Singularity and MPI applications

The Message Passing Interface (MPI) is a standard extensively used by HPC applications to implement various communication across compute nodes of a single system or across compute platforms. There are two main open-source implementations of MPI at the moment - OpenMPI and MPICH, both of which are supported by Singularity. The goal of this page is to demonstrate the development and running of MPI programs using Singularity containers.

There are several ways of carrying this out, the most popular way of executing MPI applications installed in a Sigularity container is to rely on the MPI implementation available on the host. This is called the *Host MPI* or the *Hybrid* model since both the MPI implementations provided by system administrators (on the host) and in the containers will be used.

Another approach is to only use the MPI implementation available on the host and not include any MPI in the container. This is called the *Bind* model since it requires to bind/mount the MPI version available on the host into the container.

Note

The *bind* model requires to mount storage volumes into the container to use the host MPI from the containers. This file system sharing between the host and containers is sometimes not an option on high-performance computing platforms. This restriction on some HPC systems is due to the fact that mounting a storage volume would either require the execution of privileged operations, potentially compromise the access restrictions to other users' data or go against mount options of the parallel/distributed file system where MPI is installed.

Hybrid model

The basic idea behind the *Hybrid Approach* is when you execute a Singularity container with MPI code, you will call `mpiexec` or a similar launcher on the `singularity` command itself. The MPI process outside of the container will then work in tandem with MPI inside the container and the containerized MPI code to instantiate the job.

The Open MPI/Singularity workflow in detail:

1. The MPI launcher (e.g., `mpirun`, `mpiexec`) is called by the resource manager or the user directly from a shell.
2. Open MPI then calls the process management daemon (ORTED).
3. The ORTED process launches the Singularity container requested by the launcher command.
4. Singularity instantiates the container and namespace environment.
5. Singularity then launches the MPI application within the container.

6. The MPI application launches and loads the Open MPI libraries.
7. The Open MPI libraries connect back to the ORTED process via the Process Management Interface (PMI).

At this point the processes within the container run as they would normally directly on the host.

The advantages of this approach are:

- Integration with resource managers such as Slurm.
- Simplicity since similar to natively running MPI applications.

The drawbacks are:

- The MPI in the container must be compatible with the version of MPI available on the host.
- The configuration of the MPI implementation in the container must be configured for optimal use of the hardware if performance is critical.

Since the MPI implementation in the container must be compliant with the version available on the system, a standard approach is to build your own MPI container, including the target MPI implementation.

To illustrate how Singularity can be used to execute MPI applications, we will assume for a moment that the application is *mpitest.c*, a simple Hello World:

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

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

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

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

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

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

        MPI_Finalize();
```

Note

MPI is an interface to a library, so it consists of function calls and libraries that can be used my many programming languages. It comes with standardized bindings for Fortran and C. However, it can support applications in many languages like Python, R, etc.

The next step is to build the definition file which will depend on the MPI implementation available on the host.

If the host MPI is MPICH, a definition file such as the following example can be used:

```
Bootstrap: docker
From: ubuntu:latest

%files
    mpitest.c /opt

%environment
    export MPICH_DIR=/opt/mpich-3.3
    export SINGULARITY_MPICH_DIR=$MPICH_DIR
    export SINGULARITYENV_APPEND_PATH=$MPICH_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=$MPICH_DIR/lib

%post
    echo "Installing required packages..."
    apt-get update && apt-get install -y wget git bash gcc
gfortran g++ make

    # Information about the version of MPICH to use
    export MPICH_VERSION=3.3
    export
MPICH_URL="http://www.mpich.org/static/downloads/$MPICH_VERSION/m
$MPICH_VERSION.tar.gz"
    export MPICH_DIR=/opt/mpich

    echo "Installing MPICH..."
    mkdir -p /tmp/mpich
    mkdir -p /opt
    # Download
    cd /tmp/mpich && wget -O mpich-$MPICH_VERSION.tar.gz
$MPICH_URL && tar xzf mpich-$MPICH_VERSION.tar.gz
    # Compile and install
```

If the host MPI is Open MPI, the definition file looks like:

```
Bootstrap: docker
From: ubuntu:latest

%files
    mpitest.c /opt

%environment
    export OMPI_DIR=/opt/ompi
    export SINGULARITY_OMPI_DIR=$OMPI_DIR
    export SINGULARITYENV_APPEND_PATH=$OMPI_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=$OMPI_DIR/lib

