Using Arm Scalable Vector Extension to Optimize Open MPI

Dong Zhong[1,2], Pavel Shamis[4], Qinglei Cao[1,2], George Bosilca[1,2], Shinji Sumimoto[3], Kenichi Miura[3], and Jack Dongarra[1,2]

[1]Innovative Computing Laboratory, The University of Tennessee, US
[2]{*dzhong, qcao3*}@*vols.utk.edu*, {bosilca, dongarra}@icl.utk.edu
[3]Fujitsu Ltd, {*sumimoto.shinji, k.miura*}@*jp.fujitsu.com*
[4]Arm, *Pavel.Shamis@arm.com*

***Abstract*— As the scale of high-performance computing (HPC) systems continues to grow, increasing levels of parallelism must be implored to achieve optimal performance. Recently, the processors support wide vector extensions, vectorization becomes much more important to exploit the potential peak performance of target architecture. Novel processor architectures, such as the Armv8-A architecture, introduce *Scalable Vector Extension* (SVE) - an optional separate architectural extension with a new set of A64 instruction encodings, which enables even greater parallelisms.**

In this paper, we analyze the usage and performance of the SVE instructions in Arm SVE vector Instruction Set Architecture (ISA); and utilize those instructions to improve the memcpy and various local reduction operations. Furthermore, we propose new strategies to improve the performance of MPI operations including datatype packing/unpacking and MPI reduction. 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. The resulting efforts have been implemented in the context of Open MPI, providing efficient and scalable capabilities of SVE usage and extending the possible implementations of SVE to a more extensive range of programming and execution paradigms. The evaluation of the resulting software stack under different scenarios with both simulator and Fujitsu's A64FX processor demonstrates that the solution is at the same time generic and efficient.

***Index Terms*—SVE, Vector Length Agnostic, ArmIE, datatype pack and unpack, non-contiguous accesses, local reduction**

I. Introduction

The need to satisfy the scientific computing community's increasing computational demands lead to larger HPC systems with more complex architectures, which provides more opportunities to enrich multiple levels of parallelism. One of the opportunities is to exploit data-level parallelism by code vectorization [1], and this causes the HPC systems to equip with vector processors. Comparing to scalar processors, vector processors support Single Instruction Multiple Data [2] (SIMD) Instruction Set Architectures and operate on one-dimensional arrays (vectors) rather than single elements. There are efforts to keep improving the vector processors by increasing the vector length and adding new vector instructions. Intel Knights Landing [3] introduced the Advanced Vector Extensions instruction set, AVX-512, which provides 512-bits-wide vector instructions and more vector registers; and Arm announced new Armv8 architecture embraced SVE together with extension instruction sets.

SVE is a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture [4], [5]. 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 [6], [7]. Vector length constrains in the range from a minimum of 128 bits up to a maximum of 2048 bits in increments of 128 bits.

SVE 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 limited to:

1) using rich addressing mode which enables non-linear data access that can deal with non-contiguous data;
2) 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 reductions;
3) and permitting vectorization of loops with more complex loop carried dependencies and more complex control flow.

Those extensions largely expand the usages of Arm architecture and increase opportunities for vector processing; as a result, HPC platforms and software can benefit significantly from SVE features.

Message Passing Interface (MPI) [8] is a popular and efficient parallel programming model for distributed memory systems widely used in scientific applications. As many scientific applications operate on multi-dimensional data, manipulating these data becomes complicated because the underlying memory layout is not contiguous. The MPI standard proposes a rich set of interfaces to define regular and irregular memory patterns, the so called Derived Datatypes (DDT).

DDT provides excellent functionality and flexibility by allowing the programmer to create arbitrary (contiguous and non-contiguous) structures from the MPI primitive datatypes. It is also useful for constructing messages that contain values with different datatypes and sending non-contiguous data (sub-matrix and matrix with irregular shape [9]) which eliminates the overhead of sending and receiving multiple small messages

and improves bandwidth utilization. Multiple small messages can be constructed into a derived datatype and sent/received as a single large message.

Once constructed and committed, an MPI datatype can be used as an argument for any point-to-point, collective, I/O, and one-sided functions. With DDT, MPI datatype engine automatically packs and unpacks data based on the datatype; which is convenient for the user since it hides the low-level details. However, the cost of packing and unpacking in the datatype engine is high; to reduce this cost, MPI implementations need to design more powerful and efficient pack and unpack strategies. We contribute our efforts to investigate and improve the performance of datatype pack and unpack.

Computation-oriented collective operations like MPI_Reduce perform reductions on data along with the communications performed by collectives. These collectives typically 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 [3], [10]) 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 Arm SVE capabilities such as vector product to efficiently perform these operations.

This paper makes the following contributions:

1) presenting detailed analysis of how Arm SVE vector Instruction Set Architecture (ISA) can be used to optimize memory copy for both contiguous memory regions and non-contiguous memory regions;
2) analyzing SVE hardware arithmetic instructions to speed up a variety types of reduction operations;
3) exploring the usage of Arm SVE vector rich memory access pattern to increase the performance of MPI datatype pack and unpack operations. We describe how different optimizations such as using **multiple vector load and store**, predicated non-contiguous **Gather Load** and **Scatter Store** can be used. And we implemented the SVE-enabled optimizations in OPEN MPI using related SVE intrinsics.
4) optimizing MPI local reduction operations using SVE arithmetics which highly increased the performance;
5) and performing experiments using our SVE intrinsics based pack/unpack and reduction operations in the scope of OPEN MPI with Fujitsu's A64FX processor, which is the first processor of the Armv8-A SVE architecture. Experiment results demonstrate the efficiency of SVE instructions and our implementation. Further more, provides useful insight and guideline on how Arm SVE vector ISA can be used in high performance computing platforms and software.

