AFRL-RI-RS-TR-2017-155

WORKING WITH AND VISUALIZING BIG DATA EFFICIENTLY WITH PYTHON FOR THE DARPA XDATA PROGRAM

CONTINUUM ANALYTICS, INC.

AUGUST 2017

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2017-155 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
PETER A. JEDRYSIK
Work Unit Manager

/ S /
JULIE BRICHACEK
Chief, Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* AUGUST 2017	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* OCT 2012 – MAR 2017

4. TITLE AND SUBTITLE WORKING WITH AND VISUALIZING BIG DATA EFFICIENTLY WITH PYTHON FOR THE DARPA XDATA PROGRAM	**5a. CONTRACT NUMBER** FA8750-13-C-0033
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62702E
6. AUTHOR(S) Travis Oliphant, Peter Wang, Stan Seibert, Matthew Rocklin, Bryan Van de Ven, Hunt Sparra	**5d. PROJECT NUMBER** XDAT
	5e. TASK NUMBER A0
	5f. WORK UNIT NUMBER 21
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Continuum Analytics, Inc. 221 W. 6th St #1550 Austin, TX 78701	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RISB 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2017-155

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Research performed under the XDATA program focused on computational techniques and software tools for analyzing large volumes of data, both semi-structured (e.g. tabular, relational, categorical, meta-data) and unstructured (e.g. text, documents, message traffic). Several open source project which have seen community and industry adoption grew out of this effort.

- Blaze: A collection packages for describing and accessing, and manipulating disparate data sources and types
- Numba: A just-in-time function compiler for Python, based on LLVM compiler project allowing researchers to run their Python code near native speeds on CPUs and GPUs.
- Dask: Parallelizes generic Python and extends NumPy, Pandas, and Scikit-learn with parallel variants.
- Bokeh: Create interactive web applications from Python without having to know Javascript, CSS, or HTML.

15. SUBJECT TERMS
Python, Big Data, Visualization, Interactive, Cluster

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** PETER A. JEDRYSIK
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	43	**19b. TELEPHONE NUMBER** *(Include area code)* NA

Table of Contents

List of Figures

List of Tables

1 Summary

1.1 Purpose, Scope, and Organization

The objective of the effort performed was to support the DARPA XDATA program by developing computational techniques and software tools for analyzing large volumes of data, both semi-structured (e.g. tabular, relational, categorical, metadata) and unstructured (e.g. text, documents, message traffic). The scope included development, testing, and exercise support for technologies to advance the areas of scalable analytics and data processing technologies and visual user interface technologies. Focus was on being able to use an accessible language like Python, which domain experts can easily learn, for these technologies to enable large scale data science applications creation without requiring expert knowledge in the underlying technologies.

The Continuum effort was primarily composed of three teams aligned to the underlying projects.

- Blaze (later spawned Dask): Provide Python users with a familiar interface to query data living in a variety of other data storage systems. One Blaze query can work across data ranging from a CSV file to a distributed database.
- Numba: JIT compiler for NumPy-specific and generic Python allowing Python code to run at near native speeds on CPUs and GPUs.
- Bokeh (later spawned Datashader): Rich data-driven apps and plots in the web without requiring a user to know Javascript, CSS, or HTML.

1.2 Problem Under Investigation

The research performed under this program focused on the following problems facing researchers needing to access and process large amounts of data.

- Easy expression of computation kernels and data transformations on large datasets in order to allow analysts and scientists to productively write robust and efficient code, without getting bogged down in the details of how to distribute computation, or worse, how to transport and convert data between databases, formats, proprietary data warehouses, and other silos.
- A mechanism for referencing remote data sources, and seamlessly merge views of remote data with local data
- Shift modern, large scale computing from service oriented architectures that pull data around into different computing and business object silos into a data

oriented architecture where data is described in formats where it lives and moving computation to the data.
- Scaling Python up and out to support processing big data workloads so the many researches who know and use Python can easily take advantage of modern processors, GPUs, and cluster technologies.
- A method to allow creating interactive web applications without requiring knowledge and expertise of web technologies.

1.3 Principal Results and Conclusions

Development of techniques and tools for using a higher level, accessible language like Python for large scale data science applications is a fruitful area of research. As the research and software created under this program show it is possible to create libraries and tools for reading, processing, and creating interactive visualizations of big data workloads without requiring domain experts to learn other lower level languages such as C or C++. Several technologies were developed during the course of the program:

- **Bokeh** provides a means to create rich, interactive, data driven web applications without requiring the user to learn web-specific technologies such as javascript and CSS.
- **Numba** allows Python users to effectively use the processing power on a given machine and to speed up Python to speeds approaching Fortran and C and to run the code on CPUs and GPUs, including remote, with speedups of **2x** to **250x** over native Python or NumPy code.
- **Blaze** is an “interface” to data systems, somewhat like dplyr for R, for describing structured data, querying that data on various backends, moving data between formats, and remotely executing queries.
- **Dask** parallelizes Python using a distributed scheduler. To make it easier to adopt, Dask extends NumPy, Pandas, and Scikit-Learn with parallel variants, which allows users of those libraries to use Dask without having to learn a new API. Additionally, Dask parallelizes generic code without requiring the code’s author to deal with the intricacies of multithreading their data processing pipeline.
- **Datashader** is a companion package for Bokeh that renders arbitrarily large data into fixed-size images. Datashader provides the ability to interact with data in a visual manner for hundreds of millions of points on a laptop.

1.4 Recommendations

The team sees a number of areas for future research and development that would support processing, analyzing, and making decisions on ever increasing amounts and types of data. For Bokeh and Datashader, further work on native graph support, easy to use integration for streaming data, support for tiling and partitioning large data, and support for GPU dataframes,would be valuable for an existing and growing user base. For Blaze, Dask, and Numba further work on supporting new hardware technologies, support for complex machine learning algorithms, support for disparate data, and an easy extension mechanism could produce promising advances.

2 Introduction

Python is one of the most popular languages for scientific and data analysis, largely because of the existence of NumPy, SciPy, and the broad and diverse ecosystem of libraries and tools built on them. Not only has Python made large inroads into many traditional industries with heavy computational needs, such as Finance and Oil & Gas, it is also one of the most popular languages for web application development and system administration and monitoring.

The goal of this XDATA project was to create an accessible, expressive language for analysts and visualization designers to create novel ways of looking at complex data. Additionally, the project sought to generalize and extend NumPy, Python's extremely popular array library, to handle out-of-core computations on large data that exceed the system memory capacity, as well as distributed and streaming datasets.

This report covers the technology developed by Continuum Analytics under the XDATA program and is targeted toward managers and technical managers. Each of the key technologies will be covered divided by the areas and the current project name for each component. Introduction will touch on the high level technology and the Results and Discussion will provide more detail on the technology along with some of its uses to date. This report will end with some recommendations on areas that the team has identified as fruitful for further exploration and development.

2.1 Visualization

The principal innovation in Bokeh is the integration of scene-graph style construction with Grammar of Graphics style abstract specification. Scene graph style construction is effective at direct configuration and constructing reusable components. However, it is difficult to do more abstract manipulation, such as controlling coordinate spaces or presenting semantic transformations. Grammar of Graphics style languages essentially trade these attributes. They excel at high-level transformations, but are extremely awkward at detailed definitions and customization. Properly blending the two brings the power of both together. This enables abstract, high-level definitions (through GoG style declarations) with composition and customization that scene graphs provide.

Bokeh's render information feedback loop is a significant innovation. Providing rich information about the rendering status back to the analysis system will enable many interesting visualization tools. Bokeh is designed with extension and integration derived from the lessons learned around Stencil, D3, and Protovis. One significant lesson learned from these earlier projects is how to treat different types of data structures. In conjunction with the Blaze project, Bokeh includes integration with the specific characteristics of multiple underlying data structures.

The Datashader pipeline is another significant innovation, providing a way to build accurate visualizations and deliver them interactively without causing issues with local clients or narrowband remote connections. The result will allow web browsers to visualize data many orders of magnitude larger than would otherwise be possible, while still allowing detailed interactive control.

2.2 Out-of-core, and Beyond

Blaze extends NumPy's successful model of array-oriented programming to out-of-core and distributed data. It provides a generic n-dimensional array/table object, a very-general data-type descriptor for all kinds of data but especially semi-structured, sparse, and columnar data, and a generalized calculation engine that can iterate over the array and dispatch to low-level kernels selected via the dynamic data typing mechanism. This allows analysts and scientists to productively write robust and efficient code, without getting bogged down in the details of how to distribute computation, or worse, how to transport and convert data between databases, formats, proprietary data warehouses, and other silos.

All of the core functions in Blaze and its support system of numerical libraries manipulate this multidimensional array and build expression graphs behind the scenes as the user is writing familiar Python code. At evaluation time, these expression graphs are dynamically assembled to vectorized and optimized machine code via the Low Level Virtual Machine (LLVM) library via Numba. Such robust dynamic compilation (especially with auto-parallelization) is only possible because the Blaze array object is a sufficiently rich description of the data layout that allows fast, *a priori* reasoning about code dispatch and memory and disk access.

Additionally, and more importantly, Blaze provides a mechanism for referencing remote data sources, and seamlessly merging views of remote data with local data. Its compute graphs and dynamic scheduler and compiler will be able to reason about what computations to distribute to remote data, and what reductions can be used to minimize data movement. This ability to richly reference remote data in a generic way is analogous to how the URI scheme for HTML allowed the creation of unified hypertext documents that incorporated a broad set of hypermedia of any form. Likewise, Blaze arrays and tables can be composed of data compiled from disparate, remote sources. This mechanism forms the basis of a true "data web".

As data has become so large, moving the data to compute resources has become a very expensive operation. A goal of this work is to shift modern, large scale computing from service oriented architectures that pull data around into different computing and business object silos into a data oriented architecture where data is described in formats where it lives, and code is brought to the data. Dask parallelizes Python using a distributed scheduler and, along with Numba, provides a high-level interface for users which allows domain experts to execute their code at hardware accelerated speeds across multiple machines. This allows them to be productive in producing solutions without feeling like they are giving up performance.

