Distributed MPI Cluster with Docker Swarm Mode

Nikyle Nguyen
Department of Computer Science
California State University, Fullerton
Fullerton, CA, USA
nlknguyen@csu.fullerton.edu

Doina Bein
Department of Computer Science
California State University, Fullerton
Fullerton, CA, USA
dbein@fullerton.edu

***Abstract*— MPI is a well-established technology that is used widely in high-performance computing environment. However, setting up an MPI cluster can be challenging and time-consuming. This paper tackles this challenge by using modern containerization technology, which is Docker, and container orchestration technology, which is Docker Swarm mode, to automate the MPI cluster setup and deployment. We created a ready-to-use solution for developing and deploying MPI programs in a cluster of Docker containers running on multiple machines, orchestrated with Docker Swarm mode, to perform high computation tasks. We explain the considerations when creating Docker image that will be instantiated as MPI nodes, and we describe the steps needed to set up a fully connected MPI cluster as Docker containers running in a Docker Swarm mode. Our goal is to give the rationale behind our solution so that others can adapt to different system requirements. All pre-built Docker images, source code, documentation, and screencasts are publicly available.[1]**

Keywords— HPC; MPI; Docker; Docker Swarm mode; Cluster Automation; Container; Distributed System.

I. INTRODUCTION

High-Performance Computing (HPC) is a special and important area in computer science that involves solving big problems that demand high computation capability and fast. HPC is especially popular for simulation tasks, such as weather forecasting, drugs, epidemics, nuclear weapons, or processing tasks, such as genome analytic, search for extraterrestrial intelligence (SETI), or many common Big Data problems. These tasks are too big for a single computer, so HPC often refers to a cluster of computers that work together. In fact, HPC is saving energy because having many smaller machines will require less energy than a very big one. Because of the nature of multiple connected computers, there are various parallel programming models that are developed for coordinating the cluster of computers efficiently and effectively. One popular model is Message Passing for operating communications and collaborating calculations in a cluster of computers.

MPI is the "de facto" standard of message passing in the HPC community. Although it has low (if not zero) tolerance for hardware or network errors, it provides a very flexible and well-defined mechanism for developers to develop distributed program or to build custom frameworks or tools on top of it.

Traditionally, setting up a cluster of computers, for example, an MPI cluster, is a challenging task which requires system administrators to spend considerable time to configure the system and network. However, in recent years, the Docker open source project brings Linux container technology to the spotlight with many important advantages for orchestrating distributed system. Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, and system libraries. This makes sure that the system will work as expected regardless of its environment.

We created a ready-to-use solution for developing and deploying MPI programs in a cluster of Docker containers running on multiple machines, orchestrated with Docker Swarm mode [7], to perform high computation tasks. Originally, this solution was created for teaching "Parallel Computing and Distributed Systems," an upper-division course at California State University, Fullerton.

All source code, documentation, and demo screencasts are available and continue to be updated at a GitHub repository [1]. The pre-built Docker images mentioned in this paper are hosted at a Docker Hub repository [2].

The paper is organized as follows. In Section II we give a background of technologies involved. The project overview that includes software specification is given in Section III. In Section IV and V, we explain the rationale for system design and demonstrate how to use our solution to develop MPI program and to deploy a fully connected MPI cluster as Docker containers running in a Docker Swarm mode that spans on multiple machines. Related work is presented in Section VI. We conclude in Section VII.

II. BACKGROUND

MPI is a well-established standard for a portable message-passing system. Version 1.0 was released in June 1994, and the current version is 3.1 (June 2015) [8]. MPI by itself is just a set of specifications, and it is up to vendors/developers to provide implementations. There are many popular implementations (open source and proprietary) such as MPICH, Intel MPI, or Open MPI (not to be confused with OpenMP). MPI usage is

[1] Doina Bein acknowledges the support by Air Force Office of Scientific Research under award number FA9550-16-1-0257.

