Remote Memory Access Programming in MPI-3

TORSTEN HOEFLER, ETH Zurich
JAMES DINAN, Intel Corporation
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 Corporation

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 enable new data-access modes. We present the new RMA interface and specify formal axiomatic models for data consistency and access semantics. Such models can help users reason about details of the semantics that are hard to extract from the English prose in the standard. It also fosters the development of tools and compilers, enabling them to automatically analyze, optimize, and debug RMA programs.

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:
Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote memory access programming in MPI-3. ACM Trans. Parallel Comput. 2, 2, Article 9 (June 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2780584

1. MOTIVATION

Parallel programming models can be split into three categories: (1) shared memory with implicit communication as a side effect of loads and stores to memory and explicit

This work was 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, and award numbers DE-SC0004131 and DE-FG02-13ER26138. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.
Authors' addresses: T. Hoefler, Universitaetsstrasse 6, 8092 Zurich, Switzerland; email: htor@inf.ethz.ch; J. Dinan and K. Underwood, 77 Reed Road, Hudson, MA 01749; emails: {james.dinan, keith.d.underwood}@intel.com; R. Thakur and P. Balaji, 9700 South Cass Avenue, Lemont, IL 60439; emails: {thakur, balaji}@mcs.anl.gov; B. Barrett, Sandia National Laboratories, New Mexico, P.O. Box 5800, Albuquerque, NM 87185-1319; email: bwbarre@sandia.gov; W. Gropp, 201 North Goodwin Avenue, Urbana, IL 61801-2302; email: wgropp@illinois.edu.

DOI: http://dx.doi.org/10.1145/2780584

synchronization, (2) message passing (MP) with explicit communication and implicit synchronization as a side effect of messages, and (3) remote memory access (RMA) and partitioned global address space (PGAS) where communication and synchronization are managed explicitly and independently. Some PGAS models offer combined communication and synchronization functions to achieve higher efficiency.

High-performance computing (HPC) and datacenter networking architectures have undergone a disruptive change in the past decade: they have increasingly converged toward remote direct memory access (RDMA), a mechanism that enables a process to directly read and write remote memory. RDMA offers higher performance and lower CPU load than traditional socket communications due to operating system bypass [Shivam et al. 2001]. RDMA is also relatively simple to implement in hardware, and its low-level interfaces are well understood [Buonadonna et al. 1998]. 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.

Programming in shared memory is often a convenient abstraction because remote values can just be accessed directly. This spurred a series of research works to emulate shared memory on distributed memory computers. The general conclusion was that transparent shared memory cannot be emulated efficiently with the available distributed memory technologies [Karlsson and Brorsson 1998]. RDMA and the related PGAS abstraction provide a viable middle ground where the address space is separated into a local load/store accessible part and a remote put/get accessible part.

The traditional programming model in HPC is the highly successful MP model. In this model, processes communicate only through messages. However, the semantic mismatch between RDMA and MP can only be bridged with complex and expensive protocols [Woodall et al. 2006]. Thus, MP implementations using today's RDMA networks cannot exploit the full potential of the hardware. The Message Passing Interface (MPI) 2.0 standard introduced a one-sided communication scheme in 1997 based on the technology at that time. The MPI-2 abstract machine model does not reflect today's RDMA networks well and does not support the exploitation of RDMA's full potential, which inhibited its adoption. However, architectural trends, such as RDMA networks and the increasing number of (potentially noncoherent) cores on each node, required a reconsideration of the programming model.

The Message Passing Interface Forum, the standardization body of MPI, developed new ways for exploiting RDMA networks and multicore CPUs in MPI programs. This article, written by key members of the MPI-3 Remote Memory Access Working Group, summarizes the new one-sided communication interface of MPI-3 [MPI Forum 2012]. The main contributions of this work are (1) a detailed informal specification of MPI's RMA semantics, (2) several application patterns with the RMA synchronization strategies, and (3) a guide toward a formal specification of MPI-3's RMA semantics. Our work 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, library, or 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 article 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.

We expect that different readers will be interested in different parts of this article and hence provide a small overview. Section 2 provides an intuitive description of the semantics of RMA. It also explains the relation to RDMA architectures and provides some intuition about the performance of these operations. This section is most suited for programmers aiming to use RMA. Section 3 specifies axiomatic semantics for MPI-3 RMA's memory model and is intended for researchers and advanced developers. It can be used to answer rather subtle questions about correct executions and can be encoded into correctness checkers such as MemSAT [Torlak et al. 2010]. Section 4 provides examples to build intuition about how to use the new semantics and is thus most interesting to algorithm designers.

1.1. Related Work

Efforts in the area of parallel programming models are manifold. PGAS programming models view 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 2008 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 an HPC context. Other languages, such as Global Arrays [Nieplocha et al. 1996], offer similar semantics restricted to specific structures or domains (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 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, as it is a library interface that can work with all compilers [Yang et al. 2014]. A complete specification of the library semantics enables automated compiler transformations [Danalis et al. 2009]. In addition, MPI offers a rich set of semantic concepts such as isolated process contexts (communicators), process topologies, and abstract definitions for access patterns of communication functions (MPI datatypes). These concepts enable users to specify additional properties of their code that allow more complex optimizations at the library and compiler level [Schneider et al. 2013]. 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 enable the layered implementation of libraries on top of MPI [Hoefler and Snir 2011]. Our expectation is that experts will use MPI RMA as an efficient foundation for high-level libraries that provide domain-specific extensions, which hide most of the complexity.

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 combined with parallel executing threads may lead to erroneous or slow codes. The semantics specified in this work can also be seen as a set of restrictions for serial compilers to make them MPI-aware.

1.2. Overview and Challenges of RMA Programming

The main complications for RMA programming arise from the separation of communication (remote accesses) and synchronization. In addition, the RMA interface splits synchronization further into memory consistency (i.e., when a remote process observes a communicated value) and process synchronization (i.e., when a remote process "learns" about the state of another process). Furthermore, some RMA synchronizations can be nonblocking.

The main challenges of RMA programming revolve around the semantics of operation completion and memory consistency. Practical hardware implementations only provide 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 bridge this gap by offering sequential consistency at the language level if the programmer avoids data races. Whereas Java defines the behavior of programs containing races, C++11 leaves the semantics of programs with races unspecified so that implementations are less constrained and able to deliver higher performance.

MPI models completion, memory consistency, and process synchronization as separate concepts, which enables the user to reason about them separately. RMA programming is thus slightly more complex due to interactions between operations. For example, MPI, like most RMA programming models, enables 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 more complex. In fact, in the MPI RMA model, all communication operations are nonblocking; in other words, the communication functions may return before the operation completes. Bulk synchronization functions can be 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 (e.g., domain scientists); rather, writers of high-level libraries can provide domain-specific extensions that hide most of the complexity. The MPI RMA interface aims to enable 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. However, as was the case for the MPI-2 RMA interface, there are rules that can be followed by programmers that will ensure correct behavior. These are sufficient but not necessary rules that may sacrifice some performance or expressivity for simplicity. This article focuses on the full potential and power of the MPI RMA interface; the discussion here can be used to describe subsets of the MPI RMA interface that may be easier for average programmers to use and understand.

2. SEMANTICS AND ARCHITECTURAL CONSIDERATIONS

We now proceed to discuss the conceptual underpinnings of MPI RMA programming. Two central concepts of MPI RMA are memory regions and process groups. An MPI *window* binds a memory region at a process to a group of processes. The window's process group is identical to the process group of communicator that was used to create the window. This mechanism enables two types of spatial isolation: (1) processes outside the group cannot access memory that is exposed within the group, and (2) memory that is not attached to an MPI window cannot be accessed by remote processes, even in the same group. Both principles are important for parallel software engineering. They simplify the development and maintenance of parallel programs by offering an additional separation of concerns; that is, nonexposed memory cannot be corrupted by remote processes. They also enable 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 not interfere with other libraries or user code [Hoefler and Snir 2011].

