International Journal of Trend in Scientific Research and Development (IJTSRD)
International Open Access Journal
ISSN No: 2456 - 6470 | www.ijtsrd.com | Volume - 2 | Issue – 3

# Using Docker for Containerization in High Performance Computing Applications

[1]Pradeep Murugan, [1]Suraj Subramanian, [2]Mr. V Pandarinathan, [3]Dr. D. Rajinigirinath
[1]Student, [2]Assistant Professor, [3]Head of the Department
Department of CSE, Sri Muthukumaran Institute of Technology, Chennai, Tamil Nadu, India

**ABSTRACT**

Virtualization technology plays a vital role in cloud computing. In particular, benefits of virtualization are widely employed in high performance computing (HPC) applications. Containers have a long and storied history in computing. Unlike hypervisor virtualization, where one or more independent machines run virtually on physical hardware via an intermediation layer, containers instead run user space on top of an operating system's kernel. As a result, container virtualization is often called operating system-level virtualization. Container technology allows multiple isolated user space instances to be run on a single host. In this paper we explain how to deploy high performance applications using docker.

***Keywords:*** *Docker, HPC, Virtualization, Coud computing, Hypervisor*

## 1. INTRODUCTION:

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code. In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application. Containers have also been seen as less secure than the full isolation of hypervisor virtualization. Countering this argument is that lightweight containers lack the larger attack surface of the full operating system needed by a virtual machine combined with the potential exposures of the hypervisor layer itself. Despite these limitations, containers have been deployed in a variety of use cases. They are popular for hyperscale deployments of multi-tenant services, for lightweight sandboxing, and, despite concerns about their security, as process isolation environments. Indeed, one of the more common examples of a container is a chroot jail, which creates an isolated directory environment for running processes. Attackers, if they breach the running process in the jail, then find themselves trapped in this environment and unable to further compromise a host.

## 2. Hypervisor

A hypervisor is a program that would enable you to host several different virtual machines on a single hardware. Each one of these virtual machines or operating systems you have will be able to run its own programs, as it will appear that the system has the host hardware's processor, memory and resources. In reality, however, it is actually the hypervisor that is allocating those resources to the virtual machines. In effect, a hypervisor allows you to have several virtual machines all working optimally on a single piece of computer hardware. Now, hypervisors are fundamental components of any virtualization effort. You can think of it as the operating system for virtualized systems. It can access all physical devices residing on a server. It can also access the memory and disk. It can control all aspects and parts of a virtual machine. The servers would need to execute the hypervisor. The hypervisor, in turn, loads the client operating systems of the virtual machines. The hypervisor allocates the correct CPU resources, memory, bandwidth and disk storage space for each virtual machine. A virtual machine can create requests to the hypervisor through a variety of methods, including API calls. native hypervisors run directly on the hardware while a hosted hypervisor needs an operating system to do its work. Which one is better? It depends on what you're after. Bare metal hypervisors are faster and more efficient as they do not need to go through the operating system and other layers that usually make hosted hypervisors slower. Type I hypervisors are also more secure than type II hypervisors. Hosted hypervisors, on the other hand, are much easier to set up than bare metal hypervisors because you have an OS to work with. These are also compatible with a broad range of hardware.

## 3. Containerisation

Docker helps you build and deploy containers inside of which you can package your applications and services. As we've just learnt, containers are launched from images and can contain one or more running processes. You can think about images as the building or packing aspect of Docker and the containers as the running or execution aspect of Docker. A Docker container is:

- An image format.
- A set of standard operations.
- An execution environment.

Docker borrows the concept of the standard shipping container, used to transport goods globally, as a model for its containers. But instead of shipping goods, Docker containers ship software.

Each container contains a software image -- its 'cargo' -- and, like its physical counterpart, allows a set of operations to be performed. For example, it can be created, started, stopped, restarted, and destroyed. Like a shipping container, Docker doesn't care about the contents of the container when performing these actions; for example, whether a container is a web server, a database, or an application server. Each container is loaded the same as any other container. Docker also doesn't care where you ship your container: you can build on your laptop, upload to a registry, then download to a physical or virtual server, test, deploy to a cluster of a dozen Amazon EC2 hosts, and run. Like a normal shipping container, it is interchangeable, stackable, portable, and as generic as possible. With Docker, we can quickly build an application server, a message bus, a utility appliance, a CI test bed for an application, or one of a thousand other possible applications, services, and tools. It can build local, self-contained test environments or replicate complex application stacks for production or development purposes. The possible use cases are endless.