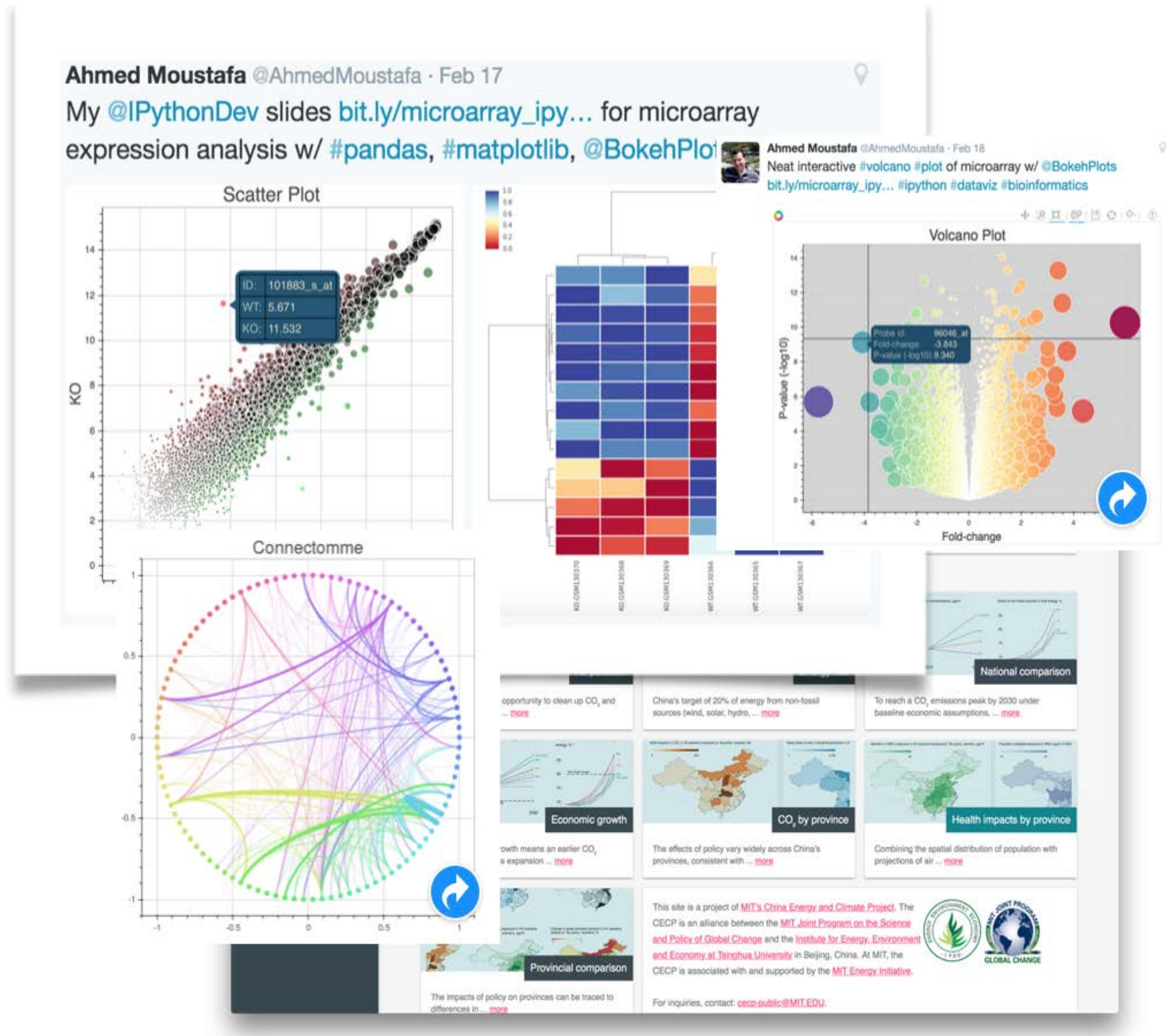
3 Methods, Assumptions, and Procedures

An open source development model was used for the research. All work was performed in open repositories with frequent builds. Following the open source development model team members presented and participated in some of the key conferences focused on Python and data science, including SciPy, PyCon, PyData, and Strata. Feedback from these conferences, as well as from users of the software, were used for real world feedback on direction. The usage by these open source consumers made it possible to use the technologies on very disparate workloads which helped uncover key features and performance bottlenecks to address. Additionally, participation in XDATA “hackathons” and “challenges” provided additional large scale problems that were used to identify additional features that would be beneficial to support. All work was tracked as issues in the respective GitHub repositories, all of which are public repositories.

- Blaze GitHub repository: https://github.com/blaze/blaze
- Numba GitHub repository: https://github.com/numba/numba
- Dask Github repository: https://github.com/dask/dask
- Bokeh Github repository: https://github.com/bokeh/bokeh
- Datashader Github Repository: https://github.com/bokeh/datashader

4 Results and Discussion

4.1 Bokeh


Ahmed Moustafa @AhmedMoustafa · Feb 17

My @IPythonDev slides bit.ly/microarray_ipy… for microarray expression analysis w/ #pandas, #matplotlib, @BokehPlot

Ahmed Moustafa @AhmedMoustafa · Feb 18

Neat interactive #volcano #plot of microarray w/ @BokehPlots bit.ly/microarray_ipy… #ipython #dataviz #bioinformatics

This site is a project of MIT's China Energy and Climate Project. The CECP is an alliance between the MIT Joint Program on the Science and Policy of Global Change and the Institute for Energy, Environment and Economy at Tsinghua University in Beijing, China. At MIT, the CECP is associated with and supported by the MIT Energy Initiative.

For inquiries, contact: cecp-public@MIT.EDU.

Figure 1 - Some Bokeh Graphs

Bokeh is a platform for creating visualizations and data applications that targets modern browsers for presentation. It provides a means to create rich, data driven web applications without requiring the user to learn web-specific technologies such as javascript and CSS. With Bokeh users can create interactive visualizations for their data, like Shiny does for R users, but now able to exploit the rich ecosystem of data-processing tools only available for Python. Higher level portions of Bokeh allow easy creation of plots and visualizations that can be customized. For truly unique visualizations, users can use lower level building blocks to create whatever they can envision. Bokeh supports streaming data and can easily create interactive, responsive plots with 100,000 points. Interactive Bokeh plots can be displayed with any modern browser and can be run with or without the Bokeh Server component. Bokeh Server works with Bokeh widgets to allow the user to perform actions, written in Python, based upon user input in the browser, such as selecting values from a dropdown that change the data being displayed or applying a different algorithm to the data. Some of the features and capabilities that Bokeh supports are:

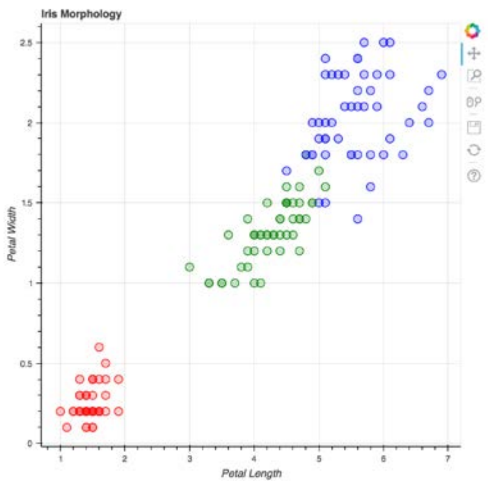
- Interactive visualization, widgets, and tools
- Versatile and high-level graphics
- Streaming, dynamic, large data
- For the browser, with or without a server
- Across multiple languages (Python, R)
- No JavaScript

4.1.1 Concentrate on Your Work

The main practical problem that Bokeh aims to solve for scientists, data scientists, and analysts is one of productivity. How to enable these groups to concentrate on the actual problems in front of them (instead of extraneous “web tech”) and stay productive with the tools and workflows they already have. In short, it aims to stay out of the way. Bokeh allows both standalone documents and server applications to be created and shared easily:

- Completely written in Python, no HTML or CSS or “webapp” coding
- Simple python scripts, no special classes of frameworks
- Useful for exploratory analysis or sharing and publishing
- Automatically mirrors and synchronizes Python and browser state
- Connect the full PyData stack to interactive web apps

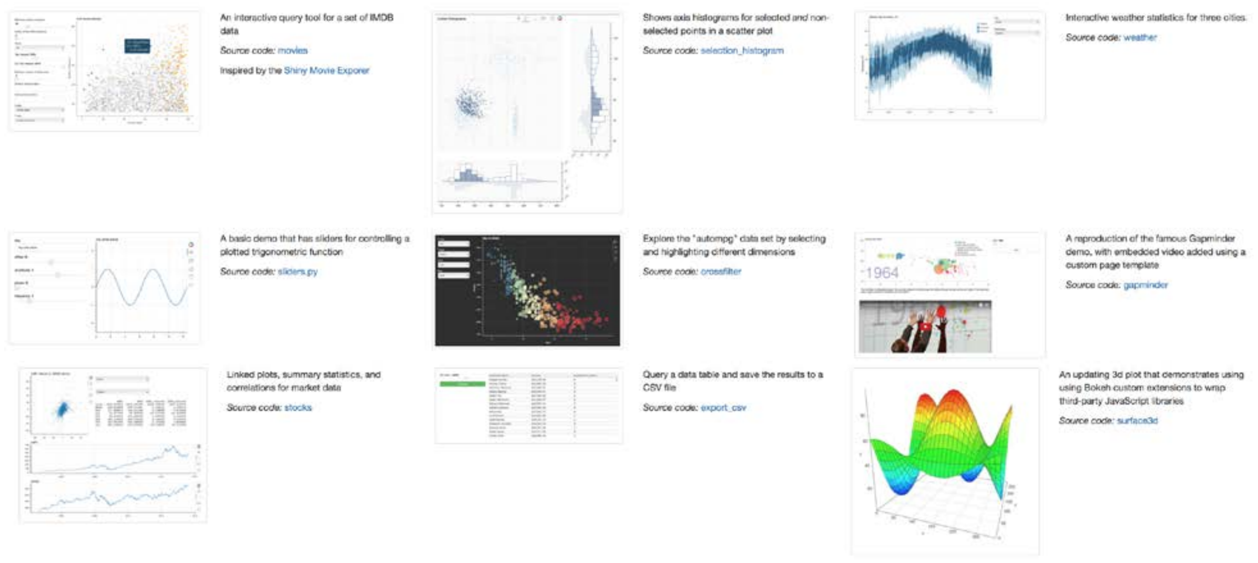
In the case of standalone documents (i.e. without a Bokeh server) simple python scripts can be used to generate plots easily. Even though these plots are “standalone’ or ‘static’ in the sense that they have no need for Python once generated, they can still have many interactive elements: tools for panning, zooming, hover tooltips for detail, and linked interactions. A small example is shown below:



```
from bokeh.plotting import figure, show, output_file
from bokeh.sampledata.iris import flowers

colormap = {'setosa': 'red', 'versicolor': 'green', 'virginica': 'blue'}
colors = [colormap[x] for x in flowers['species']]

p = figure(title = "Iris Morphology")
p.xaxis.axis_label = 'Petal Length'
p.yaxis.axis_label = 'Petal Width'

p.circle(flowers["petal_length"], flowers["petal_width"],
         color=colors, fill_alpha=0.2, size=10)

output_file("iris.html", title="iris.py example")

show(p)
```

Figure 2 - Bokeh Interactive Visualization

Bokeh also provides an optional server that can be used to develop rich and interactive data applications inside modern browsers using a live Python process. These applications can have all the usual tools available to standalone documents, but can additionally connect UI and tool events to real Python code. In this way the full ecosystem of Python data analytics packages (e.g. Pandas, scikit-learn, etc.) is available to drive and inform these applications in the browser. A small gallery of such apps can be seen at http://demo.bokehplots.com



An interactive query tool for a set of IMDB data

Source code: movies

Inspired by the Shiny Movie Explorer

Shows axis histograms for selected and non-selected points in a scatter plot

Source code: selection_histogram

Interactive weather statistics for three cities.

