Using Advanced Vector Extensions AVX-512 for MPI Reduction

Dong Zhong
dzhong@vols.utk.edu
The University of Tennessee
Knoxville, TN, USA

Qinglei Cao
qcao3@vols.utk.edu
The University of Tennessee
Knoxville, TN, USA

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

Jack Dongarra
dongarra@icl.utk.edu
The University of Tennessee
Knoxville, TN, USA

Abstract

Hardware platforms in high performance computing are constantly getting more complex to handle and satisfy the increasing computational need. This brings new challenges and opportunities to the design of software and library, especially with regard to MPI libraries. Numerous features and configuration options from novel architectures and processors with long vector extension becomes much more important to exploit the potential peak performance. Novel processor architectures, such as, Intel AVX-512 architecture introduced 512-bit Advanced Vector Extensions (SIMD) instructions for x86 instruction set architecture (ISA). These new features allow for better compliance with long vector gather load and scatter store.

In this paper, we propose new optimized strategies by utilizing AVX-512 gather and scatter feature to improve the packing and unpacking operations for non-contiguous memory access. With these optimizations, we not only provide a higher-parallelism for a single node, but also 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 larger range of programming and execution paradigms.

The evaluation of the resulting software stack suggests our design significantly improves the default Open MPI and achieves decent speedups against state-of-the-art MPI implementations on tested benchmark and applications.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; *Heterogeneous (hybrid) systems*; *Reliability*; *Fault-tolerant network topologies*; • **Software and its engineering** → *Software fault tolerance*; *Runtime environments*.

Keywords

OpenMPI, Intel AVX-512, Single Instruction Multiple Data, Long Vector Operation, MPI Pack and Unpack, Gather and Scatter

ACM Reference Format:
Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. 2019. Using Advanced Vector Extensions AVX-512 for MPI Reduction. In *26th European MPI Users' Group Meeting (EuroMPI 2019), September 11–13, 2019, Zürich, Switzerland*. ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3343211.3343225

1 Introduction

The need to satisfy the scientific computing community's increasing computational demands leads to larger HPC systems with more complex architectures, which provides more opportunities to enrich multiple levels of parallelism. Lots of researchers are focuing on exploiting data level parallelism by vector execution and code vectorization [?] which leads HPC systems to equip with vector processors. Comparing to traditioanal scalar processors, extension vector processors support Single Instruction Multiple Data (SIMD) and more powerful instructions operate on vectors with multiples elements involved rather than single element which could achieve maximum computational power.

There are efforts to keep improving the vector processors by increasing the vector length and adding new vector instructions. Intel starts from most vector integer SSE and AVX instructions, then expand to Haswell instructions as 256 bits (AVX2), the more advanced processor Knights Landing [?] introduced AVX-512 [?] support 512-bit wide SIMD registers (ZMM0-ZMM31) as in Fig**??**. 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;

These instructions represent a significant leap to 512-bit SIMD support. Programs can pack eight double precision or sixteen single precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors. This enables processing of twice the number of data elements that Intel AVX/Intel AVX2 can process with a single instruction and four times that of SSE. Intel AVX-512 instructions offer the highest degree of compiler support by including an unprecedented level of richness in the design of the instructions. Intel AVX-512 operations on packed floating point data or packed integer data, embedded rounding controls (override global settings), embedded broadcast, new operations, additional gather/scatter support, high speed math instructions, compact representation of large displacement value, and the ability to have optional capabilities beyond the foundational capabilities.

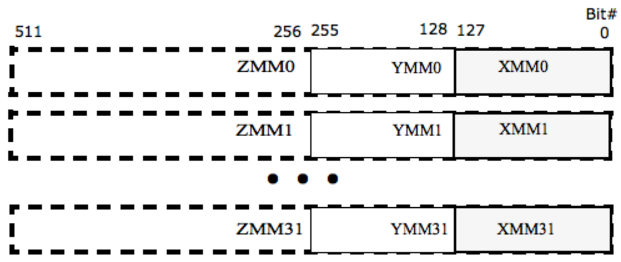

Figure 1: AVX512-Bit Wide Vectors and SIMD Register Set

AVX-512 not only takes advantage of using long vectors but also enables powerful high vectorization features that can achieve significant speedup. Those features include but not limit to:

(1) providing a valuable set of horizontal reduction operations which applies to more types of reducible loop carried dependencies including both logical, integer and floating point of high speed math reductions;
(2) and permitting vectorization of loops with more complex loop carried dependencies and more complex control flow.

