MPI: A Message-Passing Interface Standard

Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

November 25, 2019

Chapter 1

One-Sided Communications

1.1 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 to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. 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 distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. 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. The following communication calls are provided:

- Remote write: MPI_PUT, MPI_RPUT
- Remote read: MPI_GET, MPI_RGET
- Remote update: MPI_ACCUMULATE, MPI_RACCUMULATE
- Remote read and update: MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP
- Remote atomic swap operations: MPI_COMPARE_AND_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as “accumulate” operations.

MPI supports two fundamentally different *memory models*: *separate* and *unified*. The separate model makes no assumption about memory consistency and is highly portable. This model is similar to that of weakly coherent memory systems: the user must impose correct ordering of memory accesses through synchronization calls. The unified model can exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are commonly available in high-performance systems. The two different models are discussed in detail in Section 1.4. Both models support several synchronization calls to support different synchronization styles.

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

1.2 Initialization

MPI provides the following window initialization functions: MPI_WIN_CREATE, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_CREATE_DYNAMIC, which are collective on an intracommunicator. MPI_WIN_CREATE allows each process to specify 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 returns a pointer to memory allocated by the MPI implementation. MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated memory can be accessed from all processes in the window's group with direct load/store instructions. Some restrictions may apply to the specified communicator. MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control which memory is exposed by the window.

1.2.1 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	intra-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(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, INTENT(IN) ::  disp_unit
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Win), INTENT(OUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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
```

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. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section **??**). 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, rather than a basic integer type, to allow windows that span more memory than can be described with a basic integer type. (*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

latter 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 keys are predefined:

no_locks — if set to true, then the implementation may assume that passive target synchronization (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given window. 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.

accumulate_ordering — controls the ordering of accumulate operations at the target. See Section 1.7.2 for details.

accumulate_ops — if set to same_op, the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation. If set to same_op_no_op, then the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation or MPI_NO_OP. This can eliminate the need to protect access for certain operation types where the hardware can guarantee atomicity. The default is same_op_no_op.

same_size — if set to true, then the implementation may assume that the argument size is identical on all processes, and that all processes have provided this info key with the same value.

same_disp_unit — if set to true, then the implementation may assume that the argument disp_unit is identical on all processes, and that all processes have provided this info key with the same value.

Advice to users. The info query mechanism described in Section 1.2.7 can be used to query the specified info arguments for windows that have been passed to a library. It is recommended that libraries check attached info keys for each passed window. (*End of advice to users.*)

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 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 **??**) 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.

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.*)

1.2.2 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	intra-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 *baseptr, MPI_Win *win)
```

```
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    INTEGER, INTENT(IN) :: disp_unit
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    TYPE(MPI_Win), INTENT(OUT) :: win
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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 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 and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section **??** also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section **??** for an explanation of the type used for baseptr.

If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE
    SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                WIN, IERROR)
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                     WIN, IERROR)
        USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER ::  DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
        TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section **??**.

> *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. 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.

The default memory alignment requirements and the mpi_minimum_memory_alignment info key described for MPI_ALLOC_MEM in Section **??** apply to all processes with non-zero size argument. [If specified, the value of the mpi_minimum_memory_alignment info key shall be the same on all processes.]

ticket121.

1.2.3 Window That Allocates Shared Memory

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

IN	size	size of local 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	intra-communicator (handle)
OUT	baseptr	address of local allocated window segment (choice)
OUT	win	window object returned by the call (handle)

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

```
MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, INTENT(IN) ::  disp_unit
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    TYPE(MPI_Win), INTENT(OUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_ALLOCATE_SHARED(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 bytes that is shared among all processes in comm, and returns a pointer to the locally allocated segment in baseptr that can be used for load/store accesses on the calling process. The locally allocated memory can be the target of load/store accesses by remote processes; the base pointers for other processes can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a window object that can be used by all processes in comm to perform RMA operations. The size argument may be different at each process and size $= 0$ is valid. It is the user's responsibility to ensure that the communicator comm represents a group of processes that can create a shared memory segment that can be accessed by all processes in the group. The discussions of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section **??** also apply to MPI_WIN_ALLOCATE_SHARED; in particular, see the rationale in Section **??** for an explanation of the type used for baseptr. The allocated memory is contiguous across process ranks unless the info key alloc_shared_noncontig is specified. Contiguous across process ranks means that the first address in the memory segment of process i is consecutive with the last address in the memory segment of process $i - 1$. This may enable the user to calculate remote address offsets with local information only.

If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE_SHARED
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
                                       BASEPTR, WIN, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
                                            BASEPTR, WIN, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described in Section ??.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info key alloc_shared_noncontig allows the library to optimize the layout of the shared memory segments in memory.

> *Advice to users.* If the info key alloc_shared_noncontig is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (*End of advice to users.*)

> *Advice to implementors.* If the user sets the info key alloc_shared_noncontig to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (*End of advice to implementors.*)

For contiguous shared memory allocations, the default alignment requirements outlined for MPI_ALLOC_MEM in Section ?? and the mpi_minimum_memory_alignment info key apply to the start of the contiguous memory that is returned in baseptr to the first process with non-zero size argument. For noncontiguous memory allocations, the default alignment requirements and the mpi_minimum_memory_alignment info key apply to all processes with non-zero size argument. [If specified, the value of the mpi_minimum_memory_alignment info key shall be the same on all processes.] ticket121.

> *Advice to users.* If the info key alloc_shared_noncontig is not set to true (or ignored by the [MPI]MPI implementation), the alignment of the memory returned in baseptr to all but the first process with non-zero size argument depends on the value of the size argument provided by other processes. It is thus the user's responsibility to control the alignment of contiguous memory allocated for these processes by ensuring that each process provides a size argument that is an integral multiple of the alignment required for the application. (*End of advice to users.*) ticket121.

