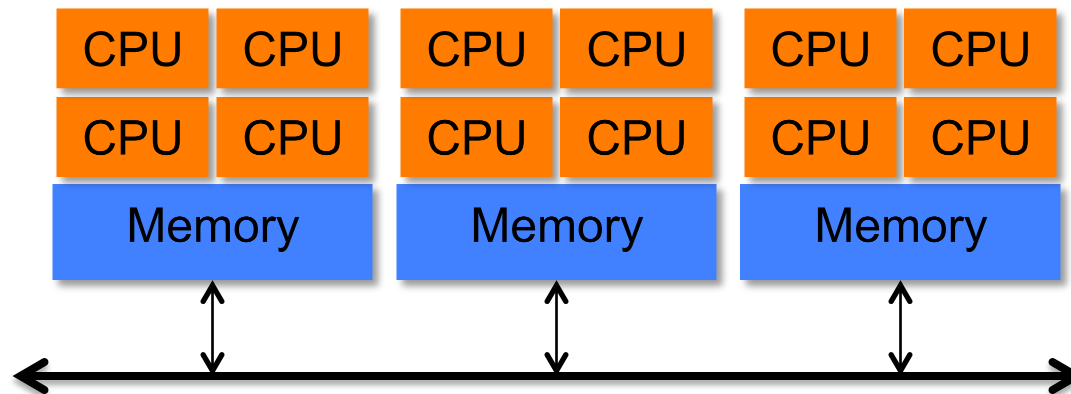
Message **P**assing **I**nterface

George Bosilca
bosilca@icl.utk.edu

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

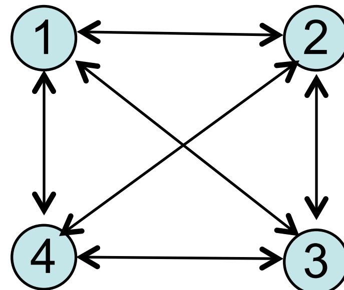
MPI programming model



- Parallelism is explicit: the programmer is responsible for identifying and implementing it using the MPI constructs
- Data is exchanged through explicit messaging
  - Two-sided: point-to-point (send + recv)
  - One-sided: Remote Memory Access (put / get)
- Targets distributed architectures
  - Also supports shared memory

MPI Standard

- http://www.mpi-forum.org
- Current version: 3.1
- MPI = Message Passing Interface



Reasons for using MPI

- Standardization: unique interface blessed by the standardization committee (MPI Forum)
- Portability: Same application works across any existing and future platforms
- Performance: Provide highly tuned constructs
- Functionality: over 450 functions (MPI 3.0)
- Availability: a variety of implementations are available (vendor and open source)

How to use MPI in C

```
#include <mpi.h>
/* other declarations */
                         .
                         .    Serial Code
                         .
Initialize MPI /* parallel code begins */
                         .
      /* compute and communicate */
                         .
Terminate MPI /* parallel code ends */
                         .
                         .    Serial Code
                         .
Application ends
```

- MPI_* namespace is reserved for MPI
- All MPI functions return an error code
- MPI_Init(): initialize the MPI application
- MPI_Finalize(): Completes the MPI application

The 5 must-know MPI function

- MPI_Init / MPI_Finalize
  - Parallel processing setup and tear-down
- MPI_Abort
  - Escape route in unpleasant situations
- MPI_Comm_size and MPI_Comm_rank
  - Process identification

- Your first MPI application

MPI Point-to-point communications

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

Send & Receive

- Explicit communications (FIFO per peer per communicator)
- Move data from one process to another (possibly local) process
  - The data is described by a data-type, a count and a memory location
  - The destination process by a rank in a communicator
  - The matching is tag based

```
int MPI_Send( void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
int MPI_Recv( void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status* status)
```

Blocking Communications

- The process is blocked in the MPI function until:
  - For receives the remote data has been safely copied into the receive buffer
  - For sends the send buffer can be safely modified by the user without impacting the message transfer

```
int MPI_Send( void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
int MPI_Recv( void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status* status)
```

Communication modes

- a send in **standard** mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted.
  - successful completion of the send operation may depend on the occurrence of a matching receive
- **Buffered** mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted.
  - its completion does not depend on the occurrence of a matching receive
- send that uses the **Synchronous** mode can be started whether or not a matching receive was posted. It will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message
  - Its completion does not only indicates that the send buffer can be reused, but it also indicates that the receiver started executing the matching receive

Communication modes

- send that uses the **Ready** communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined.
  - completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused

	Buffered	Synchronous	Ready
Send	MPI_Bsend	MPI_Ssend	MPI_Rsend

Semantics of Point-to-Point Communication

- Order: Messages are non-overtaking
- Progress: No progression guarantees except when in MPI calls
- Fairness: no guarantee of fairness. However, usually a best effort approach implemented in the MPI libraries.
- Resource limitations: Best effort

Quality implementation: a particular implementation of he standard, exhibiting a set of desired properties.

Non-Blocking Communications

- The process returns from the call as soon as possible, before any data transfer has been initiated.
- All flavors of communication modes supported.
- Subsequent MPI call required to check the completion status.

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request )
int MPI_Irecv( void* buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Request *request )
```

Communication Completion

- Single completion
  - completion of a send operation indicates that the sender is now free to update the locations in the send buffer
  - completion of a receive operation indicates that the receive buffer contains the received message

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
int MPI_Test( MPI_Request *request,
                  int *flag, MPI_Status *status)
```

Communication Completion

- Multiple Completions (ANY)
  - A call to MPI_WAITANY or MPI_TESTANY can be used to wait for the completion of one out of several operations.

```
int MPI_Waitany( int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status )
int MPI_Testany( int count, MPI_Request *array_of_requests,
                 int *index, int *flag, MPI_Status *status )
```

Communication Completion

- Multiple Completions (SOME)
  - A call to MPI_WAITSOME or MPI_TESTSOME can be used to wait for the completion of at least one out of several operations.

```
int MPI_Waitsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
int MPI_Testsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
```

Communication Completion

- Multiple Completions (ALL)
  - A call to MPI_WAITALL or MPI_TESTALL can be used to wait for the completion of all operations.

```
int MPI_Waitall( int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses )
int MPI_Testall( int count, MPI_Request *array_of_requests,
                 int *flag, MPI_Status *array_of_statuses )
```

Persistent Communications

- A communication with the same argument list repeatedly executed within the inner loop of a parallel computation
  - Allow MPI implementations to optimize the data transfers
- All communication modes (buffered, synchronous and ready) can be applied

```
int MPI_Send_init( void* buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm,
                   MPI_Request *request )
int MPI_Recv_init( void* buf, int count, MPI_Datatype datatype,
                   int source, int tag, MPI_Comm comm,
                   MPI_Request *request )
```

MPI Derived Datatypes

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

MPI Datatypes

- Abstract representation of underlying data
  - Handle type: MPI_Datatype
- Pre-defined handles for intrinsic types
  - E.g., C: MPI_INT, MPI_FLOAT, MPI_DOUBLE
  - E.g., Fortran: MPI_INTEGER, MPI_REAL
  - E.g., C++: MPI::BOOL
- User-defined datatypes
  - E.g., arbitrary / user-defined C structs

MPI Data Representation

- Multi platform interoperability
- Multi languages interoperability
  - Is MPI_INT the same as MPI_INTEGER?
  - How about MPI_INTEGER[1,2,4,8]?
- Handling datatypes in Fortran with MPI_SIZEOF and MPI_TYPE_MATCH_SIZE

Multi-Platform Interoperability

- Different data representations
  - Length 32 vs. 64 bits
  - Endianness conflict
- Problems
  - No standard about the data length in the programming languages (C/C++)
  - No standard floating point data representation
    - IEEE Standard 754 Floating Point Numbers
      - Subnormals, infinities, NANs …
    - Same representation but different lengths

How About Performance?

- Old way
  - Manually copy the data in a user pre-allocated buffer, or
  - Manually use MPI_PACK and MPI_UNPACK
- New way
  - Trust the [modern] MPI library
  - High performance MPI datatypes

MPI Datatypes

- MPI uses “datatypes” to:
  - Efficiently represent and transfer data
  - Minimize memory usage
- Even between heterogeneous systems
  - Used in most communication functions (MPI_SEND, MPI_RECV, etc.)
  - And file operations
- MPI contains a large number of pre-defined datatypes

Some of MPI's Pre-Defined Datatypes

MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER*2
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL
MPI_DOUBLE	double	DOUBLE PRECISION
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

Datatype Matching

- Two requirements for correctness:
  - Type of each data in the send / recv buffer matches the corresponding type specified in the sending / receiving operation
  - Type specified by the sending operation has to match the type specified for receiving operation
- Issues:
  - Matching of type of the host language
  - Match of types at sender and receiver

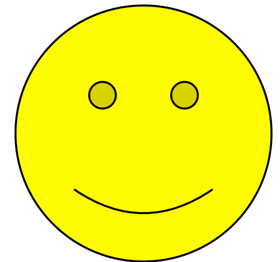
Datatype Conversion

- "Data sent = data received"
- 2 types of conversions:
  - Representation conversion: change the binary representation (e.g., hex floating point to IEEE floating point)
  - Type conversion: convert from different types (e.g., int to float)

→ Only representation conversion is allowed

Datatype Conversion

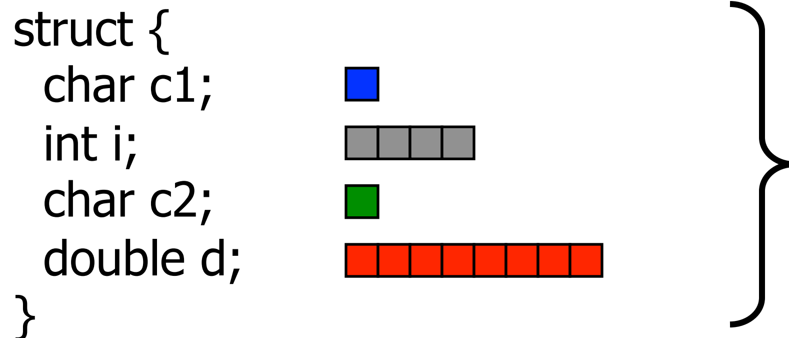
```
if( my_rank == root )
    MPI_Send( ai, 1, MPI_INT, … )
else
    MPI_Recv( ai, 1, MPI_INT, … )
```

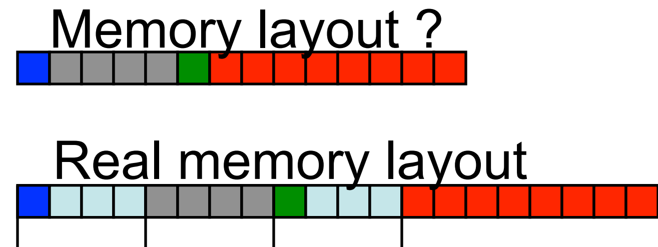


```
if( my_rank == root )
    MPI_Send( ai, 1, MPI_INT, … )
else
    MPI_Recv( af, 1, MPI_FLOAT, … )
```

Memory Layout

- How to describe a memory layout ?





