Contention-Aware Kernel-Assisted MPI Collectives for Multi-/Many-core Systems

Sourav Chakraborty, Hari Subramoni, Dhabaleswar K Panda
Department of Computer Science and Engineering, The Ohio State University
Email: {chakraborty.52, subramoni.1, panda.2}@osu.edu

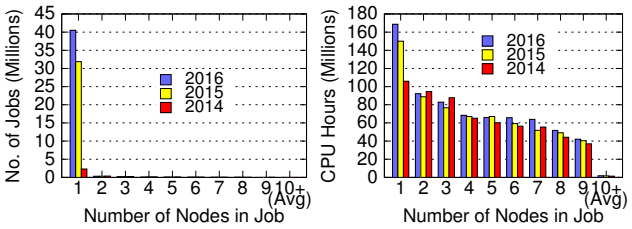
***Abstract*—Multi-/many-core CPU based architectures are seeing widespread adoption due to their unprecedented compute performance in a small power envelope. With the increasingly large number of cores on each node, applications spend a significant portion of their execution time in intra-node communication. While shared memory is commonly used for intra-node communication, it needs to copy each message once at the sender and once at the receiver side. Kernel-assisted mechanisms transfer a message using a single copy but suffer from significant contention with a large number of concurrent accesses. Consequently, naively using Kernel-assisted copy techniques in collectives can lead to severe performance degradation. In this work, we analyze and propose a model to quantify the contention and design collective algorithms to avoid this bottleneck. We evaluate the proposed designs on three different architectures - Xeon, Xeon Phi, and OpenPOWER and compare them against state-of-the-art MPI libraries - MVAPICH2, Intel MPI, and Open MPI. Our designs show up to 50x improvement for One-to-all and All-to-one collectives (Scatter and Gather) and up to 5x improvement for All-to-all collectives (Allgather and Alltoall).**

***Keywords*-Collective Communication, Kernel-assisted, Cross-Memory Attach, Multi-core, CMA, MPI, HPC**

I. Introduction

Modern High-Performance Computing (HPC) systems allow scientists and engineers to tackle grand challenges in their respective domains and make significant contributions to their fields. The design and deployment of such ultra-scale systems is fueled by the increasing use of multi-/many-core environments (Intel Xeon, Xeon Phi, and upcoming OpenPOWER architectures). These many-core architectures offer unprecedented compute power to end users within a single node enabling them to perform their compute within one or a few such high-power nodes. In fact, the usage trends of the various HPC systems that are part of the US National Science Foundation (NSF) [1] funded Extreme Science and Engineering Discovery Environment (XSEDE) [2] project indicate that jobs with one or a few nodes (≤ 9) account for the lion's share of jobs being submitted and total CPU hours consumed. These trends are illustrated by Figures 1(a) and 1(b) respectively.

The Message Passing Interface (MPI) [4] is a very popular parallel programming model for developing parallel scientific applications. MPI-based applications typically spend a large portion of the overall execution time inside the MPI library to progress communication operations. Thus, the performance of end applications is closely tied to the performance the underlying MPI implementation being used. There exists several high-performance implementations of the MPI standard (e.g. OpenMPI [5], IntelMPI [6], and MVAPICH2 [7]) that offer excellent performance and scalability on modern multi-/many-core architectures interconnected using high-performance interconnects like InfiniBand [8] and Omni-Path [9].



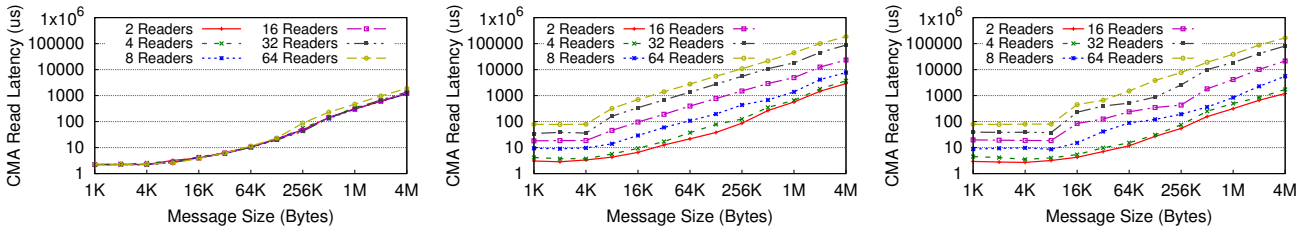
(a) Number of Jobs Submitted (b) Total CPU Hours Consumed

Fig. 1. Number of submitted jobs and total CPU hours consumed by jobs of different sizes over past three years in XSEDE clusters. Small scale jobs are the majority in both categories [3].

Given the usage trends seen with XSEDE systems, it is expected that the performance MPI implementations are able to deliver for communication happening within a node (intra-node communication) will have a significant impact on the overall performance of end applications. Researchers have done a vast amount of work to enhance the performance of MPI-based intra-node point-to-point and collective communication operations [10–16]. Based on the insights gained from these efforts, most high-performance MPI libraries offer two different options for high-performance intra-node communication — 1) the shared memory based two-copy transfer and 2) kernel-assisted single-copy transfer. These studies have also shown that, due to the overhead involved in terms of exchanging control information and context switches with from user space to kernel space, the kernel-assisted single-copy transfer scheme is typically beneficial for larger messages ($\geq$ 16KB). Multiple kernel modules (e.g. LiMIC [17], KNEM [18], and CMA [19]) are available to MPI libraries to implement these kernel-module-based transfer operations. Table I summarizes the support for various kernel assisted copy mechanisms available in various state-of-the-art MPI libraries.

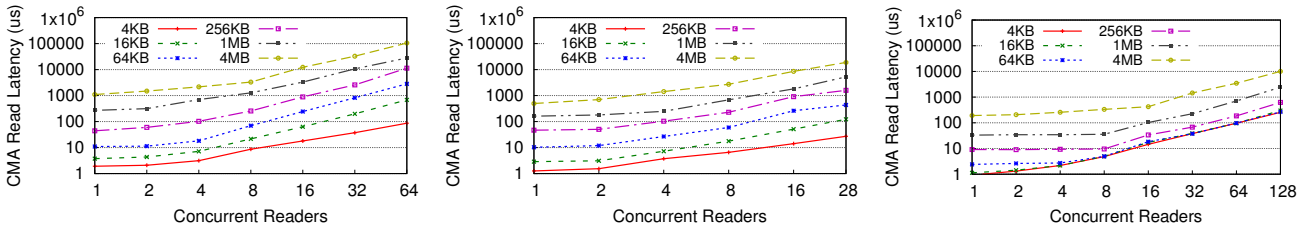
TABLE I. Support for various kernel assisted copy mechanisms in modern MPI libraries. CMA is the most widely supported.

	CMA [19]	**KNEM** [18]	**LiMIC** [17]
MVAPICH2 2.3a [7]	✓	×	✓
Intel MPI 2017 [6]	✓	×	×
Open MPI 2.1.0 [5]	✓	✓	×



(a) Different Source Processes (All-to-all) (b) Same Process, Same Buffer (One-to-all) (c) Same Process, Different Buffers (One-to-all)

Fig. 2. Impact of different communication patterns on CMA Read latency on Knights Landing



(a) Knights Landing, (68 physical cores) (b) Broadwell, (28 physical cores) (c) Power8 (20 physical cores)

Fig. 3. One-to-all communication with different number of concurrent readers and message sizes on different architectures

A. *Motivation*

While the design space for kernel-assisted intra-node point-to-point communication operations has been extensively studied, the challenges associated with and the potential performance benefits such single-copy schemes bring to collective operations are not well known. For instance, while the point-to-point based studies indicate that the kernel-assisted schemes are beneficial for larger messages ($\geq$ 16KB), this may not hold true for collective operations where there is increased concurrency in the communication. Similarly, the performance trade-off between using read-based vs. write-based approaches are not well known. In this section, we study the performance characteristics of the kernel-based single-copy transfer for different data access patterns on the Intel Xeon, Intel Knights Landing (KNL), and IBM OpenPOWER architectures. While the raw communication performance of LiMIC, CMA and KNEM are quite similar [19], a CMA based solution offers the most portability since it is available on newer Linux kernels by default and does not require installation of a kernel module. Furthermore, the CMA-based approach is more secure as it handles process permissions appropriately and does not incur the overhead of cookie creation like LiMIC and KNEM. Thus, we choose CMA for our study.

