Parallel Computing 109 (2022) 102871

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Using long vector extensions for MPI reductions

Dong Zhong [*], Qinglei Cao, George Bosilca, Jack Dongarra

The University of Tennessee, 1122 Volunteer Blvd, Knoxville, TN 37996, United States of America

ARTICLE INFO

Keywords:
Long vector extension
Vector operation
Intel AVX2/AVX-512
Instruction level parallelism
Single instruction multiple data
MPI reduction operation
Scalable Vector Extension (SVE)

ABSTRACT

The modern CPU's design, including the deep memory hierarchies and SIMD/vectorization capability have a more significant impact on algorithms' efficiency than the modest frequency increase observed recently. The current introduction of wide vector instruction set extensions (AVX and SVE) motivated vectorization to become a critical software component to increase efficiency and close the gap to peak performance.

In this paper, we investigate the impact of the vectorization of MPI reduction operations. We propose an implementation of predefined MPI reduction operations using vector intrinsics (AVX and SVE) to improve the time-to-solution of the predefined MPI reduction operations. The evaluation of the resulting software stack under different scenarios demonstrates that the approach is not only efficient but also generalizable to many vector architectures. Experiments conducted on varied architectures (Intel Xeon Gold, AMD Zen 2, and Arm A64FX), show that the proposed vector extension optimized reduction operations significantly reduce completion time for collective communication reductions. With these optimizations, we achieve higher memory bandwidth and an increased efficiency for local computations, which directly benefit the overall cost of collective reductions and applications based on them.

1. Introduction

The need to satisfy the scientific computing community's increasing computational demands drives larger HPC systems with more complex architectures. This provides more opportunities to enhance various parallelism levels. Instruction-level (ILP) and thread-level parallelism (TLP) has been extensively studied, but data-level parallelism (DLP) is usually underutilized in CPUs, despite its vast potential. While ILP's importance subsides, DLP's becomes a critical factor in improving the efficiency of microprocessors [1–5]. The most widespread vector implementation is based on single-instruction multiple-data (SIMD) extensions. Vector architectures are designed to improve DLP by simultaneously processing multiple input data with a single instruction, usually applied to vector registers. SIMD instructions have been gradually included in microprocessors, with each new generation providing more sophisticated, powerful, and flexible instructions. The higher investment in SIMD resources per core makes extracting these vector units' full computational power highly significant.

A large body of literature has focused on employing DLP via vector execution and code vectorization [6–8], transforming the way compilers generate code for these architectures. HPC, with its ever-growing demand for computing capabilities, has been quick to embrace vector processors and harness the additional compute power. As an essential factor of processors' capability to apply a single instruction on multiple data, vectorization continuously improves from one CPU generation to the next, including using increasingly larger vector registers, and gathering/scattering capabilities. Compared to traditional scalar processors, extension vector processors support SIMD and more powerful instructions operating on vectors with multiples elements. They can generate memory accesses and data computations faster by orders of magnitude. Over the last decade, the difference between a scalar code and its vectorized equivalent increased from a factor of 4 with SSE, up to a factor of 16 with AVX-512 [9–11], highlighting the importance of employing vectorized code whenever possible. The conversion of a scalar code into a vectorized equivalent can be relatively straightforward for many affine classes of algorithms and computational kernels, and can be done automatically, with little human intervention, by a compiler with auto-vectorization capabilities. The compiler can provide a baseline for more complex codes, but developers are also encouraged to offer optimized versions using widely available compilers intrinsics.

There are efforts to improve the vector processors by increasing the vector length and adding new vector instructions. For example, Intel's first version of vectorized instruction set, MMX, was quickly superseded by more advanced vector integer SSE and AVX instructions [12–14]. Later, it was expanded to Haswell instructions as 256 bits (AVX2), and followed with the arrival of the Knights Landing processor [11]. The more advanced AVX-512 [15] was introduced supporting 512-bit wide SIMD registers (ZMM0-ZMM31), as shown in Fig. 1. The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM

* Corresponding author.
E-mail addresses: dzhong@vols.utk.edu (D. Zhong), qcao3@vols.utk.edu (Q. Cao), bosilca@icl.utk.edu (G. Bosilca), dongarra@icl.utk.edu (J. Dongarra).

https://doi.org/10.1016/j.parco.2021.102871
Received 15 February 2021; Received in revised form 27 September 2021; Accepted 8 November 2021
Available online 5 December 2021

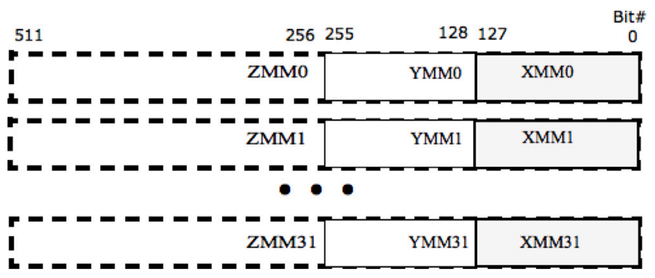

Fig. 1. AVX512-Bit wide vectors and SIMD register set.

registers, and the lower 128-bit are aliased to the respective 128-bit XMM registers. The AVX-512 features and instructions provide a significant advantage to 512-bit SIMD support. This product offers a high degree of compiler support in designing the instructions. Compared to previous architectures, AVX-512 leverages longer and more powerful registers capable of packing eight double-precision, or sixteen single-precision floating-point numbers, eight 64-bit integers, or sixteen 32-bit integers within the same 512-bit vector. It also enables processing twice the amount of data elements than Intel AVX2, and four times than SSE with a single instruction. Furthermore, AVX-512 also supports a richer set of features, such as operations on packed floating-point data or packed integer data, new arithmetic and shifting operations, additional gather/scatter support, high-speed math instructions, and the ability to have optional capabilities beyond the basic instruction set.

AVX-512 takes advantage of using long vectors and enables powerful vectorization features that can achieve significant speedup. A small subset of these features is listed below.

1. a rich addressing mode enabling non-linear data accesses for support of non-contiguous data;
2. a set of horizontal reduction operations, which apply to more types of reducible loop carried dependencies including both logical, integer, and floating-point of high-speed math reductions, and
3. the capability of vectorization of loops with complex loop carried dependencies and control flow constraints.

Similarly, Arm announced the new Armv8 architecture embracing SVE- a vector extension for the AArch64 execution mode for the A64 instruction set of the Armv8 architecture [16,17]. Unlike other SIMD architectures, SVE does not define the size of the vector registers. Instead, it provides a range of different values that permit vector code to automatically adapt to the current vector length at runtime with the feature of *Vector Length Agnostic* (VLA) programming [18,19]. Currently, vector length constrains are in the range from a minimum of 128 bits up to a maximum of 2048 bits in increments of 128 bits.

At the other end of the programming spectrum, Message Passing Interface (MPI) [20] is a popular, efficient, and portable parallel programming paradigm for distributed memory systems. It is widely used in scientific applications. The MPI standard provides a fully fledged set of communication primitives, between pairs or between groups of processes, allowing applications to precisely tailor the communication pattern to their needs. Optimized support for two-sided communications and collective communications has been beneficial for a large number of parallel applications. For example, machine learning applications running on distributed systems critically depend on the performance of MPI_Allreduce, a reduction operation for extensive data sets to synchronize updating the weights matrix.

Computation-oriented collective operations such as MPI_Allreduce and MPI_Reduce perform reductions on data along with the communications performed by collectives. These collectives typically encompass a linear, memory-bound operation, which forces the computation to become the main bottleneck and limit the overall performance of the collective implementation. The existence of advanced architectural features introduced with wide vector extension and specialized arithmetic operations, highlights a path forward toward addressing this bottleneck in MPI libraries, by providing support for such extensions via specialized reduction operators capable of extracting most of the processor's computational power.

Unlike more traditional HPC applications that embraced MPI long ago, machine learning and data science, in general, were more reticent. However, a new trend arise lately, certainly linked to the growth of the problems' size and the strain it puts on the memory bandwidth, toward an increased use of MPI for the distributed training.

