unifyFS Tutorial

PRESENTER: **CHEN WANG**
CONTRIBUTORS: MICHAEL BRIM, ADAM MOODY, SEUNG-HWAN LIM, ROSS MILLER, SWEN BOEHM, CAMERON STANAVIGE, KATHRYN MOHROR(PI), SARP ORAL

What is UnifyFS?

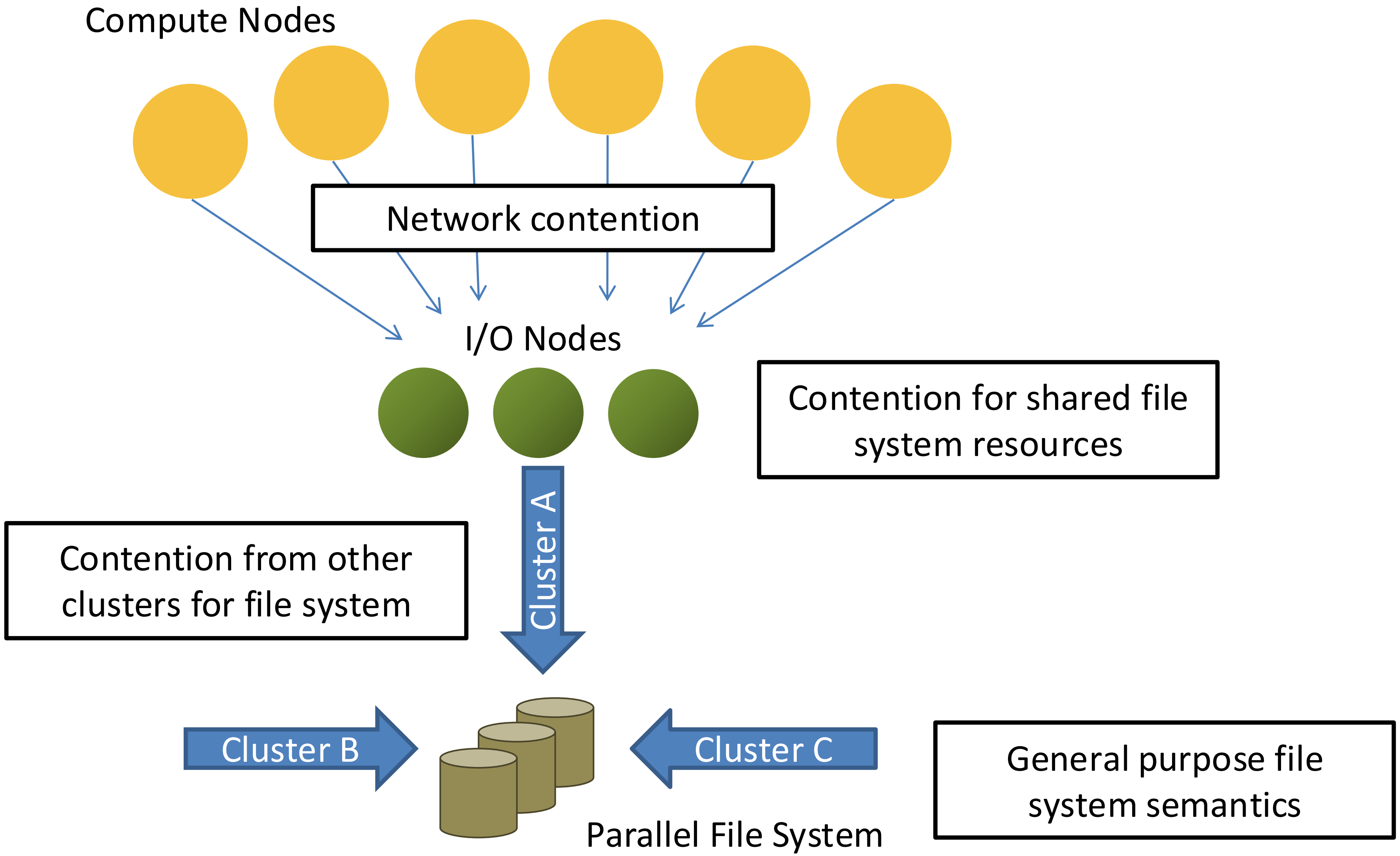
- **An ephemeral, user-level shared file system for burst buffers**
- Goal: make using burst buffers as *easy* as writing to the parallel file system and orders of magnitude *faster*

```
int main(int argc, char **argv) {
 MPI_Init(argc, argv);

 for (t = 0; t < TIMESTEPS; t++) {

  /* do work ... */

  checkpoint();
 }

 MPI_Finalize();

 return 0;
}
```