978-1-5090-4228-9/17/$31.00 ©2017 IEEE

commonly known in HPC setting where there are large computation problems that demand multiple computers to solve collaboratively. At its heart, MPI is a simplified inter-process communication interface that handles sending and receiving messages in a way that abstracts the underlying network. The purpose of higher abstraction is for developers and scientists to focus on writing parallel code of application without being concerned about any specific system network API thus making source code portable on different systems. MPI is agnostic about network architecture and can be used with any network transport protocol. For HPC number crunching applications, usually the Infiniband [9] or 10-Gigabit Ethernet network are used to interconnect computers in the HPC cluster for faster speed.

Docker is an open-source project that automates software deployment using Linux containers, and it is one of the world's leading software containerization platforms. It runs on most modern Linux distributions, macOS, Windows, and various cloud service providers. It uses built-in Linux Kernel containment features like CGroups, Namespaces, (Unification File System (UnionFS) [3], chroot to run applications in virtual environments called Docker containers. In contrast to virtual machines, which are low-level abstraction technology that virtualizes hardware and requires a secondary guest operating system on top of host operating system, Docker containers are much more lightweight because of the higher level of abstraction that virtualizes only operating system thus doesn't require a real secondary operating system. There are advantages and disadvantages for both, but container technology is thriving to be a much more developer friendly choice in many use cases that once belonged to virtual machines.

A Docker image is the build component of Docker. It is a read-only template from which Docker engine instantiate containers. Each image consists of a series of layers. Docker uses union file systems to combine these layers into a single image. Union file systems allow files and directories of separate file systems to be transparently overlaid, forming a single coherent file system of arbitrary depth. Layering allows Docker to quickly update and distribute only what is needed and not the entire image. A Docker image is defined in a Dockerfile which is a text file containing Docker commands to build an image starting from a base image that could be from local machine or online image registry. After you build a Docker image, you can push it to a public registry such as Docker Hub [4] or a private registry. Docker Hub is a free public image registry for storing and building images.

Docker Swarm mode is Docker's solution for managing a cluster of Docker Engines running on multiple machines called a swarm, where you can deploy application services which are Docker containers. Before Docker Swarm mode, an external orchestration software, such as Kubernetes, CoreOS Fleet, or Mesosphere Marathon, is needed to manage Docker Engines on multiple hosts. There are also many cloud providers that provide container orchestration platform such as Amazon ECS, Azure Container Service, Google Container Engine, etc. However, we choose Docker Swarm mode for two reasons. First, it is entirely free and is not locked into any platform. Second, it is baked in the same Docker daemon and uses similar command syntax. In this paper, we are not concerned about any particular feature difference between Docker Swarm mode and other orchestration tools.

III. Project Overview

With this project, we strive to provide: a consistent development environment for MPI projects (see Section IV) and a simple workflow to deploy MPI applications in a real cluster (see Section V).

The Docker version that we target in this paper is 1.12.1 that comes with Docker Swarm mode, which is our choice of distributed system orchestration method. Interestingly, Docker is the only software required to be installed on a host machine for everything explained in this paper to work. All the steps to be explained are for real multi-host networking, but they can be tested on a single computer as Docker Swarm mode can be formed by a single machine.

We describe next the main software in the Docker image. The actual software or versions should not matter because the main point of the paper is to explain the rationale of the setup so that it can be replicated for different system requirements.

The MPI implementation we use is MPICH because it is known to be the most up-to-date with MPI specification. We chose Alpine as base Linux distribution mainly because of its minimalistic size of only 5 MB. By including essential build tools (such as gcc, make, etc) and MPICH library, the image size is around 150 MB, which is decently small for a fully functional OS image with development toolchain. We developed Dockerfiles [5] to build the base image and the onbuild image [6] (see Figure 1). The two Docker images are prebuilt and hosted at [2], and the Dockerfile source code is in the GitHub repository [1]. The base image can be used as development environment for developing MPI programs interactively. The onbuild image inherits the base image with SSH network setup for cluster; it is intended to be based on in user's custom Dockerfile in order to build a custom image that will be deployed to a cluster. The next section explains the usages of these images in more details.

