Applying MPI to Manage HPC-scale Datasets

10th Annual MVAPICH User Group (MUG) Meeting

Adam Moody

August 23, 2022

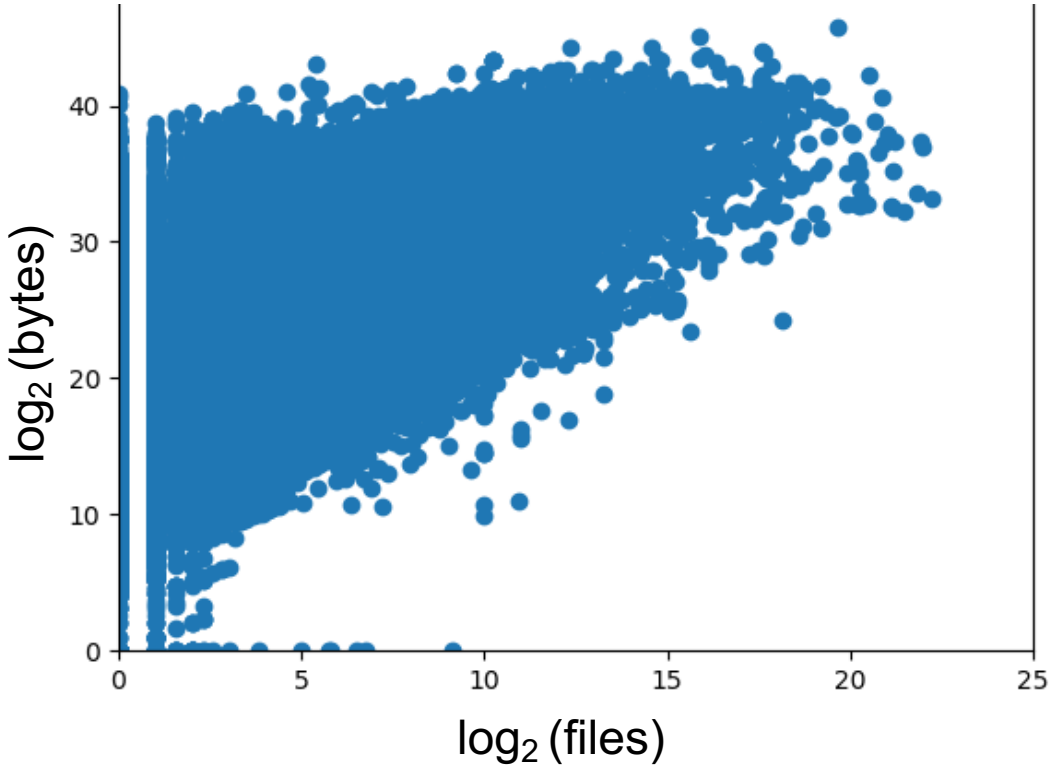
LLNL-PRES-838892

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Dataset file and byte counts increase as users run larger MPI jobs

Scatter plot of dataset
bytes vs files
top 20 Lustre users



Motivating example:
Consider the task of copying a large dataset

Single directory

200,000 files

128 MB per file

24.4 TB total

Goal: `cp -r dir dir2`

Approach 1:
Use cp on a login node

```
>>: date; cp -r dir dir2
Tue Jun 11 10:20:35 PDT 2019
```

... check in after 8 hours ...

```
>>: date; ls -ltr dir2/ | wc
Tue Jun 11 18:20:36 PDT 2019
   114352 1029161 7712420
```

Completed ~114,352 of 200,000 files in 8 hours.

Some math ...
(200,000 / 114,352) * 8 hours ~ **14 hours to finish**

Approach 2:
Use dcp on 8 compute nodes

```
>>: bsub -nnodes 8 -Is /bin/bash

>>: mpirun -np 320 dcp dir dir2

[2019-06-10T18:14:16] Started: Jun-10-2019,18:08:01
[2019-06-10T18:14:16] Completed: Jun-10-2019,18:14:16
[2019-06-10T18:14:16] Seconds: 374.766
[2019-06-10T18:14:16] Items: 200001
[2019-06-10T18:14:16]   Directories: 1
[2019-06-10T18:14:16]   Files: 200000
[2019-06-10T18:14:16]   Links: 0
[2019-06-10T18:14:16] Data: 24.414 TB (26843545600000 bytes)
[2019-06-10T18:14:16] Rate: 66.708 GB/s (26843545600000
bytes in 374.766 seconds)
```

Result:
dcp is slightly faster than cp

cp		dcp
14 hours	>	6 minutes

dcp is ~140x faster

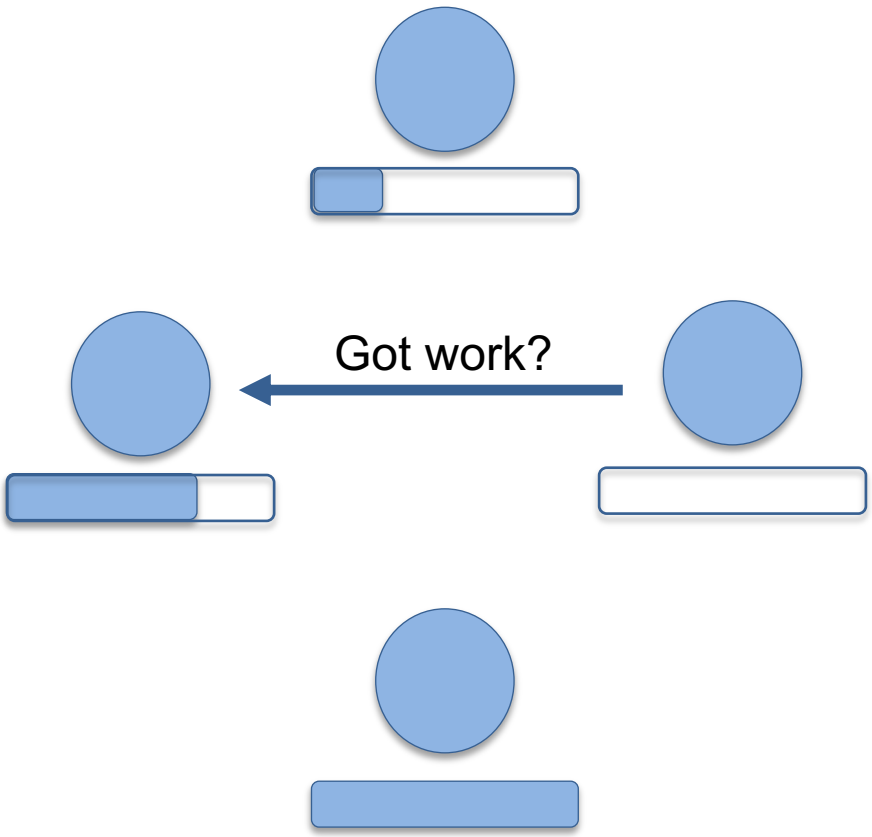
mpiFileUtils enables users to manage datasets with same resources they used to create them

