A framework for hierarchical single-copy MPI collectives on multicore nodes

George Katevenis, Manolis Ploumidis, and Manolis Marazakis

Institute of Computer Science (ICS), Foundation for Research and Technology – Hellas (FORTH), Greece
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {gkatev, ploumid, maraz}@ics.forth.gr

2022 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-6654-9856-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLUSTER51413.2022.00024

***Abstract*—Collective operations are widely used by MPI applications to realize their communication patterns. Their efficiency is crucial for both performance and scalability of parallel applications. For deriving efficient MPI implementations, significant effort is put to keep pace with advances and capabilities of the underlying hardware and interconnect. Recent processor advances have led to nodes with higher core counts and complex internal structures and memory hierarchies. Such nodes are able to host tens to hundreds of processes and thus, performance of MPI collectives at the intra-node level becomes critical.**

In this work, we propose a framework for collective operations at the intra-node level, that aims to lower latency and increase bandwidth. Our approach utilizes knowledge of internal node structure to construct hierarchical algorithms, and XPMEM to achieve single-copy transfers. Pipelining is used to overlap communication at different levels of the hierarchy. We evaluate the proposed approach through several microbenchmarks and real-world MPI applications. For evaluation purposes, we compare the proposed approach with implementations of similar schemes from two recent studies.

Our evaluation with microbenchmarks for Broadcast and Allreduce shows speedup up to $2.5x$ and $3x$, respectively, over UCC and OpenMPI's default collectives implementation. Compared to recent research studies, we improve Broadcast by up to $5x$, and Allreduce by up to $7x$. We reduce the time of three applications – PiSvM, miniAMR and CNTK, by up to 12%, 52% and 12%, respectively, over the next best-performing alternative.

***Index Terms*—MPI, collectives, intra-node, hierarchical, single-copy**

I. Introduction

HPC systems continuously grow in size and complexity to meet the ever-increasing demand for computing capacity and capability. More than ever, efficient runtimes are needed, to utilize the available resources optimally while concealing the system's complexity from applications. A large number of applications, from a wide range of scientific fields, rely on parallel programming to exploit the resources of HPC systems. Message Passing Interface (MPI) is the de-facto standard for parallel programming, with several widely adopted implementations: OpenMPI [1], MPICH [2], MVAPICH2 [3], and more.

MPI offers Collective operations which allow communication between groups of processes. Several studies have outlined how application performance can significantly benefit from well-optimized collectives implementations [4]–[7]. A lot of effort has been put into optimizing them depending on the type of interconnect, system topology [6], [8], [9] and the presence of special hardware assists [10], [11]. Modern HPC systems often include highly complex nodes with 10s or even 100s of cores and elaborate internal topologies. Considering the large number of MPI processes that each node hosts, the performance of MPI collectives at the intra-node level is crucial.

A well studied approach consists of algorithms that implement the collective primitives using point-to-point operations [12]–[15]. While the actual underlying transport itself can be highly efficient, the one-to-one nature of point-to-point messages limits the achievable performance gains. Furthermore, overheads of the point-to-point layer (e.g. tag matching, rendezvous protocol) can induce additional performance loss.

Another method which has received significant attention, is direct implementation over a memory segment that is shared among communicating MPI processes [16]–[18]. Direct implementation suggests that no point-to-point exchanges are utilized; both data movement and synchronization must be explicitly handled by the algorithm. However, the shared-memory method entails that *two* copies are performed for each one transmitted message – one copy for the sender to copy it into the shared segment, and another for the receiver to copy it out, resulting in significant overhead and increased CPU load (when offloading is not possible) for large messages. Furthermore, prior work has pointed out that this approach can negatively impact performance even further, as it causes cache pollution, triggering eviction of application data [19], [20].

To avoid the shortcomings of dual copies, other memory-based approaches instead rely on kernel-assisted single-copy mechanisms, such as KNEM [20] and XPMEM [21], [22]. While works that implement memory-based collectives often take the *direct* approach, the above techniques are not exclusive to them; with the appropriate underlying implementation, point-to-point messages can also be transported over shared memory or via single-copy mechanisms.

Irrespective of whether data movement is achieved through point-to-point messages, shared memory, or the single copy paradigm, another important parameter is whether these schemes take into account the platform's topology (e.g. NUMA domains, processor sockets). Examples of topology aware schemes suggest hierarchical algorithms or algorithms

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00024

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

that dynamically adapt to the underlying topology and process-to-core mappings [23], [24]. In case it is disregarded, the mismatch between the physical topology and the corresponding communication pattern may incur phenomena like increased traffic to distant locations or contention due to fan-in, fan-out patterns.

In this work, we tackle the challenge of deriving efficient algorithms for collectives at the intra-node level. To that end, we implement within OpenMPI the *XPMEM-based Hierarchical Collectives* (XHC) component, which currently implements the Broadcast and Allreduce primitives; these primitives are commonly utilized in several MPI applications, and are representative of the different MPI collective communication patterns. The key contributions of this work are as follows:

- We identify several parameters that influence the node-level performance of collectives, and present experimental results that motivate appropriate design decisions.
- We design and implement hierarchical algorithms for the Broadcast and Allreduce operations, aimed to increase performance for all messages sizes. Specifically:
  - We combine the benefits of shared-memory for small messages, with the single-copy capabilities of XPMEM for larger ones.
  - The algorithms' communication hierarchy reflects the internal node structure (i.e. NUMA nodes, sockets, L3 caches). These topology-aware algorithms group nearby cores, and arrange for the majority of communication to take place locally, limiting crossings to distant topological domains. Moreover, hierarchy partitions traffic on different levels and avoids congestion from fan-in or fan-out patterns.
  - Pipelining is utilized to overlap communication at different levels of the hierarchy.
  - We provide direct implementations with explicit handling of synchronization, following the *single-writer, multiple-readers* paradigm, avoiding the overhead of atomics or shared-memory locks.
- We evaluate the proposed approach using both microbenchmarks and full MPI applications. Platforms with different and elaborate internal structures and high core counts are employed. For performance comparison we rely on OpenMPI's collective components, the UCC library, and frameworks from two recent relevant studies.
- We demonstrate how the system's cache hierarchy can implicitly speed up fan-out synchronization, and we discuss differences in that regard between platforms with last-level caches and a system-level cache.

XHC speeds-up Broadcast by up to $2.5x$ and Allreduce by up to $3x$, compared to OpenMPI's default implementation and to the UCC library. The benefit is higher on platforms with higher core counts. XHC's hierarchical variant reduces broadcast latency by up to $1.3x$, $2.2x$, and $4.5x$, from the smaller to the largest system, respectively, over the flat-tree one. Compared to two recent research studies, XHC improves broadcast by up to $5x$ and Allreduce by up $7x$. These benefits translate to higher performance for MPI applications, reducing the time of PiSvM, miniAMR and CNTK by up to 12%, 52% and 12%, respectively, over the next-best alternative.

II. Background

A. OpenMPI

In this work, we take advantage of OpenMPI's modular component architecture (MCA) [1] and implement our approach as a discrete component under its collectives (*coll*) framework. Two other noteworthy components in the framework are *tuned* and *sm*. The *tuned* component is based on algorithms that use point-to-point message exchanges while *sm* implements collectives *directly* over shared memory.

B. XPMEM

The XPMEM (Cross-Partition Memory) kernel module allows a process to expose an area of its address space to other processes on the same node. Peer processes can create mappings to this area, and interact with the exposed memory via ordinary user-level load and store accesses.

