Optimizing the Bruck Algorithm for Non-uniform All-to-all Communication

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

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

Valerio Pascucci
pascucci@sci.utah.edu
University of Utah
Salt Lake City, UT, USA

Xuan Huang
xuanhuang@sci.utah.edu
University of Utah
Salt Lake City, UT, USA

Kristopher Micinski
kkmicins@syr.edu
Syracuse University
Syracuse, NY, USA

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

ABSTRACT

In MPI, collective routines `MPI_Alltoall` and `MPI_Alltoallv` play an important role in facilitating all-to-all inter-process data exchange. `MPI_Alltoallv` is a generalization of `MPI_Alltoall`, supporting the exchange of *non-uniform* distributions of data. Popular implementations of MPI, such as MPICH and OpenMPI, implement `MPI_Alltoall` using a combination of techniques such as the Spread-out algorithm and the Bruck algorithm. Spread-out has a linear complexity in P, compared to Bruck's logarithmic complexity (P: process count); a selection between these two techniques is made at runtime based on the data block size. However, `MPI_Alltoallv` is typically implemented using only variants of the spread-out algorithm, and therefore misses out on the performance benefits that the log-time Bruck algorithm offers (especially for smaller data loads).

In this paper, we first implement and empirically evaluate all existing variants of the Bruck algorithm for uniform and non-uniform data loads– this forms the basis for our own Bruck-based non-uniform all-to-all algorithms. In particular, we developed two *open-source* implementations, *padded Bruck* and *two-phase Bruck*, that efficiently generalize Bruck algorithm to non-uniform all-to-all data exchange. We empirically validate the techniques on three supercomputers: Theta, Cori, and Stampede, using both microbenchmarks and two real-world applications: graph mining and program analysis. We perform weak and strong scaling studies for a range of average message sizes, degrees of imbalance, and distribution schemes, and demonstrate that our techniques outperform vendor-optimized Cray's `MPI_Alltoallv` by as much as 50% for some workloads and scales.

CCS CONCEPTS

• **Theory of computation** → **Massively parallel algorithms**; • **Computing methodologies** → *Massively parallel algorithms.*

KEYWORDS

MPI, Alltoallv, Bruck algorithm, Collective communication

ACM Reference Format:
Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the Bruck Algorithm for Non-uniform All-to-all Communication . In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27–30, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531468

1 INTRODUCTION

Motivation: Message passing [22] has been the dominant programming model in HPC for decades. There are three essential interfaces for data exchange among processes: point-to-point [19, 21], one-sided [11, 16], and collective communication [6, 32]. Point-to-point is the most granular data-exchange method, and it is performed using variants of `MPI_Send` and `MPI_Recv`. One-sided communication allows accessing the memory of a remote process by methods such as `MPI_Get` and `MPI_Put`. Collective functions involve communication among all processes within an MPI communicator. Due to their global nature, collective functions are the most difficult to scale and optimize. The overall scalability of applications [12, 27, 37] that rely on collective functions for their data exchange depends significantly on the scalability of the collective routines themselves.

`MPI_Alltoall` is a commonly used collective routine that facilitates data exchange between every pair of processes, allowing a process to send and receive a *fixed* amount of data from every other process. Our work focuses on optimizing non-uniform all-to-all communication, which is performed via `MPI_Alltoallv`, a more general version of `MPI_Alltoall` where each process may send and receive a *variable* amount of data. The state-of-the-art implementations of MPI, such as MPICH [2, 23] and openMPI [3, 15], implement `MPI_Alltoall` using a cocktail of techniques, including the Spread-out algorithm [26] and the Bruck algorithm [9]. Spread-out has a linear complexity in P compared to Bruck, which has a logarithmic complexity (P is the total number of processes). A selection between these two algorithms is made at runtime based on data block size. However, `MPI_Alltoallv` is implemented within these popular MPI implementations using only variants of the Spread-out algorithm [26]. In this work, we develop open-source implementations of variants of the Bruck algorithm for non-uniform all-to-all communication and demonstrate its efficacy for realistic microbenchmarks and applications. Our open-source implementation can be directly adopted by applications that rely on the usage of MPI_Alltoallv or by vendors that implement MPI.

Limitation of state-of-art approaches The performance of communication operations is a function of their latency and bandwidth costs. Latency is the fixed cost per communication step, which is independent of communication size, whereas bandwidth is the transfer time per byte [43]. Typically, short-message communication is dominated by latency, while long-message communication is dominated by bandwidth [38]. Short-message exchange, therefore, yields better performance with fewer underlying communication steps. The Bruck algorithm [9] is a well-known technique to reduce the total number of these internal communication steps in an all-to-all exchange 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. It is challenging to directly use the Bruck algorithm for messages of varying sizes as it requires processes to be aware of how much data to expect during each of the intermediate $\log(P)$ iterations [39].

Bruck's algorithm, in its original form, requires three phases: local data rotation, $\log(P)$ communication steps, and a final rotation (see Figure 1). Prior work [39] has modified Bruck to eliminate the final rotation phase by organizing data in a different order in the first two phases. This optimization has achieved incremental performance improvement over the traditional Bruck algorithm, but is also limited to uniform all-to-all data communication. Independently, the SLOAV algorithm [44] has explored techniques for extending Bruck's algorithm to allow non-uniform all-to-all data communication. It presented a coupled two-phase method that involved a meta-data-exchange phase followed by a data-exchange phase at each internal communication step. A key drawback of SLOAV is that it lacks an open-source implementation and has shown limited empirical evaluation (up to 1K processes) – a probable cause that has prevented it from getting adopted by popular MPI libraries. We have also identified some limitations of the SLOAV method from the paper: (a) an inefficient meta-data transmission scheme, (b) complex internal buffer management, (c) a redundant rotation phase, and (d) unnecessary final scan overhead (see Section 6.1).

Key insights and contributions In this paper, we survey and implement all variants of the Bruck algorithm for uniform data loads, to extract insights that can be used for developing techniques for non-uniform all-to-all algorithms. In particular, we implement the modified Bruck and zero-copy Bruck first presented in [39]. While the modified Bruck implementation gets rid of the final rotation phase, the zero-copy Bruck uses MPI-derived datatypes to eliminate explicit local memory copies. In addition to these two techniques, we also present our implementation of a uniform Bruck algorithm, *Zero Rotation Bruck*, which eliminates both the initial and final rotation phases. The implementation derives ideas from the modified Bruck to remove the final rotation and the usage of rotation index buffers (from SLOAV) to remove the initial rotation.

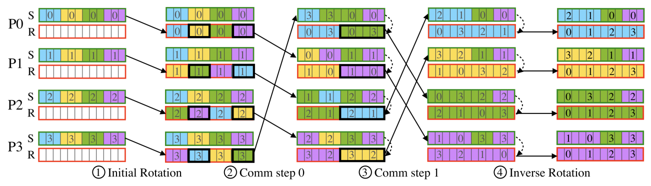
We present two alternative techniques for extending the Bruck algorithm to support *non-uniform* data distributions: *padded Bruck* and *two-phase Bruck*. Both these algorithms are built on top of the *Zero Rotation Bruck*. In padded Bruck, we convert the non-uniform communication pattern into a uniform one by padding data messages into equal-sized buffers, followed by the Bruck-style data exchange and a scan to filter out the actual data from its padding. The idea of *padded Bruck* was presented in [39] (section 3.5 of [39]). However, the paper neither provided any implementation details nor any empirical evaluations. Our other implementation, the *two-phase Bruck*, uses a meta-data exchange phase and a monolithic working buffer to facilitate non-uniform all-to-all data exchange. The meta-data exchange phase prepares for actual data transfer, and the monolithic buffer facilitates seamless movement of data during the communication phases. It synthesizes the techniques of SLOAV [44] (coupled two-phase IO, getting rid of the initial rotation phase) and Modified-Bruck [39] (getting rid of the final rotation phase), streamlining the algorithm and improving efficiency further with the use of a monolithic working buffer. Our two-phase Bruck technique also overcomes the shortcomings of the SLOAV approach with an efficient meta-data and buffer management scheme and by getting rid of all local rotation and scan phases. Additionally, we have performed a thorough empirical evaluation and have an open-source implementation. We make the following contributions:

(1) We survey and implement all variants of the Bruck algorithm (for uniform all-to-all) and present our implementation *Zero Rotation Bruck*, a synthesis of techniques that gets rid of rotation and scan overheads. (Section 2)
(2) We present open-source, reproducible implementations of two all-to-all communication algorithms for *non-uniform* data, suitable for small to moderate loads: *padded Bruck* and *two-phase Bruck*. (Section 3)
(3) We perform a detailed evaluation of our techniques using scaling (up to 32K processes) and sensitivity analysis using both microbenchmarks and practical applications on the three supercomputers. (Section 4 and 5)

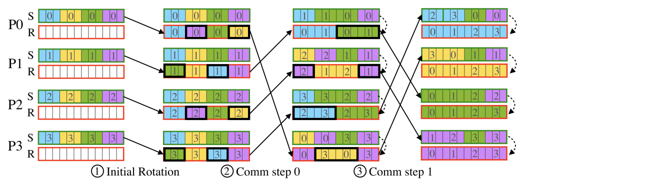
Compared to the vendor's MPI_Alltoallv implementation (Cray's proprietary MPI, based on MPICH [1]), our approach is up to 50% faster for some micro-benchmarks and up to 15% faster for more practical graph-mining and program-analysis applications.

