Beyond MPI: New Communication Interfaces for Database Systems and Data-Intensive Applications

Feilong Liu[1,*], Claude Barthels[2,*], Spyros Blanas[1], Hideaki Kimura[3], Garret Swart[3]
[1]The Ohio State University, [2]ETH Zurich, [3]Oracle Corp.
[1]{liu.3222, blanas.2}@osu.edu, [2]claudeb@inf.ethz.ch, [3]{hideaki.kimura, garret.swart}@oracle.com

ABSTRACT

Networks with Remote Direct Memory Access (RDMA) support are becoming increasingly common. RDMA, however, offers a limited programming interface to remote memory that consists of read, write and atomic operations. With RDMA alone, completing the most basic operations on remote data structures often requires multiple round-trips over the network. Data-intensive systems strongly desire higher-level communication abstractions that support more complex interaction patterns.

A natural candidate to consider is MPI, the de facto standard for developing high-performance applications in the HPC community. This paper critically evaluates the communication primitives of MPI and shows that using MPI in the context of a data processing system comes with its own set of insurmountable challenges. Based on this analysis, we propose a new communication abstraction named RDMO, or Remote Direct Memory Operation, that dispatches a short sequence of reads, writes and atomic operations to remote memory and executes them in a single round-trip.

1. INTRODUCTION

Popular high-speed interconnects such as InfiniBand offer Remote Direct Memory Access (RDMA) support to accelerate communication. Using RDMA in a database system is not straightforward: database systems need to organize their data in such a way that direct data access mechanisms can be employed [5]. This entails redesigning data processing algorithms [2], data structures [17], and communication mechanisms [10, 15]. The Oracle database has been using RDMA for performance and scalability, but its use has been limited to operations that do not require many rounds of messaging, such as accessing the Commit Cache and the Undo blocks during transaction logging [16]. The fundamental challenge is the limited programming surface of RDMA consisting only of message passing, read/write, and single-word atomic operations.

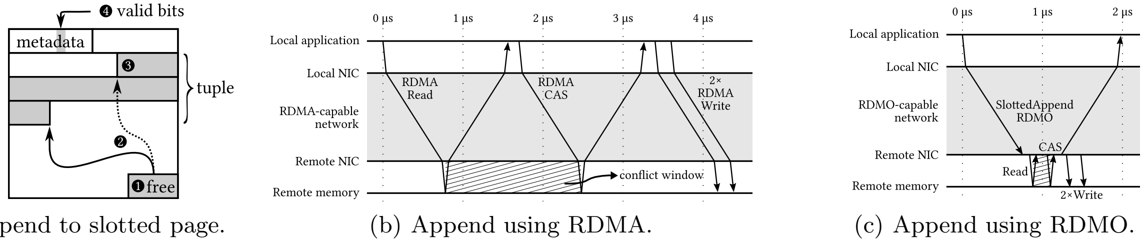
A natural higher-level alternative to consider is MPI, the de facto standard for developing portable and highly parallel applications in the HPC community. MPI has been used in distributed database systems [6] and to implement scalable join algorithms [3]. At a first glance, one is inclined to believe that MPI would be a good fit for data processing. However, Section 2 shows that MPI is profoundly unsatisfactory for database systems:

- (§2.1) MPI has a process-centric model that lacks the concept of a thread for multi-threading.
- (§2.2, §2.3) Many MPI operations can only be performed synchronously (collectively) between a group of processes. Implementations often use barrier-like synchronization which exposes communication delays and underutilizes the CPU. In addition processes cannot dynamically join and leave groups, which is needed for operations whose communication pattern is determined by the data.
- (§2.4) For correctness, remote memory operations need to either acquire coarse-grained locks or manually serialize the execution of operations, akin to flushing I/O buffers to persistent storage.
- (§2.5 – §2.7) MPI cannot (1) notify the remote side of the completion of a memory operation, (2) convey quality of service (QoS) and traffic differentiation information, and (3) support elasticity, fault tolerance and high availability.

Instead of adopting MPI, our idea is to augment RDMA to support the dispatch of simple data processing logic in one "unit" to the remote side which will be executed in one round-trip. We term this a <u>R</u>emote <u>D</u>irect <u>M</u>emory <u>O</u>peration, or an RDMO.

