Benchmarking the State of MPI Partitioned Communication in Open MPI

Axel Schneewind[1] and Christoph Niethammer[1] (✉)

HLRS, Nobelstr. 19, 70569 Stuttgart, Germany
niethammer@hlrs.de

Abstract. The MPI 4.0 standard introduced partitioned communication for point-to-point messaging in 2021. This work evaluates the current state of the implementation in Open MPI for multi-threaded applications using a synthetic benchmark. A comparison to solutions using alternative communication schemes is made. The obtained results show that there can be some performance benefits with Open MPI. To understand the results for Open MPI, a brief look into the implementation is made. Based on this, some suggestion to improve the performance using a simple partition aggregation algorithm is made. Initial results for this are presented showcasing gains for small partition sizes, i.e., large partition counts.

Keywords: MPI · Benchmarking · Partitioned Communication.

1 Introduction

Looking at the TOP 500 list [5] today's HPC systems come normally equipped with CPUs that have several dozens of cores and are accompanied by GPGPUs with several hundreds of compute units that can run thousands of threads in parallel. This hardware trend requires HPC applications to use hybrid parallelism where classical MPI parallelizations are combined with shared memory parallelism, i.e., MPI+threads. However, this leads to issues when it comes to data exchange using MPI. MPI requires that message buffers are not modified after passing it to some MPI communication method. In case such buffers are related to thread parallel computations, those have to be synchronized before the MPI operation using the buffer can be started. As a result load imbalances in the shared memory parallelization are exposed and limit the scalability of the application. Further potential for overlapping communication and computation is lost - especially for larger buffers. In an alternative approach, each thread can issue MPI communication operations on its own - i.e., using the MPI_THREAD_MULTIPLE style. However, the aforementioned high number of threads brings MPI libraries and their internal messaging system to their limits in terms of scalability for a single MPI process due to overheads. A solution for this is the partitioned communication interface introduced in MPI 4.0 in 2021 [1]. Since then MPI libraries have now picked up and implemented this new interface. As a next step, these implementations are now getting optimized [2].

Within this work, the implementation of Open MPI [4] is reviewed in terms of its current performance. Therefore a benchmark is developed to compare various schemes that implement the partitioned communication pattern with various MPI methods against the now native partitioned communication interface implementation.

The contribution is structured as follows: First, some background around partitioned communication and its application for MPI+thread parallel programming in a CPU-centric execution context is provided. Section 3 looks at the implementations in Open MPI to elaborate on the later results and we suggest an aggregation algorithm to improve performance. In Section 4 the benchmark used for the evaluation is described together with the different schemes for marking partitions ready for transfer. In the following Section 5 results from the measurements are shown and discussed. Initial results for a basic implementation of our proposed aggregation algorithm are provided that show gains for small partitions, i.e., large partition counts.

2 Background

MPI offers different Point-to-point communication methods, which mainly differ in their respective interfaces and the amount of control they provide over the different phases of a transfer (i.e. initialization, starting, completion and freeing) [3]. Figure 1 shows flow diagrams of the ones that are relevant for this work. The most simple transfer mechanism is MPI_Send (blocking send), which performs all initialization, starting, completion and freeing steps associated with a transfer. Performing all transfer steps through a single function call, interrupts the calling application, preventing it from performing useful computations.

For this reason, nonblocking mechanisms can provide better overall performance. These separate the starting step from the completion of a transfer, allowing the application to continue running while a transfer is in progress. In the following, persistent send operations will be considered, which offers a separate function call for each transfer step.

Compared to persistent send, partitioned communication increases the granularity of the starting phase. Instead of marking an entire buffer of data as ready for transfer, the application can mark parts of it as ready through the additional MPI_Pready method. In the initialization step the application has to specify the partitioning of the data (by providing the size and number of partitions). The number of partitions may differ between the sender and receiver side.

2.1 Possible optimizations and performance improvements

Early-bird effect: As computations within an MPI process are often parallelized using CPU or GPU threads, parts of the data can be ready to transfer before others. By allowing partitions of the data to be marked ready individually, the MPI implementation can start transferring data as soon as the first partition is ready, instead of waiting for the entire buffer to be marked ready. As a result,

partitioned communication can provide more overlap between computation and communication, reducing overall time.

