Software Quality Assurance for High Performance Computing Containers

July 2023

Matthew R Sgambati, Matthew William Anderson
DISCLAIMER

This information was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness, of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. Government or any agency thereof.
Software Quality Assurance for High Performance Computing Containers

Matthew R Sgambati, Matthew William Anderson

July 2023

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517
Software Quality Assurance for High Performance Computing Containers

Matthew Sgambati
Matthew Anderson
matthew.sgambati@inl.gov
matthew.anderson2@inl.gov
Idaho National Laboratory
Idaho Falls, Idaho, USA

ABSTRACT
Software containers are a key channel for delivering portable and reproducible scientific software in high performance computing (HPC) environments. HPC environments are different from other types of computing environments primarily due to usage of the message passing interface (MPI) and drivers for specialized hardware to enable distributed computing capabilities. This distinction directly impacts how software containers are built for HPC applications and can complicate software quality assurance efforts including portability and performance. This work introduces a strategy for building containers for HPC applications that adopts layering as a mechanism for software quality assurance. The strategy is demonstrated across three different HPC systems, two of them petaflops scale with entirely different interconnect technologies and/or processor chipsets but running the same container. Performance consequences of the containerization strategy are found to be less than 5-14% while still achieving portable and reproducible containers for HPC systems.

KEYWORDS
singularity, containers, message passing interface, software quality assurance

1 INTRODUCTION
Quality assurance in scientific computing software is fundamental to research computing and consists of multiple components including verification and validation, reproducibility, and portability. Additional components generally considered tangential to computing software quality assurance but critical to the long-term usability of an application workload are security and insulation against dependency updates occasioned by regular system security patching. The capability for software containers to package necessary dependencies and components so that scientific computing results are both reproducible and portable has driven the creation of containers for a wide range of scientific computing codes including GROMACS [4], NAMD [10], and the MOOSE framework [8]. There are multiple container registries providing containerized software including Docker Hub [2], NVIDIA [11], and Sylabs [13] that provide a container as a single file that a user can download and then run the software without needing to request a system administrator to install the software or any dependencies for them. There are also many other Open Container Initiative registries, such as Harbor [6], that provide a channel for distributing containerized software outside of the large container registries. Because of docker breakout risk [9], privilege escalation, requiring a daemon, etc., many scientific computing software containers use the Singularity and/or Apptainer platform to satisfy security requirements on shared computing resources.

Scientific computing software built for high performance computing (HPC) systems presents an additional complication for container builders due to the need to integrate drivers and software stacks for specialized hardware specific to HPC systems as well as the frequent integration of the message passing interface (MPI) libraries [5] used for leveraging distributed computing. There are three modalities by which MPI can be incorporated into a software container that are distinguished by where the MPI libraries are installed. In the container-only modality, the MPI libraries are only installed in the container. While this is the simplest approach, the container will not be able to run across multiple nodes of the host system. In the bind modality, the MPI libraries are only installed on the host system but not in the container. The MPI libraries from the host are bound into the container at run time, which allows it to run across distributed systems. However, in order for this modality to work, the host operating system and the container operating system must be compatible. In the hybrid modality [12], both the container and the host have MPI installed. For this to work on distributed systems, the host MPI and the container MPI need only be application binary interface (ABI) compatible. The bind and hybrid approaches are the most compelling for containers on HPC systems, but both present not only portability challenges but also software quality assurance issues because of the reliance on the host MPI installation and drivers for specialized hardware specific to the host. To address this concern, this work presents a hybrid container strategy that leverages layering where components of the software stack are placed in separate containers that progressively build off of each other until reaching the application layer. Changes
in host system interconnect drivers or MPI installations require updating an isolated layer and then rebuilding dependent layers to continue to ensure software quality assurance. The strategy relies on the following assumptions to help minimize the amount of layer rebuilds to improve longevity:

1. Host systems will use a long term support (LTS) version of drivers and/or software stacks when possible;
2. Host system administrators will install an ABI-compatible version of MPI if one does not exist on the target system.

