Framework for Scalable Intra-Node Collective Operations using Shared Memory

Surabhi Jain, Rashid Kaleem, Marc Gamell Balmana, Akhil Langer,
Dmitry Durnov, Alexander Sannikov, and Maria Garzaran
Intel Corporation
Email: {surabhi.jain, rashid.kaleem, marc.gamell.balmana, akhil.langer,
dmitry.durnov, alexander.sannikov, maria.garzaran} @intel.com

***Abstract*—Collective operations are used in MPI programs to express common communication patterns, collective computations, or synchronization. In many collectives, such as MPI_Allreduce, the *intra*-node component of the collective lies on the critical path, as the *inter*-node communication cannot start until the *intra*-node component has completed. With increasing number of core counts in each node, *intra*-node optimizations that leverage shared memory become more important.**

In this paper, we focus on the performance benefit of optimizing intra-node collectives using POSIX shared memory for synchronization and data sharing. We implement several collectives using basic primitives or steps as building blocks. Key components of our implementation include a dedicated *intra*-node collectives layer, careful layout of the data structures, as well as optimizations to exploit the memory hierarchy to balance parallelism and latencies of data movement. A comparison of our implementation on top of MPICH shows significant performance speedups with respect to the original MPICH implementation, MVAPICH, and OpenMPI.

I. Introduction

Collective operations are used in MPI programs to express common communication patterns, collective computation operations (e.g. allreduce) or synchronizations (e.g. barrier) [1]. For each collective operation, different algorithms can be used depending on the message size and the number of the ranks participating in the collective [2]. Algorithms can be adapted to better exploit the underlying network topology and to increase concurrency [3], [4], [5].

Some of the largest supercomputers, like the Oakforest-PACS or Stampede-2 [6], are built with many-core nodes, such as the Intel® Xeon Phi™. On these systems, applications usually run several MPI ranks per node and therefore the intra-node component of the collective can significantly impact the performance of the overall collective operation. In fact, in collectives such as reduce, allreduce, or barrier, the intra-node component of the collective is on the critical path. With more cores per node, the latency along the critical path becomes detrimental to the overall collective performance. Thus, optimizations that leverage intra-node shared memory become increasingly crucial [7], [8], [9]. Thus, its performance is critical to the overall performance of the collective. Current MPI implementations optimize intra-node component of the collective in three different ways. *First*, using intra-node point to point communication, which uses shared memory as a transport layer inside a node. *Second*, allocating a shared memory space that ranks in the same node can access and that can be used for the communication across ranks. This is necessary because MPI ranks are usually implemented as processes that have separate memory space. *Third*, using zero-memory copy mechanisms, such as those provided by XPMEM [10], [11], where a rank exposes its memory space, avoiding the copy into to the shared memory space.

In this paper, we optimize the intra-node component of the collectives using shared memory. Zero copy mechanisms have the potential to provide higher performance benefit, specially for large messages, but this paper focuses on shared memory as it is a more general mechanism and support for zero-copy mechanism may not always be provided by the kernel of the target system. Our implementation is done on top of MPICH [12]. We use a *release/gather* approach for synchronization between ranks that was inspired by the *gather/release* steps used inside the OpenMP implementation in the LLVM compiler [13] to implement barrier, and extended it to support broadcast and reduce, as well as synchronization between processes that do not share the memory space.

We compare the performance of our proposed scheme against the current implementation of MPICH with two different devices, ch3 and ch4 [12], MVAPICH [14], and OpenMPI [15]. The two main features that distinguish our design are: *a)* a dedicated shared memory layer optimized for collectives based on the *release/gather* steps used as building blocks to implement collectives and *b)* a topology aware design of intra-node trees that takes into account the memory hierarchy of the node and minimizes data movement, while increasing concurrency of the collective operation. MVAPICH also uses a dedicated shared memory layer, but OpenMPI and MPICH use the same code path as the point-to-point communication between ranks on the same node. This forces OpenMPI and MPICH to implement MPI_Bcast() with point to point messages from each parent to each child. With our approach, the parent rank performs a single copy to a shared memory buffer and all the children ranks copy the data out in parallel. To the best of our knowledge, neither OpenMPI, MPICH or MVAPICH use topology aware trees for intra-node collectives.

We focus on the optimization of three MPI collectives: MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(), although other collectives can be implemented using the building

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

blocks discussed in this paper. Our experimental results show that when running with 40 ranks (one rank per core) on Intel® Skylake, with respect to MVAPICH our approach is up-to 1.32×, 14.27×, and 3.17× faster for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce() and 1.22×, 4.43×, and 1.80× faster on average, respectively. With respect to Open-MPI our approach is up-to 5.96×, 13.37×, and 7.71× faster for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce() and , 4.00×, 4.71×, and 3.38× faster on average, respectively. MPICH with the new ch4 device (discussed later) is up-to 3.96×, 6.32×, and 4.17× faster and 2.99×, 3.23×, and 2.72×, on average, for the same three collectives.

The paper is organized as follows. Section II presents an overview of our approach; Section III discusses potential optimizations; Section IV discusses some issues about memory consistency and atomicity; Section V presents our environmental setup, while section VI shows our experimental results; Section VII discusses related work; Finally, Section VIII concludes.

II. Overview

In this section, we focus on the baseline implementation of broadcast and reduce and then discuss how allreduce can be implemented using the same basic building blocks.

Our approach uses shared memory buffers that can be accessed by all the ranks participating in the collective and two synchronization flags per rank. One of the flags is used when a rank copies its data to the shared buffer, to notify that new data is available. The other flag is used when a rank copies the data out of the shared buffer, to signal that it has read the contents of the buffer and the buffer can be reused. We present a detailed description of the flags and the buffers in detail later in the Section.

Our collective implementation breaks down the collective into an intra-node collective and an inter-node one. To support them, the communicator used by the collective call is split into two different communicators, an intra-node communicator, which contains all the ranks in the node and is used by the intra-node collective and an inter-node communicator, that contains a lead rank per node, which in our implementation is rank 0 in the intra-node communicator and is used by the inter-node collective. From now on, references to the communicator, unless otherwise specified, refer to the intra-node communicator.

Our intra-node collective implementations are based on tree algorithms. A tree is generated for each communicator with all the ranks as its nodes. Each rank only stores its children and parent rank. In this section, we assume a flat tree or a tree where the branching factor is large enough that all the ranks are the children of the root node (tree optimizations are discussed in Section III-A). We first describe the data structure as well as its memory layout in shared memory and then proceed to describe the two primitive operations (*Release* and *Gather*) that are used to implement the collectives.

A. Memory layout and usage

Our implementation uses POSIX-compliant Unix mmap() system call to allocate memory that is shared across the multiple ranks or processes within the node. This memory is allocated lazily the first time ranks in a communicator call a collective. A pictorial view of the layout of the shared memory region is shown in Figure 1. The region contains memory space for synchronization flags as well as intermediate storage for broadcast and reduce operations. For each region, a single buffer of $8KB$ for broadcast and a per-rank buffer of $8KB$ for reduce operations is maintained. The buffer sizes can be changed.

Each rank maintains a set of private variables per-communicator to track its own progress in each step. These include the pvt_release_state and the pvt_gather_state which are used in *release* and *gather* phases, respectively. Each communicator also maintains a set of shared flags, one per rank, used for *synchronization*. Gather flags (sh_gather_flag) are used for the child to communicate to its parent i.e. *all children gather for its parent*, while release flags (sh_release_flags) are used for the parent to communicate with its children i.e. *a parent releases its children*. Each child rank writes to its own sh_gather_flag, while the parent reads the sh_gather_flag of its children. For a sh_release_flag, only the parent writes to it while the children read it. To avoid false sharing, each flag uses a single cache line. Given the single-writer to these flags, loads and stores of these flags do not require a critical region, but may need memory fences, as discussed in Section IV.

