A Visual Guide to MPI All-to-all

Nick Netterville*, Ke Fan*, Sidharth Kumar*, Thomas Gilray*
* University of Alabama at Birmingham, Birmingham, AL, USA

***Abstract*—The standard implementation of `MPI_Alltoall` in MPI libraries (e.g., MPICH, Open-MPI) uses a combination of techniques, such as the spread-out and Bruck algorithms. The spread-out algorithm uses a linear number iterations, in process count P, while the Bruck algorithm is logarithmic. The Bruck algorithm transfers more data overall, but with fewer communication steps, and is thus better suited for smaller sized (latency-dominated) messages. MPI implementations dynamically choose the underlying algorithm to use depending upon process count and message size.**

We have created an easy-to-use, parameterized, interactive web-based visualization that shows the implementation details of both the linear-step spread-out algorithm and the log-step Bruck algorithm, along with the decision tree used to choose between these two algorithms. Our tool visually illustrates and animates the two algorithms, pointing out key differences such as number of iterations, communication pattern and whether they are in-place.

I. Introduction

MPI collectives perform an important set of global communication tasks such as broadcast, gather, and reduction, and are an integral component of a variety of HPC applications [1]–[5]. `MPI_Alltoall` is a commonly used collective routine that facilitates all-to-all inter-process data exchange, allowing a process to send and receive a fixed amount of data from every other process. The state-of-the-art implementations of MPI, such as MPICH [6] and OpenMPI [7], implement `MPI_Alltoall` using a combination of algorithms, including the Spread-out algorithm [8] and the Bruck algorithm [9]. Both these techniques are implemented using an internal sequence of point-to-point communication. The spread-out algorithm exchanges $P-1$ (P: total number of processes) data-blocks with other processes in $P-1$ steps, with each step exchanging one data-block. On the other hand, the Bruck algorithm transfers data blocks in $\log(P)$ steps, with each step exchanging n (where $1 \leq n < P$) data blocks [10]. In an all-to-all communication, P of these linear spread-out transfers take place in parallel. Therefore, the spread-out algorithm has a linear time complexity in P while the Bruck algorithm has a logarithmic time complexity in P.

The Hockney performance model [11], assesses the minimum communication cost of collective operations in terms of both latency (the required number of communication steps) and bandwidth (the actual data transfer time). In `MPI_Alltoall`, the Bruck algorithm transfers more data with fewer communication steps than the spread-out algorithm. As a result, the Bruck algorithm works well for short messages (latency-dominated), whereas the spread-out algorithm performs well for larger messages (bandwidth-dominated) [12]. The standard `MPI_Alltoall` implementation in major libraries selects between these algorithms at runtime depending on a pre-defined configuration file. This file defines a decision tree that specifies the appropriate algorithm for any given process counts and message sizes. MPI libraries typically offer a default configuration file based on heuristic numbers. Although this file can be customized, most users are unaware of it when using the `MPI_Alltoall` function.

To help users understand the actual implementation of `MPI_Alltoall` fully and quickly, and in a more intuitive way, we developed a web-based visualization system to demonstrate the implementation details of both spread-out algorithm and Bruck algorithms, and how to select between them based on the decision tree. The dashboard of our visualization system is an area plot of the decision tree that reveals which algorithm is used for the given number of processes and message size. The implementation details of any algorithm can then be viewed by clicking on the matching legend. To fully grasp each algorithm, our system allows users to explore it step by step or play all the steps at once, with the exchanged data emphasized by animations. Overall, our system is helpful and useful for learning the implementation of `MPI_Alltoall`, which can save users a considerable amount of time. The visualization system can be used in a pedagogical environment to introduce collectives to new HPC users. We make the following contributions to the literature:

1) We developed an interactive parameterized visualization to demonstrate the functioning of the Bruck and spread-out algorithms.
2) We use this visualization to illustrate key differences between the two algorithms.
3) We have open-sourced and hosted our visualization at https://naexris.github.io/mpi-vis/.

II. All-to-all implementations

`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. With P processes, `MPI_Alltoall` can be expressed as follows (see Fig. 1). Every process has a *send buffer* (initialized with data), logically made out of P data-blocks ($S[0 \ldots P-1]$), each with n 1-byte elements. Similarly, processes also have a *receive buffer* (initially empty), logically made out of P data-blocks ($R[0 \ldots P-1]$) with n 1-byte elements. Both the send buffer and the receive buffer are contiguous 1-D arrays of size $P \times n$ bytes where all data-blocks $S[0 \ldots P-1]$ and $R[0 \ldots P-1]$ are laid out in increasing block order. During communication, every process

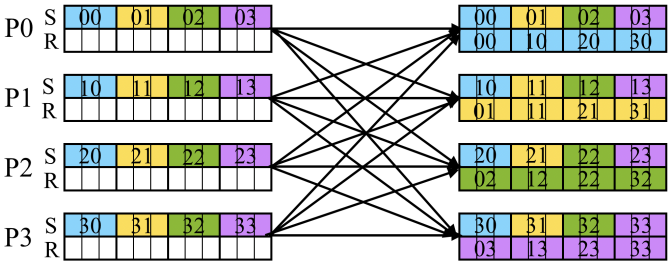


Fig. 1: Example of all-to-all communication with 4 processes, each with Send (S) and Receive (R) buffers made of P ($= 4$) data blocks differentiated by colors. Each data-block has n ($= 3$) data-elements, split by black lines. The index of each data-block is xy (x: original rank, y: target rank).

