On the Potential of Compression Hiding in MPI Applications

Yicheng Li[1] and Michael Jantz[2]

[1] University of Tennessee, Knoxville, Knoxville TN 37996, USA
yli137@vols.utk.edu
[2] University of Tennessee, Knoxville, Knoxville TN 37996, USA
mrjantz@utk.edu

Abstract. The increasing disparity between computing capabilities and communication bandwidth has become a major bottleneck in High Performance Computing (HPC) applications. To address this challenge, we introduce a framework that leverages early data compression for communication data within the Open MPI library with the use of userfaultfd (uffd) for efficient write detection. By integrating the high-speed LZ4 compression algorithm, the proposed framework minimizes communication overhead by reducing the size of data transmitted among processes while hiding compression overhead behind either pack or communication overhead. Applying our uffd framework onto Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) highlights the potential of the framework in reducing data communication volumes and overall communication latency, paving the way for improved performance in HPC environments.

Keywords: HPC · MPI · Open MPI · Userfaultfd

1 Introduction

The rapid advancement of computational capabilities has significantly outpaced the improvements in communication speed, creating a notable imbalance that often hinders the performance of many scientific applications. This growing disparity between computing power and the ability to swiftly transfer data across systems poses a considerable bottleneck, affecting not only the efficiency but also the scalability of complex computations. In the realm of HPC, where vast amounts of data are processed and exchanged, the impact of this imbalance is especially pronounced.

To combat this challenge, the strategy of minimizing the volume of data that needs to be exchanged emerges as a particularly effective solution. By reducing the amount of data transmitted, it is possible to alleviate the pressure on communication channels, thus enhancing the overall performance of the application.

In this context, the implementation of data compression techniques within MPI communication protocols offers a promising avenue for mitigating the disparities between computational speed and communication efficiency. By com-

pressing data before transmission and decompressing it upon receipt, it is possible to significantly decrease the size of the data being exchanged. This reduction in data volume directly translates to reduced communication time, effectively narrowing the gap between computing and communication speeds.

Many prior works have employed compression to reduce MPI communication costs. However, these works are mainly focused on how to compress application data, but do not consider the question of when and where to compress data. Existing approaches always compress MPI buffers immediately before sending the data to the Network Interface Card (NIC). Thus, they can miss opportunities to overlap compression with other useful application activity.

In this work, we investigate the potential of compressing application data early so that it can be overlapped with other application activity as it is being compressed within Open MPI [9] framework. First, we investigate the potential of hiding compression overhead by surveying several mini-applications/benchmarks in Section 4. Then we introduce the proposed framework with uffd in Section 5 and summarize the performance and future improvements in Section 6.

This work makes the following important contributions:

1. A comprehensive survey of existing and state-of-the-art techniques for incorporating compression into MPI communication, highlighting their benefits and limitations.
2. The design and implementation of a novel framework that offloads compression tasks to idle compute resources and strategically overlaps compression with communication by exploiting the gap between the last memory write and data transmission.
3. An in-depth investigation and practical application of the Linux's uffd mechanism for efficient write-fault detection and handling in support of compression-aware communication strategies.
4. A detailed analysis of the trade-offs among compression ratio, compression overhead, and communication performance, providing clear guidance on scenarios where compression delivers net benefits.

In summary, the application of compression in MPI communication represents a strategic response to the growing divide between the capabilities of modern computing hardware and the limitations of data transfer speeds. By intelligently managing the volume of data that must be communicated, it is possible to significantly improve the performance and efficiency of scientific applications, paving the way for the next generation of HPC breakthroughs.

2 Related Work and Motivation

Reducing the volume of communicated data is a well-established strategy to mitigate the communication bottleneck in HPC applications, particularly in MPI-based parallel programs. Data compression has emerged as a promising solution to this challenge by reducing the amount of data transmitted over the network, potentially leading to improved overall performance. However, to realize the

benefits of compression, it is crucial to carefully manage the trade-offs among three key factors: the compression ratio, the overhead associated with compression and decompression, and the resulting impact on communication latency. Several studies have explored integrating compression into MPI communication. Early efforts by Filgueira et al. [2–4] proposed adaptive compression strategies in collective MPI operations. These work dynamically select compression techniques based on runtime characteristics such as data size and system load. Shan et al. [12] investigated how lightweight compression algorithms could be leveraged to reduce message size in MPI_Alltoall operations, reporting measurable performance gains on large-scale systems.

A particularly promising direction involves integrating the compression and decompression processes into existing MPI communication or pack operations. By integrating compression seamlessly into MPI_Pack, prior work has demonstrated that it is possible to maintain portability while improving communication efficiency. For example, Zhou et al. [13] explored modifying the MPI datatype engine to insert compression at pack time, allowing compression to be performed just before the data are transferred, thus minimizing its perceived overhead. Similarly, Ramesh et al. [11] introduced a modular framework that allows transparent compression insertion within MPI libraries, focusing on preserving MPI semantics while optimizing runtime behavior.

