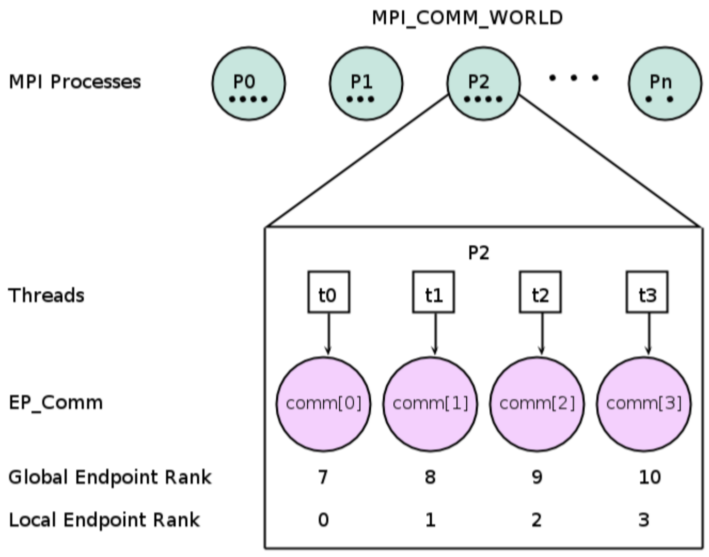
Note for MPI Endpoints

June 26, 2018

1 Purpose

Use threads as if they are MPI processes. Each thread will be assigned a rank and be associated with a endpoints communicator (EP_Comm). Convention: one OpenMP thread corresponds to one endpoint.

2 MPI Endpoints Semantics



Endpoints are created from one MPI communicator and the number of available threads:

```
int MPI_Comm_create_endpoints(MPI_Comm parent_comm, int num_ep,
                              MPI_Info info, MPI_Comm out_comm_hdls[])
```

"In this collective call, a single output communicator is created, and an array of `num_ep` handles to this new communicator are returned, where the i^{th} handle corresponds to the i^{th} rank requested by the caller of `MPI_Comm_create_endpoints`. Ranks in the output communicator are ordered sequentially and in the same order as the parent communicator. After it has been created, the output communicator behaves as a normal communicator, and MPI calls on each endpoint (i.e., communicator handle) behave as though they originated from a separate MPI process. In particular, collective calls must be made once per endpoint."[1]

"Once created, endpoints behave as MPI processes. For example, all ranks in an endpoints communicator must participate in collective operations. A consequence of this semantic is that endpoints also have MPI process progress requirements; that operations on that endpoint are required to make progress only when an MPI operation (e.g. `MPI_Test`) is performed on that endpoint. This semantic enables an MPI implementation to logically separate endpoints, treat them independently within the progress engine, and eliminate synchronization in updating their state."[3]

3 EP types

MPI_Comm

`MPI_Comm` is composed by:

- `bool is_ep`: true $\implies$ EP, false $\implies$ MPI classic;
- `int mpi_comm`: handle to the parent MPI communicator;
- `OMPbarrier *ep_barrier`: openMP barrier, used for in-process synchronization and is different from `omp barrier`;
- `int[2] size_rank_info[3]`: topology information of the current endpoint:
  - rank of parent MPI process;
  - size of parent MPI communicator;
  - rank of endpoint, returned by `MPI_Comm_rank`;
  - size of EP communicator, returned by `MPI_Comm_size`;
  - in-process rank of endpoint;
  - in-process size of EP communicator, also noted as the number of endpoints in one MPI process.
- `MPI_Comm *comm_list`: pointer of the first endpoint communicator of one process;
- `Message_list *message_queue`: location of in-coming messages for each endpoint;
- `RANK_MAP *rank_map`: a map composed by an integer and a pair of integers. The integer key represents the rank of an endpoint. The mapped type (pair of integers) gives the in-process rank of the endpoint and the rank of its parent MPI process:

  ```
  rank_map->at(ep_rank)=(ep_rank_local, mpi_rank)
  ```

- `BUFFER *ep_buffer`: buffer (of type `int`, `float`, `double`, `char`, `long`, and `unsigned long`) used for in-process communication.

3.1 MPI_Request

`MPI_Request` is composed by:

- `int mpi_request`: handle to the MPI request;
- `int ep_datatype`: data type of the communication;
- `MPI_Comm comm`: handle to the EP communicator;
- `int ep_src`: rank of the source endpoint;
- `int ep_tag`: tag of the communication.
- `int type`: type of the communication:
  - 1 $\Longrightarrow$ non-blocking send;
  - 2 $\Longrightarrow$ pending non-blocking receive;
  - 3 $\Longrightarrow$ non-blocking matching receive.

3.2 MPI_Status

`MPI_Status` consists of:

- `int mpi_status`: handle to the MPI status;
- `int ep_datatype`: data type of the communication;
- `int ep_src`: rank of the source endpoint;
- `int ep_tag`: tag of the communication.

3.3 MPI_Message

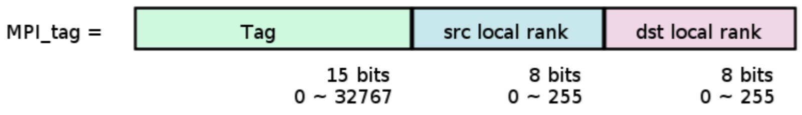
`MPI_Message` includes:

- `int mpi_message`: handle to the MPI message;
- `int ep_src`: rank of the source endpoint;
- `int ep_tag`: tag of the communication.