As mentioned before, reduction operations with a large amount of data (from the weights on a layer of neurons) is an expensive step in the learning process. Such reduction operations in machine learning applications are commonly seen in synchronous parameter updates of the distributed Stochastic Gradient Descent (SGD) optimization [21]. This is used extensively in, for example, neural networks, linear regressions, and logistic regressions. Moreover, there are two important aspects to the use of these reduction operations: 1) the process is iterative until a convergence criteria is met, which translates to a large number of reduction operations, and 2) the amount of data involved in each reduction operation is considerable (with an extensive training model, the data could be in the hundreds of megabytes). Li's [22] work explores the performance of MPI_Allreduce algorithms and uses task-based frameworks to improve their performance. Specifically when referring to AlexNet on ImageNet [23], it is highlighted that each step needs to perform a weights reduction with an estimated size of 200MB for extensive model training. Similarly, [24] illustrates that with SparkNet, updating the weights of AlexNet, a single reduce operation takes almost 20 s, even with only five processes. While it is relatively simple to scale the number of execution processes to the thousands, the bottleneck remains the MPI_Allreduce of the gradient values at each step. Indeed, the size of the gradient being reduced is equivalent to the size of the model, and independent of the number of participating processes. When scaling to large numbers of processes, the full parameter set, commonly hundreds of megabytes, must be summed globally at each iteration, until convergence. This explains why the reduction operation dominates the overall time-to-solution in distributed neural network training, highlighting the need for an efficient reduction implementation.

Thus, it will be crucial for many applications to have access to a highly optimized version of MPI_Allreduce, and this requires addressing the challenge of improving the performance of the predefined MPI reduction operations. We address the above challenges and provide designs and implementations for most of the predefined reduction operations, which are used by MPI_Reduce, MPI_Allreduce and MPI_Reduce_local. We propose extensions to multiple MPI reduction methods to take full advantage of long vector extension capabilities to efficiently perform these operations.

This paper makes the following contributions:

1. We investigate and utilize long vector arithmetic instructions/intrinsics (AVX and SVE) to optimize and speed up various MPI reduction operations.
2. We perform experiments using the new implementations of reduction operations in the scope of Open MPI on different architectures. Different types of experiments are conducted with MPI benchmark, performance evaluation tools, and deep learning benchmark. Furthermore, our implementation provides useful insight and guidelines on how vector ISA can be used in high-performance computing platforms and software.

The rest of this paper is organized as follows: Section 2 presents related studies about the use of vectorized extensions, AVX and SVE, in software development, together with a survey about MPI optimizations making use of novel hardware. Section 3 describes the implementation details of our optimized reduction methods in the scope of Open MPI using AVXs and SVE intrinsics and instructions. Section 4 describes the performance difference between a vectorized and a non-vectorized implementation of the MPI_Allreduce collective and provides a distinct insight on how MPI implementations can benefit from these new vector instructions. Section 5 uses a performance tool on Xeon processor to details the relationship between performance and vectorized instruction counts. Section 6 illustrates the performance benefits achieved by running tests using LAMMPS benchmark. Section 7 illustrates our optimized reduction operation's performance benefits in Open MPI using a deep learning application.

2. Related work

Techniques can be classified according to the level at which the hardware supports parallelism with multi-core and multi-processor computers having multiple processing elements within a single machine. Different level of parallelization, including bit-level, instruction-level, data-level, and task parallelism, are studied. In this section, we survey related work on techniques taking advantage of advanced hardware or architectures, which focus on data-level parallelization. Novel processors and hardware architectures from vendors, such as Intel and Arm, come equipped with long vector extensions, and multiple researchers have studied the usage of those new technologies in high-performance computing with various programming models and applications.

2.1. Long vector extension

Lim [25] explored the generalized matrix–matrix multiplication (GEMM) based on a blocked matrix multiplication algorithm to improve data reuse. Using Intel AVX-512 intrinsics together with a carefully designed data prefetching and loop unrolling they optimized the blocked matrix multiplications, achieving an outstanding level of performance for the GEMM operation with single and multiple cores. Kim [26] presented an optimal implementation of single-precision and double-precision general matrix–matrix multiplication (GEMM) routines based on an auto-tuning approach with the Intel AVX-512 intrinsic functions. The implementation significantly diminished the search space and derived optimal parameter sets, including the size of submatrices, prefetch distances, loop unrolling depth, and parallelization scheme. Bramas [27] introduced a novel quicksort algorithm with a new Bitonic sort and a new partition algorithm designed for the AVX-512 instruction set, which showed superior performance on Intel Skylake in all configurations against two standard reference libraries. A little closer to MPI, Dosanjh et al. [28] proposed and evaluated a novel message matching method Fuzzy-matching to improve the point to point communication performance in MPI with multithreading enabled. The proposed algorithm took advantage of the AVX vector operation to accelerate matches and demonstrated that the benefits of vector operation are not only restricted to computational intensive operations, but can positively impact MPI matching engines. They also presented an optimistic matching scheme that uses partial truth in matching elements to accelerate matches. Intel AVX is not the only ISA to propose vectorized extensions. Similar studies have been done using Arm's new Scalable Vector Extensions (SVE). In [29], the authors leveraged the characteristics of SVE to implement and optimize stencil computations, ubiquitous in scientific computing, which showed that SVE enabled the easy deployment of optimizations like loop unrolling, loop fusion, load trading, or data reuse. Petrogalli's work [30] explored the usage of SVE vector multiply instructions to optimize matrix multiplication in machine learning algorithms. Zhong [31,32] used new features from long vector extension from different architectures to optimize Open MPI. [33] propose an optimized Sparse Matrix–Vector multiplication using long vector ISAs to handle sparse and irregular data access. All these studies focused on using the new vector instructions to improve a specific application's or a specific mathematical algorithm performance. In the work described in this paper, we provide a comprehensive study of AVX-512 usage in the context of MPI, more specifically for all supported mathematical reduction functions and provide a detailed analysis of the efficiency obtained in these context using the related intrinsics.

2.2. MPI *reduction operation*

There is a rich literature on optimizing the MPI reduction operations, covering all aspects of the software and hardware stack. Traff [34] proposed an implementation of MPI library internal reduce operators allowing MPI reduction operations to be performed on sparse input vectors, to accommodate for the sparsity of the connection weights in neuronal networks. Hofmann [35] presented a pipeline algorithm for MPI Reduce that used a Run Length Encoding scheme to improve the global reduction of sparse floating-point data. Chu [36] analyzed the limitations of the compute oriented CUDA-Aware collectives and proposed alternative designs and schemes by combining the exploitation GPU's compute capability and their fast communication path using the GPUDirect RDMA feature to alleviate these limitations efficiently. Luo [37] presented HAN, a hierarchical, architecture-aware autotuned collective communication framework in Open MPI. HAN selects suitable homogeneous collective communication modules as sub-modules for each hardware level, and uses collective operations from the sub-modules as asynchronous tasks with data dependencies, and organizes these tasks to perform efficient collective operations.

Patarasuk's work [38] investigated implementations of the MPI_Allreduce operation with large data sizes, derived a theoretical lower bound on this operation's communication time, and developed a bandwidth optimal MPI_Allreduce algorithm on tree topologies. Shan [39] proposed using idle threads on a many-core node to accelerate the local reduction computations and utilized the data compression technique to compress sparse input data for reduction. Both approaches (threading and exploitation of sparsity) helped accelerate MPI reductions on large vectors when running on many-core supercomputers.

Most of those works focus on improving the performance of collective communication either by making a better use of the network resources or by hiding the communication latency behind computation, or by specializing the reduction operator to the target application or hardware. Our long vector extension arithmetic reduction optimizations seek to be more general and take advantage of vector extensions to provide a straightforward set of predefined MPI reduction operations with no specialized or restricted data representation or operations limitation. The provided implementation supports multiple ISAs, covering most processor versions from different vendors, and supports multiple generations of vector instructions, including legacy SSE, advanced AVXs, and SVE.

3. Design and implementation

3.1. Intel Advanced Vector Extension

Intel Advanced Vector Extension 2 (Intel AVX2), is a significant improvement to Intel Architecture, and extends the previous generation of 128-bit SIMD float-point and integer instructions to operate on larger registers, 256-bit YMM registers, therefore executing twice as many operations in the same number of cycles. In addition to these extensions it adds new data manipulation primitives, such as broadcast, permute/variable-shift instructions, masked operations and instructions to fetch and store non-contiguous data elements to and from memory. Starting from the Haswell processors family, all Intel processors and microarchitectures support these 256-bit AVX2 instructions with low latency and high throughput.

