ADAPT: An Event-Based Adaptive Collective Communication Framework

Xi Luo
University of Tennessee
Knoxville, Tennessee
xluo12@vols.utk.edu

Wei Wu
Los Alamos National Laboratory
Los Alamos, New Mexico
wwu@lanl.gov

George Bosilca
University of Tennessee
Knoxville, Tennessee
bosilca@icl.utk.edu

Thananon Patinyasakdikul
University of Tennessee
Knoxville, Tennessee
tpatinya@vols.utk.edu

Linnan Wang
Brown University
Providence, Rhode Island
linnan_wang@brown.edu

Jack Dongarra
University of Tennessee
Knoxville, Tennessee
Oak Ridge National Laboratory
Oak Ridge, Tennessee
University of Manchester
Manchester, UK
dongarra@icl.utk.edu

ABSTRACT

The increase in scale and heterogeneity of high-performance computing (HPC) systems predispose the performance of Message Passing Interface (MPI) collective communications to be susceptible to noise, and to adapt to a complex mix of hardware capabilities. The designs of state of the art MPI collectives heavily rely on synchronizations; these designs magnify noise across the participating processes, resulting in significant performance slowdown. Therefore, such design philosophy must be reconsidered to efficiently and robustly run on the large-scale heterogeneous platforms. In this paper, we present ADAPT, a new collective communication framework in Open MPI, using event-driven techniques to morph collective algorithms to heterogeneous environments. The core concept of ADAPT is to relax synchronizations, while maintaining the minimal data dependencies of MPI collectives. To fully exploit the different bandwidths of data movement lanes in heterogeneous systems, we extend the ADAPT collective framework with a topology-aware communication tree. This removes the boundaries of different hardware topologies while maximizing the speed of data movements. We evaluate our framework with two popular collective operations: broadcast and reduce on both CPU and GPU clusters. Our results demonstrate drastic performance improvements and a strong resistance against noise compared to other state of the art MPI libraries. In particular, we demonstrate at least 1.3× and 1.5× speedup for CPU data and 2× and 10× speedup for GPU data using ADAPT event-based broadcast and reduce operations.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; • **Theory of computation** → *Concurrent algorithms*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*;

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208054

KEYWORDS

MPI, event-driven, system noise, collectives operations, GPU, heterogeneous system

1 INTRODUCTION

The need to satisfy the scientific computing community's increasing computational demands leads to larger HPC systems with more elaborate architectures, many levels of memory, highly multi-threaded hardware components, and complex high-speed network topologies. Many of these scientific applications rely on collective data movement patterns—namely, collective communications. In distributed-memory systems, most of these parallel applications take advantage of the Message Passing Interface (MPI) [13] paradigm to satisfy their data transfer needs. Therefore, it is crucial for MPI libraries to sustain the parallel applications by providing the most optimal communication capabilities, including highly-optimized collective routines. With the increasing scale and complexity of HPC systems, performance scalability on such machines becomes more challenging. Overall, there are two challenges preventing the performance scalability of collective operations in large HPC systems:

Propagation of noise. HPC systems tend to increase in size, with thousands of computer nodes and millions of, potentially different, cores. In such large and heterogeneous systems, system noise can be easily amplified with all types of synchronizations including the implicit synchronizations within collective operations and drastically hurts an application's performance. Operating system noise was studied in [2] [17] and [11]. With the increase in system size, noise should be extended to include the delay caused by fault tolerance [12, 22], in situ-analytics [24], and power management [14]. As suggested in previous research [17], noise can dramatically slow down large-scale parallel applications. For some applications, noise of only 2.5% can drop the performance more than 400× on 500 nodes and 1800× on 2500 nodes [10]. The main reason for the slowdown is MPI collective operations. Noise occurring locally has little impact by itself, but by delaying local communications it is being propagated to other processes, and becomes magnified through certain synchronizations within collective operations. Usually, a collective

operation consists of many fine-grained MPI one-sided or two-sided communication routines. Carelessly handling dependencies of these point to point (P2P) routines brings extra synchronizations, which potentially leads to noise propagation and amplification, and therefore delivers sub-optimal performance.

Hardware heterogeneity and hierarchy. Another important factor as we approach exascale is resource heterogeneity, resulting in increasingly complex hardware hierarchies. A compute node on such heterogeneous system usually contains multiple CPU sockets, connected by high-speed inter-socket connections (e.g., Intel QPI or AMD Hyper-transport). Scaling up, several compute nodes are coupled together through the high-performance network interface and organized into racks, then finally into super-computers.

Benefiting from massive parallelism with low power consumption, HPC systems are increasingly incorporating accelerators (NVIDIA GPUs, Intel KNL or specialized FPGA). Hence, more and more applications, including traditional scientific applications [37] [41] and deep learning applications [38], are adopting accelerators to boost their performance. However, embracing accelerators increases the already complicated architecture hierarchy, as accelerators are connected to the host via PCI Express bus, and in some cases such as GPUs connected to other GPUs via GPU-GPU interconnects (NVLink). Clearly all these advancements at the hardware level cause a drastic increase in performance and capability differences between levels of the hierarchy. Communication time between processes greatly varies depending on the physical distance and types of networks between them. Thus, maintaining good network performance requires the holistic integration of process placement and architecture capabilities. Recent advances in MPI collectives implementations have demonstrated that such performance issues can be partially solved by integrating hardware topology information into collective operations [15, 19, 23]. However, the insufficient cooperation of communications of different topology levels (i.e., intra-socket, inter-socket, PCI bridges and inter-node levels) leads to sub-optimal overlapping of communications at different levels. Also, the algorithms are not adaptable to fluctuating network conditions. This calls for a collaborative approach, between multiple levels of collective algorithms, dedicated to holistically managing all levels of the network hierarchies.

Contributions. This work's contributions are: an asynchronous event-driven framework able to expose and take advantage of parallelism between independent data movements composing collective communications, a novel multi-level architecture aware implementation of some collective communications algorithms (reduce and broadcast), an event-driven composition mechanism allowing multiple collective algorithms to be composed in a data-dependent hierarchical manner, and the efficient integration of this event-driven framework in the communication engine of Open MPI. More precisely, facing noise and complex hardware hierarchies, we propose "ADAPT," a new event-driven collective framework in Open MPI, which treats the completion of non-blocking P2P routines within collective operations as events, and each event completion allowing the high-level logic to unfold dependent P2P routines. With the help of the event-driven design, we can relax synchronization dependencies and only maintain the minimal data dependencies. By relaxing synchronizations, our framework offers more potential to absorb the system noise instead of propagating or even amplifying it further. Combining this feature with a carefully built topology-aware tree, the ADAPT framework provides greater opportunities to concurrently communicate over networks of different hierarchies on heterogeneous systems. Specifically, the advantages of ADAPT framework include:

- Relax the synchronizations in collective operations to alleviate the effects of noise;
- Express a collective communication as a topology-aware communication tree, to maximize the concurrent communications over different hardware levels;
- Enable highly efficient topology-aware NVIDIA GPU collective operations.

The rest of this paper is organized as follows. Section 2 analyzes the noise propagation with two kinds of dependencies in existing implementations of MPI collectives, describes the design of the ADAPT collective operations framework, and explains its noise-resistant capability; Section 3 describes how the proposed framework builds a topology-aware tree from different levels in the hardware hierarchy to support CPU topology-aware collective operations; Section 4 presents two optimizations for the ADAPT framework to better support GPU topology-aware collective operations; and Section 5 is dedicated to performance evaluation of the ADAPT collectives for different systems, including CPU and GPU. This work concludes with related works described in Section 6, followed by a summary and future directions outlined in Section 7.

2 DESIGN

With the increasing scale of high-performance computers, there are more and more sources of interference that can impact the performance of applications. Even though local noise often causes very little delay per process, such delays can affect the overall performance of applications significantly when noise is propagated to other processes through communications [17].

Compared with P2P communications, collective communications are easily affected by noise for two reasons. First, the noise propagation raises with the number of participants in collective communications. Since the number of processes is determined by application developers, reducing the noise cannot be done by limiting the number of processes. Second, there are many sequential dependencies in the implementations of collective operations that allow noise to propagate. In this section, we identify the dependencies in the implementations of collectives operations in mainstream MPI libraries and analyze how these dependencies propagate noise. We then introduce the ADAPT collective operations framework, which adopts an event-driven idea to relax dependencies.