The rest of this paper is organized as follows. Section II presents related researches taking advantage of Arm SVE for specific mathematics applications, together with a survey about optimizations of MPI DDT pack and unpack, and mpi local reduction by novel hardware. Section III provides details about SVE features and related tools that can simulate SVE instructions and evaluate the performance. Section IV describes the implementation details of our generic memory copy and reduction methods using Arm SVE intrinsics and instructions. Section V shows our optimized implementation details in the scope of OPEN MPI takes advantage of Arm SVE intrinsics and instructions. Section VI describes the performance difference between OPEN MPI and SVE-optimized OPEN MPI and provides a distinct insights on the how SVE can benefit OPEN MPI.

II. RELATED WORK

In this section, we survey related work on techniques taking advantage of advanced hardware or architectures. Petrogalli [11] gives instructions on how SVE can be used to replace and optimize some commonly used general C functions. In a later work [12], it explores the usage of SVE vector multiple instruction to optimize matrix multiplication in machine learning such as GEMM algorithm. In another work [13], 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 SVE for a specific application. In our work we study SVE enabled features in a more comprehensive way, and also provides detailed analysis about the efficiency achievements of using SVE instructions. We are focusing on networking runtime and not generic compute bound applications.

There have been several efforts to use similar techniques to improve MPI communication by optimizing pack and unpack procedure. Wu et al. [14] 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. This work [15] 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 [16] 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. Dosanjh et al. [17] took advantage of using AVX vector operation for MPI message matching to accelerate matches which demonstrated the efficiency of long vectors. We expand our investigation of SVE capabilities to optimize MPI from different aspects directly at processor instruction level which is more straightforward and without the need for external or extra hardware compared to previous work [14], [16].

Additionally, different techniques and efforts have been

studied to optimize MPI reduction operations. Jesper [18] proposed a simple implementation of MPI library internal functionality that enables MPI reduction operations to be performed more efficiently with increasing sparsity of the input vectors. Also [19], [20] provided design and implementation of computed-oriented collective using GPU accelerators. Luo [20] offloaded the reduction operations to the GPU asynchronously by using multiple CUDA streams which allowed the overlap of communications and reduction operations. Michael [21] presented a pipeline algorithm for MPI Reduce that used a Run Length Encoding scheme to improve the global reduction of sparse floating-point data. Compared to those work, our SVE arithmetic reduction optimization is more general which has no limitation of data representation and is using CPU resources only.

III. ARM SVE OVERVIEW

Remarkable features of SVE includes: (1) Support for scalable vector and predicate registers, which together deliver a set of instructions that operate on wide vectors and predicates, and extend efficient vectorization to even broader scope with more control operation for active elements only. (2) VLA allows the same SVE code to run on platforms using different vector lengths with no need to modify or recompile the code. This feature provides high portability of software in the space of various combinations of hardware and software stacks to embrace massive parallelism as well as a deeper and more complex component hierarchy to continue the growth in compute capabilities. (3) Gather load and scatter store strengthen the capabilities to selectively transfer non-contiguous data, resulting in better performing pack and unpack algorithms which can dramatically reduce the number of memory copies for non-contiguous data.

A. Arm Intrinsic Instructions

Intrinsics are C or C++ pseudo-function calls that the compiler replaces with the appropriate SIMD instructions. The Arm C language extensions (ACLE) [22] for SVE provide a set of types, accessors and intrinsic functions that a C and C++ compiler can directly convert into SVE assembly. It provides function-to-instruction mapping of SVE vectors and predicates, and function interfaces for relevant SVE vectorization instructions. It enables high level language user to explicitly use the datatypes and operations available in the Arm SVE ISA. Our SVE enabled optimized implementation is written in ACLE.

B. Arm Instruction Emulator and Gem5

SVE vector instruction extension offers many opportunities to optimize memory access and compute workloads, but with the availability of SVE-enabled hardware is not publicly available, we have rely on simulation techniques in order to evaluate our implementations.

ARMIE [23] is a tool that converts instructions not supported by hardware to native Armv8-A instructions, such as those from the SVE instruction set. ARMIE enables developers to run SVE executable on existing Armv8-A hardware paired with dynamic binary instrumentation, enabling full application tracing without the overhead found in simulators. ARMIE includes an emulation client for SVE and optional instrumentation clients (e.g., Instruction count client emulated SVE, Opcodes Count), which communicate between each other using the emulator API. Details about ARMIE's capability are not in the scope of this paper, but we still want to mention the feature of collecting dynamic characteristics and metrics from the executing application, such as memory traces and instruction counts, allowing a more in-depth and more insightful analysis. ARMIE is not capable of producing timing information and incurs an emulation and binary instrumentation overhead on the running application. However, we can use instruction counts to measure how many actual instructions are executed during the execution of applications which includes the total number of instructions and the number of emulated SVE instructions. This feature will be a key method to evaluate the performance of our implementation in this paper.

Gem5 [24], [25] simulator is an open source modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. Gem5 has multiple execution modes, such as an atomic mode in which an instruction level simulation is performed, and an O3 mode in which an accurate execution cycle number can be estimated by simulating an out-of-order pipeline. In addition, Gem5 supports many existing processor architectures such as Alpha, Arm, SPARC and x86. For the interests of this paper, Gem5 can model up to 64 heterogeneous cores of an Arm platform, and Arm implementation supports 32 or 64-bit kernels and applications. It provides support for modeling Arm SVE instructions which decouple SVE ISA semantics from its CPU models, enabling effective support of SVE instruction and providing cycle-accurate counts for those instructions. Gem5 has a full system mode and a system emulation mode. Under system emulation mode, memory management and system calls are executed as services in Gem5. Gem5 provides statistical information of accurate cycle counts which is not available using ARMIE. In our work, we investigate using both methods to get instruction count and cycle count.

IV. OPEN MPI OPERATION WITH SVE

A. SVE *rich memory access addressing for pack and unpack*

SVE vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate transfer registers. It provides several types of instruction that can be used to optimize memory operations as:

1) Predicated single vector contiguous element accesses load and store (ld1 and st1).
2) Predicated multiple vector contiguous structure load and store (ldNx and stNx, N = 2,3,4).
3) Predicated non-contiguous element accesses Gather Load and Scatter Store.