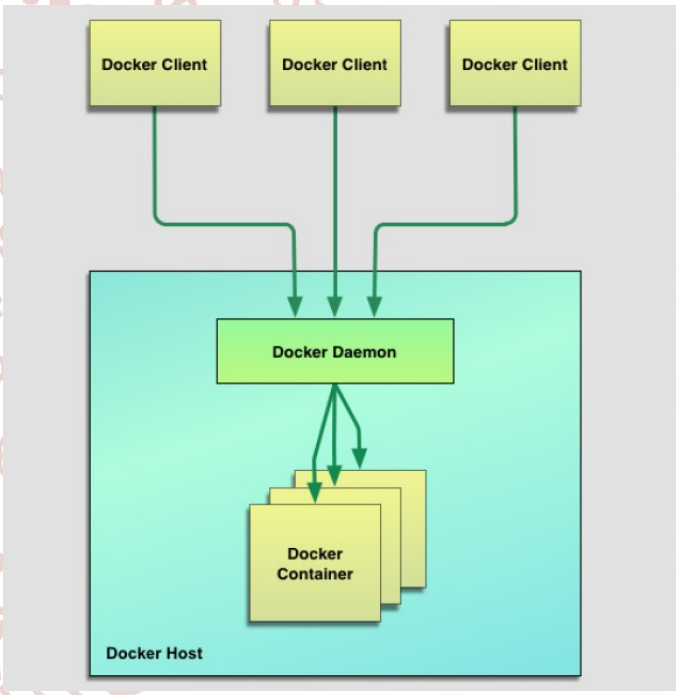

Figure 3.1 Docker Architecture

## 4. Communication between the containers using API

Remote API is provided by the Docker daemon. By default, the Docker daemons binds to a socket, unix:///var/run/docker.sock, on the host on which it is running. The daemon runs with root privileges so as to have the access needed to manage the appropriate resources. If a group named docker exists on your system, Docker will apply ownership of the socket to that group. Hence, any user that belongs to the docker group can run Docker without needing root privileges.

This works fine if we're querying the API from the same host running Docker, but if we want remote access to the API, we need to bind the Docker daemon to a network interface. This is done by passing or adjusting the  H flag to the Docker daemon. On most distributions, we can do this by editing the daemon's startup configuration files. For Ubuntu or Debian, this would be the /etc/default/docker file; for those releases with Upstart, it would be the /etc/init/docker.conf file. For Red Hat, Fedora, and related distributions, it would be the /etc/sysconfig/docker file; for those releases with Systemd, it is the /usr/lib/systemd/system/docker↩ .service file. Let's bind the Docker daemon to a network interface on a Red Hat derivative running Systemd. We can then use the Docker.url method to specify the location of the Docker host we wish to use. In our code, we specify this via an environment variable, DOCKER_URL, or use a default of http://localhost:2375. We've also used the Docker.options to specify options we want to pass to the Docker daemon connection. We can test this idea using the IRB shell locally. Let's try that now. You'll need to have Ruby installed on the host on which you are testing. Let's assume we're using a Fedora host.

## 5. Experimental results for docker

With the model of deploying distributed applications on Docker containers, we investigate experiments about practical applications. Concretely, we use computing intensive and data intensive applications, namely High Performance Linpack (HPL) and Graph500. Our target is to test the efficiency of HPC applications deployed on Docker containers by the sharing model and compare to VMs.

