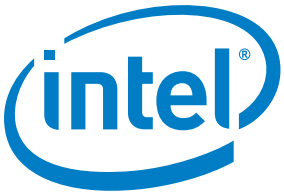


Building Containers for Intel® Omni-Path Fabrics using Docker* and Singularity*

Application Note

October 2017

Document Number: J57474-4.0

Legal Disclaimer

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting: http://www.intel.com/design/literature.htm

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at http://www.intel.com/ or from the OEM or retailer.

No computer system can be absolutely secure.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Contents

Figures

§

Revision History

For the latest documentation, go to http://www.intel.com/omnipath/FabricSoftwarePublications.

Date	Revision	Description
October 2017	4.0	Updated the Docker CE version running on CentOS.
August 2017	3.0	Minor updates for clarification and installation troubleshooting.
April 2017	2.0	Updated “yum install” list and application example.
February 2017	1.0	Initial release.

§

1 Introduction

1.1 Overview

This application note provides basic information for building and running Docker* and Singularity* containers on Linux*-based computer platforms that incorporate Intel® Omni-Path networking technology.

Most of the examples in this document use Docker*, however, other container platforms can be used. For example, HPC administrators may find Singularity* offers advantages for their usage models. However, that decision is for the site administrator to make based on their own considerations, including security, and their specific HPC and/or Cloud use case requirements.

This application note does not purport to offer decision guidance, nor to offer or imply support for any referenced, illustrated, derived, or implied examples.

An experienced reader may wish to go straight to Section 2, for a Quick Start overview. Following the Quick Start example, the rest of this document offers some step-wise limited scope examples that may be helpful when developing containers for your specific operating environments and requirements.

Container technology continues to innovate and evolve. Please consult with your Intel support specialist to discuss how we might be able to address your needs.

1.2 Intel® Omni-Path Architecture

The Intel® Omni-Path Architecture (Intel® OPA) offers high performance networking interconnect technology with low communications latency and high bandwidth characteristics ideally suited for High Performance Computing (HPC) applications.

The Intel® OPA technology is designed to leverage the existing Linux* RDMA kernel and networking stack interfaces. As such, many HPC applications designed to run on RDMA networks can run unmodified on compute nodes with Intel® OPA network technology installed, benefitting from improved network performance.

When these HPC applications are run in containers, using techniques described in this application note, these same Linux* RDMA kernel device and networking stack interfaces can be selectively exposed to the containerized applications, enabling them to take advantage of the improved network performance of the Intel® OPA technology.

More information about Intel® Omni-Path can be found at http://www.intel.com/omnipath/

1.3 Containers

Container platforms, most prominently Docker*, are popular for packaging an application and its dependencies in a virtual container that can run on a computer platform's base operating system, such as Linux*. This helps enable flexibility and portability on where the application can run, whether on premises, public cloud, private cloud, in bare metal, or virtualized hardware environments.

A container provides an environment in which an application, or applications, can run. A container is created out of a container image that is static and has no state. A container image contains an environment that includes a file system, devices, utilities, and an application, or applications, within it. A container is the running instance of a container image. Containers running on a host use services provided by a common underlying operating system, resulting in a light-weight implementation when contrasted to a hypervisor-based VM infrastructure where the entire underlying OS and the applications running in them are replicated for each guest.

1.3.1 Docker* Containers

Docker* is a well-known software containerization platform, with a focus on network service virtualization. It uses the resource isolation features of the Linux* kernel, such as cgroups and kernel namespaces, and a union-capable filesystem, to allow independent containers to run within a single Linux* instance, avoiding the overhead of starting and maintaining virtual machines.

Given the popularity of Docker* as a container technology, the examples in this application note generally describe how to do things based on Docker*. The examples can be useful when working with other container technology as well, as some of those technologies enable importing from Docker* in their technology.

More information about Docker* can be found at https://www.docker.com/

1.3.2 Singularity* Containers

Singularity* is preferred in some circles as a purpose-built alternative to Docker* technology that is better aligned with the needs of High Performance Computing (HPC). Singularity* is designed to allow you to leverage the resources of the host you are on, including HPC interconnects, resource managers, GPUs, and accelerators. Highlighted features include image-based containers, prohibitions against user contextual changes and root escalations, and no root owned daemon process.

This application note shows that configuring MPI is very straightforward in a Singularity* environment. This document also describes how to use container images previously exported from Docker* and import them into Singularity*.

More information about Singularity* can be found at http://singularity.lbl.gov/

1.3.3 Other Container Platforms

There are other container technologies besides Docker* and Singularity* in use. These technologies include:

- LXC: https://linuxcontainers.org/ an early Linux* container technology
- rkt (rocket): https://coreos.com/rkt which runs on CoreOS*
- Shifter project: https://github.com/NERSC/shifter

2 Quick Start

For customers that are familiar with how to configure to run containers on InfiniBand* with Docker* or Singularity*, as a quick start, running on Intel® Omni-Path Architecture (OPA) will already be familiar.

The same standard OpenFabrics Enterprise Distribution (OFED*)/OpenFabrics Alliance* Software interfaces that are used when running over an InfiniBand* link layer, for example, as used for IPoIB, verbs, and RDMA are also used when running over the OPA link layer, so you should find that configuring to run containers on either is similar.

The basic steps are:

1. Install the latest IntelOPA-Basic release drivers and libraries from the Intel support site for your Linux* distribution (for example, RHEL* 7.3), as detailed in the *Intel® Omni-Path Fabric Software Installation Guide*.
2. Decide which container technology is appropriate for your needs, install it, and run it.
   a. For Singularity*, the Intel® Omni-Path device is already available to a running container. Additional steps are not needed to use the Intel® Omni-Path device interface.
   b. For Docker*, add the Intel® Omni-Path device `hfi1_0` in addition to the InfiniBand devices to the run line, as shown in the following example:

```
# ./docker run --net=host
        --device=/dev/infiniband/uverbs0 \
        --device=/dev/infiniband/rdma_cm \
        --device=/dev/hfi1_0 \
    -t -i centos /bin/bash
```

3. Install your application and any additional required user space libraries and software into a container image, and run it.

   User space libraries should include both the InfiniBand* user space libraries and the user space libraries needed to interface with the Intel® Omni-Path driver. In the CentOS* container, install `libhfi1` and `libpsm2`, along with the desired program and its dependencies (for example, `ib_send_bw`).

```
bash# yum install -y libhfi1 libpsm2
bash# yum install –y perftest
```

For a simple recipe, Intel recommends that you use “like on like,” which means running on a standard supported OS distribution, such as CentOS* 7.3, with a container image based on the same kernel and OS distribution, with a matching version of the IntelOPA-basic release used by each, such as IntelOPA-basic release 10.4.2. Other combinations may work, but there is no support implied.

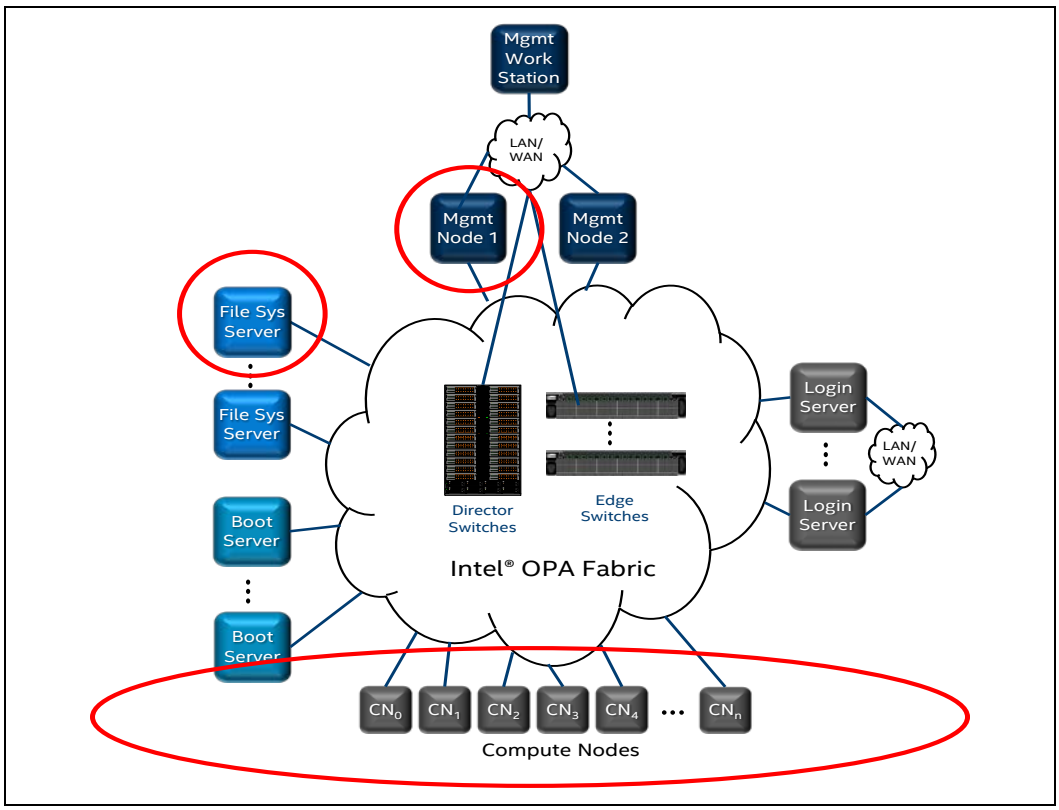
3 Prerequisites

3.1 Identify Compute Nodes to Run Containers

In HPC, groups of nodes in the fabric are often specialized to perform certain functions. Typical specializations are performing compute functions, performing file system/IO functions, and management functions. In some deployments, additional nodes are provided for boot services, and for user login services. A typical fabric configuration illustrating nodes with these specialized roles is illustrated in Figure 1.

Intel® Xeon Phi™ processors may be used in the compute nodes. Other node types are typically implemented using Intel® Xeon technology.

Figure 1. Identifying Compute Nodes for Containerization



Though container technology can be deployed on any node type, for HPC, the emphasis is on containerizing applications and the libraries needed to run on the compute nodes.