- Large HPC jobs generate datasets while using thousands of compute nodes running tens of thousands of processes

- The problem:
  - Users must often manage those datasets using single-node, single-process tools like cp, chgrp, and rm
  - Users create a dataset with 10,000 cores but then copy it with one.

- mpiFileUtils is designed to:
  - Let users perform common data management tasks with the same HPC resources they used to run their jobs
  - Scale to saturate available compute, network, and file system bandwidth
  - Operate across common HPC file systems like Lustre, GPFS, and NFS

Project origin: mpiFileUtils was created by extending existing tools - libcircle and dcp



- libcircle
  - Each process pulls work items from its local queue.
  - A process can add new work items to its local queue.
  - If a process runs out of work, it requests work from a random process.
  - "On distributed file tree walk of parallel file systems", Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst, SC'12.
  - http://dl.acm.org/citation.cfm?id=2389114
  - https://github.com/hpc/libcircle

- dcp (original)
  - Built all copy functions on libcircle
  - walk, file create, data copy, metadata update
  - https://github.com/hpc/dcp

mpiFileUtils = Library + Tools + File Format

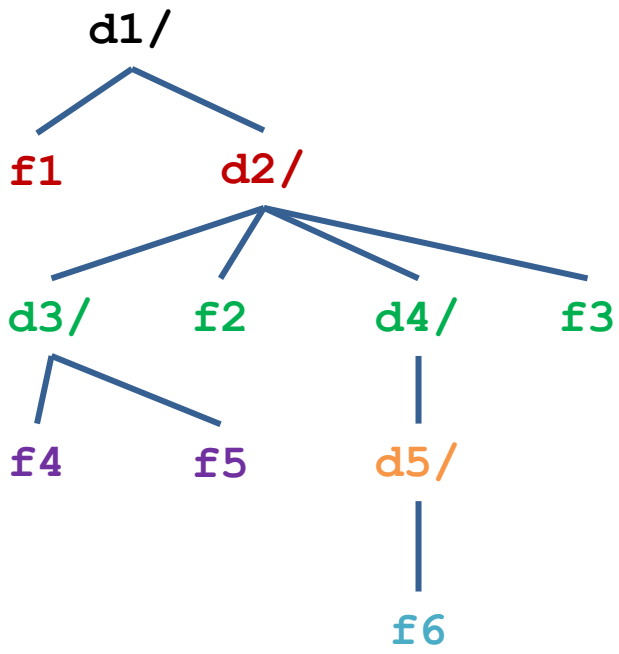
- Library – common data structures and routines
  - Quickly create new tools
  - Apps can even invoke API directly

- Tools – MPI-based versions of cp, rm, etc.
  - Apps create datasets with thousands of processes.
  - Why manage them using a single process?

- File format – common format for interoperability
  - Allows one to compose tools into pipelines
  - Allows third-party software to generate input or process output (e.g., Hopper)

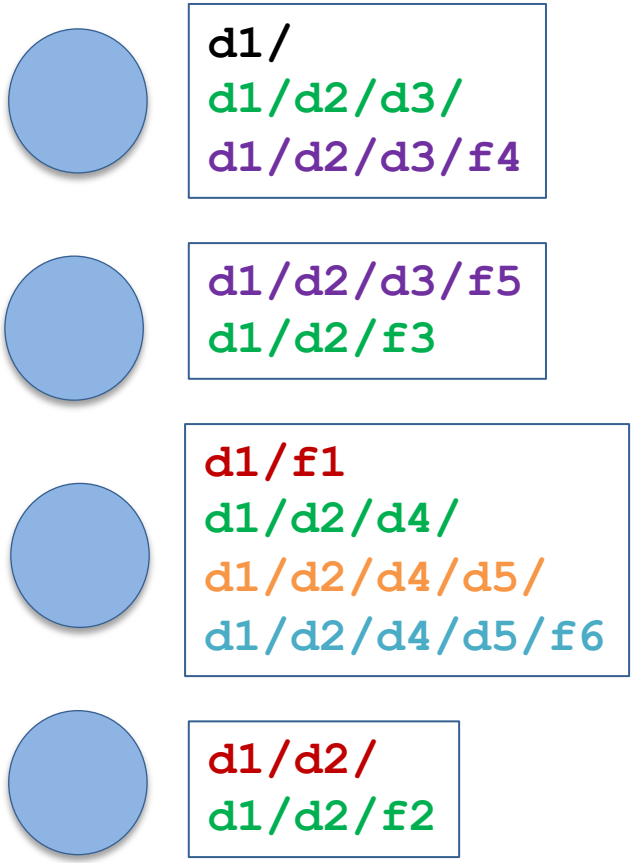
The file list (mfu_flist) is the primary data structure within the libmfu common library

