Distributed Asynchronous Array Computing with the JetLag Environment

Steven R. Brandt
Center for Computation & Technology
Louisiana State University
Baton Rouge, LA, USA
sbrandt@cct.lsu.edu

Bita Hasheminezhad
Center for Computation & Technology
Louisiana State University
Baton Rouge, LA, USA
bhashe1@lsu.edu

Nanmiao Wu
Center for Computation & Technology
Louisiana State University
Baton Rouge, LA, USA
wnanmi1@lsu.edu

Sayef Azad Sakin
Dept. of Computer Science
University of Arizona
Tucson, AZ, USA
sayefsakin@email.arizona.edu

Alex R. Bigelow
Dept. of Computer Science
University of Arizona
Tucson, AZ, USA
0000-0002-4593-2675

Katherine E. Isaacs
Dept. of Computer Science
University of Arizona
Tucson, AZ, USA
0000-0002-9947-928X

Kevin Huck
OACISS
University of Oregon
Eugene, OR, USA
0000-0001-7064-8417

Hartmut Kaiser
Center for Computation & Technology
Louisiana State University
Baton Rouge, LA, USA
0000-0002-8712-2806

***Abstract*—We describe JetLag, a Python-based environment that provides access to a distributed, interactive, asynchronous many-task (AMT) computing framework called Phylanx. This environment encompasses the entire computing process, from a Jupyter front-end for managing code and results to the collection and visualization of performance data.**

We use a Python decorator to access the abstract syntax tree of Python functions and transpile them into a set of C++ data structures which are then executed by the HPX runtime. The environment includes services for sending functions and their arguments to run as jobs on remote resources.

A set of Docker and Singularity containers are used to simplify the setup of the JetLag environment. The JetLag system is suitable for a variety of array computational tasks, including machine learning and exploratory data analysis.

I. Introduction

Interactive runtimes based on high-level languages such as Python, R, or Julia are becoming increasingly common, displacing traditional large, statically-compiled codebases. Typically, high-level languages provide user-friendly interfaces to the underlying, more complex statically-compiled codebases. Python is a simple, well-designed, object-based framework and mature libraries for interfacing with other languages (e.g. Pybind11 [1] which is used in this project) make it an obvious choice for creating a distributed array toolkit.

In the current context, we make use of Python's decorator functionality and reflection libraries to access the abstract syntax tree (AST) of functions. We transpile the AST to expose a subset of Python which we implement using a set of C++ data structures.

Notebooks are popular interfaces, enabling researchers to experiment or play with data. Notebooks are easily deployed both on laptops for smaller datasets or on remote systems with access to more computing power through interfaces like JupyterHub.

Yet another trend in these frameworks is to make and use services based on web-based frameworks. Toward this end, JetLag employs the RESTful interface known as Tapis or Agave [2] for remote job execution, and the web-based performance visualization framework known as Traveler to help users interpret performance data.

The JetLag project has been reported on previously [3], [4]. What is new in this work is the ability to run distributed code, to gather performance data from such code (including PAPI [5] counters) and to visualize said performance data. In addition, this work provides a deeper description of the Python interface used to access the underlying C++ code. Finally, we present results from a distributed alternating least squares (ALS) benchmark, and discuss tiling and other considerations for running distributed codes.

Phylanx is based on HPX, an open source C++ library for parallelism and concurrency [6]. HPX provides a high performance, cutting edge implementation of the C++ parallel standard.

II. Related work

There are a number of Python-based environments for distributed computing. GAiN [7] is one of the first distributed re-implementations of NumPy functionality. GAiN has overcome

978-0-7381-1086-8/20/$31.00 ©2020 IEEE
DOI 10.1109/PyHPC51966.2020.00011

the challenge of a single node memory constraints in array computing and is compatible with MPI.

Spartan [**?**] offers an automatic tiling on multidimensional arrays. It is a distributed lazy evaluation framework that converts user code to a graph of high-level operators like map, fold, and filter. Then it finds a solution considering the improvement in data locality.

Dask [9] is a flexible parallel computing framework for tasks and "Big Data," which is how they refer to their distributed array framework that is similar in form to numpy and pands.

PyCOMPSs [**?**] offers a sequential interface but is able to identify and exploit implicit parallelism on both clusters and clouds.

Legate Numpy [10] is a distributed array framework on top of the Legion runtime system that has native support for GPUs. Like Spartan, it can replace NumPy without any extra code or configuration changes.

NumPyWren [11] is a serverless linear algebra interface on top of PyWren that can execute its algorithms on AWS Lambda. Serverless computing cannot compete with high-performance clusters with their fast networks, however, using the cloud is easier than maintaining a cluster especially for a domain scientist. We studied these frameworks and many other similar ones to design JetLag.

Arkouda [12] is a distributed numpy-like array toolkit based on Chapel, an asynchronous many task language for HPC.