To demonstrate and explore the performance consequences of this strategy, this work examines containers for three classes of applications: microbenchmarks, mini-applications, and full HPC applications. The Ohio State University benchmarks [15] serve as the demonstration for microbenchmarks while the LULESH miniapp [7] serves the mini-application space. The full HPC application explored with this strategy is MCNP 6.2.0 [17]. These application containers are tested on three different systems, two of them petaflops scale, with different operating systems, chipsets, and network technologies. The performance of each container is compared against the natively compiled and optimized version of the application on each supercomputing system.

The structure of this work is as follows. In Section 2, a review of alternative container build strategies for HPC systems is examined and related work is discussed. In Section 3, the container strategy for HPC proposed in this work is detailed and discussed. In Section 4, the performance results for the three classes of HPC applications that are containerized as part of this work are given and the performance consequences of the containerization strategy are quantified. In Section 5, conclusions on the empirical tests of this containerization strategy are presented and future work is discussed.

2 Related Work

Multiple performance studies with a discussion on portability have been done for HPC container strategies. Torrez et al. [14] reported minimal or no performance impact due to using an HPC container but tested portability using two different supercomputers with the exact same OmniPath interconnect, motherboard, and chipset. That study used SysBench, STREAM, and HPCG as applications for memory and performance analysis and built the containers using Charliecloud, Shifter, and Singularity as the container platforms within a hybrid modality. No full applications were included in the Torrez et al. study but they concluded that the performance question is close to a solved problem. Wang et al. [16] followed a non-portable bind-modality strategy in their container performance study and used four full applications as part of their investigation: Weather Research Forecasting (WRF), the lattice gauge theory MILC code, NAMD, and GROMACS and only tested on one supercomputer thereby avoiding the question of portability across different supercomputing systems. Like Torrez et al., they also conclude that containerized versions of HPC software did not sacrifice performance compared with the native non-container installations. Canon et al. [1] point out that the mechanisms needed to achieve portability and reproducibility in containers can inadvertently cause performance degradation. They point out that methods to leverage specialized HPC hardware in the container can have the unintended consequence of breaking long-term reproducibility. Canon et al. observe that, at present, the techniques for using containers on HPC systems are still ad hoc.

This work complements those studies by detailing an HPC container strategy that is portable across supercomputers even if they are hosting completely different interconnect technologies and minimizes the risk of breaking long-term reproducibility. This strategy does have a performance consequence, and this study empirically explores that performance impact.

3 Container Strategy

This section presents a hybrid container strategy to provide traceability and portability across different supercomputers while also minimizing the risk of breaking long-term reproducibility. Example recipe files for the strategy are provided to explicitly illustrate the different layers and the traceability elements added to the each layer.

There are three key components to the strategy:

- HPC application containers are built from stacks of individual containers called layers. These layers compartmentalize different software and driver components.
- Specialized hardware drivers and software stacks for interconnects are grouped into a single layer. MPI libraries are likewise grouped into their own layer. Portability among HPC systems is achieved via ABI compatibility of the MPI libraries between host and container along with series compatibility for interconnect drivers and software stacks. The UCX framework is also leveraged in the container to assist with portability.
- Each layer of the container stack can be individually updated and is always reproducible by storing local mirrors of all layer components rather than relying on the network for rebuilding the layer.

This strategy is designed to provide maximum traceability due to the local static mirrors and separating components into different layers in addition to improving the portability and reproducibility needed for software quality assurance.

In a hybrid modality such as detailed in this strategy, the container has both an MPI installation as well as the necessary interconnect drivers and software stacks that can be independent of the system MPI installation and system interconnect drivers and software stacks. Unlike the bind modality, this hybrid approach does not require compatibility between the host operating system and the container operating system which substantially improves portability. There are, however, two limitations to portability. This first is that the MPI installed in the container must be ABI compatible with the host MPI. This is a limitation typical of all hybrid container approaches. For example, if MVAPICH2 is installed on the host, some ABI compatible MPI versions that could be used in the container would include Intel MPI, MPICH, and MVAPICH2. The second limitation is that the interconnect drivers and software stacks in the container must be series compatible with the host drivers and software stacks. For example, if the InfiniBand driver and software stack on the host is version 5.1, the container would need to have InfiniBand driver and software stack versions also that are series compatible, e.g. 5.x. For portability across different
HPC systems with different network technologies, the drivers and software stacks for each would need to be included in the container. For example, if the container will be used on a host system with InfiniBand interconnect as well as on another host system with OmniPath interconnect, series compatible drivers and software stacks for each network technology would need to be included in the container.