Figure 2 illustrates the performance trends seen on the Intel Knights Landing architecture for three different communication patterns — 1) All-to-all, 2) One-to-all, and 3) All-to-one. Figure 2(a) shows the performance seen with an All-to-all access pattern where processes are trying to access different memory locations at peer processes. The peer processes are chosen carefully to avoid multiple processes talking with one process at the same time. This access pattern scales well as the number of communicating pairs increases. Figure 2(b) shows the performance of a One-to-all access pattern where all peer processes are attempting to access the same memory location in the address space of one target process. We observe that such an access pattern scales poorly as the number of processes participating in the operation increases. This indicates that the kernel-based single-copy transfer scheme has some bottlenecks when multiple process are trying to access the same buffer from the same process. In order to narrow down the cause for the degradation, we perform a secondary experiment for the One-to-all access pattern where all peer processes attempt to access different memory locations in the address space of a single source process. As seen in Figure 2(c), similar degradation with increased concurrency exists even when the memory locations are different. This indicates that **the bottleneck occurs when the source process is the same**. Similar performance trends can be observed for All-to-one communication using CMA write operations. To validate the general applicability of the above insights, we measure the communication latency of the One-to-all access pattern on other state-of-the-art multi-/many-core processor architectures. As shown in Figures 3(a), 3(b), and 3(c), the **contention trends are similar across all three architectures** Knights Landing, Intel Xeon and IBM OpenPOWER.

To identify the root cause of the bottleneck, we use *ftrace* [20] kernel tracer to profile the One-to-all access pattern where all peer processes are attempting to access different memory locations in the address space of one target process. Figure 4 shows the breakdown of a CMA read operation with varying number of pages as well as different levels of contention. As shown here, the majority of the time is spent inside the get_user_pages function, which takes a lock on the page table structure of the source/target process once per page. It also shows that for the same number of pages, the time taken by the *Lock* operation increases with contention. Hence, we can conclude that **the lock operation inside the** get_user_pages **function is the source of the contention**. All three kernel-based copy mechanisms (CMA, LiMIC, and KNEM) use this function to ensure that the pages of the remote process are available and are equally affected.

While many different collective algorithms have been pro-

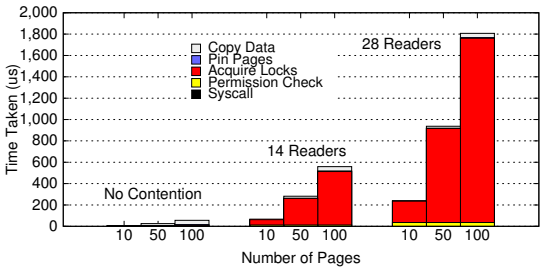


Fig. 4. Breakdown of one-to-all communication with CMA read for varying buffer sizes and process counts on Broadwell.

posed and studied on shared-memory systems, the impact of this lock contention on the performance and applicability of collective algorithms is not well-known. Due to the adoption of increasingly dense many-core architectures, the impact of this bottleneck is only going to be exacerbated.

B. Contributions

These observations lead us to the following broad challenge - ***How can we design contention-aware, kernel-assisted collectives for emerging dense many-core systems?*** In this paper, we take up this broad challenge and propose an analytical model to predict the performance of kernel-assisted communication. Our model validation indicates that the proposed model is able to accurately predict the actual performance. We use this model to evaluate known algorithms for One-to-all, All-to-one, and All-to-all collectives and determine their applicability to a given message size and architecture. We also propose new contention-aware algorithms that significantly outperform the existing algorithms. Experimental evaluation on three different architectures show that the proposed designs are able to significantly outperform the existing state-of-the-art MPI libraries such as Intel MPI, OpenMPI and MVAPICH2. To summarize, the major contributions of this paper are:

- Identify and quantify the sources of contention in kernel-assisted transfers on modern HPC systems
- Propose models to predict the performance of kernel-assisted collectives
- Design contention-aware, kernel-assisted One-to-all, All-to-one, and All-to-all collective operations
- Validate the proposed model by comparing it with observed performance
- Demonstrate the benefits of the proposed designs over state-of-the-art MPI libraries on multiple architectures

II. Cost Modeling

Based on the insights obtained from the kernel tracing, we construct a model for the communication cost by extending the model used by Thakur et. al. in [21]. The notations used in the model is described in Table II. In this model, the cost of transferring a message with no contention is $\alpha + n\beta + l\lceil\frac{\eta}{s}\rceil$. With contention, average time to acquire the lock for each page increases by a factor of γ. Thus, the cost of transferring a message with contention becomes $\alpha+n\beta+l\gamma\lceil\frac{\eta}{s}\rceil$. For CMA, the startup cost α consists of the system call overhead and the time to check if the calling process has the appropriate permissions to access the memory of the remote process.

TABLE II. Notations used in the cost model

Symbol	Description
α	Startup cost per message
β	Transfer time per Byte
η	Number of Bytes transferred
s	Number of Bytes in a page
p	Number of processes per Node
l	Time to lock and pin a page with no contention
γ_c	Contention factor with c concurrent readers/writers
T_{coll}^{sm}	Time taken to execute an intra-node collective $< coll >$ with a very small message

To measure these costs on different systems, we trigger individual steps inside the CMA read function by passing different arguments to it. The signature for the function is as follows:

```
ssize_t process_vm_readv(pid_t pid,
        const struct iovec *local_iov,
        unsigned long liovcnt,
        const struct iovec *remote_iov,
        unsigned long riovcnt,
        unsigned long flags);
```

Setting different values for $liovcnt$ and $riovcnt$ causes it to execute different steps, For example, setting $liovcnt$ to 0 and $riovcnt$ equal to the buffer size causes all the pages to be locked and pinned but no data to be copied. This can be used to measure the time taken to lock and pin different number of pages. A set of such experiments is shown in Table III. The measured times include the previous steps, hence $T_4 \geq T_3 \geq T_2 \geq T_1$. The model parameters can be calculated from these numbers by varying the value of N. For example. $\alpha = T_2$, $l = \frac{T_3 - T_2}{N}$, and $\beta = \frac{T_4 - T_3}{Ns}$. Table IV lists the values of the model parameters on different architectures.

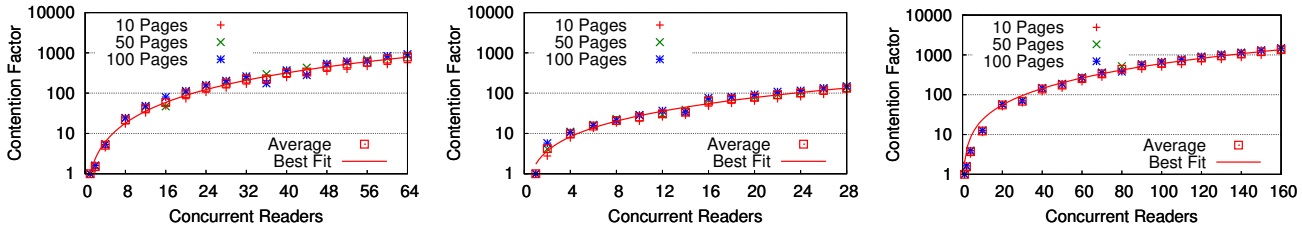
TABLE III. Determining the time taken by various steps in a CMA based transfer

Operation	Time Taken	Buffer Size	liovcnt	riovcnt
System Call	T_1	0 Byte	0 Byte	0 Byte
Access Check	T_2	1 Byte	0 Byte	1 Byte
Lock+Pin	T_3	N Pages	0 Byte	N Pages
Copy Data	T_4	N Pages	N Pages	N Pages

TABLE IV. Empirically obtained values for the model parameters on different architectures. These values are dependent on the system hardware and operating system.

