Efficient Software Defined Systems using Common Core Components

Hyungro Lee and Geoffrey C. Fox
School of Informatics and Computing
Indiana University
Bloomington, IN 47408
Email: {lee212,gcf}@indiana.edu

***Abstract*—With advent of Docker containers, an application deployment using container images gains popularity over scientific communities and major cloud providers to ease building reproducible environments. While a single base image can be imported multiple times from different containers to reduce storage consumption by a sharing technique, copy-on-write, duplicates of package dependencies are often observed over containers. In this paper, we propose new approaches to the container image management for eliminating duplicated dependencies. We create Common Core Components (3C) to share package dependencies by version control system commands; submodules and merge. 3C with submodules provides a collection of required libraries and tools in a separate branch, while keeping their base image same. 3C with merge offers a new base image including domain specific components thereby reducing duplicates in similar base images. Container images built with 3C enable efficient and compact software defined systems and disclose security information for tracking Common Vulnerability and Exposure (CVE). As a result, building application environments with 3C-enabled container images consumes less storage compared to existing Docker images. Dependency information for vulnerability is provided in detail for further developments.**

Keywords*—*Software Defined Systems, Common Core Components, Containers, DevOps, Dependencies

I. Introduction

Deployment for modern applications requires frequent changes for new features and security updates with a different set of libraries on various platforms. DevOps tools and containers are used to complete application deployments with scripts but there are problems to reuse and share scripts when a large number of software packages are required. For example, Ansible Galaxy - a public repository provides over ten thousand scripts (called roles) and Docker Hub has at least fourteen thousand container images but most of them are individualized and common libraries and tools are barely shared. This might be acceptable if a system runs only one or two applications without multi tenants but most systems in production need to consider how to run applications efficiently. Container technology i.e. Docker permits a repeatable build of an application environment using container image layers but redundant images with unnecessary layers are observed because of a stacked file system. In this paper, we introduce two approaches about building Common Core Components (3C) in containers therefore building application environments is optimized and contents are visible in detail for further developments.

Common Core Components is a collection of libraries and tools which aims to share dependencies at one place thereby minimizing storage usage for container images. Docker stores container images efficiently and gains speedup in launching a new container instance because images on union mounts reuse same contents i.e. identical image layers over multiple containers without creating copies. The copy-on-write technique adds a new layer to store changes and keeps the original unchanged. In practice, however, many duplicates of package dependencies are observed between old and new container images with version updates as well as containers in similar application groups. Docker images represent contents of image layers using a directed tree, and duplicates in child image layers can occur when a parent image layer is different although contents in child layers are same. This is normal in version control systems and the goal of 3C is to resolve these issues using dependency analysis and revision control functions. We notice that the build of Docker images is transparent through Dockerfile, a script for building a Docker image and Docker history metadata, therefore 3C is able to be established based on these information. The process of building 3C is following. First, installed packages are collected and then dependencies are analyzed. The two functions that we have chosen from version control systems; submodules and merge typically support unifying two separate repositories and branches. If there are containers that change software versions frequently but use same dependencies, the 3C with submodules provides an individual image layer to share dependencies but no changes in existing images. If there are containers that have similar interests but created by different users, the 3C with merge provides a new parent image layer to suggest common dependencies. The effectiveness and implementation of 3C are described in detail at the section III.

We demonstrate that 3C optimizes both consuming disk space and detecting security vulnerability by determining shared components of containers and analyzing dependencies. 3C also suggests a collection of the most commonly required dependencies from High Performance Computing Enhanced Apache Big Data Stack (HPC-ABDS) [1] and survey data, where sampling is done from public Dockerfiles and project repositories. Performance comparison is presented to show efficiency regarding to disk space usage against existing container images. 3C achieves improvements in storing Nginx container images by 37.3% and detects 109 duplicate dependencies out of 429 from survey data of the HPC-ABDS streams layer with 50% of overlaps. We illustrate security vulnerabilities for Ubuntu 16.04 according to system packages and libraries.

II. Background

Reproducibility is ensured with container images which are stored in a stackable union filesystem, and "off the shelf" software deployment is offered through scripts e.g. Dockerfile to build an equivalent software environment across various platforms. Each command line of scripts creates a directory (called an image layer) to store results of commands separately. Container runs an application on a root filesystem merged by these image layers while a writable layer is added on top and other layers beneath it are kept as readable only, known as copy-on-write. The problem is that system-wide shared libraries and tools are placed on an isolated directory and it prevents building environments efficiently over multiple versions of software and among various applications that may use the same libraries and tools. We use collections of HPC-ABDS (Apache Big Data Stack) [1] and github API to present surveyed data in different fields about automated software deployments. In this case, we collected public Dockerfiles and container images from Docker Hub and github.com and analyzed tool dependencies using Debian package information.

