Padding to Extend the Bruck Algorithm for Non-uniform All-to-all Communication

Ke Fan
kefan@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

Thomas Gilray
gilray@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

Sidharth Kumar
sid14@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

1 Introduction

`MPI_Alltoallv` is a Message Passing Interface (MPI) collective routine that facilitates non-uniform data exchange between every pair of processes. This routine is one of the most extensively utilized operations in MPI. However, it is difficult to scale and optimize due to its global and non-uniform nature. `MPI_Alltoallv` is implemented within popular MPI libraries using variants of the *SpreadOut* [1] algorithm that take P (the number of processes) communication steps. Such linear-complexity performs poorly when applications are deployed on millions of cores, especially for short-message communication which is dominated by latency [2].

Bruck's algorithm [2] is a classic *uniform* all-to-all algorithm that reduces the total number of communication steps from P to $log(P)$. This is achieved by transmitting an overall larger amount of data, but over a smaller number of iterations. The algorithm is therefore well suited for relatively smaller-sized data messages and has been adopted by popular MPI industrial libraries. However, *Bruck's* algorithm fails to support messages of varying sizes.

In this paper, we present the *Padded Bruck* algorithm, a natural extension strategy for applying *Bruck's* algorithm to *non-uniform* all-to-all communication. This algorithm converts the *non-uniform* communication pattern into a *uniform* one by padding data messages into equal-sized buffers, which is then followed by the *Bruck* algorithm. We also implemented two variants of the Bruck algorithm, with and without MPI datatypes, and evaluate their efficacy.

2 Bruck's Algorithm

Assume that each process has a send buffer (filled with data), which is logically comprised of P data-blocks. Each data block is made up of some number of data elements that are always transferred together. For *uniform* all-to-all, the number of elements is the same for all data-blocks, whereas for *non-uniform* all-to-all the total number of elements varies across the data blocks. Similarly, processes also have a receive buffer (initially empty). Both the send buffer and the receive buffer are contiguous 1-D arrays where all data-blocks are laid out in increasing block order. During all-to-all communication, each process i needs to send a data-block j ($0 \leqq j \leq P$) to and receive a data-block i from the corresponding process j.

Bruck's algorithm, in its original form, requires three phases: an initial rotation, $\log(P)$ communication steps, and a final rotation, as shown in Figure 1. The initial rotation phase performs a local copy and moves its data up by p (the rank of the process) data-blocks from the send buffer to the receive buffer. After that, the data-block to be sent to itself is at the top of the receive buffer, as shown in the first two sub-figures in Figure 1. In each communication step k, process p sends all data-blocks whose k_{th} bit is 1 to process $(p + 2^k)$ (with wrap around), receives from process $(p - 2^k)$, and overwrites the receive buffer with the incoming data. As in Figure 1, each process sends data-blocks with indices 1(0001), 3(0011) to the next process at communication step 0, and overwrites the sent data-blocks with the incoming ones. After the $\log(P)$ communication steps, all processes receive the necessary data, however not in the correct order. Hence a final rotation phase is performed to arrange them in the correct order, as seen in the last two sub-figures of Figure 1.

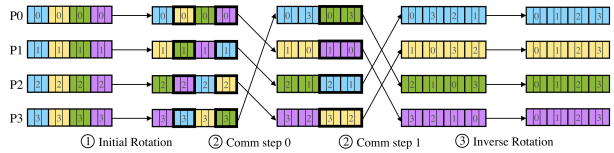


Figure 1. An example of Bruck's Algorithm.

However, *Bruck's* algorithm fails to support messages of varying sizes for two reasons. Firstly, the *Bruck's* algorithm needs every process to know exactly how many data-elements to expect during each of the intermediate $\log(P)$ steps. Furthermore, as *Bruck's* algorithm requires, received data-blocks must overwrite the sent data-blocks in each communication step as they will probably be sent again in future communication steps. However, the received data blocks may be too large to fit into the sent segments.

3 Padded Bruck Algorithm

The *Padded Bruck* algorithm is a natural extension strategy for applying *Bruck's* algorithm to *non-uniform* all-to-all problems by transforming them into *uniform* all-to-all problems. The *Padded Bruck* algorithm is implemented in three phases, as shown in Figure 2. Each process in the figure has four data-blocks, each of which is colored differently, and the number of data-elements in each data block varies. The first phase is to calculate the maximum data-block size across all $P \times P$ data-blocks (P data-blocks for every P process). To accomplish this, every process first finds its largest data-block size locally, and then uses `MPI_allreduce` to find the maximum overall block size (N) across all processes. After that, all processes pad all of P local data-blocks to the maximum size (N).

For example, the global maximum size of data-blocks in Figure 2 is 5 bytes, assuming each data element is one byte, and data-blocks are padded with grey elements. In the second step, Bruck's algorithm is applied to the padded buffers. As a result, all processes receive all the required data-blocks, as seen in the figure, where the data-blocks are all the same color. Finally, all processes perform a local scan (and filter) to retrieve the actual data from the padded buffer.

