Scientific Programming 22 (2014) 75–91
DOI 10.3233/SPR-140383
IOS Press

Enabling highly-scalable remote memory access programming with MPI-3 One Sided [1]

Robert Gerstenberger [*], Maciej Besta [**] and Torsten Hoefler [***]
Department of Computer Science, ETH Zurich, Zurich, Switzerland
E-mails: {robertge, bestam, htor}@inf.ethz.ch

Abstract. Modern interconnects offer remote direct memory access (RDMA) features. Yet, most applications rely on explicit message passing for communications albeit their unwanted overheads. The MPI-3.0 standard defines a programming interface for exploiting RDMA networks directly, however, it's scalability and practicability has to be demonstrated in practice. In this work, we develop scalable bufferless protocols that implement the MPI-3.0 specification. Our protocols support scaling to millions of cores with negligible memory consumption while providing highest performance and minimal overheads. To arm programmers, we provide a spectrum of performance models for all critical functions and demonstrate the usability of our library and models with several application studies with up to half a million processes. We show that our design is comparable to, or better than UPC and Fortran Coarrays in terms of latency, bandwidth and message rate. We also demonstrate application performance improvements with comparable programming complexity.

Keywords: Programming systems, programming models, Remote Memory Access (RMA), Remote Direct Memory Access (RDMA), MPI-3, one sided communication

1. Motivation

Network interfaces evolve rapidly to implement a growing set of features directly in hardware. A key feature of today's high-performance networks is remote direct memory access (RDMA). RDMA enables a process to directly access memory on remote processes without involvement of the operating system or activities at the remote side. This hardware support enables a powerful programming mode similar to shared memory programming. RDMA is supported by on-chip networks in, e.g., Intel's SCC and IBM's Cell systems, as well as off-chip networks such as InfiniBand [30,37], IBM's PERCS [2] or BlueGene/Q [21], Cray's Gemini [1] and Aries [9], or even RoCE over Ethernet [5].

From a programmer's perspective, parallel programming schemes 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 side-effect of communication) and (3) remote memory access and partitioned global address space (PGAS) where synchronization and communication are managed independently.

Architects realized early that shared memory can often not be efficiently emulated on distributed machines [19]. Thus, message passing became the *de facto* standard for large-scale parallel programs [28]. However, with the advent of RDMA networks, it became clear that message passing over RDMA incurs additional overheads in comparison with native remote memory access (RMA, aka. PGAS) programming [6, 7,29]. This is mainly due to message matching, practical issues with overlap, and because fast message passing libraries over RDMA usually require different protocols [41]: an eager protocol with receiver-side buffering of small messages and a rendezvous protocol that synchronizes the sender. Eager requires additional copies, and rendezvous sends additional messages and may delay the sending process.

In summary, directly programming RDMA hardware has benefits in the following three dimensions:

[1]This paper received the Best Paper Award at the SC2013 conference and is published here with permission of ACM.

[*]The author performed much of the implementation during an internship at UIUC/NCSA while the analysis and documentation was performed during a scientific visit at ETH Zurich. The author's primary e-mail: gerro@hrz.tu-chemnitz.de.

[**]M. Besta is the recipient of the 2013 Google European Doctoral Fellowship in Parallel Computing.

[***]Corresponding author: Torsten Hoefler, Department of Computer Science, ETH Zurich, Universitätstr. 6, 8092 Zurich, Switzerland. E-mail: htor@inf.ethz.ch.

(1) *time* by avoiding message matching and synchronization overheads, (2) *energy* by reducing data-movement, e.g., it avoids additional copies of eager messages, and (3) *space* by removing the need for receiver buffering. Thus, several programming environments allow to access RDMA hardware more or less directly: PGAS languages such as Unified Parallel C (UPC [38]) or Fortran Coarrays [17] and libraries such as Cray SHMEM [3] or MPI-2.2 One Sided [27]. A lot of experience with these models has been gained in the past years [7,29,42] and several key design principles for remote memory access (RMA) programming evolved. The MPI Forum set out to define a portable library interface to RMA programming using these established principles. This new interface in MPI-3.0 [28] extends MPI-2.2's One Sided chapter to support the newest generation of RDMA hardware.

However, the MPI standard only defines a programming interface and does not mandate an implementation. Thus, it has yet to be demonstrated that the new library interface delivers comparable performance to compiled languages like UPC and Fortran Coarrays and is able to scale to large process numbers with small memory overheads. *In this work, we develop scalable protocols for implementing MPI-3.0 RMA over RDMA networks. We demonstrate that* (1) *the performance of a library implementation can be competitive to tuned, vendor-specific compiled languages and* (2) *the interface can be implemented on highly-scalable machines with negligible memory overheads. In a wider sense, our work answers the question if the MPI-3.0 RMA interface is a viable candidate for moving into the post-petascale era.*

Our key contributions are:

- We describe scalable protocols and a complete implementation for the novel MPI-3.0 RMA programming interface requiring $\mathcal{O}(\log p)$ time and space per process on p processes.
- We provide a detailed performance evaluation and performance models that can be used for algorithm development and to demonstrate the scalability to future systems.
- We demonstrate the benefits of RMA programming for several motifs and real-world applications on a multi-petaflop machine with full-application speedup of more than 13% over MPI-1 using more than half a million MPI processes.

2. Scalable protocols for MPI-3.0 One Sided over RDMA networks

We describe protocols to implement MPI-3.0 One Sided purely based on low-level remote direct memory access (RDMA). In all our protocols, we assume that we only have small bounded buffer space at each process, no remote software agent, and only put, get, and some basic atomic operations for remote memory access. This makes our protocols applicable to all current RDMA networks and is also forward-looking towards exascale interconnect architectures.

MPI-3.0 offers a plethora of functions with different performance expectations and use-cases. We divide the RMA functionality into three separate concepts: (1) window creation, (2) communication functions, and (3) synchronization functions. In addition, MPI-3's One Sided (RMA) specification differentiates between two memory models, *unified* and *separate*. The separate memory model is offering MPI-2 semantics and is thus portable to a wide variety of network architectures. The unified model provides additional semantic guarantees allowing direct access to hardware features. For example, it permits to wait (*polling*) for remote updates on a memory location and it allows fine-grained access to window memory during epochs. The user can query an MPI window attribute to determine the memory model supported in a specific window. The implementation described in this paper offers the stronger unified semantics for all windows, since it is supported by all current RDMA networks.

Figure 1(a) shows an overview of MPI's synchronization functions. They can be split into active target mode, in which the target process participates in the synchronization, and passive target mode, in which the target process is passive. Figure 1(b) shows a similar overview of MPI's communication functions. Several functions can be completed in bulk with bulk synchronization operations or using fine-grained request objects and test/wait functions. However, we observed that the completion model only minimally affects local overheads and is thus not considered separately in the remainder of this work.

Figure 1 also shows abstract definitions of the performance models for each synchronization and communication operation. The precise performance model for each function depends on the exact implementation. We provide a detailed overview of the *asymptotic* as well as *exact* performance properties of our protocols and our implementation in the next sections. The different performance characteristics of commu-

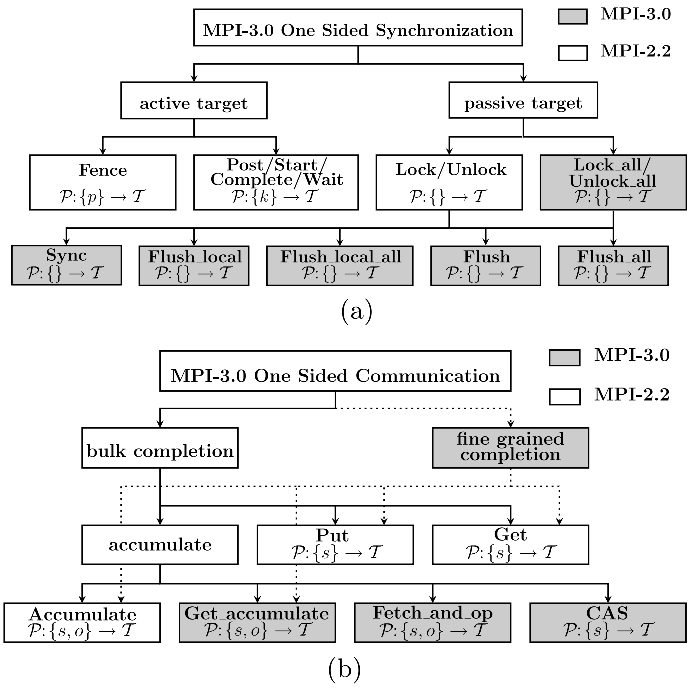

MPI-3.0 One Sided Synchronization
MPI-3.0
MPI-2.2
active target
passive target
Fence $\mathcal{P}:\{p\} \to \mathcal{T}$
Post/Start/ Complete/Wait $\mathcal{P}:\{k\} \to \mathcal{T}$
Lock/Unlock $\mathcal{P}:\{\} \to \mathcal{T}$
Lock_all/ Unlock_all $\mathcal{P}:\{\} \to \mathcal{T}$
Sync $\mathcal{P}:\{\} \to \mathcal{T}$
Flush_local $\mathcal{P}:\{\} \to \mathcal{T}$
Flush_local_all $\mathcal{P}:\{\} \to \mathcal{T}$
Flush $\mathcal{P}:\{\} \to \mathcal{T}$
Flush_all $\mathcal{P}:\{\} \to \mathcal{T}$
(a)
MPI-3.0 One Sided Communication
MPI-3.0
MPI-2.2
bulk completion
fine grained completion
accumulate
Put $\mathcal{P}:\{s\} \to \mathcal{T}$
Get $\mathcal{P}:\{s\} \to \mathcal{T}$
Accumulate $\mathcal{P}:\{s,o\} \to \mathcal{T}$
Get_accumulate $\mathcal{P}:\{s,o\} \to \mathcal{T}$
Fetch_and_op $\mathcal{P}:\{s,o\} \to \mathcal{T}$
CAS $\mathcal{P}:\{s\} \to \mathcal{T}$
(b)