Using iovecs (list of addresses)

<pointer to memory, length>

<baseaddr c1, 1>,<addr_of_i, 4>,
<addr_of_c2, 1>, <addr_of_d, 8>

- Waste of space
- Not portable …

Using displacements from base addr

<displacement, length>

<0, 1>, <4, 4>, <8, 1>, <12, 8>

- Sometimes more space efficient
- And nearly portable
- What are we missing ?

Datatype Specifications

- Type signature
  - Used for message matching

    $\{ type_0, type_1, \ldots, type_n \}$
- Type map
  - Used for local operations

    $\{ (type_0, disp_0), (type_1, disp_1), \ldots, (type_n, disp_n) \}$

➔ It's all about the memory layout

User-Defined Datatypes

- Applications can define unique datatypes
  - Composition of other datatypes
  - MPI functions provided for common patterns
    - Contiguous
    - Vector
    - Indexed
    - …

➔ Always reduces to a type map of pre-defined datatypes

Handling datatypes

- MPI impose that all datatypes used in communications or file operations should be committed.
  - Allow MPI libraries to optimize the data representation

MPI_Type_commit(MPI_Datatype*)

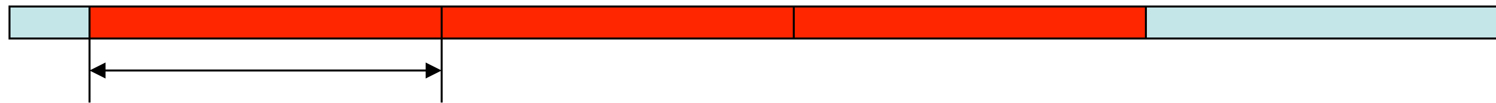
MPI_Type_free(MPI_Datatype*)

- All datatypes used during intermediary steps, and never used to communicate does not need to be committed.

Contiguous Blocks

- Replication of the datatype into contiguous locations.

MPI_Type_contiguous(3, oldtype, newtype)

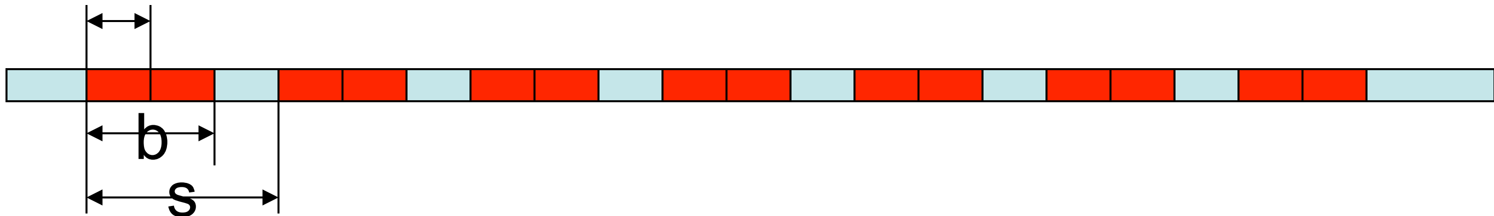


MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)		
IN	count	replication count(positive integer)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Vectors

- Replication of a datatype into locations that consist of equally spaced blocks

MPI_Type_vector(7, 2, 3, oldtype, newtype)



MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)		
IN	count	number of blocks (positive integer)
IN	blocklength	number of elements in each block (positive integer)
IN	stride	number of elements between start of each block (integer)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Indexed Blocks

- Replication of an old datatype into a sequence of blocks, where each block can contain a different number of copies and have a different displacement

MPI_TYPE_INDEXED(count, array_of_blocks, array_of_displs, oldtype, newtype)

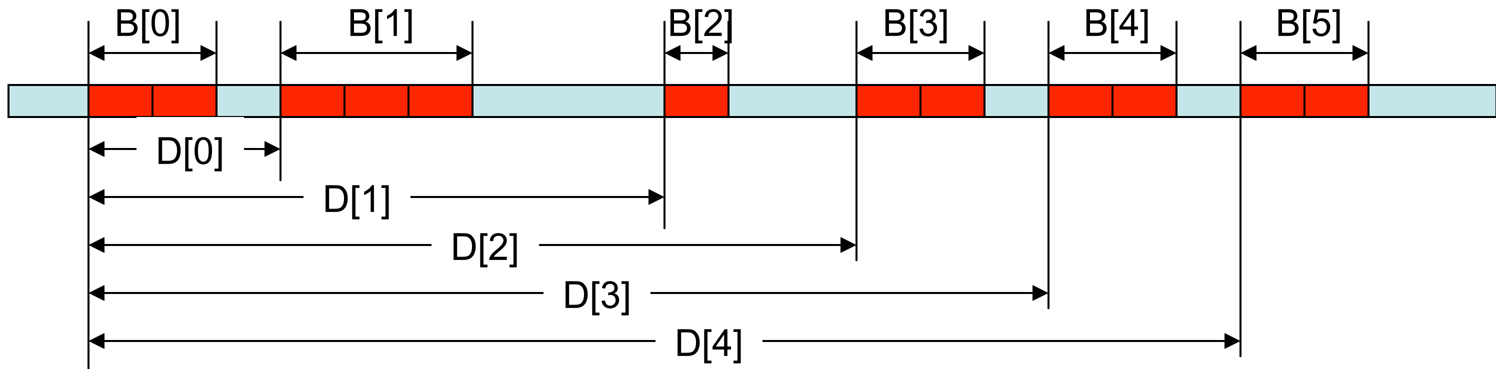
IN	count	number of blocks (positive integer)
IN	a_of_b	number of elements per block (array of positive integer)
IN	a_of_d	displacement of each block from the beginning in multiple multiple of oldtype (array of integers)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Indexed Blocks

array_of_blocklengths[] = { 2, 3, 1, 2, 2, 2 }

array_of_displs[] = { 0, 3, 10, 13, 16, 19 }

```
MPI_Type_indexed( 6, array_of_blocklengths,
                  array_of_displs, oldtype, newtype )
```



Datatype Composition

- Each of the previous functions are the super set of the previous

  CONTIGUOUS < VECTOR < INDEXED

- Extend the description of the datatype by allowing more complex memory layout
  - Not all data structures fit in common patterns
  - Not all data structures can be described as compositions of others

“H” Functions

- Displacement is not in multiple of another datatype
- Instead, displacement is in bytes
  - MPI_TYPE_HVECTOR
  - MPI_TYPE_HINDEX
- Otherwise, similar to their non-”H” counterparts

Arbitrary Structures

- The most general datatype constructor
- Allows each block to consist of replication of different datatypes

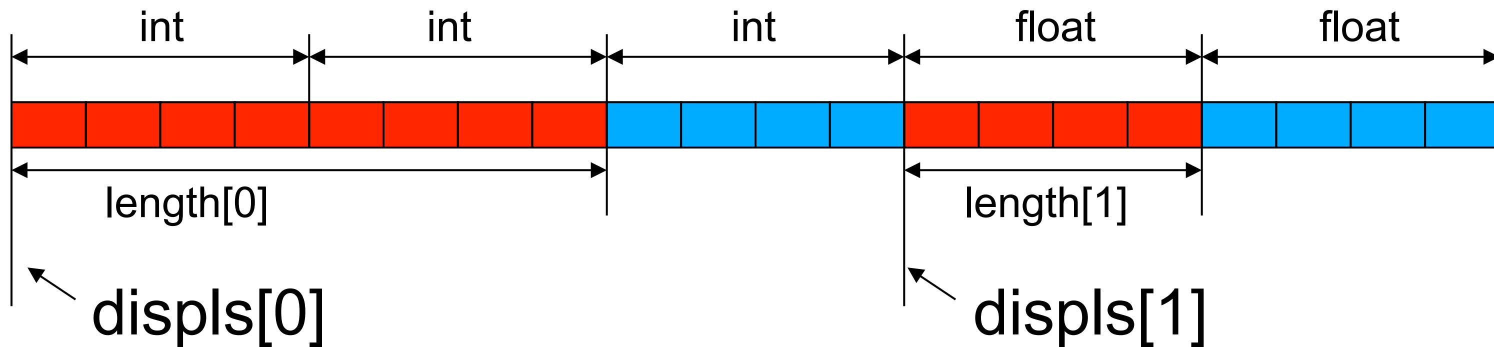
MPI_TYPE_CREATE_STRUCT(count, array_of_blocklength,
array_of_displs, array_of_types, newtype)

IN	count	number of entries in each array (positive integer)
IN	a_of_b	number of elements in each block (array of integers)
IN	a_of_d	byte displacement in each block (array of Aint)
IN	a_of_t	type of elements in each block (array of MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Arbitrary Structures

```
struct {
  int i[3];
  float f[2];
} array[100];
```

```
Array_of_lengths[] = { 2, 1 };
Array_of_displs[] = { 0, 3*sizeof(int) };
Array_of_types[] = { MPI_INT, MPI_FLOAT };
MPI_Type_struct( 2, array_of_lengths,
  array_of_displs, array_of_types, newtype );
```



Portable Vs. non portable

- The portability refer to the architecture boundaries
- Non portable datatype constructors:
  - All constructors using byte displacements
  - All constructors with H<type>, MPI_Type_struct
- Limitations for non portable datatypes
  - One sided operations
  - Parallel I/O operations

