Evaluating Strong Scaling for Feature Vector Encodings

William Watson
Johns Hopkins University
`wwatso13@jhu.edu`

Abstract

I explored the scalability and speedup performance of various methods of parallelization by using Python's multiprocessing module and IPython's Parallel Programming API. I applied these tools to feature vector encodings as a benchmark on performance. The goal of this project is to determine if IPython's Client-Engine Parallel Programming schema outperforms the native Python multiprocessing implementation for the feature vector encodings of images.

1 Introduction

For this project, I parallelized two algorithms: Histogram of Oriented Gradients (HOG) and Color Histograms. These two methods were chosen since the conversion of images into feature vectors was the most computationally intensive process of Paintings 2 Artist (P2A).

The focus of this project is to analyze the scalability of the most costly functions using different tools and methods. The implementation of Python used is CPython. The Modules used for parallelism are Python's multiprocessing, IPython's Direct View Client Interface, and IPython's Load Balanced View Client Interface. All tests were run on Amazon's C3.2XLarge EC2 Instance for high performance computing with 8 vCPUs.

2 Background

2.1 Data

To test the effectiveness of our attempts at parallelization we will make use of two data sets.

Our original data set created for P2A includes a collection of paintings from six artists: Dali, Monet, Turner, Rembrandt, Van Gogh, and Picasso. All images were resized so that the largest dimension is set to 500 pixels while preserving the aspect ratio of the original image. This standardizes the information being processed by our encoders. Therefore, no image dominates any other in size.

The second set of images comes from a collection of Flickr photos, split into 6 categories: buildings, food, people, faces, cars, trees. Unlike the first data set, these images are skewed in the variation of size, so we expect this to be a major issue in performance.

2.2 Histogram of Oriented Gradients

The HOG Feature Extraction and Description algorithm is based on evaluating an image's normalized local histograms of gradient orientations in a dense grid. The practical uses of HOG descriptors that it capitalizes on the idea that an image can be characterized by its distribution of local intensity gradients or edge directions. (**?**).

2.3 Color Histograms

The purpose of the Color Histogram is to encode an image with the frequency of pixel values. The size of the histograms are set to be 2^n, where n is the number of bits to encode.

We use a simple voting-scheme for our histograms. For every pixel in the image, we encode it to create a specific "bin" for that color. Then we increment the vote for that "bin". This will ensure that all similar colors share the same "bin".(**?**).

3 Python Multiprocessing

Python offers a module that allows for a program to spawn processes to leverage multiple cores on a machine. We use the Pool object, allowing us to create a pool of workers that we can apply arguments to functions to be run on the workers. This offers us a convenient means of parallelizing the execution of a function across our input images, distributing the input data across processes, known as data parallelism. (**?**)

There are three methods that we will use to test Pool: apply, map, and apply-async.

3.1 Result

The apply method takes a function and applies an argument to it to be run on a processor. This is a blocking operation, so as we apply each image, we must wait for the computation to finish. We can see this in the speedup charts, where the speedup for the Apply method is actually decreasing as we increase the amount of processors. (All charts can be found in the appendix). This enforces synchronization of our calls, contributing to a high cost from interference. When partitioning was enabled on 8 cores, the runtime decreased consistently, since we give each processor more work to do instead of sending it 1 image file and waiting for it to complete. Thus we mitigate startup costs from data sending. This occurred in both the HOG and Color Histogram algorithms.

For HOG, our speedup is 0.97 for paintings, 0.98 for Flickr. According to Amdahl's Law, our portion of parallel code p is -0.03 and -0.02.

The map method takes a function and maps the arguments to it, spreading the work across the pool of workers. This method has much better performance than the apply for both algorithms, since we can push all the work onto a queue for the workers to process, blocking until all the computation is complete. From both data sets, we achieved a higher speedup from the Paintings set, because our paintings were standardized to aspect ratio, thus having less skew than the Flickr set, where each process did uneven amounts of work. When run on 8 processes with partitioning enabled, as we increased the partition size, our time would decrease from 1 to 10 images per batch, but then increased as we sent batches of 50 and 100. This increase was more profound in the Flickr set, highlighting the issues of skew, since the larger batches may contain slightly larger images and force the other processors to wait. For HOG, our speedup is 4.76 for paintings, 4.83 for Flickr. According to Amdahl's Law, our portion of parallel code p is 0.90 and 0.91. For Color, our speedup is 3.92 for paintings. According to Amdahl's Law, our portion of parallel code p is 0.85.