Arm announced new Armv8 architecture embraced SVE- a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture [? ?]. 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 [? ?]. Vector length constrains in the range from a minimum of 128 bits up to a maximum of 2048 bits in increments of 128 bits.

Message Passing Interface (MPI) [?] is a popular and efficient parallel programming model for distributed memory systems widely used in scientific applications. As many scientific applications operate on lagre amount of data, manipulating and opreating these data becomes complicated. Especially for machine learning applications running on distributed systems, processes need to use reduction operations for very large data sets to synchronize updating the weights matrix.

Computation-oriented collective operations like MPI_Reduce perform reductions on data along with the communications performed by collectives. These collectives normally require intensive CPU compute resources, which force the computation 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. We tackle the above challenges and provide designs and implementations for reduction operations which are most commonly used by computation intensive collectives - MPI_Reduce, MPI_Reduce_local, MPI_ALLreduce. We propose extensions to multiple MPI reduction methods to fully take advantage of the AVX-512 capabilities such as vector product to efficiently perform these operations.

This paper makes the following contributions:

(1) analyzing AVX-512 hardware arithmetic instructions to speedup variety types of reduction operations and optimizing MPI local reduction operations using related intrinsics which highly increased the performance;
(2) and performing experiments using our new reduction operations in the scope of Open MPI on an local cluster comparising Intel processors. Experiment results demonstrate the efficiency of AVX-512 mathematical instructions and our implementation. Further more, provides useful insight and guideline on how vector ISA can be used in high performance computing platforms and softwares.

The rest of this paper is organized as follows. Section **??** presents related researches taking advantage of AVX-512 and SVE for specific mathematics applications, together with a survey about optimizations of MPI that taking advantages of novel hardwares. Section **??** describe the implementation details of our optimized reduction methods in the scope of AVX-512 intrinsics and instructions. Section **??** describes the performance difference between Open MPI and AVX-512 optimized Open MPI and provides a distinct insights on the how the new vector instructions can benefit MPI.

2 Related Work

In this section, we survey related work on techniques taking advantages of advanced hardwares or architectures. Lim [?] explores 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. [?] took the advantage of using AVX vector operation for MPI message matching to accelerate matches which demonstrated the efficiency of long vectors. This work [?] presented a new zero-copy scheme to efficiently implement datatype communication over InfiniBand scatter gather work to optimize non-contiguous point to point communication. Mellanox's InfiniBand [?] 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 [?], they leverage 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. However, all those work focus on using new instructions for a specific application. This work [?] explores the usage and performance of SVE vector instructions to optimize memory access operation and reduction operation. Hashmi [?] proposes a Fast and Low-overhead Communication framework for optimized zero-copy intra-node derived datatype communication on emerging CPU/GPU architectures. Michael [?] 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 [?] proposed GPU datatype engine which offloads the pack and unpack work to GPU in order to take advantage of GPU's parallel capability to provide high efficiency in-GPU pack and unpack. Luo [?] presents a new hierarchical autotuned collective communication framework in Open MPI, which selects suitable homogeneous collective communication modules as submodules for each hardware level, uses collective operations from the submodules as tasks, and organizes these tasks to perform efficient hierarchical collective operations. Zhong [?] explores the usage of long vector extension for reduction operation in MPI, which speedups the local computation in MPI that directly benefit the overall performance of collective reductions for applications. Bayatpour [?] proposes 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 [?] describes 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 offer an evaluation that demonstrates scaling to 524,288 cores.

In our work we study AVX-512 enabled features in a more comprehensive way, and also provides detailed analysis about the efficiency achievements of using related intrinsics. Our optimization is more general at processor instruction level which is more straightforward and has no limitation of data representation and is using CPU resources only without the need of external or extra hardwares.

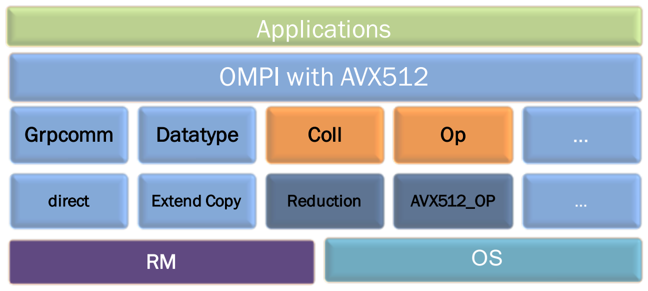

Figure 2: Open MPI architecture. The orange boxes represent components with added AVX-512 reduction features. The dark blue colored boxes are new modules.

