Scaling PyData Up and Out

Making NumPy- and Pandas-style code faster and run in-parallel.

Travis E. Oliphant, PhD
Python Enthusiast since 1997
Scaling Up vs. Out

- **Vertical Scaling** (Bigger Nodes)
  - Big Memory and ManyCore /GPU Box

- **Horizontal Scaling** (More Nodes)
  - Many commodity nodes in a cluster

- **Best of Both** (e.g. GPU cluster)
"Python's Scientific Ecosystem"
I spent the first 15 years of my “Python life” working to build the foundation of the Python Scientific Ecosystem as a practitioner/developer.
I plan to spend the next 15 years ensuring this ecosystem can scale both up and out as an entrepreneur, evangelist, and innovator.
In 2012 we “took off” in this effort to form the foundation of scalable Python — Numba (scale-up) and Blaze (scale-out) were the major efforts.

In early 2013, I “retired” from NumPy maintenance to pursue “next-generation” NumPy.

Not so much a single new library, but a new emphasis
Rally a community

“Apache” for Open Data Science

Community-led Conference Series with sponsorship proceeds going directly to NumFocus
Organize a company

Purpose
Empower people to solve the world’s greatest challenges.

Mission
We help people discover, analyze, and collaborate by connecting their curiosity and experience with any data.
Start working on key technology

Blaze — blaze, odo, datashape, dynd, dask
Bokeh — interactive visualization
Numba — Array-oriented Python subset compiler

Conda — Cross-platform package manager with environments

Bootstrap (self) and seed funding through 2014
VC funded in 2015
Milestone success — 2016

**Numba** is delivering on scaling out
- NumPy’s ufuncs on multiple CPU and GPU threads
- General GPU code
- General CPU code

**Dask** is delivering on scaling up
- `dask.array` (numpy scaled up)
- `dask.dataframe` (pandas scaled up)
- `dask.bag` (lists scaled up)
- `dask.delayed` (general scale-up construction)
Numba + Dask

Look at **all** of the data with **Bokeh’s datashader**. Decouple the data-processing from the visualization. Visualize arbitrarily large data.

E.g. Open Street Map data:

- About **3 billion** GPS coordinates
- [https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/](https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/).

- This image was rendered in one minute on a standard MacBook with 16 GB RAM
- Renders in a milliseconds on several 128GB Amazon EC2 instances
Categorical data: 2010 US Census

• One point per person
• 300 million total
• Categorized by race
• Interactive rendering with Numba+Dask
• No pre-tiling
Easiest way to get everything!

**ANACONDA®** is….

the Leading Open Data Science Platform powered by Python…
the fastest growing open data science language

- **Accelerate Time-to-Value**
- **Connect Data, Analytics & Compute**
- **Empower Data Science Teams**
Anaconda now with MKL as default

- Intel MKL (Math Kernel Libraries) provide enhanced algorithms for basic math functions.

- Using MKL provides optimal performance for basic BLAS, LAPACK, FFT, and math functions.

- Anaconda since version 2.5 has MKL provided as the default in the free download of Anaconda (you can also distribute binaries linked against these MKL-enhanced tools).
Bottom Line
10-100X faster performance

- Interact with data in HDFS and Amazon S3 natively from Python
- Distributed computations without the JVM & Python/Java serialization
- Framework for easy, flexible parallelism using directed acyclic graphs (DAGs)
- Interactive, distributed computing with in-memory persistence/caching

- Leverage Python & R with Spark

Python & R ecosystem
MPI
Overview of Numba
Compiler overview

Parsing Frontend

C++
C
Fortran
ObjC

Intermediate Representation (IR)

Code Generation Backend

x86
ARM
PTX
Compiler overview

Parsing Frontend

Numba

Python

Intermediate Representation (IR)