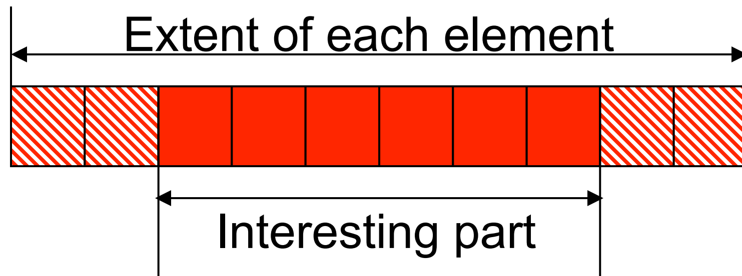
MPI_GET_ADDRESS

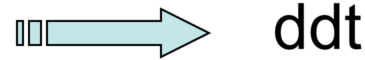
- Allow all languages to compute displacements
  - Necessary in Fortran
  - *Usually* unnecessary in C (e.g., "&foo")

MPI_GET_ADDRESS(location, address)		
IN	location	location in the caller memory (choice)
OUT	address	address of location (address integer)

And Now the Dark Side…

- Sometimes more complex memory layout have to be expressed



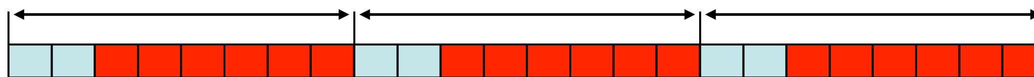


ddt

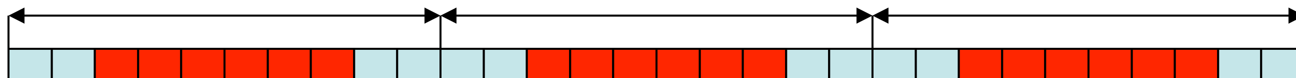
```
Struct {
  int gap[2];
  int i[6];
  int gap2[2];
}
```

MPI_Send(buf, 3, ddt, …)

What we just did …



And what we expected to do…



Lower-Bound and Upper-Bound Markers

- Define datatypes with “holes” at the beginning or end
- 2 pseudo-types: MPI_LB and MPI_UB
  - Used with MPI_TYPE_STRUCT

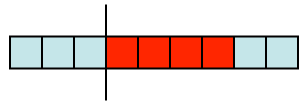
$$\text{Typemap} = \{ (\text{type}_0, \text{disp}_0), \ldots, (\text{type}_n, \text{disp}_n) \}$$

$$\text{lb(Typemap)} \begin{cases} \text{Min}_j \ \text{disp}_j & \text{if no entry has type lb} \\ \text{min}_j(\{\text{disp}_j \text{ such that type}_j = \text{lb}) & \text{otherwise} \end{cases}$$

$$\text{ub(Typemap)} \begin{cases} \text{Max}_j \ \text{disp}_j + \text{sizeof}(\text{type}_j) + \text{align} & \text{if no entry has type ub} \\ \text{Max}_j\{\text{disp}_j \text{ such that type}_j = \text{ub}\} & \text{otherwise} \end{cases}$$

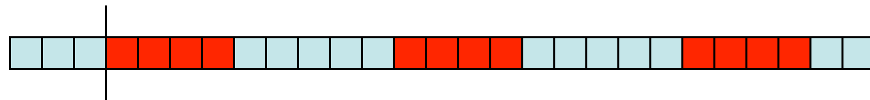
MPI_LB and MPI_UB

```
displs = ( -3, 0, 6 )
blocklengths = ( 1, 1, 1 )
types = ( MPI_LB, MPI_INT, MPI_UB )
MPI_Type_struct( 3, displs, blocklengths,
  types, type1 )
```



Typemap= { (lb, -3), (int, 0), (ub, 6) }

```
MPI_Type_contiguous( 3, type1, type2 )
```



Typemap= { (lb, -3), (int, 0), (int, 9), (int, 18), (ub, 24) }

MPI 2 Solution

- Problem with the way MPI-1 treats this problem: upper and lower bound can become messy, if you have derived datatype consisting of derived dataype consisting of derived datatype consisting of… and each of them has MPI_UB and MPI_LB set
- There is no way to erase LB and UB markers once they are set !!!
- MPI-2 solution: reset the extent of the datatype

```
MPI_Type_create_resized ( MPI_Datatype datatype,
MPI_Aint lb, MPI_Aint extent, MPI_Datatype*newtype );
```

- Erases all previous lb und ub markers

True Lower-Bound and True Upper-Bound Markers

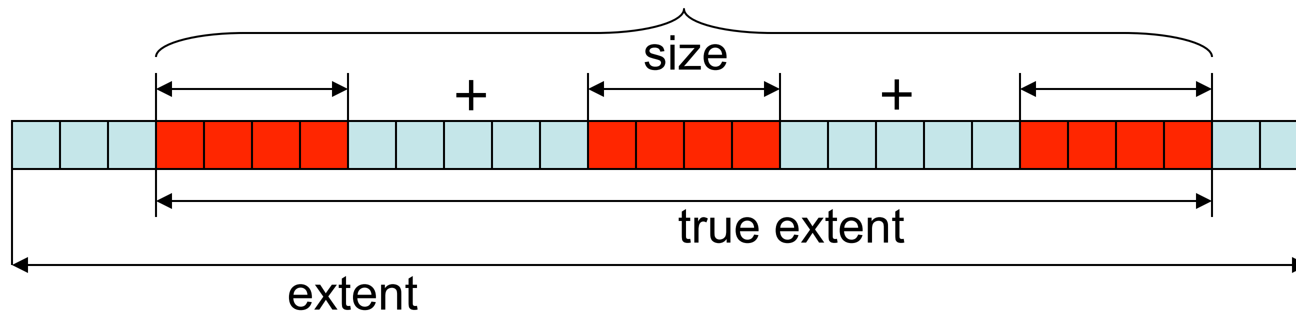
- Define the real extent of the datatype: the amount of memory needed to copy the datatype inside
- TRUE_LB define the lower-bound ignoring all the MPI_LB markers.

$$\text{Typemap} = \{ (\text{type}_0, \text{disp}_0), \ldots, (\text{type}_n, \text{disp}_n) \}$$

$$\text{true_lb(Typemap)} = \min_j \{ \text{disp}_j : \text{type}_j \mathrel{!=} \text{lb} \}$$

$$\text{true_ub(Typemap)} = \max_j \{ \text{disp}_j + \text{sizeof}(\text{type}_j) : \text{type}_j \mathrel{!=} \text{ub} \}$$

Information About Datatypes



MPI_TYPE_GET_{TRUE_}EXTENT(datatype, {true_}lb, {true_}extent)

IN	datatype	the datatype	(MPI_Datatype handle)
OUT	{true_}lb	{true} lower-bound of datatype	(MPI_AINT)
OUT	{true_}extent	{true} extent of datatype	(MPI_AINT)

MPI_TYPE_SIZE(datatype, size)

IN	datatype	the datatype (MPI_Datatype handle)
OUT	size	datatype size (integer)

Decoding a datatype

- Sometimes is important to know how a datatype was created (eg. Libraries developers)
- Given a datatype can I determine how it was created ?
- Given a datatype can I determine what memory layout it describe ?

MPI_Type_get_enveloppe

```
MPI_Type_get_envelope ( MPI_Datatype datatype,
                        int *num_integers, int *num_addresses,
                        int *num_datatypes, int *combiner );
```

- The combiner field returns how the datatype was created, e.g.
  - MPI_COMBINER_NAMED: basic datatype
  - MPI_COMBINER_CONTIGUOS: MPI_Type_contiguous
  - MPI_COMBINER_VECTOR: MPI_Type_vector
  - MPI_COMBINER_INDEXED: MPI_Type_indexed
  - MPI_COMBINER_STRUCT: MPI_Type_struct
- The other fields indicate how large the integer-array, the datatype-array, and the address-array has to be for the following call to MPI_Type_get_contents

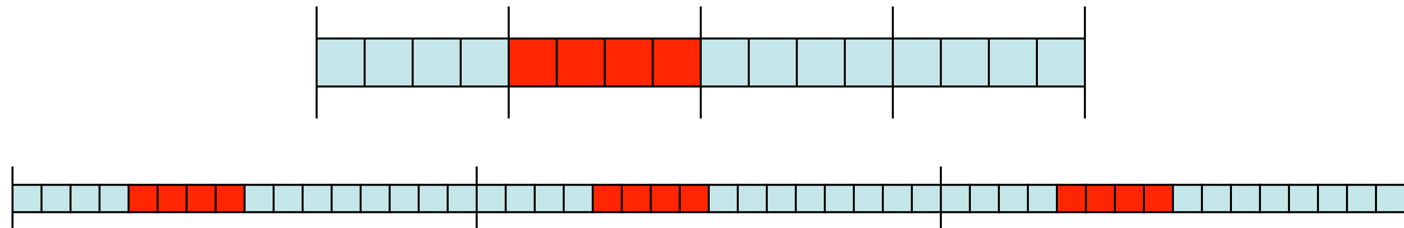
MPI_Type_get_contents

```
MPI_Type_get_contents ( MPI_Datatype datatype,
    int max_integer, int max_addresses, int max_datatypes,
    int *integers, int *addresses, MPI_Datatype *dts);
```

- Call is erroneous for a predefined datatypes
- If returned data types are derived datatypes, then objects are duplicates of the original derived datatypes. User has to free them using MPI_Type_free
- The values in the integers, addresses and datatype arrays are depending on the original datatype constructor

One Data By Cache Line

- Imagine the following architecture:
  - Integer size is 4 bytes
  - Cache line is 16 bytes
- We want to create a datatype containing the second integer from each cache line, repeated three times



- How many ways are there?

