OpenSHMEM over MPI as a Performance Contender: Thorough Analysis and Optimizations

Min Si[1], Huansong Fu[2], Jeff R. Hammond[3], and Pavan Balaji[4]

[1] Argonne National Laboratory, USA
minsi.atwork@gmail.com
[2] Amazon, USA
huansong.fu1@gmail.com
[3] NVIDIA Corporation, USA
jeff_hammond@acm.org
[4] Facebook, USA
pavanbalaji.work@gmail.com

Abstract. OpenSHMEM is a Partitioned Global Address Space (PGAS) style programming model for one-sided scalable communication over distributed-memory systems. The community has always focused on high levels of performance for specific communication operations such as RMA, atomics, and collectives and encourages native implementations directly porting onto each network hardware in order to pursue minimal instructions from the application to the network hardware. OSHMPI is an OpenSHMEM implementation on top of MPI, which aims to provide portable support of the OpenSHMEM communication over mainstream HPC systems. Because of the generalized functionality of MPI, however, OSHMPI incurs heavy software overheads in the performance-critical path.
Why does OpenSHMEM over MPI not perform well? In order to answer this question, this paper provides an in-depth analysis of the software overheads of the OSHMPI performance-critical path, from the aspects of both the semantics and the library implementation. We also present various optimizations in the MPI and OSHMPI implementations while maintaining the full MPI functionality. For remaining performance overheads that fundamentally cannot be avoided based on the MPI-3.1 standard, we recommend extensions to the MPI standard to provide efficient support for OpenSHMEM-like PGAS programming models. We evaluate the optimized OSHMPI by comparing with the native implementation of OpenSHMEM on an Intel Broadwell cluster with the Omni-Path interconnect. The evaluation results demonstrate that the optimized OSHMPI/MPI environment can deliver performance similar to that of the native implementation.

1 Introduction

OpenSHMEM is a widely used Partitioned Global Address Space (PGAS) style programming model for distributed-memory systems. As the fundamental principle of the OpenSHMEM model, the community has heavily focused on high

levels of performance for specific one-sided or collective communication patterns through explicit data transfer operations. The intent of the OpenSHMEM specification is to get to "close to zero instructions" from the application to the network hardware. For instance, each data transfer operation has the unique typed version (i.e., separate function for each basic type such as `shmem_int_put` and `shmem_double_put`). These functions embed the data type information as a constant value at OpenSHMEM library compile time. Consequently the library code can be highly optimized for each type without any type check overhead. Following such a principle, the community has developed native implementations that are highly optimized for different vendor platforms (e.g., SGI SHMEM, Cray SHMEM). Alternatively, some OpenSHMEM implementations tend to gain portability by porting onto low-level network frameworks (e.g., Sandia OpenSHMEM (SOS) over Open Fabrics Interfaces (OFI) and OSHMEM over Unified Communication X (UCX)). Nevertheless, these implementations still optimize for a specific platform (e.g., SOS/OFI is optimized primarily for the Intel Omni-Path architecture) and require the user to manually find the appropriate solution.

OpenSHMEM over MPI is the way to gain broader portability and vendor support. In fact, MPI is recognized as the de facto standard for communication on distributed-memory systems and supported by all major high-performance computing (HPC) vendors and common parallel computing platforms. More importantly, the MPI ecosystem covers powerful performance and debugging tools, all of which are now available for use in OpenSHMEM programs. OSHMPI [**?**] is the OpenSHMEM implementation built on top of the MPI-3 one-sided communication model (also as known as RMA). However, it is treated primarily as a functionality reference rather than as a serious performance contender. The general belief in the community is that such a heavy software stack (e.g., OSHMPI/MPI/OFI) often generates bulky communication instructions and may even cause significant performance loss.

Why does OpenSHMEM over MPI RMA not perform well? The primary goal of this paper is to answer this question through a detailed deep-dive and scientifically thorough analysis. From a high-level overview, we believe the performance loss can be caused by two reasons. First, many MPI implementations do not optimize the one-sided communication routines. Second, the MPI standard provides more generalized functionality than that of OpenSHMEM. The generalization makes various complex algorithms possible to write, but it comes with additional cost. For instance, a user can specify arbitrarily complex noncontiguous derived datatypes in a call to `MPI_Put`. MPI has to always check even if such a functionality is not needed, such as in the context of OpenSHMEM over MPI where only basic datatypes are used.

To diagnose all performance issues, we systemically analyze all instructions generated for the OpenSHMEM over MPI context. Based on the analysis, we further optimize the OSHMPI and MPI implementation to enable a fast path for the OpenSHMEM context while still maintaining the full MPI functionality. For any overhead that fundamentally cannot be removed, we recommend extensions to the MPI standard to enhance support for the generic PGAS over MPI

scheme. We employ a refactored version of the OSHMPI library and the MPICH implementation with the `ch4:ofi` configuration to demonstrate the performance study. MPICH provides a highly optimized MPI implementation by reducing the software overhead and using techniques such as static builds and link-time interprocedural optimization (IPO) inlining. It enables a very fast MPI Put/Get path for both OFI and UCX [**?**].

To correctly capture and evaluate only the performance overhead of the OpenSHMEM over MPI approach, we compare our implementation with the SOS implementation of OpenSHMEM on an Intel Omni-Path platform where both implementations are built on top of the same underlying OFI framework. The experimental results demonstrate that the optimized OSHMPI/MPICH can deliver performance similar to that of native implementations.

Scope of this paper: While OpenSHMEM defines several kinds of communication routines, this paper focuses on the fundamental limits in implementing the most performance-critical and essential RMA routines in OpenSHMEM on top of MPI RMA. We believe similar observations can be extended to other functions.

2 Background

In this section we compare the semantics of the one-sided models in OpenSHMEM and MPI and briefly introduce the design of OSHMPI as the reference OpenSHMEM over MPI implementation.

2.1 Semantics Overview

Both OpenSHMEM and MPI define the one-sided communication model. The semantics, however, have several key differences. We summarize the differences below.

Memory Exposure: OpenSHMEM defines the concept of symmetric data objects including symmetric heap and global/static variables. The symmetric data is remotely accessible for all processes. Unlike OpenSHMEM, MPI requires the user to explicitly expose a remotely accessible memory region called *window*. Each window object is associated with a *communicator* (i.e., a group of processes). This semantics allows the user to benefit from *communication virtualization*. For instance, a user can create multiple communicators with the same group of processes. With each communicator, the user can also create multiple windows for the same memory buffer. The communication with each communicator (or window) is fully isolated. We note that OpenSHMEM specification 1.5 introduces the teams concept that provides similar communication virtualization. In this paper, we focus only on the implementation of the default symmetric data objects.

Operation Expression: In OpenSHMEM, the RMA operation routines directly deal with the absolute virtual address of the remote buffer, and the data

type of each operation is encoded in the function interface (e.g., `shmem_int_put`). The interface of MPI RMA operations has two differences. First, it requires the relative *displacement* of the remote buffer rather than an absolute virtual address. This was designed to meet the requirements of various networks, some of which require relative offset whereas others require an absolute virtual address. The second difference is that both basic data types (e.g., *integer* or *float*) and complex user-defined data layout (e.g., *vector* or *struct*) are specified as a datatype input parameter. This allows an MPI RMA operation to carry arbitrary data layout.

