Accelerating Function-Centric Applications by Discovering, Distributing, and Retaining Reusable Context in Workflow Systems

Thanh Son Phung
University of Notre Dame
Notre Dame, IN, USA
tphung@nd.edu

Colin Thomas
University of Notre Dame
Notre Dame, IN, USA
cthoma26@nd.edu

Logan Ward
Argonne National Laboratory
Lemont, IL, USA
lward@anl.gov

Kyle Chard
University of Chicago
Chicago, IL, USA
chard@uchicago.edu

Douglas Thain
University of Notre Dame
Notre Dame, IN, USA
dthain@nd.edu

ABSTRACT

Workflow systems provide a convenient way for users to write large-scale applications by composing independent tasks into large graphs that can be executed concurrently on high-performance clusters. In many newer workflow systems, tasks are often expressed as a combination of function invocations in a high-level language. Because necessary code and data are not statically known prior to execution, they must be moved into the cluster at runtime. An obvious way of doing this is to translate function invocations into self-contained executable programs and run them as usual, but this brings a hefty performance penalty: a function invocation now needs to piggyback its context with extra code and data to a remote node, and the remote node needs to take extra time to reconstruct the invocation's context before executing it, both detrimental to lightweight short-running functions.

A better solution for workflow systems is to treat functions and invocations as first-class abstractions: subsequent invocations of the same function on a worker node should only pay for the cost of context setup once and reuse the context between different invocations. The remaining problems lie in discovering, distributing, and retaining the reusable context among workers. In this paper, we discuss the rationale and design requirement of these mechanisms to support context reuse, and implement them in TaskVine, a data-intensive distributed framework and execution engine. Our results from executing a large-scale neural network inference application and a molecular design application show that treating functions and invocations as first-class abstractions reduces the execution time of the applications by 94.5% and 26.9%, respectively.

CCS CONCEPTS

• **Computer systems organization** → Cloud computing.

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06…$15.00
https://doi.org/10.1145/3625549.3658663

KEYWORDS

Workflow Systems; Serverless Computing; Distributed Storage; Burst Buffers

ACM Reference Format:
Thanh Son Phung, Colin Thomas, Logan Ward, Kyle Chard, and Douglas Thain. 2024. Accelerating Function-Centric Applications by Discovering, Distributing, and Retaining Reusable Context in Workflow Systems. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658663

1 INTRODUCTION

Modern scientific research applications demand large amounts of compute and storage power, reaching tens of thousands of CPUs/GPUs and petabytes of data. This demand transcends any specific scientific field, for example the GAMESS application in the Exascale Computing Project reports using 95% of the Summit supercomputer for computations involving 20k atoms [26], the Dark Energy Science Collaboration (DESC) scans and processes an outer space region of $(4.225Gpc)^3$ volume, producing 1PB of intermediate and final data products [4], the Large Hadron Collider generates petabytes of data per year through four experimental detectors [22], and Meta trains its largest LLM (LLaMA 65B) on 2,048 GPUs in 21 days [17].

To match this enormous demand for data and computation, many applications break their complex computations into a directed acyclic graph (DAG) of independent tasks and linearly scale the rate of computation with the number of compute nodes. Workflow execution engines [1, 30, 38] enable such a distributed programming model, allowing users to express computational needs as a DAG of tasks and automating node acquisition and release via batch systems, task scheduling, result retrievals, etc.

However, expressing a DAG of computations is a non-trivial task even for technical users, as it usually requires them to understand the computations and their associated I/O behaviors, learn to effectively use the workflow execution engine and its capabilities, and write complex customized scripts describing computations wrapped as tasks and their associated data inputs and outputs. Therefore, a number of parallel programming libraries (e.g., Parsl [6], Dask [23], RADICAL [32]) have been developed with the aim to automatically

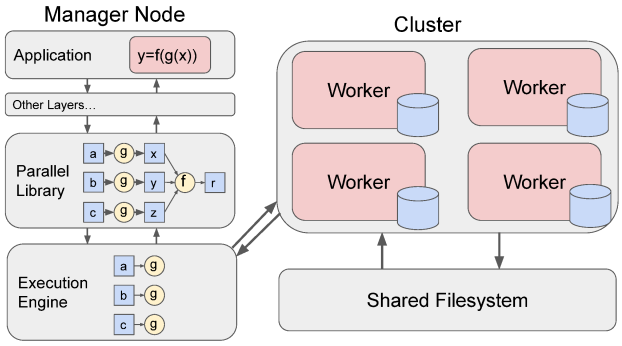


Figure 1: Software Stack for Task-based Applications
The parallel library receives functions as computations from the application and builds a DAG of tasks. The execution engine receives ready tasks from the library and sends them to workers in the cluster for execution. Once done, results are transparently sent from workers back to the application.

	State	Worker Requirement	Execution Requirement
Task	Stateless	None	Code+Data+Args
Invocation	Stateful	Code+Data	Args

Table 1: Differences between a Task and an Invocation

construct the DAG of tasks and feed it to an underlying execution engine, allowing users the ability to conveniently write and invoke computations as regular functions in popular programming languages (e.g, Python, Perl, etc.).

Figure 1 shows an overview of the workflow software stack. The control plane often includes the application and the workflow system, which consists of a parallel library and an execution engine, all deployed on the same manager node, whereas the data plane usually consists of a common data storage (e.g., shared filesystem, AWS S3) and many small local storage resources (e.g., burst buffers) from compute nodes in a cluster. At the top, an application uses the parallel library's APIs to invoke functions. The parallel library then automatically creates and maintains a DAG of function invocations, transforms invocations into tasks, and sends ready tasks to the execution engine. The execution engine spawns worker processes on compute nodes in the cluster, schedules tasks to workers for execution, and sends results from completed tasks back to the parallel library and the application. Data management policy can vary greatly between specific systems, ranging from relying exclusively on a common data storage for data inputs and outputs to self-staging data inputs and outputs to and from compute nodes.

A naive transformation of function invocations into tasks in the parallel library layer in Figure 1 can incur a large overhead penalty however. For example, the most straightforward way to transform a function invocation into a task is to serialize the function along with its arguments to a file and send it as an input to a wrapper task. This wrapper task's job is to deserialize the file to retrieve the function and arguments, execute it, and return the result. While this approach effectively turns a function invocation into a standard task and thus enjoys all benefits of optimized task scheduling and execution from the execution engine layer, it neglects key differences between a task and an invocation.

	Total Time (s)	Overhead per Worker (s)	Overhead per Invocation (s)
Local Invocation	$8.89 * 10^{-5}$	0	0
Remote Task	211.06	20.65	0.19
Remote Invocation	22.46	19.94	$2.52 * 10^{-3}$

Table 2: Overhead of Executing 1,000 Python Functions

Table 1 summarizes such differences between a task and an invocation. A task is usually a stateless independent unit of execution that can run on any worker. It thus doesn't require the worker to have any dependencies pre-installed, and instead brings its code, data, and input arguments (or metadata of them) along with it upon execution. On the other hand, a function invocation needs its **context** (e.g., code and data) to execute: it requires a worker that already hosts its context, and then only needs to bring along the input arguments for remote execution. With careful design, subsequent invocations of the same function can efficiently reuse the same context. The naive mapping in the above example however forces a function invocation to reload its context every time it's executed, and thus (1) prevents the opportunity of context reuse between invocations, and (2) is detrimental to short-running invocations (in the order of seconds or minutes) as the context setup and reload can easily dominate the execution time.

Table 2 shows a detailed overhead breakdown when executing 1,000 simple Python functions, each of which performs an addition and then returns, with 3 execution modes: (1) Local Invocation, which runs functions natively in the Python Interpreter, (2) Task, which packages a function as a task runnable on any worker (1 worker in this example) and therefore must load its context for every execution, and (3) Remote Invocation, which retains and reuses the same function context on a worker between invocations. Execution via Task and Remote Invocation is done using TaskVine[28], a data-intensive distributed framework and execution engine. While Local Invocation runs the fastest as functions are executed using a local Python Interpreter process, it can't scale to more than 1 node. More importantly, we see the efficiency of retaining and reusing function contexts between invocations, as while Task and Remote Invocation both spend roughly 20 seconds to set up the worker and its data assets, Remote Invocation only has 2.52 ms overhead per subsequent invocation in contrast to 0.19s of Task.