Fig. 1. Overview of MPI-3.0 One Sided and associated cost functions. The figure shows abstract cost functions for all operations in terms of their input domains. The symbol p denotes the number of processes, s is the data size, k is the maximum number of neighbors, and o defines an MPI operation. The notation $\mathcal{P} : \{p\} \to \mathcal{T}$ defines the input space for the performance (cost) function $\mathcal{P}$. In this case, it indicates, for a specific MPI function, that the execution time depends only on p. We provide asymptotic cost functions in Section 2 and parametrized cost functions for our implementation in Section 3. (a) Synchronization. (b) Communication.

nication and synchronization functions make a unique combination of implementation options for each specific use-case optimal. However, it is not always easy to choose this best variant. The exact models can be used to design such optimal implementations (or as input for model-guided autotuning [10]) while the simpler asymptotic models can be used in the algorithm design phase (cf. [20]).

To support post-petascale computers, all protocols need to implement each function in a scalable way, i.e., consuming $\mathcal{O}(\log p)$ memory and time on p processes. For the purpose of explanation and illustration, we choose to discuss a reference implementation as use-case. However, all protocols and schemes discussed in the following can be used on any RDMA-capable network.

2.1. *Use-case*: *Cray DMAPP and XPMEM*

We introduce our implementation FOMPI (fast one-sided MPI), a fully-functional MPI-3.0 RMA library implementation for Cray Gemini (XK5, XE6) and Aries (XC30) systems. In order to maximize asynchronous progression and minimize overhead, FOMPI interfaces to the lowest available hardware APIs.

For inter-node (network) communication, FOMPI[2] uses the lowest-level networking API of Gemini and Aries networks, DMAPP (Distributed Memory Application), which has direct access to the hardware (GHAL) layer. DMAPP provides an RDMA interface and each process can expose (register) local memory regions. Accessing remote memory requires a special key (which is returned by the registration call). DMAPP offers put, get, and a limited set of atomic memory operations, each of them comes in three categories: blocking, explicit nonblocking, and implicit nonblocking. All explicit nonblocking routines return a handle that can be used to complete single operations, implicit nonblocking operations can only be finished by bulk completion (gsync) functions. DMAPP put and get can operate on 1, 4, 8 and 16 Byte chunks while atomic memory operations (AMO) always operate on 8 Bytes.

[2]The FOMPI source code can be found on http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI.

For intra-node communication, we use XPMEM [40], a portable Linux kernel module that allows to map the memory of one process into the virtual address space of another. Similar to DMAPP, processes can expose contiguous memory regions and other processes can attach (map) exposed regions into their own address space. All operations can then be directly implemented with load and store instructions, as well as CPU atomics (e.g., using the x86 lock prefix). Since XPMEM allows direct access to other processes' memory, we include it in the category of RDMA interfaces.

FOMPI's performance properties are self-consistent [22] and thus avoid surprises for users. We now proceed to develop algorithms to implement the window creation routines that expose local memory for remote access. After this, we describe protocols for synchronization and communication functions over RDMA networks.

2.2. Scalable window creation

A *window* is a region of process memory that is made accessible to remote processes. MPI-3.0 provides four collective functions for creating different types of windows: MPI_Win_create (traditional windows), MPI_Win_allocate (allocated windows), MPI_Win_create_dynamic (dynamic windows) and MPI_Win_allocate_shared (shared windows). We assume that communication memory needs to be registered with the communication subsystem and that remote processes require a remote descriptor that is returned from the registration to access the memory. This is true for most of today's RDMA interfaces including DMAPP and XPMEM.

Traditional windows. These windows expose existing user-memory to remote processes. Each process can specify an arbitrary local base address for the window and all remote accesses are relative to this address. This essentially forces an implementation to store all remote addresses separately. This storage may be compressed if addresses are identical, however, it requires $\Omega(p)$ storage on each of the p processes in the worst case.

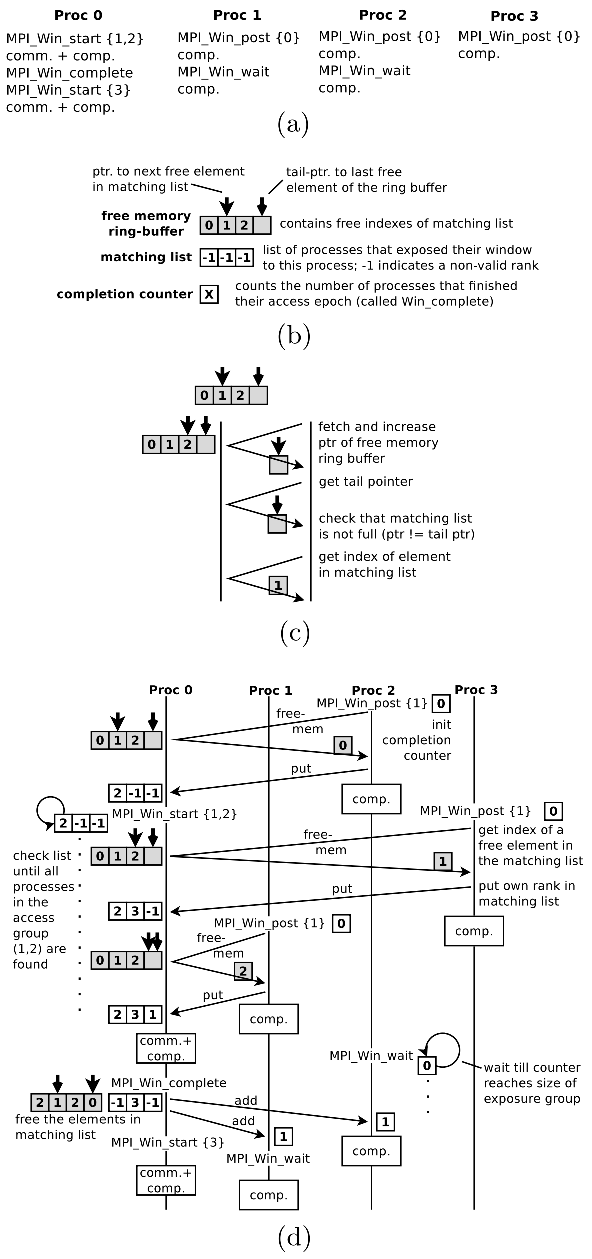
Each process discovers intra-node and inter-node neighbors and registers the local memory using XPMEM and DMAPP. Memory descriptors and various information (window size, displacement units, base pointer, et cetera) can be communicated with two MPI_Allgather operations: the first with all processes of the window to exchange DMAPP information and the second with the intra-node processes to exchange XPMEM information. Since traditional windows are fundamentally non-scalable, and only included in MPI-3.0 for backwards-compatibility, their use is strongly discouraged.

Allocated windows allow the MPI library to allocate window memory and thus use a symmetric heap, where the base addresses on all nodes are the same requiring only $\mathcal{O}(1)$ storage. This can either be done by allocating windows in a system-wide symmetric heap or with the following POSIX-compliant protocol: (1) a leader (typically process zero) chooses a random address which it broadcasts to all processes in the window, and (2) each process tries to allocate the memory with this specific address using `mmap()`. Those two steps are repeated until the allocation was successful on all processes (this can be checked with MPI_Allreduce). Size and displacement unit can now be stored locally at each process. This mechanism requires $\mathcal{O}(1)$ memory and $\mathcal{O}(\log p)$ time (with high probability).

Dynamic windows. These windows allow the dynamic attach and detach of memory regions using MPI_Win_attach and MPI_Win_detach. Attach and detach operations are non-collective. In our implementation, attach registers the memory region and inserts the information into a linked list and detach removes the region from the list. Both operations require $\mathcal{O}(1)$ memory per region.

The access of the list of memory regions on a target is purely one-sided using a local cache of remote descriptors. Each process maintains an id counter, which will be increased in case of an attach or detach operation. A process that attempts to communicate with the target first reads the id (with a get operation) to check if its cached information is still valid. If so, it finds the remote descriptor in its local list. If not, the cached information are discarded, the remote list is fetched with a series of remote operations and stored in the local cache.

