Accelerating Flash-X Simulations with Asynchronous I/O

Rajeev Jain*, Houjun Tang†, Akash Dhruv*, J. Austin Harris‡, Suren Byna†
* Argonne National Laboratory, † Lawrence Berkeley National Laboratory, ‡ Oak Ridge National Laboratory

Abstract—**Most high-fidelity physics simulation codes, such as Flash-X, need to save intermediate results (checkpoint files) to restart or gain insights into the evolution of the simulation. These simulation codes save such intermediate files synchronously, where computation is stalled while the data is written to storage. Depending on the problem size and computational requirements, this file write time can be a substantial portion of the total simulation time. In order to hide the I/O latency of checkpointing, asynchronous I/O methods have been introduced. These methods use background threads for performing I/O while the main threads continue with the simulation. The usage of background threads can compete for resources on the node as well as with communication. In this paper, we evaluate the overheads and the overall benefit of asynchronous I/O in HDF5 to simulations. Results from real-world high-fidelity simulations on the Summit supercomputer show that I/O operation is overlapped with application communication or computation or both, effectively hiding some or all of the I/O latency. Our evaluation shows that while using asynchronous I/O adds overhead to the application, the I/O time reduction is more significant, resulting in overall up to 1.5X performance speedup.**

I. Introduction

Most high-performance computing (HPC) multiphysics simulations use parallel optimized I/O routines to write intermediate simulation data. This data can be categorized into two broad types: plot data and checkpoint data. Plot data is used for visualization and keeping track of simulation variables as the simulation evolves. It generally uses lower precision and tracks only the variables of interest, to preserve space and compute time. Plot data is written more frequently than checkpoint data. Checkpoint data is used to restart a simulation in the event of a failure or to extend long-running simulations. Typically, checkpoint files store data for the grid and all the variables required to restart the simulation at a given point in the simulation. Compared with plot files, checkpoint files are much larger because the data is stored in full precision. Checkpoint files also consume a much bigger chunk of total I/O time but are saved less often than plot files; frequent checkpointing can lead to significant overhead on system resources. Researchers [1] studied the needs of astrophysics simulations and identified the requirements of system-level understanding of data and computation. Other physics simulation areas such as earthquake simulations [2] have implemented similar asynchronous I/O and tailored solutions specific to a hardware architecture.

Writing data to a parallel file system is expensive because of several factors such as network contention, contention of shared file system resources, and contention from I/O operations from other running simulations that share the file system. Physics-based HPC applications require careful attention in setting up the frequency of plot, checkpointing, and other I/O operations. Also important are fine-tuning the key parameters and optimizing the actual implementation of the I/O routines to accelerate the overall simulation and optimally utilize the resources provided by the HPC hardware.

HDF5, ADIOS, VeloC, and PnetCDF are popular high-level external libraries that are used by most physics simulation codes for I/O operations. These operations are often performed synchronously, without compression, and use default system configurations. Understanding the HPC architecture and resources is important but often difficult because it requires manual tuning to get optimal performance from the machine. The recent move toward many-core nodes with graphics processing units (GPUs) has paved the way to hybrid programming models and has raised new challenges for physics simulation developers. Many physics simulation codes are unable to exploit the hardware resources and spend too much time with suboptimal use of I/O routines.

Flash-X uses the HDF5 I/O library for the majority of its simulation needs. In this work we describe the implementation of our recently developed asynchronous I/O capability. Our results show how the I/O performance of the simulations is affected by use of key parameters such as the number of MPI I/O hints, hardware threads, problem distribution, and GPU-based settings. The findings presented here also pave the way for automating the allocation of resources for compute and I/O tasks. We show that asynchronous HDF5-based I/O acceleration of simulation codes enables a considerable reduction in total execution time of a variety of community-based physics simulation codes. Results show a small overhead and substantial savings in the total simulation time for a typical simulation with our asynchronous I/O implementation. We also describe the implementation of a novel mechanism to show a significant reduction in the total runtime of the simulation. This mechanism decouples compute and I/O operations of the simulations and provides the ability to continue the simulation to the next time step without waiting for the I/O operation to complete.

