DFMan: A Graph-based Optimization of Dataflow Scheduling on High-Performance Computing Systems

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

***Abstract*—Scientific research and development campaigns are materialized by workflows of applications executing on high-performance computing (HPC) systems. These applications consist of tasks that can have inter- or intra-application flows of data to achieve the research goals successfully. These dataflows create dependencies among the tasks and cause resource contention on shared storage systems, thus limiting the aggregated I/O bandwidth achieved by the workflow. However, these I/O performance issues are often solved by tedious and manual efforts that demand holistic knowledge about the data dependencies in the workflow and the information about the infrastructure being utilized. Taking this into consideration, we design *DFMan*, a graph-based dataflow management and optimization framework for maximizing I/O bandwidth by leveraging the powerful storage stack on HPC systems to manage data sharing optimally among the tasks in the workflows. In particular, we devise a graph-based optimization algorithm that can leverage an intuitive graph representation of dataflow- and system-related information, and automatically carry out co-scheduling of task and data placement. According to our experiments, *DFMan* optimizes a wide variety of scientific workflows such as Hurricane 3D on Cloud Model 1 (CM1), Montage Carina Nebula (NGC3372), and an emulated dataflow kernel of the Multiscale Machine-learned Modeling Infrastructure (MuMMI I/O) on the Lassen supercomputer, and improves their aggregated I/O bandwidth by up to 5.42×, 2.12× and 1.29×, respectively, compared to the baseline bandwidth.**

I. Introduction

Inter-disciplinary scientific research campaigns take place regularly to solve important problems in medical science, environmental science, astrophysics, etc. [1], [2]. Generally, researchers and scientists solve real-world problems via the workflows of applications executing on HPC infrastructures. These applications usually have thousands to millions of inter-dependent tasks and can transfer terabytes or even petabytes of data [3], [4]. For instance, the cancer research workflow for investigating the interaction of RAS proteins and the cell membrane managed by a Multiscale Machine-Learned Modeling Infrastructure (MuMMI) [4] demands 154 PB of usable storage with 1.5 TB/s peak bandwidth from Sierra [5]. Besides the sheer volume of data, another limiting factor in HPC workflows is the data dependencies among the tasks of the applications. These dependencies generate a highly-convoluted relationship between task and data that is difficult to manage [6]. This situation mandates using shared storage systems for data movement and limits the I/O performance due to resource contention issues. Hence, data movement's volume and velocity often hinder the pace and quality of crucial scientific research advancements via HPC workflows.

Data management is considered one of the most challenging tasks in scientific and industry application development for HPC systems [4], [7], [8]. Even though this I/O maintenance is usually kept untouched until it becomes a potential bottleneck in application performance, many efforts have taken place over the years that address the understanding and exploration of I/O workloads [6], [9], [10]. Moreover, efficient scheduling of tasks on distributed systems is vital for reaching HPC performance targets, such as maximizing task parallelism, I/O, and communication bandwidth [11]–[14]. Efficient data management in HPC workflows requires the development of not only optimized scheduling strategies for data placement, but also proper assignment of computation resources to tasks. This scenario demands the development of optimal task-data co-scheduling policies.

In an HPC workflow, there can be thousands of producer-consumer data dependencies among the tasks. Proper understanding of the task-data dependencies exposes the complexity of the dataflow and helps extract the semantics of the workflow I/O behavior. On the other hand, the latest HPC systems with heterogeneous computation and storage resources have dense relationships. Both the task-data and computation-storage relationships can be organized using graph data structures. Careful use of this workflow- and system-specific information is required to design optimized co-scheduling strategies for assigning tasks and data to computation and storage resources, respectively. This task-data co-scheduling can be formulated as a general assignment problem [15], which is an NP-hard combinatorial optimization problem [16].

Taking the need for HPC dataflow optimization into consideration, we design and develop *DFMan*, a dataflow management system to provide task-data co-scheduling strategies for resource management systems using the information about the dataflow structures and systems resource hierarchy. We devise methods for representing the task-data dependencies in a workflow as a graph and extracting the directed acyclic graph (DAG). Additionally, *DFMan* provides a system information management module to maintain another graph for keeping track of the computation and storage resource

relationships. We construct a variable space with the task-data and computation-storage pairs and formulate the assignment problem as a constrained max bipartite graph matching problem. We use linear programming to find optimal task-data to computation-storage mapping that maximizes the aggregated I/O bandwidth in the workflow.

We evaluate *DFMan*'s automatic scheduling policies by using synthetic workflows built by Wemul [6], two HPC application workflows: Hurricane 3D on CM1 [17] and Carina Nebula image mosaic visualization workflow on Montage [18], and two HPC dataflow kernels: the I/O module of Hardware/Hybrid Accelerated Cosmology Code (HACC I/O) and MuMMI I/O [4]. According to our experiments on the Lassen supercomputer [19], we achieve optimized I/O performance in all cases using *DFMan*'s strategies and match the informed policies applied manually to leverage the system's resources optimally.

In summary, we make the following key contributions.

- We introduce a method for representing HPC workflow as a graph, managing task-data dependency information, and extracting important DAG for scheduling.
- We design the system's resource hierarchy information management module for constructing the computation-storage resource relationship graphs.
- Most importantly, we develop a novel technique for formulating the task-data co-scheduling on computation-storage as a general assignment problem, redefining it as a bipartite graph matching problem, and solving it using linear programming methods.
- We develop a prototype and evaluate its efficacy in solving dataflow management issues by leveraging the modern HPC storage stack.

II. Background

A. HPC Workflow and Dataflow

A scientifically important objective or a scientific campaign can be operated by a set of single or multiple applications [1]. These applications are often executed by a complex network of thousands to even millions of inter-dependent or independent tasks [2]. These sets of applications running on HPC systems create HPC workflows. A task in a workflow can depend on or consume the data produced by other tasks. This situation creates sharing data among the tasks in different or the same application causing inter- or intra-application data dependencies. The data movement caused by this data-sharing among the tasks, or the inter-task data transfer in a workflow, is referred to as dataflow in this paper. Usually, dataflows in different HPC workflow types, such as simulation, and experimental and observational data analysis, can be organized by DAGs. More complicated workflows with feedback mechanisms create cyclic dependencies that pose more challenges in handling the tasks and data efficiently [4], [6].

B. Scheduling in Resource and Workflow Managers

Batch job scheduling in distributed HPC supercomputers is mainly handled by resource management software systems. These systems are responsible for allocating HPC resources to jobs or tasks while considering administrative policies and resource availability [7]. Classic resource management systems like SLURM [20], IBM Spectrum LSF [21], etc., are frequently used in leadership supercomputers for scheduling batch jobs for executing scientific application workflows. Flux is an emerging workload manager that can allocate resources into even finer granularity by allowing hierarchical task scheduling per core in addition to that per node [22]. These resource managers generally provide some built-in scheduling schemes, such as first-come, first-served (FCFS), ALCF WFP [23], Conservative and EASY Backfilling, etc., for managing resource utilization in large workflows.

Workflow management systems, such as Pegasus [24], MaestroWF [25], Cylc [26], etc., can simplify job and data management complexities in HPC workflows. These workflow managers provide user-friendly syntax to define a workflow and often deal with challenges, such as basic task scheduling [27], tracking dataset [24], fault-handling, etc. Most of the workflow managers, except Cylc, expect a workflow to be acyclic. Cylc requires information about the starting tasks of a cyclic workflow from the workflow developers. However, none of the present managers has support for automatically determining optimized task and data co-scheduling strategies using dataflow representation of a workflow and underlying storage stack information.

