TOSS-2020: A Commodity Software Stack for High-Performance Computing

E. A. Leon, T. D'Hooge, N. Hanford, I. Karlin , R. Pankajakshan, J. Foraker, C. Chambreau, M. Leininger

April 27, 2020

International Conference for High Performance Computing, Networking, Storage and Analysis
Atlanta, GA, United States
November 15, 2020 through November 20, 2020

TOSS-2020: A Commodity Software Stack for HPC

Edgar A. León, Trent D'Hooge, Nathan Hanford, Ian Karlin, Ramesh Pankajakshan,
Jim Foraker, Chris Chambreau, and Matthew L. Leininger
Livermore Computing
Lawrence Livermore National Laboratory
Livermore, California, USA
Email: {leon, dhooge1, nhanford, karlin1, pankajakshan1, foraker1, chambreau1, leininger4}@llnl.gov

***Abstract*—The simulation environment of any HPC platform is key to the performance, portability, and productivity of scientific applications. This environment has traditionally been provided by platform vendors, presenting challenges for HPC centers and users including platform-specific software that tend to stagnate over the lifetime of the system. In this paper, we present the Tri-Laboratory Operating System Stack (TOSS), a production simulation environment based on Linux and open source software, with proprietary software components integrated as needed. TOSS, focused on mid-to-large scale commodity HPC systems, provides a common simulation environment across system architectures, reduces the learning curve on new systems, and benefits from a lineage of past experience and bug fixes. To further the scope and applicability of TOSS, we demonstrate its feasibility and effectiveness on a leadership-class supercomputer architecture. Our evaluation, relative to the vendor stack, includes an analysis of resource manager complexity, system noise, networking, and application performance.**

***Index Terms*—Scientific computing, Accelerator architectures, Parallel architectures, Multicore processing, Multiprocessor interconnection networks, Parallel machines, Supercomputers, Processor scheduling, Cluster computing, High performance computing, Software performance, Software reusability, System software, Operating systems, Utility programs, Programming environments, Runtime, Runtime environment, Software libraries.**

I. Introduction

A critical aspect of any HPC platform is the simulation environment provided to address the programming, code building, performance analysis, and workflow requirements of the supercomputing center's user community. A tremendous amount of technical effort and time is required to harden this environment for a given HPC hardware platform. A new system and simulation environment can easily take a year or more to debug and harden, with ongoing support and system upgrades requiring additional debugging and hardening over the lifetime of the system.

Traditionally, the simulation environment has been provided by the platform vendor, and strongly coupled to the provider's hardware and software offerings as part of an end-to-end *turnkey* solution. To date, Lawrence Livermore National Laboratory (LLNL) has followed this approach for its leadership class Advanced Technology Systems (ATS), such as *Sierra* and *Sequoia*, for the National Nuclear Security Administration's (NNSA) Advanced Simulation and Computing (ASC) program. However, these platform-specific simulation environments present significant challenges. They vary across system generations and providers, and vendor-provided simulation environments often stagnate over the lifetime of the system, with limited to no software updates after the first few years of use. Performance, stability, and scalability bugs are often repeated across vendors and even across software updates from one vendor. All of these challenges result in additional labor overheads for computing center support staff, and user inefficiencies in managing and performing their computational workflows.

With the rise of commodity clusters in the early 2000's, LLNL began providing a production simulation environment based on Linux and open source software, with proprietary software components integrated as needed. This work provided the foundation for the Tri-Laboratory Operating System Stack (TOSS) environment [1] that was deployed on commodity technology systems at LLNL and other NNSA laboratories beginning in 2007. Ever since, TOSS has provided a common simulation environment across multiple generations of commodity HPC systems ranging from tens to over 3,000 nodes, and from (x86 and arm) CPU-only nodes to those containing GPUs. Currently, TOSS is running on most LLNL systems, with a limited support staff administering a growing number of systems. This common simulation environment has created a more robust and efficient user experience that enables a wide range of HPC workflows, benefits from a lineage of past experience and bug fixes, propagates improvements across all TOSS systems, reduces the user learning curve on new systems, and provides the flexibility to evolve the simulation environment over the lifetime of the system.

In this paper, we present the TOSS simulation environment and demonstrate its feasibility on a CORAL [2] platform supercomputer similar to the top two supercomputers in the world, *Summit* and *Sierra*, but of lesser scale. The architecture of these leadership-class systems have shifted from highly customized hardware and software towards commodity processors, accelerators, and open-source software, thus aligning with the standard technologies supported by TOSS. We demonstrate the effectiveness of TOSS and address the TOSS feature and performance gaps relative to the vendor-provided simulation environment.

Enabling user productivity is an important aspect of a software environment. Being able to express resource needs for a job rapidly without error is important to users, who interact with resource and job management software [3]–[6] provided by the software stack. They present different tradeoffs in terms of features, scalability, and usability, with recent software managers enabling more complex workflows natively [7]. A highly scalable system requires, in part, a software stack that minimizes system noise. System noise is caused by operating system (OS) processes interfering with application threads on compute nodes, which impacts application performance and scalability, and produces runtime variability [8]–[11]. Because applications have no control over when and where these system processes run, the interference appears as *noise* to the application. In addition, network and communication performance is important for applications to communicate scalably across a parallel machine [12]–[14]. In *turn-key* solutions this layer is usually enabled by vendor-provided software. Thus, this work evaluates the success of TOSS by demonstrating that it: (1) enables user productivity natively; (2) sufficiently mitigates system noise; (3) provides high-performance networking capability; and (4) enables good application performance.

The rest of this paper is organized as follows. Section II describes TOSS while Section III highlights *initial* key challenges in running TOSS on ATS systems. The experimental methodology and execution environment are described in Section IV. Sections V, VI, VII, and VIII evaluate TOSS relative to the vendor stack in the context of resource manager complexity, system noise, networking, and application performance, respectively. Finally, Section IX concludes this work with our key findings and recommendations on additional work to enable future systems.

II. TOSS: The Tri-laboratory Operating System Stack

TOSS provides a common, robust operating environment for HPC clusters across the NNSA laboratories. TOSS integrates a commodity base operating system with cluster administration tools, a parallel filesystem, batch scheduler, and reference development environment. These components are then rigorously tested on hardware platforms and with codes that are relevant to the NNSA missions.

TOSS follows a regular release cadence, with monthly updates to address routine bug and security fixes. Each update is integration tested using LLNL's Synthetic WorkLoad to ensure that it is stable for production use. Minor releases include more significant component updates, and occur every 6-9 months, generally in line with upstream vendor releases. New major releases occur every 3-5 years, and involve major version changes to the base operating system. TOSS endeavors to provide a stable user ABI throughout the entire life cycle of a major release, limiting the need for users to recompile their code after system updates.

A major goal of TOSS is the reduction of costs across NNSA laboratories. Standardization on a single operating environment across platforms reduces the time spent by application developers porting and optimizing codes and workflows, and enables system administrators to more easily work across clusters without requiring platform-specific training. Standardization across multiple sites enhances collaboration and reduces duplication of efforts. Maintaining the project across platform lifecycles helps to ensure that previously-encountered issues are not replicated on new platforms. Finally, TOSS reduces development and support costs by using off-the-shelf components when possible, and performing local development only when necessary to fill gaps or ensure efficient code execution.

A. *TOSS Components*

Red Hat Enterprise Linux (RHEL) [15] is the base operating system for TOSS, and provides the majority of packages (see Table I). Additional packages are brought in from Fedora's Extra Packages for Enterprise Linux (EPEL [16]) collection. The current major version of TOSS, TOSS 3, is based on RHEL 7. Building on a commercial Linux distribution enables TOSS developers to leverage Red Hat's extensive engineering resources to mitigate many non HPC-specific bugs and security issues. Additionally, managing and working on a TOSS system remains similar to RHEL, and software built for RHEL generally works out-of-the-box on TOSS.

TABLE I
TOSS 3.5-8 PACKAGES BY SOURCE AND ARCHITECTURE

Source	x86_64	ppc64le	aarch64
RHEL (unmodified)	4009	3851	3893
RHEL (rebuilt)	35	37	42
EPEL	707	657	757
Vendor-proprietary (unmodified)	48	11	31
Locally built	203	186	180
Total	5002	4742	4903

While nearly all RHEL packages are used unmodified, a small number are rebuilt. These modified versions often remain in place for only a few releases, providing additional diagnostics for root cause analysis, critical bug fixes, or new hardware support until RHEL's packages incorporate the necessary funcitonality. The RHEL kernel is also minimally-modified to add additional missing hardware drivers and critical bug fixes. TOSS thus further hardens RHEL for existing NNSA platforms and workloads, while rapidly supporting incoming platforms of interest.

TOSS integrates several additional components to create a robust HPC environment. These include the Slurm batch scheduler [3], and support for Lustre clients and servers [17], including support for ZFS [18]. Singularity [19] and CharlieCloud [20] are provided for containerized workflows. The Mellanox OpenFabrics Enterprise Distribution for Linux (MOFED [21]) is provided as an optional replacement for RHEL's InfiniBand fabric stack.

A number of HPC-centric system management tools are also added, many developed at LLNL or by other national laboratories. These include conman [22], munge [23], pdsh [24], powerman [25], and the LDMS metrics collection system [26].

TOSS includes a small reference development environment (DE), including multiple compilers, and GPU and MPI libraries. This environment is used for integration testing, verification, and cross-site compatibility.

