DTIO: Data Stack for AI-driven Workflows

Keith Alex Bateman
Illinois Institute of Technology
Chicago, IL, USA
kbateman@hawk.iit.edu

Neeraj Rajesh
Illinois Institute of Technology
Chicago, IL, USA
nrajesh@hawk.iit.edu

Jaime Cernuda
Illinois Institute of Technology
Chicago, IL, USA
jcernudagarcia@hawk.iit.edu

Luke Logan
Illinois Institute of Technology
Chicago, IL, USA
llogan@hawk.iit.edu

Bogdan Nicolae
Argonne National Laboratory
Lemont, IL, USA
bnicolae@anl.gov

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@mcs.anl.gov

Xian-He Sun
Illinois Institute of Technology
Chicago, USA
sun@iit.edu

Anthony Kougkas
Illinois Institute of Technology
Chicago, IL, USA
akougkas@iit.edu

Abstract

HPC, Big Data Analytics, and Machine Learning have become increasingly intertwined as popular models such as LLMs and Diffusion Models have been driving discovery in scientific fields. However, each of these domains has its own storage infrastructure with unique I/O interfaces and storage systems, requiring feature sets that are often incompatible. Users with experience in one domain lack the expertise to change their applications to match the data stacks of the other domains, necessitating expensive conversions. There is a need for a transparent solution for the unification of disparate data stacks for the triple convergence of HPC, Big Data, and ML that can provide the required functionality while achieving higher performance. To better support converged HPC, Big Data, and ML workflows, this paper proposes DTIO, a scalable I/O runtime that unifies the disparate I/O stack for modern scientific ML workflows. DTIO utilizes a unique DataTask abstraction to express the movement of data, its ordering, and its dependencies on other data as a task. DTIO achieves a unification of scientific and ML workflows by utilizing intelligent mapping of interfaces, and automatically determines the best method to relate their unique semantics. DTIO's online translation with DataTask caching can improve performance by 49.6% compared to offline translation methods. DTIO also offers numerous optimizations, such as asynchronous I/O and aggregation.

CCS Concepts

• **Computing methodologies** → *Cooperation and coordination*; • **Computer systems organization** → **Distributed architectures**.

Keywords

Task Systems, Data Stacks, Systems for AI Workflows

SSDBM 2025, Columbus, OH, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1462-7/25/06
https://doi.org/10.1145/3733723.3733736

ACM Reference Format:
Keith Alex Bateman, Neeraj Rajesh, Jaime Cernuda, Luke Logan, Bogdan Nicolae, Franck Cappello, Xian-He Sun, and Anthony Kougkas. 2025. DTIO: Data Stack for AI-driven Workflows. In *The International Conference on Scalable Scientific Data Management 2025 (SSDBM 2025), June 23–25, 2025, Columbus, OH, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3733723.3733736

1 Introduction

HPC, Big Data Analytics, and Machine Learning (ML) are increasingly converging [14]. While Big Data and ML have long operated together in applications such as recommendation systems and fraud detection, the rise of tightly coupled patterns, especially in ML, has prompted an acceleration of this convergence with HPC systems. For scientific applications, this trend presents an unprecedented opportunity. For example, recent work has showcased the value of AI-enabled HPC workflows, replacing traditional heuristic approaches in tasks like adaptive workflow steering [17]. Conversely, the rise of surrogate models enables the acceleration of traditional simulations by orders of magnitude [1, 33]. Revolutionary progress in ML natural language processing and reasoning capabilities is beginning to empower scientists with versatile research assistants. Transformer capabilities are swiftly evolving and are close to or even surpassing human levels in many advanced tests (ARC AGI, Frontier Math, GPQA). These capabilities are emerging in scientific domains too, as illustrated by science-oriented LLMs such as Google AI co-scientist.

Converged workflows that combine HPC, Big Data and ML tasks generate massive datasets that are continuously increasing exponentially in both complexity and volume. To obtain the most value out of these massive datasets, different types of tasks need to access them, compute intermediate results and share the intermediate results at large scale. This aspect is particularly challenging and keeps increasing in complexity as the I/O runtimes and data management solutions employed by different communities keep growing organically and keep diverging from each other.

Challenges: Specifically, HPC, Big Data Analytics, and Machine Learning feature a variety of data formats, data models, and semantics, as well as I/O libraries that implement them. Table 1 summarizes several aspects of interest. Fundamental differences exist in the unit of data and APIs, while other differences concern aspects such as: in-place updates vs. new objects to match computational paradigms (imperative vs. functional) and/or support required features (versioning, provenance tracking), structured vs. unstructured data (the latter being especially popular in the Big Data Community) and the implications this has on metadata management. From a non-functional perspective, I/O patterns and parallelism considerations have influenced HPC applications to use more regular I/O patterns that involve large, collective I/O operations to achieve scalability. Big Data Analytics applications also tend use large but less coordinated I/O operations. ML applications feature the most difficult patterns [14], as they operate with irregular I/O patterns that often involve small I/O operations (e.g., needed for sampling) at large scale, which complicates concurrency control.

Category	Big Data	HPC	ML
Data unit	Objects	Files	DataFrames
Update	New object	In-place	New table
APIs	REST/SOAP	PFS/POSIX	Pythonic
Metadata	Custom	Fixed	Fixed
Structure	Unstructured	Structured	Both
I/O Patterns	Lg/Irregular	Lg/Regular	Sm/Irregular
I/O Parallelism	Irregular	Collectives	Both

Table 1: Interoperability at the intersection of Big Data Analytics, HPC and ML: Categories of interest.

Second, hybrid workflows that combine Big Data Analytics, HPC, and ML tasks often rely on producer-consumer patterns that need to exchange inputs and outputs. Some are ephemeral and do not need to be persisted. Therefore, conversions and translations between data formats and access models need to connect data sources and sinks in real time, as directly as possible (i.e., direct links between compute nodes) and need to support streaming (i.e., to enable partial processing before all data is available). Therefore, it is not enough to offer simple read/write access semantics. Instead, there is a need to capture more information about data sources and sinks, access pattern and intent, as well as persistency requirements.

Third, the differences in the computational paradigms corresponding to each domain resulted in different benefits and limitations. Big Data is more resilient [26] due to stateless and loosely coupled tasks that synchronize through persistent states shared on the storage subsystem, the latter being subject to synchronization bottlenecks under concurrent access. HPC features mostly tightly coupled tasks, and this influenced the interactions with the storage (e.g. MPI I/O collectives to match MPI computational collectives). Despite the potential for scalability [26], the underlying parallel file systems are based on the aging POSIX model and feature limited aggregated I/O bandwidth. The ML ecosystem is rapidly evolving and incorporates ideas from both the Big Data Analytics and HPC ecosystem of I/O libraries. Despite its flexibility and opportunities for optimization, this increases the complexity and fragility of the storage subsystem. Thus, there is a need to combine the best practices from each of these domains.

Limitations of state-of-art: Unfortunately, the Big Data Analytics, HPC and ML communities have developed their own rich ecosystem of data management frameworks and I/O runtimes independently of each other. Thus, the diversity of data formats and access models leads to a major inter-operability limitation that remains largely unaddressed by state-of-art. Users of hybrid workflows typically implement their own ad-hoc conversion between different data formats, and then use separate I/O runtimes optimized for each data format and type of task in the workflow. This leads to large conversion overheads and expensive data copies, which leads to suboptimal use of storage resources. To avoid the complexity of direct communication between the tasks, storage repositories such as parallel file systems and object stores (e.g., Amazon S3) are typically used as proxies, at the expense of high (de)serialization overheads and low I/O bandwidth, especially under concurrency. Such repositories are not optimized for ephemeral data or streaming semantics that are needed by producer-consumer patterns, which leads to unnecessary serialization of computations and missed opportunities for overlapping computations and I/O.

Contributions: To address the aforementioned limitations, we propose DTIO, a scalable I/O runtime that unifies the disparate data stack for modern scientific and ML workflows. To unify diverse interdisciplinary data stacks, DTIO utilizes a unique DataTask abstraction, which expresses data content, user-intended data transformations, and dependencies (e.g., data sources and data sinks or consistency guarantees like read-after-write). The DataTask enables DTIO to flexibly support conversion of data across a variety of high- and low-level I/O interfaces, while enabling key optimizations that can maintain or improve I/O performance. DTIO's contributions are as follows:

(1) **DataTask Abstraction**: A unique representation of data that encapsulates content, dependencies, and intent.
(2) **Unification across Big Data Analytics, HPC and ML**: DataTasks inherently permit a wide variety of data format transformations and storage backends in order to satisfy the diverse requirements of triple-converged HPC, Big Data, and ML workflows.
(3) **Optimized I/O Pipelines**: DataTasks enable a new accelerated I/O pipeline that includes optimizations such as aggregations, asynchronous lazy loading, and accelerated I/O resolution.

