D R A F T

Document for a Standard Message-Passing Interface

MPI-3 One Sided Working Group

October 27, 2010

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 11

One-Sided Communications

11.1 Introduction

Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or update at other processes. However, processes may not know which data in their own memory need to be accessed or updated by remote processes, and may not even know the identity of these processes. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This may require all processes to participate in a time consuming global computation, or to periodically poll for potential communication requests to receive and act upon. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form `A = B(map)`, where `map` is a permutation vector, and `A`, `B` and `map` are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver; and *synchronization* of sender with receiver. The RMA design separates these two functions. [Three]Five communication calls are provided: MPI_PUT (remote write), MPI_GET (remote read), [and] MPI_ACCUMULATE (remote update), MPI_GET_ACCUMULATE (remote fetch and update), and MPI_COMPARE_AND_SWAP (remote atomic swap operations).

MPI supports two fundamentally different memory models. The first model makes no assumption about memory consistency and is highly portable. This model is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be imposed by the user, using synchronization calls; the implementation can delay communication operations until the synchronization calls occur, for efficiency. The second model can exploit cache-coherent hardware and hardware-accelerated one-sided operations which are commonly available in high-performance systems. In this model, communications can be independent of synchronization calls. The two different models are discussed in detail in Section 11.5. A large number of synchronization calls is provided for both models to support different synchronization styles.

The design of the RMA functions allows implementors to take advantage, in many cases, of fast or asynchronous communication mechanisms provided by various platforms, such as coherent or noncoherent shared memory, DMA engines, hardware-supported put/get operations, communication coprocessors, etc.. The most frequently used RMA communication mechanisms can be layered on top of message-passing. However, support for asynchronous communication agents in software (handlers, threads, etc.) [is]might be needed, for certain RMA functions, in a distributed memory environment.

We shall denote by **origin** the process that performs the call, and by **target** the process in which the memory is accessed. Thus, in a put operation, source=origin and destination=target; in a get operation, source=target and destination=origin.

11.2 Initialization

[The initialization operation]MPI provides [two]three initialization functions, MPI_WIN_CREATE[and], MPI_WIN_ALLOCATE, and MPI_WIN_CREATE_DYNAMIC. MPI_WIN_CREATE allows each process in an intracommunicator group to specify, in a collective operation, a "window" in its memory that is made accessible to accesses by remote processes. The call returns an opaque object that represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from MPI_WIN_CREATE in that the user does not pass allocated memory; MPI_WIN_ALLOCATE allocates memory and returns a pointer to it. MPI_WIN_CREATE_DYNAMIC creates a window that allows to attach (register) and detach (deregister) process memory locally.

11.2.1 Window Creation

MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)

IN	base	initial address of window (choice)
IN	size	size of window in bytes (non-negative integer)
IN	disp_unit	local unit size for displacements, in bytes (positive integer)
IN	info	info argument (handle)
IN	comm	communicator (handle)
OUT	win	window object returned by the call (handle)

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
              MPI_Comm comm, MPI_Win *win)
```

```
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
```

```
{static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int
              disp_unit, const MPI::Info& info, const MPI::Intracomm& comm)
              (binding deprecated, see Section 15.2) }
```

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. A process may elect to expose no memory by specifying size = 0.

The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor disp_unit specified by the target process, at window creation.

> *Rationale.* The window size is specified using an address sized integer, so as to allow windows that span more than 4 GB of address space. (Even if the physical memory size is less than 4 GB, the address range may be larger than 4 GB, if addresses are not contiguous.) (*End of rationale.*)

> *Advice to users.* Common choices for disp_unit are 1 (no scaling), and (in C syntax) sizeof(type), for a window that consists of an array of elements of type type. The later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (*End of advice to users.*)

The info argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info key is predefined:

no_locks — if set to true, then the implementation may assume that the local window is never locked (by a call to MPI_WIN_LOCK). This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.

unordered — if set to true, then the implementation may assume that the application will explicitly handle ordering of RMA operations through explicit synchronization.

The various processes in the group of comm may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the get, put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to [erroneous]undefined results.

> *Rationale.* The reason for specifying the memory that may be accessed from another process in an RMA operation is to permit the programmer to specify what memory can be a target of RMA operations and for the implementation to enforce that specification. For example, with this definition, a server process can safely allow a client process to use RMA operations, knowing that (under the assumption that the MPI implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (*End of rationale.*)

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM (Section 8.2, page 296) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

(WDG COMMENT: Note the above description of MPI_ALLOC_MEM. The behavior of MPI_WIN_REGISTER should be similar (which is an argument for register with size and free with just the pointer, and no registration handles).)

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

Implementors should document any performance impact of window alignment. (*End of advice to implementors.*)

11.2.2 Window That Allocates Memory

MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)

IN	size	size of window in bytes (non-negative integer)
IN	disp_unit	local unit size for displacements, in bytes (positive integer)
IN	info	info argument (handle)
IN	comm	communicator (handle)
OUT	baseptr	initial address of window (choice)
OUT	win	window object returned by the call (handle)

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
              MPI_Comm comm, void **base, MPI_Win *win)
```

```
MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This is a collective call executed by all processes in the group of comm. On each process, it allocates memory of at least size size bytes, returns a pointer to it, and returns a window object that can be used by all processes in comm to perform RMA operations. The returned memory consists of size bytes local to each process, starting at address baseptr and is associated with the window as if the user called MPI_WIN_CREATE on existing memory. The size argument may be different at each process and size = 0 is valid, however, a library might allocate and expose more memory in order to create a fast, globally symmetric allocation. The discussion of MPI_ALLOC_MEM in Section 8.2 also applies to MPI_WIN_ALLOCATE.

Rationale. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access significantly. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (*End of rationale.*)

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

11.2.3 Window of Dynamically Allocated Memory

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 implementors (special steps may be taken to make one-sided access to such memory more efficient). However, it makes other uses of RMA more difficult. For example, consider accessing, using RMA operations, a linked list that is modified. As new items are added to that 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 an MPI_Win object with a predefined amount of memory and then implement routines for allocating memory from within that memory. In addition, there is no easy way to handle the situation where the predefined amount of memory turns out to be inadequate. To support this model, the routine MPI_WIN_CREATE_DYNAMIC creates an MPI_Win that makes it possible to expose memory without remote synchronization. This is combined with local routines to add/remove memory from this window.

MPI_WIN_CREATE_DYNAMIC(info, comm, win)

IN	info	info argument (handle)
IN	comm	communicator (handle)
OUT	win	window object returned by the call (handle)

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

```
MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
    INTEGER INFO, COMM, WIN, IERROR
```

This is a collective call executed by all processes in the group of comm. It returns a window win without memory attached. Existing process memory can be attached (registered) as described below. This routine returns a window object that can be used by these processes to perform RMA operations on registered memory.

Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE.

no_localexclusive — if set to true, then the implementation may assume that the local window is never locked (by a call to MPI_WIN_LOCK) with lock mode MPI_LOCK_EXCLUSIVE by the local process.

(WDG COMMENT: We should remove this info key if no implementor speaks up for it.)

(COMMENT: we should make Info more useful in general, i.e., add attach and query functions for communicator and window (at least) so that it works with libraries. However, this issue is orthogonal to this proposal because MPI-2 has the same problem with no_locks, I shall go ahead and propose a generic attach/query interface which would also be useful for other assertion-like infos.)

Memory in this window may not be used as the target of one-sided accesses in this window until it is registered using the function MPI_WIN_REGISTER. That is, in addition to using MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use MPI_WIN_REGISTER before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be registered. For simplicity in use, memory may be registered multiple times (though this is not encouraged).

MPI_WIN_REGISTER(win, base, size)

IN	win	window object (handle)
IN	base	initial address of memory to be registered
IN	size	size of memory to be registered in bytes

```
int MPI_Win_register(MPI_Win win, void *base, MPI_Aint size)
```

```
MPI_WIN_REGISTER(WIN, BASE, SIZE, IERROR)
    INTEGER WIN, IERROR
    <type> base
    INTEGER (KIND=MPI_ADDRESS_SIZE) size
```

Registers a local memory region of size size beginning at base for remote access within the given window.

Rationale. Requiring that memory be explicitly registered before it is exposed to one-sided access by other processes can significantly help implementations, including ensuring high performance. The ability to make memory available for RMA operations without requiring a collective MPI_WIN_CREATE call is needed for some one-sided programming models. (*End of rationale.*)

> *Advice to users.* Memory registration may require the use of scarce resources; thus, registering large regions of memory is not recommended in portable programs. Memory registration may fail if sufficient resources are not available; this is similar to the behavior of MPI_ALLOC_MEM.
>
> The user is also responsible for ensuring that memory registration at the target has completed before a process attempts to target that memory with an MPI RMA call.
>
> Performing an RMA operation to unregistered memory from a window created with MPI_WIN_CREATE_DYNAMIC is erroneous. (*End of advice to users.*)

> *Advice to implementors.* A high-quality implementation will attempt to make as much memory available for registration as possible. Any limitations should be documented by the vendor. (*End of advice to implementors.*)

Memory registration is a local operation as defined by MPI; that means that the call is not collective and completes without requiring any MPI routine to be called on any other process. Memory may be deregistered with the routine MPI_WIN_DEREGISTER. If memory was registered n times, then it is only deregistered after it was passed to MPI_WIN_DEREGISTER n times. After memory has been deregistered, it may not be the target of an MPI RMA operation in that window (unless that memory is re-registered with MPI_WIN_REGISTER).

MPI_WIN_DEREGISTER(win, base, size)

IN	win	window object (handle)
IN	base	initial address of memory to be deregistered
IN	size	size of memory to be deregistered in bytes