In Section II, we present a brief background to the simulation codes and tools used in this study, and show the code changes required for using asynchronous I/O. In Section III, we present the experimental setup of various simulation codes. We evaluate the performance overhead and improvements in Section IV and conclude in Section V.

978-1-6654-7562-4/22/$31.00 ©2022 IEEE
DOI 10.1109/PDSW56643.2022.00008

II. Background

A. Flash-X

Flash-X [3] is a highly composable multiphysics software system that can be used to simulate physical phenomena in several scientific domains. It is derived from FLASH, which has been a community code for several communities over the past 20 years. The Flash-X architecture has been redesigned to be compatible with increasingly heterogeneous hardware platforms. Part of the redesign utilizes a newly designed performance portability layer that is language agnostic. Flash-X can write checkpoint and plot files in HDF5 format, and we have recently enabled Flash-X to perform asynchronous I/O with the HDF5 asynchronous VOL connector.

B. HDF5 and Asynchronous VOL Connector

HDF5 is a popular I/O library and self-describing file format that provides an abstraction layer to manage data and the metadata within a single file [4]. HDF5 has recently provided a feature, called the Virtual Object Layer (VOL) [5], that enables HDF5 to support dynamic control to the library at runtime. VOL allows intercepting the high-level HDF5 public application programming interface and implementing various optimizations for different types of storage media, thus enabling better data management transparently to the application. The asynchronous I/O VOL connector [6], [7] provides asynchronous I/O support for HDF5 operations. It uses background threads to perform I/O operations, which are managed by the Argobots threading framework [8], which uses lightweight threads running on CPUs.

C. UnifyFS

UnifyFS [9] is a user-level file system under active development that supports shared file I/O over distributed storage on HPC systems, for example, node-local SSDs. With UnifyFS, applications can write to fast, scalable, node-local burst buffers as easily as they do to the parallel file system. We have evaluated Flash-X's I/O performance on UnifyFS using node-local SSDs in addition to the disk-based GPFS.

D. Compressible Flow Explosion Simulation – Sod

We use a 3D Sod problem (a compressible flow explosion problem widely used for verification of shock-capturing simulation codes) with tracer particles. It checks Flash-X's ability to deal with strong shocks and non-planar symmetry. The problem involves the self-similar evolution of a cylindrical or spherical blast wave from a delta-function initial pressure perturbation in an otherwise homogeneous medium. Figure 1 shows a zoomed-in view of contours of energy, E, as the simulation progresses from time t_1 to t_3. The adaptive grid follows the shock as it propagates through the domain. This problem tests the performance of AMReX refinement procedure when using asynchronous I/O.

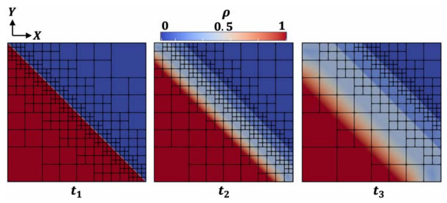


Fig. 1: Contours of energy (E) for time $t_3 > t_2 > t_1$, and an example of block structured AMR grids.

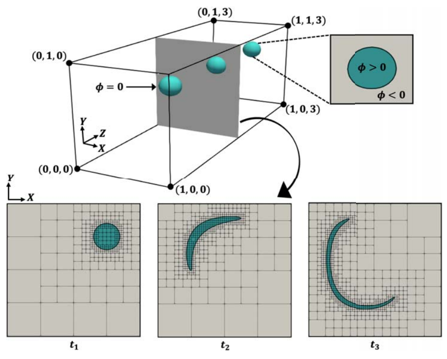


Fig. 2: Schematic of the deforming bubble problem: The bubbles are defined by using a signed distance function, ϕ, that undergoes deformation under a prescribed velocity field.

E. Deforming Bubble Simulation

The deforming bubble problem is a benchmark problem for multiphase computational fluid dynamics (CFD) applications using Flash-X. The deformation of bubbles is computed by using a level-set advection and redistancing algorithm, which is used in multiphase CFD simulations to resolve the evolution of the liquid-gas interface. [10]. A schematic of this problem is provided in Fig. 2, which shows isocontours of the signed distance function, ϕ, used to represent bubbles that undergo deformation under a prescribed velocity field. In this problem grid changes occur frequently and provide an opportunity to test I/O performance with different adaptive mesh refinement (AMR) implementations in Flash-X.

