MPI Ticket #456 Overview and Solution

Rolf Rabenseifner

Shared Memory Synchronization – The Text Basis

MPI-3.0 Sect.11.2.3 on MPI_WIN_ALLOCATE_SHARED

- page 409, line 16

  "The locally allocated memory can be the target of load/store accesses by remote processes;"

- page 410, line 15-21

  "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."

MPI_Win_fence

Section 11.5 Synchronization Calls

- Page 437 lines 44ff

  “The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

  This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.”

MPI_Win_fence

With RMA calls

Rank 0	Rank 1
MPI_Win_fence	MPI_Win_fence
MPI_Put(val_1,…to A in win on rank 1)	
MPI_Win_fence	MPI_Win_fence
	buf=A
May be repeated	print buf

First fence is to switch from computational to exposure/access epoch.
Second fence is needed to locally and remotely finish the put operation, and to synchronize, i.e., to inform process rank 1 that data in A is available, and to switch back to the computational epoch, i.e., that A can now be locally loaded in process rank 1.

Shared memory expectation, based on cited MPI-3.0 wording:

Rank 0	Rank 1
MPI_Win_fence	MPI_Win_fence
A = val_1 (A is in shared memory win)	
MPI_Win_fence	MPI_Win_fence
	buf=A
	print buf

First fence is to switch from computational to exposure/access epoch.
Second fence is needed ~~to locally and remotely finish the put operation, and~~ to synchronize, i.e., to inform process rank 1 that data in A is available, and to switch back to the computational epoch, i.e., that A can now be locally loaded in process rank 1.

MPI_Win_fence

With RMA calls

Rank 0	Rank 1
A=val_1 (A is in win on rank 0)	
MPI_Win_fence	MPI_Win_fence
	MPI_Get(buf,..win)
MPI_Win_fence	MPI_Win_fence
	print buf

May be repeated

First fence is needed to synchronize, i.e., to inform process rank 1 that data in A is available, i.e., to switch from computational to exposure/access epoch.
Second fence is needed to locally and remotely finish the get operation and to switch back to the computational epoch, i.e., that A can now be overwritten in process rank 0.

Shared memory expectation, based on cited MPI-3.0 wording:

Rank 0	Rank 1
A=val_1 (A is in shared memory window win)	
MPI_Win_fence	MPI_Win_fence
	buf = A
	print buf
MPI_Win_fence	MPI_Win_fence

First fence is needed to synchronize, i.e., to inform process rank 1 that data in A is available, i.e., to switch from computational to exposure/access epoch.
Second fence is needed ~~to locally and remotely finish the get operation and~~ to switch back to the computational epoch, i.e., that A can now be overwritten in process rank 0.

MPI_Win_fence on shared memory can be summarized with:

Three rules for A, B, C in a shared memory windows:
(accessed in both processes through the same variable name)

Process rank 0	Process rank 1	It is guaranteed that …
A=val_1 MPI_Win_fence	 MPI_Win_fence load(A)	… the load(A) in P1 loads val_1 (this is the write-read-rule)
load(B) MPI_Win_fence	 MPI_Win_fence B=val_2	… the load(B) in P0 is not affected by the store of val_2 in P1 (read-write-rule)

C=val_3
MPI_Win_fence
	
	MPI_Win_fence
	C=val_4
	load(C)
		… that the load(C) in P1 loads val_4 (write-write-rule)

MPI_Win_post / _start / _complete / _wait (based on MPI-3.0 pages 438-439)

With RMA calls

Rank 0	Rank 1
A=val_1 (A is in win)	
MPI_Win_post	MPI_Win_start
	MPI_Get(buf,..win)
MPI_Win_wait	MPI_Win_complete
May be repeated	print buf

MPI_Win_post + _start is needed to synchronize, i.e., to inform process rank 1 that data in A is available, i.e., to switch from computational to exposure/access epoch.
MPI_Win_complete is needed to locally and remotely finish the get operation and to switch back to the computational epoch in process rank 1.
MPI_Win_wait is needed to finish the exposure/access epoch in process rank 0, i.e., that A can now be overwritten.

