Flexible RMA Synchronization for MPI-3

Hubert Ritzdorf

NEC Laboratories Europe
IT Research Division
ritzdorf@it.neclab.eu

15. April 2008
Version 1.0

1.1 Introduction

Aim of the proposal is to give a single process or a set of processes of application programs the possibility to signal a target process (i.e. a MPI process from which data was read or to which data was written by RMA communication calls) that the writing of data or reading of data is completed and that the target process is able to proceed. In many native shared memory programs, such a completion is signaled by shared memory “flags” which are set to 0 or 1. In order to improve the scalability and shared memory usage of this proposal, shared memory “counters” are used instead of “flags”. These “counters” count the number of processes which have signaled the completion of RMA communications calls to a target process.

In the actual proposal, the RMA communication calls could be used unchanged together with the newly proposed synchronization calls.
Note: Multi-threaded application programs and possibly the error checking in standard application programs would probably benefit if future RMA calls would have the possibility to specify the corresponding synchronization counters/objects in the RMA communication call.

The actual proposal can be implemented on cache-coherent and on non-cache-coherent systems.

1.2 Synchronizing RMA Requests

As already mentioned in the beginning of this proposal, new synchronizations objects "counters", which could be realized by (atomic) counters, are defined. These synchronization counters have to be probably allocated in special memory regions (shared memory, special registers ...) so that special MPI allocation and free routines have to be provided for these synchronization counters.

A new MPI type MPI_Sync is defined in order to manage the specific data on synchronization counters. However, the special data on the new MPI type has to be managed not only on the local process which contains the "counter" but also on the remote process which has to remotely manipulate (increment/decrement) this synchronization counter. Therefore, this new MPI type MPI_Sync acts in 2 ways depending on the location of the synchronization counter:

[local] Within the actual process, the MPI_Sync variables contain the data on the local synchronization counter which have to be allocated by function MPI_WIN_ALLOC_SYNC_OBJECTS (see Section 1.2.1).

[remote] Within the remote process, the MPI_Sync variables contain the data how to access and manipulate the synchronization counters which are located on a target process. The corresponding MPI_Sync variables can be allocated in standard way (statically or dynamically by standard malloc function). The data required to access and manipulate the synchronization counters on the remote process can be transferred with the newly introduced datatype MPI_HANDLE_SYNC (cf. Section 1.2.2).

Note: These two different ways of allocation of MPI_Sync *variables are not really optimal and might cause problems when implementing the synchronization counters into application programs. But I didn't want to define 2 different kinds of* MPI *types.*

1.2.1 Allocation of Synchronization Objects

This section describes the functions to allocate and free the synchronization objects which contain the data on the local synchronization counter.

MPI_WIN_ALLOC_SYNC_OBJECTS(n_sync, sync_counters, win, info)

IN	n_sync	number of sync objects to be allocated (integer)
OUT	sync_counters	sync objects (handles)
IN	info	info argument (handle)
IN	win	window object (handle)

```
int MPI_Win_alloc_sync_objects(int n_sync,
            MPI_Sync *sync_counters, MPI_Win win,
            MPI_Info info)
```

```
MPI_WIN_ALLOC_SYNC_OBJECTS (N_SYNC, SYNC_COUNTERS, WIN,
            INFO, IERROR)
   INTEGER N_SYNC, SYNC_COUNTERS(*), WIN, INFO, IERROR
```

```
void MPI::WIN::Alloc_sync_objects(int n_sync,
            MPI_Sync *sync_counters, MPI_Info info)
            const
```

A call to MPI_WIN_ALLOC_SYNC_OBJECTS allocates n_sync synchronization counters and returns the handles to these counters in sync_counters.

MPI_WIN_FREE_SYNC_OBJECTS(n_sync, *sync_counters, win)

IN	n_sync	number of sync objects to be freed (integer)
INOUT	sync_counters	sync objects (handles)
IN	win	window object (handle)

```
int MPI_Win_free_sync_objects(int n_sync,
            MPI_Sync *sync_counters, MPI_Win win)
```

```
MPI_WIN_FREE_SYNC_OBJECTS (N_SYNC, SYNC_COUNTERS, WIN,
            IERROR)
   INTEGER N_SYNC, SYNC_COUNTERS(*), WIN, IERROR

void MPI::WIN::Free_sync_objects(int n_sync,
            MPI_Sync *sync_counters) const
```

A call to MPI_WIN_FREE_SYNC_OBJECTS frees n_sync synchronization counters. The entries of sync_counters[] are set to MPI_SYNC_NULL.

