MPI Semantic Terms

Quo Vadis 4.1

Ideas by Rolf Rabenseifner

Based on many discussions within the Semantic Terms Working Group

What is still missing?

- Example for MPI_RSEND
- Example for MPI_MRECV
- May be text about progress (instead of referencing to the rationale about MPI_WIN_COMPLETE in Fig. 11.8 in Section 11.7.3)

Clarification of RSEND

- E.g. to be added as a paragraph on „Localness“
  at the end of Section 3.5 Semantics of Point-to-Point Communication

In a correct MPI program, a call to MPI_(I)RSEND requires that the receiver has already started the corresponding receive. Under this assumption, the call to MPI_RSEND and the call to MPI_WAIT corresponding to an MPI_IRSEND are local.

```
process 0              process 1
                       MPI_IRECV(tag1,rq)
                       MPI_SEND(tag2)
MPI_RECV(tag2)
MPI_RSEND(tag1)
MPI_SEND(tag3)         MPI_RECV(tag3)
                       MPI_WAIT(rq)
```

Figure X.2: No deadlock with MPI_RSEND

Consider the code illustrated in Figure X.2. This code will not deadlock.
Note that the message with tag2 only illustrates any method to guarantee that the receive is already started in process 1 before the ready send is started in process 0.
Once the MPI_IRECV(tag1) and the MPI_RSEND(tag1) is started, then process 0 can proceed to completion.
The rationale about MPI_WIN_COMPLETE in Fig. 11.8 in Section 11.7.3 does also apply to MPI_RSEND.

Clarification of MRECV

- E.g. to be added after the definition of MPI_MRECV in Section 3.8.2 Matching Probe

Consider the code illustrated in Figure X.1.

```
process 0               process 1
                        MPI_ISEND(tag1,rq)
MPI_MPROBE(tag1,msg1)
MPI_MRECV(msg1)
MPI_SEND(tag2)          MPI_RECV(tag2)
                        MPI_WAIT(rq)
```

Figure X.1: No deadlock with MPI_MRECV

This code will not deadlock.
Once the MPI_ISEND(tag1) is started, then MPI_MPROBE(tag1) must return and process 0 can proceed to completion.
The rationale about MPI_WIN_COMPLETE in Fig. 11.8 in Section 11.7.3 does also apply to MPI_MRECV and the completion of an MPI_WAIT corresponding to a call to MPI_IMRECV.

May be that we add a text about weak progress into Chapter 2.

What is still missing?

- RMA (Current status Feb 2 / May 20, 2020):
  - Text in Semantic Terms, advice to users:

    Nonblocking procedures:

    - incomplete and local: MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_PUT, MPI_GET, MPI_ACCUMULATE, MPI_IMPROBE, MPI_SEND_INIT, MPI_RECV_INIT, ...
  - Text in Annex A.2

Procedure	Stages	Cpl	Loc	Blk	Op	Collective			Blocked resources
						C	sq	S/W	and remarks
MPI_PUT, MPI_GET, MPI_ACCUMULATE	i-s----	ic	l	**nb**	nb-op	-			buffer **14) 21)**
Other one-sided procedures									21)

14) Nonblocking procedure without an **I** prefix.

21) In some cases, more than one MPI procedure may be needed to implement one stage of an MPI one-sided operation. For details on the semantics of one-sided operations, see Chapter 11.

- Partitioned Communication
  - Currently no investigation / the proposer of the partitioned communication may check whether their text still fits to the modified Semantic Terms, especially the use of “*nonblocking*”

Problems with RMA

- Do we have any problem with the change
  - from MPI-1 – 3: nonblocking := incomplete
  - to MPI-4: nonblocking := incomplete AND local
- Is there any wrong usage of nonblocking as
  - operation vs. procedure is nonblocking
  - procedure does not block until …
  - procedure is local

We first address these questions

These questions should be also checked by the chapter committee of "Partitioned communication"

- Which table entries would be correct for the RMA synchronization procedures?
- Are the RMA communication procedures really nonblocking?

RMA small wording corrections Part II.a

MPI-3.1 Section 11.3 Communication Calls, page 417, lines 10-23 should read

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET transfer data from the target memory to the caller memory; MPI_ACCUMULATE and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these Locations values sent from the caller memory; MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote atomic compare and swap operation. These operations are nonblocking. Additionally, these particular procedures are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent synchronization call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.5.

