mpiFileUtils Documentation

Release 0.11.1

HPC

Oct 26, 2022

Contents

CHAPTER 1

Overview

High-performance computing users generate large datasets using parallel applications that can run with thousands of processes. However, users are often stuck managing those datasets using traditional single-process tools like cp and rm. This mismatch in scale makes it impractical for users to work with their data.

The mpiFileUtils suite solves this problem by offering MPI-based tools for basic tasks like copy, remove, and compare for such datasets, delivering orders of magnitude in performance speedup over their single-process counterparts. Furthermore, the libmfu library packages common functionality to simplify the creation of new tools, and it can even be invoked directly from within HPC applications.

Video Overview: "Scalable Management of HPC Datasets with mpiFileUtils", HPCKP'20.

The figure below, taken from the above presentation, illustrates the potential performance improvement that one can achieve when scaling a tool like dcp to utilize more compute resources.

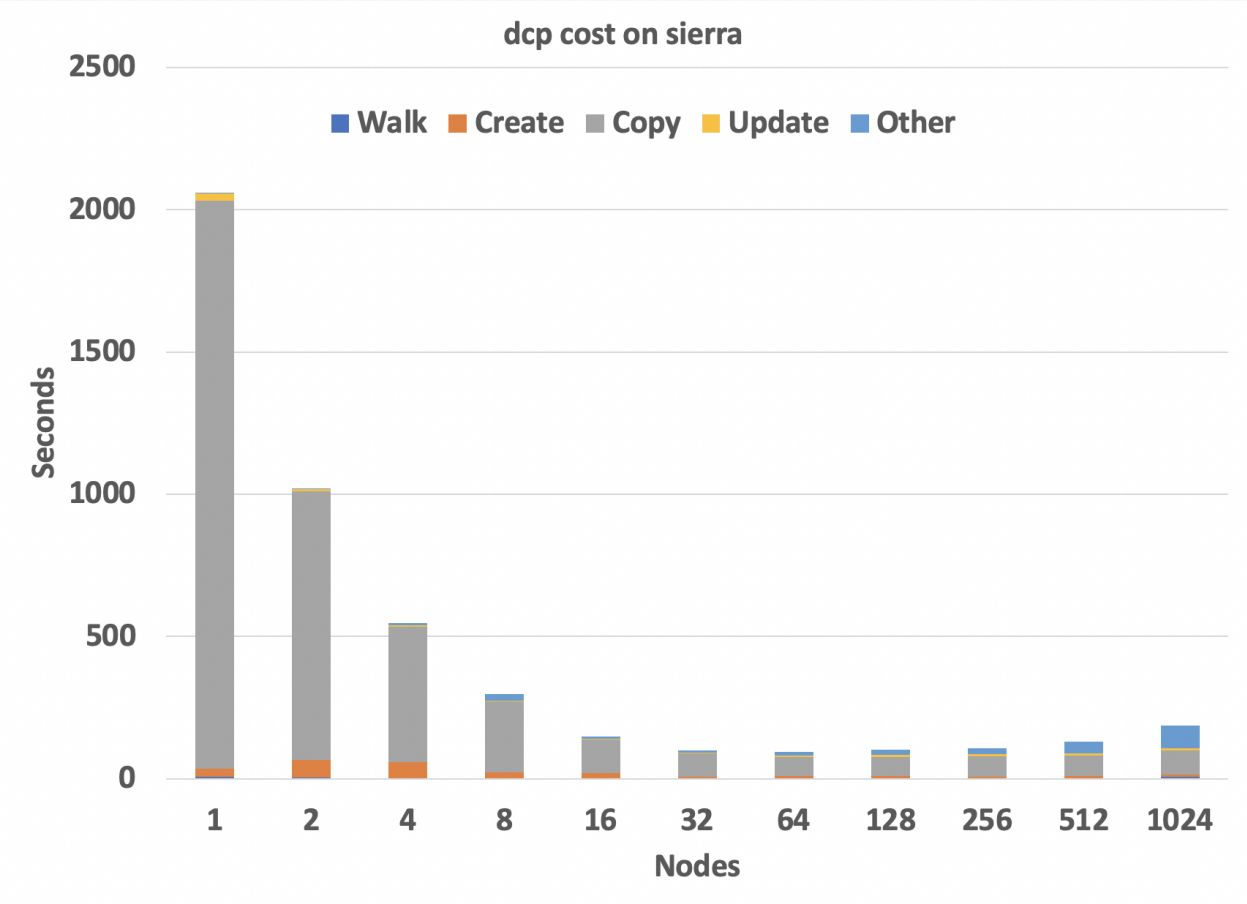

Fig. 1: dcp scaling performance on the Sierra cluster at LLNL using 40 processes/node. Shows the time required to copy a single directory of 200k files totaling 24.4 TiB of data. The minimum time of 93 seconds at 64 nodes is 495x faster than the 12.75 hours taken by the cp command.

CHAPTER 2

User Guide

2.1 Build

mpiFileUtils and its dependencies can be installed with CMake or Spack. Several build variations are described in this section:

- CMake
- Spack
- development build with CMake
- development build with Spack

2.1.1 CMake

mpiFileUtils requires CMake 3.1 or higher. Before running cmake, ensure that the MPI wrapper scripts like mpicc are loaded in your environment.

The simplest mpiFileUtils install for many users is to build from a release package. This packages the source for a specific version of mpiFileUtils along with the corresponding source for several of its dependencies in a single tarball. mpiFileUtils release packages are available as attachments from their respective GitHub Releases page:

https://github.com/hpc/mpifileutils/releases

