Message Passing Interface

George Bosilca
bosilca@eecs.utk.edu

MPI 1 & 2

- MPI 1
  - MPI Datatype
  - Intra/Inter Communicators
- MPI 2
  - Process management
  - Connect/Accept
  - MPI I/O

MPI Point-to-point communications
**Send & Receive**

- Explicit communications
- 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

```c
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

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

---

**MPI Derived 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 Datatypes**

- 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**

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C datatype</th>
<th>Fortran datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
<td></td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
<td>DOUBLE PRECISION*8</td>
</tr>
</tbody>
</table>

**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

```c
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?

  ```c
  struct {
    char c1;
    int i;
    char c2;
    double d;
  }
  ```

  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 ?

Memory Layout

```c
```
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.
  \[ \text{MPI\_Type\_contiguous}( 3, \text{oldtype}, \text{newtype} ) \]

Vectors

• Replication of a datatype into locations that consist of equally spaced blocks
  \[ \text{MPI\_Type\_vector}( 7, 2, 3, \text{oldtype}, \text{newtype} ) \]

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
  \[ \text{MPI\_Type\_indexed}( \text{count}, \text{array\_of\_blocks}, \text{array\_of\_disps}, \text{oldtype}, \text{newtype} ) \]
Indexed Blocks

array_of_blocklengths[] = { 2, 3, 1, 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

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

```c
Array_of_lengths[] = { 2, 1 };  
Array_of_displs[] = { 0, 2*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<truetype>, 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")

```c
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

```c
Struct {
  int gap[2];
  int [6];
}
```

---

**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

```c
Typemap = { (type_o, disp_o), ..., (type_n, disp_n) }

lb(Typemap) = Min\_disp[n] (disp such that type\_n = lb) if no entry has type lb
b otherwise

ub(Typemap) = Max\_disp[n] + size(type\_n) + align if no entry has type ub
b otherwise
```
**MPI LB and MPI UB**

\[
\text{displs} = (-3, 0, 6) \\
\text{blocklengths} = (1, 1, 1) \\
\text{types} = (\text{MPI LB, MPI INT, MPI UB}) \\
\text{MPI Type struct}(3, \text{displs}, \text{blocklengths}, \text{types}, \text{type1})
\]

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

\[
\text{MPI Type contiguous}(3, \text{type1}, \text{type2})
\]

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 datatype 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

\[
\text{MPI Type create resized (MPI Datatype datatype, MPI Aint lb, MPI Aint extent, MPI Datatype*newtype)}
\]

- 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.

\[
\text{Typemap} = \{(\text{type}_\text{lb}, \text{disp}_\text{lb}), \ldots, (\text{type}_\text{ub}, \text{disp}_\text{ub})\}
\]

\[
\text{true}_{\text{lb}}(\text{Typemap}) = \min\{\text{disp}_\text{lb}, \ldots, \text{disp}_\text{ub}\} \\
\text{true}_{\text{ub}}(\text{Typemap}) = \max\{\text{disp}_\text{lb} + \text{sizeof}\text{(type}_\text{lb}) , \ldots, \text{disp}_\text{ub} + \text{sizeof}\text{(type}_\text{ub})\}
\]
Information About Datatypes

Decoding a datatype

- Sometimes is important to know how a datatype was created (e.g., 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_envelope

- The combiner field returns how the datatype was created, e.g.
  - MPI_COMBINER_NAMED: basic datatype
  - MPI_COMBINER_CONTIGUOUS: 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 )

Exercise

• Goals:  
  – Create a datatype describing a matrix diagonal  
  – What's different between C and Fortran?

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 * ∈ {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 * ∈ {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 * ∈ {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( comm1, 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

```c
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 assignment

<table>
<thead>
<tr>
<th>rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>color</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>key</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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, C, I\}
3. \{H\} with ranks \{1\} => \{H\}

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

- Intercommunicator
- Group (A)
- Group (B)

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

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

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

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, new_intercomm)
  - `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)

```
high = false
high = true
```

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

N = 3

Not MPI safe if the receive was not posted before.

Intercommunicators – P2P

On process 0:
MPI_Send(buf, MPI_INT, 1, 0, tag, intercomm)

- Intracommunicator

- Intercommunicator

N = 3
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

<table>
<thead>
<tr>
<th></th>
<th>Who generate the result</th>
<th>Who receive the result</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-to-all</td>
<td>One in the local group</td>
<td>All in the local group</td>
</tr>
<tr>
<td>All-to-one</td>
<td>All in the local group</td>
<td>One in the local group</td>
</tr>
<tr>
<td>All-to-all</td>
<td>All in the local group</td>
<td>All in the local group</td>
</tr>
<tr>
<td>Others</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Extended Collectives

From each process point of view

<table>
<thead>
<tr>
<th></th>
<th>Who generate the result</th>
<th>Who receive the result</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-to-all</td>
<td>One in the local group</td>
<td>All in the remote group</td>
</tr>
<tr>
<td>All-to-one</td>
<td>All in the local group</td>
<td>One in the remote group</td>
</tr>
<tr>
<td>All-to-all</td>
<td>All in the local group</td>
<td>All in the remote group</td>
</tr>
<tr>
<td>Others</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
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 )
```

Before

![Before](image)

After

![After](image)

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 )
```

Before

![Before](image)

After

![After](image)
Allreduce

<table>
<thead>
<tr>
<th>All-to-all</th>
<th>All in the local group</th>
<th>All in the local group</th>
</tr>
</thead>
</table>

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

Before

After

Size doesn't matter

Extended Allreduce

<table>
<thead>
<tr>
<th>All-to-one</th>
<th>All in the local group</th>
<th>All in the remote group</th>
</tr>
</thead>
</table>

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

Before

After

Size doesn't matter

AllGather

<table>
<thead>
<tr>
<th>All-to-all</th>
<th>All in the local group</th>
<th>All in the local group</th>
</tr>
</thead>
</table>

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 )

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.