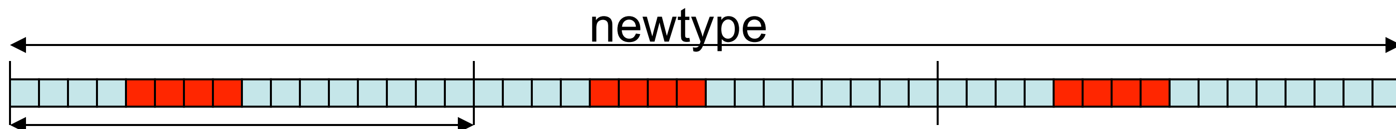
Solution 1

```
MPI_Datatype array_of_types[] = { MPI_INT, MPI_INT, MPI_INT, MPI_UB };
MPI_Aint start, array_of_displs[] = { 0, 0, 0, 0 };
int array_of_lengths[] = { 1, 1, 1, 1 };
struct one_by_cacheline c[4];

MPI_Get_address( &c[0], &(start) );
MPI_Get_address( &c[0].int[1], &(array_of_displs[0]) );
MPI_Get_address( &c[1].int[1], &(array_of_displs[1]) );
MPI_Get_address( &c[2].int[1], &(array_of_displs[2]) );
MPI_Get_address( &c[3], &(array_of_displs[3]) );

for( i = 0; i < 4; i++ ) Array_of_displs[i] -= start;

MPI_Type_create_struct( 4, array_of_lengths,
                        array_of_displs, array_of_types, newtype )
```

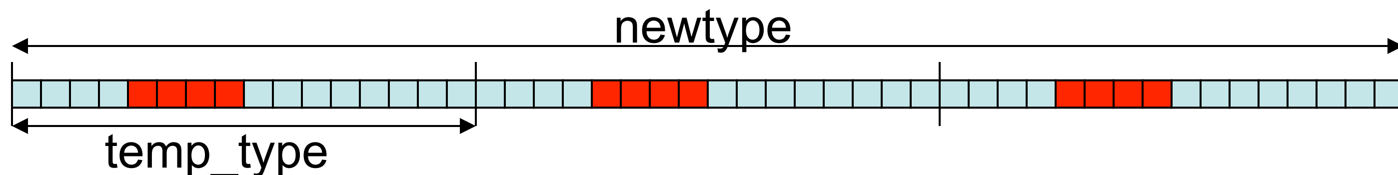


Solution 2

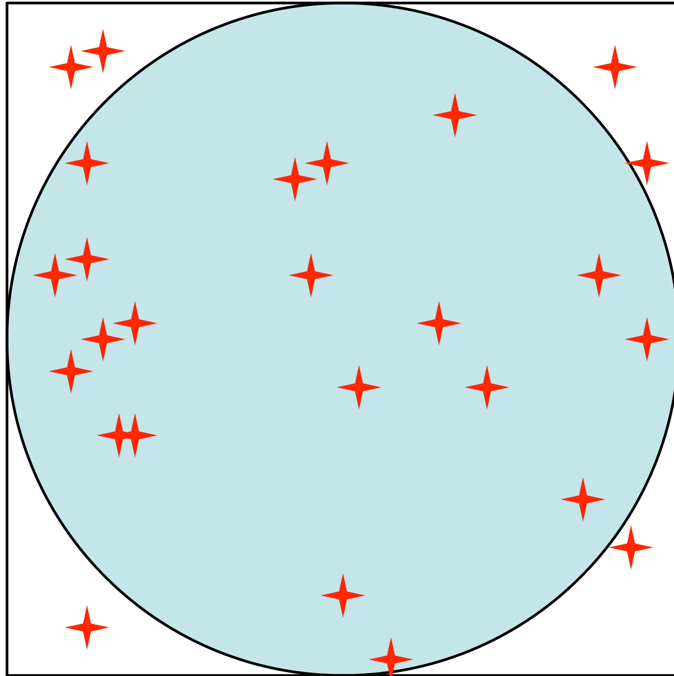
```
MPI_Datatype array_of_types[] = { MPI_INT, MPI_UB };
MPI_Aint start, array_of_displs[] = { 4, 16 };
int array_of_lengths[] = { 1, 1 };
struct one_by_cacheline c[2];

MPI_Get_address( &c[0], &(start) );
MPI_Get_address( &c[0].int[1], &(array_of_displs[0]) );
MPI_Get_address( &c[1], &(array_of_displs[1]) );

Array_of_displs[0] -= start;
Array_of_displs[1] -= start;
MPI_Type_create_struct( 2, array_of_lengths,
                 array_of_displs, array_of_types, temp_type )
MPI_Type_contiguous( 3, temp_type, newtype )
```



Compute π



$$A_{square} = (2r)^2 = 4r^2 \Rightarrow r^2 = \frac{A_{square}}{4}$$

$$A_{circle} = \pi r^2 \Rightarrow r^2 = \frac{A_{circle}}{\pi}$$

$$\pi = 4 \times \frac{A_{circle}}{A_{square}}$$

- Random sampling
  - The more point the better the approximation

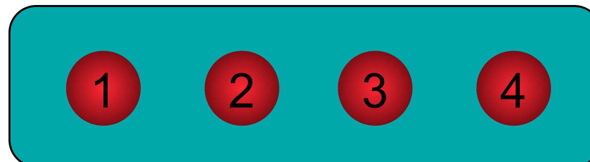
```
for i in [0 .. npoints]
  x = random( [0 .. 1])
  y = random( [0 .. 1])
  if point(x, y) in circle
    circle_points++

pi = 4 * circle_points / npoints
```

Intra and Inter Communicators

Groups

- A group is a set of processes
  - The group have a size
  - And each process have a rank
- Creating a group is a local operation
- Why we need groups
  - To make a clear distinction between processes
  - To allow communications in-between subsets of processes
  - To create intra and inter communicators …



Groups

- MPI_GROUP_*(group1, group2, newgroup)
  - Where * $\in$ {UNION, INTERSECTION, DIFFERENCE}
  - Newgroup contain the processes satisfying the * operation ordered first depending on the order in group1 and then depending on the order in group2.
  - In the newgroup each process could be present only one time.
- There is a special group without any processes MPI_GROUP_EMPTY.

Groups

- group1 = {a, b, c, d, e}
- group2 = {e, f, g, b, a}
- Union
  - newgroup = {a, b, c, d, e, f, g}
- Difference
  - newgroup = {c, d}
- Intersection
  - newgroup = {a, b, e}

Groups

- MPI_GROUP_*(group, n, ranks, newgroup)
  - Where * $\in$ {INCL, EXCL}
  - N is the number of valid indexes in the ranks array.

- For INCL the order in the result group depend on the ranks order
- For EXCL the order in the result group depend on the original order

Groups

- Group = {a, b, c, d, e, f, g, h, i, j}
- N = 4, ranks = {3, 4, 1, 5}
- INCL
  - Newgroup = {c, d, a, e}
- EXCL
  - Newgroup = {b, c, f, g, h, i, j}

Groups

- MPI_GROUP_RANGE_*(group, n, ranges, newgroup)
  - Where * $\in$ {INCL, EXCL}
  - N is the number of valid entries in the ranges array
  - Ranges is a tuple (start, end, stride)
- For INCL the order in the new group depend on the order in ranges
- For EXCL the order in the new group depend on the original order

Groups

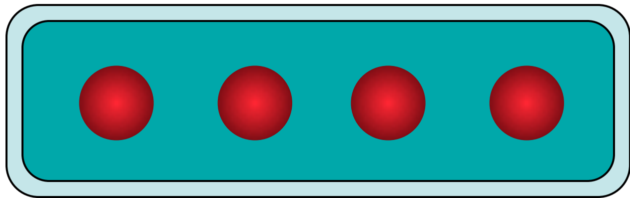
- Group = {a, b, c, d, e, f, g, h, i, j}
- N=3; ranges = ((6, 7, 1), (1, 6, 2), (0, 9, 4))
- Then the range
  - (6, 7, 1) => {g, h} (ranks (6, 7))
  - (1, 6, 2) => {b, d, f} (ranks (1, 3, 5))
  - (0, 9, 4) => {a, e, i} (ranks (0, 4, 8))
- INCL
  - Newgroup = {g, h, b, d, f, a, e, i}
- EXCL
  - Newgroup = {c, j}

Communicators

- A special channel between some processes used to exchange messages.
- Operations creating the communicators are collectives, but accessing the communicator information is a local operation.
- Special communicators: MPI_COMM_WORLD, MPI_COMM_NULL, MPI_COMM_SELF
- MPI_COMM_DUP(comm, newcomm) create an identical copy of the comm in newcomm.
  - Allow exchanging messages between the same set of nodes using identical tags (useful for developing libraries).

Intracommunicators

- What exactly is a intracommunicator ?

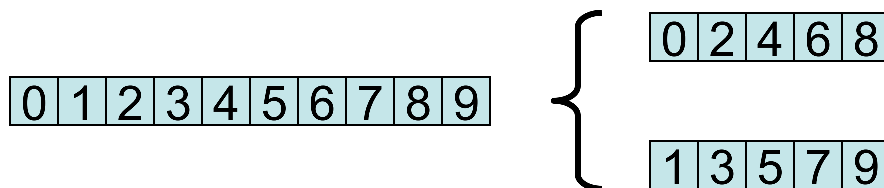


- some processes
- **ONE** group
- one communicator

- MPI_COMM_SIZE, MPI_COMM_RANK
- MPI_COMM_COMPARE(conn1, comm2, result)
  - MPI_IDENT: comm1 and comm2 represent the same communicator
  - MPI_CONGRUENT: same processes, same ranks
  - MPI_SIMILAR: same processes, different ranks
  - MPI_UNEQUAL: otherwise

Intracommunicators

- MPI_COMM_CREATE(comm, group, newcomm)
  - Create a new communicator on all processes from the communicator comm who are defined on the group.
  - All others processes get MPI_COMM_NULL



MPI_Group_range_excl(group, 1, (0, 9, 2), odd_group);
MPI_Group_range_excl(group, 1, (1, 9, 2), even_group);
MPI_Comm_create(comm, odd_comm, odd_comm);
MPI_Comm_create(comm, even_group, even_comm);

Intracommunicators

- MPI_COMM_SPLIT(comm, color, key, newcomm)
  - Color : control of subset assignment
  - Key : control of rank assignement

rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
color	0	⊥	3	0	3	0	0	5	3	⊥
key	3	1	2	5	1	1	1	2	1	0

3 different colors => 3 communicators

1. {A, D, F, G} with ranks {3, 5, 1, 1} => {F, G, A, D}
2. {C, E, I} with ranks {2, 1, 3} => {E, I, C}
3. {H} with ranks {1} => {H}

B and J get MPI_COMM_NULL as they provide an undefined color (MPI_UNDEFINED)

