Optimized Alltoall Algorithm for Low-Diameter Network Topologies

Marcel Ferrari, Christopher Lompa, Nicola Lo Russo, Francesco Cavalli, You Wu

Department of Computer Science, ETH Zurich Switzerland

Abstract

Low-diameter network topologies offer many advantages in communication latency, cost and energy efficiency [1]. Standard implementations of collective algorithms may however introduce artificial bottlenecks by creating network congestion that can reduce these benefits. This work presents a simple yet effective solution to mitigate this problem through communication randomization, buffer scattering and request queueing. We showcase a high performance implementation of Alltoall that demonstrates the potential of these techniques for the Slim Fly topology [1]. Our implementation achieved up to 3 times faster execution compared to the most efficient implementation of Alltoall in OpenMPI.

Keywords: Alltoall, MPI, OpenMPI, Slim Fly, High Performance Computing, Benchmarking

1. Introduction

Congestion Problem. Standard implementations of MPI collectives generally do not fully leverage the advantages of low-diameter topologies. These algorithms usually schedule communication in an ordered sequence by iterating over all ranks in a predefined and ordered pattern. This can be harmful as the communication schedule will not be uniformly distributed over all paths involved in the collective function. Instead paths will alternate between states of complete inactivity and critically high activity, leading to network imbalances, congestion and ultimately performance loss.

There are two ascertainable ways to mitigate this effect: smart routing algorithms and optimization of MPI collective functions. These solutions are not mutually exclusive: routing algorithms are designed to improve the communication by selecting the most efficient path for the data to travel between nodes. This can involve techniques such as load balancing, traffic shaping, and congestion avoidance [2]. On the other hand MPI algorithms can optimize the schedule in which MPI processes communicate.

This work focuses on optimizing the Alltoall collective, as it was a topic that had not been previously explored for the Slim Fly topology.

Processes Randomization. To tackle the problems described above, we propose a randomized scheduling scheme in the MPI collective algorithms. With this work we want to show that randomization, supported by other optimizations, is a valid strategy to improve the latency[1] of collectives.

It is worth mentioning that communication randomization can be achieved, in principle, by simply reordering the ranks when launching an MPI job. This produces some performance benefits when selectively testing single collective algorithms. However, this approach can be detrimental in real-world scenarios, as scientific applications are often optimized to exploit neighbouring process communication.

Our approach is instead based on randomizing only the algorithms responsible for collective communication, while maintaining the order of the ranks. This ensures a probabilistic, unbiased and uniformly distributed scheduling scheme, while still allowing for the exploitation of neighbouring process communication.

In the next section we showcase our implementation of Alltoall. In section 3 we present the results compared to the fastest OpenMPI implementation and we end with our considerations and conclusions in section 4.

[1] In this context, latency is defined as the time required to complete an MPI collective operation.

Function 1: get shuffled ranks function

```
Input  : size, seed
Output: array with ranks in random order
Function get_shuffled_ranks(size, seed) → int[size] is
    rax = int[size]
    // Fill with ranks from 0 to size-1
    fill(rax, 0, size)
    // Shuffle ranks in O(N) using seed
    shuffle(rax, seed)
    return rax
end
```

2. Randomized Alltoall algorithm

In this section we describe in detail our Alltoall algorithm and optimization strategies used in our implementation.

Dynamic Algorithm Selection Strategy. Our implementation employs a dynamic selection strategy, which differentiates various scenarios based on message buffer size and total number of processes.

The implementation distinguishes three different scenarios in increasing order of buffer size. For small buffers[2] our implementation is not randomized. For medium buffers[3] and a small number of processes, we utilize our randomized scatter algorithm (1), while for a larger number of ranks we opt for our randomized Sendrecv algorithm (2). Finally for large buffers[4] we use a randomized segmented Sendrecv algorithm (3).

Rank shuffling. Function 1 showcases the *get_shuffled_ranks* function, which is used by all our algorithms. It returns an array containing the ranks from 0 to $size-1$ shuffled randomly according to a given seed.

