Userspace Squash Filesystem for Launching Linux Containers on HPC Systems

A Thesis

Presented to the Graduate Division

College of Arts and Sciences

New Mexico Highlands University

LA-UR-19-24107

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By Ronald Shane Goff

June 8th 2019

Userspace Squash Filesystem for Launching Linux Containers on HPC Systems

A Thesis Submitted to the Graduate Division

Department of Computer Science

New Mexico Highlands University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By

Ronald Shane Goff

Approved by Examining Committee:

Gil Gallegos Department Chair	Gil Gallegos Chair of Committee Discipline of Computer Science
Brandon Kempner Dean, College of Arts and Sciences	Richard Medina Member Discipline of Computer Science
Ian Williamson Dean of Graduate Studies	Reid Priedhorsky Member Discipline of Computer Science

ABSTRACT

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 kernel mounted squash filesystem (squashfs) has proven fast and efficient for this task but requires root-level access. This paper will show a user space mounted squashfs is an efficient and secure solution for container image distribution.

Table of Contents

List of Figures

List of abbreviations

API	Application Programming Interface
FUSE	Filesystems in Userspace
GPFS	General Parallel Filesystem
HPC	High Performance Computing
IB	InfiniBand
IP	Internet Protocol
LANL	Los Alamos National Laboratory
MDS	Metadata Server
MDT	Metadata Target
MGS	Management Server
MPI	Message Passing Interface
NERSC	National Energy Research Scientific Computing Center
NFS	Network Filesystem
OPA	Intel Omni-Path
ORNL	Oakridge National Laboratory
OSS	Object Storage Server
OST	Object Storage Target
RDMA	Remote Direct Memory Access
RPC	Remote Procedure Call
squashfs	Squash Filesystem
TCP	Transmission Control Protocol
tmpfs	Temporary Filesystem
UDSS	User Defined Software Stack
VFS	Virtual Filesystem
XDR	External Data Representation

Introduction

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 (Priedhorsky & Randles, 2017). This provides a great deal of stability to the software available to users. In addition, the software stack is often optimized for the cluster on which it is running. 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 one can request new software be added to the managed stack, it is unlikely to be installed without many users requesting the same software. Adding software to the managed stack takes resources to install and maintain the new software. Without enough people requesting the software, it isn’t always worth the resources to install it. If you cannot 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 for results to be accepted (Priedhorsky & Randles, 2017).

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 many 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 many require you to create your container image 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 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 to understand the container landscape. 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 is a container runtime tool developed at Los Alamos National Laboratory. Charliecloud handles the execution of your containers, 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 little 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 function is a Linux filesystem like directory tree (Priedhorsky & Randles, 2017). Containers are a simple, efficient, and relatively easy way of creating this directory tree for use with Charliecloud, but how you create that directory and deliver it is up to you.

Other tools are more feature-rich to support a variety of things for users and keep everything in the scope of their tool. Instead of directly using the supporting software, an interface to the supporting software is made. 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 several services to users such as an image gateway (Singularity Hub), image building (Singularity recipes), and mounting container image filesystems (https://singularity.lbl.gov/quickstart). It provides a unified interface for users to integrate different container tools such as Docker, or 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. This would mean having access to container build tools, in addition to run time all in one tool as opposed to handling them separately. However, as you can imagine the support effort required to achieve this available to the users can be high. Support burden is a concern with adding any new software and is a concern for supporting containers in HPC (Priedhorsky & Randles, 2017).

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. Using a feature rich container system can escalate concern as this can require the tools have the ability to execute some components as root. To manipulate containers, package them, create them and such the tool handling all the work can need root access. The build steps for container images often requires root. 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. Charliecloud relies on Docker or other tools not running on the cluster for container image building beforehand. This helps avoid having some level of root access on the cluster, as image build time is not built in.

The kernel mounted squash filesystem is a clear winner for performance of containers in HPC. It is fast and efficient for distributing container images to HPC resources. Singularity version 2.4 released October 2017 makes squashfs the default container image format for building containers with Singularity (https://github.com/sylabs/singularity/releases?after=2.4.2). Shifter release 18.03.0 in March 2018 removed support for all other filesystems excluding squashfs (https://github.com/NERSC/shifter/releases). However, the use of kernel mounted squash file systems is not entirely desirable in the HPC community. Mounting requires root, which is a boundary some 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 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 releases. 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. Additionally, Shifter provides services beyond container run time such as an image gateway. This means support cost beyond installing a package. You must set up and host an image gateway or allow connections to an outside server for a remote image gateway. 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 (Vangoor, Tarasov, & Zadok, n.d.). Application start up time may be slower using fuse as opposed to a kernel mount due to fuse overhead, but it makes the squash file system accessible to users that would not otherwise be able to use them and could be fast enough.

Literature Review

The Squash Filesystem is a compressed read-only filesystem in a file. 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. To mount 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 workflow. 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.

The squashfs provides some benefits to containers over distributing a tar ball. Squashfs is network mounted, so all compute nodes only read and load files that are required to start the container. Often in containers you start with a base image and add tools that you need for your application. Some of the files inside a container are waste, as they don’t require loading at run time for containerized applications but are required to build the application. Tools like compilers are used at build time to create libraries and executables, but aren’t involved in the run time. This can free up a bit of memory that would otherwise be occupied if you use a tarball for your container. Squashfs also provides file deduplication to reduce the amount of space it uses. This is again useful for the container user trying to minimize image size. However, being read only does create some challenges for scientific applications. Many applications open and write temporary files in arbitrary locations which could be inside the read only file system, causing run times to fail. Additionally, some users may wish to modify files in the container at or around run time. This can be subverted by adding the files you need beforehand, or by unsquashing the file system then adding the files and re-squashing the directory tree during job time. While manageable, this does create hassle for users.

FUSE

Many filesystems are traditionally developed in kernel space. The squashfs has an implementation in the Linux Kernel, and has an implementation developed using filesystems in userspace (FUSE). FUSE can be used to create filesystems that do not need direct kernel access (Vangoor, Tarasov, & Zadok, n.d.). Instead of directly using the kernel FUSE starts user space daemons to communicate with the kernel side FUSE driver. The FUSE driver is trusted and handles kernel calls for the user space daemon. Filesystems written using FUSE do not require root privilege operations and can be mounted by any user. There are other filesystems designed in user space that do not use FUSE such as General Parallel Filesystem (GPFS), but FUSE is the most popular method of delivering filesystems in user space (Vangoor, Tarasov, & Zadok, n.d.).