A job scheduler, running on a management node, or elsewhere in the system, would schedule the containerized applications to run on available compute nodes. Neither the management nodes nor the I/O nodes need to be containerized.

3.1.1 Compute Nodes

Typically, an application that can be run directly on a host OS, can be included in a Docker* image that can be run in a container on that same host.

The examples included in this application note illustrate some of the steps in how this might be done.

3.1.2 Management Nodes

The cluster's Fabric Manager (FM) runs on management nodes.

The *Intel® Omni-Path Fabric Suite Fabric Manager User Guide* is a useful reference for configuring and managing a fabric. This document assumes a bare metal on OS installation without containerization.

Generally, if you do not need the complication of containerization on this node type, then containerizing this node type is not recommended.

3.1.3 File System / IO Nodes

Storage routers may be implemented on file system/IO nodes.

The *Intel® Omni-Path Storage Router Design Guide* is a useful reference for configuring file system/IO nodes. This document assumes a bare metal on OS installation without containerization.

Generally, if you do not need the complication of containerization on this node type, then containerizing this node type is not recommended.

3.2 Intel® OPA HFI Interface(s)

For a compute node to access the fabric, one or more Intel® Omni-Path Architecture (Intel® OPA) Host Fabric Interfaces (HFIs) need to be installed on the node.

Their presence can be checked for using `lspci`. The following example illustrates a pair of Intel® OPA HFI interfaces present on a host. Based on what is installed in your system, the command output may differ.

```
# lspci | grep Omni
7f:00.0 Fabric controller: Intel Corporation Omni-Path HFI Silicon 100
Series [integrated] (rev 11)
ff:00.0 Fabric controller: Intel Corporation Omni-Path HFI Silicon 100
Series [integrated] (rev 11)
```

More information about Intel® Omni-Path Host Fabric Interface (HFI) options can be found at http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html

3.3 Base Operating System

Containers use kernel services of the base operating system that is installed on a host. For containers to use networking services, those services need to be installed and running in the host OS. To access Intel® OPA networking, the Intel® OPA drivers must be installed and running in the host OS.

Linux* is used as the base OS for the examples shown in this application note. There are multiple distributions of Linux available to users. For this application note, Red Hat* Enterprise Linux* (RHEL*) and CentOS* distributions were used. We chose RHEL*/CentOS* because there are flavors of Intel® OPA releases available from the Intel download site pre-tested for these distributions.

Intel® OPA networking services are included in the kernel. They are available in-box in the RHEL*/CentOS* and SLES* distributions starting with RHEL*/CentOS* 7.3 and SLES* 12SP2, respectively. Users can run with stock versions of these distributions without kernel package modification, if they need to. The requisite packages can be obtained via update from their respective distributions, for example, using `yum update` for RHEL*.

The version of the kernel drivers that are in-box with the distribution typically do not have the latest features, bug fixes, and enhancements that are available from Intel's latest release. Intel recommends that you install the latest kernel updates in the base operating system, before running containers, so that those features can be available. (This is described in Section 3.4.)

Installed packages can be queried using the `rpm -qa` command, and compared against the versions available from Intel's latest release.

To verify that the requisite `hfi1` and `rdmavt` driver modules for Intel® OPA are running (along with other modules from the InfiniBand* core), you can check for them using the `lsmod` command. For illustrative purposes, as a quick check, look for both `hfi1` and `rdmavt` drivers being in use by `ib_core`. Your output may differ from the following sample:

```
$ lsmod | grep hfi1
hfi1                   641484  4
rdmavt                  65351  1 hfi1
ib_core                210381  14
hfi1,rdma_cm,ib_cm,iw_cm,rpcrdma,ib_ucm,rdmavt,ib_iser,ib_umad,ib_uverbs,
rdma_ucm,ib_ipoib,ib_isert
i2c_algo_bit            13413  3 hfi1,mgag200,nouveau
i2c_core                40756  7
drm,hfi1,i2c_i801,drm_kms_helper,mgag200,i2c_algo_bit,nouveau
```

3.4 Installing Intel® OPA Components in Base OS

This section describes how to install the latest updates in the base operating system, before running containers.

3.4.1 Step 1: Read Intel® OPA Release Installation Instructions

The *Intel® Omni-Path Fabric Software Installation Guide* describes OS RPMs installation prerequisites and installation instructions.

Intel® Omni-Path Fabric Software Installation, User, and Reference Guides are available at: www.intel.com/omnipath/FabricSoftwarePublications

As described in the Installation Guide, the installation of the Intel® Omni-Path Software is accomplished using a Text User Interface (TUI) script, named INSTALL, that guides you through the installation process. You have the option of using command line interface (CLI) commands to perform the installation or you can install rpms individually.

3.4.2 Step 2: Obtain and Unpack Latest IntelOPA Release Package

As described in the installation guide from step 1, the latest Omni-Path release (IntelOPA release) can be found on the Intel download center.

Drivers and Software (including Release Notes) are available at: www.intel.com/omnipath/downloads

There are two packages available to download for each distribution: BASIC package and IFS package.

Select the BASIC package for containerizing compute nodes. The IFS package is for management nodes. It includes a Fabric Manager and specialized management node tools that can only operate correctly when run from management enabled nodes.

The filename for the BASIC package uses the format: `IntelOPA-Basic.`*`DISTRO`*`.`*`VERSION`*`.tgz`.

For example: `IntelOPA-Basic.RHEL73-x86_64.10.3.0.0.81.tgz`

Unpack the software on the machine you will be using to build containers and on your compute nodes. Follow the instructions to download and extract installation packages from chapter 3 of the *Intel® Omni-Path Fabric Software Installation Guide* that you obtained in step 1.

The INSTALL script and required RPMs are present in the unpacked folder. We'll use this next for updating OS components, and later for building our Intel® OPA container.

3.4.3 Step 3: Update OS Components Before Running Docker*

You have the option of updating just the kernel components, or both the kernel components and user space components.

Intel recommends that the full install of IntelOPA BASIC components for both the kernel components and user space components is completed on all the compute nodes, independent of what is later placed in the container image to run. This ensures kernel components are updated to the latest revisions and that software needed to support fabric diagnostic tests to be run from the Fabric Manager are present on the

node. It also facilitates running Singularity* containers, by pre-installing MPI and other HPC-related tools and libraries on the compute nodes.

Later in this application note, we show how the user space libraries and components of a release that an application may want to run with can be installed in a container image. When the container image is run, whatever is installed in the container can be used by the application in place of what may (or may not) have been installed on the base OS.

Complete step 3 for your compute nodes by following the instructions to install the Intel® Omni-Path software from chapter 4 of the *Intel® Omni-Path Fabric Software Installation Guide* that you obtained in step 1.

3.5 Installing Docker*

To create and/or run Docker* images as containers on compute nodes, Docker* must be installed on all the compute nodes. Some Linux* distributions already include a version of the Docker* runtime software, or a method to get a version they maintain, from their distro repository. Alternatively, the latest version of Docker* for your OS can be downloaded from the Docker* site at: https://docs.docker.com

3.5.1 Step 1: Consult Docker* Installation Guide

To install Docker* for a given OS, visit the Docker* site for the latest instructions: https://docs.docker.com/engine/installation/

The installation instructions for a particular Linux* distribution get updated often, and should be checked periodically. For example, Docker* recently introduced a choice of Docker* editions that can be installed.

3.5.2 Step 2: Install Docker* on Build and Compute Nodes

Follow the instructions in the Installation Guide you consulted in step 1 to install Docker* on the node you will be using to build container images, and on your fabric's compute nodes.

3.6 Installation Notes

This section describes some of the steps we took for our install of Docker* when following the installation instructions described at the Docker* site. They are included in this application note not as recommendations or instructions for you to follow, but simply to offer illustrative examples that may help you during your install.

For example, we ran into access issues getting the hello-world example to work. If you experience similar problems, it could be because of proxy issues. You can view our notes on how we added proxy access to resolve our issue, recognizing that you will have to tailor a solution for your specific circumstances.

For the following examples, as `root`, install Docker* on a RHEL*/CentOS* 7.3 OS.

Verify existing packages are up-to-date:

```
# yum update
```

Create /etc/yum.repos.d/docker.repo with the following contents:

```
# cat /etc/yum.repos.d/docker.repo
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
```

Install the Docker* engine:

```
# yum install docker-engine
```

Once installed, enable the service to start at boot, and then manually start docker, without a reboot:

```
# systemctl enable docker
# systemctl start docker
```

3.6.1 Docker* Version

The Docker* version running on a system can be determined using the docker version command.

For many of the examples included in this application note, the version we were running with was identified as shown below. We did all our installs and runs on X86 machines.

```
# docker version
Client:
 Version:      1.13.0
 API version:  1.25
 Go version:   go1.7.3
 Git commit:   49bf474
 Built:        Tue Jan 17 09:55:28 2017
 OS/Arch:      linux/amd64

Server:
 Version:      1.13.0
 API version:  1.25 (minimum version 1.12)
 Go version:   go1.7.3
 Git commit:   49bf474
 Built:        Tue Jan 17 09:55:28 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

3.6.2 Hello-World

After installing Docker*, we then ran the Docker* hello-world example to verify the installation. After we resolved our proxy issues, it ran successfully. You should get results similar to what is shown here.

