2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00041

# Towards Architecture-aware Hierarchical Communication Trees on Modern HPC Systems

Bharath Ramesh, Jahanzeb Maqbool Hashmi, Shulei Xu, Aamir Shafi, Mahdieh Ghazimirsaeed,
Mohammadreza Bayatpour, Hari Subramoni, Dhabaleswar K. Panda
*Department of Computer Science and Engineering*
*The Ohio State University*
Columbus, USA
{ramesh.113, hashmi.29, xu.2452, shafi.16, ghazimirsaeed.3, bayatpour.1, subramoni.1, panda.2}@osu.edu

***Abstract*—Modern HPC systems built with emerging multi-/many-core architectures have high core-counts and deep memory hierarchies. It is challenging to design communication libraries on these systems with the conventional wisdom of using OS processes as the basic building block to build communication algorithms. Instead, the next generation of communication libraries should treat hardware as the "first-class citizen" and utilize the underlying topology as the basic building block. Driven by this overarching principle, we present a framework for Optimized Shared Memory Processing (OSMP) and communication for these platforms. An abstract representation of the underlying hardware topology is maintained by OSMP in the form of a topology tree, which is later exploited by runtime libraries to execute communication operations in a topology-aware manner. This can be done by simply traversing the topology tree with an existing communication primitive as the base-case. OSMP does not mandate any changes to the original communication algorithm. We focus on collective operations such as barrier, reduction, and broadcast as candidate communication patterns. We demonstrate the efficacy of OSMP by decoupling the implementation of collective algorithms and system topology and evaluate it on four state-of-the-art multi-/many-core architectures : Intel Cascade Lake, AMD Rome, ARM A64fx and IBM POWER9. Results show that even the basic algorithms can be made topology-aware by exploiting OSMP. This provides significant benefits over state-of-the-art algorithm implementations for intra-node communication. Using various micro-benchmarks and applications, we demonstrate that our proposed designs can achieve up to 7.8× improvements at the micro-benchmark level, and 15% for applications over state-of-the-art intra-node collective communication designs employed by production MPI libraries.**

***Index Terms*—Multi-Core, Many-Core, NUMA, Cache Coherence, SMP, SMP topology, Shared Memory, MPI**

## I. Introduction

The next generation of HPC systems are mainly driven by emerging multi-/many-core processor architectures, which allow them to offer higher core-density, deeper memory hierarchies, and diverse architectural features. The adoption of multi-/many-core hardware is becoming widespread at an increasing pace. An evidence of this is the recently launched SDSC's Expanse system equipped with dual 64-core (AMD Rome) processors—offering an impressive 128 cores in a single node. This trend towards the adoption of multi-/many-core processors is expected to continue as the HPC community is edging closer towards the exascale systems like Aurora [1] and Frontier [2]. This uptake also puts the burden of scaling user applications on communication runtime libraries as intra-node optimizations are likely to dominate much of the system optimizations. These optimizations are vital to maintain the dominance of the Message Passing Interface (MPI) standard as the *defacto* parallel programming API over other alternatives. The hybrid programming model—also known as MPI+X (X being any shared memory model)—might offer competitive alternative although it requires modifying user applications that is a major hindrance in its adoption. Pure shared memory approaches like OpenMP [3], however, are attractive on these newly emerging platforms especially for "long tail" of science applications—these represent small-sized scientific workloads that result in significant scientific discoveries.

The communication libraries, such as MPI, OpenSHMEM, and others, expose an OS process to applications as a logical processing element (PE) and provide it a numeric identifier—also called a process rank. These applications, in turn, use these processes as basic functional units to construct parallel algorithms (e.g., near-neighbor communication in stencils) and higher-level communication primitives (e.g., MPI collectives). While this abstraction—of representing PEs as processes with ranks—provides a powerful mechanism to implement high-level algorithms, the performance of such algorithms on modern HPC systems is dependent on the topology of the underlying architecture. For instance, on an architecture with two sockets within the system, an algorithm that performs a *gather-at-root* operation by iteratively receiving data from every *non-root* process would perform well if all processes are within the socket but not if processes are spread out across sockets. Also, high-level algorithms could rely on the assumption that processes with rank numbers closer to each other also have lower core-to-core latency, which need not be the case. While binding processes to cores appropriately using topology detection tools such as *hwloc* [4] could be a potential solution to this problem, different algorithms and/or applications would require different process-to-core bindings to perform efficiently, which adds to the complexity. This

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00041

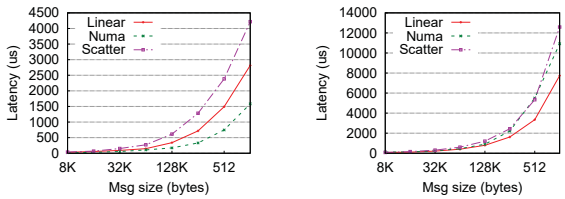
brings us to the question: ***Can we define a set of abstractions that can offer the convenience of defining algorithms at a high-level whilst treating hardware as the first-class citizen?***

The performance of inter-process communication (IPC) depends on the core-to-core communication latency between threads/processes running on the physical cores of the system and any unfavorable communication pattern can lead to performance penalties. In order to address this performance limitation, topology-aware design patterns are proposed in the literature [5], [6]. Parallel applications typically rely on an underlying communication runtime such as MPI to perform communication operations as efficiently as possible. The MPI standard provides support for virtual topologies including graph and cartesian topologies [7]. This provides an opportunity for application developers to utilize these virtual topologies and create MPI communicators to efficiently orchestrate their communication patterns on a given architecture. However, this approach requires application developers to have a deep understanding of the underlying architecture and mandates changes to the code, which is undesirable.

