# Emulating I/O Behavior in Scientific Workflows on High Performance Computing Systems

Fahim Chowdhury† Yue Zhu† Francesco Di Natale‡ Adam Moody‡
Elsa Gonsiorowski‡ Kathryn Mohror‡ Weikuan Yu†
†Florida State University ‡Lawrence Livermore National Laboratory
{fchowdhu, yzhu, yuw}@cs.fsu.edu {dinatale3, moody20, gonsiorowski1, mohror1}@llnl.gov

***Abstract*—Scientific application workflows leverage the capabilities of cutting-edge high-performance computing (HPC) facilities to enable complex applications for academia, research, and industry communities. Data transfer and I/O dependency among different modules of modern HPC workflows can increase the complexity and hamper the overall performance of workflows. Understanding this complexity due to data-dependency and dataflow is an essential prerequisite for developing optimization strategies to improve I/O performance and, eventually, the entire workflow. In this paper, we discuss dataflow patterns for workflow applications on HPC systems. As existing I/O benchmarking tools lack in identifying and representing the dataflow in modern HPC workflows, we have implemented *Wemul*, an open-source workflow I/O emulation framework, to mimic different types of I/O behavior demonstrated by common and complex HPC application workflows for deeper analysis. We elaborate on the features and usage of *Wemul*, demonstrate its application to HPC workflows, and discuss the insights from the performance analysis results on Lassen supercomputing cluster at Lawrence Livermore National Laboratory (LLNL).**

## I. Introduction

The leadership HPC supercomputers continuously run thousands of scientific application workflows every day [29]. In most cases, these workflows are executed to answer important inter-disciplinary research questions in astronomy, environmental science, medical studies, etc., by multiple data-dependent applications. The workflow applications can consist of thousands or even millions of dependent or independent tasks [21]. These workflows can generate or transfer terabytes to even petabytes of data per science campaign [13], [22]. Complex data-dependency among different modules of the workflows and transfer of a humongous volume of data can create a severe bottleneck and hinder the research and development in critical scientific studies. Holistic perception of the dataflow present in a workflow is an indispensable prerequisite to the eventual optimization of the I/O behavior and improvement of the workflow runtime.

At present, there are numerous benchmarking tools for evaluating the storage systems on HPC facilities [30], [8], [5]. Besides, popular profiling tools, e.g., Darshan [3], expose internal I/O patterns in HPC applications. While the existing benchmarking tools provide parameterized methods to replicate a real application I/O workload tentatively; there is not much emphasis on the data-dependency and dataflow related complexity posed by HPC workflows. Oftentimes, it is difficult to study dataflow issues in real-world HPC workflows. Access to actual workflow source code due to proprietary reasons or tight coupling of a workflow with specific supercomputing infrastructure are some mentionable reasons behind this challenge.

Taking these challenges into account, we develop *Wemul*, a workflow I/O emulation framework that can generate I/O workloads like conventional benchmarking tools through user-defined parameters and mimic complex dataflow with or without cycles represented by graphs. This framework provides users with a generic interface to flexibly replicate both complex and straightforward HPC workflow workloads. Moreover, it can push the workloads to real systems and help the storage system researchers expose HPC storage systems' capabilities or limitations in handling dataflow challenges through systematic characterization and performance analysis. All in all, it focuses on lessening the semantic gap between existing synthetic and real application benchmarks for deep comprehension of the HPC workflow I/O behavior.

In this paper, we discuss three simple I/O workloads in HPC workflows, i.e., deep learning (DL) training, producer-consumer, and checkpoint/restart I/O, and emulate those for further analysis. We demonstrate a technique of representing complex workflows as graphs and feeding to the emulation framework for performance analysis and real systems' evaluation. To precisely examine the I/O behavior and challenges in HPC workflows, we make the following contributions in this work:

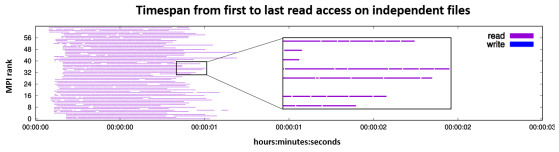
- We present our study on HPC workflow I/O challenges and workloads.
- We develop *Wemul* to generate different types of HPC workloads and discuss its functionalities.
- We run a performance analysis of Lassen's storage system by I/O benchmarking via *Wemul*'s features.