Source code: weather

A basic demo that has sliders for controlling a plotted trigonometric function

Source code: sliders.py

Explore the "autompg" data set by selecting and highlighting different dimensions

Source code: crossfilter

A reproduction of the famous Gapminder demo, with embedded video added using a custom page template

Source code: gapminder

Linked plots, summary statistics, and correlations for market data

Source code: stocks

Query a data table and save the results to a CSV file

Source code: export_csv

An updating 3d plot that demonstrates using using Bokeh custom extensions to wrap third-party JavaScript libraries

Source code: surface3d

Figure 3 - http://demo.bokehplots.com

Writing such apps does not require creating or learning any special classes or framework, and can often be accomplished in a few dozen lines of code. The full source code for the “sliders” demo is shown below:

```
import numpy as np
from bokeh.io import curdoc
from bokeh.layouts import row, widgetbox
from bokeh.models import ColumnDataSource, Slider, TextInput
from bokeh.plotting import figure

x = np.linspace(0, 4*np.pi, 200)
y = np.sin(x)
source = ColumnDataSource(data=dict(x=x, y=y))

plot = figure(title="my sine wave", tools="pan,reset,save,wheel_zoom",
              x_range=[0, 4*np.pi], y_range=[-2.5, 2.5])

plot.line('x', 'y', source=source, line_width=3, line_alpha=0.6)

def update_title(attrname, old, new):
    plot.title.text = text.value

text = TextInput(title="title", value='my sine wave')
text.on_change('value', update_title)

def update_data(attrname, old, new):
    a = amplitude.value
    b = offset.value
    w = phase.value
    k = freq.value

    x = np.linspace(0, 4*np.pi, 200)
    y = a*np.sin(k*x + w) + b

    source.data = dict(x=x, y=y)

offset = Slider(title="offset", value=0.0, start=-5.0, end=5.0, step=0.1)
amplitude = Slider(title="amplitude", value=1.0, start=-5.0, end=5.0)
phase = Slider(title="phase", value=0.0, start=0.0, end=2*np.pi)
freq = Slider(title="frequency", value=1.0, start=0.1, end=5.1)
for w in [offset, amplitude, phase, freq]:
    w.on_change('value', update_data)

inputs = widgetbox(text, offset, amplitude, phase, freq)

curdoc().add_root(row(inputs, plot, width=800))
```

Bokeh apps often follow this simple pattern: set up data and plots, add widget and callbacks, finally place things inside a layout for display.

4.1.2 Implementation

At a high level, Bokeh consists of a sophisticated Javascript library (named "BokehJS"), whose data models are designed to be reflected by parallel libraries in data analysis languages. These user-facing libraries, for instance Bokeh.py and rbokeh, are able to generate declarative JSON models of the data visualization, which then drives the BokehJS runtime to materialize the actual graphic in the HTML DOM of the web page. The Document class is a container for Bokeh Models to be reflected to the client side BokehJS library. This contains all the necessary information to render layouts and plots.

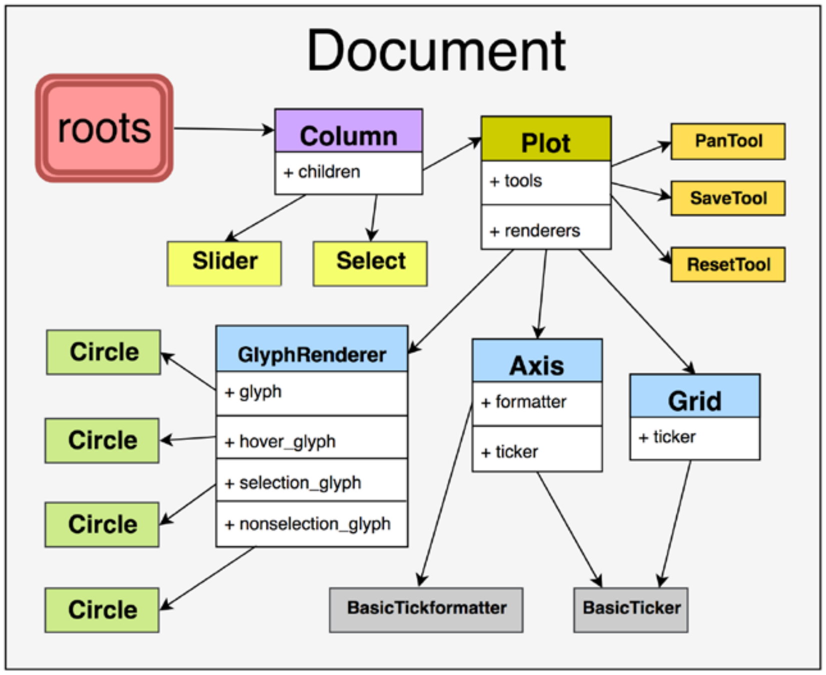


Figure 4 – Simplified Bokeh Document Representation

The optional Bokeh Server allows plots to be updated "live" from Python or R code, with changes reflected in the client-side web page. The Bokeh server automatically sends changes to the Document to the client browser to be rendered.

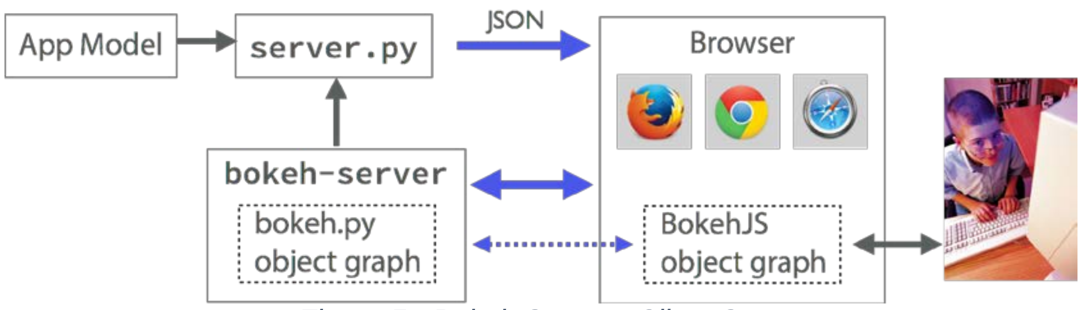


Figure 5 - Bokeh Server - Client Sync

When there are multiple clients connected, each client is given a unique Document instance. This prevents actions being performed by one client impacting the other clients.

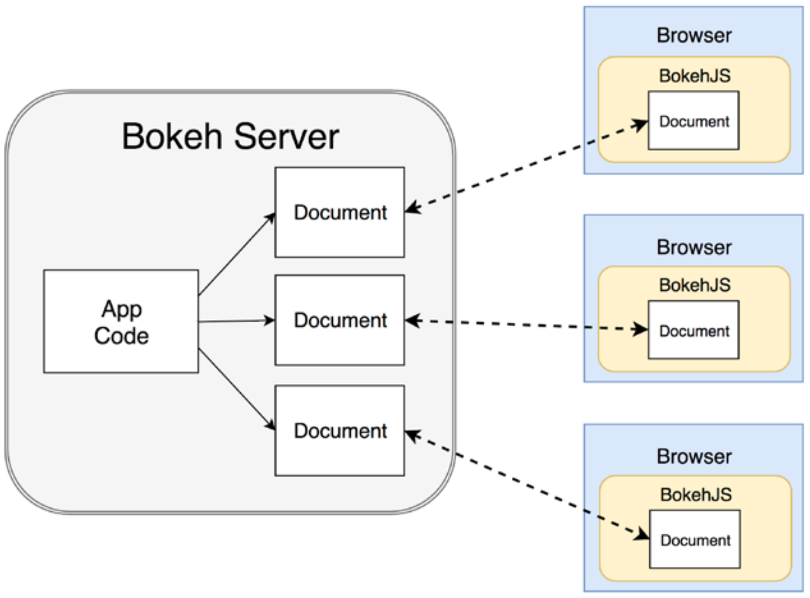


Figure 6 - Bokeh Server with Multiple Browsers

4.1.3 Lessons Learned

Over the course of developing Bokeh, three main lessons were realized:

4.1.3.1 Always Send Explicit Models

As mentioned above, Bokeh documents consist of collections of models that represent elements of a plot or application. There is a 1-1 correspondence between Python model objects and BokehJS model objects, expressed by a JSON representation that both sides can understand. Early in Bokeh development, there were instances where some BokehJS models did not have full Python counterparts. Instead, some ad-hoc collection of property values was mapped internally to a set of BokehJS objects. In every instance, this practice eventually proved problematic and limiting in some way. Now, the project maintains full 1-1 parity for all models.

4.1.3.2 Splitting Datashader

The original project vision for Bokeh included handling large datasets directly. This functionality was briefly and partially implemented in a sub-component called Abstract Rendering, whose code was coupled very tightly to the rest of the BokehJS codebase. This proved difficult to manage, as well as difficult to develop and test. Eventually this functionality was split off into the Datashader project (described separately). Separating things into two projects allowed for independent, de-coupled development with defined interfaces for integration. This has proved beneficial for both projects, leading to a simpler codebase for Bokeh and close integration between Datashader and the separate HoloViews projects, which allows Datashader to be used easily in both Bokeh and Matplotlib.

