CSE 5449: Intermediate Studies in Scientific Data Management

Lecture 17: Asynchronous I/O

Dr. Suren Byna

The Ohio State University

E-mail: byna.1@osu.edu

https://sbyna.github.io

03/23/2023

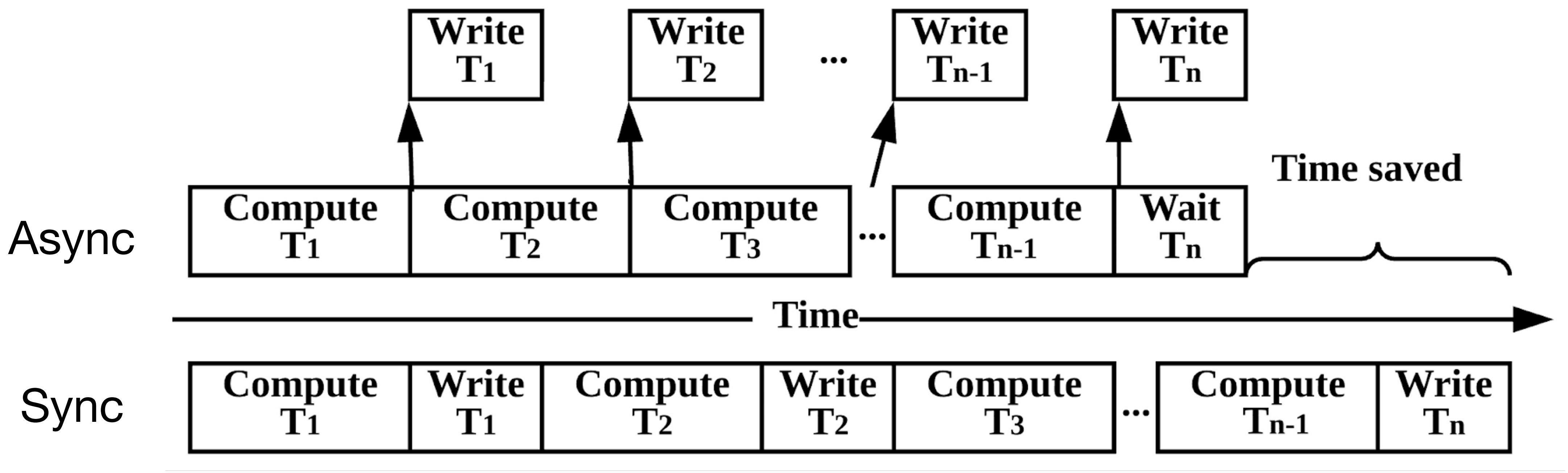
Today's class

- Any questions?

- Class presentation topic

- Today's class –
  - HDF5 optimizations – Async I/O

Why Async?



Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

POSIX asynchronous I/O (AIO)

- Applications initiate one or more I/O operations that are performed asynchronously (i.e., in the background)

- `aio_read()`
- `aio_write()`
- `aio_fsync()`
- `lio_listio()` – Enqueue multiple I/O requests using a single function call