B. *TOSS Limitations*

TOSS's focus on cross-platform compatibility and production stability impose some limitations on the platforms and software features that it can easily support. For instance, system architectures that diverge considerably from commodity platforms, such as IBM Blue Gene/Q [27], would impose significant maintenance costs for TOSS. Similarly, TOSS is not a fitting proving ground for experimental software features that may have unintended stability or performance consequences, such as invasive changes to the Linux kernel's scheduler or memory management.

While TOSS provides the necessary tools to operate an HPC cluster, it is not a turn-key solution. Sites retain great latitude in how TOSS runs on their systems, including choices in configuration management, provisioning, and monitoring. Consequently, sites running TOSS still require highly skilled HPC system architects and administrators, albeit in fewer numbers.

Rather than providing a complete and fast-moving development environment, TOSS aims to provide stable systems and ABIs that rich DEs can be built upon. LLNL separately maintains a DE on TOSS systems, containing a broad range of compilers, debuggers, MPIs, and other development tools. This separation enables the DE to more rapidly incorporate new software and respond to user needs with few worries about overall system stability. Separate work is being done to expand this Tri-Laboratory Computer Environment (TCE) for use at other computer centers with the next major version of TOSS. LLNL also supports Spack [28], which enables users to easily build and deploy custom development environments on TOSS systems.

C. *Comparisons with other HPC Operating Environments*

Scientific Linux [29] shares a common goal with TOSS—providing a common, reliable OS platform for a particular group of scientific sites. Like TOSS, Scientific Linux is also based on RHEL, but re-compiles the RHEL source packages rather than including them directly. By doing so, Scientific Linux is unencumbered by Red Hat's licensing and is freely distributable, but is not eligible for Red Hat support. Scientific Linux also does not directly target HPC, and as such does not integrate features like a batch scheduler or parallel filesystem.

The Cray Linux Environment (CLE) [30] also extends a commodity Linux distribution (SUSE [31]) to support HPC users. Unlike TOSS, it is highly optimized to run only on Cray's systems, and is not supported across multiple hardware vendors. Similarly to TOSS, CLE supports the Slurm resource manager, but also provides support for several additional batch systems. CLE also bundles custom provisioning, configuration management, and monitoring tools into a turn-key operating environment.

The OpenHPC project provides repositories of HPC-centric packages compiled for multiple operating systems [32]. Unlike TOSS, OpenHPC does not provide a base operating system, and much of its focus is on the development environment rather than system software. The TCE environment provides similar functionality to OpenHPC, but better optimized for TOSS and NNSA platforms and workloads. As a community project, OpenHPC is not able to integrate commercial components, such as compilers, and support is only provided on an ad-hoc basis.

III. TOSS CHALLENGES ON ADVANCED TECHNOLOGY SYSTEMS

The shift in ATS architectures from highly customized (e.g., IBM Blue Gene) to standard commodity (e.g., CORAL) components allows us to explore the possibility of leveraging TOSS across a wider range of NNSA platforms. Our investigation identified gaps in software support or features that had to be addressed in order to support the CORAL architecture. We describe those *initial* challenges and the employed solutions that helped to narrow the gaps and allow us to begin developing TOSS support for CORAL systems.

The most significant challenge was to determine the component versions necessary to pass various functionaity, performance, and stability (FPS) tests. Since TOSS includes more recent versions of many of the components found in the IBM stack, initial expectations were that few modifications would be necessary. Early performance results however identified a number of issues. In some cases, such as OpenMPI, upgrading to the latest available version improved results. However, incompatibilities and regressions were determined to still exist when using the latest versions of the RHEL kernel, MOFED stack, and NVIDIA drivers together. Since prior versions of these components were known to perform acceptably in the IBM stack, an iterative process was used to identify and install versions of these components from prior TOSS releases that did not show these misbehaviors.

In other cases, interactions between existing TOSS and CORAL software caused performance issues. For instance, a significant slowdown was discovered when moving very large blocks of data from main to GPU memory. Profiling revealed that the slowdown was due to an NVIDIA driver kernel thread spending significant time in ZFS filesystem memory management code, even with no ZFS filesystems mounted. Removing the unused ZFS modules eliminated the slowdown. No similar behavior has been witnessed on other ZFS systems, but without access to the NVIDIA driver source code, it is difficult to conclusively determine which component is at fault.

The complexity and abundance of software regressions in CORAL components demonstrated the need to thoroughly test all components at each layer. Effort may be wasted by, for example, attempting to optimize an MPI when the underlying GPU and network drivers have not been sufficiently tested. We also found that it was important for the tests to do *meaningful* work. Many existing testing and reporting tools perform simple operations that do not catch most issues. For example, while the CUDA `deviceQuery` call may succeed, more complicated calls such as `threadMigration` may still routinely fail, exposing a deeper underlying problem.

While such compatibility issues are common when bringing up new platforms, these challenges are compounded when dealing with multiple proprietary software components. As a result, vendors have typically focused on providing a single working OS version, which is only upgraded with relatively major sofware upgrades. In contrast, TOSS provides smoother rolling updates that can incorporate minor changes and security fixes without significant FPS issues. This is enabled through the combination of open source software and extensive collaborative integration testing by Red Hat and LLNL.

IV. Methodology and Execution Environment

We use the *Lassen* supercomputing system at LLNL, an IBM system comprised of 792 heterogeneous nodes. Each node has two IBM Power9 processors coupled with two NVIDIA Volta GPUs per processor. There are 22 cores per Power9 processor and 4 hardware threads per core (SMT-4). Each node has 256 GB of DDR4 memory and 16 GB of HBM memory per GPU. A Power9 processor has up to 170 GB/s peak DDR4 bandwidth and each GPU 900 GB/s peak bandwidth to local HBM. As Figure 1 shows, there are two groups within a node, each consisting of two GPUs and a Power9 processor. Each pair of devices within a group is interconnected with three NVLink channels for a 75 GB/s unidirectional bandwidth. Each node includes a dual-port EDR InfiniBand NIC. Each port's theoretical peak bandwidth is 12.5 GB/s (100 Gb/s).

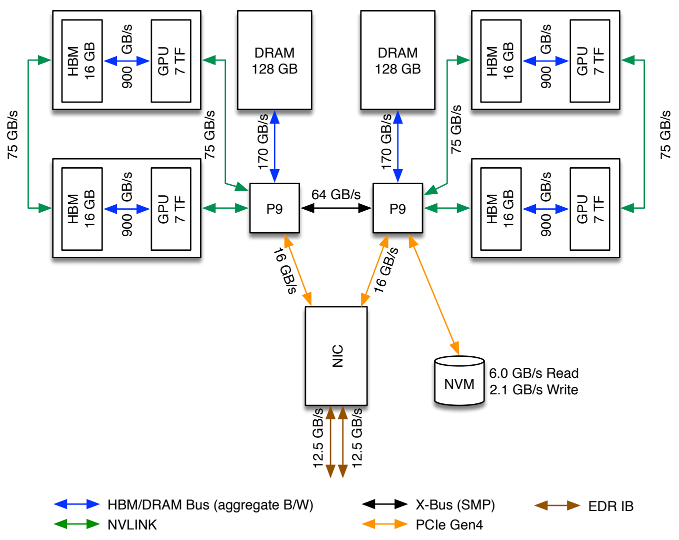


Fig. 1. Node architecture of the *Lassen* supercomputer.

We configure the *Lassen* machine using two different software stacks, one at a time. The first machine configuration—*Ice*—utilizes the vendor-provided software stack, while the second configuration—*Fire*—uses TOSS. The only difference between *Ice* and *Fire* is the software; we summarize the key components in Table II.

One of the differences between *Fire* and *Ice* is the batch scheduler and resource manager components. The batch scheduler is responsible for finding and allocating the resources that fulfill a job's request. When a job is scheduled to run, the scheduler instructs the resource manager to launch the application across the job's allocated resources. *Fire* uses Slurm [3] while *Ice* uses the IBM Spectrum Load Sharing Facility (LSF) and the Cluster Systems Management (CSM) [4]. Another difference is the MPI software stack. *Fire* uses the OpenMPI library implemented on top of the UCX communication framework [33], while *Ice* uses IBM Spectrum MPI, an MPI library based on OpenMPI and implemented on top of the Parallel Active Message Interface (PAMI) [34]. In addition to Spectrum MPI, we also built OpenMPI–UCX on *Ice* to have another point of reference for our network evaluation.

TABLE II
Common Components of TOSS and the IBM Stack.

	TOSS	IBM Stack
OS	RHEL 7.6	RHEL 7.6
Network	MOFED 4.5	MOFED 4.5
GPUs	NVIDIA 440.64.00 CUDA 10.2 GDRCopy	NVIDIA 418.87.00 CUDA 10.1 GDRCopy
MPI	OpenMPI 4.03 UCX 1.7	Spectrum MPI 10.3.0, OpenMPI 4.03 PAMI, UCX 1.7
BatchSched ResMgr	Slurm 19.05.5	IBM LSF 10.1 IBM CSM 1.6.0
Compilers	GCC 7.3.1 NVCC 10.2.89	IBM XL 16.1.1, GCC 7.3.1 NVCC 10.1.243

Testing a full HPC software stack is a complex endeavor. Instead of evaluating each component, our strategy focuses on assessing the efficacy of the system determined through key micro-benchmarks, mini-applications, and full-fledged applications. We perform an empirical evaluation that measures various aspects of the system to quantitatively score the readiness of the TOSS software stack. We focus on evaluating how well each software stack deals with system noise, leverages the network capability, and, ultimately, delivers application performance. The next few sections describe our experiments and findings in each of these areas.