C. I/O Acceleration Opportunities in HPC Infrastructure

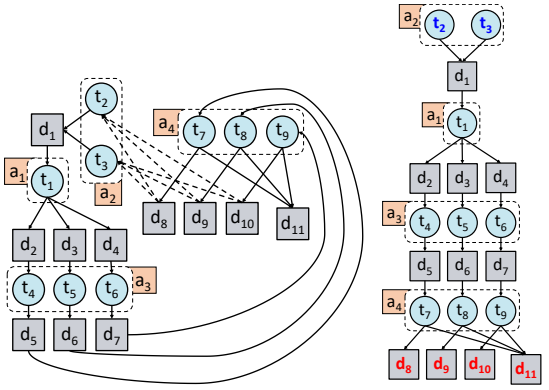
HPC storage stack typically contains fast ephemeral node-local storage devices on I/O nodes, parallel file systems, campaign, and archival storage systems. The storage capacity and data lifetime increase from top to bottom in the stack, while the latency and bandwidth performance degrade. At the top of the stack, node-local storage systems and storage-class memory devices, such as 3D XPoint, Z-NAND flash memory, etc., have the fastest performance with the lowest capacity and data lifetime. Applications can access these storage devices directly through I/O libraries or file systems on top of device drivers. Most of the modern supercomputers are equipped with disaggregated storage through dedicated I/O nodes, usually handled by burst-buffer management systems, such as Cray DataWarp. Parallel file systems (PFS) sit on the next level of the hierarchy built on top of network-attached storage devices and handled by PFS client and server software. PFS can usually have up to several months of data lifetime, depending on the system's administrative policies Campaign storage systems are kept one step lower in the hierarchy for supporting long-running application campaigns. Finally, the archive storage system is situated at the bottom of the stack, usually consisting of magnetic tape devices. It has plodding access speed but almost infinite data lifetime and capacity for keeping huge archival scientific data. It is essential for any data-intensive workflow to be designed and executed to effectively leverage the rich storage stack present in modern HPC systems.

III. Motivation

A. Why Intelligent Scheduler: An Illustrative Example

1) System Specification: Let us assume an HPC cluster has three available nodes, n_1, n_2, and n_3, with two compute cores each, c_1 and c_2 on n_1, c_3 and c_4 on n_2, and c_5 and c_6 on n_3. The system has three types of storage systems such as node-local ram disk (RD), shared burst buffer file system (BB), and global parallel file system (PFS). As shown in TABLE 2(b), let us assume s_1, s_2 and s_3 are instances of node-local RD accessible from n_1, n_2 and n_3, respectively. Each of those RD instances has a read bandwidth of 6 data units per time unit (size/time) and 3 size/time write bandwidth. s_4 is a BB instance accessible from n_2 and n_3 with read and write bandwidth of 4 and 2 size/time, respectively. Lastly, s_5 is the PFS instance accessible from all three nodes and has read and write bandwidth of 2 and 1 size/time, respectively.

2) Workflow Definition: As shown in Fig. 1(a), the workflow has four applications, a_1, a_2, a_3 and a_4. a_1 runs one task, t_1. a_2 runs t_2, and t_3. a_3 and a_4 runs three tasks each, t_4-t_6 and t_7-t_9, respectively. We assume that the size of each data instance is the same, i.e., 12 data units. As a result, RD, BB and PFS take 2, 3 and 6 time units to read, and 4, 6 and 12 time units to write a data instance, respectively.



(a) Original graph (b) Extracted DAG

Fig. 1: Representation of task-data dependencies in workflow as graphs. Round and square vertices specify tasks and data; solid and dashed edges represent required and optional dependencies, respectively. See § IV-B1 for more details.

TABLE 2(a) shows the estimated duration for each task when a task uses a specific storage system. We extract the DAG in Fig. 1(b) from the cyclic graph in Fig. 1(a) using the techniques discussed in § IV-B1. The starting vertices of each iteration of the cyclic workflow are t_2 and t_3 tasks, and the ending vertices are d_8, d_9, d_{10}, d_{11} data instances.

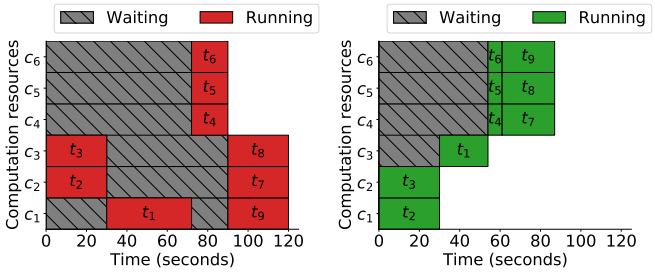
3) Scheduling Strategies: We consider a naïve strategy where the tasks are triggered serially from t_1 to t_9 and assigned to the computation resources on an FCFS basis. On the other hand, the scheduler is not aware of the task-data dependencies and the storage stack information; hence, it places all the data to the PFS so that the data can be accessed from any computation resource. As demonstrated in Fig. 2(c), the I/O time taken for an iteration of the example workflow is 120 seconds. The runtime degradation happens because the consumer tasks wait for the producer tasks to finish.

Task ID	Est. I/O time (s)		
	RD	BB	PFS
t_1	14	21	42
t_2	10	15	30
t_3	10	15	30
t_4	6	9	18
t_5	6	9	18
t_6	6	9	18
t_7	10	15	30
t_8	10	15	30
t_9	10	15	30

(a) Estimated I/O time of tasks for storing related data on each storage type.

Storage ID	Storage type	B/W (size/time)		Data instances
		Read	Write	
s_1	RD	6	3	-
s_2	RD	6	3	d_5
s_3	RD	6	3	d_6, d_7
s_4	BB	4	2	d_2 d_3 d_4
s_5	PFS	2	1	d_1 d_8, d_9 d_{10}, d_{11}

(b) Optimal placement of data on the available storage systems.



(c) Naïve scheduling (d) Task-data co-scheduling

Fig. 2: Benefits of using an intelligent task-data co-scheduler.

Let us consider an intelligent scheduler that uses the task-data dependencies and systems' information. This scheduler lessens the I/O wait time in consumer tasks by alleviating resource contention using node-local storage systems. As shown in Fig. 2(d), this type of intuitive scheduling can improve the runtime of the example workflow to 87 seconds. In this case, the scheduler uses all the storage systems available to optimally store the data instances and collocates the tasks according to the task-data relationships. We observe 27.5% runtime improvement in this tiny illustrative workflow iteration. This observation sets up a plot for the performance opportunities using a similar informed and optimized scheduling in real-world scenarios.

B. Key Takeaways

The above discussion demonstrates that:

- Existing resource and workflow management systems do not generally consider the powerful storage systems available on modern supercomputers.
- HPC workflows suffer I/O issues due to complex data dependencies among the tasks and resource contention for storing data in shared storage systems.
- HPC dataflows can be managed efficiently by an intelligent dataflow- and system-aware scheduler providing optimized task-data co-scheduling strategies to alleviate I/O performance issues.