4.1.3.3 Migrating Server to Tornado

The original Bokeh server was implemented using Flask, and stored JSON representations of Bokeh documents in a Redis database. This architecture proved to be problematic in a number of ways. It did not scale well, as it required explicit replication to Redis. It was difficult to support across platforms, as Redis is not supported or easily available for Windows. Finally it did not perform well. The REST approach did not offer any real path towards streaming protocols, and the constant marshalling and unmarshalling to and from Redis added a great deal of overhead.

For Bokeh 0.11 a new second-generation Bokeh server was introduced. It was based on Tornado, which uses a pure websocket protocol. Documents for active sessions are simply stored in memory (as opposed to memory and a backing store) which makes the system much more horizontally scalable. Additionally the websocket protocol was more performant and also provided avenues for even further optimizations such as efficient binary encodings and streaming protocols.

4.1.4 Future Direction

Many of the original large-scale goals of the project have been attained, and we are looking forward to a 1.0 release in 2017. There is still a great deal of polish and bug-fixing yet to to do, but the list of larger tasks that are “must have” for a 1.0 release is now short:

- Migrate BokehJS to TypeScript

  BokehJS was originally written in CoffeeScript, which facilitated rapid development, especially by the initial core developers who largely had mostly Python experience. However, for long term stability and maintenance, TypeScript offers advantages that are especially useful in a cross-language, cross-runtime project, where accurate exchange of type information is crucial.

- Increase WebGL coverage

  Bokeh currently has limited WebGL support for a subset of glyphs. Extending this to include patches would allow Bokeh to be more useful for larger geographical and map type plots.

- Support for scripted animations and visual transitions

  Bokeh supports animation through server updates to data sources, including an efficient streaming protocol. However there are cases where users would like lighter weight solutions purely in the browser, or solutions that offer very smooth intermediate transitions between visual states.

- Static image (PNG) generation

  A long-requested feature, necessary for anyone wanting to include static images of Bokeh plots in presentations or spreadsheets or anywhere JavaScript execution is not an option. This work is currently funded and ongoing, based on Selenium, PhantomJS, and eventually Chrome Headless. It is scheduled for inclusion in a mid-2017 release.

- Network/Graph capabilities

  Another long-requested feature. This work is also funded and ongoing. The support will be split between Bokeh (for small graphs) and Datashader (for large graphs). The Bokeh component will see the addition of a new data source and renderer specific to network and graph data, with layout either supplied up front from the Python API, or computed in the browser dynamically.

4.2 Numba

Numba is a function-at-a-time, Just-in-Time (JIT) compiler for the standard Python interpreter (AKA “CPython”). Numba was spun out of the Blaze project into a separate effort in order to specifically tackle the difficult problem of compiling complex expressions and user-defined functions operating on NumPy arrays to efficient machine code. Frequently, developers would need to switch to other languages, such as C or FORTRAN, to achieve high performance on numerical algorithms. Numba enables nearly the same level of performance, and in many cases even higher performance, to be achieved with standard Python functions that have been explicitly designated for compilation. At runtime, Numba will compile the desired functions when they are first called, specializing the machine code for the specific data types used in the call. If the function is called again with a different set of input data types (as is quite common in a dynamically typed language like Python), Numba will compile another machine code implementation, and dispatch to the correct version on subsequent calls.

Numba is implemented as a library that can be loaded by programs running in the CPython interpreter and does not replace the interpreter itself. Its current focus is to target a Python subset that makes heavy use of NumPy arrays and numeric scalars in loops. Numba uses the type and memory layout information stored in the NumPy array header to generate specialized machine code for each operation. As Numba continues to develop, the supported subset of the Python language and standard library data types is expanding. Numba currently has support for the following Python language features:

- int, float, bool, complex, tuple, and enum types
- Support for lists and sets of a single element type
- Standard operators and control flow constructs
- Raising exceptions and asserts
- Generators
- Recursion (in most cases)
- Calling ctypes-wrapped C functions

- Random number generation

Numba was initially developed to optimize the inefficient use-cases of NumPy. This includes iterative functions that need to access individual array elements, which Numba can typically speed up by a factor of 100x or more. However, Numba can also translate a regular Python function into a special kind of function called a “universal function” in NumPy. Universal functions define an operation on scalar elements (or smaller dimensional arrays) that are implicitly *broadcast* over all elements of input arrays using standard rules. This allows the same function to be used with scalar inputs, input arrays of the same shape, or even input arrays of dimensionality in some cases. Most of the math functions in NumPy are in fact universal functions. Before Numba, the only way to create a fast universal function was to write it in some compiled language, like C. Due to the explicitly parallel nature of universal functions, Numba can also automatically generate multi-threaded or GPU-accelerated implementations of universal functions with very little user intervention.

One novel feature of Numba is its support for targeting different hardware. It currently provides an NVIDIA GPU back-end, and there is a experimental support for AMD GPUs as well. Rather than attempt to create a portable execution model supported by all targets, Numba directly exposes the execution model of each GPGPU architecture to Python. Users who wish to use the GPU (aside from the universal function support noted above) need to tailor their code to the specific features and performance characteristics of each GPU architecture. Numba also provides access to platform specific operations, such as thread barriers and atomic operations on GPGPU targets. This allows GPU-accelerated algorithms to be developed very rapidly within a Python application, while still achieving high performance in the generated GPU machine code.

4.2.1 Implementation

Unlike most JIT compilers for interpreted languages, Numba does not perform tracing nor replace the interpreter. Instead, it relies on the user actively transforming the Python functions that need compiling at runtime. In Python, this is done by applying a decorator to the function:

```
@jit
def example(a, b):
    acc = 0.0
    for a_i, b_i in zip(a, b):
        if a_i > b_i:
            acc += a_i
    return a_i
```

The decorator replaces the original Python function with a special object that just-in-time compiles the function when it is first called with a new type signature. To relieve user from the burden of explicit type annotation, Numba inspects the types of the arguments and performs local type inference on the function. As a result, the compiler can have accurate type information for each value in the function without tracing the execution.

The full compiler pipeline is shown below:

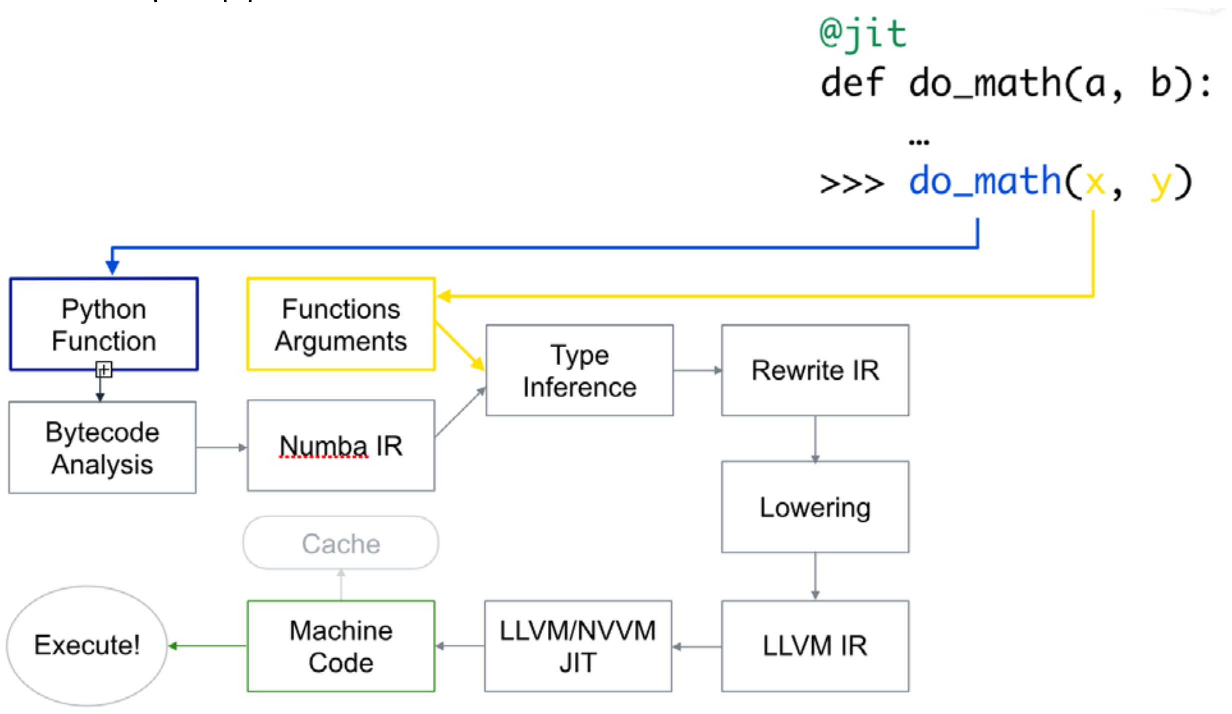


Figure 7 - Numba Compiler Pipeline

The bytecode of the input Python function is analyzed and translated to a Numba-specific internal representation ("Numba IR") which undergoes type inference. After type inference, certain language constructs, such as array expressions, are rewritten to simplify later compilation stages. After rewriting, the Numba IR is translated through a process called "lowering" to a low-level machine code form that is specialized for the actual data types present in the function.

The lowered form of the function is represented using the LLVM intermediate representation language ("LLVM IR"). LLVM is a very popular open source compiler framework with broad industry support, including Intel, AMD, IBM, NVIDIA, Apple, and many other companies. As a result, LLVM can generate machine code for a wide range of CPU and GPU architectures, which allows Numba to support many platforms. The Numba-generated LLVM IR is just-in-time translated to machine code, cached, and executed. The complexity of the compilation process means that the first execution of a function is fairly slow, but subsequent execution are extremely fast. Users can mitigate the compilation time using some of the caching or ahead-of-time compilation features in Numba.

Numba's support for targeting both CPU and GPU hardware is novel among Python compilation projects. Numba currently provides a NVIDIA CUDA GPU back-end using the NVVM library, and an AMD GPU backend using the ROCm libraries. Both NVVM and ROCm provide vendor-specific versions of LLVM with additional support for their hardware.

4.2.2 Lessons Learned

Python is a challenging language to compile due to its dynamic nature. The data type associated with a variable can change during execution, functions can freely operate on any data type, and functions and types themselves can even be modified during execution. This flexibility is what makes Python a very productive language, but in practice most code within an application does not take advantage of all the dynamic properties of Python. For example, a given function could operate on any data type, but in a particular application might only operate on 1 dimensional floating point arrays. This is the situation where type-specializing compilation, as provided by Numba, can be hugely beneficial. In addition, these core algorithms tend to be the sections of a program where most execution time is spent, and therefore compilation will improve the application's runtime the most if applied there.

