Validation of OpenMPI transferred data using Merkle Trees

Pavel Kratochvíl
Brno University of Technology
Faculty of Information Technology
xkrato61@vutbr.cz

***Abstract*—This paper introduces an efficient protocol for validating data transfers and correcting errors to ensure data integrity in distributed systems. The proposed implementation adds validation layers to Message Passing Interface (MPI) function wrappers, which use hash-based verification and enable rapid error correction with minimal increase in transmission size. Any data discrepancies are detected by comparing computed hashes between sender and receiver ranks. Merkle trees are employed to ensure optimal execution in high-performance computing (HPC) environments, making it possible to identify and retransmit faulty data blocks in logarithmic time, preserving the system's overall efficiency.**

***Index Terms*—MPI, Merkle trees, data validation, OpenMP**

I. Introduction

The growth in computational power has allowed us to solve increasingly complex problems. Solving tasks such as drug discovery, protein folding, or climate modeling requires massively parallel systems—many individual computers that work in a precise and coordinated manner on a single task. Many tools enable such critical coordination, which involves synchronization, facilitating data exchange, and managing parallel process communication across distributed systems. The de facto standard in the High Performance Computing (HPC) domain is Message Passing Interface (MPI).

A. MPI

MPI defines the syntax and semantics of library routines a programmer calls from C, C++, or Fortran. As multicore systems have become ubiquitous, HPC applications usually run on clusters of multicore nodes. In current MPI distributions such as OpenMPI, MPICH, or IntelMPI, each MPI process is mapped as an OS process. Then, an MPI communicator is a collection of processes that can send messages to each other. A member process in a particular communicator is uniquely identifiable by its rank. Depending on the mapping of processes on individual cores on one or multiple physical machines, communication between them can be either intranode or internode[3].

MPI can implement several protocols for reliable communication between ranks. While a shared memory of the machine can be used for intra-node message passing, a protocol such as TCP/IP, Infiniband, or RoCE (RDMA over Converged Ethernet) is used for inter-node communication.

B. Error detection in transmission protocols

All of the communication protocols mentioned employ some error detection mechanism. For example, TCP/IP and Infiniband typically include checksums or the Cyclic Redundancy Check (CRC). Furthermore, each MPI message has a tag that allows the receiver rank to effectively filter out irrelevant messages, thus receiving only the data it was supposed to receive. Arguments for the transmission routines include the count and type of data transmitted. The types can be simple, like `MPI_INT` or `MPI_DOUBLE`, or, if needed, MPI offers functions (e.g., `MPI_Type_create_vector` to detect invalid transmission during development, and mechanisms such as including checksums or CRC codes into the transmission then ensure data validity during runtime. However, some rare scenarios could still lead to errors.

1) Hardware malfunctions and memory corruption: Memory corruption or other hardware malfunctions, such as errors in the network interface card (NIC), could cause incorrect data to be sent or received. The former is preventable to some extent by using a type of computer memory that uses error correction codes (ECC). Such memory is a standard in machines used in the HPC domain. It attempts to detect and correct n-bit data corruption in the main memory caused by background radiations, namely neutrons and cosmic ray secondaries[5].

2) Race Conditions and Buffer Overflows: Poorly managed concurrent access to shared resources in code, which is out of MPI's control, could lead to invalid memory accesses or buffer overruns, resulting in corrupted transmitted data.

3) Implementations Bugs: Although rare, errors in specific implementations of MPI libraries could introduce data transmission problems. In addition, these tend to manifest in particular edge cases and can hardly be tested in the software development phase.

II. Validation using Merkle trees

Merkle trees, a cryptographic data structure first proposed by Ralph Merkle's thesis in 1979, play a pivotal role in this paper. They offer an efficient method for validating data integrity in distributed systems and blockchain technologies, making them particularly valuable for data validation in high-performance computing environments and large-scale data transfers.

In comparison with more straightforward approaches to data validation in one or several blocks, where a hash would be generated for the entire data segment or several data blocks, Merkle trees offer a balance between an added computational complexity trade-off and a logarithmic verification complexity of O(log n) operations for a single block. This advantage makes them particularly well-suited for large datasets and distributed systems. The structure's effectiveness can be further enhanced using effective hash functions such as XXH3, which provides computation of data segment hashes at near-RAM speed, further lowering added costs to high-performance applications[1]. Merkle trees enable rapid error detection and correction, allowing faulty data blocks to be identified and retransmitted in logarithmic time, preserving overall system efficiency.