Enabling such efficiency from context reuse is not a problem with a straightforward solution however, as discussed above. It requires workflow systems to treat functions and invocations as first-class abstractions. For a given function, its context C needs to be **discovered** by a workflow system, either via analysis of the function object and/or manual specification from users. C can include anything related to the function's code and external data (software dependencies, input data, function's language-specific code, etc.) and materialize in any format (on disk, in memory, in GPUs, etc.) on a worker node. C then needs to be packaged in a portable format and efficiently **distributed** among all workers in the system to shorten the time needed to prepare workers for the incoming invocations. Finally, C needs to be **retained** on a given worker using local resources as long as necessary, such that

invocations of the same function can efficiently reuse C to avoid paying the cost of context reload.

Our contributions in this paper are as follows:

(1) We identify a performance problem within the workflow system software stack. We then set the goal of removing this problem by treating functions and invocations as first-class abstractions and devising new mechanisms to support them.
(2) We enumerate three mechanisms to support acceleration of function-centric applications via reuse of function contexts: discover, distribute, and retain. An analysis of the rationale and design requirement for each mechanism is provided.
(3) We implement these mechanisms in TaskVine, a data-intensive distributed framework and execution engine, and describe in detail our implementation.
(4) We evaluate our work using two applications: a large-scale neural network inference (LNNI) using the ResNet50 [13] model and ExaMol [35], a real-world scientific application that combines molecular simulations with machine learning training and inference to design new molecules. LNNI contains 100k tasks running in total 1.6 million inferences. ExaMol contains 10k tasks. We show that our work reduces the execution time of LNNI by 94.5% (7,485 to 414 seconds) and ExaMol by 26.9% (4,600 to 3,364 seconds).

2 FUNCTION CONTEXT

A key characteristic of function-centric applications is that necessary computations are expressed in the form of functions in a given programming language. We first characterize several important subjects (application, function, function context, invocation), and then describe the three capabilities (discover, distribute, retain) for workflow systems to support efficient context reuse in large-scale function-centric applications.

2.1 Overview

2.1.1 Application. An application is the driving process for a large-scale computational need. This process usually resembles a feedback loop: the application requests some initial computations, and depending on the results of them, the application may decide that the computation need is satisfied and terminate, or it may request more computations and continue to the next phase. Extreme forms of this process are also possible and quite common. On one end, an application deploys all needed computations in one phase, and terminates once results are returned (e.g., a full non-overlapping sweep of a dataset). On the other end, the application acts like a service that waits for a certain event to occur and deploys computations on-the-fly. The application may run for an indefinite amount of time. A function-centric application falls within this characterization, and special care is needed to address challenges from both ends of the spectrum.

A function-centric application is usually designed to tackle large-scale computational needs via the divide-and-conquer approach: a big computational problem is broken down into many smaller independent subproblems. The blueprint to each subproblem is wrapped into a function in the top layer of the manager node in Figure 1, with the specification of individual subproblems passed to the function as arguments.

An application starts computations by invoking functions with some arguments in a given programming language. Such invocation is usually asynchronous and mediated by underlying frameworks (e.g., workflow systems) for remote execution. For each computation request received, the framework returns to the application a promise that the application will know and receive the result when a function is successfully executed. The application is otherwise oblivious to any other problems: it does not know how functions are transferred over to remote workers, what dependencies are required, which part of the functions can be effectively reused between invocations of the same function, etc. It's the framework's job to make remote execution as close to a local execution as possible.

2.1.2 Function. A function is an independent unit of computation that has inputs and outputs. It usually is not self-contained however, as inputs can come from several sources: language-specific code objects via the function's arguments, access to external data storage that's buried deep inside a function's code, implicit references to package dependencies, etc. The same applies to outputs, where a function can return results via its return value but also by writing results to an external data storage and exiting with a success code.

A function's code can be separated into two parts: one that sets up a reusable context and one that invokes computations with the given arguments. The former usually consists of setting up the context of the function, such as importing relevant modules (this may imply compiling language-specific code on-the-fly), opening files to read in necessary states of relevant objects (e.g., load parameters of a model into memory), or preparing the execution environment with local resources for the invocation (e.g., move a given model from memory into a GPU). The latter concerns more about the execution of a function and is usually dictated by the arguments given to an invocation. While the function context sets up a shareable and reusable environment to a function, the invocation executes the computation and may mutate the environment on the go, with each invocation likely very different than others to reflect different subproblems. Therefore, it is important to separate out the reusable context of a function such that subsequent invocations can reuse the context efficiently.

2.1.3 Function Context. As discussed above, a function context takes care of setting up the environment for an incoming invocation. Since this environment setup happens on a fresh worker with no dependencies pre-installed as the application runs, this setup process can be arbitrarily complex, from fetching datasets from external sources to the worker's local disk, to caching objects in memory for subsequent invocations or loading models into accelerators. Therefore, the setup of function context is best represented by an executable object that allows arbitrary code execution, e.g., an auxiliary function. This function's job is not to directly execute the necessary computations, but to allow users to specify all steps to prepare the environment in a programmatic way. Most importantly, once the setup is complete, the executable object must terminate without relinquishing all the setup work done.

Since the context setup can execute arbitrary code and leaves behind an environment when terminated, this environment will occupy arbitrary local resources on the worker. This implies that a worker must be able to account for such resource occupation and

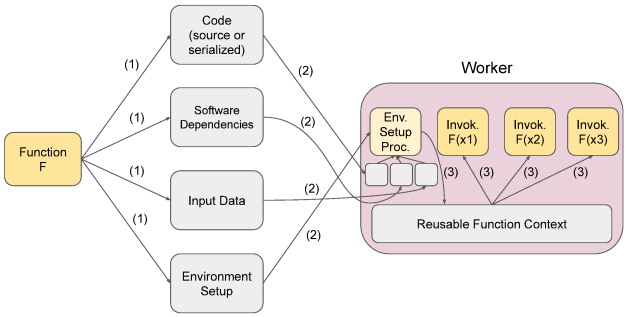


Figure 2: Discovering, Distributing, and Retaining Function Context for Remote Invocation on Workers
(1) Function context is discovered by analyzing and detecting code, software dependencies, input data, and environment setup code. (2) These elements are packaged and sent to a worker. (3) An environment setup process prepares a reusable function context, and subsequent invocations use one copy of context instead of reloading each time.

monitor its consumption, especially in cacheable resources (memory, disk, GPUs), and report such consumption back to the manager for scheduling decisions. The resource accounting for functions is therefore more complex than tasks, as tasks are designed to be independent of each other and thus have independent resource allocations[21], while functions are designed to maximize sharing and it can be difficult to precisely keep track of such sharing in the worker's local system (e.g., memory pages shared between concurrent invocations).

An avid reader may notice that the context setup concept is very similar to the concept of code hoisting in compiler literature in the sense of hoisting the expensive but deterministic operations out of a given loop to prevent duplicate computations. The reader would be right: this paper aims to hoist out the context setup part of a function such that subsequent invocations can benefit from the price of setting up the environment once. Complexities arise when this concept is brought to the distributed systems' world: what mechanisms are needed to efficiently support this function context concept when faced with tens of thousands of invocations and hundreds of machines. Note that this paper doesn't aim at automating context hoisting as in the compiler literature: it provides the supporting mechanisms for users or other systems to do so. Thus, while automatic context detection is a promising idea, it is out of scope of this paper.

2.1.4 Invocation. An invocation is the execution part of a function and mainly characterized by the arguments given to it. Invocations are independent and typically different from one another to reflect different computations. An invocation is dependent on the existence of a function context: it executes based on the environment the context has set up, and may mutate the environment during its execution. A large portion of the environment is usually left untouched or read-only from the invocation however, such as the function's software dependencies or input datasets. A reusable function context is this portion of the environment: invocations of the same function can freely mutate the environment in its memory space, but the part of the environment left untouched can be efficiently shared between invocations. Additionally, an invocation is much more lightweight than a task, as shown in Table 1 - only the arguments along with some metadata of an invocation are needed for a worker with proper context setup to execute it.