The RDMO abstraction overcomes limitations of RDMA when manipulating data structures. Consider the common database operation of inserting a tuple in a slotted page, shown in Figure 1(a). A latch-free algorithm checks if there is enough free space, then atomically modifies the pointer to the "free" segment of the page, writes the data, and fi-

[*]Contributed equally.



(a) Append to slotted page. (b) Append using RDMA. (c) Append using RDMO.

Figure 1: A <u>R</u>emote <u>D</u>irect <u>M</u>emory <u>O</u>peration, or RDMO, dispatches data processing logic to the remote side in one "unit".

nally marks the entry as valid to read. An RDMA implementation issues the sequence of requests shown in Figure 1(b). The RDMO implementation issues a single `SlottedAppend` RDMO as shown in Figure 1(c). In this example, the RDMO abstraction reduces the number of transmitted messages by 4× and reduces latency by as much as 2×. Furthermore, the window of a conflict for the atomic operation is now reduced by 10×. Section 3 introduces more RDMO examples.

Prior research has proposed other programming models and abstractions to overcome the shortcomings of MPI. Global Arrays [7] is a programming model that presents the view of consistent global memory. Like MPI, it is limited to one-sided Get, Put, Accumulate and atomic operations that are oblivious to the payload. A more recent effort is DPI [1], an interface definition for modern networks with "flow" and "memory" operations. Like MPI, DPI does not define mechanisms for pushing computation into the network. We envision that RDMOs will be one such mechanism.

2. A CRITIQUE OF MPI

The Message Passing Interface (MPI) is a communication interface that allows HPC applications to communicate in an efficient and portable manner. We evaluate two popular implementations of MPI, MVAPICH 2.2 [12] and OpenMPI 2.0 [13], on a cluster connected with InfiniBand FDR (56 Gbps). We focus on instances where the MPI abstractions are unsatisfactory for data management.

2.1 Multi-threading support

The unit of parallelism in MPI is a process. The MPI library controls the binding of processes to compute cores and hides this binding behind a namespace where processes are addressed solely through their rank number. However, in a database system, the sender of a message often targets a recipient based on its position in the local compute topology, such as when transmitting to a specific NUMA node or thread. No mechanism in MPI can address a message to a specific thread of an MPI process.

The performance of multi-threaded programs suffers due to contention inside the MPI library. Figure 2 shows the throughput of MVAPICH on a cluster with two nodes, where one node keeps sending data to the other with both one-sided and two-sided functions. In the "multi-process" result, each node runs 20 separate MPI processes. In the "multi-thread" result, each node runs 1 MPI process with 20 threads. The multi-threaded MPI performance is 2× to 20× slower than multi-process MPI.

2.2 Communication groups

Every communication in MPI targets a specific *communication group*. Within a communication group, the unit of parallelism is a *process* which is identified by a group-specific integer called a *rank*. To manage communication, database systems have to use different communication groups per operation. Furthermore, the communication patterns in a database system are often data-dependent. A weakness of MPI is that it does not permit processes to join or leave a communication group at runtime. Creating new groups at runtime is inefficient: the creation time is about 30ms with 4 nodes (60 processes) per group. As a result, MPI groups need to be generated conservatively to include every process that could potentially contribute to the result. MPI processes that have no data to contribute still have to participate in remote operations with zero-sized payloads for synchronization purposes.

2.3 Blocking collective operations

Many MPI operations must be executed synchronously by every process. MPI refers to these operations as *collective operations*. Some collective operations are implemented as blocking calls, meaning that a process is only allowed to continue once all other processes have executed the operation as well. Examples of collective operations include reduce, gather, scatter, and broadcast. Management operations, such as window allocation and deallocation, are also implemented as collective operations.

In the context of data management, using blocking collectives is problematic due to the inability

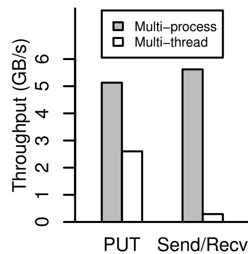


Figure 2: MPI performs poorly with multiple threads.