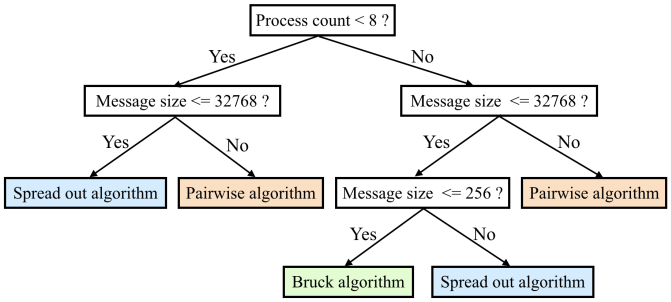


Fig. 2: The decision tree in the default configuration file of the MPICH library selects the appropriate algorithm for a given process count and message size.

with rank p $(0 \leq p \leq P-1)$ 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 $R[i]$.

Popular MPI libraries, such as MPICH and OpenMPI, implement `MPI_Alltoall` using a cocktail of algorithms, such as pairwise, spread-out and Bruck. These algorithms are classified into linear (P) algorithms and logarithmic ($log(P)$) algorithms (P : process count). A selection between these these algorithms is made at runtime depending on the pre-defined configuration file, which defines which algorithm is utilized for the given process count and message size. For example, Fig. 2 depicts the default decision tree that is defined in the configuration file of MPICH. This configuration file can be customized based on the running machine.

As stated previously, 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 message size, whereas bandwidth is the transfer time per byte, which relies on the message size directly. Therefore, the required number of communication steps and the total amount of data transferred are two important metrics to assess the time complexity of communication algorithms. In terms of P, the Bruck algorithm requires logarithmic steps while the spread-out algorithm requires linear steps. Other algorithms such as modified Bruck [13] and pair-wise exchange [12] used to implement all-to-all are mostly variants of these two core algorithms. Our visualization therefore, focuses on these two core algorithms.

A. Bruck algorithm

The Bruck algorithm is an efficient log-step implementation of all-to-all communication that is suitable for latency-bound short messages. In its original form, requires three phases: an initial data rotation phase, a communication phase that contains log_2^P steps, and a final data rotation phase (seen Fig. 3). 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 S to the receive buffer R. After that, the data-block to be sent to itself is at the top of the receive buffer R, as shown in the first two sub-figures in Fig. 3. In

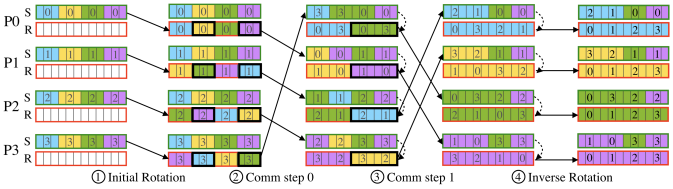


Fig. 3: Example of the Bruck algorithm with 4 processes (P0, P1, P2 and P3). Both R and S are used during the $log_2^4 = 2$ communication steps.

each communication step k, process p sends all data-blocks whose k_{th} bit of binary represented index is 1 to process $(p+2k)$ (with wrap around), receives from process $(p-2k)$, and overwrites the receive buffer with the incoming data. As in Fig. 3, 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_2^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 Fig. 3.

These three phases can be represented as follows:

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 k^{th} 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 used in the communication step because some received data-blocks will have to be resent in a later communication step. Pseudocode of this algorithm is shown as *Algorithm* 1. It can be seen from the algorithm (*Algorithm* 1 lines 6-16), that the total number of iterations performed by the Bruck algorithm is $O(\log P)$, unlike spread-out, which is $O(P-1)$ (*Algorithm* 2 lines 3-10).

Algorithm 1 The Bruck Algorithm

```
1: p ← process rank id
2: if p ∈ P then
3:    for i ∈ [0, P − 1] do
4:       R[i] = S[(p + i)%P] // initial rotation
5:    end for
6:    for k = 1; k < P; k <<= 1 do
7:       for i ∈ [1, P − 1] do
8:          if i & k then
9:             pack data for exchanging
10:         end if
11:      end for
12:      recvp ← (p + k)%P
13:      sendp ← (p − k + P)%P
14:      send data to sendp and receive data from recvp
15:      replace sent data with received data
16:   end for
17:   for i ∈ [0, P − 1] do
18:      R[i] = R[(p − i + P)%P] // final rotation
19:   end for
20: end if
```

B. Spread-out algorithm

The spread-out algorithm is a general implementation of the all-to-all communication, which divides all-to-all communication into n $(= P-1)$ communication rounds, and each round exchanges one data-block between two processes using non-blocking point-to-point communication functions. Specifically, each process posts all receiving requests using `MPI_Irecv` in a loop, then all sending requests using `MPI_Isend` in a loop, followed by an `MPI_Waitall`. Furthermore, in order to avoid communication congestion, each process distributes the sources and destinations based on its rank, such that each process sends to a different target process at each communication step. Pseudocode of this algorithm is shown as *Algorithm* 2:

