SAND2018-5929C

# Improving MPI Multi-threaded RMA Communication Performance

Nathan Hjelm*†
Los Alamos National Laboratory
hjelmn@lanl.gov

Matthew G. F. Dosanjh‡
Ryan E. Grant
Sandia National Laboratories
Center for Computational Research
mdosanj,regrant@sandia.gov

Taylor Groves§
Lawrence
Berkeley National Laboratory
tgroves@lbl.gov

Patrick Bridges
University of New Mexico
bridges@cs.unm.edu

Dorian Arnold
Emory University
dorian.arnold@emory.edu

## ABSTRACT

One-sided communication is crucial to enabling communication concurrency. As core counts have increased, particularly with many-core architectures, one-sided (RMA) communication has been proposed to address the ever increasing contention at the network interface. The difficulty in using one-sided (RMA) communication with MPI is that the performance of MPI implementations using RMA with multiple concurrent threads is not well understood. Past studies have been done using MPI RMA in combination with multi-threading (RMA-MT) but they have been performed on older MPI implementations lacking RMA-MT optimizations. In addition prior work has only been done at smaller scale (<=512 cores).

In this paper, we describe a new RMA implementation for Open MPI. The implementation targets scalability and multi-threaded performance. We describe the design and implementation of our RMA improvements and offer an evaluation that demonstrates scaling to 524,288 cores, the full size of a leading supercomputer installation. In contrast, the previous implementation failed to scale past approximately 4,096 cores. To evaluate this approach, we then compare against a vendor optimized MPI RMA-MT implementation with microbenchmarks, a mini-application, and a full astrophysics code at large scale on a many-core architecture. This is the first time that an evaluation at large scale on many-core architectures has been done for MPI RMA-MT (524,288 cores) and the first large scale application performance comparison between two different RMA-MT optimized MPI implementations. The results show a 8.6% benefit to our optimized open source MPI for a full application code running on 512K cores.

**ACM Reference format:**
Nathan Hjelm, Matthew G. F. Dosanjh, Ryan E. Grant, Taylor Groves, Patrick Bridges, and Dorian Arnold. 2018. Improving MPI Multi-threaded RMA Communication Performance. In *Proceedings of 47th International Conference on Parallel Processing, Eugene, OR, USA, August 13–16, 2018 (ICPP 2018),* 10 pages.
https://doi.org/10.1145/3225058.3225114

*Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA. Los Alamos National Laboratory is operated by Los Alamos National Security, LLC for the NNSA. LA-UR-16-xxxxxx.
†Also with Computer Science Department, University of New Mexico.
‡Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.
§Work primarily done as an employee of Sandia National Laboratories.

## 1 INTRODUCTION

Most traditional high performance computing (HPC) applications leverage process-level parallelism via message passing, for example using Message Passing Interface (MPI). However, proposed exascale systems (those capable of $10^{18}$ floating point operations per second) will have to leverage levels of parallelism several orders of magnitude beyond that of current systems. The expected increase in intra-node and inter-node parallelism motivates two seemingly orthogonal techniques: (1) hybrid programming models that integrate multi-threading and (2) alternative communication methods such as one-sided communication (or *Remote Memory Access* (RMA)). Multi-threading provides intra-node parallelism in a single memory address space, while one-sided communication provides opportunities for optimizing communication by decoupling communication and synchronization.

One-sided or RMA communication has been supported by the MPI standard for many years, beginning in MPI 2.0 [16] and enhanced in MPI 3 [23]. Past inefficiencies in MPI RMA implementations, as well as challenges in exploiting communication/computation overlap have held back RMA adoption rates, but with the push towards increased thread parallelism, the benefits of RMA are beginning to be recognized. The major benefits of RMA are its ability to be easily offloaded to existing network hardware for data movement and its lack of message matching requirements. This means that it can more efficiently exploit today's network hardware without the need for strong MPI progress engines running on compute cores.

Traditionally, multi-threaded communication in MPI is limited by serial data structures needed to support the message matching requirements inherent to two-sided MPI. A lack of ordering requirements outside of completion/barrier semantics with RMA means that many threads can interact with MPI with fewer serialization points and rarely conflict with each other. This makes MPI RMA an excellent candidate for multi-threaded environments and a compelling area of study. We refer the combination of remote memory access and multi-threading in MPI as RMA-MT.

A limiting factor to the adoption of MPI RMA is that the performance when using multiple threads (called RMA-MT) is not well studied and no comparisons have been done between any of the MPI implementations that have been recently optimized RMA-MT communication. Prior work in the area of RMA-MT performance evaluation was performed on unoptimized RMA-MT

MPI libraries [9]. In addition, this work only focuses on micro-benchmarks and mini-applications as, at the time, there were no applications available that used RMA-MT. Prior work was also limited to relatively small scale (<=512 processes) and traditional CPU architectures. While prior work was a motivating factor in the improvement of MPI implementations' RMA-MT operation, there is no work demonstrating the current progress that has been made. Such work would be useful for developers to determine if MPI RMA-MT performance is sufficient to meet their application's needs. The performance of RMA-MT is also not understood for many-core architectures and, as these are a prime motivating factor in moving to RMA-MT, given their large core counts, such information is essential to the applications and MPI communities.

In this paper, we describe the design, implementation, and evaluate the performance of a high-performance, scalable, thread-safe RMA component in Open MPI. This component works for any underlying network fabric that supports native put/get *Remote Direct Memory Access* (RDMA) operations and basic remote *Atomic Memory Operation* (AMO) primitives. By providing a native RMA (RDMA-based) communication component for Open MPI that supports thread-level parallelism, the module provides significant performance improvements over previous versions of Open MPI. While we focus on a specific MPI implementation and hardware platform, our design and implementation is generalizable to any MPI implementation or hardware platform, from small clusters to other leadership class supercomputers. For several networks, our solution is the only open-source optimized RMA-MT MPI implementation available (e.g. Cray Aries). Open source highly optimized libraries are essential for rapid troubleshooting and bug fixing versus closed-source implementations as the users themselves can solve issues as they arise and diagnose observed behaviors without requiring assistance from an outside party (that controls the source code).

Our study uses Trinity, a Cray XC supercomputer located at Los Alamos National Lab (LANL). Trinity is a hybrid system with an equal number of nodes utilizing Intel Haswell and Intel Phi *Knights Landing* (KNL) processors. This study investigates the performance on both the Haswell and Xeon Phi partitions independently. A performance comparison is run with Open MPI against Cray-MPI at both small scale with micro-benchmarks and at large scale with mini-applications and WOMBAT [21].

This paper makes several contributions:

(1) A complete, optimized open-source implementation of multi-threaded RMA in Open MPI
(2) An evaluation of multi-threaded MPI RMA performance at scale on one of the top 10 supercomputers in the world.
(3) The first comparison of two RMA-MT optimized MPI implementations at scale (closed-source Cray-MPI and open-source Open MPI).
(4) The first study of RMA-MT performance on a many-core architecture, i.e. KNL.

The rest of this paper is organized as follows: after offering a background on the concepts and MPI functions relevant to our work (Section 2), we describe the design and implementation of the enhanced RMA support in Open MPI as well as improvements that provide a scalable, performant and thread-safe RMA component in (Section 3). Then we present our experimental framework (Section 4) followed by our experimental results and an analysis thereof (Section 5). Finally, we discuss related works (Sections 6) and our conclusions and future work (Section 7).