2 Motivating Scenario

To illustrate the challenges presented in § 1, we consider the case of PtychoNN, a deep learning ptychography workflow that solves the data inversion problem in order to perform imaging beyond the resolution limits of typical x-ray optics [7]. PtychoNN combines aspects of all three domains: Big Data Analytics, HPC and ML: it features a Big Data Analytics pre-processing stage (Map-Reduce pattern) that takes the raw images from instruments, applies various filters and prepares them in a common format (HDF5) to be reused by multiple applications (including PtychoNN). Then, an HPC stage running on an HPC machine reads the common format and converts the HDF5 files to *npz* and *npy* format that is popular in the Python ecosystem. Both reads and writes are sequential. From there, a distributed ML training (data parallel) reads the npz and npy files concurrently (random small I/O access) and feeds it to a PyTorch training pipeline, which ultimately produces checkpoints of the trained model (HDF5). The lack of interoperability between these formats and access patterns results in several challenges: (1) the three stages happen sequentially despite opportunities to streamline them as a pipeline; (2) a parallel file system is used to store intermediate results instead of on-the-fly conversions and direct communication between the tasks; (3) I/O operations are blocking instead of asynchronous, which limits the opportunity to overlap I/O with computations. We aim to solve such challenges.

3 Background and Related Work

Data Formats: Scientific data is often stored in higher-level formats such as HDF5 [16], Zarr [24], and numpy arrays [31]. Some useful features of these formats include multidimensional data selection and I/O on non-character data. HDF5 in particular has a wide variety of configurations for improving performance, such as compression, metadata compaction, and caching [9]. DTIO aims to provide efficient interopability between diverse formats to improve the performance of hyperconverged workflows.

Data Format Translation: Conversions between different data formats exist in various forms. Pandoc is a tool to convert between various document formats such as PDF, Microsoft Word, Latex, and more [20]. Bioconvert maps various forms of life science data, including CSV, YAML, BAM, and FastQ [4]. Typically, data format conversions are performed offline by reading in the data in one format, converting it, and writing out in the alternative format. A data format translator with numerous formats will tend to use an intermediate representation in order to simplify the in-memory translation [8]. On-the-fly translation is similar, but uses the in-memory representation instead of writing back to disk. On-the-fly translation is performed directly within the application workload, which can be more efficient but also presents an overhead. On-the-fly translation is effectively a data transformation, and as such is often performed on real-time data [30]. Unlike DTIO, most on-the-fly translation is not transparent to the user, as it requires the application to use both interfaces being translated in its code.

I/O Task Systems: I/O tasks have been explored in the literature, usually with the intent of leveraging task systems to improve performance rather than creating a standard for task-based I/O. Labios [18] provides a task-based storage system which uses tasks to asynchronously perform the requested I/O operation on one of a pool of workers. Labios is the basis for DTIO, but DTIO differs due to its focus on providing for the triple convergence of ML, Big Data, and HPC. ExaHDF5 [3] is an advanced high-performance I/O task system designed for large-scale scientific applications that incorporates asynchronous I/O via a VOL layer which leverages the Argobots [28] task system. DataStates [25] is an I/O task system designed to manage large datasets efficiently in parallel processing environments. It is a data model rather than a runtime, but is comparable to DTIO in its expression of data lineage as a series of states, each containing content and metadata. Compared to related task systems, DTIO features expanded interface interception and task definitions.

Storage Bridging: Some work has been done to provide a bridge which allows transparent access to different storage systems. IRIS [19] provides a middleware library that bridges the gap between file systems and object stores. SciDP [13] provides a similar bridge for HPC and big data applications. NIOBE [12] bridges the same gap, but as a transparent service instead of a library, and therefore it enables asynchronous I/O abstractions. Typically, transparency and asynchronicity are considered valuable aspects in a bridge, as the bridge can be viewed as a sort of I/O task manager which decides where to place I/O operations and presents the data associated with them in a manner that users can access them across interfaces. Unlike Iris and NIOBE, DTIO uses the intermediary DataTask representation, which gives it more flexibility when it comes to intercepting and mapping a variety of data formats. DTIO also provides more optimizations for the data, such as advanced caching and prefetching capabilities.

Summary: Despite a large variety of I/O runtimes and tools in the Big Data Analytics, HPC and ML, interoperability without sacrificing performance and scalability is still a major challenge. To our knowledge, we are the first to explore the unification of various I/O optimizations from HPC, Big Data, and AI ecosystems and their applicability to efficient data interoperability in hyperconverged workflows.

4 A Novel DataTask I/O Stack Design

In this work, we present a new scalable I/O system that aims to transparently unify the disparate data stacks for modern HPC, Big Data, and ML workflows. This system is designed around the DataTask, a novel data structure that encapsulates the content, intent, and dependencies of data. Unlike conventional approaches that treat I/O as passive and arbitrary data movement, DataTasks are powerful enough to represent data from a wide variety of I/O formats across HPC, Big Data, and ML while enhancing overall I/O performance through composable, data-intensive operations executed transparently near the data itself. This includes transparent data format conversion operators required for efficient data interoperability. DataTasks can be orchestrated by higher-level scheduling algorithms to improve the position and composition across DataTasks in the system while considering the computational demands and interference caused by them. For example, aggregating smaller DataTasks into larger ones to reduce latency penalties caused by complex metadata operations on backend storage. This section details the structure and properties of DataTasks that can be used to build efficient I/O stacks.

The design of a DataTask-based I/O system is built upon the following key objectives:

(1) **Transparent**: The DataTask abstraction should transparently and conveniently capture user intent and data.
(2) **Lightweight**: Unification of diverse I/O interfaces should have minimal overhead to applications.
(3) **Scalable**: The system should scale for increasing numbers of producer or consumer clients, as well as larger data.

4.1 DataTask: An Active, Composable I/O Unit

DataTask Specification: In order to unify various unique I/O interfaces to account for their varied structure and intentions, we introduce a novel abstraction, the DataTask. Unlike conventional approaches that treat I/O as passive data movement, DataTasks encapsulates I/O operations and associated transformations as active, composable units.

The specification for a DataTask is shown in Listing 1. DataTasks have fields for their ID, content, dependencies, type, property list, and context. ID is a unique identifier for the DataTask. Content is an object representing the data of the task, generally a string or other allocated data buffer in addition to operational information such as filenames, offsets, and size data. Dependencies are other DataTasks which this DataTask is known to depend on. This is used to establish sequencing in asynchronous modes when Read After

Write consistency is needed. The type is the operation associated with the DataTask. They include types for performing reads, writes, staging in and out of data, etc. The property list contains configurations particular to that DataTask, such as asynchronous mode, row-wise vs columnar ordering, and which interface to translate to. The context contains special arguments that get passed to a particular interface. For example, context can specify interface-specific compression mechanisms, chunking styles, or anything else that gets passed as an argument to the interface.

Listing 1: The DataTask Structure

```
typedef struct DataTask {
    int64_t task_id;
    content *C;
    DataTask *dependencies;
    task_type t_type; // e.g., read, write, super
    dt_properties *plist;
    context *ctx;
}
```

SuperTasks: One important operation type is called `super`, which can be used to create a SuperTask. A SuperTask is a specialized form of DataTask associated with metadata and used to represent higher-level structure of data, such as directories, HDF5 files and groups, etc. When creating a SuperTask, existing Dependencies will be included as members of the SuperTask, but new members can also be added with an `add_to_super` operation type. The dependencies of a SuperTask are tracked in a DataTask registry, and the updates occur there. SuperTasks allow the system to preserve the user-facing structure of the data, for example, a single HDF5 file can have multiple groups and datasets that need to be accessed as if they were a folder with multiple files.