V. Reducing Resource Manager Complexity

As described in Table II, TOSS and the IBM stack use different batch schedulers and resource managers. On one hand, the IBM stack uses IBM-specific packages: the LSF batch scheduler and the CSM resource manager. On the other, TOSS uses the Slurm workload manager, an open-source, broadly-used software package for Linux systems. In this section, we qualitatively compare the resource managers of both stacks focusing on the complexity associated with mapping applications efficiently onto the hardware.

CSM uses `jsrun`, a powerful tool that allows a high degree of customization. CSM exposes a myriad of ways to map an application's MPI tasks to the hardware through the notion of a *resource set*, an abstraction to assign resources to a task or a set of tasks. A resource set assigns cores and GPUs to MPI tasks and can be modified with affinity and binding options to prevent unnecessary migrations by the Linux scheduler. A common application configuration is to use the same number of MPI tasks per GPU. A program launch with two tasks per GPU on *Ice* follows:

$$jsrun -r4 -a2 -c10 -g1 -dpacked -bpacked{:}5 \quad app$$

This commands creates four resource sets per node, each set with two MPI tasks, ten cores, and one GPU; assigns tasks into the first resource set before using the next set (`-d`); and binds each task to five cores (40 cores per node / 8 tasks) within a resource set (`-b`). With CSM's high degree of customization comes high complexity in expressing a simple concept: *two MPI tasks per GPU*. Note that without specifying similar controls for placement, the default policy is to map the tasks to the first eight cores and the first GPU. This results in oversubscription of certain resources while leaving most of the node resources idle.

Slurm uses `srun` to launch commands onto the hardware. While it may not have the same degree of customization as CSM, particularly for heterogeneous systems, it presents a familiar interface to most Linux users. Slurm also provides options for resource affinity and binding including `--cpu-bind` and `--accel-bind`. These compute-driven options are available in TOSS.

However, the default TOSS affinity and binding policy is provided by coupling Slurm with *mpibind* [35]–[37], which is called implicitly with the number of tasks as input. Mpibind is a memory-driven algorithm that maps hybrid applications onto heterogeneous hardware. It is designed to map applications to the memory hierarchy, not the compute resources. Mapping by the memory hierarchy, mpibind prioritizes spreading tasks evenly across NUMA domains within a node, reduces remote memory accesses, and leverages locality of CPUs and GPUs within NUMA domains transparently to users.

On *Fire*, the same program configuration and affinity presented above is expressed as follows:

$$srun -ntasks\text{-}per\text{-}node = 8 \quad app$$

To launch an application in TOSS a user only needs to know the ratio of MPI tasks to GPUs and the number of GPUs in a node. However, with the IBM stack, a user needs to know the number of cores and GPUs on a compute node, distribute them accordingly to MPI tasks, bind each task to the corresponding CPUs, and distribute the ranks across the resource sets. Not following this recipe can have substantial penalties on performance and scalability (e.g., default policy).

From a user's perspective using CSM, it is easy to inadvertently over- or under-subscribe resources, run into errors, or have conflicting options. The high learning curve and complexity of using CSM, prompted us to develop an abstraction similar to the TOSS systems where mpibind provides the default mapping, but jsrun is still employed to launch jobs. Such abstraction is called *lrun*. The example above on *Ice* with lrun follows:

$$lrun - T8 \quad app$$

With lrun's simple interface, users can rapidly execute their applications onto the system with a reasonably efficient mapping. And, if custom mappings are necessary, jsrun is still an option.

TOSS presents a familiar and easy-to-use interface that leverages the investments made on a broadly used resource manager for commodity systems. The IBM stack provides a rich resource manager capable of highly customizable mappings, but is more prone to user error due its complexity and is limited to IBM systems.

VI. Mitigating system noise

System noise is a prominent cause of performance degradation and scalability limitations [38]–[40], particularly on commodity technology systems. For instance, the performance of a finite-element hydrodynamics application can vary up to 2.4X [8]. We refer to system noise as any process, hardware or software, that delays an application's execution and is not directly controlled by the application. In this section, we study the impact of system noise on performance using FWQ-MPI, a parallel version of a well-known noise benchmark (we discuss the impact on applications in Section VIII). We use a number of approaches to mitigate noise and compare them on both TOSS and the IBM software stack. These approaches include variations of core specialization [27], [41], [42] and what we call the free-core strategy. Other strategies to mitigate noise can be found in the literature [9], [40], [43]–[46]. Table III summarizes the studied configurations. We also note that this study does not consider I/O, which is a subject of future work.

TABLE III
System noise configurations. *System cores* refer to cores dedicated exclusively for OS services, while *App-free cores* refer to cores not used by applications—it is up to the OS to schedule system services on them. Core specialization is achieved with IBM CSM on *Ice* and RHEL Tuna on *Fire*.

	Configuration	#usr cores	System cores	App-free cores	CSM Blink
Ice w/ IBM CSM	Baseline	44	—	—	—
	FreeCore	40	—	0,1,22,23	—
	CoreSpec	40	0,1,22,23	0,1,22,23	off
	CoreSpecBlink	40	0,1,22,23	0,1,22,23	on
Fire w/ RHEL Tuna	Baseline	44	—	—	—
	FreeCoreFirst	40	—	0,1,22,23	—
	FreeCoreLast	40	—	20,21,42,43	—
	CoreSpec	40	0,1,22,23	0,1,22,23	—

The rest of this section is structured as follows. First, we describe FWQ-MPI and provide a performance baseline—performance without noise mitigation. Then, we describe and evaluate a simple technique that leaves one or more compute cores free from an application's perspective so that the OS can use them for system processing. Finally, we evaluate core specialization, which uses dedicated cores for system processing.

A. FWQ-MPI: Parallel FWQ

The Fixed Work Quantum (FWQ) benchmark [47] records a series of samples, where each sample records the time required to execute a fixed amount of work. FWQ-MPI executes multiple concurrent instances of FWQ as an MPI job [8], where each instance or task is bound to a core. The tasks only communicate to synchronize their start time and to aggregate sample data at the end. On a noiseless system, each sample time should be identical. On a *noisy* system, some samples take longer than normal to complete, because the application process is interrupted to allow system processes to execute.

B. Baseline: Use all the cores of a node

The baseline configuration consists of running FWQ-MPI *using all the cores of a node* in a manner similar to many applications: one MPI task per core for a total of 44 MPI tasks. In this section, we refer to FWQ-MPI as the *application* to distinguish it from OS and system processes.

Figure 2a shows the baseline results on *Fire*. The horizontal axis is the sample number, while the vertical axis is the time taken for each sample in milliseconds. All cores recorded samples in parallel, and all are displayed individually on the graph. FWQ was configured to record 100,000 samples each with a nominal execution time of $\sim$1.4 milliseconds.

In a noise-free environment, each iteration would take roughly the same time. As the figure shows however, some iterations take much longer. This is the result of interference from the OS and system processes running on the same cores as FWQ. Since many of these processes execute at regular intervals and with a fixed duration, some of the affected iterations show at regular intervals on the horizontal axis and with a fixed added delay on the vertical axis. For example, core 19 shows repeated execution times of $\sim$2.6 ms—about 4 instances per 20,000 iterations.

Although possible to discern from Figure 2a, the most affected cores are salient with a box-and-whiskers plot as shown in Figure 2c. System processes are running mostly on cores 1 and 19. We ran this experiment multiple times and found that only one or two cores are significantly affected and the actual cores vary from run to run. Table IV shows basic statistics for the most affected core, based on the standard deviation (σ), for any given noise configuration. In particular, core 19 was the most affected core in *Fire*'s baseline.

TABLE IV
STATISTICS FOR THE VARIOUS SYSTEM NOISE CONFIGURATIONS FOR THE MOST-AFFECTED CORE BASED ON THE STANDARD DEVIATION (σ).

			Min	Mean	Max	Std	
Ice		Baseline	1.483	1.582	10.552	0.095	C22
		FreeCore	1.461	1.539	3.373	0.081	C5
		CoreSpec	1.463	1.532	1.640	0.016	C38
		CoreSpecBlink	1.485	1.533	1.713	0.011	C42
Fire		Baseline	1.483	1.580	2.836	0.069	C19
		FreeCoreFirst	1.463	1.537	2.764	0.066	C15
		FreeCoreLast	1.463	1.536	2.764	0.066	C18
	CoreSpec	Clean+irqbal	1.463	1.541	2.762	0.067	C24
		Clean+snmpd	1.505	1.531	2.749	0.023	C2
		Clean	1.507	1.536	2.523	0.011	C32
		Clean-nvidia	1.484	1.535	2.675	0.010	C40

We now contrast the noise from *Fire* with *Ice*. Figure 2b shows a large number of samples are being affected on many cores, and the magnitude of the interference is significantly larger than on *Fire*—as table IV shows, the maximum sample time on *Fire* is 2.8 ms and on *Ice* is 10.6 ms (the y-axis of *Ice*'s plot is capped to the same limit as *Fire*'s). Thus, the system processes on *Ice* cause significantly more interference.

The box-and-whisker representation of the data from Figure 2d also shows that over half the cores are affected significantly and most of them are on the first socket (cores 0-21). We obtained a similar result across multiple repetitions of the same experiment.

