Parallel Computing

General Failure Detection and Propagation in HPC and Distributed Systems

--Manuscript Draft--

Manuscript Number:	PARCO-D-19-00079
Article Type:	VSI: EuroMPI 2019
Keywords:	Fault Tolerance; Failure Detection; Reliable Broadcast; High performance computing; Runtime Systems; Distributed Systems
Corresponding Author:	dong zhong University of Tennessee Knoxville, TN UNITED STATES
First Author:	Dong Zhong
Order of Authors:	Dong Zhong
	Aurelien Bouteiller
	Xi Luo
	George Bosilca
Abstract:	As the scale of high-performance computing (HPC) systems continues to grow, mean-time-to-failure (MTTF) of these HPC systems is negatively impacted and tends to decrease. In order to efficiently run long computing jobs on these systems, handling system failures becomes a prime challenge. We present here the design and implementation of an efficient runtime-level failure detection and propagation strategy targeting large-scale, dynamic systems that is able to detect both node and process failures. Multiple overlapping topologies are used to optimize the detection and propagation, minimizing the incurred overheads and guaranteeing the scalability of the entire framework. The resulting framework has been implemented in the context of a system-level runtime for parallel environments, PMIx Reference RunTime Environment (PRRTE), providing efficient and scalable capabilities of fault management to a large range of programming and execution paradigms. The experimental evaluation of the resulting software stack on different machines and programming models demonstrate that the solution is at the same time generic and efficient.
Suggested Reviewers:	Anthony Skjellum Professor and Director, SimCenter, University of Tennessee at Chattanooga Tony-Skjellum@utc.edu

Paper title

General Failure Detection and Propagation in HPC and Distributed Systems

Statement

We present the design and implementation of an efficient runtime-level failure detection and propagation strategy targeting large-scale, dynamic systems that can detect both node and process failures. Multiple overlapping topologies are used to optimize the detection and propagation, minimizing the incurred overheads and guaranteeing the scalability of the entire framework.

This paper is an extended version of the conference proceedings [1] that considers not only the case of synthetic communication benchmarks, but also application benchmarks. We use the heavily communication-bound benchmark Graph500 which 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. We conducted our experiments at a larger scale. At the same time, we expanded the study to consider the case of applications using parallel global address space (PGAS) programming models, and study and compare the overhead incurred by enabling failure detection in both MPI and OpenSHMEM applications, using RDaemon$^{\#}$ infrastructure as a shared backend.

Section 6, and figures 16, 17, 18, 19, 20 are additions that present the new application results with both OpenSHMEM and MPI on the Cori supercomputer. We have updated the abstract, introduction, conclusion and title of the paper to reflect the supplementary content.

[1] D. Zhong, A. Bouteiller, X. Luo, G. Bosilca, Runtime level failure detection and propagation in hpc systems, in: Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI '19, ACM, New York, NY, USA, 2019, pp. 14:1–14:11. doi:10.1145/3343211.3343225.

- Design and implementation of an efficient runtime-level failure detection and propagation strategy for HPC and distributed systems.
- Detection of both node and process failures depends on heartbeats and timeouts.
- Reliable broadcast with a low-degree topology that scales with the number of nodes rather than the number of processes.
- Support multiple hybrid applications and programming models. (e.g. MPI and OpenSHMEM)

General Failure Detection and Propagation in HPC and Distributed Systems

Dong Zhong[a], Aurelien Bouteiller[a], Xi Luo[a], George Bosilca[a,*]

[a]*The University of Tennessee, 1122 Volunteer Blvd, knoxville, TN 37996*

Abstract

As the scale of high-performance computing (HPC) systems continues to grow, mean-time-to-failure (MTTF) of these HPC systems is negatively impacted and tends to decrease. In order to efficiently run long computing jobs on these systems, handling system failures becomes a prime challenge. We present here the design and implementation of an efficient runtime-level failure detection and propagation strategy targeting large-scale, dynamic systems that is able to detect both node and process failures. Multiple overlapping topologies are used to optimize the detection and propagation, minimizing the incurred overheads and guaranteeing the scalability of the entire framework. The resulting framework has been implemented in the context of a system-level runtime for parallel environments, PMIx Reference RunTime Environment (PRRTE), providing efficient and scalable capabilities of fault management to a large range of programming and execution paradigms. The experimental evaluation of the resulting software stack on different machines and programming models demonstrate that the solution is at the same time generic and efficient.

Keywords: Fault Tolerance, Failure Detection, Reliable Broadcast, High Performance Computing, Runtime Systems, Distributed Systems

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, either to perform the experiments, collect the observation, 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, more accurate problems, which entails larger compute workloads, fueling an unending need for 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 continue the growth in compute capabilities. This has stressed the traditional HPC software infrastructure in different ways, but it notably put to prominence two different issues that had been largely disregarded in the last two decades: fault tolerance and novel programming models.

The Message Passing Interface (MPI) has been instrumental in permitting the efficient programming of massively parallel systems, scaling along from early systems with tens of processors to current systems routinely encompassing hundreds of thousands of cores. As failures become more common on large and complex systems [1], 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 [2], or various efforts to integrate tightly checkpoint-restart with MPI [3]. The second source of stress comes from programming systems that are inherently hierarchical. This has brought forth a renaissance in the field of programming models, leading to a variety of contenders challenging the hegemony of MPI as the sole method of harnessing the power of parallel systems. Naturally, these alternatives to MPI also have to handle fault tolerance [4, 5, 6, 7, 8]. In addition, the convergence between big data infrastructure and the HPC infrastructure, as well as the emergence of machine learning as a massive consumer of compute capabilities, is gathering around HPC systems new communities that have long-held expectations that the infrastructure provide resilience as a core feature [9].

