MarFS Install Process

This guide is intended for a general non-LANL audience, and is for configuring Multi-Component MarFS for parallel data transfer over NFS. Another guide will be used for setting up MarFS over RDMA.

This guide assumes familiarity with Linux, ZFS, Infiniband, GPFS, and the Linux software build process, though not all of these technologies are necessary for a working MarFS install.

Quick Architecture Overview

MarFS uses separate storage for data and for metadata (MD). Data is stored as erasure-coded objects. MarFS metadata can use any file-system that provides extended-attributes and sparse-files. For scalability, it should also be a distributed file-system. Previously, we have used object-storage systems, accessed with the S3 protocol, but this install guide covers a more-recent implementation called Multi-Component. A Multi-Component object-store is a set of N+E identical file-systems (where N is the number of data-elements, and E the number of erasure-elements, in an erasure-coding). When MarFS is configured for Multi-Component repositories, it will break data "objects" into N pieces, perform the erasure-coding on N data-elements to produce N+E new elements, and will "stripe" those out across the N+E file-systems.

Install architecture

MarFS can be installed in a variety of settings, from a single VM to many racks in a datacenter, but for illustrative purposes we'll discuss a small setup. All components could be scaled up or down, and while there would be minor configuration differences, the general install process will remain the same.

- 4 Storage Nodes comprised of 4 zpools each in RAIDZ3(17+3)
- 2 Metadata nodes, running GPFS.
- 2 File Transfer Agent nodes(FTAs), for batch data movement using pftool
- 1 node for unprivileged user access that will NFS mount the FUSE MarFS mount and use pftool to move data.

While we use GPFS for storing metadata in this install document, any distributed file system that can support sparse data and extended attributes will work. Some tools for garbage collection and quotas currently use GPFS management features, however, so some modifications would need to be made to MarFS to use those features with another metadata filesystem.

It is recommended that a high-speed interconnect such as Infiniband be used for data and metadata transfer between the nodes, alongside an Ethernet network

for SSH access. Infiniband will be required for using future MarFS features, such as RDMA data transfer. We use Mellanox's MOFED for Infiniband.

MarFS has been tested and used on Enterprise Linux distributions, primarily RedHat and RHEL derivatives. Results may vary on other distributions.

This document will use certain MarFS specific terms, most notably in the MarFS configuration file; we'll define those here. A **repo** is where all the object data for a MarFS Filesystems lives, but it's a logical storage unit of MarFS comprised of data and metadata management parameters. A **namespace** in MarFS is a logical partition of the MarFS filesystem with a unique mount point and attributes like permissions, similar to ZFS datasets. Each namespace in MarFS must be associated with a repo, and you can have multiple namespaces per repo. Both repos and namespaces are arbitrarily named.

All commands should be run as root.

Install Process

Bring up all nodes with the latest version of RHEL and ensure network connectivity between them. We can lock down access after the cluster is setup if necessary. Install MOFED on all nodes and ensure Infiniband connectivity.

Prerequisites

FTA and Interactive nodes (for building MarFS and PFTool)

It's useful to have an NFS exported directory where you build the software required to run MarFS. This export can then be mounted on the nodes that need access to these tools, specifically the File-Transfer Agent (FTA) and interactive nodes, and you can avoid needing to compile and make install on each node. This is especially useful on diskless nodes. This guide assumes this setup, and will mount this directory on the nodes that must mount MarFS or run PFTool in /opt/marfs_build. The server this directory is exported from must have the same kernel version as the nodes where it is mounted.

All MarFS software building only occurs on the nodes that will either mount MarFS through FUSE or run PFTool. The storage and Metadata nodes do not need to have /opt/marfs_build mounted, nor do they necessarily need the development tools installed.

Download the code from Github into /opt/marfs_build:

```
git clone https://github.com/mar-file-system/marfs.git
git clone https://github.com/mar-file-system/PA2X.git
```

```
git clone https://github.com/mar-file-system/erasureUtils.git
git clone https://github.com/mar-file-system/aws4c
git clone https://github.com/pftool/pftool.git
git clone https://github.com/01org/isa-l.git
```

MarFS — the core MarFS libraries
PA2X --- an XML parser for parsing the MarFS configuration file
ErasureUtils --- The erasure coding layer used for Multi-Component storage
Pftool --- a tool for parallel data movement
Aws4c --- C library for AWS, used for S3 and RDMA authentication
ISA-L --- Intel's Intelligent Storage Acceleration Library. Multi-Component uses its erasure coding library. Instead of cloning the repository, it may be better to download a tagged release. It must be release 2.16+.

