Image Processing in MapReduce

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Let us now look at another example of an algorithm in MapReduce – Image Processing.

The Problem

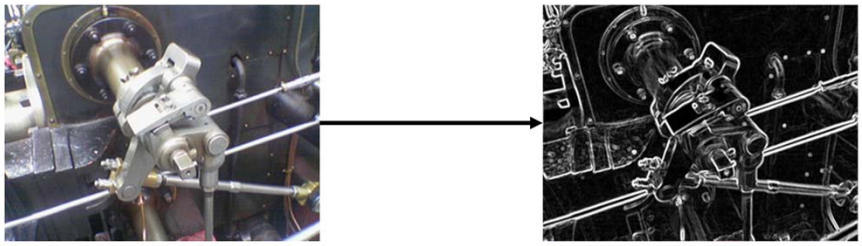
- Image processing involves the application of some operation on an Image
  - Images are typically represented as a 2D-Matrix of values (Binary, Grayscale or RGB/CYMK color values)
  - Operations on images are typically matrix operations.
- Image Processing can be a computationally intensive process

Original Image Processed Image

Image processing involves the application of some operation on an Image. These operations can be feature extraction, application of some color filter etc. Image are typically represented as a 2D-matrix of color values that are either binary, grayscale or RGB/CYMK color values. These operations themselves are typically matrix operations.

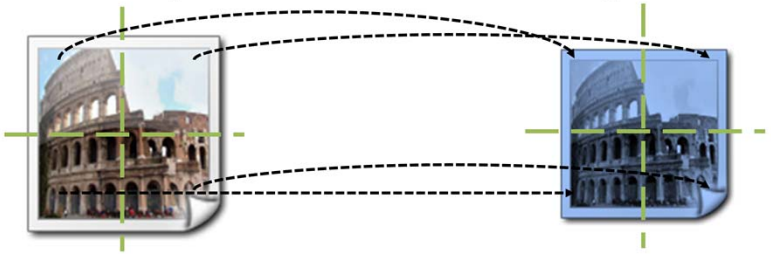
Hence Image processing can be a computationally intensive process, particularly for large images or a large number of images.

Example: Sobel Edge Detection



How to speed up Image Processing?

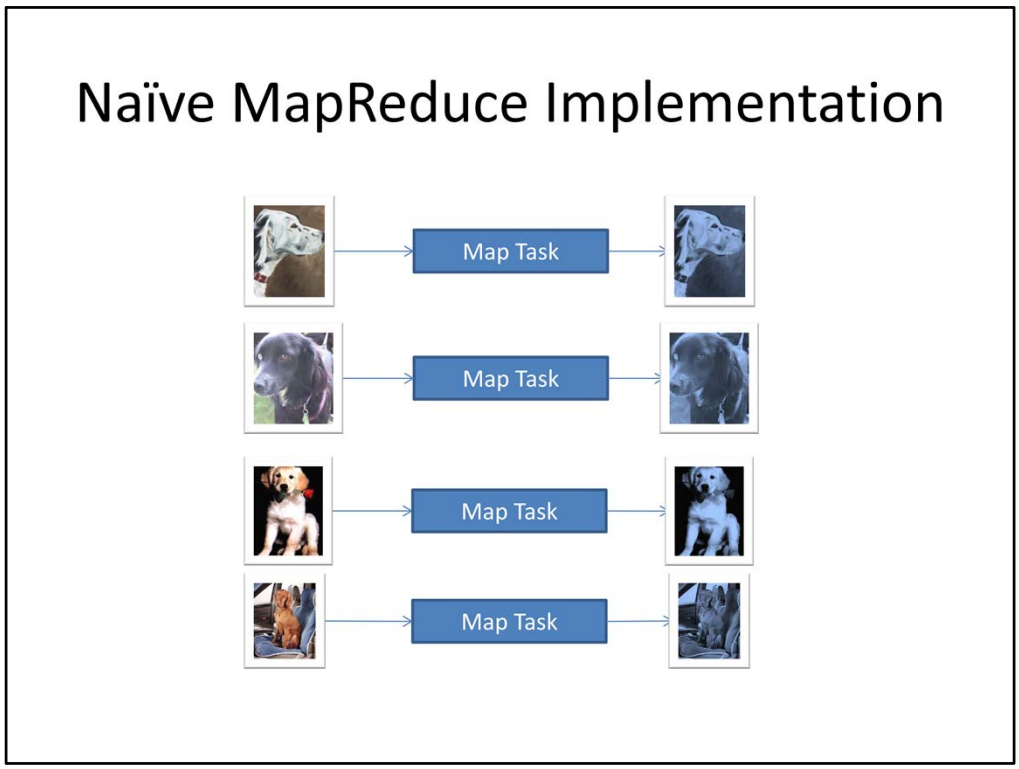
- Parallelize operations within an image.



- Parallelize operations across multiple images.

There are 2 ways to speed up image processing. For large images, the operations within an image can be parallelized. In this technique, a large image would be split into smaller subimages and the image processing operation would be applied to the sub-images in parallel. The sub-images would then have to be stitched together to get the final, processed image.