In the context of communication runtimes such as MPI, collective communication algorithms exhibit various communication patterns. For example, a *ring-based* MPI_Allreduce algorithm has different communication characteristics than a *recursive-doubling* based algorithm. Past work has shown that redesigning MPI collectives using topology-aware design patterns yield significant performance benefits for scientific applications [5], [6]. The broad idea is to orchestrate collective algorithms in such a way that the memory accesses are localized and cross-link (expensive) traffic is minimized. For instance, a NUMA-aware MPI reduction will first accumulate result from the ranks residing on local NUMA domains before reducing the data from ranks on different NUMA domains. However, due to lack of modularity in MPI libraries, introducing new topology-aware design-patterns for newer architectures requires significant effort. This leads us to the next question: ***Can we design a framework that provides highly efficient topology-aware implementations of communication operations that can be integrated with existing communication runtimes as well as be used standalone by applications with minimal changes to code?***

## II. Motivation

We motivate the need for architecture-aware designs—and hence our work—by demonstrating the lack of adaptability of MPI (and other programming models) to emerging multi-/many-core architectures. First, we use the osu_allreduce benchmark from OSU Micro-benchmarks [8] to evaluate the performance of two different allreduce algorithms namely; 1) reduce-scatter-allgather, and 2) recursive-doubling. Figure 1 presents the results on a dual-socket AMD EPYC 7742 (Rome) system. The idea here is to emphasize the sensitivity of these two allreduce algorithms—or communication patterns—to different process-to-core mapping policies on a modern multi-/many-core HPC system, showing a discrepancy between the topology and the high-level implementation of the algorithms.




(a) Reduce Scatter (b) Recursive Doubling

**Figure 1:** Performance of reduce-scatter-allgather and recursive-doubling based MPI_Allreduce for 64 processes on AMD Rome using Linear, Numa and Scatter mapping policies in MVAPICH2

As it can be seen, the *linear* policy works best for the recursive-doubling algorithm in Figure 1(b). However, this trend is completely reversed in Figure 1(a) where *linear* shows close to worst performance while *numa* policy dominates others by showing significantly better performance. This clearly depicts that the performance of high-level algorithms and applications is not guaranteed to perform well on newly emerging multi-/many-core architectures. The reason is that the conventional wisdom of designing such algorithms and applications by decoupling algorithmic details—by relying on the abstract notion of *ranks*—from underlying architecture is not sustainable and will not lead to highly efficient algorithms on future multi-/many-core HPC systems.

Next, we evaluate the performance of osu_allreduce by using a *two-phase* MPI_Allreduce algorithm; a hierarchical reduction phase followed by a hierarchical broadcast phase. We consider two different views of the topology in the algorithm: A "socket-level" view and a "numa-level" view. The "socket-level" algorithm orchestrates the hierarchical reduction operation by performing it for all processes within each socket in parallel followed by a reduction between designated *leader* processes on each socket. On the other hand, the "numa-level" algorithm performs a reduction for all processes sharing a NUMA domain, followed by a reduction amongst leader processes across NUMA domains. The broadcast phase for both algorithms is essentially a mirror of the hierarchical reduction operations.

Figure 2 shows the performance comparison of the two schemes for a job size of 48 processes ordered in a way that there are 12 processes per socket (6 per NUMA domain). We observe that the "socket-level" algorithm performs better than the "numa-level" algorithm, showing a non-trivial dependence between the implementation of the algorithm, the number of processes involved and the underlying architecture. However, the performance trends may certainly change for future emerging architectures, which might have different characteristics for NUMA domains/sockets or add additional levels to the hierarchy. This would generally require re-designing existing algorithms to adapt to new changes, which is undesirable and complex. We propose abstracting the topology as an internal communication tree that represents a virtual hierarchy of the architecture, and designing a *base-case* algorithm that can be applied on each level of the tree. By decoupling the

topology and the implementation of the algorithm, efficient communication operations can be implemented for emerging multi-/many-core architectures with minimal changes in the runtimes/applications by simply building an architecture-aware virtual tree and executing the *base-case* algorithm.

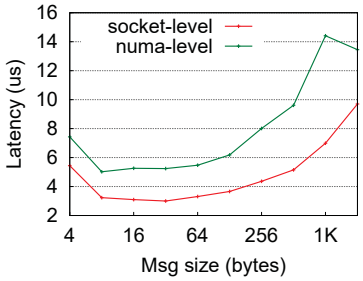



**Figure 2:** Performance of a direct reduce followed by broadcast implementation of MPI_Allreduce for 48 processes on an Intel Xeon Gold 6348H (Cooper Lake) CPU.

## III. Contributions

This paper revisits intra-node communication on emerging multi-/many-core HPC systems and proposes a framework called OSMP for optimized shared memory based communication on such systems.

The core principle of OSMP is to treat hardware as a *"first-class"* citizen. OSMP creates generalized, hierarchical, and architecture-aware communication trees—a fundamental building-block to abstract underlying hardware topology—for efficient communication. These communication trees group together hardware resources based on *locality domains* including cores, last-level-caches (LLCs), NUMA nodes, and sockets. *A fundamental contribution of OSMP is that it makes developing complex high-level communication algorithms a simple and straightforward task. This can be accomplished by simply traversing the communication tree and executing the base algorithm—specified by the user—on each level of the tree.*

OSMP is generic and can support any communication pattern, however, in order to demonstrate the efficacy of our proposed solution, we focus on collective communication patterns, specifically Barrier, Reduce, Broadcast, and Allreduce. We choose MPI as a candidate runtime and show how it can exploit OSMP for collective communication. Other runtimes have not been included in the paper for brevity. Our proposed design is able to reduce the intra-node communication latency of various MPI collectives by up to **7.8**× and improve the performance of different applications such as **miniAMR** by up to **13%** and **AMG** by up to **15%**.

As a summary, this paper makes the following key contributions:

1) Identify and highlight that the conventional wisdom of designing high-level communication operations—by abstracting processes as a basic functional unit—leads to performance degradation on the newly emerging multi-/many-core processors HPC systems.
2) Design and implement OSMP with a fundamental construct called *communication tree* that abstracts the underlying architecture and topology of the system. This structure has support for *virtual* topologies on systems with a flat memory hierarchy and/or high core-count.
3) Orchestrate architecture-aware high-level collective communication operations, including existing ones, using OSMP by simply traversing the communication tree and specifying *base* algorithms.
4) Demonstrate the efficacy of OSMP by integrating it within an MPI library.
5) Conduct performance evaluation of OSMP against state-of-the-art MPI libraries on four emerging architectures—**Intel Cascade Lake**, **AMD Rome**, **ARM A64fx** and **IBM POWER9** using OSU Micro-Benchmarks (OMB) and several applications.

## IV. Overview of The Proposed Framework

In this section, we provide a high-level overview of various design elements of our proposed framework, OSMP. We aspire for the following design goals: 1) modularity, 2) architecture-awareness, and 3) high-level abstractions. A high-level layered view of the OSMP framework is depicted in Figure 3. Rest of this section provide details for design components of OSMP.