## II. Understanding I/O in HPC Workflows

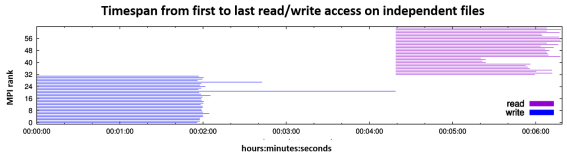
### A. HPC Workflow I/O Workloads

*1) Workflows with Simple Data-dependency:*

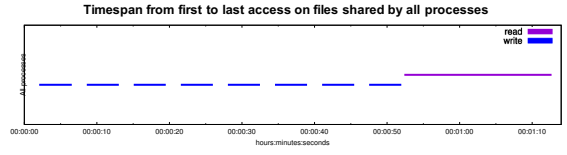
*a) Deep Learning Training I/O:* In DL application training with data-parallel setup, a dataset, typically kept on parallel file systems (PFS), is randomly shuffled and distributed among the processes of an application to import at the beginning of each epoch. The data units or files in the dataset are usually tiny (i.e., 100KB) in size, which creates huge metadata




(a) DL training




(b) Inter-node producer-consumer




(c) Checkpoint/restart

Fig. 1: Simple dataflow emulation timelines on GPFS

overhead [19]. In Fig. 1(a), we demonstrate how *Wemul* generates a workload similar to a typical DL training I/O pattern, where each process is assigned multiple files to read from a small dataset of 320 1 MiB files on a General Parallel File System (GPFS) mount point.

*b) Producer-Consumer I/O:* Simulation and analysis workflows or experimental and observational data analysis workflows often create producer-consumer relationships among the workflow's applications or tasks [29]. When the producer and consumer tasks are assigned to the same node to create an intra-node data-dependency, node-local fast storage systems like burst buffers can be used to transfer data. On the contrary, inter-node producer-consumer cannot take advantage of the client-side caching in PFS or the on-node storages. Besides, mutually interacting applications can engender a chain of I/O requests. Fig. 1(b) depicts an emulation of inter-node producer-consumer dataflow. This experiment with *Wemul* on GPFS demonstrates a gap between write and read operations that can slow down the entire workflow runtime.

*c) Checkpoint/Restart I/O:* Checkpointing is one of the most common I/O workloads posed by HPC workflows. It mainly helps with fault tolerance [29], [27], [17]. In a typical case, one process or a set of processes is assigned to create checkpoints in a user-defined frequency. In the latest supercomputers, the checkpoint files can be staged on node-local or shared burst buffer made of fast persistent storage devices. These data can be flushed to PFS asynchronously according to user specification. In the case of any process crash, all the processes restart by loading data from the latest checkpoint file and restoring the application to the last stable state. As shown in Fig. 1(c), one process is assigned to write checkpoint files to GPFS in a loop. Later, *Wemul* randomly breaks the checkpointing loop with a user-defined error rate to emulate a crash. Finally, all the processes emulate a restart by searching for the latest checkpoint file and reading it.

*2) Workflow with Complex Data-dependency:* Unlike the straightforward well-known I/O behaviors previously discussed, the modern HPC workflows can have much complexity in the dataflow pattern. One example of a complicated workflow is the one for the cancer moonshot pilot 2 (CMP2) project run by a Multi-scale Machine-learned Modeling Infrastructure (MuMMI) [22] on Sierra [11] at LLNL. This project aims to improve cancer diagnosis by leveraging HPC systems. It simulates the RAS protein and cell membrane interaction for early-stage cancer cell detection. As shown in Fig. 2, this

