Asynchronous Execution of Python Code on Task-Based Runtime Systems

R. Tohid*, Bibek Wagle*, Shahrzad Shirzad*, Patrick Diehl*,
Adrian Serio*, Alireza Kheirkhahan*, Parsa Amini*,
Katy Williams†, Kate Isaacs†, Kevin Huck‡, Steven Brandt * and Hartmut Kaiser*
* Louisiana State University, † University of Arizona, ‡ University of Oregon
E-mail: {mraste2, bwagle3, sshirz1, patrickdiehl, akheir1}@lsu.edu, {hkaiser, aserio, sbrandt, parsa}@cct.lsu.edu,
khuck@cs.uoregon.edu, kisaacs@cs.arizona.edu, kawilliams@email.arizona.edu
URL: Patrick Diehl (https://orcid.org/0000-0003-3922-8419)

***Abstract*—Despite advancements in the areas of parallel and distributed computing, the complexity of programming on High Performance Computing (HPC) resources has deterred many domain experts, especially in the areas of machine learning and artificial intelligence (AI), from utilizing performance benefits of such systems. Researchers and scientists favor high-productivity languages to avoid the inconvenience of programming in low-level languages and costs of acquiring the necessary skills required for programming at this level. In recent years, Python, with the support of linear algebra libraries like NumPy, has gained popularity despite facing limitations which prevent this code from distributed runs. Here we present a solution which maintains both high level programming abstractions as well as parallel and distributed efficiency. Phylanx, is an asynchronous array processing toolkit which transforms Python and NumPy operations into code which can be executed in parallel on HPC resources by mapping Python and NumPy functions and variables into a dependency tree executed by HPX, a general purpose, parallel, task-based runtime system written in C++. Phylanx additionally provides introspection and visualization capabilities for debugging and performance analysis. We have tested the foundations of our approach by comparing our implementation of widely used machine learning algorithms to accepted NumPy standards.**

***Index Terms*—Array computing, Asynchronous, High Performance Computing, HPX, Python, Runtime systems**

I. Introduction

The ever-increasing size of data sets in recent years have given the rise to the term "big data." The field of big data includes applications that utilize data sets so large that traditional means of processing cannot handle them [1], [2]. The tools that operate on such data sets are often termed as big data platforms. Some prominent examples are Spark, Hadoop, Theano and Tensorflow [3], [4].

One field which benefits form big data technology is Machine learning. Machine learning techniques are used to extract useful data from these large data sets [5], [6]. Theano [7] and Tensorflow [8] are two prominent frameworks that support machine learning as well as deep learning [9] technology. Both frameworks provide a Python interface, that has become the *lingua franca* for machine learning experts. This is due, in part, to the elegant math-like syntax of Python that has been popular with domain scientists. Furthermore, the existence of frameworks and libraries catering to machine learning in Python such as NumPy, SciPy and Scikit-Learn have made Python the de facto standard for machine learning.

While these solutions work well with mid-sized data sets, larger data sets still pose a big challenge to the field. Phylanx tackles this issue by providing a framework that can execute arbitrary Python code in a distributed setting using an asynchronous many-task runtime system. Phylanx is based on the open source C++ library for parallelism and concurrency (HPX [10], [11]).

This paper introduces the architecture of Phylanx and demonstrates how this solution enables code expressed in Python to run in an HPC environment with minimal changes. While Phylanx provides general distributed array functionalities that are applicable beyond the field of machine learning, the examples in this paper focus on machine learning applications, the main target of our research.

This paper makes the following contributions:

- Describe the *futurization* technique used to decouple the logical dependencies of the execution tree from its execution.
- Illustrate the software architecture of Phylanx.
- Demonstrate the tooling support which visualizes Phylanx's performance data to easily find bottlenecks and enhance performance.
- Present initial performance results of the method.

We will describe the background in Section III, Phylanx's architecture in Section IV, study the performance of several machine learning algorithms in Section V, discuss related work in Section II, and present conclusions in Section VI.