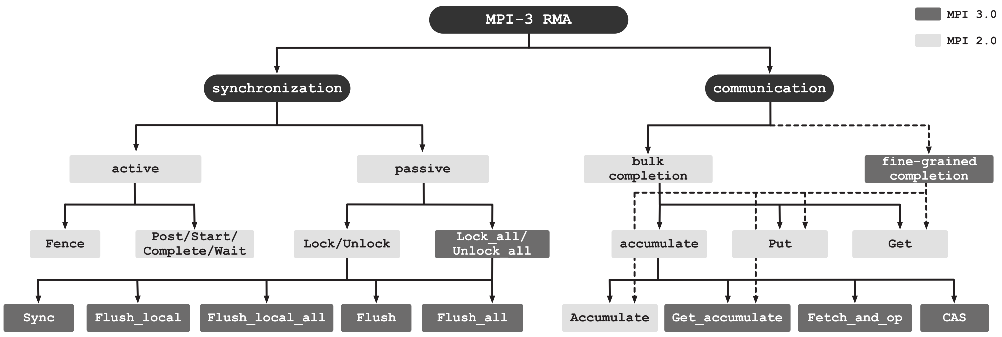


Fig. 1. Overview of communication options in the MPI-3 specification.

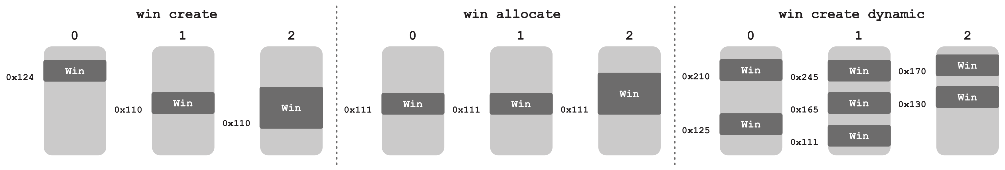


Fig. 2. MPI-3 memory window creation variants.

MPI RMA offers the basic data-movement operations *put* and *get* and additional predefined atomic operations called *accumulate*. Put and get are designed to enable direct usage of the shared memory subsystem or hardware-enabled RDMA. Accumulates require computation, but they can, in some cases, also use hardware acceleration directly. All communication functions are nonblocking and are completed by using either bulk completion functions or request-based completion. Bulk completions are generally faster than separate request-based completions. Figure 1 shows an overview of communication and synchronization functions in the MPI specification. We explain each function in the following sections.

2.1. 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 fourth 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.

The first (legacy) variant is the normal *win create* function: each process specifies an arbitrary consecutive memory region (≥ 0 bytes) to be exposed and a communicator from which the process group is derived. The function returns an opaque window object that can be used for remote accesses. Remote accesses are addressed 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 supports the specification of the least addressable unit in each window (called the *displacement unit*). The fact that processes can attach the window to consecutive memory regions at arbitrary addresses may lead to large translation tables on systems that offer RDMA functions. These tables may be distributed [Mellor-Crummey et al. 2009], but the necessary remote lookups may reduce performance.

The second creation function, *win allocate*, transfers the responsibility for memory allocation to MPI. RDMA networks that require large translation tables for win create

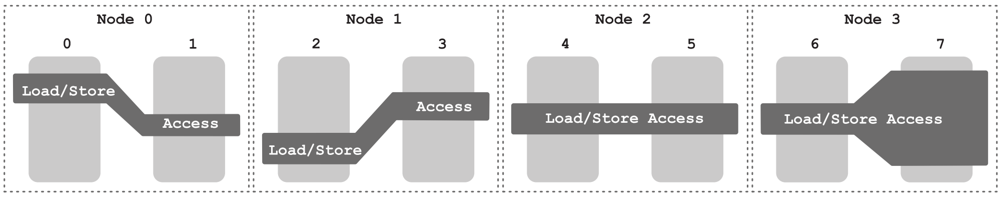


Fig. 3. Example MPI-3 shared memory window layout for a job running on four dual-core nodes. Each node has its own window that supports load/store and RMA accesses. The different shapes indicate that each process can pick its local window address and size independently of other processes.

may be able to avoid such tables by allocating memory at identical addresses on all processes (sometimes called *symmetric allocation* [Gerstenberger et al. 2013]). Otherwise, the semantics are identical to the win create function.

The third creation function, *create dynamic*, does not expose memory in the created window. Instead, it binds only a process group where each process can use subsequent local function calls for exposing memory in the window. This mode naturally maps to many RDMA network architectures; however, it may be more expensive than allocated windows because the MPI library may need to maintain additional structures for each exposed memory region. This mode can, however, be used for more dynamic programs that require process-local memory management, such as dynamically sized hash tables or object-oriented languages.

The fourth and last creation function, *shared memory window allocation*, enables processes to directly map memory regions into the address space of other processes. For example, if an MPI job is running on multicore nodes, then each process could share its memory directly with all other processes on the same node. 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.

The helper function *comm split type* enables programmers to determine groups of processes that enable 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 for a job running on four dual-core nodes.

Just like a process can be in multiple communicators, it can also expose multiple windows with different exposure calls. The MPI standard also allows a process to expose overlapping memory in different windows.

2.2. 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 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 weakly consistent or noncoherent memory systems.

2.3. 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. Larger types, such as double complex numbers, may not be supported in hardware. In this case, the MPI library can use software locking of the remote window to guarantee atomicity.

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 strong consistency at the granularity of primitive datatypes by default, which is most convenient for programmers but may come at a cost to performance. However, the strong consistency 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.

2.4. Request-Based Operations

Bulk local completion of communications has the advantage that no handles need to be maintained 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 resources and thus need to be able to complete specific messages. For such cases, request-based operations, prefixed with `R` (e.g., `MPI_Rput`), can be used in passive mode. These operations return a *request* object similar to nonblocking point-to-point communication that can be tested or awaited for completion using test or wait functions. Completion refers only to local completion in this context: for request-based put and accumulate operations, local completion means that the local buffer may be reused. For request-based get and get accumulate operations, local completion means that the remote data has been delivered to the local buffer.

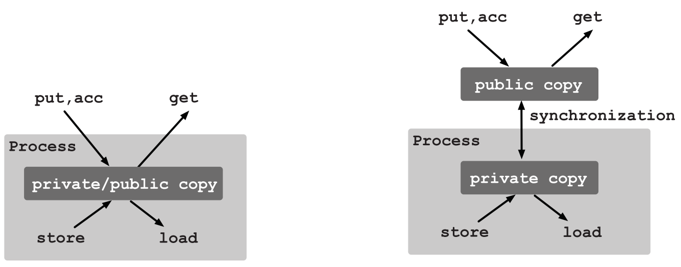
Request-based operations are useful when the application issues a number of nonblocking RMA operations and waits for the completion of a subset of them before it can continue computation. An application may start several get operations and compute the data in the order of their completion (see Listing 1). Note that the completion order may be different from the order in which the operations were started.

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

2.5. Memory Models

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 conceptually 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) target the public copy of the window.