1.2.2 MPI Datatype MPI_HANDLE_SYNC

The MPI processes which perform RMA function calls, and which have to call the synchronization function MPI_WIN_SYNC_OPS_INIT, need the information of the synchronization counter on the target process. In order to transfer this information MPI provides the MPI predefined datatype MPI_HANDLE_SYNC, which can be used to transfer data on the MPI_Sync synchronization counters allocated by function MPI_WIN_ALLOC_SYNC_OBJECTS to other processes.

The predefined datatype MPI_HANDLE_SYNC is allowed to be used only with MPI pt2pt communication functions and MPI collectives without the reduce and scan functions. It's not allowed to use datatype MPI_HANDLE_SYNC in generation of derived datatypes.
Note: One sided communication of datatype MPI_HANDLE_SYNC is not allowed since the receiving process must be able to translate the information received.

When using datatype MPI_HANDLE_SYNC in communication functions, the input values should be handles returned by MPI_WIN_ALLOC_SYNC_OBJECTS and the output are MPI_SYNC handles which contain to the corresponding remote synchronization information.

1.2.3 Synchronization Calls

Many interconnects have hardware coprocessors which can transfer the data independently on the actual processors which perform the computations of the application program. Therefore, the synchronization calls are designed as persistent function calls (persistent for performance reasons) which return MPI requests.

This means that RMA synchronization requests can be started by MPI_Start or MPI_Startall, the status of the synchronizations requests can be evaluated by the MPI test functions and it can be waited for the synchronization by the MPI wait functions.

Notes: It doesn't make me happy to allow canceling of such synchronization requests since the performance of this synchronization objects should not be disturbed by additional communication in order to enable canceling such request.

What does cancel of a synchronization object mean ?

The function MPI_WIN_SYNC_OPS_INIT is designed to wait for the completion of locally issued RMA function calls to a target process and to signal the target process that these RMA function calls are completed. Function MPI_WIN_SYNC_OPS_INIT uses the remote data on the synchronization counters ((cf. Section 1.2)).

The function MPI_WIN_SYNC_OBJECT_INIT is designed to inform the actual process that (remotely) issued RMA function calls are completed and the actual process can access the data. Function MPI_WIN_SYNC_OBJECT_INIT uses the local data on the synchronization counters (cf. Section 1.2) and the synchronization counters must be allocated by function MPI_WIN_ALLOC_SYNC_OBJECTS.

MPI_WIN_SYNC_OPS_INIT(target_rank, sync_mode, sync_counter, win, info, req)

IN	target_rank	rank of target (nonnegative integer)
IN	sync_mode	synchronization mode (integer)
INOUT	sync_counter	sync object (handle)
IN	win	window object (handle)
IN	info	info argument (handle)
OUT	req	request (handle)

```
int MPI_Win_sync_ops_init(int target_rank,
           int sync_mode, MPI_Sync sync_counter,
           MPI_Win win, MPI_Info info,
           MPI_Request *req)
```

```
MPI_WIN_SYNC_OPS_INIT (TARGET_RANK, SYNC_MODE, WIN,
            INFO, REQ, IERROR)
   INTEGER TARGET_RANK, SYNC_MODE, SYNC_COUNTER, WIN,
   INFO, REQ, IERROR
```

```
MPI_Prequest MPI::WIN::Sync_ops_init(int target_rank,
            int sync_mode, MPI_Sync sync_counter,
            MPI_Info info) const
```

A call to MPI_WIN_SYNC_OPS_INIT creates and initializes a persistent request handle for the synchronization of RMA communication requests to/from (remote) process target_rank.

There are currently 3 RMA communication calls (MPI_Get, MPI_Put and MPI_ACCUMULATE). The application program can specify synchronization modes to the synchronization functions where the synchronization mode corresponds the currently available RMA communication calls. The following synchronization modes are supported (specified in sync_mode, a bit vector OR of the following integer constants) for target process target_rank:

- MPI_MODE_WIN_PUT – synchronize MPI_PUT's to the target process
- MPI_MODE_WIN_GET – synchronize MPI_GET's from the target process
- MPI_MODE_WIN_ACCUMULATE – synchronize MPI_ACCUMULATE's to the target process

Future "Test and Set" or other communication calls may be integrated by other synchronization modes.