Optimizations can be done similar to other distributed cache protocols. For example, instead of the id counter, each process could maintain a list of processes that have a cached copy of its local memory descriptors. Before returning from detach, a process notifies all these processes to invalidate their cache and discards the remote process list. For each communication attempt, a process has to first check if the local cache has been invalidated (in which case it will be reloaded). Then the local cache is queried for the remote descriptor. If the descriptor is missing, there has been an attach at the target and the remote descriptor is fetched into the local cache. After a cache invalidation or a first time access, a process has to register itself on the target for detach notifications. We explain a scalable data structure that can be used for the remote process list in the General Active Target Synchronization part (see Fig. 2(c)).



```
Proc 0                Proc 1            Proc 2            Proc 3
MPI_Win_start {1,2}   MPI_Win_post {0}  MPI_Win_post {0}  MPI_Win_post {0}
comm. + comp.         comp.             comp.             comp.
MPI_Win_complete      MPI_Win_wait      MPI_Win_wait
MPI_Win_start {3}     comp.             comp.
comm. + comp.
```

(a)
ptr. to next free element in matching list
tail-ptr. to last free element of the ring buffer
free memory ring-buffer
0 1 2
contains free indexes of matching list
matching list
-1 -1 -1
list of processes that exposed their window to this process; -1 indicates a non-valid rank
completion counter
X
counts the number of processes that finished their access epoch (called Win_complete)
(b)
fetch and increase ptr of free memory ring buffer
get tail pointer
check that matching list is not full (ptr != tail ptr)
get index of element in matching list
(c)
Proc 0
Proc 1
Proc 2
Proc 3
MPI_Win_post {1}
init completion counter
free-mem
put
comp.
MPI_Win_start {1,2}
MPI_Win_post {1}
get index of a free element in the matching list
put own rank in matching list
check list until all processes in the access group (1,2) are found
MPI_Win_post {1}
comm.+ comp.
MPI_Win_wait
wait till counter reaches size of exposure group
MPI_Win_complete
add
free the elements in matching list
MPI_Win_start {3}
MPI_Win_wait
(d)

Fig. 2. Example of General Active Target Synchronization. The numbers in the brackets for MPI_Win_start and MPI_Win_post indicate the processes in the access or exposure group. (a) Source code. (b) Data structures. (c) Free-storage management: protocol to acquire a free element in a remote matching list, denoted as "free-mem" below. (d) Possible execution of the complete protocol.

The optimized variant enables better latency for communication functions, but has a small memory overhead and is suboptimal for frequent detach operations.

Shared memory windows. Shared memory windows can be implemented using POSIX shared memory or XPMEM as described in [13] with constant memory overhead per core. Performance is identical to our direct-mapped (XPMEM) implementation and all operations are compatible with shared memory windows.

A note on DMAPP overheads. DMAPP requires the storage of one handle (with a size of 48 Bytes) per remote process. Thus, currently, all window creations have an additional memory overhead of $\Theta(p)$, and the communication is bound by the MPI_Allgather to gather those descriptors. However, this linear overhead is a current limitation of DMAPP and not our implementation or design. Assuming DMAPP had a collective context and similar displacement units per process, one would trivially be able to implement this function with $\mathcal{O}(\log p)$ time and $\mathcal{O}(1)$ space complexity.

We now show novel protocols to implement synchronization modes in a scalable way on pure RDMA networks without remote buffering.

2.3. Scalable window synchronization

MPI differentiates between *exposure* and *access* epochs. A process starts an exposure epoch to allow other processes to access its memory. In order to access exposed memory at a remote target, the origin process has to be in an access epoch. Processes can be in access and exposure epochs simultaneously and exposure epochs are only defined for active target synchronization (in passive target, window memory is always exposed).

Fence. MPI_Win_fence, called collectively by all processes, finishes the previous exposure and access epoch and opens the next exposure and access epoch for the whole window. An implementation must guarantee that all remote memory operations are committed before it leaves the fence call. Our implementation uses an x86 `mfence` instruction (XPMEM) and DMAPP bulk synchronization (gsync) followed by an MPI barrier to ensure global completion. The asymptotic memory bound is $\mathcal{O}(1)$ and, assuming a good barrier implementation, the time bound is $\mathcal{O}(\log p)$.

General active target synchronization synchronizes a subset of processes of a window. Exposure

(MPI_Win_post/MPI_Win_wait) and access epochs (MPI_Win_start/MPI_Win_complete) can be opened and closed independently. However, a group argument is associated with each call that starts an epoch and it states all processes participating in the epoch. The calls have to guarantee correct *matching*, i.e., if a process i specifies a process j in the group argument of the post call, then the next start call at process j that has process i in the group argument *matches* the post call.

Since our RMA implementation cannot assume buffer space for remote messages, it has to ensure that all processes in the group argument of the start call have called a matching post before the start returns. Similarly, the wait call has to ensure that all matching processes have called complete. Thus, calls to MPI_Win_start and MPI_Win_wait may block, waiting for the remote process. Both synchronizations are required to ensure integrity of the accessed data during the epochs. The MPI specification forbids matching configurations where processes wait cyclically (deadlocks).

We now describe a scalable implementation of the matching protocol with a time and memory complexity of $\mathcal{O}(k)$ if each process has at most k neighbors across all epochs. In addition, we assume k is known to the implementation. The scalable algorithm can be described at a high level as follows: each process i that *posts* an epoch announces itself to all processes $j_1, \ldots, j_l$ in the group argument by adding i to a list local to the processes $j_1, \ldots, j_l$. Each process j that tries to *start* an access epoch waits until all processes $i_1, \ldots, i_m$ in the group argument are present in its local list. The main complexity lies in the scalable storage of this neighbor list, needed for *start*, which requires a remote free-storage management scheme (see Fig. 2(c)). The *wait* call can simply be synchronized with a completion counter. A process calling *wait* will not return until the completion counter reaches the number of processes in the specified group. To enable this, the *complete* call first guarantees remote visibility of all issued RMA operations (by calling `mfence` or DMAPP's gsync) and then increases the completion counter at all processes of the specified group.

Figure 2(a) shows an example program with two distinct matches to access three processes from process 0. The first epoch on process 0 matches with processes 1 and 2 and the second epoch matches only with process 3. Figure 2(b) shows the necessary data structures, the free memory buffer, the matching list, and the completion counter. Figure 2(c) shows the part of the protocol to acquire a free element in a remote matching list and Fig. 2(d) shows a possible execution of the complete protocol on four processes.

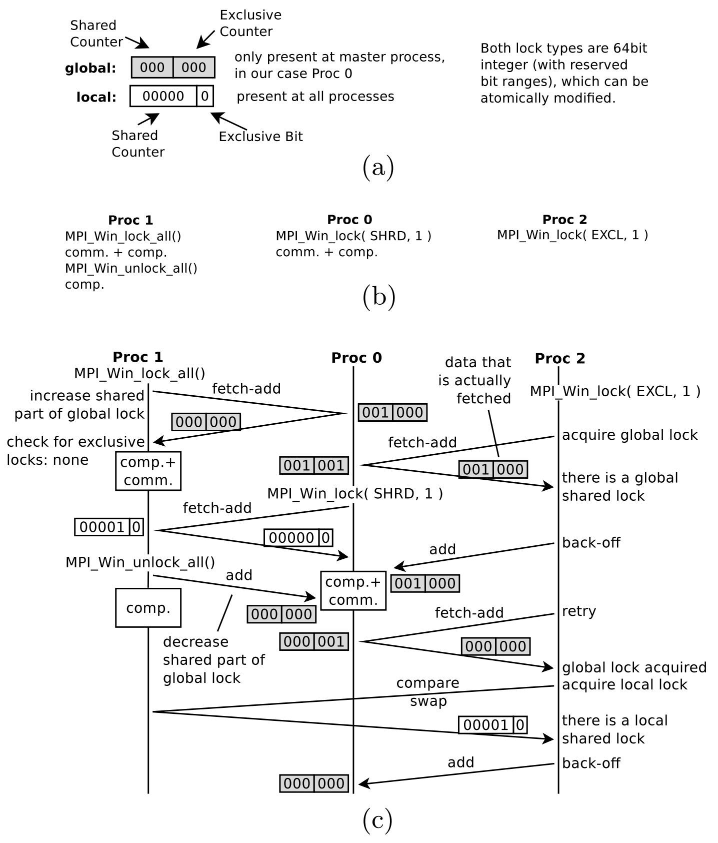
If k is the size of the group, then the number of messages issued by post and complete is $\mathcal{O}(k)$ and zero for start and wait. We assume that $k \in \mathcal{O}(\log p)$ in scalable programs [12].

Lock synchronization. We now describe a low-overhead and scalable strategy to implement shared global, and shared and exclusive process–local locks on RMA systems (the MPI-3.0 specification does not allow exclusive global lock all). We utilize a two-level lock hierarchy: one global lock variable (at a designated process, called *master*) and p local lock variables (one lock on each process). We assume that the word-size of the machine, and thus each lock variable, is 64 bits. Our scheme also generalizes to other t bit word sizes as long as the number of processes is not more than $2^{\lfloor t/2 \rfloor}$.