1) Contiguous memory layout

Memory copy is the most widely used memory operation. C standard library provides function as memcpy, and 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. Existing architecture using vectors which can copy 64 or 128 bits depending on vector length related to specific architecture. But with Arm SVE the maximum vector length we can apply is 2048 bits. SVE vector load and store work efficiently when copying large contiguous data or non-contiguous data with large block length. SVE also provides more paralleled memory copy operation by which single instruction using multiple vector for load and store, such as

$$svint8x4\_t\ svld4(svbool\_t\ pg,\ const\ int8\_t\ *base)$$

which interprets a single instruction uses four vectors loading data simultaneously.

2) Non-Contiguous memory layout

For non-contiguous datatype layouts as show in Figure 1, vector type Fig 1(a) 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 1(c) 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 1(b) cannot be described in a concise format using only block-length and stride.

Vector loads and stores of most processors can only access elements stored in consecutive locations. Thus, for non-contiguous data either issue loads and stores for each block using one register, or possibly issue several scalar load and store operations to load/store a vector register and then execute in vector mode, the resulting vector code obtains low speedups, or even slowdowns, with respect to its scalar counterpart. However SVE introduces new subsets of instruction that provides multiple addressing access mode to enable gather load and scatter store for non-contiguous memory. There are two kinds of addressing that have the same format with a base component with a displacement component: *vector plus immediate* and *scalar plus vector*. The base is the start point of source data, and the displacement represents offsets of all primitive data by a common offset described by an immediate value from the base address in each element of the vector register. In *scalar plus vector* addressing, it points to the memory that is separated from common base register by the offset in each element of the offsets vector with an option to shift the offset according to the element size to be loaded.

In our case, we use *scalar plus vector of offsets* mode, with a specified explanation as

$$svint32\_t\ svld1\_gather\_u32base\_offset\_s32(svbool\_t\ pg,$$
$$svuint32\_t\ bases, int64\_t\ offset)$$

is a gather load (ld1_gather) of signed 32-bit integer (_s32) from a vector of unsigned 32-bit integer base addresses (_u32base) plus an offset in bytes (_offset). We develop an optimized pack and unpack algorithm specialized for a vector-like datatype. Gather load and scatter store processes multiple non-contiguous small blocks simultaneously instead of using a for loop copy block by block. Gather load and scatter store is ideal for pack and unpack of derived regular vector type, we generate the offsets vector once based on block length and gaps then it can be repeatedly used. We can visualize that for less regular memory it may have a repeating pattern of memory layout, thus if we can generate offset vectors for the repeat pattern then we can apply multiple gather loads and scatter stores for each repetition and apply to all repetitions.

For column access pattern, SVE has a special instruction to generate offsets vector for this particular need as:

$$svint32\_t\ svindex\_s32(int32\_t\ base,\ int32\_t\ step)$$

with pattern {*base, base + step, base + step*2, ...*}. With gather load and scatter store, users can copy a whole vector of data which is much more efficient compared to cherry picking a single element per vector. To summarize, gather load and scatter store can efficiently pack and unpack non-contiguous data by generating reasonable offsets vector or vectors. Due to the space constraints, we focus our efforts on regular vector type in the rest of this paper.

B. Reduction operation

A reduction is a typical 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 probably could be vectorized under a particular circumstance. Sometimes it may suffer from dependencies across multiple loop iterations. SVE reduction instructions perform arithmetic horizontally across active elements of a single source vector and deliver a scalar result. SVE provides arithmetic reduction operation for integer and float-pointing, also supports logical reduction operations for integer type, an example format would be

$$svint32\_t\ svmul[\_s32]\_x(svbool\_t\ pg,\ svint32\_t\ op1,$$
$$svint32\_t\ op2)$$

this function produces the product results of two vectors. This gives the chance to create Arm intrinsic reduction in MPI, which will highly increase the parallelization and performance

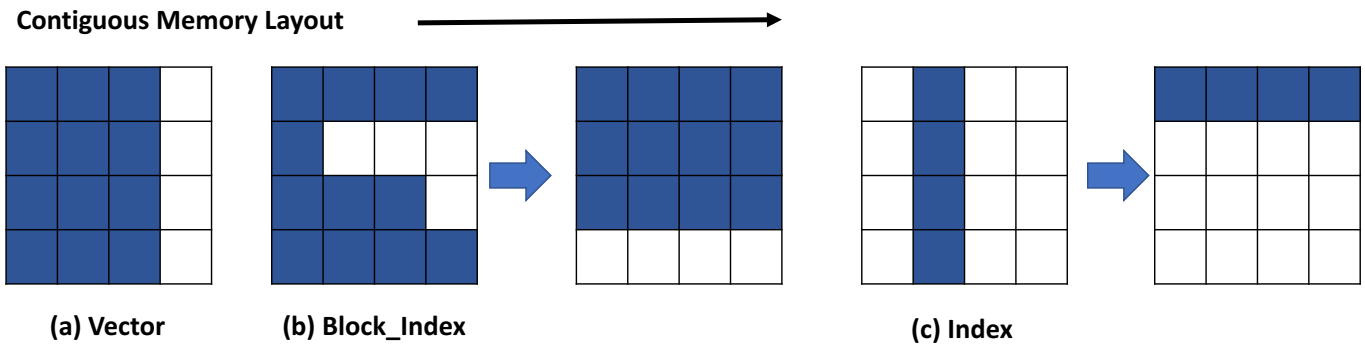

Fig. 1: Memory layout of non-contiguous datatype

TABLE I: Parameters and notations.

Symbol	Description
ARRAYSIZE	Number of elements for copying
Z0-Z31	Scalable vector registers
P1-P15	Predicate registers
Step	4∗Vector length in bytes
VL	Vector Length

of MPI local reduction. Additionally, SVE can performs scatter reduction operation with the accomplished support of predicate vector register which behaves in a vectorized manner. This profoundly expands the limitation of consecutive memory layout for reduction operation to non-contiguous at the same time generic and efficient.

