JetLag: An Interactive, Asynchronous Array Computing Environment

Steven R. Brandt
Louisiana State University
sbrandt@cct.lsu.edu

Alex Bigelow
University of Arizona
alexrbigelow@email.arizona.edu

Sayef Azad Sakin
University of Arizona
sayefsakin@email.arizona.edu

Katy Williams
University of Arizona
kawilliams@email.arizona.edu

Katherine E. Isaacs
University of Arizona
kisaacs@cs.arizona.edu

Kevin Huck
University of Oregon
khuck@cs.uoregon.edu

R. Tohid
Louisiana State University
mraste2@lsu.edu

Bibek Wagle
Louisiana State University
bwagle3@lsu.edu

Shahrzad Shirzad
Louisiana State University
sshirz1@lsu.edu

Hartmut Kaiser
Louisiana State University
hkaiser@cct.lsu.edu

ABSTRACT

We describe an interactive computing environment called JetLag. JetLag implements the following features of Phylanx project: (1) Phylanx, a Python-based asynchronous array computing toolkit; (2) the APEX performance measurement library; (3) a performance visualization framework called Traveler; (4) the Tapis/Agave Science as a Service middleware; and (6) a container infrastructure that includes Docker-based Jupyter notebook for the client and a singularity image for the server.

The running system starts with a user performing array computations on their workstation or laptop. If, at some point, the calculation the user is performing becomes sufficiently intensive or numerous, it can be packaged and sent to another machine where it will run (through the batch queue system if there is one), produce a result, and have that result sent back to the user's local interface. Whether the calculation is local or remote, the user will be able to use APEX and Traveler to diagnose and fix performance related problems.

The JetLag system is suitable for a variety of array computational tasks, including machine learning and exploratory data analysis.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**; • **Human-centered computing → Interactive systems and tools**.

KEYWORDS

asynchronous, array, research environment, interactive computing, performance tuning, performance visualization, cloud computing

ACM Reference Format:
Steven R. Brandt, Alex Bigelow, Sayef Azad Sakin, Katy Williams, Katherine E. Isaacs, Kevin Huck, R. Tohid, Bibek Wagle, Shahrzad Shirzad, and Hartmut Kaiser. 2020. JetLag: An Interactive, Asynchronous Array Computing Environment. In *Practice and Experience in Advanced Research Computing (PEARC '20), July 26–30, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3311790.3396657

1 INTRODUCTION

A trend in modern computing and data analysis is to move away from huge, statically-compiled codebases and toward dynamic, interactive runtimes based on high-level languages such as Python, R, or Julia. This trend best supports problems confined to a single host. When analysis requires more compute, users are left reimplementing analytic pipelines using cloud compute environments emphasizing resiliency through distributed disk storage.

The JetLag environment allows programmers to send Phylanx functions and arguments to remote hosts. It serializes both the code and the arguments, sends them to the specified destination, runs the code, serializes the results and performance data and sends them back. While the code is running, its status can easily be checked. In this fashion, a large number of calculations may be launched and managed simultaneously.

Phylanx provides a framework that can execute arbitrary Python code using an asynchronous many-task runtime system [12]. Using decorators, Python functions are transpiled into C++ data structures that can be analyzed and reconfigured to provide optimal performance. The Phylanx environment, JetLag, integrates Phylanx with performance monitoring and analysis tools, and remote execution and job control capabilities. Phylanx aims to enable the user to make use of multiple resources, including cloud resources.

Phylanx is based on HPX, an open source C++ library for parallelism and concurrency [8]. HPX provides a high performance,

Number of Threads	Walltime(s)	Speedup
1	123.97	1
2	69.28	1.78
4	51.30	2.41
8	49.12	2.52
16	61.01	2.03

Table 1: Execution time of the Logistic Regression example when the number of threads are varied. Speedup in the table is calculated with respect to the same example running on a single thread.

Number of Threads	Walltime(s)	Speedup
1	24.96	1
2	17.43	1.43
4	12.52	1.99
8	8.7	2.86
16	8.35	2.98

Table 2: Execution time of the Alternating Least Squares example when the number of threads are varied. Speedup in the table is calculated with respect to the same example running on a single thread.

cutting edge implementation of the C++ parallel standard. In addition, it provides extensions to enable computations.

