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

ACM Reference Format:
Matthew Sgambati and Matthew Anderson. 2023. Software Quality Assurance for High Performance Computing Containers. In *Practice and Experience in Advanced Research Computing (PEARC '23), July 23–27, 2023, Portland, OR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3569951.3593596

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

PEARC '23, July 23–27, 2023, Portland, OR, USA

ACM ISBN 978-1-4503-9985-2/23/07.
https://doi.org/10.1145/3569951.3593596

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 internet 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.

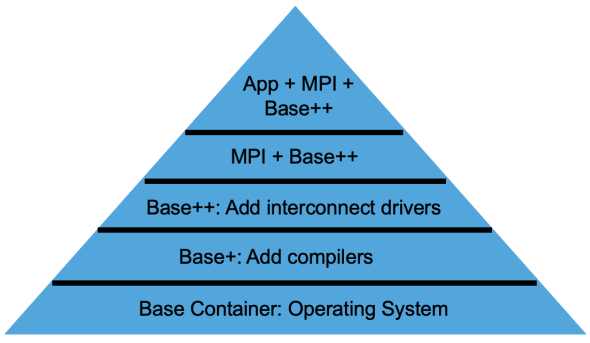


Figure 1: Hybrid container layering strategy where the operating system, compilers, interconnect drivers and software stack, MPI, and application are added in different container layers. When a layer needs to be modified, only that layer and layers stacked on it need to be rebuilt; lower layers can be reused.

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 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.

```
Bootstrap: docker
From: rockylinux:8.6

%post
    # Capture useful system information
    echo "Architecture" $(lscpu | grep "^Architecture" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
    echo "CPU" $(lscpu | grep "^Model name" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
    echo "uname" $(uname -srvpio) >> "${SINGULARITY_LABELS}"

%test
    if grep -q 'NAME="Rocky Linux"' /etc/os-release; then
        echo "SUCCESS: Container base is Rocky Linux as expected."
    else
        echo "ERROR: Container base is not Rocky Linux."
        exit 1
    fi

%labels
    Authors Matthew.Sgambati@inl.gov Matthew.Anderson2@inl.gov
    Version 1.0.0

%help
    Rocky Linux 8.6 Base Container

```

Figure 2: Singularity recipe file for the base layer container. This base container only has the operating system.

```
Bootstrap: oras
From: <container_registry>/hpcbase/base_00:1.0.0

%post
    # Change repos to point to local static mirror
    sed -i 's/^mirrorlist/#mirrorlist/' /etc/yum.repos.d/Rocky-*.repo
    sed -i 's#.*baseurl=http://dl.rockylinux.org/\$contentdir#baseurl=http://<local_static_mirror>/repos/rocky-linux/20221208#'
    ↪  /etc/yum.repos.d/Rocky-*.repo

    dnf clean all
    dnf makecache

    # Install commonly used packages for building code and modifiying files
    dnf install -y bzip2 gcc gcc-gfortran gcc-c++ gdb git make python39 python39-pip python39-setuptools tar vim
    dnf clean all

    # Capture useful system information
    echo "Architecture" $(lscpu | grep "^Architecture" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
    echo "CPU" $(lscpu | grep "^Model name" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
    echo "uname" $(uname -srvpio) >> "${SINGULARITY_LABELS}"

%test
    GCC=$(which gcc)
    if [ $? -eq 0 ]; then
        echo "SUCCESS: gcc is available at ${GCC}"
    else
        echo "ERROR: gcc is not installed"
        exit 1
    fi

%labels
    Authors Matthew.Sgambati@inl.gov Matthew.Anderson2@inl.gov
    Version 1.0.0

%help
    Rocky Linux 8.6 Base Container
    Extra dependencies for building code and modifiying files are installed in this layer.

```

Figure 3: Singularity recipe file for the base+ layer container. This base container adds dependencies for building code like compilers, modifying files, etc.