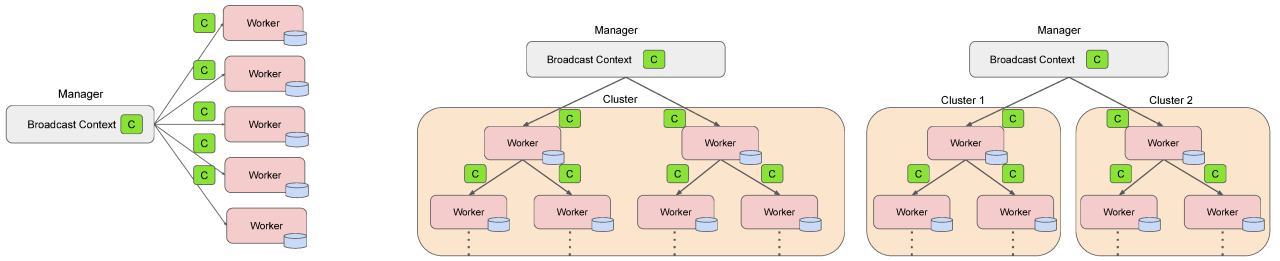
2.2 Mechanisms

2.2.1 Discover. To enable remote execution of invocations, workflow systems first need a mechanism that discovers the context of the invocations' function — all necessary dependencies and arbitrary environment setup — to send the context to a given worker. The context includes four distinct elements: the function code itself, the code's dependencies, input data, and arbitrary environment setup. Point 1 of Figure 2 summarizes these elements of function context discovery.

Function code. A worker first needs the function code to execute invocations. This code by itself can have two formats: code as raw strings of bytes (source code) and code as a language-specific object. Source code is fairly straightforward to discover: the workflow system can simply inspect the file containing the source code, identify and copy all lines of the relevant function, and add it to the context object of a given function. In many cases, such source code is not easy or even possible to obtain. The function given to a workflow system may have passed through many layers of software and the workflow system thus can't easily locate the source code. Additionally, the function itself may not even have a source form, such as functions that result from dynamic execution of a given string in a programming language or lambda functions. These cases require the workflow system to serialize the function's code object into a binary string or a file, which is usually achieved by walking the abstract syntax tree (AST) of the function code. Once sent over the network to a remote worker, the worker then deserializes the given string or file and attempts to reconstruct the object on its side. Thus, the serialized format of such functions is the important bit that is added to the context object in these cases.

Software Dependencies. Many approaches can encapsulate the software dependencies of a given function. A user might directly provide a specification of all software dependencies of the function, with or without versions specified, to the workflow system along with the function invocation[24]. The workflow system's job then is to send the invocation to a remote worker, install all dependencies, and execute it. Another approach is for users to provide a container image that already contains all software dependencies, and the workflow system only has to ship the image to a worker and execute the invocation within this container. Finally, if the user doesn't provide any hints, a workflow system can walk the function code's AST and try to infer necessary dependencies from the code. Once identified, the workflow system can either choose to install all dependencies into a container and then send it to workers, or to let workers install dependencies themselves.

Input data. Generally speaking, input data covers anything an invocation might use in the beginning of its execution. Therefore, the function code and software dependencies once discovered should also be treated as a source of input. Since invocations mainly differ in their arguments, special care is needed to prevent shareable data to be unnecessarily sent along with every invocation. This



(a) No communication between workers (b) Full communication between workers (c) Limited comm. between sets of workers

Figure 3: Solutions to Distributing Function Contexts in Workflow Systems

```
def context_setup(args):
    # load model parameters from disk to memory
    # move model from memory to GPU
    # register model variable to global namespace

def infer(img, context=context_setup):
    global model
    model.infer(img)
```

Figure 4: Example: Separating Context Setup from Computation in a ML Inference Function

can be achieved by having explicit data-to-invocation and data-to-worker bindings, such that each invocation is bound to several data inputs before dispatched to a remote worker, and each data input is registered to the worker once fetched and locally cached. In this way, multiple invocations concurrently executed on a worker can share just 1 copy of the set of input data, and an incoming invocation only needs to send its arguments to the worker for execution.

Arbitrary environment setup. This is the final step to fully create an environment for the incoming invocations on a remote worker. Since it is an executable object with arbitrary execution, users can define the setup computation in a script to be executed as a process, a function to be executed as a thread, etc. This setup also should be run only when needed to avoid repeatedly setting up the same environment and to allow for a maximal degree of sharing between invocations of the same function. Thus, there needs to be an invocation-to-context binding for every invocation such that workflow systems can utilize this binding in invocation scheduling and deduplicating contexts in a remote worker.

2.2.2 Distribute. Once the function context is discovered, workflow systems need to broadcast the context to all connected workers as fast as possible to reduce the wait time of tasks on the manager node[20]. While the problem of maximum broadcasting efficiency has been extensively studied in the computer network literature and related fields, the solution to this problem solely depends on one condition: the viability of direct data transfers between workers. Figure 3 shows three possible solutions depending on this condition.

On one end, workers cannot communicate with each other at all and rely only on the manager for all data transfers. This mode of communication can happen in clusters that have strict networking policy between machines and thus no internal communication is allowed. In this scenario, the broadcasting problem has only one solution: the manager must sequentially send data to workers.

On the other end, workers have no obstructions in communicating and transferring data between one another. This scenario is more common in clusters that have relaxed internal networking policy and/or are built to support large MPI jobs. In this scenario, the broadcasting problem can utilize the bandwidth between machines to maximize the data transfer throughput and avoid bottlenecking the manager's bandwidth. Most solutions then revolve around sending data like a spanning tree of workers. Note that any transferable data in the system has to be uniquely identified and read-only, otherwise data corruption can silently happen. This can be implemented in a variety of ways, such as maintaining a table of files in the manager, naming files based on the hash of their contents, etc. This approach is most suitable to send function contexts within a cluster as function contexts are shareable and read-only.

In the middle, workers can communicate with each other but the bandwidth is limited between certain sets of workers. This situation may arise when a set of workers are launched from a local cluster and another set are launched via commercial clusters (e.g., Google Cloud, AWS, etc.) This strategy has the advantage of using cheap cloud resources as needed, but requires a workflow system to be aware of which cluster a worker belongs to. Once this information is figured out, a manager simply transfers data sequentially between clusters and instructs workers within the same cluster to broadcast data like a spanning tree.

2.2.3 Retain. Finally, workflow systems need a mechanism that retains the function context on a worker for reuse between invocations. As discussed in subsection 2.1.3, workflow systems can support discovering and moving necessary elements of the context to a worker, but it is the job for the environment setup function to properly load required dependencies and datasets from disk into the worker's memory, accelerators, or other local resources. Figure 4 shows an example of an environment setup function that prepares the worker for incoming inference invocations on a ML model.

Once the reusable context is set up, subsequent invocations can inherit and execute in this environment in many ways. The reusable context can continue to live as a daemon process or a thread that runs in the background with the worker process and waits for invocations as events. Upon receiving invocations, it can execute invocations directly within its memory space. Additionally, if permitted by the application, the context process can also fork to execute so many invocations can concurrently benefit from a shared environment. A worker's job then is simply to accept the invocation request from the manager, send the request to the context process,

wait for the notification of a successful execution, and return results back to the manager.

3 IMPLEMENTATION

Section 2 discusses many requirements, tradeoffs, and possible implementation routes for all three mechanisms to support context reuse in large-scale function-centric applications. We now describe in detail our implementation.

3.1 Overview

We target applications written in Python as it has increasingly become a programming language of choice for large-scale scientific applications. Much of our implementation is done with TaskVine, a data-intensive framework for large-scale applications. TaskVine's components include a Python frontend where users can specify Python functions for remote execution, and a C backend that handles task execution, result retrieval, worker acquisition and release, fault tolerance, etc. For the parallel library layer, we implement the function context API in the TaskVine frontend. To increase the potential impact of our work to other established applications with their own software stack, we also integrate TaskVine as a backend execution engine to Parsl [6], resulting in the Parsl *TaskVineExecutor*. For the execution engine layer, we implement all the mechanisms in the TaskVine backend with changes to many of its components: task scheduling, task representation and deployment, worker's local management of resources, etc. We first describe our implementation for each mechanism, then show how we enhance TaskVine and design *TaskVineExecutor* to incorporate these mechanisms and support the new function and invocation abstractions with context reuse.