Numba [13], Pythran [14], and Cython [15] use decorators to access the abstract syntax tree and transpile into low level C or C++.

III. Components

A. HPX

HPX is a C++ standard library for distributed and parallel programming built on top of an AMT. It has been described in detail in other publications [16], [17], [18], [19], [20], [21], [22]. Such AMT runtimes provide a means for helping programming models to fully exploit available parallelism on complex emerging HPC architectures. The HPX runtime includes the following essential components:

- An ISO C++ standard-conforming API that enables wait-free asynchronous parallel programming, including *Futures*, *Channels*, and other primitives for asynchronous execution. The exposed API ensures syntactic and semantic equivalence of local and remote operations, which greatly simplifies writing complex applications [23], [24].
- A work-stealing lightweight task scheduler [25] that enables finer-grained parallelization and synchronization, exposes greatly reduced overheads related to threading, and ensures automatic load balancing across all local compute resources.
- HPX features an Active Global Address Space (AGAS) [19], [26] that supports load balancing via object migration, enables runtime-adaptive data placement, distributed garbage collection, and an active-message networking layer that enables running functions close to the objects they operate on [25], [27].
- APEX [28], an *in-situ* profiling and adaptive tuning framework that utilizes HPX's sophisticated performance counter framework [29].

In the context of the present work, we use HPX because of its full conformance to the recent C++ standards [30], [31], its extensive support for asynchronous and parallel computation, its distributed scheduling capabilities, and its sophisticated performance measurement and *in-situ* profiling capabilities provided by APEX (see Section III-D for more information).

B. Phylanx

Phylanx is a software framework that is based on the HPX runtime system, which in turn was designed from first principles to address well known key challenges of high-performance computing applications. As such, Phylanx implicitly and naturally benefits from inheriting the advantages of applying fine-grain parallelism, message driven computation, constraint-based synchronization (never synchronize more than needed for the local progress of execution), implicit overlapping computation with communication, and runtime-adaptive granularity control.

1) Phylanx and Python: There are a number of frameworks that use Python as a front-end for an underlying, higher performance framework, e.g. numpy [32], scipy [33], Spartan [34], Tensorflow [35], and Keras [36]. Many of these frameworks use operator overloading as a primary mechanism for accessing high performance code. Because of this, in the code snippet in Listing 1, Python will typically execute the `while` loop and orchestrate the calls to `transpose`, the multiply operator, etc. Often, the amount of time spent in Python is trivial compared to the overall execution.

```
while k < iterations:
    YtY = np.dot(np.transpose(Y), Y) +
        regularization * I_f
    XtX = np.dot(np.transpose(X), X) +
        regularization * I_f
```

Listing 1. Code Snippet

The Phylanx project, however, uses a decorator (i.e. `@Phylanx`) to enable transpiling of entire Python functions into C++ data structures. That means that, for the duration of the function's execution, the orchestration of the calculation, including the `while` loop itself, is carried out by C++. This choice comes with certain tradeoffs.

On the negative side, it means that inside the body of the function, certain Python features, functions, libraries, and capabilities will not be supported. The goal is not to provide a complete Python interpreter but to ensure that functionality relating to control flow and distributed array programming is present. However, this choice comes with the risk of creating semantic differences with regular Python code.

On the positive side, we have the opportunity for doing higher level optimization on a larger fraction of the code (though, at this stage of development, that is a future work). For those cases where it matters, less time is spent calling across the Python/C++ barrier. In addition, we have the capability of collecting performance data on a complete subroutine.

C. Distributed Computing in Phylanx

Many applications of Phylanx, especially in the Machine Learning (ML) and Deep Learning (DL) fields require computation on multidimensional arrays that do not always fit into the memory of a single node. This is the result of a common technique in ML/DL. When we require an explicit *for loop* to repeat an operation on multiple tensors, we add a new dimension to the tensor. For example, to train a neural network on a set of samples, we need to execute a learning algorithm repeatedly on all samples for the given number of epochs. Thus, we represent the corresponding input tensor with a sample dimension and one or more data dimensions. In the same way, parameters may contain a filter dimension that stacks different filters to train. Although the sample dimension usually is usually divided into mini-batches, the large-scale datasets necessitate adopting a distributed data parallelism algorithm to train deep neural networks. Therefore, a distributed representation of arrays is the next logical step.

To process multidimensional arrays on multiple nodes, Phylanx has defined the notion of distributed arrays using an attached annotation. Annotations represent the information about all partitions of the distributed array. The annotations are metadata about the array. They contain a unique name that is recognized by AGAS, a generation number (which counts the number of times the array was updated), the ID of the current locality, the number of localities the array is distributed across, and tiling of the array.

