(The blue text are only title lines for the review process. They are not planned being included into the MPI standard)

The goal of this text is to describe the reality of MPI-4.0 (and older) more understandable, accurately and precisely, i.e. not to change anything.

It should be placed in MPI-4.1 as a new Section 2.9 Progress (after error handling, before implementation issues).

2.9 Progress

Definition of a "progress chunk":

For a stage of an MPI operation (or parts of it) to be completed, it is often necessary for one or more other MPI processes to perform some activities.
These activities may occur during the starting or completing MPI procedures for the operation or they may be separated from the operation-related MPI procedures. We name an activity separated in this way a *progress chunk*. An example of a progress chunk is the transfer of message data from a buffered mode send operation that completed before the matching receive operation was started.

Kommentiert [RR1]: Wording: chunk is used, e.g., in OpenMP schedules and often, the default chunk size is 1. Chunk is rarely used in MPI-4.0 and none of these uses would be conflicting with this usage because it is always a chunk of xxx and xxx is different. Only here it is progress.

Definition of a "blocked MPI procedure call":

An MPI procedure is *blocked* if it delays its return until some specific activity or state-change has occurred in another MPI process.

A blocked MPI procedure call can be
- a non-local MPI procedure call that delays its return until a specific semantically-related MPI call on another MPI process, or
- a local MPI procedure call that delays its return until some unspecific MPI call in another MPI process causes a specific state-change in that MPI process, or
- an MPI finalization procedure (MPI_FINALIZE or MPI_SESSION_FINALIZE) that delays its return or exit because the MPI finalization must guarantee that all remaining progress chunks will be executed before the MPI finalization is finished.

Kommentiert [RR2]: Example would be sending out the buffered message of a Bsend, i.e., doing a progress chunk

Some examples of a non-local blocked MPI procedure call:
- MPI_SSEND delays its return until the matching receive operation is started at the destination MPI process (for example, by a call to MPI_RECV or to MPI_IRECV).
- MPI_RECV delays its return until the matching send operation is started at the source MPI process (for example, by a call to MPI_SEND or to MPI_ISEND).

Some examples of a local blocked MPI procedure call:
- MPI_RSEND, when the message data cannot be entirely buffered, delays its return until the destination MPI process has received the portion of message data that cannot be buffered, which may require one or more unspecific MPI procedure call(s) at the destination MPI process.
- MPI_RECV, in the use case when the message was buffered at the source MPI process (e.g. with MPI_BSEND), delays its return until the message is received, which may require one or more unspecific MPI procedure calls at the source MPI process to send the buffered data.

Definition of "progress":

All MPI processes are required to "guarantee progress", i.e., all progress chunks will eventually be executed. This guarantee is required to be provided by

- blocked MPI procedures, and
- repeatedly called MPI test procedures (see below) that return flag=false.

The progress must be provided independently of whether a progress chunk belongs to a specific session or to the world model (see Sections 11.2 and 11.3). Other ways of fulfilling this guarantee are possible and permitted (for example, a dedicated progress thread, or off-loading to a network interface controller (NIC)).

MPI test procedures are MPI_TEST, MPI_TESTANY, MPI_TESTALL, MPI_TESTSOME MPI_IPROBE, MPI_IMPROBE, MPI_REQUEST_GET_STATUS, MPI_WIN_TEST, and MPI_PARRIVED.

Definition of "strong progress":

Strong progress is provided by an MPI implementation if all local procedures return independently of MPI procedure calls in other MPI processes (operation-related or not).

Kommentiert [RR3]: Comment from RMA WG meeting Occt. 27, 2022: one may add an additional environmental inquiry (like MPI_TAG_UB or MPI_IO) in MPI_4.0 Section 9.1.2

Definition of "weak progress":

An MPI implementation provides *weak progress* if it does not provide strong progress.

Correctness of applications when using MPI implementations with different quality of progress.