```
int MPI_Win_deregister(MPI_Win win, void *base, MPI_Aint size)
```

```
MPI_WIN_DEREGISTER(WIN, BASE, SIZE, IERROR)
    INTEGER WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_SIZE) size
    <type> base
```

Deregisters a previously registered memory region of size size beginning at base. The arguments base and size must match the arguments passed to a previous call to MPI_WIN_REGISTER. (COMMENT: if we allow something like MPI_ANY for size, then we force the MPI implementation to store base and size for all registrations. This would not be necessary if page-based registration (with refcounts) are used as e.g., OpenMX does. However, if we do pure range-based registration and deregistration then we're hosed too with ref-counting and base-addresses. Another question is if base has be a registration base, i.e., would it be possible to register(1,4), deregister(2,4), deregister(1,3) or such. This becomes really complex in the genral case – I am tending towards returning handles to the user. This allows highest flexibility for implementation and user. Let's talk ... handles can be used to identify registered memory easily and can just be NULL if registration is not needed, however, users would then still need to save them :-()

> *Advice to users.* Deregistering memory may permit the implementation to make more efficient use of special memory or provide memory that may be needed by a

subsequent MPI_WIN_REGISTER. Users are encouraged to deregister memory that is no longer needed. (*End of advice to users.*)

(WDG COMMENT: WIN_REGISTER and ALLOC_MEM (and WIN_DEREGISTER and FREE_MEM) should have similar interfaces if at all possible.)

(COMMENT: begin) An alternative version of register and deregister could have handle arguments to identify registrations:

MPI_WIN_REGISTER(win, base, size, reg)

IN	win	window object (handle)
IN	base	initial address of memory to be registered
IN	size	size of memory to be registered in bytes
OUT	reg	registration (handle)

```
int MPI_Win_register(MPI_Win win, void *base, MPI_Aint size, MPI_Reg *reg)
```

```
MPI_WIN_REGISTER(WIN, BASE, SIZE, REG, IERROR)
    INTEGER WIN, REG, IERROR
    <type> base
    INTEGER (KIND=MPI_ADDRESS_SIZE) size
```

MPI_WIN_DEREGISTER(win, reg)

IN	win	window object (handle)
INOUT	reg	registration (handle)

```
int MPI_Win_deregister(MPI_Win win, MPI_Reg *reg)
```

```
MPI_WIN_DEREGISTER(WIN, REG, IERROR)
    INTEGER REG, IERROR
```

(COMMENT: end)

If the window was created with MPI_WIN_CREATE_DYNAMIC, any memory registered with MPI_WIN_REGISTER may become unregistered when the window is freed.

Advice to users. It is recommended that users deregister all memory before freeing a dynamic window. (*End of advice to users.*)

(COMMENT: I think we should make deregistration mandatory because after the window is freed, memory cannot be deregistered (needs valid win) and this is essentially a resource leak.)

In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for all RMA functions is the address at the target. I.e., the effective window_base is MPI_BOTTOM and the disp_unit is one. Users should use MPI_GET_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI_AINT (cf. Table 3.3 on Page 29) is able to store addresses from any process. (*End of advice to implementors.*)

11.2.4 Window Destruction

MPI_WIN_FREE(win)

INOUT	win	window object (handle)

```
int MPI_Win_free(MPI_Win *win)
```

```
MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR
```

{`void MPI::Win::Free()` *(binding deprecated, see Section 15.2)* }

Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: i.e., the process has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. [When the call returns, the window memory can be freed.]The memory associated with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_ALLOCATE.

Advice to implementors. MPI_WIN_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win called free. This, to ensure that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. (WDG COMMENT: This statement is not if nolocks was true (no passive-target access).) (*End of advice to implementors.*)

11.2.5 Window Attributes

The following [three] attributes are cached with a window[,] when the window is created.

MPI_WIN_BASE	window base address.
MPI_WIN_SIZE	[ ]window size, in bytes.
MPI_WIN_DISP_UNIT	displacement unit associated with the window.
MPI_WIN_CREATE_FLAVOR	how window was created.

In C, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag)[and] MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag)[] and MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag) will return in base a pointer to the start of the window win, and will return in size[and], disp_unit, and in create_kind pointers to the size[and], displacement unit of the window, and the kind of routine used to create the window, respectively. [And similarly, in C++.]And similarly, in C++ *(binding deprecated, see Section 15.2)*.

In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror)[and], MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror)[] and MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror) will return in base, size[and], disp_unit and create_kind the (integer representation of) the base address, the size[and], the displacement unit of the window win, and the kind of routine used to create the window, respectively.

The values of create_kind are

MPI_WIN_FLAVOR_CREATE	Window was created with MPI_WIN_CREATE.
MPI_WIN_FLAVOR_ALLOCATE	Window was created with MPI_WIN_ALLOCATE.
MPI_WIN_FLAVOR_DYNAMIC	Window was created with MPI_WIN_CREATE_DYNAMIC.

In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address is MPI_BOTTOM and the size is 0. In C, pointers to integers (of size MPI_Aint) are returned and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 6.7.3, page 252.)

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

MPI_WIN_GET_GROUP(win, group)

IN	win	window object (handle)
OUT	group	group of processes which share access to the window (handle)

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR
```

{MPI::Group MPI::Win::Get_group() const *(binding deprecated, see Section 15.2)* }

MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window. associated with win. The group is returned in group.

11.3 Communication Calls

MPI supports [three]five RMA communication calls: MPI_PUT transfers data from the caller memory (origin) to the target memory; MPI_GET transfers data from the target memory to the caller memory; [and] MPI_ACCUMULATE updates locations in the target memory, e.g., by adding to these locations values sent from the caller memory[.]; MPI_GET_ACCUMULATE atomically returns the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and at the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.4, page 369. Transfers can also be completed with calls to flush routines, see Section 11.6.5 for details. When a reference is made to "accumulate" operations in the following, it refers to all three operations: MPI_ACCUMULATE, MPI_GET_ACCUMULATE, and MPI_COMPARE_AND_SWAP.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call, until the [subsequent synchronization call completes.]operation completes at the origin.

[It is erroneous to have concurrent conflicting accesses to the same memory location in a window]The outcome of conflicting accesses to the same memory locations is undefined; if a location is updated by a put or accumulate operation, then [this location cannot be accessed by a load or another RMA operation]the outcome of local loads or other RMA operations is undefined until the updating operation has completed at the target. There is one exception to this rule; namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred in some**a sequential** order **from the same origin to the same destination window and memory location**. **The user can relax the ordering of such updates by using the info argument of unordered while creating the window.** In addition, [[if] a window cannot concurrently be updated by a put or accumulate operation and by a local store operation. This, even if these two updates access different locations in the window. The last restriction enables more efficient implementations of RMA operations on many systems.]the outcome of concurrent local and RMA updates to the same memory location is undefined. These restrictions are described in more detail in Section 11.7, page 385.

The calls use general datatype arguments to specify communication buffers at the origin and at the target. Thus, a transfer operation may also gather data at the source and scatter it at the destination. However, all arguments specifying both communication buffers are provided by the caller.

For all [three]five calls, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory.

> *Rationale.* The choice of supporting "self-communication" is the same as for message-passing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (*End of rationale.*)

MPI_PROC_NULL is a valid target rank in [the MPI RMA calls MPI_ACCUMULATE, MPI_GET, and MPI_PUT]all MPI RMA communication calls. The effect is the same as for MPI_PROC_NULL in MPI point-to-point communication. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch.

11.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN	origin_addr	initial address of origin buffer (choice)
IN	origin_count	number of entries in origin buffer (non-negative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	win	window object used for communication (handle)

```
int MPI_Put(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
```

```
{void MPI::Win::Put(const void* origin_addr, int origin_count,
             const MPI::Datatype& origin_datatype, int target_rank,
             MPI::Aint target_disp, int target_count,
             const MPI::Datatype& target_datatype) const
```
(binding deprecated, see Section 15.2) }

Transfers origin_count successive entries of the type specified by the origin_datatype, starting at address origin_addr on the origin node to the target node specified by the win, target_rank pair. The data are written in the target buffer at address target_addr = window_base + target_disp×disp_unit, where window_base and disp_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target_count and target_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, and comm is a communicator for the group of win.

(WDG COMMENT: Above is a bit strange as there is no tag in the rma calls.)

The communication must satisfy the same constraints as for a similar message-passing communication. The target_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window.

The target_datatype argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the target datatype object was defined at the target process by the same sequence of calls used to define it at the origin process. The target datatype must contain only relative displacements, not absolute addresses. The same holds for get and accumulate.

Advice to users. The target_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment if only portable datatypes are used (portable datatypes are defined in Section 2.4, page 11).

The performance of a put transfer can be significantly affected, on some systems, [from]by the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI_ALLOC_MEM may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This, both for debugging purposes, and for protection with client-server codes that use RMA. I.e., a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurred. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

MPI_RMA_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, rma_req)

IN	origin_addr	initial address of origin buffer (choice)
IN	origin_count	number of entries in origin buffer (non-negative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	win	window object used for communication (handle)
OUT	rma_req	RMA request (handle)

```
int MPI_RMA_put(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win,
             MPI_RMA_req rma_req)
```

```
MPI_RMA_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, RMA_REQ, IERROR
```

Similar to MPI_PUT, except that it returns a request handle that can be waited or tested on. The user can pass the value MPI_RMA_REQUEST_IGNORE, which causes MPI_RMA_PUT to behave in the same way as MPI_PUT.

