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 a very important 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 are lacking in identifying and representing the dataflow in modern HPC workflows, we have implemented a 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 the emulation framework, demonstrate its application to HPC workflows, and discuss the insights from the performance analysis results on the Lassen supercomputing cluster.**

I. Introduction

The leadership HPC supercomputers continuously run thousands of scientific application workflows everyday [35]. In most of the 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 applications in a workflow can be composed of thousands or even millions of dependent or independent tasks [26]. These workflows can generate or transfer terabytes to even petabytes of data per science campaign [16], [27]. Complex data-dependency among different modules of the workflows and transfer of a humongous volume of data can create severe bottleneck and hinder the research and development in important 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 [36], [10], [7]. 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 tentatively replicate a real application I/O workload; there is not much emphasis on the data-dependency and dataflow related complexity posed by HPC workflows. Oftentimes, it is challenging 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 interests into account, we develop a workflow I/O emulation framework that can not only generate I/O workloads like conventional benchmarking tools through user-defined parameters, but also can mimic complicated dataflow represented by directed acyclic graphs (DAG). 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:

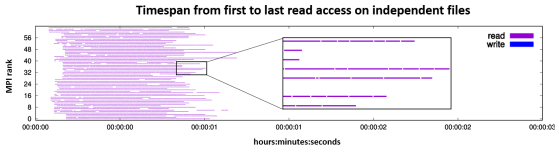
- We present our study on HPC workflow I/O challenges and workloads.
- We develop a workflow I/O emulation framework to generate different types of HPC workloads and discuss its functionalities.
- We run a performance analysis of Lassen's storage systems by I/O benchmarking using the features available in the framework.

II. Understanding I/O in HPC Workflows

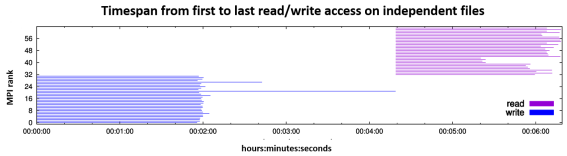
A. HPC Workflow I/O Workloads

1) Workflows with Simple Data-dependency:

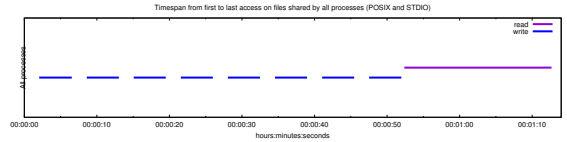
a) Deep Learning Training I/O: In DL application training with data-parallel setup, a dataset, typically kept on parallel file systems (PFS), is randomly shuffled and distributed among the processes of an application to import at the beginning



(a) DL training



(b) Inter-node producer-consumer



(c) Checkpoint/restart

Fig. 1: Simple dataflow emulation timelines on GPFS

of each epoch. The data units or files in the dataset are usually tiny (i.e., 100KB) in size, which creates huge metadata overhead [24]. In Fig. 1(a), we demonstrate how the emulation framework 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 General Parallel File System (GPFS).

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 [30]. 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 chain of I/O requests. Fig. 1(b) depicts an emulation of inter-node producer-consumer dataflow. Experiment with the I/O emulation framework 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 [35], [33], [22]. In a typical case, one or a set of processes are 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 asynchronously as specified by the user. In the case of any process crash, all the processes restart by loading data from the latest checkpoint file and save time by avoiding recalculation. As shown in Fig. 1(c), one process is assigned to write checkpoint files to GPFS in a loop. Later, the emulation framework 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 that runs for the cancer moonshot pilot 2 (CMP2) project [27]. This project aims to improve cancer diagnosis by leveraging HPC systems [18], [12], It simulates the RAS protein and cell membrane interaction for early-stage cancer cell detection. As shown in Fig. 2, this workflow has

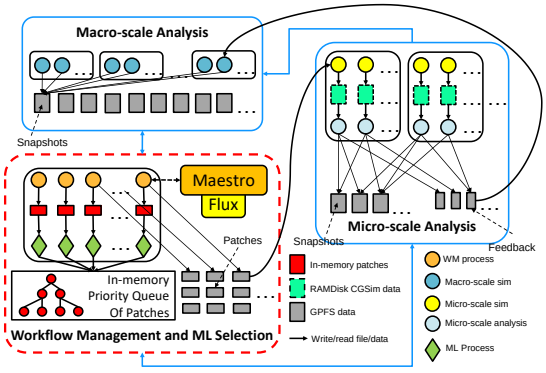


Fig. 2: Dataflow in CMP2 HPC implementation

three basic steps. *Firstly*, it writes snapshot data to GPFS via the macro 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. Hence, the CMP2 workflow's dataflow creates a chain of data transfer. Later, the output from CG simulation is written to RAMDisk on Sierra [14] and fed to CG analysis. CG analysis generates the final output snapshot and sends feedback data to the macro analysis application.