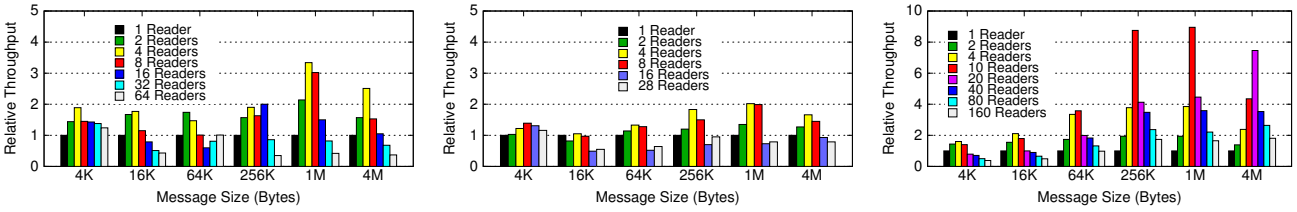
Parameter	KNL	Broadwell	Power8
α	1.43 us	0.98 us	0.75 us
β^{-1}	3.29 GBps	13.2 GBps	3.17 GBps
l	0.25 us	0.11 us	0.53 us
s	4,096 Bytes	4,096 Bytes	65,536 Bytes
γ_p	$0.11p^2 + 1.6p$	$0.18p^2 + 0.83p$	$0.04p^2 + 1.95p$

To determine the value of the contention factor, we measure the time taken to lock different number of pages with different concurrency, as shown in Figure 5. The trends match our assumption that contention factor is independent of the number of pages being locked (message size) and depends only on the concurrency. The difference between intra-socket and inter-socket contention can also be observed. Figure 5(b) shows that there is a noticeable increase beyond 14 concurrent readers in the Broadwell node which has two sockets, each equipped with 14 cores. A similar increase beyond 10 processes can



(a) KNL (b) Broadwell (c) Power8

Fig. 5. Determination of the Contention Factor (γ) on different architectures using the nonlinear least-squares (NLLS) algorithm [22].



(a) KNL (b) Broadwell (c) Power8

Fig. 6. CMA Read Throughput (relative to throughput of single reader for a given message size) with different number of concurrent readers and message sizes. Based on One-to-all communication pattern.

be seen in Figure 5(c) with the Power8 machine as well. In contrast, no such sudden increase exists in the single-socket KNL architecture, as shown in Figure 5(a).

Although contention and latency increases with higher concurrency, the non-linear nature of the contention factor suggests that there may be a "sweet spot" where the achieved throughput is maximized. To determine this region, we compare the throughput obtained with different levels of concurrency and message sizes in Figure 6. Values shown here are normalized to the throughput of a single reader (no contention). As we can see, depending on the architecture and the message size, **a particular degree of concurrency achieves the maximum throughput**.

Assumptions: Our model assumes that time taken for copying data increases linearly with the message size. Figure 2(a) shows that this assumption holds true for the message range considered here ($\geq$ 1KB). The model also does not differentiate between read vs. write bandwidth. Hence, we consider both read-based and write-based approaches while designing our algorithms.

III. Designing Native Kernel-Assisted Collectives

While most modern MPI libraries support point-to-point operations based on CMA, to the best of our knowledge, no existing MPI library has support for native CMA based collectives. However, native CMA based collectives can provide significant advantages over a design based on point-to-point transfers. Performing a point-to-point operation with CMA requires the knowledge of the PID and the buffer address of the source/target process. These information are typically exchanged through control packets (RTS/CTS) before the actual system call to copy the data is issued. For a dense collective, many such control messages must be transferred, which adds unnecessary overhead. In our design, each processes exchanges their PID with other processes on the same node during initialization. This mapping from local rank to PID is used to issue the CMA read/write operations. Note that this step is required for point-to-point operations as well. When a collective operation is initiated, some of the processes must exchange their buffer addresses based on the communication pattern. Since the message size involved in this step is very small (size of one pointer), shared memory or loopback based transfers are used. After this step, the processes can issue CMA read/write operations without further communication with the source/target process. However, for some algorithms additional synchronization is required, which is realized through 0-Byte messages sent through shared memory. In the following sections, we propose and evaluate different algorithms that take advantage of the single-copy feature and avoid contention for both personalized and non-personalized collectives.

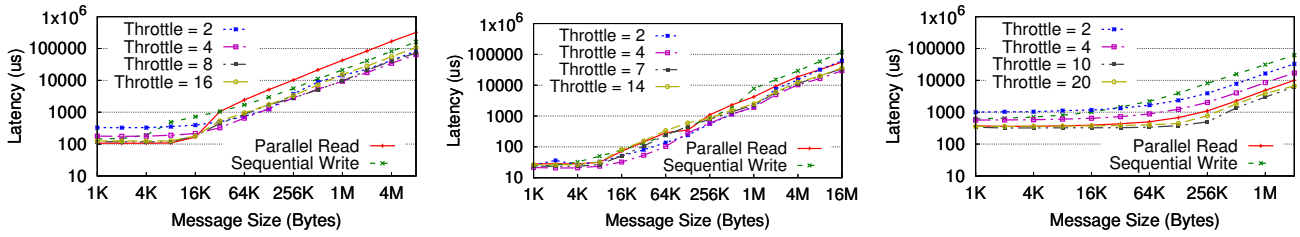
IV. Personalized Collectives

A. One-to-all (Scatter)

1) Parallel Reads: In Scatter, the root sends one personalized message to each of the non-root processes, totaling $p-1$ messages. The simplest way to achieve this is each non-root process initiate a read from the root process's buffer. The root broadcasts the address of its send buffer to the non-root processes using shared memory. The non-root processes then calculate the required offset by multiplying their local rank with the number of bytes to be received and initiates the read operation. If the send buffer is different than the receive buffer, the root copies its own message using memcpy. This step is not required if MPI_IN_PLACE is used. Upon completion of the read, the non-root processes notify the root process and the operation is completed when the root has received all $p-1$ notifications. The total cost is:

$$T_{par\_read} = T^{sm}_{bcast} + \alpha + \eta\beta + l\gamma_p\lceil\frac{\eta}{s}\rceil + T^{sm}_{gather}$$

2) Sequential Writes: In this algorithm, the root writes the data to each non-root process's memory in a sequential fashion. This requires $p-1$ steps but has no contention. Since the root is no longer idle, the copy operation for its own data cannot be overlapped with other transfers if MPI_IN_PLACE is not used. Also, the order of the intra-node gather and



(a) KNL, 64 Processes (b) Broadwell, 28 Processes (c) Power8, 160 Processes

Fig. 7. Performance comparison of different algorithms for Scatter

broadcast is reversed as the root must gather the address of the receive buffer from each non-root process and notify them upon completion. The cost is

$T_{seq\_write} = T_{memcpy} + T^{sm}_{gather} + p(\alpha + \eta\beta + l\lceil\frac{\eta}{s}\rceil) + T^{sm}_{bcast}$

where T_{memcpy} is 0 if MPI_IN_PLACE is used and $\eta\beta$ otherwise.