Despite these advancements, challenges remain in selecting appropriate compression schemes for various data characteristics, and in managing the overheads associated with compression algorithms. Our work builds upon these previous studies by further hiding and offloading compression in between writes and communication. We focus specifically on offloading compression and utilize Linux's uffd interface for write-detect functionality to attemptively delay compression until the last memory modification before communication, thereby hiding compression latency within naturally occurring communication gaps.

The Linux uffd interface was originally introduced to enable user-space handling of page faults, offering mechanisms to trap and respond to page accesses, including reads, writes, and missing pages. This feature has since attracted attention in a variety of system-level research efforts due to its flexibility in controlling memory access behavior and enabling novel memory management policies. Gidra et al. [5] were among the first to demonstrate the utility of uffd in distributed shared memory systems, where they used it to implement fine-grained page migration and consistency management in the user space. Similarly, Peng et al. [10] and McFadden et al. [8] utilized uffd in the context of UMap, a user-space page-fault handling system designed to virtualize and remap large memory regions in HPC environments. Their work showcased the potential of uffd to abstract remote or slow memory devices behind traditional memory interfaces.

Despite these advances, the application of uffd to performance-critical workflows, especially in HPC applications, is still immature. Zussman et al. [14] investigated using custom page access monitoring via uffd to optimize data movement, but their work stopped short of tightly coupling such mechanisms with communication layers such as MPI. The integration of uffd into core application

logic, particularly for latency-sensitive use cases such as dynamically compressing data between the last memory write and an MPI communication, remains an underexplored area.

In this paper, we aim to bridge this gap by proposing a novel use of uffd to attemptively detect the last write to a memory region before communication. By leveraging write-detection semantics, we are inserting compression into the gap between the final write and subsequent data transfer, enabling compression to be performed asynchronously and hidden behind other computation or communication. To the best of our knowledge, this work represents a new and practical use of uffd for overlapping computation and communication with compression in MPI-based applications.

3 Experiment Setup

3.1 Platform

We conducted our experiments on a system equipped with an Intel Xeon Gold 6246R CPU featuring 16 physical cores and 32 hyper-threading cores with Debian GNu/Linux 11 (bullseye). Our framework, integrated with LULESH, was executed using 8 processes, supplemented by an additional 16 processes dedicated to handling write-faults and performing compression. The supplementary processes were assigned to separate physical cores, distinct from those assigned to the primary 8 processes. Since LULESH requires a cubic number of processes for execution, this configuration represents the maximal deployment feasible on the current system. All our tests are conducted using Open MPI version 4.1.7.

3.2 LZ4 Compression Ratio and Speed Performance

LZ4 [1] is a lossless compression algorithm known for its exceptionally fast compression and decompression speeds, making it highly suitable for performance-critical scenarios. We selected LZ4 for our compression/decompression algorithm because our uffd framework has the potential to effectively mask the compression overhead, while decompression overhead remains serialized with the main execution thread. Figure 1, in log scale for y axis, illustrates the compression and decompression speeds along with the compression ratios. In this test, we varied the percentage that represents the likelihood that the data sequences are identical. LZ4 excels specifically in compressing identical data sequences, thus, longer sequences of identical data lead to higher compression ratios and faster compression/decompression speeds.

To complement our evaluation among benchmark performances, we also examined the high-compression variant of LZ4, known as LZ4 HC. LZ4 HC trades off compression speed for improved compression ratios by employing a more exhaustive search algorithm to find longer and more optimal matches. Although this increases compression overhead, the decompression performance remains as fast as standard LZ4, preserving its suitability for our proposed framework.

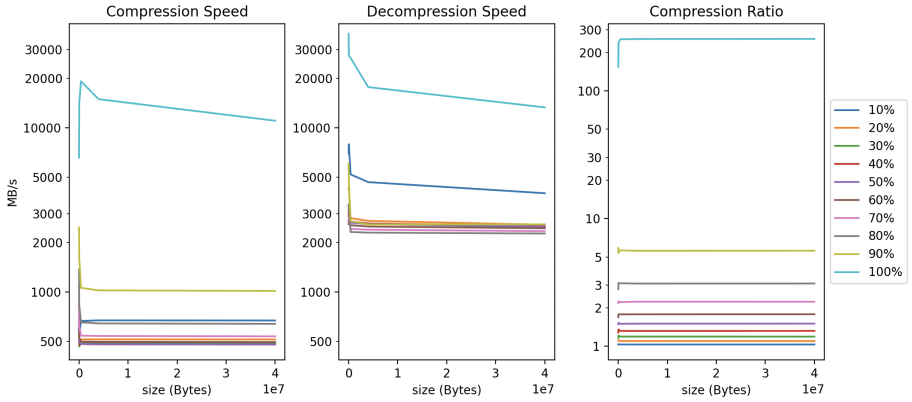

Fig. 1: LZ4 compression algorithm speed and compression ratio by percentage of the same consecutive data