It is challenging to mimic and analyze this type of complex workflow behavior using existing benchmarking tools. Besides, running the entire workflow is tightly bound to specific supercomputers [14], [15]. 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. Workflow I/O Emulation Framework

1) Software Architecture: The workflow I/O emulation framework is an MPI-enabled C++ application. It has five 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 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. Interleaved-Or-Random (IOR) [9], is used as a static library to provide flexibility in finer granularity. For now, it can be optionally utilized from *deep_learning* module through *ior_runner* class, but is extensible to be used more robustly in the future. Asynchronous Transfer Library (AXL) [1] is also imported as a library from *checkpoint_restart* module for staging the data files in and out according to user parameters.

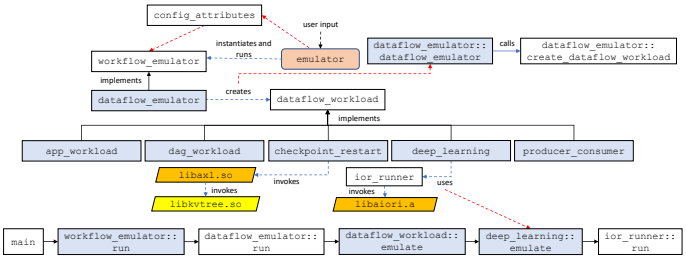
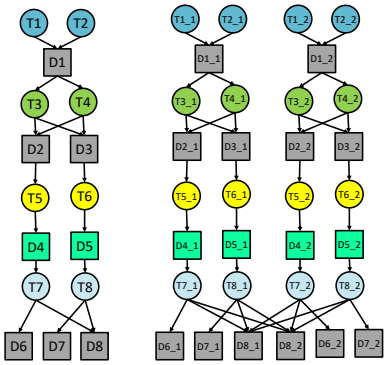


Fig. 3: Class diagram of workflow I/O emulation framework

2) Emulating Complex Workflow Data-dependencies: The workflow I/O emulation framework provides an interface to describe an entire workflow in two types of strategies. *Firstly*, users can define applications with the unique application ID, name, number of processes, and estimated walltime. *Secondly*, the user can go into finer granularity and define each workflow's tasks where multiple tasks can represent each application. In both cases, the user can specify data units, typically files, present in the workflow with its name and size. Finally, there is a section to express the parent-child relationship between applications or tasks and data units. DAG representation examples are discussed in Appendix A.



(a) 1 node 8 processes emulation

(b) 2 nodes 16 processes emulation

Fig. 4: Emulating dataflow in CMP2

As depicted in Fig. 4, we can specify a small scale adaptation of the CMP2 workflow. In a single node setup with only eight processes shown in Fig. 4(a), we represent T1 and T2 tasks as macro analysis processes, T3 and T4 as machine learning application, T5 and T6 as CG simulation, and T7 and T8 as CG analysis. The Dx vertices represent the data units associated to 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: The workflow I/O emulation framework has five basic execution modes.

1) DL training: the emulation framework 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 producer only or consumer only application.
3) Checkpoint/restart: we 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: in this mode, the framework runs as a standalone application. We can set the list of files to read or write, block size and segment count, and access patterns through parameters.
5) DAG-based: we can take a DAG representation of the entire workflow as a parameter, and the emulation framework can mimic that workflow. See Sec. III-A2.

4) Functionality and Usage: The I/O emulation framework 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 starting information for the framework, i.e., *type* of emulation (we keep this parameter for future extension), *subtype* to specify an execution mode, and *input directory* to use for emulation. 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, "Application-based" category has parameters to set the file list for reading and writing by the emulation framework and access patterns. "DAG-based" category has only one parameter that takes the path to the file with DAG specification 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. The parameters and their usage are shown in more detail in Table I.

IV. Experimental Results

A. Testbed and Workload

We run all the experiments on Lassen [11], a 795 nodes IBM Power9 supercomputer with 44 cores and 256 GB memory per node situated at Lawrence Livermore National Laboratory (LLNL). Besides, it has a 24 PB IBM's Spectrum Scale GPFS,

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
	$--input\_dir < path >$	Mountpoint 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	$--inter\_node$	Set to place producer and consumer processes on different nodes
	$--producer\_only$	Set to run emulator as a standalone producer application
	$--consumer\_only$	Set to run emulator 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	$--read\_filenames < file1 : file2 : .. >$	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)
	$--write\_filenames < file1 : file2 : .. >$	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 DAG representation of workflow

TABLE I: User parameters of workflow I/O emulation framework

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] profiling 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 execution five times and take the mean and standard error of the bandwidth and latency values. During a prior experiment, using IOR benchmark with sequential file-per-process I/O on GPFS, we got ∼186 GiB/s read and ∼190 GiB/s write bandwidth for 16 nodes.

