PONCHO: Dynamic Package Synthesis for Distributed and Serverless Python Applications

Barry Sly-Delgado
University of Notre Dame
bslydelg@nd.edu

Nick Locascio
University of Notre Dame
nlocasci@nd.edu

David Simonetti
University of Notre Dame
dsimone2@nd.edu

Brett Wiseman
University of Notre Dame
bwisema3@nd.edu

Ben Tovar
University of Notre Dame
btovar@nd.edu

Douglas Thain
University of Notre Dame
dthain@nd.edu

ABSTRACT

An increasing number of distributed applications operate by dispatching function invocations across the nodes of a distributed system. To operate correctly, the code and data dependencies of the function must be distributed along with the invocations in some way. When translating applications to work on large scale distributed systems, managing these dependencies becomes challenging: delivery must be scalable to thousands of nodes; the dependencies must be consistent across the system; and the method must be usable by an unprivileged developer. As a solution, in this paper we present PONCHO, which is a lightweight Python based toolkit which allows users to discover, package, and deploy dependencies as an integral part of distributed applications. PONCHO encapsulates a set of commands to be executed within an environment. PONCHO offers a lightweight solution to create and manage environments increasing the portability of scientific applications as well as reproducibility. In this paper, we evaluate PONCHO with real-world applications in the fields of physics, computational chemistry, and hyperparameter optimization, We observe the challenges that arise when creating and distributing an environment and measure the overheads that emerge as a result.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

Distributed systems, Systems, Serverless

ACM Reference Format:
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 *Proceedings of the 2nd Workshop on High Performance Serverless Computing (HiPS '22), June 30, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3526060.3535459

HiPS '22, June 30, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9311-9/22/06…$15.00
https://doi.org/10.1145/3526060.3535459

1 INTRODUCTION

An increasing number of scientific applications are structured around the concept of remote function invocation. In this model, a distributed application consists of a manager process that dispatches function invocations to a fleet of cooperating worker nodes, whether drawn from an HPC facility or a cloud provider. These remote invocations can take several forms, depending on how much needs to be "shipped" to the worker node. At one extreme, the worker has nothing pre-installed, and the function along with all of its inputs must be sent on each invocation. At the other extreme, all the components of the function are already present at the worker, and the manager must simply send a lightweight invocation record. Various intermediate forms are also possible.

Regardless of the method, the software environment is an essential part of development and execution of scientific applications. Users, in turn, need a method to determine, retrieve and consolidate the dependencies for their applications. Execution models within serverless and distributed computing must enable the means to have dependencies be accessible to a user's application. Producing the package needed for the environment is often a manual process. Users need a certain amount of understanding of an application to install any needed dependencies. This generates issues when attempting to share or reproduce work. How does a user go about determining the dependencies needed for an application?

There are multiple challenges that arise when generating an environment. For example, a user's application may function with Python 3.5 but not with Python 3.8. This might occur for various reasons such as the application itself or one of its dependencies require that version. However, if Python 3.5 is not installed on the system, how does the user go about getting it installed? For local applications, this issue can be easily solved using a package manger like Conda, Spack and Nix [3, 10, 11], which are package managers primarily used for finding and downloading packages. However, challenges appear when attempting to execute these applications on distributed systems.

Distributed applications can be executed in environments such as HPCs, clusters and clouds where many users can execute tasks simultaneously. With these applications, each user must have a method to ensure that their application executes in the proper environment. Using the same method as one would locally may not be feasible for multiple reasons. For example, using the local method with a distributed application would require each machine to use a package manager and install the needed dependencies. In this scenario, each machine is making a request over a shared

network for the same dependencies. Using this method does not scale well on large distributed systems containing many nodes. Not only will this affect the performance of a user's application but other users applications will be affected as well. Furthermore, there is also the possibility that each request could produce different results by downloading different versions of the requested dependencies. Thus, a new method must be able to scale up to thousands of nodes and have consistent environment delivery such that each node used for an application will be receiving the same environment.

Our solution, PONCHO, is a lightweight Python based toolkit which allows users to synthesize environments from a concise, human-readable JSON file containing the necessary information required to build a self-contained Conda virtual environment needed to execute scientific applications on distributed systems. The PONCHO toolkit is composed of multiple components. Each component can be used individually or in tandem to suit a user's needs. **poncho_package_analyze** is a tool that performs a static analysis of a python application to determine all of its top-level module dependencies, the interpreter version it uses, and creates a JSON specification file. **poncho_package_create**, consolidates each component needed to run the application via a specification file into an portable environment tarball, which can then be transported to remote systems. **poncho_package_run** unpacks and activates an environment on a given system and executes a task within said environment. Finally, **poncho_package_serverize** combines an environment with a given Python function and converts it into a continuously running module that can be invoked in a serverless manner.