3.3 Benchmarks

We selected four benchmarks from the CORAL-2 [6] benchmarks and the SPEChpc [7] benchmarks, together with LULESH, whose communication patterns closely resemble the use case for our framework. Table 1a presents the descriptions of the benchmarks, while Table 1b gives the full input command that we used for each benchmark. We then applied LZ4 and LZ4 HC compression algorithms to perform on-the-fly compression/decompression into specific parts of the point-to-point communication and extract measurements to see if our uffd framework would theoretically improve benchmark's runtime.

We then applied our framework onto LULESH for evaluation. LULESH is a highly simplified application developed by Lawrence Livermore National Laboratory. Through continuous development, LULESH has evolved into a widely analyzed proxy application in the DOE co-design efforts for Exascale computing. This application represents a step towards addressing the complex requirements of hydrodynamics modeling in computational simulations.

Running LULESH with Open MPI enables the application to leverage distributed computing environments effectively. To do this, LULESH requires the number of processes initiated by MPI to be a cube of any number, such as 1, 8, 27, 64, 216, and so forth. This constraint ensures that the spatial problem domain is partitioned into a collection of volumetric elements that can be distributed and processed efficiently across the computing nodes in a structured manner.

The default LULESH is computation bound. It separates computation from communication as there is no overlap to make computation more efficient since communication is only a negligible portion of the entire application runtime. We chose LULESH because of its communication pattern, where each iteration is using the same buffer with the same size. Because of LULESH's relatively small

code base, it stands out as one of the easiest applications to integrate and to test our framework.

Benchmark	Description
PENNANT	A mini-app from CORAL-2 modeling unstructured mesh physics, focusing on Lagrangian and radiation hydrodynamics. It serves as a compact proxy for large-scale multi-physics codes.
LBM	Simulates fluid dynamics using the Lattice Boltzmann Method. It models mesoscopic flow behavior and emphasizes parallel scalability and memory bandwidth.
SPH-EXA	Implements the Smoothed Particle Hydrodynamics method for fluid and solid dynamics. Designed to explore portability and scalability across HPC platforms.
Minisweep	Models sweep-based transport used in neutron and radiative transfer simulations. Captures communication-heavy patterns with directional dependencies.
LULESH	A proxy for shock hydrodynamics on unstructured meshes, modeling core compute patterns of typical hydrodynamics codes for performance benchmarking.

(a) Descriptions of Selected Mini-Apps

Benchmark	Test Command
PENNANT leblanc	mpirun -np 32 –bind-to hwthread –map-by hwthread ./build/pennant ./test/leblanc/leblanc.pnt
PENNANT leblancbig	mpirun -np 32 –bind-to hwthread –map-by hwthread ./build/pennant ./test/leblancbig/leblancbig.pnt
LBM	runhpc –config=config.cfg –action=ref 505.lbm_t
SPH-EXA	runhpc –config=config.cfg –action=ref 532.sph_exa_t
Minisweep	runhpc –config=config.cfg –action=ref 521.miniswp_t
LULESH	mpirun -np 8 –bind-to hwthread –map-by hwthread ./lulesh2.0 -s ${Problem Size} -i 50

(b) Test Commands for Selected Mini-Apps

Table 1: HPC benchmarks and corresponding test commands

4 Potential Compression Hiding for Applications

We ran most of the benchmarks from Table 1a with 32 ranks, bound, and mapped to hardware thread. Since each benchmark has a large code base, we tried our

best to insert on-the-fly compression into the communication. As MPI benchmarks, it is common to pack and unpack data before communication. However, for some benchmark, we are unable to identify the pack or the unpack. For benchmarks that uses non-blocking communication, we timed the entire communication from the first invoked communication function to the MPI_Wait or MPI_Waitall, thus individual overhead for send or receive is not timed. We represent the time of data-over-the-wire as "Total_Communication." We collect the entire duration of the communication, then subtract the serialized component (other attributes in Figure 2) to calculate the supposedly data-over-the-wire duration.

Most of the benchmarks in Figure 2 contain one non-blocking MPI communication from the sender and the receiver or both the sender and the receiver are using blocking MPI communication. We ignored communication that is completely non-blocking for easier compression insertion, thus the number we present in Figure 2 only reflects a portion of the total communication for some benchmarks.