B. Performance Analysis using I/O Emulation Framework

1) Emulation of Simple Data-dependency:

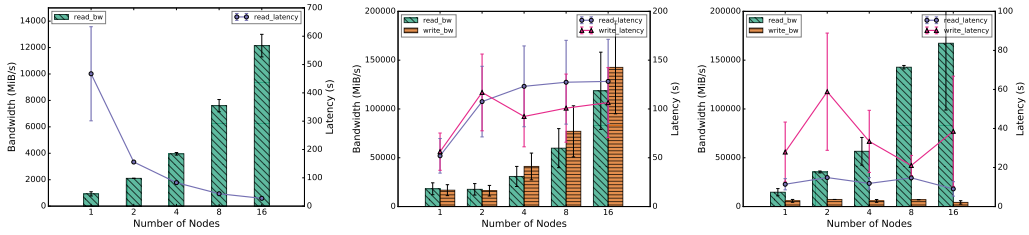
a) Deep Learning Training Emulation: For the experiments with DL training execution mode, we keep a 320 GiB dataset on Lassen's GPFS mountpoint. The dataset has 327680 1 MiB files arranged equally in 320 subdirectories. The framework traverses the dataset directory structure and generates a file list, and distributes the files equally among all the processes to read. We run each training emulation for three epochs with a random shuffling of the list at the beginning of each epoch. As shown in Fig. 5(a), with an increasing number of nodes, each process is assigned less files to read. Hence, the read latency decreases from 467 to 27 seconds for 1 to 16 nodes, respectively. Consequently, average aggregated read bandwidth increases to around 12 GiB/s for 16 nodes, but it is much lower than GPFS's read bandwidth capabilities.

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 a file is ready and later reads it using MPI collective I/O operation. This is an inter-node producer-consumer I/O. The file size of 32 G is defined by 256 MiB blocks arranged in 128 segments aggregating ∼2.2 TiB data for 16 nodes. As shown in Fig. 5(b), for 16 nodes, the emulation framework execution demonstrates 118 GiB/s read and 142 GiB/s write bandwidth, and 128 seconds read and 106 seconds write latency. This experiment shows good bandwidth due to the simplicity of the dataflow.

c) Checkpoint/restart Emulation: For emulating basic checkpoint/restart workload, we run the I/O emulation framework to write 32 G 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 read the whole file. Fig. 5(c) depicts that the checkpoint writing bandwidth is about 4 GiB/s for 16 nodes as it is written by only one process. The maximum read bandwidth is 160 GiB/s for 16 nodes. The latency values show a random trend due to the randomness of crash emulation in the implementation.

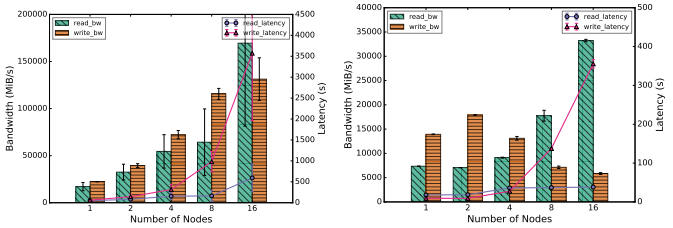
2) Emulation of Complex Data-dependency:

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 G files on Lassen's GPFS in a shared write manner. *Secondly*, another application instance reads the files from the first one through shared access and writes 16 G files in the file-per-process access type. *Finally*, another application instance reads the files written by the second application in a file-per-process pattern and writes half of the number of processes 32 G files via shared access. The total data size increases up to ∼6 TB 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.



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

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



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

Fig. 6: Analysis of complex dataflow emulation on GPFS

b) DAG-based Emulation: In case of the experiment as shown in Fig. 6(b), we run the same workflow discussed in Sec. III-A2. The dataflow here is more convoluted than that in producer-consumer and application-based workloads, as 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 about 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. Each file in the experiment is of 32 G size. This workload cannot leverage the full potential of Lassen's GPFS due to data-dependency. On the other hand, the read latency keeps down to around 50 seconds, and the write latency goes up to about 370 seconds for 16 node runs. The results clearly show how these workflows with complex dataflow put I/O challenges on PFSs.