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

```
Bootstrap: oras
From: <container_registry>/hpcbase/extra_01:1.0.0

%post
    # Create environment variables of software versions
    MLNX_OFED=MLNX_OFED_LINUX-5.4-3.6.8.1-rhel8.6-x86_64
    OPX=CornelisOPX-Basic.RHEL86-x86_64.10.12.1.0.7

    # Make src in /opt to hold source code
    mkdir -p /opt/src

    # Download the MLNX_OFED and OPX files to /opt
    cd /opt
    curl -O http://<local_static_mirror>/interconnects/MLNX/20221208/${MLNX_OFED}.tgz
    curl -O http://<local_static_mirror>/interconnects/OPX/20230212/${OPX}.tgz

    # Install required packages for InfiniBand
    dnf install -y python39 pciutils lsof ethtool tcsh libnl3 tk numactl-libs tcl

    # Extract MLNX_OFED and install it
    tar xf ${MLNX_OFED}.tgz -C /opt/src
    /opt/src/${MLNX_OFED}/mlnxofedinstall --without-fw-update --skip-unsupported-devices-check --basic --user-space-only --distro RHEL8.6
    ↪  --without-depcheck -q

    # Install required packages for OmniPath
    dnf install -y irqbalance kernel-modules-extra kmod libgcc perl perl-Getopt-Long perl-Socket opensm-libs python2 libatomic
    dnf download ibacm*x86_64
    rpm -ivh --nodeps ibacm*.rpm

    # Extract OPX and install it
    tar xf ${OPX}.tgz -C /opt/src
    cd /opt/src/${OPX}
    ./INSTALL -i intel_hfi -i opa_stack --user-space

    # Clean up tarballs/downloads and source directories
    cd /opt
    rm -rf /opt/src
    rm -f /opt/*.tgz
    rm -f /opt/*.rpm

    # Capture useful system information
    echo "Architecture" $(lscpu | grep "^Architecture" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
    echo "CPU" $(lscpu | grep "^Model name" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
    echo "uname" $(uname -srvpio) >> "${SINGULARITY_LABELS}"

%test
    OFED=$(which ofed_info)
    if [ $? -eq 0 ]; then
        echo "SUCCESS: ofed_info is available at ${OFED}"
    else
        echo "ERROR: ofed_info is not installed"
        exit 1
    fi

    OPX=$(which opainfo)
    if [ $? -eq 0 ]; then
        echo "SUCCESS: opainfo is available at ${OPX}"
    else
        echo "ERROR: opainfo is not installed"
        exit 1
    fi

%labels
    Authors: Matthew.Sgambati@inl.gov Matthew.Anderson2@inl.gov
    Version: 1.0.0

%help
    Rocky Linux 8.6 Base Container
    Extra dependencies for building code and modifiying files are installed in this layer.
    InfiniBand driver and OFED stack is installed in this layer.
    OmniPath driver and OPA stack is installed in this layer.
```

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.

Another feature of this strategy is that application layers can choose which layer to build against. For example, if an application does not need MPI, it can be built against the base++ layer or if it does not need MPI or the interconnect, then it can build against the base+ layer. This helps keep the size of the containers down,

```
Bootstrap: oras
From: <container_registry>/hpcbase/interconnect_02:1.0.0

%post
    # Install required packages
    dnf install -y hwloc findutils

    # Create directories to store files and source code
    mkdir -p /opt/mpi/examples
    mkdir -p /opt/src
    mkdir -p /opt/tars

    # Create mpitest.c program from here doc
    cat <<- EOF > /opt/mpi/examples/mpitest.c
        #include <mpi.h>
        #include <stdio.h>
        #include <stdlib.h>

        int main (int argc, char **argv) {
                int rc;
                int size;
                int myrank;

                rc = MPI_Init (&argc, &argv);
                if (rc != MPI_SUCCESS) {
                    fprintf (stderr, "MPI_Init() failed");
                    return EXIT_FAILURE;
                }

                rc = MPI_Comm_size (MPI_COMM_WORLD, &size);
                if (rc != MPI_SUCCESS) {
                    fprintf (stderr, "MPI_Comm_size() failed");
                    goto exit_with_error;
                }

                rc = MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
                if (rc != MPI_SUCCESS) {
                    fprintf (stderr, "MPI_Comm_rank() failed");
                    goto exit_with_error;
                }

                fprintf (stdout, "Hello, I am rank %d/%d\n", myrank, size);

                MPI_Finalize();

                return EXIT_SUCCESS;

                exit_with_error:
                    MPI_Finalize();
                    return EXIT_FAILURE;
        }

        EOF

    # Install UCX
    UCX_VERSION=1.13.1
    UCX_NAME="ucx-${UCX_VERSION}"
    UCX_URL="http://<local_static_mirror>/mpi/ucx/20230105/${UCX_NAME}.tar.gz"
    UCX_DIR="/opt/mpi/${UCX_NAME}"

    echo "Installing UCX-${UCX_VERSION}..."
    ## Download
    cd /opt/tars
    curl -O ${UCX_URL}
    tar xf ${UCX_NAME}.tar.gz -C /opt/src

    ## Compile and install
    cd /opt/src/${UCX_NAME}
    mkdir build
    cd build
    ../configure --prefix=${UCX_DIR}
    make -j 16 |& tee log.make
    make check |& tee log.make_check
    make install |& tee log.make_install
```