# 2 BACKGROUND

In this section we provide background necessary for understanding the performance of MPI RMA.

## 2.1 MPI Remote Memory Access

The RMA interface was introduced in version 2.0 [16] of the MPI specification to expose network hardware features for one-sided communication. To use RMA in MPI, process groups, encapsulated by an MPI communicator, are associated with MPI windows, regions of memory that may be read from or written to by a remote nodes. In other words, RMA operations are performed within the context of the MPI window to transfer data between the local MPI process (*origin*) and a remote MPI process (*target*). The RMA interface provides support for reading (`MPI_Get()`), writing (`MPI_Put()`), and atomic memory operations (`MPI_Accumulate()`) on data exposed in an MPI window.

In MPI, all RMA communication must occur while the initiator has an *access epoch*, a time period for which a memory region is guaranteed to be in a consistent state, to the target. Access epochs are opened and closed using synchronization functions. For the requisite synchronization between target and origin processes, MPI-2 RMA provides three methods; *Post-Start-Complete-Wait* (PSCW), fence, and lock. The PSCW and fence methods provide 'active target' synchronization [12], where the target process must participate in the synchronization. For example, fence necessitates a collective operation that involves all the processes in the process group associated with the MPI window to open or close an access epoch. PSCW opens an access epoch on a subset of the processes participating in the MPI window. Lock/unlock provides passive target synchronization that does not require any involvement from the target process; this method manages access epochs on a single *target* at a time.

MPI-3 RMA introduced global shared locks (lock-all) as well as various forms of flush. The global shared lock provided by lock-all allows multiple processes associated with a memory window simultaneous access. This allows a lock-all operation to potentially replace multiple MPI-2 lock operations. Flush synchronization can be called during any passive-target access epoch and ensures either local (flush-local, flush-local-all) or both local and remote (flush, flush-all) completion of all preceding RMA operations to the target with the benefit of not dropping the lock. This provides a way to synchronize without the overhead associated with re-obtaining a lock.

Note that, while the RMA in MPI provides a one-sided communication interface, the MPI standard does not require that the underlying implementation to be one-sided. The implementation simply must provide some level of progress for passive target operations without requiring the user to call into the MPI library to be standard compliant. In fact, the initial MPI-3 RMA implementation in Open MPI used two sided communication (e.g. `MPI_Send()` and `MPI_Irecv()`) to provide an MPI-3 compliant RMA interface. While this approach allows for compliance with the specification, it defeats the purpose of RMA communication by imposing many of the overheads associated with two-sided communication.

## 2.2 MPI Threading Modes

The number of processing elements per node for HPC systems has increased dramatically. For example, platforms based on Intel Xeon-Phi processors have approximately 4X more cores per node than their counterparts based on older Intel Haswell processors. Accordingly, HPC communication libraries are being adapted to accommodate the increased levels of intra-node parallelism; these adaptations

include considerations for thread-safe library implementations. MPI provides only one thread mode (MPI_THREAD_MULTIPLE) that guarantees thread safety.Additionally, existing MPI thread-safety features are not performant due to the complex (and high overhead) synchronization requirements of serial data structures, such as the matching engine. The matching engine in MPI must enforce the message ordering requirements in the MPI Specification [23], which state that two-sided communication must be matched in the order in which it was posted. This means that the matching list must be operated on in a serial fashion in order to prevent overtaking on the list and correct matching behavior in general. In addition, there are two matching lists, a priority list for expected messages and an unexpected list, such that when a new receive operation is posted both lists must be locked while the unexpected list is searched in case the receive request needs to be appended to the priority list.

The combination of RMA and multi-threading, RMA-MT, is a compelling pairing as it can exploit hardware device capabilities to provide higher message rates and greater efficiency by using multiple hardware device contexts without the need for locking or serialization. This is because RMA communications do not require strict data ordering; the synchronization of windows is the only requirement, but it does not state what order data needs to be placed into a window.

RMA-MT has recently attracted enough attention that applications are beginning to leverage the approach. WOMBAT [21], an astrophysics code, uses RMA as its main communication mechanism and relies on multi-threading through OpenMP.

### 2.3 Cray XC-40

The Cray XC-40 uses a high-performance RDMA capable network known as Aries. The Aries network is deployed on a number of Top500 systems including Trinity at Los Alamos and Cori at Lawrence Berkeley National Laboratories. The features of this network can be accessed with two Cray libraries called ugni[6] and DMAPP[6][5]. The ugni library is targeted at MPI and similar programming models and provides support for send/recv as well as access to many of the low-level RDMA and atomic memory features of the network. Access to the network is performed through device contexts. Multi-threaded access to these device contexts is not allowed and external serialization is required to protect access. The DMAPP library is targeted at one-sided programming models like *Partitioned Global Address Space* (PGAS) and SHMEM. This library provides support for additional Aries features and is thread-safe (allows concurrent access from multiple threads). Unlike ugni the DMAPP library is deeply integrated with Cray's runtime environment. This makes it difficult to make use of DMAPP when using alternative run-times such as the Open MPI *Open Run-Time Environment* (ORTE).

Both network libraries provide access to two modes of operations; *Fast Memory Access* (FMA), and *Block Transfer Engine* (BTE). The FMA mode gives low-latency access to remote memory and can be used without registering the origin buffer when using RDMA put. The BTE mode gives high-bandwidth access but requires that both the origin and target buffers be registered with the NIC.

## 3 IMPROVING MULTI-THREADED OPEN MPI

To overcome the inherent limitations of Open MPI's original *One-Sided Communication* (OSC) implementation (*pt2pt*, which leveraged a two-sided back-end), we designed and implemented an enhanced OSC component (*rdma*) for Open MPI v2.0.0. This new component was designed to support multi-threaded applications in a scalable manner and to support any network that provides RDMA, the fetch-and-add, and compare-and-swap network AMOs. We now describe the key optimizations and tradeoffs we made in the design and implementation of the *rdma* component to ensure high performance and scalability.

### 3.1 Memory Scaling

Generally, RDMA networks require memory regions that will be the target of RDMA or AMOs operations to be registered with the network stack. The registration process usually produces an access handle in the range of 8-16 bytes. The registration handle and the target pointer are then used to initiate a network RDMA operation. A naive RMA implementation might perform an all-gather operation and store the registration handles and base pointers for every process participating in the MPI window, requiring $O(n)$ memory where $n$ is the number of processes. Such a strategy is similar to pre-connecting all MPI processes for point-to-point.

In our *rdma* component, we avoided this potentially prohibitive memory overhead by distributing the registration handles and pointers across all participating nodes and storing the array in shared memory. The registration data and pointers associated with the processes holding the distributed array are gathered and stored in a shared memory region on each node. The first time an MPI process attempts to open an access epoch on a remote MPI process it reads the registration and pointer data on-demand from the distributed array and caches it locally. This scheme reduces the up-front memory requirements to $O(N)$ where $N$ is the number of participating nodes. This approach will prove to be very useful particularly for future systems with high degrees of intra-node parallelism.

### 3.2 Lock Scaling Improvements for Lock-all Synchronization

Lock-all is an important synchronization method that facilitates passive data transfer with an epoch spanning multiple processes. We have implemented two different performance-aware lock-all schemes into Open MPI v2: an on-demand locking strategy and a two-level locking strategy. The best strategy is application-dependent, so an optimized implementation must support both well.