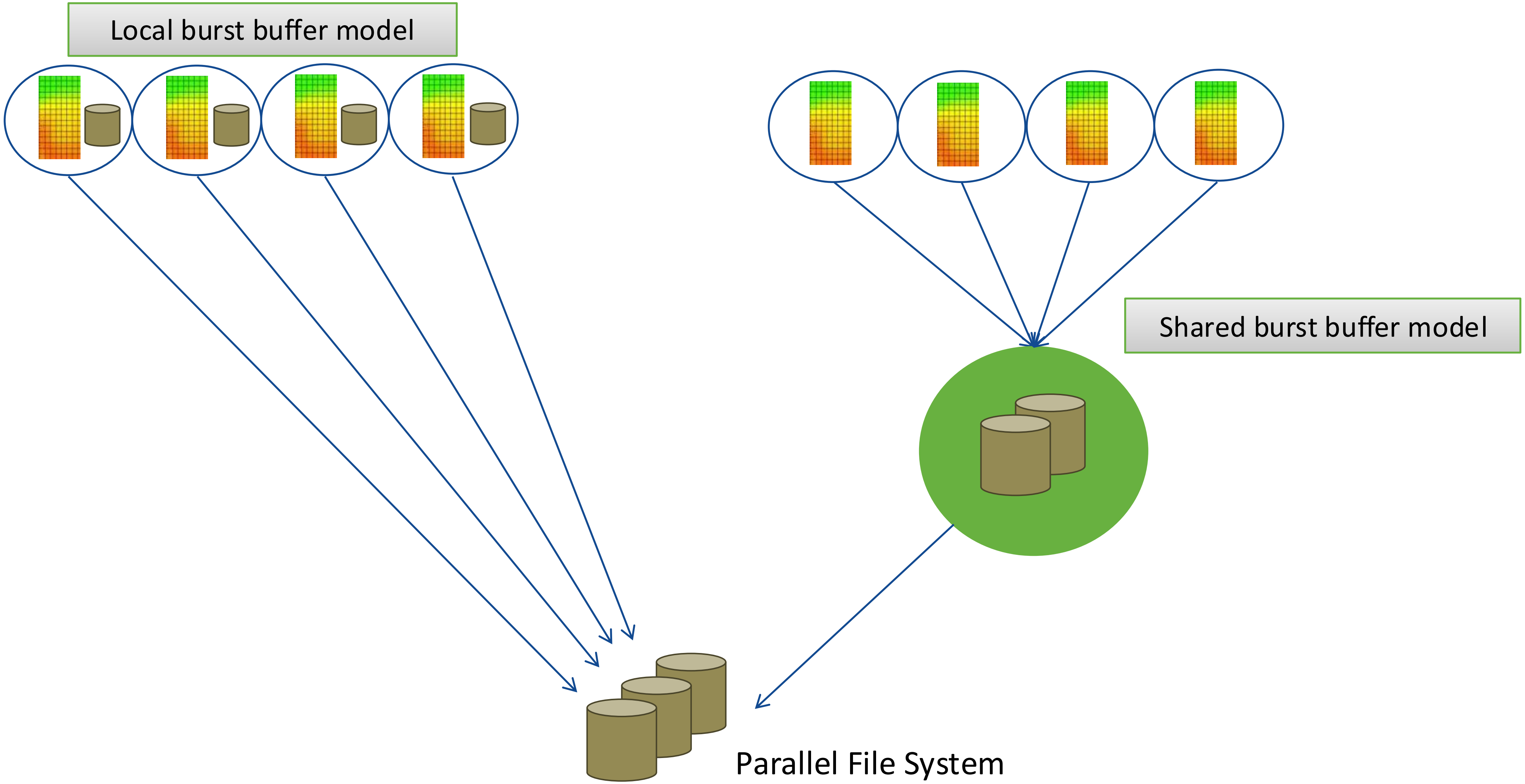
```
void checkpoint(void) {
 int rank;

 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 // file = “/pfs/shared.chpt”;
 file = “/unifyfs/shared.ckpt”;

 File *fs = fopen(file, “w”);

 if (rank == 0)
  fwrite(header, ..., fs);

 long offset = header_size +
               rank*state_size;
 fseek(fs, offset, SEEK_SET);
 fwrite(state, ..., fs);
 fclose(fs);
}
```

