Viability of Squashfs as a container distribution method in HPC

Ronald Shane Goff, May 2018

The demand for user defined software stacks (UDSS) has been increasing in the high performance computing (HPC) community. Container technology has become popular due to the flexibility and isolation it provides to HPC users. Container images must be available to all nodes involved for use in HPC and can be distributed to compute nodes in a variety of ways. A common method for container image distribution is to simply copy the container image to memory on each compute node, which can be time consuming at scale, and uses valuable memory on each node. The squash filesystem has proven valuable and efficient for this task, but requires root level access to use. We are interested in exploring the value of user space mounting squash filesystems, and examining the performance difference.

In the world of HPC the software stack available in the compute center is managed by a team of people to meet the needs of users. This provides a great deal of stability to the software available to users. In addition, the software stack is often optimized for the cluster it is running on. This system is fantastic for users who only need the software provided. There are some users who have need of more to perform their work. While you can request new software be added to the managed stack, unless there is sufficient demand it is unlikely the additional work created by this need will be serviced to support a small handful of users will be done. If you can’t get support for the software you require, you can always build the software you need from source. However, depending on your needs this can become a rather large and bothersome task. Often users will have very complex software needs with many dependencies, and or versions of software already in use that are older or newer than what is available. If you have many dependencies, the amount of work to build them all can become rather high. If for some reason you had to move your work to a new cluster, you would have to start over and rebuild everything. Being able to acquire and build dependencies can also require an internet connection which may be difficult when working within a computer cluster as they are not commonly connected to the outside world. In some fields of research you may be required to use a validated software stack in order for results to be accepted. Using a UDSS also makes your work easily repeatable for others. Instead of trying to replicate work as best you can, the entire software stack is available to you with all the same versions and setup. This also means you can easily repeat the experiment for yourself as well. Using a UDSS can aid the majority of these problems. Taking a complicated software stack, and building it in a container means you have the freedom to use any software you need. As container images are built on your workstation you have access to everything you could need such as internet access. Need to move to a new cluster? No problem. Simply copy your container image to a new cluster and you're ready to go. As the entire dependency stack is contained, you can build once and run anywhere.

The use of containers requires some steps before you can execute your work on a cluster. There are a variety of tools available for container execution in HPC, but they all require you to create your container beforehand. Naturally there is also an assortment of tools to complete this work. Docker is a common method for creating Linux containers. Docker has a large public repository of container images ready to be used to make your build process easier. The Docker ecosystem provides all the tools required for you to build and execute containers on any machine where you have root access. This is great for building containers on your workstation to be later deployed on clusters. However, the work involved with supporting Docker on a cluster is very steep. Docker requires a daemon to be running on all computers where a container would run. Docker also requires root to use. Giving all users root to be able to use Docker is an idea that would make cluster administrators very nervous. In order to build containers, Docker also needs unfettered internet access which is not common to have on clusters. There are tools for running containers in HPC that do not require Docker for execution. We only need Docker to build our containers.

There are many tools available for running containers. We will talk about only a few of these as examples, and to understand the container landscape. Charliecloud is a container runtime tool developed at Los Alamos National Laboratory. This tool is very lightweight, and provides little in services outside of simply executing containers. There are many steps involved in executing a container. You must build your container, make it available to all compute nodes in a cluster, and more. Charliecloud handles the execution, leaving the remaining effort of creating the container, transporting it to the cluster, and making it available to nodes in the hands of the user. This is both good and bad depending on your opinion. The advantage is you are free to manage that work however you like. There is also no support burden for using Charliecloud. What it lacks in features it makes up for with ease of use, and flexibility. The only thing Charliecloud needs to do what it does is a filesystem like directory tree. Containers are simply a very efficient and easy way of creating this directory tree.

Other tools are more feature rich to support a variety of things for users and keep everything in the scope of their tool. For many it is easier to use the single tool to manage everything, and work within the limits of that tool. In order to support all features and tools users want, there is a great deal of work invested integrating these. Singularity is a tool that provides a great deal of features to users. It provides a unified interface for users to integrate different container tools such as Docker, busybox or even Singularity Hub, which is their own online container repository. With a wide variety of systems brought under one roof, having a unified interface to use them can be very attractive to users. However, as you can imagine the support effort required to make this available to the users can be very high which is a common concern for system administrators looking to add containers to their HPC systems.

