15-319 Introduction to Cloud Computing

Project 2
Large-Scale Data Processing using Hadoop

Assigned Date: **January 28th, 2010**
Deadline: **February 18th, 2010** at 11:59 p.m.

Goals:

1. Write Serial, Threaded and Hadoop Java code to process large amounts of data
2. Understand parallelization and its limits in various programming models
3. Get an appreciation of the benefits offered by models like MapReduce in solving such large-scale problems on distributed hardware

Background:

You are working for a top-secret intelligence organization. We have obtained some surveillance footage that is to be shared with a rival organization, but the problem is that a few of our double agents are visible in this footage and we need to conceal their identity before we send the footage across to our rival. You will have to process and blur out our agents in each frame of this surveillance video. Time is of the essence!

The surveillance footage is available in <u>/afs/qatar.cmu.edu/course/15/319/data/</u> Process the data according to the instructions below. Fortunately our double agents are seated the whole time so we just need to blur a segment of each image in the video. After identifying the agents, you may input the static image coordinates to your program to blur that portion of each image. You may refer to the sample images included in this dossier for more information.

More information on the recommended APIs and tools that can be used is attached as an appendix. You are free to use any Java API or external library to complete this project.

Part I: Decompose the video into individual JPEG frames

1. The input video is located in **/afs/qatar.cmu.edu/course/15/319/data**
2. Use the **ffmpeg** program in **hadoop.qatar.cmu.edu** to decompose the video into its constituent frames. (Refer to the Appendix for instructions on how to use **ffmpeg)**

3. You have been assigned a workspace area in **/afs/qatar.cmu.edu/course/15/319/workspace**, use this area to work with these large files.

Part II: Serial Image Processing Code

1. Write a serial java application that takes each image generated in Part I and blurs the required area in each frame.
2. Process all frames of the video using your java application on **unix.qatar.cmu.edu** and record the duration to complete this task.
3. **Deliverable:** Submit a processed JPEG frame taken from the video in a folder (**part1**) along with a text file containing your code and the duration for your code to process all the frames as a comment in your program.

Part III: Java Multithreaded Image Processing Code

1. Attempt to speed up your code by writing a multithreaded version of the program you have written in Part I.
2. You may use the standard Java concurrency libraries to accomplish this task.
3. Run your code on **unix.qatar.cmu.edu** on all the frames of the video, and record the time taken to process the entire video.
4. **Deliverable:** Submit a processed JPEG frame taken from the video in a folder (**part2**) along with a text file containing your code and the time taken for your code to process all the frames as a comment in your program.

Part IV: Hadoop Image Processing Code

1. Read the attached Appendix to learn about Hadoop Sequence Files.
2. Put all the JPEG frames of the video into a TAR archive
3. Use the **tar-to-seq.jar** application located in **/afs/qatar.cmu.edu/course/15/319/workspace/tools/** to convert your TAR file into a Hadoop Sequence File and upload it to your cloud.
4. The Hadoop skeleton code (**BlurVideoHadoopProject**) for loading each file from this sequence file is located in **/afs/qatar.cmu.edu/course/15/319/workspace/tools/.** Import this folder as an eclipse project into your eclipse workspace.
5. Complete the Hadoop program by filling in the required areas inside the map function (as marked using comments).
6. Package the code into a Hadoop JAR file (along with your external libraries). Run your program using the **hadoop jar** command on your cloud.
7. Use the **SeqToFile.jar** Hadoop program to unpack your output files into individual JPEG images (located in **/afs/qatar.cmu.edu/course/15/319/workspace/tools/**)

8. **Deliverable:** Submit a processed JPEG frame taken from the video. Export your Hadoop project from Eclipse as a Folder. Save the JobTracker HTML page that shows your Hadoop project running to completion (see sample JobTracker Output). Place all of these inside a folder called **part3**.

Submission:

Hand in a project write-up discussing the problem and the three methods used. Make sure to include a results section along with a performance comparison of the three methods including a bar graph. Use the 2-column ACM format for this paper (abstract, problem definition, methods used, results and comparison, conclusions and references). Creative extensions to the three methods discussed above or a detailed comparison section will lead to a **bonus**. Including a face-detection module will lead to a much **greater bonus**.

The person with the fastest serial, threaded and Hadoop code (when we run it) will get a **bonus**.

Add the write-up and all the deliverable folders from each part into a single zip file (project2.zip) and place it in:

```
/afs/qatar.cmu.edu/course/15/319/handins/username/
```

This file is to be submitted once and the final timestamp on the server will determine your submission time.

Grading:

As mentioned in the syllabus, this project is worth 15% of your final grade. You have two weeks to finish the project.

Project 2 Appendix

General Notes: It is advisable to perform the following tasks from the **/afs/qatar.cmu.edu/course/15/319/workspace/userid** folder as we have setup enough disk quotas for all the users there.

Decomposing a Video into Image Frames

A great utility for breaking a video file down into its individual images is **ffmpeg**. ffmpeg is an open-source video encoder/decoder application that can virtually convert to/from any media format. It can even be used to transcode video for iPods and other devices.