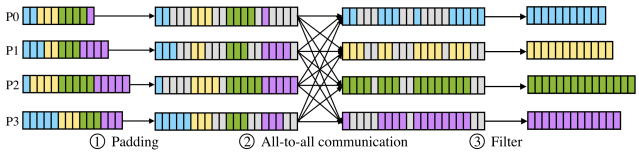


Figure 2. An example of the Padded Bruck Algorithm.

Although our *Padded Bruck* approach increases transferred data loads across communication steps, it has the same number of communication steps as *Bruck's* algorithm. It, therefore, does not increase the latency cost. Thus, our approach could potentially be effective for exchanging small data-blocks where the cost is dominated by latency.

Implementation approaches: The communication pattern of *Bruck's* algorithm is inherently non-contiguous [3]. Each communication step sends at most $P/2$ data-blocks to other processes. These blocks are in non-contiguous segments of memory. *Bruck's* algorithm, therefore, can be implemented with explicit buffer management or by using MPI-derived datatypes (`MPI_Type_create_indexed_block`). We made a thorough analysis of these two implementations and incorporated the one that performed the best into our *Padded Bruck* algorithm.

4 Evaluation

All our experiments are performed on the Theta supercomputer at the Argonne Leadership Computing Facility (ALCF). We ran each experiment 25 times and plotted the medians along with the median-absolute deviations (as error bars).

Figure 3 shows the systematic evaluation of two implementations of the *Bruck* algorithm. The *Bruck-dt* is implemented by using MPI-derived datatypes, while *Bruck* is done with *memcpy*. We fixed the length of the data-block to 32 bytes while varying the number of processes from 128 to 4096 processes. From the figure, we observe that implementation with an MPI-derived datatype performs poorly as compared to the one with *memcpy*.

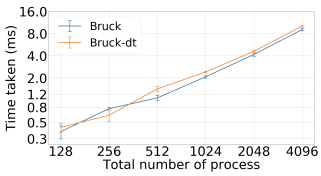
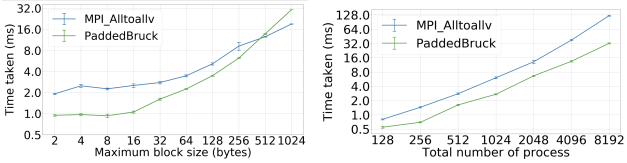


Figure 3. Running time of Bruck Implementations ($N = 32$).

With this *Bruck* implementation, we then report two sets of experiments: data scaling and weak scaling, as shown in Figure 4. In our experiments, every process generates data-blocks whose sizes follow the continuous uniform distribution. This distribution ensures that data-block sizes are uniformly sampled between 0 and maximum data-block size (N), thus yielding an average data-block of size $N/2$.



(a) Data-scaling ($P = 512$) **(b)** Weak-scaling ($N = 32$)

Figure 4. *Padded Bruck* vs. Cray MPI's `MPI_Alltoallv`.

Figure 4a depicts the outcome of a 512-process data-scaling experiment. The maximum data-block size (N) in this experiment ranged from 2 bytes to 1, 024 bytes. When the maximum data-block size (N) is smaller than 435 bytes, our *Padded Bruck* technique outperforms `MPI_Alltoallv`. For example, the *Padded Bruck* algorithm is 58.24% faster than `MPI_Alltoallv` at 16 bytes, and 32.52% faster at 256 bytes. The efficiency of our algorithm is further demonstrated by weak-scaling result shown in Figure 4b. In this experiment, the maximum data-block size is fixed to 32 bytes, and the number of processes is varied from 128 to 8, 192. In a weak-scaling experiment on all-to-all communication, the workload increased by 4× with the doubling of processes. Therefore, we observe an increase in execution time with increasing process counts. For $N = 32$, the total communication time of our *Padded Bruck* increases from 0.55 milliseconds at 128 processes to 32.19 milliseconds at 8, 192 processes. However, we also observe that for these data-block sizes, *Padded Bruck* outperforms `MPI_Alltoallv` at up to 8, 192 processes. For example, we observe a 73.95% improvement in performance at 8, 192 processes.

5 Conclusion and Future Work

In this paper, we demonstrate several instances where our *Padded Bruck* implementation outperforms the vendor-optimized `MPI_Alltoallv`. In the future, we will investigate the *two-phase Bruck* approach that extends *Bruck's* algorithm into *non-uniform* all-to-all communication by two phases: the metadata-exchange phase followed by the data-exchange phase at each communication step.

References

[1] Qiao Kang, Robert Ross, Robert Latham, Sunwoo Lee, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2020. Improving all-to-many personalized communication in two-phase I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[2] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[3] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. 2014. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*. 135–144.