Several other approaches have relied on other mechanisms that offer single-copy communication capabilities, including KNEM, CMA [25], and LiMIC [26], [27]. However, these mechanisms have a number of significant disadvantages. As researchers have studied in [28], they suffer from increased lock contention inside the kernel as the number of processes in the node increases, rendering them inefficient for modern systems. Furthermore, because these mechanisms only support copying data, truly single-copy reductions are not possible; an additional *copy-in* step is required before reduction is possible, in contrast to XPMEM which allows reduction directly from other processes' buffers.

However, XPMEM also has important associated overheads, which can make new attachments expensive. Apart from the overhead of the system calls themselves, new mappings will generate page faults that trigger kernel involvement for page table manipulation – a costly ordeal. Furthermore, mappings need to be detached when no longer in use, also contributing to the overall overhead. These overheads have also been studied in [5]. Nevertheless, XPMEM's strength lies in the ability to reuse the established mappings multiple times. After the initial attachment and associated necessary actions, no kernel involvement is required in subsequent accesses concerning the same address range, and overhead can be amortized throughout multiple operations. For this reason, XPMEM is frequently paired with a registration cache: a data structure that keeps already established inter-process mappings, and allows for them to be retrieved and re-used at a later time.

III. Design of the XHC Collectives Framework

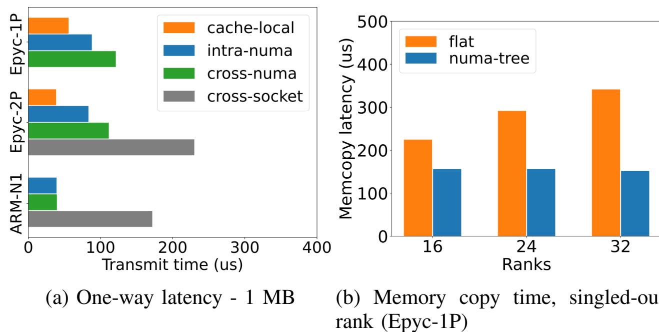
In this section we discuss the parameters of the proposed collectives framework – XHC. We also present measurements to motivate our design decisions. The measurements are performed on the same systems that we use for our evaluation, namely Epyc-1P, Epyc-2P, and ARM-N1. More details about these systems are available in Table I (Section V).

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

A. Hierarchy

The nodes that comprise modern HPC systems continuously grow in size while becoming increasingly more complex. In order to achieve performance scaling on these large systems, designers segment the processor's resources and construct localized clusters, which allow their members to operate more independently and with decreased interference between them. This results in heterogeneity inside the node, with modern machines featuring multiple NUMA nodes and processor packages, as well as elaborate cache hierarchies.

To demonstrate this heterogeneity in our targeted systems, we measure performance within and across different topological domains. Fig. 1a shows results concerning bandwidth performance (1 MB messages) – we have observed a similar pattern latency-wise (4 byte messages). Details about the systems considered in the figure are available in Table I (Section V). Message transmission times are derived through point-to-point message exchanges for pairs of ranks residing on different cores. We observe that transmission time is significantly larger when the cores are on different sockets (cross-socket). Note that Epyc-1P only contains a single socket. Furthermore, transmission time is lower when the two cores involved share a last level cache (LLC - *cache-local*). Note that this feature is only present on Epyc-1P and Epyc-2P. When cores are on the same NUMA node, but no longer share an LLC (intra-numa), transmission time exhibits a notable increase, and further increasing the distance so that the two cores are on different NUMA nodes (cross-numa), further elevates it. On ARM-N1, this elevation is marginal, and both intra-NUMA and inter-NUMA transmission times are effectively the same. Note that ARM-N1 features a system-level cache (SLC), but not a shared LLC, and the cache-local transfers we measure for the two other systems are not applicable here.



(a) One-way latency - 1 MB

(b) Memory copy time, singled-out rank (Epyc-1P)

Fig. 1. Performance across topological domains and memory congestion

On nodes with high core counts, algorithms that cause fan-in or fan-out patterns are expected to cause congestion. To explore this phenomenon, we perform two different experiments on the Epyc-1P platform. In the first experiment, ranks are organized in a flat tree and concurrently perform a memory copy of 1 MB from a root rank. In the second one, they are organized according to a two-level NUMA-wise hierarchy, in a similar manner to the one depicted in Fig. 2. In this case, members copy concurrently from their respective leaders instead of the root. In Fig. 1b we show how the performance of one specific rank scales, with either scheme, as more ranks that perform copies are added. We see that with the flat tree arrangement, the memory copy latency increases along with the number of participating processes, indicating congestion. On the other hand, with the hierarchical variant, ranks in different NUMA nodes concurrently read from their respective leader, without affecting ranks in foreign NUMA nodes. Note that, the NUMA node that the rank whose performance we measure resides in, is fully occupied in all scenarios.

To address this heterogeneity, we design and implement the *XPMEM-based Hierarchical Collectives* (XHC) framework, which groups neighbouring cores and organizes them in a *n*-level topology-aware hierarchy. The goal is to minimize costly traffic between distant domains, replacing it with localized communication to the degree possible. In addition to avoiding distant accesses, localizing communication helps avoid congestion. As an implicit benefit, it achieves better cache locality, since neighbouring cores often share a common last-level cache. The hierarchy is used to dictate the communication patterns of the framework's algorithms. An example hierarchy constructed by XHC is shown in Fig. 2. This hierarchy is based on a hypothetical system with a total of 16 cores across 2 sockets, and 4 cores per NUMA node (2 NUMA nodes per socket). The specified *numa+socket* sensitivity has resulted in a 3-level hierarchy, such that cores are grouped according to their NUMA locality, and the NUMA-local groups are further grouped according to their socket locality. At each group, one of the participants is elected as leader and its role is to exchange data on behalf of its group with same-level leaders.

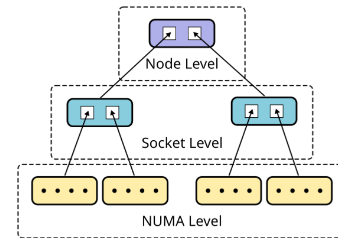


Fig. 2. XHC 3-level hierarchy, with "numa+socket" sensitivity

XHC relies on Portable Hardware Locality (hwloc) [29] for discovering the internal node structure. Hwloc identifies the location of cores per NUMA node and per socket/package, as well as the system's cache cache hierarchy. It is integrated within OpenMPI, and its information is conveniently made part of an MPI communicator's process list.

B. Pipelining

An inherent downside to tree-based communication is that it introduces a degree of serialization. Progress at one level of the hierarchy cannot be made until all steps in the previous level have completed. We apply pipelining to minimize the effect of hierarchy-induced overheads and to overlap communication on multiple levels. Each message is split up into smaller chunks, and processing at each level occurs in chunk granularity. The

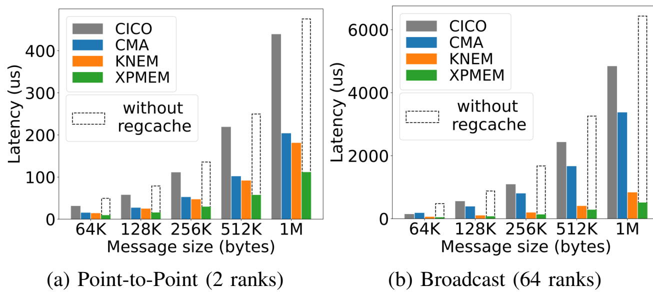
Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

shorter processing time allows for pieces of the message to be forwarded to the next levels quicker, thus reducing overall idle time. The chunk size in our implementation is run-time configurable (through OpenMPI's built-in tuning mechanism). Moreover, each level can be configured to have a different chunk size that best matches the corresponding link.

