TAPS: A Performance Evaluation Suite for Task-based Execution Frameworks

J. Gregory Pauloski,*† Valerie Hayot-Sasson,*† Maxime Gonthier,*† Nathaniel Hudson,*†
Haochen Pan,* Sicheng Zhou,* Ian Foster,*† and Kyle Chard*†
*Department of Computer Science, University of Chicago, Chicago, IL, USA
†Data Science and Learning Division, Argonne National Laboratory, Lemont, IL, USA

***Abstract*—Task-based execution frameworks, such as parallel programming libraries, computational workflow systems, and function-as-a-service platforms, enable the composition of distinct tasks into a single, unified application designed to achieve a computational goal. Task-based execution frameworks abstract the parallel execution of an application's tasks on arbitrary hardware. Research into these task executors has accelerated as computational sciences increasingly need to take advantage of parallel compute and/or heterogeneous hardware. However, the lack of evaluation standards makes it challenging to compare and contrast novel systems against existing implementations. Here, we introduce TAPS, the Task Performance Suite, to support continued research in parallel task executor frameworks. TAPS provides (1) a unified, modular interface for writing and evaluating applications using arbitrary execution frameworks and data management systems and (2) an initial set of reference synthetic and real-world science applications. We discuss how the design of TAPS supports the reliable evaluation of frameworks and demonstrate TAPS through a survey of benchmarks using the provided reference applications.**

***Index Terms*—Performance Evaluation, Parallel Computing, Open-Source, Python, Workflows**

I. Introduction

Task-based execution frameworks, such as Dask [1], Parsl [2], and Ray [3], have enabled many advances across the sciences. These *task executors* manage the complexities of executing the tasks comprising an application in parallel across arbitrary hardware. Decoupling the application logic (e.g., what tasks to perform, how data flow between tasks) from the execution details (e.g., scheduling systems or communication protocols) simplifies development and results in applications which are portable across diverse systems. Task executors come in many forms, from a simple pool of processes to sophisticated workflow management systems (WMSs), and the rapid increase in the use of task-based applications across the computational sciences has spurred further research in the area.

Consistent and reliable benchmarking is fundamental to evaluating advances within a field over time. Benchmarks and other performance evaluation systems offer a common ground and objective metrics that enable researchers to assess the efficiency, performance, scalability, and robustness of their solutions under controlled conditions. Benchmarks foster transparency and reproducibility, ensuring that results can be consistently replicated and verified by others in the field. This, in turn, accelerates the pace of innovation as researchers can identify best practices, optimize existing methods, and uncover new areas for improvement. Benchmarks facilitate meaningful comparisons between competing approaches—a valuable aspect for researchers, reviewers, and readers alike.

Access to open source benchmarks democratizes research, and many fields have found great success through the creation of standards. LINPACK [4], for example, is used to evaluate the floating point performance of hardware systems. The Transaction Processing Performance Council (TPC) [5] provides a variety of standard benchmarks for database systems, and UnixBench [6] can evaluate basic performance of Unix-like systems from file copies to system call overheads. Machine learning (ML) has demonstrated this success with benchmarks for every level of the ML stack. MLPerf [7, 8] has continued to support the development of ML hardware and frameworks. Novel algorithms are compared against prior work by using open source datasets, as exemplified by the Papers with Code Leaderboards [9] that comprise results of tens of thousands of papers across thousands of datasets.

However, the parallel application and workflows communities lack such established benchmarks. The NAS parallel benchmarks date back to the 1990s [10]. For workflows, with the exception of a few common applications (e.g., Montage [11, 12]), papers typically evaluate their solutions on purpose-built synthetic benchmarks or forks of real world science applications. Unfortunately, the ad hoc nature of these solutions means that the code is often not open sourced, not maintained beyond publication of the corresponding paper, or so specific to an implementation that it is challenging to appropriately compare against in later works. Recent work has introduced a standard for recording execution traces and tools for analyzing those traces [13], but there remains a need for realistic reference applications for benchmarking.

To address these challenges, we introduce TAPS, the Task Performance Suite, a standardized framework for evaluating task-based execution frameworks against synthetic and realistic science applications. With TAPS, applications can be written in a framework-agnostic manner and then benchmarked using any one of many supported task executors and data management systems. We make the following contributions:

1) TAPS, a standardized benchmarking framework for task-based applications with an extensible plugin system for comparing task executors and data management systems. TAPS is available at https://github.com/proxystore/taps.

2) Support for popular task executors (Dask, Globus Compute, Parsl, Ray, and TaskVine) and data management systems (shraed file systems and ProxyStore).
3) Reference implementations within TAPS for six real (Cholesky factorization, protein docking, federated learning, MapReduce, molecular design, and Montage) and two synthetic applications.
4) Insights into the performance of the reference implementations across the supported frameworks.

The rest of this paper is as follows: Sec II discusses related work; Sec III describes the design and implementation details of the TAPS framework; Sec IV introduces the initial set of applications provided by TAPS; Sec V presents our experiences using TAPS to evaluate system components; and Sec VI summarizes our contributions and future directions.

II. Background and Related Work

Task executors, which manage the execution of tasks in parallel across distributed resources, come in many forms. A *task* refers to discrete unit of work, and tasks are combined into a larger *application*. Tasks can take data as input, produce output data, and may have dependencies with other tasks; i.e., a dependent task cannot start until a preceding tasks completes. Dask Distributed, Python's `ProcessPoolExecutor`, Globus Compute [14], Radical Pilot [15], and Ray all provide mechanisms for executing tasks in parallel across distributed systems.

Workflow management systems (WMSs), a subset of task executors, are designed to define, manage, and execute workflows represented by a directed acyclic graph (DAG) of tasks. WMSs commonly provide mechanisms for automating and optimizing task flow, monitoring, and resource management. WMSs can be categorized as supporting explicit or implicit dataflow patterns. Explicit systems, such as Apache Airflow [16], Fireworks [17], Makeflow [18], Nextflow [19], Pegasus [20], and Swift [21], rely on configuration files or domain specific languages (DSLs) to statically define a DAG before execution. Implicit systems, such as Dask Delayed, Parsl, Swift/T [22], and TaskVine [23], derive the application's dataflow through the dynamic evaluation of a procedural script.