IV. Consistent MPI Development Environment Across Various Operating Systems

The first goal the project was to give users a way to easily obtain a unified working environment for MPI programming on different operating systems (macOS, Windows, Linux). Although Docker is a native Linux technology, it is available in various forms on macOS and Windows. Users on any system with Docker available can spin up a development environment for MPI programming in their current directory simply by executing this command in a shell:

```
$ docker run --rm -it -v $(pwd):/project
          nlknguyen/alpine-mpich
```

This effectively spins up a Docker container based on the prebuilt Docker image named **nlknguyen/alpine-mpich**, hosted at [2] (See Fig. 1). This is the base image that we referred above. Regardless of the host machine's operating system, as long as it has Docker installed, the Linux container

will function identically. This is assumed that the host machine and the container use the same architecture, in this case, x64. If they have different architectures, for example, the host is x64 and the container is ARM, then it will not work unless we use some kind of machine emulator, such as QEMU[10], but we do not discuss that in this paper.

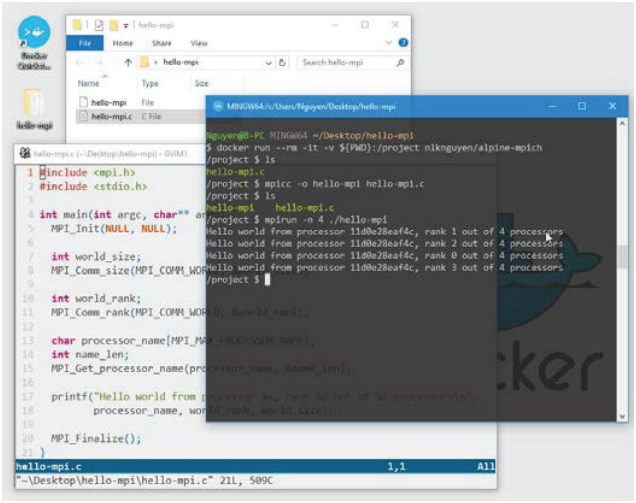

Fig. 1. Spin up container to use interactively as MPI development environment

The argument **-v $(pwd):/project** instructs Docker Engine to mount the current directory in the host machine to the **/project** location in the container. The subshell command **$(pwd)** is evaluated to become the full directory path to the current directory. This, however, assumes it is running in a UNIX / Linux shell like bash or zsh. In case of Windows Command Prompt or PowerShell, the equivalent command to obtain the full path of the current directory is different and will not be discussed in this paper.

The flag **-it** is simply needed to keep STDIN open for interactive usage (like a shell), and flag **--rm** means that the container will be removed after done using it so that every time it is spinned up, it will be a clean state as we expected from the Docker image. If users need different system state, such as install / remove packages, they should create a custom Dockerfile that inherits the Dockerfile hosted at [2], build the custom image, and run from there. This is one of Docker best practices because we want to make sure that every change to the system is declared and therefore reproducible.

V. MPI Cluster Deployment with Docker Swarm Mode

The actual MPI program is independent on this cluster setup. We describe next the steps needed to create a fully connected MPI cluster as Docker containers running in a Docker Swarm that spans on multiple machines. Furthermore, since there are many cloud providers that allow users to provision Docker hosts to form a Swarm easily, such as Docker Cloud, Digital Ocean, Amazon AWS, Joyent Triton, etc., this workflow makes the task of deploying a MPI cluster straightforward on these popular services. Whether the developers target a public cloud provider or internal networked computers, the abstraction of Docker provides a convenient way for us to reason about the distributed system in order to create this setup that helps developing and deploying MPI applications in a cluster effectively.