The asynchronous apply allows us to delegate our tasks to the pool, and returns to us an asynchronous object that we can then wait for completion. So unlike apply, we can delegate our tasks and then wait for all of them to complete. From our tests, we achieved the same speedup in the Flickr data set, but started to lose performance on the Paintings set, when compared to the map function. Our asynchronous apply suffers the same partitioning problem as map, where if the partitions are too large, then the processes lose the flexibility to take on work evenly as skew dominates the larger partitions. This is most obvious in the Flickr data set, where doubling the partition size drastically increases the runtime cost for partitions above 10 images. For HOG, our speedup is 4.55 for paintings, 4.85 for Flickr. According to Amdahl's Law, our portion of parallel code p is 0.89 and 0.90. For Color, our speedup is 3.92 for paintings. According to Amdahl's Law, our portion of parallel code p is 0.85.

4 IPython Parallel Programming API

IPython's architecture allows programs to take advantage of both parallel and distributed computing. The architecture consists of four components: Engine, Hub, Schedulers, Clients. (**?**)

The Engine is a Python instance that takes Python commands over a network connection. The Engine blocks when user code is being executed.

The Controller processes provide an interface for working with a set of engines. A IPython controller consists of a Hub and a collection of Schedulers. IPython provides two models for interacting with engines: Direct and Load Balanced. A Direct interface addresses the engines explicitly. A Load Balanced interface lets the Scheduler assign work to appropriate engines. (**?**)

The Hub is a process that keeps track of engine connections, schedulers, clients, as well as all task requests and results.

The Client connects to a cluster to submit tasks to be run in parallel.

4.1 IPython Direct Client View

We apply the same methods from Pool for consistency. We also address the clients directly by submitting work to each engine.

The apply method suffers from the same affect as the native Python implementation. The synchronous blocking forces the program to wait for one Engine to complete its task before submitting the next task to another engine, and so on. Therefore we do not get any speedup from this method.

For the map method, IPython submits its tasks to the engines on the cluster. This version does not do dynamic load balancing. We see good speedup as we increase the number of engines. However, the speedup is slightly less than the native map, due to the extra overhead between the communication of the various components of an IPython cluster. We get better speedup on the Painting data set over the Flickr set, since IPython Direct View does not attempt to do load balancing on the arguments, instead opting for static scheduling. Our speedup is 4.35 for paintings, 3.95 for Flickr, and our portion p of parallel code is 0.88 and 0.85 for HOG.

In the Asynchronous Apply, as we apply every argument to an engine before waiting for all computation to finish, we get drastically different speedups from the Flickr set over the Paintings for HOG. In the paintings, our speedup tails off just under 2 relatively quickly, but for the Flickr set it outperformed the map until 16 engines. For the paintings set, the map consumes our list, breaks it into chunks, and sends it to the worker pool of engines. Breaking our list into chucks performs better than applying each item in the list between processes one at a time, causing an increase in startup costs to run our functions.Our speedup is 1.89 for paintings, 4.2 for Flickr, and our portion p of parallel code is 0.54 and 0.87 for HOG.

When partitioned on 8 Engines, the direct view exhibits the same behavior as the native python implementation, where we see performance gains for small increases in partition size, but soon see a degradation after setting the size to over 50. This is clearly due to trade off between starting up processes on tasks and waiting for results that end at different time from skew when the partition number is too high.

4.2 IPython Load-Balanced Client View

The Load Balanced View for IPython is the class for load-balanced execution via the task scheduler. It allows for the scheduler to determine which engine the task should run on. This allows for a dynamic scheduling of work to our engines. (**?**)

Since we no longer directly address the engine in the client interface, we let the scheduler pass our tasks. This clearly will add more to the startup costs in dividing the tasks between engines, leading to a reduction in speedup. We see that this is true for both data sets, but is more pronounced in the paintings tests than the Flickr set, compared to the Direct and Native Python implementations.

For the color histogram applied to the Paintings set, we get almost equal performance from both the map and asynchronous apply.

For the HOG, we get different speedup performance depending on the data set and method. For the paintings data set, we see similar speedup trends as we did in the direct view, where the map function performed better than the apply-async method. However, in the Flickr set, we see that both methods do perform relatively well to the same degree. This implies that for HOG, having a skewed data set allows for better speedup across these two methods than compared to the standardized paintings set, suggesting that the Load Balanced View is better equipped to handle skewed data. According to Amdahl's law, we have a higher p for the Flickr set over paintings.