C. *Evaluation of a generic* ACLE *memory copy*

Listing 1: Memcpy and ACLE code with Assembly

```
1  /* Region of interest */
2  memcpy(&dst, &src, ARRAYSIZE);
3  400780: ld1b z0.b, p0/z, [x10, x8]
4  400784: st1b z0.b, p0, [x11, x8]
5  400788: addvl x8, x8, #1
6  40078c: whilelo p0.b, x8, x9
7  400790: b.mi 400780<main+0x54>
```

```
1  /* Region of interest: copy with 4
       vectors simultaneous */
2  for(i=0; i<ARRAYSIZE; i+=step) {
3     svuint8x4_t vsrc =svld4(Pg, &src[i]);
4     svst4(Pg, &dst[i], vsrc);}
5  400708: ld4b z0.b-z3.b, p0/z, [x9,x10]
6  40070c: st4b z0.b-z3.b, p0, [x8,x10]
7  400710: add x10, x10, x11
8  400714: lsr x12, x10, #30
9  400718: cbz x12, 400708 <main+0x1c>
```

To understand the SVE instructions performance and extend the usage to OPEN MPI implementation. We implemented several micro benchmarks using ACLE to compare with extracted related kernel code from OPEN MPI. Experiments are conducted to demonstrate the efficiency of customized ACLE memory copy algorithms for both contiguous memory and non-contiguous vector type. Furthermore, we evaluate the effectiveness of SVE vector-based reduction operations. All experiments are conducted under the same experimental setting. We present the average and standard deviation of 30 times(which is too small to be noticeable). For the configuration, we use: *Arm HPC compiler 19.2* with *gcc 8.2.0*, compiled with *-O3* and *-march=armv8-a+sve* option by which will trigger automatic vectorization that targets SVE VLA techniques. It supports auto-vectorization for SVE as well as in-line assembly. For SVE instruction simulator and instruction clients, we use ARMIE *version 19.1*. For Gem5 we use Arm's internal version (corresponding upstream is gem5/sve/beta1) which has SVE enabled support. Furthermore, under *-O3* mode, the memory access instruction is divided into the processing of generating the cache request and the processing after receiving the data from the cache. This means that the Gem5 simulated results represent the complete data fetching operation instead of just issuing the cache request. Table 1 summarizes some of the notations we will employ to describe the algorithms and performance analysis. Equation 1 describes how we calculate the instruction count for all experiments, in order to evaluate and strengthen the number of instructions executed of the target code section, we need to eliminate the baseline - the number of instructions executed besides the target code (e.g. initialize and finalize). The baseline is calculated by running the same test codes but no data being processed.

1) *Contiguous memory*

For contiguous large data buffer, our ACLE customized memory copy implementation adapts two features of SVE. It not only adapts the usage of using long scalable vector for load and store which can copy more data compared to normal data register, but also uses four SVE vectors instead of single vector which highly reduces the number of control instructions. List 1 shows the code snippet comparison of memcpy and our ACLE optimized memory copy algorithm together with related assemble code. We can see that for memcpy method it is using $z0$ for load and store. For ACLE method it uses $z0$ ~$z3$ vectors for load and store. Those two methods have the same total amount of assembly code, and ACLE method copy four times of the data more than memcpy which highly decreases the number of total instructions. Also it needs less control and branching instructions, which will be reflected to performance gain.

$$\textbf{Instruction\_count} = \textbf{(Actual - Base)} \quad (1)$$

Actual: when dealing with $number\_of\_elements > 0$
Base: when dealing with $number\_of\_elements = 0$

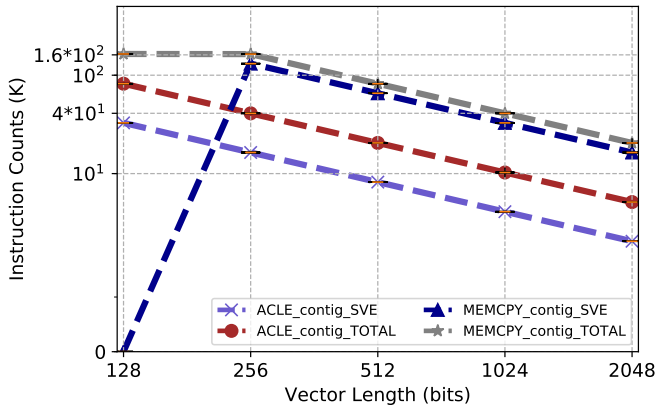

Fig. 2: Instruction counts of using MEMCPY and ACLE customized memory copy for 1MB contiguous buffer

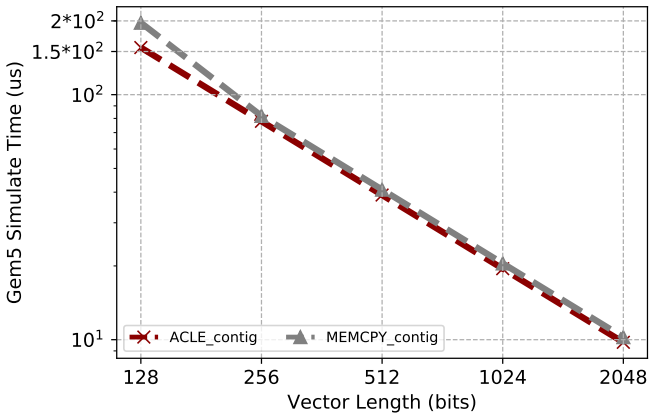

Fig. 3: Gem5 simulated time of MEMCPY and ACLE customized memory copy for 1MB contiguous memory

Figure 2 shows the instruction count results by ARMIE. Experiments are conducted with different vector length varies from 128 to 2048 bits for both memcpy and ACLE implementation copying 1MB data. We can see that for both cases the instruction count shows a linear decrease when VL increases, which demonstrate the efficiency of Arm SVE long vector. Also the total number of instructions executed by memcpy is 400% of ACLE, which shows a consistent behavior as the assembly code and illustrates using multiples vectors for load and store will highly decrease the number of instructions. The only exception is that when $VL = 128$ bits memcpy uses advanced SIMD instead of SVE vector, which demonstrates that by implicitly using ACLE intrinsic, it helps the compiler to optimize code assembling better.

