SLOAVx: Scalable LOgarithmic AlltoallV Algorithm for Hierarchical Multicore Systems

Cong Xu[†] Manjunath Gorentla Venkata[‡] Richard L. Graham[‡] Yandong Wang[†] Zhuo Liu[†] Weikuan Yu[†]
Auburn University[†] Oak Ridge National Laboratory[‡]
{congxu, wangyd, zhuoliu, wkyu}@auburn.edu {manjugv, rlgraham}@ornl.gov

***Abstract*—Scientific applications use collective communication operations in Message Passing Interface (MPI) for global synchronization and data exchanges. Alltoall and AlltoallV are two important collective operations. They are used by MPI jobs to exchange messages among all of MPI processes. AlltoallV is a generalization of Alltoall, supporting messages of varying sizes. However, the existing MPI AlltoallV implementation has linear complexity, i.e., each process has to send messages to all other processes in the job. Such linear complexity can result in suboptimal scalability of MPI applications when they are deployed on millions of cores.**

To address above challenge, in this paper, we introduce a new *S*calable *LO*garithmic *A*lltoall*V* algorithm, named *SLOAV*, for MPI AlltoallV collective operation. SLOAV aims to achieve global exchange of small messages of different sizes in a logarithmic number of rounds. Furthermore, given the prevalence of multicore systems with shared memory, we design a hierarchical AlltoallV algorithm based on SLOAV by leveraging the advantages of shared memory, which is referred to as *SLOAVx*. Compared to SLOAV, SLOAVx significantly reduces the inter-node communication, thus improving the entire system performance and mitigating the impact of message latency. We have implemented and embedded both algorithms in Open MPI. Our evaluation on large-scale computer systems shows that for the 8-byte and 1024-process MPI_AlltoallV operation, the SLOAV can reduce the latency by as much as 86.4%, when compared to the state-of-the-art, and SLOAVx can further optimize the SLOAV by up to 83.1% in terms of message latency on multicore systems. In addition, experiments with NAS Parallel Benchmark (NPB) demonstrate that our algorithms are very effective for real-world applications.

***Keywords*-MPI, AlltoallV Algorithm, Collectives, Scalability**

I. Introduction

Existing scientific applications heavily rely on Message Passing Interface (MPI) to make the best use of the underlying systems. Among all the MPI functions, collective routines are the key factors that determine the scalability of the applications due to their global characteristics. Of particular importance are *Alltoall* and *AlltoallV* collective operations in MPI. They allow all the processes involved in the communication to exchange data with each other. Alltoall has a constraint that all the processes can only send data of the same size, which limits many applications, such as optimization of 3D Fast Fourier Transforms (3D-FFT) in CASTEP code [6] and quantum mechanical molecular dynamics simulation in CPMD [1], *etc*. AlltoallV instead supports all processes to exchange data of varying sizes.

Although much research work [5], [15], [16], [3] has been conducted on optimizing the Alltoall operation, little work has been carried out to examine the scalability of AlltoallV communication. *Bruck algorithm* [5] is a significant effort to improve the scalability for Alltoall communication. However, such strategy fails to support messages of varying sizes. In addition, so far, in all existing implementations of MPI AlltoallV, such as MVAPICH2, Open MPI, *etc*., the number of required messages to accomplish AlltoallV increases linearly with the number of processes involved in the communication. Such linear complexity severely limits the scalability of the large-scale scientific applications.

To address this critical issue, in this paper, we introduce a new *S*calable *LO*garithmic *A*lltoall*V* algorithm, called *SLOAV*. SLOAV aims to accomplish the AlltoallV collective communication for small messages of different sizes in a logarithmic number of rounds, thus significantly improving the performance of applications by reducing the number of messages for a global exchange of messages. To allow messages of various sizes, SLOAV provides applications with more flexibilities. We have systematically evaluated the efficiency of SLOAV algorithm on large-scale clusters, the results demonstrate that SLOAV can significantly outperform the state-of-the-art by up to 86.4%.

Multicore systems with shared memory are becoming ubiquitous in supercomputing infrastructure. Efficient usage of shared memory can dramatically mitigate the overhead of network message latency during the collective communication. Therefore, in this work, we further exploit the optimization spaces of SLOAV algorithm on multicore systems by leveraging the advantages of shared memory. Based on SLOAV, we introduce a new algorithm, called *SLOAVx*. SLOAVx is designed to orchestrate multi-layer AlltoallV communication. On each node, SLOAVx elects a group leader to manage data collection and transmission through using shared memory. To realize global exchange in AlltoallV, only group leaders conduct inter-node communication through SLOAV algorithm. The resulting scheme effectively cuts down on the number of messages over the network, thus reducing the impact of network message latency. We demonstrate experimentally that SLOAVx can further optimize the performance of SLOAV by 83.1% on multicore systems and provide near optimal scalability.