C. Single-copy

Our design utilizes XPMEM to achieve single-copy transfers. The selection of XPMEM is based on the advantages that it possess over similar mechanisms (CMA, KNEM), as those were outlined in Section II-B. To leverage XPMEM's functionality, XHC integrates with OpenMPI's *shared-memory-single-copy* (SMSC) component. Furthermore, as we also discussed in Section II-B, an important complementary component to XPMEM is a registration cache; XHC utilizes SMSC's integrated registration cache.

To further motivate our use of XPMEM instead of the other single-copy mechanisms, we utilize two MPI micro-benchmarks from the OSU micro-benchmark suite [30]: a point-to-point (*osu_latency*) with two processes in different NUMA nodes (but in the same socket), and a Broadcast (*osu_bcast*). For both tests, the point-to-point layer will be utilized; directly, for the point-to-point test, and indirectly through OpenMPI's *tuned* component for the Broadcast one. OpenMPI delegates the underlying data transportation to the SMSC component, which can be configured to use either XPMEM, CMA, or KNEM. Furthermore, by completely disabling SMSC, we obtain a measurement with the copy-in-copy-out (CICO) scheme. The results are shown in Fig. 3, reporting the performance with each of the four mechanisms, across multiple medium to large message sizes.



(a) Point-to-Point (2 ranks) (b) Broadcast (64 ranks)

Fig. 3. Performance with different data copy schemes (Epyc-2P)

Looking at the solid bars initially, XPMEM outperforms the alternative single-copy approaches, across all tested message sizes, reaching speedups up to $1.6x$ over KNEM and $6.6x$ over CMA in the broadcast test. Furthermore, we notice that all single-copy mechanisms outperform the CICO one, in almost all scenarios. XPMEM achieves a speedup of $9.5x$ over CICO in broadcast.

A second result, shown as a dashed outline, reports XPMEM's latency in the scenario that the registration cache is disabled. In that case, XPMEM's overhead is "paid" at every operation, instead of only once when the buffer is mapped for the first time; XPMEM's performance worsens drastically, rendering it worse than both of the alternatives. This result further highlights the importance of registration caching when XPMEM is leveraged. In the following sections, we therefore always complement XPMEM with a registration cache.

D. Copy-in-Copy-out

For small messages, the overhead of the extra copy under the copy-in-copy-out (CICO) scheme is minimal, and thus the benefit of performing single-copy transfers would be insignificant. Switching to a copy-in-copy-out path for these messages allows avoiding XPMEM's overheads, without any performance degradation. In fact, even if an XPMEM mapping has already been established, a single-copy transfer might still end up being slower than a CICO-based one. The reason for this is the overhead of the required registration cache lookup. In an experimental scenario for small messages, we observed that the time for this lookup was comparable to the actual data-copy time, comprising (for some ranks) about half of the overall copy time. In XHC, all collective operations below certain size threshold are performed using a CICO path.

E. Synchronization

XHC implements each collective primitive *directly*, without relying on the point-to-point layer. This entails that synchronization must be handled explicitly. XHC keeps internally a number of control flags, used to synchronize progress between the different MPI processes. The flags reside in shared memory, and follow the single-writer paradigm, i.e. they have a single owner-writer, while one or more other processes only read from them. Furthermore, the flags are carefully placed on different cache lines, where that is necessary in order to avoid false sharing phenomenons. This scheme has also been studied in [18]. Since multiple processes access each flag, read- and write-memory barriers are used to ensure proper ordering. Thanks to the single-writer technique, no exclusive access and no atomic operations are required when interacting with the synchronization flags.

To demonstrate the importance of avoiding atomic operations, we devise a simple test; we employ an implementation of MPI Broadcast, based on a flat tree with communication over shared-memory. Initially, the implementation uses synchronization flags controlled following to the single-writer method. We modify the control path, to instead rely on atomic operations and specifically on an *atomic_fetch_add*. The test is run on ARM-N1. Fig. 4 shows the performance of the two approaches as the number of processes in the node increases, which leads to increasingly more contention. At full node occupancy (160 ranks), the atomics-reliant version is worse by a factor of $23x$.

IV. Implementing XPMEM-based Hierarchical Collectives

A. MPI Broadcast

For the Broadcast collective, we implement a hierarchical, pipelined, pull-based algorithm, to move data from the root,

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

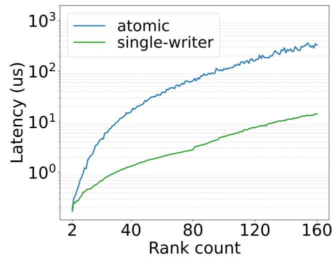


Fig. 4. MPI Broadcast (4 bytes) performance with atomics versus single-writer–based synchronization (ARM-N1)

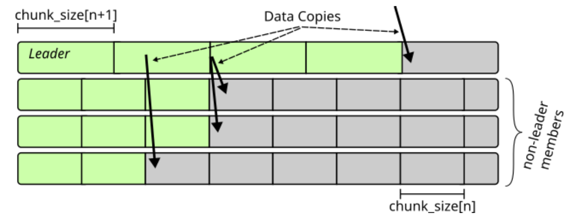


Fig. 5. Dissection of hierarchical & pipelined Broadcast.

to the leaders, and towards the leaf ranks. At the beginning of the operation, leaders expose their buffer by sharing its address with their children. An accompanying shared counter, controlled by the leader, lets the children know how many bytes are currently available to be copied. Each rank attaches to its parent's buffer and waits on the parent's counter until a chunk is available, at which point it proceeds to copy it. If the rank is a leader on one or more levels of the hierarchy, it updates its respective shared counter, thus notifying its own children of the newly available data. For the finalization stage, before exiting the operation, a hierarchical acknowledgment step takes place. Each rank that finishes receiving data adjusts a personal *acknowledgment* flag, signaling its completion. Leaders monitor their children's flag, waiting for them to finish, ensuring that the synchronization structs and the buffer are no longer is use, and can be released.

Fig. 5 illustrates an example XHC Broadcast, on some level n of the hierarchy. The leader of the level has copied 4 chunks from its leader on level $n+1$ and currently copies the next one. The children copy from the leader's buffer and into their own, according to their respective progress. Note also, that in this example the chunks sizes on levels n and $n+1$ are different, as we discussed is possible in Section III-B.

B. MPI Allreduce

XHC's Allreduce implementation performs a reduce to an internal root, followed by a broadcast that overlaps with the reduce in a pipelined fashion. The algorithm's core methodology is described below.

Step 1: Preparation: All members of an XHC communicator share the address of their *sbuf* (send buffer parameter to MPI_Allreduce) which contains the source data to be reduced. Additionally, leaders share their *rbuf* (receive buffer). Each member initializes a *reduce_ready* counter, to indicate the amount of data readily available for reduction.

Step 2a: Intra-group reduction: Each non-leader member of an XHC communicator takes on a number of data indices, and assumes the responsibility of reducing the data of all members on those indices. For each such index, the member fetches data from every peer (at that index), reduces it, and places the result on the leader's *rbuf*. The member adjusts a *reduce_done* counter to signal its reduction progress. This step is shown in phases 1 and 2 of Fig. 6.

Note that a minimum limit applies to the number of indices that a member can take up. Thus, depending on the amount of data, some members might not perform any reductions. For example, with a single or only a few elements, only one member in each group will reduce.