IV. Design and Methodology

A. DFMan Overview

We design *DFMan* to ensure efficient data movement among the tasks of an HPC workflow by leveraging the dataflow infor-

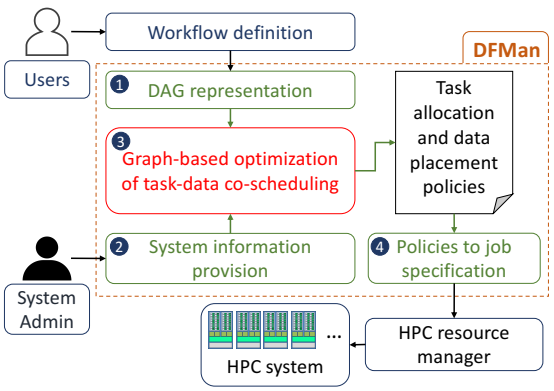


Fig. 3: *DFMan* graph-based optimization framework design.

mation from the workflow developers and the system's information from the administrators. Fig. 3 depicts an overview of the key techniques used in *DFMan*. Four building blocks work together to achieve and deploy the optimized co-scheduling of tasks and data. **(1)** The user defines an HPC workflow along with the data dependencies among the applications or tasks. *DFMan* determines if there is any cycle in the graph representation of the dataflow and extracts the DAG from the original graph. **(2)** On the other hand, system administrators maintain the information about the computation and storage resources' relationships. *DFMan* manages these relationships with graph data structures and constructs auxiliary in-memory hashmaps for fast data retrieval. **(3)** The main component of *DFMan* devises the graph algorithm formulation and linear programming optimization of the co-scheduling problem. *DFMan* feeds the extracted workflow DAG and system information graph to the optimization component for deducing task assignment and data placement policies to improve workflow runtime by maximizing the aggregated I/O bandwidth. **(4)** Finally, *DFMan* maps the scheduling policies provided by the optimizer to job specifications that are comprehensible to the HPC workflow and resource managers.

B. Graph-based Optimization Framework Design

1) Dataflow Graph Construction and DAG Extraction: *DFMan* constructs a directed graph for assisting the optimization component to deduce the task-data co-scheduling policies. The graph consists of two types of vertices to represent tasks and data instances. Edges connect the vertices in the directed graph to specify the relationships among tasks and data. A flow of data is represented by an edge between a task vertex and a data vertex. An incoming edge to a task vertex from a data vertex means the task consumes the data instance. The incoming edges can be of two types: required (task needs to consume the data to start) and optional (task can start without the input data). An outgoing edge from a task vertex to a data vertex means the task produces the data instance. If an edge is between two task nodes, it simply specifies the order of execution of the tasks. In the graph representation, there can be no edge between two data vertices because a data instance cannot create another data instance without the help of a task. *DFMan* constructs a complete picture of the workflow, including the data transfers among its modules. To programmatically schedule a cyclic workflow without human intervention, it uses an efficient linear-time graph coloring algorithm with depth-first search (DFS) to find if any back-edge exists in the graph [16] and removes the optional edges in the cyclic path to extract the DAG. *DFMan* automatically detects the starting and ending vertices of the workflow graph. While traversing the graph with DFS, it creates a topologically sorted list of tasks by assigning producer tasks of a data instance higher priority scores than the consumer tasks. Later, task-data dependency information from this DAG is passed to the optimization component. This component obtains the optimal co-scheduling policies for the extracted DAG. The final workflow goal is achieved by executing the DAG for multiple iterations.

2) Organizing Systems' Information for Scheduler: *DFMan* manages the information about the computation and storage resources of an HPC system as a tree of the resource hierarchy. For instance, it maintains the data about the compute nodes, the storage stack and the accessibility of each computation resource to storage systems. Besides, it keeps track of several auxiliary information, such as system administrator's data and available I/O libraries. *DFMan* analyzes the elements of the tree and internally constructs a bipartite graph to specify the computation to storage resource accessibility. *DFMan* feeds this graph representing the computation-storage relationship to the optimization component for building the variables and constraints of the task-data co-scheduling optimization model.

*3) **The Intelligent Task-data Co-scheduler**:*

a) Problem Specification: The principal target of *DFMan* is to automatically improve the dataflow in an HPC workflow by leveraging cutting-edge storage resources on modern supercomputers. For reaching this target, *DFMan* co-schedules the data placement to the appropriate storage system while ensuring task assignment to the proper computation resources. These assignments and placements need to work under some strict constraints. For instance, if there is a dependency of a task to a particular data or vice versa, the computation resource running the task should have access to the storage where the corresponding data is situated. The data placement policy must ensure enough space in the storage system on which a data instance is scheduled to be stored. Moreover, The task assignment strategy needs to check if the scheduling exceeds the estimated wall time specified by the user. There can be multiple combinations of these task-data co-scheduling strategies on computation-storage resources. *DFMan* determines the co-scheduling policy that maximizes the aggregated I/O bandwidth to execute an I/O efficient workflow with the improved data transfer time. Therefore, the task-data co-scheduling is a combination of two generalized assignment problems (GAP) [15] with multiple constraints, which is a complex NP-hard combinatorial problem [28].

To mathematically express the GAPs, let us assume a dataflow is defined by T and D, and an HPC system is defined by C and S. The details of the basic mathematical notations are described in TABLE I. To solve these GAPs, we need

Category	Notation	Definition	Description
Dataflow specification	T	$T = \{t_i : i \in \mathbb{N}\}$	A set of all tasks, where $\mathbb{N}$ is the set of all natural numbers.
	T^w	$T^w = \{t_i^w : i \in \{1, 2, \ldots, \lvert T\rvert\}\}$	A set of estimated wall time for each task.
	D	$D = \{d_i : i \in \mathbb{N}\}$	A set of all data instances.
	D^s	$D^s = \{d_i^s : i \in \{1, 2, \ldots, \lvert D\rvert\}\}$	A set of sizes for each data instance.
	R	$R = \{r_i : i = \{1, 2, \ldots, \lvert D\rvert\}\}$	A set of binary numbers to specify if a file is read from the storage by a task, where r_i is 1 if d_i is read, 0 otherwise.
	W	$W = \{w_i : i = \{1, 2, \ldots, \lvert D\rvert\}\}$	A set of binary numbers to specify if a file is written to the storage by a task, where w_i is 1 if d_i is written, 0 otherwise.
	D^{rt}	$D^{rt} = \{d_i^{rt} : i \in \{1, 2, \ldots, \lvert D\rvert\}\}$	A set of number of reader tasks per data instance.
	D^{wt}	$D^{wt} = \{d_i^{wt} : i \in \{1, 2, \ldots, \lvert D\rvert\}\}$	A set of number of writer tasks per data instance.
System information	C	$C = \{c_i : i \in \mathbb{N}\}$	A set of computation resources.
	S	$S = \{s_i : i \in \mathbb{N}\}$	A set of storage system instances.
	S^c	$S^c = \{s_i^c : i \in \{1, 2, \ldots, \lvert S\rvert\}\}$	A set of capacity for each storage instance.
	B^r	$B^r = \{b_i^r : i \in \{1, 2, \ldots, \lvert S\rvert\}\}$	A set of read bandwidths for each storage instance.
	B^w	$B^w = \{b_i^w : i \in \{1, 2, \ldots, \lvert S\rvert\}\}$	A set of write bandwidths for each storage instance.
	S^p	$S^p = \{s_i^p : i \in \{1, 2, \ldots, \lvert S\rvert\}\}$	A set of the maximum number of tasks on the same topological level recommended for each storage instance. $s_i^p \leq ppn$ for node-local and $s_i^p \leq ppn \times n_n$ for global storage, where ppn is the number of processes per node and n_n is number of nodes.
Task-data co-scheduling	TD^b	$TD^b = \{td_{ij}^b : (td_{ij}^b \in [0,1]) \wedge \exists(t_i, d_j) \wedge (t_i \in T) \wedge (d_j \in D)\}$	A set of binary numbers to indicate task-data dependencies, where td_{ij}^b is 1 if there is dependency between t_i and d_j, 0 otherwise.
	TD	$TD = \{td_{ij} : td_{ij} = (t_i, d_j) \wedge (t_i \in T) \wedge (d_j \in D) \wedge (td_{ij}^b = 1) \wedge (td_{ij}^b \in TD^b)\}$	A set of task-data pairs, where the task in each pair reads or writes the data instance.
	CS^b	$CS^b = \{cs_{ij}^b : (cs_{ij}^b \in [0,1]) \wedge \exists(c_i, s_j) \wedge (c_i \in C) \wedge (s_j \in S)\}$	A set of binary numbers to specify computation-storage resource accessibilities, where cs_{ij}^b is 1 if s_j is accessible from c_i, 0 otherwise.
	CS	$CS = \{cs_{ij} : cs_{ij} = (c_i, s_j) \wedge (c_i \in C) \wedge (s_j \in S) \wedge (cs_{ij}^b = 1) \wedge (cs_{ij}^b \in CS^b)\}$	A set of computation-storage resource pairs, where the computation resource in each pair can access the storage.