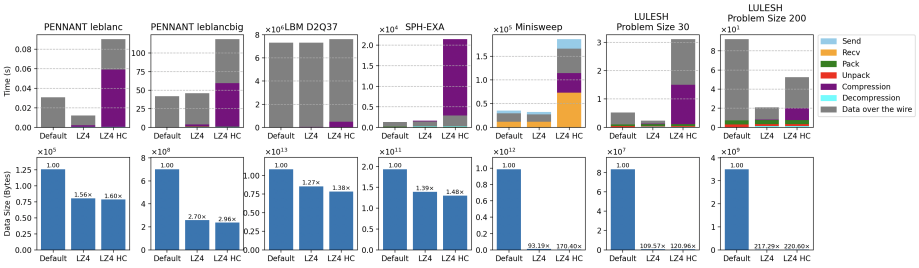

Fig. 2: Aggregate time comparison, data size comparison and compression Ratio for different benchmarks, LZ4 and LZ4 HC on-the-fly compression applied

The benchmarks shown in Figure 2 achieve moderate compression ratios on communication data, except for Minisweep and LULESH, however, the effectiveness varies significantly. Although some benchmarks achieve reasonable compression ratios, their overall performance remains suboptimal. As demonstrated by our tests with LZ4 compression in Section 3.2, the data from these benchmarks may not be ideally suited for LZ4 compression, as it typically achieves high compression ratios only when the data contain extensive sequences of identical values.

In case of suitable data for LZ4, data in Minisweep give a 93 compression ratio. However, the improved runtime is only 1.091x compared to the default run as Minisweep stresses more on network latency instead of total throughput. For each send in Minisweep, the size of the data per communication only goes up to 8MB. In comparison, each MPI_Send in LBM can put 1GB over the wire.

In addition to the compression ratio, excessive compression overhead would also result in significant performance degradation. Using the higher-overhead

compression algorithm, LZ4 HC, results in compression overhead dominating the communication process, causing the receiver to stall and significantly increasing the overall communication time for on-the-fly approach. However, some benchmarks are more tolerable for higher compression overhead, as their communication overhead is much higher, which gives an opportunity to hide compression overhead behind communication.

Our objective in this paper is to introduce an efficient approach to both benefit from reduced-size communication and hide the potentially high compression overhead.

5 Approach for Automated Compression Hiding

In the realm of high-performance computing, the Open MPI represents a cornerstone for the implementation of the MPI standard. This standard facilitates communication among processes that perform a parallel task, typically across a variety of hardware and network configurations. One of the most common approaches in HPC applications to exchange data is to use MPI point-to-point communication. MPI point-to-point communication refers to the direct exchange of messages between two specific processes in a parallel computing environment. It involves a sender and a receiver, where the sender uses functions like MPI_Send to transmit data, and the receiver uses functions like MPI_Recv to accept it. This form of communication is fundamental in MPI as it provides precise control over how data is exchanged, making it suitable for fine-grained synchronization and data exchange. MPI point-to-point operations can be either blocking or non-blocking, allowing flexibility depending on the communication and computation overlap requirements. These operations are essential for building more complex communication patterns in HPC applications.

In a typical MPI point-to-point communication, developers can either let MPI datatype engine handle non-contiguous data transfer or they can pack data explicitly into contiguous form by hand. Some applications would choose to use non-blocking communication to hide pack overhead behind network latency. In this workflow, the common way to apply compression is to compress data right before the communication and send the compressed counterpart, and on the receiver side, decompress the received buffer and copy it into the user buffer. Such approach does not take into the account of compression overhead, as in Section 4, the results show that the benefit from exchanging reduced-size data could be offset by the compression and decompression overhead. In our uffd framework, we aim to hide compression overhead by offloading compression onto idle computing resources while trying to compress data early by detecting the last write-fault before communication.

Figure 3 presents a sophisticated design tailored for augmenting the Open MPI framework with a compression-offloading mechanism that operates via a dedicated worker thread and another dedicated thread for detecting writes via uffd. It also presents the data structures employed behind the scenes.