3 Design and implementation in OMPI

We implemented AVX512 reduction operation work in a set of components in OMPI which is based on a Modular Component Architecture [?] that permits easily extending or substituting the core subsystem with new features. As shown below, we added our AVX512 optimization work in a components to OMPI architecture that implements all MPI reduction operations with AVX512 vector reduction instructions as in fig??; also we integrate our new module to automatically detect the hardware information to enable the AVX-512 reduction feature or fallback to the default basic module if it is not supported by the processor as show in fig??. To be noted, this component can be extend out the scope of local reduction to general mathematics and logic operations.

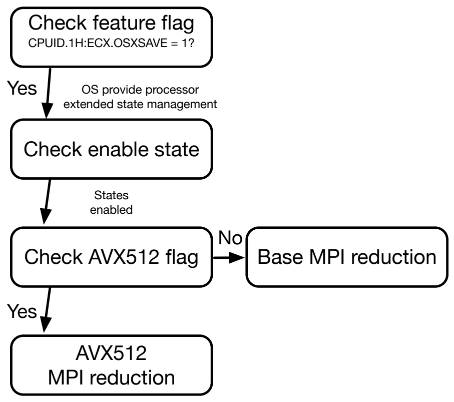

Figure 3: Integrate and automatically activate the AVX component into the OMPI build system

3.1 Memory access pattern

Memory operation is the most widely used operation during communication including point-to-point and collectives. Data need to be packed at the sender side before sending and to 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 show in Figure ??.

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

For **non-contiguous** datatype layouts as show in Figure ??, vector type Fig ??(b) is the most regular and certainly the most widely used MPI datatype constructor. Vector allows replication of a datatype into locations that consist of equally spaced blocks, describing the data layout by 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 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 Fig ??(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 Fig ??(c) cannot be described in a concise format using only block-length and stride.

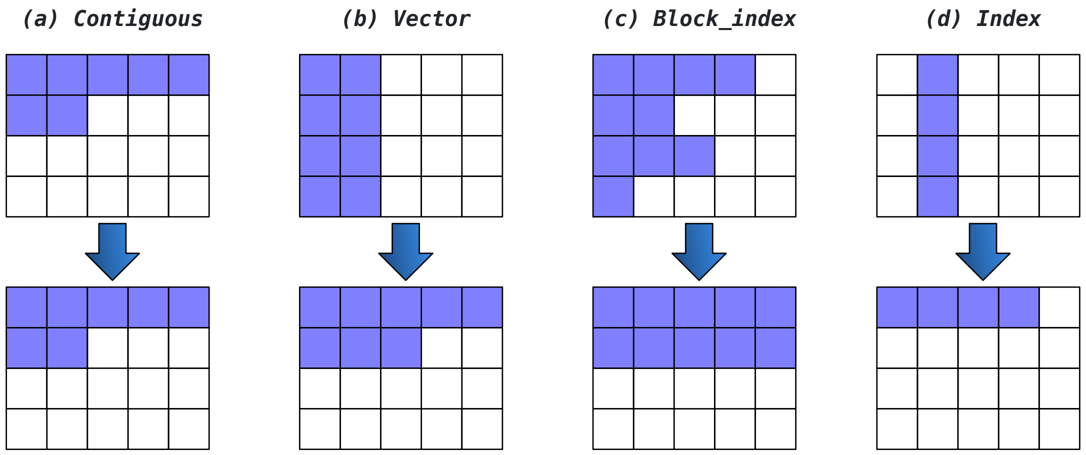

Figure 4: Memory layout of datatype (contiguous and non-contiguous)in MPI

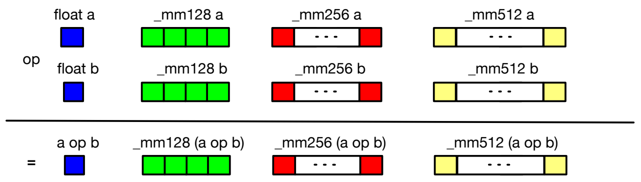

Figure 5: Example of single precision floating-point values using : (🟦) scalar standard C++ code, (🟩) AVX SIMD vector of 4 values , (🟥) AVX2 SIMD vector of 8 values, (🟨) AVX512 SIMD vector of 16 values

