8-2021

Toward Reliable and Efficient Message Passing Software for HPC Systems: Fault Tolerance and Vector Extension

Dong Zhong
dzhong@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Dong Zhong entitled "Toward Reliable and Efficient Message Passing Software for HPC Systems: Fault Tolerance and Vector Extension." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Jack Dongarra, George Bosilca, Michael Jantz, Yingkui Li

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Toward Reliable and Efficient Message Passing Software for HPC Systems:

Fault Tolerance and Vector Extension

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Dong Zhong

August 2021

To my parents Zhong Yuan and Yanxiong Tuo,

my brothers Geng Zhong and Lei Zhong for their love, trust, and support.

Acknowledgments

I would like to thank my advisor, Dr. Jack Dongarra, for giving me the opportunity to join Innovative Computing Laboratory (ICL) as a graduate research assistant. I appreciate the unconditional support and professional guidance. It is a great privilege to work with him, and his experience and wisdom will always be the guidance for my future work and life.

I would also like to thank my co-advisor and group leader Dr. George Bosilca for his guidance, motivation, and support during my graduate study. His comprehensive and thorough approach to thinking through a research problem inspired me to think more before taking action. His kindness, patience, and encouragement made my study and research a pleasant experience, and I feel lucky to be in this great group. He introduced me to the high-performance computing field and has been guiding me through my whole Ph.D. study. I could not have finished this dissertation without his help.

I would like to thank my committee members Dr. Michael Jantz and Dr. Yingkui Li for serving on my dissertation committee. I appreciate the invaluable guidance and insightful comments they gave to me on my research and studies. I would like to thank all professors and staff at UT and ICL for their inspiring and amazing assistance.

I would like to express my appreciation to my current and former colleagues at ICL, including Dr. Thomas Herault, Dr. Aurelien Bouteiller, Dr. Wei Wu, Dr. Chongxiao Cao, Dr. Reazul Hoque, Dr. Thananon Patinyasakdikul, Dr. David Eberius, Dr. Xi Luo, Yu Pei, Qinglei Cao for their help and company. I also thank my friend Dr. Yunhe Feng for the friendship and happiness. I wish them all the best.

Last but not least, I would like to express my deepest gratitude to my family. I would not be able to achieve anything without their love and support.

Abstract

As the scale of High-performance Computing (HPC) systems continues to grow, researchers are devoted to achieving the best performance of running long computing jobs on these systems. My research focuses on reliability and efficiency for HPC software.

First, as systems become larger, mean-time-to-failure (MTTF) of these HPC systems is negatively impacted and tends to decrease; thus, handling system failures becomes a primary challenge. My research aims to present a general design for the implementation of an efficient runtime-level failure detection and propagation strategy that is able to detect both node and process failures, targeting large-scale, dynamic systems. The strategy employs multiple overlapping topologies to optimize detection and propagation, minimizing the incurred overheads and guaranteeing the scalability of the entire framework. My design and implementation are evaluating using results from different machines using benchmarks to compare to related works. The results show that my design and implementation outperform non-HPC solutions significantly and are competitive with specialized HPC solutions that can manage only MPI applications.

Second, I endeavor to employ instruction-level parallelization to achieve optimal performance. Novel processors support long vector extensions, which enables researchers to exploit the potential peak performance of target architectures. Intel introduced Advanced Vector Extension (AVX512 and AVX2) instructions for x86 Instruction Set Architecture (ISA). Arm introduced Scalable Vector Extension (SVE) with a new set of A64 instructions. Both enable greater parallelisms. My research utilizes long vector reduction instructions to improve the performance of MPI reduction operations. Also, I use the gather and scatter feature to speed up the packing and unpacking operation in MPI. 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.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The complexity and vastness of the questions posed by modern science has fueled the emergence of an era where exploring the boundaries of matter, life, and human knowledge requires large instruments to perform the experiments, collect the observations, and, in the case of high-performance computing (HPC), perform the compute-intensive analysis of scientific data. As the march of science continues, small and easy problems have already been solved, and significant advances increasingly require tackling finer-grain problems with compute workloads, fueling an unending need for computational platforms and larger HPC systems.

In turn, facing hard limits on power consumption and chip frequency, HPC architects have been forced to embrace massive parallelism as well as a deeper and more complex component hierarchy (e.g., non-uniform memory architectures, GPU-accelerated nodes) to maintain the growth in computing capabilities. This has stressed the traditional HPC software infrastructure in many different ways and highlighted two critical issues in the last two decades: **fault tolerance**, as an encompassing term for everything related to correctness, completion, validation and verification of scientific results, and **novel programming models**, as a means of efficiently and productively developing and running large and complex applications on large and complex platforms.

1.1 Motivation

1.1.1 Resiliency

As failures become more common on large and complex systems [32], it is necessary to develop solutions to ensure applications always complete their execution correctly and that the delivered results are scientifically sound. Many such solutions have been explored, from hardening the hardware itself to replicating the applications to changing the algorithms to take advantage of natural capabilities for correctness. In the context of this dissertation, however, we were interested in solutions at the level of the programming paradigm, or at the level of runtime supporting the programming paradigm. Naturally, I turn my attention toward the Message Passing Interface, the de facto parallel programming paradigm. The MPI standard is in the process of evolving to integrate fault tolerance capabilities, as proposed in the User-Level Failure Mitigation (ULFM) specification draft [14], and various efforts to integrate checkpoint-restart with MPI [25]. The source of stress comes from programming systems that are inherently hierarchical. This has brought forth a renaissance in the field of resilience support in programming models leading to a variety of research to handle fault tolerance [21, 70, 44, 17, 43].

Thus, communities with a vested interest in fault tolerance need the capability to efficiently, quickly and accurately detect and report failures that manifest as error codes from the programming interface or trigger implicit recovery actions. Prior works [16] have designed a specialized failure detector for MPI that deploys finely tuned optimizations to provide high accuracy and few false positives, while avoiding any impact on the performance of the MPI implementation. Unfortunately, these optimizations are strongly tied to the MPI internal infrastructure. For example, a key parameter to the performance of that detector is the access to low-level remote memory access routines, which may not be typically available in a less MPI-centric context. Similar concepts could be applied to other HPC networking interfaces (e.g., OpenSHMEM), but at the expense of a significant infrastructure rewrite for each one.

Many projects have proposed fault management techniques, either automatic, driven by the application, or driven by an intermediary library. Most of these approaches rely on

their own specialized infrastructure to detect, propagate and react to failures. This leads to a large number of partial and insufficiently maintained solutions, where no portable and efficient support to build resilient applications or programming models exists. This lack of portable reliable software infrastructure also makes comparing existing or proposed solutions difficult, not only in terms of potential capabilities but also in terms of performance. Here are some examples.

ULFM provides a set of MPI interface extensions to enable MPI programs to restore MPI communication and continue the operation of programs after failures. ULFM repairs the MPI infrastructure after a failure [14]. A communicator can be reconfigured after a process failure detection, with the failed processes excluded with MPI_Comm_shrink. Missing processes can be re-spawned using the MPI function, MPI_Comm_spawn. The specialized failure detector provided in ULFM operates only on the MPI_COMM_WORLD scope and relies on non-portable optimization to mitigate issues with accuracy, as it is executed in the context of the MPI process.

OpenSHMEM is a one-sided partitioned global address space (PGAS) programming model. While OpenSHMEM does not currently have a fault tolerance model, several teams are exploring checkpoint and restart [44]. A failure detection and propagation service is needed in runtime to provide the notification to trigger the recovery. For more exploratory works, application developers can experiment with modulating the frequency and placement of restart points within the application and employ the failure detector directly or through OpenSHMEM interfaces.

EREINIT is a global-restart failure recovery model based on a fast re-initialization of MPI [25]. This work is a co-design between MVAPICH and the Slurm resource managers to add process and node failure detection and propagation features. It exhibits interesting detection capabilities, but unfortunately the implementation uses an inefficient propagation method, forcing the controller to individually send the notification and is tied to a single resource manager (Slurm). A portable fault detection and more efficient propagation are required to enable EREINIT to run on machines with different resource managers (Slurm, PBS, LSF, TORQUE, etc.) and to reduce the stabilization and recovery time of EREINIT.

DataSpaces and FTI are persistent data storage services. Fault Tolerance Interface (FTI) provides a fast and efficient multilevel checkpointing functionality [12]. Its interface lets users decide what data need to be protected and when it is reasonable to do so. The checkpointing routine then saves the marked data into a hierarchical storage using a variety of encoding and caching strategies and staging to mitigate the cost of checkpointing. DataSpaces is a data sharing framework that supports the complex interaction and coordination patterns required by coupled data-intensive application workflows [71]. It can asynchronously capture and index data, which allows for dynamic interactions and in-memory data exchanges between coupled applications.

I believe it is critical to level the field and provide a resilient, efficient and portable fault detector and propagator, integrated into one of the most widely-used parallel execution runtimes, that also allows other libraries and programming models to build on and support resilience at any scale. This runtime-level failure detector resolves the first issue and opens the gate for efficient management of failures for an emerging field of libraries, programming models and runtime systems operating on large-scale systems. Resilience support guarantees a move forward in the study of efficiency and productivity. Chapter 2 introduces failure resilience-related work, the MPI standard, the MPI reduction operation and communication operations, and one implementation of the MPI standard - Open MPI.

1.1.2 Long Vector Extension

The need to satisfy the scientific computing community's increasing computational demands drives the development of larger HPC systems with more complex architectures. This provides more opportunities to enhance various levels of parallelism. Instruction-level (ILP) and thread-level parallelism (TLP) have been extensively studied, but data-level parallelism (DLP) is usually underutilized in CPUs, despite its vast potential [20, 63, 34, 75, 57]. The most widespread vector implementation is based on single-instruction multiple-data (SIMD) extensions. Vector architectures are designed to improve DLP by processing multiple input data simultaneously 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.

A growing body of literature focuses on employing DLP via vector execution and code vectorization [19, 52, 56]; HPC, with its ever-growing demand for computing capabilities, has been quick to embrace vector processors and harness this 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 by using longer registers. Different CPU vendors follow the same trend to provide new architectures and processors with long vector extension. Intel prompted Advanced Vector Extensions (AVXs), including AVX, AVX2, AVX512. AMD's new Zen architecture supports the 256-bits AVX2 vector instructions. Arm launched Arm-v8 architectures with Scalable Vector Extension (SVE) that support vector lengths up to 2048 bits.

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

The difference between an Intel scalar code and its vectorized equivalent increased largely [59, 42, 68], highlighting the importance of employing vectorized code. The conversion of a scalar code into a vectorized equivalent can be relatively straightforward for algorithms and computational kernels, as it can be done transparently by a compiler with auto-vectorization, the compiler can provide a baseline for more complex codes. Also, developers are encouraged to offer optimized versions using widely available compiler intrinsics.

Similarly, Arm announced the new Armv8 architecture embracing SVE- a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture [7, 36]. SVE is a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture [7, 36]. 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 [15, 9]. Vector length constraints in the range from a minimum of 128 bits up to a maximum of 2048 bits in increments of 128 bits.

Long vector encapsulates more elements compared to general register. Vector horizontal reduction instructions process multiple elements concurrently, which potentiality could be used to optimize computations. Computation-oriented collective operations like MPI_Reduce performs reductions on data along with the communications performed by collectives and point-to-point. These communications 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, MPI libraries are required to provide state-of-the-art design for advanced vector extension-based versions (AVX and SVE). I tackle the above challenges and provide design and implementation for the reduction operations most commonly used by computation intensive collectives - MPI_Reduce, MPI_Reduce_local, MPI_ALLreduce.

As many scientific applications operate on multi-dimensional data, manipulating these data becomes complicated because the underlying memory layout is complex. The MPI standard proposes a rich set of interfaces to define regular and irregular memory patterns 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 [13]), 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. Long vector extensions provide rich memory access features, such as gather and scatter, which can significantly reduce the cost of non-contiguous memory operations. Chapter 2 shows a detailed introduction and related work to long vector extension within x86 and AArch64 architectures.

1.2 Contributions

This dissertation is divided into three parts: failure detection and propagation in runtime systems, computation optimization in MPI, and communication optimization in MPI. Each part is addressing one of the challenges introduced in the previous section.

1.2.1 Failure Detection and Propagation in Runtime Systems

A generic failure detection and propagation strategy (called RDaemon$^{\#}$) is implemented and delivered as an infrastructure service in the context of PRRTE. The overarching goal is to deliver a flexible and accurate failure detector while exploiting the specificities of the HPC machine model to sustain high detection accuracy and speed while incurring a limited amount of noise on the monitored application. The detailed design and implementation is discussed in chapter 3.

1.2.2 Computation Optimization in MPI

I investigate the impact of the vectorization of MPI reduction operations, and propose an implementation of predefined MPI reduction operations using vector intrinsics (AVXs 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 generic and efficient. 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. This approach is detailed in chapter 4.

1.2.3 Communication Optimization in MPI

Collective operations dealing with non-contiguous data require intensive memory management resources, which force the memory bandwidth to become the bottleneck and limit the collectives' performance. Long vector gather and scatter instructions can access multiple data from different addresses simultaneously. I take advantage of this feature to improve the packing and unpacking operation performance for non-contiguous data movement in MPI. This optimization provides high instruction level parallelism and accelerates the packing and unpacking procedure during communication, which results an efficient communication scheme of message exchanging. This approach is introduced in chapter 5.

Chapter 2

Background and Literature Review of Related Work

2.1 Overview

This chapter describes the background of this dissertation and reviews some related work. In section 2.2, I introduce the MPI standard, and an open source implementation of the MPI standard – Open MPI. Section 2.3 and section 2.4 review previous work related to fault tolerance, long vector extensions (AVX and SVE), and optimizations in MPI.

2.2 MPI

MPI stands for Message Passing Interface, which defines a library interface to describe the communication in HPC systems. It has been instrumental in permitting the efficient programming of massively parallel systems, scaling along hundreds of thousands of cores. MPI was first introduced in 1993, and at that time it mainly focused on point-to-point communications. Later, more functionalities were added to the MPI standard, such as collective operations, remote-memory access operations, dynamic process creation, parallel I/O, etc. The latest version of the MPI standard [37] (MPI-4.0 draft) was published in 2020.

2.2.1 The Open MPI Library

Open MPI [40] is an open source, freely available implementation of the MPI standard. It builds upon prior research LAM/MPI [69], LA-MPI [11], FT-MPI [35]. It starts as an all-new, production quality of MPI-2 implementation that is fundamentally centered around Modular Component Architecture (MCA) and provides both a stable platform for third-party research as well as enables the run-time composition of independent software add-ons. Open MPI provides a unique combination of novel features previously unavailable in an open-source, production-quality implementation of MPI. It is designed, developed and maintained by an active community of volunteers from academia and industry. There are three main layers in Open MPI:

- Open MPI component (OMPI). This component contains the implementations of MPI functions.
- PMIx Reference RunTime Environment (PRRTE). This component supports different back-end run-time systems.
- Open Portable Access Layer (OPAL). This component glues the code of Open MPI and PRRTE.

The Open MPI library is the foundation of my work. The reduction operation optimization and datatype pack/unpack operation optimization introduced in this dissertation are implemented in Open MPI.

2.2.2 PMIx and PRRTE

PMIx interface [24] – an abstract set of interfaces by which not only applications and tools can interact with the resident system management stack (SMS), but also the various SMS components can interact with each other. Many communication libraries, resource managers and job scheduling systems are currently employing PMIx in production, and many more are under development. Meanwhile, the Slurm batch scheduler and job starter ships with native PMIx support, meaning that an application interoperates with Slurm through PMIx.

The PRRTE runtime serves as the demonstrator and reference implementation for the PMIx specification. Technically, it is a fork of the Open RTE runtime, and thus inherits most of its capabilities to launch and monitor MPI jobs. Thanks to a well documented, and recently standardized PMIx interface, PRRTE has increased its capabilities, outgrown the MPI world it was originally designed for, and is currently capable of deploying a wide variety of parallel applications and tools. Although PRRTE provides rudimentary support for clients' fault detection and reporting, detection of failed nodes is unstable, and the reporting broadcast topology itself is not resilient, allowing process fault detection and propagation, at best. The current work expands on the existing capabilities of PRRTE by adding advanced failure detection and reporting methodologies that can efficiently operate despite the failure of the runtime daemon.

2.3 Fault Tolerance

2.3.1 Failure Detection

The areas of failure detection have been extensively studied. Chandra and Toueg [26] proposed the first unreliable failure detector oracle that could solve consensus and atomic broadcast problems for unreliable distributed systems. Many implementations [27, 51, 49] based on this oracle are using all-to-all heartbeat patterns where every node periodically communicates with all other nodes. However, these implementations, due to the communication patterns employed, are inherently not scalable beyond systems with only a few hundred nodes. An optimized version, the gossip-style protocol [74, 62, 41, 29], in which nodes randomly pick peers to monitor and exchange information with, is another popular approach for failure detection in unstructured systems where the group membership is not a priori established or varies dynamically and rapidly. Unfortunately, gossip methods perform poorly with large numbers of simultaneous node crashes, and given the random nature of the communication pattern, the time to detect a failure is not strictly bounded producing non-deterministic detection time. Furthermore, the gossip methods have the disadvantage

of generating a large number of redundant detection and gossip messages that decrease the scalability.

Recently, Bosilca proposed a deterministic failure detector for HPC systems based on network overlays [16], where each participant only observes a single peer following a recoverable ring topology. The results demonstrate the efficiency of the algorithm; however, the implementation performed at the application level in ULFM can only detect MPI process failures. The implementation employs multiple optimization techniques and shortcuts that are only possible due to its tight and deep integration within the MPI library and the availability of its highly optimized communication primitives. This resilient PRRTE work avoids these limitations and has the capability to detect both process and node failures with a smaller observation topology, and is not limited to MPI applications only.

2.3.2 Reliable Broadcast

Gossip-style [31, 29] dissemination mechanisms emulate the spread of gossip in society. Initially, members are inactive except for one member which is aware of an event of interest. It propagates this information by randomly pinging other members, until it pings someone who already was already notified. Notified members use the same strategy to gossip the information. Gossip-style is resilient to process failure and spreads quickly in the group; however, in the worst case, some members may never get notified.

Regarding deterministic reliable broadcast algorithms, a fully connected topology can handle a large number of failures but has scalability issues since it generates too many messages. At the other extreme, a mendable ring topology might be good for scalability (as each process only has 2 neighbors) but offers poor propagation latency and suffers in scenarios with multiple node failures. Circulant k-nomial graphs [6, 66] provide a balance between the previous two methods. Among circulant graphs, the binomial graph (BMG) has the lowest diameter, which minimizes the number of hops for a dissemination to reach all processes and the smallest fault diameter, which guarantees the number of hops in the dissemination path will remain scalable even when some processes on the delivery path have failed. In this work, I expand on these properties to maintain the efficiency of the

dissemination by integrating elements of the architecture hierarchy to design a multi-level propagation strategy that reduces the cost of propagation on typical HPC systems.

2.4 Long Vector Extension

In this section, I survey related work on techniques taking advantage of advanced hardware and architectures.

Petrogalli [60] gives instructions on how SVE can be used to replace and optimize some commonly used general C functions. A later work [48] explores the usage of SVE multiple vector instructions to optimize matrix multiplication in machine learning such as GEMM algorithm. Another work [10] leverages the characteristics of SVE to implement and optimize stencil computations, ubiquitous in scientific computing. This finding shows that SVE enables easy deployment of optimizations like loop unrolling, loop fusion, load trading or data reuse.

Mellanox's InfiniBand [39] explored the use of hardware scatter gather capabilities to eliminate CPU memory copies selectively, and offload data scatter and gather onto the supported Host Channel Adapter. Lim [53] explored matrix matrix multiplication based on blocked matrix multiplication improves data reuse by data prefetching, loop unrolling, and the Intel AVX-512 to optimize the blocked matrix multiplications. Dosanjh et al. [33] took advantage of using AVX vector operation for MPI message matching to accelerate matches demonstrating the efficiency of long vectors. 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. Kim [50] 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 [18] introduced a novel quicksort algorithm with a

new Bitonic sort and a new partition algorithm that has been designed for the AVX-512 instruction set, which showed superior performance on Intel Skylake in all configurations against two standard reference libraries.