Figure 3 displays the comparison results of Gem5 accurately simulated cycles of both methods. The size of the L1 cache is set to 32 Kbytes with 4 ways and the size of the L2 cache is set to 2 Mbytes with 16 ways. One load and one store operation are available in a cycle. Continuous load and store instructions access to the L1 cache by the vector length in a cycle. For both methods, the configuration uses the same number of physical registers. Copying the same amount of data, for both cases, it needs to create equal amounts of load and store requests. But for our ACLE customized method it largely reduced the number of control and branching instructions, which results in a performance gain of 5% as shown in the figure, when $VL = 128$ ACLE is 25% faster than memcpy which strengthens that this SVE instruction is more efficient than general instruction.

2) Non-Contiguous memory

The support of SVE instruction-level gather load and scatter store addressing capabilities provides a mechanism for defining new memory operation for non-contiguous memory (e.g. for sparse linear algebra operations), which can potentially deal with multiple small gapped memory pieces or indexed elements simultaneously based on the length of SVE vector. As such, multiple memory copies can be compressed as a single copy. To understand the performance of gather and scatter instruction relative to packing the data into a contiguous buffer before sending, and unpacking to non-contiguous buffer upon receiving. Several experiments were performed. The source data is non-contiguous but with a regular pattern as vector type and destination is contiguous. The same as above, we compare the performance of memcpy and ACLE code with different VL.

Figure 4 shows the instruction counts when copy 1MB non-contiguous data with $blocklen = 1$ and $gap = 1$. We can see that memcpy implementation cannot take advantage of using different vector length, and the amounts of instruction counts of different VL stay the same which is limited by the fact of copying block by block in the loop. However, the ACLE code takes the optimization of using SVE long vector and also gather load and scatter store feature which copies more data per instruction. It shows the decrease of instruction counts start from $10\times$ to $100\times$ as the VL increases compared to the non-optimized implementation. Also it shows a linear decrease with the vector length increases which proves that the load and store instruction fully fills the vector register. Also for ACLE method the ratio of SVE instruction compared to the total instruction is higher than memcpy method.

Figure 5 illustrates the Gem5 simulated cycle results for both methods when packing and unpacking non-contiguous data. We can see that with the same configuration ACLE gather and scatter implementation is faster than memcpy under all VL settings and shows a speedup of 15% to 20%. Compared to the decrease of instruction counts the simulated cycles are not significant; it is because the SVE supported implementation in Gem5 is not quite efficient. We believe with actual hardware, the clock cycles of gather and scatter instruction will be further reduced. Thus, we can see that by using Arm SVE instruction, our optimized ACLE memory copy algorithms provide good performance results for both contiguous and non-contiguous memory layout.

D. Evaluation of SVE *arithmetic reduction operations*

This section compares the performance of maximum reduction operation with two implementations. For OPEN MPI extracted kernel by C it performs element-wise maximum operation across two input buffers. For each loop iteration, it compares two elements. Our ACLE implementation we

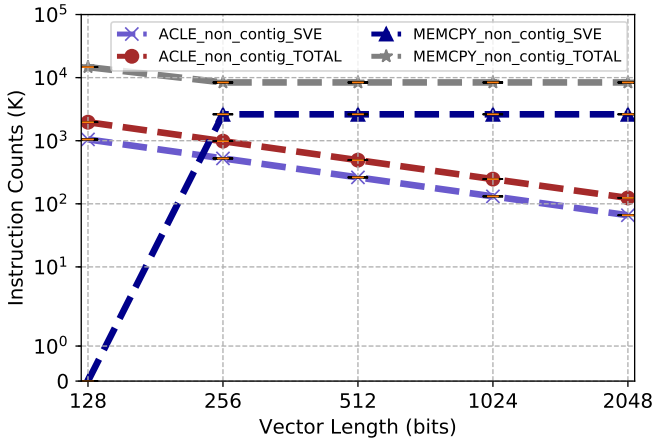

Fig. 4: Instruction counts of MEMCPY and ACLE customized memory copy for non-contiguous memory

use SVE vector reduction instruction executing maximum 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.

Figure 6 shows the instruction counts by using ARMIE for both cases with buffer size 4MB processing float point data. We can see for both methods that with $VL = 128$ bits the instruction count is $16\times$ compared to $VL = 2048$ bits, which demonstrates the efficiency of SVE long vector. Also by implicitly using ACLE vector instruction the instruction count is only 66% by using C with element-wise operation. With -O3 option, vectorizing compiler that automatically translates sequences of scalar operations, represented in the form of loops, into vector instructions which substitute element-wise operation to a vector version behavior. Our ACLE using intrinsics which gives us complete control of the low-level details at the expense of productivity and portability. Figure 7 shows the accurate execution cycles of the two experiments with Gem5 results. We can see that with the same configuration ACLE implementation is 30% faster. And it shows a linear decrease with vector length increase. Similar investigations are conducted for other reduction operations, including but not limited to SUM, PROD, and logical operation BXOR which show a similar performance benefit.

