Comprehensive Study for Just-In-Time Pack Functions in Open MPI

Yicheng Li
University of Tennessee, Knoxville
yli137@vols.utk.edu

Joseph Schuchart
University of Tennessee, Knoxville
schuchart@icl.utk.edu

George Bosilca
University of Tennessee, Knoxville
bosilca@icl.utk.edu

***Abstract*—Among many of the communications capabilities of the Message Passing Interface (MPI), the manipulation of datatypes, i.e. contiguous and non-contiguous, regular or not, memory locations has been heavily underrated and underutilized. This paper introduces an enhancement to the Open MPI Datatype Engine by incorporating Just-In-Time (JIT) generation of tailored packing functions. The proposed approach aims at optimizing data serialization and communication performance by dynamically generating packing functions tailored to specific datatypes and communication patterns. Leveraging the JIT pack mechanism eliminates branching overhead and enables efficient handling of non-contiguous data movement. Our implementation demonstrates a maximum speedup of up to 3.65x, showcasing the potential performance gains achievable in synthetic scenarios. Furthermore, in real-world application communication patterns, we achieve a notable speedup of 2x, emphasizing the practical relevance of our approach in improving communication performance for various datatypes and application workloads within the Open MPI framework.**

***Index Terms*—MPI, Open MPI, JIT, Datatype Engine**

I. Introduction

Efficient data packing is a critical factor in achieving high-performance communication in parallel and distributed computing systems. The Open MPI (OMPI) library provides a flexible datatype engine that enables optimized data representation and transfer. However, in real-world application communication patterns, where data transmission occurs incrementally via a communication pipeline, traditional packing approaches may suffer from performance limitations. To address this challenge, we combine the capabilities of the Open MPI datatype engine with libgccjit, a just-in-time (JIT) compilation library, to dynamically generate packing functions tailored for pipelined communication and memory layouts.

In real-world applications, data is transmitted in segments through various stages, enabling overlapping of buffer packing and communication, reducing latency, and improving system efficiency. Our approach leverages libgccjit to generate two JIT packing functions: one for packing the entire datatype and another for packing partial datatypes when pipelining is utilized during communication. These dynamically generated packing functions optimize data transfer for improved performance.

The default packing approache in OMPI involves traversal of the internal datatype representation, which incurs overhead for identifying the type of each datatype element, calls to (non-inlined) data movement functions (e.g., memcpy), and conditional branches. While branch instructions are typically cheap on their own, excessive use of conditional branch instructions can lower datatype packing performance, especially for small datatypes. In contrast, our approach attempts to eliminate conditional branches as much as possible by generating tailored JIT packing functions at commit time, allowing the compiler to inline data movement functions and optimizing code generation to further remove branch instructions.

The contributions of our work include:

- The integration of libgccjit with the OMPI datatype engine to dynamically generate optimized packing functions.
- The provision of JIT packing functions and discussion of partial packing strategies in the context of JIT datatype packing
- Identifying the optimal utilization scenarios for JIT packing functions entails highlighting their advantages, while also acknowledging the potential drawbacks associated with their usage.

In this paper, we present the design, implementation, and evaluation of our approach. We conduct experiments to compare the performance of the Open MPI's state-of-the-art packing function with the JIT-generated packing functions for both entire and partial datatypes. The results demonstrate the effectiveness of our approach in enhancing communication performance, especially in pipelined communication scenarios.

Section II delves into an examination of prior work regarding MPI datatypes. In Section III, we elucidate the motivation behind undertaking this research. Section IV provides details on the current OMPI datatype engine. Section V provides an introduction to libgccjit, the fundamental framework employed in this study. Section VI offers a comprehensive description of our benchmarks, sample datatypes, and the testing environments utilized. Section VII outlines the proposed pack implementation with libgccjit. Section VIII presents the results of the experimental evaluation and performance analysis. Finally, Section IX concludes the paper by emphasizing potential avenues for future research that aim to optimize packing for pipelined communication scenarios.