2.2 OSHMPI

As indicated in the semantics comparison, MPI provides more generalized functionalities than OpenSHMEM does. Thus, the one-sided communication of OpenSHMEM can be fully expressed by using MPI RMA. OSHMPI-1.0 is a reference implementation of OpenSHMEM 1.2 over MPI-3 [**?**]. As the basis of this study, we redeveloped the OSHMPI library to fully support OpenSHMEM 1.4 and released it as OSHMPI-2.0b1.[1] We give here a brief overview of its high-level design.

OSHMPI internally creates two MPI windows at OpenSHMEM initialization, one for symmetric heap and the other for global/static variables. Every process locks the two windows by calling `MPI_Win_lock_all` immediately after window creation. Thus, each OpenSHMEM Put/Get operation can be implemented by using the corresponding MPI operation followed with a call to `MPI_Win_flush_local`.[2] The `shmem_quiet` synchronization can be implemented by using `MPI_Win_flush_all`[3] and `MPI_Win_sync`.[4] At finalization, OSHMPI unlocks the internal windows on all processes and frees the windows before making a call to `MPI_Finalize`.

3 Related Work

In this section we describe the related work in the following three categories.

Native OpenSHMEM Implementations: The original implementations of SHMEM were native implementations directly on top of hardware such as the Cray T3D [**?**]. Subsequent native implementations included QSHMEM for the Quadrics Elan network [**?**]. Many SGI platforms offered an optimized native implementation of SHMEM, including ccNUMA systems. Cray SHMEM is the

[1] *https://github.com/pmodels/oshmpi/releases/tag/v2.0b1*

[2] Flush_local locally completes all outstanding RMA operations initiated by the calling process to the remote process specified by rank on the window.

[3] Flush_all ensures all outstanding RMA operations issued by the calling process to any remote process on the window will have completed both at the local and at the remote side.

[4] Win_sync synchronizes memory updates on the specific window.

highly optimized native implementation for Cray XC and XK series supercomputers. It is directly implemented on top of the low-level DMAPP API. Cray OpenSHMEMX [?] is a OpenSHMEM specification version 1.4 compliant native implementation for current and future-generation Cray systems. On InfiniBand (IB) platforms, three OpenSHMEM implementations are commonly used. OSHMEM[?] is an implementation of OpenSHMEM API that is distributed within the Open MPI distribution. It is implemented on top of the low-level UCX communication framework. Scalable-SHMEM [?] is the native implementation for Mellanox IB and works with the OpenFabrics RDMA for Linux stack (OFED). MVAPICH2-X is the hybrid MPI+PGAS release of MVAPICH library and is highly optimized for IB systems [?]. On Intel Omni-Path systems, SOS [?] is the primary native implementation. It is implemented on top of the low-level OFI communication framework. Our analysis for the OSHMPI/MPICH stack utilizes the same OFI framework. Thus we choose SOS as the representative of OpenSHMEM native implementations and compare it with our implementation in this paper.

Other PGAS over MPI Implementations: MPI is often used as the portable underlying communication runtime of high-level PGAS libraries. Dinan et al. [?] analyzed the semantic mismatch between the the ARMCI communication interface of Global Arrays and MPI-2 RMA and evaluated the performance of Global Arrays applications on the resulting implementation, ARMCI-MPI [?], on three different HPC platforms. Since the introduction of MPI-3, ARMCI-MPI is able to use RMA quite naturally, and the current implementation maps ARMCI's one-sided operations directly to MPI's. DASH [?] is a C++ template library following a PGAS-like programming model. DART-MPI [?] is a portable implementation of the DASH runtime based on the MPI-3 shared memory support and RMA operations. OpenCoarrays is a library that supports the Fortran 2008 coarrays PGAS model using MPI (and possibly other communication protocols), which is used by GCC Fortran today [?]. The Intel Fortran implementation of coarrays is based on MPI-3 one-sided communication [?]. Bonachea and Duell [?] analyzed the usage of the MPI-1 two-sided model and MPI-2 RMA for Global Address Space (GAS) languages such as Unified Parallel C (UPC) and Co-Array Fortran (CAF). Their analysis showed that those MPI-1 and MPI-2 models are unsuitable for GAS languages. Yang et al. [?] then demonstrated that the more comprehensive MPI-3 RMA framework can be used as the runtime of CAF with a broader goal of enabling a single application to use both MPI and CAF with high interoperability. All these previous studies focused on the complete functionality and high-level performance. In contrast, our work pursues more fine-grained semantics-mismatch and overhead analysis together with a comprehensive performance fine-tuning. None of these aspects are covered by previous studies. We also note that the outcome from our work may also apply to the other PGAS over MPI libraries.

Software Overhead Analysis: Raffenetti et al. [?] analyzed the software overhead of the MPICH implementation of MPI. Their analysis focused primarily on the instruction-overhead critical paths including MPI send/and MPI Put

operations, and the optimizations were proposed for general MPI applications. In contrast, this paper focuses on the OpenSHMEM context that gives more restricted semantics than that of the underlying MPI layer, thus exposing different overheads and optimization opportunities. Our analysis covers both the instruction-overhead critical MPI Put/Get operations and the time-consuming synchronization routines. We note that our work is based on the MPICH implementation that includes all optimizations presented in [**?**].

4 Analysis of Performance Loss in Contiguous RMA

Although OSHMPI is functional, many of the generalized features of MPI are unnecessary for the support of OpenSHMEM and even cause performance loss, especially in the performance-critical Put/Get routines in comparison with a native implementation of OpenSHMEM. We demonstrate and analyze the performance loss in the rest of this section. We note that we present only the instruction analysis of the Put path, but the observations can be fully applied also to the Get path. Thus, we omit the description of Get.

Our analysis and optimizations are based on the OSHMPI-2.0b1 and MPICH-3.3[5] releases. OSHMPI-2.0b1 fully supports the OpenSHMEM 1.4 specification and enables function inlining for all OSHMPI internal routines. We utilize the `ch4:ofi` configuration of MPICH that provides highly optimized MPI RMA [**?**]. In the remainder of this paper OSHMPI refers to the OSHMPI-2.0b1 version and MPICH refers to the `ch4:ofi` configuration of MPICH-3.3 unless otherwise specified. To emphasize the extra software overhead caused by the MPI layer, we compare the internal implementation and the instructions of OSHMPI/MPICH with those of SOS 1.4.2 release as the representative of native implementations. Our analysis utilizes a basic latency scenario where one process performs `shmem_putmem` followed with a call `shmem_quiet` to the remote process. We discuss their internal implementations and analyze the overhead separately.

shmem_putmem: It issues a Put operation to the remote process, and returning from this function ensures the source buffer can be reused. In other words, the Put operation is locally completed. A native implementation of `shmem_putmem` usually consists of only a few internal steps. For instance, SOS implements this routine with two steps: (1) preparing OFI write parameters and making a call to `ofi_inject_write` or `ofi_write`,[6] and (2) waiting the local completion of the outstanding write by calling `fi_cntr_read` and `fi_cntr_wait`.[7] In contrast, OSHMPI/MPICH involves a number of additional steps, as demonstrated in Figure 1. We separate these steps into three phases and describe each step below.

[5] *http://www.mpich.org/downloads/*

[6] `ofi_inject_write` is used for data smaller than 64Bytes, and `ofi_write` is used for other data sizes. The latter only initiates a write to remote memory, but the former also guarantees local completion.