Experimental methodology and artifact availability A key contribution of our work is an open-source implementation and a thorough evaluation of our techniques. Our work tackles an important problem that has the potential to improve a range of applications, and therefore, we have performed a rigorous evaluation of our algorithms. We have presented results for weak/data scaling up to 32K processes. Additionally, we performed a sensitivity analysis to study the subtle impact of distribution on performance. We used three techniques to create our distributions: (a) uniform, (b) Gaussian, and (c) power-law. Finally, we also created an empirical performance model to carve out the parameter space where our techniques perform better than the Spread-out algorithm. All experiments are performed on the Theta Supercomputer and for several iterations (mapping one MPI rank per core). To show the generality of our approach across platforms, we also show a subset of the results on the Cori and Stampede supercomputers (Section 7).

Limitations of the proposed approach Although we have performed thorough experiments on the Theta supercomputer and



(a) Basic Bruck.



(b) Modified Bruck.

Figure 1: Example of (a) Basic Bruck and (b) Modified Bruck with 4 processes (P0, P1, P2 and P3), each with Send (S) and Receive (R) buffers made of n (= 3) byte-sized P (= 4) data blocks. Both R and S are used during the comms steps. Note, the modified Bruck eliminates final rotation.

have conducted a small set of experiments on two other supercomputers, we believe that more work needs to be done to completely generalize the usability of our techniques. Experiments must be performed targeting multiple HPC systems, different workloads, and more applications. Ultimately, all experiment results must be used to develop a robust performance model. The model would enable vendor implementations of MPI to use both the Spread-out and the Bruck algorithms (two-phase and padded) for their implementations of `MPI_Alltoallv`. We have also not explored the applicability of our techniques for mixed datatypes, as used by `MPI_Alltoallw`. Finally, all our experiments were performed using the MPI-everywhere [45] programming model that maps a rank per core; in the future, we plan to explore techniques for a hybrid programming model [29].

2 IMPLEMENTATION OF UNIFORM BRUCK

The Bruck algorithm [38] is an efficient log-time implementation of all-to-all communication that is suitable for latency-bound short messages. Bruck's algorithm is comprised of three major passes: an initial data rotation, $\log(P)$ internal data transfer steps, and a final data rotation. In this section, we review existing research on variants of the Bruck algorithm in the context of uniform all-to-all communication. We compare the performance of these algorithms and study the underlying causes of their behavior to extract insights that can be used for non-uniform all-to-all.

2.1 Variants of the Bruck algorithm

With P processes, a uniform all-to-all can be expressed as follows. Every process has a *send buffer* (initialized with data), logically made out of P data-blocks ($S[0 \dots P-1]$), each with n 1-byte elements. Similarly, processes also have a *receive buffer* (initially empty), logically made out of P data-blocks ($R[0 \dots P-1]$) with n 1-byte elements. When implemented by `MPI_Alltoall`, both the send buffer and the receive buffer are contiguous 1-D arrays of size $P \times n$ bytes where all data-blocks $S[0 \dots P-1]$ and $R[0 \dots P-1]$ are laid out in increasing block order. During communication, every process with rank p ($0 \leqq p \leqq P-1$) transmits the data-block $S[i]$ ($0 \leqq i \leqq P-1$) to a process with rank i and receives a data-block from rank i into the data-block $R[i]$.

Basic Bruck [9] is a store-and-forward algorithm that takes $\log(P)$ steps for collective communication. The algorithm has three phases that are carried out in sequence:

(1) Local shift of data-blocks: $R[i] = S[(p+i)\%P]$. Each data-block (i.e., $R[i]$, $S[i]$) is a fixed-length buffer of n bytes.
(2) Global communication with $\log(P)$ steps. In each step k ($0 \leqq k < \log(P)$), process p sends to process $((p+2^k)\%P)$ all the data-blocks $R[i]$ whose kth bit of i is 1, and receives data from process $((p-2^k)\%P)$ into S, and replaces $R[i]$ (just sent) locally.
(3) Local inverse shift of data-blocks from R to R: $R[i] = R[(p-i)\%P]$.

Note that buffers S and R are both involved in the communication step because some received data-blocks will have to be resent in a later communication step. An example of basic Bruck with all three phases can be seen in Figure 1a.

Modified Bruck [39] improves upon the *Basic Bruck* algorithm by eliminating the final rotation phase. The send and receive processes are reversed and the initial rotation is modified to remove the final rotation required in the *Basic Bruck* (Figure 1b). It consists of the following phases:

(1) Local shift of data-blocks: $R[i] = S[(2*p-i)\%P]$.
(2) Global Communication with $\log(P)$ steps. In each step k ($0 \leqq k < \log(P)$), process p sends to process $((p-2^k)\%P)$ all the data-blocks $R[i+p]$ whose kth bit of i is 1, and receives data from process $((p+2^k)\%P)$ into S, and replaces $R[i+p]$ locally.

Zero-copy Bruck [39] avoids copying the received data-blocks from buffer S into R at the end of each communication step. This is achieved by creating a temporary buffer T and using it alternately with R to send and receive data (getting rid of the local copies (dashed-arrows) in Figure 1b). Similar to R, T (initially empty), is logically made up of P data-blocks ($T[0 \dots P-1]$) with n 1-byte elements. It has two phases:

(1) Local shift of data-blocks: $R[i] = S[(2*p-i)\%P]$.
(2) Global communication with $\log(P)$ steps. In each step k ($0 \leqq k < \log(P)$), for all the data-blocks whose kth bit of i is 1, we calculate the number b of non-zero bits with index at least k. If ($b\%2 == 0$), process p sends the block $T[i+p]$ to process $((p-2^k)\%P)$ and receives data in $R[i+p]$ from process $((p+2^k)\%P)$, otherwise, p sends the block $R[i+p]$ and receives data into $T[i+p]$.

Zero Rotation Bruck is our implementation, which is a synthesis of two techniques: (a) modified Bruck to eliminate final rotation, and (b) use of a rotation index array [44] to get rid of the initial rotation. We create a rotation index array, I, to store the desired order of data-blocks and use it to avoid actual shifting of data in the initial rotation phase. The cost of creating the rotation index array (I) is $O(P)$, which is less than the cost of local copies $O(Pn)$, and I can also be cached for repeated use. It has two phases:

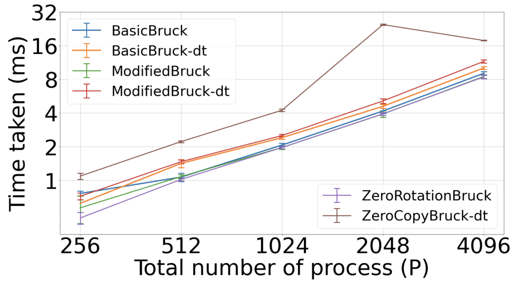
(1) Create local rotation array I: $I[i] = (2*p-i)\%P$

(2) Global communication with $\log(P)$ steps. In each step k ($0 \leqq k < \log(P)$), process p sends to process $(p - 2^k)\%P$ all the data-blocks $R[I[i+p]]$ whose k^{th} bit of i is 1, and receives data from process $((p + 2^k)\%P)$ into S, and replaces $R[I[i+p]]$ locally.

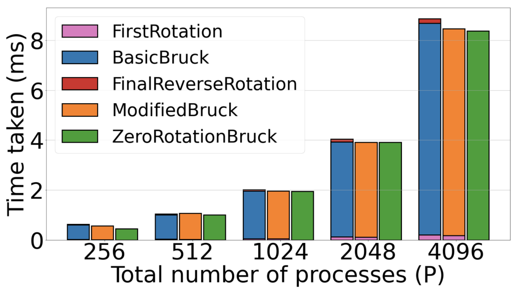
This implementation avoids both rotation phases and forms the basis for building our non-uniform Bruck algorithms.

2.2 Performance Comparison

The communication pattern of the Bruck algorithm and all its variants is inherently non-contiguous. Each communication step sends $(P+1)/2$ data-blocks to other processes (except the last step, which may send fewer blocks if the number of processes is not a power of 2). These blocks are in non-contiguous segments of memory. Variants of Bruck can be implemented with explicit buffer management (using memcpy) or by using MPI-derived datatypes [39]. MPI datatypes can implicitly pack and unpack non-contiguous blocks that need to be sent and received in each communication step. We therefore implemented two versions of the *Basic Bruck* algorithm and the *Modified Bruck* algorithm, with MPI-derived datatypes (*ModifiedBruck-dt* and *BasicBruck-dt*) and without them (*BasicBruck* and *ModifiedBruck*). Similar to [39], we implemented the Zero-copy Bruck algorithm using only MPI-derived datatypes (`MPI_Type_create_struct`), which we refer to as *ZeroCopyBruck-dt*. Our implementation, (*ZeroRotationBruck*) algorithm, is implemented with explicit memory management.



(a) Running time of all six variants.



(b) Breakdown time of non-MPI datatype variants.

Figure 2: Performance of Bruck variants ($N = 32$ bytes)

We evaluated all six variants of the Bruck algorithm using the Theta supercomputer [4]. All experiments were run for 20 iterations, and we plotted the median along with the median absolute deviation [24] as error bars in Figure 2a. We fixed the size of data-blocks to 32 bytes while varying the number of processes from 256 to 4,096. From the results, we observe three key trends: (a) *ZeroRotationBruck* consistently yields the best performance; (b) *ZeroCopyBruck-dt* is the least efficient; and (c) implementations with the MPI-derived datatype consistently perform poorly compared to the ones with explicit memory management (see red vs. green trendlines and orange vs. blue trendlines). For example, with $P = 256$ and $P = 4{,}096$, *ZeroRotationBruck* is 39.64% and 7.13% faster than *BasicBruck*, and 18.80% and 0.83% faster than *ModifiedBruck*.