*On-demand locking.* The on-demand locking support in the *rdma* component is similar to that of the *pt2pt* component. When executing a lock-all synchronization, the current access epoch is marked as lock-all and a shared lock on the local process is obtained. The implementation will not attempt to acquire any other lock. Before executing the first RMA operation on a target process (during the access epoch), the origin process acquires the target's remote lock using a network fetch-and-add and checks for an existing exclusive lock. If an exclusive lock was already set, the lock is decremented and the lock operation is re-attempted until the lock is obtained. When the lock-all epoch is complete the implementation releases each lock obtained during the access epoch. This lock-all implementation requires $O(m)$ network atomic memory operations to obtain and release the global lock where $m$ is the number of processes that are the target of an RMA operation during the lock-all access epoch. On-demand locking is optimal for application that either do not use lock-all or make extensive use of exclusive locks.

*Two-level locking.* Two-level locking is a lock strategy that enables the support of a lock-all operation without needing to perform a lock operation per-target. The two-level lock implementation in Open

MPI is similar to the design described in [10]. Two-level locking is used for both lock-all and exclusive locks. The first level is a 64-bit global counter held by the lowest rank process in the communicator used to create the RMA window. This counter is divided into two 32-bit counters; an exclusive lock counter and a global shared lock counter. When obtaining either an exclusive lock or a global shared lock, the corresponding counter of the global lock is incremented using a 64-bit network atomic fetch-and-add. The result is checked for a competing lock. If no competing lock is found, either the lock was obtained (lock-all) or an attempt is made to acquire a local lock at the target process using an atomic compare-and-swap (exclusive lock). If a competing lock is found, the global counter is decremented and the operation is retried until the lock is obtained. Single target shared locks only modify the local lock at the target using atomic fetch-and-add and do not modify the global counters.

One of the benefits of the two-level lock design is that when executing a lock-all under low contention it only requires $O(1)$ network atomic operations instead of $O(m)$ needed by the on-demand locking implementation. The trade-off is that this locking implementation introduces a network bottle-neck at the node that holds the global lock when there is heavy usage of exclusive locks. The implementation also imposes the overhead of an additional network atomic operation when obtaining an exclusive lock that is not present in the on-demand locking implementation.

## 3.3 Byte Transport Improvements

The *Byte Transport Layer* (BTL) interface [11] in Open MPI supports data movement between communication endpoints for many types of interconnects. We identified several multi-threading bottlenecks and limitations in the BTL interface that were largely a product of the original design which was meant to support high-bandwidth two-sided operations using network RDMA. This RDMA support was necessary to provide access to the full bandwidth available on the network by eliminating extra copies when delivering a message to a target buffer.

BTL operations (put, get, send) were implemented using information stored in an internal structure known as a fragment. For RDMA operations, fragments were allocated using a BTL module's `prepare_src()` and `prepare_dst()` functions. The `prepare_src()` function additionally supported the send functionality in the BTL module. These functions registered the local memory with the network and prepared the fragments for the BTL `put()` and `get()` functions. To support single send-receive operations, a BTL fragment could only be used for a single RDMA operation. This limitation forced *rdma* to allocate and release a fragment for each RMA operation, thereby introducing additional serialization points and function call overheads into the RMA critical path. We fixed this bottleneck by removing the `prepare_dst()` function and removing the overloaded usage of the `prepare_src()`. The parameters originally passed to these functions are now passed directly to the BTL module's `put()` and `get()` functions.

The original BTL interface did not provide functions to perform atomic network operations supported by many high-performance network APIs. To support these operations, we added additional functions to the BTL interface to support compare-and-swap, and both fetching and non-fetching atomic operations. The new interface supports 32 and 64-bit integer and floating point operations.

*3.3.1 uGNI BTL Improvements.* Additional improvements were made to further optimize the RMA-MT performance of Open MPI on Intel KNL with Aries and Gemini networks. In prior versions of Open MPI access to the ugni library was done using locks and a single network device context. The locking is required due to the single-threaded nature of the ugni library. To reduce the thread contention on the network resources and provide better support for many-core architectures we updated Open MPI with support for multiple simultaneous ugni device contexts. This is similar to an optimization in use in Cray-MPI [21]. These device contexts are assigned round robin to threads for RMA operations (put, get, atomics) only. Since there is no benefit to using multiple contexts with single-threaded operation this new support is only enabled when when the MPI_THREAD_MULTIPLE threading model is requested. By default Open MPI currently bases the number of device contexts on the number of local ranks to reduce contention of the underlying kernel and hardware resources. The number of device contexts can be controlled by setting the *btl_ugni_virtual_device_count Modular Component Architecture* (MCA) variable via an environment variable or on the mpirun command line. In general, to optimize network utilization, a user should attempt to match the virtual device count with the number of threads that are expected to make concurrent calls into the RMA interface.

# 4 EXPERIMENTAL SETUP

We designed our experiments to help us answer the following questions:

- Is the enhanced *rdma* RMA implementation better than the *pt2pt* version?
- How does *rdma* in Open MPI compare to a highly optimized vendor MPI?
- How does the *rdma* scale with multi-threaded applications?

To answer these questions we used Trinity, a large system located at LANL and run by Alliance for Computing at Extreme Scale (ACES). As of November 2017 the Trinity system ranked in the top 10 of the Top500 [22]. This system was chosen as it supported the necessary RDMA and AMO features and has a highly tuned vendor MPI for performance comparisons. This system is made up of 9408 nodes each with two 16 core Intel E5-2698v3 processors and 9848 nodes with a single 68 core Intel KNL processor. All nodes have 128 GB of DRAM. The interconnect is a Cray Aries network which is arranged in a dragonfly topology. To evaluate at small scale with the micro-benchmarks, we used a small test bed cluster, Trinitite, which provides the same hardware and software environment. Both systems were running Cray CLE 6.0 UP01 and all benchmarks were compiled with gcc 5.3.0. We had dedicated time on Trinity during two independent open-science periods and were able to run on 8,192 Haswell nodes with 262,144 cores and 8,192 KNL nodes with 557,056 cores.

We used a version of Open MPI pulled from the master branch of the Open MPI git repo hash *f858647*. This version contained most the improvements described in Section 3. This version did not include multi-device context support and is equivalent to setting the *btl_ugni_virtual_device_count* MCA variable to 1.

For performance evaluation, we used the Latency and Bandwidth benchmarks from the RMA-MT benchmark suite [9], which was designed to test and analyze the RMA interfaces of MPI under a Thread Multiple invocation. For scaling studies, we used a Mantevo miniapp [26] (HPCCG). HPCCG addresses implicit unstructured partial differential equations. HPCCG targets the sparse iterative solver and creates a 27-point finite difference matrix, for which each MPI rank is designated a user-defined sub-block. The miniapp

**Table 1: *pt2pt* vs. *rdma* (Put and Get) bandwidth and latency with relative performance between *pt2pt* and *rdma*.**

| | Small Messages ≤ 16KiB | | | Large Messages > 16KiB | | |
|---|---|---|---|---|---|---|
| | pt2pt | RDMA | % | pt2pt | RDMA | % |
| **Bandwidth MiB/s (more is better)** | | | | | | |
| **get** | 205 | 395 | +92% | 4846 | 6815 | +40% |
| **put** | 209 | 414 | +98% | 4202 | 6471 | +54% |
| **Latency $\mu$s (less is better)** | | | | | | |
| **get** | 82 | 73 | -11% | 159 | 103 | -35% |
| **put** | 64 | 59 | -8% | 136 | 99 | -28% |

was modified to use lock-all synchronization. All benchmarks were run with the Open MPI process affinity set to none to allow the application threads to move between cores.