```
# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c04b14da8d14: Pull complete
Digest:
sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working
correctly.
```

3.6.3 Troubleshooting: Adding Proxy

The Linux* section of the Post-Installation instructions on the Docker* site contains a Troubleshooting section that may be useful. You may find you need to add a proxy reference to get the install or hello-world example to run successfully. We found this reference useful: https://docs.docker.com/engine/admin/systemd/#/http-proxy

For example, to get Docker* to communicate with its repository you probably need to create a systemd directory:

```
$ mkdir -p /etc/systemd/system/docker.service.d
```

then create a file:

```
vi /etc/systemd/system/docker.service.d/https-proxy.conf
```

with these contents:

```
[Service]

Environment="HTTP_PROXY=http://proxy.example.com:80/"
```

In addition, you may find it necessary to add proxies to your `/etc/yum.conf` file if you do not already have them:

```
proxy=https://proxy.example.com:80/
```

The Docker* troubleshooting guide can also be useful to resolve other problems as well. For example, we saw this error message when we forgot to run a Docker* command as root:

```
Cannot connect to the Docker daemon. Is 'docker daemon' running on this
host?
```

3.6.4 Troubleshooting: Dependency Resolution Errors During yum Install

As of August 2017, Docker* CE requires the container-selinux-2.9 package. This package is available in the CentOS* extras repository, or it can be downloaded from the centos.org site for your version of CentOS*. Install this before attempting to install docker-engine from the Docker* repository or from a downloaded version of the docker-ce RPM.

4 Creating an Intel® OPA Docker* Container Image

With Docker* installed, you can create customized Docker* images that can be run as containers.

To create a customized image, it is convenient to start with an existing container image, and then build it up by adding libraries and executables to create a new base image. That new base image can then be run as a container, or can be used as the base for creating additional customized containers. This document describes that approach.

In some cases, it may be possible to run a container that was built starting with a base image for an OS distribution that is different from the OS distribution environment that the Docker* engine is running on. For example, building a CentOS*-based Docker* container to run on a SLES*-based system. However, this is not guaranteed to work for all applications.

To minimize complications and confusion, Intel recommends building your customized images from a base OS image that is known to be compatible with the Linux* distribution of the Docker* host you are targeting to run the container on. This type of configuration, called “like on like,” was illustrated in the Quick Start section.

In this example, we used a CentOS* base image for our customized images that will run as containers in Docker* on a RHEL* 7.3 system.

4.1 Step 1: Select a Base Image

We pulled the latest CentOS* base image from the Docker* hub as our base image. We used a syntax similar to this:

```
# docker pull centos
Using default tag: latest
latest: Pulling from library/centos
Digest:
sha256:c577af3197aacedf79c5a204cd7f493c8e07ffbce7f88f7600bf19c688c38799
Status: Image is up to date for centos:latest
```

The pull option `centos` is a synonym for `centos:latest`. You can specify a specific image version using the option `centos:`*`version`*.

An alternative method is to let the image be pulled automatically when using it for the first time in building a Docker* file.

4.2 Step 2: Verify Local Images

We used the `docker images` command to list the images in our Docker* and some information about them.

```
# docker images
REPOSITORY      TAG           IMAGE ID         CREATED        SIZE
hello-world     latest        48b5124b2768     3 weeks ago    1.84 kB
centos          latest        67591570dd29     7 weeks ago    191.8 MB
```

Note that the `centos` image is in the list of images in our local repository. Docker* uses cached versions when available, unless told otherwise.

In this example, image ID `67591570dd29` has been cached in our local repository, without change, for the last 7 weeks. A local Docker* file that builds a new image from `centos` will use this cached image, instead of having to look for and fetch it from the Docker* site.

4.3 Step 3: Create an Intel® OPA Docker* File

This section describes how we created a sample Docker* file that resulted in an Intel® OPA container image with all the libraries and executables normally installed bare metal for an IntelOPA Release. Including all libraries and executables is more than most typical applications will need. A minimal installation may include only the `libhfi1` and `libpsm2` user space libraries, as described in the Quick Start section of this document.

In this example, this containerized Intel® OPA image was used as the base image from which user application images were built.

Refer to the complete reference of Docker* file commands and their syntax at: https://docs.docker.com/engine/reference/builder/

4.3.1 Sample Docker* File

We edited a new file with the name `dockfile`. This file is edited outside of the container. The contents are shown below and a detailed explanation follows.

```
# Create an OPA Image from the latest CentOS base
FROM centos:latest
LABEL maintainer.name=<yourname> maintainer.email=<email@address.com>
ENV container docker

# We need proxy statements to get through our lab's firewall
ENV http_proxy http://proxy-chain.intel.com:911
ENV https_proxy https://proxy-chain.intel.com:911

# Packages needed for a full install of IntelOPA-basic tools
RUN yum -y install perl atlas infinipath-psm libpsm2 libibverbs qperf pciutils \
tcl papi tcsh expect sysfsutils librdmacm libibcm perftest rdma bc which \
elfutils-libelf-devel libstdc++-devel gcc-gfortran rpm-build pci-utils \
iproute compat-rdma-devel libibmad libibumad ibacm-devel libibumad-devel \
net-tools libibumad-static libuuid-devel libhfi1 opensm-libs numactl-libs \
irqbalance openssl kernel && \
yum clean all
```


```
# Here we use a tempdir mapped to the unpacked IntelOPA Release folder
RUN mkdir ./tempdir
COPY . ./tempdir
RUN cd ./tempdir && ./INSTALL –-user-space –n && cd .. && rm -rf ./tempdir
CMD ["/bin/bash"]
```

In this example, certain packages are optional, but have been included to facilitate testing with the example.

You can customize a Docker* file to suit your specific needs. The more items you pull into the container, the larger the resultant image may become, increasing load times the first time it is loaded. (It may get cached by the Docker* runtime after that, speeding load times. See the Docker* documentation for details.)

In the sample Docker* file named `dockfile`, we added the `LABEL` line for illustration purposes. This line is optional, and may be omitted. Update this appropriately for your use.

Also shown are `ENV` statements we needed to work through the proxy server in our lab. You may not need these statements or you may need to customize these for your proxy environment.

In the `RUN yum –y install` portion of the example, we wanted the packages listed to be in the container to support the Intel® OPA additions to the image. These additional packages differentiate our image from the packages already included in the standard RHEL* 7.3 installation we were using. Your application may not need all of these packages or it may need other packages. For example, infinipath-psm may not be needed unless are looking to recompile openmpi in your container. Include all required packages for your application in your file.

These additions are overlaid onto the file system within the container at runtime without modifying anything in the actual file system of the OS that the container is running on. This container feature enables different containers to have their own runtime libraries, and other dependencies, without affecting the base OS or other users on the system. The optional `yum clean all` minimizes the size of the resultant image.

A useful technique to temporarily bring things into the environment while customizing an image is shown via the `RUN` command to `mkdir` to create a temporary directory called `tempdir`, use it, and then remove it. In this case, those files are not permanent members of the image and do not end up in the runtime container.

In this example, the Docker* file will cd into this folder, and then attempt to run the IntelOPA script called `./INSTALL` with the `--user-space` and `-n` options supplied, before exiting the folder, and removing the `tempdir` from the container's filesystem.

Using a `tempdir` and running `./INSTALL` like this is optional. You may wish to be more selective about what gets installed in the container, and use the `–i` option of the INSTALL script to pick a list of just the components you want. Alternatively, you may not use the INSTALL script, and just install the rpms you want manually. This is possible, and is illustrated in the next section.

Continuing with this example, this sample Docker* file is shown ending with a command to run /bin/bash. A different executable could be substituted here which will automatically get executed when the image is run as a container. Alternatively, CMD can be left out of the image, and the docker run command can be used to specify an executable to run from within the container.

Selecting /bin/bash as the executable to run from within the container results in an interactive terminal session being created within the environment of the container. As will be seen later in this application note, when the docker run command is executed from a command line for this image, the command line interface changes from host context to the context of the new container. You have the option to run interactive commands from within the running container, to exit the container leaving it running with the ability to attach to it again later, or to exit the container and stop running.

The contents of the running container can be interactively modified by a user with command line context into the running container. This is useful to layer new libraries and executables into an existing base image to customize a new application image that can then be exported, and run as its own new container type. This is illustrated in the following section.

4.3.2 How to Include Intel® OPA Release Files in an OPA Image

All the user space libraries that are required by an application to be containerized, should be included in the container image. As shown, such dependencies can be added to a container image through use of a Docker* file.

In the example Docker* file, both yum update and ./INSTALL techniques were illustrated to get the additional packages and executables we wanted to include to end up in the container image.

If using ./INSTALL technique to include Intel® OPA release libraries and files in the Intel® OPA image, you'll need to untar the contents of the IntelOPA Release file, and cd into it for use as a tempdir when the Docker* file builds, or in another folder accessible for interactive ./INSTALL, as described in the next section.

4.4 Step 4: Build Intel® OPA Docker* Image Using a Docker* File

The docker build command is used to create a container image from a Docker* file. Refer to: https://docs.docker.com/engine/reference/commandline/build/.

Before we ran docker build based on the dockfile created in the last section, we noted the location of the unpacked latest IntelOPA release we wanted in the custom base Intel® OPA image (see Section 3.4).

Next, we created an empty local folder, and cd into it.

We copied our dockfile into this folder. Alternatively, we could have substituted the path to it when running the following docker build command.

In the following `docker build` example, optionally, you can add `--pull`, to the `docker build` command line to always attempt to pull a newer version, instead of using a version from cache (see Section 4.14.1).

In this example, the resultant build image was called `opa_centos`.

Here's an example of a `docker build` command with the Docker* file you created in a local folder:

```
# docker build -t opa_centos -f dockfile
</path/to/directory/that/has/OPA/release/>
```

Consult the Docker* file reference to explore other options.

4.5 Alternative Example

As described in the Quick Start section, a minimal image includes both the InfiniBand* user space libraries and the Intel® Omni-Path driver user space libraries. In RHEL* 7.3 or later, you can get the in-box version of the user space libraries by installing `libhfi1`, `libpsm2`, and `perftest`, which brings in their dependencies (this includes `libibverbs`, `libibumad`, `librdmacm`, and others). For our base image, we chose to list them, plus bring in a few more packages for convenience of our testing.

In this example, we edited the Docker* file and removed the lines to run ./INSTALL to create an image without MPI sources included. This Docker* file is called `centoslatest_bare`. The following example is for illustrative purposes only.