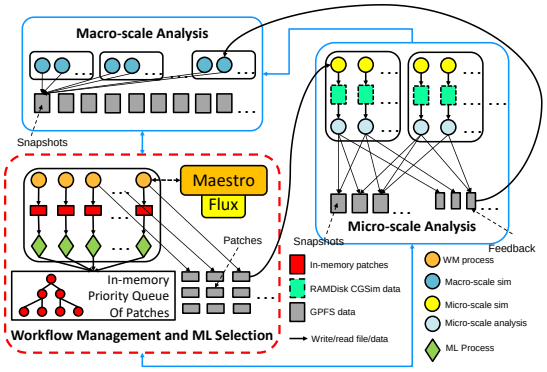



Fig. 2: Dataflow in CMP2 HPC implementation

workflow has three basic steps. *Firstly*, it writes snapshot data to GPFS via the macro-scale analysis application. *Secondly*, the machine learning module starts when a snapshot file is ready and creates a priority queue of patches. These patch files are written to the PFS again and treated as input to the further coarse-grained (CG) particle analysis. *Finally*, in the CG setup stage, the input patches go through preprocessing and pass to the CG simulation step, i.e., micro-scale sim. Hence, the CMP2 workflow's dataflow creates a chain of data transfer. Later, the CG simulation output is written to RAMDisk on Sierra [11] and fed to CG analysis, i.e., micro-scale analysis. CG analysis generates the final output snapshot and sends feedback data to the macro-scale simulation application.

It is challenging to mimic and analyze this type of complex workflow behavior using existing benchmarking tools that lack the provision of specifying data-dependency among the workflow modules. Besides, running the entire workflow like this is tightly bound to specific supercomputers [11], [12]. These situations motivate us to design a workflow I/O emulation framework for an in-depth study of HPC workflows with complex dataflow.

## III. Emulating HPC Workflow I/O

### A. Wemul: A Workflow I/O Emulation Framework

*1) Software Architecture:* *Wemul* is an open-source[1] MPI-enabled C++ framework for emulating HPC workflow I/O

[1]https://github.com/LLNL/Wemul

workloads with the provision for specifying data-dependencies among workflow applications and tasks. Currently, it has five execution modes, i.e., DL training, producer-consumer I/O, checkpoint/restart, app-based, and dag-based I/O workloads. As shown in Fig. 3, *emulator* is the entry point that exposes the functionalities of the framework to the users. The parameter values are recorded in the *config_attributes* module. The *dataflow_emulator*, derived from a generic *workflow_emulator*, is the factory for creating different types of I/O workloads according to the user-defined configuration. Besides, *app_workload*, *dag_workload*, *deep_learning*, etc., implement the base *dataflow_workload* class. At present, Interleaved-Or-Random (IOR) [7] can be optionally utilized from *deep_learning* module through *ior_runner* class. The extensible design of *Wemul* allows more robust usage of IOR in the future to generate workload with a finer granularity of I/O configuration parameters. Moreover, Asynchronous Transfer Library (AXL) [1] can be imported as a library from the *checkpoint_restart* module for staging the data files in and out according to user parameters.

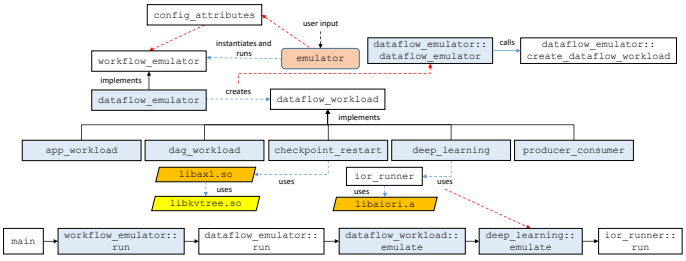
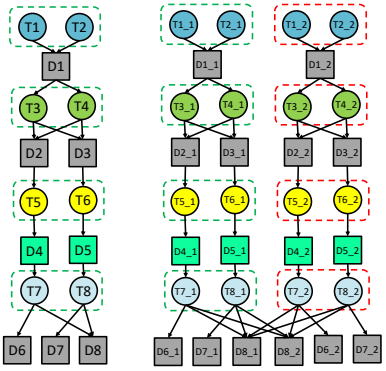