At this stage of development, Phylanx is a Single Instruction Multiple data (SIMD) project, and the *primitives* (our term for built-in functions) for working with distributed data reflect that programming style. Each node executes the same program on a different part of the distributed array and they communicate as necessary. To execute the program on a node, sometimes a part of the data that is on a remote node is required. In that case, the primitive fetches that partition using the information provided by the annotation. For instance, let us consider a naive implementation of vector-vector-multiplication on two localities. For the case that we have the first half of two vectors on node 1 and the second half on node 2, each node calculates the dot product on the local array partitions and an `all_reduce` operation generates the final result on both nodes. For the case that the second array has fewer elements on node 1, node 1 fetches the elements it needs from node 2 to evaluate its dot product result and the `all_reduce` operation finalizes the result afterward.

Distributed primitives and algorithms that are implemented in Phylanx can be categorized into four groups.

The first group contains distributed primitives that propagate the arguments to their non-distributed form. These primitives update the annotations for the calculated results. All element-wise operations are in this category; they do not need any communication to perform the operation and only the generation in their annotations needs to increment.

The primitives in the second group may use the non-distributed form of the operations but also require additional steps to finalize the result. The naive implementation of vector-vector-multiplication belongs to this group. A non-distributed vector-vector-multiplication is performed on each tile. The distributed operation includes an `all_reduce` and might include fetching on some nodes. Another example of primitives in the second group is *argmin*. In order to find the global index with the minimum value, we must keep the minimum value on each tile.

The third group constitutes distributed algorithms. Primitives of this group are optimized taking the communication between distributed tiles of the array into account. Take, for example, the Cannon product algorithm [37]. This algorithm is an iterative distributed implementation of matrix-matrix-multiplication. It determines the computation and communication of each iteration based on the number of tiles.

Finally, the last group consists of the generative primitives which create data. Distributed forms of *random*, *full*, *ones*, *zeros* and *file_read_csv* are examples of primitives in this group.

Some primitives perform better in the presence of certain tilings. For example, the Cannon product algorithm performs best if the matrices are block distributed. Phylanx benefits from a retiling operation that retiles an array to any arbitrary but consecutive tiles. The user can utilize the *retile* operation manually, feeding all tiles specifications, or they can provide one of the defined tiling types, e.g. *row*, *column*, *page* or *sym* which stands for symmetric tiling.

Implementing a data parallel method of training a Deep Neural Network (DNN) using the distributed and local arrays of Phylanx is straightforward. Training a DNN consists of three phases: the forward pass, the backward pass, and an optimization step to update parameters. In the data parallel approach, every worker (computational resource) has a replica of the entire DNN along with a partition of samples. In each iteration, samples in a mini-batch are distributed between the active workers. Each worker independently goes through two computation-intensive phases, the forward and backward passes. Then a collective operation aggregates the output from all workers and updates the parameters of the DNN on all workers. At the end of the iteration, the workers have a DNN with identical parameters. To implement this in Phylanx, we use local arrays for all the parameters and distributed arrays for other tensors.

```
@Phylanx
def initial_kernel(kernel_size, input_channels,
    output_channels, bound, seed):
    return random([kernel_size, input_channels,
        output_channels], ["uniform", -bound, bound
        ], seed=seed)

@Phylanx(debug=True)
def input_conv(csv_file_name, rows_per_page, kernel,
     padding):
    input_array = file_read_csv_d(csv_file_name,
        rows_per_page, tiling_type = "page")
    return conv1d_d(input_array, kernel, padding)
```

Listing 2. Example of a 1D convolution operation tiled on its sample dimension

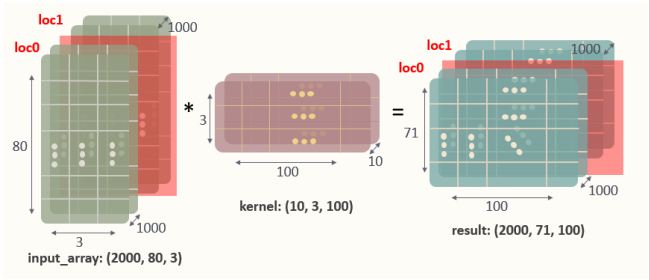

Fig. 1. Convolution of the input array and the kernel on two localities. The result is a distributed array with the same tiling as the input array

Here we show an example of the first iteration of a convolution operation in a DNN that is trained using the data parallel approach. Listing 2 demonstrates the very first convolution in a convolutional neural network that is to be trained in parallel with respect to samples. Here we assume the input consists of one-dimensional data. For instance, a human activity recognition dataset is a time series that is represented as a one-dimensional data. Along with samples and feature dimensions, the dataset constitutes a three-dimensional array that can be stored in a csv file. At the first iteration, parameters are initialized randomly using non-distributed arrays. As such, each locality has a copy of parameters that for this convolution layer is stored in `kernel`. The `read_csv_file_d` generates a distributed array where the `input_array` is tiled along its `page` dimension (sample dimension). The operation `conv1d_d` is called with a distributed array tiled on samples and a non-distributed kernel. It performs the index calculations and, utilizing `conv1d`, generates the distributed results. Therefore, the input to the next layer is also a distributed array tiled on its `page` dimension. As in this example we did not specify the number of nodes in either of `read_csv_file_d` or `conv1d_d` operations, the number of nodes sets to its default value which is the number of nodes that the runtime uses.

