Full-stack Approach to HPC testing

Pascal Jahan Elahi , Craig Meyer
Pawsey Supercomputing Research Centre, Kensington, WA, Australia

***Abstract*—A user of an High Performance Computing (HPC) system relies on a multitude of components, both on the user-facing side, such as `modules`, and lower-level system software, such as Message Passing Interface (MPI) libraries. Thus, all these different aspects must be tested to guarantee an HPC system is production ready. We present here a suite of tests that cover this larger space, which not only focus on benchmarking or sanity checks but also provide some diagnostic information in case failures are encountered. These tests cover the job scheduler, here SLURM, the MPI library, critical for running jobs at scale, and GPUs, a vital part of any energy efficient HPC system. Some tests were critical to uncovering a number of underlying issues with the communication libraries on a newly deployed HPE-Cray EX Shasta system that had gone undetected in other acceptance tests. Others identified bugs within the job scheduler. The tests are implemented in a REFRAME framework and are open source.**

MPI SLURM Software Stack Benchmarks Automation Regression Tests

I. Introduction

Characterizing the environment of an HPC system is critical in providing a quality service to end users. Typically this is performed by running regression tests across various aspects of the system regularly over an extended period of time, as well as acceptance testing. Common areas to test, and of particular relevance to this work, include data exchange and communication operations (e.g. MPI implementation), and the job scheduler being used on the system (e.g. SLURM).

Message Passing Interface standard (MPI, 1) is the de facto standard defining communication operations for exchanging data in parallel computing environments. The MPI standard defines multiple operations for various communication purposes and provides bindings for C and Fortran programming languages. MPI is critical to all supercomputing environments. At the Pawsey Supercomputing Research Centre, well over 90% of the compute cycles are consumed by multi-node jobs using this standard.

Job schedulers are another vital component of HPC systems, responsible for distributing the available computing resources amongst the jobs requested by users. The SLURM batch system [2] is a prominent scheduler, used by many of the top supercomputers in the world, including on our systems at Pawsey. They are responsible for managing resources, scheduling, and executing jobs. As such, they are crucial to the quality of service provided to users. A poor-performing job scheduler could leave much of the system resources unused or leave user jobs in a queue, waiting to be executed, for extended periods of time. Testing the entire functionality of a job scheduler would be a massive, if not impossible, undertaking, but a suite of tests to check the core, basic functionality, would be important in characterizing its performance and diagnosing potential issues.

At Pawsey, we provide a specific set of packages, built using SPACK [3] accessed through the `lmod` module system [4], or provided as containers-as-modules through `shpc` [5]. SPACK, although ideal for building a package and all it's dependencies, does not test the result, hence the need to test user access to software and check for desired functionality.

To this end, we have developed a suite of tests, which are now a part of our regular regression and acceptance testing at Pawsey, designed to run:

MPI:	Contains the Ohio State University Micro-Bencmarks (OMB, 6) along with specific more complex tests that contain a large variety of communication patterns
SLURM:	Covers areas of resource management (including memory, cores, threads), accounting and billing, and affinity at the process and thread levels for both CPU and GPU.
Software:	Checks SPACK installations provides desired functionality, no missing external dependencies, and also checks modules produced.

All tests also provide detailed logging (e.g. node state, memory utilisation, available memory on nodes, CPU core affinity) in order to bypass the need of a debugger and profiler, which can limit the number of processes on which the test is run. These tests have been integrated into a REFRAME [7] framework and are freely available at [8]. REFRAME is a Python framework designed for running benchmarking and regression tests on HPC systems that abstracts away the low-level details of a test, such as system and environment configuration and setup. The REFRAME framework allows one to separate test logic from system setup and configuration. This framework allowed us to write the sort of tests we wanted with the desired functionality, scope, and flexibility.

Our paper is organized as follows. In the first section, we describe our test design, providing motivation by way of the issues encountered by Pawsey staff and researchers using Pawsey systems. We discuss how these tests are critical to properly characterize an HPC environment, with some example results from a selection of tests. We end with a summary of the tests and key conclusions.

*correspondence: `pascal.elahi@pawsey.org.au`

II. Test Design

A. Why standard acceptance tests were not enough

Our system had passed a (initially thought to be rigorous) suite of acceptance tests, containing HPL, a wide variety of OMB tests, production codes like LAMMPS, amongst others. However, these tests stressed a limited number of MPI communication patterns. Moreover, they are designed to benchmark a system rather than stress a system while providing diagnostic information.

Initially, we did not have any unit tests that could replicate the sheer number of `pt2pt` messages some production codes place when loading input data and running an MPI domain decomposition using a large number of MPI processes. Nor did OMB replicate all the varied communication patterns used by codes running on our systems. This pushed us to delve into the source code of software encountering issues and develop a variety of unit tests.