A feature that's commonly needed by these communities with a vested interest in fault tolerance is the capability to efficiently, quickly and accurately detect and report failures, so that they can manifest as error codes from the programming interface, or trigger implicit recovery actions. In prior works [10], we have designed a tailor-made failure detector for MPI that deploys finely tuned optimizations to improve its performance. These optimizations are unfortunately 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 major infrastructure rewrite for each and every one of them. In this paper, we test the hypothesis that a fully dedicated MPI solution is not necessary to achieve great accuracy and performance, and that a generic failure detection solu-

*Corresponding author

Email addresses: dzhong@vols.utk.edu (Dong Zhong), bouteill@icl.utk.edu (Aurelien Bouteiller), xluo12@vols.utk.edu (Xi Luo), bosilca@icl.utk.edu (George Bosilca)

tion, provided by an external runtime entity that does not have access to the MPI context and communication conduits can deliver a similar level of service. In order to test that hypothesis, and further, to define how a generic a solution can be, we designed a multi-level failure detection algorithm, which we refer to as RDaemon# in this paper, which operates within the runtime infrastructure to monitor both node and process failures. We implemented that algorithm as a component in the PMIx [11] runtime reference implementation (PRRTE), which is a fully fledged runtime that is used in production to deploy, monitor and serve multiple HPC networking stack clients. We 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 standing as a state-of-the-art detector for unstructured peer-to-peer systems. This paper is an extended version of the conference proceedings [12] that considers not only the case of synthetic communication benchmarks, but also application benchmarks, like the Graph500, at a larger scale. It also studies the overhead incurred in both MPI and OpenSHMEM applications, using RDaemon# infrastructure as a shared backend. Henceforth we highlight that 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.

The rest of this paper is organized as follows. Section 2 motivates our study and provides use cases and background on the MPI specific failure detector implementation in ULFM. Section 3 presents related work on failure detectors followed by Section 4 where we describe the algorithm and implementation details of our generic failure detector. Section 5 describes the performance and accuracy comparison between three different failure detectors providing a distinct trade-off on the general to specific scale.

2. Motivation and Background

Many projects have proposed fault management techniques, either automatic, driven by the application, or 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 solutions, insufficiently maintained where no portable and efficient support to build resilient applications or programming models exists. This lack of portable reliable software infrastructure also makes comparing fairly existing or proposed solutions difficult, not necessary in terms of potential capabilities but in terms of performance. We 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 allows other libraries and programming models to build on and support resilience at any scale. Here are some examples of usages of such a resilient framework that we are actively pursuing.

ULFM repairs the MPI infrastructure after a failure [2]. 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 due to being executed in the context of the MPI process. Using RDaemon# alleviates these issues by cleanly splitting the MPI rank ordering, progress engine, and thread initialization modes from the operation of the failure detector. We will discuss in the experimental section how the generality of RDaemon# does not incur a large overhead compared to the specialized ULFM detector.

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 [6]. RDaemon# failure detection and propagation attributes can provide the notification to trigger the recovery. For more exploratory works, application developers can experiment with modulating the frequency and placement of restart point 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 [3]. This work is a co-design between MVAPICH and Slurm resource manager to add process and node failure detection and propagation features. It exhibit interesting detection capabilities, but unfortunately it use an inefficient propagation method and is tied to a single resource manager (Slurm). RDaemon# can substitute a portable fault detection capability to enable EREINIT to run on machines with different resource managers (Slurm, PBS, LSF, TORQUE, etc) and a more efficient propagation 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 [13]. 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 which supports the complex interaction and coordination patterns required by coupled data-intensive application workflows [14]. It can asynchronously capture and index data which allows for dynamic interactions and in-memory data exchanges between coupled applications. For both these software, RDaemon# can provide the basic service to detect and report failures of the distributed infrastructure storage service, which, thus far, has not been fault tolerant.

3. Related Work

In this section, we survey related work on large-scale distributed runtime environments, different kinds of heartbeat based and random gossip based failure detectors, together with reliable broadcast algorithms to propagate fault information.

3.1. Runtime Environments

A wide range of approaches to the problem of exascale distributed computing runtime environments has been studied,

each primarily emphasizing a particular key aspect of the overall problem.

MPICH provides several runtime environments, such as MPD [15], Hydra [16] and Gforker [16]. MPD connects nodes through a ring topology but it is not resilient; two node failures could separate nodes into two separates groups that prevent communication with one another. Another drawback of MPD is that this approach has proved to be non-scalable [17]. Hydra scales well for large numbers of processes on a single node and interacts efficiently with hybrid programming models that combine MPI and threads. While Hydra can monitor and report MPI process failures, it does not cope with daemon failures. Open RTE [18, 19] is the Open MPI runtime environment to launch, monitor, and kill parallel jobs, as well as managing I/O forwarding. It also connects daemons through various topologies, however the communication is not reliable. In general, these runtimes have limited applicability outside of the related MPI implementation that has motivated their creation.

The PRRTE runtime serves as the demonstrator and reference implementation for the PMIx specification [11]. 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, outgrowing 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 is itself not resilient, allowing at best process fault detection and propagation. 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 themselves.

3.2. Failure Detection

Research in the areas of failure detection has been extensively studied. Chandra and Toueg [20] proposed the first unreliable failure detector oracle that could solve consensus and atomic broadcast problems for unreliable distributed systems. Many implementations [21, 22, 23] 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 low hundreds of nodes. An optimized version, the gossip-style protocol [24, 25, 26, 27], in which nodes pick at random 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 dynamically and rapidly varies. 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, leading to 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, we proposed a deterministic failure detector for HPC systems based on network overlays [10], where each participant only observes a single peer following a recoverable ring topology. The experimentation results demonstrate the efficiency of the algorithm; however, the implementation in ULFM being done at the application level can only detect MPI process failures. The implementation employs multiple optimization 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. For example, limitations on the accuracy of the detector when the MPI implementation is not actively communicating are circumvented by using passive target Remote Memory Access primitives (RMA) which are initially provided for supporting the MPI communication; the operational mode, overhead, and accuracy of the detector are impacted by the thread model used during the MPI initialization (i.e., MPI_THREAD_SINGLE results in lower overhead but a higher chance of false positive than MPI_THREAD_MULTIPLE); and, in manycore systems, every MPI process is observed and reported as an independent entity, which can impart that the overhead scales with the number of MPI processes rather than the number of compute nodes; last, the detection topology is tied to the MPI_COMM_WORLD handle which limits the type of topologies that can be employed. 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 application only.

3.3. Reliable Broadcast

Gossip-style [28, 27] 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 exponentially 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 generate 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 [29, 30] 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 guarantee the number of hops in the dissemination path will remain scalable even when some processes on the delivery path have failed. In this work we 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.

4. A Generic HPC Failure Detection Service

In this section, we describe the design of a generic failure detector (called RDaemon# in the remainder of this paper) that we have implemented and delivered as an infrastructure service in the context of PPRTE. 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.

4.1. Machine Model

We 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 a unique identifier (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 [20]). The identity and number of daemons and processes participating in the application is known a priori, or is established through explicit operations that do not require group membership discovery.