Scatter algorithm. Algorithm 1 outlines our randomized Scatter algorithm, which is particularly well-suited for medium buffers with few number of processes. It performs Alltoall communication by randomly posting all *Irecv* requests first and then all *Isend* calls.

Algorithm 1: Randomized Scatter

```
Input: send_buf, recv_buf, byte_count
size = get_num_procs()
my_rank = get_rank()
ranks = get_shuffled_ranks(size, size)
send_to = -1, recv_from=-1
reqs = Request[2×size]
preq = reqs[0]
for i = 0 to size -1 do
    recv_from = ranks[i]
    temp_recv = recv_buf + recv_from × byte_count
    Irecv(temp_recv, byte_count, recv_from, preq++)
end
for i = 0 to size -1 do
    send_to = ranks[i];
    temp_send = send_buf + send_to × byte_count
    Isend(temp_send, byte_count, send_to, preq++)
end
Waitall(2×size, reqs)
```

Randomized algorithms. Algorithm 2 displays our randomized Sendrecv algorithm. The communication scheme is handled by a randomized anti-circulant matrix[5], where the first row is chosen to be the randomized *ranks* array. Given the mathematical properties of this matrix, we do not need to store the whole matrix, as we can obtain the entries via modular arithmetic.

The matrix is interpreted in the following way: each row of the matrix corresponds to a rank and each column to an iteration. At every iteration a process will send data to the correct entry in the row corresponding to its rank. Then it will receive data from the process whose row contains its rank at that iteration. Figure 1 gives an example from the point of view of rank 1 of a 4-process job. In this example the randomized *ranks* array is $\{1, 3, 2, 0\}$.

Note that with this algorithm it is important to

[2] empirically less than 512 bytes
[3] empirically between 512 and 16384 bytes
[4] empirically greater than 16384 bytes

[5] An anti-circulant matrix is a special type of matrix where each row is obtained by shifting the entries of the previous row to the left by one positions with wrap-around.

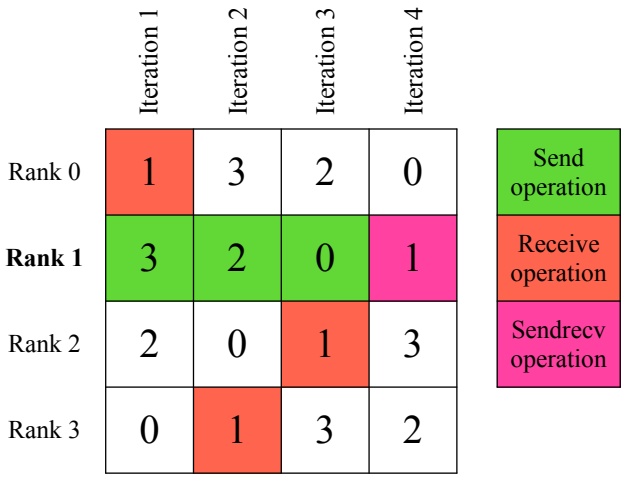

Figure 1: Alltoall communication pattern of rank 1 of a 4 process job. The randomized ranks array of this anti-circulant matrix is $\{1, 3, 2, 0\}$.

generate the *ranks* array with the same seed, as we need to obtain the same shuffled array on all processes to prevent deadlocks.

We also introduced a queue system to keep a constant average number of posted Isend and Irecv requests. This maximizes the data throughput of the network by balancing the number of ongoing communication operations and their individual data-flow. The queue keeps track of how many requests have been posted and executes a *Waitall* when a certain threshold is surpassed.

Finally algorithm 3 presents our segmented randomized Sendrecv algorithm. It builds upon Algorithm 2 by additionally segmenting the data into blocks, which are then communicated in a randomized pattern. This improves temporal uniformity of the path selection scheme. In this case, the queue system has an even greater impact on performance, as the number of total requests is *n_blocks* times that of the other algorithms.