MPI-3.1 Section 11.5.2, page 442, lines 38-42 about Example 11.4 should read

One can also have implementations where the call to MPI_WIN_START ~~is nonblocking~~ may not block, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations where the first two calls ~~are nonblocking~~ may not block, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred;

(similar to the unchanged text in MPI-3.1 Section 11.5.3, page 448, lines 22-25 about Example 11.5:

The call to MPI_WIN_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired – the update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs.)

RMA small wording corrections Part II.a, continued

MPI-3.1 Section 11.5.2, page 443, lines 32-33 on MPI_WIN_TEST should read

> This is the ~~nonblocking~~ local version of MPI_WIN_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding.

MPI-3.1 Section 11.7.3 Progress, page 462, lines 37-38 about Example 11.6 should read

> The post calls ~~are nonblocking~~ do not block, and should complete.

MPI-4 Issue #96 Annex A.2, amendment to Footnote 21 on RMA procedures read

> 21) In some cases, more than one MPI procedure may be needed to implement one stage of an MPI one-sided operation.
> The locality of the RMA procedures is analyzed from the user's perspective, i.e., independent of the choices to implementors about weak synchronization as described in Sections 11.5, 11.5.2 (on Example 11.4), and 11.5.3 (on Example 11.5).
> For details on the semantics of one-sided operations, see Chapter 11.

- These changes are under the commits

  In chap-applang/

  – v10-->v11: Semantic terms / RMA small wording corrections in A.2 Footnote 21 (Part II.a)

  In chap-one-sided/ (v00= version 2020-02-02 of one-sided-2.tex)

  – v00-->v12: Semantic terms / RMA small wording corrections (Part II.a)

RMA small wording corrections Part II.b

MPI-3.1 Section 11.5.2, page 442, lines 27-46 about Example 11.4 should read

Consider the sequence of calls in the example below.

Example 11.4

```
MPI_Win_start(group, flag, win);
MPI_Put(..., win);
MPI_Win_complete(win);
```

The call to MPI_WIN_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process.

> *Advice to implementors.* This still leaves much choice to implementors. The call to MPI_WIN_START can block until the matching call to MPI_WIN_POST occurs at all target processes. One can also have implementations where the call to MPI_WIN_START may not block, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations where the first two calls may not block, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred; or even implementations where all three calls can complete before any target process has called MPI_WIN_POST – the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence above must complete, without further dependencies. *(End of advice to implementors.)*
>
> *Advice to users.* In order to ensure a portable deadlock free program, a user must assume that MPI_WIN_START may block until the corresponding MPI_WIN_POST has been called. *(End of advice to users.)*

This existing part from MPI-3.1 now defined as advice to implementors.

New advice to users to make clear that MPI_WIN_START has to be assumed as non-local for portable application programming.

RMA small wording corrections Part II.b, continued

MPI-3.1 Section 11.5.3 *Lock*, page 448, lines 14-28 about Example 11.5 should read

Consider the sequence of calls in the example below.