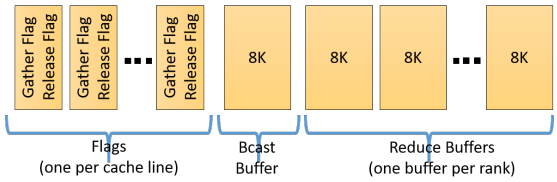


Fig. 1: Memory Layout

B. Release and Gather primitives

A collective is implemented with two synchronization steps, a *release* and a *gather* step. The pseudocode for the *release* step is shown in Figure 2. Upon entering the *release* step, each rank increments its private pvt_release_state(③). The parent (root) writes the value of pvt_release_state to the shared sh_release_flag of each child (non-root)(④-⑤). Each child spin waits on its own shared sh_release_flag until the value of the shared sh_release_flag is equal to that of its private pvt_release_state(⑥-⑦). Once the shared sh_release_flag and private pvt_release_state are equal, the child knows the parent has arrived at the corresponding *release* step(⑧-⑨). The pseudocode has additional details for broadcast operations, which are explained later.

The pseudocode for the *gather* step is shown in Figure 3. Upon entering the *gather* step, each rank increments the value of its private pvt_gather_state(③). The leaf

```
1  if (broadcast && root)
2    copy data from user_buffer to sh_bcast_buffer;
3  pvt_release_state++;
4  for (i=0;i<num_children;i++)
5       *child[i].sh_release_flag = pvt_release_state;
6  if (non_root)
7    wait_until(*sh_release_flag == pvt_release_state);
8  if (broadcast && non_root)
9    copy data from sh_bcast_buffer to user_buffer
```

Fig. 2: Release step

ranks write the value of the pvt_gather_state to its shared sh_gather_flag(⑨). The parent rank waits until each of its children has updated its shared sh_gather_flag, at which point it can update its own shared sh_gather_flag to indicate to its parent that all of its children have progressed in the collective(④-⑥). The pseudocode also includes details for reduce operations, which will be described later.

```
1  if (reduce)
2    copy data from user_buffer to sh_reduce_buffer;
3  pvt_gather_state++;
4  for (i=0;i<num_children;i++){
5    wait_until(*child[i].sh_gather_flag ==
6                          pvt_gather_state);
7    if (reduce) reduce_operation();
8  }
9  *sh_gather_flag = pvt_gather_state;
```

Fig. 3: Gather step

Next, we discuss how the *release* and *gather* steps are used to implement broadcast (Section II-C), reduce (Section II-D) and allreduce (Section II-E).

C. Implementing broadcast

A broadcast is implemented using a *release* followed by a *gather* step. For broadcast, during the *release* step, the parent copies the data to broadcast into the buffer space reserved for that purpose (see Figure 1) and updates the children's release flag sh_release_flag with its pvt_release_state value. Child ranks read the shared sh_release_flag and when its value is equal to the value of its private pvt_release_state, the child rank can copy out the data from the broadcast buffer. After the *release* step, in the *gather* step the children ranks signal the parent that they have completed copying the data by updating the sh_gather_flag with the value of its pvt_gather_state. The parent rank waits for the gather flag of all its children to have the appropriate value before it can continue. A child rank waits for the parent rank to copy the data in the broadcast buffer, copies out the data, updates sh_release_flag, and it leaves. The parent rank can copy the data into the broadcast buffer and update the sh_release_flag as soon as it arrives, but needs to wait for all the children to update their sh_release_flag before it can leave. This guarantees that when the parent copies the data into the system buffer during the *release* step of the next broadcast, all the children have already copied the data out of the system buffer. This synchronization can be hidden if there are additional buffers, as described in the optimizations in Section III.

D. Implementing reduce

A reduce operation is also implemented with a *release* followed by a *gather* step. In this case, during the *release* step, the parent updates the children's sh_release_flag with its pvt_release_state to inform them that is has already copied the data from the corresponding reduce buffer so that the child rank can reuse the buffer. During the *gather* step, each child copies the data to be reduced to its own reduce buffer space (see Figure 1) and then writes the value of its pvt_gather_state to the sh_gather_flag. The parent rank reads each child sh_gather_flag to find out if the data to be reduced have been copied into the reduce buffer. When the sh_gather_flag has the appropriate value, the parent can copy-out the data from the child buffer space and perform the reduction operation.

Figure 4 shows a pictorial view of the *gather* and *release* steps for a reduction operation, where only the parent rank (rank X) and a child rank (rank Y) are involved.

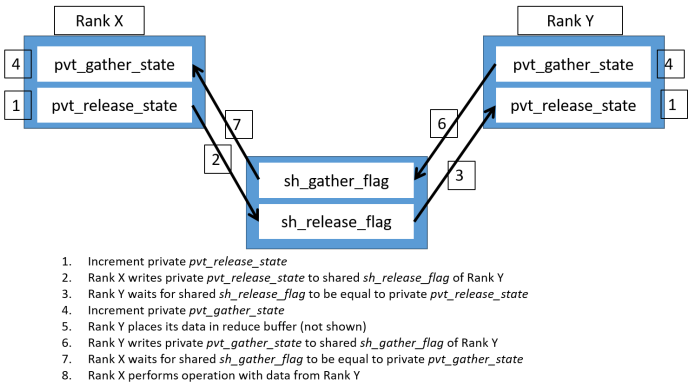


Fig. 4: Pictorial view of release and gather step in reduce

E. Implementing allreduce

An implementation of allreduce with ranks in multiple nodes requires *three* steps. *First*, a reduce operation to reduce the data across the ranks on the same node. At the end of this step, one rank per node, which we call lead rank for that node, has the result of the intra-node reduction. This step can run in parallel on all the nodes that have ranks participating in the reduction. *Second*, an inter-node allreduce, where the lead ranks perform an allreduce (this stage can use any algorithm and is not part of the optimizations studied in this paper). After this second step, the lead rank in each node has the final result of the reduction. *Third*, a broadcast operation, where the lead rank in each node broadcasts the final results to all the ranks inside its node. This step can be done in parallel across all the nodes. Notice that allreduce can also be implemented using other algorithms such as recursive doubling, which we do not consider in this paper. Recursive doubling is a good algorithm for the inter-node allreduce as all the lead ranks can have the result of the reduction in $log_2 n$ steps, but requires

additional traffic and does not provide any benefit for the intra-node reduce, where only the lead rank needs the intra-node reduced value.

III. Optimizations

In this section, we describe some of the optimizations that we have studied: the usage of trees to propagate flags and data (Section III-A), *read from parent flag* on the *release* step (Section III-B), *pipelining* for large messages (Section III-C), *data inlining* (Section III-D) and *data copy* optimization (Section III-E).

A. Trees to propagate flags and data

This section discusses the different trees we considered to propagate flags and data. Specifically, we discuss topology-unaware trees, topology-aware trees, right/left skewed trees, and the best trees for broadcast and reduce.

The discussion in this section assumes that the root of the tree is always rank 0. For all nodes, except for the node containing the root of the collective, rank 0 can be chosen as the lead rank to perform the inter-node communication. On the root node (the node that contains the root of the collective), if rank 0 is not the root, the root can send a point-to-point message with the data to rank 0. While this point-to-point message can be removed, our performance experiments, with the root not being rank 0, show that the performance difference is insignificant, hence we did not consider the optimization for our final evaluation.

