DCSPARK: VIRTUALIZING SPARK USING DOCKER CONTAINERS

Zhou Lei,Hongguang Du, Shengbo Chen,Caixin Zhu, Xianyang Liu

School of Computer Engineering and Science, Shanghai University, Shanghai 200436, P. R. China

ABSTRACT

As MapReduce has become a popular model for large-scale data procession in recent years, companies and researchers take advantage of this model to solve their problems. The applications may run on the same MapReduce cluster, with their own system-wide configure settings and library dependencies, respectively. Sometimes, their configure settings and library dependencies are conflicted with each other. How to ensure these applications to run together correctly without mutual interference and achieve high resources utilization gives a challenge to the researchers. In this paper, we propose *DCSpark*, a framework that leverages the power of Docker containers that allows users to run Spark applications which have conflicting configurations and library dependencies in one physical cluster. In addition, it's presented an implementation of our framework called *DCM* which is aimed at managing the physical cluster, processing scheduling problem and building the container-based Spark cluster images automatically according to the dependence environment of the applications. Our experimental evaluation shows that *DCSpark* introduces negligible overhead for CPU and memory performance compared with the native Spark cluster.

Index Terms— MapReduce, Container-based Spark, Cluster scheduling

1. Introduction

In recent years, the MapReduce [1] as a parallel computing framework has emerged as an important parallel acceleration for the traditional algorithm. MapReduce is good at processing large amount of data in parallel, which is to breakdown the large input into smaller chunks and each can be processed separately on different machines [2]. The Spark infrastructure helps programmers out of the complexity of distributed computing and the problem of the machine failure, data availability and coordination. Howerver, solving the application runtime environment issues under one physical cluster has always been a difficult task for both the programmers and researchers.

In traditional, lots of algorithms run on the single computer and their dependence packages and configurations can be isolated by the virtual machines. The past studies have shown that the traditional hypervisor-based virtualizations (such as Xen, VMware and KVM) have a high performance overhead[3]. The container-based virtualization implementations offer a lightweight virtualization layer which promises a near-native performance. The paper [4] shows that containers are equal to VMs or better performance than VMs in almost all cases. The resource virtualization and management not only can make the isolations among each different tasks, and can increase the system's performance and efficiency by running a mix of workloads in a shared cluster. The paper [3] found that all container-based systems have a near-native performance of CPU, memory, disk and network in HPC cluster.

LXC (Linux Containers[5]) is designed to provide OS-level virtualization method of a shared kernel. It allows a number of other processes running in a sandbox relatively independent space, and can easily control their resource scheduling. Since without repeated loading kernel and kernel shared with host, LXC can greatly speed up the start-up process and significantly reduce memory consumption. In the actual test, the performance of LXC based virtualiztion is closed baremetal system. Docker[6] was using LXC earlier, and switched to runC, which runs in the same operating system as its host. This allows it to share a lot of the host operating system resources. The container has similar resource isolation and allocation benefits as virtual machines but a different architectural approach allows users to be much more portable and efficient.

Spark[7] is a general parallel framework similar to the Hadoop from UC Berkeley AMP lab. Spark's parallel operations fit into the MapReduce model. However, it operates on RDDs (Resilient Distributed Dataset) that can persist across operations. Spark has an advanced DAG(Directed Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing. It can run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

In this context, we use Docker-based virtualization to build the Spark cluster as *DCSpark Cluster Image* for multiple runtime environment types tasks automatically which the library dependencies and configurations are isolated in one physical cluster. The architecture of virtualization cluster images is similar to Spark, including master, worker and driver. In addition, It's presented an implementation of our framework called *DCM* which is able to manage the physical cluster and build the container-based Spark cluster images automatically according to the dependence environment of the applications and processes scheduling problem. To support our claim, we would like to illustrate three usage scenarios:

Custom environments: Multiple applications running in the shared clusters must be isolated for that they have different requirements in terms of software package. Precise isolation between applications's CPU, IO, Memory.

Auto build: Our framework provides for reliable and frequent container image build and deployment with quick and easy rollbacks.

Unified resource management: Different applications can share resources which increases the system's performance and efficiency by running a mix of workloads on the same machine: CPU- and memory-intensive ,and small and large ones.

To evaluate and demonstrate our framework, we implement some benchmarking experiments and two algorithms which has different requirements in terms of library dependencies and configuration in one physical cluster. Some benchmarking experiments are used to compare the performance between our framework and the native cluster. We implement a face recognition system and a video synopsis system to test the isolation of our framework , which the former system relies on OpenCV 3.1[8] and the latter relies on OpenCV 2.3.0[9].