Figure 5: Singularity recipe file for the MPI+base++ layer container showing the MPI "hello world" test code and UCX installation steps.

```
75
76      # Install MPICH
77      MPICH_VERSION=3.4.3
78      MPICH_NAME="mpich-${MPICH_VERSION}"
79      MPICH_URL="http://<local_static_mirror>/mpi/mpich/20221208/${MPICH_NAME}.tar.gz"
80      MPICH_DIR="/opt/mpi/${MPICH_NAME}"
81
82      echo "Installing MPICH-${MPICH_VERSION}..."
83      ## Download
84      cd /opt/tars
85      curl -O ${MPICH_URL}
86      tar xf ${MPICH_NAME}.tar.gz -C /opt/src
87
88      ## Compile and install
89      cd /opt/src/${MPICH_NAME}
90      mkdir build
91      cd build
92      ../configure --prefix=${MPICH_DIR} --with-ucx=${UCX_DIR}
93      make -j 16 |& tee log.make
94      make check |& tee log.make_check
95      make install |& tee log.make_install
96
97      # Install OpenMPI
98      OPENMPI_VERSION=4.1.4
99      OPENMPI_NAME="openmpi-${OPENMPI_VERSION}"
100     OPENMPI_DIR="/opt/mpi/${OPENMPI_NAME}"
101     OPENMPI_URL="http://<local_static_mirror>/mpi/openmpi/20221208/${OPENMPI_NAME}.tar.gz"
102
103     echo "Installing OPENMPI-${OPENMPI_VERSION}..."
104     ## Download
105     cd /opt/tars
106     curl -O ${OPENMPI_URL}
107     tar xf ${OPENMPI_NAME}.tar.gz -C /opt/src
108
109     ## Compile and install
110     cd /opt/src/${OPENMPI_NAME}
111     mkdir build
112     cd build
113     ../configure --prefix=${OPENMPI_DIR} --with-ucx=${UCX_DIR}
114     make -j 16 |& tee log.make
115     make check |& tee log.make_check
116     make install |& tee log.make_install
117
118     echo "Compiling the MPI application..."
119     cd /opt/mpi/examples
120     echo "MPICH-${MPICH_VERSION}..."
121     PATH=${MPICH_DIR}/bin:${PATH} LD_LIBRARY_PATH=${MPICH_DIR}/lib:${LD_LIBRARY_PATH} mpicc -o mpitest_${MPICH_NAME} mpitest.c
122
123     echo "OPENMPI-${OPENMPI_VERSION}..."
124     PATH=${OPENMPI_DIR}/bin:${PATH} LD_LIBRARY_PATH=${OPENMPI_DIR}/lib:${LD_LIBRARY_PATH} mpicc -o mpitest_${OPENMPI_NAME} mpitest.c
125
126     # Clean up downloads and source
127     rm -rf /opt/tars
128     rm -rf /opt/src
129
130     # Capture useful system information
131     echo "Architecture" $(lscpu | grep "^Architecture" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
132     echo "CPU" $(lscpu | grep "^Model name" | cut -d ':' -f 2) >> "${SINGULARITY_LABELS}"
133     echo "uname" $(uname -srvpio) >> "${SINGULARITY_LABELS}"
134
```

Figure 6: Singularity recipe file for the MPI+base++ layer container showing the MPICH and OpenMPI installation steps as well as compiling the "hello world" test code for each MPI.