4.2. Failure Model

We strive to report crash failures; that is, 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—for example, an illegal instruction is performed by a process because of a processor overheating, an entire node or cabinet loses power, or a software bug manifests by interrupting unexpectedly or rendering some processes permanently non-responsive. In the context of this work, we further distinguish between two subtypes of crash failures. First, application process failures[1], which may impact any number of hosted application processes without necessarily being concomitant with the failure of other processes, even hosted on the same node. Second, node failures, which we consider 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. Our work detects both types of failures. We will discuss in the following sections how this distinction helps improve the scalability of the failure detection algorithm.

4.3. Notations

Table 1 summarizes some of the notations we will employ 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 we propose employs heartbeats between daemons and timeouts to detect node failures.

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

Table 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

4.4. Detection of Process Failures

As illustrated in Figure 1, the failure detector we propose employs two distinct strategies to detect process failures on one hand and node failures on the other hand.

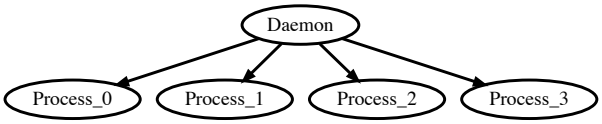

Figure 1: Hierarchical notification of hosted processes through PMIx notification routines. The PRRTE daemon is in charge of observing, and forward notifications to the node-local managed application processes. The detection and reliable broadcast topology operates at the node level between daemons.

To detect process failures that are not congruent with a node failure, we leverage 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 [11] to capture non-terminating crash failures that may arise from software defects, like live-locks, deadlocks and infinite loops.

4.5. 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.

We arrange all N daemons to a logistic ring topology, as illustrated in Figure 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 $m_1, m_2, \ldots$ at time $\tau_1, \tau_2, \ldots$ to its observer $\boldsymbol{o}$. 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 ***d***.

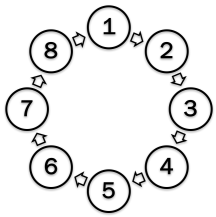

Figure 2: Daemons monitor one another along a ring topology to detect node failures.

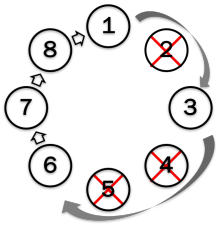

Figure 3: The algorithm mends the detection ring topology when a node failure occurs by requesting heartbeats from the closest live ancestor in the ring.

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. Daemon ***o*** tries to observe the predecessor of ***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, ***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 we get 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.

4.6. 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, we use the scalable and fault-tolerant BMG topology [29]. BMG has good fault-tolerant properties such as optimal connectivity, low fault-diameter, strongly resilient 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 consists of an inner BMG overlay between daemons, and an outer star overlay from each daemon to its local managed processes.

Figure 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.

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

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**
3: Add Eid to msg
4: **if** Eid is a daemon **then**
5: Obtain $Hosted\{Eid\}$
6: add $Hosted\{Eid\}$ to msg
7: ReliableBroadcast(i, N, msg)
8: 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 ($(N + i + 2^k)$ **mod** N)
4: **i** sends **msg** to ($(N + i - 2^k)$ **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 $Reported_j$

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 that are part of a binomial spanning tree rooted at the emitter are sent first. Figure 5 shows the a 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 $O(log_2N)$ steps (given that less than $2log_2N$ 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 thus 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, $O(Nlog_2N)$ 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 can significantly reduce the effective cost of the algorithm when compared to a flat topology between application processes.

4.7. Implementation

4.7.1. PMIx *Interface*

We implemented RDaemon# as a set of components in PRRTE. PRRTE is a fork of the Open MPI runtime, Open RTE [18]. 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 [11]—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 and thus 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 interoperate with Slurm through PMIx can be ported over PRRTE without effort.

In RDaemon#, we leverage the interfaces specified by PMIx [31] to interoperate with the client application, communication library, or programming language, as well as with the SMS. To the best of our 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 [32]: we use it to perform the local propagation of failure information from the daemon to the client processes.

4.7.2. 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 our 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 6, within this architecture, each of the major subsystems is defined as an MCA framework, with a well-defined interface, and multiple components implementing that framework can coexist.

We added two new frameworks and four components 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; to be noted, 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, and last, the `error_ppg` component prepares the content of the reliable 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, as 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.

5. Experimental Evaluation

5.1. 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 internode network, 32 cores per node. Our RDaemon# is based upon PRRTE (#71ef547), with external PMIx (#21d7c9). We compare with ULFM revision #77f9157, which is based on the same base version of Open MPI we use to evaluate RDaemon#

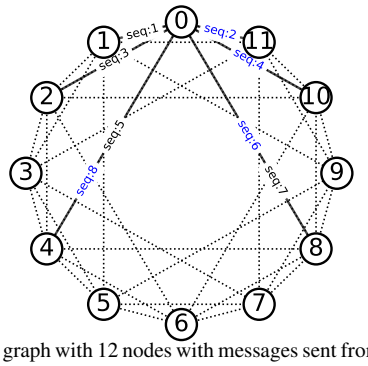

Figure 4: Binomial graph with 12 nodes with messages sent from 0 highlighted.

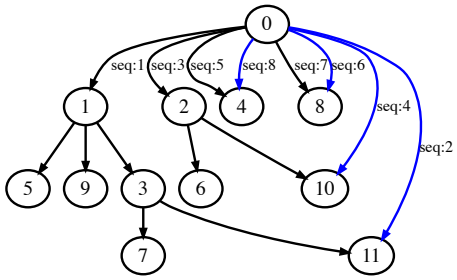

Figure 5: Binomial spanning tree in broadcast from node 0, redundant messages from 0 are colored in blue.

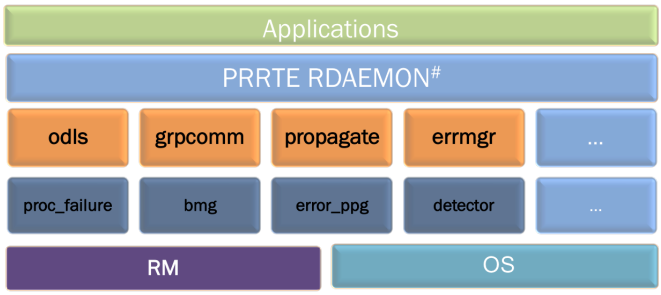

Figure 6: Resilient PRRTE architecture. The orange boxes represent components with added resilience features. The dark blue colored boxes are new modules.

in MPI workloads. Each experiment is repeated 30 times and we present the average. We use Intel MPI Benchmark (IMB v2019.2) [33] for MPI performance measurements for point-to-point (P2P) and collective communications (one MPI rank per core). For all experiments we 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 we use the map-by node, bind-by node policy to set communicating MPI ranks on different nodes.