A consequence of these portability conditions is that there will be times it will be necessary to update the container’s version of MPI or interconnect drivers and software stacks in order to ensure continued portability. Rebuilding a monolithic container would normally be required, but that also requires rebuilding multiple components that are unaffected by the updates needed for portability. To avoid this a layering approach is used where operating system, compilers, interconnect drivers and software stack, MPI libraries, and applications are all placed in separate containers. compartmentalizing components into individual layers of the container enables the container builder to update only those pieces needed to continue to ensure portability and long-term reproducibility. These different layers are stacked with the layers least likely to change near the bottom and the layers most likely to change near the top as illustrated in Figure 1. Rather than build a monolithic container by pulling everything from the internet all at once, using a layering approach it is possible to pull and modify only the pieces necessary for that layer, which not only reduces build time, but minimizes the number of configuration items for an audit or during verification and validation. When a layer needs to be updated and rebuilt, only that layer and the layers above it have to be rebuilt while layers below can be reused in the final stack. For example, because the interconnect driver series does not change frequently on HPC systems and sometimes lasts as long as 5-7 years, the interconnect driver layer is lower in the pyramid in Figure 1 than the MPI libraries which change more rapidly.

Pulling components for a layer from the internet can present a problem with regards to reproducibility and auditing. When container components are pulled from the internet, what happens in that space may not be reproducible. For example,

```
$ apt install python
```

in a container recipe file won’t always give the same version. In contrast, pulling from static locally stored mirrors helps ensure that each layer of the container stack can be fully rebuilt at any time without reproducibility concerns. This is the option employed in this strategy to reduce and/or eliminate reproducibility concerns. Additionally, each container layer is signed and then verified in the next layer’s build step via the Fingerprint header in the definition files ensuring that layers are built with the expected sub-layers. Also, the following host system attributes are added to each layer during the build process for audit purposes to assist with reproducibility if building the layer on the same hardware is desired:

- CPU information
  - Architecture
  - Model Name
- uname information
  - Kernel name
  - Kernel release
  - Kernel version
  - Processor type
  - Hardware platform
  - Operating system

To illustrate this strategy, an application executing “hello world” using MPI is detailed in the following section with the Singularity recipe files for each layer described. The Fingerprint headers as described above have been removed in the following example for clarity.

### 3.1 "Hello World" Example

The base container contains the operating system for which the entire stack will be based on. This layer is the only layer that reaches out to the internet and would have a recipe file similar to that illustrated in Figure 2. If tighter control of the base container operating system is required then a different bootstrap agent, such as yum, can be used, which can be pointed to the static local mirrors if desired.

The base+ container stacks on the base container. It modifies all of the base OS repositories to point to the static local mirrors instead of the internet based ones. Then it adds dependencies for building code, pulling down source code via git, handling tar and compressed files, as well as an editor for viewing and modifying files. Finally, it captures some of the build host systems attributes and stores them in the metadata of the container as illustrated in the recipe file in Figure 3 and stacks on the base container in Figure 2.

The base++ container adds the HPC system interconnect drivers and software stacks. Drivers and software stacks for multiple interconnect technologies can be added to this layer to provide portability. For instance, both OmniPath and InfiniBand drivers can be added to this layer. An example recipe file adding both InfiniBand and OmniPath in a layer is shown in Figure 4.
Finally, the MPI+base++ layer adds MPI to the container and enables building the application layers, which require MPI. Additional frameworks or software can be added to this layer that enhance the capabilities or functionality of MPI to support greater portability. An example recipe file for this layer is shown in Figures 5-7.