mpiFileUtils optionally depends on libarchive, version 3.5.1. If new enough, the system install of libarchive may be sufficient, though even newer versions may be incompatible with the required version. To be certain of compatibility, it is recommended that one install libarchive-3.5.1 with commands like the following

```
#!/bin/bash
mkdir install
installdir=`pwd`/install

wget https://github.com/libarchive/libarchive/releases/download/v3.5.1/libarchive-3.5.
↪1.tar.gz
```

(continues on next page)

(continued from previous page)

```
tar -zxf libarchive-3.5.1.tar.gz
cd libarchive-3.5.1
  ./configure --prefix=$installdir
  make install
cd ..
```

To build on PowerPC, one may need to add `--build=powerpc64le-redhat-linux-gnu` to the configure command.

Assuming libarchive has been installed to an *install* directory as shown above, one can then build mpiFileUtils from a release like v0.11.1 with commands like the following:

```
wget https://github.com/hpc/mpifileutils/releases/download/v0.11.1/mpifileutils-v0.11.
↪1.tgz
tar -zxf mpifileutils-v0.11.1.tgz
cd mpifileutils-v0.11.1
  mkdir build
  cd build
    cmake .. \
      -DWITH_LibArchive_PREFIX=../../install \
      -DCMAKE_INSTALL_PREFIX=../../install
    make -j install
  cd ..
cd ..
```

Additional CMake options:

- `-DENABLE_LIBARCHIVE=[ON/OFF]` : use libarchive and build tools requiring libarchive like dtar, defaults to `ON`
- `-DENABLE_XATTRS=[ON/OFF]` : use extended attributes and libattr, defaults to `ON`
- `-DENABLE_LUSTRE=[ON/OFF]` : specialization for Lustre, defaults to `OFF`
- `-DENABLE_GPFS=[ON/OFF]` : specialization for GPFS, defaults to `OFF`
- `-DENABLE_EXPERIMENTAL=[ON/OFF]` : build experimental tools, defaults to `OFF`

2.1.2 DAOS support

To build with DAOS support, first install mpiFileUtils dependencies as mentioned above, and also make sure DAOS is installed. If DAOS is installed under a standard system path then specifying the DAOS path with `-DWITH_DAOS_PREFIX` is unnecessary.

```
cmake ../mpifileutils \
  -DCMAKE_INSTALL_PREFIX=../install \
  -DWITH_DAOS_PREFIX=</path/to/daos/> \
  -DENABLE_DAOS=ON
make -j install
```

Some DAOS-enabled tools require HDF5. To use the *daos-serialize* and *daos-deserialize* tools, HDF5 1.2+ is required. To copy HDF5 containers with *dcp*, HDF5 1.8+ is required, along with the daos-vol.