Michael [47] presented a pipeline algorithm for MPI Reduce that used a Run Length Encoding scheme to improve the global reduction of sparse floating-point data. Wu [77] proposed GPU datatype engine that offloads the pack and unpack work to GPU to take advantage of GPU's parallel capability and provide high efficiency in-GPU pack and unpack. Also, Chu [28] 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 GPUDirect RDMA feature to alleviate these limitations efficiently. Luo [54] presented a new hierarchical autotuned collective communication framework in Open MPI called "HAN". HAN selects suitable homogeneous collective communication modules as sub-modules for each hardware level, uses collective operations from the sub-modules as tasks and organizes these tasks to perform efficient hierarchical collective operations. Hofmann [47] presented a pipeline algorithm for MPI Reduce that used a Run Length Encoding scheme to improve the global reduction of sparse floating-point data. Patarasuk's work [58] investigated implementations of the allreduce operation with large data sizes, derived a theoretical lower bound on this operation's communication time and developed a bandwidth optimal allreduce algorithm on tree topologies. Shan [67] 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.

Chapter 3

Failure detection and propagation in HPC systems

This chapter presents the design and implementation of an efficient runtime-level failure detection and propagation strategy for large-scale, dynamic systems that is able to detect both node and process failures. Multiple overlapping topologies are used to optimize detection and propagation, minimizing the incurred overhead and guaranteeing the scalability of the entire framework. The resulting framework has been implemented in the context of a system-level runtime PRRTE, providing efficient and scalable capabilities of fault management to a large range of programming and execution paradigms. Section 3.2 shows the experimental evaluation of the resulting software stack on different machines and programming models demonstrating that the solution is at the same time generic and efficient. Section 3.3 demonstrates that my design and implementation supports different programming models and covers different kinds of applications including one-sided and two-sided.

3.1 A Generic HPC Failure Detection Service

This section describes the design of a generic failure detector (called RDaemon$^{\#}$ in the remainder of this dissertation) that implements and delivers an infrastructure service in the context of PRRTE. The overarching goal is to deliver a flexible and accurate failure

detector while exploiting the specificities of the HPC machine model to sustain high detection accuracy and speed, while incurring a limited amount of noise on the monitored application.

3.1.1 Machine Model

I consider a machine model representative of a typical HPC system. The machine is a distributed system comprised of compute nodes with an interconnection network. Each node can host runtime daemons and one or more application processes. Daemons and processes have unique identifiers (e.g., a rank) that can be used to establish communication between any given pair. Messages take an unknown, but bounded, amount of time to be delivered (i.e., the network is pseudo-synchronous [26]). The identity and number of daemons and processes participating in the application is either known a priori or is established through explicit operations that do not require group membership discovery.

3.1.2 Failure Model

The detection strives to report crash failures, which occur when a compute entity stops emitting messages unexpectedly and permanently. A crash failure may manifest as the ultimate effect of a variety of underlying conditions, such as an illegal instruction performed because a processor is overheating, an entire node or cabinet losing power, or a software bug that manifests by interrupting a process unexpectedly or rendering some processes permanently non-responsive. The detection also distinguishes between two sub-types of crash failures: application process failures and node failures. Application process failures[1] may impact any number of hosted application processes without necessarily being concomitant with the failure of other processes, even hosted on the same node. Node failures are considered congruent with the observation of a daemon process failure. When a daemon failure occurs, all hosted application processes on that node also undergo a process failure. My work observes both types of failures. I will discuss in the following sections how this distinction helps improve the scalability of the failure detection algorithm.

[1]Note that application process failures are crash failures; this work does not deal with other types of application failures like incorrect code or dataset corruption resulting in wrong results or silent errors.

3.1.3 Notations

Table 3.1 summarizes some of the notations to describe the algorithm. The daemon is the infrastructure process deployed on each node to launch and monitor the execution of application processes on that node. The failure detector employs heartbeats between daemons and timeouts to detect node failures.

3.1.4 Detection of Process Failures

As illustrated in Figure 3.1, the failure detector employs two distinct strategies to detect process failures on one hand and node failures on the other.

To detect process failures that are not congruent with a node failure, the detection leverages the direct observation of application processes that can be performed by the node-local daemon. Since a process failure does not impact the execution of the runtime daemon managing that process, that daemon can execute localized observation operations, which are dependent upon node-local operating system services. For example, the Open RTE Daemon Local Launch Subsystem (ODLS) monitors SIGCHLD signals to detect discrepancies in the core-binding affinity with respect to the user-requested policy. That same signal also permits, from the node-local daemon, an extremely fast and efficient observation of the unexpected termination of a local application process. As a substitute, or in complement, a daemon may also deploy a watchdog mechanism [24] to capture non-terminating crash failures that may arise from software defects, such as live-locks, deadlocks and infinite loops.

3.1.5 Detection of Node/Daemon Failures

Resilient PRRTE's algorithm for node/daemon failure detection has two components: a node-level observation ring, and a reliable broadcast overlay network between daemons.

All N daemons are arranged to a logistic ring topology, as illustrated in Figure 3.2. Thus, initially, each daemon $\boldsymbol{d}$ observes its predecessor $d-1 \mod N$ and is observed by its successor $d+1 \mod N$. The predecessor periodically sends heartbeat messages to $\boldsymbol{d}$ (with a configurable period δ). At the same time, $\boldsymbol{d}$ sends heartbeat messages to its own observer. For each node, a daemon emits heartbeats m_1, m_2, ... at time τ_1, τ_2, ... to its observer $\boldsymbol{o}$.

Table 3.1: Parameters and notations.

Symbol	Description
N	Number of Daemons (or nodes)
Daemon	Runtime environment process; one per node
Process	Application process; a node may host multiple application processes
δ	Heartbeat period between daemons
η	Timeout for assuming a daemon failure
$Reported_i$	Set of failed daemon and processes identifiers known at process/daemon i

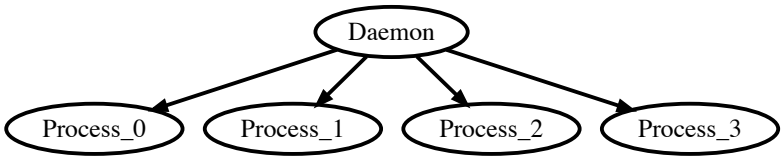

Figure 3.1: Hierarchical notification of hosted processes through PMIx notification routines. The PRRTE daemon is in charge of observing and forwarding notifications to the node-local managed application processes. The detection and reliable broadcast topology operates at the node level between daemons.

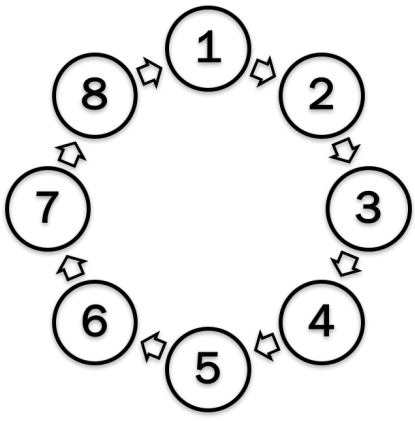

Figure 3.2: Daemons monitor one another along a ring topology to detect node failures.

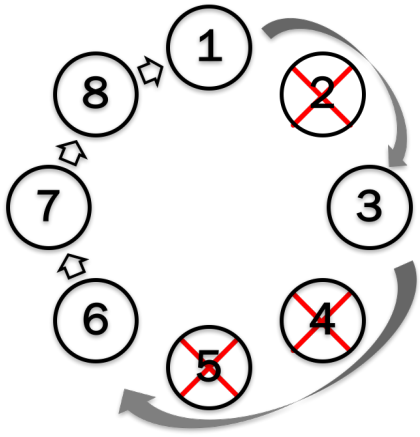

Figure 3.3: The algorithm mends the detection ring topology when a node failure occurs by requesting heartbeats from the closest live ancestor in the ring.

Let $\tau'_i = \tau_i + t$. At any time $\boldsymbol{t} \in [\tau'_i, \tau'_{i+1})$, $\boldsymbol{o}$ knows that $\boldsymbol{d}$ is alive if it has received the heartbeat message m_i or higher. Otherwise, $\boldsymbol{o}$ suspects that $\boldsymbol{d}$ has failed and initiates the propagation of the failure of $\boldsymbol{d}$.

When the observer detects that its predecessor has failed, it undergoes two major steps. First, it needs to reconnect the ring topology, as illustrated in Figure 3.3. Daemon $\boldsymbol{o}$ tries to observe the predecessor of $\boldsymbol{d}$ (the daemon it previously observed). It sets ***d-1*** as its new predecessor and then sends a request to ***d-1*** to initiate heartbeat emission. Of course, it is possible that ***d-1*** has also failed, which will be detected at the next timeout. In order to speed up the reconnection process, $\boldsymbol{o}$ may skip over daemons that have already been reported as failed in the past (i.e., daemons whose identifier is in $Reported_o$ because they have been observed and reported by another daemon). Each time a daemon is marked as failed, all the processes it managed are also marked as failed. After getting the list of all those affected processes and nodes, the observer component calls the propagation component to broadcast the fault information to other daemons and then notify its local processes.

3.1.6 Broadcasting Fault Information

Considering that the observation topology is static, it does not provide automatic or probabilistic dissemination of fault information. Thus, to complete the reporting of failures, failures identified by an observer must be broadcasted to inform all other daemons and application processes. An important aspect when considering a runtime that tolerates node/daemon failures is that the propagation algorithm itself needs to be resilient to failures.

For broadcasting fault information between daemons, the algorithm uses the scalable and fault-tolerant BMG topology [6]. BMG has good fault-tolerant properties such as optimal connectivity, low fault-diameter, strongly resilience and good optimal probability in failure cases. Note that unlike prior works, the propagation Algorithm 1 is not a flat BMG between application processes, but rather, it consists of an inner BMG overlay between daemons and an outer star overlay from each daemon to its locally managed processes.

Figure 3.4 shows an example of the execution of the BMG broadcast with 12 nodes. For simplicity, the local stars connecting each daemon to its local processes are not represented.

Algorithm 1 Two-Level Reliable Broadcast Algorithm.

N ▷ Number of nodes (value from environment)
Eid ▷ Identifier of a process observed as failed (input parameter)
Reported$_i$ ▷ Set of identifiers of previously reported failures, local to daemon **i** (initially empty)
msg ▷ Message containing the set of process identifiers to report (initially empty)
Hosted{*Did***}** ▷ Set of process identifiers managed by the daemon *Did* (initially empty, obtained from environment)

1: **procedure** StartPropagation(*Eid*) ▷ Daemon **i** starts the propagation
2: **if** ($Eid \notin Reported_i$) **then** Add *Eid* to *msg*
3: **if** *Eid* is a daemon **then**
4: Obtain $Hosted\{Eid\}$
5: add $Hosted\{Eid\}$ to *msg*
6: ReliableBroadcast(i, N, msg)
7: Add *msg* to $Reported_i$

1: **procedure** ReliableBroadcast(i, N, msg) ▷ Daemon **i** sends error messages to all its neighbors
2: **for** $k \leftarrow 0$ to $\log_2 N$ **do** ▷ Neighbors in the BMG
3: **i** sends **msg** to ($(N + \text{i} + 2^k)$ **mod** N)
4: **i** sends **msg** to ($(N + \text{i} - 2^k)$ **mod** N)
5: **for all** $lp \in Hosted\{i\}$ **do** ▷ Local application processes
6: **i** sends **msg** to **lp**

1: **procedure** Forwarding(*msg*) ▷ Triggered when daemon or process **j** receives *msg*; decides if the message needs to be forwarded and notified locally
2: **if** $msg \not\subset Reported_j$ **then**
3: **if** j is a daemon **then**
4: ReliableBroadcast(j, N, msg)
5: Add *msg* to $Reported_j$

1. In this example, daemon 0 is the initial reporter, and its observer component starts the propagation by calling the StartPropagation reliable broadcast algorithm.
2. This prepares a broadcast message containing the identifier of the failed process (or daemon) and the associated application processes, when relevant. Daemon 0 issues the message to its neighbors in the BMG topology.
3. Upon receiving a broadcast message, a daemon considers if the message needs to be forwarded. If the message carries a list of processes that are already known to have failed, then the daemon already triggered the propagation and no further action is needed. Thus, every daemon forwards the message once, ensuring that all edges of the BMG carry exactly one message per detection.

The propagation message issued at each daemon is ordered so that the messages sent first are part of a binomial spanning tree rooted at the emitter. Figure 3.5 shows the spanning tree for a broadcast originating from node 0; the redundant messages (colored in blue) are extra messages that provide reliability and ensure that any node in the BMG can always be reached within $O(log_2 N)$ steps (given that less than $2log_2 N$ failures strike; with more failures, statistically rare scenarios can degenerate in a linear propagation time). The advantages of this new broadcast algorithm are:

1. Sequence ordering brings higher parallelism. messages to node {10, 11, 7} can arrive from any redundant forwarding path rather than only from the 0-rooted spanning tree. This may decrease the apparent height of the tree, and reduce the average notification latency.
2. Limited network degree: the maximum degree for every daemon is logarithmic, which avoids hot-spot effects that are common in randomized gossip algorithms.
3. Deterministic number of messages: the total number of messages is exactly the number of links in the BMG topology, that is, $O(Nlog_2 N)$ messages overall. In contrast, random march gossip algorithms have to balance between the probability of not reaching every participant and the number of messages.
4. The number of heartbeats and propagation messages is dependent upon the number of nodes, not the number of managed application processes. In manycore systems, this

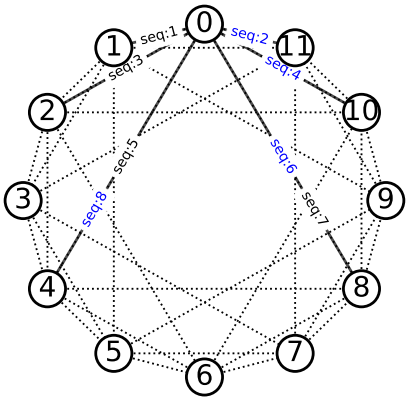

Figure 3.4: Binomial graph with 12 nodes with messages sent from 0 highlighted.

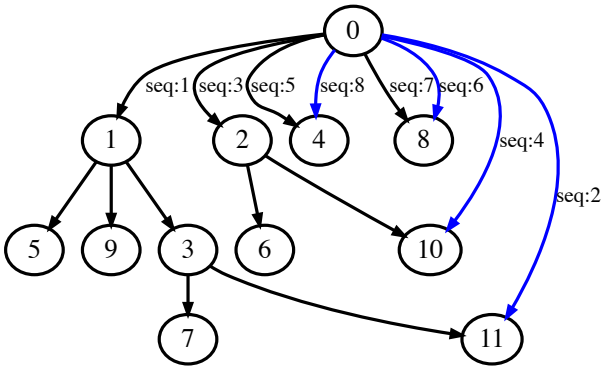

Figure 3.5: Binomial spanning tree in broadcast from node 0, redundant messages from 0 are colored in blue.

can significantly reduce the effective cost of the algorithm when compared to a flat topology between application processes.

3.1.7 PMIx Interface

RDaemon# is implemented as a set of components in PRRTE. PRRTE is developed and maintained by the PMIx community as a demonstrator and enabler technology that demonstrates and exercises the features of the PMIx interface [24]—an abstract set of interfaces by which not only applications and tools can interact with the resident system management stack (SMS), but also the various SMS components can interact with each other. Many communication libraries, resource managers and job scheduling systems are currently employing PMIx in production, and many more are under development.

For example, Open MPI has now substituted Open RTE with a shim layer over PMIx; therefore, it can be launched and monitored by PRRTE. Similarly, OpenSHMEM uses PRRTE as the default launcher. Meanwhile, the Slurm batch scheduler and job starter ships with native PMIx support, meaning that an application that interoperates with Slurm through PMIx can be ported over PRRTE without effort.

RDaemon# leverages the interfaces specified by PMIx [23] to interoperate with the client application, communication library, or programming language, as well as with the SMS. To the best of my knowledge, RDaemon# is the first implementation to populate the PMIx interfaces with a truly resilient implementation. An important feature of the interface is the PMIx Event Notification [22], which performs the local propagation of failure information from the daemon to the client processes.

3.1.8 RDaemon# in the PRRTE Architecture

While a full depiction of the architecture and feature set of PRRTE is out of the scope of this paper, some are relevant to my implementation effort. PRRTE is based on a Modular Component Architecture (MCA) which permits easily extending or substituting the core subsystem with experimental features. As shown in in Figure 3.6, within this architecture,

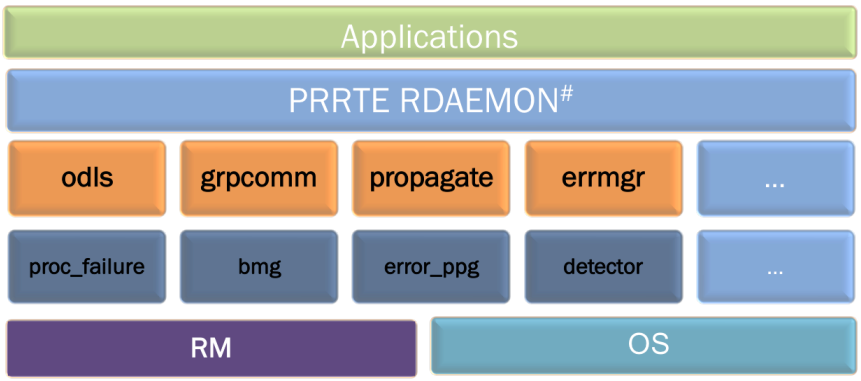

Figure 3.6: Resilient PRRTE architecture. The orange boxes represent components with added resilience features. The dark blue colored boxes are new modules.

each of the major subsystems is defined as an MCA framework with a well-defined interface and multiple components implementing that framework can coexist.

Two new frameworks and four components are added to PRRTE daemons. The `proc_failure` component is in charge of detecting the failure of locally hosted processes (using SIGCHLD signals from the operating system). The `BMG` component implements a broadcast algorithm in a reliable way; this component abides by the normal interface for a daemon broadcast and can reliably broadcast any type of information. The `detector` component emits heartbeats and monitors timeouts. Last, the `error_ppg` component prepares the content of the reliably broadcast messages (i.e., the list of failed processes). In order to populate the list of failed processes in node failure cases, the list of processes hosted by a particular daemon needs to be obtained (line 5 of procedure StartPropagation in Algorithm 1). This information is queried from the key-value store of PMIx. Note, however, that multiple daemons querying that information could cause a storm of network activity within the SMS in order to fetch this information or require its replication (memory overhead). Fortunately, when a given daemon is observed by a single other daemon, there is a single initiator to the propagation routine, and this potential non-scalable usage of the PMIx key-value store can be avoided.

3.2 Experimental Evaluation

3.2.1 Experimental Setup

Experiments are conducted on two different machines: (1) ICL's NaCl is an Infiniband QDR Linux cluster comprising 66 Intel Xeon X5660 compute nodes, 12 cores per node; (2) NERSC's Cori is a Cray XC40 supercomputer with Intel Xeon "Haswell" processors and the Cray "Aries" high speed inter-node network, 32 cores per node. My RDaemon$^{\#}$ is based upon PRRTE (#71ef547) with external PMIx (#21d7c9). It is compared with ULFM revision #77f9157, which is based on the same base version of Open MPI to evaluate RDaemon$^{\#}$ in MPI workloads. Each experiment is repeated 30 times and the average is used here. Intel MPI Benchmark (IMB v2019.2) [1] is used for MPI performance

measurements for point-to-point (P2P) and collective communications (one MPI rank per core). All experiments use the map-by node, bind-to core binding policy, which puts sequential MPI ranks on adjacent cores. The only exception is the IMB P2P experiment where it uses the map-by node, bind-by node policy to set communicating MPI ranks on different nodes.