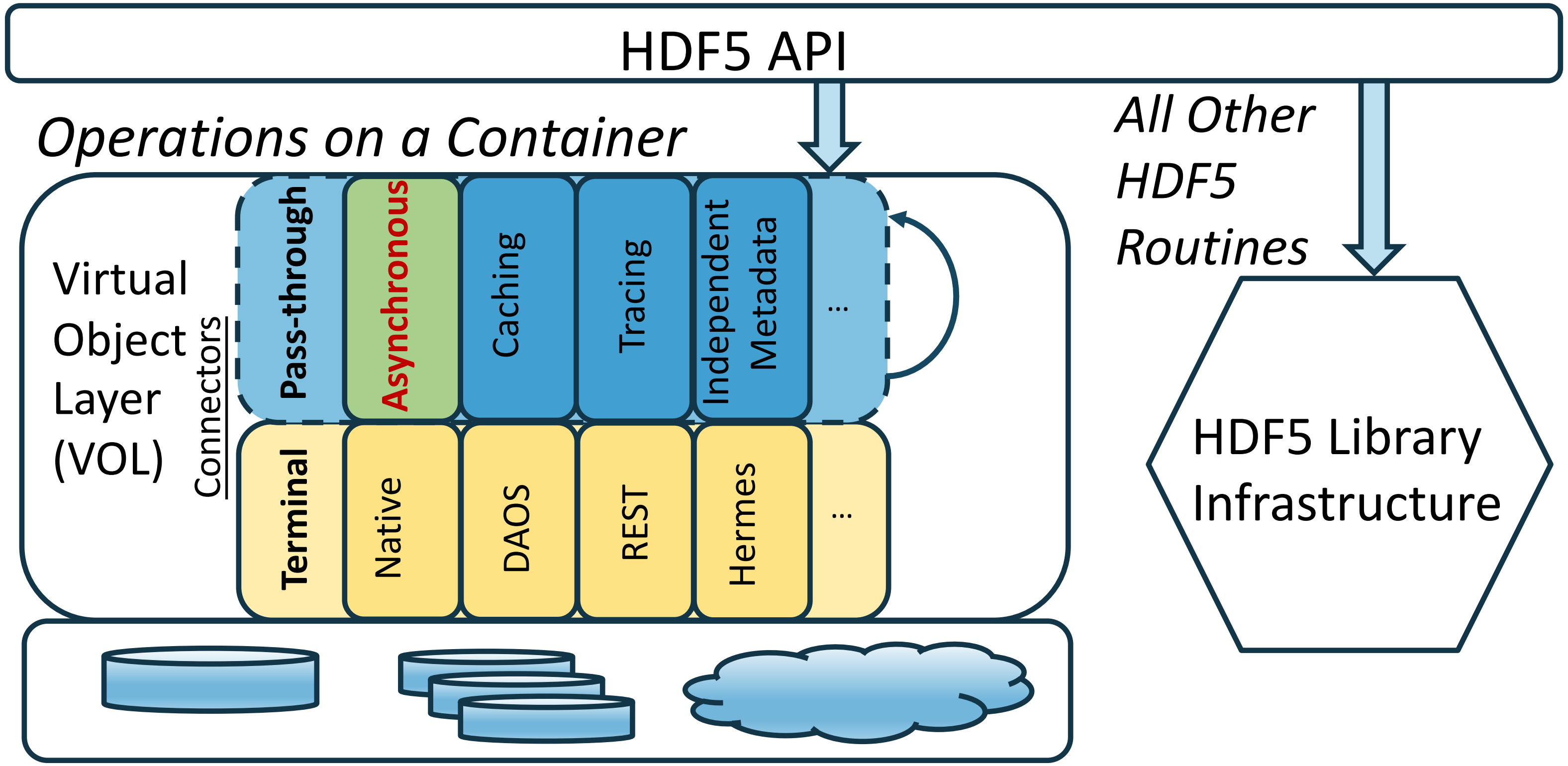
Asynchronous I/O control block – to control how asynchronous I/O operations are performed.

```
struct aiocb {
int aio_fildes; /* File descriptor */
off_t aio_offset; /* File offset */
volatile void *aio_buf; /* Location of buffer */
size_t aio_nbytes; /* Length of transfer */
int aio_reqprio; /* Request priority */
struct sigevent aio_sigevent; /* Notification
method */ int aio_lio_opcode; /* Operation to be
performed; lio_listio() only */ /* Various
implementation-internal fields not shown */ };
```

MPI-IO - Non-blocking I/O functions

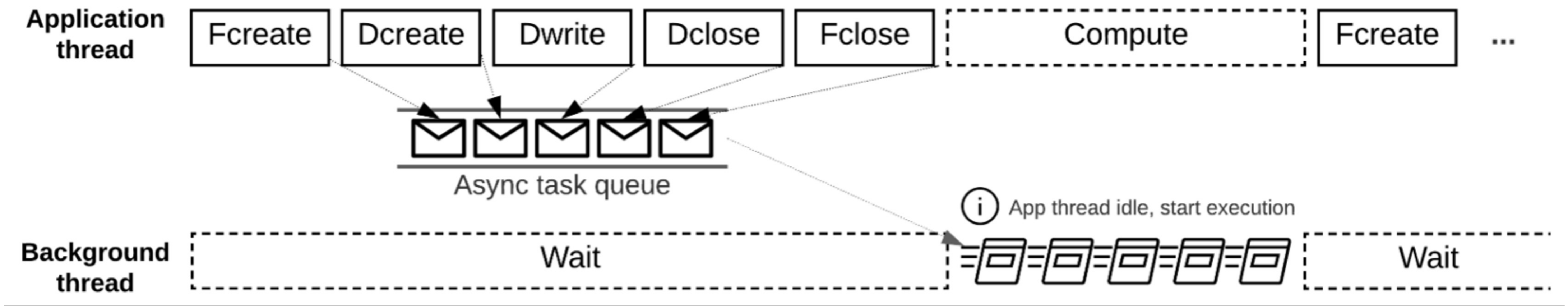
- MPI-IO non-blocking I/O functions
  - `MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)`
  - `MPI_File_iwrite_at`
  - `MPI_File_iwrite_all`
  - `MPI_File_iwrite_at_all`
  - `MPI_file_iread`
  - `MPI_file_iread_at`
  - `MPI_file_iread_all`
  - `MPI_file_iread_at_all`
- All these functions return a request ID
  - One can use this request ID to check on the status or wait for completion
  - `MPI_Wait(MPI_Request *request, MPI_Status *status)`

HDF5 Virtual Object Layer (VOL)