Step 2b: Propagation: The leader of each XHC communicator monitors the members' *reduce_done* counters in a polling manner. When he detects that the intra-comm reduction process has produced a new result, he adjusts the *reduce_ready* counter that he himself keeps on the next level of the hierarchy, thus letting its peers on that level know that new data is available for reduction. In Fig. 6, this step occurs as part of the end of phase 1.

Steps 2a & 2b repeat for every level of the hierarchy until the (now fully reduced) chunk reaches the top-level leader (i.e. the internal root).

Step 3: Broadcast: Once the chunk reaches the top, the root adjusts a counter, triggering a broadcast operation. The broadcast step is implemented in the same way as our MPI Broadcast, and is shown -in the context of Allreduce- in phases 3 and 4 of Fig. 6.

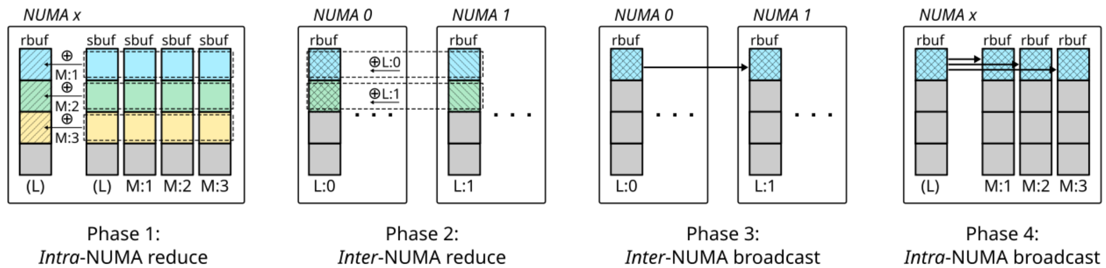


Fig. 6. Hierarchical Reduce+Broadcast Allreduce

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

C. Copy-In-Copy-Out Path

As discussed in Section III-D, XHC uses a copy-in-copy-out (CICO) scheme for collectives below a certain size threshold. The threshold is run-time configurable and defaults to 1 KB. Each process allocates a shared memory segment at communicator creation. Ranks attach to their peers' segments as necessary, and cache the attachment throughout the communicator's lifetime. CICO buffers are used to exchange data between ranks, instead of direct attachments to application buffers. The algorithms themselves remain the same as in the single-copy path, with only minor adjustments in order to accommodate for the alternate source and destination buffers.

V. Experimental Evaluation

A. Benchmarks

For our evaluation, we rely on both micro-benchmarks and full-fledged MPI applications. For micro-benchmarks, we use the OSU suite [30] (version 5.8), which offers a microbenchmark for every MPI collective primitive. These microbenchmarks initially perform a set of warmup runs and then multiple actual iterations, reporting their mean latency.

However, we have identified a shortcoming in the microbenchmarks; the same buffer is passed at every iteration, without any modification. As a result, ranks will implicitly cache the buffer's data in their local cache, and will be able to retrieve it rapidly in subsequent calls. In Section III-A we showed how accesses across topology domains are slower than local ones. Furthermore, localized accesses avoid contention for bandwidth, especially for fan-in or fan-out patterns. If data originating from remote peers is cached locally, the effects of these accesses are concealed. Operating under the assumption that actual applications do not send the same message multiple consecutive times, the benchmark measures the performance of an unrealistic scenario. The importance of cache effects and their implications to MPI performance benchmarking has also been studied in [31], [32]. To better emulate the behaviour of actual applications, we modify the microbenchmarks to alter the buffer's contents before each call.

In Fig. 7, we perform a Broadcast benchmark, and demonstrate the difference between the reported performance of the original micro-benchmark (osu_bcast) and of our variant of it (osu_bcast_mb). The graph shows the latency of two Broadcast implementations in XHC: one based on a flat tree and a hierarchical one (labeled as *tree*). Focusing initially on the performance of the flat tree with either benchmark, we notice two trends: the performance changes drastically for medium to large messages (2 KB to 1 MB), while it remains the same for small messages (less than 1 KB) and for larger ones (more than 1 MB). For the 2KB–1MB range, we observe how the flat tree's performance greatly benefits thanks to the caching effects that result in data retrieval from the local cache. For messages of 1 KB or less in size, we do not observe any difference; this is the range for which XHC performs copy-in-copy-out transfers. The copy-in phase modifies the buffer from which non-root ranks will copy, in the same way that our microbenchmark variant does in the single-copy scenario. For larger messages, above 1 MB, the buffer's data no longer fit in the system's cache, so reaping its benefits is not possible. For the hierarchical tree, the performance also does not change across the message range. This implementation does not benefit from the described caching effect. Instead, a similar situation to the one with copy-in-copy-out in the flat tree occurs: when a leader copies the root's data into his own buffer, eviction of the leader's buffer from his children's caches is triggered. Finally, comparing the two Broadcast implementations, we note that the cache-unaware benchmark leads to inaccurate conclusions, as it shows the flat tree performing better than the hierarchical one. However, our variant reveals that the inverse is in fact true.

TABLE I
Evaluation systems

Codename	Processor	Arch	Cores	NUMA	Sockets
Epyc-1P	1x AMD Epyc 7551P	x86_64	32	4	1
Epyc-2P	2x AMD Epyc 7501	x86_64	64	8	2
ARM-N1	2x ARM Neoverse N1	arm64	160	8	2

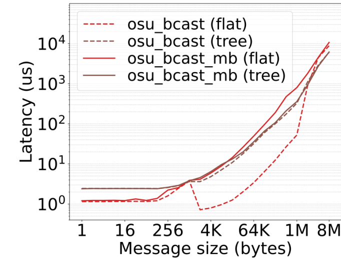


Fig. 7. Comparison of osu_bcast variants (2x AMD Epyc 7501)

In the rest of our evaluation, all performance figures from micro-benchmarks are obtained using our variants that alter the transmitted data before each iteration, and we report the average over 10 runs.

For evaluation using MPI applications, we rely on miniAMR [33], CNTK [34], and PiSvM [35]. MiniAMR is an application that mimics Adaptive Mesh Refinement (AMR) workloads and their communication patterns. The recurring refine process of miniAMR utilizes the MPI Allreduce operation. CNTK is a toolkit for distributed deep learning that implements Stochastic Gradient Descent (SGD), using the Allreduce primitive. Finally, PiSvM is a parallel implementation of the Support Vector Machine (SVM) algorithm. The majority of PiSvM's MPI communication time is inside MPI Broadcast.

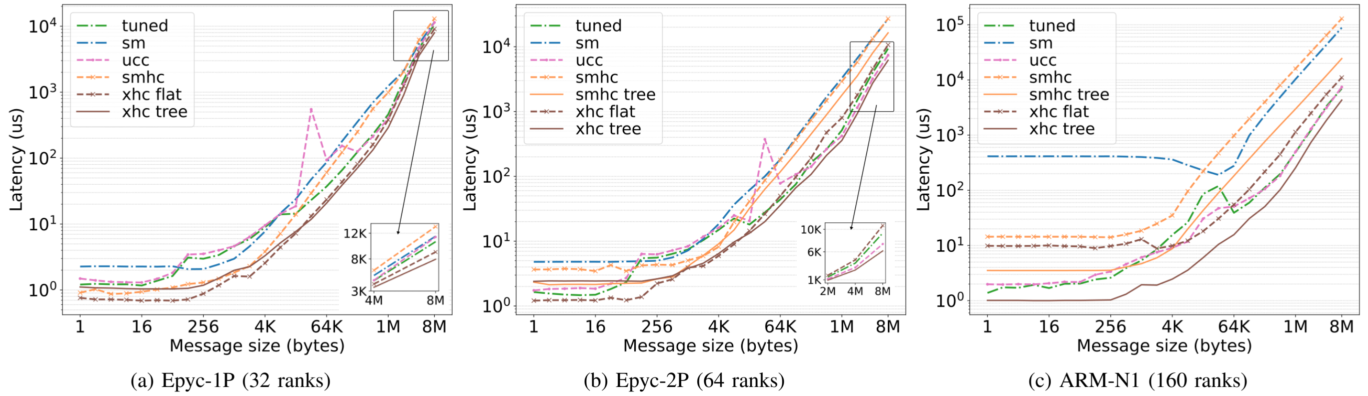
B. Evaluation platforms

We evaluate our implementation on multiple platforms, including one single-socket and one dual-socket AMD Epyc system, as well as, a dual-socket ARM-based one. Table I shows the systems and their internal features in detail.