5.2. Accuracy

For the first experiment, we explore the accuracy of RDaemon#'s detector. The accuracy experiment is conducted by (1) Starting with a large value for the detection timeout η; (2) Verify that no failure is detected when there is no injection, and all injected failures are reported; (3) If the previous test is accurate, decrease η (and accordingly the heartbeat period δ) until we notice false positive detection. We set a constant ratio $\eta = \delta * 2$. 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 8 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, we measured that the heartbeat message is neither delayed by communication congestion nor compute pressure, but we found out that daemons need some time to launch the processes when starting the job which causes heartbeat delay and false detection during job startup.

5.3. Noise

We also investigate the noise overhead incurred on an MPI application by the heartbeat emission and management from RDaemon#. Figure 7 illustrates the overhead incurred with P2P and collective communications running IMB. In order to contextualize the incurred overhead, we present, in shaded grey, the band of natural variability of the benchmark without a failure detector active ($average \pm \sigma$), and, for clarity, we plot error bars for $\delta = 1ms$, the only case where the variability sometime exceeds the natural variability of the benchmark. For the PingPong benchmark, we use the `-multi` mode of IMB with one rank per core on 2 nodes. This ensures that all cores are active with the communication pattern and thus compete for resources with RDaemon# activities. For the collective benchmarks, we run on 64 nodes using all cores. For each message size, we set 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:

$$Overhead = \frac{(\text{RDaemon}^{\#} - \text{PRRTE})}{\text{PRRTE}} \quad (1)$$

From the graph we can see that the latency performance and bandwidth performance are barely affected with 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, slightly higher than the standard deviation of the benchmark itself, at four percent. 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.

5.4. Comparison with SWIM

This section compares the failure detection latency and scalability of RDaemon# with SWIM [27]—a random-probing based failure detection protocol and gossip membership updates. To

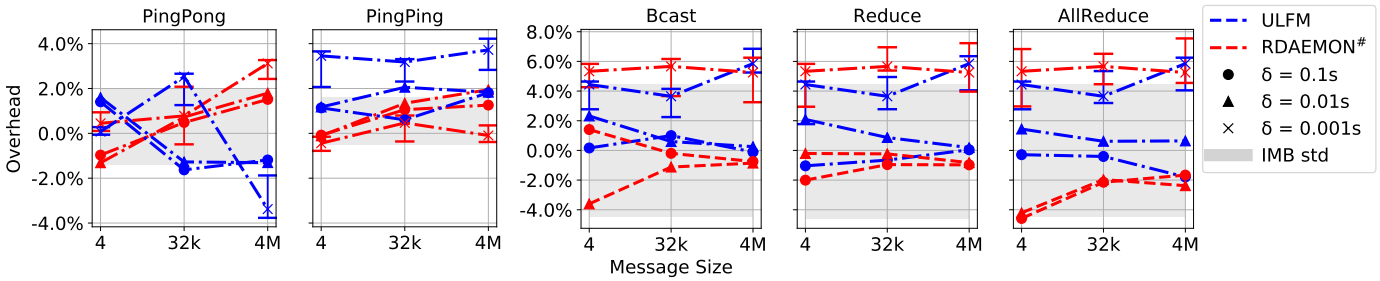

Figure 7: PRRTE with fault tolerance overhead over PRRTE and ULFM using IMB.

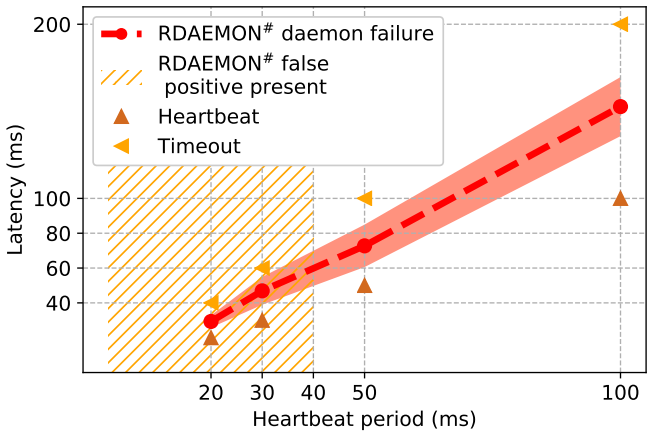

Figure 8: Accuracy with short detection timeout.

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

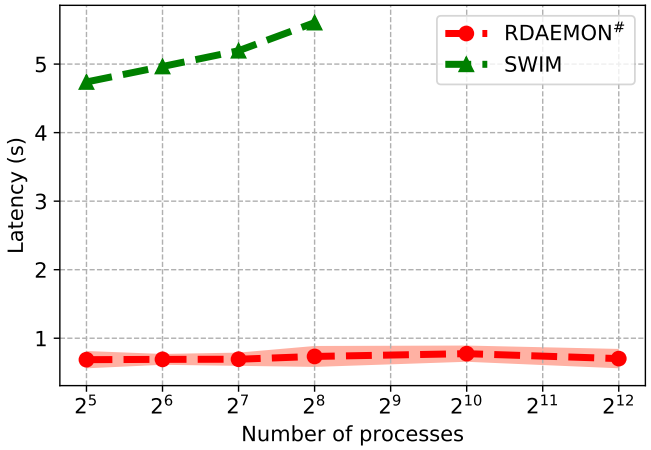

Figure 9: Detection latency comparison between RDaemon# and SWIM with increasing number of processes ($\delta = 0.5s$).

subgroup: if any node in the subgroup receives an acknowledge 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 acknowledgements messages. For the SWIM implementation, we use Go-Memberlist (#a8f83c6). We used a go-MPI interface to replicate our MPI detection benchmark with SWIM.