Performance evaluation of task executors is challenging due to a lack of standards. Frameworks provide examples designed to aid in learning the framework, but these are often too trivial to be used in benchmarking. Pegasus provides a catalogue of real, end-to-end scientific workflows in AI, astronomy, and bio-informatics which are suitable for benchmarking [24]; Dask maintains a repository of performance benchmarks [25]; WorkflowHub provides a service for sharing scientific workflows [26]; and Workbench [18], designed for analyzing workflow patterns, was released alongside Makeflow. However, these reference applications and benchmarks are typically valid only for evaluating optimizations within the framework they were implemented in. In other words, the majority of these code bases are not suitable for comparing different task executors. This also means available benchmarks are susceptible to code rot if maintenance of the associated framework ceases.

Porting benchmark applications between frameworks is onerous when the structure and syntax is completely different. Subtle errors in the ported implementation can lead to inaccurate comparisons between systems. Access to datasets or sufficient compute resources for certain applications can further hinder the creation of realistic benchmarking applications. To assuage these challenges within the workflows community, prior work [27] published a gallery of execution traces from real workloads using Pegasus, a synthetic workflow generator, and a simulator framework. WfCommons [13] introduces a standardized format for representing execution logs (WfFormat), an open source package for analyzing logs and generating synthetic logs (WfGen), and a workflow execution simulator (WfSim). WfCommons currently provides 180 execution instances from three workflow systems (Makeflow, Nextflow, and Pegasus). Similarly, WRENCH [28] provides a WMS simulation framework built on SimGrid [29]. In contrast, an Application Skeleton supports the design and development of systems by mimicking the performance of a real application [30].

FunctionBench [31], FaaSDom [32], and SeBS [33] address a similar set of challenges as TAPS but in the context of cloud-hosted function-as-a-service (FaaS) platforms. SeBS provides a benchmark specification, a general model of FaaS platforms, and an implementation of the framework and benchmarks. This model is valuable because each benchmark is platform agnostic, relying only on the abstract FaaS model provided by SeBS. Implementing the concrete model for a new platform need only be performed once, and then any benchmark can be executed on that platform. Part of SeBS's platform model is support for persistent and ephemeral cloud storage systems. Supporting the evaluation of the compute and data aspects of task-based applications is crucial, but currently lacking outside of specific areas (i.e., SeBS for FaaS).

III. Design and Implementation

TAPS is a Python package that provides a common framework for writing task-based, distributed applications; a plugin system for running applications with arbitrary task executors and data management systems; and a benchmarking framework for running experiments in a reproducible manner. We choose Python for its pervasiveness in task-based, distributed applications, and we describe here the high level concepts that make the framework possible and the implementation details. Our goal is to create an easy-to-use framework for researchers to benchmark novel systems and an extensible framework so future applications and plugins can be incorporated into TAPS.

A. Application Model

TAPS provides a framework for the creation and execution of application benchmarks. As described in Sec II, applications are composed of tasks which are the remote execution of a function which takes in some data and produces some data. Tasks can have dependencies such that the result of one task is consumed by one or more tasks.

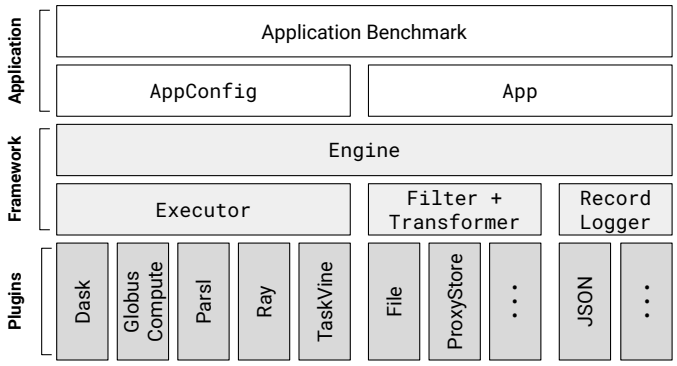

Fig. 1. Overview of the TAPS stack.

```python
@register('app')
class FooAppConfig(AppConfig):
    name: str = 'foo'
    sleep: float = Field(description='...')
    count: int = Field(1, description='...')

    def get_app(self) -> FooApp: ...

class FooApp:
    def __init__(self, ...) -> None: ...

    def run(self, engine: Engine, run_dir: Path) -> None:
        ...

    def close(self) -> None: ...
```

Listing 1. Example application structure within TAPS.

Supporting applications written using the explicit and implicit workflow models described in Sec II is challenging because the two philosophies are fundamentally at odds with each other and, within the scope of explicit systems, the different configuration formats and use of DSLs further complicates the design of a unified, abstract task executor interface.

TAPS supports writing applications as Python code using implicit dataflow dependencies. (Though, it is not a requirement that tasks have dataflow dependencies within an application.) We take this approach for two reasons. First, the scope of applications compatible with implicit models is a super-set of those compatible with explicit models. Specifically, WFMs which use a static graph for execution are not expressive enough for writing more dynamic and procedural applications, whereas the implicit model enables arbitrarily complex applications composed through a procedural program. Second, WMFs which use DSLs require the application design to be tightly coupled to the WMF. This inherently makes it challenging to construct an application that is compatible with a multitude of frameworks.

B. Writing Applications

In TAPS, an application is composed of two parts: an `AppConfig` and an `App` class (see Fig 1). The `AppConfig` contains all configuration options required to execute the corresponding applications (e.g., hyperparameters, paths to datasets, or flags). `AppConfig` exposes a `get_app()` method which initializes an `App` instance from the user-specified configuration. `App.run()` is the entry point to the application code and is invoked with two arguments: an `Engine` instance (discussed in detail in Sec III-C) and the path to a unique directory for the current application invocation. The `run()` method can contain arbitrary code, provided application tasks are executed via the provided `Engine` interface.

TAPS provides a CLI framework for executing application benchmarks. For example, the `foo` application in Listing 1 is started with: `python -m taps.run --app foo {args}`. An `AppConfig` can be registered with the CLI using the `@register('app')` decorator. This will automatically add the application's name as one of the CLI choices and add CLI arguments based on the `AppConfig` attributes.

When invoked, the CLI (1) constructs an `AppConfig` instance from the user's arguments, validating that options can be parsed into the correct type and that all required arguments are present; (2) initializes the `App` using `get_app()`; (3) constructs an `Engine` according to user-supplied arguments; and (4) invokes `App.run()` to execute the application benchmark. The framework automatically writes a configuration file, log files, and task record files to the run directory. The configuration file contains a record of all configuration options used to execute the application. A configuration file path can be provided to the CLI as an alternative to CLI arguments; thus, configuration files can be shared for reproducibility.