For each configuration, we ran the RMA-MT micro-benchmarks 10 times with each comprising 100 iterations. For each configuration, we ran HPCCG four times.

For the comparison study with Cray-MPI we used the Intel KNL partition of the Trinity supercomputer. At the time this partition was comprised of 8909 compute nodes. The system was running Cray CLE 6.0 UP03. For these tests we used a version of Open MPI pulled from the master branch of the Open MPI git repo hash *6886c12*. Open MPI was compiled with gcc 6.3.0 and Cray Fortan for the language-specific bindings. The *btl_ugni_virtual_device_count* MCA variable was set to auto-set the number of device contexts.

For the micro-benchmarks and mini-apps we used the same programming environment used with Open MPI but for WOMBAT we used the Cray programming environment. WOMBAT was built with Cray Fortan in all cases and was configured to use passive-target RMA. To evaluate the performance of Cray-MPI with DMAPP, all executables were linked against the Cray DMAPP library and with MPICH_RMA_OVER_DMAPP=1, and MPICH_MAX_THREAD_SAFETY=multiple set.

# 5 EXPERIMENTAL RESULTS

## 5.1 Evaluation of OSC RDMA

*5.1.1 OSC point-to-point vs. RDMA.* In this section, we present our performance evaluation based on the latency and bandwidth benchmarks. Our micro-benchmark-based evaluation included a comparison of our Open MPI improvements versus the original code using different synchronization methods, thread counts, message size, and RMA operations. Each benchmark was run 10 times, with 100 put/get operations per run. This combination of features resulted in approximately 45,000 latency and bandwidth measurements. We summarize key results (as opposed to results for each evaluated message size) due to data volume. In particular, we have divided message sizes into two classes: small and large (less than or equal to 16KiB; greater than 16KiB, respectively[1]). We also split the results by (1) put or get, (2) synchronization method, and (3) thread count, in Tables 1-3 respectively. For each split in the data we report the averages with respect to the attribute being split on. For example, Table 1 shows small-message get operations averaged 205 MiB/s, across all synchronization modes and thread counts for *pt2pt*.

*Put and get.* Table 1 shows that (1) *rdma* outperforms *pt2pt*; (2) for small messages, the best latency is achieved with put operations; (3)

[1] 16KiB being a common transition point between eager and rendezvous protocols in MPI implementations

**Table 2: *pt2pt* vs. *rdma* (Thread Count) bandwidth and latency with relative performance between *pt2pt* and *rdma*.**

| | Small Messages ≤ 16KiB | | | Large Messages > 16KiB | | |
|---|---|---|---|---|---|---|
| | pt2pt | RDMA | % | pt2pt | RDMA | % |
| **Bandwidth MiB/s (more is better)** | | | | | | |
| **1 thread** | 361 | 1062 | +194% | 7261 | 8780 | +21% |
| **2 thread** | 248 | 590 | +138% | 6493 | 8413 | +30% |
| **4 thread** | 222 | 293 | +32% | 5120 | 7833 | +53% |
| **8 thread** | 166 | 117 | -29% | 3942 | 6608 | +68% |
| **16 thread** | 108 | 63 | -42% | 2749 | 5031 | +83% |
| **32 thread** | 61 | 27 | -56% | 1582 | 3198 | +102% |
| **Latency $\mu$s (less is better)** | | | | | | |
| **1 thread** | 26 | 19 | -24% | 62 | 50 | -19% |
| **2 thread** | 28 | 23 | -18% | 64 | 49 | -23% |
| **4 thread** | 32 | 33 | +3% | 78 | 53 | -32% |
| **8 thread** | 49 | 51 | +4% | 100 | 65 | -35% |
| **16 thread** | 89 | 90 | +1% | 165 | 110 | -33% |
| **32 thread** | 268 | 232 | -13% | 419 | 280 | -33% |

**Table 3: *pt2pt* vs. *rdma* (Synchronization methods) bandwidth and latency with relative performance between *pt2pt* and *rdma*.**

| | Small Messages ≤ 16KiB | | | Large Messages > 16KiB | | |
|---|---|---|---|---|---|---|
| | pt2pt | RDMA | % | pt2pt | RDMA | % |
| **Bandwidth MiB/s (more is better)** | | | | | | |
| **fence** | 216 | 388 | +79% | 3755 | 6638 | +77% |
| **lock** | 205 | 425 | +107% | 4636 | 6633 | +43% |
| **lock-all** | 198 | 391 | +97% | 4740 | 6679 | +41% |
| **PSCW** | 216 | 388 | +79% | 3755 | 6638 | +77% |
| **Latency $\mu$s (less is better)** | | | | | | |
| **fence** | 66 | 76 | +15% | 147 | 112 | -24% |
| **lock** | 89 | 67 | -24% | 157 | 100 | -36% |
| **lock-all** | 80 | 66 | -18% | 150 | 98 | -35% |
| **PSCW** | 56 | 56 | 0% | 138 | 94 | -32% |

for large messages the best bandwidth is provided by get operations. Put and Get performance are largely dependent on underlying network architecture design, and therefore, the results observed reflect the network performance of the *Aries* network as used by the Open MPI *rdma* component. Specific to the *Aries* network [1], a 64B read (or get) operation requires 3 request flits and 12 response flits, whereas a put operation results in 14 request flits and 1 response flit. Furthermore, it is expected that get latencies are higher than put latencies because put operations require a PCI-e read on the remote node to complete.

*Thread count.* On examination of Table 2, there are a few occasions where *pt2pt* outperforms *rdma* in bandwidth (small messages, 8, 16, and 32 thread). At these thread counts, the message is divided into smaller and smaller segments across the threads. *rdma* provides lower bandwidth in this case due to a feature of *pt2pt* that aggregates multiple small puts into a single MPI send. *rdma* has a similar feature for small puts to consecutive memory locations but it is not currently supported when using MPI_THREAD_MULTIPLE. This means each `MPI_Put()` operation translates to a separate network transaction.

This highlights a scenario where a developer might want to leverage a two-sided point-to-point implementation of RMA operations. A caveat of this observation is that to realize the higher potential small message put bandwidth of *pt2pt* either the BTL component needs to support asynchronous progress or the target MPI process needs to enter the MPI library to progress the RMA

operations. In the case of the RMA-MT benchmark the target process is spinning in a call to `MPI_Barrier()`. During our study we found that for message sizes smaller than 2MiB, *pt2pt* provides greater bandwidth than *rdma*. This is not surprising since RDMA is designed to facilitate efficient transfer of large messages, due to the inherent costs of registering memory. However, for messages in this range, the absolute difference in bandwidth is just tens of megabytes per second. Our improvements to one-sided communication reflect the idea that small messages should be optimized for latency and large messages should be optimized for bandwidth.

*Synchronization method.* Table 3 provides an overview of the performance of different synchronization methods in MPI-3. While this table shows some differences between performance of different synchronization techniques, it does not fully illustrate the benefits of passive data transfer that lock-all synchronization provides. Lock-all is most beneficial when an application can take advantage of the passive data transfer to overlap computation and communication.

Tables 1-3 show that our enhancements to *rdma* result in widespread improvement in both bandwidth and latency over *pt2pt*. In particular, *rdma* outperforms *pt2pt* 89 out of 96 times. At its best, *rdma* achieves an almost 3X increase to bandwidth (1 thread, small messages) and a 35% decrease to latency (lock-all, large messages).