The *separate* memory model assumes systems where coherency is managed by software. In this model, remote updates target the public copy and loads/stores target the



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

Fig. 4. Unified and separate memory models.

```
int main(int argc, char **argv) {
      double buf[100][1000];

      ... /* MPI initialization and window creation */

      for (i = 0; i < 100; i++)
          MPI_Rget(buf[i], 1000, MPI_DOUBLE, ..., &req[i]);

      for (i = 0; i < 100; i++) {
           MPI_Waitany(100, req, &idx, MPI_STATUS_IGNORE);
           process_data(buf[idx]);
       }

       ... /* Window free and MPI finalization */
       return 0;
}
```

Listing 1. Example (pseudo) code for using request-based operations.

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. In other words, 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. These rules force the programmer to perform separate synchronization.

The *unified* memory model relies on hardware-managed coherence. 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 similar to cache-coherence protocols on multicore CPUs and memory semantics of RDMA networks that propagate writes to their destination eventually. It enables programmers to exploit the whole performance potential of architectures in which both the processor and network provide such progress guarantees. Moreover, it places a lower burden on the programmer, as it requires less explicit synchronization. Figure 4 shows a comparison between the two memory models.

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 the separate memory model is more portable but may require additional synchronization calls.

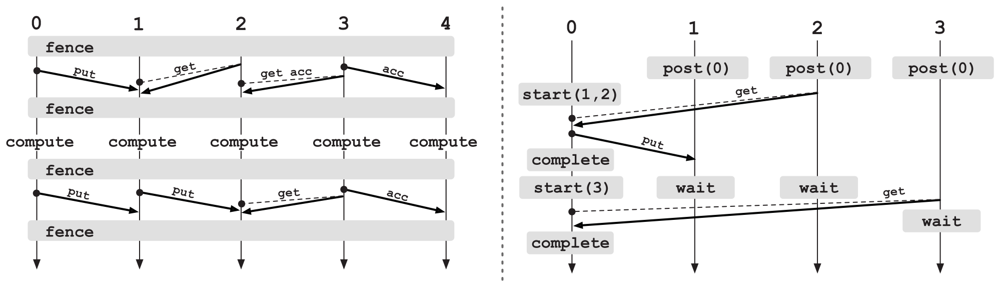


Fig. 5. Active target synchronization: fence mode for bulk-synchronous applications (left) and scalable active target mode for sparse applications (right).

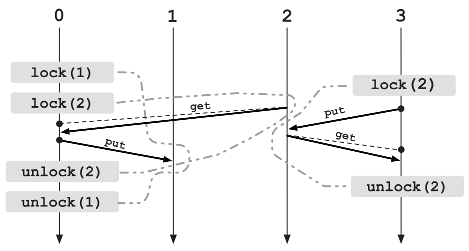
2.6. Synchronization

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 (the various completion calls are discussed later). Epochs can conceptually be divided into *access* and *exposure* epochs: the process-local window memory can only be accessed remotely if the process is in an exposure epoch, and a process can only access remote memory when it is in an access epoch. Naturally, a process can be simultaneously in access and exposure epochs.

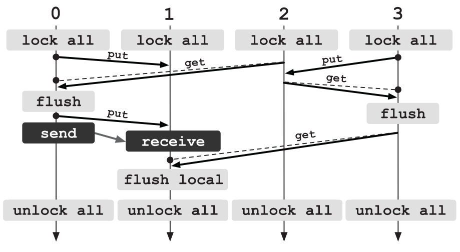
MPI offers two main synchronization modes based on the involvement of the target process: active target synchronization and passive target synchronization. In *active target* synchronization, the target processes expose their memory in exposure epochs and thus participate in process synchronization. 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 tailored to different use cases. Active target synchronization supports bulk-synchronous applications with a relatively static communication pattern, whereas passive target synchronization is best suited for random accesses with quickly changing target processes.

2.6.1. 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, as it does not involve synchronization among all processes. Exposure epochs begin with a call to post (which exposes the window memory to a selected group) and complete 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 and few communication partners, such as many stencil access applications [Datta et al. 2008]. Figure 5 shows example executions for both active target modes. The dashed



(a) Lock/unlock epochs



(b) Lock all epoch

Fig. 6. Passive target mode examples.

lines between processes represent get request commands, and the solid lines between processes represent data transfers.

2.6.2. Passive Target Synchronization. The concept of exposure epoch is not relevant in passive mode, as 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 between local load/store and remote RMA accesses (see Section 3), 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). Acquiring a lock guarantees local memory consistency, and releasing a lock guarantees remote consistency as well. More semantic details are explained in Section 3. Figure 6(a) shows an example with multiple lock/unlock epochs and remote accesses. The gray dashed-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 does not have to wait for the lock to be acquired (in this case, accesses have to be buffered). Lock is also one sided; thus, it does not enforce an ordering of epochs across processes, and access epochs to process 2 in Figure 6(a) can occur in the order shown or in the reverse order.

In the global lock model, each process starts a *lock all* epoch to all other processes. As opposed to fence, lock all is not collective. Lock all epochs are shared locks by definition. 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 4.3), can be implemented in this mode. Figure 6(b) shows an example of a lock all epoch with several communications and flushes. A flush ends the current epoch and ensures memory consistency. MPI also allows mixing both lock modes and point-to-point communication freely.

3. SEMIFORMAL 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 with sufficient precision to enable other researchers to derive a formal specification of the MPI-3 RMA memory models and develop valid transformations for such programs.[1] For this purpose, we follow the conventions from Manson et al. [2005] and Boehm and Adve [2008]. Yet, the level of detail creates complex interactions and readers who are

[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 foster human reasoning.

primarily interested in gaining an intuitive understanding about the semantics for practical use may skip to Section 4, where we present use cases and examples.

Formal semantics can be used for proving consistency of the standard; indeed, we found two issues in MPI-3 while establishing the models: (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 concisely describe corner cases as *litmus tests* [Mador-Haim et al. 2011], many of which resolve themselves after formalization. In addition, applications written in MPI-3 RMA could be verified for correctness and determinism similar to programs on multicore CPUs [Torlak et al. 2010]. Formal semantics could also be used to design and verify semantics-preserving compiler transformations [Boehm 2005].

MPI's memory semantics are specified in terms of regions of exposed memory called *MPI windows*. Each MPI RMA call is constrained to target a single window. Each such 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. Our model considers only operations on memory associated with a window. MPI RMA comprises memory operations and synchronization operations. Window memory can also be accessed by the program using local load and store operations (induced by statements in the source code or MPI operations). To simplify our notation, we assume that each primitive datatype occupies distinct memory locations in a window (with a per-byte granularity). We also assume that all operations are aligned at multiples of the size of the accessed primitive datatype.

A correct MPI RMA program is a program where each conflicting access is synchronized with process synchronization (we call this *happens before*) as well as memory synchronization (we call this *consistency*). In addition, correct MPI programs do not deadlock or livelock. MPI offers different synchronization calls to achieve both orders. Our semantics consider executions of programs. First, we introduce two initial types of actions: memory and synchronization. Following Manson et al. [2005] and Boehm and Adve [2008], we define a *memory action* as the tuple

$$\langle a, o, t, rl, wl, u, p\rangle$$

with the following elements:

- a: [$\underline{a}$ction type] can be one of the following: local store (ls), local load (ll), remote communication put (rcp), remote communication get (rcg), remote accumulate get (rag, with the special case fetch and op), remote accumulate (rac), or remote accumulate compare and swap (ras).
- o: [$\underline{o}$rigin] contains the MPI process rank for the origin of the action.
- t: [$\underline{t}$arget] contains the MPI process rank for the destination of the action. Actions of type ll and ls can have only the local process as destination.
- rl: [$\underline{r}$ead $\underline{l}$ocation] contains the location read by the action. This is not specified for ls and is a tuple of the form ⟨compare location, swap location⟩ for ras.
- wl: [$\underline{w}$rite $\underline{l}$ocation] contains the location written by the action (not specified for ll).
- u: an arbitrary [$\underline{u}$nique] identifier for the action.
- p: [source $\underline{p}$oint] is a label identifying the source program point.