The only required change is to use **/unifyfs** instead of /pfs

Writing data to the parallel file system is expensive



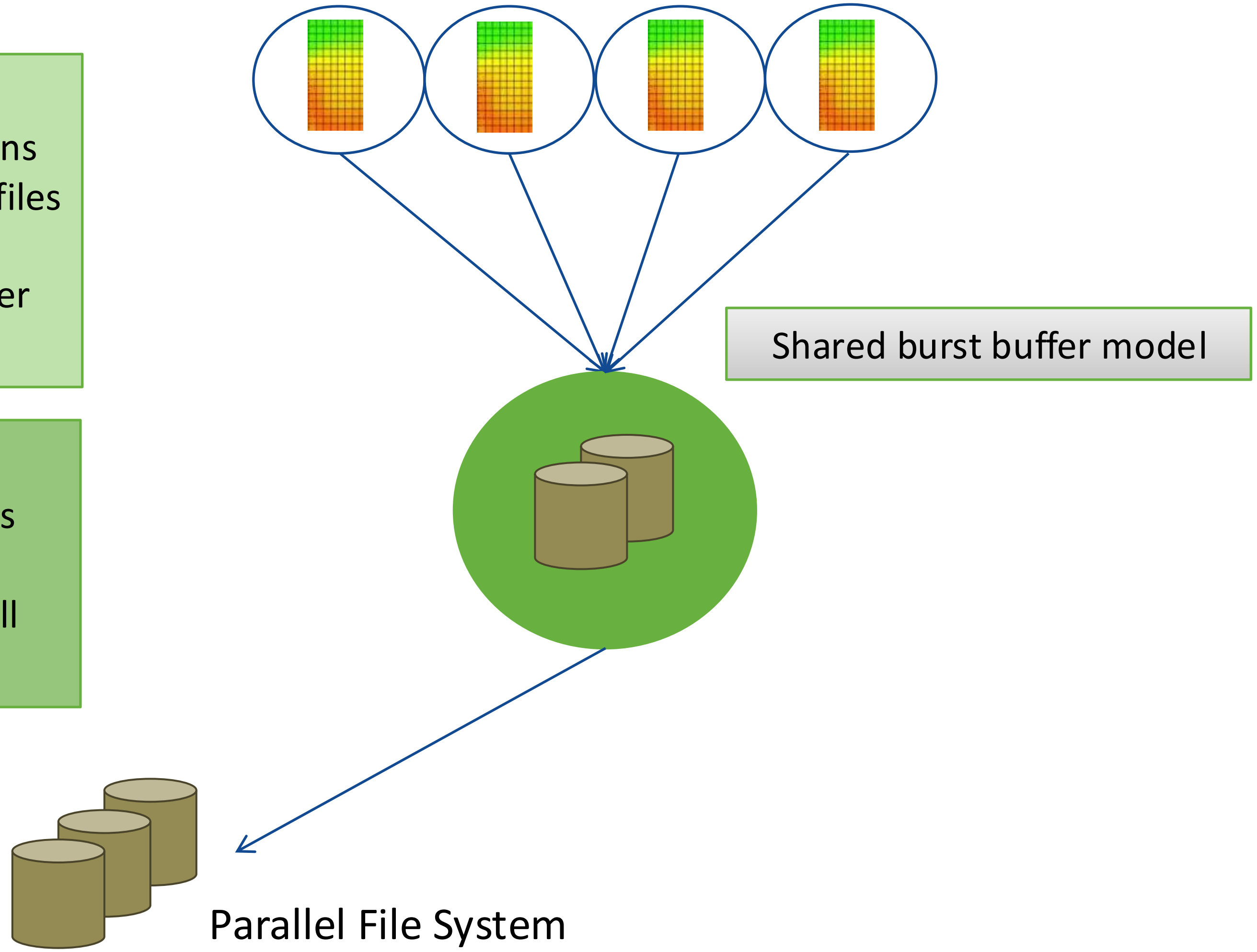
HPC Storage is becoming more complex



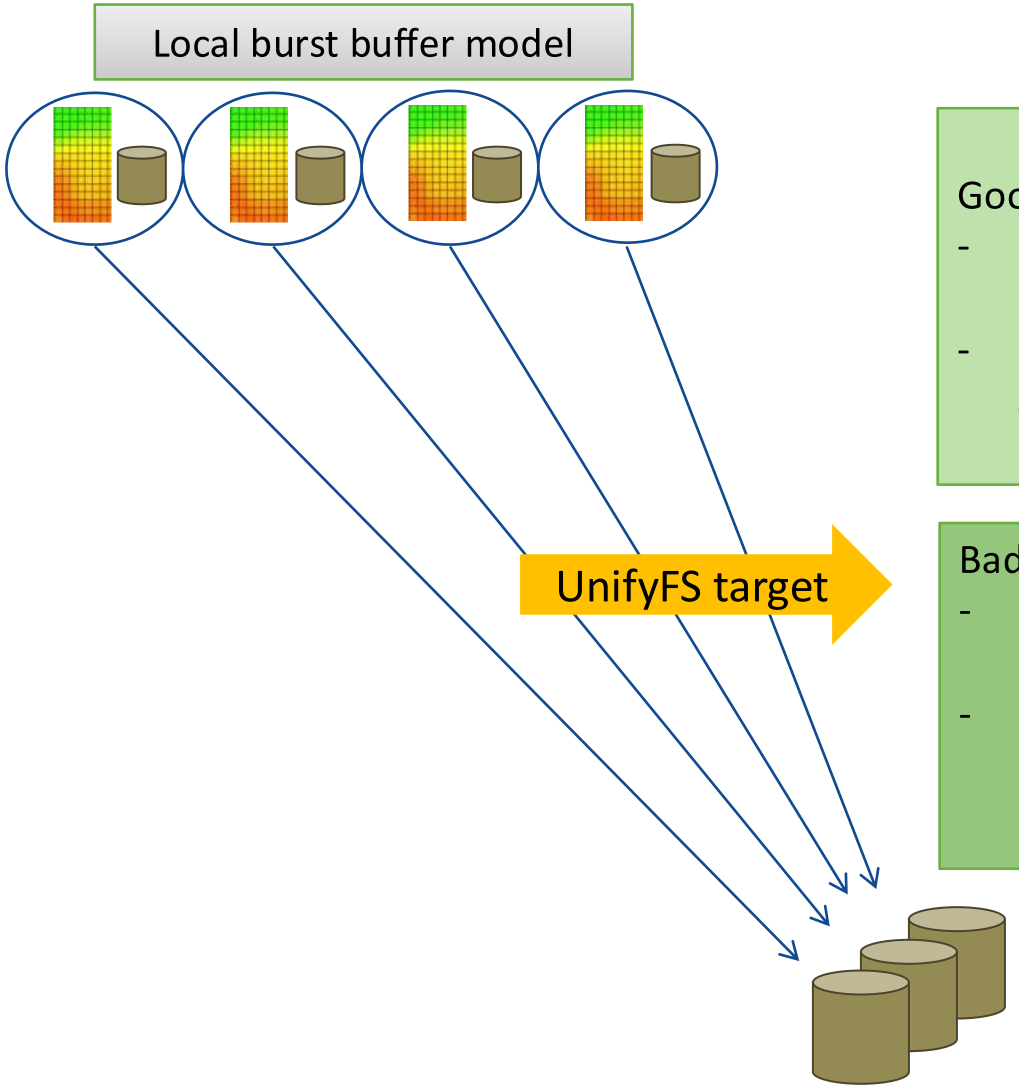
HPC Storage is becoming more complex

Good:
- Easy for applications that write shared files
- Easy for producer/consumer applications

