RMA Chapter Changes In MPI 4.1

MPI Forum Meeting, September 2023, Bristol, UK
On behalf of the RMA Chapter Committee

Change Overview: Minor Edits

#850: Review use of %ALLOWLATEX% and address issues
#714: Memory Allocation Kind Info
#749: Add info key for RMA accumulate behavior preference*
#740: Hints must be adhered to by the app
#846: Minor fixes for LaTeX warnings
#833: Update for getlatex and fixes for explicit font commands
#611: Fix semantic terms in RMA chapter
#708: Allow MPI_WIN_SHARED_QUERY on created and allocated windows*
#768: Wrap text about blocking MPI functions in Examples 12.4-12.5 in AtoI
#769: Make all RMA communication procedures local
#671: Example 12.4: RMA communication calls may not block in PSCW
#753: Progress - new section in terms
#722: Deprecate mpif.h
#664: Both MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_SHARED allocate memory
#637: Replace 'operation' with 'operator' where appropriate
#659: MPI_Win_free is synchronizing, except when it isn't*
#663: Add missing win-sync calls to Example 12.21
#747: Rephrase MPI_WIN_SYNC to make it clear what it's used for*
#729: MPI_WIN_TEST with same progress as MPI_TEST*
#774: Handle FIXMEs for multi-column code examples

#732: Put the register optimization code into an example
#720: Move description of lock types to front of Lock section
#716: Clean up presentation of Window info keys
#709: Use listing with fancyvb for multicolumn code examples
#705: Minor edits from MPI-2.2 review
#702: Improve primary index entries
#701: Consistent section titles for RMA functions
#698: Editor fixes
#697: Improve example index
#693: Review use of emph where mpindex should be used
#689: Fix table formatting and squash overfull boxes
#686: Consistent formatting for lists of MPI constants and objects.
#685: Replaces use of \const with more specific macros
#683: Replace which with that where appropriate
#677: Change non-xxx to nonxxx in most cases.
#631: Implementation of enhanced example index.
#656: Code format changes
#642: scrbook document style
#630: v4.x: remove dead code and comments

```
% git log --first-parent --pretty=format:"%Cred%h%Creset %cs %Cgreen%<(25,trunc)%s%Creset %b" mpi-4.0..HEAD - chap-one-side
```

#749: Add info key for RMA accumulate behavior preference

"mpi_accumulate_granularity" (integer, default `0`): provides a hint to implementations about the desired synchronization granularity for accumulate operations, i.e., the size of memory ranges in bytes for which the implementation should acquire a synchronization primitive to ensure atomicity of updates. If the specified granularity is not divisible by the size of the type used in an accumulate operation, it should be treated as if it was the next multiple of the element size. For example, a granularity of 1 byte should be treated as 8 in an accumulate operation using MPI_UINT64_T. By default, this info key is set to `0`, which leaves the choice of synchronization granularity to the implementation. If specified, all MPI processes in the group of a window must supply the same value.

Advice to users. Small synchronization granularities may provide improved latencies for accumulate operations with few elements and potentially increase concurrency of updates, at the cost of lower throughput. For example, a value matching the size of a type involved in an accumulate operation may enable implementations to use atomic memory operations instead of mutual exclusion devices. Larger synchronization granularities may yield higher throughput of accumulate operation with large numbers of elements due to lower synchronization costs, potentially at the expense of higher latency for accumulate operations with few elements, e.g., if atomic memory operations are not employed. By dividing larger accumulate operations into smaller segments, concurrent accumulate operations to the same window memory may update different segments in parallel. (*End of advice to users.*)

Advice to implementors. Implementations are encouraged to avoid mutual exclusion devices in cases where the granularity is small enough to warrant the use of atomic memory operations. For larger granularities, implementations should use this info value as a hint to partition the window memory into zones of mutual exclusion to enable segmentation of large accumulate operations. (*End of advice to implementors.*)

#708: Allow MPI_WIN_SHARED_QUERY on created and allocated windows

- Make memory of allocated window available as shared memory
- Add definition of shared shared memory domain to semantic terms

Implementations may make the memory provided by the user available for load/store accesses by MPI processes in the same *shared memory domain*. A communicator of such processes can be constructed as described in Section 7.4.2 using MPI_COMM_SPLIT_TYPE. Pointers to access a *shared memory segment* can be queried using MPI_WIN_SHARED_QUERY.

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. For details on how to create a consistent view see the description of MPI_WIN_SHARED_QUERY.

MPI processes reside in the same **shared memory domain** if it is possible to share a segment of memory between them, i.e., to make a segment of memory (**shared memory segment**) concurrently accessible from all of those MPI processes through load/store accesses. For a group of processes belonging to more than one *shared memory domain* the creation of a subgroup of processes belonging to the same *shared memory domain* is defined in Section 7.4.2.

Only MPI_WIN_ALLOCATE_SHARED is guaranteed to allocate *shared memory*. Implementations are permitted, where possible, to provide *shared memory* for windows created with MPI_WIN_CREATE and MPI_WIN_ALLOCATE. However, availability of *shared memory* is not guaranteed. When the remote memory segment corresponding to a particular process cannot be accessed directly, this call returns size = 0 and a baseptr as if MPI_ALLOC_MEM was called with size = 0.