A. *Implementations in Existing Systems*

Blockchain systems and several widely used tools have adopted Merkle trees for data validation and consistency checking, demonstrating the broad applicability of this solution. For example, no-SQL distributed database systems such as Apache Cassandra and Amazon DynamoDB utilize Merkle trees to detect inconsistencies between data replicas, ensuring data integrity in distributed environments[2][4]. Furthermore, the version control system utilizes them to manage distributed projects, facilitating efficient comparisons and updates of repository states. Google's Cloud Storage has implemented automatic client-side validation using checksums for data transfer verification, highlighting the versatility of hash-based verification methods in cloud computing scenarios.

B. *Merkle Tree Structures for Efficient Segment Validation*

Merkle trees store data hashes and subsequent hashes of hashes in inner nodes to achieve low memory overhead and minimal performance cost added to HPC MPI applications. For data consisting of segments $Y_1, Y_2, ..., Y_n$ each leaf node corresponds to a data segment Y_i, storing the pointer to the segment, its length (up to `data_size/segment_size`) long, hash of the segment $H(Y_i)$ and node id. An inner node then stores the subsequent hashes of the descendants' hashes $H(Y_i) = H(H(Y_{2i+1}), H(Y_{2i+2}))$. The data segment size covered by a leaf node is configurable during runtime.

Upon receiving the data from the sender and constructing a Merkle tree structure, validity can be ensured by simply comparing the root hashes on both sides. If the root hash matches, the data segments are identical on both sides; otherwise, at least one data segment is faulty.

C. *Critical Paths*

A comparison of the root hash is sufficient for fault detection in an arbitrary data segment. However, some hashes of the inner nodes are needed to detect the faulty data segment. The smallest set of hashes necessary for validating any segment contains the tree root hash and the hash of every left descendant (highlighted in red in Fig. 2). This set of critical paths hashes is shared between the sender and receiver with the data.

Although the set of hashes shared between sender and receiver is not negligible (the size equals the number of leaf nodes), it allows the receiver to locate the faulty segment in logarithmic time. Only the faulty segment is retransmitted to repair the data transferred. The hashes are then recalculated from the updated data and the root hash is compared again. If there are several faulty segments, this process must be repeated until the root hash does match.

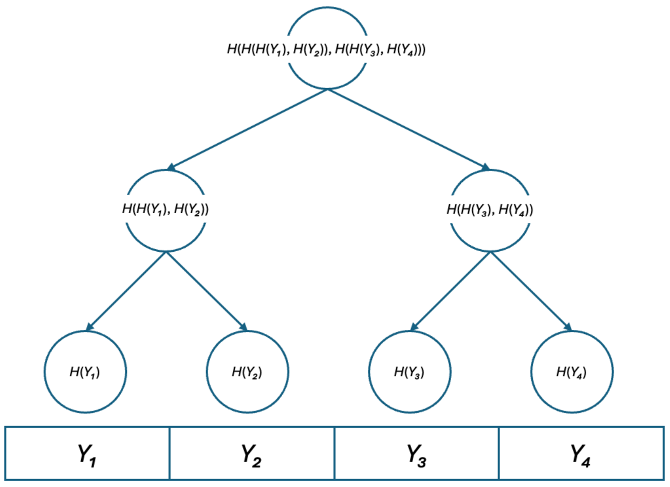

Fig. 1. Merkle tree structure on data of four segments.