Building over AVX2, Intel Advanced Vector Extensions 512 (Intel AVX-512) provides more powerful packing capabilities with longer vector length (512 bits instead of 256) allowing to encapsulating eight

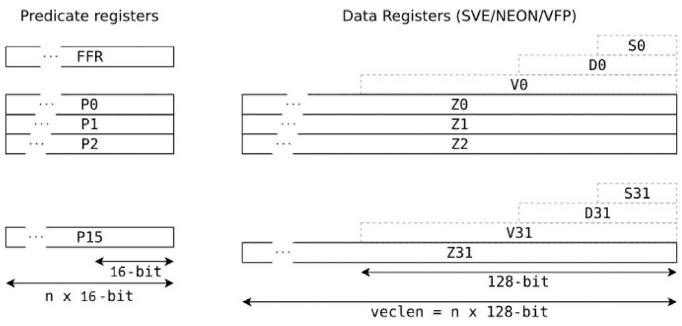

Fig. 2. Arm SVE registers.

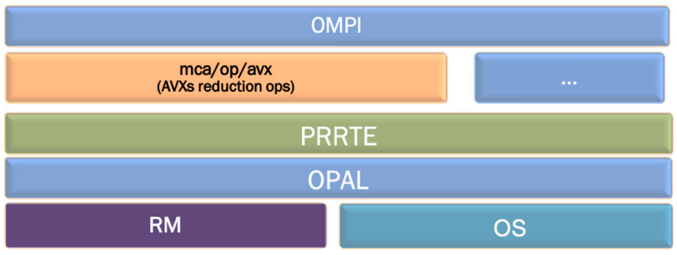

Fig. 3. Open MPI architecture. The orange box represents component with added AVX-512 reduction features.

double-precision, sixteen single-precision floating-point numbers, eight 64-bit integers, or sixteen 32-bit integers within a single vector register. The longer vector registers allow to process twice the number of data elements than what the Intel AVX/Intel AVX2 could process with a single instruction and four times than that of SSE. The larger number of vector registers (32 vector registers, each 512 bits wide, and eight dedicated mask registers), increase the opportunities for data parallelism at the processor level, providing more compute power for demanding computational tasks.

Furthermore, some performance-impacting restrictions have been lifted compared with prior version. As an example, applications using AVX and SSE instruction simultaneously suffered performance penalties, while mixinf AVX-512 instructions with any prior AVX version is supported with no penalties. AVX registers YMM0–YMM15 map into the Intel AVX-512 registers ZMM0–ZMM15, similar to how SSE registers map into AVX registers. Therefore, in processors with Intel AVX-512 support, AVX and AVX2 instructions operate on the lower 128 or 256 bits of the first 16 ZMM registers.

3.2. Arm-v8 Scalable Vector Extension

Arm introduced Scalable Vector Extension (SVE) [40] starting with the Arm-v8 architecture. As show in Fig. 2, SVE introduced 16 predicate (P) registers and 32 data (Z) registers. With the long vector extension, the new architecture supports variable vector length in the range of 128 bits up to 2048 bits. It provides support allowing the vectorized code to automatically adapt to the current vector length at runtime when using the Vector Length Agnostic (VLA) programming feature. Similar to AVX, SVE also supports the entire family of horizontal reduction instructions including integer and floating-point summation, minimum, maximum, and bit-wise logical reductions.

3.3. Intrinsics

Intrinsics are built-in functions providing a more user-friendly access to the ISA functionality, using C/C++ style coding instead of assembly language. There is a clear lack of portability at this level, each vendor defining its own set of intrinsic functions, either with full support on some compilers, or as compiler-agnostic header files. Access to these intrinsics empower programmers, providing direct access to low-level instructions and enable algorithm design and implementation where the compiler will perform the complex task of register allocation and instruction scheduling wherever possible. The use of intrinsic allows developers to obtain performance close to the levels achievable and feasible with assembly. The cost of writing and maintaining programs with intrinsics is considerably less than writing assembly code, and considerable help is provided by the compilers. The major drawback of intrinsics is their limited portability, each set of intrinsics are only portable among a specific architecture (x86 and AArch64) of processors. In summary, the intrinsic function provides the capability for SIMD instructions to be manipulated faster, more proficiently and more effectively. The following AVX-512 and SVE intrinsic functions are of interested for the effort described in this paper:

_m512i _mm512_loadu_si512 (void const* mem_addr) Load 512-bits of integer data from memory into a register. The mem_addr does not need to be aligned on any particular boundary. Generally, this intrinsic is converted into: ***vmovdqu32 zmm, m512***.

_m512i _mm512_ ⟨op⟩ _epi32 (_m512i a, _m512i b) Apply ⟨op⟩ between packed 32-bit integers in “a" and “b", and store the results in a 512-bits vector. Here we use 32-bits integer as an example. Generally, this intrinsic is converted into: ***vp ⟨op⟩ m512, m512, m512***.

_m512i _mm512_storeu_si512 (void const* mem_addr, _m512i a) Store 512-bits of integer data from “a" into memory. mem_addr does not need to be aligned on any particular boundary. Generally, this intrinsic is converted into: ***vmovdqu32 m512, zmm***.

svint32_t vsrc = svld1(svbool_t pg, void const* mem_addr) Load data from memory into a SVE long vector with predicate register.

svint32_t vsrc = svst1(svbool_t pg, void const* mem_addr, svint32_t a) Store data from “a" into memory, data length adapt automatically to the current vector length at runtime.

svint32_t sv_ ⟨op⟩_x (svbool_t pg, svint32_t a, svint32_t b) Apply ⟨op⟩ with SVE reduction intrinsic between packed 32-bit integers in “a" and “b".

3.4. Reduction operation in Open MPI

We implement our advanced reduction operations with AVX, AVX2, AVX-512 support in a component in Open MPI, based on a Modular Component Architecture [41,42] that facilitates extending or substituting Open MPI core subsystem with new features and innovations, as in Fig. 3. Open MPI architecture has three main abstraction layers: Open MPI layer (OMPI), Runtime layer (PRRTE) and Open Portable Access Layer (OPAL) We add our long vector reduction optimization in a specialized component at OMPI layer that implements all predefined MPI reduction operations with vector reduction instructions. From a practical standpoint, our module extracts the processor feature flag and check related capabilities, selecting at runtime the best set of functions supporting the most advanced ISA (AVX-512, AVX2 or AVX/SSE), or fallback to the default basic module if the processor has no support for such extensions, as shown in Fig. 4. To be more specific, we explicitly check CPUID — an instruction allowing software to discover the processor details, determine processor type, and list the supported features, such as SSE/AVXs.

Vector instructions can be integrated in applications in several manners: (a) automatic vectorization support provided by the compiler; (b) each application explicitly calls vector instructions from assembly

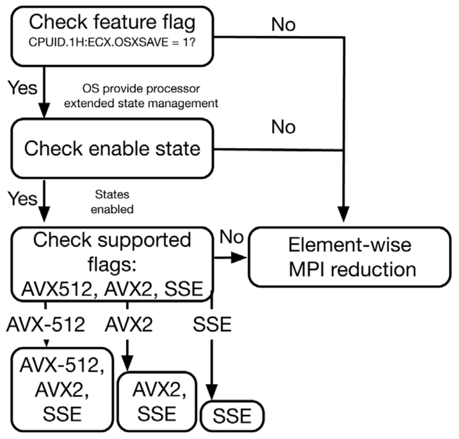
Check feature flag
CPUID.1H:ECX.OSXSAVE = 1?
No
Yes
OS provide processor extended state management
Check enable state
No
Yes
States enabled
Check supported flags: AVX512, AVX2, SSE
No
Element-wise MPI reduction
AVX-512
AVX2
SSE
AVX-512, AVX2, SSE
AVX2, SSE
SSE

Fig. 4. Procedural flow for detection and automatically activate the AVX component into the Open MPI build system.

or via intrinsic functions; (c) adapting intrinsic functions into programming models or languages for applications to use. The first strategy by using auto-vectorization, relies entirely on the compiler capabilities, but is portable and "future-proof", which means that it can adapt code to any generation of processors with a simple re-compilation of the code. However, to effectively use automatic vectorization, programmers must follow strict guidelines and restrictions for vectorizable code that are dependent on the target architecture and provide compile-time options largely dependent on a specific compiler's capability and efficiency. Programmers also need to be aware of the specifics of the instructions that are supported by a processor. Additionally, compilers have substantial limitations in the analysis and code transformations phases that in many cases prevent an efficient identification of SIMD parallelism in real applications [43]. The second method allows more control over the very low-level instruction stream, but the use of intrinsics is time-consuming and error-prone for application programmers and users. For this work, to integrate the use of AVX-512 features in the widely used programming model Open MPI with the third approach — we choose to use intrinsics and compile flags to guide the compiler in the vectorization phase to maximize performance. We hide the implementation complications in the Open MPI middleware, and therefore many applications benefit from it without dealing with low-level instructions.