While writing the paper, we notice that some terms are loosely used as different meanings in different contexts, so here we define some terms that are often used:

- *Swarm*: Docker Swarm mode, which available since Docker version 1.12, not the older Docker Swarm that required external service discovery.
- *Manager node*: a Docker Swarm manager node
- *Worker node*: a Docker Swarm worker node
- *Docker host*: a Manager node or Worker node (part of the Swarm)
- *MPI master node*: a container running as MPI master
- *MPI worker node*: a container running as MPI worker
- *MPI node*: either a MPI master node or MPI worker node

When we mention "master" and "worker" in the same context, we mean MPI master node and MPI worker node.

A. Cluster project

This is an example structure of a MPI project for deploying in Docker Swarm mode. All source code is available in the GitHub repository [1]. We will refer to this as "cluster project".

```
├── Dockerfile
├── project
│   └── mpi_hello_world.c
├── ssh
│   ├── id_rsa
│   └── id_rsa.pub
└── swarm.sh
```

The Dockerfile inherits the prebuilt image that has the basic setup such as non-root user, project directory, and SSH (Secure Shell) config, and the purpose of this Dockerfile is to build the Docker image that contains the compiled MPI program. Below is the entire content of the Dockerfile to build a very simple hello-world MPI program. Of course, it is to be modified to suit each MPI application. Also, there is no need to declare the entry point program because we will assign it later when we create the services.

```
FROM nlknguyen/alpine-mpich:onbuild
COPY project/ .
RUN mpicc -o mpi_hello_world mpi_hello_world.c
```

The onbuild image tag has triggers that will be run when the downstream image, which bases on the onbuild image, is being build. In our case, when the image is built, the content of "ssh" directory within the host directory will be copied over to the Docker image. That directory should have a newly generated private and public SSH keys because they will be used for the communication between the MPI programs running on separate Docker containers, possibly on separate Docker host machines. Moreover, the SSH public key in the host directory will be used to SSH login to the master MPI node in order to run tasks. The key is the only way to connect to the cluster from the outside, and there is no password

whatsoever. We decided to use SSH as the communication channel of MPI cluster because it is common and convenient even though there are alternatives (see Fig. 3).

B. Bootstrapping method

We use the same Docker image for both MPI master node and worker nodes, which offer an advantage of identical system configuration. However, there is a clear difference in the way master node and worker nodes start running. Therefore, we have a shell script program called "mpi_bootstrap" that accepts some argument that indicates whether a node should run as master node or worker node to behave accordingly. We explain how the bootstrap program is called later when we discuss about creating services. Now, let's just focus on the behavior of the master and the worker node. Both should have SSH server daemon running so that they can accept connections from each other. The MPI workers connect to MPI master by using service name of the MPI master node as address. The MPI master probes who is connected and maintains a list of connected addresses. We explain next the difference when bootstrapping.

The MPI master node is where a user login to run some MPI program in the cluster, which requires a way to obtain the addresses of itself and worker nodes in the network. A typical MPI command to run in the master node is something like:

```
$ mpirun -f hosts ./mpi_hello_world
```

where "hosts" is a file that contains the list of addresses of MPI nodes to inform a resource manager the available nodes to distribute the work. To obtain this list, we wrote a short shell script program called "get_hosts" to probe the addresses of established TCP connections through a network program such as *netstat*, and the address of the master node can be obtained in */etc/hosts* file, which is maintained by Docker.

To avoid having to get the list of hosts every time the cluster size changes, we set a default location of this host file so that users don't need to provide it, and we wrote a program to automatically update that file if the cluster changes:

```
$ mpirun ./mpi_hello_world
```

to execute an MPI program across all connected MPI nodes.

At each MPI worker node, we initiate a remote command via SSH to the master node. The goal is only to establish a connection indefinitely so that the master node can probe who is connected. A useless remote command **tail -f /dev/null** is useful enough for this purpose of staying in the master node indefinitely. If the master node goes down by any reason, the connection is lost, and when the master node is spinned up again by Docker Swarm, the worker node should be able to re-establish the connection with the master node.