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

```
%test
    MPICH=mpich-3.4.3
    OPENMPI=openmpi-4.1.4

    if [ ! -f /opt/mpi/examples/mpitest_${MPICH} ]; then
        echo "ERROR: MPICH mpitest application not found."
        exit 1
    else
        echo "Running mpitest (mpi hello world) with np 2."
        OUTPUT=$(/opt/mpi/${MPICH}/bin/mpirun -np 2 /opt/mpi/examples/mpitest_${MPICH})
        echo -e "OUTPUT:\n${OUTPUT}"
        if echo "${OUTPUT}" | grep -q "0/2" && echo "${OUTPUT}" | grep -q "1/2"; then
            echo "SUCCESS:  MPICH mpitest application compiled and ran correctly."
        else
            echo "ERROR: MPICH mpitest application did not run correctly."
            exit 1
        fi
    fi

    if [ ! -f /opt/mpi/examples/mpitest_${OPENMPI} ]; then
        echo "ERROR: OpenMPI mpitest application not found."
        exit 1
    else
        echo "Running mpitest (mpi hello world) with np 2."
        OUTPUT=$(OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 /opt/mpi/${OPENMPI}/bin/mpirun -np 2
        ↪  /opt/mpi/examples/mpitest_${OPENMPI})
        echo -e "OUTPUT:\n${OUTPUT}"
        if echo "${OUTPUT}" | grep -q "0/2" && echo "${OUTPUT}" | grep -q "1/2"; then
            echo "SUCCESS:  OpenMPI mpitest application compiled and ran correctly."
        else
            echo "ERROR: OpenMPI mpitest application did not run correctly."
            exit 1
        fi
    fi

%labels
    Authors: Matthew.Sgambati@inl.gov Matthew.Anderson2@inl.gov
    Version: 1.0.0

%help
    Rocky Linux 8.6 Base Container
    Extra dependencies for building code and modifiying files are installed in this layer.
    InfiniBand driver and OFED stack is installed in this layer.
    OmniPath driver and OPA stack is installed in this layer.
    MPICH is installed in this layer.
    OpenMPI is installed in this layer.
```

Figure 7: Singularity recipe file for the MPI+base++ layer container showing the test, labels, and help sections.

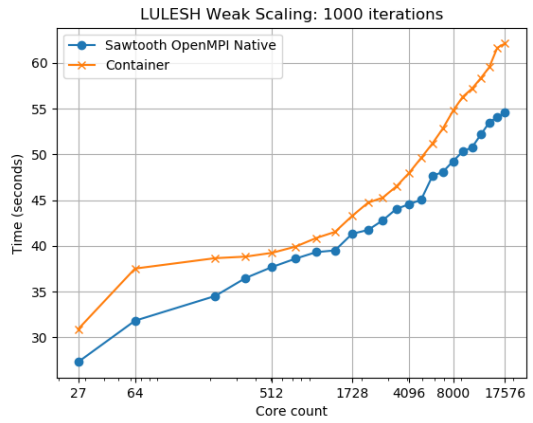
System Name	Core Count	Chipset	Interconnect / Version	OS
Sawtooth[1,2]	99,792	Intel Xeon 8268	InfiniBand EDR / 4.9-4.1.7	CentOS Linux release 7.9.2009 (Core)
Lemhi[1,2]	20,160	Intel Xeon 6148	OmniPath / 10.11.0.2-1	Rocky Linux release 8.7 (Green Obsidian)
Hoodoo[1,2]	352	AMD EPYC 7302	InfiniBand HDR / 5.5-1.0.3	Rocky Linux release 8.5 (Green Obsidian)
Galena[1]	40	Intel Xeon E5-2698	InfiniBand EDR / 5.4-3.5.8	Ubuntu 20.04.5 LTS (Focal Fossa)

Table 1: Systems used for container portability[1] and performance testing[2]. The same container was used on all systems to test portability. Performance of the container was compared against the natively compiled and optimized application on each system.

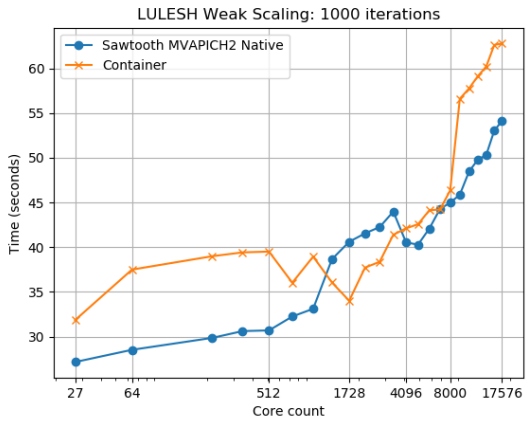
was run for 1000 iterations with 30^3 points per domain. The same container was used to run on each system. To facilitate performance measurements between different MPI installations, the container has two LULESH executables: one built against OpenMPI 4.1.4 and one built against MPICH 3.4.3. This enables 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.