A. Software Deployment for dynamic computing environments

Software development has evolved with rich libraries and building a new computing environment (or execution environment) requires a set of dependencies to be successfully installed with minimal efforts. The environment preparation on different infrastructures and platforms is a challenging task because each preparation has individual instructions to build a similar environment, not an identical environment. The traditional method of software deployment is using shell scripts. A system package manager such as apt, yum, dpkg, dnf and make help to automate installing, compiling, updating and removing tools but shell scripts can be easily difficult to understand once it handles more systems and various configurations. A large number of packages are actively updated and added to communities and proper managing in a universal way is necessary to deal with them sustainably. Python Package Index (PyPI) has 107,430 packages in 2017 with 40+ new packages on a daily basis. Public version control repository, Github.com has 5,649,489 repositories with about 20,000 daily added repositories. Most software packages, libraries and tools can be found on their website and using their API. DevOps tools i.e. Configuration management software supports automated installation with repeatable executions and better error handling compared to bash scripts but there is no industry standards for script formats and executions. Puppet, Ansible, Chef, CFEngine and Salt provide community contributed repositories to automate software installation, for example, Ansible Galaxy has 11353 roles available, Chef Supermarket has 3,261 cookbooks available although there are duplicated and inoperative scripts for software installation and configuration. Building dynamic computing environments on virtual environments is driven by these DevOps tools and container technologies during the last few years for its simplicity, openness, and shareability. Note that this effort is mainly inspired by the previous research activities [2], [3], [4], [5], [1], [6].

B. Scripts

Building compute environments needs to ensure reproducibility and constant deployment consistently [7], [8]. Most applications these days run with dependencies and setting up compute environments for these applications requires an exact version of software and configure systems with same options. Ansible is a DevOps tool and one of the main features is software deployment using a structured format, YAML syntax. Writing Ansible code is to describe action items in achieving desired end state, typically through an independent single unit. Ansible offers self-contained abstractions, named Roles, by assembling necessary variables, files and tasks in a single directory. For example, installing software A or configuring system B can be described as a single role. Compute environments are supplied with several software packages and libraries and selectively combined roles build compute environments where new systems require software packages and libraries installed and configured. Although the comprehensive roles have instructions stacked with tasks to complete a software deployment with dependencies, the execution of applications still need to be verified. In consequence, to preserve an identical results from the re-execution of applications, it is necessary to determine whether environments are fit for the original applications.

C. Containers with Dockerfile

Container technology has brought a lightweight virtualization with a Linux kernel support to enable a portable and reproducible environment across laptops and HPC systems. Container runtime toolkit such as Docker [9], rkt [10] and LXD [11] uses an image file to initiate a virtualized environment including necessary software packages and libraries without an hypervisor. These tools create an isolated environment on a same host operating system using the Linux kernel features such as namespaces, cgroups, seccomp, chroot and apparmor. Recent research [12] shows that containers outperform the traditional virtual machine deployments but running containers on HPC systems is still an undeveloped area. Shifter [13] and Singularity [14] have introduced to support containers on HPC with a portability and MPI support along with docker images. These efforts will be beneficial to scientific applications to conduct CPU or GPU intensive computations with easy access of container images. For example, a neuroimaging pipelines, BIDS Apps [15], is applied to HPCs using Singularity with existing 20 BIDS application images and Apache Spark on HPC Cray systems [16] is demonstrated by National Energy Research Scientific Computing Center (NERSC) using shifter with a performance data of big data benchmark. Both researches indicate that workloads for scientific applications and big data are manageable by container technologies on HPC systems with a reproducibility and portability.

Listing 1: Dockerfile Example

```
FROM ubuntu:14.04

MAINTAINER Hyungro Lee <lee212@indiana.edu>

RUN apt-get update && apt-get install -y \\
    build-essential wget git
...
```

Dockerfile (See Listing 1) uses a custom template to describe installation steps of building docker images in a bash-like simple format. There are certain directives to indicate particular objectives of the commands, for example, FROM indicates a base image to use and RUN indicates actual commands to run. When an image is being generated, each directive of Dockerfile creates a single directory to store execution results of commands. Meta-data of these directories is recorded in a final image to provide a unified logical view by merging them. The tag for an image is a reference for stacked image layers. For example in Listing 1, *ubuntu:14.04* is a tag to import stacked image layers of Ubuntu 14.04 distribution and the following directives i.e. *MAINTAINER* and *RUN*, will be added. This allows users to import other image layers and start building own images.

D. Environment Setup

Preparing environment is installing all necessary software, changing settings and configuring variables to make your application executable on target machines. Container technology simplifies these tasks using a container image which provides a repeatable and pre-configured environment to your application therefore you can spend more time on an application development rather than software installation and configuration. One of the challenges we found from container technologies in preparing environment is managing dependencies for applications. Container users who want to run applications with particular libraries have to find relevant container images otherwise they have to create a new image from scratch thereby brining all required tools and libraries. One possible solution for this problem is to offer a common core component (3C) when environment is being built. We noticed that there is a common list of libraries for particular type of applications based on the survey from Docker images and Dockerfile scripts. The idea is to offer curated collection of libraries for domain-specific applications and use the list of libraries surveyed from communities. For example, libraries for linear algebra calculation i.e. liblapack-dev and libopenblas-dev are commonly used for applications in the analytics layer of HPC-ABDS according to the survey (shown in Table I). Additional package installation might be required if a suggested list of dependencies does not satisfy all requirements of an application.