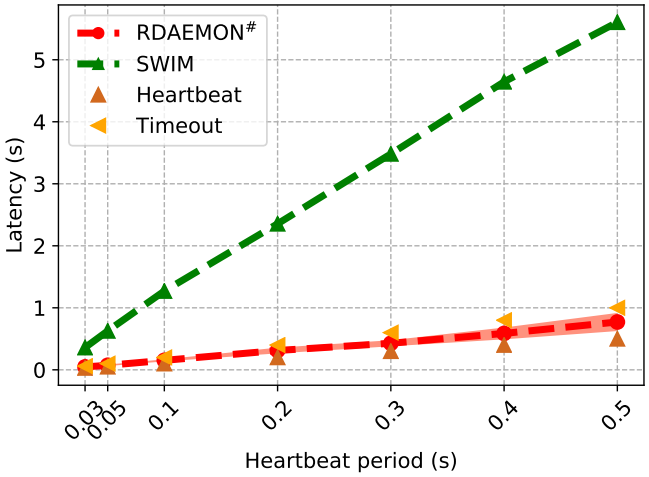

Figure 10: Detection and Propagation delay comparison between RDaemon# and SWIM with varying heartbeat period.

Figure 9 compares the scalability of the two detectors with regard to the number of deployed processes with $\eta = 1s$, $\delta = 0.5s$. We could run 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#, we run all tests 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 will be the bottleneck when scaling up (assuming the maximum connection requests limit issue can be solved).

Figure 10 compares single node failure detection and propagation latency between RDaemon# and SWIM with different heartbeat period settings. For all tests we set $\eta = \delta * 2$. The experiment uses 64 nodes in both cases; RDaemon# deploys on all 768 cores, but SWIM uses only 256 cores (due to not being able to 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, which demonstrates the efficiency of our 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.

5.5. *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:

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. For receiver it needs to poll 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 active polling BTL engine.
3. Using in-band detection directly from the high-performance network fabric to report unreachable error directly to the propagation component.

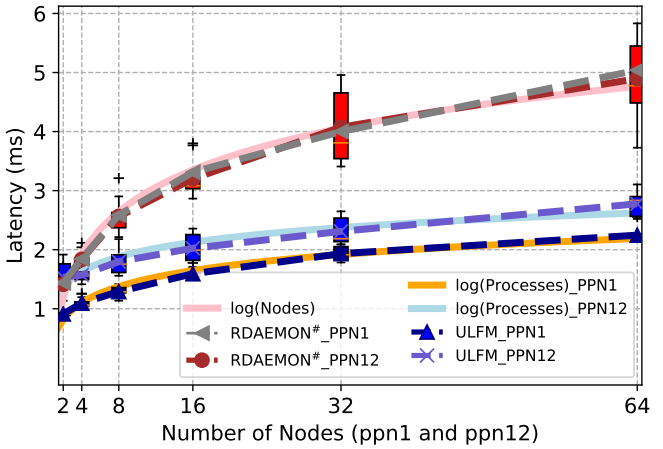

Figure 11: Process failure detection and propagation delay compared to ULFM.

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 (however, a different tag is employed to avoid disrupting the MPI matching). 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 progress the processing of 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 manycore systems. This helps reduce the total messages transferred and forwarded compared to ULFM, and the the lower bound for a full propagation is $\log_2(Number\ of\ Nodes)$.

Figure 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, in this process failure experiment, we do 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 performing a large part of the propagation between co-hosted processes (using the SM BTL transport) and employs InfiniBand communication for inter-node messages. RDaemon# uses TCP to broadcast between daemons, and each daemon uses a PMIx's notification to distribute the error information to all hosted processes. We can see that our implementation enjoys the same performance as ULFM but greatly reduces the complexity. The detection and propagation time is less than 5 mil-

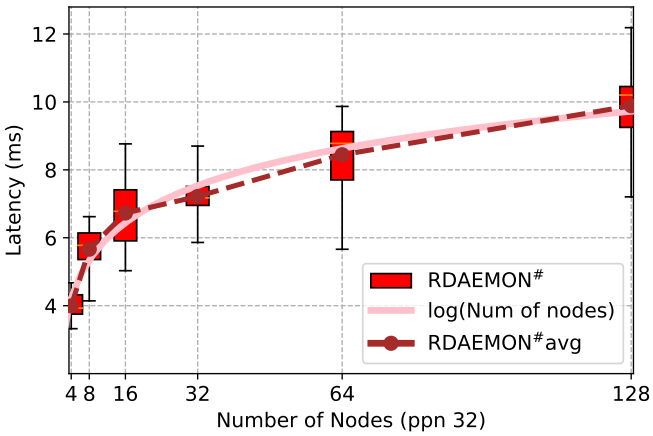

Figure 12: Process failure detection and propagation delay on Cori.

liseconds 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 fit $a * \log_2(N) + b$, which can be easily scale 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# scala-

bility, we scales the evaluation on the larger Cori system (with more processes per node). We can see in Figure 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).

5.6. Node Failures Detection

We now compare the detection latency 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.

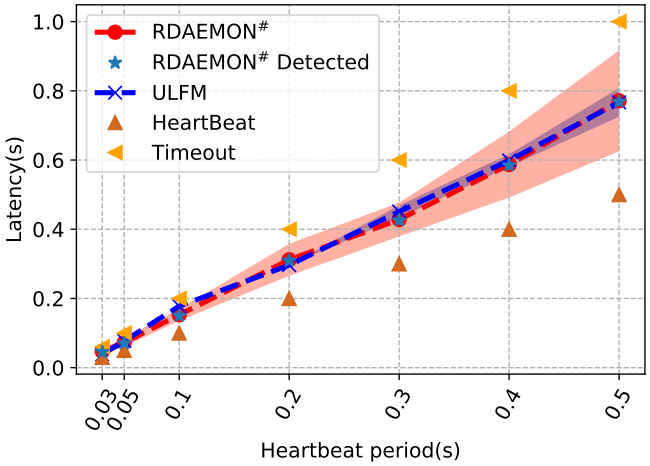

Figure 13: Single Daemon Failure detection and propagation delay compared to ULFM with different heartbeat period.

Figure 13 presents the behavior observed when injecting a single daemon failure under different heartbeat period settings. We conducted the experiments on 64 nodes with 764 processes. For RDaemon$^{\#}$ after synchronizing, we inject a node crash by ordering a process to kill its host daemon. For ULFM, all application processes on the target node suicide as a group. For the heartbeat period setting we start from 30 milliseconds to 0.5 second for both RDaemon$^{\#}$ and ULFM. For all heartbeat period, we set $\eta = \delta * 2$. From the figure, we can see that the detection latency in all cases lands in the interval $[\delta, \eta]$.

