

Enabling Highly Scalable Remote Memory Access Programming with MPI-3 One Sided

By Robert Gerstenberger,* Maciej Besta, and Torsten Hoefler

Abstract

Modern high-performance networks offer remote direct memory access (RDMA) that exposes a process' virtual address space to other processes in the network. The *Message Passing Interface* (MPI) specification has recently been extended with a programming interface called MPI-3 *Remote Memory Access* (MPI-3 RMA) for efficiently exploiting state-of-the-art RDMA features. MPI-3 RMA enables a powerful programming model that alleviates many message passing downsides. In this work, we design and develop bufferless protocols that demonstrate how to implement this interface and 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 RMA functions that enable rigorous mathematical analysis of application performance and facilitate the development of codes that solve given tasks within specified time and energy budgets. We validate the usability of our library and models with several application studies with up to half a million processes. In a wider sense, our work illustrates how to use RMA principles to accelerate computation- and data-intensive codes.

1. INTRODUCTION

Supercomputers have driven the progress of various society's domains by solving challenging and computationally intensive problems in fields such as climate modeling, weather prediction, engineering, or computational physics. More recently, the emergence of the "Big Data" problems resulted in the increasing focus on designing high-performance architectures that are able to process enormous amounts of data in domains such as personalized medicine, computational biology, graph analytics, and data mining in general. For example, the recently established Graph500 list ranks supercomputers based on their ability to traverse enormous graphs; the results from November 2014 illustrate that the most efficient machines can process up to 23 trillion edges per second in graphs with more than 2 trillion vertices.

* RG 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. RG's primary email address is gerstenberger.robert@gmail.com.

Supercomputers consist of massively parallel nodes, each supporting up to hundreds of hardware threads in a single shared-memory domain. Up to tens of thousands of such nodes can be connected with a high-performance network, providing large-scale distributed-memory parallelism. For example, the Blue Waters machine has >700,000 cores and a peak computational bandwidth of >13 petaflops.

Programming such large distributed computers is far from trivial: an ideal programming model should tame the complexity of the underlying hardware and offer an easy abstraction for the programmer to facilitate the development of high-performance codes. Yet, it should also be able to effectively utilize the available massive parallelism and various heterogeneous processing units to ensure highest scalability and speedups. Moreover, there has been a growing need for the support for *performance modeling*: a rigorous mathematical analysis of application performance. Such formal reasoning facilitates developing codes that solve given tasks within the assumed time and energy budget.

The *Message Passing Interface* (MPI)¹¹ is the *de facto* standard API used to develop applications for distributed-memory supercomputers. MPI specifies message passing as well as remote memory access semantics and offers a rich set of features that facilitate developing highly scalable and portable codes; message passing has been the prevalent model so far. MPI's message passing specification does not prescribe specific ways how to exchange messages and thus enables flexibility in the choice of algorithms and protocols. Specifically, to exchange messages, senders and receivers may use eager or rendezvous protocols. In the former, the sender sends a message without coordinating with the receiver; unexpected messages are typically buffered. In the latter, the sender waits until the receiver specifies the target buffer; this may require additional control messages for synchronization.

Despite its popularity, message passing often introduces time and energy overheads caused by the rendezvous control messages or copying of eager buffers; eager messaging may also require additional space at the receiver. Finally, the fundamental feature of message passing is that it *ouples communication and synchronization*: a message both transfers the data and synchronizes the receiver with the sender. This may prevent effective overlap of computation and

The original version of this paper was published in the *Proceedings of the Supercomputing Conference 2013 (SC'13)*, Nov. 2013, ACM.

communication and thus degrade performance.

The dominance of message passing has recently been questioned as novel hardware mechanisms are introduced, enabling new high-performance programming models. Specifically, 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), enabling a process to directly access virtual memory at remote processes without involvement of the operating system or activities at the remote side. RDMA is supported by on-chip networks in, for example, Intel's SCC and IBM's Cell systems, as well as off-chip networks such as InfiniBand, IBM's PERCS or BlueGene/Q, Cray's Gemini and Aries, or even RDMA over Ethernet/TCP (RoCE/iWARP).

The RDMA support gave rise to *Remote Memory Access* (RMA), a powerful programming model that provides the programmer with a Partitioned Global Address Space (PGAS) abstraction that unifies separate address spaces of processors while preserving the information on which parts are local and which are remote. A fundamental principle behind RMA is that it *relaxes synchronization and communication* and allows them to be managed independently. Here, processes use independent calls to initiate data transfer and to ensure the consistency of data in remote memories and the notification of processes. Thus, RMA generalizes the principles from shared memory programming to distributed memory computers where data coherency is explicitly managed by the programmer to ensure highest speedups.