3.2 Discover

As discussed in subsection 2.2.1, the mechanism to discover function contexts requires the analysis and detection of four elements: function code, software dependencies, input data, and arbitrary environment setup.

Function code. Our implementation supports both approaches in detecting the function's code. We add to TaskVine an API that allows a user to specify a list of function objects or names to be included in the function context. Upon receiving the list, TaskVine first tries to extract the source code of such functions using the built-in `inspect` module in the Python standard library. If successful, TaskVine adds the source code of the functions to the context so that the worker can simply invoke these functions by their names. Otherwise, TaskVine serializes the functions to files using `cloudpickle` [8] and adds those files as inputs to the relevant context. The functions will later be reconstructed on a worker and ready for invocations.

Software dependencies. We modify TaskVine and the Poncho [27] toolkit to detect, install, and package all necessary dependencies of a function. After extracting the functions' code, TaskVine gives them to Poncho to scan their ASTs for imported modules, create a local Conda [14] environment containing these modules with versions resolved, and package the environment into a specially formatted tarball using the `conda-pack` [15] tool. Once the environment tarball is created, TaskVine then binds the tarball to the functions' context as an input file. TaskVine extensively uses this binding to send only the environment tarball when needed: a context process on a worker will reuse a copy of the tarball to execute an invocation if it is available in the worker's cache, and otherwise will request a transfer of tarball from either the manager or another worker which has it.

Input data. Any input data that is shareable between invocations can be declared to TaskVine as an input file to the relevant function's context and mark itself as cacheable and transferable. Once all shareable input data are bound to a given context, we create a special script that will act like the daemon process on a worker node as mentioned in subsection 2.2.3. We then create a special task from this script (this task does no actual work and cooperates with the worker process to invoke functions instead, more in subsection 3.4), bind all input data to this task, and use TaskVine's regular task scheduler and data-intensive capabilities (e.g., unique data naming, data caching on worker nodes, data transferring between nodes) to send the environment setup process and stage input data to the remote node as necessary. Once the special task is ready, subsequent invocations use the same copy of the shareable input data as described above.

Environment Setup. Each function F submitted to TaskVine can specify another helper function H to setup its context. F itself can just contain the invocation part, and defer the setup to H. We modify TaskVine such that upon receiving F, TaskVine registers H to the context of F. A worker is then instructed to execute H as part of its work setting up F's context. Each invocation of F then only executes its distinctive computation with the guarantee that all the setup work it needs from H is present and ready to be used on a worker node.

3.3 Distribute

TaskVine supports distribution modes 3a and 3b in Figure 3. When an application starts up and the first workers join the system, TaskVine sequentially sends input files from the manager node to these workers. If the worker-to-worker transfer feature is available and enabled, once a worker reports back with a success-transfer notification, the manager then directs that worker to start sending relevant input files to other workers instead. Each worker is capped to N transfers of input files at any given time to avoid a sink in the spanning tree of data transfers between workers. To quickly distribute function contexts to connected workers, we package a given function context into a set of files as described in subsection 3.2 and use TaskVine's built-in data distribution capability.

3.4 Retain

Once a function context is discovered by the manager (see subsection 3.2), TaskVine creates a special task called a "library" that runs like a daemon until terminated and cooperates with the worker process to execute invocations. The library is a Python script and acts like a service as described in subsection 2.2.3: it waits for instructions from the worker to execute invocations, and once done lets the worker know that results are available to retrieve. TaskVine then sends the library with the function context to a given worker. The protocol between a library and a worker is as follows:

```
1  import ndcctools.taskvine as vine
2  def context_setup(y):
3      ...
4  def f(x):
5      ...
6  manager = vine.Manager(...)
7  library = manager.create_library_from_functions('lib', f,
8              context=context_setup, context_args=[y])
9  dataset_file = vine.File('dataset.tar.gz', cache=True,
10             peer_transfer=True)
11 library.add_input(dataset_file)
12 manager.install_library(library)
13 ...
14 for i in range(10):
15     invocation = vine.FunctionCall('lib', 'f', args=[i])
16     manager.submit(invocation)
17 ...
```

Figure 5: Support of Function and Invocation Abstractions via a Code Sample in TaskVine

A user creates a library for function f, adds a common input data to the library, and registers the library to the manager. A function is pinpointed by its library's name and its name, and subsequent invocations only need to specify their arguments.

(1) The worker executes the library as a normal task by forking then exec'ing the Python script.
(2) The library starts up, reads in its configurations, executes all context setup functions provided to it, sends back a notification to the worker to let it know that it's ready to execute invocations, and waits for a message from the worker.
(3) The manager sends an invocation to the worker, and the worker waits for the ready message from the library. Upon receiving the ready message, the worker sets up a sandbox specifically for the invocation, and sends the invocation metadata, its arguments, and the sandbox to the library.
(4) Each invocation has two execution options: direct or fork. If the option is direct, then the library changes the working directory to the invocation's sandbox, then loads the arguments of the invocation into memory and executes it synchronously. Once the invocation is complete and the control is returned to the library, it serializes the result in to a result file in the invocation's sandbox, changes its working directory back, and lets the worker know. If the option is fork, then the library instead forks itself and waits for the child process to be done via the SIGCHLD signal or a message from the worker. The child process changes the working directory into the sandbox, loads the arguments, executes the invocation, dumps the result to the result file, and exits. The library upon receiving the SIGCHLD signal lets the worker know that it has a result to be fetched. In either case, the worker sends back the result file to the manager and destroys the invocation's sandbox. Both the library and the worker go back to waiting for invocations as in step (3).

3.5 TaskVine Enhancement

With all three mechanisms' implementation described, we now present the function context API via a code sample and the changes to the internal structures of TaskVine.

3.5.1 Function Context API. Figure 5 shows an example of how TaskVine supports function and invocation abstractions. Assume a user has broken the computation into f and *context_setup*, one first creates a TaskVine Manager object in line 6. Lines 7-8 tell the manager to create a library for a given function and its context, and the created library will automatically discover 3 out of 4 elements in subsection 3.2: function code, software dependencies, and environment setup. If there's a common input data between invocations, then the user can declare it to the manager and add it as an input to the library (lines 9-11). Once the context discovery via the f's library is done, it is registered to the manager in line 12. To invoke a function from a known library, the user simply needs to specify the relevant library and function's names with the invocation's distinct arguments (line 15).

3.5.2 Internals. We now describe actions a TaskVine manager takes once an invocation is submitted in a newly created workflow (line 16 in Figure 5.) The manager first pinpoints the invocation's library via the library and the function's names. A library by default takes all resources of a worker, but it can be configured to run on a portion of a worker as well. A library by default runs 1 invocation of a function at a given time, but this can also be changed by setting its number of invocation slots. Once the runtime details of a library are determined, the manager sequentially checks a hash ring of connected workers to see if any is available to run the library. The first available worker is sent an instance of the library to be executed as described in subsection 3.4 along with its pre-specified input data. The manager then holds on to that worker and sends as many invocations as available slots the library currently has. If the library is full, the manager can either send another library to the worker provided that the worker still has adequate amount of resources, or it moves on to the next worker in the ring.

Resource allocation for libraries and invocations is tricky as discussed in subsection 2.1.3. We currently employ a resource model where the library owns an arbitrary but fixed allocation of resources on a worker node in terms of cores, memory, and disk. A library has a logical type of resource called invocation slots, in which each slot runs at most 1 invocation at a time. For example, to run 8 invocations concurrently on a 32-core worker node where each invocation uses at most 4 cores, one can set the library to occupy the whole worker node and set the number of invocation slots to 8. An alternative strategy is to set each library to use 4 cores and have 1 invocation slot, so the manager can run 8 libraries concurrently.