Benefits of DataTasks: The DataTask abstraction provides several key advantages over traditional, monolithic I/O approaches. First, it enables fine-grained control over I/O operations. Each DataTask represents a distinct unit of work, allowing for precise scheduling and resource allocation. Second, it facilitates composability. Complex I/O workflows can be built by chaining together multiple DataTasks, with dependencies explicitly defined. This modularity simplifies development and promotes code reuse. Third, it enhances portability. By abstracting away the specifics of the underlying storage system, DataTasks can be executed on diverse platforms without modification. Fourth, it enables optimization. The system can analyze DataTask dependencies and characteristics to apply various optimizations, such as request coalescing, prefetching, and data placement strategies, transparently to the application. Fifth, it fosters asynchronicity. DataTasks can be executed concurrently, overlapping I/O with computation and maximizing resource utilization. This is crucial for hiding I/O latency and improving overall application performance. Sixth, it improves debuggability and explainability. Because a DataTask represents a logical unit of work with clear inputs, outputs, and dependencies, it becomes much easier to track the flow of data and identify the source of performance bottlenecks or errors.

4.2 Flexible I/O through DataTasks

DataTask Pipeline: The design of the I/O system is built around a structured pipeline that transforms traditional I/O operations into a flexible, schedulable, and optimized process using DataTasks. The pipeline is built around three main components: Task Composition, Task Scheduling, and Task Execution, each responsible for a distinct phase of the pipeline.

The pipeline begins with Task Composition, executed in the user space of the application. This component applies predefined logic to transform intercepted I/O requests into DataTasks. Tasks are composed with awareness of their broader context (e.g. dependencies, or destination). This awareness enables the implementation of composition policies such as aggregation, reordering, and replication that can be flexibly applied to DataTasks based on their use case.

Once DataTasks are generated, they are queued for scheduling. Since DataTasks carry knowledge of its transformations and dependencies, the Task Scheduling can combine this knowledge with factors such as data locality, or system load to have an novel understanding of both the supply and demand of the system. This knowledge can then be leveraged to determine optimal execution strategies that dynamically consider the best executor for the DataTask, ensuring efficient execution across distributed nodes.

Finally, the Task Execution phase, this phase is executed on an application-independent, distributed pool of executors that interface with the storage backends. With an understanding of the data characteristics, final interface, and transformation requirements, these executors can execute the DataTasks by: transforming the data to match the expected storage interface; applying performance optimizations such as batching and interface-aware compression; and executing the actual I/O operations.

Composition Policies: To accommodate the diverse behaviors and requirements of different domains, pipelines can be assigned composition policies. These policies define broad operational behaviors that dictate how data is processed and managed. They are inherently flexible to support a wide range of domain-specific needs.

For example, in Big Data environments, where resilience and availability are critical, DataTasks can employ a duplication policy. This policy creates multiple duplicates of a DataTask, scheduling them immediately across different destinations to ensure redundancy and fault tolerance. Similarly, in HPC environments, where workloads often rely on HDD-backed PFSs and must handle many small I/O operations efficiently, DataTasks can apply an aggregation policy. This policy ensures that DataTasks within a pipeline are combined and only delivered once a predefined time or size threshold is reached, improving performance. Beyond these, additional behaviors can be defined. For instance, prefetching can be implemented through a predictive policy, which leverages knowledge from previous DataTasks to generate speculative DataTasks. These DataTasks are dynamically assigned to I/O operations as they are intercepted, reducing latency. This policy is most commonly applied during Task Execution, when a file is opened and stages its data into an executor before any read has occurred; DataTasks can be generated to read that data from executors before any requests are truly received.

4.3 Optimizing DataTask management

Leveraging DataTask dependencies: DataTasks can mark and store data dependencies with other DataTasks. For example, a write-append DataTask can mark another as a dependency to ensure proper ordering, or a read DataTask can mark a write one as a dependency to ensure correctness. Beyond serving to implement high-level relationships between the data, intelligent use of this knowledge can help build an asynchronous, lazy scheduling of Data Tasks.

When DataTasks are created, they get stored temporarily at the client in a circular buffer awaiting scheduling. If, instead of enabling scheduling immediately, the system tracks the dependencies of the DataTask, we can maintain it in local memory until one of the dependent DataTasks gets submitted, or some maximum time or memory size is reached.

A dispatch algorithm (algorithm 1) is used to decide when and what DataTask needs to be scheduled. First, the algorithm selects the stalest DataTask (p) in the pool enabling it for scheduling. Once done, a second pass through the pool looks for any DataTask still in it that p is dependent on and also marks them for scheduling. The staleness of a DataTask refers to time since its last dependency was added to the pipeline, by waiting until the dependencies of a DataTasks have been established in the pipeline, the system can: resolve them together to improve performance, extend the time to execute composition policies, and minimize I/O movement.

Algorithm 1 Scheduling Dispatch Algorithm

```
1:  procedure DISPATCH(DataTask $dts$[])
2:      Let $P$ be the pool of tasks to be scheduled
3:      Let $T$ be the (empty) pool of tasks to be dispatched
4:      Append $dts$ to $T$
5:      while sizeof($P$) exceeds memory threshold do
6:          $p \leftarrow$ stalest task in $P$
7:          remove $p$ from $P$ and append $p$ to $T$
8:      for $p \in P$ do
9:          if stale($p$) then
10:             remove $p$ from $P$ and append $p$ to $T$
11:             continue
12:         for $t \in T$ do
13:             if $t$ depends on $p$ then
14:                 remove $p$ from $P$ and append $p$ to $T$
15:                 break
16:     for $t \in T$ do
17:         send $t$ to scheduler
```

Minimizing I/O movement through accelerated I/O resolution: As a result of this lazy scheduling, the system can attempt to accelerate the resolution of DataTasks by obtaining part or all of its I/O from the local ring buffer. Note that this allows serving I/O requests without hitting storage, so long as the data exists in the memory pool systems. The methodology for resolution is shown in algorithm 2.

To summarize, resolution of a DataTask from existing buffers involves querying the DataTask registry for DataTasks associated with the current file ordered by recency (line 2), calculating sections of the current DataTask satisfied by existing buffers (lines 4-9), and performing buffer copies from existing buffers into the result (lines 10-16). After this buffer resolution, it is possible that the DataTask will not be completely satisfied by existing buffers, in which case additional reads from storage may remain necessary. The *range_bound* variable from the resolution algorithm can be utilized to determine which indices within the file remain to be fetched and convert them to DataTasks.

DataTasks enable a new paradigm of I/O management, that can leverage enriched knowledge of the data to transparently, and efficiently manage diverse I/O while staying flexible enough to support

Algorithm 2 DataTask Buffer Resolution Algorithm

```
1:  procedure RESOLVE-DATATASK(DataTask $dt$, buffer $result$)
2:      Let $T$ be the pool of DataTasks associated with the file $dt.filename$ (queried from the task registry, by recency)
3:      Let $resolve\_dts \leftarrow \{\}$
4:      Let $range\_bound$ represent an array of the locations requested by $dt$
5:      for $t \in T$ do
6:          if $t$ satisfies part of $range\_bound$ which is not already satisfied then
7:              Store the offsets and sizes of $t$ that are required as $t\_offsets$ (calculated during satisfaction)
8:              Append ($t$, $t\_offsets$, and the starting offset of $t$ in $dt$) to $resolve\_dts$
9:              Mark the satisfied range of $range\_bound$
10:     for $(rt, t\_offsets, dt\_offset) \in resolve\_dts$ do
11:         Read $rt.data$ into $rt\_buffer$
12:         for $(t\_offset, t\_size, dt\_offset) \in t\_offsets$ do
13:             if $previous\_offset$ is not initialized then
14:                 Let $dt\_offset \leftarrow dt\_offset + t\_offset - previous\_offset$
15:             Let $previous\_offset \leftarrow t\_offset + t\_size$
16:             Copy $t\_size$ elements from $rt\_buffer$ at $t\_offset$ to $result$ starting at $dt\_offset$
```

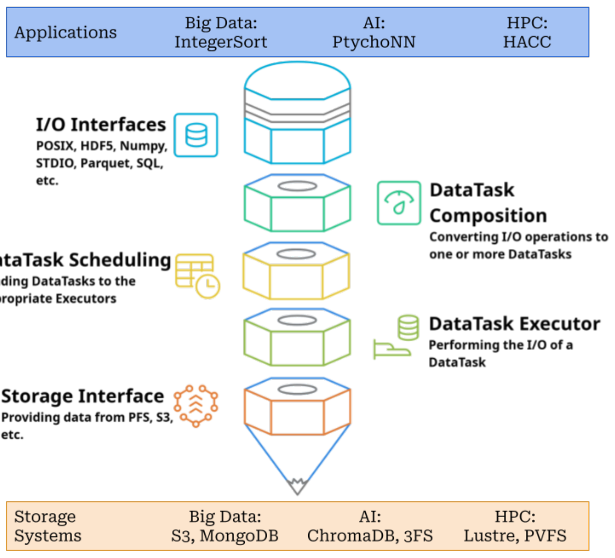


Figure 1: DTIO software stack: offers a unified storage interface, support for domain-specific interfaces, the DataTask abstraction, an I/O optimization layer, and the concept of vertical slices representing DTIO's middleware role across application domains.

complex, converged scientific workflows. We believe that DataTask-based systems could serve as a viable alternative to traditional monolithic I/O systems.