A device is registered by the FUSE driver /dev/fuse, which acts as an intermediate space that the daemon and driver can communicate through. A driver for Linux Virtual Filesystem (VFS) is also created by the FUSE driver. When filesystem operations are made to a FUSE mounted device by a user application they are sent to VFS, which sends the operations to the FUSE driver. The driver creates FUSE requests, and places them in the FUSE queue while they wait to be serviced. The unprivileged daemon process then reads the requests from /dev/fuse, and performs the operations to the underlying filesystem if there is one, or if your FUSE filesystem is a block device it reads directly from the block device. When the request is complete the daemon writes the response back to /dev/fuse. The FUSE kernel driver then marks the request complete and wakes up the user's application process.

When people mention FUSE filesystems they often don't think of them as having good performance (Vangoor, Tarasov, & Zadok, n.d.). The aforementioned process of how using FUSE complicates reads and writes is overhead for disk operations, which are often already thought of as one of the slower processes in computing. However, FUSE has improved over the years, and can seemingly deliver great performance. An optimized FUSE filesystem called Stackfs was benchmarked against an ext4 filesystem actually performed faster in some cases. This was not the case for all workloads. While most workloads FUSE performed within +/- 5% of native ext4, some workloads appear unfriendly to FUSE (Vangoor, Tarasov, & Zadok, n.d.). Specifically, file creation on FUSE filesystem tested suffered steep performance degradation on both hard drives and solid-state drives.

The FUSE library is composed of two levels. The low level is responsible for handling requests from the kernel, sending replies, file system configuration, mounting, and hiding the difference between kernel and userspace (Vangoor, Tarasov, & Zadok, n.d.). This part exports the low-level FUSE application programming interface (API). The high-level FUSE API builds on top of the low-level API, and it allows developers to skip path to inode mapping. The high-level API is feature rich and shields the developer from the low-level API, making it easier to develop with. However, you do give up the precise control and potential optimizations of the low-level API. The high-level API is a little slower, but is more portable and easier to develop with.

Squashfuse is an open source tool built to make the squash filesystem available through FUSE. There are two levels of fuse API. There is a low-level fuse API, and a high-level fuse API. Squashfuse builds a separate binary for each of these APIs, squashfuse_ll and squashfuse respectively. Stackfs was written only using the low-level FUSE API to maximize performance (Vangoor, Tarasov, & Zadok, n.d.). Having access to both APIs with squashfs will shed some light on the real performance implications of the high-level API versus the low-level API.

OpenMPI/MPI

Message passing interface (MPI) is a communication protocol that has become the standard for HPC. MPI is supported by a variety of industry and academic organizations and is widely accepted as a standard. There are multiple implementations of MPI available to choose from such as OpenMPI and MPICH. The MPI interface is meant to provide synchronization and communication between processes. MPI works on multiple platforms and programming languages such as C, C++, and Fortran. MPI is used in HPC applications because of its speed and portability. MPI implementations come with the tools to start and map all processes needed across the entire computer cluster such as mpirun. While communication between processes is often of the highest importance the application we are testing with does not perform process communication. MPI is being used to start processes, but not using the mpirun provided. As there are many options available for choosing an MPI implementation OpenMPI was chosen because it is open source and widely accepted. This choice is arbitrary, as any implementation could realistically be chosen.

High speed networking IB/OPA

High performance computers are typically built with special network hardware such as InfiniBand (IB) or Intel Omni-Path (OPA). High performance networking hardware is designed to have low latency and high bandwidth to service the expected traffic of high-performance computing. Having a lower latency means quicker network responses when communications occur. When data is requested by a client from a remote filesystem it takes time to send signals from the client to the host. Lower latency minimizes the time spent waiting for a request to get a response. High bandwidth is also important in a high-performance network. Many jobs will need to transfer large amounts of data from network storage to clients. With more bandwidth available you can achieve higher transfer rates.

The way high performance networks achieve such low latency and high throughput is Remote Direct Memory Access (RDMA). RDMA is a zero-copy network system supported by the network adapter and enables memory access to remote machines without the need to copy the memory between buffers in the operating system. This keeps the CPU of the machines free to continue their normal business rather than spend time copying data between buffers. This is accomplished using verbs, which is a set of functions designed to work with high performance network hardware.

Not all applications and tools need to use RDMA to take advantage of high-speed networking hardware. High speed networks such as IB can also use some network protocols outside of verbs. Internet Protocol over InfiniBand (IPoIB) is also available, which is built on top of RDMA. This provides network protocols to tools that don’t or can’t take advantage of RDMA.

The Pynamic Benchmark

For this experiment we will be using the Pynamic benchmark to evaluate the performance of container execution in HPC. Scientific software sometimes must 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. Not all operations on containers will look like a ‘typical’ HPC workflow so we will also perform a recursive grep across the entire filesystem. While this type of workflow is not expected it is bound to arise for someone. This type of workflow would likely perform poorly, but someone will try it. The Pynamic benchmark reports several different timings during execution. Each process will report the timings only for that process. Pynamic reports python startup time, the time required to import all shared object files, the time to call a visit function in each shared object, and finally a small MPI calculation time. We are primarily concerned with application startup times are what we are interested in, but the MPI time will be included in our total timings. Each of these timings are reported by each process. To get a total time, the pynamic output is all piped to a single file. The longest of each individual timing is collected for the job and reported at the end.

NFS

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 a communication protocol built on Remote Procedure Call (RPC) and External Data Representation (XDR). RPC enables a computer to execute a procedure on a remote machine as if it were a local procedure like opening and reading files. The core design of NFS is easy recovery, independent of transport protocols, operating systems and filesystems, simplicity, and good performance (Shepler et al., 2003). NFS uses transmission control protocol (TCP) to communicate requests from clients on the network to the daemon running on the server which performs the actual reads and writes to disk NFS is made available to other computers on the network, such that they can all treat the NFS share as a local disk. Systems with the NFS share mounted can read and write to the disk hosted by another computer. The underlying filesystem can be chosen by the individual setting it up. This system is quite effective at creating a single shared space all nodes can access. However, all nodes must make their requests to a single process running on a single node for all disk operations. While NFS is a distributed operating system typically a single host is responsible for answering all RPCs from clients. NFS version 4.1 aimed to add support for clustered server deployments to provide scalable parallel access to files distributed across multiple servers in 2010. NFS v4 was ten years old before gaining acceptance (Chen et al., 2015), NFS v4.1 is likely to see similar resistance. NFS is not exactly suited towards serving the needs of an at scale container launch we will still perform some tests with it.