```
# cat centoslatest_bare
FROM centos:latest
ENV container docker
ENV http_proxy http://proxy-chain.intel.com:911
ENV https_proxy https://proxy-chain.intel.com:911
RUN yum -y install perl libibmad libibumad libibumad-devel librdmacm \
libibcm qperf perftest infinipath-psm libpsm2 elfutils-libelf-devel \
libstdc++-devel gcc-gfortran atlas tcl expect tcsh sysfsutils bc \
rpm-build redhat-rpm-config kernel-devel which iproute net-tools \
libhfi1 && yum clean all
CMD ["/bin/bash"]

# sudo docker build -t opa_centos -f centoslatest_bare .
Sending build context to Docker daemon 2.795 GB
Step 1/6 : FROM centos:latest
---> 67591570dd29
Step 2/6 : ENV container docker
---> Using cache
---> 3dc38729b1ce
Step 3/6 : ENV http_proxy http://proxy-chain.intel.com:911
---> Using cache
---> 5554959e5c01
Step 4/6 : ENV https_proxy https://proxy-chain.intel.com:911
---> Using cache
---> 8cfa5f5879cf
Step 5/6 : RUN yum -y install perl libibmad libibumad libibumad-devel librdmacm
libibcm qperf perftest infinipath-psm libpsm2 elfutils-libelf-devel libstdc++-
devel gcc-gfortran atlas tcl papi expect tcsh sysfsutils bc rpm-build redhat-rpm-
config kernel-devel which iproute net-tools libhfi1 libibverbs pciutils papi rdma
compat-rdma-devel libibumad-static libuuid-devel opensm-libs openssl kernel
irqbalance && yum clean all
---> Using cache
```

```
---> 22470cd0a2fa
Step 6/6 : CMD /bin/bash
---> Running in 5b12c7766ec2
---> c7f926843ecf
Removing intermediate container 5b12c7766ec2
Successfully built c7f926843ecf
[root@myhost Docker]# docker images
REPOSITORY TAG    IMAGE ID     CREATED        SIZE
opa_centos latest c7f926843ecf 55 seconds ago 425 MB
```

If this image is run, it will not have the IntelOPA release installed in it. To add it, we could create a new Docker* file that includes a FROM line that names this image as its base, installs necessary libraries and executables, and then creates a new image from that.

Alternatively, we could interactively add to a container that is running this image, using the terminal interface. This is detailed in Section 5.3 Interactive Modifications to a Running Container.

5 Running Docker* Containers

5.1 Generic Run

Any built image is stored on the machine where it was built, and can be run with a `sudo docker run` command. The container-name or container-id can be obtained from the `docker images` command, previously illustrated.

```
# sudo docker run  -t -i <container-name or container-id>
```

5.2 Example Run

Using the opa_centos example from the previous section, let's illustrate running that image as a container, and then interactively performing an INSTALL of IntelOPA-IFS components into it. Note you can build container images on a different machine from where you intend to run. In this example, the `--device` statements described in the Quick Start section were not included as part of the Docker* run command.

Since the IntelOPA-IFS components were not part of the original image, we exposed a folder from outside the filesystem into the running container, using the `-v` option. For our example:

```
# sudo docker run -v ~/IntelOPA-IFS.RHEL73-x86_64.10.4.0.0.135:/myifs -t
-i opa_centos
```

This command starts the container and mounts the local directory `~/IntelOPA-IFS.RHEL73-x86_64.10.4.0.0.135` onto the directory `/myifs` within the container.

In this example, since there is a `CMD /bin/bash` to run a shell that is included in the build image opa_centos, when you run the image as a container, you end up at a command prompt running in the container, as root. If you then enter the directory `/myifs` from inside the container's filesystem, you see the contents of the folder you mapped during the run command.

For example, if you mapped a folder containing the IFS versions of pre-release 10.4 into `myifs` from outside of the container, your output might look like this:

```
# sudo docker run -v ~/IntelOPA-IFS.RHEL73-x86_64.10.4.0.0.135:/myifs -t -i
opa_centos

[root@40b33be13ba3 /]# cd /myifs

[root@40b33be13ba3 myifs]# ls -l
total 684
-rwxr-xr-x 1 11610 2222 480502 Jan 27 07:20 INSTALL
drwxr-xr-x 4 11610 2222    151 Jan 27 07:09 IntelOPA-FM.RHEL73-x86_64.10.4.0.0.127
drwxr-xr-x 5 11610 2222    231 Jan 27 07:20 IntelOPA-OFED_DELTA.RHEL73-
x86_64.10.4.0.0.128
drwxr-xr-x 8 11610 2222    201 Jan 27 06:24 IntelOPA-Tools-FF.RHEL73-
x86_64.10.4.0.0.128
drwxr-xr-x 2 11610 2222   4096 Jan 27 07:20 OFED_MPIS.RHEL73-x86_64.10.4.0.0.24
-rw-r--r-- 1 11610 2222  95311 Jan 27 07:20 Pre-Release_Notice_v.2.pdf
-rw-r--r-- 1 11610 2222   3237 Jan 27 07:20 README
```

```
-rw-r--r-- 1 11610 2222  87766 Jan 27 07:20
Third_Party_Copyright_Notices_and_Licenses.docx
-rw-r--r-- 1 11610 2222      7 Jan 27 07:20 arch
-rw-r--r-- 1 11610 2222      7 Jan 27 07:20 distro
-rw-r--r-- 1 11610 2222      5 Jan 27 07:20 distro_version
-rw-r--r-- 1 11610 2222     14 Jan 27 07:20 os_id
-rw-r--r-- 1 11610 2222     13 Jan 27 07:20 version
```

5.3 Interactive Modifications to a Running Container

Use of a Docker* file for automated building of container images may be preferred for system administators. However, there is an alternative manual technique that developers and experimenters often use. That technique is to build a custom Docker* image file by running a base image with an interactive shell, as shown in the previous sections, load things into it, exit the container while keeping it running, and then save it into a new Docker* image. This section describes how the manual technique may be used to build an Intel® OPA container image.

5.3.1 Run INSTALL to Add IntelOPA Release to a Running Container

Continuing with the run example, while still in the container's shell, `cd` to the `/myifs` folder, and then execute the `./INSTALL` script there. As described in the installation guide, this brings up a menu. From the menu, you can choose which items you wish to install. Alternatively, if you invoke the script using the `-a` option (`./INSTALL -a`), all user prompts are skipped but you do not have any control over the default options.

```
# ./INSTALL --user-space

Intel OPA 10.3.0.0.81 Software

   1) Install/Uninstall Software
   2) Reconfigure OFA IP over IB
   3) Reconfigure Driver Autostart
   4) Generate Supporting Information for Problem Report
   5) FastFabric (Host/Chassis/Switch Setup/Admin)

   X) Exit
```

1. Choose option 1. This lists the packages that are present.
2. Choose the packages to install (select all the available packages if in doubt) then select option P) Perform selected options.
3. Accept all default configuration options.
4. Choose not to auto-start anything.
5. Exit the installation menu.

Your container now has the IntelOPA Release components installed in it.

Note that certain file system locations do not exist in the container that do exist on the bare metal OS, such as `/boot`. Given that the `INSTALL` was written assuming you are doing an install on the bare metal OS, some scripted install operations will fail when run from within the container. You'll need to examine these failures, to see if any are relevant for the applications you are intending to run.

After the ./INSTALL is complete, you'll want to exit the running container, but keep it running, so that you can then commit it to a new image that you can reuse later.

5.3.2 Exit a Modified Container and Keep it Running

To exit a running interactive container, instead of typing exit, enter the control characters: <Ctrl>+p <Ctrl>+q

You can then use the docker ps command to see the still running containers.

The following example also shows that the new container opa_centos is running concurrent on this system with another container that was started a couple of days ago:

```
[root@40b33be13ba3 myifs]#   (type ctrl-p, ctrl-q to exit the container)

[root@myhost Docker]#
[root@myhost Docker]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
40b33be13ba3        opa_centos          "/bin/bash"         39 minutes ago      Up
39 minutes                              wizardly_lumiere
b529546f239f        centos73ifs104fm    "/opafm_run.sh"     2 days ago          Up
2 days                                  quizzical_bassi
```

5.3.3 Save a Modified Container as a New Image

After exiting an interactive container and keeping it running, use the docker commit command to save a copy of it as a new container image.

For example, to commit the modified container from the previous section as a new image called opa_base, use the following command from outside of the container.

In this example, the <:tag> (:rev1 in this case) is optional:

```
# docker commit 40b33be13ba3 opa_base:rev1
sha256:086690dbeddada6560e916308f9abfb14ee1eb99979bcbd395e920e83d698f12
[root@myhost Docker]# docker images
REPOSITORY      TAG                 IMAGE ID            CREATED             SIZE
opa_base        rev1                086690dbedda        10 seconds ago      925 MB
```

5.3.4 Re-join a Running Container

To re-join an interactive session in a running container, use the docker attach command, with either the container ID, or container name, as shown in the docker ps output.

```
[root@myhost Docker]# docker attach 40b33be13ba3
[root@40b33be13ba3 myifs]#
```

5.3.5 Terminate a Running Container

From outside of the container, you can terminate a running container by issuing a `docker kill <container id>` command, using the `<container id>` shown from the `docker ps` command.

If in an interactive shell of a container, simply type `exit` to both exit and kill the container.

If the container is still visible when using `docker ps -a`, you can use the `docker rm` command to remove it, as described in the Docker* documentation.

6 Saving and Loading a Docker* Image

You can save an image, or export a container, to make it available on other systems.

Docker* offers both save and export command options. An article that explains some of the differences is https://tuhrig.de/difference-between-save-and-export-in-docker/

For this example, we used the `docker save` command to create a copy of an image in a file that can then be distributed to another machine, and then loaded on the new machine.

1. Create a tar file from the image by running the following command:
   ```
   docker save -o centos73ifs.tar centos73ifs
   ```
   In this example, `-o` indicates the tar file, else it copies to STDOUT.
2. Use scp to copy to the other machine.
3. Create the image on the other machine by loading it in from the tar file:
   ```
   docker load -i centos73ifs.tar
   ```
   This adds the container image to that machine.

7 Running Applications in Containers

To run a containerized application, you must have an image that includes that application. A simple way to do this is to start with an Intel® OPA image as a base and then add the necessary application libraries and executables to that image to create a new application image. Methods to do this can be patterned off of the examples in the previous section, where an Intel® OPA image was built from a base CentOS* image.

7.1 Running a Bare Metal Container

Running a bare metal container (a single container image running on a bare metal server) is a simple form of virtualization, where one container at a time per node/server is run. In this environment, there is no conflict between containers on the same server for resources.

To run containerized applications on Intel® OPA, the images that contain these applications must include Intel® OPA components, as previously described in this document. Additionally, at runtime, the container must have access to devices on the host that are supported by drivers in the base OS.

7.1.1 InfiniBand* Devices

Running verbs and RDMA-based applications on OPA, including most benchmark tests and MPI, requires access to the host OS's InfiniBand* devices and the Intel® OPA hfi1 interfaces. This access is granted to a running container via the Docker `--device` run parameter. Some of a host's InfiniBand* devices can be seen by checking the contents of the `/dev/infiniband/` folder.