Hardware-supported RMA has benefits over message passing in the following three dimensions: (1) *time* by avoiding synchronization overheads and additional messages in rendezvous protocols, (2) *energy* by eliminating excessive copying of eager messages, and (3) *space* by removing the need for receiver-side buffering. Several programming environments embrace RMA principles: PGAS languages such as Unified Parallel C (UPC) or Fortran 2008 Coarrays and libraries such as Cray SHMEM or MPI-2 One Sided. Significant experience with these models has been gained in the past years^{1,12,17} and several key design principles for RMA programming evolved. Based on this experience, MPI's standardization body, the MPI Forum, has revamped the RMA (or One Sided) interface in the latest MPI-3 specification.¹¹ MPI-3 RMA supports the

newest generation of RDMA hardware and codifies existing RMA practice. A recent textbook⁴ illustrates how to use this interface to develop high-performance large-scale codes.

However, it has yet to be shown how to implement the new library interface to deliver highest performance at lowest memory overheads. In this work, we design and develop scalable protocols for implementing MPI-3 RMA over RDMA networks, requiring $\mathcal{O}(\log p)$ time and space per process on p processes. We demonstrate that the MPI-3 RMA interface can be implemented adding negligible overheads to the performance of the utilized hardware primitives.

In a wider sense, our work answers the question if the MPI-3 RMA interface is a viable candidate for moving towards exascale computing. Moreover, it illustrates that RMA principles provide significant speedups over message passing in both microbenchmarks and full production codes running on more than half a million processes. Finally, our work helps programmers to rigorously reason about application performance by providing a set of asymptotic as well as detailed performance models of RMA functions.

2. SCALABLE PROTOCOLS FOR RMA

We now describe protocols to implement MPI-3 RMA based on low-level RDMA functions. In all our protocols, we assume that we only have small bounded buffer space at each process ($\mathcal{O}(\log p)$ for synchronization, $\mathcal{O}(1)$ for communication), no remote software agent, and only put, get, and some basic atomic operations (atomics) for remote accesses. Thus, our protocols are applicable to all current RDMA networks and are forward-looking towards exascale network architectures.

We divide the RMA functionality of MPI into three separate concepts: (1) window creation, (2) communication functions, and (3) synchronization functions.

Figure 1a 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 1b 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 rest of this work.

Figure 1. An overview of MPI-3 RMA and associated cost functions. The figure shows abstract cost functions for all operations in terms of their input domains. (a) Synchronization and (b) Communication. 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 $\mathcal{P}: \{p\} \rightarrow \mathcal{T}$ 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.

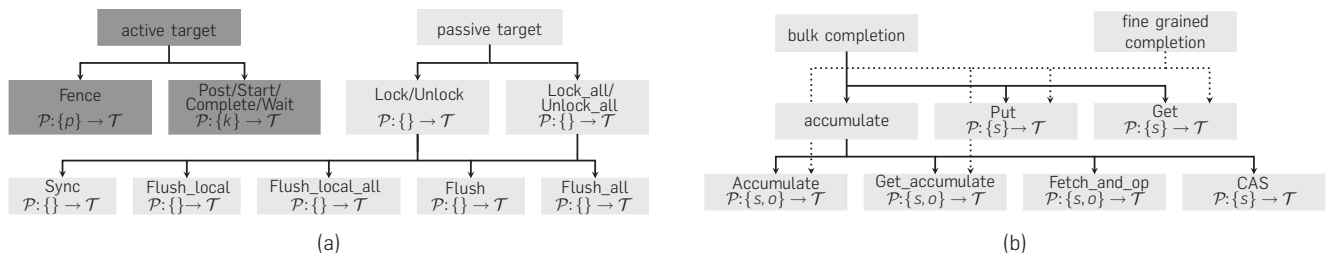


Figure 1 also shows abstract definitions of the performance models for each synchronization and communication operation. The performance model for each function depends on the exact implementation. We provide a detailed overview of the *asymptotic* as well as *exact* performance properties of our protocols and our implementation in the next sections. The different performance characteristics of communication and synchronization functions make a unique combination of implementation options for each specific use-case optimal. Yet, it is not always easy to choose this best variant. The exact models can be used to design close-to-optimal implementations (or as input for model-guided autotuning) while the simpler asymptotic models can be used in the algorithm design phase as exemplified by Karp et al.⁷

To support post-petascale computers, all protocols need to implement each function in a scalable way, that is, 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 a 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

Our reference implementation used to describe RMA protocols and principles is called FOMPI (fast one sided MPI). FOMPI is a fully functional MPI-3 RMA library implementation for Cray Gemini (XK5, XE6) and Aries (XC30)³ systems. In order to maximize asynchronous progression and minimize overhead, FOMPI interfaces to the lowest-level available hardware APIs.