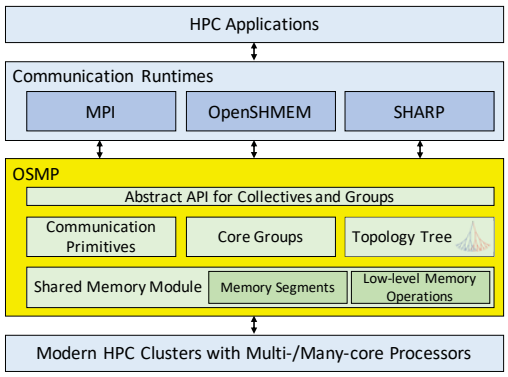



**Figure 3:** Architectural Overview of the OSMP Framework.

### A. Programming Model

Remote Memory Access (RMA) is a popular programming model for designing communication primitives on modern SMP systems as it offers direct load/store semantics on remote PE's memory. On the other hand, *send/recv* based model employed by communication libraries such as MPI makes it easier to reason the performance trends due to explicit communication, however, the underlying implementations also use RMA like memory mapping based designs e.g., shared memory. OSMP uses posix shared memory as the backend for inter-process communication and uses a RMA like model and supports *put/get* and atomic operations on remote memories. While the OSMP designs are flexible enough to support MPI-like explicit communication semantics as well, we only focus on RMA semantics to keep the discussion focused.

### B. Topology Tree

The topology module is primarily responsible for abstracting the machine topology within the node and building a tree composed of cores that share local memory in each level of the intra-node hierarchy. A hierarchical tree is built during the initialization stage of OSMP and later used throughout the execution of the program. Different communication algorithms can be executed in a hierarchical manner by simply traversing the tree in a top-down or bottom-up fashion. The traversal order and direction depends on the nature of the communication algorithm. For example, the broadcast operation requires top-down traversal while the reduce operation requires bottom-up traversal.

### C. Shared Memory Module

The memory module is responsible for all operations pertaining to memory such as allocating memory regions, low level memory ordering operations, memory copies and others. The memory operations are carried out using the RMA model, as discussed earlier. This module interfaces with the topology module by creating these shared-memory regions at each level in the hierarchy. Architecture specific read/write barriers (e.g., memory fences) and other memory operations are used to ensure correctness on weak-consistency architectures such as ARM and POWER. All the data-structures as well as memory operations used within the OSMP library are cache-aligned.

### D. Communicating Core Groups

OSMP groups the set of active cores within the same memory domains and encapsulate them into abstract representation called *communicating core groups*. This is analogous to *communicators* in MPI. Each level in the tree is represented as a group of cores with shared memory segments. Each group designates a *leader* core for co-ordination/communication above and below the current level of the tree.

### E. Communication Primitives

Basic communication primitives are the implementation of a given communication algorithm e.g., collective operations such as reduce, allreduce, barrier, and broadcast, that are executed on a group of core at a given level of the tree. This essentially decouples the communication operation and its corresponding architecture-aware implementation relieving the application developers from dealing with the diversity of many-core architectures. For instance, a basic implementation of a gather operation where one process (the root) reads from shared memory in a loop and others write to shared memory can be made topology-aware by executing the same algorithm at each level of a given topology tree. This shifts the focus of algorithm developers to just having a "base-case" implementation after which our framework can tune itself to deduce the right algorithm to use at each level based on the hardware topology.

## V. Design and Implementation

This section describes specific implementation details of OSMP. It is modular by design and can be dynamically linked with other communication runtimes such as MPI.

### A. Abstracting Topology Tree

OSMP tries to abstract out the topology of the machine in the `topology` object, which is dynamically generated at runtime and built using the hardware locality (hwloc) [4] library. While hwloc has its own framework to get the topology objects, OSMP only considers metadata useful for orchestrating communication operations. Our proposed library's focus is more towards building architecture-aware tree-based communication operations, with hwloc playing a role only in querying hardware information. The `topology` object is queried by one process, and broadcast using XML to other process for efficiency reasons. After querying the topology from hwloc, we start building an internal representation of the tree using a pre-defined `hierarchy` object which contains information about various memory domains in the topology. Each memory domain object contains a bitmap of the cores that belong to that memory domain, the group of processes that participate in operations at that memory domain, and other metadata. Finally, we assign memory domain objects for the abstract core object pertaining to the process from the bottom level to the top level. For instance, on a system with a hierarchy levels containing the machine, socket and NUMA nodes, the abstract core object's "leaf" memory domain will point to the abstract NUMA object it belongs to and the "root" will point to the abstract global memory domain for the machine.

### B. Creating shared-memory segments

OSMP relies on a multi-process paradigm for executing communication operations. We create shared-memory regions to facilitate group communication operations. OSMP generates a shared-memory region using the `mmap` system call for each process group formed described in sectionV-C. The file name is kept unique for each process group by having a composite key containing the index of the memory domain the process belongs to (representing a horizontal relationship), the level in which the memory domain exists in the hierarchy (representing a vertical relationship) and job specific information. The per-memory domain shared-memory segment is logically divided for each communication operation. As a first version, we implement four widely used collective operations namely barrier, broadcast, allreduce and reduce. Each collective operation has its own abstract object, which contains flags for synchronization of processes as well as a pointer to the data segments where applicable. All memory allocations are cache-aligned with synchronization flags given adequate padding to ensure they are cache-line sized to avoid false sharing. A first-touch is then performed on all allocated memory segments to avoid page faults during the execution of collective algorithms. Each process group contains a pointer to the shared-memory segment to facilitate inter-process communication within the group.

### C. Grouping cores in the topology

The topology of the system is obtained at a global level. OSMP maintains an active set of cores and filters out the topology by removing the set of cores that are not currently being used in a specific OSMP context. After filtering the topology and grouping memory domains together, there must be an explicit grouping of processes within each level in the hierarchy so that the group of processes can perform communication operations. After generating the topology described in section V-A, we get the abstract `core` object for a given process and start from the "leaf" memory domain for that core and go all the way up to the root. At the "leaf" level, every process calls a grouping function with its core ID, level in the hierarchy and the index of the memory domain the process resides in at that level. In subsequent levels in the hierarchy that are higher than the leaf, only a designated *leader* from each memory domain participates in group creation. This is done for convenient implementation of collective algorithms that can be composed of smaller version of themselves. However, the grouping method is flexible and can be modified to form an arbitrary group of processes within the node.