F. Streaming Sine Wave Test Problem (GPU)

We include neutrino transport in Flash-X by interfacing to `thornado`[11]. `thornado` is independently developed and maintained as a toolkit for solving neutrino radiation hydrodynamics in astrophysical simulations. For Flash-X, we exercise the capability of `thornado` to evolve the neutrino radiation field with spectral neutrino transport using a two-moment model.

Moments of the neutrino phase-space distribution function representing spectral energy and momentum densities are evolved. Higher-order moments (e.g., components of the radiation pressure tensor) are approximated by using algebraic expressions involving the lower-order evolved moments to form a closed system of equations. We configured `thornado` to use OpenACC to offload the computation to GPUs.

The *streaming sine wave* test problem is an important benchmark for verifying the correctness and performance of

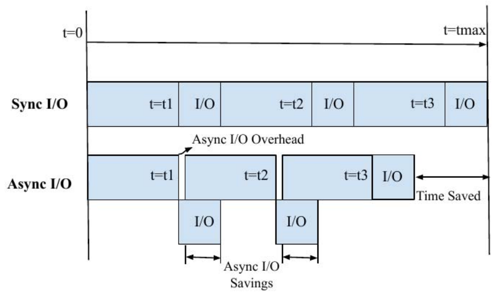


Fig. 3: Timeline of synchronous I/O and asynchronous I/O. Three time steps t1, t2, and t3 are shown. Overlap of I/O time and compute time is shown in the Async I/O case for time steps t2 and t3.

the streaming advection operator in `thornado` as well as the Flash-X interface to `thornado`. It also has a relatively high number of degrees-of-freedom per cell providing different I/O performance characteristics than the other tests. The problem also tests the correctness of asynchronous I/O for a problem evolved using GPUs.

III. Method

The asynchronous I/O VOL connector [7] enables asynchronous I/O for HDF5 operations using background threads. This implementation can be compiled as a dynamically linked library and can be linked to a user's application directly, remaining separate from the installed version of HDF5 and making it easy to adopt. The background threads are managed by Argobots, a lightweight low-level threading framework [8]. The VOL connector maintains a queue of asynchronous tasks and tracks their dependencies as a directed acyclic graph, where a task can be executed only when all its parent tasks have been completed successfully.

Figure 3 shows two simulations, Sync I/O and Async I/O, starting at the same time `t=0`. At time `t=t1`, the simulation starts to write the first checkpoint file. The Async I/O simulation starts the next time step after a very small overhead that signals the simulation that I/O operation is complete. In the background, Async I/O uses the operating system of the compute nodes hardware and requests threads for writing the file in the background. The figure shows two more time steps, `t=t2` and `t=t3`. We note that usually at the time of writing the final checkpoint file no more computation is required, and I/O is the only operation. At this point the Async I/O signals the threads to complete all pending I/O operations before stopping the simulation. In the current implementation, a copy of the data is stored in memory for asynchronous I/O operations, which could result in a peak memory requirement double the size of the synchronous application run. Async VOL has an internal memory-probing mechanism that can detect when not enough system memory is available for copying the write buffer, and it will start writing the data synchronously without a copy until more memory becomes available (as a result of freeing existing copied buffers after finishing their writes).

HPC systems typically include a parallel file system (PFS) such as GPFS and Lustre to provide high-performance I/O operations for massive parallel applications. Some systems may also have an additional burst buffer layer that utilizes SSDs to provide even higher I/O throughput. The SSDs in the burst buffer reside on each compute node and offer an order of magnitude better I/O performance than does the hard-disk-based parallel file system. Disk-based parallel file systems are usually suited for medium-sized streaming input/outputs, whereas node-local storage is suited for bursts of data accesses with exceptionally large I/O streaming requirements. For example, the Summit supercomputer uses GPFS as the global file system. A few I/O libraries such as UnifyFS [9] and Spectral [12] allow accessing data distributed in different node-local storage devices with a single namespace and can automatically transfer the data to and from GPFS. We note that node-local storage is deleted at the end of the simulation, and hence the data needs to be copied out to GPFS for evaluation.

```
#ifdef FLASH_IO_ASYNC_HDF5
  status = H5Dwrite_async(dataset, ... , buf, es_id);
#else
  status = H5Dwrite(dataset, ... , buf);
#endif
if (status < 0)
  printf("Error writing %s\n", dname);
```

Fig. 4: Example change in Flash-X to support both synchronous and asynchronous I/O with HDF5.