2 THE COMPONENTS OF JETLAG

In this section, we describe the individual components of the JetLag framework.

2.1 Phylanx and HPX

Phylanx [12] is a framework that allows us to build a C++ data structure that describes an execution tree from any Python3 function. Phylanx uses a Python decorator to gain access to the function's abstract syntax tree. Using this tree, it generates a representation of the function in PhySL, a human-readable, intermediate language used by the Phylanx project to assemble the data structure.

The C++ execution tree is executed in an asynchronous fashion, using HPX [8], a high-performance Asynchronous Many-Task (AMT) runtime system extending C++ programming language to expose uniform API for distributed and parallel programming. HPX has proven itself capable of running on many thousands of cores [9] in a distributed setting and provides a solid backend for the Phylanx execution system. HPX also comes with performance measurement and adaptive optimizations through customizable runtime policies, which are further extended and maintained through APEX (Section 2.2).

Array data structures in Phylanx are implemented using Blaze [7], a smart expression-template library for C++. In addition to its flexibility and high performance, Blaze can make use of HPX for thread level parallelism make it an ideal choice for this framework.

Listing 1: Binary Logistic Regression

```
# Logistic Regression Algorithm
def lra(x, y, alpha, iterations, enable_output):
    weights = np.zeros(np.shape(x)[1])
    transx = np.transpose(x)
    pred = np.zeros(np.shape(x)[0])
    error = np.zeros(np.shape(x)[0])
    gradient = np.zeros(np.shape(x)[1])
    step = 0
    while step < iterations:
        if (enable_output):
            print("step:␣", step, ",␣", weights)
        pred = 1.0 / (1.0 + np.exp(-np.dot(x, weights)))
        error = pred - y
        gradient = np.dot(transx, error)
        weights = weights - (alpha * gradient)
        step += 1
    return weights
```

Listing 2: Alternating Least Squares

```
# Alternating Least Square
def ALS(ratings, reg, num_factors, iterations, alpha, X, Y):
    num_users = np.shape(ratings)[0]
    num_items = np.shape(ratings)[1]
    conf = alpha * ratings
    conf_u = np.zeros((num_items,1))
    conf_i = np.zeros((num_items,1))
    c_u = np.zeros((num_items, num_items))
    c_i = np.zeros((num_users, num_users))
    p_u = np.zeros((num_items,1))
    p_i = np.zeros((num_users,1))
    I_f = np.identity(num_factors)
    I_i = np.identity(num_items)
    I_u = np.identity(num_users)
    i = 0
    u = 0
    k = 0

    XtX = np.zeros((num_factors, num_factors))
    YtY = np.zeros((num_factors, num_factors))
    A = np.zeros([num_factors, num_factors])
    b = np.zeros([num_factors])
    while k < iterations:
        YtY = np.dot(np.transpose(Y), Y) + reg * I_f
        XtX = np.dot(np.transpose(X), X) + reg * I_f
        while u < num_users:
            conf_u = conf[u, :]
            c_u = np.diag(conf_u)
            p_u = conf_u != 0
            A = YtY + np.dot(np.dot(np.transpose(Y), c_u), Y)
            b = np.dot(np.dot(
                np.transpose(Y), c_u + I_i), np.transpose(p_u))
            X[u, :] = np.dot(inverse(A), b)
            u = u + 1
        while i < num_items:
            conf_i = conf[:, i]
            c_i = np.diag(conf_i)
            p_i = conf_i != 0
            A = XtX + np.dot(np.dot(np.transpose(X), c_i), X)
            b = np.dot(np.dot(
                np.transpose(X), c_i + I_u), np.transpose(p_i))
            Y[i, :] = np.dot(inverse(A), b)
            i = i + 1
        u = 0
        i = 0
        k = k + 1
    result = np.vstack((X, Y))
    return result
```

Binary Logistic Regression [1] Algorithm (LRA), implemented in Python, a snippet of which is shown in Listing 1 was used for experimental purpose. We used a custom dataset consisting of 10000 features and 10000 observations. Table 1 shows the speedup of the Logistic Regression example with respect to single threaded execution. Execution time reported is for 1000 iterations. Furthermore, we also implemented Alternating Least Squares [5] (ALS) algorithm in Python as shown in Listing 2. The speedup of the Phylanx execution of the Alternating Least Squares example with respect to single threaded execution is shown in Table 2. The execution time reported corresponds to the Alternating Least Squares example running with the MovieLens Dataset [4] for one iteration. The number of movies was set at 1000 and the factor was set at 40.