Similarly, regarding SLURM tests, our initial standard acceptance tests were largely limited to testing whether SLURM was operational and performing its primary functions. Tests of correctness and robustness were largely non-existent. Some issues with SLURM were only seen when certain types of job requests were submitted by users (e.g. specific types of resources requests or varying levels of parallelism). Moreover, inaccuracies in SLURM's automatic resource calculations are often not evident until job accounting is observed or resource-hungry user jobs are being run.

These limitations, combined with the need for increased automation and flexibility, led us to develop a suite of new tests. These tests both target some of the specific MPI and SLURM issues we encountered and solve the aforementioned limitations in our already existing test suite. Additional functionality included in our tests (detailed below) increases their utility compared to standard tests and make them more useful to the HPC community as a whole.

In the rest of this section we briefly outline some of the test design decisions we made which we feel are the most impactful and beneficial compared to standard tests.

B. Detailed logging

We have included as standard in most of our tests (all MPI tests) detailed logging to aid in diagnosing and triaging issues. An external library we developed provides detailed memory usage, affinity reporting, and timing statistics. This library is used in the majority of our tests, with the memory usage and affinity reporting being particularly useful. It provides the functionality of traditional debuggers and profilers, except it is embedded in the code and tests themselves, rather than run separately. The MPI tests we developed also make use of polling the memory state and the kernel ring buffer of the nodes before and after the MPI-enabled job is run in the job submission script.

These logs proved to be invaluable in our testing. The memory and affinity information recorded proved key to investigating some of the issues we encountered. Unexpected memory footprints were immediately identifiable, as was poor affinity which could adversely affect performance. Some of the MPI issues we encountered were not actually caused by our MPI implementation, but rather completely different causes, which was able to be identified through error messages in the `dmesg` node health logs (e.g. node-specific issues, system-wide configuration error).

C. Stressing the system with MPI

A key difference in our MPI tests compared to standard acceptance tests is the scale of the MPI communication being performed. We wanted to have tests which mimicked typical user workflows on our system. Therefore, we needed tests with many processes, operating across many nodes, dealing with large quantities of data, and executing the same types of MPI communications that are common in user codes. Several of the issues in our MPI implementation were only discovered when we started testing large-scale MPI jobs, far more demanding on the system and implementation than standard acceptance testing.

D. Extended scope of SLURM testing

We had SLURM tests as part of our regression testing suite already, however, those were mostly limited to checking that SLURM was operational and functioning at the most fundamental level. However, during the initial months of Setonix service we noticed some inconsistencies in SLURM's behaviour. This led us to investigate further and come to the conclusion that we needed tests which exposed more aspects of SLURM.

Rather than check for core basic functionality, many of our new SLURM tests are sanity checks that a given job request leads to a job that has the correct environment configuration, resource limitations, and billing. These tests have allowed us to quickly pick up subtle issues that would otherwise go undetected, since many jobs can run apparently fine even with these problems, especially if users aren't precise with their resource requests. They have also allowed us to see when some aspect of behaviour has changed, perform follow-up investigative testing, and either implement a fix or workaround ourselves, or submit a ticket to schedmd.

E. Expanded user experience testing

Initially, our tests focused on the user experience consisted of checks of a specific set of modules and applications. Though these tests were useful, they were incomplete and required updates as application versions changed. To improve automation and completeness of tests of the software stack and the modules that provide access to users, we have improved the integration of our tests with SPACK [3], our package manager of choice. In this way, all packages and modules are checked to see if they provide the desired basic functionality.

III. Issues

We provide here a brief summary of the issues encountered by users and staff running MPI-enabled codes and using

SLURM on a newly commissioned HPE Cray HPC system, Setonix, which debuted at 15 in the Top 500 list [9] and 4 in the Green 500 list [10]. Although some of the issues discussed below are specific to the deployment of Phase-1 Setonix[1] and its associated Test Deveolpment System (TDS) Joey[2][3], the general issues could be encountered in any MPI deployment and SLURM. It is important to note that Phase-1 Setonix passed acceptance tests, which consisted of MPI-enabled production codes like LAMMPS [11] and benchmarking codes like HPL [12] and OMB, as well as SLURM tests. Yet there were issues in both the MPI deployment and SLURM that went undetected, speaking to the need for further testing. Many of our initial MPI issues were solved with an upgrade to our libfabric library, however the SLURM issues persisted through several different version upgrades[4] and through both Setonix Phase-1 and Phase-2.

A. MPI Issues

The primary issues encountered with MPI-enabled codes were:

Mem-Leak: Multi-node jobs with many MPI processes and large message sizes (sent across nodes) were crashing with a variety of reported errors: so-called bus errors; generic SLURM kill errors; out-of-memory (OOM) errors; errors reporting address issues with `xpmem` library; or errors referencing Open Fabrics Interface (OFI) library.