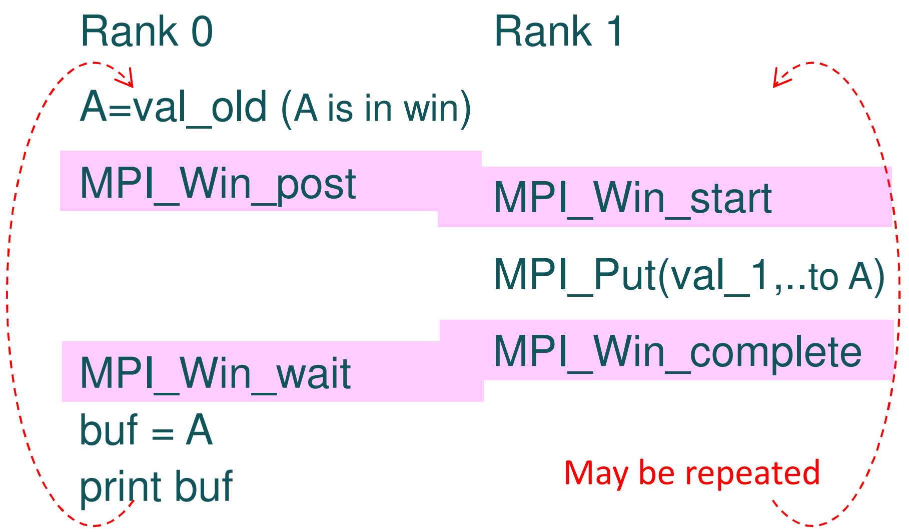
Shared memory expectation:

Rank 0	Rank 1
A=val_1 (A is in shared memory window win)	
MPI_Win_post	MPI_Win_start
	buf = A
	print buf
MPI_Win_wait	MPI_Win_complete

MPI_Win_post + _start is needed to synchronize, i.e., to inform process rank 1 that data in A is available, i.e., to switch from computational to exposure/access epoch.
MPI_Win_complete is needed ~~to locally and remotely finish the get operation and~~ to switch back to the computational epoch in process rank 1.
MPI_Win_wait is needed to finish the exposure/access epoch in process rank 0, i.e., that A can now be now overwritten.

MPI_Win_post / _start / _complete / _wait

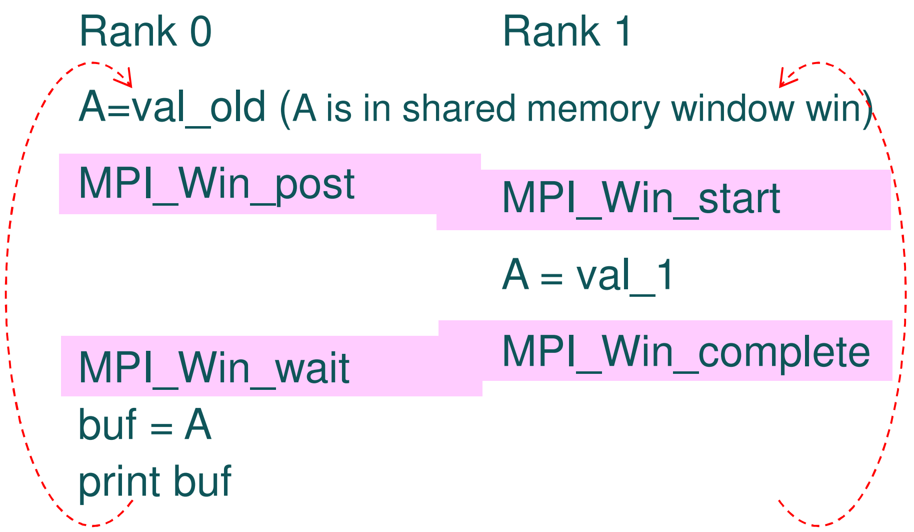
With RMA calls



Rank 0	Rank 1
A=val_old (A is in win)	
MPI_Win_post	MPI_Win_start
	MPI_Put(val_1,..to A)
MPI_Win_wait	MPI_Win_complete
buf = A	
print buf	

MPI_Win_post + _start is needed to synchronize, i.e., to inform process 1 that data in A can be overwritten, i.e., to switch from computational to exposure/access epoch. MPI_Win_complete is needed to locally and remotely finish the put operation and to synchronize, i.e., to inform process 0 that new data in A is available. MPI_Win_wait is needed to finish the exposure/access epoch in process rank 0, i.e., that A can now be locally loaded.

Shared memory expectation:



Rank 0	Rank 1
A=val_old (A is in shared memory window win)	
MPI_Win_post	MPI_Win_start
	A = val_1
MPI_Win_wait	MPI_Win_complete
buf = A	
print buf	

MPI_Win_post + _start is needed to synchronize, i.e., to inform process 1 that data in A can be overwritten, i.e., to switch from computational to exposure/access epoch. MPI_Win_complete is needed ~~to locally and remotely finish the put operation and~~ to synchronize, i.e., to inform process 0 that new data in A is available. MPI_Win_wait is needed to finish the exposure/access epoch in process rank 0, i.e., that A can now be locally loaded.