```
# ls /dev/infiniband
issm0  issm1  rdma_cm  ucm0  ucm1  umad0  umad1  uverbs0  uverbs1

# ls /dev/hfi*
/dev/hfi1_0  /dev/hfi1_1
```

In this example, there are two hfi1 devices on the host, resulting in two ucm, umad, and uverbs interfaces in `/dev/infiniband`. At runtime, you choose which devices are exposed to which running containers. For example, when running a single bare metal container, you may choose to expose one, or the other, or both interfaces to the running container.

7.1.2 Docker* Run Example

The syntax for exposing devices to a running container is illustrated in the following example.

```
# docker run --device=/dev/infiniband/rdma_cm --
device=/dev/infiniband/uverbs0 --device=/dev/infiniband/ucm0 --
device=/dev/infiniband/issm0 --–device=/dev/infiniband/umad0 --
device=/dev/hfi1_0 –-net=host –-ulimit memlock=-1 --ulimit nofile=128 -t
-i opa_image <command in container>
```

In this example, the command `docker run` is used to run the image named `opa_image` as a container. Optionally, `<command in container>` can be specified as the application you want to run from inside the container image, such as `/bin/bash`.

For this example, the InfiniBand* and hfi1 devices associated with the first Intel® OPA port on the host (port 0, hfi1_0) are exposed to the running container. The interfaces associated with the second port on the host (port 1, hfi1_1) were not included as exposed devices in this example, and thus are not visible to the applications of this running container.

For Ethernet interfaces, Docker's default is for all container Ethernet networks to be hidden from the real Ethernet network. Various options are available to tailor this behavior. In this example, the host's view of its Ethernet interfaces are passed through to the container (`--net=host`) so that applications running in the container see the same Ethernet interfaces and addresses on those interfaces that the host sees. Lastly, ulimit adjustments are made to Docker's defaults for `memlock` and number of open files to better support HPC applications.

For details on other `docker run` options, see:
https://docs.docker.com/engine/reference/commandline/run/

7.1.2.1 Ulimit Boundaries

Adjusting ulimit boundaries are some of the `docker run` options available to users. As suggested in the previous run example, `memlock`, and `nofile` adjustments may be made to accommodate your running HPC program.

While running some programs that use IPoIB or verbs such as `ib_send_bw`, the installed IntelOPA release package needs memory or other resources different than the default provided to a container. During our testing with MPI, we found it necessary to adjust the ulimits on memory and open files, using the options shown in the previous sample `docker run` command.

7.1.2.2 Running Privileged

You can run a container in privileged mode (using the option `–-privileged`) to give all capabilities to the container. This also lifts all limitations enforced by the device cgroup controller, effectively allowing the container to do almost everything the host can do. Running in privileged mode may solve some problems for running a bare metal application in a containerized environment, but this is generally not a recommended method. You'll have to evaluate your site-specific needs when

considering to run privileged or not. Other `docker run` options may be more suitable for your application.

7.1.3 Using OPTIONS

As an alternative to specifying `docker run` parameters on the command line, they can be pre-configured for all containers using the Docker* configuration file. This section describes two methods.

Method 1:

For example, you can set default ulimit values for all containers in the Docker* configuration file `/etc/sysconfig/docker` by appending ulimit options to the `OPTIONS` line in this file. Several OS distributions already provide this file. Edit this file to provide your options there as shown in the following example:

```
OPTIONS="--ulimit nofile=256 --–ulimit memlock=-1"
```

You can check if Docker* is using that file in its environment. You can see if your Docker* system service already comes with `EnvironmentFile` definitions by running the command:

```
$ systemctl cat docker
```

If not, then you can create a drop-in file (for example `something.conf`) in the `/etc/systemd/system/docker.service.d` directory. You may need to create this directory if it does not exist. For details on creating this file, see: https://docs.docker.com/engine/admin/systemd/

The drop-in file must have the entries shown in the following example:

```
[Service]
EnvironmentFile=-/etc/sysconfig/docker
ExecStart=
ExecStart=/usr/bin/dockerd –-default-ulimit memlock=-1 –default-ulimit
nofile=256
```

Method 2:

Instead of putting the OPTIONS in the `/etc/sysconfig/docker` file, you can simply put entries like these into the drop-in file. Each option must have a separate entry and must be present before the `ExecStart` command.

```
[Service]
Environment="OPTIONS=$OPTIONS \"--default-ulimit nofile=256\""
Environment="OPTIONS=$OPTIONS\"--default-ulimit memlock=-1\""
```

After creating or modifying a drop-in file while the docker service is running, run the command `systemctl restart docker` to restart the docker service.

7.1.4 Networking Examples

There are a number of container runtime options available when working with Ethernet networking. This section is intended to offer some examples to contemplate when considering your own networking choices.

IPoIB is often used in HPC networking, and is one of the InfiniBand* protocols that users might be interested in using from their containerized applications. IPoIB (also known as IPoFabric), uses in-band methods to provide host-to-host IP-based sockets connectivity.

There are tools that can be used to check in-band verbs and RDMA connectivity between fabric endpoints, like `ib_send_bw`, for example. Tools like `ib_send_bw` rely on being able to establish a control connection over sockets to run successfully, in addition to establishing in-band verbs or RDMA connectivity. Other tools require `ssh` connectivity to operate correctly. Getting the control connection requires some configuration to work correctly in a containerized environment. The following examples demonstrate some configuration methods.

7.1.4.1 Net=host

The underlying devices from the host operating system can be exposed to the container when it is started, as shown by the command below, where `opa_base:rev1` is an example container image that contains the installed IntelOPA release. As described in earlier section of this application note, the option `--net=host` exposes all the host's Ethernet network interfaces to the container.

```
docker run --device=/dev/infiniband/rdma_cm
--device=/dev/infiniband/uverbs0 --device=/dev/infiniband/ucm0
--–device=/dev/infiniband/umad0 --device=/dev/hfi1_0 --–ulimit memlock=-1
–-net=host -t -i opa_base:rev1
```

The test configuration is a small test fabric consisting of two systems with the `ib0` interfaces (IPoIB interfaces) on the hosts statically assigned the IP addresses 11.11.1.1 and 11.11.1.2.

7.1.4.1.1 Ping Example

From an interactive terminal running within the `opa_base:rev1` container, we reached the other host by running the `ping` command from the container's terminal interface command line. In this example, this host has IP address 11.11.1.2, which is assigned to this host's `ib0` interface and is exposed to the container via the `--net=host docker_run` parameter.

For illustrative purposes only, a snippet sample from `ping` initiated from within the container to an external compute node on the fabric that has the IP address 11.11.1.1 assigned to its `ib0` interface is shown here.

```
root@edf69197097b /]# ping –I ib0 11.11.1.1
PING 11.11.1.1 (11.11.1.1) 56(84) bytes of data.
64 bytes from 11.11.1.1: icmp_seq=1 ttl=63 time=0.177 ms
64 bytes from 11.11.1.1: icmp_seq=2 ttl=63 time=0.143 ms
. . .
```

7.1.4.1.2 ssh Example

With a simple `ping` confirmed, we also demonstrated using `ssh` to login from within the container to another host. You may need to install openssh-clients if not already available to your container. With `--net==host`, the login request is handled by the `sshd` that is running on the other host.

```
[root@81439b837f76 /]# ssh 11.11.1.1
root@11.11.1.1's password:
Last login: Thu Dec 15 21:14:02 2016
[root@myhost ~]# exit
logout
Connection to 11.11.1.1 closed.
```

7.1.4.2 Port Mapping

Next, we'll look at an example when the host Ethernet interfaces are not exposed directly to the container.

You can test IPoIB without directly exposing the host networking interface. Instead, the container uses its local bridge device `docker0`, and then locally routes to the right interface.

The `docker run` implementation allows you to specify the host IP address and the host and container port through which the communication can take place. In this example, we enabled `ib_send_bw` testing by passing through the default port number that `ib_send_bw` listens on. We determined this connection port number from the Linux* man page for the `ib_send_bw` command. The default port number that `ib_send_bw` listens on is port number 18515.

Run a container interactively using the `-p port` option, mapping port 18515 on the ib0 IP interface (with address 11.11.1.2) on the host to port 18515 within the container. The following snippet shows a similar example:

```
# docker run –-device=/dev/infiniband/rdma_cm
--device=/dev/infiniband/uverbs0 --device=/dev/infiniband/ucm0
–-device=/dev/infiniband/umad0 --device=/dev/hfi1_0 –-ulimit memlock=-1
-p 11.11.1.2:18515:18515 -t -i centosifsnet
```

In the container on one host, run `ib_send_bw` as the server. In the container on the other host, run `ib_send_bw` as a client. This establishes a control path connection between the client and server, and the requested test runs to completion. A snippet from a sample runtime is shown here.

From inside the container on the first host, run as server.

```
[root@a11d6b11fb74 /]# ib_send_bw

************************************
* Waiting for client to connect... *
```

From inside the container on the second host, run as client, connecting to the host with the server that is running inside the container with the port mapped.