In summary, we make the following contributions on the

AlltoallV collective communication in this paper:

- We introduce a *S*calable *LO*garithmic *A*lltoall*V* (*SLOAV*) algorithm. It is designed, implemented and demonstrated to support AlltoallV in logarithmic complexity, significantly reducing the number of messages for a global data exchange in AlltoallV.
- We optimize the SLOAV algorithm on multicore systems and introduce the *SLOAVx* algorithm. Compared to the pure SLOAV algorithm, SLOAVx further reduces the impact of network message latency by leveraging shared memory.
- We have implemented both algorithms in Open MPI and conducted a systematic evaluation on large-scale supercomputing infrastructure. The experimental results demonstrate that SLOAV can outperform the existing implementations by up to 86.4% in terms of the message latency. SLOAVx can further outperform SLOAV on multicore system by as much as 83.1%.

In the rest of the paper, we discuss related work in section II. This is followed by a detailed description and analysis of SLOAV in section III. We then describe the SLOAV algorithm and introduce the SLOAVx algorithm in section IV. Experimental results are presented in section V and we conclude this paper in section VI.

II. RELATED WORK

Collective communication has been extensively researched in High-Performance Computing (HPC), however, very little research has been conducted for AlltoallV collective communications. In addition, none of the early work has studied AlltoallV communication with logarithmic complexity. Jackson and Booth [11] have proposed *Planned AlltoallV* to optimize AlltoallV in the clustered architecture. Their optimization collects data into one single message from all processes on the same node before conducting inter-node communication, so that the number of messages sent between different nodes can be dramatically reduced. Goglin *et al.* [3] proposed a kernel-assisted memory copy module (KNEM). It can bring benefits to collective intra-node AlltoallV communication. Later on, Ma *et al.* [12] optimized the KNEM by making use of memory architecture on NUMA architecture. However, the complexity of above algorithms and optimizations still require linear complexity to accomplish AlltoallV communication.

Brightwell and Underwood [4] carried out a deep analysis of the advantages of leveraging offload in MPI collectives to overlap the communication and computation and demonstrated the performance improvements for the NAS Parallel Benchmark [2]. However, the performance of MPI_AlltoallV was not improved. Faraj and Yuan [7] optimized MPI programs by leveraging *compiled communication*, taking advantage of the compiler's knowledge of network architecture and application communication requirements. The effectiveness was also been shown on NAS Parallel benchmark [2]. However, it was only effective for static communication with fixed patterns at compilation time. This work was unable to optimize dynamic communications, such as AlltoallV operation, since the compiler is unable to do array analysis. Plummer and Refson [14] optimized the MPI AlltoallV for materials science code CASTEP [6], their approach is to breakdown AlltoallV into multiple groups of processors and only require the group leader to participate in AlltoallV communication. However, the improvement is limited due to linear complexity and the organization of processes on each node can cause performance bottleneck.

Recursive Doubling (RDB) [16] and Bruck [5] algorithms are two logarithmic algorithms used in Alltoall communication to exchange small messages. When the process number is power of two, Bruck algorithm sends fewer amounts of data in comparison to RDB, and it works much better than RDB in realistic cases. However both of them cannot support messages of variant sizes.

III. SLOAV: A SCALABLE LOGARITHMIC ALLTOALLV ALGORITHM

Traditional MPI logarithmic collective algorithms, such as Bruck algorithm, can only work for Alltoall operation whose message sizes are uniform. In this section, we introduce our new *S*calable *LO*garithmic *A*lltoall*V* algorithm (SLOAV) for processing AlltoallV collective communication.

Our work is built on top of Cheetah [9], which is a collective communication framework embedded in Open MPI. In this framework, the MPI-level communication is controlled by a component in the Multi-Level (ML) manager, named COLL in Open MPI. Subgrouping (SBGP) component is used to extract topology information. At the communicator creation time, COLL takes advantage of SBGP to discover communication hierarchies and makes use of the subgroup information generated by SBGP to do collective communication within subgroups using the BCOL (Basic Collectives framework).

A. Drawbacks of Existing Solutions

Bruck algorithm emerged as a logarithmic algorithm to conduct Alltoall for small messages. In this algorithm, three imperative steps are carried out sequentially. Assume that the total number of processes that participate in the Alltoall communication is N and the rank of current process is n. In Step 1, the *Local Rotation Step*, message elements are rotated inside each process to prepare data for the next Step 2. Then in Step 2, inter-process communication is performed for $\lceil log_2 N \rceil$ rounds. At each round, data elements for the same destination in one process are merged before being sent out. In round s (starting from 1), process n sends data to process $(n + 2^{s-1}) \, mod \, N$ and process n receives data from process $(n - 2^{s-1} + N) \, mod \, N$. After $\lceil log_2 N \rceil$ rounds of communications, all the data elements can arrive at their final destination process. In Step 3, another intra-process rotation is proceeded to relocate the data elements to correct positions.

However, Bruck algorithm cannot deal with data elements of different sizes because the length of data input buffer used in this algorithm is fixed. We term the segment for holding an initial element in the data input buffer as an

element segment. Such algorithm cannot be directly applied to AlltoallV because, during the inter-process communication, the received intermediate data element can be much larger than the capacity of the element segment, thus making the communication inapplicable.

To address such issue, we introduce a logarithmic algorithm - SLOAV with new data structure support and a two-phase data transmission scheme for AlltoallV.