For inter-node (network) communication, FOMPI uses the RDMA API of Gemini and Aries networks: Distributed Memory Application (DMAPP). DMAPP offers put, get, and a limited set of atomic memory operations for certain 8 Byte datatypes. For intra-node communication, we use XPMEM,¹⁶ a portable Linux kernel module that allows to map the memory of one process into the virtual address space of another. All operations can be directly implemented with load and store instructions, as well as CPU atomics (e.g., using the x86 lock prefix).

FOMPI's performance properties are self-consistent (i.e., respective FOMPI functions perform no worse than a combination of other FOMPI functions that implement the same functionality) 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 communication and synchronization functions over RDMA networks.

2.2. Scalable window creation

An MPI *window* is a region of process memory that is made accessible to remote processes. 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.

FOMPI can be downloaded from http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI.

Traditional Windows. These windows expose existing user-memory by specifying an arbitrary local base address. All remote accesses are relative to this address. Traditional windows are not scalable as they require $\Omega(p)$ storage on each of the p processes in the worst case. Yet, they are useful when the library can only access user-specified memory. Memory addresses are exchanged with two MPI_Allgather operations: one for DMAPP and one for XPMEM.

Allocated Windows. These windows allow the MPI library to allocate window memory and thus use identical base addresses on all nodes requiring only $\mathcal{O}(1)$ storage. This can be done with a system-wide *symmetric heap* or with the following POSIX-compliant protocol: (1) a leader process chooses a random address and broadcasts it to other 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 the processes (this can be checked with MPI_Allreduce). This mechanism requires $\mathcal{O}(\log p)$ time (with high probability).

Dynamic Windows. Here, windows can be dynamically resized by attaching or detaching memory regions with local MPI_Win_attach and MPI_Win_detach calls. They can be used in, for example, dynamic RMA-based data structures. In our implementation, the former call registers a memory region and inserts the information into a linked list; the latter removes a region from the list. Both calls require $\mathcal{O}(1)$ memory per region. The access to the list on a target is purely one sided. We use a local cache to reduce the number of remote accesses; a simple protocol uses gets to ensure the cache validity and to update local information if necessary.

Shared Memory Windows. These windows are only valid for intra-node communication, enabling efficient load and store accesses. They can be implemented with POSIX shared memory or XPMEM with constant memory overhead per core.⁵ We implement the intra-node case as a variant of allocated windows, providing identical performance and full compatibility with shared memory windows.

2.3. Communication functions

Communication functions map nearly directly to low-level hardware functions, enabling significant speedups over message passing. 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 atomics (for 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 ensures that the target is not involved in the communication for true passive mode. It can be improved if we allow buffering (enabling a space-time trade-off¹⁸) and active messages to perform the remote operations atomically.

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

2.4. Scalable window synchronization

MPI defines *exposure* and *access* epochs. A process starts an exposure epoch to allow other processes access to its

memory. 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. 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. All remote memory operations must be committed before leaving the fence call. We use an x86 `m fence` 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. This mode (also called “PSCW”) synchronizes a subset of processes of a window and thus enables synchronization at a finer granularity than that possible with fences. Exposure (`MPI_Win_post/MPI_Win_wait`) and access epochs (`MPI_Win_start/MPI_Win_complete`) can be opened and closed independently. A group argument is associated with each call that starts an epoch; it states all processes participating in the epoch. The calls have to ensure correct *matching*: if a process i specifies a process j in the group argument of the post call, then the next start call at process j with i in the group argument *matches* the post call.

Since our RMA implementation cannot assume buffer space for remote operations, it has to ensure that all processes in the group argument of the start call have issued a matching post before the start returns. Similarly, the wait call has to ensure that all matching processes have issued 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 matching protocol with a time and memory complexity of $\mathcal{O}(k)$ if each process has at most k neighbors across all epochs. We assume k is known to the protocol. We start with a high-level description: process i that *posts* an epoch announces itself to all processes j_1, \dots, j_l in the group argument by adding i to a list local to the processes j_1, \dots, j_l . Each process j that tries to *start* an access epoch waits until all processes i_1, \dots, 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. 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.

If k is the size of the group, then the number of operations 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.