Setting up container systems can also raise security concerns. Providing users a whole extra stack of software that has been unvetted often causes brows to furrow and pearls to be clutched. Using a feature rich container system can escalate concern as this can require the tools have the ability to execute some components as root. In order to manipulate containers, package them, create them and such the tool handling all the work can need root access. This doesn't mean these tools are insecure. This means you must place a great deal of trust in the tool you choose, and the developers of said tool. Depending on what work is common in your HPC center this can matter very little or a great deal. With more privilege comes more risk.

The Squash Filesystem is a compressed read only filesystem in a file. The process to create a squash filesystem is simple. The directory you want to squash must exist somewhere on disk. The metadata and attributes of the directory tree are extracted, and the underlying data is compressed into blocks. Any item not large enough to occupy a block is grouped and compressed as well if possible, including the metadata. This is all done using existing open source tools. In order to use a squash filesystem with no additional tools, you must have root. The squash filesystem can be mounted in any empty directory using the code in the kernel. This is good, as it means the code is very stable and available in linux distributions. This is also bad, as it requires root. As feature rich tools already require root in some capacity there is little additional risk involved in supporting this. This makes the squash filesystem well suited for work with containers in HPC. Creating a squashfs is trivial with existing tools, and can be created using a directory tree of our containers. There are also existing tools to mount squash filesystems without root. These tools are of particular interest for this experiment.

In order to make use of the squash filesystem for container use in HPC we need somewhere to store them that is accessible to all compute nodes in the cluster. There are a variety of filesystems commonly used in HPC. Network File System (NFS) is such a filesystem. NFS is made available to other computers on the network, so they can all treat the NFS share as a local disk. Systems with the NFS mounted can read and write to the disk hosted by another computer. The underlying filesystem can be chosen by the individual setting it up. Common UNIX file systems such as ext4 are typical, as they are natively supported, well tested, secure, and provides features for data integrity. NFS uses transmission control protocol (TCP) to communicate requests from computers in the network to the daemon running on the server which performs the actual reads and writes to disk. This system is quite effective at creating a single shared space all nodes can access, with data integrity in mind. However, all nodes must make their requests to a single process running on a single node for all disk operations. At larger scales all the reads and writes pile up.

Lustre is another example of a file system often used in HPC. Like NFS, it is shared over the network so all nodes in the network can access it. Lustre however is a more specialized tool. Lustre is a parallel distributed file system. By design Lustre is made for highly parallel disk operations for HPC. According to the Lustre documentation “It is best known for powering many of the largest high-performance computing (HPC) clusters worldwide, with tens of thousands of client systems, petabytes of storage, and hundreds of gigabytes per second (GB/sec) of I/O throughput." Lustre operates quite differently compared to your garden variety UNIX file system. Lustre stores data as objects, as opposed to files or blocks. In order for this to happen there are several abstracts that require explanation. Lustre uses a set of servers to store and host information for different purposes. Meta data, which is data describing your data is written to a meta data target (MDT), which is hosted in a metadata server (MDS). There can be multiple MDTs per MDS. There are also object storage targets (OST), which are hosted by object storage servers (OSS). There can also be multiple MDSs, and many OSSs. Targets are block devices used to store information. The MDT stores metadata, and the OSTs store object data. There is also a management server (MGS) which coordinates traffic and requests. Object storage differs from block or file storage systems. Objects are generally unstructured pieces of data, with its metadata stored separately instead of all together. All of the Lustre components are presented as a single entity to the users. All they see if a mounted space on the computers that have access. The Lustre disk they see is actually made up of many OSTs/OSSs and one or more MDTs/MDSs. As OSSs contain many OSTs, and a Lustre system contains many OSSs, this means you can have very large Lustre disks. The purpose of this system is to increase the speed at which you can read data. Each OSS has a read speed, and all of them together gives the users a very fast aggregate read/write speed. With many disks available to Lustre, it often writes a single file in a series of stripes, which are typically 1MiB in size across disks. This is to spread your data more, so more OSSs can be engaged in the read process improving read speeds. The metadata is fetched from the MDS, then the read can begin. The MDS is considered a weakness for Lustre, if you aren't using Lustre as intended. Lustre is intended for very large files being read in parallel. This is because it’s strength comes from distributed files. Metadata operations are not fast or flexible for Lustre. This means many small reads is inefficient for Lustre. This poses a problem for container use. While containers can be rather large, the majority of read operations for a container are small. Starting an application in a container looks and feels just like starting a native application. When started only the parts of the container needed to run the application are read from the container. This means you generally will have many small reads of shared objects and imported files for run time, as opposed to reading the entire container. This is where the squash filesystem makes a great deal of sense. The metadata for the squash filesystem is compressed within a single file, along with all the data. As far as lustre is concerned, there is only one file. It cannot see the underlying directory tree. This can help make the most of Lustre. The parallel read speed is great at high node counts, and we mitigate the metadata weakness we would normally see for our use case.