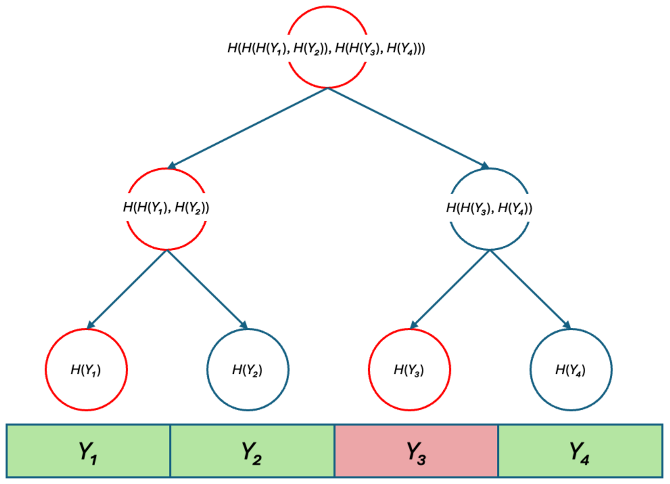

Fig. 2. Merkle tree structure with a highlighted set of critical paths hashes. The segment Y_3 is faulty; thus, the fault is detected after comparing the root hash. Descending the tree and comparing each hash from the received critical paths hashes, the fault can be pinpointed to the one corrupt segment—Y_3.

D. *Hash Function in Merkle tree*

The XXH3 hash algorithm is used to compute the hashes for the nodes in a Merkle tree structure. XXH3, a 64-bit variant of the xxHash family, is chosen because of its high speed, low latency, and ability to produce uniformly distributed 64-bit hash values. These properties make XXH3 particularly suitable for high-throughput and rapid hashing scenarios, such as high-performance computing (HPC) and real-time data verification.

XXH3 achieves exceptional performance through vector-friendly inner loops, optimized access patterns for small inputs, and intrinsic instruction selection that maintains efficiency even on 32-bit systems. For use in Merkle trees, XXH3 is particularly advantageous due to its high throughput on small inputs (20-30 bytes) and variable-length data while maintaining strong avalanche properties and collision resistance[1].

This efficiency translates well in parallelized environments, which is valuable for HPC tasks requiring rapid data validation. The algorithm is widely adopted for applications such as data deduplication, real-time analytics, and distributed storage systems. Additionally, XXH64's support for SIMD instructions enhances its compatibility with vectorized processing, a frequent requirement in HPC, further demonstrating its value as an ideal hashing algorithm for this Merkle tree-based data integrity solution.

III. Implementation

Among languages supported by OpenMPI and OpenMP (Fortran, C, C++), the C language standard C11 (ISO/IEC 9899: 2011) was chosen for the implementation because of its wide usage in the HPC domain and its interoperability with C++ projects. The final library contains wrappers for OpenMPI functions `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, `MPI_Allgather`, additional pure validation functions, and the underlying core functionality for tree construction and validation. The wrappers are named per the MPI standard but have `_Merkle` suffix (e.g. MPI_Send_Mekle). Pure validation functions can be used for post hoc validation and correction. The core library functionality includes tree creation and deallocation, recursive hash comparison across the entire tree, critical paths' hashes retrieval, and recalculation of hashes. The implementation details will be briefly described in the following sections.

The final library is publicly available on GitHub[1].

A. Merkle tree creation

To create a Merkle tree data structure for data validation between the sender and the receiver, a `create_merkel_tree` function was implemented. It takes a pointer to the data, the data size in bytes, and the segment size. It effectively partitions the data into segments after iteratively creating the binary tree structure of `merkel_node` structure. The segment size is configurable but defaults to $2048\,\mathrm{B}$. Furthermore, the segments (leaves) count is then `(data_size + segment_size - 1) / segment_size`. Next, the `struct merkle_node` array, which contains both leaf and inner nodes of size $2 * leaf\_count - 1$, is allocated.

The optimal segment size depends on many factors, such as CPU core count, the number of OpenMP threads (set via the `OMP_NUM_THREADS` environmental variable, and the total data size in transmission. Generally, the greater the data segment size, the faster the transmission. Additionally, setting the segment size to the total size of transmitted data degrades the Merkle tree to a single leaf node, which might be helpful in some cases for a swift data validation using only one hash for the entire transmitted data.

[1]https://github.com/raspbeep/merkle-mpi

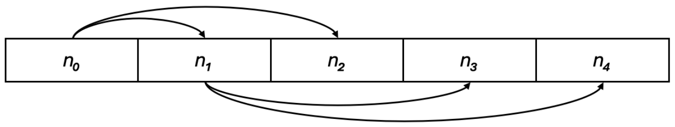

Fig. 3. Merkle tree creation example for 3 data (leaf) nodes. Five node structures in total need to be allocated ($2 * leaf\_count - 1$).