11.3.2 Get

MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

OUT	origin_addr	initial address of origin buffer (choice)
IN	origin_count	number of entries in origin buffer (non-negative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	win	window object used for communication (handle)

```
int MPI_Get(void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
            TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
```

```
{void MPI::Win::Get(void *origin_addr, int origin_count,
            const MPI::Datatype& origin_datatype, int target_rank,
            MPI::Aint target_disp, int target_count,
            const MPI::Datatype& target_datatype) const (binding deprecated,
            see Section 15.2) }
```

Similar to MPI_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window, and the copied data must fit, without truncation, in the origin buffer.

MPI_RMA_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, rma_req)

OUT	origin_addr	initial address of origin buffer (choice)
IN	origin_count	number of entries in origin buffer (non-negative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	win	window object used for communication (handle)
OUT	rma_req	RMA request (handle)

```
int MPI_RMA_Get(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win,
             MPI_RMA_req rma_req)
```

```
MPI_RMA_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, RMA_REQ, IERROR
```

Similar to MPI_GET, except that it returns a request handle that can be waited or tested on. The user can pass the value MPI_RMA_REQUEST_IGNORE, which causes MPI_RMA_GET to behave in the same way as MPI_GET.

11.3.3 Examples

Example 11.1 We show how to implement the generic indirect assignment `A = B(map)`, where `A`, `B` and `map` have the same distribution, and `map` is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)

INTEGER otype(p), oindex(m),   & ! used to construct origin datatypes
     ttype(p), tindex(m),      & ! used to construct target datatypes
     count(p), total(p),       &
```

```
     win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

! This part does the work that depends on the locations of B.
! Can be reused while this does not change

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,   &
                     comm, win, ierr)

! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change

! Compute number of entries to be received from each process

DO i=1,p
  count(i) = 0
END DO
DO i=1,m
  j = map(i)/m+1
  count(j) = count(j)+1
END DO

total(1) = 0
DO i=2,p
  total(i) = total(i-1) + count(i-1)
END DO

DO i=1,p
  count(i) = 0
END DO

! compute origin and target indices of entries.
! entry i at current process is received from location
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
! j = 1..p and k = 1..m

DO i=1,m
  j = map(i)/m+1
  k = MOD(map(i),m)+1
  count(j) = count(j)+1
  oindex(total(j) + count(j)) = i
  tindex(total(j) + count(j)) = k
END DO

! create origin and target datatypes for each get operation
DO i=1,p
```

```
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, oindex(total(i)+1),   &
                                   MPI_REAL, otype(i), ierr)
  CALL MPI_TYPE_COMMIT(otype(i), ierr)
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, tindex(total(i)+1),   &
                                   MPI_REAL, ttype(i), ierr)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO

! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
  CALL MPI_TYPE_FREE(ttype(i), ierr)
END DO
RETURN
END
```

Example 11.2 A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i),m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. This will allow, for example, the accumulation of a sum by having all involved processes add their contribution to the sum variable in the memory of one process. The accumulate functions have slightly different semantics than the put and get functions; see Section 11.7 for details.

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

IN	origin_addr	initial address of buffer (choice)
IN	origin_count	number of entries in buffer (non-negative integer)
IN	origin_datatype	datatype of each buffer entry (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	op	reduce operation (handle)
IN	win	window object (handle)

```
int MPI_Accumulate(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR
```

```
{void MPI::Win::Accumulate(const void* origin_addr, int origin_count,
             const MPI::Datatype& origin_datatype, int target_rank,
             MPI::Aint target_disp, int target_count,
             const MPI::Datatype& target_datatype, const MPI::Op& op) const
```
(binding deprecated, see Section 15.2) }

Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count and origin_datatype) to the buffer specified by arguments target_count and target_datatype, at offset target_disp, in the target window specified by target_rank and win, using the operation op. This is like MPI_PUT except that data is combined into the target area instead of overwriting it.

Any of the predefined operations for MPI_REDUCE can be used. User-defined functions cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative function $f(a, b) = b$; i.e., the current value in the target memory is replaced by the value supplied by the origin.

MPI_REPLACE can be used only in MPI_ACCUMULATE[,] and MPI_GET_ACCUMULATE , and not in collective reduction operations such as MPI_REDUCE.

Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

MPI_RMA_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, rma_req)

IN	origin_addr	initial address of buffer (choice)
IN	origin_count	number of entries in buffer (non-negative integer)
IN	origin_datatype	datatype of each buffer entry (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	target_count	number of entries in target buffer (non-negative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	op	reduce operation (handle)
IN	win	window object (handle)

```
int MPI_RMA_accumulate(void *origin_addr, int origin_count,
              MPI_Datatype origin_datatype, int target_rank,
              MPI_Aint target_disp, int target_count,
              MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
              MPI_RMA_req rma_req)
```

```
MPI_RMA_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
```

```
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
TARGET_DATATYPE, OP, WIN, RMA_REQ, IERROR
```

Similar to MPI_ACCUMULATE, except that it returns a request handle that can be waited or tested on. The user can pass the value MPI_RMA_REQUEST_IGNORE, which causes MPI_RMA_ACCUMULATE to behave in the same way as MPI_ACCUMULATE.

Example 11.3 We want to compute $\mathtt{B(j)} = \sum_{\mathtt{map(i)=j}} \mathtt{A(i)}$. The arrays A, B and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i),m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL,   &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 11.2, page 366, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, then the code computes $\mathtt{B} = \mathtt{A}(\mathtt{map}^{-1})$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 11.1, page 364, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

11.3.5 Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the sent data is accumulated into the remote data, and the remote data before the accumulate is returned to the caller. The get and accumulate steps are executed atomically. MPI_REPLACE can be used to emulate fetch-and-set behavior.

(WDG COMMENT: do we want to say provide instead of emulate?)

MPI_GET_ACCUMULATE(origin_addr, result_addr, datatype, target_rank, target_disp, op, win)

IN	origin_addr	initial address of buffer (choice)
OUT	result_addr	initial address of result buffer (choice)
IN	datatype	datatype of the buffer entry (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	op	reduce operation (handle)
IN	win	window object (handle)

```
int MPI_Get_accumulate(void *origin_addr, void *result_addr,
             MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
             MPI_Op op, MPI_Win win)
```

```
MPI_GET_ACCUMULATE(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
             TARGET_DISP, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR
```

Accumulate one element of type datatype of the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation.

The datatype argument must be a predefined datatype. The operation is executed atomically.

A new predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function $f(a, b) = a$; i.e., the current value in the target memory is returned in the result buffer at the origin. MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, not in MPI_ACCUMULATE or collective reduction operations, such as MPI_REDUCE and others.

Advice to users. MPI_GET is a special case of MPI_GET_ACCUMULATE, with the operation MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

11.3.6 Compare and Swap

Another useful [functionality]ooperation is an atomic compare and swap where the value at the origin is compared bitwise to the value at the target, which is atomically replaced by a third value only if origin and target are equal.

MPI_COMPARE_AND_SWAP(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win)

IN	origin_addr	initial address of buffer (choice)
IN	compare_addr	initial address of compare buffer (choice)
OUT	result_addr	initial address of result buffer (choice)
IN	datatype	datatype of buffer entry (handle)
IN	target_rank	rank of target (non-negative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
IN	win	window object (handle)

```
int MPI_Compare_and_swap(void *origin_addr, void *compare_addr,
             void *result_addr, MPI_Datatype datatype, int target_rank,
             MPI_Aint target_disp, MPI_Win win)
```

```
MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
             TARGET_RANK, TARGET_DISP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
```

This function compares one element of type datatype in the compare buffer compare_addr with the buffer at offset target_disp, in the target window specified by target_rank and win and replaces the value at the target with the value in the origin buffer origin_addr if the compare buffer and the target compare buffer are bitwise identical. The original value at the target is returned in the buffer result_addr. The parameter datatype must be one of the following predefined datatypes: C integer, Fortran integer, Logical, Complex, Byte as specified in Section 5.9.2 on page 164, or can be of type MPI_AINT or MPI_OFFSET. Operations with overlapping types of different sizes or target displacements are erroneous.

(COMMENT: add advice to users to check consistency model in order to see which datatypes are supported fast (in hw)?) (WDG COMMENT: They should check the documentation (even though it may be wrong.))

11.4 RMA Test and Wait Functionality

MPI_RMA_WAIT(rma_req)

INOUT	rma_req	RMA request (handle)

```
int MPI_RMA_wait(MPI_RMA_req rma_req)
```

```
MPI_RMA_WAIT(RMA_REQ, IERROR)
INTEGER RMA_REQ, IERROR
```

Waits for an RMA request to complete locally (local buffer is free to be reused).

MPI_RMA_WAITALL(count, array_of_rma_reqs)

IN	count	RMA request list length
INOUT	array_of_rma_reqs	Array of RMA requests (handles)

```
int MPI_RMA_waitall(int count, MPI_RMA_req *array_of_rma_reqs)
```

```
MPI_RMA_WAITALL(COUNT, ARRAY_OF_RMA_REQS, IERROR)
    <type> ARRAY_OF_RMA_REQS(*)
    INTEGER IERROR
```

Waits for an array of RMA requests to complete locally (local buffer is free to be reused).

MPI_RMA_WAITANY(count, array_of_rma_reqs, index)

IN	count	RMA request list length
INOUT	array_of_rma_reqs	Array of RMA requests (handles)
OUT	index	index of handle for operation that completed

```
int MPI_RMA_waitany(int count, MPI_RMA_req *array_of_rma_reqs, int *index)
```

```
MPI_RMA_WAITANY(COUNT, ARRAY_OF_RMA_REQS, INDEX, IERROR)
    <type> ARRAY_OF_RMA_REQS(*), INDEX(*)
    INTEGER IERROR
```

Waits for any one RMA request in an array of RMA requests to complete locally (local buffer is free to be reused).

MPI_RMA_WAITSOME(incount, array_of_rma_reqs, outcount, array_of_indices)

IN	incount	RMA request list length
INOUT	array_of_rma_reqs	Array of RMA requests (handles)
OUT	outcount	RMA completion list length
OUT	array_of_indices	array of indices of operations that completed