Figure 14 shows single node failure detection and propagation performance with a fixed heartbeat period $\delta = 0.5s$ and an increasing total number of 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 information, then distributes the packed message. From the figure see that RDaemon$^{\#}$ can detect and propagate a node failure between (0.5s, 1s) for all tested number of nodes.

The last experiment, presented in Figure 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. We 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 entails 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, the ordering of daemons on the detection ring should avoid setting nodes that have a correlated chance of failure sequentially (e.g., avoid choosing predecessor and successor from the same cabinet), which is easier to achieve when the detection infrastructure is split from the MPI rank ordering. To study the average behavior, we also inject failures at random nodes. In this case, the 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.

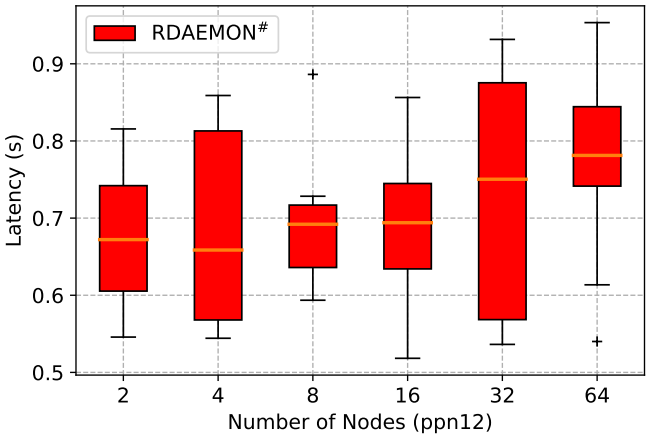

Figure 14: Single Daemon Failure Detection and Propagation delay with different number of nodes.

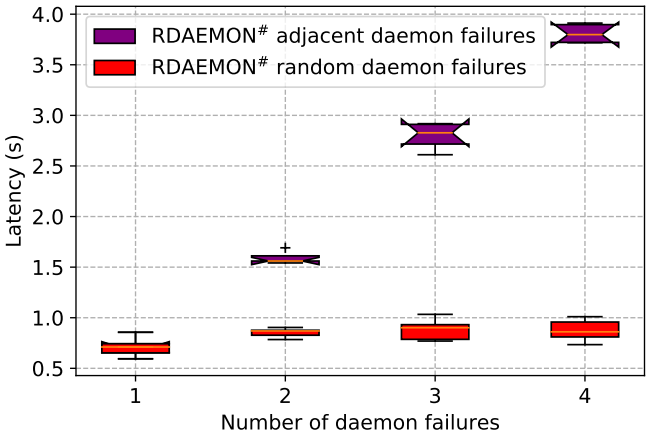

Figure 15: Multiple daemon failures at the same time.

6. 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. 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 sys-

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

In this section we investigate application support of RDaemon# with different programming models. Our interests focus on 1) demonstrating how our generic capabilities can support multiple programming languages, 2) how much overhead (if any) is incurred on both two-sided (e.g., MPI) and one-sized programming models (e.g., OpenSHMEM), and 3) provide the blueprints for supporting applications using multiple programming models. In hybrid applications and models, there is no "standard" method by which programming models can coordinate. For example, MPI has the standard MPI_Init function that must be called to initialize the library providing a "hook" within that function to notify others that it has been called. In contrast, OpenMP does not have an explicit call to "init" and is instead initialized on first use; older versions of OpenSHMEM also allow implicit initialization. Figure 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 type of applications when launching a single PRRTE Distributed Virtual Machine (DVM) (using the ***prte*** command), and then using 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;

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, we use the heavily communication-bound benchmark Graph500 [34]. Graph500 is an open specification effort to offer a standardized graph-based

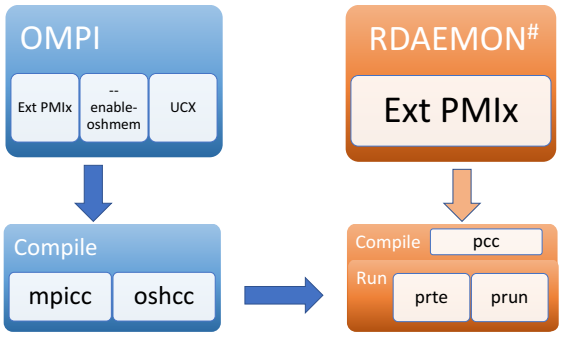

Figure 16: Hybrid programming model support of MPI and OpenSHMEM

benchmark across large-scale distributed platforms which captures the behavior of common communication-bound graph algorithms. Graph500 differs from other large-scale benchmarksm such as HPL, and HGPGMG in the way it primarily highlights data access patterns. Graph500 performs a breadth-first search (BFS) in parallel on a large randomly generated undirected graph. Our experiments use a the open source project OpenSHMEM Benchmark (OSB) suite [35] that features both MPI and OpenSHMEM based Graph500 implementations. For the application setting we use $scale\_factor = 20$, $edge\_factor = 16$ which generates an undirected graph with 2^{scale_factor} vertices and $2^{scale\_factor} * edge\_factor$ edges. The benchmark collects the statistics of the generation of the breadth-first search tree of 64 randomly selected vertices. It also collect the statistics of validation time which ensures that all connected components are visited which generate large amount of communications. For the experiments, we use NERSC Cori with 1K nodes. This results in a deployment with 32K MPI ranks, or 32K OpenSHMEM Processing Elements (PEs).

6.1. Two-sided Application

The ***mpi_test_simple*** benchmark is the baseline implementation of the BFS that uses two-sided MPI_Send and MPI_Recv and MPI_AllReduce. We evaluate the noise overhead incurred from heartbeats messages with different heartbeat periods on the point-to-point and collectives that are used in this benchmark.

Figure 17 shows the overhead incurred with the P2P communication during the BFS generation phase. We present in shaded gray, the variability of the BSF without our heartbeat detection enabled ($mean\_time\_of\_BFS \pm \sigma$). We calculate the overhead the same as in equation (1). For comparison, we plot the overhead with error bars for with different δ values. We can see that in 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 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 our failure detection heartbeats has barely no impact in communication intensive applications with both P2P and collective communications.