Tools that require root can easily mount the squashfs for you, as the ability to escalate to root is already written into the tool. Mounting a filesystem typically requires escalation of privilege. Fortunately there are tools at our disposal that allow us to mount a squash file system without using the code in the linux kernel. File systems in user space (fuser or fuse) allows for the mounting, reading, and writing to file systems without the use of system calls or escalation of privilege. Fuse is commonly found on Linux systems. The way fuse works is whenever a filesystem is mounted through fuse it starts a daemon. Instead of communicating with the kernel, users send their filesystem requests to the fuse daemon. The fuse daemon has root privilege, and translates fuse requests into system calls to perform the requests to disk. Fuse is well supported, documented, and accepted in the Linux community. It enables users to securely mount and use disks and disk like objects without root. Squashfuse is an open source tool built to make the squash filesystem available through fuse. This is ideal for our us as we are interested in maintaining an unprivileged workflow. There is a minor hang up however. The translation from system call to fuse request and back to system call through the fuse daemon adds overhead to disk operations. The fuse API adds overhead, and slows down disk operations. There are two levels of fuse API. There is a low level fuse API, and a high level fuse API. The low level API is designed to be slimmer and faster, but not as portable. The high level API is more feature rich and focused of ease of use. Squashfuse builds a separate binary for each of these APIs. This means there are several options available for making use of the squash filesystem.

The squash file system is currently used in some feature rich container tools for HPC. Singularity and Shifter make use of kernel mounted squash file systems in their tool base to orchestrate container launches. In the past Shifter used the ext4 file system to create container images, mounting them on loopback devices similarly to the way they use the squash file system now. The idea behind this was to use lustres strengths while keeping metadata operations off the lustre metadata server. In 2015 they performed some benchmarking of container runtime using the Pynamic benchmark tool they had developed. They gathered several different methods for managing containers on different filesystems and mounts to compare which method was the fastest for run time. Shifter using the ext4 file system was nearly the fastest, losing out just barely to logical volume management over general parallel file system. It is mentioned in this document “none of these benchmarks test scaling” (Jacobsen & Canon, 2015), but they expect the shifter method to scale quite well. Further down the line, they tested the load time of the ATLAS simulation software on Shifter, Lustre, and Burst Buffer.

The kernel mounted squash filesystem via Shifter was a clear winner at all scales. There is an example of Shifter running the Pynamic benchmark at 4800 message passing interface (MPI) ranks compared lustre scratch, project space on NFS, temporary file system (tmpfs) which is simply having the container in memory on every node. The kernel mounted squash file system is a narrow winner, even over tmpfs.

The squash filesystem is a clear winner for the use of containers in HPC. However, the use of kernel mounted squash file systems is not entirely desirable in the HPC community. Mounting these requires root, which is a boundary many HPC centers do not wish to cross. Giving root access to users would allow them to use this method without needing to install a feature rich tool such as Shifter. Clusters are very carefully managed, and giving root to the entire user base is often out of the question. Using a tool like shifter moves that risk factor off of the user base, but onto the tool. This involves putting a great deal of trust in the developers of these tools to maintain an aspect of security for you. If you have ever used software ever, you are probably aware bugs happen. Security vulnerabilities are often found after software release. Whether it be a tool you use, or a tool your tools depend on. Many are uncomfortable with giving up that trust to others when they don't absolutely need to. Squashfuse provides user space tools to make use of the squash file system without the need for escalation of privilege. The use of fuse will create some overhead. Application start up time will likely be slower using fuse as opposed to a kernel mount, but it makes the squash file system accessible to users that would not ordinarily be able to use them.