Figure 4 shows the conditionals for an appropriate HDF5 write to call. It calls asynchronous HDF5 when the code is set up with the asynchronous option. The extra parameter "es_id" (event set id) is used for the asynchronous version. It is obtained from a call to the `H5EScreate()` function, and it manages the asynchronous operation. The function `H5EWait()` is called after the write operation; it waits until the write is complete in the event set buffer and is ready for a new buffer as new set of output is ready to be stored in the buffer for overlapping the writing and computation/-communication. The final checkpoint file cannot be written asynchronously because it is the last step in the process and no more calculations are left to overlap.

Using the async I/O VOL requires having `MPI_THREAD_MULTIPLE` support when initializing MPI, because the background thread may execute I/O operations that involve MPI collective operations at the same time as the application's MPI communications. Enabling it increases the MPI overhead (on average $\approx 4\%$), as shown in Section IV. Additionally, since the background thread would perform I/O operations at the same time that the application is doing computation or communication, they may compete for the CPU resources. If the CPU cores become oversubscribed, the application's progress can be a significantly slowed down. Thus, in order to avoid resource contention, it is recommended to leave one core or one hyperthread for the asynchronous background thread. This can often be achieved with GPU-accelerated applications, where the computation and communication could all happen on the GPUs, leaving the CPU resources idle, and can fully benefit from asynchronous I/O with a small overhead.

IV. Performance Evaluation

A. Experiment Setup

We ran three Flash-X simulations at various scales on the Summit supercomputer at the Oak Ridge Leadership Computing Facility. Each Summit node comprises 2 IBM POWER9 CPUs and 6 NVIDIA Volta 100 GPUs and has 608 GB of fast memory (96 GB HBM2 + 512 GB DDR4) along with 1.6 TB of non-volatile memory (node-local SSD). The nodes are connected with the dual-rail Mellanox EDR InfiniBand network and have access to a 250 PB IBM file system.

We have evaluated three Flash-X cases: Sod, deforming bubble problem, and streaming sine wave (SSW) problems. The former two use only the CPU, while the SSW case uses both the CPU and GPU. All three problems use less than half of available memory on Summit nodes, with sufficient space for keeping the duplicated buffer used by asynchronous write. Each MPI rank used 1 Argobots thread for executing the I/O operations asynchronously.

Inter-node communication in Flash-X is usually latency-bound and does not interfere significantly with overlapping I/O. For problems with sufficiently large degrees of freedom to make inter-node communication bandwidth bound and cause contention for network resources (e.g., Section II-F) there is a concomitant increase in node-local computational load where I/O can be performed without the presence of inter-node communication. For problems large enough to break weak-scaling ($\gtrsim$10000 MPI ranks), this issue of resource contention will need to be revisited, but is beyond the proof-of-concept scope we present here.

B. Sod

The Sod problem in Flash-X is often used to test and benchmark the performance of various physics simulation modules. The setup of Sod problem in the experiments performed here uses the AMReX library for mesh representation. Figure 5 shows the weak- and strong-scaling results running the Sod case using 16 to 128 nodes with various configurations: (1) synchronous I/O with `MPI_THREAD_SINGLE` to GPFS, (2) synchronous I/O with `MPI_THREAD_MULTIPLE` to GPFS, (3) asynchronous I/O with `MPI_THREAD_MULTIPLE` to GPFS, (4) synchronous I/O with `MPI_THREAD_SINGLE` to UnifyFS, and (5) asynchronous I/O with `MPI_THREAD_SINGLE` to UnifyFS. At the time of running the experiments, UnifyFS had a bug that prevented moving the data from node-local SSDs to the global GPFS file system; thus the total times recorded for cases (4) and (5) are not directly comparable with those for cases (1) to (3), since they are writing to different storage devices.