Bad:
- Not quite as fast as node-local
- Contention can still be an issue



HPC Storage is becoming more complex



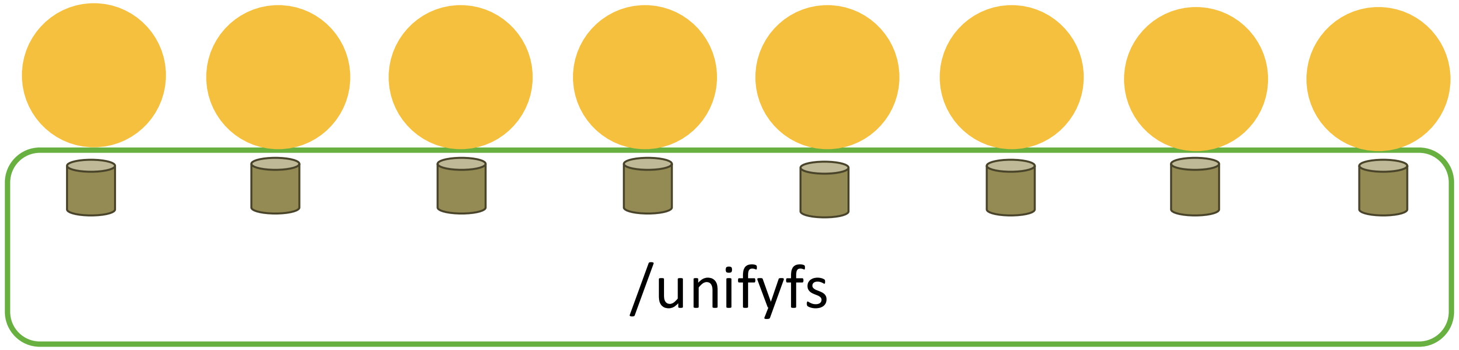
Parallel File System

Good:
- Fast, no contention problems
- Potential for excellent scaling performance

Bad:
- What about shared files?
- Users have to explicitly mange local storage devices

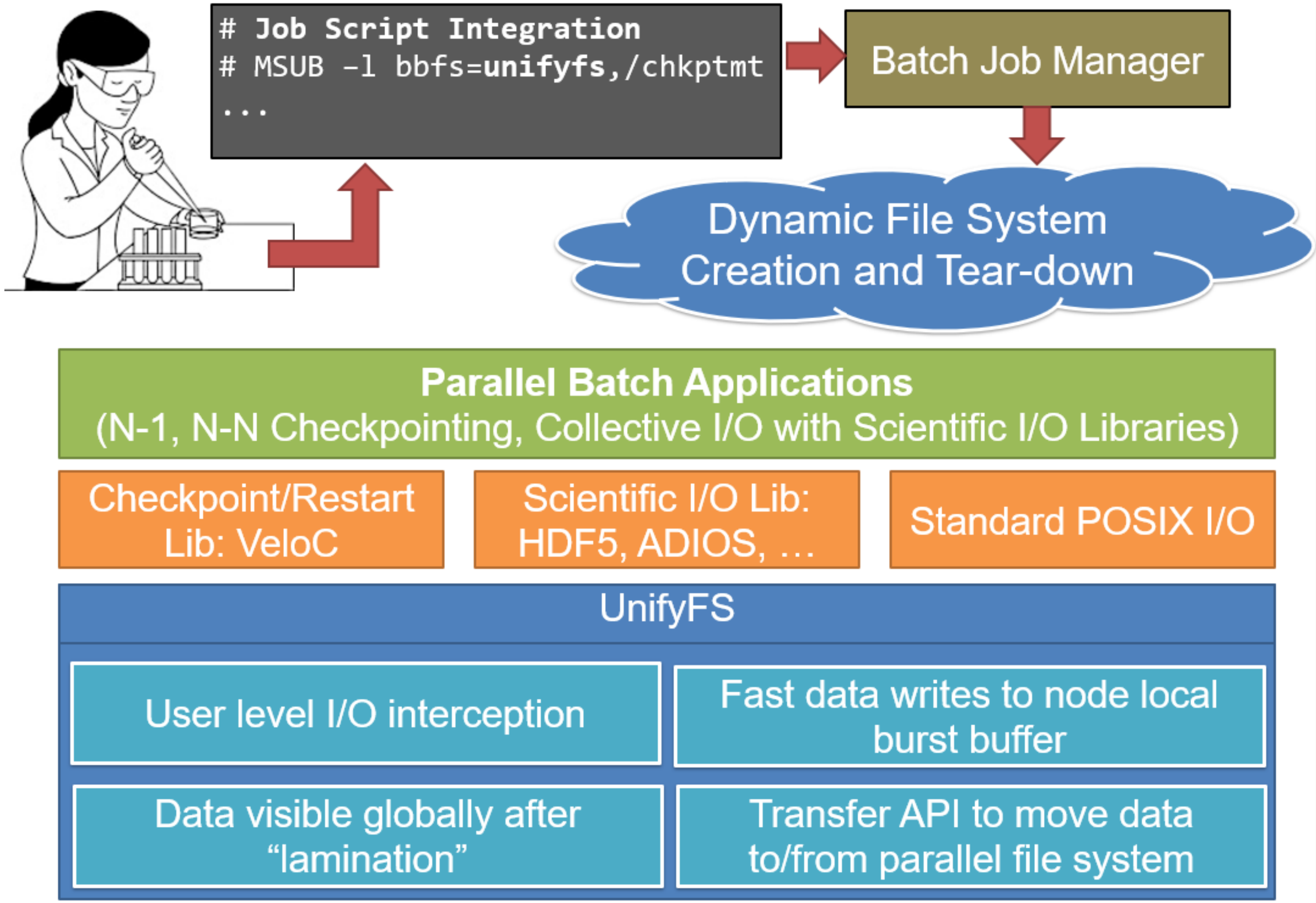
UnifyFS makes sharing files on node-local storage easy and fast