On the machines you will need to build MarFS and PFTool install the development environment:

```
yum install gcc glibc-devel fuse-devel libattr-devel make curl-devel
curl openssl-devel openssl git libxml2-devel yasm libtool openmpi
openmpi-devel
```

Ensure you have yasm 1.2.0 or later(nasm 2.13 will work too) for the ISA-L

Note: You will need specific versions of Autoconf and Automake. Download Autoconf 2.69 and Automake 1.13.4

Ensure OpenMPI's bin dir is in your $PATH. It may also be necessary to add OpenMPI's lib directory to LD_LIBRARY_PATH if it is not added automatically.

Storage Nodes

Install ZFS on these nodes. We've stayed on the 0.6.5.x series. 0.7.x can be used, but we experienced performance problems with the earlier 0.7.x releases and prefer the mature code on 0.6.5.x. DKMS or KMOD ZFS are both ok.

Metadata Nodes and FTA nodes

While any distributed filesystem can theoretically be used to store MarFS metadata, we use GPFS for its parallelism, replication feature, and lifecycle tools. Install GPFS on the dedicated metadata nodes and the FTA nodes(or any nodes that will mount MarFS directly). Create a GPFS cluster using these nodes with the dedicated nodes as quorum nodes. Ensure the cluster is up before continuing.

Setup Paths for Data and Metadata storage

MarFS Multi-Component needs a place to write data and metadata. For metadata, the entire path is arbitrary and decided by your configuration file, but for data, part of the storage path is hard-coded into the MarFS libraries. It follows a

pattern like `$MountPointOnFTAs/$RepoName/pod$p/block$b/cap$c/`. The capacity unit(`cap$c`) corresponds to an individual zpool on a single storage node(`block$b`), within a set of storage nodes(`pod`). There may be multiple pods, in which case objects will be hashed across all of them. This path is only necessary on the FTAs (which will access the zpools via NFS), but for maintenance purposes it's useful to create them on the storage nodes too. As an example this directory on an FTA

```
/marfs_storage/mc/repo3+1/pod0/block0/cap0
```

will be the NFS mount point for the storage dataset on the first zpool of the first storage node. That ZFS dataset might be mounted on the storage node at

```
/zfs/exports/repo3+1/pod0/block0/cap0
```

So, if you're wondering if that means you will need to create a bunch of directories and mount a bunch of NFS shares for MarFS to work, yes it does. There would be similar NFS mount-points for N+E different blocks on each FTA, with corresponding ZFS exports from the storage nodes, to which the mounts refer. `For`-loops really help here. But note that only the `pod$p/block$b/cap$c/` part of the path is immutable; MarFS will look for it in under any arbitrary path you point it to in your config file(which we'll look at later).

Storage Nodes (to hold object-Data)

Our setup is for four storage nodes, with four zpools each, each zpool in a RAIDZ3 configuration, so we'll create four zpools on each storage node. For ease of maintenance and potential scripting, it's recommended to give them sequential numbers. E.g. assuming the first storage node is called sn000:

```
zpool create sn000-pool0 raidz3 $disks
zpool create sn000-pool1 raidz3 $disks
zpool create sn000-pool2 raidz3 $disks
zpool create sn000-pool3 raidz3 $disks
```

And so on, for each Storage node.

Once the zpools are created, modify attributes on them for better MarFS performance. A for loop makes this easy:

```
for i in {0..3}; do zfs set recordsize=1M sn001-pool$i; done
for i in {0..3}; do zfs set mountpoint=none sn001-pool$i; done
for i in {0..3}; do zfs set compression=lz4 sn001-pool$i; done
for i in {0..3}; do zfs set atime=off sn001-pool$i; done
```

It is possible to configure the storage nodes to be diskless. For diskless nodes, create another dataset on the first zpool on each storage node called nfs(or whatever you like) and set its mountpoint to be /zfs (or whatever the top level directory will be for the other dataset mountpoints you will make):

```
zfs create sn003-pool0/nfs
zfs set mountpoint=/zfs sn003-pool0/nfs
```

Make sure this dataset is mounted first, on reboot. This is needed because NFS will stat the underlying mountpoint, which is tmpfs in diskless setups, and get the wrong block size.