A reduction is a common operation found in many scientific applications. Those applications have large amounts of data level parallelism and should be able to benefit from SIMD support for reduction operation. Traditional reduction operation performs element by element of the input buffer which executes as a sequential operation or it is possibly could be vectorized under particular circumstance. Sometimes it may suffer from dependencies across multiple loop iterations. As show in Figure ?? illustrates the difference between a scalar operation and a vector operation for AVX, AVX2 or AVX512 respectively. An AVX512 SIMD-vector is able to store 8 double precision floating point numbers or 16 integer values, for example. AVX-512 reduction instructions perform arithmetic horizontally across active elements of a single source vector and deliver a scalar result. AVX-512 provides arithmetic reduction operation for integer and float-pointing, also supports logical reduction operations for integer type, an example format would be *__m512i _mm512_add_epi32 (__m512i a, __m512i b)* this function produces the add results of two vectors. This gives the chance to create AVX-512 intrinsic reduction in MPI which will highly increase the parallelization and performance of MPI local reduction. Also AVX-512 can performs scatter reduction operation with the accomplished support of predicate vector register which behaves in a vectorized manner. This highly expands the limitation of consecutive memory layout for reduction operation to non-contiguous at the same time generic and efficient.

Algorithm 1 AVX based reduction algorithm

```
types_per_step    ▷ Number of elements in vector
left_over         ▷ Number of elements waiting for reduction
count             ▷ Total number of elements for reduction operation
in_buf            ▷ Input buffer for reduction operation
inout_buf         ▷ Input and output buffer for reduction operation
1: procedure ReductionOp( in_buf, inout_buf, count )
2:     types_per_step = vector_length(512) / (8 × sizeof_type)
3:     for k ← types_per_step to count do
4:         _mm512_loadu_si512 from in_buf
5:         _mm512_loadu_si512 from inout_buf
6:         _mm512_reduction_op
7:         _mm512_storeu_si512 to inout_buf
8:     if ( left_over ≠ 0 ) then
9:         Update types_per_step >>= 1
10:        if ( types_per_step ≤ left_over) then
11:            _mm256_loadu_si256 from in_buf
12:            _mm256_loadu_si256 from inout_buf
13:            _mm256_reduction_op
14:            _mm256_storeu_si256 to inout_buf
15:    if ( left_over ≠ 0 ) then
16:        Update types_per_step >>= 1
17:        if ( types_per_step ≤ left_over) then
18:            _mm_lddqu_si128 from in_buf
19:            _mm_lddqu_si128 from inout_buf
20:            _mm128_reduction_op
21:            _mm_storeu_si128 to inout_buf
22:    if (left_over ≠ 0 ) then
23:        Duff device
```

For our optimized reduction opreation we use multiple methods try to achieve the most optimal performance as show in algorithm??. For the main for-loop section we explicitly use AVX-512 bits vector loads and stores for memory operation instead of using memcpy. Because some systems and compilers may not provide the best assembling techniques of using ZMM registers to load and store.

Table 1: Supported types and operations

Types	uint8 - uint64	float	double
MAX	✓	✓	✓
MIN	✓	✓	✓
SUM	✓	✓	✓
PROD	✓	✓	✓
BOR	✓	–	–
BAND	✓	–	–
BXOR	✓	–	–

Table 2: Supported CPU flags

Instruction Sets	CPU flags			
AVX	AVX512BW	AVX512F	AVX2	AVX
SSE	SSE4	SSE3	SSE2	SSE

And the use vector mathematic operation to perform on those wide vectors. Handling of the remainder we automatic use YMM registers processing elements that fit in the 256 bits registers. For the last part of the remainder we use Duff's device which we manually implementing loop unrolling by interleaving two syntactic constructs of C: the do-while loop and a switch statement which helps the compiler to correctly optimize the device; it may also interfere with pipelining and branch prediction on some architecture. Table**??** shows the variety of MPI_Types and MPI_Ops are supported in our optimiezed reduction operation module. We can see our implenmentation supprts all combination of all types and operations in MPI stardard. Table**??** lists the supported x86 instruction set architectures and related CPU flags from legacy SSE to the latest AVX512 instruction sets.

4 Performance tool evaluation

In the section, we study the benefits of our AVX-512 intrinsics enabled Open MPI by using PAPI [?] – tool that can measure application performance in these increasingly complex environments must also increase the richness of their measurements to provide insights into the increasingly intricate ways in which software and hardware interact. PAPI (the Performance API) has provided consistent platform and operating system independent access to CPU hardware performance counters.

Figure **??** and figure **??** illustrate the instruction count details of branching instructions and load/store instructions of both AVX512 implementation and the default element-wise reduction method. By using long vectors we largely decreased the "for loop" of the reduction operaion. Consequently, the AVX512 code has much less control and branching instructions. For the load and store instruction, longer vector can load and store more elements for each instruction compared to non-vector load and store, which means that we neeed less load and store instrutions dealing with the same amount of reduction data.