- **Problem:** Sharing files on node-local storage is not natively supported



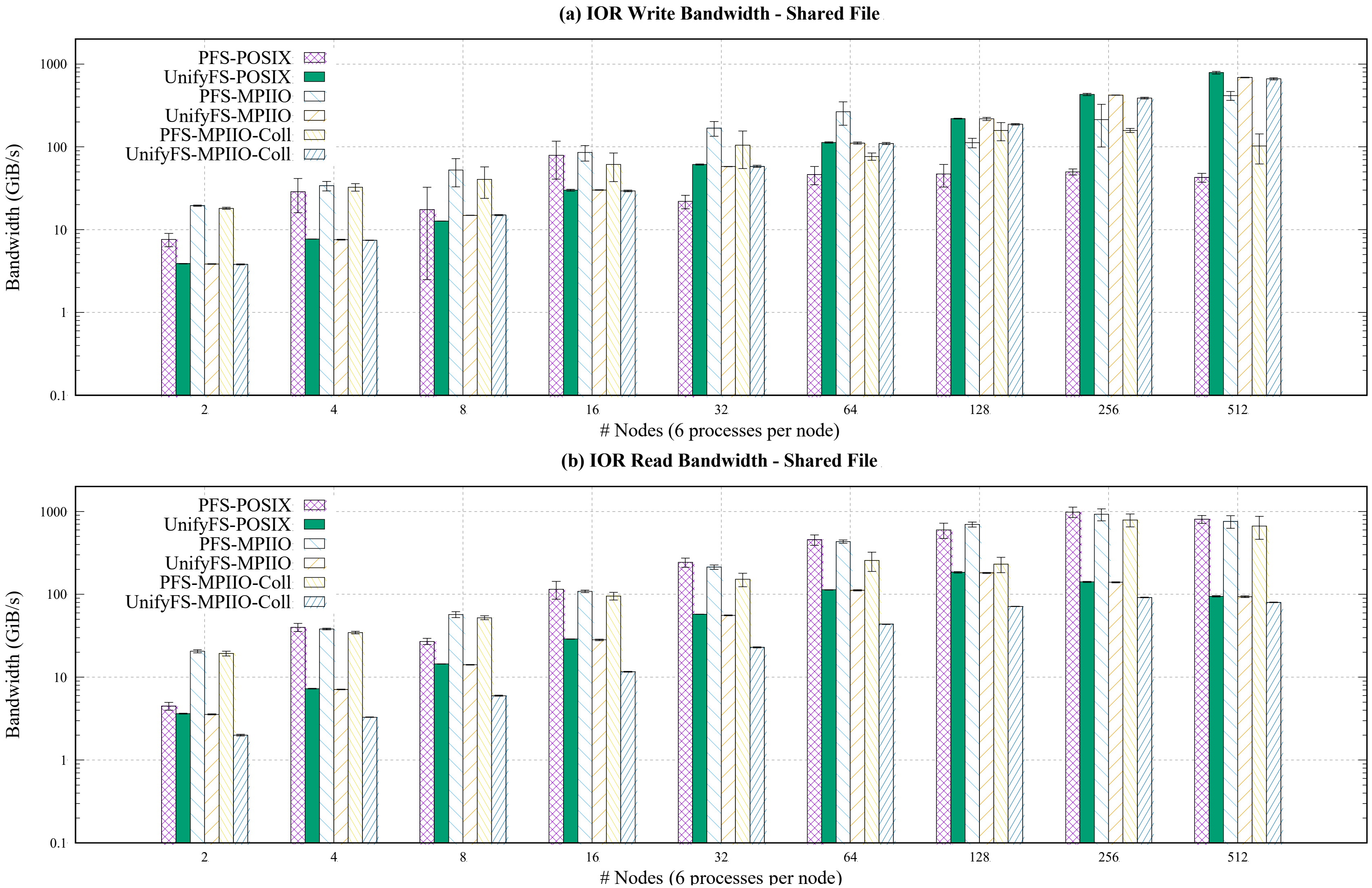
- UnifyFS makes sharing files **easy**
  - Presents a shared namespace across distributed, independent storage devices
  - Used directly by applications or indirectly via higher level libraries like MPI-IO, HDF5, PnetCDF, ADIOS, etc.
- UnifyFS is **fast**
  - Tailored for specific HPC workloads, e.g., checkpoint/restart, visualization output
  - Each UnifyFS instance exists only within a single job, no I/O contention with other jobs on the system
  - UnifyFS can use a combination of memory-backed and file-backed local storage

UnifyFS is designed to work completely in user space for a single job



UnifyFS targets local burst buffers because they are fast and scalable

- IOR v3.3 shared-file scaling on OLCF Summit
- UnifyFS (v1.0c)
  - All write data stored in NVMe (not using memory storage)
    - NVMe provides peak 2 GiB/s write and 5 GiB/s read per node
  - Write performance scaling well
    - up to 128 nodes, follows the the cumulative theoretical throughput of the node-local burst buffers
    - fairly consistent performance regardless of I/O method
  - Read performance (without metadata caching) scales less well
- Alpine parallel file system (PFS) performance is highly variable due to contention
  - MPI-IO has better write scaling performance than POSIX-IO
  - GPFS read caching works well



UnifyFS offers customizable file system semantics to meet varied application requirements

- By default, UnifyFS makes simplifying assumptions about how you access your data
  - **Assumptions meet common use cases for HPC I/O: checkpointing, output, producer/consumer**
  - I/O occurs in phases (except in limited circumstances, e.g., reads by a process of the data it wrote)
  - No two processes write to the same byte/offset concurrently
  - Without explicit synchronization, processes may not see updates written by processes on another node
  - Go here for more information: https://unifyfs.readthedocs.io/en/latest/assumptions.html
- The default semantics are compatible with MPI-IO and HDF5 parallel-independent I/O

VerifyIO: Is my application compatible with UnifyFS?