Similarly, a *synchronization action* is defined as the tuple

$$\langle a, o, t, u, p\rangle$$

with the following elements:

a: [action type] can be one of the following: fence (*sfe*), lock shared (*sls*), lock exclusive (*sle*), unlock (*sul*), lock all (*sla*), unlock all (*sula*), flush (*sfl*), flush local (*sfll*), flush all (*sfla*), flush local all (*sflla*), win-sync (*sws*), and external synchronization (*ses*, e.g., matching send/recv pairs or collective operations).
o: [origin] contains the process rank for the origin of the action.
t: [target] contains the process rank for the destination of the action. The actions *sla*, *sula*, *sfla*, *sflla* have a special identifier $\star$ as destination, which stands for the entire set of processes associated with the window.
u: an arbitrary [unique] identifier for the action.
p: [source point] is a label identifying the source program point.

We omit the generalized active target synchronization and request-based RMA operations 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, adding significant complexity and jeopardizing our goal of readability.

For brevity, we name actions of type z just as z instead of "x where $x.a = z$". For example, *sle* reads "the action *sle* of type lock exclusive." We use the notation $v = (x|y)$ to indicate that v is either of type x or y. For example, we write $(sls|sle)$ to denote a single action of type lock shared or lock exclusive. Ordering relations such as $(sls|sle) \rightarrow (sls|sle)$ can be read like "an action of type lock shared or lock exclusive is ordered with another action of type lock shared or lock exclusive." That means that the expression orders two actions of all possible type combinations, such as $sls \rightarrow sls, sle \rightarrow sle, sle \rightarrow sls, sls \rightarrow sle$.

We define the following abbreviations for common action types: a put/get action $rc* := (rcp|rcg)$, a remote accumulate action $ra* := (rac|rag|ras)$, a remote memory action $r* := (rc*|ra*)$, a local memory action $l* := (ll|ls)$, and a synchronization action $s* := (sfe|sls|sle|sul|sla|sula|sfl|sfll|sfla|sflla|sws|ses)$.

In addition, remote memory actions $r*$ as well as unlock actions $(sul|sula)$ and remote flush actions $(sfl|sfla)$ create virtual actions $(vac|vas)$ at the target process. A virtual communication action *vac* represents the event when the effect of the communication action takes place at the target, and a virtual synchronization action *vas* represents the event when a synchronization action takes place at the target. For example, for an *rcp*, the corresponding *vac* is the action that commits the data to the destination memory, and for an *rcg*, it is the action that reads the data from destination memory. Virtual actions are necessary to define consistency and process order for purely one-sided remote events. We arbitrary abbreviate virtual actions as $va* = (vac|vas)$.

We can now define the execution of a program as a set of actions, orders, and functions

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

where

P: is the program to be executed;
A: is the set of all actions (types $s*|r*|l*|va*$);
$\xrightarrow{po}$ specifies a total order (program order) of actions $(s*|r*|l*)$ at the same process much like the "sequenced-before" order in C++ or the "program order" in Java, which specifies the order of executions in a single-threaded execution;
W: is a function that returns the store (ls), remote put (rcp), or remote accumulate ($ra*$) 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 partial order of the synchronization relations of process synchronization and virtual actions ($s*|va*$) including external waiting-for relationships (e.g., arise from *ses* actions like collective operations and matched send/recv pairs);

$\xrightarrow{hb}$ is a transitive relation between pairs of actions, in which the relation $\xrightarrow{hb}$ is the transitive closure of the union of $\xrightarrow{po}$ and $\xrightarrow{so}$; and

$\xrightarrow{co}$ is a partial order of memory actions and virtual actions ($r*|l*|va*$), in which a consistency edge $x \xrightarrow{co} y$ guarantees that the memory effects of action x are visible to y.

Consistency order $\xrightarrow{co}$ and happens-before order $\xrightarrow{hb}$ are representing the memory consistency and process synchronization concepts, respectively. If $x \xrightarrow{co} y$ in an execution, then y observes the effect of x. If y originates at a different process than x, then unbounded network latencies may delay the commit and it is generally not specified when y commits. Thus, $\xrightarrow{co}$ alone cannot guarantee consistent semantics. However, MPI specifies that $x \xrightarrow{co} y$ implies that if y happens arbitrary late (or is repeated), then y observes the effect of x *eventually* in unified memory windows; this guarantee is needed for polling and does not require an $\xrightarrow{hb}$ ordering (see Section 3.5 for an example). To establish a guaranteed order between actions on different processes, MPI specifies process synchronization operations that imply $\xrightarrow{so}$ by delaying a process's execution until a matching synchronization is issued and has arrived. Both $\xrightarrow{co}$, which only ensures synchronized memory, and $\xrightarrow{hb}$ which only ensures synchronized process actions, have to be combined to enforce correct executions. Indeed, the active mode synchronization primitives guarantee both orders. However, some complex synchronization mechanisms allow the user to separate the two orders. We thus abbreviate a consistent *happens-before order* between two actions x and y as

$$x \xrightarrow{cohb} y := x \xrightarrow{hb} y \wedge x \xrightarrow{co} y. \tag{1}$$

For two actions x and y and an order of type $O \in \{\xrightarrow{po}, \xrightarrow{so}, \xrightarrow{hb}, \xrightarrow{co}\}$, we denote that x and y are not ordered by O as

$$x \parallel_O y := \neg(x \xrightarrow{O} y \vee y \xrightarrow{O} x). \tag{2}$$

For example, $x \parallel_{hb} y$ means that x and y happen concurrently in happens-before order, and $x \nparallel_{hb} y$ means that x and y are ordered in happens-before order. Unless otherwise stated, all relations and orders assume actions at the same process.

3.1. Valid Executions and Process Synchronization

We now specify valid programs in terms of their executions. In a correct and deadlock-free MPI program, all possible executions must be valid. An execution is valid under the following conditions:

(1) Each action is generated by executing a program statement, and all actions occur in an order ($\xrightarrow{po}$) consistent with the control flow of each process in the program.
(2) Passive target (lock/unlock) must be called 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). Let

$$[x..y] := \{z : x \xrightarrow{po} z \xrightarrow{po} y\}. \tag{3}$$

Formally, for an action $x = (sls|sle)$,

$$\exists\, y : x \xrightarrow{po} y \wedge y.a = sul \wedge x.t = y.t \wedge \forall\, z \in [x..y], z.a \neq sfe \wedge z.a \in \{sls, sle\} \Rightarrow z.t \neq x.t \quad (4)$$

and similarly, when $x = sla$,

$$\exists\, y : x \xrightarrow{po} y \wedge y.a = sula \wedge \forall\, z \in [x..y], z.a \notin \{sfe, sls, sle, sul\}. \quad (5)$$

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

(4) Windows may not be locked and exposed concurrently such that for an action $x = (sls|sle|sla)$: $x \nparallel_{hb} sfe$ and for all $x \xrightarrow{hb} sfe$ there exists an unlock y where $y.a \in \{sul, sula\}$ that matches x and $y \xrightarrow{hb} sfe$.