The mfu_flist is a distributed list of stat-like info for each item



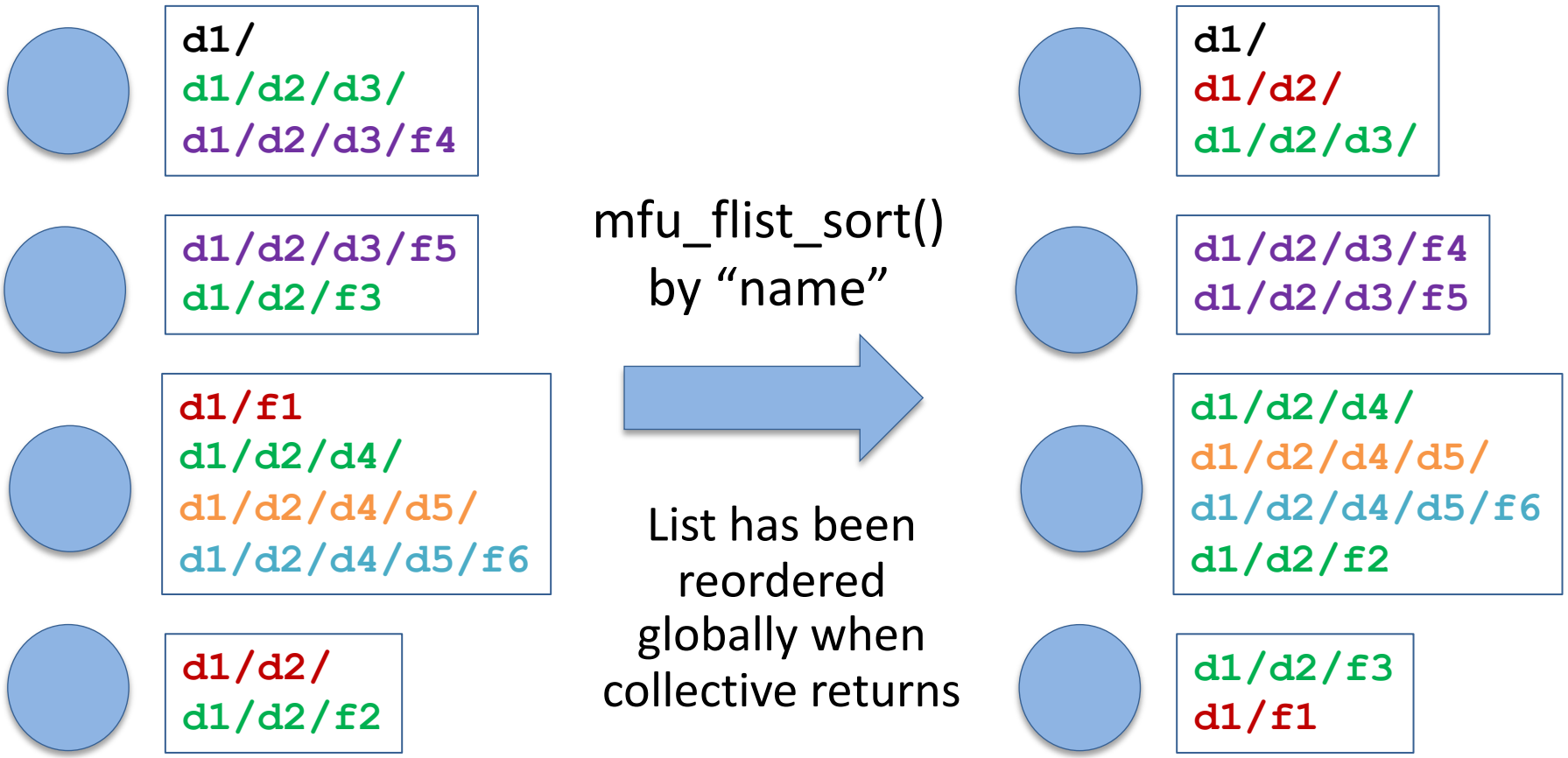
Recursively walk and stat items under **d1/** with libcircle

Each process ends up with stat data for a random subset



Collective mfu_flist operations operate across all processes, for example, to do a global sort

Sort items alphabetically by their full path



mpiFileUtils has a growing set of production and experimental tools (v0.11); 'd' is for distributed

- **dbcast** – broadcast file to compute nodes
- **dbz2** – compress/decompress file with bz2
- **dchmod** – change perms/owner/group
- **dcmp** – compare directories/files
- **dcp** – copy data
- **ddup** – find duplicate files
- **dfilemaker** – generate test files
- **dfind** – filter files
- **dreln** – update symlinks
- **drm** – delete files
- **dstripe** – restripe files (Lustre)
- **dsync** – synchronize directory trees
- **dtar** - create / unpack tar files
- **dwalk** – list, sort, summarize files
- **dgrep** – parallel grep
- **dparallel** – MPI-based parallel
- **dsh** – interactively list, summarize, and remove files

The common file format lets one compose tools, e.g., purge all files last accessed over 180 days ago