C. Application Execution

The `Engine` is the unified interface used by applications to execute tasks and exposes an interface similar to Python's `concurrent.futures.Executor`. The `Engine` interface must be expressive enough to build arbitrary applications yet simple enough to incorporate third-party task executors and other plugins. We chose to adopt a model similar to Python's `Executor` because it is a *de facto* standard for managing asynchronous task execution across the Python ecosystem and many third-party libraries provide `Executor`-like implementations, including Dask Distributed, Globus Compute, Loky, TaskVine, and Parsl. An additional benefit of this choice is that it is trivial to port applications already using an `Executor` interface into a TAPS application.

`Executor` is an abstract class with two primary methods, `submit()` and `map()`, designed to execute functions asynchronously. The `submit()` method takes a callable object and associated arguments, schedules the callable for execution, and returns back to the client a `Future` that will eventually contain the result of the callable. `Engine` implements both of these methods, but returns `TaskFuture` objects rather than `Future` instances. Functionally, `TaskFuture` behaves like `Future` but includes additional functionality for performance monitoring and task dependency management.

An `Engine` is created from four components: `Executor`, `Transformer`, `Filter`, and `RecordLogger`. This conceptual hierarchy of components in TAPS is illustrated in Fig 1. The dependency model approach used by the `Engine` means that component plugins can be easily created and/or swapped to compare, for example, different task executors or data management systems. Further, the `Engine` can be extended with

additional components in the future to enhance benchmarking capabilities.

D. Task Executor Model

The fundamental component of the `Engine` is an `Executor`, an interface to the underlying task executor. We choose the `Executor` model again for the same reasons as with the `Engine`. In Sec III-E, we describe the details of each executor currently supported in TAPS. Similar to the `App` model, TAPS has a notion of a `ExecutorConfig` which can be registered with the framework to automatically add argument parser groups for the specific executor. `ExecutorConfig` has a method, `get_executor()`, which will initialize an instance of the executor from the user specified configuration.

A limitation of Python's `Executor` interface is the lack of support for dataflow dependencies between tasks. Some `Executor` implementations (Dask Distributed, Parsl, and TaskVine) do support implicit dataflow dependencies by passing the future of one task as input to one or more tasks, but many others (e.g., Python's `ProcessPoolExecutor` and Globus Compute) do not. The `Engine` requires it's `Executor` to support implicit dataflow patterns with futures, so TAPS provides a `FutureDependencyExecutor` wrapper to add this functionality if needed. This wrapper scans task inputs for futures and will delay submission of a task until the results of all input futures are available (in an asynchronous, non-blocking manner).

E. Supported Task Executors

Here, we briefly describe the task executors currently supported by TAPS (summarized in Table I). As previously mentioned, the plugin system makes it easy to support more executors in the future, but our initial goal is to support a broad range. We support Python's `ProcessPoolExecutor` which provides a good baseline for low-overhead, single-node execution. We also support the `ThreadPoolExecutor`, but this is primarily intended to support development and quick testing because Python's Global Interpreter Lock prevents true parallelism with threading.

Dask Distributed [1] provides dynamic task scheduling and management across worker processes distributed across cores within a node or across several nodes. Tasks in Dask are Python functions which operate on Python objects; the scheduler tracks these task in a dynamic DAG. Globus Compute [14] is a cloud-managed function-as-a-service (FaaS) platform which can execute Python functions across federated compute systems. Globus Compute provides an `Executor` interface but does not manage dependencies between functions. Parsl [2] is a parallel programming library for Python with comprehensive dataflow management capabilities. Parsl supports many execution models including local compute, remote compute, and batch scheduling systems. Ray [3] is a general purpose framework for executing task-parallel and actor-based computations on distributed systems in a scalable and fault tolerant manner. TaskVine [23] executes dynamic DAG workflows with a focus on data management features including transformation, distribution, and task data locality.

F. Task Data Model

Optimizing the transfer of task data and placement of tasks according to where data reside is a core feature of many task executors. To support further research into data management, TAPS supports a plugin system for *data transformers*. A data transformer is an object that implements the `Transformer` protocol. This protocol defines two methods: `transform` which takes an object and returns an identifier, and `resolve`, the inverse of `transform`, which takes an identifier and returns the corresponding object. Data transformer implementations can implement object identifiers in any manner, provided identifier instances are serializable. For example, an identifier could simply be a UUID corresponding to a database entry containing the serialized object.

A `Filter` is a callable object, e.g., a function, that takes an object as input and returns a boolean indicating if the object should be transformed by the data transformer. The `Engine` uses the `Transformer` and `Filter` to transform the positional arguments, keyword arguments, and results of tasks before being sent to the `Executor`. For example, every argument in the positional arguments tuple which passes the filter check is transformed into an identifier using the data transformer. Each task is encapsulated with a wrapper which will *resolve* any arguments that were replaced with identifiers when the task executes. The same occurs in reverse for a task's result.

`Filter` implementations based on object size, pickled object size, and object type are provided. We initially provide two `Transformer` implementations: `PickleFileTransformer` and `ProxyTransformer`. The `PickleFileTransformer` pickles objects and writes the pickled data to a file. The `ProxyTransformer` creates proxies of objects using the ProxyStore library [35, 36]. ProxyStore provides a pass-by-reference like model for distributed Python applications and supports a multitude of communication protocols including DAOS [37], Globus [38, 39], Margo [40], Redis [41], UCX [42], and ZeroMQ [43].

G. Logging and Metrics

Recording logs and metrics for post-execution analysis is core to any benchmarking framework. TAPS records the high-level application logs and low-level details of each executed task. The `RecordLogger` interface is used to log records of all tasks executed by the `Engine`. These records include metrics and metadata of the task, such as the unique task ID, the function name, task IDs of any parent tasks, submission time, completion time, data transformation and resolution times, and execution makespan. By default, TAPS uses the `JSONRecordLogger` which logs a JSON representation of the task information to a line-delimited file. In future work, we would also like to support WfCommon's WfFormat.

H. Task Life-cycle

An application creates a task by submitting a Python function with corresponding arguments to `Engine.submit()`

TABLE I
OVERVIEW OF THE EXECUTION ENGINES SUPPORTED WITHIN TAPS.