Code Generation Backend

LLVM

x86

ARM

PTX

Numba

Python

LLVM

x86

ARM

PTX
@numbajit
def simple():
    total = 0.0
    for i in range(9999):
        for j in range(1, 9999):
            total += (i / j)
    return total

define double __numba_specialized_0__main__2E_simple() nounwind readnone {
    entry:
        br label "%for_condition_7:17.preheader"
    "for_condition_7:17.preheader":
        ; preds = "%exit_for_7:8", %entry
        %total_28 = phi double [ 0.000000e+00, %entry ], [ %3, "%exit_for_7:8" ]
        %storemerge7 = phi i64 [ 0, %entry ], [ %0, "%exit_for_7:8" ]
        br label "%loop_body_8:12"
    "exit_for_6:4":
        ; preds = "%exit_for_7:8"
        ret double %3
    "exit_for_7:8":
        ; preds = "%exit_for_7:8"
        %0 = add i64 %storemerge7, 1
        %exitcond9 = icmp eq i64 %0, 9999
        br 1 %exitcond9, label "%exit_for_6:4", label "%for_condition_7:17.preheader"
    "loop_body_8:12":
        ; preds = "%loop_body_8:12", "%for_condition_7:17.preheader"
        %1sr.iv = phi i64 [ %lsr.iv.next, "%loop_body_8:12" ], [ %1, "%for_condition_7:17.preheader" ]
        %total_36 = phi double [ %total_28, "%for_condition_7:17.preheader" ], [ %3, "%loop_body_8:12" ]
        %1 = sdiv i64 %storemerge7, %1sr.iv
        %2 = f2s double %1 to double
        %3 = fadd double %total_36, %2
        %lsr.iv.next = add i64 %lsr.iv, 1
        %exitcond = icmp eq i64 %lsr.iv.next, 9999
        br 1 %exitcond, label "%exit_for_7:8", label "%loop_body_8:12"}
Image Processing

```python
@jit('void(f8[:,:],f8[:,:],f8[:,:])')
def filter(image, filt, output):
    M, N = image.shape
    m, n = filt.shape
    for i in range(m//2, M-m//2):
        for j in range(n//2, N-n//2):
            result = 0.0
            for k in range(m):
                for l in range(n):
                    result += image[i+k-m//2,j+l-n//2]*filt[k, l]
            output[i,j] = result

~1500x speed-up
```
How Numba works

```
@jit
def do_math(a,b):
    ...

>>> do_math(x, y)
```

Diagram:
- Python Function
  - Bytecode Analysis
  - Numba IR
  - Type Inference
  - Rewrite IR
  - Lowering
  - LLVM IR
  - LLVM JIT
  - Machine Code
  - Execute!
  - Cache
Numba Features

- Numba supports:
  - *Windows*, *OS X*, and *Linux*
  - 32 and 64-bit x86 CPUs and NVIDIA GPUs
  - Python 2 and 3
  - NumPy versions 1.6 through 1.9
- Does *not* require a C/C++ compiler on the user’s system.
- < 70 MB to install.
- Does *not* replace the standard Python interpreter (all of your existing Python libraries are still available)
Numba Modes

- **object mode**: Compiled code operates on Python objects. Only significant performance improvement is compilation of loops that can be compiled in nopython mode (see below).

- **nopython mode**: Compiled code operates on “machine native” data. Usually within 25% of the performance of equivalent C or FORTRAN.
1. Create a realistic benchmark test case. *(Do not use your unit tests as a benchmark!)*
2. Run a profiler on your benchmark. *(cProfile is a good choice)*
3. Identify hotspots that could potentially be compiled by Numba with a little refactoring. *(see rest of this talk and online documentation)*
4. Apply `@numba.jit` and `@numba.vectorize` as needed to critical functions. *(Small rewrites may be needed to work around Numba limitations.)*
5. Re-run benchmark to check if there was a performance improvement.
@teoliphant

The Basics

In [87]: @jit(nopython=True)
    ...: def nan_compact(x):
    ...:     out = np.empty_like(x)
    ...:     out_index = 0
    ...:     for element in x:
    ...:         if not np.isnan(element):
    ...:             out[out_index] = element
    ...:             out_index += 1
    ...:     return out[:out_index]

In [88]: a = np.random.uniform(size=10000)
    ...: a[a < 0.2] = np.nan
    ...: np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])

In [89]: %timeit a[~np.isnan(a)]
%timeit nan_compact(a)

10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
The Basics

Array Allocation
Looping over ndarray `x` as an iterator
Using numpy math functions
Returning a slice of the array

Numba decorator
(nopython=True not required)

```python
In [87]: @jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

