HAN: a Hierarchical AutotuNed Collective Communication Framework

Xi Luo[1], Wei Wu[2*], George Bosilca[1*], Yu Pei[1], Qinglei Cao[1],
Thananon Patinyasakdikul[3], Dong Zhong[1], and Jack Dongarra[1]

[1]University of Tennessee, Knoxville, TN, USA
[2]Los Alamos National Laboratory, Los Alamos, NM, USA
[3]Cray, Bloomington, MN, USA

***Abstract*—High-performance computing (HPC) systems keep growing in scale and heterogeneity to satisfy the increasing computational need, and this brings new challenges to the design of MPI libraries, especially with regard to collective operations.**

To address these challenges, we present "HAN," a new hierarchical autotuned collective communication framework in Open MPI, which selects suitable homogeneous collective communication modules as submodules for each hardware level, uses collective operations from the submodules as tasks, and organizes these tasks to perform efficient hierarchical collective operations. With a task-based design, HAN can easily swap out submodules, while keeping tasks intact, to adapt to new hardware. This makes HAN suitable for the current platform and provides a strong and flexible support for future HPC systems.

To provide a fast and accurate autotuning mechanism, we present a novel cost model based on benchmarking the tasks instead of a whole collective operation. This method drastically reduces tuning time, as the cost of tasks can be reused across different message sizes, and is more accurate than existing cost models. Our cost analysis suggests the autotuning component can find the optimal configuration in most cases.

The evaluation of the HAN framework suggests our design significantly improves the default Open MPI and achieves decent speedups against state-of-the-art MPI implementations on tested applications.

***Index Terms*—MPI, hierarchical collective operation, autotuning, cost model**

I. Introduction

The increasing computational need of the scientific computing community requires high-performance computing (HPC) systems to continue to grow in scale and heterogeneity. Compared to fast-growing computation power, the speed of communications falls behind, causing the communications to become bottlenecks in many applications.

Message Passing Interface (MPI) standard provides various communication primitives to facilitate the development of HPC applications, and it is the most widely used programming paradigm in the HPC community. Collective operations, one type of communication primitive defined in the MPI standard, are used to exchange data among multiple processes. As indicated in previous studies [1, 2], collective operations are a critical component of most MPI applications, and their performances are significant factors in determining the performance and scalability of these applications. Hence, it is crucial for an MPI library to provide highly efficient collective operations.

*corresponding authors

A. Hierarchical collective communication framework

HPC systems are becoming more heterogeneous, resulting in increasingly complex hardware hierarchies. To utilize all hardware capabilities at each level of the hierarchies, and improve the performance of collective operations on these systems, collective communication implementations need to adapt and embrace the hierarchical approach. A hierarchical collective operation is usually implemented as a combination of multiple fine-grained collective operations, where each of them handles the communication on one hierarchical level [3]. When implementing a well-performing hierarchical collective communication framework, there are three crucial factors that need to be considered.

First, on each level of the hierarchy, the algorithms of the fine-grained collective operations need to fully exploit the hardware capabilities. For example, traditional tree-based algorithms are sub-optimal for intra-node collective operations as they introduce extra memory copies. To minimize the memory copies, some collective frameworks [4, 5] utilize the shared memory space to exchange data across processes on the same node. The same hardware-aware design goes for inter-node level, as in [6], collective operations need to leverage the full-duplex mode to maximize network bandwidth. Moreover, if network switch level information is available, collective operations can be further optimized [7, 8].

Second, an optimal design of hierarchical collective communication framework should maximize the communication overlap, especially for large messages. From a hardware perspective, data transfers on different levels are mostly independent from each other since they mainly occupy different hardware devices (or different DMA engines). However, from a software implementation perspective, some problems, such as lacking enough segmentation and sharing software resources, would limit the communication overlap.

Last, facing the fast-changing hardware, a hierarchical collective framework needs to be flexible enough to adapt to new architectures, and network capabilities and topologies. In the

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00013

inter-node level, various interconnects have been introduced with different network topologies, such as hypercube [9], polymorphic-torus [10], fat-tree [11], and dragonfly [12]. Inside a node, with adopting co-processors, how these computing units are connected changes drastically as well.

In this paper, we present "HAN" (Hierarchical AutotuNed), a flexible task-based hierarchical collective communication framework in Open MPI, which addresses the three factors discussed before. First, it selects the proper collective frameworks as submodules to utilize the hardware capabilities of each level. Second, it adopts a pipelining technique to overlap communications on different levels. Finally, due to its modularized design it can easily switch out the submodules to adapt to hardware updates. The detailed design of the framework is explained in section III.

B. Autotuning of MPI collective operations

Autotuning is a well-known technique to automatically find the best set of parameters to optimize a certain problem. In the context of collective communications, autotuning optimizes the configuration (algorithms, segment sizes, ...) of a collective operation. There exist several approaches to perform autotuning, but they can be categorized around two methods.

The first approach is screening all possible configurations of a collective operation with benchmarks. An implementation of this approach, such as MPITUNE, is to exhaustively search every possible configuration, in order to identify the best, or the most suitable combination of parameters. This approach is extremely costly, but it has the potential to guarantee the optimal configuration. This method could be usable at small scale; however, its search space explodes as the size of the system increases, and would therefore limit its usage on modern large scale systems. To address this, efforts have been made to reduce the search space with heuristics [13, 14]. However, with more heuristics, more assumptions are made, increasing the opportunities for misprediction, which could reduce the accuracy of the autotuning process.

The second approach is using cost models [1, 14] to estimate the time of collective operations and select the best configuration(s) based on these estimations. Instead of directly measuring the cost of all collective operations, this method only benchmarks a few network specifications, such as gap, bandwidth, and latency, and uses the models to infer the cost of the collective, drastically reducing the cost of autotuning. However, as stated in [1, 14], cost models have their own set of drawbacks, and, in many cases, are not accurate enough to find the best configuration as they oversimplify modern heterogeneous systems. Conventional models such as Hockney [15], LogP [16], LogGP [17] and PLogP [18] assume the cost of MPI point-to-point (P2P) operations between any two processes remains constant. However, this assumption is no longer valid on heterogeneous systems, where the cost of P2P varies a lot based on the location of processes. SALaR [2] extends LogGP with different gaps (Gs) for different networks to model a hierarchical MPI_Allreduce, but its G is fixed in each level. The fixed G limits this model to large messages, one segment of which can saturate the network bandwidth.

Besides network heterogeneity, other factors, such as the congestion on a switch and the shared resources of communications on different levels, are not considered in these cost models. Previous study [19] suggests when one process communicates with many processes concurrently, the congestion on that process could drastically affect overall communication performance. Others [2, 20, 21] assume data transfers on different levels, such as inter- and intra-node, are totally independent when modeling hierarchical collective operations. However, practical experiments (section III-A) shows different levels are not entirely independent, and their communications can not be perfectly overlapped because of the shared resources, such as memory buses and DMA engines.

To achieve a fast and accurate autotuning, we propose a drastically different approach, combining the benefits of the previous two methods and using a task-based autotuning component to handle the communications and their potential overlap. Our approach utilizes a cost model; but instead of relying on network specifications, our cost model is based on empirical benchmarking of independent sub-communication patterns (or tasks), thanks to the task-based design of HAN. Compared to the first method, since we only benchmark tasks instead of a whole collective operation, our method can reduce the search space significantly, which is discussed in section III-C. Compared to the second method, our model improves the accuracy since it considers more factors, i.e. different bandwidths of different levels, changing gap with increasing message sizes, congestion on a process and overlap rate of communications on different levels. All of these factors can affect the performance of collective operations, but as they are hard to model, they have been usually excluded from the existing models; while in our autotuning approach, instead of modeling them, we choose to directly measure their influence on tasks to provide better estimations.