3) Throttled Reads: In this algorithm, k processes are allowed to concurrently read from the root process. This number k is referred to as the **"Throttle Factor".** This requires $\lceil\frac{p}{k}\rceil$ steps and the concurrency cost in each step is γ_k. While this can be achieved using $\lceil\frac{p}{k}\rceil - 1$ barriers, we reduce the synchronization cost using the following scheme. Each process posts a blocking receive from $rank - k$ if $rank - k \leq 0$. This condition is false for the first k processes and they immediately start copying the data from the root. Once the data has been copied, each process posts a send to $rank + k$ as long as $rank + k < p$. These sends unblock k of the waiting readers and this process continues until everyone has finished. Note that the algorithm does not require k to be a divisor of p. The final gather phase to identify completion is avoided by the root posting k receives from the processes involved in the last step. A single acknowledgement from rank $p-1$ is not sufficient since there are k processes performing concurrent reads at the final step. Assuming the cost of these point-to-point operations is negligible,

$$T^{k}_{throttled} = T^{sm}_{bcast} + \lceil\frac{p}{k}\rceil(\alpha + \eta\beta + l\gamma_k\lceil\frac{\eta}{s}\rceil)$$

Note that the parallel read and the sequential writes algorithms can be considered to be special cases of the throttled read algorithm with $k = p$ and $k = 1$ respectively.

4) Performance: Figure 7 shows the performance of the different algorithms for Scatter on different architectures. As shown in Figure 7(a), for small messages parallel read outperforms sequential writes. However, with larger messages, parallel read performs the worst. This matches with the trends shown in Figure 6(a), which shows that compared to single reader, throughput obtained with 64 readers is greater for small messages but smaller for large messages. The throttled read algorithm performs worse for small messages due to the synchronization overhead. This overhead is lower for larger values of throttle factor as the number of steps is smaller. For medium to large messages, throttle factors of 4 and 8 performs the best, which again matches the trends seen in Figure 6(a).

Figure 7(b) shows that the performance difference between different algorithms is smaller for Broadwell. This matches with Figure 6(b) which shows that the difference in throughput for different number of readers is only about 2x. This is likely due to the lower maximum bandwidth of DDR memory and higher clock speed, which reduces the impact of lock contention. Overall, throttle factor of 4 performs the best for most message sizes on Broadwell.

For Power8, the trends are slightly different as shown in Figure 7(c). Due to the significantly larger page size, number of locks required for a given message size is smaller compared to KNL and Broadwell. Due to the large system bandwidth of the Power8 system, algorithms with higher concurrency (large throttle factor) outperform the variants with less concurrency (small throttle factor). As expected from Figure 6(c), throttle factor of 10 performs the best by avoiding inter-socket lock contention.

B. All-to-one (Gather)

In Gather, the root collects $p-1$ messages from the non-root processes. The algorithms for Gather are very similar to the ones considered for Scatter, but with the direction of read/write operations reversed.

1) Parallel Writes: The root broadcasts the address of the receive buffer to the non-root processes through shared memory. The non-root processes then calculates the offset based on their local rank and writes the message in the appropriate location. The root determines the completion by waiting on a shared-memory based gather. Cost:

$$T_{par\_write} = T^{sm}_{bcast} + \alpha + \eta\beta + l\gamma_p\lceil\frac{\eta}{s}\rceil + T^{sm}_{gather}$$

2) Sequential Reads: The root gathers the address of receive buffers from all non-root processes and reads the messages in a sequential order. A shared-memory based broadcast is used to notify non-root processes of the completion. Total cost:

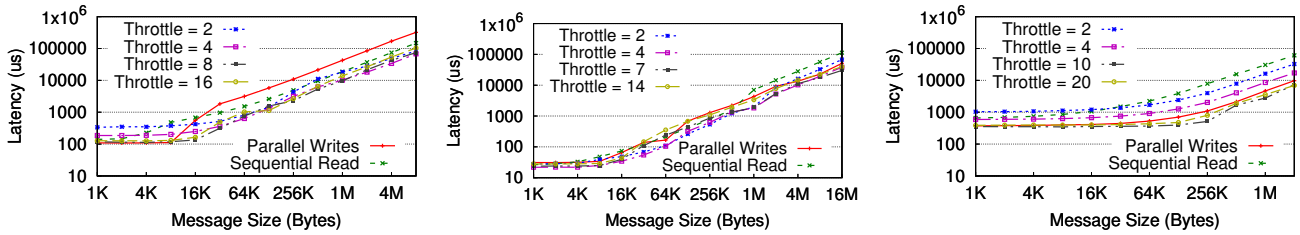
$T_{seq\_read} = T_{memcpy} + T^{sm}_{gather} + p(\alpha + \eta\beta + l\lceil\frac{\eta}{s}\rceil) + T^{sm}_{bcast}$

Similar to the sequential Scatter algorithm, $T_{memcpy} = 0$ if MPI_IN_PLACE is used as the send buffer.

3) Throttled Writes: In this algorithm, k processes are allowed to concurrently write to the receive buffer of the root process. A point-to-point message based scheme similar to the throttled read algorithm for scatter is used to synchronize the non-root processes. Time taken for this algorithm is same as the throttled read algorithm as well.

$$T^{k}_{throttled} = T^{sm}_{bcast} + \lceil\frac{p}{k}\rceil(\alpha + \eta\beta + l\gamma_k\lceil\frac{\eta}{s}\rceil)$$

4) Performance: As shown in Figure 8, performance trends for the different algorithms for Gather are very similar to their Scatter counterparts. Overall, throttle factors of 8 and 4 perform well on KNL and Broadwell while throttle factor of 10 performs the best on the Power8 architecture.



(a) KNL, 64 Processes (b) Broadwell, 28 Processes (c) Power8, 160 Processes

Fig. 8. Performance comparison of different algorithms for Gather

C. All-to-all

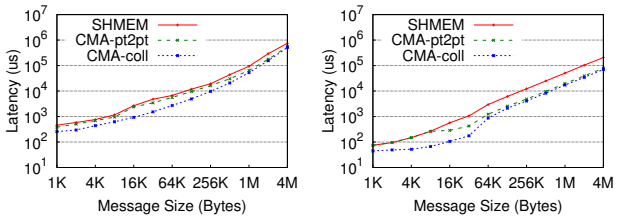
In Alltoall, each process sends a personalized message to every other process, totaling $p*(p-1)$ messages. We consider the following algorithms for Alltoall:

1) Pairwise: The pairwise algorithm is implemented in two phases. If MPI_IN_PLACE is not used, each process copies its own data from the send buffer to the receive buffer. Then the processes perform a shared-memory based allgather to collect the addresses of the source buffer of other processes. The second phase has $p-1$ steps. If p is a power-of-two, in the ith step, each process reads the message from its peer process $rank \oplus i$. If p is not a power-of-two, it reads the message from $(rank-i) \mod p$. Since each process reads from a different process, there is no lock contention. If a process sends η bytes to every other process, time taken by this algorithm is:

$$T_{pairwise} = T^{sm}_{allgather} + (p-1)(\alpha + \eta\beta + l\lceil\frac{\eta}{s}\rceil)$$

2) Bruck's Algorithm: Bruck's algorithm [23] requires $\lceil \lg p \rceil$ steps for both power-of-two and non-power-of-two processes. However, due to the additional memory copying required, it performs poorly for medium and large messages where CMA is applicable.