HDF5 Async VOL Implementation

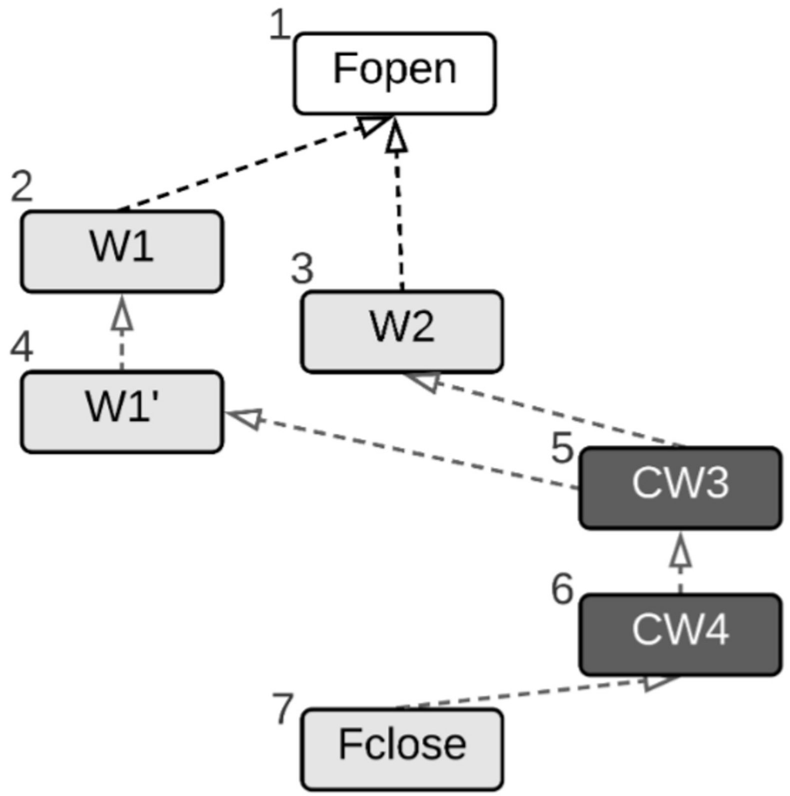
- Asynchronous task queue
- Transparent background thread execution using Argobots



Argobots: A Lightweight Low-level Threading Framework (https://www.argobots.org/)

Transparent Task Dependency Management

- All I/O operations can only be executed after a successful file create/open.
- A file close operation can only be executed after all previous operations in the file have been completed.
- All read or write operations must be executed after a prior write operation to the same object.
- All write operations must be executed after a prior read operation to the same object.
- All collective operations must be executed in the same order with regard to other collective operations.
- Only one collective operation may be in execution at any time.



Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

Explicit Control with Async and EventSet APIs

- Async version of HDF5 APIs
  - `H5Fcreate_async(fname, …, es_id);`
  - `H5Dwrite_async(dset, …, es_id);`
  - …
- Track and inspect multiple I/O operations with an ***EventSet ID***
  - `H5EScreate();`
  - `H5ESwait(es_id, timeout, &remaining, &op_failed);`
  - `H5ESget_err_info(es_id, ...);`
  - `H5ESclose(es_id);`

Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

Converting Existing HDF5 Codes

```
// Synchronous file create
fid = H5Fcreate(...);
// Synchronous group create
gid = H5Gcreate(fid, ...);
// Synchronous dataset create
did = H5Dcreate(gid, ..);
// Synchronous dataset write
status = H5Dwrite(did, ..);
// Synchronous dataset read
status = H5Dread(did, ..);
...
// Synchronous file close
H5Fclose(fid);
// Continue to computation

...




...
// Finalize
```

```
// Create an event set to track async operations
es_id = H5EScreate();
// Asynchronous file create
fid = H5Fcreate_async(.., es_id);
// Asynchronous group create
gid = H5Gcreate_async(fid, .., es_id);
// Asynchronous dataset create
did = H5Dcreate_async(gid, .., es_id);
// Asynchronous dataset write
status = H5Dwrite_async(did, .., es_id);
// Asynchronous dataset read
status = H5Dread_async(did, .., es_id);
...
// Asynchronous file close
status = H5Fclose_async(fid, .., es_id);
// Continue to computation, overlapping with asynchronous
    operations
...
// Finished computation, Wait for all previous operations in the
    event set to complete
H5ESwait(es_id, H5ES_WAIT_FOREVER, &n_running, &op_failed);
// Close the event set
H5ESclose(es_id);
...
// Finalize
```

Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

Example Code from AMReX

```
#ifdef AMREX_USE_HDF5_ASYNC
        hid_t dataset = H5Dcreate_async(grp, dataname.c_str(), H5T_NATIVE_DOUBLE, dataspace, H5P_DEFAULT, dcpl_id, H5P_DEFAULT, es_id_g);
#else
        hid_t dataset = H5Dcreate(grp, dataname.c_str(), H5T_NATIVE_DOUBLE, dataspace, H5P_DEFAULT, dcpl_id, H5P_DEFAULT);
#endif
        if(dataset < 0)
            std::cout << ParallelDescriptor::MyProc() << "create data failed!  ret = " << dataset << std::endl;

#ifdef AMREX_USE_HDF5_ASYNC
        ret = H5Dwrite_async(dataset, H5T_NATIVE_DOUBLE, memdataspace, dataspace, dxpl_col, a_buffer.dataPtr(), es_id_g);
#else
        ret = H5Dwrite(dataset, H5T_NATIVE_DOUBLE, memdataspace, dataspace, dxpl_col, a_buffer.dataPtr());
#endif
        if(ret < 0) { std::cout << ParallelDescriptor::MyProc() << "Write data failed!  ret = " << ret << std::endl; break; }
```

https://github.com/AMReX-Codes/amrex/blob/development/Src/Extern/HDF5/AMReX_PlotFileUtilHDF5.cpp#L721