To build with HDF5 support, add the following flags during CMake. If HDF5 is installed under a standard system path then specifying the HDF5 path with `-DWITH_HDF5_PREFIX` is unnecessary.

```
-DENABLE_HDF5=ON \
-DWITH_HDF5_PREFIX=</path/to/hdf5>
```

2.1.3 Spack

To use Spack, it is recommended that one first create a *packages.yaml* file to list system-provided packages, like MPI. Without doing this, Spack will fetch and install an MPI library that may not work on your system. Make sure that you've set up spack in your shell (see these instructions).

Once Spack has been configured, mpiFileUtils can be installed as:

```
spack install mpifileutils
```

or to enable all features:

```
spack install mpifileutils +lustre +gpfs +experimental
```

2.1.4 Development build with CMake

To make changes to mpiFileUtils, one may wish to build from a clone of the repository. This requires that one installs the mpiFileUtils dependencies separately, which can be done with the following commands:

```
#!/bin/bash
mkdir install
installdir=`pwd`/install

mkdir deps
cd deps
  wget https://github.com/hpc/libcircle/releases/download/v0.3/libcircle-0.3.0.tar.gz
  wget https://github.com/llnl/lwgrp/releases/download/v1.0.4/lwgrp-1.0.4.tar.gz
  wget https://github.com/llnl/dtcmp/releases/download/v1.1.4/dtcmp-1.1.4.tar.gz
  wget https://github.com/libarchive/libarchive/releases/download/v3.5.1/libarchive-3.
↪5.1.tar.gz

  tar -zxf libcircle-0.3.0.tar.gz
  cd libcircle-0.3.0
    ./configure --prefix=$installdir
    make install
  cd ..

  tar -zxf lwgrp-1.0.4.tar.gz
  cd lwgrp-1.0.4
    ./configure --prefix=$installdir
    make install
  cd ..

  tar -zxf dtcmp-1.1.4.tar.gz
  cd dtcmp-1.1.4
    ./configure --prefix=$installdir --with-lwgrp=$installdir
    make install
  cd ..

  tar -zxf libarchive-3.5.1.tar.gz
  cd libarchive-3.5.1
    ./configure --prefix=$installdir
    make install
  cd ..
cd ..
```

One can then clone, build, and install mpiFileUtils:

```
git clone https://github.com/hpc/mpifileutils
mkdir build
cd build
cmake ../mpifileutils \
  -DWITH_DTCMP_PREFIX=../install \
  -DWITH_LibCircle_PREFIX=../install \
  -DWITH_LibArchive_PREFIX=../install \
  -DCMAKE_INSTALL_PREFIX=../install
make -j install
```

The same CMake options as described in earlier sections are available.

2.1.5 Development build with Spack

One can also build from a clone of the mpiFileUtils repository after using Spack to install its dependencies via the *spack.yaml* file that is distributed with mpiFileUtils. From the root directory of mpiFileUtils, run the command *spack find* to determine which packages Spack will install. Next, run *spack concretize* to have Spack perform dependency analysis. Finally, run *spack install* to build the dependencies.

There are two ways to tell CMake about the dependencies. First, you can use *spack load [depname]* to put the installed dependency into your environment paths. Then, at configure time, CMake will automatically detect the location of these dependencies. Thus, the commands to build become:

```
git clone https://github.com/hpc/mpifileutils
mkdir build install
cd mpifileutils
spack install
spack load dtcmp
spack load libcircle
spack load libarchive
cd ../build
cmake ../mpifileutils
```

The other way to use spack is to create a "view" to the installed dependencies. Details on this are coming soon.

2.2 Project Design Principles

The following principles drive design decisions in the project.

2.2.1 Scale

The library and tools should be designed such that running with more processes increases performance, provided there are sufficient data and parallelism available in the underlying file systems. The design of the tool should not impose performance scalability bottlenecks.