Fig. 3: Class diagram of *Wemul*

*2) Emulating Complex Workflow Data-dependencies:* *Wemul* provides an interface to describe an entire workflow in two types of strategies using a workflow specification file format inspired by DAGMan [20]. *Firstly*, users can define applications with a unique application ID, name, number of processes, and estimated wall time. *Secondly*, the user can go into a finer granularity and define the tasks, where multiple tasks can represent each application of the workflow. In both cases, the user can specify data units, typically files, present in the workflow with its name and size. Finally, a section in the file expresses strict or non-strict parent-child relationships between applications or tasks and data units. If the relationship is strict, a child task or application cannot proceed until a parent data is ready to be consumed. This task-data or application-data associativity specification can expose dataflow in cyclic or acyclic workflows as a generic graph data structure. If the dataflow creates a cycle, *Wemul* extracts the directed acyclic graph (DAG) from the original graph and runs the DAG multiple times. If a cycle does not have any non-strict relationship, *Wemul* throws an error as the workflow can create a deadlock.

Fig. 4 depicts how a small scale adaptation of the CMP2 workflow can appear after the DAG extraction in *Wemul*. In a single node setup with eight processes shown in Fig. 4(a),




(a) 1 node 8 processes emulation

(b) 2 nodes 16 processes emulation

Fig. 4: Emulating dataflow in CMP2

we represent T1 and T2 as macro-scale analysis tasks, T3 and T4 as machine learning application, T5 and T6 as micro-scale simulation, and T7 and T8 as micro-scale analysis tasks. The Dx vertices represent the data units associated with the tasks. An incoming edge to a task vertex denotes data read, and an outgoing one represents data write. The workflow looks like Fig. 4(b) when scaled up to two nodes.

*3) Execution Modes:* *Wemul* has five execution modes.

1) DL training: *Wemul* can be fed with a dataset directory full of files that are traversed, assigned equally to each process, and read in parallel.
2) Producer-consumer: there can be two types of producer-consumer workloads, i.e., intra-node and inter-node. Besides, the framework can be run as a producer-only or consumer-only application.
3) Checkpoint/restart: user can assign one or more processes to perform periodic checkpointing and random crash according to an error rate. Later, the framework searches the latest checkpoint file and reads it from all the processes on emulated restart.
4) Application-based: *Wemul* runs as a standalone application in this mode. Users can set the lists of files and mount points to read or write, block size and segment count, and access patterns through parameters.
5) DAG-based: user can set a path to a graph representation file of the entire workflow as a parameter. *Wemul* extracts the DAG and mimics the workflow.

*4) Functionality and Usage:* *Wemul* takes the information about a workload execution from the user via command-line parameters. The parameters are categorized into six basic classes. Firstly, the "General" category has the parameters related to the initialization of the framework, i.e., *type* of emulation (kept for future extension), *subtype* to specify an execution mode, and *input directory* to specify storage system mount point. Besides, there are some generic I/O pattern-related parameters like *block size* and *segment count* of the I/O requests in a workload. The rest of the categories are execution mode-specific. For instance, the "Application-based" category has parameters to set the file and mount point lists