The tree creation function creates the binary tree structure in a lever order fashion where each node with index i has index $2i+1$ and $2i+2$. The node with index i is a leaf node if and only if the indexes $2i+1$ and $2i+2$ do not exceed the total number of nodes. Each leaf node n_i corresponds to precisely one data segment with index $i-(n\_nodes-n\_leaves)$, where n_nodes is the total number of nodes and n_leaves is the total number of leaves.

In cases where the number of data segments is not a power of two, a slightly unbalanced binary tree is constructed. The approach in such cases would involve padding the data to the nearest size, a power of two, or expanding the tree so that surplus leaves would point to the same last data segment. Both of these approaches turned out to be less efficient. In both cases, additional computational cost would be added; in the former case, additional memory would also be used.

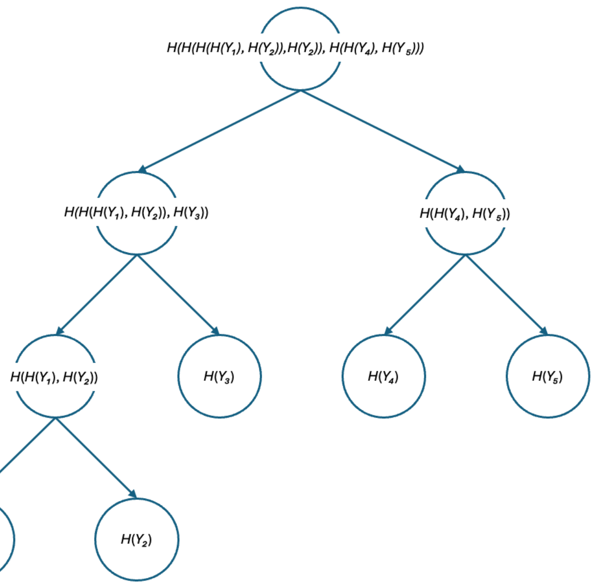
$H(H(H(H(Y_1), H(Y_2)),H(Y_2)), H(H(Y_4), H(Y_5)))$
$H(H(H(Y_1), H(Y_2)), H(Y_3))$
$H(H(Y_4), H(Y_5))$
$H(H(Y_1), H(Y_2))$
$H(Y_3)$
$H(Y_4)$
$H(Y_5)$
$H(Y_1)$
$H(Y_2)$

Fig. 4. An example of a Merkle tree data structure for data segment count different from a power of two. The tree is slightly unbalanced.

B. Merkle tree node hash calculation

After an iterative tree creation, the hashes of the leaf and inner nodes are calculated by traversing the tree and using the `XXH3()` function from the xxhash library in postorder. Thus, each leaf node contains a hash of its data segment, and each inner node i contains the hash of its two descendants at index

$2i+1$ and $2i+2$. Since the XXH3() requires a $64\,\mathrm{bit}$ value as the seed for the hash calculation, the node index in the tree is used. Therefore, the hash values are unique even if two leaf nodes have identical data segments.

Moreover, to accelerate this process, OpenMP tasks are used for the recursive postorder traversal and hash calculation. Upon descent into a non-NULL node, a new task is created, which descends into the left and right descendant, effectively calculating their hashes (from the data segments or their descendants' hashes). Then, the task calculates the hash of the current node. Using OpenMP for the hash calculation helps to saturate all available CPU resources. However, the number of threads has to be limited (using OMP_NUM_THREADS) to avoid overwhelming the CPU resources when many ranks construct larger Merkle tree structures.

C. Data Transmission and Validation Protocol

Since the provided wrapper functions in the final library have the same signature as the MPI standard defines, no other changes are necessary when using the library in an existing code base other than replacing the functions' names.