A reduction is a typical operation encountered in many scientific applications, and consist of applying the same, arithmetic, logic or bit-wise operation on each data element of the input buffers. As such, these operations have large amounts of data-level parallelism and should be able to benefit from SIMD support.

A reduction operation performs element by element on the input buffers, and traditionally is translated into code that executes as a sequential operation but could possibly be vectorized under particular circumstance or with a specific compiler or constraints. Sometimes it may suffer from dependencies across multiple loop iterations. Fig. 5 illustrates the difference between a scalar operation and a vector operation with AVXs and SVE instructions of different vector length, respectively. It is an example of a vector instruction processing multiple elements simultaneously, compared to executing the additions sequentially. A scalar processor would have to perform one load, one computation, and one store instruction for every element. With some code reordering, the load and stores could be rearranged to maximize the use of available registers, but overall the performance of the code is defined by the amount of data being fetched from the memory and the depth of the arithmetical instructions. A vector processor performs one load, one computation, and one store for multiple elements. More specifically, AVX-512 SIMD-vector can process up to eight double-precision floating-point numbers or 16 integer values. It also allows

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
sizeof_type	▷ Number of bytes of the type of the in_buf and inout_buf

```
1:  procedure ReductionOp( $in\_buf, inout\_buf, count, type$ )
2:      $types\_per\_step = vector\_length\ /\ (8 \times sizeof\_type)$
3:      #pragma unroll
4:      for $k \leftarrow 0$ to $count$ with increment of $types\_per\_step$ do
5:          _mm512_loadu_si512 from $in\_buf + offset$
6:          _mm512_loadu_si512 from $inout\_buf + offset$
7:          _mm512_reduction_op
8:          _mm512_storeu_si512 to $inout\_buf + offset$
9:          Update left_over and offset
10:     if ( $left\_over \neq 0$ ) then
11:         Update $types\_per\_step$ >>= 1
12:         if ( $types\_per\_step \leq left\_over$) then
13:             _mm256_loadu_si256 from $in\_buf + offset$
14:             _mm256_loadu_si256 from $inout\_buf + offset$
15:             _mm256_reduction_op
16:             _mm256_storeu_si256 to $inout\_buf + offset$
17:             Update left_over and offset
18:     if ( $left\_over \neq 0$ ) then
19:         Update $types\_per\_step$ >>= 1
20:         if ( $types\_per\_step \leq left\_over$) then
21:             _mm_lddqu_si128 from $in\_buf + offset$
22:             _mm_lddqu_si128 from $inout\_buf + offset$
23:             _mm128_reduction_op
24:             _mm_storeu_si128 to $inout\_buf + offset$
25:             Update left_over and offset
26:     if ($left\_over \neq 0$ ) then
27:         while ( $left\_over \neq 0$ ) do
28:             Set case_value
29:             Switch(case_value) : {8 Cases}
30:             Update left_over
```

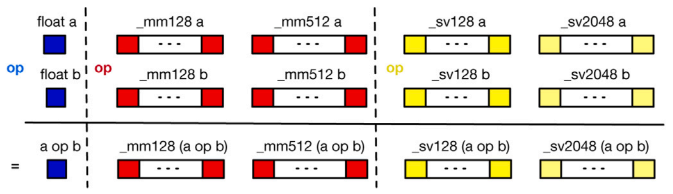

Fig. 5. Example of single precision floating-point operation using : (🟦) scalar standard C code operation, (🟥) AVXs 128 bits ~ 512 bits SIMD vector of 4,8,16 values operation; (🟨) SVE 128 bits ~ 2048 bits SIMD vector of different values operation.

the computation of those elements by executing a single instruction. AVX-512 reduction instructions perform arithmetic horizontally across active elements of a single source vector and deliver a scalar result. Arm SVE supports vector length up to 2048 bits, allowing more extensive reduction operations with more elements in a long vector.

3.5. Implementations with AVXs

Intel AVX-512 intrinsic provides arithmetic reduction operation for integer and float-pointing, and also supports logical and bit-wide reduction operations on integer type. This gives the chance to create AVX-512 intrinsic-based reduction support in MPI which will highly increase MPI local reduction's performance. Additionally, AVX-512 can

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 (op_avx/sse_support value)
AVXs	AVX512BW (0 × 200) AVX512F (0 × 100)
	AVX2 (0 × 020)) AVX (0 × 010)
SSE	SSE4 (0 × 08) SSE3 (0 × 04)
	SSE2 (0 × 02)) SSE (0 × 01)

perform scatter reduction operations with the support of predicate register, which behaves in a vectorized manner. This could lift the restriction of a contiguous memory layout for reduction operation, and allow for non-contiguous data sets, but such operations are not needed for the predefined MPI reduction operations. For our optimized reduction operation, we employ and apply multiple methods to investigate how to achieve the best performance on different processors, as shown in algorithm 1. For a more detailed description, in the rest of the paper, we assume that the hardware supports AVX-512. In the algorithm's for-loop section we explicitly use 512 bits long vector loads and stores for memory operation rather than using the memory copy (memcpy) function provided by the standard library, because some compilers may not perform the best assembling techniques of using ZMM registers for load and store. Once we have the elements loaded in registers, the corresponding vector operation is used to perform the reduction on the entire vector register. We repeat this pattern with a full 512 bits until the remainders cannot fulfill a 512 bits vector, then we fallback to use a lesser vectorization technique, such as using YMM registers to process elements that fit in the 256 bits registers, then 128 bits operations and finally, where necessary, executing few the operation on the remaining few elements.

We have noticed that during the last part of the reduction operation and depending on the number of elements on which to apply the operation, significant execution time is often spent in the epilogue, that deals with the remainder, those few elements that cannot fill a full vector register. Intel provides AVX mask intrinsics for mask operations that can vectorize the remainder loop. Still, significant overhead is involved in creating and initializing the mask and executing a separate and additional code path, which can result in lower SIMD efficiency. The vectorized remainder loops can be even slower than the scalar executions due to the overhead of masked operations and hardware. Typically, the compiler can determine if the remainder should be vectorized based on an estimate of the potential performance benefit. When trip count information for a loop is unavailable, however, it will be difficult for the compiler to make the right decision. Therefore, for the remainder, we use Duff's device, manually implementing a loop unrolling approach by interleaving two syntactic constructs of C: the do-while loop and a switch statement, which helps the compiler to optimize the device correctly.

Table 1 shows the variety of data types and abbreviations for MPI reduction operation handle names that are supported in our optimized reduction operation module, which matches the combination of types and operations defined by the MPI standard. Table 2 lists the supported x86 instruction set architectures and related CPU flags from legacy SSE to the latest AVX-512 instruction sets, together with the corresponding ***op_avx_support*** values that can be used to select which AVXs to use if they are supported by the hardware. To be noted, our work mainly focuses on the "Fundamental" feature instruction set with flag AVX512F, available on Knights Landing processors and Intel Xeon processors. It contains vectorized arithmetic operations, comparisons, type conversions, data movement, data permutation, bitwise logical operations on vectors and masks, and miscellaneous math functions. The AVX-512BW (Byte and Word) support offers basic arithmetic operations and masking for 512-bit vectors of byte-size (8-bit) and word-size (16-bit) integer elements. This is similar to the core feature set of the AVX2 instruction set, but with more comprehensive and more extended registers, and more functional supports for float-pointing and integer.