II. Related Work

Because of the popularity of Python, there have been many efforts to improve the performance of this language. Some specialized their solutions to machine learning while others provide wider range of support for numerical computations in general. NumPy [12] provides excellent support for numerical computations on CPUs within a single node. Theano [13] provides a syntax similar to NumPy, however, it supports multiple architectures as the backend. Theano uses a symbolic representation to enable a range of optimizations through its compiler. PyTorch [14] makes heavy use of GPUs for high performance execution of deep learning algorithms. Numba [15] is a jit compiler that speeds up Python code by using decorators. It makes use of LLVM compiler to compile and optimize the decorated parts of the Python code. Numba relies on other libraries, like Dask [16] to support distributed computation. Dask is a distributed parallel computation library implemented purely in Python with support for both local and distributed executions of the Python code. Dask works tightly with NumPy and Pandas [17] data objects. The main limitation of Dask is that its scheduler has a per task overhead in the range of few hundred microseconds, which limits its scaling beyond a few thousand of cores. Google's Tensorflow [8] is a symbolic math library with support for parallel and distributed execution on many architectures and provides many optimizations for operations widely used in machine learning. Tensorflow is a library for dataflow programing which is a programming paradigm not natively supported by Python and, therefore, not widely used.

III. Technologies utilized to implement Phylanx

HPX [10], [11] is an asynchronous many-task runtime system capable of running scientific applications both on a single process as well as in a distribued setting on thousands of nodes. HPX achieves a high degree of parallelism via lightweight tasks called HPX threads. These threads are scheduled on top of the Operating System threads via the HPX scheduler, which implements an $M : N$ thread scheduling system. HPX threads can also be executed remotely via a form of active messages [18] known as Parcels [19], [20]. We briefly introduce the technique of *futurization*, which is utilized within Phylanx. For more details we refer to [11].

```
//Definition of the function
int convert(std::string s){ return std::stoi(s); }
//Asynchronous execution of the function
hpx::future<int> f = hpx::async(convert, "42");
//Accessing the result of the function
std::cout << f.get() << std::endl;
```

Listing 1. Example for the concept of futurization within HPX. Example code was adapted from [21].

The concept of futurization [22] is illustrated in Listing 1. The function in Line 2 is intended to be executed in parallel on one of the lightweight HPX threads. Line 4 shows the usage of the asynchronous return type `hpx::future<T>`, the so-called *Future*, of the asynchronous function call `hpx::async`. Note that `hpx::async` returns the future immediately even though the computation within `convert` may not have started yet. In Line 6, the result of the future is accessed via its member function `.get()`. Listing 1 is just a simple usecase of futurization which does not handle synchronization very efficiently. Consider the call to `.get()`, if the Future has not become "ready" `.get()` will cause the current thread to suspend. Each suspension will incur a context switch from the current thread which adds overhead to the execution time. It is very important to avoid these unnecessary suspensions for maximum efficiency.

Fortunately, HPX provides barriers for the synchronization of dependencies. These include: `hpx::wait_any`, `hpx::wait_any`, and `hpx::wait_all().then()`. These barriers provide the user with a means to wait until a `future` is ready before attempting to retrieve its value. In HPX we have combined the `hpx::wait_all().then()` facility and provided the user with the `hpx::dataflow` API [22] demonstrated in Listing 2.

```
template <typename Func>
future<int> traverse(node& n, Func && f)
{
    // traversal of left and right sub-tree
    future<int> left =
        n.left ? traverse(*n.left, f)
                        : make_ready_future(0);
    future<int> right =
        n.right ? traverse(*n.right, f)
                        : make_ready_future(0);

    // return overall result for current node
    return dataflow(
        [&n, &f](future<int> l, future<int> r)
                        -> int
        {
            // calling .get() does not suspend
            return f(n) + l.get() + r.get();
        },
        std::move(left), std::move(right)
    );
}
```

Listing 2. Example for the concept of `hpx::dataflow` for the transverse of a tree. Example code was adapted from [21].

Listing 2 uses `hpx::dataflow` to traverse a tree. In Line 5 and Line 8 the futures for the left and right traversal are returned. Note that these futures may have not been computed yet when they are passed into the `dataflow` on Line 13. The user could have used an `hpx::async` here instead of `hpx::dataflow`, but the Future passed to the called function may have suspended the thread while waiting for its results in the `.get()` function. The `hpx::dataflow` will not pass the Future arguments

to the function until all of the Futures passed to the `hpx::dataflow` are "ready". This avoids the suspension of the child function call. In Section IV *futurization* and the facility `hpx::dataflow` are heavily utilized to construct the asynchronous architecture of Phylanx.

Finally, the last technology we used to guide the development of Phylanx is NumPy. NumPy [12] is a highly optimized numerical computation library for Python. NumPy is used in many scientific applications and supports highly performant, multidimensional array operations for scientific computing. Phylanx uses the library's API as the interface to the user. In addition, Phylanx supports numerical computation on NumPy data objects through pybind11 [23] without the need for data copies.