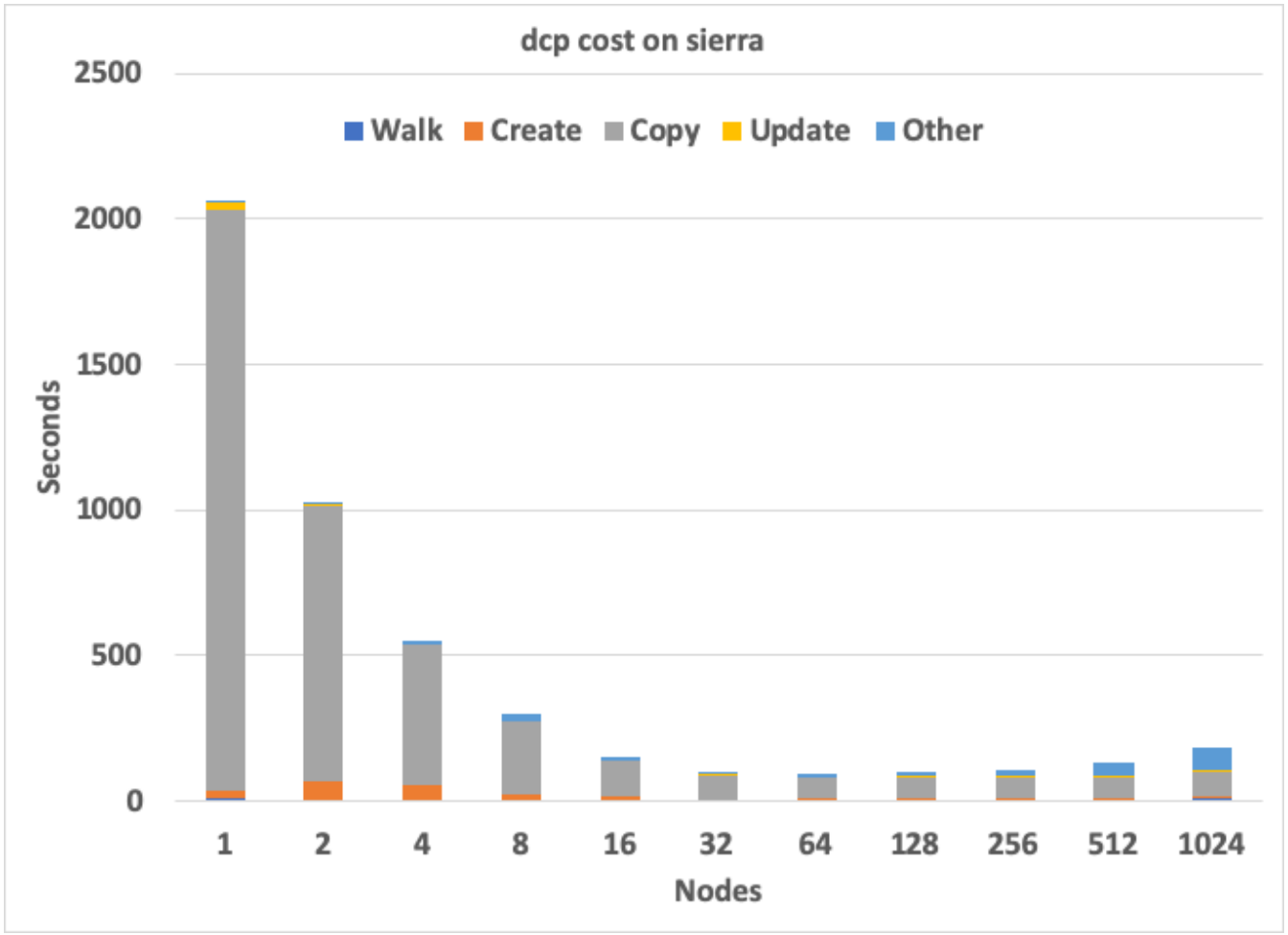
```
# walk directory to stat all files, record list
in file

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
```

Performance at scale

Sierra Scaling Study: dcp, dcmp, dchmod, drm

- Given our example dataset:
  - Single directory
  - 200,000 files
  - 128 MB per file
  - 24.4 TB total
- Do the following:
  1. dcp: Make a copy of the dataset
  2. dchmod: Change group on all files in the copy
  3. drm: Remove the copy
- Scale from 1 node (40 procs) up to 1k nodes (40k procs)

dcp walks source directory, creates destination files, copies data, then updates file metadata



Single directory
200,000 files
128 MB per file
24.4 TB total

40 procs/node

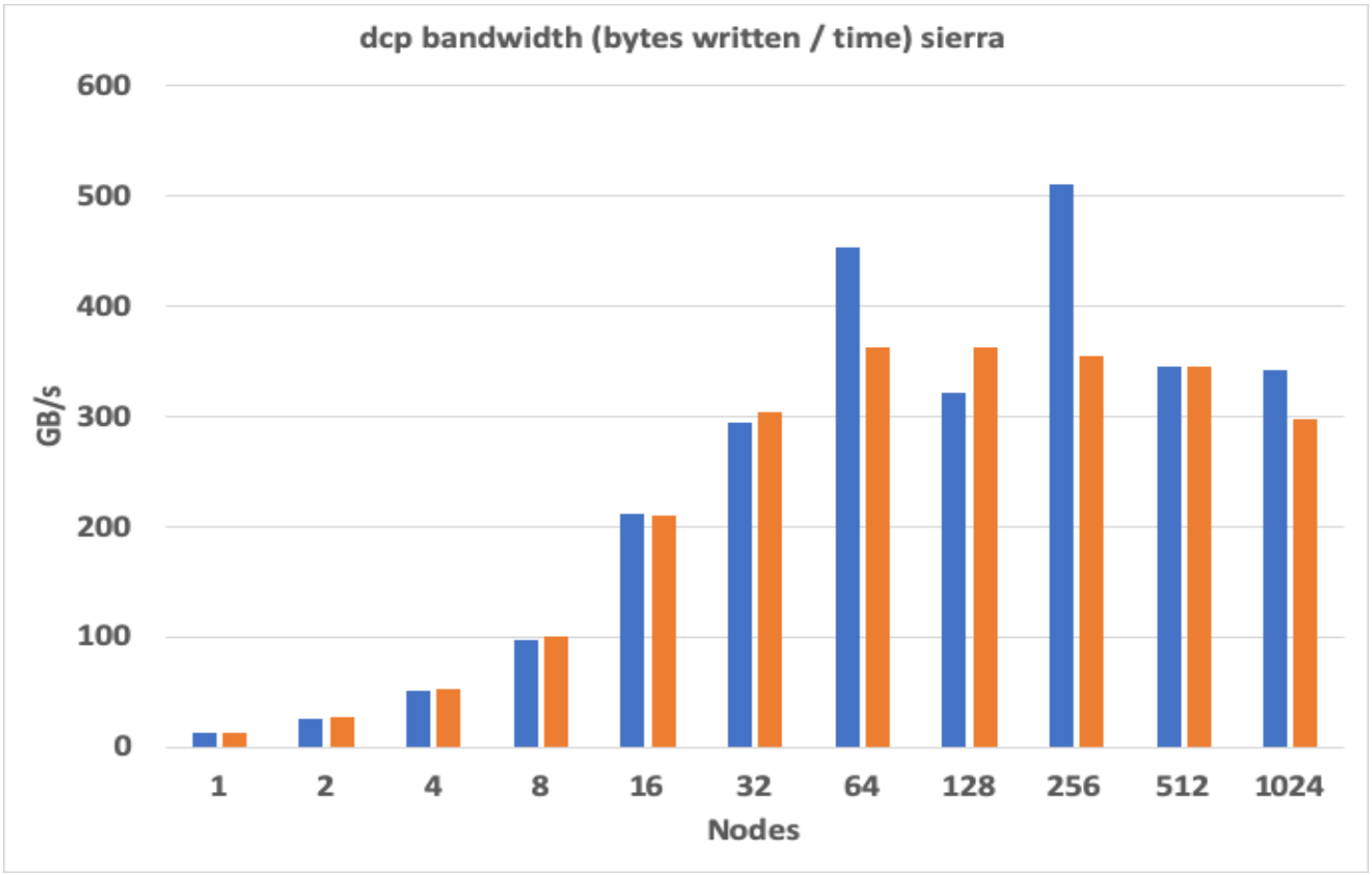
cp time
12 hours, 45 mins

dcp time
93 secs (64 nodes)