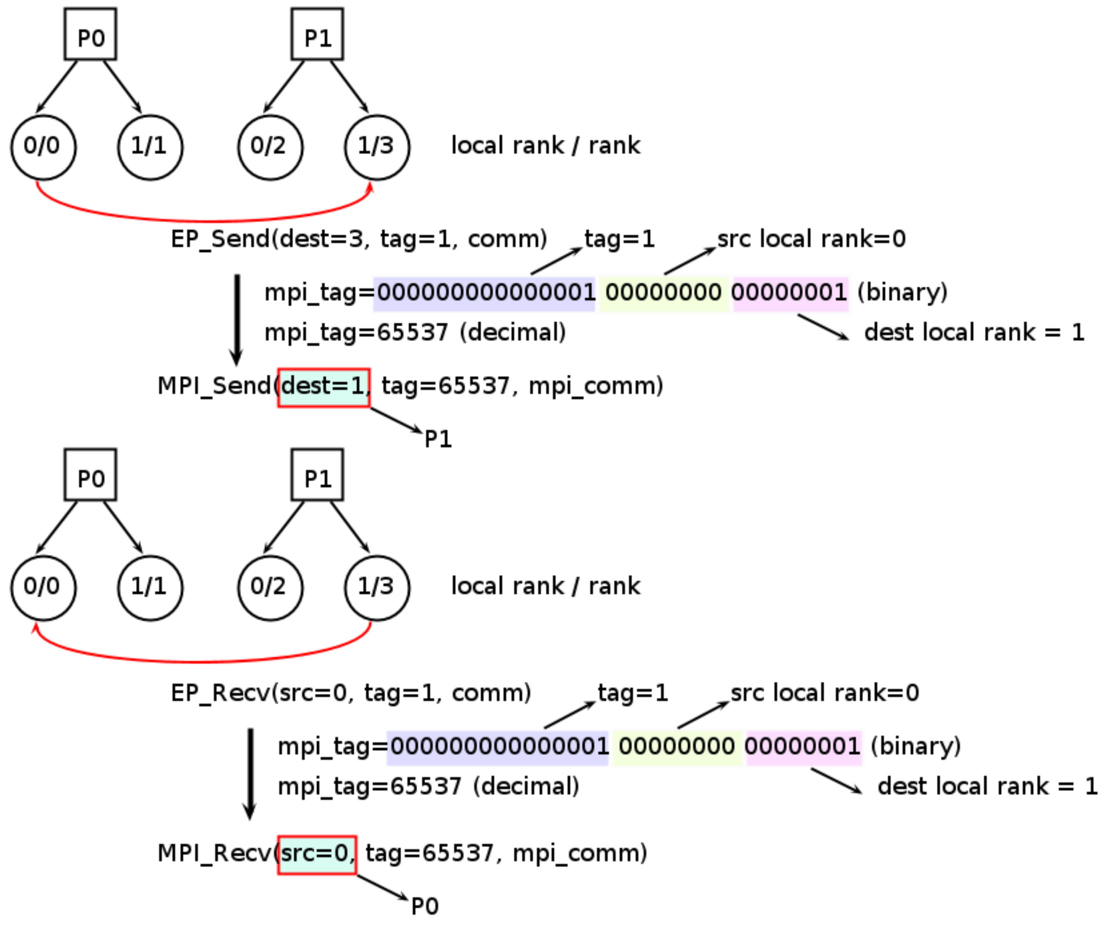
Other types, such as `MPI_Info`, `MPI_Aint`, and `MPI_Fint` are defined in the same way.

4 P2P communication

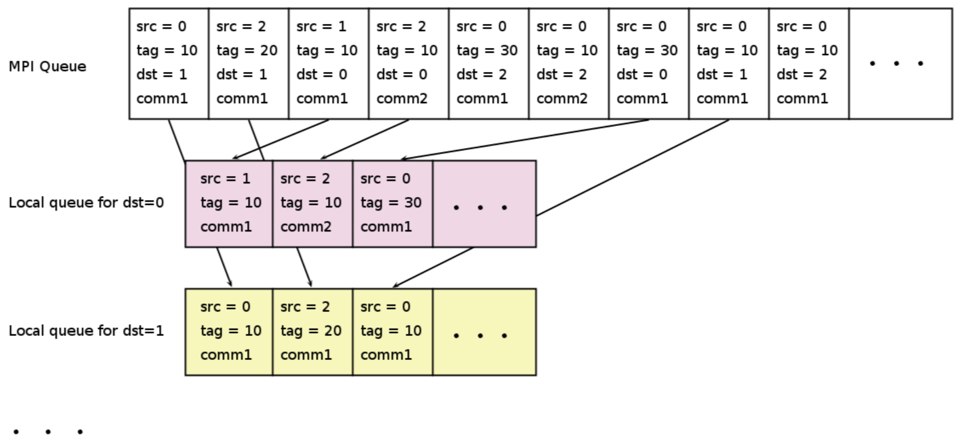
All EP point-to-point communication use tag to distinguish the source and destination endpoint. To be able to add these extra information to tag, we require that the tag value is represented using 31 bits in the underlying MPI inmplemention.



EP_tag is user defined. MPI_tag is internally computed and used inside MPI calls. Because of the extension of tag, wild-cards as `MPI_ANY_SOURCE` and `MPI_ANY_TAG` will not be usable directly. An extra step of tag analysis is needed which leads to the message dequeuing mechanism.


P0
P1
0/0
1/1
0/2
1/3
local rank / rank
EP_Send(dest=3, tag=1, comm)
tag=1
src local rank=0
mpi_tag=000000000000001 00000000 00000001 (binary)
mpi_tag=65537 (decimal)
dest local rank = 1
MPI_Send(dest=1, tag=65537, mpi_comm)
P1
P0
P1
0/0
1/1
0/2
1/3
local rank / rank
EP_Recv(src=0, tag=1, comm)
tag=1
src local rank=0
mpi_tag=000000000000001 00000000 00000001 (binary)
mpi_tag=65537 (decimal)
dest local rank = 1
MPI_Recv(src=0, tag=65537, mpi_comm)
P0

In MPI environment, each MPI process has an incoming message queue. In EP case, messages for all threads inside one MPI process are stored in this MPI queue. With the MPI 3 standard, we use the `MPI_Improbe` routine to inquire the message queue and relocate the incoming message in the local message queue for the corresponding thread/endpoint.



