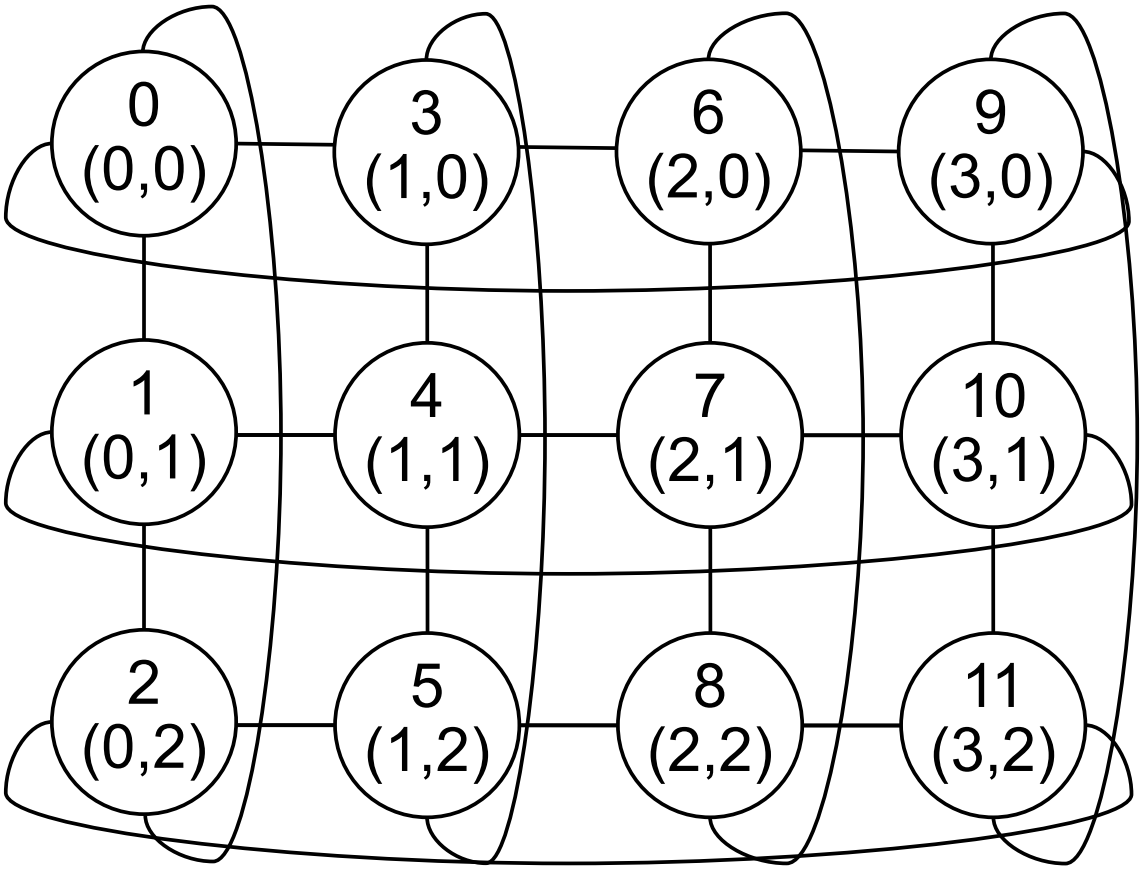
Topology types

- Cartesian topologies
  - each process is “connected” to its neighbors in a virtual grid.
  - boundaries can be cyclic, or not.
  - processes are identified by cartesian coordinates.
- Graph topologies
  - general graphs
  - not covered here

How to use a Virtual Topology

- Creating a topology produces a new communicator
- MPI provides “mapping functions”
- Mapping functions compute processor ranks, based on the topology naming scheme

Example - A 2-dimensional Torus



rank
(coords(1), coords(2))

Ranks in
row-major order

```
NDIMS=2    DIMS(1)=3     PERIODS(1)=.TRUE.
           DIMS(2)=4     PERIODS(2)=.TRUE.
```

Creating a Cartesian Virtual Topology

- C:

```
int MPI_Cart_create (MPI_Comm comm_old,
      int ndims, int *dims, int *periods,
      int reorder, MPI_Comm *comm_cart)
```