As a result of these observations, Numba's current design is very conservative, compared to other projects. Numba compilation is *opt-in*; the user must indicate which functions should be compiled, and users are encouraged to only target functions which constitute the bulk of the runtime. Numba also will only generate efficient type-specialized code if the entire function can be compiled this way. For greater predictability, the current Numba releases will not flip between unoptimized and optimized code within the same function, as previous versions of Numba did. Switching in and out of the optimized code generation mode within a function made the compiler very complex, and resulted in some cases with code that ran slower than the original, uncompiled version. For scientific use cases, it was decided that it was much better for Numba to have a narrower scope of supported Python language features but to be able to always generate efficient code for functions that fell within that scope. Future versions of Numba may allow greater mixing of optimized and unoptimized code paths, but for now the more limited scope has helped users achieve predictable results.

In addition, by limiting Numba's scope to primarily numerical algorithms and basic language constructs, Numba was able to be ported to both CPU and GPU targets much more easily. It is very difficult to port a compiler based on tracing, as is done in many Javascript runtimes, as well as the PyPy JIT compiler for Python, to support compilation for coprocessor-type hardware like a GPU.

4.3 Blaze

The Blaze project is an ecosystem of packages that help users describe, transform, and query data. Python has become an extremely popular language for data science due to its ease with which a user can combine different software packages into an analysis. Blaze strives to improve that capability, but for combining different data sources. External constraints (and performance requirements) seldom allow all data in an organization to be centralized into single data store with a single computational API. Instead, data scientists will frequently have to move between systems, local and remote, SQL and non-SQL based. Blaze introduces interfaces for data types and expressions on data that simplify moving between data stores. In addition, Blaze includes an array server that serves up data from any of the data store backends Blaze understands to clients using a single JSON-based web API.

Blaze began with the goal of making NumPy & Pandas more scalable. Tackling this problem requires a multipronged approach, with a strong need for separation of concerns. In many ways, Blaze has been successful as an incubator and inspiration for a wide range of spinoff projects:

- Numba (for compilation of Python expressions)
- Dask (for distributed and out-of-core computing)
- Datashape (for portable description of data)
- Odo (for easy translation between data formats)

Numba and Dask are described in other sections of this document, so this section will focus on Datashape, Blaze and Odo.

4.3.1 Datashape

Datashape is a grammar for describing array-like data in-situ, without requiring data to be translated into a single canonical form. Ideally, any data source should be able to describe its contents using datashape, which enables the rest of the Blaze ecosystem to understand the field names, field types, nested structures, array dimensions and array shapes of a data set. The grammar is designed to be human readable, though in most cases it is machine-generated.

An example datashape describing the records for 100 people would look like:

```
100 * {
    name: string,
    birthday: date,
    address: {
        street: string,
        city: string,
        postalcode: string,
        country: string
    }
}
```

This demonstrates how datashape can represent nested structures, such as address.street within arrays. Multidimensional arrays are represented by separating multiple shapes with the asterisk symbol, such as in:

```
5 * 10 * 20 * int
```

which describes a 3 dimensional integer array (5 by 10 by 20). Datashape can express all of the following type concepts:

- Scalar types: fixed size signed and unsigned integers, floats, complex numbers and booleans
- Unicode and byte strings, dates, times
- Optional types (value could be null or None)
- Records (collections of named fields)
- Multidimensional arrays of fixed shape, or ragged arrays

4.3.2 Blaze Core

The Blaze package provides a data() constructor that takes a URI description of a data source (such as a filename, like `iris.csv`, or a URI for a remote service, like `postgresql://username:password@hostname:port`), and returns a Blaze data object. Blaze has support for a wide variety of backends including CSV, SQL databases, AWS S3, AWS Redshift, JSON, Redshift, HDF5, Hadoop, Hive, Spark, MongoDB, and SAS.

This object can then be used in Blaze expressions, which will filter, group, and transform the data to a result in memory when executed. Blaze expressions are *lazy*, which means they do not evaluate until the entire computation has been described.

This allows the Blaze backends to move the computation to execute on the data source (as in the case of a SQL database), which is typically much more efficient than bringing all the data to the client and performing the operations locally. Blaze Expressions can include operations like:

- Projection: extract a subset of columns
- Selection: extract a subset of rows
- Arithmetic: math operations on values
- Split-apply-combine: similar to group-by operations in databases
- Join: combine two datasets using common columns

4.3.3 Blaze Server

The Blaze Server is a lightweight web application that exposes Blaze data objects to clients using a JSON-based protocol. This allows an administrator to create a data catalog out of a possibly heterogeneous collection of data sources.   For example, a single Blaze server could serve up a mixture of CSV files, SQL databases, and MongoDB databases using a common interface. In addition, any client computation done on Blaze data sources without built-in compute capabilities (such as plain data files) will be performed on the server, rather than on the client. This reduces the compute requirements of the client dramatically.

4.3.4 Odo

Odo is a subproject of Blaze allows data to be easily transformed from one format to another, or one data storage server to another. Internally, odo takes a set of known conversion operations (such as CSV->Pandas Dataframe) and builds a conversion graph:

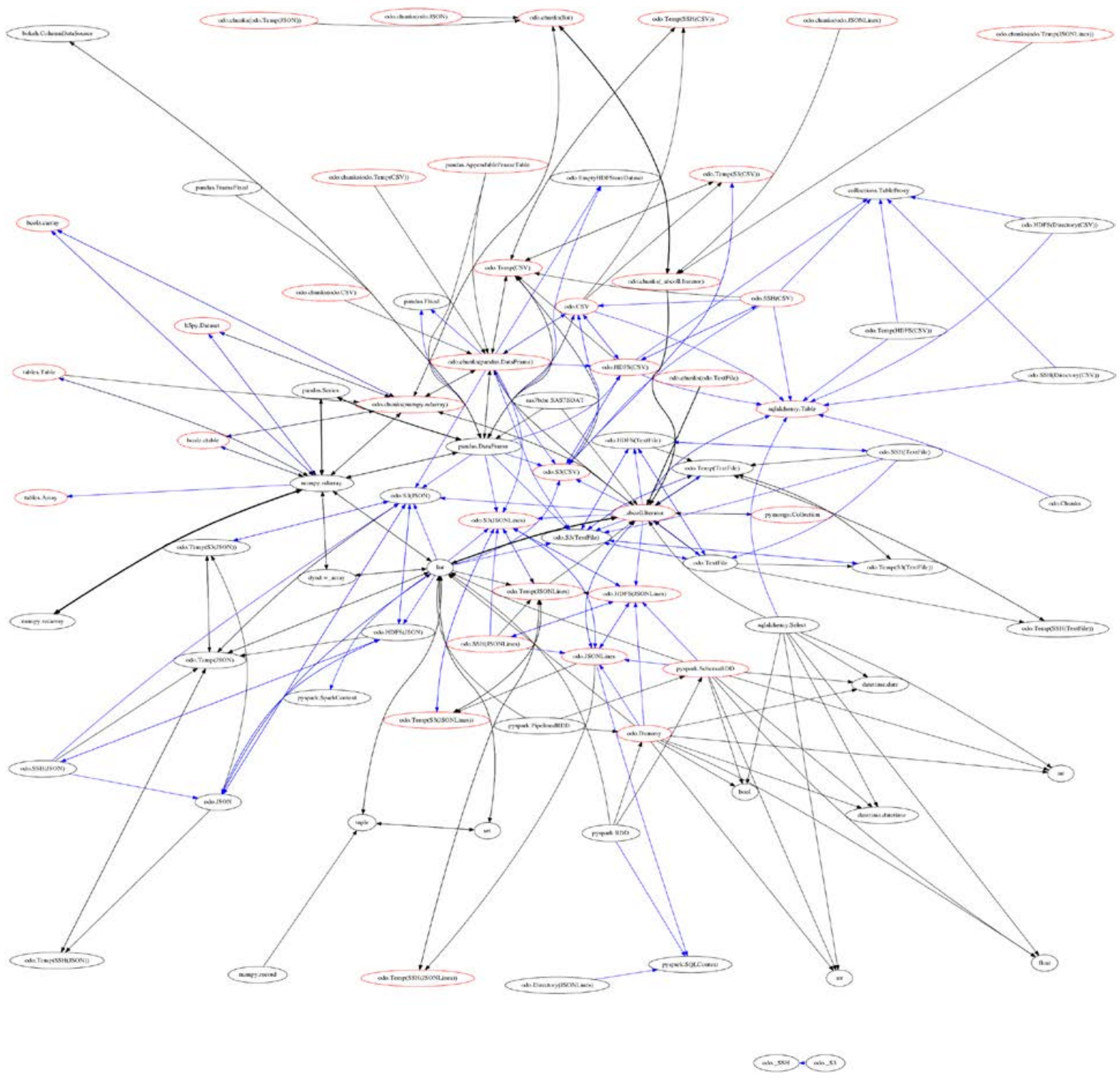


Figure 8 - ODO Format Conversion Graph

Then Odo can transform any format to any other format by finding the least expensive path through the conversion graph connecting the two formats together, even if no direct conversion method is known.

4.4 Dask

Dask grew out of Blaze development. When developing a unified front-end for array computing it became clear that out-of-core and distributed multi-dimensional array backends were lacking. Dask started as an out-of-core multi-dimensional NumPy library. Multi-dimensional array algorithms were more complex than traditional

MapReduce-style algorithms and so we had to build a more general purpose task scheduler. It quickly became apparent that this task scheduler was useful for more applications than just arrays, so parallel dataframes and generic APIs for parallel computing quickly followed.

Experiences with Blaze encouraged us to keep the scope of Dask small and to remove barriers to adoption like a new user API, exotic dependencies, and new type systems. From a user's perspective, Dask introduces very few new concepts and so is easy to integrate into existing workloads without significant proselytizing on our part. It copies existing APIs from NumPy, Pandas, Scikit-learn, concurrent.futures, etc. and uses existing data structures. This allowed a user community to come online within a short time and start valuable feedback cycles which improved the design and led to unforeseen advantages.

Dask enables parallel computing for Python libraries. It has been used both to provide parallel variants of popular libraries like NumPy and Pandas and to build completely new parallelized libraries and solutions. It is in wide use today within the PyData open source ecosystem, empowering data scientists from a number of disciplines.