%post
    echo "Installing required packages..."
    apt-get update && apt-get install -y wget git bash gcc
gfortran g++ make file

    echo "Installing Open MPI"
    export OMPI_DIR=/opt/ompi
    export OMPI_VERSION=4.0.1
    export OMPI_URL="https://download.open-
mpi.org/release/open-mpi/v4.0/openmpi-$OMPI_VERSION.tar.bz2"
    mkdir -p /tmp/ompi
    mkdir -p /opt
    # Download
    cd /tmp/ompi && wget -O openmpi-$OMPI_VERSION.tar.bz2
$OMPI_URL && tar -xjf openmpi-$OMPI_VERSION.tar.bz2
    # Compile and install
    cd /tmp/ompi/openmpi-$OMPI_VERSION && ./configure --
prefix=$OMPI_DIR && make install
    # Set env variables so we can compile our application
```

Bind model

Similarily to the *Hybrid Approach*, the basic idea behind *Bind Approach* is to start the MPI application by calling the MPI launcher (e.g., *mpirun*) from the host. The main difference between the hybrid and bind approach is the fact that with the bind approach, the container usually does not include any MPI implementation. This means that Singularity needs to mount/bind the MPI available on the host into the container.

Technically this requires two steps:

1. Know where the MPI implementation on the host is installed.
2. Mount/bind it into the container in a location where the system will be able to find libraries and binaries.

The advantages of this approach are:

- Integration with resource managers such as Slurm.
- Container images are smaller since there is no need to add an MPI in the containers.

The drawbacks are:

- The MPI used to compile the application in the container must be compatible with the version of MPI available on the host.
- The user must know where the host MPI is installed.
- The user must ensure that binding the directory where the host MPI is installed is possible.
- The user must ensure that the host MPI is compatible with the MPI used to compile and install the application in the container.

The creation of a Singularity container based on the bind model is based on the following steps:

1. Compile your application on a system with the target MPI implementation, as you would do to install your application on any system.
2. Create a definition file that includes the copy of the application from the host to the container image, as well as all required dependencies.

3. Generate the container image.

As already mentioned, the compilation of the application on the host is not different from the installation of your application on any system. Just make sure that the MPI on the system where you create your container is compatible with the MPI available on the platform(s) where you want to run your containers. For example, a container where the application has been compiled with MPICH will not be able to run on a system where only Open MPI is available, even if you mount the directory where Open MPI is installed.

A definition file for a container in bind mode is fairly straight forward. The following example shows the definition file for NetPIPE-5.1.4 compiled on the host in `/tmp/NetPIPE-5.1.4`:

```
Bootstrap: docker
From: ubuntu:disco

%files
      /tmp/NetPIPE-5.1.4/NPmpi /opt

%environment
      MPI_DIR=/opt/mpi
      export MPI_DIR
      export SINGULARITY_MPI_DIR=$MPI_DIR
      export SINGULARITYENV_APPEND_PATH=$MPI_DIR/bin
      export SINGULARITYENV_APPEND_LD_LIBRARY_PATH=$MPI_DIR/lib

%post
      apt-get update && apt-get install -y wget git bash gcc
gfortran g++ make file
      mkdir -p /opt/mpi
      apt-get clean
```

In this example, the application, NetPIPE-5.1.4, is copied into `/opt`, as a result, the path to the executable to use on the `mpirun` command is `/opt/NPmpi`. Also, this definition file prepares the environment to have the host MPI mounted in `/opt/mpi`; it sets all the required environment variables (PATH and LD_LIBRARY_PATH) for the system to find all MPI binaries and libraries at run-time.

Execution

The standard way to execute MPI applications with hybrid Singularity containers is to run the native `mpirun` command from the host, which will start Singularity containers and ultimately MPI ranks within the containers.

Assuming your container with MPI and your application is already build, the `mpirun` command to start your application looks like when your container has been built based on the hybrid model:

```
$ mpirun -n <NUMBER_OF_RANKS> singularity exec
<PATH/TO/MY/IMAGE> </PATH/TO/BINARY/WITHIN/CONTAINER>
```

Practically, this command will first start a process instantiating `mpirun` and then Singularity containers on compute nodes. Finally, when the containers start, the MPI binary is executed.

For containers built based on the bind model, the command simply needs to include the appropriate bind option:

```
$ mpirun -n <NUMBER_OF_RANKS> singularity exec --bind
<PATH/TO/HOST/MPI/DIRECTORY>:<PATH/IN/CONTAINER>
<PATH/TO/MY/IMAGE> </PATH/TO/BINARY/WITHIN/CONTAINER>
```

Based on the example presented in the previous sub-section, and assuming MPI is installed in `/opt/openmpi` on the host, the command will look like:

```
$ mpirun -n <NUMBER_OF_RANKS> singularity exec --bind
/opt/openmpi:/opt/mpi <PATH/TO/MY/IMAGE> /opt/NPmpi
```

If your target system is setup with a batch system such as SLURM, a standard way to execute MPI applications is through a batch script. The following example illustrates the context of a batch script for Slurm that aims at starting a Singularity container on each node allocated to the execution of the job. It can easily be adapted for all major batch systems available.

```
$ cat my_job.sh
#!/bin/bash
#SBATCH --job-name singularity-mpi
#SBATCH -N $NNODES # total number of nodes
#SBATCH --time=00:05:00 # Max execution time

mpirun -n $NP singularity exec /var/nfsshare/gvallee/mpich.sif
/opt/mpitest
```

In fact, the example describes a job that requests the number of nodes specified by the `NNODES` environment variable and a total number of MPI processes specified by the `NP` environment variable.

The example is also assuming that the container is based on the hybrid model; if it is based on the bind model, please add the appropriate bind options.

A user can then submit a job by executing the following SLURM command:

```
$ sbatch my_job.sh
```