5 The DTIO System

DTIO is a lightweight DataTask-based I/O middleware. Its architecture can be viewed as a layered software stack, shown in figure 1. DTIO sits between applications and underlying storage systems, which it accesses via I/O and storage interfaces. This stack encompasses several key components, each contributing to DTIO's ability

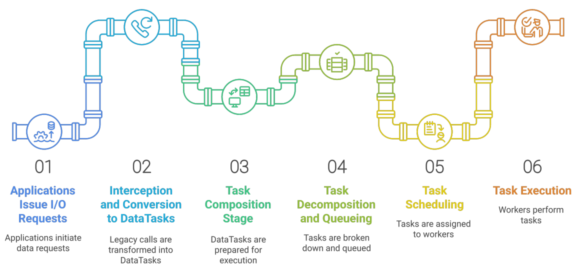

01
Applications Issue I/O Requests
Applications initiate data requests
02
Interception and Conversion to DataTasks
Legacy calls are transformed into DataTasks
03
Task Composition Stage
DataTasks are prepared for execution
04
Task Decomposition and Queueing
Tasks are broken down and queued
05
Task Scheduling
Tasks are assigned to workers
06
Task Execution
Workers perform tasks

Figure 2: Data Flow within the DTIO Architecture: six stages of I/O processing within DTIO: (1) Applications initiate I/O requests; (2) Legacy I/O calls are intercepted and converted into DataTasks; (3) DataTasks are prepared for execution during the Task Composition stage; (4) Tasks are decomposed and queued; (5) Tasks are scheduled and assigned to executors; and (6) Executors perform the I/O operations.

to unify and optimize I/O. At the highest level, a unified storage interface provides a consistent access point for diverse storage types. Below this, support for domain-specific I/O interfaces caters to the specialized needs of different application areas (e.g., HPC, Big Data, ML). DTIO intercepts a wide variety of interfaces such as POSIX, STDIO, HDF5, and Numpy. When interacting with storage, DTIO can also utilize these interfaces to access different storage and data layouts. The core of DTIO's functionality lies in its handling of DataTasks, which represent I/O operations as *schedulable* units.

5.1 Data Flow in DTIO

Figure 2 showcases the six key stages of data flow within the DTIO system, which transforms traditional I/O requests to DataTasks, optimizes them, and delivers the data to the storage system. Applications begin by issuing standard I/O requests (Stage 1). These requests are transparently intercepted and converted into DataTasks (Stage 2), decoupling the application from the underlying I/O implementation. The Task Composition stage (Stage 3) creates the DataTasks, and applies data policies. Subsequently, DataTasks are queued for execution (Stage 4). The Task Scheduling stage (Stage 5) assigns DataTasks to available worker processes, considering factors such as the DataTask characteristics, Quality of Service constraints, and data locality. Finally, executors execute the DataTasks, adapting the data to the required storage system, and performing I/O operations (Stage 6). The three key components of DTIO and their interactions can also be seen in this figure, as DataTask Composition, DataTask Scheduling, and DataTask Execution.

5.1.1 DTIO API. The DTIO native API can be seen in table 2. It is composed of a DataTask Creation API that enables creating, editing, and destroying DataTasks, and a DataTask Introspection API that allows users to obtain information about the DataTasks in relation to the system and other DataTasks.

DataTask Creation API: Task construction is performed by calling createtask() passing a buffer alongside optional dependency, property list, and context arguments. The scheduletasks() call pushes DataTasks to the scheduler. It gives IDs for the DataTasks that are being scheduled, and their results can be viewed with a synchronous call to wait(). The managed buffers used by the lazy scheduling algorithms also have functions in this API to request and free them, as well as to read from and write to them.

Function	Input	Output
createtask	type, content, deps, plist, ctx	DataTask
scheduletasks	DataTask[]	ID[]
wait	ID	result
requestbuffer	size	buffer
freebuffer	buffer	N/A
readbuffer	offset, count	buffer
writebuffer	offset, count, data	buffer
taskposition	DataTask	position
taskperformance	DataTask	time_to_run
contentsize	DataTask	size_t
contenttype	DataTask	type

Table 2: DTIO's API

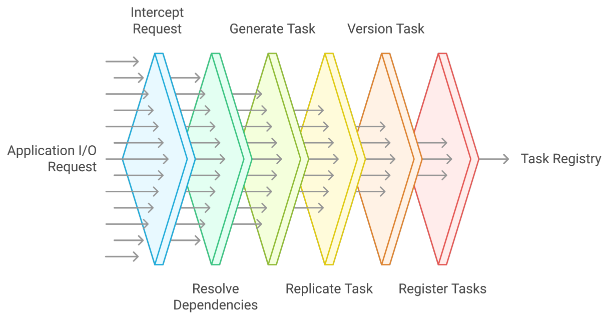


Figure 3: DTIO DataTask Composer: application I/O requests are processed in parallel, undergoing stages of interception, dependency resolution, DataTask generation, replication (optional), versioning, and registration in the Task Registry. Key technologies include DataTasks, Thallium/HCL communication libraries, and zero-copy networking (RDMA/RDSA).

DataTask Introspection API: The intent of this API is to enable analysis of DataTasks as the system runs for various purposes. For example, taskposition() allows the user to get the current position of a DataTask, to determine if it has been scheduled or not. The taskperformance() call gives the time that it took to run the task, and can only be utilized once a task has completed. The contentsize() and contenttype() calls give information about the size of content being stored in the DataTask and the type of the content (e.g., int, char, string).

5.1.2 DataTask Composition Component. The DataTask Composer translates the user I/O into DataTasks transparently to the user application. The DataTask Composer executes in the application userspace and has been designed to be as lightweight as possible by minimizing the amount of metadata management performed to the minimum necessary to ensure correctness to applications.

The DataTask composer consists of several steps, as shown in figure 3. First, is acquisition, where DTIO gains ownership of the application data. Acquisition can be achieved through *LD_PRELOAD*

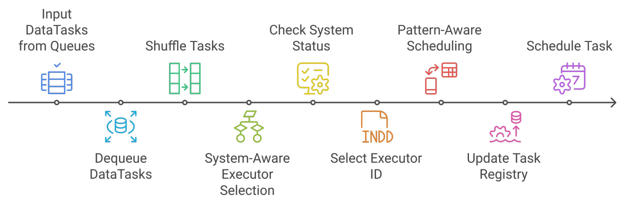


Figure 4: The DTIO Scheduler uses: (1) System-Aware Executor Selection, leveraging real-time telemetry (load, data locality) to choose the optimal executor; and (2) Pattern-Aware Scheduling, consulting the task registry for dependency management and contextual information. These paths converge to drive the 'Schedule Task' decision, ensuring both performance and correctness. The task registry maintains a persistent record of schedule actions.

interception or through direct use of the DTIO API. When intercepting pre-existing applications, and in order to maintain transparency, DTIO allows the use of external configuration files to define intents, sets global defaults for properties, and define dependencies. Once an I/O request is acquired, DTIO extracts any metadata possible, and resolves the dependencies on other operations. Afterward, the I/O request is translated to the DataTask format. Translation is separated into generation and operators. Generation executes DTIO logic to map the data interface to DataTasks. Before DataTasks get scheduled, DTIO applies optional operators to transform and annotate the data, such as replication, versioning, and predictive generation. These operators can modify existing DataTasks as well as create new ones. Finally, the translated DataTasks are registered in the DataTask registry, a distributed hashmap that allows other clients and DTIO itself access to information about non-local DataTasks.

DTIO contains translation logic, which consists of small code snippets that define the conversion from an I/O interface to DataTask and vice versa. For example, DTIO can translate HDF5 dataset writes or reads to a write or read DataTask and HDF5 files and groups to SuperTasks.

5.1.3 DataTask Scheduler Component. The DataTask scheduler is in charge of intelligently mapping DataTasks to the pool of executors that will execute them. This scheduling is performed based on requirements of the data defined in the DataTask, and the current status and characteristics of the executors.