Flexibility concerning the implementation of Pready As the MPI standard poses few requirements on the Pready operation (most importantly, Pready does not have to start any transfer), an implementation can optimize the function for concurrent access with high thread counts. As Pready does not have to interact with network interfaces, it can also be implemented without requiring locks (in contrast to e.g. MPI_Send), possibly reducing overhead. Further, MPI_Pready can also be implemented such that it can be called by GPU threads.

Message aggregation: Generally, transferring a few large messages instead of many smaller ones reduces communication time, as each message is associated with some overhead on the sending and receiving side. Therefore, a partitioned transfer can be optimized by aggregating multiple partitions in a single message. In particular, while a transfer is in progress, an implementation does not have to initiate new transfers for each MPI_Pready call as these would have to wait. Instead, the implementation might wait until the running transfer is done and check if there are partitions ready that can be aggregated.

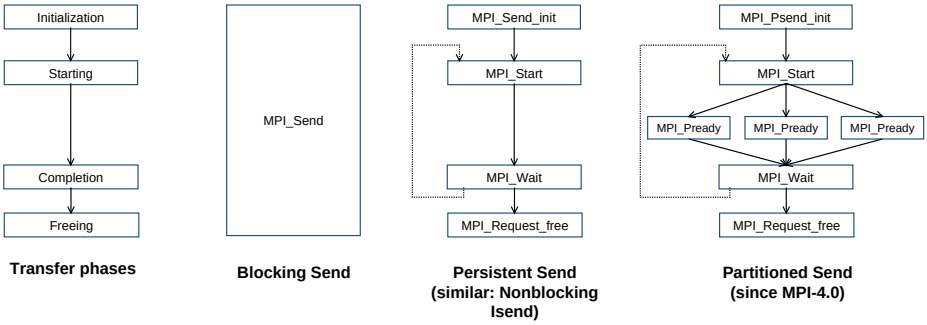


Fig. 1: Flow diagrams of different point-to-point interfaces

3 Current implementation and suggested improvement

3.1 Current implementation

OpenMPI As for version 5.0.3, OpenMPI implements partitioned transfers using its persistent send. More precisely, each partition is mapped to an internal persistent send request. On MPI_Pready() being called on any partition, the corresponding send request is started. Further, OpenMPI's progress engine occasionally calls MPI_Test() on the currently active requests.

3.2 Partition aggregation

In the following, a simple method of partition aggregation is described (see also Figure 2). It mainly consists of an m-to-1-mapping of user-provided partitions to internal partitions (assuming that the number of user partitions is a multiple of m). For the benchmarks, this is done by translating the Psend request that the user interacts with to an internal Psend-request with a different partitioning. However, this method could be integrated into OpenMPI directly without much modification required.

To track whether internal partitions are ready for transfer, each of them is associated with an atomic counter (initialized to 0 on MPI_Start). A Pready-call to user-partition p is then handled by incrementing the counter associated with the corresponding internal partition p'. If the counter reaches m (i.e. the number of user-partitions corresponding to p'), the internal partition can be marked ready for transfer by calling Pready on the internal request.

Using such a mapping from user-partitions to internal ones, an MPI-implementation can select the optimal number and size of the transferred messages, independent of the number of partitions that the application reports through MPI_Psend_init.

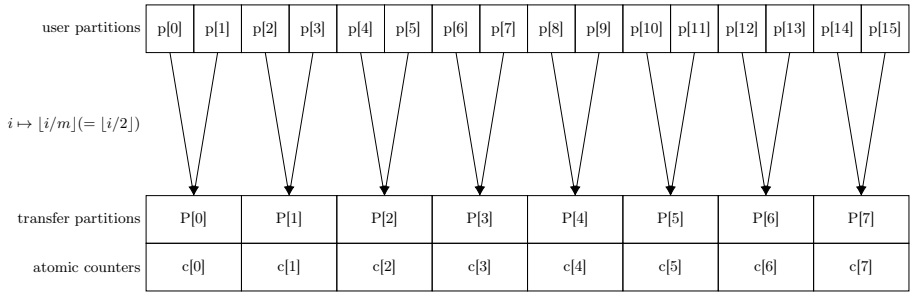


Fig. 2: General approach of the aggregation scheme with a $2 \to 1$-mapping

Table 1: Operations used on the send side for the different transfer mechanisms. Calls made for individual partitions p are marked with (p).