2.2 APEX and Performance Measurement

APEX [6] (Autonomic Performance Environment for Exascale) is a performance measurement library for distributed, asynchronous multitasking runtime systems such as HPX, the runtime upon which Phylanx is built. It provides lightweight measurement (capable of supporting tasks of duration less than 1ms) and high concurrency. To support performance measurement in systems that employ user level threading, APEX uses a dependency chain rather than the call stack to produce traces. APEX supports both synchronous and asynchronous introspection. The synchronous module of APEX uses an event API and event listeners. Whenever an event occurs, APEX, using this aforementioned API, makes a decision to start, stop, yield or resume timers for correct measurements. The asynchronous module, however, does not rely on events, rather it executes desired functionality periodically.

The policy engine of APEX provides a lightweight API to engineer policies that can improve the performance of the application, execute a desired functionality on the runtime or select important runtime and application parameters. There are two ways to register a policy: either explicitly triggered or asynchronously periodic. A triggered policy can be initiated by a specific event within the HPX runtime. There is a set of generic events provided by APEX, such as initialization/finalization, creation of a new thread, timer start/stop events, or message send/receive events. Additionally, it is also possible to provide a user defined event, also known as a custom trigger. The second class of policies, the periodic policy, operates without any event. Instead, this policy uses a defined timer which is specified during the policy's registration. All policies are stored in a policy queue and executed as instructed. The policy engine is integrated with Active Harmony [10], an online tuning library. Defined policies can use this library to search for a set of optimum parameters by minimizing a measurement value from APEX, such as wall time of a measured region/task or by looking at any other introspection data gathered by APEX.

APEX has native support for performance profiling, in which all tasks scheduled by the runtime are measured. At any point during the execution, the profile contains the number of times each task was executed and the total time spent executing that type of task. In order to perform detailed performance analysis involving task dependency analysis, full event traces including event identification and start/stop times have to be captured. To that end, APEX is integrated with the Open Trace Format 2 [3] (OTF2) library—an open, robust format for large scale parallel application event trace data. OTF2 is a robust reader/writer library and binary format specification that is typically used for high-performance computing (HPC) trace data. In order to capture full task dependency chains in HPX applications, all tasks are uniquely identified by their GUID (globally unique identifier) and the GUID of their parent task. These GUIDs are captured as part of the OTF2 trace output. Figure 3 depicts an execution where OTF2 data is used to bring back PAPI [11] performance counter data.

Furthermore, it should be noted, that APEX has introduced minimal overhead. In our LRA experiments, for example, it increased execution time by approximately 1%.

2.3 Traveler and Performance Visualization

Traveler, shown in Figures 1 and 3, is a web-based performance visualization framework designed with asynchronous many-task runtimes (AMTs) in mind. It supports charts commonly used to visualize performance in HPC, such as time series, histograms, source code, and Gantt charts but also aggregated execution graphs [14] as more commonly analyzed to understand AMT execution.

These visualizations can be used to quickly verify expected behavior of the system or to locate anomalies or unexpected patterns leading to poor performance. We've previously consulted the execution graph visualizations in the tuning of performance parameters [13].

In support of Phylanx and Jetlag performance analysis, Traveler manages *dynamic linking* between charts—clicking on a Phylanx primitive (a building block of the Phylanx execution tree, basically a single function) will highlight instances of that primitive in all charts and time axes have linked panning and zooming. Figure 1 shows an example. Design and implementation to support further charts, create default configurations, and optimize responsiveness for larger datasets is ongoing.

Traveler can read three types of data output by JetLag: (1) OTF2 data, which includes task traces and can optionally be annotated with PAPI counters (as in Figure 3) as well as both sampled OS information and extra dependency information through APEX (as in Figure 1), (2) execution graph data generated directly by Phylanx, and (3) raw source code. A Jetlag run may send to Traveler any subset of this data.

While Traveler can be run independently, loading and serving data files from the command line, it is also designed to integrate with the JetLag workflow. JetLag can send the collected performance data directly to a running Traveler server. The overall workflow is summarized in Figure 2.