DTIO achieves DataTask scheduling with several steps, as shown in figure 4. First, tasks are dequeued from the scheduling queue. The DataTasks are then ordered based on dependencies, so that dependent DataTasks will be scheduled together and executed in-order. After a DataTask has been scheduled, the DataTask registry gets updated with that DataTask's status before sending it to the selected executor. DataTasks can be specified as either synchronous or asynchronous, with asynchronous DataTasks returning immediately. Asynchronous mode is the default when using the DataTask API, while synchronous is the default on interception.

To inform the mapping decision, the scheduler tracks and makes use of the system status to inform its placements. Its goal is to balance two factors: First, it aims to balance the compute load across the executor pool, since DataTasks contain information about their time complexity, and the DataTask registry is informed when execution of a DataTask is completed; the scheduler can maintain understanding of the current load of all the executors in the pool, steering new DataTasks away from executors that are busy. Second, it aims to maximize data locality. A DataTask executor may contain in-memory data required by the DataTask, either if it has just executed a dependent DataTask, or if it has staged-in some data (another DTIO policy). In these situations, DTIO allows user to define a "maximum load threshold" of a executor, under which the scheduler will bypass all load balancing decisions.

5.1.4 DataTask Execution Component. The DataTask executors are responsible for translating DataTasks into a desired or required interface for the storage endpoint. Further, by understanding the data origin and mapping DataTasks to an output interface, executors can determine any incompatibility between for data representations. As such, DTIO can offer users the ability to automatically have their I/O converted to a more efficient interface for the target storage systems without requiring extensive I/O knowledge or how the mapping is performed.

First, they receive DataTasks from the scheduler via a queue. When available, an executor will pop the next DataTask or batch of DataTasks (DTIO allows a configurable batch size for asynchronous operations) from the queue. Execution begins by ensuring the availability of the destination, this is especially important when requiring access to remote or cloud-based resources. To this end, the DataTask Executor contains a separate process, the Endpoint Manager, which uses a heartbeat mechanism to periodically check available storage either through dedicated calls or performing a minimal (i.e., 1 byte) read. Finally, the DataTask Executor executes the DataTask, running all operations needed, and executing the I/O using the required client for its specified interface. Once done, the DataTask Executor will use the task registry, to find the process that triggered the DataTask, send a result to it, and update the completion status of the DataTask on the task registry.

The executors are also responsible for establishing a configuration for the storage interfaces. While some I/O interfaces are simple and offer limited configurability, it is common for the configuration of the I/O operations to have significant effects on their executions. As such, it is imperative that the Task Executor is capable of configuring the translated interface. Out of the box, for most interfaces, DTIO provides the users with 3 pre-defined profile configurations: space-saving, performance, and balanced. Users can select the profile if using the DTIO API or set one globally on the configuration (performance by default). Profiles are defined on external files, and are expandable and user-editable. As an example, in HDF5, one available option is compression. A performance profile will disable compression in favor of efficient I/O, while a space-saving profile will configure HDF5 to use gzip with significant compression. Balanced will use the lzf option, which provides a good balance between space and performance.

6 Implementation details

6.1 Using DTIO

Figure 5 demonstrates a simple file write and read operation using the DTIO API. During initialization, a filename and a write buffer

```
// --- Initialization (typically done once) ---
filename = "test_file.txt";
write_buf = "Testing R/W with DTIO. This is msg body.";
size = length(write_buf);  // Or a fixed size like 50
// Create a write task.  (0 = WRITE, 1 = READ)
write_task = createtask(type=WRITE, content={filename, offset=0,
size, data=write_buf}, priority=0, dependencies=0, flags=0);
write_task_id = scheduletasks([write_task]); // Schedule the task
wait(write_task_id); // Wait for the write task to complete
// --- Read Operation ---
read_buf = requestbuffer(size) // Allocate a buffer for reading
read_task = createtask(type=READ, content={filename, offset=0,
size, data=read_buf}, priority=0, dependencies=0, flags=0);
read_task_id = scheduletasks([read_task]); // Schedule the read task
wait(read_task_id);  // Wait for the read task to complete
result = readbuffer(read_buf, offset=0, size); // Retrieve the data
```

Figure 5: By representing I/O requests as schedulable DataTasks, DTIO enables fine-grained control over data movement, facilitating optimizations such as deferred I/O, request coalescing, and computational storage.

containing the data to be written are defined. A `createtask()` call is then used to create a DataTask representing the write operation. Crucially, the type parameter is set to *WRITE* (a defined constant for clarity), and the content parameter encapsulates the filename, offset, size, and a pointer to the data buffer. This write task is then scheduled using `scheduletasks()`, and the wait function is used to block until the write operation completes. For the read operation, a buffer is allocated using `requestbuffer()`. A read DataTask is created similarly to the write task, but with the type set to *READ* and the buffer field of the content parameter pointing to the allocated read buffer. This read task is scheduled and waited upon. Finally, `readbuffer()` retrieves the data from the buffer.

6.2 Considerations

Alleviating I/O interference: The DataTask composer can be configured to limit the total size and time that DataTasks can be kept unscheduled on the local ring buffer. If a DataTask has not been marked to schedule within the time limit or if the ring buffer size exceeds the size limit, DataTasks in the pool will be scheduled. The pool size limitation can be set to one in order to force immediate scheduling, but this prevents a number of DTIO's optimizations. Setting small and varied limits on each client can help alleviate I/O interference issues, but is not expected to give a performance benefit on its own.

Asynchronous Intercepted Operations: DTIO supports asynchronous execution of I/O operations, with asynchronous mode as the default in the DataTask API and available for intercepted I/O. Synchronous intercepted operations mapped asynchronously can be volatile, as DTIO returns the requested write size immediately while scheduling the actual operation. This can lead to discrepancies if errors or partial writes occur, as the program receives a return value before completion. The actual result can be inspected with the DTIO API, but this requires small program changes, To support this hybrid approach, DTIO has a small, independent library with an API to acquire the DataTask ID and request its status. In this case, users have to be careful with reads that may try to access asynchronous writes before completion. To solve this, DTIO offers mechanisms to enforce dependencies through a configuration file, ensuring reads wait for relevant writes.

Asynchronous reads: Asynchronous read operations are also possible, but DTIO does not implement them for purely intercepted I/O because they require DTIO to own the read buffer to ensure consistency. In DTIO's API, where there is more control, the user can, in fact, request a DTIO-managed buffer and then read into it, but all access to that buffer must occur through the DTIO API to ensure that the data access waits for unresolved DataTasks.

Sessions: Scheduling of DataTasks need not occur immediately, and can be naturally delayed by the queue. This is the basis for DTIOs lazy scheduling and accelerated I/O resolution, but can also be used to implement session logic. A session can be created with an ID, any process can join this session and generate DataTasks. These DataTasks are logically aggregated and are only sent together to the scheduler once every process has exited the session, This allows finer delineation of I/O and compute phases in step-based simulations, and helps DTIO resolve I/O dependencies in advance.

I/O mapping in Task Executors: Task Executors have a powerful tool to accelerate performance: I/O mapping. However, performance depends heavily on the semantics of the interfaces being mapped. For example, CSV or json files are slower and take more space compared to efficient scientific formats such as Parquet or Feather [6, 32]. This is largely due to the difference between character and binary data reads, and added optimizations such as compression.

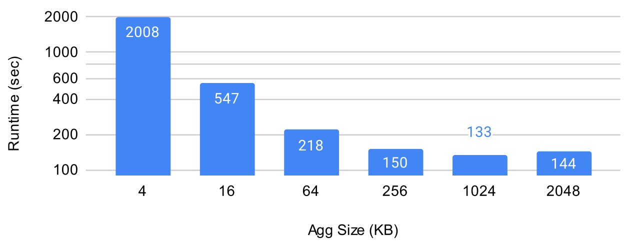
Metadata Management: Shared metadata management in DTIO (e.g. DataTask register) is kept lightweight via HCL, a state-of-the-art lock-free distributed data structure library. HCL leverages RPC calls under-the-hood to perform operations on its data structures, but allows the use of RDMA or RDSA for zero-copy networking where available and avoids the overhead of RPCs entirely when data structures are local to the current node.

DataTask size limitation: DTIO has a DataTask size limitation of roughly 64 MiB introduced by its communication libraries Thallium and HCL. This requires breaking up DataTasks in the rare event of an operation too large to package; split DataTasks can then be scheduled and executed together.