Messages are *non-overtaking* Incoming messages' order is important! If one thread is receiving multiple messages from the same source with the same tag. The receive order should be the same order in which the messages are sent. That is to say, the n-th sent message should be the n-th received message.

Progress "If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process." [2]

When one `EP_Irecv` is issued, we first dequeue the MPI incoming message queue and distribute all incoming messages to the local queues according to the destination identifier. Next, the nonblocking receive request is added at the end of the request pending list. Third, the pending list is checked and requests with matching source, tag, and communicator will be accomplished.

Because of the importance of message order, some communication completion functions must be discussed here such as `MPI_Test` and `MPI_Wait`. "The functions `MPI_Wait` and `MPI_Test` are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive. The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated)." [2]

Example 1 `MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`

1. If `request->type == 1`, communication to be tested is indeed issued from a non-blocking send. The completion status is returned by:

```
MPI_Test(& request->mpi_request, flag, & status->mpi_status)
```

2. If `request->type == 2`, it means that a non-blocking receive is called but the corresponding message is not yet probed. The request is in the pending list thus not yet completed. All incoming message is once again probed and all pending requests are checked. If after the second check, the matching message is found, thus a `MPI_Imrecv` is called and the type is set to 3. Otherwise, the type is still 2, then `flag = false` is returned.

3. If `request->type == 3`, this indcates that the request is issued from a non-blocking receive call and the matching message is probed thus the status of the communication lies in the status of the `MPI_Imrecv` function. The completion result is returned by:

   ```
   MPI_Test(& request->mpi_request, flag, & status->mpi_status)
   ```

Example 2 `MPI_Wait(MPI_Request *request, MPI_Status *status)`

1. If `request->type == 1`, communication to be tested is indeed issued from a non-blocking send. Jump to step 4.

2. If `request->type == 2`, it means that a non-blocking receive is called but the corresponding message is not yet probed. The request is in the pending list thus not yet completed. We repeat the incoming message probing and the pending request checking until the matching message is found, thus a `MPI_Imrecv` is called and the type is set to 3. Jump to step 4.

3. If `request->type == 3`, this indcates that the request is issued from a non-blocking receive call and the matching message is probed thus the status of the communication lies in the status of the `MPI_Imrecv` function. Jump to step 4.

4. We force the completion by calling:

   ```
   MPI_Wat(& request->mpi_request, & status->mpi_status)
   ```

5 Collective communication

All MPI classic collective communications are performed as the following pattern:

1. Intra-process communication using OpenMP. *e.g.* Collect data from slave threads to master thread.

2. Inter-process communication using MPI collective calls on master threads.

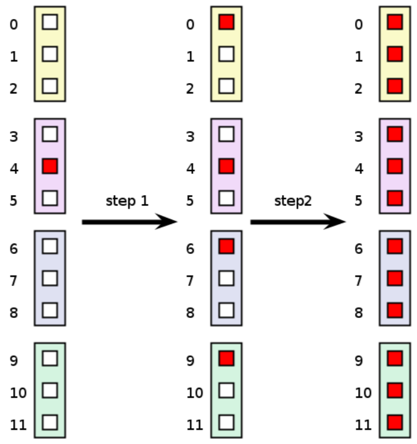
3. Intra-process communication using OpenMP. *e.g.* Distribute data from master thread to slave threads.

Example 1 `EP_Bcast(buffer, count, datatype, root = 4, comm)` with `comm` composed by 4 MPI processes and 3 threads per process:

We can consider the communicator as $\{\underbrace{(0,1,2)}_{\text{proc 0}}\ \underbrace{(3,{\color{red}4},5)}_{\text{proc 1}}\ \underbrace{(6,7,8)}_{\text{proc 2}}\ \underbrace{(9,10,11)}_{\text{proc 3}}\}$.

This collective communication is performed by the following three steps:

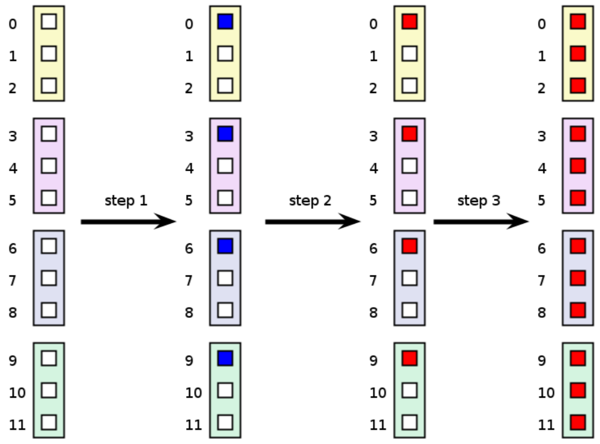
1. We call `MPI_Bcast(buffer, count, datatype, mpi_root = 1, mpi_comm)` with EP processes rank 0, 4, 6, and 9.
2. EP processes rank 0, 4, 6, and 9 send the buffer to its slaves.



Example 2 `EP_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)` with `comm` the same as in example 1.

This collective communication is performed by the following three steps:

1. We perform a intra-process "allreduce" operation: master threads collect data from its slaves and perform the reduce operation.
2. Master threads call the classic `MPI_Allreduce` routine.
3. All master threads send the updated reduced data to its slaves.



