Background:

In the process of doing computational research, it is important that ideas are treated as first class citizen. Being able to focus more on testing a model or implementation, rather than performance is highly desirable. However, a typical project workflow can be split into two phases: an exploratory phase and a scaleup phase. During exploration, rapid idea iteration is the highest priority while during scaleup, the emphasis shifts towards execution speed.

Unfortunately, when running experiments on more complicated models, while it becomes important to try out many different configurations of the model hyperparameters as well as iterate on many different formulations of the model itself, the time that it takes to train or evaluate the model can often cause this process to overtake the ideation phase. With access to machines with more compute, though, more parallel implementations can allow for researchers to spend less time waiting for training to complete and spend more time reasoning about the models, and ideating based on what they discovered about the performance of the models. Though, it is important that these performance gains not come at a significant cost to the ability to reason about the code and its structure, the assumptions it makes, as well as the ability to "tinker" with it. Since in this use case exploratory ease is more important than performance, some more performant parallelization techniques such as creating with a message passing scheme for the algorithm becomes less feasible since it would take significant time and effort, as well as obscure the original intent of the model. We are thus interested in alternative schemes that provide significant performance gains, while requiring minimal modifications to the code in terms of its clarity and purpose.

Currently, the transition from exploration to scaleup is challenging as this often involves requiring faster execution as well as a larger problem size. To address these challenges, researchers are turning to parallel programming methods as solutions to scaleup. However, this introduces a new host of problems as effective parallelization of existing solutions often requires refactoring code for compatibility with various parallelization frameworks.

One specific method of speedup in the data science/machine learning community is to vectorize functions over highly optimized data structures such as Numpy arrays. Rather than take a single value as input, these functions require the entire Numpy array as input, and operations are performed on the data structure as a whole rather than a single value. While this approach offers significant speedups, it is often quite challenging to ensure that the task can be completed using operators and predefined functions that are compatible with the Numpy framework.

To offer researchers and data scientists an alternative method that provides speedup without significant code rewriting, we turn to parallelization methods in Python that are compatible with typical data exploration data structures. Specifically, we identify two libraries, Dask and Ray, that offer this functionality over a popular Python data exploration library, Pandas. The first library, Dask, is a project around numpy and pandas that allows for parallel and out of memory computations by evaluating operations lazily, recording them into a computation graph that is

computed when requested, caching relevant intermediate quantities. It is a very mature and well supported open-source project that has an interface very similar to that of numpy and pandas. Dask functions both on personal computer scale to take advantage of multi-core machines and allowing for out-of-memory computations, as well as scaling up to entire distributed clusters, all with the same code. In contrast to Dask, Ray is designed to be as simple as possible for the user to parallelize their existing Pandas code, claiming that by simply importing Ray as the library, the existing code is effectively parallelized.

Research Question

To investigate whether parallelization is capable of bridging the gap between exploration code and scaleup code, we compare the performance of Dask and Ray with a vectorized implementation of a typical data exploratory task. We evaluate whether the speedup from this parallelization with minimal code rewriting is significant enough such that a data scientist can benefit from this middle ground between exploration and scaleup code.

Methods

We evaluate our methods on a task involving two datasets, one small dataset to reflect an exploratory phase and a larger dataset corresponding to a typical scaleup phase. The exploratory dataset is roughly 200k in size and contains the coordinates of the hotels in New York State, provided by Expedia.com. The scaleup dataset is a repetition of the same hotel coordinate data to ensure that we are creating the same problem but larger. This dataset is roughly 2G in size and better reflects a dataset where execution performance makes an impact.

A typical task for exploring data involves some transformation or computation of some of the dataset features. We choose to compute the Haversine distance function on the coordinates of the various hotels in the dataset. It computes the distance between two points on a sphere given their latitude and longitude, allowing us to compute the distance between each hotel and a location we specify. We follow the implementation provided by Sofia Heisler and her talk from PyCon2017 as a standard implementation that has already been benchmarked.

Our tests consist of implementing the Haversine computation under 4 conditions: Dask, Ray, Vectorized, and Serial using Pandas. This suite of tests will be performed on both datasets and conducted using an AWS c5.9 xlarge instance with 36 vCPUs. This represents the typical resource a data scientists would be working with that bridge exploration and scaleup. If larger instances are required, then it is unlikely that any additional exploration is necessary and the user should be focused solely on performance. Smaller resources will likely not benefit from parallelization due to overhead and poor fit for bridging exploration and scaleup phases.

Results

Small Dataset (1631 entries)

Implementation	Runtime	Speedup
Dask	29.2 ms	1.56
Ray	511 ms	0.089
Serial (Pandas)	45.6 ms	1
Vectorized	1 ms	45.6

Large Dataset (18,358,536 entries: 11,256x increase)

Implementation	Runtime	Speedup
Dask	18.2 s	31.04
Ray	Crashes	-------
Serial (Pandas)	9min 25s	1
Vectorized	313 ms	1805.112

One advantage of Ray that we attempted to explore was its ability to parallelize the vectorized implementations since it was a drop-in replacement for the data structures that were already being used. However, we found that Ray was not capable of performing faster than the serial vectorized implementation even under the large dataset and 36 vCPU setting.

Discussion

We investigated the tradeoff between parallelizing code and development abstractions, where ease of development generally results in less achieved parallelism and thus, performance. The three methods we investigated of speeding up code lie on different points along this tradeoff: Vectorization offers strong speedups but significant refactoring, Dask offered parallelization with some code rewrite, while Ray promised some parallelization with little to no code-rewrite.

We found that Ray doesn't really seem to follow through on its promise of zero effort drop-in parallelism, since when it runs, Ray is slower than the alternatives, and in the large dataset case it crashes altogether. Dask, requiring only minimal changes to the original pandas code, gives (in the large dataset case, where parallelism really shines) a 31 times speedup, as it distributes the workload in parallel across all the CPUs as opposed to having to serially performing the same computation as the pandas implementation. It does this at the expense of maintaining the computation graph of all operations to be performed and using that to schedule workers to calculate all dependencies of the desired output in as parallel a fashion as possible. In the small dataset case, it is clear that vectorizing the code leads to enormous speedups, with the effect

being even more pronounced (an 1805x gain) in the large dataset case. In general, though, vectorization is not always necessarily a possibility depending on the desired computation, and also often requires a significant restructuring of the architecture.

Despite the high potential for parallelism of the task at hand (pointwise mathematical operations over matrix-like data structures), we demonstrated that at the extreme end of this tradeoff, drop-in parallelization does not offer enough speedup, if any, while libraries that offer additional parallelization by incurring more development time such as Dask are indeed capable of bridging the gap between idiomatic code and increased performance.

Contributions

The writeup was a joint effort between Tony and Lawrence. Lawrence explored Dask. Tony explored Ray. We jointly explored the serial and vectorized implementations.