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 1.4) by utilizing the window synchronization functions (see Section 1.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

IN	win	shared memory window object (handle)
IN	rank	rank in the group of window win (non-negative integer) or MPI_PROC_NULL
OUT	size	size of the window segment (non-negative integer)
OUT	disp_unit	local unit size for displacements, in bytes (positive integer)
OUT	baseptr	address for load/store access to window segment (choice)

```
int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
             int *disp_unit, void *baseptr)
```

```
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  size
    INTEGER, INTENT(OUT) ::  disp_unit
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
    INTEGER WIN, RANK, DISP_UNIT, IERROR
    INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This function queries the process-local address for remote memory segments created with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local addresses for the same physical memory on different processes. The returned memory can be used for load/store accesses subject to the constraints defined in Section 1.7. This function can only be called with windows of flavor MPI_WIN_FLAVOR_SHARED. If the passed window is not of flavor MPI_WIN_FLAVOR_SHARED, the error MPI_ERR_RMA_FLAVOR is raised. When rank is MPI_PROC_NULL, the pointer, disp_unit, and size returned are the pointer, disp_unit, and size of the memory segment belonging the lowest rank that specified size > 0. If all processes in the group attached to the window specified size $= 0$, then the call returns size $= 0$ and a baseptr as if MPI_ALLOC_MEM was called with size $= 0$.

If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different specific procedure name:

```
INTERFACE MPI_WIN_SHARED_QUERY
    SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
                                    BASEPTR, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER WIN, RANK, DISP_UNIT, IERROR
        INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
                                         BASEPTR, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: WIN, RANK, DISP_UNIT, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in Section **??**.

1.2.4 Window of Dynamically Attached 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 MPI implementation (special steps may be taken to make one-sided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using `malloc` or `new` respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory. 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 a window that makes it possible to expose memory without remote synchronization. It must be used in combination with the local routines MPI_WIN_ATTACH and MPI_WIN_DETACH.

MPI_WIN_CREATE_DYNAMIC(info, comm, win)

IN	info	info argument (handle)
IN	comm	intra-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)
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Win), INTENT(OUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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 as described below. This routine returns a window object that can be used by these processes to perform RMA operations on attached 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.

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. For dynamic windows, the target_disp argument to RMA communication operations is not restricted to non-negative values. 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 users.* Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

> *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 (see Table **??**) is able to store addresses from any process. (*End of advice to implementors.*)

Memory at the target cannot be accessed with this window until that memory has been attached using the function MPI_WIN_ATTACH. That is, in addition to using MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use MPI_WIN_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI_WIN_ATTACH(win, base, size)

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

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

```
MPI_Win_attach(win, base, size, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
    INTEGER WIN, IERROR
    <type> BASE(*)
    INTEGER (KIND=MPI_ADDRESS_KIND) SIZE
```

Attaches a local memory region beginning at base for remote access within the given window. The memory region specified must not contain any part that is already attached to the window win, that is, attaching overlapping memory concurrently within the same window is erroneous. The argument win must be a window that was created with MPI_WIN_CREATE_DYNAMIC. The local memory region attached to the window consists of size bytes, starting at address base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see Section **??**). Multiple (but non-overlapping) memory regions may be attached to the same window.

> *Rationale.* Requiring that memory be explicitly attached before it is exposed to one-sided access by other processes can simplify implementations and improve 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.* Attaching memory to a window may require the use of scarce resources; thus, attaching large regions of memory is not recommended in portable programs. Attaching memory to a window 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 MPI_WIN_ATTACH at the target has returned before a process attempts to target that memory with an MPI RMA call.
>
> Performing an RMA operation to memory that has not been attached to 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 attaching as possible. Any limitations should be documented by the implementor. (*End of advice to implementors.*)

Attaching memory is a local operation as defined by MPI, which means that the call is not collective and completes without requiring any MPI routine to be called in any other process. Memory may be detached with the routine MPI_WIN_DETACH. After memory has been detached, it may not be the target of an MPI RMA operation on that window (unless the memory is re-attached with MPI_WIN_ATTACH).

MPI_WIN_DETACH(win, base)

IN	win	window object (handle)
IN	base	initial address of memory to be detached

```
int MPI_Win_detach(MPI_Win win, const void *base)
```

```
MPI_Win_detach(win, base, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_DETACH(WIN, BASE, IERROR)
    INTEGER WIN, IERROR
    <type> BASE(*)
```

Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.

> *Advice to users.* Detaching 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_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. (*End of advice to users.*)

Memory becomes detached when the associated dynamic memory window is freed, see Section 1.2.5.

1.2.5 Window Destruction

MPI_WIN_FREE(win)

INOUT	win	window object (handle)

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

```
MPI_Win_free(win, ierror)
    TYPE(MPI_Win), INTENT(INOUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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: e.g., 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. 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. If the window was created with MPI_WIN_ALLOCATE_SHARED, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_ALLOCATE_SHARED.

Freeing a window that was created with a call to MPI_WIN_CREATE_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls to MPI_WIN_DETACH.

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

1.2.6 Window Attributes

The following 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 the window was created.
MPI_WIN_MODEL	memory model for window.

In C, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag), MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag), MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag), and MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag) will return in base a pointer to the start of the window win, and will return in size, disp_unit, create_kind, and memory_model pointers to the size, displacement unit of the window, the kind of routine used to create the window, and the memory model, respectively. A detailed listing of the type of the pointer in the attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR is shown in Table 1.1.

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), MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror),

Attribute	**C Type**
MPI_WIN_BASE	void *
MPI_WIN_SIZE	MPI_Aint *
MPI_WIN_DISP_UNIT	int *
MPI_WIN_CREATE_FLAVOR	int *
MPI_WIN_MODEL	int *

Table 1.1: C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR.

MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in base, size, disp_unit, create_kind, and memory_model the (integer representation of) the base address, the size, the displacement unit of the window win, the kind of routine used to create the window, and the memory model, 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.
MPI_WIN_FLAVOR_SHARED	Window was created with MPI_WIN_ALLOCATE_SHARED.

The values of memory_model are MPI_WIN_SEPARATE and MPI_WIN_UNIFIED. The meaning of these is described in Section 1.4.

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 are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section ??.) The value returned for an attribute on a window is constant over the lifetime of the window.

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)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Group), INTENT(OUT) ::  group
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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.

1.2.7 Window Info

Hints specified via info (see Section ??) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or use system resources more efficiently. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_WIN_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per window basis, in window creation functions and MPI_WIN_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_WIN_SET_INFO there will be no effect on previously set or default hints that the info does not specify.

> *Advice to implementors.* It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for the hint. (*End of advice to implementors.*)

MPI_WIN_SET_INFO(win, info)

INOUT	win	window object (handle)
IN	info	info object (handle)

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