A representation of what the *input_array* contains is shown at Listing 3. Assume we run the program in Listing 2 on two nodes. The input csv file demonstrates each sample using 80 timestamps of one human activity type and three features. Here, activity types are jogging, walking, sitting, etc. and features are accelerations on the x-axis, y-axis and, z-axis. Having 2000 samples on each mini-batch, the input tensor should be of *(2000, 80, 3)* shape. Let us train 100 filters of length 10. Using a *valid* padding, the output tensor has a shape of *(2000, 71, 100)*. Since in Listing 2, we tile the *input_array* on its sample dimension, samples are divided between the two active nodes; so, each node trains the network for 1000 samples in each iteration. We illustrate this operation in Figure 1. Both nodes execute the program (here the *print* function) in Listing 3. On both nodes, annotations are identical except for the locality index.

```
>> print(input_array) # both localities print their
    arrays
[[[-0.7, 12.2, 0.5],[...], ..., [...]], ...,
    [[...],[...], ..., [...]]], annotation("
    localities",
["meta_0", ["tile", ["pages", 0, 1000], ["rows", 0,
    80], ["columns", 0, 3]]],
["meta_1", ["tile", ["pages", 1000, 2000], ["rows",
    0, 80], ["columns", 0, 3]]],
["locality", 0, 2],
["name", "HAR_csv_1/0"])

[[[0.3, 1.5, 13.0],[...], ..., [...]], ...,
    [[...],[...], ..., [...]]], annotation("
    localities",
["meta_0", ["tile", ["pages", 0, 1000], ["rows", 0,
    80], ["columns", 0, 3]]],
["meta_1", ["tile", ["pages", 1000, 2000], ["rows",
    0, 80], ["columns", 0, 3]]],
["locality", 1, 2],
["name", "HAR_csv_1/0"])
```

Listing 3. A demonstration of a distributed array

D. APEX

APEX [28] (Autonomic Performance Environment for Exascale) is a performance measurement library for distributed, asynchronous multitasking runtime systems such as HPX. It provides lightweight measurement while maintaining high concurrency. To support performance measurement in systems that employ user level threading, APEX can use 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 will start, stop, yield or resume timers for correct measurements. These timers can also capture hardware metrics using the PAPI [5] library. The asynchronous module does not rely on events, rather it periodically interrogates hardware, operating system and user-level runtime system counters. Beyond the scope of its role in this paper, APEX also includes a *policy engine* for using the performance measurements to reconfigure system, application or library parameters in an optimization or feedback-control capacity.

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 [38] (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.

In order to correctly write OTF2 traces, each process needs to know the number of processes involved and its own index within the group. HPX typically initializes APEX with the

current process' *locality* (rank index) and the total number of localities (distributed processes participating). HPX can be configured with many different *parcel ports* (e.g. MPI, TCP, etc.) to provide support for networking. APEX uses locality information from HPX to manage the recording of distributed performance data, so that each process in the distributed execution can write to its own unique file in a common location.

E. Traveler

Traveler [39] is a web-based visualization platform for parallel performance data. The goal of Traveler is to provide interactive access to performance data at multiple levels of abstraction, thereby enabling dynamic exploration of the data in familiar contexts. To this end, Traveler supports Gantt charts (trace data timelines with dependencies), source code, expression tree [40] and the recently added aggregated time series line charts for counter data and, task-level histograms.

Time views supported linked navigation—zooming or panning in one view will update the others. A utilization chart shows full run information and can be brushed to jump to a specific point.

All views support linked highlighting—selecting a specific primitive in one view will highlight the others of same type in the other views. Selecting points in a time series will highlight the related intervals in the Gantt chart.

We introduce a histogram that plots counts of a primitive by their duration, allowing investigators to view the distribution of task length. Durations of interest, such as exceedingly short tasks or long tasks can be selected via the histogram view and then further investigated in the other views. Figure 2 shows this feature in action.

The time series plots have been updated to show minimum, maximum, and average values across all hardware threads. The gray area in the short shows the standard deviation across hardware threads at each time point. Figure 3 shows an example of this chart. Though one chart is shown in this example and the previous, Traveler allows adding arbitrarily many to the view.

Traveler's Python server component ingests and processes OTF2 [41] trace files to provide time-dependent features, with HPX-specific counters packaged into OTF2 by APEX. Phylanx also ships its expression tree with profile data and source code correspondence, which is what Traveler uses for the expression tree and source code views.