C. Relevant collectives frameworks

To evaluate the performance of XHC, we initially compare with the various collectives components in OpenMPI. One

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.



(a) Epyc-1P (32 ranks)

(b) Epyc-2P (64 ranks)

(c) ARM-N1 (160 ranks)

Fig. 8. Performance comparison – MPI Broadcast

of them is *sm*: an implementation of collectives that uses shared memory and does not rely on point-to-point primitives. Another is *tuned*, which implements collectives using point-to-point communication primitives. We pair tuned with *Unified communication X* (UCX) [36] for the underlying implementation of the point-to-point layer. This is also the default setting in OpenMPI. The last component, *ucc*, delegates the implementation of collectives to the *Unified Collective Communication* (UCC) [37] library. Both UCX and UCC will take advantage of XPMEM for single-copy transfers.

Besides OpenMPI's components, we implement and compare with two frameworks from recent studies. They are assigned indicative names to ease reference to them. The first scheme, *Shared-Memory–based Hierarchical Collectives* (SMHC) is based on the design presented in [18]. It utilizes shared memory for both synchronization and data transportation. We consider two variants of it: a flat one, and a hierarchical one that takes into account different processor sockets. On Epyc-1P, only the flat variant is considered, as this system only has a single socket. The second framework, *XPMEM-Based Reduction Collectives* (XBRC) is based on [5]. This work suggests shared-address space implementations for the Reduce and Allreduce operations, leveraging XPMEM's single-copy capabilities. In this paper, we focus on the intra-node phase of the Allreduce implementation. Note that our implementations are as close as possible to the descriptions in the corresponding studies, and we have made the source code available [38].

We consider two XHC variants: XHC-flat, with no hierarchy, and XHC-tree, which is *numa+socket*-aware. XHC-tree results in a 3-level hierarchy for Epyc-2P and ARM-N1, and in a 2-level hierarchy for Epyc-1P, which only has one socket.

D. Evaluation results

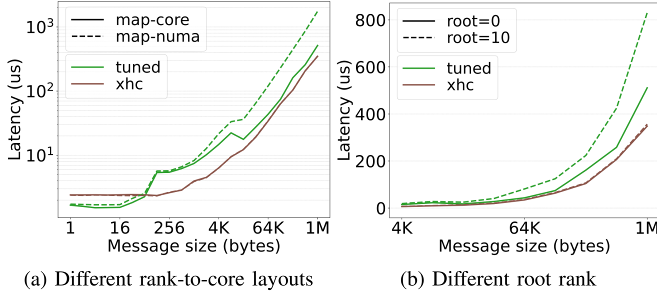
1) MPI_Bcast: In Fig. 8, we compare XHC with all aforementioned frameworks in terms of Broadcast latency. Focusing initially on medium to large message sizes, XHC-tree achieves lower latency than all other schemes. Compared to shared-memory copy schemes like SMHC and OpenMPI-sm, it relies on a single copy to transfer the message. Against SMHC(-tree), XHC-tree achieves speedups up to $4x$, $5x$, and $15x$, on Epyc-1P, Epyc-2P, and ARM-N1, respectively. Compared to tuned and ucc, XHC-tree manages speedups up to $2x$ on Epyc-1P and Epyc-2P, and up to $2.5x$ on ARM-N1. One key contributing factor to XHC's better performance, is that it explicitly takes node topology into account, in order to limit messages to distant NUMA nodes and sockets, and avoid congestion. OpenMPI's sm and tuned, as well as ucc, use static, predetermined communication schedules, that might not be the best fit to the underlying physical topology. The effect of access to distant domains and congestion on performance has also been discussed in Section III-A.

To further explore the effects of mismatch between the implementation's communication pattern and the platform's topology, we test the broadcast performance of OpenMPI's *tuned*, in scenarios that alter the traffic pattern. Specifically, we perform a micro-benchmark with two different policies for mapping ranks to available cores: one that assigns them to cores sequentially (map-core), and one that assigns them to different NUMA nodes in a round-robin manner (map-numa). Fig. 9a shows the results, with XHC-tree, a topology-aware component, as a reference. We observe that XHC-tree's performance remains robust with both policies, while that of tuned varies significantly. For large messages, there is a performance difference of up to $3.4x$ between the two scenarios. Besides the mapping policy, another factor that can affect the communication pattern is the specified root rank. In Fig. 9b we show the results of a test that alters the root. Repeating the trend, tuned's performance swings by up to $1.6x$, while XHC-tree's remains the same.

The aforementioned phenomena becomes clearer when the number of messages that cross different NUMA or socket domains are considered. Table II depicts these message counts for the scenarios explored in Fig. 9. For the case of broadcast with rank 10 as the root instead of 0, the tuned component has significantly more inter-socket and inter-numa messages, while XHC-tree's pattern remains consistent.

An interesting observation is how the benefit of XHC scales

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.



(a) Different rank-to-core layouts (b) Different root rank

Fig. 9. Broadcast: different rank-to-core layouts and root rank (Epyc-2P)

TABLE II
NUMBER AND DISTANCE OF EXCHANGED MESSAGES (EPYC-2P)

Component	Scenario	Inter-Socket	Inter-NUMA	Intra-NUMA
tuned	map-core	2	9	52
tuned	map-numa	8	48	7
tuned	root=0	2	9	52
tuned	root=10	10	17	36
XHC-tree	any	1	6	56

on different platforms. The expectation is that the performance gap increases when moving to denser platforms, with more cores and more topological domains. In accordance with this theory, XHC-tree reaches over XHC-flat speedups up to $1.32x$, $2.23x$, and $4.46x$, on Epyc-1P, Epyc-2P, and ARM-N1, respectively. The highest speedups over all other components are observed on our largest system, ARM-N1.

Focusing on small messages, another interesting phenomenon emerges. Contrary to our expectation, on Epyc-1P and Epyc-2P, XHC-flat does better than XHC-tree. Generally speaking, the hierarchy incurs a certain overhead; progress on any level depends on progress in the previous one. The cumulative overhead will be higher when more levels are present, and this is observable going from Epyc-1P (2 levels), to Epyc-2P (3 levels). Normally, the benefit of the hierarchy is greater than the overhead; the question arises why this is not the case here. We attribute the success of XHC-flat in small messages, to implicit assistance from the processor's internal cache hierarchy. A major deciding factor in the performance of especially-small messages is synchronization. In XHC, a single shared flag is used to signal progress during the operation; the leader writes the flag (single writer), while the children (multiple readers) read it. Because in Epyc-1P and Epyc-2P 4 cores share an L3 cache, one of the four cores will bring the flag into the shared cache, where its cache-peers will be able to retrieve it, instead of fetching it from the root. This effectively increases locality, and reduces congestion at the root. In this way, XHC-flat essentially achieves in hardware what XHC-tree offers in software. At this point, adding hierarchy, as XHC-tree does, affords no extra benefit while introducing overhead.