[7] `fi_cntr_read` reads an OFI event counter that is updated at operation completion, and `fi_cntr_wait` is its blocking version.

```
1 shmem_putmem(dest, source, nelems, pe) {
2     translate_win_and_disp(dest, &win, &disp); // (1)
3
4     /* nonblocking put */
5     MPI_Put(source, nelems, src_dtype=MPI_BYTE, pe, disp, nelems,
          dest_dtype=MPI_BYTE, win) {
6         win_get_ptr(win, &win_ptr); // (2)
7
8         trans_rank_to_netaddr(pe, win_ptr->comm, &nw_addr); // (3)
9
10        decode_dtype(src_dtype, &src_size, &src_contig,...); // (4)
11        decode_dtype(dest_dtype, &src_size, &dest_contig,...);
12        if (src_contig && dest_contig && bytes <= ofi_max_bytes) { // (5)
13            prepare_ofi_write_parameters(...); // (6)
14            dest_vaddr = disp + win_ptr->abse; // (7)
15            ofi_inject_write(...);
16        }
17    }
18
19    /* ensure local completion of nonblocking put */
20    MPI_Win_flush_local(pe, win) {
21        win_get_ptr(win, &win_ptr); // (8)
22        wait_ofi_completion(...); // (9)
23        target_ptr = win_find_am_target(win_ptr, pe); // (10)
24        do {
25            MPI_full_progress(); // (11)
26        } while (target_ptr && target_ptr->local_cmpl_cnts != 0);
27    }
28 }
```

Fig. 1: Pseudo code of `shmem_putmem` implementation in OSHMPI/MPICH

- *Phase-1: MPI parameter preparation.* The OSHMPI layer translates the `dest` buffer address to its corresponding window handle (i.e., either the window for symmetric heap or the one for global/static variables) and the relative displacement (step (1) in Figure 1).
- *Phase-2: MPI Put.* It then makes a call to `MPI_Put` with the `MPI_BYTE` datatype. The implementation of `MPI_Put` can be further divided into six steps. It first gets the internal object pointer of the `win` handle (step (2)). The internal object is used to store window attributes such as the initial address, size, and displacement unit of remote memory regions associated with this window. It next translates the remote process's rank in the window's communicator to its physical network address (step (3)). The network address will be used when posting an OFI write. Because the ranks in each communicator can be arbitrarily reordered, the address lookup is an expensive operation. It then decodes the source and destination datatypes to obtain the data layout such as data sizes and whether the data is contiguous (step (4)). After that, it checks whether both source and destination datatypes are contiguous and other OFI conditions are met (step (5)). If so, it then prepares OFI write parameters (step (6)), calculates the absolute virtual address of the destination buffer (step (7)), and makes a call to `ofi_inject_write` or `ofi_write` similarly to the implementation of SOS.
- *Phase-3: Local completion.* Because `MPI_Put` is a nonblocking operations, we need to issue `MPI_Win_flush_local` on the corresponding window to ensure its local completion. The internal implementation of flush_local can be broken into four steps. It first gets the internal object pointer of the `win` handle (step (8)). It next waits for the completion of any outstanding writes in OFI by calling a loop of `fi_cntr_read` and `fi_cntr_wait` (step (9)). It then

checks whether there is a target active message object associated with the remote rank (step (10)). This step is necessary for MPICH because some RMA operations (e.g., Put with very sparse noncontiguous data or Accumulates with a network-unsupported reduce operation) cannot be offloaded to hardware and have to fall back to the active-message-based approach. If the target active message object exists, it then triggers MPI full progress until all outstanding active messages on the target process are locally completed; otherwise, it makes the full progress once (step (11)). MPICH ensures that the full progress is always triggered in blocking communication calls in order to guarantee prompt progress for all MPI communication types such as point-to-point, collectives, and internal active messages.

Obviously, these additional steps in OSHMPI/MPICH generate a significant number of CPU instructions on the performance critical operation path. We used the Intel SDE tool to emulate instructions generated from the OpenSHMEM latency program statically linked with the OSHMPI and MPICH libraries and that linked with SOS. The instructions were generated with `nelem`=4 in `shmem_putmem`. Table 1 summarizes the instruction counts generated by each internal step of OSHMPI/MPICH (see the Original Count column) and that of SOS. As expected, OSHMPI/MPICH consumes more significant instructions than does SOS. The total instruction count of OSHMPI/MPICH is 333 whereas SOS consumes only 71, without counting the instructions of the underlying OFI library. We especially emphasize the instructions caused by the requirement of MPI semantics (rows are highlighted in gray), which are completely unnecessary for the SOS implementation.

Table 1: `shmem_putmem` instruction count analysis with parameter `nelem`=4. Gray rows indicate instructions caused by the requirement of MPI semantics. The others instructions in MPI and SOS are implementation-specific; we omit the description.

OSHMPI Internal Step	Orig Cnt	Opt Cnt	SOS Internal Step	Cnt
OSHMPI: calling overhead	14	16	SOS: calling overhead	16
(1) OSHMPI: trans win and disp	12	5	-	-
MPI_Put: calling overhead	9	0	-	-
(2) MPI_Put: get win obj	14	9	-	-
(3) MPI_Put: trans rank to network address	17	5	-	-
(4) MPI_Put: decode dtypes	22	0	-	-
(5) MPI_Put: check OFI conditions	13	7	SOS: check OFI conditions	7
(6) MPI_Put: prepare OFI param	14	8	SOS: prepare OFI param	24
(7) MPI_Put: compute dest_vaddr, mr_rkey	8	1	-	-
Flush_local: calling overhead	8	0	-	-
(8) Flush_local: get win obj	7	8	-	-
(9) Flush_local: wait OFI completion	38	17	SOS: wait OFI completion	-*
(10) Flush_local: find targets with active msg	59	0	-	-
(11) Flush_local: MPI full progress	81	2	-	-
MPI: others	15	15	SOS: others	24
OSHMPI total	333	93	SOS total	71

* SOS skips completion waiting for data smaller than 64 bytes because it uses `fi_inject_write`, which ensures local completion at return. Such an optimization cannot be done in MPICH because flush_local cannot determine whether other RMA operations (Get or large Put) has been issued. Thus, it has to always check the OFI completion counters (step (9)).

shmem_quiet: We then perform a similar analysis for the `shmem_quiet` routine. This ensures completion of all remote memory access and memory updates. In SOS, the synchronization is done by waiting for OFI completion counters. OSHMPI, however, has to take several MPI calls to ensure the semantics correctness required by `shmem_quiet`. Figure 2 demonstrates the internal implementation of `shmem_quiet` in OSHMPI/MPICH. OSHMPI internally calls `MPI_Win_flush_all` and `MPI_Win_sync` at quiet. Because OSHMPI creates two windows, one for symmetric heap (`symm_heap_win`) and the other for global/static variables (`symm_data_win`), it must call the two MPI functions twice.