To examine PONCHO we tested it with various real-world applications in the fields, of particle physics, computational chemistry and hyperparameter optimization. In these tests, we measure the various overheads that can arise when using PONCHO. These overheads are measured in environment creation and delivery using various configurations. Our results show that PONCHO can consistently create and deliver environments without extraordinary overhead.

2 DISTRIBUTED PYTHON APPS

2.1 Background

Enabling distributed computing within a high-level language such as Python provides many benefits to users. Python-native distributed applications allows users to access high-throughput and parallel computation using the Python interface. This eases the process of creating and executing distributed applications for users who are familiar with Python since they are working within a familiar interface. Distributed python applications, like other distributed applications, need to manage several functions, such as managing allocations, managing resources, and scheduling tasks to form one coherent and reliable system. It is critical that no segment of the process is prone to faults as it would affect the system as a whole.

There are currently various Python-native frameworks that enables distributed computing. Parsl is a scripting library which enables users to scale their Python applications on distributed systems [5]. The library is able to dynamically create dependency graphs of the tasks needed to be executed by the application. Dask is a tool that allows user to encode parallel algorithms in Python using primitive Python data types [19]. PythonTask is an extension to Work Queue [1] which allows users to run Python functions as tasks. These tasks are returned as Python values which enable the user to manipulate data easier as opposed to returning results in a file, which may add unnecessary I/O. PythonTask allows users to specify the environment needed to run each task, as it is possible that different tasks within an application require different environments, where an environment is a set of dependencies that a task needs to execute properly.

2.2 Challenges

Challenges arise when distributing tasks using a distributed application. Some remote systems may not have the specific packages that are required installed. As a result, the application is unable to run. Thus, being able to generate an environment with the needed components is crucial. Furthermore, if the incorrect versions of said packages are installed instead, the application may produce unexpected results. On rarer occasions, Python itself or the correct Python version may not be installed either. One way one might attempt to solve this would be the same way one does locally, by installing any needed dependencies on each system that the user will use. However, as an applications evolves, new packages will have to be installed manually or old packages will need to be updated. This can produce inconsistency across the machines in a distributed system where different machines have different versions of the same packages. As a result applications may produce unexpected results or may fail to execute completely. Thus, having one transportable environment that is sent to each machine is preferred as each environment has the same source and is guaranteed to be identical on each machine.

In the case of transportable environments for distributed applications, a challenge arises when deploying an environment on remote systems. The level in which an environment is deployed can vary on different types of systems. For example, if there are multiple tasks to be executed on the same node, deploying the environment to each task could create an unnecessary overhead if these tasks share the same dependencies. This can reduce the scalability of an application if there is a large amount of nodes being used. However, if these tasks require different environments it is preferable to do so as each task may not be able to execute on another task's specified environment. Conversely, the environment could be deployed per node. This may work well for certain computation models such as FaaS, where a single node is dedicated to a single function. However, for other distributed computation models the opposite issue could arise when multiple tasks require different environments. In this situation, one solution may be to create one large environment that encompasses each task's dependencies. If an application is running and a new task arrives at a node requiring a different environment than the one already deployed on the node, merging the environments may be a possible solution. However, there may be conflicts with the packages when trying to merge these environments. Also, if these environments share dependencies, an inflated environment containing many redundancies may be created, that is if redundant packages are not omitted during the merging process. As more tasks with different dependencies are created, the larger that environment becomes, using a larger amount of allocated resources

and reducing the scalabilty of an application. Thus, understanding configuration of the application is necessary to minimize the overheads that arise when using transportable environments.

3 THE PONCHO TOOLKIT

The PONCHO Toolkit allows users to dynamically synthesize environments in which their application can be executed in. This environment is specified using a declarative JSON file. This environment can then be distributed to remote machines ensuring that applications execute properly. The toolkit is comprised of four parts: `poncho_package_analyze`, `poncho_package_create`, `poncho_package_run`, and `poncho_package_serverize`. Each tool can be used individually or with each other to aid in the creation and distribution of the environment and the execution of the application. The environment is packed within a tarball which can include Conda packages, pip packages, remote git repositories, and data retrievable via http or https. Figure 1 depicts the intended workflow of using the PONCHO toolkit's base operations.