When implementing this algorithm particular care must be given when the buffer size is not a multiple of the block size. This can be handled by implementing a trailing Scatter Alltoall, which keeps track of the offset of already transmitted blocks and communicates only the remaining bytes.

Algorithm 2: Randomized Sendrecv

```
Input  : send_buf, recv_buf, byte_count
Constants: max_queue
size = get_num_procs(), my_rank = get_rank()
ranks = get_shuffled_ranks(size, size)
reqs = Request[max_queue], preq = reqs[0]
queue_size = 0, send_to = -1
recv_from = index_of(my_rank, ranks) + 1
for i = 0 to size -1 do
    send_to = ranks[(i+my_rank) % size]
    recv_from = (recv_from - 1 + size) % size
    if queue_size ≥ max_queue then
        Waitall(queue_size, reqs)
        queue_size = 0
        preq = reqs[0]
    end
    temp_send = send_buf + send_to × byte_count
    temp_recv = recv_buf + recv_from × byte_count
    Irecv(temp_recv, byte_count, recv_from, preq++)
    Isend(temp_send, byte_count, send_to, preq++)
    queue_size = queue_size + 2
end
if queue_size ≠ 0 then
    Waitall(queue_size, reqs)
end
```

Algorithm 3: Randomized segmented Sendrecv

```
Input: send_buf, recv_buf, byte_count
Constants: max_queue, block_size
size = get_num_procs(), my_rank = get_rank()
ranks = get_shuffled_ranks(size, size)
reqs = Request[max_queue]
n_blocks = byte_count / block_size
send_to = -1, recv_from=-1
for block = 0 to n_blocks-1 do
    preq = reqs[0], queue_size = 0
    recv_from = index_of(my_rank, ranks) + 1
    for i = 0 to size -1 do
        send_to = ranks[(i+my_rank) % size]
        recv_from = (recv_from - 1 + size) % size;
        temp_send = send_buf + send_to × byte_count
        + block × block_size
        temp_recv = recv_buf + recv_from × byte_count
        + block × block_size
        if queue_size ≥ max_queue then
            Waitall(queue_size, reqs)
            queue_size = 0
            preq = reqs[0]
        end
        Isend(temp_send, block_size, send_to, preq++)
        Irecv(temp_recv, block_size, recv_from, preq++)
        queue_size = queue_size + 2
    end
    if queue_size ≠ 0 then
        Waitall(queue_size, reqs)
    end
end
```

3. Results

In this section we summarize the results of our randomized algorithm and compare them to the standard implementation.

3.1. Experimental Setup

CPU Model	CPU Core Info	Memory	OS
Intel(R) Xeon(R) CPU E5-2660 v2	2x10 @ 2.20 GHz	32 GB	CentOS Linux release 7.9.2009
Network Adapter	**Network Stack**	**Switch**	**IB Links**
MT27500 Family ConnectX-3	OFED LINUX 4.9-3.1.5.0	Mellanox SX6036 switch (56 Gbps)	56 Gbps FDR links

Table 1: hardware configuration of compute nodes.

The experimental results were obtained on the Slim Fly cluster at CSCS (Swiss National Supercomputing Centre). The cluster comprises 200 nodes.

We performed the benchmarks using OpenMPI V1.10.7 compiled via Spack 0.19.0[6] using GCC version 12.2.0. OpenMPI was run over IB (Infiniband) using the *–mca btl self,sm,openib* flag to enable both IB and SM (shared memory) communication.

Our implementation was developed as a plug-in on top of OpenMPI: it utilizes the MPI profiling interface to replace calls to Alltoall with our custom algorithms, while preserving the option to use the standard implementations when randomization is not suitable. It was compiled using GCC version 12.2.0 with the flags *-O3 -march=native*.

The synthetic benchmarks were performed using a modified version of OSU Micro-Benchmarks v7.0[7] which generated all the data needed for a scientifically and statistically sound assessment of the results [3]. Even though micro benchmarks are well suited for this work, we are aware of the limitations and potential problems of such experiments [4].