II. Related Work

The ongoing endeavor to enhance MPI datatype support aims at harnessing available communication resources effectively. However, the lack of performance portability and the overhead associated with a generic (non-optimized) datatype

engine [10], [13] have hindered the adoption of MPI datatypes in various scientific applications [2]. Efforts have also been made to overhaul and to revamp the datatype description within MPI itself [4], [8], [15], [17] to reduce generic packing overhead. JIT compilation has also been corporated into the datatype engine [9], however, a comprehensive reasoning for the addition of JIT compilation is still needed to differentiate between best and worst use case.

With the presence of high-performance interconnects like Infiniband (IB) [1], MPI libraries can leverage efficient network features. Previous studies have demonstrated the advantages of utilizing gather/scatter and Remote Direct Memory Access (RDMA) [11], [14], [16] as an efficient approach to eliminate the need for pack and unpack operations during communication. This allows direct read/write of data blocks from sender/receiver. To further exploit the interconnect's capabilities, Mellanox introduced User-mode Memory Registration (UMR) [7], enabling RDMA-based access to multiple non-contiguous data blocks using Scatter-Gather-Lists (SGL). Zero-copy strategies [5], [6], [11], [16] were also introduced to reduce the overhead of additional data copies incurred during packing/unpacking. However, for small data size segments, using zero-copy or UMR may result in higher overhead compared to employing an extra copy [7]. Therefore, enhancing pack/unpack performance remains crucial for small data segments.

Utilizing x86 Advanced Vector Extensions (AVX) and Arm's Scalable Vector Extension (SVE) on the host has also shown improved time-to-solution in predefined MPI reduction operations [18]–[20]. Leveraging the vectorization support of processors can automatically vectorize at runtime, hence, enhance the efficiency of the datatype engine and narrow the gap towards achieving peak performance.

III. Motivation

Performance Optimization: memcpy is a standard library function used to copy blocks of memory, but it may not be the most efficient solution for copying fixed-length data in certain scenarios. By using JIT compilation, the code can be optimized specifically for the fixed length, potentially resulting in faster and more efficient memcpys. JIT compilation allows for runtime code generation and optimization, tailoring the code to the specific data and operation at hand.

Reduced Overhead: Using a fixed length for the length field in memcpy function eliminates the need to branch into memcpy function in certain cases, thereby reducing overhead. This can be beneficial in situations where the length is known and constant, allowing for more streamlined and efficient data copying operations.

Simplified Code: The existing Open MPI pack function for non-contiguous datatype involves numerous checks related to traversing the datatype representation and bookkeeping. This results in considerable overhead. To mitigate these issues, a specialized JIT pack function can be implemented, which predefines a pack routine. By doing so, both types of overhead can be efficiently eliminated.

IV. Open MPI Datatype Engine

A. Datatype Engine

In Open MPI, the term datatype engine refers to the underlying infrastructure that handles the creation, manipulation, and efficient communication of complex data structures within MPI applications. MPI datatypes allow developers to define in-memory representation of custom data structures and enable efficient communication of non-contiguous or structured data across distributed memory systems.

The datatype engine in Open MPI provides functions and mechanisms for creating, manipulating, and optimizing MPI datatypes. It allows developers to define custom datatypes using various features such as structuring, indexing, subsetting, and vectorization. These capabilities enable efficient packing and unpacking of data for communication, reducing the amount of data transmitted and minimizing communication overhead.

The Open MPI datatype engine also includes advanced features like datatype optimization and support for derived datatypes. It employs sophisticated algorithms and techniques to optimize the layout and representation of data, ensuring efficient communication and reducing the impact of data serialization and deserialization.

By leveraging the Open MPI datatype engine, developers can enhance the performance of their MPI applications by utilizing customized data structures and minimizing data movement and communication costs. It allows for improved scalability and efficiency in parallel and distributed computing scenarios, where data communication is a critical aspect of performance.

B. Communication Pipelining

Communication pipelining is a strategy that allows overlapping communication and computation to minimize the impact of communication latency on the overall performance of parallel applications. In a parallel environment, communication latency refers to the time it takes for a message to traverse between processes, which can significantly affect the execution time of an application.

By employing pipelining techniques, Open MPI is able to optimize data transfer and reduce the overall time required for communication. Pipelining achieves this by dividing a communication operation into multiple smaller tasks or stages, each of which can be executed concurrently with other computations. This approach allows Open MPI to overlap communication tasks with computation, effectively hiding or mitigating the impact of communication latency.