C. Contribution

The key contributions of this paper are:

- **Task-based hierarchical collective operations.** Our HAN framework breaks a hierarchical collective operation into a sequence of smaller collective communication patterns (or tasks), with each task containing fine-grained collective operations. These fine-grained collective operations are selected from available submodules to utilize the hardware capabilities of the intra- and inter-node level and overlap communications on these levels.
- **Task-based autotuning.** We present a cost model based on empirical results of tasks used in HAN, along with an cost model based autotuning component. Unlike other autotuners, such as MPITUNE of Intel MPI, our autotuning component operates on tasks instead of a whole collective operation. Because we can reuse the cost of tasks, our autotuning component greatly reduces the tuning time, while providing a similar level of accuracy.

The rest of this paper is organized as follows: section II relates this work to previous efforts; section III describes the implementation of our framework with MPI_Bcast and MP_Allreduce as examples and presents our autotuning method; section IV evaluates the performance of our design; and section V concludes.

II. Related Work

A. Hierarchical Collective Operations

To take advantage of the communication differences at different hardware levels, some previous studies manage to minimize data transfers on the slow communication channels by grouping processes based on their locations. MagPie [22] optimizes collective operations for wide area systems, where processes are group by clusters. In contrast, MPICH2 [23] groups processes by nodes to limit the number of inter-node communication. Later, the groups are further divided to explore more levels of hardware hierarchies [24]. MVAPICH2 [7, 8] adds another hierarchy level with the network switch information.

Others focus on strategies to select leaders of the groups at each level of the hierarchy. Parsons et al. [25] select leaders dynamically to overcome imbalanced process arrival times, and Bayatpour et al. [20] create multiple leaders to better explore the parallelism in networks for MPI_Allreduce. These methods provide better performance compared to the isotropic approaches [26], which assume equal cost for any pair of processes; but since they are not able to overlap communications on different levels, their performance for big messages would be sub-optimal.

Other approaches overlap the communication on two levels, intra-node and inter-node. HierKNEM [5] tries to make intra-node communication asynchronous by offloading intra-node communication with KNEM [27]. SALaR [2] implements an inter-node allreduce with non-blocking one-sided communication to make its inter-node communication asynchronous. ADAPT [28] allows asynchronous progressing on both levels by adopting an event-driven design and utilizing non-blocking P2P operations on both levels. In HAN, our task-based design allows asynchronous communication on any level.

Cheetah [26] uses a Directed Acyclic Graph (DAG) to describe hierarchical collective operations, which is similar to our task-based design. However, our framework provides two advantages as compared to it. First, our framework has a pipelining mechanism that can overlap communications on different levels; second, Cheetah lacks an autotuning component. Without an autotuning component, its best performance cannot always be achieved on a given machine.

B. Autotuning of Collective Operations

In [13], Vadhiyar et al. notice collective operations may not give good performance in all situations. Hence, they perform an exhaustive search to find the best arguments for every case and use these arguments to automatically tune collective operations. It also provides some heuristic ideas and gradient descent methods to limit the search space. These heuristics are complementary with our approach and they can be combined to further reduce the testing time, as discussed in section III-C. Tuned [29], the current default collective selection mechanism in Open MPI, built its decision functions long ago, on hardware with completely different parameters than most today's HPC machines (a cluster of AMD64 processors using Gigabit Ethernet and Myricom interconnect). Since HPC systems have changed drastically, this default decision is not optimized for current platforms.

With the increasing scale of HPC systems, the search space of exhaustive approaches grows exponentially, rendering this approach unrealistic and resulting in a shift toward an increase use of cost models to guide autotuning. In [1], Pješivac-Grbović et al. use multiple models to estimate the cost of MPI_Bcast. However, as the authors point out in the paper, the cost models are not accurate enough to optimally tune collective operations. SALaR [2] improves the LogGP model with different gaps for different levels. However, even though this model is more accurate than previous cost models for hierarchical collective operations, it fails to find the best configuration directly in most cases. In SALaR, the authors only use the cost model to provide a starting point of its online tuning. Eller et al. [21] further improve the accuracy of a postal model of MPI_Allreduce by considering network congestion, network distance, communication and computation overlap, and process mappings. However, its assumption of the perfect overlap of communications on different levels and only supporting one algorithm make it less suitable for autotuning.

Online tuning is another approach to pinpoint the best configuration by timing collective operations and changing the configuration (or the decision function) dynamically while the MPI application is running. With this approach, STAR-MPI [30] selects algorithms dynamically, and SALAR [2] refines its segment size online. The time to converge to the best selection is uncertain, and the cost of timing and maintaining the decision matrix online inevitably brings overhead. Both downsides can hurt the performance of collective operations, which limits the usage of this approach to general cases, and that is why we choose offline tuning in the HAN framework.

III. Design

As mentioned before, our goal in HAN is not to provide different implementations of MPI collective communication algorithms, but to build upon the existing collective communication infrastructure, reuse these existing algorithms as submodules, and combine them to perform efficient and hierarchical collective operations. HAN groups processes based on their physical locations in the hardware hierarchies, e.g. node, NUMA-node or even socket level, and hence divides collective operations into multiple levels. While such information is generally available from the MPI runtime (PMIx, Hydra), the only portable MPI 3.1 function to expose architectural information (MPI_Comm_split_type) allows splitting processes intra- and inter-nodes. Therefore, we limit HAN to the topology information obtained through this portable API,

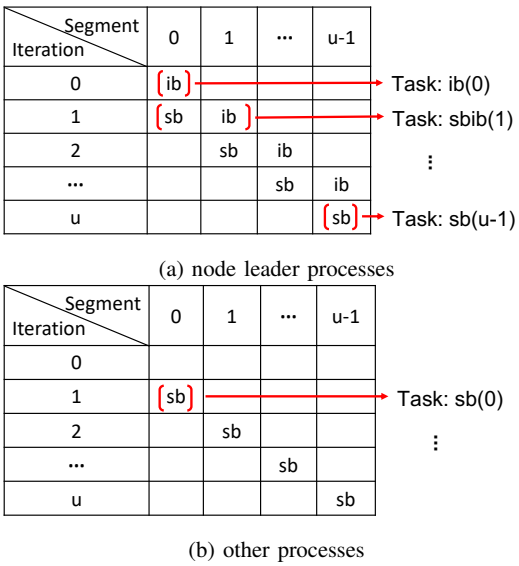
and we only use two levels (intra- and inter-node) in the rest of the paper.

The design contains three parts. The first part is finding suitable submodules for each level. As discussed in the introduction, overlapping communications on different levels is an important factor to the performance of hierarchical collective operations. To attain good overlap of inter- and intra-node communications, HAN relies on non-blocking collective operations for inter-node communication, as from a practical standpoint there are no good intra-node non-blocking collective algorithms in Open MPI. Hence, HAN utilizes the only two modules that support non-blocking collective operations in Open MPI: (1) Libnbc [31], a default legacy module, and (2) ADAPT [28], a new module with an event-driven design. As for intra-node collective operations, Open MPI provides two modules, SM and SOLO. SM is a module utilizing shared memory buffers to exchange data between processes; and SOLO is an experimental module that relies on MPI one-sided communication. Both modules take advantage of the shared memory space; however, due to the differences in algorithms and implementations, SM has better performance for small messages while SOLO performs significantly better as the communication size increases.