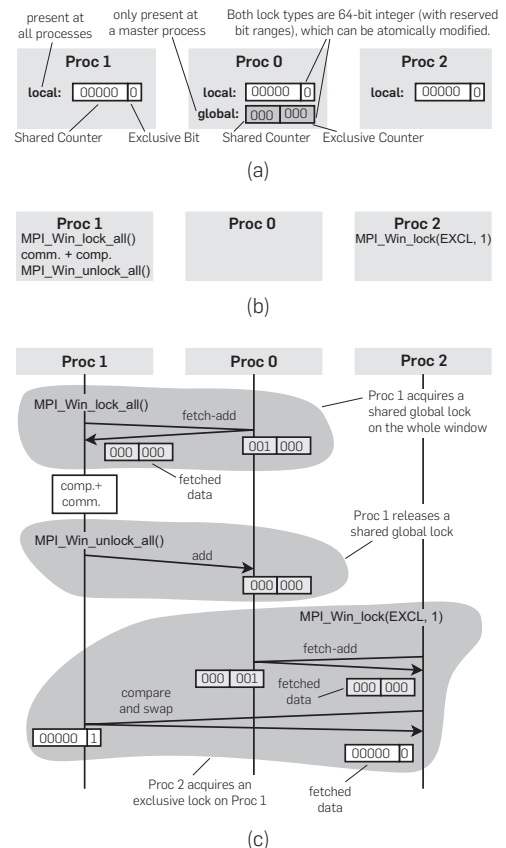
A more detailed explanation can be found in our SC13 paper.

Lock Synchronization. We now sketch a low-overhead and scalable strategy to implement shared global, shared process-local, and exclusive process-local locks on RMA systems (the MPI specification does not allow exclusive global locks). These mechanisms allows to synchronize processes and memories at very fine granularities. 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).

Each local lock variable is used to implement a reader-writer lock that allows one writer (*exclusive* lock), but many readers (*shared* locks). The highest order bit of the variable indicates a write access; the other bits are used to count the number of shared locks (cf. Ref.⁸). The global lock variable is split into two parts; they count the number of processes holding a shared global lock in the window and the number of exclusively locked processes, respectively. These variables enable all lock operations to complete in $\mathcal{O}(1)$ steps if a lock can be acquired immediately; they are pictured in Figure 2a.

Figure 2b shows an exemplary lock scenario for three processes. We omit a detailed description of the protocol due to the lack of space (the source code is available online); we describe a locking scenario to illustrate the core idea behind the protocol. Figure 2c shows a possible execution schedule for the scenario from Figure 2b. 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.

Figure 2. Example of lock synchronization. (a) Data structures, (b) Source code, and (c) A possible schedule.



An acquisition of a shared global lock (MPI_Win_lock_all) only involves the global lock on the master. The origin (Process 1) fetches and increases the lock in one atomic operation. Since there is no exclusive lock present, Process 1 can proceed. Otherwise, it would repeatedly (remotely) read the lock until no writer was present; exponential back off can be used to avoid congestion.

For a local exclusive lock, the origin needs to ensure two invariants: (1) no shared global lock **and** (2) no local shared or exclusive lock can be held or acquired during the local exclusive lock. For the first part, the origin (Process 2) atomically fetches the global lock from the master and increases the writer part to register for an exclusive lock. If the fetched value indicates lock all accesses, the origin backs off. As there is no global reader, Process 2 proceeds to the second invariant and tries to acquire an exclusive local lock on Process 1 using a compare-and-swap (CAS) with zero (cf. Ref.⁸). It succeeds and acquires the lock. If one of the two steps fails, the origin backs off and repeats the operation.

When unlocking (MPI_Win_unlock_all) a shared global lock, the origin only atomically decreases the global lock on the master. The unlocking of an exclusive lock requires two steps: clearing the exclusive bit of the local lock, and then atomically decreasing the writer part of the global lock.

The acquisition or release of a shared local lock (MPI_Win_lock/MPI_Win_unlock) is similar to the shared global case, except it targets a local lock.

If no exclusive locks exist, then shared locks (both local and global) only take one remote atomic operation. The number of remote requests while waiting can be bound by using MCS locks.⁹ An exclusive lock will take in the best case two atomic communication operations. Unlock operations always cost one atomic operation, except for the 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 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 (on x86) to the critical path.

3. DETAILED PERFORMANCE MODELING AND EVALUATION

We now analyze the performance 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 Coarrays, both specially tuned for Cray systems. We execute all benchmarks on the Blue Waters supercomputer, using Cray XE6 nodes. Each node contains four 8-core AMD Opteron 6276 (Interlagos) 2.3GHz CPUs and is connected to other nodes through a 3D-Torus Gemini network. Additional results can be found in the original SC13 paper.