- Fortran:

```
MPI_CART_CREATE (COMM_OLD, NDIMS, DIMS,
     PERIODS, REORDER, COMM_CART, IERROR)

  INTEGER COMM_OLD, NDIMS, DIMS(*),
     COMM_CART, IERROR
  LOGICAL PERIODS(*), REORDER
```

Cartesian Mapping Functions (I)

Mapping process grid coordinates to ranks

- C:

```
int MPI_Cart_rank (MPI_Comm comm,
                   int *coords, int *rank)
```

- Fortran:

```
MPI_CART_RANK (COMM, COORDS, RANK, IERROR)

    INTEGER COMM, COORDS(*), RANK, IERROR
```

Cartesian Mapping Functions (II)

Mapping ranks to process grid coordinates

- C:

```
int MPI_Cart_coords (MPI_Comm comm, int rank,
                     int maxdims, int *coords)
```

- Fortran:

```
MPI_CART_COORDS (COMM, RANK, MAXDIMS, COORDS,
                 IERROR)

    INTEGER COMM, RANK, MAXDIMS, COORDS(*),
                 IERROR
```

Cartesian Mapping Functions (III)

Computing ranks of neighbouring processes

- C:

```
int MPI_Cart_shift (MPI_Comm comm,
        int direction, int disp,
        int *rank_source, int *rank_dest)
```

- Fortran:

```
MPI_CART_SHIFT (COMM, DIRECTION, DISP,
        RANK_SOURCE, RANK_DEST, IERROR)

    INTEGER COMM, DIRECTION, DISP,
        RANK_SOURCE, RANK_DEST, IERROR
```

nonexistent neighbour: `MPI_PROC_NULL`

Cartesian Partitioning

- Cut a grid up into 'slices'.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.
- `MPI_Cart_sub` and `MPI_CART_SUB` generate new communicators for the slices.

Cartesian Partitioning with MPI_Cart_sub

- C:

```
int MPI_Cart_sub (MPI_Comm comm,
                  int *remain_dims,
                  MPI_Comm *newcomm)
```

- Fortran:

```
MPI_CART_SUB (COMM, REMAIN_DIMS, NEWCOMM,
                  IERROR)

  INTEGER COMM, NEWCOMM, IERROR
      LOGICAL REMAIN_DIMS(*)
```

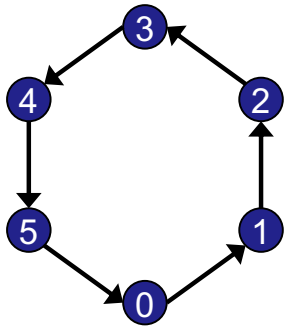
ring from torus:

```
remain_dims(0)=.FALSE.
remain_dims(1)=.TRUE.
```

Exercise 3

Exercise 3: Communication with derived data types

- A set of processes are arranged in a ring.
- Each process stores its rank in `MPI_COMM_WORLD` in an integer.
- Different from Ping-Pong
  - There is no start point – all processes do the same work
  - SIMD (single instruction - multiple data)



Exercise 3, Part 1: Derived Datatypes

Part 1:

- Modify the passing-around-a-ring exercise.
- Calculate two separate sums:
  - rank integer sum, as before
  - rank floating point sum
- Use a *struct* datatype for this.

Part 2:

- Instead of passing around the ring use *global reduction* and a *user-defined function* for the sum of the struct.

Exercise 3: Derived Datatypes (2)

- Defining the *struct* data type
- Fortran

```
module module_my_struct
    type my_struct
        real    :: x
        integer :: i
    end type my_struct
end
```

Later declare a variable

```
use module_my_struct
type(my_struct) :: s
```

Access values:

```
s%x
s%i
```

- C

```
struct myStruct { float x; int i; };
```

Later declare a variable

```
struct myStruct s;
```

Access values:

```
s.x
s.i
```

(or with -> for pointers)

Exercise 3, Part 2: Ring Topology

- Rewrite the exercise passing numbers round the ring using a one-dimensional ring topology.
- Rewrite the exercise in two dimensions, as a torus. Each row of the torus should compute its own separate result.
- **Extra exercise**: Let MPI calculate the number of processes per coordinate direction (`MPI_Dims_create`).

Write the program such that it can handle an arbitrary number of tasks (e.g. a prime number) for the generation of the two-dimensional topology where some processes remain unused (`MPI_COMM_NULL`).