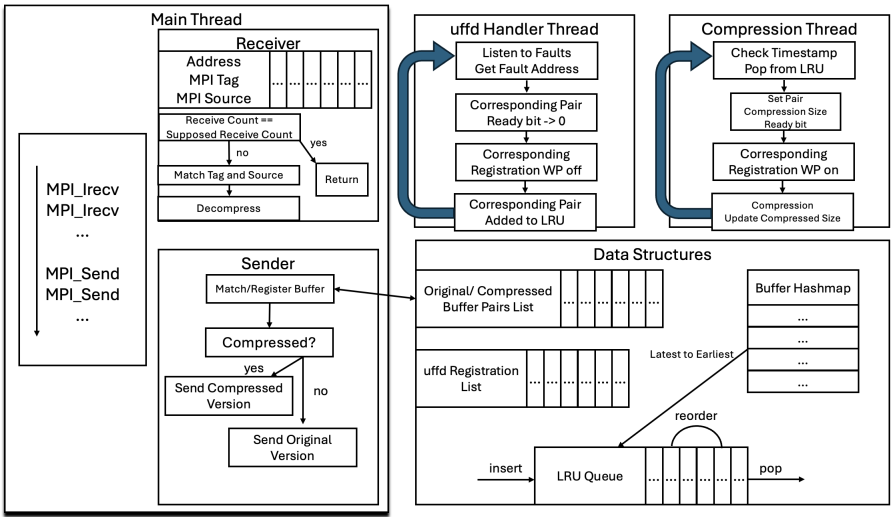

Fig. 3: Thread compression design for the Open MPI

5.1 Data Structures

The designed system incorporates tracking pairs of the original send buffers and their compressed counterparts, maintaining a Least Recently Used (LRU) queue to identify the earliest writes. It also includes a list of registered memory regions and a structure that stores receiving addresses, matching MPI tags and source ranks.

An array is used to store pairs of send buffers and their compressed counterparts. Initially, the array is allocated to hold 30 pairs, doubling in size whenever it reaches capacity. For applications like LULESH, 30 pairs are sufficient. Each pair contains additional tracking data, including the buffer's aligned page address, its aligned page size, and synchronization flags used for coordination among the main thread, the uffd handler thread, and the compression thread. When an send buffer is first encountered, a compressed buffer is allocated with a size equal to the original buffer plus an extra 100 bytes, ensuring compression always succeeds. The final compressed size is then recorded after a successful compression.

The LRU mechanism ensures efficient tracking of buffer usage while minimizing overhead. It includes a hashmap for fast buffer lookup and a double-linked list for finding the earliest write buffer. Given LULESH's communication pattern, the total number of possible inserted buffers is considerably low (9 entries per rank). When a buffer is being compressed, its entry is removed from the LRU. This removal involves deleting the corresponding node from the double-linked list and deleting its reference from the hashmap.

Each insertion or lookup operation begins by hashing the send buffer's address using the predefined hash function (buffer address mod its size mod 1024).

If the buffer is found in the hashmap, its corresponding node is moved to the front of the double-linked list, ensuring that it remains the most recently modified entry. Otherwise, a new node is created and inserted in the front.

Our first implementation of the LRU naively pops an item from the LRU queue and lets the compression thread perform compression on the buffer. However, if there are excessive writes going into one uffd registration, the back-and-forth write-protect on and off will limit LULESH's ability to advance main thread. Thus, our first implementation let the compression thread sleep for certain (1000) microseconds, before putting write-protect back onto the registration region. However, this approach is not as efficient as the node popped from the LRU queue could be written during the sleep. Our second approach adds a timestamp onto each of the nodes when queueing the send buffers into the LRU queue. It will check the current timestamp against the first node in the queue, if there is enough time (1000 microseconds) passed, then it will pop off the node and ask compression thread to perform compression.

The delay before popping a node from the LRU can vary across buffers, as some receive more frequent writes than others. In our experiments, we observed that if this delay is too short, the write instruction will not have enough time to be processed, thus it can prevent the main thread from advancing, ultimately causing the application to stall.

To optimize synchronization among the main thread, the uffd handler thread, and the compression thread, several mutex locks are utilized, preventing race conditions when updating the LRU structure. Additionally, since the number of tracked buffers is small, traversal and updates in the double-linked list remain computationally inexpensive. The combination of a hashmap and a double-linked list provides efficient O(1) lookup, insertion, and deletion, making the LRU implementation well-suited for handling the buffer management needs.

When MPI send function is called, uffd registrations are recorded to track buffer usage. The recorded information includes the buffer's original address and size, along with its aligned page address and aligned size, which ensure coverage from the buffer's start to its end. Since uffd operates at the page level, all registered addresses and sizes must be aligned with the page boundaries of the system.

The aligned page address is particularly important because fault addresses resulting from write protection are also aligned at the page level. If the buffer's starting address is unaligned, uffd registration will fail.

Furthermore, it is possible for data writes to occur outside the designated buffer but still within the aligned buffer region. While this does not impact correctness, it may result in additional buffers being queued for handling, particularly those that begin within the first page of the aligned buffer region. However, this behavior does not compromise data integrity, as only the relevant pages will be treated as dirty pages and queued in LRU for compression.

Since overlapping registrations are not allowed, every new buffer registration is checked against all existing registered entries to detect potential overlaps within the aligned buffer region. If an overlap is found, the system first removes

the write-protect from the affected region and unregisters the previous entry. The overlapping buffers are then merged into a single contiguous region, which is subsequently re-registered with uffd.

Merging ensures efficient management of the uffd mechanism. Additionally, merging minimizes the number of registered memory regions, reducing the overhead associated with tracking and handling multiple disjoint registrations. To maintain consistency, a mutex lock is used to prevent any uffd operations on the memory region in between registration and unregistration.