3.6. Implementations with SVE

We implemented our SVE-based reduction with Arm C language extension (ACLE) using intrinsics. As shown in algorithm 2, ACLE uses a variable vector length which can be accessed at runtime by function call of ***svcntb()*** | ***svcnth()*** | ***svcntw()*** | ***svcntd()*** to determine the number of 8, 16, 32, 64-bit elements in the vector. As previously mentioned, Open MPI uses a modular architecture, and we added another reduction module in the operation framework enabled only on Arm architectures with SVE support. We compiled using Arm HPC compile 20.0, enabling SVE extensions using the flag ***-march=armv8-a+sve***. As with AVX reduction, our SVE implementation also supports different data types and abbreviations for MPI reduction operations, as defined by the MPI standard.

Algorithm 2 Arm SVE 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( $in\_buf, inout\_buf, count, type$ )
 2:     #svcnt*: Count the number of 8,16,32,64-bit elements in a vector
 3:     $types\_per\_step$ = svcntb | svcnth | svcntw | svcntd
 4:     #pragma unroll
 5:     for $k \leftarrow 0$ to $count$ with increment of $types\_per\_step$ do
 6:         svld1 from $in\_buf + offset$
 7:         svld1 from $inout\_buf + offset$
 8:         sv_reduction_op
 9:         svst1 to $inout\_buf + offset$
10:         Update left_over and offset
11:     if ($left\_over \neq 0$ ) then
12:         while ( $left\_over \neq 0$ ) do
13:             Set case_value
14:             Switch(case_value) : {8 Cases}
15:             Update left_over
```

4. MPI reduction benchmark evaluation

4.1. Intel Xeon architecture

We conduct our experiments on a local cluster, which is an Intel(R) Xeon(R) Gold 6254 (AVX512F) based server running at 3.10 GHz. Our work is based upon Open MPI master branch, git commit hash #75a539 [44]. Each experiment is repeated 30 times, and we present the average results. We use a single node with one process for all tests, because our optimization aims to improve the performance of the computation part of reduction operation rather than the communication.

This section compares the performance of the reduction operations with two implementations. For Open MPI default reduction operation

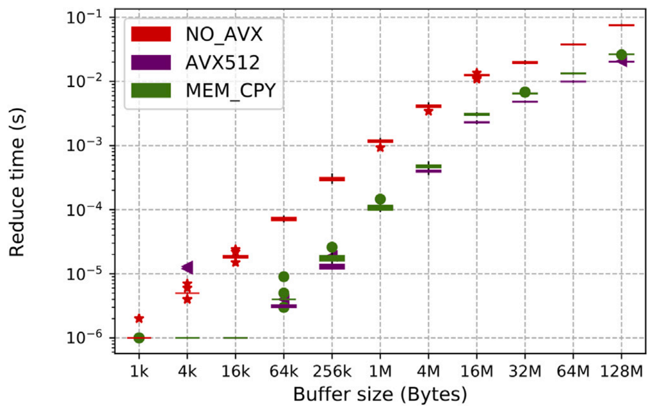

Fig. 6. Comparison of MPI_SUM for MPI_UINT8_T with and without AVX-512, with the memcpy operation.

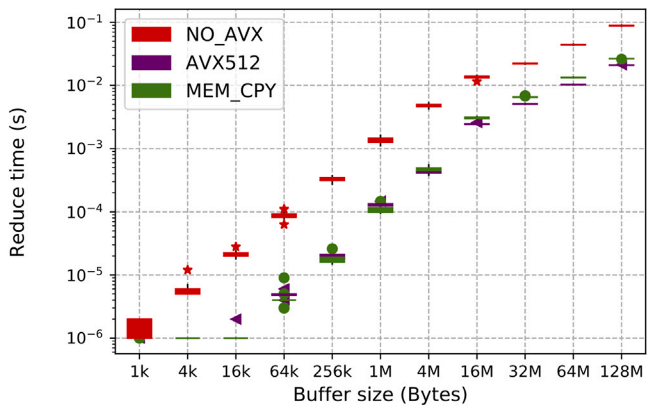

Fig. 7. Comparison of MPI_BAND for MPI_UINT8_T with and without AVX-512, with the memcpy operation.

base module, it performs element-wise computation across two input buffers. For each loop iteration, it processes two elements. Our new implementation uses AVX-512 vector 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 represent 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 reduction operations utilizing an array of different sizes.

We compare the predefined MPI operations, the arithmetic SUM and the logical BAND using input buffers with sizes in the range from 1 KB to 128MB. For the experiments, we minimized the potential impact of preloaded caches by flushing the L1 and L2 cache after each test to ensure we are not reusing data from the cache (especially for buffers size below the cache size).

Figs. 6 and 7 show the time-to-completion for the MPI_SUM and MPI_BAND for the same MPI predefined type (MPI_UINT8_T). Different shapes of symbols (stars, circles, arrows) represent outlier data that extend beyond the whiskers. It should be noted that the default compiler on the platform, failed to generate auto-vectorized code despite our best efforts (i.e. providing all the documented optimization flags). Our optimization uses intrinsics which give us complete control of the low-level details at the expense of productivity and portability.

Results demonstrate that using AVX-512 enabled operation the performance can be improved seven times faster compared with the default, element-wise operation. As expected, the improvement is dependent to the number of elements in the reduction buffer, small number of elements showing a small improvement that increases once the buffer size becomes larger than 4 KB, where the performance

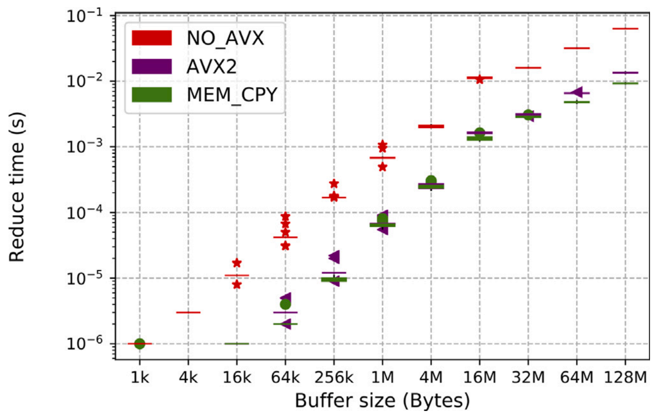

Fig. 8. AMD EPYC 7302 16-Core Processor: Comparison of MPI_BAND for MPI_UINT8_T with and without AVX2, with the memcpy operation.

improvement becomes considerable. For the sake of completeness, We compare the MPI operations with the memory copy (memcpy) operation, under the assumption that the vendor provided implementation of memcpy is highly optimized for the target architecture, and would therefore provide an upper bound. To make a fair comparison, we list the complete execution sequence of reduction operation and memory copy operation. In terms of memory accesses, the MPI reduction operation needs two loads from both input buffers, the computation between these two elements, followed by one store to save the results into memory. The memcpy operation needs only one load from the source buffer and one store to the destination buffer. The result shows that even with an additional computation included, our optimized AVX-512 reduction operation achieves a high level of memory bandwidth comparable to memory copy. When the reduction buffer size increases, our implementation achieves similar performance as memory copy, which indicates that our approach is capable to take full advantage of all the available memory bandwidth.

4.2. *AMD Zen 2 architecture*

AMD's new Zen architecture supports all the x86 vector instructions such as SSE and AVX2. However, the data paths are only 128 bits wide, and as a result 256-bit wide operations are carried out as two independent 128-bit operations, which means 256-bit operations will use up twice as many hardware resources to complete (registers and compute units). Thus, the peak throughput is four SSE/AVX-128 instructions or two AVX-256 instructions per cycle.

The Zen 2 architecture doubles the physical registers' width, execution units, and data paths to 256 bits. This improvement doubles the peak throughput of AVX-256 instructions to four per cycle, or in other words, up to 32 FLOPs/cycle in single precision or up to 16 FLOPs/cycle in double precision.

We conducted our benchmark experiments on an EPYC 7302 processor-based cluster, which is based on the Zen 2 microarchitecture with a base frequency of 3.0 GHz, supporting AVX and AVX2 instructions.

Fig. 8 show the result for the MPI_SUM operation on buffers with various sizes ranging from 1 KB to 128MB. We can see that our AVX2 reduction operations perform about five times faster than the default operations in Open MPI for all the tested sizes. When compared with the memory copy operations, our optimized operations achieve almost the same memory bandwidth, which implies that the computation is totally overlapped with memory operation.

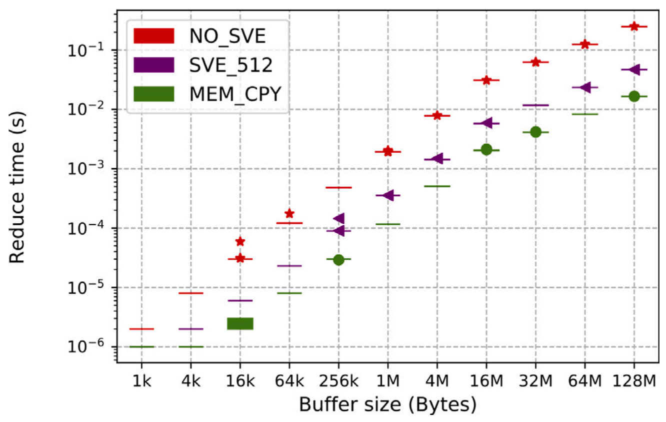

Fig. 9. Arm A64FX: Comparison of MPI_SUM with SVE (Vector Length = 512bits) reduction enable and disable for MPI_UINT8_T together with memcpy.

4.3. *Arm-v8 architecture: A64FX*