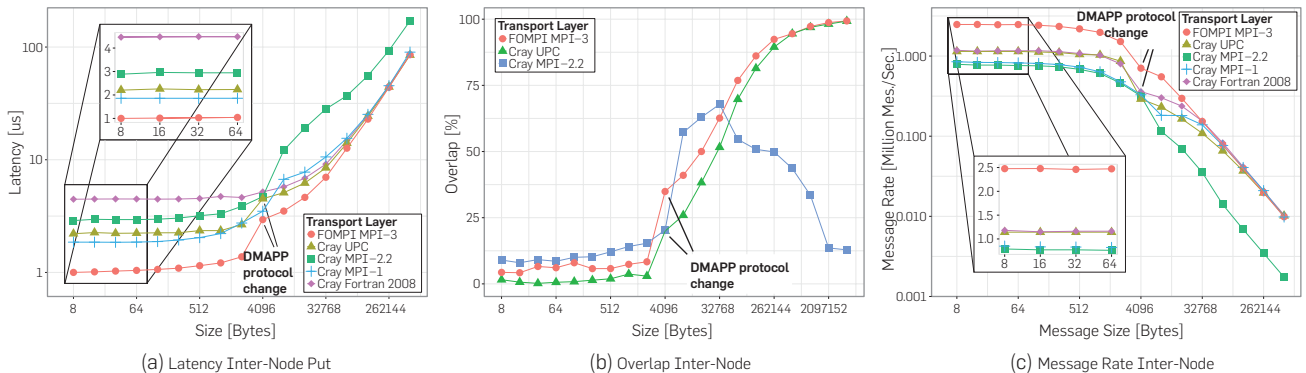
3.1. Communication

Comparing latency and bandwidth between RMA and point-to-point communication is not always fair since RMA communication may require extra synchronization to notify the target. For all RMA latency results we ensure remote completion (the data is committed in remote memory) but no synchronization. We analyze synchronization costs separately in Section 3.2.

Latency and Bandwidth. We start with the analysis of latency and bandwidth. The former is important in various latency-constrained codes such as interactive graph processing frameworks and search engines. The latter represents a broad class of communication-intensive applications such as graph analytics engines or distributed key-value stores.

We measure point-to-point latency with standard ping-pong techniques. Figure 3a shows the latency for varying message sizes for inter-node put. Due to the highly optimized fast-path, FOMPI has >50% lower latency than other PGAS models while achieving the same bandwidth for larger messages. The performance functions (cf. Figure 1) are: $\mathcal{P}_{put} = 0.16ns \cdot s + 1\mu s$ and $\mathcal{P}_{get} = 0.17ns \cdot s + 1.9\mu s$.

Figure 3. Microbenchmarks: (a) Latency comparison for put with DMAPP communication. Note that message passing (MPI-1) implies remote synchronization while UPC, Fortran 2008 Coarrays, and MPI-2.2/3 only guarantee consistency. (b) Communication/computation overlap for put over DMAPP, Cray MPI-2.2 has much higher latency up to 64 KB (cf. a), thus allows higher overlap. (c) Message rate for put communication.



Overlapping Computation. Overlapping computation with communication is a technique in which computation is progressed while waiting for communication to be finished. Thus, it reduces the number of idle CPU cycles. Here, we measure how much of such overlap can be achieved with the compared libraries and languages. The benchmark calibrates a computation loop to consume slightly more time than the latency. Then it places computation between communication and synchronization and measures the combined time. The ratio of overlapped computation is then computed from the measured communication, computation, and combined times. Figure 3b shows the ratio of the overlapped communication for Cray’s MPI-2.2, UPC, and FOMPI.

Message Rate. This benchmark is similar to the latency benchmark. However, it benchmarks the start of 1000 transactions without synchronization to determine the overhead for starting a single operation. Figure 3c presents the results for the inter-node case. Here, injecting a single 8 Byte operation costs only 416ns.

Atomics. As the next step we analyze the performance of various atomics that are used in a broad class of lock-free and wait-free codes. Figure 4a 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 $\mathcal{P}_{acc,sum} = 28ns \cdot s + 2.4\mu s$, $\mathcal{P}_{acc,min} = 0.8ns \cdot s + 7.3\mu s$, and $\mathcal{P}_{CAS} = 2.4\mu s$. The DMAPP acceleration lowers the latency for small operations while the locked implementation exhibits a higher bandwidth. However, this does not consider the serialization due to the locking.

3.2. Synchronization schemes

Finally, we evaluate synchronization schemes utilized in numerous parallel protocols and systems. The different synchronization modes have nontrivial trade-offs. For example PSCW performs better for small groups of processes and fence performs best for groups that 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).