5.2 At uffd Registration

When send is called, the framework anticipates that the buffers passed to it will be reused frequently, though the actual data may change. After registration, it is important to clear the write-protect mode first, since the memory could have been used and written already, otherwise, the uffd registration will not work even though there is no error from setting write-protect. This step is unnecessary if, for certain, the to-be registered memory region has not been touched.

5.3 uffd Handler Thread

During MPI initialization, a dedicated uffd handler thread is spawned from the main process to manage write faults on registered buffers. Its primary role is to clear the write-protect on memory regions when writes occur.

When an incoming write triggers a page fault, the fault address is aligned to the system's page size. The handler thread checks this aligned fault address against the recorded page-aligned addresses to determine which registered buffer has been modified. Once identified, the handler proceeds to remove the write-protect from the corresponding memory region.

The main thread halts execution until the write-protect is cleared. Once the protection is removed, the handler thread queues the corresponding send buffer into the LRU queue, updating its access history with a timestamp in the double-linked list.

To minimize performance overhead, the uffd handler thread is designed to perform only essential operations, avoiding unnecessary operations, as its priority is to clear write-protect as quickly as possible, preventing delays in the main thread's execution.

5.4 Compression Thread

At MPI initialization, a dedicated compression thread is spawned alongside the main thread to handle data compression and reapply write-protect on registered memory regions. This thread operates asynchronously to ensure efficient memory management and apply early compression without blocking MPI communications.

The compression thread continuously monitors the LRU queue for available send buffers. When a buffer is detected, the compression thread will compare

the current timestamp to the timestamp attached to the first node in LRU, if enough time has passed, the first node is popped from the queue and processed for compression. Then, the compression algorithm is applied to the buffer, aiming to reduce its size while preserving data integrity.

If compression is successful and the resulting compressed buffer is smaller than the original, the compression thread sets the compressed size for the buffer, signaling that it is available for transmission in its compressed form. At this stage, the write-protect has already been re-applied before the start of compression to the registered memory region to ensure that subsequent writes will trigger the uffd mechanism again.

The compression thread operates in an endless loop, following these steps:

1. Check the LRU queue for an entry to compress.
2. Pop the entry from LRU (if enough time has passed) and locate its corresponding buffer pair.
3. Pre-set compression ready bit to be true and pre-set compressed size to be larger than the original buffer.
4. Apply write-protect to the corresponding uffd registration.
5. Perform compression on the buffer. The compressed size is recorded if compression is successful and the resulting data size is smaller than the original buffer size.

If a write occurs during compression, the ready bit is set to false. The compression thread, unaware of this change, still proceeds with compression. But since ready bit is first set to be true before the compression, the final check on the ready bit ensures that the send function will be notified if additional writes occurred even if there is a compressed buffer available.

5.5 Synchronization

Each registered memory region is associated with a unique uffd ID, which is created at the initialization of each MPI rank (MPI_Init or MPI_Init_thread). This unique ID establishes a strict one-to-one correspondence among the main thread, uffd handler thread, and compression thread, ensuring proper coordination. Multiple mutex locks are employed to maintain synchronization across different operations:

1. Buffer Pairs Array Lock – Ensures orderly operations upon creating, resizing, and accessing the buffer pairs array.
2. uffd Registration Lock – Protects against race conditions when unregistering, registering, or merging registered memory regions.
3. LRU Lock – Maintains orderly insertion, popping, and rearrangement of entries in the LRU queue.
4. Per-Buffer Pair Lock – Protects against race conditions when the compression thread modifies the compressed buffer size upon successful compression.

Mutex locks alone are insufficient to verify a valid compressed counterpart. Hence, a combination of a ready bit and compressed size ensures that send function always accesses an up-to-date compressed buffer.

6 Performance

In most scientific applications, non-contiguous data is packed into a contiguous buffer before communication to ensure efficiency, since sending the entire data is unnecessary and costly. LULESH is one of the applications that packs data before sending the packed buffer. In our framework, the buffer addresses are registered and recorded at communication, i.e. MPI_Send, MPI_Isend, while compression thread is running in the background compressing invalidated buffers. Thus, as LULESH packs buffer, the registered memory region will receive a write-fault, and then the uffd handler thread will proceed to invalidate all the recorded buffer pairs that reside within the same uffd registration region.

In our current implementation, the registration regions are merged if they are contiguous in memory. In LULESH, it allocates several packed buffers for communication, and these packed buffers are contiguous in memory, thus, in our current implementation applied onto LULESH, all the packed buffers belong to one uffd registration. As a result, one write-fault will invalidate all buffer pairs, and insert all pairs into LRU, and compression thread will have to compress these buffers even some of them do not get written.