The second part of the design is the use of a task-based approach to organize and overlap communications on different levels. Our framework utilizes a pipelining technique [27, 28] by dividing a message into smaller segments and sending them in order, to increase the overlap between network communications. In HAN, segments are transferred via tasks. To perform a hierarchical collective operation, each task contains one or more finer-grained collective operations from different submodules. With the task-based design, the underlying submodules used for collective operations are interchangeable, allowing our framework to adopt submodules for new architectures easily.

The last part of our design is to provide an autotuning component using a novel cost model. Some submodules, such as ADAPT, offer multiple algorithms to each collective operation. For example, MPI_Ibcast in ADAPT contains various algorithms, such as chain, binary tree, and binomial tree. For each algorithm, the underlying configurations, such as segment size, can also affect the performance of the collective operations. Therefore, we need an autotuning component to pinpoint the optimal configuration. We take advantage of the task-based design of HAN to create a new cost model based on the empirical costs of tasks. Costs of tasks are obtained by benchmarking submodules. Since submodules are tightly coupled in our framework, testing the performance of an individual submodule is not sufficient to represent the overall performance. To accurately estimate the performance, we benchmark the submodules when they are working together and use these results in our cost model to estimate the cost of a collective operation and perform autotuning.

In the following sections, we use MPI_Bcast and MPI_Allreduce as examples to present the design of MPI one-to-all and all-to-all collective operations in HAN. Similar designs can be extended to other collective operations, such as MPI_Reduce, MPI_Gather, and MPI_Allgather, as long as the collective operations can be divided into a serial of tasks.



(a) node leader processes

(b) other processes

Fig. 1: Design of MPI_Bcast

A. MPI_Bcast

1) Implementation: MPI_Bcast is a widely used one-to-all, or rooted, collective operation, which propagates data from the root to all other processes within an MPI communicator. Figure 1 shows the implementation of MPI_Bcast in HAN. Starting from the root, each segment firstly goes through an inter-node broadcast (ib) to reach node leaders; then, each node leader issues an intra-node broadcast (sb, s stands for shared memory) to distribute the segment to the other local processes. Since ib and sb mainly occupy different hardware paths, these two broadcasts have the potential to overlap. To maximize this overlap, we define three types of tasks:

- Task $ib(i)$ means an inter-node broadcast of segment i.
- Task $sbib(i)$ includes an intra-node broadcast of segment $i-1$ received in the previous iteration and an inter-node broadcast of segment i.
- Task $sb(i)$ means an intra-node broadcast of segment i.

Assuming there are u segments in total. With the task-based design, to perform a hierarchical MPI_Bcast, node leaders execute $ib(0)$, $sbib(1)$, ... $sbib(u-1)$ and $sb(u-1)$, and the other processes execute $sb(0)$, ... $sb(u-1)$, as in figure 1.

2) The Cost Model: To find the optimal configuration of MPI_Bcast in our framework, it is crucial to have an accurate cost model. We consider the cost of a collective operation to be the longest time among all the processes, since the cost of a collective operation on each process may be different depends on implementations. This definition has been used by multiple cost models [1, 32], and it is the maximum value reported by Intel MPI Benchmark (IMB) [33] and OSU Benchmark [34].

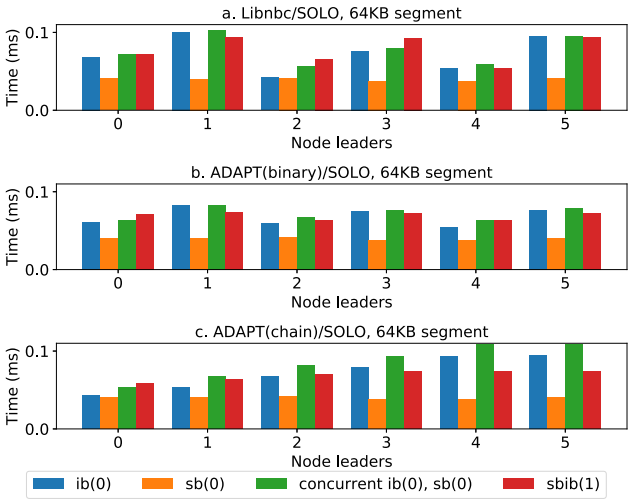


Fig. 2: Cost of tasks ib, sb and $sbib$ (0 is the root)

We compute the cost of MPI_Bcast by aggregating the cost of tasks in each iteration in figure 1, so the time spent in node leader processes is:

$$\max_i(T_i(ib(0)) + T_i(sbib(1)) + ... + T_i(sbib(u-1)) + T_i(sb(u-1))), \quad (1)$$

and the time spent in the other processes is:

$$\max_i(T_i(ib(0)) + T_i(sb(0)) + ... + T_i(sb(u-1)), \quad (2)$$

where u is the total number of segments, and $T_i(t)$ means the duration or cost of task t on the process i. Usually, the cost of $sbib(x)$ is larger than $sb(x)$ since it has an extra ib to do; therefore, comparing equation 1 and 2, we use the time spent in node leader processes (equation 1) as the cost of MPI_Bcast.

Since $ib(0)$ is the first task, we assume each process issues it simultaneously. Hence, its cost can easily be measured by a simple benchmark using a loop around a timed task. The blue bars in figure 2 show the benchmark results of $ib(0)$ on each node leader with rank 0 as the root process, when transferring 64KB segments on 6 nodes with different configurations. These results suggest that different submodules and algorithms behave differently and every node leader finishes $ib(0)$ at a different time.

The last task $sb(u-1)$ only contains an intra-node broadcast, which is independent of the processes on the other nodes. Since the segment size is the same among all segments, we use $T_i(sb(0))$ to represent $T_i(sb(u-1))$. The cost of $sb(0)$ can be measured the same as $ib(0)$, and its result is shown as the orange bars in figure 2.

$T_i(sbib(1)) + ... + T_i(sbib(u-1))$ contributes to the major cost of MPI_Bcast when u is big enough. To get an accurate estimation of this part, two essential factors need to be considered.

The first factor is the overlap of ib and sb. ib mainly operates on the interconnect between nodes, while sb communicates over the memory bus, which means these two broadcasts can be overlapped to some degree. Thus, $T_i(sbib(x))$ should be less than $T_i(ib(x))+T_i(sb(x))$. Prior studies [2, 21] assume the overlap of communications on different levels is perfect, which suggests $T_i(sbib(x)) = max(T(ib(x)), T(sb(x)))$. However, it is not always true. The overlap may not be perfect because: (1) ib needs to push the data back to memory which competes with sb for the memory bus; (2) in single-threaded MPI, ib and sb share the same CPU resource to progress, which affects the performance of both when they are running simultaneously. The blue, orange and green bars in figure 2 shows the cost of task $ib(0)$, $sb(0)$ and concurrent $sb(0)$ and $ib(0)$ (issue an ib with an sb simultaneously and wait for them to complete), respectively. It proves that no matter what algorithm is used to perform collective operations, the overlap between ib and sb is significant, but usually not perfect. Thus, neither $T_i(ib(x)) + T_i(sb(x))$ nor $max(T_i(ib(x), T_i(sb(x)))$ can be used to accurately represent $T_i(sbib(x))$.