Advice to users. The type of progress may influence the performance of MPI operations. A correct MPI application must be written under the assumption that only weak progress is provided. Every MPI application that is correct under weak progress will be correctly executed if strong progress is provided. The other way around, i.e. that correctness under the assumption of strong progress implies also correctness if only weak progress is provided, is not proven, but the MPI standard is designed such that it should be true. *(End of advice to users.)*

Rationale. The design of MPI restricts any use of synchronization methods that are not based on MPI communication procedures, which would likely result in deadlock without guaranteed strong progress in MPI, see for example Section 2.7 and Example 12.x in Section 12.7.3. *(End of rationale.)*

Further reading

For further rules, see Sect 2.4.2 the definition on local MPI procedures, Sect. 3.5 on point-to-point communication, and especially Sect. 3.7.4, paragraphs "Progress" on page 75, especially the paragraph on MPI_TEST, Sect. 3.8.1 and 3.8.2 on MPI_(I)(M)PROBE, Sect. 3.8.4 the advice to implementors for MPI_TEST_CANCELLED, Sect. 4.2.2 on MPI_PARRIVED, especially the paragraph on repeated calls to MPI_PARRIVED, Sect. 5.12 on collective procedure, Sect. 11.2.2 Example 11.6 on MPI_FINALIZE and especially the related advice to implementors, Sect. 11.3.1. on MPI_SESSION_FINALIZE, especially the paragraph on "MPI_SESSION_FINALIZE may be synchronizing" together with the related rationale, the first advice to implementors and Example 11.8, Sect. 11.6 on MPI and threads, Sect. 12.7.3 on progress with one-sided communication, especially the rationale at the end, and Sect. 14.6.3 on MPI parallel file I/O.

Limits of minimal progress / special rule for shared memory RMA (this restriction would be new in MPI-4.1, but it was always reality since the MPI shared memory was introduced in MPI-3.0):

(To be added in the RMA chapter, e.g. at the end of RMA Section 12.7.3 *Progress*)

The MPI standard does not support the use of MPI shared memory loads and/or stores for synchronizing purposes between MPI processes. If this rule is ignored then a deadlock may occur if an MPI implementation does not provide strong progress as shown in Example 12.x.

Example 12.x. Possible deadlock through the use of a shared memory variable for synchronization.
comm_sm shall be a shared memory communicator with at least two processes. win_sm is a shared memory window with the AckInRank0 as window portion in process rank 0.

```
Process with rank 0                        Process with rank 1

MPI_Win_shared_query(win,                  MPI_Win_shared_query(win,
   /*rank=*/ 0, ..., AckInRank0);             /*rank=*/ 0, ..., AckInRank0);

*myAck = 0;
MPI_Win_fence(win_sm)                      MPI_Win_fence(win_sm)
MPI_Buffer_attach(myHugeBuffer, ...);
MPI_Bsend(myHugeMessage, ...,
          /*rank=*/ 1, ..., comm_sm);
sleep(10); // to guarantee that the while-loop starts
           // after rank 1 is blocked in MPI_Recv
                                           sleep(5); // to ensure
                                             // that the MPI_Bsend
                                             // in rank 0 returned
                                           MPI_Recv(&myHugeMessage, ...
                                             /*rank=*/ 0, ..., comm_sm,
...);
                                           *AckInRank0 = 222;
while(*AckInRank0 != 222)
          /*empty polling loop*/;
MPI_Buffer_detach(&pTemp, &size);
// deadlock                                // deadlock
```

As long the MPI_Recv in process rank 1 is blocked until an unspecific MPI procedure call in process rank 0 happens to send the buffered data, then the subsequent statement cannot change the value of the shared window buffer AckInRank0. As long as this value is not changed, the while loop in process rank 0 will not return and therefore the next MPI procedure call (MPI_Buffer_detach) cannot happen, which is then a deadlock.
(End of example 12.x)

Kommentiert [RR4]: MPI_Recv and MPI_Buffer_attach: mixed notation, because it cites the call in language C in the verbatim above.

Note that both communication patterns (A) BSEND-RECV-DETACH and (B) the shared memory store/load for synchronization purpose, can be in different software layers and each layer would work properly, but the combination of (A) and (B) can cause the deadlock.

Kommentiert [RR5]: I put this note outside of the example because it is an important observation that should not be only part of the example.

History of changes
Rolf Rabenseifner, Oct. 4, 2022
Dan Holmes, Oct. 5, 2022
Rolf Rabenseifner, Oct. 18, 2022
Joseph Schuchart, Oct. 18, 2022
Dan Holmes, Oct. 18, 2022
Rolf Rabenseifner, Oct. 19, 2022
Dan Holmes, Oct. 21, 2022
Rolf Rabenseifner, Oct. 25, 2022
Dan Holmes, Oct. 25, 2022
Rolf Rabenseifner, Oct. 25, 2022
Rolf Rabenseifner, Oct.26, 2022
Claudia Blaas-Schenner, Oct. 27, 2022
Rolf Rabenseifner, Oct. 31, 2022