ZeroRotationBruck reduces $O(Pn)$ memory-copying time of the first rotation phase, but incurs $O(P)$ penalty in populating the rotation array. Similar to *Modified Bruck*, it also eliminates the final local rotation phase (cost $O(Pn)$). As a result, when compared to *BasicBruck*, *ZeroRotationBruck* saves $O(2Pn - P)$ cost. This improvement can be seen in Figure 2b, which shows the performance breakdown of the three Bruck variants without MPI datatypes. The pink portion at the bottom is the first rotation phase, and the red portion at the top is the final rotation phase. The middle portion with different colors is for global communication. The communication time for these three variants is roughly the same since they are all implemented without an MPI datatype. From the figure (see bars at 4,096 processes), it is evident that *ZeroRotationBruck* (green bar) is the most efficient as it does not have any rotation phase (takes little time to create the rotation indexes array). We also note that the time percentages of the two rotation phases increase with the number of processes.

The *ZeroCopyBruck-dt* does not perform well when using a derived datatype to avoid copies at the end of each step. The overhead of using MPI-derived datatypes was also observed by Traff, et al. [39] where they found that *ZeroCopyBruck-dt* does not perform well if the data-block size is less than 250 bytes. In general, for all process counts, we found that MPI-derived datatype added an additional overhead which led to sub-optimal performance. We use these observations in designing our own optimized *non-uniform* all-to-all algorithm based on *ZeroRotationBruck*, which avoids using MPI-derived datatypes and eliminates both rotation phases.

3 IMPLEMENTATION OF NON-UNIFORM BRUCK

The Bruck algorithm in its existing form cannot be directly applied to *non-uniform* all-to-all communication for two reasons: (1) processes do not know the size of a data-block they will receive during each of $\log(P)$ communication steps in the Bruck algorithm; and (2) the receive buffer R or send buffer S cannot be reused as intermediate storage buffers during the communication steps since the intermediate received data can be larger than their capacity. To address these issues, we propose two advanced Bruck algorithms: padded Bruck (Section 3.1) and two-phased Bruck (Section 3.2).

Similar to uniform all-to-all, non-uniform all-to-all also has a *send buffer* (S) and a *receive buffer* (R), both of which are logically made up of P data-blocks. However, these data-blocks are of different sizes, and thus to differentiate them, we need explicit size arrays ($sendcounts[0 \ldots P-1]$ and $recvcounts[0 \ldots P-1]$) to store the size of data-blocks and offset arrays ($sdispls[0 \ldots P-1]$ and $rdispls[0 \ldots P-1]$) to store the starting positions of data-blocks.

3.1 Padded Bruck

Padded Bruck converts a *non-uniform* all-to-all problem into a *uniform* all-to-all problem through padding—a natural extension. There are three main phases: (a) padding all non-uniform buffers to a fixed-sized buffer, (b) invoking Bruck-style communication for the uniform buffers, and (c) scanning the received buffers to extract the actual data. The first step in the padding phase is to compute the size of the largest data-block (N) across all $P \times P$ data-blocks (P data-blocks for every P process). Every process first finds its largest data-block size locally, and then uses `MPI_Allreduce` to find the largest overall block size (N) across all processes. All processes locally pad all of P local data-blocks to the maximum size (N). After padding, the Bruck algorithm is used to perform uniform all-to-all data exchanges. Finally, all processes perform a local scan to extract the actual data from the padded buffer—this is accomplished using the *recvcounts* array, which holds the actual size of the data-blocks. For small-sized messages, the data-transfer time is dominated by latency, not bandwidth. The Bruck algorithm significantly reduces the latency from αP to $\alpha \log(P)$, where α is the fixed cost of initiating an internal communication. Although our approach increases the communication load, it does not increase the latency cost. The approach is therefore potentially effective for exchanging small data-blocks where the data exchange cost is dominated by latency. We also note that we derived the idea of padded Bruck from [39] (section 3.5), but the paper did not have any implementation details nor any empirical evaluation.

3.2 Two-phase Bruck

The *Two-phase Bruck* algorithm addresses the challenges of extending the Bruck algorithm for non-uniform data loads by performing a coupled two-phase data exchange (for all $\log(P)$ communication steps) and by using a large monolithic buffer. The two-phase communication involves a meta-data exchange followed by actual data transfer, where the meta-data prepares processes for the actual data-exchange. The monolithic working buffer facilitates seamless intermediate data exchanges, pre-allocated to an upper bound on overflow data. The approach requires more space in the transfer phases to optimize communication time.

The two-phase Bruck is built on top of *Zero Rotation Bruck algorithm. It indirectly* synthesizes techniques of the modified Bruck (removing the final rotation) and SLOAV (removing the initial rotation) and uses the idea of coupled metadata/data exchange first proposed in SLOAV. In addition to synthesis of all these different ideas, our work yields additional performance improvements by using a monolithic buffer that streamlines and simplifies both meta-data and data management.

Meta-data transmission. Each of the $\log(P)$ Bruck communication steps sends $(P+1)/2$ data-blocks from each process to another receiving process (except for the last step, which may send fewer blocks if the number of processes is not a power of 2). At each communication step, a process first sends meta-data containing data-block sizes—a buffer of size $(P+1)/2$ that holds the size of each data-block that needs to be transmitted. We demonstrate this step with an example in Figure 3, which shows the two ($\log_2 4$) communication steps for four processes. In each step, a process first exchanges the sizes of the two data-blocks it is sending. In the example shown, for the first step $k = 0$, every process sends data-blocks 1 and 3 to a specific send process. After meta-data exchange, every process knows how many bytes it will receive. A process,

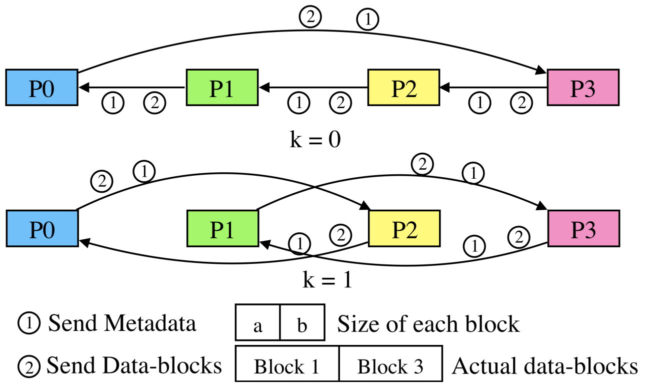


Figure 3: $\log_2(P = 4)$ comm steps with coupled meta-data and data exchanges in our two-phase Bruck algorithm.

Algorithm 1 Two Phase Non-Uniform Bruck Algorithm

```
Find maximum data-block length $N$ with MPI_Allreduce;
Allocate monolithic working buffer $W$ with length $(N * P)$;
for $i \in [0, P]$ do
    $I[i] = (2 * p - i)$ % $P$ // initiate rotation array;
end for
for step $k \in [0, \log(P)]$ do
    $n = 0$;
    for $i \in [2^k, P]$ whose $k^{th}$ bit is 1 do
        $sd[n{+}{+}] = (p + i)$ % $P$ // find $n$ send data-block indices;
    end for
    for $i \in [0, n]$ do
        $M[i] = sendcounts[I[sd[i]]]$ // prepare meta-data;
    end for
    $sendrank = (p - 2^k)$ % $P$;
    $recvrank = (p + 2^k)$ % $P$;
    Send $M$ to $sendrank$ and receive updated $M$ from $recvrank$;
    for $i \in [0, n]$ do
        if $status[i] == 1$ then
            Copy data for data-block $i$ from monolithic working buffer $W[sd[i] * N]$;
        else
            Copy data for data-block $i$ from send buffer $S[sdispls[I[sd[i]]]]$;
        end if
    end for
    Send these reorganized blocks to $sendrank$;
    for $i \in [0, n]$ do
        if $(sd[i] - p)\%P < 2^{k+1}$ then
            Receive data-block $i$ from $recvrank$ into $R[rdispls[sd[i]]]$;
        else
            Receive data-block $i$ from $recvrank$ into $W[sd[i] * N]$;
        end if
        $status[I[sd[i]]] = 1$;
        $sendcounts[I[sd[i]]] = M[i]$;
    end for
end for
```

therefore, can send and receive the actual data-blocks successfully in the subsequent data-transmission phase.

Data transmission. Each process transmits actual data after meta-data transmission. With the Bruck algorithm, received intermediate data-blocks are likely to be sent in future communication steps and are typically put back in the corresponding segments of the send buffer (basic Bruck and modified Bruck, see Figure 1b) or in the temporary buffer (Zero-copy Bruck). For uniform Bruck, this step is simple as the receive and send buffers are of the same size, never resulting in memory overflow. It is a challenge for *non-uniform* data communication where the sizes of received intermediate data-blocks can be larger than the sizes of the sent data-blocks (which must temporarily hold the received data-block). We address this challenge by allocating a large monolithic working buffer W to hold all received data-blocks. The size of W must be $P \times N$ to ensure sufficient space, where N is the global maximum size among all data-blocks. All received data-blocks which will be transferred in the future communication steps are stored in this buffer. We use a boolean variable to track if a data-block was exchanged in previous communication; if so, the block will be sent from the working buffer, otherwise, from the send buffer.

