Remote Memory Access Programming in MPI-3

Torsten Hoefler, ETH Zurich
James Dinan, Argonne National Laboratory
Rajeev Thakur, Argonne National Laboratory
Brian Barrett, Sandia National Laboratories
Pavan Balaji, Argonne National Laboratory
William Gropp, University of Illinois at Urbana-Champaign
Keith Underwood, Intel Inc.

The Message Passing Interface (MPI) 3.0 standard, introduced in September 2012, includes a significant update to the one-sided communication interface, also known as remote memory access (RMA). In particular, the interface has been extended to better support popular one-sided and global-address-space parallel programming models, to provide better access to hardware performance features, and to enable new data-access modes. We present the new RMA interface and extract formal models for data consistency and access semantics. Such models are important for users, enabling them to reason about data consistency, and for tools and compilers, enabling them to automatically analyze, optimize, and debug RMA operations.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*

General Terms: Design, Performance

Additional Key Words and Phrases: MPI, One-sided communication, RMA

ACM Reference Format:
T. Hoefler et al., 2013. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* 1, 1, Article 1 (March 2013), 29 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. MOTIVATION

Parallel programming models can be split into three categories: (1) shared memory with implicit communication and explicit synchronization, (2) message passing with explicit communication and implicit synchronization (as a side effect of communication), and (3) remote memory access and partitioned global address space (PGAS) where synchronization and communication are managed independently.

At the hardware side, high-performance networking technologies have converged toward remote direct memory access (RDMA) because it offers the highest performance (operating system bypass [Shivam et al. 2001]) and is relatively easy to implement. Thus, current high-performance networks, such as Cray's Gemini and Aries, IBM's PERCS and BG/Q networks, InfiniBand, and Ethernet (using RoCE), all offer RDMA functionality.

This work is partially supported by the National Science Foundation, under grants #CCF-0816909 and #CCF-1144042, and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award number DE-FC02-10ER26011 and contract DE-AC02-06CH11357.

DOI: http://dx.doi.org/10.1145/0000000.0000000

Shared memory often cannot be emulated efficiently on distributed-memory machines [Karlsson and Brorsson 1998], and message passing incurs additional overheads on RDMA networks. Implementing fast message-passing libraries over RDMA usually requires different protocols [Woodall et al. 2006]: an eager protocol with receiver-side buffering of small messages and a rendezvous protocol that synchronizes the sender. Eager delivery requires additional copies, and the rendezvous protocol sends additional control messages and may delay the sending process. The PGAS model thus remains a good candidate for directly exploiting the power of RDMA networking.

High-performance computing (HPC) has long worked within the message-passing paradigm, where the only means of communication across process boundaries is to exchange messages. MPI-2 introduced a one-sided communication scheme, but for a variety of reasons it was not widely used. However, architectural trends, such as RDMA networks and the increasing number of (potentially noncoherent) cores on each node, necessitated a reconsideration of the programming model.

The Message Passing Interface Forum, the standardization body for the MPI standard, developed new ways for exploiting RDMA networks and multicore CPUs in MPI programs. We summarize here the new one-sided communication interface of MPI-3 [MPI Forum 2012], define the memory semantics in a semi-formal way, and demonstrate techniques for reasoning about correctness and performance of one-sided programs.

This paper, written by key members of the MPI-3 Remote Memory Access (RMA) working group, is targeted at advanced programmers who want to understand the detailed semantics of MPI-3 RMA programming, designers of libraries or domain-specific languages on top of MPI-3, researchers thinking about future RMA programming models, and tool and compiler developers who aim to support RMA programming. For example, a language developer could base semantics of the language on the underlying MPI RMA semantics; a tool developer could use the semantics specified in this paper to develop static-analysis and model-checking tools that reason about the correctness of MPI RMA programs; and a compiler developer could design analysis and transformation passes to optimize MPI RMA programs transparently to the user.

1.1. Related Work

Efforts in the area of parallel programming models are manifold. PGAS programming views the union of all local memory as a globally addressable unit. The two most prominent languages in the HPC arena are Co-Array Fortran (CAF [Numrich and Reid 1998]), now integrated into the Fortran standard as coarrays, and Unified Parallel C (UPC [UPC Consortium 2005]). CAF and UPC simply offer a two-level view of local and remote memory accesses. Indeed, CAF-2 [Mellor-Crummey et al. 2009] proposed the notion of teams, a concept similar to MPI communicators, but it has not yet been widely adopted. Higher-level PGAS languages, such as X10 [Charles et al. 2005] and Chapel [Chamberlain et al. 2007], offer convenient programmer abstractions and elegant program design but have yet to deliver the performance necessary in the HPC context. Domain-specific languages, such as Global Arrays [Nieplocha et al. 1996], offer similar semantics restricted to specific contexts (in this case array accesses). MPI-2's RMA model [MPI Forum 2009, §11] is the direct predecessor to MPI-3's RMA model, and indeed MPI-3 is fully backward compatible. However, MPI-3 defines a completely new memory model and access mode that can rely on hardware coherence instead of MPI-2's expensive and limited software-coherence mechanisms.

In general, the MPI-3 approach integrates easily into existing infrastructures since it is a library interface that can work with all compilers. A complete specification of the library semantics enables automated compiler transformations [Danalis et al. 2009],

for example, for parallel languages such as UPC or CAF. In addition, MPI offers a rich set of semantic concepts such as isolated program groups (communicators), process topologies, and runtime-static abstract definitions for access patterns of communication functions (MPI datatypes). Those concepts allow users to specify additional properties of their code that allow more complex optimizations at the library and compiler level. In addition, communicators and process topologies [Traff 2002; Hoefler et al. 2011] can be used to optimize process locality during runtime. Another major strength of the MPI concepts is the strong abstraction and isolation principles that allow the layered implementation of libraries on top of MPI [Hoefler and Snir 2011].

Since MPI RMA offers direct memory access to local and remote memory for multiple threads of execution (MPI processes), questions related to memory consistency and memory models arise. Several recent works deal with understanding complex memory models of architectures such as x86 [Owens et al. 2009] and specifications for programming languages such as Java [Manson et al. 2005] and C++11 [Boehm and Adve 2008]. We will build on the models and notations developed in those papers and define memory semantics for MPI RMA. The well-known paper demonstrating that threads cannot be implemented with a library interface [Boehm 2005] also applies to this discussion. Indeed, serial code optimization mixed with parallel executing schedule may lead to erroneous or slower codes. In this work, we define a set of restrictions for serial compilers to make them MPI-aware.

1.2. Contributions of This Work

The specific contributions of this work are as follows.

(1) Proposal of new semantics for a library interface enabling remote memory programming
(2) Description of the driving forces behind the MPI-3 RMA standardization far beyond the actual standard text
(3) Considerations for optimized implementations on different target architectures
(4) Analysis of common use cases and examples

2. OVERVIEW AND CHALLENGES OF RMA PROGRAMMING

The main complications for remote memory access programming arise from the separation of communication (remote accesses) and synchronization. In addition, the MPI interface splits synchronization further into memory synchronization or consistency (i.e., a remote process can observe a communicated value with a local read) and process synchronization (i.e., when a remote process gathers knowledge about the state of a peer process). Furthermore, such synchronization can be nonblocking.

The main challenges of RMA programming revolve around the semantics of operation completion and memory consistency. Most programming systems offer some kind of weak or relaxed consistency because sequential consistency is too expensive to implement. However, most programmers prefer to reason in terms of sequential consistency because of its conceptual simplicity. C++11 and Java offer sequential consistency at the language level if the programmer follows certain rules (i.e., avoids data races). While Java attempts to define the behavior of programs containing races, C++11 leaves the topic unspecified.

MPI models consistency, completion, and synchronization as separate concepts and allow the user to reason about them separately. RMA programming is thus slightly more complex because of complex interactions of operations. For example, MPI, like most RMA programming models, allows the programmer to start operations asynchronously and complete them (locally or remotely) later. This technique is necessary to hide single-message latency with multiple pipelined messages; however, it makes

reasoning about program semantics much more complex. In the MPI RMA model, all communication operations are nonblocking; in other words, the communication functions may return before the operation completes, and bulk synchronization functions are used to complete previously issued operations. In the ideal case, this feature enables a programming model in which high latencies can be ignored and processes never "wait" for remote completion.

The resulting complex programming environment is often not suitable for average programmers (i.e., domain scientists); rather, writers of high-level libraries can provide domain-specific extensions that hide most of the complexity. The MPI RMA interface enables expert programmers and implementers of domain-specific libraries and languages to extract the highest performance from a large number of computer architectures in a performance-portable way.

3. SEMANTICS AND ARCHITECTURAL CONSIDERATIONS

In this section, we discuss the specific concepts that we use in MPI RMA programming to enable performance-portable and composable software development.

The two central concepts of MPI RMA are memory regions and process groups. Both concepts are attached to an object called the MPI window. A memory region is a consecutive area in the main memory of each process in a group that is accessible to all other processes in the group. This enables two types of spatial isolation: (1) no process outside the group may access any exposed memory, and (2) memory that is not attached to an MPI window cannot be accessed by any process, even in the correct group. Both principles are important for parallel software engineering. They simplify the development and maintenance process by offering an additional separation of concerns; that is, nonexposed memory cannot be corrupted by remote processes. They also allow the development of spatially separated libraries in that a library can use either a dedicated set of processes or a separate memory region and thus does not interfere with other libraries and user code [Hoefler and Snir 2011].

MPI RMA offers the basic data-movement operations put and get and additional predefined remote atomic operations called accumulates. Put and get are designed to enable direct usage of the shared-memory subsystem or hardware-enabled RDMA. Accumulates can, in some cases, also use functions that are hardware-accelerated.

All such communication functions are nonblocking. Communication functions are completed by using either bulk-completion functions (all synchronization functions include bulk completion as a side-effect) or single-completion (if special, generally slower, MPI-request-based communication operations are used). Figure 1 shows an overview of communication options in the MPI specification.