Example 11.5

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);
MPI_Put(..., rank, ..., win);
MPI_Win_unlock(rank, win);
```

The call to MPI_WIN_UNLOCK will not return until the put transfer has completed at the origin and at the target.

> *Advice to implementors.* This still leaves much freedom to implementors. The call to MPI_WIN_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired – the update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs. However, if the call to MPI_WIN_LOCK is used to lock a local window, then the call must block until the lock is acquired, since the lock may protect local load/store accesses to the window issued after the lock call returns. *(End of advice to implementors.)*

> *Advice to users.* In order to ensure a portable deadlock free program, a user must assume that MPI_Win_lock may block until an exclusive lock on the window is acquired. *(End of advice to users.)*

- These changes are under the commits

  In chap-one-sided/

  – v12-->v13: Semantic terms / RMA small wording corrections (Part II.b)

Problems with RMA

- Do we have any problem with the change
  - from MPI-1 – 3: nonblocking := incomplete
  - to MPI-4: nonblocking := incomplete AND local
- Is there any wrong usage of nonblocking as
  - operation vs. procedure is nonblocking
  - procedure does not block until …
  - procedure is local

Second question

- Which table entries would be correct for the RMA synchronization procedures?
- Are the RMA communication procedures really nonblocking?

Additional changes to Issue 96 to more completely handle RMA in Annex A.2 (part 3)

MPI-4 Issue #96 Annex A.2 2nd table on RMA procedures reads

Procedure	**Stages**	**Cpl**	**Loc**	**Blk**	**Op**	**Collective**			**Blocked resources**
						C	**sq**	**S/W**	**and remarks**
MPI_PUT, MPI_GET, MPI_ACCUMULATE	i-s----	ic	l	**nb**	nb-op	-			buffer **14)** **21)**
Other one-sided procedures									21)

but should read

Procedure	**Stages**	**Cpl**	**Loc**	**Blk**	**Op**	**Collective**			**Blocked resources**
						C	**sq**	**S/W**	**and remarks**
On an origin process with Fence Synchronization									
MPI_WIN_FENCE	i------		nl			C	sq	W1	
MPI_PUT, MPI_GET, MPI_ACCUMULATE, ...	i-s----	ic	l	**nb**	nb-op	-			buffer **14)** 21)
MPI_WIN_FENCE	----c-f	c+f	l			C	sq	W1	
On a target process									
MPI_WIN_CREATE/_ALLOCATE/..._SHARED	i------		nl			C	sq	W1	buffer address
MPI_WIN_FENCE	--s----	ic	l			C	sq	W1	buffer address+content
MPI_WIN_FENCE	----c--	c	nl			C	sq	W1	buffer address
MPI_WIN_FREE	------f	f	nl			C	sq	W1	

Continuation on next slide

Additional changes to Issue 96 to more completely handle RMA in Annex A.2 (part 3, continued)

and

Procedure	**Stages**	**Cpl**	**Loc**	**Blk**	**Op**	**Collective C**	**sq**	**S/W**	**Blocked resources and remarks**
On an origin process with General Active Target Synchronization									
MPI_WIN_START	i------		nl			-			21)
MPI_PUT, MPI_GET, MPI_ACCUMULATE, …	i-s----	ic	l	**nb**	nb-op	-			buffer **14)** 21)
MPI_WIN_COMPLETE	----c-f	c+f	l			-			21)
On a target process									
MPI_WIN_CREATE/_ALLOCATE/…_SHARED	i------		nl			C	sq	W1	buffer address
MPI_WIN_POST	--s----	ic	l			-			buffer address+content
MPI_WIN_WAIT	----c--	c	nl			-			buffer address
MPI_WIN_FREE	------f	f	nl			C	sq	W1	
On an origin process with Lock/Unlock Synchronization									
MPI_WIN_LOCK and others	i------		nl			-			21)
MPI_PUT, MPI_GET, MPI_ACCUMULATE, …	i-s----	ic	l	**nb**	nb-op	-			buffer **14)** 21)
MPI_WIN_UNLOCK and others	----c-f	c+f	l			-			21)
On a target process									
MPI_WIN_CREATE/_ALLOCATE/…_SHARED	i------		nl			C	sq	W1	buffer address
MPI_WIN_FREE	------f	f	nl			C	sq	W1	

Continuation on next slide

Additional changes to Issue 96 to more completely handle RMA in Annex A.2 (part 3, continued)

and

Procedure	Stages	Cpl	Loc	Blk	Op	Collective			Blocked resources
						C	sq	S/W	and remarks
On an origin process with Lock/Unlock Synchronization									