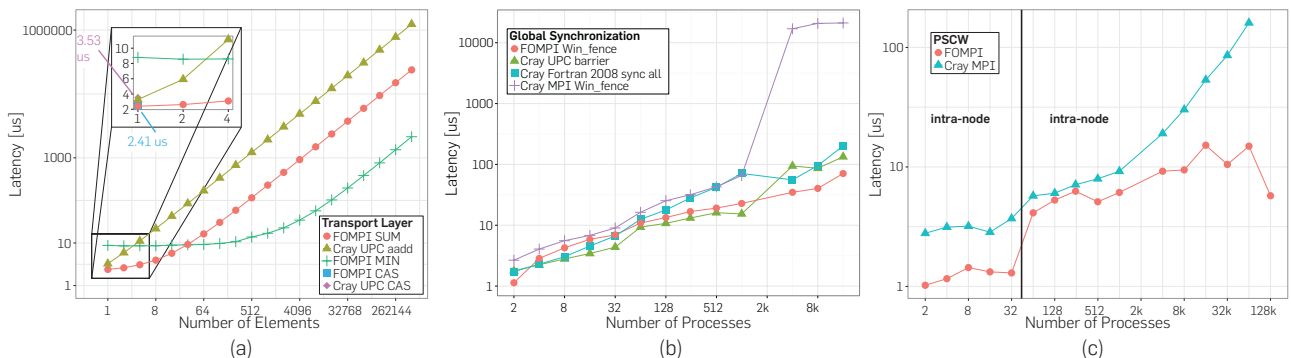
Global Synchronization. Global synchronization is performed in applications based on the Bulk Synchronous Parallel (BSP) model. It is offered by fences in MPI. It can be directly compared to Fortran 2008 Coarrays sync all and UPC’s upc_barrier which also synchronize the memory at all processes. Figure 4b compares the performance of FOMPI with Cray’s MPI-2.2, UPC, and Fortran 2008 Coarrays implementations. The performance function for FOMPI’s fence implementation is: $\mathcal{P}_{fence} = 2.9\mu s \cdot \log_2(p)$.

General Active Target Synchronization (PSCW). This mode may accelerate codes where the communication graph is static or changes infrequently, for example stencil computations. Only MPI offers PSCW. Figure 4c shows the performance for Cray MPI-2.2 and FOMPI when synchronizing a ring where each process has exactly two neighbors ($k = 2$). An ideal implementation would exhibit constant time. We observe systematically growing overheads in Cray’s MPI as well as system noise (due to network congestion, OS interrupts and daemons, and others) on runs with >1000 processes with FOMPI. We model the performance with varying numbers of neighbors and FOMPI’s PSCW synchronization costs involving k off-node neighbor are $\mathcal{P}_{post} = \mathcal{P}_{complete} = 350ns \cdot k$, and $\mathcal{P}_{start} = 0.7\mu s$, $\mathcal{P}_{wait} = 1.8\mu s$ (without noise).

Passive Target Synchronization. Finally, we evaluate lock-based synchronization that can be utilized to develop high-performance distributed-memory variants of shared-memory lock-based codes. The performance of lock/unlock is constant in the number of processes as ensured by our protocols and thus not graphed. The performance functions are $\mathcal{P}_{lock,excl} = 5.4\mu s$, $\mathcal{P}_{lock,shrd} = \mathcal{P}_{lock,all} = 2.7\mu s$, $\mathcal{P}_{unlock,shrd} = \mathcal{P}_{unlock,all} = 0.4\mu s$, $\mathcal{P}_{unlock,excl} = 4.0\mu s$, $\mathcal{P}_{flush} = 76ns$, and $\mathcal{P}_{sync} = 17ns$.

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

Figure 4. Performance of atomic accumulate operations and synchronization latencies. (a) Atomic Operation Performance, (b) Latency for Global Synchronization, and (c) Latency for PSCW (Ring Topology).



4. ACCELERATING FULL CODES WITH RMA

To compare our protocols and implementation with the state of the art, we analyze a 3D FFT code as well as the MIMD Lattice Computation (MILC) full production application with several hundred thousand lines of source code that performs quantum field theory computations. Other application case-studies can be found in the original SC13 paper, they include a distributed hashtable representing many big data and analytics applications and a dynamic sparse data exchange representing graph traversals and complex modern scientific codes such as n-body methods.

In all the codes, we keep most parameters constant to compare the performance of PGAS languages, message passing, and MPI RMA. Thus, we did not employ advanced concepts, such as MPI datatypes or process topologies, which are not available in all designs (e.g., UPC and Fortran 2008).