```
MPI_Win_set_info(win, info, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_SET_INFO(WIN, INFO, IERROR)
    INTEGER WIN, INFO, IERROR
```

MPI_WIN_SET_INFO updates the hints of the window associated with win using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

> *Advice to users.* Some info items that an implementation can use when it creates a window cannot easily be changed once the window has been created. Thus, an

implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether info changes were ignored by the implementation. (*End of advice to users.*)

MPI_WIN_GET_INFO(win, info_used)

IN	win	window object (handle)
OUT	info_used	new info object (handle)

```
int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
```

```
MPI_Win_get_info(win, info_used, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Info), INTENT(OUT) ::  info_used
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
    INTEGER WIN, INFO_USED, IERROR
```

MPI_WIN_GET_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints related to this window is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

1.3 Communication Calls

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET transfer data from the target memory to the caller memory; MPI_ACCUMULATE and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 1.5. Transfers can also be completed with calls to flush routines; see Section 1.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section ??.

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 operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same memory locations is undefined; if a location is updated by a put or accumulate operation, then the outcome of 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 order. In addition, the outcome of concurrent load/store and RMA updates to the same memory location is undefined. These restrictions are described in more detail in Section 1.7.

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 RMA 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 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.

1.3.1 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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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
```

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, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.

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 or in attached memory in a dynamic 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 operations.

> *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 **??**).
>
> The performance of a put transfer can be significantly affected, on some systems, 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 or MPI_WIN_ALLOCATE 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 is important both for debugging purposes and for protection with client-server codes that use RMA. That is, 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 occurs. 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.*)

1.3.2 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(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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
```

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 or within attached memory in a dynamic window, and the copied data must fit, without truncation, in the origin buffer.

1.3.3 Examples for Communication Calls

These examples show the use of the MPI_GET function. As all MPI RMA communication functions are nonblocking, they must be completed. In the following, this is accomplished with the routine MPI_WIN_FENCE, introduced in Section 1.5.

Example 1.1 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),       &
     disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint

! 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, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, 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:total(i)+count(i)), &
                                    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:total(i)+count(i)), &
                                    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)
disp_aint = 0
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 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 1.2

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 disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,  &
                    comm, win, ierr)

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

1.3.4 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 than replacing it. This will allow, for example, the accumulation of a sum by having all involved processes add their contributions to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 1.7 for details.

Accumulate Function

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 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(const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
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
```

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. The parameter 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, MPI_RACCUMULATE, MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but 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.*)

Example 1.3 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, disp_int
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
size = m * realextent
disp_int = realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  disp_aint = MOD(map(i),m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 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 1.2, except that a call to get has been replaced by a call to accumulate. (Note that, if `map` is one-to-one, the code computes $\mathtt{B} = \mathtt{A(map^{-1})}$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 1.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 1.7 for details). The predefined operation MPI_REPLACE provides fetch-and-set behavior.

MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr, result_count, result_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 origin buffer (non-negative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
OUT	result_addr	initial address of result buffer (choice)
IN	result_count	number of entries in result buffer (non-negative integer)
IN	result_datatype	datatype of each entry in result buffer (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_Get_accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, void *result_addr,
             int result_count, MPI_Datatype result_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

```
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
             result_count, result_datatype, target_rank, target_disp,
             target_count, target_datatype, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    INTEGER, INTENT(IN) ::  origin_count, result_count, target_rank,
```

```
              target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype,
              result_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

Accumulate origin_count elements of type origin_datatype from 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, specified by target_disp, target_count, and target_datatype. The data transferred from origin to target must fit, without truncation, in the target buffer. Likewise, the data copied from target to origin must fit, without truncation, in the result buffer.

The origin and result buffers (origin_addr and result_addr) must be disjoint. Each datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All 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 or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 1.7 for details.

Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE can be specified as op. User-defined functions cannot be used. 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 and no operation is performed on the target buffer. When MPI_NO_OP is specified as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored. MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others.

> *Advice to users.* MPI_GET is similar to 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.*)

Fetch and Op Function

The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetch-and-increment or fetch-and-add calls that might be supported by special hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.

MPI_FETCH_AND_OP(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 entry in origin, result, and target buffers (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_Fetch_and_op(const void *origin_addr, void *result_addr,
             MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
             MPI_Op op, MPI_Win win)
```

```
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
             target_disp, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_FETCH_AND_OP(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 DATATYPE, TARGET_RANK, OP, WIN, IERROR
```

Accumulate one element of type datatype from 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 origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be specified as op; user-defined functions cannot be used. The datatype argument must be a predefined datatype. The operation is executed atomically.

Compare and Swap Function

Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at 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 the element in all buffers (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(const void *origin_addr, const 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(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  compare_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
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 buffer are identical. The original value at the target is returned in the buffer result_addr. The parameter datatype must belong to one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, Multi-language types, or Byte as specified in Section **??**. The origin and result buffers (origin_addr and result_addr) must be disjoint.

1.3.5 Request-based RMA Communication Operations

Request-based RMA communication operations allow the user to associate a request handle with the RMA operations and test or wait for the completion of these requests using the functions described in Section ??. Request-based RMA operations are only valid within a passive target epoch (see Section 1.5).

Upon returning from a completion call in which an RMA operation completes, the MPI_ERROR field in the associated status object is set appropriately (see Section ??). All other fields of status and the results of status query functions (e.g., MPI_GET_COUNT) are undefined. It is valid to mix different request types (e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The end of the epoch, or explicit bulk synchronization using MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally.

MPI_RPUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

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	request	RMA request (handle)

```
int MPI_Rput(const 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_Request *request)
```

```
MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, request,
             ierror)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

MPI_RPUT is similar to MPI_PUT (Section 1.3.1), except that it allocates a communication request object and associates it with the request handle (the argument request). The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) indicates that the sender is now free to update the locations in the origin buffer. It does not indicate that the data is available at the target window. If remote completion is required, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL can be used.

MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

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	request	RMA request (handle)

```
int MPI_Rget(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_Request *request)
```

```
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, request,
             ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

MPI_RGET is similar to MPI_GET (Section 1.3.2), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET operation indicates that the data is available in the origin buffer. If origin_addr points to memory attached to a window, then the data becomes available in the private copy of this window.