A persistent active request created by MPI_WIN_SYNC_OPS_INIT is completed if

> all the corresponding RMA communications calls (see sync_mode) which were issued since the last completed synchronization request to/from that target process target_rank are completed.

If a MPI test or wait functions detects that the request is completed, the remote counter sync_counter, which is located in window win in process target_rank, is atomically decremented (see function MPI_WIN_SYNC_OBJECT_INIT below).

The counter sync_counter should not be allocated by function MPI_WIN_ALLOC_SYNC_OBJECTS and the data on the counter should be received by some communication call.
Note: There is the special case, that a MPI process synchronizes with itself. In this case, it's allowed to pass the MPI_Sync counter returned by MPI_WIN_ALLOC_SYNC_OBJECTS to this function.

MPI_WIN_SYNC_OBJECT_INIT(sync_counter, counter, win, info, req)

INOUT	sync_counter	sync object (handle)
IN	count	count (integer)
IN	win	window object (handle)
IN	info	info argument (handle)
OUT	req	request (handle)

```
int MPI_Win_sync_object_init(MPI_Sync sync_counter,
            int count, MPI_Win win, MPI_Info info,
            MPI_Request *req)
```

```
MPI_WIN_SYNC_OBJECT_INIT (SYNC_COUNTER, COUNT, WIN,
            INFO, REQ, IERROR)
   INTEGER SYNC_COUNTER, COUNT, WIN, INFO, REQ, IERROR
```

```
MPI_Prequest MPI::WIN::Sync_object_init(
            MPI_Sync sync_counter, int count,
            MPI_Info info) const
```

A call to MPI_WIN_SYNC_OBJECT_INIT sets the value of the local synchronization counter sync_counter to count (*future MPI_Count type ?*). and creates and initializes a persistent request handle for this counter. The counter sync_counter must be allocated by function MPI_WIN_ALLOC_SYNC_OBJECTS.

This synchronization counter sync_counter is atomically decremented when persistent requests issued by function MPI_WIN_SYNC_OPS_INIT are completed in a MPI test or wait routine. If the counter sync_counter was decremented to 0, the request issued by MPI_WIN_SYNC_OBJECT_INIT is completed. A following function call of MPI_Start or MPI_Startall resets the value of sync_counter is reset to the initial value count.

The memory locations read in by the corresponding MPI_GET requests can be accessed and the memory locations transferred by the corresponding MPI_PUT requests can be changed after the request is completed.

Advice to implementers: A negative value of sync_counter should result in an error code of error class MPI_ERR_WIN_COUNTER.

The contents of the status object after completion of such requests is undefined.

1.2.4 Examples

Example A: The following examples show MPI processes which perform some relaxation and which perform the overlap exchange by putting the data into the halo region of neighbouring processes. This is a generic iterative code which corresponds to Chapter 6.5 in the MPI-2 document. The synchronization counter “my_halos_updated” counts the number of neighbour processes which have updated the overlap (halo) regions of the actual process. After the persistent request “my_halos_updated_req” is completed the overlap regions can be used for the iterative application. The synchronization objects ”to_halos_updated” are used to signal the neighboured processes that the overlap regions were updated and can be accessed.
The window of each process consists of array A, which contains the origin and target buffers of the put calls. The ranks of the neighbour processes are contained in “toneighbor[]”.