C. Free-core strategy

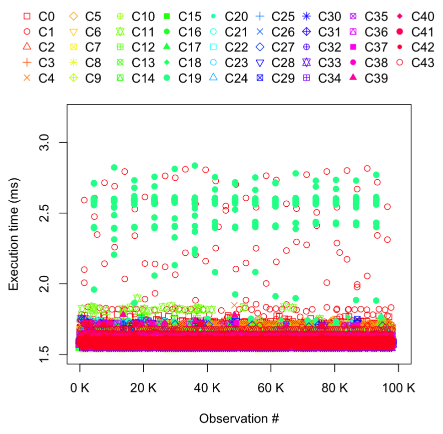
This configuration leaves *two cores per socket idle* from an application's perspective, i.e., FWQ-MPI uses only 20 cores per socket. The idea is that the idle cores can be utilized by the OS to perform system services, thus, reducing application interference. There is no guarantee however that the OS will do so based on how the system is being utilized. As we will also show, the choice of idle cores matters. We study two modes: idling the first two cores of each socket and the last two cores of each socket.

The free-core strategy can be particularly beneficial when a compute node has many cores. Application developers typically reach performance bottlenecks, such as memory and network contention, with far fewer cores than the total cores on the node. Thus, a few cores can be dedicated for this purpose. Another advantage, compared to core specialization and from a system administration point of view, is that there is no special setup to migrate system processes to the idle cores and isolate these cores from the application.

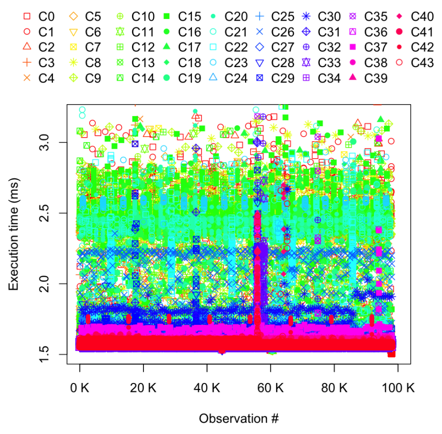
First, we analyze the performance of *Fire* when leaving the first two cores of each socket idle as shown in Figure 3a. Compared to the baseline, we observe an improvement: the number of cores significantly affected is down to one (core 15) and the iteration elapsed times on this core have not increased—the OS is running some of its services on the free cores, which is the desired behavior. Table IV shows an improved maximum execution time and standard deviation compared to the baseline for the most affected core.

Second, Figure 3b shows that leaving the last two cores of each socket free still results in two cores being significantly affected—cores 18 and 1—as is the case for the baseline. Unlike the baseline though, the most affected core shows improvement as shown in Table IV. Further examination shows that Cores 0 and 1 are still being used by the OS to run services even though other cores are free. This finding is consistent with a previous observation that certain services scheduled by the OS have an affinity to the first (few) core(s) of a system [48]. Thus, when choosing cores for the OS, using the first (few) core(s) compared to the last is more effective.

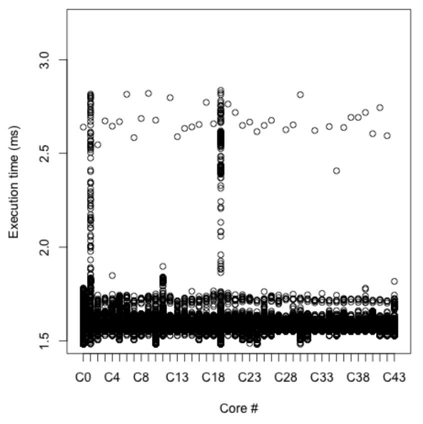
Third, we analyze the free-core strategy on *Ice*. Compared to the baseline, Figure 3c and Table IV show a major improvement. For example, the maximum execution time on



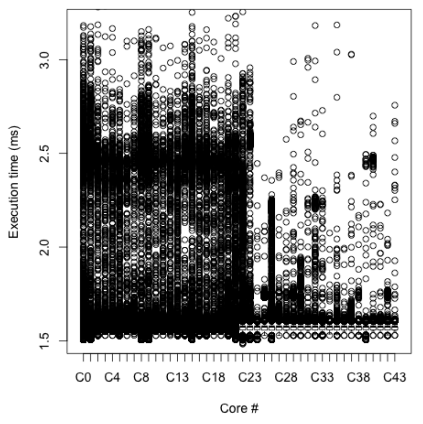
(a) FWQ-MPI on *Fire*: Scatterplot.



(b) FWQ-MPI on *Ice*: Scatterplot.



(c) FWQ-MPI on *Fire*: Boxplot by core.



(d) FWQ-MPI on *Ice*: Boxplot by core.

Fig. 2. Execution time of FWQ-MPI for 100,000 samples per core. All cores recorded samples in parallel. Scatter plots on the top and box-and-whisker plots on the bottom. *Ice* with the IBM software stack is significantly affected by system noise, while *Fire* with TOSS is more resilient.

the most affected core went from 10.6 ms to 3.4 ms. However, the variability in execution time still varies substantially ($\sigma = 0.081$)—*Fire* is significantly better.

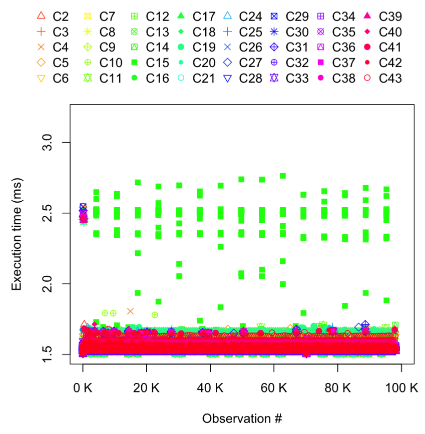
The free-core strategy on the IBM stack mitigates noise noticeably over the baseline, but FWQ performance still varies widely. The free-core strategy on TOSS is more effective when using the first two cores of each socket (rather than the last) due to system affinity to these cores.

D. Core specialization

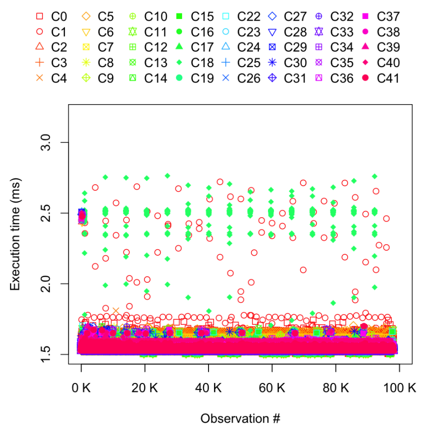
In this configuration, a number of cores are dedicated for running the OS and system processes—*system cores*. Unlike the free-core strategy, this configuration requires system administration tasks to explicitly bind system processes to the system cores, leaving the remaining *user* cores free for application's work. *Fire* and *Ice* use two different strategies for core specialization. On *Ice*, IBM provides two configurations. The first one dedicates a configurable number of cores for system processes and binds those processes to these cores. The second one, as far as we know, is a more comprehensive method moving as much system processing as possible to the system cores [4]. We refer to the first configuration as *CoreSpec* and the second as *CoreSpecBlink*. All of the core specialization configurations described in this section use the first two cores of each socket as the dedicated system cores.

We now describe the core specialization strategy used for *Fire*, which follows a similar strategy found in prior work [8]. Instead of grossly moving as many system processes as possible to the system cores, we determined what processes matter most and selectively moved those to the system cores. Since there are hundreds of system processes running on a compute node, we sorted them by the amount of CPU time each had accumulated. Using the FWQ-MPI benchmark as an indicator, we moved each of these processes to the system cores until we reached a state where noise was significantly reduced—*clean configuration*—as shown in Figure 4a. The resulting processes leading to the clean configuration are shown in Table V. It is important to mention that the clean configuration substantially reduces noise; the most affected core has a standard deviation of $\sigma = 0.011$.

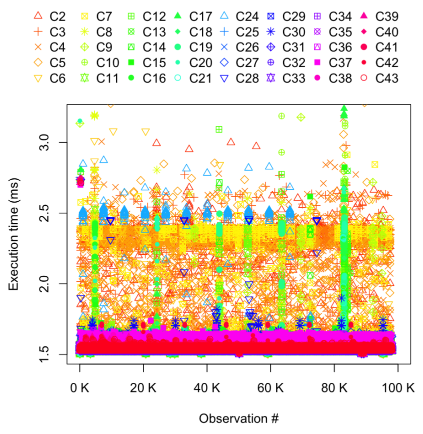
Then, from the selected processes, we determined which



(a) Free cores on *Fire*: Cores 0,1,22,23.

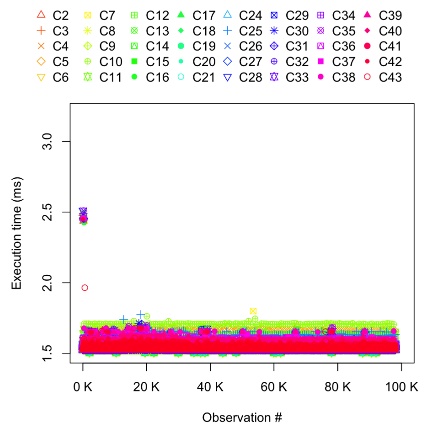


(b) Free cores on *Fire*: Cores 20,21,42-43.

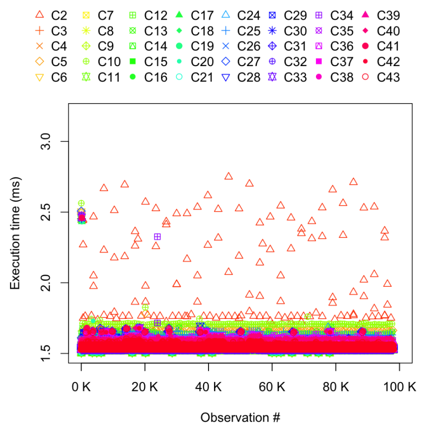


(c) Free cores on *Ice*: Cores 0,1,22,23.