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

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 entry in origin buffer (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)
OUT	request	RMA request (handle)

```
int MPI_Raccumulate(const 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_Request *request)
```

```
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 1.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.

MPI_RGET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr, result_count, result_datatype, target_rank, target_disp, target_count, target_datatype, op, win, request)

IN	origin_addr	initial address of 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)
OUT	result_addr	initial address of result buffer (choice)
IN	result_count	number of entries in result buffer (non-negative integer)
IN	result_datatype	datatype of each entry in result buffer (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)
OUT	request	RMA request (handle)

```
int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, void *result_addr,
             int result_count, MPI_Datatype result_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
             MPI_Request *request)
```

```
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
             result_addr, result_count, result_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
              target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
              result_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
             RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
    IERROR
```

MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 1.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

1.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and *private* window copies. We assume that systems have a public memory region that 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, 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**, 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 local 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 1.1.

In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are eventually observed by load operations without additional RMA calls. A store access to a window is eventually visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance.

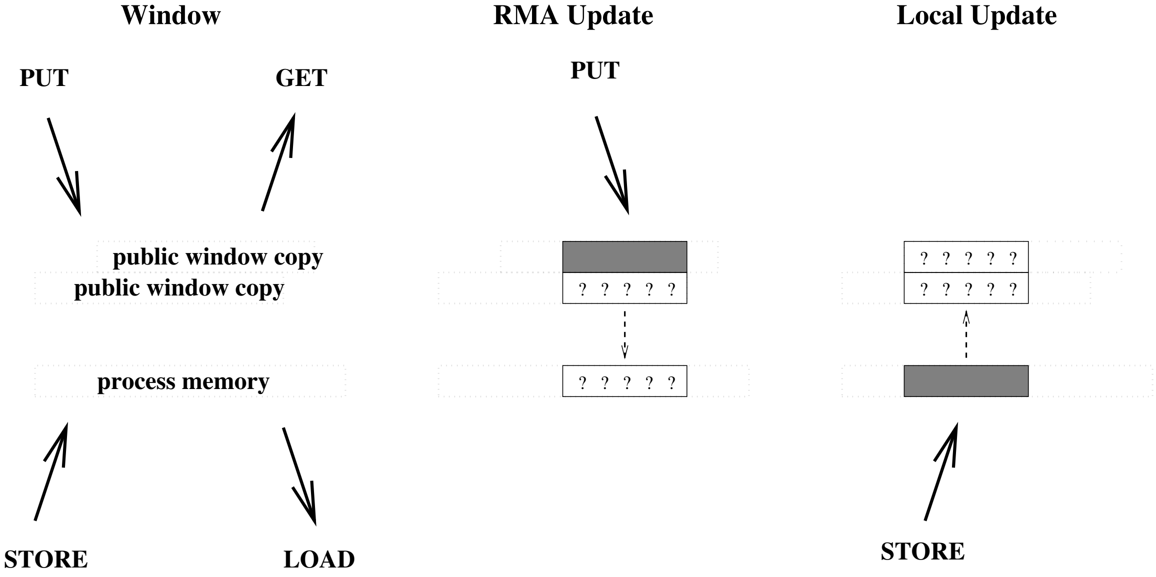

Figure 1.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows.

> *Advice to users.* If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 1.5.3), 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.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI_WIN_MODEL. If the memory model is the unified model, the value of this attribute is MPI_WIN_UNIFIED; otherwise, the value is MPI_WIN_SEPARATE.

1.5 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.

MPI provides three 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.

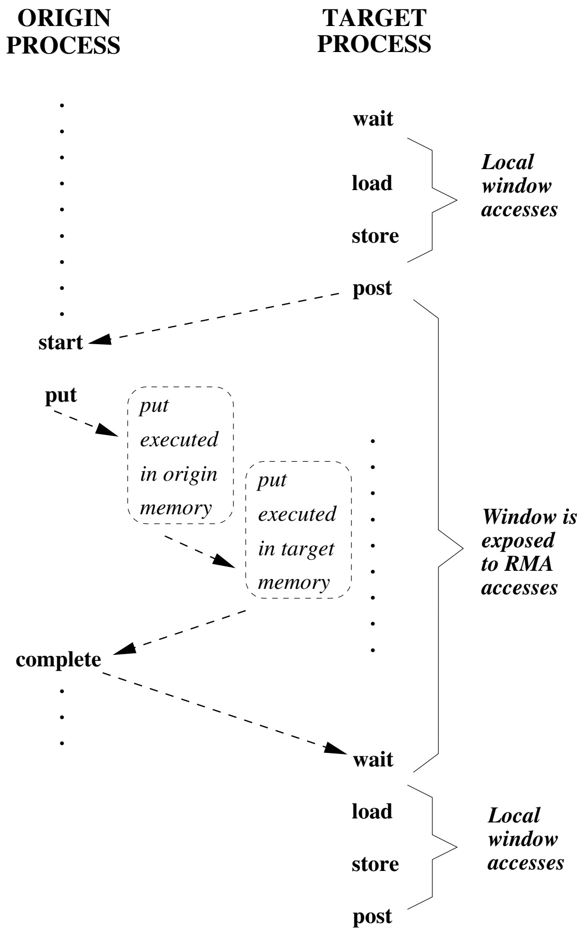

Figure 1.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

3. Finally, shared lock access is provided by the functions MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL. MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide exclusive lock capability. 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 four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively.

Figure 1.2 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.

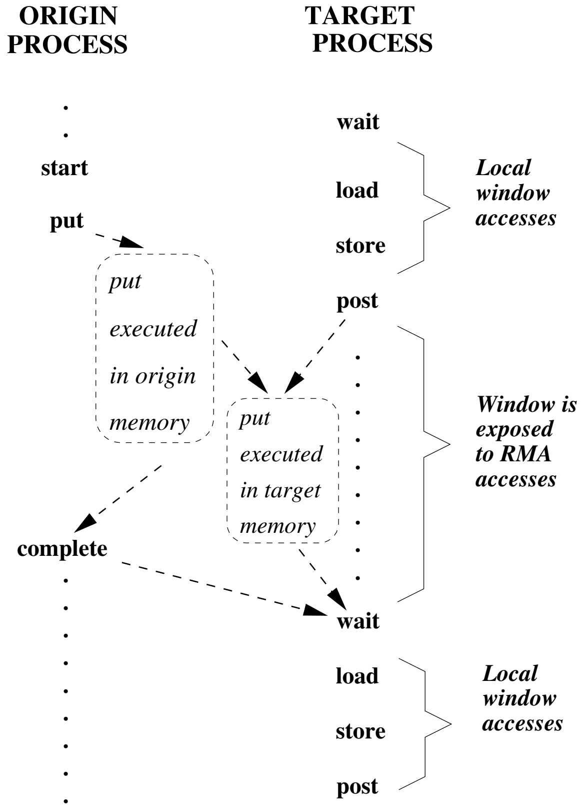

Figure 1.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

Figure 1.2 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 1.3. 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 1.4 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 concurrently. However, they do *not* ensure that the `put` by origin 1 will precede the `get` by origin 2.

> *Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarse-grained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End of rationale.*)

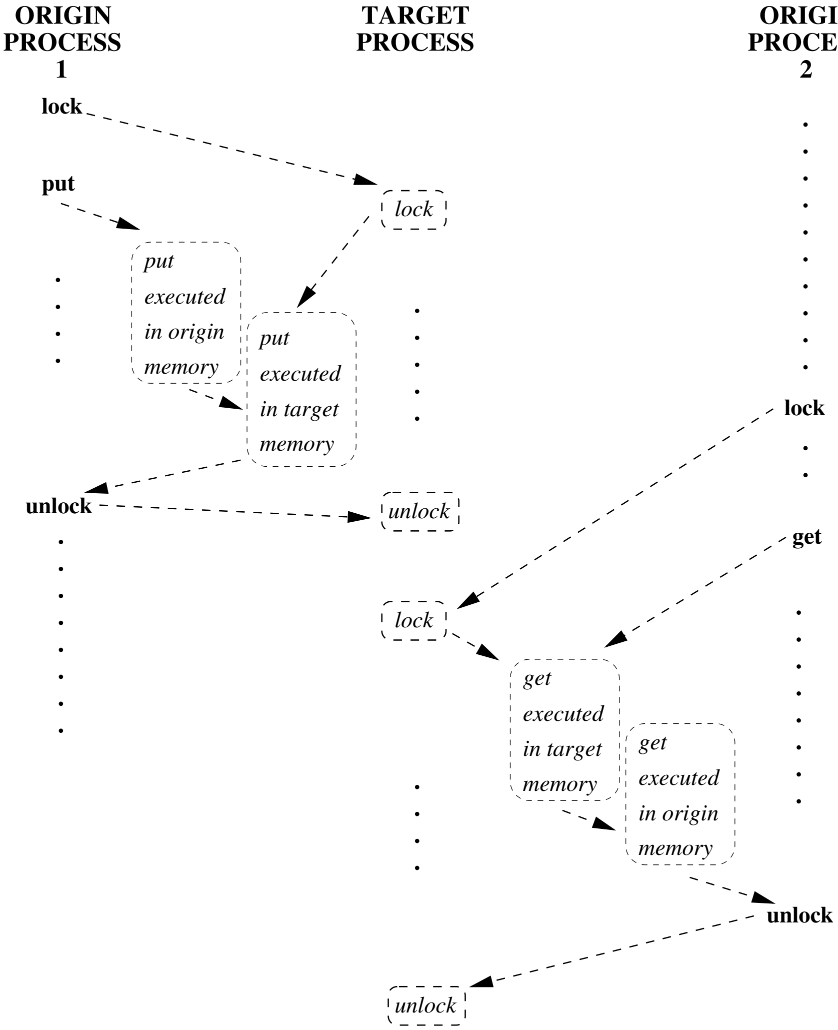

Figure 1.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

1.5.1 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)
```

```
MPI_Win_fence(assert, win, ierror)
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

