

# Customizing Input File Formats for Image Processing in Hadoop

Jeff Conner  
Arizona State University

## Abstract:

*This paper describes in a general sense how the Hadoop API can be extended to deal with multiple file formats beyond ASCII text. This technique is applied to binary image files in order to enable Hadoop to implement image processing techniques on a large scale. Several image processing algorithms are utilized to demonstrate this technique as well as a few different approaches to image file segmentation, similar to what is already done for ASCII text file segmentation.*

## Introduction:

Since its conception, Hadoop has traditionally been thought of as an ASCII text file processing utility, however, given the nature of the technology driving cloud computing and the growing development of toolsets and techniques, it becomes useful to extend Hadoop to deal with a wide variety of file types beyond ASCII text files. By extending the current API in the Hadoop library we have built a system that allows for large scale image analysis using any number of image processing techniques. In this paper I will introduce the API changes we made in order to allow Hadoop to handle images as well demonstrate this technique using a few sample image processing algorithms.

From a cursory examination of the Hadoop API it appears that it was originally designed to work with ASCII files. However, the API allows for creation of custom input formats, by implementing the “FileInputFormat” and “RecordReader” interfaces. Through these interfaces, it becomes possible to define the way in which data is segmented and sent to the Mapper for processing. We created custom input formats to support each image processing algorithm we implemented. We will introduce each algorithm and its custom input format, and explain the implementation details of each one.

## Customizing the Input Format:

In this section I will explain in a general sense what needs to be done in order to tell Hadoop how to deal with files it finds when processing a request.

There are two interfaces that need to be implemented in order to allow Hadoop to work with custom file formats. The first interface to implement is the FileInputFormat interface that defines the key value pair types and the second is the RecordReader interface which specifies how the file is segmented and sent to the Mapper. Within the RecordReader interface there are a several methods that need to be implemented. One method that is of particular importance is the “next” method. This method determines what portion of the file will be read and sent to the Mapper. It is here that the user will

determine the segmentation behavior for the file reading and analysis process. In the ASCII text file example, the input file is segmented by the newline character. We will discuss the segmentation behavior of each of our image processing algorithms in their respective sections later on.

Finally, when designing a program to run in the Hadoop environment, one typically specifies the input and output formats to use when processing files, in this case our custom input and output (should the user specify one) formats. This is usually done in the main method of driver class that initializes the JobConf object. The user can specify a variety of arguments when setting up the job including the input and output format. When dealing with text files usually the TextInputFormat and TextOutputFormat classes are specified for the input and output formats respectively. It is here that the user would specify their custom input and output classes.

### **Image Processing Algorithms:**

We chose to implement three algorithms to determine the feasibility of image processing in Hadoop. We haven't done much in depth experimenting to see what kind of performance gains can be achieved using Hadoop over more conventional methods, but rather whether or not it could be done at all.

#### ***Simple Pixel Detection:***

By implementing this simple algorithm, the details of which will be discussed below, we were trying to determine if we could read an image from disc and send it to the Mapper using some data structure already provided by Hadoop. Through the interfaces provided by Hadoop we could extend the current API to allow images to be parsed, using a BytesWritable container. In order to demonstrate this functionality we had to first develop a custom input format and record reader. This interface implementation allowed us to ingest, analyze and determine how to split up each section of the image and send the information to the Mapper for processing.

In order to test this functionality we needed to develop a simple image processing algorithm to deal with the sub images in the Mapper. Therefore, we wrote a simple pixel detection algorithm. The Mapper reads a control pixel value in RGB format from the distributed cache that is provided by a text file read by the driver. It then passes this pixel value, along with the sub image to the pixel finder class that compares values of each pixel in the input image 1-by-1 to the control pixel value. If a match is found, within a certain tolerance level (in the runs used  $\pm 5$  in each RGB channel), the pixel is marked as a match. The location and filename are passed to the Reducer, which collects all values per file and writes the part-xxxx file.

The data used to test this algorithm was a set of control images created by painting the desired control pixel into the original image in the four corners and then randomly throughout the image. We also created a custom input format and record reader that sent

the input image as a whole to be worked on by the Mapper. A sample input image is provided below.



Figure 0: Control image for the pixel detector algorithm.