Dask is composed of two components:

1. At its core, Dask is a dynamic task scheduler optimized for computational workloads. Dask runs small tasks / functions on data on parallel hardware. It tracks dependencies between these tasks and moves data around as necessary. Dask schedulers exist for single multi-core workstation machines as well as moderately large distributed clusters.
2. Algorithms for Big Data collections like parallel arrays and dataframes that extend common interfaces like NumPy and Pandas to larger-than-memory or distributed environments. These parallel collections run on top of the dynamic task schedulers.

4.4.1 Dask Collections

4.4.1.1 Arrays

Dask Array implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays. This lets us compute on arrays larger than memory using all of our cores. Dask arrays

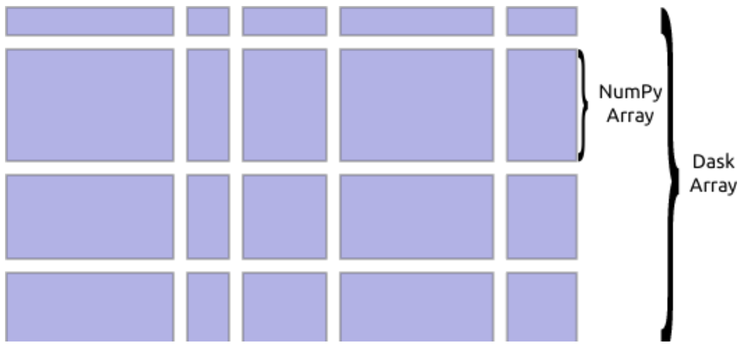


Figure 9 - Dask Array

coordinate many NumPy arrays arranged into a grid. These NumPy arrays may live on disk or on other machines.
Today Dask array is commonly used in the sort of gridded data analysis that arises in weather, climate modeling, or oceanography, especially when data sizes become inconveniently large. Dask array complements large on-disk array stores like HDF5, NetCDF, and BColz. Additionally Dask array is commonly used to speed up expensive in-memory computations using multiple cores, such as you might find in image analysis or statistical and machine learning applications.

4.4.1.2 DataFrames

Dask Dataframe implements a subset of the Pandas Dataframe interface using blocked algorithms, cutting up the large DataFrame into many small Pandas DataFrames. This lets us compute on dataframes that are larger than memory using all of our cores or on many dataframes spread across a cluster. One operation on a dask.dataframe triggers many operations on the constituent Pandas dataframes. Dask dataframes coordinate many Pandas DataFrames/Series arranged along the index. Dask.dataframe is partitioned row-wise, grouping rows by index value for efficiency. These Pandas objects may live on disk or on other machines.

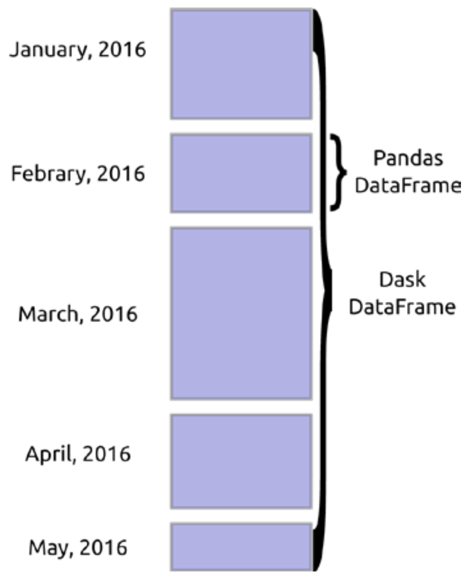


Figure 10 - Dask Data Frame

4.4.2 Dask Task Schedulers

The Dask collections (arrays, dataframes, etc.) encode task graphs. These are collections of functions linked with dependencies that may be executed in parallel. The outputs of some functions may be used as the inputs to others. It is the job of a task scheduler to take such a task graph and some parallel hardware, either a multi-core machine or a multi-machine cluster, and run that task graph efficiently on that hardware, taking into account dependencies, data locality, communication costs, expected runtimes, failed computers, failed tasks, etc.

4.4.2.1 Single-machine

Dask's original task scheduler was optimized for single machines, either personal laptops or large workstations. This scheduler focuses on running the computation in a small memory footprint, so that we run tasks that allow us to quickly release intermediate results. This proved to be most valuable to the communities that were using Dask at the time (climate science, geospatial, data science, etc.) because it let them analyze 100+GB datasets comfortably from their personal laptop. Additionally the single machine scheduler was used to accelerate computations on large workstations with a large number of cores. Speedups of 10x were common on computationally bound Pandas computations running on larger workstations.

4.4.2.2 Distributed

Dask.distributed is a centrally managed, distributed, dynamic task scheduler. The central dask-scheduler process coordinates the actions of several dask-worker processes spread across multiple machines and the concurrent requests of several clients. The scheduler is asynchronous and event driven, simultaneously responding to requests for computation from multiple clients and tracking the progress of multiple workers. The event-driven and asynchronous nature makes it flexible to concurrently handle a variety of workloads coming from multiple users at the same time while also handling a fluid worker population with failures and additions. Workers communicate amongst each other for bulk data transfer over TCP. Internally the scheduler tracks all work as a constantly changing directed acyclic graph of tasks. A task is a Python function operating on Python objects, which can be the results of other tasks. This graph of tasks grows as users submit more computations, fills out as workers complete tasks, and shrinks as users leave or become disinterested in previous results.

This scheduler has about a 10ms latency and a 200 microsecond task overhead. This makes it less powerful than MPI for high performance computing, such as is common for simulation codes, but a very easy-to-use and flexible system for data analysis. The flexible task scheduling APIs of Dask make it a very approachable way for non-expert Python programmers to use their institution's cluster.

4.4.3 Applications

Dask has had broad impact throughout the PyData ecosystem and further (other languages like Julia have copied the model). It has also had specific impact within a few particular domains:

- Gridded geospatial analysis: Atmospheric, oceanographic, and land analysis frameworks like XArray and Iris are now built on top of Dask and provide researchers within these communities with interfaces that are both intuitive and scale up to modern data sizes.
- Time series analysis: The broader Pandas community has taken on development work and use of the dask.dataframe project, bringing it near parity with the original Pandas codebase for most common workflows.

- Machine learning: A number of researchers have chosen to implement their algorithms in Dask. This is both because it provides a scalable multi-dimensional array construct and because it allows for arbitrary task scheduling, which enables cutting edge researchers to implement newer algorithms easily.
- Bespoke pipelines: Research groups and companies have built their own systems on top of the Dask schedulers, taking the engine that powers large arrays and dataframes and repurposing it for entirely new applications within their domain. This support of custom applications is by far the fastest growing use of Dask.

4.4.4 Ongoing and Future work

Dask has integrated itself into the software of a variety of user communities and scientific domains. Its ability to provide lightweight parallelism without requiring a significant paradigm shift has made it attractive to existing software projects. We continue to work with these communities in outreach activities to empower them to operate on larger datasets and on larger clusters. Existing communities include geospatial analysis, time series analysis, machine learning, real-time analysis systems, and more. Many other communities and domains may also benefit, such as genomics and medical imaging.

As Dask is used in more communities we find ways to improve the internal schedulers to become smarter about more novel situations. As we run on more hardware, such as traditional HPC systems, we find ways in which we can improve our communication stack. As we run at more institutions we find ways to improve the launching and cluster management process to lower barriers to entry for an increasingly broad population of scientists.

4.5 Datashader

Datashader provides a highly optimized computational pipeline that accurately transforms data into an image that can be plotted:

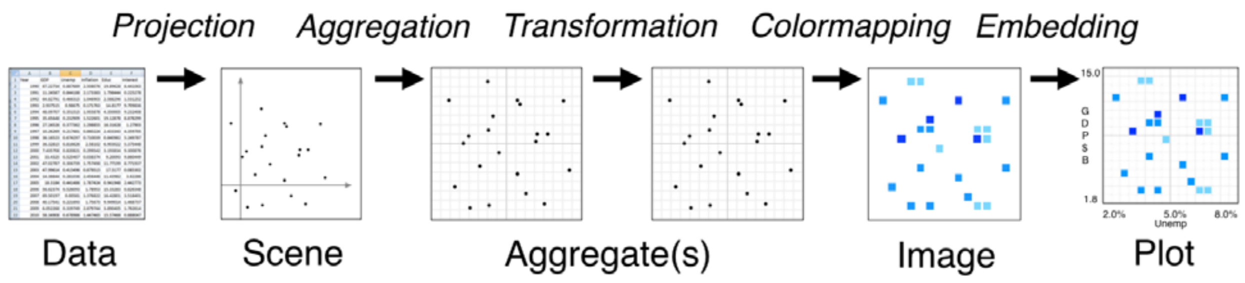


Figure 11 - Datashader Pipeline

Here, a columnar **Data** set is mapped in some way onto a 2D plane to create a **Scene**, which is then aggregated into a regular grid, with the **Aggregate** potentially transformed in some way to select or modify grid cells of interest, and then rendered as an **Image**. In this way, arbitrarily large datasets can be rendered into a fixed-sized image that can be delivered to a remote or local client. For interactive use, the client can then generate requests for other images covering different regions of the **Scene** space, allowing exploration within an interactive Bokeh **Plot** or dashboard while avoiding the need to ever send the full **Data** to the local browser.

4.5.1 Details of the pipeline

In this pipeline, the **Data** can currently be either a Pandas or a Dask dataframe. The Dask dataframe allows out of core execution (for datasets larger than memory), multithreaded execution (to make use of multiple cores on a single machine), and distributed processing (to make use of multiple compute nodes). By delegating the responsibility for such parallelization to Dask, the Datashader code becomes much simpler and can benefit from improvements to Dask supported by other projects.

The *Projection* is a symbolic or logical step rather than a computation, consisting of the user's declaration that they wish to see certain columns of this dataset mapped onto the x axis, y axis, or a categorical axis, along with a specification for the ranges of the data to be considered and the corresponding height and width of the aggregate array to be created in the next step. The actual computation of this projection is performed in a single step during aggregation (next).

The *Aggregation* step is the only computationally expensive stage, because it requires an entire pass through the dataset. Here, Numba-optimized code is used to calculate the number, average, or other reduction of all the datapoints falling into each bin of the aggregate array. The array itself is an xarray datatype, which allows storing multi-dimensional data (indexed by x, y, and category, with arbitrarily many associated value dimensions) efficiently and conveniently.

After aggregation, the originally arbitrarily large dataset has been reduced down to a fixed-size array containing counts per pixel, means per pixel, and so on. The value in each cell in that array can then be mapped into a color, from which an image can be generated and displayed.