4.1. 3D fast Fourier transform

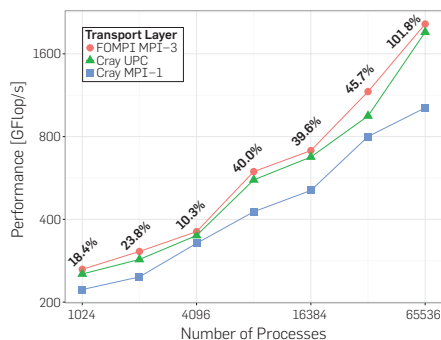
We now discuss how to exploit overlap of computation and communication in a 3D Fast Fourier Transformation. We use Cray's MPI and UPC versions of the NAS 3D FFT benchmark. Nishtala et al.¹² and Bell et al.¹ demonstrated that overlap of computation and communication can be used to improve the performance 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 5 illustrates the results for the strong scaling class D benchmark ($2048 \times 1024 \times 1024$). UPC achieves a consistent speedup over message passing, mostly due to the communication and computation overlap. FOMPI has a somewhat lower static overhead than UPC and thus enables better overlap (cf. Figure 3b) and slightly higher performance.

4.2. MIMD lattice computation

The MIMD Lattice Computation (MILC) Collaboration studies Quantum Chromodynamics (QCD), the theory of strong interaction.² The group develops a set of applications, known as the MILC code, which regularly gets one of the largest allocations at US NSF supercomputer centers. The

Figure 5. 3D FFT Performance. The annotations represent the improvement of foMPI over message passing.



su3_rmd module, which is part of the SPEC CPU2006 and SPEC MPI benchmarks, is included in the MILC code.

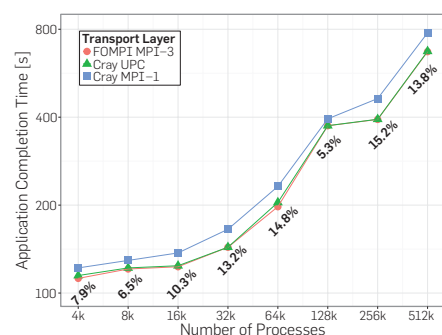
The program performs a stencil computation on a 4D rectangular grid and it decomposes the domain in all four dimensions to minimize the surface-to-volume ratio. To keep data consistent, neighbor communication is performed in all eight directions. Global allreductions are done regularly to check the solver convergence. The most time-consuming part of MILC is the conjugate gradient solver which uses nonblocking communication overlapped with local computations.

Figure 6 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 computation phases (e.g., CG) execute up to 45% faster, yet, we chose to report full-code performance. Cray's UPC and FOMPI exhibit essentially the same performance, while the UPC code uses Cray-specific tuning¹⁵ and the MPI-3 code is portable to different architectures. The full-application performance gain over Cray's MPI-1 version is more than 15% for some configurations. The application was scaled successfully to up to 524,288 processes with all implementations. This result and our microbenchmarks demonstrate the scalability and performance of our protocols and that the MPI-3 RMA library interface can achieve speedups competitive to compiled languages such as UPC and Fortran 2008 Coarrays while offering all of MPI's convenient functionalities (e.g., Topologies and Datatypes). Finally, we illustrate that the new MPI-3 RMA semantics enable full applications to achieve significant speedups over message passing in a fully portable way. Since most of those existing codes are written in MPI, a step-wise transformation can be used to optimize most critical parts first.

5. RELATED WORK

PGAS programming has been investigated in the context of UPC and Fortran 2008 Coarrays. For example, an optimized UPC Barnes Hut implementation shows similarities to MPI-3 RMA programming by using bulk vectorized memory transfers combined with vector reductions instead of shared pointer accesses.¹⁷ Highly optimized PGAS applications often use a style that can easily be adapted to MPI-3 RMA.

Figure 6. Full MILC code execution time. The annotations represent the improvement of foMPI over message passing.



The intricacies of MPI-2.2 RMA implementations over InfiniBand networks have been discussed by Jian et al.⁶ and Santhanaraman et al.¹⁴ Zhao et al.¹⁸ describe an adaptive strategy to switch from eager to lazy modes in active target synchronizations in MPICH 2. This mode could be used to speed up these of FoMPI's atomics that are not supported in hardware.

The applicability of MPI-2.2 RMA has also been demonstrated for some applications. Mirin and Sawyer¹⁰ discuss the usage of MPI-2.2 RMA coupled with threading to improve the Community Atmosphere Model (CAM). Potluri et al.¹³ show that MPI-2.2 RMA with overlap can improve the communication in a Seismic Modeling application. However, we demonstrated new MPI-3 features, such as lock-all epochs, flushes, and allocated windows, which can be used to further improve performance by utilizing state-of-the-art RDMA features and simplified implementations.