The second factor is the starting time of $sbib$ on each node leader process. The costs of $ib(0)$ in figure 2 show node leader processes finish $ib(0)$ at different time steps, resulting in different starting time of the following $sbib$. Hence, using the same benchmark as for $ib(0)$ to estimate $sbib$ would not deliver accurate results. To accurately measure $sbib(1)$, we need to delay the participation of each process by the duration of the $ib(0)$ step to simulate the different starting time of $sbib(1)$, and the results of the new benchmark is shown as the red bars in figure 2. The performance differences between the red bars and the green bars prove the importance of considering previous tasks since the only difference between these two is whether there is an $ib(0)$ before timing $sbib$. Therefore, to get the accurate cost of $sbib(1)$, task $ib(0)$ needs to be executed before timing, and to get the accurate cost of $sbib(2)$, task $ib(0)$ and $sbib(1)$ need to be performed. In this way, to get the cost of $sbib(i)$ where $1 \leq i \leq u-1$, all previous tasks from $ib(0)$, $sbib(1)$ to $sbib(i-1)$ need to be executed, which is highly expensive and contains a lot of redundant tasks.

To reduce the redundant tasks, we start to look at the performance trend of the $sbib$ tasks. Figure 3 shows the cost of $sbib(i)$ where $1 \leq i \leq 8$ with different algorithms and submodules on a node leader (node leader 2). All sub-figures show a similar trend that after the first few tasks, the cost of $sbib$ is stabilized. It is because when executing the first few $sbib$s, the pipeline of $sbib$ is not fully constructed, leading to some delays. Once the pipeline is fully constructed, the cost of $sbib$ becomes stable. Thus, instead of benchmarking all $sbib$s, we use the stabilized cost ($sbib(s)$) times $u-1$ to estimate the time of $T_i(sbib(1)) + ... + T_i(sbib(u-1))$. Therefore, equation 1 can turn into:

$$\max_i(T_i(ib(0)) + (u-1)T_i(sbib(s)) + T_i(sb(u-1))) \quad (3)$$

3) Model Validation: Figure 4 shows the comparison of the estimated time calculated from the cost model and the actual time of doing a 4MB MPI_Bcast with different combinations of submodules, algorithms, and segment sizes. In some cases, such as when segment size is 16KB in figure 4.e and figure 4.f,

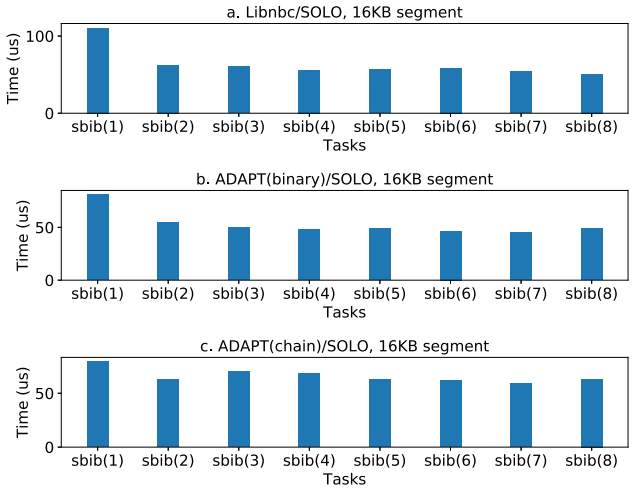


Fig. 3: Cost of tasks on one node leader

because of the inaccurate estimation of the stabilized cost of $sbib(s)$, the prediction is not that accurate. However, as seen in the figure, the cost model is accurate in most cases. Moreover, the trends of the estimated and actual time still match well, which are helpful to find the optimal configuration of MPI_Bcast. Comparing the cost of MPI_Bcast across all the configurations, we can see that the optimal configurations of either estimated or actual cost (the lowest red bar and blue bar) are the same, which is to use 128KB segment with the binary algorithm in the ADAPT submodule for the inter-node communication and the SOLO submodule for the intra-node communication. The result suggests that the cost model can be used for the autotuning of collective communications, which is further discussed in section III-C.

B. MPI_Allreduce

1) Implementation: In this section, as an example of all-to-all collective operations, we describe HAN's implementation of MPI_Allreduce. As indicated in figure 5, a hierarchical MPI_Allreduce requires four steps to transfer one segment: intra-node reduction (sr), inter-node reduction (ir), inter-node broadcast (ib) and intra-node broadcast (sb) (assuming a commutative operation). The implementation of our MPI_Bcast has explored the overlap of collective operations on different levels (i.e. ib with sb); in the implementation of MPI_Allreduce, collective operations overlap even within the same level. For example, ir and ib could overlap if their communications occupy opposite directions of the same inter-node network. Figure 6 compares the performance of ib, ir and concurrent ib with ir of different submodules and algorithms, and strongly indicates a high degree of overlap. To maximize the opportunity of such overlap, when it is possible to specify the algorithm, we select the same algorithm with the same root to perform ir and ib. It is worth mentioning that previous studies [2, 20] use inter-node allreduce to transfer segments across nodes. We choose to break the inter-node allreduce into two explicit operations, the reduce ir and the broadcast ib, to further increase the pipeline and improve the performance for large messages. Considering both kinds of overlaps, we define the following tasks in our MPI_Allreduce:

- Tasks $sr(i)$ and $sb(i)$ represent an sr and sb of segment i, respectively.
- Task $irsr(i)$ includes an ir and sr of segment $i-1$ and i, respectively.
- Task $ibirsr(i)$ contains an ib, ir and sr of segment $i-2$, $i-1$ and i, respectively.
- Task $sbibirsr(i)$ is consisted of an sb, ib, ir and sr of segment $i-3$, $i-2$, $i-1$ and i, respectively.
- Task $sbibir(i)$ includes an sb, ib and ir of segment $i-2$, $i-1$ and i, respectively.
- Task $sbib(i)$ contains an sb and ib of segment $i-1$ and i, respectively.
- Task $sbsr(i)$ is only executed in the processes which are not node leaders. It receives reduced segment $i-3$ from its leader process via an sb, and then reduces segment i to its leader process with an sr.

2) The Cost Model: Similar to MPI_Bcast, we use the maximum time on node leaders to represent the cost of MPI_Allreduce and estimate $sbibirsr(3) + sbibirsr(4) + ... + sbibirsr(u-1)$ with the stabilized cost of $sbibirsr$ ($T_i(sbibirsr(s))$). In this way, the cost of MPI_Allreduce is:

$$\begin{aligned}&\max_i(T_i(sr(0)) + T_i(irsr(1)) + T_i(ibirsr(2)) + \\ &(u-3) * T_i(sbibirsr(s)) + T_i(sbibir(u-1)) + \\ &T_i(sbib(u-1)) + T_i(sb(u-1))).\end{aligned} \tag{4}$$

Then we use a similar benchmark as in section III-A2 to measure the cost of different tasks.

3) Model Validation: Figure 7 compares the time of MPI_Allreduce estimated by the cost model against the measured time. As an example, the cost model predicts that the optimal configuration for an MPI_Allreduce with a 4MB message is to use a 1MB segment with a binary algorithm from the ADAPT submodule and the SOLO submodule for inter- and intra-node communications, respectively. This prediction matches the best measured experimentally.

C. Automatic Tuning

Autotuing is critical to ensure performance for collective operations. Generally, there are two steps in autotuning:

1) Find the optimal configuration for some inputs to generate a lookup table. As shown in table I, the input of autotuning contains four entries: number of nodes n, number of processes per node p, message size m, and the collective operation type t. The output entries of the lookup table are shown in table II.