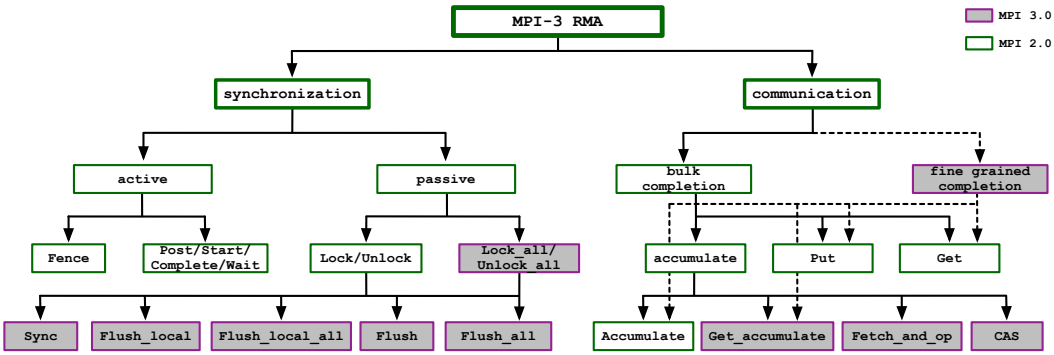


Fig. 1. Overview of communication options in the MPI-3 specification.

3.1. Memory Exposure

MPI RMA offers four calls to expose local memory to remote processes. The first three variants create windows that can be remotely accessed only by MPI communication operations. Figure 2 shows an overview of the different versions. The last variant enables users to exploit shared-memory semantics directly and provides direct load/store access to remote window memory if supported by the underlying architecture.

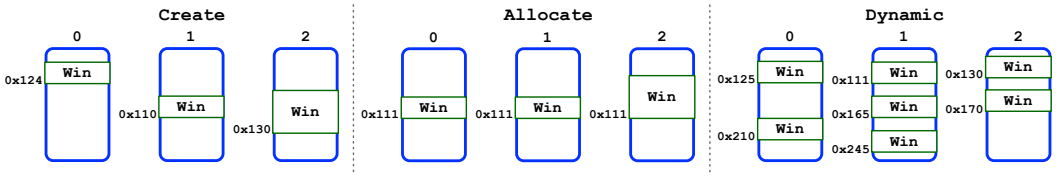


Fig. 2. MPI-3 memory window creation variants.

The first (legacy) variant is the normal *win create* function. Here, each process can specify an arbitrary amount (≥ 0 bytes) of local memory to be exposed and a communicator identifying a process group for remote access. The function returns a window object that can be used later for remote accesses. Remote accesses are always relative to the start of the window at the target process, so a put to offset zero at process k updates the first memory block in the window that process k exposed. MPI allows the user to specify the block size for addressing each window (called displacement unit). The fact that processes can specify arbitrary memory may lead to large translation tables on systems that offer RDMA functions.

The second creation function, *win allocate*, transfers the responsibility for memory allocation to MPI. RDMA networks that require large translation tables for *win create* may be able to avoid such tables by allocating memory at identical addresses on all processes. Otherwise, the semantics are identical to the traditional creation function.

The third creation function, *create dynamic*, does not bind memory to the created window. Instead, it binds only a process group where each process can use subsequent local functions for adding (exposing) memory to the window. This mode naturally maps to many RDMA network architectures; however, it may be more expensive than allocated windows since additional structures for each registration may need to be maintained by the MPI library. This mode can, however, be used for more dynamic programs that may require process-local memory management, such as dynamically sized hash tables or object-oriented domain-specific languages.

3.2. Shared-Memory Support

Shared-memory window allocation allows processes to directly map memory regions into the address space of all processes, if supported by the underlying architecture. For example, if an MPI job is running on multiple multicore nodes, then each of those nodes could share its memory directly. This feature may lead to much lower overhead for communications and memory accesses than going through the MPI layer. The *win allocate shared* function will create such a directly mapped window for process groups where all processes can share memory directly.

The additional function *comm split type* enables programmers to determine maximum groups of processes that allow such memory sharing. More details on shared-memory windows and detailed semantics and examples can be found in [Hoefler et al. 2012]. Figure 3 shows an example of shared-memory windows on a dual-core system with four nodes.

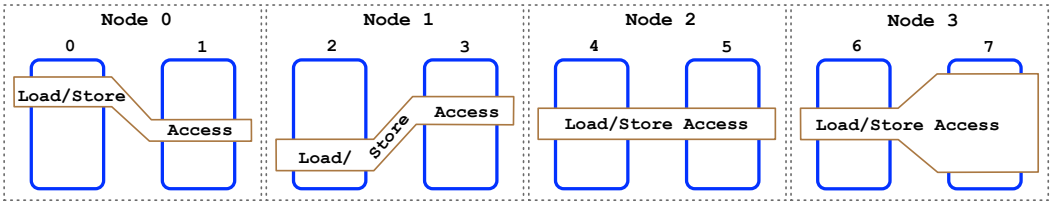


Fig. 3. MPI-3 shared-memory window layout on a dual-core system with four nodes. Each node has its own window that allows load/store and RMA accesses. The different shapes indicate that each process can pick its local window size independently of the other processes.

3.3. Memory Access

One strength of the MPI RMA semantics is that they pose only minimal requirements on the underlying hardware to support an efficient implementation. For example, the *put* and *get* calls require only that the data be committed to the target memory and provide initiator-side completion semantics. Both calls make no assumption about the order or granularity of the commits. Thus, races such as overlapping updates or reads conflicting with updates have no guaranteed result without additional synchronization. This model supports networks with nondeterministic routing as well as noncoherent memory systems.

3.4. Accumulates

Similar to put and get, accumulates strive to place the least possible restrictions on the underlying hardware. They are also designed to take direct advantage of hardware support if it is available. The minimal guarantee for accumulates are atomic updates (something much harder to achieve than simple data transport). The update is atomic only on the unit of the smallest datatype in the MPI call (usually 4 or 8 bytes), which is often supported in hardware. For larger types that may not be supported in hardware, such as the "complex" type, the library can always fall back to a software implementation.

Accumulates, however, allow overlapping conflicting accesses only if the basic types are identical and well aligned. Thus, a specification of ordering is required. Here, MPI offers strict ordering by default, which is most convenient for programmers but may come at a cost to performance. However, strict ordering can be relaxed by expert programmers to any combination of read/write ordering that is minimally required for the successful execution of the program. The fastest mode is to require no ordering.

Accumulates can also be used to emulate atomic put or get if overlapping accesses are necessary. In this sense, *get accumulate* with the operation *no op* will behave like an atomic read, and *accumulate* with the operation *replace* will behave like an atomic write. However, one must be aware that atomicity is guaranteed only at the level of each basic datatype. Thus, if two processes use *replace* to perform two simultaneous accumulates of the same set of two integers (either specified as a count or as a datatype), the result may be that one integer has the value from the first process and the second integer has the value from the second process.

3.5. Request-Based Operations

Bulk local completion of communications has the advantage that no handles need to be maintained in order to identify specific operations. These operations can run with little overhead on systems where this kind of completion is directly available in hardware, such as Cray's Gemini or Aries interconnects [Alverson et al. 2010; Faanes et al. 2012]. However, some programs require a more fine-grained control of local buffer re-

sources and thus need to be able to complete specific messages. For such cases, request-based operations, `MPI_Rput`, `MPI_Rget`, `MPI_Raccumulate`, and `MPI_Rget_accumulate` can be used. These operations return an `MPI_Request` object similar to nonblocking point-to-point communication that can be tested or can wait for completion using `MPI_Test` and `MPI_Wait`, or the equivalent. Here, completion refers only to local completion. For `MPI_Rput` and `MPI_Raccumulate` operations, local completion means that the local buffer is to be reused. For `MPI_Rget` and `MPI_Rget_accumulate` operations, local completion means that the remote data has been delivered to the local buffer.

Request-based operations are expected to be useful in the model where the application issues a number of outstanding RMA operations and waits for the completion of a subset of them before it can start its computation. A common case would be for the application to issue data fetch operations from a number of remote locations (e.g., using `MPI_Rget`) and process them out of order as each one finishes (see Listing 1).

```
1  int main(int argc, char **argv)
2  {
3        /* MPI initialization and window creation */
4
5        for (i = 0; i < 100; i++)
6            MPI_Rget(buf[i], 1000, MPI_DOUBLE, ..., &req[i]);
7
8        while (1) {
9            MPI_Waitany(100, req, &idx, MPI_STATUS_IGNORE);
10           process_data(buf[idx]);
11       }
12
13       /* Window free and MPI finalization */
14       return 0;
15 }
```

Listing 1. Example (pseudo) code for using Request-based Operations

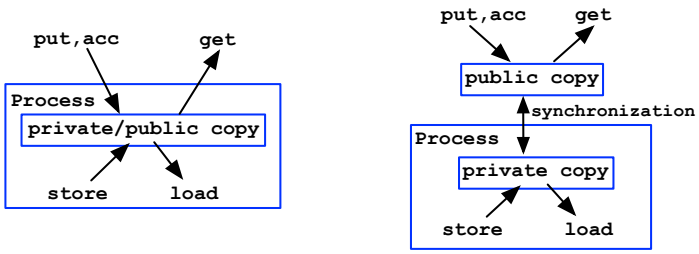
Request-based operations allow for finer-grained management of individual RMA operations, but users should be aware that the associated request management can also cause additional overhead in the MPI implementation.

3.6. Memory Models

In order to support different applications and systems efficiently, MPI defines two memory models: separate and unified. These memory models define the conceptual interaction with remote memory regions. MPI logically separates each window into a private and a public copy. Local CPU operations (also called load and store operations) always access the local copy of the window whereas remote operations (get, put, and accumulates) access the public copy of the window. Figure 4 shows a comparison between the two memory models.

The *separate* memory model models systems where coherency is managed by software. In this model, remote updates target the public copy and loads/stores target the private copy. Synchronization operations, such as lock/unlock and sync, synchronize the contents of the two copies for a local window. The semantics do not prescribe that the windows *must* be separate, just that they *may* be separate. That is, remote updates may also update the private copy. However, the rules in the separate memory model ensure that a correct program will always observe memory consistently. Those rules force the programmer to perform separate synchronization.