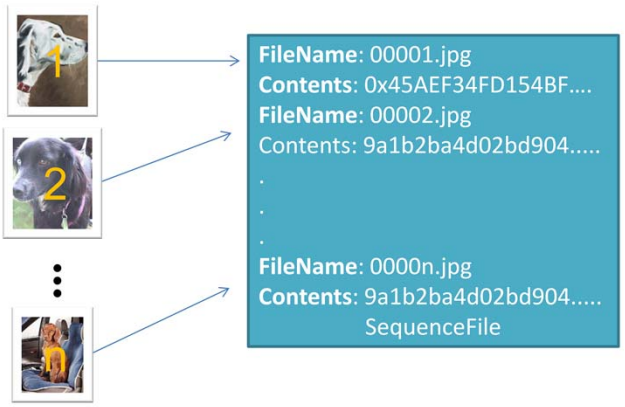
The other strategy is to parallelize operations across images. This is a feasible approach when there size of an individual image is small, but there are a large number of distinct images to process.



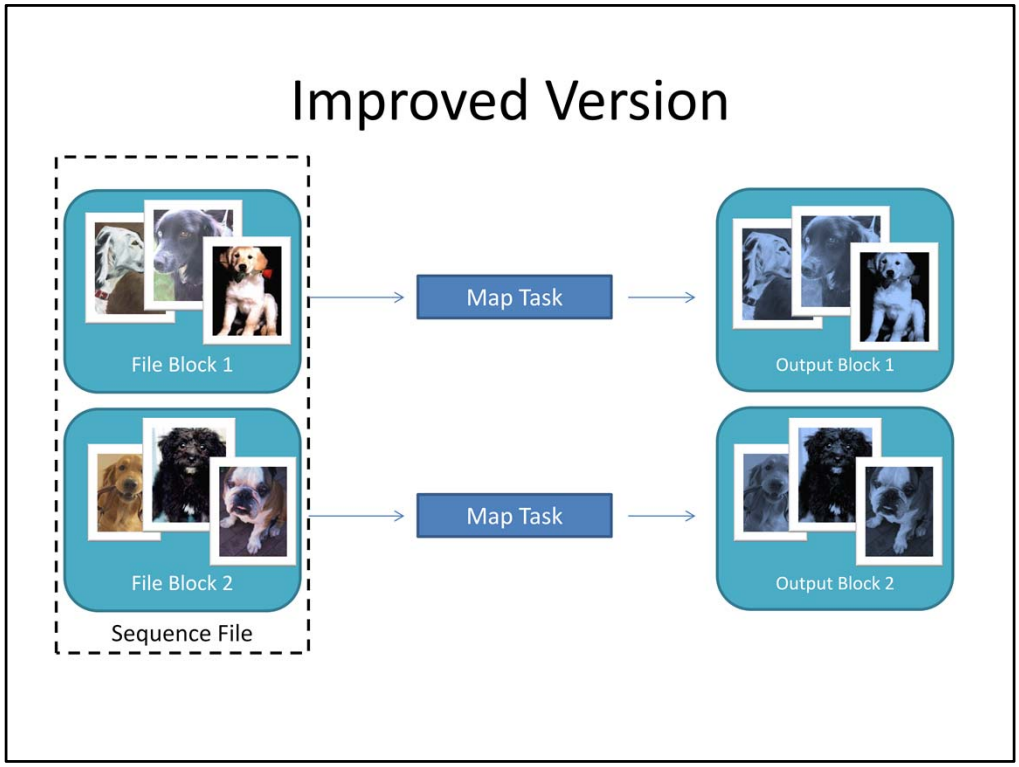
Lets look at a Naïve MapReduce implementation of an image processing system in MapReduce. In the naïve implementation, we send each image file to its own map task which then processes each image idenpendently and produces the output. Please not that a reducer is not required here as the computation can be applied on each image independently.

Improving Efficiency

- Large Number of Mappers
- HDFS is inefficient for small files.
- Use SequenceFile



There are a couple of problems with this approach. When we have a large number of small images, say 1 Million images, we have 1 million map tasks, which may take a substantial amount of time to finish on a cluster. Each map task comes with overheads , which may actually exceed the actual computation required. In addition, HDFS is not efficient in dealing with small hence. Hence, to speed up the computation there should be some way to combine multiple files into a single file. Hadoop makes availabile a special format called SequenceFile which can be used to combine multiple files into a single file.



Hence, a sequence file that contains all the images to be processed gets split into individual blocks by HDFS. These blocks will be sent to individual mappers, which can iteratively process multiple images and write them to disk. This technique can greatly speed up the processing of images, at the expense of some preprocessing where the images have to be combined into a sequence file.