To confirm our hypothesis, we perform the following test: we derive variants of XHC-flat and XHC-tree where the single flag used at each level to synchronize the root/leader with the members is replaced with multiple flags, one for each member. We consider a scenario where these flags reside in the same cache line, and one where they occupy different ones. As long as the flags of neighboring cores are in the same cache line, the phenomenon that we observe with the single flag should still occur. In Fig. 10 we observe that when multiple flags are in the same cache line (shared), the scenario closest to the actual design of XHC, the flat configuration achieves lower latency than its hierarchical counterpart. However, when each flag is in its own cache line (separated), the trend reverses: the latency of the flat variant increases drastically, rendering it worse than the hierarchical one. The latency of the hierarchical version remains similar with either scheme; its explicit handling of flags traversal through the system's topology leaves minimal margin for implicit assistance.

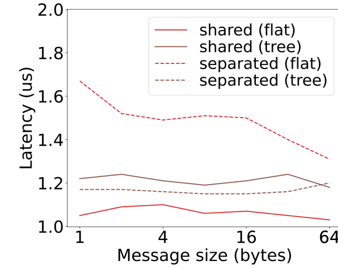


Fig. 10. Broadcast: Comparison of cache-line sharing schemes (Epyc-1P)

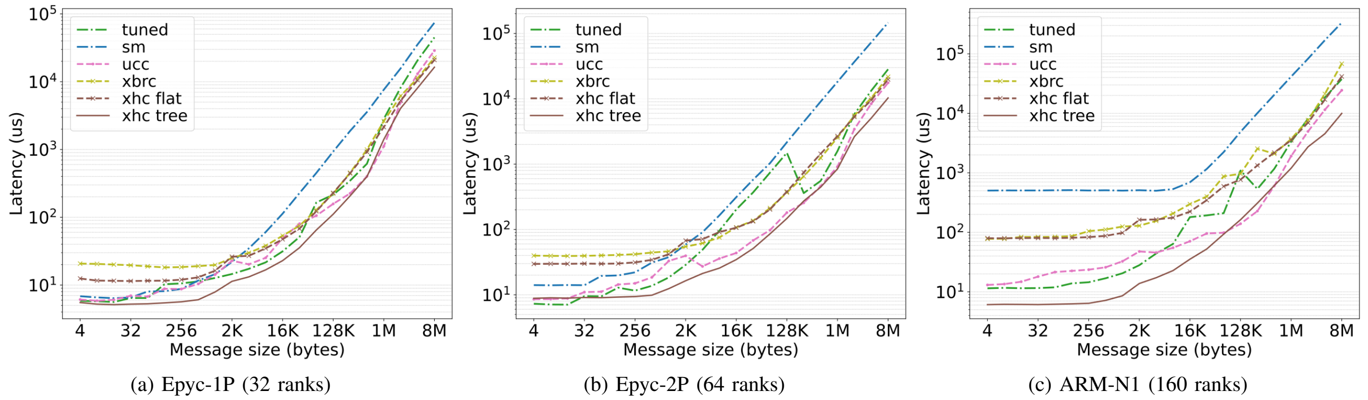
Interestingly, the same performance pattern regarding small size messages is not observed on ARM-N1. Here, XHC-tree performs better than all other components, and XHC-flat has significantly elevated latency. The difference is that this system does not possess the same hierarchical cache structure that the other two systems do. On ARM-N1 [39], each core has a private L1 and L2 cache, but there are no L3 caches shared between neighboring cores. Instead, cores have access through the *Coherent Mesh Network (CMN-600)* [40], to a physically tagged system-level cache, where each address is cached at a single location. Therefore, better locality is not possible implicitly in the way that it is on Epyc-1P and Epyc-2P. In the case of XHC-flat, the root's singular control flag will be located at a single point, and the fan-in pattern will result to contention, bloating latency. XHC-tree avoids this phenomenon, by partitioning communication and thus distributing the load to multiple locations.

Finally, as Fig. 8 also shows, we notice that OpenMPI's sm exhibits exceedingly high small-message latency on ARM-N1. Through analysis and timing of the component's individual phases, we find that this is due to the use of the *atomic_fetch_add* operation for one of the control flags. This result corresponds to our findings in the scenario explored in Section III-E. Consequently, atomics-based synchronization schemes are prohibitive for dense nodes like ARM-N1.

2) MPI_Allreduce: In Fig.11, we present the Allreduce performance of XHC compared to the relevant components.

Focusing initially on small messages, XHC-tree achieves better performance than all other schemes, with the exception

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.



(a) Epyc-1P (32 ranks) (b) Epyc-2P (64 ranks) (c) ARM-N1 (160 ranks)

Fig. 11. MPI Allreduce performance

of messages sized 4-32 bytes on Epyc-2P. According to the design of XHC's algorithm for small messages, only one rank performs reductions on each level of the hierarchy, which introduces a degree of linearization; this inefficiency is responsible for XHC-tree's slightly higher latency, that ranks it the second position after tuned. We also notice that, in contrast to the broadcast primitive, XHC-flat does worse than XHC-tree on all platforms. XHC-flat is also affected by linearization; however, the singular level of its (flat) hierarchy will contain considerably more ranks and therefore more buffers to be reduced. This amplifies the phenomenon, and significantly elevates XHC-flat's latency. Furthermore, implicit assistance from the system's cache hierarchy is not possible in reduce as it was in broadcast; in reduce, each rank's source data resides in a distinct buffer, and the reducer-rank needs to fetch all of them. The performance of XBRC is quite similar to that of XHC-flat. Its ranking with respect to XHC-flat is expected since it does not limit memory accesses across different NUMA nodes or sockets. Finally, as in the evaluation of broadcast, *sm* does especially poor on ARM-N1, as it again relies on atomic operations for synchronization.

Switching our focus to medium messages, XHC-tree does better than all other components at the low-end of the range. For some message sizes between $128K$-$1M$, depending on the system, ucc performs as well or slightly better than XHC-tree. This is attributed to sub-optimal configuration of XHC's pipeline for these messages; fine-tuning the chunk size with the size of the messages in mind alleviates the shortcoming. For larger messages, XHC-tree re-gains the lead. On ARM-N1, XHC-tree achieves speedups up to $4x$, $2.5x$, and $6.8x$, against tuned, ucc, and xbrc, respectively. On the two other systems, XHC reaches speedups of $1.5x$ and $1.75x$, over the next best component, on Epyc-1P and Epyc-2P, respectively. Both XBRC and XHC-flat surpass tuned and ucc on Epyc-1P, but fail to do so in larger systems where the effects of distant memory accesses are amplified.

3) Evaluation with MPI Applications: In this section, we compare XHC's performance with the schemes discussed so far, using full MPI applications.

In Fig. 12, we showcase **PiSvM**'s performance with the various components. Note that, on Epyc-1P, SMHC uses a flat tree, as this system only has a single socket. We use the *mnist_train_576_rbf_8vr* dataset, with PiSvM's default configuration. On ARM-N1, XHC-tree achieves speedups of $1.13x$, $1.57x$, and $3.6x$, over tuned, ucc, and smhc, respectively. On Epyc-1P and Epyc-2P, XHC's performance is on par with tuned, while it does slightly better than ucc (5-8% improvement). SMHC does respectably on Epyc-1P, but fails to keep up in the larger systems.