*5.1.2 Discussion.* We offer two explanations of *rdma*'s performance benefit observed in this section:

- Reduced network usage from implicit synchronization
- Improved algorithms for explicit synchronization

Our *rdma* component provides reduced implicit synchronization costs because it doesn't incur the same implicit synchronization of two sided communication. There is less transfer facilitation data that needs to be sent. For instance, the call already has a destination in memory from the time the put or get is issued, thus the rendezvous protocol is not needed for large message. The improvements here are likely dependent on network congestion and topology. This may explain some of the degradation of the miniapp performance as the scale increases and the network becomes a bottleneck.

The effect of the improved explicit synchronization can be seen in the miniapp results. As we increase the number of processes participating in the explicit synchronization call, the calls get more expensive. We can attribute a part of the scale based performance degradation in the *pt2pt* miniapp results to an unoptimized implementation of `MPI_Win_Lock_All()` that sends a lock request to each peer process in the MPI window.

*Addressing the questions of Section 4.* Going back to the questions we put forth at the start of Section 4, our results support several conclusions. First, the *rdma* component performed correctly. Within the RMA-MT benchmarks are checks to verify data is transferred across windows as expected. While some of the early benchmarking of RMA-MT [9] found occasional errors within these windows, we did not observe any such errors in the *rdma* component. Additionally, our mini-applications did not observe any change in the residual, compared with the *pt2pt* component. Our *rdma* component saw significant improvements at scale compared to the *pt2pt* component when running multi-threaded MPI code. Specifically, our *rdma* was able to scale up to the full system size of 262,144 cores for HPCCG. Making it the first implementation of RMA-MT to be validated at such scales. For these weak-scaled problems, *rdma* did not show any signs of slowdown until reaching 65,536 cores. In contrast, the *pt2pt* component began to experience significant scaling problems beginning at 1,024 cores, failing to scale past 4,096 cores. The *rdma* component has improved the scalability of Open MPI RMA-MT by a factor of 64X. The impact of the *rdma* component on application performance varies according to scale. For small scale applications this impact is limited to minor improvements, but as the application increases in core count the benefits of *rdma* increase. At scales of 4,096 cores, the *rdma* component cuts the application runtime in half compared to the native *pt2pt* component.

*General applicability.* Traditionally applications have been written for single-threaded two-sided communication. Modern HPC networks support one-sided communication (e.g. Cray Aries, Omni-Path, InfiniBand and RoCE) and if we want to get the most out of these networks, applications need to adopt asynchronous, multi-threaded communication. While this work focuses on the combination of multi-threaded and RMA, many of our improvements would benefit applications using single-threaded RMA and multi-threaded independently. Our work is generally applicable to MPI applications in the HPC space, since any two-sided application could be adapted to utilize one-sided operations. As HPC applications explore alternative communication paradigms some high-level knowledge of the MPI RMA implementation may help drive algorithm design. Ideally, the improvements to RMA-MT performance made in this paper will help drive adoption of more scalable communication methods.

## 5.2 Performance Comparison to Cray-MPI

*5.2.1 Micro-benchmark Results.* In order to compare the two implementations in terms of basic MPI performance we used a modified version of the RMA-MT benchmark suite [9]. The new version includes support for flush synchronization and a number of improvements to reduce overhead within the benchmark. These benchmarks provide basic latency and bandwidth tests in MPI's multi-threaded mode. Our evaluations sweep the following attributes: message size, put and get, and thread count (1, 2, 4, 8, 16, 32, 64). Each test reports the mean of 1000 iterations of bandwidth or latency and we run 10 tests per data point. In all cases the reported bandwidth is the cumulative bandwidth of all threads for a particular process. Similarly, reported latency is the latency of the slowest thread for a particular process. This creates roughly 22,000 data points which we post-process to provide a general overview of performance in Tables 4 - 5. In each of these tables we present slices of the data filtered by a particular attribute. Each value in the table corresponds to the median value. Micro-benchmark results were collected on an active production system. For Open MPI we modified the default MCA variable configuration to set the FMA/BTE switch-over point to 16k. This switch-over point was picked as it represents the best multi-threaded performance with Open MPI when using RMA-MT. The aprun and mpirun process launchers were used for Cray-MPI and Open MPI to launch the benchmark with two MPI processes placed across two nodes on the same Aries ASIC. MPI process bindings were disabled using the -cc none and –bind-to none options for aprun and mpirun respectively.

For Example in Table 4, we have binned all results as either put or get operations (represented as sub-rows) and small, medium, and large messages (columns $\leq$ 64B, 128 to 16 KiB and $>$ 16KiB, respectively). As such, Table 4 masks the impact of thread count and synchronization method.

These tables reveal that DMAPP performance is generally better than Open MPI, by a significant amount. This difference is most noticeable for small messages and medium sized messages and at higher thread counts. This is not all that surprising given DMAPP

**Table 4: Cray-MPI with DMAPP and Open MPI with ugni (Put and Get).**

| | ≤ 64B | | 128B to 16KiB | | > 16KiB | |
|---|---|---|---|---|---|---|
| | DMAPP | OMPI | DMAPP | OMPI | DMAPP | OMPI |
| **Bandwidth MiB/s (higher is better)** | | | | | | |
| **get** | 17 | 2 | 964 | 306 | 7414 | 7427 |
| **put** | 22 | 4 | 962 | 646 | 8532 | 8225 |
| **Latency $\mu$s (less is better)** | | | | | | |
| **get** | 37 | 73 | 56 | 84 | 155 | 140 |
| **put** | 32 | 43 | 55 | 56 | 137 | 129 |

**Table 5: DMAPP and OMPI (Thread Count).**

| | ≤ 64B | | 128B to 16KiB | | > 16KiB | |
|---|---|---|---|---|---|---|
| | DMAPP | OMPI | DMAPP | OMPI | DMAPP | OMPI |
| **Bandwidth MiB/s (higher is better)** | | | | | | |
| **1 thread** | 7 | 2 | 442 | 339 | 7557 | 7727 |
| **2 thread** | 13 | 2 | 668 | 451 | 7560 | 7713 |
| **4 thread** | 23 | 2 | 804 | 525 | 7482 | 7679 |
| **8 thread** | 42 | 2 | 722 | 533 | 7220 | 7623 |
| **16 thread** | 25 | 3 | 864 | 528 | 7569 | 7280 |
| **32 thread** | 21 | 2 | 1380 | 468 | 7632 | 7170 |
| **64 threads** | 17 | 2 | 1372 | 410 | 7736 | 5820 |
| **Latency $\mu$s (less is better)** | | | | | | |
| **1 thread** | 18 | 22 | 16 | 25 | 72 | 76 |
| **2 thread** | 20 | 27 | 24 | 34 | 80 | 84 |
| **4 thread** | 22 | 35 | 35 | 50 | 118 | 111 |
| **8 thread** | 28 | 57 | 50 | 70 | 152 | 155 |
| **16 thread** | 41 | 85 | 87 | 111 | 213 | 189 |
| **32 thread** | 65 | 121 | 162 | 180 | 285 | 293 |
| **64 threads** | 125 | 253 | 340 | 355 | 465 | 509 |

includes architecture specific enhancements for the Cray Aries network not available when using the ugni library that Open MPI uses. The exception to this is large messages using less than 8 threads, where Open MPI outperforms Cray-MPI's DMAPP RMA implementation by about approximately 5%. In addition, many of the DMAPP performance benefits come in RMA modes that are not typically used, where we have concentrated on the most popular/useful RMA mode, lock_all.