TABLE I: Description of basic mathematical notations used in the task-data co-scheduling optimization model.

to deduce two sets of binary numbers, i.e., A^{TC} and P^{DS}, while reaching the objective of maximum aggregated read and write bandwidth for ensuring minimum I/O time. Here, $A^{TC} = \{a_{ij}^{TC} : (a_{ij}^{TC} \in [0,1]) \wedge (i \in \{1, 2, \ldots, |T|\}) \wedge (j \in \{1, 2, \ldots, |C|)\}\}$. So, there can be $|T| \times |C|$ elements in A^{TC}, and a_{ij}^{TC} is 1 if task t_i is assigned to computation resource c_j, and 0 otherwise. $P^{DS} = \{p_{ij}^{DS} : (p_{ij}^{DS} \in [0,1]) \wedge (i \in \{1, 2, \ldots, |D|\}) \wedge (j \in \{1, 2, \ldots, |S|\})\}$. Hence, there can be $|D| \times |S|$ elements in P^{DS}, and p_{ij}^{DS} is 1 if data d_i is placed on s_j, and 0 otherwise. Hence, the objective of the basic optimization model can be defined as:

$$maximize \sum_{i=1}^{i=|D|} \sum_{j=1}^{j=|S|} p_{ij}^{DS} \times (b_j^r \times r_i + b_j^w \times w_i) \quad (1)$$

We first devise a binary integer linear programming [29] optimization strategy to solve this problem from a straightforward perspective. Unfortunately, this approach needs exponential time complexity and requires to satisfy quadratic constraints. According to our empirical experiments, it is not feasible for a variable space with even thousands of tasks and data.

b) Bipartite Graph Matching Formulation: The task-data inter-dependencies for a specific HPC workflow and computation-storage accessibility in an HPC system are usually invariable for a particular workflow campaign. We leverage this property and carefully rethink the co-scheduling optimization method by reducing the original multiple assignment problems into one. Besides, this technique allows us to transfer the task-data and computation-storage relationships from constraints' to variables' space; hence, we can avoid the cost of satisfying quadratic constraints. The idea is to construct a set of pairs of interrelated tasks and data by extracting the information from dataflow specification and treat this set as the set of agents in the assignment problem. We create another set of computation and storage resources pairs in the system information provision module. The storage system in a pair is accessible from the computation resource. Consequently, the environment is reduced to a bipartisan system where we need to assign a set of task-data pairs to a set of computation-storage resource pairs. Hence, we redefine the co-scheduling issue as a bipartite graph matching optimization problem with constraints.

While constructing the graph from the dataflow specification, as discussed in § IV-B1, we build a data structure, TD, to indicate the task-data dependency. Similarly for expressing computation-storage accessibility compiled from system information provision module, see § IV-B2, we define another set CS. TD and CS are defined in TABLE I. *DFMan* finds the optimal assignments for the elements of TD to that of CS by achieving the objective expressed in Equation 1, while satisfying the dataflow- and system-related constraints, see § IV-B3c.

In Fig. 4, we demonstrate an optimal assignment of task-data to computation-storage resources in the problem space defined in § III-A. We specify a bipartite directed graph where

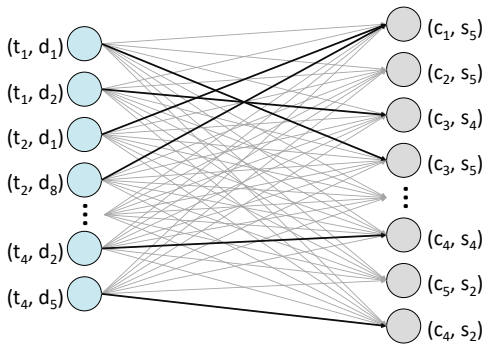


Fig. 4: Max bipartite graph matching to maximize the aggregated I/O bandwidth.

the edge represents an assignment. There can be multiple possible assignments, as indicated by the fade edges, but the optimizer selects the one that results in maximum aggregated I/O bandwidth. For instance, the pair of task-data (t_2, d_1) is assigned to computation-storage (n_1c_1, s_5), as it reaches the optimal objective value and satisfies all the constraints. We cannot use classic polynomial-time methods, such as Hungarian algorithm [30], for solving this optimization issue due to the dataflow- and system-related constraints that the problem needs to satisfy.

c) The Linear Programming Optimization Model: We solve the constrained bipartite graph matching optimization by a linear programming (LP) model. Note that we do not need quadratic constraints for task-data dependencies and computation-storage accessibility; hence, we can now formulate a model with simple linear constraints. We utilize the mathematical expressions discussed in TABLE I to define a set of continuous variables X to express the assignments of task-data to computation-storage. We avoid taking integer variables, as it is not time-efficient for large variable space [31]. This **variable set** is defined as,

$$X = \{x : x = (td_{jk}, cs_{lm}) \ \wedge \ 0 \leq x \leq 1 \ \wedge \ td_{jk} \in TD \ \wedge \ cs_{lm} \in CS\} \tag{2}$$

where x is 1 if t_j is assigned to c_l and d_k is assigned to s_m, 0 otherwise.

The **objective function** in Equation 1 is expressed using X as below.

$$maximize \sum_{x \in X} x \times (b^r_m \times r_k + b^w_m \times w_k) \tag{3}$$

The **constraints** of the optimization model are stated below.

1) Placing data instance to a storage system should not overflow the available storage capacity.

$$\sum_{x \in X} x \times d^s_k \leq s^c_m \tag{4}$$

2) Estimated I/O time should not exceed the task's walltime limit.

$$\sum_{x \in X} x \times d^s_k \times (r_k/b^r_m + w_k/b^w_m) \leq t^w_j \tag{5}$$

3) One task-data pair should be assigned to at best one storage system.

$$\sum_{cs_{lm} \in CS} x \times s_m \leq 1 \tag{6}$$

4) The number of tasks on the same topological level associated with a data instance should not exceed the maximum parallelism recommended for the storage system that holds the data.

$$\sum_{x \in X} x \times d^{rt}_k \leq s^p_m \quad and \sum_{x \in X} x \times d^{wt}_k \leq s^p_m \tag{7}$$

DFMan provides the optimal placement of all the data and one task associated with each data instance. After returning from the LP model, *DFMan* traverses through the topology of tasks and checks the associated data with the unassigned tasks. Then, it finds the available computation resources accessible from the storage that holds the data. Then, *DFMan* assigns the task such that no two tasks on a particular topological level are assigned to the same core. Finally, *DFMan* performs a sanity check to see if the storage is accessible from the computation resource for each task-data assignment. If any of those is not a valid co-scheduling scheme, *DFMan* falls back to default by moving the data to the global storage system.