2.1 Existing Implementations

In general, collective operations implemented in major MPI libraries are based on P2P communications, either blocking or non-blocking. Carelessly managing these P2P communications introduces unnecessary sequential dependencies between them and such dependencies bring synchronizations and order between otherwise independent P2P communications. As discussed in [17], a delayed process postpones its P2P communications, and further delays other processes. Therefore, noise on one process delays the P2P communications, and then delayed P2P propagates noise to other P2P

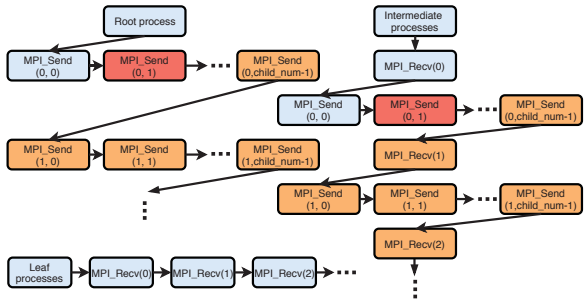


Figure 1: Implementation of MPI_Bcast using blocking P2P communications. Red: noise source; Orange: affected P2P. child_num: number of childen of the process

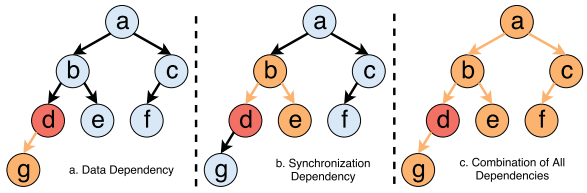


Figure 2: Noise propagation of dependencies. Red: noise source; Orange: affected processes/P2P routines

communications via dependencies between them. Later, all delayed P2P communications stall the processes participating in these P2P communications. In this fashion, noise is propagated from a single process to others and slows down the performance of entire collectives.

We analyze two implementations of *MPI_Bcast*, one using blocking and one using non-blocking communications, to identify hidden dependencies and highlight their noise propagation patterns.

2.1.1 Collectives Using Blocking Point-to-Point Communications. Figure 1 presents a pipelined implementation of *MPI_Bcast* using blocking P2P routines which can support any kind of tree-based algorithms. In the figure, $MPI\_Send(x, y)$ means sending segment x to child y and $MPI\_Recv(x)$ means receiving segment x from parent. With pipelining, big messages are segmented into several pieces and propagated in order. In this implementation, root process issues an *MPI_Send* for each of its children to transfer a segment. After they are finished, the same procedure applies on the following segments. Intermediate processes post an *MPI_Recv* to receive a segment from their parent, and then issue multiple *MPI_Sends* to send the received segment to their children. After these *MPI_Sends* are done, they start to receive the next segment until all segments are processed. Leaf processes work similarly to intermediate processes without delivering received segments.

Since blocking P2P communication routines involve synchronizations like handshakes between sender and receiver, noise on any of these two processes can slow down the blocking P2P routines, which further delays the process on the other side. When there are dependencies between P2P routines, noise can be propagated from one to others, resulting in slowdown of the entire collectives. In the blocking P2P implementations of *MPI_Bcast*, we identify two kinds of dependencies, which can propagate noise significantly:

- **Data Dependency**. If input data of some P2P routines depends on the output data of another P2P routine, then there is data dependency between them. Thus, they have to be executed in order to get the correct results. In the blocking P2P implementation of *MPI_Bcast*, intermediate processes have to receive one segment before sending it to their children. As in Figure 1, $MPI\_Recv(i)$ must occur before $MPI\_Send(i, m)$ ($m \in [0, child\_num - 1]$) for any segment i. This dependency is necessary for the correctness of the broadcast operation. With data dependency, noise on intermediate processes can be propagated to all their children. Figure 2.a represents the noise propagation pattern caused by data dependency with a binomial tree broadcast. If noise on process d delays the *MPI_Recv* from b to d, then following *MPI_Send* from d to g is delayed, leading to the delay of g.
- **Synchronization Dependency**. This kind of dependency is caused by synchronizations between P2P routines. A blocking P2P routine waits until the operation is done, which naturally brings a hidden synchronization. Such synchronization leads to a dependency between the blocking P2P routine and all future routines, which is called *Synchronization Dependency*. This dependency brings unnecessary ordering of the routines, and can propagate noise to other processes. As in Figure 1, root and intermediate processes always send any segment to child m before child n, for all $m < n$ (i.e., $MPI\_Send(0, 0)$ always before $MPI\_Send(0, 1)$), even though there is no data dependency between them. Thus, if $MPI\_Send(0, 0)$ is delayed (marked red), all the following *MPI_Sends* and *MPI_Recvs* are affected (marked orange). Figure 2.b presents how noise is propagated from one process to another as a result of this dependency. If noise on process d delays the *MPI_Recv* of segment *0* from b to d ($MPI\_Recv(0)$ on process d), then $MPI\_Send(0, d)$ on process b is also delayed since noise can be propagated through blocking P2P communications, and thus the parent of process d—process b—is delayed. Later, because of synchronization dependency, the delay of $MPI\_Send(0, d)$ on process b affects the $MPI\_Send(0, e)$ on process b, resulting in the delay of process e, the sibling of process d. Therefore, a delayed process can affect its siblings and parent in this case.

Based on the analysis above, via data dependency, noise on a process is propagated to its children and further to grandchildren, which is not avoidable. However, unnecessary synchronization dependency propagates noise to parent and siblings. With this noise propagation pattern, noise can be propagated to grandchildren, grandparents and descendants of grandparents, and consequently, all processes could be affected by noise (Figure 2.c). Therefore, we can conclude that the blocking P2P implementation of *MPI_Bcast* is able to amplify noise.

2.1.2 Collectives Using Nonblocking Point-to-Point Communications. An improvement over the previous implementation is using non-blocking P2P communications (*MPI_Isend*, *MPI_Irecv*) instead of blocking ones. Figure 3 presents the pipelined implementation of *MPI_Bcast* in Open MPI using non-blocking P2P routines. $MPI\_Isend(x, y)$ means sending segment x to child y, $MPI\_Irecv(x)$ means receiving segment x from parent, and $Wait(x)$ means wait for segment x. In this implementation, root process issues an *MPI_Isend* for each of its children to transfer a segment and uses *Waitall* to

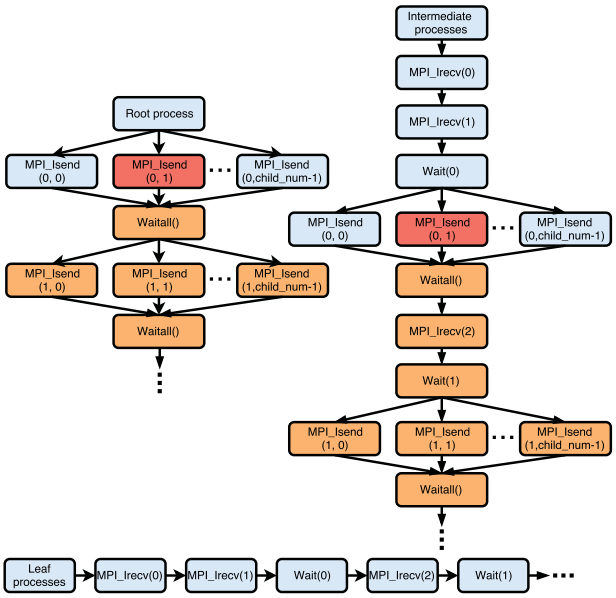


Figure 3: Implementation of MPI_Bcast using non-blocking P2P communications. Red: noise source; Orange: affected P2P

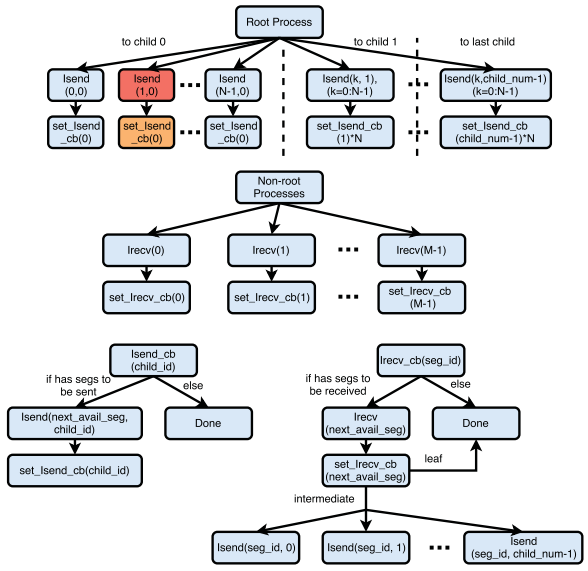


Figure 4: Implementation of MPI_Bcast in ADAPT. Red: noise source; Orange: affected operations

wait for all the previous *MPI_Isends*. After they are finished, it starts to send the next segment. A leaf process posts two *MPI_Irecvs* for the first two segments, but only waits for the first one segment. When it receives the first segment, it posts an *MPI_Irecv* for the next segment and waits for the second segment. Intermediate processes behave similarly to leaf processes except that they need to send the received segment to its children in the same fashion as the root process. The reason the non-root processes post two *MPI_Irecvs* instead of one is to handle out of order segments.