E. Package Dependencies

Software packages have many dependencies especially if the packages are large and complex. Package management software e.g. apt on Debian, yum on CentOS, dnf on Fedora and pkg on FreeBSD automates dependency installation, upgrading or removal through a central repository package and a package database. The information of package dependencies along with version numbers controls a whole process of software installation and avoids version conflicts and breaks. In addition, reverse dependencies show which packages will be affected if the current package is removed or changed. *nodejs* and *ruby* in Table I have a few dependencies but a large number of reverse dependencies exist . Software incompatibility can easily occur to other packages if these packages are broken or missing. Figure 1 shows relations between dependencies (depends), reverse dependencies (rdepends), package size (size) and package size including dependencies (total_size)

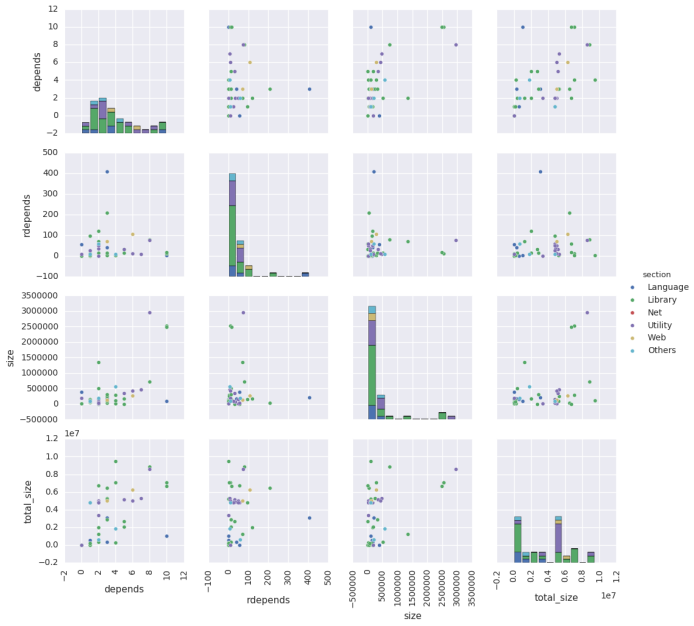


Fig. 1: Debian Package Relations between Dependencies, Reverse Dependencies and Package Sizes (itself and including dependencies) for Popular Docker Images

among six different sections. Interestingly the size of package itself does not increase when the number of dependencies are incremented but it shows positive correlation between the number of dependencies and the total package size including dependencies. It explains that shared libraries are common for most packages to manage required files efficiently on a system. This is based on the survey of Docker scripts i.e. Dockerfile from public software repositories on github.com. Note that there are several package managers available on Linux distributions, see details in Table II.

F. Application Domains

Debian packages are categorized in 57 sections ranging from administration utilities (abbreviation is *admin*) to X window system software (abbreviation is *x11*) and it helps us to better understand the purpose of a package. An application typically requires several packages installed and a certain choice of packages is found in common according to interests of applications. SciPy [17] is a collection of python packages for scientific computing, for example, and the dependencies include a math library i.e. libquadmath0 - GCC Quad-Precision Math Library and basic linear algebra packages i.e. libblas3 - shared library of BLAS (Basic Linear Algebra Subroutines) and liblapack3 - Library of linear algebra routines 3. The classification of Big Data and HPC applications is well established in the HPC and Apache Big Data Stack (HPC-ABDS) layers [1]. Figure 2 shows six dependency sections for selected HPC-ABDS layers such as Layer 6) Application and Analytics - (Analytics with green dot), Layer 11B) NoSQL - (Nosql with purple dot) and Layer 14B) Streams - (Stream with beige dot). Library dependencies (2a) including development tools, utilities and compilers are observed in most layers as well as reverse dependencies (2b), especially in the analytics layer and the machine learning layer. Note that, the machine learning