Algorithm. Pseudocode is shown in Algorithm 1. We first find the maximum size N among the data-blocks (line 1). This variable is used to allocate the monolithic working buffer W (line 2). We then initialize a rotation array I instead of shuffling the actual data (lines 3-5). In each communication step k, we compute the indices of the n data-blocks, sd, that need to be sent (lines 8-10) and then prepare the meta-data M (lines 11-13). Each process sends its meta-data to its particular receiving process (lines 14-16). With this information, each process then conducts the data transmission phase (lines 17-33). For the data transmission phase, we receive the data-blocks into the receive buffer R if they will not be sent again in the future communication steps, otherwise, into the monolithic working buffer $W[(i+p) * N]$, where i is data-block index and p is the rank of the current process (lines 24-33). We use a Boolean *status* array to track if a data-block has been exchanged before or not (lines 17-23) and update it at each communication step (line 31). A data-block will be drawn from the working buffer W if its *status* is 1, otherwise, data will be fetched from the original send buffer S.

Figure 4 depicts all of the $\log(P)$ communication steps in the two-phase Bruck with four processes. The first sub-figure shows the initial state, where we color the data-blocks and meta-data that will be transmitted in the first comm step. Note that the initial state also shows the locally copied data-blocks sent by a process to itself (i.e., send process rank = receive process rank). The next three sub-figures show all buffers (data and meta-data) after the two ($\log_2 4$) communication steps. The fourth sub-figure corresponds to the final state, where each process has received data-blocks from other processes in the correct order (omitting any rotation).

We further illustrate one coupled metadata and data exchanges for communication step $k = 0$ between process 0 and process 1 in Figure 5. The metadata and data exchanges are shown in yellow and blue, respectively. As in the *Zero Rotation Bruck*, process 1 finds out the indexes of data-blocks that need to send using the rotation index array (I). The sent data-blocks are $I[i+p]$, where i is 1, and 3 (as Bruck: k^{th} bit of block index is 1) and $p = 1$ (process with rank 1). Therefore, process 1 sends data-blocks $I[(1+1)\%4] = 0$ and $I[(3+1)\%4] = 2$ (shown by engraved black boxes). Once the indexes of sent data-blocks are computed, we first transmit the size of those blocks (using the *sendcounts* buffer (C)), as shown in yellow, followed by actual data (S), as shown in blue. A similar index computation is performed on process 0 to facilitate the receiving of data-blocks from process 1. We can note that the data received by process 0 is stored in both the receive buffer (R) and the working buffer (W). The working buffer data can be transmitted in the future communication steps.

3.3 Theoretical Performance model

We developed a simple model to estimate the cost of both algorithms in terms of latency and bandwidth. Assume that the overhead for exchanging a message between any two processes can be modeled as $\alpha + n\beta$, where α is the latency cost per communication exchange, independent of message size, β is the transfer time per bytes, and n is the number of bytes transferred. For all Bruck variants, over all $\log P$ communication steps, at most $\log P \times ((P+1)/2)$ blocks are sent and received per process. In padded Bruck, all data-blocks are the same size N. Each process therefore sends $\log P \times ((P+1)/2) \times N$ bytes to others at each round. Communication time of a process in padded Bruck is therefore:

$$\alpha \log P + \beta \log P \times ((P+1)/2) \times N \tag{1}$$

The *two-phase Bruck* algorithm has two transmission phases, which doubles the latency cost to $2\alpha \log P$. Meta-data sent during the first transmission phase contains $((P+1)/2)$ integers (4 bytes each). Actual data sent during the second transmission phase is $N/2 \times \log P \times ((P+1)/2)$, assuming data-blocks are distributed uniformly (see Section 4.1) and the average size of the data-block being transmitted is $N/2$. Communication time of a process in two-phase Bruck is therefore:

$$2\alpha \log P + 4\beta \log P \frac{(P+1)}{2} + \frac{N}{2}\beta \log P \frac{(P+1)}{2} \tag{2}$$

Comparing the time of padded Bruck with the two-phase Bruck, i.e. performing $(1) < (2)$, we obtain:

$$(N-8)(P+1)\beta < 4\alpha \tag{3}$$

Padded Bruck outperforms two-phase Bruck when inequality (3) holds true—this certainly happens when the N is less than 8 bytes. This is expected as padded Bruck transmits on average twice the amount of data compared to two-phase Bruck, and padded Bruck will only outperform two-phase Bruck when the amount of data transmitted is very small and the overall performance is bound by latency rather than bandwidth. This observation is also confirmed in our performance evaluation.

4 EVALUATION

We conduct a thorough evaluation of our algorithms using synthetic microbenchmarks (this Section) and applications (Section 5) on the Theta Supercomputer [4] of Argonne National Lab (ANL). Theta is a Cray machine with a peak performance of 11.69 petaflops, 281,088 compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM, and 10 PiB of disk storage. We compare the performance of our algorithms against vendor-optimized Cray MPI's `MPI_Alltoallv`, which is a proprietary, closed-source implementation from Cray based on the MPICH distribution [1]. We perform a series of experiments where we vary the maximum per-process data load (N), total number of processes (P), and data distribution types, resulting in four scaling studies: data scaling (Section 4.1), weak scaling (Section 4.1), sensitivity analysis (Section 4.2), and distribution study (Section 4.3). All our experiments were performed for a minimum of 20 iterations and used the MPI-everywhere programming model that maps one rank per core.

4.1 Scaling Analysis

In non-uniform all-to-all communication involving P processes, every process has P data-blocks of different sizes, where it transmits one data-block to every other process. In our experiments, every process generates data-blocks whose sizes follow the continuous

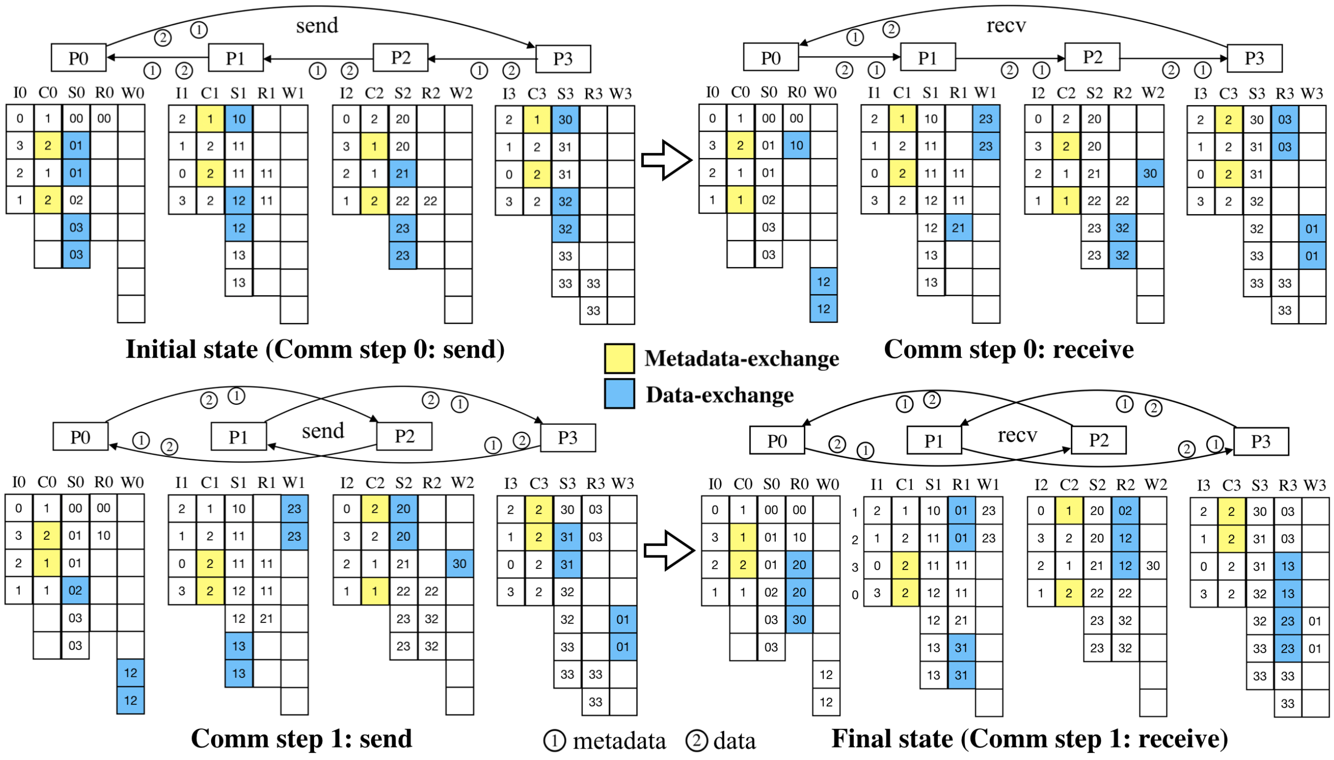


Figure 4: An example of two-phase Bruck with $P = 4$, showing $\log_2 4$ comm steps (I: *rotation index* array, C: *sendcounts* array, S: *send* buffer (data), R: *receive* buffer, W: *working* buffer.). Yellow shows the metadata and blue shows the data. The data-elements in the send buffer (S) follow the format: ij, where i is the local rank and j is the target process rank.