IV. Performance

We test the performance of the distributed algorithm on Queen Bee 3 system maintained by LONI [42], where the information about the compute nodes is shown in Table I. Specifically, we test the distributed ALS (Alternating Least Squares) algorithm. For the source code, see Listing 4). We use a non-distributed ALS based on Numpy as a benchmark. Phylanx is built on HPX 1.5.0, master of Blaze, BlazeTensor, and Pybind11 as of Sep 2020. Both Phylanx and the Numpy version use openblas 0.3.10.

TABLE I
SPECIFICATION OF THE COMPUTE NODES OF THE QUEEN BEE SUPERCOMPUTER.

Processor type	Intel Cascade Lake Xeon 64bit
Processor	2 24-Core
Processor speed	2.4GHz
Memory	192 GB

First, we compare the time performance for distributed ALS (Phylanx) and non-distributed ALS (Numpy) on the MovieLens 20M dataset [43] for two runs. See Figure 4. The distributed ALS scales well as the increasing of number of nodes. Furthermore, we find that distributed Phylanx version on 1 node is better than the non-distributed Numpy version. To figure out the reason, we take a closer look at Figure 5, which compares their performance. It demonstrates that the time of Phylanx version is better able to exploit parallelism than the Numpy version.

In Fig.6, we test the speedup of distributed ALS on four different dataset sizes: which are 4800 x 4800, 9600 x 9600, 19200 x 9600, and 19200 x 19200 datasets, respectively. Firstly, it can be seen that distributed ALS scales well on all the datasets as the number of nodes increases. Secondly, we find that the larger dataset has better scalability than smaller dataset.

V. Future Work

Phylanx will be an ongoing target of development work, both in expanding the built-in functions and capabilities it has for distributed calculations, and in optimizing performance.

While Traveler's interface can manage data from multiple runs, allowing side-by-side comparison of charts, it does not yet support charts that combine data from multiple runs. Some of these charts may be better suited to be Jupyter-native, so we are exploring improving our API for managing data in this use case, as well as extending our in-interface support for multiple datasets.

Although the OTF2 trace format used by APEX has been sufficient for our needs so far, it has its drawbacks. All measurements are tied to specific operating system resources (processes, threads, devices), and the format doesn't flexibly allow for interleaved or asynchronous measurement. We are evaluating other trace formats and approaches that will enable time-series abstractions of performance data that are not tied to bulk synchronous performance models. We also aim to merge the dependency graphs collected by APEX and Phylanx into a unified task graph representing the algorithm as imagined by the Phylanx developer. In addition, APEX policies for Phylanx applications could be developed to help tune so-called "magic number" (heuristically set) algorithmic parameters such as group sizes and error bounds.

VI. Conclusion

We have described JetLag, a Python-based environment for working with distributed arrays. We have described the integration of this environment with performance data gathering

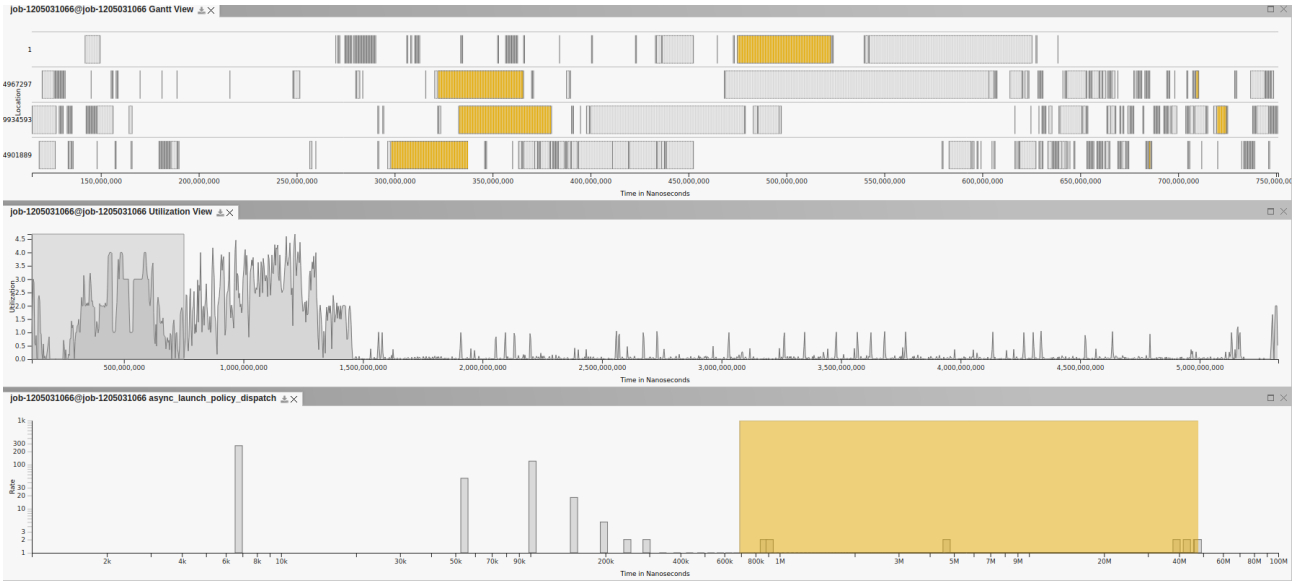

Fig. 2. Traveler interface showing performance data collected during a distributed run. The top window shows the task intervals in time across four locations. The middle window shows utilization over the run. The gray box indicates which portion is being shown in the other two windows. The bottom window is a histogram showing the distribution of durations of the async_launch_policy_dispatch task. The yellow rectangle selects durations in the histogram. The corresponding intervals are then colored yellow in the timeline.