(5) For actions $sfe^0 \xrightarrow{po} r* \xrightarrow{po} sfe^1$, sfe^0 may only open a fence epoch when the `MPI_MODE_NOSUCCEED` assertion is not given and

$$s* \notin [sfe^0..r*]. \quad (6)$$

The sfe^1 action may only close a fence epoch when the `MPI_MODE_NOPRECEDE` assertion is not given and

$$s* \notin [r*..sfe^1]. \quad (7)$$

(6) The synchronization actions $(sfl|sfla|sfll|sflla)$ can only be called in passive target mode; that is,

$$\exists y = (sls|sle|sla) : y \xrightarrow{po} x \xrightarrow{po} (sfl|sfla|sfll|sflla) \wedge x.a \neq sfe. \quad (8)$$

(7) The program is deadlock free; that is, the directed graph $G = (A, \xrightarrow{hb})$ contains no cycles (this excludes the synchronization orders introduced by matching fences, single unlocks, and single flushes).

(8) For each remote memory action $r*$, the origin process $r*.o$ is in an epoch of type access (see Section 3.2). The target process $r*.t$ is in an epoch of type exposure if the accessing origin process's last synchronization operation (in $\xrightarrow{po}$) was of type sfe.

The orders $\xrightarrow{po}$ and $\xrightarrow{so}$, and thus $\xrightarrow{hb}$, 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.

3.2. Epochs, Synchronization, and Consistency

Synchronization operations logically split the execution at each process into *epochs*. Epochs are a concept to enforce consistency for local and remote operations. We now define all significant interactions between all $(r*|l*|s*)$ actions of valid executions with regard to process synchronization $\xrightarrow{so}$, program order $\xrightarrow{po}$, and consistency order $\xrightarrow{co}$.

3.2.1. Epochs. Epochs have a total order per process for a given pair of window and target arguments 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 both simultaneously. Each epoch starts with a synchronization action $(sfe|sle|sls|sla)$ and ends with a *locally matching* synchronization action $(sfe|sul|sula)$. Two synchronization operations sfe match locally if no other sfe occurs between them, sla matches the next $sula$ in $\xrightarrow{po}$ and vice versa, and sle or sls match the next sul (with the same target process) in $\xrightarrow{po}$ and vice versa. A synchronization action $(sfl|sfla)$ has the effect of closing and opening an epoch.

Each memory action $x = (r*|l*)$ happens in exactly one epoch that is bounded by matching synchronization actions (in $\xrightarrow{po}$). Two epochs u and v are ordered by $\xrightarrow{hb}$ if the ending synchronization action $s*_u$ of u is ordered with the starting synchronization action $s*_v$ of v as $s*_u \xrightarrow{hb} s*_v$.

3.2.2. Virtual Actions. Virtual actions belong to a real action in the trace. However, virtual actions happen at a different process than the originating action. They indicate when the effect of a real action commits at the target process.

Each virtual synchronization action vas is part of a real synchronization action $s*$, and it is coherent as well as synchronized with its originating action. We abbreviate coherency as well as synchronization order between two actions x and y as $x \xrightarrow{coso} y = x \xrightarrow{co} y \wedge x \xrightarrow{so} y$. Since it blocks until all messages commit remotely, it introduces orders in both directions. To model this, we split the $s*$ action into start $s*_s$ and end action $s*_e$ and define $s*_s \xrightarrow{coso} vas$ and $vas \xrightarrow{coso} s*_e$. We abbreviate this split action for brevity with $s* \xrightarrow{coso} vas$. Each virtual communication action vac is part of a real communication action $r*$, and it is coherent as well as synchronized with its originating action. We also split communication actions into start and end parts $r*_s$ and $r*_e$ and define $r*_s \xrightarrow{coso} vac$ and $vac \xrightarrow{coso} r*_e$ abbreviated as $r* \xleftrightarrow{coso} vac$. A virtual action $va*$ happens at the target $(s*|r*).t$ of its originating action $(s*|r*)$. The location where a virtual action happens is its origin $va*.o$.

3.2.3. Remote Memory Actions and Local Effects. Remote memory actions $r*$ cause local load/store operations to read or write the data at the origin $r*.o$ and a virtual remote communication action at $r*.t$ when the action takes place at the target. Thus, each $r*$ can be modeled by a set of $l*$ and the virtual communication action vac. The action vac acts on the public window copy at the target, whereas $l*$ act on the private window copy. Again, we assume that $r*$ is split into start and end actions. We define the following optional replacement rules to model local and remote effects of communication actions: $rcg_e = ls$, $rcp_s = ll$, $rac_s = ll$, $ras_s = ll$, $ras_e = ls$, $rag_s = ll$, and $rag_e = ls$. However, we keep the remote actions $r*$ in the following to define their interaction with locks and flushes precisely.

Some synchronization operations complete remote actions $r*$, whereas others only enforce consistency of local actions $l*$. Thus, we will consider interactions between local memory actions $l*$, remote memory actions $r*$, and virtual actions $(vas|vac)$ with the synchronization actions $s*$ in the following.

3.2.4. Active Target Synchronization. The active target synchronization mode uses sfe actions to collectively transition all processes from epoch i to epoch $i+1$. A fence can essentially be seen as a single action invoked by all processes in the group associated with the window. It introduces $\xrightarrow{co}$ and $\xrightarrow{so}$ between *remote matching* fences at all pairs of processes i and j. As for the virtual action, we split sfe into start and end actions and the orders of actions between arbitrary processes i and j $sfe_s^i \xrightarrow{co} sfe_e^j$ and $sfe_s^i \xrightarrow{so} sfe_e^j$. As for virtual actions, we abbreviate the orders and split using $sfe^i \xleftrightarrow{coso} sfe^j$ in the following. Fences match remotely in the order of issuing in the window; that is, the k^{th} fence on process j matches the k^{th} fence on process i. In addition, a fence guarantees local consistency for all $\xrightarrow{hb}$-ordered load/store and virtual communication actions

$$(r*|l*|vac) \xrightarrow{hb} sfe \Rightarrow (r*|l*|vac) \xrightarrow{co} sfe \text{ and } sfe \xrightarrow{hb} (r*|l*|vac) \Rightarrow sfe \xrightarrow{co} (r*|l*|vac). \quad (9)$$

3.2.5. 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 to a single process j is opened at process i after an action $(sls|sle).t = j \wedge (sls|sle).o = i$ and ends with a $sul.t = j \wedge sul.o = i$ action. A process-local access epoch to all processes is opened at process i by an action $sla.o = i$ and ends with action $sula.o = i$.

Lock / Unlock. Lock operations can be either shared or exclusive. In a valid execution, a shared lock $(sls|sla)$ is ordered by a synchronization order $sul \xrightarrow{so} sls$ with all previous unlocks sul for which $sls.t = sul.t$ and sul is matching an exclusive lock sle. An exclusive lock sle is ordered by a synchronization order $(sul|sula) \xrightarrow{so} sle$ with all previous unlocks sul for which $sle.t = sul.t$ and all previous $sula$. Lock actions guarantee local consistency

$$(sls|sle|sla) \xrightarrow{hb} l* \Rightarrow (sls|sle|sla) \xrightarrow{co} l* \text{ if } (sls|sle|sla).o = l*.o \tag{10}$$

and

$$vas \xrightarrow{hb} (sls|sle|sla) \Rightarrow vas \xrightarrow{co} (sls|sle|sla) \text{ if } (sls|sle|sla).o = vas.o. \tag{11}$$

Unlock completes remote actions locally

$$r* \xrightarrow{po} (sul|sula) \Rightarrow r* \xrightarrow{co} (sul|sula) \tag{12}$$

and also guarantees that local memory is consistent, such that

$$(sul|sula) \xrightarrow{po} l* \Rightarrow (sul|sula) \xrightarrow{co} l* \tag{13}$$

if and only if the value accessed by $l*$ was accessed by an $r*$ action (its related ls) after the matching $(sle|sls)$ and $sul.t = r*.t$. Unlock also propagates updates to the public window

$$ls \xrightarrow{po} (sul|sula) \Rightarrow ls \xrightarrow{co} (sul|sula) \text{ and } (sul|sula) \xrightarrow{po} vas \Rightarrow (sul|sula) \xrightarrow{co} vas. \tag{14}$$

In addition, an unlock completes actions remotely and generates a virtual action vas at $sul.t$ that synchronizes the memory access to the remote public window

$$vas \xrightarrow{hb} vac \Rightarrow vas \xrightarrow{co} vac \text{ and } vac \xrightarrow{hb} vas \Rightarrow vac \xrightarrow{co} vas \tag{15}$$

if $vac.o = vas.o$.