*d) **Complexity Analysis**:* We employ interior-point method [32], based on Karmarkar's algorithm [33], to solve the LP optimization problem. The worst-case time complexity of this algorithm is $O(n^{3.5}L^2)$, where n is the dimension of the problem and L is the binary bits encoding length of the input data. In our use case, n can be specified by the sizes of C, S, T and D, which are the sets of computation resources, storage systems, tasks and data, respectively. In the worst-case scenario, if each computation resource accesses all the storage systems, and every task in a workflow is related to all the data instances, n can be denoted by $|C|\times|S|\times|T|\times|D|$. In practice, n is equal to $|A^{TC}| \times |P^{DS}|$ as described in § IV-B3a, which is much less than $|C| \times |S| \times |T| \times |D|$. The term L is a constant for fixed precision variable and objective function values [33]. Besides, the task to the computation resource assignment and fallback mechanism discussed in § IV-B3c is $O(|C|\times|S|\times|T|\times|D|)$ because we use auxiliary hashmap data structures with constant search time complexity to maintain the computation-storage and task-data relationships' information. Hence, the total time complexity of our optimization model is $O((|C| \times |S| \times |T| \times |D|)^{3.5} + (|C| \times |S| \times |T| \times |D|))$ or $O((|C| \times |S| \times |T| \times |D|)^{3.5})$ overall.

V. Implementation Details

A. Graph Representation of Dataflow

The graph representation of an HPC dataflow is expressed and handled by three Python classes in the *DFMan* prototype. Firstly, the `graph` class implements the basic graph manipulation techniques, such as finding all cycles in a graph, removing the cyclic patterns, and extracting the DAG.

Secondly, the `dag_parser` class has the logic of parsing the dataflow specification file and storing the information as an adjacency list via a hashmap of parent to a vector of children vertices. Besides, it extracts the information about the application to tasks mapping, relationships among tasks and data, and HPC dataflow structure-related metadata. Finally, the `dag_generator` class contains `graph` and `dag_parser` class instances as members and is the entry point for the optimizer to access the graph manipulation-related mechanisms. The LP optimizer fetches the information about the task and data dependencies using the APIs in this module.

B. System Information Hashmap

DFMan manages the system's information using an XML database handled by `cElementTree` package in Python. The system administrators have provision for accessing and updating the contents in the database. *DFMan*'s optimizer can use the API in this module to extract the data required for the optimization. *DFMan* maintains a collection of in-memory hashmaps to manage the system's data. Hence, the optimizer obtains O(1) access to the hashmap with information about the mapping to depict the accessibility of a certain storage system from a computation resource.

C. Optimization Model

We use Pyomo-6.1.2 [34] for solving the LP optimization problem. In particular, we leverage the `pyomo` Python package to use the APIs to define and execute the LP formulation. We implement the `optimizer` module that internally uses the `dag_generator` to extract the DAG from the user-defined dataflow and construct the variable space of optimization that represents the task and data dependency. The `optimizer` also utilizes the `system_info_db` module to build the computation and storage accessibility-related variables. Finally, it outputs the task-data co-scheduling strategies that provide tasks to computation resources and data to storage placement policies for an optimal aggregated I/O bandwidth in the HPC workflow under consideration.

D. Interaction with HPC Resource Managers

The optimization model works on pre-allocated computation and storage resources for the corresponding workflow and reruns when the allocation changes. *DFMan* deals with the output from the model on two fronts. Firstly, it applies the task to computation resource assignment strategies by constructing MPI `rankfiles` [35] for each application involved in the workflow. These rankfiles are parameterized to the application execution commands in the batch scheduling scripts for the workflow. Hence, any HPC resource manager supporting MPI, such as LSF, SLURM, Flux, etc., can be used effectively. Secondly, *DFMan* materializes the data to storage resource placement policies by configuring the applications' data access. We modify the source of each application workflow for *DFMan*'s evaluation. We will utilize a middleware [36] to intercept I/O requests and automatically redirect data to proper storage systems in the future.

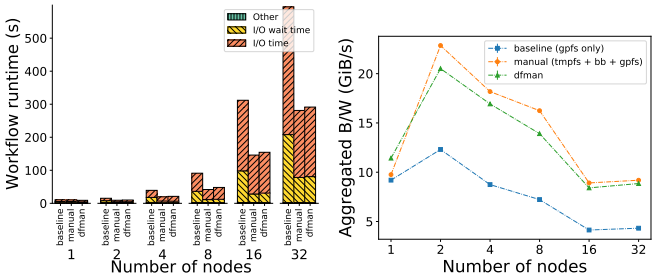
VI. Experimental Evaluation

We evaluate the task-data co-scheduling policies from *DFMan* using synthetic and real HPC application workflows on the Lassen supercomputer [19]. Lassen is an IBM Power9 machine with 44 cores and 256 GB memory per node. In our experiments, we use the three layers of storage systems that Lassen has, such as a 24 PiB global IBM General Parallel File System (GPFS), node-local 256 GiB per node ram disk (tmpfs), and node-local 1 TiB per node IBM burst buffer (BB). We leverage Wemul [6] to generate synthetic HPC workflow workloads. We further experiment with two application workflows: Hurricane 3D on CM1 and NGC3372 on Montage; and two HPC dataflow kernels: HACC I/O and MuMMI I/O. We compare I/O time and bandwidth against a baseline, where workflow is unaware of the task-data dependencies and system's information. It always uses the globally accessible storage system, and the task assignment depends on the resource manager's scheduling policy. Then, we manually tune the workflow to leverage the system's maximum possible benefits. Finally, we apply the automatic task-data co-scheduling strategies from *DFMan* to run the workflows.

A. Synthetic Workflows

1) Workloads: We generate two types of synthetic dataflow workloads solely performing I/O operations. The first type of dataflow (type 1) has a three-stage cyclic workflow. Each stage creates producer-consumer data dependency, and the data access pattern is posed alternatively as file-per-process and shared file access on every stage. The output data of the third stage are fed to the first stage with non-strict dependency for creating the cycle. We extract the DAG from type 1 workflow and run it for ten iterations. We increase the number of tasks per stage with increasing nodes. The second type of workload (type 2) represents a best-case scenario, where all the stages consist of file-per-process data access patterns. We keep the number of resources fixed and vary either the width or height of the dataflow for further synthetic experimentation. We report the aggregated I/O bandwidth and total runtime for read and write across all the stages. The runtime includes I/O time and I/O wait time, i.e., the time that the consumer task waits after being scheduled until the data is produced. The time taken by the resource manager processing, DAG extraction, etc., is referred to as "other".

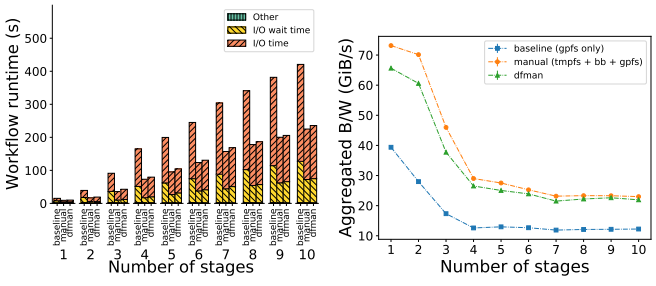
2) Three-stage Workflow with Cycle: We demonstrate a runtime breakdown for type 1 workload in Fig. 5(a). In this case, each data file is 4 GiB, and the total data size reaches up to 2 TiB for 32 nodes. We allocate 300 GB burst buffer space per node and allow 100 GB ram disk (tmpfs) space per node. We observe that the total workflow runtime improves when we incorporate *DFMan* task-data co-scheduling strategies. The benefit comes from taking advantage of the dataflow structure and systems' information, and utilizing the node-local faster storage devices, along with the global PFS. In particular, automatic scheduling policies of *DFMan* help the workflow achieve 51.4% runtime improvement compared to the baseline.