MPI_WIN_LOCK and others	i------		nl			-			21)
MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, …	i-s----	ic	l	**nb**	nb-op	-			buffer **14)** 21)
MPI_WAIT	----c-f	c+f	l			-			21)
MPI_WIN_UNLOCK and others	-------		l			-			21)
On a target process									
MPI_WIN_CREATE/_ALLOCATE/..._SHARED	i------		nl			C	sq	W1	buffer address
MPI_WIN_FREE	------f	f	nl			C	sq	W1	

Lets look at pdf on Issue #96 and the small changes on PR 116

- pdf: https://github.com/mpi-forum/mpi-issues/files/4708015/mpi32-report-semantic-terms-2020-05-31-annotated.pdf
- Issue: https://github.com/mpi-forum/mpi-issues/issues/96
  - Here, you can find also these slides:
  - EuroMPI2019-SematicTerms-fromPuri-2020-04-28+RMA-rab-2020-05-31.pdf
- Pull request: https://github.com/mpi-forum/mpi-standard/pull/116
- Any open questions?

Problems with RMA

- Do we have any problem with the change
  - from MPI-1 – 3: nonblocking := incomplete
  - to MPI-4: nonblocking := incomplete AND local
- Is there any wrong usage of nonblocking as
  - operation vs. procedure is nonblocking
  - procedure does not block until …
  - procedure is local
- Which table entries would be correct for the RMA synchronization procedures?

Third question

- Are the RMA communication procedures really nonblocking?

Recap of RMA in MPI-3.1

11.3 Communication Calls (page 417, lines 10-23)

MPI supports the following RMA communication calls: **MPI_PUT** and MPI_RPUT transfer data from the caller memory (origin) to the target memory; **MPI_GET** and MPI_RGET transfer data from the target memory to the caller memory; **MPI_ACCUMULATE** and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these Locations values sent from the caller memory; MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic ...-writ... and return the data before the accumulate operation; and MPI_COMPA..._AND_SWAP performs a remote atomic compare and swap operation. **These operations ... nonblocking.** **Additionally, these particular procedures are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns.** The transfer is completed, at the origin or both the origin and the target, when a subsequent synchronization call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.5.

That is what we expected:
MPI_PUT, MPI_GET, MPI_ACCULMULATE, ..., they are *nonblocking* procedures according to MPI-3.1 definition of nonblocking.
Are they also *local*?

Recap of RMA in MPI-3.1

11.7.3 Progress (page 462, lines 16-20)

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. **RMA calls must have <u>local</u> semantics, except when required for synchronization with other RMA calls.**

Yes, they are also *local*!

What does this exception mean?
Can only RMA synchronization calls be *non-local*?

Recap of RMA in MPI-3.1

11.5.2 General Active Target Synchronization (page 441, line 28-32, 47 - page 442, line 4)

MPI_WIN_START(group, assert, win)

Starts an RMA access epoch for win. RMA calls issued on win during this epoch must access only windows at processes in group. Each process in group must issue a matching call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary, until the target process executed the matching call to MPI_WIN_POST. **MPI_WIN_START is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not required to.**

Okay, MPI_WIN_START is *non-local*!

Recap of RMA in MPI-3.1

11.5.2 General Active Target Synchronization (page 442, lines 10, 21-26)

MPI_WIN_COMPLETE(win)

Completes an RMA access epoch on win started by a call to MPI_WIN_START. All RMA communication calls issued on win during this epoch will have completed at the origin when the call returns.

MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

MPI_WIN_COMPLETE has no defined right to block.
Expectation: It is *local* !?!?

Recap of RMA in MPI-3.1

11.5.2 General Active Target Synchronization (page 442, lines 27-46)

Consider the sequence of calls in the example below.

Example 11.4