B. Scalable Logarithmic AlltoallV Algorithm (SLOAV)

In this section, we firstly describe the data structure for SLOAV and then introduce the two-phase message transmission technique. We then explain how data elements are rotated, and transmitted in the SLOAV algorithm.

1) New Data structure for SLOAV Algorithm: The new data structure support for SLOAV is shown in Fig. 1. Suppose there are 5 processes. For each process in the AlltoallV communication, there is an input data buffer that contains N data elements of various sizes ready to send to N processes respectively. The input data buffer is enough for a linear communication algorithm. To enable a logarithmic algorithm for AlltoallV, we add two new data structures: Element Index Table (EIT) and SLOAV buffer.

The EIT includes N entries, each of which is for a single data element. Every entry contains a length specifying the size of element and an offset pointing the location of the element in the input buffer or SLOAV buffer. Three benefits are obtained by using the EIT structure. First, each element is not required to be of the same length, and data can be placed anywhere, thus providing high flexibility. Second, it can significantly avoid memory copy overhead, because we only need to modify the length and pointer in the EIT without shifting actual data when a data movement is required. Compared to Bruck algorithm, which requires local memory rotation and data movement in both Step 1 and 3, EIT can effectively reduce the amount of memory copy. Thirdly, this structure is space-efficient and provides fast search speed.

SLOAV buffer is a temporary buffer to hold the intermediate elements that are too large to fit in its element segment. The SLOAV buffer is acquired from the Cheetah buffer pool (allocated to each process during MPI_INIT). Its size is adjustable, depending on the number of processes and average element size.

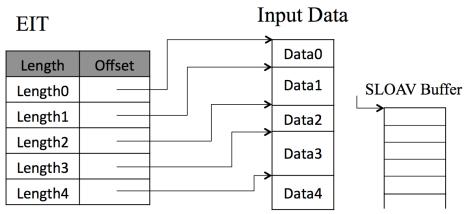


Fig. 1: Data structure for SLOAV

2) Two-phase message transmission: A logarithmic algorithm for AlltoallV operation requires data transmission to intermediate processes. But intermediate processes have no knowledge of the exact size of intermediate data they will receive. Each process only knows how much data it needs to send out and the length of each data element to receive. Precalculating the sizes of data elements at each round requires significant extra collective operations, which can cause extra communication and computation overhead. In order to address this, we propose an effective two-phase transmission approach.

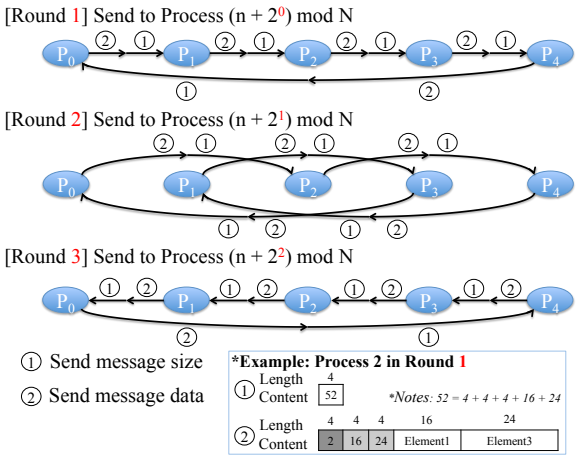


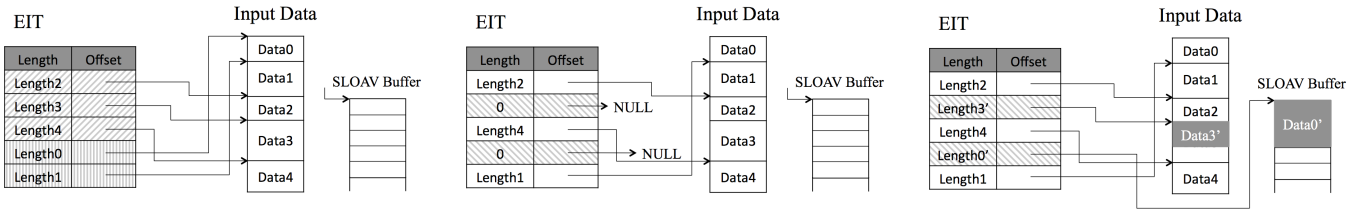
Fig. 2: SLOAV's Process to Process Communication

In this approach, a process combines the elements that have the same destination process id, then send the total message size and data to the targeted process. As shown in Fig. 2, three rounds of communications are required for an AlltoallV operation among five processes. In round s, process n firstly merges all of the elements e with $(e/2^{s-1}) \, mod \, 2$ equal to 1, then sends the size and data of the merged message to process $(n + 2^{s-1}) \, mod \, N$. Similarly, it receives the size and data of a combined message from process $(n - 2^{s-1} + N) \, mod \, N$. In the example, for round 1, process 2 merges its element1 (16B) and element3 (24B) as a message, next, it sends the size of the message as a new message (4B) to process 3, and then send the merged (52B) to process 3. The merged message contains the number of elements, the lengths of each element and the data. At the same time, process 2 receives the message size and data from process 1. Totally, the two-phase message transmission requires $2 * log_2 N$ start-up costs for each process but is still orders of magnitudes faster than linear algorithm which takes N start-up costs, especially for communication among large number of processes and small sizes of messages.