The pipelining technique in Open MPI can be particularly beneficial in scenarios where data needs to be transferred between multiple processes or nodes, such as in parallel matrix computations, iterative algorithms, or distributed data processing. By overlapping computation with communication, Open MPI enables more efficient utilization of system resources and better overall performance.

The process of communication pipelining in Open MPI involves breaking down a communication operation into distinct

stages, each responsible for a specific task. These stages can include tasks like message preparation, message transmission, message reception, and data processing. By carefully designing and organizing these stages, Open MPI can overlap their execution and achieve improved performance.

C. Commit Time Optimization

Datatype commit time optimization is a critical area of research and development in Open MPI. The aim is to reduce the overhead incurred during datatype commit operations and improve the overall performance of parallel applications. As datasets grow larger and computing clusters become more extensive, any datatype optimization in the datatype commit can have a substantial impact on the efficiency of parallel computations. Ultimately, the goal of Open MPI datatype commit time optimization is to make parallel computing more accessible and efficient to harness the full potential of modern HPC systems and accelerate scientific applications.

V. LIBGCCJIT

libgccjit is a library that provides JIT compilation capabilities as a frontend of the GNU Compiler Collection (GCC). JIT compilation is a technique where code is compiled at runtime, allowing for dynamic adjustment and optimization based on a program's input data. It provides an API that allows user to create an abstract syntax tree (AST) representing the desired code, perform optimizations on the AST, and then generate machine code from it. This enables applications to generate and execute code dynamically, which can be useful in scenarios where runtime code generation is required, such as in dynamic language interpreters or just-in-time compilers for virtual machines.

By using libgccjit, developers can take advantage of GCC's powerful optimization capabilities and leverage the existing infrastructure of GCC for code generation. Through libgccjit's documentation, user will find it easier to integrate dynamic code generation into applications among most JIT libraries and take advantage of the optimizations provided by GCC

VI. BENCHMARKS AND TEST ENVIRONMENT

A. DDTBench

DDTBench [12] is a benchmark suite developed by the High-Performance Computing group at ETH Zurich. It focuses on evaluating the performance of distributed data structures in high-performance computing environments.

The main goal of DDTBench is to provide a standardized set of tests for assessing the performance of distributed data structures, specifically those implemented using the MPI. Distributed data structures are essential in HPC applications, where data is distributed across multiple nodes or processors to enable parallel processing.

DDTBench comprises a set of micro-benchmarks that focus on diverse facets of distributed data structures, encompassing communication patterns, data distribution, and memory access. These benchmarks can help researchers and developers understand the performance characteristics and limitations of different distributed data structure implementations.

By running DDTBench, users can measure key performance metrics such as latency, bandwidth, and scalability of distributed data structures.

B. Open MPI datatype benchmark

The Open MPI Datatype Benchmark focuses on measuring the bandwidth and data correctness during the transfer of entire derived datatypes with a user-given count in a single core environment. It allows users to create custom data types that are tailored to their specific application's needs. Another crucial aspect of the benchmark is to ensure the correctness of the data transferred. Data integrity and consistency are essential in parallel computing, as any errors during data transmission can lead to incorrect results or program crashes.

C. Sample Datatypes

Within the Open MPI framework, it is feasible to produce the datatype representation, exemplified in Figure 1. This output is structured in the following format: basic type, count, displacement, block length, extent, (total data size).

In the context of the Open MPI datatype benchmark, we investigate a vector datatype that comprises one eight-byte element on each cache line (64-byte) with a count of 8. Additionally, we analyze an indexed datatype, as depicted in Figure 1a, which consists of a 4-byte data element followed by a repeating 40-byte element with a count of 9, and finally, a single 36-byte element at the end.

The DDTBench application comprises 16 distinct data types distributed among 6 distinct application classes. Nevertheless, owing to existing constraints in the JIT pack functionality within the partial pack approach, we have specifically chosen 6 data types from 3 application classes, as detailed in Table I. The atmospheric science data types depicted in Figure 1b are composed of multiple vector elements, while the data types from the other two application classes in Figure 1c are indexed data types, characterized by a substantial number of elements of the same length but lacking a constant pattern that can be unified into a single element.