5 Experimental evaluation

Experimented on a local cluster which is a Intel(R) Xeon(R) Gold 6254 based server running at 3.10 GHz. Our work is based upon OMPI master branch, revision #406bd3a. Each experiment is repeated 30 times and we present the average. For all experiment we use a single node with 16 process.

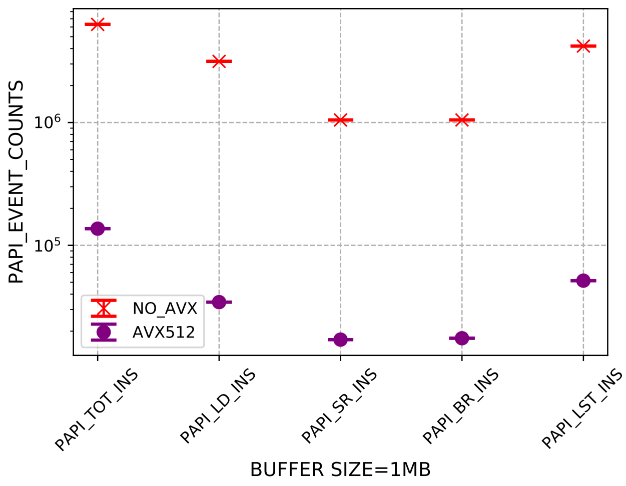

Figure 6: MPI_SUM with AVX-512 enable and disable with PAPI instruction events overview

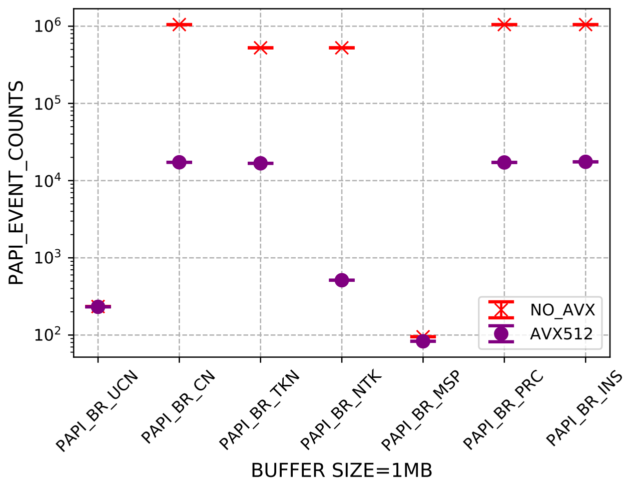

Figure 7: MPI_SUM with AVX-512 enable and disable for PAPI branch instructions

This section compares the performance of reduction operation with two implementations. For Open MPI default operation base module it performs element wise reduction operation across two input buffers. For each loop iteration it processes two elements. Our new implementation we use AVX-512 vector reduction instruction executing reduction operation on the same inputs but for each iteration it deals with two vectors containing all the elements within the vectors which represents a vector-wise operation. For the reduction benchmark we use the MPI_Reduce_local function call to perform the local reduction for all supported MPI operations using an array with different sizes.

We present to compare arithmetic SUM and logical BAND. For the experiments we flushed cache to ensure we are not reusing cache for fair comparison.

Figure **??** and figure **??** show the result for the MPI_SUM and MPI_BAND, due to the limited length of the paper, we cannot include the assemble code here. But it should be noted that the compiler, despite the optimization flags provided, did not generate auto-vectorized code for the default Open MPI. Our optimization using intrinsics which gives us completely control of the low level details at the expense of productivity and portability.

Results demonstrate that with AVX512-enabled operation it is 10X faster than element-wise operation. We also compare MPI operation together with memcpy which indicates the peak memory bandwidth. For mpi reduction operation it needs 2 loads, 1 store and additional computation. For memcpy it only needs 1 load and 1 store. The result shows even with computation included AVX512 reduction operation achieves a similar level of memory bandwidth as memcpy.

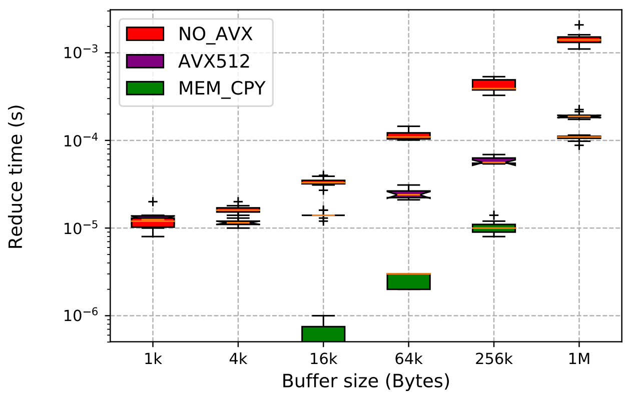

Figure 8: Comparison of MPI_SUM with AVX-512 reduction enable and disable for MPI_SUM together with memcpy