### 5.1 Testing environment

In this paper, we set up an evaluation environment with the limitation of unimportant services to reduce overhead. This means that there are merely pure OS and related dependencies on the verified system. We use native performance as a standard to evaluate the overhead of virtualized environment. Objectively, we propose many benchmark scenarios with different configurations on Docker containers and VMs.

| The number of instances | Virtual machine (vCPU, vRAM, execution processes) | Docker (execution processes) |
|---|---|---|
| 2 | 16, 32, 16 | 16 |
| 4 | 8, 16, 8 | 8 |
| 8 | 4, 8, 4 | 4 |
| 16 | 2, 4, 2 | 2 |
| 32 | 1, 2, 1 | 1 |

Table 5.1.1 Scenarios of configuration

The resources allocated in VMs or containers have to saturate with the resources of system under test, e.g. RAM, cores. We implement all of tests on Intel System with two compute nodes. There are 16 physical cores totally (along with Hyper Threading technology) with Intel Xeon CPU E5-2670 @ 2.6GHz and 64 GB of RAM. Between two compute nodes, the network equipped under system is 1Gbps Gigabit Ethernet. For consistency, the OS that we use is CentOS 7 64-bit 3.10.0 on both of physical and virtual environment. In further, VMs and Docker containers are deployed by the latest version such as QEMU 2.4.0.1/KVM and Docker 1.7.1 respectively. Our testbeds use full of resources (e.g. CPU, RAM) of physical system with the problem size is equivalent to the capacity of system under test. First, we target the reasonable sizes of problems which our system can obtain the best performance on each environment. Second, we benchmark on many configurations of VMs and Docker containers with the increasing number of generated instances. This allows to observe the changes of performance when we run more than one virtual instance. The VMs or Docker containers are deployed with different quantities. We have 64GB of RAM and 32 logical cores which the real system can allocate for virtual environment. Table I shows the configurations of VMs and Docker containers with the difference of each case, being the number of generated instances among the values of CPU, RAM and execution processes.

### 5.2 Docker container with computing intensive application

Computing intensive application is represented by HPL benchmark. This is a portable implementation of Linpack benchmark.

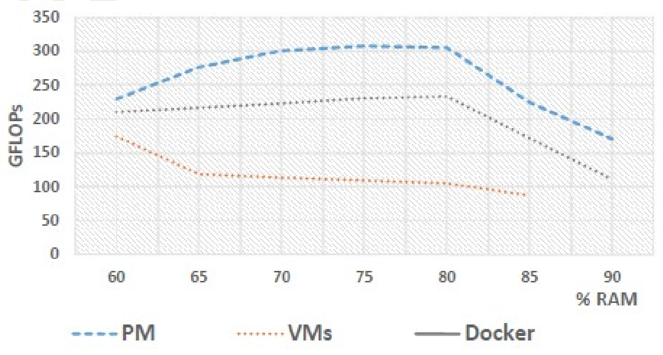

Figure 5.2.1 HPL benchmark with different problem sizes

The problem of HPL is to generate, solve, check and time the random dense linear system of equations.

This is a familiar problem which its size can scale up or down. HPL uses double-precision floating point arithmetic and portable routines for linear algebra operations, message passing. Afterwards, the benchmark counts floating-point operations per second and returns performance results. To run an application as HPL benchmark on distributed system, we need three main parts including math library, message passing interface (MPI) and HPL package.

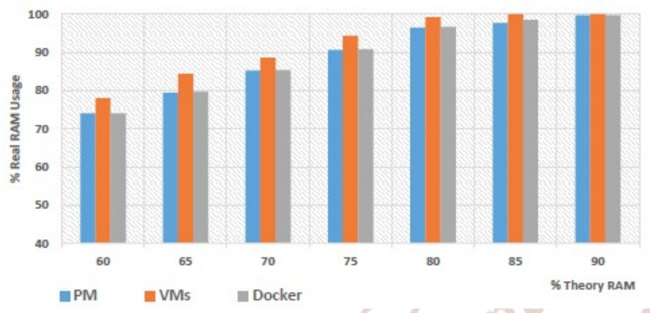

Figure 5.1.2 Cost of memory usage

we configure HPL inside each Docker container, but math library and MPI are mounted from the host OS. Concretely, we use OpenBLAS and OpenMPI representing math library and MPI, which they are installed under host. As figure 3, because each Docker container performs as a process inside host OS,all of libraries and dependencies being necessary for HPL are mounted directly to container. This facilitates to create multiple containers on a node. For VMs, HPL needs to setup along with OpenBLAS and OpenMPI inside each instance.