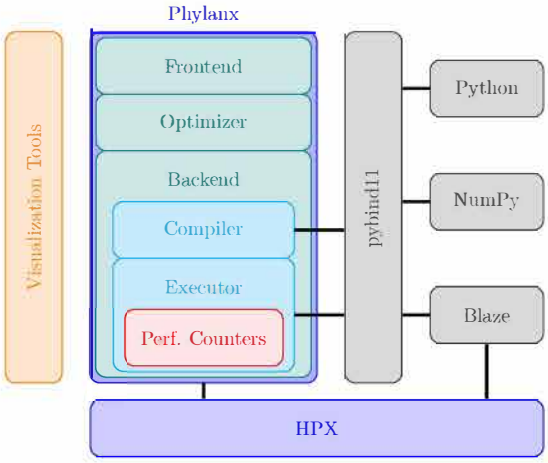

Fig. 1. Overview of the Phylanx toolkit and its interactions with external libraries.

IV. Phylanx

In Python, the order of code blocks determines the execution order of a program and implicit parallelism is only available within each block. Therefore, asynchrony and parallelism across code blocks must be explicitly explored by the programmer, a process which is tedious and error prone. In this section we discuss the implementation of our approach in Phylanx for automatic generation of task graphs and the infrastructure used for running them on HPX for parallel, asynchronous, distributed execution. We also discuss a suite of analysis and optimization tools included in the Phylanx toolkit. Figure 3 provides an overview of program flow in Phylanx.

A. Frontend

The Phylanx frontend provides two essential functionalities:

- Transform the python code into a Phylanx internal representation called PhySL (Phylanx Specification Language).
- Copy-free handling of data objects between Python and Phylanx executor (in C++).

In addition, the frontend exposes two main functionalities of Phylanx that are implemented in C++ and required for generation and evaluation of the execution tree in Python. These functions are the `compile` and `eval` methods.

1) Code Transformation: Performance benefits of many-task runtime systems, like HPX, are more prominent when the compute load of the system exceeds the available resources. Therefore, using these runtimes for sections of a program which are not computationally intensive may result in little performance benefit. Moreover, the overhead of code transformation and inherent extraneous work imposed by runtime systems may even cause performance degradation. Therefore, we have opted to limit our optimizations to performance critical parts of the code which we call computational *kernels*. The Phylanx frontend provides a decorator (`@Phylanx`) to trigger transformation of kernels into the execution tree.

We have developed a custom internal representation of Python AST in order to facilitate the analysis of static optimizations and streamline the generation of the execution tree. The human-readable version of the AST, aka PhySL, is automatically generated and compiled into the execution tree by the frontend. More details on this compilation process can be found in IV-B. The benefit of using PhySL as the intermediate representation is twofold: (1) it closely reflects the nodes of the execution tree as each PhySL node represents a function that will be run by an HPX task during evaluation, and (2) it can be used for debugging and analyzing purposes for developers interested in custom optimizations.

The compiled kernel is cached and can be be invoked directly in Python or in other kernels.

2) Data Handling: Phylanx's data structures rely on the high-performance open-source C++ library Blaze [24], [25]. Blaze already supports HPX as a parallelization library backend and it perfectly maps its data to Python data structures. Each Python list is mapped to a C++ `vector` and 1-D and 2-D NumPy arrays are mapped to a Blaze vector and Blaze matrix respectively. To avoid data copies between Python and C++, we take advantage of Python buffer protocol through pybind11 library. Figure 1 shows how Phylanx manages interactions with external libraries.

B. Execution Tree

After the transformation phase, the frontend passes the generated AST to the Phylanx compiler to construct the execution tree where nodes are *primitives* and edges represent dependencies between parents and children pairs.

Primitives are the cornerstones of the Phylanx toolkit and building blocks of the Phylanx execution tree. Primitives are C++ objects which contain a single execute function. This function is wrapped in a `dataflow` and can be