D. Testing environment

We ran our tests on Intel Xeon Gold 6254 CPU @ 3.10GHz, which has 1.1 MiB 36 MiB and 49.5 MiB cache for L1, L2 and L3 respectively.

VII. PACK WITH LIBGCCJIT

A. Pack for entire datatype buffer

With JIT compilation capabilities, pack functions can be dynamically generated for each derived datatypes. The JIT pack function translates each element in the datatype representation into a series of builtin memcpy operations with a fixed length specified in the length parameter. This enables efficient data packing by avoiding unnecessary overhead when branching into memcpy.

Application Class	Testname	Access Pattern
Atmospheric Science	WRF_x_vec WRF_y_vec	struct of 2D/3D/4D face exchanges in different directions (x,y), using different (semantically equivalent) datatypes: nested vectors (_vec)
Molecular Dynamics	LAMMPS_full LAMMPS_atomic	unstructured exchange of different particle types (full/atomic), indexed datatypes
Geophysical Science	SPECFEM3D_oc SPECFEM3D_cm	unstructured exchange of acceleration data for different earth layers, indexed datatypes

TABLE I: DDTBench Micro-Apps

```
OPAL_UINT1      count 1 disp 0x0 (0) blen 4 extent 4 (size 4)
OPAL_UINT1      count 9 disp 0x8 (8) blen 40 extent 44 (size 360)
OPAL_UINT1      count 1 disp 0x194 (404) blen 36 extent 36 (size 36)
OPAL_LOOP_E     prev 3 elements first elem displacement 0 size of data 400
```

(a) Indexed Datatype

```
OPAL_FLOAT4 count 3 disp 0x7cad94 (8170900) blen 23 extent 128 (size 276)
OPAL_FLOAT4 count 3 disp 0x7cc7a4 (8177572) blen 23 extent 128 (size 276)
OPAL_FLOAT4 count 3 disp 0x7ce1b4 (8184244) blen 23 extent 128 (size 276)
OPAL_FLOAT4 count 3 disp 0x7cfbc4 (8190916) blen 23 extent 128 (size 276)
OPAL_FLOAT4 count 195 disp 0x7df5d4 (8254932) blen 23 extent 128 (size 17940)
OPAL_FLOAT4 count 195 disp 0x848fe4 (8687588) blen 23 extent 128 (size 17940)
OPAL_FLOAT4 count 195 disp 0x8b29f4 (9120244) blen 23 extent 128 (size 17940)
OPAL_FLOAT4 count 195 disp 0x985e04 (9985540) blen 23 extent 128 (size 17940)
OPAL_FLOAT4 count 195 disp 0xa59214 (10850836) blen 23 extent 128 (size 17940)
OPAL_LOOP_E prev 9 elements first elem displacement 8170900 size
```

(b) WRF_y_vec Datatype

```
OPAL_FLOAT8 count 1 disp 0x1678f88 (23564168) blen 3 extent 24 (size 24)
OPAL_FLOAT8 count 1 disp 0x167a710 (23570192) blen 3 extent 24 (size 24)
OPAL_FLOAT8 count 1 disp 0x167b3a0 (23573408) blen 3 extent 24 (size 24)
OPAL_FLOAT8 count 1 disp 0x1679f00 (23568128) blen 3 extent 24 (size 24)
OPAL_FLOAT8 count 1 disp 0x168ccc8 (23645384) blen 3 extent 24 (size 24)
OPAL_FLOAT8 count 1 disp 0x167df50 (23584592) blen 3 extent 24 (size 24)
...
```

(c) LAMMPS Datatype

Fig. 1: Sample Datatypes

The generated pack function is tailored to the specific datatype's layout, resulting in improved serialization performance. This dynamic code generation approach with libgccjit also allows developers to fine-tune the pack function for different platforms given the correct compilation optimization enabling parameters.

B. Pack for pipelined datatype buffer

The partial pack function is designed to handle packing operations within the Open MPI library, specifically for cases where MPI communication is being pipelined. This function allows for partial packing, which means that data can be packed incrementally or in segments, as opposed to packing the entire data in one step.