TABLE I: Inputs of autotuning

Symbol	Description
n	Number of Nodes
p	Number of Processes per Node
m	Message Size
t	Collective Operation Type (Bcast, Reduce, ...)

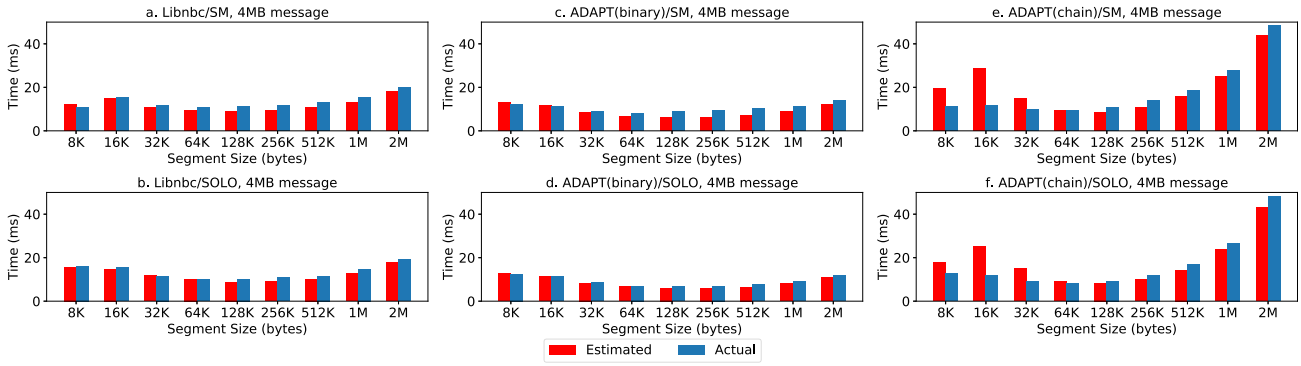
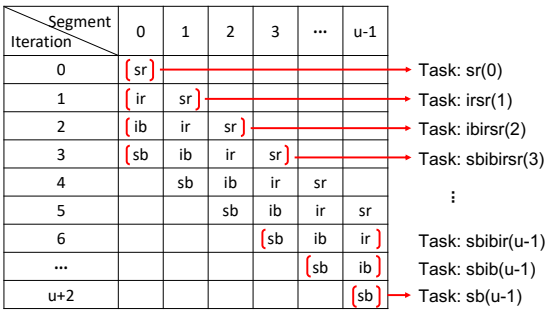
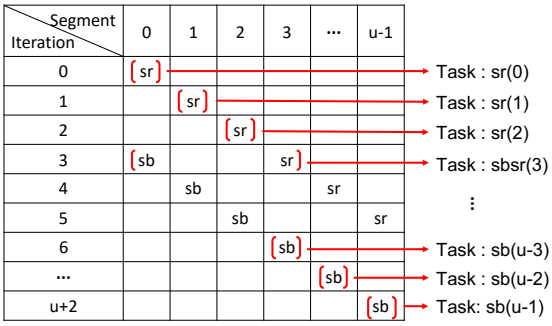


Fig. 4: MPI_Bcast on 64 nodes (12 processes/node) with the combinations of different submodules



(a) node leader processes



(b) other processes

Fig. 5: Design of MPI_Allreduce

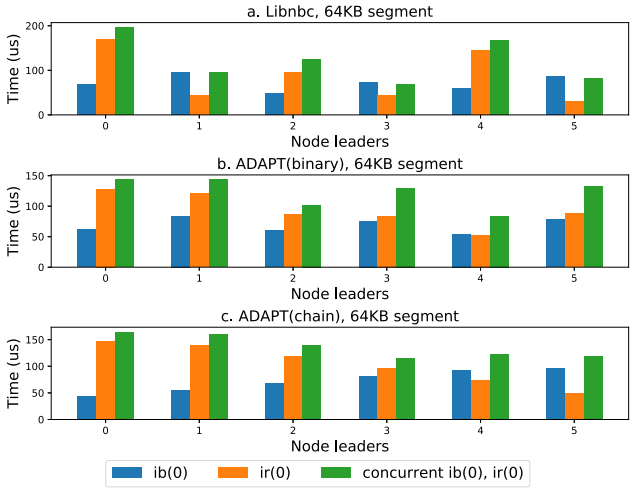


Fig. 6: The overlap between ib and ir (0 is the root)

TABLE II: Autotuned parameters (output of autotuning) for MPI_Bcast and MPI_Allreduce in HAN

Symbol	Description
fs	Segment Size in the HAN module
imod	submodule used for inter-node
smod	submodule used for intra-node
ibalg	Inter-node Bcast Algorithm if supported
iralg	Inter-node Reduce Algorithm if supported
ibs	Inter-node Bcast Segment Size if supported
irs	Inter-node Reduce Segment Size if supported

They are all parameters tuned by our autotuning framework. Usually, m is continuous, but it is impractical to test every message size; thus, most approaches use discrete message sizes such as 4B, 8B, 16B, 32B, ..., to sample the continuous value and form a search space. The same sampling method can be applied to other entries such as n and p.

2) Use the lookup table from the previous step to generate decisions for any inputs (n, p, m and t). As message sizes in the lookup table are not continuous, if the input message size falls between two message sizes in the lookup table, the autotuning component needs to find the optimal configuration for it.

Some studies focus on the second step, where many methods such as quadtree encoding [35], decision trees [36] have been studied to improve its accuracy and/or the code complexity. However, the first step, which takes a significant amount of time and is the foundation to ensure the accuracy of the second step, has not been well studied. In this paper, we focus on reducing the cost of the first step. We use offline autotuning, which only needs to be performed once when installing the MPI to a new machine. It first benchmarks all the tasks within