In the first example, the synchronization counters are only used to signal the update of the overlap regions by MPI_PUT. In order to synchronize, that the halo regions are free for update a simple barrier is used.

```
#define NUMBER_OF_NEIGHBOURS 4
   int i;

   MPI_Sync my_counters[1]; /* local counters */
   MPI_Sync  my_halos_updated;
   MPI_Sync to_halos_updated[NUMBER_OF_NEIGHBOURS]; /* remote counters */

   MPI_Request send_req[NUMBER_OF_NEIGHBOURS], recv_req[NUMBER_OF_NEIGHBOURS];
   MPI_Request put_req [NUMBER_OF_NEIGHBOURS];
   MPI_Request my_halos_updated_req;
```

```
/* Allocate local sync objects:
   my_halos_updated = Local counter for my updated halo regions
 */

MPI_Win_alloc_sync_objects (1, my_counters, win, MPI_INFO_NULL);

my_halos_updated = my_counters[0];

/* Exchange synchronization counters with neighbours:
   to_halos_updated = Remote counters for updated halo regions of neighbours
 */

for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
   MPI_Irecv (&to_halos_updated[i], 1, MPI_HANDLE_SYNC, toneighbor[i], 1,
              MPI_COMM_WORLD, &recv_req[i]);

   MPI_Isend (&my_halos_updated,    1, MPI_HANDLE_SYNC, toneighbor[i], 1,
              MPI_COMM_WORLD, &send_req[i]);
}

MPI_Waitall (NUMBER_OF_NEIGHBOURS, recv_req, MPI_STATUSES_IGNORE);
MPI_Waitall (NUMBER_OF_NEIGHBOURS, send_req, MPI_STATUSES_IGNORE);

/* Initialize persistent requests and synchronization counters
   my_halos_updated_req: Wait for NUMBER_OF_NEIGHBOURS syncs
   put_req[i]          : Wait for completion of MPI_Put's
                         to process "toneighbor[i]"
 */

MPI_Win_sync_object_init (my_halos_updated, NUMBER_OF_NEIGHBOURS,
                          win, MPI_INFO_NULL, &my_halos_updated_req);

for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
   MPI_Win_sync_ops_init (toneighbor[i], MPI_MODE_WIN_PUT, to_halos_updated[i],
                          win, MPI_INFO_NULL, &put_req[i]);
}

...

while (! converged(A)) {
   MPI_Startall (NUMBER_OF_NEIGHBOURS, put_req);
   MPI_Start (&my_halos_updated_req);

   update (A);

   /* Signal that my halo regions are free to be updated by a
      simple barrier.
    */

  MPI_Barrier (MPI_COMM_WORLD);

   /* Update halo regions */

   for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
      MPI_Put (&frombuf[i], 1, fromtype[i], toneighbor[i],
               todisp[i], 1, totype[i], win);
```

```
         ...
      }

      MPI_Waitall (NUMBER_OF_NEIGHBOURS, put_req, MPI_STATUSES_IGNORE);

      /* Wait that my halo regions are updated */

      MPI_Wait (&my_halos_updated_req, MPI_STATUS_IGNORE);
   }
   MPI_Win_free_sync_objects (1, my_counters, win);
```

In the second example, the synchronization counters are only used to signal the update of the halo (overlap) regions by MPI_PUT. In order to signal that the halo regions are free for update, the MPI process sends a message to the neighbour process that the halo regions are free for update a simple barrier is used.

```
#define NUMBER_OF_NEIGHBOURS 4
   int i, index;
   int dummy_buf[1];

   MPI_Sync my_counters[1]; /* local counters */
   MPI_Sync  my_halos_updated;
   MPI_Sync to_halos_updated[NUMBER_OF_NEIGHBOURS]; /* remote counters */

   MPI_Request send_req[NUMBER_OF_NEIGHBOURS], recv_req[NUMBER_OF_NEIGHBOURS];
   MPI_Request put_req [NUMBER_OF_NEIGHBOURS];
   MPI_Request my_halos_updated_req;

   /* Allocate local sync objects:
      my_halos_updated = Local counter for my updated halo regions
    */

   MPI_Win_alloc_sync_objects (1, my_counters, win, MPI_INFO_NULL);

   my_halos_updated = my_counters[0];

   /* Exchange synchronization counters with neighbours:
      to_halos_updated = Remote counters for updated halo regions of neighbours
    */

   for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
      MPI_Irecv (&to_halos_updated[i], 1, MPI_HANDLE_SYNC, toneighbor[i], 1,
                 MPI_COMM_WORLD, &recv_req[i]);

      MPI_Isend (&my_halos_updated,    1, MPI_HANDLE_SYNC, toneighbor[i], 1,
                 MPI_COMM_WORLD, &send_req[i]);
   }

   MPI_Waitall (NUMBER_OF_NEIGHBOURS, recv_req, MPI_STATUSES_IGNORE);
   MPI_Waitall (NUMBER_OF_NEIGHBOURS, send_req, MPI_STATUSES_IGNORE);

   /* Initialize persistent requests and synchronization counters
      my_halos_updated_req: Wait for NUMBER_OF_NEIGHBOURS syncs
      put_req[i]          : Wait for completion of MPI_Put’s
```

```
                         to process "toneighbor[i]"
 */

MPI_Win_sync_object_init (my_halos_updated, NUMBER_OF_NEIGHBOURS,
                          win, MPI_INFO_NULL, &my_halos_updated_req);

for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
   MPI_Win_sync_ops_init (toneighbor[i], MPI_MODE_WIN_PUT, to_halos_updated[i],
                          win, MPI_INFO_NULL, &put_req[i]);
}

...

while (! converged(A)) {
   MPI_Startall (NUMBER_OF_NEIGHBOURS, put_req);
   MPI_Start (&my_halos_updated_req);

   update (A);

   /* Signal that my halo regions are free to be updated by
      a dummy message with tag SYNCTAG.  */

   for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
      MPI_Irecv (dummy_buf, 0, MPI_INT, toneighbor[i], SYNCTAG,
                 MPI_COMM_WORLD, &recv_req[i]);

      MPI_Isend (dummy_buf, 0, MPI_INT, toneighbor[i], SYNCTAG,
                 MPI_COMM_WORLD, &send_req[i]);
   }

   /* Wait that halo regions of neighbours become free to be updated by
      actual process and update halo regions */

   for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
      MPI_Waitany (NUMBER_OF_NEIGHBOURS, recv_req, &index, MPI_STATUS_IGNORE);

      MPI_Put (&frombuf[index], 1, fromtype[index], toneighbor[index],
               todisp[index], 1, totype[index], win);
      ...
   }

   MPI_Waitall (NUMBER_OF_NEIGHBOURS, put_req,  MPI_STATUSES_IGNORE);
   MPI_Waitall (NUMBER_OF_NEIGHBOURS, send_req, MPI_STATUSES_IGNORE);

   /* Wait that my halo regions are updated */

   MPI_Wait (&my_halos_updated_req, MPI_STATUS_IGNORE);
}
MPI_Win_free_sync_objects (1, my_counters, win);
```