```
    INTEGER ASSERT, WIN, IERROR
```

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 call with assert equal to 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 1.5.5. A value of assert = 0 is always valid.

> *Advice to users.* Calls to MPI_WIN_FENCE should both precede and follow calls to RMA communication functions that are synchronized with fence calls. (*End of advice to users.*)

1.5.2 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)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

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 1.5.5. 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)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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 1.4

```
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 occurs; 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 has called MPI_WIN_POST — the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence above must complete, without further dependencies.

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)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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 1.5 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.

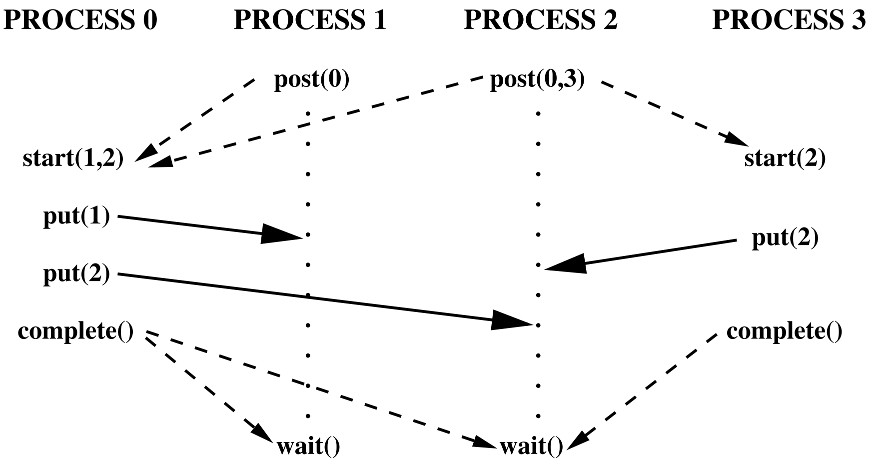

Figure 1.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

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)
    TYPE(MPI_Win), INTENT(IN) ::  win
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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 calls can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

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

MPI_WIN_START(group,0,win) initiates 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) initiates 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) initiates 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, 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 $G = \langle V, E \rangle$, 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.*)

1.5.3 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, INTENT(IN) :: lock_type, rank, assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

Starts an RMA access epoch. The window at the process with rank rank can be accessed by RMA operations on win during that epoch. Multiple RMA access epochs (with calls to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a different process.