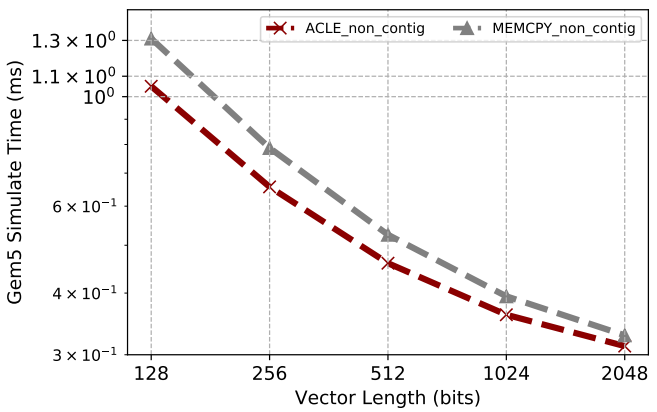

Fig. 5: Gem5 simulated time of MEMCPY and ACLE customized memory copy for non-contiguous memory

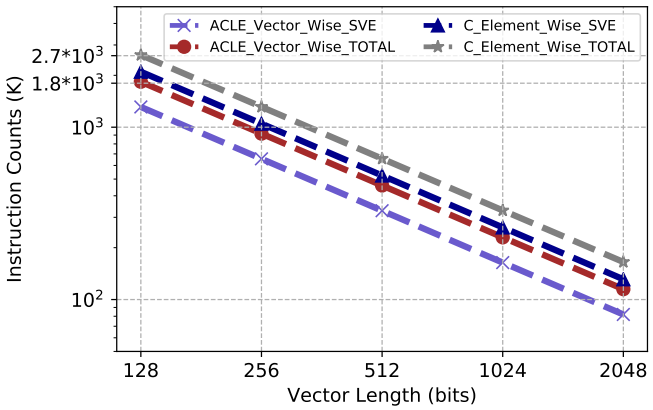

Fig. 6: Instruction counts of C element wise and ACLE vector wise maximum reduce operation for float point

V. Design and Implementation in Open MPI

We implemented our SVE optimization work in a set of components in Open MPI. While a full depiction of the architecture and feature set of Open MPI is out of the scope of this paper, some are relevant to our implementation effort. Open MPI is based on a Modular Component Architecture [26] which permits easily extending or substituting the core subsystem with experimental features. As shown in figure 8, within this architecture, each of the major subsystems is defined as an MCA framework, with a well-defined interface, and multiple components implementing that framework can coexist.

We added our SVE optimization work in two components to Open MPI architecture. The `SVE Pack Unpack` related component is in charge of using the high parallelization ACLE memory copy service mentioned in section IV-C. The improvement includes the optimization for pack and unpack with both contiguous data using four SVE vectors to load and store simultaneous, also taking advantage of gather load and scatter store instructions for non-contiguous small block data as revealed in algorithm 1. The `Arm_SVE_OP` component implements several reduction operations (eg, max, sum, prod) with Arm SVE vector reduction instructions; to be noted, this component can be extended out the scope of local reduction to general mathematics and logic operations. To the best of our

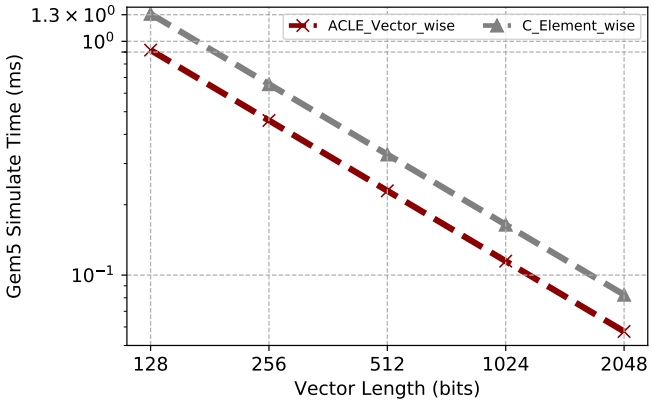

Fig. 7: Gem5 simulated time of C element wise and ACLE vector wise maximum reduce operation for float point

knowledge, our work is the first implementation to populate Arm SVE with a MPI implementation.

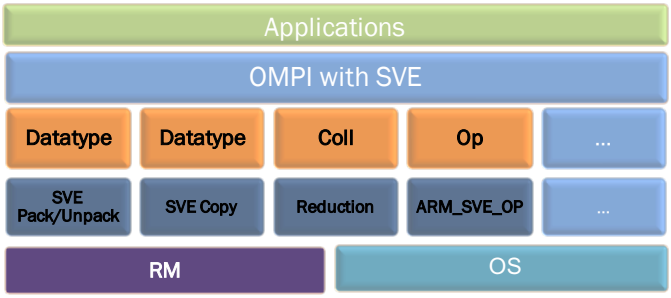

Fig. 8: SVE OPEN MPI architecture. The orange boxes represent components with added SVE features. The dark blue colored boxes are new modules or existing related modules.

Algorithm 1 SVE-based packing algorithm

svldN ▷ Using N vectors to load
svstN ▷ Using N vectors to store
svp ▷ SVE predicate type
svcntb ▷ vector length in bytes
blocklen ▷ Number of bytes of a contiguous memory block
offset_vector ▷ Displacement is a vector, and each element specifies a offset

procedure MEMCPYWITHMULTIPLEVECTORS($DST, SRC, blocklen$)
  $full\_vector\_copies$ = $blocklen$ / ($svcntb$ × N)
  for $k \leftarrow 0$ to $full\_vector\_copies$ **do**
    svldN from SRC
    svstN to DST
  if ($remaining \neq 0$) **then**
    Generate svp
    Partially ld/st using svp

procedure SVEBASEDPACK($Count, blocklen, Extend$)
▷ Example for Vector type 1(c)
  if ($blocklen \geqslant svcntb$) **then**
    for $k \leftarrow 0$ to $Count$ **do**
      MemcpyWithMultipleVectors(blocklen,Src,Dst)
  else
    $Blocks\_per\_vector$ = $svcntb$ / $blocklen$
    Generate $offset\_vector$
    for $k \leftarrow 0$ to ($Count$ / $blocks\_per\_vector$) **do**
      Sve gather_load using $offset\_vector$
    Generate svp
    Processing remaining blocks

VI. EVALUATION USING ARMIE

In this section, we discuss our experimental setup. We experimented on Arm HPC system which is a ThunderX2-based server running at 2 GHz. Our work is based upon OPEN MPI master branch, revision #75a539. Each experiment is repeated 30 times, and we present the average. For all experiment we use a single node with one process, because our optimization aims to improve the performance of local operation for all processes either with pack unpack or reduction. We compile and install OPEN MPI using Arm HPC compile 19.2 with flag "-march=armv8-a+sve". Without SVE supported publicly available hardware at the moment, we use ARMIE to evaluate the performance of our optimization. Instead of presenting the time to completion of the different benchmarks, we rely upon instead on the number of instructions to be executed, a metric that correctly highlights the performance implications.