3) Performance: Since there is only one efficient algorithm for large message Alltoall, we take this opportunity to show the advantage of native CMA based collectives over a point-to-point based design. Figure 9 compares three different designs for the pairwise exchange algorithm - a) shared memory based (SHMEM), b) point-to-point CMA operations (CMA-pt2pt), and c) native CMA collective (CMA-coll). Due to the single-copy design, CMA-pt2pt is significantly better than the SHMEM design for large messages. CMA-coll avoids the exchange of control messages (e.g. RTS/CTS) necessary for the point-to-point operations, resulting in good improvement for the small and medium messages. For very large messages, the cost of exchanging control messages is small compared to the data movement cost and thus both CMA-coll and CMA-pt2pt show similar performance.



(a) KNL, 64 Processes (b) Broadwell, 28 Processes

Fig. 9. Performance comparison of different implementations of the Pairwise exchange algorithm for Alltoall

V. Non-personalized Collectives

A. All-to-all (Allgather)

In Allgather, each process gathers $p-1$ η-byte messages from other processes. We consider the following algorithms:

1) Ring-Neighbor: In the traditional Ring algorithm, the processes form a virtual ring and each process receives the message from process $rank-1$ and sends it to process $rank+1$. This requires $p-1$ steps. Since CMA reads are initiated by the receiver, in this scheme each process must notify its neighbor when it has completed the previous receive. The same holds true for CMA writes as well. This algorithm can be generalized where each process reads from process $(rank-j) \mod p$. We refer to this algorithm as "Ring-Neighbor-j". However, the algorithm is correct only for those values of j where $gcd(p,j)=1$. $j=1$ is the regular ring algorithm and works for any value of p.

2) Ring-Source: An alternative approach is to directly read the message from the original source. In step i, each process reads the message directly from the process $rank-i$. In this approach the read buffer is always valid and no additional synchronization is required. It is also contention free unless multiple processes end up concurrently reading from the same buffer due to skew. This approach is referred to as "Ring-Source-Read" or "Ring-Source-Write" based on the type of the CMA operation.

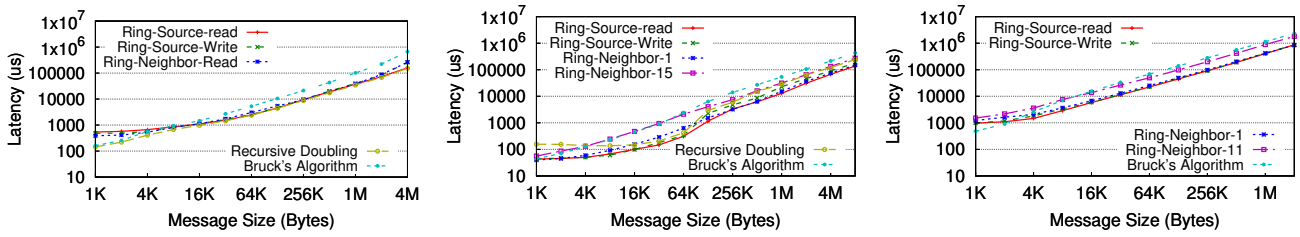
An allgather and a barrier is required to exchange the buffer addresses and to identify completion. Since each process reads from a different process, the cost is:

$$T_{ring} = T_{memcpy} + T^{sm}_{allgather} + (p-1)(\alpha + \eta\beta + l\lceil\frac{\eta}{s}\rceil) + T^{intra}_{barrier}$$

3) Recursive Doubling: Recursive doubling requires $\lg p$ steps for power-of-two processes. In step i, processes that are at distance i exchange their data along with the data received in previous steps. Thus, at each step the amount of data exchanged is $\frac{i\eta}{p}$. For non-power-of-two processes, additional steps are required to exchange the data within the subtree. Total time of this algorithm:

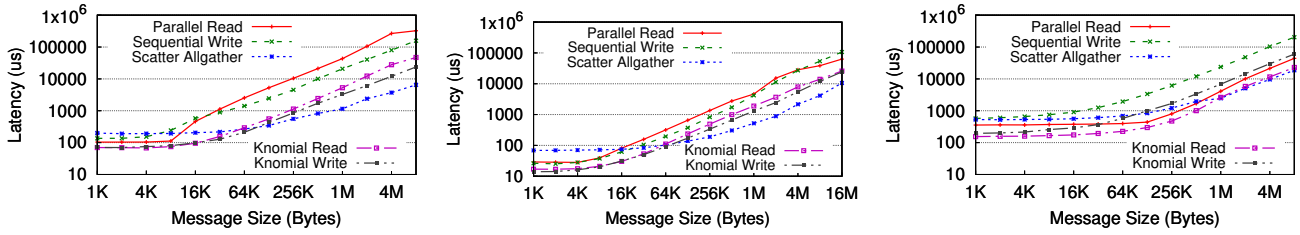
$$T_{rd} = T_{memcpy} + T^{sm}_{allgather} + \lg p\alpha + (p-1)(\eta\beta + l\lceil\frac{\eta}{s}\rceil) + T^{intra}_{barrier}$$

4) Brucks: Bruck's algorithm [23] is initialized by each process copying the input data to the beginning of the output buffer. In step i, each process copies the data from process $rank+2^i$ and appends it to the output buffer. For power-of-two processes, $\lfloor \lg p \rfloor$ such steps are required. For non-power-of-two processes, one additional step where each process receives and appends $(p-2^{\lfloor \lg p \rfloor}$ messages is required. Finally, the data in the output buffer is shifted downwards by $rank$ blocks. This



(a) KNL, 64 Processes (b) Broadwell, 28 Processes (c) Power8, 160 Processes

Fig. 10. Performance comparison of different algorithms for Allgather



(a) KNL, 64 Processes (b) Broadwell, 28 Processes (c) Power8, 160 Processes

Fig. 11. Performance comparison of different algorithms for Broadcast

final step requires $(p-1)\eta\beta$ time in the worst case. Total cost: $T_{bruck} = T^{sm}_{allgather} + \lceil \lg p \rceil \alpha + (p-1)(2\eta\beta + l\lceil \frac{\eta}{s} \rceil) + T^{intra}_{barrier}$

5) Performance: Figure 10 compares the performance of different algorithms on different architectures. On KNL (Figure 10(a)), both Recursive Doubling and Bruck's perform well for small messages due to $\lg p$ steps compared to $p-1$ steps of Ring. However, for large messages, Bruck's perform poorly due to the additional memcpy costs while Ring and Recursive Doubling perform similarly as their bandwidth terms are the same. Direct reads from source performs marginally better than reading from neighbor due to lower synchronization overhead.

On Broadwell (Figure 10(b)), the advantage of Recursive Doubling is lost due to non-power-of-two processes. However, since number of steps in Bruck's remains logarithmic and it performs well with small messages but poorly with large messages. Interestingly, the Ring algorithms perform better than recursive doubling, especially for larger messages. This is because in a Ring, most reads are intra-socket while in Recursive Doubling, the final and largest messages are inter-socket. To verify this, we evaluate two variants of the Ring algorithm. In the first one each process reads from process $rank+1$, which makes most transfers intra-socket. In the other version, each process reads from $rank+15$, thus making most of the transfers inter-socket. (The Broadwell node has two sockets, each with 14 cores.) The Neighbor-1 scheme performs significantly better than the Neighbor-15 scheme due to lower number of inter-socket transfers. As shown in Figure 10(c), similar trends can be observed on Power8.

B. One-to-all (Broadcast)

The algorithms discussed for Scatter in Section IV-A can be utilized for broadcast as well. However, since all the processes receive the same message, we can devise new algorithms that can reduce the contention.