For the weak-scaling cases, comparing configurations (1) and (2) shows an insignificant time increase when using `MPI_THREAD_MULTIPLE` (less than 5%). Comparing (1) and (3) demonstrates the effectiveness of the asynchronous I/O: the total application runtime is reduced by up to 35%, and the I/O time observed by the application is reduced by up to 85%. Configurations (4) and (5) write data to UnifyFS, which stores the data to the node-local SSDs. As a result of the faster I/O, the total execution time is further reduced. However, because of the need to reserve at least one core for UnifyFS, the total number of MPI ranks used by the Flash-X application is reduced from 42 to 40 (1 core is mapped to 2 MPI ranks with Intel's simultaneous multithreading support), thus increasing the total compute time. Using asynchronous I/O on top of UnifyFS can further speed up the application's runtime but not as significantly as when data is written to the slower GPFS. We also observe that the computation time of synchronous I/O is larger than the asynchronous I/O with UnifyFS in 8 and 16 nodes cases. We believe it is caused by the UnifyFS's RDMA data transfers from the FLASH-X processes vs. the background threads to the UnifyFS server processes. With relatively few UnifyFS processes (1 process per node) to write ≈ 1 TB data in total, it causes more interference to the application processes than the MPI TM overhead.

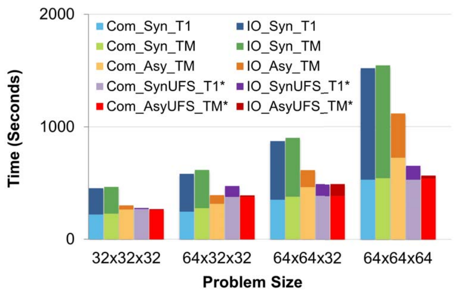
We observe a similar trend with the strong-scaling cases. However, we found that the smaller number of MPI ranks/-cores used by the Flash-X application has more significant impact, since it takes longer time in the 8- and 16-node cases. With 8 to 128 nodes, the background threads' I/O time overlap with 4%, 12%, 31%, 59%, and 90% of the computation and communication time. As mentioned previously, Flash-X performs MPI communications in every time step, which overlap with the I/O operations. Such overlap results in a small overhead, and is insignificant compared with the total I/O time reduction from enabling asynchronous I/O in all our test cases.

C. Deforming Bubble

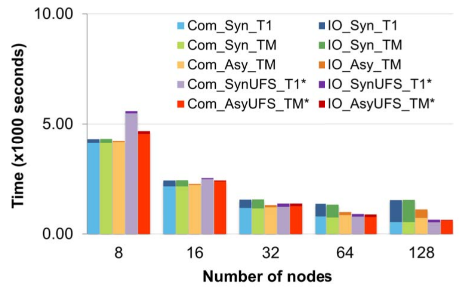
For each run in the strong-scaling study the number of bubbles per MPI process is varied. Results for strong scaling of the deforming bubble problem are shown in Fig. 6. Each bar shows compute/communication (lighter color, marked as *Com_*) and I/O time (darker color, marked as *IO_*). One can see that while keeping the problem size fixed, using more nodes results in less total simulation time. For all the cases the time taken by synchronous I/O is greater than that of asynchronous I/O; this is due to the savings provided by using asynchronous I/O. The checkpoint file is written in the background by using threading while communication and computation for the simulation continue. For the 64-node case the I/O time as a percentage of the total simulation 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; this is due to the fact that a lot of communication is required to write the file to disk from 256 nodes (or 5,376 MPI ranks) and the GPFS file system on *Summit* does not scale well. 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.

D. Streaming Sine Wave

We set up a Cartesian grid `+cartesian` in three dimensions `-3d` and set the number of cells per block to 16 in each dimension `-nxb=16 -nyb=16 -nzb=16`. The



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



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

Fig. 5: Weak and strong scaling for the Sod problem, each node with 42 MPI ranks. Com: compute and communication time (with light color), IO: I/O time (with dark color), Syn: synchronous I/O, Asy: asynchronous I/O, SynUFS: synchronous with UnifyFS, AsyUFS: asynchronous with UnifyFS, T1: MPI with single thread, TM: MPI with thread multiple support. The two UnifyFS cases (annotated with * at the end) are writing data to node-local SSD; others are writing to the disk-based GPFS.