Unlike blocking P2P communication routines, non-blocking routines are more noise resistant; if one process is delayed, the process on the other side is still able to progress other non-blocking P2P routines without hanging. Thus, in most cases, noise on one process is less likely to be propagated to another via non-blocking P2P routines [17], except when the other process has nothing to work but waits for the delayed non-blocking P2P communication. Even though non-blocking P2P communication has a higher potential to absorb noise, in the non-blocking P2P implementation of *MPI_Bcast*, there are still dependencies that can propagate noise. The following describes the two dependencies and noise propagation patterns resulting from them:

- **Data Dependency**. It is the same as the blocking P2P implementation. This dependency is required for correctness of broadcast operation. Therefore, like the blocking P2P implementation, noise on intermediate processes can be propagated to their all children with data dependency.
- **Synchronization Dependency**. As seen in Figure 3, by using *MPI_Isends*, data movements from one process to all of its children become independent and they can be progressed in any sequence by MPI's progress engine. However, the *Waitall* and *Wait* act as synchronizations that order P2P routines between them. Thus, any delays on these two routines can affect the following routines. This type of dependency can also propagate noise to siblings and parent. For example, in Figure 2, if noise on process d delays the *MPI_Irecv* of segment *0* from b to d ($MPI\_recv(0)$ on process d), then process b is not delayed directly since b can still progress other *MPI_Isend*, such as $MPI\_Isend(0, e)$ on process b. However, because of the *Waitall*, process b can be affected if all other non-blocking P2P communications are completed, except for the delayed one. In this case, *Waitall* on process b only waits for the delayed P2P, and hence, process b is delayed. Later, because of synchronization dependency, the delay of *Waitall* on process b affects the following $MPI\_Isend(1, e)$ on process b, resulting the delay of $Wait(1)$ on process e. Thus, process e, the sibling of process d, is also delayed.

With the combination of data dependency and synchronization dependency, noise can be propagated to all the processes. Compared with the blocking P2P implementation discussed before, the non-blocking implementation of *MPI_Bcast* is more tolerant to noise since non-blocking routines offer out-of-order executions, instead of waiting for delayed P2P routines. However, the *Waitall* and *Wait* in the nonblocking version still bring heavy synchronizations, and thus the nonblocking version is not sufficient to absorb noise and minimize noise propagation.

2.2 ADAPT: Event-Driven Design

2.2.1 Implementation of ADAPT. In this section, to better exploit the available parallelisms in collectives and minimize noise propagation, we present the ADAPT collective communication framework. The key of ADAPT is to design collective communications algorithms with events and callbacks, which eliminates the need to wait for P2P communications to complete. This type of programming model is called "event-driven." In a typical event-driven program, there is an event loop to detect events, and when an event occurs, the corresponding callback is triggered. In this way, the execution flow of a program is determined by events and their callbacks. To implement events and callbacks, ADAPT is deeply

integrated with the communication engine in Open MPI: we use the completion of a non-blocking P2P communication as an event, which triggers a detailed analysis of the state of the collective algorithms, and we enable next data movements if necessary by posting new non-blocking routines. One thing worth mentioning is that the non-blocking P2P communications, where callbacks are attached, are at a lower level than *MPI_Isend*/*MPI_Irecv* (shown as "*Isend*/*Irecv*" in the following) since *MPI_Isend*/*MPI_Irecv* does not support callbacks. Instead of waiting for each non-blocking P2P, we create a request for each collective operation and do not mark it as complete until the collective is done. Therefore, Open MPI's progress engine keeps progressing all the non-blocking P2P routines until this request is completed.

The implementation of the ADAPT broadcast algorithm is shown in Figure 4. Following the event-driven pattern, all segments are propagated to all processes via a series of the *Isends*/*Irecvs* and their callbacks.

- Root: the root process posts N *Isends* to send the first N segments to each child, then uses *set_Isend_cb* to attach a callback to each of these *Isends*. When any *Isend* is completed, the *Isend_cb* will be called to post another *Isend* to send the next available segment.
- Non-root: a non-root process posts M *Irecvs* to receive the first M segments from its parents and attaches callbacks to these *Irecvs* with *set_Irecv_cb*. When any *Irecv* is completed, *Irecv_cb* is called to post another *Irecv* for receiving the next available segment. If the process is an intermediate process, besides receiving the next available segment, it posts multiple *Isends* to send the received segment to its children in *Irecv_cb*.

In ADAPT, we issue N *Isends* to a single child and M *Irecvs* from the parent to handle multiple segments simultaneously to maximize the usage of the network resources and absorb noise (will be discussed in 2.2.2). Usually, M is set to be larger than N. This is because there is an issue of matching an *Isend* (of a segment) to a corresponding *Irecv*: if the segment arrives on the receiver side before the receiver posts a corresponding *Irecv*, the segment will be considered "unexpected." In this case, MPI needs to store it into a temporary buffer and match it later when the corresponding *Irecv* is posted by the receiver. This introduces significant latency, as the procedure requires memory allocation and data copying; thus, it is very important to ensure an *Irecv* is always posted before the arrival of its corresponding segment. To address this issue, we need to make sure M is bigger than N to minimize the chance of unexpected segments.

2.2.2 Analysis of Dependencies in ADAPT. As discussed in Section 2.1, in existing broadcast implementations, there are two types of dependencies (data dependency and synchronization dependency). In this section, we demonstrate how the ADAPT framework relaxes synchronization dependencies and minimizes noise propagation by making every segment and every child independent of each other using an event-driven design.

Data Dependency. Any process needs to receive the data before starting to send the data to its children. This dependency is necessary for the correctness of the broadcast operation.

Synchronization Dependency. In ADAPT, the completion of a non-blocking P2P routine triggers a callback, then posts another non-blocking routine. For example, on root process, $Isend(N, 0)$ can only be issued after the earliest one of $Isend(i, 0)$ $(i \in [0, N-1])$ is completed. This leads to a synchronization dependency between these two P2P routines. However, this synchronization dependency can hardly propagate noise because of the following two reasons. First, in ADAPT, segments can be handled in any order (segment independence). Take the root process for example. All segments are put into a virtual "segment pool" in the beginning. The root process then posts N *Isends* to send the first N segments to child 0 ($Isend(i, 0)$ $(i \in [0, N-1])$). If any of them are done, its *Isend_cb* issues another *Isend* to send the next available segment from the segment pool. Thus, there are N concurrent *Isends* between root and each child. If any one is delayed, segments can be re-balanced to other *Isends*. Therefore, delay of one segment can hardly delay the communication of other segments between root and one child—and thus, noise can be absorbed. Second, each child can transfer data segments independently from each other (child independence). In the existing implementations of *MPI_Bcast*, a process always waits for a segment to be transferred to all its children before transferring the next segment. While in ADAPT, every child keeps its own state and transfer data segments independently. In this way, noise cannot be propagated to a process's siblings. In a word, in ADAPT, segment independence absorbs the noise magnitude and child independence limits the range of noise propagation, and thus the synchronization dependency in ADAPT can hardly propagate noise.

Based on the above analysis, we can conclude that, benefits from the event-driven design, ADAPT relaxes synchronization dependencies, and minimizes noise propagation.