The *unified* memory model relies on hardware-managed coherence. Thus, it assumes that the private and public copies are identical; that is, the hardware automatically propagates updates from one to the other (without MPI calls). This model is close to today's existing RDMA networks where such propagation is always performed. It



(a) Unified memory model (b) Separate memory model

Fig. 4. Unified and separate memory models.

allows one to exploit the whole performance potential from architectures in which both the processor and network provide strong ordering guarantees. Moreover, it places a lower burden on the programmer since it requires less explicit synchronization.

A portable program would query the memory model for each window and behave accordingly. Programs that are correct in the separate model are always also correct in the unified model. Thus, programming for separate is more portable but may require additional synchronization calls.

3.7. Synchronization

All communication operations are nonblocking and arranged in epochs. An epoch is delineated by synchronization operations and forms a unit of communication. All communication operations are completed locally and remotely by the call that closes an epoch (additional completion calls are also available and are discussed later). Epochs can conceptually be split into access and exposure epochs, where the process-local window memory can be accessed remotely only if the process is in an exposure epoch, and a process can access remote memory only while in an access epoch itself. Naturally, a process can be simultaneously in access and exposure epochs.

MPI offers two main synchronization modes based on the involvement of the target process: active target synchronization and passive target synchronization. In active target synchronization, the target processes expose their memory in exposure epochs and thus participate in process synchronization. In passive target synchronization, the target processes are always in an exposure epoch and do not participate in synchronization with the accessing processes. Each mode is targeted at different use cases. Active target synchronization supports bulk-synchronous applications with a relatively static communication pattern, while passive target synchronization is best suited for random accesses with quickly changing target processes.

3.7.1. Active Target Synchronization. MPI offers two modes of active target synchronization: fence and general. In the fence synchronization mode, all processes associated with the window call fence and advance from one epoch to the next. Fence epochs are always both exposure and access epochs. This type of epoch is best suited for bulk synchronous parallel applications that have quickly changing access patterns, such as many graph-search problems [Willcock et al. 2011].

In general active target synchronization, processes can choose to which other processes they open an access epoch and for which other processes they open an exposure epoch. Access and exposure epochs may overlap. This method is more scalable than fence synchronization when communication is with a subset of the processes in the window, since it does not involve synchronization among all processes. Exposure

epochs are started with a call to post (which exposes the window memory to a selected group) and completed with a call to test or wait (which tests or waits for the access group to finish their accesses). Access epochs begin with a call to start (which may wait until all target processes in the exposure group exposed their memory) and finish with a call to complete. The groups of start and post and complete and wait must match; that is, each group has to specify the complete set of access or target processes. This type of access is best for computations that have relatively static communication patterns, such as many stencil access applications [Datta et al. 2008]. Figure 5 shows example executions for both active target modes.

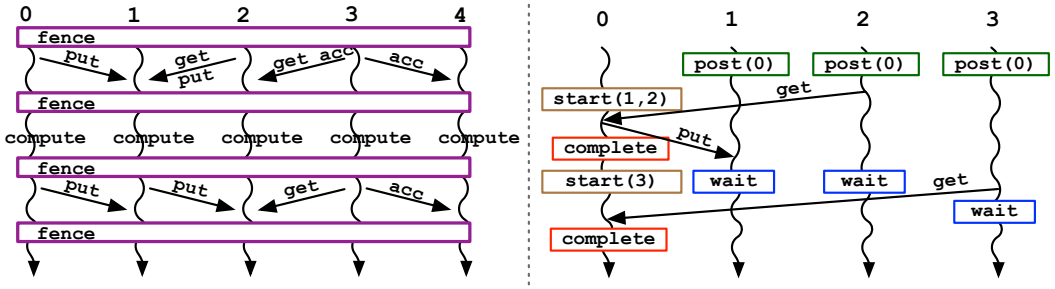


Fig. 5. Active target synchronization: (left) fence mode for bulk-synchronous applications and (right) scalable active target mode for sparse applications.

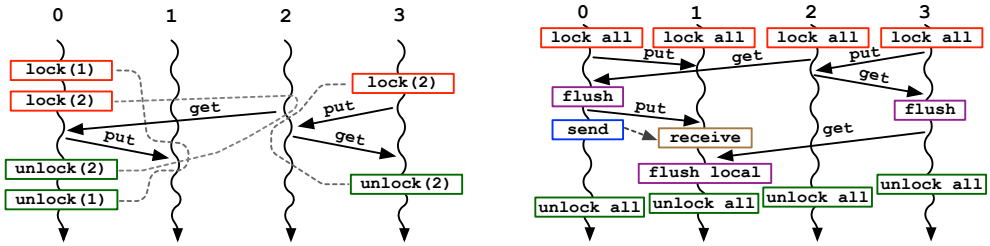
3.7.2. Passive Target Synchronization. The concept of exposure epoch is not relevant in passive mode, since all processes always expose their memory. This feature leads to reduced safety (i.e., arbitrary accesses are possible) but also potentially to improved performance. Passive mode can be used in two ways: single-process lock/unlock as in MPI-2 and global shared lock accesses.

In the single process lock/unlock model, a process *locks* the target process before accessing it remotely. To avoid conflicts with local accesses (see Section 4), a process may lock its local window exclusively. Exclusive remote window locks may be used to protect conflicting accesses, similar to reader-writer locks (shared and exclusive in MPI terminology). Figure 6(a) shows an example with multiple lock/unlock epochs and remote accesses. The dotted lines represent the actual locked region (in time) when the operations are performed at the target process. Note that the lock function itself is a nonblocking function—it need not wait for the lock to be acquired.

In the global lock model, each process starts a *lock all* epoch (it is by definition shared) to all other processes. Processes then communicate via RMA operations to update data and use point-to-point communication or synchronization operations for notification. Fine-grained data-structure locks, such as MCS (see Section 6.3), could be implemented in this mode. Figure 6(b) shows an example of a lock all epoch with several communications and flushes. MPI also allows mixing both models freely.

4. SEMI-FORMAL DEFINITION OF SEMANTICS

Our specification of the memory model tries to be as precise as possible while still being readable by professional programmers. We aim to specify the semantics sufficiently precisely in order to allow others to derive a formal specification of the MPI-3 RMA memory models, programs with defined semantics, and valid transformations of such



(a) Lock/unlock epochs (b) Lock all epoch

Fig. 6. Passive target mode examples.

programs.[1] For this purpose, we follow the conventions from [Manson et al. 2005] and [Boehm and Adve 2008].

Formal semantics can be used for proving consistency of the standard; indeed, we found two issues in MPI-3.0 while using formal modeling: (1) a loose definition that allows interpretation of memory consistency rules in different, conflicting ways and (2) a missing definition for the interaction between active and passive target mode. We also found that the formal notation can be used to better describe corner cases, many of which resolve themselves after formalization. In addition, applications written in MPI-3 RMA can be verified for correctness and determinism. Formal semantics could also be used to design and verify semantics-preserving compiler transformations.

MPI's memory semantics are specified in terms of regions of exposed memory called MPI windows. All MPI RMA calls are constrained to target a single window. In addition, the window's memory can be accessed by the program using load and store operations (induced by statements in the program). Our model considers only operations on memory associated with a window. To simplify our notation, we assume that each (named) scalar value occupies a distinct memory location in a window (with a per-byte granularity).

Each memory window has an associated set of MPI processes that may perform MPI RMA operations on the window memory of any process in the set. MPI-3 RMA offers memory operations and synchronization operations. Both are needed in order to update remote memory consistently. The effect of memory or synchronization operations during program execution is modeled by memory or synchronization actions.

Following [Manson et al. 2005] and [Boehm and Adve 2008], a memory action is defined as the tuple.

$$\langle t, o, d, rl, wl, u \rangle$$

- t : can be one of the following: memory write (w), memory read (r), remote put (rp), remote get (rg), remote get accumulate (rga, with the special case fetch and op), remote accumulate (rac), or remote compare and swap ($rcas$).
- o : indicates the MPI process number for the origin of the action.
- d : indicates the MPI process number for the destination of the action. Actions of type w and r can have only the local process as destination.
- rl : indicates the location read by the action. This is not specified for w and is a tuple of the form $\langle$compare location, swap location$\rangle$ for $rcas$.
- wl : indicates the location written by the action (not specified for r).

[1] Small additions and the removal of simplifications are necessary for defining a full formal model. The model presented in this work is kept simple and readable to allow human reasoning.

u : indicates a label identifying the source program point.

A synchronization action is defined as the tuple.

$$\langle t, o, d, u \rangle$$

t : can be one of the following: fence (f), lock shared (ls), lock exclusive (le), unlock (ul), lock all (la), unlock all (ula), flush (fl), flush local (fll), flush all (fla), flush local all ($flla$), win-sync (ws), and external synchronization (ES, e.g., matching send/recv pairs or collective operations).
o : indicates the process number for the origin of the action.
d : indicates the process number for the destination of the action. The actions la, ula, fla, $flla$ have a special identifier $\diamond$ as destination, which stands for the entire set of processes associated with the window.
u : indicates a label identifying the source program point.

We omit the generalized active target synchronization and request-based RMA operations in order to keep the notation simple. They are conceptually similar to the modeled operations, and their omission does not affect the conclusions drawn. However, modeling these would require several new symbols and interactions and significantly complicate the text, jeopardizing our goal of readability.

We introduce the following groups of actions: put/get actions $RPG = \{rp, rg\}$, accumulate actions $RA = \{rac, rga, rcas\}$, communication actions $CA = \{RPG, RA\}$, memory actions $M = \{CA, r, w\}$, and synchronization actions $S = \{fe, ls, le, ul, la, ula, fl, fll, fla, flla, ws, ES\}$. For convenience, we abbreviate an instance of an action a with type z as z instead of $a.t = z$. Additionally, we abbreviate an instance of an action a where $a.t \in Z$ as z.