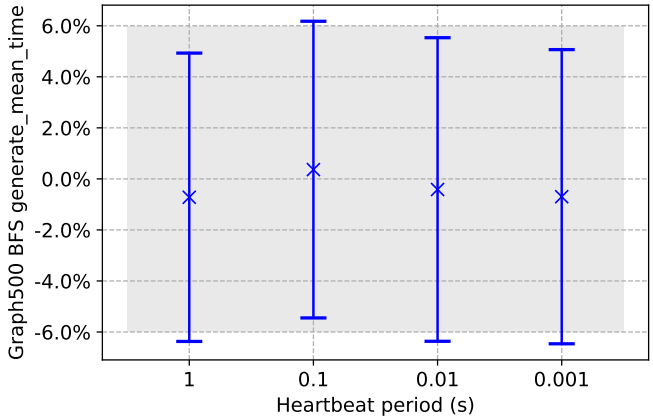

Figure 17: Overhead for generating BFS running mpi_test_simple when using PRRTE with fault tolerance over PRRTE (32K MPI ranks; the gray area represents the normal variability of the benchmark).

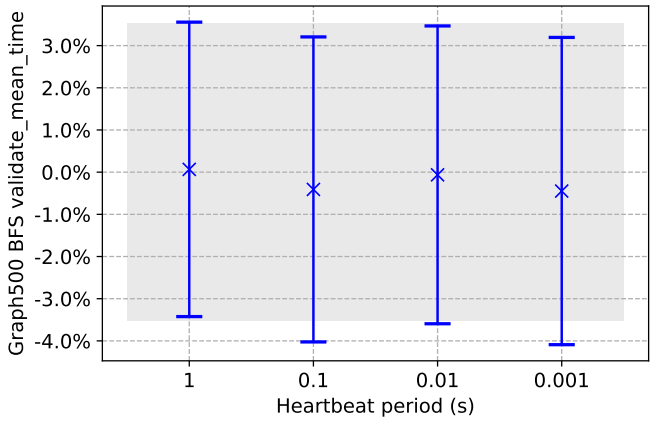

Figure 18: Overhead for validating BFS in mpi_test_simple when using PRRTE with fault tolerance over PRRTE (32K MPI ranks; the gray area represents the normal variability of the benchmark).

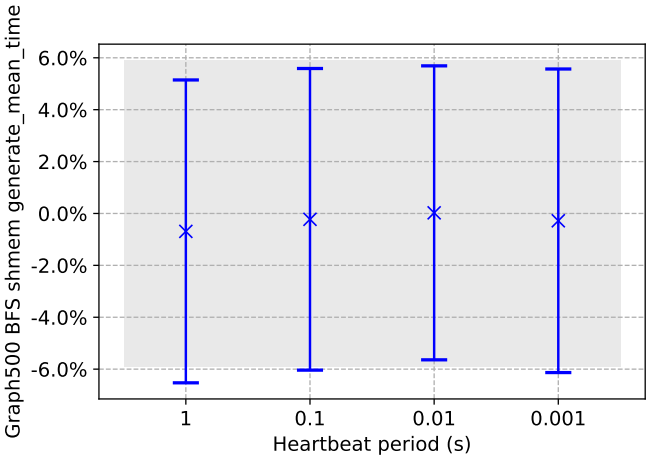

Figure 19: Overhead for generating BFS running graph500_shmem_one_sided upon PRRTE with fault tolerance over PRRTE (32K OpenSHMEM PEs; the gray area represents the normal variability of the benchmark).

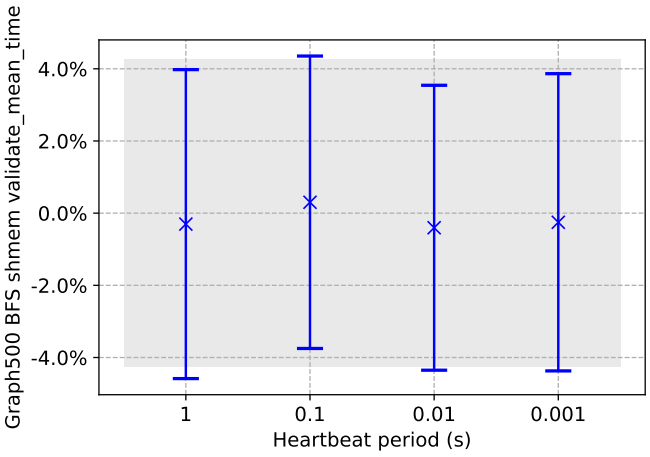

Figure 20: Overhead for validating BFS running graph500_shmem_one_sided upon PRRTE with fault tolerance over PRRTE (32K OpenSHMEM PEs; the gray area represents the normal variability of the benchmark).

6.2. One-sided Application

For the OpenSHMEM application, we 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 19 and Figure 20 shows 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 our detector does not stress the applications' communication.

7. Conclusion

Failure detection and propagation is a critical service for resilient systems. In this work, we present an efficient failure detection and propagation design and implementation for distributed systems. 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 depends on heartbeats and timeouts. Unlike gossip-based algorithms, it enjoys deterministic communication bounds and overhead to provide a reliable solution that works at scale, yet it doesn't require an over-specialization detrimental to applicability. Our design and implementation takes into account the intricate relationship and trade-offs between 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. Our solution addresses those concerns and is capable of tolerating 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. Our 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, we demonstrate in application benchmarks that our 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.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. (1725692); and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

[1] C. Di Martino, Z. Kalbarczyk, R. Iyer, Measuring the Resiliency of Extreme-Scale Computing Environments, in: Principles of Performance and Reliability Modeling and Evaluation, Springer, 2016, pp. 609–655. doi:10.1007/978-3-319-30599-8_24.

[2] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, J. J. Dongarra, An evaluation of user-level failure mitigation support in MPI, Computing 95 (12) (2013) 1171–1184. doi:10.1007/978-3-642-33518-1_24.

[3] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, H. Subramoni, Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications, Concurrency and Computation: Practice and Experience 0 (0) e4863. doi:10.1002/cpe.4863.

[4] C. Cao, T. Herault, G. Bosilca, J. Dongarra, Design for a soft error resilient dynamic task-based runtime, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 765–774. doi:10.1109/IPDPS.2015.81.

[5] O. Subasi, T. Martsinkevich, F. Zyulkyarov, O. Unsal, J. Labarta, F. Cappello, Unified fault-tolerance framework for hybrid task-parallel message-passing applications, The International Journal of High Performance Computing Applications 32 (5) (2018) 641–657. doi:10.1177/1094342016669416.