C. Work with Docker Swarm mode

The whole deployment workflow can be highly automatic. In fact, we developed a utility script **swarm.sh** that wraps multiple Docker commands into simpler command interface, for example **./swarm.sh up size=10** to spin up an entire MPI cluster of size 10, **./swarm.sh scale size=30** to resize the cluster, or **./swarm.sh login** to SSH login to MPI master node. Full documentation and screencasts are in the GitHub repository [1]. Figures 2 and 3 show a sample usage of this script.

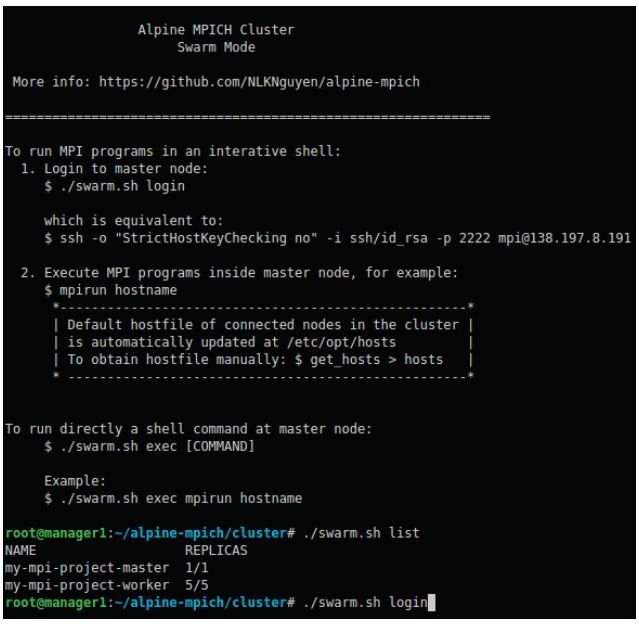

Fig. 2. Use swarm.sh to spin up cluster and login to MPI master node

Fig. 3. SSH session at MPI master node

Despite having that script, we still show how to manually execute some of the actual Docker commands to set up the MPI cluster. In next section, we will input parameters to Docker commands manually, but in actual usage with swarm.sh script, many things are automatic.

Assuming a Docker Swarm mode is already configured (it is quite simple), SSH to the swarm manager node, and put your "cluster project" source code there. Note: if you want to test on

your own machine without having to provision a real cluster, just run the command

```
$ docker swarm init
```

on your own machine, and it becomes a swarm manager node.

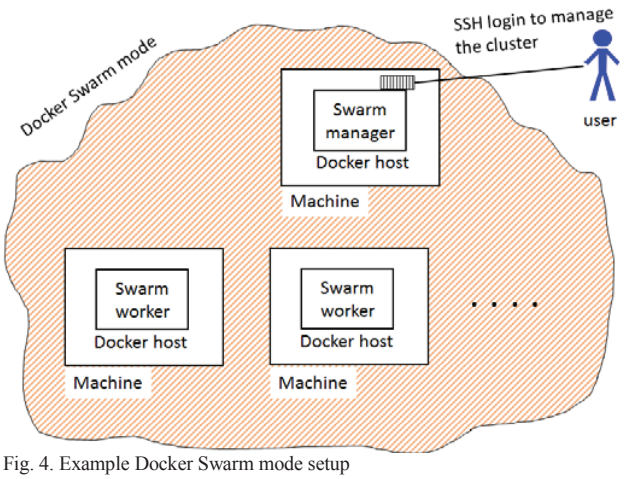

Fig. 4. Example Docker Swarm mode setup

D. Build image and push to image registry

The first step is to build the Docker image then push it to some image registry, whether it is public or private, so that Docker hosts can pull from it. To make things simpler, let's just use Docker Hub free image registry. After running

```
$ docker login
```

to authenticate, it is possible to push the local build of the image to the registry.

```
$ docker build -t nlknguyen/mpi .
$ docker push nlknguyen/mpi
```