An execution of a program can be defined as follows.

$$X = \langle P, A, \xrightarrow{po}, W, V, \xrightarrow{so}, \xrightarrow{hb}, \xrightarrow{co} \rangle$$

P : is the program to be executed.
A : is the set of all actions (S and M).
$\xrightarrow{po}$ specifies the program order of actions at the same process much like the "sequenced-before" order in C++ or the "program order" in Java. This specifies the order of executions in a single-threaded execution.
W : is a function that returns the write (w), remote put (rp), or remote accumulate (ras, $rcas$, rga) that wrote the value into the location read by the specified action.
V : is a function that returns the value written by the specified action.
$\xrightarrow{so}$ is a total order of the synchronization relations between synchronization actions including external waiting-for relationships (such as arise from ES actions, e.g., collective operations and matched send/recv pairs).
$\xrightarrow{hb}$ is a transitive relation between each pair of actions. The relation $\xrightarrow{hb}$ is the transitive closure of the union of $\xrightarrow{po}$ and $\xrightarrow{so}$.
$\xrightarrow{co}$ defines a partial order of the memory actions; a consistency edge $a \xrightarrow{co} b$ guarantees that the memory effects of action a are visible before b. This order is necessary because some synchronization actions (e.g. flush) order memory accesses without synchronizing processes.

The consistency order $\xrightarrow{co}$ does not introduce any $\xrightarrow{hb}$ relations; in fact, if $a \xrightarrow{co} b$, then the effects are *guaranteed visible* only if also $a \xrightarrow{hb} b$. Otherwise, b could happen (in real time) before a, and thus a's effects are not visible to b even though $a \xrightarrow{co} b$ holds for a particular execution (see Section 4.7). It is guaranteed, however, that $a \xrightarrow{co} b$ implies

that operations that happen later than b will *eventually* observe the effects of a. This guarantee is needed for polling and does not require an $\xrightarrow{hb}$ ordering (see Section 4.5).

On the other hand, an operation that synchronizes processes and is thus part of $\xrightarrow{hb}$ may not synchronize memory accesses; that is, $a \xrightarrow{hb} b$ does not imply $a \xrightarrow{co} b$. Some synchronization operations, such as $ul \xrightarrow{hb} le$, also ensure $ul \xrightarrow{co} le$ if $ul.d = le.d$, while others, such as a point-to-point communication between two ranks, guarantee only $\xrightarrow{hb}$, or a flush fl guarantees only $\xrightarrow{co}$. A consistent *happens-before order* between a and b is also abbreviated

$$a \xrightarrow{cohb} b \equiv a \xrightarrow{hb} b \wedge a \xrightarrow{co} b. \tag{1}$$

For two actions a and b and an order of type Z, we say that a and b are not ordered by Z as follows.

$$a \parallel_Z b \equiv \neg(a \xrightarrow{Z} b \vee b \xrightarrow{Z} a) \tag{2}$$

For example, $a \parallel_{hb} b$ means that a and b happen concurrently in happens-before order. For three actions a, b, c,

$$[a, b] \xrightarrow{Z} c \equiv a \xrightarrow{Z} c \vee b \xrightarrow{Z} c \tag{3}$$

which reads "a or b precede c in order Z". Let d be a dummy action (defined later).

4.1. Valid Executions

We now specify valid programs in terms of operational semantics of their executions. In a valid program, all executions must be valid. An execution is valid under the following conditions.

(1) The actions occur in an order consistent with the program, namely, $\xrightarrow{po}$. In particular, for actions a and b,

$$a \xrightarrow{po} b \Rightarrow a \xrightarrow{hb} b \tag{4}$$

(2) Passive target (lock/unlock) epochs must be well-formed; they must be opened and closed in the correct order at each process (e.g., each unlock is preceded by a matching lock in program order and each lock is followed by an unlock). In addition, an origin process may not lock a target process if the process is currently locked by the origin process or if it an active target access epoch is in-progress between that origin and target. Formally, for an action a where $a.t = [ls, le]$,

$$\exists b : a \xrightarrow{po} b \wedge b.t = ul \wedge a.d = b.d \wedge \forall c \in \{a..b\}, c.t \neq fe \wedge c.t = [ls, le] \Rightarrow c.d \neq a.d \tag{5}$$

and similarly, when $a.t = la$,

$$\exists b : a \xrightarrow{po} b \wedge b.t = ula \wedge \forall c \in \{a..b\}, c.t \neq [fe, ls, le, ul]. \tag{6}$$

Where,

$$\{a..b\} \equiv \{c : a \xrightarrow{po} c \xrightarrow{po} b\}. \tag{7}$$

(3) Fence actions are matched correctly; that is, for each fence fe_i on process i, there must be a corresponding fence fe_k on each other process k (for all processes in the window) such that $fe_i \parallel_{hb} fe_k$.

(4) Windows may not be locked and exposed concurrently. For actions $fe_0 \xrightarrow{po} ca \xrightarrow{po} fe_1$, fe_0 opens a fence epoch when the `MPI_NOSUCCEED` is not given and

$$\{fe_0..ca\} \cap S = \emptyset. \tag{8}$$

The fe_1 action closes a fence epoch when the MPI_NOPRECEDE assertion is not given and

$$\{ca..fe_1\} \cap S = \emptyset. \tag{9}$$

(5) The program is deadlock-free; that is, the directed graph $G = (A, \xrightarrow{hb})$ contains no cycles (this excludes the synchronization orders introduced by fence, unlocks, and flushes).
(6) For each communication action ca, the origin process $ca.o$ is in an epoch of type access (see Section 4.2). The target process $ca.t$ is in an epoch of type exposure if the accessing origin process's last synchronization operation (in $\xrightarrow{po}$) was of type fe.

The orders $\xrightarrow{po}$ and $\xrightarrow{so}$ are uniquely defined by the execution schedule and the rules for a well-formed execution. The consistency order is defined by the semantics of epochs and synchronization operations. We define these in the following sections.

4.2. Epochs and Synchronization Semantics

Epochs have a total order per process and can be of type *access* (the process acts as source of RMA operations), type *exposure* (the process acts as destination of RMA operation), or a combination of both. Each epoch starts with a synchronization action $[fe, le, ls, la]$ and ends with a matching synchronization action $[fe, ul, ula]$. Each memory action a is assigned to one epoch by $E(a)$, and each epoch is limited by matching synchronization actions (in $\xrightarrow{po}$). Two epochs x and y are ordered by $\xrightarrow{hb}$ if the ending synchronization action s_x of x is ordered with the starting synchronization action s_y of y as $s_x \xrightarrow{hb} s_y$.

4.2.1. Active Target Synchronization. In the fence synchronization mode, the transition from epoch i to epoch $i+1$ occurs collectively such that all processes are always in the same epoch. The $\xrightarrow{co}$ of a fence orders all memory actions before (in $\xrightarrow{po}$) the fence before all memory actions after the fence. For our definition of the memory model, we assume that a fence also synchronizes all processes in $\xrightarrow{so}$, even though such is not always true.

A fence introduces $\xrightarrow{co}$ and $\xrightarrow{so}$ between all pairs of processes. In addition, a fence guarantees local consistency:

$$[r, w, CA, d] \xrightarrow{po} fe \Rightarrow [r, w, CA, d] \xrightarrow{co} fe \tag{10}$$

and

$$fe \xrightarrow{po} [r, w, CA, d] \Rightarrow fe \xrightarrow{co} [r, w, CA, d]. \tag{11}$$

4.2.2. Passive Target Synchronization. In the passive target synchronization mode, the concept of an exposure epoch does not exist, and all processes can be accessed at any time without any MPI call at the ("passive") target. A process-local access epoch is opened to a single process k after an action $[ls, le].d = k$ and ends with a $ul.d = k$ action.

Lock operations can be either shared or exclusive. In a valid execution, a shared lock ls has a synchronization order $ul \xrightarrow{so} ls$ to all previous unlocks ul with $ls.d = ul.d$, and the ul is unlocking an exclusive lock epoch. An exclusive lock le has a synchronization order $ul \xrightarrow{so} le$ to all previous unlocks ul with $le.d = ul.d$. From the program order,

$$[ls, le, la] \xrightarrow{po} [r, w] \Rightarrow [ls, le, la] \xrightarrow{co} [r, w]. \tag{12}$$

Unlock completes communications locally and remotely. The local completion occurs as follows:

$$ca \xrightarrow{po} ul \Rightarrow ca \xrightarrow{co} ul \tag{13}$$

$$ul \xrightarrow{po} ca \wedge ul.d = ca.d \Rightarrow ul \xrightarrow{co} ca. \tag{14}$$

An unlock also guarantees that local memory is consistent, such that $ul \xrightarrow{po} [r, w] \Rightarrow ul \xrightarrow{co} [r, w]$ if-and-only-if the values written or read by w and r, respectively, were accessed by some ca action, and $ul.d = ca.d$.

In addition, each unlock generates a virtual action d at $ul.d$ with $ul \xrightarrow{co} d$, $d \xrightarrow{co} ul$, $ul \xrightarrow{so} d$, and $d \xrightarrow{so} ul$. This virtual action represents the access to the remote public window and guarantees the following:

$$d \xrightarrow{hb} [ws, CA, ls, le, la] \Rightarrow d \xrightarrow{co} [ws, CA, ls, le, la] \tag{15}$$

$$[ws, ls, le, la] \xrightarrow{hb} d \Rightarrow [ws, ls, le, la] \xrightarrow{co} d \tag{16}$$

and in addition in the unified memory model,

$$d \xrightarrow{hb} [r, w] \Rightarrow d \xrightarrow{co} [r, w] \tag{17}$$

$$[r, w] \xrightarrow{hb} d \Rightarrow [r, w] \xrightarrow{co} d. \tag{18}$$

Moreover, the new d operation will introduce a consistency relation (see Section 4.3) to all $[ra, rg]$ actions that originate from the same process, namely,

$$[ra.d, rg.d] = d.o \Rightarrow d \xrightarrow{co} [ra, rg]. \tag{19}$$

A lock-all synchronization action la opens an access epoch to all processes. The access epoch ends with an ula action. Lock-all synchronizations have the same semantics as shared access epochs to all processes.