When partitioning is enabled, we found that the load balanced view sees significant improvements with larger partitions. In fact, we see speedup by a factor of 3 when increasing the partition size from 1 to 50. This suggests that a user must give the load balanced client view large partitions to effectively balance the tasks between engines. This is directly related to the costs of letting a task scheduler handle the execution of tasks on engines, and sending to many messages clearly results in a degradation of performance across the cluster.

5 Conclusion

From the tests conducted on AWS, we have evaluated the strong scaling for the Python multiprocessing library and IPython's Direct vs Load-Balanced Client Views.

The Native Python Multiprocessing implementation achieved the best speedups for both the HOG and color histogram charts. Python is able to handle both skewed and non-skewed data sets reasonably well between the map and apply-async methods, we see speedup gains almost being equivalent.

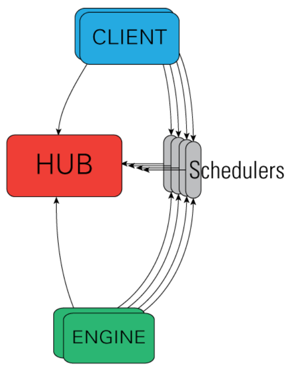
In comparison, IPython's Direct Client View achieved less speedup than its native Python implementation. For HOG, the Direct View struggles to achieve a decent speedup with the apply-async method. It is outperformed by the map function. However, when given a skewed data set, both implementations saw equivalent speedup. The Color Histogram had no noticeable difference between the methods.

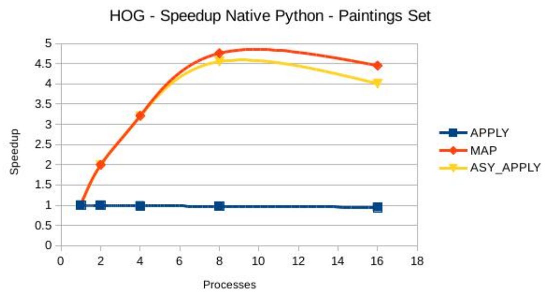
The Load Balanced view worked the best with partitioning enabled on a skewed data set. When partitions were either too large or small, the costs of load balancing the tasks contributed to the total runtime costs, decreasing speedup performance.

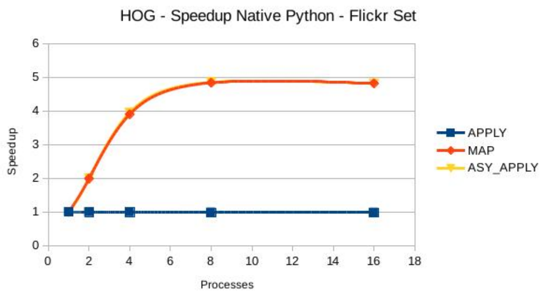
In this project, we have evaluated the strong scaling performance of two tools, implemented with various methods and functions. Originally, I expected IPython to perform better with its cluster load balanced view. However, the native Python implementation achieved the best results in performance. However IPython is more flexible in its applications of parallelism, including its features that allow for distributed computing across multiple clients.

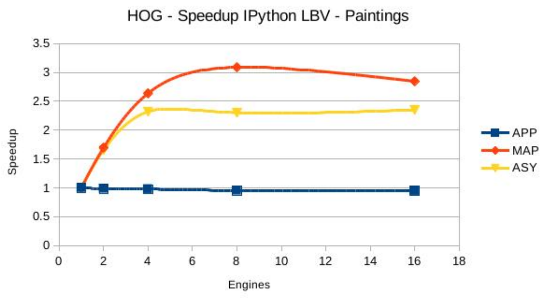
6 Appendix

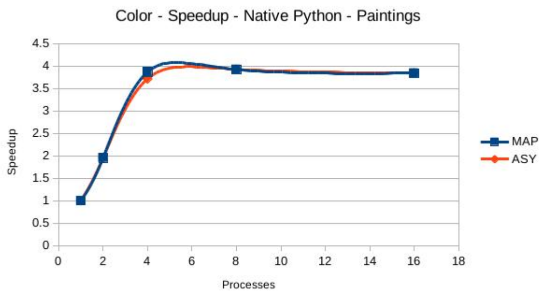
Figure 1: IPython Parallel Architecture.