Each local lock variable is used to implement a reader–writer lock, which allows only one writer (*exclusive* lock), but many readers (*shared* locks). The highest order bit of the lock variable indicates a write access, while the other bits are used to count the number of shared locks held by other processes (cf. [24]). The global lock variable is split into two parts. The first part counts the number of processes holding a global shared lock in the window and the second part counts the number of exclusively locked processes. Both parts guarantee that there are only accesses of one type (either exclusive or lock all) concurrently. This data structure enables all lock operations to complete in $\mathcal{O}(1)$ steps if a lock can be acquired immediately. Figure 3(a) shows the structure of the local and global lock variables (counters).

Figure 3(b) shows an exemplary lock scenario for three processes. We do not provide an algorithmic description of the protocol due to the lack of space (the source-code is available online). However, we describe a locking scenario to foster understanding of the protocol. Figure 3(c) shows a possible execution schedule for the scenario from Fig. 3(b). Please note that we permuted the order of processes to $(1, 0, 2)$ instead of the intuitive $(0, 1, 2)$ to minimize overlapping lines in the figure.

Process 1 starts a lock all epoch by increasing the global shared counter atomically. Process 2 has to wait with its exclusive lock request until Process 1 finishes its lock all epoch. The waiting is performed by an atomic fetch and add. If the result of this operation finds any global shared lock then it backs off its request and does not enter the lock, and proceeds to retry.



```
Proc 1
MPI_Win_lock_all()
comm. + comp.
MPI_Win_unlock_all()
comp.

Proc 0
MPI_Win_lock( SHRD, 1 )
comm. + comp.

Proc 2
MPI_Win_lock( EXCL, 1 )
```

Fig. 3. Example of Lock Synchronization. (a) Data structures. (b) Source code. (c) Possible schedule.

If there is no global shared lock then it enters its local lock phase. An acquisition of a shared lock on a specific target (MPI_Win_lock) only involves the local lock on the target. The origin process (e.g., Process 0) fetches and increases the lock in one atomic operation. If the writer bit is not set, the origin can proceed. If an exclusive lock is present, the origin repeatedly (remotely) reads the lock until the writer finishes his access. All waits/retries can be performed with exponential back off to avoid congestion.

Summarizing the protocol: For a local exclusive lock, the origin process needs to ensure two invariants: (1) no global shared lock can be held or acquired during the local exclusive lock *and* (2) no local shared or exclusive lock can be held or acquired during the local exclusive lock. For the first part, the locking process fetches the lock variable from the master process and also increases the writer part in one atomic operation to register its wish for an exclusive lock. If the fetched value indicates lock all accesses, then the origin backs off by decreasing the writer part of the global lock. In case there is no global reader, the origin proceeds to the second invariant and tries to acquire an exclusive local lock on its target using compare-and-swap with zero (cf. [24]). If this succeeds, the origin acquired the lock and can proceed. In the example, Process 2 succeeds at the second attempt to acquire the global lock but fails to acquire the local lock and needs to back off by releasing its exclusive global lock. Process 2 will repeat this two-step operation until it acquired the exclusive lock. If a process already holds any exclusive lock, then it can immediately proceed to invariant two.

When unlocking (MPI_Win_unlock) a shared lock, the origin only has to atomically decrease the local lock on the target. In case of an exclusive lock it requires two steps. The first step is the same as in the shared case, but if the origin does not hold any additional exclusive locks, it has to release its global lock by atomically decreasing the writer part of the global lock.

The acquisition or release of a global shared lock for all processes of the window (MPI_Win_lock_all/ MPI_Win_unlock_all) is similar to the shared case for a specific target, except it targets the global lock.

If no exclusive locks exist, then shared locks (both MPI_Win_lock and MPI_Win_lock_all) only take one remote atomic update operation. The number of remote requests while waiting can be bound by using MCS locks [25]. The first exclusive lock will take in the best case two atomic communication operations. This will be reduced to one atomic operation if the origin process already holds an exclusive lock. Unlock operations always cost one atomic operation, except for the last unlock in an exclusive case with one extra atomic operation for releasing the global lock. The memory overhead for all functions is $\mathcal{O}(1)$.

Flush. Flush guarantees remote completion and is thus one of the most performance-critical functions on MPI-3.0 RMA programming. FOMPI's flush implementation relies on the underlying interfaces and simply issues a DMAPP remote bulk completion and an x86 `mfence`. All flush operations (MPI_Win_flush, MPI_Win_flush_local, MPI_Win_flush_all and MPI_Win_flush_all_local) share the same implementation and add only 78 CPU instructions (x86) to the critical path.

2.4. Communication functions

Communication functions map nearly directly to low-level hardware functions. This is a major strength of RMA programming. In FOMPI, put and get simply use DMAPP put and get for remote accesses or local memcpy for XPMEM accesses. Accumulates either use DMAPP atomic operations (for many common integer operations on 8 Byte data) or fall back to a simple protocol that locks the remote window, gets the data, accumulates it locally, and writes it back. This fallback protocol is necessary to avoid involvement of the receiver for true passive mode. It can be improved if we allow buffering (enabling a space–time trade-off [43]) such that active-mode communications can employ active messages to perform the remote operations atomically.

Handling datatypes. Our implementation supports arbitrary MPI datatypes by using the MPITypes library [33]. FOMPI offers 2 different strategies to handle MPI datatypes for putting data. The *Maximal Block Strategy* communicates directly in each step the maximum possible number of elements that are contiguous in the local and the remote memory. This is repeated until both MPI datatypes are fully processed. This approach avoids copying and is bufferless, but uses the maximum number of communication operations. The *Fixed Buffer Size Strategy* reduces the number of communication operations by overlapping copying with communication. While elements are contiguous in the remote memory, elements are copied into a temporary buffer until a predefined size is reached, then the temporary buffer is communicated. The Maximal Block Strategy is used to handle MPI datatypes for inter-node get and accumulate operations and always for intra-node communication to avoid data serialization. A detailed analysis of the different strategies was done by Schneider et al. [35], while a more detailed explanation of the FOMPI implementation can be found in [11].

While offering the full functionality of the rich MPI interface, our implementation is highly tuned for the common case of contiguous data transfers using intrinsic datatypes (e.g., MPI_DOUBLE). Our full implementation adds only 173 CPU instructions (x86) in the optimized critical path of MPI_Put and MPI_Get. We also utilize SSE-optimized assembly code to perform fast memory copies for XPMEM communication.

2.5. Blocking calls

The MPI standard allows an implementation to block in several synchronization calls. Each correct MPI program should thus never deadlock if all those calls are blocking. However, if the user knows the detailed behavior, she can tune for performance, e.g., if locks block, then the user may want to keep lock/unlock regions short. We describe here which calls may block depending on other processes and which calls will wait for other processes to reach a certain state. We point out that, in order to write (performance) portable programs, the user cannot rely on such knowledge in general!

With our protocols, (a) MPI_Win_fence waits for all other window processes to enter the MPI_Win_fence call, (b) MPI_Win_start waits for matching calls of MPI_Win_post from all processes in the access group, (c) MPI_Win_wait waits for MPI_Win_complete calls from all processes in the exposure group, and (d) MPI_Win_lock and MPI_Win_lock_all wait until they acquired the desired lock.

3. Detailed performance modeling and evaluation

We now describe several performance features of our protocols and implementation and compare it to Cray MPI's highly tuned point-to-point as well as its relatively untuned one sided communication. In addition, we compare FOMPI with two major HPC PGAS languages: UPC and Fortran 2008 with Coarrays, both specially tuned for Cray systems. We did not evaluate the semantically richer Coarray Fortran 2.0 [23] because no tuned version was available on our system. We execute all benchmarks on the Blue Waters system, using the Cray XE6 nodes only. Each compute node contains four 8-core AMD Opteron 6276 (Interlagos) 2.3 GHz and is connected to other nodes through a 3D-Torus Gemini network. We use the Cray Programming Environment 4.1.40 for MPI, UPC, and Fortran Coarrays, and GNU gcc 4.7.2 when features that are not supported by Cray's C compiler are required (e.g., inline assembly for a fast x86 SSE copy loop).

Our benchmark measures the time to perform a single operation (for a given process count and/or data size) at all processes and adds the maximum across all ranks to a bucket. The run is repeated 1,000 times to gather statistics. We use the cycle accurate x86 RDTSC counter for each measurement. All performance figures show the medians of all gathered points for each configuration.

3.1. Latency and bandwidth

Comparing latency and bandwidth between one sided RMA communication and point-to-point communication is not always fair since RMA communication may require extra synchronization to notify the target. All latency results presented for RMA interfaces are guaranteeing remote completion (the message is committed in remote memory) but no synchronization. We analyze synchronization costs separately in Section 3.2.

