Dask & Numba: Simple Libraries for Optimizing Scientific Python Code

James Crist
Continuum Analytics
Austin, Texas
jcrist@continuum.io

***Abstract*—Python is a high level language that is used by scientists for numeric computations. However, the performance of the language can be a hindrance when scaling to larger data sets, requiring some operations to be rewritten in a lower level language. To address this problem, we propose two libraries to allow numeric Python code to be optimized incrementally, requiring minimal changes. Here we describe Numba, a compiler for a subset of the Python language, and Dask, a flexible parallel programming library.**

I. Introduction

The Scientific Python stack comprises over two decades worth of battle-tested code. Leveraging Python's expressive syntax, complicated computations can be described in only a few lines of code [1]. Due to the simple application program interface (API) of the NumPy array [2], the standard container for numerical data in Python, the language has found wide-spread use in both industry and academia.

Being an interpretted language, performance comes from describing high-level operations in Python, and pushing the bulk of the work down into fast C or Fortran code [2]. However, this model can break down as computation complexity or data size scales. As complexity scales, computations may not be able to be expressed using high-level operations, and the user may need to write slow loops in Python. As data size scales, the need for data-parallelism increases, which is hindered by a historically single-threaded stack. The number of users relying on these packages makes a rewrite impossible. Instead, we propose software tools to improve performance without requiring a full rewrite.

Here we discuss two proposed solutions for these problems. Numba [3] is a just-in-time (JIT) compiler for a subset of the Python language, allowing numeric Python code to be compiled down to fast machine code. Dask [4] is a flexible system for running parallel computations across single machines and clusters. Both of these are designed with simple APIs to make the improved performance accessible to users who may not have a classical computer science background.

II. Numba

The Numba compiler allows users to annotate expensive functions, which are then JIT compiled by LLVM [5] upon being called. The API makes use of Python's decorator syntax, which allows its use to be applied incrementally, without requiring a major rewrite. For example, here is a function for computing a summation over a 2-dimensional array:

```
import numba as nb

# The jit decorator tells numba to
# compile this function
@nb.jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result
```

When applied to a 1000×1000 array, the numba compiled function was $\sim$200 times faster than the uncompiled function on the author's computer. This is because the compiled version is able to operate directly on the NumPy array's data buffer, bypassing the slower Python interpretter.

Numba also contains facilities for parallelizing data-parallel array operations using threads, and compiling functions to run on the GPU. As with the function above, minimal changes to existing Python code is needed for these to work.

III. Dask

Dask is a flexible parallel computing library. Computations are described as directed acyclic graphs (DAG), using a simple specification built from dicts, tuples, and functions – structures that any Python programmer should be familiar with. These DAGs can then be run using a variety of schedulers, from a single machine scheduler using threads, to a distributed scheduler that scales across a cluster. The decoupling of algorithm description from the schedulers allows for easy switching from single machine to cluster as size scales.

On top of these core components, several *collections* are built. These collections provide parallel versions that mirror familiar single-threaded collections – dask.array for NumPy arrays, and dask.dataframe for Pandas [6] DataFrames.

By mimicking the existing interfaces, transitioning to the parallel versions is simple for users, often requiring only

a few small changes. For example, below we compute a matrix multiplication of two random arrays, in both NumPy and Dask:

```
# numpy version
import numpy as np

a = np.random.normal(size=(1000, 10000))
b = np.random.normal(size=(10000, 1000))
result = a.dot(b)

# dask version
import dask.array as da

a = da.random.normal(size=(1000, 10000),
                     chunks=1000)
b = da.random.normal(size=(10000, 1000),
                     chunks=1000)
result = a.dot(b).compute()
```

To convert the NumPy code to Dask code, the chunks keyword needed to be added to tell Dask how to block the arrays, and the compute method needed to be called to tell Dask to execute the DAG. Otherwise, the NumPy and Dask code look roughly the same. However, the Dask code is able to run in parallel and out-of-core, allowing the user to compute on larger-than-memory datasets.

IV. Conclusion

With the increasing size of datasets, performance of numerical code is becoming more important. To address this problem, we propose some tools for optimizing existing Python code incrementally, rather than requiring a full rewrite. We show that Numba, with its simple decorator API, is able to compile Python functions to native code, often matching equivalent code written in C. We also show that Dask, with its familiar collection interfaces, ease the process of scaling existing NumPy or Pandas code to run in parallel or out-of-core. We believe that through using these tools, existing Python code can be optimized in a way that is both performant and transparent.

Acknowledgment

The author would like to thank the Numba and Dask development teams. Development of these packages is sponsored by the Gordon and Betty Moore Foundation, the DARPA XDATA program, and Continuum Analytics.

References

[1] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2007.58

[2] S. van der 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, mar 2011. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2011.37

[3] S. K. Lam, A. Pitrou, and S. Seibert, "Numba," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM 2015*. Association for Computing Machinery (ACM), 2015. [Online]. Available: http://dx.doi.org/10.1145/2833157.2833162

[4] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, 2015, pp. 130–136.

[5] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 51 – 56.