(a) Three-stage workflow (b) Three-stage workflow

Fig. 5: Benefits of *DFMan*'s automatic task-data co-scheduling on cyclic workflows.

In the case of manual workflow tuning, keeping file-per-process data on tmpfs and shared files in GPFS, we can achieve 53.9% improvement. We notice a significant improvement in I/O wait time, i.e., 31.3% for baseline to 19.9% and 19.3% for manual and automatic by *DFMan*, respectively. As shown in Fig. 5(b), the bandwidth does not scale well for the workflow under consideration with increasing resources for all the cases. However, we observe significant aggregated bandwidth improvement in *DFMan* scheduling strategies, i.e., 1.74× the baseline bandwidth. The manual tuning achieves up to 1.85× bandwidth improvement on average.



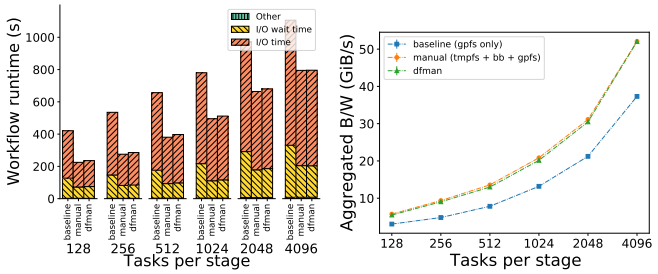
(a) Variable stages (b) Variable stages

Fig. 6: *DFMan*'s automatic task-data co-scheduling on variable number of stages.

3) Variable Dataflow with Fixed Resources:

a) Varying the Number of Stages: We evaluate *DFMan*'s co-scheduling policies by varying the structure of type 2 workflow while keeping the number of nodes fixed to 16 with eight processes per node. We allocate 100 GB burst buffer space and allow 100 GB tmpfs space to be used per node. As shown in Fig. 6(a), we change the number of stages in the dataflow from one to ten and report the runtime breakdown. We observe significant runtime improvement of 50.6% using *DFMan*, and we achieve 53.7% in the manual-tuning case. The aggregated I/O bandwidth lessens with an increasing number of stages as we reach the maximum capacity of the node-local storage systems and end up using GPFS. Nevertheless, we observe 1.91× bandwidth improvement in *DFMan*'s case, where it is 2.12× the baseline bandwidth via manual tuning.

b) Varying the Number of Tasks per Stage: We keep the number of stages fixed to ten and change the number of tasks per stage; hence, we vary the width of the dataflow to experiment further. We keep the resources fixed to 16 nodes and eight processes per node. As depicted in Fig. 7(a), we reach the maximum capacity for node-local storage systems for tasks per node more than 512. Hence, we observe 36.6% runtime improvement using *DFMan*'s policies, where the manual tuning case runtime improvement is 34.9%. As shown in Fig. 7(b), the bandwidth scales well with increasing tasks per node and reaches up to 52.03 GiB/s for 4096 tasks per stage. The aggregated bandwidth improvement is 1.49× and 1.52× the baseline with *DFMan* and manual tuning, respectively.

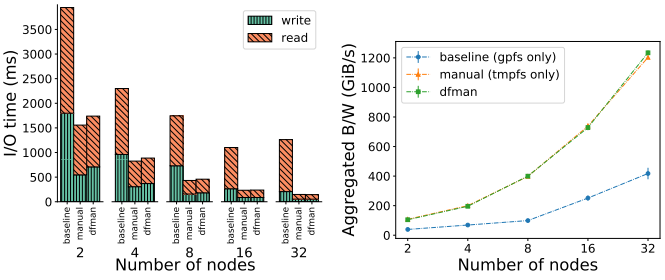


(a) Variable tasks per stage (b) Variable tasks per stage

Fig. 7: *DFMan*'s automatic task-data co-scheduling on increasing number of tasks per stage.

B. HPC Application Workflows and Dataflow Kernels

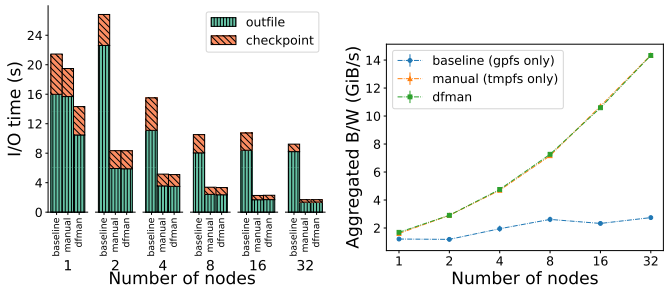
1) HACC I/O: Hardware/Hybrid Accelerated Cosmology Code (HACC) [37] is an N-body cosmological simulation framework. It simulates the structures' formation in collisionless fluids in the universe. HACC I/O is an I/O kernel and popular HPC I/O benchmark that captures the I/O patterns of HACC framework. We run it with a file-per-process checkpoint/restart pattern to evaluate *DFMan* in a simple workflow scenario. As depicted in Fig. 8, the automatic task-data co-scheduling suggestions from *DFMan* assist HACC I/O to achieve I/O performance almost the same as that attained by manual data management. HACC I/O chooses to use node-local tmpfs suggested by *DFMan*'s optimizer achieving 2.96× more bandwidth than the baseline, and the total I/O time decreases up to 11.44% of baseline I/O time.



(a) Total I/O time (b) Aggregated read and write bandwidth

Fig. 8: I/O time and bandwidth improvement in HACC I/O using *DFMan*.

2) Hurricane 3D on CM1: Cloud Model 1 (CM1) [17] is an MPI and OpenMP-enabled data-intensive application for atmospheric research. It runs a three-dimensional non-linear numerical model for analyzing phenomena like thunderstorms, hurricanes, etc. We run a model called Hurricane 3D that produces mainly two types of files in a user-defined frequency, i.e., file-per-process output files and node-per-process checkpoint files. As shown in Fig. 9, CM1 application workflow execution using *DFMan*'s co-scheduling policies achieves similar performance as manual-tuning. It maximizes the aggregated I/O bandwidth by up to 5.42× more than the baseline. *DFMan* chooses node-local tmpfs on Lassen to store both output and checkpoint files. The I/O time decreases up to 19.08% of the baseline I/O time.

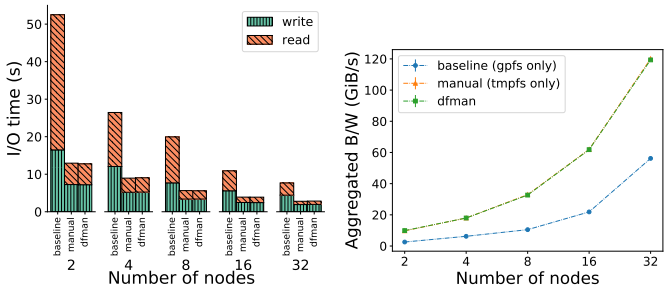


(a) Write time breakdown (b) Write bandwidth

Fig. 9: I/O performance improvement in Hurricane 3D on Cloud Model 1 (CM1) using *DFMan*.