2.2.3 Extend ADAPT to other collective operations. From the three implementations of tree-based *MPI_Bcast* mentioned above, we can notice that there is a common communication pattern: a process sends data to its children or receives data from its parents, which we call **basic building block**. In the blocking P2P version,

Algorithm 1: Blocking P2P Implementation

```
1 for i ← 0 to k do
2     MPI_Send(i)/MPI_Recv(i);
```

the basic building block can be shown as Algorithm 1, where k is the number of needed P2P communications. A similar pattern appears in the implementation of collective operations in MPICH and MVAPICH. In the non-blocking P2P version, the basic building

Algorithm 2: Nonblocking P2P Implementation

```
1 for i ← 0 to k do
2     MPI_Isend(i)/MPI_Irecv(i);
3 Waitall()
```

block becomes Algorithm 2; this pattern exists in MVAPICH and Open MPI. Compared to the blocking version, it provides more parallelism by adopting *MPI_Isend*/*MPI_Irecv*. The basic building block of ADAPT is Algorithm 3, which uses non-blocking P2P as Algorithm 2 and reduces synchronizations by removing the *Waitall*. For the *MPI_Bcast* algorithms are not based on trees, the

Algorithm 3: Adapt Implementation

```
for i ← 0 to k do
    Isend(i)/Irecv(i);
    set_Isend_cb(i)/set_Irecv_cb(i);
```

event-driven design can still be used as long as the basic building blocks exist in the algorithms. For example, a scatter followed by an allgather is a common algorithm to do big message *MPI_Bcast*. In the scatter phase, a process may send data to multiple other processes which is similar to the *MPI_Bcast* discussed above and the same technique can be applied to it. For other one-to-all, all-to-one and some all-to-all collectives, a process always need to send or receive data from other processes. In this way, the basic building block is a common part in various collectives, and thus the event-driven design can be extended to them.

2.2.4 Support Different Collectives with Multiple Communication Trees. In collective operations, the relationship of parents-children forms a communication tree. In ADAPT, the communication tree of a collective operation can be any type of tree, e.g., a chain, binary tree, binomial tree, or other advanced trees [31]. The design of the ADAPT framework allows most operations to be independent of the underlying communication tree. Therefore, it is easy to adapt the trees based on network topology to boost performance, which is discussed in Section 3.

3 TOPOLOGY-AWARE SUPPORT IN ADAPT

This section describes how to equip the ADAPT framework with topology-aware capabilities in order to handle complex hardware hierarchies in heterogeneous systems.

3.1 Topology-Aware Collectives with Multi-Level Communicators

Existing methods of implementing hierarchical collective operations use multiple communicators to group processes based on topology [1, 19, 33]. A communicator is created for each group. Take a broadcast operation, for example: a broadcast starts from the top level communicator and the next level cannot start until the upper-level broadcast is finished. Thus, a broadcast consists of several broadcasts within multiple levels' communicators. In this way, this method is suboptimal for large messages, since the upper-level broadcast and lower-level broadcast do not overlap with each other.

3.2 Topology-Aware Collectives with a Single Communicator

To eliminate the boundaries between multiple levels, we present a new topology-aware collective operation based on a topology-aware tree, which "virtually" groups all processes in a single communicator instead of dividing the processes into multiple communicators.

3.2.1 Build Topology-Aware Communication Tree. As discussed in Section 2, the ADAPT framework can be plugged in with a

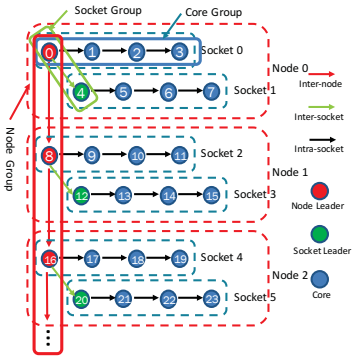


Figure 5: Topology-aware communication tree of a broadcast algorithm on a multi-core cluster (4 cores per socket and 2 sockets per node)

carefully built communication tree to provide topology-aware capabilities. This section describes how to build such a topology-aware tree. The following description assumes the topology information and that the placement of each process is known. In Open MPI, this information is available to all processes and can be extracted from the PMIx runtime using Portable Hardware Locality (hwloc) [4].

Using the topology information for all processes, we can build a topology-aware communication tree. We start by grouping processes bottom-up. Figure 5 presents an example of the topology-aware tree. There are three groups: node, socket, and core. Since the network within a group is homogeneous, the way processes communicate with each other within each group can be decided, optimally, by existing approaches [29]. Also, because the network of each group is independent from others, processes within different groups can communicate using a different pattern. As in Figure 5, P4, P5, P6 and P7 form a group and communicate using a chain fashion. P0 and P4 form a chain in the upper-level group and P4 glues these two chains together.

3.2.2 ADAPT vs. Blocking/Non-Blocking P2P Collectives. The blocking/non-blocking P2P implementations of one-to-all and all-to-one collectives (discussed in Section 2.1) can also support different kinds of tree algorithms; it is possible to plug the topology-aware tree discussed above into blocking/non-blocking implementations of collectives to make them topology-aware. It is known that the nonblocking performs better than the blocking version because it can relax the dependencies and exploit more parallelism by posting several *MPI_Isends*, and these *MPI_Isends* may transfer data concurrently if they occupy different physical networks. For example, in the broadcast operation of the tree in Figure 5, P0 posts three *MPI_Isends* to P1, P4 and P8—which occupy inter-node, inter-socket and intra-socket communication channels—and can be progressed independently at different speeds. However, the *Waitall* in the non-blocking version forces the three *MPI_Isends* to complete at the same time, and thus, they all run at the slowest speed of the three inter-connections, which is the inter-node communication. Therefore, the non-blocking P2P implementations of collectives are not able to fully utilize the network's resources.

The ADAPT framework removes the *Waitall*, so in the previous example, the three *Isends* can be progressed independently and

data can be propagated at full network speed. Therefore, the event-driven design of ADAPT utilizes network resources more efficiently.

4 SUPPORT FOR HETEROGENEOUS ARCHITECTURES

Most GPU-aware MPI implementations assume each MPI process is bound to one GPU, transforming inter-process communications to/from GPU memory to inter-GPU communications. With its release of CUDA version 4.0, NVIDIA introduced CUDA Inter-Process Communication (IPC) to allow GPU remote direct memory access (RDMA) between two GPUs within the same socket. For communications between GPUs across the socket, there are two approaches: going through CPU memory, or using GPUDirect (starting from CUDA 5.0) by going through a third-party device—in general an InfiniBand (IB) adapter—attached to each socket. Most machines are not equipped with multiple InfiniBand adapters, thus, using InfiniBand for inter-socket communications delays inter-node communications because they will be sharing the same InfiniBand adapter and the same PCI-Express bus. In this paper, we assume inter-socket communications go through CPU memory; for inter-node communication, it can use either GPUDirect or go through intermediate CPU memory; but both approaches occupy Network Interface Controllers (NICs), which in our case is InfiniBand. Therefore, collective operations on GPU data need to handle multiple types of networks, including PCI-Express, InfiniBand, and CPU memory bus. To handle communications over such contrasting networks, topology-aware collectives are more efficient than traditional collective algorithms. Therefore, in this section, we describe how we extended and optimized ADAPT to efficiently handle GPU data.

4.1 Minimize Communications over PCI-Express

As seen in Figure 5, for broadcast operations, the node leader is the busiest MPI process of the entire communications because it receives data from the previous node leader and sends data to the next node leader, the next socket leader, and the next process within the same socket. All of these communications go through PCI-Express, and might introduces heavy congestions on the node's PCI-Express. Figure 6.a shows the movements of the data on the node leader when using GPUDirect. Communications between node leaders go through NICs via PCI-Express (red line). When the node leader sends the data to a socket leader, data goes through an implicit intermediate CPU buffer to the next socket leader's GPU. Such data movement uses the PCI-Express and Intel QuickPath Interconnect (QPI) bus (purple line). When the node leader sends data to the next GPU within the same socket, data flows through the PCI-Express as well (blue line). Since these three communications occupies the same direction on PCI-Express, only one third of the PCI-Express's bandwidth can be available for each communication. As shown in Figure 6.b, when GPUDirect is disabled, inter- and intra-socket communications do not change, but inter-node communications need to take an extra step to go through an intermediary CPU buffer. Since the CPU buffer is implicitly managed by Open MPI, each P2P communication would use different CPU buffers, even if it is transmitting the same data. This consumes a large amount of CPU memory and PCI-Express bandwidth. To tackle the congestion of PCI-Express in previous implementations, we allocate an explicit CPU buffer for the node leader process to cache GPU data. Non-root node leaders cache the received data into this CPU buffer so that it can send the data to the next node and socket leader directly from this cached CPU buffer, without pulling data from GPU memory via PCI-Express again. Later, cached data is flushed into the corresponding GPU memory via an asynchronous CUDA copy. Root process also caches data in CPU memory to relax the workload on the PCI-Express. Figure 6.c shows the optimized data flow of the node leader process. As long as the NICs and GPUs are not connected to the same PCI-Express switch, communications between (1) NIC and explicit CPU buffer, (2) CPU buffer to GPU, and (3) GPU to neighbor GPU will use different PCI-Express lanes. Thus, they can be simultaneously progressed. With an explicit CPU buffer, we are able to map intra-socket, inter-socket and inter-node communications to use different physical networks, and achieve communication overlapping.