495x faster

13 hours → 2 mins

The dcp write bandwidth increases with node count and holds steady at 320 GB/s on Sierra

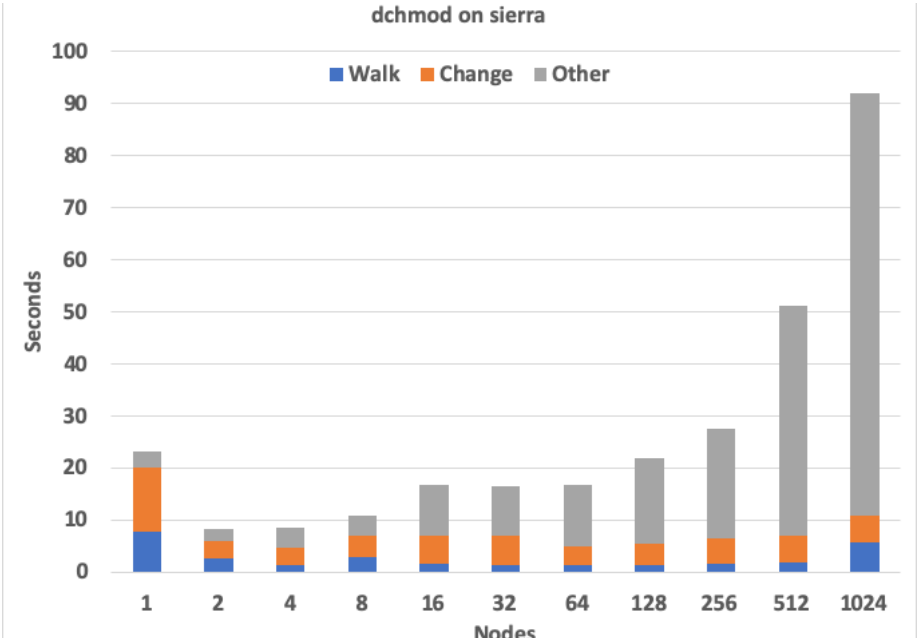


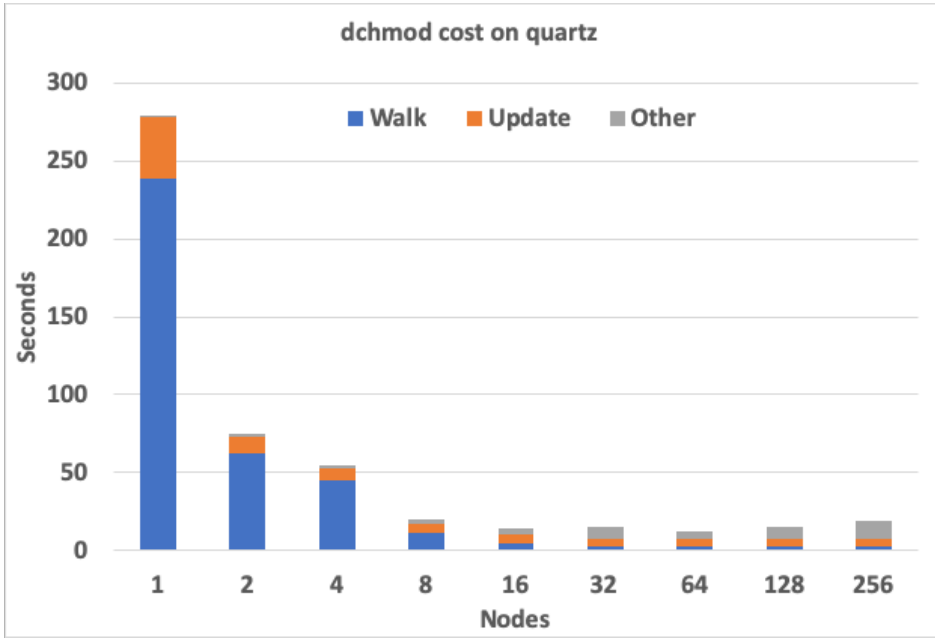
Single directory
200,000 files
128 MB per file
24.4 TB total

40 procs/node

Once the bandwidth is saturated, there is no value to scale to higher node counts.

dchmod changes group on files after the copy, a test that stresses file system metadata ops