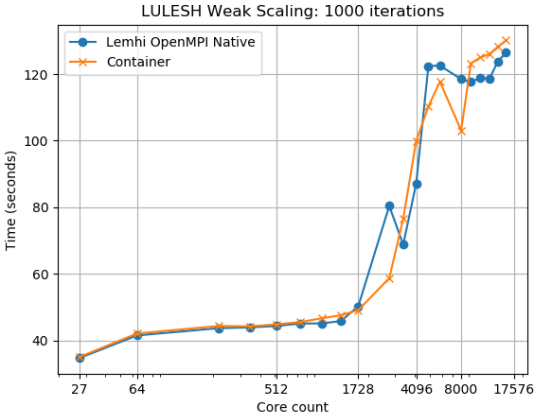
Performance comparisons between the container and native compiled version of the OSU All-to-Allv microbenchmark across the three supercomputer systems are shown in Figure 9. All tests were run on 10 nodes of the system. Interesting, the container average latency was occasionally a little *lower* for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



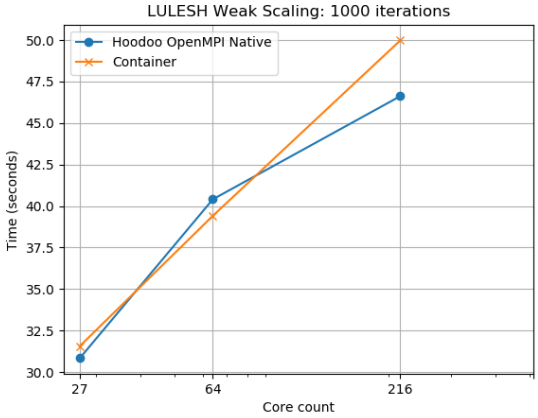
(a) Sawtooth with OpenMPI 4.1.4 as the host MPI



(b) Sawtooth with MVAPICH2 2.3.5 as the host MPI



(c) Lemhi with OpenMPI 4.1.1 as the host MPI



(d) Hoodoo with OpenMPI 4.0.5 as the host MPI

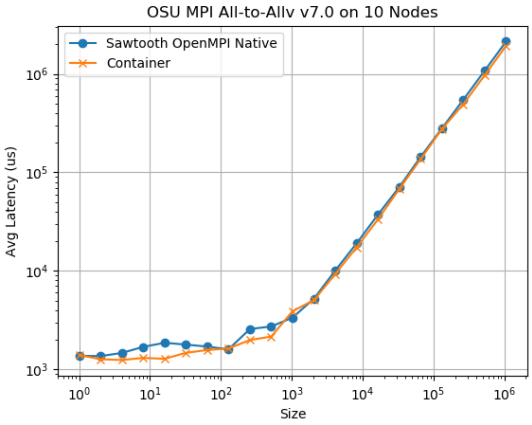
Figure 8: LULESH performance comparison between the container and natively compiled versions. In this plot, lower is better. The same container was used in each of these comparisons on three different supercomputers featuring different interconnect technologies, operating systems, and chipsets. The container 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.

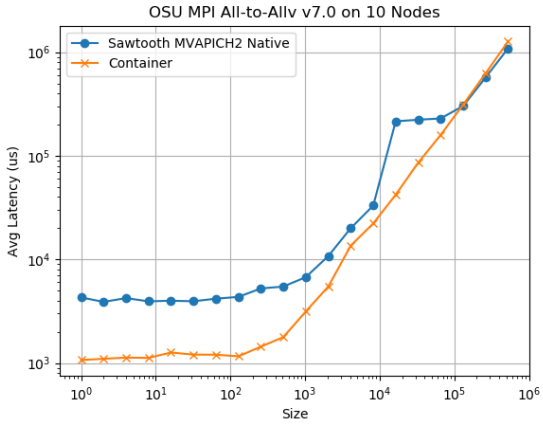
Performance comparisons between container and native compiled version of an MCNP benchmark across the two supercomputer systems are shown in Figure 10. In this full application, the percentage difference in container performance from the natively compiled version of MCNP is shown: positive percentage 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