- Recorder
  - Tracing framework that can capture I/O function calls at multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX I/O
  - GitHub: https://github.com/uiuc-hpc/Recorder/
  - Publications:

- VerifyIO
  - VerifyIO takes Recorder traces and determines whether I/O synchronization is correct based on the underlying file system semantics (e.g., posix, commit) and synchronization semantics (e.g., posix, MPI)
  - Use "commit" semantics to check compatibility with UnifyFS
  - GitHub: https://github.com/wangvsa/VerifyIO/
  - Publications:

Can I use UnifyFS if I use an I/O library?

- Yes! UnifyFS works with HDF5 I/O as well as other I/O libraries (e.g., MPI-IO)
  - We are partnered with HDF5 in ECP ExaIO so we test it the most

```
int main(int argc, char* argv[])
{
  MPI_Init(argc, argv);

  for(int t = 0; t < TIMESTEPS; t++)
  {
      /* ... Do work ... */

      checkpoint(dset_data);
  }

  MPI_Finalize();
  return 0;
}
```

```
void checkpoint(dset_data)
{
    char file[256];

    sprintf(file, “/lustre/shared.ckpt”);

    file_id = H5Fopen(file, …);
    dset_id = H5Dopen2(file_id, “/dset”, …);

    H5DWrite(dset_id, …, dset_data);
    H5Dclose(dset_id);
    H5Fclose(file_id);
    return;
}
```

Can I use UnifyFS if I use an I/O library?

- **Build and run your application with UnifyFS, change the file path(s)**

```
int main(int argc, char* argv[])
{
  MPI_Init(argc, argv);

  for(int t = 0; t < TIMESTEPS; t++)
  {
      /* ... Do work ... */

      checkpoint(dset_data);
  }

  MPI_Finalize();
  return 0;
}
```

```
void checkpoint(dset_data)
{
    char file[256];

    sprintf(file, “/unifyfs/shared.ckpt”);

    file_id = H5Fopen(file, …);
    dset_id = H5Dopen2(file_id, “/dset”, …);

    H5DWrite(dset_id, …, dset_data);
    H5Dclose(dset_id);
    H5Fclose(file_id);
    return;
}
```

Tutorial: What happens when I run my code **without** UnifyFS?

1. user application calls "fopen"

```
fopen("/<path>/dset.txt")
```

```
fopen()
```

(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@glibc
fprintf	fprintf@glibc
fclose	fclose@glibc

Tutorial: What happens when I run my code **without** UnifyFS?

1. user application calls "fopen"

```
fopen("/<path>/dset.txt")
```

2. Address to glibc fopen is looked up via GOT/PLT

(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@glibc
fprintf	fprintf@glibc
fclose	fclose@glibc

```
fopen()
```

Tutorial: What happens when I run my code **without** UnifyFS?

1. user application calls “fopen”

```
fopen(“/<path>/dset.txt”)
```

2. Address to glibc fopen is looked up via GOT/PLT

(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@glibc
fprintf	fprintf@glibc
fclose	fclose@glibc

3. fopen@glibc is executed

```
fopen()
```

Tutorial: What happens when I run my code **with** UnifyFS and Gotcha?

1. user application calls "fopen"

```
fopen("/<path>/dset.txt")
```

(libunifyfs_gotcha)

```
UNIFYFS_WRAP(fopen)
{
 if(intercept(path){
   …

  //using UnifyFS

   …
  } else {
   __real_fopen(path)
  }
}
```

(glibc)

```
fopen()
```

(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@unifyfs_gotcha
fprintf	fprintf@unifyfs_gotcha
fclose	fclose@unifyfs_gotcha

Tutorial: What happens when I run my code **with** UnifyFS and Gotcha?

1. user application calls "fopen"

2. Address to glibc fopen is rewritten to UnifyFS's "fopen" in GOT/PLT

```
fopen("/<path>/dset.txt")
```

(libunifyfs_gotcha)

```
UNIFYFS_WRAP(fopen)
{
 if(intercept(path){
   …

  //using UnifyFS

   …
  } else {
   __real_fopen(path)
  }
}
```

(glibc)

```
fopen()
```

(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@unifyfs_gotcha
fprintf	fprintf@unifyfs_gotcha
fclose	fclose@unifyfs_gotcha

Tutorial: What happens when I run my code **with** UnifyFS and Gotcha?

1. user application calls “fopen”

```
fopen(“/<path>/dset.txt”)
```

2. Address to glibc fopen is rewritten to UnifyFS’s “fopen” in GOT/PLT

(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@unifyfs_gotcha
fprintf	fprintf@unifyfs_gotcha
fclose	fclose@unifyfs_gotcha

(libunifyfs_gotcha)

```
UNIFYFS_WRAP(fopen)
{
 if(intercept(path){
   …

  //using UnifyFS

   …
  } else {
   __real_fopen(path)
  }
}
```

3. fopen call is re-directed to UnifyFS library for processing

(glibc)

```
fopen()
```

Tutorial: What happens when I run my code **with** UnifyFS and Gotcha?