Summarizing MPI_Win_post / _start / _complete / _wait

Three rules for A, B, C in a shared memory windows:
(accessed in both processes through the same variable name)

Process rank 0	Process rank 1	It is guaranteed that …
A=val_1 MPI_Win_post	MPI_Win_start load(A)	… the load(A) in P1 loads val_1 (this is the write-read-rule)
load(B) MPI_Win_post	MPI_Win_start B=val_2	… the load(B) in P0 is not affected by the store of val_2 in P1 (read-write-rule)

C=val_3
MPI_Win_post
	MPI_Win_start
	C=val_4
	load(C)
		… that the load(C) in P1 loads val_4 (write-write-rule)

Summarizing MPI_Win_post / _start / _complete / _wait

Three rules for A, B, C in a shared memory windows:
(accessed in both processes through the same variable name)

Process rank 0	Process rank 1	It is guaranteed that …
A=val_1 MPI_Win_complete	MPI_Win_wait load(A)	… the load(A) in P1 loads val_1 (this is the write-read-rule)
load(B) MPI_Win_complete	MPI_Win_wait B=val_2	… the load(B) in P0 is not affected by the store of val_2 in P1 (read-write-rule)

C=val_3
MPI_Win_complete
	MPI_Win_wait
	C=val_4
	load(C)
		… that the load(C) in P1 loads val_4 (write-write-rule)

Summarizing MPI_lock / _unlock

Three rules for A, B, C in a shared memory windows (in both processes with the same name) and for the case that the lock in process 0 is granted before process 1:

Process rank 0	Process rank 1	It is guaranteed that …
MPI_Win_lock A=val_1 MPI_Win_unlock	MPI_Win_lock load(A) MPI_Win_unlock	… the load(A) in P1 loads val_1 (this is the write-read-rule)
MPI_Win_lock load(B) MPI_Win_unlock	MPI_Win_lock B=val_2 MPI_Win_unlock	… the load(B) in P0 is not affected by the store of val_2 in P1 (read-write-rule)

MPI_Win_lock
C=val_3
MPI_Win_unlock
	MPI_Win_lock
	C=val_4
	load(C)
	MPI_Win_unlock
		… that the load(C) in P1 loads val_4 (write-write-rule)

Summarizing MPI_Win_sync

Three rules for A, B, C in a shared memory windows (in both processes with the same name):

Process rank 0	Process rank 1	It is guaranteed that …
A=val_1 MPI_Win_sync Any-process-sync	Any-process-sync MPI_Win_sync load(A)	… the load(A) in P1 loads val_1 (this is the write-read-rule)
load(B) MPI_Win_sync Any-process-sync	Any-process-sync MPI_Win_sync B=val_2	… the load(B) in P0 is not affected by the store of val_2 in P1 (read-write-rule)

C=val_3
MPI_Win_sync
Any-process-sync
	Any-process-sync
	MPI_Win_sync
	C=val_4
	load(C)
		… that the load(C) in P1 loads val_4
		(write-write-rule)

"Any-process-sync" may be done with methods from MPI (e.g. with send-->recv as in Example 11.13, but also with some synchronization through MPI shared memory loads and stores as in Example 11.14).

Summarizing these rules

- Always three rules: W-R, R-W, W-W
- All is based on the small MI-3.0 text

  "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)"
- and the description in Section 11.5, pages 437-439.

Open questions

- How to get the details right?
  - (A) Remote shared memory load/store
    = Remote memory access operation (**RMA operations**)
    in the sense of Section 11.7 Semantics and Correctness

  or

  - (B) Saying that **RMA operations** are only RMA procedure calls
    and writing a new section on
    The semantics of Shared Memory Synchronization
    (in the RMA-WG-telcon on … we preferred this second approach)

  or

  - (C) Saying that **RMA operations** are only RMA procedure calls
    and removing the shared memory consistency-requirement
    on MPI-3.0 page 410, line 15-21

Pros and Cons

(A) RMA Operations

= Both, shared memory RMA (load/store) + RMA routines (Get, Put, …)

- Pro: This is more or less the current text
- Con: Hard to read, text may not fit to both, hard to fix such details
- Con: *"Remote"* and *"local"* shared memory load/store does not fit to the nature of shared memory: We only have accesses from different processes, no process is *"owner"* of a portion.

(B) New section on shared memory load/store synchronization semantics

- Pro: Can define exactly the shared memory synchronization services
- Pro: Maintenance independent of RMA operations
- Con: Not easy to fit exactly to the current text and its implied expectations

(C) No shared memory synchronization semantics

- Pro: No conflict with missing/vague language/compiler support
- Con: Portable MPI programming is language dependent
- Con: Risk of double/multiple memory fences (one/some inside of MPI process-to-process synchronization and one/some on application level → **high latency!**