```
[root@c8ad4470e08c /]# ib_send_bw 11.11.1.2
Send BW Test
Dual-port       : OFF          Device         : hfi1_0
Number of qps   : 1            Transport type : IB
Connection type : RC           Using SRQ      : OFF
TX depth        : 128
CQ Moderation   : 100
Mtu             : 4096[B]
Link type       : IB
Max inline data : 0[B]
rdma_cm QPs     : OFF
Data ex. method : Ethernet
---------------------------------------------------------------------------
local address: LID 0x01 QPN 0x0006 PSN 0x3cdb1d
remote address: LID 0x02 QPN 0x000a PSN 0x1e8bf1
---------------------------------------------------------------------------
. . .
```

The output shows `device: hfi1_0` and `Data ex. Method: Ethernet`.

In the next example, we used `rdma_cm` to send the traffic and used the `ib_send_bw -R` option. We also used `net=host` in our container run line in this example. The output shown here is for illustrative purposes only.

```
[~]$  docker run --device=/dev/infiniband/rdma_cm --
device=/dev/infiniband/uverbs0 --device=/dev/infiniband/ucm0 --
device=/dev/hfi1_0 --device=/dev/infiniband/umad0 --rm --net=host --
ulimit memlock=-1 -ti centosifsnet

[/]# ib_send_bw -R 10.228.216.150
---------------------------------------------------------------------------
                    Send BW Test
 Dual-port       : OFF          Device         : hfi1_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 TX depth        : 128
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 0[B]
 rdma_cm QPs     : ON
 Data ex. method : rdma_cm
---------------------------------------------------------------------------
 local address: LID 0x01 QPN 0x000a PSN 0x18c277
 remote address: LID 0x02 QPN 0x000a PSN 0xf95318
---------------------------------------------------------------------------
. . .
```

7.1.4.3 Pipework Overview

Pipework is a script that has been used to provide software-defined network connectivity between containers. Refer to: https://github.com/jpetazzo/pipework

Though Pipework can be used with Docker*, there are updated bridging controls and options in the more recent versions of Docker* containers that deprecates the need for Pipework. See the Docker* documentation for `docker network` and its child commands at: https://docs.docker.com/engine/reference/commandline/network/

Using Pipework offers an interesting example of the use of networking options in the containerized environment, as discussed in the following section.

7.1.4.4 Pipework Example

Run a container without the `-p` or `--net=host` options used previously.

From outside the container, find the container id (or name) of the container, and then use `pipework` to expose the host's ib0 network interface to the container.

```
# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
8050d74567a4        centosifsnet        "/bin/bash"         30 seconds ago      Up
29 seconds                              reverent_mahavira
# /usr/local/bin/pipework ib0 reverent_mahavira  11.11.1.2/24
```

Re-attach, or otherwise get back to the interactive terminal of the running container, and you'll see that the ib0 interface is now visible within the container.

```
$ docker attach reverent_mahavira
[root@8050d74567a4 /]# ifconfig -a
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:2  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 648 (648.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ib0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 2044
        inet 11.11.1.1  netmask 255.255.255.0  broadcast 11.11.1.255
        inet6 fe80::211:7501:165:b0ce  prefixlen 64  scopeid 0x20<link>
Infiniband hardware address can be incorrect! Please read BUGS section in
ifconfig(8).
        infiniband 80:00:00:1A:FE:80:00:00:00:00:00:00:00:00:00:00:00:00:00:00
txqueuelen 256  (InfiniBand)
        RX packets 18  bytes 1337 (1.3 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 17  bytes 1280 (1.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
```

You can selectively expose network interfaces to a container via Pipework. If you have two interfaces or more, and you wish to run two containers each connected to one interface, Pipework can be used. The link to the Pipework web page given at the beginning of this section shows how a container can be bound to a specific network interface. The paragraph on the web page that deals with this is titled "Connect a container to a local physical interface".

In the following example, two containers are running on this machine.

```
$docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
fd8c4c8511a1        centosifsnet        "/bin/bash"         5 seconds ago       Up
4 seconds                               stupefied_hodgkin
df5c1189b97b        centosifsnet        "/bin/bash"         2 minutes ago       Up
2 minutes                               distracted_cray
```

Running `ifconfig` on both the machines shows the same interfaces local to the container.

```
[root@fd8c4c8511a1 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:2  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 648 (648.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@df5c1189b97b /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.3  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:3  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:03  txqueuelen 0  (Ethernet)
        RX packets 16  bytes 1296 (1.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The host has the following interfaces.

```
$ifconfig | grep mtu
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
ens20f1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
ens20f2: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
ens20f3: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
ib0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 2044
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
veth9cb7480: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
vethabc54ce: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
```

Attach the physical interface ens20f1 on the host to the container with the ID fd8c4c8511a1.

```
sudo $(which pipework)  --direct-phys ens20f1 fd8c4c8511a1 11.11.1.1/24
```

Running ifconfig on this container shows us two interfaces eth0 and eth1.

```
[root@fd8c4c8511a1 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:2  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 648 (648.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 11.11.1.1  netmask 255.255.255.0  broadcast 11.11.1.255
        ether 00:25:90:fd:42:5d  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device memory 0xc7200000-c727ffff

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Attach the interface ib0 to the container with the ID df5c1189b97b.

```
$ sudo $(which pipework)  --direct-phys ib0 df5c1189b97b 10.228.216.150/24
```

Run ifconfig within the container.

```
[root@df5c1189b97b /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.3  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:3  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:03  txqueuelen 0  (Ethernet)
        RX packets 16  bytes 1296 (1.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ib0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 2044
        inet 10.228.216.150  netmask 255.255.255.0  broadcast 10.228.216.255
        inet6 fe80::211:7501:165:b0ec  prefixlen 64  scopeid 0x20<link>
Infiniband hardware address can be incorrect! Please read BUGS section in
ifconfig(8).
        infiniband 80:00:00:46:FE:80:00:00:00:00:00:00:00:00:00:00:00:00:00:00
txqueuelen 256  (InfiniBand)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The output shows that the two containers have different physical interfaces attached.

7.1.5 Docker* Containers and MPI

Docker* containers have not yet been fully integrated with MPI-based programs. The biggest hurdle is that MPI uses ssh to log into hosts and it is not recommended that a Docker* container package sshd along with the application that it is expected to run.

There have been attempts at removing the dependency on sshd and instead running programs in pre-loaded containers using existing utilities and an ad hoc mixture of shell scripts. For an example, see: http://www.qnib.org/2016/03/31/dssh/

A whitepaper that discusses the example is: Developing Message Passing Application (MPI) With Docker.

7.2 Running Concurrent Containers

Multiple containers can be run concurrently on the same node/server. This section describes several examples of two instances of the same image running as separate containers on the same system at the same time.

7.2.1 Shared Network Interface

In this example, a single network interface is shared by multiple running containers.

```
[RHEL7.3 myhost]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
e054b7095dac        centos73ifs104      "/bin/bash"         46 seconds ago      Up
45 seconds                          agitated_ritchie
bb16524e2961        centos73ifs104      "/bin/bash"         19 minutes ago      Up
19 minutes                          nostalgic_banach
```

In this example, the same docker run line was executed twice, resulting in two running container instances of the same image. This use case is similar to when multiple instances of the same process are run on a bare metal OS, but in this case it is multiple instances of containers running at the same time on the same system.

7.2.2 Dedicated Network Interfaces

In this example, multiple network interfaces exist on the host. Each running container is pass-through assigned to their own interface.

Run an opainfo command from the OS, outside the containers. The results show two single port interfaces available on the system (hfi1_0, hfi1_1). These ports have separate PortGUID. Both links are Active, with separate LIDs assigned.

```
[RHEL7.3 myhost]# opainfo
hfi1_0:1                          PortGID:0xfe80000000000000:00117501010ce8c9
   PortState:     Active
   LinkSpeed      Act: 25Gb         En: 25Gb
   LinkWidth      Act: 4            En: 4
   LinkWidthDnGrd ActTx: 4  Rx: 4   En: 3,4
   LCRC           Act: 14-bit       En: 14-bit,16-bit,48-bit       Mgmt: True
   LID: 0x00000001-0x00000001       SM LID: 0x00000004 SL: 0
         PassiveCu, 2m   Hitachi Metals    P/N IQSFP26C-20       Rev 02
   Xmit Data:                17 MB Pkts:                35451
   Recv Data:                 4 MB Pkts:                35391
   Link Quality: 5 (Excellent)
hfi1_1:1                          PortGID:0xfe80000000000000:00117501010ce97f
   PortState:     Active
   LinkSpeed      Act: 25Gb         En: 25Gb
   LinkWidth      Act: 4            En: 4
   LinkWidthDnGrd ActTx: 4  Rx: 4   En: 3,4
   LCRC           Act: 14-bit       En: 14-bit,16-bit,48-bit       Mgmt: True
   LID: 0x00000003-0x00000003       SM LID: 0x00000004 SL: 0
         PassiveCu, 2m   Hitachi Metals    P/N IQSFP26C-20       Rev 02
   Xmit Data:                17 MB Pkts:                35276
   Recv Data:                 4 MB Pkts:                35262
   Link Quality: 5 (Excellent)
```

Start each container, from the same Intel® OPA image, but with each container assigned to a different interface.

Run the opainfo command from the container's perspective. The result shows the visible interfaces are constrained to just those exposed to the container at runtime.

```
[RHEL7.3 myhost]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
centos73ifs104      latest              9e08b64791c8        4 days ago
900 MB

[RHEL7.3 myhost]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

[RHEL7.3 myhost]# docker run --device=/dev/infiniband/rdma_cm --
device=/dev/infiniband/uverbs1 --device=/dev/infiniband/ucm1 --
device=/dev/infiniband/issm1 --device=/dev/infiniband/umad1 --device=/dev/hfi1_1 -
-net=host --ulimit memlock=-1 --rm -ti centos73ifs104

[root@myhost /]# opainfo
hfi1_1:1                          PortGID:0xfe80000000000000:00117501010ce97f
   PortState:     Active
   LinkSpeed      Act: 25Gb         En: 25Gb
   LinkWidth      Act: 4            En: 4
   LinkWidthDnGrd ActTx: 4  Rx: 4   En: 3,4
   LCRC           Act: 14-bit       En: 14-bit,16-bit,48-bit       Mgmt: True
   LID: 0x00000003-0x00000003       SM LID: 0x00000004 SL: 0
         PassiveCu, 2m   Hitachi Metals    P/N IQSFP26C-20       Rev 02
   Xmit Data:                17 MB Pkts:                35203
   Recv Data:                 4 MB Pkts:                35189
   Link Quality: 5 (Excellent)
```

Enter `ctrl-p, ctrl-q` to exit this container, but leave it running.