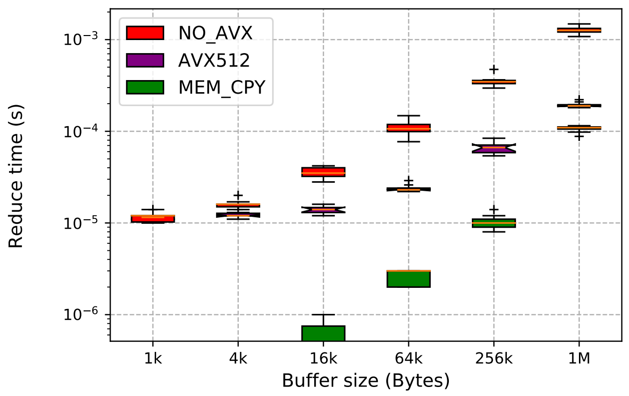

Figure 9: Comparison of MPI_BAND with AVX-512 reduction enable and disable for MPI_SUM together with memcpy

6 Application Evaluation

Over the past few years, advances in deep learning have driven tremendous progress in image processing, speech recognition, and forecasting. Currently, one of the significant challenges of deep learning is it is a very time-consuming process. Designing a deep learning model requires design space exploration of a large number of hyper-parameters and processing big data. Thus, accelerating the training process is critical for our research and development. Distributed deep learning is one of the essential technologies in reducing training time. In this section we investigate an application Horovod [?] - an open-source component of Michelangelo's deep learning toolkit which makes it easier to start and speed up distributed deep learning projects with TensorFlow.

The important aspect to understand is that in deep learning it needs to calculate and update the gradient in order to be able to adjust the weights. Without this learning can't happen. In order to calculate that gradient, it needs to process all of the data which is normally very large. When such data is too big it needs to parallelize these calculations. This means that it will have distribted computing nodes working in parallel on a subset of the data. When each of these processing units or workers (they could be CPUs, GPUs, TPUs, etc.) is done calculating the gradient for its subset, they then need to communicate its results to the rest of the processes involved. Actually, every process/node needs to communicate its results with every other process/node.

Horovod utilize Open MPI to launch all copies of the TensorFlow program. MPI then transparently sets up the distributed infrastructure necessary for workers to communicate with each other. All the user needs to do is modify their program to average gradients using an Allreduce operation. Conceptually Allreduce has every process share its data with all other processes and applies a reduction operation. This operation can be any reduction operation, such as sum, multiplication, max or min. In other words, it reduces the target arrays in all processes to a single array and returns the resultant array to all processes. Horovod uses a ring-allreduce approach as an optimal method in the sense of both usability and performance. Ring-allreduce utilizes the network in an optimal way if the tensors are large enough, but does not work as efficiently or quickly if they are very small. Horovod introduced Tensor Fusion - an algorithm that fuses tensors together before it calls ring-allreduce. The fusion method allocates a largde fusion buffer and execute the allreduce operation on the fusion buffer. In the ring-allreduce algorithm, each of N nodes communicates with two of its peers 2 $*$ (N - 1) times. During this communication, a node sends and receives chunks of the data buffer. In the first $N - 1$ iterations, received values are added to the values in the node's buffer. In the second $N - 1$ iterations, when each process receives the data from the previous process it'd then apply the reduce operator, and then proceeds to send it again to the next process in the ring which is bandwidth optimal [?]. We can see that during the allreduce processing phase, $P *$ (N - 1) reduction operations occured with big fusion buffer size. Consequently, our AVX512 enabled reduction operations can particularly improve the performance of the collective operation.