Name	PCT1	PCT2	PCT3	PCT4	Description	Section	CT1	CT2	Dependencies	Size	Important
software-properties-common	0.01	0.06	0.02	0.03	manage the repositories that you install software from (common)	admin	8	4	python3-dbus, python-apt-common, python3-software-properties, gir1.2-glib-2.0, ca-certificates, python3:any, python3-gi, python3	9418 (630404)	optional
build-essential	0.14	0.16	0.03	0.05	Informational list of build-essential packages	devel	5	32	dpkg-dev, libc6-dev, gcc, g++, make	4758 (2705548)	optional
g++	0.15	0.06	0.02	0.01	GNU C++ compiler	devel	4	57	cpp, gcc, g++-5, gcc-5	1506 (22034848)	optional
gcc	0.03	0.05	0.02	0.01	GNU C compiler	devel	2	57	cpp, gcc-5	5204 (6735366)	optional
groovy	-	-	**0.01**	-	Agile dynamic language for the Java Virtual Machine	universe/devel	14	10	libbsf-java, libservlet2.5-java, antlr, libxstream-java, libcommons-logging-java, libjline-java, libasm3-java, libjansi-java, libregexp-java, libmockobjects-java, junit4, default-jre-headless, ivy, libcommons-cli-java	9729202 (3257906)	optional
libatlas-base-dev	-	**0.06**	-	-	Automatically Tuned Linear Algebra Software, generic static	universe/devel	2	8	libatlas-dev, libatlas3-base	3337570 (2690424)	optional
liblapack-dev	-	**0.03**	-	-	Library of linear algebra routines 3 - static version	devel	2	22	liblapack3, libblas-dev	1874498 (2000176)	optional
ruby	-	0.01	0.01	0.01	Interpreter of object-oriented scripting language Ruby (default version)	interpreters	1	987	ruby2.1	6026 (73880)	optional
maven	-	-	0.02	0.01	Java software project management and comprehension tool	universe/java	2	5	default-jre, libmaven3-core-java	17300 (1441844)	optional
libffi-dev	-	0.03	-	**0.01**	Foreign Function Interface library (development files)	libdevel	2	11	libffi6, dpkg	162456 (2101914)	extra
libssl-dev	0.12	0.07	0.01	0.03	Secure Sockets Layer toolkit - development files	libdevel	2	70	libssl1.0.0, zlib1g-dev	1347070 (1258956)	optional
net-tools	0.01	0.02	0.03	0.05	NET-3 networking toolkit	net	1	51	libc6	174894 (4788234)	important
chrpath	-	-	-	**0.05**	Tool to edit the rpath in ELF binaries	utils	1	0	libc6	12932 (4788234)	optional
git	0.33	0.21	0.06	0.07	fast, scalable, distributed revision control system	vcs	8	75	perl-modules, liberror-perl, libpcre3, libcurl3-gnutls, git-man, zlib1g, libc6, libexpat1	2951026 (8563378)	optional
nodejs	0.01	0.04	-	0.02	evented I/O for V8 javascript	universe/web	6	287	libssl1.0.0, libc6, libstdc++6, zlib1g, libv8-3.14.5, libc-ares2	683742 (7551922)	extra

TABLE I: Common Debian Packages from Sample Survey Data
(PCT1: Percentage by General Software, PCT2: Percentage by Analytics Layer, PCT3: Percentage by Data processing Layer, PCT4: Nosql Layer, CT1: Count of Dependencies, CT2: Count of Reverse Dependencies)

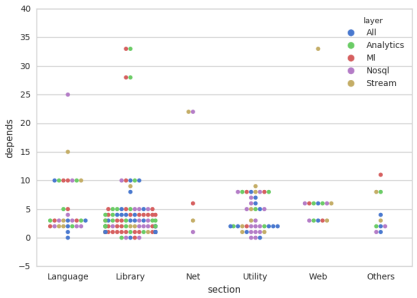
Name	Distribution	Package Type	file format	License	Language
dpkg	Debian	Binary	.deb	GPL	C, C++
apt	Ubuntu	Binary	.deb	GPL	C++
Nix	NixOS	Binary	.nix	LGPL	C++
RPM	RedHat	Binary	.rpm	GPL	C, Perl
dnf	Fedora	Binary	.rpm	GPL v2	C, Python
yum	CentOS	Binary	.rpm	GPL v2	Python
zypper	OpenSUSE	Binary	.rpm	GPL	C++
pacman	ArchLinux	Binary	.pkg.tar.xz	GPL v2	C
pkg	FreeBSD	Binary	.txz	GPL	C

TABLE II: Package managers of Linux Distributions

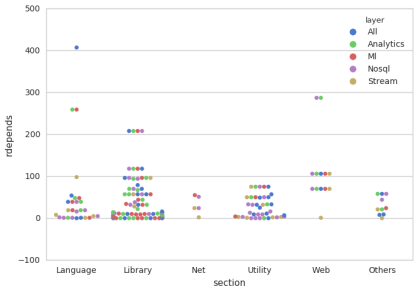
layer is not a part of HPC-ABDS but is manually added to demonstrate other interesting collections as an example. Sub groups of the library section will be necessary to identify a common collection of dependencies for the particular application domains in detail.

