AVX-512 Gather and Scatter for Non-contiguous Data Movement in OPEN MPI

Dong Zhong[1,2], Qinglei Cao[1,2], George Bosilca[1,2], and Jack Dongarra[1,2]

[1]Innovative Computing Laboratory, The University of Tennessee, US
[2]{*dzhong, qcao3*}@*vols.utk.edu*, {bosilca, dongarra}@icl.utk.edu

***Abstract*—Modern advanced architectures and processors in High-Performance Computing (HPC) are continually getting more complex to handle and satisfy the increasing computational and communication needs. Innovative features and configuration options from novel architectures and processors with long vector extension become much more important to exploit the potential peak performance. This brings new challenges and opportunities to the design of software and libraries, especially regarding MPI. Intel Xeon processors introduced Advanced Vector Extension 512 (AVX-512), which expanded vector length to 512 bits with more powerful features. These new features allow for better compliance with rich memory access patterns and long vector instructions, such as, gather load and scatter store.**

This paper proposes new optimized strategies by utilizing the AVX-512 gather and scatter feature to improve the packing and unpacking operations for non-contiguous data movement. With this optimization, we provide higher parallelism for a single node and achieve a more efficient communication scheme of message exchanging. We implemented our proposed optimization in the context of OPEN MPI, providing efficient and scalable capabilities of AVX-512 usage and extending the possible implementations of AVX-512 to a more extensive range of programming and execution paradigms.

The evaluation of the resulting software stack demonstrates our design significantly improves the pack and unpack performance of default OPEN MPI and achieves decent speedups against state-of-the-art memory copy based implementations on tested benchmark and applications.

***Index Terms*—OPEN MPI, Intel AVX-512, Single Instruction Multiple Data, Long Vector Operation, MPI Pack and Unpack, Gather and Scatter**

I. INTRODUCTION

Modern advanced architectures and processors are the driving force behind today's large-scale High-Performance Computing (HPC) systems. The emergence of a broader range of parallel processor architectures continues to present opportunities to develop an effective programming model that provides access to those architectures' capabilities and novel features. The availability of these architectures in modern HPC systems has significantly enhanced scientists from different research domains to explore and investigate more complex and multiple levels of parallelism.

Many researchers focus on exploiting data-level parallelism by vector execution and code vectorization [1], [2], which leads HPC systems to equip with vector processors. Compared to traditional scalar processors, extension vector processors support single Instruction Multiple Data (SIMD). More effective instructions operate on vectors with multiples elements involved rather than a single element that could achieve maximum computational power.

978-1-7281-6677-3/20/$31.00 ©2021 IEEE

There are efforts to improve the vector processors by increasing the vector length and adding new vector instructions. Intel launched different generations of processors that equipped with Advanced Vector Extensions (AVXs). The Haswell processors introduced the 256 bits registers and instructions (AVX2). Moreover, Xeon Phi processor code-named Knights Landing (KNL) [3], [4] expanded the vector length to 512 bits, as shown Figure 1, with novel and more powerful features. The new registers are an extension of previous 256 bits YMM registers to 512-bit wide SIMD ZMM registers. The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers, and the lower 128-bit are aliased to the respective 128-bit XMM registers. First, the long vector can encapsulate more data than traditional registers; it packs eight double-precision or sixteen single-precision floating-point numbers, eight 64-bit integers, or sixteen 32-bit integers within a single 512-bit vector. This enables processing twice the number of data elements than Intel AVX/Intel AVX2, and four times than SSE. Secondly, it introduced more complex and powerful memory operations, including mask operations and gather load and scatter store capabilities. For our region of interests, AVX-512 takes not the only advantage of using long vectors but also enables powerful high vectorization features that can achieve significant speedup that can be integrated into OPEN MPI.

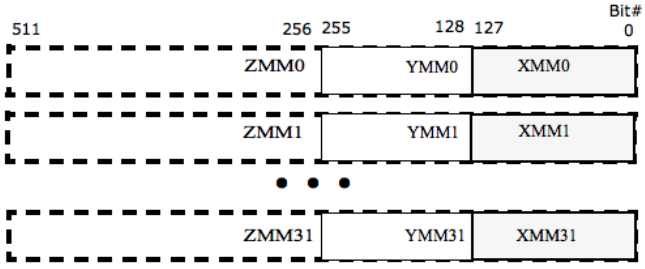


Fig. 1: AVX-512 Bits Wide Vectors and SIMD Register Set

Arm also announced the new Armv8 architecture embraced SVE- a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture [5], [6]. Unlike

other SIMD architectures, SVE does not define the size of the vector registers, instead it provides a range of different values which permit vector code to adapt automatically to the current vector length at runtime with the feature of *Vector Length Agnostic* (VLA) programming [7], [8]. Vector length constrains in the range from a minimum of 128 bits up to a maximum of 2048 bits in increments of 128 bits. Fujitsu released the first processor A64FX [9] to implement SVE. It has 48 calculation cores and two or four assistant cores. In addition, it can perform single-precision and half-precision floating-point calculations, as well as 8-bit and 16-bit calculations with 512-bit wide SIMD.

The Message Passing Interface (MPI) [10] has been a popular and efficient parallel programming model for distributed memory systems that widely used in scientific applications for the last couple of decades. Many scientific applications operate on and communicating with different type of data with rich memory layout patterns, which poses challenges to researchers dealing with the complex data movements and achieves good performance at the same time. However, MPI has been very successful in implementing regular, iterative parallel algorithms with well-defined communication behaviors. Data-driven applications often pose challenges associated with dealing with contiguous and non-contiguous data. These issues are harder to address with a traditional message-passing programming paradigm. Thus, MPI offers a mechanism called derived datatype (DDT) to specify arbitrary memory layouts for sending and receiving messages for both point-to-point and collective communications. This powerful mechanism allows integrating communication into the parallel algorithm and data layout and is likely to become an essential part of application development and optimization. DDT provides an abstract and versatile interface to specify arbitrary data layouts and a portable, high-performance abstraction for data. As we mentioned, new architectures and processors provide more opportunities for MPI libraries, which enables MPI implementations to take advantage of those new features to be carefully designed to deliver the best possible performance for different communication primitives to the end application.