V. Related Work

Classic synthetic I/O benchmarking tools are typically used to evaluate underlying system by mimicking real world applications [34], [20], [28], [28]. On a basic level, there is UNIX command-line utility named *dd* [4] that takes blocksize and segment count as input and outputs basic statistics of a data transfer. IOzone [10] 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) [7] and Filebench [5] are leveraged to generate application workloads and test 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 [36], [9] exposes users to many flexible I/O request parameters to create HPC application-like I/O workloads. However, it does not provide mechanisms to depict the data-dependency in workflows. In our emulator's DL workload module, a slightly modified IOR is used as a library with additional flexibility to pose multiple files-per-process workloads.

Contributions on characterizing and understanding workflow I/O have been made in HPC community by directly running HPC application benchmarks with profiling tools [37], [32], [31], [38], [23]. There are some interesting I/O intensive scientific applications, e.g., CM1 [2], Montage [13], etc., that are often used to evaluate storage systems. There have been efforts on extracting the I/O kernel from important HPC applications for isolated analysis of the data movement. For example, HACC I/O [8], FLASH3 I/O [21], VPIC I/O [39], etc., are the I/O kernels of Hardware/Hybrid Accelerated Cosmology Code (HACC) [29], Vector Particle in-cell (VPIC) [17], FLASH code [6], 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. Behzad et al. [19] proposed a technique to automate the I/O kernel generation of any scientific application. This work can be leveraged to translate the I/O kernel to dataflow specification understandable to our emulation framework.

In summary, we develop the workflow I/O emulation framework to create a bridge between synthetic and application benchmarks by providing a generic platform to mimic parallel I/O workload posed by common HPC scientific application workflows.

VI. Conclusion

It is difficult to perceive the behavior of scientific application workflow on HPC systems. Complex dataflow poses additional I/O challenges and hinders workflow performance. Clear understanding and proper characterization of I/O is a prerequisite for developing optimization strategies to improve HPC workflow performance. Our workflow I/O emulation framework shows promise in perceiving complex HPC workflow and dataflow in a user-intuitive way. In the future, this framework can be used as a benchmarking tool to analyze the I/O issues in more complicated workflows, and evaluate optimization strategies and policies to overcome those challenges.

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-DRAFT.

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] dd (Unix). https://en.wikipedia.org/wiki/Dd_(Unix).

[5] Filebench. http://www.iozone.org.

[6] FLASH3 Code. http://flash.uchicago.edu/site/flashcode.

[7] Flexible I/O Tester (FIO). https://fio.readthedocs.io/en/latest.

[8] HACC I/O. https://github.com/glennklockwood/hacc-io.

[9] IOR and MDTest. https://github.com/hpc/ior.

[10] IOzone. http://www.iozone.org.

[11] Lassen. https://hpc.llnl.gov/hardware/platforms/lassen.

[12] MaestroWF. https://github.com/LLNL/maestrowf.

[13] Montage. http://montage.ipac.caltech.edu/docs/grid.html.

[14] Sierra. https://hpc.llnl.gov/hardware/platforms/sierra.

[15] Summit. https://www.olcf.ornl.gov/summit.

[16] The Large Hadron Collider. http://home.cern/topics/large-hadron-collider.

[17] Vector Particle in-cell (VPIC). https://github.com/lanl/vpic.

[18] Dong H Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Joseph Koning, Tapasya Patki, Thomas RW Scogland, Becky Springmeyer, et al. Flux: overcoming scheduling challenges for exascale workflows. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 10–19. IEEE, 2018.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. *Workflow Management in Condor*, pages 357–375. Springer London, London, 2007.

[26] 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.

[27] Francesco Di Natale, Harsh Bhatia, Timothy S. Carpenter, Chris Neale, Sara Kokkila Schumacher, Tomas Oppelstrup, Liam Stanton, Xiaohua Zhang, Shiv Sundram, Thomas R. W. Scogland, Gautham Dharuman, Michael P. Surh, Yue Yang, Claudia Misale, Lars Schneidenbach, Carlos Costa, Changhoan Kim, Bruce D'Amora, Sandrasegaram Gnanakaran, Dwight V. Nissley, Fred Streitz, Felice C. Lightstone, Peer-Timo Bremer, James N. Glosli, and Helgi 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.

[28] Samuel A. Fineberg. Implementing the nht-1 application i/o benchmark. *SIGARCH Comput. Archit. News*, 21(5):23–30, December 1993.

[29] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, and et al. Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, Jan 2016.