### D. Designing basic communication primitives

OSMP provides basic implementations of collective communications—as exemplars—for barrier, broadcast, reduce. These primitives can be considered as "base-cases", intended to be executed by process groups in conjunction with the topology tree maintained by OSMP. Each process gets its abstract `core` object, traverses its list of memory domains, and executes the algorithm with process groups defined at each memory domain. The traversal can be either be bottom-up—for the likes of reduce—or top-down—for the likes of broadcast. Appropriate read and write barriers are placed to ensure correctness on processors with out-of-order execution support.

Figure 4 shows how a gather operation can be orchestrated on the topology built by OSMP on a dual socket AMD EPYC processor, with 2 sockets, 4 NUMA nodes per socket and 4 LLCs per NUMA. Each step is denoted as a yellow circle, with the iteration number inside it. In step 1, all processes sharing a CCX memory domain perform a gather to a designated root process in their respective CCX domain. In step 2, the designated leaders perform a gather operation using shared-memory regions created at the NUMA level. Step 3 involves a gather operation on all leader processes within each socket and finally, in step 4, one leader from each socket performs a gather to the root. Thus, a simple direct implementation is easily applied to the topology shown, thereby bringing in architecture awareness without any changes to the algorithms' base-case.

### E. Orchestrating a topology-aware barrier using OSMP

Consider a basic implementation of a barrier algorithm involving 2 phases, *arrival* and *notify*. In the arrival phase, a root process waits for all other processes to set a flag. The notify phase involves the root setting a flag and other processes waiting on that flag. This is shown in Algorithms 1 and 2, respectively. The algorithm is written in a generic manner by considering a set of monotonically increasing logical IDs called ranks. The intention here is for the algorithm developer to only consider implementing a base-case algorithm, and offloading the job of making it architecture aware to the OSMP runtime. The "ranks" passed on to the base-case implementation are the logical ID of a core in the group created at a certain level in the hierarchy.

**Algorithm 1:** Naïve base-case algorithm implementation for barrier arrival

```
Input  : G — Communication group
Output: All processes signal their arrival to logical rank 0
Function: barrier_arrival(G)
begin
    logical_rank ← get_logical_rank_in_group(G)
    if logical_rank = 0 then
        for j ← 1 to get_group_size(G) do
            wait_for_arrival(j);
        end
    end
    else
        mark_process_arrived(logical_rank);
    end
end
```

**Algorithm 2:** Naïve base-case algorithm implementation for barrier notify

```
Input  : G — Communication group
Output: Logical rank 0 notifies all other processes
Function: barrier_notify(G)
begin
    logical_rank ← get_logical_rank_in_group(G)
    if logical_rank = 0 then
        for j ← 1 to get_group_size(G) do
            mark_process_notified(j);
        end
    end
    else
        wait_for_notification(logical_rank);
    end
end
```

Algorithm 3 shows how the arrival and notify functions are made architecture aware by traversing the automatically generated topology tree exposed by OSMP . It only takes around 10-20 lines of code to orchestrate base-case algorithms in an efficient architecture topology-aware manner.

## VI. Virtual Hierarchies

OSMP supports building virtual hierarchies in cases where a 1:1 tree of the architecture's topology is relatively inefficient. One example where this can happen is in processors with shallow memory hierarchies like the Intel Xeon (Cascade Lake) processors. This feature is useful in order to eliminate bottlenecks in parallelism due to having too many processes on one memory domain. This section provides details on how

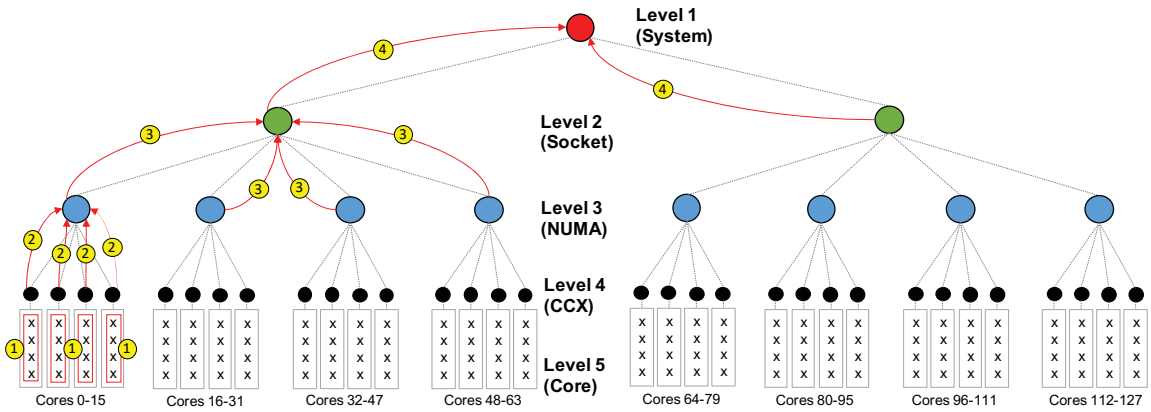



**Figure 4:** Sample Topology tree built by the OSMP framework for dual-socket AMD EPYC 7742 (Rome) 64-core processors.

**Algorithm 3:** Using OSMP to make the base-case implementation architecture topology aware

```
Input  : T — Abstracted topology of the machine
Input  : global_rank — Logical rank of a process
Output: An Inter-process Barrier is executed
begin
    core ← get_core_object(T, global_rank)
    domain ← last_level_memory_domain(core)
    while domain ≠ ∅ do
        comm_group ← get_comm_group(domain)
        if comm_group_valid(comm_group) then
            barrier_arrival(comm_group)
        end
        domain ← parent(domain)
    end
    domain ← top_level_memory_domain(core)
    while domain ≠ ∅ do
        comm_group ← get_comm_group(domain)
        if comm_group_valid(comm_group) then
            barrier_notify(comm_group)
        end
        domain ← child(domain)
    end
end
```

OSMP optimizes hierarchical execution of base communication primitives.