We measure MPI-1 point-to-point latency with standard ping-pong techniques. For Fortran Coarrays, we use a remote assignment of a double precision array of size SZ:

```
double precision , dimension(SZ) ::
    buf[*]
do memsize in all sizes<SZ
  buf(1:memsize)[2] = buf(1:memsize)
  sync memory
end do
```

In UPC, we use a single shared array and the intrinsic function memput, we also tested shared pointers with very similar performance:

```
shared [SZ] double *buf;
buf = upc_all_alloc(2, SZ);
for(size=1 ; size<=SZ ; size*=2) {
  upc_memput(&buf[SZ], &priv_buf[0],
      size);
  upc_fence;
}
```

In MPI-3.0 RMA, we use an allocated window and passive target mode with flushes:

```
MPI_Win_allocate(SZ,... , &buf, &win);
MPI_Win_lock(excl, 1, ..., win);
for(size=1 ; size<=SZ ; size*=2) {
  MPI_Put(&buf[0], size, ..., 1, ...,
      win);
  MPI_Win_flush(1, win);
}
MPI_Win_unlock(1, win);
```

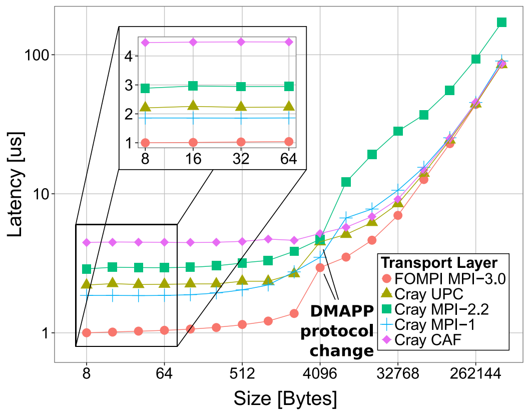
Figures 4(a), (b) and 5(a) show the latency for varying message sizes for intra- and inter-node put, get. Due to the highly optimized fast-path, FOMPI has more than 50% lower latency than other PGAS models while achieving the same bandwidth for larger messages. The performance functions (cf. Fig. 1) are: $\mathcal{P}_{\text{put}} = 0.16\ \text{ns} \cdot s + 1\ \mu\text{s}$ and $\mathcal{P}_{\text{get}} = 0.17\ \text{ns} \cdot s + 1.9\ \mu\text{s}$.

3.1.1. Overlapping computation

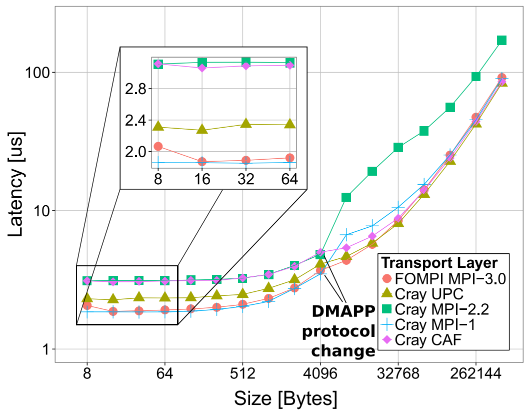
The overlap benchmark measures how much of the communication time can be overlapped with computation. It calibrates a computation loop to consume slightly more time than the latency. Then it places computation between the communication and the synchronization and measures the combined time. The ratio of overlapped computation is then computed from the measured communication, computation, and combined times. Figure 5(b) shows the ratio of the communication that can be overlapped for Cray's MPI-2.2, UPC and FOMPI MPI-3.0.

3.1.2. Message rate

The message rate benchmark is very similar to the latency benchmark, however, it benchmarks the start of 1,000 transactions without synchronization. This determines the overhead for starting a single operation. The Cray-specific PGAS pragma defer_sync was used in the UPC and Fortran Coarrays versions for full optimization. Figure 6(a) and (b) show the message rates for DMAPP and XPMEM (shared memory) commu-

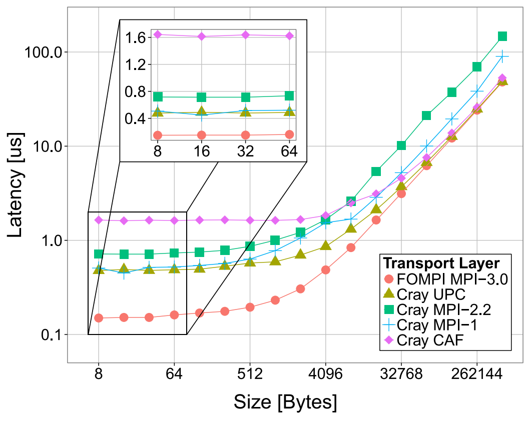


(a)

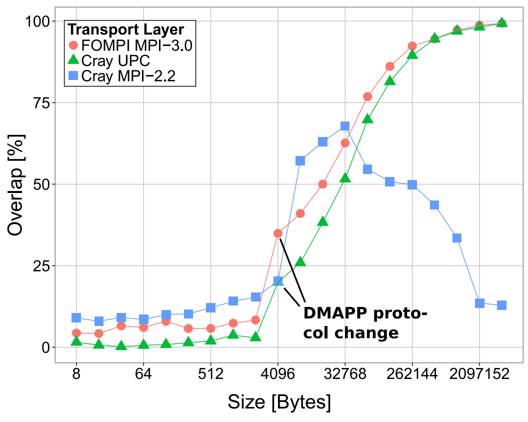


(b)

Fig. 4. Latency comparison for remote put/get for DMAPP communication. Note that MPI-1 Send/Recv implies remote synchronization while UPC, Fortran Coarrays and MPI-2.2/3.0 only guarantee consistency. (a) Latency inter-node put. (b) Latency inter-node get. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)



(a)



(b)

Fig. 5. (a) shows a latency comparison for remote put/get for XPMEM (shared memory) communication. Note that MPI-1 Send/Recv implies remote synchronization while UPC, Fortran Coarrays and MPI-2.2/3.0 only guarantee consistency. (b) demonstrates the communication/computation overlap for put over DMAPP, Cray MPI-2.2 has much higher latency up to 64 kB (cf. Fig. 4(a)), thus allows higher overlap. XPMEM implementations do not support overlap due to the shared memory copies. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)

nications, respectively. Injecting a single 8-Byte message costs 416 ns for inter-node and 80 ns (∼190 instructions) for intra-node case.

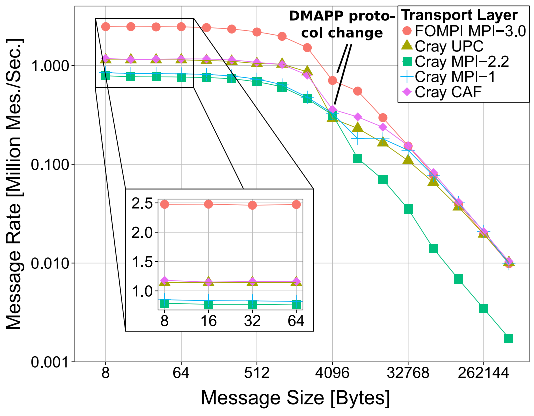
3.1.3. *Atomics*

Figure 7 shows the performance of the DMAPP-accelerated MPI_SUM of 8-Byte elements, a non-accelerated MPI_MIN and 8-Byte CAS. The performance functions are $\mathcal{P}_{\text{acc,sum}} = 28\text{ ns} \cdot s + 2.4\ \mu\text{s}$, $\mathcal{P}_{\text{acc,min}} = 0.8\text{ ns} \cdot s + 7.3\ \mu\text{s}$ and $\mathcal{P}_{\text{CAS}} = 2.4\ \mu\text{s}$. The DMAPP acceleration lowers the latency for small messages while the locked implementation exhibits a higher bandwidth. However, this does not consider the serialization due to the locking.

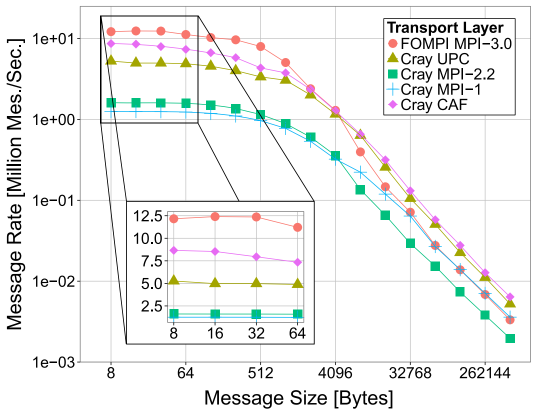
3.2. *Synchronization schemes*

In this section, we study the overheads and model the performance of the various synchronization modes. Our performance models can be used by the programmer to select the best option for the problem at hand.

The different modes have nontrivial trade-offs. For example General Active Target Synchronization performs better if small groups of processes are synchro-



(a)



(b)

Fig. 6. Message rates for put communication for all transports. (a) Message rate inter-node. (b) Message Rate intra-node. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)

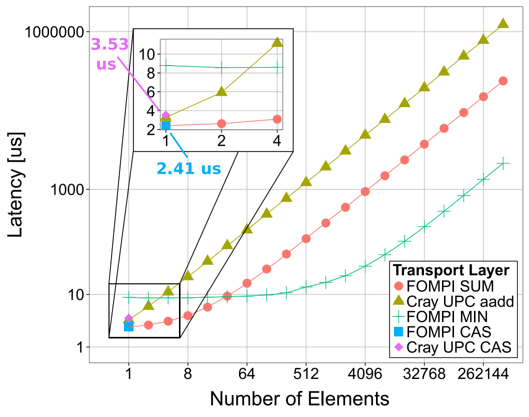


Fig. 7. Atomic operation performance. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)

nized and fence synchronization performs best if the synchronization groups are essentially as big as the full group attached to the window. However, the exact crossover point is a function of the implementation and system. While the active target mode notifies the target implicitly that its memory is consistent, in passive target mode, the user has to do this explicitly or rely on synchronization side effects of other functions (e.g., allreduce).