Communication-oriented collective operations dealing with non-contiguous require intensive memory management resources, which force the memory bandwidth to become the bottleneck and limit its performance. However, with the presence of advanced architecture technologies introduced with wide vector extension and specialized arithmetic operations, it calls for MPI libraries to provide state-of-the-art design for advanced vector extension (SVE and AVX-512) based versions.

This paper tackles the above challenges and performs an extensive study of the novel gather and scatter feature from AVX-512 and its applicability for MPI datatype communication. After that, we designed and implemented a new strategy in a widespread MPI implementation OPEN MPI to provide our optimized MPI datatype pack and unpack operation to speed up the communication that deals with non-contiguous small blocks datatype. To summarize, this paper makes the following contributions:

1) analyzing AVX-512 gather load and scatter store instructions to speed up the packing and unpacking operation for non-contiguous datatype using related intrinsic which highly increased the performance;
2) and performing benchmark experiment and analysis using this work in the scope of OPEN MPI on a local cluster with Intel Xeon processor that supports AVX-512. Experiment results demonstrate the efficiency of gather and scatter instructions and our implementation.
3) Furthermore, demonstrate the efficiency with two popular applications in scientific computing: Stencil and 2D-FFT, strengthening the performance benefit from of design and implementation. This provides valuable insight and guideline on how long vectors can be used in high-performance platforms and software.

The rest of this paper is organized as follows. Section II presents related researches and efforts taking advantage of long vector extension from Intel and Arm for different applications and libraries, together with a survey about optimizations of MPI that taking advantage of novel hardware. Section III describes the design and implementation details of our optimized packing and unpacking methods in the scope of OPEN MPI using AVX-512 intrinsic and instructions. Section IV describes the performance difference between default OPEN MPI and OPTIMIZED OPEN MPI and provides a distinct insight into how the new vector instructions can benefit MPI. Section V illustrates the performance benefit we get from OPTIMIZED OPEN MPI with two applications that demonstrate that our design and implementation can speed up scientific applications.

II. RELATED WORK

In this section, we survey related work on techniques taking advantages of advanced hardware or architectures. Lim [11] explored matrix matrix multiplication based on blocked matrix multiplication improves data reuse by data prefetching, loop unrolling, and the Intel AVX-512 to optimize the blocked matrix multiplications. Dosanjh et al. [12] took advantage of using AVX vector operation for MPI message matching to accelerate matches demonstrating the efficiency of long vectors. This work [13] presented a new zero-copy scheme to efficiently implement datatype communication over InfiniBand scatter and gather work to optimize non-contiguous point-to-point communication. Mellanox's InfiniBand [14] explored the use of hardware scatter gather capabilities to eliminate CPU memory copies selectively and offload handling data scatter and gather to the supported Host Channel Adapter. This capability is used to optimize small data all-to-all collective. In another work [15], they leveraged the characteristics of SVE to implement and optimize stencil computations, ubiquitous in scientific computing, which shows that SVE enables easy deployment of optimizations like loop unrolling, loop fusion, load trading or data reuse. This work [16] explored the usage and performance of SVE vector instructions to optimize memory access operation and reduction operation, simulation

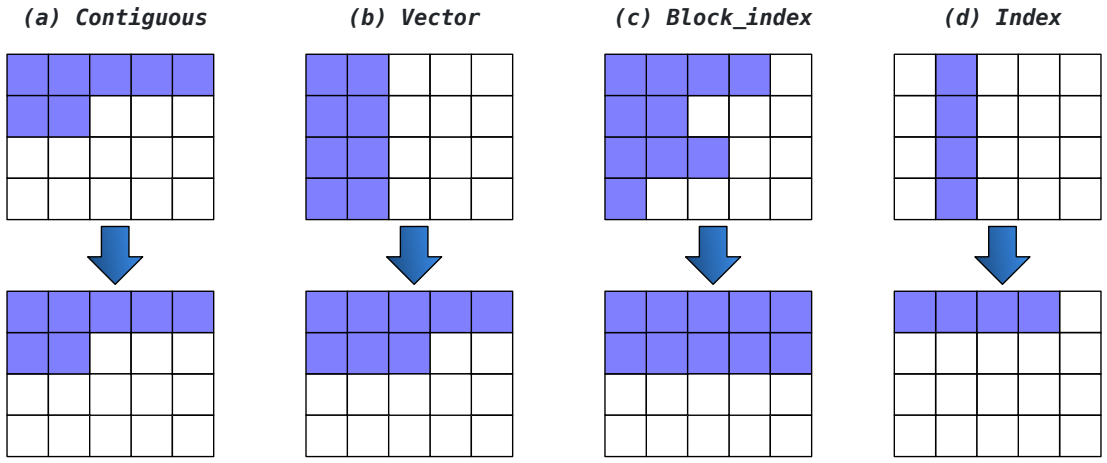


Fig. 2: Memory layout of datatype (contiguous and non-contiguous) in MPI

and benchmark results show that SVE long instruction can speedup related operations. Hashmi [17] proposed a Fast and low-overhead communication framework for optimized zero-copy intra-node derived datatype communication on emerging CPU/GPU architectures. Hofmann [18] presented a pipeline algorithm for MPI Reduce that used a Run Length Encoding scheme to improve the global reduction of sparse floating-point data.