Other collective communications have the similar execution pattern.

6 Inter-communicator

In XIOS, inter-communicator is an very important component. Thus, our EP library must support inter-communications.

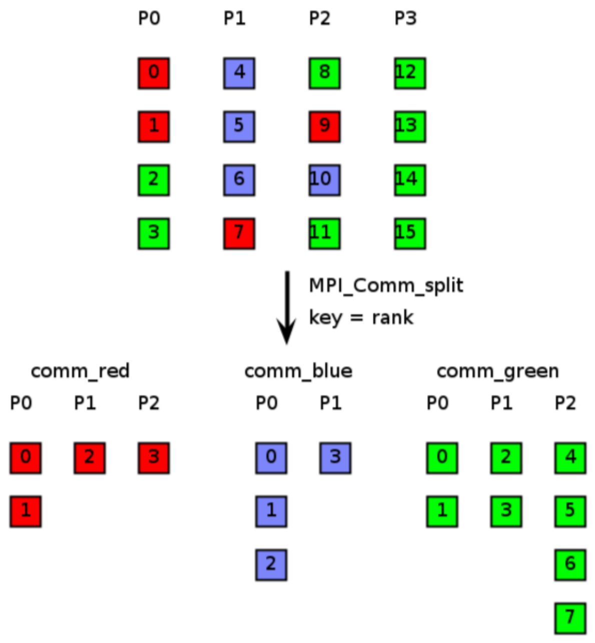
6.1 The splitting of intra-communicator

Before talking about the inter-communicator, we will start by splitting intra-communicator. The C prototype of the splitting routine is

```
int  MPI_Comm_split(MPI_Comm comm, int color, int key,
                                            MPI_Comm *newcomm)
```

"This function partitions the group associated with `comm` into disjoint subgroups, one for each value of `color`. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument `key`, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in `newcomm`. A process may supply the color value `MPI_UNDEFINED`, in which case `newcomm` returns `MPI_COMM_NULL`. This is a collective call, but each process is permitted to provide different values for color and key."[2]

By definition of the routine, in the case of EP, each thread participating the split operation will have only one color (`MPI_UNDEFINED` is also considered to be one color). However, in the process's point of view, it can have multiple colors as shown in the following figure.



This figure shows the result of the EP communicator splitting. Here we used the EP rank as key to assign the new rank of the thread in the resulting split intra-communicator. If the key is anything else than the EP rank, we follow the convention that the key takes effect only inside a process. This means that the threads are at first ordered by the MPI process rank and then by the value of key.

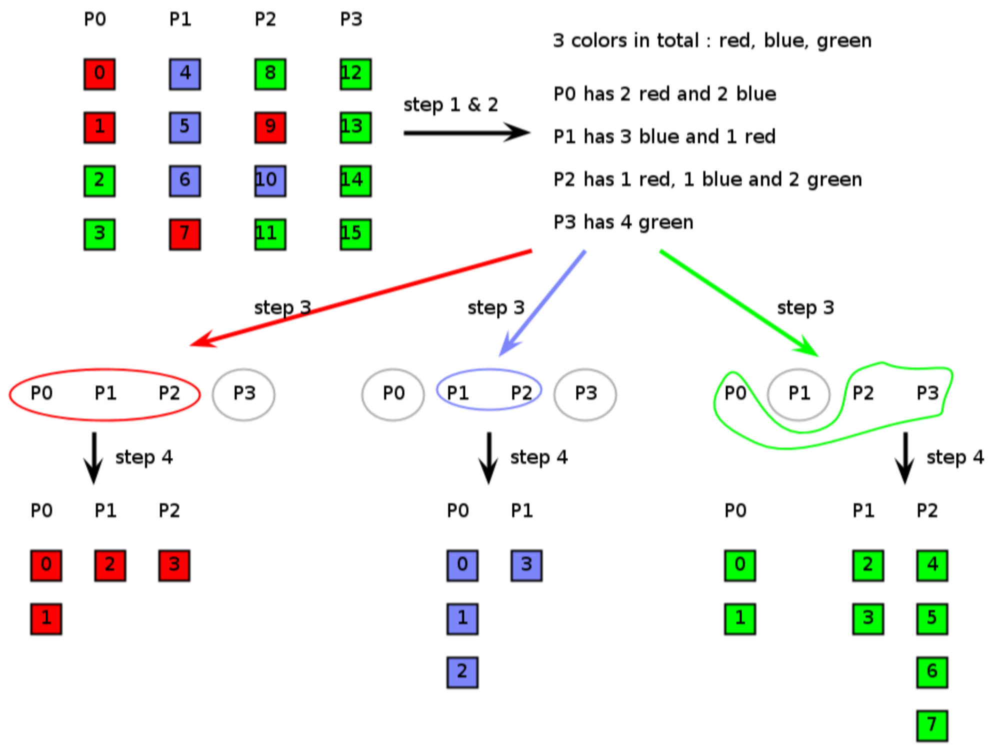
Due to the fact that one process can have multiple colors for its threads, the splitting operation is executed by the following steps:

1. Master threads collect all colors from its slaves and communicate with each other to determine the total number of colors across the communicator.

2. For each color, the master thread check all its slave threads to obtain the number of threads having the same color.

3. If at least one of the slave threads holds the color, then the master thread takes this color. If not, the master thread takes color `MPI_UNDEFINED`. All master threads call classic communicator splitting routine with key = MPI rank.

4. For master threads holding a defined color, we execute the endpoint creation routine according to the number of slave threads holding the same color. The resulting EP communicators are then assigned to these slave threads.