Flush. A flush can be used to synchronize RMAs. A flush sfl has the same semantics as an unlock. It also generates a virtual synchronization action vas at its target with the same rules as stated in Equation (15). A flush all $sfla$ behaves like a flush to all processes. Flush local $sfll$ completes remote operations locally:

$$r* \xrightarrow{po} sfll \Rightarrow r* \xrightarrow{co} sfll \text{ if } r*.t = sfll.t. \tag{16}$$

Flush local all completes all remote operations locally:

$$r* \xrightarrow{po} sflla \Rightarrow r* \xrightarrow{co} sflla. \tag{17}$$

Flush local does not create a virtual action.

Sync. A win sync call sws has the effect synchronizing remote and local accesses.

$$(l*|vac) \xrightarrow{hb} sws \Rightarrow (l*|vac) \xrightarrow{co} sws \text{ and } sws \xrightarrow{hb} (l*|vac) \Rightarrow sws \xrightarrow{co} (l*|vac). \tag{18}$$

Local load / Store. For normal reads and writes interacting with RMA calls,

$$l* \xrightarrow{po} r* \Rightarrow l* \xrightarrow{co} r*. \tag{19}$$

Consistent remote operations. For virtual communication actions in the unified memory window,

$$l* \xrightarrow{hb} vac \Rightarrow l* \xrightarrow{co} vac. \tag{20}$$

3.3. Conflicts and Races

We now specify the rules for defined memory operations in MPI RMA. Those are needed to reason about the possible result of a series of memory actions originating at different processes.

3.3.1. Conflicting Actions. In the separate memory model (see Section 2.5), two memory actions x and y are called *conflicting* if they are directed toward overlapping memory locations at the same process and either (1) one of the two operations is a put rcp, (2) exactly one of the operations is an accumulate ($ra*$), or (3) one operation is a get (rcg) and the second one a local store (ls). In addition, remote writing operations (rcp and $ra*$) that access the same process conflict with local store (ls) operations issued by the target process regardless of the accessed location.

In the unified model, two actions x and y are called *conflicting* if they are directed toward overlapping memory locations at the same process and either (1) one of the two operations is a put (rcp), (2) exactly one of the operations is an accumulate ($ra*$), or (3) one operation is a get (rcg) and the second one a local store (ls).

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

$$\neg(x \xrightarrow{cohb} y \vee y \xrightarrow{cohb} x); \tag{21}$$

that is,

$$x \parallel_{hb} y \vee x \parallel_{co} y. \tag{22}$$

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

3.3.3. Conditions for Well-Defined Memory Semantics. Only data race–free executions have well-defined memory semantics. If an execution has well-defined semantics, then a read action ll will always return the last written value (last as defined by the consistent happens-before order):

$$W((ll|vac)) \xrightarrow{cohb} (ll|vac) \wedge V((ll|vac)) = V(W((ll|vac))). \tag{23}$$

In addition, the following property is guaranteed:

$$W((ll|vac)) \xrightarrow{cohb} ls \xrightarrow{cohb} (ll|vac), \text{ then } (ll|vac).rl \neq ls.wl \wedge V((ll|vac)) = V(W((ll|vac))). \tag{24}$$

In other words, in a program with well-defined memory semantics, for every read action $(ll|vac)$,

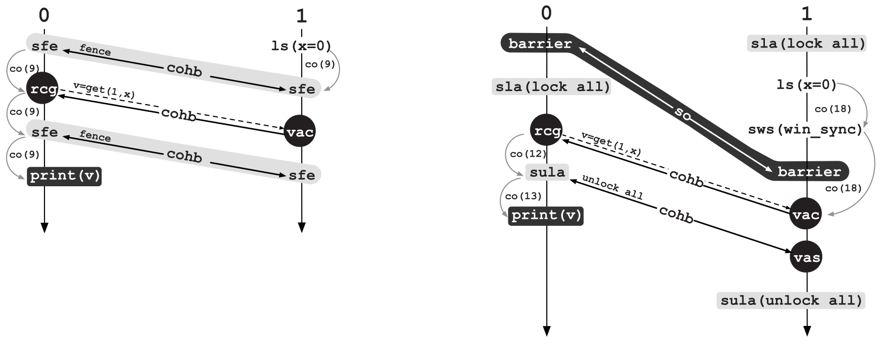
$$\neg((ll|vac) \xrightarrow{cohb} W((ll|vac))). \tag{25}$$

3.4. Ordering Rules for Accumulates

Let x and y be of type $ra*$ and update the same variable:

$$x.wl = y.wl \wedge x \xrightarrow{po} y \Rightarrow x \xrightarrow{co} y. \tag{26}$$

However, the user can relax any of the possible combinations of write and read ordering (`waw`, `war`, `raw`, `rar`). For local ls and ll memory actions and the local reads and writes



(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.

associated with $r*$ actions, we assume sequential ordering. The effects cannot be observed remotely, as no consistent ordering exists for those operations; thus, many local compiler transformations (e.g., changing the order of instructions) that do not modify sequential correctness are possible. Remote put and get $rc*$ actions and $r*$ actions with different destination addresses or processes have no specified ordering.

3.5. Eventual Remote Completion

The unified memory model enables 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 $r*$ action targeted at process B, and process B could wait in an infinite loop for the arrival for the message.

The *vac* action that is generated on process B will not have a happens-before relation, whereas it will have a $(sul|sfl|sfla) \xrightarrow{co} vac$. If process B waits (potentially an unbounded number of steps) for the message to arrive (by polling on $r*.wl$), it is guaranteed that the message will eventually arrive; that is, a consistent happens-before relation will be established between the $(sul|sfl|sfla)$ and one of the polling reads. However, MPI provides no timing guarantees, and thus the process may need to wait for an unbounded number of steps.

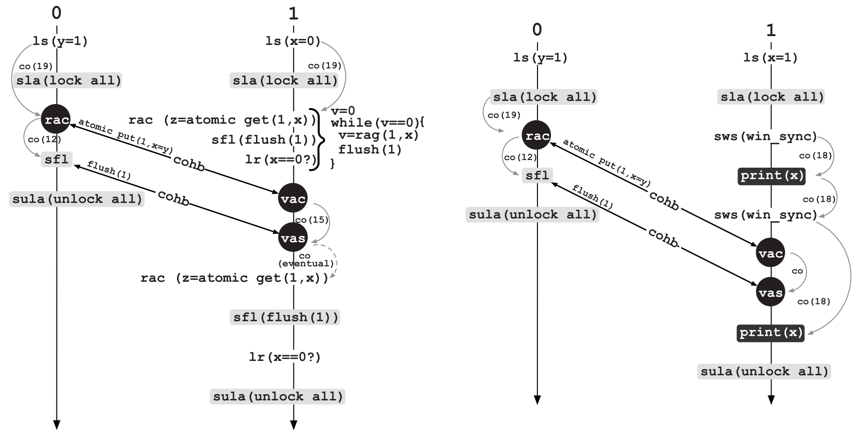
3.6. Shared Memory Windows

All of the preceding discussions apply to shared memory windows. As stated earlier, however, there are no guarantees about the consistency order of *ll* and *ls* actions (which can now be observed directly by remote processes), as this is a function of the architecture's memory model (e.g., x86 [Owens et al. 2009] or POWER [Adve and Gharachorloo 1996; Sarkar et al. 2012]).