Since a library by default takes over a whole worker, one type of library can occupy every worker in the system and prevent invocations of functions in other libraries from running. To avoid this, note that a library is a special task to a worker and by itself doesn't do any actual work. Therefore, when the manager is scheduling an invocation from another library and finds a library on a worker with no slots being actively used (an empty library), the manager instructs the worker to remove that library and reclaim resources. Invocation scheduling then happens as described above.

3.6 Parsl-TaskVineExecutor

We briefly describe our integration of TaskVine with Parsl (TaskVineExecutor) here, and reserve a full explanation to future work. Going back to Figure 1, the integration sits between the parallel

Group	Machine Prefix	CPU Model (# of Machines, GFlops)	DRAM Capacity (# of Machines)
1	d32cepyc [001-070]	AMD EPYC 7532 32-Core Processor (58, 4.4)	256GBs (58)
2	d32cepyc [076-260]	AMD EPYC 7543 32-Core Processor (117, 5.4)	256GBs (95) 2TBs (22)
3	qa-a10- [001-022]	Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz (14, 1.9)	256GBs (14)
4	qa-a40- [001-010]	Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz (7, 1.9)	256GBs (7)
5	sa-rtx6ka- [001-005]	Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz (5, 1.9)	256GBs (5)

Table 3: Major Machine Groups in the Local Cluster

library layer (Parsl) and the execution engine (TaskVine). Since Parsl maintains the DAG of invocations and sends ready ones to TaskVine, from TaskVineExecutor's perspective, it receives an arbitrary stream of function invocations. Therefore, the executor is designed to be a service process: it waits for any invocation of any function coming in at any time, packages the invocation into either a TaskVine Task or FunctionCall, executes it, and returns the result. Upon startup, the executor spawns a manager process to coordinate work, and a factory process to coordinate the number of workers in a cluster. The executor sits in the application process and sends details of ready invocations to the manager process for execution. The execution service ends when it is explicitly noted by a user or when the Python interpreter is exiting. If the interpreter exits normally, then the executor sends a shutdown signal to the manager process to stop any work and the factory process to remove workers in the cluster. For an abnormal exit (e.g., receiving SIGKILL signal), the manager and factory processes continuously check their ppids and exit/cleanup upon change.

4 EVALUATION

This section starts with a summary of the LNNI and ExaMol applications in greater detail. We then describe the general settings of all the experiments, with special settings explicitly noted in certain experiments. Lastly, our evaluation aims to answer these following questions:

- Q1: What is the effect of context reuse in the execution time of these applications?
- Q2: How is the benefit of context reuse affected when invocations take more time to execute?
- Q3: If we change the amount of available compute power by increasing or reducing the number of workers, what is the change in the execution speedup for an application?
- Q4: How many times does a function context on a worker get shared between invocations? Does the share value increase over time?
- Q5: What is the invocation overhead breakdown of different levels of context reuse, and how does increasing the level of context reuse affect this overhead?

4.1 Application Summary

4.1.1 Large-Scale Neural Network Inference. The LNNI application runs 10k to 100k inference invocations, each of which runs 16 to 1,600 inferences, on a pretrained ResNet50 model. ResNet50 is a convolutional neural network with the goal of classifying a given image to 1,000 predefined classes of objects. The application invokes functions by calling the TaskVine frontend API. The TaskVine frontend prepares the invocations and, depending on the configuration, sends ready tasks or invocations to the TaskVine backend for execution. The TaskVine backend then manages the execution of tasks, result retrievals, worker control, etc., as shown in Figure 1.

4.1.2 ExaMol. ExaMol implements workflows to explore materials design through a combination of quantum chemistry and machine learning tasks. We utilize an ExaMol application which implements a single-objective optimization of ionization potential through an active learning approach [11]. The task-scheduling logic is defined using Colmena [2, 36] and deploys PM7 calculations with OpenMOPAC [29] to gather new data concurrently with training or inference tasks implemented with Scikit-Learn and RDKit [16, 19]. Each type of task is defined using Python functions and invoked via Parsl's API [6]. Parsl maintains a graph of pending functions and sends ready ones to the *TaskVineExecutor*, which deploys functions remotely using the TaskVine backend, as described above. The total number of tasks is around 10k.

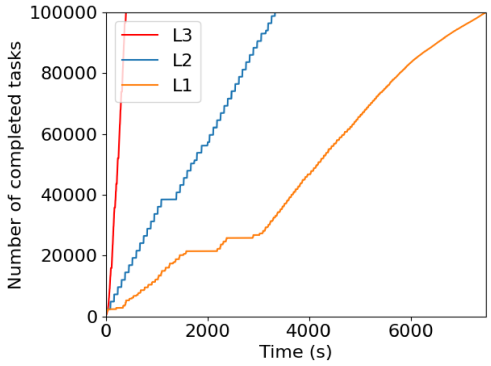
4.2 Experiment Settings

We run all applications using machines from a local heterogeneous HTCondor [31] cluster. All machines in the cluster have SATA 6GB/s SSD as local disk with 10 Gbs Ethernet link running Red Hat Enterprise Linux release 8.9 (Ootpa) as their OS. Table 3 shows 5 major machine groups with varying CPU models and DRAM capacity in the cluster that account for 96.2% of all machines used in any run. All experiments are run with a similar proportion of machine groups to that of Table 3 unless explicitly noted otherwise. Workers and invocations in applications have generous memory allocations (e.g., LNNI invocations never exceed 1GB and are allocated 4 GBs of DRAM.) Note that while a faster CPU generally results in a faster execution time, our work reduces execution time instead via state sharing and reducing the amount of unnecessarily repeated computation between invocations.

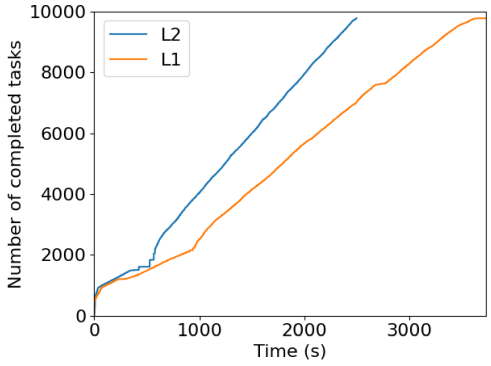
Applications are run using 10 to 150 TaskVine workers, noted explicitly in every experiment. Each worker is allocated 32 cores and 64GBs of memory and disk. For the LNNI application, each inference invocation is allocated 2 cores and 4 GBs of memory and disk, so a worker can run 16 concurrent invocations. For the Examol application, each invocation is allocated 4 cores and 8 GBs of memory and disk, resulting in a maximum of 8 concurrent invocations per worker. An application starts its execution when submitted invocations are known by the workflow system and at least 95% of the requested workers are connected.

We investigate three levels of context reuse in this paper:

- **L1**: No context reuse. This level consists of running invocations purely as tasks without use of local resources for sharing or caching on any worker. Invocations are wrapped



(a) LNNI, 100k invoc.



(b) ExaMol, 10k invoc.

Figure 6: Execution Time of Selected Applications with Different Levels of Context Reuse

as tasks with a generic Python script, and all tasks are instructed to pull all data and software dependencies from the local Panasas ActiveStor 16 [25, 37] shared file system with 77 nodes supporting up to 84 Gb/s read bandwidth and 94,000 read IOPS. This level represents a stateless task with no worker requirement and execution requirement of code, data, and arguments as shown in Table 1.

- **L2**: Context reuse on disk. This level supports data-to-invocation bindings and thus extensively uses the local disk of every worker to execute invocations. Invocations are still wrapped as tasks with a generic Python script. However, relevant data and dependencies are fetched and cached to a worker once in the first invocation, and subsequent invocations share one copy of data from the function context in the worker's cache. This level represents a stateful task as the middle ground with worker requirement of data and execution requirement of code and arguments.
- **L3**: Context reuse on disk and memory. This level supports both data-to-invocation and context-to-invocation bindings. This level goes beyond L2 and additionally utilizes the library process (subsection 3.4) to fetch and load all functions and their associated context setups to the local memory of a worker before invoking any invocation. Invocations then inherit the environment cached in the library with necessary data and contexts loaded and execute requiring only their arguments. This last level represents a stateful invocation with worker requirement of code and data and execution requirement of arguments.