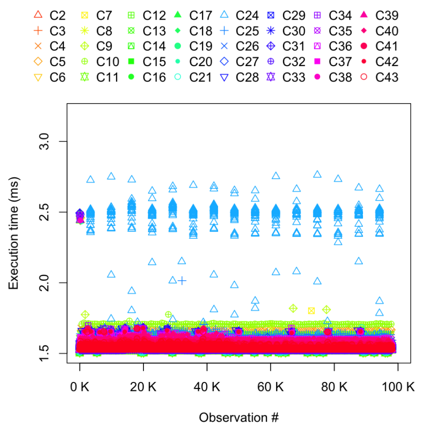
Fig. 3. Execution time of FWQ-MPI for 100,000 samples per core under the free-core strategy. *Ice* has less noise compared to the baseline, but *Fire* is significantly better. When given a choice, the first cores of each socket absorb noise more effectively than the last cores.



(a) Clean configuration.



(b) Clean+snmpd.



(c) Clean+irqbalance.

Fig. 4. Execution time of FWQ-MPI on *Fire* under the core specialization strategy.

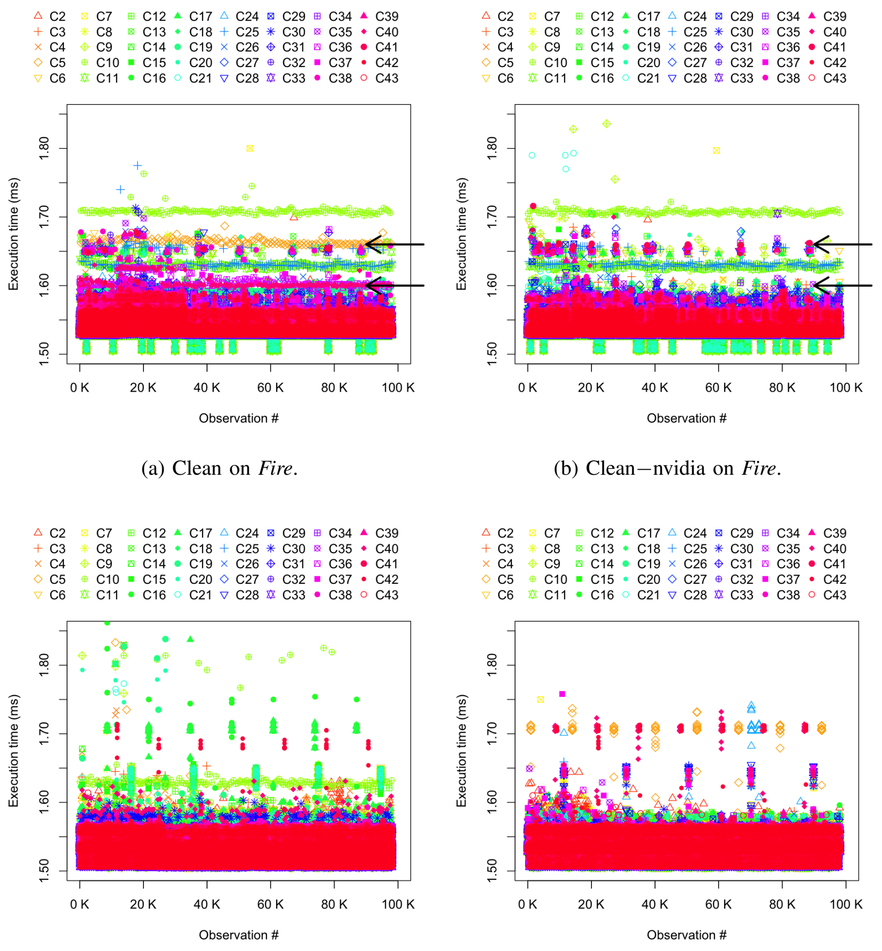
TABLE V
SELECTED PROCESSES FOR CORE SPECIALIZATION BASED ON CPU TIME ACCUMULATED. THE *clean* CONFIGURATION DOES NOT INCLUDE ISOLATING THE *nvidia* PROCESSES.

Processes	Description
`slurm*`	Slurm resource manager compute node daemons.
`snmpd`	Agent servicing SNMP management software requests.
`cerebrod`	TOSS agent for cluster monitoring.
`crond`	Daemon to execute scheduled commands.
`systemd*`	Linux initialization system and service manager.
`ntpd`	Network time protocol daemon.
`irqbalance`	Daemon to distribute hardware interrupts across CPUs.
`nvidia*`	NVIDIA's processes for GPU management.

ones most impacted noise. We started with the clean configuration, which isolates the selected processes to the system cores, and moved one process at a time to the user cores to determine its impact on FWQ-MPI. As shown in Figures 4b and 4c, we found that two processes have a significant impact: *snmpd* and *irqbalance*. As Table IV indicates, irqbalance has a stronger effect on noise, $\sigma = 0.067$, than snmpd, $\sigma = 0.023$. Note that there is an insightful body of work characterizing the sources of noise [10], [11], [39], [49]–[53].

Given that *Ice* and *Fire* are heterogeneous systems with GPUs, we then evaluated the impact of NVIDIA's processes that help manage the GPUs, e.g., *nvidia-persistenced*. We started from the clean configuration and then moved the nvidia processes from the user cores to the system cores. Compared to the clean configuration from Figure 5a, the new setup shows an improved noise signal as shown in Figure 5b—fewer horizontal lines in this configuration as shown by the arrows. It is important to note that these figures use a lower Y-axis range compared to previous plots to visualize the impact. In other words, the impact of the nvidia processes on execution time is much lower than that of snmpd and irqbalance. However, these nvidia processes fire frequently as shown in Figure 5a.

We now compare the core specialization strategies from *Ice* and *Fire* using Figure 5. Starting with *Ice*, on the right-hand side, the Blink configuration reduces noise further compared to



(a) Clean on *Fire*.

(b) Clean−nvidia on *Fire*.

(c) CoreSpec on *Ice*.

(d) Blink on *Ice*.

Fig. 5. Zoomed-in execution time of FWQ-MPI on *Fire* and *Ice* under the core specialization strategy. The noise signal in these configurations is low; to show the differences between them we use a lower y-range than previous figures.

CoreSpec—most of the green data points are eliminated—and both of these configurations show a substantial improvement over the baseline and the free core strategy. For example, as shown in Table IV, the maximum execution time went from 10.6 ms and 3.4 ms (Baseline and FreeCore) to 1.6 ms and 1.7 ms (CoreSpec and Blink). Thus, the core specialization strategy on *Ice* is effective (and necessary) to mitigate noise. *Fire* shows a similar range of execution times as shown by the left-hand side plots, but it is not as effective as *Ice* (2.5 ms maximum time) since only a selected set of processes were isolated into the system cores. The core specialization process on *Fire*, on the other hand, is simpler than *Ice*.

Core specialization is an effective technique to mitigate noise on both the IBM stack and TOSS. IBM's robust strategy shows the best results. At the same time, using a selective approach to isolating system processes yields similar low-noise signatures. Important processes contributing to noise include snmpd and irqbalance.

VII. Assessing messaging workflows capability

Communication at scale is important for application performance. This section evaluates some of the key communication operations by stress-testing the communication stacks of different MPI implementations on both platforms. This evaluation includes a MPI micro-benchmarks comparison between the *Fire* and *Ice* MPI implementations, Spectrum MPI and OpenMPI, leveraging CPUs and GPUs in collective and point-to-point operations.

GPU accelerators are increasing node complexity. They increase the number of locations data can be sent from between nodes, and new transport paradigms, including GPUDirect and GPUDirect async, increase the options of how to send messages. In this section, we investigate how performance varies with messaging location, and the performance and coding tradeoffs presented by new technologies.

Our goal is to compare the different software stacks. To do this we tested many microbenchmarks and present here the most interesting subset relevant to many applications. Benchmarks are used to compare communications stack implementations rather than stress-testing the interconnect itself. After testing, the following MPI Benchmarks were selected from the OSU Micro Benchmark Suite [54] to present:

1) `osu_allgather`: A benchmark testing the collective `MPI_Allgather` operation.
2) `osu_bibw`: A point-to-point, bi-directional bandwidth test using the `MPI_Send` operation.

Collective operations have long presented challenges for MPI implementers. [55]. We chose to benchmark the collective operations `MPI_Allgather` and `MPI_Allreduce`. The `MPI_Allreduce` results are omitted in the interest of brevity and because they demonstrated the same performance pattern.

Point-to-point operations are common in codes that exchange nearest-neighbor data, commonly known as halo-exchanges [56]. Hence, the MPI implementation communications stack is meaningfully utilized while the latency characteristics of the interconnect itself are minimized.

The above benchmarks were run in the following different scenarios:

1) Between CPU/Host-allocated buffers
2) Between GPU device-allocated buffers
3) Between GPU driver-managed ("unified") buffers with CPU-based integrity checking

The three different MPI implementations described in Section IV, Table II were tested. All of the benchmarks run with OpenMPI were built with the GNU C Compiler while the benchmarks run with Spectrum MPI were built with the XL C compiler.