[30] Anthony Kougkas, Hariharan Devarajan, Jay Lofstead, and Xian-He Sun. Labios: A distributed label-based i/o system. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, pages 13–24, New York, NY, USA, 2019. ACM.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Kesheng Wu, Surendra Byna, and Bin Dong. Vpic io utilities. [Computer Software] https://doi.org/10.11578/dc.20181218.4, dec 2018.

APPENDIX

A. Example DAG Representation of Complex Workflows

A complex workflow can be represented as a DAG of applications and data described by a file with two sections. The first section has the definition of the applications and the data units. As shown in Fig. 7(a), each line for application definition has 5 tokens. "APP" is the keyword to represent an application. The second token is a unique ID of the application. The next tokens are application name, number of processes involved and estimated walltime for the application. The data unit definition line starts with "DATA" keyword. Then it denotes the data ID, name and the size of data in bytes. In the second section, it has the parent-child relationship between applications and data units. As depicted in Fig. 7(b), when an application is "PARENT" of a data "CHILD", that means the application writes the data, otherwise the application reads. Besides, there is a an indication about the "ACCESS" type, i.e., 0 for shared and 1 for file-per-process access. For example, when D2, D3 and D4 are the parents of A3 with access type 1, it denotes that D2 is read by the first process in the three processes involved with A3. The second and third processes read D3 and D4.

```
APP A1 ml_process 1 12
APP A2 macro_an 2 8
APP A3 micro_sim 3 5
APP A4 micro_an 3 6
DATA D1 macro_snapshot 3
DATA D2 patch_1 3
DATA D3 patch_2 3
DATA D4 patch_3 3
DATA D5 sim_patch_1 3
DATA D6 sim_patch_2 3
DATA D7 sim_patch_3 3
DATA D8 feedback_1 3
DATA D9 feedback_2 3
DATA D10 feedback_3 3
DATA D11 micro_snapshot 3
```

(a) Definition

```
PARENT A2 CHILD D1 ACCESS 0
PARENT D1 CHILD A1 ACCESS 0
PARENT A1 CHILD D2 D3 D4 ACCESS 0
PARENT D2 D3 D4 CHILD A3 ACCESS 1
PARENT A3 CHILD D5 D6 D7 ACCESS 1
PARENT D5 D6 D7 CHILD A4 ACCESS 1
PARENT A4 CHILD D8 D9 D10 ACCESS 1
PARENT A4 CHILD D11 ACCESS 0
NS_PARENT D8 D9 D10 NS_CHILD A2 ACCESS 0
```

(b) Relation

Fig. 7: Example app-data DAG file

The DAG representation can also be expressed by finer granularity task-data relationships. The first section, showed in Fig. 8(a), has task definition lines starting with "TASK" keyword. The next tokens of the line include task ID, name and estimated runtime. The data definitions are same as discussed for previous app-data strategy. The relationship, as

```
TASK T1 ml_process 12
TASK T2 macro_1 8
TASK T3 macro_2 8
TASK T4 micro_sim_1 5
TASK T5 micro_sim_2 5
TASK T6 micro_sim_3 5
TASK T7 micro_an_1 6
TASK T8 micro_an_2 6
TASK T9 micro_an_3 6
DATA D1 macro_snapshot 3
DATA D2 patch_1 3
DATA D3 patch_2 3
DATA D4 patch_3 3
DATA D5 sim_patch_1 3
DATA D6 sim_patch_2 3
DATA D7 sim_patch_3 3
DATA D8 feedback_1 3
DATA D9 feedback_2 3
DATA D10 feedback_3 3
DATA D11 micro_snapshot 3
```

(a) Definition

```
PARENT T2 T3 CHILD D1
PARENT D1 CHILD T1
PARENT T1 CHILD D2 D3 D4
PARENT D2 CHILD T4
PARENT D3 CHILD T5
PARENT D4 CHILD T6
PARENT T4 CHILD D5
PARENT T5 CHILD D6
PARENT T6 CHILD D7
PARENT D5 CHILD T7
PARENT D6 CHILD T8
PARENT D7 CHILD T9
PARENT T7 CHILD D8
PARENT T8 CHILD D9
PARENT T9 CHILD D10
NS_PARENT D8 D9 D10 NS_CHILD T2 T3
PARENT T7 T8 T9 CHILD D11
```

(b) Relation

Fig. 8: Example task-data DAG file

demontrated in Fig. 8(b), is also expressed by "PARENT" and "CHILD" keywords. This strategy allows a single data transfer stage with multiple I/O access types. For example, T1 being the parent of D2, D3 and D4 means T1 sequentially writes these three files. This dataflow representation strategy is highly inspired by DAGMan [25].