4.5.2 Examples

As an example, here is an image created by Datashader in 3 seconds on a Macbook Pro laptop from a 300-million-point dataset, one point for each person in the US 2010 Census, approximately located at that person's residence:

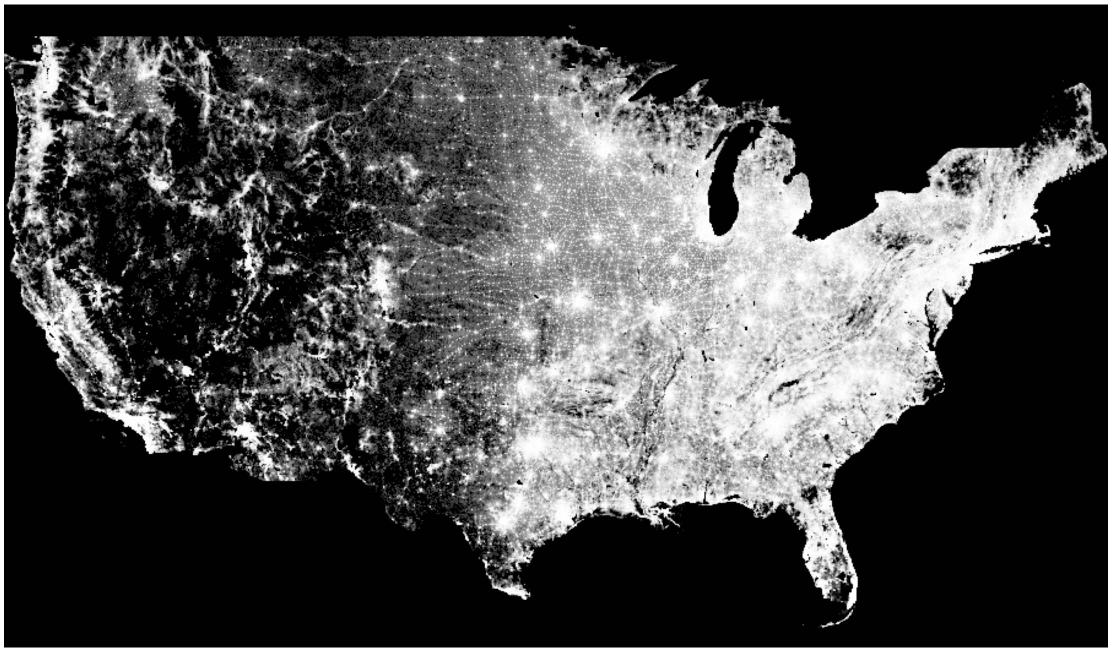


Figure 12 - Datashader US Census Image (300 Million Points)

To make this image, Datashader iterated over a three-column Dask dataframe collecting counts (number of people) per pixel in the final image into each cell of the aggregate array, and then scaled that value into a brightness value for the pixel in that image.

The *Colormapping* stage in Datashader is designed to construct such images in a way that maximizes the bandwidth of the output device and of the human visual system. If there are 255 colors available in a given display device, a naive approach would be to map a count of 0 to black, and the highest observed count per pixel to 255, but unfortunately this approach gives a nearly unusable plot that looks black with only a few white points, even though it is a plot of 300 million points:

Figure 13 - US Census Data with Normal Linear Alpha

Instead, Datashader uses a nonlinear mapping by default (as in the first census image above) that equalizes the number of pixels that are assigned to each of the available colors, providing a rank-order mapping that maximizes information about the distribution of the original dataset (unlike the mostly redundant information provided by a linear transformation, for data that is highly nonlinearly distributed). Using this histogram-equalization approach, a highly informative image can be created without *any* user-adjusted parameters, by simply automatically finding the minimum and maximum values (which map to the minimum and maximum values in the color range) and nonlinearly mapping the intermediate values according to the histogram.

Crucially, each of the steps in the pipeline is available to the Datashader user. For instance, for the following dataset of New York City Taxi trips, a Datashader user can easily aggregate over all dropoffs and all pickups separately, then select only those pixels where pickups are more common than dropoffs to plot in red, and those where dropoffs are more common than pickups to plot in blue. The result very cleanly distinguishes arterial thoroughfares from residential side streets, which would be a calculation very difficult to express or measure on the non-aggregated data:

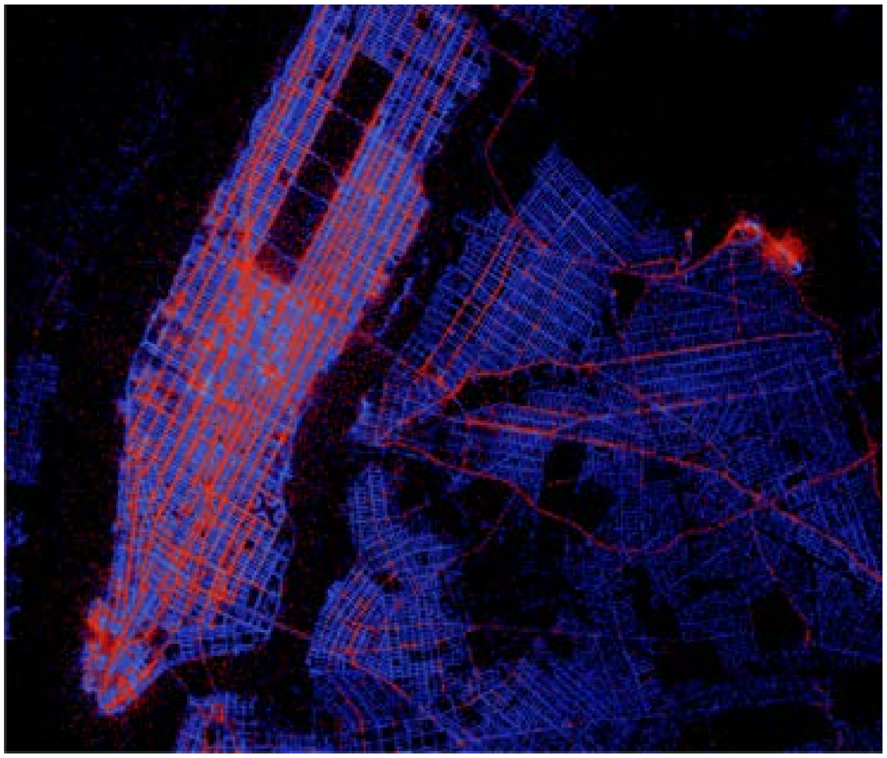


Figure 14 - NYC Taxi Pickup vs Dropoffs

The resulting images can then easily be embedded into Bokeh plots or Bokeh apps and combined with data from other sources, such as maps:

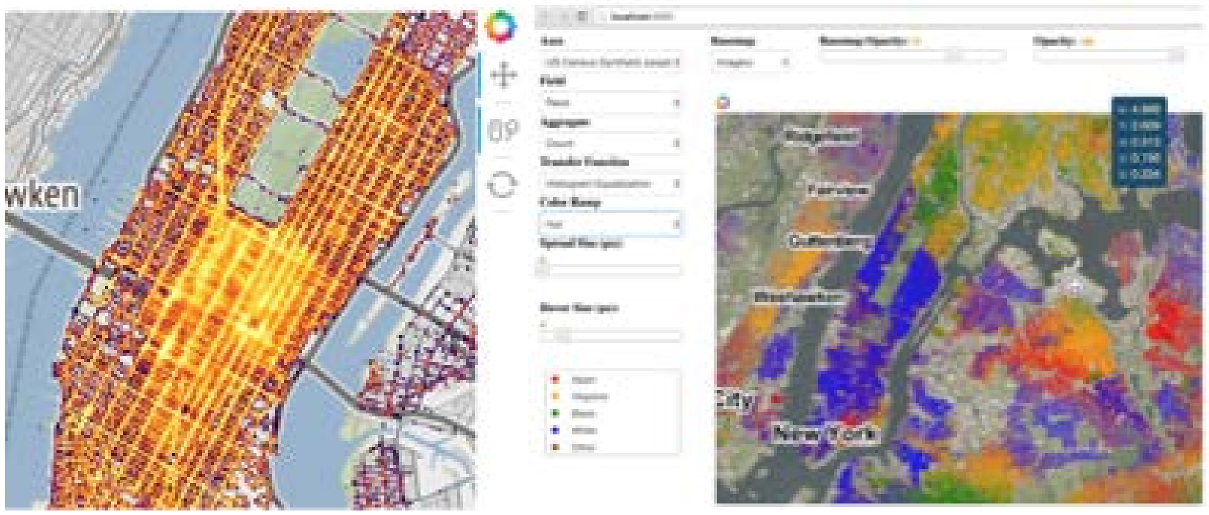


Figure 15 - Datashader in Bokeh

The above plots are all for point data (people or taxi locations), but similar techniques work for trajectories, such as this Opensky data on flight paths in Europe (with the left plot indicating overall frequency of travel, and the others showing ascending flights in blue and descending (and often circling) flights in red:

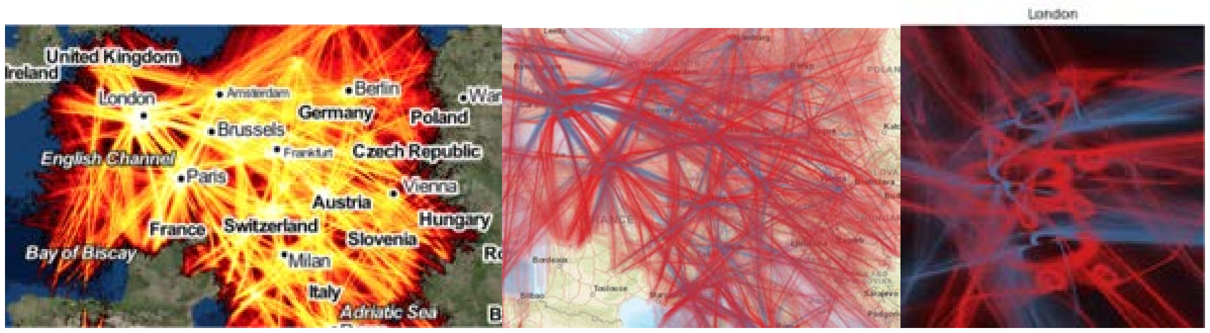


Figure 16 - Datashader European Flight Paths

Similar techniques can be used for graph data (here showing 100,000 UK research collaborations) or time series data (here showing millions of points, plotted to show the true density of overlap in every location, avoiding overplotting):