The transmission of data with Merkle tree validation has the following steps:

1) Data are transmitted between the sender and the receiver.
2) Merkle tree data structures are constructed on both sides.
3) Critical paths' hashes are transmitted to the receiver.
4) The receiver hierarchically compares the hashes of their tree with the received ones.
5) If no faulty data is detected (the root hash matches), a no-correction-needed signal is transmitted back to the sender. Otherwise, a faulty data segment index is sent.
6) The receiver awaits confirmation or a faulty data segment index from the receiver.
7) If the sender receives a faulty segment index, it retransmits only the data segment at the received faulty index. The receiver then recalculates the hashes and returns to Step 4. Otherwise, the transmission is complete once the sender receives a successful confirmation from the receiver.

For collective transmission functions (MPI_Allgather or MPI_Bcast, confirmations of correct data reception or the faulty node index from the receiver are collected using MPI_Gather. The correction steps are then executed with every receiving node sequentially. For example, if nodes one and two received faulty data at index one, they sent this index to the sender. The sender follows the data transmission and validation protocol from step 7 with node one and then with node two until all the data are confirmed to be correct from all the receiving nodes.

This sequential approach was chosen for performance reasons, as collective communication with multiple nodes receiving faulty data would require creating a new MPI communicator, which can be costly.

D. Standalone Data Validation

The MPI wrapper functions provided in the final proposed library only cover some use cases, as their number is minimal, partly due to the asynchronous nature of several MPI functions and the synchronous nature of the Merkle tree validation between the sender and the receiver. For asynchronous communication in MPI, a communication request handle is passed to the function, which is later waited for by using functions such as MPI_Wait or MPI_Waitall or MPI_Waitsome. This feature of asynchronous communication in MPI enables the overlay of computation and communication and, therefore, shortens the total execution time. However, this is contradicting what the validation with Merkle trees is trying to achieve since the data must be validated upon transmission. Any corrupted data segment must be retransmitted, and the critical paths' hashes recalculated and confirmed to the sender.

A solution to this problem provided by the final library is using functions for standalone data validation without tying them to a particular MPI function in a wrapper. This standalone validation is provided by functions validate_sender_merkle and validate_receiver_merkle. These can be used post hoc after the data are confirmed to be transmitted (synchronously or asynchronously) and still offer many benefits for data validation, such as logarithmic-time detection of faulty data segment and the reduced cost of retransmitting only the faulty segment instead of the entire payload.

E. Simulation of Corruption

Data corruption was simulated during development by alternating the received values in the received data to ensure the correct functioning of the final library. This corruption can be enabled with SIMULATE_CORRUPTION defined in src/mpi_interface.c.

IV. Benchmarking and Performance Analysis

A compute node with AMD EPYC 7H12 64-Core CPU and $250\,\mathrm{GB}$ RAM on the Karolina supercomputer was used for performance analysis. Performance was measured on a single compute node with a variable transmitted data size of $128\,\mathrm{B}$ up to $268\,\mathrm{MB}$. For collective communication (MPI_Bcast and MPI_Allgather), the number of ranks is also variable and ranges from 2 to 32 or 64. Furthermore, the given benchmark test for a given operation, data size, and number of ranks in communicator, the operation is repeated 100 times to produce tangible time results even for small data sizes. Lastly, all benchmark tests were done without artificial data corruption.

The test executable test_merkle can be built using instructions in the provided readme file. All of the presented results were produced launching the test_merkle executable in different configuration using the benchmark.sh script.

A. Send and Receive

A performance comparison between standard MPI Send-Receive operations and their Merkle tree-enhanced counterparts across different buffer sizes ranging from $128\,\mathrm{B}$ to

268 MB is visible in Fig. 5. The standard Send-Receive operation (shown in blue) consistently performs fastest. In contrast, the Merkle tree implementations show varying overhead levels depending on their item count (segment size), a crucial parameter during tree construction. The Merkle variants with larger item counts (2048-4096) perform better than those with more minor counts (16-32), as more significant segments reduce the total number of hash calculations and tree nodes required. All implementations show a generally linear scaling pattern on the log-log plot, with the performance gap between standard and Merkle versions remaining relatively consistent across buffer sizes.