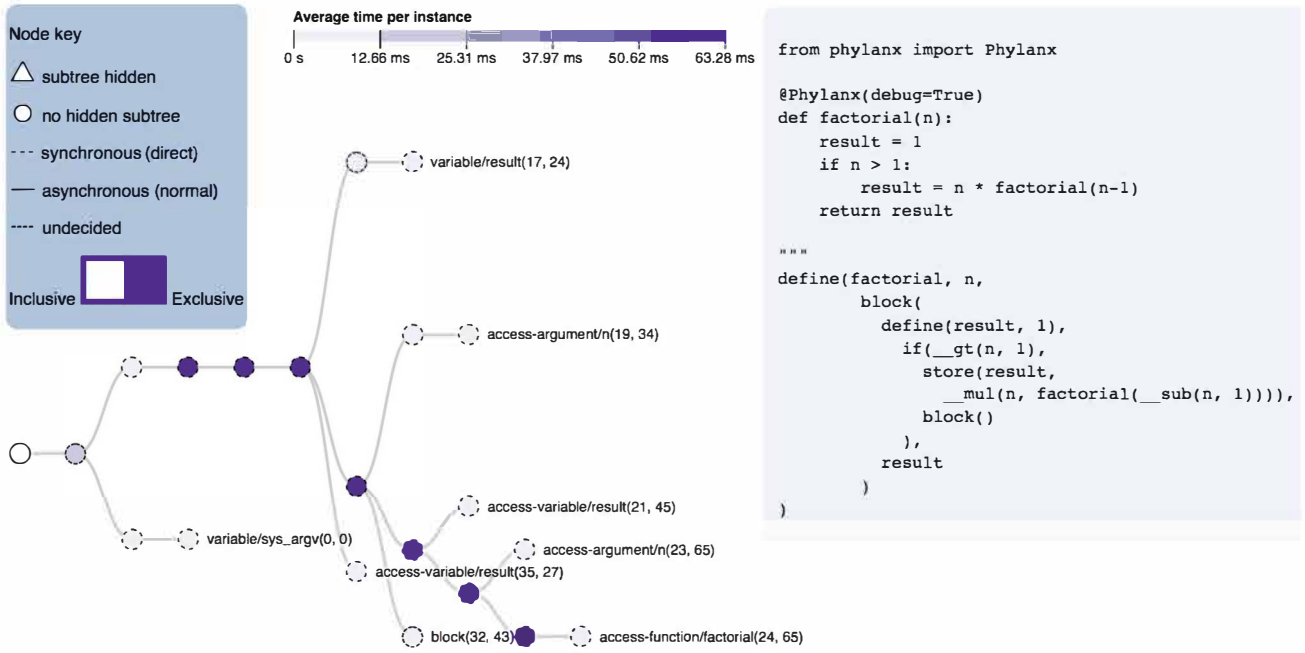

Fig. 2. Phylanx visualization tool provides a side-by-side view of the code and the corresponding expression tree along with performance information collected by builtin performance counters.

as simple as a single instruction or as complex as a sophisticated algorithm. We have implemented and optimized most Python constructs as well as many NumPy methods as primitives. Futurization and asynchronous execution of tasks are enabled through these constructs. One can consider primitives as lightweight tasks that are mapped to HPX threads. Each primitive accepts a list of futures as its arguments and returns the result of its wrapped function as a future. In this way, the primitive can accept both constant values known at compile time as well as the results of previous primitives known only after being computed.

C. Futurized Execution

Upon the invocation of a kernel, Phylanx triggers the evaluation function of the root node. This node represents the primitive corresponding to the result of the kernel. In the evaluation function, the root node will call the evaluation function of all of its children and those primitives will call the evaluation functions of their children. This process will continue until the the leaf nodes have been reached where the primitives evaluation functions do not depend on other primitives to be resolved (e.g. a primitive which is a constant, a primitive which reads from a file, etc.). It is important to note that it does not matter where each primitive is placed in a distributed system as HPX will resolve its location and properly call its eval function as well as return the primitive's result to the caller.

As the leaf primitives are reached and their values, held in futures, are returned to their parents the tree will unravel at the speed of the critical path through the tree. The results from each primitive satisfy one of the inputs of its parent node. After the root primitive finishes its execution, the result of the entire tree is then ready to be consumed by the calling function.

D. Instrumentation

Application performance analysis is a critical part of developing a parallel application. Phylanx enables performance analysis by providing performance counters to provide insight into its intrinsics. *Time* performance counters show the amount of time that is spent executing code in each subtree of the execution tree, and *count* performance counters show how many times an execution tree node is executed. This data aides in identifying performance hotspots and bottlenecks, which can either be directly used by the users or fed into APEX [26] for adaptive load balancing. The data can also be used by the visualization tools described in the next section.