To enhance portability of this layer, the UCX framework and two "base" MPIs were added to support a wider range of host system MPI libraries, MPICH and OpenMPI. MPICH is ABI compatible with multiple MPI libraries, such as Intel MPI, MVAPICH2, and MPICH. and OpenMPI is ABI compatible with itself. This makes it so that when the application layers are built, they can build two
Figure 4: Singularity recipe file for the base++ layer container. This container adds the drivers and software stacks for two HPC interconnect technologies, InfiniBand and OmniPath.

versions of the application, one against MPICH and one against OpenMPI, allowing for much greater portability with host systems and decreasing the likelihood of needing to make changes to host systems.
Figure 5: Singularity recipe file for the MPI+base++ layer container showing the MPI “hello world” test code and UCX installation steps.
simplifies the verification and validation steps, and stills maintains the reproducibility and portability offered by this strategy.

4 PERFORMANCE MEASUREMENTS AND RESULTS

The results in this section originate from runs executed on three different types of HPC systems: an Intel Cascade Lake based system with InfiniBand EDR interconnect (Sawtooth), an Intel Skylake based system with OmniPath interconnect (Lemhi), and an AMD EPYC based system with InfiniBand HDR interconnect (Hoodoo). These systems are summarized in Table 1. One additional system was used just for testing portability but not performance. One container was built for each of the test applications (OSU microbenchmark, LULESH, and MCNP). This one container was run on all the HPC systems and performance was compared against the natively compiled and optimized application version. As noted in the Section 3, because MVAPICH2 and OpenMPI are not ABI compatible, the container contained application builds of both MPICH (ABI compatible with MVAPICH2) and OpenMPI for performance comparison.

Performance comparisons for the container version of LULESH are shown in Figure 8. In these performance measurements, LULESH
was run for 1000 iterations with $30^3$ points per domain. The same container was used to run on each system. To facilitate performance measurements between different MPI installations, the container has two LULESH executables: one built against OpenMPI 4.1.4 and one built against MPICH 3.4.3. This enables performance measurement for the container with different host MPI installations. At each core count, the simulation was run five times and the average run time is reported.

At most core counts, the container version of LULESH ran slower than the natively compiled and optimized version by an amount varying between 3% to 14% at the largest core counts. At some specific core counts, the container consistently outperformed the natively compiled version. But in general, there was a relatively small performance penalty as part of the container strategy to ensure conditions for software quality assurance.

Performance comparisons between the container and native compiled version of the OSU All-to-Allv microbenchmark across the three supercomputer systems are shown in Figure 9. All tests were run on 10 nodes of the system. Interestingly, the container average latency was occasionally a little lower for some message

<table>
<thead>
<tr>
<th>System Name</th>
<th>Core Count</th>
<th>Chipset</th>
<th>Interconnect / Version</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sawtooth</td>
<td>99,792</td>
<td>Intel Xeon 8268</td>
<td>InfiniBand EDR / 4.9-4.1.7</td>
<td>CentOS Linux release 7.9.2009 (Core)</td>
</tr>
<tr>
<td>Lemhi</td>
<td>20,160</td>
<td>Intel Xeon 6148</td>
<td>OmniPath / 10.11.0.2-1</td>
<td>Rocky Linux release 8.7 (Green Obsidian)</td>
</tr>
<tr>
<td>Hoodoo</td>
<td>352</td>
<td>AMD EPYC 7302</td>
<td>InfiniBand HDR / 5.5-1.0.3</td>
<td>Rocky Linux release 8.5 (Green Obsidian)</td>
</tr>
<tr>
<td>Galena</td>
<td>40</td>
<td>Intel Xeon E5-2698</td>
<td>InfiniBand EDR / 5.4-5.5.8</td>
<td>Ubuntu 20.04.5 LTS (Focal Fossa)</td>
</tr>
</tbody>
</table>
The same container was used in each of these comparisons on three different supercomputers featuring different interconnect technologies, operating systems, and chipsets. The container has the LULESH executable compiled with OpenMPI 4.1.4 and another compiled with MPICH 3.4.3; the LULESH executable run was the version that is ABI compatible with the host MPI. The host MPI used to run the container was varied to observe any potential performance impact. The performance difference between container and natively compiled version at the highest core count on each system varied from 3% to 14%.