			Scheduler			Deployment	
Name	Reference	Languages	Distributed	Dataflow	Locality-Aware	Distributed	Batch Systems
`ThreadPoolExecutor`	[34]	Python					
`ProcessPoolExecutor`	[34]	Python					
Dask Distributed	[1]	Python		✓	✓	✓	✓
Globus Compute	[14]	Python				✓	✓
Parsl	[2]	Python		✓		✓	✓
Ray	[3]	C++, Java, Python	✓	✓	✓	✓	✓
TaskVine	[23]	C, Python		✓	✓	✓	✓

which returns a corresponding `TaskFuture`. (Applications can also create many tasks by mapping a function onto an iterable of arguments via `Engine.map()`. For simplicity, we discuss single task submission here, but the same process applies with map.) The `Engine` generates a unique ID for the task and wraps the function in a task wrapper. The `Transformer` is then applied to the arguments according to the `Filter`. Then, the wrapped function and arguments (some or all of which may have been transformed) are passed to the `Executor` for scheduling and execution. The `Executor` returns a future specific to the executor type (e.g., a Globus Compute future for a `GlobusComputeExecutor`). This low-level future is then wrapped in a `TaskFuture`, and the `TaskFuture` is returned to the client. If a `TaskFuture` were passed as input to a task, the `Engine` will also replace the `TaskFuture` with the low-level future of the `Executor`. This is necessary to ensuring the `Executor` can schedule the tasks according to the implicit inter-task dependencies.

When a task begins execution, the task wrapper will record information about the execution to propagate back to the `Engine`. The task wrapper will also resolve any transformed arguments prior to invoking the original function provided by the client and possibly transform the function result. The completion of a task (i.e., when the result of the future is set) will trigger a callback which logs all of a task's information and metrics. If the function result was transformed, the `TaskFuture` will resolve the result inside of `TaskFuture.result()`.

IV. APPLICATIONS

We initially provide eight applications within TAPS, summarized in Table II and Fig 2. These distributed and parallel applications are diverse, spanning many domains, datasets, and structures to support comprehensive performance evaluation of existing and future systems.

A. Cholesky Factorization

Cholesky factorization (also referred to as decomposition) is a fundamental linear algebra operation used in many domains. The tiled version of Cholesky factorization has been studied extensively, for example, in the context of NUMA machines [56] and from the perspective of communication overhead [57]. The tiled version produces an arbitrarily complex DAG depending on the number of tiles, which makes it a good candidate for evaluating task executors. The 4×4 tiled DAG is portrayed in Fig 2.

The `cholesky` application implements a tiled Cholesky factorization which, given an input matrix A that is positive-definite, computes L where $A = L \times L^T$ [44]. The algorithm comprises four task types: GEMM, a tiled matrix multiplication requiring three inputs; SYRK, a symmetric rank-k update requiring three inputs; TRSM, which solves a triangular matrix equation with two inputs; and POTRF, an untiled Cholesky factorization which operates on a tile of A.

The `cholesky` application takes two user-supplied parameters: N, the side length of the input matrix to generate, and b, the side length of each square block in the tiled matrix. As b approaches N, the number of blocks in the tiled matrix, and thus the number of tasks required for the factorization, decreases. Given B, a randomly generated $N \times N$ matrix, the positive definite input matrix A is computed by using $A = (B + B^T) + \delta I$, where $\delta = N$ and I is the $N \times N$ identity matrix.

B. Protein Docking

Protein docking aims to predict the orientation and position of one molecule to another. It is commonly used in structure-based drug design as it helps predict the binding affinity of a ligand (the candidate drug) to the target receptor. Simulations required to compute docking score are computationally expensive, and the search-space of potential molecules can be expansive. To improve the time-to-solution, this implementation of protein docking is parallelized and includes ML-in-the-loop. A model is trained using the results of previous simulations to predict which molecules are most likely to have strong binding scores, thereby significantly reducing the search space.

The `docking` workflow is based on a reference implementation written in Parsl [45]. The workflow uses Autodock Vina [58] for the docking simulations and scikit-learn [59] to construct a KNN-based transformer for the ML model. It is composed of three task types: (1) data preparation, (2) simulation, and (3) ML training and inference. The workflow has two primary parameters: a CSV file containing the search space of candidate ligands and their associated SMILES strings and a PDBQT file containing the target receptor. One of the tasks launches a subprocess to execute a `set-element.tcl` script (provided in the reference implementation) that adds

TABLE II
OVERVIEW OF THE APPLICATIONS IMPLEMENTED WITHIN TAPS.

Name	Reference	Domain	Dataset(s)	Task Types(s)	Data Format(s)
`cholesky`	[44]	Linear Algebra	Randomly Generated	Python Functions	In-memory
`docking`	[45]	Drug Discovery	C-ABL Kinase Domain [46], Zinc Ord. Compounds [47]	Executable, Python Functions	File
`fedlearn`	[48]	Machine Learning	MNIST [49], FEMNIST [50], CIFAR-10/100 [51]	Python Functions	In-memory
`mapreduce`	[52]	Text Analysis	Randomly Generated, Enron Corpus [53]	Python Functions	File, In-memory
`moldesign`	[54]	Molecular Design	QM9 [55]	Python Functions	In-memory
`montage`	[11]	Astronomy	Montage Images [11]	Executable	File
`failures`	—	—	—	Executable, Python Functions	File, In-memory
`synthetic`	[35]	—	Randomly Generated	Python Functions	In-memory

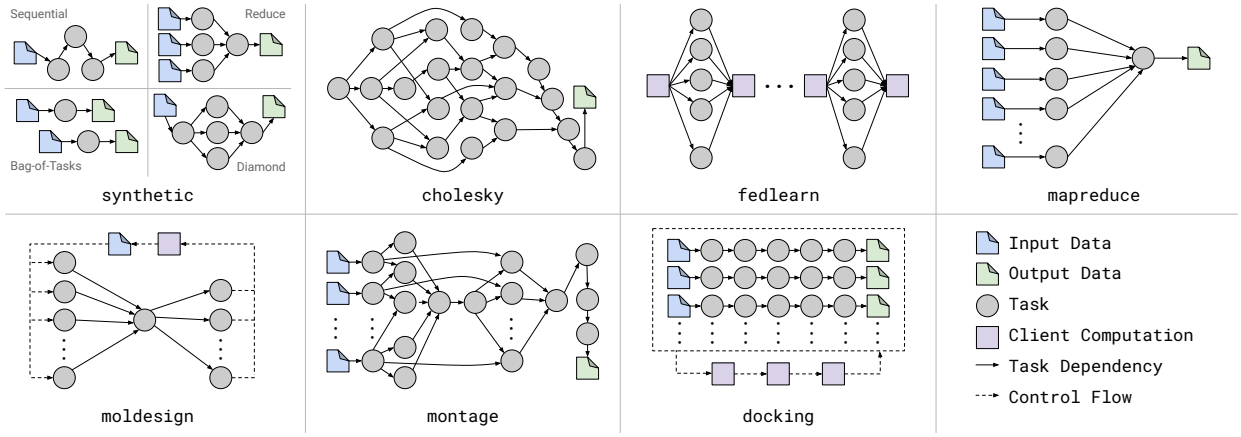

Fig. 2. Example task dependency diagrams for each application. In most applications, the exact structure depends on the application configuration.

coordinates to the PDB file using VMD [60], a program used to display and analyse molecular assemblies.