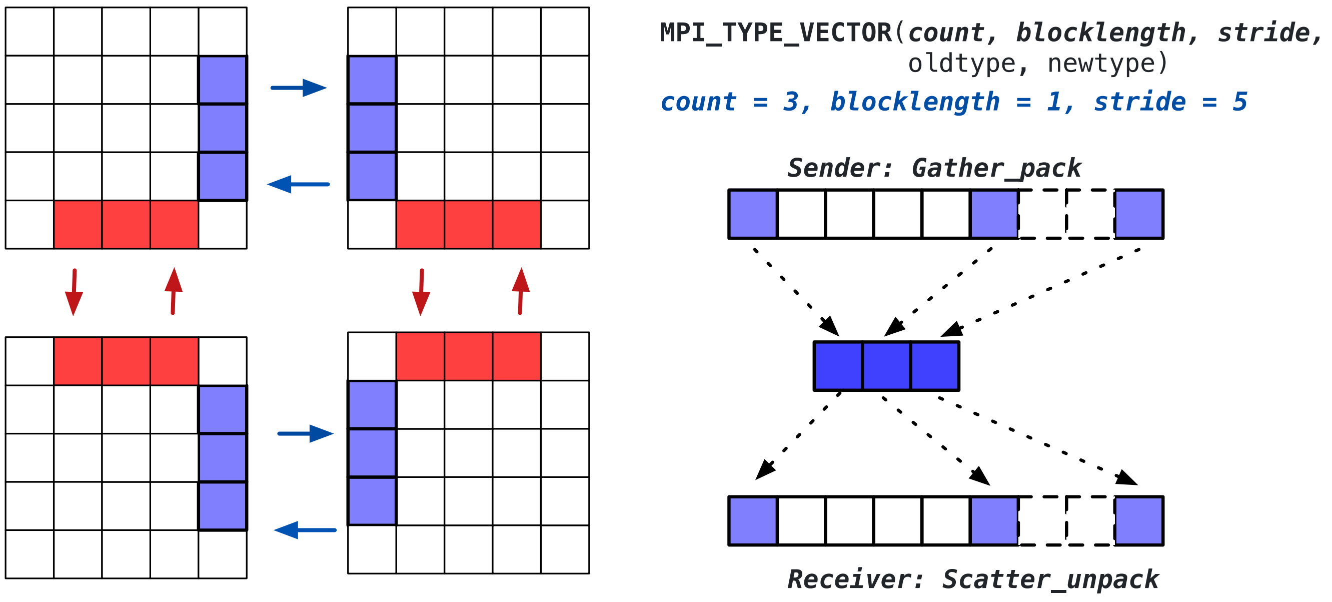
MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)
count = 3, blocklength = 1, stride = 5
Sender: Gather_pack
Receiver: Scatter_unpack

Figure 10: Domain-decomposed 2D stencil. Data exchanged in east-west direction must be packed and unpacked in communication

We conducted our experiments on Stampede2 with Intel Xeon Platinum 8160 ("Skylake") nodes, each node has 48 cores. We experimented with TensorFlow CNN benchmarks using Horovod with tensorflow-1.13.1.

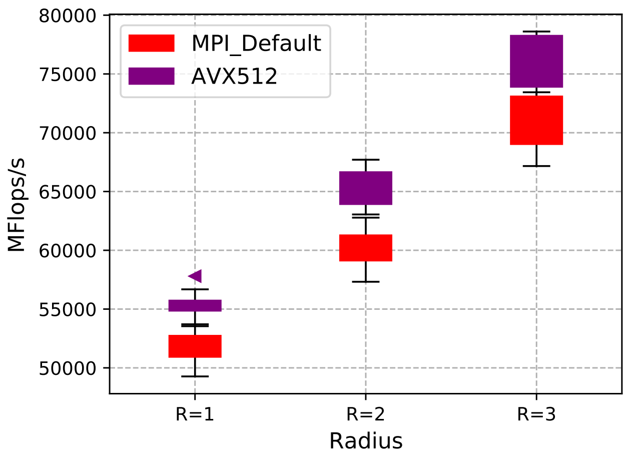

Figure 11: 2-d Stencil results with and without AVX512 gather pack and scatter unpack for different radius

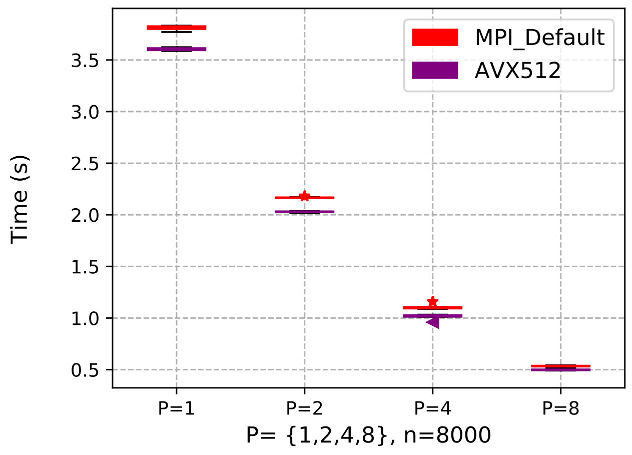

Figure 12: 2-d FFT results with and without AVX512 gather pack and scatter unpack for different number of processes

6.1 Fast Fourier Transforms (FFTs)

The Fast Fourier Transform (FFT) is one of the most significant algorithms across various disciplines in science and engineering. Applications range from image analysis and signal processing to the solution of partial differential equations through spectral methods. It is well known that 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 compute the 1d-FFT, a matrix transpose needed go be performed among MPI processes normally 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 [?]. More details about the implmentation can be found in this paper [?] We compare the performance of MPI_default with the proposed optimized design.