### A. Topology Trees with Virtual Hierarchies

The strategy to represent memory domains in a hierarchical data-structure such as a tree, and later use it for orchestrating communication primitives is an effective one. However, this leads to performance degradation on HPC systems built with shallow memory hierarchies such as TACC Frontera that is equipped with dual-core Intel Cascade Lake processors with 28 cores per socket. In the context of a topology tree without virtual hierarchies, a reduce implementation would require one root process to read through all vectors written to shared memory by all other non-root (27) processes leading to contention and generating more work for the root. There are opportunities to exploit more parallelism—that OSMP makes use of—through virtual hierarchies. This kind of support can be provided in traditional messaging libraries, but would require significant effort. OSMP is able to support virtual hierarchies to make a system with shallow hierarchy look "deeper" than it actually is. Using this approach, instead of running a reduction on 28 processes on a socket, we can further split the socket into 2 virtual hierarchical domains (with 14 cores each) and execute the algorithm hierarchically. Here, the factor 2 by which a domain is split is called degree and can be configured at runtime.

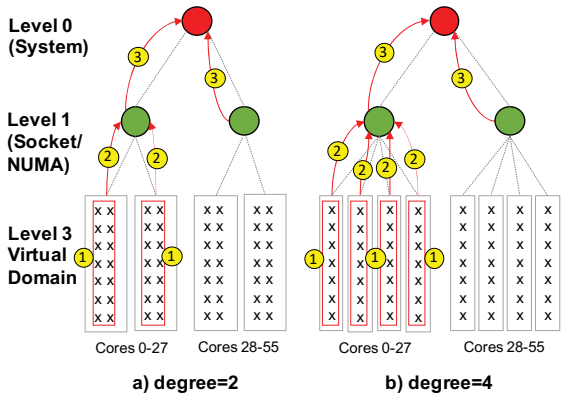



**Figure 5:** Sample Topology trees built by the OSMP framework for dual-socket Intel Xeon Platinum 8280 (Cascade Lake) 28-core processors. a) Topology tree with a virtual hierarchy/domain at level 3 (degree=2), and b) Topology tree with a virtual hierarchy/domain at level 3 (degree=4).

An example of a topology tree built by OSMP for a dual-socket Intel Xeon Platinum 8280 (Cascade Lake) 28-core processor is shown in Figure 5. The topology tree consists of 3 levels including system, sockets/NUMA domains, and *virtual* hierarchies. This Xeon processor has a shallow memory hierarchy, which means that all 28 cores on the socket are on the same NUMA domain and also share LLC. The example in Figure 5 splits the socket memory domain into two configurations: a) 2 virtual domains (degree 2), and b) 4

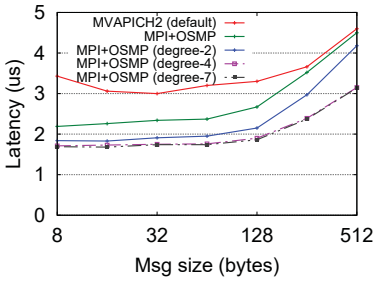



**Figure 6:** Average latency of MPI_Allreduce comparing OSMP used within a production MPI library with and without virtual domains and the default implementation in MVAPICH2 on a dual-socket Intel Xeon Platinum 8280 (Cascade Lake) 28-core processor. We observe that splitting the last memory domain (sockets) by 7 (MPI+OSMP with degree-7) shows the best results.

virtual domains (degree 4). A gather operation is orchestrated in the same way as explained in section V-D.

### *B. Performance Benefits of Virtual Topologies*

Figure 6 shows the performance gains obtained for the MPI_Allreduce operation by creating virtual domains on a dual-socket Intel Xeon Platinum 8280 (Cascade Lake) processor. OSMP-default is slightly better than the default implementation in MVAPICH2 (which uses the same algorithm) due to architecture specific optimizations, and the latency gets better as we increase the number of virtual domains due to the inherent parallelization of compute operations as well as exploitation of locality between cores that are close to each other.

## VII. Experimental Evaluation

In this section, we choose MPI as the candidate runtime and provide an in-depth performance evaluation when comparing our proposed designs with state-of-the-art designs employed by production MPI libraries.

We evaluate our designs on four different state-of-the-art architectures. The AMD Rome system consists of dual-socket AMD EPYC 7742 64-core CPUs, the TACC Frontera system consists of dual-socket Intel Xeon Platinum 8280 Cascade Lake CPUs, the Lassen system consists of dual-socket 44-core IBM POWER9 CPUs and the Ookami system consists of four-socket Fujitsu A64fx ARM-based CPUs. The detailed configuration of these systems is shown in Table I.

On each system, we compare the performance of our design against a tuned version of the MVAPICH2-2.3.4 MPI library and a state-of-the-art vendor-specific MPI library (HPC-X v2.7.0 on the AMD Rome system, Intel MPI 2019 Update 7 on TACC Frontera, IBM Spectrum MPI 10.3.1.03rtm0 on Lassen and Open MPI 4.0.3rc4 on Ookami). For comparisons at the benchmark level, we use the OSU Micro-Benchmark (OMB) suite [8], with the latency reported being an average of 1000 iterations over 5 runs. For all our experiments, we use the default set of parameters for each MPI library. For application level evaluations, we use miniAMR [9], AMG [10] and OpenFOAM V7[11].

**Table I:** Hardware specification of different tested clusters

| Specification | AMD Rome | TACC Frontera | Lassen | Ookami |
|---|---|---|---|---|
| Processor Family | AMD EPYC | Intel Cascade Lake | IBM POWER | Fujitsu |
| Processor Model | EPYC 7742 | Xeon Platinum 8280 | POWER9 | A64fx |
| Clock Speed | 3.4 GHz | 2.7 GHz | 3.8 GHz | 2.2GHz |
| Sockets | 2 | 2 | 2 | 4 |
| Cores Per socket | 64 | 28 | 22 | 12 |
| NUMA nodes | 8 | 2 | 6 | 4 |
| CCX Per NUMA | 4 | N/A | N/A | N/A |
| RAM (DDR4) | 512 GB | 192 GB | 256 GB | 256 GB |
| Interconnect | IB-HDR(200G) | IB-HDR(100G) | IB-EDR (100G) | IB-HDR(200G) |