C. Federated Learning

Federated Learning (FL) is a paradigm for deep learning across decentralized devices with their own private data. FL offloads the task of model training to the decentralized devices to avoid communicating their raw training data across the network, providing some level of privacy and reducing data transfer costs. FL is organized into multiple rounds. In each round, a central server is responsible for collecting locally-updated model parameters from each device and aggregating the parameters to produce/update a global model. The new global model is then redistributed to the decentralized devices for further training and the loop repeats for future rounds [61].

We implement a simple FL application, `fedlearn`, that simulates a decentralized system with varying number of simulated devices and data distributions. `Fedlearn` follows the flow of execution described above and consists of three tasks: local training, model aggregation, and global model testing. The first task emulates the local training that is performed on a simulated remote device. The second task takes the returned locally-trained models for a given round as input to perform a model aggregation step to update the global model. The third task takes the recently-updated global model and evaluates it using a test dataset that was not used during training. All tasks are implemented as pure Python functions with model training and evaluation performed using PyTorch [62].

The application can be tuned in several ways, including, but not limited to, the total number of aggregation rounds, the number of simulated devices, the distribution of data samples across the simulated devices via the Dirichlet distribution, training hyperparameters (e.g., epochs, learning rate, mini-batch size), and fraction of devices randomly sampled to participate in each round. The application supports four standard deep learning datasets (MNIST [49], Fashion-MNIST [50], CIFAR-10, CIFAR-100 [51]), each of which is split into disjoint subsets across each simulated device for local training. A multi-layer perceptron network with three layers and ReLU activations is used with MNIST and Fashion-MNIST, and a small convolutional neural network with ReLU activations is used with CIFAR-10 and CIFAR-100.

D. MapReduce

MapReduce [52] is a programming model for parallel big data processing comprised of two tasks types. Map tasks filter or sort input data, and a reduce task performs a summation operation on the map outputs. The canonical example for MapReduce is computing words counts in a text corpus. Here, the map tasks take a subset of documents in the corpus as input and count each word in the subset. The subset counts are then summed by the reduce task.

The `mapreduce` application implements this word frequency example. The goal of this application is to evaluate system responsiveness when processing large datasets. The implementation can operate in two modes, one in which a text corpus of

arbitrary, user-defined size is generated, and another in which user-provided text files can be read. For a real dataset, we use the publicly available Enron email dataset [53]. Beyond specifying the input corpus or parameters of the randomly generated corpus, the number of mapping tasks and n, the number of most frequent words to save, are configurable.

The map task, implemented in Python, takes as input either a string of text or a list of files to read the text string from and returns a `collections.Counter` object containing the frequencies of each work. The reduce task takes a list of `Counter` objects and returns a single `Counter`. The application produces an output file containing the n most frequent words and their frequencies.

E. Molecular Design

Molecules with high ionization potentials (IP) are important for the design of next-generation redox-flow batteries [63, 64]. Active learning, a process where a surrogate ML model is used to determine which simulations to perform based on previous computations, is commonly employed to efficiently discover high-performing molecules.

The `moldesign` application is based on a Parsl implementation of ML-guided molecular design [54]. The application has three task types. Simulation tasks compute a molecule's IP, training tasks retrain an ML model based on the results of simulation tasks, and inference tasks use the ML model to predict which molecules will have high IPs and should be simulated. This application is highly dynamic and does not have strong inter-task dependencies—the client processes task results to determine which new tasks should be submitted. Molecules are sampled from the open-source QM9 dataset [55]. The number of initial simulations to perform, simulation batch size, and number of molecules to evaluate in total are configurable. These parameters control the maximum parallelism of the application and the length of the campaign.

F. Montage

Montage is a toolkit for creating mosaics from astronomical images [65]. The Montage Mosaic workflow streamlines the creation of such mosaics by invoking a series of Montage tools on the provided input data. This workflow was adapted from Montage's "Getting Started" tutorial [66].

The `montage` application is executed using a directory of input images and parameters for the table and header file names. The 2MASS input images are made available by Montage [67]. The application consists of a series of image processing tasks that will (1) reproject the images, (2) update metadata, (3) remove overlaps, and (4) combine images into a mosaic. Parallelism within the workflow occurs during the reprojection of the images, removing overlaps between two images, and removing the background in each input image. Tasks read and write intermediate files so all workers require access to a shared file system.

G. Failure Injection

The `failures` application can inject failures into another TAPS application. Injecting failures enables analyzing the failure recovery characteristics of executors. Task-level failure types include runtime exceptions (e.g, divide-by-zero, import error, out-of-memory, open file limit (ulimit) exceeded, and walltime exceeded) and dependency errors from a failed parent task. System-level failures include task worker, worker manager, and node failures. The failure type, failure rate, and base application to inject failures into are configurable.

H. Synthetic Workflow

The `synthetic` application is used to create synthetic computational workflows and is useful for stress testing systems. Tasks in this application are no-op sleep tasks which take in some random data and, optionally, produce some random data. One of four structures for the workflow DAG can be chosen: sequential, reduce, bag-of-tasks, and diamond, as described in Fig 2. The number of tasks, input and output data sizes, and sleep times are all configurable.

V. Evaluation

We showcase the kinds of performance evaluations possible with TAPS using the provided applications. We draw some general conclusions but do not make an exhaustive comparison between executors. Rather, we aim to demonstrate the varied performance characteristics of our supported applications and plugins, highlight the kinds of investigations or analyses that can be performed with TAPS, and pose interesting questions for future investigations. We use a `compute-zen-3` node, with two 64-core CPUs and 256 GB memory, on Chameleon Cloud's CHI@TACC cluster for evaluation [68].