Algorithm 2 The spread-out Algorithm

```
1: p ← process rank id
2: if p ∈ P then
3:    for i ∈ [0, P − 1] do
4:       src ← (p + i)%P
5:       p receives data-block src, p from src
6:    end for
7:    for i ∈ [0, P − 1] do
8:       tag ← (p − i + P)%P
9:       p sends data-block p, tag to tag
10:   end for
11:   Wait for all communication requests to be completed.
12: end if
```

III. Our Visualization tool

We have developed a visualization tool to clearly demonstrate all steps of the Bruck and spread-out algorithms. The web-based tool is free and open for everyone hosted at https://naexris.github.io/mpi-vis/ with the code base at https://github.com/naexris/mpi-vis/. It has the potential to be used in a pedagogical environment such as parallel computing classes and MPI related workshops. Along with demonstrating the workings of Bruck and spread-out, it can also illustrate the key differences between them through the visual and dynamically updated information that is presented in a clear and concise way. It focuses on readability and interactivity to ease comparing various inputs and, in particular, gives insight about how those inputs affect the number of communication steps and the amount of aggregate data sent. We hope to make this tool available to users who want to teach, present, or learn about the HPC topics presented through this tool. We believe that users who want to teach or present these topics, such as professors, will benefit from the added clarity of visuals and animations compared to traditional methods seen from PowerPoint slides and static figure representations. Users who want to learn, such as students, will benefit from the easy to understand step by step approach, where each visual step can offer insight through dynamically updated readings and visuals.

The visualization tool comes with information to help those interested in learning more about how these `MPI_Alltoall` algorithms work. For this version, `MPI_Alltoall` is the focus of the tool and the only listed collective routine on the front page. Information on `MPI_Alltoall` is given when navigated to to provide more information on the collective. A list of the supported algorithms, spread-out and Bruck, are shown with extended information summarizing how these two algorithms work. Clicking on an algorithm will allow a user to navigate to that algorithm's visualization page.

A. Creating a parameterized visualization

The visualization page, shown in Fig. 4, is the main feature of the tool. Its layout offers a step by step visual that is designed with the following goals in mind:

1) Illustrate the step-by-step process of a selected algorithm with intermediate visuals.
2) Offer key insights to the current displayed step (Fig. 5).
3) Offer the ability to parameterize process count and elements per data-block.
4) Supply reasoning why certain algorithms might be used for certain parameter values.

Each part of this visualization is dynamically updated such that information is supplied for every step, and intermediate step, of the process. As for the content itself, users can select either spread-out or Bruck to visualize. Once an algorithm is selected, the view displays three sections. At the top, the tool allows users to adjust parameters such as the number of processes and the elements per data block. Process count can currently be altered from $2-32$ and elements per data-block can be altered from $1-4$. These values were chosen mainly due to screen space and visual clarity. Initial data values, shown in the panels in the middle, will update in real time to reflect the changes done from these parameters. The parameters can be adjusted as many times as needed until a step is initiated. Once