G. Docker Images on Union Mounting

Union mount implementations e.g. aufs and overlayfs enable Docker containers to have stackable image layers thereby ensuring storage efficiency for images where a base image layer includes common contents. Additional image layers only carry changes made to the base image while multiple containers share a same base image. This enables containers to reduce storage and booting up time when a new container starts. From a practical point of view, a base image is a set of image layers built from *scratch*, for a linux distribution with a version e.g. *ubuntu:latest, xenial, or 16.04* or *centos:latest or 7* which is a starting point of most images. Common tools or special packages can be added and declared as an another base image for a particular purpose, for example, NVIDIA's CUDA and cuDNN packages are defined as a base image for



(a) Dependencies



(b) Reverse Dependencies

Fig. 2: Debian Package Dependencies for HPC-ABDS Layers

GPU-enabled computation including deep neural network on top of a Ubuntu or CentOS image. This approach is widely adopted because of the following reasons. First, "off the shelf" container images provide application environments to normal users and a standard collection of required software packages

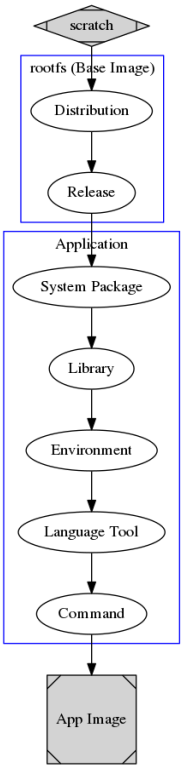


Fig. 3: Dockerfile Workflow

is built for communities of interest. It is also more convenient to update a single base image rather than multiple images, if there are changes to apply. Note that using a same base image reduces storage in its database and avoids duplicates. We see a flattened view of docker images from Figure 3. Base images start from *scratch* as a first image layer and most applications are diverged out from base images.

III. Results

While there are advantages of using layered file systems for containers, we noticed that redundancy in storing docker container images exists. The duplication of image contents occurs when an identical software installation completes with different parents of an image layer. As shown in Figure 4, tree structure is preserved to represent container images, for example, two images (#1 and #2) are identified as a distinct image although the change applied to them is indistinguishable. In this section, we demonstrate two approaches of reducing these duplicates using package dependencies.

A. Common Core Components

The general software deployment retrieves and installs dependencies to complete installation and ensure proper execution of software on a target machine. As shown in Figure 5, Nginx, a lightweight HTTP server, requires 40 more libraries and tools installed, although Nginx itself is only about 3MB of an installed size. We identify these package dependencies and define them as common Core Components (3C).

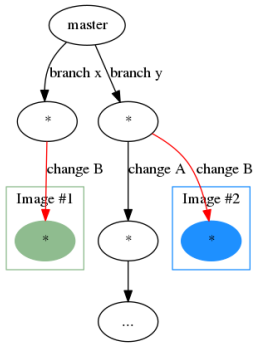


Fig. 4: Union File System Tree

B. Approach I: Common Core Components by Submodules

In version control systems, submodules keep repository commits separate but allow cloning other repositories in a sub directory. With submodules, common core components (3C) can be dispatched but in a separated image layer (See Figure 6). This approach lets you include an image layer without concerning a parent image layer and reduces duplicates without creating a new base image. 3C is supposed to contain dependencies for application and we can find out the dependency information after reviewing current docker images with Dockerfiles and searching package manager databases. Dockerfile is a text file and a blueprint of building an image and installation commands are recorded to replicate an image anytime. Dockerfile has *RUN* directives to execute commands and package manager commands i.e. *apt-get* and *yum* are executed with *RUN* to install libraries and tools. Dependencies of these libraries and tools are described in a package manager cache file (Packages) and stored in its internal database. 3C is built by looking up dependency information from the database with package keywords obtained from Dockerfile. Docker history can be used to examine image construction or composition if Dockerfile is not available. In Figure 7, we created Nginx-3C (about 59.1MB) and re-generated Nginx docker images including the new 3C. The current Nginx docker has 9 individual images (in total 1191.5MB) among various versions of Nginx ranging from 1.9 to 1.13. The base image also varies from Debian 9 (in a slim package) to Debian Jessie 8.4 and 8.5. Whereas the size of new images including Nginx-3C increase about 2.9MB per each version change. The accumulated size of new images is 747.1MB in total to provide 9 individual Nginx images from version 1.9 to 1.13. 37.3% improvements regarding to storing docker images is observed compared to the current Nginx docker images. We notice that 3C by submodules reduce duplicates of contents, especially if software changes its versions but uses equivalent libraries. Nginx 1.9.0 and 1.13.0 have similar constraints of dependencies including version numbers. According to the Debian package information, the similar constraints are following: C library greater than or equal to 2.14 (libc6), Perl 5 Compatible Regular Expression Library greater than or equal to 1:8.35 (libpcre3), Secure Sockets Layer toolkit greater than or equal to 1.0.1 and zlib compression library greater than or equal 1:1.2.0 (zlib1g). Backward compatibility of libraries is ensured for general packages therefore 3C with the latest version of dependencies may cover most cases.