The benchmarks were run on 14, 16, 28, 32, 56 and 64 nodes.

3.2. Benchmark Results

We compare the performance of the standard implementation of Alltoall to our randomized algorithms and report the results. For the purpose of performance analysis, randomization was enabled even for small buffer sizes.

The benchmarks were performed with the standard powers of two buffer sizes selected by OSU using 1 ppn (processes per node) and 10 ppn. The buffer size limit was determined automatically based on the detected hardware and the number of processes per job. Each set of parameters was measured by timing 50 calls to Alltoall. We report the median latency to avoid excessive weighting of non-representative results. Furthermore, we compute the 95% confidence interval using the bootstrapping method [5].

Figure 2 shows the speedup of the randomized algorithm compared to the standard implementation and reports the size of the confidence interval (in superscript) as percentage of the median. Values smaller than 1% are omitted. The total speedup was computed as the ratio between the sum of the random latencies and the sum of the normal latencies.

Figure 3 presents a log-log plot of the median latency of Alltoall as a function of buffer size. The upper row depicts results for the standard implementation, while the lower row shows data for the randomized version. The graph also displays the confidence intervals as color shadings.

The data suggests polynomial asymptotic convergence of the form $l = C \cdot s^K$, with latency l and buffer size s. The values of C and K have been computed as averages of the slopes of the linear regressions starting from buffer size 32768. R is the ratio between the C constants of the randomized data set and the normal data set.

Speedup. Figure 2 shows the relative speedup or slowdown for different buffer sizes.

For medium and large buffers the randomized Alltoall consistently delivers significantly better performance compared to the standard version, with up to 3× improvement. For small buffer sizes randomization is not beneficial.

The threshold after which randomization is convenient seems to be independent of the number of nodes and ppn, and is either 256 or 512 bytes. There is no performance gain for jobs with few nodes, as the majority of communication occurs through shared mem-

[6] https://spack.readthedocs.io/
[7] https://mvapich.cse.ohio-state.edu/benchmarks/