Due to LULESH's communication pattern, packing before communication, all buffers will be invalidated and will be asked to be compressed. For that reason, no buffer will be compressed and ready for communication. Thus, we implemented a wait loop before each MPI send call to wait for all items in the LRU to be popped off. Such strategy will create additional overhead to the main thread when the overhead of compression cannot be overshadowed by the reduced communication. This additional overhead is shown in Figure 5 where there is a gap in aggregate compression duration between the default LULESH and the uffd version at small problem sizes.

6.1 Exchanged Data Size

Figure 4 shows the difference in exchanged sizes for all three versions. LZ4 works exceptionally well on the benchmark's data and the compression ratio could reach 217. This difference in exchanged data sizes reflects in the total aggregate communication duration in Figure 5. The uffd uses the same compression algorithm, but the exchanged data size exceeds the plain version. This difference is caused by the implementation of the uffd framework. We used a naive hash for a quick address and size pair lookup, and there is collision in the hash. These collisions cause addresses not added to the LRU queue and thus not compressed.

The collision effect also reflects in the aggregate decompression overhead. Since the uffd version compresses less data, the time it takes to decompress is less than the plain version.

6.2 uffd Framework Performance on LULESH

Figure 5 presents the performance comparison among the default LULESH, the plain compression version, and the uffd version. We ran LULESH using 8 ranks,

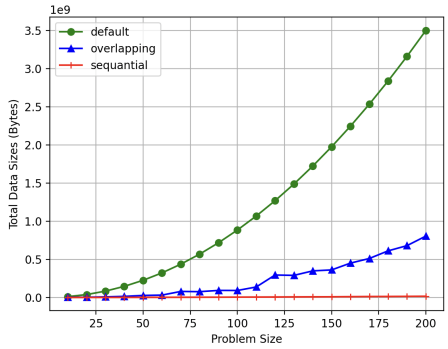

Fig. 4: Communication data size comparison between the default LULESH, the uffd version and the plain compression version

bound and mapped them to hardware thread #1 to #8. Due to the limitation of our machine (32 hardware thread), 8 ranks is the maximum number of processes we can deploy with uffd and compression threads running without interfering the main thread.

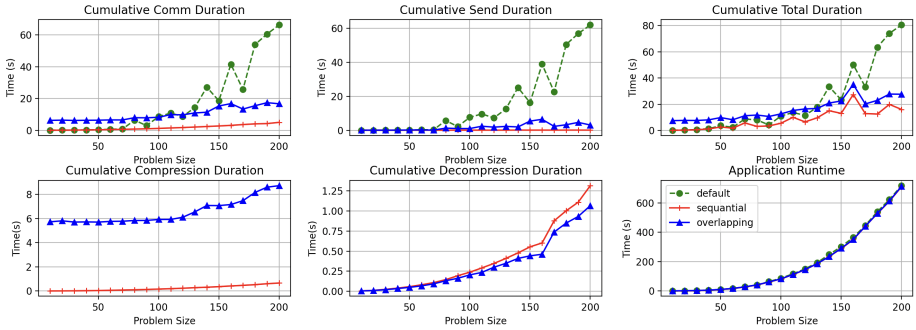

Fig. 5: Performance comparison for the default version, the sequential compression version and the overlapping compression version on LULESH

In default LULESH, it issues non-blocking sends all at once, followed by an MPI_Waitall to ensure the completion. On receive side, it issued non-blocking receives, each followed by an MPI_Wait and then unpack the received buffer into the user buffer.

In order to correctly timed the sends separately, we modified the non-blocking sends to blocking sends. Although blocking send in MPI does not guarantee the completion on the receiver side, it gives a better sense on how fast data is transmitted into the NIC. We timed each operation (compression, send, wait,

decompression) separately so that we are able to compare the uffd version to plain version and understand what situation it would be useful.

Table 2 shows the aggregate communication time for LULESH with 50 cycles for problem size 200. The aggregate send duration is proportional to the data size.

Table 2: Aggregate time comparison for LULESH with problem size 200 at 50 iterations

	Cumulative Total Duration (s)	Send Duration (s)	Compression Duration (s)
Default LULESH	80.52964	62.01993	N/A
Sequential Compression LULESH	15.99959	0.10532	0.66322
Overlapping Compression LULESH	27.60461	3.11604	8.71220

Because communication in LULESH only takes an insignificant amount of the total runtime, any reduction in communication can hardly be reflected. However, combining the results of Figure 5 and Table 4, most of the improvement in communication comes from how well the compression algorithm is able to compress the data. As the compression ratio increases while the compression overhead stays relatively low, there will be an improvement in the communication.