In flush_all, MPICH first gets the internal object pointer of the `win` handle (step (1)). It next waits for OFI completion (step (2)). It then traverses all target objects that are associated with the specific window to ensure any outstanding active-message-based operations in this window are completed remotely (step (4)). While waiting for the active message completion, it iteratively makes MPI full progress. Similar to flush_local, the full progress is made at least once (step(3)). The next MPI function is win_sync, which is used for memory synchronization.

```
1 shmem_quiet() {
2     /* ensure remote completion */
3     MPI_Win_flush_all(win=symm_heap_win) {
4         win_get_ptr(win, &win_ptr); // (1)
5         wait_ofi_completion(...); // (2)
6
7         target_am_all_cmpl = TRUE;
8         do {
9             MPI_full_progress(); // (3)
10
11            // (4)
12            /* traverse targets that received active message to ensure
                  remote completions on all targets */
13            for (pe = 0; pe < win_ptr->comm_ptr->local_size; pe++) {
14                target_ptr = win_find_am_target(win_ptr, pe);
15                if (target_ptr && target_ptr->remote_cmpl_cnts) != 0) {
16                    target_am_all_cmpl = FALSE; break;
17                }
18            }
19        } while (!target_am_all_cmpl);
20    }
21    /* ensure memory updates */
22    MPI_Win_sync(win=symm_heap_win) { // (5)
23        memory_barrier();
24    }
25    MPI_Win_flush_all(win=symm_data_win) { // (6)
26        /* same as above */
27    }
28    MPI_Win_sync(win=symm_data_win) { // (7)
29        /* same as above */
30    }
31 }
```

Fig. 2: Pseudo code of `shmem_quiet` implementation in OSHMPI/MPICH

Table 2 summarizes the instruction count of `shmem_quiet` generated by OSHMPI/MPICH (see the Original Count column). The dominant cost in the OSHMPI/MPICH path comes from the MPI full progress and the traversal of target objects (steps (3–4)) in `MPI_Win_flush_all`, both are required by MPI semantics. Such a cost is even doubled because OSHMPI internally maintains two win-

dows. As a result, OSHMPI/MPICH consumes 544 instructions whereas SOS consumes only 91. We note that the result captures the instructions taken by the Put latency program where only one Put is issued prior to a quiet. Thus, OSHMPI/MPICH does not issue any active-message-based operation and makes the MPI full progress only once.

Table 2: `shmem_quiet` instruction count analysis. Gray rows highlights instructions caused by the requirement of MPI semantics. The others instructions in MPI and SOS are implementation-specific; we omit the description.

OSHMPI Internal Step	Orig Cnt	Opt Cnt	SOS Internal Step	Cnt
OSHMPI: calling overhead	15	15	SOS: calling overhead	15
Flush_all: calling overhead	4	0	-	-
(1) Flush_all: get win obj	7	7	-	-
(2) Flush_all: wait OFI completion	14	14	SOS: wait OFI completion	51
(3) Flush_all: MPI full progress	81	2	-	-
(4) Flush_all: traverse targets with active msg	130	0	-	-
Win_sync: calling overhead	4	0	-	-
(5) Win_sync: memory barrier	1	1	-	-
(6) Flush_all for global/static var	267	3	-	-
(7) Win_sync for global/static var	5	0	-	-
MPI: others	16	2	SOS: others	25
OSHMPI total	544	44	SOS total	91

5 Optimizations for Fast RMA

Based on the overhead analysis in the preceding section, we then investigate ways to optimize the `shmem_putmem` and `shmem_quiet` in the OSHMPI/MPICH environment. We note that although our optimizations and discussion are based on the MPICH implementation, most address general issues also exist in other MPI implementations.

5.1 Basic Datatype Decoding with IPO Link-Time Inlining

Each OpenSHMEM RMA operation directly encodes the datatype in the function calls and supports only the standard RMA types. The datatype information is treated as a constant in the native implementations. Unlike OpenSHMEM, MPI allows the user to specify arbitrary datatypes such as the basic datatype `MPI_INT` or a complex user-defined derived datatype (e.g., vector, struct). The datatype description is encoded into the `MPI_datatype` object passed to MPI calls as an input variable. MPICH cannot optimize the datatype decoding process at compile time because the value of the datatype variable is unknown. Because of such a semantics limitation, the constant information of datatypes was lost in OSHMPI/MPICH and caused 22 additional instructions at the RMA fast path (see Table 1 step (4)). Many of these instructions are expensive pointer dereferences (i.e., to extract the attributes of the datatype object).

The interprocedural (IPO) optimization technique allows compiler to optimize code across source files and libraries at link time. This feature is provided by mainstream modern compilers such as the Intel compiler and the LLVM family. One of the IPO features is to inline functions across libraries and apply constant propagation for all inlined functions.

We note that IPO is extremely time-expensive when the optimizing space is large. Thus, we need to carefully define the inlining scope. Specifically, we make the following two configurations: (1) We inline only OSHMPI and MPICH libraries at link time, and (2) we explicitly exclude any *non-performance-critical* path in OSHMPI such as `shmem_init`. After applying IPO link-time inlining, we observe that MPICH can recognize the basic datatype defined for each RMA operation as a constant (e.g., `MPI_INT` is for `shmem_int_put`).

Once the datatype parameter becomes a constant, we then reconstruct the MPI datatype decoding routine to eliminate pointer dereferences. Specifically, we embed the required datatype attributes into the object handle rather than storing them as object fields. Such an approach works for basic datatypes because they require only two essential attributes when issuing an RMA operation: datatype kind (i.e., basic or derived) and size in bytes. The former is to distinguish a basic datatype from more complex derived datatypes; thus the fast-path code can be chosen. The latter is required for issuing the corresponding network data transfer. MPI implementations such as MPICH, MVAPICH, and Cray MPI represent the object handle as a 32-bit integer. It allows us to reserve a few bits for the two attributes. We note that the handle-embedded approach might be more complicated for MPI implementations whose object handle is represented as address pointers (e.g., OpenMPI). However, most architectures require some level of alignment for all pointer allocations (typically 4-byte or 8-byte alignment). Thus, even though the pointer uses 64 bits to represent the address, the two or three least significant bits are unused for alignment reasons. Therefore, the MPI implementation can reserve those bits to embed such information.

The attribute extraction now becomes bit-and and bit-shift instructions operated on the datatype handle. Thanks to IPO, these instructions can be fully eliminated by the compiler since the handle is recognized as a constant value at link time. Hence, no instruction is generated for datatype decoding in our optimized OSHMPI/MPICH, just as that in native implementations.

5.2 Fast Window Attributes Access

MPI implementations usually maintain an internal data object for each window. The object stores window attributes such as the associated communicator, network address, network endpoint (`ep`), remote window's memory registration key (`mr_rkey`), and remote window's displacement unit (`disp_unit`). At each RMA operation, the MPI implementation has to load these window attributes to prepare necessary parameters for network data transfer as well as for optimizations (e.g., one may compare the target rank with the rank of the local process in the communicator and perform local copy if they are identical). Accessing each

attribute field is essentially a pointer dereference, however, and may involve expensive memory access overhead. Such an overhead can be significant especially when multilevel pointer dereferences are involved (e.g., accessing any attribute of the window's internal communicator is a two-level dereference).

Table 3: Pointer dereferences and instruction counts caused by window attributes access inside `MPI_Put`.