1) Direct Reads/Writes: The Direct Read/Write algorithms are similar to the parallel read and sequential write algorithms for Scatter. The costs are:

$$T_{direct\_read} = T^{sm}_{bcast} + \alpha + \eta\beta + l\gamma_p\lceil \frac{\eta}{s} \rceil + T^{sm}_{gather}$$

$$T_{direct\_write} = T^{sm}_{gather} + p(\alpha + \eta\beta + l\lceil \frac{\eta}{s} \rceil) + T^{sm}_{bcast}$$

2) k-nomial: Based on the trends seen in Figure 6, we know that throttled reads can provide increased performance for Scatter. The analogous algorithm for broadcast is a k-nomial tree based broadcast, where up to k readers can read from the same source in parallel. Unlike Scatter where each process becomes idle after receiving the message from root, in this case they continue to send the messages down the tree. Cost of k-nomial broadcast is:

$$T_{knomial} = T^{sm}_{bcast} + \lceil \log_k p \rceil (\alpha + \eta\beta + l\gamma_k \lceil \frac{\eta}{s} \rceil)$$

3) Scatter Allgather: The Scatter-Allgather algorithm for broadcast was originally proposed by Van-der-Gejin [24]. The root divides the message into p equal chunks and scatters them to $p-1$ non-root processes. After this step, all processes perform an allgather operation on the $\frac{\eta}{p}$-byte chunks to complete the broadcast. In this scheme, the first step where the root scatters the data suffers from contention but the allgather step has no contention as there are no concurrent reads from the same process. An initial allgather step is required to exchange everyone's buffer addresses. Total time taken depends on the algorithms chosen for the Scatter and the Allgather steps.

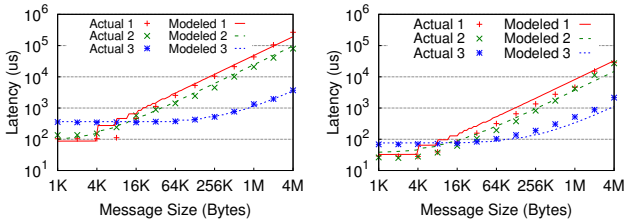
$$T_{scat-allg} = T^{sm}_{allgather} + T_{scatter}(\frac{\eta}{p}) + T_{allgather}(\frac{\eta}{p})$$

4) Performance: We compare the performance of different Broadcast algorithms in Figure 11. As expected, k-nomial algorithms perform better than direct read or write algorithms on all three architectures. For small messages, the overhead of scatter-allgather is high. However for large messages, scatter allgather performs the best due to its contention avoidance. While the step of dividing up the message among all processes has some contention, the latter steps (allgather) are contention free. The performance of k-nomial closely matches that of scatter-allgather on Power8, as shown in Figure 11(c). Due to the large number of non-power-of-two processes, individual

chunks are not page aligned and incurs some extra overhead. On the other hand, k-nomial with concurrency of 10 provides good throughput due to the trends seen in Figure 6.

VI. Model Validation

To evaluate the efficacy of our proposed communication model, we compare the predicted and observed cost for three different algorithms for broadcast - 1) Direct Read, 2) Direct Write, and 3) Scatter-Allgather. The Scatter-Allgather is implemented as a Sequential Write based scatter followed by a Ring-based allgather. We use the values shown in Table IV to calculate the predicted cost. As we can see from Figure 12, **the observed performance closely matches the predicted cost**. By using Scatter-Allgather, we indirectly validate the models for Scatter and Allgather as well.



(a) KNL, 64 Processes (b) Broadwell, 28 Processes

Fig. 12. Predicted vs. Observed performance of MPI_Bcast with different algorithms (1 = Direct Read, 2= Direct Write, 3 = Scatter Allgather) on KNL and Broadwell architectures. Observed results ("Actual") are shown as points and predicted costs ("Modeled") are shown as lines.

VII. Experimental Evaluation

In this section we compare the performance of our proposed designs against the existing algorithms in MVAPICH2 as well as other state-of-the-art MPI libraries. Table V lists the hardware specification for the three different architectures used. Our design selects the appropriate CMA algorithm for a given collective based on the architecture and message size (referred to as **"Tuned CMA"**). This was compared against the latest versions of the MPI libraries - MVAPICH2-2.3a, Intel MPI 2017, and Open MPI 2.1.0. All libraries were configured with CMA support enabled. Intel MPI was not available on the OpenPOWER system. Hence, some of the results from this cluster are omitted to save space.

TABLE V. Hardware specification of the clusters used

Specification	Xeon	Xeon Phi	OpenPOWER
Processor Family	Intel Broadwell	Knights Landing	IBM POWER-8
Processor Model	E5 v2680	KNL 7250	PPC64LE
Clock Speed	2.4 GHz	1.4 GHz	3.4 GHz
No. of Sockets	2	1	2
Cores Per Socket	14	68	10
Threads per Core	1	4	10
Mesh Config	NUMA	Cache	NUMA
RAM (DDR)	128GB	96GB	256GB
MCDRAM	-	16GB	-
Interconnect	IB-EDR(100G)	Omni-Path(100G)	IB-EDR(100G)

A. Summary of Results

In this section we present a high-level summary of the results. Numbers reported here are obtained from a single node with full subscription. Table VI compares the maximum speedup obtained with the proposed designs compared to other state-of-the-art libraries on three different architectures. As we can see, **the proposed designs can reduce the latency of personalized collectives such as Scatter and Gather by up to 50 times. It also improves performance of non-personalized collectives such as Broadcast and Allgather by up to 4 times depending on the architecture. We also reduce the latency of Alltoall by up to 5 times.**

Since kernel-assisted copy techniques are geared towards large messages, we also compare the speedups obtained for the largest message size available for a given collective on a given architecture ($\geq$4MB for KNL and Broadwell, $\geq$2MB for Power8). As shown in Table VII, collectives such as Scatter and Gather continue to show factors of improvement for the largest messages. Low contention collectives such as Alltoall and Allgather show up to 10-50% improvement compared to the state-of-the-art libraries. This reduction is consistent with the fact that for these collectives, the dominant factor for very large messages is the data movement cost and savings from reduced overhead is relatively small.

B. MPI_Scatter

Figure 13 compares the performance of the proposed contention-aware designs for MPI_Scatter against existing designs. As shown in Figure 13(a), the proposed design is up to 5 times faster than the existing MVAPICH2 design and up to 4 times faster than other state-of-the art MPI libraries on KNL with 64 processes. On Broadwell with 28 processes, we see improvements of up to 5 times compared to the best state-of-the-art implementation. For very large messages (>1MB), performance improvement is smaller in Broadwell compared to KNL. This matches with the trends in Figure 6, which shows that with properly tuned level of concurrency KNL can deliver higher relative throughput than Broadwell. We also see large improvements compared to the default designs in the Power8 architecture, as shown in Figure 13(c). Due to the large process count, naive algorithms suffer from significant contention.

C. MPI_Gather

The performance trends observed for Gather is similar to scatter, as illustrated in Figure 14(a). On KNL, we observe 5-13 times improvement compared to the existing designs depending on the message size and 2-5 times improvement for very large messages (8MB). Similar improvements are obtained on Broadwell and Power8 as well, as shown in Figure 14(b) and Figure 14(c). They also show that **CMA can be beneficial for messages as small as 1KB**.