	Mean	Std Deviation	Min	Max
L1	21.59	34.78	6.71	289.72
L2	13.48	3.68	6.09	45.33
L3	**4.77**	**3.43**	**2.67**	**39.51**

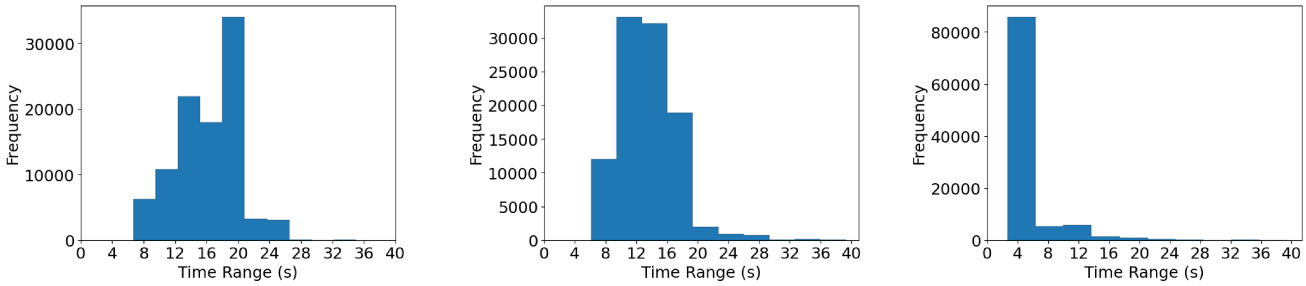
Table 4: Statistics for Invocation Run Time with Three Levels of Context Reuse in LNNI-100k (in seconds)

Lastly, we run LNNI on all 3 levels and Examol on L1 and L2. L3 is not supported yet for Examol since it's unclear whether arbitrary functions can fit in and be compatible to each other within a function context process.

4.3 Q1: Effect of Different Levels of Context Reuse on the Execution Time of Applications

To evaluate the degree of execution speedup from different levels of context reuse, we run the LNNI application with 100k inference invocations and the ExaMol application with 10k invocations both using 150 workers. Figure 6 shows the expected outcome when contexts of invocations are discovered, distributed, and retained using local resources of workers. Reusing function contexts on disk cuts the execution time of both the LNNI and the ExaMol application by 55.1% and 26.9%, respectively. For the LNNI application, reusing function context in memory further improves application execution time by 87.7% compared to that but on disk. The results follow closely to the discussion in Section 2. At the L1 level, invocations simply pull relevant data and dependencies from a shared file system, resulting in no context sharing between concurrent invocations on a worker. L2 brings the function context to the local disk of a worker, and this not only allows concurrent and subsequent invocations on the same worker to benefit from this disk utilization, but also removes the shared file system as a possible I/O bottleneck. Finally, invocations in the LNNI application at L3 reuses the context loaded to the memory space of the worker, and this removes most of the overhead per invocation. Once an invocation gets to a worker, it only has to wait for the worker to load the arguments to memory and can quickly start its execution.

Figure 7 shows the histograms of all invocations' run time of the LNNI application with 3 levels of context reuse. We do not show values greater than 40 seconds for better visualization. Shifting the level of context reuse from left to right, we see that the histogram gradually shifts to the left, with less spreading and more stable pattern. In L1, most invocations tend to execute within 12-20s, while invocations in L2 spread around 10-16s, and those in L3 cluster around 3-7s. This shows another benefit of context reuse: invocations are executed faster and in a more stable way. Table 4 presents detailed statistics of all invocations in the LNNI application under different levels of context reuse, with L3 having the fastest run time per invocation, the smallest standard deviation in run time, and the smallest maximum run time.



(a) L1 context reuse (b) L2 context reuse (c) L3 context reuse

Figure 7: Histograms of Invocation Run Time for the LNNI Application with 100k Invocations

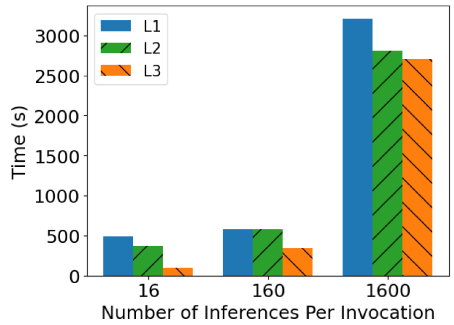


Figure 8: Effect of Increasing Individual Invocations' Run Time on LNNI's Execution Time

4.4 Q2: Effect of Increasing Invocation's Run Time on the Execution Speedup

To study the effect of increasing individual invocation's run time, we run the LNNI application with 10k inference invocations on 100 workers and vary the amount of inferences performed by each invocation: 16, 160, and 1,600. Average invocation run time for each is 6.2 seconds, 40.9 seconds, and 379.7 seconds, respectively. For this experiment, the run with L1 and 16 inferences uses a significant amount (89%) of group 2 machines. Figure 8 shows the execution time of LNNI across different number of inferences per invocation and different levels of context reuse. When each invocation runs 16 inferences in L3, the application's execution time is vastly reduced by 81% and 75% compared to L1 and L2, respectively. However, when each invocation has more computations, the speedup reduces to 41.3% and 41.2% for 160 inferences and 15.6% and 3.7% for 1,600 inferences. This is expected since while an invocation runs longer, the overhead of context reload stays the same. The benefit of context reuse, while still significant, accordingly diminishes. We can also see that context reuse L2 slightly outperforms L1 as it converts remote I/O to local I/O using the data-to-invocation binding and removes the shared file system as a bottleneck. This bottleneck is additionally shown with the run of L1 with 16 invocations as invocations are not run faster even when they are run with better hardware compared to the runs of L2 and L3 with 16 invocations. In summary, the shorter a function invocation, the more important it is for invocations to reuse their function context.

4.5 Q3: Effect of Varying the Number of Connected Workers on the Execution Speedup

To study the potential change in the execution speedup of an application, we run the LNNI application with 10k inference invocations and vary the number of connected workers between 50, 100, and 150. For this experiment, the run with L3 and 50 workers has no group 2 machines. Figure 9 shows that the LNNI application under L3 does not improve much if at all. This is because invocations are deployed, executed, and retrieved so quickly that the workflow system doesn't need additional computation power: a small amount of available computation power is sufficient when the overhead per invocation is minimal. When the number of workers is reduced to 10 and 25 for L3 however (not shown), the execution time of LNNI goes up to 455 and 145 seconds, respectively, indicating that the workflow system has idle time between sending invocations and retrieving results when the number of workers is too small.

On the other hand, the LNNI application under L1 and L2 shows slight improvements when the number of workers is increased from 50 to 150. This means that the overhead per invocation is large enough that the workflow system needs to use additional workers to occupy itself with invocation deployment, execution, and retrieval. In summary, an efficient context reuse between invocations removes a large portion of unnecessary computations and allows applications to finish the same amount of work with much less computation power.

4.6 Q4: Share Value of Deployed Function Contexts Over Time

As mentioned in subsection 3.4, TaskVine creates and deploys the library task as a daemon process on a worker that loads function contexts to memory and other local resources via executing context setup functions of invocations. Figure 10 shows the number of deployed libraries as invocations complete their executions for the LNNI application with 100k invocations on 150 workers. We see a quick ramp up in the amount of deployed libraries to respond to many connected workers waiting to execute invocations. The number of deployed libraries still increases as more invocations complete, but gradually falls off to around 2,000 active libraries since they are enough to execute many short-running invocations.