| Category | Parameter | Description |
|---|---|---|
| General | $--type < type\_name >$ | Type of the emulation, i.e., data |
| | $--subtype < subtype\_name >$ | Subtype of dataflow emulation, i.e., app, cr, dag, dl, pc |
| | $--input\_dir < path >$ | Mount point or path to storage system to use |
| | $--block\_size < sizeinbytes >$ | Block size per read or write request |
| | $--segment\_count < number >$ | Total number of blocks or segments, i.e., filesize = blocksize x #(segments) |
| DL training | $--use\_ior$ | Enable using IOR as a library |
| | $--num\_epochs < number >$ | Number of epochs in the DL training experiment |
| | $--comp\_time\_per\_epoch < timeinseconds >$ | Computation emulation per epoch |
| Producer-consumer | $--inter\_node$ | Enable placing producer and consumer processes on different nodes |
| | $--producer\_only$ | Enable running *Wemul* as a standalone producer application |
| | $--consumer\_only$ | Enabling running *Wemul* as a standalone consumer application |
| | $--ranks\_per\_node < number >$ | Feed ranks per node number to help intra- or inter-node data transfer |
| Checkpoint/restart | $--num\_ck\_ranks < number >$ | Number of checkpoint file writer ranks |
| | $--num\_ck\_files\_per\_rank < number >$ | Number of checkpoint files to write by each rank |
| | $--checkpointing\_interval < timeinseconds >$ | Interval between two checkpointing |
| | $--ck\_error\_rate < percentagevalue >$ | Error rate at which application crash emulation occurs |
| | $--num\_ck\_iter < number >$ | Maximum iteration count for the checkpointing |
| Application-based | $--read\_input\_dirs < dir1 : dir2 : .. >$ | Colon separated list of mount points to storage systems for reading |
| | $--read\_filenames < file1 : file2 : .. >$ | Colon separated list of files to be read |
| | $--read\_block\_size < sizeinbytes >$ | Block size for the files to be read |
| | $--read\_segment\_count < number >$ | Segment count for the files to be read |
| | $--file\_per\_process\_read$ | Enable file-per-process read (shared read by default) |
| | $--write\_input\_dirs < dir1 : dir2 : .. >$ | Colon separated list of mount points to storage systems for writing |
| | $--write\_filenames < file1 : file2 : .. >$ | Colon separated list of files to be written |
| | $--write\_block\_size < sizeinbytes >$ | Block size for the files to be written |
| | $--write\_segment\_count < number >$ | Segment count for the files to be written |
| | $--file\_per\_process\_write$ | Enable file-per-process write (shared write by default) |
| DAG-based | $--dag\_file < filepath >$ | Path to the file with the graph representation of workflow |

TABLE I: Important user parameters of *Wemul*

for reading and writing, and file access patterns. "DAG-based" category has the parameter that takes the path to the file with graph representation of the entire workflow. Some mentionable parameters for "DL training", "Producer-consumer", and "Checkpoint/restart" are the number of epochs, inter-node enabler, and the number of checkpointing ranks, respectively. Table I shows the usage of some important parameters in more detail.

## IV. Experimental Results

### A. Testbed and Workload

We run all the experiments on Lassen [9], a 795 nodes IBM Power9 supercomputer with 44 cores and 256 GB memory per node situated at LLNL. Besides, it has a 24 PB IBM's Spectrum Scale GPFS, burst buffer with 1.6 TB NVMe PCIe SSD on each node, and node-local RAMDisk. We run all the five execution modes with Darshan-3.1.7 [3] and report the read/write bandwidth and latency for an increasing number of nodes from 1 to 16 with eight processes per node. We run each experiment five times and take the mean and standard error of the bandwidth and latency values. During a prior experiment, using the IOR benchmarking tool with sequential file-per-process I/O on GPFS, we got $\sim$186 GiB/s read, and $\sim$190 GiB/s write bandwidth for 16 nodes.