```python
In [88]: a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

```python
In [89]: %timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop

2.7x speedup over NumPy!
In [85]:

@jit
def norm(vec):
    mag = 0.0
    for element in vec:
        mag += element**2
    mag **= 0.5

    ret = np.empty_like(vec)
    for i, element in enumerate(vec):
        ret[i] = element / mag

    return ret

@jit
def clamp(x):
    if x > 1.0:
        return 1.0
    elif x < -1.0:
        return -1.0
    else:
        return x

@jit
def angle_between(vec1, vec2):
    norm_vec1 = norm(vec1)
    norm_vec2 = norm(vec2)

    cos_angle = (norm_vec1 * norm_vec2).sum()
    return np.arccos(clamp(cos_angle))
Calling Other Functions

```python
In [85]:

@jit
def norm(vec):
    mag = 0.0
    for element in vec:
        mag += element**2
    mag **= 0.5

    ret = np.empty_like(vec)
    for i, element in enumerate(vec):
        ret[i] = element / mag

    return ret

@jit
def clamp(x):
    if x > 1.0:
        return 1.0
    elif x < -1.0:
        return -1.0
    else:
        return x

@jit
def angle_between(vec1, vec2):
    norm_vec1 = norm(vec1)
    norm_vec2 = norm(vec2)

    cos_angle = (norm_vec1 * norm_vec2).sum()
    return np.arccos(clamp(cos_angle))
```

This function is not inlined

This function is inlined

9.8x speedup compared to doing this with numpy functions
NumPy Ufuncs

NumPy ufuncs (and gufuncs) are functions that operate “element-wise” (or “sub-dimension-wise”) across an array without an explicit loop.

This implicit loop (which is in machine code) is at the core of why NumPy is fast. Dispatch is done internally to a particular code-segment based on the type of the array. It is a very powerful abstraction in the scientific computing stack.

Making new ufuncs used to be only possible in C — painful!
Making Ufuncs with Numba

```python
In [7]: @numba.vectorize(nopython=True)
class game_wins:
    def game_wins(win_probability, max_wins, max_losses):
        wins = 0
        losses = 0
        while wins < max_wins and losses < max_losses:
            if np.random.rand() < win_probability:
                wins += 1
            else:
                losses += 1
        return wins

In [21]: sim_input = np.tile(np.linspace(0.0, 1.0, 100), (5000, 1))
sim_results = game_wins(sim_input, 12, 3)

In [22]: %timeit game_wins(sim_input, 12, 3)
    
10 loops, best of 3: 50 ms per loop
Case-study -- j0 from scipy.special

• scipy.special was one of the first libraries I wrote (in 1999)
• extended Numeric’s umath module by adding new universal functions (ufuncs) to compute many scientific functions by wrapping C and Fortran libs.

• Bessel functions are solutions to a particular differential equation:

\[ x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 - \alpha^2)y = 0 \]

\[ y = J_\alpha(x) \]

\[ J_n(x) = \frac{1}{\pi} \int_0^\pi \cos(n\tau - x \sin(\tau)) \, d\tau \]
scipy.special.j0 wraps cephes algorithm

Don’t need this anymore!
Result --- equivalent to compiled code

In [6]: %timeit vj0(x)
10000 loops, best of 3: 75 us per loop

In [7]: from scipy.special import j0

In [8]: %timeit j0(x)
10000 loops, best of 3: 75.3 us per loop

But! Now code is in Python and can be experimented with more easily (and moved to the GPU / accelerator more easily)!
Numba is now popular!

A numba mailing list reports experiments of a SciPy author who got 2x speed-up by removing their Cython type annotations and surrounding function with numba.jit (with a few minor changes needed to the code).

With Numba’s ahead-of-time compilation one can now legitimately use Numba to create a library that you ship to others (who then don’t need to have Numba installed).