```
int MPI_RMA_waitsome(int incount, MPI_RMA_req *array_of_rma_reqs,
              int *outcount, int array_of_indices[])
```

```
MPI_RMA_WAITSOME(INCOUNT, ARRAY_OF_RMA_REQS, OUTCOUNT, ARRAY_OF_INDICES,
              IERROR)
    <type> ARRAY_OF_RMA_REQS(*), INDEX(*)
    INTEGER IERROR
```

Waits for at least one RMA request in an array of RMA requests to complete locally (local buffer is free to be reused).

MPI_RMA_TEST(rma_req)

INOUT	rma_req	RMA request (handle)

```
int MPI_RMA_test(MPI_RMA_req rma_req)
```

```
MPI_RMA_TEST(RMA_REQ, IERROR)
INTEGER RMA_REQ, IERROR
```

Tests whether an RMA request has completed locally (local buffer is free to be reused).

MPI_RMA_TESTALL(count, array_of_rma_reqs)

IN	count	RMA request list length
INOUT	array_of_rma_reqs	Array of RMA requests (handles)

```
int MPI_RMA_testall(int count, MPI_RMA_req *array_of_rma_reqs)
```

```
MPI_RMA_TESTALL(COUNT, ARRAY_OF_RMA_REQS, IERROR)
    <type> ARRAY_OF_RMA_REQS(*)
    INTEGER IERROR
```

Tests whether an array of RMA requests have completed locally (local buffer is free to be reused).

MPI_RMA_TESTANY(count, array_of_rma_reqs, index)

IN	count	RMA request list length
INOUT	array_of_rma_reqs	Array of RMA requests (handles)
OUT	index	index of handle for operation that completed

```
int MPI_RMA_testany(int count, MPI_RMA_req *array_of_rma_reqs, int *index)
```

```
MPI_RMA_TESTANY(COUNT, ARRAY_OF_RMA_REQS, INDEX, IERROR)
    <type> ARRAY_OF_RMA_REQS(*), INDEX(*)
    INTEGER IERROR
```

Tests whether any one RMA request in an array of RMA requests have completed locally (local buffer is free to be reused).

MPI_RMA_TESTSOME(incount, array_of_rma_reqs, outcount, array_of_indices)

IN	incount	RMA request list length
INOUT	array_of_rma_reqs	Array of RMA requests (handles)
OUT	outcount	RMA completion list length
OUT	array_of_indices	array of indices of operations that completed

```
int MPI_RMA_testsome(int incount, MPI_RMA_req *array_of_rma_reqs,
              int *outcount, int array_of_indices[])
```

```
MPI_RMA_TESTSOME(INCOUNT, ARRAY_OF_RMA_REQS, OUTCOUNT, ARRAY_OF_INDICES,
              IERROR)
    <type> ARRAY_OF_RMA_REQS(*), INDEX(*)
    INTEGER IERROR
```

Tests whether at least one RMA request in an array of RMA requests have completed locally (local buffer is free to be reused).

11.5 Memory Model

The memory semantics of RMA is best understood by using the concept of public and private window copies. We assume that systems have a public memory region which is addressable by all processes (e.g., the shared memory in shared memory machines or the exposed main memory in distributed memory machines). In addition to this, most machines have fast private buffers (e.g., transparent caches or explicit communication buffers) local to each process where copies of data elements from the main memory can be stored for faster access. Such buffers are either coherent, i.e., all updates to main memory are reflected in all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory need to be synchronized and updated in all private copies explicitly. Coherent systems allow direct updates to remote memory without any participation of the remote side. Non-coherent systems, however, need to call RMA functions in order to reflect updates to the public window in their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two memory models called *RMA unified*, if public and private window are logically identical, and *RMA separate*, [if they remain separate]otherwise.

In the RMA separate model, there is only one instance of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.5.

In the RMA unified model, public and private copy are identical and updates via put or accumulate calls are observed by load operations without additional RMA calls. A store access to a window is immediately visible to remote get or accumulate calls. Those stronger semantics allow a programming model that is similar to shared memory.

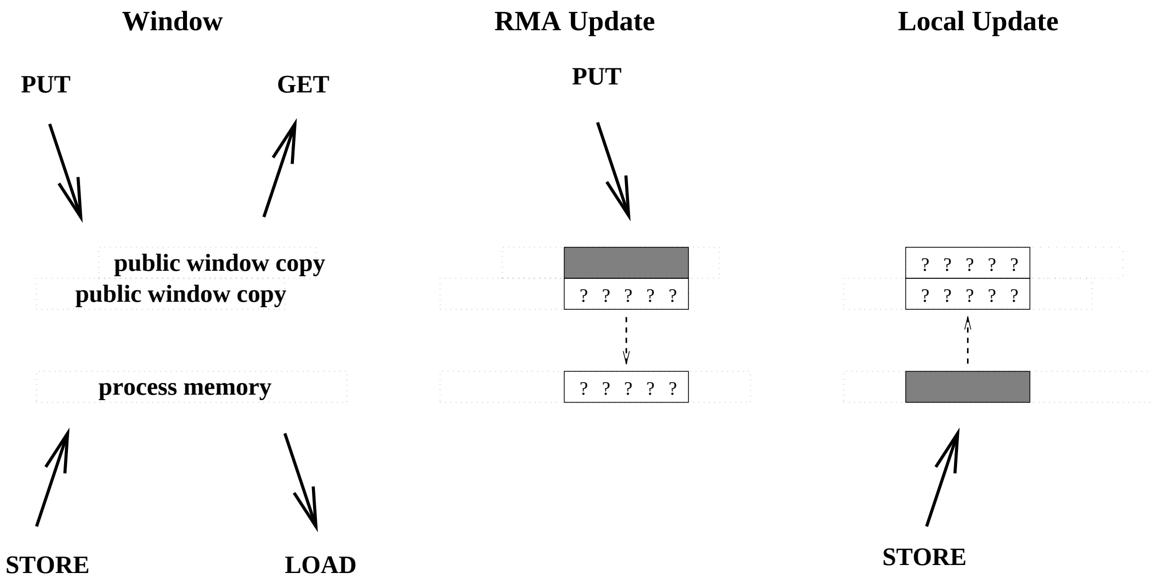

Figure 11.1: Schematic description of window (COMMENT: Rolf said this figure doesn't print well – the content is also now well described in the text and somewhat confusing)

(WDG COMMENT: What does immediately mean? Isn't a memory flush implicit here?)

> *Advice to users.* If accesses in the RMA unified model are not synchronized (with locks or flushes), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

11.5.1 Memory Model Query

RMA provides an interface to query the memory model of the underlying hardware. The RMA unified model strengthens some of the semantic guarantees of the RMA separate model and enables more flexible programming. An application can then adapt to and optimize for the underlying hardware model. This query functionality is a similar approach as used in MPI-2 for thread-safety — define several possibilities and then allow the user to both request and determine, at runtime, what level is available. This provides a way to compromise between a minimum (but universally implementable) functionality and a more powerful set of capabilities that may require additional hardware and software support from the MPI environment.

MPI_RMA_QUERY(optype, datatype, win, model)

IN	optype	operation type (integer)
IN	datatype	datatype (handle)
IN	win	window object (handle)
OUT	model	memory model (integer)

```
int MPI_RMA_query(int optype, MPI_Datatype datatype, MPI_Win win,
              int *model)
```