E. Visualization

Embedding annotations and measurements for visualizations and performance analysis within the runtime provides a way to determine where performance bottlenecks are occurring and to gain insight into the resource management within the machines. We show an example of Phylanx's visualization capabilities in Figure 2. This tree shows the execution tree from a test run of the factorial algorithm, implemented in Python. In the tree, nodes are

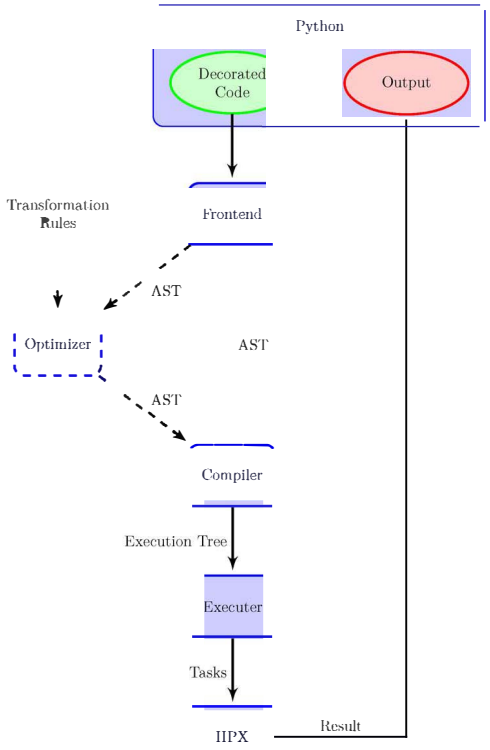

Fig. 3. Phylanx program flow. Phylanx frontend generates AST (PhySL) of the decorated Python code. The AST could be directly passed to the compiler to generate the execution tree or, optionally, fed to the optimizer first and then the compiler. Once the Kernel is invoked, Phylanx triggers the evaluation of the the execution tree on HPX. After finishing the evaluation, the result is returned in Python.

Phylanx primitives and edges show parent/child relationships regarding how the child was called. The nodes are colored purple for the *inclusive time*, the total time spent executing that primitive and its children. A switch in the toolbox in the upper left corner allows for the user to switch from inclusive time to *exclusive time*, the time spent executing only that primitive. This allows for identification of hotspots in the tree. Each primitive can be executed asynchronously or synchronously in the parent thread. This distinction is shown in dotted versus solid circles for nodes in the tree. The tree is interactive, allowing users to drill down and focus by expanding or collapsing tree nodes and hover for more details. The visualization is linked with a code view showing the Python source code (the corresponding PhySL is shown as well). Hovering over a node or line in one will highlight the corresponding line or node in the other.

V. Experiments

This section details the performance comparison of PhySL to a corresponding Python implementation utilizing multiple cores on top of NumPy using OpenBlas for BLAS/LAPACK routines. We used reference implementations of Alternating Least Squares [27] and Binary Logistic Regression [28] algorithms to analyze the performance of equivalent code written in PhySL. The Logistic Regression coupled with Alternating Least Squares provides a wide variety of computationally intensive operations which makes them useful for experimentation and are also used as benchmarks for the Intel MKL Library [29].

A. Experimental Testbed

We ran our experiments on LSU's Rostam cluster. These experiments were performed on a node consisting of Intel(R) Xeon(R) CPU E5-2660 v3 clocked at 2.6GHZ, with 10 cores (20 threads), and 128 GB DDR4 Memory. All Experiments were performed on HPX v1.2 commit 9182ac6182, Phylanx v0.1 commit 116c46a8 Python v3.5.1, NumPy v1.15.0 , OpenBlas v0.3.2 and Blaze v3.3.

B. LRA

We implemented the Binary Logistic Regression algorithm in Python and used the Phylanx decorator to generate the corresponding PhySL code. In order to test the performance of the two implementations of the Logistic regression algorithm, we created a custom binary classification dataset with 10,000 features and 10,000 observations.

Figure 5 shows the performance of the Python and PhySL codes in terms of the execution time. Our experiments show that on a single thread both PhySL and Python perform on par with each other. However, PhySL scales faster up to eight cores and plateaus afterwards while Python scales at a lower rate but up to 20 cores.

C. Alternating Least Squares

Alternating Least Squares is a method used in collabora tive filtering based on matrix factorization [27]. Collabora tive filtering as a recommender system is utilized to predict a user's interest in a set of items based on other users interaction with those items, and also the user's interactions with other items. In order to test the implementation of the Alternating Least Squares algorithm in Phylanx, we implemented the algorithm in Python using NumPy and generated the corresponding PhySL implementation using the Phylanx decorator . The two implementations of the algorithms were tested on MovieLens-20M dataset [30], which is a collection of 20 million ratings gathered by 138,000 users over 27,000 movies.