Intracommunicators

0	1	2	3	4	5	6	7	8	9

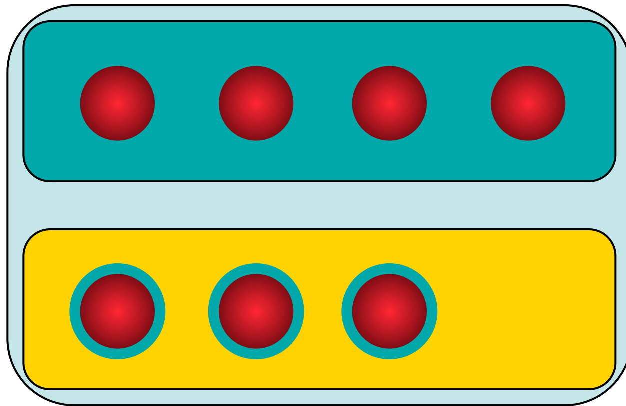
0	2	4	6	8

1	3	5	7	9

Rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
Color	0	1	0	1	0	1	0	1	0	1
Key	1	1	1	1	1	1	1	1	1	1

Intercommunicators

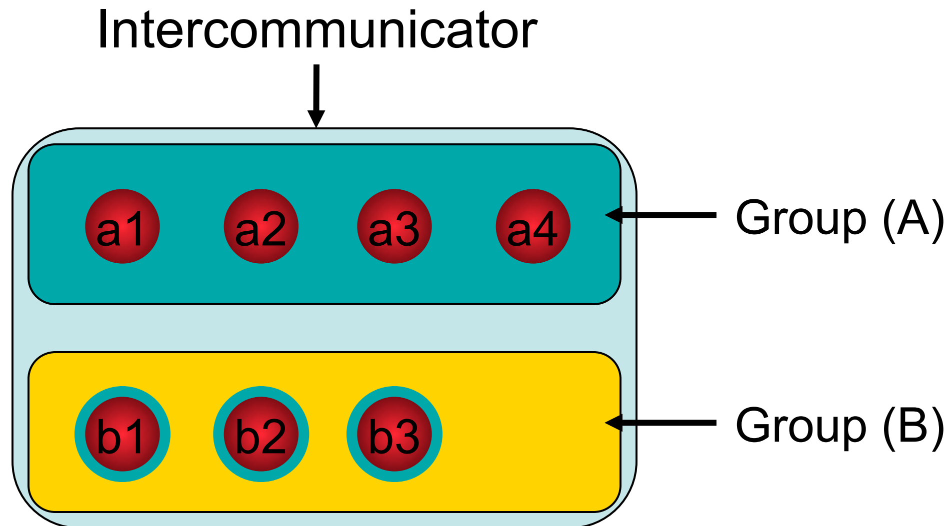
- And what’s a intercommunicator ?



- some more processes
- **TWO** groups
- one communicator

- MPI_COMM_REMOTE_SIZE(comm, size)
  MPI_COMM_REMOTE_GROUP(comm, group)
- MPI_COMM_TEST_INTER(comm, flag)
- MPI_COMM_SIZE, MPI_COMM_RANK return the local size respectively rank

Anatomy of a Intercommunicator

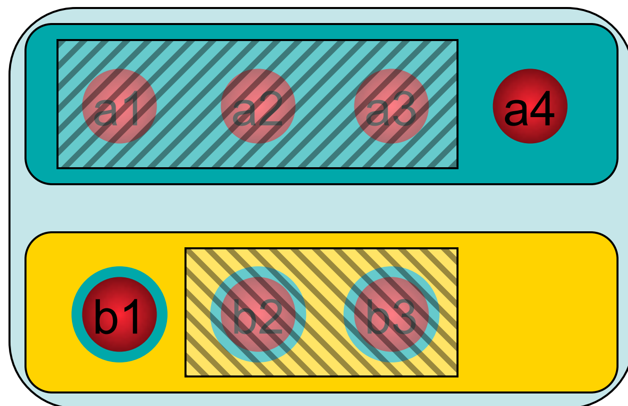


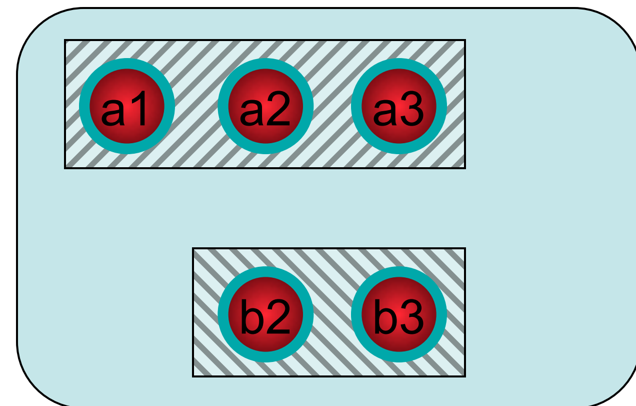
It’s not possible to send a message to a process in the same group using this communicator

For any processes from group (A)	For any processes from group (B)
• (A) is the local group • (B) is the remote group	• (A) is the remote group • (B) is the local group

Intercommunicators

- MPI_COMM_CREATE(comm, group, newcomm)
  - All processes on the left group should execute the call with the same subgroup of processes, when all processes from the right side should execute the call with the same subgroup of processes. Each of the subgroup is related to a different side.





Intercommunicators

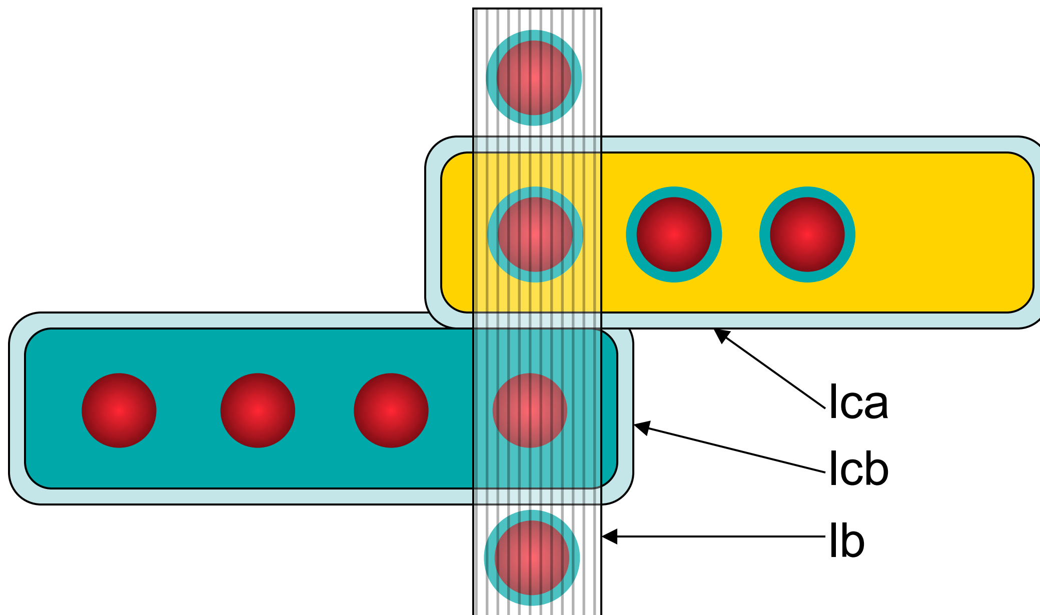
- MPI_INTERCOMM_CREATE(local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)