The paper is organized as follows. Section 2 introduces the related work. Section 3 and 4 describe the architecture and current implementation of our framework. Section 5 reports experiments with our *DCSpark* solution and compares its performance with native Spark cluster. Finally, we conclude our current work and introduce future work in Section 6.

2. Related Work

Large-scale compute clusters are expensive, so it is important to use them well. The resource virtualization and resource management not only can make the isolation each other different tasks, and can increase the system's performance and efficiency by running a mix of workloads on the same machine. The traditional virtual machine and container-based virtualization are the current popular virtualization solutions. Many cluster manager architectures use container-based virtualization to abstract and schedule cluster resources in the field of distributed computing.

The *virtual machine* have become very popular in recent years, bringing forth several software solutions (such as Xen, VMware and KVM) and the incorporation of hardware support in commodity processors (such as Intel VT and AMD-V). The main benefits of virtualization include hardware independence, availability, isolation and security. However, the use of virtualization has been traditionally avoided in most HPC facilities because of its inherent performance overhead [3][10].

The *container-based virtualization* implementations (such as Linux-VServer, OpenVZ and LXC) offer a lightweight virtualization layer, which promises a near-native performance. They are appropriate for many usage scenarios especially data center and compute cluster that require system virtualization with high degrees of both isolation and efficiency[11]. Our research provides strong justification in using containers over virtual machines to virtualize Spark.

Apache Mesos[12] is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI with container. It introduces DRF[13] scheduling algorithm and a distributed two-level scheduling mechanism. Mesos can achieve near-optimal data locality when sharing the cluster among diverse frameworks, can scale to 50,000 (emulated) nodes, and is resilient to failures. While Mesos can't run multiple container clusters of the same framework on the compute cluster for it is used to managed resources among frameworks and isolate them.

Kubernetes[14] is an open-source system for automating deployment, operations, and scaling of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes increased the concepts of *Pods*, *Services* and *Replication Controllers* which bear the container management main function and make the Kubernetes can be able to run a variety of applications. While using Kubernetes construct multiple Spark clusters is a very complicated process and and there is no relevant optimization in scheduling for the cluster task.

3. DCSpark Cluster Image Architecture

Our framework consists of two parts, one is the *DCSpark Cluster Image*, the other is *DCM* which is responsible for the resource management and scheduling of the cluster. In this section, we briefly introduce the function of the *DCSpark Cluster Image*. In the next section, we will introduce the *DCM* construction and implementation. The architecture of the *DCSpark Cluster Image* is similar to Spark, including master, worker and driver. These components are composed of Docker images and the corresponding operating environment. The specific functions of those components are as follows:

DCSpark-master: Runs a Spark master in Standalone mode and exposes a port for Spark and a port for the WebUI.

DCSpark-worker: Runs a Spark worker in Standalone mode and connects to the corresponding Spark master via DNS name spark-master-*number*, the *number* is the number of clusters with different runtime environments.

DCSpark-driver: Allows running things like pyspark to connect to spark-master.

4. DCM Architecture

The *DCM* is a cluster management framework and responsible for the management of the hardware resources of the physical cluster, the *DCSpark* job scheduling and life cycle management. We begin our description of *DCM* framework by discussing it's overview. We then describe the components of *DCM*, including container build , container management and scheduling mechanisms.

4.1. Overview

The *DCM* aims to provide easy-to-use, scalable and configurable runtime environment framework for the *DCSpark* task. This framework allows users to deploy, run (a single machine or across machine) and manage the cluster agnostic to the underlying hardware.

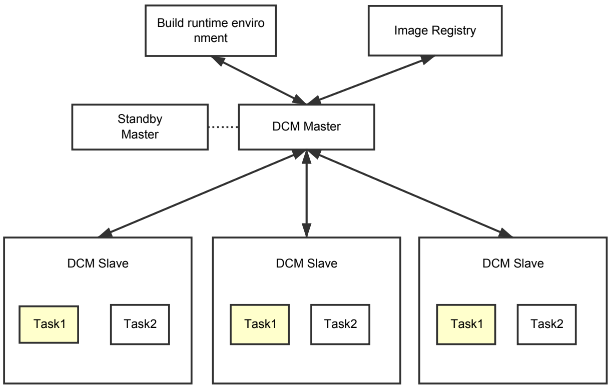


Fig. 1. The DCM architecture diagram, showing two running tasks which can have different runtime environment.