2.2.2 Performance

While it is tempting to mimic the interface, behavior, and file formats of familiar tools like cp, rm, and tar, when forced with a choice between compatibility and performance, mpiFileUtils chooses performance. For example, if an archive file format requires serialization that inhibits parallel performance, mpiFileUtils will opt to define a new file format that enables parallelism rather than being constrained to existing formats. Similarly, options in the tool command line

interface may have different semantics from familiar tools in cases where performance is improved. Thus, one should be careful to learn the options of each tool.

2.2.3 Portability

The tools are intended to support common file systems used in HPC centers, like Lustre, GPFS, and NFS. Additionally, methods in the library should be portable and efficient across multiple file systems. Tool and library users can rely on mpiFileUtils to provide portable and performant implementations.

2.2.4 Composability

While the tools do not support chaining with Unix pipes, they do support interoperability through input and output files. One tool may process a dataset and generate an output file that another tool can read as input, e.g., to walk a directory tree with one tool, filter the list of file names with another, and perhaps delete a subset of matching files with a third. Additionally, when logic is deemed to be useful across multiple tools or is anticipated to be useful in future tools or applications, it should be provided in the common library.

2.3 Utilities

The tools in mpiFileUtils are MPI applications. They must be launched as MPI applications, e.g., within a compute allocation on a cluster using mpirun. The tools do not currently checkpoint, so one must be careful that an invocation of the tool has sufficient time to complete before it is killed.

- dbcast - Broadcast a file to each compute node.
- dbz2 - Compress and decompress a file with bz2.
- dchmod - Change owner, group, and permissions on files.
- dcmp - Compare contents between directories or files.
- dcp - Copy files.
- ddup - Find duplicate files.
- dfind - Filter files.
- dreln - Update symlinks to point to a new path.
- drm - Remove files.
- dstripe - Restripe files (Lustre).
- dsync - Synchronize source and destination directories or files.
- dtar - Create and extract tape archive files.
- dwalk - List, sort, and profile files.

2.4 Experimental Utilities

Experimental utilities are under active development. They are not considered to be production worthy, but they are available in the distribution for those who are interested in developing them further or to provide additional examples.

- dgrep - Run grep on files in parallel.

- dparallel - Perform commands in parallel.
- dsh - List and remove files with interactive commands.
- dfilemaker - Generate random files.

2.5 Usage tips

Since the tools are MPI applications, it helps to keep a few things in mind:

- One typically needs to run the tools within a job allocation. The sweet spot for most tools is about 2-4 nodes. One can use more nodes for large datasets, so long as tools scale sufficiently well.
- One must launch the job using the MPI job launcher like mpirun or mpiexec. One should use most CPU cores, though leave a few cores idle on each node for the file system client processes.
- Most tools do not checkpoint their progress. Be sure to request sufficient time in your allocation to allow the job to complete. One may need to start over from the beginning if a tool is interrupted.
- One cannot pipe output of one tool to the input of another. However, the –input and –output file options are good approximations.
- One cannot easily check the return codes of tools. Instead, inspect stdout and stderr output for errors.

2.6 Examples and frequently used commands

If your MPI library supports it, most tools can run as MPI singletons (w/o mpirun, which runs a single-task MPI job). For brevity, the examples in this section are shown as MPI singleton runs. In a real run, one would precede the command shown with an appropriate MPI launch command and options, e.g.,:

```
mpirun -np 128 dwalk /path/to/walk
```

In addition to the man page, each tool provides a help screen for a brief reminder of available options.:

```
dwalk --help
```

The normal output from dwalk shows a summary of item and byte counts. This is useful to determine the number of files and bytes under a path of interest:

```
dwalk /path/to/walk
```

When walking large directory trees, you can write the list to an output file. Then you can read that list back without having to walk the file system again.:

```
dwalk --output list.mfu /path/to/walk
dwalk --input list.mfu
```

The default file format is a binary file intended for use in other tools, not humans, but one can ask for a text-based output:

```
dwalk --text --output list.txt /path/to/walk
```

The text-based output is lossy, and it cannot be read back in to a tool. If you want both, save to binary format first, then read the binary file to convert it to text.:

```
dwalk --output list.mfu /path/to/walk
dwalk --input list.mfu --text --output list.txt
```

dwalk also provides a sort option to order items in the list in various ways, e.g., to order the list by username, then by access time:

```
dwalk --input list.mfu --sort user,atime --output user_atime.mfu
```