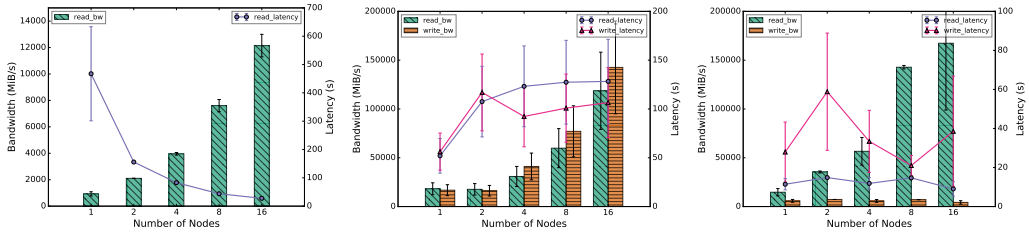
### B. Performance Analysis using Wemul

*1) Emulation of Simple Data-dependency:*

*a) Deep Learning Training Emulation:* For the experiments with DL training execution mode, we keep a 320 GiB dataset on Lassen's GPFS mount point. The dataset has 327680 1 MiB files arranged equally in 320 subdirectories. *Wemul* traverses the dataset directory tree and generates a file list, and distributes the files equally among all the processes. We run each training emulation for three epochs with a random shuffling of the file list at the beginning of each epoch. Each process is assigned fewer files to read with an increasing number of nodes. Hence, as shown in Fig. 5(a), the read latency decreases from 467 to 27 seconds for 1 to 16 nodes, respectively. Consequently, the average aggregated read bandwidth increases to $\sim$12 GiB/s for 16 nodes. It is much lower than GPFS's read bandwidth capabilities due to random read access and huge metadata overhead compared to the read size.

*b) Producer-consumer Emulation:* We generate a simple producer-consumer emulation in this mode. Each producer process is assigned exactly one file to write with the user-defined block size and segment count. The consumer counterpart polls until the file assigned to it is ready and reads it using MPI collective I/O operation. This is an inter-node producer-consumer workload. The file size of 32 GiB is defined by 256 MiB blocks arranged in 128 segments aggregating $\sim$2.2 TiB data for 16 nodes. As shown in Fig. 5(b), for 16 nodes, *Wemul* demonstrates 118 GiB/s read and 142 GiB/s write bandwidth, and 128 seconds read and 106 seconds write latency. This experiment shows adequate bandwidth due to the simplicity of the dataflow.

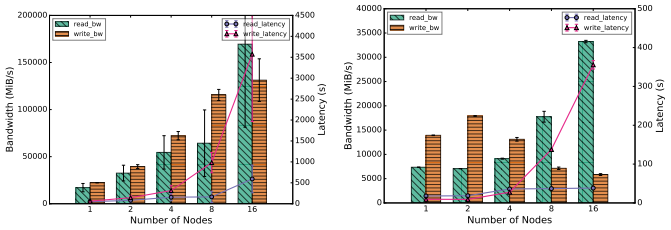
*c) Checkpoint/restart Emulation:* For emulating basic checkpoint/restart workload, we run *Wemul* to write 32 GiB checkpoint files with 10% error rate. Rank 0 is assigned to write the checkpointing file, and on a random crash emulation, each process looks for the latest checkpoint and reads the whole file. Fig. 5(c) depicts that the checkpoint writing bandwidth is low, $\sim$4 GiB/s for 16 nodes, as it is written by only one process. The maximum read bandwidth is 160 GiB/s for




(a) Deep learning training (b) Producer-consumer (c) Checkpoint/restart

Fig. 5: Analysis of workflow emulation with simple dataflow on GPFS

16 nodes. Write latency values show a random trend due to the randomness of crash emulation in *Wemul*'s implementation.




(a) Application-based (b) DAG-based

Fig. 6: Analysis of complex dataflow emulation on GPFS

*2) Emulation of Complex Data-dependency:*

*a) Application-based Emulation:* In this case, we generate a workload with three stages of data movement. *Firstly*, all the processes in the first application instance write half of the number of processes 32 GiB files on Lassen's GPFS in a shared write manner. *Secondly*, another application instance reads the files from the first stage through shared access and writes 16 GiB files in a file-per-process access pattern. *Finally*, another application instance reads the files written by the second application in a file-per-process manner and writes half of the number of processes 32 GiB files via shared access. The total data size increases up to ∼6 TiB for 16 nodes. As shown in Fig. 6(a), both read and write bandwidth reach up to 160 GiB/s and 130 GiB/s for 16 nodes, respectively. We observe that the data transfer cannot reach GPFS's IOR reported performance due to data-dependency. There can be two possible reasons behind the high variability of the results. Firstly, the experiments might have used GPFS in a busy window. Secondly, avoiding cache cancellation mechanisms in each experiment run to replicate real-world workflow scenarios can bring about very high bandwidth in some of the runs by leveraging GPFS caching.