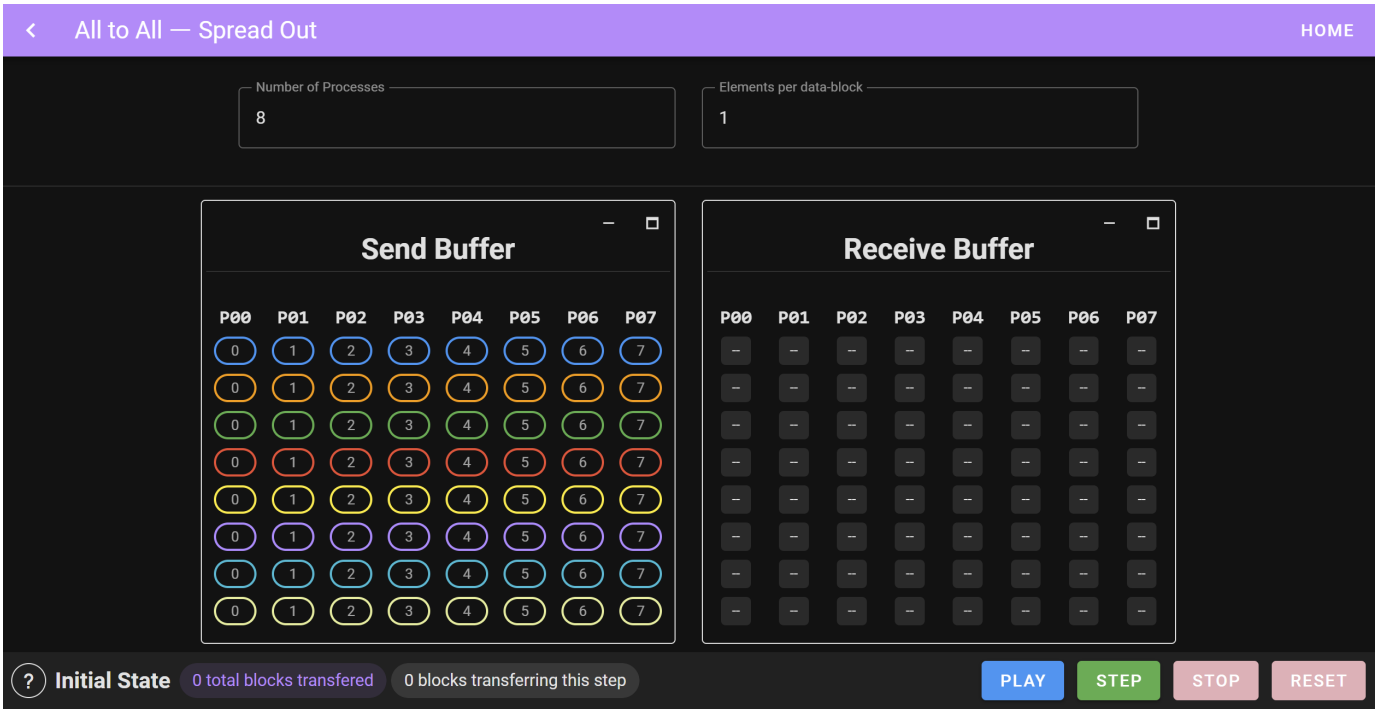


Fig. 4: Example of visualization page for the spread-out algorithm with initialized data. Eight processes and one elements per data-block has been parameterized.

parameters have been set, a user can execute each step of the selected algorithm to see how data is moved between processes and buffers. The footer of the visualization page contains four buttons that allow the user to control how steps proceed. Steps can advance manually by pressing `STEP` or automatically with `PLAY`. `STOP` will stop the currently playing animation, and `RESET` will return to the initial state of data. Info on each step is also provided as the user reaches them from the left side of the footer. A clickable menu popup, shown in Fig. 5, provides context about the current step that is displayed. All info in this panel is dynamically updated between each step. For example, the bit value example mentioned at the end of Fig. 5 is updated for each Bruck communication step. To the right of the step name, two sections also exist that show the total blocks transferred and the blocks transferring this step. Various animations are displayed to accompany intermediate changes to these values. Overall, this design provides an informative experience tailored for each algorithm so that can help users learn as they go.

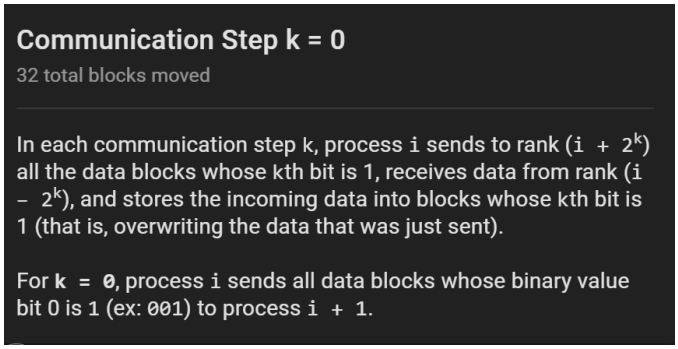


Fig. 5: Example of a current step (Bruck Comm step $k = 0$) information panel. It can be opened from the footer in Fig. 4.

For the main view, different kinds of visualization panels are provided to display step information for the algorithm. Currently, two types of panels exist: data view panels and the adjacency matrix panel. These panels can be specified when implementing an algorithm to provide the best visualization approach on a per algorithm basis. The most important and always available panel is the data view. In the data view, processes and their data blocks are denoted by colors and integers for clarity. Each initial data in a process is denoted by its respective process ID, and each data block within a process is denoted by a unique color. This setup allows the visualization to easily differentiate between data while not being overly cumbersome to look at. The end result makes for a clean visualization with easily discernible data. Additionally, a data value's shape and variant is altered to further denote when it is being manipulated during a communication step when data is present. Outlined ovals show data that is currently not being altered (Fig 6a), outlined rectangles show data that is about to be manipulated (Fig 6b), and filled in chips show data that has been altered or moved (Fig 6c). Note that if an integer is not present, no data is present in that block (Fig 6d). The data view can work in-place with one panel, or with a send and receive buffer between two different panels. The different visualization panel options can be configured on a per

(a) Not in use (b) In use (c) Moved (d) Null

Fig. 6: Example of different types of data blocks. The color corresponds to the block and the integer corresponds to the process it came from.

algorithm basis for newly added algorithms. Panels are chosen based on the proposed best way to convey the algorithm at hand.