The implementation for JIT partial packing is very similar to packing the whole datatype. Because of JIT packing functions are extremely restricted to pre-owned knowledge such as pack routine and memcpy size, the JIT partial packing always starts and stops at the beginning of a datatype element. The difference between packing the whole datatype and packing the partial datatype is that checks for remaining space during pipeline procedure are being added to the beginning of each datatype element.

In our work, we extensively examine the use of JIT functions in communication where pipeline for communication is employed.

C. Merging Pack Functions

Given that JIT pack functions may not encompass all possible scenarios during packing, we have devised a pack routine that combines Open MPI's original packing function, JIT packing function, and JIT partial packing function. This routine is divided into three distinct parts:

- Prologue: In the prologue phase, any remaining elements of a datatype are packed, and subsequently, the JIT partial pack function is invoked to pack the remaining elements until the end of the datatype.
- Loop: During the loop phase, the JIT packing function is called to pack the entire datatype, handling as many entire datatype from user given count as it can in one go.
- Epilogue: The epilogue phase is akin to the prologue but in reverse order. After the pack routine has traversed the prologue and loop phases, the epilogue commences with JIT partial packing. In cases where there is still space left for a partial element, the Open MPI packing function is called to fill the remaining pipeline.

By integrating these three distinct parts, the pack routine achieves a comprehensive and efficient packing process that leverages the benefits of JIT packing while addressing specific packing scenarios not covered by JIT functions. In section VIII, we also look into the percentages of usage of each pack function.

D. JIT Pack Limitations

While the JIT pack mechanism endeavors to create a custom pack function while mitigating the branching overhead, its limitations are apparent. Specifically, it cannot accommodate any unknown factors that may lead to a variable in the length parameter of the memcpy function. We could use variable sizes for memcpy but that would prevent certain optimizations and would potentially nullify the benefits of JIT compilation.

The second limitation is concerning the pipeline size during communication, which may prompt the avoidance of using the JIT pack mechanism. Given that the JIT pack is pre-generated, it does not handle segments of the datatype element. In scenarios involving large contiguous elements, it is conceivable that the pipeline size may never cover the entire element's length, thus leading to the avoidance of employing the JIT pack function.