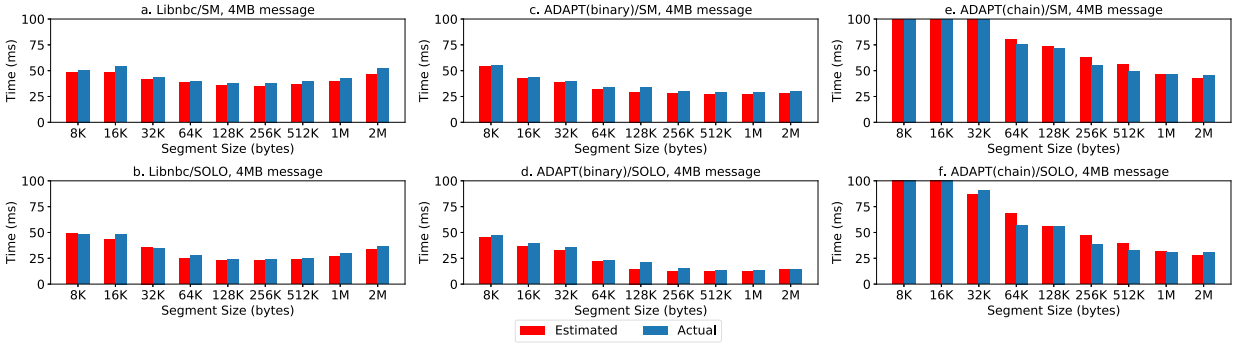


Fig. 7: MPI_Allreduce on 64 nodes with the combinations of different submodules

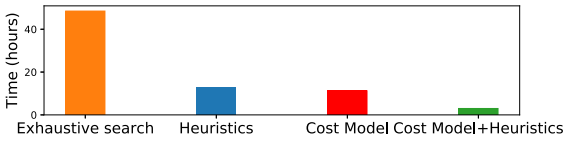


Fig. 8: Time of total searches of MPI_Bcast and MPI_Allreduce on 64 nodes

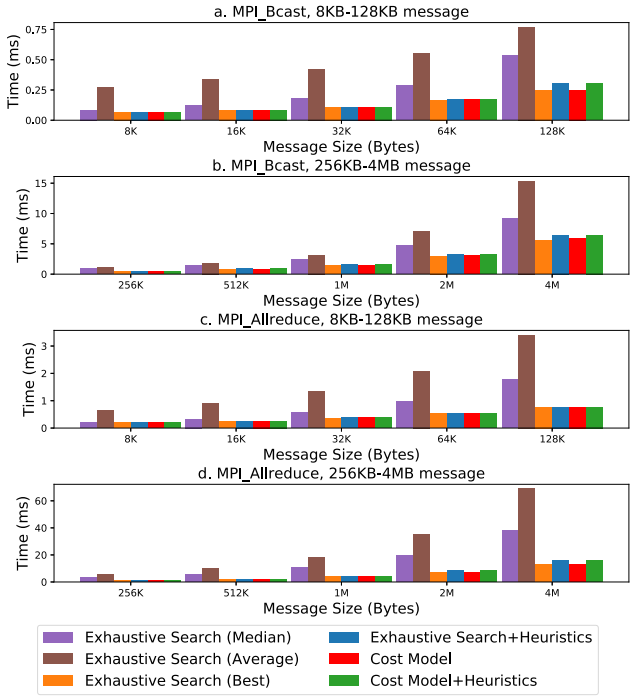


Fig. 9: MPI_Bcast and MPI_Allreduce on 64 nodes (12 processes/node) with different tuning methods

user-defined range; then, it uses the cost model to estimate the cost of collective operations and stores the estimated best configuration for each input to a lookup table in a file.

A straightforward implementation of the first step is to perform an exhaustive search for each input in table I, which tests every possible configuration of a collective operation and then find the fastest one. Take MPI_Bcast for an example. Assuming the sizes of the search spaces of messages, segment sizes, nodes, processes per node are, M, S, N and P, respectively, and the number of available algorithms is A (including submodules $\times$ algorithms per submodule). Exhaustive search tests all possible combinations of S and A for each input in M, N and P. Therefore, the size of the whole search space is $M \times S \times N \times P \times A$. The orange bar in figure 8 show the extremely expensive cost of this exhaustive search, on a small setup (64 nodes, 12 cores per node). It is worth mentioning that even though the exhaustive search is expensive, it guarantees to always find the optimal configuration since its search space would cover all possible configurations. In the context of this study we did the exhaustive search once, and use its results to evaluate the prediction accuracy of our approach.

Thanks to the task-based design of HAN, instead of benchmarking a whole collective operation, we only need to benchmark tasks. As tasks operate on segments, the search spaces needed for one type of task are S, N, P and A. Suppose there are T types of tasks (3 for MPI_Bcast and 8 for MPI_Allreduce); therefore, the size of the whole search space of our approach becomes $T \times S \times N \times P \times A$. For different message sizes, the cost of tasks is reused; hence, compared to the previous approach, HAN can reduce M, one of the largest search spaces, to a constant T. Besides the smaller search space, the cost of performing each search is also much shorter since tasks are just a part of a whole MPI collective operation. Moreover, the cost of tasks can be reused for different types of collective operations, e.g. sb is in both MPI_Bcast and MPI_Allreduce. With the three improvements, the cost of autotuning is drastically reduced. Figure 8 shows the cost of autotuning with different approaches. As seen in the figure, our autotuning component reduces the tuning time by 77% as compared to the exhaustive search. Even though with much fewer searches, our model can still estimate the optimal configuration accurately. The purple, brown, and orange bars in figure 9 show the median, average, and best time-to-completion of MPI_Bcast and MPI_Allreduce of all possible configurations, using exhaustive search. As seen in the figure, both the median and the average time are much higher than the best one, which indicates the importance of

finding the optimal configuration. The red bars in figure 9 show the performance of MPI_Bcast and MPI_Allreduce obtained by our autotuning method, which is exactly the same as the best results (red bars) of exhaustive search in most cases, indicating that our approach produces a similar accuracy as the exhaustive search.

Previous studies [1, 13] suggest that heuristics is an effective way to reduce the search space. In HAN, we assume a prior understanding of the collective submodules and the algorithms available and it can be used to create heuristic strategies. For instance, we only use the SOLO submodule when the segment size is larger than 512KB since experimental results suggest SM has better performance than SOLO for small messages. Besides limiting the selections of submodules, we can also limit the algorithm selections heuristically. For example, we know that the chain algorithm in ADAPT can only perform well when there are enough segments to kick-start the pipelining, we can therefore prevent the chain algorithm from being tested when there are less than a certain number of segments depending on the number of processes involved. Due to the moderate novelty of these heuristics and the space constraint, we will not discuss more the details of the heuristic methods available in HAN, but we will use it to show the resulting reduction in the gathering of the performance for the search. The blue bars in figure 8 show the searching time of the heuristics method, and highlight a drastic reduction, down to 26.8% of the original exhaustive search cost. As mentioned in section II-B, heuristics can be beneficial outside the exhaustive search, and they can be combined with our cost model on the benchmarking of tasks, to further reduce the search space. The cost of the combined approach is shown as the green bars in figure 8, and takes only 4.3% time of the exhaustive search. However, by using the heuristic approaches, we are making assumptions to narrow down the search space, which might result in lower accuracy. The blue and the green bars in figure 9 show the results of applying the same heuristics to the exhaustive search and our approach, respectively, and indicate that adding heuristics produce less accurate results compared to the original approach. To balance the searching cost and accuracy, HAN provides an option for users to enable or disable the heuristics based on the available resources and other requirements.

In conclusion, a careful configuration of our task-based autotuning component can significantly reduce the time of searching the optimal configuration, while still maintain high accuracy.

IV. Performance Evaluation

In this section, we evaluate HAN on two supercomputers: Shaheen II and Stampede2, and compare it with other state-of-the-art MPI libraries using benchmarks and applications.

Shaheen II is a Cray XC40 system equipped with dual-socket Intel Haswell 16 cores CPUs running at 2.3GHz and 128GB DDR4 RAM, using Cray Aries with a Dragonfly topology as interconnect. On Stampede2, we use the Intel Skylake compute nodes; each node has 48 cores with two sockets, and 192GB DDR4 RAM. The nodes are connected via the Intel Omni-Path network.