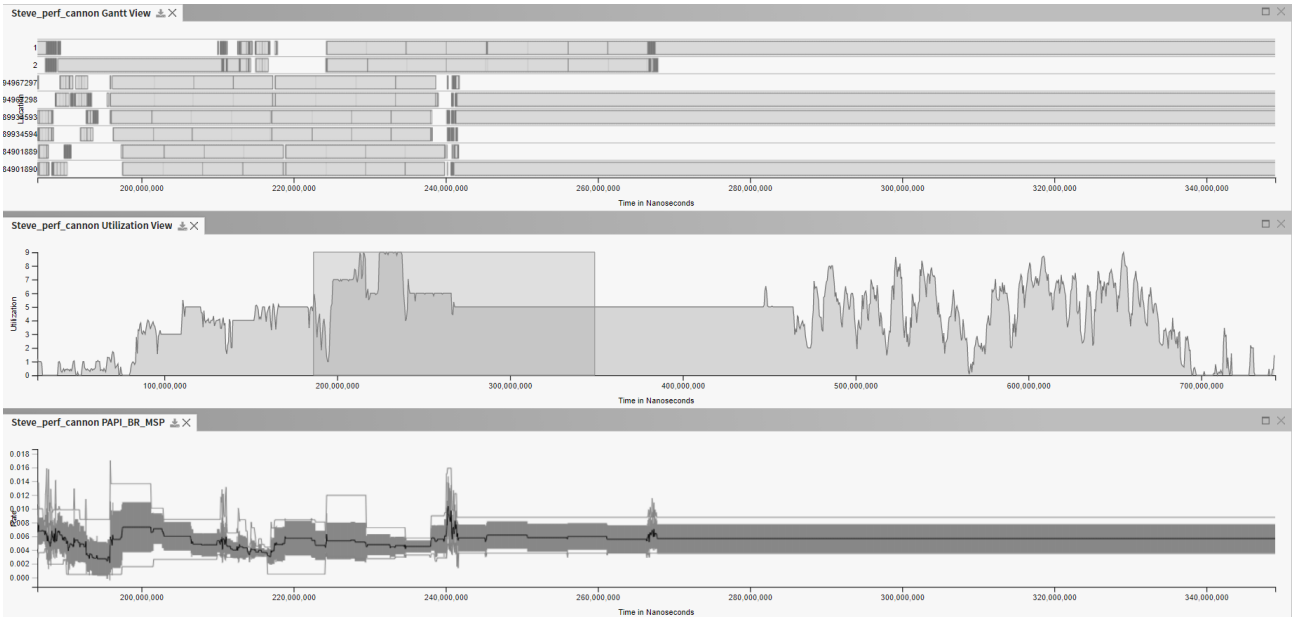

Fig. 3. Traveler interface showing performance data collected during a distributed run. The top window shows the task intervals in time across eight work threads. The middle window shows utilization over the run. The gray box indicates which time range is being shown in the other two windows. The bottom window is a time series plot showing the maximum, minimum, and average values of the PAPI branch misprediction counter as lines. The gray area in the time series shows standard deviation across all worker threads in time.