In the third example, the synchronization counters are used to signal the update of the halo (overlap) regions by MPI_PUT and to signal that the halo regions are free for update.

```
#define NUMBER_OF_NEIGHBOURS 4
```

```
int i;

MPI_Sync my_counters[2];
MPI_Sync my_halos_updated, dest_halos_free;
MPI_Sync to_halos_free[NUMBER_OF_NEIGHBOURS];
MPI_Sync to_halos_free[NUMBER_OF_NEIGHBOURS];

MPI_Request send_req[2*NUMBER_OF_NEIGHBOURS], recv_req[2*NUMBER_OF_NEIGHBOURS];
MPI_Request put_req [NUMBER_OF_NEIGHBOURS], free_req[NUMBER_OF_NEIGHBOURS];
MPI_Request my_halos_updated_req, dest_halos_free_req;

/* Allocate local sync objects:
   my_halos_updated = Counter for my updated halo regions
   dest_halos_free  = Counter for free destination halo regions
 */

MPI_Win_alloc_sync_objects (2, my_counters, win, MPI_INFO_NULL);

my_halos_updated = my_counters[0];
dest_halos_free  = my_counters[1];

/* Exchange synchronization counters with neighbours:
   to_halos_free    = Counters for free    halo regions of neighbours
   to_halos_updated = Counters for updated halo regions of neighbours
*/

for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
   MPI_Irecv (&to_halos_updated[i], 1, MPI_HANDLE_SYNC, toneighbor[i], 1,
              MPI_COMM_WORLD, &recv_req[i]);

   MPI_Isend (&my_halos_updated,    1, MPI_HANDLE_SYNC, toneighbor[i], 1,
              MPI_COMM_WORLD, &send_req[i]);

   MPI_Irecv (&to_halos_free[i], 1, MPI_HANDLE_SYNC, toneighbor[i], 1,
              MPI_COMM_WORLD, &recv_req[NUMBER_OF_NEIGHBOURS+i]);

   MPI_Isend (&dest_halos_free,  1, MPI_HANDLE_SYNC, toneighbor[i], 1,
              MPI_COMM_WORLD, &send_req[NUMBER_OF_NEIGHBOURS+i]);
}

MPI_Waitall (2*NUMBER_OF_NEIGHBOURS, recv_req, MPI_STATUSES_IGNORE);
MPI_Waitall (2*NUMBER_OF_NEIGHBOURS, send_req, MPI_STATUSES_IGNORE);

/* Initialize persistent requests of (local) synchronization counters
   my_halos_updated_req: Wait for NUMBER_OF_NEIGHBOURS syncs
   dest_halos_free_req : Wait for NUMBER_OF_NEIGHBOURS syncs
 */

MPI_Win_sync_object_init (my_halos_updated, NUMBER_OF_NEIGHBOURS, win,
                          MPI_INFO_NULL, &my_halos_updated_req);

MPI_Win_sync_object_init (dest_halos_free, NUMBER_OF_NEIGHBOURS, win,
                          MPI_INFO_NULL, &dest_halos_free_req);

/* Initialize persistent requests of (remote) RMA operations
   free_req[i] : No wait, no RMA call (synchronization mode 0);
                 simply notify process "toneighbor[i]"
   put_req[i]  : Wait for completion of MPI_Put's to process "toneighbor[i]"
*/

for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
   MPI_Win_sync_ops_init (toneighbor[i], MPI_MODE_WIN_PUT, to_halos_updated[i],
                          win, MPI_INFO_NULL, &put_req[i]);
   MPI_Win_sync_ops_init (to_neighbor[i], 0, to_halos_free[i],
                          win, MPI_INFO_NULL, &free_req[i]);
}

...

while (! converged(A)) {
   MPI_Startall (NUMBER_OF_NEIGHBOURS, put_req);
   MPI_Start (&my_halos_updated_req);
   MPI_Start (dest_halos_free_req);

   update (A);

   /* Signal that my halo regions are free to be updated. */

   MPI_Startall (NUMBER_OF_NEIGHBOURS, free_req);

   /* Wait that halo regions of neighbours become free to be updated by
      actual process and update halo regions */

   MPI_Wait (dest_halos_free_req);

   for (i = 0; i < NUMBER_OF_NEIGHBOURS; i++) {
      MPI_Put (&frombuf[i], 1, fromtype[i], toneighbor[i],
               todisp[i], 1, totype[i], win);
   }

   MPI_Waitall (NUMBER_OF_NEIGHBOURS, put_req, MPI_STATUSES_IGNORE);

   /* Wait that my halo regions are updated */

   MPI_Wait (&my_halos_updated_req, MPI_STATUS_IGNORE);
}
MPI_Win_free_sync_objects (2, my_counters, win);
```