```
MPI_RMA_QUERY(OPTYPE, DATATYPE, WIN, MODEL, IERROR)
    INTEGER OPTYPE, DATATYPE, WIN, MODEL, IERROR
```

This call queries the memory model for a particular RMA operation and datatype. Possible returned memory models are MPI_RMA_SEPARATE and MPI_RMA_UNIFIED. MPI_RMA_SEPARATE is the weakest model and is returned if MPI_RMA_UNIFIED cannot be supported. Operation types can be either MPI_RMA_PUT, MPI_RMA_GET, MPI_RMA_ACCUMULATE, MPI_RMA_GET_ACCUMULATE, or MPI_RMA_COMPARE_AND_SWAP to query the model for each operation type separately, or MPI_RMA_EVERYTHING which returns MPI_RMA_UNIFIED only if all operation types support the RMA unified model. The datatype argument can be any MPI datatype that is allowed for the queried operation or MPI_TYPE_NULL. The call returns MPI_RMA_UNIFIED only if the datatype at either origin or target and the specified operation supports MPI_RMA_UNIFIED. If MPI_TYPE_NULL is passed as datatype, then MPI_RMA_UNIFIED is only returned if all valid datatypes for the selected operation support the RMA unified model.

The memory model indicates the relation between the public and the private view of local memory windows, see Section 11.5. The memory model is specific to an operation type.

Rationale. Different memory models can be returned for different operations types and datatypes. Some remote direct memory access hardware might offer coherent hardware-assisted MPI_PUT and MPI_GET while not supporting MPI_ACCUMULATE. Some complex datatypes might require additional (software) functionality for packing at the origin and/or unpacking at the target process. (*End of rationale.*)

(COMMENT: Brian said something about progress (in Bill's notes) and I forgot :-()

11.6 Synchronization Calls

RMA communications fall in two categories:

- **active target** communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

- **passive target** communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch. Passive target communication and the RMA unified memory model allows a synchronization mode where neither access nor exposure epochs are used and all synchronization is performed by the user.

MPI provides [three]four synchronization mechanisms:

1. 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.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI_WIN_START and is terminated by a call to MPI_WIN_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.

3. [Finally, s]Shared and exclusive locks are provided by the two functions MPI_WIN_LOCK and MPI_WIN_UNLOCK. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

   These two calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK and terminated by a call to MPI_WIN_UNLOCK. [Only one target window can be accessed during that epoch with win.]

4. Finally, a lock-free passive-target synchronization mode can be started with a call to MPI_WIN_LOCKFREE and stopped with the activation of another synchronization mode. In this mode, all synchronization is performed by remote accumulates, loads and stores, and flushes.

(COMMENT: the lock-free mode seems somewhat similar to lock/unlock with a shared lock. However, it seems that the lock-free mode is a cleaner solution because shared locks don't allow concurrent conflicting accesses (which is consistent with the literature/common sense). However, the lock-free mode needs such conflicting accesses. We also need to determine of lock-free can be implemented efficiently on non-cache coherent machines and what the granularity of access is. The second rational in Section 11.8 ("The last constraint ...") provides some points for discussion.)

Figure 11.1 illustrates the general synchronization pattern for active target communication. The synchronization between `post` and `start` ensures that the put call of the origin process does not start until the target process exposes the window (with the `post` call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between `complete` and `wait` ensures that the put call of the origin process completes before the window is unexposed (with the `wait` call). The target process will execute following local accesses to the target window only after the `wait` returned.

Figure 11.1 shows operations occurring in the natural temporal order implied by the synchronizations: the `post` occurs before the matching `start`, and `complete` occurs before the matching `wait`. However, such **strong** synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow **weak** synchronization, as illustrated in Figure 11.2. The access to the target window is delayed until the window is exposed, after the `post`. However the `start` may complete earlier; the `put` and `complete` may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.3 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The `lock` and `unlock` calls ensure that the two RMA accesses do not occur

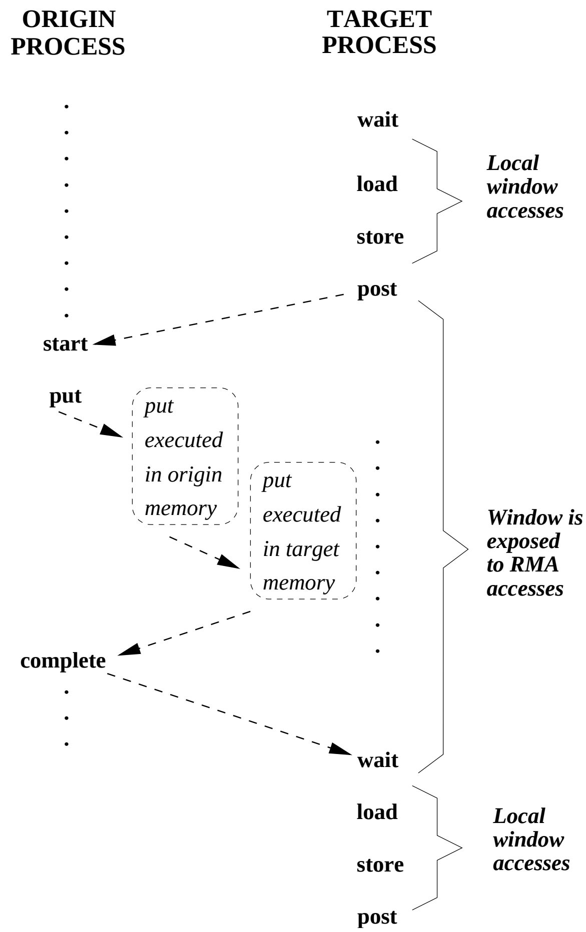

Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

concurrently. However, they do *not* ensure that the `put` by origin 1 will precede the `get` by origin 2.

(COMMENT: Some example (Figure?) for the new lockfree synch mode)

Rationale. RMA does not define fine-grained mutexes in memory (only logical coarse-grained process locks). If such semantics are needed then one can emulate mutexes or semaphores with compare and swap and accumulates. (*End of rationale.*)

11.6.1 Fence

MPI_WIN_FENCE(assert, win)

IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_fence(int assert, MPI_Win win)
```

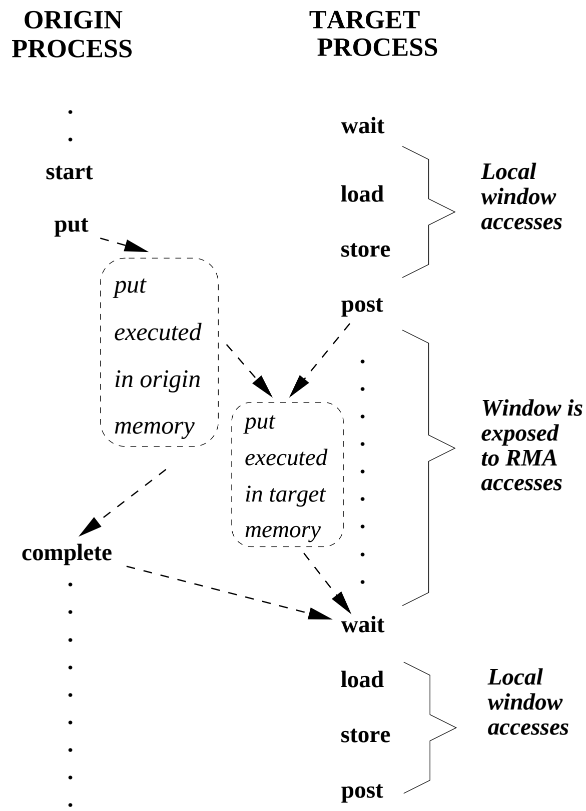

Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

```
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

{`void MPI::Win::Fence(int assert) const` *(binding deprecated, see Section 15.2)* }

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and the local process issued RMA communication calls on win between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of `post, start, complete, wait`.

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a

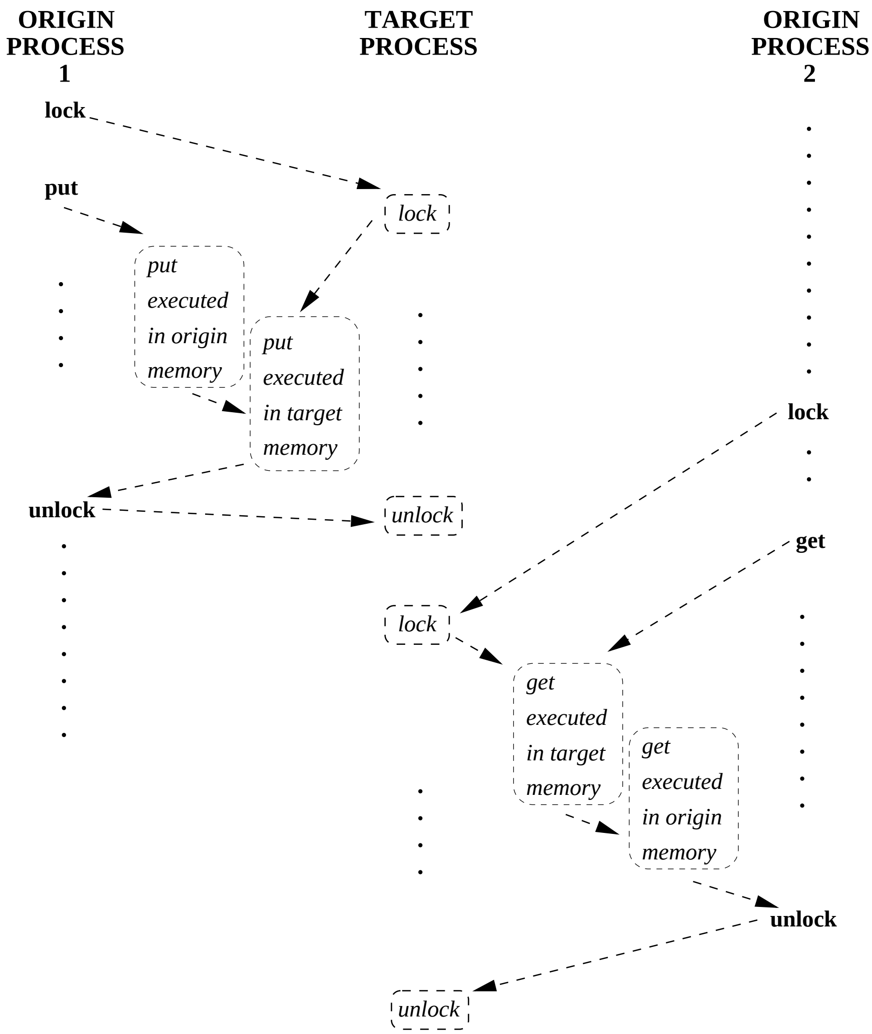

Figure 11.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

call with assert = MPI_MODE_NOPRECEDE) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.4.4. A value of assert = 0 is always valid.

Advice to users. Calls to MPI_WIN_FENCE should both precede and follow calls to put, get or accumulate that are synchronized with fence calls. (*End of advice to users.*)

11.6.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

{`void MPI::Win::Start(const MPI::Group& group, int assert) const` *(binding deprecated, see Section 15.2)* }

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.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.4.4. A value of assert = 0 is always valid.

MPI_WIN_COMPLETE(win)

IN	win	window object (handle)

```
int MPI_Win_complete(MPI_Win win)
```

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

{`void MPI::Win::Complete() const` *(binding deprecated, see Section 15.2)* }

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.

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. 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 is nonblocking, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurred; or implementations where the first two calls are nonblocking, 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 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.

MPI_WIN_POST(group, assert, win)

IN	group	group of origin processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

{`void MPI::Win::Post(const MPI::Group& group, int assert) const` *(binding deprecated, see Section 15.2)* }

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.

MPI_WIN_WAIT(win)

IN	win	window object (handle)

```
int MPI_Win_wait(MPI_Win win)
```

```
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

{`void MPI::Win::Wait() const` *(binding deprecated, see Section 15.2)* }

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.