Internal Step	#Ptr Deref	Instr Cnt
1. Translate rank to network address	2	8
2. Check target_rank for self message optimization	2	3
3. Prepare OFI parameters (ep, base, mr_rkey, disp_unit)	4	13
4. OFI counter update for tracking completion	1	4

Table 3 shows the pointer dereferences and relevant instructions taken inside each internal step of an `MPI_Put` call. We note that the network address translation (step 1) is required by the MPI semantics because the process's rank can be arbitrarily reordered in different communicators. Thus, MPICH has to maintain a lookup table to translate the process's rank in each communicator to the physical network address. The lookup table implementation was highly optimized in MPICH especially for common communicator patterns [**?**]. Figure 3a shows the assembly code generated for this step within the context of an OSHMPI-issued Put. The communicator is duplicated from `COMM_WORLD` (i.e., defined as the `DIRECT_INTRA` communicator mode). Thus it can utilize the fast lookup path with only 8 instructions. In order to load the communicator mode of the window and choose the fast code path, however, two pointer dereferences cannot be avoided (lines 1–2). We observed a similar situation in step 2. That is, in order to check whether the target process is the process itself (i.e., a self message), MPICH has to access the communicator's internal field, causing a two-level dereference (see lines 1–2 in Figure 4a). We note that most MPI implementations contain this step in every RMA operation because it allows a self-message to be directly transferred in the MPI layer through `memcpy`. Steps 3 and 4 are required by the semantics of OFI data transfer and also can be found in a native implementation of OpenSHMEM. Thus, our optimization focuses only on the former two steps.

Similar to the object handle of datatypes, we noticed unused bits also in the window handle. Thus, we can identify whether the communicator is the `DIRECT_INTRA` mode when creating the window, and we can reserve a "window attribute" bit from the window handle to store such information. When issuing an RMA operation, we first check the value of the "window attribute" bit rather than loading the communicator's mode through pointer dereferences. We note that the window handle has already been loaded into the CPU register when converting to the internal window object; thus, checking a bit of the handle is very lightweight. In the context of OSHMPI, the windows are always created over the simplest `DIRECT_INTRA` communicators. Thus, the optimization can effectively eliminate the communicator dereferences in steps 1 and 2 for all RMA operations. Figures 3b and 4b show the optimized assembly code.

```
1 /* load win->comm */
2 mov r9, qword ptr [rdi+0x70]
3 /* load comm->mode */
4 mov edx, dword ptr [r9+0x1b8]
5 /* mode == DIRECT_INTRA? */
6 cmp rdx, 0xb
7 jnbe 0x41db85
8 jmp qword ptr [rdx*8+0x769560]
9 /* load table */
10 mov rax, qword ptr [rip+0x60522f]
11 /* shift to table[target_rank] */
12 add rax, 0x28
13 jmp 0x41db85
```

(a) Original version

```
1 /* handle & DIRECT_INTRA_MASK? */
2 test ebx, 0x2000000
3 jz 0x41cfb5
4
5
6
7
8
9 /* load table */
10 mov rax, qword ptr [rip+0x647740]
11 /* shift to table[ target_rank ] */
12 add rax, 0x28
13 jmp 0x41d203
```

(b) Optimized version

Fig. 3: Translating rank to network address in Put operation with optimization of embedded window attributes.

```
1 /* load win->comm */
2 mov rdx, qword ptr [rdi+0x70]
3 /* comm->rank == target_rank? */
4 cmp dword ptr [rdx+0x50], 0x1
5 jz 0x41dc32
```

(a) Original version

```
1 /* load global comm_world_rank */
2 mov edx, dword ptr [rip+0x6278a0]
3 /* comm_world_rank == target_rank? */
4 cmp edx, 0x1
5 jnz 0x41d238
```

(b) Optimized version

Fig. 4: Checking self-message in Put operation with optimization of embedded window attributes.

5.3 Avoiding Virtual Address Translation

Unlike OpenSHMEM, MPI defines generic relative offset (i.e., *displacement*) to describe the address of the remote RMA buffer. This allows MPI to be compatible with different requirements for remote memory access performed by the network hardware. For instance, some networks require the relative offset of the remote buffer (e.g., the OFI/psm2 provider for Intel Omni-Path), but others may require an absolute virtual address of the remote buffer (e.g., the OFI/gni provider for Cray Aries interconnect and UCX for InfiniBand networks). When utilizing MPI RMA in OSHMPI, however, we always must translate the remote absolute virtual address defined in OpenSHMEM to the corresponding relative displacement for every RMA operation. For networks that prefer absolute virtual address, a consequent translation (i.e., from relative displacement to virtual address) has to be performed again in the MPI layer. Obviously, such a translation is redundant.

Unfortunately, we cannot eliminate the redundant translation if we treat the MPI standard as a constant. To demonstrate the more efficient approach, we extended the MPI standard with a set of new functions called `MPI_Put|Get_abs` that can directly take the absolute virtual address as the input parameter. Figure 5 gives the API definition. Compared with the standard `MPI_Put|Get`, the only change is `target_vaddr`, which was originally a *displacement*.

This way allows us to avoid the intermediate remote address translation in OSHMPI and MPICH for networks that prefer absolute virtual address (e.g., Cray Aries and InfiniBand). However, we noticed that such an optimization can cause an extra translation in the MPI layer for networks that require relative

```
1 int MPI_Put_abs(const void *origin_addr, int origin_count,
2                 MPI_Datatype origin_datatype,
3                 int target_rank, MPI_Aint target_vaddr, int target_count,
4                 MPI_Datatype target_datatype, MPI_Win win);
5 int MPI_Get_abs(void *origin_addr, int origin_count,
6                 MPI_Datatype origin_datatype,
7                 int target_rank, MPI_Aint target_vaddr, int target_count,
8                 MPI_Datatype target_datatype, MPI_Win win);
```

Fig. 5: API definition of the *abs* extension for `MPI_Put|Get`.

offset (e.g., Intel Omni-Path) at each of the extended *abs* function. To eliminate such a translation, we require the user of MPI to use either only the basic RMA functions or only the extended functions for each window. The user should choose the preferred mode based on the application context. For instance, in OSHMPI the abs functions are clearly more suitable. We then defined a window info hint "`rma_abs`" (value is `true` or `false`) to indicate whether the window is exclusively used by the extended abs operations. If `rma_abs` is `true` and the underlying network requires relative offset, then MPICH internally registers `MPI_BOTTOM` as the base address of the virtual memory region on each process. For each RMA operation, the relative offset can be calculated by (`target_vaddr`−`MPI_BOTTOM`). Because `MPI_BOTTOM` is a predefined constant in MPI, the arithmetic calculation instructions can be fully eliminated by the compiler.

5.4 Optimizing MPI Progress

MPI implementations usually make expensive "full progress" in various MPI blocking functions. The full progress guarantees that all types of MPI communication (i.e., point-to-point, collectives, and active message based RMA) can be promptly progressed. For instance, for an active message based communication, the remote process has to trigger the MPI progress in an MPI call to complete the exchange of internal data packets. The MPI progress also internally triggers low-level network progress by making network synchronization calls such as `fi_cq_read` for OFI or `ucp_worker_progress` for UCX. These calls ensure prompt progress for any internal software emulation (e.g., active message based RDMA) or data processing (e.g., to move data out from a preregistered internal buffer) at the low-level network libraries.

Both OpenSHMEM RMA and quiet operations involve the MPI full progress in OSHMPI/MPICH. Table 4 analyzes the instructions that are taken for progress-relevant internal steps in `shmem_putmem` and `shmem_quiet`. We note that these steps are expensive not only in instruction counts but also in time because they force memory synchronization with the network hardware.

The expensive progress steps are required for general MPI programs. *Are they necessary also in the special OSHMPI context*? To answer this question, we systemically analyze the MPI progress requirements below.