3) Element rotation, send and receive in SLOAV: Fig. 3 illustrates how one process rotates, sends and receives the message elements in SLOAV for AlltoallV communication among five processes.

Fig. 3(a) depicts the status of a process after it rotates by two positions. Note that for local data rotation in one process, we only need to rotate the entries in EIT without data moving.

As shown in Fig. 3(b), after some elements are sent out, the value of that element entry in the table is set to NULL. Before a process receives new data element, the former data element at the same position of input data buffer must be sent



(a) Local Rotation (b) Send Data to Another Process (c) Receive Data from Another Process

Fig. 3: Operations of each process in *SLOAV* algorithm

out as required by the algorithm to avoid conflicts.

In Fig. 3(c), when an element arrives, if the length of received element is smaller than the capacity of the element segment in the data input buffer, the received data element is placed into the data input buffer like data element Data3. Otherwise, that element is placed into SLOAV buffer such as Data0 and the offset of that entry is set to point to its location in the SLOAV buffer. We use a unified range for the offset value in the EIT, if the offset value i is smaller than the size of the input buffer S, it directly points to the position of i in the input data buffer. If not, it points to position of $i - S$ in the SLOAV buffer. This SLOAV buffer can be recycled if the occupying data element has been sent out later.

C. Theoretical Analysis of Communication Complexity

During the AlltoallV collective communication, the size of data sent by one process to the other is unknown to the receiver. Therefore, it is imperative for the sender to firstly notify the receiver about the size of message before conducting the data transportation. This educes the basic model for a process to accomplish data transfer in AlltoallV operations:

$$T_{SLOAV} = T_{size} + T_{msg} \tag{1}$$

where T_{size} is the time for the sender to notify the receiver about the size, and the T_{msg} is the time to send message data. From the Hockney model [10], assuming there's no contention in the network, the latency of sending data between a pair of end-points can be modeled as

$$T = S + \frac{c}{B} \tag{2}$$

, where S stands for the cost of start-up, B is the network bandwidth, and c the amount of data transferred between endpoints. Based on this point to point exchange formula in which the time required to perform local memory copy is ignored, the time of transmitting message size and data can be expressed by the following two equations:

$$T_{size} = \lceil log_2 N \rceil * (S + \frac{4}{B}) \tag{3}$$

$$T_{msg} = \lceil log_2 N \rceil * S + \frac{\sum_{i=0}^{\lceil log_2 N \rceil - 1} \vec{C}_k * \vec{M}_{\lceil log_2 N \rceil - 1 - i}}{B} \tag{4}$$

In the two formulas above, N is the total number of processes in the communication. $\lceil log_2 N \rceil$ rounds are required to complete process to process communication. A 4-byte *Int* variable is large enough to indicate the size of message. The time used in each round to send the message size is the sum of start-up time and the 4-byte integer tranmission time.

To calculate the message latency, besides the cost of start-up, the time of sending all of the data by each process in the network needs to be calculated. Equation (4) is the latency of adopting SLOAV to transfer data elements of various lengths. Vector $\vec{C}_k$ is a $1 \times N$ array, storing the length of process k's each data element that needs to be sent to N processes (including itself) in the current round. Thus $\vec{C}_k$ changes at every round. M is a $N \times \lceil log_2 N \rceil$ binary matrix, $\vec{M}_k$ is M's k-th column, and the combination of 1 or 0 at each row is the binary number of the corresponding row index. For example, when N is 5, the M is shown in Equation 6. M indicates which elements the sender needs to send at each round, starting from right to left.

$$\vec{C}_k = \begin{bmatrix} c_{k0} & c_{k1} & \cdots & c_{k(N-1)} \end{bmatrix} \tag{5}$$

$$M_{5*\lceil log_2 5 \rceil} = \begin{bmatrix} \vec{M}_0 & \vec{M}_1 & \vec{M}_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{6}$$

In contrast, the cost of linear AlltoallV algorithm is:

$$T_{linear} = N * S + \frac{\sum_{j=0}^{N-1} c_{kj}}{B} \tag{7}$$

As can be seen from the above formulas, when the average message size is small, the collective communication is dominated by the overhead of start-up operation (S). So the logarithmic algorithm can achieve better performance, which only requires $\lceil 2 * log_2 N \rceil$ times of start-up costs. However, when the message size becomes large, the network transmission time turns to be bottleneck. In addition, because in logarithmic AlltoallV, intermediate processes need to transfer or relay the messages, it can cause lots of duplicated messages sent/received in the network. For that reason, the performance of SLOAV is not ideal for dealing with large messages. If the maximum amount of data needs to be received in the AlltoallV operation is greater than a threshold, the program will switch to the large message algorithm.