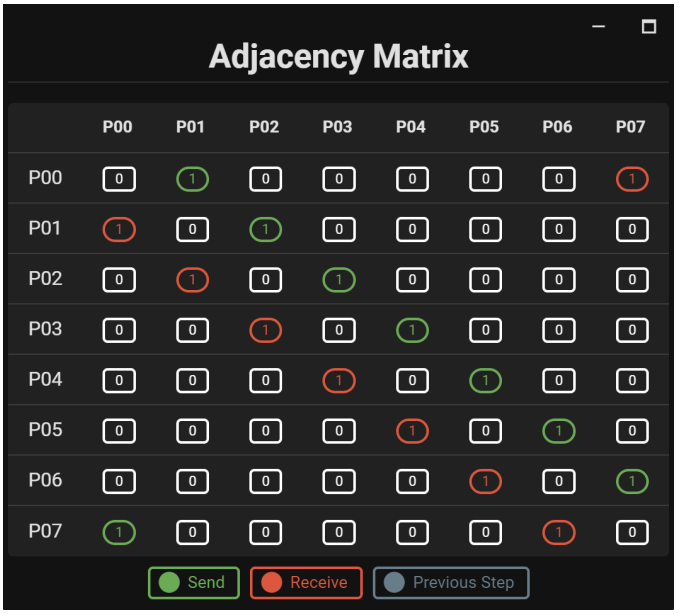


Fig. 7: Example of the adjacency matrix after Bruck communication step $k = 0$ has completed with 8 processes.

B. Bruck Visualization

For Bruck, the visualization tool's middle section comes with a data view on the left and an adjacency matrix on the right (Fig. 7). Because Bruck communication steps are altered in memory, a receive buffer becomes unnecessary and is instead replaced with an adjacency matrix. The send buffer is renamed to a data view, where the data is altered in-place and displayed to the user. As mentioned before, the colors, integer values, and fill correspond to the types of data that are present within the data view (Fig. 6). The adjacency matrix adds a history of where processes are sending and receiving data-blocks from. Its color meaning is slightly different and is labeled in Fig. 7. Green indicates when a process sends data to another process. Red indicates when a process receives data from another process. Grey indicates when either a send or receive happened on a previous step. These two panels combined offer a helpful look into algorithms like Bruck, because it can be difficult to follow in-place movements between multiple processes.

In Fig. 8, example images of the beginning of each step for Bruck with eight processes and a block size of one is shown. In Fig. 8a, the data is initialized across the eight processes. Fig. 8b performs the first rotation, where $S[i] = S[i + p]$ (p: rank). For example, P01's data blocks are shifted by one. No communication is done in this step so the adjacency matrix is unmodified.

Because the example in Fig. 8 has eight processes, three communication steps are performed for $k = [0..2]$. Fig. 8c involves the initial setup for communication step $k = 0$. Data blocks that will be moved are marked, and the adjacency matrix is updated to reflect the send operations that will be made. That is, process i will send to rank $(i + 2^k)$ all data blocks whose kth bit is 1. Fig. 8d shows the initial setup for communication step $k = 1$. All data blocks whose kth bits are 1 are marked for sending. Similarly, Fig. 8e shows the initial setup for communication step $k = 2$. Finally, Fig. 8f shows the final rotation that is performed to ensure that data blocks are in order. Again, no communication is involved in this step so the adjacency matrix is unaltered. At this point, the visualization has completed and can be rerun with the same or updated parameters as many times as needed.

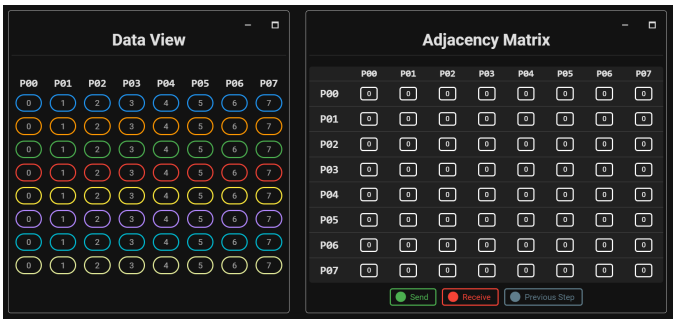
C. Spread-out Visualization

The Spread-out visualization page offers a slightly different view that comes with a send buffer and receive buffer panel. This is done because spread-out is different from Bruck in that it is not altered in place as data is instead copied from one buffer to another. The initial data setup (Fig. 9a) is however exactly the same, depending on the process count and block size. For Fig. 9, eight processes and a block size of one is shown. As noted previously, data blocks are presented differently based on its role in the current step (Fig. 6). Since data is copied over from one buffer to another, blocks will stay filled in the send buffer after they are sent to the receive buffer. This is contrary to Bruck, which only marks them at each step due to being in place.

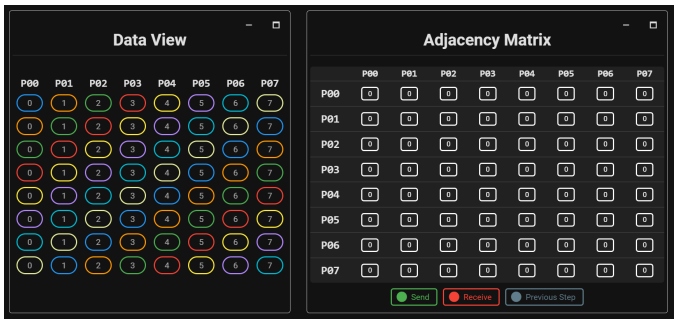
Fig. 9b to 9h shows the communication steps for $k = [0, 6]$. For $k = 0$ in Fig. 9b, process p sends to process $(rank+k)\%P$, where P is the total number of processes. So, in step $k = 0$, processes send data one process over into the sending process's rank index. Step $k = 1$, in Fig. 9c, will increment this by one and each process will send data two processes over. For eight processes, communication steps continue for $(P - 1)$ steps until all data has been transferred like in Fig. 9i.