### ***Pixel Detection Results:***

The results for this algorithm just confirmed that the pixel detection algorithm works. This was mainly just to establish that we could send an image to the Mapper and have it process it correctly. A sample section of output is provided below. The first column is the filename, the second and third is the row and column (respectively) that the pixel was found at.

```
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 3
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 400
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 401
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 402
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 403
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 404
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 405
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 406
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 407
hdfs://s49-1:9001/user/jsconner/data/blob/C15R23C00R00_0000.jpg 0 408
...
```

### ***Blob Detection:***

After verifying that we could send an image to the Mapper, we decided to experiment with ways that the image could be segmented for better performance (the segmentation process will be discussed later on). We implemented a simple blob detection algorithm to test this functionality. The algorithm takes an input image and starting at the pixel in row 0 and column 0, progressively builds groups of like pixels (called blobs). The algorithm takes a naïve approach to the comparisons in that given an input pixel, it compares all the pixels in all the blobs to determine: (a) is the input pixel located 1 away from the current blob pixel in either +- x or y, (b) if the input pixel passes the first test, is the average RGB color of the pixel within the tolerance level of the current blob pixel. If both the input pixel passes both tests then it is added to the current blob and the next pixel is checked, until all pixels in the image have been assigned to a blob. As a finished product the algorithm assigns a random color to each blob and blends that color with the original image to visually identify the location of each blob in the original image.

### ***Blob Detection Results:***

This algorithm is inefficient for large images as it is a function of the number of pixels times the number of blobs times the number of pixels in each blob. In the worst case the algorithm has exponential growth. In order to keep the images relatively small we created a custom input format and record reader that progressively halves both the width and height values until they are less than or equal to some threshold value (in this case 100 pixels).

When testing the performance of the serial implementation of the algorithm versus the Hadoop implementation we used a test image of 410 x 410 pixels. The serial implementation processed the image and created the blob clusters in 4 minutes and 52 seconds. The Hadoop implementation split the input images into 8 smaller images (51 x 51 pixels). Each of these sub images was sent to the Mapper to be worked on independently. When the algorithm finished in the Mapper, the color blended blob sub image was written to the HDFS. The name of the original image, the sub image name, as well as the dimensional information of the sub image and location in the original image was sent to the Reducer, which read each sub image and stitched them back into 1 image. The Hadoop method took 23 seconds to run using the same input image.

There was a significant speed up obtained by moving to the Hadoop implementation over the serial implementation. However due to independent processing of each section of the original image, and no re-composition effort beyond simple stitching, the resulting Hadoop image appears badly segmented due to the random color assignment of the blobs. An example is provided below with the serial implementation shown as a comparison.



Figure 1: Original input image for blob detection algorithm.

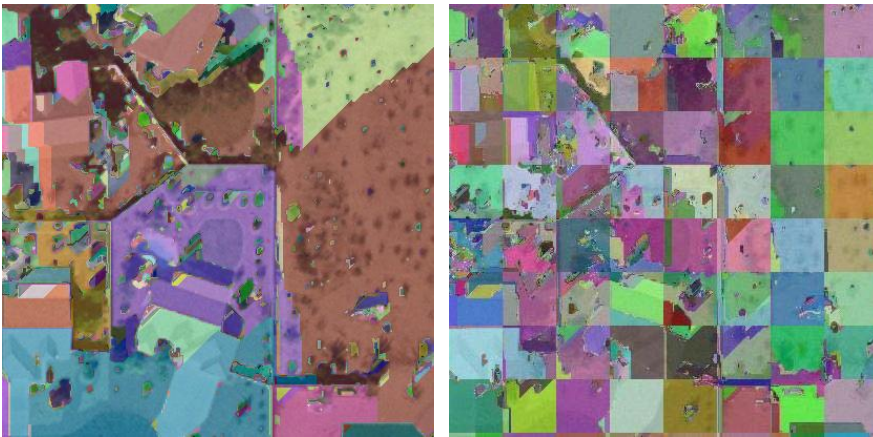


Figure 2: Product comparison of the serial implementation (left) versus the Hadoop implementation with random coloring of the blobs (right).