3.2.2 Accuracy

The first experiment explores the accuracy of RDaemon$^{\#}$'s detector. The accuracy experiment is conducted by (1) Starting with a large value for the detection timeout η; (2) Verifying that no failure is detected when there is no injection and that all injected failures are reported; (3) If the previous test is accurate, decrease η (and accordingly the heartbeat period δ) until false positive detection is noticed. The constant ratio is $\eta = \delta * 2$. This methodology exposes the behavior in normal deployment (100ms period) as well as the behavior at the limit for very short η timeout values (in the order of milliseconds). Figure 3.7 presents the results on NaCl 64 nodes. In heavily communicating benchmarks (IMB point-to-point and collective tests), all tests succeed until the heartbeat period is lower than 20 milliseconds. To further investigate, the heartbeat message is neither delayed by communication congestion nor compute pressure, but daemons need some time to launch the processes when starting the job, which causes heartbeat delay and false detection during job startup.

3.2.3 Noise

I also investigate the noise overhead incurred on an MPI application by the heartbeat emission and management from RDaemon$^{\#}$. Figure 3.8 illustrates the overhead incurred with P2P and collective communications running IMB. In order to contextualize the incurred overhead, the band of natural variability of the benchmark without an active failure detector is shaded in gray ($average \pm \sigma$), and, for clarity, error bars are plotted for $\delta = 1ms$, the only cases where the variability exceeds the natural variability of the benchmark at sometime. PingPong benchmark uses the `-multi` mode of IMB with one rank per core on 2 nodes.

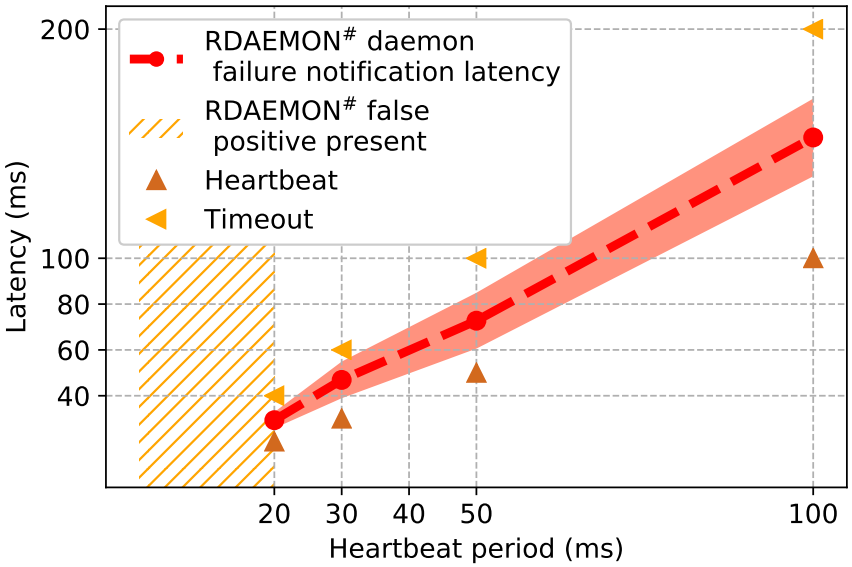

Figure 3.7: Accuracy with short detection heartbeat and timeout.

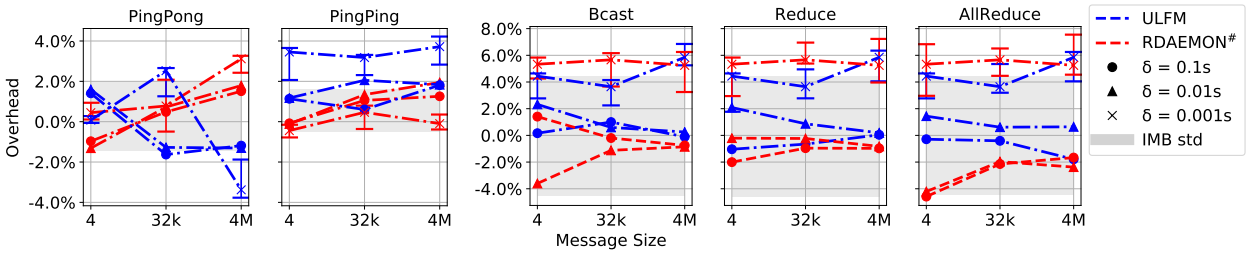

Figure 3.8: PRRTE with fault tolerance overhead over PRRTE and ULFM using IMB.

This ensures that all cores are active with the communication pattern and thus compete for resources with RDaemon$^{\#}$ activities. Collective benchmarks run on 64 nodes using all cores. Each message size sets the number of repetitions for the test to last at a minimum 20 seconds so that multiple heartbeat emissions occur during the experiment. Overhead is calculated by using the maximum latency result, normalized by the non-fault tolerant performance:

$$Overhead = \frac{(\text{RDaemon}^{\#} - \text{PRRTE})}{\text{PRRTE}} \tag{3.1}$$

From the graph we can see that the latency performance and bandwidth performance are barely affected, with the heartbeat period ranging from milliseconds to seconds. Notably, when $\delta \geq 10ms$, it has trivial influence on the system, as illustrated by the fact that the average overhead is within the band of natural variability of the benchmark. When $\delta = 1ms$, the incurred noise varies in a band that increases the PingPong latency by up to three percent. In collective communication, the noise overhead is less than eight percent, which, at four percent, is slightly higher than the standard deviation of the benchmark itself. In a general comparison with ULFM (normalized to its performance without failure detection active), we can see that RDaemon$^{\#}$ achieves a similar level of incurred noise for a given heartbeat period and communication pattern.

3.2.4 Comparison with SWIM

This section compares the failure detection latency and scalability of RDaemon$^{\#}$ with SWIM [29]—a random probing-based failure detection protocol with gossip membership updates. To decrease the chance of false detection, SWIM uses a suspicion mechanism. When a node does not reply to a probing in time, the initiator then judges this node as suspicious (but not yet failed). It then broadcasts this suspicion information within a subgroup: if any node in the subgroup receives an acknowledgment before the timeout, it declares the suspected node as alive; otherwise it declares a failure. In order to improve the efficiency of multi-cast, SWIM uses the infection-style dissemination mechanism and piggybacks the information to be disseminated in the detection's pings and acknowledgment messages. For

the SWIM implementation, it uses Go-Memberlist (#a8f83c6), and a go-MPI interface is used to replicate the MPI detection benchmark.

Figure 3.9 compares the scalability of the two detectors with regard to the number of deployed processes with $\eta = 1s$, $\delta = 0.5s$. SWIM tests only up to 256 members; after that limit, some nodes exceed the maximum connection backlog set in the operating system for `listen` operations on TCP sockets, causing an application crash during initialization. For RDaemon$^{\#}$, tests run up to 768 processes on 64 nodes. As the number of processes increases, latency of RDaemon$^{\#}$ remains almost the same. For 4K processes, the stabilization of RDaemon$^{\#}$ is still below the range of the heartbeat period and timeout. SWIM latency shows a linear increase when the number of processes increase, which becomes the bottleneck when scaling up (assuming the limits on maximum connection requests issue can be solved).

Figure 3.10 compares single node failure detection and propagation latency between RDaemon$^{\#}$ and SWIM with different heartbeat period settings. All tests set $\eta = \delta * 2$. The experiment uses 64 nodes in both cases; RDaemon$^{\#}$ deploys on all 768 cores, while SWIM uses only 256 cores because it cannot deploy with more processes, as discussed above. We can clearly see that for RDaemon$^{\#}$ the detection latency is between (δ, η), and the last notification happens very soon after the detection, demonstrating the efficiency of my propagation algorithm (variability in the results comes from the randomness of when the node failure happened with respect to the heartbeat period). However, for SWIM, even considering the advantage of managing a smaller number of processes, the latency is still more than $10 * \delta$, because after the initial timeout declares a suspicion, the gossip protocol and confirmation mechanism have to be executed before the failure is reported.

3.2.5 Comparison with ULFM for Process Failures

This section compares RDaemon$^{\#}$ with the other extreme on the spectrum of general versus specialized—ULFM. The ULFM implementation also has two main components: process-level detection ring and propagation overlay with all launched processes. The detection ring is built at Byte Transfer Layer (BTL) level, which provides the portable low-level transport abstraction in Open MPI. ULFM's current implementation provides several mechanisms to ensure the timely activation and delivery of heartbeats:

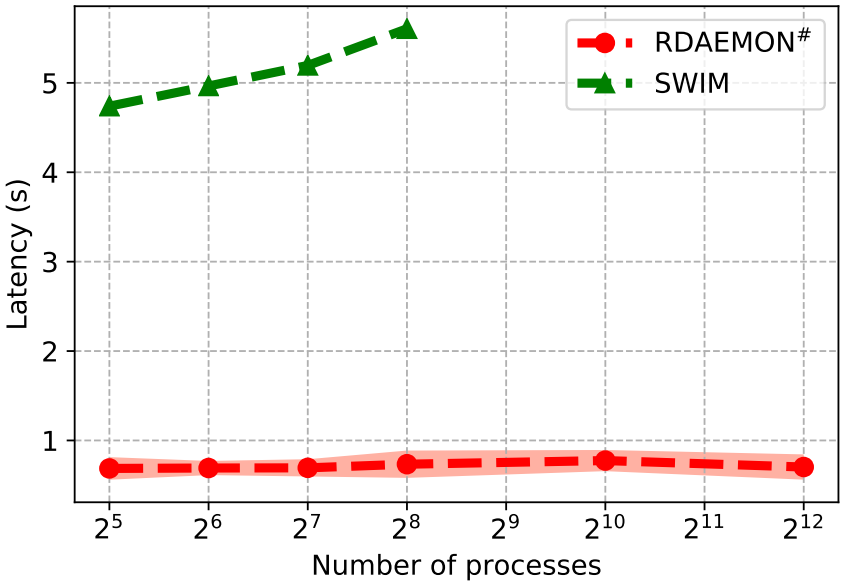

Figure 3.9: Detection latency comparison between RDaemon# and SWIM with increasing number of processes ($\delta = 0.5s$).

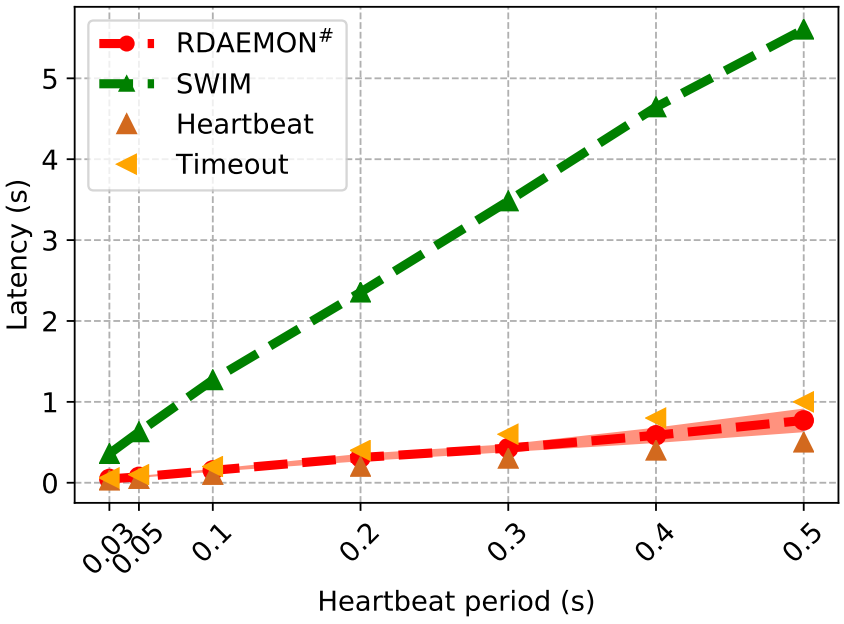

Figure 3.10: Detection and Propagation delay comparison between RDaemon# and SWIM with varying heartbeat period.

1. Using a separate, library-internal thread to send the heartbeats in order to be separated from the application's communication. This also mitigates the drift in heartbeat emission dates (which would cause false positive detection) in compute-intensive applications. The receiver then needs to poll the BTL engine to check the aliveness of its successor.
2. Using RDMA put to raise a flag in the receiver's registered memory. By using the hardware accelerated put operations, ULFM avoids the problem of actively polling the BTL engine.
3. Using in-band detection directly from the high-performance network fabric to report unreachable error directly to the propagation component.

The propagation overlay is also built at the BTL level. Reliable broadcast messages are sent using the same active message infrastructure employed to deliver short MPI messages and matching fragments; a different tag is employed to avoid disrupting the MPI matching, however. Because the propagation happens at the application process level, all MPI processes are part of the reliable broadcast algorithm; thus, the lower bound for reaching all processes is $\log_2(Number\ of\ Processes)$.

In contrast, RDaemon#'s process failure detection is implemented at the daemon level. This mechanism doesn't pressure the application communication resources, and can continue processing heartbeats without the need for RDMA hardware. The broadcast overlay in RDaemon# is built at the daemon level which decreases the number of participants to the number of nodes—a potentially large saving in many-core systems. This helps reduce the total messages transferred and forwarded compared to ULFM. The the lower bound for a full propagation is $\log_2(Number\ of\ Nodes)$.

Figure 3.11 compares the latency of process failure detection and propagation between ULFM and RDaemon#. For process failures (as opposed to node failures), both RDaemon# and ULFM rely on non-heartbeat–based detection. ULFM uses the shared-memory transport (SM BTL) between co-hosted processes, and this BTL features a very rapid (almost instantaneous) in-band reporting of the endpoint failure. For RDaemon#, the daemons detect process failures with operating system signals. So, this process failure

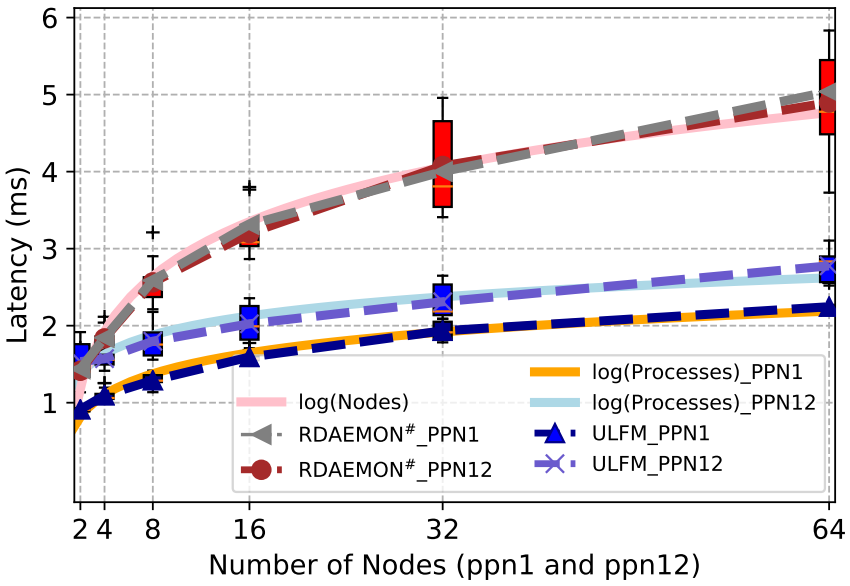

Figure 3.11: Process failure detection and propagation delay compared to ULFM.

experiment does not measure the effectiveness of the heartbeat mechanism (and timeout). Instead, we stress the broadcast component exclusively.

Experiments are conducted on NaCl up to 64 nodes using all 12 cores on each node. The process mapping results in ULFM perform a large part of the propagation between co-hosted processes (using the SM BTL transport) and employ InfiniBand communication for inter-node messages. RDaemon$^{\#}$ uses TCP to broadcast between daemons, and each daemon uses a PMIx notification to distribute the error information to all hosted processes. We can see that my implementation is slightly slower than ULFM for process failure case, but it greatly reduces the complexity. The detection and propagation time is less than 5 milliseconds despite using TCP. For ULFM the detection and propagation delay increases from 2 milliseconds to 3 milliseconds as the number of processes increases. For both RDaemon$^{\#}$ and ULFM the latency increase trend fits $a * \log_2(N) + b$, which can be easily scaled up to hundreds of thousands of nodes, but for ULFM the trend follows the number of processes rather than the number of nodes.

To further validate the logarithmic trend of RDaemon$^{\#}$ scalability, the experiments scale the evaluation on the larger Cori system (with more processes per node). We can see in Figure 3.12 that with 4K processes the detection and propagation latency is about 10 milliseconds, and the scalability trend remains logarithmic with the number of nodes (not processes).

3.2.6 Node Failures Detection

The detection latency is compared for full-node failures. In RDaemon$^{\#}$, node failures result in the loss of a daemon. For ULFM, they result in the loss of multiple consecutive processes in the ring topology. In both cases, the node failure is detected by the absence of heartbeats before the timeout expiration.

Figure 3.13 presents the behavior observed when injecting a single daemon failure under different heartbeat period settings. Experiments are conducted on 64 nodes with 764 processes. For RDaemon$^{\#}$, after synchronizing, a node crash is injected by ordering a process to kill its host daemon. For ULFM, all application processes on the target node

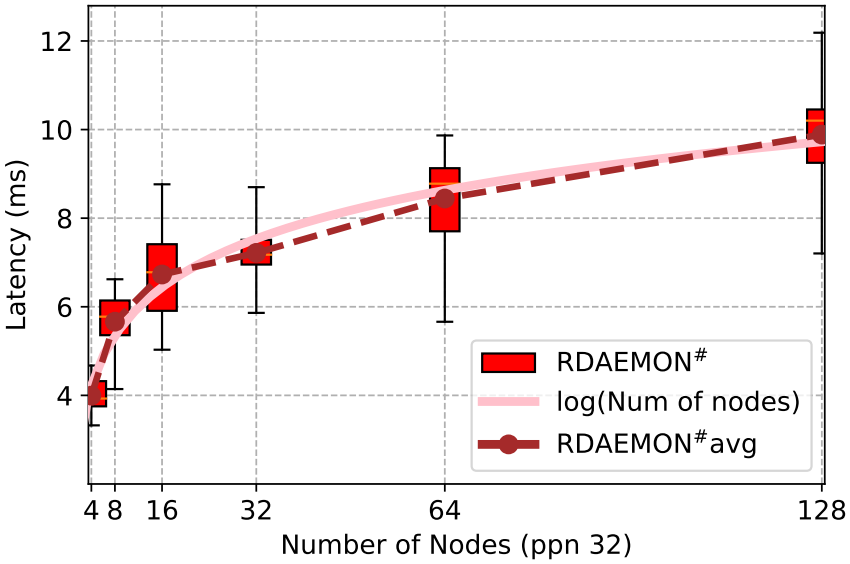

Figure 3.12: Process failure detection and propagation delay on Cori.

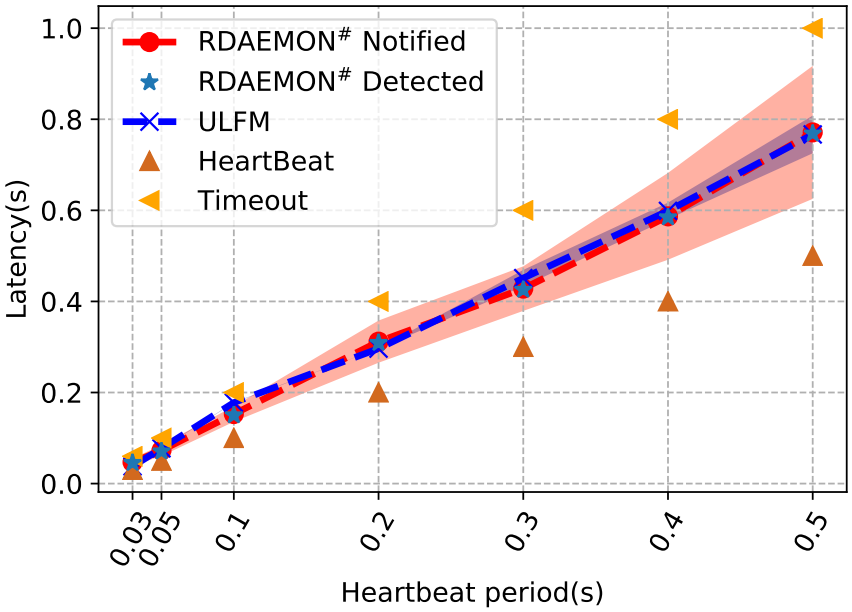

Figure 3.13: Single Daemon Failure detection and propagation delay compared to ULFM with different heartbeat period.

commit suicide as a group. The heartbeat period setting starts from 30 milliseconds to 0.5 seconds for both RDaemon$^{\#}$ and ULFM. For all heartbeat periods, it sets $\eta = \delta * 2$. From the figure, we can see that the detection latency in all cases lands in the interval $[\delta, \eta]$.