Additionally, different techniques and efforts have been studied to optimize MPI communication operations. Wu [19] proposed GPU datatype engine that offloads the pack and unpack work to GPU to take advantage of GPU's parallel capability to provide high efficiency in-GPU pack and unpack. Luo [20] presented a new hierarchical autotuned collective communication framework in OPEN MPI, which selects suitable homogeneous collective communication modules as sub-modules for each hardware level, and uses collective operations from the sub-modules as tasks, and organizes these tasks to perform efficient hierarchical collective operations. Zhong [21] explored the usage of long vector extension for reduction operation in MPI, which speedups the local computation in MPI that directly benefits the overall performance of collective reductions for applications. Bayatpour [22] proposed a hardware tag matching aware MPI library and discusses various aspects and challenges of leveraging this feature in MPI library. Moreover, it characterizes hardware Tag Matching using different benchmarks and provides guidelines for the application developers to develop Hardware Tag Matching-aware applications to maximize their usage of this feature. Hjelm [23] described a new RMA implementation for OPEN MPI. The implementation targets scalability and multi-threaded performance. It describes the design and implementation of RMA improvements and offers an evaluation that demonstrates scaling to 524,288 cores.

In our work, we study AVX-512 enabled features more comprehensively and provide detailed analysis about the efficiency achievements of using related intrinsic. Our optimization is general at the processor instruction level, which is more straightforward, and uses CPU resources only without external or extra hardware.

III. DESIGN AND IMPLEMENTATION IN OPEN MPI

A. Intel Advanced Vector Extension

Intel Advanced Vector Extension is a significant improvement to Intel Architecture, which enriches significant supports with longer vectors compared to 128-bits single instruction multiple data instructions. It supports the vast majority of previous generations' instructions to operate on 256- and 512-bits registers to support more powerful and efficient instruction. AVX enhances broadcast, reduction and mask instructions to accelerate numerical computations. The Intel micro-architecture Haswell supports the 256-bit AVX2 instructions implementing 256-bit data path with higher throughput. Knights Landing processor extends this feature to more advanced 512-bit wide registers. AVX-512 provides enhanced functionalities for broadcast and permute operations on data elements, vector reduction instructions, and instructions to load-/store contiguous and non-contiguous data elements from/to memory. It also has more powerful packing capabilities with longer vectors, which can encapsulate eight double-precision or sixteen single-precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within a vector. Furthermore, it contributes to better performance for the most demanding computational tasks with more vectors(32 vector registers, each 512 bits wide, eight dedicated mask registers), enhanced high-speed math instructions, and compact representation of large displacement value.

AVX-512 supports more features that we intend to explore and use in this work, such as operations on packed floating-point data or packed integer data, new operations, additional gather/scatter support for non-contiguous memory access, and the ability to have optional capabilities beyond the basic data type and instruction sets.

B. Memory access pattern

A memory operation is the most widely used operation during communication, including point-to-point and collectives.

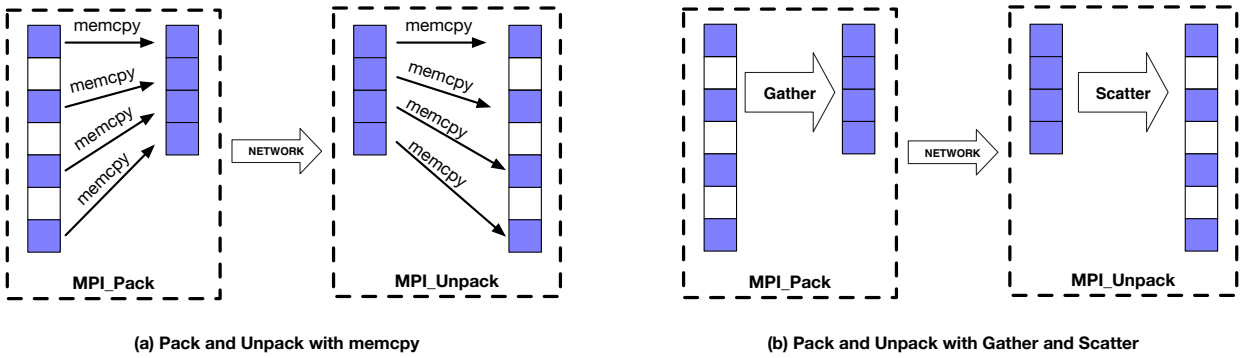


Fig. 3: Comparison between general memory copy and AVX gather/scatter implementation for packing and unpacking

Data need to be packed at the sender side before sending and be unpacked at the receiver side. The datatype constructs provided by the MPI standard give one the capability to define contiguous and non-contiguous memory layouts, allowing developers to reason at a higher level of abstraction, thinking about data instead of focusing on the memory layout of the data (for the pack/unpack operations). MPI defines data layouts of varying complexity: contiguous and non-contiguous data layout, as shown in Figure 2.

Contiguous type is the simplest derived type: a number of repetitions of the same datatype without gaps in-between, as shown in Figure 2(a). C standard library provides function as memory copy to manipulate the contiguous type. With the help of modern compiler it is converted to assembly code which is represented as a loop of load and store instructions using vector registers.

For **non-contiguous** datatype layouts, as shown in Figure 2, vector type Figure 2(b) is the most regular and certainly the most widely used MPI datatype constructor. Vector allows replication of a datatype into locations that consists of equally spaced blocks, describing the data layout using block-length, stride and count – *Block-length* refers to the number of primitive datatypes that a block contains, *stride* refers to the number of primitive datatypes between blocks, and *count* defines the number of blocks that needs to be processed. A distinctive flavor of vector datatype, frequently used in computational sciences and machine learning, access a single column of matrix as presented in Figure 2(d) and can be represented by a specialized vector type with block-length equal one.

Datatypes other than vector exposes less and less regularity, and neither the size of each block nor the displacements between successive blocks are constant. In order of growing complexity, MPI supports INDEXED_BLOCK (constant block-length different displacements), INDEXED (different block-lengths and different displacements), and finally STRUCT (different block-lengths, different displacements, and different composing datatypes). Such datatypes Figure 2(c) cannot be described in a concise format using only block-length and stride.