Global synchronization. Global synchronization is offered by fences in MPI-2.2 and MPI-3.0. It can be directly compared to Fortran Coarrays `sync all` and UPC's `upc_barrier` which also synchronize the memory at all processes. Figure 8(a) compares the performance of FOMPI with Cray's MPI-2.2, UPC, and Fortran Coarrays implementations. The performance function for FOMPI's fence implementation is: $\mathcal{P}_{\text{fence}} = 2.9\ \mu\text{s} \cdot \log_2(p)$.

General active target synchronization. Only MPI-2.2 and MPI-3.0 offer General Active Target (also called "PSCW") synchronization. A similar mechanism (`sync images`) for Fortran Coarrays was unfortunately not available on our test system. Figure 8(b) shows the performance for Cray MPI and FOMPI when synchronizing an one-dimensional Torus (ring) where each process has exactly two neighbors ($k = 2$). An ideal implementation would exhibit constant time for this benchmark. We observe systematically growing overheads in Cray's implementation as well as system noise [15,31] on runs with more than 1,000 processes with FOMPI. We model the performance with varying numbers of neighbors and FOMPI's PSCW synchronization costs involving k off-node neighbor are $\mathcal{P}_{\text{post}} = \mathcal{P}_{\text{complete}} = 350\ \text{ns} \cdot k$, and $\mathcal{P}_{\text{start}} = 0.7\ \mu\text{s}$, $\mathcal{P}_{\text{wait}} = 1.8\ \mu\text{s}$.

Passive target synchronization. The performance of lock/unlock is constant in the number of processes (due to the global/local locking) and thus not graphed. The performance functions are $\mathcal{P}_{\text{lock,excl}} = 5.4\ \mu\text{s}$, $\mathcal{P}_{\text{lock,shrd}} = \mathcal{P}_{\text{lock\_all}} = 2.7\ \mu\text{s}$, $\mathcal{P}_{\text{unlock}} = \mathcal{P}_{\text{unlock\_all}} = 0.4\ \mu\text{s}$, $\mathcal{P}_{\text{flush}} = 76\ \text{ns}$ and $\mathcal{P}_{\text{sync}} = 17\ \text{ns}$.

We demonstrated the performance of our protocols and implementation using microbenchmarks comparing to other RMA and message passing implementations. The exact performance models for each call can be utilized to design and optimize parallel applications, however, this is outside the scope of this paper. To demonstrate the usability and performance of our protocols for real applications, we continue with a large-scale application study.

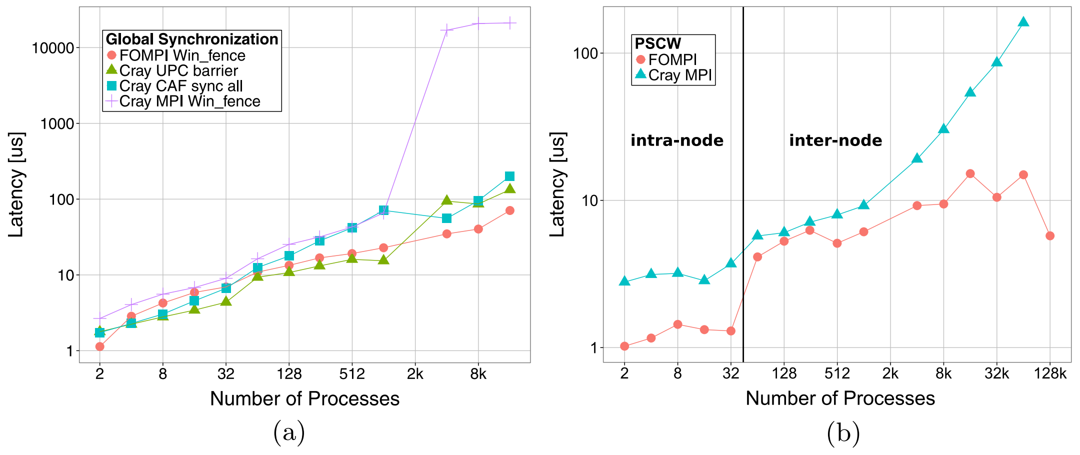


Fig. 8. Synchronization latencies. (a) Latency for global synchronization. (b) Latency for PSCW (ring topology). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)

4. Application evaluation

We selected two motif applications to compare our protocols and implementation with the state of the art: a distributed hashtable representing many big data and analytics applications and a dynamic sparse data exchange representing complex modern scientific applications. We also analyze the application MIMD Lattice Computation (MILC), a full production code with several hundred thousand source lines of code, as well as a 3D FFT code.

In all codes, we tried to keep most parameters constant to compare the performance of PGAS languages, MPI-1 and MPI-3.0 RMA. Thus, we did not employ advanced concepts, such as MPI datatypes or process topologies, that are not available in all implementations (e.g., UPC and Fortran Coarrays).

4.1. Distributed hashtable

Our simple hashtable represents data analytics applications that often require random access in distributed structures. We compare MPI point-to-point communication, UPC, and MPI-3.0 RMA. In the implementation, each process manages a part of the hashtable called the *local volume* consisting of a table of elements and an additional overflow heap to store elements after collisions. The table and the heap are constructed using fixed-size arrays. In order to avoid traversing of the arrays, pointers to most recently inserted items as well as to the next free cells are stored along with the remaining data in each local volume. The elements of the hashtable are 8-Byte integers.

The MPI-1 implementation is based on MPI Send and Recv using an *active message* scheme. Each process that is going to perform a remote operation sends the element to be inserted to the owner process which invokes a handle to process the message. Termination detection is performed using a simple protocol where each process notifies all other processes of its local termination. In UPC, table and overflow list are placed in shared arrays. Inserts are based on proprietary (Cray-specific extensions) atomic compare and swap (CAS) operations. If a collision happens, the losing thread acquires a new element in the overflow list by atomically incrementing the next free pointer. It also updates the last pointer using a second CAS. UPC_fences are used to ensure memory consistency. The MPI-3.0 implementation is rather similar to the UPC implementation, however, it uses MPI-3.0's standard atomic operations combined with flushes.

Figure 9(a) shows the inserts per second for a batch of 16k operations per process, each adding an element to a random key (which resides at a random process). MPI-1's performance is competitive for intranode communications but inter-node overheads significantly impact performance and the insert rate of a single node cannot be achieved with even 32k cores (optimizations such as coalescing or message routing and reductions [39] would improve this rate but significantly complicate the code). FOMPI and UPC exhibit similar performance characteristics with FOMPI being slightly faster for shared memory accesses. The spikes at 4k and 16k nodes are caused by different job layouts in the Gemini torus and different network congestion.

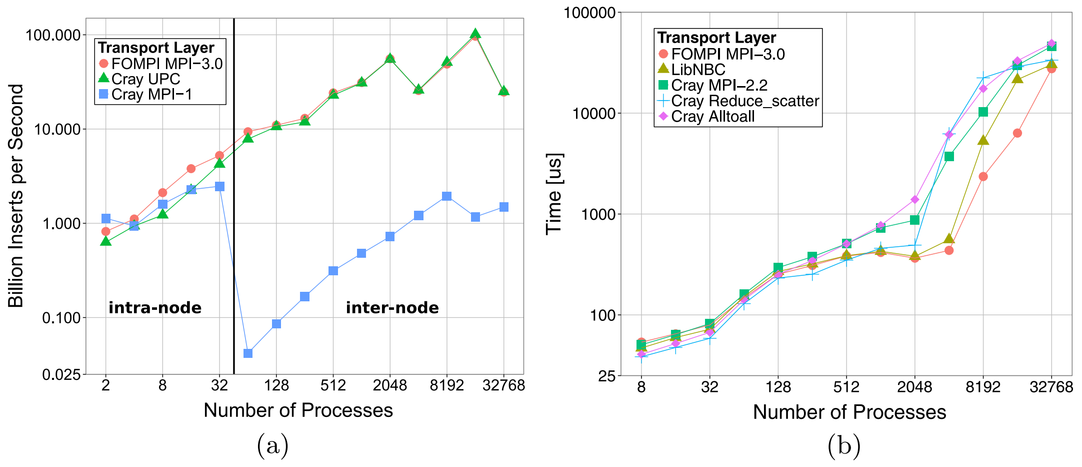


Fig. 9. Application motifs: (a) Hashtable representing data analytics applications and key-value stores, (b) dynamic sparse data exchange representing graph traversals, n-body methods, and rapidly evolving meshes [16]. (a) Inserts per second for inserting 16k elements per process including synchronization. (b) Time to perform one dynamic sparse data exchange (DSDE) with 6 random neighbors. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)

4.2. *Dynamic sparse data exchange*

The dynamic sparse data exchange (DSDE) represents a common pattern in irregular applications [16]. DSDE is used when a set of senders has data destined to arbitrary target processes but no process knows the volume or sources of data it needs to receive. The DSDE pattern is very common in graph-based computations, n-body simulations, and adaptive mesh refinement codes. Due to the lack of space, we use a DSDE microbenchmark as proxy for the communication performance of such applications [16].