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, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

Starts an RMA access epoch to all processes in win, with a lock type of MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on all processes in win by using RMA operations. 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 a lock on all members of the group of the window.

> *Advice to users.* There may be additional overheads associated with using MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when possible (see Section 1.5.5). (*End of advice to users.*)

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, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

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

Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window 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)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

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

Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on window win. RMA operations issued during this epoch 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 calls, and to protect load/store accesses to a locked local or shared memory window executed between the lock and unlock calls. 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. For example, 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 **??**), MPI_WIN_ALLOCATE (Section 1.2.2), MPI_WIN_ALLOCATE_SHARED (Section 1.2.3), or attached with MPI_WIN_ATTACH (Section 1.2.4). Locks can be used portably only in such memory.

> *Rationale.* The implementation of passive target communication when memory is not shared 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 third party communication in shared memory machines.
>
> (*End of rationale.*)

Consider the sequence of calls in the example below.

Example 1.5

```
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 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.

1.5.4 Flush and Sync

All flush and sync functions can be called only within passive target epochs.

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, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target.

MPI_WIN_FLUSH_ALL(win)

IN	win	window object (handle)

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

```
MPI_Win_flush_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

All RMA operations issued by the calling process to any target on the specified window 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, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

Locally completes at the origin all outstanding RMA operations initiated by the calling process to the target process specified by rank on the specified window. For example, after this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations.

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)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

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

MPI_WIN_SYNC(win)

IN	win	window object (handle)

```
int MPI_Win_sync(MPI_Win win)
```

```
MPI_Win_sync(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_WIN_SYNC(WIN, IERROR)
    INTEGER WIN, IERROR
```

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

1.5.5 Assertions

The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE, MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL 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 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's 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. 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_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 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_WIN_FENCE:

MPI_MODE_NOSTORE — the local window was not updated by 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_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.

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.*)

1.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments 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.

1.6 Error Handling

1.6.1 Error Handlers

Errors occurring during calls to routines that create MPI windows (e.g., MPI_WIN_CREATE (...,comm,...)) 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 **??**).

1.6.2 Error Classes

The error classes for one-sided communication are defined in Table 1.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP or MPI_ERR_RANK.

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	invalid 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 attached)
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource exhaustion)
MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the group of the specified communicator cannot expose shared memory)
MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called function

Table 1.2: Error classes in one-sided communication routines

1.7 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, 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.

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, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_SYNC 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.

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, MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or MPI_WIN_SYNC 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 eventually 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. In the RMA separate memory model, the update of a 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 must complete in the RMA unified model without additional synchronization calls. If fence or post-start-complete-wait synchronization is used, updates to a public window copy can be delayed in both memory models until the window owner executes a synchronization call. When passive target synchronization is used, it is necessary to update the public window copy even if the window owner does not execute any related synchronization call.

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 may be *undefined* in certain situations. For example, the result of several origin processes performing concurrent MPI_PUT operations to the same target location is undefined. In addition, the result of a single origin process performing multiple MPI_PUT operations to the same target location within the same access epoch is also undefined. The result at the target may have all of the data from one of the MPI_PUT operations (the "last" one, in some sense), 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 difficult to use MPI RMA to implement programming models—such as Unified Parallel C (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 enforces this rule, as it would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (*End of rationale.*)

> *Advice to implementors.* Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not generate an MPI exception. (*End of advice to implementors.*)

A program with a well-defined outcome in the MPI_WIN_SEPARATE memory model must obey the following rules.

S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.

S2. 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. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype, on the same window. Additional restrictions on the operation apply, see the info key accumulate_ops in Section 1.2.1.

S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to 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.

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 updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library would 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.*)

Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize the private and public window copies (that is, updates to one are made visible to the other).

In the MPI_WIN_UNIFIED memory model, the rules are simpler because the public and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with a well-defined outcome must obey in this case are:

U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.

U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (*End of advice to users.*)

U3. Updating a location in the window with a store operation that is also the target of a remote read (but not update) is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.

U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 1.2.1.

U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.

Advice to users. In the unified memory model, in the case where the window is in shared memory, MPI_WIN_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI_WIN_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 1.21. (*End of advice to users.*)

A program that violates these rules has undefined behavior.

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 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 with store operations while 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 load/store accesses and for RMA accesses.

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 or MPI_WIN_UNLOCK_ALL 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 1.6 The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK and MPI_WIN_UNLOCK calls around the store to `X` in process B are necessary to ensure consistency between the public and private copies of the window.

```
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 1.7 In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining load/stores and multi-process synchronization. Although the following example appears correct, the compiler or hardware may delay the store to `X` after the barrier, possibly resulting in the MPI_GET returning an incorrect value of `X`.

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

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

MPI_BARRIER provides process synchronization, but not memory synchronization. The example could potentially be made safe through the use of compiler- and hardware-specific notations to ensure the store to `X` occurs before process B enters the MPI_BARRIER. The use of one-sided synchronization calls, as shown in Example 1.6, also ensures the correct result.

Example 1.8 The following example demonstrates the reading of a memory location updated by a remote process (Rule 6) in the RMA separate memory model. Although the MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is necessary to synchronize the private copy with the public copy.

```
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 in this example, the barrier is not critical to the semantic correctness. The use of exclusive locks guarantees a remote process will not modify the public copy after MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation looking for changes in X on process B would be semantically correct. The barrier is required to ensure that process A performs the put operation before process B performs the load of X.

Example 1.9 Similar to Example 1.7, the following example is unsafe even in the unified model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. While Process B does not need to explicitly synchronize the public and private copies through MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the window, the scheduling of the load could result in old values of X being returned.

Compiler and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

```
Process A:                  Process B:
                            window location X
MPI_Win_lock_all
MPI_Put(X) /* update to window */
MPI_Win_flush(B)

MPI_Barrier                 MPI_Barrier
                            load X
MPI_Win_unlock_all
```

Example 1.10 The following example further clarifies Rule 5. MPI_WIN_LOCK and MPI_WIN_LOCK_ALL do *not* update the public copy of a window with changes to the private copy. Therefore, there is no guarantee that process A in the following sequence will see the value of `X` as updated by the local store by process 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 be the X before the store */
MPI_Win_unlock(B)
                            MPI_Win_unlock(B)
                            /* update on X now visible in public window */
```

The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would guarantee process A would see the updated value of `X`, as the public copy of the window would be explicitly synchronized with the private copy.

Example 1.11 Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. 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.