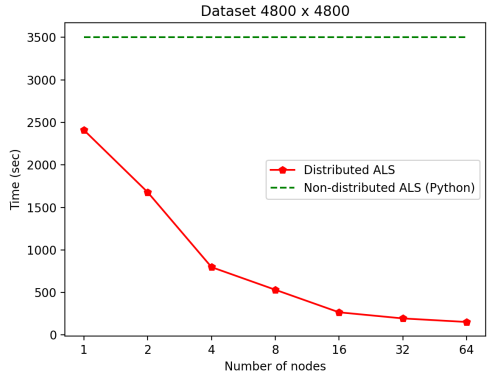

Fig. 4. Performance of different number of nodes on 4800 x 4800 dataset

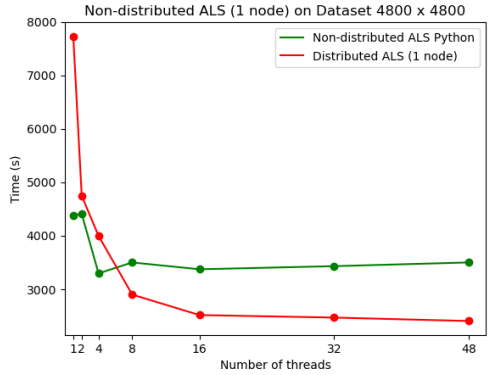

Fig. 5. Performance of different number of threads on 4800 x 4800 dataset

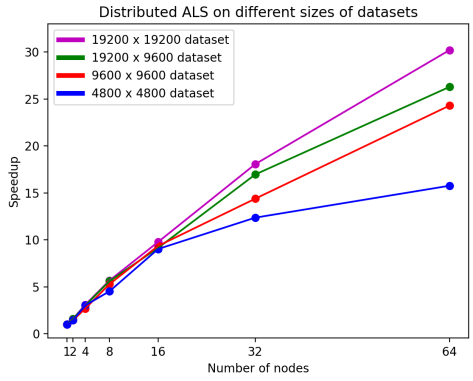

Fig. 6. Performance of distributed ALS on different sizes of datasets

and analysis tools and shown sample plots of the kinds of analysis that are possible with JetLag.

In addition, we have described the integration with Python and provided a few small examples of what kind of codes the environment can run (e.g. distributed ALS). We've also shown how our ALS implementation scales.

References

[1] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2017, https://github.com/pybind/pybind11.

[2] R. Dooley, S. R. Brandt, and J. Fonner, "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, 2018, p. 28.

[3] S. R. Brandt, A. Bigelow, S. A. Sakin, K. Williams, K. E. Isaacs, K. Huck, R. Tohid, B. Wagle, S. Shirzad, and H. Kaiser, "Jetlag: An interactive, asynchronous array computing environment," in *Practice and Experience in Advanced Research Computing*, 2020, pp. 8–12.

[4] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck *et al.*, "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, 2018, pp. 37–45.

[5] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[6] 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*, 2014, pp. 1–11.

[7] J. Daily and R. R. Lewis, "Using the global arrays toolkit to reimplement numpy for distributed computation," in *Proceedings of the 10th Python in Science Conference*, 2011.

[8] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao, "Spartan: A distributed array framework with smart tiling."

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

[10] M. Bauer and M. Garland, "Legate numpy: accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[11] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

[12] M. Merrill, W. Reus, and T. Neumann, "Arkouda: interactive data exploration backed by chapel," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, 2019, pp. 28–28.

[13] 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*, 2015, pp. 1–6.

[14] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, and A. Raynaud, "Pythran: Enabling static optimization of scientific python programs," *Computational Science & Discovery*, vol. 8, no. 1, p. 014001, 2015.

[15] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[16] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "Hpx - the c++ standard library for parallelism and concurrency," *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020. [Online]. Available: https://doi.org/10.21105/joss.02352

[17] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX Execution Model to Stencil-based Problems," in *Proceedings of the International Supercomputing Conference ISC'12, Hamburg, Germany*, 2012. [Online]. Available: http://stellar.cct.lsu.edu/pubs/isc2012.pdf

```
from phylanx import Phylanx

@Phylanx
def als_d(ratings_row,ratings_column, \
        regularization, num_factors, \
        iterations, alpha, enable_output):
    total_num_users = shape_d(ratings_row, 0)
    total_num_items = shape_d(ratings_row, 1)
    num_users = shape(ratings_row, 0)
    num_items = shape(ratings_column, 1)
    conf_row = alpha * ratings_row
    conf_column = alpha * ratings_column
    conf_u = constant(0.0, [total_num_items])
    conf_i = constant(0.0, [total_num_users])
    c_u = constant(0.0, [total_num_items,
        total_num_items])
    c_i = constant(0.0, [total_num_users,
        total_num_users])
    p_u = constant(0.0, [total_num_items])
    p_i = constant(0.0, [total_num_users])
    X_local = random_d([total_num_users, \
        num_factors], nil, nil, nil, "row")
    Y_local = random_d([total_num_items, \
        num_factors], nil, nil, nil, "row")
    X = all_gather_d(X_local)
    Y = all_gather_d(Y_local)
    I_f = identity(num_factors)
    I_i = identity(total_num_items)
    I_u = identity(total_num_users)
    k = 0
    i = 0
    u = 0
    XtX = dot(transpose(X), X) + \
        regularization * I_f
    YtY = dot(transpose(Y), Y) + \
        regularization * I_f
    A = constant(0.0, [num_factors, \
        num_factors])
    b = constant(0.0, [num_factors])
    while k < iterations:
        if enable_output:
            print("iteration", k)
            print("X:", X_local)
            print("Y:", Y_local)
        while u < num_users:
            conf_u = slice_row(conf_row, u)
            c_u = diag(conf_u)
            p_u = __ne(conf_u, 0.0, true)
            A = dot(dot(transpose(Y), c_u), Y) + YtY
            b = dot(dot(transpose(Y), c_u + I_i), \
                p_u)
            X_local[u] = dot(inverse(A), b)
            u = u + 1
        u = 0
        X = all_gather_d(X_local)
        XtX = dot(transpose(X), X) + \
            regularization * I_f
        while i < num_items:
            conf_i = slice_column(conf_column, i)
            c_i = diag(conf_i)
            p_i = __ne(conf_i, 0.0, true)
            A = dot(dot(transpose(X), c_i), X) + XtX
            b = dot(dot(transpose(X), c_i + I_u), \
                p_i)
            Y_local[i] = dot(inverse(A), b)
            i = i + 1
        i = 0
        Y = all_gather_d(Y_local)
        YtY = dot(transpose(Y), Y) + \
            regularization * I_f
        k = k + 1
    return (X,Y)
```