Create a dataset named "datastore" (or whatever you want to call the data storage dataset) on each zpool on each storage node:

```
for i in {0..3}; do zfs create sn002-pool$i/datastore; done
```

On each storage node, create a top-level directory under which you will mount and export over NFS all of your ZFS datasets.

```
mkdir –p /zfs/exports
```

Now create the `pod$p/block$b/cap$c` structure. This is where you will mount the datastore datasets you created, and these correspond to your capacity units. First, consider the repo name: it can be anything you want, but make sure it's unique and memorable. In testing, we've often named them after the erasure coding level used for the repo, so that's what we'll use in this document. The repo directory will be underneath the `/zfs/exports` directory you just created.

We are assuming a single *pod* of storage-servers, so your directory structure would look something like this:

```
/zfs/exports/repo3+1/pod0/block0/cap0
/zfs/exports/repo3+1/pod0/block0/cap1
/zfs/exports/repo3+1/pod0/block0/cap2
/zfs/exports/repo3+1/pod0/block0/cap3
```

And on the second storage node:

```
/zfs/exports/repo3+1/pod0/block1/cap0
/zfs/exports/repo3+1/pod0/block1/cap1
/zfs/exports/repo3+1/pod0/block1/cap2
/zfs/exports/repo3+1/pod0/block1/cap3
```

`For` loops are still your friend here:

```
for i in {0..3}; do mkdir -p /zfs/exports/repo3+1/pod0/block3/cap$i
```

Then mount the datasets at the directories you just created:

```
for i in {0..3};
```

```
do zfs set mountpoint=/zfs/exports/repo3+1/pod0/block0/cap$i sn003-
pool$i/datastore;
done
```

To create more repos in the same storage nodes, create another set of datasets on the pools and directory structure/mountpoints. (Not shown.)

Now, edit the /etc/exports file on each to export your datasets:

```
/zfs/exports      *(rw,fsid=0,no_subtree_check,sync,crossmnt)
```

Metadata nodes

With the GPFS cluster setup, the work needed for the metadata nodes is limited.

Create a GPFS filesystem named meta (or whatever you like).

Make a local directory /marfs (for example) on the metadata and FTA nodes.

Mount the GPFS filesystem on all the metadata and FTA nodes under /marfs/meta (for example).

Make the **trash directory** underneath /marfs/meta.

```
mkdir /marfs/meta/mc-trash
```

Create a fileset for the MDFS for each **filesystem namespace** you want. This fileset will contain all the metadata for that namespace.

Link that fileset to /marfs/meta/$fileset.

Underneath the fileset, make a directory named mdfs (for example)

```
mkdir /marfs/meta/$fileset/mdfs
```

Ensure all MDFS directories are readable by everyone. The permissions of the namespace mdfs directory(`/marfs/meta/$fileset/mdfs`) will be reflected in the namespace directories presented by FUSE when we mount MarFS later, so it should be chowned to the particular user and group that will be using the namespace.

There's a file MarFS will expect to see in the mdfs of each namespace it builds. This file is called fsinfo. The MarFS build process will not make it automatically, and MarFS will fail to mount with no explanation if it is not present. You'll need to make it yourself, so underneath each namespace:

```
touch /marfs/meta/$fileset/fsinfo
```

Nodes mounting MarFS (FTAs)

As these are the nodes that are actually running the MarFS code, they are the ones that need the `pod$p/block$b/cap$c` directory structure. These nodes will also be mounting the MDFS from the metadata nodes and the ZFS datasets(capacity units) from the storage nodes.

Since we made the /marfs directory on the FTAs in the previous step, you can make another directory under it for the data. We'll call it *mc*, for Multi-Component.

```
mkdir /marfs/mc
```

Underneath this directory, make a series of paths corresponding to the paths to the repo and capacity units you made on the storage nodes. You'll be mounting those over NFS to these directories. Use a for loop:

```
for b in {0..3}; do
  for c in {0..3}; do
    mkdir -p /marfs/mc/repo3+1/pod0/block$b/cap$c; done; done
```

Then mount all the capacity units of all storage nodes on each FTA. Be sure to mount over Infiniband. For our four storage nodes, we would do:

```
for i in {0..3}; do
  mount -t nfs iboip-sn000:/zfs/exports/repo3+1/pod0/block0/cap$i
/gpfs/mc/repo3+1/pod0/block0/cap$i;
…
for i in {0..3}; do
  mount -t nfs iboip- sn003:/zfs/exports/repo3+1/pod0/block3/cap$i
/gpfs/mc/repo3+1/pod0/block3/cap$i;
```