  Local_comm : local intracommunicator

  Local_leader : rank of root in the local_comm

  Bridge_comm : "bridge" communicator …

  Remote_leader : rank of remote leader in bridge_comm



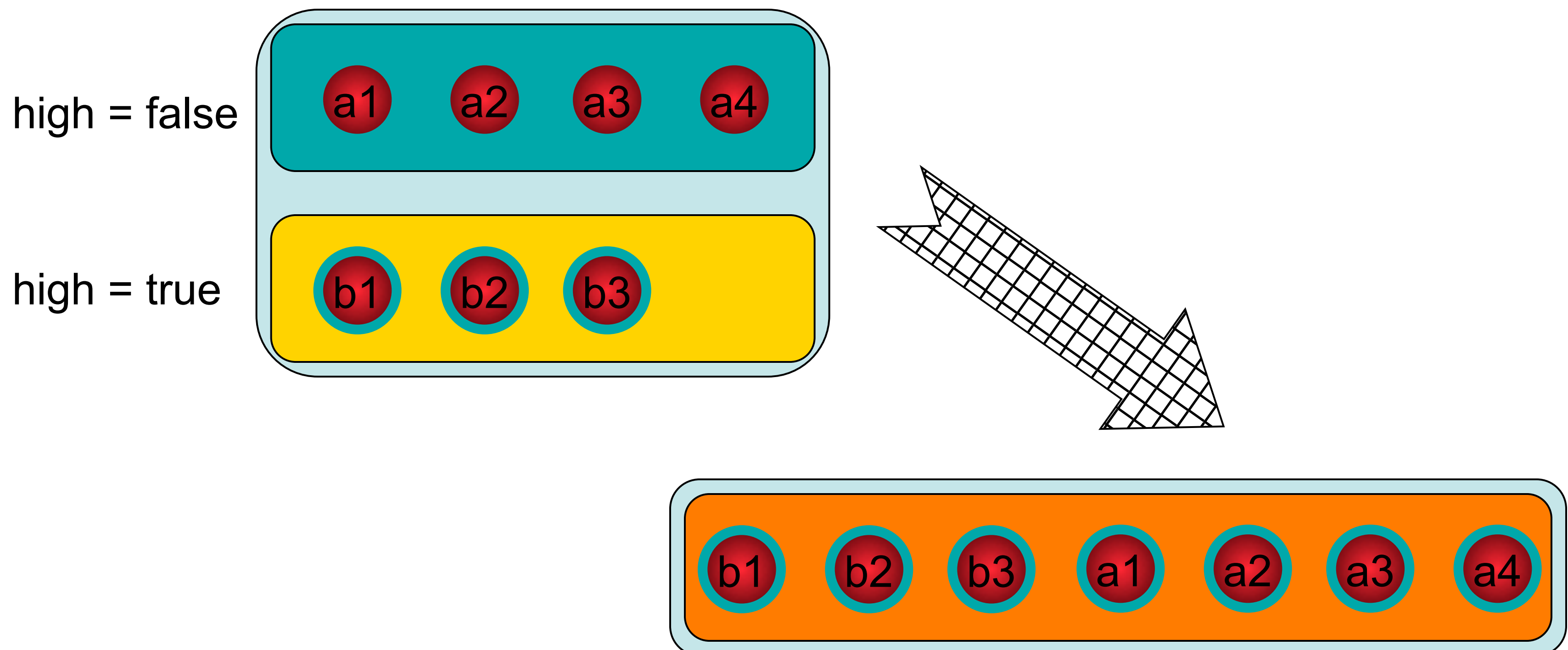
MPI_INTERCOMM_CREATE

lca, 0, lb, 2, tag, new

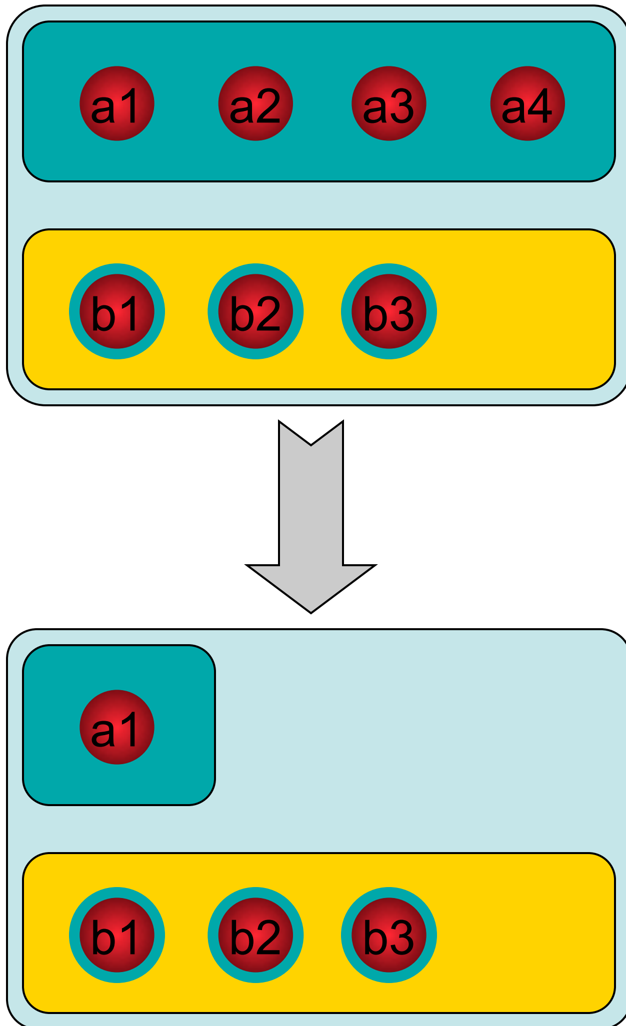
lcb, 4, lb, 1, tag, new

Intercommunicators

- MPI_INTERCOMM_MERGE(intercomm, high, intracomm)
  - Create an intracomm from the union of the two groups
  - The order of processes in the union respect the original one
  - The high argument is used to decide which group will be first (rank 0)



Example

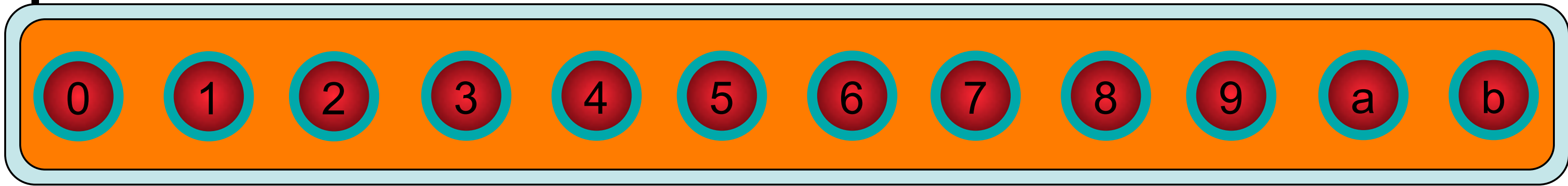


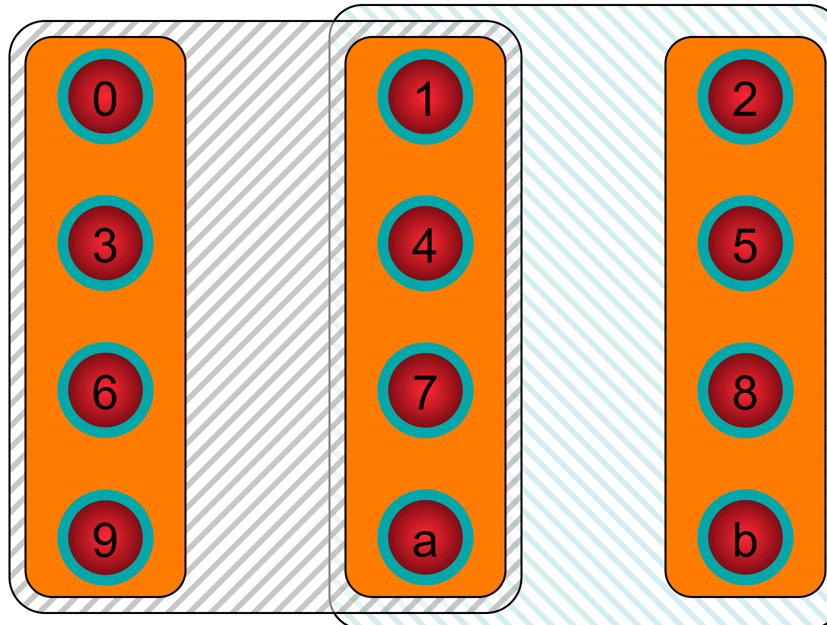

```
MPI_Comm inter_comm, new_inter_comm;
MPI_Group local_group, group;
int rank = 0;

if( /* left side (ie. a*) */ ) {
  MPI_Comm_group( inter_comm, &local_group);
  MPI_Group_incl( local_group, 1, &rank, &group);
  MPI_Group_free( &local_group );
} else
  MPI_Comm_group( inter_comm, &group );

MPI_Comm_create( inter_comm, group,
                              &new_inter_comm );
MPI_Group_free( &group );
```

Exercice



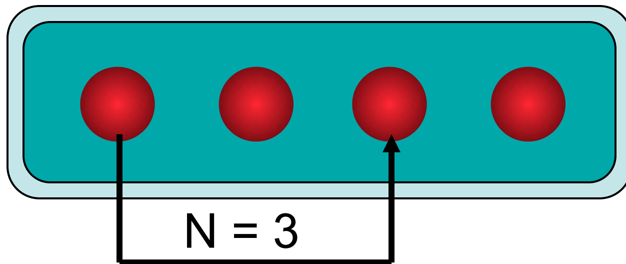


Intercommunicators – P2P

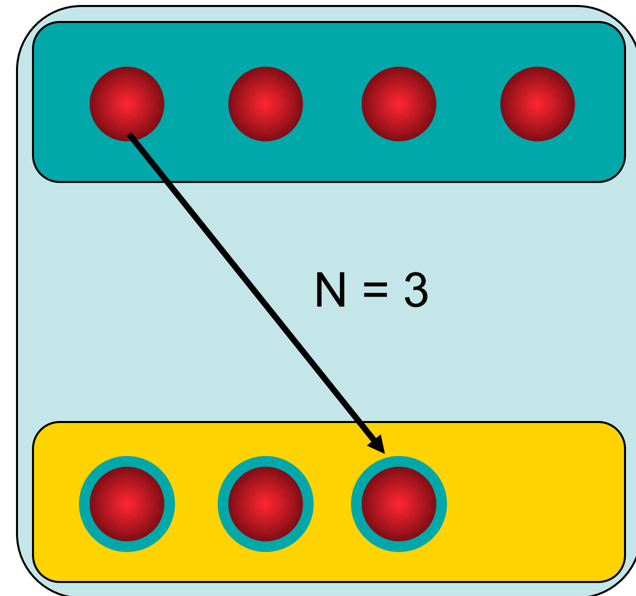
On process 0:

MPI_Send(buf, MPI_INT, 1, n, tag, intercomm)

- Intracommunicator



- Intercommunicator

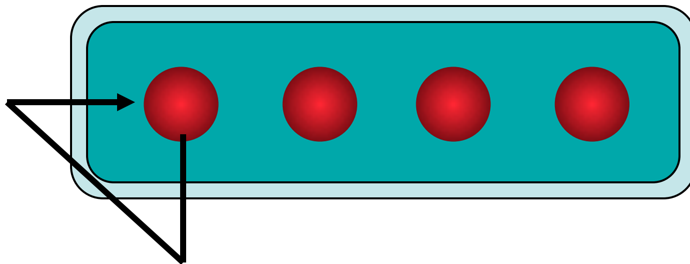


Intercommunicators– P2P

On process 0:

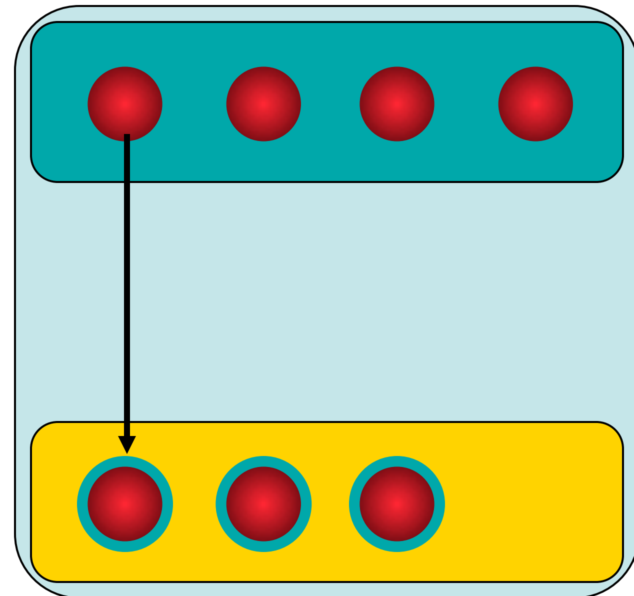
MPI_Send(buf, MPI_INT, 1, 0, tag, intercomm)

- Intracommunicator



Not MPI safe if the receive was not posted before.