For MPI Point-to-Point/Collectives: Both MPI point-to-point and collectives require two-sided communication between local and remote processes. Thus, the remote process must ensure prompt progress, For instance, the eager protocol designed for small point-to-point messages requires the receiver process to

Table 4: Progress-relevant internal steps in `shmem_putmem` and `shmem_quiet` in OSHMPI/MPICH.

shmem_putmem	Instr Cnt
Flush_local: wait completion of outstanding RMA operations	38
Flush_local: MPI full progress	24

shmem_quiet	Instr Cnt
Flush_all: wait completion of outstanding RMA operations	14
Flush_all: MPI full progress	24

copy data from the MPI internal buffer to the user receive buffer in order to complete the data transfer. This step may be performed in the progress routine on the receiver process. For collectives, for example, a process involved in an `MPI_Bcast` call may receive the data from the root process and then need to forward the data to another member process. Such a protocol is commonly used to overlap multiple data transfer in collective algorithms. The data receiving and forwarding steps are performed by the progress routine on each member process. The point-to-point and collective semantics require all processes involved in the communication to make the call. Hence, a process need perform such steps only when a collective or point-to-point call has been made.

For Active-Message-Based MPI RMA: MPI implementations may utilize internal active messages for an RMA operation if the underlying network hardware cannot efficiently handle it. For instance, a pack+AM+unpack-based approach may be chosen for a noncontiguous Put if the data layout is very sparse. An `MPI_Accumulate` has to be implemented by using an active message if the network hardware cannot guarantee atomic updates to the remote buffer with the specified datatype or if the MPI implementation chooses to use only CPU-based atomicity in order to be compatible with direct load/store-based intranode Accumulates. Nevertheless, the MPI implementation always must assume that the process may receive an active message from the other processes because the above situations may potentially occur. Consequently, the progress routine always has to be performed to promptly handle any incoming active message.

One may consider that the MPI implementation may predict whether active messages will be used by remote processes and skip the progress routine when possible. Such an approach, unfortunately, is complex because of two limitations. First, we need information from both the user program and the underlying network. To be specific, the user program must provide the (1) operation type (i.e., for atomic operations), (2) the basic datatype and data layout (e.g., contiguous or sparse noncontiguous), and (3) the data length for each operation. The network library must provide the (4) supported data layout for each operation together with (5) the data length limitation (e.g., for ordered message or for atomic message). By combining all the information, a correct prediction can be made. We note that many of these information are required to check whether an MPI Accumulate can directly leverage native network atomics or requires active message. For simple Put/Get, only (2) and (3) are essential. Nevertheless, to disable the active message progress on a remote process, we have to check all information. Second, a process requires all the other processes in the window to

share their local information in advance before any communication occurs (ideally at window creation). The network-provided information is usually identical on all processes; thus each process can simply query it locally. The user information, however, may vary on each process. More important, the user has to specify such information before communication occurs, likely through MPI info hints. The hint may become significantly complex if the user program involves several different combinations of (1–3) in a window. Clearly, such an approach is impractical for MPI users.

Alternatively, we apply an engineer approach to resolve this issue. Specifically, we assume that all RMA operations can be handled directly by a network library when starting an MPI program. Thus, we trigger the active message progress with a very low frequency. For instance, we trigger the progress once only every 100 times RMA flush calls are made. This allows MPICH to catch any unexpected incoming active message. Once an active message is received, we then revert to normal frequency (i.e., trigger progress at least once at each RMA flush call). The mechanism exposes two MPI control variables (CVAR) for flexible user adjustment. Specifically, we define `MPIR_CVAR_CH4_RMA_ENABLE_DYNAMIC_AM_PROGRESS` to enable or disable the optimization (`false` by default) and `MPIR_CVAR_CH4_RMA_AM_PROGRESS_LOW_FREQ_INTERVAL` to set the interval of progress polling at the low frequency mode. The former is `true` at `shmem_init` in OSHMPI. We expect that the active message progress is always triggered with the low frequency for OpenSHMEM programs because all OpenSHMEM RMA and atomic operations can be handled via native network operations.

For OFI/UCX Internal Progress: The first step of each MPI flush call in Table 4 already triggers necessary network progress for RMA data transfer. Thus, it is unnecessary to make MPI full progress again for such a purpose.

To summarize, the MPI full progress can be safely skipped in both `MPI_Win_flush_local` and `MPI_Win_flush_all`, thus significantly reducing overhead for both `shmem_putmem` and `shmem_quiet` functions in OSHMPI/MPICH.

5.5 Reducing Synchronization in OSHMPI

Although we have eliminated the MPI full progress step in the flush calls, the overhead of an `MPI_Win_flush_local` or `MPI_Win_flush_all` is still expensive because the first step of each call always makes a call to network synchronization. We note that such synchronization is required to complete a network data transfer even in OpenSHMEM native implementations. In OSHMPI, however, we may unnecessarily trigger the synchronization call (i.e., `MPI_Win_flush_all`) twice at `shmem_quiet`, one for the window of symmetric heap and the other for that of global/static data objects. If only one of the windows contains outstanding operations, we need trigger the synchronization call only on that "active" window. Thus, we set a flag for each window in OSHMPI to keep track of the existence of outstanding operations. The same optimization applies to `MPI_Win_sync`.

5.6 Other Implementation-Specific Optimizations

The instruction analysis also provides useful guidance for us to reduce unnecessary instructions at the performance critical paths. These optimizations are MPICH-specific. Specifically, we apply four optimizations in MPICH: (1) eliminating repeated `MPI_PROC_NULL` check,[8] (2) removing unused signal checks in MPI full progress, (3) statically triggering subprogressing hooks (e.g., for collectives) instead of dynamic function pointer access, and (4) optimizing the hash search for checking the existence of target active message objects.

6 Evaluation

In this section we evaluate the performance of OSHMPI/MPICH on the Argonne Bebop cluster.[9] Each Bebop node uses two sockets of the 18-core Intel Xeon E5-2695 v4 processor (Broadwell) and is connected with the Intel Omni-Path interconnect. We used the Intel compiler (version 17.0.4) and libfabric-1.7.0 as the OFI network low-level library. We configured OSHMPI with the `ch4:ofi` configuration of MPICH and compared it with the SOS 1.4.2 release.[10] We linked both the MPICH and SOS libraries with the same underlying libfabric library. We also measured the OFI native Put latency by using a customized version of the `fi_pingpong` test included the libfaric official release. It mimics the data transfer pattern of `osu_oshm_put`. We use it to demonstrate the ideal performance of OFI-based data transfer. For each measurement we collected the execution time of 10 runs and report the average and the standard deviation (shown as error bars in the graphs). The error bars are very small for most results (less than 5%) and thus can barely be seen.

6.1 Instruction Analysis

We first break down the instruction counts of optimized `shmem_putmem` and `shmem_quiet` following the same approach as that used in Section 4. We statically linked the latency program against the OSHMPI and MPICH libraries with IPO link-time optimization. We explicitly disabled inline functions in the latency program layer to make a fair comparison with SOS. In other words, both OSHMPI and SOS are unaware of the variable values defined in the latency program layer

[8] `MPI_PROC_NULL` is an MPI predefined dummy process rank. An MPI RMA operation using `MPI_PROC_NULL` as the remote rank is a no-op.