```
MPI_Win_start(group, flag, win);
MPI_Put(..., win);
MPI_Win_complete(win);
```

The call to MPI_WIN_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process.

Advice to implementors.
This still leaves much choice to implementors. The call to MPI_WIN_START can block until the matching call to MPI_WIN_POST occurs at all target processes. One can also have implementations where the call to MPI_WIN_START may not block, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations where the first two calls may not block, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred; or even implementations where all three calls can complete before any target process has called MPI_WIN_POST – the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence above must complete, without further dependencies. *(End of advice to implementors.)*

Advice to users. In order to ensure a portable deadlock free program, a user must assume that MPI_WIN_START may block until the corresponding MPI_WIN_POST has been called. *(End of advice to users.)*

Wow!!!!
"MPI_PUT is allowed to block until …"
Is MPI_PUT still a local procedure???

From the users view, these sentences are not relevant:
The user has already to expect that MPI_WIN_START has blocked until …

This means:
Only the BLUE sentence is relevant for the user's perspective for writing deadlock-free MPI applications …

… as also mentioned in the new advice to users.

Recap of RMA in MPI-3.1

Often overseen in many discussions!

1.8 Who Should Use This Standard? (page 5, lines 1-9)

This standard is intended for use by all those who want to write portable message-passing programs in Fortran and C (and access the C bindings from C++). This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

Let's formalize our definition of non-local From the "application programmers'" viewpoint

How to resolve the problem that MPI_Put should be local and is allowed to block

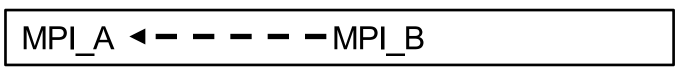
Semantic Terms, **definition of non-local**:

> **Non-local procedure** An MPI procedure is non-local if returning may require the execution of some specific semantically-related MPI procedure on another MPI process during the execution of the MPI procedure.

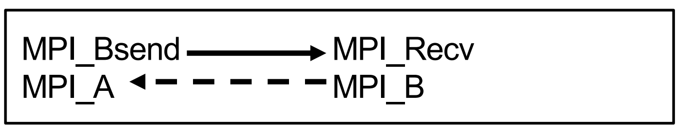