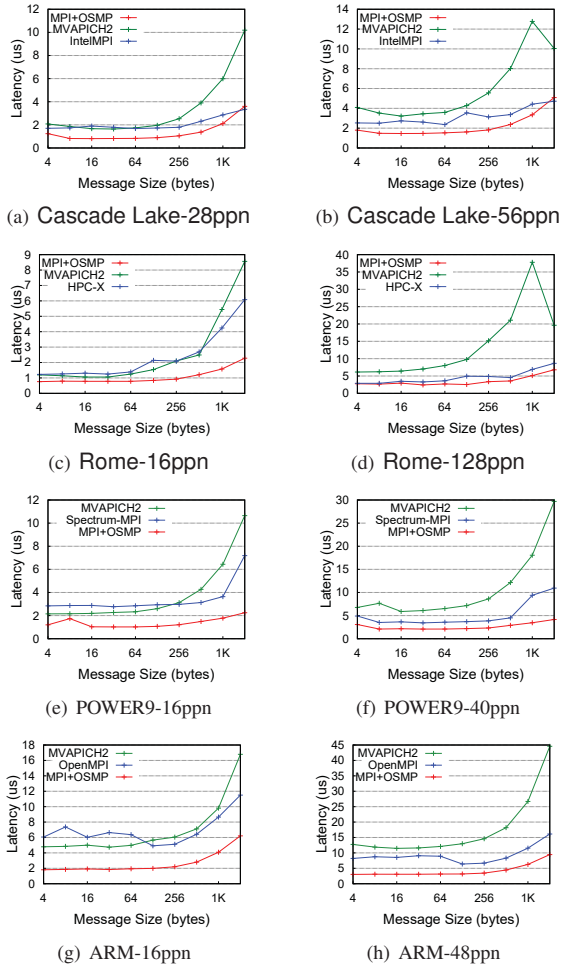
### *A. Micro-Benchmark Evaluation*

In this section, we present results using the OSU Micro-Benchmarks (OMB) suite [8] for select MPI collective operations. We compare the performance of state-of-the-art MPI libraries and a version of a production MPI library integrated with a tuned version of OSMP (referred to as MPI+OSMP in the result graphs). We report the average latency in microseconds for MPI_Allreduce, MPI_Barrier and MPI_Bcast operations. For MPI_Reduce, we report the maximum latency amongst all processes since the average latency might not be the representative of the performance of this collective. We present results for two cases : One in which all processes use cores on the same memory domain, and the other in which processes use up all cores in the node.

*1) **Reduce**:* The performance of MPI_Reduce is shown in Figure 7. On AMD Rome, we observe up to 7× improvements for MPI+OSMP over MVAPICH2 default and 20% improvements over HPC-X. The benefits of MPI+OSMP increase as the message size increases. This is due to the fact that architecture-level optimizations play a larger role as the message size increases. More specifically, locality and parallelization of compute operations are vital to performance when performing reduction on larger vectors. On the Intel Cascade Lake (TACC Frontera) system, we observe improvements of up to 3× over the base-line of MVAPICH2 and up to 30% improvements over Intel-MPI. On the POWER9 system, we observe up to 7.1× improvement over MVAPICH2 and up to 3.2× over Spectrum-MPI. Similar trends are observed on the ARM system, with up improvements of up to 4.7× over MVAPICH2 and up to 3.9× over OpenMPI.

*2) **Bcast**:* Figure 8 shows the performance of MPI_Bcast. On TACC Frontera, we observe up to 2× improvements over both Intel MPI as well as the MVAPICH2 baseline. On AMD Rome, we observe a speed up of up to 6X when compared to HPC-X and up to 2X over MVAPICH2 default. On the POWER9 system, we observe up to 7.8× improvement over Spectrum-MPI and up to 4× over MVAPICH2. On the ARM system, we observe improvements of up to 2.6× over MVAPICH2 and up to 3.3× over OpenMPI. The results demonstrate the effectiveness of using virtual domains, especially for operations like Bcast which involve multiple processes accessing a buffer at the same time.

*3) **Allreduce**:* The performance of MPI_Allreduce is shown in Figure 9. Our Allreduce implementation is essentially a reduce followed by a broadcast, so the trends are largely the same as what we observe with MPI_Reduce. On AMD Rome, we observe up to 3× improvements for MPI+OSMP



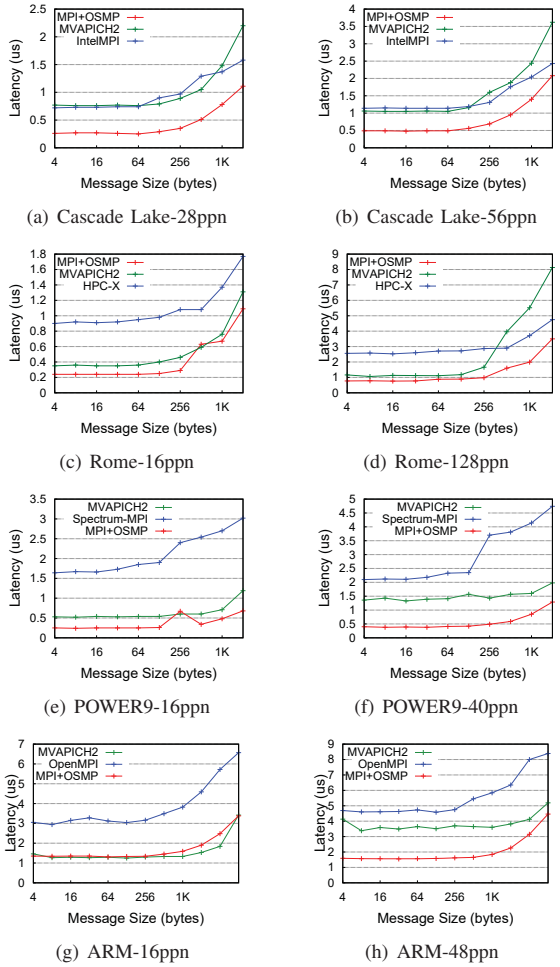

(a) Cascade Lake-28ppn (b) Cascade Lake-56ppn

(c) Rome-16ppn (d) Rome-128ppn

(e) POWER9-16ppn (f) POWER9-40ppn

(g) ARM-16ppn (h) ARM-48ppn

**Figure 7:** Performance of MPI+OSMP and state-of-the-art libraries for MPI_Reduce on the Intel Cascade Lake, AMD Rome, IBM POWER9 and ARM A64fx architectures up to a message size of 2K bytes for different process counts.