Mem-Reduced: Idle nodes would slowly have decreasing amounts of available memory after running a number of MPI jobs, regardless of whether these jobs ran successfully or not. These jobs appeared to leave a permanent memory footprint even after the code had completed executing, which slowly decreased the available memory on a node over time.

Scaling: Tests of several codes and software showed that `pt2pt` communication did not scale well past two nodes. The scaling was sufficiently poor that even a much older system outperformed Setonix by factors of 5 or greater when using $\gtrsim 96$ cores across multiple nodes. If processes were restricted to communicate only with neighbouring processes, the observed scaling was far better, so only codes in which processes communicated with large fractions of the communicator would be affected.

We note that almost all of these issues had the same root cause, bugs in the `libfabric` library. However, determining the root cause required the development of specific unit tests that reproduced the communication patterns of real-world codes as well as being able to stress the network.

B. SLURM Issues

The primary issues we encountered with SLURM were:

Memory-Calc: The automatic memory calculation SLURM performs when using `–mem` or `–mem-per-cpu` options is not correct, and SLURM will reject some jobs which it mistakenly believes are asking for too much memory.

Affinity: The affinity of processes and threads is not optimal in shared-node access, at times looking to be entirely random.

Billing: Users run jobs under an account and that account is charged accordingly for the resources they consume. However, the billing calculated by SLURM was found to be incorrect in certain scenarios.

Config: Node-level settings set by the SLURM configuration file were found to change suddenly and unexpectedly. This did not occur as part of a reboot, but while the nodes were still operating.

All of these issues are relatively subtle and, therefore, it is understandable that they were not picked up in initial acceptance testing. Most of them have no direct impact on the ability of most jobs to run. The primary consequences were bad affinity adversely affecting performance in some circumstances, certain valid job requests being rejected, and billing miscalculations in certain job configurations leading to accounts being charged the wrong amount of service units.

IV. Tests

A. MPI tests

In this section we detail a of the key tests we developed to either track or aid in diagnosing/fixing the MPI issues outlined in the previous section. We also expand on the results from our tests and how those assisted in determining the root problem at hand. The solutions and/or workarounds for each issue are also discussed.

All of our MPI tests follow the same general structure. Each test starts with a build phase where any required source code is compiled, then the primary execution phase occurs, which includes a node health check before and after code execution and in-built profiling, then finally sanity and/or performance checks are performed to determine whether the test succeeded or failed.

1) Mem-Leak: : Multi-node jobs with many MPI processes with large message sizes and internode communication were crashing with numerous different reported errors during Phase-1 Setonix. The error messages were varied and a single code could encounter all of them if run multiple times, so initially, it was not obvious what was causing the wide

[1] Setonix is a HPE Cray EX system and is deployed in two phases: Phase-1 mainly consisted of 512 compute nodes with 2 AMD Milan CPUs; Phase-2 involves upgrades to software as well as an additional 1024 AMD Milan nodes and 192 GPU nodes with 4 AMD MI250x GPUs and AMD Trento CPUs.

[2] Joey has 12 AMD Milan compute nodes.

[3] Phase-1 Setonix and Joey were deployed running Cray Shasta < 1.7, used Mellanox Network Interface Cards (NICs), `cray-mpich`$<$ 8.1.12, and `libfabric`/1.11.x.

[4] During the development of the tests presented in this work, our Setonix supercomputer used SLURM versions 22.02.x, and 22.05.2 to 22.05.11

variety of errors. Due to the severity of the crashes, core dumps were not particularly useful. Further investigations using open-source codes with useful logging information (such as VELOCIRAPTOR, 13 and SWIFTSIM, 14) indicated the errors occurred during MPI communication and were more likely to occur with increasing per node memory usage and/or increasing message sizes.

Our MPI tests showed available node memory shrinking during MPI communication. Increasing the number of processes would increase the unexpected excess memoryy usage, along with the likelihood of encountering uninformative errors like "bus error". This issue was severe as there was no workaround other than running codes with more nodes but fewer processes per node to reduce a job's memory footprint per node.

Additionally, some nodes would become unresponsive after crashing with bus errors or would crash or hang immediately upon trying to launch a new MPI-enabled job. Capturing the messages in the kernel ring buffer before and after the MPI-enabled code was run allowed us to quickly see what issues these nodes encountered after a crash. This information is vital for diagnosing issues by the HPC vendor (HPE Cray) and critical for quickly checking if there are unexpected memory leaks in MPI associated libraries.

The only solution to this issue was upgrading to a newer `libfabric`.

2) Mem-Reduced: : Related to the previous issue, we noticed more jobs failing due to Out-Of-Memory (OOM) errors during Phase-1 Setonix. Polling the nodes showed an ever increasing number of nodes reporting less than the expected available memory, with the amount slowly decreasing over time.