1) Topology-unaware-trees: Section II assumes a flat tree, a tree where the branching factor is large enough that all the ranks are the children of the root rank. The problem with a flat tree is that the root has to read the flags from all the children during the *gather* step and write them during the *release* step. To decrease the overheads of the root rank, we use a tree that is created at the same time the shared memory is allocated.

When a tree is used, the *release/gather* flags are propagated level by level from the root to the leaf ranks or vice versa. The tree changes the data movement pattern in a reduce operation but does not affect the data movement in a broadcast operation. With a flat tree, there is no concurrency on the reduction operation, because the root has to perform all the reductions sequentially. However, with a tree, all the parents that have leaf children can start the reduction concurrently. In our implementation, we have considered K-ary and K-nomial trees, where the value of K is a tuning parameter that can be selected with a configuration variable.

2) Topology aware trees: While a K-nomial or K-ary tree does increase the parallelism, it assumes that the cost of data movement is uniform, which is not the case on current architectures that have a deep memory hierarchy and multiple sockets. Thus, an arbitrary tree might result in many reduction operations taking place across sockets, which can result in increased latency and hurt the overall performance of the collective. A topology aware tree takes the machine topology into account by partitioning the ranks based on their bindings.

To describe the different alternatives explored, let us consider a hypothetical machine with 5 sockets, where each socket has 4 cores, as shown in Figure 5. Furthermore, assume that each rank is bound to a single core and rank 0 is the root of the collective. In this example, the topology aware tree considers two kinds of trees: *per-socket* trees and *socket-leader trees*. The per-socket trees describe how a collective communicates within a socket. The assumption is that within the socket the data movement and synchronization cost is uniform across all ranks. A K-ary or K-nomial tree can be generated for each socket, which will be used for the collective within that socket. An example is shown in Figure 6, with branching factor of 3.

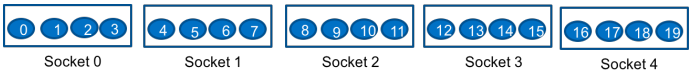


Fig. 5: Node with five sockets and four cores per socket.

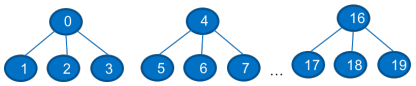


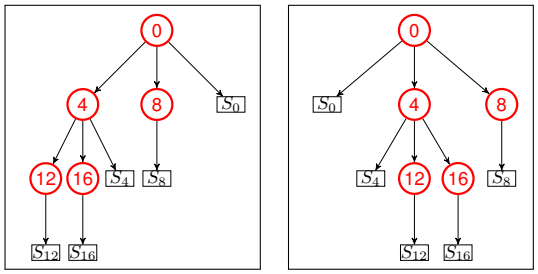
Fig. 6: Per-socket K-ary tree with branching factor 3 for the machine in Figure 5.

The root of the per-socket tree is the lead rank and is used in the socket-leader tree, which describes the communication *across* sockets. The per-socket and the socket-leader trees are two independent trees. Next, we define how to connect these two trees (that only share the socket leaders) and consider the performance implications.

socket-leader-first: In the socket-leader-first approach, we generate a complete tree by appending the per-socket sub-trees to the socket-leader tree *after* the socket leader ranks. An example is shown in Figure 7(a), where socket leaders are represented with red circular nodes, whereas the per-socket trees (shown in Figure 6) are shown in black rectangles. For this specific instance, we chose the socket-leader tree to be a K-ary tree with $K = 2$. The per-socket trees are unconstrained and can be configured independently. Reduction proceeds upward, from the leaves to the root, with parent ranks going over its children in sequential order from left to right. Thus, the socket-leaders are processed first followed by the per-socket tree members. For the example in Figure 7(a), the root rank 0 will process socket leaders 4 and 8 first, and then process its own per-socket sub-tree S_0.

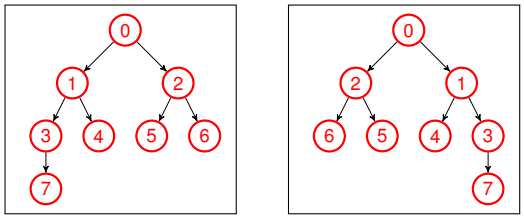
socket-leader-last: The other option to merge the socket-leader trees and per-socket trees is to append the per-socket sub-trees *before* the socket-leader trees. An example is shown in Figure 7(b), where rank 0 has a sub-tree S_0 consisting of ranks $\{1, 2, 3\}$ in some tree configuration. The socket-leader tree has ranks 4 and 8 as children of rank 0. In the socket-leader-last configuration, the ranks 4 and 8 appear *after* 0's socket-tree children.

The socket-leader-first trees seem more appropriate for the release path. On the *release* step, flag updates flow from



(a) socket-leader-first (b) socket-leader-last

Fig. 7: Different topology aware trees. Red circular nodes represent socket leaders, and black rectangular boxes represent per-socket trees.



(a) Left skewed tree (b) Right skewed tree

Fig. 8: K-ary tree ($K = 2$) with different skews.

the root to the leaves, starting with the children on the left. Thus, with the socket-leader-first tree, we start sending the data through the critical path of the tree to the ranks that are farther away, minimizing the time for the last rank to receive the flags. However, on the *gather* step where data reduction is performed, data propagate upwards from the leaves to the root and the socket-leader-first tree limits concurrency. The reason is that parent nodes start reducing the children on its left and need to wait for the sub-trees to finish the reduction before it itself can start to work. As an example, take rank 0 in the socket-leader-first tree in Figure 7(a), it needs to wait for the data on all its sub-trees to be reduced before it can start do any useful work. However, with the socket-leader-last tree in Figure 7(b), rank 0 can start to reduce the data from its sub-tree, while other per-socket sub-trees perform their reductions.

3) Right/Left Skewed trees: So far we have considered the default ordering of trees, such as K-ary and K-nomial, where ranks are placed in increasing order. This leads to a tree which fills up the left side first. We refer to such a tree as a left-skewed tree. Consider a k-ary tree with $K = 2$ for 8 ranks. This will generate the tree shown on Figure 8(a). As we can see, the rank 7 appears as the left-most leaf. If, however, we reverse the order of children for each rank, we get a tree that mirrors the left-skewed tree, and that we call a right-skewed tree. The right-skewed tree is shown Figure 8(b), where 7 appears as the right-most leaf. We expect the left-skewed trees to perform better for the *release* step and the right-skewed to perform better for the *gather* step. The reasons are similar to those discussed before for the socket-leader-first and socket-leader-last approaches.

4) Different trees for broadcast *and* reduce*:* To better adapt to the different communication patterns, the tree for broadcast is different from the tree for reduce. For the broadcast, the updates to flags are propagated *downwards* during the *release* step, while for the reduce, the data and updates to flags propagate *upwards* during the *gather* step.

With this approach, the same tree is used for the *gather* and *release* step of a collective. While it is possible to use different trees in the *gather* and *release* steps for a given collective, this is an optimization that we have not explored. Also, the optimization explained in the next section, improves the performance of the *release* step.

B. Read from parent flag on the release step

In our baseline, we have assumed that in the *release* step, the parent rank writes to the shared release flag of each child rank. This adds some overhead for the parent, especially when the branching factor of the tree is high. Thus, an optimization that we have explored is to have the parent rank write to its own shared flag and have all the children read from the same shared flag. This decreases the overhead for each parent rank, as it only needs to perform one store versus branching-factor number of stores. In addition, when the children ranks share the cache, the approach can decrease latency of the load operation, as one rank can prefetch the cache line for the other ranks. This approach cannot be applied to the gather flags, because in this case each child needs to communicate to the parent rank, so we still need a separate flag per rank.