How to decide whether an MPI procedure is non-local, especially, if an MPI implementation is allowed to postpone a "may block until …" to a later MPI procedure call?

In this case, we may formalize our definition of non-local:

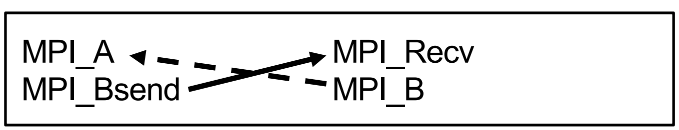
An MPI procedure MPI_A is non-local, if there exists a correct (i.e. also deadlock-free) MPI program with a call to MPI_A on one process and if there exists a call to an MPI procedure MPI_B on another process



so that if we add a test message as in the following figure



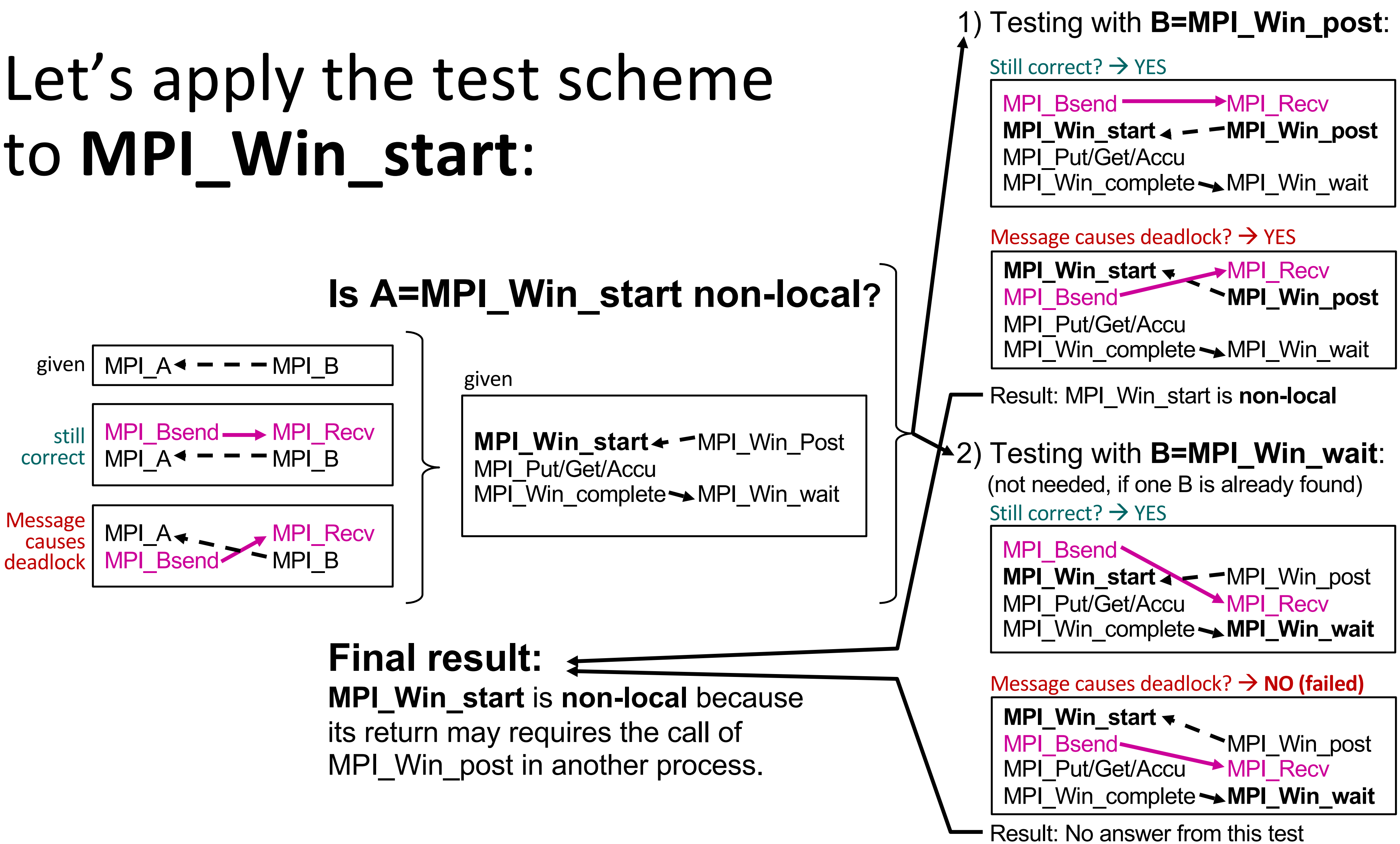
then the application is still correct (i.e. also deadlock-free), whereas the following modification may cause a deadlock:



If such a correct MPI program and such MPI_B exist, then MPI_B is such a "specific semantically-related MPI procedure on another MPI process" in the definition of ***non-local***.

If no such correct MPI program with any such MPI_B exist, then MPI_A is ***local***.

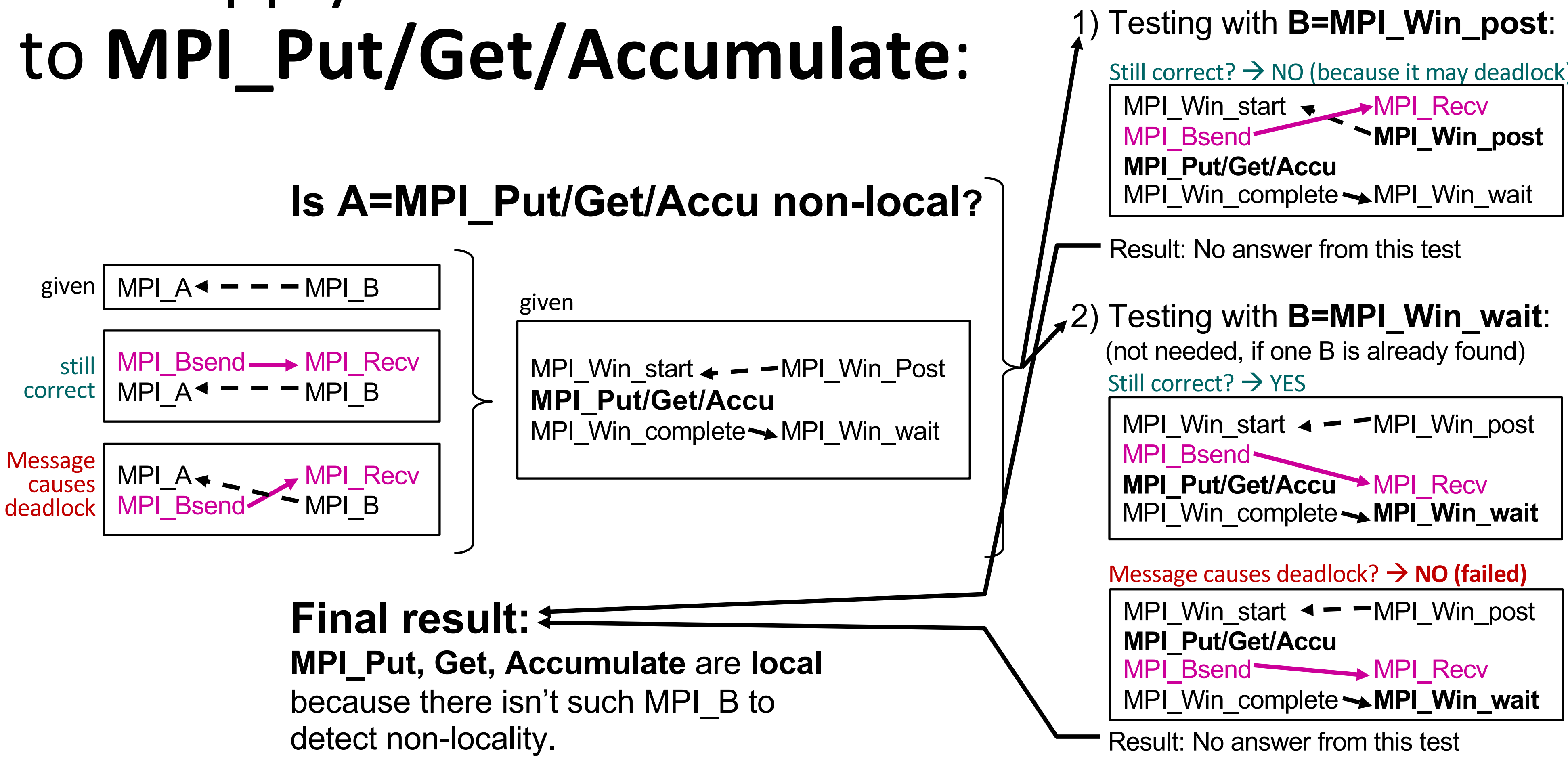
Let’s apply the test scheme to **MPI_Win_start**:



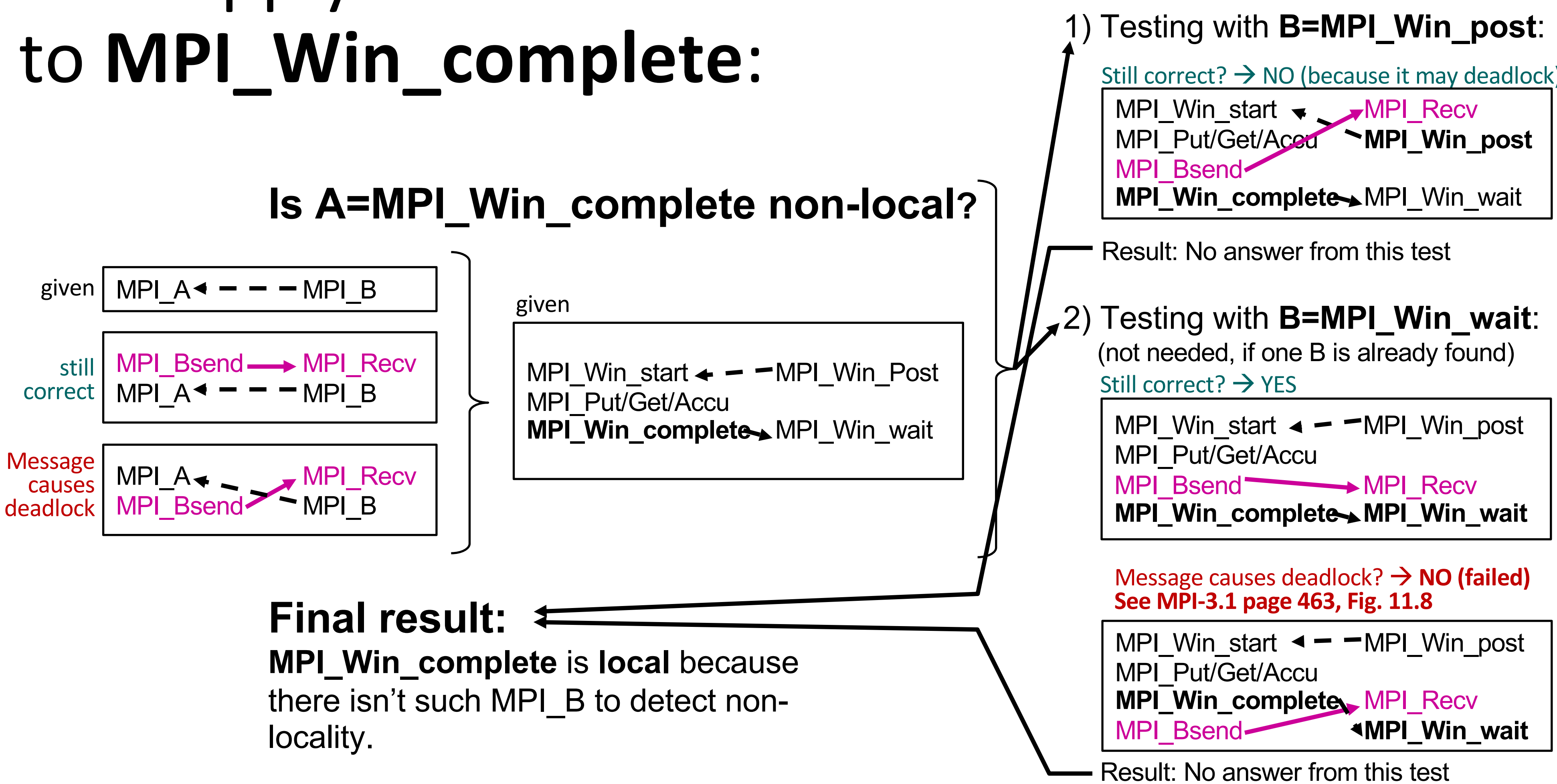
Final result:

MPI_Win_start is **non-local** because its return may requires the call of MPI_Win_post in another process.

Let's apply the test scheme to **MPI_Put/Get/Accumulate**:



Let’s apply the test scheme to **MPI_Win_complete**:



Recap of RMA in MPI-3.1

11.5.2 General Active Target Synchronization (page 443, lines 1, 17-19, 22, 33-38)

MPI_WIN_POST(group, assert, win)

Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process in group must issue a matching call to MPI_WIN_START. **MPI_WIN_POST does not block**.

It is *local*

MPI_WIN_WAIT(win)

Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. **The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred**. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

It is non-*local*

Results, independent from implementation choices:

MPI_Win_Start	non-local
MPI_Put	local (AND incomplete → nonblocking)
MPI_Get	local (AND incomplete → nonblocking)
MPI_Accumulate	local (AND incomplete → nonblocking)
MPI_Win_complete	local

From the MPI text and of course also with same testing one can see:

MPI_Win_post	local
MPI_Win_wait	non-local
MPI_Win_test	local
MPI_Win_fence	non-local
MPI_Win_lock	non-local
MPI_Win_unlock	local

As already shown in the proposal for the RMA tables in Annex A.2