MarFS uses a "scatter tree", or scatter directories, to spread the load of object, trash, and log files, so we don't have too many files in any single directory. The number of scatter directories needed depends on the size of your filesystem. You'll need to make these yourself, as well. For a relatively small filesystem like ours, 1024 is sufficient. Do this once you've mounted the capacity units on the FTAs:

```
for b in {0..3}; do
  for c in {0..3}; do
    for s in {1..1024}; do
      mkdir -p /marfs/mc/repo3+1/pod0/block$b/cap$c/scatter$s; done;
done; done
```

This is also a good connectivity test before you move on to doing real MarFSing.

Building and Mounting MarFS

Build Environment

As stated in the Prerequisites above, we'll be building under the directory /opt/marfs_build, which will be shared across the FTA and interactive nodes.

This will greatly simplify the build and mount process. Ensure you have all the software downloaded.

An environment file like the one shown below will be necessary for the build process. Note that this file must be "sourced", rather than executed.

```
#!/bin/bash
BASE=/opt/marfs_build/
export MARFS=$BASE/marfs
export
conf=$MARFS/common/configuration/src/marfs_configuration_blueprint.cf
g
export AWS4C=$BASE/aws4c-lanl
export PA2X=$BASE/PA2X
export SRC=$MARFS
export PF=$BASE/pftool
export PFTOOL=$BASE/pftool/install/bin/pftool
export NEBASE=$BASE/erasureUtils
export LIBNE=$BASE/erasureUtils/erasureLib
export PARSE_DIR=$PA2X

export LIBISAL=/opt/marfs_build/isa-l/
export LIBAWS4C=$AWS4C
export MARFS_CFLAGS="-I$LIBAWS4C -I$MARFS/install/include -
I/usr/include/libxml2 -I$LIBNE"
export MARFS_LDFLAGS="-L$LIBAWS4C -L$MARFS/install/lib -L/usr/lib64 -
L$LIBISAL -L$LIBNE -Wl,-R/opt/openmpi-1.6-gnu/lib"

export LDFLAGS=$MARFS_LDFLAGS
export CPPFLAGS=$MARFS_CFLAGS

export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$LIBISAL:$LIBNE"

export MARFSCONFIGRC=/opt/marfs_build/marfs.config/marfs-1.cfg
```

Ensure the paths match your environment. Source this file on each node that will mount MarFS or run PFTool.

