Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Dealing with small Files in HPC Environments: automatic Loop-Back Mounting of Disk Images

Marcin Krotkiewski[*]

University of Oslo/SIGMA2

Abstract

Processing of large numbers (hundreds of thousands) of small files (i.e., up to a few KB) is notoriously problematic for all modern parallel file systems. While modern storage solutions provide high and scalable bandwidth through parallel storage servers connected with a high-speed network, accessing small files is sequential and latency-bounded. Paradoxically, performance of file access is worse than if the files were stored on a local hard drive. We present a generic solution for large-scale HPC facilities that improves the performance of workflows dealing with large numbers of small file. The files are saved inside a single large file containing a disk image, similarly to an archive. When needed, the image is mounted through the Unix loop-back device, and the contents of the image are available to the user in the form of a usual directory tree. Since mounting of disks under Unix often requires super-user privileges, security concerns and possible ways to address them are considered. A complete Python implementation of image creation, mounting, and unmounting framework is presented. A seamless integration into HPC environments managed by SLURM is discussed on an example of read-only software modules created by administrators, and user-created disk images with read-only application input data. Finally, results of performance benchmarks carried out on the Abel supercomputer facility in Oslo, Norway, are shown.

1. Introduction

Modern storage solutions for HPC environments strive to provide scalable IO performance for supercomputers with thousands of compute nodes. This goal is challenging considering the large variety of applications, and their CPU and IO characteristics. When it comes to IO, some jobs require large bandwidth, while others may perform many meta-data operations, random IO access, and short reads. On top of that, modern HPC facilities serve hundreds of users and often support a large application base. It is common that parallel network file systems are used to host home directories, which often contain large numbers of small files, and to provide centralized software and development stacks to the compute nodes, in addition to providing compute jobs with space for input/output data.

Most modern storage architectures are based on specialized server nodes interconnected with high-speed, low-latency networks. Dedicated storage servers provide clients with high total network bandwidth and scalable access to the back-end disk arrays. On the other hand, meta-data servers deal with file name, modification date, and operations like file creation, deletion, opening, closing, and locking. Communication with the storage infrastructure is usually implemented in the form of a network file system (FS), e.g., BeeGFS [1], Lustre [2], GlusterFS [3], OrangeFS [4]. The role of such global parallel FS is to provide fast, parallel storage, to share fairly the IO resources among the clients, and to assure data integrity and a consistent data state across thousands of independent clients that can at the same time read, and write data.

Depending on the characteristics of the software run by an HPC facility the number of server nodes and their configuration can be adjusted to provide the best overall storage performance in a cost-effective manner. This is of course important from the perspective of the system administrators, but more importantly from the user

[*] Corresponding author. *E-mail address*: marcin.krotkiewski@usit.uio.no

perspective is to achieve high performance with his or her individual workflow. These expectations are not always both met. One example that is notoriously problematic for all modern parallel file systems is processing of large numbers (hundreds of thousands) of small files (i.e., up to a few KB) [5, 6, 7]. While large files can be split into smaller chunks and saved in parallel across many servers, small files are usually stored sequentially in a single chunk on one server. In this regard adding new IO servers increases the total bandwidth of the FS, but does not affect the time needed for the user to process his data. In addition, for small files, meta-data server communication required for operations such as creating, opening, closing, and deletion takes significant amount of time relative to the actual data transfer. Since in most cases individual jobs access the files one after another, performance of user's workflow can be limited by the latency of communication with the servers, and not by the theoretical peak bandwidth of the storage solution. Finally, from the system administrator's perspective, applications that work with large numbers of files impose a significant load on the servers, affect the applications of other users, and decrease the performance of the entire system. In practice, when working with small files on general-purpose, global network File Systems frequent synchronous communication with the servers imposes a large overhead, which can result in 2-3 orders of magnitude slowdown compared to the peak achievable bandwidth of the storage solution. Interestingly, it is common that the expensive parallel network disks perform much worse than standard SATA disks available on common desktops. As an illustrative example, uncompressing the Linux kernel source (~60 thousand files) on a modern SATA HDD (70MB/s bandwidth) takes less than 10s, while the same operation can take 160s on an HPC-grade parallel FS running over Infiniband at the University of Oslo (bandwidth).

Workflows that deal with small files are a significant practical problem. Some fields of science, e.g., bioinformatics, are known for producing tens of millions of files, e.g., during DNA sequencing. First, the data must be first copied over the network. The files can be archived for transfer, but in the end, they have to be uncompressed and staged on the parallel FS for processing. Finally, the compute jobs process all the individual files. In this type of workflows the file access overhead is much larger than the computation time. One solution is to convert the file format, or to store the individual files inside an HDF5 dataset. This approach requires changes on the application side, and hence it cannot be implemented in a generic way on the HPC facility level. Another challenging case arises from the fact that parallel FS is often used to provide the entire software and development stacks to the compute nodes. Numerous software packages come with many files (e.g. Intel development tools - 25k files, MATLAB - 330k files), and many of those files are accessed during runtime, e.g., as dynamically linked libraries. On Unix systems, dynamic libraries are linked at application startup by searching through a list of directories defined system-wide, or appended to the LD_LIBRARY_PATH environment variable. Applications dependent on many external libraries have a large LD_LIBRARY_PATH list, and at startup need to search the library paths multiple times. This slows down job execution and imposes heavy load on the IO servers, especially for jobs that run many executables in a pipelined fashion. Since most of the software stack is read-only from the perspective of the running application, the complex and expensive machinery of a general-purpose parallel file system is not required. This fact has been previously recognized by the developers of CernVM-FS - a POSIX read-only file system designed to distribute software in parallel computing environments. It is used to host millions of files and directories, and provides optimized access through aggressive caching - something a general purpose parallel file system cannot do if parallel read/write capabilities across thousands of compute nodes are required.

In this work, another approach is used to improve the performance of workflows dealing with small file. The files are saved inside a single large file containing a disk image, similarly to an archive. When needed, the image is mounted through the Unix loop-back device, and the contents of the image are available to the user in the form of a usual directory tree. Since mounting of disks under Unix often requires super-user privileges, security concerns and possible ways to address them are considered. A complete Python implementation of image creation, mounting, and unmounting framework is presented. A seamless integration into a SLURM-managed HPC environment is discussed on an example of read-only software modules created by administrators, and user-created disk images with read-only application input data. Finally, results of performance benchmarks carried out on the Abel supercomputer facility in Oslo, Norway, are shown.

2. Disk images

Creating a disk image on a Linux system can be done using the following commands:

```
truncate image.name -s SIZE_IN_BYTES
mkfs -t ext4 -T small image.name
```

After executing the above, file image.name will contain a disk image, which can then be mounted:

```
sudo mount image.name /path/to/mount/point
```

At this point files written to `/path/to/mount/point` will be stored inside the `image.name` file.

On systems that support it, an empty disk image file is a sparse file, meaning that physical space is not allocated for the storage until the actual data is written. This means that one can efficiently create 'large enough' disk images without needing to worry about excessive space usage.

For this work the EXT4 file system has been chosen because of its popularity, stability, security, and the fact that the size of an EXT4 partition can be extended. Any other file system could be used instead. Since this solution is intended for small files, the EXT4 disk image is created with the '-T small' flag, which optimizes the file system for this usage scenario by decreasing the block size, and decreasing the inode ratio.

The file containing the disk image itself is stored on the parallel file system and as such continues to be available on all compute nodes. However, once mounted on a compute node all the meta-data operations are performed locally rather than on the meta-data servers. In addition, accessing data on the IO servers is equivalent to accessing a single file, which also removes communication overhead.

The disk image approach has some limitations. Since the IO operations on disk images are not in any way synchronized between multiple compute nodes (the local kernel assumes it has exclusive access to the disk), the images can only be used in parallel in read-only mode. When read-write access is required, to assure data integrity each image can only be mounted by a single compute node. Considering the above the disk images provide a good solution to host the (read-only) software stack, and in most cases the application input data. The approach is also suitable for staging of the data into the HPC facility, which is usually done on a single node. For application output the images are limited to software that runs on a single compute node.

One limitation is that the image size needs to be supplied at creation time. This is not a major concern in the HPC setting: software size and input data size are known, and output data size can usually be accurately estimated by the users. Moreover, thanks to sparse storage users can allocate larger images than necessary without wasting space. Finally, EXT4-based images can be extended after creation using `resize2fs`.

3. Security

Using the mount command requires super-user access for several reasons:

- Mounting a file system at some location shadows the original contents of that location. If a user was allowed to mount at arbitrary locations, he could shadow the `/etc` directory and provide his own password and shadow files, effectively gaining super-user access.
- Mounted file systems can contain harmful files that allow a malicious user to gain root access, e.g., suid root programs, or device files.
- A malicious user could exploit bugs in filesystem drives by mounting specially crafted disk images.

The last issue is not a real concern on modern systems since users are usually allowed to mount removable devices, such as USB sticks. The most common file systems (such as EXT4) have been thoroughly tested and are widely considered to be safe.

In the presented solution, the two first security concerns are addressed by allowing the users sudo access to only two python scripts `mount_image.py` and `umount_image.py` that mount and unmount the images, respectively. On Linux systems, the following lines need to be added to a file in `/etc/sudoers.d`

```
ALL        ALL=(ALL) NOPASSWD:      /path/to/mount_image.py
ALL        ALL=(ALL) NOPASSWD:      /path/to/umount_image.py
```

Sudo is a relatively safe way to provide users with temporary super-user privileges: it sanitizes the environment and makes sure that malicious libraries are not used instead of system libraries. Effectively, only the intended code is executed with escalated privileges. In the considered case security of the solution depends on the correct implementation of mount/unmount scripts. This is discussed in detail later in this document. In short, the mount point is validated, and the mount operation is performed with `nosuid` and `nodev` flags. Similar methods to address the first two concerns are used in many services that allow users to mount removable media (udisks, pmount, usbmount).

Alternatively, instead of using sudo the images could be mounted from user-space using the FUSE framework. This approach may become useful in the future, but at this point the number of FUSE file system drivers is limited mostly to network file systems (e.g., sshfs, fspfs). Stable EXT4 drivers with production-grade support are not available.

4. Implementation

The image creation and mounting framework is implemented in Python2. It distinguishes between two different types of images:

- software module images store the read-only software stack. They are created by system administrators and reside at a well-known trusted location (see below). The framework only allows read-only mounting of those images, and mounts them over the directory where the original software module resides. This facilitates seamless integration with the module infrastructure, and provides a fail-safety mechanism in case of mount failure as well as a simple way to turn the framework on and off.
- user images can be created by unprivileged users. They can be placed at any location on the filesystem and can be mounted in read-write, or read-only. The framework only allows mounting of such images under a well-known location (see below)

The following environment variables define the required paths that are used internally in the framework for storing and mounting of images:

- `SI_IMAGE_PATH:=/cluster/software/IMAGES/`
  points to the directory, where software module images are stored
- `SI_MOUNT_PATH:=/cluster/software/VERSIONS/`
  points to the directory, where software module images can be mounted
- `SI_USR_MOUNT_PATH:=/var/run/user_images/`
  points to the directory, under which the user images can be mounted

The framework comprises of several utilities summarized below, and discussed in detail in the following sections:

• `create_software_image.py`	create a software module image (administrator only)
• `create_user_image.py`	create a user disk image
• `mount_image.py`	mount a disk image
• `umount_image.py`	unmount a disk image
• `module_load`	bash script that integrates images into the module framework
• `umount_all_images.py`	unmounts all disk images on a compute node (administrator only)
• `list_images.py`	list images mounted and loop devices used on a compute node
• `cleanup_images.py` (administrator only)	unmount unused images and free loop devices

4.1. Create images

Creation of software images is performed using `create_software_image.py`:

```
usage: create_software_image.py [-h] [-f] [--oversize OVERSIZE]
                                [--workdir WORKDIR]
                                <module_name>/<module_version>

Package and deploy a software module.

positional arguments:
  <module_name>/<module_version>
                        Name of the module to deploy (installed software module
                    must reside in $SI_MOUNT_PATH/<module_name>/<module_version>)

optional arguments:
  -h, --help            show this help message and exit
  -f, --overwrite       Overwrite existing image file [default: False].
  --oversize OVERSIZE   Image size = oversize * space in bytes required for
                        the files [default: 1.3]
  --workdir WORKDIR     Temporary location where to create the image. A
                        finished imaged will be moved to
                         $SI_IMAGE_PATH/<module_name>/<module_version>.ext4
                        [default: /cluster/tmp]
```

The script requires administrator rights because it calls `/bin/mount` directly. It is assumed that the original software module is installed in `$SI_MOUNT_PATH/<module_name>/<module_version>` located on a network filesystem, and is available to all compute nodes. The script calculates the space required for the files, creates a sparse disk image a bit larger than the calculated space (by OVERSIZE factor), mounts the disk image in RW mode using `/bin/mount` and copies the files using `rsync`. The temporary image is created in a working directory specified as program argument. After successful packaging the image is moved to `$SI_IMAGE_PATH /<module_name>/<module_version>.ext4`, and is ready to use.

Non-privileged users can create disk images at arbitrary locations they have permissions to write to, except for the location of the software module images. This is done using `create_user_image.py`:

```
usage: create_user_image.py [-h] [-f] [-s SIZE] [--oversize OVERSIZE]
                            image_name [src]

Create a disk image, optionally populate it with data.

positional arguments:
  image_name            Name of the disk image file, with path.
  src                   Directory, contents of which to copy into the image.

optional arguments:
  -h, --help            show this help message and exit
  -f, --overwrite       Overwrite existing image file [default: False].
  -s SIZE, --size SIZE  Size of the disk image (in MB), if src is not given.
  --oversize OVERSIZE   Image size = oversize * space in bytes required for
                        the files in src [default: 1.3]
```

In contrast to `create_software_image.py`, this script uses `mount_image.py` instead of `/bin/mount`. Hence, it does not require administrator rights, but only `sudo` access to the `mount_image.py` script. When source directory is provided as argument, the script temporarily mounts the image under $SI_USR_MOUNT_PATH /<username>/temp, and copies the source data into the image using rsync.

4.2. Mount/unmount images

Disk images can be mounted by non-privileged users using sudo mount_image.py:

```
usage: mount_image.py [-h] [--job_id JOB_ID] [--rw] image_name [mount_point]

Mount a disk image.

positional arguments:
  image_name       Name of the software module, or path to disk image file.
  mount_point      Mount point under $SI_USR_MOUNT_PATH/$USER

optional arguments:
  -h, --help       show this help message and exit
  --job_id JOB_ID  job identifier, for use with, e.g., SLURM [default
                   NOJOBID].
  --rw             try to mount image in read-write mode. Only one compute node can
                   do that at a time, and the framework will refuse to mount if
                   other mounts have already been performed.
```

When mount point is specified it is assumed that a user image is being mounted, and image name gives a full path to the image file. In this case --rw flag can be supplied.

When mount point is not specified it is assumed that a software module image is being mounted. The --rw flag is not accepted in this case, and image name must be of the form <module_name>/<module_version>. There can be only one path separator '/' in the module name. The corresponding image file must exist under $SI_IMAGE_PATH/<module_name>/<module_version>.ext4. The mount point is implicitly set to $SI_MOUNT_PATH/<module_name>/<module_version>. Recall that this directory resides on a network file system that contains the original (unimaged) software. A successful mount shadows the original contents of that directory and transparently provides the files from inside the disk image. If mount_image.py fails for some reason (e.g., lack of free loop devices) the original software will continue to be available to the user on the parallel file system.

The job_id parameter is used to integrate the framework with job scheduling systems, such as SLURM. It allows per-job tracking and automatic image unmounting at the end of the job.

For security reasons mount_image.py has the following limitations

- mount point cannot be specified when mounting software module images: those are implicitly mounted at $SI_MOUNT_PATH/<module_name>/<module_version>. Moreover, images from anywhere under $SI_IMAGE_PATH/* cannot be mounted using explicit path, but only through a call to mount_image.py <module_name>/<module_version>
- software module images cannot be mounted in RW mode
- user images can only be mounted at mount points located under $SI_USR_MOUNT_PATH/<username>/*
- when job_id is specified, user must be the owner of the job to mount/unmount the images
- image files and mount point directories cannot be symbolic links
- specified mount point directory cannot already be a mount point for a different image, or disk

Disk images can be unmounted by non-privileged users using `sudo umount_image.py`:

```
usage: umount_image.py [-h] [--job_id JOB_ID] image_name [mount_point]

Unmount a disk image.

positional arguments:
  image_name       Name of the software module, or path to the mounted disk
                   image file.
  mount_point      Mount point under $SI_USR_MOUNT_PATH/$USER

optional arguments:
  -h, --help       show this help message and exit
  --job_id JOB_ID  SLURM job id [default NOJOBID].
```

Image name and mount point arguments must be identical to those of the corresponding mount command. The security checks performed are also identical to those in the `mount_image.py`.

4.3. Multi-user, multi-node environment

The framework is designed to work in a multi-user, multi-compute node environment common to modern HPC facilities. It keeps track of image usage to avoid duplication of mounts, and to assure save RW access where needed. A software module image is only mounted once per compute node. If a user requests mounting of a software module that is already mounted, the existing mount is used, and the image usage information is recorded. Similarly, `umount_image.py` first updates the image usage information and verifies that the image being unmounted is not used by others. If it is used, the image remains mounted.

On each compute node information about mounted images is stored in `/var/lock/software_images/<job_id>.modules` (if `job_id` argument was used when calling `mount_image.py`), or in `/var/lock/software_images/<username>.modules` (if `job_id` was not used and the module was loaded interactively). Each line in those files contains the path to the used image, e.g.,

```
$SI_IMAGE_PATH/<module_name>/<module_version>.ext4
```

Since image usage information may be accessed by many jobs/users, before any update is made to those files an exclusive lock on `/var/lock/software_images/lockfile` is obtained. This assures that only one process accesses, or updates information in that file at a time.

For every image the framework also keeps track of all compute nodes that mount the image in read-only mode by appending the client hostname to the `<imagename>.lock` file. When n nodes mount a given image, the file contains n lines:

```
<compute node name 1>
<compute node name 2>
...
<compute node name n>
```

This is not strictly necessary when only mounting images in read-only mode, but it is required for user images that need to be mounted in read-write mode. Since only one compute node can safely mount an image RW, before `mount_image.py` performs the mount it verifies that the corresponding lock file is empty, i.e., no other compute nodes use it. When an image is successfully mounted RW the `<imagename>.lock` file is updated with the following single line:

```
rw <compute node name>
```

`mount_image.py` will refuse to mount an image when it finds that first three characters in the accompanying lock file are "rw " (space at the end).

Before any update is made to `<imagename>.lock` the framework first obtains an exclusive lock on that file. Since the lock file resides in the same directory as the image file - on the network file system - the FS must support the flock command.

4.4. Integration with module framework

The Modules package and the shell command "module" provide means to dynamically modify user's shell environment. Modules are commonly used on HPC systems to allow users to conveniently choose between different versions of various installed software packages. For example, users may need to compile their code with different versions of a compiler suite, or use various versions of MPI or BLAS libraries. In order to load a module the users issue the following command

```
module load <module_name>/<module_version>
```

For example, to load a specific installed version of the GNU compiler:

```
module load gcc/6.1.0
```

Modules are defined in module files, and the definition mostly consist of various environment variables (e.g, `PATH`, `LD_LIBRARY_PATH`, etc.). Modules can also have dependencies and require loading other modules.

The purpose of integrating the image mounting framework into modules is to automatically mount and unmount images whenever a user issues module (un)load. To achieve this it is enough to add the following line at the end of the module files:

```
system /path/to/module_load $appname/$appversion \$\{SLURM_JOB_ID=NOJOBID\}
$action | logger -t software_images
```

where `appname` denotes the name of the application, and `appversion` - it's version. In the above line, the `module_load` script executes `mount_image.py`, or `umount_image.py`, depending on the module action. Hence, calling module load will mount the appropriate software image, while calling module unload (or module purge) will unmount the image. Module dependencies are handled transparently. Failure to mount an image will not result in any observable error: recall that it is required that the images are mounted over the directory that contains the original software. Consequently, mounting failure is transparent to the users in the sense that the data can be still accessed, but the runtime properties, e.g., file access time, might be different

4.5. Integration with SLURM

Most compute jobs running on modern clusters are parallelized and span multiple compute nodes. Allocation and setup of job execution environment lies with the resource allocation systems. In the case of SLURM jobs are defined by shell scripts, which setup the execution environment, stage the data, and run the application. In the context of running parallel jobs with SLURM it is important to assure that:

- images that should be mounted by the loaded modules are mounted automatically on all participating compute nodes
- images are automatically unmounted on all compute nodes after the job has finished

When SLURM starts a job it executes the job script on one of the compute nodes. Hence, modules are loaded into the environment of that job script and the required disk images are mounted on that one compute node. When starting applications that span multiple compute nodes SLURM preserves the execution environment (shell environment variables) on all nodes. However, the disk images are not automatically mounted. This can be addressed by the SLURM task prolog, which is executed once on each allocated compute node. An example task prolog script that mounts the needed module images on compute nodes is

```
## Mount software images:
modules=`module list --terse 2>&1 | grep -v "Currently Loaded"`
for m in $modules; do
    if [[ ! -e /cluster/etc/modulefiles/$m ]]; then
        logger -t software_images_prolog $SLURM_JOB_ID $m: unknown module
        continue
    fi
    logger -t software_images_prolog $SLURM_JOB_ID `hostname` loading image $m
    /cluster/bin/module_load $m $SLURM_JOB_ID load 2>&1 | logger -t software_images
done
```

Note that paths need to be adjusted for specific deployment environments. Since the environment is preserved on all compute nodes, module list will return the list of currently loaded modules. The images can be then mounted explicitly in a loop executing the `module_load` script.

Similarly, at job completion SLURM can be configured to run an epilog script on each compute node. It can be used to run `unmount_all_images.py`:

```
usage: umount_all_images.py [-h] [--job_id JOB_ID] [--cleanup] [--kill]

Unmount all mounted images, or unmount images loaded by a certain job.

optional arguments:
  -h, --help       show this help message and exit
  --job_id JOB_ID  job identifier [default=ALL].
  --cleanup        detatch used loop devices that are not mounted as images.
  --kill           kill processes that use the unmounted loop devices.
                   Requires --cleanup.
```

Recall that for the purpose of integration with a queuing system the mount/unmount scripts accept the `--job_id` parameter. Information about images mounted by individual jobs running on a compute node is stored in `/var/lock/software_images/<job_id>.modules`. This way `umount_all_images.py` knows which images have been mounted by individual jobs, and can automatically unmount them in the SLURM epilog script, e.g.

```
## Umount software images:
logger -t software_images_epilog `hostname` unmounting all images for job
$SLURM_JOB_ID
umount_all_images --job_id $SLURM_JOB_ID 2>&1 | logger -t software_images
```

4.6. Monitoring and cleanup

There are cases when mounting and unmounting of images fails and puts the image database into an inconsistent state. For example, `umount_all_images.py` can fail when user's application has not finished and keeps holding some files open on the mounted share. In this and similar cases the mount information will be updated, but the loop device will not be freed. To detect such cases the framework provides some utilities that can be used by administrators to check system health.

```
usage: list_images.py [-h] [--job_id JOB_ID] [--unreported]

List all mounted images, or list images loaded by a certain job.

optional arguments:
  -h, --help       show this help message and exit
  --job_id JOB_ID  job identifier
  --unreported     only show mounted images that are not reported in
                   /var/lock/software_images/*modules
```

Used without arguments, this tool prints all loop devices used by the framework, and all mounted images. With the `--job_id` argument it prints the image usage of a given job or user. When using the `--unreported` argument, the tool prints loop devices with attached images that do not have a corresponding mount, and 'forgotten' mounted images that are not mentioned in any of the `/var/lock/software_images/*modules` files, e.g.,

```
# list_images.py --unreported
/cluster/software/IMAGES/matlab/R2016a.ext4 is mounted but not used.
blocked loop device: /dev/loop2 - image (/cluster/software/IMAGES/gcc/6.1.0.ext4)

To fix the inconsistencies the administrator can use the cleanup_images.py script:

usage: cleanup_images.py [-h] [--cleanup] [--kill] [--verbosity VERBOSITY]

Cleanup loop-back devices.

optional arguments:
  -h, --help            show this help message and exit
  --cleanup             Detatch used loop devices that are not mounted as
                        images.
  --kill                Kill processes that use the unmounted loop devices.
                        Requires --cleanup.
  --verbosity VERBOSITY
                        Print some extra information
```

By default the script will unmount all images that do not have the corresponding mount information in `/var/lock/software_modules/*modules`. With the `--cleanup` option the script additionally frees the unused but attached loop devices. If some processes are still using that loop device, the administrator can force killing of those processes using the `--kill` argument. For convenience, the `--cleanup` and `--kill` options can also be passed to the `umount_all_images.py` script.

5. Performance analysis

For the purpose of performance tests, a single compute node of the Abel cluster in Oslo was reserved. The system consists of dual socket nodes running Intel Xeon E5-2670 CPUs, 64 GB of DDR3 memory at 1600 MHz, a 250 GiB Western Digital WD2503ABYX drive, and a Mellanox ConnectX-3 Infiniband card. The scratch and home directories are served by the BeegFS network file system configured to use RDMA over Infiniband for IO data transfers. The BeegFS server-side is running two meta-data servers and sixteen IO servers, the total benchmarked bandwidth of the system exceeding 17GB/s. Here, we demonstrate some of the advantages of the disk image approach to storing small files, compared to using the parallel network file system:

- user experiences much better performance when creating, reading, and deleting the files
- user experiences faster application startup for applications that open a lot of files, or load many dynamic libraries
- the pressure on the meta-data and IO servers is lowered (essentially no meta-data access, and fewer data sessions), which results in better over-all system responsiveness and throughput

	HDD	Image+BeegFS	Native BeegFS
`tar xJf linux-4.9.tar.xz`	9	10	175
`find . -type f -exec md5sum {} \;`	37	37	278
`rm -rf linux-4.9`	0.9	0.9	17

Table 1. Time in seconds taken by different operations on 56k small files (Linux kernel source) on local hard drive (HDD), disk image located on a BeegFS share, and natively on BeegFS.

The first advantage can benefit users that process and produce large amounts of small files in their modeling, e.g., in DNA sequencing. Table 1 shows the performance of synthetic benchmarks, which create, write, read, and remove small files. Left-most column gives the tested command. As an example, Linux kernel sources are uncompressed (780MB in 56k files), and MD5 sum of all files is computed. Finally, the directory tree is deleted. Execution time is measured for three different scenarios:

- HDD: kernel source files are stored on the local hard drive
- Image+BeegFS: files are stored in an EXT4 disk image mounted over the loopback device. The disk image itself resides on a BeegFS partition.
- Native BeegFS: files are stored directly on a BeegFS partition.

Table 1 clearly demonstrates the advantage of using disk images. The performance of the tested commands can be up to 20 times better compared to when the files reside on BeegFS directly. Moreover, IOPs speed is identical to that observed for the local hard drive. This indicates that the performance is limited by the (local) Linux kernel EXT4 drivers, and not by the storage device itself. In other words, the overhead of using a network file system has been removed. Note that performance of the parallel FS may vary for different the hardware configuration, but also for different momentary loads on the cluster. The presented tests were performed under workloads typical to the Abel cluster, which usually runs many jobs that are heavy on the IOPs and the meta-data servers. In the Native BeegFS scenario one can observe significantly better, or worse results, depending on the IO ‘weather’ on the cluster. In contrast, in our experience the Image+BeegFS scenario worked quite predictably, and much better.

	HDD	Image+BeegFS	Native BeegFS
matlab -nodisplay	8	9	107
compile OpenMPI	760	788	1228

Table 2 Time in seconds taken by two applications depending on whether they are installed on local hard drive (HDD), disk image located on a BeegFS share, and natively on BeegFS.

Images also benefit users of large applications, such as MATLAB, or applications that are started often and for short periods of time, such as compiler suits (gcc, Intel). Table 2 presents a comparison of application runtime for two such cases. First, we measure the time of starting MATLAB R2016a from the command line mode, without graphical display:

```
  $ time matlab -nodisplay -e quit;

                              < M A T L A B (R) >
                    Copyright 1984-2016 The MathWorks, Inc.
                     R2016a (9.0.0.341360) 64-bit (glnxa64)
                               February 11, 2016


  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.


          Academic License

  >>
```

MATLAB installation consists of roughly 330.000 files – dynamic libraries, java classes, and others. It is difficult to estimate how many files are opened during MALAB startup. However, Table 2 demonstrates that storing the installation inside a disk image can sometimes decrease the startup time 0 times compared to the native BeeGFS installation. Note also that in this case the startup time was almost identical to a HDD installation. Similar significant improvements can be observed for other user workflows, which run complex software, or pipelines of different short programs, for which startup time is critical, e.g., many bioinformatics workflows. As in the previous test, also here the native BeeGFS performance depends very much on the momentary cluster load.

Another test involved compilation of the OpenMPI library version 1.10.2 (~7000 files) using gcc 6.1.0 (~5500 files). The test involved measurement of the compilation process alone:

```
  $ cd /tmp
  $ wget https://www.open-mpi.org/<...>/openmpi-1.10.2.tar.gz
  $ tar xzf openmpi-1.10.2.tar.gz
  $ cd openmpi-1.10.2
  $ ./configure <…>
  $ time make -j 1
```

For simplicity, one make thread was used in the compilation process. In this case the OpenMPI sources always resided on the hard drive, while the gcc installation was located on the local hard drive (HDD), natively on a BeegFS partition, or inside a disk image located on a BeegFS partition. The test demonstrates the impact of the underlying filesystem on the startup time of gcc: assuming all OpenMPI files were compiled, the compiler was executed roughly 7000 times. Compiling individual files usually takes little time, hence the gcc startup time can introduce a significant overhead. Table 2 shows that installing and running gcc from inside a disk image improved the compilation performance almost twice, compared to the test where gcc was stored natively on a BeegFS share. Note that the improvement came solely from the gcc startup time, not from faster access to source files, nor better performance of gcc. Also in this case the performance of local HDD and Image+BeegFS gcc installations was similar.

6. Conclusions

We have addressed the problem of storing and accessing large numbers (hundreds of thousands) of small files (i.e., up to a few KB) on modern parallel network file systems. Handling small files is notoriously problematic in the HPC setting. While state-of-the-art storage solutions provide high and scalable bandwidth through parallel storage servers connected with a high-speed network, accessing small files is sequential and latency-bounded. We have demonstrated that performance of small file access can be much worse than if the files were stored on a local hard drive. The main reason for this bottleneck is the need to communicate with the meta-data servers when creating, opening, closing, and deleting files. In addition, small files are usually stored on a single IO server instead of being striped across many servers. In the end, performance of user's workflow is limited by the latency of communication with the servers, and not by the theoretical peak bandwidth of the storage solution.

We have presented a generic solution for large-scale HPC facilities that improves the performance of workflows that process large numbers of small file. The files are saved inside a single file containing an EXT4 disk image, instead of being directly placed on the parallel network file system. When needed, the image is mounted through the Unix loop-back device, and the contents of the image are available to the user in the form of a usual directory tree. The framework comes with a set of tools for creation, mounting, and unmounting of images, as well as with scripts to automatically integrate it with the SLURM resource manager, and the software modules.

There are several advantages to this approach. It is generic and application independent, and hence can be deployed by the administrators system-wide. In some cases, the users will need to update their job scripts, but the software does not require any changes. Performance improvements when working with small files can reach up to 20 times compared to storing the files directly on the network FS. This results in a better utilization of the resources for particular user jobs, but it also results in less pressure on the meta-data servers, decreased over-all system load, and better performance observed by other users. We have used this solution with very good results to deploy the software stack (toolchains and other cluster-wide software installations) at the University of Oslo.

One disadvantage is that when using the disk images on multiple compute nodes, they must be mounted in read-only mode: writing to an image from multiple compute nodes would result in data corruption. This is enforced by the framework. From the security point of view, mounting of disks under Unix often requires super-user privileges. In order to work, our framework requires sudo access to two python scripts. We have described the security concerns and our approach to address them.

References

[1] J. Heichler, "An introduction to BeeGFS," 2014.

[2] Intel, "Lustre Software Release 2.x - Operations Manual," 2017.

[3] Gluster Inc, "Cloud storage for the modern data center," 2011.

[4] Dell, "OrangeFS Reference Architecture," 2012.

[5] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel and T. Ludwig, "Small-file access in parallel file systems," in *IEEE International Symposium on Parallel&Distributed Processing* , 2009.

[6] Atul, "Improving performance of small files on Lustre," 2010. [Online]. Available: https://blogs.oracle.com/atulvid/entry/improving_performance_of_small_files.

[7] . H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance for HPC Platforms," in *Cray User Group Conference* , 2007.

Acknowledgements

Thanks to Thomas Röblitz for insightful comments and help with the manuscript.

This work was financially supported by the PRACE project funded by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.