C. Multiple Buffers and Data pipelining

The baseline approach in Section II has limitations for large and small messages. For messages larger than the allocated buffer space, the approach does not work. For small messages, a single buffer forces a synchronization during the *gather* step of the broadcast and during the *release* step of the reduce. With a single buffer, on the *gather* step of the broadcast the root rank has to wait for all its children to copy the data out from the broadcast buffer before it can proceed. On the *release* step of the reduce the children have to wait for the root to signal that it has copied the data out before they can proceed.

A solution for both the problems is to split a buffer into several chunks and use pipelining. For large messages, this allows overlapping the copy-in of one chunk with the copy-out of a previously copied chunk. Figure 9 shows one such example of a broadcast where rank Y (the root) is copying-in chunk2, while the other ranks are copying from the previously copied chunks (chunk0 and chunk1). As the figure shows, each rank can be copying from different chunks. A similar approach, but in a different direction, can be assumed for reduce, where the children ranks will be copying the data in the buffer and the parent will be copying-out each chunk from each child rank. This approach also hides the synchronization for both broadcast and reduce, when enough buffers are available.

The implementation requires several *release-gather* steps ($\lceil message\_size/chunk\_size \rceil$). It uses the values of the

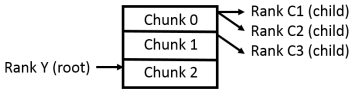


Fig. 9: Pipelining approach

private flags and the number of chunks to determine the appropriate chunk to use. The difference between the values of the private and the shared flags is used to determine when a rank needs to wait to avoid overwriting a buffer that has not been copied out yet. For the broadcast, the release flags are used and for the reduce, the gather flags are used.

An additional optimization has been added to further decrease synchronization overheads. In broadcast, the root can skip the *gather* step if it knows it has a buffer for the next broadcast call. To do that, the root can keep track of the minimum value of the sh_gather_flags across all its children last time it checked. A similar optimization can be applied by the children ranks on the release step of the reduce. This optimization is enabled by default on our pipelining implementation, although we did not see a noticeable performance difference between enabling and disabling it.

D. Data inlining for small messages

This optimization combines the data and flags in a single cache line when the message to be sent is very small. Since we reserve a whole cache line for each flag (64-bytes), there is extra space that can be used for the data when the message is small (less than 60 bytes assuming a 32-bit flag). The advantage of this approach is the decreased number of cache misses. The drawback is that it cannot be combined with the pipelining approach, as we only have space for one flag. Combined with the fact that it can only be used for small messages, this optimization has limited applicability.

E. Data copy optimizations

This optimization avoids an extra data copy from the shared memory buffer to the receive or user buffer during a reduction operation in the root rank and on the lead rank of the inter-node communicator. In our implementation, the reduction is performed during the *gather* step, where each rank first copies the data from the send buffer to its shared buffer. Then, each parent rank reads the data from the children's shared buffer and reduces it with the data in its shared buffer. Thus, at the end of the reduction, the root/lead rank need to copy the data from shared buffer to the receive buffer. Thus, to avoid this copy, the lead rank of the node that participates in the inter-node communication, instead of copying the data from the send buffer to the shared buffer, it copies the data to its receive buffer. During the *gather* step, the root reduces the data from the shared buffer from the child rank directly into its receive buffer, avoiding the unnecessary copy from the shared buffer to the receive buffer. The lead rank will use the receive buffer as send buffer to perform the reduction across nodes. This optimization can be applied for both allreduce and reduce and can have a significant impact for large messages, as it appears in the critical path of the collective, since the time to perform the copying cannot be hidden.

Notice that when the MPI_IN_PLACE option is used, the receive buffer already contains the send data. Thus, in this case, the lead rank does not need to do anything.

IV. Memory Consistency and Atomicity

In our approach, a rank copies the data to a shared buffer and sets a flag to indicate the data has been copied (*release* step for broadcast and *gather* step for reduce, in Figures 2 and 3, respectively). Since the store instructions target two different memory addresses, we need to guarantee that neither the compiler nor the hardware reorder them. To ensure cross-platform compatibility we use the OpenPA platform abstraction layer, which is included in MPICH. Our implementation uses a store fence (i.e., OPA_write_barrier()) between the store to the data and the store to the flag. In x86 platforms, where we run our experiments, store instructions are not reordered, but our implementation calls the memcpy function (from the C standard library) to perform the copy. Since the implementation might use streaming stores with the non-temporal hint, which can be reordered by the CPU, we place an sfence before setting the flag.

Similarly, ranks that read the data issue a load of the flag (to wait for it to have a pre-determined value) followed by a memcpy, which will issue a load (or multiple loads) of the data. Since these load instructions target different addresses we need to ensure that they are not reordered. We do so by placing an OPA_read_barrier() which, for x86, will become a compiler fence. On other architectures, this fence would be an actual load fence instruction, if, for instance, the load is not exposed to the cache coherence protocol. Furthermore, to ensure atomicity of loads/stores issued against the same memory address, OPA_load_int() and OPA_store_int() are used to operate with the flags. For x86 platforms this does not result in performance degradation since 64-bit integer loads and stores are translated into a single instruction.

V. Environmental Setup

Experiments were run on an Intel Xeon Gold 6138F CPU, known as Skylake and referred to as SKL, which has 40 cores, 2 threads/core and a frequency of $2.0GHz$. It has $32KB$ of L1 data and instruction cache, $1MB$ of L2 cache and $27.5MB$ of L3 cache. gcc compiler version 8.1.0 was used. The operating system was SUSE Linux Enterprise Server 12 SP3 running Linux version $4.4.132 - 94.33 - default$.

To assess the performance of our implementation, we compare with other open source MPI implementations. In particular, we compare with MPICH [12] with two different devices, ch3 and ch4. MPICH/ch3 is the *default* MPICH device, but MPICH has recently introduced ch4, a new device that optimizes communication [16]. Although MPICH/ch4 is usually faster for inter-node communication, we have found that the intra-node point to point communication in ch4, which is the one used for intra-node collectives, has not yet been optimized, and hence we compare with both devices. In this

paper, we use the master branch commit id $d815dd4$ from https://github.com/pmodels/mpich for the framework as well as ch3 and ch4 runs. We also compare with MVAPICH [14] (version $2-2.3rc1$) and OpenMPI [15] (version 3.0.0), which are the two other main MPI open source implementations. Topology-aware trees were built using hwloc [17] (version 1.11.7) that provides information about the memory hierarchy of the target system.

We use the Intel® MPI Benchmarks (IMB) [18] (version 2018 Update 1) for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(). For the experiments reported in this paper we run one rank per core, i.e., 40 ranks on a node. All ranks running the IMB benchmark accumulate, locally, the time they stay inside the collective call for the total number of iterations. At the end of the execution, each rank averages its iteration time. IMB reports the time of the fastest rank (T-min), the time of the slowest rank (T-max), as well as the average time across all ranks (T-avg). For all the figures in this section, unless otherwise stated, we use T-max. Our conclusions are similar if we use T-avg. For all the data reported, we run each experiment 5 times, discard the slowest 2 runs, and we average the other 3.