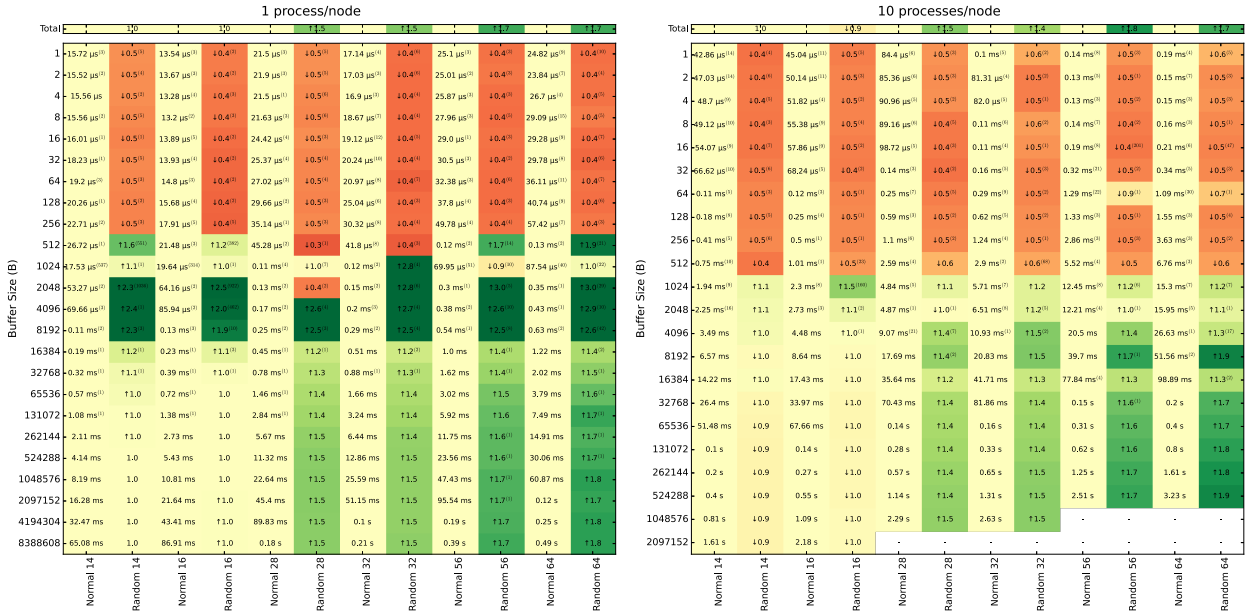

Figure 2: Heatmaps of relative speedups of randomized Alltoall algorithms over the standard implementation. "Normal"-labeled columns show the median latency of the standard Alltoall implementation. "Random"-labeled columns show the relative speedups of the randomized version. Both scale as functions of buffer size (B) and number of nodes. The 95% confidence interval is reported in superscript brackets where it is greater than 1%. The "Total" row shows the aggregated data.

ory and the IB network is not stressed enough to suffer from congestion. However, for medium to large jobs it offers a substantial performance improvement. It is also important to keep in mind that the observed decrease in performance is a result of intentionally applying randomization to tiny buffers for benchmarking purposes. Our implementation would have otherwise produced identical results to OpenMPI.

Asymptotic convergence. Figure 3 shows that for small buffer sizes the median latency is almost constant. Between 256 and 4096 B we observe unpredictable behavior with wide confidence intervals in both implementations. We believe this is caused by outdated IB network adapters, which are no longer supported. Asymptotically both implementations exhibit polynomial convergence of degree $K \approx 1$ (linear scaling) which is consistent with both algorithms having the same complexity. However, in the real world constants matter: we observe an aggregated ratio $R \approx 0.6$, indicating an average 40% reduction of the scaling factor C for our randomized algorithms, which justifies the measured speedups for medium and large buffer sizes.

4. Conclusions

Considerations. The Slim Fly cluster used in this work is a prototype of its kind. As such, some of the results may be impacted by routing, links, and hardware issues. We were also limited to running experiments on a maximum of 64 nodes at a time. Currently the cluster only supports OpenMPI 1, although using the latest version should still guarantee state-of-the-art collective algorithms. For these reasons the absolute values of the results and the relative speedups should be interpreted with care. Nonetheless we believe that the general concepts and ideas remain valid and we expect them to translate to modern implementations of low-diameter topologies as well as in future iterations of Slim Fly clusters.