To order items from largest to smallest number of bytes:

```
dwalk --input list.mfu --sort '-size' --output big_to_small.mfu
```

dfind can be used to filter items with a string of find-like expressions, e.g., files owned by user1 that are bigger than 100GB:

```
dfind --input list.mfu --user user1 --size +100GB --output user1_over_100GB.mfu
```

dchmod is like chmod and chgrp in one, so one can change uid/gid/mode with a single command:

```
dchmod --group grp1 --mode g+rw /path/to/walk
```

drm is like "rm -rf" but in parallel:

```
drm /path/to/remove
```

dbcast provides an efficient way to broadcast a file to all compute nodes, e.g., upload a tar file of a dataset to an SSD local to each compute node:

```
dbcast /path/to/file.dat /ssd/file.dat
```

dsync is the recommended way to make a copy a large set of files:

```
dsync /path/src /path/dest
```

For large directory trees, the –batch-files option offers a type of checkpoint. It moves files in batches, and if interrupted, a restart picks up from the last completed batch.:

```
dsync --batch-files 100000 /path/src /path/dest
```

The tools can be composed in various ways using the –input and –output options. For example, the following sequence of commands executes a purge operation, which deletes any file that has not been accessed in the past 180 days.:

```
# walk directory to stat all files, record list in file
dwalk --output list.mfu /path/to/walk

# filter list to identify all regular files that were last accessed over 180 days ago
dfind --input list.mfu --type f --atime +180 --output purgelist.mfu

# delete all files in the purge list
drm --input purgelist.mfu
```

2.7 Common Library - libmfu

Functionality that is common to multiple tools is moved to the common library, libmfu. This goal of this library is to make it easy to develop new tools and to provide consistent behavior across tools in the suite. The library can also be

useful to end applications, e.g., to efficiently create or remove a large directory tree in a portable way across different parallel file systems.

2.7.1 libmfu: the mpiFileUtils common library

The mpiFileUtils common library defines data structures and methods on those data structures that makes it easier to develop new tools or for use within HPC applications to provide portable, performant implementations across file systems common in HPC centers.

```
#include "mfu.h"
```

This file includes all other necessary headers.

2.7.2 mfu_flist

The key data structure in libmfu is a distributed file list called mfu_flist. This structure represents a list of files, each with stat-like metadata, that is distributed among a set of MPI ranks.

The library contains functions for creating and operating on these lists. For example, one may create a list by recursively walking an existing directory or by inserting new entries one at a time. Given a list as input, functions exist to create corresponding entries (inodes) on the file system or to delete the list of files. One may filter, sort, and remap entries. One can copy a list of entries from one location to another or compare corresponding entries across two different lists. A file list can be serialized and written to or read from a file.

Each MPI rank "owns" a portion of the list, and there are routines to step through the entries owned by that process. This portion is referred to as the "local" list. Functions exist to get and set properties of the items in the local list, for example to get the path name, type, and size of a file. Functions dealing with the local list can be called by the MPI process independently of other MPI processes.

Other functions operate on the global list in a collective fashion, such as deleting all items in a file list. All processes in the MPI job must invoke these functions simultaneously.

For full details, see mfu_flist.h and refer to its usage in existing tools.

2.7.3 mfu_path

mpiFileUtils represents file paths with the mfu_path structure. Functions are available to manipulate paths to prepend and append entries, to slice paths into pieces, and to compute relative paths.

2.7.4 mfu_param_path

Path names provided by the user on the command line (parameters) are handled through the mfu_param_path structure. Such paths may have to be checked for existence and to determine their type (file or directory). Additionally, the user may specify many such paths through invocations involving shell wildcards, so functions are available to check long lists of paths in parallel.

2.7.5 mfu_io

The mfu_io.h functions provide wrappers for many POSIX-IO functions. This is helpful for checking error codes in a consistent manner and automating retries on failed I/O calls. One should use the wrappers in mfu_io if available, and if not, one should consider adding the missing wrapper.

2.7.6 mfu_util

The mfu_util.h functions provide wrappers for error reporting and memory allocation.