C. Pack and unpack with gather and scatter

High-performance parallel algorithms and scientific applications often need to communicate non-contiguous data. Typically, applications need to pack the non-contiguous data into a temporary contiguous buffer and send it to endpoint processes. The receiver performs the unpack operation to distribute the data from the contiguous receive buffer to the non-contiguous data buffer/memory. However, this approach (known as "Manual Packing/Unpacking") limited the performance because it usually needs to create several copies of the data, which increases memory footprint of the application. Also, application developer needs to manage those temporary buffers manually, leading to poor productivity. This packing/unpacking process requires considerable time. A previous study [24] has shown that packing and unpacking data could take 90% of the total communication overhead for non-contiguous sends and receives. It is essential for MPI community to provide efficient MPI datatype communication, which could reduce or remove the packing/unpacking overhead for non-contiguous data. To resolve this problem, MPI derived datatype (DDT) provides the convenience of hiding the complexity of sending non-contiguous data from application developers.

Figure 3 give an overview of OPTIMIZED OPEN MPI transferring non-contiguous data by using the gather and scatter feature from long vector extension in packing and unpacking, respectively. For the default memory copy based packing/unpacking scheme, each segment of the data is copied into a pack buffer and transferred to the receiver, the receiver then copies the data segment by segment into a distributed memory. The gather and scatter scheme, on the other hand, on the sender side it replaces multiple small memory copies by single gather instruction to fetch/load data from different memory addresses. On the receiver side, it uses single scatter instruction to replaces multiple small memory copies to distribute/store data into different memory addresses.

The detailed gather load and scatter store instruction for non-contiguous small block data is revealed in Algorithm 1. In this optimized algorithm, we can see for the "gather_pack" procedure, we use intrinsic *_mm512_type_gather_type (Src, ..., offsets, ...)* to load data from different memory addresses based on offsets to a single long vector, and then store it into

our destination. To be noted, for each vector type we only need to generate the offsets once, and repeatedly use this for all gather instructions. For the "scatter_unpack" procedure, we first load the packed contiguous data to a long vector and then use intrinsic *_mm512_type_scatter_type (Dst, ..., offsets, ...)* to store the data to non-contiguous addresses. For the remaining blocks with total size smaller than vector length, we explicitly use mask load and store operation to partially load/store the data from/to memory to maintain the integrity and correctness of our data. This highly expands the limited performance of memory operations for small non-contiguous memory blocks.

Algorithm 1 Gather/Scatter based pack and unpack algorithm

vector_bytes ▷ Vector length in bytes
blocklen ▷ Block length in bytes
threshold ▷ Threshold to pick gather/scatter or memcpy based algorithm, calculated by block length and vector length
blocks_in_vl ▷ Number of blocks can be packed in single vector
off_sets ▷ Offsets of elements to be packed in a single vector, calculated by address, block length and extend
load_mask ▷ Mask for partial load/store

```
1:  procedure GATHER_PACK( Count, Blocklen, Extend )
2:      if ( blocklen ⩾ threshold ) then
3:          for k ← 0 to Count do
4:              memcpy(blocklen,Src,Dst)
5:      else
6:          blocks_in_vl = vector_bytes / blocklen
7:          Generate offsets
8:          for k ← 0 to (Count / blocks_in_vl) do
9:              _mm512_type_gather_type (Src, ..., offsets, ...)
10:             _mm512_store_type(Dst, ...)
11:             update address
12:             update count
13:         Generate load_mask
14:         gather remaining blocks
```

```
1:  procedure SCATTER_UNPACK( Count, Blocklen, Extend )
2:      if ( blocklen ⩾ threshold ) then
3:          for k ← 0 to Count do
4:              memcpy(blocklen,Src,Dst)
5:      else
6:          blocks_in_vl = vector_bytes / blocklen
7:          Generate offsets
8:          for k ← 0 to (Count / blocks_in_vl) do
9:              _mm512_load_type(..., Src)
10:             _mm512_type_scatter_type (Dst, ..., offsets, ...)
11:             update address
12:             update count
13:         Generate load_mask
14:         scatter remaining blocks
```

We implemented our AVX-512 based pack and unpack operation work in the Open Portable Access Layer (OPAL) in OPEN MPI. OPEN MPI is based on a Modular Component Architecture [25], [26] that permits easily extending or substituting the core subsystem with new features. As shown below, we added our AVX-512 optimization work in the lowest level (OPAL) of OPEN MPI architecture that implements all datatype related operations with AVX-512 long vector instructions as in Figure 4; also we integrate our new code to automatically detect the hardware information to enable the AVX-512 feature or fallback to the default basic module if it is not supported by the processor as show in Figure 5. To be noted, this component can be extended out the scope of the generic pack and unpack operations.

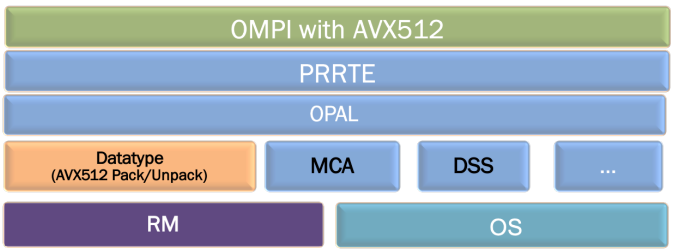


Fig. 4: OPEN MPI architecture. The orange box represent Datatype component under OPAL. We added our AVX-512 gather pack and scatter unpack feature in this component.