For small messages the Cray-MPI RMA-MT implementation benefits from the DMAPP API's non-blocking implicit functions. These allow multiple put operations to be combined into a single network transfer. The ugni library does not support this feature. This gives Cray-MPI an advantage when running a bandwidth benchmark.

For large messages sent over the Aries network. The BTE mode requires that the memory region associated with the source or target memory on the initiator (put and get respectively) be registered before it can be used for an RDMA operation. To reduce the costs associated with this registration Open MPI keeps a cache of recently used memory registrations. This cache is currently implemented with a synchronous data structure protected by a mutex. As the thread count increases, this mutex quickly becomes a bottle-neck within Open MPI. Cray-MPI and the DMAPP library also use a registration cache but we do not have any details on how Cray-MPI manages its memory registration cache. We attempted to use a Cray library (udreg) for memory registration in Open MPI but found that it reduced performance.

*Detailed analysis of lock-all.* The performance of passive target synchronization (lock, lock-all, and flush) is especially important to understand. These are the only synchronization methods that allow for a combination of passive operation [2] and shared access amongst multiple MPI processes. Because of its importance we have included detailed figures of the lock-all performance (Figures 1a - 1d). This provides the full performance details not provided in the tables. In these figures, each data point represents the median and the error bars represent the standard deviation. Open MPI results are represented by solid lines while Cray-MPI results are dashed lines.

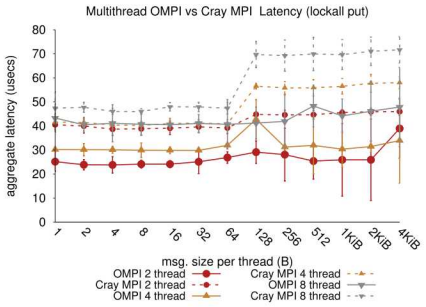
There are several general trends across all the latency results in Figures 1a-1b. Overall, Open MPI latency is better than Cray-MPI for any given thread count, with only small messages (<64B) at large thread counts (32 threads and up) having better performance in Cray-MPI over Open MPI. Thread count has a negative impact on latency, such that increasing thread count correlates to monotonic increases in latency. This trend holds across both put/get and for Open MPI and DMAPP implementations, with 64 threads reaching nearly double the latency of 32 threads. Some of this overhead is due to the inclusion of thread synchronization within the benchmark itself. These barriers are necessary to ensure that all threads have completed their communication before the synchronization calls are made. We do not have measurements on how much of the latency is due to this synchronization and how much is due to the MPI implementations. We can, however, compare the latencies produced by the two MPI implementations. Specific to the Cray-MPI DMAPP implementation, we see that very small messages (1 and 2 bytes) incur a significant penalty to latency for high thread counts. These values then decrease at 4 bytes, creating a temporary dip, which increases substantially once message sizes surpass the 64 bytes. There are additional jumps in latency as message sizes increase to 64KiB. Both of these jumps are likely due to protocol decisions within both Cray-MPI and the DMAPP library. Since both software products are closed-source we can only speculate as to why these transition points exist.

Figures 1c-1d show that thread count seems to impact bandwidth only moderately for small and large sized messages. Threading has a bigger impact on the performance of medium sized messages (128B to 16KiB). Overall Cray-MPI has a slight edge in bandwidth, with Open MPI outperforming with lower thread counts and message sizes in the critical 32KiB-256KiB range. The results show a sweet spot for bandwidth of medium size messages utilizing 8 threads for Open MPI. This contrasts with the performance dip for medium sized messages in DMAPP using 8 threads. Moving beyond 8 threads, the results show that DMAPP performance steadily increases while Open MPI performance decreases. Finally there is a substantial increase to bandwidth in Open MPI once message sizes of 32KiB are achieved. This is, again, due to the choice of the FMA/BTE transition point made for this study. The plots of DMAPP bandwidth do not contain this dramatic increase and reflect a smoother relationship between message size and aggregate bandwidth. This difference is likely due to the usage of registration cache that is better suited for multi-threaded access. Improving registration cache performance is work that is expected to be in a future Open MPI release.

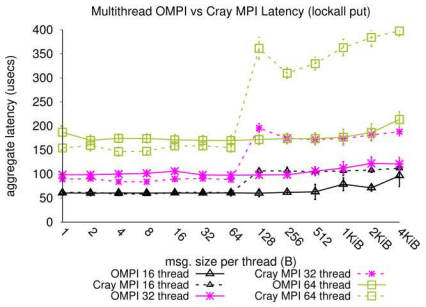
#### 5.2.2 *Mini-applications.*

*HPCCG.* Mini-applications are a proxy for larger full application codes. We examined the performance of HPCCG, a part of the Mantevo suite of mini apps that is meant to be a "best approximation to an unstructured implicit finite element or finite volume application in 800 lines or fewer." [14]. We ran the HPCCG benchmark with the aprun launcher and 64 MPI processes per

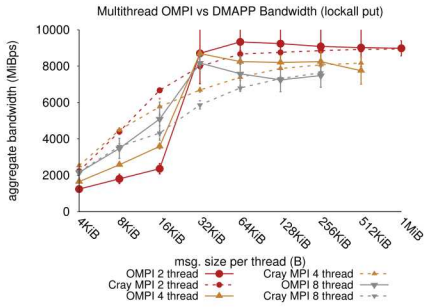
[2] Meaning it does not involve the target node CPU.



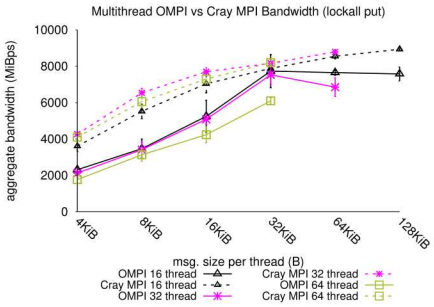

(a) Open MPI vs Cray-MPI latency for small thread counts (put-lockall)



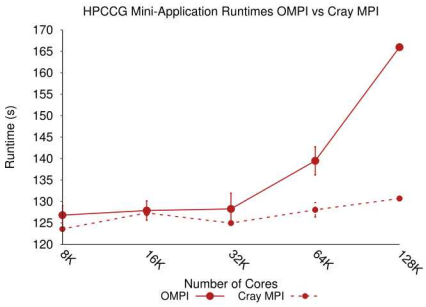

(b) Open MPI vs Cray-MPI latency for large thread counts (put-lockall)



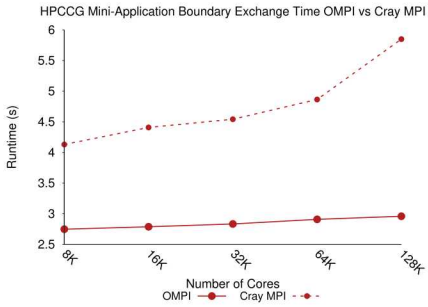

(c) Open MPI vs Cray-MPI bandwidth for small thread counts (put-lockall)




(d) Open MPI vs Cray-MPI bandwidth for large thread counts (put-lockall)