```
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)
```

To allow process 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 of process A only after the MPI_WIN_WAIT call in process A. The update to Y made before the MPI_WIN_POST call is visible in the public window after the MPI_WIN_POST call and therefore process B will read the proper value of Y. 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 process A.

Example 1.12 The following example demonstrates the interaction of general active target synchronization with local read operations with the RMA separate memory model. Rules 5 and 6 do *not* guarantee that the private copy of X at process B has been updated before the load takes place.

```
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
```

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.

1.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 1.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation 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 an accumulate operation cannot be accessed by a load or an RMA call other than accumulate until the accumulate operation 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. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

1.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. MPI specifies ordering between accumulate operations from one process to the same (or overlapping) memory locations at another process on a per-datatype granularity. The default ordering is strict ordering, which guarantees that overlapping updates from the same source to a remote location are committed in program order and that reads (e.g., with MPI_GET_ACCUMULATE) and writes (e.g., with MPI_ACCUMULATE) are executed and committed in program order. Ordering only applies to operations originating at the same origin that access overlapping target memory regions. MPI does not provide any guarantees for accesses or updates from different origin processes to overlapping target memory regions.

The default strict ordering may incur a significant performance penalty. MPI specifies the info key accumulate_ordering to allow relaxation of the ordering semantics when specified to any window creation function. The values for this key are as follows. If set to none, then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in MPI-2 but is *not* the default in MPI-3. The key can be set to a comma-separated list of required access orderings at the target. Allowed values in the comma-separated list are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-after-write ordering, respectively. These indicate whether operations of the specified type complete in the order they were issued. For example, raw means that any writes must complete at the target before subsequent reads. These ordering requirements apply only to operations issued by the same origin process and targeting the same target process. The default value for accumulate_ordering is rar,raw,war,waw, which implies that writes complete at the target in the order in which they were issued, reads complete at the target before any writes that are issued after the reads, and writes complete at the target before any reads that are issued after the writes. Any subset of these four orderings can be specified. For example, if only read-after-read and write-after-write ordering is required, then the value of the accumulate_ordering key could be set to rar,waw. The order of values is not significant.

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

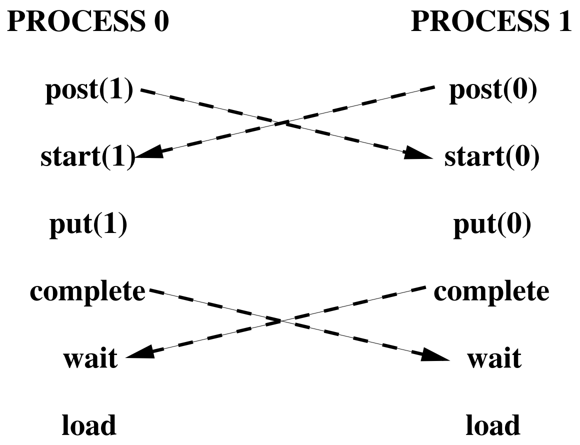

Figure 1.6: Symmetric communication

1.7.3 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 1.4. 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 1.5. 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 1.6. 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.

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 1.7. This code will deadlock: the wait

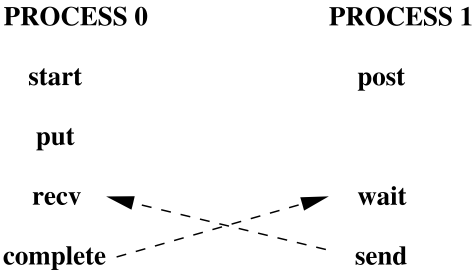

Figure 1.7: Deadlock situation

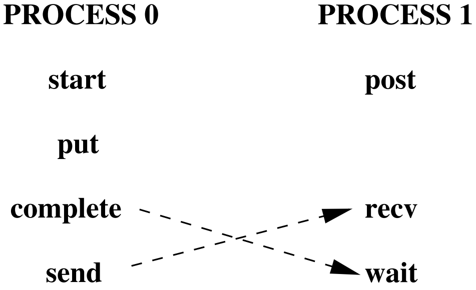

Figure 1.8: No deadlock

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 1.8. 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 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 1.8, 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, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (*End of rationale.*)

1.7.4 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 values 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. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

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 **??**.

Programs written in C avoid this problem, because of the semantics of C. 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 modules or `COMMON` blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections **??**–**??**. Sections **??** to **??** discuss several solutions for the problem in this example.

1.8 Examples

Example 1.13 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 1.14 Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides 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 1.15 Same code as in Example 1.13, 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 1.16 Same example, with split phases, as in Example 1.14.

```
...
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 1.17 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 < fromneighbors; 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 < fromneighbors; 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.

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 1.18 The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE are used to write to or read from the local public copy.

```
Process A:                                Process B:
MPI_Win_lock_all                          MPI_Win_lock_all
window location X
X=2
MPI_Win_sync
MPI_Barrier                               MPI_Barrier

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

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

MPI_Win_unlock_all                        MPI_Win_unlock_all
```

Example 1.19 Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

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

MPI_Win_lock_all                      MPI_Win_lock_all
X=1                                   Y=1
MPI_Win_sync                          MPI_Win_sync
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,               x=MPI_Get_accumulate(X,
   MPI_NO_OP, 0)                         MPI_NO_OP, 0)
while(y==1 && t==1) do                while(x==1 && t==0) do
  y=MPI_Get_accumulate(Y,               x=MPI_Get_accumulate(X,
     MPI_NO_OP, 0)                         MPI_NO_OP, 0)
  t=MPI_Get_accumulate(T,               t=MPI_Get_accumulate(T,
     MPI_NO_OP, 0)                         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_unlock_all                    MPI_Win_unlock_all
```

Example 1.20 Implementing a critical region between multiple processes with compare and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization of `A` to guarantee the public copy has been updated with the initialization value found in the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure `A` in the public copy of Process A had been updated before the barrier.