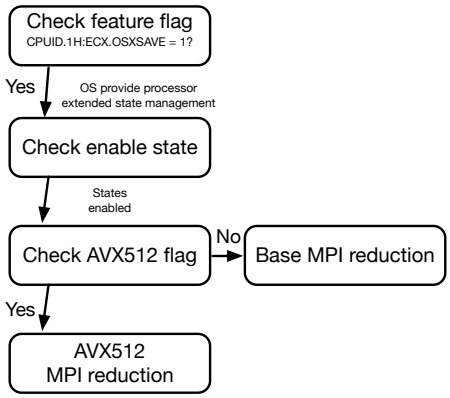


Fig. 5: Integrate and automatically activate the AVX-512 pack and unpack into the OPEN MPI build system

IV. BENCHMARK EVALUATION

Experiments are conducted on a local cluster, an Intel(R) Xeon(R) Gold 6254 based server running at 3.10 GHz. The CPU consists of 18 physical cores, which support advanced features with cpu-flags AVX-512 Foundation (avx512f) and AVX-512 Vector Length (avx512vl), new instruction set extensions, delivering wide (512-bit) vector operations capabilities, with up to 2 FMAs (Fused Multiply Add instructions), to accelerate performance.

Our work is based upon OPEN MPI master branch, git commit hash #406bd3a [27]. Each experiment is repeated 30

times, and we present the average. We use a single node for all benchmark experiments. This section compares the performance of MPI pack and unpack operation with two implementations. The OPEN MPI default version uses general memory copy method during pack and unpack operation for non-contiguous datatypes. It uses a for loop to copy all the blocks for a non-contiguous datatype.

In the new implementation, we use the AVX-512 vector gather feature for packing operation; on the receiver side, we use AVX-512 scatter feature for unpacking. For the pack and unpack benchmark, we use the official test **to_self** in OPEN MPI repository to perform packing and unpacking operation for a vector datatype with different message sizes.

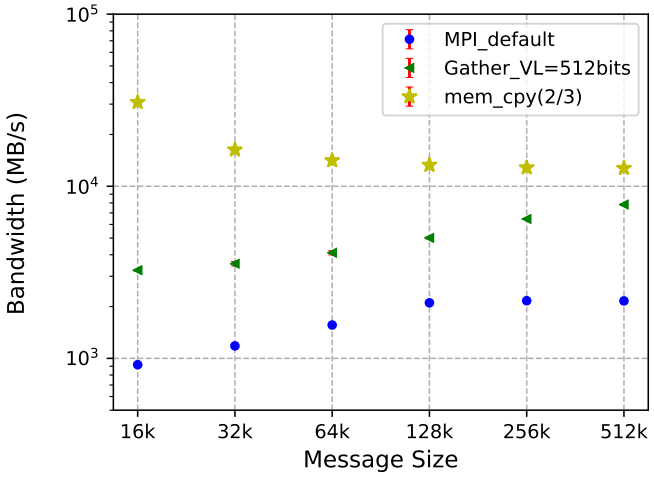


Fig. 6: Comparison of MPI_Pack with AVX-512 gather enable and disable together with memcpy for vector datatype

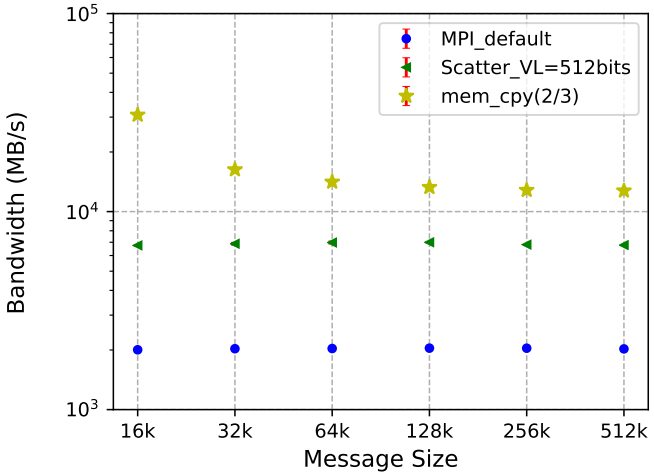


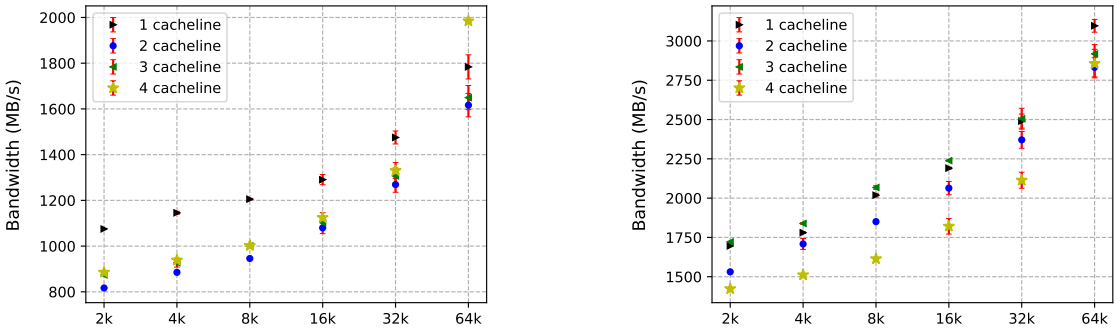
Fig. 7: Comparison of MPI_Unpack with AVX-512 scatter enable and disable together with memcpy for vector datatype

We present to compare the packing and unpacking performance speedup separately to see the benefit we get from gather load and scatter store. For the experiments, we use a vector datatype with primitive datatype MPI_INT constructed with $blocklength = 2, gap = 1, count = 1024$, which means we pack two of three integers for each repetition. Figure 6 displays the performance comparison between OPEN MPI default packing algorithm and AVX-512 gather packing algorithm. The X-axis shows the size of the packed buffer; Y-axis shows the actual bandwidth we get, which means the higher the better. We can see that by using gather feature our optimized packing strategy achieves $2.3 \sim 3.5$ times speedup. We also compare together with memory copy for contiguous data, which indicates the peak memory bandwidth. We can see that even for non-contiguous data, when the message size increased to 512KB, we achieve 41% of the peak bandwidth. Figure 7 presents the performance comparison between OPEN MPI default unpacking algorithm and AVX-512 scatter unpacking algorithm. We can see that by using scatter feature our optimized unpacking strategy achieves 3.4 times speedup. Comparing to memory copy bandwidth, our scatter method achieve 35% the performance of peak bandwidth. We can see that unpacking acquires less efficiency than packing when compare to peak bandwidth from memory copy, which is because reading from non-contiguous addresses is more efficient than writing to non-contiguous addresses. Also, compared to contiguous memory copy, gathers and scatters require the hardware to do more work than contiguous SIMD loads and stores, and are likely to access more cache lines/pages (depending on the specific access pattern). This will cause higher instruction overheads. To be noted, for the memory copy operation we only copy two-thirds the data size of the total unpacked buffer.