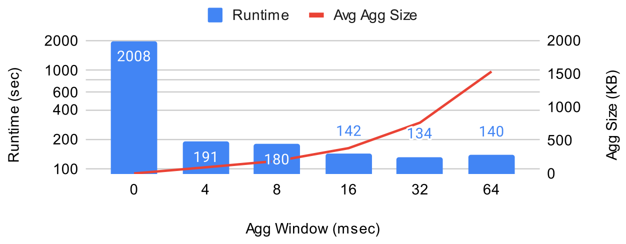
7 Evaluations

Implementation: DTIO is an open-source C++ project under the GPL v3 or later license. The source code is available at https://github.com/grc-iit/DTIO. DTIO depends on HCL [10] to manage many of its data structures. HCL runs on top of Thallium [22] from the Mochi framework [27]. The network stack from Mochi includes Thallium, Mercury [29], Margo [21], Argobots [28], and libfabric [15], as well as leveraging Cereal [23] for serialization. These can all be configured to allow different fabric setups, such as sockets, TCP, or infiniband. DTIO configuration files are in a simple YAML format [2]. DTIO also has dependencies for its intercepted and client libraries, such as HDF5 and io_uring.

Methodology: We evaluated each component of DTIO and their respective optimizations and features. Due to our target application, these evaluations focus on showcasing formats such as HDF5 and NPZ, as well as the general POSIX interface (via IOR) due to its broad applicability. Section 7.1 demonstrates the performance of aggregation. Section 7.3 demonstrates the performance of the data staging optimization that prefetches data to executors in advance of data being requested. Section 7.2 demonstrates the performance of the



(a) Fixed-Size Aggregation in DTIO. Runtime decreases as aggregation size increases, with optimal performance near 1MB, aligning with the PFS stripe size.



(b) Fixed-Time Aggregation in DTIO. Runtime decreases sharply as the aggregation window increases. The red line indicates the increasing average aggregation size (KB) with larger windows.

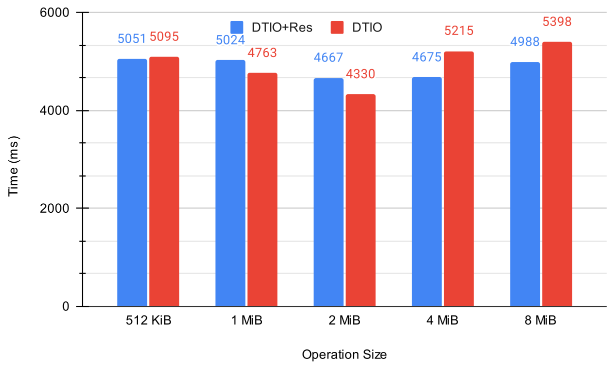
Figure 6: Impact of Aggregation Strategies on DTIO Write Performance. (a) Fixed-size aggregation shows optimal performance near a 1MB aggregation size, corresponding to the PFS stripe size. (b) Fixed-time aggregation demonstrates optimal performance with a ≈32ms window. Both strategies showcase DTIO's ability to achieve up to a 5x performance improvement by converting small 4KB writes into larger, more efficient I/O operations.

caching optimization that resolves I/O from DataTasks. Section 7.4 showcase DTIO's overhead for different small and large I/O sizes. Finally, section 7.5 demonstrates the performance of DTIO in an end-to-end setting on the PtychoNN workload.

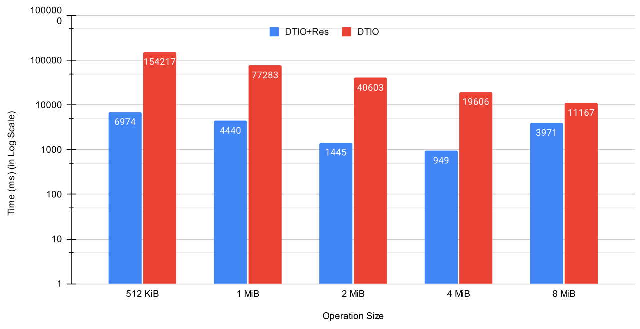
Testing Environment: IOR tests were performed on the multi-tiered Ares research cluster at the Gnosis Research Center, hosted by the Illinois Institute of Technology [5]. This cluster has 32 compute nodes. Each compute node has a dual Intel® Xeon Scalable Silver 4114 processor and 96 GiB of RAM, along with NVMe PCIe ×8 drive and a SATA SSD. PtychoNN tests were performed on the Polaris supercomputer at Argonne Leadership Computing Facility [11]. Polaris has 560 nodes. Each node has 4 NVIDIA A100 GPUs, 512 GiB of DDR4 RAM, and 2x 3.2 TB local SSD drives.

7.1 Aggregation Policies Evaluation

In this case, we demonstrate the impact of different aggregation strategies on I/O performance. To do this, we run a small write-only I/O workload using fio. In this case, fio writes in units of 4KB from 24 threads and total 8GiB in size. For the fixed-size aggregator, we vary the amount to aggregate before submitting an I/O request. For the fixed-time aggregator, we vary the window of time to merge I/O requests. DTIO is configured with 4 executors for performing I/O. In



(a) IOR Write Performance. Comparison of DTIO with and without accelerated resolution ("DTIO+Res" vs. "DTIO") across various operation sizes. Minimal performance difference is observed, as expected, since accelerated resolution is a read-focused optimization. The maximum overhead of accelerated resolution is 12%.



(b) IOR Read Performance (Log Scale). DTIO with accelerated resolution ("DTIO+Res") significantly outperforms standard DTIO across all operation sizes. Performance improvements range from 64.4% (8 MiB) to 95.5% (512 KiB), with an average reduction in read time of 89.2%.

Figure 7: Impact of Accelerated Resolution on Write and Read Performance. (a) Write performance shows minimal impact from accelerated resolution. (b) Read performance (log scale) demonstrates substantial gains with accelerated resolution, achieving up to a 95.5% reduction in read time.

figure 6, it can be seen that aggregation has a substantial impact on performance. In both cases, about 5x improvement is gained in the best case. For the fixed-size aggregator, performance peaks around 1MB. For the fixed-time aggregator, performance peaks with a window of size 32ms, which is when the average request size of a window is 768KB – almost 1MB. This is because the stripe size of the PFS is 1MB, so the bandwidth becomes saturated. Overall, by allowing I/O tasks to be aggregated, DTIO can convert unfavorable access patterns into ideal ones and accomplish significantly higher I/O bandwidth.

7.2 Accelerated I/O Resolution Evaluation

As discussed in Section 4.3, DTIO implements an accelerated I/O resolution optimization, where DataTasks are temporarily stored in a circular buffer. This allows subsequent DataTasks to retrieve data

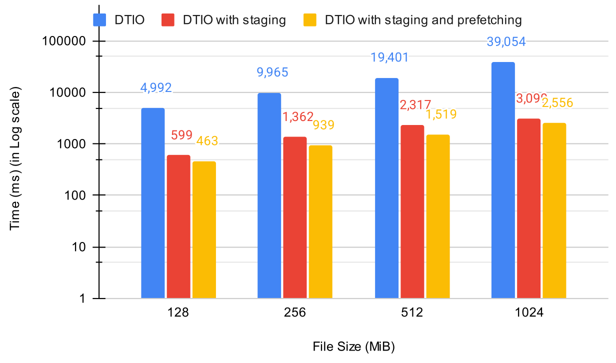


Figure 8: Read Performance with Staging and Prefetching (Log Scale). Comparison of standard DTIO, DTIO with staging, and DTIO with both staging and prefetching. Staging provides a significant performance improvement (average 88.6%), with prefetching adding a further 3.1% gain, resulting in a total improvement of 91.7% over standard DTIO.

directly from memory instead of disk. To demonstrate the benefits of this optimization, IOR is configured to emulate a producer-consumer workflow that writes and then reads a file of size 1 GiB for various operation sizes. We compare DTIO with and without accelerated resolution, which is configured to use 10% of DRAM for storing unresolved DataTasks.

The write performance is shown in subfigure 7a. Since this is a read optimization, little performance difference was observed as expected. Accelerated resolution (res) does not have a significant overhead in writes -≈12% in the worst case. The read performance is shown in subfigure 7b. DTIO accelerated I/O resolution achieves a reduction in read time by 95.5% at 512 KiB with accelerated resolution and 64.4% reduction at 8 MiB, for an average reduction of 89.2%. This is because without optimization, data is transferred using PFS over disk storage rather than main memory, which the accelerated I/O resolution enables. By enabling DataTasks to be resolved directly from other DataTasks stored in local memory, significant performance improvements can be gained in producer-consumer workloads by replacing inefficient disk-based transfers with high-performance memory transfers.