VIII. Performance

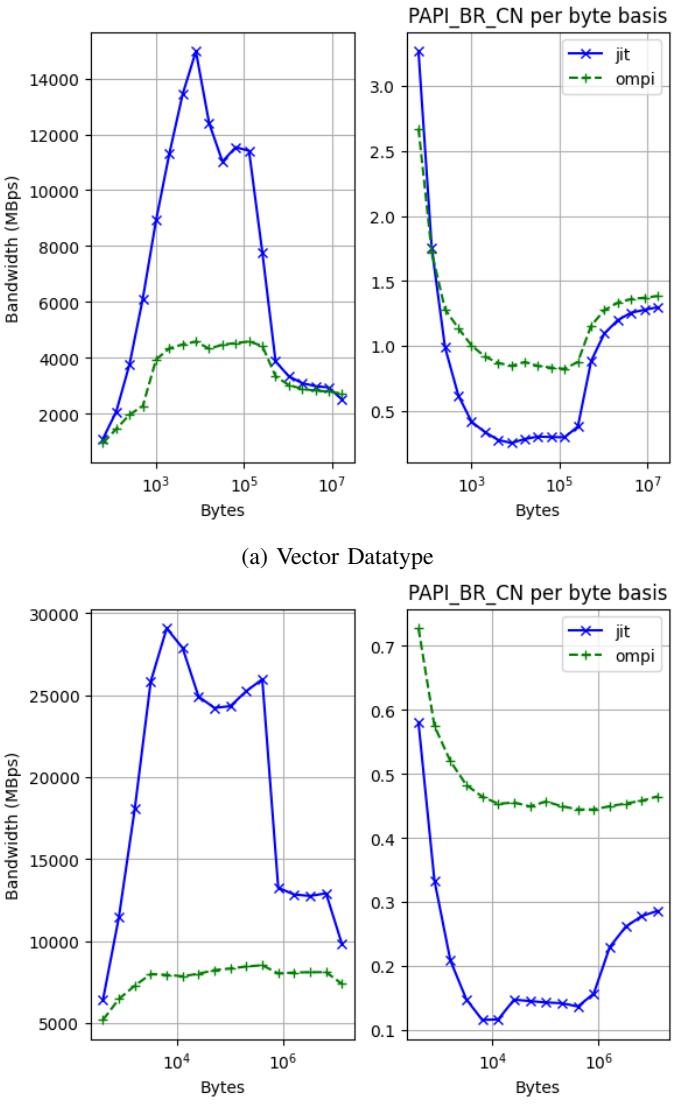
A. Open MPI datatype benchmark

In the Open MPI datatype benchmark, the focus lies on evaluating the performance of the JIT packing function that efficiently packs the entire datatype without the overhead of conditional branches related to pipelining. This approach aims to achieve a substantial performance boost by eliminating all conditional branches within the simple pack function but overhead for traversing the datatype still exists. Furthermore, providing comprehensive information (such as loop count, data access pattern, and fixed length for memcpy function) to the compiler allows the resulting JIT pack function to be fully optimized based on the specific architecture.

Figure 2 illustrates the performance results for both vector and indexed datatypes in the Open MPI datatype benchmark. The left graph in each figure represents the bandwidth for different sizes, while the right graph displays the PAPI [3] counter for conditional branches on a per byte basis.

Figure 2a presents the performance outcomes for the vector datatype. The JIT performance can achieve a significant speedup of up to 3.26x within the L1 cache for this datatype. As the datatype representation is highly optimized (minimal datatype traversing overhead), the performance gap between JIT and OMPI pack functions converges for larger sizes. In such cases, both pack functions hit the main memory latency and memory wall, and the JIT advantages over conditional branches become less pronounced.

Figure 2b exhibits the performance results for the indexed datatype. Unlike the vector datatype, the Open MPI's pack function spends considerably more time in conditional branches due to the datatype's inherent complexity. The traversal of the datatype representation necessitates conditional branches that check for the datatype element type and bookkeeping operations to track the positioning inside the datatype. As a result, the JIT pack function achieves a speedup of up to 3.65x within the L1 cache and maintains a speedup of up to 1.59x within the L2 and L3 caches.



(a) Vector Datatype

(b) Indexed Datatype

Fig. 2: Open MPI Datatype Benchmark

B. DDTBench & JIT Pack Pipelining

In the application environment, the performance is influenced significantly by the implementation of pipelining strategies. DDTBench, in contrast to the Open MPI datatype benchmark, evaluates the data access performance of scientific applications within a suite of Micro-Apps, while Open MPI employs pipelined send and receive operations within. While users have the option to modify the MCA runtime parameter within Open MPI to choose a specific pipelining strategy, we have opted to utilize the default pipelining strategy provided by Open MPI.

DDTBench encompasses 17 different shapes of datatypes. Each data point within the Micro-Apps shares the same shape but varies in terms of data size and extent. From this diverse set of shapes and sizes, we have selected six representative applications from Table 1 that effectively demonstrate the utilization of JIT packing in pipelined scenarios.

Top row of Figure 3 illustrates the performance comparison between pure OMPI and a combination of OMPI pack and JIT pack. As the data size increases along the x-axis, the difference between the performance of the two approaches varies. The utilization of JIT pack is not always 100%, leading to fluctuating performance differences between the two lines. For instance, in the WRF application, during small problem sizes in x-direction, the percentage for JIT goes from 60% to 0%, resulting in performances between the Open MPI's pack function and JIT strategy converge.

LAMMPS exhibits the most significant performance enhancement when employing JIT-based partial packing functions due to its datatype representation which cannot be optimized while each datatype elements all have the same and small sizes (8-byte or 24-byte). This architectural advantage