Listing 4. The Python source code for the distributed ALS implementation

[18] T. Heller, H. Kaiser, A. Schäfer, and D. Fey, "Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/2530268.2530269

[19] 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. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[20] 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*, ser. ESPM '15. New York, NY, USA: ACM, 2015, pp. 29–37. [Online]. Available: http://doi.acm.org/10.1145/2832241.2832244

[21] H. Kaiser, B. Adelstein-Lelbach, T. Heller, and A. B. et.al., "HPX V1.5: The C++ Standard Library for Parallelism and Concurrency," 2020, http://dx.doi.org/10.5281/zenodo.598202.

[22] T. Heller, H. Kaiser, P. Diehl, D. Fey, and M. A. Schweitzer, "Closing the Performance Gap with Modern C++," in *High Performance Computing*, ser. Lecture Notes in Computer Science, M. Taufer, B. Mohr, and J. M. Kunkel, Eds., vol. 9945. Springer International Publishing, 2016, pp. 18–31.

[23] T. Heller, B. A. Lelbach, K. A. Huck, J. Biddiscombe, P. Grubel, A. E. Koniges, M. Kretz, D. Marcello, D. Pfander, A. Serio *et al.*, "Harnessing Billions of Tasks for a Scalable Portable Hydrodynamic Simulation of the Merger of two Stars," *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 699–715, 2019.

[24] G. Daiß, P. Amini, J. Biddiscombe, P. Diehl, J. Frank, K. Huck, H. Kaiser, D. Marcello, D. Pfander, and D. Pfüger, "From Piz-Daint to the Stars: Simulation of Stellar Mergers using High-level abstractions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–37.

[25] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An Advanced Parallel Execution Model for Scaling-impaired Applications," in *2009 International Conference on Parallel Processing Workshops*. IEEE, 2009, pp. 394–401.

[26] P. Amini and H. Kaiser, "Assessing the Performance Impact of using an Active Global Address Space in HPX: A Case for AGAS," in *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, 2019, pp. 26–33.

[27] J. Biddiscombe, T. Heller, A. Bikineev, and H. Kaiser, "Zero Copy Serialization using RMA in the Distributed Task-Based HPX Runtime," in *14th International Conference on Applied Computing*. IADIS, International Association for Development of the Information Society, 2017.

[28] K. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomputing Frontiers and Innovations*, vol. 2, no. 3, 2015. [Online]. Available: https://superfri.org/superfri/article/view/64

[29] P. Grubel, H. Kaiser, J. Cook, and A. Serio, "The performance implication of task size for applications on the hpx runtime system," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 682–689.

[30] S. ISO/IEC, "ISO International Standard ISO/IEC 14882:2017(E) - Programming Language C++," *Geneva, Switzerland: International Organization for Standardization (ISO)*, 2017.

[31] ——, "ISO International Standard ISO/IEC 14882:2020(E) - Programming Language C++. [Working draft]," *Geneva, Switzerland: International Organization for Standardization (ISO)*, 2020.

[32] 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.

[33] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, "SciPy

1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.
[34] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao, "Spartan: A distributed array framework with smart tiling." in *USENIX Annual Technical Conference*, 2015, pp. 1–15.
[35] 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: http://tensorflow.org/
[36] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras
[37] H.-J. Lee, J. P. Robertson, and J. A. Fortes, "Generalized cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 44–51.
[38] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries." in *Advances in Parallel Computing*. Amsterdam, NL: IOS Press, 2012, vol. 22, pp. 481–490.
[39] S. A. Sakin, A. Bigelow, K. Williams, and K. E. Isaacs, "Traveler," https://github.com/hdc-arizona/traveler-integrated, 2020.
[40] K. Williams, A. Bigelow, and K. E. Isaacs, "Visualizing a moving target: A design study on task parallel programs in the presence of evolving data and concerns," *To appear in IEEE Transactions on Visualization and Computer Graphics (Proceedings of InfoVis '19)*, Jan. 2020.
[41] OTF2 developer community, "Open trace format version 2 (otf2)," Jul. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3356709
[42] "Louisiana optical network initiative." [Online]. Available: http://loni.org
[43] K. Abbas, "Movielens 20m dataset," 2018.