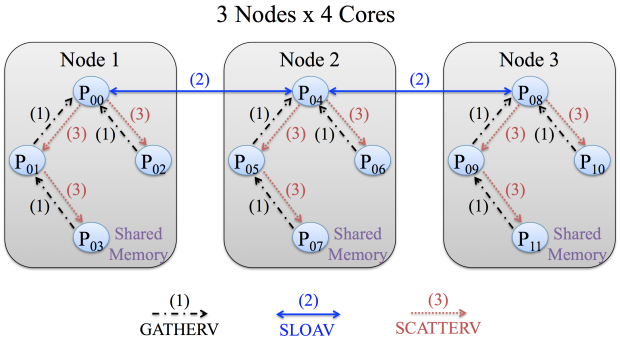


Fig. 4: *SLOAVx* Algorithm

IV. SLOAVx: Optimize SLOAV on Multicore Systems

We have described the *SLOAV* algorithm, which supports AlltoallV to realize a global message exchange in a logarithmic number of rounds. Although *SLOAV* reduces the complexity, the required number of messages to achieve AlltoallV is still a function of the number of processes involved in the communication. Nowadays, large-scale scientific applications launch hundreds of thousands of processes to fulfill task completion. At such scale, message latency can quickly prevents *SLOAV* from achieving the optimal performance. Therefore, in this section, we exploit the advantage of fast shared memory in multicore systems to optimize the *SLOAV* algorithm. We name such optimization as *SLOAVx*, which collects all messages from all processes on the same node before conducting inter-node communication. As a result, *SLOAVx* can efficiently cut down the number of messages exchanged between nodes, reducing it to be a function of the number of nodes in the cluster, thus dramatically mitigating the performance impact of network message latency.

Different from SLOAV algorithm which is unaware of the topology of processes, *SLOAVx* collects all messages from all local processors and delegates the group leader to perform inter-node communication via the *SLOAV* algorithm. Fig. 4 illustrates the high-level description of *SLOAVx* algorithm. On each node, one process is selected as the group leader. Such leader election can be achieved through using simple consensus algorithm. For the example in Fig. 4, processes P_{00}, P_{04}, and P_{08} are selected as group leaders for the processes on node 1, 2 and 3, respectively. After the leader is selected, all the rest processes on the same node are organized into a binomial tree, as shown in the Fig. 4. Such organization prevents the root process (group leader) from being the potential bottleneck and allows multiple processes to collect and forward messages simultaneously. Note that, under such tree structure, each process is aware of its parent and children processes based on the topology information provided by *SBGP* component in Cheetah framework.

Upon finishing the above setting up phase, on each node, *SLOAVx* continues AlltoallV communication by determining the offset for each process in the shared memory into which each process writes the message data. To calculate such offset, each process needs to notify its parent about the message size at the first place and then write the actual data. This step is shown in Step (1) in the Fig. 4. However, there is no existing MPI function support such functionality. Therefore, *SLOAVx* modifies *MPI_Gatherv* function to achieve such local message gathering in the shared memory.

Once all messages from all local processes have been collected, the group leader partitions the data according to the destination processes, and groups all the data going to the same compute node into the same partition. In our example, on each node, there are three such partitions, each one of which targets at a different compute node. After that, group leaders of the compute nodes apply the *SLOAV* algorithm to exchange the partitions, as shown in Step (2) in Fig. 4.

Upon receiving the data partitions for all the processes on the node, a group leader needs to transmit the data to each local process. Similar to *gatherv* in Step (1), such functionality is achieved by asking processes to read data from shared memory. Therefore, in Step (3), group leader notifies each process about the offsets from which each local process can read data from all the other processes in the cluster. Such notification goes through the tree structure instead of using broadcast. *SLOAVx* employs a modified *MPI_Scatterv* function to achieve this.

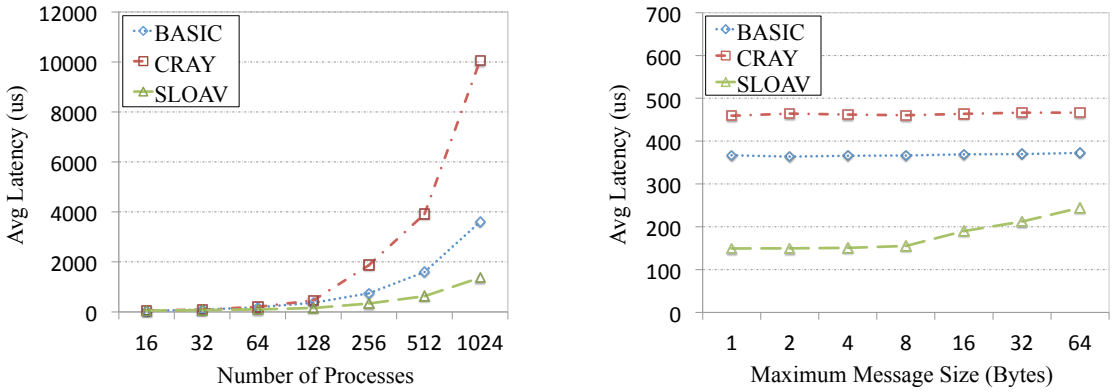
V. Experiment Evaluation

In this section, we conduct a systematic evaluation of our *SLOAV* and *SLOAVx* algorithms, which have been implemented into the *MPI_Alltoallv* in latest Open MPI.