- Intercommunicator



Communicators - Collectives

- Simple classification by operation class
- **One-To-All** (simplex mode)
  - One process contributes to the result. All processes receive the result.
    - MPI_Bcast
    - MPI_Scatter, MPI_Scatterv
- **All-To-One** (simplex mode)
  - All processes contribute to the result. One process receives the result.
    - MPI_Gather, MPI_Gatherv
    - MPI_Reduce
- **All-To-All** (duplex mode)
  - All processes contribute to the result. All processes receive the result.
    - MPI_Allgather, MPI_Allgatherv
    - MPI_Alltoall, MPI_Alltoallv
    - MPI_Allreduce, MPI_Reduce_scatter
- **Other**
  - Collective operations that do not fit into one of the above categories.
    - MPI_Scan
    - MPI_Barrier

Collectives

	Who generate the result	Who receive the result
One-to-all	One in the local group	All in the local group
All-to-one	All in the local group	One in the local group
All-to-all	All in the local group	All in the local group
Others	?	?

Extended Collectives

From each process point of view

	Who generate the result	Who receive the result
One-to-all	One in the local group	All in the remote group
All-to-one	All in the local group	One in the remote group
All-to-all	All in the local group	All in the remote group
Others	?	?

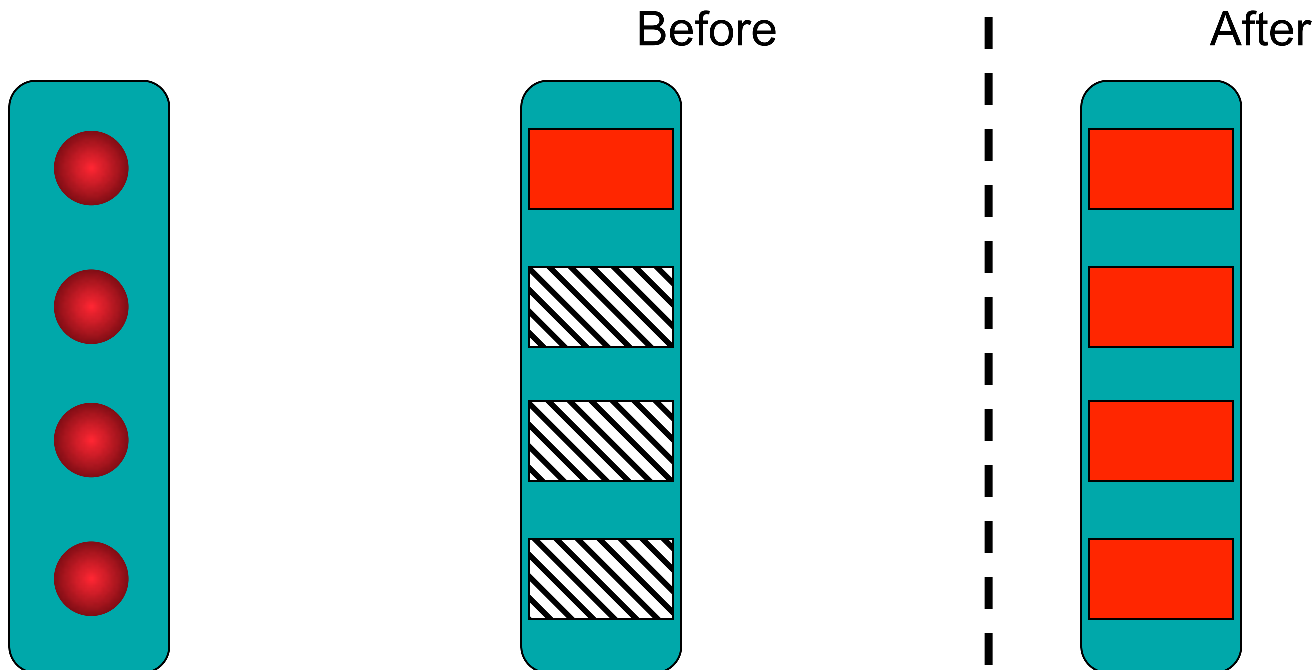
Extended Collectives

- Simplex mode (ie. rooted operations)
  - A root group
    - The root use MPI_ROOT as root process
    - All others use MPI_PROC_NULL
  - A second group
    - All use the real rank of the root in the remote group
- Duplex mode (ie. non rooted operations)
  - Data send by the process in one group is received by the process in the other group and vice-versa.

Broadcast

One-to-all	One in the local group	All in the local group

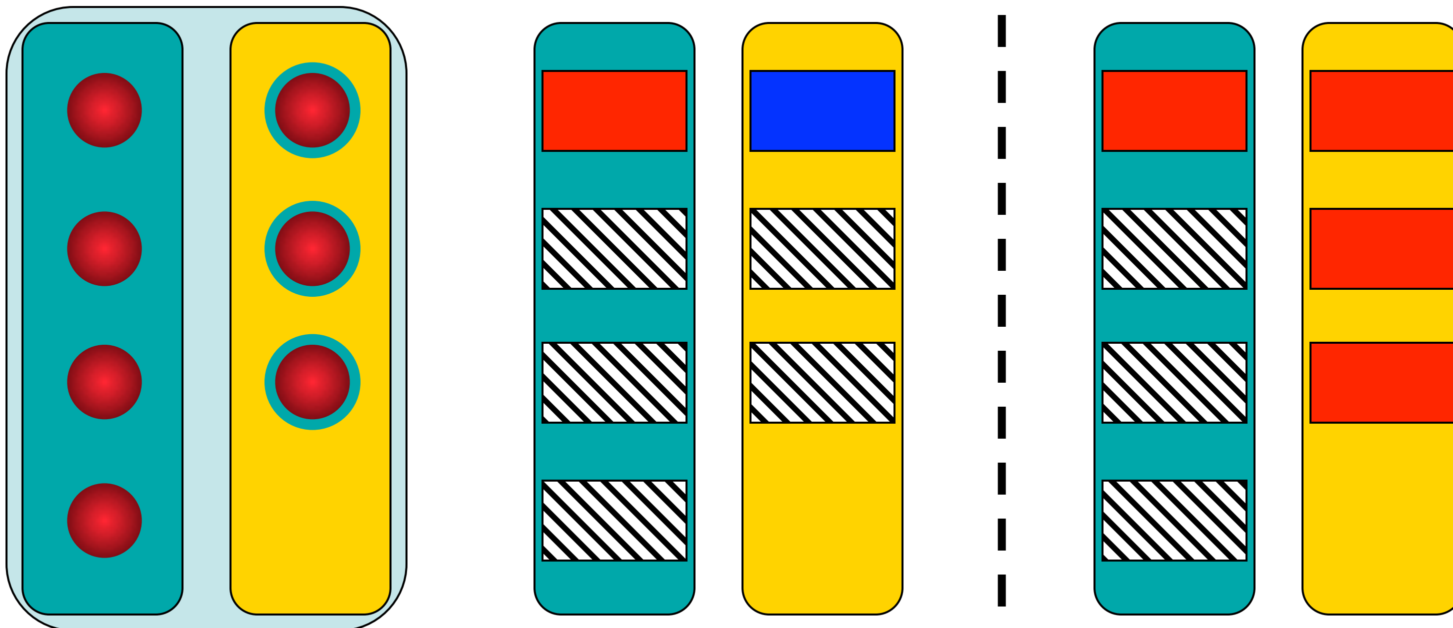
MPI_Bcast(buf, 1, MPI_INT, 0, intracomm)



Extended Broadcast

One-to-all	One in the local group	All in the remote group

Root group root process: MPI_Bcast(buf, 1, MPI_INT, MPI_ROOT, intercomm)
Root group other processes: MPI_Bcast(buf, 1, MPI_INT, MPI_PROC_NULL, intercomm)
Other group MPI_Bcast(buf, 1, MPI_INT, root_rank, intercomm)



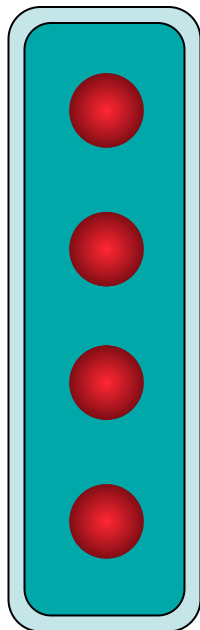
Allreduce

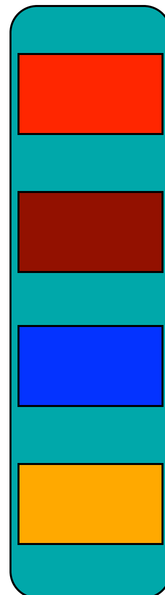
All-to-all	All in the local group	All in the local group

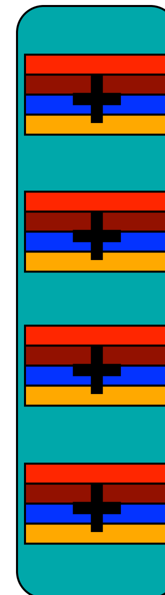
MPI_Allreduce(sbuf, rbuf, 1, MPI_INT, +, intracomm)