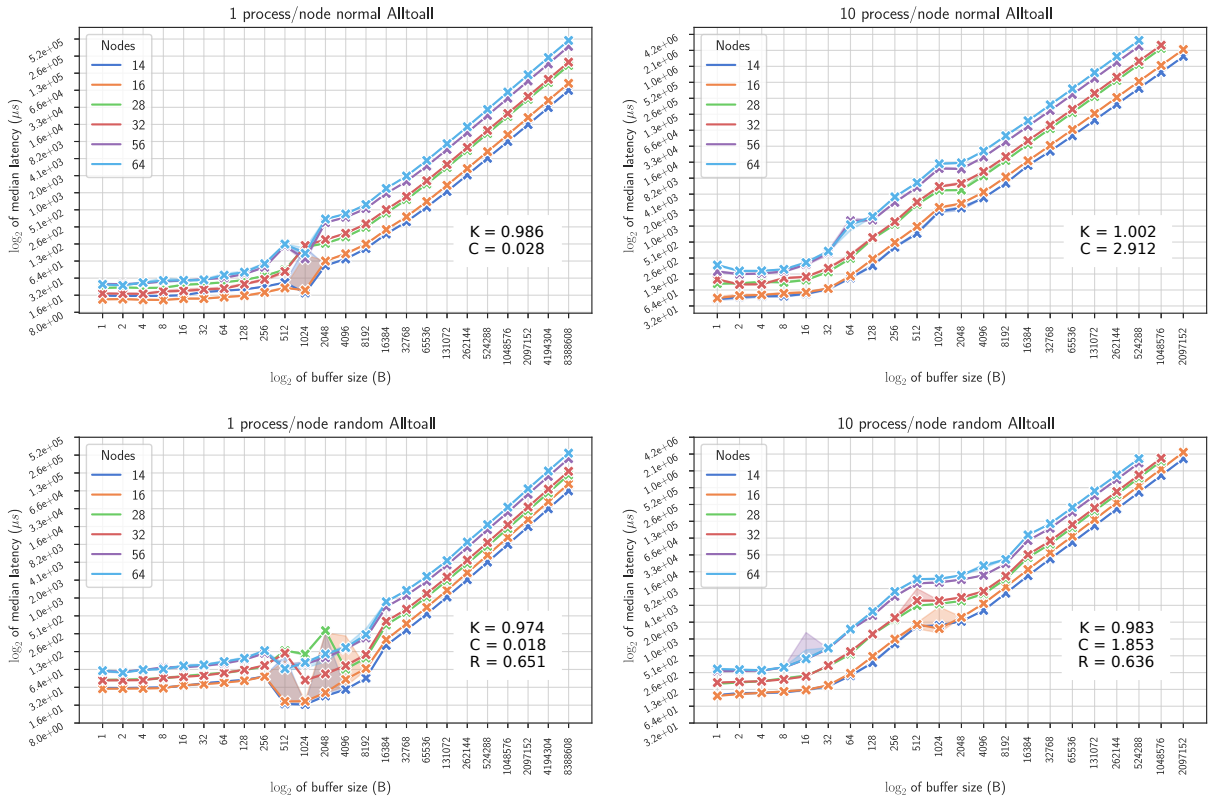

Figure 3: Graphs in log-log scale of the median latency (μs) of the Alltoall algorithm as a function of buffer size (B). The number of nodes varies from 14 to 64. The buffer size scales with powers of two.

Final Words and Future Works. This work proved the potential of randomization and other optimization techniques in collective communication algorithms. We highlighted their advantages for Alltoall, although we believe that the same concepts can improve other MPI operations as well. We were also able to provide insight on which optimization strategies can help the development of better performant collective algorithms for Slim Fly and other similar topologies. Future works could explore the application of these concepts in other collective operations as well as their effect on real-world applications. It would also be interesting to test how randomized collectives perform on more modern hardware, without the obstacles and constraints of the current Slim Fly cluster. We believe that standard MPI implementations could benefit by supporting quality randomized algorithms.

5. Aknowledgments

We would like to express our gratitude to Dr. Maciej Besta for his supervision of the project, to Nils Blach for his assistance with the project and the cluster, to Hussein N. Harake for his technical and overall support, to Dr. Andreas Jocksch for his key suggestions when designing our optimizations, to Dr. Daniele De Sensi for his intuition when debugging our code, and to the Swiss National Supercomputing Center (CSCS) for hosting the cluster. Their contributions were invaluable to the success of this work.

References

[1] M. Besta, T. Hoefler, Slim Fly: A Cost Effective Low-Diameter Network Topology, International Conference for High Performance Computing, Networking, Storage and Analysis (2012). doi:10.1109/SC.2014.34.

[2] J. Dongarra, P. Beckman, J. Chen, Optimization of Collective Communication Operations in MPI, International Journal of High Performance Computing Applications 19 (2) (2005) 173–187.

[3] T. Hoefler, R. Belli, Scientific Benchmarking of Parallel Computing Systems (2015) 1–12doi:10.1145/2807591.2807644.

[4] T. Hoefler, T. Schneider, A. Lumsdaine, Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale (2009).

[5] B. Efron, Bootstrap Methods: Another Look at the Jackknife, Annals of Statistics 7 (1) (1979) 1–26. doi:10.1214/aos/1176344552.