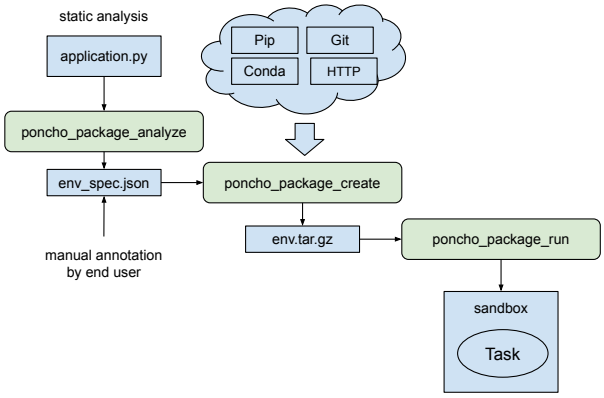

Figure 1: PONCHO Toolkit

poncho_package_analyze is a tool which preforms a static analysis on Python code to determine the Conda or pip packages the program depends on. This analysis can be performed at either the program or function level. With this information, `poncho_package_analyze` generates a file specifying the packages and dependencies needed to run the Python code. To achieve this, `poncho_package_analyze` searches through all the Python package imports in the given Python code and determines which ones come from an external source; `poncho_package_analyze` then iterates through the user's local Conda and pip environment and matches all the external Python package imports with a Conda or pip package. Afterwards, `poncho_package_analyze` generates a JSON encoded specification file that details the package dependencies needed to run the application. Users are also able to create this specification via manual annotation. The PONCHO environment specification enables users to specify which Conda and pip packages need to be included into the environment. Users may also include git repositories and files retrievable via http or https to include within the environment. Figure 2 shows an example environment specification that includes Conda packages, a pip package, a git repository, and retrievable data set.

poncho_package_create reads an environment specification file and synthesizes a Conda environment that includes the specified dependencies. Any specified git repositories will be cloned into the environment and any data sourced via http or https will be fetched and included in as well. The specification used to create the package is also saved within the package. The environment is then packed using Conda Pack [4] into a tarball. This package is a portable environment that can then be distributed to remote systems. Since this tool uses Conda to create the environment, it is optimal that the versions of required dependencies are specified in the specification file. Not listing package versions may result in a substantial increase in runtime when creating the environment.

poncho_package_run is a tool that is used to unpack and activate the environment and run specified commands within said environment. This operation has two inputs: an environment package and a command to execute as a task. This environment is unpacked into a temporary directory unless specified otherwise. The path of any included git repositories or files fetched via http are set as environment variables. When users specify the unpack directory, multiple instances of `poncho_package_run` may reuse the same directory and the environment will only need to be unpacked once.

```
"conda": {
  "channels":["defaults", "conda-forge"],
  "dependencies":["python=3.9", "pip",
    "numpy", "ndcctools", pip": {["dill"]}],
},
"git": {
  "CODE_DIR": {
    "remote": "https://github.com/...
  }
},
"http": {
  "REF_DATA": {
    "type": "tar",
    "compression": "gzip",
    "url":"http://ncbi.nih.gov/...db.tgz",
  }
}
```

Figure 2: Sample Package Specification

poncho_package_serverize takes the final step and combines a packaged environment, then name of a python function (found in that environment) and integrates it with a standard process for accepting function invocations over a socket. This combination of assets forms a **network function**: a single object that can be moved from place to place in the network, and then easily invoked as a self-contained function. This enables the easy transformation of application code from local function to a "serverless" application.

The PONCHO tools have a natural integration with Work Queue [8], a framework for building large distributed applications that span thousands of machines drawn from clusters, clouds, and grids. Integrating PONCHO with Work Queue allows for users to execute

their application with a stable environment on each machine, in a variety of configurations, depending on the need. Three different modes of invocation are available, shown in Figure 3.

The first invocation method (**package per task**) causes assets to be sent with each task. In this configuration, the user attaches a package to each task with `t.python_package(p)` prior to submitting the task. `poncho_package_run` is then added as a wrapper for the task. It takes in the environment package and the original task command as arguments. Thus, when a task arrives at a worker the environment is unpacked, activated and the function executed inside the environment. When the task is complete, the unpacked environment is removed to reclaim storage space. This method is most appropriate when different functions require different environments, and the cost of deployment is small relative to the runtime of the function.