Rationale. MPI_WIN_SHARED_QUERY may only be called on windows created by a call to MPI_WIN_ALLOCATE_SHARED, MPI_WIN_ALLOCATE, or MPI_WIN_CREATE. The potential for multiple memory regions in windows created through MPI_WIN_CREATE_DYNAMIC means that these windows cannot be used as input for MPI_WIN_SHARED_QUERY. (*End of rationale.*)

Advice to users. For windows allocated using MPI_WIN_ALLOCATE or MPI_WIN_CREATE, the group of processes for which the implementation may provide shared memory can be determined using MPI_COMM_SPLIT_TYPE described in Section 7.4.2. (*End of advice to users.*)

The consistency of load/store accesses from/to the *shared memory* as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define the semantics for accessing *shared window memory* in the *separate memory model*.

#659: MPI_Win_free is synchronizing, except when it isn't

- MPI_WIN_FREE is synchronizing if the “no_lock” info key is not “true”

Advice to implementors. MPI_WIN_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win call free. This ensures that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. The only exception to this rule is when the user sets the "no_locks" info key to "true" when creating the window. In that case, an MPI implementation may free the local window without barrier synchronization. (*End of advice to implementors.*)

MPI_WIN_FREE is required to delay its return until all accesses to the local window using passive target synchronization have completed. Therefore, it is synchronizing unless the window was created with the "no_locks" info key set to "true".

#747: Rephrase MPI_WIN_SYNC to make it clear what it's used for

- Clarify that MPI_WIN_SYNC can be used to order load/store operations in shared window memory

The call MPI_WIN_SYNC synchronizes the private and public window copies of win. For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the effect of ending and reopening an access and exposure epoch on the window (note that it does not actually end an epoch or complete any pending MPI RMA operations).

For windows in the separate memory model, a call to MPI_WIN_SYNC synchronizes the private and public window copies of win at the calling MPI process, as described in Section 11.7.

In the unified memory model, MPI_WIN_SYNC may be used to order load and store accesses to shared memory and to ensure visibility of store updates in shared memory for other threads and MPI processes.

A call to MPI_WIN_SYNC does not open or close an epoch and does not complete any pending RMA operations. A call to MPI_WIN_SYNC does not guarantee *progress* of any pending MPI operation.

#729: MPI_WIN_TEST with same progress as MPI_TEST

- MPI_WIN_TEST returns “flag = true” **eventually** once all accesses to the local window have completed

This is the nonblocking 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_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the

MPI_WIN_TEST is a local procedure. Repeated calls to MPI_WIN_TEST with the same win argument will eventually return flag = true once all accesses to the local window by the group to which it was exposed by the corresponding call to MPI_WIN_POST have been completed as indicated by matching MPI_WIN_COMPLETE calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned immediately. The

Full Chapter PDF Diff

https://draftable.com/compare/wTFGrxhxwflV

Open PRs

CC Changes: https://github.com/mpi-forum/mpi-standard/pull/857

Further changes go into separate PRs and tickets.

Github Diff

https://github.com/mpi-forum/mpi-standard/compare/mpi-4.0...mpi-4.x?expand=1#diff-abcde0810b89c0faf160df24683574bc0f017a2e677e3b8dd0d2d73116877a9e

Reviews

- Page 548, line 23: I think the phrasing "is made accessible to accesses" may come across as a bit odd. I suggest "can be accessed" instead.

→ Suggestion: “is made available to accesses”

- Page 555, line 19: the use of the word "returns" might be seen as inacurrate when refering to the fact that the call to MPI_Win_allocate_shared fills the pointer "win" passed with the window created. What is returned by this call however is the error code that MPI routines return.

→ I think this is the wording we use across the standard?

Reviews

- Group of processes -> group of MPI processes?
  → 2 occurrences across the chapter
- Use of MPI-2:
  The **MPI-2** RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make onesided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using malloc or new respectively. In **MPI-2** RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory.
  → Should be changed through CC change

Reviews

- nonnegative vs. non-negative inconsistent
  → nonnegative is the canonical way?
- MPI RMA call → MPI RMA operation:
  “The user is also responsible for ensuring that MPI_WIN_ATTACH at the target has returned before an MPI process attempts to target that memory with an MPI RMA call.”
- As all MPI RMA communication functions are incomplete, they must be completed [separately].

These examples show the use of the MPI_GET function. As all MPI RMA communication functions are incomplete, they must be completed. In the following, this is accomplished with the routine MPI_WIN_FENCE, introduced in Section 11.5.

Reviews

- "A new predefined operator, MPI_REPLACE, is defined" -> This is not "new" in any sense today.
  → "an **additional** predefined operator, MPI_REPLACE, is defined"
- Any of the predefined operators for MPI_REDUCE can be used. User-defined **functions** cannot be used.
  → User-defined operators?
- A new predefined operator MPI_NO_OP
  → A predefined operator MPI_NO_OP
- Contradicting rules about use of MPI_REQUEST_FREE

in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The closing of the epoch, or explicit bulk synchronization using MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of request-based RMA operations on the specified window. However, users must still free the request by testing, waiting, or calling MPI_REQUEST_FREE on the request handle to allow the MPI implementation to release any resources associated with these requests.

Current RMA Chapter Committee

- Bill Gropp
- Joseph Schuchart
- Jeff Hammond
- Dan Holmes
- Christoph Niethammer
- Thomas Gillis