A. Application Makespan

We first compare application makespan, which includes executor and worker initialization, application execution, and shutdown, across each task executor. The space of possible configurations for each application and executor is combinatorially explosive. Thus, we choose application parameters, where possible, which result in high numbers of short tasks to accentuate the effects of overheads in the respective executors. Parameters are summarized in Table III. We also prefer configurations which reduce run-to-run variances, except for `docking` which is inherently stochastic. For each executor, we use the respective equivalent of a default local/single-node deployment, but we note that it is reasonable to expect performance improvements by tuning each executor deployment to the specific application and hardware.

The results, presented in Fig 3, indicate that no executor is optimal and lead us to ask further questions. Why are the following 2–3$\times$ faster than the others: Ray in `cholesky`, Dask and Parsl in `moldesign`, and Dask in `montage`? How does performance correlate to average task duration or data flow volume? How do different executors deal with nested parallelism (i.e., tasks which invoke multi-threaded code)?

We observe that Dask performs the best in applications with small maximum object sizes, such as `docking`, `moldesign`, and `montage` where, as shown in Table III, the maximum serialized object sizes are less than $\sim$1 MB. However, Dask

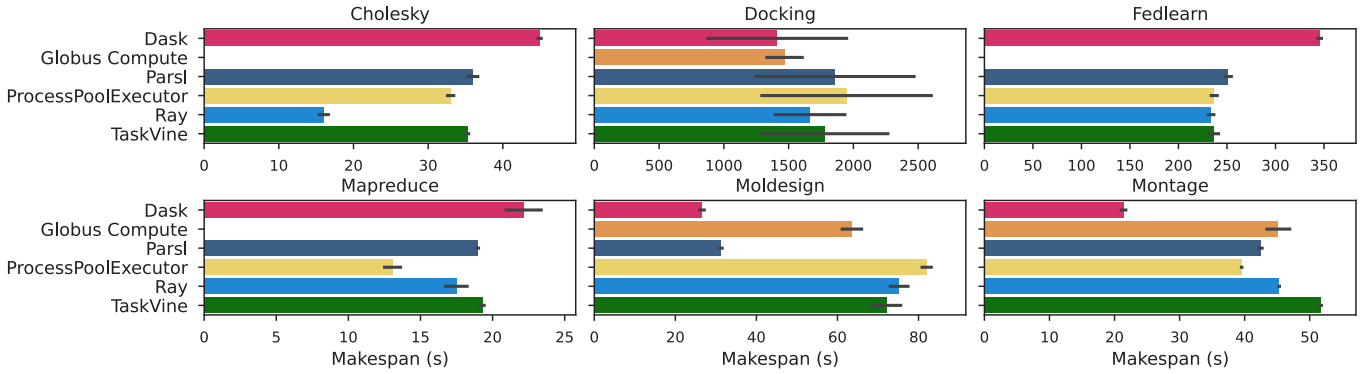

Fig. 3. Average application makespan over three runs. Error bars denote standard deviation.

TABLE III
SUMMARY OF APPLICATION CONFIGURATIONS USED IN FIG 3.

Application	Workers	Task Count	Max Serialized Object Size	Parameters
`cholesky`	64	385	24 MB	Matrix Size: 10 000×10 000, Block Size: 1000×1000
`docking`	32	192	$O(1)$ kB	Initial Simulations: 3, Batch Size: 8, Rounds: 3
`fedlearn`	32	48	20 MB	Dataset: MNIST, Clients: 16, Batch Size: 32, Rounds: 3, Epochs/Round: 1
`mapreduce`	32	33	114 MB	Dataset: Enron Email Corpus, Map Task Count: 32
`moldesign`	32	346	$O(1)$ MB	Initial Simulations: 16, Batch Size: 16, Search Count: 64
`montage`	32	419	$O(1)$ kB	—

is slow with applications that embed large objects in the task graph, such as the 114 MB mapper outputs in `mapreduce`. Ray marks input arrays as immutable enabling optimizations which yield considerable speedups in `cholesky`. Applications with nested parallelism (the simulation codes in `docking` and `moldesign` and tensor operations in `fedlearn`) lead to different outcomes. Globus Compute, Parsl, and `ProcessPoolExecutor` required setting `OMP_NUM_THREADS=1` to prevent resource contention leading to applications hanging, whereas Dask, Ray, and TaskVine worked immediately with all task types, albeit with varied performance. The Globus Compute service limits task payloads to 10 MB so the `cholesky`, `fedlearn`, and `mapreduce` applications are not natively supported and necessitate alternative data management systems (discussed further in Sec V-C).

B. Scaling Performance

We evaluate scaling performance of each executor using the `synthetic` app by executing 1000 no-op, no-data tasks and recording the task completion rate as a function of the number of workers on a single node. Here, the client submits n initial tasks where n is the number of workers and submits new tasks as running tasks complete. This configuration is intended to stress-test all aspects of the system including scheduler throughput, worker overheads, and client task result latency. We disable task result caching where applicable.

The results are presented in Fig 4. The `ProcessPoolExecutor` performs the best because, unlike the other executors, there is no scheduler. Thus, this serves as a good baseline for this single-node scaling setup; however, the lack of scheduler also means the `ProcessPoolExecutor` lacks

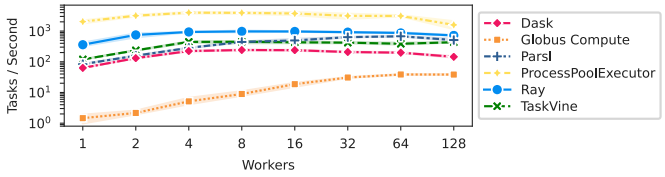

Fig. 4. Executor scaling performance with no-op tasks. Each configuration is repeated three times and shaded regions represent the standard deviation.

features useful for optimizing real applications such as multi-node support, data-aware task placement, and result caching. The general trend for Dask, Ray, and TaskVine is similar; task throughput increases up to four or eight workers and then degrades at high worker counts. However, Ray and TaskVine are both faster, with Ray being 5–10× faster than Dask. This can, in part, be attributed to Dask being pure Python while TaskVine's core is C and Ray's core is C++. Parsl, which is pure Python, exhibits superior scaling efficiency, closing the performance gap to Ray at larger scales. Globus Compute's task throughput is limited by its cloud service, but we do observe strong scaling performance with more workers as task requests and results can be more efficiently batched which amortizes cloud overheads.