Mechanism	Initialization	Partition ready	Completion	Freeing
Send	-	MPI_Send	-	-
Persistent Send	MPI_Send_init(p)	MPI_Start(p)	MPI_Waitall	MPI_Request_free(p)
Isend	-	MPI_Isend(p)	MPI_Waitall	MPI_Request_free(p)
Psend	MPI_Psend_init	MPI_Pready(p)	MPI_Wait	MPI_Request_free

Table 2: Operations used on the receiver side. Calls made for individual partitions p are marked with (p).

Mechanism	Initialization	Receive partition	Completion	Freeing
Send	-	MPI_Recv	-	-
Persistent Send	MPI_Recv_init(p)	MPI_Start(p)	MPI_Waitall	MPI_Request_free(p)
Isend	-	MPI_Irecv(p)	MPI_Waitall	MPI_Request_free(p)
Psend	MPI_Precv_init		MPI_Wait	MPI_Request_free

```
# general initialization of send/recv method if needed

for i in 0...num_iterations:
    MPI_Barrier(...) # basic synchronization for time meassurement
    start_time[i] = MPI_Wtime()

    MPI_Start() # if needed (i.e. for Psend)

    # loop over partitions which is split up over multiple threads
    for p in 0...partitione_count in some order:
        send/recv partition p

    MPI_Wait(request) # completion if required by the send/recv method

    end_time[i] = MPI_Wtime()
# cleanup

bandwidth = buffer_size / mean(end_time - start_time)
```

Listing 1: Benchmark structure and time measurement

4 Benchmarking Partitioned Communication

4.1 Benchmark description

To compare the performance of partitioned communication implementations we developed a benchmark that compares it to various alternative communication patterns and provides several different schemes to mark individual partitions as ready for transfer.

The overall structure of the benchmark is presented in Listing 1. It consists of a main loop executing the communication pattern with one of the transfer mechanisms from Table 1 and 2 for multiple iterations. For the time measurements functionality from MPI is used. The final bandwidth is computed from the mean time of the iterations. The communication pattern executed in each iteration iterates over a buffer, which is split up into a count of partitions. The individual partitions are marked ready for transfer within this loop based on different schemes.

4.2 Loop schemes for marking partitions:

As the performance of possible optimizations can depend on the order in which partitions are marked ready, we evaluate a few schemes:

- **left-to-right:** Partitions are marked as ready in sequence starting from partition number zero. If multiple threads are used the order of partitions processed by each thread is in sequence starting from the thread's lowest assigned partition number.
- **randomized:** Partitions are marked ready in random order.
- **neighbourhood exchange pattern:** This scheme mimics a typical neighbour communication with halos in two dimensions. Here, each partition corresponds to a cell in a rectangular grid.

5 Evaluation

Benchmarks were run on the HAWK supercomputer system at HLRS, which consists of dual-socket nodes with AMD EPYC 7742 CPUs with 64 cores each and 256 GB of RAM. For the inter-node connection, HAWK uses Infiniband HDR with 200 GBit/s to implement a 9D-hypercube. The benchmarks were executed with two MPI processes distributed over two nodes. As MPI version Open MPI v5.0.3 with UCX version 1.14.0 was used. Everything was compiled using GCC version 13.1.0.

5.1 Comparison between send mechanisms

Figure 3 compares the effective bandwidth depending on partition size for blocking send, persistent send, and Psend, when marking partitions using a single thread.

Transferring the buffer as a single partition allows for bandwidths close to the theoretic bandwidth of 25 GB/s. For the blocking send, the bandwidth quickly drops when splitting the transfer into more messages. Persistent and partitioned sends, however, keep the high bandwidth when reducing the partition size all the way down to 2^{16}. When further reducing the partition size, the bandwidth drops to the level of blocking sends. Between persistent and partitioned send, differences are negligible, which can be explained by Psend being implemented using persistent send-requests internally.

For multiple threads marking partitions as ready, larger differences between persistent and partitioned send can be observed, as shown in Figure 4: Here, the bandwidths are shown for thread counts between 1 and 16. For persistent send, used with larger partitions, the bandwidth already drops by approximately 25% when increasing the thread count from 1 to 16. This can possibly be explained by some locking being present in the code-path of MPI_Start(). With higher thread numbers, one can expect further reductions in performance.