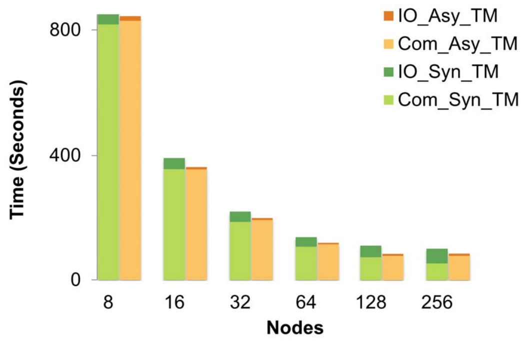


Fig. 6: Deforming bubble - strong scaling

neutrino degrees of freedom are set by choosing an energy grid of 16 elements `nE=16`, two neutrino species ν_e and $\bar{\nu}_e$ `nSpecies=2`, four moments of the neutrino distribution that include the zeroth moment and three components of the first moment `nMoments=4`, and a second-order phase space discretization `nNodes=2`. This sets the data size per block to be stored and written in the checkpoint files as $NXB \times NYB \times NZB \times nE \times nSpecies \times nMoments \times nNodes \times 8\ bytes/block = 8.4MB/block$. The total size required for the neutrino data can then be calculated by using the total number of blocks, which is set in the `flash.par` file via `nblockx`, `nblocky`, and `nblockz`. We set the closure prescription for higher-order moments to the Minerbo method with `momentClosure=MINERBO` and choose the

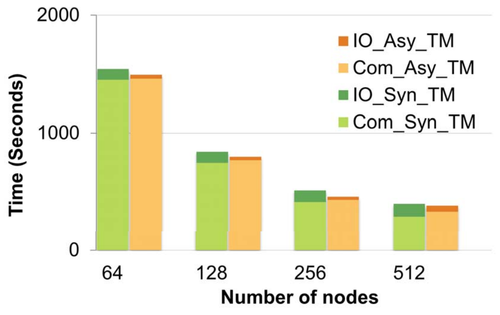


Fig. 7: Streaming sine wave - strong scaling

non-relativistic solver with `thornadoOrder=ORDER_1`. The OpenACC directives are enabled with `+thornadoACC`.

Unlike the Sod and deforming bubble problems, the streaming sine wave problem uses GPUs, using all available compute resources: GPU and CPU (threading). Allocating resources and GPUs is critical to the performance and overall simulation time. For the results presented here, we use one GPU per MPI rank, and the data is copied from GPU to CPU memory automatically by FLASH-X before written out, which takes an insignificant amount of time compared to file I/O operations. Figure 7 shows the strong-scaling results for synchronous and asynchronous I/O cases going from 64 nodes to 512 nodes. The total time required by synchronous I/O increases with increasing number of nodes. This is due to the fact that communication is time-consuming and the GPFS file-system write operation does not scale well. For the 256-node case, we see that total I/O time required is 19.1% of the total simulation time for the synchronous I/O, and it goes down to 6.2% for the asynchronous case. 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. Despite the increase in COM_ time the asynchronous I/O results in significant savings in the total time required by the simulations.

V. Conclusion

In this work we present a performance evaluation of various problems from Flash-X that show significant performance gains by enabling asynchronous I/O. Heterogeneous applications utilizing MPI threads and GPUs are carefully chosen and set up to understand the limitations and advantages of the proposed method.For all the problems, we find the total simulation time for asynchronous I/O is lower than that for the synchronous case. The Flash-X code main branch already supports this feature, and it can be invoked by simply adding the *+hdf5AsyncIO* setup option in the setup command. We study three problems: Sod uses AMReX for mesh refinement and communication, deforming bubble uses Paramesh and only MPI (no threads), and streaming sine wave uses also GPUs for computations. For all the problems the use of asynchronous I/O causes a reduction in the total simulation time. In the future, we want to add compression to the checkpoint files written asynchronously and study the performance.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

References

[1] M. Hereld, J. A. Insley, E. C. Olson, M. E. Papka, T. D. Uram, and V. Vishwanath, "Modeling resource-coupled computations," in *Proceedings of the 2009 Workshop on Ultrascale Visualization*, 2009, pp. 27–33.