2.4 Agave/Tapis and Job Submission and Control

The Agave Framework (recently rebranded as Tapis by TACC), is used to describe and submit jobs to remote resources, monitor progress, transfer files, and share data [2]. Additionally, the Agave/Tapis platform provides us with detailed provenance data. In combination with the Singularity image we create to run the Phylanx jobs, we are able to make any remote system look the same to the user and simplify their access to jobs.

For each remote system accessible to a user of JetLag, there must be an administrator who sets up and configures the machine for Agave/Tapis, providing information about the queuing system, the file system, ensuring the Singularity image is installed, and so forth. The administrator can then grant access to each user that wishes to use the Phylanx application. Users only need to know a `jetlag_id` for the machine in order to submit jobs there.

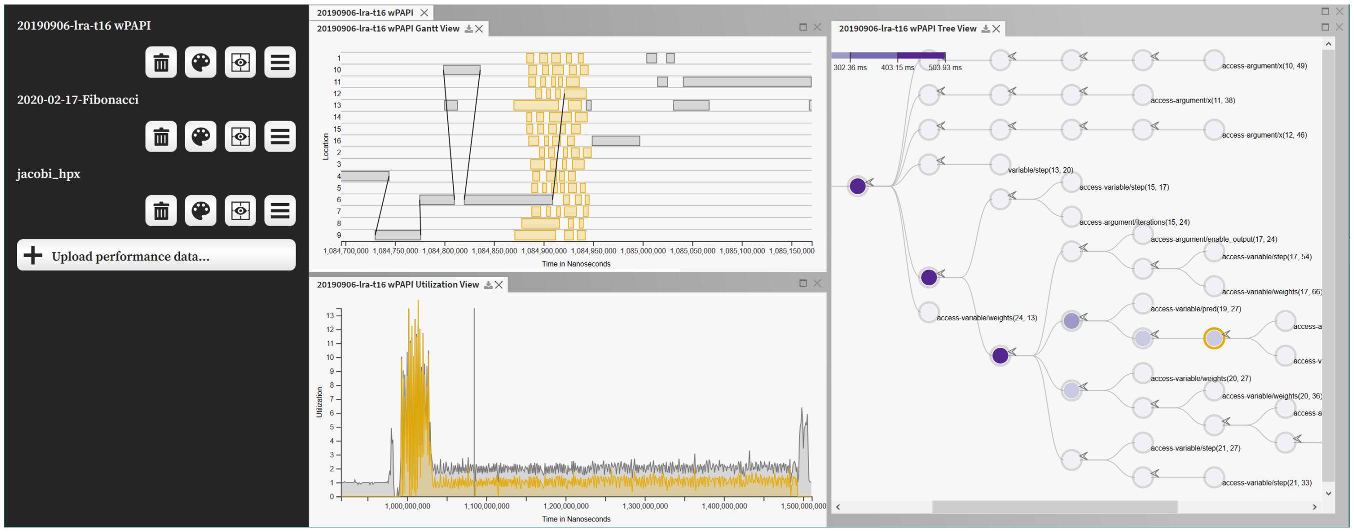
20190906-lra-t16 wPAPI
2020-02-17-Fibonacci
jacobi_hpx
Upload performance data...
20190906-lra-t16 wPAPI Gantt View
20190906-lra-t16 wPAPI Utilization View
20190906-lra-t16 wPAPI Tree View
Location
Time in Nanoseconds
Utilization
302.36 ms
403.15 ms
503.93 ms

Figure 1: Traveler web interface: The black bar on the left shows available datasets and the interface for adding and modifying charts. These can be automatically populated through Jetlag. On the right, three charts are shown for the LRA dataset: a Gantt chart of executing tasks, a Utilization chart, and a node-link diagram of the execution tree. A single primitive is selected in the execution tree (yellow outline) and all matching tasks in the Gantt chart and associated utilization in the Utilization chart are also highlighted in yellow. The black lines across locations in the Gantt chart are calculated from the GUIDs reported by APEX.