VI. Experimental Results

In this section, we assess the performance benefits of our framework as compared to MVAPICH, OpenMPI, MPICH/ch3 and MPICH/ch4 for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(). Sections VI-A and VI-B analyze the performance benefit of different optimizations for MPI_Bcast() and MPI_Reduce() respectively. Section VI-C shows results on multiple node evaluation.

Figures 10(a), 10(b), and 10(c) compare performance of the different MPI implementations with our proposed implementation, that we call Release/Gather (RG), for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(), respectively, on the SKL platform. For each plot, the X-axis shows the message sizes and the Y-axis shows the normalized execution time, where the baseline is the execution time of the proposed RG approach. For clarity, we crop the plots to omit data points that have very large values. Below the X-axis, the plot shows the absolute running time in μseconds for each message size under the RG approach. Absolute runtimes for the other approaches can be computed based on these running times.

Section VI-B and VI-A discuss the configuration parameters for RG in detail. Here we describe the values used for the evaluation. For MPI_Bcast(), we use a $32KB$ shared buffer split into 4 chunks of $8KB$ each and a flat tree for the propagation of the flags. For MPI_Reduce(), we use a $32KB$ buffer split into 4 way chunks of $8KB$ each. For messages smaller than $512B$ we use a topology unaware K-nomial tree with branching factor $K = 4$, while for $512B$ and larger we use a topology aware tree with right skew and socket-leader-last. In both cases, the data copy optimization is applied. The per-socket tree uses a K-ary tree with different value of K depending on the message size:

- $K = 3$ when 512B $\leq$ message_size $<$ 8KB
- $K = 2$ when message_size $\geq$ 8KB

Both broadcast and reduce implement the optimization that *read from parent flag* in the *release* step. RG performance can be further improved by selecting a different configuration for each individual message size and platform, but since most users run the MPI library with the default configuration, that is the one we evaluate in this paper.

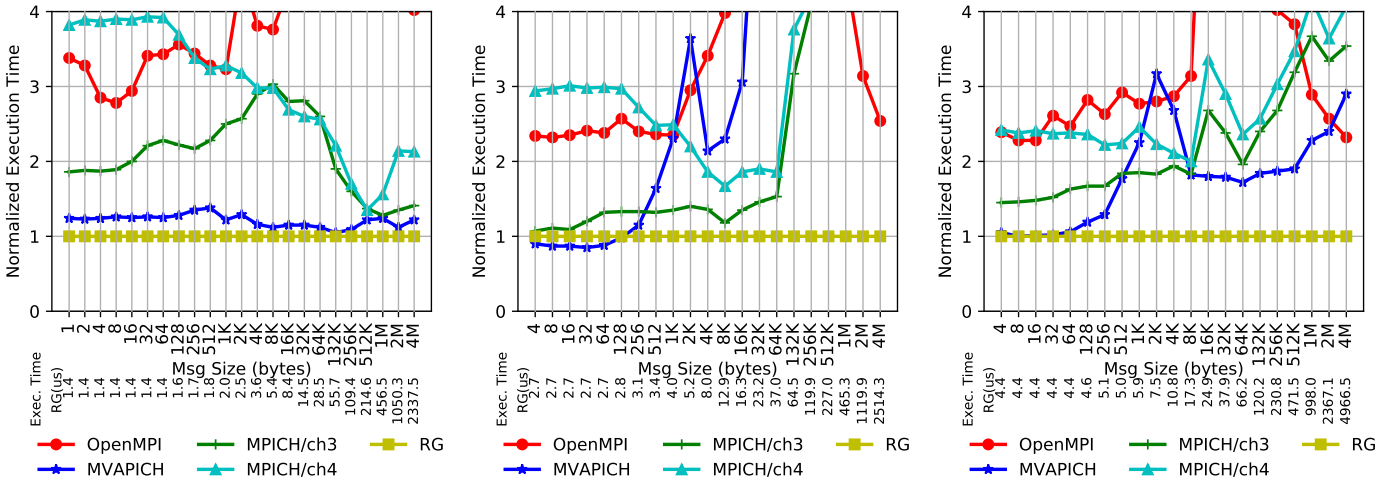
To be fair to other MPI implementations, we have searched their configuration space for each collective and compared the performance of each configuration with that of the default. In most cases, we found that the default implementation provided the best performance, but when a configuration outperformed the default by more than 5%, we changed the default to the new configuration. Using this light-weight tuning, the configuration used for each implementation is as follows. For OpenMPI, we use the default parameters. For MVAPICH, we also use the default parameters, except for the buffer size allocated for broadcast and reduce. The default configuration in MVAPICH uses 128 buffers of $8KB$ each. We have changed the MVAPICH configuration to 4 buffers of $8KB$ each, so that both MVAPICH and RG use the same amount of memory. In MPICH/ch3, broadcast uses the small messages algorithm (binomial tree) for messages up-to $32KB$, medium size message algorithm (scatter followed by recursive exchange based allgather) for messages between $32KB$ and $1MB$, and the large message algorithm(scatter followed by ring-based allgather) for messages larger than $1MB$. In MPICH/ch4, the corresponding cross-over points are $64KB$, and $512KB$. In MPICH/ch3, reduce and allreduce use the small message algorithm (binomial tree) for messages up-to $4KB$, and the large message algorithm (reduce-scatter followed by *gather* or allgather) for messages larger than $4KB$. The cross-over point for MPICH/ch4 is $8KB$.

Experimental results in Figures 10(a), 10(b), and 10(c) show that RG is usually the fastest on all the collectives evaluated, in many cases by a significant difference. For MPI_Bcast(), RG is the fastest for all message sizes, while OpenMPI and MPICH/ch3 are the second best; for MPI_Reduce(), RG is the fastest for all message sizes except for messages smaller than $128B$, where MVAPICH is 10% faster than RG. MPICH/ch3 is the second best for messages of $512B$ or larger; for MPI_Allreduce(), RG and MVAPICH are similar for messages smaller than 64B, and RG is the fastest for all other message sizes, with MVAPICH being the second best in most cases.

Next, we study the impact of each of the optimizations we enabled for each collective on the SKL platform.

A. MPI_Bcast() Optimizations

The two main optimizations that impact the performance of MPI_Bcast() are the branching factor of the tree and the number of buffers. MPI_Bcast() does not benefit from topology-aware trees, as these are only used to propagate the flags. Our results show that a *flat tree* or a tree with



(a) MPI_Bcast() (b) MPI_Reduce() (c) MPI_Allreduce()

Fig. 10: Results for different collectives on SKL

branching factor 39 (40 is the number of ranks/cores of our SKL) provides, in most cases, the best performance.

With respect to the number of buffers, our experimental results show that in general 2 buffers are enough. For messages smaller than $8KB$, MPI_Bcast() runs up-to $1.9\times$ slower with a single buffer. This can be explained by the parent stalling in the *gather* phase waiting for all the children to have copied the data out. A configuration with 8 buffers requires additional bookkeeping for large messages, running up-to $1.35\times$ slower. *Data inlining* optimization does not provide a performance benefit because, as previously discussed, it forces us to use a single buffer, and so the performance penalty due to the additional synchronization is not compensated by the small decrease on cache misses.

Finally, the *read from parent flag* optimization on the *release* step is beneficial, as the root only writes its own flag, rather than the flag of its 39 children.