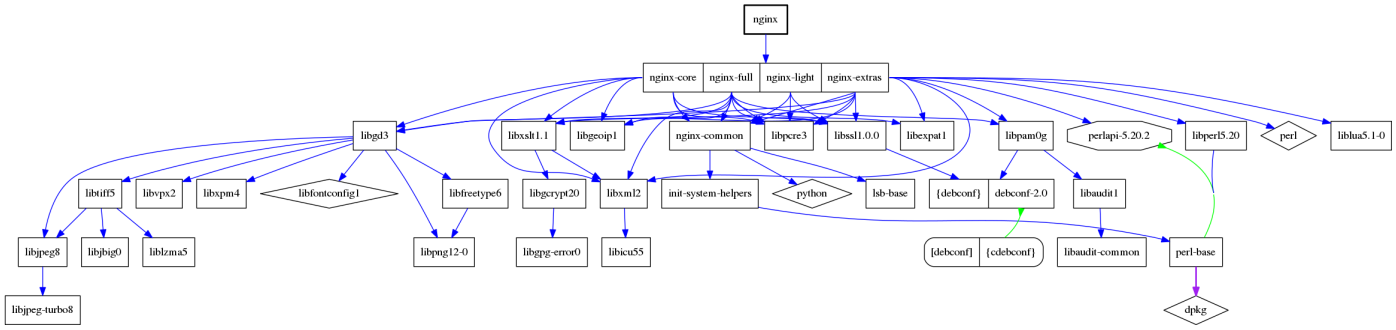


Fig. 5: Nginx Debian Package Dependencies

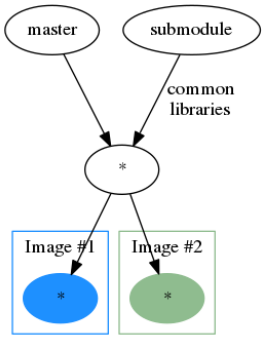


Fig. 6: Common Core Components by submodules

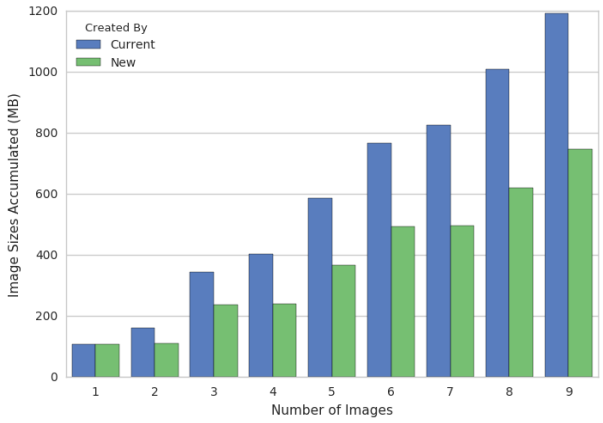


Fig. 7: Comparison of Container Images for Nginx Version Changes
(Current: Built by Official Dockerfiles, New: Built by Common Core Components)

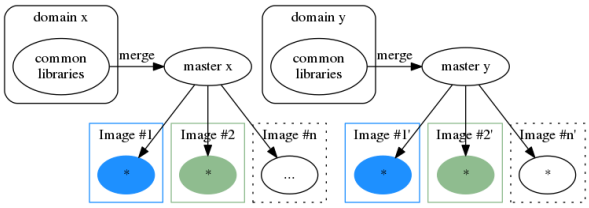


Fig. 8: Common Core Components by merge

C. Approach II: Common Core Components by Merge

The goal of this approach is preparing compute environments on the premises with domain specific common core components merged into a base image. New base images are offered with the common core components of applications. Similar application images (such as Image #1 and #2) in Figure 8 branched out from a same parent image layer. The storage might not be saved if not many images refer a same master image. One of the benefits of this approach is updating base images. Newly discovered components or vulnerable packages are updated and included through updates. Once a number of images sharing a same base image incremented, an additional survey can be conducted to follow trends of development tools and applications. In addition to that, outdated packages can be removed from the 3C. Docker offers 'bring-your-own-environment' (BYOE) using Dockerfile to create images and users can have individual images by writing Dockerfile. We observe that developers and researchers store Dockerfile on a source code version control repository i.e. github.com along with their applications. Luckily, GitHub API offers an advanced keyword search for various use and Dockerfile in particular domains is collected using API tools. Besides, we did a survey of package dependencies for application domains using the collection of HPC-ABDS and built 3C according to the survey data. To construct suitable environments with minimal use of storage, finding a optimal set of dependencies per each domain is critical. As shown in Figure 9, we found that relations between the size of components and the number of components as well as the percentage of common components among images. The first subplot for the streams layer shows that the size of most common components (between 40% and 100%) is increased slowly compared to the least common components. Based on the sample data, 109 out of 429 packages are appeared 50% of Docker images in the streams layer. Other layers of HPC-ABDS are also examined.

IV. Discussion

We achieved application deployments using Ansible in our previous work [18]. In the DevOps phase, configuration management tool i.e. Ansible automates software deployment to provide fast delivery process between development and operations [19] but preserving environments for applications is not ensured unless all required repositories are preserved.