Lustre

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. Lustre operates quite differently compared to your garden variety UNIX file system. Lustre stores data as objects, as opposed to files or blocks. For this to happen there are several components of Lustre that require explanation. Lustre uses a set of servers to store and host information for different purposes. Metadata, which is data describing your data is written to a metadata 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 its 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, read operations for a container can be quite small. The files in the container are those you would find in a Linux operating system distribution, plus whatever libraries and binaries you have added. 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 when using an unpacked file system like tree in Lustre.

There is a great deal of information regarding optimizing Lustre. Most of the information is contained in ‘how to’ type guides available on the internet from trusted sources such as the National Aeronautics and Space Administration, and the National Energy Research Scientific Computing Center (NSERC). They all advise some things in common such as avoid metadata operations as much as possible, and avoid serial IO operations. An issue we face with how to optimize Lustre is our use case being different than what most users expect for Lustre. Some users have very large files, some have very small files. Most users expect to read the whole file when they open it. We have a semi small file of 1.3GiB containing many small files. Pynamic generates a variable number of library files that are pseudo-random. There is a great deal of repeated text in these files. The container image is roughly 9GiB, but flattened and compressed to squashfs we get a much smaller file. Only parts of the file are going to be read rather than the entire file. Similarly, Lustre optimizations are intended for users performing IO operations on files with the flexibility of deciding how they read and write those files. We do not have that choice, we simply are starting a daemon across all nodes, and they will all perform IO as they will. But we will still try to follow optimization guidelines. With that in mind we will have many processes reading a single file of unimpressive size. With a many to one (N:1) read pattern having more stripes will improve read times the more processes there are. When N is small, having more stripes won’t do you very much good here, but when N is large having more stripes means less blocking occurs and processes can read faster. An experiment at Oakridge National Laboratory (ORNL) they observed “the stripe size does not affect the I/O bandwidth of a single OST” (Weikuan Yu, Vetter, & Sarp Oral, 2008), and that increasing stripe size hurt performance for both reads and writes on a Lustre system using stripes. A Stripe size of 1MiB and 4MiB performed well but increases above 4MiB resulted in far lower performance up to 64MiB. For contiguous read operations increasing the stripe size reduced performance. The IO pattern for containers is likely not contiguous, but the results could still be important. Starting up a container involves reading many small pieces of our squashfs. The experiment only appears to increase the stripe size instead of also using smaller stripes than the default 1MiB.

Shifter Benchmarking

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 Luster’s 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.

Method

The compute cluster provided has 1024 useable compute nodes. Each compute node has two Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz with 8 processor cores and 16 threads each, 32GiB DDR3 memory, and a QLogic 7322 quad data rate InfiniBand networking card for high speed communication. Each node is running CentOS 7.5 with Linux kernel 3.10. Both NFS version 3 and Lustre version 2.10 are available to all compute nodes. NFS is configured on top of an XFS filesystem. Lustre is configured to use ZFS, has a single MDS and MGS, with thirty-two OSTs. Fuse version 2.9.8 was installed from the yum packages in the EPEL repository for CentOS. Squashfuse version 0.1.103, and Charliecloud version 0.9.4 were installed from source made available on GitHub in NFS. OpenMPI version 2.1.5 was installed from a release tarball in NFS. Pynamic will be configured with 495 shared object files, each containing 1850 functions. Pynamic was built in a virtual machine using docker and is built to use OpenMPI version 2.1.5 in the container. Pynamic was built with 495 basic files, each with 1850 functions, 215 utility files, each with 1850 utility functions. The exact command was "./config_pynamic.py 495 1850 -b -e -u 21 1850 -n 100". Pynamic processes are individual processes that do not communicate. Each node will have an independent instance of Pynamic running across all 32 processes, so each node will report timings. The final timings will be combined to create a "total time" in addition to the timings reported by Pynamic.

Squashfs configuration, and mount methods

We will test several different methods of container distribution for squashfs. We will use the squash filesystem mounted via kernel code, squashfuse, and squashfuse_ll which is the low-level fuse API variant of squashfuse. We will generate our Pynamic container image using Docker and Charliecloud, then store the image as a series of squash filesystems in Lustre. There are some tunable factors we can experiment with in generating squash filesystems. The squashfs when created using the squashfs-tools uses a default block size of 131072 bytes. We can adjust this block size to be smaller or larger, up to 1 MiB. We will create both 128 KiB and 1 MiB squash file systems for testing. In previous experiments increasing the Lustre stripe size had no benefits for read speeds over 4MiB stripes, and performed basically the same as 1MiB stripes. We will use 1MiB, and 64KiB stripes sizes for Lustre. Lustre stripe counts are slightly more straight forward. More stripes will perform better as scale increases. We will use no striping, striping across two OSTs, and stripe across all thirty-two OSTs.

Optimizing the system in this situation may prove to be tricky due to the layers of interaction with other tools. We have a squashfs with its own block size and compression, sitting on top of the Lustre file system with its own block sizes and stripe patterns, being used by fuse or the kernel on each node.

Our tests will be run across a closed cluster, with no other users present competing for resources. All resources are hosted on the cluster, including the filesystems. We will run tests at 1, 128, 512, and 1024 nodes. 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 in performance, 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 performance impact caused by other users reading/writing.

Each iteration of the experiment will mount a different squashfs from Lustre. Pynamic will be run, and the longest of each timing reported will be kept. Each iteration will be completed five times.