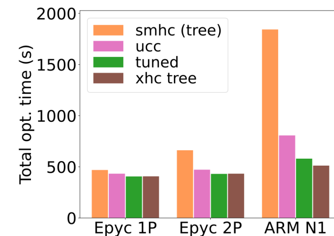


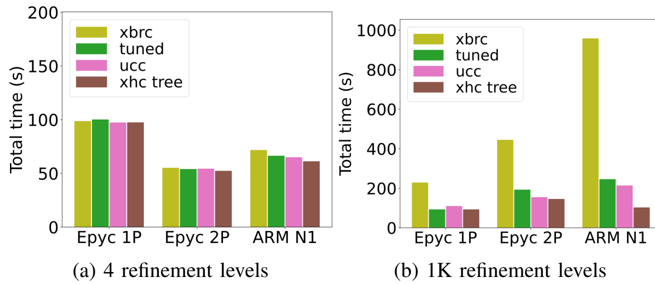
Fig. 12. PiSvM performance

It is worth mentioning that the benefit of XHC over other schemes is indeed expected to be more mild in full MPI applications, compared to microbenchmarks. This benefit depends on the total time the application spends inside the supported collectives. For example, we see that XHC-tree "only" achieves a speedup of $1.13x$ over tuned in PiSvM on ARM-N1; however, with the help of an MPI profiler we find that it actually reduces the total time spent in MPI_Bcast by a factor of 2.

Proceeding, we benchmark **miniAMR**'s performance with the various components. We run the "expanding sphere" example, initially with the default configuration and for 400 time-steps. This result is shown in Fig. 13a. We also perform a run with $1K$ refinement levels, with the refine frequency set to 1 time-step, for a total of 1000 time-steps; this run's results are shown in Fig. 13b. The latter configuration is expected to be more challenging to the Allreduce implementation; the actual set of parameters and therefore the degree to which the Allreduce operation affects the overall performance, will depend on the specific AMR application. With the first con-

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

figuration, the Allreduce operations concern small messages, averaging a couple tens of bytes per call. On Epyc-1P and Epyc-2P, the improvement with XHC is marginal to none. However, on the largest system (ARM-N1), the differences are more definite, and XHC-tree achieves speedups of 7.4%, 5.6%, and 14.5%, over tuned, ucc, and xbrc, respectively. With the second configuration, the Allreduce calls are $1KB$ in average. On Epyc-1P, XHC achieves $1.2x$ speedup over ucc, and on Epyc-2P, $1.3x$ over tuned. On ARM-N1 the differences are larger, with speedups of $2.1x$ and $2.4x$ over ucc and tuned, respectively. Finally, XBRC struggles in this configuration, with XHC's speedup ranging from $2.5x$ to $9x$. These results highlight the importance for efficient implementation not only for large messages, but also for small to medium ones.



(a) 4 refinement levels (b) 1K refinement levels

Fig. 13. MiniAMR performance (An expanding sphere)

Finally, we measure the performance of training the AlexNet neural network with **CNTK**, using the ImageNet ILSVRC12 dataset [41]. Originally, CNTK utilizes the non-blocking variant of Allreduce, Iallreduce, which neither ours nor the other components we compare with implement. Through analysis of the code, we determine that this operation can be replaced with the standard allreduce without sacrificing performance. We also confirm this by benchmarking CNTK's performance using OpenMPI's non-blocking collectives component. Because AlexNet is a large model, we run the training for one epoch only, of size $50K$. Fig. 14 shows the results. On ARM-N1, XHC-tree reduces the training time by 14.5%, 11.5%, 33.3%, over tuned, ucc, and xbrc respectively. On the other two platforms, XHC-tree achieves a 5.6% and a 9.1% reduction, on Epyc-1P and Epyc-2P respectively, over the next best performing component. By profiling CNTK's MPI calls, we observe that on ARM-N1, XHC speeds-up the time spent inside Allreduce by $2.6x$, $3x$, and $6.3x$, over ucc, tuned, and xbrc, respectively.

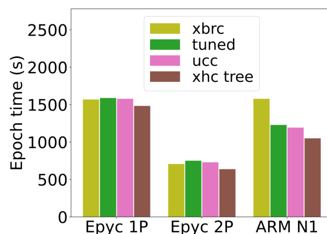


Fig. 14. CNTK performance (AlexNet ILSVRC12)

As we discussed in Section II-B, and showed in Section III-C, XPMEM attachment overheads can be detrimental to performance. However, since attachments can be reused multiple times, the cost can be amortized throughout multiple operations. Therefore, applications that consistently use the same buffer(s) for communication will see lower overall overhead and higher benefit. This indeed is the case with the HPC apps that we discussed (PiSvM, miniAMR, CNTK); profiling the registration cache reveals hit ratios above 99% for each one of them, asserting XPMEM's position as a suitable option for the transport channel.

VI. Related work

Several recent studies have addressed the challenges of designing and implementing efficient collectives at the intra-node level. A significant number of them suggest implementations that directly utilize shared memory [16]. In [17], the authors focus on Allreduce, and suggest an approach that utilizes multiple leaders, for both the reduction and the inter-node communication. In [42], optimized algorithms for several collectives are proposed, with focus on large messages.

Work in [43] discusses shared-memory–based algorithms for several collectives. Effects of node topology and process layout on performance are also considered. The authors in [18] also suggest algorithms over shared memory, with socket-aware trees to propagate flags and data. In [44], a shared-memory Broadcast algorithm with NUMA-aware notification propagation is presented. Authors in [45] present a framework for topology-aware shared-memory collectives, that decouples system hierarchy from algorithm implementation. Finally, work in [24] presents a framework for hierarchical scalable collectives, that takes node topology into account and utilizes shared memory for the on-host level. It also exploits hardware assists for both intra- and inter-node component.

In the pursuit to avoid the copy overheads associated with shared-memory approaches, multiple other studies have adopted designs that leverage single-copy mechanisms. Researchers in [23] combine the benefits of KNEM with node-level topology awareness, to derive distance-aware algorithms for Broadcast and Allgather. In [28], contention-aware kernel-assisted algorithms for all types of collective communications are proposed. Authors in [46] utilize KNEM's functionality to develop kernel-assisted and NUMA-aware collectives. They also extend KNEM, adding transfer-direction control, with the aim of reducing bottlenecking at the root process. In our work, we instead rely on XPMEM for single-copy support, due to its advantages over KNEM (discussed in Section II).

Similarly to our approach, researchers in [5] produce algorithms for Reduce and Allreduce that rely on XPMEM for single-copy intra-node exchanges, along with a registration cache to mitigate XPMEM's attach/detach overheads. This work is extended in [47], where several optimizations are applied, and designs for more primitives are introduced. A unique approach is presented in [48], where single-copy collectives are realized by leveraging SMARTMAP which allows a process to directly read/write the memory of peer processes.

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

Authors in [49] provide NUMA-aware algorithms for Allreduce, for the Hybrid MPI library (HMPI) where ranks are implemented as threads rather than processes. Shared heap data among ranks enables single copy transfers. However, as also shown in [50], thread-based MPI has applicability and performance-related shortcomings.

Differently from all of the above, this work combines the benefits of XPMEM's single copy capability with node-level topology awareness to derive hierarchical algorithms for commonly used collectives. Moreover, a shared-memory variant of each algorithm is proposed to handle small messages.