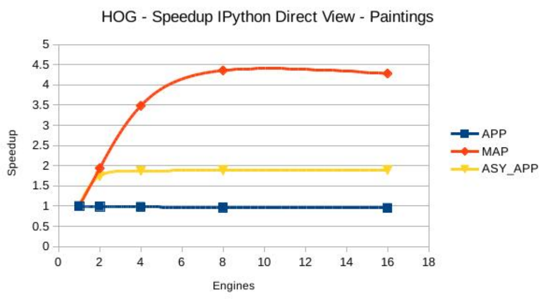


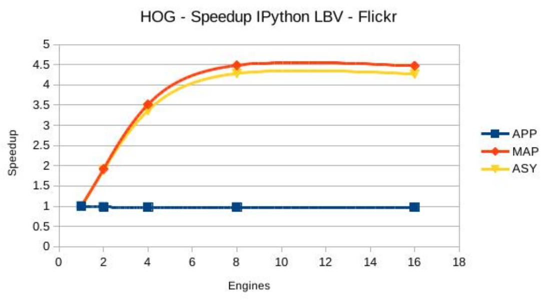


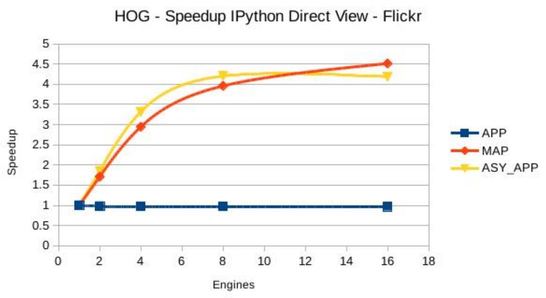

HOG - Speedup Native Python - Flickr Set
Speedup
Processes
APPLY
MAP
ASY_APPLY

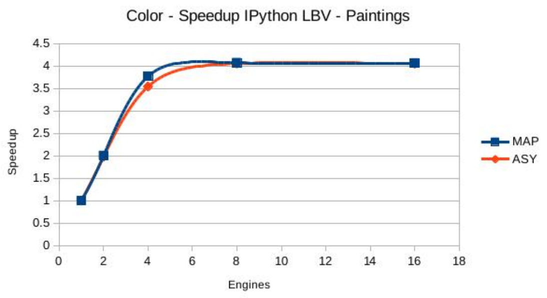

HOG - Speedup IPython LBV - Paintings
Speedup
Engines
APP
MAP
ASY

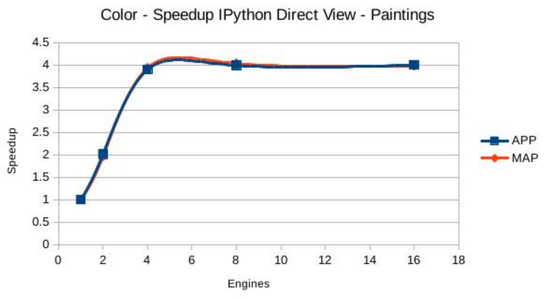

Color - Speedup - Native Python - Paintings
Speedup
Processes
MAP
ASY


HOG - Speedup IPython Direct View - Paintings
Speedup
Engines
APP
MAP
ASY_APP


HOG - Speedup IPython LBV - Flickr
Speedup
Engines
APP
MAP
ASY


HOG - Speedup IPython Direct View - Flickr
Speedup
Engines
APP
MAP
ASY_APP


Color - Speedup IPython LBV - Paintings
Speedup
Engines
MAP
ASY


Color - Speedup IPython Direct View - Paintings
Speedup
Engines
APP
MAP

References

[Blessing, 2010] Alexander Blessing and Kai Wen. 2010. *Using Machine Learning for Identification of Art Paintings.* Stanford University, Stanford, CA. Link

[Dala, 2005] Navneet Dalal and Bill Triggs. 2005. *Histogram of Oriented Gradients for Human Detection.* INRIA Rhone-Alps, Montbonnot 38334, France. Link

[Python 2.7.13 Docs] *16.6. multiprocessing - Process-based threading interface.* Link

[IPython-Docs] *Using IPython for parallel computing.* Link

[IPython-Docs] *Using IPython for parallel computing. IPython's Direct View* Link

[IPython-Docs] *Using IPython for parallel computing. Details of Parallel Computing with IPython* Link