The second invocation method (**package per worker**) causes all assets to be deployed once with each worker, and then shared among all tasks that use that worker. Workers are deployed by the Work Queue Factory, a tool that submits, monitors, and (when needed) removes workers to maintain a desired level of service. To attach a package to a worker, the user simply raises the package statement up to `wkrs.python_package(p)`. This causes the factory to attach the desired package to the worker submission, and then invoke `poncho_package_run` appropriately as it starts each worker. Thus, each worker gets a concrete Python environment. Since tasks inherit the environment of the worker it is on, all tasks will run in the desired environment. This is appropriate if all tasks require the same environment, and amortizes the setup cost across multiple tasks. However, it still results in every task invoking its own Python interpreter, and paying the cost of importing the environment into memory.

The third invocation method (**serverless**) eliminates these costs by sharing the startup costs among multiple tasks. From the user's perspective, this is accomplished by transforming an existing Python function into a network function via `poncho.NetworkFunction`. `poncho_package_analyze` then performs a static analysis on the function, generating a PONCHO specification. This specification is then created into a tarball via `poncho_package_create`. Included within the tarball is the serialized function and generalized code for the coprocess. `poncho_package_run` executes each worker node with this specified process. When the worker is initialized, it loads the network function as a coprocess and makes it a target for invocation of arriving tasks. From the user's perspective, tasks are defined and invoked in the same way as before, except that the function definition and assets are already located at the worker.

These configurations displays PONCHO's flexibility to be incorporated within various models of computation. For each configuration, little needs to be changed to modify the structure. Different configurations modify the locality of invocation of a given task that produces the same result. Changing the locality of invocation may be beneficial depending on the type of application being executed. The division between setup and invocation reflects the cost between the two. When the division is skewed towards the setup side, node setup occurs faster while task invocations will take longer. When the division is skewed towards the task side, node setup takes longer while task invocation becomes quicker. It is important to note that setup is a one time cost per node while invocations typically occur more than once. Thus, the most efficient balance between setup and invocation is application dependent. Figure 4 presents a detailed view of each invocation level. With invocation level 1, the division is skewed towards setup. Levels 2 and 3 skew this division towards invocation.

```
# Invocation Level 1: Package Per Task

import work_queue as wq

def func(x, y, z):
    ...

wkrs = wq.Factory('condor', 'name')
with wkrs:
    t = PythonTask(func, 7, 42, 88)
    t.python_package('pkg.tar.gz')
    q.submit(t)
    ...
-------------------------------------
# Invocation Level 2: Package Per Worker

import work_queue as wq

def func(x, y, z):
    ...

wkrs = wq.Factory('condor', 'name')
wkrs.python_package('pkg.tar.gz')
with wkrs:

    t  = PythonTask(func, 7, 42, 88)
    q.submit(t)
    ...
-------------------------------------
# Invocation Level 3: Serverless

import work_queue as wq

def func(x, y, z):
    ...

remote_func = poncho.NetworkFunction(func)
wkrs = wq.Factory('condor', 'name')
wkrs.coprocess(remote_func)
with wkrs:

    t  = RemotePythonTask('func',7,42,88)
    q.submit(t)
    ...
```

Figure 3: Examples of Invocation Levels

4 EXAMPLE APPLICATIONS

TopEFT [6] is a distributed python application that runs a particle physics analysis analyzing the top quark. The data is sourced from

experiments from the CMS project at CERN. These experiments use the Large Hadron Collider (LHC) which creates billions of collision events. These events are analyzed with TopEFT. The application uses a manager-worker paradigm to distribute events to be analyzed. The number of events given to each task can be augmented by specifying the chunksize. The results are generated in a sequence of events and then accumulated at the end. Since TopEFT is dependent on the Coffea framework [23], it is crucial that it is accessible to each task.

Colmena-XTB is a computational chemistry application that runs an analysis across multiple nodes. This application is dependent on Colmena [2] which is a library for building simulation workflows on HPC. Colmena is composed of "Thinkers" and "Doers". "Thinkers" generate tasks to be sent to a task server. The "Doer", receives the task requests and deploys them. Requests are asynchronous thus the order of results is non-deterministic.

Shadho [14] is a application for hyperparameter optimization. The application attempts to determine the best hyperparamters to use for a certain model based on the hardware being used. To do this, Shadho observes the results of training on various sets of hyperparameters to determine the optimal solution. This application uses Work Queue for distributed task management.