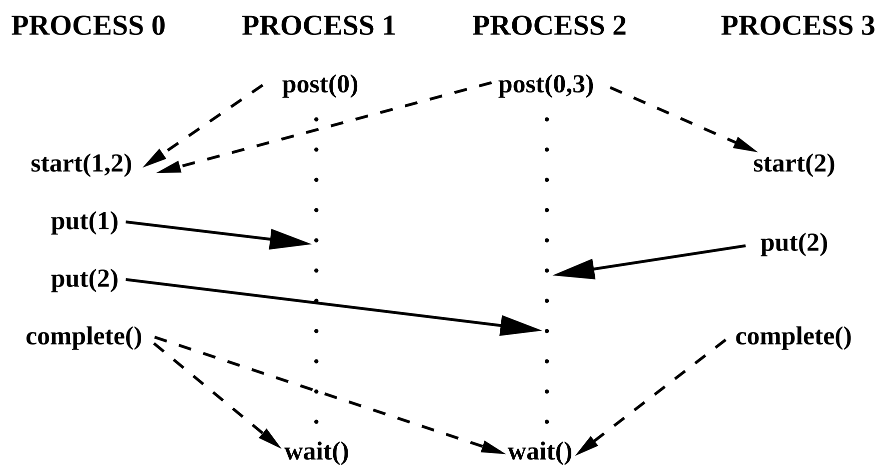

Figure 11.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

Figure 11.4 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

MPI_WIN_TEST(win, flag)

IN	win	window object (handle)
OUT	flag	success flag (logical)

```
int MPI_Win_test(MPI_Win win, int *flag)
```

```
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
```

{`bool MPI::Win::Test() const` *(binding deprecated, see Section 15.2)* }

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 effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

MPI_WIN_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process in group, using wincomm. No need to wait for the completion of these sends.

MPI_WIN_START(group,0,win) initiate a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.

MPI_WIN_COMPLETE(win) initiate a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.

MPI_WIN_WAIT(win) initiate a nonblocking receive with tag tag1 from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice[-] versa.

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each “knows” the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs, in general: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more “anonymous” communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph $G = < V, E >$, where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process i accesses the window at target process j. Then each process i issues a call to MPI_WIN_POST($ingroup_i$, ...), followed by a call to MPI_WIN_START($outgroup_i$,...), where $outgroup_i = \{j \ : \ ij \in E\}$ and $ingroup_i = \{j \ : \ ji \in E\}$. A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

11.6.3 Lock

MPI_WIN_LOCK(lock_type, rank, assert, win)

IN	lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)
IN	rank	rank of locked window (non-negative integer)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
```

```
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
```

{`void MPI::Win::Lock(int lock_type, int rank, int assert) const` *(binding deprecated, see Section 15.2)* }

Starts an RMA access epoch. Only the**The** window at the process with rank rank can be accessed by RMA operations on win during that epoch.

MPI_WIN_LOCK_WAIT(lock_type, rank, assert, win)

IN	lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)
IN	rank	rank of locked window (non-negative integer)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_lock_wait(int lock_type, int rank, int assert, MPI_Win win)
```

```
MPI_WIN_LOCK_WAIT(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
```

{`void MPI::Win::Lock_wait(int lock_type, int rank, int assert) const` *(binding deprecated, see Section 15.2)* }

Starts an RMA access epoch and waits for the epoch to be available for RMA and load/store operations.

> *Rationale.* The MPI_WIN_LOCK_WAIT function can be used on platforms where load/store operations can be performed on remote windows, such as those exposed using shared memory. (*End of rationale.*)

MPI_WIN_LOCK_ALL(assert, win)

IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_lock_all(int assert, MPI_Win win)
```

```
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

Starts a shared RMA access epoch to all processes in win. The memory on all processes in the window win can be accessed by RMA operations on win during that epoch by the calling process. A window locked with MPI_WIN_LOCK_ALL must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL refers to all members of the group of the window.

MPI_WIN_UNLOCK(rank, win)

IN	rank	rank of window (non-negative integer)
IN	win	window object (handle)

```
int MPI_Win_unlock(int rank, MPI_Win win)
```

```
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

{void MPI::Win::Unlock(int rank) const *(binding deprecated, see Section 15.2)* }

Completes an RMA access epoch started by a call to MPI_WIN_LOCK(...,win) or **MPI_WIN_LOCK_WAIT(...,win)**. RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

MPI_WIN_UNLOCK_ALL(win)

IN	win	window object (handle)

```
int MPI_Win_unlock_all(MPI_Win win)
```

```
MPI_WIN_UNLOCK_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL(assert, win). RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock call, and to protect local load/store accesses to a locked local window executed between the lock and unlock call. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. [I.e.]E.g.,, a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

> *Rationale.* An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

> *Advice to users.* Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (*End of advice to users.*)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_ALLOC_MEM (Section 8.2, page 296) MPI_WIN_ALLOCATE (Section 11.2.2, page 4), or registered with MPI_WIN_REGISTER (Section 11.2.3, page 5). Locks can be used portably only in such memory.

> *Rationale.* The implementation of passive target communication when memory is not shared [requires][might]may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for [3-rd]third party communication in shared memory machines.
>
> The downside of this decision is that passive target communication cannot be used without taking advantage of nonstandard Fortran features: namely, the availability of C-like pointers; these are not supported by some Fortran compilers[(g77 and Windows/NT compilers, at the time of writing)]. Also, passive target communication cannot be portably targeted to COMMON blocks or other statically declared Fortran arrays. (*End of rationale.*)

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. 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 call MPI_WIN_LOCK may not block, while the call to MPI_PUT blocks until a lock 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.

11.6.4 Flush and Membar

(WDG COMMENT: Should *all* of these apply only to the passive target (including lockfree) sync models?)

MPI_WIN_FLUSH(rank, win)

IN	rank	rank of target window (non-negative integer)
IN	win	window object (handle)

```
int MPI_Win_flush(int rank, MPI_Win win)
```

```
MPI_WIN_FLUSH(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling process at the specified target rank on the selected window. RMA operations issued prior to this call with rank as the target will have completed both at the origin and at the target when this call returns. This function can be called only within lock-unlock, lockall-unlockall, or lock-free epochs.

MPI_WIN_FLUSH_ALL(win)

IN	win	window object (handle)

```
int MPI_Win_flush_all(MPI_Win win)
```

```
MPI_WIN_FLUSH_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

All RMA operations issued by the calling process to any target prior to this call and in the specified window will have completed both at the origin and at the target when this call returns.

MPI_WIN_FLUSH_LOCAL(rank, win)

IN	rank	rank of target window (non-negative integer)
IN	win	window object (handle)

```
int MPI_Win_flush_local(int rank, MPI_Win win)
```

```
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

Completes all outstanding RMA operations initiated by the calling process to the target process specified by rank on the selected window locally at the origin.

MPI_WIN_FLUSH_LOCAL_ALL(win)

IN	win	window object (handle)

```
int MPI_Win_flush_local_all(MPI_Win win)
```

```
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

All RMA operations issued to any target prior to this call and in this window will have completed at the origin when this call returns.

MPI_WIN_MEMBAR(win)

IN	win	window object (handle)

```
int MPI_Win_membar(MPI_Win win)
```

```
MPI_WIN_MEMBAR(WIN, IERROR)
    INTEGER WIN, IERROR
```

MPI_WIN_MEMBAR synchronizes the private and public window copy. This function can be called only within lock-unlock, lockall-unlockall, or lock-free epochs.

11.6.5 Lockfree

MPI_WIN_LOCKFREE(ordering, assert, win)

IN	ordering	either MPI_ORDERED or MPI_UNORDERED (state)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_lockfree(int ordering, int assert, MPI_Win win)
```