*b) DAG-based Emulation:* As shown in Fig. 6(b), we run the same workflow discussed in Sec. III-A2. Each file in the experiment is of 32 GiB size. The dataflow here is more convoluted than that in producer-consumer and application-based workloads. It has a combination of both shared and file-per-process access in the same stage. We observe that the read bandwidth increases with the increasing number of nodes topping ∼34 GiB/s for 16 nodes, while the write bandwidth suffers and does not scale up well and decreases to ∼5 GiB/s for 16 nodes. On the other hand, the read latency stays down to around 50 seconds, and the write latency goes up to about 370 seconds for 16 node runs. The results clearly show how workflows with complex dataflow put I/O challenges on PFSs.

## V. Related Work

Classic synthetic I/O benchmarking tools are typically used to evaluate the underlying system by mimicking real-world applications [28], [15], [23]. IOzone [8] is a file system evaluation tool with functionalities to stress the underlying system but has less focus on relating the performance results with HPC applications. Flexible I/O tester (fio) [5] and Filebench [4] are leveraged to generate application workloads and test the I/O performance of a storage system. Unfortunately, these benchmarks are not focused on characterizing I/O on HPC workflows and do not support parallel I/O interfaces like MPI-IO, HDF5, etc., required by scientific applications. IOR [30], [7] exposes users to many flexible I/O request parameters to create HPC application-like I/O workloads. However, it does not provide mechanisms to address the data-dependency in workflows.

Contributions on characterizing and understanding workflow I/O have been made in the HPC community by directly running HPC application benchmarks with profiling tools [31], [25], [24], [33], [18]. There are some interesting I/O intensive scientific applications, e.g., CM1 [2], Montage [10], etc., that are often used to evaluate storage systems. There have been efforts to extract the I/O kernel from important HPC applications for the isolated analysis of the data movement. For example, HACC I/O [6], FLASH3 I/O [16], VPIC I/O [34], etc., are the I/O kernels of Hardware/Hybrid Accelerated Cosmology Code (HACC), Vector Particle-In-Cell (VPIC), FLASH code, etc., that focus on MPIIO, Parallel-NetCDF, and HDF5, respectively. However, all of these benchmarks are application-specific and not flexible enough to be extended as a generic platform for evaluating HPC application workflows.

I/O workload modeling and simulation tool, e.g., I/O Workload Abstraction (IOWA) [32], can generate workload from different sources, like I/O traces, I/O kernels, mathematical models, etc. Again, Multi-purpose, Application-Centric, Scalable I/O Proxy Application (MACSio) [26] provides an interface to create proxy applications that support various I/O interfaces and parallel I/O for multi-physics HPC applications. However, none of these generates workloads from a workflow perspective, and neither has any facility to address data-dependency among the modules of a workflow, and each task is assumed to deal with an independent stream of data.

## VI. Conclusion

Complex dataflow can pose additional I/O challenges and hinder workflow performance. Clear understanding and proper characterization of I/O is a prerequisite for developing optimization strategies to handle these issues. We develop *Wemul*, an open-source HPC workflow I/O emulation framework, to create a bridge between synthetic and application benchmarks by providing a generic platform to mimic parallel I/O workloads with data-dependencies posed by scientific application workflows on HPC systems. In the future, we will enable *Wemul* to generate workloads in a finer I/O pattern granularity by adapting features from IOR's configuration space. Besides, we can use existing techniques [14] to work on the automatic and user-friendly generation of graph representation files. For providing support for parallel I/O interfaces other than MPI, e.g., HDF5, NetCDF, ADIOS, etc., we plan to add another layer of abstraction for I/O library types in the implementation. HPC I/O researchers' community can use *Wemul* as a benchmarking tool to analyze the I/O challenges in complicated workflows, and evaluate optimization strategies and policies to overcome those.

## Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-813999. [2]

## References

[1] Asynchronous Transfer Library. https://github.com/ECP-VeloC/AXL.
[2] CM1. https://www2.mmm.ucar.edu/people/bryan/cm1.
[3] Darshan. https://www.mcs.anl.gov/research/projects/darshan.
[4] Filebench. http://www.iozone.org.
[5] Flexible I/O Tester (FIO). https://fio.readthedocs.io/en/latest.
[6] HACC I/O. https://github.com/glennklockwood/hacc-io.
[7] IOR and MDTest. https://github.com/hpc/ior.
[8] IOzone. http://www.iozone.org.
[9] Lassen. https://hpc.llnl.gov/hardware/platforms/lassen.
[10] Montage. http://montage.ipac.caltech.edu/docs/grid.html.
[11] Sierra. https://hpc.llnl.gov/hardware/platforms/sierra.
[12] Summit. https://www.olcf.ornl.gov/summit.
[13] The Large Hadron Collider. http://home.cern/topics/large-hadron-collider.
[14] B. Behzad, H. Dang, F. Hariri, W. Zhang, and M. Snir. Automatic generation of i/o kernels for hpc applications. In *2014 9th Parallel Data Storage Workshop*, pages 31–36, 2014.
[15] Peter M. Chen and David A. Patterson. A new approach to i/o performance evaluation: Self-scaling i/o benchmarks, predicted i/o performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '93, page 1–12, New York, NY, USA, 1993. Association for Computing Machinery.
[16] Ching, Choudhary, Wei-keng Liao, Ross, and Gropp. Efficient structured data access in parallel file systems. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 326–335, 2003.
[17] F. Chowdhury, F. Di Natale, A. Moody, E. Gonsiorowski, K. Mohror, and W. Yu. Understanding I/O Behavior in Scientific Workflows on High Performance Computing Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis 2019 (SC19), Regular Poster*, Nov. 2019.

[2] This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

[18] Fahim Chowdhury, Jialin Liu, Quincey Koziol, Thorsten Kurth, Steven Farrell, Suren Byna, and Weikuan Yu. Initial characterization of i/o in large-scale deep learning applications. 2018.
[19] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 80:1–80:10, New York, NY, USA, 2019. ACM.
[20] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. *Workflow Management in Condor*, pages 357–375. Springer London, London, 2007.
[21] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.
[22] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. K. Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. Streitz, F. C. Lightstone, P. Bremer, J. N. Glosli, and H. I. Ingólfsson. A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.
[23] Samuel A. Fineberg. Implementing the nht-1 application i/o benchmark. *SIGARCH Comput. Archit. News*, 21(5):23–30, December 1993.
[24] Glenn K Lockwood, Shane Snyder, Suren Byna, Philip Carns, and Nicholas J Wright. Understanding data motion in the modern hpc data center. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 74–83. IEEE, 2019.
[25] J. Luttgau, S. Snyder, P. Carns, J. M. Wozniak, J. Kunkel, and T. Ludwig. Toward understanding i/o behavior in hpc workflows. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 64–75, 2018.
[26] M. C. Miller. Multi-purpose, application-centric, scalable i/o proxy application, version 00, 6 2015.
[27] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.
[28] A. L. Narasimha Reddy and Prithviraj Banerjee. A study of i/o behavior of perfect benchmarks on a multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 312–321, New York, NY, USA, 1990. Association for Computing Machinery.
[29] Robert Ross, Lee Ward, Philip Carns, Gary Grider, Scott Klasky, Quincey Koziol, Glenn K. Lockwood, Kathryn Mohror, Bradley Settlemyer, and Matthew Wolf. Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery. 5 2019.
[30] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008.
[31] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright. Modular hpc i/o characterization with darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pages 9–17, 2016.
[32] Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross, Christopher Carothers, Babak Behzad, Huong Vu Thanh Luu, Surendra Byna, and Prabhat. Techniques for modeling large-scale hpc i/o workloads. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, New York, NY, USA, 2015. Association for Computing Machinery.
[33] Teng Wang, Suren Byna, Glenn K Lockwood, Shane Snyder, Philip H Carns, Sunggon Kim, and Nicholas J Wright. A zoom-in analysis of i/o logs to detect root causes of i/o performance bottlenecks. In *CCGRID*, pages 102–111, 2019.
[34] Kesheng Wu, Surendra Byna, and Bin Dong. Vpic io utilities. [Computer Software] https://doi.org/10.11578/dc.20181218.4, dec 2018.