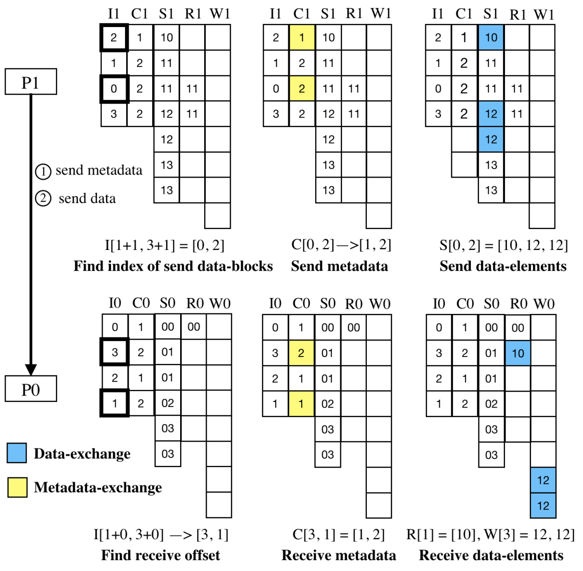


Figure 5: Sending and receiving data-blocks at communication step $k = 0$.

uniform distribution [5]. This distribution ensures that data-block sizes are randomly picked and uniformly sampled between 0 and the *maximum* data-block size (N), thus yielding an average data-block of size $N/2$. To further demonstrate the generality of our approach, we also conducted experiments for data-blocks whose sizes follow normal and power-law distributions (Section 4.3).

Data Scaling: We varied the maximum data-block size (N) from 16 to 2,048 bytes (generated using a double datatype), and process counts (P) from 128 to 32,768. In addition to our two advanced Bruck algorithms and Cray's `MPI_Alltoallv`, we also implemented the *Spread-out* and *PaddedAlltoall* algorithms. The *spread-out* algorithm uses the non-blocking point-to-point functions, `MPI_Isend` and `MPI_Irecv`. The *PaddedAlltoall* is similar to *padded Bruck*, where we apply Cray's `MPI_Alltoall` instead of our implementation of Bruck after padding. We plotted the timings in Figure 6. From these results, we made two key observations: (1) padded Bruck outperforms two-phase Bruck and others only for small data-block sizes and a narrow range of process counts; (2) two-phase Bruck consistently outperforms `MPI_Alltoallv` and others for all process counts (see red trend-line in all figures).

At most process counts, padded Bruck outperforms `MPI_Alltoallv` and two-phase Bruck for $N = 16$ bytes. For example, at 1,024 processes, padded Bruck is 28.7% faster than two-phase Bruck and 60.0% faster than `MPI_Alltoallv`. Its performance is also superior to that of the other two schemes for message sizes of up to 128 bytes for 128- and 256-process runs. However, its performance degrades rapidly for larger message sizes and higher process counts, also

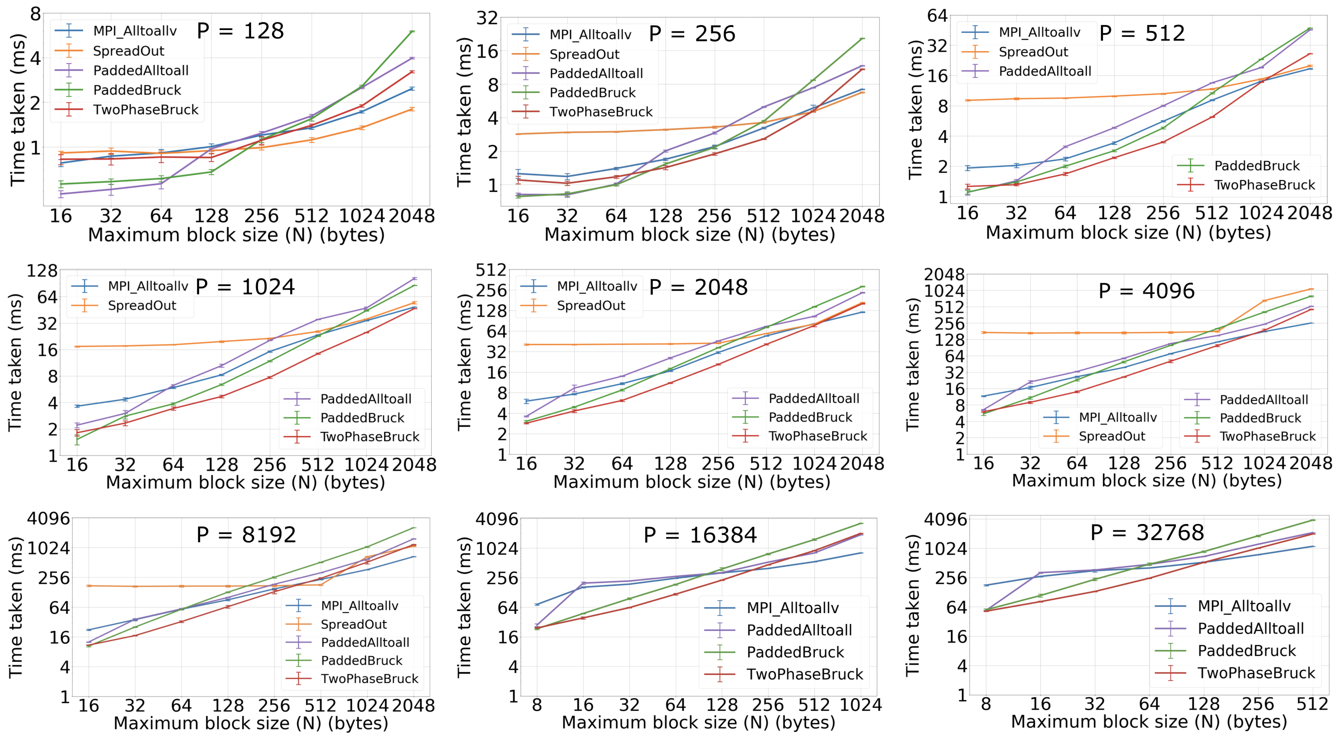


Figure 6: Data scaling (data block size (N) following the continuous uniform distribution)

in accordance with the performance model (see Section 3.3). For example, with $N = 512$ and $P = 4{,}096$, the padded Bruck takes 202.9 milliseconds compared to 91.6 milliseconds with two-phase Bruck. This trend is observed because padded Bruck transmits nearly twice the total amount of data compared to others, and therefore only works better for very small data loads or low process counts.

The two-phase Bruck outperforms `MPI_Alltoallv` and other schemes consistently for small to moderately sized data loads. From process counts of 256 to 4,096, two-phase Bruck outperforms `MPI_Alltoallv` for data-block sizes of up to 1,024 bytes. At $N = 256$, two-phase Bruck is 50.1%, 38.5%, 35.8% and 30.8% faster than `MPI_Alltoallv` at P of 512, 1,024, 2,048, and 4,096. The data-block size (N) range where the two-phase Bruck outperforms `MPI_Alltoallv` is reduced to 512, 256, and 128 bytes at process counts of 8,192, 16,384, and 32,768. In general, we observe that the data-block size range of two-phase Bruck reduces at higher process counts. This trend can be attributed to the nature of the algorithm, as with Bruck, the total amount of data transferred is $O(\log P)$ times more than the Spread-out algorithms. With the Bruck Algorithm, a process on average transmits $\log P \times (P+1)/2 \times N$ bytes, whereas `MPI_Alltoallv` only transmits $N \times P$ bytes. The performance of Bruck, therefore, starts to wane at higher core counts.

Weak Scaling: The efficiency of our algorithm is further demonstrated by the weak-scaling results shown in Figure 7. Given the inherent quadratic nature of all-to-all, we observe an increase in execution time with increasing process counts. For example, at $N = 64$, the communication time for two-phase Bruck increases from 0.47 ms at 128 processes to 34.2 ms at 8,192 processes. However, we observed that for $N = 64$, two-phase Bruck outperforms `MPI_Alltoallv` till 32,768 processes, and for $N = 512$, it outperforms `MPI_Alltoallv` till 8,192 processes. For example, for $N = 64$, we observe a 39.8% improvement in performance over `MPI_Alltoallv` at 8,192 processes. The scaling experiments clearly show a range of Ps and Ns where the two-phase Bruck outperforms the vendor-optimized `MPI_Alltoallv` implementation.

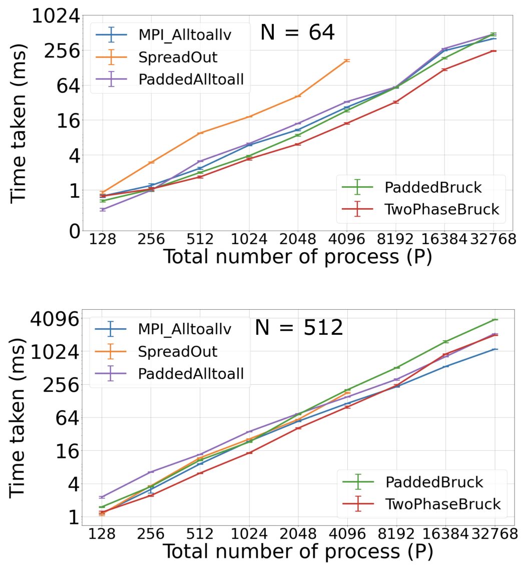


Figure 7: Weak scaling at $N = 64$ bytes (top) and $N = 512$ bytes (block sizes follow the continuous uniform distribution)

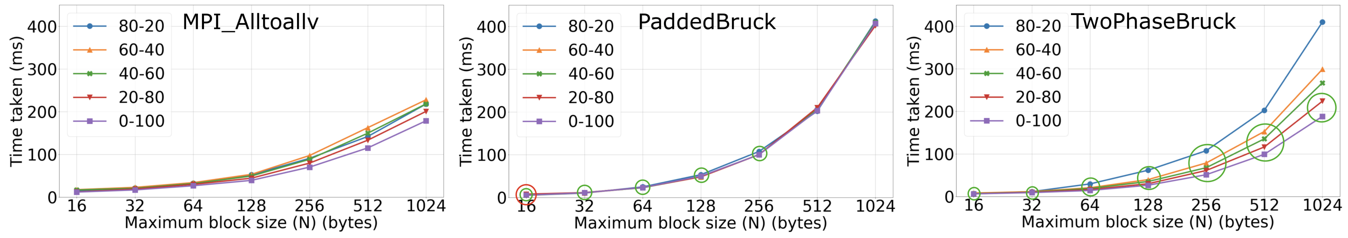


Figure 8: Sensitivity analysis at process count 4, 096.