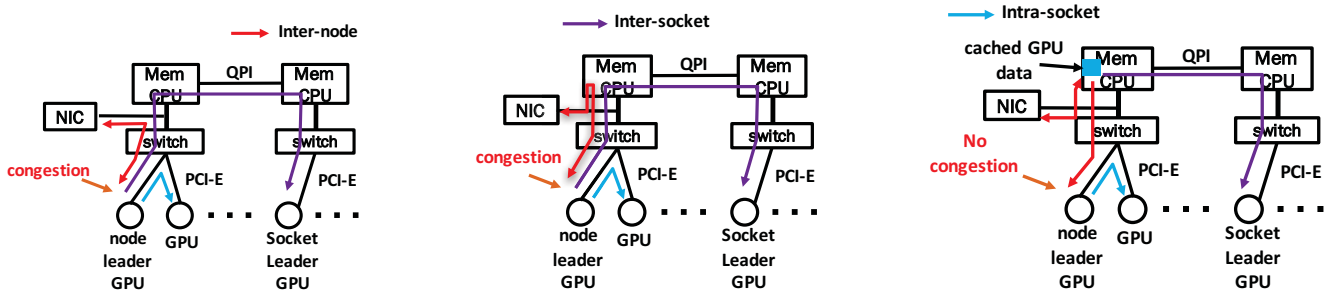
4.2 Offload Reduction Operation on GPU

Reduction operations are mathematical operations on vectors, and are embarrassingly parallel; thus, they are good candidates for GPU execution, as each CUDA thread handles reduction operation on few elements of the vector. However, as discussed in Section 6, the current GPU-aware MPIs still use CPUs to perform the reduction, which is not efficient because most MPI applications are still single-threaded, CPU-bound reduction operations occupy scarce CPU resources, and can potentially delay any other communications or computations the application might have. With this in mind, we offload the reduction operations[1] to the GPU asynchronously by using multiple CUDA streams. This allows us to seamlessly overlap communications and reduction operations.

5 EXPERIMENTAL EVALUATION

In order to assess the capability of the ADAPT framework, we first evaluate its two features: noise absorption and topology-aware design; second, we present an end-to-end evaluation with other state of the art MPI libraries; last, we study the performance of collective operations with an application. The first two parts are done with Intel MPI Benchmark (IMB) using two collective operations: broadcast and reduce. The experiment is conducted on three clusters: (1) **Cori**, a CPU cluster, on which each node is equipped with 2 Intel Xeon E5-2689 v3 CPUs, and nodes are connected by Cray Aries; (2) **Stampede2**, a CPU cluster, on which each node is equipped with 2 Intel Xeon 8160 CPUs, and nodes are connected by Intel Omni Path; and (3) **NVIDIA PSG K40** cluster, a GPU cluster with 10 nodes, where each node is equipped with 4 K40 GPUs with CUDA 7.5 and 2 deca-core Intel Xeon E5-2690v2 Ivy Bridge CPUs, and nodes are connected by 40Gb/s FDR IB. The MPI libraries compared are: Cray MPI, Intel MPI, Open MPI 2.0 default (shown as "OMPI-default"), MVAPICH2 (not support Cray Aries), and Open MPI 2.0 ADAPT (shown as "OMPI-adapt").

[1] We developed CUDA kernels for all pre-defined MPI reduction operations, but they are outside the scope of this paper.



(a) With MPI GPUDirect (b) Without MPI GPUDirect (c) Optimization with explicit CPU buffer

Figure 6: Dataflow of non-root node leader MPI process

5.1 Component Evaluations

5.1.1 Noise Impact. The section studies the performance impact of noise on ADAPT and other MPI libraries to demonstrate ADAPT's noise absorption abilities. The noise experiment is conducted in the CPU cluster, since it has more nodes and is presumably more impacted by the noise.

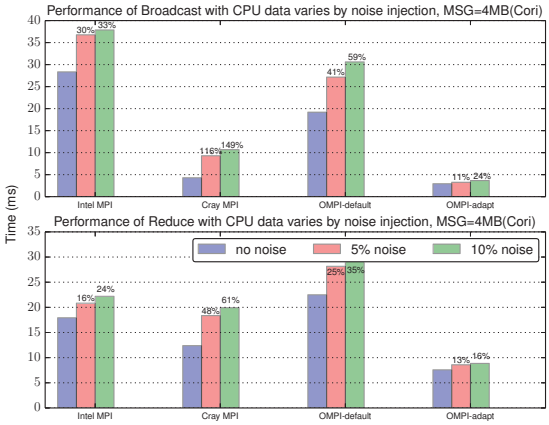
Figures 7a) and 7b) present the noise impact on the performance of broadcast and reduce operations in different MPI implementations on the Cori and Stampede2 using 1024 and 1536 processes respectively. In this experiment, message size is set to 4 MB to allow enough segments to fill the pipeline and highlight the noise effects on performance. As suggested in [10, 17], performance interference usually has greater impact on larger systems. In order to show delays of collective operations caused by noise with fewer processes, we use a method similar to the one in [2] to randomly inject 0–10ms (average 5%) and 0–20ms (average 10%) noise following a uniform distribution with a fixed frequency of 10 Hz, since low-frequency, long-duration noise has the greatest impact on performance [10]. In Figure 7a) and 7b), the bars show the average time to do a collective operation and the numbers above red and green bars show the performance slow down percentages after 5% and 10% noise injection respectively. We note that after 10% noise injection, the slowdown of MVAPICH is 868%, which is outside the scope of the figure. MVAPICH's reduce encounters segmentation fault with IMB when message size is 4MB, so there is no results for the reduce operation. As seen in the Figure 7a) and 7b), benefiting from the event-driven design, OMPI-adapt relaxes synchronization dependencies and thus is largely unaffected by the noise compared to other MPI libraries. Therefore, OMPI-adapt only slows down up to 24% and 16% on broadcast and reduce even with 10% noise injected on both machines. The broadcast and reduce of OMPI-default uses non-blocking P2P routines, so as analyzed in Section 2.1.2, it contains synchronization dependency, which can propagate noise. Thus, it slows down up to 59% and 99% on broadcast and reduce. Since Cray and Intel MPI are not open-source, we do not know their detailed implementations, but Cray MPI slows down up to 149% and 61%; the numbers for Intel MPI are 33% and 24%, which are less noise-resistant than ADAPT.

5.1.2 Topology-Aware. This section evaluates the performance of topology-aware broadcast and reduce operations in ADAPT against state of the art topology-aware implementations. Figures 8b) and 8a) present the performance of ADAPT (shown as OMPI-adapt) with CPU data on Cori and Stampede2, compared with all the topology-aware algorithms in Intel MPI. We also integrate the topology-aware communication tree to the default collective module of Open MPI (shown as "OMPI-default-topo") to demonstrate that ADAPT is better at network resource utilization. We notice that Intel MPI performs much better on Stampede2 than Cori, and we think this may be because the underlying network of Stampede2 is Intel Omni Path whereas Intel MPI has its own optimizations for its hardware. Even so, on both machines, the topology-aware broadcast of ADAPT performs the best over others for big messages. For messages smaller than 1MB, ADAPT's broadcast performs a little slower than Intel MPI on Stampede2; this is because the pipeline topology-aware algorithms in ADAPT require enough segments to fulfill the pipeline. For reduce operations on large messages, ADAPT performs better than most topology-aware algorithms in Intel MPI, except Shumulin's. There may be two reasons: first, the reduction operations in ADAPT do not have any vectorization optimizations. Second, performance of collectives relies heavily on the underlying P2P communications; the Shumulin's algorithm may be optimized for P2P over Intel Omni Path. This assumption is made by comparing the performance of Shumulin's on Stampede2 (with Omni Path) and Cori (without Omni Path). Compared with OMPI-default-topo, ADAPT is able to support independent communications over different networks; therefore, even though it has the same topology-aware communication tree with OMPI-default-topo, ADAPT still performs 20% better. Overall, ADAPT eliminates boundaries between different hierarchies and supports independent communications, and therefore it fully utilizes network resources and delivers better performance than most topology-aware MPI implementations especially for big messages.