3.7. 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 program order ($\xrightarrow{po}$). Each statement at a process is ordered with regard to the previous statements at the same 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 $\xrightarrow{cohb}$ ordering. Thus, the result of this example trace is defined, and the `print(v)` will always output "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 no-op), respectively.

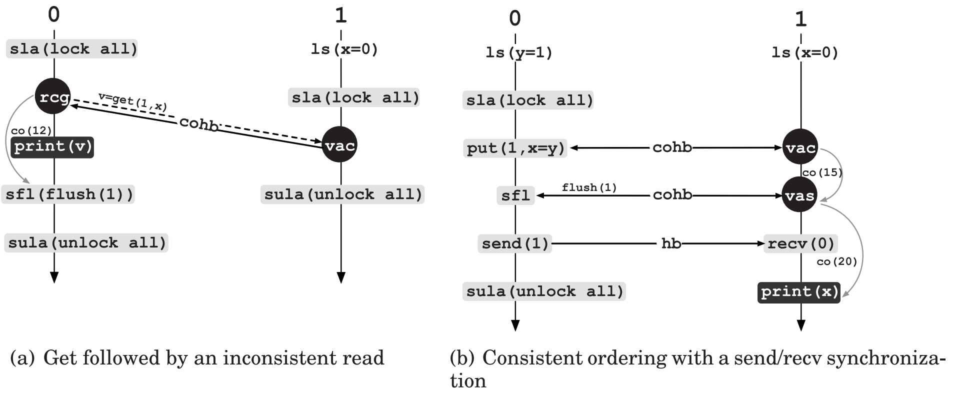
The numbers in brackets at the orders in all following figures state the rule number that generates this order.

The formal model also supports 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 1 (for the separate model, it is unnecessary in the unified model), and the unlock together with the *vas* and *vac* actions guarantees $\xrightarrow{co}$ order between the conflicting accesses. Thus, the memory semantics are well defined, and the read will always read “0.”

In Figure 8(a), process 0 puts a value into process 1’s window, which waits for the value’s arrival. In this example, $\xrightarrow{hb}$ is guaranteed to the virtual communication action (*vac*), which itself is not ordered with regard to the actions at process 1. However, since process 1 is in an infinite loop, *vac* will eventually appear in this loop and thus introduce an eventual $\xrightarrow{hb}$ and $\xrightarrow{co}$ ordering. Thus, this program is correct in the unified memory model, and v will have the value “1” at process 1 eventually. Figure 8(b) shows polling in the separate memory model. This schedule is undefined since the *vac* action can occur between a sync and a local load and may thus lead to undefined outcome of the local load.

Figure 9(a) shows an example where a consistency edge is missing for a local access. The accesses to x are conflicting on process 0, 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 x at process 1 are synchronized with a flush ($\xrightarrow{co}$) and with a send/recv pair ($\xrightarrow{hb}$). The outcome of this example is well defined to print “1.”

Figure 10 shows an example for a missing happens-before ordering. The $\xrightarrow{co}$ ordering at process 1 is established because of the stronger guarantees of the unified model; however, there is no $\xrightarrow{hb}$ ordering such that *vac* could execute concurrently with the write, making the outcome undefined.



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

Fig. 9. Examples for consistency ordering.

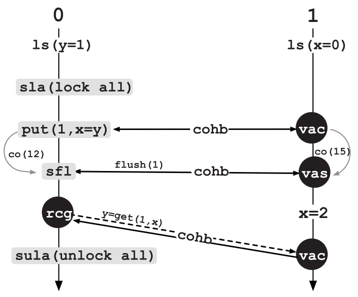


Fig. 10. Missing happens-before ordering.

4. USE CASES AND EXAMPLES

In this section, we discuss several possible uses of the new RMA 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 absence of 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.

4.1. 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 support 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

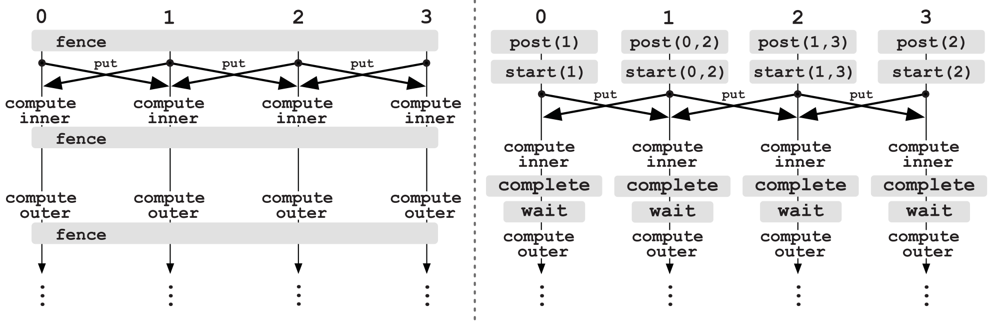


Fig. 11. 1D stencil boundary exchange example using fence (left) and general active target synchronization (right).

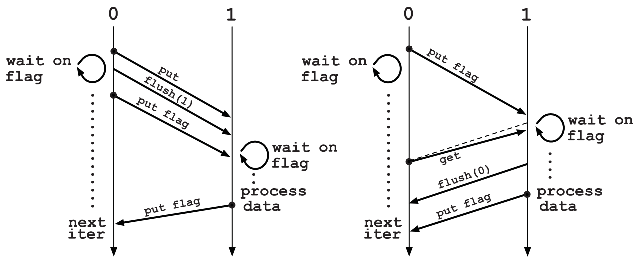


Fig. 12. Put and get protocols for passive target mode synchronization.

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 11 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.

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, flushes, 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 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, flushes, and processes the data. Both protocols require two remote accesses (or messages), and the better protocol depends on the put and get performance of the target system. Gerstenberger et al. [2013] demonstrated substantial speedups at large scale for the get-based protocol. Figure 12 shows put and get protocols for passive target synchronization.

4.2. 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 3D FFT ($X \times Y \times Z$) using a 1D data decomposition as a case study.

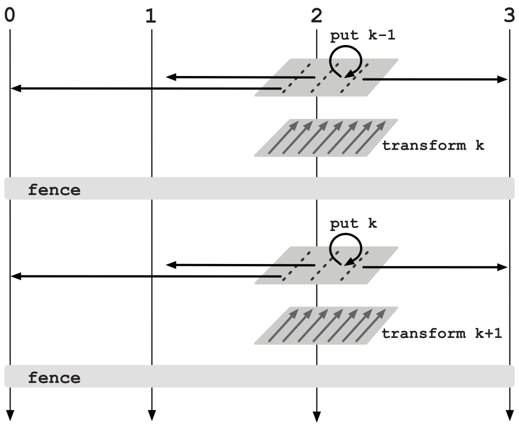


Fig. 13. Example computation of a 3D FFT.

In the decomposition, each process has a set of 2D 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 13 shows an example decomposition and execution of a 3D FFT.

4.3. 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 1991], 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 (displacement `ELEM_DISP` = 0), and the process hosting the mutex adds a second integer element that will be used as the tail pointer (byte displacement `TAIL_DISP` = `sizeof(int)`). Once the window has been created, all processes call `MPI_Win_lock_all` to initiate shared-mode access, as accesses will be performed by using only atomic operations.