5 EVALUATION

5.1 Environment Creation

With environment creation, a user's configuration can drastically increase the overhead of the application. Since Conda is used by `poncho_package_create` to create the environment, PONCHO's environment execution time is reliant on Conda's environment solver execution time. Because of this, how the environment specification file is stated can drastically change the environment creation time. More specifically, whether package versions are listed or not. Because of Conda's relatively slow environment solving time. we added the option to use Mamba [16] as well. Mamba is a CLI that acts as a replacement of Conda and often offers a faster solution to those created by Conda.

We categorized package specifications into three categories, empty version specifications, immediate version specifications, and nested version specifications. With **empty version specifications**, Packages are listed without a version. This is the minimum information required to create an environment. With **immediate version specifications**, the versions of packages that are immediately imported within an application are specified. Finally, with **nested version specifications**, every package that will be included in the environment has its version specified. We then created each type of specification for TopEFT and SHADHO. Thus, for each application and each specification type we created the environment using both Mamba and Conda. Before each environment creation, it was ensured that Conda's cache was empty so that times were consistent on each run.

When creating an environment, a large portion of the time is spent by attempting to solve the environment. Execution time is also dependent on the number of packages in an environment. As the number of required packages increase, the overhead is expected to increases as well. With empty version specifications, users can

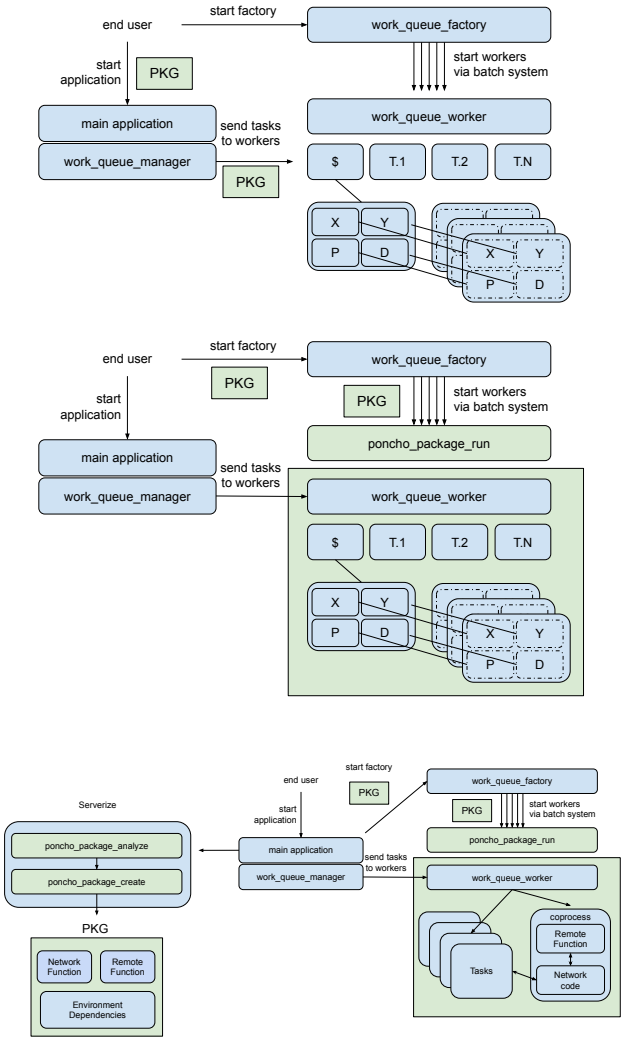

Figure 4: Detail of Invocation Levels

Detail of each of the invocation levels. Top: Each task is invoked independently with its own package deployment. Middle: Each worker is invoked with its own package deployment, that is then shared among all tasks running on that worker. Bottom: Each worker deploys a coprocess for executing tasks in a common serverless style.

expect environment creation to take a relatively long time. In comparison, far shorter times can be expected when specifying the versions of packages as shown by Table 1. Mamba is able to create a solution in a faster time with empty version specifications when compared to Conda. To test the overhead produced from environment creation, we cleared Conda's cache, however it is expected that the overhead for creating the environment would be reduced over time as Conda caches packages.