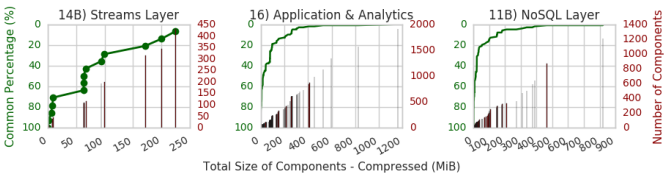


Fig. 9: Common Core Components for HPC-ABDS

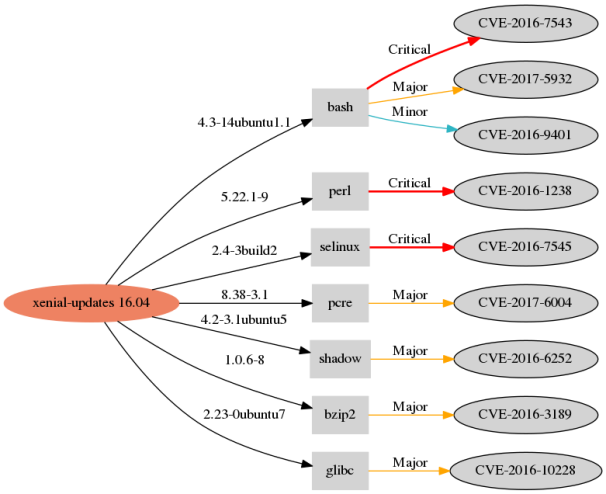


Fig. 10: Example of Security Vulnerabilities for Ubuntu 16.04 based on Libraries

TABLE III: NIST Big Data Projects

ID	Title
N_1	Fingerprint Matching
N_2	Human and Face Detection
N_3	Twitter Analysis
N_4	Analytics for Healtcare Data / Health Informatics
N_5	Spatial Big Data/Spatial Statistics/Geographic Information Systems
N_6	Data Warehousing and Data Mining

TABLE IV: Technology used in a subset of NIST Use Cases. A ✓ indicates that the technology is used in the given project. See Table III for details on a specific project. The final row aggregates ✓ across projects.

ID	Hadoop	Mesos	Spark	Storm	Pig	Hive	Drill	HBase	Mysql	MongoDB	Mahout	D3 and Tableau	nltk	MLlib	Lucene/Solr	OpenCV	Python	Java	Ganglia	Nagios	zookeeper	AlchemyAPI	R
N_1	✓		✓			✓	✓	✓	✓									✓	✓	✓	✓		
N_2		✓	✓													✓	✓				✓		
N_3				✓				✓		✓		✓	✓				✓	✓			✓	✓	✓
N_4	✓		✓					✓			✓	✓		✓	✓			✓			✓		
N_5	✓		✓								✓	✓		✓				✓					
N_6	✓		✓		✓	✓		✓		✓	✓	✓		✓	✓			✓			✓		
count	4	1	5	1	1	2	1	4	1	2	3	4	1	3	2	1	2	5	1	1	5	1	1

Linux containers resolve this problem but their scripts are not organized like DevOps tools. Instructions of DevOps tools are written in structured document formats i.e. YAML, JSON, and Ruby DSL, and there are benefits of using DevOps scripts like Ansible Roles that we wrote in our previous work. Various terminologies i.e. recipes, manifests, and playbooks are used to manage systems and deploy software but all of them have similar concepts and abstract levels. We also notice that these scripts can be converted to build container images, and vice versa if any of DevOps scripts need to be called in building compute environments. For example, Table IV shows that 27 Ansible roles are created to deploy software components among six NIST use cases in Table III. Some of the roles such as Apache Hadoop and Spark are shared frequently and we intend to provide more roles in building Big Data applications. With the experience from NIST projects [18], a few challenging tasks are identified in DevOps tools, a) offering standard Ansible Roles to ease application development with different tools, and b) integrating container technologies towards application-centric deployments. Infrastructure provisioning need to be integrated to avoid resource underutilization. We defer these considerations to future work.

V. Related Work

A. Template-Based Software Deployment

Template deployment is a means of installing software and building infrastructure by reading instructions written in a templating language such as YAML, JSON, Jinja2 or Python. The goal of a template deployment is to offer easy installation, repeatable configuration, shareability of instructions for software and infrastructure on various platforms and operating systems. A template engine or an invoke tool is to read a template and run actions defined in a template towards target machines. Actions such as installing software package and setting configurations are described in a template with its own syntax. For example, YAML uses spaces as indentation to describe a depth of a dataset along with a dash as a list and a key-value pair with a colon as a dictionary and JSON uses a curly bracket to enclose various data types such as number, string, boolean, list, dictionary and null. In a DevOps environment, the separation between a template writing and an execution helps Continuous Integration (CI) because a software developer writes deployment instructions in a template file while a system operations professional executes the template as a cooperative effort. Ansible, SaltStack, Chef or Puppet is one of popular tools to install software using its own templating language. Common features of those tools are installing and configuring software based on definitions but with different strategies and frameworks. One observation is that the choice of implementation languages for those tools influences the use of a template language. The tools written by Python such as Ansible and SaltStack use YAML and Jinja which are friendly with a Python language with its library support whereas the tools written by Ruby such as Chef and Puppet use Embedded Ruby (ERB) templating language.