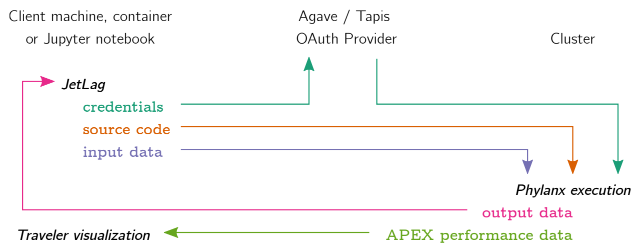

Figure 2: A diagram of the JetLag workflow. JetLag sends credentials, source code, and input data to a cluster, using Agave / Tapis for authentication. The results of executing the source code on the cluster with Phylanx are returned to JetLag on the client's machine, container, or Jupyter notebook, and as any collected APEX performance data is also returned for visualization with Traveler.

Users can also provide a callback URL to be invoked when the job completes. This URL can contain parameters such as `JOB_ID` for crafting a custom message. This URL can trigger anything from a cell phone notification through a service like Pushbullet, or the next step in a workflow. Alternatively, an email address can be given as the URL, in which case a message will be sent to that email address.

3 FUTURE WORK

As Phylanx and the tools around it continue to mature, we have a number of different directions we wish to take this research environment. First and foremost is to a distributed setting. We are actively working on various primitives for distributed computation, including multi-dimensional distributed arrays. Our work in creating a Singularity image with the complete Phylanx installation and integrating it with Agave/Tapis was preparatory for that step.

To date, much of our performance gathering experiments have been relatively small scale, and of necessity have only run on a single node. We anticipate the need to refine and optimize the data collection process as we scale to larger applications.

4 CONCLUSION

We have described an advanced research computing environment, an interactive, Python-based asynchronous array analysis tool. It uses Phylanx and HPX to achieve high concurrency, APEX for performance monitoring, and Traveler to visualize performance data. It is capable of managing runs and sharing data through the underlying Agave/Tapis Science as a Service middleware.

ACKNOWLEDGMENTS

This work was funded by the NSF Phylanx project award 1737785 and by the Department of Defense through DTIC Contract FA8075-14-D-0002/0007. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Department of Defense.