A. Experiment Environment

1) System Configuration: All experiments are conducted on the two environments, which are the Jaguar supercomputer and the Smoky development cluster at the Oak Ridge National Lab. Jaguar features 18,688 physical compute nodes, each with a 16-core 2.2GHz AMD Opteron 6274 processor and 32GB memory. All compute nodes are connected through Cray's high-performance Gemini networking system. The Smoky cluster contains 80 Linux nodes, each with 4 quad-core 2.0GHz AMD Opteron processors, 32GB of memory (2GB per core), and a gigabit Ethernet network, along with Infiniband interconnect. Both Jaguar and Smoky share a center-wide Lustre-based file system, called Spider.

2) Benchmarks: To measure the efficiency of SLOAV and SLOAVx algorithms, we employ the widely-used OSU Micro-Benchmarks 3.7 [13] and the popular NAS Parallel Benchmark (NPB) [2]. The OSU benchmark, uses MPI_Alltoallv operations in a tight loop to warm up the caches, and then measure the performance of 10,000 MPI_Alltoallv operations. The final result reports the average latency. Regarding to the NAS benchmark, we adopt IS (Integer Sort) kernel to evaluate the performance of MPI_Alltoallv function, every test runs IS for 10 times and reports the average. Note that our algorithm mainly focuses on optimizing the small message exchange, and is not very suitable for large data. Therefore, in our experiments, we focus on the evaluation of small messages.



(a) AlltoallV with Maximum 8-bytes Small Message

(b) AlltoallV with Different Maximum Message Sizes

Fig. 5: Effectiveness of *SLOAV* Algorithm

B. Benefits of SLOAV Algorithm

We start our evaluation by measuring the latency of MPI_Alltoallv on the Jaguar supercomputer, and demonstrate the efficiency of SLOAV through two test cases. In the first test case, we increase the number of processes, while the message size varies from 1 byte to 8 bytes. In the second test case, we fix the number of processes (128) in the AlltoallV communication while varying the maximum size of message from 1 byte to 64 bytes. In both cases, we compare the results of SLOAV algorithm to those of the default MPI_Alltoallv function in Open MPI *(BASIC)* and MPI in Jaguar Cray system *(CRAY)*. We have also evaluated Tunned Open MPI [8], called *Tuned* MPI. However, across the tests, we observe that the performance of AlltoallV in Tuned MPI is very close to *BASIC* with 1%-3% difference. Therefore, in the following sections, we only report the results of BASIC for succinctness and clear presentation.

Fig. 5 (a) shows the results of first test case. On average, SLOAV algorithm significantly reduces the latency by up to 40.5% and 56.7%, when compared to the BASIC and the CRAY, respectively. More importantly, we observe that the improvement ratio increases proportionally to the number of processes involved in the AlltoallV communication, indicating superior scalability of SLOAV. For instance, when the number of processes increases to 1024, SLOAV outperforms BASIC and CRAY by as much as 62.3% and 86.4% respectively. These results adequately prove the efficient design of our SLOAV algorithm.

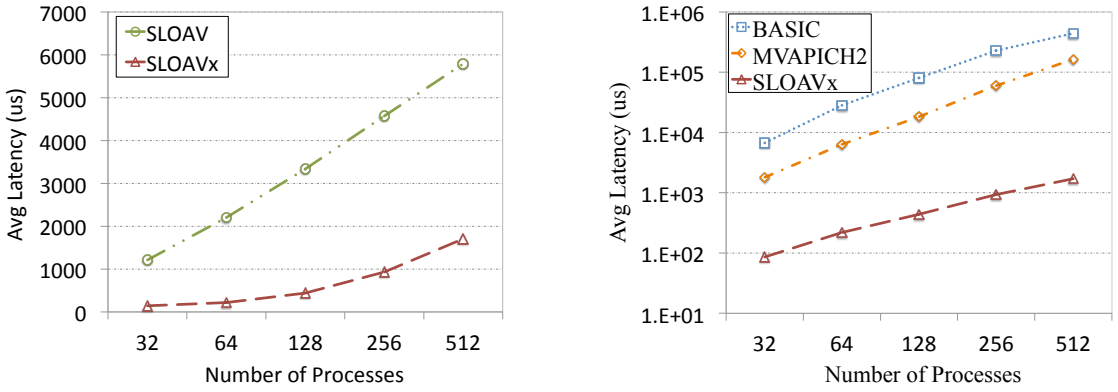
The results of second test case are shown in Fig. 5 (b), in which the x-axis represents the maximum message size sent by one process to the other. As shown in the figure, SLOAV performs 51.4% and 61.4% better on average than BASIC and CRAY, and achieves up to 59.3% and 67.5% latency reduction when the maximum size is 1 byte. However, we notice that SLOAV only maintains a constant latency until the maximum message size reaches 8 bytes, after which, when message size further increases, SLOAV shows degraded performance. This is mainly because that in log scale algorithm all of the participant processes (including source and intermediate processes) need to merge different data elements that has the same destination into one message before sending them out, this can cause large amount of memory copies. In addition, the SLOAV algorithm can generate lots of duplicated data elements in the network due to message relay through intermediate processes, leading to larger size of the merged message in each communication round.