6.2 The creation of inter-communicator

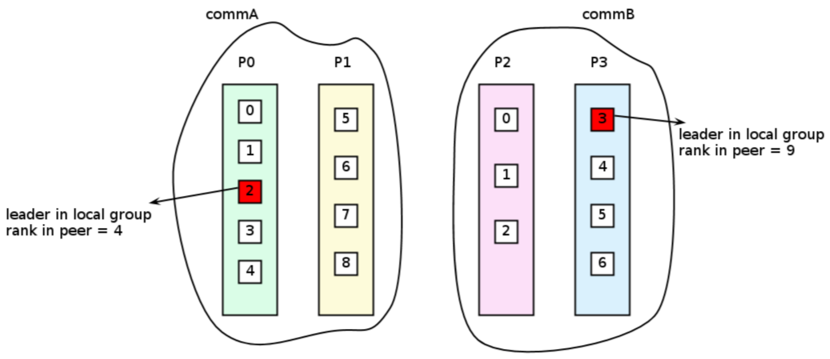
In XIOS, the inter-communicators are create by the routine `MPI_Intercomm_create` which is used to bind two intra-communicators into an inter-communicator. The C prototype is

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
                         MPI_Comm peer_comm, int remote_leader,
                               int tag, MPI_Comm *newintercomm)
```

According to the MPI standard, "an inter-communication is a point-to-point communication between processes in different groups". "All inter-communicator constructors are blocking except for `MPI_COMM_IDUP` and require that the local and remote groups be disjoint."

As in EP the threads are considered as processes, the non-overlapping condition can be translated to "non-overlapping" at the thread level which means that one thread can not belong to the local group and the remote group. However, the parent process of the thread can be overlapped. As the EP library is built upon an existing MPI implementation which follows the non-overlapping condition at the process level, we can have an issue in the case.

Before digging into this issue, we shall at first look at the case where the non-overlapping condition is perfectly respected.



As shown in the figure, we have two intra-communicators A and B and they are totally disjoint both at the thread and process level. Each of the communicators has a local leader. We also assume that both leaders belong to a peer communicator and have rank 4 and 9 respectively.

To create the inter-communicator, all threads from the left intra-comm call:

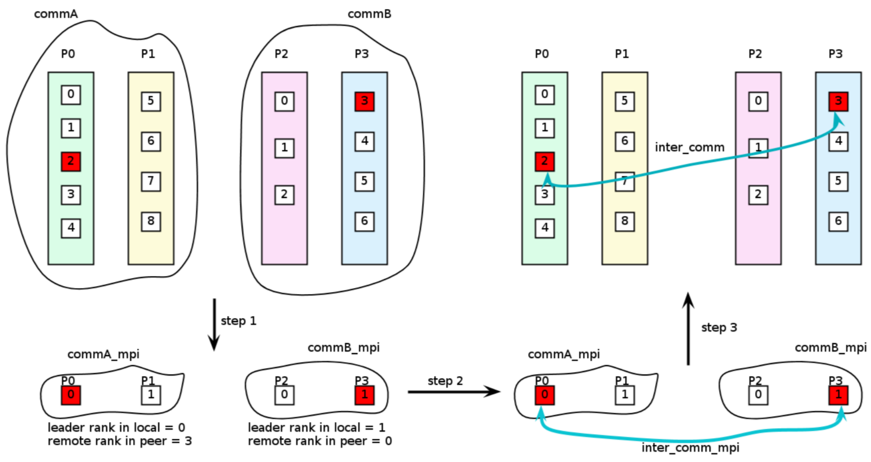
```
MPI_Intercomm_create(commA, local_leader = 2, peer_comm,
                     remote_leader = 9, tag, inter_comm)
```

and for threads of the right intra-comm, they call:

```
MPI_Intercomm_create(commB, local_leader = 3, peer_comm,
                     remote_leader = 4, tag, inter_comm)
```

To perform the inter-communicator creation, we follow the 3 steps:

1. Determine the leaders and ranks at the process level;
2. Call classic `MPI_Intercomm_create`;
3. Create endpoints from process and assigned to threads.



If we have overlapped process in the creation of inter-communicator, we should add an *priority check* to assign the process to only one intra-communicator. Several possibilities:

1. Process is shared and contains no local leader $\Longrightarrow$ process belongs to group with higher rank in peer comm;

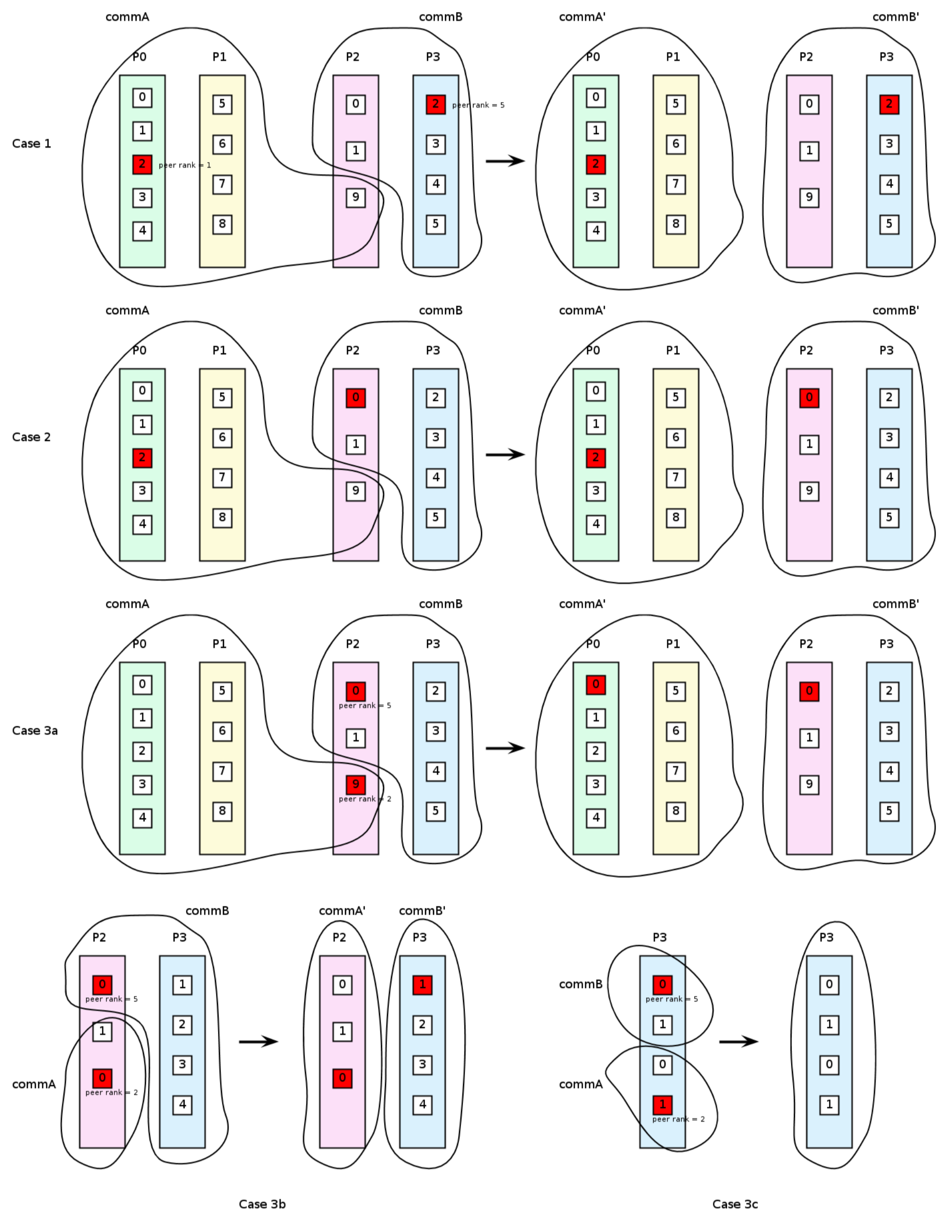
2. Process is shared and contains one local leader $\Longrightarrow$ process belongs to group with the leader;

3. Process is shared and contains both local leaders : leader change is performed and the peer communicator is `MPI_COMM_WORLD` and we note “group A” the group with smaller peer rank and “group B” the group with higher peer rank.

   3a. If group A has at least two processes, the leader of group A is changed to the master thread of the process with smallest rank except the overlapped process. The overlapped process belongs to group B.

   3b. If group A has only one processes, and group B has at least two processes, then the leader of group B is changed to the master thread of the process with smallest rank except the overlapped process. The overlapped process belongs to group A.

   3c. If both group A and group B have only one process, then an one-process intra-communicator is created though it will be considered (labeled) as an inter-communicator.

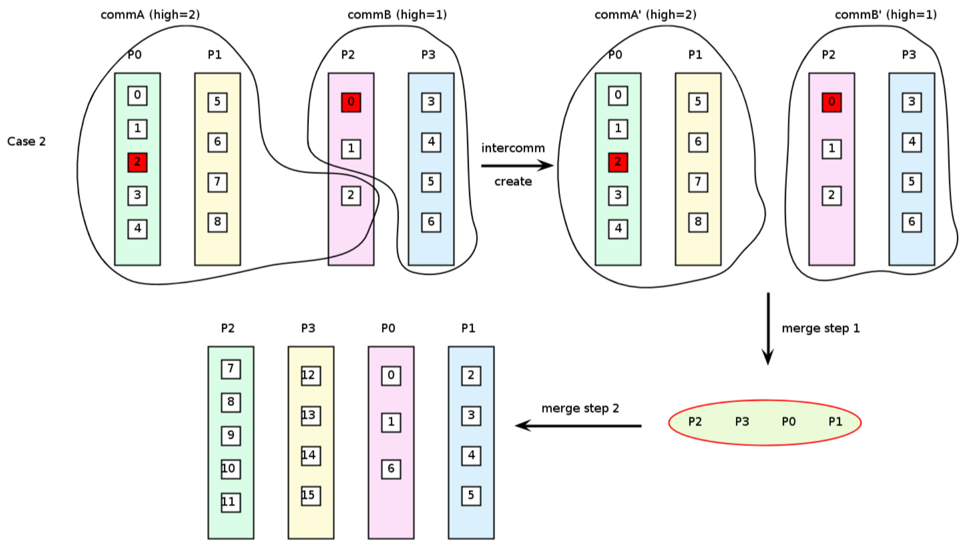


6.3 The merge of inter-communicators

`MPI_Intercomm_Merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)` creates an intra-communicator by merging the local and remote groups of an inter-communicator. All processes should provide the same `high` value within each of the two groups. If processes in one group provided the value `high=false` and processes in the other group provided the value `high=true` then the union orders the “low” group before the “high” group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups. [2]

This routine can be considered as the inverse of `MPI_Intercomm_create`. In the intercommunicator create function, all 5 cases are eventually transformed into the case where no MPI process is shared by two groups. It is from this case that the merge funtion takes place.

1. The classic `MPI_Intercomm_merge` is called and an MPI intracommunicator is created from the two disjoint groups and MPI processes are ordered by the high value of the local leader.
2. Endpoints are created based on the MPI intracommunicator and the new EP ranks are orderd firstly according to the high value of each thread and then to the origianl EP ranks in the intercommunicators.

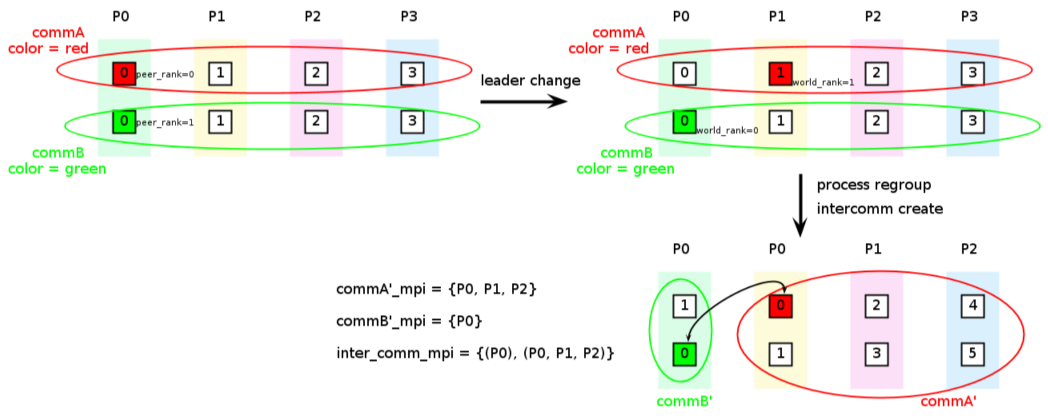


7 P2P communication on inter-communicators

In case of the intercommunicators, the `MPI_Comm` class has 3 members to determine the topology along with the original `rank_map`:

- `RANK_MAP local_rank_map[size of commA]`: composed of the EP rank in commA' or commB';
- `RANK_MAP remote_rank_map[size of commB]`: = `local_rank_map` of remote group;
- `RANK_MAP intercomm_rank_map[size of commB']`: = `rank_map` of remote group';
- `RANK_MAP rank_map`: rank map of commA' or commB'.

For example, in the following configuration:



For all endpoints in commA,