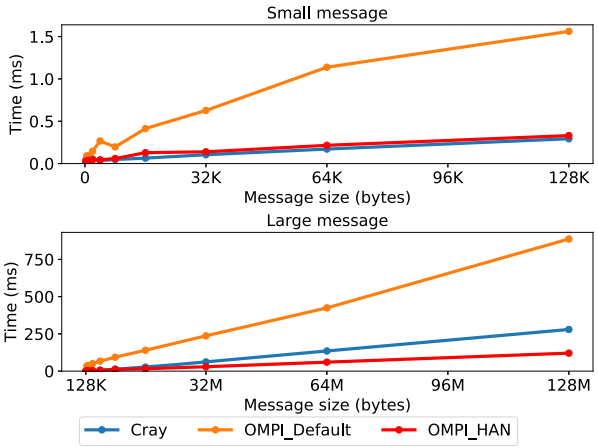


Fig. 10: MPI_Bcast on Shaheen II, 4096 processes

HAN is based on Open MPI 4.0.0. Hence for fair comparisons, we compare with the default Open MPI 4.0.0 on both machines. This default Open MPI is tuned with the conventional methods [29], and HAN is autotuned with our cost model. Additionally, we compare HAN with the system built-in Cray MPI 7.7.0 on Shaheen II, and Intel MPI 18.0.2 and MVAPICH2 2.3.1 on Stampede2 with default tuning.

A. Benchmark

We use IMB [33] to compare HAN against other MPI implementations, on a full range of message sizes. We divide this range in 2 parts: small messages up to 128K, representing the message size range for most scientific applications [37], and large messages up top 128MB, representing the usual message sizes in machine learning and data analytics applications.

1) MPI_Bcast: Figure 10 presents the cost of MPI_Bcast with 4096 processes on Shaheen II. Even though default Open MPI is expected to be tuned, HAN significantly outperforms it: up to 4.72x and 7.35x speed up on small and large messages, respectively, thanks to our task-based hierarchical implementation and cost model.

However, HAN is slightly slower than Cray MPI on small messages. To better understand the performance gap, we measure the P2P performances of both Open MPI and Cray MPI using Netpipe [38]. In most MPI implementations, collective operations rely on the underlying P2P operations to transfer data between processes; therefore, their performance directly impacts the performance of collective operations. As seen in figure 11, when the message size is between 512B and 2MB, Open MPI achieves less bandwidth comparing to Cray MPI especially for messages in the range from 16KB to 512KB, which could explain the performance differences for the small message in figure 10. As message sizes increase, both Open MPI and Cray MPI reach the same peak P2P performance; and in these cases, HAN outperforms Cray MPI up to 2.32x thanks to the communication overlap of different levels.

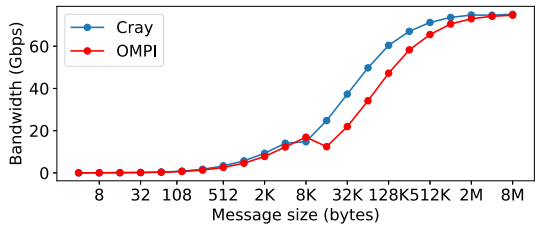


Fig. 11: P2P performance on Shaheen II

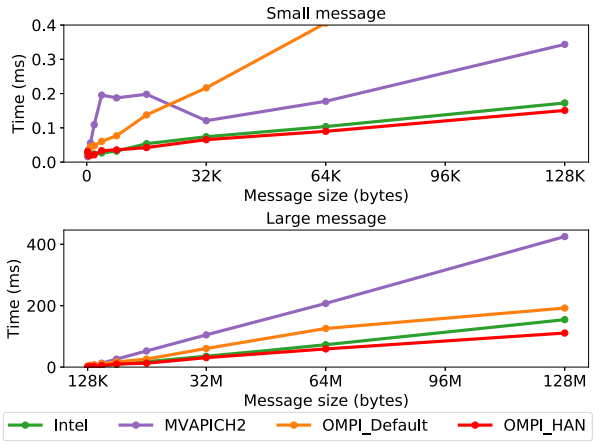


Fig. 12: MPI_Bcast on Stampede2, 1536 processes

Figure 12 exhibits the performance of MPI_Bcast with 1536 processes on Stampede2. On this machine, HAN outperforms every other tested MPI on both small and large messages. It achieves up to 1.15X, 2.28X, 5.35X speedup on small messages, and up to 1.39X, 3.83X, 1.73X speedup on large messages against Intel MPI, MVAPICH2 and default Open MPI, respectively.

2) MPI_Allreduce: Figure 13 and figure 14 present the cost of MPI_Allreduce with 4096 processes on Shaheen II and

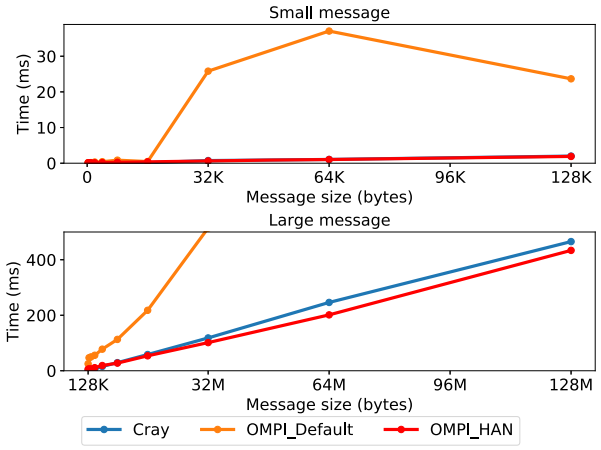


Fig. 13: MPI_Allreduce on Shaheen II, 4096 processes

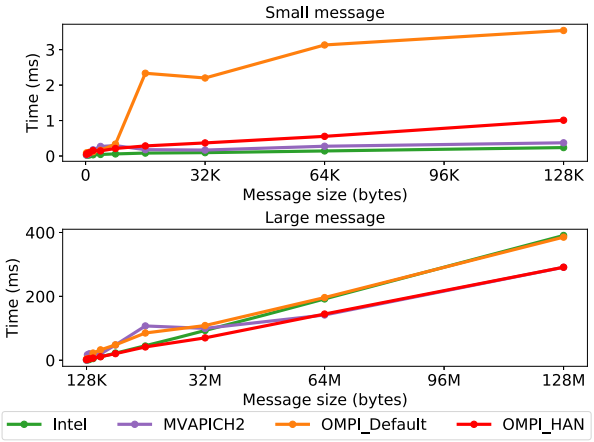


Fig. 14: MPI_Allreduce on Stampede2, 1536 processes

1536 processes on Stampede2, respectively. Compared with the default Open MPI, HAN shows significant improvements in all cases. Compared with other state-of-the-art MPIs, HAN exhibits some improvements with larger size messages:

- On Shaheen II, HAN shows better performance than Cray MPI after the message size is larger than 2MB and eventually achieves up to 1.12X speedup.
- On Stampede2, HAN is the fastest when message size is between 4MB and 64MB. Afterward, it delivers an similar performance as MVAPICH2, both significantly outperforming the others.

Besides the P2P performance discussed in the previous section, the cost of the reduction operations also impacts the performance of MPI_Allreduce. Among the four submodules currently used in HAN, only SOLO and ADAPT take advantage of the AVX instructions [39] to boost the performance of reduction operations. However, the designs of these two submodules [28] lead to high overhead on small messages. Hence, our autotuning component selects Libnbc and SM to perform MPI_Allreduce on small messages; unfortunately, neither of them supports AVX instruction, leading to lower performance compared to other MPIs. Preliminary studies have indicated that once the default Open MPI reduction operation are updated to support AVX, the HAN performance will benefit across all message sizes, overcoming the gap with the other implementations.

B. Application

We also evaluate HAN with two applications on Stampede2, each one focusing on a different type of collective operations.