Certainly, if an application has strict requirements for the packages that are imported it is best to have a nested specification. With immediate and empty specifications, there is some work to be done

with by the environment solver to insure compatibility between components. Thus, downloaded packages may change due to updates to the requested packages and resolved environments may come out different. It is estimated that given a set of pip and Conda packages 60 percent of pip packages and 80 percent of Conda packages have been modified within 10 days [21]. Thus, if an application has strict requirements maintaining a nested specification is the way to go. Otherwise, if an application is more flexible to updates to dependencies one may use immediate or empty specifications.

TopEFT Specification	Conda	Mamba
Empty Version Specification	2940.30s	281.19s
Immediate Version Specification	512.04s	107.70s
Nested Version Specification	118.87s	103.79s
Shadho Specification	Conda	Mamba
Empty Version Specification	257.11s	131.37s
Immediate Version Specification	258.18s	132.45s
Nested Version Specification	159.80s	130.66s

Table 1: Environment Creation Performance

Mean environment tarball creation times per application and specification type using Conda and Mamba

5.2 Unpacking and Activating

When using PONCHO there will be the overhead of unpacking and activating the environment on a select machine. PONCHO environments are contained within tarballs, thus they must be unpacked once they arrive on the designated machine. These tarballs can vary in size depending on the amount of required dependencies. Thus, it is important that we examine how the size of an environment tarball affects the time it takes to unpack and activate.

To measure the time that it takes to unpack an environment, we first generated environment tarballs for three applications, TopEFT, Colmena-XTB, and Shadho. Each tarball's size was recorded as shown in Table 2. Then, with a trivial python program, we used `poncho_package_run` to execute the program for each application's tarball.

From Table 2, we can observe the expected correlation between the size of the environment tarball and the time taken to unpack and activate the environment.

5.3 Invocations

With the use of Work Queue, there a various configurations to deliver an environment to remote machines and execute tasks. The configuration used for environment delivery can very much affect the runtime of the application. The variations of execution

Application	Compressed	Unpacked	Mean Time
TopEFT	594MB	2GB	20.97s
Shadho	438MB	1.4GB	12.47s
Colmena	1.4GB	4.8GB	46.30s

Table 2: Unpacking Performance

Mean time taken to unpack and activate an environment.

Application	Level 1 Per Task	Level 2 Per Worker	Level 3 Serverless
TopEFT	175.70s	28.17s	27.93s
Shadho	126.46s	24.12s	25.42s
Colmena	487.07s	59.96s	50.54s

Table 3: Performance of Invocation Levels

Average time to execute minimal 100 tasks on 10 workers with each application package and invocation level.

models include the three invocation levels: per task, per worker, and serverless. When sending the environment per task, each task will unpack and activate the environment before it executes the task it is given. In this configuration, users may benefit from the ability to give tasks different environments. When sending the environment per worker, each worker unpacks and activates the environment before receiving any tasks. As tasks arrive, they inherit the environment of the worker. Thus, each task will execute in the same environment. In the serverless configuration, each worker will again unpack and activate the environment. After, the worker will start the network function coprocess which includes the remote function to be invoked. Tasks invoke the function via the coprocess and return the results to the user.

To test the various configurations, we measured invocation times for each invocation level and application using a trivial Python function. For each application the environment has already been built using poncho_package_create. Thus, the times measured will show how each configuration would affect the runtime of the application.

From the results shown in Table 3, the runtime when distributing the environment per worker is exceedingly better than distributing the environment per task. This is due to many task redundantly extracting the environment within the per task configuration. The per worker and serverless configurations produce similar results. The reasoning behind the similarity can be to traced to the task executed being fairly minimal.

6 RELATED WORK

Package mangers - PONCHO uses Conda to manage packages. There are a variety of other applications that manage dependencies such as Nix and Spack which have been evaluated within HPC environments [9, 11]. Work has been been conducted studying the lifetimes of dependencies with Binder Python Containers. [21]. As dependencies grow, there is added resource consumption used by package managers. Landlord [22] attempts to solve this issue by reducing storage consumption by merging container specifications.

Distributed Python Frameworks PONCHO was implemented to be used along side Work Queue [8], a distributed framework. However there are various frameworks that enable the creation distributed python applications such as Parsl, Dask and PyCOMPSs [5, 19, 24]. These frameworks manage the distribution of tasks between machines.