C. Data Transfer

We examine the effects of data transfer on task latency and evaluate the `Transformer` plugins in Fig 5. We submit tasks to a pool of 32 workers and measure the average round-trip task time using the `synthetic` application. The client generates b bytes of random data as input to the task and

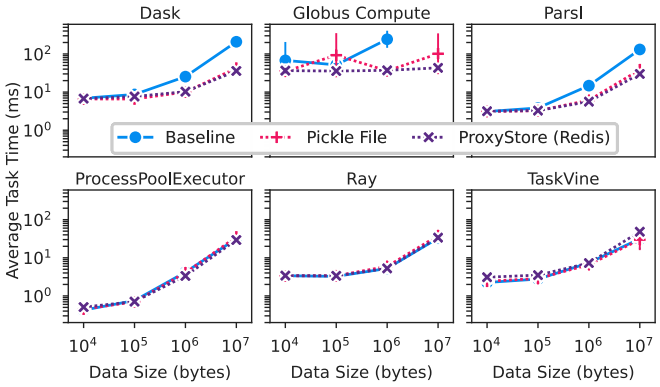

Fig. 5. Average round-trip time for no-op tasks as a function of input/output data size. Error bars denote standard deviation from three runs of 320 tasks (10×32 workers). The Globus Compute baseline is not evaluated at 10 MB due to task payload limits of the Globus Compute service.

the task returns b bytes of random data. We compare the baseline performance of the executors to using two different transformers: `PickleFileTransformer`, which writes pickled task data to the local NVMe drive, and ProxyStore, which we configured to use a Redis server to store intermediate data.

Dask and Parsl exhibit similar behaviour with task payloads greater than 100 kB inducing considerable increases in task latency. Using an alternate mechanism for data transfer alleviates much of this overhead, leading to 5.8× and 4.4× speedups for Dask and Parsl, respectively, at the largest data sizes. Globus Compute benefits the most from alternative data transfer mechanisms such as ProxyStore because the baseline method relies on data transfer to/from the cloud which is considerably slower. Use of ProxyStore also avoids Globus Compute's 10 MB task payload limit. The `ProcessPoolExecutor`, due to its simplicity, does not benefit much from either alternative transfer mechanisms. Ray and TaskVine perform well in all scenarios because Ray uses a distributed object store for large task data and TaskVine communicates intermediate data by files. Thus, these systems already employ techniques similar to the data transformers we evaluated.

VI. Conclusion

We have proposed TAPS, a performance evaluation platform for task-based execution frameworks. TAPS aims to provide a standard system for benchmarking frameworks. Benchmarking applications can be written in a framework agnostic manner then evaluated using TAPS' extensive plugin system. TAPS provides many reference applications, a diverse set of supported task executors and data management systems, and performance and metadata logging. We then showcased TAPS through a survey of evaluations to understand performance characteristics of the applications and executors, such as task overheads, data management, and scalability. Our hope is that TAPS will be a long-standing tool used to provide a common ground for evaluation and to facilitate the advancement in the state-of-the-art for parallel application execution.

Acknowledgments

We acknowledge the helpful support and guidance from the Cooperative Computing Lab at the University of Notre Dame with integrating TaskVine. This research was supported in part by Argonne National Laboratory under U.S. Department of Energy under Contract DE-AC02-06CH1135 and by the National Science Foundation under Grant 2004894 and 2209919 Results were obtained using the Chameleon testbed supported by the National Science Foundation.

References

[1] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *14th Python in Science Conference*, vol. 130, 2015, p. 136.

[2] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in Python," in *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2019. [Online]. Available: https://doi.org/10.1145/3307681.3325400

[3] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 561–577.

[4] J. J. Dongarra, "The LINPACK benchmark: An explanation," in *1st International Conference on Supercomputing*. Berlin, Heidelberg: Springer-Verlag, 1988, p. 456–474.

[5] "Transaction Processing Performance Council," https://www.tpc.org/. Accessed May 2024.

[6] "UnixBench," https://github.com/kdlucas/byte-unixbench. Accessed May 2024.

[7] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, A. Ike, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, T. Tabaru, C.-J. Wu, L. Xu, M. Yamazaki, C. Young, and M. Zaharia, "MLPerf training benchmark," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 336–349, 2020.

[8] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, mesh Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 2020, pp. 446–459.

[9] "Papers with Code," https://paperswithcode.com/datasets. Accessed May 2024.

[10] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[11] "Montage: An astronomical image mosaic engine," http://montage.ipac.caltech.edu/. Accessed Mar. 2024.

[12] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The Montage example," in *SC'08: ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[13] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. Ferreira da Silva, "WfCommons: A framework for enabling scientific workflow research and development," *Future Generation Computer Systems*, vol. 128, pp. 16–27, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21003897

[14] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcX: A Federated Function Serving Fabric for Science," in *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020. [Online]. Available: http://dx.doi.org/10.1145/3369583.3392683

[15] A. Alsaadi, L. Ward, A. Merzky, K. Chard, I. Foster, S. Jha, and M. Turilli, "Radical-Pilot and Parsl: Executing heterogeneous workflows on HPC platforms," *arXiv preprint arXiv:2105.13185*, 2021.

[16] "Apache Airflow," https://airflow.apache.org/. Accessed May 2024.

[17] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson, "FireWorks: A dynamic workflow system designed for high-throughput applications," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5037–5059, 2015.

[18] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2443416.2443417

[19] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature biotechnology*, vol. 35, no. 4, pp. 316–319, 2017.

[20] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X14002015

[21] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819111000524

[22] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 95–102.

[23] B. Sly-Delgado, T. S. Phung, C. Thomas, D. Simonetti, A. Hennessee, B. Tovar, and D. Thain, "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*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1978–1988. [Online]. Available: https://doi.org/10.1145/3624062.3624277

[24] "Pegasus examples," https://github.com/pegasus-isi/ACCESS-Pegasus-Examples/. Accessed May 2024.

[25] "Dask benchmarks," https://github.com/dask/dask-benchmarks. Accessed May 2024.

[26] C. Goble, S. Soiland-Reyes, F. Bacall, S. Owen, A. Williams, I. Eguinoa, B. Droesbeke, S. Leo, L. Pireddu, L. Rodríguez-Navas, J. M. Fernández, S. Capella-Gutierrez, H. Ménager, B. Grüning, B. Serrano-Solano, P. Ewels, and F. Coppens, "Implementing FAIR digital objects in the EOSC-life workflow collaboratory," 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4605654

[27] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling research in distributed scientific workflows," in *IEEE 10th International Conference on eScience, eScience 2014*, vol. 1, 10 2014.