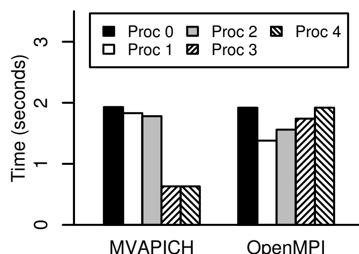


Figure 3: Collective MPI calls lead to unpredictable performance.

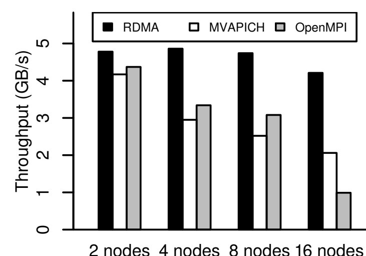


Figure 4: Bypassing MPI and directly using RDMA has better scalability.

of MPI implementations to interleave communication with computation. This is exacerbated if processes manipulate different amounts of data due to skew. Consider an `MPI_Gather` collective operation among 4 processes. Assume processes 1–4 call `MPI_Gather` to send 1 GB of data to process 0. However, processes 1–2 are late in calling `MPI_Gather`, while processes 3–4 call `MPI_Gather` immediately. Figure 3 shows the time when each process returns from the collective call. In the MVAPICH implementation, `MPI_Gather` is not blocking, hence process 3 and 4 exit the collective call earlier than process 1 and 2. However, in the OpenMPI implementation, `MPI_Gather` is blocking, so processes 3–4 are blocked — in fact, they finish after processes 1–2 complete despite starting first. This wastes valuable cycles waiting for the collective operation.

2.4 Synchronizing concurrent accesses

RMA operations in MPI are performed on memory regions referred to as *window* objects. A window is a collection of memory regions on which write ("put") and read ("get") operations can be executed. MPI provides two mechanisms to synchronize concurrent accesses to the same window.

In active target synchronization, both the origin and the target synchronize the remote memory access by adding a fence ("epoch") between RMA operations. Pending operations are only guaranteed to be visible after the epoch completes.

In passive target synchronization, MPI provides a coarse-grained locking mechanism. Before a remote memory access can take place, the entire window must be locked. These accesses are guaranteed to have completed only when the window is unlocked. MPI supports two locking modes: an exclusive lock will only grant access to a specific process to modify or read the content of the window, whereas a shared lock allows for concurrent accesses.

The problem of both concurrency control mechanisms is their granularity: locking an entire window is too coarse-grained and defining every operation in terms of epochs is too fine-grained. MPI lacks the sophistication of multi-level locking [8] that lets the system determine the granularity of access control.

2.5 Composability and low-level access

MPI assumes total control of the underlying hardware and it cannot co-exist harmoniously with low-level network access mechanisms. Many low-level optimizations are mutually exclusive with using MPI. Two examples are:

1. **Unreliable datagrams:** A database system may not require ordered data delivery if it is evaluating a query that does not rely on a sorted order. In this case, it is acceptable for the network to deliver messages out of order. Database systems can unlock better performance and scalability by using the Unreliable Datagram (UD) transport [9, 10], a datagram-based protocol similar to UDP in Ethernet. Figure 4 shows the throughput of a repartitioning operation that transmits each tuple to a destination node based on the hash value of its key. Using the UD transport in InfiniBand has better performance and scalability than the Reliable Connection (RC) transport that is used by many MPI implementations.
2. **Notified one-sided access:** When executing a query or a transaction, the database system can perform one-sided RMA operations to place data at specific locations. For many algorithms, merely writing the data to remote memory is insufficient and specific operations need to be triggered on the remote side after the data has been transmitted. Many network interfaces support notified one-sided RMA accesses to solve this problem. Exposing this functionality in MPI is a topic of active research [4], as notified one-sided accesses are not supported by the MPI standard.

2.6 Traffic differentiation

Many interconnects have the capability to provide better service to selected network traffic. For example, InfiniBand includes quality of service (QoS)

Table 1: MPI does not differentiate traffic.