1) ASP [40]: It solves the all-pairs-shortest-path problem with a parallel implementation of the Floyd-Warshall algorithm. Processes take turns to act as the root, and broadcast a row of the weight matrix to others, followed by computations, which causes MPI_Bcast to be the most time-consuming part of ASP. Table III presents the time of the first 1536 iterations in ASP on 1536 processes when the matrix size is 1M. We

TABLE III: ASP, 1536 processes on Stampede2, 1M Matrix

	Intel	MVAPICH2	Default OMPI	HAN
Comm (s)	10.44	24.12	38.52	8.99
Total (s)	20.78	34.81	47.11	19.37

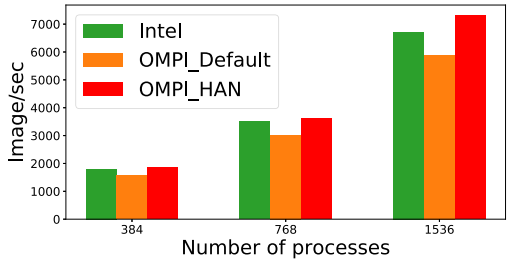


Fig. 15: Horovod on Stampede2

choose the first 1536 iterations to minimize the testing time but still cover all the possible cases by making sure each process acts as the root process once. HAN reduces the communication ratio from 50.24% (Intel MPI), 69.29% (MVAPICH2), 81.77% (default Open MPI) to 46.41%, and hence, achieves 1.08x, 1.8x and 2.43x overall speedup against Intel MPI, MVAPICH2 and default Open MPI, respectively.

2) Horovod [41]: It is a distributed training framework that uses MPI_Allreduce to average gradients. We use tf_cnn_benchmarks [42] with synthetic datasets to train AlexNet on Stampede2. Due to a configuration problem, we could only run Intel MPI 17.0.3, default Open MPI 4.0.0 and our framework. Figure 15 shows increasing gains for HAN as the number of processes increases, becoming 24.30% and 9.05% faster than default Open MPI and Intel MPI on 1536 processes, respectively.

V. Conclusion and Future Work

As a critical piece of the software infrastructure, MPI implementations need to adapt to the fast-changing HPC systems to reach users' efficiency expectations. In this paper, we present "HAN," a new hierarchical autotuned collective communication framework in Open MPI. The main contributions of this paper are twofold. First, the task-based design of HAN, which divides hierarchical collective communications into a set of tasks. With the task-based design, HAN can select suitable submodules on each level to utilize hardware capabilities, provide more opportunities to overlap communications, and minimize the effort to adapt to new hardware. Second, this design allows for a task-based autotuning component, supported by a novel cost model that is based on benchmarking the tasks. Our cost analysis indicates that our autotuning component significantly saves tuning time while maintaining high accuracy. Our experiments on two large scale HPC systems demonstrate HAN outperforms other state-of-the-art MPI implementations in most cases in both benchmarks and applications, providing a portable framework of highly efficient collective communication operations. In the future, we plan to further improve the submodules to boost the upper bound of the HAN framework and explore approaches based on an increased number of hardware levels. We also plan to add a new submodule to support intra-node GPU collective operations and combine it with the existing inter-node submodules to adapt HAN to GPU-based machines.

VI. Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and National Science Foundation under award EVLOVE #1664142. Experiments on the Shaheen II were supported by the Supercomputing Laboratory at KAUST, and experiments on the Stampede2 were supported by the Texas Advance Computing Center.

References

[1] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[2] M. Bayatpour, J. Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, and D. K. Panda, "Salar: Scalable and adaptive designs for large message reduction collectives," *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 12–23, 2018.

[3] J. L. Träff and A. Rougier, "Mpi collectives and datatypes for hierarchical all-to-all communication," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 27–32. [Online]. Available: https://doi.org/10.1145/2642769.2642770

[4] V. Tipparaju, J. Nieplocha, and D. Panda, "Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters," ser. IPDPS '03, 2003.

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

[6] P. Sanders, J. Speck, and J. L. Träff, "Two-tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan," *Parallel Comput.*, vol. 35, no. 12, pp. 581–594, Dec. 2009.

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

[8] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda, "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*, 2012.

[9] D. Agrawal and L. Bhuyan, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computers*, vol. 33, no. 04, pp. 323–333, apr 1984.

[10] H. Li and M. Maresca, "Polymorphic-torus network," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1345–1351, Sep. 1989.

[11] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, Oct 1985.

[12] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 77–88.

[13] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Nov 2000, pp. 3–3.

[14] A. Faraj and X. Yuan, "Automatic generation and tuning of mpi collective communication routines," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05, 2005.

[15] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel Comput.*, vol. 20, no. 3, Mar. 1994.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[17] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation," Tech. Rep., 1995.

[18] T. Kielmann, H. Bal, and K. Verstoep, "Fast measurement of logp parameters for message passing platforms," in *Parallel and Distributed Processing*. Springer Berlin Heidelberg, 2000, pp. 1176–1183.

[19] W. Gropp, L. N. Olson, and P. Samfass, "Modeling mpi communication performance on smp nodes: Is it time to retire the ping pong test," ser. EuroMPI 2016, 2016.

[20] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. Panda, "Scalable reduction collectives with data partitioning-based multi-leader design," ser. SC '17, 2017, pp. 64:1–64:11.

[21] P. Eller, T. Hoefler, and W. Gropp, "Using performance models to understand scalable krylov solver performance at scale for structured grid problems," ser. ICS '19, 2019.

[22] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang, "MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems," ser. PPoPP '99, 1999.

[23] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, *Hierarchical Collectives in MPICH2*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 325–326.

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

[25] B. Parsons and V. Pai, "Exploiting process imbalance to improve mpi collective operations in hierarchical systems," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015.

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

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

[28] X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, and J. Dongarra, "Adapt: An event-based adaptive collective communication framework," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18, 2018.

[29] G. Fagg, J. Pjesivac-grbovic, G. Bosilca, J. Dongarra, and E. Jeannot, "Flexible collective communication tuning architecture applied to open mpi," in *In 2006 Euro PVM/MPI*, 2006.

[30] A. Faraj, X. Yuan, and D. Lowenthal, "Star-mpi: Self tuned adaptive routines for mpi collective operations," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06, 2006.

[31] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*, Nov. 2007.

[32] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 13, pp. 1749–1783, Sep. 2007.

[33] *Intel MPI Benchmarks User Guide*, Sep 2018, https://software.intel.com/en-us/imb-user-guide.

[34] *OSU Micro-Benchmarks*, Mar 2019, http://mvapich.cse.ohio-state.edu/benchmarks/.

[35] J. Pješivac-Grbović, G. Bosilca, G. Fagg, T. Angskun, and J. Dongarra, "Mpi collective algorithm selection and quadtree encoding," *Parallel Comput.*, vol. 33, no. 9, Sep. 2007.

[36] J. Pješivac-Grbović, G. Bosilca, G. Fagg, T. Angskun, and J. Dongarra, "Decision trees and mpi collective algorithm selection problem," in *Euro-Par 2007 Parallel Processing*. Springer, 2007, pp. 107–117.

[37] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of mpi usage on a production supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 386–400.

[38] Q. Snell, A. Mikler, and J. Gustafson, "Netpipe: A network protocol independent performance evaluator," in *in IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[39] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," *Intel white paper*, vol. 19, no. 20, 2008.

[40] A. Plaat, H. Bal, and R. Hofman, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Gener. Comput. Syst.*, vol. 17, no. 6, Apr. 2001.

[41] A. Sergeev and M. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," 2018.

[42] *TensorFlow benchmarks*, Apr 2019, https://github.com/tensorflow/benchmarks.