Running many iterations of the test outlined in Sec. IV-A1 or a more simplified `pt2pt` focused test showed that even small, multi-node MPI jobs that ran successfully would leave decreasing amount of available memory. The MPI or more precisely the communication library's memory leak was not being cleaned up after the successful completion of jobs.

3) Configuration-related errors: : Another use case for our tests is verification of network configurations. Although the issue encountered was specific to Joey after it had hardware and software updates applied, bringing it in line with Phase-2 Setonix, it highlights the usefulness of the above approach.

After being upgraded, Joey was unable to run moderate sized multi-node MPI jobs with `pt2pt` communication. Despite these nodes being able to run codes like HPL, they failed to run production codes like LAMMPS at scale, which has a more stressful communication pattern (more messages being sent). The multi-node MPI jobs would crash with a variety of errors. Furthermore, after a node experienced a crash, it would be unable to initialize an MPI job.

The key to diagnosing this issue was the logs produced by `dmesg`, which allowed HPE Cray staff to quickly identify an error in the network configuration, which again had gone unnoticed by more standard tests. Once the configuration was properly set, the crashes no longer occurred.

4) Scaling: : Tests of several codes, such as LAMMPS, showed that Phase-1 Setonix with the older `libfabric` library did not scale well beyond two nodes. Further testing indicated that codes with extensive `pt2pt` communication were scaling poorly. We developed a simple unit test to replicate the communication patterns during the initial MPI domain decomposition and subsequent communication per round of compute seen in production codes. Specifically, the unit test can vary the fraction of the N-sized `MPI_COMM_WORLD` each MPI process communicates with, going from a process communicating to just two neighbouring process thereby generating $2 * N$ messages, to each process communicating to all others thereby generating $N(N-1)$ messages. The test can also vary the message size sent. It then reports statistics of the time taken to complete the communication.

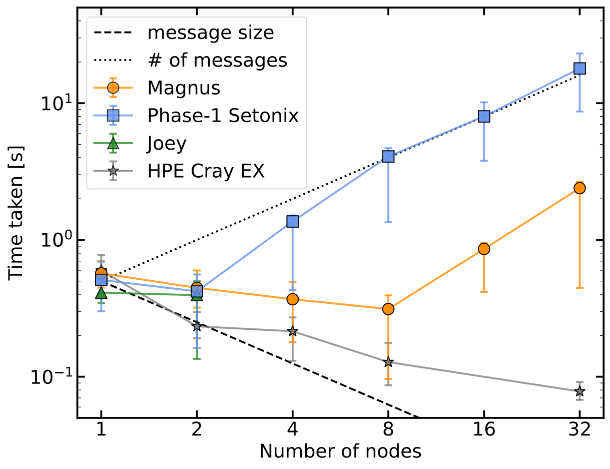

Fig. 1. Strong scaling of time taken to complete asynchronous `pt2pt` communication where each MPI process communicates to every other process. Here jobs use 24 processes. Points show the average time taken and error bars show the minimum and maximum. We also show lines indicating a scaling with message size and another for one that scales with the number of messages. We also show results from a HPE Cray EX system with an updated `libfabric` library.

Figure 1 shows the average time taken to complete a $N(N-1)$ `pt2pt` communication round, along with the minimum and maximum time taken by any given MPI process to complete send and receives. During rounds of extensive `pt2pt` communication the time taken on Phase-1 Setonix was scaling with the number of messages once the number of nodes exceeded 2, that is communication across blades (there are 2 nodes per blade). This scaling is in contrast to that seen on Magnus, our older generation Cray HPC system [5]. The minimum time on Magnus also did not increase greatly even when using 16 nodes, though at this point, the scaling seen is more like that of Phase-1 Setonix. For comparison, we also show results of running this test on LUMI, which has the

[5]Magnus has a maximum of 5 hops compared the maximum of 3 hops for Setonix

same hardware as Setonix, but had Cassini NICs and updated `libfabric` libraries.

As of this writing, the precise underlying cause for the poor scaling is unclear. A workaround for the OFI library was identified, specifically disabling the on-demand communication initialisation by setting `MPICH_OFI_STARTUP_CONNECT=1`. This moves the overhead of establishing connections at the start and significantly improved performance. The poor MPI scaling was limited to HPE Cray EX systems with `libfabric`$<$ 1.15 based on current tests.

B. SLURM tests

In this section we detail several of the key issues we have encountered in our SLURM deployment and tests we developed to either track or aid in diagnosing/fixing them. We use a vendor-provided SLURM deployment that our team have little ability to modify and/or update, so our ability to directly resolve issues is limited. Instead, these tests are used more as a tool to monitor over a long period of time the state of SLURM and to quickly identify changes (either positive or negative) when updates are applied.