For this experiment we will be using the Pynamic benchmark to evaluate the performance of container execution in HPC. Scientific software typically has to read a multitude of shared objects and libraries to perform work. The Pynamic benchmark is a python tool developed at NERSC designed to simulate such an event. Pynamic creates shared objects and libraries through a generator function. You can choose how many objects you make, and how many methods are in each shared object. It is designed to test a systems ability to dynamically link and load libraries for python applications in HPC, but represents a similar work flow of many scientific applications.

We will test several different methods of container distribution. We will use the squash filesystem mounted via kernel, squashfuse, and squashfuse_ll which is the low level fuse API variant of squashfuse. We will also a tmpfs solution, as this is a very common method of using containers in HPC currently. For the squash file system methods we will generate our image, and store the squash filesystem in Lustre. There are some tuneable factors we can experiment with in generating squash filesystems. The squash file system when created using the squashfs-tools uses a default block size of 131072 bytes, or 128 KiB. We can adjust this block size to be larger, up to 1 MiB. We will create both 128 KiB and 1 MiB squash file systems for testing. Lustre also has some tunable parameters. Lustre stripe size is commonly defaulted to 1 MiB but can range from 64 KiB to well beyond 1 MiB. We will use 64 KiB and 1 MiB. You can also choose how Lustre stripes your files. By default, Lustre will not stripe, and will place your entire file on a single OST. We will use no stripes, striping across two OSTs, and striping across all available OSTs which in our case is 32 OSTs. Pynamic will be configured with 495 shared object files, each containing 1850 methods. Our tests will be run across a closed cluster consisting of 1024 nodes available for execution. We will scale up from 1 node, to 1024 for the tests. We will run pynamic using openMPI in a container built in advance. The experiment will span across all factors sweeping out all listed possibilities. It is worth noting, Pynamic processes are individual processes that do not communicate. Say we are running on a single node with 32 processes per node, we will have 32 instances of Pynamic running, not one instance of Pynamic using 32 processes. The cluster being used will be unavailable to other users during experiment execution, so there will be no competing for resources. This means our results are to be presented as near ideal, as there is no competing activity from others on any system. The Lustre file system will also be private to the cluster, so there will not be noise or speed confusion caused by other users reading/writing.

Tuning lustre and squash file system parameters appears to be a lapse in the research area. While there are documented cases of these systems being used together, there is no evidence of anyone trying to optimize these existing features even in cases where the squash file system was benchmarked in some way with some tool. There are also no examples of anyone using userspace tools to mount squash filesystems. There are potential gains to be found, even for kernel mounted squash file systems in this experiment, but the focus is on the performance of user space squash file systems.

References

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

Jennings, M. E. (2017). *Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC*. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Priedhorsky, R., & Randles, T. (2017). Charliecloud: unprivileged containers for user-defined software stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17* (pp. 1–10). Denver, Colorado: ACM Press. https://doi.org/10.1145/3126908.3126925

Jacobsen, D. M., & Canon, R. S. (2015). Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., & Peterson, L. (2007). Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (Vol. 41, pp. 275–287). ACM.

Priedhorsky, R., Randles, T. C., & Jennings, M. E. (2018). *Containers are good for more than serving cat pictures⁉ Lightweight HPC containers with Charliecloud*. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Zhao, T., March, V., Dong, S., & See, S. (2010). Evaluation of a performance model of lustre file system. In *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual* (pp. 191–196). IEEE.

Le, E., & Paz, D. (2017). Performance analysis of applications using singularity container on sdsc comet. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* (p. 66). ACM.

Beserra, D., Moreno, E. D., Endo, P. T., Barreto, J., Sadok, D., & Fernandes, S. (2015). Performance analysis of LXC for HPC environments. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on* (pp. 358–363). IEEE.

Gerhardt, L., Bhimji, W., Canon, S., Fasel, M., Jacobsen, D., Mustafa, M., … Tsulaia, V. (2017). Shifter: Containers for HPC. *Journal of Physics: Conference Series*, *898*, 082021. https://doi.org/10.1088/1742-6596/898/8/082021

Pavlov, A. I., & Cecchetti, M. (2005). *Squashfs howto*. The Linux Documentation Project, http://ldp.pakuni.net/HOWTO/pdf/SquashFS-HOWTO.pdf

NERSC, (2018) http://www.nersc.gov/research-and-development/user-defined-images/