In the DSDE benchmark, each process picks k targets randomly and attempts to send eight Bytes to each target. The DSDE protocol can either be implemented using alltoall, reduce_scatter, a nonblocking barrier combined with synchronous sends, or one sided accumulates in active target mode. The algorithmic details of the protocols are described in [16]. Here, we compare all protocols of this application microbenchmark with the Cray MPI-2.2 and FOMPI MPI-3.0 implementations. Figure 9(b) shows the times for the complete exchange using the four different protocols (the accumulate protocol is tested with Cray's MPI-2.2 implementation and FOMPI) and $k = 6$ random neighbors per process. The RMA-based implementation is competitive with the nonblocking barrier, which was proved optimal in [16]. FOMPI's accumulates have been tuned for Cray systems while the nonblocking barrier we use is a generic dissemination algorithm. The performance improvement relative to other protocols is always significant and varies between a factor of two and nearly two orders of magnitude.

4.3. *3D Fast Fourier Transform*

We now discuss how to exploit overlap of computation and communication with our low-overhead implementation in a three-dimensional Fast Fourier Transformation. We use the MPI and UPC versions of the NAS 3D FFT benchmark. Nishtala et al. and Bell et al. [7,29] demonstrated that overlap of computation and communication can be used to improve the performance of a 2D-decomposed 3D FFT. We compare the default "nonblocking MPI" with the "UPC slab" decomposition, which starts to communicate the data of a plane as soon as it is available and completes the communication as late as possible. For a fair comparison, our FOMPI implementation uses the same decomposition and communication scheme like the UPC version and required minimal code changes resulting in the same code complexity.

Figure 10(a) shows the performance for the strong scaling class D benchmark ($2048 \times 1024 \times 1024$) on different core counts. UPC achieves a consistent speedup over MPI-1, mostly due to the overlap of communication and computation. FOMPI has a slightly lower static overhead than UPC and thus enables better overlap (cf. Fig. 5(b)), resulting in a slightly better performance of the FFT.

4.4. *MIMD lattice computation*

The MIMD Lattice Computation (MILC) Collaboration studies Quantum Chromodynamics (QCD), the theory of strong interaction [8]. The group develops a

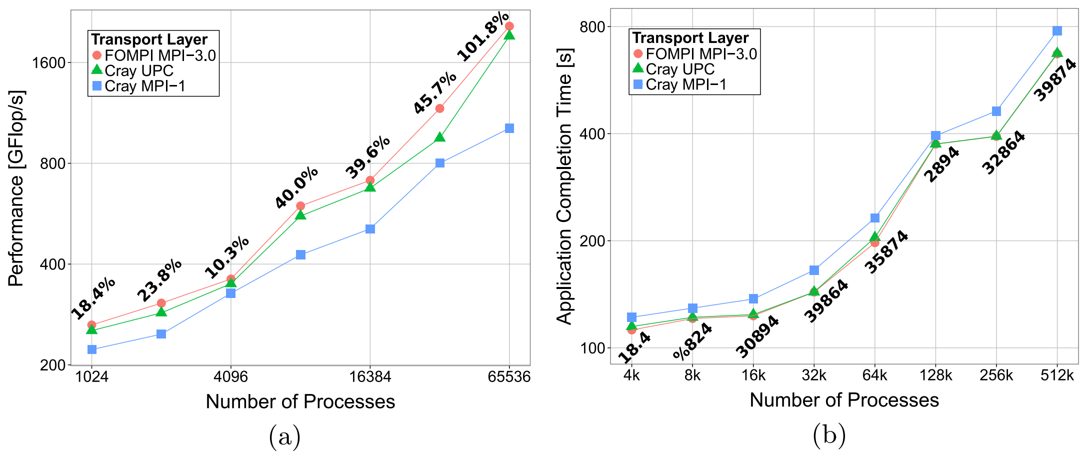


Fig. 10. Applications: The annotations represent the improvement of FOMPI over MPI-1. (a) 3D FFT performance. (b) MILC: Full application execution time. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140383.)

set of applications, known as the MILC code. In this work, we use version 7.6.3 as a base. That code regularly gets one of the largest allocations of computer time at US NSF supercomputer centers. The su3_rmd code, which is part of the SPEC CPU2006 and SPEC MPI benchmarks, is included in the MILC code.

The code performs a stencil computation on a four-dimensional rectangular grid. Domain decomposition is performed in all four dimensions to minimize the surface-to-volume ratio. In order to keep data consistent, neighbor communication is performed in all eight directions, in addition, global all reductions are done regularly to check convergence of a solver. The most time consuming part of MILC is the conjugate gradient solver which uses nonblocking MPI communication overlapped with local computations.

The performance of the full code and the solver have been analyzed in detail in [4]. Several optimizations have been applied, and a UPC version that demonstrated significant speedups is available [36]. This version replaces the MPI communication with a simple remote memory access protocol. A process notifies all neighbors with a separate atomic add as soon as the data in the "send" buffer is initialized. Then all processes wait for this flag before they get (using Cray's proprietary `upc_memget_nb`) the communication data into their local buffers. This implementation serializes the data from the application buffer into UPC's communication buffers. Our MPI-3.0 implementation follows the same scheme to ensure a fair comparison. We place the communication buffers into MPI windows and use MPI_Fetch_and_op and MPI_Get with a single lock all epoch and MPI_Win_flush to perform the communications. The necessary changes are small and the total number of source code lines is equivalent to the UPC version, We remark that additional optimizations may be possible with MPI, for example, one could use MPI datatypes to communicate the data directly from the application buffers resulting in additional performance gains [14]. However, since our goal is to compare to the UPC version, we only investigate the packed version here.

Figure 10(b) shows the execution time of the whole application for a weak-scaling problem with a local lattice of $4^3 \times 8$, a size very similar to the original Blue Waters Petascale benchmark. Some phases (e.g., CG) of the computation execute up to 45% faster, however, we chose to report full-application performance. The UPC and FOMPI codes exhibit essentially the same performance, while the UPC code uses Cray-specific tuning and the MPI-3.0 code is portable to different architectures. The full-application performance gain over the MPI-1 version is more than 15% for some configurations. The application was scaled successfully up to 524,288 processes with all implementations. This result and our microbenchmarks demonstrate the scalability and performance of our protocols and that the new RMA semantics can be used to improve full applications to achieve performance close to the hardware limitations in a fully portable way. Since most of those existing applications are written in MPI, a step-wise transformation can be used to optimize most critical parts first.

5. Related work

The intricacies of MPI-2.2 RMA implementations over InfiniBand networks have been discussed by Jiang et al. [18] and Santhanaraman et al. [34]. Zhao et al. describe an adaptive strategy to switch from eager to lazy modes in active target synchronizations in MPICH 2 [43]. This mode could be used to speed up FOMPI's atomics that are not supported in hardware.

PGAS programming has been investigated in the context of UPC and Fortran Coarrays. An optimized UPC Barnes Hut implementation shows similarities to MPI-3.0 RMA programming by using bulk vectorized memory transfers combined with vector reductions instead of shared pointer accesses [42]. Nishtala at al. and Bell et al. used overlapping and one-sided accesses to improve FFT performance [7,29]. Highly optimized PGAS applications often use a style that can easily be adapted to MPI-3.0 RMA.

The applicability of MPI-2.2 One Sided has also been demonstrated for some applications. Mirin et al. discuss the usage of MPI-2.2 One Sided coupled with threading to improve the Community Atmosphere Model (CAM) [26]. Potluri et al. show that MPI-2.2 One Sided with overlap can improve the communication in a Seismic Modeling application [32]. However, we demonstrated new MPI-3.0 features that can be used to further improve performance and simplify implementations.

6. Discussion and conclusions

In this work, we demonstrated how MPI-3.0 can be implemented over RDMA networks to achieve similar performance to UPC and Fortran Coarrays while offering all of MPI's convenient functionality (e.g., Topologies and Datatypes). We provide detailed performance models, that help choosing among the multiple options. For example, a user can use our models to decide whether to use Fence or PSCW synchronization (if $\mathcal{P}_{\text{fence}} > \mathcal{P}_{\text{post}} + \mathcal{P}_{\text{complete}} + \mathcal{P}_{\text{start}} + \mathcal{P}_{\text{wait}}$, which is true for large k). This is just one example for the possible uses of our detailed performance models.

We studied all overheads in detail and provide instruction counts for all critical synchronization and communication functions, showing that the MPI interface adds merely between 150 and 200 instructions in the fast path. This demonstrates that a library interface like MPI is competitive with compiled languages such as UPC and Fortran Coarrays. Our implementation proved to be scalable and robust while running on 524,288 processes on Blue Waters speeding up a full application run by 13.8% and a 3D FFT on 65,536 processes by a factor of two.