	MVAPICH	OpenMPI	RDMA
Short message latency (msec)	186.08	0.06	0.03
Bulk transfer latency (msec)	186.08	167.92	188.99

mechanisms inherently and offers flow prioritization. Traffic differentiation is crucial for database systems as some communications are latency-critical, such as messages for algorithmic coordination or transaction processing, while many analytical workloads can tolerate higher latency in exchange for high-bandwidth transfers. However, MPI lacks mechanisms to associate quality of service information to different requests. Database systems are thus relying on the intricacies of the specific implementation of MPI they use for timely message delivery.

To quantify this limitation, we initiate a bulk data transfer (1 GB) and a small latency-critical message transmission (32 KB) using asynchronous send/receive calls. As seen in Table 1, in MVAPICH the small message is transmitted after the large data transfer, while both OpenMPI and RDMA transmit the small message earlier. This experiment shows that the order in which concurrent messages are delivered is implementation-dependent and cannot be controlled by the database system.

2.7 Elasticity, availability, fault tolerance

The degree of parallelism in MPI is specified when the application starts and it is fixed for the entire duration of the program. MPI cannot scale in or scale out a parallel database deployment, as it lacks the functionality to add and remove processes from a running application.

In addition, many HPC applications are started with a specific lifespan in mind which rarely exceeds several hours. MPI does not offer fault-tolerance features as it is expected that every program would terminate at some point anyway. (In fact, the default error handler in MPI terminates the entire program when a single process fails.) This application-centric execution model is radically different from the service-centric model that expects nodes to fail during the lifespan of a typical deployment. In other words, a database system should outlast the hardware it is currently running on. In this model, failures have to be contained and recovered from, and the communication interface needs to support fault tolerance and high-availability mechanisms.

3. A NEW ABSTRACTION: RDMO

This section introduces a new abstraction for data-intensive applications, the Remote Direct Memory Operation (RDMO) interface. An RDMO is a request that triggers the execution of a short sequence of reads, writes and atomic memory operations that will be transmitted and executed at the remote node without interrupting its CPU.

The RDMO communication pattern bridges the gap between direct memory accesses (RDMA) and remote procedure calls (RPC). Unlike RDMA and its fixed set of verbs, the RDMO interface can both transmit data and execute simple operations on a remote node. Unlike an RPC call, the RDMO interface limits the ability of the remote operation to execute too many instructions, access too much data, or block for too long.

3.1 Implementing RDMOs

The RDMO interface builds on RDMA and reuses the queue pair (QP) and completion queue (CQ) objects. Like one-sided RDMA verbs, RDMOs are transported over a Reliable Connection. Transport errors and certain operation completion codes can trigger the server to close the connection.

The RDMO interface deviates when it comes to the processing of each message, which is shown in Figure 5. Instead of targeting a memory location with byte-level reads and writes, an RDMO request targets an abstract data type (ADT) which is identified using a capability (a secret identifier). The RDMO operation must map to a valid method of the abstract data type. Each operation accepts an input byte array, and returns a completion code and an output byte array to the CQ. The format and meaning of the byte arrays are determined by the abstract data type method.

When an RDMO-capable endpoint receives the RDMO request, it is placed in a queue of incoming operations. If the queue is full, the RDMO is rejected and the client CQ receives an error. The RDMO is processed when it reaches the head of the queue. Processing an RDMO starts by mapping the capability to an instance of an abstract data type. If there is no such instance, or the identifier does not map to a method of the abstract data type, the operation returns with an error. Otherwise, the appropriate ADT method is invoked.

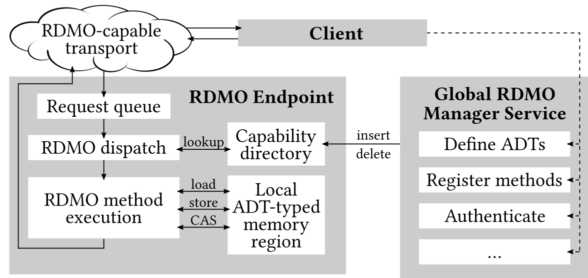


Figure 5: Implementing RDMOs.

The implementor of an RDMO method must certify that it is non-blocking. Each RDMO method has a fixed maximum resource consumption (in cycles) that is provided by the implementor when the RDMO method is registered on the server. Knowledge of the maximum RDMO resource consumption allows the RDMO-capable endpoint to bound the RDMO wait and service time. This avoids the need of timer signals or call progress acknowledgements as part of RDMO transport and execution, and reduces the overhead on the client and the server.