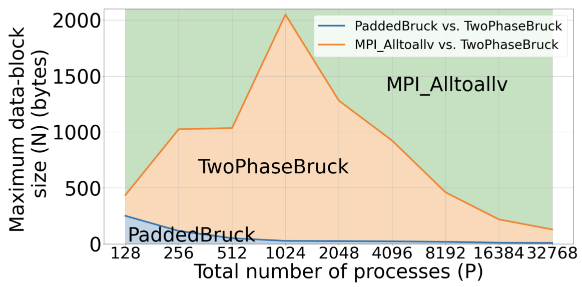


Figure 9: Empirical performance model.

Performance Model: We developed a simple, empirical performance model that carves out the range for N and P where two-phase Bruck outperforms `MPI_Alltoallv`. The model can be used to answer questions such as, with $P = 350$ and $N = 800$, should one use the two-phase Bruck, padded Bruck, or Cray's `MPI_Alltoallv`? We used data-scaling experiments in Figure 6 to identify, for each process count P, the data-block-size threshold N where two-phase Bruck ceases to perform better than `MPI_Alltoallv`. This can be found by identifying the intersection points between two-phase Bruck and `MPI_Alltoallv` in each of the 6 plots. We plotted these (N,P) pairs in Figure 9— carving out the parameter space (orange area) where two-phase Bruck performs the best. We also plotted the polyline to delineate our two approaches (two-phase Bruck and padded Bruck). As expected, we noticed two major trends: (a) the efficacy of our algorithm starts to decline at high process counts, and (b) the padded Bruck is effective only for small Ns and small Ps. Even with a high process count of 32,768, there are data-block sizes ($\leq$ 128) where our approach outperforms the vendor-optimized MPI implementation.

4.2 Sensitivity Analysis

We conducted a series of experiments to examine how our algorithm behaves with slight variations in data load. We used the continuous uniform distribution (from Section 4.1), but instead of varying the data-block sizes from 0 to N, we varied the sizes from $(100 - r)\%$ of N to N. For example, with $r = 50$, the distribution of data-block sizes would vary from $N/2$ to N. In Figure 8, we used the format $(100 - r) - r$ to indicate different variations. We varied the (N) from 16 bytes to 1,024 bytes and r from 0 to 80, and performed these experiments at process count ,4096. We circled the data points in green where two-phased Bruck outperforms `MPI_Alltoallv` and in red where padded Bruck outperforms two-phase Bruck.

We made two observations: (1) for a majority of configurations used, two-phase Bruck outperforms `MPI_alltoallv` and padded Bruck, and (2) the time taken for both `MPI_alltoallv` and two-phase Bruck reduces proportionally with data loads. Two-phase Bruck always out-performs others for $N \leq 512$ bytes, and for $N = 1024$ it shows the same performance as `MPI_alltoallv` for ranges $0 - 100$ and $20 - 80$. This can be attributed to the dominance of bandwidth costs associated with higher-r configurations, causing our algorithm to perform poorly compared to `MPI_Alltoallv`.

4.3 Standard distributions

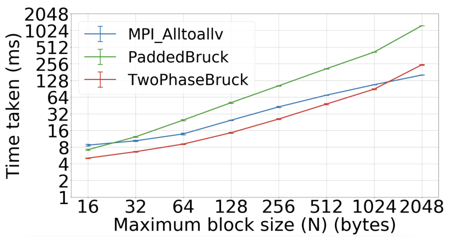
In addition to the applications we explore in Section 5, and uniform distribution in Section 4.1, we consider some standard distributions as well. In this section, we look at two power-law (exponential) and one normal (Gaussian) distributions.

Power-law: We generate P data-block sizes for every process using two power-law distributions (see Figure 10f). Figure 10(a-c) shows our results for P = 4,096 and P = 8,192. We observe three trends: (1) for both distributions, two-phase Bruck outperforms `MPI_Alltoallv` for all data-block sizes $N \leq$ 1,024, (2) padded Bruck performs poorly for all data-block sizes and process counts, and (3) the absolute timings for the power-law distribution with an exponent base of 0.99 are on an average less than those for the other distributions. For P = 8192, we observe an average speedup of 24.0% and 54.4% for the two distributions.

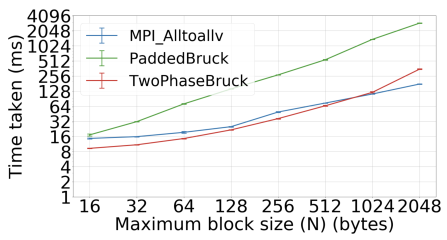
Normal: The Gaussian distribution is infinitely wide, so we use a window on this distribution, one that goes from -3σ to $+3\sigma$ (see Figure 10f). Figure 10(d-e) shows our results for P = 4,096 and P = 8,192. We observe a trend that differs from power-law distributions. With the normal distribution, the intersection point where `MPI_Alltoallv` outperforms two-phase Bruck is around N = 512, as opposed to N = 1,024 with the power-law. This can be attributed to the overall larger workload associated with the normal distribution. For example, the total data transmitted per process with the power-law at 4,096 process count is 203,928 bytes compared to 1,593,933 bytes with the normal distribution. With our scheme, we observe an average speedup of 33.9% at P = 8,192.

5 APPLICATIONS

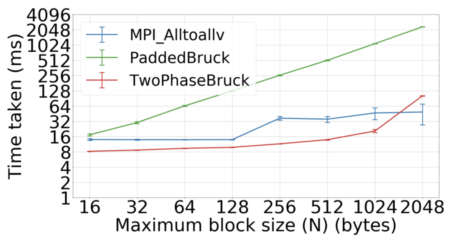
In this section, we apply and evaluate the two-phase Bruck using two core parallel algorithms that iterate non-uniform all-to-all communication in computing an output database (relation or set of relations). We implement these algorithms using an open-source implementation of balanced parallel relational algebra (BPRA), taken from previous literature, that maps database tuples to MPI processes in a dynamically balanced manner [13, 17, 27, 28]. For communication, we use a single all-to-all comm phase for all relational data



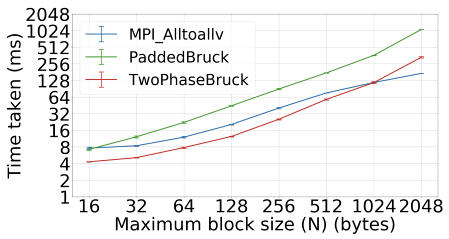
(a) P:4096, exp-base: 0.999



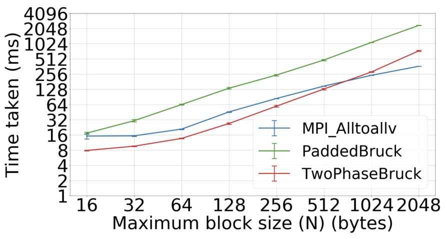
(b) P:8192, exp-base: 0.999



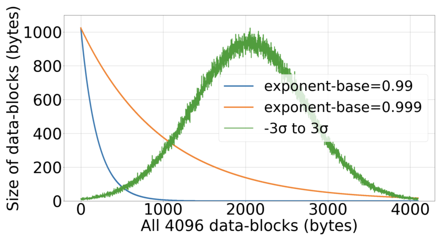
(c) P:8192, exp-base: 0.99



(d) P:4096, -3σ to $+3\sigma$

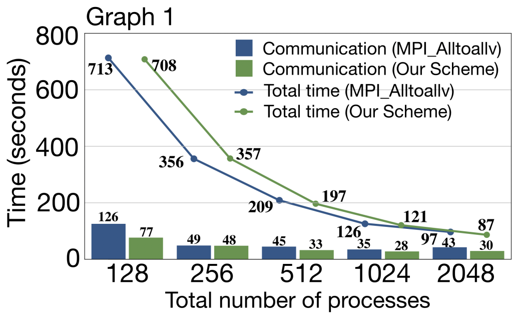


(e) P:8192, -3σ to $+3\sigma$



(f) Distributions

Figure 10: Results for power-law distributions and normal distributions.



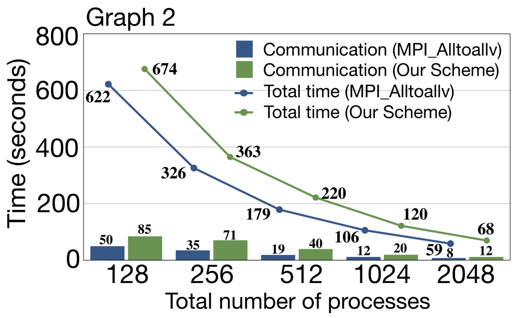


Figure 11: TC computation for *Graph 1* (top-row) and *Graph 2* (bottom-row) with `MPI_Alltoallv` and two-phase Bruck.

generated in parallel during a single round of parallel computation. Applications built on top of BPRA, such as the ones presented in the following subsections, make repeated all-to-all communication calls for 1000*s* of iterations with varying workloads.