(a) Cascade Lake-28ppn (b) Cascade Lake-56ppn

(c) Rome-16ppn (d) Rome-128ppn

(e) POWER9-16ppn (f) POWER9-40ppn

(g) ARM-16ppn (h) ARM-48ppn

**Figure 8:** Performance of MPI+OSMP and state-of-the-art libraries for MPI_Bcast on the Intel Cascade Lake, AMD Rome, IBM POWER9 and ARM A64fx architectures for different process counts.

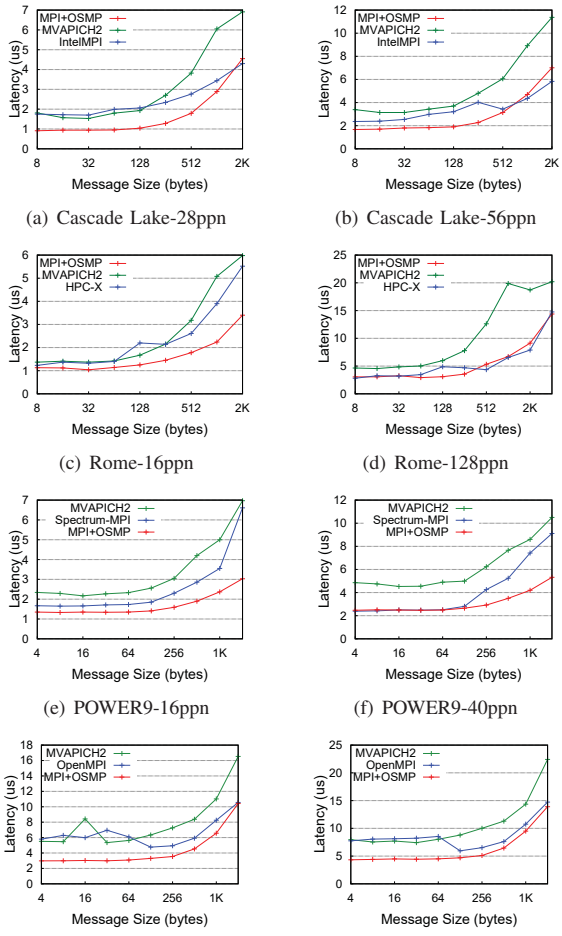
over MVAPICH2 default and up to 1.6× improvements over HPC-X. On the Intel Cascade Lake (TACC Frontera) system, we observe improvements of up to 2× over MVAPICH2 and up to 25% over Intel MPI. On the POWER9 system, we observe up to 2.18× improvement over Spectrum-MPI and up to 2.3× over MVAPICH2. On the ARM system, we observe improvements of up to 2× over MVAPICH2 and up to 2.3× over OpenMPI.

*4)* ***Barrier****:* Figure 10 shows the performance of the MPI_Barrier operation on the Intel Cascade Lake and AMD Rome architectures. On the Intel Cascade Lake system, MPI+OSMP outperforms default MVAPICH2, which uses the same algorithm, by up to 20%. MPI+OSMP performs similar to Intel MPI for up to 28 processes. However, we observe a degradation in performance when compared to Intel MPI at full subscription (56 processes per node). We attribute this to the baseline algorithm being sub-optimal in case of OSMP. On the AMD Rome system, we observe up to 4× improvement over default MVAPICH2. MPI+OSMP shows up to 10% better performance when compared to HPC-X with 64 processes.

### *B. Application Evaluation*

In this section, we present results for three applications : MiniAMR, AMG and OpenFOAM on different architectures. Timings for applications that use reduction collectives are generally dominated by compute and other point-to-point operations that induce skews within MPI, which might not be representative of the actual communication performance benefits that OSMP provides for these collective operations.

*1)* ***MiniAMR****:* MiniAMR is a proxy application which explores the performance of finite difference or volume codes that use Adaptive Mesh Refinement (AMR) [9]. It applies a stencil calculation on a unit cube computational domain, which is divided into blocks. We evaluate the performance with miniAMR on Cascade Lake (in Figure 11(a)) and Rome (in Figure 11(b)) architectures for varying number of processes. We compare our OSMP design with MVAPICH2 and Intel



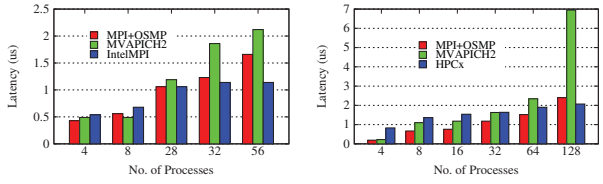

(a) Cascade Lake-28ppn (b) Cascade Lake-56ppn

(c) Rome-16ppn (d) Rome-128ppn

(e) POWER9-16ppn (f) POWER9-40ppn

(g) ARM-16ppn (h) ARM-48ppn

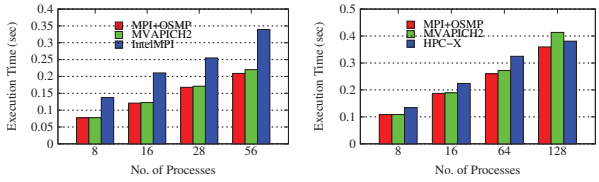
**Figure 9:** Performance of MPI+OSMP and state-of-the-art libraries for MPI_Allreduce on the Intel Cascade Lake, AMD Rome, IBM POWER9 and ARM A64fx architectures up to a message size of 2K bytes for different process counts.




(a) Cascade Lake (b) Rome

**Figure 10:** Performance of MPI+OSMP and state-of-the-art libraries for MPI_Barrier on Intel Cascade Lake and AMD Rome architectures for different process counts.

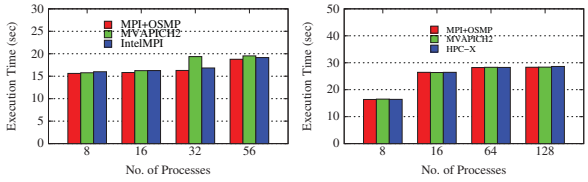
MPI or HPC-X on each system, respectively. As seen in Figure 11(a), we observe 5% improvements over MVAPICH2 and about 40% improvements over Intel MPI on the TACC Frontera system. Similarly, Figure 11(b) demonstrates that OSMP based designs show up to 13% improvements over MVAPICH2 and up to 6% improvements over HPC-X on the AMD EPYC Rome architecture. The general trend follows what we observed in benchmark-level evaluation—as the usage of processes/cores increases—our proposed designs tend to show better performance.