Fig.1 shows the main components of the *DCM*. It consists of a master manager node, slave agent nodes, runtime environment building module and image registry. The master nodes save the cluster resource information with etcd[15] cluster, and be responsible for scheduling and managing the slave nodes, and the slave nodes execute the tasks which may have different runtime environment. The *DCM Agent* is the local agent runs on each slave nodes. It manages local resources and gets commands from the master node to start and stop tasks. The *DCSpark* Orchestrating module builds the distributed runtime environment for job, which includes *DCSpark-master*, *DCSpark-worker* and *DCSpark-driver*, and pushes them to the image registry. The users can use startup command to start a *DCSpark* cluster task.

In different hosts, the Docker container is not able to communicate directly. The *DCM* must solve this problem for it's a distributed data processing platform. A feasible solution is to replan the use of the IP address for all physical nodes in the cluster. The *flannel* is a virtual network tool that gives a subnet to each host. In this way, the *DCM* builds a overlay network, that is, the TCP packets in the Docker container belong to different slave nodes are packaged into another network package for routing forwarding and communication.

4.2. DCM Master

The *DCM* is a master/slave framework. The master is lightweight management node which receives the registration of the slave nodes and manages global resource scheduling. It is designed to be run on the master host in the cluster, and its purpose is to implement the logic that runs the *DCM* cluster and schedule the resource for *DCSpark* jobs. If there is only one physical machine in the cluster, it will run in the local mode.

In resource management and scheduling module, on the one hand, the master node is responsible for receiving the user's job requests and allocating the resources to build a *DCSpark* cluster, on the other hand, monitors these resources to ensure that these resources remain in the desired state. Every ten seconds, the master node checks the running states and resource allocation situations of slaves. Then it judges whether the results are consistent with the expected results. If not, it needs to take appropriate actions, such as restart the slave node, migrate the container, delete the node, etc. Such mechanism and Spark's own fault tolerance mechanism ensure the job's fault tolerance of our framework.

Multi-master cluster mode is used to prevent single master point failure in our framework. To build High-Availability Clusters, our framework needs to ensure the fault tolerance of the main nodes at the same time. That is to say, if we just design the *DCM* as a single-master cluster, it is likely to cause the entire cluster out of availability when the master node get out of order. A very important part of the fault tolerance of the master node is the persistent problem of the cluster state. In our framework, we use the *etcd* as a distributed storage system to store the slaves and jobs information. The *etcd* includes data storage redundancy, backup, and high availability and reduced complexity of realization of the fault tolerance of the master node.

4.3. DCM Agent

The *DCM Agent* is the most important component of the cluster slave node process and it is responsible for the management and maintenance of all containers running on this host. It reports to the master the host CPU, Memory, Hard disk, IP address and other basic information meanwhile gets the resource information for each container of docker.

The *DCM Agent* is designed to be used on all hosts excluding the master in the system. It is used to monitor the host resources and manage containers. Further, it's called by the remote scheduler running on the master and their functions are triggered by RPC calls. Finally, as we all know, the cluster will produce lots of garbage containers and images in the host and brings the waste of resources and operation timeout with the operation of a large number of containers.

In our framework, the garbage collection service ensures a clean and concise container operation environment and improves the management performance of the container. For containers garbage collection, the agent gets all the container information every 5 minutes and removes the corresponding container according to the decision condition whether the container has stopped. For images garbage collection, we use the LRU (Least Recently Used) algorithm to decide which image to be removed. Eq.(1) is given the image capacity to be deleted, where F represents the capacity to release, C represents the total disk capacity, U is the usage capacity of the images and T is the predetermined threshold of usage.

$$F = \begin{cases} U/2 & ;U/C > T \\ 0 & ;others \end{cases} \tag{1}$$

4.4. Scheduler

The scheduler is responsible for carrying out most of the logic associated with starting and stopping containers based on resource informations. The *DCM* scheduler is divided into two stages, preselection and assessment. The preselection is to select the machines that be able to schedule jobs as candidate nodes. The assessment is to set priorities for each candidate node to describe the appropriate level.

Preselection: In the preselection phase, we detect each minion's CPU and memory resources whether they can meet the demands for a *DCSpark* nodes have the preinstall resources value.