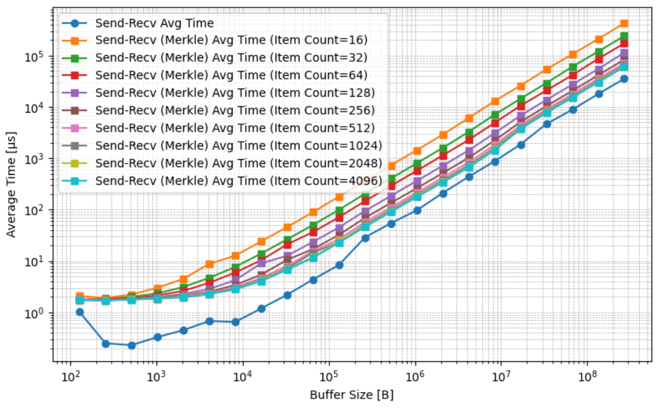

Fig. 5. Benchmark results of `MPI_Send` (bottom blue line) and `MPI_Send_Merkle` (upper lines) with varying leaf data segment size.

B. Broadcast

The 16-, 32- and 64-rank broadcast operation shows a more pronounced performance impact when using Merkle trees, with transmission times approximately 10-100 times higher than standard `MPI_Bcast` (below 1 MB). In particular, Merkle variants maintain relatively constant transmission times for small buffer sizes before showing increased transmission times with larger buffers. Part of the figure below 1 MB clearly illustrates the performance cost of constructing the Merkle tree, but with increasing buffer size, the time of transmission starts to take more and more of the execution time, which makes the added cost of the Merkle tree construction less significant.

C. All gather

In the benchmarking of `MPI_Allgather` and `MPI_Allgather_Merkle`, the maximum buffer size was reduced to 4 194 304 B due to functions requiring a receive buffer in all ranks, which has `buffer_size*n_ranks` size. As expected, these testing functions have the longest execution time and are highly memory-bound.

V. Limitations

A critical limitation of the final library is its inability to employ data validation directly in asynchronous communication. For example, MPI functions `MPI_Isend` and its complement `MPI_Irecv` can hide nonnegligible latency and allow the nodes to perform a different computation while communication occurs. Subsequently, when the transmission data is needed, a `MPI_Wait` (or similar) function is used to await completion. However, for this use case, an alternative of validating the data separately from the MPI transmission function can be utilized using the using `validate_sender_merkle` and `validate_receiver_merkle`. These can ensure data validity even when overlaying computation and communication is desired.

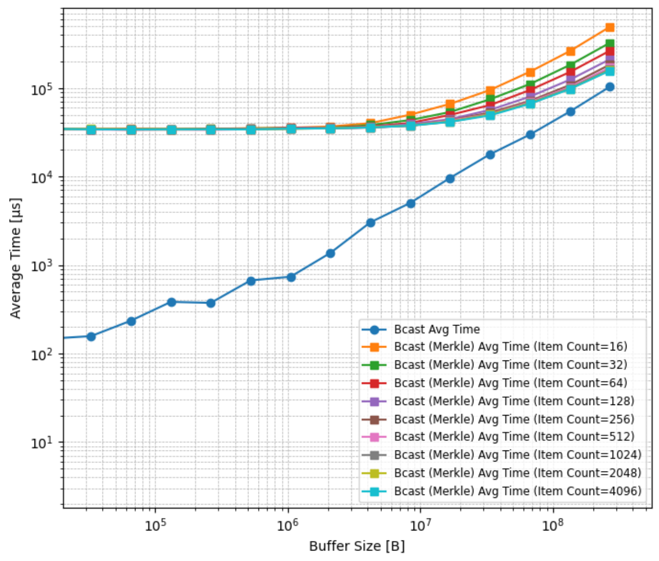

Fig. 6. Benchmark results of `MPI_Bcast` (bottom blue line) and `MPI_Bcast` (upper lines) with varying leaf data segment size and 16 ranks.