To further explore and understand the behavior of non-contiguous data movement during pack and unpack operation, we conducted experiments with different gap length between blocks. This is because with different memory layouts and patterns, vectorization efficiency is not always as expected. Also, we want to investigate gather and scatter capability across cache lines. Generally speaking, gathering and scattering data cross cache lines reduces the effective memory bandwidth. We want to study if our gather and scatter optimization maintains good efficiency with a more complex memory layout. For our test platform, the cache line size is 64-bytes. We tested with vector datatype with block length fixed to one for MPI_INT, and then we vary the stride to different cache line sizes ($1 \sim 4$) for both pack and unpack.

Figure 8 shows the result for the MPI_Pack and MPI_Unpack with MPI default and OPTIMIZED OPEN MPI. Our optimization using intrinsic, which gives us complete control of the low-level details at the expense of productivity and portability. Overall, for both gather and scatter results demonstrate that with AVX-512 enabled operation it is faster than memory copy operation with different stride size cross cache lines.

To be more specific, comparing MPI default Figure 8(a) with gather pack Figure 8(b) we noticed two aspects, (1) the performance for both implementation decreases as stride increases. One possible reason could be that data being gathered spread across multiple cache lines; software typically tests the



(a) Default MPI_Pack

(b) Gather MPI_Pack

(c) Default MPI_Unpack

(d) Scatter MPI_Unpack

Fig. 8: Pack and unpack across different number of cache-lines

completion mask to see if all elements have been read, and if not, loops back and re-executes the instruction. Consequently, the performance of a gather operation decreases as the distance (in memory) between data being gathered increases. (2) Both implementations show performance slowdown but with different trends. We can see that memory copy has a significant performance reduction when the stride increased to across two cache lines. However, for the gather results the decrease is trivial. Also, the trend for memory copy performance decreases immediately after increased the stride longer than a single cache line. Moreover, for the gather pack case, the reduction is gradual. For the unpacking operation, overall, it shows the same trend as packing, which shows performance decrease as stride increases. Furthermore, the speedup gain from scatter seems more evident than gather.

V. Application Evaluation

A. Domain-decomposed 2D stencil

Stencil computation is an important and fundamental algorithm used in a large variety of scientific simulation applications. Stencil codes are most commonly found in the codes of computer simulation in the context of scientific and engineering HPC applications. It involves a large number of iterations, in each of which the value of every element in a matrix is updated using values of its neighbors.

A 2D five-point stencil is a stencil made up of the point itself together with its four "neighbors", each point has four neighbors: up, down, left and right, as shown in Figure 9, we can see that the global domain is represented by an N*N two-dimension matrix, which gets partitioned into multiple blocks (one per process) of roughly equal size. After each computation step, the boundary regions of these partitions have to be exchanged with its four neighboring processes before the next time-step can be started. For easy explanation, we assume matrices are stored in "Row Major" order, data exchanged in the north-south direction is a contiguous pattern. In contrast, the data exchange in the east-west direction is a non-contiguous pattern. There are two ways to handle the communication: the first one is to send and receive the data in multiple small chunks, which can be inefficient due to the constant overhead associated with each send operation; The second method is that the data has to be packed into a consecutive buffer and sent in one piece. On the receiver side, this process has to be reversed (the data has to be unpacked) after such data is received.

We can see that Optimized Open MPI can speed up the packing and unpacking procedure for this east-west direction communication. We use gather and scatter to pack and unpack the boundary regions. For the east-west boundary it can be represented with a vector datatype, as shown in Figure 9. For this particular vector type it is constructed as show in table I below.

In this section we investigate the performance benefit of Optimized Open MPI against default Open MPI. The appli-

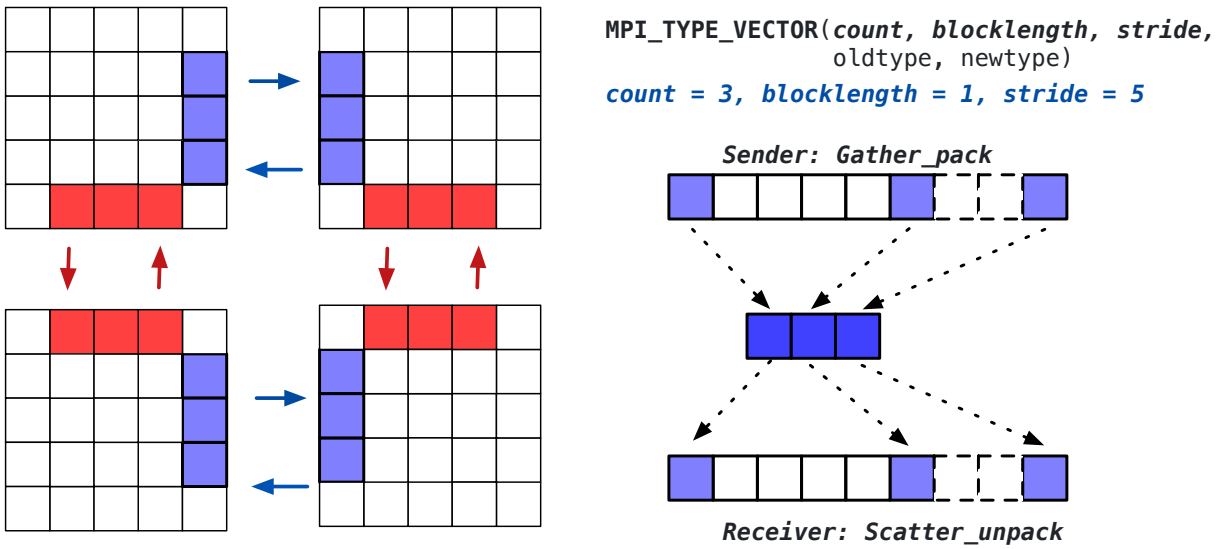


Fig. 9: Domain-decomposed 2D stencil. Data exchanged in east-west direction must be packed and unpacked in communication