[28] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, and F. Suter, "Developing accurate and scalable simulators of production workflow management systems with WRENCH," *Future Generation Computer Systems*, vol. 112, pp. 162–175, 2020.

[29] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: http://hal.inria.fr/hal-01017319

[30] D. S. Katz, A. Merzky, Z. Zhang, and S. Jha, "Application skeletons: Construction and use in escience," *Future Generation Computer Systems*, vol. 59, pp. 114–124, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X15003143

[31] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 502–504.

[32] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "FaaSdom: A benchmark suite for serverless computing," in *14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 73–84. [Online]. Available: https://doi.org/10.1145/3401025.3401738

[33] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 64–78.

[34] "Python Concurrent Execution," https://docs.python.org/3/library/concurrent.futures.html. Accessed May 2024.

[35] J. G. Pauloski, V. Hayot-Sasson, L. Ward, N. Hudson, C. Sabino, M. Baughman, K. Chard, and I. Foster, "Accelerating Communications in Federated Applications with Transparent Object Proxies," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, New York, NY, USA, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607047

[36] J. G. Pauloski, V. Hayot-Sasson, L. Ward, A. Brace, A. Bauer, K. Chard, and I. Foster, "Object Proxy Patterns for Accelerating Distributed Applications," 2024. [Online]. Available: https://arxiv.org/abs/2407.01764

[37] M. Hennecke, "DAOS: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, vol. 40, 2020.

[38] I. Foster, "Globus Online: Accelerating and democratizing science through cloud-based services," *IEEE Internet Computing*, vol. 15, no. 3, pp. 70–73, 2011.

[39] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Communications of the ACM*, vol. 55, no. 2, p. 81–88, feb 2012. [Online]. Available: https://doi.org/10.1145/2076450.2076468

[40] "Py-Margo," https://github.com/mochi-hpc/py-mochi-margo. Accessed Mar 2023.

[41] "Redis," 2023, https://redis.io/. Accessed Mar 2023.

[42] "UCX-Py," https://ucx-py.readthedocs.io/en/latest/. Accessed Mar 2023.

[43] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[44] E. Jeannot, "Performance analysis and optimization of the tiled Cholesky factorization on NUMA machines," in *5th International Symposium on Parallel Architectures, Algorithms and Programming*, 2012, pp. 210–217.

[45] J. Raicu, V. Hayot-Sasson, K. Chard, and I. Foster, "Navigating the molecular maze: A Python-powered approach to virtual drug screening," 2023, https://github.com/Parsl/parsl-docking-tutorial. Accessed May 2024.

[46] "AutoDock Vina: Python scripting," https://github.com/ccsb-scripps/AutoDock-Vina/tree/develop/example/python_scripting. Accessed May 2024.

[47] A. Clyde, X. Liu, T. Brettin, H. Yoo, A. Partin, Y. Babuji, B. Blaiszik, J. Mohd-Yusof, A. Merzky, M. Turilli *et al.*, "AI-accelerated protein-ligand docking for SARS-CoV-2 is 100-fold faster with no significant change in detection," *Scientific Reports*, vol. 13, no. 1, p. 2105, 2023.

[48] N. Kotsehub, M. Baughman, R. Chard, N. Hudson, P. Patros, O. Rana, I. Foster, and K. Chard, "FLoX: Federated learning with FaaS at the edge," in *IEEE 18th International Conference on e-Science*. IEEE, 2022, pp. 11–20.

[49] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[50] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," *arXiv:1708.07747*, 2017.

[51] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[52] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004. [Online]. Available: https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters

[53] "Enron Email Corpus," https://www.cs.cmu.edu/~enron/. Accessed May 2024.

[54] "Molecular design in Parsl," https://github.com/ExaWorks/molecular-design-parsl-demo. Accessed May 2024.

[55] R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld, "Quantum chemistry structures and properties of 134 kilo molecules," *Scientific Data*, vol. 1, 2014.

[56] E. Jeannot, "Performance analysis and optimization of the tiled Cholesky factorization on NUMA machines," in *5th International Symposium on Parallel Architectures, Algorithms and Programming*, 2012, pp. 210–217.

[57] O. Beaumont, L. Eyraud-Dubois, M. Vérité, and J. Langou, "I/O-Optimal Algorithms for Symmetric Linear Algebra Kernels," in *ACM Symposium on Parallelism in Algorithms and Architectures*, 2022. [Online]. Available: https://hal.inria.fr/hal-03580531

[58] O. Trott and A. J. Olson, "AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *Journal of Computational Chemistry*, vol. 31, no. 2, pp. 455–461, 2010.

[59] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[60] W. Humphrey, A. Dalke, and K. Schulten, "VMD: Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, no. 1, pp. 33–38, 1996.

[61] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.

[62] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[63] L. Ward, G. Sivaraman, J. G. Pauloski, Y. Babuji, R. Chard, N. Dandu, P. C. Redfern, R. S. Assary, K. Chard, L. A. Curtiss, R. Thakur, and I. Foster, "Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing," in *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments*. IEEE, 2021. [Online]. Available: http://dx.doi.org/10.1109/mlhpc54614.2021.00007

[64] L. Ward, J. G. Pauloski, V. Hayot-Sasson, R. Chard, Y. Babuji, G. Sivaraman, S. Choudhury, K. Chard, R. Thakur, and I. Foster, "Cloud services enable efficient AI-guided simulation workflows across heterogeneous resources," in *Heterogeneity in Computing Workshop*. IEEE Computer Society, 2023, https://arxiv.org/abs/2303.08803.

[65] G. Berriman, J. Good, D. Curkendall, J. Jacob, D. Katz, T. Prince, and R. Williams, "An on-demand image mosaic service for the NVO," in *Astronomical Data Analysis Software and Systems XII*, vol. 295, 2003, p. 343.

[66] "Getting Started: Creating Your First Montage Mosaic," http://montage.ipac.caltech.edu/docs/first_mosaic_tutorial.html. Accessed May 2024.

[67] "2MASS Image Dataset," http://montage.ipac.caltech.edu/docs/Kimages.tar. Accessed May 2024.

[68] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the Chameleon testbed," in *USENIX Annual Technical Conference*. USENIX Association, July 2020.