D. Implementation details

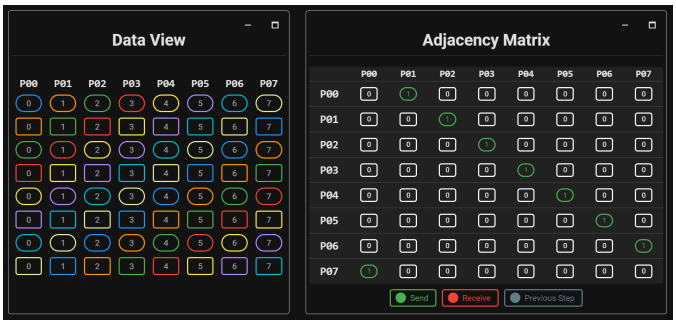
The visualization tool is built with Vue.js 3 and Vuetify. Vue.js is a progressive JavaScript Framework that enables our tool to be highly responsive as a single page application (SPA) with dynamic page routing [14]. Direct links to specific collectives or algorithms can be navigated to, and shared, so users can start visualizing data immediately. Pages are also set up in a way that makes it easy to expand this work and add support for more collectives and algorithms when needed.



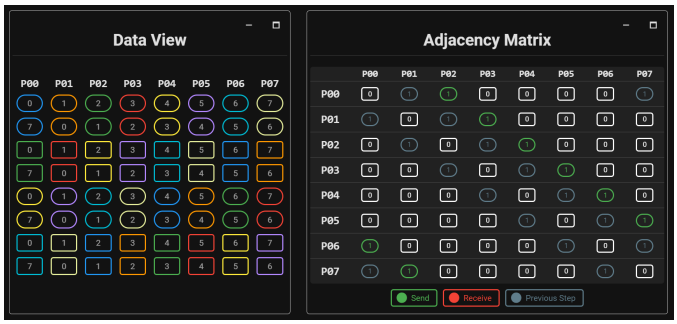
(a) Initial state



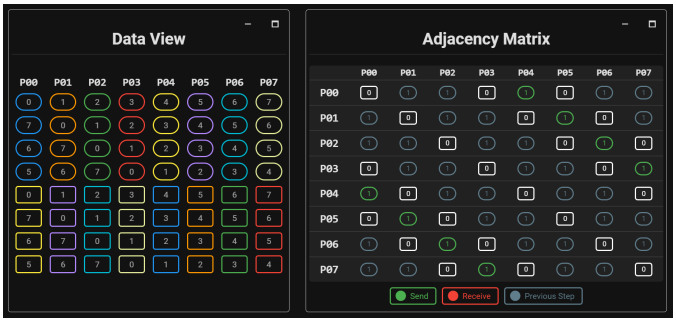
(b) First rotation



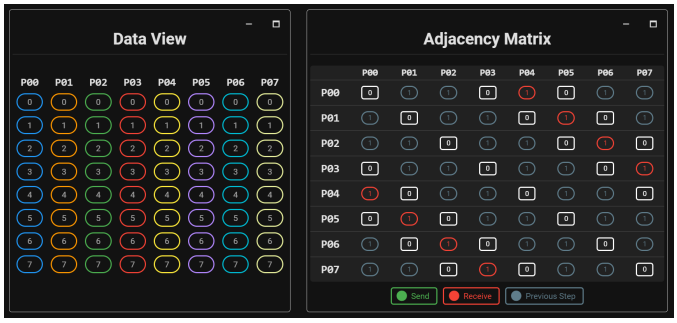
(c) Communication step $k = 0$



(d) Communication step $k = 1$



(e) Communication step $k = 2$



(f) Final rotation

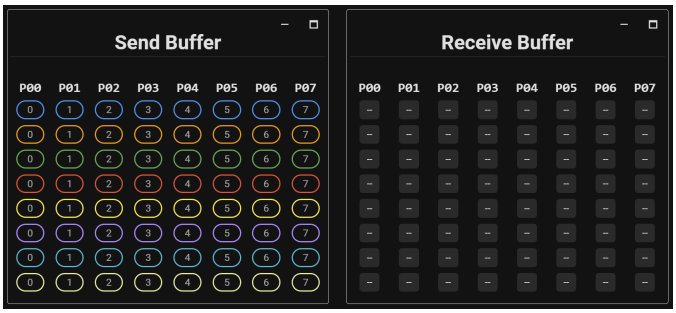
Fig. 8: Bruck Visualization Example

Under the hood, the code implementation of an algorithm can be added with relative ease. The display logic and algorithm logic are separated in such a way that adding a new algorithm will never require adjusting the display logic files. The display logic will pass the state of the data to the algorithm currently selected. Because Vue.js allows for dynamic routing, an algorithm is determined by the route (e.g. `/vis/alltoall/1` redirects to algorithm function "1" for `MPI_Alltoall`). So, if one wanted to add a new algorithm, they would stub out a function under the next increment, 2, leading it to be available at `/vis/alltoall/2`. The format of the function written will be very similar to the actual logic of the algorithm being implemented. The key difference is that one must account for the current step, which the display logic will pass in. At each step, the algorithm can pass back data to the display logic to update states and information given on the page.