Figure 6 shows the execution times of the PhySL and Python versions of Alternating least squares. Both versions were run with number of factors set to 40 while the number of movies were set to 5,000, 10,000 and 20,000. The PhySL version of the ALS algorithm starts to outperform the Numpy/Python version as the number of threads increases. The fastest time for the Phylanx implementation is seen using 16 threads and the number of movies set to 20,000. When the number of movies were set to 10,000, the fastest time was seen using 12 threads. There

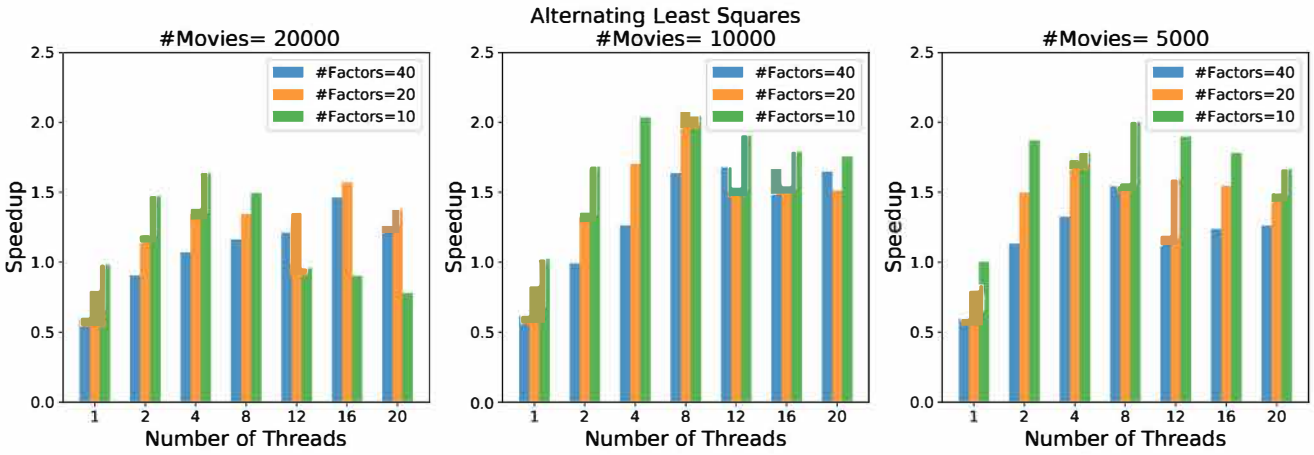

Fig. 4. Speedup of the PhySL implementation of the ALS algorithm over the Python implementation. Number of threads represents the number of OS threads used by both Phylanx and Python.

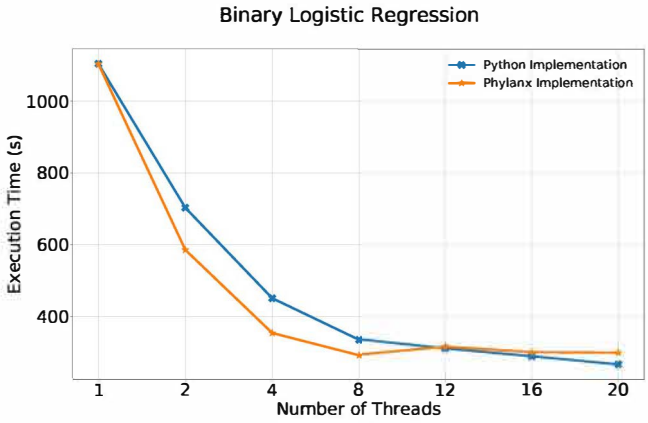

Fig. 5. Comparing execution time of the reference implementation of the Logistic Regression algorithm in Python with the corresponding PhySL code. Each datapoint represents the average execution time over ten runs.

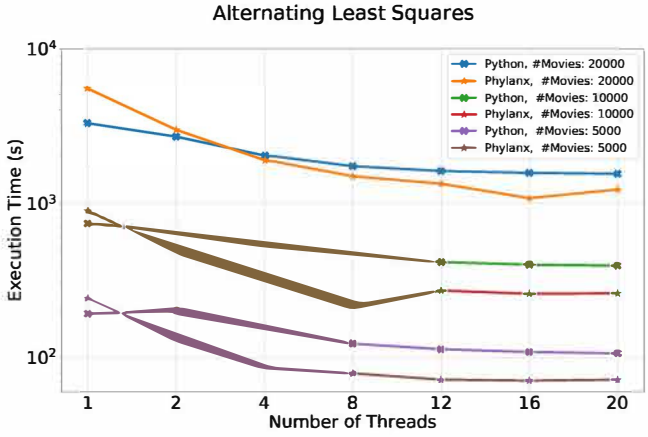

Fig. 6. Comparing execution time of the reference implementation of the Alternating Least Squares algorithm in Python and the corresponding PhySL code. Each datapoint represents the average execution time over ten runs.

is a noticeable difference in execution time between the Python and the PhySL implementation when the number of threads is set to one. In this configuration the Phylanx version is much slower than the Python implementation. Such behavior is not seen with the Logistic Regression example. This behavior is currently under investigation.