6. DISCUSSION AND CONCLUSION

In this work, we demonstrate how the MPI-3 RMA library interface can be implemented over RDMA networks to achieve highest performance and lowest memory overheads. We provide detailed performance models that help choosing among the multiple options. For example, a user can decide whether to use Fence or PSCW synchronization (if $\mathcal{P}_{fence} > \mathcal{P}_{post} + \mathcal{P}_{complete} + \mathcal{P}_{start} + \mathcal{P}_{wait}$, which is true for large k). This is just one example for the possible uses of the provided detailed performance models.

We study all overheads in detail and provide performance evaluations for all critical RMA functions. 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. These gains will directly translate to significant energy savings in big data and HPC computations.

We expect that the principles and scalable synchronization algorithms developed in this work will act as a blueprint for optimized RMA implementations over future large-scale RDMA networks. We also conjecture that the demonstration of highest performance to users will quickly increase the number of RMA programs. Finally, as the presented techniques can be applied to data-centric codes, we expect that RMA programming will also accelerate emerging data center computations.

Acknowledgments

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 NASFT. 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. MB is supported by the 2013 Google European Doctoral Fellowship in Parallel Computing. 

References

- Bell, C., Bonachea, D., Nishtala, R., Yelick, K. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the International Conference on Parallel and Distributed Processing (IPDPS'06)* (2006). IEEE Computer Society, 1–10.
- Bernard, C., Ogilvie, M.C., DeGrand, T.A., DeTar, C.E., Gottlieb, S.A., Krasnitz, A., Sugar, R., Toussaint, D. Studying quarks and gluons on MIMD parallel computers. *J. High Perform. Comput. Appl.* 5, 4 (1991), 61–70.
- Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., Reinhard, J. 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)* (2012). IEEE Computer Society, Los Alamitos, CA, 103:1–103:9.
- Gropp, W., Hoefler, T., Thakur, R., Lusk, E. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, Nov. (2014).
- Hoefler, T., Dinan, J., Buntinas, D., Balaji, P., Barrett, B., Brightwell, R., Gropp, W., Kale, V., Thakur, R. Leveraging MPI's one-sided communication interface for shared-memory programming. In *Recent Advances in the Message Passing Interface (EuroMPI'12)*, Volume LNCS 7490 (2012). Springer, 132–141.
- Jiang, W., Liu, J., Jin, H.-W., Panda, D.K., Gropp, W., Thakur, R. High performance MPI-2 one-sided communication over InfiniBand. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'04)* (2004). IEEE Computer Society, 531–538.
- Karp, R.M., Sahay, A., Santos, E.E., Schauer, K.E. Optimal broadcast and summation in the LogP model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)* (1993). ACM, New York, NY, USA, 142–153.
- Mellor-Crummey, J.M., Scott, M.L. Scalable reader-writer synchronization for shared-memory multiprocessors. *SIGPLAN Notices* 26, 7 (1991), 106–113.
- Mellor-Crummey, J.M., Scott, M.L. Synchronization without contention. *SIGPLAN Notices* 26, 4 (1991), 269–278.
- Mirin, A.A., Sawyer, W.B. A scalable implementation of a finite-volume dynamical core in the community atmosphere model. *J. High Perform. Comput. Appl.* 19, 3 (2005), 203–212.
- MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.0* (2012).
- Nishtala, R., Hargrove, P.H., Bonachea, D.O., Yelick, K.A. 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)* (2009). IEEE Computer Society, 1–12.
- Potturi, S., Lai, P., Tomko, K., Sur, S., Cui, Y., Tatineni, M., Schulz, K.W., Barth, W.L., Majumdar, A., Panda, D.K. Quantifying performance benefits of overlap using MPI-2 in a seismic modeling application. In *Proceedings of the ACM International Conference on Supercomputing (ICS'10)* (2010). ACM 17–25.
- Santhanaraman, G., Balaji, P., Gopalakrishnan, K., Thakur, R., Gropp, W., Panda, D.K. 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)* (2009), 380–387.
- Shan, H., Austin, B., Wright, N., Strohmaier, E., Shalf, J., Yelick, K. Accelerating applications at scale using one-sided communication. In *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS'12)* (2012).
- Woodacre, M., Robb, D., Roe, D., Feind, K. *The SGI Altix TM 3000 Global Shared-Memory Architecture* (2003). SGI HPC White Papers.
- Zhang, J., Behzad, B., Snir, M. Optimizing the Barnes-Hut algorithm in UPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011). ACM, 75:1–75:11.
- Zhao, X., Santhanaraman, G., Gropp, W. Adaptive strategy for one-sided communication in MPICH2. In *Recent Advances in the Message Passing Interface (EuroMPI'12)* (2012). Springer, 16–26.

Robert Gerstenberger, Maciej Besta, and Torsten Hoefler ([robertge, maciej.best, htor]@inf.ethz.ch) ETH Zurich, Switzerland.