Before

After





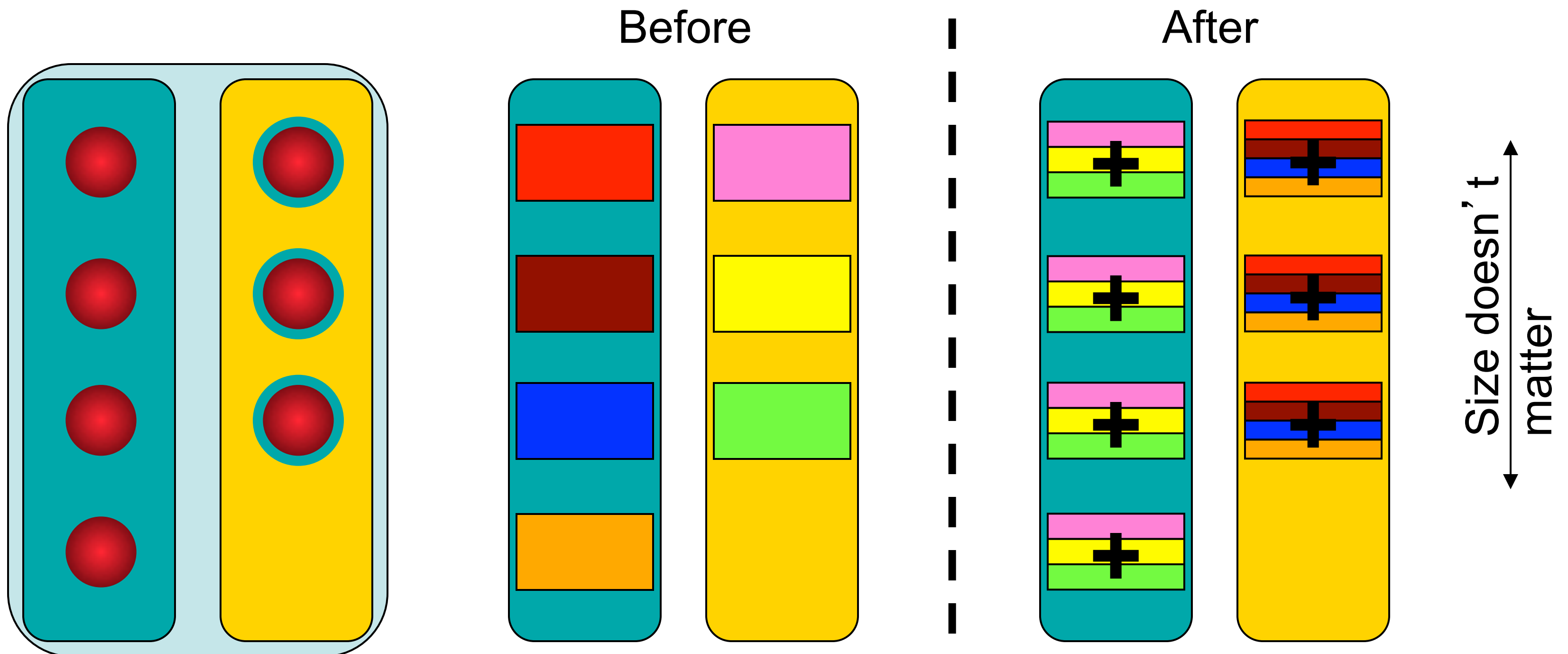


Size doesn't matter

Extended Allreduce

All-to-all	All in the local group	All in the remote group

MPI_Allreduce(sbuf, rbuf, 1, MPI_INT, +, intercomm)

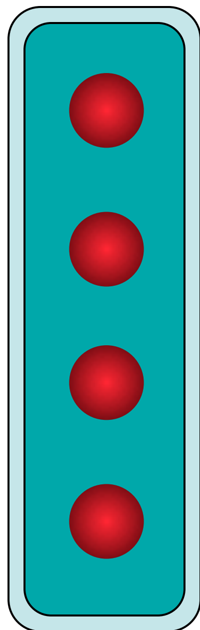


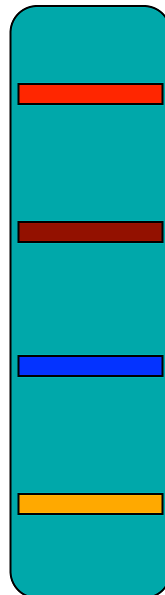
AllGather

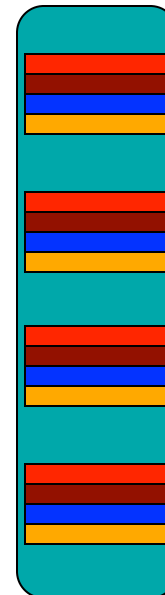
All-to-all	All in the local group	All in the local group

MPI_Allgather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, intracomm)

Before | After





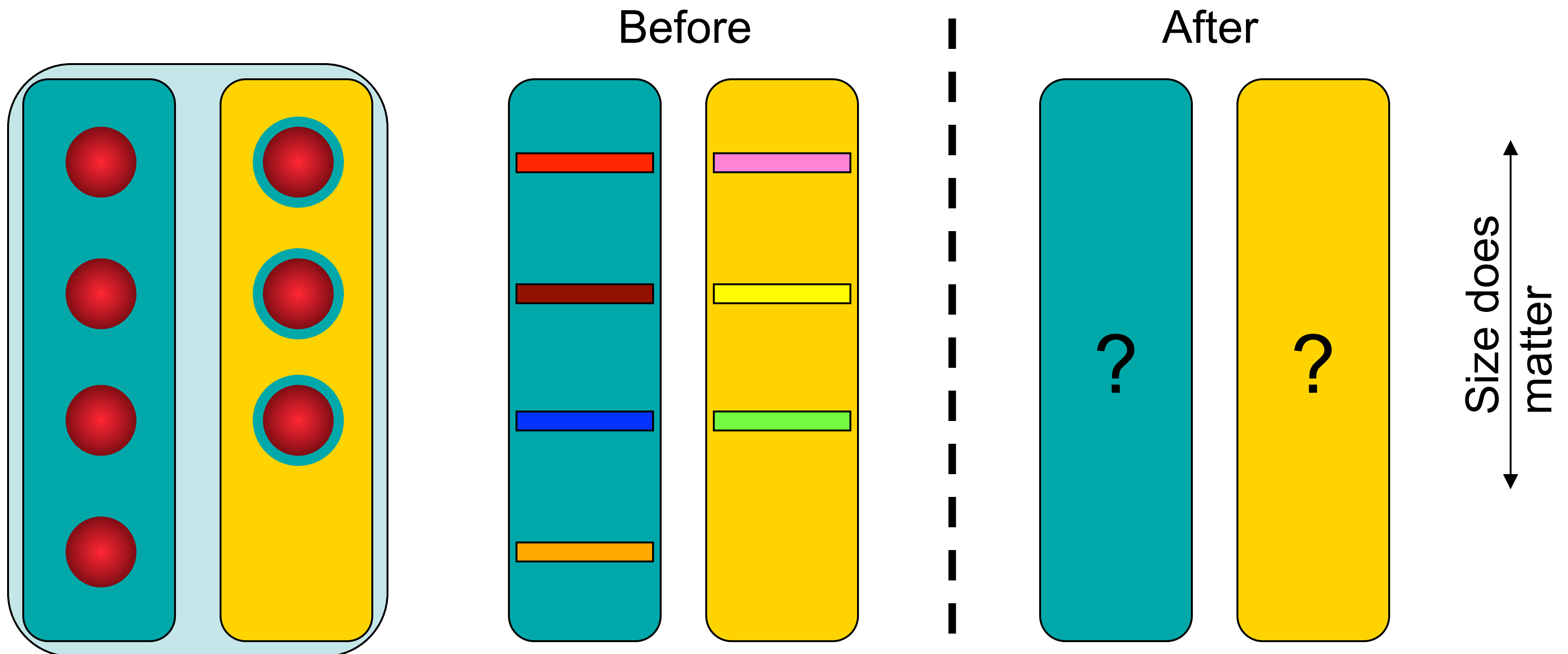


Size does matter

Extended AllGather

All-to-all	All in the local group	All in the remote group

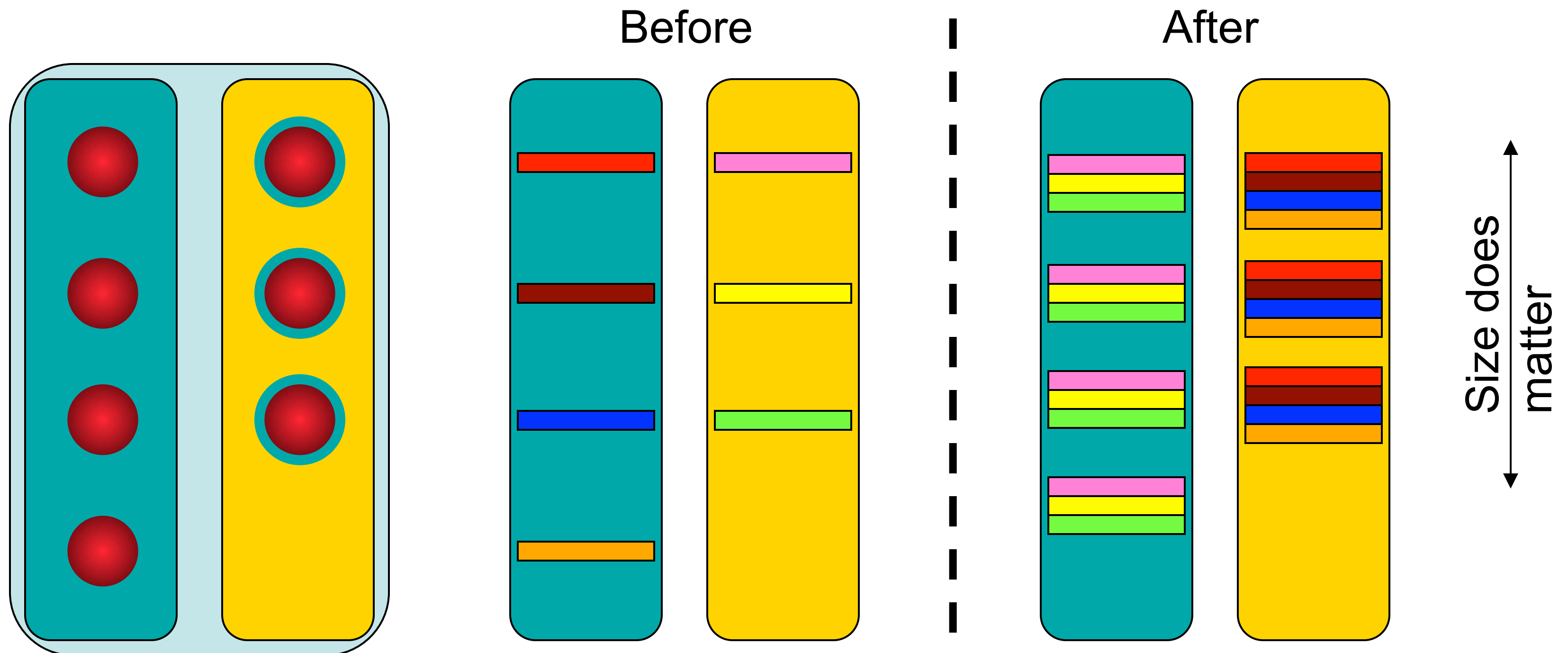
MPI_Allgather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, intercomm)



Extended AllGather

All-to-all	All in the local group	All in the remote group

MPI_Allgather(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, intercomm)



Scan/Exscan and Barrier

- Scan and Exscan are illegal on intercommunicators
- For MPI_Barrier all processes in a group may exit the barrier when all processes on the other group have entered in the barrier.