(a) Frontera (b) Rome

**Figure 11:** Performance comparison on Rome and Frontera systems for miniAMR.

*2) **AMG**:* AMG is an algebraic multigrid solver for linear systems arising from unstructured grids domain [12]. We evaluate AMG using a weak-scaling problem set at varying process counts on a single node of Frontera and Rome, and compare the total execution time of our proposed design against default MVAPICH2 (without OSMP) and Intel MPI / HPC-X. Figure 12(a) shows performance comparison of OSMP, MVAPICH2 and Intel MPI on the TACC Frontera (Cascade Lake CPU) system. We observe that OSMP shows up to 15% improvement over MVAPICH2 and up to 5% improvement over Intel MPI.




(a) Frontera (b) Rome

**Figure 12:** Performance comparison on Rome and Frontera systems for AMG.

*3) **OpenFOAM**:* OpenFOAM is a computational fluid dynamics (CFD) application. The profiling of MPI version of the code revealed MPI_Allreduce as one of the collective communication primitive being used by the application. We evaluated the performance of OpenFOAM v7 on TACC Frontera and compared the performance of OSMP and default MVAPICH2. As seen in Figure 13, the proposed designs achieve up to 5% improvements over MVAPICH2 for 32 and 56 processes.

## VIII. Related Work

In this paper, we build on our existing work [13], [14], [15] to optimize shared memory intra-node communication

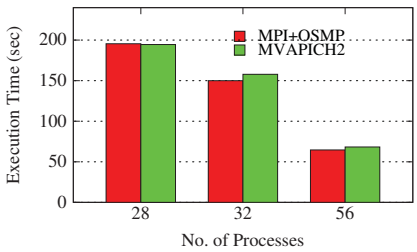



**Figure 13:** Performance of OpenFOAM v7 on TACC Frontera

on emerging multi-/many-core systems. Goglin et al. [16] proposed an interface to exploit underlying hardware topology by building matching hierarchical communicators. This approach requires modifications to the MPI standard. Chatzopoulos et al. [17] addressed the conflicting requirements of portability and efficiency on multi-core processors by abstracting the topology of multi-cores using determinism of cache-coherence protocols and latency measurements between cores. Benefits of this technique were demonstrated on five processors using a variety of applications including MapReduce library and mergesort algorithm. Niethammer et al. [18] proposed designs to enable hardware topology-aware cartesian mapping of processes. They visualized cartesian grids into three levels of hierarchy and try to minimize communication across the most expensive links. Kaeste et al. [19] proposed a library called *Smelt* that provides machine-aware tree abstractions for multi-core processors in order to enable efficient broadcast communication. Similar to Chatzopoulos et al. [17], they extract topology information through a set of micro-benchmarks and conducting analysis of results.Another unique aspect of this work is that they attempt to not only enable efficient communication but also reduce lag between communicating processes. Ma et al. [5] optimized communication in the MPI library by extracting hardware information and building tuning tuning rules for different CPU architectures, types, cache size, and locality. While significant research has been done in the broader area, none of them offer an architecture-aware hierarchical framework that delivers better performance and highly convenient abstractions for applications as well as a range of communication runtimes including MPI.

## IX. Conclusion and Future Work

Future HPC systems are likely to be equipped with multi-/many-core processor architectures with higher core counts and deeper/complex memory hierarchies. These systems present challenges to applications and communication runtime libraries alike. The conventional wisdom of designing communication primitives using processes as the fundamental building block leads to performance penalties. This paper tackles this issue by designing and implementing OSMP, which is a library that abstracts the topology of a system and builds communication trees that form the basis for architecture-aware core-to-core communication. We orchestrate communication primitives—picking common collective communication operations as exemplars—in an architecture-aware manner using the topology tree and existing base-case algorithms. We further demonstrate the performance benefits of building virtual topologies on systems with shallow memory hierarchies. Our proposed designs show up to **7.8**× improvement at the micro-benchmark level, and **15%** improvement for application over state-of-the-art MPI libraries. In the future, we plan to explore the impact of topology trees on kernel-assisted collective functions and inter-node communication operations.

## References

[1] Aurora Supercomputer, http://aurora.alcf.anl.gov.
[2] Frontier Supercomputer, https://www.amd.com/en/products/frontier.
[3] L. Dagum and R. Enon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.
[4] 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 *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 180–186.
[5] 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.
[6] Ma, Teng and Bosilca, George and Bouteiller, Aurelien and Goglin, Brice and Squyres, Jeffrey M and Dongarra, Jack J, "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.
[7] The MPI Forum Working Group, http://mpi-forum.org/.
[8] OSU Micro-benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/.
[9] Sasidharan, Aparna and Snir, Marc, "MiniAMR: A Miniapp for Adaptive Mesh Refinement," Tech. Rep., 2016.
[10] Lawrence Livermore National Security, "Algebraic multigrid benchmark," https://github.com/LLNL/AMG/tree/master.
[11] The OpenFOAM Foundation, https://openfoam.org/.
[12] AMG ECP Proxy Application, https://computing.llnl.gov/projects/co-design/amg2013.
[13] J. M. Hashmi, S. Xu, B. Ramesh, M. Bayatpour, H. Subramoni, and D. K. D. Panda, "Machine-agnostic and Communication-aware Designs for MPI on Emerging Architectures," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 32–41.
[14] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "FALCON: Efficient Designs for Zero-Copy MPI Datatype Processing on Emerging Architectures," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 355–364.
[15] 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)*. IEEE, 2018, pp. 1020–1029.
[16] B. Goglin, E. Jeannot, F. Mansouri, and G. Mercier, "Hardware topology management in mpi applications through hierarchical communicators," *Parallel Computing*, vol. 76, pp. 70 – 90, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118301480
[17] G. Chatzopoulos, R. Guerraoui, T. Harris, and V. Trigonakis, "Abstracting Multi-Core Topologies with MCTOP," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 544–559. [Online]. Available: https://doi.org/10.1145/3064176.3064194
[18] C. Niethammer and R. Rabenseifner, "An MPI Interface for Application and Hardware Aware Cartesian Topology Optimization," in *Proceedings of the 26th European MPI Users' Group Meeting*, 2019, pp. 1–8.
[19] S. Kaestle, R. Achermann, R. Haecki, M. Hoffmann, S. Ramos, and T. Roscoe, "Machine-aware Atomic Broadcast Trees for Multicores," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 33–48.