Async Error Handling

- If an async operation fails, **<u>all</u>** of its dependent children will not execute and, no further operations can be added to the event set.
- Error information can be retrieved with:

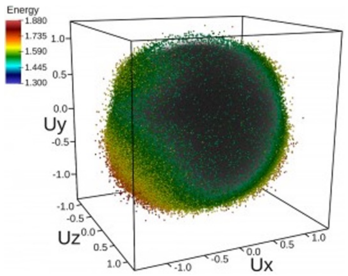
```
// Check if event set has failed operations
status = H5ESget_err_status(es_id, &es_err_status);
// Retrieve the number of failed operations in this event set
status = H5ESget_err_count(es_id, &es_err_count);
// Retrieve information about failed operations
status = H5ESget_err_info(es_id, 1, &err_info, &es_err_cleared);
// Retrieve API name, arguments list, file name, function name, and line number
printf(``API name: %s, args: %s, file name: %s, func name: %s, line number: %u'',
err_info.api_name, err_info.api_args, err_info.api_file_name, err_info.api_func_name,
err_info.api_line_num);
// Retrieve operation counter and operation timestamp
printf(``Op counter: %llu, Op timestamp: %llu'', err_info.op_ins_count, err_info.op_ins_ts);
```

How to use Async VOL

Detailed description in https://github.com/hpc-io/vol-async

- **Installation**
  - Compile HDF5 (github develop branch or released version 1.13+), with **thread-safety** support
  - Compile Argobots threading library
  - Compile Async VOL connector
- **Set environment variables**
  - `export LD_LIBRARY_PATH=$VOL_DIR/lib:$H5_DIR/lib:$ABT_DIR/lib:$LD_LIBRARY_PATH`
  - `export HDF5_PLUGIN_PATH="$VOL_DIR/lib"`
  - `export HDF5_VOL_CONNECTOR="async under_vol=0;under_info={}"`
- **Run the application (using the async and EventSet APIs)**
  - MPI must be initialized with **`MPI_THREAD_MULTIPLE`**