```
local_rank_map={(rank in commA’ or commB’,
                 rank of leader in MPI_Comm_world)}
              ={(1,0), (0,1), (2,1), (4,1)}

remote_rank_map={(remote endpoints’ rank in commA’ or commB’,
                  rank of remote leader in MPI_Comm_world)}
               ={(0,0), (1,1), (3,1), (5,1)}
```

For all endpoints in commA’

```
intercomm_rank_map={(remote endpoints local rank in commA’ or commB’,
                     remote endpoints MPI rank in commA’ or commB’)}
                  ={(0,0), (1,0)}
rank_map={(local rank in commA’, mpi rank in commA’)}
        ={(0,0), (1,0), (0,1), (1,1), (0,2), (1,2)}
```

For all endpoints in comm B,

```
local_rank_map={(rank in commA’ or commB’,
                 rank of leader in MPI_Comm_world)}
              ={(0,0), (1,1), (3,1), (5,1)}

remote_rank_map={(remote endpoints’ rank in commA’ or commB’,
                  rank of remote leader in MPI_Comm_world)}
               ={(1,0), (0,1), (2,1), (4,1)}
```

For all endpoints in commB’

```
intercomm_rank_map={(remote endpoints local rank in commA’ or commB’,
                     remote endpoints MPI rank in commA’ or commB’)}
                  ={(0,0), (1,0), (0,1), (1,1), (0,2), (1,2)}
rank_map={(local rank in commB’, mpi rank in commB’)}
        ={(0,0), (1,0)}
```

When calling a p2p communication on an inter-communicator, we should:

1. Determine if the source and the destination endpoints are in a same group by checking the “labels”.

- `src_label = local_rank_map->at(src).second`
- `dest_label = remote_rank_map->at(dest).second`

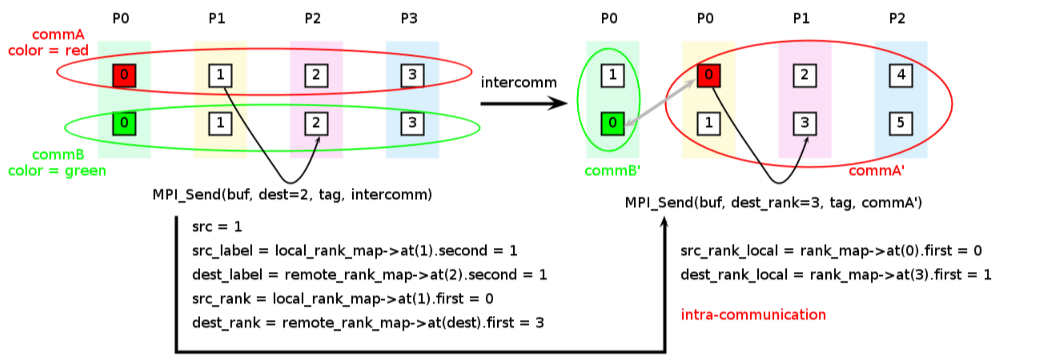
2. If `src_label == dest_label`, then the communication is in fact a intra-communication. The new source rank and destination rank, as well as the local ranks, are deduced by:

```
src_rank = local_rank_map->at(src).first
dest_rank = remote_rank_map->at(dest).first
src_rank_local = rank_map->at(src_rank).first
dest_rank_local = rank_map->at(dest_rank).first
```

3. If `src_label != dest_label`, then the inter-communication is required. The new ranks are obtained by:

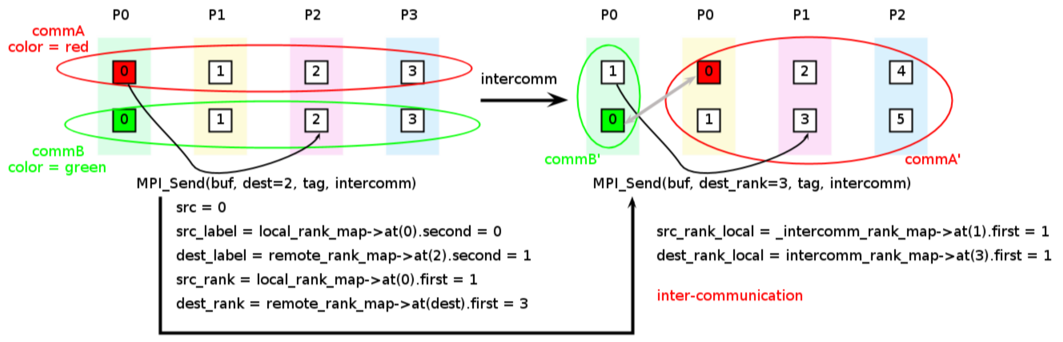
```
src_rank = local_rank_map->at(src).first
dest_rank = remote_rank_map->at(dest).first
src_rank_local = intercomm_rank_map->at(src_rank).first
dest_rank_local = rank_map->at(dest_rank).first
```

4. Call MPI P2P function to start the communication.
   - If intra-communication, `mpi_comm = commA'_mpi or commB'_mpi`;
   - If inter-communication, `mpi_comm = inter_comm_mpi`.



```
MPI_Send(buf, dest=2, tag, intercomm)
src = 1
src_label = local_rank_map->at(1).second = 1
dest_label = remote_rank_map->at(2).second = 1
src_rank = local_rank_map->at(1).first = 0
dest_rank = remote_rank_map->at(dest).first = 3

MPI_Send(buf, dest_rank=3, tag, commA')
src_rank_local = rank_map->at(0).first = 0
dest_rank_local = rank_map->at(3).first = 1
```



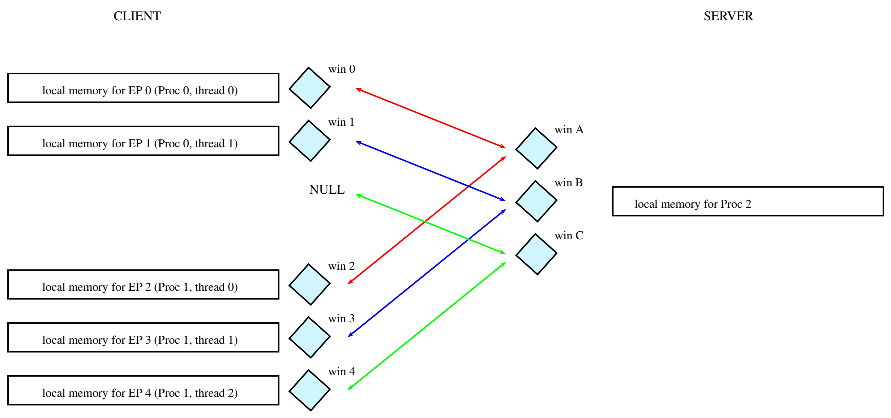
```
MPI_Send(buf, dest=2, tag, intercomm)
src = 0
src_label = local_rank_map->at(0).second = 0
dest_label = remote_rank_map->at(2).second = 1
src_rank = local_rank_map->at(0).first = 1
dest_rank = remote_rank_map->at(dest).first = 3

MPI_Send(buf, dest_rank=3, tag, intercomm)
src_rank_local = _intercomm_rank_map->at(1).first = 1
dest_rank_local = intercomm_rank_map->at(3).first = 1
```

8 One-sided communications

The one-sided communication is a type of communcation which involves only one process to specify all communication parameters, both for the sending side and the receiving side [2, Chapter 11]. To extend this type of communication in the context of endpoints, we encounter some limitations. In the current work, the one-sided communication can only be used in the client-server mode which means that RMA(remote memory access) can occur only between a server and a client.

The construction of RMA windows is illustrated by the following figure:


CLIENT
SERVER
local memory for EP 0 (Proc 0, thread 0)
local memory for EP 1 (Proc 0, thread 1)
local memory for EP 2 (Proc 1, thread 0)
local memory for EP 3 (Proc 1, thread 1)
local memory for EP 4 (Proc 1, thread 2)
win 0
win 1
NULL
win 2
win 3
win 4
win A
win B
win C
local memory for Proc 2

- we determin the max number of threads N in the endpoint environment (N=3 in the example);
- on the server side, N windows are declared and asociated with the same memory adress;
- we start a loop : i = 0, ..., N-1
  - each endpoint with thread number i declares an RMA window;
  - the link between windows on the client side and the i-th window on the server side are created via `MPI_Win_created`;
  - if the number of threads on a certain process is less than N, then a `NULL` pointer is used as memory adress.

With the RMA windows created, we can then perform some communications: `MPI_Put`, `MPI_Get`, `MPI_Accumulate`, `MPI_Get_accumulate`, `MPI_Fetch_and_op`, `MPI_Compare_and_swap`, *etc.*

The main idea of any of the mentioned communications is to identify the threads which are involved in the connection. For example, we want to perform a put operation from EP 2 to the server. We know that EP 2 is the thread 0 of process 1. Thus the 0-th window (win A) of the server side should be used. Once the sender and the receiver are identified, the `MPI_Put` communication can be established.

Other RMA functions, such as `MPI_Win_allocate`, `MPI_win_Fence`, and `MPI_Win_free`, remain nearly the same and we will skip the detail in this document.

References

[1] J. Dinan, Pavan Balaji, D. Goodell, D. Miller, M. Snir, and Rajeev Thakur. Enabling mpi interoperability through flexible communication endpoints. In *EuroMPI 2013*, Madrid, Spain, 2013.

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1.* 2015.

[3] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multi-threaded mpi communication through a library-based implementation of mpi endpoints. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498, Nov 2014.