Additional rationale to Issue 96 to more completely describe the “user’s perspective” in Annex A.2, Footnote 21 (part 4)

MPI-4 Issue #96 Annex A.2 Footnote 21 on RMA procedures read (including the change from Part 1)

> 21) In some cases, more than one MPI procedure may be needed to implement one stage of an MPI one-sided operation.
> The locality of the RMA procedures is analyzed from the user’s perspective, i.e., independent of the choices to implementors about weak synchronization as described in Sections 11.5, 11.5.2 (on Example 11.4), and 11.5.3 (on Example 11.5).
> For details on the semantics of one-sided operations, see Chapter 11.

but should read

> 21) In some cases, more than one MPI procedure may be needed to implement one stage of an MPI one-sided operation.
> The locality of the RMA procedures is analyzed from the user’s perspective, i.e., independent of the choices to implementors about weak synchronization as described in Sections 11.5, 11.5.2 (on Example 11.4), and 11.5.3 (on Example 11.5).
>
> Rationale.
>
> For this user’s perspective, the definition of non-local can be formalized as follows:
>
> (see next slide)
>
> (End of rationale.)
>
> For details on the semantics of one-sided operations, see Chapter 11.

Additional rationale to Issue 96 to more completely describe the "user's perspective" in Annex A.2, Footnote 21 (part 4, continued)

21) In some cases, more than one MPI procedure may be needed to implement one stage of an MPI one-sided operation.
The locality of the RMA procedures is analyzed from the user's perspective, i.e., independent of the choices to implementors about weak synchronization as described in Sections 11.5, 11.5.2 (on Example 11.4), and 11.5.3 (on Example 11.5). *(from PART II.)*

Rationale.

For this user's perspective, the definition of a *non-local procedure* (see Section 2.4.2) can be formalized as follows:
An MPI procedure MPI_A is non-local, if there exists a correct (i.e. also deadlock-free) MPI program with a call to MPI_A on one process and if there exists a call to an MPI procedure MPI_B on another process

so that if we add a test message as in the following figure

then the application is still correct (i.e. also deadlock-free),
whereas the following modification may cause a deadlock:

Note that MPI_B is then such a specific semantically-related MPI procedure on another MPI process as mentioned in the definition of the semantic term *non-local procedure* in Section 2.4.2.