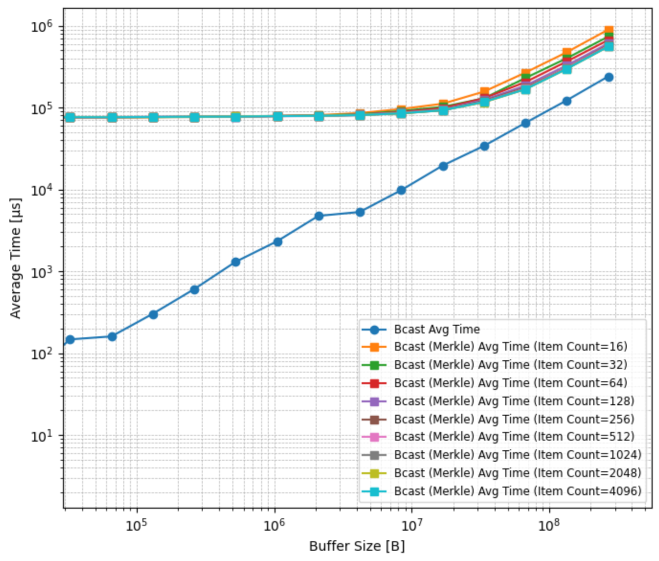

Fig. 7. Benchmark results of `MPI_Bcast` (bottom blue line) and `MPI_Bcast` (upper lines) with varying leaf data segment size and 32 ranks.

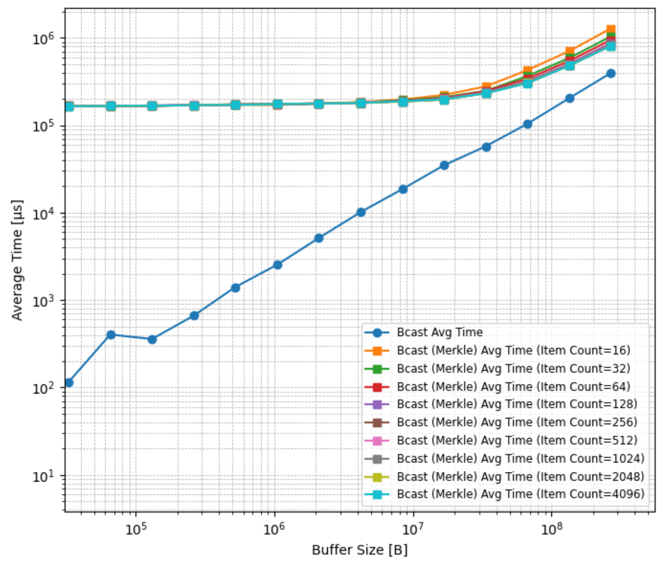

Fig. 8. Benchmark results of `MPI_Bcast` (bottom blue line) and `MPI_Bcast` (upper lines) with varying leaf data segment size and 64 ranks.

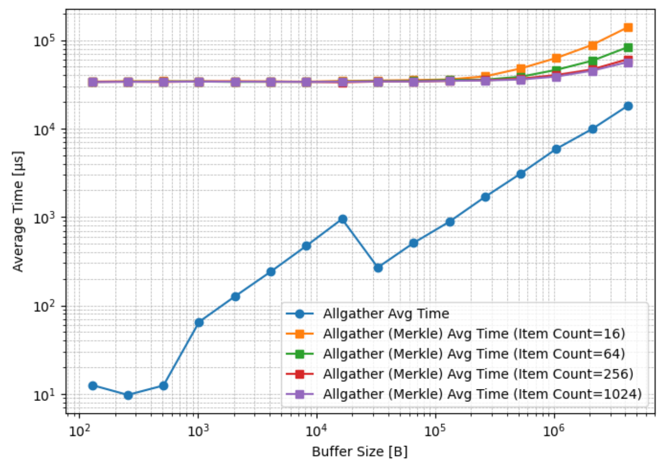

Fig. 9. Benchmark results of `MPI_Allgather` (bottom blue line) and `MPI_Allgather_Merkle` (upper lines) with varying leaf data segment size and 16 ranks.

VI. Conclusion

Implementing Merkle tree-based validation for OpenMPI data transfers provides a robust solution for ensuring data integrity in HPC environments. The key findings demonstrate several essential aspects: The validation system successfully detects and corrects data transmission errors through an efficient protocol that only requires retransmission of corrupted segments identified in logarithmic time. XXH3 as the hashing algorithm proves particularly effective thanks to its high throughput and optimization for small inputs, making it well-suited for HPC applications. Performance benchmarking reveals that while the Merkle tree validation does introduce overhead, the impact becomes less significant as data sizes increase. For basic Send-Receive operations, larger segment