SciPy (and NumPy) would look very different in Numba had existed 16 years ago when SciPy was getting started — and the PyPy crowd would be happier.
Releasing the GIL

In [22]: from concurrent.futures import ThreadPoolExecutor

@jit(nopython=True)
def mag2(z):
    return z.real * z.real + z.imag * z.imag

MAX_ITERS=250

@jit(nopython=True)
def mandel(c):
    z = 0j
    for i in range(MAX_ITERS):
        z = z*z + c
        if mag2(z) >= 4:
            return 255 * i // MAX_ITERS
    return 255

@jit(nogil=True, nopython=True)
def mandel_patch(args):
    rmin, rmax, nr, imin, imax, ni = args
    points = np.empty(nr*ni, dtype=np.complex128)
    values = np.empty(nr*ni, dtype=np.uint8)

    for i, c in enumerate(complex_grid(rmin, rmax, nr, imin, imax, ni)):
        points[i] = c
        values[i] = mandel(c)

    return points, values

Only nopython mode functions can release the GIL
Releasing the GIL

In [24]: ```python
   ...: %%timeit
   ...: with ThreadPoolExecutor(max_workers=1) as executor:
   ...:     results = list(executor.map(mandel_patch, patches))
   ...:
   ...: 1 loops, best of 3: 470 ms per loop
```

In [25]: ```python
   ...: %%timeit
   ...: with ThreadPoolExecutor(max_workers=4) as executor:
   ...:     results = list(executor.map(mandel_patch, patches))
   ...:
   ...: 10 loops, best of 3: 168 ms per loop
```

2.8x speedup with 4 cores
CUDA Python (in open-source Numba!)

```python
@cuda.jit
def array_scale(src, dst, scale):
    tid = cuda.threadIdx.x
    blkid = cuda.blockIdx.x
    blkdim = cuda.blockDim.x

    i = tid + blkid * blkdim

    if i >= n:
        return

    dst[i] = src[i] * scale

src = np.arange(N, dtype=np.float)
dst = np.empty_like(src)

threadsperblock = 32
blockspergrid = (src.size + (threadsperblock - 1)) // threadsperblock
array_scale[blockspergrid, threadsperblock](src, dst, 5.0)
```

CUDA Development using Python syntax for optimal performance!

10-20x faster than CPU

You have to understand CUDA at least a little — writing kernels that launch in parallel on the GPU.
from numba import jit

@jit
def mandel(x, y, max_iters):
    c = complex(x, y)
    z = 0j
    for i in range(max_iters):
        z = z*z + c
        if z.real * z.real + z.imag * z.imag >= 4:
            return 255 * i // max_iters
    return 255
## The Basics

<table>
<thead>
<tr>
<th></th>
<th>Mandelbrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython</td>
<td>1x</td>
</tr>
<tr>
<td>Numpy array-wide operations</td>
<td>13x</td>
</tr>
<tr>
<td>Numba (CPU)</td>
<td>120x</td>
</tr>
<tr>
<td>Numba (NVidia Tesla K20c)</td>
<td>2100x</td>
</tr>
</tbody>
</table>

![Mandelbrot Image]
Other interesting things

- CUDA Simulator to debug your code in Python interpreter
- Generalized ufuncs (@guvectorize) including GPU support and multi-core (threaded) support
- Call ctypes and cffi functions directly and pass them as arguments
- Support for types that understand the buffer protocol
- Pickle Numba functions to run on remote execution engines
- “numba annotate” to dump HTML annotated version of compiled code
- See: http://numba.pydata.org/numba-doc/0.23.0/
What Doesn’t Work?

(A non-comprehensive list)
• lists, dictionaries, user defined classes (sets and tuples do work!)
• List and dictionary comprehensions
• Recursion
• Exceptions with non-constant parameters
• Most string operations (buffer support is very preliminary!)
• yield from
• closures inside a JIT function (compiling JIT functions inside a closure works…)
• Modifying globals
• Debugging of compiled code (you have to debug in Python mode).
Recently Added Numba Features

- Support for named tuples in nopython mode
- Support for sets in nopython mode
- Limited support for lists in nopython mode
- On-disk caching of compiled functions (opt-in)
- JIT classes (zero-cost abstraction)
- Support of np.dot (and ‘@‘ operator on Python 3.5)
- Support for some of np.linalg
  - generatedjit (jit the functions that are the return values of the decorated function)
- SmartArrays which can exist on host and GPU (transparent data access).
- Ahead of Time Compilation
- Disk-caching of pre-compiled code
Overview of Dask as a Parallel Processing Framework with Distributed
Precursors to Parallelism

• Consider the following approaches first:
  1. Use better algorithms
  2. Try Numba or C/Cython
  3. Store data in efficient formats
  4. Subsample your data
• If you have to parallelize:
  1. Start with your laptop (4 cores, 16 GB RAM, 1 TB disk)
  2. Then a large workstation (24 cores, 1 TB RAM)
  3. Finally, scale out to a cluster
Overview of Dask