C. Benefits of SLOAVx Algorithm

Recall from section IV that *SLOAVx* algorithm is designed to optimize the SLOAV in clusters equipped with multi-core system by leveraging the advantages of shared memory on each node. Through aggregating all the messages from local processes before conducting inter-node communication, SLOAVx aims to reduce the number of network send/receive operations, thus improving the latency. In this section, we evaluate the performance of SLOAVx on the Smoky cluster and compare the results to that of SLOAV and other alternative MPI solutions. All the experiments in this subsection run 16 processes on each node.

We firstly compare the SLOAVx with SLOAV to assess the effectiveness of optimization. Fig. 6 (a) shows the comparison results. In the experiment, we increase the number of processes from 32 to 512, meanwhile using 16 bytes as the maximum message size. While both SLOAV and SLOAVx achieve logarithmic scaling trends, SLOAVx reduces the latency by 83.1% on average. We also observe a consistent improvement across all the tests. The improvement brought by SLOAVx is twofold. First, when running multiple processes on each node, SLOAV invokes traditional send/receive functions to conduct point-to-point communication even though they may reside on the same node. This can quickly lead to severe resource contention when the number of cores on each node increases, resulting in degraded system performance. In contrast, SLOAVx leverages fast shared memory operations, such as *gatherv* and *scatherv*, thus reducing the overhead of going through the network stacks. Secondly, SLOAVx aggregates all the messages from local processes before conducting the inter-node communication. This efficiently reduces the amount of communication occur on the network.

We further compare the SLOAVx with another two MPI



(a) Comparison to SLOAV Algorithm

(b) Comparison to Other MPI_Alltoallv Solutions

Fig. 6: Effectiveness of *SLOAVx* Algorithm

alternatives, which are BASIC (described in section V-B) and MVAPICH2 MPI. As shown in Fig. 6 (b), SLOAVx significantly reduces the latency by 99.3% and 97.4% on average when compared to BASIC and MVAPICH2, respectively.

D. Sensitivity on the Number of Cores Used on Each Node

As described above, SLOAVx strives to make the best use of shared memory on multi-core system. During the experiments, we observe that SLOAVx performs very differently when the number of cores used on each node differs. Fig. 7 presents such phenomenon. In the experiments, we carry out 3 test cases. In each case, we fix the number of total process in the AlltoallV communication, but change the number of cores used on each node. As shown in the figure, the more cores used on each node, the better performance SLOAVx is able to achieve. This effectively corroborates that SLOAVx can efficiently leverage the shared memory on each node to reduce the network operations.

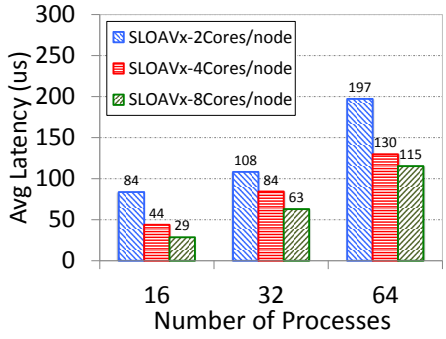
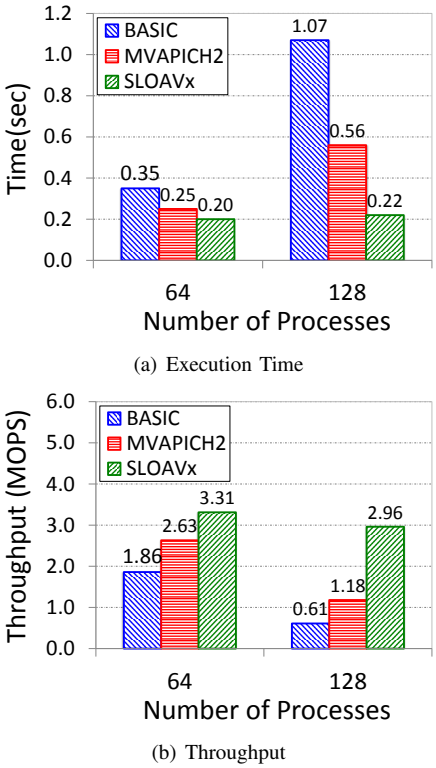


Fig. 7: Effectiveness of *SLOAVx* with Different Cores/Node

E. Evaluation with NAS Parallel Benchmark

To further investigate the efficiency of SLOAVx algorithm for real-world applications. We adopt IS parallel kernel in NAS Parallel Benchmark. The IS kernel is derived from computation fluid dynamics (CFD) which applies MPI_Alltoallv to conduct integer sorting using bucket sort. It assesses both the job execution time and computation throughput (measured by MOPS). In this subsection, we conduct our experiments on *S* class and *W* class workload in IS kernel.