A flush can be used to synchronize remote memory accesses. A flush fl behaves much like an unlock in that it guarantees that effects of all previous operations are visible at the target when a flush returns as well as local consistency. It also generates a virtual action d at its destination with $ul \xrightarrow{co} d$, $d \xrightarrow{co} ul$, $ul \xrightarrow{so} d$, and $d \xrightarrow{so} ul$ with the same semantics of d as described above. A flush-all fla behaves like a flush to all processes. In addition, for flushes and flush local,

$$[ra, rp, r, w] \xrightarrow{po} [fl, fla, fll] \Rightarrow [ra, rp, r, w] \xrightarrow{co} [fl, fla, fll] \tag{20}$$

$$[fl, fla, fll] \xrightarrow{po} [ra, rp, r, w] \wedge [ra, rp].d = [fl, fla, fll].d \Rightarrow [fl, fla, fll] \xrightarrow{co} [ra, rp, r, w]. \tag{21}$$

A win-sync call ws has the effect of atomically closing an existing epoch and opening a new epoch in a single action:

$$[r, w, d] \xrightarrow{hb} ws \Rightarrow [r, w, d] \xrightarrow{co} ws \tag{22}$$

$$d \xrightarrow{hb} [CA, d] \Rightarrow d \xrightarrow{co} [CA, d]. \tag{23}$$

For normal reads and writes interacting with RMA calls,

$$[r, w] \xrightarrow{po} ca \Rightarrow [r, w] \xrightarrow{co} ca. \tag{24}$$

4.3. Consistency

We now define the rules for consistent memory operations in MPI RMA. Those are needed to reason about the possible result of a set of memory operations originating at different processes.

4.3.1. Conflicting Actions. In the separate memory model (see Section 3.6), two memory actions a and b are called conflicting if they are directed towards a overlapping memory locations at the same process and either: (1) one of the two operations is a put rp, (2) exactly one of the operations is an accumulate (RA), or (3) one operation is a get (rg) and the second one a local write (w). In addition, remote writing operations (rp and RA) that access the same process conflict with local write (w) operations issued by the target process regardless of the accessed location.

In the unified model, two actions a and b are called conflicting if $a \parallel_{cohb} b$ and they are directed towards a overlapping memory locations at the same process and either: (1) one of the two operations is a put rp, (2) exactly one of the operations is an accumulate (RA), or (3) one operation is a get (rg) and the second one a local write (w).

4.3.2. Races. A data race between two conflicting operations a and b exists if they are not ordered by both $\xrightarrow{hb}$ and $\xrightarrow{co}$ relations.

$$\neg((a \xrightarrow{hb} b \wedge a \xrightarrow{co} b) \vee (b \xrightarrow{hb} a \wedge b \xrightarrow{co} a)) \tag{25}$$

$$\neg(a \xrightarrow{cohb} b \vee b \xrightarrow{cohb} a) \tag{26}$$

That is,

$$a \parallel_{hb} b \vee a \parallel_{co} b. \tag{27}$$

In other words, for a program to be free of data-races, all conflicting accesses must be ordered by $\xrightarrow{cohb}$.

4.3.3. Conditions for Well-Defined Memory Semantics. Only programs where all executions are data-race free have well-defined memory semantics. If a program has well-defined semantics, then a read action r will always return the last written value (last as defined by the consistent happens-before order).

$$W(r) \xrightarrow{cohb} r \wedge V(r) = V(W(r)) \tag{28}$$

In addition, the following property is guaranteed.

$$W(r) \xrightarrow{cohb} w \xrightarrow{cohb} r, \text{ then } r.rl \neq w.wl \wedge V(r) = V(W(r)) \tag{29}$$

In other words, in a program with well-defined memory semantics, for every read action r,

$$\neg(r \xrightarrow{cohb} W(r)). \tag{30}$$

4.4. Memory Ordering Rules

When a and b are of type RA and update the same variable,

$$a.wl = b.wl \wedge a \xrightarrow{po} b \Rightarrow a \xrightarrow{co} b. \tag{31}$$

However, the user can relax any of the possible combinations of write and read ordering (waw, war, raw, rar).

For local w and r memory actions and the local reads and writes associated with ca actions, we assume sequential ordering. The effects cannot be observed remotely since

no consistent ordering exists for those operations; thus, many local compiler transformations that do not modify sequential correctness are possible.

Remote put and get $[rp, rg]$ actions and RA actions with different destination addresses or processes have no specified ordering.

4.5. Eventual Remote Completion

The unified memory model allows the user to "poll" on a location and wait for a message to arrive without additional MPI operations. Thus, a flush or unlock on a process A could complete an RA action targeted at process B, and process B could wait in an infinite loop for the arrival for the message.

The d action that is generated on process B will not have a happens-before relation, while it will have a $[ul, fl, fla] \xrightarrow{co} d$. If process B waits (potentially an unbounded number of steps) for the message to arrive (by polling on ra.wl), it is guaranteed that the message will eventually arrive; that is, a consistent happens-before relation will be established between the $[ul, fl, fla]$ and one of the polling reads. However, there are no timing guarantees; and thus the process must wait for an unbounded number of steps.

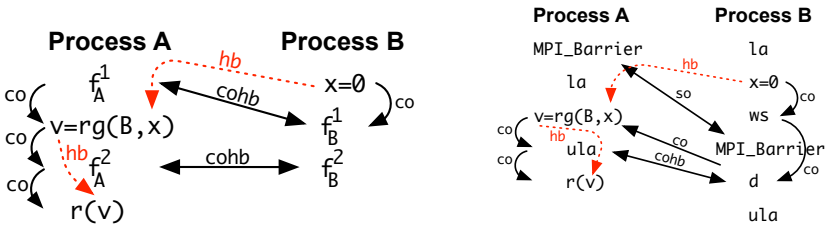
4.6. Shared-Memory Windows

All the discussions above apply to shared-memory windows. As stated before, however, there are no guarantees about the consistency order of r and w actions (which can now be observed directly by remote processes) since this is a function of the architecture's memory model (e.g., x86 [Owens et al. 2009] or POWER [Adve and Gharachorloo 1996]).

4.7. Examples

We show several examples for using the semantic definition to reason about the validity and outcome of RMA operations. To avoid cluttering the figures, we do not show process order ($\xrightarrow{po}$). Each statement at a process is ordered with regard to the previous statements at this process in $\xrightarrow{po}$, and thus $\xrightarrow{hb}$.

Figure 7(a) shows a simple example with fence synchronization. The variables x and v are accessed with conflicting operations, but the fences guarantee a $\xrightarrow{cohb}$ ordering. Thus, the result of this example trace is defined, and the read (r(v)) will always read "0."



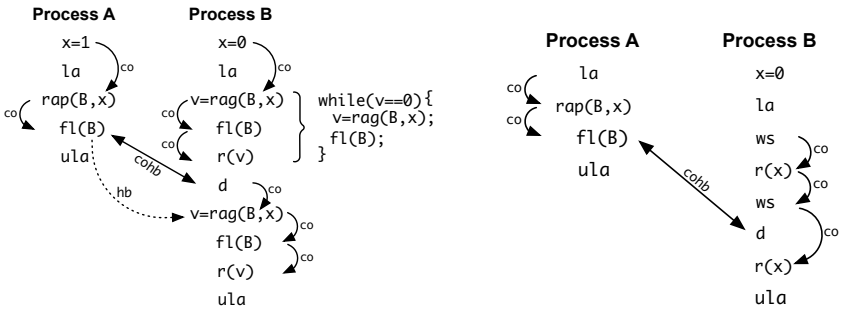
(a) Active mode fences with a remote get and output

(b) Passive mode synchronization mixed with point-to-point synchronization

Fig. 7. Simple examples for active and passive synchronization.

The formal model also allows reasoning about mixing RMA programming with traditional point-to-point programming. Figure 7(b) shows how a passive mode unlock is combined with a barrier to establish consistency and happens-before orders. The conflicting accesses are again acting on x and v. The barrier guarantees a $\xrightarrow{hb}$ ordering

between the assignment of x and the remote get. The win sync guarantees $\xrightarrow{co}$ at process b (for the separate model, it is unnecessary in the unified model), and the unlock together with the d action guarantees $\xrightarrow{co}$ order between the conflicting accesses. Thus, the program is defined, and the read will always read "0."

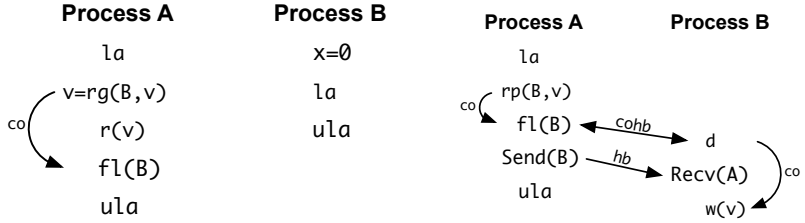


(a) Polling example in the unified model (b) Undefined polling in the separate model

Fig. 8. Examples for polling on a memory location. The abbreviations rap and rag stand for remote atomic put (accumulate with replace) and remote atomic get (get accumulate with noop), respectively.

In Figure 8(a), process A puts a value into process B's window, which waits for the value's arrival. In this example, $\xrightarrow{hb}$ is guaranteed to the d action, which itself is not ordered with regard to the actions at process B. However, since process b is in an infinite loop, d, it will eventually appear in this loop and thus introduce an eventual $\xrightarrow{hb}$ ordering. The $\xrightarrow{co}$ order is also maintained by the operations. Thus, this program is correct in the unified memory model, and v will have the value "1" at process b eventually.

Figure 8(b) shows polling in the separate memory model. This schedule is undefined since the d action can occur between a synch and a read and may thus lead to undefined outcome of the read.



(a) Get followed by an inconsistent read (b) Consistent ordering with a send/recv synchronization

Fig. 9. Examples for consistency ordering.