```
MPI_WIN_LOCKFREE(ORDERING, ASSERT, WIN, IERROR)
    INTEGER ORDERING, ASSERT, WIN, IERROR
```

{`void MPI::Win::Lockfree(int ordering, int assert) const` *(binding deprecated, see Section 15.2)* }

The call MPI_WIN_LOCKFREE starts a lock-free access and exposure epoch. The call is collective on the group associated with window win. The user-specified message ordering defines the ordering between messages from one process to overlapping memory regions at the same target process only. MPI_ORDERED guarantees that all process observe memory updates for an origin to any target in the order they were issued at the origin. No guarantees are made for updates from different origins to overlapping memory regions at the same target. MPI_UNORDERED guarantees no ordering between accesses as in all other synchronization modes.

As opposed to other synchronization modes, the lock-free mode allows concurrent load/store and remote put, get, or accumulate accesses to the local window. The outcome of overlapping conflicting accesses without explicit synchronization (flushes) is undefined. An exception are accumulate calls that allow concurrent conflicting accesses of the same address using the same operation with the same predefined datatype on the same window. See Section 11.7 for details.

(COMMENT: Think about adding ordering as Info argument (default is ordered, strongly suggested optimization is unordered? Keep in mind that requiring an ordered implementation might be very very slow or hard to implement in shared memory. I'm not sure what the use-case for ordering would be in other synchronization modes. Maybe something in lock/unlock, not sure.)

11.6.6 Assertions

The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE and MPI_WIN_LOCK is used to provide assertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program — it is erroneous to provide[s] incorrect information. Users may always provide assert = 0 to indicate a general case where no guarantees are made.

> *Advice to users.* Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent shared memory machines. Users should consult their implementation manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

> *Advice to implementors.* Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (*End of advice to implementors.*)

assert is the bit-vector OR of zero or more of the following integer constants: MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE and MPI_MODE_NOSUCCEED. The significant options are listed below for each call.

> *Advice to users.* C/C++ users can use bit vector or (|) to combine these constants; Fortran 90 users can use the bit-vector IOR intrinsic. Fortran 77 users can use (non-portably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (*End of advice to users.*)

MPI_WIN_START:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)

MPI_MODE_UNORDERED — communication calls from the same source to the same destination memory location need not be ordered; the application will explicitly handle ordering of RMA operations through explicit synchronization.

MPI_WIN_POST:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_START have not yet occurred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.

MPI_MODE_NOSTORE — the local window was not updated by local stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.

MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

MPI_MODE_UNORDERED — communication calls from the same source to the same destination memory location need not be ordered; the application will explicitly handle ordering of RMA operations through explicit synchronization.

MPI_WIN_FENCE:

MPI_MODE_NOSTORE — the local window was not updated by local stores (or local get or receive calls) since last synchronization.

MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.

MPI_MODE_NOPRECEDE — the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.

MPI_MODE_NOSUCCEED — the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

MPI_MODE_UNORDERED — communication calls from the same source to the same destination memory location need not be ordered; the application will explicitly handle ordering of RMA operations through explicit synchronization.

MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:

MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

MPI_MODE_UNORDERED — communication calls from the same source to the same destination memory location need not be ordered; the application will explicitly handle ordering of RMA operations through explicit synchronization.

MPI_WIN_LOCKFREE:

MPI_MODE_RMA_UNIFIED — all operations that will be performed on this window will have the RMA unified memory model. No asynchronous software agent is required in this case.

MPI_MODE_UNORDERED — communication calls from the same source to the same destination memory location need not be ordered; the application will explicitly handle ordering of RMA operations through explicit synchronization.

Advice to users. Note that the nostore and noprecede flags provide information on what happened *before* the call; the noput and nosucceed flags provide information on what will happen *after* the call. (*End of advice to users.*)

11.6.7 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as argument to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message-passing, datatypes must be committed before they can be used in RMA communication.

11.7 Examples

Example 11.6 The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
...
while(!converged(A)){
  update(A);
  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                          todisp[i], 1, totype[i], win);
  MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
  }
```

The same code could be written with get[,] rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 11.7 Same generic example, with more computation/communication overlap. We assume that the update phase is broken in two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither use nor provide communicated data, is updated.

```
...
while(!converged(A)){
  update_boundary(A);
  MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                    fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
  }
```

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the `update_core` call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

Example 11.8 Same code as in Example 11.6, rewritten using post-start-complete-wait.

```
...
while(!converged(A)){
  update(A);
  MPI_Win_post(fromgroup, 0, win);
  MPI_Win_start(togroup, 0, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                         todisp[i], 1, totype[i], win);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
  }
```

Example 11.9 Same example, with split phases, as in Example 11.7.

```
...
while(!converged(A)){
  update_boundary(A);
  MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
  MPI_Win_start(fromgroup, 0, win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                   fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
  }
```

Example 11.10 A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array `A0` is updated using values of array `A1`, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window `wini` consists of array `Ai`.

```
...
if (!converged(A0,A1))
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
/* the barrier is needed because the start call inside the
loop uses the nocheck option */
while(!converged(A0, A1)){
  /* communication on A0 and computation on A1 */
  update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
  for(i=0; i < neighbors; i++)
    MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
               fromdisp0[i], 1, fromtype0[i], win0);
  update1(A1); /* local update of A1 that is
                  concurrent with communication that updates A0 */
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
  MPI_Win_complete(win0);
  MPI_Win_wait(win0);

  /* communication on A1 and computation on A0 */
  update2(A0, A1); /* local update of A0 that depends on A1 (and A0)*/
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
  for(i=0; i < neighbors; i++)
    MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                fromdisp1[i], 1, fromtype1[i], win1);
  update1(A0); /* local update of A0 that depends on A0 only,
                 concurrent with communication that updates A1 */
  if (!converged(A0,A1))
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
  MPI_Win_complete(win1);
  MPI_Win_wait(win1);
  }
```

A process posts the local window associated with `win0` before it completes RMA accesses to the remote windows associated with `win1`. When the `wait(win1)` call returns, then all neighbors of the calling process have posted the windows associated with `win0`. Conversely, when the `wait(win0)` call returns, then all neighbors of the calling process have posted the windows associated with `win1`. Therefore, the nocheck option can be used with the calls to MPI_WIN_START.

Put calls can be used, instead of get calls, if the area of array `A0` (resp. `A1`) used by the `update(A1, A0)` (resp. `update(A0, A1)`) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

11.8 Error Handling

11.8.1 Error Handlers

Errors occurring during calls to [MPI_WIN_CREATE(...,comm,...)]routines that create MPI Windows (e.g., MPI_WIN_CREATE) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.

The default error handler associated with win is MPI_ERRORS_ARE_FATAL. Users may change this default by explicitly associating a new error handler with win (see Section 8.3, page 298).

11.8.2 Error Classes

The [following]error classes for one-sided communication are defined in Table 11.1.

MPI_ERR_WIN	invalid win argument
MPI_ERR_BASE	invalid base argument
MPI_ERR_SIZE	invalid size argument
MPI_ERR_DISP	invalid disp argument
MPI_ERR_LOCKTYPE	invalid locktype argument
MPI_ERR_ASSERT	invalid assert argument
MPI_ERR_RMA_CONFLICT	conflicting accesses to window
MPI_ERR_RMA_SYNC	wrong synchronization of RMA calls
MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, target memory is not registered)

Table 11.1: Error classes in one-sided communication routines

RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP or MPI_ERR_RANK.

11.9 Semantics and Correctness

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

1. An RMA operation is completed at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_FENCE [or MPI_WIN_UNLOCK] MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the origin.

2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.

3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.

4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL, then the operation is completed at the target by that same call[to MPI_WIN_UNLOCK].

5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, [or MPI_WIN_UNLOCK]MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_MEMBAR is executed on that window by the window owner. In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls when the RMA operation completes at the target.

6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE,[or MPI_WIN_LOCK]MPI_WIN_LOCK, or MPI_WIN_LOCK_ALL is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory without additional RMA calls.

The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. [On the other hand]In the RMA separate memory model, the update of private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed in the RMA separate memory model until the process executes a suitable synchronization call while they have to complete in the RMA unified model without additional synchronization calls. Updates to a public window copy can [also] be delayed in both memory models until the window owner executes a synchronization call, if fences or post-start-complete-wait synchronization is used. [Only when lock synchronization is used does it become[s] necessary to update the public window copy, even if the window owner does not execute any related synchronization call.]If the window owner does not execute any related synchronization call in the RMA separate memory model, it becomes only necessary to update the public window copy when lock synchronization is used.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations in some situations may be *undefined*. For example, the results of performing several MPI_PUT operations to the same target location from several different origin processes within the same exposure epoch is undefined. For example, the result at the target may have all of the data from one of the MPI_PUT operations (the "last" one, in some sense), or bytes from some of each of the operations, or something else. In MPI-2, such operations were *erroneous*. That meant that an MPI implementation was permitted to signal an MPI exception. Thus, user programs or tools that used MPI RMA could not portably permit such operations, even if the application code could function correctly with such an undefined result. In MPI-3, these operations are not erroneous but do not have a defined behavior.

Rationale. As discussed in [1], requiring operations such as overlapping puts to be erroneous makes it very difficult to use MPI RMA to implement programming models, such as UPC or SHMEM, that permit these operations. Further, while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any implementation that enforced this rule, as that would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (*End of rationale.*)

Advice to implementors. Because overlapping accesses (and other operations that MPI-3 specifies) are undefined, implementations may wish to provide a mode in which such operations are erroneous to aid in debugging code. Note, however, that in MPI-3, such operations must not generate an MPI exception. (*End of advice to implementors.*)

A correct program must obey the following rules.

1. A location in a window must not be accessed locally once an update to that location has started, until the update becomes visible in the private window copy in process memory.

2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. Such accesses are allowed only in the lock-free synchronization mode. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates that use the same operation, with the same predefined datatype, on the same window.

3. A put or accumulate must not access a target window once a local update or a put or accumulate update to another (overlapping) target window have started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a local update in process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

A program [is erroneous if it violates these rules]that violates these rules has undefined behavior.

Rationale. The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were locally updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library will have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*End of rationale.*)

Advice to users. A user can write correct programs by following the following rules:

fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by local stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.

post-start-complete-wait: A window should not be updated locally while being posted, if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

lock: Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for local accesses and for RMA accesses.

lockfree: Updates to the window are either accumulates or compare and swap or protected by the user (put and get). Flushes can be used in conjunction with MPI two-sided operations for synchronization.

changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two overlapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI_WIN_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

The semantics are illustrated by the following examples:

Example 11.11 Rule 5 in the RMA separate memory model and lock synchronization:

```
Process A:                    Process B:
                              window location X

                              MPI_Win_lock(EXCLUSIVE,B)
                              store X /* local update to private copy of B */
                              MPI_Win_unlock(B)
                              /* now visible in public window copy */

MPI_Barrier                   MPI_Barrier

MPI_Win_lock(EXCLUSIVE,B)
MPI_Get(X) /* ok, read from public window */
MPI_Win_unlock(B)
```

Example 11.12 Rule 5 in the RMA unified memory model and lockless synchronization mode:

```
Process A:                    Process B:
                              window location X

                              store X /* update to private&public copy of B */
                              MPI_Win_membar
MPI_Barrier                   MPI_Barrier
MPI_Get(X) /* ok, read from window */
MPI_Win_flush_local(B)
/* read value */
```

The synchronization in this example is achieved through a combination of MPI_WIN_FLUSH_LOCAL and MPI_BARRIER.