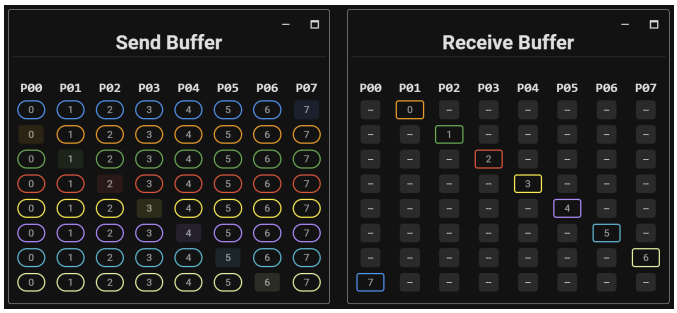
Vuetify is used on top of Vue.js to provide a design framework styled after Google's Material Design [15]. It enables the tool to be visually easy to comprehend while also being easy for any device or browser to run. Mobile device and computer browsers can run the tool with identical feature parity. This allows anyone to access it from any type of device. Only slight style changes will occur to ensure the best experience on each type of screen. We also use ApexCharts.js to provide a simple graph visualization of Bruck and spread-out complexity [16]. Overall, the technologies incorporated into this tool make it easy to use, navigate, and expand.

E. Pedagogical Environments

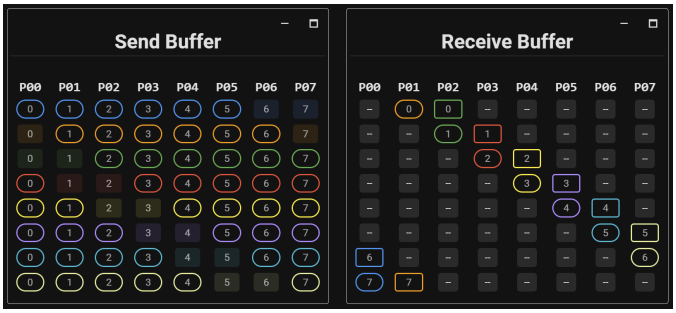
The visualization tool provides a number of potential use cases for pedagogical environments. Besides offering a visual understanding of different implementation strategies, the tool provides the ability to quantify the difference in the amount of



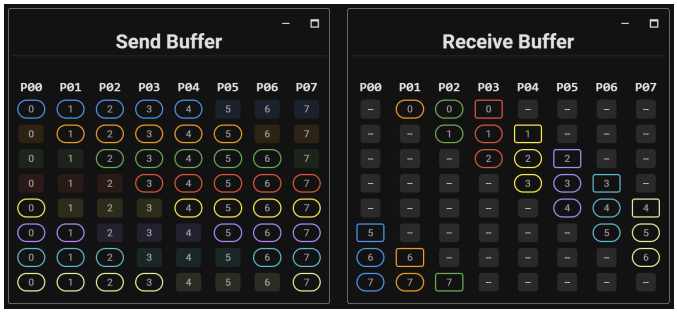
(a) Initial state



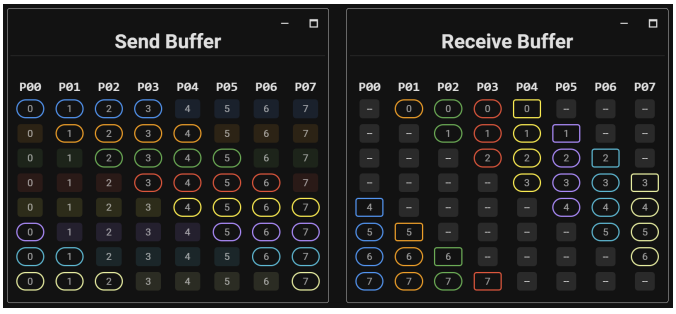
(b) Communication step $k = 0$



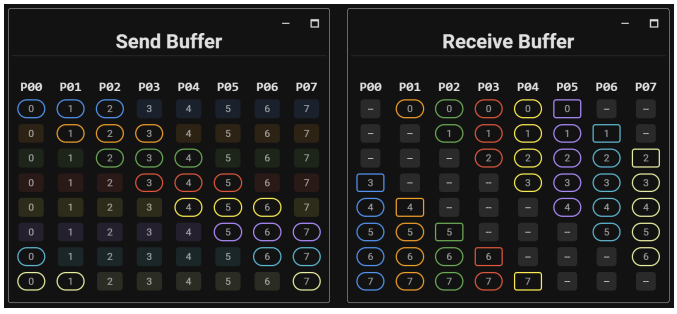
(c) Communication step $k = 1$



(d) Communication step $k = 2$



(e) Communication step $k = 3$



(f) Communication step $k = 4$

(g) Communication step $k = 5$

(h) Communication step $k = 6$

(i) Final step

Fig. 9: Spread-out Visualization Example

data transmitted between Bruck and Spread-out. The quantification provides users, such as first year HPC graduate students, insight on why a specific algorithm may be chosen over another for a specific amount of processes. For example, the insight addresses questions concerning latency and bandwidth restrictions.