Start up another instance, this time with hfi1_0, instead of hfi1_1 passed through to it.

```
[RHEL7.3 myhost]# docker run --device=/dev/infiniband/rdma_cm --
device=/dev/infiniband/uverbs0 --device=/dev/infiniband/ucm0 --
device=/dev/infiniband/issm0 --device=/dev/infiniband/umad0 --device=/dev/hfi1_0 -
-net=host --ulimit memlock=-1 --rm -ti centos73ifs104

[root@myhost /]# opainfo
hfi1_0:1                          PortGID:0xfe80000000000000:00117501010ce8c9
   PortState:     Active
   LinkSpeed      Act: 25Gb         En: 25Gb
   LinkWidth      Act: 4            En: 4
   LinkWidthDnGrd ActTx: 4  Rx: 4   En: 3,4
   LCRC           Act: 14-bit       En: 14-bit,16-bit,48-bit       Mgmt: True
   LID: 0x00000001-0x00000001       SM LID: 0x00000004 SL: 0
         PassiveCu, 2m   Hitachi Metals    P/N IQSFP26C-20       Rev 02
   Xmit Data:                 17 MB Pkts:                35389
   Recv Data:                  4 MB Pkts:                35330
   Link Quality: 5 (Excellent)
```

Enter `ctrl-p, ctrl-q` to exit this container, but leave it running.

From outside the containers, use the `docker ps –a` command to see each of the containers on the system.

```
[RHEL7.3 myhost]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
bb16524e2961        centos73ifs104      "/bin/bash"         14 minutes ago      Up
13 minutes                            nostalgic_banach
189d7f187297        centos73ifs104      "/bin/bash"         24 minutes ago      Up
24 minutes                            flamboyant_cori
```

7.3 Job Scheduling and Orchestration

Once an application image has been built, it is available to be run. When there is a demand to run a particular application, it must be distributed and scheduled to be run on available compute nodes. In a cluster, scheduling, distribution, and running of container images is typically handled by a job scheduler or orchestration tools.

Popular orchestration tools for containers include:

- Kubernetes: https://kubernetes.io/
- Docker* Swarm: https://www.docker.com/products/docker-swarm
- Intel® HPC Orchestrator: http://www.intel.com/content/www/us/en/high-performance-computing/hpc-orchestrator-overview.html

Examples of how to use these tools is beyond the scope of this application note.

8 Using Singularity* Containers

Singularity* is another container model developed with HPC in mind. The Singularity* container is lightweight and created for mobility. It works very well with MPI and its associated set of control programs like `mpirun` and other resource managers.

8.1 Install Singularity*

Singularity* can be obtained from source from github, and then built and installed locally.

```
$ git clone https://github.com/singularityware/singularity.git
$ cd singularity
$ ./autogen.sh
$ ./configure --prefix=/usr/local --sysconfdir=/etc
$ make
$ sudo make install
```

8.2 Create a Singularity* Container Image

Once Singularity* has been installed, you can either build a container image from scratch or use a Docker* container image to build a Singularity* container image.

The first step is to create an empty Singularity* container image. To create the containers you must run as root. The following command creates a container 2G in size. If you do not specify the size, it is 768M by default.

```
$ sudo singularity create –-size 2048 /tmp/container.img
```

Singularity* is typically installed in `/usr/local/bin`. If the `/etc/sudoers` file does not include `/usr/local/bin` in the defaults, then create the container image file by running:

```
$ sudo $(which singularity) create –-size <size in MB> /tmp/container.img
```

8.2.1 Import a Docker* Container

Once the container is created, you can place the contents of a Docker* container into this empty container image. The quickest way to do this is to start a Docker* container interactively . The following example uses the container in which the OPA IFS package was installed. Since it runs bash as the default command, you must run it interactively or it will exit.

```
$ docker run –t –i centosifsnet
```

Enter `ctrl-p`, `ctrl-q` to exit from the terminal interface and keep the container running.

In this example, the running container has the name `hungry_shirley`. (Use the command `docker ps` to display the name.)

Fill the empty Singularity*container with the contents of the Docker* container.

```
$ docker export hungry_shirley | sudo singularity import
/tmp/container.img
```

8.2.2 Create Singularity* Image from Bootstrap File

It is possible to create a Singularity* container with a base image from the Docker* hub. It is not necessary to have Docker* installed to do this. First create an empty Singularity* container, as previously described, named bootstrap.img. Ensure that the environment variables `http_proxy` and `https_proxy` are set appropriately for your site's configuration to allow `yum install`. Then create a sample bootstrap Singularity* file.

An example bootstrap file (centos73bootstrap) with functionality similar to what had been previously described when creating a Docker* file, is shown here. You do not need to have SINGULARITY_ROOTFS defined ahead of time. This variable is defined when run in the Singularity* environment

```
Bootstrap: docker
From: centos:latest
IncludeCmd: yes

%setup
    echo "Looking in the directory '$SINGULARITY_ROOTFS'"
    echo "Copying the dist files into '$SINGULARITY_ROOTFS'"
    mkdir -p $SINGULARITY_ROOTFS/tempdir
    if [ ! -x "$SINGULARITY_ROOTFS/tempdir" ]; then
        echo "failed to create tempdir directory..."
        exit 1
    fi
    cp -r IntelOPA-Basic.RHEL73-x86_64.10.3.1.0.20/. $SINGULARITY_ROOTFS/tempdir
    if [ ! -x "$SINGULARITY_ROOTFS/tempdir/INSTALL" ]; then
        echo "No INSTALL in directory..."
        exit 1
    fi

    exit 0

%post
    echo "Installing development tools using YUM"
    yum -y install perl atlas libpsm2 infinipath-psm libibverbs qperf pciutils tcl
tcsh
    yum -y install expect sysfsutils librdmacm libibcm perftest rdma bc
    yum -y install elfutils-libelf-developenssh-clients openssh-server
    yum -y install libstdc++-devel gcc-gfortran rpm-buildx
    yum -y install compat-rdma-devel libibmad libibumad ibacm-devel
    yum -y install libibumad-devel libibumad-static libuuid-devel
    yum –y install pci-utils which iproute net-tools
    yum –y install libhfi1 opensm-libs numactl-libs

    cd tempdir ; ./INSTALL --user-space -n ; cd ..; rm -rf tempdir
```

Run the following command (with containerfile 'bootstrap.img' and definitionfile 'centos73bootstrap'):

```
$ sudo -E $(which singularity) bootstrap bootstrap.img centos73bootstrap
```

The Singularity* container file `bootstrap.img` now contains the IntelOPA release installation.

8.3 Running Singularity* Containers

Once the Singularity* container image is created, you can copy this image to another machine and run it. You can run it as a normal user.

Note: You need to be root or sudo only when the container is being created.

8.3.1 Using the Sub-Command exec

The sub-command exec spawns a command within the Singularity* container as if it is running on the host machine. The command is executed within the container but all files, standard input/output/error are accessible by the command.

Shown here is an illustrative example of how you can execute cat within bootstrap.img to print the Linux* version of the base container.

```
$ singularity exec bootstrap.img cat /etc/os-release

NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"
```

If you run the same cat command outside the container, you can see the difference between the container's file system and the host's file system.

```
$ cat /etc/os-release

NAME="Red Hat Enterprise Linux Server"
VERSION="7.3 (Maipo)"
ID="rhel"
ID_LIKE="fedora"
VERSION_ID="7.3"
PRETTY_NAME="Red Hat Enterprise Linux Server 7.3 (Maipo)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:redhat:enterprise_linux:7.3:GA:server"
HOME_URL="https://www.redhat.com/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT="Red Hat Enterprise Linux 7"
REDHAT_BUGZILLA_PRODUCT_VERSION=7.3
REDHAT_SUPPORT_PRODUCT="Red Hat Enterprise Linux"
REDHAT_SUPPORT_PRODUCT_VERSION="7.3"
```

8.3.2 Using the Sub-Command shell

The sub-command `shell` spawns a shell within the container. From the container's shell, you can execute commands within it. Some of the file systems within the container like `/proco` are shared with the host.

Here's an example of one way to check to see if a particular benchmark test was included in your Singularity* image:

```
$ singularity shell /tmp/container.img

Singularity.container.img> cd /usr/mpi/gcc/openmpi-1.10.4-
hfi/tests/osu_benchmarks-3.1.1

Singularity.container.img> ls -la osu_bw
-rwxr-xr-x 1 root root 13803 Nov  1 07:51 osu_bibw
```

8.3.3 Executing a Singularity* Container

You can run or execute a Singularity* container like an ordinary program. This feature may facilitate users utilizing the SLURM* workload manager, or another job scheduler, to run their containerized applications in an HPC environment. If the entry point to a Singularity* container is not specified during the bootstrap phase then `/bin/sh` is executed within a container when it is run. (This is the minimum requirement that is required when building a container.)

```
[user@phsmpriv04 singularityimages]$ ./singtest.img
No Singularity runscript found, executing /bin/sh
. . .
```

To specify a `runscript`, refer to: http://singularity.lbl.gov/docs-bootstrap

8.4 Using mpirun with Singularity* Containers

You can use the host's `mpirun` to run programs that are present in the Singularity* containers that you have created. The container must reside in the same path on all machines. For example, `container.img` can be in the directory `/tmp` on all participating machines.