Assessment: In the assessment phase, our principle is to try to assign a *DCSpark* cluster on hosts of as little as possible in order to reduce the cross host network I/O. For the master node and driver node, we use the least requested priority to schedule resources to host with low resources usage rate. The usage rate is obtained by the following Eq.(2), where C and M are defined as the CPU capacity and the memory capacity in a host and, $\sum_{i=1}^{n} c_i$ and $\sum_{i=1}^{n} m_i$ are the sum of CPU and memory that a *DCSpark* clusters have requested.

$$rate = \frac{\sum_{i=1}^{n} c_i}{C} + \frac{\sum_{i=1}^{n} m_i}{M} \tag{2}$$

Every *DCSpark* clusters's node have the determined CPU and memory demand. There are a lot of shuffle process spark in the running processes, and to reduce the cross host network IO we sort all the host by the Eq.(3) to deploy DCSpark to the host as less as possible. C_r and M_r are the remainder CPU and memory resources, and, C_w and M_w are the requested CPU and memory resources by each *DCSpark* slave node.

$$v = \begin{cases} \frac{C_r}{C_w} & ; \frac{C_r}{C_w} < \frac{M_r}{M_w} \\ \frac{M_r}{M_w} & ; else \end{cases} \tag{3}$$

4.5. Runtime Environment Build

To deploy a cluster, including two phases, the first phase is to build a run-time environment mirror, and the second stage is to configure the resource requirements of one *DCSpark* cluster job. In the aspect of construction of running environment, we adopt the way similar to *DockerFile*. The difference is we build three images, *DCSpark-master*, *DCSpark-worker* and *DCSpark-driver*, then these images will be pushed to the image registry. We use the *json* file to configure resource requirements of a *DCSpark* cluster, including number of the CPU and memory size of master, driver and each worker node , and the number of worker nodes that contain. In this way, we can construct a distributed processing cluster that contains a user-defined runtime environment.

5. EVALUATION

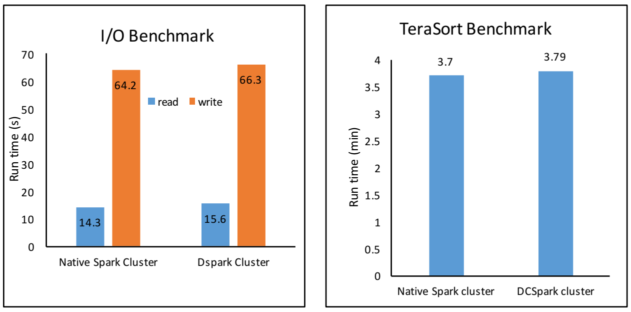
This section evaluates the performance of *DCSpark* with some practical application cases as the benchmarks test. In order to obtain meaningful comparisons of our system, we evaluated its performance against the native Spark cluster. Because *DCSpark* is a framework of distributed data processing, and use the Docker containers and virtual network technology, the isolation, calculation , disk I/O and network I/O performance are the important parts of the framework evaluation. There are a few interesting results of those experiments here.

5.1. Test Environment

Our experimental hardware environment consists of three Lenovo servers (24 Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz 32GB RAM). One *DCSpark* cluster have six worker nodes(each with 2 cores CPU and 4G memory), and the Spark cluster has the same resource configure. We choose four experiments to test the *DCSpark* cluster performance. The *TeraSort* is a CPU, Disk and Network intensive job which have lots of data to read and shuffle. The *Logistic Regression*[16] runs iterative optimization procedures and is a common classification algorithm in Spark, which is used to test the calculation performance. The *face recognition system* and *video synopsis system*, these applications have different runtime environment and are the applications running in our framework in the end.

5.2. TeraSort

The *TeraSort* benchmark suite sorts data as fast as possible to benchmark the performance of the MapReduce framework. It combines testing the HDFS and MapReduce layers of cluster and consists of three MapReduce programs. We use three steps to test it, *a)* generating large data sets to be sorted and write those data to HDFS for the I/O test, *b)* reading the data from HDFS and sort them and *c)* Validating the sorted output data via TeraValidate.



(a) (b)

Fig. 2. I/O Benchmark (a) and TeraSort Benchmark (b).

We ran the *TeraSort* on a 10G data set. This process is designed such that it uses one map task per file, i.e. it is a 1:1 mapping from files to map tasks. Fig.2(a) shows a comparison of the disk I/O benchmark runtimes between the *DCSpark* cluster and Spark cluster. Fig.2(b) shows the complete TeraSort benchmark run time of the two clusters.