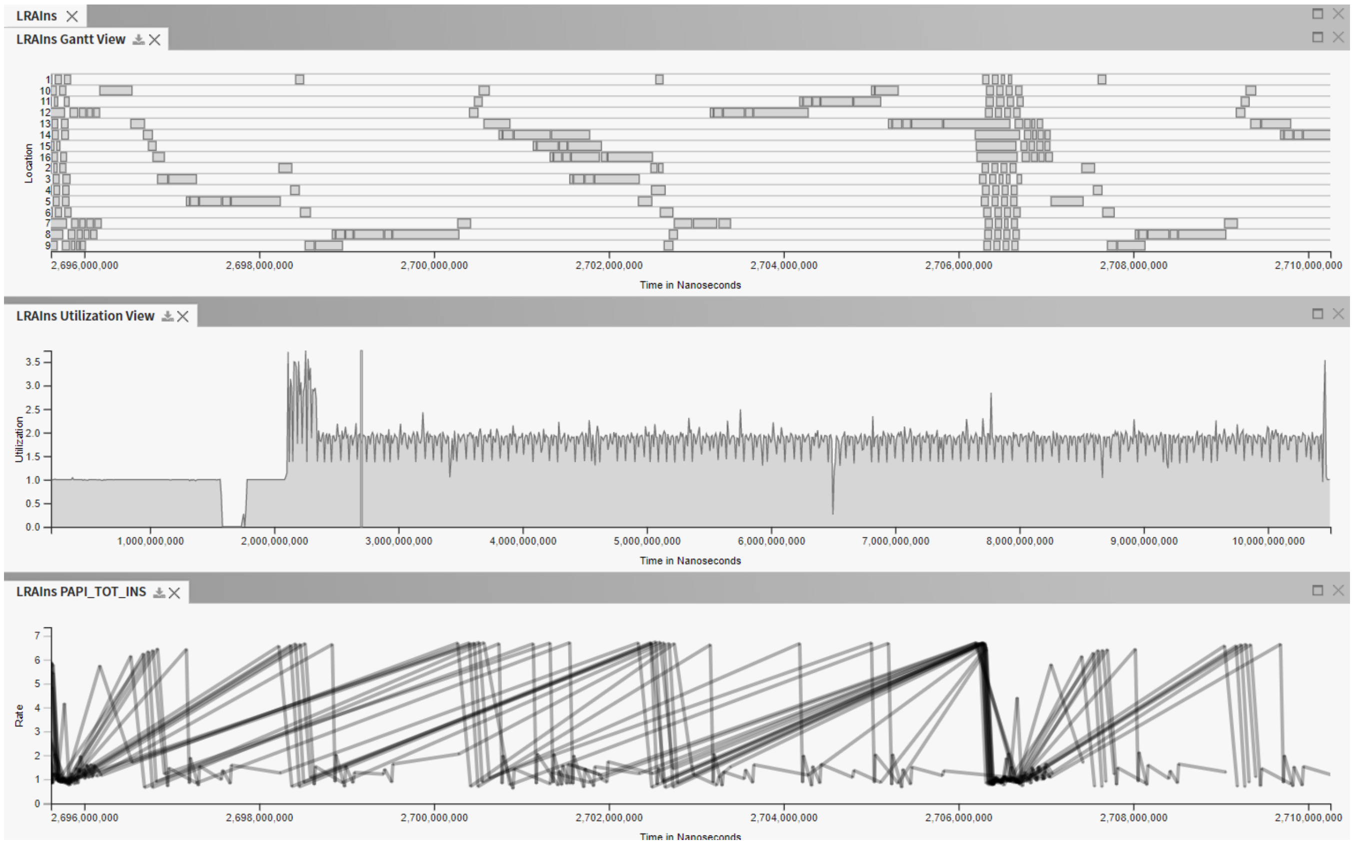

Figure 3: Traveler charts from an an LRA run, showing a Gantt chart, Utilization chart, and time series of PAPI counters collected across all locations (bottom chart). Sampled `/proc/meminfo` and related data can also be charted.

REFERENCES

[1] C Bishop. 2007. Pattern Recognition and Machine Learning (Information Science and Statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York* (2007).

[2] Rion Dooley, Steven R Brandt, and John Fonner. 2018. The Agave Platform: An Open, Science-as-a-Service Platform for Digital Science. In *Proceedings of the Practice and Experience on Advanced Research Computing*. ACM, 28.

[3] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. 2012. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *Advances in Parallel Computing*. Vol. 22. IOS Press, Amsterdam, NL, 481–490.

[4] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.

[5] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. Ieee, 263–272.

[6] Kevin Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen Malony, Thomas Sterling, and Rob Fowler. 2015. An Autonomic Performance Environment for Exascale. *Supercomputing Frontiers and Innovations* 2, 3 (2015), 49–66. https://superfri.org/superfri/article/view/64

[7] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. High performance smart expression template math libraries. In *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 367–373.

[8] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.

[9] Dominic C Marcello, Kundan Kadam, Geoffrey C Clayton, Juhan Frank, Hartmut Kaiser, and Patrick M Motl. 2016. Introducing Octo-tiger/HPX: Simulating interacting binaries with adaptive mesh refinement and the fast multipole method. *Proceedings of Science* (2016), 13–17.

[10] Cristian Ţapuş, I-Hsin Chung, and Jeffrey K Hollingsworth. 2002. Active harmony: Towards automated performance tuning. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, ACM/IEEE, Baltimore, Maryland, USA, 44–44.

[11] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[12] R Tohid, Bibek Wagle, Shahrzad Shirzad, Patrick Diehl, Adrian Serio, Alireza Kheirkhahan, Parsa Amini, Katy Williams, Kate Isaacs, Kevin Huck, et al. 2018. Asynchronous Execution of Python Code on Task-Based Runtime Systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 37–45.

[13] Bibek Wagle, Mohammad Alaul Haque Monil, Kevin Huck, Allen D. Malony, Adrian Serio, and Hartmut Kaiser. 2019. Runtime Adaptive Task Inlining on Asynchronous Multitasking Runtime Systems. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article Article 76, 10 pages. https://doi.org/10.1145/3337821.3337915

[14] Katy Williams, Alex Bigelow, and Katherine E Issacs. 2020. Visualizing a Moving Target: A Design Study on Task Parallel Programs in the Presence of Evolving Data and Concerns. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (Jan 2020), 1118–1128. https://doi.org/10.1109/TVCG.2019.2934285