We conducted our performance evaluation experiments on the new A64FX processor, which supports SVE operations with vector length of 256 bits and 512 bits. Fig. 9 shows the results of the MPI_SUM operation from the Open MPI default implementation, the SVE optimized implementation and the memory copy operation. We can see that, under all tested reduction buffer size, our SVE optimized operation is five times faster than element-wise operation, and obtains a similar memory bandwidth to the memory copy operation.

5. Performance tool evaluation

To understand the performance, we analyzed our AVX-512 enabled Open MPI reduction operation using Performance API (PAPI) [45] – a tool that can expose hardware counters, allowing developers to correlate these counters with the application performance. PAPI is a portable and efficient API to access hardware performance monitoring registers/counters found on most modern microprocessors. These counters exist as a small set of registers that count "events", which are occurrences of specific signals and states related to the processor's function. Monitoring these events facilitates correlation between the structure of executed code, and indirectly of the source or object code, with the efficiency of executing this code on the underlying architecture. This correlation has a variety of uses in performance analysis and tuning.

We aim to use PAPI's hardware performance counters to measure two aspects: (1) Memory operation instructions: the total number of load and store instructions. (2) Branching instructions: number of branch execution instructions separated into branch instructions taken and not-taken, instructions mispredicted and instructions correctly predicted. All these events have a significant impact on performance.

Fig. 10 shows the total number of instructions, and memory access instructions of load and store, and branch instructions. Due to the stability of the results we choose not to clutter the graphs with additional information, such as the standard deviation. We can see that for our optimized reduction operation, the total number of instructions is largely decreased. Also, memory access and branch instructions have decreased compared to the default implementation in Open MPI. The reason of all this is straightforward: longer vectors load and store more elements with each single instruction compared with non-vector loads and stores, which means that we need fewer loads and stores dealing with the same amount of reduction data. Our implementation decreased the number of loads and stores instructions by a factor of 90X and 60X, respectively. At the same time, for branching instructions, our optimization decreased by 60X. We also investigated the cache misses

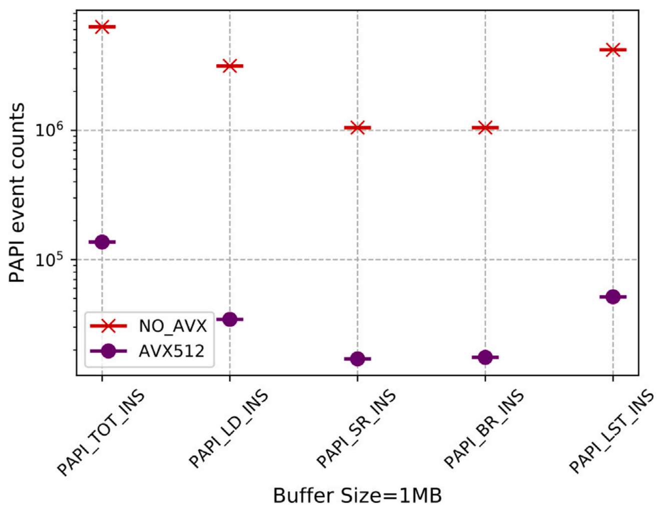

Fig. 10. Comparison between AVX-512 optimized Open MPI and default Open MPI for MPI_SUM reduction with PAPI instruction events overview.

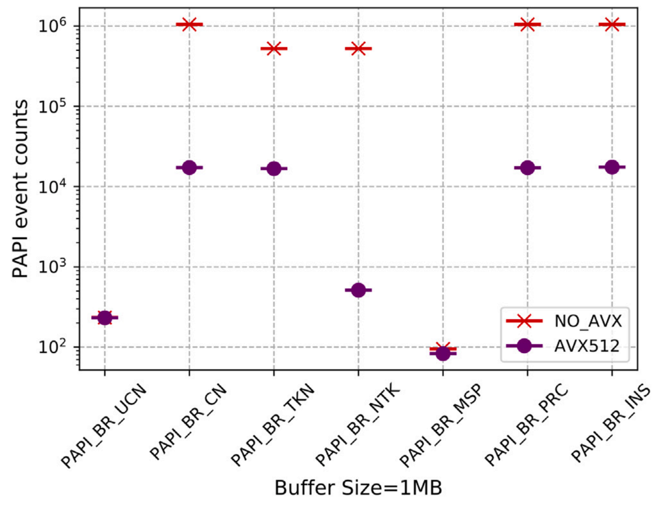

Fig. 11. Comparison between AVX-512 optimized Open MPI and default Open MPI for MPI_SUM reduction with PAPI branch counters.

of L1 and L2 caches. Because we are dealing with large buffers of contiguous data, this means data access patterns are very regular and therefore easy to predict by even a basic prefetcher algorithm. All predicted accesses would be consumed so that the cache misses do not show significant variation.

Fig. 11 illustrates the instruction count details of branch instructions of both AVX-512 optimized implementation and the default element-wise reduction method. By using long vectors, we largely decreased the "for loop" of the reduction operation. Consequently, the AVX-512 code has much less control and branching instructions, and therefore less opportunity to mispredict the branching outcome.

6. LAMMPS application evaluation

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)
[46] is a molecular dynamics simulation tool from Sandia National Laboratories. It provides different benchmark datasets representing a range of simulation styles and computational expense for molecular-level interaction forces. In our experimental analysis, we evaluate the performance of our reduction operation with LAMMPS granular flow

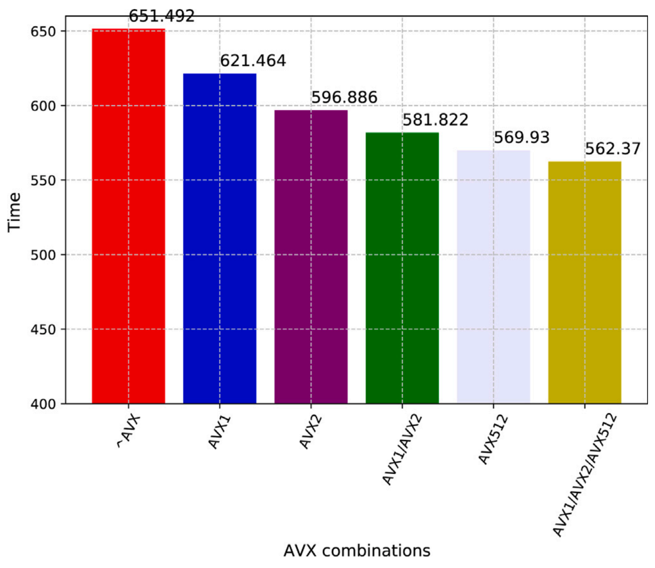

Fig. 12. LAMMPS chute: loop time on 24 procs for 100 steps with 259200000 atoms with different AVX capabilities.

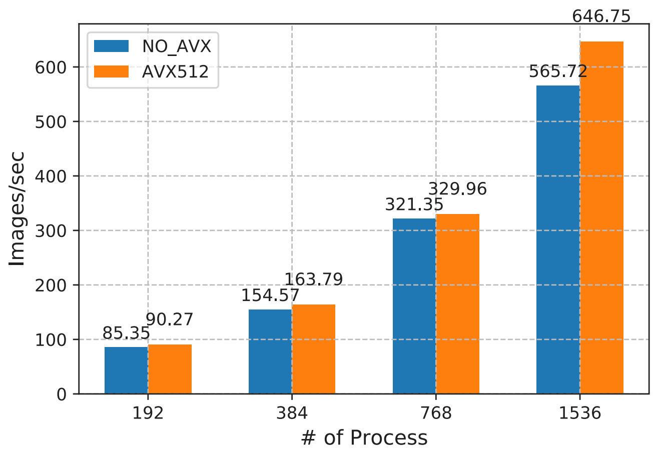

Fig. 13. tf_cnn_benchmarks results using Horovod (model: alexnet) on stampede2 with AVX-512 default Open MPI and optimized Open MPI.

benchmark using the dataset from chute flow (in.chute.scaled). The benchmark reports the “Loop Time” as a measure of time required to simulate a set of molecular interactions. We run the benchmark with 24 processes (process grid: 4 × 2 × 3) on an Intel(R) Xeon(R) Gold 6254 CPU with different capabilities of AVX support, including single AVX, AVX2 and AVX512 Our implementation allows restricting the vector capabilities used for MPI reduction operations via the modular parameter of ***–mca op_avx_support***.

Fig. 12 presents the loop time of LAMMPS chute benchmark running on 24 processes for 100 steps with 259200000 atoms using different AVX capabilities. Different collective operations are commonly and frequently used in LAMMPS benchmark (eg. MPI_Allreduce). We can see, without AVX support for the reduction operations as shown in red, the latency of the loop is 651.5. With the optimization of using AVX and AVX2, we archive 11% speedup of the total application’s executing time. Enabling AVX512 support provides an additional performance boost, to up to 13.4% speedup. Tuning the switch points between the different vector instructions provides an additional performance boost, with a maximum speedup of 14.7%.

7. Deep learning application evaluation

Over the past few years, advancements in deep learning have driven tremendous improvement, among other to image processing, computer vision, speech recognition, robotics and control, natural language processing. One of the significant challenges of deep learning is to decrease the extremely time-consuming cost of the training 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 research and production. Distributed deep learning is one of the essential technologies in reducing training time. The critical aspect to understand in deep learning is that it needs to calculate and update the gradient to adjust the overall weights. Processes need to prepare and calculate all the gradient data, which is usually very large. When such data and calculations are too extensive, users need to parallelize these calculations and computations. It indicates that the training needs to be executed on distributed computing nodes working together with each node working on a subset of the data. When each of these processing units or workers (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.