TABLE I: East-west vector data represent

Blocklen	Radius
Stride	Number of Columns for each partitioned data set
Count	Number of Rows for each partitioned data set - 2

TABLE II: MPI stencil configuration and execution on 2D grid

Number of ranks	16
Grid size	1000
Radius of stencil	1, 2, 3
Tiles in x/y-direction	2/8
Type of stencil	Star
Data type	Single precision
Number of iterations	100

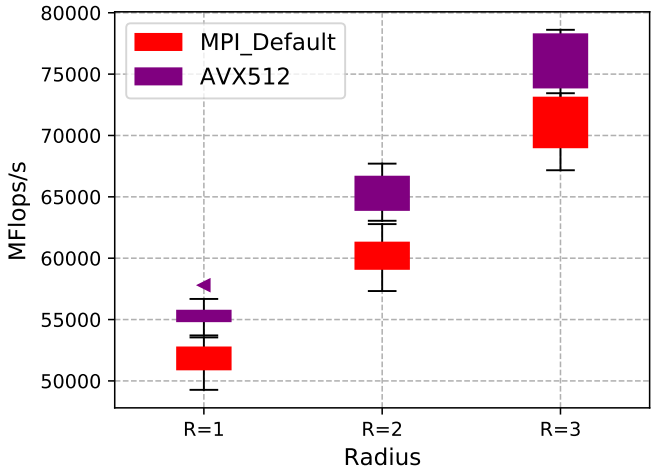


Fig. 10: 2-d Stencil results with and without AVX-512 gather pack and scatter unpack for different radius

cation benchmark is based on the Stencil MPI implementation from Parallel Research Kernels (PRK) [28], [29] – are a suite of simple kernels for the study of the efficiency of distributed and parallel computer systems, including all software and hardware components that make up the system. They cover a wide range of common parallel application patterns, especially from the area of HPC. This stencil implementation uses a for loop with memory copy function to pack the chunks for the non-contiguous data from east-west boundary. We optimized this packing implementation with the vector representation we described above.

We conducted our experiment on the same Intel Xeon Gold6254 based cluster. Table II shows our experiment configuration for this stencil application. We executed our experiments with 16 processes, we arrange processes grid as 2*8 in x/y direction. The stencil is a five points stencil using single precision executed for 100 iterations. We demonstrate the effectiveness of OPTIMIZED OPEN MPI with three tests using different radius. As demonstrated in Figure 10 we compared the performance of two implementations. We see that our gather and scatter optimized implementation decreases the packing and unpacking cost for communication during each computation step. Consequently, we improved collective operation that drives up the overall application performance by 10% for all three cases.

B. 2D Fast Fourier Transform

The Fast Fourier Transform (FFT) is one of the most significant algorithms for exascale applications in science and engineering across various disciplines. Applications range from image analysis and signal processing to the solution of partial differential equations through spectral methods. Also, diverse parallel libraries exist that rely on efficient FFT computations, particularly in particle applications ranging from molecular dynamics computations to N-Body simula-

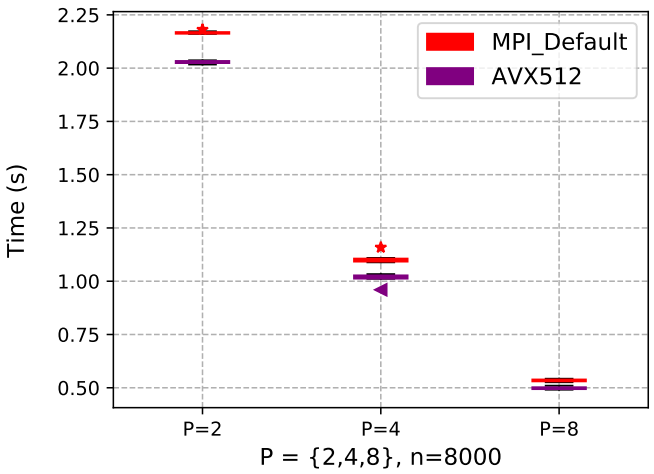


Fig. 11: 2-d FFT results with and without AVX-512 gather pack and scatter unpack for different number of processes

tions. Thus, for all these applications, it is essential to have access to a fast and scalable implementation of an FFT algorithm, an implementation that can take advantage of efficient communication library and component, and maximize these benefits for applications. We aim to expand and demonstrate the performance advantages we integrated into OPEN MPI that can benefit all-to-all communication in FFT.

An FFT on multidimensional data can be performed as a sequence of one-dimensional transforms along each dimension. For example, a two-dimensional FFT can be computed by performing 1d-FFTs along both dimensions. With multiple MPI processes, after each process computes the 1d-FFT, a matrix transpose needed to be performed among MPI processes using MPI_Alltoall operation. Also n-d FFTs can be computed by performing 1-d FFTs in all n dimensions. In this subsection, we examine the performance of our gather pack and scatter unpack approach, we measured the performance (running time) of a micro-benchmark: 2D FFT Benchmark with code version [30]. More details about the implementation can be found in this paper [31]. To our interest of region, in this implementation it uses a vector type for the all to all communication. We compare the performance of this all to all collective between MPI default and our proposed optimized design.