Step count can also be compared between Bruck and Spread-out. Students can adjust the process count and see how it affects each algorithm's communication step count. The detailed step info also provides added context of the chosen algorithm so that students can understand steps more easily.

In the future, work can be done to open the framework of this tool to allow a user to add in their own implementations directly through the web page. Currently, this can only be done offline before the website is compiled for the web. For example, an instructor could assign a project that asks a student to implement a variation of Bruck. The student would be able to see the visualization tool work in real time based on their adjustments. Change in the framework would also allow users to adjust implementations that already exist, such as adjusting the logarithmic complexity of Bruck.

IV. Conclusion and Future Work

In this paper, we designed a parameterized visualization system for `MPI_Alltoall`, one of the most widely used collective routines, that illustrates the technical details of the spread-out algorithm and the Bruck algorithm, as well as the selection decision tree between them. This system depicts an area plot of the decision tree that identifies which algorithm is being used for any given number of processes and message size. It also allows for a thorough technical comparison of the spread-out algorithm and the Bruck algorithm, two classic algorithms utilized in `MPI_Alltoall`. Our system is intended for HPC users who would like to learn or teach about the detailed implementation of `MPI_Alltoall`, which can greatly cut the learning curve.

For now, we only focus on the collective routine `MPI_Alltoall` and demonstrate the advantage of our visualization system in helping users fully understand the algorithms utilized in `MPI_Alltoall`, thereby shortening the learning curve dramatically. There is also a renewed interest [17]–[19] in optimizing all-to-all communication by modifying the Bruck algorithm to take advantage of the deepening memory hierarchy of the modern HPC system. In particular, these algorithms take advantage of the multi-node-multi-core architecture, developing hierarchical versions of the Bruck algorithm. We also plan to extend our visualisation to incorporate these modified versions of the Bruck algorithm. Additionally, several other collective routines may be visualized in this manner, such as `MPI_Allgather`, `MPI_Alltoallv` [10] and `MPI_Allreduce`. Both `MPI_Allgather` and `MPI_Allreduce` are commonly used in HPC applications. `MPI_Allgather` enables each process to collect data from all other processes, whereas `MPI_Allreduce` enables each process to combine values from all processes. The standard implementations of both these collective routines, like `MPI_Alltoall`, use a combination of several algorithms and a decision tree to select between them. As a result, we intend to enable our system to visualize these collective routines in the future.

V. Acknowledgement

This work was funded in part by NSF RII Track-4 award 2132013 and NSF collaborative research award 2217036. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the ThetaGPU supercomputer located at the Argonne National Laboratory.

References

[1] S. Kumar and T. Gilray, "Distributed relational algebra at scale," in *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019.

[2] J. Doi and Y. Negishi, "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, 2010, pp. 1–9.

[3] H. Sundar, D. Malhotra, and G. Biros, "Hyksort: a new variant of hypercube quicksort on distributed memory architectures," in *Proceedings of the 27th international ACM conference on international conference on supercomputing*, 2013, pp. 293–302.

[4] S. Kumar and T. Gilray, "Load-balancing parallel relational algebra," in *International Conference on High Performance Computing*. Springer, 2020, pp. 288–308.

[5] K. Fan, K. Micinski, T. Gilray, and S. Kumar, "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, 2021, pp. 965–972.

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

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

[8] Q. Kang, R. Ross, R. Latham, S. Lee, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving all-to-many personalized communication in two-phase i/o," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[9] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[10] K. Fan, T. Gilray, V. Pascucci, X. Huang, K. Micinski, and S. Kumar, "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*, 2022, pp. 172–184.

[11] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel computing*, vol. 20, no. 3, pp. 389–398, 1994.

[12] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[13] J. L. Träff, A. Rougier, and S. Hunold, "Implementing a classic: Zero-copy all-to-all communication with mpi datatypes," in *Proceedings of the 28th ACM international conference on Supercomputing*, 2014.

[14] https://vuejs.org, Vue.js Home Page.

[15] https://vuetifyjs.com, Vuetify Home Page.

[16] https://apexcharts.com, ApexCharts.js Home Page.

[17] A. Bienz, S. Gautam, and A. Kharel, "A locality-aware bruck allgather," *arXiv preprint arXiv:2206.03564*, 2022.

[18] P. Alizadeh, A. Sojoodi, Y. Hassan Temucin, and A. Afsahi, "Efficient process arrival pattern aware collective communication for deep learning," in *EuroMPI/USA'22: 29th European MPI Users' Group Meeting*, 2022, pp. 68–78.

[19] G. Chochia, D. Solt, and J. Hursey, "Applying on node aggregation methods to mpi alltoall collectives: Matrix block aggregation algorithm," in *EuroMPI/USA'22: 29th European MPI Users' Group Meeting*, 2022, pp. 11–17.