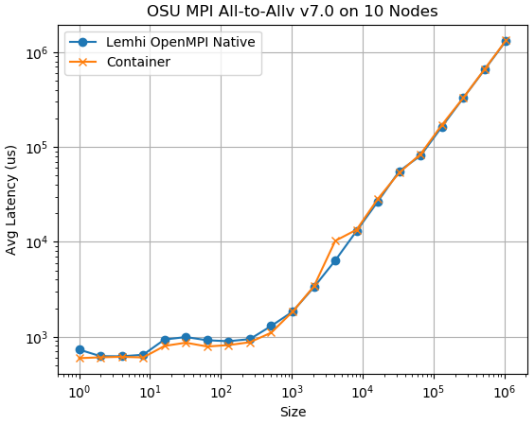
This work presented an unique approach to handling traceability, portability, and reproducibility as part of software quality assurance for containers across different host systems with different chipsets, interconnects, and OSes utilizing a layering approach. An empirical measurement of the performance costs associated



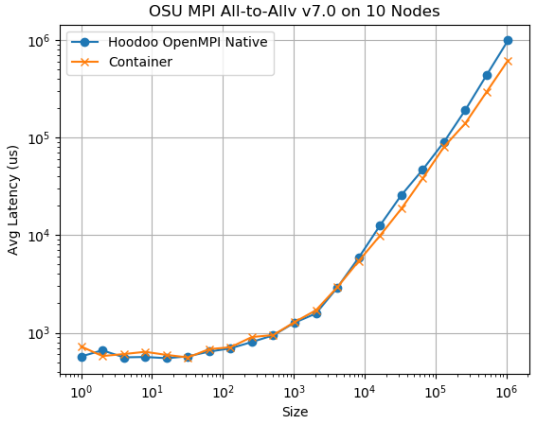
(a) Sawtooth with OpenMPI 4.1.4 as the host MPI



(b) Sawtooth with MVAPICH2 2.3.5 as the host MPI



(c) Lemhi with OpenMPI 4.1.1 as the host MPI



(d) Hoodoo with OpenMPI 4.0.5 as the host MPI

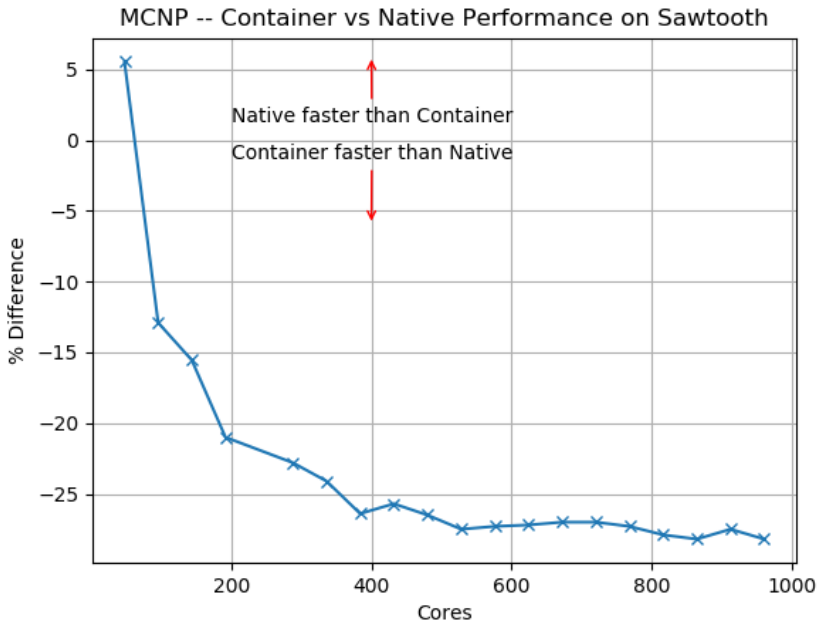
Figure 9: OSU All-to-Allv MPI microbenchmark performance comparison between the container and natively compiled versions. In this plot, lower is better. The same container was used in each of these comparisons on three different supercomputers featuring different interconnect technologies, operating systems, and chipsets and run on 10 nodes. The container has the OSU MPI All-to-Allv benchmark compiled with OpenMPI 4.1.4 and also compiled with MPICH 3.4.3. There were no negative performance consequences by using the container for this microbenchmark.

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.