VII. Conclusions and Future Work

We have presented the XHC framework for MPI collective operations at the intra-node level, for multi-core nodes. XHC combines knowledge of internal node structure to construct hierarchical algorithms, using XPMEM for single-copy transfers, and pipelining to overlap communication at different levels of the hierarchy. We have evaluated the proposed approach through microbenchmarks and real-world MPI applications, demonstrating its effectiveness for modern and emerging nodes. Our evaluation with microbenchmarks for Broadcast and Allreduce shows speedup up to $2.5x$ and $3x$, respectively, over UCC and OpenMPI's default collectives implementation. Compared to recent research studies, we improve Broadcast by up to $5x$, and Allreduce by up to $7x$. We reduce the time of PiSvM, miniAMR and CNTK by up to 12%, 52% and 12%, respectively, over the next best-performing alternative. Our ongoing work focuses on the Reduce primitive, optimizations for Allreduce, and effects regarding Barrier. We are extending XHC towards inter-node interactions, and utilization of hardware acceleration functionality. We also consider interoperability and integration with OpenMPI's (Hierarchical AutotuNed) collectives framework and with UCX/UCC.

Acknowledgment

We thankfully acknowledge the support of the European Commission and the Greek General Secretariat for Research and Innovation under the EuroHPC Research and Innovation Programme through project DEEP-SEA (Grant agreement No 955606). National contributions from the involved state members (including the Greek General Secretariat for Research and Innovation) match the EuroHPC funding.

References

[1] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open mpi: A high-performance, heterogeneous mpi," in *2006 IEEE International Conference on Cluster Computing*, 2006, pp. 1–9.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, p. 789–828, sep 1996. [Online]. Available: https://doi.org/10.1016/0167-8191(96)00024-5

[3] J. Liu, W. Jiang, P. Wyckoff, D. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, "Design and implementation of mpich2 over infiniband with rdma support," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 16–.

[4] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, and J. Dongarra, "Han: a hierarchical autotuned collective communication framework," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 23–34.

[5] 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)*, 2018, pp. 1020–1029.

[6] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "Magpie: Mpi's collective communication operations for clustered wide area systems," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 131–140. [Online]. Available: https://doi.org/10.1145/301104.301116

[7] A. Faraj, P. Patarasuk, and X. Yuan, "A study of process arrival patterns for mpi collective operations," *International Journal of Parallel Programming*, vol. 36, pp. 543–570, 11 2006.

[8] 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: Association for Computing Machinery, 2012, p. 45–54. [Online]. Available: https://doi.org/10.1145/2145816.2145823

[9] 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*, ser. ICS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 253–262. [Online]. Available: https://doi.org/10.1145/1088149.1088183

[10] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016, pp. 1–10.

[11] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Overlapping computation and communication: Barrier algorithms and connectx-2 core-direct capabilities," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[12] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, p. 49–66, feb 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[13] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Locality and topology aware intra-node communication among multicore cpus," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 265–274.

[14] 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*, 2009, pp. 462–469.

[15] L. Chai, A. Hartono, and D. K. Panda, "Designing high performance and scalable mpi intra-node communication support for clusters," in *2006 IEEE International Conference on Cluster Computing*, 2006, pp. 1–10.

[16] 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*, 2003, pp. 10 pp.–.

[17] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. D. Panda, "Scalable reduction collectives with data partitioning-based multi-leader design," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126954

[18] S. Jain, R. Kaleem, M. G. Balmana, A. Langer, D. Durnov, A. Sannikov, and M. Garzaran, "Framework for scalable intra-node collective operations using shared memory," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 374–385.

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.

[19] D. Buntinas, G. Mercier, and W. Gropp, "Data transfers between processes in an smp system: Performance study and application to mpi," in *2006 International Conference on Parallel Processing (ICPP'06)*, 2006, pp. 487–496.

[20] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis," *Proceedings of the International Conference on Parallel Processing*, 09 2009.

[21] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The sgi® altixtm 3000 global shared-memory architecture," *Silicon Graphics, Inc*, vol. 44, 2005.

[22] Cross-process memory mapping (xpmem). [Online]. Available: https://github.com/hjelmn/xpmem

[23] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, "Process distance-aware adaptive mpi collective communications," in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 196–204.

[24] R. Graham, M. G. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer, "Cheetah: A framework for scalable hierarchical collective operations," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011, pp. 73–83.

[25] Linux cma: Cross memory attach. [Online]. Available: https://lwn.net/Articles/405284

[26] H.-W. Jin, S. Sur, L. Chai, and D. Panda, "Limic: support for high-performance mpi intra-node communication on linux cluster," in *2005 International Conference on Parallel Processing (ICPP'05)*, 2005, pp. 184–191.

[27] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Lightweight kernel-level primitives for high-performance mpi intra-node communication over multi-core systems," in *2007 IEEE International Conference on Cluster Computing*, 2007, pp. 446–451.

[28] S. Chakraborty, H. Subramoni, and D. K. Panda, "Contention-aware kernel-assisted mpi collectives for multi-/many-core systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 13–24.

[29] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa Italie, 02 2010. [Online]. Available: https://www.open-mpi.org/papers/pdp-2010

[30] Osu micro-benchmarks. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks

[31] W. Gropp and E. L. Lusk, "Reproducible measurements of mpi performance characteristics," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 1999, p. 11–18.

[32] S. Hunold and A. Carpen-Amarie, "Reproducible mpi benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[33] C. T. Vaughan and R. F. Barrett, "Enabling tractable exploration of the performance of adaptive mesh refinement," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 746–752. [Online]. Available: https://github.com/Mantevo/miniAMR

[34] The microsoft cognitive toolkit. [Online]. Available: https://docs.microsoft.com/en-us/cognitive-toolkit

[35] Pisvm. [Online]. Available: http://pisvm.sourceforge.net/

[36] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[37] "Unified collective communication (ucc)," https://ucfconsortium.org/projects/ucc/, unified Communication Framework.

[38] Implementations of recent studies – smhc, xbrc. [Online]. Available: https://github.com/CARV-ICS-FORTH/XHC-OpenMPI

[39] Ampere® altra® 64-bit multi-core processor features. 80x Armv8.2 cores: https://amperecomputing.com/processors/ampere-altra/. [Online]. Available: https://d1o0i0v5q5lp8h.cloudfront.net/ampere/live/assets/documents/Altra_Rev_A1_DS_v1.27_20220331.pdf

[40] Arm corelink cmn-600 coherent mesh network technical reference manual. [Online]. Available: https://developer.arm.com/documentation/100180/0302/?lang=en

[41] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[42] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda, "Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2008, pp. 130–137.

[43] R. L. Graham and G. Shipman, "Mpi support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 130–140.

[44] M. Kurnosov and E. Tokmasheva, "Shared memory based mpi broadcast algorithms for numa systems," in *Supercomputing*, V. Voevodin and S. Sobolev, Eds. Cham: Springer International Publishing, 2020, pp. 473–485.

[45] B. Ramesh, J. M. Hashmi, S. Xu, A. Shafi, M. Ghazimirsaeed, M. Bayatpour, H. Subramoni, and D. K. Panda, "Towards architecture-aware hierarchical communication trees on modern hpc systems," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 272–281.

[46] 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*, 2011, pp. 532–541.

[47] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Design and characterization of shared address space mpi collectives on modern architectures," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 410–419.

[48] R. Brightwell and K. Pedretti, "Optimizing multi-core mpi collectives with smartmap," in *2009 International Conference on Parallel Processing Workshops*, 2009, pp. 370–377.

[49] 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: Association for Computing Machinery, 2013, p. 85–96. [Online]. Available: https://doi.org/10.1145/2462902.2462903

[50] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid mpi: Efficient message passing for multi-core systems," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

Authorized licensed use limited to: Foundation for Research and Technology (FORTH). Downloaded on November 29,2022 at 13:06:56 UTC from IEEE Xplore. Restrictions apply.