You will also need a MarFS XML configuration file(that's the $MARFSCONFIGRC variable). Place it at `/opt/marfs_build/marfs.config/marfs-1.cfg`
Here's a sample that matches what we've been building in this tutorial:

```
<config>

# 1 pod, 4 blocks(storage servers), 4 capacity units(zpools)
# (zpools are configured in a RAIDZ3(17+3), which is not defined here)

<name>marfs-cluster</name>

<version>1.6</version>

<mnt_top>/marfs-mount</mnt_top>
<mdfs_top>/marfs</mdfs_top>

#=================================================================
```

```
#      R E P O S
#=====================================================================

#.............................................................
# Multi-Component + NFS
#.............................................................

<repo>
   <name>mc3+1</name>

   <host>/marfs/mc/repo3+1/pod%d/block%s/cap%d/scatter%d</host>
   <host_offset>0</host_offset>
   <host_count>1</host_count>

   <update_in_place>no</update_in_place>
   <ssl>no</ssl>
   <access_method>SEMI_DIRECT</access_method>
   # <max_get_size>0</max_get_size>
   <chunk_size>1073741824</chunk_size> # 1G

   <max_pack_file_count>-1</max_pack_file_count> # 0=disable, -1=unlimited
   <min_pack_file_count>10</min_pack_file_count>
   <max_pack_file_size>104857600</max_pack_file_size> # = 100 MiB max
   <min_pack_file_size>1</min_pack_file_size>

   <security_method>NONE</security_method>

   <enc_type>NONE</enc_type>
   <comp_type>NONE</comp_type>
   <correct_type>NONE</correct_type>
   <latency>10000</latency>
   # <write_timeout>20</write_timeout>
   # <read_timeout>20</read_timeout>

   <timing_flags> NONE </timing_flags>

   <dal>
     <type>MC</type>
     <opt> <key_val> n : 3               </key_val> </opt>
     <opt> <key_val> e : 1                </key_val> </opt>
     <opt> <key_val> num_pods : 1         </key_val> </opt>
     <opt> <key_val> num_cap : 4          </key_val> </opt>
     <opt> <key_val> scatter_width : 1024 </key_val> </opt>
     <opt> <key_val> degraded_log_dir : /gpfs/marfs/mc-logs/degraded
</key_val> </opt>
   </dal>
</repo>

#=====================================================================
#      N A M E S P A C E S
#=====================================================================

# --- multi-component namespaces ----------------------------------
<namespace>
  <name>marfs-test</name>
  <alias>proxy3</alias>
  <mnt_path>/marfs-test</mnt_path>
  <bperms>RM,WM,RD,WD,TD,UD</bperms>
  <iperms>RM,WM,RD,WD,TD,UD</iperms>
  <iwrite_repo_name>mc3+1</iwrite_repo_name>
  <range>
    <min_size>0</min_size>
    <max_size>-1</max_size>
    <repo_name>mc3+1</repo_name>
  </range>
```

```
  <md_path>/marfs/meta/marfs-test/mdfs</md_path>
  <trash_md_path>/marfs/meta/mc-trash</trash_md_path>
  <fsinfo_path>/marfs/meta/marfs-test/fsinfo</fsinfo_path>
  <quota_space>-1</quota_space>  # -1 = unlimited
  <quota_names>-1</quota_names>  # -1 = unlimited
</namespace>



# --- namespace-root (represents the top-level mount-point directory)
<namespace>
  <name>root</name>
  <alias>proxy1</alias>
  <mnt_path>/</mnt_path>
  <bperms>RM</bperms>
  <iperms>RM</iperms>
  <iwrite_repo_name>mc3+1</iwrite_repo_name>
  <range>
    <min_size>0</min_size>
    <max_size>-1</max_size>
    <repo_name>mc3+1</repo_name>
  </range>
  <md_path>/marfs/meta</md_path>
  <trash_md_path>/should_never_be_used</trash_md_path>
  <fsinfo_path>/marfs/meta/fsinfo</fsinfo_path>
  <quota_space>-1</quota_space>
  <quota_names>-1</quota_names>
</namespace>

</config>
```

Note the `<mnt_top>`. You must make this directory. Namespaces will appear as subdirectories underneath it when you make the MarFS FUSE-mount.

Note also the root namespace. This is the parent of all the other namespaces and is the top level FUSE mountpoint. It must be present in the configuration file and must be at / for MarFS to work properly. No data or metadata is actually stored there.

Ensure all paths in the configuration file match your paths.

Note that in each namespace you can configure access to data and metadata, for both interactive (iperms for MarFS FUSE access) and batch (bperms for PFTool access). The options are RM (read metadata), WM (write metadata), RD (read data), WD (write data), TD (truncate data), and UD (unlink data), These constraints apply in addition to the usual POSIX access-controls based on user/group/other, and read/write/execute, access mode-bits. The point of these constrains is to allow system-admins to restrict access to some namespaces, if desired. For example, if storage servers are being worked on, you might still want to allow users to see metadata, or even to perform deletions (note that MarFS deletions move metadata to the trash, but the corresponding data is not destroyed until the garbage-collector utility is run).

Build ISA-L and erasureUtils

Build ISA-L and install it in /opt/marfs_build:

```
./autogen.sh
./configure --prefix=/opt/marfs_build/isa-l/  \
    --libdir=/opt/marfs_build/isa-l/
make install
```

Change to the erasureUtils directory:

```
autoconf –i
./configure  \
  CFLAGS="-I/opt/marfs_build/isa-l/include"  \
  LDFLAGS="-L/opt/marfs_build/isa-l/"
make install
```

If you miss a step, or are missing a dependency, run `autoreconf –fis` and repeat the build process for erasureUtils.

Build AWS4C

This is installed using the supplied Makefile. In their respective directories:

```
cd $AWS4C
make
```

Build MarFS

Make sure that you have an environment file like the one described above and be sure the paths are correct. Not having your paths correct in the environment is a primary source of problems during the MarFS build process. Also, make sure you have the MDFS and capacity units mounted.

The configuration parameters used below will enable logging to stdout for the make command, and helpful messages will also be written to fuse.log in the MarFS build directory. This may be useful for diagnosing problems. Once you have configured MarFS and mounted it properly, it may be wise to reconfigure and mount with `enable-logging=no`, as MarFS can be very verbose, and the volume of logging output can also degrade performance.