We attempted to implement a different coloring scheme, which assigns a color value to the pixel based on the average of averages of each pixel in the blob and then normalized to an HSV color ramp function. The resulting product image is a lot cleaner but still contains some disjoint between sub image slices. An example is provided below, again with the serial implementation shown for comparison.

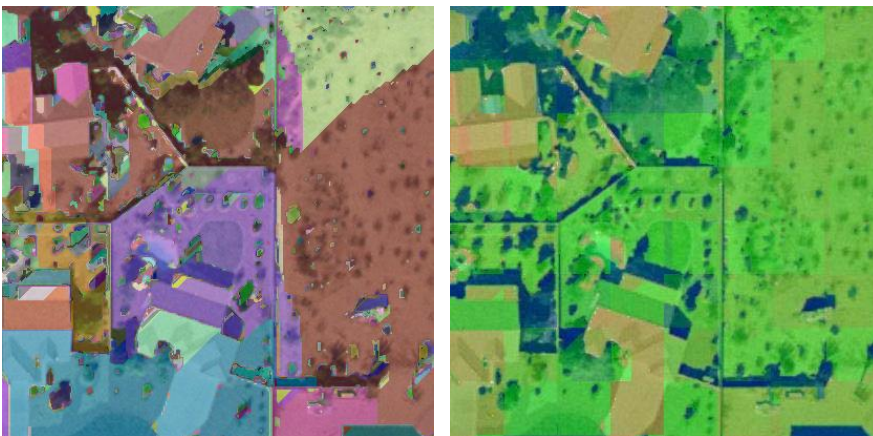


Figure 3: Product comparison of the serial implementation (left) versus the Hadoop implementation with HSV color ramp for the blobs (right).

### ***Sobel Edge Detection:***

The previous experiment showed how re-stitching can be an issue when slicing images into sub-images for the Mapper. We therefore decided to implement the Sobel Edge Detector to see if overlapping the images when slicing would solve the problem, in whole or part. The algorithm works by taking a given pixel location and checking the surrounding pixels in all eight directions to determine the gradient in the X and Y direction as well as magnitude to be assigned to the target pixel. For more detailed information on how this algorithm is implemented please read the Wikipedia article on the subject that was used as the basis for our example which can be found here: [http://en.wikipedia.org/wiki/Sobel\\_operator](http://en.wikipedia.org/wiki/Sobel_operator). We first implemented the algorithm in a single process implementation to obtain a control case. We then implemented the algorithm with the custom import format and record reader that was used in the blob detection example. Both the control and the resulting Map/Reduce product are shown below.

### ***Sobel Edge Detection Results (simple splitting, local min/max)***



Figure 4: Sobel Edge Detection Results. Left: Original Image, Center: Single processor example, Right: Map/Reduce using the Blob Detection Input Format and Record Reader.

The results shown in figure 4 demonstrate that each Mapper is going to need some global information about the image as a whole, in order to normalize the color range for the final product. The global min and max values for the pixel averages for the entire image must be known to each Mapper at run time. This involves preprocessing the image and creating a distributed cache file containing the min/max values or a two pass map/reduce phase where pass one simply analyzes the image and records global information to be passed on to the next step. When the global min/max information is known the resulting product is much better than the previous attempt. The results, along with a side-by-side comparison of the previous attempt are shown in figure 5 below.