(e) HPCCG runtime comparing both Open MPI and Cray-MPI




(f) HPCCG boundary exchange time comparing both Open MPI and Cray-MPI

**Figure 1: Open MPI vs Cray-MPI put latency, bandwidth and HPCCG Comparisons**

node. MPI processes and their threads were not bound and were allowed to float across all available cores and hyper-threads. This benchmark's RMA-MT conversion is done in a reasonable but naive way. It uses multiple threads for each neighbor communication, spawning a thread for each direction of communication in the neighbor exchange. This has the effect of over subscribing the cores during communication, as each core is only 4-way SMT capable. However, this communication can be easily overlapped as it is RMA, and therefore this over-subscription technique is not a major impediment to performance. Figure 1e shows the total runtime including initialization and 1f shows the total time spent exchanging boundary data with both Open MPI and Cray-MPI of a weak-scaling HPCCG simulation with a local grid size of $120^3$. The total runtime with the two RMA-MT implementations is similar up to 32k cores. At this point there is a divergence and Open MPI is 27% slower at 128k cores. This slowdown is not a problem with the RMA-MT implementation inside Open MPI but it due to a fixed overhead Open MPI incurs when being launched using aprun. This overhead scales with the total number of MPI processes being launched. Though the total runtime for these HPCCG runs are longer with Open MPI the total time spent exchanging data with RMA is lower than with Cray-MPI showing a 49.5% improvement over Cray-MPI at 128K cores. This is a result of Open MPI having optimized the critical lockall RMA operation, showing better results than when considering the entire range of operations, many of which are not widely used. MPI experts encourage all new RMA code to use lockall.

*5.2.3 Full Application.* The number of applications that are capable of using multi-threaded RMA is very limited due to the short time frame in which optimized RMA-MT MPI implementations have been available. One of these applications is WOMBAT [21], an astrophysics simulation code written in Fortran developed by Cray. We ran this benchmark with both Open MPI and Cray-MPI. All WOMBAT runs were performed with the aprun launcher using the depth mapper. This mapper binds the MPI process to a user specified number of cores. We set this depth to the number of threads used by WOMBAT to match the threads exactly to the number of available cores. Due to issues running this benchmark with Open MPI we disabled an internal optimization that accelerates node local RMA communication. We ran a weak scaling problem with a patch size of $48^3$ and 2 patches per thread. For optimal performance we ran with the max in-flight RMA communication parameter set to 4 for the 4 threads/MPI rank runs, 16 for the 8 and 16, and 32 for the 32 and 64. All other WOMBAT parameters were left to their defaults.

Figure 2 shows the results of running 20 time-steps of the weak-scaling problem varying the total number of threads on the x-axis and the runtime for multiple different threads/MPI process. The reported time is the total sum of just the time-steps and does not include any initialization or cleanup time. The results show that WOMBAT scales very well, with Cray-MPI results in Figure 2 represented as dashed lines and Open MPI as solid lines. There is very little increase in runtime when weak-scaling the application up to 512K cores. Both MPI implementations do a good job in maintaining scalability for a very large many-core system. An observable jump in runtime is experienced when moving to 32 or 64 thread with 64 thread variants having slightly more overhead than the 32 thread case. This increase could be due to MPI process being bound across more than one KNL quadrant. In addition, we see some scaling impacts with smaller numbers of threads, most notably the

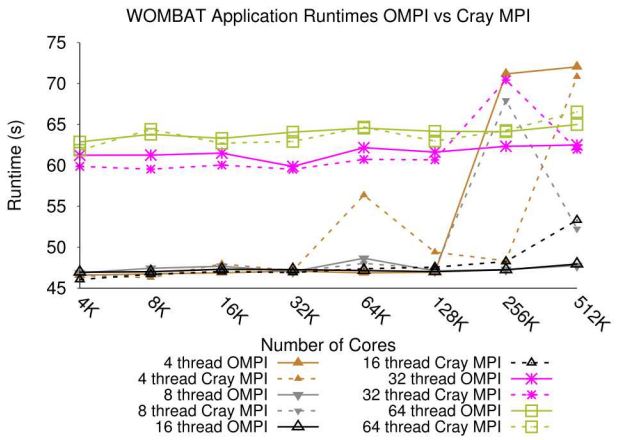



**Figure 2: WOMBAT application runtime comparing both Open MPI and Cray-MPI with varying numbers of threads**

Cray-MPI cases when reaching core counts of 256K and up. Open MPI also experiences some runtime increases which occur for the 4 thread case. However, the 8 and 16 core cases scale very well with a consistent trend, showing best performance at 8 threads at 512K cores with a 8.6% performance advantage over the best Cray-MPI thread configuration. Running at scale shows that either MPI implementation should be run at 8 threads for best performance.

The results of the WOMBAT testing show that the two MPI implementations under test have different performance for this application. This is expected based on the micro-benchmark results that showed differences in performance between the two implementations for basic data movement. However, the results are better than the micro-benchmark results imply. When we examine the bandwidth and latency curves for the micro-benchmarks in Figures 1a- 1d we can observe that the curves for latency are better for Open MPI and for bandwidth, some message sizes are very similar or better in Open MPI. These message sizes are very common for applications, and are used in WOMBAT. As such the performance observed is better for Open MPI. In addition, the performance reported by the micro-benchmarks is the performance under heavy load when multiple threads are calling into the MPI implementation simultaneously. This usage is not reflective of the usage of a real application. The serialization points uncovered by the micro-benchmarks likely have little impact on a real application.

WOMBAT shows the potential of RMA-MT application performance at scale with optimized MPI implementations. This is somewhat expected when we examine the characteristics of the application's interaction with MPI. Firstly, MPI RMA allows for a great deal of parallelism in the communications path due to its lack of per-message data ordering. This means that multiple device contexts can be used in parallel without fear of violating ordering semantics required in MPI. In addition, shared data structures such as two-sided matching lists are not required and such lists are natural serialization points due to message ordering rules. This means that MPI RMA-MT can avoid many of the traditional serialization points in an MPI library and given that the library is optimized for this parallelism, we would expect that this should lead to good scaling behavior in terms of communications. This leads us to the conclusion that at least for Astrophysics Magneto-hydrodynamics simulations, RMA-MT is a viable communication path for highly scalable codes. Realistically, the communication patterns that WOMBAT uses are more generalizable outside of its particular scientific field, and therefore we expect more applications to use RMA-MT in the future.

## 6 RELATED WORK

Independently, efforts to measure performance of RMA [12] and multi-threading in MPI [2, 27] have been addressed. In order to support measuring this performance, several benchmark suites have been developed to measure RMA performance, such as the OSU Benchmark Suite from Ohio State University [24], which supports several different measurements associated with MPI-3 RMA operations, including different window creation and synchronization methods. However, it does not measure operations in the context of multiple threads, similar to other benchmarks like the Intel MPI Benchmark suite [17]. The multi-threaded benchmark suite for Argonne National Laboratory[30] has been used to evaluate multi-threaded MPI performance, but does not address RMA. A test suite that uses both RMA and multi-threading at the same time in MPI, RMA-MT, has been detailed [9] and is publicly available. Other variants that include OpenSHMEM support are also available [31].

Recent work has centered on MPI-3 RMA compliant implementations but has not cross compared multiple different MPI implementations or examined performance for multi-threaded operation [7]. Device level RDMA has been exploited for MPI in the past [20], however such work concentrated on using RDMA as a data transport for traditional two-sided communication, not the MPI RMA interface. Message rate evaluations have been previously performed on traditional and many-core architectures in the past [3]. While RMA and RDMA are promising in a multi-threaded environment, past work [13] suggests that RDMA traffic needs to be handled carefully to avoid interference with memory subsystems, and therefore MPI implementations should be carefully designed to avoid memory contention.