Example 11.13 Rule 6 in the RMA separate memory model and lock synchronization:

```
Process A:                    Process B:
                              window location X

MPI_Win_lock(EXCLUSIVE,B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)

MPI_Barrier                   MPI_Barrier

                              MPI_Win_lock(EXCLUSIVE,B)
                              /* now visible in private copy of B */
                              load X
                              MPI_Win_unlock(B)
```

Note that the private copy of X has not necessarily been updated after the barrier, so omitting the lock-unlock at process B may lead to the load returning an obsolete value.

Example 11.14 Rule 6 in the RMA unified memory model and lockless synchronization:

```
Process A:                  Process B:
                            window location X

MPI_Put(X) /* update to window */
MPI_Win_flush(B)

MPI_Barrier                 MPI_Barrier
                            MPI_Win_membar
                            load X
```

Note that the private copy of X has been updated after the barrier.

In the next several examples, for conciseness, the expression

```
z = MPI_Get_accumulate(...)
```

means to perform an MPI_Get_accumulate with the result buffer (given by result_addr in the description of MPI_GET_ACCUMULATE) on the left side of the assignment; in this case, z. This format is also used with MPI_Compare_and_swap

Example 11.15 Implementing a naive, non-scalable counting semaphore in lockless synchronization mode.

```
Process A:                                  Process B:
window location X
X=2
MPI_Win_flush(A)
MPI_Barrier                                 MPI_Barrier

MPI_Accumulate(X, MPI_SUM, -1)              MPI_Accumulate(X, MPI_SUM, -1)

stack variable z                            stack variable z
while(z!=0) do                              while(z!=0) do
  z = MPI_Get_accumulate(X, MPI_NO_OP, 0)     z = MPI_Get_accumulate(X, MPI_NO_OP, 0)
  MPI_Win_flush(A)                            MPI_Win_flush(A)
done                                        done

MPI_Barrier                                 MPI_Barrier
```

Example 11.16 Implementing a critical region between two processes (Peterson's algorithm [?]) in lockless synchronization mode.

```
Process A:                                  Process B:
window location X                           window location Y
window location T
```

```
X=1                                      Y=1
MPI_Win_flush(A)                         MPI_Win_flush(B)
MPI_Barrier                              MPI_Barrier
MPI_Accumulate(T, MPI_REPLACE, 1)        MPI_Accumulate(T, MPI_REPLACE, 0)
stack variables t,y                      stack variable t,x
t=1                                      t=0
y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)    x=MPI_Get_accumulate(X, MPI_NO_OP, 0)
while(y==1 && t==1) do                   while(x==1 && t==0) do
  y=MPI_Get_accumulate(Y, MPI_NO_OP, 0)    x=MPI_Get_accumulate(X, MPI_NO_OP, 0)
  t=MPI_Get_accumulate(T, MPI_NO_OP, 0)    t=MPI_Get_accumulate(T, MPI_NO_OP, 0)
  MPI_Win_flush_all()                      MPI_Win_flush(A)
done                                     done
// critical region                      // critical region
MPI_Accumulate(X, MPI_REPLACE, 0)        MPI_Accumulate(Y, MPI_REPLACE, 0)
MPI_Win_flush(A)                         MPI_Win_flush(B)
```

Example 11.17 Implementing a critical region between n processes with compare and swap in lockless synchronization mode.

```
Process A:                               Process B...:
atomic location A
A=0
MPI_Win_flush(A)
MPI_Barrier                              MPI_Barrier
stack variable r=1                       stack variable r=1
while(r != 0) do                         while(r != 0) do
  r = MPI_Compare_and_swap(A, 0, 1)        r = MPI_Compare_and_swap(A, 0, 1)
  MPI_Win_flush(A)                         MPI_Win_flush(A)
done                                     done
// critical region                       // critical region
r = MPI_Compare_and_swap(A, 1, 0)        r = MPI_Compare_and_swap(A, 1, 0)
MPI_Win_flush(A)                         MPI_Win_flush(A)
```

Example 11.18 The rules do *not* guarantee that process A in the following sequence will see the value of X as updated by the local store by B before the lock.

```
Process A:                  Process B:
                            window location X

                            store X /* update to private copy of B */
                            MPI_Win_lock(SHARED,B)
MPI_Barrier                 MPI_Barrier

MPI_Win_lock(SHARED,B)
MPI_Get(X) /* X may not be in public window copy */
MPI_Win_unlock(B)
                            MPI_Win_unlock(B)
                            /* update on X now visible in public window */
```

Example 11.19 In the following sequence

```
Process A:                  Process B:
window location X
window location Y

store Y
MPI_Win_post(A,B) /* Y visible in public window */
MPI_Win_start(A)            MPI_Win_start(A)

store X /* update to private window */

MPI_Win_complete            MPI_Win_complete
MPI_Win_wait
/* update on X may not yet visible in public window */

MPI_Barrier                 MPI_Barrier

                            MPI_Win_lock(EXCLUSIVE,A)
                            MPI_Get(X) /* may return an obsolete value */
                            MPI_Get(Y)
                            MPI_Win_unlock(A)
```

it is *not* guaranteed that process B reads the value of X as per the local update by process A, because neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in the public window copy. To allow B to read the value of X stored by A the local store must be replaced by a local MPI_PUT that updates the public window copy. Note that by this replacement X may become visible in the private copy in process memory of A only after the MPI_WIN_WAIT call in process A. The update on Y made before the MPI_WIN_POST call is visible in the public window after the MPI_WIN_POST call and therefore correctly gotten by process B. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START operation, and process B would still get the value stored by A.

Example 11.20 Finally, in the following sequence

```
Process A:                  Process B:
                            window location X

MPI_Win_lock(EXCLUSIVE,B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)

MPI_Barrier                 MPI_Barrier

                            MPI_Win_post(B)
                            MPI_Win_start(B)

                            load X /* access to private window */
```

```
                                  /* may return an obsolete value */

                         MPI_Win_complete
                         MPI_Win_wait
```

rules (5,6) do *not* guarantee that the private copy of X at B has been updated before the load takes place. To ensure that the value put by process A is read, the local load must be replaced with a local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

11.9.1 Atomicity

The outcome of concurrent accumulates to the same location, with the same operation and predefined datatype, is as if the accumulates where done at that location in some serial order. On the other hand, if two locations are both updated by two accumulate calls, then the updates may occur in reverse order at the two locations. Thus, there is no guarantee that the entire call to MPI_ACCUMULATE is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to MPI_ACCUMULATE cannot be accessed by load or an RMA call other than accumulate until the MPI_ACCUMULATE call has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative.

11.9.2 Progress

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 local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 11.4, on page 375. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5, on page 380. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 11.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

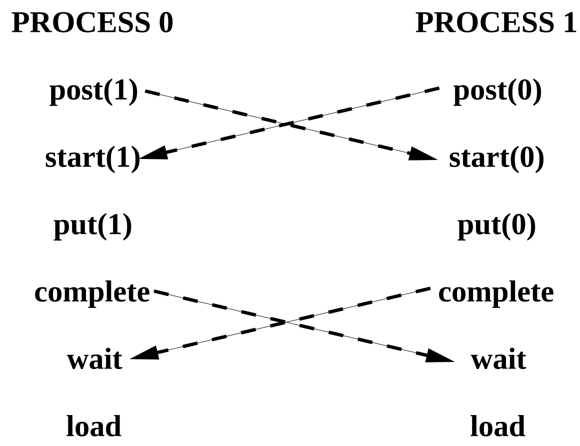

Figure 11.6: Symmetric communication

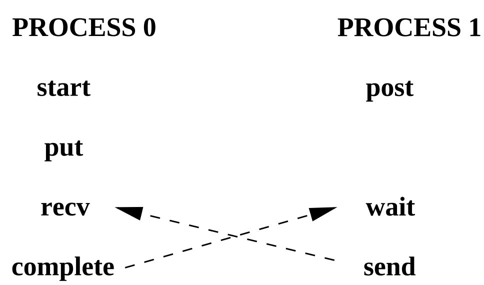

Figure 11.7: Deadlock situation

Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock, if the order of the complete and wait calls is reversed, at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice[-] versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the

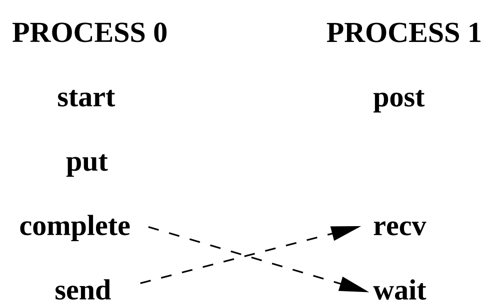

Figure 11.8: No deadlock

receive call of process 1 to complete.

Rationale. MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 11.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI [f]Forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users. (*End of rationale.*)

11.9.3 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory.

The problem is illustrated by the following code:

Source of Process 1	**Source of Process 2**	**Executed in Process 2**
`bbbb = 777`	`buff = 999`	`reg_A:=999`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
`call MPI_PUT(bbbb`		`stop appl. thread`
`into buff of process 2)`		`buff:=777 in PUT handler`
		`continue appl. thread`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
	`ccc = buff`	`ccc:=reg_A`

In this example, variable `buff` is allocated in the register `reg_A` and therefore `ccc` will have the old value of `buff` and not the new value `777`.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 16.2.2.

MPI implementations will avoid this problem for standard conforming C programs. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in COMMON blocks, or to variables that were declared VOLATILE[(while VOLATILE is not a standard Fortran declaration, it is supported by many Fortran compilers)]. Details and an additional solution are discussed in Section 16.2.2, “A Problem with Register Optimization,” on page 507. See also, “Problems Due to Data Copying and Sequence Association,” on page 504, for additional Fortran problems.

Bibliography

[1] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *IJHPCN*, 1(1/2/3):91–99, 2004. 11.9

Index