1.2.5 Possible Info Key

A possibly predefined info key for the functions MPI_WIN_SYNC_OBJECT_INIT and MPI_WIN_SYNC_OPS_INIT could be 'restart' which would automatically restart the persistent request in MPI wait and test functions without the necessity that the application "restarts" the persistent request with MPI_START or MPI_STARTALL again.

This would fit the designed usage of these synchronization counters that they are re-used frequently. In addition, it might avoid some possible race conditions in application programs and it would increase the performance. In this case, an MPI implementation which doesn't support this restart mode must return an error code. Therefore, it would be better to include such a parameter into the argument list.

1.2.6 Extending the Window

The application should be able to change the processes which use the window. This is for example necessary when MPI processes are dynamically spawned and want to participate on the RMA operations on an already existing window.

MPI_WIN_CHANGE_COMM(new_comm, peer_rank, win)

IN	new_comm	new intracommunicator of the window (handle)
IN	peer_rank	rank in the new communicator new_comm which is also member of the actual communicator of win m(nonnegative integer)
INOUT	win	window object (handle)

```
int MPI_Win_change_comm(MPI_Communicator new_comm,
            int peer_rank, MPI_Win win)
```

```
MPI_WIN_CHANGE_COMM (NEW_COMM, PEER_RANK, WIN, IERROR)
   INTEGER NEW_COMM, PEER_RANK, WIN, IERROR
```

```
void MPI::WIN::Change_comm(MPI_Communicator new_comm,
            int peer_rank) const
```

Function MPI_Win_change_comm is collective on communicator new_comm and the communicator of win. The new communicator of win will be new_comm.

1.2.7 Misc

Constants MPI_SYNC_NULL, MPI_MODE_WIN_PUT, MPI_MODE_WIN_GET, MPI_MODE_WIN_ACCUMULATE

Conversion functions MPI_SYNC_COUNTER_F2C and MPI_SYNC_COUNTER_C2F.

Error class MPI_ERR_WIN_COUNTER (or put it in MPI_ERR_RMA_SYYNC).