We evaluate the performance of our packing and unpacking SVE optimization methodology using MPI datatype benchmark in OPEN MPI code base. We present here the results using non-contiguous memory layout vector type created by MPI_Type_vector with $blocklen = 1$ and $gap = 1$, repetition 2048 to make sure the data can fullly fill SVE vector even with $VL = 2048$ bits. For message size we use 1MB. In order to eliminate the unnecessary communication between processes and focus on our local pack and unpack optimization we intentionally use one process to send to itself by MPI_Send and MPI_Recv functions.

Figure 9 shows the comparison benchmark results using OPEN MPI and SVE optimized OPEN MPI. Experiments are conducted with $VL = 256$ to $VL = 2048$ bits. For $VL = 128$ bits our implementation doesn't use gather and scatter feature due to limited benefit. We can see that for OPEN MPI with different vector length it shows a constant number of instruction count which cannot take advantage of the long vector extension. For SVE optimized implementation the instruction count is 16% less starting with $VL = 256$ bits, which dramatically decreases as vector length increases because we optimized to use full vector for load and store. With $VL = 2048$ bits it is almost $10\times$ less.

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 of 4M bytes. Figure 10 shows the result for the (MPI_MAX) reduction, but we have observed a similar outcome for all the other reduction operations. First, it should be noted that the compiler, despite the optimization flags provided, did not generate auto-vectorized code for the default OPEN MPI, leading to a number of instruction constant, directly related to the number of elements in the input array. As our code explicitly uses the ACLE intrinsic vector reduction instruction, the reduction in the number of instructions is directly proportional to the vector lengths on which the instructions operate. Thus, with $VL = 128$ bits it reduces instruction count by 50%, when vector length reaches 2048 bits, the instruction count is reduced by a factor of 30, a clear indicator of a more concise generated code and potentially of shorter execution time.

VII. EVALUATION ON SVE ENABLED HARDWARE A64FX

We evaluate our implementation on a cluster with Fujitsu's Arm SVE based processor A64FX, which is the first processor of the Armv8-A SVE architecture and used for the post-K computer targeting HPC and AI applications. Each processor

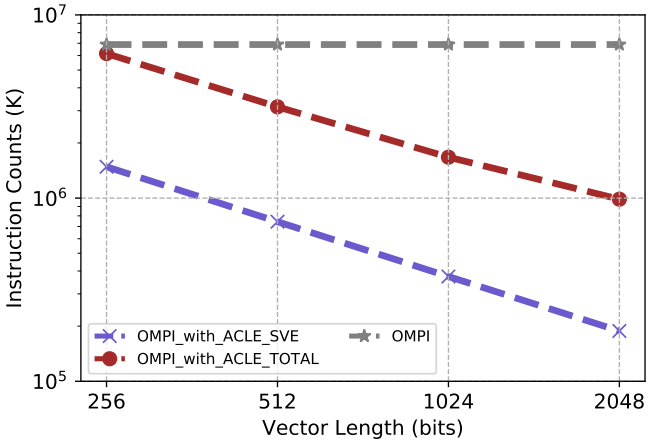

Fig. 9: Instruction counts of OMPI and OMPI with SVE datatype pack and unpack

hosts 4 Core Memory Group (CMG). A CMG consists of 13 cores, a L2 cache (8MiB, 16 way) and a memory controller.

The new processor supports enhanced SIMD and predicate operations that including:

1) 512 bits SVE vectors for 512-bits wise load/store and unaligned load crossing cache line.
2) Enhanced gather load and scatter store, enabling to return up to two consecutive elements in a 128-byte aligned block simultaneously.
3) Predicate operations by predicate register and predicate execution unit.

The pack/unpack operations have been highlighted as a major bottleneck for most applications using non-contiguous datatypes. Our work focuses on the low-level pack/unpack routines and any performance improvements on these routines will automatically transfer to MPI non-contiguous communications.

Figure 11 presents the performance of pack and unpack using gather/scatter feature with different vector length for non-contiguous buffer. The green and yellow line indicates the performance using vector length 256 bits and 512 bits respectively with our gather and scatter strategy. Compared to the blue line which is not using gather scatter feature. We can see that that optimized algorithm is $2\times$ faster which validates the Gem5 simulated results.

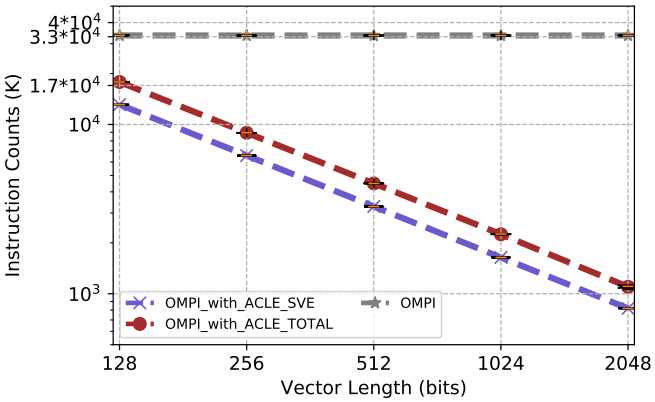

Fig. 10: Instruction counts of OMPI and OMPI with SVE local reduction of max operation