You can run mpi programs in the containers. An example of launching `mpirun` using a Singularity* container image to run on hosts `phsmpriv03` and `phsmpriv04` is shown below.

```
$ ./mpirun --mca mtl psm2 --mca pml cm -x PSM_DEVICES=self,shm,hfi1  -np
2 -host phsmpriv03,phsmpriv04 singularity exec /tmp/centosifsnet.img
/usr/mpi/gcc/openmpi-1.10.4-hfi/tests/osu_benchmarks-3.1.1/osu_bibw

# OSU MPI Bi-Directional Bandwidth Test v3.1.1
# Size     Bi-Bandwidth (MB/s)
. . .
```

The `singularity exec` command in this example runs the program `osu_bibw`. This program was known to exist in the container, as shown in Section 8.3.2.

8.5 Application Example: NWCHEM

Here is an example of containerizing an MPI application for running on an HPC cluster.

There are other ways to do this, including creating a bootstrap file instead of involving Docker* at all. However, for this example, we chose to illustrate with these essential process steps:

1. Create a dockerfile that containerizes your application.
2. Create a Docker* image from that dockerfile.
3. Run the Docker* image.
4. Create an empty Singularity* container.
5. Export the Docker* container into the Singularity* container.
6. Copy the Singularity* container to all compute nodes that will run the application.
7. Use mpirun, outside of the container, on the headnode to launch the application.

8.5.1 Download the NWChem Source

On the build host, create an empty temporary build directory, `/root/tmp`, to reduce the build context of the container.

To create the NWChem image, download the latest release from http://www.nwchem-sw.org/index.php/Download to the temporary build directory above. In this example, 6.6 was the latest available (`Nwchem-6.6.revision27746-src.2015-10-20.gz`).

8.5.2 Create the dockerfile

Create a dockerfile, `nwchem_build_file`, in the temporary build directory.

Here is an example Docker* build file for NWChem based on one posted to the NWChem website (http://www.nwchem-sw.org/index.php/Special:AWCforum/st/id2181/Docker_container_for_nwchem..html). Please adjust appropriately for your use.

```
FROM        opa_base:rev1
LABEL       maintainer.name="your_name" \
            maintainer.email=your_email@company.com
RUN   yum -y install gcc-gfortran openmpi-devel python-devel \
            rsh tcsh make openssh-clients autoconf automake libtool \
            blas which; yum clean all
WORKDIR     /opt

#Load application from tarball into container
ADD         Nwchem-6.6.revision27746-src.2015-10-20.tar.gz /opt
ENV         NWCHEM_TOP="/opt/nwchem-6.6"
WORKDIR     ${NWCHEM_TOP}

#Install nwchem patches
RUN   curl -SL http://www.nwchem-sw.org/images/Tddft_mxvec20.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Tools_lib64.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Config_libs66.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Cosmo_meminit.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Sym_abelian.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Xccvs98.patch.gz | gzip -d | patch -p0
```

```
RUN   curl -SL http://www.nwchem-sw.org/images/Dplot_tolrho.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Driver_smalleig.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Ga_argv.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Raman_displ.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Ga_defs.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Zgesvd.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Cosmo_dftprint.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Txs_gcc6.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Gcc6_optfix.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Util_gnumakefile.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Util_getppn.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Gcc6_macs_optfix.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Notdir_fc.patch.gz | gzip -d | patch -p0
RUN   curl -SL http://www.nwchem-sw.org/images/Xatom_vdw.patch.gz | gzip -d | patch -p0

#Set the environment variables used to compile your application
ENV          LARGE_FILES=TRUE \
             USE_NOFSCHECK=TRUE \
             TCGRSH="/usr/bin/ssh" \
             NWCHEM_TARGET=LINUX64 \
             NWCHEM_MODULES="all python" \
             PYTHONVERSION=2.7 \
             PYTHONHOME="/usr" \
             USE_PYTHONCONFIG=Y \
             PYTHONLIBTYPE=so \
             USE_INTERNALBLAS=y \
             USE_MPI=y \
             USE_MPIF=y \
             USE_MPIF4=y \
             MPI_LOC="/usr/mpi/gcc/openmpi-1.10.4-hfi" \
             MPI_INCLUDE="/usr/mpi/gcc/openmpi-1.10.4-hfi/include" \
             LIBMPI="-lmpi_usempi -lmpi_mpifh -lmpi" \
             MSG_COMMS=MPI \
             PATH="$PATH:/usr/mpi/gcc/openmpi-1.10.4-hfi/bin” \
             LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/lib64:/usr/mpi/gcc/ openmpi-1.10.4-
hfi/lib64" \
             MRCC_METHODS=y \
             MPIF77=mpifort \
             MPIF90=mpifort \
             MPICC=mpicc \
             FC=gfortran \
             CC=gcc

#See your application instructions for LA subroutine support
#ENV          BLASOPT="-L/usr/lib64 –lblas”

#Compile the application
WORKDIR      ${NWCHEM_TOP}/src
RUN   make clean && make nwchem_config && make

WORKDIR      ${NWCHEM_TOP}/contrib
RUN   ./getmem.nwchem

#Set the environment variables used to run your application
ENV          NWCHEM_EXECUTABLE=${NWCHEM_TOP}/bin/LINUX64/nwchem
ENV          NWCHEM_BASIS_LIBRARY=${NWCHEM_TOP}/src/basis/libraries/
ENV          NWCHEM_NWPW_LIBRARY=${NWCHEM_TOP}/src/nwpw/libraryps/
ENV          FFIELD=amber
ENV          AMBER_1=${NWCHEM_TOP}/src/data/amber_s/
ENV          AMBER_2=${NWCHEM_TOP}/src/data/amber_q/
ENV          AMBER_3=${NWCHEM_TOP}/src/data/amber_x/
ENV          AMBER_4=${NWCHEM_TOP}/src/data/amber_u/
ENV          SPCE=${NWCHEM_TOP}/src/data/solvents/spce.rst
ENV          CHARMM_S=${NWCHEM_TOP}/src/data/charmm_s/
ENV          CHARMM_X=${NWCHEM_TOP}/src/data/charmm_x/
ENV          PATH=$PATH:${NWCHEM_TOP}/bin/LINUX64

#Include a shell for Singularity to use
WORKDIR      /data
CMD          ["/bin/sh"]
```

For more information, the instructions for compiling NWChem are here: http://www.nwchem-sw.org/index.php/Compiling_NWChem.

8.5.3 Build the Docker* Image

While in the temporary build directory, build the Docker* image:

```
# docker build --rm -t centos_opa_nwchem -f nwchem_build_file .
```

8.5.4 Create a Docker* Container and Export it to Singularity*

Run the built Docker* image:

```
# docker run –ti centos_opa_nwchem
(then control-p, control-q to exit the running container)
```

Create an empty 3GB Singularity* image:

```
# singularity create --size 3072 centos_opa_nwchem.img
```

Export the running container into the empty Singularity* image:

```
# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
8f185288130a        centos_opa_nwchem   "/bin/sh"           20 hours ago
Up 20 hours                             gracious_panini
# docker export 8f185288130a | singularity import centos_opa_nwchem.img
```

8.5.5 Copy Container to Compute Nodes and use mpirun to Launch

Copy the containers to each compute node that will run the application. Use the same directory path on each machine for this copy. Then use mpirun on one of the hosts, outside the container, to run the application inside the containers on each of the compute nodes.

This example uses an `mpi_hosts` file containing the names of the hosts and one of the NWChem benchmarks included in the default distribution. Adjust np, and other parameters, as appropriate for your use.

```
mpirun -mca pml cm -mca mtl psm2 -x PSM_DEVICES=hfi1,self,shm  -np 36 --
hostfile mpi_hosts singularity exec /path/to/image/centos_opa_nwchem.img
/opt/nwchem-6.6/bin/LINUX64/nwchem <path_to_your_NWChem_definition_file>
```

8.5.6 Issues Building Containers Using the Sample Build or Bootstrap Files

The sample Docker* and Singularity* container build files (bootstrap files in the case of Singularity) provided in the sections above have been tested. However, it is possible that the user may run into issues building the files due to the base image fetched from the Docker* Hub or due to the host machine configuration. The Docker* Hub base images do change frequently and could cause problems.

In case of an error, please check all error messages to ensure that the right versions of the rpm packages are fetched and check to see if there are any rpm data base errors.

To overcome rpm database errors you can place the command `rpm --rebuilddb` before each `yum` command.

Another method is to ensure the file `/etc/yum.conf` in the container has the entry `plugins=1` and to place the following command `yum install yum-plugin-ovl` before any `yum install` listed in the build file.

9 Conclusions

Container technology can provide improvements in development efficiency, portability, and repeatability. Using containers to run applications on top of hosts connected by the Intel® Omni-Path Fabric may provide certain advantages over running them natively on a host. The containers isolate the applications from specific host OS idiosyncrasies, such as library dependencies that might otherwise be conflicting for different applications that you may want to run on the same host. It allows the host OS to contain only the minimum set of packages needed to run the container and access the Intel® Omni-Path host hardware.

With this isolation, it may be possible to update the host operating system directly and provide improvements in functionality and speed without having to recompile the applications on the host, maintain separate installations of libraries and utilities that different applications require, or provide different Linux* distributions to run applications that are only compatible with those distributions.

In addition, it is possible to maintain the same application using different versions of libraries and attendant software in different containers and deploy the specific container version that is needed for a particular project or to replicate results of previous runs or provide new information using different parameters. This can be done without having to re-install software on the host if all packaging is inside the container.

We demonstrated that both Docker* and Singularity* containers can be built and run on hosts with Intel® Omni-Path interface technology. When comparing the container technologies, we found Singularity* to be a viable alternative to Docker* for running MPI applications in our test HPC cluster environment. Singularity* interfaces with the MPI mechanisms installed on the host machines and can be used with external resource managers. It is also possible to run Singularity* directly as a normal user without needing root permissions to run certain tasks.

As container technology continues to evolve, discussions and debate will continue on the advantages of the available container technologies for different applications in various environments. Please consult with your Intel support specialist to discuss how Intel might be able to address your needs.

§