***Sobel Edge Detection Results (simple splitting, global min/max)***

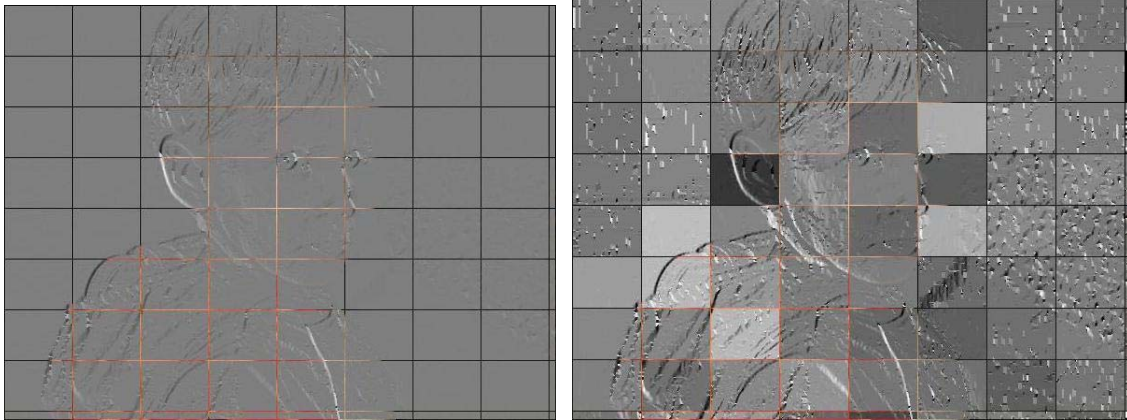


Figure 5: Known Global Min/Max (left) versus Locally Computed Min/Max (right)

The image quality has been greatly increased with the known global min/max value but there still remains a stitching problem. In order to resolve this issue a new custom input format and record reader needs to be created, that will, due to the nature of the Sobel algorithm, need to slice up the original image so that there is at least a one pixel overlap in the ideal case. We found that in a three pixel overlap was needed to produce the best results. The focus of this example is to show how re-combining can done in the Reducer to reassemble the final product image. Figure 6 below shows a side-by-side comparison of the single process image and the overlapping Map/Reduce product image.

***Sobel Edge Detection Results (overlapping split schema):***

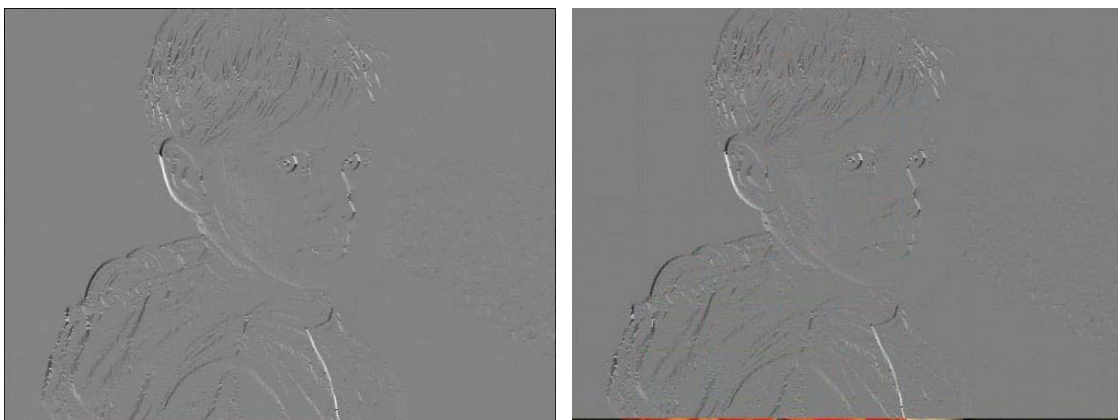


Figure 6: Sobel Edge Detection Single Process (left) versus Map/Reduce (right)

There are still some edge issues that need to be worked out in the final image, but the results are pretty good. Performance in the single process case is really fast so a fair comparison would have to be done on a large sampling of images to determine if Hadoop offers a performance speedup. We initially implemented the algorithm incorrectly, and the images above reflect the error. We went back and fixed the image algorithm and the

resulting image is shown in Figure 7 below next to the resulting image first attempt at the algorithm (both images were from the Map/Reduce implementation).



Figure 7: Comparison of the correct implementation of the Sobel Edge Detection algorithm (right), with our initial implementation of the algorithm (left)

### **Conclusion:**

The purpose of these experiments was to determine if it was possible to modify Hadoop to work with images instead of text files. In realizing this goal, we have shown that by creating a custom input format and record reader, any type of file can be read and managed by Hadoop. With the wide availability of large collections of imagery datasets it becomes necessary to develop means to analyze them quickly and accurately. Cloud computing through Hadoop could provide relatively inexpensive means process datasets of this magnitude without seriously compromising performance. We believe the process we have developed could be applied to a wide variety of image processing disciplines; the intelligence community and large scale disaster recover situations for example.

We believe the next step from here is to design a way that any image processing algorithm, or a large collection of algorithms, can be seamlessly interjected into the process allowing for on demand image processing based upon the source image type and desired analysis. This toolset would go a long way to enable cloud computing to become a solution for large scale image processing needs at a comparatively low cost.