We find that the overall performance of the two cluster is close resemblance. Based on the analysis of the jobs log, the algorithm has lots *ShuffleMapStage* processes. Due to the network I/O, *DCSpark* consumes more time and this can also explains the reasons for a slight difference performance.

5.3. Logistic Regression

The Spark is an iterative calculation framework based on memory, which is suitable for the application occasions which need to be operated by several specific data sets. The *Logistic Regression* is a common classification algorithm using gradient descent which includes many iterative calculation processes.

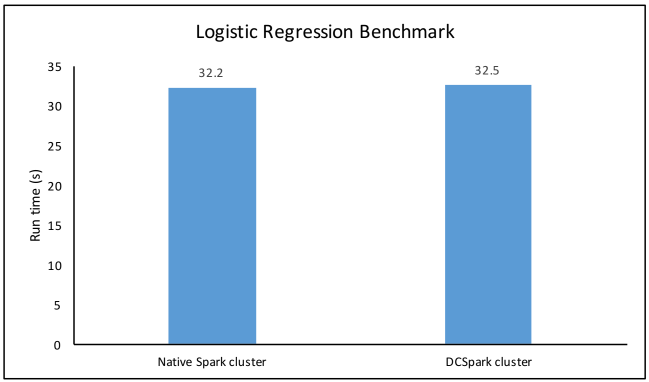


Fig. 3. Logistic Regression Benchmark.

Fig.3 shows the iterative calculation performance for influence of container virtualization. In our algorithm, we test the performance of the two clusters by reducing the amount of data and increasing the number of iterations of the algorithm. The results show that the *DCSpark* cluster has little performance loss in terms of computational performance.

5.4. Face Recognition and Video Synopsis

This part we mainly test runtime environment isolation with the *DCSpark* framework. Our framework is mainly designed to the distributed video analysis which tend to have different system configuration and may rely on different libraries. Our *face recognition system* and *video synopsis system* are two typical video analysis algorithm. One is stream processing task, and the other is a batch task. In our experiment, these two applications, which have different running environment, can completely independent running in our framework. This experiment shows that *DCSpark* leverages existing OS isolation mechanism to provide performance isolation between different applications.

6. Conclusion and Future Work

In this paper, we present *DCSpark*, a framework that allows users to run Docker container-based Spark cluster across multiple host. Our experiment shows the *DCSpark* performance is very close to the native cluster. Further, we present an implementation of our framework called *DCM* which is able to manage the physical cluster and build multiple *DCSpark* clusters with different runtime environment. We are currently using *DCM* to Manage resources 10-node cluster in our lab and different runtimes environment of streaming and batch processing jobs run in *DCSpark* cluster every day. The *DCM* performs simple resource-based scheduling of Docker container and manages the *DCSpark* jobs.

Our experimental evaluation shows that *DCSpark* introduces negligible overhead for CPU and memory performance compared with the native Spark cluster. For I/O-intensive workloads, it should be used carefully although the overhead is acceptable in most cases. In addition, containers communication across the host should pass the UDP encapsulation and the UDP unpacked by the overlay network, which will affect the network I/O to a certain extent.

In future work, we plan to further analyze the scheduling model and determine whether any extensions can improve the overall performance of the cluster. Furthermore, we will research multiple overlay networks to choose a way to reduce the overhead of network I/O.

7. References

[1] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] S. Humbetov, “Data-intensive computing with map-reduce and hadoop,” in *Application of Information and Communication Technologies (AICT), 2012 6th International Conference on*. IEEE, 2012, pp. 1–5.

[3] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 233–240.

[4] 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.

[5] R. Rosen, “Linux containers and the future cloud,” *Linux J.*, vol. 2014, no. 240, Apr. 2014.

[6] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark : Cluster Computing with Working Sets,” *HotCloud’10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, p. 10, 2010.

[8] OpenCV 3.1. [Online]. Available: http://docs.opencv.org/3.1.0/

[9] OpenCV 2.3.0. [Online]. Available: http://docs.opencv.org/2.3/

[10] N. Regola and J.-C. Ducom, “Recommendations for virtualization technologies in high performance computing,”

in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 409–416.

[11] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 275–287.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[13] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, 2011, pp. 24–24.

[14] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[15] etcd. [Online]. Available:
https://github.com/coreos/etcd

[16] T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin, "The elements of statistical learning: data mining, inference and prediction," *The Mathematical Intelligencer*, vol. 27, no. 2, pp. 83–85, 2005.