B. MPI_Reduce() Optimizations

The single most important optimization for MPI_Reduce() of large messages is topology-aware trees. Figure 11 shows the impact of topology aware trees versus topology unaware trees. To determine the best configuration of the trees, we generate K-ary and K-nomial trees for topology unaware trees and for the per-socket tree in topology aware trees. In the topology unaware case, K-nomial trees were better. In the figure, the topology unaware line corresponds to a K-nomial tree with branching factors (determined empirically): $K = 8$ when message_size $\leq 16KB$ and $K = 2$ for larger messages (branching factor $K = 4$ performs the same as $K = 8$ for messages smaller than $512B$ - RG in Figure 10(b) uses $K = 4$). For the per-socket tree (in topology aware trees) the K-ary trees were better than the K-nomial. In the figure, the topology aware lines uses a socket-leader-last, right skew, K-ary tree with branching factors (determined empirically): $K = 3$, when message_size $\leq$ 4KB; $K = 2$ when message_size $> 4KB$. As Figure 11 shows, for messages smaller than $512B$, topology un-aware trees slightly outperform topology-aware trees; for larger messages, topology-aware trees significantly outperform the topology-unaware trees. The benefit of topology-aware trees increases as the message size increases, because topology aware trees optimize data movement. In the best case, topology aware trees run up-to $1.8\times$ faster.

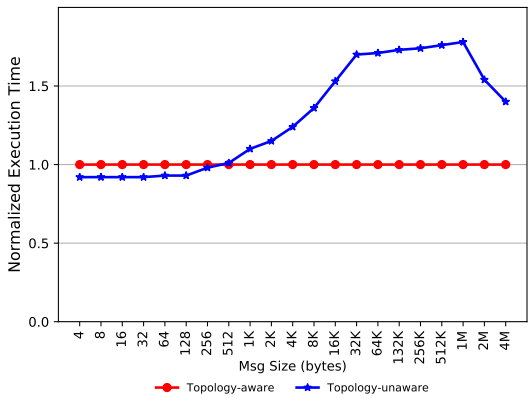


Fig. 11: Impact of topology aware trees on MPI_Reduce()

We also evaluate the performance benefits of socket-leader-first versus socket-leader-last (socket-leader-last is our default) and of right skew trees versus left skew trees (right skew is our default). Both optimizations provide speedups between $1.1\times$ and $1.25\times$ for messages smaller than $64KB$. As the message size increases, the impact of these optimizations decreases as the reduction computation becomes the dominant factor.

With respect to the branching factor (K), we find that in general larger values of branching factors are better for small message sizes. As the message size increases, smaller branching factors are better. For the topology un-aware trees used for messages smaller than $512B$, $K = 4$ or $K = 8$ provide the same performance. A branching factor $K = 2$ is

up-to $1.15\times$ slower, while $K = 3$ is up-to $1.2\times$ slower. For topology aware trees, branching factor $K = 3$ is 3% faster than $K = 2$ when $512B <$ message_size $< 8KB$. When message_size $\geq 8KB$, $K = 2$ is the best, with branching factor $K = 3$ being up-to $1.17\times$ slower than $K = 2$.

With respect to the number of buffers, our default optimization uses a total buffer size of $32KB$ (4 buffers with $8KB$ each) per rank. Thus, we assessed the benefit of allocating a single buffer of $32KB$, two buffers of $16KB$, four of $8KB$ or eight of $4KB$. All the configurations with more than one buffer outperform a single buffer, with the single buffer being up-to $1.2\times$ slower for small messages and up-to $2.3\times$ for large messages. Having multiple buffers prevents child ranks from stalling while the parent copies out the data. With 8 buffers, bookkeeping increases, and the execution time can be up-to $1.25\times$ slower for messages larger than $32KB$. Two or four buffers show similar results. Finally, notice that results vary with a larger total buffer space, but our experimental results with other buffer sizes show that $32KB$ is a good compromise.

Our baseline implementation uses the *read from parent flag* optimization during the *release* step and the *data copy* optimization. *Read from parent flag* provides small benefit, but without the *data copy* optimization MPI_Reduce() runs up-to $1.08\times$ slower than our default (with the optimization enabled). The *data inlining* optimization provides a small benefit for small messages, but since it complicates the design to also support pipeline, we did not consider it any further.

C. Multi-node evaluation

In this section, we evaluate the impact of RG when deployed in a multi-node setup. Figure 12 and 13 show the impact our proposed approach has on the overall collective execution time for MPI_Bcast() and MPI_Allreduce(), respectively[1]. For these experiments, we only compare with MPICH/ch3 and MPICH/ch4, keeping the inter-node collective implementation in all the variants the same. The only difference in the execution times can be attributed to the intra-node implementation of the collective. For MPI_Bcast(), we see that the performance difference increases as the message size increases, which is unexpected based on the results in Figure 10(a). The reason is that when running on multiple nodes, the medium and large message algorithms for broadcast ran very slowly, even though, these were the best for single-node runs. As a result, experimental results on Figure 12 use the small message algorithm for all message sizes, since those obtained better performance for multi-node runs.

For MPI_Allreduce(), Figure 13 shows that benefit increases as message size increases. This agrees with the results for MPICH/ch3 and MPICH/ch4 for MPI_Allreduce() in Figure 10(c). Overall, the runtime of MPI_Allreduce() without RG is between $1.30\times$ and $2.34\times$ slower.

As we can see from the Figures 12 and 13, the improvement provided by the RG intra-node collective has a visible impact on the execution time of the overall collective.

[1]Results for MPI_Reduce() are not shown due to lack of space. We show results for MPI_Allreduce(), which is a more important collective operation.

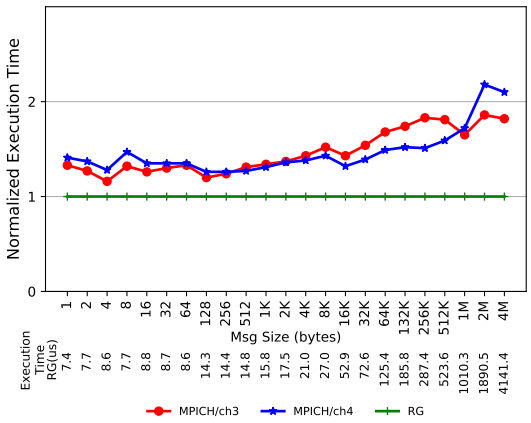


Fig. 12: MPI_Bcast() on 32 nodes with 40 ranks per node

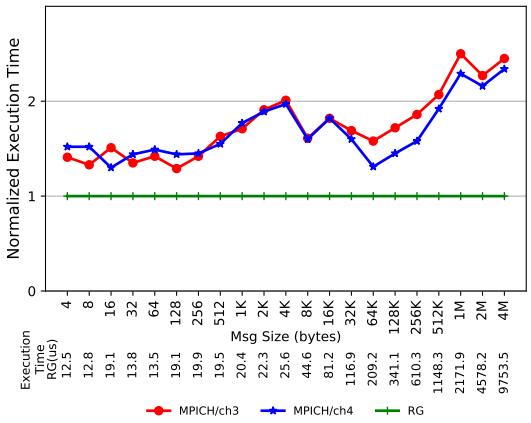


Fig. 13: MPI_Allreduce() on 32 nodes with 40 ranks per node

VII. Related Work

As previously mentioned, the *release/gather* approach discussed in this paper is also used in the LLVM *OpenMP* runtime [13] to implement a barrier, where it is used in the reverse order, first a gather and then a release. We build on top of this approach to support broadcast and reduce. In particular, we added several optimizations designed to minimize data movement and to reduce synchronization between processes. We do not compare with the LLVM implementation, as the MPI collectives require a copy-in of data to a shared buffer that is not required in the case of the OpenMP runtime.