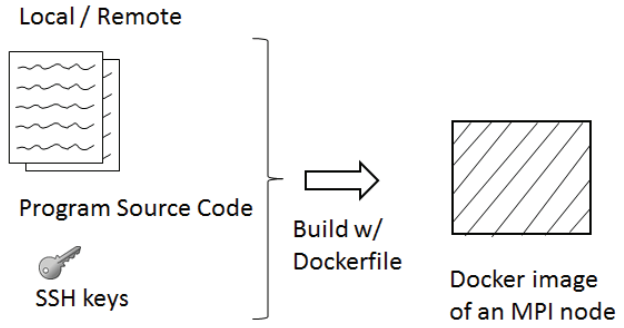

Fig. 5. Building Docker image of an MPI node

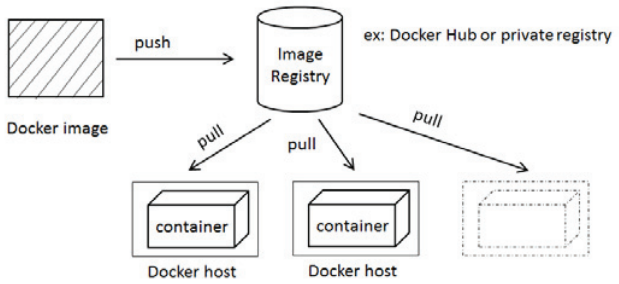

Fig. 6. Using image registry to distribute Docker image to Docker hosts for them to instantiate containers

E. Set up overlay network

Since there are multiple Docker hosts in a Swarm, we need to form an overlay network that spans across multiple nodes so that services running on different hosts but attach to this overlay network can see each other in a single, logical network (see Fig. 7).

Creating an overlay network in Docker Swarm mode is simple:

```
$ docker network create              \
            --driver overlay         \
            --subnet 10.0.9.0/24     \
            --opt encrypted          \
            mpi-network
```

F. Create master and worker services

Now we spin up a MPI master node and multiple MPI worker nodes. All of them must attach to the overlay network that we created previously (also see Fig. 7).

```
$ docker service create              \
     --name mpi-master               \
     --replicas 1                    \
     --network mpi-network           \
     --publish 2222:22               \
     --user root                     \
     nlknguyen/mpi mpi_bootstrap     \
     master_service_name=mpi-master  \
     worker_service_name=mpi-worker  \
     role=master

$ docker service create              \
     --name mpi-worker               \
     --replicas 5                    \
     --network mpi-network           \
     --user root                     \
     nlknguyen/mpi mpi_bootstrap     \
     master_service_name=mpi-master  \
     worker_service_name=mpi-worker  \
     role=worker
```

We specify the image tag as the one we built and pushed in the step before. The entry point of the container is the bootstrap program we already explained earlier. We set the role for master and worker nodes accordingly.

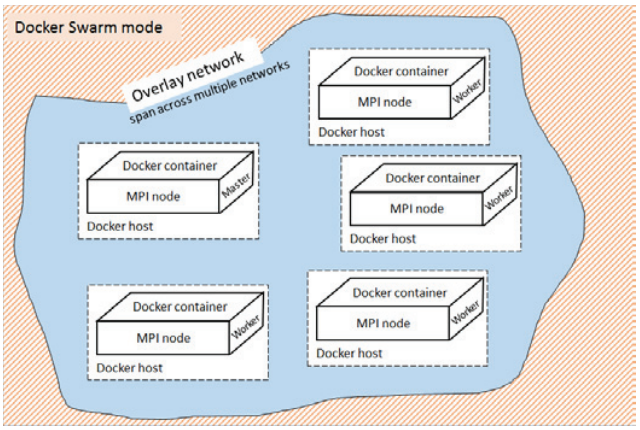

Fig. 7. Spin up containers on multiple Docker hosts in an overlay network within Docker Swarm mode

Since each container is based on the same Docker image, they all have identical public and private SSH keys setup so that they can access to each other (Fig. 8).