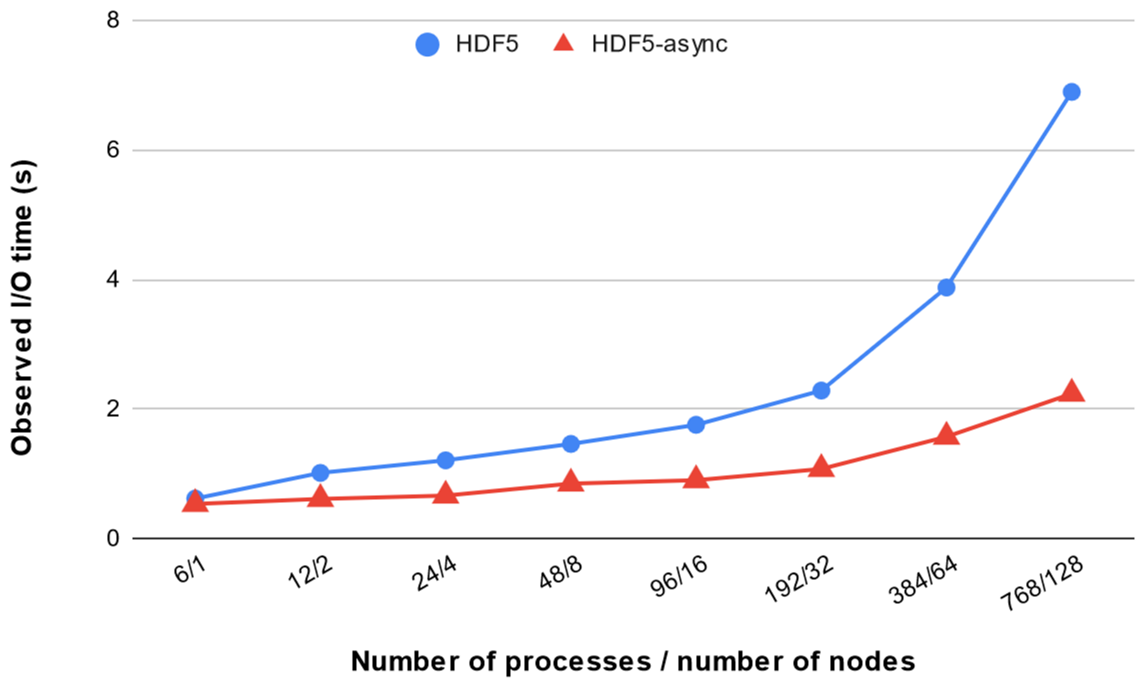
Evaluation Overview



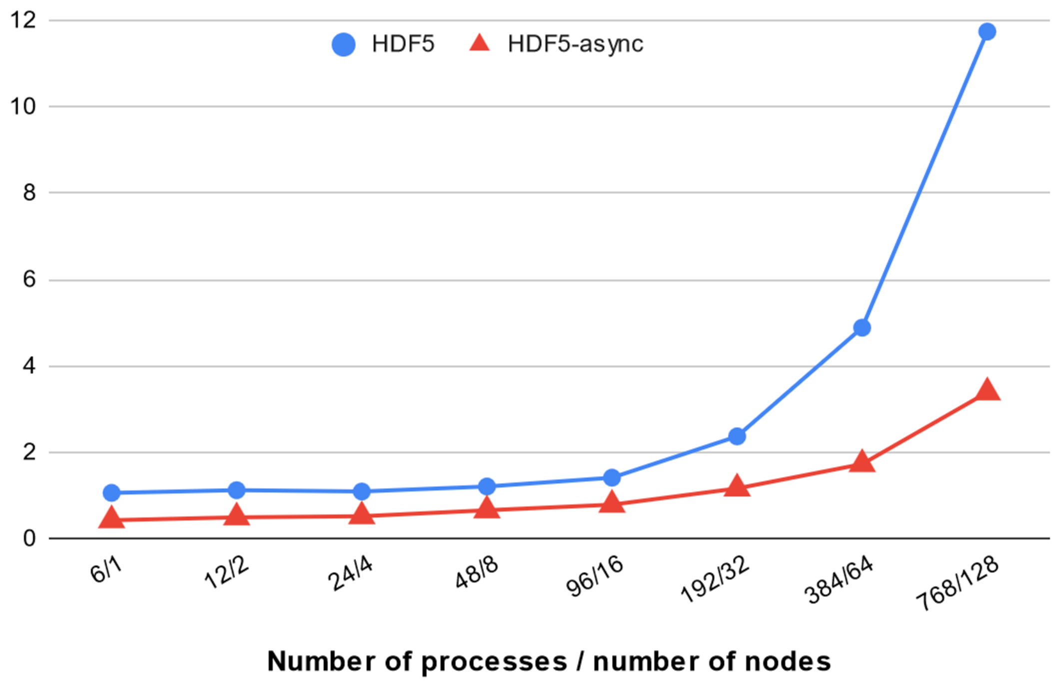
Case	Information	I/O Pattern
VPIC-IO	I/O kernel from VPIC, a plasma physics code that simulates kinetic plasma particles.	**Write**, single file for all steps, 8 variables, 256 MB per process per timestep.
BDCATS-IO	I/O kernel from BDCATS, a parallel clustering algorithm code that analyze VPIC data.	**Read**, single file, 8 variables, 256 MB per process per timestep.
AMReX/Nyx	I/O workload from Nyx, an adaptive mesh cosmological simulation code that solves equations of compressible hydrodynamics flow.	**Write**, one file for each timestep, 6 variables, *single* refinement level, with simulation metadata, 385 GB per timestep
AMReX/Castro	I/O workload from Castro, an adaptive mesh compressible radiation / MHD /hydrodynamics code for astrophysical flows.	**Write**, one file for each timestep, 6 variables, *3* refinement levels, with simulation metadata, 559 GB per timestep

Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

Speedup with VPIC-IO and BDCATS-IO on Summit



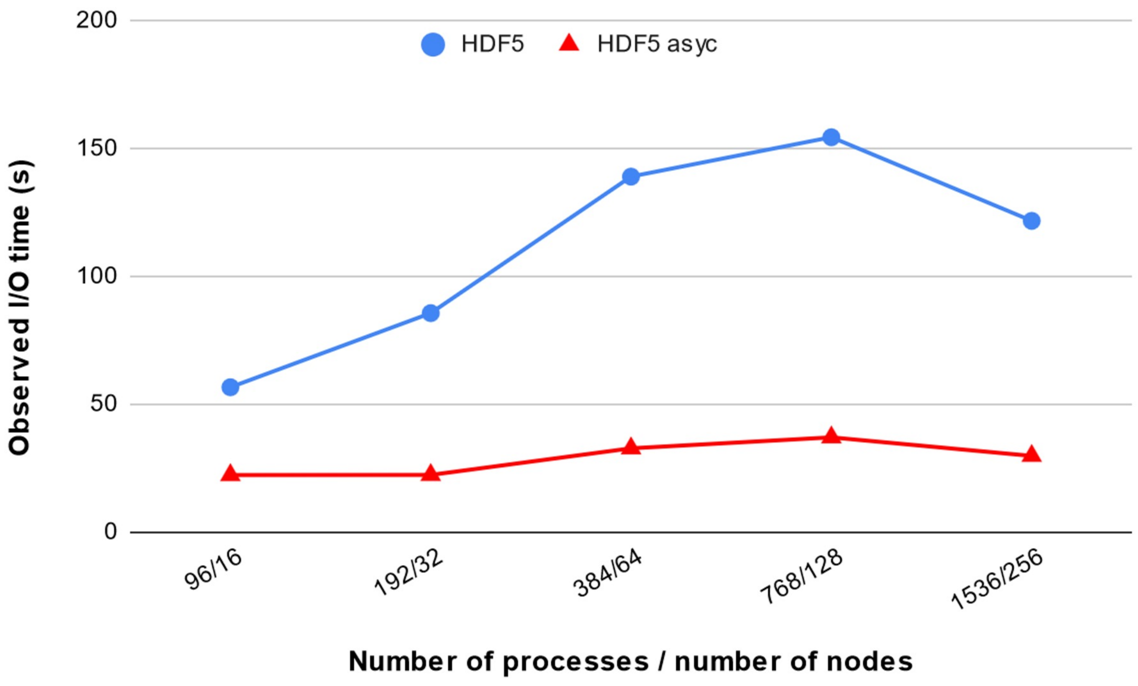
VPIC-IO, writes 256MB per process, 5 steps, emulated compute time.