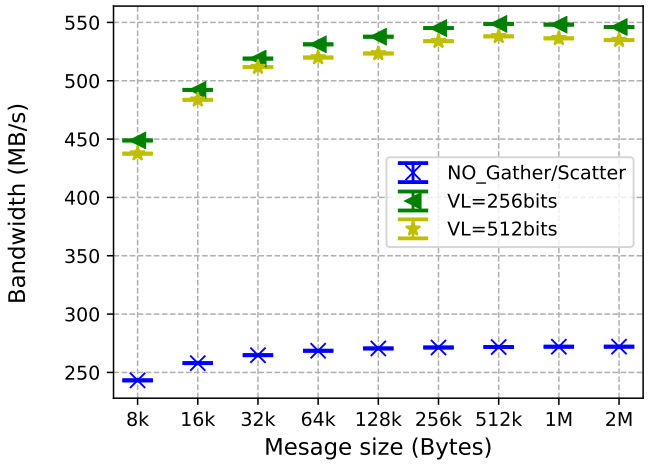

Fig. 11: MPI_PACK/UNPACK using Gather/Scatter with different vector length on A64FX processor

Figure 12 compares the performance of our SVE based mpi reduction operation. For the experiments, we flushed cache in order to make sure we are not reusing cache for a fair comparison. Our results demonstrate that with SVE-enabled operation it is $4\times$ faster than element-wise operation. We also compare MPI operation performance together with memcpy which indicates the peak memory bandwidth. For MPI reduction operation it needs $2loads + 1store + computation$, for memcpy it only needs $1load + 1store$. It shows that even with computation included our SVE reduction operation achieves a similar level of memory bandwidth as memcpy.

VIII. Conclusion

In this paper, we demonstrated the benefits of Arm SVE vector operations. We addressed the performance advantages of different features introduced by SVE with multiple vector lengths compared to non-SVE implementations. Furthermore, we extended the implementation of our investigation and analysis to introduced an optimistic MPI optimization from two aspects. For our first optimization, we adapted SVE rich memory address feature to improve the pack and unpack operations for regular MPI datatype. We use multiple vectors to simultaneously load and stores large contiguous memory

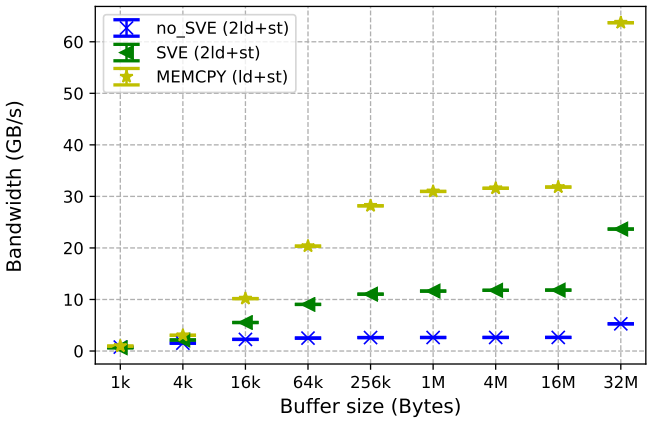

Fig. 12: Comparison of MPI local reduction together with MEMCPY for $VL = 512$ bits on A64FX processor

or large memory blocks, and use *gather load* and *scatter store* to copy multiple small blocks of non-contiguous memory using a single SVE instruction. We use ARMIE to simulate and calculate the actual number of instructions executed with MPI DDT benchmark. We reduced the instruction counts from 16% to 10X respectively depending on the vector instruction length. Also, we introduced a new reduction operation module in OPEN MPI using Arm SVE 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 $30\times$. To further validate the performance improvements, experiments are conducted using Fujitsu's A64FX processor. With our gather and scatter based pack and unpack the new algorithm is $2\times$ faster. For MPI_Local_reduce with $VL = 512$ SVE based reduction operation is $4\times$ 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.

REFERENCES

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

[2] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1493–1508. [Online]. Available: http://doi.acm.org/10.1145/2723372.2747645

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

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

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

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

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

[8] M. P. I. Forum. (September,2012) MPI: A Message-Passing Interface Standard. [Online]. Available: https://www.mpi-forum.org

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK User's Guide*, J. J. Dongarra, Ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[10] J. M. Cebrian, L. Natvig, and M. Jahre, "Scalability analysis of AVX-512 extensions," *The Journal of Supercomputing*, Apr 2019.

[11] F. Petrogalli. (2018) A sneak peek into SVE and VLA programming. [Online]. Available: https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/a-sneak-peek-into-sve-and-vla-programming

[12] D. A. Iliescu. (2018) Arm Scalable Vector Extension and application to Machine Learning. [Online]. Available: https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning

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

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

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

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

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

[18] J. L. Träff, "Transparent Neutral Element Elimination in MPI Reduction Operations," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 275–284.

[19] C. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and 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)*, May 2016, pp. 726–735.

[20] X. Luo, W. Wu, G. Bosilca, T. Patinyasakdikul, L. Wang, and J. Dongarra, "ADAPT: An Event-based Adaptive Collective Communication Framework," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 118–130. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208054

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

[22] ARM. (2017) ARM C Language Extensions for SVE. [Online]. Available: https://developer.arm.com/docs/100987/latest/arm-c-language-extensions-for-sve

[23] M. Tairum. (2018) Emulating SVE on existing Armv8-A hardware using DynamoRIO and ArmIE. [Online]. Available: https://community.arm.com/developer/tools-software/hpc/b/hpc-blog/posts/emulating-sve-on-armv8-using-dynamorio-and-armie

[24] A. A. Abudaqa, T. M. Al-Kharoubi, M. F. Mudawar, and A. Kobilica, "Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator," in *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, May 2018, pp. 317–322.

[25] Y. Kodama, T. Odajima, M. Matsuda, M. Tsuji, J. Lee, and M. Sato, "Preliminary Performance Evaluation of Application Kernels Using ARM SVE with Multiple Vector Lengths," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 677–684.

[26] D. Zhong, A. Bouteiller, X. Luo, and G. Bosilca, "Runtime Level Failure Detection and Propagation in HPC Systems," in *Proceedings of the 26th European MPI Users' Group Meeting*, ser. EuroMPI '19. New York, NY, USA: ACM, 2019, pp. 14:1–14:11. [Online]. Available: http://doi.acm.org/10.1145/3343211.3343225