Both MVAPICH [14], [8] and OpenMPI [15], [9] have optimized intra-node collectives. OpenMPI intra-node collectives are implemented through a variety of algorithms operating on the intra-node point-to-point communication, where the specific algorithm depends on the message and communicator sizes. By implementing the collectives through send and receive calls, OpenMPI allows the underlying communication to be abstracted. While this approach allows for extensibility, this comes at the expense of performance, as shown in this paper. MVAPICH provides optimized implementations of several collective operations (i.e. MPI_Bcast(), MPI_Reduce(), and MPI_Barrier()) that leverage shared-

memory regions to improve the intra-node portion of the operation, similar to our approach. However, in MVAPICH, the intra- and inter-node portions of the collective are tightly-coupled, i.e., implemented within the same function, in at least one of the variants. When compared to OpenMPI's design, this approach lays at the other end of the spectrum, as it may offer performance benefits at the expense of reduced re-usability and extensibility. Our approach separates the inter- and intra-node portions to facilitate the selection of the best combination for the target system. MPICH follows a similar approach to OpenMPI in that it uses the intra-node point to point path to implement the intra-node collective. To the best of our knowledge, MPICH, MVAPICH, and OpenMPI use topology un-aware trees. None of them use topology-aware trees that take into account the memory hierarchy of the node as we do. For MPI_Reduce(), MVAPICH uses flat trees for small messages and 4-ary trees for larger ones, while MPICH uses binomial trees. OpenMPI selects the algorithm based on message and communicator size. For MPI_Bcast(), MVAPICH uses a flat tree like we do. Their implementation resembles ours on the *release*, but differs on the *gather*.

Li et al. [7] present algorithms optimized for intra-node shared memory, but their approach assumes that a rank is based on a thread implementation [19], avoiding the copy-in and out from the shared buffer. We base our implementation on processes, since in main MPI implementations, such as MPICH or OpenMPI, ranks are based on processes. Also, although [7] evaluates K-ary trees with different branching factors that seem to take into account the different sockets of the machine, we evaluate more variations of topology aware trees (socket-leader-first and socket-leader-last and right/left skewed trees) as well as buffer pipelining for large messages. Also, Li et al. do not describe the synchronization process, such as the *release/gather* steps used in our approach. Other implementations also avoid the copy-in and copy-out, using kernel assisted memory copy (KNEM [20]) [21]. We did not base our implementation on kernel assisted memory because these kernels are not always available in the target systems, and wanted to have a general mechanism that will always work.

Algorithms for Scalable Synchronization on Shared-Memory to support barriers, such as the MCS Barrier, are discussed in [22], where different trees and dissemination algorithms for threads that share memory are evaluated.

VIII. Conclusions

In this paper, we have discussed and evaluated our *Release/-Gather* approach and possible optimizations to implement intra-node collectives using shared memory. With the prevalence of processors with large number of cores, the intra-node implementation of collectives has become critical to obtain good performance on large scale. By using the topology information of the underlying hardware, the data movement can be orchestrated to minimize high latency movement. Furthermore, for large messages the different phases can be overlapped by having different chunks work on different buffers. These optimizations, among others, allow the intra-node collectives to outperform other open source implementations. This directly benefits inter-node collectives since the intra-node collective is on the critical path of the collective in a multi-node execution. Our experimental results show that when compared to MPICH, our approach runs between $1.16\times$ and $2.18\times$ faster for MPI_Bcast(), and between $1.29\times$ and $2.50\times$ faster for MPI_Allreduce(), when running on a 32 node cluster. On a single node, our experimental results show that with respect to MVAPICH, our approach is, on the average, $1.22\times$, $4.43\times$, and $1.80\times$ faster for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(), respectively. With respect to OpenMPI, our approach is, on the average, $4.00\times$, $4.71\times$, and $3.38\times$ faster for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(), respectively. With respect to MPICH/ch4, our approach is, on average, $2.99\times$, $3.23\times$, and $2.72\times$ faster for MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(), respectively. Similar improvements are obtained with respect to MPICH/ch3. We have shown that our building blocks can be used to efficiently build three MPI collectives, MPI_Bcast(), MPI_Reduce(), and MPI_Allreduce(). The same building blocks could be used to implement other collectives, although that is left for future work.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy and Argonne National Laboratory and its Leadership Computing Facility under Award Number(s) DE-AC02-06CH11357.

IX. Disclaimer

This report was prepared as an account of work sponsored by an agency and/or National Laboratory of the United States Government, in. Neither the United States Government nor any agency or National Laboratory thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency or National Laboratory thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency or National Laboratory thereof. Intel, Intel Xeon Phi, and Intel Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. The benchmark results reported above may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

REFERENCES

[1] E. L. William Gropp and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, third edition ed. The MIT Press, 2014.

[2] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[3] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145823

[4] N. Jain and Y. Sabharwal, "Optimal bucket algorithms for large mpi collectives on torus interconnects," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 27–36. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810093

[5] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda, "Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[6] "Top500," https://www.top500.org.

[7] S. Li, T. Hoefler, and M. Snir, "Numa-aware shared-memory collective communication for mpi," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462903

[8] V. Tipparaju, J. Nieplocha, and D. Panda, "Fast collective operations using shared and remote memory access protocols on clusters," in *Proceedings International Parallel and Distributed Processing Symposium*, April 2003, pp. 10 pp.–.

[9] R. L. Graham and G. Shipman, "Mpi support for multi-core architectures: Optimized shared memory collectives," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2008, pp. 130–140.

[10] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The sgi altixtm 3000 global sharedmemory architecture," *Silicon Graphics, Inc.(2003)*, 2005.

[11] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Designing efficient shared address space reduction collectives for multi-/many-cores," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 1020–1029.

[12] "Mpich," http://www.mpich.org.

[13] "Openmp subproject of llvm," https://openmp.llvm.org.

[14] "Mvapich," http://mvapich.cse.ohio-state.edu.

[15] "Openmpi," https://www.open-mpi.org.

[16] K. Raffenetti, A. Amer, L. Oden, C. Archer, W. Bland, H. Fujita, Y. Guo, T. Janjusic, D. Durnov, M. Blocksome, M. Si, S. Seo, A. Langer, G. Zheng, M. Takagi, P. Coffman, J. Jose, S. Sur, A. Sannikov, S. Oblomov, M. Chuvelev, M. Hatanaka, X. Zhao, P. Fischer, T. Rathnayake, M. Otten, M. Min, and P. 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*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 62:1–62:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126963

[17] "Hwloc," https://www.open-mpi.org/projects/hwloc/.

[18] "Intel® mpi benchmarks," https://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[19] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma, "Ownership passing: Efficient distributed memory programming on multi-core systems," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442534

[20] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis," in *2009 International Conference on Parallel Processing*, Sept 2009, pp. 462–469.

[21] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra, "Kernel assisted collective intra-node mpi communication among multi-core and many-core cpus," in *2011 International Conference on Parallel Processing*, Sept 2011, pp. 532–541.