[9] *https://www.lcrc.anl.gov/systems/resources/bebop*

[10] We have made the following changes in SOS to ensure a fair comparison with OSHMPI/MPICH: (1) disable the OFI domain thread (set domain attribute `data_progress = FI_PROGRESS_MANUAL` at `shmem_init`) to reduce latency overhead at large data transfer; (2) reduce frequent calls to expensive `fi_cntr_wait` at `shmem_quiet`; and (3) disable bounce buffer optimization in the latency test because it increases latency overhead for medium data sizes (set environment variable `SHMEM_BOUNCE_SIZE=0`).

(e.g., `nelem`) and thus treat them as variables at compile time. Consequently, only the information defined in OSHMPI (e.g., datatype) is passed into MPICH via link-time inlining.

The Optimized Count columns in Tables 1 and 2 summarize the instruction counts generated by `shmem_putmem` and `shmem_quiet`, respectively. Roughly speaking, the instruction overhead of `shmem_putmem` is reduced to 93 and the overhead of `shmem_quiet` is reduced to 44 with all these optimizations. We especially highlight the following instruction-saving aspects. First, thanks to IPO link-time optimization, the instruction count of all cross-library overheads (e.g., calling overhead of `MPI_Put` and `Flush_local` in `shmem_putmem`) are now reduced to zero. It also helped eliminate the datatype decoding overhead (step (4) of `shmem_putmem`) with an embedded datatype handle as described in Section 5.1. We note that IPO allows more instructions to be saved throughout the implementation (i.e., partially reduced instructions in steps (5–6) and (9) of `shmem_putmem`). We omit the discussion in this paper. Second, the optimization of fast window attribute access reduces the network address translation (step (3)) to only 5 instructions, matching with the instructions demonstrated in Figure 3b. Third, the instructions for computing `dest_vaddr` (step (7)) are optimized via the *abs* extension of MPI RMA functions. Fourth, we emphasize the highly optimized progress routines (step (11) in `shmem_putmem` and step (3) in `shmem_quiet`). Because we avoid unnecessary polling for non-RMA routines and utilize a dynamic approach to deal with the active message challenge (see detail in Section 5.4) together with implementation code refactoring (see Section 5.6), the optimized version now consumes only 2 instructions for the MPI full progress step. Fifth, as shown in Table 2, skipping unnecessary window synchronization (see Section 5.5) is straightforward and effective. When only either symmetric heap or global/static variable is used for communication, such an optimization can reduce 269 instructions including expensive low-level network synchronization calls. The remaining 3 instructions are used to check the window flag.

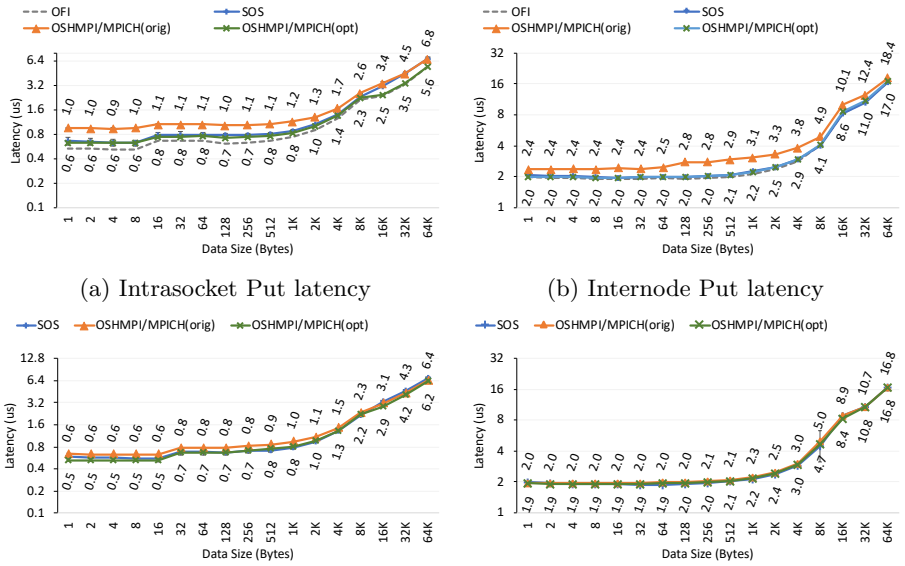
6.2 Latency

We next evaluated the latency of optimized OSHMPI/MPICH. We used the `osu_oshm_put` and `osu_oshm_get` tests from the OSU microbenchmark suite (version 5.6.2) to measure the latency of Put and Get, respectively.

Figures 6a and 6b report the Put latency. For both the intrasocket and internode results, we also include the OFI native Put latency (denoted by OFI) to indicate the ideal performance. The original OSHMPI/MPICH latency has a clear gap between that of SOS and OFI, It consumes about 1 μs latency for a 1-byte message, whereas OFI and SOS require only 0.54 μs and 0.66 μs, respectively. The optimized version significantly reduces the cost. The achieved latency is almost identical to that of SOS. The improved latency is mainly contributed by the optimization of MPI full progress in MPICH and reduced window synchronization in OSHMPI. Similar observations can be made in the internode results. Our optimizations reduce 0.4 μs latency of OSHMPI/MPICH with a

1-byte message. The achieved latency is the same as that of SOS and OFI. For other message sizes, we observe a similar trend.

The Get latency reported in Figures 6c and 6d shows less gap between the original OSHMPI/MPICH and other implementations. Nevertheless, the optimized OSHMPI/MPICH can achieve a lower latency that is the same as that of SOS.



(a) Intrasocket Put latency

(b) Internode Put latency

(c) Intrasocket Get latency

(d) Internode Get latency

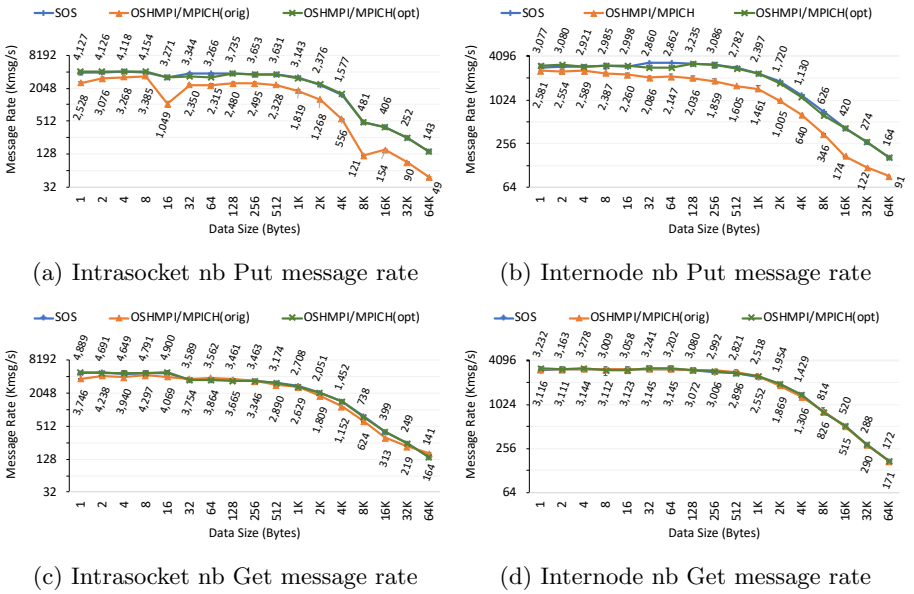
Fig. 6: Latency evaluation on Bebop. The top and bottom labeled numbers are the latency of OSHMPI/MPICH(orig) and OSHMPI/MPICH(opt), respectively.