3) Montage NGC3372 Mosaic Formation: Montage is a popular astronomical image mosaic engine that employs multiple applications to assemble flexible image transport system (FITS) images into human-perceivable image mosaic. It is used to visualize various parts of the galaxies, such as the Milky Way, by running a chain of parallel or sequential applications that share data. Hence, it creates complex dataflow and is an interesting candidate for evaluating the efficacy of *DFMan*'s optimization strategies. We build a parallel workflow using Montage applications to generate grayscale image mosaic for Carina Nebula (NGC3372) in Milky Way Galaxy. It reads FITS files in parallel and feeds the data for further information extraction and final visualization generation via a six-stage dataflow.

We run the Montage NGC3372 workflow experiments on Lassen and profile the I/O behavior of all the applications on Montage using Recorder [38] tracing tool. As shown in Fig. 10, *DFMan*'s optimization technique achieves almost the same I/O bandwidth as manual workflow tuning. In particular, *DFMan* automatically chooses faster node-local tmpfs on Lassen over GPFS and minimizes the resource contention due to data sharing. Besides, it collocates the tasks in a set of producer and consumer applications to lessen data movement through shared storage systems. Due to *DFMan*'s strategy, aggregated read and write I/O bandwidth in the Montage NGC3372 workflow scales smoothly 9.89 GiB/s to 119.36 GiB/s for 2 to 32 nodes on Lassen and achieves 2.12×

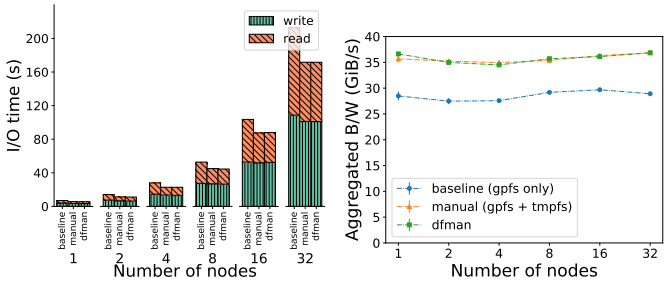


(a) Total I/O time (b) Aggregated read and write bandwidth

Fig. 10: Benefits of using *DFMan* for Montage NGC3372 workflow.

more than the baseline bandwidth. Consequently, the total I/O time drops to 37.15% of baseline.

4) MuMMI I/O: Multiscale Machine-learned Modeling Infrastructure (MuMMI) [4] is developed as a part of the Cancer Moonshot Pilot 2 project for assisting cancer diagnosis research. MuMMI executes a large-scale HPC workflow to simultaneously investigate the interaction between RAS protein and cell membrane in experimentally observable (macro) and molecular-level (micro) scales. It deals with unique data management challenges posed by petabytes of data generated during the simulation, complex multi-stage dataflow, and a cyclic feedback mechanism. We develop MuMMI I/O, an emulated version of the I/O behavior in MuMMI using Wemul [6], to perform I/O-focused detailed experimentation and address the data management issues using *DFMan*'s strategies.



(a) Total I/O time (b) Aggregated read and write bandwidth

Fig. 11: I/O time and bandwidth improvement using *DFMan* in the dataflow emulation of MuMMI.

We run MuMMI I/O with Recorder on Lassen and perform weak scaling experiments. As demonstrated in Fig. 11, *DFMan*'s scheduling policies assist MuMMI to achieve similar aggregated I/O bandwidth as that by employing manual data management strategies. *DFMan* suggests using node-local tmpfs to manage data production and consumption in the micro-scale applications and collocate the simulation and analysis tasks on the same node. *DFMan*'s task-data co-scheduling policies help MuMMI I/O achieve up to 1.29× the baseline aggregated I/O bandwidth and 21.28% improved I/O time.

VII. Related Work

a) Data Management in HPC Systems: Data migration among and retrieval from storage systems has been a focused research area in HPC [39], [40]. For instance, Data Elevator is an I/O abstraction library to intercept HDF5 I/O requests and move data among burst buffers and PFSs [41]. Hermes offers optimized data placement policies for maximizing bandwidth and data locality [42]. Univistor improved the ease of use by providing a unified single mount point for the entire storage stack with I/O acceleration mechanisms [43]. Besides, there have been projects on innovative and flexible data representation for enabling I/O asynchrony [44]. Tanaka et al. utilized multi-constraint graph partitioning in Pwrake, and leveraged the locality information provision feature in Gfarm FS for data movement minimization [45]–[47]. However, these data management strategies do not consider intuitive task-data co-scheduling by using the dataflow- and system-related information simultaneously.

b) Data Management as an Optimization Problem: Optimization techniques are generally applied to manage heterogeneous computation resources. Alsched [14] and TetriSched [48] formulate optimization methods to schedule tasks using a bin-packing algorithm and mixed-integer linear program (MILP), respectively. AlloX addresses the non-polynomiality issue posed by solving combinatorial task scheduling problems using MILP and proposes a fair and efficient scheduling optimization policy through formulating and solving a min-cost bipartite graph matching problem [13]. Recently, Adaptive Scheduling Architecture (ASA) leverages reinforcement learning to optimize computation resource usage [12]. Besides, recent works have taken place on application-specific scheduling schemes specially designed for workloads like deep learning training [49]. Utilizing intelligent analytical methods can be effective in data management [50]. Netco solves a MILP constrained by network bandwidth and storage capacity to maximize the number of jobs reaching deadline and prefetch data from slow remote storage to faster local storage in the cloud [31]. BBSched formulates job scheduling and data placement on burst-buffer storage as a multi-objective optimization problem and uses a genetic algorithm to find optimal resource utilization [7].

Nevertheless, most optimization strategies for data-intensive task scheduling are developed for cloud computing or data analytics frameworks. On the other hand, the data management projects on HPC systems mainly provide ease of use through I/O abstraction with various I/O acceleration methods or focus on using only one type of storage system for I/O improvement. None of these address dataflow management by lessening the semantic gap between task scheduling and data allocation by considering the dataflow information in HPC workflows and the complexity of involving multiple modules in the storage stack. To the best of our knowledge, our work uniquely addresses this issue by devising graph-based optimization techniques to automate task-data co-scheduling and improve HPC workflow I/O performance.

VIII. Future Opportunities

A few limitations are present in *DFMan* that give us many opportunities for future work. Firstly, the optimizer in *DFMan* is an offline scheduler. If the workflow is dynamic where the number of stages and width of the workflow changes in runtime, the optimizer needs this updated information from the user. *DFMan* depends on user input for getting the information about the task and data dependencies in the workflow. In the future, we will work on incorporating automation to extract useful information about the dataflow using I/O tracing and interception tools like Recorder [38]. Secondly, the system information provision module does not automatically update the information while the workflow runs. Multiple concurrent workflows using *DFMan* can create consistency issues in the capacity detection of the storage stack. Dynamically using functionalities in modern resource managers [22] to detect the system's current status in finer detail can overcome this limitation. Lastly, in exceptional cases, when the task-data co-scheduling scheme is deemed invalid, *DFMan* reallocates the data to the globally accessible storage system. Hence, this fallback mechanism will not work if a cluster does not have global storage. Fortunately, most HPC systems at least have one global network-attached file system.

IX. Conclusion

The performance hindrance in HPC workflows due to rapid data movement among workflow applications and its tasks often hampers the scientific research momentum. Solving this issue requires careful observation and understanding of dataflow and HPC storage stack through manual effort. In this paper, we introduce *DFMan* that offloads the data management responsibilities from the workflow research and development cycle. The novel technique of using classic graph algorithms in solving the NP-hard task-data co-scheduling problems facilitates the optimization process and helps it execute feasibly.