1) Memory-Calc: : We discovered that some SLURM jobs were not launching due to a reported invalid node specification, despite requesting what should be valid resources. Further testing showed that the automatic memory calculation of SLURM was not working properly. Specifically, the memory per CPU was not being calculated or enforced properly if there was no specific memory request in the job submission script.

In light of this discovery we wrote a test to check SLURM's automatic memory calculation. The test attempts to submit a job script with parameters of the job script, such as number of tasks, number of CPUs, memory, etc. being able to be set and modified in an easy manner. The test will attempt to launch the job, and if the job is launched, query SLURM to check the reported memory.

The nodes on our system allow for 128 CPUs to be requested per node, however, we found that if the memory was not explicitly stated in the job submission script, SLURM would only allow jobs with up to 66 CPUs to be submitted. If the memory was explicitly set to what SLURM should be automatically calculating then larger jobs would submit successfully. Our test showed that for all jobs without any memory specification which submit successfully, the memory as reported by SLURM was correct. Thus, it appears that what SLURM is reporting is not what is being used (at least not consistently) in the determination of whether a resource request is valid in this context. Our best guess is that when the memory per CPU is not explicitly stated, SLURM's automatic calculation is not consistently enforced.

At this stage we have not identified the specific cause of this problem. Our SLURM has been upgraded twice since first identifying this issue and it has not been resolved. As a workaround for this we have recommended in our user-facing documentation that users always specify the memory.

2) Affinity: : Binding of processes and threads can impact performance and during benchmarking tests we noted some oddly poor behaviour with default SLURM binding. To that end, we run tests to check both thread-level affinity (e.g. OMP) and task-level affinity (e.g. binding MPI processes to CPUs) in different job configurations - multi-node vs. single node, exclusive vs. shared access, job packing, job arrays, etc.

The test consists of running simple codes from our library, `profile_util`, to report the affinity from within the code at run-time. Based on the job configuration and the different affinity options, the test's logic for what is considered "optimal" affinity changes.

Our testing showed that we could get consistent optimal affinity when using exclusive node access. In our case, optimal was spreading threads/processes across L3 cache regions, repeating if necessary. For Milan CPUs, L3 cache regions contain 8 cores, giving 0, 8, 16, 24, 32, 40, 48, 56, 1, 9, etc. We achieve this with exclusive access with regular jobs, job packing, and job arrays. Modifying OMP environment variables related to thread-level affinity modifies the affinity as would be expected.

However, this changes with shared node access and several jobs running at the same time. Here the affinity is generally not optimal. Process level affinity tends to hop across sockets, filling up an L3 cache in each socket before moving onto the next L3 cache in each socket. When ensuring only one job is running (using only a portion of the node), the affinity is optimal in simple cases where no job packing, job arrays, or hyperthreading are used. More advanced job configurations did not yield optimal affinity, at both the thread and process level. The OMP environment variables which affect affinity behaved as they should, as they did in exclusive access, but achieving the desired spread of processes/threads across L3 caches could not be consistently achieved in shared access.

3) Billing: : As access to resources is budgeted through allocations at Pawsey, it is important that jobs be charged correctly for consumed resources. Our billing related test submits a job, queries SLURM for job properties, checks that those match the requested resource allocation and checks that the account is correctly charged. We test a wide range of different job configurations, primarily varying the relation between an explicit memory request, and what the total memory would be given the number of CPUs we request, with hyperthreading disabled and enabled.

At Pawsey, SLURM is configured to set the memory based on the number of CPUs requested and accounts are billed the service units (SUs) using CPUs requested with this request being equal equivalent number of CPUS if there is explicit memory. Our tests showed that billing, specifically the `TRES` determined by SLURM, was correctly set except for jobs that used hyperthreading and where the explicit requested memory based number of CPUs was greater than the explicit number of CPUs requested. For these jobs, the account would be billed based on the CPU request, the smaller amount. When using one thread per core, test results were consistent across all job configurations.

4) Config: : We encountered an issue on GPU nodes where part of the SLURM configuration would revert to a different value without notice or apparent cause on active nodes running jobs. The unexpected change is very worrying given there are no notifications a change has occurred and the change affects many aspects of jobs. It also did not happen universally across all nodes at once - nodes would revert in bunches at seemingly random times.

This motivated a test that checks all GPU nodes by querying the configuration reported by SLURM and verifying that a job is able to run as if the configuration was normal. This test is somewhat unique in that we need to run it often and respond to failures by manually reconfiguring the nodes.