6.3 uffd Framework Evaluation and Future Improvements

Applying our current uffd framework onto LULESH improves LULESH's communication on large problem sizes, but still comes short compared to the plain compression version. We conclude that there are several key factors and propose some future improvements to our uffd framework:

1. uffd's hash collisions are causing some send buffers not to be compressed for each iteration in LULESH. Since each iteration sends exactly the same buffers, the collided buffers will never be compressed.
2. Due to the merging of registration, and all send buffers in LULESH are either overlapped or contiguous in memory, one write-fault will invalidate all previously compressed buffers. With that, our current strategy of waiting for all items to be compressed adds additional overhead to each communication. It might be beneficial to change uffd registration from per memory region to per page basis, thus, one write-fault will only invalidate buffers that resides within the same page.
3. LULESH's pack then send communication pattern will cause the send buffer to be invalidated, thus, the compressed counterpart will never be ready at the send call. Ideally, if LULESH is able to perform all pack operations then invoke communication on all pack buffers, uffd will have a much better chance finishing all the compression depending on how heavy the pack or

the communication is. Depending on how heavy the pack is and the latency from the intra- or inter-communication, the compressions from uffd could be completely hidden.

While our current uffd framework demonstrates potential for improving communication performance in LULESH, several inherent issues still hinge, such as persistent hash collisions, coarse-grained registrations, and communication patterns that conflict with compression overhead compared to plain compression version. Addressing these identified challenges will present promising pathways for hiding compression overhead while gaining the benefit of exchanged compressed counterpart.

7 Conclusion

In conclusion, we introduced an innovative framework for early compression in the Open MPI, utilizing the uffd mechanism for efficient write detection combined with the high-performance LZ4 compression algorithm. Our approach effectively reduces communication overhead by compressing data prior to transmission, significantly improving performance. Through detailed experimentation using the LULESH, we demonstrated that the proposed framework has the potential to hide compression overhead while decreasing communication overhead with reduced volume of data exchanged. Additionally, we explored the framework's applicability to other HPC benchmarks, revealing potential benefits across various applications. Our work in uffd shows promising results for its application in real-world scenarios, even though uffd research itself remains relatively immature. Future enhancements, including collision-resistant hashing, finer-grained uffd registrations, adjustments to application communication patterns, and selecting alternative compression algorithms to achieve higher compression ratios, offer promising avenues to further optimize this framework. Ultimately, our work contributes valuable insight toward application of the uffd and addressing communication bottlenecks in high-performance computing environments, laying a foundation for more efficient parallel computing solutions.

References

1. Yann Collet. LZ4 - Extremely fast compression. https://lz4.org, 2024. Accessed: 2024-03-27.
2. Rosa Filgueira, Jesus Carretero, David E Singh, Alejandro Calderon, and Alberto Núñez. Dynamic-compi: Dynamic optimization techniques for mpi parallel applications. *The Journal of Supercomputing*, 59:361–391, 2012.
3. Rosa Filgueira, David E Singh, Alejandro Calderón, and Jesús Carretero. Compi: enhancing mpi based applications performance and scalability using run-time compression. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings 16*, pages 207–218. Springer, 2009.

4. Rosa Filgueira, David E Singh, Jesús Carretero, Alejandro Calderón, and Félix García. Adaptive-compi: Enhancing mpi-based applications' performance and scalability by using adaptive compression. *The International Journal of High Performance Computing Applications*, 25(1):93–114, 2011.
5. Lokesh Gidra, Hans-J Boehm, and Joel Fernandes. Utilizing the linux userfaultfd system call in a compaction phase of a garbage collection process. 2020.
6. Lawrence Livermore National Laboratory. Coral-2 benchmarks. https://asc.llnl.gov/coral-2-benchmarks. Accessed: March 11, 2025.
7. Junjie Li, Alexander Bobyr, Swen Boehm, William Brantley, Holger Brunst, Aurelien Cavelan, Sunita Chandrasekaran, Jimmy Cheng, Florina M Ciorba, Mathew Colgrove, et al. Spechpc 2021 benchmark suites for modern hpc systems. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, pages 15–16, 2022.
8. Marty McFadden, Keita Iwabuchi, Eric Green, Roger Pearce, Maya Gokhale, Kai Wu, and Dong Li. Umap: A user level memory mapping library.
9. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.
10. Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78. IEEE, 2019.
11. Bharath Ramesh, Qinghua Zhou, Aamir Shafi, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar K Panda. Designing efficient pipelined communication schemes using compression in mpi libraries. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 95–99. IEEE, 2022.
12. Hongzhang Shan, Samuel Williams, and Calvin W Johnson. Improving mpi reduction performance for manycore architectures with openmp and data compression. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–11. IEEE, 2018.
13. Q Zhou, C Chu, NS Kumar, Pouya Kousha, Seyedeh Mahdieh Ghazimirsaeed, Hari Subramoni, and Dhabaleswar K Panda. Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 444–453. IEEE, 2021.
14. Tal Zussman, Teng Jiang, and Asaf Cidon. Custom page fault handling with ebpf. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 71–73, 2024.