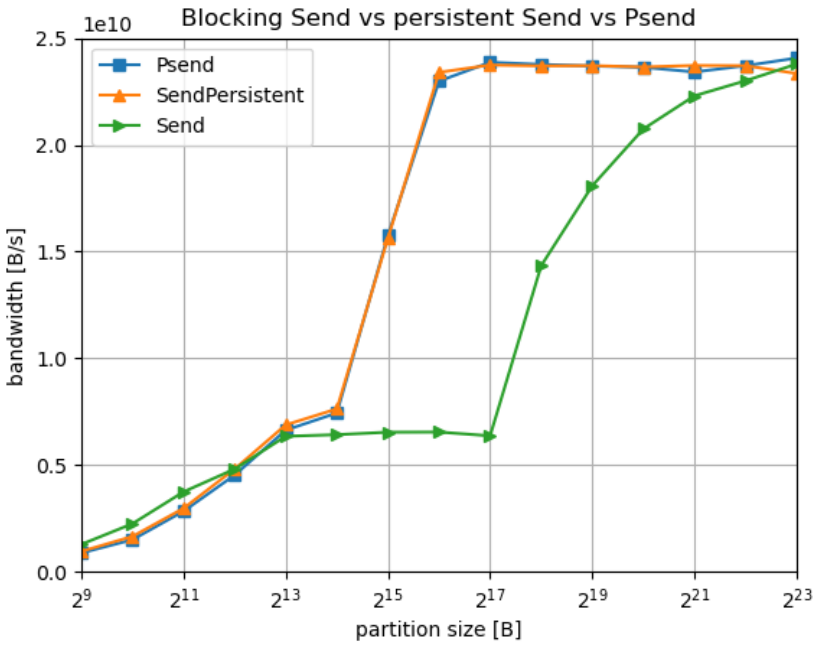


Fig. 3: Blocking, persistent, and partitioned for pure Open MPI

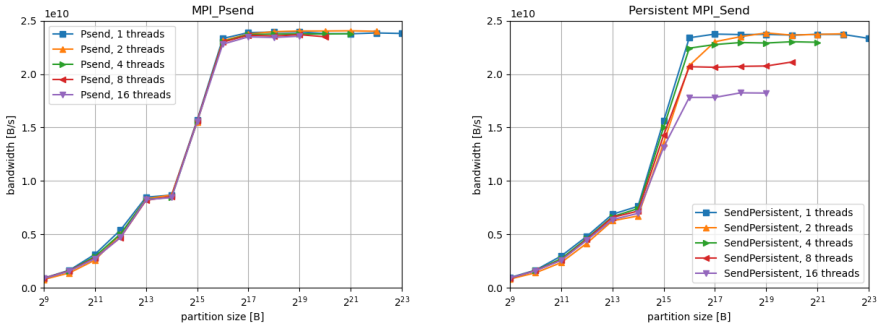


Fig. 4: Comparison of persistent to partitioned in multi-threaded setting

5.2 Performance depending on arrival patterns

Marking partitions in different orders does not influence the effective bandwidth, as seen in Figure 5. This can be expected, as OpenMPI performs one transfer per partition, regardless of the pattern in which they are marked ready.