Figure 11 shows the number of invocations a library serves as they complete their executions, on average. We can see that the

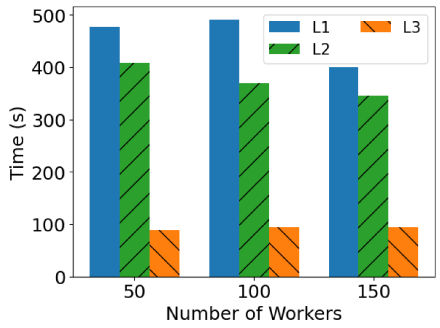


Figure 9: Effect of Increasing the Number of Workers on LNNI's Execution Time

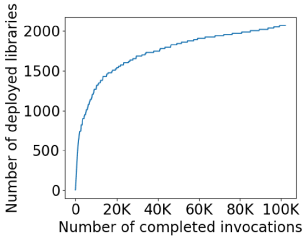


Figure 10: Number of Deployed Libraries w.r.t. Its Invocations

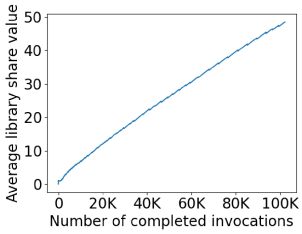


Figure 11: Average Library Share Value w.r.t. Its Invocations

share value of libraries on average grows linearly as invocations complete, since a library is prepared to serve an invocation immediately after completing a previous invocation. This result corresponds to the intuition that a deployed library is a one-time cost: subsequent invocations serviced by this library can efficiently share the same environment context with negligible overhead.

4.7 Q5: Overhead Breakdown of Different Levels of Context Reuse

We now investigate the overhead breakdown of executing functions with L2 and L3 context reuse (we skip L1 due to its high variance as shown in Table 4) using the LNNI application. All experiments in this subsection are run with both the manager and worker on the same machine to have a degree of consistency and avoid interferences from other sources (e.g., network). For level L2, we execute two remote invocations sequentially such that the first invocation (L2-Cold) pays the cost of caching data to the local disk of the worker and expanding it into a reusable format, and the second one (L2-Hot) reuses such data from the disk. For level L3, we execute one remote invocation and measure the overhead of the invocation with the associated library. We break the overhead down into 4 parts: (1) time to transfer invocation details and its data (Invoc. & Data Transfer), (2) time for the worker to setup the environment (Worker Overhead), (3) time for the invocation or library to reconstruct its necessary states in its process (Library/Invoc. Overhead), and (4) execution time of the invocation (Exec. Time).

Table 5 shows the overhead breakdown of each type of task/invocation. LNNI invocations' software dependencies contain 144 Python packages and amount to 3.1GBs of disk size in the reusable

	Invoc. & Data Transfer	Worker Overhead	Library/Invoc. Overhead	Exec. Time
L2 (Cold)	1.004	15.435	0.403	5.469
L2 (Hot)	$5.22 * 10^{-4}$	$1.18 * 10^{-3}$	0.327	5.046
L3 (Library)	0.989	15.251	2.729	N/A
L3 (Invoc.)	$\mathbf{2.34 * 10^{-4}}$	$\mathbf{2.75 * 10^{-4}}$	$\mathbf{5.14 * 10^{-4}}$	**3.079**

Table 5: Overhead Breakdown of LNNI invocations with L2 and L3 (in seconds)

format and 572 MBs when tarballed, dominating the size of input data. In L2 (Cold), we see that it takes the manager 1.004s to send the invocation details and input data to the worker, and 15.435s for the worker to receive and process the execution request from the manager. The majority of the worker overhead comes from unpacking the tarball of software dependencies into a directory to be reused by invocations. Once it is set up, the worker creates a sandbox for the first invocation and links relevant input files. The invocation takes 0.403s to deserialize Python objects from input files and reconstruct them in memory. Once it's done, it takes 5.469s to execute 16 inferences using the reconstructed Resnet50 model. Comparing L2 (Cold) and L2 (Hot), we can see the benefit of reusing context on disk as data transferring time and environment setup time are reduced significantly. This is due to less data being transferred and the data on disk being reused. Invocation overhead and execution time are roughly similar, as both L2 (Cold) and L2 (Hot) need to load the ResNet50 model's parameters from the disk to memory and build the model as a Keras Model object. Note that this model object can be reused between invocations, and L3 exploit this insight by using the library to set up a reusable environment.

L3 (Library) has comparable data transfer and worker overhead time compared to L2 (Cold) as it also needs to unpack the tarball of software dependencies locally, but has a higher overhead to set up its states. This is because it takes extra time to execute the helper function that further sets up states in memory by loading parameters and building the model in advance. Once this is done, the library reports back to the worker and is ready to receive invocations. Note that the library does no actual work as stated in subsection 3.4 so there is no measurement on its execution time. Once this cost is paid, we can see that L3 (Invoc.)'s overhead is cut down by several magnitudes in many overhead parts as it reuses the model object cached in memory by the library. Furthermore, since it only has to execute its own distinct part of computation, its execution time is cut down by about 2 seconds. This time reduction directly comes from the 2.7 seconds that the library spends once to set up the reusable states in memory (including the model building), while L2 invocations have to unnecessarily repeat the reusable part of its computation. In short, overhead per invocation with context reuse is significantly cut down as more sharing is enabled.

5 RELATED WORK

Serverless Computing. Serverless computing is broadly defined as the on-demand allocation of resources for short bursts of remote

execution, providing Functions-as-a-Service (FaaS). Prominent services include Amazon Lambda [5], Apache OpenWhisk [12], and Azure Serverless [9]. These platforms allow users to invoke remote functions using the appropriate scheme as defined by the vendor. To utilize these platforms however, functions must be manually registered and invoked in a platform-specific way. In many cases, proprietary cloud storage must be used, and compute resources must be provided by the vendor. In contrast, workflow systems that enable context reuse as stated above are more application-specific. Functions don't have to be registered with workflow systems in advance. All dependencies of a function can be dynamically discovered and packaged instead of a user's manual dependency specification. Computational power (i.e., workers/compute nodes) is not vendor-specific: an application using TaskVine's context setup APIs can spawn workers anywhere, on cloud or on premise. Finally, workflow systems have a more proactive approach to data movement and tracking: instead of being hidden behind a cloud storage, data is explicitly moved between workers to maximize the aggregated bandwith of the system.

Workflow Systems. Workflow systems encompass a variety of softwares assisting in the deployment of work across computational resources [10, 32, 39]. These systems strike a balance between ease of application development, performance benefits, and other desirable characteristics of scalable applications. Here we pay specific attention to the distinction between task-based and invocation-based workflow systems. Workflows are traditionally viewed as task-based applications. Invocation-based systems imply that context resides on the remote execution site, reducing startup cost and latency. Ray [18] is an integrated parallel workflow system in which tasks are expressed as decorated Python functions which may be invoked. There may only be one actor, or context per Ray worker, and data is typically accessed through a shared file system. TaskVine however may have multiple libraries installed on a worker, and data may be shared across invocations inside a worker, leveraging the benefit of many cores on a single node.

Parallel Libraries. Parallel libraries offer the convenience to express and execute a workflow via invoking functions in a given programming language. Function invocations are represented in a DAG in which concurrent tasks are easily recognizable. Parallel libraries such as Ray, Parsl [6], and Dask [23] construct a DAG of tasks in which an underlying task scheduler may utilize to execute the workflow. Our contribution in TaskVine is a sophisticated scheduler which directly supports invocation as a standalone unit of computation and an alternative execution model to task. With the ability to have many libraries, or contexts, on a single worker, TaskVine function invocations can be efficiently matched with execution platforms at a fine-grained level. While users have the option of expressing their workflows using the TaskVine frontend APIs, the choices of a parallel library to use are much wider. The TaskVine backend is fully integrated with popular libraries like Parsl and Dask, in which TaskVine acts like the execution engine for workflows described in the language of either library. This combines the sophisticated workflow expression and DAG creation of the parallel libraries while also utilizes the proactive data management and efficient computation execution features of TaskVine.

Data Staging Technologies. HPC-scale distributed file systems such as Lustre [7] and Panasas [37] are effective in their purpose and often the best solution in cases where the size of application data exceeds the available local storage on compute nodes. However a predominant contemporary issue in such distributed file systems is the scale and concurrency at which they can handle and will become a bottleneck for data-intensive applications. This has led to the development of various data staging technologies such as burst buffers and ephemeral file systems which reside across node local storage. Examples of these include BeeOND [3], GekkoFS [33], and BurstFS [34]. A key component of TaskVine is data management and distribution. For many applications we encourage the use of local storage and inter-node communication instead of relying on distributed file systems. The fundamental difference between TaskVine and other node-local storage implementations is that TaskVine manages both the workflow and storage, fully leveraging the awareness of data locality in making decisions for task scheduling. This holistic capability of TaskVine makes it possible to efficiently utilize the local resources of compute nodes to support both the task and invocation execution modes of data-intensive and/or function-centric workflows. Other burst buffer and ad-hoc file system implementations must be deployed and managed independently, and while providing generally favorable performance, valuable information about storage is hidden from the workflow manager, blocking the full potential of using local storage for stateful computations.