Dask is a Python parallel computing library that is:

- **Familiar**: Implements parallel NumPy and Pandas objects
- **Fast**: Optimized for demanding numerical applications
- **Flexible**: for sophisticated and messy algorithms
- **Scales up**: Runs resiliently on clusters of 100s of machines
- **Scales down**: Pragmatic in a single process on a laptop
- **Interactive**: Responsive and fast for interactive data science

Dask **complements** the rest of Anaconda. It was developed with NumPy, Pandas, and scikit-learn developers.
Spectrum of Parallelization

Explicit control: Fast but hard

Implicit control: Restrictive but easy

Threads
Processes
MPI
ZeroMQ

Dask

Hadoop
Spark

SQL:
Hive
Pig
Impala
Dask: From User Interaction to Execution

Collections $\rightarrow$ Graphs $\rightarrow$ Schedulers

- array
- bag
- dataframe
- imperative

- synchronous
- threaded
- multiprocessing
- distributed
Dask Collections: Familiar Expressions and API

**Dask array** (mimics NumPy)

```
x.T - x.mean(axis=0)
```

**Dask dataframe** (mimics Pandas)

```
df.groupby(df.index).value.mean()
```

**Dask bag** (collection of data)

```
b.map(json.loads).foldby(...)
```

**Dask imperative** (wraps custom code)

```
def load(filename):
    def clean(data):
    def analyze(result):
```
Dask Graphs: Example Machine Learning Pipeline
Dask Graphs: Example Machine Learning Pipeline + Grid Search
(((A + 1) * 2) ** 3)
(B - B.mean(axis=0)) + (B.T / B.std())
Dask Schedulers: Example - Distributed Scheduler

Scheduler

Worker → Worker
Worker → Worker
Worker → Worker

Same network

Client → Client

Scheduler

Computation graph

worker queues

sockets

report queue

who has what

center/deletion

Network: Same network

Clients: 2

Workers: 4
Cluster Architecture Diagram

- Client Machine
- Compute Node
- Compute Node
- Compute Node
- Head Node
Using Anaconda and Dask on your Cluster

- Single machine with multiple threads or processes
- On a cluster with SSH (dcluster)
- Resource management: YARN (knit), SGE, Slurm
- On the cloud with Amazon EC2 (dec2)
- On a cluster with Anaconda for cluster management
  - Manage multiple conda environments and packages on bare-metal or cloud-based clusters
Scheduler Visualization with Bokeh
### Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Analyzing NYC Taxi CSV data using distributed Dask DataFrames</td>
</tr>
<tr>
<td>2</td>
<td>Distributed language processing with text data using Dask Bags</td>
</tr>
<tr>
<td>3</td>
<td>Analyzing global temperature data using Dask Arrays</td>
</tr>
<tr>
<td>4</td>
<td>Handle custom code and workflows using Dask Imperative</td>
</tr>
</tbody>
</table>

- Demonstrate Pandas at scale
- Observe responsive user interface
- Explore data using a distributed memory cluster
- Interactively query data using libraries from Anaconda
- Visualize complex algorithms
- Learn about dask collections and tasks
- Deal with messy situations
- Learn about scheduling
Example 1: Using Dask DataFrames on a cluster with CSV data

- Built from Pandas DataFrames
- Match Pandas interface
- Access data from HDFS, S3, local, etc.
- Fast, low latency
- Responsive user interface
Example 2: Using Dask Bags on a cluster with text data

- Distributed natural language processing with text data stored in HDFS
- Handles standard computations
- Looks like other parallel frameworks (Spark, Hive, etc.)
- Access data from HDFS, S3, local, etc.
- Handles the common case
Example 3: Using Dask Arrays with global temperature data

- Built from NumPy
  n-dimensional arrays
- Matches NumPy interface (subset)
- Solve medium-large problems
- Complex algorithms
Example 4: Using Dask Delayed to handle custom workflows

- Manually handle functions to support messy situations
- Life saver when collections aren't flexible enough
- Combine futures with collections for best of both worlds
- Scheduler provides resilient and elastic execution
Contribute to the next stage of the journey

Blaze and the Blaze ecosystem (dask, odo, dynd, datashape, data-fabric, and beyond...)