Figure 9(a) shows an example where a consistency edge is missing for a local access. The accesses to x are conflicting on process A, and there is no $\xrightarrow{co}$; thus, the outcome is undefined. Figure 9(b) shows an example with correct consistency ordering. Two conflicting accesses to v at process b are synchronized with a flush ($\xrightarrow{co}$) and with a send/recv pair ($\xrightarrow{hb}$). The outcome of this example is well defined.

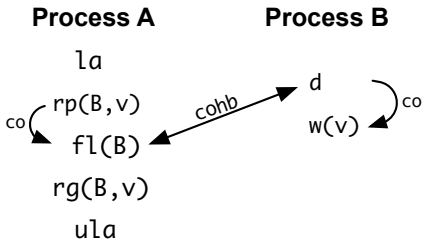


Fig. 10. Missing happens-before ordering

Figure 10 shows an example for a missing happens-before ordering. The $\xrightarrow{co}$ ordering at process B is established because of the stronger guarantees of the unified model; however, there is no $\xrightarrow{hb}$ ordering such that d could execute after the write, making the outcome undefined.

We now show how to use the semantics of MPI-3 RMA to prove correctness of a simple program.

4.7.1. Peterson's Lock. We use the following implementation of Peterson's two-process lock with MPI-RMA. The direct translation from the textbook version [Herlihy and Shavit 2012] is below:

```
1  // declarations omitted
2
3  if(rank == 0) size = 3; else size = 0;
4  MPI_Win_allocate(size, sizeof(int),
5        info_null, comm, &mem, &win);
6  // flag = offsets 0,1; victim = offset 2
7  MPI_Win_lock_all(0, win); // start lock-all MPI RMA epoch
8  peterson_lock(win, mem); // acquire the lock
9  peterson_unlock(win, mem); // release the lock
10 MPI_Win_unlock_all(win); // end the lock-all MPI RMA epoch
11
12 void peterson_lock(MPI_Win win) {
13   int j = 1-rank;
14   MPI_Put(&one, ..., 0, rank, ..., win);
15   MPI_Put(&rank, ..., 0, 2, ..., win);
16   while() {
17     int rflag, rvict;
18     MPI_Get(&rflag, ..., 0, j, ..., win);
19     MPI_Get(&rvict, ..., 0, 2, ..., win);
20     if(!rflag || rvict != rank) break;
21   }
22 }
23
24 void peterson_unlock(MPI_Win win) {
25   MPI_Put(&zero, ..., 0, rank, ..., win);
26 }
```

Listing 2. Simple (incorrect) Peterson Lock in MPI-3

Let $P_A(x = z)$ denote a put rp with $rp.o = A$, $rp.wl = x$, and $V(rp) = z$, and similarly let a $G_A(x)$ denote a get rg with $rg.o = A$ and $rg.rl = x$. We denote offset 0 and 1 in the window as f_A and f_B (for process A's and B's flag) and offset 2 as v for the victim variable in the textbook version. CR denotes the critical region to be protected from concurrent access.

4.7.2. Consistency. The code above defines the orders

$$P_A(f_A = 1) \xrightarrow{po} P_A(v = A) \xrightarrow{po} rf_A = G_A(f_B) \xrightarrow{po} rv = G_A(v) \xrightarrow{po} R_A(rf_A) \xrightarrow{po} R_A(rv_A) \xrightarrow{po} CR$$
$$P_B(f_B = 1) \xrightarrow{po} P_B(v = B) \xrightarrow{po} rf_B = G_B(f_A) \xrightarrow{po} rv_B = G_B(v) \xrightarrow{po} R_B(rf_B) \xrightarrow{po} R_B(rv_B) \xrightarrow{po} CR.$$

Since there is no $\xrightarrow{hb}$ order between the puts and gets, the program has a race (between $P_A(v = A)$ and $P_B(v = B)$), which makes the outcome undefined under all MPI memory models.

One way to guarantee a defined outcome for this program is to introduce epochs and $\xrightarrow{hb}$ to avoid conflicts. In the unified memory model, one could use an le epoch for the two puts in line 14/15 and an ls epoch for the two gets in line 18/19. The separate model would require separate epochs for lines 14 and 15. However, this would require an exclusive window lock in order to implement the two-process Peterson lock algorithm, and one would wonder why a single window lock would not be sufficient.

A second option would be to change the accesses to accumulates, which do not cause conflicts. The MPI standard describes how to use Fetch_and_op() and Accumulate() to simulate atomic get and put functionality. The two orders above would still be valid (simply assuming P and G are atomic and thus do not cause races). In addition, one would need to introduce a local $\xrightarrow{co}$ edge between the gets in line 18/19 and the local read in line 20; this can be achieved with a flush local fll before line 20. The outcome of this transformed program is now defined in terms of MPI. The full source code of the correct lock is shown in Listing 3.

4.7.3. Correctness. The proof that the lock provides mutual exclusion fails because the two put calls are not ordered with respect to the get calls. They could align in a way that the get fetches the value before the put commits to memory (due to the missing $\xrightarrow{co}$ relation). This relation can be introduced by adding a fl action before line 16. In addition, the two writes need to appear in order, so another mfl action is necessary at before line 15.

Proof by contradiction: Assume processes A and B are in the critical region. The program enables the following possible execution for process A.

$$P_A(f_A = 1) \xrightarrow{po} fl^1_A(B) \xrightarrow{po} P_A(v = A) \xrightarrow{po} fl^2_A(B) \xrightarrow{po} G_A(f_B)G_A(f_B) \xrightarrow{po} G_A(v) \xrightarrow{po} fll_A(B) \xrightarrow{po} CR$$

The semantic rules for RMA imply the following.

$$P_A(f_A = 1) \xrightarrow{cohb} P_A(v = A)] \xrightarrow{cohb} [G_A(f_B), G_A(v)] \xrightarrow{cohb} [R_A(rf_A), R_A(rv_A)] \xrightarrow{hb} CR$$

Similar, execution and consistency orders can be established for process B. Without loss of generality, assume that $P_A(v = A)$ commits after $P_B(v = B)$, that is, $P_B(v = B) \xrightarrow{cohb} P_A(v = A)$. This establishes the following order.

$$P_B(f_B = 1) \xrightarrow{cohb} P_B(v = A)] \xrightarrow{cohb} P_B(v = B) \xrightarrow{cohb} [G_A(f_B), G_A(v)] \xrightarrow{cohb} [R_A(rf_A), R_A(rv_A)] \xrightarrow{hb} CR$$

This implies that thread A cannot have entered the CR since it must have read $f_B = 1$ and $v = A$.

```
1  // declarations omitted
2
3  if(rank == 0) size = 3; else size = 0;
4  MPI_Win_allocate(size, sizeof(int),
5        info_null, comm, &mem, &win);
6  // flag = offsets 0,1; victim = offset 2
7  MPI_Win_lock_all(0, win); // start lock-all MPI RMA epoch
8  peterson_lock(win, mem); // acquire the lock
9  peterson_unlock(win, mem); // release the lock
10 MPI_Win_unlock_all(win); // end the lock-all MPI RMA epoch
11
12 void peterson_lock(MPI_Win win) {
13   int j = 1-rank;
14   MPI_Put(&one, ..., 0, rank, ..., win);
15   MPI_Win_flush(0);
16   MPI_Put(&rank, ..., 0, 2, ..., win);
17   MPI_Win_flush(0);
18   while() {
19     int rflag, rvict;
20     MPI_Get(&rflag, ..., 0, j, ..., win);
21     MPI_Get(&rvict, ..., 0, 2, ..., win);
22     MPI_Win_flush_local(0);
23     if(!rflag || rvict != rank) break;
24   }
25 }
26
27 void peterson_unlock(MPI_Win win) {
28   MPI_Put(&zero, ..., 0, rank, ..., win);
29 }
```

Listing 3. Simple (correct) Peterson Lock in MPI-3

5. IMPLEMENTATION ISSUES

A wide variety of implementation choices exists based on the communication and addressing features provided by the system. Interconnects that provide only messaging (e.g., Ethernet) require a remote software agent in order to perform RMA operations in the target process's memory, whereas networks that provide RDMA or active messaging capabilities can rely on hardware and system-level agents to move data to and from the target's memory. Software agents allow the origin process to shift processing that requires knowledge of the target process state (e.g., displacement unit and base address) to the target, potentially improving the scalability of origin data structures that track the target process state. In contrast, hardware and system-level agents consume fewer computational resources but may require additional steps (e.g., memory registration and handshaking) to configure the agent for communication with a given target. If a hardware agent cannot perform the desired operation directly (e.g., double-precision or complex accumulate), a software solution requiring additional processing at the origin and/or target must be used.

5.1. Implementation on Message-Based Networks

Messaging-based networks require the use of a software agent at the target to process RMA operations [Dinan et al. 2013]. These agents are implemented within the MPI progress engine, and they can be triggered through one or more threads that dedicate CPU resources to processing incoming messages in the progress engine. Spawning additional threads ensures asynchronous progress for RMA operations. However, for applications that are not sensitive to asynchronous progress for performance and do not rely on asynchronous data consistency in the unified model, an MPI implementation

may allow communication helper threads to be disabled by using an info argument, and processing of RMA operations can be performed by the target process when it enters an MPI call.

This implementation effectively treats RMA operations as remote procedure calls, in which the origin packages the arguments to the operation and ships them to the target, where the operation will be applied. Such an implementation simplifies the processing of complex RMA operations. Derived datatypes that describe the target buffer can be serialized and shipped to the target process for use in applying the RMA operation; complex operations in calls to accumulate (e.g., `MPI_MAXLOC` or the use of MPI pair types) can be applied programmatically at the target; and atomic operations can use system-supported atomic operations and synchronization mechanisms. In addition, generalized message processing at the target enables multiple opportunities for performance optimization, through pipelining of operations and piggybacking of RMA synchronization operations on top of RMA messages.