(End of rationale.)

(PART IV.)

For details on the semantics of one-sided operations, see Chapter 11.

Status (May **31**, 2020)

- A set of additional changes:
  - No-no-votes requested for June 29-July 1, 2020 meeting (Munich → online)
  - Additional changes for the RMA chapter

    In chap-applang/
    - v15-->v16: Semantic terms / RMA: Rationale for A.2 Footnote 21 (1 change)

    In figures/
    - Modified MPI-semantics-appendix.pptx (now including also img5a-c)
    - New MPI-semantics-appendix_img5a.png / .eps / .pdf (for the rationale)
    - New MPI-semantics-appendix_img5b.png / .eps / .pdf
    - New MPI-semantics-appendix_img5c.png / .eps / .pdf

    } PART IV.
- For first vote: Version from May **31**, 2020

non-local An MPI procedure is non-local if returning may require the execution of some specific semantically-related MPI procedure on another MPI process.

Is our definition of „non-local" correct?

- Why should we go with

This "may" allows of course that the call on the other process is already done earlier.

non-local An MPI procedure is non-local if returning may require, **during its execution,** some specific semantically-related MPI procedure **to be called** on another MPI process.

```
process 0   process 1
            MPI_A
MPI_B
MPI_C
```

We do not want to use a stage, because for some procedures, it is the starting stage and for others (MI_Bcast_init) it is the initialization stage..

If return from MPI_B requires call of (i.e., the entry to) MPI_A and call of MPI_C requires return from MPI_B (and no additional call), is then MPI_C non-local?

Examples are

- Entry to MPI_(I)Send → MPI_Recv → MPI_Get_count
- Entry to MPI_(I)Send → MPI_Mprobe → MPI_Mrecv
- **Entry to MPI_Win_post → MPI_Win_start → MPI_Put**

Lets look at pdf on Issue #96 and the small changes on PR 116

- Final version from **May 31, 2020**
- pdf: https://github.com/mpi-forum/mpi-issues/files/4708015/mpi32-report-semantic-terms-2020-05-31-annotated.pdf
- Issue: https://github.com/mpi-forum/mpi-issues/issues/96
    - Here, you can find also these slides:
    - EuroMPI2019-SematicTerms-fromPuri-2020-04-28+RMA-rab-2020-05-31.pdf
- Pull request: https://github.com/mpi-forum/mpi-standard/pull/116
- Any open questions?

Thanks for listening

Questions?