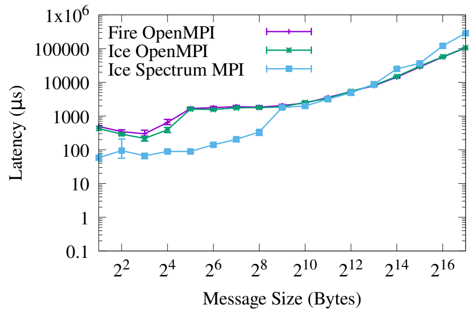
A. Collective Benchmark Results

In all of the collective operation benchmarks, 64 nodes were used. For CPU-based benchmarks, tasks were distributed among the 40 CPUs available for computation. For GPU benchmarks, the ranks were allocated to each of the 4 GPUs on each of the 64 nodes.

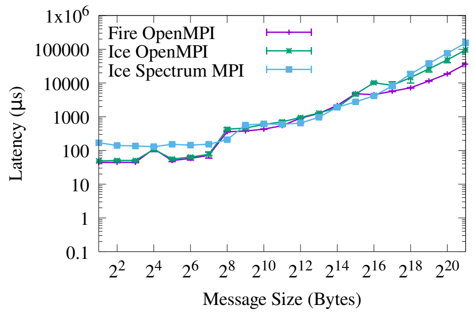
Figure 6a shows Spectrum MPI to have the lowest CPU-CPU latency on *Ice*. However, when GPU buffers are leveraged, OpenMPI has a lower overall latency, as shown in Figure 6b and 6c. In all of the collective benchmarks, OpenMPI outperforms Spectrum MPI for 64 KiB or larger messages. Figure 6b, shows for the GPU device-allocated buffer benchmark *Fire* outperforms both MPIs on *Ice*.

B. Point-to-Point Benchmarks

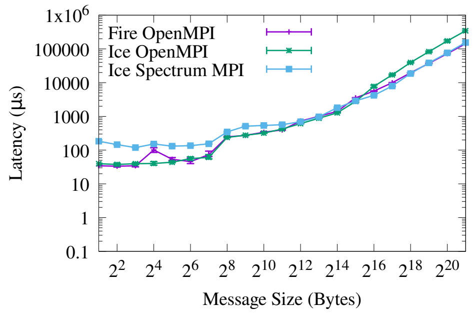
The Bidirectional Bandwidth (`osu_bibw`) micro benchmark is a point-to-point benchmark which mixes throughput and latency characteristics of messaging workflows. The benchmark was run between two nodes placed as closely as possible on each system, limiting the max hop count to 2. While this may not fully replicate real-world scenarios, this was done in order to stress the MPI implementations rather than the interconnects themselves. Furthermore, it provides a controlled environment in which contention from other coresident applications is minimized. The original OSU micro benchmarks were modified by adding the buffer flushing and checking code mentioned earlier and `cudaMemAdvise` hints



(a) CPU/Host-allocated buffers



(b) GPU Device-allocated buffers



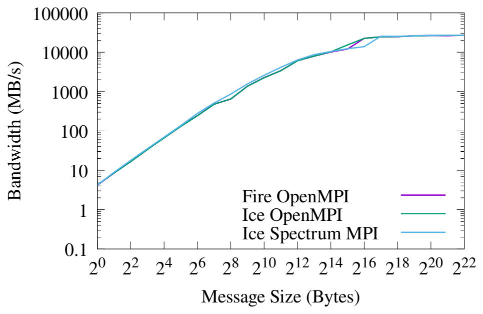
(c) GPU Managed buffers

Fig. 6. An `MPI_Allgather` latency benchmark comparison of different systems and MPI implementations, with different buffer allocation strategies. The source and target buffer allocation strategies always match. These are log-log plots.

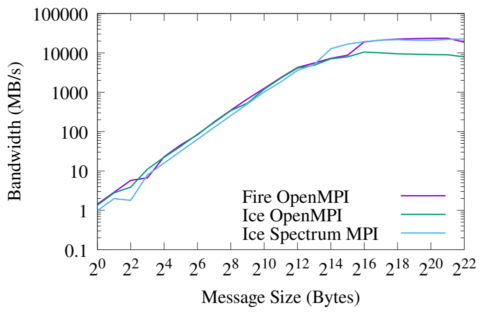
whenever a managed buffer was allocated to push the buffer on to the device. As with the collective benchmarks, Host/CPU, GPU Device and GPU driver-managed buffers were tested.

CPU-to-CPU point-to-point bandwidth was comparable between the different MPI libraries (as shown in Figure 7a). In the GPU device-allocated scenario, OpenMPI running on *Fire* outperformed *Ice*, starting at the 64 KiB threshold, shown in Figure 7b. While Spectrum MPI performed better in the GPU driver-managed buffer scenario, the performance was highly variable across different message sizes.

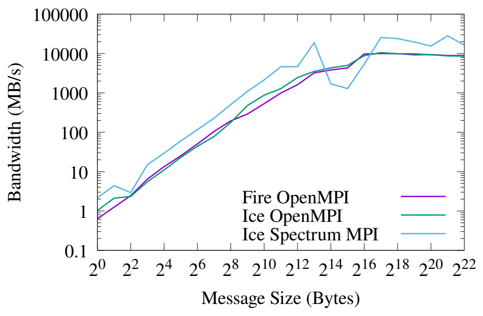
Strong GPU-based collective performance illustrates the benefit of rapidly deploying emerging technologies in *Fire*'s open-source platform. This performance benefit is crucial since the GPU represents the vast majority of available cycles on the hardware underlying *Fire* and *Ice*.



(a) CPU/Host-allocated buffers



(b) GPU Device-allocated buffers



(c) GPU Managed buffers

Fig. 7. An `MPI_ISend + MPI_Barrier` Bidirectional Bandwidth benchmark comparison of different systems and MPI implementations, with different buffer allocation strategies. The source and target buffer allocation strategies always match. These are log-log plots.

VIII. TOSS EFFECTIVENESS ON SCIENTIFIC APPLICATIONS

We evaluate two scientific codes to determine the effectiveness of TOSS relative to the vendor stack. While the benchmark-based evaluation presented in the previous sections is important, ultimately, application runtime or throughput performance—Figure of Merit (FOM)—is the key indicator. From the Exascale Computing Project [57], we employ a proxy application and a full-fledged application that simulates ground motion due to earthquakes. We chose these codes because they stress different resources: CPUs vs GPUs. Finally, we use the open-source OpenMPI on TOSS and the vendor-proprietary Spectrum MPI on the IBM stack.

SW4 solves the seismic wave equations on Cartesian and curvilinear grids [58]–[60]. In the RAJA port of SW4 [61], the solve phase accounts for 99.6% of the runtime with all of the computation taking place on GPUs while four CPU cores per node (one task per GPU) are used for kernel launches, data transfers, message passing, and I/O. The solve phase of SW4 was built with NVCC on both machines and linked using GCC on TOSS and XL on IBM's stack (see Table II). For our tests a problem with 107,518,000 grid points per node was used.

The measured runtimes in Table VI show comparable performance of the solve phase between TOSS and the vendor stack. Since most of the CPU cores are not used, system noise effects on SW4 runtime are minimal, but present in the CoreSpecBlink configuration. Outside this configuration, TOSS shows slightly better performance, but it is unclear why. We had to use 69 nodes compared to 72 nodes with the vendor stack—three nodes experienced transient hardware failures and the differences could be due to problem decomposition or something about the job rather than the software stack.

TABLE VI
SW4 RUNTIME (SECS) ON *Ice* AND *Fire* WITH VARIOUS NOISE STRATEGIES.

	Baseline	Core Specialization		Nodes
Ice	84.22	CoreSpec 84.05	CoreSpecBlink 80.80	72
Fire	82.82	CoreSpecClean 82.70		69

AMG is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [62]. We use problem 1, which uses conjugate gradient preconditioned with AMG to solve a Laplace-type problem on a 3D grid of $1600x800x500$ with a 27-point stencil. This is a CPU-intensive code and was chosen to represent a worst case scenario for TOSS. AMG is highly sensitive to noise due to frequent AllReduce operations, and Spectrum MPI on *Ice* benchmarks better for this operation than OpenMPI on *Fire*. We built AMG using GCC on both systems and ran it with 40 tasks per node (one per core) on 8, 16, 32, and 64 nodes.

Figure 8 shows the mean FOM (higher is better) of AMG on both configurations. The FOM trend is the same across number of nodes, thus, we focus on the 64-node case. With the baseline configuration, *Fire* is 5.3% better than *Ice*, while with core specialization *Ice* is better: Compared to CoreSpecClean on *Fire*, CoreSpec is 2.5% better and CoreSpecBlink is 4.5% better. In all cases run to run variation was less than 1% of runtime.

The FOM trends are consistent with the noise and MPI measurements from the FWQ-MPI and OSU benchmarks discussed in Section VI and Section VII: IBM's comprehensive strategy performs better, while TOSS's selective and easy-to-setup configuration provides a competitive approach. Finally, while noise was less than 1% for all runs, the best noise mitigation strategies on both systems drove this down to less than 0.1%, which provides a better base for large-scale runs.

TOSS provides comparable performance to the vendor stack for a CPU-intensive code (within 5%) that is a worst case scenario due to frequent AllReduce operations and noise sensitivity. For a GPU intensive-code, the gap was smaller (within 3%). Both of these represent important codes for the US Department of Energy.