The file /root/.awsAuth is used by the aws4c library, and contains storage passwords that can be read only by *root*. You'll need to have /root/.awsAuth present , populated, and chmodded 600 for MarFS to build, otherwise it will fail with "Error opening config file: No such file or directory" in fuse.log. It consists of one or more lines, of the form `<username>:<storage_user>:<password>`. This file will not be used in a Multi-Component setup, but it's needed by the MarFS library.

```
echo "root:root:6676fdf87688" >> /root/.awsAuth
chmod 600 /root/.awsAuth
```

Change to the /opt/marfs_build/marfs (or wherever you have cloned the MarFS repo)

```
autoconf –i
```

If autoconf fails for some reason, fix any errors and try `autoreconf –fis`

Configure MarFS:

```
./configure --prefix=$PWD/install MARFS_MNT=$mnt_path --enable-
logging=syslog --enable-mc
```

Where $mnt_path is the `<mnt_top>` from your MarFS XML config file. You can also set MARFS_MNT as an environment variable outside of the ./configure command, in which case it would not need to be supplied here.

If configure completes, try to install:

```
make install
```

This step is necessary to build the MarFS library files, which are also used for PFTool's MarFS compatibility. In the course of the build, the PA2X library will also be built, using some of the environment variables defined in the example shown earlier. This parser will be used later, to read the MarFS XML config file.

Finally, try to mount MarFS. This is done from the Makefile:

```
make mnt
```

If all goes according to plan, marfs_fuse will start and give you PID, and you'll have a FUSE mount at $MARFS_MNT. If you don't see a PID, there is something amiss. Because we configured to generate logging to syslog, MarFS will report diagnostics on the FUSE mount attempt to syslog. Consulting the log may reveal any problems (e.g. problems with the MarFS config file, missing directories, etc.). This logging will also be performed from MarFS internals used by `pftool` (described below). Therefore, once you confirm that the FUSE-mount is coming up successfully, it may make sense for performance reasons to reconfigure without `--logging=syslog`, then rebuild, then run '`make mnt`' again. This will improve performance, and reduce the logging burden, for both fuse and pftool.

PFTool Install

PFTool provides fast, parallel MarFS data movement and is preferred over using the FUSE mount.

Ensure you have mpicc in your path and the MarFS libraries are installed. Make sure you still have the MarFS environment file above sourced.

Change to the /opt/marfs_build/pftool directory(or wherever you cloned the PFTool Github repository).

```
./configure --enable-mc --prefix=$PWD/install
```

if you receive an error that looks like “Could not open configuration file (./parse-inc/config-structs.h). Aborting. we have a NULL problem”

You will need to point to the MarFS libraries directly:

```
./configure --enable-mc --prefix=$PWD/install LDFLAGS="-
L/opt/marfs_build/marfs/install"
```

When the pftool binary is installed in /opt/marfs_build/pftool/install/, you can put that directory into your $PATH and test pftool on an FTA using mpirun:

```
mpirun -H <comma separated IPs of nodes> -x MARFSCONFIGRC(or other
envs) -np 4(number of workers you want, minus 3 for overhead ) pftool
-p $src -c $dst -w 0(copy) -r -vv
```

Run ‘pftool -h’ to see more options. (Note: ‘-vv’ provides maximum verbosity, and may be overkill when moving large amounts of data.

Python-based wrapper scripts are included in /opt/marfs_build/pftool/install/, such as `pfcp`. You will need a pftool.cfg for this to work. Place it in /opt/marfs_build/pftool/install/etc/. Here is a sample:

```
[num_procs]
#smaller number (than in {source_dir}/etc/pftool.threaded.cfg), for
mpi ranks
pfls: 15
pfcp: 15
pfcm: 15
min_per_node: 2

[environment]
#set to False for mpi mode
threaded: False

#path to mpirun
mpirun: mpirun

#log to syslog
logging: True

#Enables n-to-1 writing
parallel_dest: True
```

```
#Enable a darshan logging tool
darshanlib: /usr/projects/darshan/sw/toss-x86_64/lib/libdarshan.so

[options]
#1 MB
writesize: 1MB

#10 GB
chunk_at: 10GB

#10 GB
chunksize: 10GB

[active_nodes]

#be sure these aren't nodename.localhost
#specify all: ON to automatically use all nodes
#all: ON
fta001: ON
fta002: ON
```

Change the $hostname: ON section to whatever FTAs you will be using.