Figure 8: LULESH performance comparison between the container and natively compiled versions. In this plot, lower is better. The same container was used in each of these comparisons on three different supercomputers featuring different interconnect technologies, operating systems, and chipsets. The container has the LULESH executable compiled with OpenMPI 4.1.4 and another compiled with MPICH 3.4.3; the LULESH executable run was the version that is ABI compatible with the host MPI. The host MPI used to run the container was varied to observe any potential performance impact. The performance difference between container and natively compiled version at the highest core count on each system varied from 3% to 14%.

Figure 8: LULESH performance comparison between the container and natively compiled versions. In this plot, lower is better. The same container was used in each of these comparisons on three different supercomputers featuring different interconnect technologies, operating systems, and chipsets. The container has the LULESH executable compiled with OpenMPI 4.1.4 and another compiled with MPICH 3.4.3; the LULESH executable run was the version that is ABI compatible with the host MPI. The host MPI used to run the container was varied to observe any potential performance impact. The performance difference between container and natively compiled version at the highest core count on each system varied from 3% to 14%.

Figure 8: LULESH performance comparison between the container and natively compiled versions. In this plot, lower is better. The same container was used in each of these comparisons on three different supercomputers featuring different interconnect technologies, operating systems, and chipsets. The container has the LULESH executable compiled with OpenMPI 4.1.4 and another compiled with MPICH 3.4.3; the LULESH executable run was the version that is ABI compatible with the host MPI. The host MPI used to run the container was varied to observe any potential performance impact. The performance difference between container and natively compiled version at the highest core count on each system varied from 3% to 14%.

sizes than the natively compiled version. This was especially true in the case where the host was MVAPICH2 2.3.5 and the container used MPICH 3.4.3. For this microbenchmark, there were essentially no negative performance consequences to running the container built using the described software quality assurance strategy.

Performance comparisons between container and native compiled version of an MCNP benchmark across the two supercomputer systems are shown in Figure 10. In this full application, the percentage difference in container performance from the natively compiled version of MCNP is shown: positive percentage differences indicate the container ran slower than the native build while negative percentage differences indicate the container ran faster than the native build. On Sawtooth, the container consistently outperformed the native build by over 20% at larger core counts which suggests that the production natively compiled MCNP application may need further optimization. On Lemhi, the MCNP container is generally 5% slower or less than the natively compiled application. In these performance tests, the host was OpenMPI and the exact same MCNP executable was used in the container for both systems.

5 CONCLUSIONS

This work presented an unique approach to handling traceability, portability, and reproducibility as part of software quality assurance for containers across different host systems with different chipsets, interconnects, and OSes utilizing a layering approach. An empirical measurement of the performance costs associated
with this software quality assurance strategy has been presented for three different applications across three different supercomputers. The upper bound for the performance cost for this strategy was 14% but this was not uniform across the applications and core counts. In several instances, the software quality assurance container consistently outperformed the natively compiled application. The software quality assurance container was also tested for portability on one additional DGX-1 system with Ubuntu 20.04. While the strategy was validated for Singularity and/or Apptainer in this work, the strategy is not limited to just these container platforms.

Future work will explore the software quality assurance strategy for additional widely used HPC applications including the Vienna Ab Initio Simulation package and GROMACS+CP2K [3]. Because the performance metrics collected in this work used largely unoptimized versions of the software in the containers, future work includes exploring how much optimization could be performed on the containerized software without affecting portability. It might be possible to reduce some of the performance gasps reported here via specific optimizations. Even though there was generally a small performance loss in order to achieve portability, this work has shown that this strategy for containers can provide a reproducible, traceable, and portable container with minimal to no changes required on host systems.

ACKNOWLEDGMENTS
This research made use of the resources of the High Performance Computing Center at Idaho National Laboratory, which is supported
Figure 10: MCNP benchmark comparison between the container and natively compiled versions. The same container was used on both systems. On Lemhi, the natively compiled MCNP performance is generally comparable to the container performance while on Sawtooth the container performance was significantly better than the natively compiled version.
REFERENCES