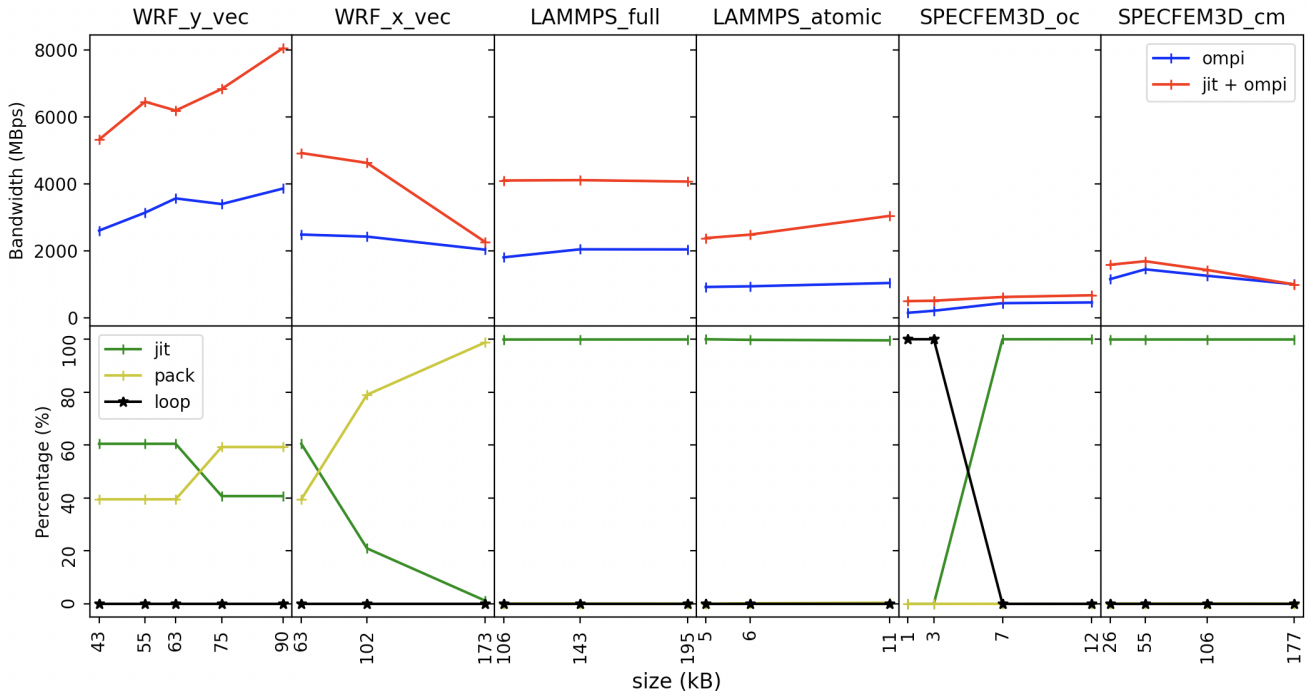


Fig. 3: Various pack costs from DDTBench test cases

yields a consistent 2x increase in bandwidth across the entire application.

C. JIT Overhead

Despite the potential bandwidth enhancement offered by JIT pack in applications, it is crucial to address the inherent side effects associated with its construction. The process of building and compiling the JIT pack is executed at commit time, which results in a significant time overhead, outweighing the benefits gained from running the JIT pack function just once. Table II presents an illustrative example of the Micro-App utilizing datatype WRF_y_vec, revealing that the overhead incurred during JIT pack creation can be more than 30,000 times higher than the regular commit time overhead and would have to call the same pack function at least 4500 times to upset the creation overhead.

Given the unexpectedly substantial creation overhead, future endeavors will concentrate on mitigating this issue. Several proposed solutions include:

- Implementing a mechanism to generate JIT pack functions only when the application necessitates repeated usage of the JIT pack, at least 30,000 times, to offset the creation overhead.
- Offloading the commit time optimization and JIT function generation to a separate thread to reduce the impact on the main process.
- Establishing a datatype signature and locally storing JIT pack functions for reuse, thereby avoiding redundant overhead during commit time.

IX. Conclusion

In conclusion, the integration of JIT generation into the Open MPI Datatype Engine, enabling the creation of tailored packing functions, has demonstrated significant performance gains in data serialization and communication. The achieved maximum speedup of up to 3.65x in benchmark evaluations and 2x in application environments underscores the effectiveness of our approach in optimizing communication performance for diverse data types and workloads.