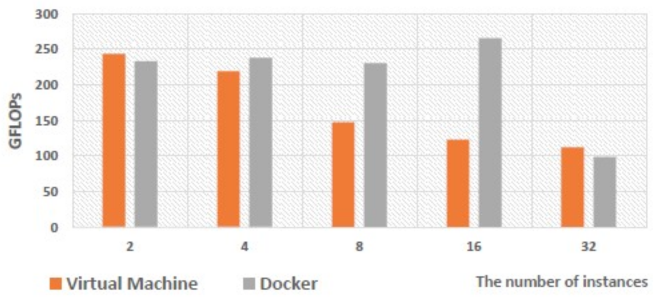

Figure 5.1.3 HPL results in VM and docker containers

Hence, the overhead proliferates over the increasing number of generated VMs.

The problem of HPL is matrix size and it is one of parameters directly affecting to computing performance. First, we construct a scenario with a rage of matrix sizes between VMs and Docker containers. Matrix sizes stretch from 60%to 90% of RAM in system under test. In this case, figure 4shows the efficiency of Docker container when running HPC application based on our deployment method in section V. The native performance is the best result and it is considered as a based case, accounting for roughly 308 GFLOPs in figure3. HPL performance of Docker container is better than virtual machine, especially, the matrix size occupies from 75% to 80%of RAM is suitable for Docker. By contrast, VMs obtain the better performance in matrix size being smaller than 65% of RAM. Additionally, HPL results cannot be returned if the size is over 85% of RAM. Having analyzed the real cost of memory that physical system has to pay, figure 5 compares the real memory utilization between VMs and Docker containers when performing HPL benchmark. The percentage of RAM usage on VMs is larger than containers, leading to the overhead increases. Therefore, when the matrix size scales up, the performance gets lower.

## 6. Conclusion

Docker provides the fundamental building block necessary for distributed container deployments. By packaging application components in their own containers, horizontal scaling becomes a simple process of spinning up or shutting down multiple instances of each component. Docker provides the tools necessary to not only build containers, but also manage and share them with new users or hosts. While containerized applications provide the necessary process isolation and packaging to assist in deployment, there are many other components necessary to adequately manage and scale containers over a distributed cluster of hosts.

## References

1) C. Boettiger, “An introduction to docker for reproducible research,” ACM SIGOPS Operating Systems Review, vol. 49, no. 1, pp. 71–79, 2015.

2) J. J. Dongarra, P. Luszczek, and A. Petitet, “The linpack benchmark: past, present and future,” Concurrency and Computation: practice and experience, vol. 15, no. 9, pp. 803–820, 2003.

3) R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, “Introducing the graph 500,” Cray Users Group (CUG), 2010.

4) R. Morabito, J. Kjallman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in Cloud Engineering

(IC2E), 2015 IEEE International Conference on. IEEE, 2015, pp. 386–393.

5) J. E. Smith and R. Nair, "The architecture of virtual machines," Computer, vol. 38, no. 5, pp. 32–38, 2005.

6) J. Hwang, S. Zeng, F. Y. Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on. IEEE, 2013, pp. 269–276.

7) W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in Proceedings of the 20th annual international conference on Supercomputing. ACM, 2006, pp. 125–134.

8) R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014, pp. 610–614.

9) (2014) Docker homepage. [Online]. Available: https://www.docker.com/

10) C. P. Wright and E. Zadok, "Kernel korner: unionfs: bringing filesystems together," Linux Journal, vol. 2004, no. 128, p. 8, 2004.

11) A. M. Joy, "Performance comparison between linux containers and virtual machines," in Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in. IEEE, 2015, pp. 342– 346.

12) S. Soltesz, H. P¨otzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, highperformance alternative to hypervisors," in ACM SIGOPS Operating Systems Review, vol. 41, no. 3. ACM, 2007, pp. 275–287.

13) S. G. Soriga and M. Barbulescu, "A comparison of the performance and scalability of xen and kvm hypervisors," in Networking in Education and Research, 2013 RoEduNet International Conference 12th Edition. IEEE, 2013, pp. 1–6.

14) C. Pahl, "Containerization and the paas cloud," IEEE Cloud Computing, no. 3, pp. 24–31, 2015.

15) W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. IEEE, 2015, pp. 171–172.