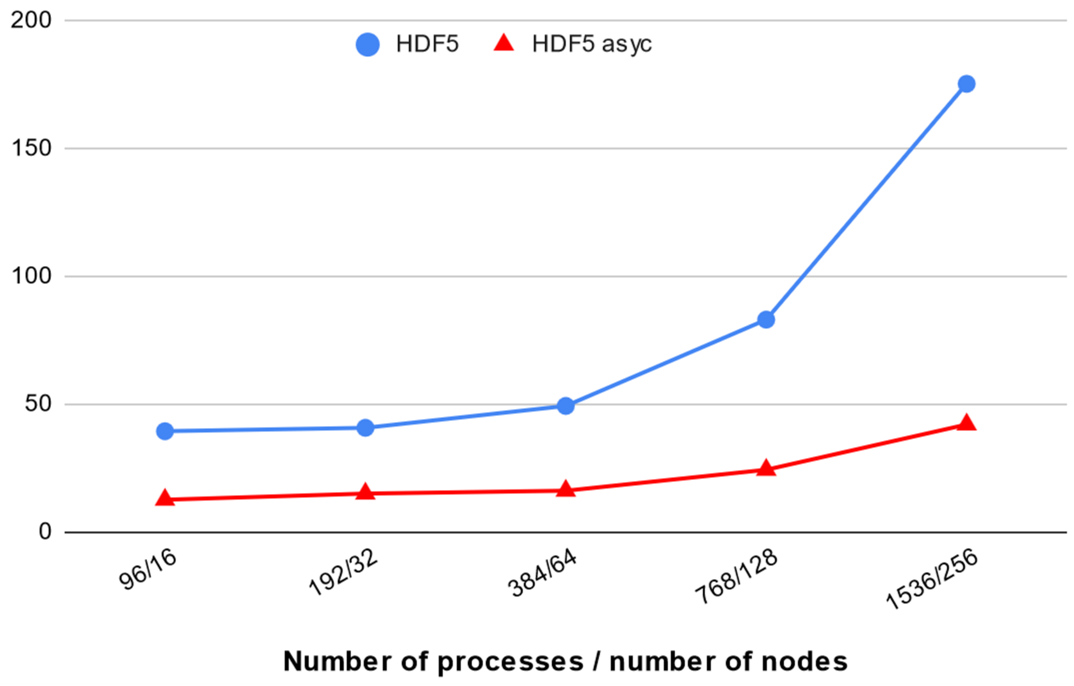
BDCATS-IO, reads 256MB per process, 5 steps, emulated compute time.

Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

Speedup with AMReX Applications on Summit



NyX workload, single refinement level,
writes 385GB x 5 steps, emulated compute time.



Castro workload, 3 refinement levels,
writes 559GB x 5 steps, emulated compute time.

Slides from Houjun Tang, ECP Annual Meeting 2022 presentation

Async I/O in Flash-X

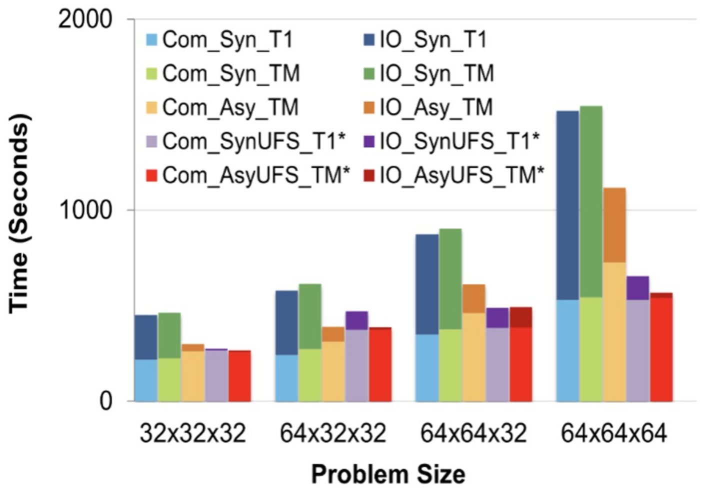
- Highly scalable multiphysics simulation code for heterogeneous compute architecture
- Supports “uniform” and “adaptive” mesh
- Primarily written in Fortran
- Component based code
- Eulerian base discretization
- AMR is used to:
  - Reduce memory footprint
  - Reduce computation
- Used for various simulations:
  - Galaxy clusters to
  - Turbulent Nuclear Burning

```
...
         /* create a parallel hdf5 dataset */
#ifdef FLASH_IO_ASYNC_HDF5
         dataset =
H5Dcreate_async(*file_identifier,
record_label_new,
                  H5T_NATIVE_DOUBLE, dataspace,
H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT, io_es_id);
#else
         dataset = H5Dcreate(*file_identifier,
record_label_new,
                  H5T_NATIVE_DOUBLE, dataspace,
H5P_DEFAULT,H5P_DEFAULT, H5P_DEFAULT);
#endif
...
```