Figure 4 shows the speedup of the Phylanx implementation of the Alternating least square using the Python implementation's performance as the baseline. Both implementations were run with the number of factors set to 10, 20 and 40. while the number of movies were set to 5,000, 10,000, and 20,000. It is seen that the Phylanx implementation outperforms the Python implementation as the number of threads are increased on wide variety of problem sizes.

VI. Conclusion

Despite the solutions provided by current machine learning frameworks, better methodologies are needed to process the large amounts of data consumed by cutting edge machine learning applications in a timely manner. These new tools need to be accessible to the domain scientists who currently use them as well as efficient with the computational resources provided to them.

In this paper, we have introduced a novel approach for transformation and execution of high-productivity languages on top of the highly performant, low-level HPX runtime system. We have implemented our approach along with a suit of performance and visualization tools in the Phylanx array processing toolkit. Phylanx enables automatic generation of asynchronous task graphs from regular Python code and facilitates finer grain configurability. Our early experiments on representative applications and datasets demonstrate the performance benefits of our methods on a single node. However, we expect that the real benefits of our approach will manifest in a distributed runtime environment. Here the asynchronous execution and

locality abstractions provided by HPX stand to benefit users immensely by drastically decreasing execution times with little effort from the user.

VII. Future work

As the Phylanx technology matures we intend focus on two major goals: first, to improve the performance of single node runs and second to extend the framework to automatically run user supplied codes in distributed settings. Improving single node runs will entail implementing more basic algorithms utilized by the machine learning community and using that experience to improve the underlying toolkit. We anticipate that we will be able to uncover performance bugs as well as opportunities to improve the performance of our toolkit from this experience. One such opportunity is the support for hardware accelerators such as GPUs.

Phylanx plans to enable users to execute their existing Python code on clusters. This provides them with the ability to handle large data sets and achieve better scaling. While Phylanx is built with distributed runs in mind (execution trees can span across several nodes and evaluation is done using features from HPX), distributed runs will require extensions to the current toolkit to determine optimal data layout and data tiling given the algorithms provided by the user. We also intend to look at using code transformations to replace slower user-written algorithms with more efficient ones. While these goals present a research challenge, we believe that the support provided by the HPX runtime system will substantially reduce the barriers to providing distributed execution capabilities.

VIII. Acknowledgments

This work was funded by the NSF Phylanx project award #1737785. 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.

References

[1] C. Snijders, U. Matzat, and U.-D. Reips, "" big data": big gaps of knowledge in the field of internet science," *International Journal of Internet Science*, vol. 7, no. 1, pp. 1–5, 2012.

[2] A. D. Mauro, M. Greco, and M. Grimaldi, "A formal definition of big data based on its essential features," *Library Review*, vol. 65, no. 3, pp. 122–135, 2016. [Online]. Available: https://doi.org/10.1108/LR-06-2015-0061

[3] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of big data on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98 – 115, 2015.

[4] S. Sagiroglu and D. Sinanc, "Big data: A review," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, May 2013, pp. 42–47.

[5] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, "A survey of open source tools for machine learning with big data in the hadoop ecosystem," *Journal of Big Data*, vol. 2, no. 1, p. 24, Nov 2015.

[6] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha, "Efficient machine learning for big data: A review," *Big Data Research*, vol. 2, no. 3, pp. 87 – 93, 2015, big Data, Analytics, and High-Performance Computing.

[7] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[10] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "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*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11.

[11] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser, "HPX – An open source C++ Standard Library for Parallelism and Concurrency," in *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo'17)*, 2017, p. 5.

[12] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[13] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[14] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," 2017.

[15] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015, p. 7.

[16] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: http://dask.pydata.org

[17] W. McKinney, "pandas: a foundational python library for data analysis and statistics."

[18] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2. ACM, 1992, pp. 256–266.

[19] B. Wagle, S. Kellar, A. Serio, and H. Kaiser, "Methodology for adaptive active message coalescing in task based runtime systems," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 1133–1140.

[20] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 394–401.