[2] C. Uphoff, S. Rettenberger, M. Bader, E. H. Madden, T. Ulrich, S. Wollherr, and A.-A. Gabriel, "Extreme scale multi-physics simulations of the Tsunamigenic 2004 Sumatra earthquake," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126948

[3] A. Dubey, K. Weide, J. O'Neal, A. Dhruv, S. Couch, J. A. Harris, T. Klosterman, R. Jain, J. Rudi, B. Messer *et al.*, "Flash-x: A multiphysics simulation software instrument," *SoftwareX*, vol. 19, p. 101168, 2022.

[4] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *EDBT/ICDT*, 2011, pp. 36–47.

[5] S. Byna, M. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "ExaHDF5: Delivering efficient parallel I/O on exascale computing systems," *JCST*, vol. 35, pp. 145–160, 2020.

[6] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling transparent asynchronous I/O using background threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2019, pp. 11–19.

[7] H. Tang, Q. Koziol, S. Byna, and J. Ravi, "Transparent asynchronous parallel I/O using background threads," *IEEE TPDS*, p. 1, 2021.

[8] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE JPDS*, 2017.

[9] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror, D. Ivanov, T. Wang, C. P. Steffen, and U. N. N. S. Administration, "UnifyFS: A distributed burst buffer file system - 0.1.0," 10 2017. [Online]. Available: https://www.osti.gov//servlets/purl/1408515

[10] A. Dhruv, E. Balaras, A. Riaz, and J. Kim, "An investigation of the gravity effects on pool boiling heat transfer via high-fidelity simulations," *International Journal of Heat and Mass Transfer*, vol. 180, p. 121826, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0017931021009315

[11] "Thornado," https://github.com/endeve/thornado.

[12] OLCF, "Spectral library," https://www.olcf.ornl.gov/spectral-library/.

Appendix

A. HDF5 Async VOL connector Setup

```
export HDF5_PLUGIN_PATH="<path>/vol-async/src"
export HDF5_VOL_CONNECTOR="async under_vol=0;under_info={}"
export ABT_THREAD_STACKSIZE=100000
export HDF5_ASYNC_EXE_FCLOSE=1
```

B. UnifyFS Setup

```
module use /sw/summit/unifyfs/modulefiles
module load unifyfs/1.0-beta/mpi-mount-gcc9
export UNIFYFS_LOGIO_SPILL_DIR=/mnt/ssd/$USER/data
export UNIFYFS_LOG_DIR=$JOBSCRATCH/logs
export share_dir=/gpfs/alpine/$PROJ/scratch/$USER/jobs/
unifyfs start --share-dir=$share_dir
```

C. MPI-IO Hints

We set the MPI-IO hints (using the "ROMIO_HINTS" environment variable) to substantially reduce the total time to write the HDF5 file. ROMIO_HINTS directs the use of optimized MPI directives for writing the file in much bigger chunks. Using these, one can reduce the total I/O time by a factor of 100. Below is an example setup for using 128 Summit nodes.

```
romio_cb_write = enable
romio_ds_write = disable
romio_cb_read  = enable
cb_buffer_size = 16777216
cb_nodes       = 128
cb_config_list = *:1
```

D. Flash-X Setup

We used the following Flash-X setup commands for the three sets of experiments in our paper:

```
# Sod
./setup Sod -auto -3d +hdf5async +cube16 Bittree=True +amrex +hdf5AsyncIO

# Deforming Bubble
./setup incompFlow/DeformingBubble -auto -3d -nxb=16 -nyb=16 -nzb=16 +amrex --objdir=df1 +parallelIO +hdf5asyncio -
    makefile=gcc

#Streaming Sine Wave
./setup StreamingSineWave -auto -3d +cartesian -nxb=16 -nyb=16 -nzb=16 nE=16 nSpecies=2 nNodes=2 nMoments=4 momentClosure=
    MINERBO -parfile=test_paramesh_3d.par +amrex +thornadoACC thornadoOrder=ORDER_1
```