Looking ahead, there are several promising avenues for future research and development. First, we aim to explore the possibility of offloading JIT generation during commit time, reducing the overhead associated with JIT function creation and enhancing overall performance. Secondly, finding the optimal balance between the number of pack function calls and JIT generation overhead is critical to ensuring efficient time-to-solution for all scientific applications. Additionally, investigating datatype representation alterations to maximize the benefits of JIT pack functions holds great potential for further performance improvements.

In summary, the successful integration of JIT generation into the Open MPI Datatype Engine represents a significant advancement in enhancing communication performance. The observed speedups in both benchmark and application scenarios demonstrate the practical applicability of our approach. As we delve into future research endeavors, we are optimistic about refining and extending our JIT-based approach to address evolving communication challenges in parallel computing systems.

WRF_y_vec (kB)	43	55	63	75	90
OMPI (μs)	149	152	177	189	170
JIT (μs)	4583623	4803514	5398213	4950740	5123496

TABLE II: Datatype commit overhead

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. (1725692); and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] InfiniBand Trade Association et al. Infinibandtm architecture specification. *http://www. infinibandta. org*, 2004.

[2] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of mpi usage in the us exascale computing project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851, 2020.

[3] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications*, 14(3):189–204, 2000.

[4] William Gropp, Ewing Lusk, and Deborah Swider. Improving the performance of mpi derived datatypes. In *Proceedings of the Third MPI Developer's and User's Conference*, pages 25–30. Citeseer, 1999.

[5] Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. Falcon: Efficient designs for zero-copy mpi datatype processing on emerging architectures. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 355–364. IEEE, 2019.

[6] Jahanzeb Maqbool Hashmi, Ching-Hsiang Chu, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. Falcon-x: Zero-copy mpi derived datatype processing on modern cpu and gpu architectures. *Journal of Parallel and Distributed Computing*, 144:1–13, 2020.

[7] Mingzhe Li, Hari Subramoni, Khaled Hamidouche, Xiaoyi Lu, and Dhabaleswar K Panda. High performance mpi datatype support with user-mode memory registration: Challenges, designs, and benefits. In *2015 IEEE International Conference on Cluster Computing*, pages 226–235. IEEE, 2015.

[8] Yicheng Li, Joseph Schuchart, and George Bosilca. Mars: Memory access rearrangements in open mpi. In *2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH)*, pages 26–33. IEEE, 2022.

[9] Tarun Prabhu and William Gropp. Dame: A runtime-compiled engine for derived datatypes. In *Proceedings of the 22nd European MPI Users' Group Meeting*, pages 1–10, 2015.

[10] Robert Ross, Daniel Nurmi, Albert Cheng, and Michael Zingale. A case study in application i/o on linux clusters. In *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 59–59. IEEE, 2001.

[11] Gopalakrishnan Santhanaraman, Jiesheng Wu, and Dhabaleswar K Panda. Zero-copy mpi derived datatype communication over infiniband. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 47–56. Springer, 2004.

[12] Timo Schneider, Robert Gerstenberger, and Torsten Hoefler. Micro-applications for communication data access patterns and mpi datatypes. In *European MPI Users' Group Meeting*, pages 121–131. Springer, 2012.

[13] Xian-He Sun et al. Improving the performance of mpi derived datatypes by optimizing memory-access cost. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 412–419. IEEE, 2003.

[14] Monika ten Bruggencate and Duncan Roweth. Dmapp-an api for one-sided program models on baker systems. In *Cray User Group Conference*, 2010.

[15] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: Efficient handling of mpi derived datatypes. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 109–116. Springer, 1999.

[16] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. High performance implementation of mpi derived datatype communication over infiniband. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 14. IEEE, 2004.

[17] Qingqing Xiong, Purushotham V Bangalore, Anthony Skjellum, and Martin Herbordt. Mpi derived datatypes: Performance and portability issues. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–10, 2018.

[18] Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. Using advanced vector extensions avx-512 for mpi reductions. In *27th European MPI Users' Group Meeting*, pages 1–10, 2020.

[19] Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. Using long vector extensions for mpi reductions. *Parallel Computing*, 109:102871, 2022.

[20] Dong Zhong, Pavel Shamis, Qinglei Cao, George Bosilca, Shinji Sumimoto, Kenichi Miura, and Jack Dongarra. Using arm scalable vector extension to optimize open mpi. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 222–231. IEEE, 2020.