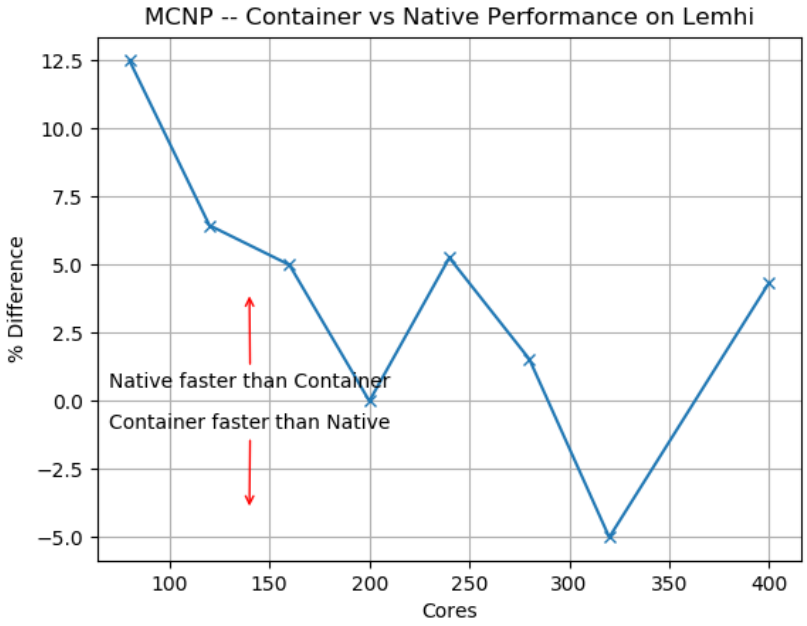
Future work will explore the software quality assurance strategy for additional widely used HPC applications including the Vienna Ab Initio Simulation package and GROMACS+CP2K [3]. Because the performance metrics collected in this work used largely unoptimized versions of the software in the containers, 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



(a) Sawtooth with OpenMPI 4.1.4 as the host MPI



(b) Lemhi with OpenMPI 4.1.1 as the host MPI

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.

by the Office of Nuclear Energy of the U.S. Department of Energy and the Nuclear Science User Facilities under Contract No. DE-AC07-05ID14517. This manuscript has been authored by Battelle Energy Alliance, LLC under Contract No. DE-AC07-05ID14517 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

REFERENCES

[1] Richard S. Canon and Andrew Younge. 2019. A Case for Portability and Reproducibility of HPC Containers. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 49–54. https://doi.org/10.1109/CANOPIE-HPC49598.2019.00012

[2] Docker. 2023. Docker Hub. https://hub.docker.com/

[3] GROMACS. [n. d.]. Hybrid Quantum-Classical simulations (QM/MM) with CP2K interface. https://manual.gromacs.org/documentation/2022-beta1/reference-manual/special/qmmm.html

[4] GROMACS. 2022. GROMACS Container. https://hub.docker.com/r/gromacs/gromacs

[5] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22, 6 (1996), 789–828. https://doi.org/10.1016/0167-8191(96)00024-5

[6] Harbor. 2023. Harbor Open Source Registry. https://goharbor.io/

[7] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

[8] Idaho National Laboratory. 2023. MOOSE Framework. https://mooseframework.inl.gov/

[9] NIST. 2019. CVE-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736

[10] NVIDIA. 2023. NAMD HPC Container. https://catalog.ngc.nvidia.com/orgs/hpc/containers/namd

[11] NVIDIA. 2023. NVIDIA Containers. https://catalog.ngc.nvidia.com/containers

[12] Singularity. 2023. Singularity and MPI Applications. https://docs.sylabs.io/guides/3.5/user-guide/mpi.html

[13] Sylabs. 2023. Sylabs Cloud. https://cloud.sylabs.io/library

[14] Alfred Torrez, Timothy Randles, and Reid Priedhorsky. 2019. HPC Container Runtimes have Minimal or No Performance Impact. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 37–42. https://doi.org/10.1109/CANOPIE-HPC49598.2019.00010

[15] Ohio State University. [n. d.]. OSU Microbenchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/

[16] Yinzhi Wang, R. Todd Evans, and Lei Huang. 2019. Performant Container Support for HPC Applications. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)* (Chicago, IL, USA) *(PEARC '19)*. Association for Computing Machinery, New York, NY, USA, Article 48, 6 pages. https://doi.org/10.1145/3332186.3332226

[17] Christopher Werner. [n. d.]. MCNP User's Manual. https://mcnp.lanl.gov/pdf_files/TechReport_2017_LANL_LA-UR-17-29981_WernerArmstrongEtAl.pdf