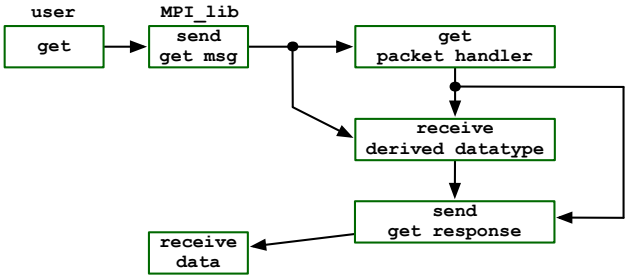


Fig. 11. Origin (left) and target (right) actions taken when performing a get operation in a messaging-based implementation of MPI RMA.

Active target synchronization can be handled efficiently through two-sided and collective communication, as discussed in [Thakur et al. 2005]. For passive target synchronization, a lock queue is managed at each process, and lock request messages can be merged and piggybacked with RMA operations in many cases, for example, when the user specifies `MPI_MODE_NOCHECK` or performs one operation within an epoch. Remote completion for flush and unlock operations can be achieved by requesting a completion acknowledgment from the target. For exclusive epochs, notification is not necessary because other processes will not be granted access to the target window until the current epoch has completed at the target.

5.2. Implementation on RDMA Networks and Shared-Memory Systems

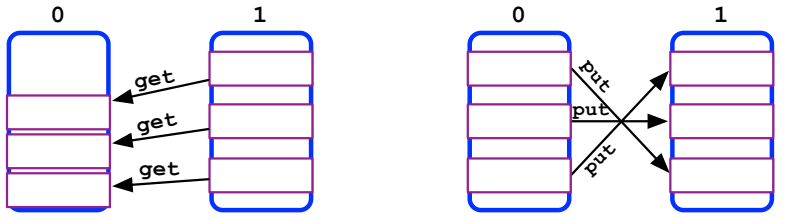
If the network supports direct remote memory access, then one can use those facilities directly to implement MPI-3 RMA. In fact, one of the major strengths of the defined interface is that the control path between the MPI call and the hardware can be kept at a minimum (less than 200 CPU instructions for contiguous data). This enables high message rates and asynchronous progression.

Window-creation operations must register the exposed memory locally in order to initialize the virtual-to-physical memory translation tables in the network card. This action may cause significant overhead [Mietke et al. 2006], however, so window-creation routines should be used sparingly. The dynamic windows offer a more flexible mechanism for adding memory to an existing window. Nevertheless, its costs may not be trivial if the network hardware relies on memory registration.

Synchronization operations depend on the capabilities of the underlying hardware. If the hardware offers remote completion, then a fence can be implemented by completing all remote operations and followed by a barrier call. General active target synchronization is more complex, and a scalable protocol requires correct matching of access and exposure groups. An $\mathcal{O}(\log p)$ time and space protocol is described in [Gerstenberger et al. 2013]. Passive target locks—global (lock all) and local (process lock) shared and exclusive locks—can be implemented by using reader-writer global and local locking [Mellor-Crummey and Scott 1991b].

Communication operations, such as put and get, can usually be translated directly into RDMA calls. Some accumulates may be directly supported in hardware by remote atomic operations. If they are not supported, they can be emulated with a simple (but inefficient) protocol that locks the remote window, gets the data to the origin, performs the operation locally, and puts the data back to the target. The window lock must protect from accesses or updates to the memory that is targeted by the accumulate.

MPI datatypes can be handled easily if the hardware provides scatter/gather support for RDMA operations. If not, datatypes can simply be split into the smallest contiguous blocks from source to target buffers and issued as separate transfers. Depending on the local and remote types, this approach may result in very small messages, and, if the message-injection rate of the hardware is the limiting factor, protocols based on pack/unpack may be used to utilize the network bandwidth more efficiently. Figure 5.2 shows how an RDMA put and get can be used to implement communication from different source datatypes to different target datatypes. Figure 12(a) shows a read of a vector with three elements at the target into a vector with a different stride at the origin. Figure 12(b) shows a write of a vector that permutes the order of vector elements. In both cases, a single MPI access results in multiple (three) low-level transfers.



(a) Noncontiguous access using RDMA get

(b) Noncontiguous access using RDMA put

Fig. 12. Noncontiguous access using RDMA

6. USE CASES AND EXAMPLES

In this section, we discuss several possible uses of the new remote memory access interface. Some of those applications can be implemented with other mechanisms, such as traditional MPI-1 communication or even other new MPI-3 features such as neighborhood collectives [Hoefler and Schneider 2012]. We note that RMA programming can be faster because of the missing message-matching overhead; however, it is impossible to make general statements about performance across a wide variety of architectures. Here, we focus on MPI-3 RMA examples and provide some high-level hints to potential users. We often cannot provide detailed advice about which mechanism to use; however, we encourage MPI vendors to provide detailed performance models for all operations to help guide the user's decisions.

6.1. Stencil Boundary Exchange

Many applications follow the stencil parallel pattern, the basis of many PDE and ODE computations. In this pattern, each process is running an iterative computation and communicates with a set of neighbors in each iteration. The communication exchanges the boundary zones of the local process domains. The neighborhood relations are often fixed for several iterations (or, in fact, may never change). The computation generally follows the bulk synchronous paradigm of repeated computation and communication phases and may allow overlapping of computation and communication.

If each of the p processes communicates with a large number of neighbors k ($k > \log(p)$), then fence synchronization may be the best solution. However, if the number of neighbors is relatively small (or constant) and the neighborhood relationship is not changing often, then the general active target synchronization seems most natural. Remote memory put operations are often faster than get. If the target address can be computed at the origin, using put operations is often beneficial. Figure 13 shows an example execution of the 1D stencil exchange with overlap using fence and general active target synchronization. `Compute inner` is independent of the halo-zone values, and `compute outer` then computes the boundary that depends on the halo zone.

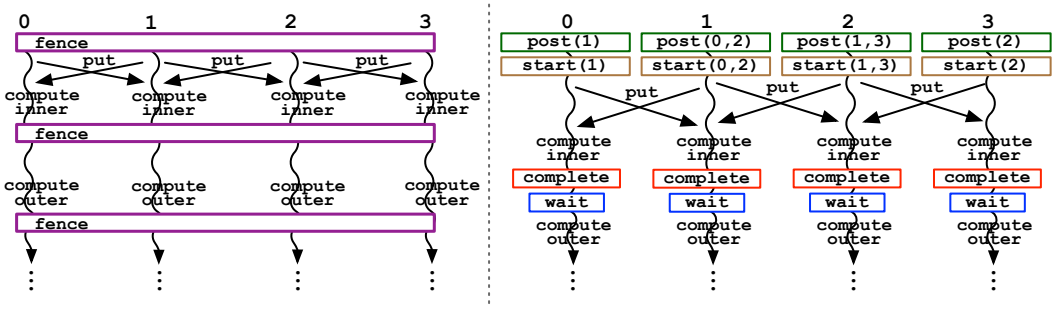


Fig. 13. 1D stencil boundary exchange example using fence (left) and general active target synchronization (right).

One can also use passive target synchronization to implement stencil codes. The benefit of passive mode is that separate targets can be completed separately and a target process can pipeline the computations that use the incoming data. Depending on the operation, different protocols must be used. For put, the source process simply puts the message into the target window and notifies the target (either by setting a notification byte or with a message). In addition, the target has to notify the source when the data can be overwritten in the next iteration in order to satisfy the output dependence at the target window. In a get-based protocol, the origin would send a notification to the target which then fetches the data and processes it. Both protocols require two remote accesses (or messages), and the better protocol depends on the put and get performance of the target system. Figure 14 shows put and get protocols for passive target synchronization.

6.2. Fast Fourier Transform

Fast Fourier transforms (FFTs) are an important kernel in many scientific applications. The computation patterns can often be arranged in different layouts by using twiddle factors. Here, we discuss a three-dimensional FFT ($X \times Y \times Z$) using a one-dimensional data decomposition as a case study.

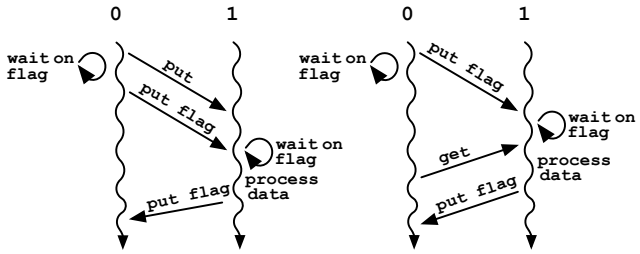


Fig. 14. Put and get protocols for passive target mode synchronization.

In the decomposition, each process has a set of two-dimensional planes. If we assume that the X dimension is distributed initially, each process can perform local Y-Z 2D FFTs. Then, a global transpose step follows such that X is contiguous at each process for performing the final 1D FFT. An optional transpose can be used to copy the data back into the original layout if needed.

An optimized RMA implementation could issue puts to remote processes as soon as the data becomes available (e.g., start all puts for a plane after the Y-Z transform of this plane completed). After all planes have been started, all processes end the epoch with a fence before moving to the X FFT. This scheme enables high overlap of computation and communication. Figure 15 shows an example decomposition and execution of a 3D FFT.

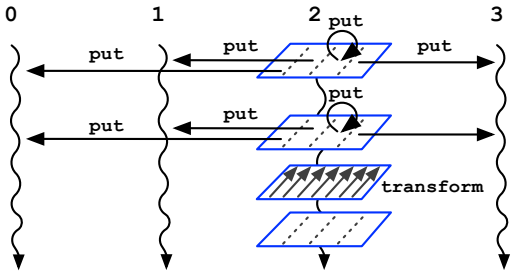


Fig. 15. Example computation of a 3D FFT.

6.3. Distributed Locks

The new atomic operations added in MPI-3 make it possible to build asynchronous, distributed, lock-free data structures. Such structures are central to a variety of scalable algorithms; the well-known MCS mutual exclusion algorithm [Mellor-Crummey and Scott 1991a], for example, uses a lock-free queue. In this queue, the process at the head holds the lock and forwards it to the next process when it has completed its critical section. Thus, the queue supports two operations: removing the element at the head of the queue and adding new elements to the tail.