5.1 Graph Mining: Transitive Closure

Graph-pattern mining (GPM) [30, 35, 42] provides a rich source of core problems that may be implemented using iterated all-to-all comm. A classic application from this domain computes the transitive closure (TC) [20, 31] of an input graph. Consider a relation $G \subseteq \mathbb{N}^2$ encoding a graph where each point $(a, b) \in G$ encodes the existence of an edge from vertex a to vertex b. Computing the transitive closure of a graph G is a classic fixed-point algorithm that may be implemented efficiently via MPI [28]. We can encapsulate a single round of path inference in a function F_G, which takes a lower-bound for the TC and improves it by finding additional paths. In the general case, for any finite graph G, there exists some $n \in \mathbb{N}$ such that ${F_G}^n(\bot)$ encodes the transitive closure of G. The TC of G may be computed by repeatedly applying F_G in a loop until reaching an n where ${F_G}^n(\bot) = {F_G}^{n-1}(\bot)$ in a process of *fixed-point iteration*. In this formulation, after a number of iterations equal to the length of the longest path in G, iteration of F_G will stabilize. In an MPI-based implementation, new paths found in every iteration must be transmitted to the appropriate process using all-to-all comm for the next iteration to continue. The number of paths created every iteration and the total number of iterations required to attain a fixed-point depend on the connectivity and topology of the graph. Therefore, depending on the graph, we can see varying workloads generated for all-to-all comm.

The MPI implementation for TC uses `MPI_Alltoallv` to perform all-to-all data exchanges. We were easily able to add the option to use our two-phase Bruck algorithm. This step was simple as our algorithm has the same function signature as `MPI_Alltoallv`. We performed strong scaling experiments and plotted our results in Figure 11. We use two graphs of different topologies and edge counts: *Graph 1* (412,148 edges) and *Graph 2* (1,014,951 edges), obtained from the Suite Sparse Matrix collection [10]. For *Graph 1*, we observe a nearly 10% (97 to 87 seconds) improvement in performance at 2,048 processes with two-phase Bruck over Cray's `MPI_Alltoallv`. We also observe that the rate of improvement in communication time increases with increasing process counts. This trend is expected with strong scaling, where the per-process data load decreases with increasing process counts, making it more suited for our Bruck-algorithm-based schemes. However, we observe a negative impact on performance with our algorithm for *Graph 2*. This diverging

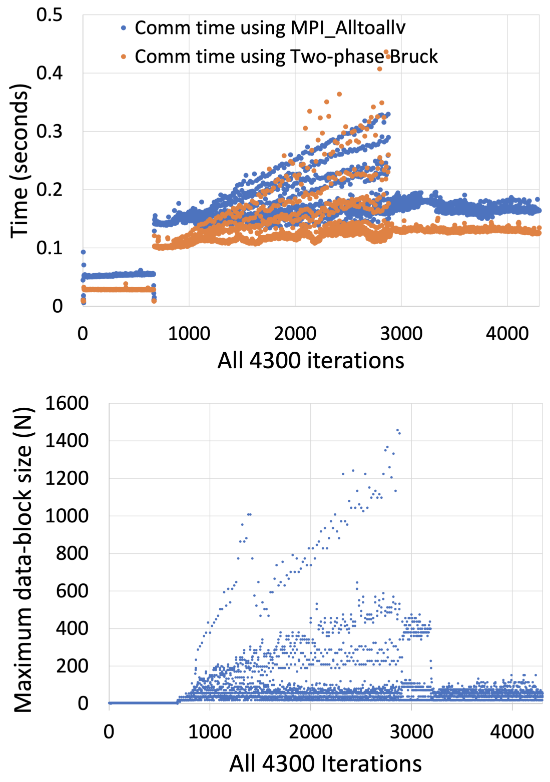


Figure 12: Comm time (top) and N (bottom) for all iterations for kCFA-8 (P = 4,096).

trend can be attributed to the difference in data load generated by the two graphs. *Graph 1* attained its fixed point after 2,933 iterations, generating a total of 1,676,697,415 edges, whereas *Graph 2* attained its fixed point after 89 iterations, generating a total of 508,931,041 edges. The number of paths generated per iteration in *Graph 2* is significantly (approx 10.0 times) higher than in *Graph 1*— also implies a significantly larger workload per iteration for all-to-all comm. The Bruck algorithm is only effective for smaller workloads, and therefore has a negative impact on performance for *Graph 2*.

5.2 Program Analysis: kCFA

Program analysis attempts to develop an accurate, bounded model of program behavior based only on the program's source text. Many practical models, as well as techniques for making them parallel, have been investigated in the literature [7, 17]. Program analyses are constructed using a variety of different theories and approaches, but they are usually fixed-point algorithms in much the same spirit as TC. Implemented via MPI, an all-to-all exchange propagates analysis facts, produced by a set of rules, to their managing process. One classic instance of static analysis is the *k-call-sensitive control-flow analysis* (kCFA), a well-ordered hierarchy of analyses with increasing precision and complexity as parameter k is increased. We use the input generator presented in prior literature on kCFA [40] to generate workloads for our experiments.

We perform a *kCFA-8* experiment at 4,096 processes, using both `MPI_Alltoallv` and the two-phased Bruck. The experiment took 4,300 iterations (involving 4,300 non-uniform all-to-all exchanges) to converge while generating a total of 1,286,254,830 facts. The total time taken by the experiment was 271 seconds using `MPI_Alltoallv` compared to 235 seconds with two-phase Bruck—translating to a 1.15× speedup. The all-to-all time came down from 74 seconds to 38 seconds. This overall improvement in performance can be understood by tracking the amount of time spent during all-to-all exchanges and by correlating that with the maximum data-block size (N) generated by processes during every iteration. We plot both communication time and maximum data-block size N in Figure 12. From the communication-time figures, we make two observations: (1) for a majority of iterations our scheme outperforms `MPI_Alltoallv` (majority of the orange points are below the corresponding blue points), and (2) the communication time varies significantly across iterations. The first trend can be understood by looking at the other set of graphs where N is plotted for all iterations. It can be seen that for a majority of iterations, N is smaller than 1000—a data-block size range where our algorithm outperforms `MPI_Alltoallv`. The second trend can be attributed to the inherent nature of the application where the underlying data load varies significantly across iterations.

6 RELATED WORK

The influential paper [38] presented a suite of optimizations to improve the performance of MPI collective functions. In particular, it introduced the Bruck's algorithm in the context of all-to-all data exchanges implemented with MPICH. Although much research work [9, 14, 39, 41] has been conducted on optimizing all-to-all operations for *uniform* data, little research has been conducted for all-to-all operations for *non-uniform* data loads. Sanjay et al. [34] proposed a two-stage algorithm that decomposes many-to-many communication with *non-uniform* messages into two all-to-all communications with uniform messages. They showed good performance when compared to a single-stage algorithm. However, their evaluation was limited to small process counts. Goglin et al. [18] proposed a hardware-independent kernel model (KNEM) that offers efficient intra-node MPI communication by allowing direct copying between processes. This work showed improvements in MPI collective operations by removing serialization at the root process. However, this work was designed for optimizing point-to-point send-receive instead of collective operations, and it is only effective for long-messages.

There is a body of research that focuses on reducing the number of processes involved in intra-node comm. Jackson et al. [25] presented a planned Alltoallv that transmits data from all processes on the same node to a chosen master process before conducting inter-node communication. This scheme reduces network congestion by restricting the number of processes participating in the all-to-all data exchange. They compared their work with standard `MPI_Alltoallv` and showed a significant improvement with short-message communication. However, the scheme is only effective for shared-memory clusters and is best suited for tasks requiring repeated executions with a fixed, *non-uniform* data load. On similar lines, Plummer et al. [33] introduced an algorithm that partitions all processes into non-overlapping communication groups, where only the leader process of a group participates in all-to-all communication. Within a group, processes use `MPI_Gatherv` and `MPI_Scatterv` to exchange data with the group leader process. These approaches are suited for applications with a *non-uniform* data load that is fixed across time.

6.1 SLOAV

SLOAV [44] was the first *logarithmic* running time algorithm proposed for non-uniform all-to-all data exchanges. It used a coupled two-phase communication method involving meta data exchange followed by data exchange. Although our two-phase Bruck algorithm is also based on this key idea, we identify some areas where we were able to improve on the original SLOAV algorithm [44].