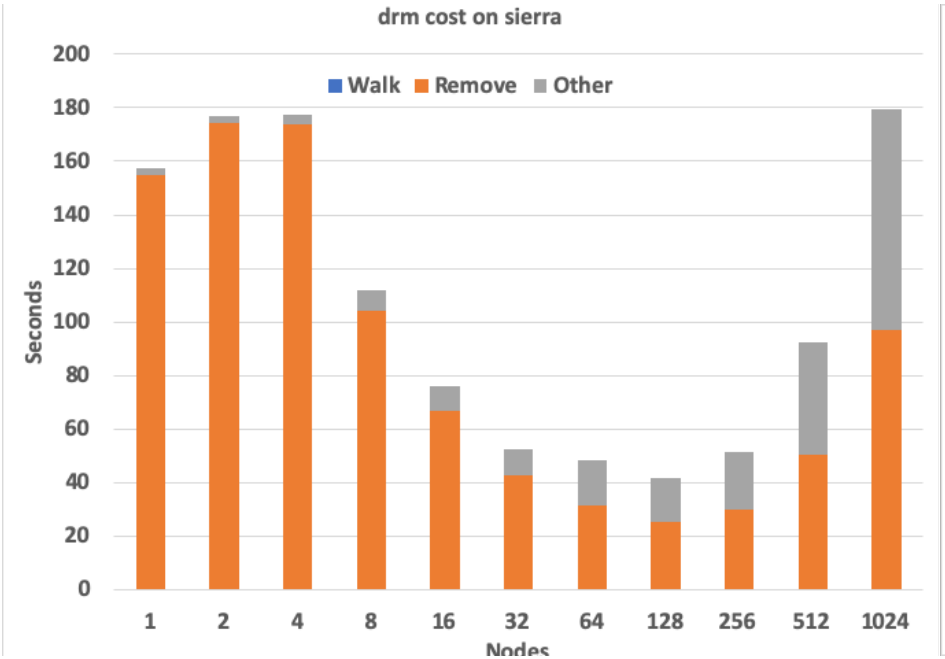


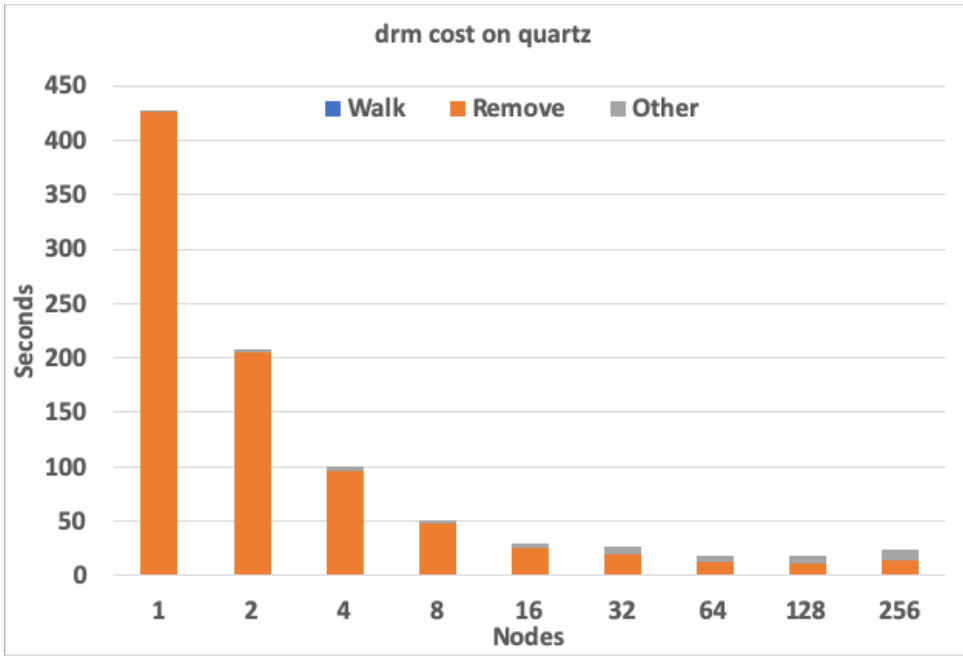
Sierra
GPFS, IBM Spectrum MPI, jsrun

Quartz
Lustre, MVAPICH2 MPI, SLURM

For quick jobs, the MPI job launch time can be significant.
Run those with smaller node counts.

drm removes the copy after the test





Sierra
GPFS, IBM Spectrum MPI, jsrun

Quartz
Lustre, MVAPICH2 MPI, SLURM

An example of bad scaling. It is unknown whether the cause in this case is mpiFileUtils or thrashing within the parallel file system.

Final remarks

https://github.com/hpc/mpifileutils

Things to keep in mind since mpiFileUtils are MPI programs

- Typically need to run in a job allocation
  - The sweet spot for most tools is about 2-4 nodes.
  - Grab more nodes for large datasets.
- Launch your job with mpirun
  - Plan to max out the CPU cores.
  - 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.
  - You may need to start over from the beginning if a tool is interrupted.
- Cannot pipe output of one tool to the input of another
  - The --input and --output file options are good approximations.
- Cannot easily check the return codes of tools
  - Check the output for errors.

mpiFileUtils are designed to be portable across all HPC sites and file systems

- Designed to work on any HPC system
  - Built with basic C, MPI, and POSIX/libc, so uses programming language and programming models that are ubiquitous in HPC
  - Available in Spack

```
spack install mpifileutils
```

- Designed to work on all HPC file systems
  - Lustre, GPFS, NFS have been targets from start
  - Intel actively working to add DAOS
  - Default algorithms should work with any POSIX-compliant file system
  - One may include file-system specific optimizations and algorithms

Take away: For most of your work, use standard tools like cp. Too slow? Then try mpiFileUtils.

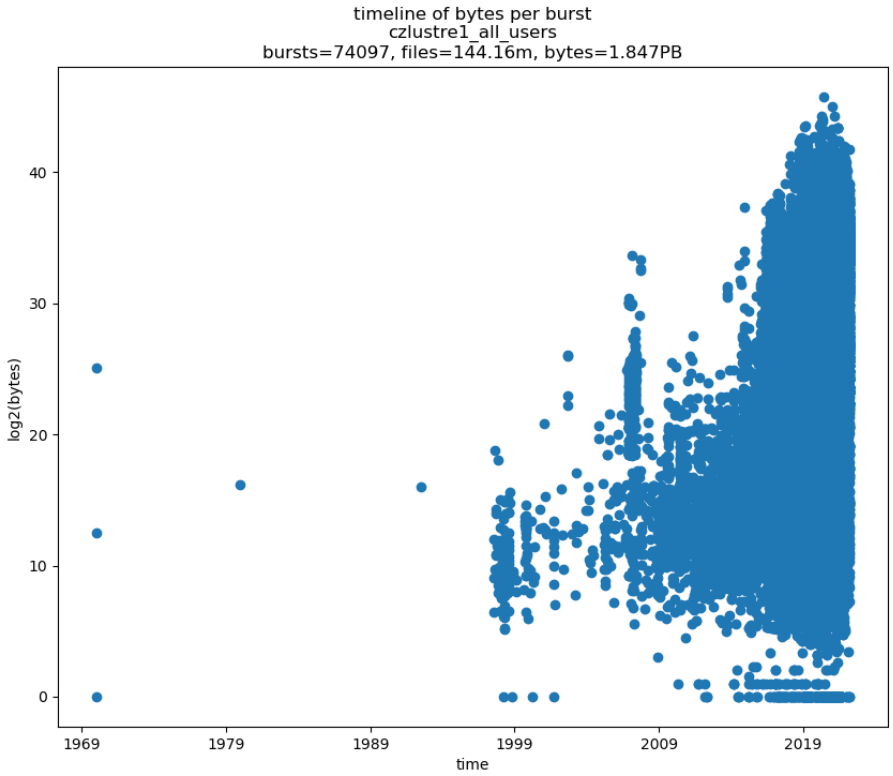
- Resources:
  - Documentation: https://mpifileutils.readthedocs.io
  - Code: https://github.com/hpc/mpifileutils
  - New users and new collaborators are always welcome!