We expect that the principles and extremely scalable synchronization algorithms developed in this work will act as a blue print for optimized MPI-3.0 RMA implementations over future large-scale RDMA networks. We also expect that the demonstration of highest performance to users will quickly increase the number of MPI RMA programs.

Acknowledgements

We thank Timo Schneider for early help in the project, Greg Bauer and Bill Kramer for support with Blue Waters, Cray's Duncan Roweth and Roberto Ansaloni for help with Cray's PGAS environment, Nick Wright for the UPC version of MILC, and Paul Hargrove for the UPC version of NAS-FT. This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-FC02-10ER26011, program manager Lucy Nowell. This work is partially supported by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

The research was partly supported by the Google European Doctoral Fellowship in Parallel Computing program.

References

[1] R. Alverson, D. Roweth and L. Kaplan, The Gemini system interconnect, in: *Proceedings of the IEEE Symposium on High Performance Interconnects (HOTI'10)*, IEEE Computer Society, 2010, pp. 83–87.

[2] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni and R. Rajamony, The PERCS high-performance interconnect, in: *Proceedings of the IEEE Symposium on High Performance Interconnects (HOTI'10)*, IEEE Computer Society, 2010, pp. 75–82.

[3] R. Barriuso and A. Knies, *SHMEM User's Guide for C*, 1994.

[4] G. Bauer, S. Gottlieb and T. Hoefler, Performance modeling and comparative analysis of the MILC lattice QCD application su3_rmd, in: *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12)*, IEEE Computer Society, 2012, pp. 652–659.

[5] M. Beck and M. Kagan, Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure, in: *Proceedings of the Workshop on Data Center – Converged and Virtual Ethernet Switching (DC-CaVES'11)*, ITCP, 2011, pp. 9–15.

[6] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome and K. Yelick, An evaluation of current high-performance networks, in: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, IEEE Computer Society, 2003.

[7] C. Bell, D. Bonachea, R. Nishtala and K. Yelick, Optimizing bandwidth limited problems using one-sided communication and overlap, in: *Proceedings of the International Conference on Parallel and Distributed Processing (IPDPS'06)*, IEEE Computer Society, 2006, pp. 1–10.

[8] C. Bernard, M.C. Ogilvie, T.A. DeGrand, C.E. DeTar, S.A. Gottlieb, A. Krasnitz, R. Sugar and D. Toussaint, Studying quarks and gluons on MIMD parallel computers, *International Journal of High Performance Computing Applications* **5**(4) (1991), 61–70.

[9] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins and J. Reinhard, Cray Cascade: A scalable HPC system based on a Dragonfly network, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, IEEE Computer Society, 2012, pp. 103:1–103:9.

[10] B.B. Fraguela, Y. Voronenko and M. Pueschel, Automatic tuning of discrete Fourier transforms driven by analytical modeling, in: *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'09)*, IEEE Computer Society, 2009, pp. 271–280.

[11] R. Gerstenberger, Handling datatypes in MPI-3 one sided, ACM Student Research Competition Poster at the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13), 2013.

[12] A.Y. Grama, A. Gupta and V. Kumar, Isoefficiency: measuring the scalability of parallel algorithms and architectures, *Parallel and Distributed Technology: Systems and Technology* **1**(3) (1993), 12–21.

[13] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale and R. Thakur, Leveraging MPI's one-sided communication interface for shared-memory programming, in: *Recent Advances in the Message Passing Interface (EuroMPI'12)*, LNCS, Vol. 7490, Springer, 2012, pp. 132–141.

[14] T. Hoefler and S. Gottlieb, Parallel zero-copy algorithms for Fast Fourier Transform and conjugate gradient using MPI datatypes, in: *Recent Advances in the Message Passing Interface (EuroMPI'10)*, LNCS, Vol. 6305, Springer, 2010, pp. 132–141.

[15] T. Hoefler, T. Schneider and A. Lumsdaine, Characterizing the influence of system noise on large-scale applications by simulation, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, IEEE Computer Society, 2010, pp. 1–11.

[16] T. Hoefler, C. Siebert and A. Lumsdaine, Scalable communication protocols for dynamic sparse data exchange, in: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, ACM, 2010, pp. 159–168.

[17] ISO Fortran Committee, Fortran 2008 Standard (ISO/IEC 1539-1:2010), 2010.

[18] W. Jiang, J. Liu, H.-W. Jin, D.K. Panda, W. Gropp and R. Thakur, High performance MPI-2 one-sided communication over InfiniBand, in: *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'04)*, IEEE Computer Society, 2004, pp. 531–538.

[19] S. Karlsson and M. Brorsson, A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2, in: *Proceedings of the International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC'98)*, Springer, 1998, pp. 189–201.

[20] R.M. Karp, A. Sahay, E.E. Santos and K.E. Schauser, Optimal broadcast and summation in the LogP model, in: *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)*, ACM, 1993, pp. 142–153.

[21] S. Kumar, A. Mamidala, D.A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen and B.D. Steinmacher-Burrow, PAMI: A parallel active message interface for the Blue Gene/Q supercomputer, in: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, IEEE Computer Society, 2012, pp. 763–773.

[22] J. Larsson Traff, W.D. Gropp and R. Thakur, Self-consistent MPI performance guidelines, *IEEE Transactions on Parallel and Distributed Systems* **21**(5) (2010), 698–709.

[23] J. Mellor-Crummey, L. Adhianto, W.N. Scherer III and G. Jin, A new vision for Coarray Fortran, in: *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS'09)*, ACM, 2009, pp. 5:1–5:9.

[24] J.M. Mellor-Crummey and M.L. Scott, Scalable reader-writer synchronization for shared-memory multiprocessors, *SIGPLAN Notices* **26**(7) (1991), 106–113.

[25] J.M. Mellor-Crummey and M.L. Scott, Synchronization without contention, *SIGPLAN Notices* **26**(4) (1991), 269–278.

[26] A.A. Mirin and W.B. Sawyer, A scalable implementation of a finite-volume dynamical core in the community atmosphere model, *International Journal of High Performance Computing Applications* **19**(3) (2005), 203–212.

[27] MPI Forum, MPI: A message-passing interface standard. Version 2.2, 2009.

[28] MPI Forum, MPI: A message-passing interface standard. Version 3.0, 2012.

[29] R. Nishtala, P.H. Hargrove, D.O. Bonachea and K.A. Yelick, Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap, in: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, IEEE Computer Society, 2009, pp. 1–12.

[30] OpenFabrics Alliance (OFA), OpenFabrics Enterprise Distribution (OFED), available at: www.openfabrics.org.

[31] F. Petrini, D.J. Kerbyson and S. Pakin, The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'03)*, ACM, 2003.

[32] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K.W. Schulz, W.L. Barth, A. Majumdar and D.K. Panda, Quantifying performance benefits of overlap using MPI-2 in a seismic modeling application, in: *Proceedings of the ACM International Conference on Supercomputing (ICS'10)*, ACM, 2010, pp. 17–25.

[33] R. Ross, R. Latham, W. Gropp, E. Lusk and R. Thakur, Processing MPI datatypes outside MPI, in: *Recent Advances in*

Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI'09), LNCS, Vol. 5759, Springer, 2009, pp. 42–53.

[34] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp and D.K. Panda, Natively supporting true one-sided communication in MPI on multi-core systems with InfiniBand, in: *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, IEEE Computer Society, 2009, pp. 380–387.

[35] T. Schneider, R. Gerstenberger and T. Hoefler, Compiler optimizations for non-contiguous remote data movement, in: *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'13)*, 2013.

[36] H. Shan, B. Austin, N. Wright, E. Strohmaier, J. Shalf and K. Yelick, Accelerating applications at scale using one-sided communication, in: *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, 2012.

[37] The InfiniBand Trade Association, *Infiniband Architecture Specification*, Vol. 1, Release 1.2, InfiniBand Trade Association, 2004.

[38] UPC Consortium, UPC language specifications, v1.2, 2005, LBNL-59208.

[39] J. Willcock, T. Hoefler, N. Edmonds and A. Lumsdaine, Active Pebbles: Parallel programming for data-driven applications, in: *Proceedings of the ACM International Conference on Supercomputing (ICS'11)*, ACM, 2011, pp. 235–245.

[40] M. Woodacre, D. Robb, D. Roe and K. Feind, *The SGI Altix TM 3000 Global Shared-Memory Architecture*, 2003.

[41] T.S. Woodall, G.M. Shipman, G. Bosilca and A.B. Maccabe, High performance RDMA protocols in HPC, in: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI'06)*, LNCS, Vol. 4192, Springer, 2006, pp. 76–85.

[42] J. Zhang, B. Behzad and M. Snir, Optimizing the Barnes–Hut algorithm in UPC, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, ACM, 2011, pp. 75:1–75:11.

[43] X. Zhao, G. Santhanaraman and W. Gropp, Adaptive strategy for one-sided communication in MPICH2, in: *Recent Advances in the Message Passing Interface (EuroMPI'12)*, Springer, 2012, pp. 16–26.