1. user application calls “fopen”

2. Address to glibc fopen is rewritten to UnifyFS’s “fopen” in GOT/PLT

```
fopen(“/<path>/dset.txt”)
```

(libunifyfs_gotcha)

```
UNIFYFS_WRAP(fopen)
{
 if(intercept(path){
   …

  //using UnifyFS

   …
  } else {
   __real_fopen(path)
  }
}
```

3. fopen call is re-directed to UnifyFS library for processing

(glibc)

```
fopen()
```

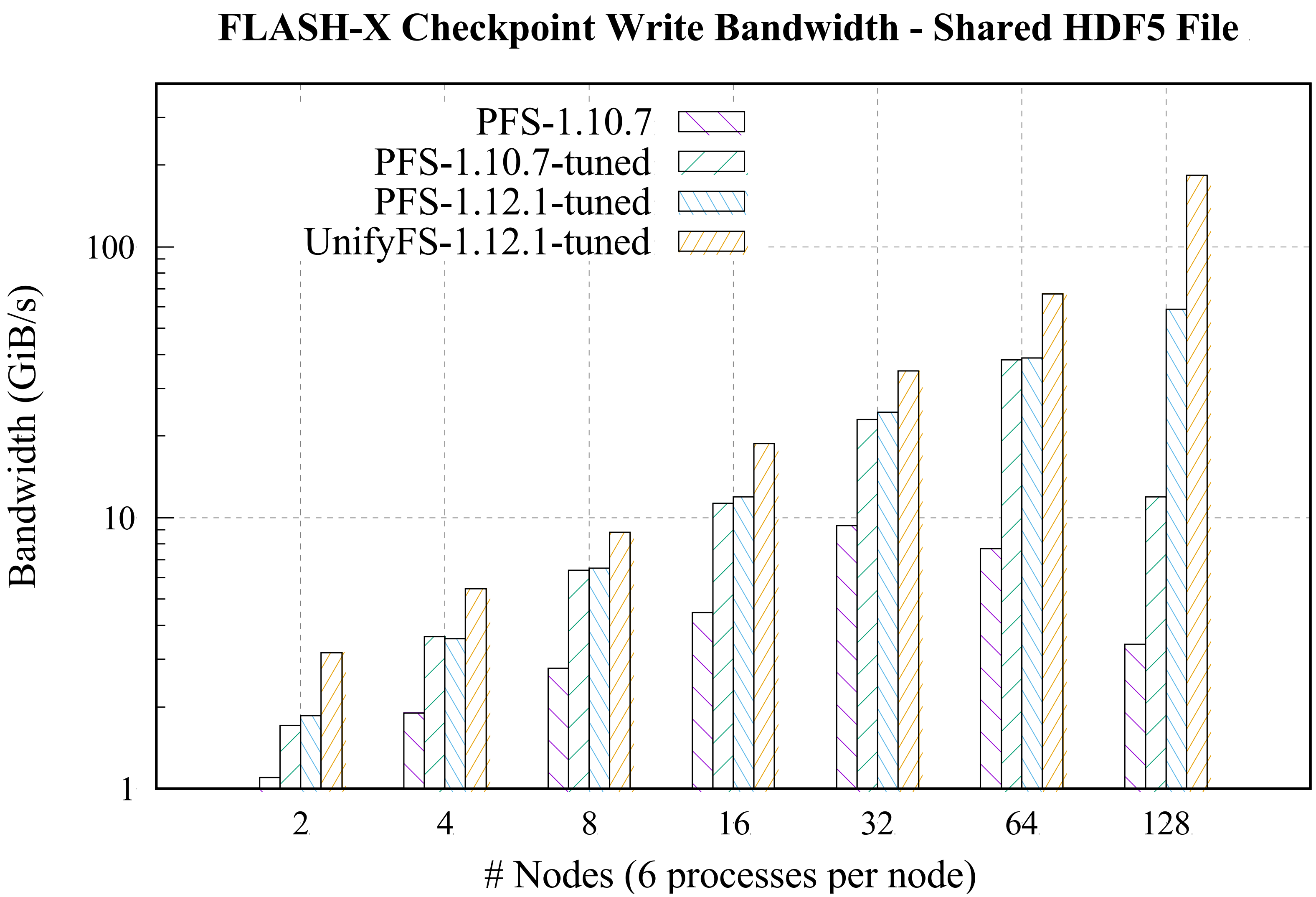
(GOT/PLT)

Symbol Name	Symbol Location
fopen	fopen@unifyfs_gotcha
fprintf	fprintf@unifyfs_gotcha
fclose	fclose@unifyfs_gotcha