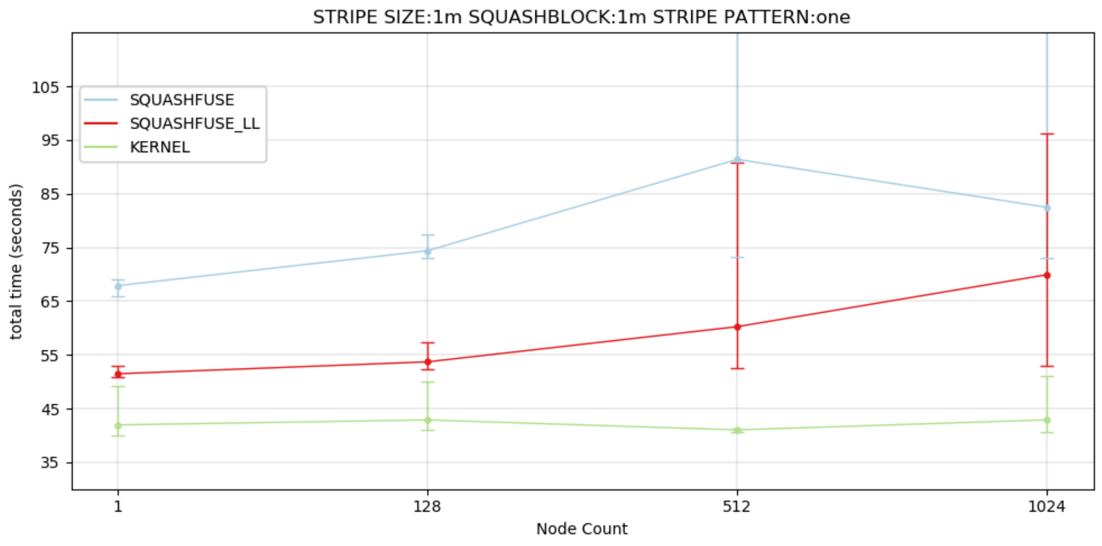
Factors	**Levels**
Node Count	1,128,512,1024
Mount Method	kernel, squashfuse, squashfuse_ll
Squashfs Block size	128Kib, 1MiB
Lustre Stripe Size	64KiB, 1MiB
Lustre Stripe Pattern	1 OST, 2 OSTs, 32 OSTs

The squashfs can be useful for non HPC purposes as well with containers. Some iterations will be run on NFS as well. While unlikely to be suited for large scale computation and containers, testing limits for squashfs is important. We will also perform a recursive grep `grep -r “testing” $path_to_mount > /dev/null` across the entire squashfs in Lustre, to see how it performs for workloads different than what we intend. All 32 processors will be engaged in performing the exact same task of walking the filesystem and searching the contents of each file on every node.

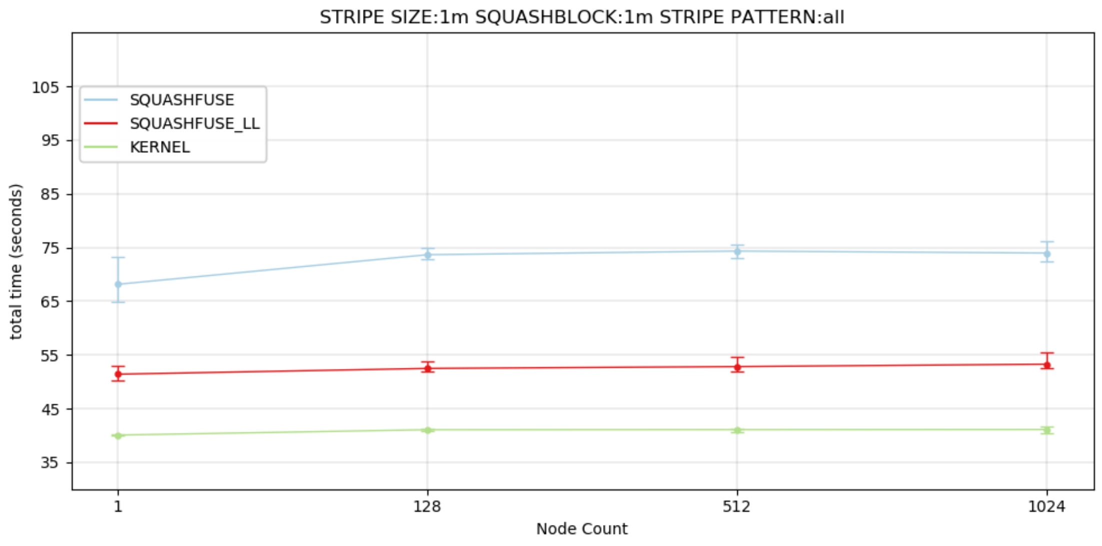
Results

The kernel mounted squashfs is the king of the crop in all respects at all factors as expected. The only factor to have any impact at all on the kernel mounted squashfs is the Lustre stripe pattern. With no striping the performance can in some cases decrease resulting in longer run times, but the difference is quite small even at scale. Other experiment factors have to clear benefit or detriment to using the kernel mounted squashfs. The performance of the kernel mounted squashfs provides a baseline to measure other mount methods and distribution methods.

Lustre stripe patterns show valuable impact for both FUSE mounted squashfs methods. Striping across two OSTs up from one provides improvement in stability for run times. The variance between the average, maximum, and minimum run times are reduced just by adding a single stripe. For 1MiB Lustre stripe sizes there was also a small performance increase as the number of nodes increases over no striping. But for 64KiB Lustre stripe sizes the performance difference was not consistent. Sometimes using two stripes was faster than no stripes, sometimes it was not. Striping across all 32 OSTs makes these benefits clearer. Up to 128 nodes average run time for Pynamic was rather close between no stripes, two stripes, and all stripes. The variance is also rather small. But at 512 nodes and above having two stripes shows a clear benefit of reducing variance of run time from no stripes to two stripes. Average performance is similar between one and two stripes, but is clearly faster as stripes are added as scale increases. Using all 32 stripes also improves stability at scale. For all stripe sizes and squashfs block sizes increasing the stripe pattern shows improvement to both run time and stability of run time.

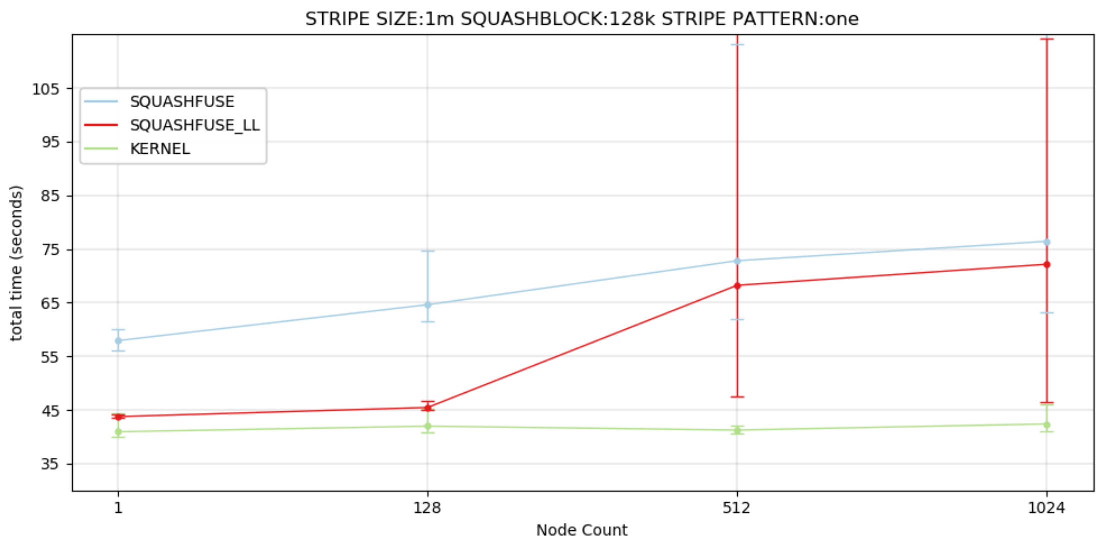


The above graph shows the average total time for Pynamic at all node sizes for no stripes, a squashfs block size of 1MiB, and a Lustre stripe size of 1MiB. The minimum and maximum are shown as flat caps below and above the average which is plotted as a dot. (Figure 1)



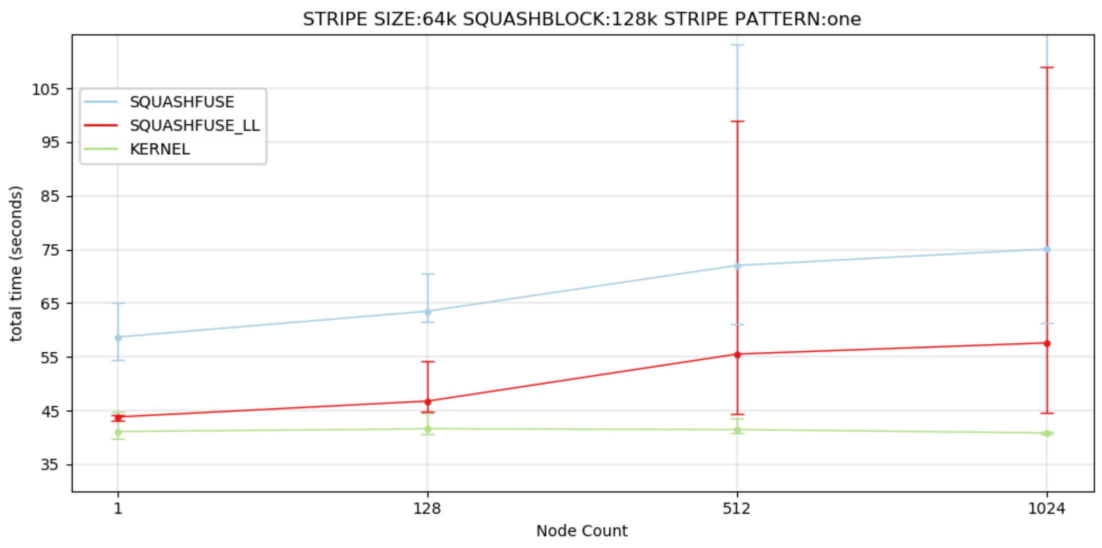
Increasing the number of stripes used improves run time as scale increases, and reduces variance in performance times at all scales. (Figure 2)

The Lustre stripe size does have some impact on the FUSE mounted squashfs. With no striping the performance improvement is clearest. For both the low-level and high-level FUSE APIs having a smaller block size improved run time as scale increases. The high-level FUSE mounted squashfs has fewer clear benefits, while the low-level appears to have more clear benefit from smaller stripe sizes.



Default for all settings. Reducing only the squashfs block size from 1MiB to 128KiB shows performance improvements at smaller scale. (Figure 3)

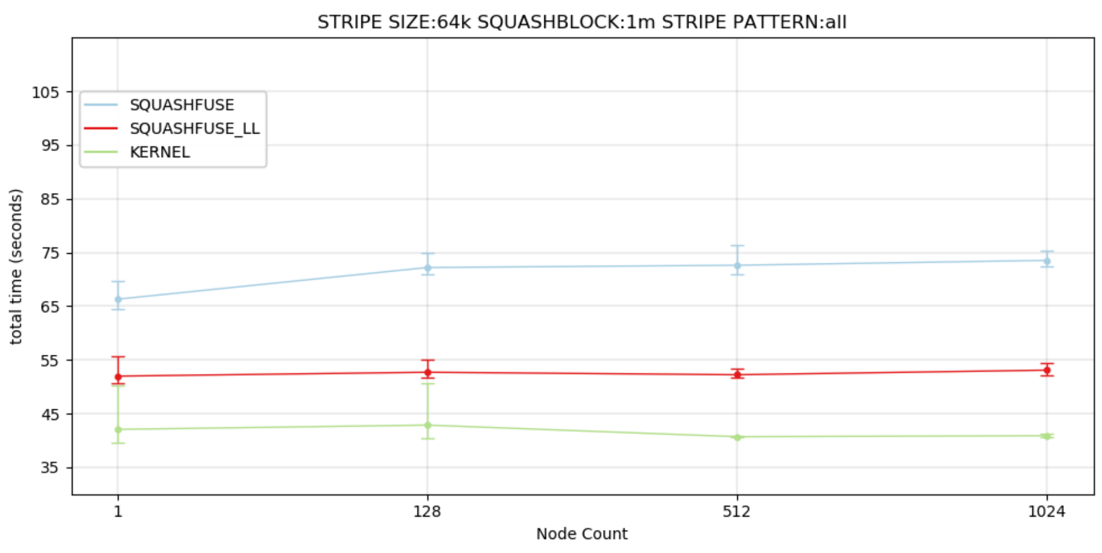
Squashfuse low-level shows performance improvement when reducing stripe size from 1MiB to 64Kib.



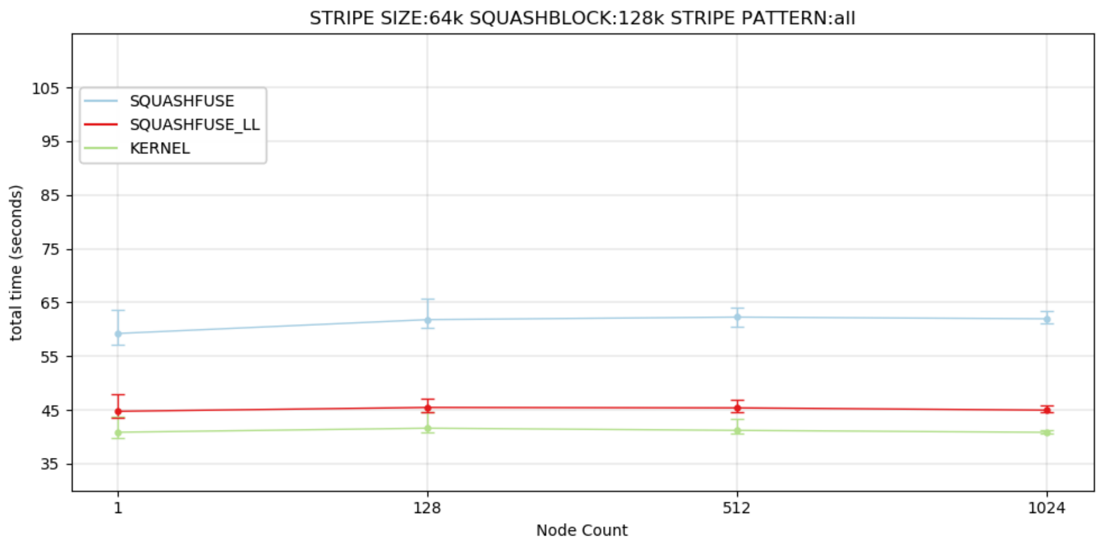
Using a smaller Lustre stripe size improves run time for FUSE based methods. The improvement shows more as scale increases. (Figure 4)

Using two stripes the results are more mixed and appear less meaningful. It appears the benefit of smaller stripes continues when striping across more OSTs. However, the difference between stripe sizes when striping across all OSTs is insignificant for both FUSE APIs despite a small improvement in run time.

The squashfs block size shows important changes in performance across all factors of the experiment. Both FUSE APIs at all node sizes, stripe sizes, and stripe patterns have better run time when using a smaller squashfs block size. Increasing the squashfs block size only increased run time across all factors.



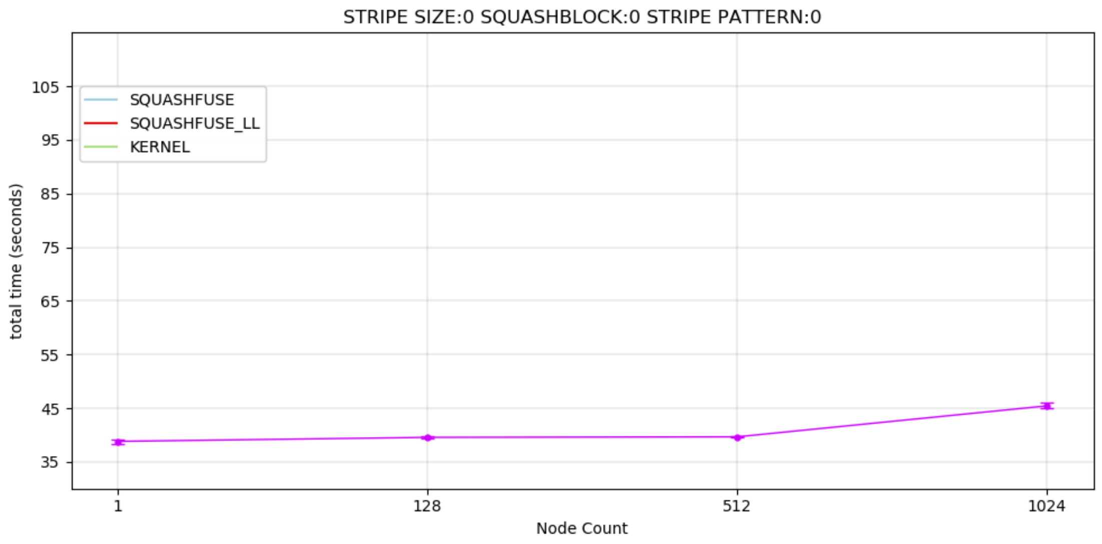
Increasing the stripe pattern to use all available OSTs improves performance and reduces variance in run times. (Figure 5)



The most optimal configuration for FUSE based methods. (Figure 6)

Finally, running Pynamic from a container image in memory was tested. The distribution time of the tarball was not measured in this case, only execution time. This is a different measurement compared to the squashfs methods. For the squashfs methods the distribution time is baked in to the run time. With an in-memory container image you must first distribute the image before you can run.

The distribution time was not collected making it an apples to oranges comparison. As the image is not in Lustre and not a squashfs our experiment factors do not exist in this test.



In memory pynamic execution run time is very similar to the run time + image distribution time for kernel mounted squashfs, and squashfuse_ll mounted squashfs. Image distribution time for in memory not measured or shown here. (Figure 7)

The performance of an in-memory containers execution only of Pynamic is quite close to the time it takes for the kernel mounted squashfs to load files and execute. At 1024 nodes in memory performance is actually closer to squashfuse_ll. At 1024 nodes there is a clear rise in Pynamic execution time in memory, but not with squashfuse_ll when optimized.

In addition to the tests performed on Lustre some iterations were performed on NFS, just to examine limits and performance despite expectations being low. The experiment was run only with the default squashfs block size, and only for limited node sizes. Starting small and scaling up by powers of two it wasn't long before run time performance took a dive. At small node sizes running on NFS is no problem. But anything above 32 nodes Pynamic suffers. The NFS testing ended at 64 nodes, because Pynamic run time is around ten minutes.

Performing a recursive grep across the entire squashfs has very divided performance between the high-level and low-level FUSE APIs. The low-level API sees much faster performance, but hits a wall where it simply stopped working as scale increases. At 64 nodes the low-level API appears to be unable to complete the recursive grep. The high-level API however, continues. The high-level API runs drastically slower than the low-level API, but finished the job at 64 nodes after around 500 seconds.