D. MPI_Alltoall

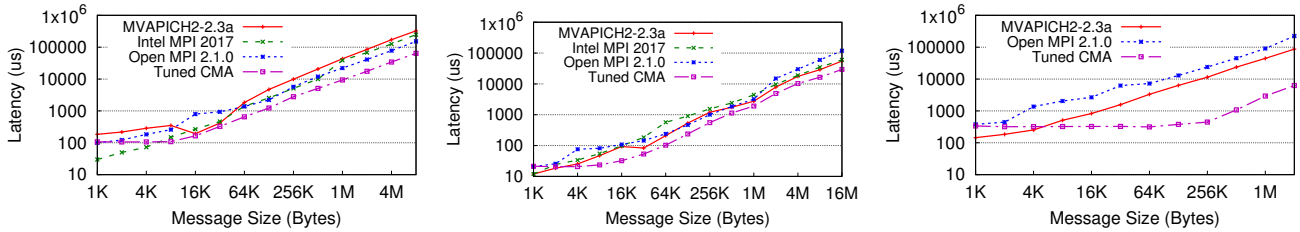
As discussed in Section IV-C, the pairwise exchange is contention-free and optimal for large message alltoall. Thus, while our design performs significantly better than a Shared-memory based design, it does not does not have a significant bandwidth advantage over an implementation based on point-to-point Kernel-based transfers. However, since a native collective design avoids the need for RTS/CTS packets, we can expect some improvement for small to medium messages. This matches with the trends shown in Figure 15, which shows 2-5

TABLE VI. Maximum speedup obtained with proposed designs ("Tuned CMA") compared to state-of-the art MPI libraries

	KNL			Broadwell			Power8	
	MVAPICH2	Intel MPI	Open MPI	MVAPICH2	Intel MPI	Open MPI	MVAPICH2	Open MPI
Bcast	2.23	1.09	3.74	1.46	1.84	3.57	4.71	5.58
Scatter	5.08	4.14	4.81	2.88	5.53	4.03	10.4	53.3
Gather	13.7	2.51	5.46	4.97	4.41	4.54	39.9	38.8
Allgather	2.07	1.55	1.95	3.70	3.22	7.09	2.51	2.52
Alltoall	2.99	1.88	2.49	4.82	4.62	2.58	3.30	5.41

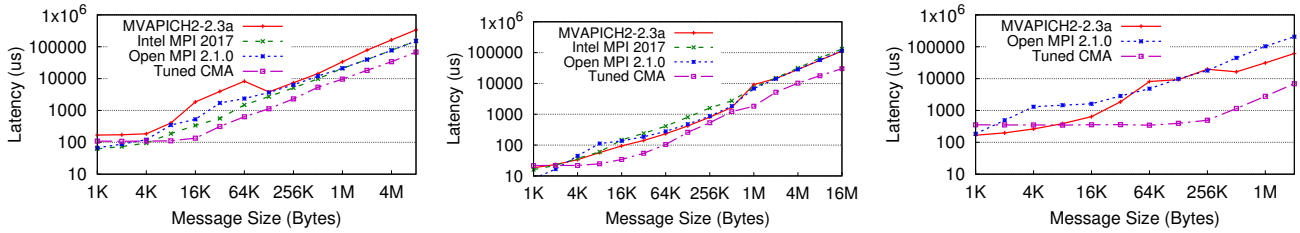
TABLE VII. Speedup obtained for the largest message size evaluated compared to state-of-the art MPI libraries

	KNL			Broadwell			Power8	
	MVAPICH2	Intel MPI	Open MPI	MVAPICH2	Intel MPI	Open MPI	MVAPICH2	Open MPI
Bcast	1.52	1.09	3.74	1.18	1.20	1.86	2.14	2.02
Scatter	5.08	3.87	2.37	1.87	2.04	4.03	13.8	35.6
Gather	4.99	2.24	2.23	3.71	4.35	3.80	8.86	29.8
Allgather	1.41	1.31	1.65	1.41	1.23	1.78	1.25	1.24
Alltoall	1.18	1.10	1.07	1.33	1.14	1.11	1.07	1.06



(a) KNL, 64 Processes (b) Broadwell, 28 Processes (c) Power8, 160 Processes

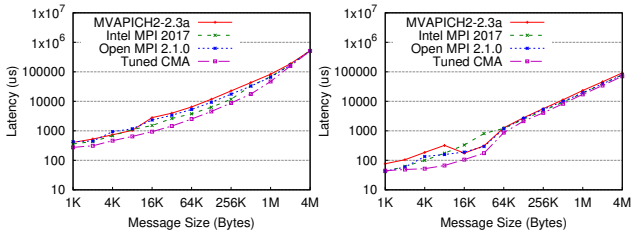
Fig. 13. Performance comparison of MPI_Scatter with proposed designs on different architectures



(a) KNL, 64 Processes (b) Broadwell, 28 Processes (c) Power8, 160 Processes

Fig. 14. Performance comparison of MPI_Gather with proposed designs on different architectures

times improvement over existing libraries for small to medium messages. For very large messages, time to copy the messages become the dominating factor and the improvement is reduced to 5-15%.

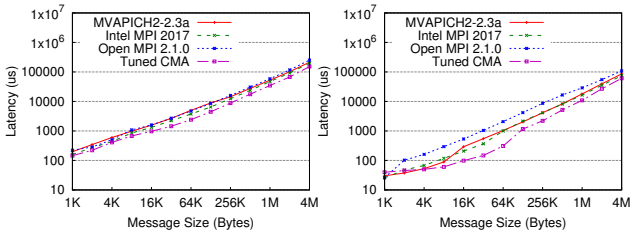


(a) KNL, 64 Processes (b) Broadwell, 28 Processes

Fig. 15. Performance comparison of MPI_Alltoall

E. MPI_Allgather

Figure 16 shows the performance of the proposed Allgather design. As shown in Figure 16(a), the native CMA based design performs 1.5-2 times better than other state-of-the art libraries and shows benefit till the largest message sizes. Due to the intra- and inter-socket awareness, our proposed designs perform up to 3-7 times faster on Broadwell and up to 2.5 times on Power8.

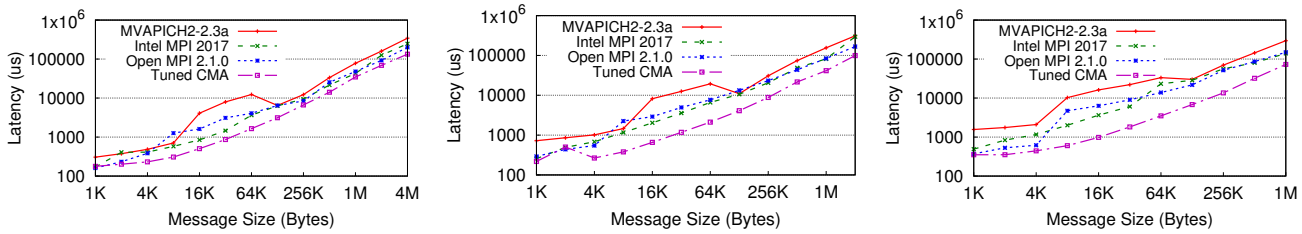


(a) KNL, 64 Processes (b) Broadwell, 28 Processes

Fig. 16. Performance comparison of MPI_Allgather

F. MPI_Bcast

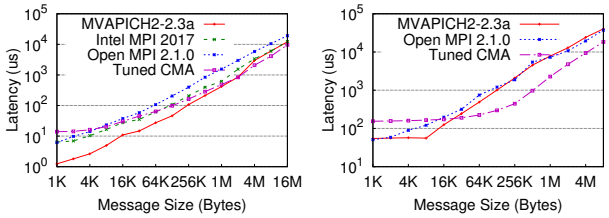
Figures 18(a) and 18(b) show the latency of MPI_Bcast on Broadwell and Power8 systems. On Broadwell, the CMA based design does not show benefit for messages $\leq$2MB. This is because a shared memory based broadcast implementation requires p copies while a CMA/kernel-assisted implementation



(a) 2 Nodes, 128 Processes (b) 4 Nodes, 256 Processes (c) 8 Nodes, 512 Processes

Fig. 17. Latency comparison of MPI_Gather on different number of KNL nodes

requires $p-1$ copies. This minor difference in the number of copies is overshadowed by the contention in CMA. Hence, on Broadwell, shared memory based broadcast performs better for message smaller than $\leq$2MB and CMA is beneficial for larger messages. While we show the entire message range here for comparison, we use the collective tuning framework of MVAPICH2 to **automatically select either CMA or shared memory based designs to provide the best performance** for a given message size and process count. On Power8, our proposed design (k-nomial read) is able to take advantage of the higher aggregate throughput and perform better than existing designs for message sizes $\geq$32KB. Overall, the proposed designs are able to achieve up to 3-4 times reduction in latency in the large message range.