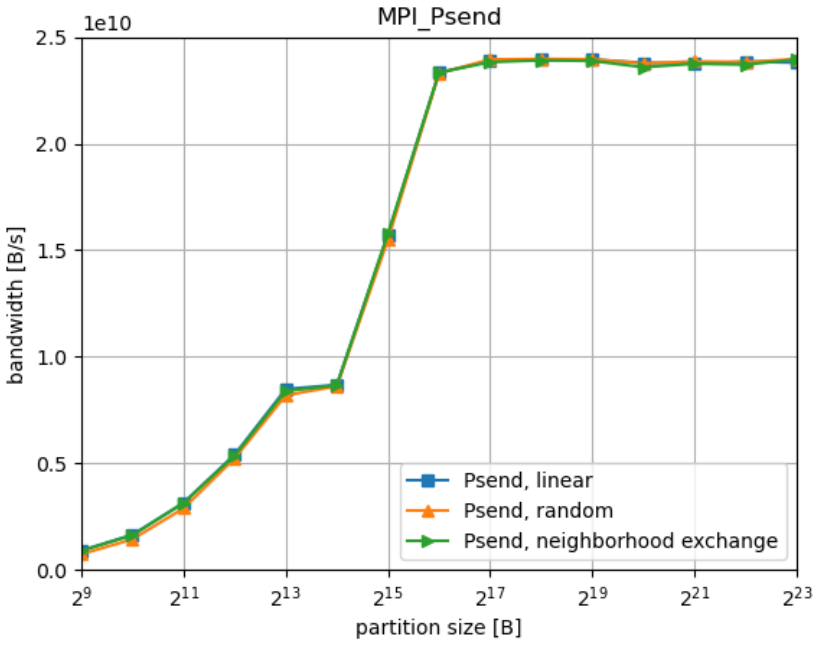


Fig. 5: Psend with different arrival patterns

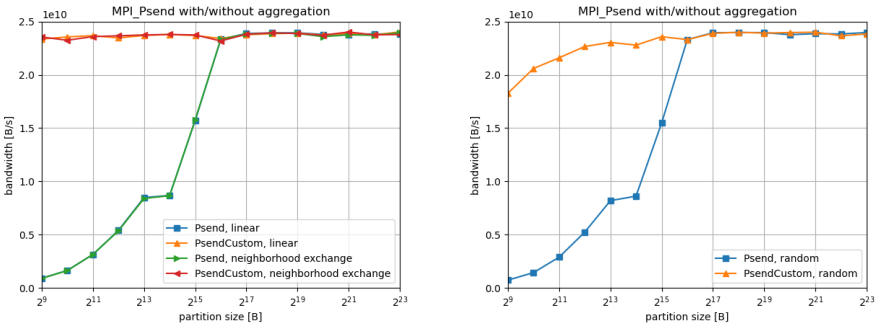
5.3 Partition aggregation

In case of the setup used here, the drop in effective bandwidth when using partitions with a size smaller than 64 kB can be prevented by mapping more fine-grained partitionings to internal partitions with a size of 64 kB. For other systems, the optimal size of partitions might differ and would need to be configured accordingly.

The effective bandwidth of this custom implementation of Psend is shown in Figure 6, compared to the current implementation of Open MPI. For the *left-to-right-* and *neighbourhood exchange*-patterns (Fig. 6a), the high bandwidth of larger partitions is kept, regardless of the application-provided partitioning. For the *randomized* pattern (Fig. 6b), however, a drop in bandwidth can be seen for larger partitions. This can be explained by the fact that for random marking of

user-partitions, each internal partition can be expected to become ready within the last few insertions. Therefore, the first transfer can be expected to happen far later than for linear insertion patterns.

Note that the benchmark does not simulate computations on the send side and as a result, the Pready-calls happen in close succession. For actual applications, the calls can occur further distributed in time, possibly increasing the influence of the sending pattern on the effective bandwidth. For this reason, it might be necessary to implement more sophisticated methods of message aggregation in such cases.



(a) left-to-right pattern (b) random pattern

Fig. 6: Bandwidths for the custom Psend-implementation with different patterns to mark partitions ready

6 Conclusion

In this paper, we evaluate and improve on the current state of the partitioned communication interface in MPI implementations. Therefore we present a synthetic benchmark that compares partitioned communication to alternative communication patterns in a hybrid MPI+thread context using different schemes to mark partitions as ready for transfer. Our findings show, that the implementation in Open MPI already can achieve better performance with the new partitioned interface compared to the alternative patterns. However, the implementation has some issues with small partitions, i.e., large numbers of partitions. Therefore, we present an optimization based on an aggregation scheme for the partitions. This optimization shows clear benefits, especially for schemes where partitions are marked ready in a sequential ordering.

7 Outlook

The benchmark that we developed for MPI partitioned communication allows us to test quickly the performance of MPI implementations with different communication patterns. We plan to extend our performance tests to MPICH as well as other MPI libraries and compare our findings with other recent studies [2]. Based on our prototype implementation for the aggregation of partitions, we intend to come up with a native implementation in Open MPI.

Acknowledgments. This study was performed within the context of the 3xa project that received funding from the Federal Ministry of Education and Research, grant number 16ME0654.

References

1. Dosanjh, M.G., Worley, A., Schafer, D., Soundararajan, P., Ghafoor, S., Skjellum, A., Bangalore, P.V., Grant, R.E.: Implementation and evaluation of MPI 4.0 partitioned communication libraries. Parallel Computing **108** (2021)
2. Gillis, T., Raffenetti, K., Zhou, H., Guo, Y., Thakur, R.: Quantifying the performance benefits of partitioned communication in MPI. In: Proceedings of the 52nd International Conference on Parallel Processing. ICPP 2023, ACM (Aug 2023)
3. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.1 (Nov 2023), https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf
4. Open MPI, https://www.open-mpi.org/
5. TOP500 List (June 2024). https://top500.org/lists/top500/2024/06/, accessed: 2024-08-02