Figure 3.14 shows performance of single node failure detection and propagation with a fixed heartbeat period $\delta = 0.5s$ and an increasing number of total nodes. After a node crash, all processes hosted on this node are affected. The observer node fetches and packs the information of all affected processes and then distributes the packed message. From the figure we see that RDaemon$^{\#}$ can detect and propagate a node failure between (0.5s, 1s) for every number of nodes tested.

The last experiment (presented in Figure 3.15) investigates the effect of multiple concurrent node failures. The experiment is similar to the single node failure case, except for the number of processes that inject failures. I first consider the worst-case scenario, in which failures strike contiguous nodes. In this case, the daemon that detects the first failure undergoes the ring-mending operation, which enacts a linear number of timeouts before all failures are notified. Note that ULFM exhibits the same behavior, even for single node failures. In the map-by-slot binding policy, consecutive ranks fail simultaneously with a node failure. From a fault tolerance perspective, daemons on the detection ring should be ordered to avoid setting nodes with a correlated chance of failure sequentially (e.g., avoid choosing predecessor and successor from the same cabinet). This is easier to achieve when the detection infrastructure is split from the MPI ranking. To study the average behavior, failures are also injected at random nodes. In this case, detection and propagation are independently conducted by different observer nodes and neatly overlap resulting in a marginal increase in the overall detection latency for reporting all failures.

3.3 Communication Models Coverage and Application Evaluation

Nowadays, more and more systems in HPC feature a hierarchical hardware design; shared memory nodes with several multi-core CPUs are connected via a network infrastructure.

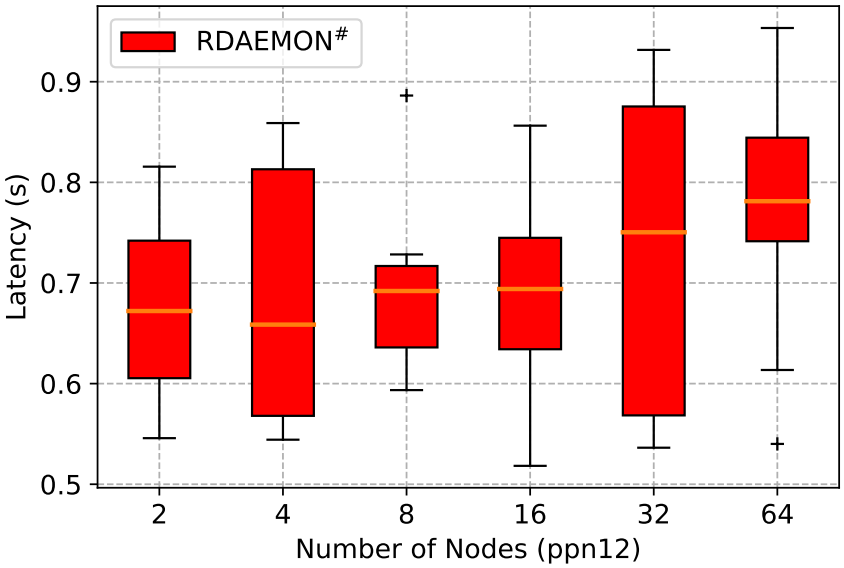

Figure 3.14: Single Daemon Failure Detection and Propagation delay with different number of nodes ($\delta = 0.5s$).

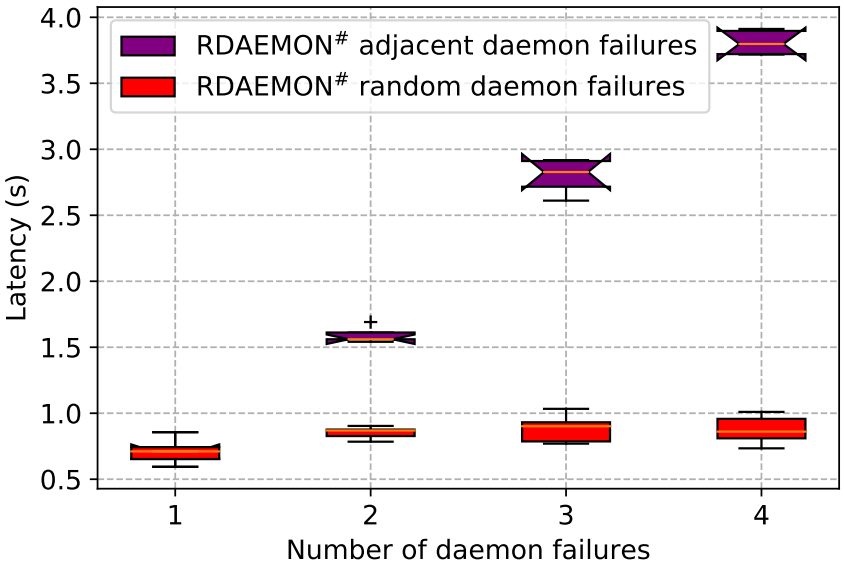

Figure 3.15: Multiple daemon failures at the same time ($\delta = 0.5s$, 64 Nodes).

This trend has disrupted the long status-quo in which parallel applications are written in MPI and has promoted the emergence of multiple alternatives for programming parallel systems. On one hand, some programming styles combine shared memory parallelization, or GPU acceleration inside each node, and distributed memory parallelization between nodes separated by the interconnect. On the other hand, parallel applications may alternate between library calls that utilize different programming environments and programming models to perform internode communication. For example, message passing and parallel global address space models may coexist in the same application. Consequently, the runtime environment needs to handle cooperation between different programming models. Together, failure detection and management techniques need to be expanded across different models.

This section investigates application support of RDaemon$^{\#}$ with different programming models. For example, MPI has the standard MPI_Init function that must be called to initialize the library, providing a "hook" within that function to notify others that it has been called. In contrast, OpenMP does not have an explicit call to "init" and is instead initialized on first use; older versions of OpenSHMEM also allow implicit initialization. Figure 3.16 shows how to coordinate between two different models. We can see that as both communication libraries employ the PMIx library to interface with the runtime and job scheduling system, the different programming languages have a common interface to exchange information. The calls into PMIx_Init from each programming model enters the same code space and offers an opportunity for coordination. The event notification mechanism within PMIx can then be used to share the information and coordinate between those models.

In practice, PRRTE supports different types of applications when launching a single PRRTE Distributed Virtual Machine (DVM) (using the ***prte*** command), and then uses the ***prun*** launcher to execute the binaries, as long as they are compiled in the following fashion:

1. PMIx-based application use ***pcc*** for compilation;
2. MPI applications need to install MPI and RDaemon$^{\#}$ with the same external PMIx, then use ***mpicc*** for compilation;

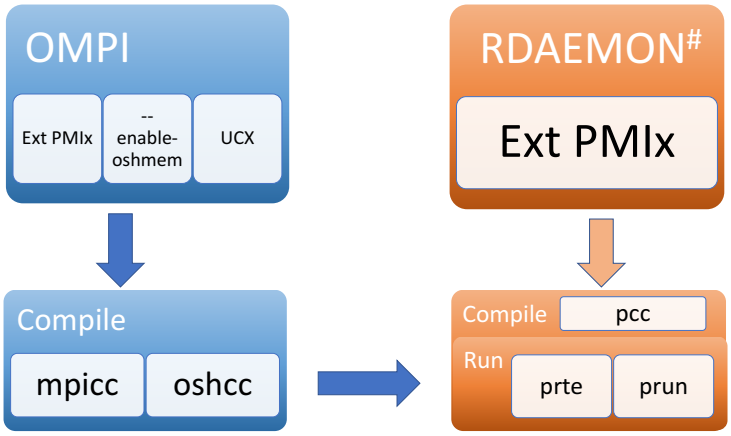

Figure 3.16: Hybrid programming model support of MPI and OpenSHMEM

3. OpenSHMEM applications need to install OpenSHMEM and RDaemon# with the same external PMIx, then use ***oshcc*** for compilation. In Open MPI, MPI +OpenSHMEM programs are directly supported when compiling with OpenSHMEM support (using the option ***-enable-shmem***).

To evaluate the overhead on performance from RDaemon# in MPI and OpenSHMEM applications, I use the heavily communication-bound benchmark Graph500 [5]. Graph500 is an open specification effort to offer a standardized graph-based benchmark across large-scale distributed platforms, which captures the behavior of common communication-bound graph algorithms. Graph500 differs from other large-scale benchmarks such as HPL, and HGPGMG in the way it primarily highlights data access patterns. Graph500 performs a breadth-first search (BFS) in parallel on a large randomly generated undirected graph. The experiments use the open source project OpenSHMEM Benchmark (OSB) suite [30] that features both MPI and OpenSHMEM based Graph500 implementations. For the application setting it uses $scale\_factor = 20$ **and** $edge\_factor = 16$, which generates an undirected graph with 2^{scale_factor} vertices and $2^{scale\_factor} * edge\_factor$ edges. The benchmark collects the statistics of the generation of the breadth-first search tree of 64 randomly selected vertices. It also collects the statistics of the validation time, which ensures that all connected components which generate large amounts of communications are visited. The experiments use NERSC Cori with 1K nodes. This results in a deployment with 32K MPI ranks, or 32K OpenSHMEM Processing Elements (PEs).

3.3.1 Two-sided Application

The ***mpi_test_simple*** benchmark is the baseline implementation of the BFS that uses two-sided MPI_Send, MPI_Recv and MPI_Allreduce. I evaluate the noise overhead incurred from heartbeat messages with different heartbeat periods based on the point-to-point (P2P) and collectives used in this benchmark.

Figure 3.17 shows the overhead incurred with the P2P communication during the BFS generation phase. Presented in shaded gray, the variability of the BSF without the heartbeat detection enabled ($mean\_time\_of\_BFS \pm \sigma$). Overhead is calculated the same as in equation (3.1). For comparison, I plot the overhead with error bars for different δ values. In

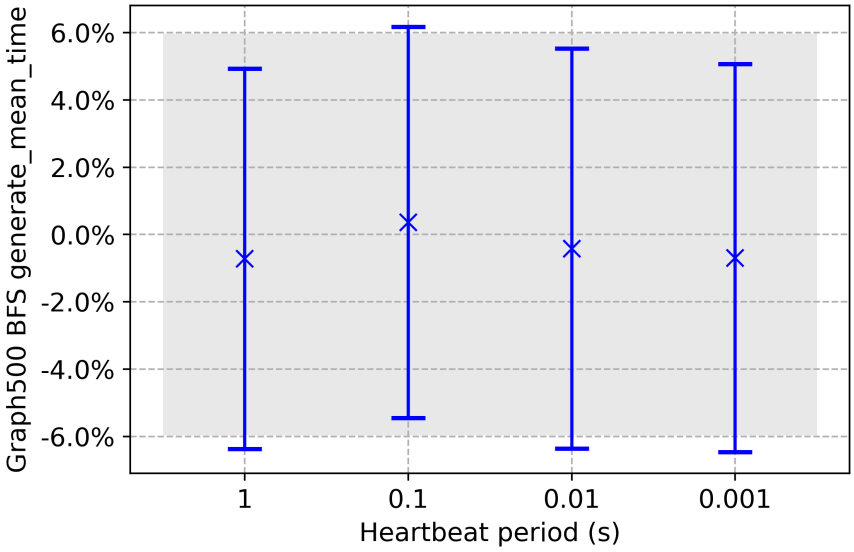

Figure 3.17: Overhead for generating BFS running mpi_test_simple when using PRRTE with fault tolerance over PRRTE (32K MPI ranks; the gray area represents the normal variability of the benchmark).

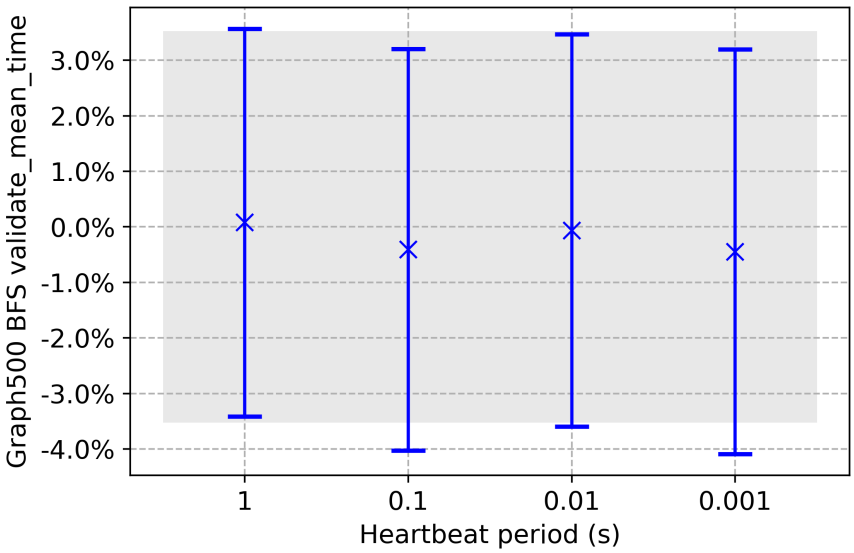

Figure 3.18: Overhead for validating BFS in mpi_test_simple when using PRRTE with fault tolerance over PRRTE (32K MPI ranks; the gray area represents the normal variability of the benchmark).

all cases, the variability without the detector active is comparable to the maximum spread of the overhead when fault tolerance is enabled, and the average overhead is close to 0. Figure 3.18 shows the overhead incurred in the MPI_AllReduce during the validation phase. Again, the application with failure detection enabled achieves the same performance, which demonstrates that my failure detection heartbeats have minimal impact in communication-intensive applications with both P2P and collective communications.

3.3.2 One-sided Application

For the OpenSHMEM application, it selected the implementation of ***graph500_shmem_one_sided*** that is derived from the MPI-2 one-sided code base. For the communication it uses shmem_put/getmem, which are similar to MPI_put/get. It also uses a shmem reduce collective as a replacement for MPI_AllReduce. Figure 3.19 and Figure 3.20 show the overhead of those two types of communications during BFS generation and validation. Again, for all different heartbeat periods, they show similar trends in which my detector does not stress the applications' communication.

As a result, The algorithm is integrated within PRRTE so that the detection service can be employed by a wide variety of clients through a well specified and popular interface (PMIx). The process and node failure detection strategy presented in this work depend on heartbeats and timeouts. My design and implementation takes into account the intricate relationships and trade-offs among system overhead, detection efficiency, and risks: low detection time requires frequent emission of heartbeats messages, increasing the system noise and the risk of false positive. My solution addresses those concerns and is capable of tolerating a high frequency of node and process failures with a low-degree topology that scales with the number of nodes rather than the number of managed processes. The results from different machines and benchmarks compared to related works shows that RDaemon$^{\#}$ outperforms non-HPC solutions significantly, and is competitive with specialized HPC solutions that can manage only MPI applications. At the same time, I demonstrate in application benchmarks that my detector can sustain the operation of MPI and non-MPI applications (like OpenSHMEM) with no noticeable overhead. Thus, this runtime-level

failure detector opens the gate for efficient management of failures for an emerging field of libraries, programming models, and runtime systems operating on large-scale systems.

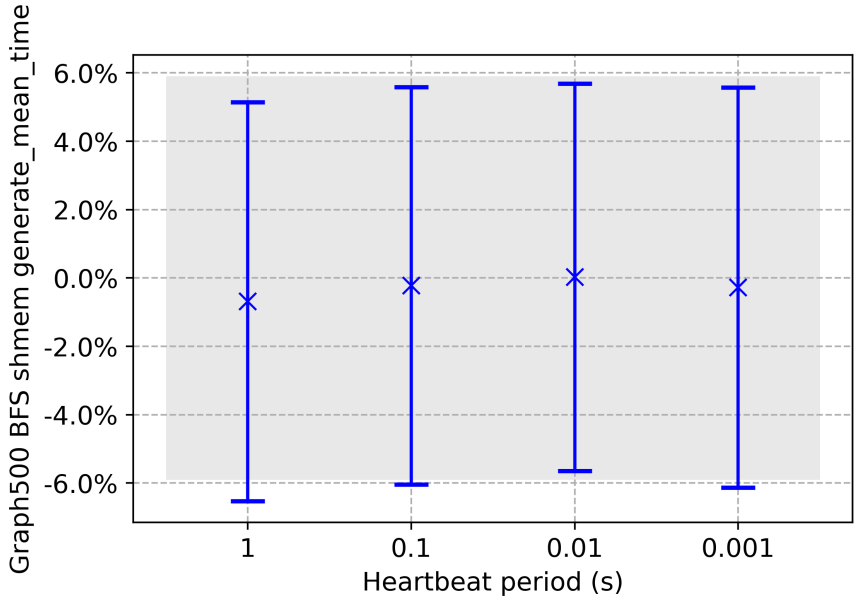

Figure 3.19: Overhead for generating BFS running graph500_shmem_one_sided upon PRRTE with fault tolerance over PRRTE (32K OpenSHMEM PEs; the gray area represents the normal variability of the benchmark).

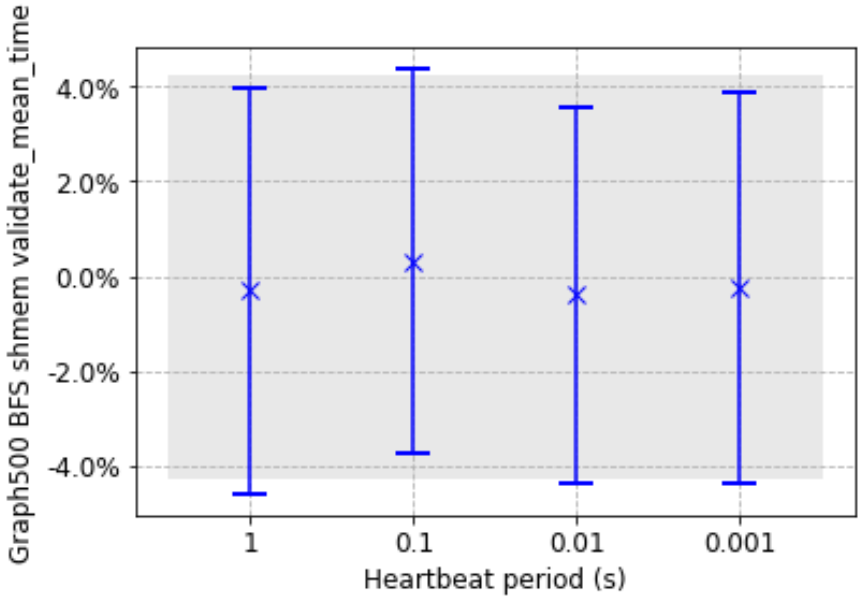

Figure 3.20: Overhead for validating BFS running graph500_shmem_one_sided upon PRRTE with fault tolerance over PRRTE (32K OpenSHMEM PEs; the gray area represents the normal variability of the benchmark).

Chapter 4

Reduction Operation Using Long Vector Extension

4.1 Overview

Different techniques can be roughly classified according to the level at which the hardware supports parallelism with multi-core and multi-processor computers that have multiple processing elements within a single machine. Different levels of parallelization, including bit-level, instruction-level, data-level and task parallelism, are studied here.

Novel architectures and processors integrate with long vector extension. This extension provides the possibility of further parallelization in MPI. It will be crucial for many applications to have a highly optimized version of reduction operations, which creates a challenge of improving the performance of the predefined MPI reduction operations. I tackle the above challenges and provide design and implementation for vector based reduction operations, which are most commonly used by the computation collectives - MPI_Reduce and MPI_Allreduce. Multiple MPI reduction methods fully take advantage of long vector extension capabilities to efficiently perform these operations.

This chapter describes the approach to implement long vector based reduction operations. The new approach uses AVXs and SVE to design and implement long vector based reduction operations, and integrate the operations in Open MPI. Section 4.2 introduces the concept and implementation of the vector based reduction operations. Section 4.3 explains the

benchmark evaluation results of my design. Section 4.4 displays the performance tools and evaluates the results of my design. Finally, section 4.5 evaluates the performance of this implementation with HPC and machine learning applications.

4.2 Design and Implementation of Vector Based Reduction

4.2.1 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 256-bit YMM registers, 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 and 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) encapsulating eight 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 can 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 versions. As an example, applications using AVX and SSE instruction simultaneously suffered performance penalties, while mixing AVX-512 instructions with any prior AVX version is supported with no penalties. Figure 4.1 displays the 512-bit registers (ZMM0-ZMM31). AVX registers YMM0–YMM15 map into

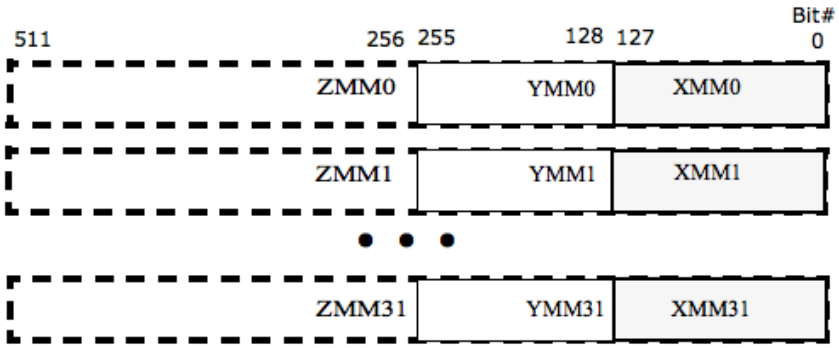

Figure 4.1: AVX512-Bit Wide Vectors and SIMD Register Set

the Intel AVX-512 registers ZMM0–ZMM15, which is similar to how SSE registers maps 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.

4.2.2 Arm-v8 Scalable Vector Extension

Arm introduced Scalable Vector Extension (SVE) [8] on the latest Arm-v8 architecture. As show in figure 4.2, SVE introduced 16 predicate (P) registers and 32 data (Z) registers with the long vector extension; the new architecture supports vector length from 128 bits up to 2048 bits. It provides a range of different values that permit vector code to adapt automatically to the current vector length at runtime with the feature of Vector Length Agnostic (VLA) programming. Similar to AVX, SVE also has a family of horizontal reduction instructions which include integer and floating-point summation, minimum, maximum, and bit-wise logical reductions.

4.2.3 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, as each vendor defines its own set of intrinsic functions, with either full support on some compilers or compiler-agnostic header files. Access to these intrinsics empowers programmers by providing direct access to low-level instructions and enabling algorithm design and implementation where the compiler will perform the complex task of register allocation and instruction scheduling wherever possible. Using intrinsics 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 the compilers provide considerable help. The major drawback of intrinsics is their limited portability. Each set of intrisics are only portable among a specific architecture (x86 and AArch64) of processors. In summary, the intrinsic functions provide the capability for SIMD instructions to be manipulated faster, more proficiently and more effectively. The following AVX-512 and SVE intrinsic functions are relevant to this work:

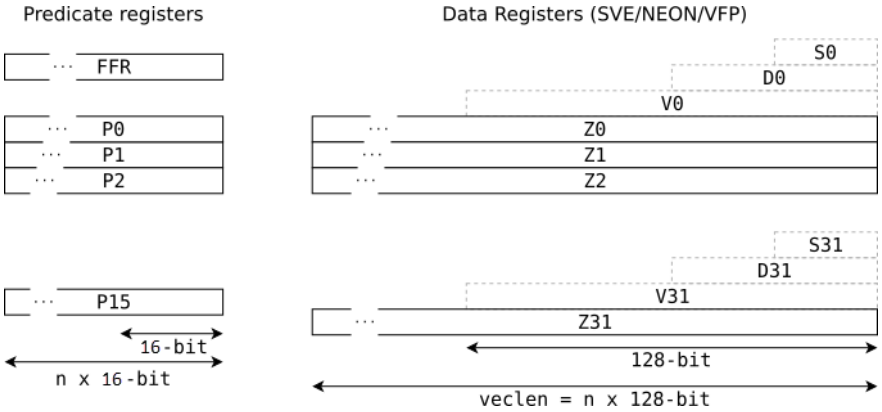

Figure 4.2: Arm SVE Registers

__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 instrinsic 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 destination, 32-bits of integer data is used as an example here. Generally, this instrinsic 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 instrinsic is converted into:
vmovdqu32 m512, zmm.

svint32_t vsrc = svld1(Pg, void const* mem_addr) Load data from memory into an SVE long vector with predicate registers.

svint32_t vsrc = svst1(Pg, void const* mem_addr, svint32_t a) Store data from "a" into memory. Data length adapts 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 the SVE reduction intrinsic between the packed 32-bit integers in "a" and "b".

4.2.4 Reduction Operations in Open MPI

Advanced reduction operation with AVX, AVX2, AVX-512 support is implemented in a component in Open MPI, based on a Modular Component Architecture [38, 78] that facilitates extending or substituting Open MPI core subsystems with new features and innovations. Long vector reduction optimization is added in a specialized component that implements all predefined MPI reduction operations with vector reduction instructions, as in Figure 4.3. From a practical standpoint, the module that extracts the processor feature flags and checks related capabilities, selecting at runtime the best set of functions supporting the

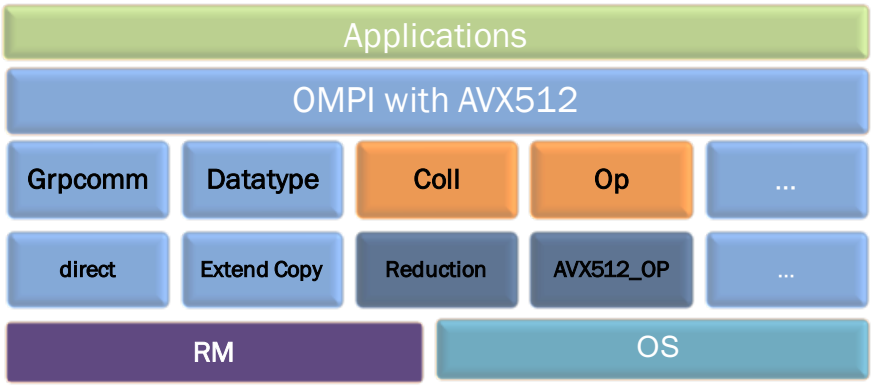

Figure 4.3: Open MPI architecture. The orange boxes represent components with added AVX-512 reduction features. The dark blue colored boxes are new modules.

most advanced ISA (AVX-512, AVX2 or AVX/SSE), or reverts to the default basic module if the processor has no support for such extensions, as shown in Figure 4.4.

To be more specific, the code explicitly checks CPUID – an instruction allowing software to discover the processor details, determine processor type and list the supported features such as SSE/AVXs.

To be noted, the computational benefits of my component and modules can be extended depending on the scope of reduction operation or general mathematics and logic operations. This advanced operation module/code-snippet can be easily adapted to other computational-intensive software stacks.

Vector instructions can be integrated in applications in several manners: (a) automatic vectorization support provided by the compiler; (b) explicitly calling vector instructions from assembly or via intrinsic functions; or (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 that are 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 transformation phases, which prevents an efficient identification of SIMD parallelism in real applications in many cases [55]. The second method allows more control over the very low-level instruction stream; however, the use of intrinsics is time-consuming and error-prone for application programmers and users. This work adopts the third approach to integrate the use of AVX-512 features in the Open MPI stack. Intrinsics and compile flags in the programming model (Open MPI) provide long vector support for applications to use.

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, reduction operations have large amounts of data-level parallelism

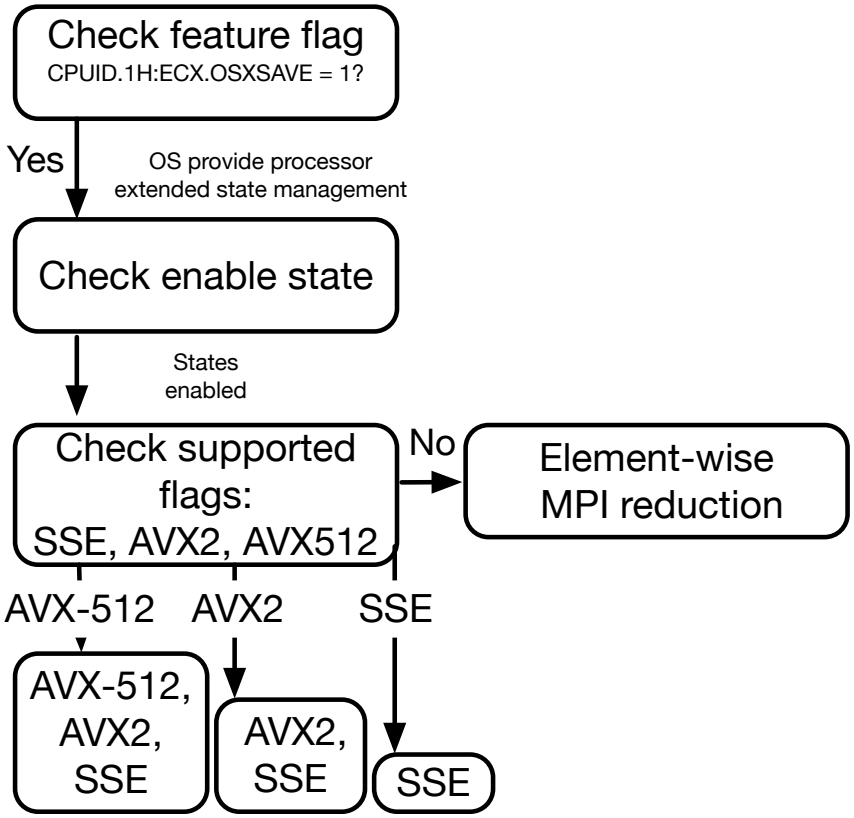

Figure 4.4: Integrate and automatically activate the AVX component into the Open MPI build system

and should benefit from SIMD support. A reduction operation performs element by element on the input buffers, which is traditionally translated into code that executes as a sequential operation, but could possibly be vectorized under particular circumstances or with a specific compiler or specific constraints. Sometimes it may suffer from dependencies across multiple loop iterations.

Figure 4.5 illustrates the difference between a scalar operation and a vector operation with AVXs and SVE instructions of different vector lengths. In this example, a vector instruction processes multiple elements simultaneously, as 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 can 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 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 lengths up to 2048 bits, allowing more extensive reduction operations with more elements in a long vector.

4.2.5 Implementation with AVXs

Intel AVX-512 intrinsic provides arithmetic reduction operations for integer and float-point and also supports logical and bit-wide reduction operations on integer types. This gives the chance to create AVX-512 intrinsic-based reduction support in MPI, which will highly increase the performance of MPI local reduction. Additionally, AVX-512 can perform scatter reduction operations with the support of a predicate register, which behaves in a vectorized manner. This could lift the restriction of a contiguous memory layout for reduction operations and allow for non-contiguous data sets, but such operations are not needed for the predefined MPI reduction operations.

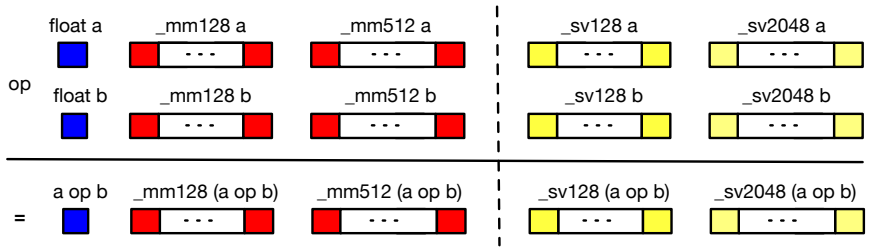

Figure 4.5: Example of single precision floating-point values using : (■) scalar standard C code, (■) AVXs 128 bits $\sim$ 512 bits SIMD vector of 4,8,16 values; (■) SVE 128 bits $\sim$ 2048 bits SIMD vector of different values

Algorithm 2 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 / inout_buf

```
1: procedure ReductionOp( in_buf, inout_buf, count )
2:     types_per_step = vector_length(512) / (8 × sizeof_type)
3:     #pragma unroll
4:     for k ← types_per_step to count do
5:         _mm512_loadu_si512 from in_buf
6:         _mm512_loadu_si512 from inout_buf
7:         _mm512_reduction_op
8:         _mm512_storeu_si512 to inout_buf
9:         Update left_over
10:    if ( left_over ≠ 0 ) then
11:        Update types_per_step >>= 1
12:        if ( types_per_step ≤ left_over) then
13:            _mm256_loadu_si256 from in_buf
14:            _mm256_loadu_si256 from inout_buf
15:            _mm256_reduction_op
16:            _mm256_storeu_si256 to inout_buf
17:            Update left_over
18:    if ( left_over ≠ 0 ) then
19:        Update types_per_step >>= 1
20:        if ( types_per_step ≤ left_over) then
21:            _mm_lddqu_si128 from in_buf
22:            _mm_lddqu_si128 from inout_buf
23:            _mm128_reduction_op
24:            _mm_storeu_si128 to inout_buf
25:            Update left_over
26:    if (left_over ≠ 0 ) then
27:        while ( left_over ≠ 0 ) do
28:            Set case_value
29:            Switch(case_value) : {8 Cases}
30:            Update left_over
```

The optimized reduction operation employs and applies multiple methods to investigate how to achieve the best performance on different processors, as shown in algorithm2. In the algorithm's for-loop section, it explicitly uses 512 bits long vector to load and store for memory operations rather than using the memory copy (memcpy) function provided by the standard library, because some compilers may not perform the best assembling techniques when using ZMM registers for load and store. Once the elements are loaded in registers, the corresponding vector operation is used to perform the reduction on the entire vector register. The algorithm repeats this pattern with a full 512 bits until the remainders cannot fulfill a 512 bits vector. Then, it falls back 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 a few operation on the remaining few elements.

It is 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 remaining 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, Duff's device [76] is used to manually implement 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 4.1 shows the variety of data types and abbreviations for MPI reduction operation handle names that are supported in the optimized reduction operation module, which matches the combination of types and operations defined by the MPI standard. Table4.2 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

Table 4.1: Supported types and operations

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

Table 4.2: Supported CPU flags

Instruction Sets	CPU flags and op_avx_support value	
AVXs	AVX512BW (0x200)	AVX512F (0x100)
	AVX2 (0x020)	AVX (0x010)
SSE	SSE4 (0x08)	SSE3 (0x04)
	SSE2 (0x02)	SSE (0x01)

are supported by the hardware. To be noted, this 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.

4.2.6 Implementation with SVE

The SVE instruction based reduction is implemented with Arm C language extension (ACLE) using intrinsic functions. As shown in algorithm3. ACLE uses a variable vector length, which can be accessed at runtime by calling the function ***svcntb()*** | ***svcnth()*** | ***svcntw()*** | ***svcntd()*** to determine the number of 8, 16, 32 and 64-bit elements in the vector. As previously mentioned, Open MPI uses a modular architecture, and I added another reduction module in the operation framework enabled only on Arm architectures with SVE support. The code is compiled using Arm HPC compile 20.0, enabling SVE extensions using the flag ***-march=armv8-a+sve***. As with AVX reduction, the SVE implementation also supports different data types and abbreviations for MPI reduction operations, as defined by the MPI standard.

4.3 MPI Reduction Benchmark Evaluation

4.3.1 Intel Xeon Architecture

Experiments are conducted on a local cluster, which is an Intel(R) Xeon(R) Gold 6254 (AVX512F) based server running at 3.10 GHz. This work is based upon Open MPI master branch, git commit hash #75a539 [4]. Each experiment is repeated 30 times, and the average results are presented here.

Algorithm 3 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 )
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 ← types_per_step to count do
6:         svld1 from in_buf
7:         svld1 from inout_buf
8:         sv#op_sign#size_z
9:         svst1 to inout_buf
10:        Update left_over
11:    if (left_over ≠ 0 ) then
12:        while ( left_over ≠ 0 ) do
13:            Set case_value
14:            Switch(case_value) : {8 Cases}
15:            Update left_over
```

The experiments use a single node with one process for all tests, because the optimization aims to improve the performance of the computation part of reduction operations rather than the communication part.

This section compares the performance of the reduction operation with two implementations. For Open MPI default reduction operation base module, it performs element-wise computation across two input buffers. For each loop iteration, it processes two elements. The new implementation uses AVX-512 vector instructions to execute reduction operations on the same inputs. But for each iteration, it deals with two vectors, which contain all the elements within the vectors, representing a vector-wise operation. The reduction benchmark uses the MPI_Reduce_local function call to perform the local reduction for all supported MPI reduction operations utilizing an array of different sizes.

Predefined MPI operations are compared, the arithmetic SUM and the logical BAND using input buffers with sizes in the rage from 1KB to 128MB. For the experiments, I

minimized the potential impact of preloaded caches by flushing the L1 and L2 cache after each test to ensure the experiments are not reusing data from the cache (especially for buffers size below the cache size).

Figure 4.6 and Figure 4.7 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 flier data that extend beyond the whiskers. It should be noted that the default compiler on the platform failed to generate auto-vectorized code, despite my best efforts (i.e. providing all the documented optimization flags). The optimization uses intrinsics which give the code complete control of the low-level details at the expense of productivity and portability.

Results demonstrate that, when using AVX-512 enabled operation, performance can be improved seven times faster compared with the default, element-wise operation. As expected, the improvement is dependent on the number of elements in the reduction buffer, with a small number of elements producing a small improvement that increases once the buffer size becomes larger than 4KB, where the performance improvement becomes considerable. For the sake of completeness, the MPI operations are compared with the memory copy (memcpy) operation, under the assumption that the vendor provided implementation of memcpy is highly optimized for the target architecture, providing an upper bound. To make a fair comparison, I list the complete execution sequence of reduction operations and memory copy operations. 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, the optimized AVX-512 reduction operation achieves a high level of memory bandwidth comparable to memory copy. When the reduction buffer size increases, the implementation achieves similar performance as memory copy, which indicates that this approach is capable to take full advantage of all the available memory bandwidth.

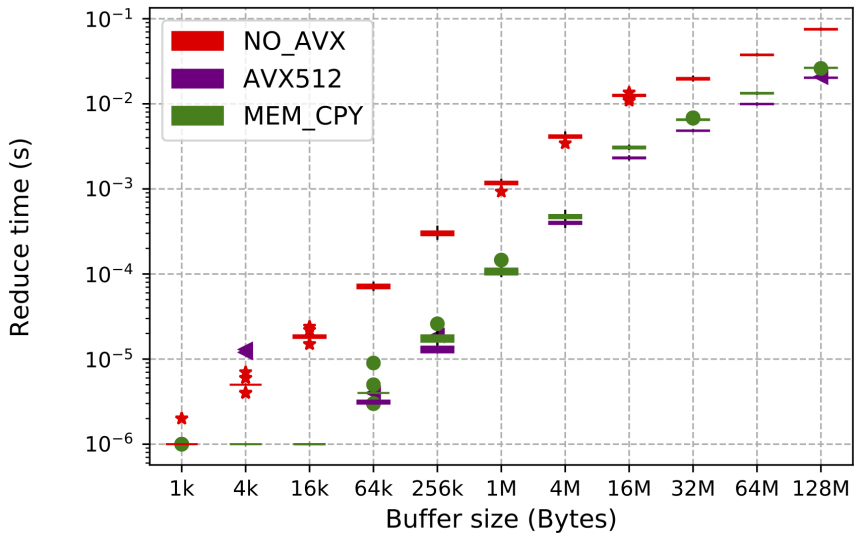

Figure 4.6: Comparison of MPI_SUM with AVX-512 reduction enable and disable for MPI_UINT8_T together with memcpy

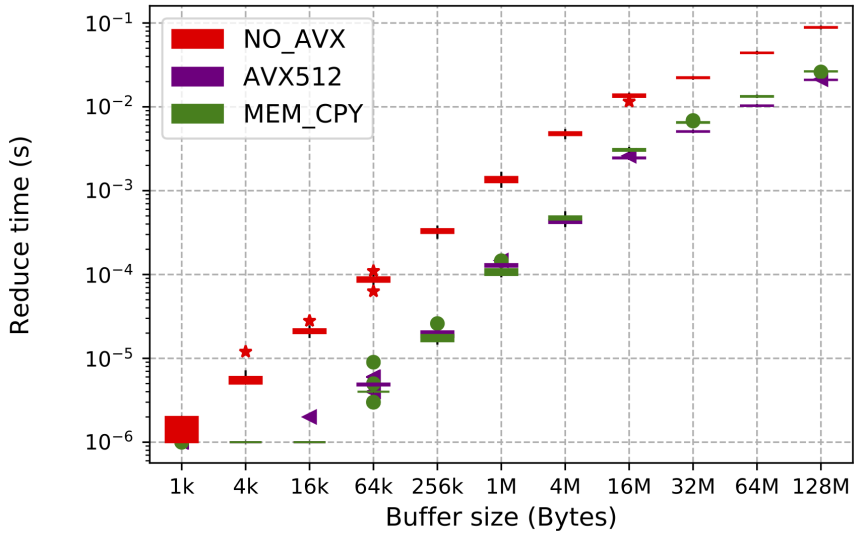

Figure 4.7: Comparison of MPI_BAND with AVX-512 reduction enable and disable for MPI_UINT8_T together with memcpy

4.3.2 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. Thus, 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.

Benchmark experiments are conducted 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.

Figure 4.8 shows the result of the MPI_SUM operation on buffers with various sizes ranging from 1KB to 128MB. 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, the optimized operations achieve almost the same memory bandwidth, which implies that the computation is totally overlapped with memory operations.

4.3.3 Arm-v8 Architecture: A64FX

Performance evaluation experiments of SVE based reduction operation are conducted on the new A64FX processor, which supports vector lengths of 256 bits and 512 bits. Figure 4.9 shows the results of the MPI_SUM operation from the Open MPI default implementation, the SVE optimized implementation and the memory copy operations. Under all tested reduction buffer sizes, the SVE optimized operation is five times faster than element-wise operation, and obtains a similar memory bandwidth compared to the memory copy operation.

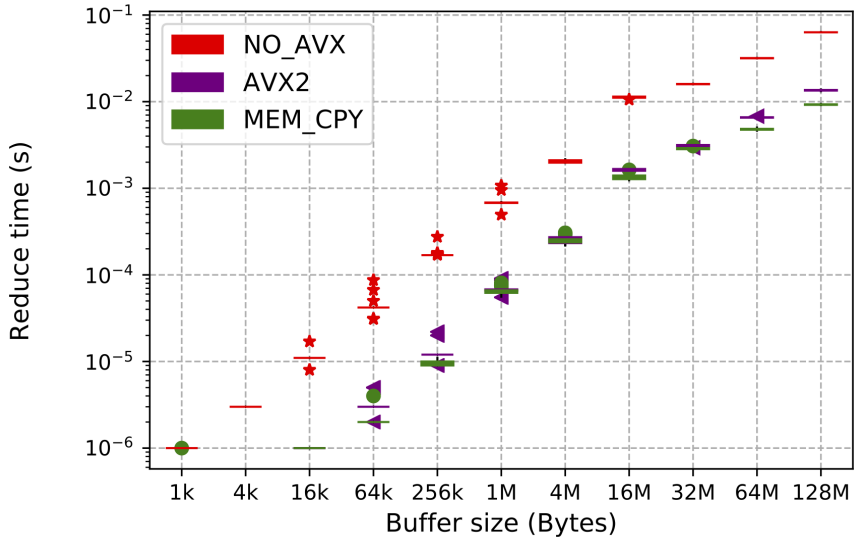

Figure 4.8: AMD EPYC 7302 16-Core Processor: comparison of MPI_BAND for MPI_UINT8_T with and without AVX2, with the memcpy operation

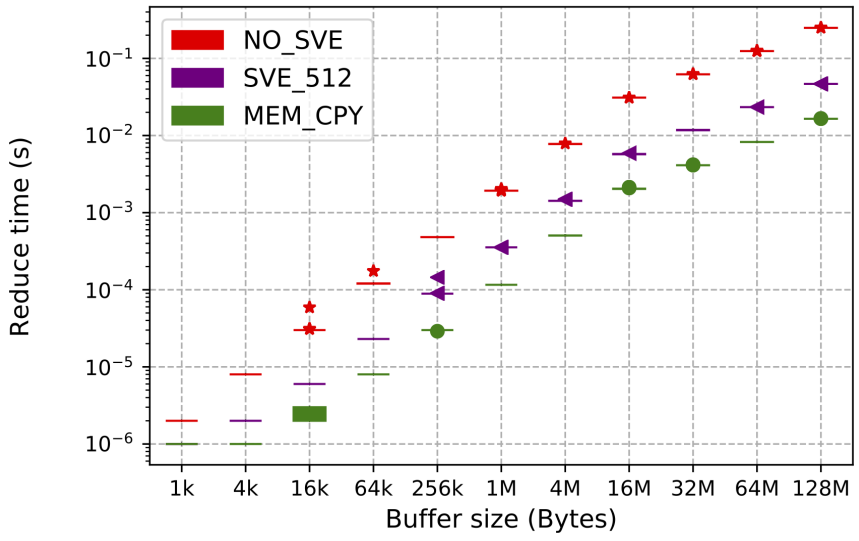

Figure 4.9: Arm A64FX: comparison of MPI_SUM with SVE (Vector Length = 512bits) reduction enable and disable for MPI_UINT8_T together with memcpy

4.4 Performance Tool Evaluation

To evaluate the performance, the AVX-512 enabled Open MPI reduction operation is analyzed using Performance API (PAPI) [72] – 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 the executed code and, indirectly, 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.

Experiments 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: the 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. Figure 4.10 shows the total number of instructions, memory access instructions for load and store and branch instructions. Due to the stability of the results, I choose not to clutter the graphs with additional information, such as the standard deviation. For the 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 to non-vector loads and stores, which means that it needs fewer loads and stores dealing with the same amount of reduction data.

The implementation decreased the number of load and store instructions by a factor of 90X and 60X, respectively. At the same time, for branching instructions, this optimization decreased by 60X. Cache misses of L1 and L2 caches are investigated. Because the operation is dealing with large buffers of contiguous data, this means data access patterns are very

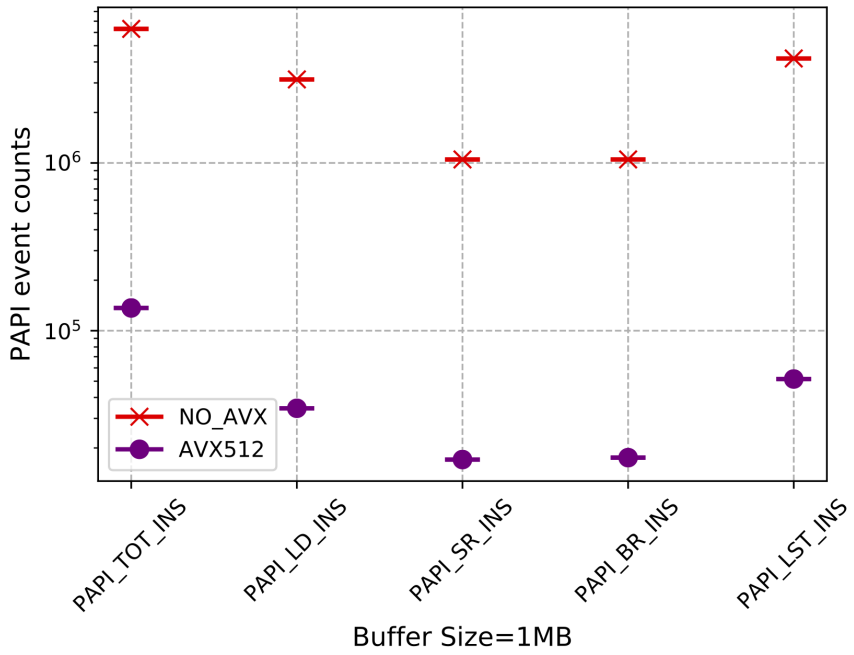

Figure 4.10: Comparison between AVX-512 optimized Open MPI and default Open MPI for MPI_SUM reduction with PAPI instruction events overview

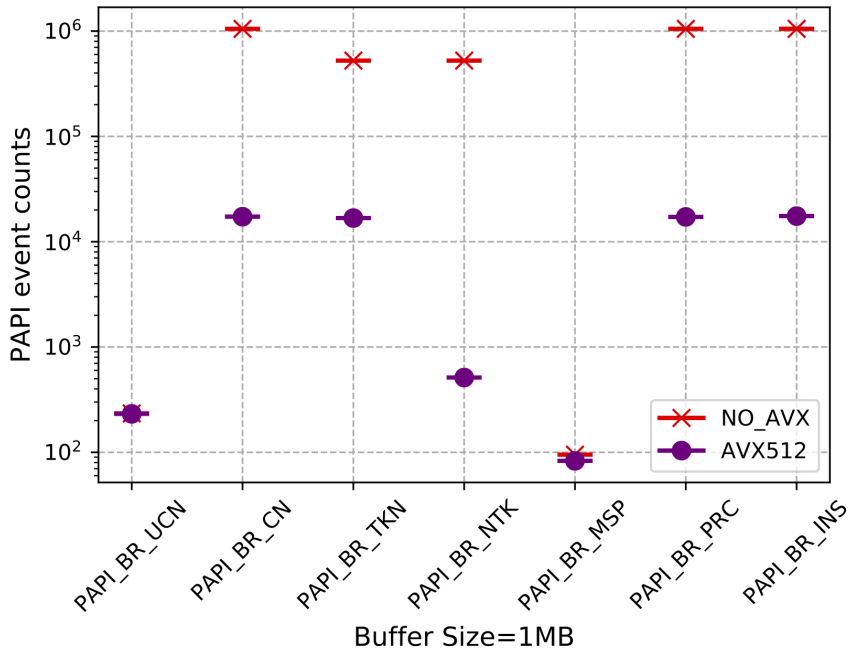

Figure 4.11: Comparison between AVX-512 optimized Open MPI and default Open MPI for MPI_SUM reduction with PAPI branch counters

regular and easy to predict by even a basic prefetch algorithm. All predicted accesses would be consumed, and cache misses would not show significant variation.

Figure 4.11 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, the AVX-512 based reduction largely decreases the "for loop" of the reduction operation. Consequently, the AVX-512 code has much less control and branching instructions; therefore, there is less opportunity to mispredict the branching outcome.

4.5 Application Evaluation

4.5.1 LAMMPS Application Evaluation

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [61] is a molecular dynamics simulation tool from Sandia National Laboratories. It provides different benchmark datasets representing a range of simulation styles and computational expenses for molecular-level interaction forces. In the experimental analysis, the performance of the optimized reduction operation is evaluated with LAMMPS granular flow benchmark using the dataset from chute flow (in.chute.scaled). The benchmark reports the "Loop Time" as a measure of the time required to simulate a set of molecular interactions. Benchmark runs with 24 processes (process grid: 4x2x3) on an Intel(R) Xeon(R) Gold 6254 CPU with different capabilities of AVX support, including single AVX, AVX2 and AVX512. The implementation restricts the vector capabilities used for MPI reduction operations via the modular parameter of $-$***mca op_avx_support***.

Figure 4.12 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). Without AVX support (Open MPI MCA command option "op $\wedge$ AVX") for the reduction operations as shown in red, the latency of the loop is 651.5. With the optimization of using AVX and AVX2, the new design archives an 11% speedup of the total application's executing time. Enabling AVX512 support and provides an additional performance boost,

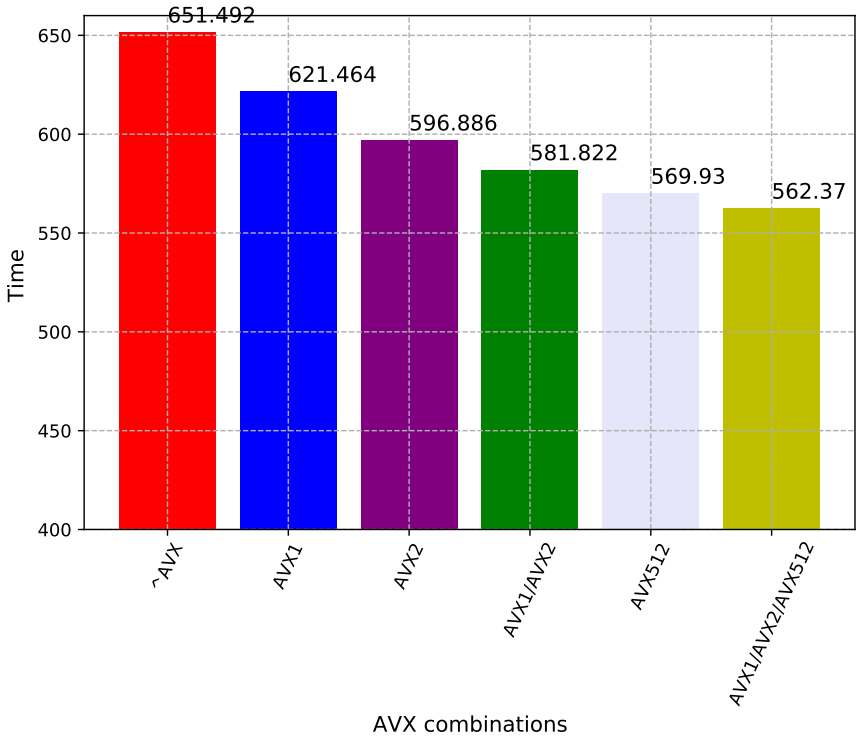

Figure 4.12: LAMMPS chute: loop time on 24 procs for 100 steps with 259200000 atoms with different AVX capabilities

up to a 13.4% speedup. Tuning the switch points between the different vector instructions provides an additional performance boost with a maximum speedup of 14.7%.

4.5.2 Deep Learning Application Evaluation

Over the past few years, advancements in deep learning have driven tremendous improvements to image processing, computer vision, speech recognition, robotics and control and natural language processing. One of the significant challenges of deep learning is decreasing 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. Therefore, 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.

This section investigates and experiments on Horovod [65] - an open-source component of Michelangelo's deep learning toolkit that 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, 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 resulting array to all processes. Horovod uses a ring-allreduce approach, which is a bandwidth optimal [58] algorithm if the tensors are large enough, but it 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-allreduce. The fusion method allocates a large fusion buffer and executes the AllReduce operation on the fusion buffer. In the ring-allreduce algorithm, each of N nodes communicates with two of its peers $2 * (N - 1)$ times. During this communication, a node sends and receives chunks of the data buffer. In the first $N - 1$ iterations, received values are added to the values in the node's buffer. In the second $N - 1$ iterations, 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. During the AllReduce processing phase, there are $P * (N - 1)$ reduction operations that occurred with a big fusion buffer size, which is very computation intensive. The AVX-512 optimized reduction operations can significantly improve the performance of the computation and reduction part of those collective operations.

Experiments are conducted on Stampede2 with Intel Xeon Platinum 8160 nodes. Each node has 48 cores with two sockets and it has 192GB DDR4 memory. Each core has 32KB 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.

Figure 4.13 shows the performance comparison of the 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 [3]. Compared to default element-wise reduction implementations with the increasing number of processes, this design shows increasing improvements, which start at 5.45% and eventually rise to 12.38% faster than the default Open MPI on 192 processes and 1536 processes, respectively. It can be observed that the performance benefit increases with more processes/nodes, because of the strong-scaling nature of the application, which translates to more MPI processes to participate in the reduction operation, the larger the data is. Because each one of them is simultaneously using our AVXs optimized Open MPI operations, overall application performance is improved.

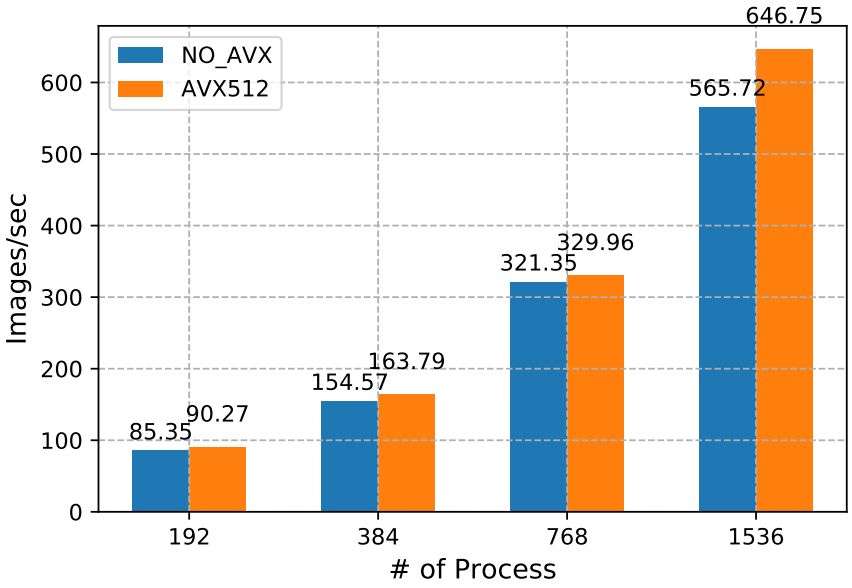

Figure 4.13: tf_cnn_benchmarks results using Horovod (model: alexnet) on stampede2 with AVX-512 optimized Open MPI and default Open MPI

Chapter 5

Pack and Unpack Using Long Vector Gather and Scatter

5.1 Overview

Hardware platforms in high performance computing are constantly getting more complex to handle and satisfy increasing computational needs. This brings new challenges and opportunities to the design of software and libraries, especially with regard to MPI libraries. Numerous features and configuration options from novel architectures and processors with long vector extension become much more important to exploit the potential peak performance. Novel processor architectures such as the Intel AVX-512 architecture introduced 512-bit instructions for x86 ISA. Additionally, Arm introduced SVE with a maximum 2048 bits long vector extension for the AArch64 architecture. These new features allow for better compliance with long vector gather load and scatter store. This work proposes new optimized strategies by utilizing the gather and scatter feature to improve the packing and unpacking operations for non-contiguous memory access. With these optimizations, this work not only provides a higher-parallelism for a single node, but also it achieves a more efficient communication scheme for message exchange. The optimization implementation is proposed in the context of Open MPI, providing efficient and scalable capabilities of gather and scatter usage and extending the possible implementations to a larger range of programming and execution paradigms.

5.2 Design and Implementation in Open MPI

5.2.1 Memory Access Pattern

A memory operation is the most widely used operation during communication, including point-to-point and collectives. Data need to be packed on the sender side before sending and be unpacked on the receiver side. The datatype constructs provided by the MPI standard create the capability to define contiguous and non-contiguous memory layouts, allowing developers to reason at a higher level of abstraction, thinking about data instead of focusing on the memory layout of the data (for the pack/unpack operations). MPI defines data layouts of varying complexity, including contiguous and non-contiguous data layout, as shown in Figure 5.1.

Contiguous type is the simplest derived type: a number of repetitions of the same datatype without gaps in-between, as shown in Figure 5.1(a). C standard library provides function as memory copy to manipulate the contiguous type. With the help of a modern compiler, it is converted to assembly code which is represented as a loop of load and store instructions using vector registers.

For **non-contiguous** datatype layouts, as shown in Figure 5.1, vector type Figure 5.1(b) is the most regular and certainly the most widely used MPI datatype constructor. Vector allows replication of a datatype into locations that consists of equally spaced blocks, describing the data layout using block-length, stride and count. **Block-length** refers to the number of primitive datatypes that a block contains, **stride** refers to the number of primitive datatypes between blocks, and **count** defines the number of blocks that need to be processed. A distinctive flavor of vector datatype, frequently used in computational sciences and machine learning, accesses a single column of matrix as presented in Figure 5.1(d) and can be represented by a specialized vector type with block-length equal one.

Datatypes other than vector expose 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

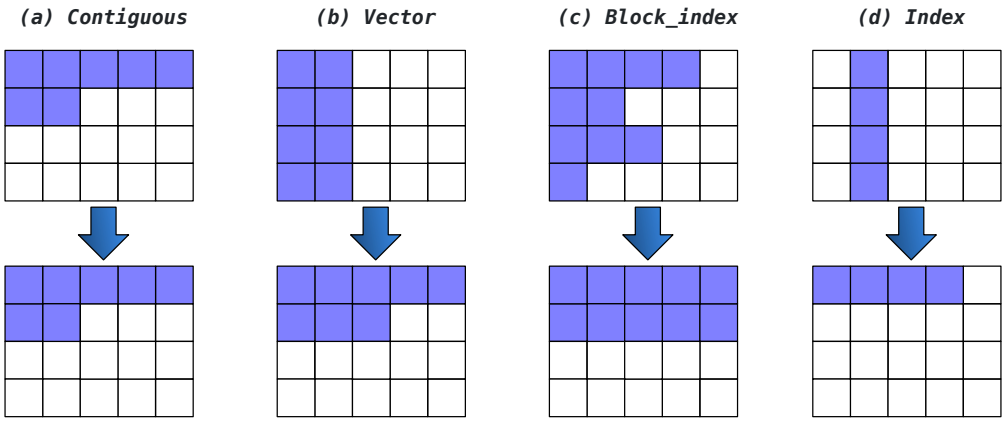

Figure 5.1: Memory layout of datatype (contiguous and non-contiguous) in MPI

datatypes). Such datatypes Figure 5.1(c) cannot be described in a concise format using only block-length and stride.

5.2.2 Pack and Unpack with Gather and Scatter

High-performance parallel algorithms and scientific applications often need to communicate non-contiguous data. Typically, applications need to pack the non-contiguous data into a temporary contiguous buffer and send it to endpoint processes. The receiver performs the unpack operation to distribute the data from the contiguous receiver buffer to the non-contiguous data buffers. However, this approach (known as "Manual Packing/Unpacking") limits performance, because it needs to create multiple copies of the data and increase its memory footprint. Also, application developers need to manage those temporary buffers manually, leading to poor productivity. This packing/unpacking process involves considerable time. A previous study [64] has shown that packing and unpacking data could take 90% of the total communication overhead for non-contiguous sends. To resolve this problem, MPI derived datatype (DDT) provides the convenience of hiding the complexity of sending non-contiguous data from application developers. It is essential for the MPI community to provide efficient MPI datatype communication, which could reduce or remove the packing/unpacking overhead for non-contiguous data.

Figure 5.2 gives an overview of Optimized Open MPI transferring non-contiguous data by using the gather and scatter feature from long vector extension in packing and unpacking, respectively. With the default packing/unpacking scheme, data is first copied into a pack buffer and transferred to the receiver, the receiver then unpacks data into its user buffer. On the other hand, the gather and scatter scheme on the sender side replaces multiple small memory copies by single gather instruction to fetch/load data from different memory addresses. On the receiver side, it uses single scatter instruction to replace multiple small memory copies to distribute/store data into different memory addresses.

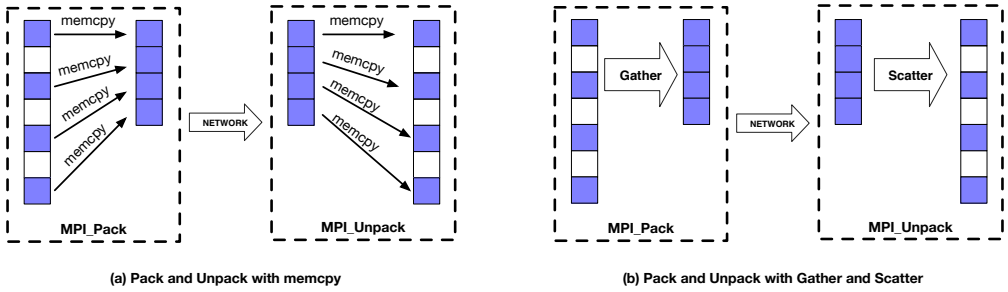

Figure 5.2: Comparison between general memory copy and AVX/SVE gather/scatter implementation for packing and unpacking

AVX-512 Gather and Scatter

The detailed gather load and scatter store instructions for non-contiguous small block data are revealed in Algorithm 4. In this optimized algorithm, the "gather_pack" procedure uses intrinsic *_mm512_type_gather_type (Src, ..., offsets, ...)* to load data from different memory addresses based on offsets to a single long vector, and then store it to destination. To be noted, for each vector type it only needs to generate the offsets once and repeatedly use this for all gather instructions. The "scatter_unpack" procedure first loads the packed contiguous data to a long vector and then uses intrinsic *_mm512_type_scatter_type (Dst, ..., offsets, ...)* to store the data to non-contiguous addresses. For the remaining blocks with total size smaller than the vector length, it uses explicitly masked load and store operations to partially load/store the data from/to memory to maintain the integrity and correctness of the data. This highly expands the limited performance of memory operations for small non-contiguous memory blocks.

SVE Gather and Scatter

SVE introduces new subsets of instruction that provides multiple addressing access modes to enable gather load and scatter store for non-contiguous memory. There are two kinds of addressing that have the same format as a base component with a displacement component: *vector plus immediate* and *scalar plus vector*. The base is the starting 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 this case, it uses *scalar plus vector of offsets* mode, with a specified explanation as

$$svint32\_t\ svld1\_gather\_u32base\_offset\_s32(svbool\_t\ pg,$$

$$svuint32\_t\ bases, int64\_t\ offset)$$

Algorithm 4 Gather/Scatter based pack and unpack algorithm

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

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

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

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). An optimized pack and unpack algorithm is developed 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. Less regular memory may have a repeating pattern of memory layout; thus, if it can generate offset vectors for the repeated pattern, then it can apply multiple gather loads and scatter stores for each repetition and apply the results 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 offset vector or vectors. SVE optimization work is added 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. 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 5.

5.2.3 Benchmark Evaluation

AVX-512 Implementation Evaluation

Experimented on a local cluster, an Intel(R) Xeon(R) Gold 6254 based server running at 3.10 GHz. The CPU consists of 18 physical cores, which support advanced features:

Algorithm 5 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

```
1: procedure MemcpyWithMultipleVectors( DST, SRC, blocklen )
2:     full_vector_copies = blocklen / (svcntb × N)
3:     for k ← 0 to full_vector_copies do
4:         svldN from SRC
5:         svstN to DST
6:     if ( remaining ≠ 0 ) then
7:         Generate svp
8:         Partially ld/st using svp
1: procedure SveBasedPack( Count, blocklen, Extend )
2:     if ( blocklen ⩾ svcntb ) then
3:         for k ← 0 to Count do
4:             MemcpyWithMultipleVectors(blocklen,Src,Dst)
5:     else
6:         Blocks_per_vector = svcntb / blocklen
7:         Generate offset_vector
8:         for k ← 0 to (Count / blocks_per_vector) do
9:             Sve gather_load using offset_vector
10:        Generate svp
11:        Processing remaining blocks
```

Intel Advanced Vector Extensions 512 (AVX-512), new instruction set extensions, delivering ultra-wide (512-bit) vector operations capabilities, with up to 2 FMAs (Fused Multiply Add instructions), to accelerate performance for most demanding computational tasks.

This work is based upon the Open MPI master branch, git commit hash #406bd3a [4]. Each experiment is repeated 30 times; here presents the average. All experiments are conducted on a single node. This section compares the performance of MPI pack and unpack operations with two implementations. The Open MPI default version uses general memory copy method during pack and unpack operation for non-contiguous datatypes. It uses a for loop to copy all the blocks for a non-contiguous datatype.

In the new implementation, on the sender side, it uses the AVX-512 vector gather feature for packing operations; on the receiver side, it uses AVX-512 scatter feature for unpacking. Pack and unpack benchmark uses the official test **to_self** in the Open MPI repository to perform packing and unpacking operations for a vector datatype with different message sizes. Comparing the packing and unpacking performance speedup separately reveals the benefits of gather load and scatter store. Experiments use a vector datatype with primitive datatype MPI_INT constructed with $blocklength = 2, gap = 1, count = 1024$, which means it packs two of three integers per block. Figure 5.3 displays the performance comparison between the Open MPI default packing algorithm and the AVX-512 gather packing algorithm. The X-axis shows the size of the packed buffer; Y-axis shows the actual bandwidth, which means the higher the better. By using gather, the optimized packing strategy achieves $2.3 \sim 3.5$ times speedup. I also compare the algorithms together with memory copy for contiguous data, which indicates the peak memory bandwidth. We can see that even for non-contiguous data, when the message size is increased to 512KB, it achieves 41% of the peak bandwidth. Figure 5.4 presents the performance comparison between the Open MPI default unpacking algorithm and the AVX-512 scatter unpacking algorithm. We can see that by using scatter feature, the optimized unpacking strategy achieves 3.4 times speedup. Comparing to memory copy bandwidth, the scatter method achieves 35%. We can see that unpacking acquires less efficiency than packing when compared to peak bandwidth from memory pack, as reading from non-contiguous addresses is more efficient than writing to non-contiguous addresses. Also, compared to contiguous memory copy, gather and scatter

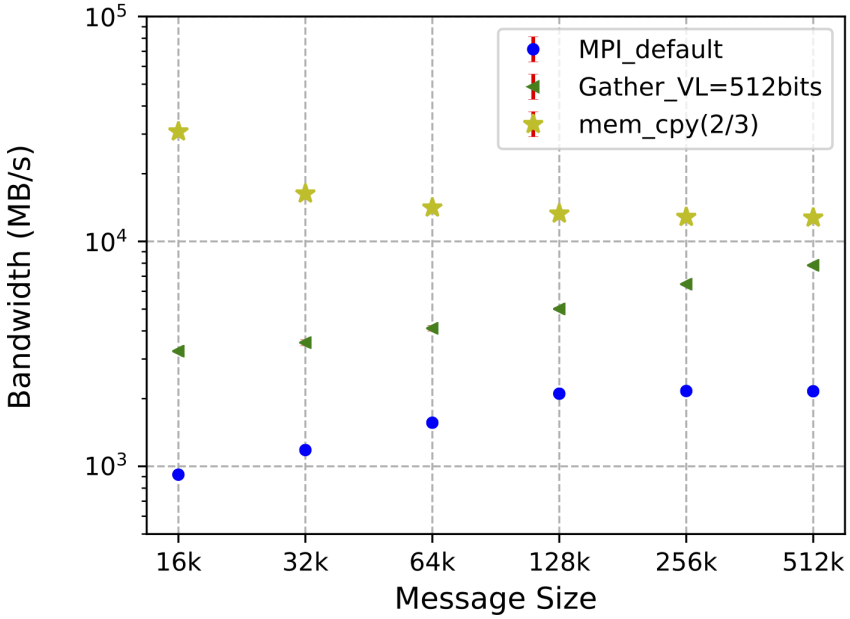

Figure 5.3: Comparison of MPI_Pack with AVX-512 gather enable and disable together with memcpy for vector datatype

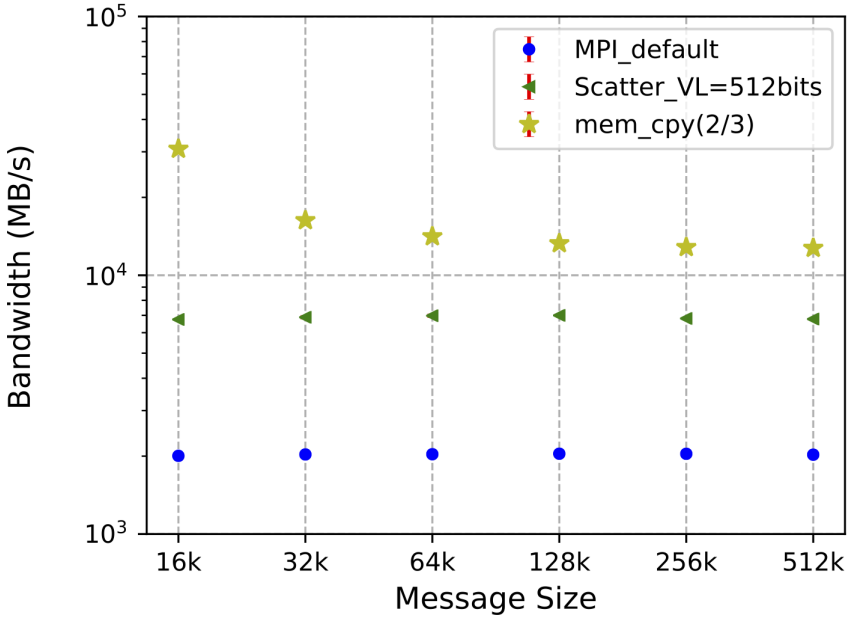

Figure 5.4: Comparison of MPI_Unpack with AVX-512 scatter enable and disable together with memcpy for vector datatype

operations require the hardware to do more work than contiguous SIMD loads and stores and are likely to access more cache lines/pages (depending on the specific access pattern). This will cause higher instruction overheads. To be noted, for the memory copy operation it only copies 2/3 of the total unpacked buffer.

SVE Implementation Evaluation

The implementation is evaluated on a cluster with Fujitsu's Arm SVE based processor A64FX, which is the first processor of the Armv8-A SVE architecture. Each processor 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 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 the return of 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. This 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 5.5 presents the performance of pack and unpack using the SVE gather and scatter feature with different vector length for a non-contiguous buffer. The green and yellow line indicates the performance using vector length 256 bits and 512 bits respectively with the 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× faster which validates the Gem5 simulated results.

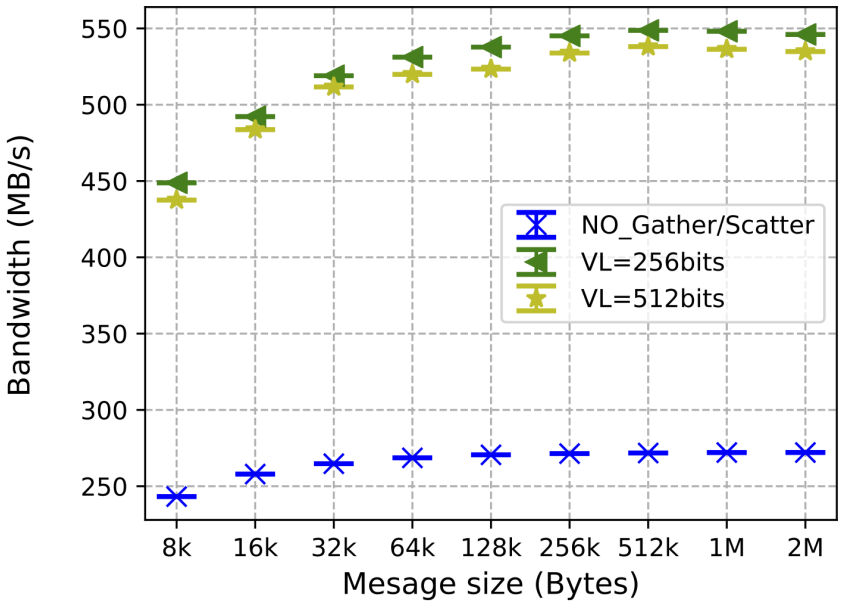

Figure 5.5: Comparison of MPI_Pack/Unpack with SVE gather/scatter enable and disable together with memcpy for vector datatype

5.3 Application Evaluation with AVX-512 Implementation

5.3.1 Domain-decomposed 2D Stencil

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

A 2D five-point stencil is a stencil made up of the point itself together with its four "neighbors". Each point has four neighbors: up, down, left and right; as shown in Figure 5.6, we can see that the global domain is represented by an N*N two-dimension matrix, which gets partitioned into multiple blocks (one per process) of roughly equal size. After each computation step, the boundary regions of these partitions have to be exchanged with its four neighboring processes before the next time-step can be started. For easy explanation, it assumes matrices are stored in "Row Major" order where data exchanged in the north-south direction is a contiguous pattern. In contrast, the data exchanged in the east-west direction is a non-contiguous pattern. There are two ways to handle the communication: the first one is to send and receive the data in multiple small chunks, which can be inefficient due to the constant overhead associated with each send operation; the second method is that the data has to be packed into a consecutive buffer and sent in one piece. On the receiver side, this process has to be reversed (the data has to be unpacked) after such data is received.

We can see that AVX-512 enabled Open MPI can speed up the packing and unpacking procedure for east-west direction communication, which uses gather and scatter to pack and unpack the boundary regions. The east-west boundary can be represented with a vector datatype, as shown in Figure 5.6. This particular vector type is constructed as shown in table 5.1.

This section investigates the performance benefit of AVX-512-enabled Open MPI against default Open MPI. The application benchmark is based on the Stencil MPI implementation

Table 5.1: East-west vector data represent

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

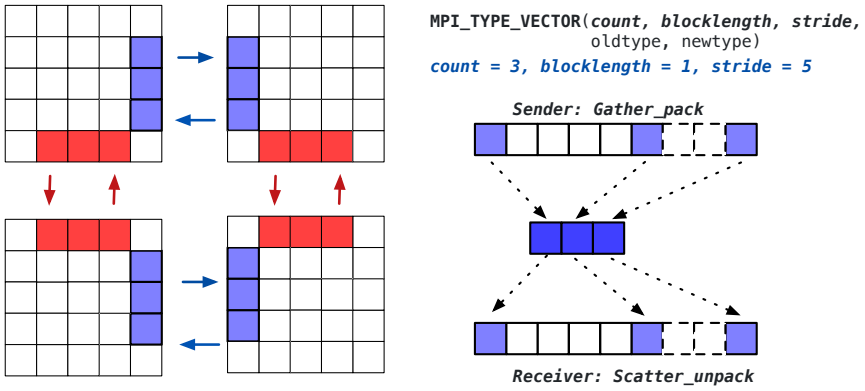

Figure 5.6: Domain-decomposed 2D stencil. Data exchanged in east-west direction must be packed and unpacked in communication

from Parallel Research Kernels (PRK) [2, 73] – a suite of simple kernels for the study of the efficiency of distributed and parallel computer systems, including all software and hardware components that make up the system. They cover a wide range of common parallel application patterns, especially from the area of HPC. This stencil implementation uses a for loop with a memory copy function to pack the chunks for the non-contiguous data from the east-west boundary. I optimized this packing implementation with the vector representation described above.

Experiments are conducted on the same Intel Xeon Gold6254 based cluster with 16 processes, processes are arrange as 2*8 in x/y direction. Table 5.2 shows the experiment configuration for this stencil application. The stencil is a five point stencil using single precision execution for 100 iterations. The experiments demonstrate the effectiveness of AVX-512 enabled Open MPI with three tests, each using a different radius. As demonstrated in Figure 5.7, experiments compared the performance of two implementations. The gather and scatter optimized implementation decreases the packing and unpacking cost for communication during each computation step. Consequently, it improves collective operation that drives up the overall application performance by 10% for all three cases.

5.3.2 2D Fast Fourier Transform

The Fast Fourier Transform (FFT) is one of the most significant algorithms for exascale applications across various disciplines in science and engineering. Applications range from image analysis and signal processing to solving partial differential equations through spectral methods. Also, there are diverse parallel libraries that rely on efficient FFT computations, particularly in particle applications ranging from molecular dynamics computations to N-Body simulations. Thus, for all these applications, it is essential to have access to a fast and scalable implementation of a FFT algorithm and an implementation that can take advantage of efficient communication libraries and components and maximize these benefits for applications.

A FFT on multidimensional data can be performed as a sequence of one-dimensional transforms along each dimension. For example, a two-dimensional FFT can be computed by performing 1D-FFTs along both dimensions. With multiple MPI processes, after each

Table 5.2: MPI stencil configuration and execution on 2D grid

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

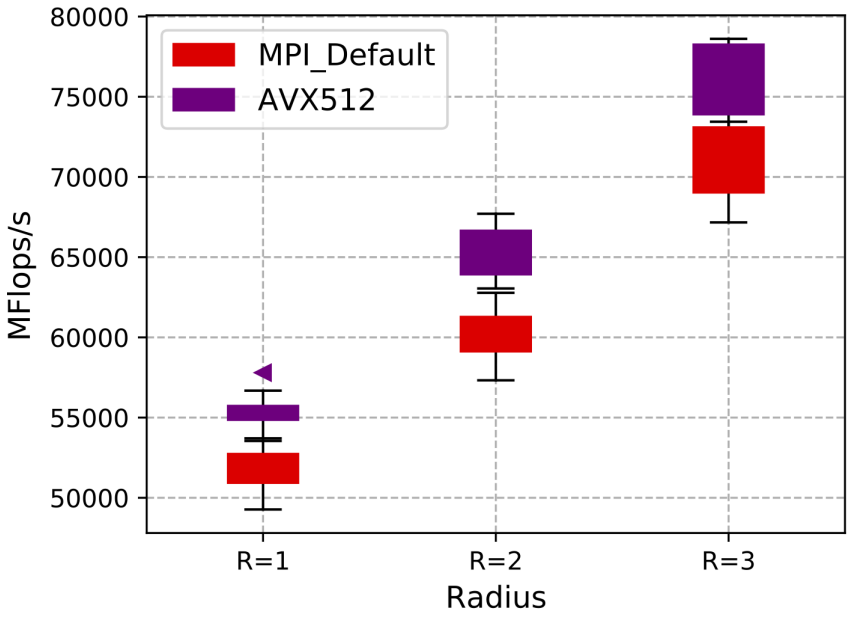

Figure 5.7: 2-d Stencil results with and without AVX-512 gather pack and scatter unpack for different radius

process computes the 1D-FFT, a matrix transpose needs to be performed among MPI processes using MPI_ALLtoall operation. Also ND-FFTs can be computed by performing 1D-FFTs in all N dimensions. This subsection examines the performance of the gather pack and scatter unpack approach, it measures the performance (running time) of a micro-benchmark: 2D-FFT Benchmark with code version [45]. More details about the implementation can be found in this paper [46]. In this implementation, a vector type is used for all to all communication. The results compare the performance of this all to all collective between MPI_default and the proposed optimized design.

Figure 5.8 shows the performance comparison of the 2D-FFT Benchmark completion time between the AVX-512 enabled pack and unpack operation and the default operation in Open MPI. The X-axis shows the number of processes. The number of elements in each dimension is 8000. Optimized implementation achieves 8% performance speedup under all three cases for the entire completion time.

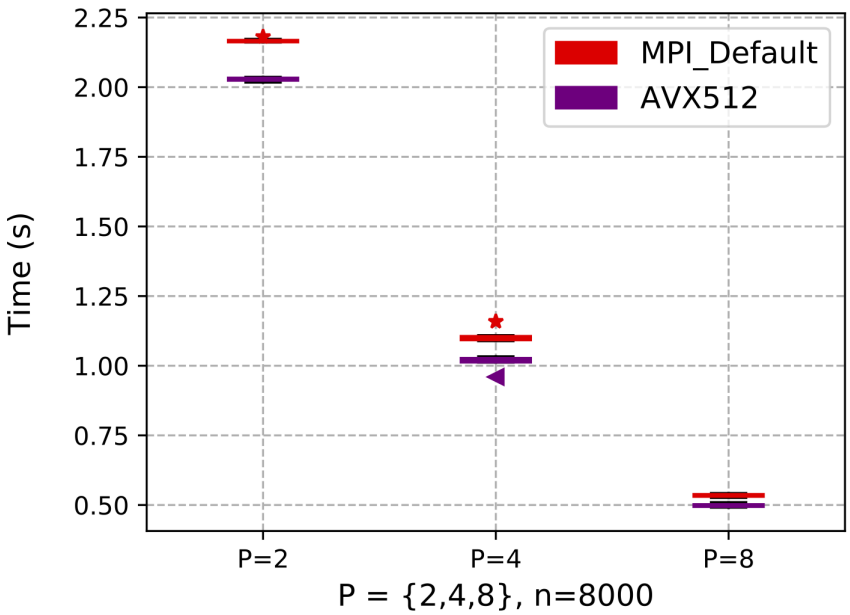

Figure 5.8: 2-d FFT results with and without AVX-512 gather pack and scatter unpack for different number of processes

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The scientific computing community's increasing computational need demands more powerful HPC systems, and the increasing scale and complexity of these HPC systems brings new challenges to the design of parallel tools and libraries.

As the scale of those systems grows, the mean-time-to-failure is negatively impacted and diminished, which presents issues designing failure detection and propagation strategies to ensure the correct completion of long computing jobs. Thus, the first challenge is to provide resilience and reliability. It is critical to integrate a resilient, efficient and portable fault detector and propagator into one of the most widely used parallel execution runtimes, allowing other libraries and programming models to build on and support resilience at any scale. Resilience provides the foundation to run long computing jobs on such systems, which prompts the investigation of the potential performance benefits on those systems.

The second challenge is to provide high performance software uses for complex and novel hardware architectures from different vendors on modern HPC systems. For instance, Intel released Advanced Vector Extension with Haswell processor that supports the 256-bit AVX2 instructions. Knights Landing processor extends this feature to more advanced 512-bit wide SIMD registers. Arm promoted its new Arm-v8 architecture with a Scalable Vector Extension equipped with vector length from 128 bits to 2048 bits.

To achieve reliability and high performance for libraries and software, first, a multi-level failure detection algorithm is designed, which operates within the runtime infrastructure to monitor both node and process failures. Then, the algorithm is implemented as a component in the PRRTE, which is a fully fledged runtime that is used in production to deploy, monitor and serve multiple HPC networking stack clients. I then compare this generic failure detection service with the fully dedicated MPI detector from ULFM Open MPI on one hand, and with the Scalable Weakly-Consistent Infection-style Membership (SWIM) protocol on the other hand, the latter stands as a state-of-the-art detector for unstructured peer-to-peer systems. There is a performance trade-off in generality, but a satisfactory level of performance can be achieved in a portable and reusable component that can satisfy the needs of a variety of HPC networking systems.

Second, I pragmatically demonstrated the benefits of Intel AVX, AVX2, AVX-512 and Arm SVE vector operations in the context of MPI reduction operations. It assesses the performance advantages of different features introduced by AVX and extended the investigation and analysis to a fully-fledged implementation of all predefined MPI reduction operations. To further validate the performance improvements, experiments are conducted with different applications: (1) LAMMPS benchmark with variety AVXs support shows a speedup from 14% to 34% with different AVX capability combinations (2) Experiments with a deep learning application, using distributed model Horovod, calculates and updates the gradient to adjust the weights using an MPI_Allreduce. The new reduction strategy achieved a significant speedup across all ranges of processes with a 12.38% improvement with 1536 processes.

Last, I present the benefits of using the gather and scatter feature from long vector extension. This work evaluates the performance advantages of the gather and scatter feature to load and store non-contiguous data. A new packing and unpacking strategy is introduced in the datatype engine under OPAL level in Open MPI using intrinsics to speed up the communication for a non-contiguous datatype. MPI_to_self benchmark results demonstrate the efficiency of the new pack and unpack algorithm. Both AVX-512 and SVE based implementations achieve considerable performance speedup with vector datatype (blocklen = 2, gap =1). To further validate the performance improvements, experiments are conducted

with two applications: five-point stencil and 2D-FFT. The proposed design outperforms default Open MPI by 10% and 8%, respectively.

6.2 Future Work

To further improve and explore my research with resilient and performance advantages from long vector extension. I am considering the following two possible directions.

- The resilience research designs and implements failure detection and propagation strategy in runtime systems. However, the current detection and propagation algorithm treats all processes and nodes as participants. Features that can indicate the scope of detection and propagation can be added. With this new feature, it can support partial detection, which means only a subgroup of nodes are involved. This feature will decrease the overhead cost of detection and provide resilience as needed.

- Another aspect of my research is extending the long vector usage to more components and modules in Open MPI and, further, out the scope of MPI to other libraries. It is essential to utilize those new features in different programming models and applications.

Bibliography

[1] (2019a). *Intel MPI Benchmarks User Guide*. https://software.intel.com/en-us/imb-user-guide. 25

[2] (2019). *Parallel Research Tools*. https://github.com/ParRes/Kernels. 86

[3] (2019b). *TensorFlow benchmarks*. https://github.com/tensorflow/benchmarks. 69

[4] (2021). *Open MPI main development repository*. https://github.com/open-mpi/ompi. 58, 80

[5] Ang, J., Barrett, B. W., Wheeler, K. B., and Murphy, R. C. (2010). Introducing the graph 500. 39

[6] Angskun, T., Bosilca, G., and Dongarra, J. (2007). Binomial graph: A scalable and fault-tolerant logical network topology. In Stojmenovic, I., Thulasiram, R. K., Yang, L. T., Jia, W., Guo, M., and de Mello, R. F., editors, *Parallel and Distributed Processing and Applications*, pages 471–482, Berlin, Heidelberg. Springer Berlin Heidelberg. 12, 19

[7] ARM (2018). *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*. 5

[8] ARM (2020). *Porting and Optimizing HPC Applications for Arm SVE Version 2.1*. 47

[9] Armejach, A., Caminal, H., Cebrian, J. M., González-Alberquilla, R., Adeniyi-Jones, C., Valero, M., Casas, M., and Moretó, M. (2018). Stencil codes on a vector length agnostic architecture. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 13:1–13:12, New York, NY, USA. ACM. 6

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

[11] Aulwes, R. T., Daniel, D. J., Desai, N. N., Graham, R. L., Risinger, L. D., Taylor, M. A., Woodall, T. S., and Sukalski, M. W. (2004). Architecture of la-mpi, a network-fault-tolerant mpi. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 15–. 10

[12] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S. (2011). FTI: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA. ACM. 4

[13] Blackford, L. S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997). *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 6

[14] Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., and Dongarra, J. J. (2013). An evaluation of user-level failure mitigation support in MPI. *Computing*, 95(12):1171–1184. 2, 3

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

[16] Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., and Dongarra, J. (2016). Failure detection and propagation in hpc systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 312–322. 2, 12

[17] Bouteiller, A., Bosilca, G., and Venkata, M. G. (2016). Surviving errors with openshmem. In Gorentla Venkata, M., Imam, N., Pophale, S., and Mintz, T. M., editors, *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*, pages 66–81, Cham. Springer International Publishing. 2

[18] Bramas, B. (2017). A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *International Journal of Advanced Computer Science and Applications*, 8(10). 13

[19] Callahan, D., Dongarra, J., and Levine, D. (1988). Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*,

Supercomputing '88, pages 98–105, Washington, DC, USA. IEEE Computer Society Press. 5

[20] Caminal, H., Caballero, D., Cebrián, J. M., Ferrer, R., Casas, M., Moretó, M., Martorell, X., and Valero, M. (2018). Performance and energy effects on task-based parallelized applications. *The Journal of Supercomputing*, 74(6):2627–2637. 4

[21] Cao, C., Herault, T., Bosilca, G., and Dongarra, J. (2015). Design for a soft error resilient dynamic task-based runtime. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 765–774. 2

[22] Castain, R. H. (2017a). *RFC0002:PMIx Event Notification*. 23

[23] Castain, R. H. (2017b). *RFC0015:Job Control And Monitoring APIs*. 23

[24] Castain, R. H., Hursey, J., Bouteiller, A., and Solt, D. (2018). Pmix: Process management for exascale environments. *Parallel Computing*, 79:9 – 29. 10, 17, 23

[25] Chakraborty, S., Laguna, I., Emani, M., Mohror, K., Panda, D. K., Schulz, M., and Subramoni, H. (2018). Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 0(0):e4863. 2, 3

[26] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267. 11, 16

[27] Chen, W., Toueg, S., and Aguilera, M. K. (2002). On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32. 11

[28] Chu, C., Hamidouche, K., Venkatesh, A., Awan, A. A., and Panda, D. K. (2016). 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)*, pages 726–735. 14

[29] Das, A., Gupta, I., and Motivala, A. (2002). Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312. 11, 12, 28

[30] D'Azevedo, E. F. and Imam, N. (2015). Graph 500 in OpenSHMEM. In Gorentla Venkata, M., Shamis, P., Imam, N., and Lopez, M. G., editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies*, pages 154–163, Cham. Springer International Publishing. 39

[31] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA. ACM. 12

[32] Di Martino, C., Kalbarczyk, Z., and Iyer, R. (2016). Measuring the Resiliency of Extreme-Scale Computing Environments. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 609–655. Springer. 2

[33] Dosanjh, M. G. F., Schonbein, W., Grant, R. E., Bridges, P. G., Gazimirsaeed, S. M., and Afsahi, A. (2019). Fuzzy Matching: Hardware Accelerated MPI Communication Middleware. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 210–220. 13

[34] Espasa, R., Valero, M., and Smith, J. E. (1998). Vector architectures: past, present and future. In *Proceedings of the 12th international conference on Supercomputing*, pages 425–432. 4

[35] Fagg, G. E. and Dongarra, J. J. (2000). Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In Dongarra, J., Kacsuk, P., and Podhorszki, N., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, Berlin, Heidelberg. Springer Berlin Heidelberg. 10

[36] Flur, S., Gray, K. E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., and Sewell, P. (2016). 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, pages 608–621, New York, NY, USA. ACM. 5

[37] Forum, M. P. I. (November 15,2020). *MPI: A Message-Passing Interface Standard Version 4.0 (draft)*. 9

[38] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary. 49

[39] Gainaru, A., Graham, R. L., Polyakov, A., and Shainer, G. (2016). Using InfiniBand Hardware Gather-Scatter Capabilities to Optimize MPI All-to-All. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 167–179, New York, NY, USA. ACM. 13

[40] Graham, R. L., Woodall, T. S., and Squyres, J. M. (2006). Open mpi: A flexible high performance mpi. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg. Springer Berlin Heidelberg. 10

[41] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. (2001). On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 170–179, New York, NY, USA. ACM. 11

[42] Hammarlund, P., Martinez, A. J., Bajwa, A. A., Hill, D. L., Hallnor, E., Jiang, H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., Osborne, R. B., Rajwar, R., Singhal, R., D'Sa, R., Chappell, R., Kaushik, S., Chennupaty, S., Jourdan, S., Gunther, S., Piazza, T., and Burton, T. (2014). Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20. 5

[43] Hamouda, S. S., Herta, B., Milthorpe, J., Grove, D., and Tardieu, O. (2016). Resilient x10 over mpi user level failure mitigation. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, pages 18–23, New York, NY, USA. ACM. 2

[44] Hao, P., Pophale, S., Shamis, P., Curtis, T., and Chapman, B. (2015). Check-pointing approach for fault tolerance in openshmem. In *Revised Selected Papers of the Second Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies - Volume 9397*, OpenSHMEM 2015, pages 36–52, New York, NY, USA. Springer-Verlag New York, Inc. 2, 3

[45] Hoefler, T. (2012). *MPI Derived Datatype (Benchmark) Page: 2D FFT Benchmark*. http://unixer.de/research/datatypes/. 88

[46] Hoefler, T. and Gottlieb, S. (2010). Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using mpi datatypes. EuroMPI'10, page 132–141, Berlin, Heidelberg. Springer-Verlag. 88

[47] Hofmann, M. and Rünger, G. (2008). MPI Reduction Operations for Sparse Floating-point Data. In Lastovetsky, A., Kechadi, T., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 94–101, Berlin, Heidelberg. Springer Berlin Heidelberg. 14

[48] Iliescu, D. A. (2018). *Arm Scalable Vector Extension and application to Machine Learning*. 13

[49] Kawazoe Aguilera, M., Chen, W., and Toueg, S. (1997). Heartbeat: A timeout-free failure detector for quiescent reliable communication. In Mavronicolas, M. and Tsigas, P., editors, *Distributed Algorithms*, pages 126–140, Berlin, Heidelberg. Springer Berlin Heidelberg. 11

[50] Kim, R., Choi, J., and Lee, M. (2019). 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, pages 101–110, New York, NY, USA. Association for Computing Machinery. 13

[51] Larrea, M., Fernandez, A., and Arevalo, S. (2000). Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 52–59. 11

[52] Levine, D., Callahan, D., and Dongarra, J. (1991). A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17(10):1223 – 1244. Benchmarking of high performance supercomputers. 5

[53] Lim, R., Lee, Y., Kim, R., and Choi, J. (2018). An implementation of matrix–matrix multiplication on the intel knl processor with avx-512. *Cluster Computing*, 21(4):1785–1795. 13

[54] Luo, X., Wu, W., Bosilca, G., Pei, Y., Cao, Q., Patinyasakdikul, T., Zhong, D., and Dongarra, J. (2020). Han: a hierarchical autotuned collective communication framework. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 23–34. 14

[55] Maleki, S., Gao, Y., Garzar'n, M. J., Wong, T., and Padua, D. A. (2011). An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382. 51

[56] Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., and Zhou, J. (2013). 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*, pages 1107–1116. 5

[57] Molka, D., Hackenberg, D., Schöne, R., Minartz, T., and Nagel, W. E. (2012). Flexible workload generation for hpc cluster efficiency benchmarking. *Computer Science - Research and Development*, 27(4):235–243. 4

[58] Patarasuk, P. and Yuan, X. (2009). Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124. 14, 68

[59] Pentkovski, V., Raman, S. K., and Keshava, J. (2000). Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(04):47–57. 5

[60] Petrogalli, F. (2018). *A sneak peek into SVE and VLA programming*. 13

[61] Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1 – 19. 66

[62] Ranganathan, S., George, A. D., Todd, R. W., and Chidester, M. C. (2001). Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4(3):197–209. 11

[63] Röhl, T., Eitzinger, J., Hager, G., and Wellein, G. (2016). Validation of hardware events for successful performance pattern identification in high performance computing. In Knüpfer, A., Hilbrich, T., Niethammer, C., Gracia, J., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2015*, pages 17–28, Cham. Springer International Publishing. 4

[64] Schneider, T., Gerstenberger, R., and Hoefler, T. (2012). Micro-Applications for Communication Data Access Patterns and MPI Datatypes. In *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490, pages 121–131. Springer. 74

[65] Sergeev, A. and Balso, M. D. (2018). Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*. 68

[66] Shamis, P., Graham, R., Venkata, M. G., and Ladd, J. (2011). Design and implementation of broadcast algorithms for extreme-scale systems. In *2011 IEEE International Conference on Cluster Computing*, pages 74–83. 12

[67] Shan, H., Williams, S., and Johnson, C. W. (2018). 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)*, pages 1–11. 14

[68] Sodani, A., Gramunt, R., Corbal, J., Kim, H., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y. (2016). Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46. 5

[69] Squyres, J. M. and Lumsdaine, A. (2003). A component architecture for lam/mpi. In Dongarra, J., Laforenza, D., and Orlando, S., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 379–387, Berlin, Heidelberg. Springer Berlin Heidelberg. 10

[70] Subasi, O., Martsinkevich, T., Zyulkyarov, F., Unsal, O., Labarta, J., and Cappello, F. (2018). Unified fault-tolerance framework for hybrid task-parallel message-passing applications. *The International Journal of High Performance Computing Applications*, 32(5):641–657. 2

[71] Sun, Q., Romanus, M., Jin, T., Yu, H., Bremer, P.-T., Petruzza, S., Klasky, S., and Parashar, M. (2016). In-staging data placement for asynchronous coupling of task-based scientific workflows. In *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ESPM2, pages 2–9, Piscataway, NJ, USA. IEEE Press. 4

[72] Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In Müller, M. S., Resch, M. M., Schulz, A., and Nagel, W. E., editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg. Springer Berlin Heidelberg. 64

[73] Van der Wijngaart, R. F. and Mattson, T. G. (2014). The parallel research kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. 86

[74] van Renesse, R., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. In Davies, N., Jochen, S., and Raymond, K., editors, *Middleware'98*, pages 55–70, London. Springer London. 11

[75] Watson, W. J. (1972). The ti asc: a highly modular and flexible super computer architecture. In *AFIPS '72 (Fall, part I)*. 4

[76] Wikipedia contributors (2020). Duff's device — Wikipedia, the free encyclopedia. [Online; accessed 2-May-2020]. 56

[77] Wu, W., Bosilca, G., vandeVaart, R., Jeaugey, S., and Dongarra, J. (2016). GPU-Aware Non-contiguous Data Movement In Open MPI. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 231–242, New York, NY, USA. ACM. 14

[78] Zhong, D., Bouteiller, A., Luo, X., and Bosilca, G. (2019). Runtime level failure detection and propagation in hpc systems. In *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, New York, NY, USA. Association for Computing Machinery. 49

Vita

Dong Zhong was born in Zizhou, Shaanxi, China, on March 21, 1991. He received his Bachelor's degree in computer science from Tongji University (2012) and Master's degree in information science and electronic engineering from Zhejiang University (2015) in China.

After finishing his Master's degree he was enrolled in the Ph.D. program in Computer Science at the University of Tennessee, Knoxville. During his studies, he worked as a graduate research assistant at the Innovative Computing Laboratory (ICL) under the supervision of Dr. Jack Dongarra and Dr. George Bosilca. His research interests involve distributed computing, parallel programming paradigms, including Open MPI, PMIx and PRRTE, failure detection and notification and long vector extension analysis and usage of Arm SVE and Intel AVXs.

Dong Zhong is expected to receive his Doctor of Philosophy degree in Computer Science in August 2021.