7.3 Data Staging Evaluation

DTIO implements a data policy that allows DataTask Executors to stage data before read operations arrive for it. When this optimization is active, the Scheduler prioritizes data locality by sending DataTasks to the executor that has their staged file. In addition to this policy, DTIO can also attempt to send the staged data to the circular buffers of the clients asynchronously so that it can be accessed through the accelerated I/O resolution mechanism. To demonstrate the benefits of these optimizations, we show the performance of IOR with DTIO-added data staging optimizations in figure 8. For this test, IOR is configured to perform 1 MiB reads from files of varying total sizes. The performance is tested with DTIO added, one version without staging, one version that stages the file into the executors and another version that preloads data into the circular buffers using the accelerated I/O resolution mechanism to pull DataTasks that get automatically populated from staged data.

The performance of the staging optimization shows a significant improvement over DTIO without staging, with an average improvement of 88.6%. Adding the prefetching optimization to staging improves performance by a further 3.1%, bringing the performance improvement over standard DTIO to 91.7%. By predictively and asynchronously loading data before it is accessed, significant performance improvements can be gained by avoiding synchronous I/O stalls. However, the effectiveness of this approach is largely dictated by memory capacity and the amount of data that needs to be staged.

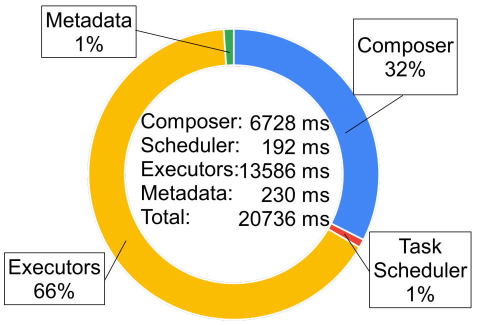
7.4 Overhead Analysis

For this evaluation, we wanted to explore where DTIO spends most of its time in order to understand potential bottlenecks and overheads in ordinary scenarios. The overhead is not significant, but it is worth investigating to understand the life cycle of an average operation in the DTIO pipeline. Figure 9 shows the overheads of DTIO under read and write settings. Timers are placed within DTIO to demonstrate how much time DTIO spends in composition, scheduling, and execution phases. Most metadata updates are performed during execution, but the timings for these are separated out for better understanding. We performed two sets of tests, one of 8 MiB operations as can be seen in Figure 9a and the other of 256 KiB operations in Figure 9b with a fixed file size of 1 GiB.

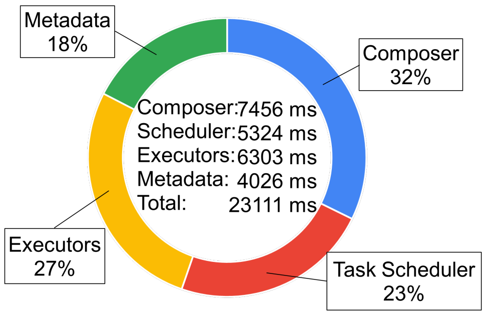
Figure 9a shows that in DTIO, when the I/O operations are larger and fewer, the Metadata operations and Task Scheduler overheads shrink by ≈94% and ≈96% respectively when compared to doing smaller I/O and a larger number of I/O operations which can be seen in Figure 9b. In the best case, however, the overheads of Metadata and Task Scheduling are negligible, at roughly 1% of the total time each. In both cases, DTIO spends most of its time composing and executing I/O operations. For composing, a copy of the data is created because DTIO runs in a separate address space from the application. For execution, this is where the actual I/O occurs. It is notable that if metadata overhead were to become excessive, DTIO provides batching functionality to allow executors to wait and perform multiple I/O operations and their corresponding metadata updates simultaneously. This slows down throughput, but it can significantly reduce metadata overhead.

7.5 End-to-End Evaluations

For the end-to-end evaluations, PtychoNN is run with and without DTIO interception. The PtychoNN workflow consists of a producer which reads in HDF5 data and converts it to Numpy and a consumer which reads the Numpy data as tensors. The producer and consumer are run in sequence for the ordinary test, but with DTIO acting as an intermediary they can be run at the same time (as DTIO will allow access to partial results). The total file sizes employed in this test are multiples of 3.2 GiB, and individual I/O operations are kept to roughly 3 MiB so that DTIO's system can be stressed with a larger number of operations. DTIO utilizes all optimizations to improve performance, including staging, asynchronicity, and accelerated I/O resolution.



(a) DTIO Overhead Breakdown (8 MiB Operations). Distribution of time spent within DTIO components for a 1 GiB workload using 8 MiB operations. Executors (66%) and Composer (32%) dominate, with minimal overhead from Task Scheduler (1%) and Metadata operations (1%).



(b) DTIO Overhead Breakdown (256 KiB Operations). Time distribution for a 1 GiB workload using smaller 256 KiB operations. Compared to (a), Task Scheduler (23%) and Metadata (18%) overheads increase significantly, while Executors (27%) and Composer (32%) represent a smaller proportion of the total time.

Figure 9: DTIO Overhead Analysis. Breakdown of time spent in DTIO components (Composer, Task Scheduler, Executors, Metadata) for a 1 GiB workload. (a) With large 8 MiB operations, Executor and Composer time dominates. (b) With smaller 256 KiB operations, Task Scheduler and Metadata overheads increase substantially, highlighting the impact of operation size on DTIO's internal overhead distribution.

The bandwidth of each part of the I/O pattern of the PtychoNN workflow with and without DTIO is shown in figure 10. The consumer read in DTIO is very efficient compared to PtychoNN without DTIO, showing an average bandwidth increase of 2516 MiB/s. However, the producer read and write phases shows better bandwidth in PtychoNN without DTIO. PtychoNN outperforms DTIO in the producer read phase by an average of 165.5 MiB/s. This is expected, as the producer reads are transferring data from the filesystem over to DTIO, while the consumer reads are able to leverage DTIO's accelerated I/O resolution. The producer writes also outperform DTIO in PtychoNN by an average of 221 MiB/s, but only if we force DTIO to trigger these writes. Since DTIO can use DataTasks to resolve future reads, storing the intermediate result is not strictly necessary. DTIO can save storage space by using its own buffers exclusively,

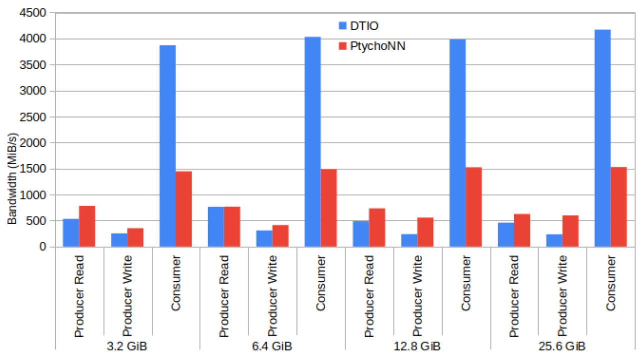


Figure 10: PtychoNN End-to-End Bandwidth Comparison: DTIO vs. Standard PtychoNN across various file sizes. Bandwidth (MB/s) for Producer Read (HDF5 to Numpy), Producer Write, and Consumer Read (Numpy to tensors) phases. DTIO significantly improves Consumer Read bandwidth (average increase of 2516 MB/s) due to accelerated I/O resolution. Furthermore, DTIO can eliminate the need for Producer Writes altogether, saving storage space and potentially improving overall workflow efficiency by using in-memory DataTasks.

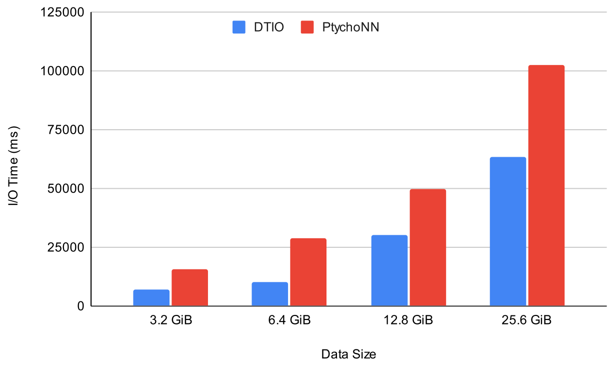


Figure 11: PtychoNN End-to-End I/O Time: Impact of Eliminating Producer Writes with DTIO. Total I/O time (ms) for the PtychoNN workflow, comparing standard PtychoNN with DTIO. By avoiding unnecessary producer writes, leveraging in-memory data transfers DTIO achieves performance improvements ranging from 38% (for 25.6 GiB data) to 65% (for 6.4 GiB data), with an average improvement of 49.6%, or a 21.6-second reduction in I/O time.

and if it does, then the producer writes do not need to be performed. Figure 11 shows the I/O time to run PtychoNN with and without DTIO, where DTIO is optimizing performance by not performing the producer writes. In this case, DTIO shows a performance improvement over PtychoNN that varies from 38% for large I/O (25.6 GiB) to 65% for smaller workloads (6.4 GiB). The average I/O performance improvement is 49.6%, or 21.6 seconds of PtychoNN's I/O time.