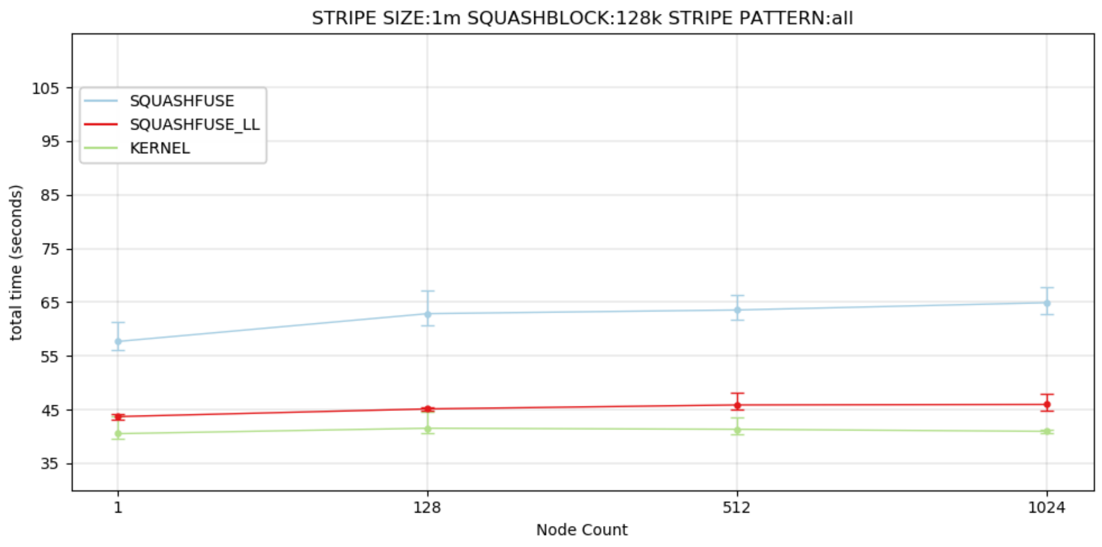
Discussion

The low-level FUSE API was difficult to troubleshoot for recursive grep. There were no errors, no stack trace, or exit codes, the processes just hang. When mounting squashfs there are options to get debug information keeping the FUSE daemon in the foreground. Unfortunately, when using this for recursive grep no information is displayed when using the low-level API. This kind of workload is not one you would expect to find in HPC. But expectations are not always aligned with user behavior so it is important to think about oddball edge cases, and non HPC workloads as well.

The performance difference between high-level and low-level FUSE APIs for this task is very steep. The low-level has superior performance than the high-level, with the caveat that when you are performing a recursive filesystem walk and grepping for a string in every file with 32 processes the low-level API seems to quit at 64 nodes. Despite this, it is unclear as to why someone would be doing this. It is clear that the low-level API performs some operations much faster than the high-level API. Metadata operations specifically are faster for the low-level API, making many metadata operations such as a filesystem walk faster. This speed difference is most obvious in a test of an extreme case, but the speed benefits still exist for a single process performing operations on the squashfs. The one advantage for the high-level API here is it appears to continue to work when the low-level dies. At 64 nodes the low-level API was unable to complete the filesystem walk, but the high-level did finish despite taking a great deal longer. For workloads that don't require performance the stability of the high-level API could be desirable.

Testing on NFS shows what was expected. While NFS V4.1 has some parallel components available NFS V3 does not. Even if a version of NFS was to be used it is unlikely to have the parallel performance desired for container image distribution. The limits and performance of parallel NFS were not really considered for this experiment. However, NFS can still be used for small containers that don't actually have an HPC workload using similar tools like Charliecloud for some purposes. Often in HPC as there are many versions of many different software applications centrally managed you need a way to switch from one set of tools to another. An example of this would be Environment Modules, which can be used to automatically set your PATH environment variable to all the right places. This means you can have a great deal of software managed, and users can select the things they need. In some cases, this isn't available. Instead of setting this up and loading modules for software that was not provided a container with all the software needed was created and kept on hand in NFS. A great deal of work is involved in running experiments on HPC resources, such as data processes and visualization. These tasks aren't HPC tasks, just basic workflow and things that need to be done. All of that work can be done with software unavailable to a cluster using Charliecloud and the squashfs. Just prepare a container with the desired software stack and away we go. Fortunately, this means NFS and even the high-level FUSE API are easy to use and fast enough for non HPC tasks. This does share a drawback with using a managed HPC stack. If you need to make changes to the software stack, you have to rebuild the container and make it available to the systems that will need it, creating some maintenance and slow down.

Concerning the performance of container run time with squashfs there is still a bit to unpack. Our Lustre system uses a default stripe size of 1MiB, and does not stripe across OSTs. The low-level FUSE API using all default settings (Figure 3) performs quite well up to 128 nodes. The variance between average and minimum/maximum is small, the speed is also quite close to the kernel mounted squashfs. If you increase from no striping to striping across two OSTs, the low-level API performs similarly well at 512 nodes, but sees a performance hit at 1024. For users running jobs at less than 512 nodes, it is possible they don’t need to do anything in any way to have close to optimal performance. Even with no striping the performance impact can be small, and only really impacts the time it takes for the job to start. While the variance is high, most users would probably be ok with running default settings all the time for FUSE based squashfs. Users scaling up beyond 512 nodes, or any user looking to make the most of things can get most of the performance improvements just by striping their squashfs in Lustre. Two stripes seem fine as scale goes up, many stripes are ideal. Most systems will likely have more than two OSTs you can take advantage of. Our system had thirty-two OSTs, when we stripe across only two the other thirty are left idle, or more available to other users if there were any. The more striping, the less variance there is, and runtime improves as scale increases.



Default squashfs block size, and Lustre stripe size, striping across all 32 OSTs. This configuration is close to the fastest, and requires almost no modification to settings for users (Figure 8)

Many users are educated about Lustre, and how to pick the right striping method for your work by their data center administration. Without reading this paper at all, many would expect to increase the number of OSTs you stripe across as scale increases. With a performance difference of a few seconds between a single node, and 1024 nodes for the low-level API users can really get the bulk of benefits just by increasing the number of OSTs striped across. Users who do not run at scale who would maybe not have this information wouldn’t need to act, as the low-level FUSE API is relatively fast with no effort at all up to at least 128 nodes.

Overall the system at optimal setting (Figure 6) is smaller stripe size and smaller squashfs block size. The difference between 1MiB stripes and 64KiB stripes is almost nothing. For the low-level API the difference between the Luster stripe sizes is so small you can barely see it. Fortunately, the high-level API is slower and has a slightly larger reflection of the performance increase. The improvement in either case is likely not worth doing, unless you’re really showing performance problems.

Increasing the squashfs block size also hurt performance, rather than helped. Changing the block size also had a much more notable impact on performance than the Lustre stripe size. Increasing the squashfs block size from 128KiB to 1MiB showed decreased performance across all other factors. Comparing results in figure 5 and figure 6, we see that both FUSE mount methods showed slower performance with a 1MiB squashfs block size compared to 128KiB block size. Similarly, if we compare figure 8 to figure 2, there is an increase in run time with an increase in squashfs block size that appears unaffected by stripe size.