The specific change we observed in our configuration was the `gres` parameter would revert from `8(S:0-7)` to `8`. This broke the relationship between GPUs and CPUs on our nodes such that each GPU was no longer associated with 8 CPUs. The wrong number of CPUs were being allocated per GPU and could only be allocated in blocks of four GPUs, so in our setup, all jobs were given either 32 or 64 CPUs, regardless of what they actually asked for. This effect flowed into billing in that accounts were charged the wrong amount of SUs due to being allocated the incorrect amount of resources. Similarly, GPU-CPU affinity was affected as well. CPUs were no longer associated with the correct GPU and processes would be distributed across the incorrectly granted resources, leading to unexpected (and at times, inefficient) affinity.

C. GPU tests

We have developed a set of tests to check the functionality and performance of common GPU operations (e.g. allocation and deallocation of memory, host-to-device data transfer and vise versa) and GPU-direct MPI communication. These tests primarily serve as long-term performance monitoring, although the GPU-MPI tests also serve as an additional test of our MPI implementation. The tests are as follows:

Warmup: This test runs a few simple instructions (e.g. (de)allocation of memory, host-to-device and device-to-host data transfer) for a certain set of rounds and then runs a larger set of device instructions for a certain number of iterations. Support for multiple compilers and GPU offloading frameworks is included.

Multi-GPU: This test runs several rounds of kernel instructions concurrently across multiple GPUs. Support for multiple compilers and GPU offloading frameworks is included.

GPU-MPI: This consists of a set of tests, each of which focusses on a specific MPI communication pattern. The tests are run across multiple nodes and use multiple GPUs per node.

The GPU-MPI tests include the same node diagnostics as our other MPI tests and all the GPU tests use the `profile_util` library to achieve the same level of logging throughout the code as the MPI tests do. All GPU tests include an additional level of diagnostic information, which is usage statistics for all GPUs in the job allocation. Regular calls are made to the ROCM command line interface `rocm-smi` to capture GPU usage. This allows us to ensure the number of GPUs being engaged is what we expect along with when and how often the GPUs are used, which is critical when analysing performance.

Figure 2 shows the performance of two of our GPU-MPI tests across two different system environments. The top row is in an environment running CPE/23.03 and the bottom row is running CPE/23.09. Each test was run across three separate ROCM versions in each environment (5.2.3, 5.5.3, and 5.7.1). There are two primary findings from these test results: (i) GPU-GPU MPI does not run on our system with ROCM/5.5.3 or later versions when running CPE/23.03, a problem that is resolved in CPE/23.09; (ii) there is a significant performance improvement in point-to-point MPI communication between CPE/23.03 and CPE/23.09, up to factors of ~ 2 for 1 node and ~ 5 across 2 and 4 nodes.

Additionally, the Warmup and MultiGPU tests are designed to test multiple compilers, flags, and environment setup and test different offload options like ACC or OMP offloading. The reason is that moving to a new CPE means not only moving to different compilers but moving to new ROCM versions. Thus, they act not just as tests of GPU performance, but also compiler performance with respect to GPU code and ROCM performance.

D. Module and Software Tests

Modules, such as those provided by Lmod [4], are a common method for providing users access to software on HPC systems. With the increasing complexity of software stacks (e.g., dependencies, number of builds, number of compilers), the deployment and management has also become complex, leading to HPC package management tools like SPACK [3], which we utilise at Pawsey. Even with these tools, there can be build failures and modules not being produced.

We developed a set of software stack tests focused on checking that software has been built (by SPACK in our case), an appropriate module has been constructed, and that the library or executable function at the most fundamental level. These tests have been integrated into our software stack installation pipeline such that they are automatically run for all software and libraries at appropriate stages and results can be quickly viewed once installation has completed.

There are four separate tests we have developed, the flow of which in the software stack installation process is shown in Figure 3. The first test is run during concretization, the stage where SPACK generates concrete specs for individual instances of a package from the more generalised abstract specs. Our abstract specs are grouped into environments and the test is run once for each environment. It checks that for each abstract spec in the environment the corresponding concretized spec(s) are defined and, therefore, ready to install.

The final three tests form a test dependency chain, where each one only runs if the previous one passed. This allows a quick diagnosis of where the issue lies in case of a failure and has sped up the process of identifying the cause of failures. They are run immediately after package installation for each package for which a module is to be created:

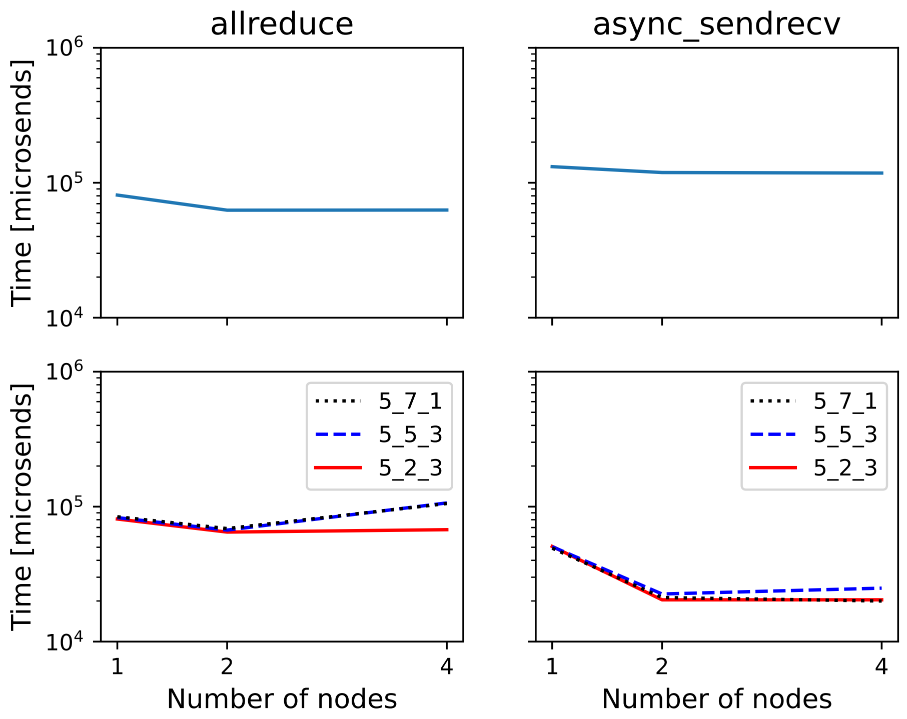

Fig. 2. Scaling of time taken to complete different types of GPU-GPU MPI communication. Here, jobs use one process per GPU. The top row are results from a CPE/23.03 environment and the bottom row is from CPE/23.09. The left column is results from running an MPI all-reduce operation while the right column is from running asynchronous point-to-point send/receive operations on data. Results from running with different versions of ROCM are also shown (note that GPU-GPU MPI did not function in later ROCM versions in CPE/23.03, which is why there is only one set of data in the panels in the top row).

Module: This checks that the module SPACK should have created exists. For each module, the existence of any modules which it implicitly loads (dependencies) is also checked. This particular issue has arisen in several versions of SPACK and does not seem to have a deterministic cause.

Load-module: This test checks that created module can be loaded without any errors or warnings and that all load dependencies are also loaded in the environment.

Exec/Lib: This test performs the most basic sanity check - for software packages this checks the binary via an invocation of `-help` or `-version` while for libraries it is a check that all library dependencies are present. While this does not guarantee correctness or performance of installed software, these tests are run during the software installation process, and as such, we opted for a basic sanity check over a proper full run of the software.

After running our full software installation pipeline alongside the above tests, we found that SPACK was not properly generating module files, although in a subtle way that we had not detected previously. While all software and libraries which installed successfully had a corresponding module file generated successfully, some of the dependencies of these packages did not have their module files generated by SPACK.

Often the packages missing modules were just dependencies of other packages, rather than final packages often used by users. We typically do not require dependencies of a package to have an explicit `LOAD` statement due to the rpaths used by SPACK. In these circumstances, no warnings or errors are produced and the basic sanity check test and any dedicated software tests would pass. However, in circumstances where we do require explicit loads to be present, such as when there is a Python dependency and the missing module is a Python one, basic sanity tests would fail.

We consider this an important result since it points to unexpected behaviour in SPACK. It can be benign in many circumstances but is not guaranteed to remain so. The nature of this issue means that it is easy to miss, given SPACK issues no warnings about the behaviour and issues would only be encountered by end users trying to load the module or use the package.

V. Conclusion

We have presented a suite of tests designed to check a multitude of aspects of an HPC system, both user-facing and lower level, from modules and job schedulers to MPI implementations. Our tests go beyond standard tests like HPL and OMB aimed at benchmarking the system. They not only test more varied MPI communication patterns and more aspects of user-facing elements, including replicating common user workflows, but also provide a wealth of diagnostic information.