In this section, we investigate and experiment on Horovod [47] - 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. Horovod utilizes Open MPI to launch copies of the TensorFlow program. Open MPI will transparently set up the distributed infrastructure necessary for processes to communicate with each other. All the user needs to do is to modify their program to average gradients using an MPI_Allreduce operation. Conceptually, MPI_Allreduce forces each participating process to share its data with all other processes and applies a reduction operation. This operation can be any reduction operation, such as a sum, max, or min. In other words, it reduces the target arrays in all processes to a single array and returns the result array to all processes. Horovod uses a ring MPI_Allreduce approach, which is a bandwidth optimal [38] algorithm if the tensors are large enough, but does not work as efficiently for smaller tensors. Horovod can also use a Tensor Fusion — an algorithm that fuses tensors together before it calls ring MPI_Allreduce. The fusion method allocates a large fusion buffer and executes the MPI_Allreduce operation on the fusion buffer. In the ring MPI_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, after each process receives the data from the previous process, it applies the reduction and proceeds to send it again to the next process in the ring. We can see that during the MPI_Allreduce processing phase, there are P ∗(N - 1) reduction operations that occurred with big fusion buffer size, which is very computation intensive. Our AVX-512 optimized reduction operations can significantly improve the performance of the computation and reduction part of those collective operations.

We conducted our experiments on Stampede2 with Intel Xeon Platinum 8160 nodes. Each node has 48 cores with two sockets and it has 192 GB DDR4 memory. Each core has 32 KB L1 data cache and 1MB L2. The nodes are connected via Intel Omni-Path network. We experimented with TensorFlow CNN benchmarks using Horovod with tensorflow-1.13.1.

Fig. 13 shows the performance comparison of our AVX-512 optimized reduction operation and the default reduction operation in Open MPI for Horovod (with synthetic datasets and AlexNet model) to train an application called tf_cnn_benchmarks [48]. Comparing to default element-wise reduction implementations, with the increasing number of processes, our design shows increasing improvements, which start at 5.45% and eventually rise to 12.38% faster than default Open MPI on 192 processes and 1536 processes, respectively. We intend to include both communication and computation in the measurement to show the performance benefit from our design to the application's overall completion time. It can be observed that the performance benefit increases with more processes/nodes, because in the experiments we set $batch\_size = 32$ which means 32 samples from the training dataset are used to calculate the gradient to update the weights. This translate into the more MPI processes participate in the reduction operation, the larger the data is. Thus, the fact that each one of them is simultaneously using our AVXs optimized Open MPI operations drives up the overall application performance.

8. Conclusion

In this paper, we pragmatically demonstrated the benefits of Intel AVX, AVX2, AVX-512 and Arm SVE vector operations in the context of MPI reduction operations. We assess the performance advantages of different features introduced by AVX and extended our investigation and analysis to a fully-fledged implementation of all predefined MPI reduction operations. We implemented these new reduction operation modules in Open MPI using AVXs' and SVE intrinsic supporting different kinds of MPI reduce operations for multiple MPI types. We demonstrated the efficiency of our vector reduction operation using a benchmark calling MPI_Reduce_local. Experiments are conducted on an Intel Xeon Gold cluster, which highlights that AVX-512 enabled reduction operations can achieve 10X performance improvement. To further validate the performance improvements, experiments are conducted with different applications: (1) Using LAMMPS benchmark with variety AVXs support show speedup from 14% to 34% with different AVX capability combination. (2) Experiment with a deep learning application using distributed model Horovod, which calculates and updates the gradient to adjust the weights using an MPI_Allreduce. Our new reduction strategy achieved a significant speedup across all ranges of processes, with a 12.38% improvement with 1536 processes. Our analysis and implementation of Open MPI optimization provide useful insights and guidelines on how wide vector operations, in this case, Intel AVX extensions, can be used in actual high-performance computing platforms and software to improve the efficiency of parallel runtimes and applications. Our long vector optimized Open MPI proves that taking advantage of hardware capabilities remains of critical interest to software development, and that even a small improvement in the MPI implementation can have a significant impact on applications.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2021.102871.

Acknowledgments

This material is based upon work supported by the National Science Foundation, United States under Grant No. (1664142); and the Exascale Computing Project, United States (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, United States. The authors would also like to thank the Texas Advanced Computing Center (TACC). For computer time, this research used the Stampede2 flagship supercomputer of the Extreme Science and Engineering Discovery Environment (XSEDE) hosted at TACC.

References

[1] H. Caminal, D. Caballero, J.M. Cebrián, R. Ferrer, M. Casas, M. Moretó, X. Martorell, M. Valero, Performance and energy effects on task-based parallelized applications, J. Supercomput. 74 (6) (2018) 2627–2637.
[2] T. Röhl, J. Eitzinger, G. Hager, G. Wellein, Validation of hardware events for successful performance pattern identification in high performance computing, in: A. Knüpfer, T. Hilbrich, C. Niethammer, J. Gracia, W.E. Nagel, M.M. Resch (Eds.), Tools for High Performance Computing 2015, Springer International Publishing, Cham, 2016, pp. 17–28.
[3] R. Espasa, M. Valero, J.E. Smith, Vector architectures: past, present and future, in: Proceedings of the 12th International Conference on Supercomputing, 1998, pp. 425–432.
[4] W.J. Watson, The TI ASC: a highly modular and flexible super computer architecture, in: AFIPS '72 (Fall, Part I), 1972.
[5] D. Molka, D. Hackenberg, R. Schöne, T. Minartz, W.E. Nagel, Flexible workload generation for HPC cluster efficiency benchmarking, Comput. Sci. - Res. Dev. 27 (4) (2012) 235–243.
[6] D. Callahan, J. Dongarra, D. Levine, Vectorizing compilers: A test suite and results, in: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Supercomputing '88, IEEE Computer Society Press, Washington, DC, USA, 1988, pp. 98–105.
[7] D. Levine, D. Callahan, J. Dongarra, A comparative study of automatic vectorizing compilers, in: Benchmarking of High Performance Supercomputers, Parallel Comput. 17 (10) (1991) 1223–1244, http://dx.doi.org/10.1016/S0167-8191(05)80035-3, URL http://www.sciencedirect.com/science/article/pii/S0167819105800353.
[8] G. Mitra, B. Johnston, A.P. Rendell, E. McCreath, J. Zhou, Use of SIMD vector operations to accelerate application code performance on low-powered arm and intel platforms, in: 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, 2013, pp. 1107–1116.
[9] V. Pentkovski, S.K. Raman, J. Keshava, Implementing streaming SIMD extensions on the pentium III processor, IEEE Micro 20 (04) (2000) 47–57, http://dx.doi.org/10.1109/40.865866.