Figure ?? shows the performance comparison of 2D FFT Benchmark completion time with our AVX512 enabled pack and unpack operation and the default operation in Open MPI.

Nowadays, more and more systems in HPC feature a hierarchical hardware design: Shared memory nodes with several multi-core CPUs are connected via a network infrastructure. This trend has disrupted the long status-quo in which parallel applications are written in MPI, and has promoted the emergence of multiple alternatives for programming parallel systems.

On one hand, one may encounter programming styles that combines distributed memory parallelization between nodes separated by the interconnect, and shared memory parallelization, or GPU acceleration inside each node. On the other hand, parallel applications may alternate between library calls that utilize different programming environments and programming models to perform internode communication, for example, message passing and parallel global address space models may coexist in the same application. Consequently, runtime environment needs to handle the cooperations between different programming models. Together, failure detection and management techniques need to be expanded across different models.

In this section we investigate application support of with different programming models. Our interests focus on 1) demonstrating how our generic capabilities can support multiple programming languages, 2) how much overhead (if any) is incurred on both two-sided (e.g., MPI) and one-sized programming models (e.g., OpenSHMEM), and 3) provide the blueprints for supporting applications using multiple programming models. In hybrid applications and models, there is no “standard” method by which programming models can coordinate. For example, MPI has the standard MPI_Init function that must be called to initialize the library – providing a "hook" within that function to notify others that it has been called. In contrast, OpenMP does not have an explicit call to "init" and is instead initialized on first use; older versions of OpenSHMEM also allow implicit initialization. Figure ?? shows how to coordinate between two different models. We can see that as both communication libraries employ the PMIx library to interface with the runtime and job scheduling system, the different programming languages have a common interface to exchange information. The calls into PMIx_Init from each programming model enters the same code space and offers an opportunity for coordination. The event notification mechanism within PMIx can then be used to share the information and coordinate between those models.

To evaluate the overhead on performance from in MPI and OpenSHMEM applications, we use the heavily communication-bound benchmark Graph500 [?]. Graph500 is an open specification effort to offer a standardized graph-based benchmark across large-scale distributed platforms which captures the behavior of common communication-bound graph algorithms. Graph500 differs from other large-scale benchmarksm such as HPL, and HGPGMG in the way it primarily highlights data access patterns. Graph500 performs a breadth-first search (BFS) in parallel on a large randomly generated undirected graph. Our experiments use a the open source project OpenSHMEM Benchmark (OSB) suite [?] that features both MPI and OpenSHMEM based Graph500 implementations. For the application setting we use $scale\_factor = 20$, $edge\_factor = 16$ which generates an undirected graph with 2^{scale_factor} vertices and $2^{scale\_factor} * edge\_factor$ edges. The benchmark collects the statistics of the generation of the breadth-first search tree of 64 randomly selected vertices. It also collect the statistics of validation time which ensures that all connected components are visited which generate large amount of communications. For the experiments, we use NERSC Cori with 1K nodes. This results in a deployment with 32K MPI ranks, or 32K OpenSHMEM Processing Elements (PEs).

7 Conclusion

In this paper, we demonstrated the benefits of Intel AVX512f and AVX2 vector operations. We addressed the performance advantages of different features introduced by AVX with longer vector length compared to non-AVX implementations. Furthermore, we extended the implementation of our investigation and analysis to introduced an optimistic MPI optimization. We introduced a new reduction operation module in Open MPI using AVX512 intrinsics supporting different kinds of MPI reduce operations for multiple MPI types. We demonstrated the efficiency of our vector reduction operation by a benchmark calling MPI_Local_reduce. From $VL = 128$ to $VL = 2048$ bits we decreased the instruction count from 50% to 30×. To further validate the performance improvements, experiments are conducted using Fujitsu’s A64FX processor. For MPI_Local_reduce with $VL = 512$ SVE based reduction operation is 4× faster. Our analysis and implementation of Open MPI optimization provides useful insights and guidelines on how Arm SVE vector ISA can be used in actual high performance computing platforms and software to improve the efficiency of parallel runtimes and applications.

Acknowledgement

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.