Example: Sobel Edge Detection

```
public static class ImagePMapper extends Mapper<Text,
BytesWritable, Text, BytesWritable>{

    //Sobel Kernels
    float[] xKernel = {
    -1.0f, 0.0f, 1.0f,
    -2.0f, 0.0f, 2.0f,
    -1.0f, 0.0f, 1.0f
    };

    float[] yKernel = {
    -1.0f, -2.0f, -1.0f,
    0.0f, 0.0f, 0.0f,
    1.0f, 2.0f, 1.0f
    };
```

As an example, we can look at the Sobel edge detection algorithm implemented in MapReduce. Since this is a map-only application, we will present only the map class, as shown on screen as the “ImagePMapper”.
It expects input in the form of “text”, “byteswritable”, where text corresponds to the filename and byteswritable is the binary contents of the file. The Map outputs are also the same.
We define two float arrays, xKernel and yKernel to be the Sobel operators and these kernels are variables defined within the map class.

Example: Sobel Edge Detection

```
public void map(Text key, BytesWritable value, Context
context) throws IOException, InterruptedException{

//Read Image
InputStream in = new ByteArrayInputStream(value.getBytes());
BufferedImage bImageFromConvert = ImageIO.read(in);

//Perform Sobel Operations on Image
ConvolveOp blurX = new ConvolveOp(new Kernel(3, 3, xKernel));
BufferedImage x = blurX.filter(bImageFromConvert, null);

ConvolveOp blurY = new ConvolveOp(new Kernel(3, 3, yKernel));
BufferedImage y = blurY.filter(x, null);
```

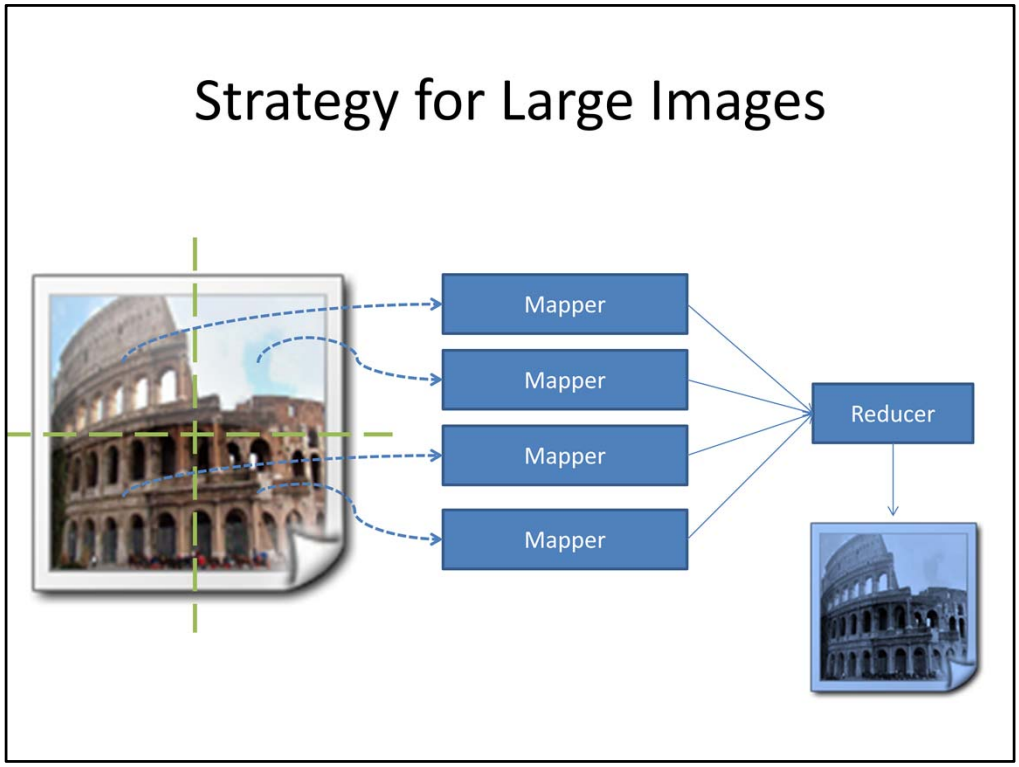
Now we have the map function. The function is simple, we simply take byteswritable value read it as an image. Once we have read it as a BufferedImage object, we apply the two sobel kernels to the image.

Example: Sobel Edge Detection

```
    //Create Output ByteStream
    BytesWritable outputBytes = new BytesWritable();
    ByteArrayOutputStream baos = new
    ByteArrayOutputStream();
    ImageIO.write( y, format, baos );
    baos.flush();
    byte[] imageInByte = baos.toByteArray();
    baos.close();
    outputBytes.set(imageInByte, 0, imageInByte.length);

    //Send output key,value pairs.
    context.write(key, outputBytes);
    }
}
```

The final operations within the map function simply assemble the output value in the form of a byteswritable, and it is written as the output of the map operation.



There is also the case where an Image is too large to fit within a single HDFS block, and hence spans multiple blocks. In this case, we have split the image into multiple sub-images and assign each sub-image to a mapper. Once the mappers have processed their assigned sub-image, they can send the data along with a relevant key to the reducer which can assemble the final image.