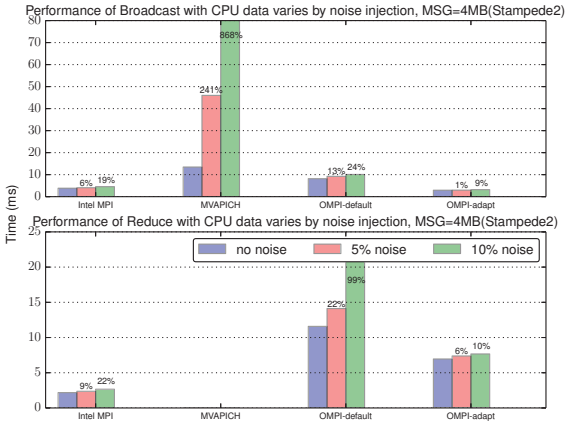
5.2 End-to-End Evaluations

To study the overall impact of event-driven and topology-aware design in ADAPT, we conduct two types of experiments with both CPU and GPU data: first, the total number of processes is fixed and we measure the performance for different message sizes; second, we look at the strong scalability, which measures the performance by varying the number of processes with a fixed message size.

5.2.1 Collective Communications with CPU Data. Figure 9a) and Figures 9b) present the communication time of broadcast and reduce of ADAPT compared with other state-of-the-art MPI implementations with different message sizes on the Cori and Stampede2

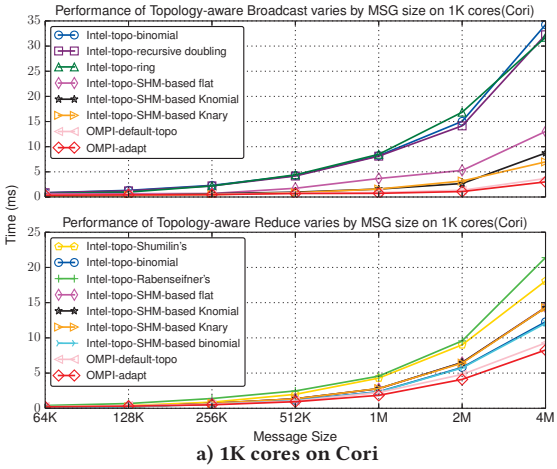


a) 1K cores on Cori

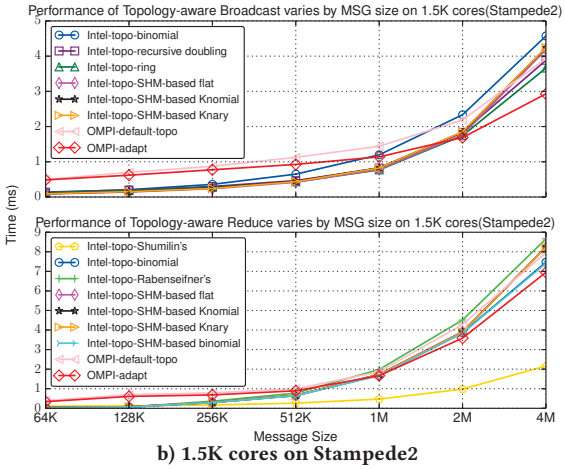


b) 1.5K cores on Stampede2

Figure 7: Performance of broadcast and reduce on CPU data with noise injection. Numbers on bars represent performance slowdown caused by noise

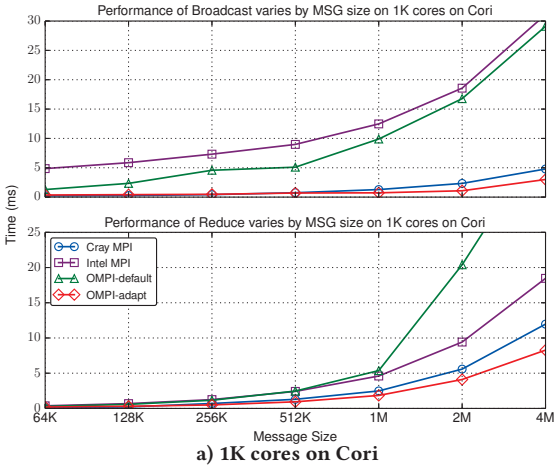


a) 1K cores on Cori

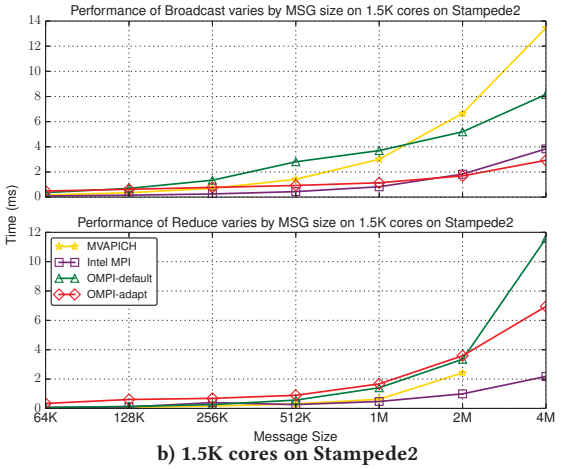


b) 1.5K cores on Stampede2

Figure 8: Performance of broadcast and reduce on CPU data, compared with all Topology-aware in Intel MPI



a) 1K cores on Cori



b) 1.5K cores on Stampede2

Figure 9: Performance of broadcast and reduce on CPU data varies by message size

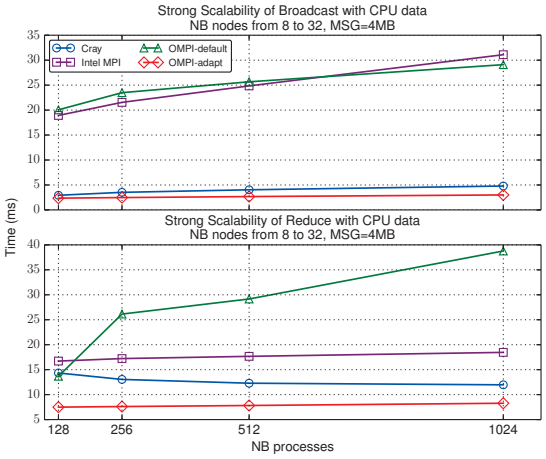


Figure 10: Strong scalability of broadcast and reduce with CPU data varies by number of nodes on Cori, message size is 4 MB

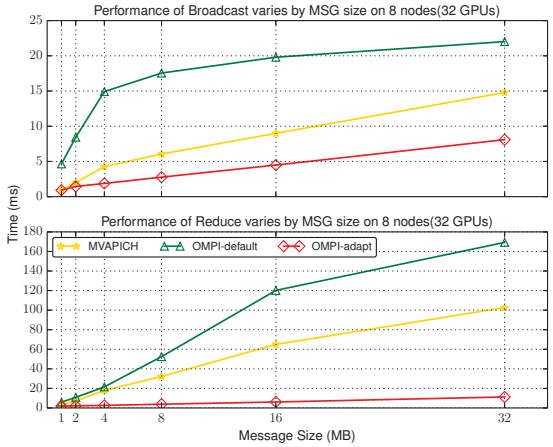
machines. Cray MPI does not support Omni-Path interconnect, so it is not tested on Stampede2. Similarly, MVAPICH does not support Aries interconnect, so it is not tested on Cori. The default collective module in Open MPI is the tuned module, which can switch algorithms based on different parameters. This is shown in Figure 9a), where the OMPI-default broadcast algorithm changed after 256KB. In OMPI-adapt, both the broadcast and the reduce operations are pipelined algorithms, in which messages are split into several segments. In order to better understand the performance graph, we adopt Hockney's cost model [16] to model the cost of collective operations. This model assumes that the time to send a message of size m between two nodes is $T = \alpha + \beta m$, where α is the latency (or startup time) per message, independent of message size, and β is the transfer time per byte or reciprocal of network bandwidth. For the reduction operation, we assume that the time spent in computation on data in a message of size m is γm, where γ is computation time per byte. Therefore, the entire time of sending a message of size m between two processes is $T = \alpha + \beta m + \gamma m$.

A perfect pipeline needs to meet two criteria: a large enough segment size and a sufficient number of segments. If the segment size is too small, message latency as α in Hockney's model becomes dominant, preventing the full utilization of network bandwidth. If there are not enough segments, the pipeline initialization time will be predominant and the overall performance will be affected. Therefore, it is difficult for small messages to meet both criteria. This means the ADAPT framework will show lesser improvement over other implementations when the messages are small. For larger messages, the benefit of concurrent communication in the ADAPT framework becomes the dominant factor. On Cori, OMPI-adapt provides 10×, 10× and 1.6× speedup against OMPI-default, Intel MPI and Cray MPI for broadcast operations and 5×, 2× and 1.5× speedup for reduction operations when the message size is 4MB. For the same message size on Stampede2, compare with OMPI-default, Intel MPI and MVAPICH, OMPI-adapt achieves 2.8×, 1.3× and 4.6× speedup for broadcast. OMPI-adapt's reduce operation is slower than Intel MPI's on Stampede 2 and the reason is explained in the previous section.