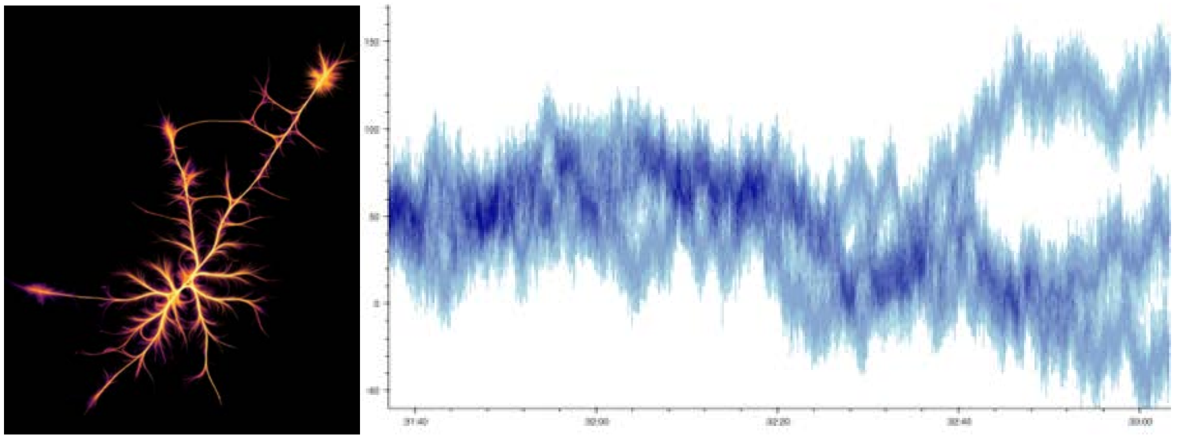


Figure 17 - Datashader Graph and Time Series Data

Constructing these plots takes remarkably little code, and typically no user intervention, because of the automated processing at each stage. For instance, using the new high-level HoloViews interface to Bokeh and datashader, changing a Bokeh plot of 1000 points (Figure 18 - Code for Display Small Number of Points and Paths) to a datashader-based plot of one million points requires adding only the single word “datashade” (Figure 19 - Code to Display Millions of Points and Paths). In this way, researchers and analysts can now work with even very large datasets interactively and conveniently in the web browser, making it much simpler to discover the properties of their datasets and convey them to others.

```
points = hv.Points(np.random.multivariate_normal((0,0), [[0.1, 0.1], [0.1, 1.0]], (1000,)),label="Points")
paths = hv.Path([random_walk(2000,30)], label="Paths")

points + paths
```

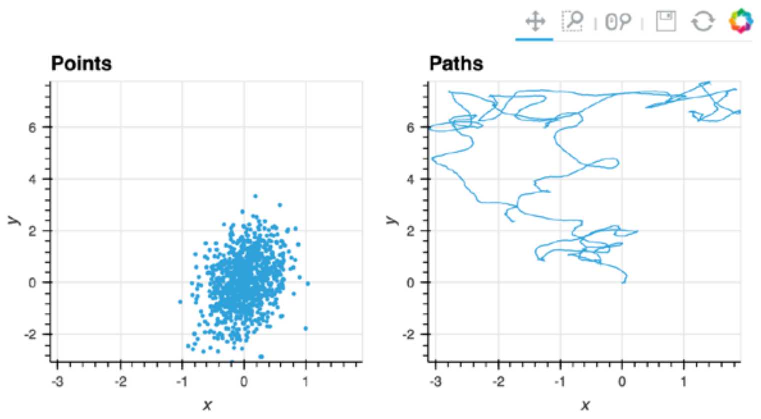


Figure 18 - Code for Display Small Number of Points and Paths

```
points = hv.Points(np.random.multivariate_normal((0,0), [[0.1, 0.1], [0.1, 1.0]], (1000000,)),label="Points")
paths = hv.Path([0.15*random_walk(100000) for i in range(10)],label="Paths")

decimate(points) + datashade(points) + datashade(paths)
```

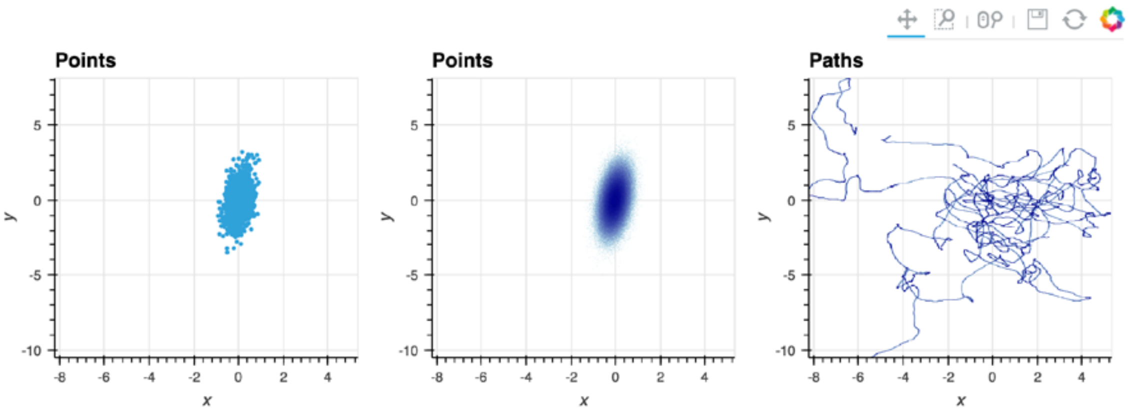


Figure 19 - Code to Display Millions of Points and Paths

5 Conclusions

There continues to be great interest in being able to use an accessible language like Python, which domain experts can easily learn, for large scale data science applications. Historically, languages like Python lacked the performance for large scale data analysis, a way to for users to easily access data used in such analysis, and a means to easily create exploratory and compelling interactive visualizations of the data. Through the XDATA program Continuum has been able to provide tools to address some of the scaling needs and there has been a broadening community support and usage for these tools. Increasing, industry leaders such as Intel, AMD, NVIDIA, IBM, and Microsoft are using the tools and technologies developed under this research as either foundations for their offerings or as a means to make their technologies more accessible. As data continues to grow at an accelerated pace we see continued demand for faster and more scalable computing capability and evolving visualization capabilities accessible from easy to use languages such as Python.

Over the past year we have also seen continued, increasing interest from the open source community in supporting and contributing as reflected in the numbers below.

Table 1 - Community Involvement Stats

	Latest Version	# of releases	Github Stars	# contributors	Monthly download count (Dec 2016)
Blaze	0.11.0	49	1830	51	28k
datashape	0.5.4	19	91	22	
odo	0.5.0	33	582	29	
Bokeh	0.12.5dev11	43	5360	207	102k
Dask	0.13.0	29	1257	83	101k
Datashader	0.4.0	5	478	10	1.5k
Numba	0.30.1	78	2100	68	88k

6 Recommendations

The team sees a number of areas for future research and development that would support processing, analyzing, and making decisions on ever increasing amounts and types of data.

For Blaze

- Work on integration with Dask to provide an easy to use interface that can be used to access, transform and process data in a distributed manner.
- Research and develop GUIs to allow non-technical users to be able to access and perform basic analysis on data of different types without having to know how to access the data (ex. SQL or how to get data on a Hadoop cluster).
- Continue building-out the backend to provide access to different data types, including access to more specialized data, such as satellite imagery.

Numba

- Continue to increase SIMD performance with support for fast math flags and improved support for AVX, Intel's large vector instruction, and Xeon Phi Intel's many core processors with massive parallelism. For example, AVX-512 lets a Xeon Phi core operate on 16 floats at once
- Improved the user experience for developers with better debug support and better error messages.
- Support for "partial compilation" of functions (mix compiled and interpreted code in the same function)
- Stand-alone extension module production
- More Python language supported (jit-classes, comprehension)

Dask

- Work on making Dask accessible to R users.
- Support complex machine learning algorithms
- Easier deployment on clusters
- Support for Graph algorithms
- Further integration with the rest of the PyData ecosystem
- Integration with MPI-based sub-clusters and GPU sub-clusters

Bokeh

- R support in Bokeh Server so R users can leverage the features of Bokeh Server.
- Native visualization for Graph / Network data

- Capability for programatic static (.png, .svg, etc) image generation. Currently Bokeh images require a browser.
- Mechanism to make Bokeh extensions easily sharable, discoverable, and installable.
- Support nested coordinate systems and axes
- Integration with VegaLite / Altair
- Datasource views, to support client-side animations, filters, and group-bys

Datashader

- Large graph/network rendering
- Rendering surface meshes (e.g. altitude measurements, LIDAR) as orthographic projections
- Full support for datetime axes (for time series plots)
- Automated legends, color keys, color bars, and hover support for use with Bokeh and other libraries
- Improved integration into Bokeh and HoloViews, adding additional interactive features (selection, linking, etc.)
- Tiling/partitioning support for input data and output images
- Interfaces for streaming data
- Support for GPU-based dataframes

7 References

7.1 URLs

Blaze GitHub repository: https://github.com/blaze/blaze
Numba GitHub repository: https://github.com/numba/numba
Dask Github repository: https://github.com/dask/dask
Bokeh Github repository: https://github.com/bokeh/bokeh
Datashader Github Repository: https://github.com/bokeh/datashader

8 List of Acronyms

Table 2- List of Acronyms

Term	Description
Bcolz	A columnar data container that can be compressed.
CPU	Central Processing Unit for a computer.
CSS	Cascading Style Sheets - A style sheet language used for describing the presentation of a document written in a markup language. It is used in conjunction with HTML to control the appearance of web page elements.
CSV	Comma delimited file format
GPGPU	General-purpose computing on graphics processing units - The use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).
GPU	Graphics Processing Unit
HDF5	A data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data
HDFS	The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.
HPC	High Performance Computing
HTML	Hypertext Markup Language - The standard markup language for creating web pages and web applications
JIT	Just-In-Time - Compilation done during execution of a program – at run time – rather than prior to execution.
JSON	JavaScript Object Notation - A lightweight data-interchange format.
LLVM	A collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends.

LLVM IR	The intermediate representation, a low-level programming language similar to assembly generated by the LLVM compiler.
ND-Array	A multidimensional container of items of the same type and size.
NetCDF	Network Common Data Form - a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.
NVVM	A compiler intermediate representation (IR) based on LLVM IR designed to represent GPU kernels.
REST	Representational state transfer - One way of providing interoperability between computer systems on the Internet.
ROCm	A platform for GPU Enabled HPC and UltraScale Computing
SIMD	Single Instruction Multiple Data