Concentrating on Performance Aspects

(A) + (B) No shared memory synchronization semantics

- Pro: MPI library can minimize the total number of memory fences
- Pro: User need not to learn about
  - C11 / C++11, Section 7.17.4.1 The atomic_thread_fence function page 278-278, http://port70.net/~nsz/c/c11/n1570.pdf, (pdf: 296-297)
  - How to use C11 atomic_thread_fence through Fortran???
- Pro: Fortran only need to add C11 memory/optimization semantics to buffers declared as ASYNCHRONOUS
- Con: If an application uses on a shared memory window both, shared memory load/store and MPI_Put/Get, then MPI synchronization may issue not needed memory fences
  - Can be easily resolved with appropriate assertions

Concentrating on Performance Aspects

(C) shared memory fences as part of MPI RMA synchronization
without further fences on application programming level

- Con: MPI library can minimize the total number of memory fences
  - Risk of double/multiple memory fences (one/some inside of MPI process-to-process synchronization and one/some on application level → **high latency!**
- Con: User need to learn about
  - C11 / C++11, Section 7.17.4.1 The atomic_thread_fence function page 278-278, http://port70.net/~nsz/c/c11/n1570.pdf, (pdf: 296-297)
- Con: No support for Fortran
- Con: Does not fit to currently implemented behavior
  - Including books / tutorials / publications from forum members
- Con: Not backward compatible to current text
  - Requires long-term changes → deprecating current text → removing

Ticket #456 as a possible solution

- Based on the rules on slides 6 and 9-12
- These rules should be a complete set of necessary and sufficient rules
- Parts:
  - "The rules for RMA operations do not apply to these remote load/store accesses"
    → MPI-3.0 Sect.11.2.3 on MPI_WIN_ALLOCATE_SHARED, page 409, lines 13-22
  - "The visibility of loads and stores in several processes may not be at the same time and not in the same sequence" + a rationale
    → MPI-3.0 Section 11.4 Memory Model, page 436, lines 37-41 –
    text on the RMA unified model
    Remark: I do not describe "local" and "remote" loads/stores.
    I describe loads and stores "from different processes".
  - **New Section "Shared Memory Synchronization"**
    → before MPI-3.0 11.5.5 Assertions, page 451, line 4
  - Assertions: "stores" → "local stores" + "In the case of a shared memory window (i.e., allocated with MPI_WIN_ALLOCATE_SHARED), such local stores can be issued to any portion of the shared memory window."

Ticket #456 as a possible solution (continued)

- Parts (continued):
  - Necessary corrections to passed ticket #413:
    - Advice to users: can be removed due to new **Section “Shared Memory Synchronization”**
    - Correction to example about MPI_Win_sync:
      Second pair of MPI_Win_sync due to read-write-rule
      (was overseen when developing this example)
  - New MPI shared memory example with a MPI-communication-free synchronization

Open questions

- I understood the implications for shared memory synchronizations with
  - MPI_Win_fence
  - MPI_Win_post / _start / _complete / _wait
  - MPI_Win_lock / _unlock
  - MPI_Win_sync
- I did not understood nor included
  - MPI_Win_lock_all / _unlock_all
  - MPI_Win_flush / _flush_all / _flush_local / _flush_local_all

  Should be included if useful and/or implied from the existing consistency rule
- Which additional rules do we need in the next Fortran standard for ASYNCHRONOUS buffers?
  - Can be included when including TS29113 into next Fortran standard.

Other problems:

MPI-3.0, page 248 lines 1-6 define for MPI_COMM_SPLIT_TYPE():

> "MPI_COMM_TYPE_SHARED this type splits the communicator into subcommunicators, each of which can create a shared memory region.
>
> > *Advice to implementors*. Implementations can dene their own types, or use the info argument, to assist in creating communicators that help expose platform-specific information to the application. (*End of advice to implementors.*)"

- Missing information: MPI_Win_allocate_shared() is allowed on any communicator with a group of processes being a subgroup the group of processes of the returned subcommunicator. On such communicator, MPI_Win_allocate_shared() will return a window in the unified model.
- Can be included into this ticket or handled separately.

Acknowledgements

- Many people helped and teached me how to put the puzzle together
  - Bill Gropp – especially for showing me Boehm's problem report
  - Dave Goodell – showing me C11/C++11 memory fence
    and giving me the pointer to Paul E. McKenney's report:
    http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.pdf
  - Pavan Balaji and Torsten Hoefler – for showing me how weak the current wording is
  - Jeff Hammond – for all his hints on #456 and its precursors
  - Jim Dinan – for his help with the MPI_Win_sync example
  - Jeff Squyres – especially for pointing me to the consistency rule
  - Fab Tillier – showing me the problem of duplicated memory fences
  - Hubert Ritzdorf – the idea that the MPI library should also take into account
    what readers expect from the text & his initial request 2 years ago
  - Keith Underwood – for his discussion on C11
  - Rajeev Thakur – especially for remembering me that there are two issues in #456,
    the process synchronization and the memory barriers
  - Bill Long – for all about Fortran
  - And my apologies, that this list may be incomplete and unsorted