B. Linux Containers on HPC

The researches [20], [21], [22] indicate the difficulty of software deployments on High Performance Computing (HPC). Linux containers is adopted on HPC with the benefits

of a union file system, i.e. Copy-on-write (COW) and a namespace isolation and is used to build an application environment by importing an existing container image [14], [13], [23]. The container runtime tools on HPC e.g. Singularity, Shifter and chroot import Docker container images and wish to provide an identical environment on HPC as one on other platforms.

VI. Conclusion

We presented two approaches to minimize image duplicates using package dependencies, named Common Core Components (3C). The current stacked docker images create redundancies of storing contents in several directories when software packages are installed with different parent image layers and we build dependency packages that mostly shared with other images and provide where it needs. First approach is building 3C based on the analysis of current Docker images and scripts i.e. Dockerfile and combines with a master image using submodules. This is useful where software is updated frequently with new versions but equivalent dependencies are shared. In our experiment, Nginx with 3C shows 37.3% improvements in saving image layers compared to the current docker images. The other approach is building 3C based on the surveyed data for application domains and provides a certain set of dependencies to provide a common collection for various applications. Besides that, security concerns are raised with container technologies and inspecting Common Vulnerabilities and Exposures (CVE) is inevitable to prevent attacks. Improving security using dependency information from 3C is promising to detect security bugs and mitigate possible security issues. For example, Ubuntu 16.04 packages have several CVEs and the dependency graph (Figure 10) represents affected packages with version numbers and vulnerability severity ratings from NIST National Vulnerability Database (NVD). Our future work is to indicate security information using dependencies and suggest fixed versions of packages to update.

Acknowledgment

We gratefully acknowledge generous support from CIF-DIBBS 143054: Middleware and High Performance Analytics Libraries for Scalable Data Science and NSF RaPyDLI 1415459. We thank Intel for their support of the Juliet system. We appreciate the support from IU Precision Health Initiative and the FutureSystems team. We thank Gregor von Laszewski for many discussions.

References

[1] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow, "Hpc-abds high performance computing enhanced apache big data stack," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 1057–1066.

[2] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and hpc convergence," in *Workshop on Big Data Benchmarks*. Springer, 2015, pp. 3–17.

[3] G. Fox, J. Qiu, and S. Jha, "High performance high functionality big data software stack," 2014.

[4] J. Qiu, S. Jha, A. Luckow, and G. C. Fox, "Towards hpc-abds: an initial high-performance big data stack," *Building Robust Big Data Ecosystem ISO/IEC JTC 1 Study Group on Big Data*, pp. 18–21, 2014.

[5] G. Fox and W. Chang, "Big data use cases and requirements," in *1st Big Data Interoperability Framework Workshop: Building Robust Big Data Ecosystem ISO/IEC JTC 1 Study Group on Big Data*. Citeseer, 2014, pp. 18–21.

[6] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "White paper: Big data, simulations and hpc convergence," in *BDEC Frankfurt workshop. June*, vol. 16, 2016.

[7] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, 2007.

[8] A. Goodman, A. Pepe, A. W. Blocker, C. L. Borgman, K. Cranmer, M. Crosas, R. Di Stefano, Y. Gil, P. Groth, M. Hedstrom *et al.*, "Ten simple rules for the care and feeding of scientific data," *PLoS computational biology*, vol. 10, no. 4, p. e1003542, 2014.

[9] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[10] "Coreos/rkt: a container engine for linux designed to be composable, secure, and built on standard," https://github.com/coreos/rkt, 2016, [Online; accessed 09-November-2016].

[11] "Ubuntu lxd: a pure-container hypervisor," https://github.com/lxc/lxd, 2016, [Online; accessed 09-November-2016].

[12] 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.

[13] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[14] G. M. Kurtzer, "Singularity 2.1.2 - Linux application and environment containers for science," Aug. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60736

[15] K. J. Gorgolewski, F. Alfaro-Almagro, T. Auer, P. Bellec, M. Capota, M. M. Chakravarty, N. W. Churchill, R. C. Craddock, G. A. Devenyi, A. Eklund *et al.*, "Bids apps: Improving ease of use, accessibility and reproducibility of neuroimaging data analysis methods," *bioRxiv*, p. 079145, 2016.

[16] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 97–110.

[17] E. Jones, T. Oliphant, and P. Peterson, "{SciPy}: open source scientific tools for {Python}," 2014.

[18] B. Abdul-Wahid, H. Lee, G. von Laszewski, and G. Fox, "Scripting deployment of nist use cases," 2017.

[19] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[20] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: Bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 40.

[21] M. Geimer, K. Hoste, and R. McLay, "Modern scientific software management using easybuild and lmod," in *Proceedings of the First International Workshop on HPC User Support Tools*. IEEE Press, 2014, pp. 41–51.

[22] A. Devresse, F. Delalondre, and F. Schürmann, "Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems," in *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. ACM, 2015, pp. 25–31.

[23] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," Los Alamos National Laboratory (LANL), Tech. Rep., 2016.