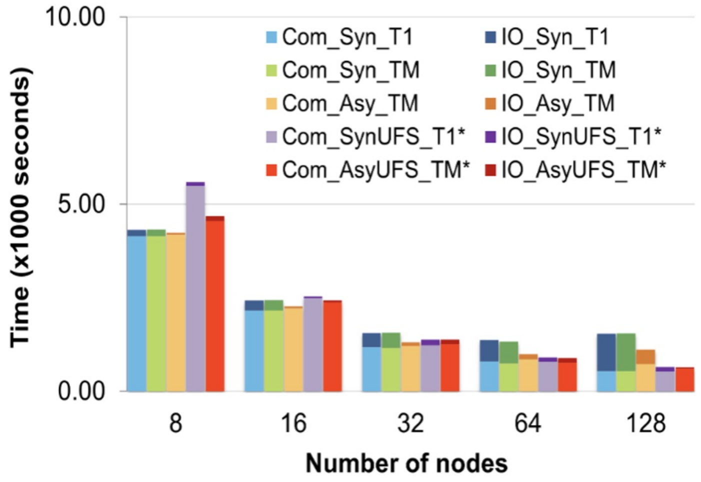

Slides from Houjun Tang and Rajeev Jain, PDSW 2022 paper presentation

Async I/O performance in Flash-X with SOD configuration

- Sod is a compressible flow explosion problem widely used for verification of shock-capturing simulation codes.
- We used a 3D Sod problem with tracer particles.
- Each runs for 109 steps, writes a checkpoint file every 33 steps, a plot file every 10 steps, and compared the total execution time with 5 different configurations that uses Synchronous and Asynchronous I/O, with and without MPI_THREAD_MULTIPLE, and using GPFS and UnifyFS.
- For cases with async, the majority of the write operations are overlapping with Flash-X's computation. Exceptions include the initial data writes and the last step as there is no computation to overlap with.



(a) Sod - weak scaling, 16 to 128 nodes



(b) Sod - strong scaling, problem size 64x64x64

Slides from Houjun Tang and Rajeev Jain, PDSW 2022 paper presentation

Results: Streaming Sine Wave

- The streaming sine wave test problem is a test problem for verifying the correctness of the streaming advection operator in thornado as well as the Flash-X interface to thornado.
- Uses GPUs and data is copied to CPU for writing
- At a higher number of nodes the interference between COM_ time and IO_ is higher as the I/O time as a whole increases: it is 27.1% for the 256-node synchronous case.

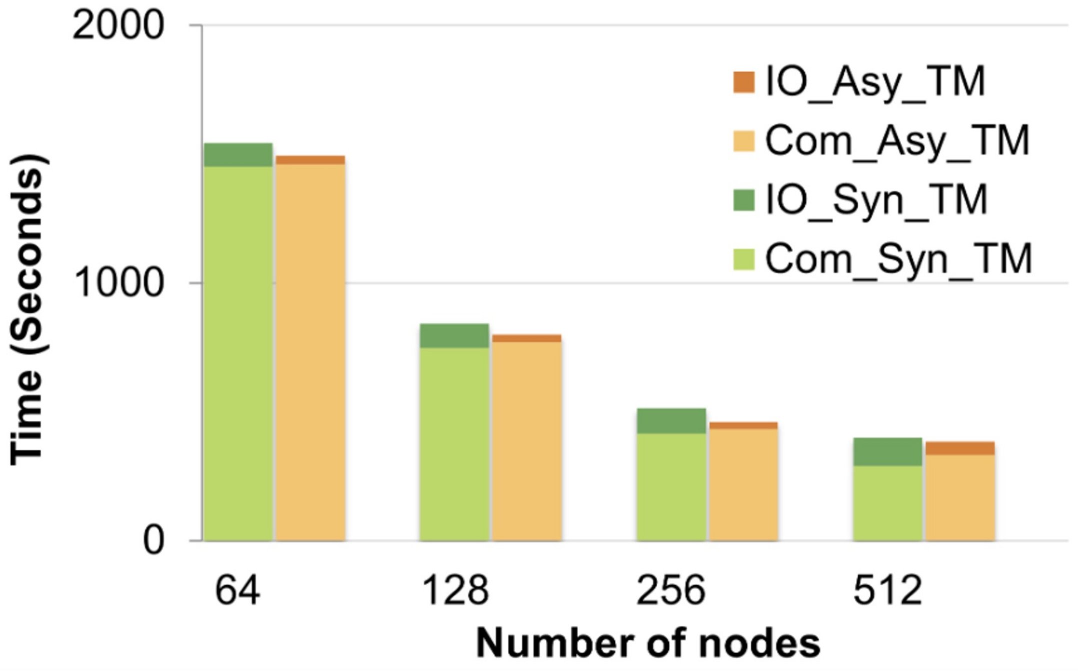


Fig. 7: Streaming sine wave - strong scaling

The total time required by synchronous I/O increases with increasing number of nodes. This is because communication is time-consuming and the GPFS file-system write operation does not scale well.