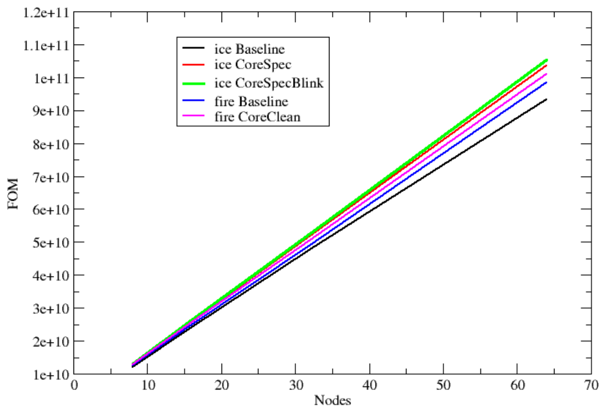


Fig. 8. AMG FOM on *Ice* and *Fire* comparing noise mitigation strategies.

IX. Conclusion and Recommendations

In this paper we analyzed the system deployment challenges and benefits, application and microbenchmark performance impacts, user support advantages, and user workflow advantages of using the TOSS simulation environment on an advanced technology system. Using a vendor-supplied software stack comes with a set of tradeoffs versus using a customer-supplied stack. Vendor stacks are often, but not always, better optimized for their their hardware. We found mixed results for key microbenchmarks, and a small 2-5% advantage for the vendor stack on applications. Customer-supplied stacks require more customer effort for configuration, integration, and testing to deploy. However, a customer-deployed stack allows uniformity across machines within a computer center. Uniformity benefits system administrators, user environment support staff, and users who now only need to learn or support a single environment.

While there are significant savings from a uniform environment, there are costs as well. Configuring middleware libraries, such as MPI, to get good performance takes significant effort. Determining how to integrate various vendor-supplied software stacks with a customer stack is time-consuming and can lead to dead ends. In addition, some performance features may not be available at certain times. However, using one's own software stack allows new features to be integrated more quickly than using the vendor-supplied stack, which often only has a few major releases per year.

Moving a customer-supplied stack onto a system that is already deployed with a vendor stack for production purposes is not the goal of this paper. Instead, we provide empirical evidence that a customer-supplied stack is viable on advanced technology systems, which are now composed of commodity technology nodes. This transition gives TOSS a timely opportunity to extend its scope to leadership-class systems that, in the past, were operated under vendor-specific software environments. We judge the advantages of moving in this direction significant for large computer centers with many machines. Considering the productivity advantages, many users have expressed a preference for a familiar, commodity, and uniform environment, even at a small performance loss. As a computer center, building-in a plan from the beginning to either run the center's own stack or have the option to will make this process easier. Our recommendations for HPC centers, based on the lessons we learned from evaluating TOSS on an advanced technology system, include the following:

- Include in a request for proposals (RFP) details of the customer-supplied stack and what a vendor needs to do to make it work.
- Provide mechanisms for the system vendor to ask for non-recurring engineering (NRE) funding to make sure various pieces of their software work with the customer environment.
- Make certain support contracts for the system cover the need to continue to update and integrate new software releases and features.
- Retrain support and system administration staff to better serve this different environment.

While some of these requirements may impose new work on a vendor, others involve shifting effort from the vendor to the HPC center, which may reduce system costs. In addition, while there will be a one-time extra cost on computer center support staff from the switch, the long-term savings of user effort and staff time should more than make up for the slight performance loss caused by moving from a vendor stack, which should improve over time (as a result of open-source community-driven components).

In this paper, we demonstrated the viability of the TOSS simulation environment on an advanced technology system. Our results show comparable performance with the vendor-based software stack. While not a forgone conclusion since more testing is needed, we believe the shift to a commodity-based software stack on advanced technology systems will be a large productivity win.

We plan to continue to work on TOSS on advanced technology systems. The next steps in this process involve demonstrating TOSS at scale on *Lassen* and *Sierra*. Tests with more complex and diverse applications, including I/O, will be used to show the stack is capable of full production runs on one of the largest machines in the world.

Acknowledgment

We would like to thank Ben Casses, Jim Silva, and Olaf Faaland from LLNL and Ben Woodard from Red Hat for their contributions to TOSS enabling the work presented in this paper. We are also thankful to Py Watson for his help configuring the *Lassen* system during dedicated system access. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-809186.

References

[1] Lawrence Livermore National Laboratory, "TOSS: Speeding up commodity cluster computing," accessed 2020-05-01. [Online]. Available: https://computing.llnl.gov/projects/toss-speeding-commodity-cluster-computing

[2] CORAL: Collaboration of Oak Ridge, Argonne and Livermore National Laboratories, "Draft CORAL build statement of work," Office of Science and the National Nuclear Security Administration's Advanced Simulation and Computing (ASC) Program, U.S. Department of Energy, RFP No. B604142, LLNL-PROP-636244, Dec. 2013.

[3] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2862.

[4] IBM Corporation, "IBM CSM: Cluster systems management," 2018, accessed 2020-01-10. [Online]. Available: https://cast.readthedocs.io/en/cast_1.5.x/csmd/index.html

[5] A. computing, "Torque resource manager." [Online]. Available: http://docs.adaptivecomputing.com/torque/6-0-0/Content/topics/torque/2-jobs/monitoringJobs.htm

[6] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *Cluster computing and Grid 2005 (CCGrid05)*. Cardiff, United Kingdom: IEEE, 2005. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00005106

[7] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for large HPC centers," in *43rd International Conference on Parallel Processing Workshops, ICPPW 2014, Minneapolis, MN, USA, September 9-12, 2014*, 2014, pp. 9–17. [Online]. Available: https://doi.org/10.1109/ICPPW.2014.15

[8] E. A. León, I. Karlin, and A. T. Moody, "System noise revisited: Enabling application scalability and reproducibility with SMT," in *International Parallel and Distributed Processing Symposium*, ser. IPDPS'16. Chicago, IL: IEEE, May 2016.

[9] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *International Parallel & Distributed Processing Symposium*, ser. IPDPS'11. Anchorage, AK: IEEE, May 2011.

[10] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. New Orleans, LA: ACM/IEEE, Nov. 2010.

[11] K. B. Ferreira, R. Brightwell, and P. G. Bridges, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'08. Austin, TX: IEEE/ACM, Nov. 2008.

[12] C. Zimmer, S. Atchley, R. Pankajakshan, B. E. Smith, I. Karlin, M. L. Leininger, A. Bertsch, B. S. Ryujin, J. Burmark, A. Walker-Loud, M. A. Clark, and O. Pearce, "An evaluation of the CORAL interconnects," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'19. Denver, Colorado: ACM, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356166

[13] T. Groves, Y. Gu, and N. J. Wright, "Understanding performance variability on the Aries dragonfly network," in *International Conference on Cluster Computing*, ser. Cluster'17. IEEE, Sep. 2017, pp. 809–813.

[14] E. A. León, I. Karlin, A. Bhatele, S. H. Langer, C. Chambreau, L. H. Howell, T. D'Hooge, and M. L. Leininger, "Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'16, 2016, pp. 909–920.

[15] Red Hat, "Red Hat Enterprise Linux," accessed 2020-03-01. [Online]. Available: https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[16] ——, "EPEL," accessed 2020-03-01. [Online]. Available: https://fedoraproject.org/wiki/EPEL

[17] Seagate Technology, LLC, "Lustre," accessed 2020-03-01. [Online]. Available: http://lustre.org/

[18] Lawrence Livermore National Laboratory, "ZFS on Linux," accessed 2020-03-01. [Online]. Available: https://zfsonlinux.org/

[19] SyLabs.io, "Singularity," accessed 2020-03-01. [Online]. Available: https://sylabs.io/singularity/

[20] R. Priedhorsky and T. Randles, "Charliecloud: unprivileged containers for user-defined software stacks in hpc," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC'17. IEEE/ACM, 2017.

[21] Mellanox Technologies, "Linux infiniband drivers," accessed 2020-03-01. [Online]. Available: https://www.mellanox.com/products/infiniband-drivers/linux/mlnx_ofed

[22] Lawrence Livermore National Laboratory, "ConMan: The console manager," accessed 2020-03-01. [Online]. Available: https://github.com/dun/conman

[23] ——, "MUNGE: MUNGE Uid 'N' Gid Emporium," accessed 2020-03-01. [Online]. Available: https://github.com/dun/munge

[24] ——, "pdsh," accessed 2020-03-01. [Online]. Available: https://github.com/chaos/pdsh

[25] ——, "powerman," accessed 2020-03-01. [Online]. Available: https://github.com/chaos/powerman

[26] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "Lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *International Conference for High Performance Storage, Networking, and Analysis*, ser. SC'14. IEEE/ACM, 2014.

[27] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[28] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: Bringing order to HPC software chaos," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'15. Austin, Texas: ACM, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807623

[29] Fermilab, "Scientific Linux," accessed 2020-03-01. [Online]. Available: https://www.scientificlinux.org/

[30] Cray, "Cray Linux Environment," accessed 2020-03-01. [Online]. Available: https://pubs.cray.com/content/S-2393/CLE\%207.0.UP01/xctm-series-system-administration-guide

[31] SUSE Group, "SUSE Linux Enterprise Server," accessed 2020-03-01. [Online]. Available: https://www.suse.com/products/server/

[32] Linux Foundation, "OpenHPC: Community building blocks for HPC systems," 2018. [Online]. Available: http://www.openhpc.community/

[33] N. Papadopoulou, L. Oden, and P. Balaji, "A performance study of UCX over InfiniBand," in *International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID. IEEE/ACM, 2017, pp. 345–354.

[34] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow, "PAMI: A parallel active message interface for the Blue Gene/Q supercomputer," in *International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 763–773.

[35] E. A. León, "mpibind: A memory-centric affinity algorithm for hybrid applications," in *International Symposium on Memory Systems*, ser. MEMSYS'17. Washington, DC: ACM, Oct. 2017.