[6] P. Hao, S. Pophale, P. Shamis, T. Curtis, B. Chapman, 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, Springer-Verlag New York, Inc., New York, NY, USA, 2015, pp. 36–52. doi:10.1007/978-3-319-26428-8_3.

[7] A. Bouteiller, G. Bosilca, M. G. Venkata, Surviving errors with openshmem, in: M. Gorentla Venkata, N. Imam, S. Pophale, T. M. Mintz (Eds.), OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments, Springer International Publishing, Cham, 2016, pp. 66–81. doi:10.1007/978-3-319-50995-2_5.

[8] S. S. Hamouda, B. Herta, J. Milthorpe, D. Grove, O. Tardieu, Resilient x10 over mpi user level failure mitigation, in: Proceedings of the 6th ACM SIGPLAN Workshop on X10, X10 2016, ACM, New York, NY, USA, 2016, pp. 18–23. doi:10.1145/2931028.2931030.

[9] J. A. Zounmevo, D. Kimpe, R. Ross, A. Afsahi, Using mpi in high-performance computing services, in: Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, ACM, New York, NY, USA, 2013, pp. 43–48. doi:10.1145/2488551.2488556.

[10] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, J. Dongarra, Failure detection and propagation in hpc systems, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 312–322. doi:10.1109/SC.2016.26.

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

[12] D. Zhong, A. Bouteiller, X. Luo, G. Bosilca, Runtime level failure detection and propagation in hpc systems, in: Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI '19, ACM, New York, NY, USA, 2019, pp. 14:1–14:11. doi:10.1145/3343211.3343225.

[13] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, S. Matsuoka, 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, ACM, New York, NY, USA, 2011, pp. 32:1–32:32. doi:10.1145/2063384.2063427.

[14] Q. Sun, M. Romanus, T. Jin, H. Yu, P.-T. Bremer, S. Petruzza, S. Klasky, M. Parashar, 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, IEEE Press, Piscataway, NJ, USA, 2016, pp. 2–9. doi:10.1109/ESPM2.2016.12.

[15] R. Butler, W. Gropp, E. Lusk, A Scalable Process-Management Environment for Parallel Programs, 2000. doi:10.1007/3-540-45255-9_25.

[16] Mathematics, C. S. D. A. N. Laboratory, Hydra process management framework (2014).
URL https://wiki.mpich.org/mpich/index.php?title=Hydra_Process_Management_Framework

[17] G. Bosilca, T. Herault, A. Rezmerita, J. Dongarra, On scalability for mpi runtime systems, in: 2011 IEEE International Conference on Cluster Computing, 2011, pp. 187–195. doi:10.1109/CLUSTER.2011.29.

[18] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, G. E. Fagg, The open run-time environment (openrte): A transparent multi-cluster environment for high-performance computing, in: B. Di Martino, D. Kranzlmüller, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 225–232. doi:10.1007/11557265_31.

[19] J. M. Squyres, The architecture of open source applications:open mpi (2012).
URL http://www.aosabook.org/en/openmpi.html

[20] T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267. doi:10.1145/226643.226647.

[21] W. Chen, S. Toueg, M. K. Aguilera, On the quality of service of failure detectors, IEEE Transactions on Computers 51 (1) (2002) 13–32. doi:10.1109/12.980014.

[22] M. Larrea, A. Fernandez, S. Arevalo, Optimal implementation of the weakest failure detector for solving consensus, in: Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000, 2000, pp. 52–59. doi:10.1109/RELDI.2000.885392.

[23] M. Kawazoe Aguilera, W. Chen, S. Toueg, Heartbeat: A timeout-free failure detector for quiescent reliable communication, in: M. Mavronicolas, P. Tsigas (Eds.), Distributed Algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 126–140. doi:10.1007/BFb0030680.

[24] R. van Renesse, Y. Minsky, M. Hayden, A gossip-style failure detection service, in: N. Davies, S. Jochen, K. Raymond (Eds.), Middleware'98, Springer London, London, 1998, pp. 55–70.

[25] S. Ranganathan, A. D. George, R. W. Todd, M. C. Chidester, Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters, Cluster Computing 4 (3) (2001) 197–209. doi:10.1023/A:1011494323443.

[26] I. Gupta, T. D. Chandra, G. S. Goldszmidt, On scalable and efficient distributed failure detectors, in: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC '01, ACM, New York, NY, USA, 2001, pp. 170–179. doi:10.1145/383962.384010.

[27] A. Das, I. Gupta, A. Motivala, Swim: scalable weakly-consistent infection-style process group membership protocol, in: Proceedings International Conference on Dependable Systems and Networks, 2002, pp. 303–312. doi:10.1109/DSN.2002.1028914.

[28] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database maintenance, in: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '87, ACM, New York, NY, USA, 1987, pp. 1–12. doi:10.1145/41840.41841.

[29] T. Angskun, G. Bosilca, J. Dongarra, Binomial graph: A scalable and fault-tolerant logical network topology, in: I. Stojmenovic, R. K. Thulasiram, L. T. Yang, W. Jia, M. Guo, R. F. de Mello (Eds.), Parallel and Distributed Processing and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 471–482. doi:10.1007/978-3-540-74742-0_43.

[30] P. Shamis, R. Graham, M. G. Venkata, J. Ladd, Design and implementation of broadcast algorithms for extreme-scale systems, in: 2011 IEEE International Conference on Cluster Computing, 2011, pp. 74–83. doi:10.1109/CLUSTER.2011.17.

[31] R. H. Castain, Rfc0015:job control and monitoring apis (2017).
URL https://pmix.org/pmix-standard/job-control-and-monitoring/

[32] R. H. Castain, Rfc0002:pmix event notification (2017).
URL https://pmix.org/pmix-standard/event-notification/

[33] G. S. (Blackbelt), Introducing intel mpi benchmarks.
URL https://software.intel.com/en-us/articles/intel-mpi-benchmarks

[34] J. Ang, B. Barrett, K. Wheeler, R. Murphy, Introducing the Graph 500 (05 2010).
URL https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-Murphy-paper.pdf

[35] E. F. D'Azevedo, N. Imam, Graph 500 in OpenSHMEM, in: M. Gorentla Venkata, P. Shamis, N. Imam, M. G. Lopez (Eds.), OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies, Springer International Publishing, Cham, 2015, pp. 154–163. doi:10.1145/3144779.3144781.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: