Accelerating Python Applications with Dask and ProxyStore

J. Gregory Pauloski
University of Chicago
Argonne National Laboratory
Chicago, Illinois, USA

Klaudiusz Rydzy
Loyola University Chicago
Chicago, Illinois, USA

Valerie Hayot-Sasson
University of Chicago
Argonne National Laboratory
Chicago, Illinois, USA

Ian Foster
University of Chicago
Argonne National Laboratory
Chicago, Illinois, USA

Kyle Chard
University of Chicago
Argonne National Laboratory
Chicago, Illinois, USA

Abstract

Applications are increasingly written as dynamic workflows underpinned by an execution framework that manages asynchronous computations across distributed hardware. However, execution frameworks typically offer one-size-fits-all solutions for data flow management, which can restrict performance and scalability. ProxyStore, a middleware layer that optimizes data flow via an advanced pass-by-reference paradigm, has shown to be an effective mechanism for addressing these limitations. Here, we investigate integrating ProxyStore with Dask Distributed, one of the most popular libraries for distributed computing in Python, with the goal of supporting scalable and portable scientific workflows. Dask provides an easy-to-use and flexible framework, but is less optimized for scaling certain data-intensive workflows. We investigate these limitations and detail the technical contributions necessary to develop a robust solution for distributed applications and demonstrate improved performance on synthetic benchmarks and real applications.

CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**; • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Software libraries and repositories**.

Keywords

Distributed and Parallel Systems, Open-Source Software, Python

1 Introduction

Contemporary computational science applications executed at scale are increasingly written as workflows, collections of many distinct tasks interconnected through data dependencies. This trend has necessitated the development of advanced computational tools and frameworks that can glue software components together and provide a platform for scalable and flexible execution on arbitrary hardware. Dynamic workflow execution frameworks, including Dask [11], Dragon [10], Parsl [1], Ray [6], and TaskVine [13], have emerged as powerful solutions to this challenge in the high-performance Python community. Applications can be expressed as fine-grained tasks, typically a function, with special constructs, such as futures, used to implicitly express inter-task data dependencies. The framework then abstracts the complexities of executing tasks in parallel and managing intermediate data across personal, cloud, or high-performance computing (HPC) systems.

However, this class of systems, which typically use one-size-fits-all solutions to facilitate intermediate data movement, often fail to meet the data flow needs of modern, dynamic, and data-intensive applications. Workflow systems commonly rely on a shared file system, as in Parsl and TaskVine, or peer-to-peer TCP communication, as in Dask Distributed, due to simplicity and availability of these approaches. Thus, workflow systems, and therefore applications, often fail to take advantage of advanced technologies available or suffer from functional but sub-optimal solutions.

Recent work has used the transparent object proxy paradigm to decouple data flow complexities from control flow-optimized execution frameworks [8, 9]. In this case, proxies function as lightweight, wide area references to objects located in arbitrary data stores. Proxies can be communicated cheaply and are resolved just-in-time via performant bulk transfer methods in a manner which is transparent to the consumer code. ProxyStore, a Python framework that implements this paradigm, has found success in many scientific domains and with many distributed execution frameworks [2–4, 14, 15].

Here, we investigate how to build scalable and portable scientific workflows through the careful integration of ProxyStore with Dask Distributed. Dask, with 12.3k stars on GitHub[1] and 4.9M downloads per month in September 2024,[2] is one of the most popular libraries for distributed and parallel computing in Python. We first give a brief overview of Dask Distributed, ProxyStore, and related efforts in optimizing data flow. We then investigate limitations of Dask's data management model and detail the technical contributions necessary to overcome these limitations with ProxyStore. The result is a robust and easy-to-use solution for building sophisticated computational science workflows, which we demonstrate through synthetic performance evaluations and real-world applications.

2 Background and Motivation

Dask [11] is a parallel computing library in Python that enables efficient parallel computations on large datasets by breaking them down into smaller, manageable tasks. Dask Distributed extends the Dask and Python `concurrent.futures` APIs to provide a lightweight and easy-to-use library for distributed computing. A centralized scheduler manages the dynamic execution of tasks across local cores or multiple nodes in a cluster and is optimized for low-latency task dispatching, spending between a 100 μs and 1 ms on each task. However, this overhead can drastically increase when the

[1] Dask GitHub: https://github.com/dask/dask.
[2] Dask Distributed PyPI downloads: https://pypistats.org/packages/distributed.

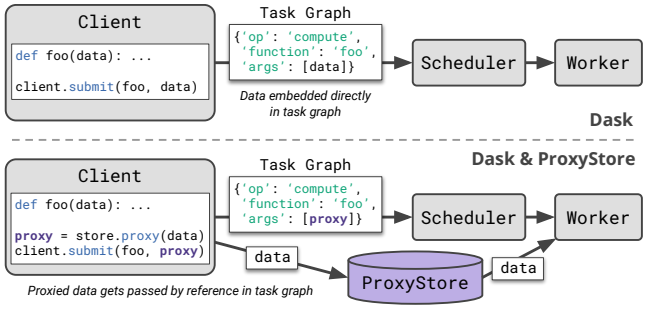

Figure 1: Pass-by-proxy semantics reduce data flow through the Dask scheduler without altering application behavior.

graph of a task is large, such as when task parameters are large or complex. Large task graphs can incur significant I/O overheads in the scheduler for serialization, communication, and deserialization of messages.

Dask provides mechanisms to optimize data transfer: (1) array-like data can be scattered and gathered directly across workers; (2) native interfaces optimize common data operations [3] (e.g., through Dask Arrays, Bags, DataFrames, and Delayed); and (3) objects already located on workers, such as the results of tasks, will be communicated directly between workers rather than through the scheduler. The goal of these solutions is to prefer passing task data by reference rather than embedding data directly in the graph; however, these solutions do not cover all data types or application structures. For example, frequently moving large objects between the client and workers is considered an anti-pattern; Dask prefers that data remain on the worker cluster. Yet, this is a common pattern in scientific applications (e.g., active learning [14, 15]) that is not supported as well by Dask.

ProxyStore [9] is a library that facilitates efficient data flow management in distributed Python applications. The transparent object proxy, a reference-like object, is the core building block of ProxyStore, and, unlike traditional references that are only valid within the virtual address space of a single process, the proxy refers to an object in distributed storage and can be implicitly dereferenced in arbitrary processes, even on remote machines. The proxy is transparent in that it dereferences its target object when used—referred to a just-in-time resolution—and it forwards all operations on itself to its target object. This paradigm results in the best of both pass-by-reference and pass-by-value semantics.

A proxy is initialized with a factory, a self-contained callable object that is invoked when the proxy is resolved to retrieve the target object. This self-contained nature and transparency of the proxy means a consumer is not aware of the low-level communication mechanisms used by the proxy; rather, this is unilaterally determined by the producer of the proxy. This paradigm improves performance and portability by reducing transfer overheads through intermediaries, abstracting low-level communication methods, and reducing code-complexity.

ProxyStore separates the high-level interface, the `Store`, responsible for creating proxies from the low-level interface, the `Connector`, responsible for interfacing with the byte-level mediated communication and storage channels. Connectors to many storage systems (object stores, shared file-systems) and transfer protocols (Grid FTP, TCP, RDMA, and WebRTC) are provided. These interfaces have been used to build high-level patterns for distributed futures, object streaming, and distributed memory management [8].

Related work has investigated ways to improve the performance of data-intensive workflows with Dask. Dask provides a UCX communication protocol implementation as an alternative to the TCP default to leverage advanced networking technologies such as Inifiniband or NVLink. Later work developed an MPI-based communication interface for Dask for GPU-accelerated programs [12]. TaskVine [13], a distributed workflow engine that exploits node-local storage to optimize task placement and execution, and Ray [6], a popular distributed computing library, provide alternative scheduler implementations for Dask workflows. These solutions can yield considerable performance gains in certain applications but also have limited deployment scenarios. ProxyStore, in contrast, provides more fine-grained data flow customization with wider support for communication protocols and storage mediums.

3 Integration Model

ProxyStore can alleviate data transfer overheads in Dask by proxying large task objects instead of embedding them directly in the task graph (Fig 1). Importantly, use of ProxyStore does not require modification to task code and is not mutually exclusive with Dask optimization options. Here, we discuss the methods for integrating ProxyStore into Dask applications, the technical challenges overcome to ensure compatibility and performance, and the features added to support development of robust applications.

Methods: There are three methods, exemplified in Fig 2, to integrate ProxyStore into a Dask application: (1) manually proxy objects using ProxyStore's existing tooling, (2) use our custom Dask client to automatically proxy objects; or (3) use our custom executor interface to intelligently proxy objects and manage memory. For simple applications, the manual approach works well, but it can require significant code changes in more sophisticated applications. The custom client provides a drop-in replacement for existing Dask applications. The `StoreExecutor`, which extends Python's `concurrent.futures` interface, is the most powerful approach: it is compatible with many other executor client types, such as those provided by Parsl and TaskVine; custom policies can be defined to determine what objects to automatically proxy and, when combined with ProxyStore's `MultiConnector`, what mediated storage option to use; and it uses ProxyStore's ownership model [8], inspired by Rust's ownership and borrowing semantics, to perform safe and automatic memory management of proxies.

Compatibility: Dask performs introspection on objects included in task graphs to enable optimizations. Each task has an associated key, a hash of the function and arguments. The scheduler uses the key to reuse results of previously computed pure functions (functions that always return the same result given the same inputs). Similarly, Dask inspects task object types to apply specialized serializers. These optimizations enhance performance but interacted poorly with proxy types. For example, Dask serialization would crash when accessing the `__module__` property of a proxy, and

[3] https://docs.dask.org/en/stable/best-practices.html#load-data-with-dask

```python
from dask.distributed import Client
from proxystore.ex.connectors.daos import DAOSConnector
from proxystore.store import Store

client = Client()
connector = DAOSConnector(pool=..., container=...)

with Store('example', connector) as store:
    proxy = store.proxy([1, 2, 3])
    future = client.submit(sum, proxy)
    assert future.result() == 6
```

(a) A proxy can be manually created via the `Store` interface and passed directly to tasks in place of the actual object.

```python
from proxystore.ex.plugins.distributed import Client
from proxystore.ex.connectors.daos import DAOSConnector
from proxystore.store import Store

connector = DAOSConnector(pool=..., container=...)

with Store('example', connector) as store:
    client = Client(ps_store=store, ps_threshold=1000)
    future = client.submit(sum, [1, 2, 3])
    assert future.result() == 6
```

(b) The custom Dask Distributed `Client` will automatically proxy task input and output objects larger than a user-defined threshold (e.g., 1 kB).

```python
import sys
from dask.distributed import Client
from proxystore.ex.connectors.daos import DAOSConnector
from proxystore.store import Store
from proxystore.store.executor import StoreExecutor

client = Client()
connector = DAOSConnector(pool=..., container=...)

with StoreExecutor(
    client,
    store=Store('example', connector),
    should_proxy=lambda x: sys.getsizeof(x) >= 1000,
) as executor:
    future = executor.submit(sum, [1, 2, 3])
    assert future.result() == 6
```

(c) The `StoreExecutor` can combine a `Store` and Dask `Client` and supports custom policies for what objects should be automatically proxied (here, objects larger than 1 kB) and automatically manages proxy lifetimes.

Figure 2: ProxyStore is easily compatible with existing applications. Here we demonstrate the three integration patterns. The `DAOSConnector`, introduced in Sec 3, is used, but this specific connector can be exchanged depending on the application requirements and execution environment.

hashing or checking the type of a proxy would resolve the proxy, incurring unexpected I/O costs. We resolved this by creating a custom implementation of Python's `@property` decorator and modified the proxy to cache common read-only attributes of a proxied object. These include the module path, the class type, and the hash of the target object to ensure that a proxy need not be resolved when Dask accesses common object metadata.

Performance: We improved ProxyStore's performance for scientific workloads by overhauling the serialization system to minimize memory copies and support custom serialization mechanisms for specific data types. We have provided initial support for NumPy arrays, Pandas DataFrames, and Polars DataFrames. Serialization of these types is 2–3× faster compared to pickle, which ProxyStore previously used.

We also extend ProxyStore to support Distributed Asynchronous Object Storage (DAOS) as a mediated storage system. DAOS is a distributed object store designed for high-speed non-volatile memory like Intel Optane [5] and NVMe and is available on next-generation compute clusters like Aurora at the Argonne Leadership Computing Facility. DAOS is typically deployed across a machine in a similar fashion to a shared file system like Lustre. Thus, using the DAOS within ProxyStore is easy—minimal configuration is required—and performance is superior to shared file systems. The user need only provide the name of their DAOS pool and container to use.

Robustness: The introduction of type hints and static type checkers such as mypy has significantly improved code quality and maintainability, ultimately leading to more robust software. The proxy model, however, relies strongly on Python's duck typing and, thus, code that uses ProxyStore cannot be statically analyzed and validated, leading to often cryptic errors when a proxy type is used incorrectly at runtime. We created a mypy extension that can statically infer usage of proxy types. For example, mypy can understand that any attributes or methods on a type `T` are also available on a `Proxy[T]` or that a function that accepts a `ProxyOr[T]` should work with a `T` or `Proxy[T]`. This tool ensures that scientific software developers can write code that will work with and without ProxyStore, improving code compatibility and maintainability.

4 Evaluation

We used the TaPS benchmark suite [7] to evaluate the performance benefits of using ProxyStore within Dask applications. Experiments were performed using the Sunspot system at the Argonne Leadership Computing Facility. [4] Sunspot has 128 nodes interconnected by an HPE Slingshot 11 network and a high-performance DAOS storage system. Each node contains two Intel Xeon Max CPUs with 52 physical cores, 64 GB of high-bandwidth memory, 128 GB of DDR5 memory per CPU, and six Intel Data Center Max GPUs. We used Python 3.11, Dask Distributed 2024.7.1, ProxyStore 0.7.1, ProxyStore Extensions v0.1.4, and TaPS 0.2.1. Analysis, code, and results are available at https://github.com/proxystore/hppss24-demo.

We performed ProxyStore experiments using Redis due to DAOS outages on Sunspot at the time of writing; a Redis server was started on the rank 0 node of each batch job. Configuring ProxyStore to use DAOS would be even easier, as described in Sec 3, and we expect comparable performance outcomes due to DAOS leveraging NVMe storage distributed throughout the racks of the cluster.

Overheads: We measure the round-trip time of no-op tasks with payloads of varying sizes in Fig 3. This experiment represents a worst-case scenario for the Dask scheduler: all data is sent between the client and workers and no data is reused across multiple tasks. Using ProxyStore's pass-by-proxy model improves round-trip time for larger task payloads (> 100 kB) by up to 50%. This improvement is attributed to (1) smaller messages to be serialized and communicated, (2) less data transferred through the scheduler, and (3)

[4] This work was done on a pre-production supercomputer with early versions of the Aurora software development kit.

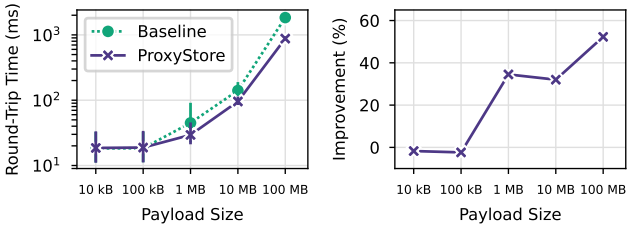

Figure 3: (Left) No-op task round-trip time with various payload sizes. (Right) Relative improvement in round-trip time compared to the baseline when using ProxyStore.

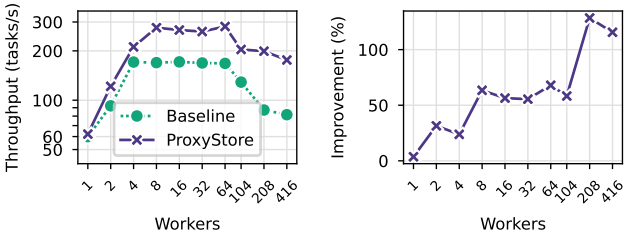

Figure 4: (Left) No-op task throughput with various worker counts. Tasks consume and produce 1 MB of random data. (Right) Relative improvement in throughput compared to the baseline when using ProxyStore. ProxyStore alleviates data flow burdens from the Dask scheduler, enabling the scheduler to dispatch tasks faster.

improvements to ProxyStore's serialization that reduce memory copies.

Scaling: We measure task throughput with and without ProxyStore as a function of the number of Dask workers n. Each node hosts up to 104 workers, the number of physical cores per node. We execute 10 000 tasks that consume and produce 1 MB of random data (chosen based on the results in Fig 3). An initial batch of n tasks are submitted; as current tasks complete, new tasks are submitted until all tasks are finished. Tasks are essentially no-ops besides the result data generation which takes only $O(1)$ ms; thus, the goal of this experiment is to stress the Dask scheduler and understand its limits. As depicted in Fig 4, task throughput with Dask quickly plateaus around 170 tasks per second and degrades when utilizing 104 workers. Use of ProxyStore alleviates data transfer burdens from the scheduler, enabling higher sustained throughput; however we still observe the same drop in performance at 104 workers which may indicate the presence of performance limitations in the Dask scheduler that are independent of data volume.

Applications: We use three reference applications from TaPS representing an array of data patterns. Cholesky decomposition has short tasks that consume and produce large arrays, federated learning has long tasks that consume and produce large models, and molecular design has short tasks that consume and produce small datasets and models. We chose these three applications because they are implemented in a manner which accentuates data transfer between the client and workers. As demonstrated in Fig 5, ProxyStore yields the greatest benefits to Dask applications with larger

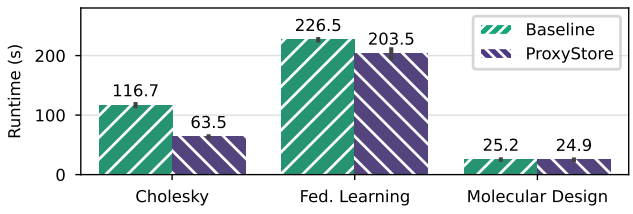

Figure 5: ProxyStore can reduce Dask overheads applications that embed large objects in the task graph, such as the Cholesky decomposition example and federated learning simulation provided by TaPS.

tasks payloads and shorter running tasks—applications where task overheads represent a larger proportion of overall runtime.

5 Conclusion

The pass-by-proxy model of ProxyStore is a viable alternative to data flow management in distributed Dask applications. We discussed the ways in which Dask applications can be extended with ProxyStore, the diverse storage systems and communications channels supported by ProxyStore, and the technical contributions necessary to make the integration possible. Experiments show that ProxyStore reduces task overheads when task graphs are large, improves task throughput at scale, and accelerates applications which suffer from I/O bottlenecks in the Dask scheduler.

Acknowledgment

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

[1] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36.

[2] Nicholson Collier, Justin M. Wozniak, Abby Stevens, Yadu Babuji, Mickaël Binois, Arindam Fadikar, Alexandra Würth, Kyle Chard, and Jonathan Ozik. 2023. Developing Distributed High-performance Computing Capabilities of an Open Science Platform for Robust Epidemic Analysis. arXiv:2304.14244 [cs.DC]

[3] Gautham Dharuman, Logan Ward, Heng Ma, Priyanka V. Setty, Ozan Gokdemir, Sam Foreman, Murali Emani, Kyle Hippe, Alexander Brace, Kristopher Keipert, Thomas Gibbs, Ian Foster, Anima Anandkumar, Venkatram Vishwanath, and Arvind Ramanathan. 2023. Protein Generation via Genome-scale Language Models with Bio-physical Scoring. In *SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA). Association for Computing Machinery, New York, NY, USA, 95–101.

[4] Hassan Harb, Sarah N. Elliott, Logan Ward, Ian T. Foster, Stephen J. Klippenstein, Larry A. Curtiss, and Rajeev Surendran Assary. 2023. Uncovering novel liquid organic hydrogen carriers: A systematic exploration of chemical compound space using cheminformatics and quantum chemical methods. *Digital Discovery* 2 (2023), 1813–1830. Issue 6. https://doi.org/10.1039/D3DD00123G

[5] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. 2020. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In *Supercomputing Frontiers*, Dhabaleswar K. Panda (Ed.). Springer International Publishing, Cham, 40–54.

[6] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 561–577.

[7] J. Gregory Pauloski, Valerie Hayot-Sasson, Maxime Gonthier, Nathaniel Hudson, Haochen Pan, Sicheng Zhou, Ian Foster, and Kyle Chard. 2024. TaPS: A Performance Evaluation Suite for Task-based Execution Frameworks. In *IEEE 20th International Conference on e-Science*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/e-Science62913.2024.10678702

[8] J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Alexander Brace, André Bauer, Kyle Chard, and Ian Foster. 2024. Object Proxy Patterns for Accelerating Distributed Applications. https://arxiv.org/abs/2407.01764

[9] J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Nathaniel Hudson, Charlie Sabino, Matt Baughman, Kyle Chard, and Ian Foster. 2023. Accelerating Communications in Federated Applications with Transparent Object Proxies. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 59, 15 pages. https://doi.org/10.1145/3581784.3607047

[10] Nick Radcliffe, Kent Lee, and Pete Mendygral. 2023. Dragon Proxy Runtimes and Multi-system Workflows. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, New York, NY, USA, 648–651.

[11] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling.

[12] Aamir Shafi, Jahanzeb Maqbool Hashmi, Hari Subramoni, and Dhabaleswar Kumar Panda. 2021. Efficient MPI-based Communication for GPU-Accelerated Dask Applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE Computer Society, New York, NY, USA, 277–286. https://api.semanticscholar.org/CorpusID:231692991

[13] Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennessee, Ben Tovar, and Douglas Thain. 2023. TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1978–1988.

[14] Logan Ward, J. Gregory Pauloski, Valerie Hayot-Sasson, Ryan Chard, Yadu Babuji, Ganesh Sivaraman, Sutanay Choudhury, Kyle Chard, Rajeev Thakur, and Ian Foster. 2023. Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources. In *Heterogeneity in Computing Workshop*. IEEE Computer Society, New York, NY, USA. https://doi.org/10.48550/ARXIV.2303.08803

[15] L. Ward, G. Sivaraman, J. Pauloski, Y. Babuji, R. Chard, N. Dandu, P. C. Redfern, R. S. Assary, K. Chard, L. A. Curtiss, R. Thakur, and I. Foster. 2021. Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 9–20. https://doi.org/10.1109/MLHPC54614.2021.00007