- From its beginning, mpiFileUtils has been a multi-organizational open-source effort
  - Numerous people have contributed over time
  - https://github.com/hpc/mpifileutils/blob/master/AUTHORS
  - https://github.com/hpc/mpifileutils/graphs/contributors
  - Particularly: Jon Bringhurst, Jharrod LaFon, Danielle Sikich, Dalton Bohning, Elsa Gonsiorowski, Feiyi Wang, Xi Li, and Zheng Gu

Thank you MVAPICH team!

- The MPI programming model has enabled HPC users to continually scale up their problems for decades …
- … but only because of the hard work done by the people who implement MPI, and in particular due to the leading R&D from the MVAPICH team.

MPI = scalability.h
MVAPICH = scalability.c



Extra

Frequently used mfu_flist functions. Some are collectives, some are local to the calling process.

- **walk** - recursively walk one or more paths to create a list
- **stat** - given a list, execute stat on each item
- **filter** - apply find-like tests to find a matching subset of items
- **depth** - split list into sublists dividing items by depth
- **remap/spread** - reassign elements to different ranks
- **sort** - globally order elements across ranks, based on item properties (path, name, size, user, access time)
- **copy** - copy items in source list to a destination on the file system
- **mkdir/mknod** - create directories or inodes on file system
- **unlink** - remove all items named in a list from the file system
- **read/write** - load list from a file, write list to a file
- **pack/unpack** - serialize an element for network transfer
- **query properties of global list**, such as global list size and offset of local rank
- **iterate over local elements**
- **query/set properties of a local element**

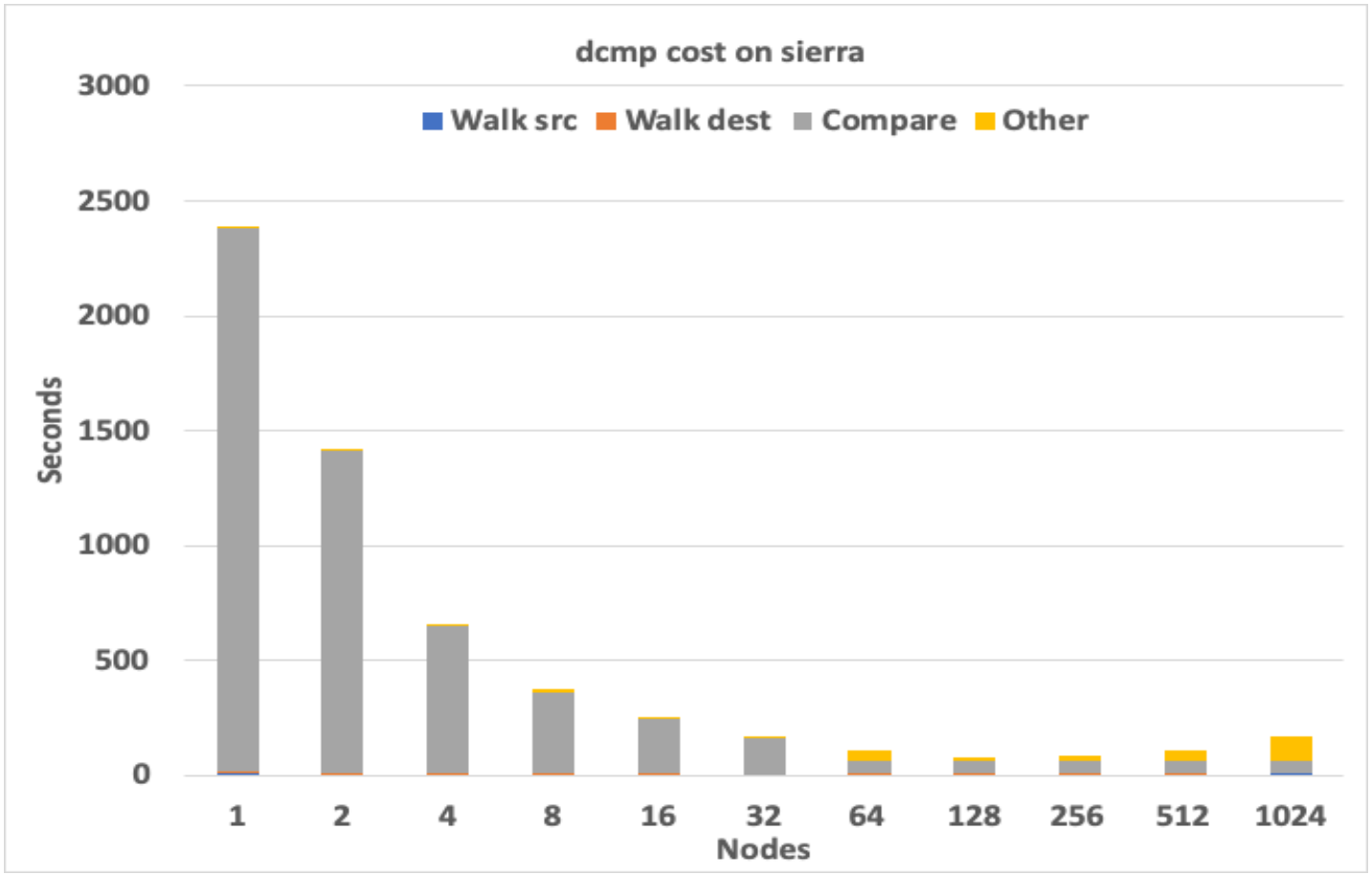
An example tool: walk a path, write out list of all regular files having more than one hardlink

```
mfu_flist flist = mfu_flist_new();                          walk path(s) to get
mfu_flist_walk_param_paths(numpaths, paths,                 initial list of items
    walk_opts, flist);

                                                            create subset list
mfu_flist flist_links = mfu_flist_subset(flist);
uint64_t size = mfu_flist_size(flist);                      iterate over all local
for (uint64_t idx = 0; idx < size; idx++) {                 items in initial list
    mfu_filetype type = mfu_flist_file_get_type(flist, idx);
    if (type != MFU_TYPE_FILE)                              check whether item is a
        continue;                                           regular file

    const char* file = mfu_flist_file_get_name(flist, idx);
    mfu_lstat(file, &statbuf);                              copy item to subset if
    if (statbuf.st_nlink > 1)                               hardlink count is more
        mfu_flist_file_copy(flist, idx, flist_links);       than one
}
mfu_flist_summarize(flist_links);                           finalize subset list

mfu_flist_write_cache(outputname, flist_links);             write subset list out
                                                            to a file
mfu_flist_free(&flist_links);
mfu_flist_free(&flist);                                     free lists
```