[21] H. Kaiser, *HPX and C++ Dataflow*, 2015. [Online]. Available: http://stellar-group.org/2015/06/hpx-and-cpp-dataflow/

[22] H. Kaiser, T. Heller, D. Bourgeois, and D. Fey, "Higher-level parallelization for local and distributed asynchronous task-based programming," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM, 2015, pp. 29–37.

[23] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2016, https://github.com/pybind/pybind11.

[24] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, "Expression templates revisited: a performance analysis of current methodologies," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C42–C69, 2012.

[25] ——, "High performance smart expression template math libraries," in *2012 International Conference on High Performance Computing Simulation (HPCS)*, July 2012, pp. 367–373.

[26] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.
[27] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on.* Ieee, 2008, pp. 263–272.
[28] C. Bishop, "Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn," *Springer, New York*, 2007.
[29] A. Iyer and V. Saletore, "Accelerating apache spark mllib with intel math kernel library (intel mkl)," 2017, https://blog.cloudera.com/blog/2017/02/accelerating-apache-spark-mllib-with-intel-math-kernel-library-intel-mkl/.
[30] [Online]. Available: http://grouplens.org/datasets/movielens/

APPENDIX

A. Abstract

The appendix contains the information on how to build and run the experiments presented in the paper "Asynchronous Execution of Python Code on Task-Based Runtime Systems". We provide the list of compilers and libraries used to build HPX and Phylanx as well as instructions to build and run the experiments.

B. Description

1) Check-list (artifact meta information):

- **Program: HPX , Phylanx**
- **Compilation: GCC 8.1**
- **Data set: MovieLens, Custom Dataset**
- **Hardware: Intel(R) Xeon(R) CPU E5-2660, 128 G DDR4 Memory**
- **Experiment workflow:** Range of input sizes, number of threads
- **Publicly available?: Yes**

2) How software can be obtained:

HPX can be obtained from https://github.com/STEllAR-GROUP/hpx, and Phylanx from https://github.com/STEllAR-GROUP/phylanx

3) Software dependencies:

- HPX
- Blaze
- Pybind11
- hwloc
- OpenBlas
- NumPy

4) Datasets:

MovieLens dataset: https://grouplens.org/datasets/movielens/

Custom dataset: http://stellar.cct.lsu.edu/files/espm2_2018/custom_dataset_10kx10k.tar.gz

C. Installation

Please refer to Phylanx wiki for build instructions, found at: https://github.com/STEllAR-GROUP/phylanx/wiki/Build-Instructions

D. Experiment workflow

For Phylanx LRA, set the environment variable OMP_NUM_THREADS to 1 and Run lra_csv from the bin directory of phylanx with the following command line options

```
--hpx:threads=num_threads_you_want_to_run_with
--data_csv=/path/to/custom/dataset
--hpx:bind=balanced
--hpx:numa-sensitive
--n=10000
--row_stop=10000
--col_stop=10000
```

For Phylanx ALS, set the environment variable OMP_NUM_THREADS to 1 and Run als_csv_instrumented from the bin directory of phylanx with the following command line options

```
--hpx:threads=num_threads_you_want_to_run_with
--data_csv=/path/to/MovieLens/dataset
--hpx:bind=balanced
--hpx:numa-sensitive
--iterations=1
--f={40,20,10}
--row_stop=700
--col_stop={20000, 10000, 5000}
```

For Python LRA, set the environment variable OMP_PLACES to *cores* and OMP_NUM_THREADS to the number of threads you want to run the python example with. Run the python example by varying the number of threads for 10000 iterations.

For Python ALS, set the environment variable OMP_PLACES to *cores* and OMP_NUM_THREADS to the number of threads you want to run the python example with. Run the python example by varying the number of threads. Also vary the number of factors and number of movies. In the paper, experiments were performed with number of factors set to 10, 20 and 40 whereas the number of movies were set to 5000, 10000 and 20000.

E. Evaluation and expected result

The execution time obtained from both the Phylanx runs and the Python runs should be compared. The expected result should look like the one in the graphs presented in the paper. In order to test whether the output is correct, the last seven values printed by the alternating least square phylanx implementation for number of factors set to 40, and number of movies set to 20000 should be as follows:
-0.00167161, 0.000985893, -0.00264197, -0.00110344, 0.00372654, 0.00290297, 0.00105101
Similarly, the last seven values printed in case of Logistic Regression Phylanx implementation when run for 10000 iterations with the custom dataset should be as follows:
-1.19866, 1.32052, 0.0529683, -0.790137, -1.09337, -1.12403 , -1.30093