Results: Deforming Bubble Problem

- For the 64-node case - I/O time as a percentage of the total time goes down from 22.3% to 4.7%.
- For the 256-node case, the I/O time is significantly higher for the synchronous case;
- The asynchronous I/O time for 256 nodes remains the same as for other cases, but the Com_ time has increased because a greater percentage of Com_ time overlaps with IO_ time.

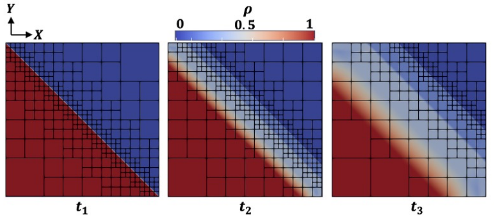


Fig. 1: Contours of energy (E) for time $t_3 > t_2 > t_1$, and an example of block structured AMR grids.

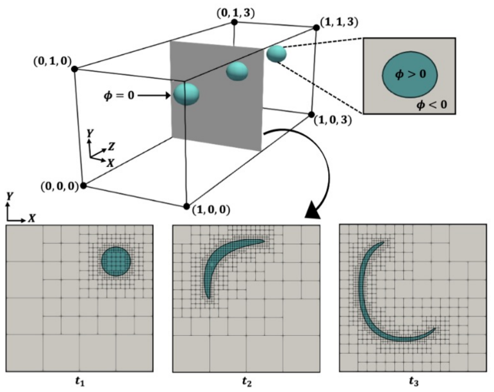


Fig. 2: Schematic of the deforming bubble problem: The bubbles are defined by using a signed distance function, ϕ, that undergoes deformation under a prescribed velocity field.

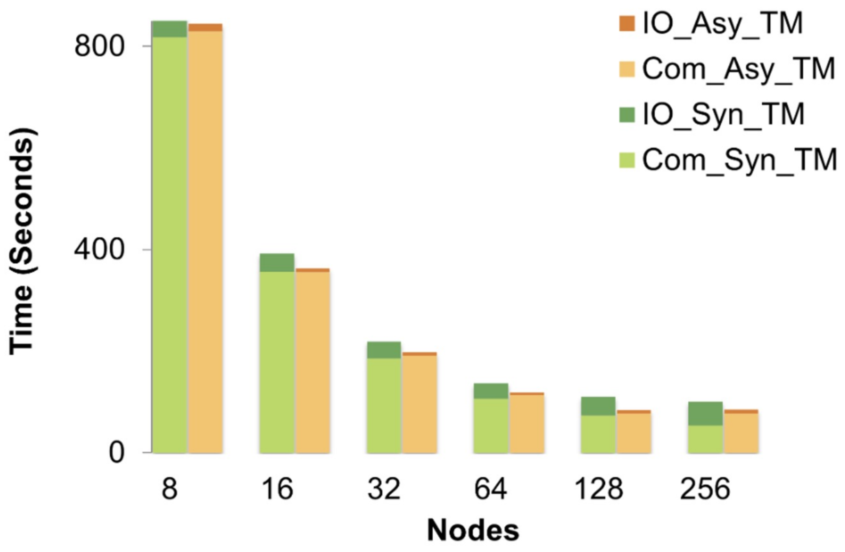


Fig. 6: Deforming bubble - strong scaling

Slides from Houjun Tang and Rajeev Jain, PDSW 2022 paper presentation

Best Practice & Lessons Learned

- Async is effective when I/O time is a significant portion of the total application execution time, and there is enough compute time to overlap with.
- Some operations cannot be done asynchronously, avoid if possible.
  - E.g. `H5Dget_space` need to perform sync I/O before returning.
    - Async debug log available for identification.
- **`MPI_THREAD_MULTIPLE`** has overhead.
  - 3-5% observed performance slowdown.
- Background thread interference.
  - Minimal interference for GPU-accelerated applications.
  - OpenMP applications should leave 1 core/thread for the async background thread.
- Memory allocation needs to be handled properly.
  - Peak memory usage could be higher than sync mode, due to double buffering.
  - Will switch to sync mode when not enough system memory is available.

Summary of today's class

- Asynchronous I/O

- Next Class –
  - More evaluation of async I/O
  - Caching and prefetching

- Class project –
  - Status update on Apr 4^{th}
  - Final presentation on Apr 20^{th}
  - Final exam on Apr 25^{th}