Containers on HPC - Much work has been done to bring containerization onto HPC environments. Though PONCHO is not a container itself, its intended use is similar to those of containers. Primarily to be accompanied with the execution of a distributed application. PONCHO differs from traditional containers by being

a able to statically analyze code and automatically synthesizing the needed dependencies. PONCHO along with Work Queue can be easily integrated within a user's application to provide consistency within the execution process. The container applications Singularity[15], Shifter[12], and Charliecloud [18], differs from other container applications such as Docker [17], by being intended for use in HPC environments. Though, there were early attempts to integrate docker within HPCs [13]. Studies have been conducted to evaluate container usage in HPC environments such as comparing Singularity, Shifter and Docker [20], where Singularity was found to be the most suitable for use with HPCs. Shifter alone has been studied in use with scientific applications at Blue Waters [7].

7 CONCLUSION

In this paper we presented the PONCHO toolkit for dynamically synthesizing environments for distributed and severless applications. We've measured overheads using various configurations to depict the expected overhead from using PONCHO. These results show that added overhead is both application and configuration dependent. Future work may include dynamically optimizing execution configurations at the application level, minimizing task dependencies, and exploring other deployment methods.

ACKNOWLEDGEMENT

This work was supported in part by NSF grant OCI-1931348.

REFERENCES

[1] Cctools. https://github.com/cooperative-computing-lab/cctools.
[2] Colmena, ai-steering for hpc, https://colmena.readthedocs.io.
[3] Anaconda software distribution, https://docs.anaconda.com/, 2020.
[4] Anaconda Inc. conda-pack: 0.6.0. https://conda.github.io/conda-pack/.
[5] 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 *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 25–36, New York, NY, USA, 2019. Association for Computing Machinery.
[6] A. Basnet, K. Bloom, F. Canelli, S. S. Cruz, J. E. P. Cortezon, J. R. G. Fernández, A. T. Fernandez, R. Goldouzian, B. A. Gonzalez, M. Hildreth, K. Lannon, J. Lawrence, S. P. Liechti, C. E. Mcgrady, K. Mohrman, H. Nelson, B. Tovar, Y. Wan, A. Wightman, B. Winer, F. Yan, B. R. Yates, H. Yockey, and M. Zarucki. Topeft/topcoffea: Topcoffea 0.1, Aug. 2021.
[7] M. Belkin, R. Haas, G. W. Arnold, H. W. Leong, E. A. Huerta, D. Lesny, and M. Neubauer. Container solutions for hpc systems: a case study of using shifter on blue waters. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–8. 2018.
[8] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.
[9] B. Bzeznik, O. Henriot, V. Reis, O. Richard, and L. Tavard. Nix as hpc package management system. In *Proceedings of the Fourth International Workshop on HPC User Support Tools*, HUST'17, New York, NY, USA, 2017. Association for Computing Machinery.
[10] E. DOLSTRA, A. LÖH, and N. PIERRON. Nixos: A purely functional linux distribution. *Journal of Functional Programming*, 20(5-6):577–615, 2010.
[11] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.
[12] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia. Shifter: Containers for hpc. In *Journal of physics: Conference series*, volume 898, page 082021. IOP Publishing, 2017.
[13] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, pages 33–49, 2015.
[14] J. Kinnison, N. Kremer-Herman, D. Thain, and W. Scheirer. SHADHO: Massively Scalable Hardware-Aware Distributed Hyperparameter Optimization. In *IEEE Winter Conference on Applications of Computer Vision*, pages 1–10, 2018. doi: 10.1109/WACV.2018.00086.
[15] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.
[16] Mamba. https://github.com/mamba-org/mamba: The fast cross-platform package manager.
[17] D. Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.
[18] R. Priedhorsky and T. Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.
[19] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015.
[20] O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vázquez. Containers in hpc: A scalability and portability study in production biological simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 567–577, 2019.
[21] T. Shaffer, K. Chard, and D. Thain. An Empirical Study of Package Dependencies and Lifetimes in Binder Python Containers. In *IEEE International Conference on e-Science*, 2021.
[22] T. Shaffer, N. Hazekamp, J. Blomer, and D. Thain. Solving the Container Explosion Problem for Distributed High Throughput Computing. In *International Parallel and Distributed Processing Symposium*, 2020. doi: 10.1109/IPDPS47924.2020.00048.
[23] N. Smith, L. Gray, M. Cremonesi, B. Jayatilaka, O. Gutsche, A. Hall, K. Pedro, M. Acosta, A. Melo, S. Belforte, et al. Coffea columnar object framework for effective analysis. In *EPJ Web of Conferences*, volume 245, page 06012. EDP Sciences, 2020.
[24] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.