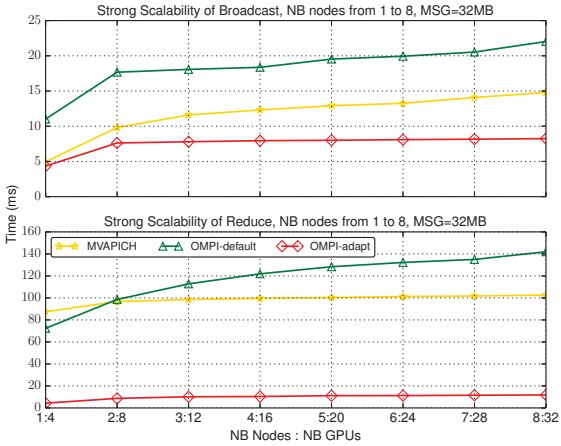
Scalability is another important factor of MPI libraries. Figure 10 shows the performance of strong scaling for broadcast and reduction operations with a 4MB message on Cori. OMPI-adapt uses the chain algorithm as the communication tree for all topology levels groups based on [29]. Since ADAPT allows concurrent communications over independent paths, the cost of broadcast and reduction can be calculated through the longest chain in the communication tree. Based on Hockney's Model, the cost of chain is $T = (P + n_s - 2) \times (\alpha + \beta m)$ [29] where P is number of processes participating in a collective operation and n_s is the number of segments. If the message size is large enough to ignore the cost of pipeline initialization, the cost of the chain algorithm can be treated as $T = n_s \times (\alpha + \beta m)$. Thus, theoretically, performance does not depend on the number of processes within the chain. Therefore, as seen in Figure 10, the time of our broadcast and reduction does not increase significantly with the number of processes, and the operations tend to be stable as the number of nodes increases. Compared with other MPIs, OMPI-adapt consistently achieves the best strong scalability, thanks to its event-driven design and the topology-aware communication tree.

5.2.2 Collective Communications with GPU Data. Figure 11a) shows the time of broadcast and reduction operations with GPU data on 8 nodes (32 GPUs in total). With the help of data caching in CPU memory to minimize traffic over PCI-Express and the minimal dependencies in ADAPT's design, OMPI-adapt outperforms MVAPICH2 and OMPI-default by 2–3 times for broadcast. For the reduction, in addition to the same advantages as broadcast, OMPI-adapt benefits from the asynchronous reduction operations on GPUs. As a result, OMPI-adapt reduction is almost 10 times faster than the other two MPI libraries. It should also be noted that with OMPI-adapt the CPU remains available for other computation, as most of the operations are executed either by direct memory access (DMA) engines or by the GPUs.

Figure 11b) presents the result of a strong scaling experiment for broadcast and reduction with a fixed message size and a variable number of nodes. Since each topology level occupies an independent communication path, each level is able to achieve perfect overlap. According to the Hockney model presented above and similar to the CPU results, the performance of the GPU broadcast and reduce should be unaffected by the increase in the number of GPUs. This is proved by the almost ideal strong scalability result shown in Figure 11b). When the number of nodes is larger than 1, inter-node and intra-socket communication of the root process occupies the same direction as the PCI-Express, leading to performance drops if not correctly handled by the MPI library. As discussed in Section 5.2.1, OMPI-default uses a decision tree to guide collective algorithm selection; however, the decision tree strategy was not designed for GPUs, thus it would not select the optimal algorithm for multi-GPU settings, leading to significantly slower performance compared to MVAPICH2 and OMPI-adapt. For example, when using only one node, OMPI-default does not use the chain algorithm (which would be optimal), resulting in a significant performance drop. With ADAPT's design, we achieve better scalability over OMPI-default and MVAPICH2.



a) Performance of broadcast and reduce with GPU data varies by message size using 8 nodes 32 GPUs



b) Strong scalability of broadcast and reduce with GPU data varies by number of nodes, message size is 32 MB

Figure 11: Performance of broadcast and reduce on PSG cluster

Table 1: Performance of ASP with 1K cores on Cori

	Cray	Intel MPI	OMPI-adapt	OMPI-tuned
Communication (s)	2.98	15.26	1.99	14.18
Total Runtime (s)	6.20	18.46	5.21	17.40

5.3 Application Performance

We choose ASP [30] to evaluate how OMPI-adapt performs in real application. ASP uses the parallel Floyd-Warshall algorithm to solve the all pairs shortest path problem. In the beginning of each iteration, one process broadcasts a row of the square matrix representing edges weight to all peers in the communicator, in order to distribute the workload. The outer loop of the algorithm iterates on rows, until the entire matrix is processed. Overall, if a matrix size is N, the ASP contains N broadcast of $N \times type\_size$ bytes each. Compare to the light computation, the communication is dominant and the *MPI_Bcast* takes the major time of the runtime of ASP.

Table 1 shows the performance of ASP with problem size equals 256K. For ADAPT, the communication takes 38% of the runtime. This number rises to 48% for Cray MPICH, and more than 80% for Intel MPI and OMPI-default. Therefore, the event-driven design combined with topology-aware tree in OMPI-adapt shows a significant improvement for this application, even with smaller message sizes (1MB).

6 RELATED WORK

6.1 Performance Interference

With the increasing scale of HPC systems and more potential sources of performance interference, finding a way to alleviate the effects of interference is crucial to large-scale parallel application. One way is to eliminate the interference itself. Performance interference introduced by system noise can be reduced by system designers [2, 42]. There are also efforts to reduce other sources of performance interference. In [43], the authors minimize interference between the simulation and the situ analytics by using fine-grained scheduling to get idle resources. Another way is to alleviate the propagation of the performance interference. [35] shows how system noise impacts the performance of the collective operation and [39] highlights how non-blocking collective operation has potential to mitigate certain types of performance interference.

Despite in-depth research on the causes and impacts of performance interference mentioned above, our work is the first to focus on the implementation of MPI collective operation, which performs well against such noise.

6.2 Topology-Aware Collective Operations

Several related works use the topology-aware idea for collective operations to take advantage of communication difference among the levels in the network. MagPIe [21] creates hierarchical algorithms for clustered wide-area systems to minimize the data transfers on the slowest links. MPICH2 [44] implements several collective operations that exploit knowledge of the underlying topology. But these works only consider two network layers. Karonis et al. [20] extends the previous work and presents a multi-level, topology-aware tree to support more network layers. Later, MVAPICH2 [19, 33] introduced neighbor-joining techniques to detect network topology on switch levels, and adds one more level in the network hierarchy for collective operations. However, all these approaches focus on exploiting an increasing number of network topology levels. From a performance standpoint, they provide increased levels of performance compared with single-level approaches, but their inter- and intra-level communications are dissociated and do not cooperate tightly, leading to a deficit in communication overlap between different topology levels.

Other researchers have tried to benefit from node-level shared memory and propose hierarchical collective operations. Tipparaju et al. [34] use shared memory as an intermediate buffer to reduce the number of memory copies. Cheetah [15] is a hierarchical collective communication framework that constructs a directed acyclic graph (DAG) based on the characteristics of communication topology. Each node in the DAG is a collective operation. It can take

advantage of shared memory for intra-node communications and InfiniBand P2P or CORE-Direct for inter-node communications. Parsons et al. [28] decouples the choice of inter-node and intra-node communication algorithms. However, all these previous works completely lack communication overlap between levels. HierKNEM [23] enables tight collaboration between the collective algorithms pertaining to different layers of the hierarchy. It combines KNEM (a Linux kernel for memcopy in shared memory), pipelining, and hierarchical ideas to allow overlap of inter-node and intra-node communication. It only supports two topology levels and in each level the tree is fixed.

With a carefully built topology-aware tree, our collective framework can support multiple topology levels, and each level can select a different algorithm based on the characteristics of the level, such as the number of processes, message size, available bandwidth, and other hardware characteristics.