[10] P. Hammarlund, A.J. Martinez, A.A. Bajwa, D.L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R.B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, T. Burton, Haswell: The fourth-generation intel core processor, IEEE Micro 34 (2) (2014) 6–20.
[11] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y. Liu, Knights landing: Second-generation intel xeon phi product, IEEE Micro 36 (2) (2016) 34–46, http://dx.doi.org/10.1109/MM.2016.25.
[12] Intel, Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture, 2019, URL https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-1-basic-architecture.
[13] Intel, Intel 64 and IA-32 architectures software developer manuals, 2016, URL https://software.intel.com/en-us/articles/intel-sdm.
[14] D.S. McFarlin, V. Arbatov, F. Franchetti, M. Püschel, Automatic SIMD vectorization of fast Fourier transforms for the larrabee and AVX instruction sets, in: Proceedings of the International Conference on Supercomputing, ICS'11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 265–274, http://dx.doi.org/10.1145/1995896.1995938.
[15] Intel, 64-Ia-32-architectures instruction set extensions reference manual, 2019, URL https://software.intel.com/en-us/articles/intel-sdm.
[16] Arm, Arm architecture reference manual armv8, for Armv8-A architecture profile, 2018, URL https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.
[17] S. Flur, K.E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, 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, POPL '16, ACM, New York, NY, USA, 2016, pp. 608–621, http://dx.doi.org/10.1145/2837614.2837615, URL http://doi.acm.org/10.1145/2837614.2837615.
[18] M. Boettcher, B.M. Al-Hashimi, M. Eyole, G. Gabrielli, A. Reid, Advanced SIMD: Extending the reach of contemporary simd architectures, in: 2014 Design, Automation Test in Europe Conference Exhibition (DATE), 2014, pp. 1–4, http://dx.doi.org/10.7873/DATE.2014.037.
[19] A. Armejach, H. Caminal, J.M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, M. Moretó, Stencil codes on a vector length agnostic architecture, in: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18, ACM, New York, NY, USA, 2018, pp. 13:1–13:12, http://dx.doi.org/10.1145/3243176.3243192, URL http://doi.acm.org/10.1145/3243176.3243192.
[20] M. P. I. Forum, MPI: A message-passing interface standard version 4.0, 2020, URL https://www.mpi-forum.org.
[21] L. Bottou, Large-scale machine learning with stochastic gradient descent, in: Y. Lechevallier, G. Saporta (Eds.), Proceedings of COMPSTAT'2010, Physica-Verlag HD, Heidelberg, 2010, pp. 177–186.

[22] Z. Li, J. Davis, S. Jarvis, An efficient task-based all-reduce for machine learning applications, 2017, pp. 1–8, http://dx.doi.org/10.1145/3146347.3146350.

[23] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 25, Curran Associates, Inc., 2012, pp. 1097–1105, URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[24] P. Moritz, R. Nishihara, I. Stoica, M.I. Jordan, SparkNet: Training deep networks in spark, 2015, arXiv:1511.06051.

[25] R. Lim, Y. Lee, R. Kim, J. Choi, An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512, Cluster Comput. 21 (4) (2018) 1785–1795.

[26] R. Kim, J. Choi, M. Lee, Optimizing parallel GEMM routines using auto-tuning with intel AVX-512, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 101–110, http://dx.doi.org/10.1145/3293320.3293334.

[27] B. Bramas, A novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake, Int. J. Adv. Comput. Sci. Appl. 8 (10) (2017) http://dx.doi.org/10.14569/ijacsa.2017.081044.

[28] M.G.F. Dosanjh, W. Schonbein, R.E. Grant, P.G. Bridges, S.M. Gazimirsaeed, A. Afsahi, Fuzzy matching: Hardware accelerated MPI communication middleware, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 210–220, http://dx.doi.org/10.1109/CCGRID.2019.00035.

[29] A. Armejach, H. Caminal, J.M. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, M. Moretó, Using arm's scalable vector extension on stencil codes, J. Supercomput. (2019).

[30] D.A. Iliescu, Arm scalable vector extension and application to machine learning, 2018, URL https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning.

[31] D. Zhong, P. Shamis, Q. Cao, G. Bosilca, S. Sumimoto, K. Miura, 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.

[32] D. Zhong, Q. Cao, G. Bosilca, J. Dongarra, Using advanced vector extensions AVX-512 for MPI reductions, eurompi/usa '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–10, http://dx.doi.org/10.1145/3416315.3416316.

[33] C. Gómez, F. Mantovani, E. Focht, M. Casas, Efficiently running SpMV on long vector architectures, in: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 292–303, http://dx.doi.org/10.1145/3437801.3441592.

[34] J.L. Träff, Transparent neutral element elimination in MPI reduction operations, in: R. Keller, E. Gabriel, M. Resch, J. Dongarra (Eds.), Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 275–284.

[35] M. Hofmann, G. Rünger, MPI Reduction operations for sparse floating-point data, in: A. Lastovetsky, T. Kechadi, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 94–101.

[36] C. Chu, K. Hamidouche, A. Venkatesh, A.A. Awan, D.K. Panda, CUDA kernel based collective reduction operations on large-scale GPU clusters, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 726–735, http://dx.doi.org/10.1109/CCGrid.2016.111.

[37] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, J. Dongarra, HAN: a hierarchical AutotuNed collective communication framework, in: 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 23–34, http://dx.doi.org/10.1109/CLUSTER49012.2020.00013.

[38] P. Patarasuk, X. Yuan, Bandwidth optimal all-reduce algorithms for clusters of workstations, J. Parallel Distrib. Comput. 69 (2) (2009) 117–124, http://dx.doi.org/10.1016/j.jpdc.2008.09.002.

[39] H. Shan, S. Williams, C.W. Johnson, Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression, in: 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2018, pp. 1–11.

[40] Arm, Porting and optimizing HPC applications for arm SVE version 2.1, 2020, URL https://developer.arm.com/documentation/101726/0210/Port-and-Optimize-your-Application-to-SVE-enabled-Arm-based-processors.

[41] 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, 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, 2004, pp. 97–104.

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

[43] S. Maleki, Y. Gao, M.J. Garzar'n, T. Wong, D.A. Padua, An evaluation of vectorizing compilers, in: 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 372–382.

[44] Open MPI main development repository, URL https://github.com/open-mpi/ompi.

[45] D. Terpstra, H. Jagode, H. You, J. Dongarra, Collecting performance data with PAPI-c, in: M.S. Müller, M.M. Resch, A. Schulz, W.E. Nagel (Eds.), Tools for High Performance Computing 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 157–173.

[46] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1) (1995) 1–19, http://dx.doi.org/10.1006/jcph.1995.1039, URL http://www.sciencedirect.com/science/article/pii/S002199918571039X.

[47] A. Sergeev, M.D. Balso, Horovod: fast and easy distributed deep learning in TensorFlow, 2018, arXiv preprint arXiv:1802.05799.

[48] A benchmark framework for Tensorflow, URL https://github.com/tensorflow/benchmarks.