As shown in Listing 2, when processes request the lock, they atomically exchange their rank, which acts as a pointer to their list element (initialized to `MPI_PROC_NULL`), with the tail pointer. If the tail pointer is `MPI_PROC_NULL`, the process has successfully acquired the lock. Otherwise, it updates the element of the process that was the old

```
/* This store cannot occur concurrently with a remote write */
mutex->base[ELEM_DISP] = MPI_PROC_NULL;
MPI_Win_sync(mutex->window);

MPI_Fetch_and_op(&rank, &prev, MPI_INT,        /* new-val, old-val, type, */
                 mutex->tail_rank, TAIL_DISP,  /* target rank, disp,      */
                 MPI_REPLACE, mutex->window);  /* op, and window handle   */
MPI_Win_flush(mutex->tail_rank, mutex->window);

/* If there was a previous tail, update their next pointer and wait for
 * notification.  Otherwise, the mutex was successfully acquired. */
if (prev != MPI_PROC_NULL) {
    MPI_Status status;

    MPI_Accumulate(&rank, 1, MPI_INT, /* src-val, src-count, src-type, */
                   prev, ELEM_DISP,   /* target rank, displacement,    */
                   1, MPI_INT,        /* dst-count, dst-type           */
                   MPI_REPLACE, mutex->window); /* op, and window hdl  */
    MPI_Win_flush(prev, mutex->window);
    MPI_Recv(NULL, 0, MPI_BYTE, prev, MCS_MUTEX_TAG, mutex->comm, &status);
}
```

Listing 2. MCS mutex lock acquire algorithm.

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.

Similarly, when releasing the lock, shown in Listing 3, 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 `MPI_PROC_NULL`. 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.

5. SUMMARY

In this article, we described the MPI-3 one-sided interface, presented the semantics in a semiformal way, and showed several use cases. This new interface is expected to deliver the highest performance on novel network architectures that offer RDMA access directly in hardware. Highly optimized implementations of MPI-2 RMA exist [Larsson Traff et al. 2000], but the interface has only been adopted in very limited settings. The new MPI-3 RMA interface enables an extremely efficient implementation on modern hardware [Gerstenberger et al. 2013] while offering 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 enables the programmer to choose between implicit notification in active target mode and explicit notification in passive target mode. This large variety of options enables users to create complex programs.

Our formalization of remote access semantics enables one to reason about complex applications written in MPI-3 RMA. This can be used to prove that 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 provided three examples: (1) a stencil exchange pattern to show a possible implementation of neighbor exchanges, (2) a fast Fourier transformation to demonstrate

```
/* Read my next pointer.  FOP is used since another process may write to
 * this location concurrent with this read. */
MPI_Fetch_and_op(NULL, &next, MPI_INT, rank, ELEM_DISP, MPI_NO_OP,
                 mutex->window);
MPI_Win_flush(rank, mutex->window);

if (next == MPI_PROC_NULL) {
    int tail, nil = MPI_PROC_NULL;

    /* Check if we are at the tail of the lock queue.  If so, we're
     * done.  If not, we need to send notification. */
    MPI_Compare_and_swap(&nil, &rank, &tail, /* new-val, cmpare-val, old-val */
                         MPI_INT, mutex->tail_rank, /* type, target rank     */
                         TAIL_DISP, mutex->window); /* displ, and window hdl */
    MPI_Win_flush(mutex->tail_rank, mutex->window);

    if (tail != rank) {
        for (;;) {
            int flag;

            MPI_Fetch_and_op(NULL, &next, MPI_INT, rank, ELEM_DISP,
                             MPI_NO_OP, mutex->window);
            MPI_Win_flush(rank, mutex->window);
            if (next != MPI_PROC_NULL) break;
}   }   }

/* Notify the next waiting process */
if (next != MPI_PROC_NULL) {
    MPI_Send(NULL, 0, MPI_BYTE, next, MCS_MUTEX_TAG, mutex->comm);
}
```

Listing 3. MCS mutex lock release algorithm.

the capabilities to overlap communication and computation, and (3) an MCS lock to demonstrate a more complex data structure in the passive mode model.

ACKNOWLEDGMENTS

We thank Martin Vechev and Andrei Dan for many helpful discussions and comments on the semantics. We thank the anonymous reviewers for their many insightful comments and valuable suggestions.

REFERENCES

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12, 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, Los Alamitos, CA, 83–87. DOI:http://dx.doi.org/10.1109/HOTI.2010.23

Hans-J. Boehm. 2005. Threads cannot be implemented as a library. *ACM SIGPLAN Notices* 40, 6, 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. *ACM SIGPLAN Notices* 43, 6, 68–78. DOI:http://dx.doi.org/10.1145/1379022.1375591

Philip Buonadonna, Andrew Geweke, and David Culler. 1998. An implementation and analysis of the virtual interface architecture. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (Supercomputing'98)*. IEEE, Los Alamitos, CA, USA, 1–15.

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, 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. *ACM SIGPLAN Notices* 40, 10, 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, 316–325. DOI:http://dx.doi.rg/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, Los Alamitos, CA, Article No. 4. http://dl.acm.org/citation.cfm?id=1413370.1413375

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, Los Alamitos, CA, Article No. 103. http://dl.acm.org/citation.cfm?id=2388996.2389136

Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Enabling highly-scalable remote memory access programming with MPI-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'13)*. ACM, New York, NY, Article No. 53. DOI:http://dx.doi.org/10.1145/2503210.2503286

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)*. 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, 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, Los Alamitos, CA, Article No. 98. 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)*. 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 2nd International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC'98)*. 189–201. http://dl.acm.org/citation.cfm?id=646092.680546

Jesper Larsson Traff, Hubert Ritzdorf, and Rolf Hempel. 2000. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (Supercomputing'00)*. IEEE, Los Alamitos, CA, Article No. 1. http://dl.acm.org/citation.cfm?id=370049.370878

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. 2011. Litmus tests for comparing memory consistency models: How long do they need to be? In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, New York, NY, 504–509. DOI:http://dx.doi.org/10.1145/2024724.2024842

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, 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 3rd Conference on Partitioned Global Address Space Programming Models (PGAS'09)*. ACM, New York, NY, Article No. 5. DOI:http://dx.doi.org/10.1145/1809961.1809969

John Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9, 1, 21–65. DOI:http://dx.doi.org/10.1145/103727.103729

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

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

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

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

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, Vol. 5674. Springer, 391–407. DOI:http://dx.doi.org/10.1007/978-3-642-03359-9_27

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and power. *ACM SIGPLAN Notices* 47, 6, 311–322. DOI: http://dx.doi.org/10.1145/2345156.2254102

Timo Schneider, Robert Gerstenberger, and Torsten Hoefler. 2013. Compiler optimizations for non-contiguous remote data movement. In *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing*.

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 (Supercomputing'01)*. ACM, New York, NY, 57–57. DOI: http://dx.doi.org/10.1145/582034.582091

Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking axiomatic specifications of memory models. *ACM SIGPLAN Notices* 45, 6, 341–350. DOI: http://dx.doi.org/10.1145/1809028.1806635

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

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

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, 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)*. 76–85. DOI: http://dx.doi.org/10.1007/11846802_18

Chaoran Yang, Wesley Bland, John Mellor-Crummey, and Pavan Balaji. 2014. Portable, MPI-interoperable Coarray Fortran. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, 81–92. DOI: http://dx.doi.org/10.1145/2555243.2555270

Received March 2013; accepted December 2014