(a) Broadwell, 28 Processes (b) Power8, 160 Processes

Fig. 18. Performance comparison of MPI_Bcast

G. Multi-Node Scalability

While the proposed designs focus on improving the intra-node performance, these improvements are applicable to multi-node jobs as well. Figure 17 compares the performance of the proposed designs with MPI_Gather against state-of-the art libraries on 2, 4, and 8 KNL nodes with 128, 256, and 512 processes respectively. As we can see, these configurations show up to 2x, 3x, and 5x improvement respectively compared to the best performing state-of-the-art library. This counter intuitive increase in improvement with increasing node count is due to the two-level Gather design. To take advantage of the improved intra-node performance, we design a two-level Gather where the local rank 0 on each node gathers the data from its same-node peers, followed by a inter-node gather by the global rank 0. Most modern MPI libraries have been using single-level Gather algorithms for large messages due to poor performance of intra-node Gather designs. However, **with the proposed improvements, it is possible to design new hierarchical algorithms that can deliver better performance at large scale.** More advanced designs such as pipelined two-level gather can be used to overlap inter- and intra-node transfers to further improve the performance. Similar performance improvements were observed with MPI_Scatter.

VIII. Related Work

Benefits of kernel-assisted single-copy transfers have been shown in [25–27]. LiMIC [17] and KNEM [18] are two kernel module based solutions that support single-copy transfers. SMARTMAP [28] and XPMEM [29] are two solutions specific to Cray and SGI that provide similar capabilities. CMA was introduced in Linux kernel 3.2 to address the limitations of LiMIC and KNEM and provide a portable way for any MPI library to use single-copy transfers [30]. An evaluation of CMA in MPI libraries can be found in [19].

Collective performance of MPI on shared-memory systems has been a popular topic among researchers [11, 13–16, 31, 32]. However, while many of the algorithms discussed in this work are well-known, their performance and applicability in context of kernel-assisted designs on modern many-core architectures are not. In this work, we develop an analytical model to predict the cost of different algorithms with kernel-assisted transfers. We implement and evaluate the known algorithms on different architectures and propose new algorithms where required. Ma et. al. proposed kernel-assisted [10] and hierarchical designs [33] for Broadcast and Allgather using KNEM. However, their work do not consider the lock contention. In this paper, we show that this is a significant issue with high concurrency and requires careful designs to extract the best performance. For experimental evaluation, we compare our designs against the latest OpenMPI collective module, which to the best of our knowledge incorporates their proposed designs.

IX. Conclusion and Future Work

In this paper, we identified a major contention present in kernel-assisted single-copy techniques and proposed an analytical model to quantify this. Based on this model, we designed contention-aware, kernel-assisted collective algorithms for personalized and non-personalized One-to-all, All-to-one, and All-to-all communication. We also validated our model and demonstrated that the proposed model is able to accurately predict the actual performance obtained on modern HPC systems. Experimental evaluation on different architectures like Xeon, KNL, and OpenPOWER showed that the proposed designs are able to outperform the state-of-the-art solutions available in multiple MPI libraries like IntelMPI, OpenMPI and MVAPICH2 by up to 50x for personalized and up to 5x for non-personalized collectives. Going forward, we plan to extend these designs to other collectives and design efficient multi-level collectives that overlap intra-node and inter-node communication to achieve the best performance at scale.

ACKNOWLEDGMENTS

This research is supported in part by National Science Foundation grants #CNS-1419123, #CNS-1513120, #ACI-1450440 and #CCF-1565414.

REFERENCES

[1] "US National Science Foundation," https://www.nsf.gov/.

[2] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "XSEDE: Accelerating Scientific Discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[3] "XDMoD: Comprehensive HPC System Management Tool," https://xdmod.ccr.buffalo.edu/.

[4] "MPI-3 Standard Document," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[5] The Open MPI Development Team, "Open MPI : Open Source High Performance Computing," http://www.open-mpi.org.

[6] Intel Coporation, "Intel MPI Library," http://software.intel.com/en-us/intel-mpi-library/.

[7] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, https://mvapich.cse.ohio-state.edu/.

[8] InfiniBand Trade Association, http://www.infinibandta.org/.

[9] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics," in *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*. IEEE, 2015, pp. 1–9.

[10] 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 *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 532–541.

[11] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda, "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics," in *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*. IEEE, 2008, pp. 130–137.

[12] 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.

[13] S. Sistare, R. Vaart, and E. Loh, "Optimization of MPI Collectives on Clusters of Large-scale SMP's," in *Supercomputing, ACM/IEEE 1999 Conference*. IEEE, 1999, pp. 23–23.

[14] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of MPI Collective Communication on BlueGene/L systems," in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 253–262.

[15] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, "Hierarchical Collectives in MPICH2," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2009, pp. 325–326.

[16] 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*. ACM, 2013, pp. 85–96.

[17] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster," in *Parallel Processing, 2005. ICPP 2005. International Conference on*. IEEE, 2005, pp. 184–191.

[18] B. Goglin and S. Moreaud, "KNEM: A Generic and Scalable Kernel-Assisted Intra-Node MPI Communication Framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

[19] J. Vienne, "Benefits of Cross Memory Attach for MPI libraries on HPC Clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 2014, p. 33.

[20] T. Bird, "Measuring Function Duration with Ftrace," in *Proceedings of the Linux Symposium*, 2009, pp. 47–54.

[21] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[22] D. W. Marquardt, "An Algorithm for Least-squares Estimation of Nonlinear Parameters," *Journal of the society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[23] J. Bruck, D. Dolev, C.-T. Ho, M.-C. Roşu, and R. Strong, "Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. ACM, 1995, pp. 64–73.

[24] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. Van De Geijn, "On Optimizing Collective Communication," in *Cluster Computing, 2004 IEEE International Conference on*. IEEE, 2004, pp. 145–155.

[25] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda, "Designing an Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 2008, pp. 222–229.

[26] S. Moreaud, B. Goglin, D. Goodell, and R. Namyst, "Optimizing MPI Communication within Large Multicore Nodes with Kernel Assistance," in *Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010*, 2010, pp. 7–p.

[27] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, Intranode, Large-message

MPI Communication with MPICH2-Nemesis," in *Parallel Processing, 2009. ICPP'09. International Conference on*. IEEE, 2009, pp. 462–469.

[28] R. Brightwell and K. Pedretti, "Optimizing Multi-core MPI Collectives with SMARTMAP," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 370–377.

[29] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The SGI AltixTM 3000 Global Shared Memory Architecture," *Silicon Graphics, Inc.(2003)*, 2005.

[30] C. Yeoh, "Cross Memory Attach," https://lwn.net/Articles/405284/.

[31] R. L. Graham and G. Shipman, "MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives," *Lecture Notes in Computer Science*, vol. 5205, p. 130, 2008.

[32] B. Tu, M. Zou, J. Zhan, X. Zhao, and J. Fan, "Multi-core Aware Optimization for MPI Collectives," in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 322–325.

[33] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra, "HierKNEM: an Adaptive Framework for Kernel-assisted and Topology-aware Collective Communications on Many-core Clusters," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 970–982.