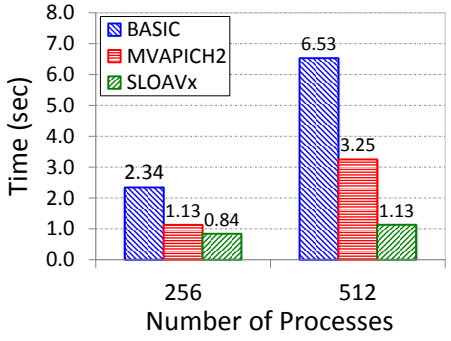
(a) Execution Time

(b) Throughput

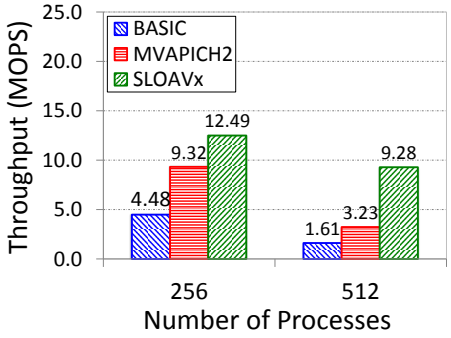
Fig. 8: Improvement on *S* Class

Fig. 8 shows the results of running *S* class workload, which only supports up to 128 processes and sorts 64KB data size. As shown in the Fig. 8(a), SLOAVx can always achieve superior performance improvement over BASIC and MVAPICH2 regardless of the number of processes involved. The improvement can be as much as 79.4% and 60.7%, compared to BASIC and MVAPICH2, respectively when the number of processes is 128. In addition, Fig. 8(b) shows that SLOAVx can not only reduce the job execution time but more importantly increase the entire system throughput as well, reaching as much as 385.2% and 150.8% MOPS improvement when compared to BASIC and MVAPICH2.

The results of running *W* class workload is shown in



(a) Execution Time



(b) Throughput

Fig. 9: Improvement on *W* Class

Fig. 9. It aims to sort 1MB data and can support very large number of processes. As shown in Fig. 9(a) and 9(b), SLOAVx outperforms the other two alternatives in terms of execution time and throughput when the number of processes increases, and achieving as much as 476.4% and 187.3% throughput improvement ratio compared to BASIC and MVAPICH2 when running under 512 processes. Overall, these experimental results adequately demonstrate that SLOAVx can efficiently reduce the job execution time and increase the throughput.

VI. Conclusion

The linear complexity in existing MPI AlltoallV operations severely hinders the scalability of scientific applications. In this study, we introduce a *S*calable *LO*garithmic *A*lltoall*V* algorithm, named as *SLOAV*, to reduce the complexity of AlltoallV collective communication for small messages of different sizes to logarithmic complexity. With such algorithm, the number of necessary messages for AlltoallV can be significantly reduced, thus the scalability of scientific applications can be effectively improved. Furthermore, to leverage the advantages of shared memory on multicore systems, we design the *SLOAVx* algorithm based on SLOAV. SLOAVx elects a group leader for all the processes on the same node and delegates the leaders on all the nodes to conduct inter-node communication via SLOAV. Both SLOAV and SLOAVx have been implemented and integrated into Open MPI. To assess their efficiency, we have systematically evaluated their performance on the Jaguar supercomputer and the Smoky cluster at Oak Ridge National Lab. Our experimental results demonstrate that SLOAV can significantly reduce the latency by up to 86.4%, when compared to the existing implementations. SLOAVx can further improve the performance of SLOAV on multicore systems by as much as 83.1%.

Acknowledgment

This work is funded in part by an Alabama Innovation Award to Weikuan Yu and a UT-Battelle award 4000087151. It is also enabled by an NSF award CNS-1059376 to Auburn University. This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] CPMD. http://cpmd.org/.

[2] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[3] Brice Goglin and Stephanie Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing (JPDC)*, 2012.

[4] R. Brightwell and K. D. Underwood. An analysis of the impact of mpi overlap and independent progress. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 298–305, New York, NY, USA, 2004. ACM.

[5] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, pages 298–309, New York, NY, USA, 1994. ACM.

[6] Castep Developers Group (CDG). Calculating the properties of materials from first principles. http://www.castep.org/, June 2012.

[7] A. Faraj and X. Yuan. Communication characteristics in the nas parallel benchmarks. In S. G. Akl and T. F. Gonzalez, editors, *International Conference on Parallel and Distributed Computing Systems, PDCS 2002, November 4-6, 2002, Cambridge, USA*, pages 724–729. IASTED/ACTA Press, 2002.

[8] A. Faraj and X. Yuan. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA, 2005. ACM.

[9] 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 *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 73–83, Washington, DC, USA, 2011. IEEE Computer Society.

[10] R. W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994.

[11] A. Jackson and S. Booth. Planned alltoallv. Technical report, EPCC (Edinburgh Parallel Computing Centre), July 2004.

[12] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra. Kernel assisted collective intra-node mpi communication among multi-core and many-core cpus. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 532–541, Washington, DC, USA, 2011. IEEE Computer Society.

[13] Ohio State University. Osu micro-benchmarks 3.7. http://mvapich.cse.ohio-state.edu/benchmarks/, September 2012.

[14] M. Plummer and K. Refson. An lpar-customized mpi_alltoallv for the materials science code castep. Technical report, EPCC (Edinburgh Parallel Computing Centre), July 2004.

[15] M. G. Venkata, R. L. Graham, J. Ladd, and P. Shamis. Exploring the all-to-all collective optimization space with connectx core-direct. In *ICPP*, pages 289–298, 2012.

[16] W. Yu, D. K. Panda, and D. Buntinas. Scalable, high-performance nic-based all-to-all broadcast over myrinet/gm. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 125–134, Washington, DC, USA, 2004. IEEE Computer Society.