```
Process A:                             Process B...:
MPI_Win_lock_all                       MPI_Win_lock_all
atomic location A
A=0
MPI_Win_sync
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_unlock_all                     MPI_Win_unlock_all
```

Example 1.21 The following example demonstrates the proper synchronization in the unified memory model when a data transfer is implemented with load and store in the case of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if `MPI_ASYNC_PROTECTS_NONBLOCKING` is `.FALSE.` or the variable `X` is not declared as `ASYNCHRONOUS`, reordering of the accesses to the variable `X` must be prevented with MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)

The variable `X` is contained within a shared memory window and `X` corresponds to the same memory location at both processes. The MPI_WIN_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI_WIN_SYNC operation performed by process B ensures that process A's updates to `X` are visible to process B.

```
Process A                        Process B

MPI_WIN_LOCK_ALL(                MPI_WIN_LOCK_ALL(
      MPI_MODE_NOCHECK,win)            MPI_MODE_NOCHECK,win)

DO ...                           DO ...
  X=...

  MPI_F_SYNC_REG(X)
  MPI_WIN_SYNC(win)
  MPI_SEND                         MPI_RECV
                                   MPI_WIN_SYNC(win)
                                   MPI_F_SYNC_REG(X)

                                   print X

                                   MPI_F_SYNC_REG(X)
  MPI_RECV                         MPI_SEND
  MPI_F_SYNC_REG(X)
END DO                           END DO

MPI_WIN_UNLOCK_ALL(win)          MPI_WIN_UNLOCK_ALL(win)
```

Example 1.22 The following example shows how request-based operations can be used to overlap communication with computation. Each process fetches, processes, and writes the result for `NSTEPS` chunks of data. Instead of a single buffer, `M` local buffers are used to allow up to `M` communication operations to overlap with computation.

```
int         i, j;
MPI_Win     win;
MPI_Request put_req[M] = { MPI_REQUEST_NULL };
MPI_Request get_req;
double      *baseptr;
double      data[M][N];
```

```
MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
  MPI_COMM_WORLD, &baseptr, &win);

MPI_Win_lock_all(0, win);

for (i = 0; i < NSTEPS; i++) {
 if (i<M)
   j=i;
 else
   MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);

 MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
          &get_req);
 MPI_Wait(&get_req,MPI_STATUS_IGNORE);
 compute(i, data[j], ...);
 MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
          &put_req[j]);
}

MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
MPI_Win_unlock_all(win);
```

Example 1.23 The following example constructs a distributed shared linked list using dynamic windows. Initially process 0 creates the head of the list, attaches it to the window, and broadcasts the pointer to all processes. All processes then concurrently append `N` new elements to the list. When a process attempts to attach its element to the tail of the list it may discover that its tail pointer is stale and it must chase ahead to the new tail before the element can be attached. This example requires some modification to work in an environment where the layout of the structures is different on different processes.

```
...
#define NUM_ELEMS 10

#define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
                               offsetof(llist_ptr_t, rank) )
#define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
                               offsetof(llist_ptr_t, disp) )

/* Linked list pointer */
typedef struct {
  MPI_Aint disp;
  int      rank;
} llist_ptr_t;

/* Linked list element */
typedef struct {
  llist_ptr_t next;
```

```
  int value;
} llist_elem_t;

const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };

/* List of locally allocated list elements. */
static llist_elem_t **my_elems = NULL;
static int my_elems_size  = 0;
static int my_elems_count = 0;

/* Allocate a new shared linked list element */
MPI_Aint alloc_elem(int value, MPI_Win win) {
  MPI_Aint disp;
  llist_elem_t *elem_ptr;

  /* Allocate the new element and register it with the window */
  MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
  elem_ptr->value = value;
  elem_ptr->next  = nil;
  MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));

  /* Add the element to the list of local elements so we can free
     it later. */
  if (my_elems_size == my_elems_count) {
    my_elems_size += 100;
    my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
  }
  my_elems[my_elems_count] = elem_ptr;
  my_elems_count++;

  MPI_Get_address(elem_ptr, &disp);
  return disp;
}

int main(int argc, char *argv[]) {
  int           procid, nproc, i;
  MPI_Win       llist_win;
  llist_ptr_t   head_ptr, tail_ptr;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &procid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);

  /* Process 0 creates the head node */
  if (procid == 0)
```

```
    head_ptr.disp = alloc_elem(-1, llist_win);

  /* Broadcast the head pointer to everyone */
  head_ptr.rank = 0;
  MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
  tail_ptr = head_ptr;

  /* Lock the window for shared access to all targets */
  MPI_Win_lock_all(0, llist_win);

  /* All processes concurrently append NUM_ELEMS elements to the list */
  for (i = 0; i < NUM_ELEMS; i++) {
    llist_ptr_t new_elem_ptr;
    int success;

    /* Create a new list element and attach it to the window */
    new_elem_ptr.rank = procid;
    new_elem_ptr.disp = alloc_elem(procid, llist_win);

    /* Append the new node to the list.  This might take multiple
       attempts if others have already appended and our tail pointer
       is stale. */
    do {
      llist_ptr_t next_tail_ptr = nil;

      MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
          (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
          MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
          llist_win);

      MPI_Win_flush(tail_ptr.rank, llist_win);
      success = (next_tail_ptr.rank == nil.rank);

      if (success) {
        MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
            MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
            MPI_AINT, MPI_REPLACE, llist_win);

        MPI_Win_flush(tail_ptr.rank, llist_win);
        tail_ptr = new_elem_ptr;

      } else {
        /* Tail pointer is stale, fetch the displacement.  May take
           multiple tries if it is being updated. */
        do {
          MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
              1, MPI_AINT, tail_ptr.rank,
              MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
```

```
              1, MPI_AINT, MPI_NO_OP, llist_win);

          MPI_Win_flush(tail_ptr.rank, llist_win);
        } while (next_tail_ptr.disp == nil.disp);
        tail_ptr = next_tail_ptr;
      }
    } while (!success);
  }

  MPI_Win_unlock_all(llist_win);
  MPI_Barrier(MPI_COMM_WORLD);

  /* Free all the elements in the list */
  for ( ; my_elems_count > 0; my_elems_count--) {
    MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
    MPI_Free_mem(my_elems[my_elems_count-1]);
  }
  MPI_Win_free(&llist_win);
...
```

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. 1.7