Figure 11 shows the performance comparison of the 2D FFT application completion time between our AVX-512 enabled pack and unpack operation and the default operation in OPEN MPI. We tested with a different number of processes as show in X-axis. The number of elements in each dimension is 8000. We can see that we achieve 8% performance speedup under all three cases for the entire completion time.

VI. CONCLUSION

In this paper, we presented the benefits of using the new features from Intel AVX-512 long vector extension. We evaluated the performance advantages of the gather and scatter feature to load and store non-contiguous data layout. Furthermore, based on our investigation and analysis, we designed and implemented an optimistic MPI optimization.

We introduced a new packing and unpacking strategy in the Datatype engine under OPAL level in OPEN MPI using AVX-512 intrinsic to speedup the communication for a non-contiguous datatype. We demonstrated the efficiency of our new pack and unpack operation by a benchmark (**to_self**) in OPEN MPI repository. For a vector type ($blocklen = 2, gap = 1$), we achieve $2.3 \sim 3.5$ times speedup with gather pack and 3.5 times speedup with scatter unpack. We also investigated gather and scatter across cache lines and achieved good performance. To further validate the performance improvements, we conducted experiments with two applications: five-point Stencil and 2D-FFT. Our proposed designs outperforms default OPEN MPI by 10% and 8%, respectively.

Our analysis and implementation of OPEN MPI optimization provide useful insights and guidelines on how novel long vector features could be used in high-performance computing platforms and software to improve the efficiency of parallel libraries and applications.

VII. ACKNOWLEDGMENTS

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] S. Maleki, Y. Gao, M. Garzarán, T. Wong, and D. Padua, "An evaluation of vectorizing compilers," 10 2011, pp. 372–382.

[2] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: Past, present and future," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 425–432. [Online]. Available: https://doi.org/10.1145/277830.277935

[3] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016.

[4] Intel. (2019) 64-ia-32-architectures instruction set extensions reference manual. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[5] ARM. (2018) Arm architecture reference manual armv8, for armv8-a architecture profile. [Online]. Available: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[6] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 608–621. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837615

[7] M. Boettcher, B. M. Al-Hashimi, M. Eyole, G. Gabrielli, and A. Reid, "Advanced SIMD: Extending the reach of contemporary SIMD architectures," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–4.

[8] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Stencil codes on a vector length agnostic architecture," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 13:1–13:12. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243192

[9] Fujitsu. (2021) A64fx microarchitecture manual v1.4. [Online]. Available: https://github.com/fujitsu/A64FX/

[10] M. P. I. Forum. (November 15,2020) MPI: A Message-Passing Interface Standard Version 4.0 (draft). [Online]. Available: https://www.mpi-forum.org

[11] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the intel knl processor with avx-512," *Cluster Computing*, vol. 21, no. 4, pp. 1785–1795, Dec 2018.

[12] M. G. F. Dosanjh, W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Gazimirsaeed, and A. Afsahi, "Fuzzy Matching: Hardware Accelerated MPI Communication Middleware," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2019, pp. 210–220.

[13] G. Santhanaraman, J. Wu, and D. K. Panda, "Zero-Copy MPI Derived Datatype Communication over InfiniBand," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 47–56.

[14] A. Gainaru, R. L. Graham, A. Polyakov, and G. Shainer, "Using InfiniBand Hardware Gather-Scatter Capabilities to Optimize MPI All-to-All," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 167–179. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966918

[15] A. Armejach, H. Caminal, J. M. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Using Arm's scalable vector extension on stencil codes," *The Journal of Supercomputing*, Apr 2019.

[16] D. Zhong, P. Shamis, Q. Cao, G. Bosilca, S. Sumimoto, K. Miura, and J. Dongarra, "Using arm scalable vector extension to optimize open mpi," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 222–231.

[17] J. M. Hashmi, C.-H. Chu, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Falcon-x: Zero-copy mpi derived datatype processing on modern cpu and gpu architectures," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 1–13, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731520302872

[18] M. Hofmann and G. Rünger, "MPI Reduction Operations for Sparse Floating-point Data," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 94–101.

[19] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and J. Dongarra, "GPU-Aware Non-contiguous Data Movement In Open MPI," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 231–242. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907317

[20] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, and J. Dongarra, "Han: a hierarchical autotuned collective communication framework," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 23–34.

[21] D. Zhong, Q. Cao, G. Bosilca, and J. Dongarra, "Using advanced vector extensions avx-512 for mpi reductions," ser. EuroMPI/USA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–10. [Online]. Available: https://doi.org/10.1145/3416315.3416316

[22] M. Bayatpour, S. M. Ghazimirsaeed, S. Xu, H. Subramoni, and D. K. Panda, "Design and characterization of infiniband hardware tag matching in mpi," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 101–110.

[23] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving mpi multi-threaded rma communication performance," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225114

[24] T. Schneider, R. Gerstenberger, and T. Hoefler, "Micro-Applications for Communication Data Access Patterns and MPI Datatypes," in *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, vol. 7490. Springer, Sep. 2012, pp. 121–131.

[25] D. Zhong, A. Bouteiller, X. Luo, and G. Bosilca, "Runtime level failure detection and propagation in hpc systems," in *Proceedings of the 26th European MPI Users' Group Meeting*, ser. EuroMPI '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3343211.3343225

[26] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[27] (2021) Open mpi main development repository. [Online]. Available: https://github.com/open-mpi/ompi

[28] (2019) Parallel research tools. [Online]. Available: https://github.com/ParRes/Kernels

[29] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[30] T. Hoefler. Mpi derived datatype (benchmark) page: 2d fft benchmark. [Online]. Available: http://unixer.de/research/datatypes/

[31] T. Hoefler and S. Gottlieb, "Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using mpi datatypes," ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 132–141.