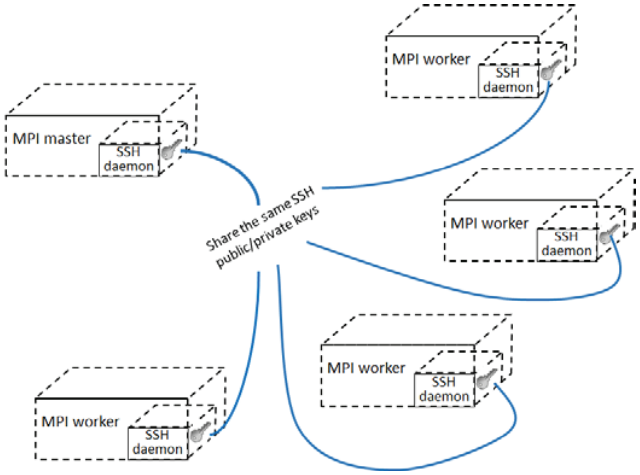

Fig. 8. SSH as the communication channel between MPI nodes.

Also, we need supply the names of the services of the MPI nodes. It is important because those names are what a node can expect the name of another node in the network to talk to. In particular, the MPI worker nodes need to know the service name of master node in order to establish connection there for the MPI master node to probe addresses of connected MPI worker nodes (Fig. 9). Docker Swarm mode includes service discovery that allows this capability of referencing through service names without requiring extra work for us.

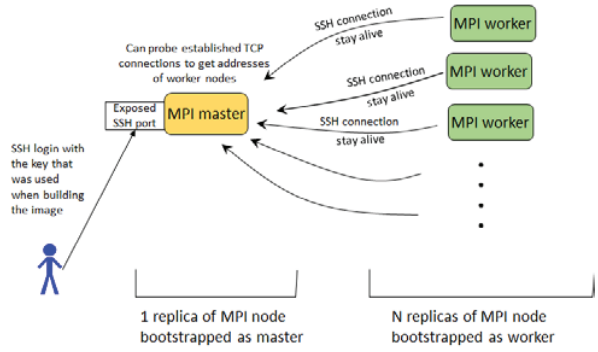

Fig. 9. Bootstrapping MPI master and MPI worker nodes

We also publish the network port 2222 (could be other) which maps to SSH port 22 in the MPI master container so that we can SSH login to the MPI master node via port 2222 at the Swarm manager's IP address. It is the only public port that is exposed to the outside.

In this example, we spin up 6 containers in total, and Docker Swarm mode will automatically balance the containers on the available Docker hosts. In other words, if we have 6 Docker hosts in the swarm, each host will contain one container. There is nothing stopping you from allocating more than one MPI container in a host, but let just say we want to give a container maximum resource that a host can offer.

G. *Login and start MPI program*

When the services are all running, we can SSH login to the MPI master node using the SSH key we have. This can be run within the swarm manager node or from the outside. If it is run from the outside, make sure that the key is available there.

```
$ ssh -o "StrictHostKeyChecking no" -i ssh/id_rsa -p
2222 mpi@<Swarm manager IP>
```

Once inside, we can run the MPI program (see Fig. 10):

```
$ mpirun ./mpi_hello_world
```

```
385c432c0a66:/project$ ls
mpi_hello_world    mpi_hello_world.c
385c432c0a66:/project$ mpirun ./mpi_hello_world
Hello world from processor c520568bf882, rank 3 out of 6 processors
Hello world from processor 601566bfb56f, rank 5 out of 6 processors
Hello world from processor c3cf2c328b0d, rank 4 out of 6 processors
Hello world from processor 385c432c0a66, rank 0 out of 6 processors
Hello world from processor 11732d6bf6a8, rank 2 out of 6 processors
Hello world from processor 5d00dad408c8, rank 1 out of 6 processors
```

Fig. 10. Run MPI program from master node

Remember that we have a default host file location, and it is automatically maintained by one of our shell script programs, we don't need to supply the host file manually to the mpirun command. However, if we want to do that manually:

```
$ get_hosts > hosts
$ mpirun -f hosts ./mpi_hello_world
```

The program will be run in the cluster of Docker containers reside on multiple host machines in the Swarm.