[22] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991. [Online]. Available: http://doi.acm.org/10.1145/103727.103729

APPENDIX

A. Artifact Description

1) Description:

1) **MPICH:** MPICH was obtained from https://github.com/pmodels/mpich/commit/ master branch commit id $d815dd40d2241ac80f65d01e826e25e7657f3c1b$ MPICH/ch3 was configured without any special configuration flags. MPICH/ch4 was configured in the same way as our RG framework, which is described later in this section.
2) **MVAPICH:** MVAPICH (version $2 - 2.3rc1$) was obtained from http://mvapich.cse.ohio-state.edu/downloads/. It was configured to use psm2: "–disable-mcast –with-device=ch3:psm –with-psm2-include=/path/to/psm2/include/ –with-psm2-lib=/path/to/psm2/lib64/ CFLAGS=-O3 CXXFLAGS=-O3".
3) **OpenMPI:** OpenMPI (version 3.0.0) was obtained from https://github.com/open-mpi/ompi. It was configured to use psm2 and libfabric: "–with-libfabric=/path/to/libfabric/ –with-psm2=/path/to/psm2/ LD_FLAGS=-L/path/to/psm2/ -lpsm2"
4) **Compiler:** We used the gcc compiler (version $8.1.0$) for the experiments shown in this paper.
5) **Autotools:** autoconf version 2.69, automake version 1.15 and libtool version $2.4.6$ was used in the paper for all the MPI implementations.
6) **PSM2:** opa-psm2 was obtained from https://github.com/intel/opa-psm2.git commit id $0f9213e7af8d32c291d4657ff4a3279918de1e60$. The default configuration was used.
7) **Libfabric:** Libfabric was obtained from https://github.com/ofiwg/libfabric.git commit id: $91669aa6681680e28abbb9fbec5c4be87330132c$. It was configured with psm2 using the option "–enable-psm2=/path/to/psm2/"
8) **Operating System:** The experiments were done on a machine running SUSE Linux Enterprise Server 12 SP3, Linux version $4.4.132 - 94.33 - default$.
9) **IMB Benchmarks:** Intel MPI Benchmarks were obtained from https://github.com/intel/mpi-benchmarks. Version 2018 update 1 was used. The same configuration was used for each of the MPI implementations. The experiments were run with parameters "-iter 5000 -msglog 22 -sync 1 -imb_barrier 1 -root_shift 0". IMB benchmarks use a custom barrier (not MPI_Barrier()) across invocations of the collective. This setup guarantees that the same barrier is executed by all the evaluated MPI implementations. If the benchmark were to call MPI_Barrier, the barrier would be different based on the MPI implementation. The custom barrier in IMB is based on recursive doubling, providing a tighter synchronization than a tree-based implementation, for instance. The running time of this custom barrier is not measured.
10) **Publicly available:** The data shown in this paper can be made publicly available if needed. The goal is to make our framework publicly available as well by merging it with the pmodels/mpich repository at github.com.

2) Installation: Command to generate the mpich library. ./configure –prefix=/path/to/install '–disable-perftest' '–with-libfabric=/path/to/libfabric/' '–disable-ft-tests' '–with-fwrapname=mpigf' '–with-filesystem=ufs+nfs' '–enable-timer-type=linux86_cycle' '–enable-romio' '–with-mpe=no' '–with-smpcoll=yes' '–with-assert-level=0' '–enable-shared' '–enable-static' '–enable-error-messages=yes' '–enable-visibility' '–enable-large-tests' '–enable-strict' '–enable-g=none' '–enable-error-checking=no' '–enable-fast=all,O3' '–disable-debuginfo' '–with-device=ch4:ofi:sockets' '–enable-handle-allocation=default' '–enable-threads=multiple' '–without-valgrind' '–enable-timing=none' '–enable-ch4-shm=exclusive:posix' '–enable-thread-cs=global' '–with-ch4-netmod-ofi-args=' 'MPICHLIB_CFLAGS=-O3 -Wall -ggdb -ggdb -Wall -mtune=generic -std=gnu99 -DMPIDI_OFI_FORCE_AM -I/path/to/numactl/include/ -L/path/to/numactl/lib/ -lnuma' 'MPICHLIB_CXXFLAGS=-O3 -Wall -ggdb -ggdb -Wall -mtune=generic -I/path/to/numactl/include/ -L/path/to/numactl/lib/ -lnuma' 'MPICHLIB_FCFLAGS=-O3 -Wall -ggdb -ggdb -Wall -mtune=generic' 'MPICHLIB_F77FLAGS=-O3 -Wall -ggdb -ggdb -Wall -mtune=generic' 'MPICHLIB_LDFLAGS=-O3 -L/usr/lib64 -mtune=generic' 'CC=gcc' 'LDFLAGS= -Wl,-z,muldefs -Wl,-z,now' 'CXX=g++' 'FC=gfortran' 'F77=gfortran'

B. Tuning MPI implementations

1) **MVAPICH:** We tuned MVAPICH by trying different values for the parameters listed in the user guide: *MV2_SHMEM_BCAST_MSG*, *MV2_SHMEM_REDUCE_MSG*, *MV2_SHMEM_ALLREDUCE_MSG*, *MV2_BCAST_2LEVEL_MSG*, *MV2_REDUCE_2LEVEL_MSG*, *MV2_ALLREDUCE_2LEVEL_MSG*. These parameters correspond to the thresholds to choose between the short size message, the medium size message, and the long size message algorithms. In our experiments, we ran all the message sizes with all the algorithms and found that the default were the best. We also tried different values $(2, 4, 8, 16, 32, 40)$ for *MV2_KNOMIAL_INTRA_NODE_FACTOR*, but found that the default value (4) was the best.
2) **OpenMPI:** To tune OpenMPI, we determined the relevant parameters from the ompi_info utility. For MPI_Reduce(), the relevant parameters were *coll_tuned_reduce_algorithm*, *coll_tuned_reduce_algorithm_tree_fanout*, and *coll_tuned_reduce_algorithm_chain_fanout*. *coll_tuned_reduce_algorithm* selects the algorithm to be used for reduction and ranges from 0 to *coll_tuned_reduce_algorithm_count*. The default algorithm is set to 0, which uses

a dynamic algorithm. The other parameters are used to control the tree or chain fanout degree for different algorithms. Both are set to 4 by default. We found that the default configuration performed the best for both MPI_Reduce() and MPI_Bcast().

3) **MPICH/ch3:** MPICH has several configuration variables to choose the algorithm to use based on the message size. We tuned MPICH/ch3 and MPICH/ch4 separately by running all the different algorithms for all the message sizes and chose the configuration that provided the best performance. For MPI_Bcast() we ran the search for single node and for the 32 node experiments, since the best algorithm for single node had poor performance when running 32 node experiments.
   - MPI_Bcast(): For single node runs, we set *MPIR_CVAR_BCAST_SHORT_MSG_SIZE* to $64KB$ and *MPIR_CVAR_BCAST_LONG_MSG_SIZE* to $2MB$. For 32 nodes run, we set *MPIR_CVAR_BCAST_SHORT_MSG_SIZE* to $5MB$ and *MPIR_CVAR_BCAST_LONG_MSG_SIZE* to $5MB$.
   - MPI_Reduce(): For single node run, we set *MPIR_CVAR_REDUCE_SHORT_MSG* to $4KB$.
   - MPI_Allreduce(): For single as well as 32 nodes run, we set *MPIR_CVAR_ALLREDUCE_SHORT_MSG* to $4KB$.
4) **MPICH/ch4**
   - MPI_Bcast(): For single node runs, we set *MPIR_CVAR_BCAST_SHORT_MSG_SIZE* to $128KB$ and *MPIR_CVAR_BCAST_LONG_MSG_SIZE* to $1MB$. For 32 nodes run, we set *MPIR_CVAR_BCAST_SHORT_MSG_SIZE* to $5MB$ and *MPIR_CVAR_BCAST_LONG_MSG_SIZE* to $5MB$.
   - MPI_Reduce(): For single node run, we set *MPIR_CVAR_REDUCE_SHORT_MSG* to $8KB$.
   - MPI_Allreduce(): For single as well as 32 nodes run, we set *MPIR_CVAR_ALLREDUCE_SHORT_MSG* to $8KB$.