4. If path is not under UnifyFS, then the wrapper calls glibc fopen

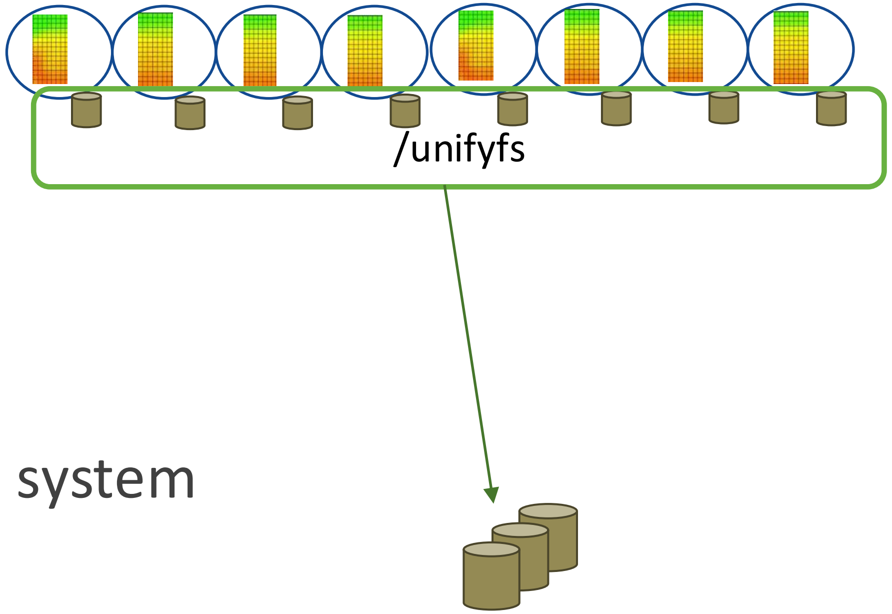
HDF5 Example Application: FLASH-IO on OLCF Summit

- FLASH-X Astrophysics code
  - https://flash-x.org/
  - FLASH-IO benchmark configuration writes checkpoint and plot files
    - ~ 72 GB of checkpoint data per node
    - ~ 220 MB of plot data per node
- "PFS" is Parallel File System
  - OLCF Alpine (IBM Spectrum Scale FS)
- UnifyFS uses only node-local NVMe devices (2 GiB/s peak write bandwidth)
- HDF5 versions
  - v1.10.7 is Summit default
  - v1.12.1 includes recent improvements
- "tuned" application includes two optimizations:
  1. a good MPI-IO configuration for Alpine
  2. elimination of a redundant `H5Fflush()` call per write operation



Walkthrough Tutorial

- get and build UnifyFS
- build your application to use UnifyFS
- set up your application environment to use UnifyFS
- run your application with UnifyFS
- move your data between UnifyFS and the parallel file system



Tutorial: How do I get and build UnifyFS?

Tutorial Cluster

- Recent release is already installed

```
$ module load unifyfs
```

Got Spack?

```
$ spack install unifyfs
$ spack load unifyfs
```

Tutorial: How do I get UnifyFS without Spack?

- Not using Spack?
- UnifyFS can be found on GitHub: https://github.com/LLNL/UnifyFS

```
$ git clone https://github.com/LLNL/UnifyFS.git
```

unifyFS

We need you!

- Our goal is to provide **easy, portable,** and **fast** support for I/O-intensive applications.
- UnifyFS
  - https://github.com/LLNL/UnifyFS
- Recorder
  - https://github.com/uiuc-hpc/Recorder
- VerifyIO:
  - https://github.com/wangvsa/VerifyIO

Backup

Tutorial: How do I get and build UnifyFS?

UnifyFS variants for Spack installation

```
$ spack info unifyfs
⋮
Variants:
    Name        [Default]  Description
    ==========  =========  ==============================

    auto-mount  [True]     Enable automatic mount/unmount in MPI_Init/Finalize
    boostsys    [False]    Have Mercury use preprocessor headers from boost
    fortran     [True]     Build with gfortran support
    pmi         [False]    Enable PMI2 build options
    pmix        [False]    Enable PMIx build options
    preload     [False]    Enable support for optional LD_PRELOAD library
    spath       [True]     Normalize relative paths
```

Tutorial: How do I build UnifyFS without Spack?

- Build and install UnifyFS's dependencies
  - GOTCHA: https://github.com/LLNL/GOTCHA
  - Margo: https://github.com/mochi-hpc/mochi-margo
    - Mercury: https://github.com/mercury-hpc/mercury
      - libfabric: https://github.com/ofiwg/libfabric and/or bmi: https://github.com/radix-io/bmi/
    - Argobots: https://github.com/pmodels/argobots
    - JSON-C: https://github.com/json-c/json-c
  - Run our bootstrap script to automatically download and install our dependencies

```
$ cd UnifyFS
$ ./bootstrap
```

Tutorial: How do I build UnifyFS without Spack?

- Then build and install UnifyFS

```
$ export PKG_CONFIG_PATH=$INSTALL_DIR/lib/pkgconfig:
  $INSTALL_DIR/lib64/pkgconfig:$PKG_CONFIG_PATH
$ export LD_LIBRARY_PATH=$INSTALL_DIR/lib:$INSTALL_DIR/lib64:$LD_LIBRARY_PATH

$ ./configure --prefix=$INSTALL_DIR --enable-mpi-mount --enable-pmi
  --enable-fortran CPPFLAGS=-I$INSTALL_DIR/include
  LDFLAGS="-L$INSTALL_DIR/lib –L$INSTALL_DIR/lib64"

$ make
$ make install
```

- On Cray systems, the detection of MPI compiler wrappers requires passing `MPICC=cc` and `MPIFC=ftn` to the configure command

Tutorial: How do I modify my MPI application for UnifyFS?

- Example MPI application without UnifyFS (using native file system)
  - Simple application that writes “Hello World” to Lustre at `/lustre/dset.txt`

```
int main(int argc, char * argv[]) {
    FILE *fp;
     // program initialization
     // MPI setup

     // perform I/O
    fp = fopen(“/lustre/dset.txt”, “w”);
    fprintf(fp, “Hello World! I’m rank %d”, rank);
    fclose(fp);

     // clean up
    return 0;
}
```