The MCS algorithm uses a tail pointer in a fixed location at a specific process, which is initialized to `MPI_PROC_NULL`. In addition, each process maintains a single queue element, which contains a next pointer that is also initialized to `MPI_PROC_NULL`. These pointers are integer values that will be used to enqueue processes by rank. Thus an MPI window is created, where each process posts an integer location that will be used as its queue element, and the process hosting the mutex adds a second integer element

that will be used as the tail pointer. Once the window has been created, all processes call `MPI_Win_lock_all` to initiate shared-mode access, since accesses will be performed by using only atomic operations.

As shown in Listing 4, when processes request the lock, they atomically exchange a pointer to their list element (initialized to nil) with the tail pointer. If the tail pointer is nil, the process has successfully acquired the lock. Otherwise, it updates the element of the process that was the old list tail and waits for that process to forward the lock. All concurrent accesses are performed by using atomic operations to enable a shared lock access mode.

```
1  /* This store cannot occur concurrently with a remote write */
2  mutex->base[MCS_MTX_ELEM_DISP] = MPI_PROC_NULL;
3  MPI_Win_sync(mutex->window);
4
5  MPI_Fetch_and_op(&rank, &prev, MPI_INT, mutex->tail_rank, MCS_MTX_TAIL_DISP,
6                   MPI_REPLACE, mutex->window);
7  MPI_Win_flush(mutex->tail_rank, mutex->window);
8
9  /* If there was a previous tail, update their next pointer and wait for
10  * notification. Otherwise, the mutex was successfully acquired. */
11 if (prev != MPI_PROC_NULL) {
12     MPI_Status status;
13
14     MPI_Accumulate(&rank, 1, MPI_INT, prev, MCS_MTX_ELEM_DISP, 1, MPI_INT,
15                    MPI_REPLACE, mutex->window);
16     MPI_Win_flush(prev, mutex->window);
17     MPI_Recv(NULL, 0, MPI_BYTE, prev, MCS_MUTEX_TAG, mutex->comm, &status);
18 }
```

Listing 4. MCS mutex lock algorithm.

Similarly, when releasing the lock, shown in Listing 5, processes perform an atomic compare-and-swap of the tail pointer. If the process releasing the lock is still at the tail of the queue, the tail pointer is reset to nil. If not, the process forwards the lock to the next process in the queue, potentially waiting for that process to update the releasing process's queue element. As an optimization, processes can first check their local queue element to determine whether the lock can be forwarded without checking the tail pointer.

```
1  /* Read my next pointer. FOP is used since another process may write to
2   * this location concurrent with this read. */
3  MPI_Fetch_and_op(NULL, &next, MPI_INT, rank, MCS_MTX_ELEM_DISP, MPI_NO_OP,
4                   mutex->window);
5  MPI_Win_flush(rank, mutex->window);
6
7  if (next == MPI_PROC_NULL) {
8      int tail, nil = MPI_PROC_NULL;
9
10     /* Check if we are the at the tail of the lock queue. If so, we're
11      * done. If not, we need to send notification. */
12     MPI_Compare_and_swap(&nil, &rank, &tail, MPI_INT, mutex->tail_rank,
13                          MCS_MTX_TAIL_DISP, mutex->window);
14     MPI_Win_flush(mutex->tail_rank, mutex->window);
15
16     if (tail != rank) {
17         for (;;) {
18             int flag;
19
20             MPI_Fetch_and_op(NULL, &next, MPI_INT, rank, MCS_MTX_ELEM_DISP,
21                              MPI_NO_OP, mutex->window);
22             MPI_Win_flush(rank, mutex->window);
```

```
23                 if (next != MPI_PROC_NULL) break;
24  }    }    }
25
26  /* Notify the next waiting process */
27  if (next != MPI_PROC_NULL) {
28      MPI_Send(NULL, 0, MPI_BYTE, next, MCS_MUTEX_TAG, mutex->comm);
29  }
```

Listing 5. MCS mutex unlock algorithm.

7. DISCUSSION AND CONCLUSIONS

In this paper we described the MPI-3 one-sided interface, presented the semantics in a semi-formal way, and showed several use cases. This new interface is expected to deliver highest performance on novel network architectures that offer RDMA access directly in hardware. While being extremely efficient and close to the hardware, the new interface still offers several convenient and easy-to-use programming constructs such as process groups, exposed memory abstraction (windows), MPI datatypes, and different synchronization models. The RMA interface separates communication and synchronization and offers different collective and noncollective synchronization modes. In addition, it allows the programmer to choose between implicit notification in active target mode and explicit notification in passive target mode. This large variety of options allows users to create complex programs.

Our formalization of remote access semantics allows one to reason about complex applications written in MPI-3 RMA. We show how to prove whether programs have defined outcomes, and one can easily derive deadlock conditions from our specification of happens-before orders. Thus, we expect that the semantics will lead to powerful tools to support programmers in using MPI for RMA.

We also demonstrated some application examples for the use of MPI-3 RMA. This collection covers several important classes but is, by far, not complete.

REFERENCES

Sarita V Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *computer* 29, 12 (1996), 66–76.

Robert Alverson, Duncan Roweth, and Larry Kaplan. 2010. The Gemini System interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects (HOTI '10)*. IEEE Computer Society, Washington, DC, USA, 83–87. DOI:http://dx.doi.org/10.1109/HOTI.2010.23

Hans-J. Boehm. 2005. Threads cannot be implemented as a library. *SIGPLAN Not.* 40, 6 (June 2005), 261–268. DOI:http://dx.doi.org/10.1145/1064978.1065042

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. *SIGPLAN Not.* 43, 6 (June 2008), 68–78. DOI:http://dx.doi.org/10.1145/1379022.1375591

Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. DOI:http://dx.doi.org/10.1145/1103845.1094852

Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. 2009. MPI-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd international conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 316–325. DOI:http://dx.doi.org/10.1145/1542275.1542321

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 4, 12 pages. http://dl.acm.org/citation.cfm?id=1413370.1413375

James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2013. An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface. Preprint ANL/MCS-P4014-0113 (Jan. 2013).

Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. 2012. Cray cascade: a scalable HPC system based on a Dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 103, 9 pages. http://dl.acm.org/citation.cfm?id=2388996.2389136

Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Implementing Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. Submitted.

Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. http://books.google.ch/books?id=qGURkdAr42cC

Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2012. Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming. In *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface (EuroMPI'12)*. Springer-Verlag, Berlin, Heidelberg, 132–141. DOI:http://dx.doi.org/10.1007/978-3-642-33518-1_18

Torsten Hoefler, Rolf Rabenseifner, Hubert Ritzdorf, Bronis R de Supinski, Rajeev Thakur, and Jesper Larsson Träff. 2011. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience* 23, 4 (2011), 293–310.

Torsten Hoefler and Timo Schneider. 2012. Optimization Principles for Collective Neighborhood Communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 98, 10 pages. http://dl.acm.org/citation.cfm?id=2388996.2389129

Torsten Hoefler and Marc Snir. 2011. Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface (EuroMPI'11)*. Springer-Verlag, Berlin, Heidelberg, 345–355. http://dl.acm.org/citation.cfm?id=2042476.2042521

Sven Karlsson and Mats Brorsson. 1998. A Comparative Characterization of Communication Patterns in Applications Using MPI and Shared Memory on an IBM SP2. In *Proceedings of the Second International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC '98)*. Springer-Verlag, London, UK, UK, 189–201. http://dl.acm.org/citation.cfm?id=646092.680546

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05)*. ACM, New York, NY, USA, 378–391. DOI:http://dx.doi.org/10.1145/1040305.1040336

John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. 2009. A New Vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models (PGAS '09)*. ACM, New York, NY, USA, Article 5, 9 pages. DOI:http://dx.doi.org/10.1145/1809961.1809969

John M. Mellor-Crummey and Michael L. Scott. 1991a. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. DOI:http://dx.doi.org/10.1145/103727.103729

John M. Mellor-Crummey and Michael L. Scott. 1991b. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. *SIGPLAN Not.* 26, 7 (1991), 106–113. DOI:http://dx.doi.org/10.1145/109626.109637

Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. 2006. Analysis of the memory registration process in the Mellanox InfiniBand software stack. In *Proceedings of the 12th international conference on Parallel Processing*. Springer-Verlag, Berlin, Heidelberg, 124–133. DOI:http://dx.doi.org/10.1007/11823285_13

MPI Forum. 2009. MPI: A Message-Passing Interface Standard. Version 2.2. (September 2009).

MPI Forum. 2012. MPI: A Message-Passing Interface Standard. Version 3.0. (September 2012).

Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. 1996. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing* 10, 2 (1996), 169–189.

Robert W. Numrich and John Reid. 1998. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum* 17, 2 (1998), 1–31.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. (2009), 391–407. DOI:http://dx.doi.org/10.1007/978-3-642-03359-9_27

Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. 2001. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM) (Supercomputing '01)*. ACM, New York, NY, USA, 57–57. DOI:http://dx.doi.org/10.1145/582034.582091

Rajeev Thakur, William Gropp, and Brian Toonen. 2005. Optimizing the Synchronization Operations in MPI One-Sided Communication. *Intl. J. High-Performance Computing Applications* 19, 2 (2005), 119–128.

Jesper Larsson Traff. 2002. Implementing the MPI process topology mechanism. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (Supercomputing '02)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=762761.762767

UPC Consortium. 2005. *UPC Language Specifications, v1.2*. Technical Report. Lawrence Berkeley National Laboratory. LBNL-59208.

Jeremiah Willcock, Torsten Hoefler, Nick Edmonds, and Andrew Lumsdaine. 2011. Active Pebbles: Parallel Programming for Data-Driven Applications. In *Proceedings of the international conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 235–244. DOI:http://dx.doi.org/10.1145/1995896.1995934

Tim S. Woodall, Galen M. Shipman, George Bosilca, and Arthur B. Maccabe. 2006. High performance RDMA protocols in HPC. In *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface (EuroPVM/MPI'06)*. Springer-Verlag, Berlin, Heidelberg, 76–85. DOI:http://dx.doi.org/10.1007/11846802_18