6 CONCLUSION AND FUTURE WORK

Workflow systems allow users to express and execute large scale applications on remote computational resources via a DAG of independent stateless tasks. These workflows are often interpreted and executed in a task-based model, in which tasks and its dependencies are bundled together and delivered to remote nodes at runtime and live in the cluster only as long as the task is running. This execution model is detrimental to the performance of function-centric applications however. Invocations differentiate themselves via their arguments, but when executed as tasks, must also reload their contexts, including code and data, for every execution. Our work addresses this execution deficiency by treating functions and invocations as first-class abstractions for execution: reusable function contexts are first discovered via analysis of the function code and/or user's specification, distributed efficiently across all workers in the system, and then retained indefinitely on the workers waiting for invocations. Subsequent invocations then can reuse the function context and its already set-up environment, and only have to wait for arguments to load into their memory space before promptly starting to execute. Future work includes further improvements to the function-centric programming model in order to facilitate a seamless discovery of high-level contexts among invocations to the same function, with necessary code, data, and dependencies packaged automatically without the need for user intervention.

ACKNOWLEDGEMENT

This work was supported by National Science Foundation grant OAC-1931348. Logan Ward was supported by the Exascale Computing Project (17-SC-20-SC) of the U.S. Department of Energy (DOE) and by DOE's Advanced Scientific Research Office (ASCR) under contract DE-AC02-06CH11357.

REFERENCES

[1] [n. d.]. Altair Grid Engine. https://www.altair.com.

[2] 2021 [Online]. Colmena. ExaLearn and Parsl Teams. Available: https://colmena.readthedocs.io/en/latest/index.html.

[3] 2023. BeeGFS. https://www.beegfs.io/c/. [Online; accessed 20-July-2023].

[4] Bela Abolfathi, David Alonso, Robert Armstrong, Éric Aubourg, Humna Awan, Yadu N Babuji, Franz Erik Bauer, Rachel Bean, George Beckett, Rahul Biswas, et al. 2021. The lsst desc dc2 simulated sky survey. *The Astrophysical Journal Supplement Series* 253, 1 (2021), 31.

[5] Inc Amazon.com. [n. d.]. Amazon Lambda. https://aws.amazon.com/lambda/

[6] 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. https://doi.org/10.1145/3307681.3325400

[7] Sean Cochrane, Ken Kutzer, and L McIntosh. 2009. Solving the HPC I/O bottleneck: Sun™ Lustre™ storage system. *Sun BluePrints™ Online, Sun Microsystems* (2009).

[8] Cloudpickle contributors. 2023. Cloudpickle. https://github.com/cloudpipe/cloudpickle

[9] Microsoft Corporation. [n. d.]. Microsoft Azure. https://azure.microsoft.com/en-us

[10] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[11] Hieu A. Doan, Garvit Agarwal, Hai Qian, Michael J. Counihan, Joaquín Rodríguez-López, Jeffrey S. Moore, and Rajeev S. Assary. 2020. Quantum Chemistry-Informed Active Learning to Accelerate the Design and Discovery of Sustainable Energy Storage Materials. *Chemistry of Materials* 32, 15 (May 2020), 6338–6346. https://doi.org/10.1021/acs.chemmater.0c00768

[12] The Apache Software Foundation. [n. d.]. Apache OpenWhisk. https://openwhisk.apache.org/

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[14] Anaconda Inc. 2020. Anaconda Software Distribution. https://docs.anaconda.com/.

[15] Anaconda Inc. 2023. conda-pack. https://anaconda.org/conda-forge/conda-pack.

[16] Greg Landrum. 2013. Rdkit documentation. *Release* 1, 1-79 (2013), 4.

[17] Meta. 2023. Pursuing groundbreaking scale and accelerating research using Meta's Research SuperCluster. https://ai.meta.com/blog/supercomputer-meta-research-supercluster-2023/

[18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[19] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[20] Thanh Son Phung, Ben Clifford, Kyle Chard, and Douglas Thain. 2023. Maximizing Data Utility for HPC Python Workflow Execution. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (<conf-loc>, <city>Denver</city>, <state>CO</state>, <country>USA</country>, </conf-loc>) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 637–640. https://doi.org/10.1145/3624062.3624136

[21] Thanh Son Phung, Logan Ward, Kyle Chard, and Douglas Thain. 2021. Not All Tasks Are Created Equal: Adaptive Resource Allocation for Heterogeneous Tasks in Dynamic Workflows. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. 17–24. https://doi.org/10.1109/WORKS54523.2021.00008

[22] Christopher J. Rhodes. 2013. Large Hadron Collider (LHC). *Science Progress* 96, 1 (2013), 95–109. https://doi.org/10.3184/003685013X13623370524107 arXiv:https://doi.org/10.3184/003685013X13623370524107 PMID: 23738440.

[23] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.

[24] Tim Shaffer, Thanh Son Phung, Kyle Chard, and Douglas Thain. 2023. Landlord: Coordinating Dynamic Software Environments to Reduce Container Sprawl. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1376–1389. https://doi.org/10.1109/TPDS.2023.3241598

[25] Tim Shaffer and Douglas Thain. 2017. Taming metadata storms in parallel filesystems with metaFS. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. 25–30.

[26] Andrew Siegel, Erik W. Draeger, Jack Deslippe, Thomas Evans, Marianne M. Francois, Timothy C. Germann, Daniel F. Martin, and William Hart. 2021. *Map Applications to Target Exascale Architecture with Machine-Specific Performance Analysis, Including Challenges and Projections*. Technical Report. https://doi.org/10.2172/1838979

[27] Barry Sly-Delgado, Nick Locascio, David Simonetti, Brett Wiseman, Ben Tovar, and Douglas Thain. 2022. PONCHO: Dynamic Package Synthesis for Distributed and Serverless Python Applications. In *Workshop on High Performance Serverless Computing*. doi: 10.1145/3526060.3535459.

[28] 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. https://doi.org/10.1145/3624062.3624277

[29] James J. P. Stewart. 2012. Optimization of parameters for semiempirical methods VI: more modifications to the NDDO approximations and re-optimization of parameters. *Journal of Molecular Modeling* 19, 1 (Nov. 2012), 1–32. https://doi.org/10.1007/s00894-012-1667-x

[30] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2003. Condor and the Grid. In *Grid Computing: Making the Global Infrastructure a Reality*, Fran Berman, Anthony Hey, and Geoffrey Fox (Eds.). John Wiley. isbn: 0-470-85319-0.

[31] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience* 17, 2-4 (2005), 323–356. doi: 10.1002/cpe.v17:2/4.

[32] Matteo Turilli, Vivek Balasubramanian, Andre Merzky, Ioannis Paraskevakos, and Shantenu Jha. 2019. Middleware building blocks for workflow systems. *Computing in Science & Engineering* 21, 4 (2019), 62–75.

[33] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. GekkoFS - A Temporary Distributed File System for HPC Applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 319–324. https://doi.org/10.1109/CLUSTER.2018.00049

[34] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An Ephemeral Burst-Buffer File System for Scientific Applications. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 807–818. https://doi.org/10.1109/SC.2016.68

[35] L. Ward. 2023. ExaMol. https://github.com/exalearn/examol

[36] Logan Ward, Ganesh Sivaraman, J Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C Redfern, Rajeev S Assary, Kyle Chard, Larry A Curtiss, et al. 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, 9–20.

[37] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. 1–17.

[38] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*.

[39] Charles (Chao) Zheng, Ben Tovar, and Douglas Thain. 2017. Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*. doi: 10.1109/CCGRID.2017.9.