Same tool in six lines of python

```
import os
import stat
import mpifileutils as mfu                    import mpifileutils


flist = mfu.FList("/path/to/walk")            walk target path


hardlinks = flist.subset(lambda f:            sublist of
  f.type == mfu.TYPE_FILE and                   reg files with
  os.stat(f.name)[stat.ST_NLINK] > 1)           hardlink > 1


hardlinks.write("hardlinks.mfu")              write sublist to file
```

dcmp checks the new copy with the original, reads back and compares all bytes

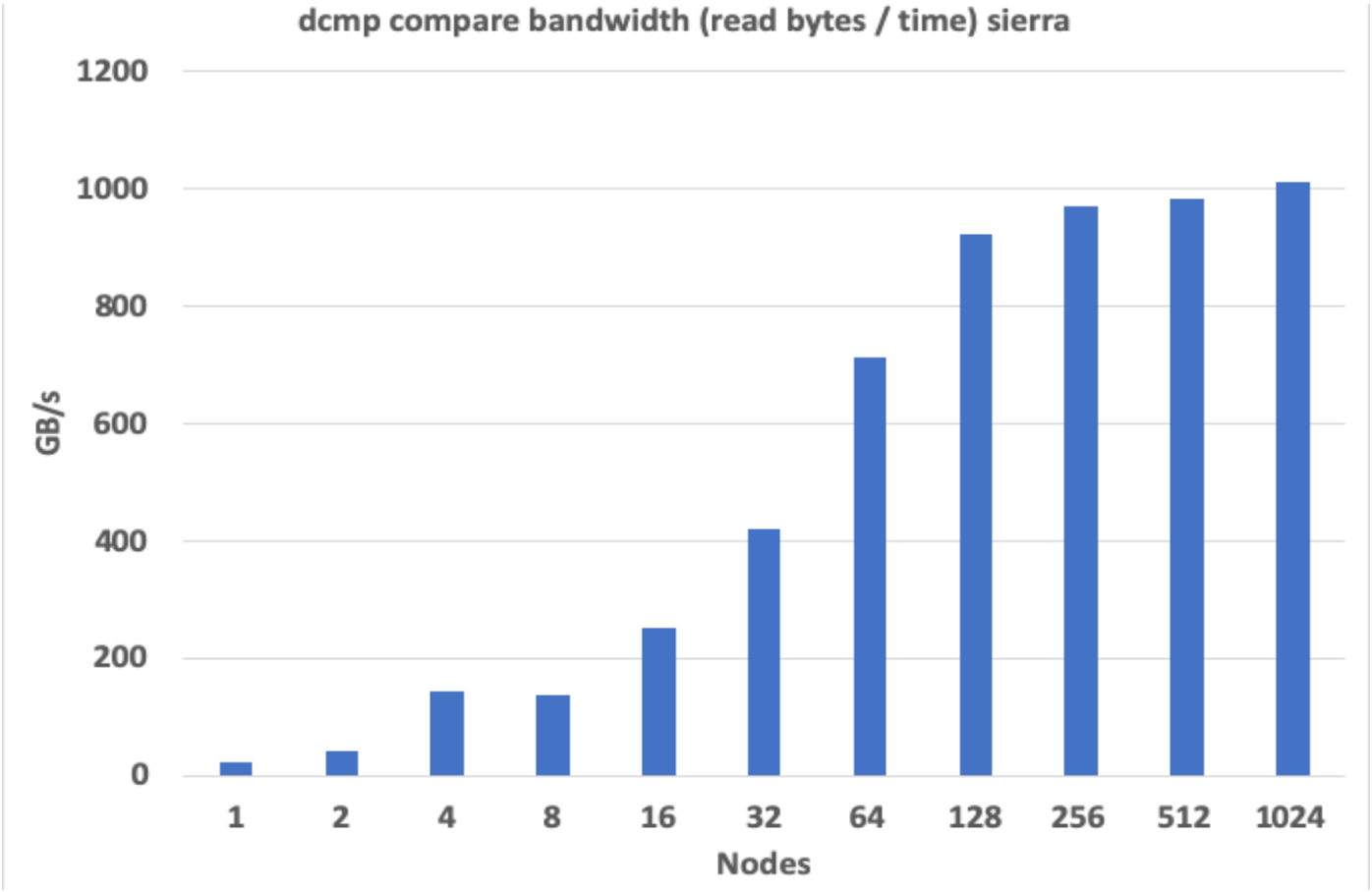


Single directory
200,000 files
128 MB per file
24.4 TB total

40 procs/node

It is recommended to run dcmp after dcp to verify contents.

dcmp saturates read bandwidth and holds steady at 1 TB/s on Sierra



Single directory
200,000 files
128 MB per file
24.4 TB total

40 procs/node

dcmp is read heavy and can often reach peak file system bandwidth.