8 Conclusions

In conclusion, this paper has presented DTIO, a scalable I/O runtime which unifies the disparate data stacks for modern HPC, Big Data, and ML workflows. DTIO utilizes a unique DataTask structure to express the movement of data, its ordering, and its dependencies on other data as a task. DTIO achieves a unification of converged HPC, Big Data, and ML workflows by utilizing intelligent mapping of interfaces, and automatically determines the best method to relate their unique semantics. It achieves this with minimal overhead of 9% compared to offline translation methods. In addition, DTIO's accelerated I/O resolution can provide a performance benefit of 89.2% for reads, and data staging can provide an average performance improvement of 88.6% without prefetching and 91.7% with prefetching. Finally, in an end-to-end evaluation against the PtychoNN data processing workflow, DTIO achieved an average I/O performance improvement of 49.6%.

For future work, we plan to expand DTIO across more interfaces such as netCDF and test it on more hybrid workflows. Further, we plan to leverage the DataTask abstraction to explore new scheduling and task execution techniques that take a user-specified quality of service into account, considering certain needs such as fault-tolerance and improved locality.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF), Division of Computer and Network Systems (CISE/CNS), under Grant CCRI-CISE 2346504, Grant CSSI-2411386, and the Department of Energy (DOE), Office of Advance Scientific Computing Research (ASCR), under Grant DE-AC02-06CH11357.

References

[1] Gustaf Ahdritz, Nazim Bouatta, Christina Floristean, Sachin Kadyan, Qinghui Xia, William Gerecke, Timothy J O'Donnell, Daniel Berenberg, Ian Fisk, Niccolò Zanichelli, et al. 2024. OpenFold: Retraining AlphaFold2 yields new insights into its learning mechanisms and capacity for generalization. *Nature Methods* (2024), 1–11.

[2] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2009. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008* 5, 11 (2009).

[3] Suren Byna, M Scot Breitenfeld, Bin Dong, Quincey Koziol, Elena Pourmal, Dana Robinson, Jerome Soumagne, Houjun Tang, Venkatram Vishwanath, and Richard Warren. 2020. ExaHDF5: Delivering efficient parallel I/O on exascale computing systems. *Journal of Computer Science and Technology* 35 (2020), 145–160.

[4] Hugo Caro, Sulyvan Dollin, Anne Biton, Bryan Brancotte, Dimitri Desvillechabrol, Yoann Dufresne, Blaise Li, Etienne Kornobis, Frédéric Lemoine, Nicolas Maillet, et al. 2023. BioConvert: a comprehensive format converter for life sciences. *NAR Genomics and Bioinformatics* 5, 3 (2023), lqad074.

[5] Gnosis Research Center. 2024. Hardware Overview | Gnosis Research Center. https://grc.iit.edu/resources/hardware-overview

[6] Avi Chawla. 2025. I/O Optimization in Data Projects - by Avi Chawla. https://blog.dailydoseofds.com/p/io-optimization-in-data-projects

[7] Mathew J Cherukara, Tao Zhou, Youssef Nashed, Pablo Enfedaque, Alex Hexemer, Ross J Harder, and Martin V Holt. 2020. AI-enabled high-resolution scanning coherent diffraction imaging. *Applied Physics Letters* 117, 4 (2020).

[8] Tran Khanh Dang, Ta Manh Huy, and Nguyen Le Hoang. 2021. Intermediate data format for the elastic data conversion framework. In *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*. IEEE, 1–5.

[9] Hariharan Devarajan, Gerd Heber, and Kathryn Mohror. 2024. H5Intent: Autotuning HDF5 with User Intent. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[10] Hariharan Devarajan, Anthony Kougkas, Keith Bateman, and Xian-He Sun. 2020. HCL: Distributing Parallel Data Structures in Extreme Scales. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE.

[11] Argonne Leadership Computing Facility. 2025. Machine Overview - ALCF User Guides. https://docs.alcf.anl.gov/polaris/machine-overview/

[12] Kun Feng, Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun. 2019. NIOBE: An intelligent i/o bridging engine for complex and distributed workflows. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 493–502.

[13] Kun Feng, Xian-He Sun, Xi Yang, and Shujia Zhou. 2018. SciDP: Support HPC and big data applications via integrated scientific data processing. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 114–123.

[14] NITRD Group et al. 2019. The Convergence of High Performance Computing, Big Data, and Machine Learning.

[15] OpenFabrics Interfaces Working Group. 2024. ofiwg/libfabric: Open Fabric Interfaces. https://github.com/ofiwg/libfabric

[16] The HDF Group. 2024. HDFGroup/hdf5: Official HDF5(R) Library Repository. https://github.com/HDFGroup/hdf5

[17] Helgi I Ingólfsson. 2023. MuMMI, a machine learning-driven modeling infrastructure for coupling different simulation scales; showcased for RAS-RAF activation. *Biophysical Journal* 122, 3 (2023), 463a.

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

[19] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Iris: I/o redirection via integrated storage. In *Proceedings of the 2018 International Conference on Supercomputing*. 33–42.

[20] Albert Krewinkel and Robert Winkler. 2017. Formatting Open Science: agilely creating multiple document formats for academic manuscripts with Pandoc Scholar. *PeerJ computer science* 3 (2017), e112.

[21] Argonne National Laboratory. 2024. mochi-hpc/mochi-margo: Argobots bindings for the Mercury RPC library. https://github.com/mochi-hpc/mochi-margo

[22] Argonne National Laboratory. 2024. mochi-hpc/mochi-thallium: Thallium is a C++14 library wrapping Margo, Mercury, and Argobots and providing an object-oriented way to use these libraries. https://github.com/mochi-hpc/mochi-thallium

[23] Argonne National Laboratory. 2024. USCiLab/cereal: A C++11 library for serialization. https://github.com/USCiLab/cereal

[24] Josh Moore, Daniela Basurto-Lozada, Sébastien Besson, John Bogovic, Jordão Bragantini, Eva M Brown, Jean-Marie Burel, Xavier Casas Moreno, Gustavo de Medeiros, Erin E Diel, et al. 2023. OME-Zarr: a cloud-optimized bioimaging file format with international community support. *Histochemistry and Cell Biology* 160, 3 (2023), 223–251.

[25] Bogdan Nicolae. 2022. Scalable Multi-Versioning Ordered Key-Value Stores with Persistent Memory Support. In *IPDPS 2022: The 36th IEEE International Parallel and Distributed Processing Symposium*. Lyon, France, 93–103. doi:10.1109/IPDPS53621.2022.00018

[26] Mahantesh K Pattanshetti. 2020. Design of High-Performance Computing System for Big Data Analytics. *JOURNAL OF ALGEBRAIC STATISTICS* 11, 1 (2020), 115–121.

[27] Robert B Ross, George Amvrosiadis, Philip Carns, Charles D Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K Gutierrez, Robert Latham, et al. 2020. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology* 35 (2020), 121–144.

[28] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, et al. 2017. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 512–526.

[29] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. 2013. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–8.

[30] Markus Stabrin, Fabian Schoenfeld, Thorsten Wagner, Sabrina Pospich, Christos Gatsogiannis, and Stefan Raunser. 2020. TranSPHIRE: automated and feedback-optimized on-the-fly processing for cryo-EM. *Nature communications* 11, 1 (2020), 5716.

[31] NumPy team. 2024. NumPy -. https://numpy.org/

[32] How to Optimize Data I/O by Choosing the Right File Format with Pandas. 2025. Shittu Olumide. https://www.statology.org/how-to-optimize-data-io-by-choosing-the-right-file-format-with-pandas/

[33] Maxim Zvyagin, Alexander Brace, Kyle Hippe, Yuntian Deng, Bin Zhang, Cindy Orozco Bohorquez, Austin Clyde, Bharat Kale, Danilo Perez-Rivera, Heng Ma, et al. 2023. GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. *The International Journal of High Performance Computing Applications* 37, 6 (2023), 683–705.