6.3 CUDA-Aware Collective Operations

State-of-the-art MPI libraries such as Open MPI [40] and MVAPICH2 [32, 36] provide CUDA-aware P2P and collective communications. But they never consider offloading reduction operations into GPU; therefore, they are not able to exploit the GPU parallelism to handle large parallel reduction operations. Later, Chu et al. [5] and Oden et al. [26] proposed CUDA-aware reduce operations by leveraging CUDA kernels to handle reduction. Chu et al. [6] present a hardware multicast-based broadcast which benefits from IB hardware multicast. However, none of them encompasses network hierarchical topology for heterogeneous GPU-based clusters. NVIDIA introduced The NVIDIA Collective Communications Library (NCCL) [25], which targets multi-GPU platforms. However, their broadcast and reduce only use chain algorithms, are not able to adapt based on topology, and they are not really equivalent to the MPI collective. Awan et al. [1] integrated NCCL into MVAPICH2 to provide hierarchical broadcast operations by using NCCL to handle intra-node communications. However, similar to MVAPICH2 in CPU clusters discussed before, there is no communication overlap between different topology levels. Our topology-aware framework has been extended to GPU clusters and provides communication overlap between different levels, as discussed in Section 4.

6.4 Event-driven Collective Operation

Event-driven programming is a long existing programming model which is used to solve various problems, including decreasing the memory overhead in embedded systems [9], achieving high throughput in server applications [27] and forming service objects across a mobile network [7]. This model is also used to handle I/O operations (event-driven I/O [3] [8]). Normally, I/O operations are extremely slow compared to the processing of data, therefore, the CPU's computing power is wasted if it is blocked to wait for I/O operations. Alternatively, with the event-driven design, instead of waiting for I/O operations, CPU can continues to work on other jobs until it gets a notification of an I/O operation is completed.

In this paper, we use event-driven design to implement MPI collective operations. To the best of our knowledge, the only paper related to this is [18]. In this paper, the authors define GOAL, an abstract domain-specific language able to dynamically represent a collective operation as a set of scheduled operations and then execute the DAG using a scheduler. We use a similar idea to break down the collective operation into P2P operations, but our framework is the first to use the event-driven idea to alleviate the effects of performance interference and integrate the event-driven collective framework with the topology-aware tree to enable efficient, heterogeneous topology-aware collective operation. Also compared to previous work, we control concurrent communication, allow segmentation of a message to achieve better performance and we never build the entire collective schedule description.

7 CONCLUSION

MPI implementations need to adapt to the increasing scale and complexity of HPC systems to fulfill users' expectations. In this paper, we address this problem by presenting ADAPT, a collective communication framework in Open MPI based on an event-driven infrastructure. Through events and callbacks, ADAPT relaxes synchronization dependencies and maintains the minimal data dependencies. This approach provides more tolerance to system noise. Adding to this capability, ADAPT supports fine-grained, multi-level topology-aware collective operations which is able to exploit the parallelism of heterogeneous architectures. We demonstrate experimentally that (1) our framework is less affected by noise than other state-of-the-art MPI libraries; (2) it outperforms most state-of-the-art MPI libraries on heterogeneous architectures using CPU and GPU data; and (3) it demonstrates an almost ideal strong scalability as the number of nodes increases. We are now looking at increasing the collective communications coverage, enabling non-blocking collective communications with asynchronous progress, and integrating network switch level information to further improve its adaptability.

REFERENCES

[1] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda. 2016. Efficient Large Message Broadcast Using NCCL and CUDA-Aware MPI for Deep Learning *(EuroMPI 2016)*. 15–22.

[2] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. 2006. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *2006 IEEE International Conference on Cluster Computing*. 1–12.

[3] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. 2003. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*. 371–386.

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. 2010. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 180–186.

[5] C. H. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and D. K. Panda. 2016. CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 726–735.

[6] C. H. Chu, X. Lu, A. A. Awan, H. Subramoni, J. Hashmi, B. Elton, and D. K. Panda. 2017. Efficient and Scalable Multi-Source Streaming Broadcast on GPU Clusters for Deep Learning. In *2017 46th International Conference on Parallel Processing (ICPP)*. 161–170.

[7] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. 2007. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the*. 3–12. https://doi.org/10.1109/SCCC.2007.12

[8] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. 2002. Event-driven Programming for Robust Software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop (EW 10)*. ACM, 186–189.

[9] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*. ACM, 29–42.

[10] K. B. Ferreira, P. Bridges, and R. Brightwell. 2008. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[11] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti. 2010. The Impact of System Design Parameters on Application Noise Sensitivity. In *IEEE Cluster 2010*. 146–155.

[12] Kurt B. Ferreira, Patrick Widener, Scott Levy, Dorian Arnold, and Torsten Hoefler. 2014. Understanding the Effects of Communication and Coordination on Checkpointing at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. 883–894.

[13] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/. (September 2012).

[14] V. W. Freeh, Feng Pan, N. Kappiah, D. K. Lowenthal, and R. Springer. 2005. Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster. In *19th IEEE International Parallel and Distributed Processing Symposium*. 4a–4a.

[15] R. Graham, M. G. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer. 2011. Cheetah: A Framework for Scalable Hierarchical Collective Operations. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 73–83.

[16] Roger W. Hockney. 1994. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (March 1994), 389–398.

[17] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. 1–11.

[18] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2009. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*. 574–581.

[19] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. 2010. Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8.

[20] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. 2000. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *IPDPS 2000*. 377–384.

[21] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. 1999. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems *(PPoPP '99)*. 131–140.

[22] Scott Levy, Bryan Topp, Kurt B. Ferreira, Dorian Arnold, Torsten Hoefler, and Patrick Widener. 2014. *Using Simulation to Evaluate the Performance of Resilience Strategies at Scale*. Springer International Publishing, Cham, 91–114.

[23] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra. 2012. HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 970–982.

[24] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener. 2016. Scheduling In-Situ Analytics in Next-Generation Applications. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 102–105.

[25] NVIDIA. 2016. NCCL. https://github.com/NVIDIA/nccl. (2016).

[26] L. Oden, B. Klenk, and H. Fröning. 2014. Energy-Efficient Collective Reduce and Allreduce Operations on Distributed GPUs. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 483–492.

[27] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 1999. Flash: An efficient and portable Web server. (1999).

[28] Benjamin S. Parsons and Vijay S. Pai. 2014. Accelerating MPI Collective Communications Through Hierarchical Algorithms Without Sacrificing Inter-Node Communication Flexibility *(IPDPS '14)*. 208–218.

[29] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing* 10, 2 (2007), 127–143.

[30] A. Plaat, H. E. Bal, and R. F. H. Hofman. 1999. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. 244–253.

[31] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. 2009. Two-tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.* 35, 12 (Dec. 2009), 581–594.

[32] A. K. Singh, S. Potluri, H. Wang, K. Kandalla, S. Sur, and D. K. Panda. 2011. MPI Alltoall Personalized Exchange on GPGPU Clusters: Design Alternatives and Benefit. In *2011 IEEE International Conference on Cluster Computing*. 420–427.

[33] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda. 2012. Design of a Scalable InfiniBand Topology Service to Enable Network-topology-aware Placement of Processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. Article 70, 12 pages.

[34] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar Panda. 2003. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters *(IPDPS '03)*. 84.1–.

[35] Nisheeth K. Vishnoi. 2007. *The Impact of Noise on the Scaling of Collectives: The Nearest Neighbor Model*. Springer, 476–487.

[36] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K. Panda. 2011. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development* 26, 3 (2011), 257.

[37] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. 2016. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*. 20.

[38] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 41–53.

[39] Patrick M Widener, Scott Levy, Kurt B Ferreira, and Torsten Hoefler. 2016. On Noise and the Performance Benefit of Nonblocking Collectives. *Int. J. High Perform. Comput. Appl.* 30, 1 (Feb. 2016), 121–133.

[40] Wei Wu, George Bosilca, Rolf vandeVaart, Sylvain Jeaugey, and Jack Dongarra. 2016. GPU-Aware Non-contiguous Data Movement In Open MPI. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 231–242.

[41] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. 2015. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 156–165. https://doi.org/10.1109/IPDPS.2015.56

[42] K. Yoshii, K. Iskra, H. Naik, P. Beckmanm, and P. C. Broekema. 2009. Characterizing the Performance of Big Memory on Blue Gene Linux. In *2009 ICPP Workshops*. 65–72.

[43] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. 2013. GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[44] Hao Zhu, David Goodell, William Gropp, and Rajeev Thakur. 2009. *Hierarchical Collectives in MPICH2*. Springer Berlin Heidelberg, Berlin, Heidelberg, 325–326.