H. Scale cluster

We can even scale the cluster size while having these existing containers running. For example, back to the swarm manager node and run:

```
$ docker service scale mpi-worker=10
```

That will spin up 5 more MPI worker containers, and they will be connected to the MPI master automatically. There is no need to restart the existing containers. Going back to the SSH session at the MPI master node and executing the previous mpirun command again will give work to all 11 containers.

VI. Related Work

The authors of [11] present the major steps for building an HPC cluster using Docker containers. They use Ansible [12] and a resource scheduler SLURM (Simple Linux Utility for Resource Management) [13]. In our paper, we just use the process manager which comes with the MPI implementation that we choose, and that does not require a different set of commands to run MPI jobs.

The authors of [14] present the major steps for building an HPC cluster using Docker containers. Each Docker container has a dynamically allocated IP address. The communication between the Docker containers that belong to the same HPC cluster uses Consul [15], a distributed service discovery to manage the IP assigned dynamically. Their work was before Docker released Docker Swarm mode, which is why they had to use additional tools like Consul. In our paper, we only need to use Docker Swarm mode, which is part of Docker now.

VII. Conclusion

Containerization is indeed a very useful technology because it enables developers to think operationally about their distributed system at an abstraction level that is high enough to reason about.

The beauty of Docker is that it does not require developers to change their application code. The existing MPI code is still the same whether it runs on Docker container, virtual machine, physical machine. The isolation of layers of abstraction is the key to this solution that we created. The abstraction layers go as follow. At the top, MPI library provides a simplified network API that abstracts the actual underlying network API of an operating system. The layer below is Docker container which provides a virtualized operating system that abstracts the actual underlying operating system of a machine. At the bottom, Docker Swarm mode provides a unified orchestration tool that abstracts the underlying network infrastructure of a cluster of machines. These layers of abstractions are distinct and isolated. That means each layer does not need to know what layer underneath it. For example, Docker Swarm mode is just one of many tool for orchestrating containers, and we can surely substitute it with something else.

With the tremendous rise of Docker container technology, creating and managing a distributed MPI cluster for solving high computation problems become much easier to developers and scientists even if they do not have an expensive in-house cluster. Since there are many cloud providers that allow users to provision Docker hosts to form a Swarm easily, such as Docker Cloud, Digital Ocean, Amazon AWS, Joyent Triton, etc., this workflow makes the task of deploying a MPI cluster straightforward on these popular services.

References

[1] https://github.com/NLKNguyen/alpine-mpich

[2] https://hub.docker.com/r/nlknguyen/alpine-mpich

[3] https://en.wikipedia.org/wiki/UnionFS

[4] https://hub.docker.com

[5] https://docs.docker.com/engine/reference/builder

[6] https://docs.docker.com/engine/reference/builder/#/onbuild

[7] https://docs.docker.com/engine/swarm

[8] http://mpi-forum.org/

[9] https://en.wikipedia.org/wiki/InfiniBandG. Eason, B. Noble, and I.N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529-551, April 1955. (*references*)

[10] http://wiki.qemu.org

[11] C. Kniep, “Containerization of High Performance Compute Workloads using Docker”, doc.qnib.org, 2014. [Online]. Available: http://doc.qnib.org/2014-11-05_Whitepaper_Docker-MPI-workload.pdf.

[12] http://www.ansible.com/home

[13] https://computing.llnl.gov/linux/slurm/

[14] H.-E. Yu and W. Huang , “Building a Virtual HPC Cluster with Auto Scaling by the Docker”, arxiv.org, 2015. [Online]. Available: https://arxiv.org/pdf/1509.08231.pdf

[15] https://www.consul.io