Lustre comes equipped with some tools that allow us to harvest information about the RPCs sent during a Slurm job called Lustre Jobstats. Read stats for each OST are recorded for a job and can be viewed. The stats include minimum, maximum, and average request size. How many requests total were sent to each OST, and the sum of all those requests. Using Lustre Jobstats we can see for each OST how many bytes were requested total, and the minimum and maximum bytes read by a single RPC. While there are many stats you can get, we are really only interested in the read stats. Using information gathered from Lustre Jobstats, it would appear Lustre is in no way bound by squashfs block size, nor stripe size when reads are performed. When using 1MiB stripe sizes, and 1MiB squashfs block size, Lustre did not perform a single 1MiB read when launching Pynamic with a FUSE mounted container image. Across all 32 OSTs there was never a single instance of reading a single whole megabyte during job time. This could have something to do with FUSE having a maximum request size of 128KiB. While Lustre did receive RPCs for reads larger than 128KiB, it is unclear how many there were and how much of the total reads were composed of these larger reads. This also may explain why increasing squashfs block size didn’t improve read times. How does FUSE organize reads under the hood? If there is a contiguous 1MiB block needed does it line them up and have Lustre send a single large RPC? If you have a 1MiB squash block, but only need one file from the block does it read and decompress everything up to the file needed but not anything after?

There were no attempts to optimize FUSE in this experiment. There appears to be some extra overhead from having larger squashfs block sizes. This could potentially be resolved by modifying the maximum page size in FUSE, which is 32. With this default only 128KiB FUSE requests can be made. If FUSE were to be changed and a custom kernel compiled to support FUSE requests of 1MiB this overhead could be minimized, making 1MiB squashfs block size more competitive. However, the results as is are likely much more practical. Why force yourself to compile a custom kernel for compute node images to make a more complex solution work when doing almost nothing is already quite fast? It would be prudent to test this sometime in the future, to see if there were some speed gains to be had. But ideally the least amount of work that must be placed on users is the way to go. Using nonstandard configurations and tweaking everything can result in more speed but requires more education and work for users. The speed improvement would have to be quite high to put in the effort. More in between steps for block size and stripe size may have helped with stronger conclusions. Aside from the default block and stripe sizes only a single other size was sampled. The minimum Lustre stripe size was used, and the default but nothing between. The squashfs can also be created using smaller block sizes than the default. With the results at hand it can be stated that the smallest block size used was fastest. It is possible that even smaller block sizes for squashfs could improve performance as the block size gets smaller. In the future the squashfs block size and Lustre stripe size should be expanded for another sweep of the experiment.

Conclusions

There is additional work to be done regarding this experiment. No attempt at optimizing FUSE was made. Newer versions of FUSE will likely have performance differences compared to the version used in the experiment. Notably, in FUSE v3 the FUSE request size is based on kernel configuration, rather than defaulting to a static 128KiB. More time should be invested into identifying improvements made in FUSE v3 for possible performance increases to the FUSE mounted squashfs. There are configurations that could be made to FUSE that were never explored.

At some point it was expected the FUSE mounted squashfs would see performance issues as scale increases. The overhead of FUSE seems to be a single performance hit that does not increase as the number of compute nodes increases. The performance is much more related to the underlying filesystem, in this case Lustre. When Lustre has reached its limits the squashfs will eventually suffer. But eventually this limit should be reached, and a solution should be explored. At 1024 nodes it appears a 32 OST Lustre share can provide more than enough read speed to satisfy container image distribution with squashfs. Eventually an iteration using more compute nodes will be run. Ideally enough nodes to confirm the squashfs performance is tied to the backing filesystem rather than the squashfs or FUSE. Los Alamos National Laboratory houses the Trinity supercomputer which has roughly nineteen thousand compute nodes available. That might be enough nodes to push Lustre to the limit during a container launch. At some point the experiment will be repeated using only user space squashfs.

Trinity also provides a different backing filesystem for testing the squashfs. Trinity has a flash storage-based file system called Burst Buffer.

There are several concerns to be addressed regarding the use of the FUSE mounted squashfs. The kernel mounted squashfs requires granting some form of root to user run software and containers on a cluster to be useable. In some computing environments this isn’t an option due to security concerns. Using FUSE allows container users to mount a squashfs without root. While FUSE does have a trusted kernel driver, it is open source, public, mature, and considered to be secure. The other tools required to use a FUSE mounted squashfs do not require root. Some other container tools use setuid wrappers to manage the squashfs mounting and unmounting, which escalate privilege to the tool. Charliecloud does not currently make use of root in any way to execute container images.

Support burden for introducing the extra tools to use the squashfs is minimal. All tools required for supporting squashfs are available as distribution packages that can be installed to the compute node image. Once those packages are added the tools will be installed to all nodes, similar to adding any other package. The only additional tools absolutely required on compute node images to run squashfs containers are FUSE and squashfuse.

Finally, container image execute performance for the user space squashfs compared to in memory or kernel mounted squashfs containers is addressed.

The performance impact on the FUSE mounted squashfs compared to kernel mounted squashfs can be managed. The kernel mounted squashfs is a brainless plug and play solution that works well with no thought of performance. The performance overhead when using the FUSE mounted squashfs varies. If proper Lustre striping is done the low-level FUSE API performs very closely to the kernel mounted squashfs. The performance difference between the two is rather small, and doesn’t appear to get worse with scale. The high-level FUSE API performs slower than the low-level as expected, but could be useful for containers that don’t run the way it has been anticipated in this experiment.

Using the FUSE mounted squashfs reduces attack vectors, and makes container execution more secure for users. The effort exerted to support the squashfs is minimal for cluster administration. When properly optimized the FUSE low-level API mounted squashfs performs very similarly to the kernel mounted squashfs.

References

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

Priedhorsky, R., & Randles, T. (2017). Charliecloud: unprivileged containers for user-defined software stacks in HPC. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17*, 1–10. 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*.

Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., & Noveck, D. (2003). *Network file system (NFS) version 4 protocol*.

Chen, M., Hildebrand, D., Kuenning, G., Shankaranarayana, S., Singh, B., & Zadok, E. (2015). Newer Is Sometimes Better: An Evaluation of NFSv4.1. *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '15*, 165–176. https://doi.org/10.1145/2745844.2745845

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

Weikuan Yu, Vetter, J. S., & Sarp Oral, H. (2008). Performance characterization and optimization of parallel I/O on the Cray XT. *2008 IEEE International Symposium* *on Parallel and Distributed Processing*, 1–11. https://doi.org/10.1109/IPDPS.2008.4536277

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

Vangoor, B. K. R., Tarasov, V., & Zadok, E. (n.d.). *To FUSE or Not to FUSE: Performance of User-Space File Systems*. 15.