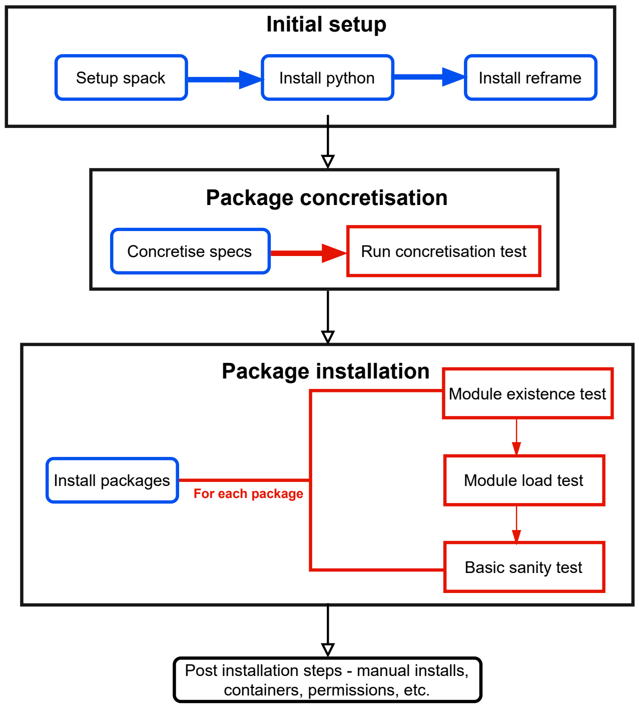

Fig. 3. Low-level flowchart for our software stack installation process. Red rectangles mark our tests and blue rounded rectangles mark other standard, already-existing, parts of our pipeline. A test to check for successful concretization in SPACK is run in the concretization phase, while three tests are run for each package during the installation phase.

The diagnostic information is perhaps of greatest importance since issues will undoubtedly be encountered. It has already proven useful in determining root causes, and eventually fixing, problems we have faced.

We believe tests of this scope should be a part of every HPC cluster's testing suite. The replication of common user workflows, increased testing on user-facing elements, and MPI tests which stress the system, have allowed us to discover several issues. Through this we have been able to update and refine advice to researchers using Pawsey, official documentation, and even adjust the setup of our system so as to minimize the likelihood of encountering these problems. Additionally, through notifying vendors and providing diagnostic reports and logs, previously unknown problems have been detected and triaged quicker than they otherwise would have been if they had instead been encountered in standard operation by a user.

Consequently, we have incorporated these tests as regression tests into our already existing suite of REFRAME benchmarking tests. They complement existing REFRAME tests and will be a standard part of our testing framework moving forward. Additionally, in the near future we plan to augment this suite of tests by setting up an automated framework for running tests at regular intervals. This would also log test results into an internal database and visualise results (e.g. failure rates, system performance, software benchmarks) on a web interface for long-term system monitoring. This will give a more comprehensive picture of our system as a whole. We have also written separate generalised versions of these tests, designed to be able to be run on any HPC system with a minimal amount of work and adaptation required. These tests are available in a public github repo at [8].

ACKNOWLEDGEMENTS

We would like to acknowledge the Whadjuk people of the Noongar nation as the traditional custodians of this country, where the Pawsey Supercomputing Research Centre is located and where we live and work. We pay our respects to Noongar elders past, present, and emerging. This work was supported by resources provided by the Pawsey Supercomputing Research Centre with funding from the Australian Government and the Government of Western Australia. The authors thank HPE Cray and all Pawsey staff, especially I. Abdurakhmanov, C. Di Pietrantonio, D.K. Deeptimahanti and S. Yates, for useful discussions. The authors also thank the CSC team of the LUMI data centre for useful conversations and access to their LUMI HPC system.

REFERENCES

[1] "Message passing interface forum, mpi: A message-passing interface standard." https://hpc.nmsu.edu/discovery/mpi/introduction/, 1994. Accessed: 2022-12-12.
[2] M. Jette, A. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," 07 2003.
[3] 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," Supercomputing 2015 (SC'15), (Austin, Texas, USA), November 15-20 2015. LLNL-CONF-669890.
[4] W. L. B. R. McLay, K. W. Schulz and T. Minyard, "Best practices for the deployment and management of production hpc clusters," vol. 9 of *In State of the Practice Reports, SC11*, pp. 1–11, Nov. 2011.
[5] V. Sochat and A. Scott, "Collaborative container modules with singularity registry hpc," *Journal of Open Source Software*.
[6] "Osu micro-benchmarks.." http://mvapich.cse.ohio-state.edu/benchmarks/, 2015. Accessed: 2022-12-12.
[7] "Reframe: A regression framework for checking the health of large hpc systems," 2017. Accessed: 2022-12-12.
[8] P. J. Elahi and C. Meyer, "Reframe mpi stress tests." https://github.com/PawseySC/Reframe-MPI-Stress-Tests.
[9] "Top 500 list." https://www.top500.org/lists/top500/2022/11/, 2022. Accessed: 2022-12-12.
[10] "Green 500 list." https://www.top500.org/lists/green500/2022/11/, 2022. Accessed: 2022-12-12.
[11] A. P. Thompson and *et al.*, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.
[12] "Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers."

https://netlib.org/benchmark/hpl/, 2018. Accessed: 2022-12-12.

[13] P. J. Elahi and *et al.*, “Hunting for galaxies and halos in simulations with VELOCIraptor,” *Publications of the Astronomical Society of Australia*, vol. 36, p. e021, May 2019.

[14] M. Schaller and *et al.*, “SWIFT: SPH With Inter-dependent Fine-grained Tasking.” Astrophysics Source Code Library, record ascl:1805.020, May 2018.