[36] ——, "Mapping MPI+X applications to multi-GPU architectures: A performance-portable approach," in *GPU Technology Conference*, ser. GTC'18, San Jose, CA, Mar. 2018.

[37] E. A. León and M. Hautreux, "Achieving transparency mapping parallel applications: A memory hierarchy affair," in *International Symposium on Memory Systems*, ser. MEMSYS'18. Washington, DC: ACM, Oct. 2018.

[38] T. B. Tabe, J. P. Hardwick, and Q. F. Stout, "Statistical analysis of communication time on the IBM SP2," *Computing Science and Statistics*, vol. 27, pp. 347–351, 1995.

[39] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Conference on Supercomputing*, ser. SC'03. Phoenix, AZ: ACM/IEEE, Nov. 2003.

[40] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Conference on Supercomputing*, ser. SC'03. Phoenix, AZ: ACM/IEEE, Nov. 2003.

[41] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, “Leveraging the Cray Linux Environment core specialization feature to realize MPI asynchronous progress on Cray XE systems,” in *Cray User Group*, ser. CUG’12, Stuttgart, Germany, Apr. 2012.

[42] T. Kato and K. Hirai, *K Computer*. Singapore: Springer Singapore, 2019, pp. 183–197. [Online]. Available: https://doi.org/10.1007/978-981-13-6624-6_11

[43] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira, “Designing and implementing lightweight kernels for capability computing,” *Concurrency and Computation: Practice and Experience*, vol. 21, no. 6, pp. 793–817, Apr. 2009.

[44] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, “Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene’s CNK,” in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC’10. New Orleans, LA: ACM/IEEE, Nov. 2010.

[45] P. De, V. Mann, and U. Mittal, “Handling OS jitter on multicore multithreaded systems,” in *International Symposium on Parallel & Distributed Processing*, ser. IPDPS’09. Rome, Italy: IEEE, May 2009.

[46] T. Jones, “Linux kernel co-scheduling for bulk synchronous parallel applications,” in *International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS’11, Tucson, AZ, May 2011.

[47] “CORAL benchmark codes,” December 2013. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[48] E. Rosenthal, E. A. León, and A. T. Moody, “Mitigating system noise with simultaneous multi-threading,” in *International Conference for High Performance Computing, Networking, Storage and Analysis; Research Poster*, ser. SC’13. Denver, CO: ACM/IEEE, Nov. 2013.

[49] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, “The influence of operating systems on the performance of collective operations at extreme scale,” in *International Conference on Cluster Computing*, ser. Cluster’06. IEEE, Sep. 2006.

[50] P. De, R. Kothari, and V. Mann, “Identifying sources of operating system jitter through fine-grained kernel instrumentation,” in *International Conference on Cluster Computing*, ser. Cluster’07. Austin, TX: IEEE, Sep. 2007.

[51] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, “The impact of system design parameters on application noise sensitivity,” in *International Conference on Cluster Computing*, ser. Cluster’10. Heraklion, Greece: IEEE, Sep. 2010.

[52] D. Doerfler, M. Epperson, J. Ogden, C. T. Vaughan, and M. Rajan, “Application performance on the Tri-Lab Linux capacity cluster-TLCC,” *International Journal of Distributed Systems and Technologies*, vol. 1, no. 2, Apr. 2010.

[53] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, “Extreme scale computing: Modeling the impact of system noise in multicore clustered systems,” in *International Symposium on Parallel & Distributed Processing*, ser. IPDPS’10. Atlanta, GA: IEEE, Apr. 2010.

[54] D. K. Panda. (2019, aug) OSU microbenchmarks. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[55] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, “MPI on a million processors,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 20–30.

[56] D. De Sensi, S. Di Girolamo, and T. Hoefler, “Mitigating network noise on dragonfly networks through application-aware routing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356196

[57] Exascale Computing Project, “ECP proxy apps suite, release 3.0,” accessed 2020-02-01. [Online]. Available: https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/

[58] B. Sjögreen and N.A.Petersson, “A fourth order accurate finite difference scheme for the elastic wave equation in second order formulation,” *J. Scient. Comput.*, vol. 52, pp. 17–48, 2012.

[59] N.A.Petersson and B. Sjögreen, “Wave propagation in anisotropic elastic materials and curvilinear coordinates using a summation-by-parts finite difference method,” *J. Comput. Phys.*, vol. 229, pp. 820–841, 2015.

[60] ——, “Super-grid modeling of the elastic wave equation in semi-bounded domains,” *Commun. Comput. Phys.*, vol. 16, pp. 913–955, 2014.

[61] R. Pankajakshan, P. Lin, and B. Sjogreen, “Porting a 3D seismic modeling code (SW4) to CORAL machines,” *IBM Journal of Research and Development*, 2019.

[62] “Algebraic multigrid benchmark,” GitHub repository, 2018. [Online]. Available: https://github.com/LLNL/AMG

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

(1) We ran FWQ-MPI, a modified version of FWQ available at https://asc.llnl.gov/CORAL-benchmarks/#ftq, on LLNL's Lassen supercomputer with both TOSS and with the IBM software stack. We used several noise mitigation strategies including no noise-mitigation (baseline), the free-core strategy, and core specialization, as described in the paper. We ran 100,000 samples per core and reported the min, median, max, and standard deviation. In addition to scatter plots, we included two representative box-and-whisker plots.

(2) We ran the OSU Micro-Benchmarks, available at http://mvapich.cse.ohio-state.edu/benchmarks/ with minor modifications so that the receive buffers are checked and refreshed after each transfer. In order to check buffers, a simple buffer comparison function was added to `/util/osu_util_mpi.c` and the corresponding header file. The benchmark source code files in `/mpi/` were then modified to call this function outside of their timed loops. For buffer refreshing, after transfers, and outside the timed loops, extra calls were made to `set_buffer_pt2pt(...)` to disable caching. For point-to-point communications and for each message size, the benchmarks execute 100 runs and report averages. For collective communications and for each message size, the benchmarks execute 1000 runs for message sizes up to 16 KiB and then 100 runs after that. The graphs include error bars, but there were such small margins that the error bars appear as if they are ordinary markers (they are displayed in the legends, though).

(3) We ran the RAJA version of SW4 available at https://github.com/geodynamics/sw4.git on Lassen with both TOSS and the IBM software stack. In both cases, NVCC was used to compile RAJA code with GCC and XLC as the host compilers, respectively. We ran each experiment 3 times and reported averages. Run-to-run variation was less than 1% of runtime.

(4) We ran the AMG proxy-application, available at https://github.com/LLNL/AMG.git on Lassen with both TOSS and the IBM software stack. AMG was compiled using the default options in the Makefile using GCC. We ran each experiment 10 times and reported averages. Run-to-run variation was less than 1% of runtime.

ARTIFACT AVAILABILITY

Software Artifact Availability: There are no author-created software artifacts.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Each compute node has two IBM Power9 processors coupled with two NVIDIA Volta GPUs per processor. There are 22 cores per Power9 processor and 4 hardware threads per core (SMT- 4). Each node has 256 GB of DDR4 memory and 16 GB of HBM memory per GPU. A Power9 processor has up to 170 GB/s peak DDR4 bandwidth and each GPU 900 GB/s peak bandwidth to local HBM.

Operating systems and versions: TOSS 3.5-4: Linux 4.14.0-115.17.1.1chaos.ch6a.ppc64le, RHEL 7.6, MOFED 4.5, NVIDIA 440.64.00 and GDRCopy, Slurm 19.05.5. IBM stack: Linux 4.14.0-115.10.1.1chaos.ch6a.ppc64le, RHEL 7.6, MOFED 4.5, NVIDIA 418.87.00, and GDRCopy, IBM LSF 10.1, IBM CSM 1.6.0.

Compilers and versions: TOSS: GCC 7.3.1, NVCC 10.2.89. IBM stack: IBM XL 16.1.1, GCC 7.3.1, NVCC 10.1.243.

Applications and versions: FWQ-MPI based on FWQ 1.1. OSU Micro-Benchmarks 5.6.2, modified. SW4 (RAJA branch) commit f0efb7712f310035a4977c288274073f6ab96f36. AMG commit 3ada8a128e311543e84d9d66344ece77924127a8.

Libraries and versions: TOSS: OpenMPI 4.0.3 with UCX 1.7.0. IBM stack: IBM Spectrum MPI 10.3.0 with PAMI, OpenMPI 4.0.3 with UCX 1.7.0.

Key algorithms: Two simulation environments on the same machine: TOSS and the vendor-based stack. Several mitigation strategies on each simulation environment. On the IBM vendor stack: IBM CSM with '-core_isolation 0,' '-core_isolation 2,' and '-core_isolation 2 -alloc_flags "cpublink autonumaoff"'. On TOSS: various derivatives of 'tuna -t slurm* snmpd cerebrod crond systemd* ntpd irqbalance -c0-7,88-95 -m -P'

Input datasets and versions: FWQ with parameters '-n100000 -w4096'. OSU Micro-Benchmarks Collectives Parameters '-m 2:4194304 -x 20 -f' with '-d' and '-r' arguments used with appropriate parameters for GPU collectives. OSU Micro-Benchmarks Point-to-Point Parameters 'D D' and 'M M' were used for the point-to-point cases. The range of message sizes was set in the constants in the code before compilation. AMG with options '-n 200 100 100 -P 8 8 5 -problem 1' for the 8 node case. The '-P' task-topology values differ for increasing node counts while the rest of the options are kept constant. SW4 with parameters '69.in'