Limiting RDMO methods in this way during registration achieves higher availability, because the completion of an RDMO does not depend (wait) on other activities of the system. In addition, since the RDMO implementation can be fully determined and inlined, it can run outside of the typical application environment. This acts as a layer of abstraction between the endpoint capabilities and the endpoint implementation: an RDMO-capable endpoint can run as a separate user process, an interrupt handler in the OS kernel, a separate VM sharing memory, an FPGA, or even a special purpose chip.

We emphasize that the RDMO interface is not meant to replace existing RDMA or RPC interfaces. To the contrary, the RDMO interface depends on RDMA and RPC: First, the RDMO interface reuses key components of the RDMA mechanism. Second, the RDMO interface requires a higher-level RPC facility that defines new ADT instances, authenticates agents, and authorizes the use of a given ADT by creating a RDMO capability.

3.2 Attributes of RDMOs vs. MPI

RDMOs are embedded into a larger context of a network interface with the following properties:

Multiple Threads: As shown in Section 2.1, multi-threaded MPI programs suffer from poor performance. MPI connections are bound to processes, thus making the process rank both a unit of computation and an address for data accesses. The RDMO interface implements endpoints that are independent of the compute components instead.

Non-Blocking Operations: Section 2.3 shows that the behavior of MPI depends on the specific implementation and the blocking implementations fails to interleave the communication and computation. All functions of the RDMO interface are non-blocking such that the initiator of an operation can interleave communication and computation.

Quality of Service: Section 2.6 shows that MPI cannot prioritize different communication flows. However, database systems have to support many types of workloads that exhibit different communication patterns and some flows need to be prioritized. The RDMO interface provides QoS functionality using the RDMA QoS semantics.

Table 2: Attributes of RDMO operations.

RDMO	Schema aware?	Conditional?	Message savings
SlottedAppend	×	✓	4× or more
ConditionalGather	✓	✓	Up to 30×
SignaledRead	×	✓	3× or more
WriteAndSeal	×	×	2×
ScatterAndAccumulate	✓	✓	Up to 30×

Fault Tolerance: Section 2.7 highlights the lack of fault tolerance mechanisms in MPI. However, database systems offer services that must be highly available. The RDMO interface has a clearly-defined failure model, inherited by the RDMA interface, and allows applications to react to failures.

Schema awareness: Database systems operate on structured data. Pushing down schema information to the network using the abstract data type (ADT) support of RDMOs enables novel in-network processing applications and operations.

Conditional operations: Conditional operations allow the developer to evaluate simple *if-then* operations in the remote node. For an RDMO operation with condition check, the remote network card first evaluates if the remote data is in the specified state before applying the operation. This eliminates several round trips and reduces the need for running expensive synchronization or agreement protocols.

3.3 Five database RDMOs

This section presents five common operations in database systems that can be accelerated using the RDMO interface. The attributes of the five RDMO functions are summarized in Table 2. "Message savings" represents the number of messages saved compared with an RDMA implementation of the same function, if one assumes the same number of scatter/gather entries per request (30) as provided by modern InfiniBand network cards. These operations are used to demonstrate the potential of the RDMO interface and are not an exhaustive list.

1. SlottedAppend. This is the common operation of appending a tuple in a buffer, which has been highlighted in Figure 1(a) in Section 1. In case of contention, lock conflicts will increase exponentially. Performing this operation as an RDMO shrinks the conflict window and permits more concurrency under contention.

2. ConditionalGather. Traversing common data structures often requires following pointers. This requires several lookups over RDMA. This RDMO traverses pointer-based data structures, evaluates a user-defined predicate and gathers the matching

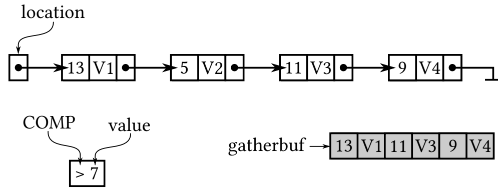


Figure 6: `ConditionalGather` follows a linked list of versions, performs visibility checks and returns the matching tuples in one operation.

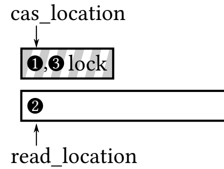


Figure 7: `SignaledRead` can elide a lock when reading any lock-based data structure in one request.

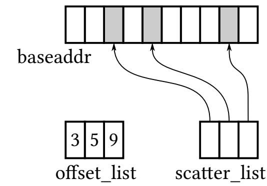


Figure 8: `ScatterAndAccumulate` accumulates elements in the list at user-defined offsets.

elements in a buffer that is transmitted back in a single request. Up to 30 elements (the gather list) can be retrieved in a single RDMO request, instead of one message per element with RDMA.

The `ConditionalGather` RDMO would be useful in OLTP workloads when reading versioned tuples, as shown in Figure 6, where it compares timestamps for visibility checking and only returns visible versions in one operation. In an OLAP workload that involves a join, this RDMO retrieves all tuples in a hash bucket that match a key in one round-trip. Short-circuiting the conditional operation performs projection in the network.

3. SignaledRead. Many data structures use locks to serialize concurrent operations. Exposing lock-based data structures over RDMA, however, requires at least three RDMA requests: two operations target the lock and one performs the intended operation. This RDMO saves at least two messages by "eliding" these lock operations, akin to speculative lock elision in hardware [14]. As shown in Figure 7, the `SignaledRead` RDMO attempts a compare-and-swap operation. If the swap fails, the RDMO retries the compare-and-swap a few times and returns the value of the last read. Else, the RDMO reads the requested data and resets the flag.

4. WriteAndSeal. This RDMO first writes data to a buffer, then writes to the seal location to mark the completion of the write. This would require two messages in an RDMA implementation. This RDMO will be used in lock-based synchronization to update data and release the lock in one operation.

5. ScatterAndAccumulate. This RDMO performs a scatter operation that involves indirect addressing to the destination through a lookup table, as shown in Figure 8. Instead of overwriting the data at the destination, this RDMO accumulates the transmitted values to what is already present in the destination address. `ScatterAndAccumulate` reduces the substantial network cost of hash-based parallel aggregation for high-cardinality domains [11].

4. CONCLUSIONS

The RDMA communication primitives offered by fast networks are too low-level and verbose for common data processing operations. One way to overcome the complexity of RDMA is to use MPI. Our analysis shows that MPI lacks many desirable attributes. This paper introduces the Remote Direct Memory Operation (RDMO) interface which permits a sequence of reads, writes and atomic operations on remote memory to be executed in one round-trip. Performing five common database operations as RDMOs cuts down the number of network round-trips by as much as one order of magnitude.

5. REFERENCES

[1] G. Alonso, C. Binnig, et al. DPI: the data processing interface for modern networks. In *CIDR*, 2019.
[2] C. Barthels et al. Rack-Scale In-Memory Join Processing Using RDMA. In *SIGMOD*, 2015.
[3] C. Barthels et al. Distributed Join Algorithms on Thousands of Cores. *PVLDB*, 2017.
[4] R. Belli and T. Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *IPDPS*, 2015.
[5] C. Binnig et al. The End of Slow Networks: It's Time for a Redesign. *PVLDB*, 2016.
[6] A. Costea et al. VectorH: Taking SQL-on-Hadoop to the Next Level. In *SIGMOD*, 2016.
[7] Global Arrays. `http://hpc.pnl.gov/globalarrays`.
[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[9] A. Kalia et al. Using RDMA Efficiently for Key-value Services. SIGCOMM, 2014.
[10] F. Liu et al. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*, 2017.
[11] F. Liu et al. Chasing similarity: Distribution-aware aggregation scheduling. *PVLDB*, 12(3):292–306, 2018.
[12] MVAPICH. `http://mvapich.cse.ohio-state.edu/`.
[13] OpenMPI. `https://www.open-mpi.org/`.
[14] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, 2001.
[15] W. Rödiger et al. High-speed Query Processing over High-speed Networks. *PVLDB*, 2015.
[16] K. Umamageswaran et al. Exadata Deep Dive: Architecture and Internals. Oracle OpenWorld, 2017.
[17] E. Zamanian et al. The End of a Myth: Distributed Transaction Can Scale. *PVLDB*, 2017.