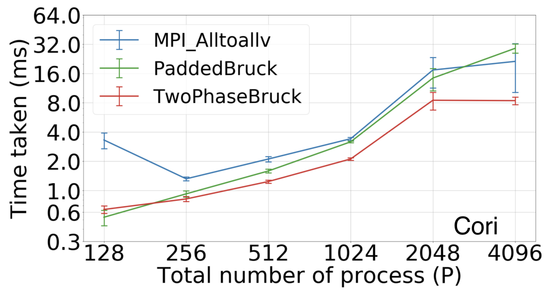
We present a detailed account of these improvements over SLOAV: *(1) Meta-data management:* during the intermediate $\log(P)$ communication steps, a process sends multiple data-blocks along with a block-size-array that stores the sizes of the data-blocks. SLOAV couples the meta-data (block-size-array) and the actual data (data-blocks) into one single combined buffer. The meta-data phase first transmits the size of the combined buffer, which is then followed by the data-transfer phase. This scheme potentially requires an extra *pack* step to combine the data-blocks and the block-size-array and also requires an additional *unpack* on the receiving side to separate the data from meta-data. Our approach instead transmits the block-size-array in the meta-data exchange phase – decoupling meta-data from data, and thus avoiding the computational cost associated with the *pack* and *unpack* steps. *(2) Buffer management:* SLOAV uses a two-layer data structure (size-array and pointer array) along with a temporary buffer. It requires explicit buffer management, resizing of the temporary buffer, and access to non-contiguous memory. Our approach uses a large monolithic buffer that is always sufficient to accommodate all intermediate data-blocks and therefore does not require an additional pointer array and any explicit memory copy. *(3) Rotation overhead:* SLOAV removes the initial rotation, but has an explicit final rotation phase. Our method, built on top of the *Zero Rotation Bruck*, eliminates both rotation phases. *(4) Scan overhead:* SLOAV has a final scan phase that copies all the data-blocks from send buffer and temporary buffer into the receive buffer. We get rid of this scan phase by preempting the final location and directly copying data into the output buffer.

Reproducibility and scaling: The empirical evaluation given for the SLOAV approach is limited in both scale and load. Although the paper compared the algorithm with `MPI_Alltoallv` and showed improvements for short-messages, it lacks key implementation details and is not open-sourced, making it difficult to reproduce. Our code is open-sourced [1] and reproducible, with a more rigorous evaluation (using synthetic benchmarks and real applications).

7 CONCLUSION

`MPI_Alltoallv` is a commonly used collective primitive that performs non-uniform all-to-all data exchanges. In this paper, we demonstrate several instances where our two-phase and padded Bruck outperforms the vendor-optimized version. Our techniques can potentially improve a range of applications that rely on non-uniform all-to-all comm. Our open-source code that can be easily adopted by applications and vendors will have the following implementations:

(1) Modified Bruck: A Bruck variant for uniform data loads (adapted from [39]) Same function signature as `MPI_Alltoall`.
(2) Zero Rotation Bruck: Our implementation of Bruck for uniform data loads gets rid of both rotation phases. (The same function signature as `MPI_Alltoall`)
(3) Padded Bruck: Our extension of Bruck for non-uniform all-to-all. (Same function signature as `MPI_Alltoallv`)
(4) two-phase Bruck: Our extension of Bruck for non-uniform data loads. (Same function signature as `MPI_Alltoallv`)



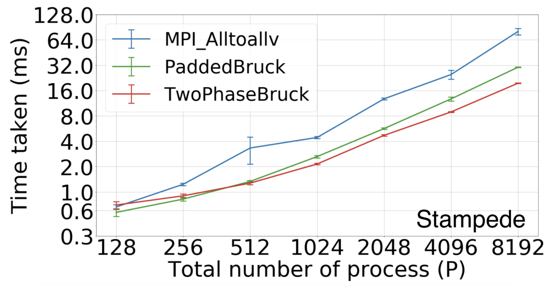


Figure 13: Weak scaling results comparing two-phase Bruck and padded bruck against vendor implementation of `MPI_Alltoallv` for Normal distribution data with $N = 64$ on Cori (top) and Stampede (bottom).

To further show the generalizability of our algorithm, we also conducted experiments on the Cori [8] (from ORNL) and Stampede2 [36] (from TACC) supercomputers. We show a subset of our results in Figure 13, where our algorithm outperforms the vendor-optimized implementation on both machines.

We believe that more work needs to be done in order to completely generalize the usability of the two-phase Bruck algorithm, targeting multiple HPC systems and different workloads—all leading to developing a more rigorous performance model. However, with the results of our paper showing substantial speedups, we can make a strong case for revisiting the Bruck Algorithm for *non-uniform* all-to-all communication. Implementations of MPI can use insights from this paper to directly optimize their `MPI_Alltoallv`.

8 ACKNOWLEDGEMENT

This work was funded in part by NSF RII Track-4 award 2132013. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the Theta Supercomputer located at the Argonne National Laboratory.

[1]https://github.com/harp-lab/bruck-alltoallv

REFERENCES

[1] MPI on Theta. MPI on Theta. https://www.alcf.anl.gov/support-center/theta/mpi-theta.

[2] MPICH Home Page. https://www.mpich.org.

[3] OpenMPI Home Page. https://www.open-mpi.org.

[4] Theta ALCF Home Page. https://www.alcf.anl.gov/theta.

[5] Uniform distribution. https://mathworld.wolfram.com/UniformDistribution.html

[6] George Almási, Philip Heidelberger, Charles J Archer, Xavier Martorell, C Chris Erway, José E Moreira, Burkhard Steinmacher-Burow, and Yili Zheng. 2005. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing*. 253–262.

[7] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 25–30.

[8] Katie Antypas, Nicholas Wright, Nicholas P Cardo, Allison Andrews, and Matthew Cordery. 2014. Cori: a cray xc pre-exascale system for nersc. *Cray User Group Proceedings. Cray* 1 (2014).

[9] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems* 8, 11 (1997), 1143–1156.

[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.

[11] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2016. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience* 28, 17 (2016), 4385–4404.

[12] Jun Doi and Yasushi Negishi. 2010. Overlapping methods of all-to-all communication and FFT algorithms for torus-connected massively parallel supercomputers. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–9.

[13] Ke Fan, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. 2021. Exploring MPI Collective I/O and File-per-process I/O for Checkpointing a Logical Inference Task. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 965–972.

[14] Ahmad Faraj and Xin Yuan. 2005. Automatic generation and tuning of MPI collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*. 393–402.

[15] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

[16] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Enabling highly-scalable remote memory access programming with MPI-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[17] Thomas Gilray and Sidharth Kumar. 2021. Compiling Data-parallel Datalog. In *International Conference on Compiler Construction*. IEEE.

[18] Brice Goglin and Stephanie Moreaud. 2013. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 176–188.

[19] Richard L Graham, Brian W Barrett, Galen M Shipman, Timothy S Woodall, and George Bosilca. 2007. Open mpi: A high performance, flexible implementation of mpi point-to-point communications. *Parallel Processing Letters* 17, 01 (2007), 79–88.

[20] Oded Green, Zhihui Du, Sanyamee Patel, Zehui Xie, Hang Liu, and David A Bader. 2021. Anti-Section Transitive Closure. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 192–201.

[21] William Gropp. 2002. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 7–7.

[22] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[23] William Gropp and Ewing Lusk. 1996. User's Guide for mpich, a Portable Implementation of MPI.

[24] David C Howell. 2005. Median absolute deviation. *Encyclopedia of statistics in behavioral science* (2005).

[25] Adrian Jackson and Stephen Booth. 2004. Planned AlltoAllv a Cluster Approach. (2004).

[26] 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 *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[27] Sidharth Kumar and Thomas Gilray. 2019. Distributed Relational Algebra at Scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE.

[28] Sidharth Kumar and Thomas Gilray. 2020. Load-balancing Parallel Relational Algebra. In *ISC High Performance*. IEEE.

[29] Ewing Lusk and Anthony Chan. 2008. Early experiments with the OpenMP/MPI hybrid programming model. In *International Workshop on OpenMP*. Springer, 36–47.

[30] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 56–65.

[31] Sarthak Patel, Bhrugu Dave, Smit Kumbhani, Mihir Desai, Sidharth Kumar, and Bhaskar Chaudhury. 2021. Scalable parallel algorithm for fast computation of Transitive Closure of Graphs on Shared Memory Architectures. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 1–9.

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

[33] Martin Plummer and Keith Refson. 2004. An lpar-customized mpi alltoallv for the materials science code castep. *Technical Report, EPCC (Edinburgh Parallel Computing Centre)* (2004).

[34] Sanjay Ranka, Ravi V Shankar, and Khaled A Alsabti. 1995. Many-to-many personalized communication with bounded traffic. In *Proceedings Frontiers' 95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 20–27.

[35] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1906–1917.

[36] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, Susan Mehringer, Eric Wernert, H Tufo, D Panda, et al. 2017. Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. 1–8.

[37] Hari Sundar, Dhairya Malhotra, and George Biros. 2013. Hyksort: a new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th international ACM conference on international conference on supercomputing*. 293–302.

[38] 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.

[39] 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*.

[40] David Van Horn and Harry G Mairson. 2008. Deciding k CFA is complete for EXPTIME. *ACM Sigplan Notices* 43, 9 (2008), 275–282.

[41] Manjunath Gorentla Venkata, Richard L Graham, Joshua Ladd, and Pavel Shamis. 2012. Exploring the all-to-all collective optimization space with connectx core-direct. In *2012 41st International Conference on Parallel Processing*. IEEE.

[42] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 763–782.

[43] Thomas Worsch, Ralf Reussner, and Werner Augustin. 2002. On benchmarking collective MPI operations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 271–279.

[44] Cong Xu, Manjunath Gorentla Venkata, Richard L Graham, Yandong Wang, Zhuo Liu, and Weikuan Yu. 2013. Sloavx: Scalable logarithmic alltoallv algorithm for hierarchical multicore systems. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 369–376.

[45] Rohit Zambre, Aparna Chandramowlishwaran, and Pavan Balaji. 2020. How i learned to stop worrying about user-visible endpoints and love MPI. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–13.