Applying *DFMan*'s optimization strategies demonstrates promising I/O bandwidth maximization by automatically prioritizing node-local storage systems over shared ones, which alleviates resource contention. We establish the theoretical basis of the optimizer and evaluate its efficacy using data-intensive HPC workflows. In the future, we will extend this work by applying *DFMan*'s scheduling policies on a larger scale and critical scientific application workflows with more complex data dependencies. We will introduce further automation by upgrading *DFMan* to an online task-data co-scheduler for handling more dynamic scenarios.

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-827797. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program. This work is supported in part by the National Science Foundation awards 1561041,

1564647, and 1763547, and has used the NoleLand facility that is funded by the U.S. National Science Foundation grant CNS-1822737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] R. Ross, L. Ward, P. Carns, G. Grider, S. Klasky *et al.*, "Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery," 2019.

[2] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[3] "The Compact Muon Solenoid experiment at CERN," http://cms.cern.

[4] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale *et al.*, "A massively parallel infrastructure for adaptive multiscale simulations: Modeling ras initiation pathway for cancer," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY: ACM, 2019.

[5] "Sierra," https://hpc.llnl.gov/hardware/platforms/sierra.

[6] F. Chowdhury, Y. Zhu, F. Di Natale, A. Moody, E. Gonsiorowski, K. Mohror, and W. Yu, "Emulating i/o behavior in scientific workflows on high performance computing systems," in *5th International Parallel Data Systems Workshop*. IEEE, 2020.

[7] Y. Fan, Z. Lan, P. Rich, W. E. Allcock, M. E. Papka, B. Austin, and D. Paul, "Scheduling beyond cpus for hpc," in *28th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY: ACM, 2019, pp. 97–108.

[8] "Maestro Data Orchestration," https://www.maestro-data.eu.

[9] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning," in *48th International Conference on Parallel Processing*. New York, NY: ACM, 2019, pp. 80:1–80:10.

[10] F. Chowdhury, J. Liu, Q. Koziol, T. Kurth, S. Farrell, S. Byna, and W. Yu, "Initial characterization of i/o in large-scale deep learning applications," 2018.

[11] M. Fatih Aktas, G. Haldeman, and M. Parashar, "Scheduling and flexible control of bandwidth and in-transit services for end-to-end application workflows," *Future Gener. Comput. Syst.*, vol. 56, no. C, pp. 284–294, Mar. 2016.

[12] A. Souza, K. Pelckmans, D. Ghoshal, L. Ramakrishnan, and J. Tordsson, "Asa - the adaptive scheduling architecture," in *29th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY: ACM, 2020, p. 161–165.

[13] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "Allox: Compute allocation in hybrid clusters," in *15th European Conference on Computer Systems*. New York, NY: ACM, 2020.

[14] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *3rd ACM Symposium on Cloud Computing*. New York, NY: ACM, 2012.

[15] "Generalized assignment problem," https://en.wikipedia.org/wiki/Generalized_assignment_problem.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[17] "Cloud Model 1," https://www2.mmm.ucar.edu/people/bryan/cm1.

[18] R. Sakellariou, H. Zhao, and E. Deelman, "Mapping workflows on grid resources: Experiments with the montage workflow," in *Grids, P2P and Services Computing*, F. Desprez, V. Getov, T. Priol, and R. Yahyapour, Eds. Boston, MA: Springer US, 2010, pp. 119–132.

[19] "Lassen," https://hpc.llnl.gov/hardware/platforms/lassen.

[20] "SLURM," https://en.wikipedia.org/wiki/Slurm_Workload_Manager.

[21] "IBM Spectrum LSF," https://www.ibm.com/docs/en/spectrum-lsf.

[22] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein *et al.*, "Flux: overcoming scheduling challenges for exascale workflows," in *2018 IEEE/ACM Workflows in Support of Large-Scale Science*. IEEE, 2018, pp. 10–19.

[23] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-aware, utility-based job scheduling on blue, gene/p systems," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–10.

[24] "Pegasus," https://pegasus.isi.edu.

[25] "MaestroWF," https://github.com/LLNL/maestrowf.

[26] "Cylc," https://cylc.github.io.

[27] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *SIGMOD Rec.*, vol. 34, no. 3, pp. 44–49, Sep. 2005.

[28] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. USA: Prentice-Hall, Inc., 1982.

[29] H. P. Williams, *Logic and Integer Programming*, ser. Logic and Integer Programming. Springer, 2009, vol. 130. [Online]. Available: https://doi.org/10.1007/978-0-387-92280-5

[30] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[31] V. Jalaparti, C. Douglas, M. Ghosh, A. Agrawal, A. Floratou, S. Kandula, I. Menache, J. S. Naor, and S. Rao, "Netco: Cache and i/o management for analytics over disaggregated stores," in *ACM Symposium on Cloud Computing*. New York, NY: ACM, 2018, pp. 186–198.

[32] I. Dikin, "Iterative solution of problems of linear and quadratic programming," in *Doklady Akademii Nauk*, vol. 174, no. 4. Russian Academy of Sciences, 1967, pp. 747–748.

[33] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *16th annual ACM symposium on Theory of computing*, 1984, pp. 302–311.

[34] "Pyomo," http://www.pyomo.org.

[35] "IBM Spectrum MPI," https://www.ibm.com/docs/en/smpi.

[36] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W. Yu, "Direct-fuse: Removing the middleman for high-performance fuse file system support," in *8th International Workshop on Runtime and Operating Systems for Supercomputers*. New York, NY: ACM, 2018.

[37] "HACC I/O," https://github.com/glennklockwood/hacc-io.

[38] "Recorder," https://github.com/uiuc-hpc/Recorder.

[39] P. Subedi, P. Davis, S. Duan *et al.*, "Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018.

[40] D. Ghoshal, L. Ramakrishnan, and D. Agarwal, "Dac-man: Data change management for scientific datasets on hpc systems," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018.

[41] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, N. Keen *et al.*, "Data elevator: Low-contention data movement in hierarchical storage system," in *IEEE 23rd International Conference on High Performance Computing*. IEEE, 2016, pp. 152–161.

[42] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A heterogeneous-aware multi-tiered distributed i/o buffering system," in *27th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY: ACM, 2018, pp. 219–230.

[43] T. Wang, S. Byna, B. Dong, and H. Tang, "Univistor: Integrated hierarchical and distributed storage for hpc," in *2018 IEEE International Conference on Cluster Computing*, 2018, pp. 134–144.

[44] A. Kougkas, H. Devarajan, J. Lofstead, and X.-H. Sun, "Labios: A distributed label-based i/o system," in *28th International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY: ACM, 2019, pp. 13–24.

[45] M. Tanaka and O. Tatebe, "Workflow scheduling to minimize data movement using multi-constraint graph partitioning," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, May 2012, pp. 65–72.

[46] "Gfarm File System," https://en.wikipedia.org/wiki/Gfarm_file_system.

[47] M. Tanaka, O. Tatebe, and H. Kawashima, "Applying pwrake workflow system and gfarm file system to telescope data processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 124–133.

[48] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *11th European Conference on Computer Systems*. New York, NY: ACM, 2016.

[49] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *Fifteenth European Conference on Computer Systems*. New York, NY: ACM, 2020.

[50] A. Nachman, G. Yadgar, and S. Sheinvald, "Goseed: Generating an optimal seeding plan for deduplicated storage," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 193–207.