Efforts to harness high-granularity tasking models have been explored through detailed investigation of tasking implementations that utilize MPI for communication [29]. In addition, work has been done to quantify the impact of highly parallel fine-grained tasking with MPI and has found that it may be beneficial [4]. Many threading and tasking libraries exist outside of MPI and are commonly used in conjunction, the most common being MPI+OpenMP [25].

Some alternative support for multiple threads in MPI have been proposed, such as the endpoints proposal for MPI [8, 28] that seeks to offer enhanced network performance for multi-threaded MPI applications. Other alternatives, such as native MPI threads have been implemented in alternative MPI libraries such as FG-MPI [18].

This work is the first to compare two MPI libraries that have been optimized for RMA-MT. Cray's DMAPP interface [5] provides multiple device contexts with multi-threaded one-sided communication in its Aries networks. DMAPP's use in WOMBAT [21] is the first case known to the authors' of a major RMA-MT enabled application. It is expected that more applications will move in this design direction now that optimized MPI libraries are available.

The closest RMA-MT performance evaluation at scale was done using Cray-MPI with optimizations for multi-threading and RMA through the DMAPP interface where tests showed good scaling up to 32K cores [19], which is approximately one order of magnitude less than the results shown in this paper. To the best of the authors' knowledge this is the largest number of cores used to test RMA-MT MPI ever performed, including the largest RMA-MT application run.

## 7 CONCLUSIONS AND FUTURE WORK

In this paper, we described our new RMA support for Open MPI, compared it to the old method, and conducted a study comparing Open MPI to Cray-MPI for multi-threaded MPI remote memory access optimized for a large Xeon Phi system utilizing a Cray Aries network. In particular, we leveraged the RMA-MT benchmark suite [9] and WOMBAT [21] to analyze performance of Open MPI's RDMA module [15] and the Cray-MPI's DMAPP solution [5]. The results of this study show that our improved Open MPI is better for the critical lock-all synchronization method, showing better performance on communication in miniapps and RMA applications. For mini-apps, Cray-MPI has better performance for very large scale runs including startup times, but Open MPI is competitive at large scale and superior for actual data exchange. Our study of WOMBAT demonstrated that for applications, Open MPI can outperform Cray-MPI by 8.6% at full scale.

The KNL many-core processors allowed us to evaluate multi-threaded RMA at core counts much larger than previous studies. It was found that both implementations allow for excellent scaling to large numbers of cores. This is encouraging for RMA-MT as an application design decision, leading us to believe that RMA-MT may be a viable programming approach for very large scale codes when optimized MPI implementations are available. While Cray's implementation is meant for specific hardware, Open MPI's optimizations at a general level can be applied to many different networks.

## 8 ACKNOWLEDGMENTS

The authors would like to thank the WOMBAT team (Peter Mendygral, Nick Radcliffe, Krishna Kandalla, David Porter, Brian J. O'Neill, Chris Nolting, Paul Edmon, Julius M. F. Donnert, and Thomas W. Jones) for providing access to their source code in order to test WOMBAT as well as guidance on configuration and execution of the code. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

## REFERENCES

[1] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray xc series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward efficient support for multithreaded mpi communication. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 120–129. Springer, 2008.

[3] B. W. Barrett, R. Brightwell, R. Grant, S. D. Hammond, and K. S. Hemmert. An evaluation of MPI message rate on hybrid-core processors. *International Journal of High Performance Computing Applications*, 28(4):415–424, 2014.

[4] R. F. Barrett, D. T. Stark, C. T. Vaughan, R. E. Grant, S. L. Olivier, and K. T. Pedretti. Toward an evolutionary task parallel integrated mpi+ x programming model. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 30–39. ACM, 2015.

[5] M. t. Bruggencate and D. Roweth. Dmapp: An api for one-sided programming model on baker systems. *Cray Users Group (CUG)*, 2010.

[6] Cray Inc. Using the GNI and DMAPP APIs. In *Cray Software Document*, volume S-2446-5202, Oct. 2014.

[7] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An implementation and evaluation of the mpi 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 2016.

[8] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible MPI endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[9] M. G. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges. Rma-mt: A benchmark suite for assessing mpi multi-threaded rma performance. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid 2016)*, 2016.

[10] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. Nov. 2013. IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13).

[11] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[12] W. D. Gropp and R. Thakur. Revealing the performance of mpi rma implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 272–280. Springer, 2007.

[13] T. Groves, R. E. Grant, and D. Arnold. Nimc: Characterizing and eliminating network-induced memory contention. In *IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2016.

[14] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep*, 2009.

[15] N. Hjelm. An evaluation of the one-sided performance in open mpi. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 184–187, New York, NY, USA, 2016. ACM.

[16] S. Huss-Lederman, B. Gropp, A. Skjellum, A. Lumsdaine, B. Saphir, J. Squyres, et al. MPI-2: Extensions to the message passing interface. *University of Tennessee, available online at http://www. mpiforum. org/docs/docs. html*, 1997.

[17] Intel. Intel MPI benchmarks 4.0. https://software.intel.com/en-us/articles/intel-mpi-benchmarks, 2015.

[18] H. Kamal and A. Wagner. Fg-mpi: Fine-grain mpi for multicore and clusters. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[19] K. Kandalla, P. Mendygral, N. Radcliffe, B. Cernohous, D. Knaak, K. McMahon, and M. Pagel. Optimizing cray mpi and shmem software stacks for cray-xc supercomputers based on intel knl processors. 2016.

[20] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304. ACM, 2003.

[21] P. Mendygral, N. Radcliffe, K. Kandalla, D. Porter, B. J. O'Neill, C. Nolting, P. Edmon, J. M. Donnert, and T. W. Jones. Wombat: A scalable and high-performance astrophysical magnetohydrodynamics code. *The Astrophysical Journal Supplement Series*, 228(2):23, 2017.

[22] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top500 supercomputing sites. http://www.top500.org/, 2013.

[23] MPI Forum. MPI: A message-passing interface standard version 3.1. Technical report, University of Tennessee, Knoxville, 2015.

[24] Ohio State University. OSU micro-benchmarks 4.4.1. http://mvapich.cse.ohio-state.edu/benchmarks/, 2015.

[25] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[26] Sandia National Laboratory. Mantevo project home page. https://mantevo.org, 2010.

[27] W. Schonbein, M. G. F. Dosanjh, R. E. Grant, and P. G. Bridges. Measuring multithreaded message matching misery. In *Proceedings of the International European Conference on Parallel and Distributed Computing*, 2018.

[28] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2014.

[29] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan. Early experiences co-scheduling work and communication tasks for hybrid mpi+ x applications. In *Proceedings of the 2014 Workshop on Exascale MPI*, pages 9–19. IEEE Press, 2014.

[30] R. Thakur and W. D. Gropp. Test suite for evaluating performance of mpi implementations that support mpi_thread_multiple. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 46–55. Springer Berlin Heidelberg, 2007.

[31] H. Weeks, M. G. Dosanjh, P. G. Bridges, and R. E. Grant. Shmem-mt: A benchmark suite for assessing multi-threaded shmem performance. In *Workshop on OpenSHMEM and Related Technologies*, pages 227–231. Springer International Publishing, 2016.