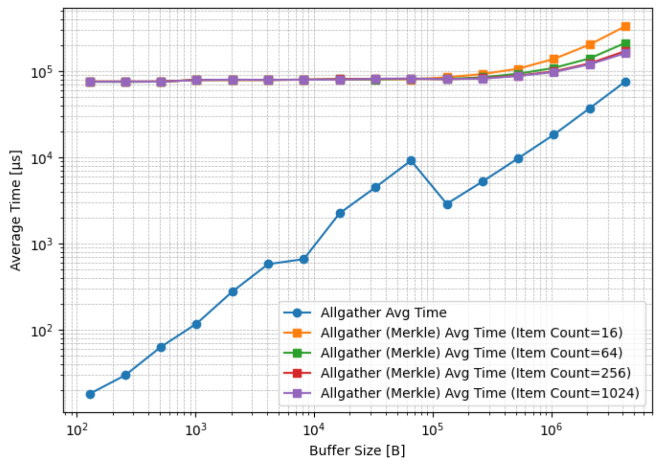

Fig. 10. Benchmark results of `MPI_Allgather` (bottom blue line) and `MPI_Allgather_Merkle` (upper lines) with varying leaf data segment size and 32 ranks.

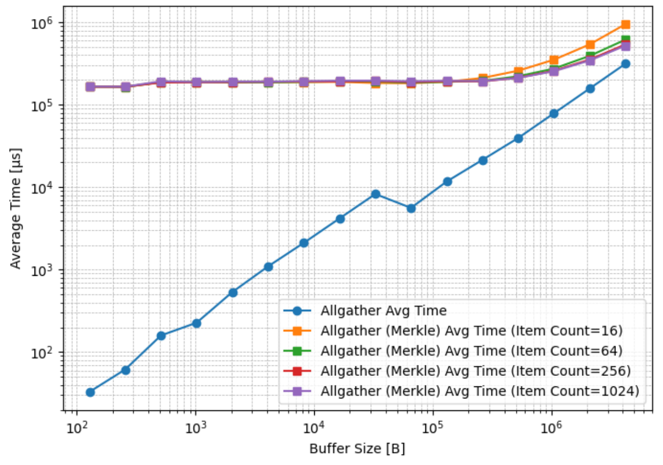

Fig. 11. Benchmark results of `MPI_Allgather` (bottom blue line) and `MPI_Allgather_Merkle` (upper lines) with varying leaf data segment size and 64 ranks.

sizes (2048-4096 bytes) perform notably better than smaller ones (16-32 bytes). In collective operations like Broadcast and Allgather, the performance impact is more pronounced, showing 10-100x higher transmission times for smaller buffer sizes. However, this overhead becomes proportionally less significant with larger data transfers. A notable limitation exists regarding asynchronous communication, where the synchronous nature of Merkle tree validation conflicts with MPI's asynchronous operations. However, this is mitigated through standalone validation functions that can be used post-transmission. The implementation successfully balances security with performance, offering a practical solution for applications where data integrity is crucial while maintaining reasonable performance characteristics for large-scale data transfers in HPC environments.

References

[1] Cyan. *Presenting XXH3*. Mar. 2019. URL: https://fastcompression.blogspot.com/2019/03/presenting-xxh3.html.

[2] Dennis Huo and Julien Phalip. *New file checksum feature lets you validate data transfers between HDFS and Cloud Storage*. Mar. 2019. URL: https://cloud.google.com/blog/products/storage-data-transfer/new-file-checksum-feature-lets-you-validate-data-transfers-between-hdfs-and-cloud-storage.

[3] Zhiqiang Liu, Junqiang Song, and Shaoliang Peng. "MPIActor: A thread-based MPI program accelerator". In: *2010 IEEE 18th International Workshop on Quality of Service (IWQoS)*. 2010, pp. 1–2. DOI: 10.1109/IWQoS.2010.5542703.

[4] Nishant Neeraj. *Mastering Apache Cassandra*. PACKT Publishing, Sept. 2013, pp. 44, 63. ISBN: 9781782162681.

[5] E. Normand. "Single event upset at ground level". In: *IEEE Transactions on Nuclear Science* 43.6 (1996), pp. 2742–2750. DOI: 10.1109/23.556861.