6.3 Message Rate

The third set of experiments focus on message rate. We used the `osu_oshm_put_mr_nb` and `osu_oshm_get_mr_nb` tests from the OSU microbenchmark suite. The communication pattern involves multiple calls to the nonblocking `shmem_putmem_nbi` (`shmem_getmem_nbi` for the Get test) followed by a call to `shmem_quiet`. Thus, these tests present the overhead of the lightweight nonblocking RMA calls.

Figures 7a and 7b report the message rate of nonblocking Put. We observe that the optimized OSHMPI/MPICH significantly improves the message rate of Put. It achieves an average improvement of 2.1x for intrasocket Put with varying data size and 1.6x for internode Put. Since OSHMPI `shmem_putmem_nbi` internally contains only an `MPI_Put`, we confirm that the improvement is contributed by the fast path optimizations (i.e., datatype decoding, fast window attribute access, and RMA abs extension). The optimized message rate is almost identical to that of SOS.

We observe a similar trend with nonblocking Get. However, the gap between the original OSHMPI/MPICH and SOS is much less than that of nonblocking Put. Thus, the improvement ratio is reduced. We report an average improvement of 10.3% for intrasocket Get with varying data size and 7.3% for internode Get.



(a) Intrasocket nb Put message rate

(b) Internode nb Put message rate

(c) Intrasocket nb Get message rate

(d) Internode nb Get message rate

Fig. 7: Message rate evaluation on Bebop. The top and bottom labeled numbers are the latency of OSHMPI/MPICH(orig) and OSHMPI/MPICH(opt), respectively.

7 Conclusion and Future Work

OpenSHMEM and MPI are two widely used communication models for distributed-memory systems. The OpenSHMEM functionalities can be implemented by using MPI. For instance, mapping the essential OpenSHMEM RMA operations to MPI Put/Get with appropriate MPI window synchronization is straightforward. However, a general belief in the community is that such an OpenSHMEM over MPI implementation will not deliver the same level of performance as that of native OpenSHMEM implementations. This is mainly caused by the additional instructions generated for OpenSHMEM to MPI mapping. Therefore, OpenSHMEM over MPI is often used only as a short-term solution for platforms where a native OpenSHMEM is not yet available. In this paper we demonstrated that OpenSHMEM over MPI can actually become a performance contender. We showcased the OSHMPI and MPICH implementations and focused on the essential RMA routines. We first made a thorough analysis to understand the instruction

overhead generated in the RMA critical path of the OSHMPI and MPICH layers. Based on the observed performance bottlenecks, we further optimized several key aspects including datatype decoding, MPI window attribute access, virtual destination address translation, and the expensive MPI progress. Our evaluation was performed on an Intel Broadwell cluster with the Intel Omni-Path interconnect. We compared the optimized OSHMPI/MPICH with the native OpenSHMEM implementation on that platform. We concluded that the optimized OSHMPI/MPICH can deliver the same level of performance in both latency and message rate as that of a native OpenSHMEM implementation.

Although the analysis and optimizations focused on the RMA routines, most can be easily adapted also for other OpenSHMEM routines. As future work, we plan to optimize atomics and collective routines in the OSHMPI and MPICH environment. Furthermore, we note that our performance evaluation used only microbenchmarks on the Intel Omni-Path platform. We therefore also plan to look into the performance of miniapplications and evaluate other platforms.

Acknowledgment

This research was supported by the United States Department of Defense (DoD). This material was based upon work supported by the United States Department of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resource for this paper was provided by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory.

References

1. ARMCI-MPI. *https://github.com/pmodels/armci-mpi*.
2. Mellanox ScalableSHMEM User Manual. Technical report, Mellanox Technologies, Ltd.
3. Open MPI: Open Source High Performance Computing. *https://www.open-mpi.org/*.
4. Essential Guide to Distributed Memory Coarray Fortran with the Intel Fortran Compiler for Linux. *https://software.intel.com/content/www/us/en/develop/articles/distributed-memory-coarray-fortran-with-the-intel-fortran-compiler-for-linux-essential.html*, 2018.
5. Ray Barriuso and Allan Knies. SHMEM User's Guide for C. Technical report, Cray Research Inc, 1994.
6. Dan Bonachea and Jason Duell. Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations. *International Journal of High Performance Computing and Networking*, 1(1–3):91—-99, August 2004.
7. James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 739–750, 2012.

8. Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. OpenCoarrays: Open-Source Transport Layers Supporting Coarray Fortran Compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, New York, NY, USA, 2014. Association for Computing Machinery.
9. K. Fürlinger, C. Glass, J. Gracia, A. Knüpfer, J. Tao, Denis Hünich, K. Idrees, Matthias Maiterth, Yousri Mhedheb, and H. Zhou. DASH: Data Structures and Algorithms with Support for Hierarchical Locality. In *Euro-Par Workshops*, 2014.
10. Yanfei Guo, Charles J. Archer, Michael Blocksome, Scott Parker, Wesley Bland, Ken Raffenetti, and Pavan Balaji. Memory Compression Techniques for Network Address Management in MPI. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1008–1017, 2017.
11. Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In Stephen Poole, Oscar Hernandez, and Pavel Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 44–58, Cham, 2014. Springer International Publishing.
12. Jithin Jose, Krishna Kandalla, Miao Luo, and Dhabaleswar K. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *2012 41st International Conference on Parallel Processing*, pages 219–228, 2012.
13. Naveen Namashivayam, Bob Cernohous, Dan Pou, and Mark Pagel. Introducing Cray OpenSHMEMX – A Modular Multi-communication Layer OpenSHMEM Implementation. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, pages 41–55, Cham, 2019. Springer International Publishing.
14. David Ozog, Md. Wasi ur Rahman, Kayla Seager, and James Dinan. Design and Optimization of OpenSHMEM 1.4 for the Intel Omni-Path Fabric 100 Series. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, pages 22–40, Cham, 2019. Springer International Publishing.
15. Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A Kendall. A Generalized Portable SHMEM Library for High Performance Computing. Technical report, Ames Lab., Ames, IA (US), 2000.
16. Ken Raffenetti, Abdelhalim Amer, Lena Oden, Charles Archer, Wesley Bland, Hajime Fujita, Yanfei Guo, Tomislav Janjusic, Dmitry Durnov, Michael Blocksome, Min Si, Sangmin Seo, Akhil Langer, Gengbin Zheng, Masamichi Takagi, Paul Coffman, Jithin Jose, Sayantan Sur, Alexander Sannikov, Sergey Oblomov, Michael Chuvelev, Masayuki Hatanaka, Xin Zhao, Paul Fischer, Thilina Rathnayake, Matt Otten, Misun Min, and Pavan Balaji. Why Is MPI So Slow?: Analyzing the Fundamental Limits in Implementing MPI-3.1. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 62:1–62:12, New York, NY, USA, 2017. ACM.
17. Chaoran Yang, Wesley Bland, John Mellor-Crummey, and Pavan Balaji. Portable, MPI-Interoperable Coarray Fortran. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 81—-92, New York, NY, USA, 2014. Association for Computing Machinery.
18. Huan Zhou, Kamran Idrees, and José Gracia. Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems. In *Euro-Par 2015: Parallel Processing*, pages 373–384, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.