To split a video file (video.mpg) into individual image files (image1.jpg, image2.jpg…) use the following command:

```
ffmpeg -i video.mpg image%d.jpg
```

To create a video file (video.mpg) from individual image files (image1.jpg, image2.jpg…) use the following command:

```
ffmpeg -f image2 -i image%d.jpg video.mpg
```

Refer to http://www.nerdlogger.com/2008/09/ffmpeg-command-line-quickies.html for more cool stuff that you can do with ffmpeg.

Image Operations in Java

Java AWT Image Libraries allow image handling. You may use the *ImageIO* class to open image files and store them as a *BufferedImage* Object.

A good external library that can be used to blur individual images is the JHlabs imaging filters. The JAR file containing various imaging filter implementations in Java is available at http://www.jhlabs.com/ip/filters/Filters.zip

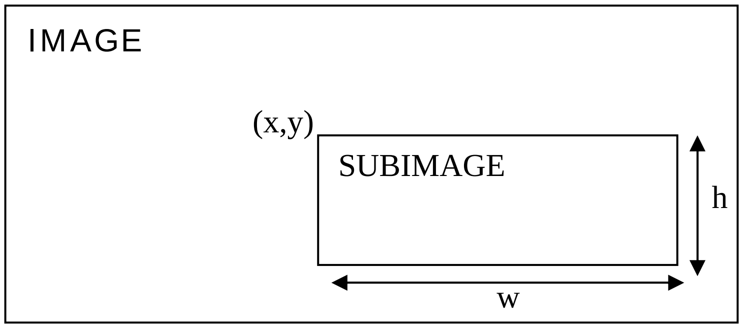
We recommend that you use the following *BoxBlur* Filter in your application. The following code fragment opens a file "*filename*" to perform a blur operation using the value 10 as the horizontal and vertical radii and iterates the blur operations twice.

```
BufferedImage img = ImageIO.read(new File(filename));

BufferedImage dest = new BufferedImage( img.getWidth(),
img.getHeight(), img.getType());

BoxBlurFilter BlurOp = new BoxBlurFilter(10,10,2);
dest= BlurOp.filter(img,dest);
```

```
ImageIO.write(dest,"jpeg",output);
```

Visit http://www.jhlabs.com/ip/blurring.html for more information on these image filters.

Blurring a Portion of an Image



Blurring a portion of an image is a little trickier. First, you will need the *x,y* coordinates of the top left corner of this portion (which we shall call *subimage*). You will also need the width *w* and height *h* of the *subimage*. Once you have this information, you can create a new *BufferedImage* with the contents of the *subimage*. Once we blur the *subimage*, we then use the *Grapchics2D* java libraries to overlay this new *subimage* onto our destination image. The following code fragment blurs a portion of *img* and stores it in *dest*.

```
BufferedImage subimg = img.getSubimage(x, y, w, h);

//Perform BlurOperation
BoxBlurFilter BlurOp = new BoxBlurFilter(10,10,2);
subimg= BlurOp.filter(subimg, subimg);

//Overlay the Image on top of original
Graphics2D g=dest.createGraphics();
g.drawImage(img, null, 0, 0);
g.drawImage(subimg,null,x,y);
```

Refer to the Java Graphics libraries for more information.
http://java.sun.com/docs/books/tutorial/2d/images/

Multithreaded Programming in Java

Multithreaded programming in Java is accomplished using the Java concurrency library. It is based on the fork-join threading model. We recommend that you parallelize your java code by launching several threads from within the program. Each thread could work on a part of the input to be processed. You will need to write a class that performs the required processing as a subclass of *Runnable* in java. The following code skeleton will be useful in writing your threaded java application:

```
import java.io.File;
import java.io.IOException;
import java.lang.Thread;


public class SampleThreadedApp
{
  //Class Thread for Blurring a single image read from filename
  private static class SampleThread implements Runnable
  {
      private int i;
      private int start;
      private int num;

      //Constructor to Initalize thread variables
      SampleThread(int value, int s, int n)
      {
        i=value;
        start=s;
        num=n;
      }

      //Every class implementing Runnable must implement the run()
method
      public void run()
      {
         //Get the start point and endpoint for this thread's work

         //Perform Processing
         //...
         //...
      }
   }
   //End SampleThreadedApp


   //Main Program
   public static void main(String[] args)
   {

      //Set total number of Threads
      int numThreads = 4;

      //Create thread instances
      Thread t[] = new Thread[numThreads];
```

```
        //Figure out the number of items each thread must process
        int thread_size=total_size/numThreads;

        //Fork all Threads
        for(int i=0; i<numThreads; i++)
        {
              //Initalize each thread with it's startpoint and end point
              t[i]= new Thread(new  BlurThread(i,i*size,size));
              //Fork the Thread!
              t[i].start();
        }

        //Join all Threads after they have finished processing
        for(int i=0; i<numThreads; i++)
        {
              t[i].join();
        }

        //Stop Clock, Print Timing Messages
        long end= System.currentTimeMillis();
        System.out.println("Image Processing Complete!");
        System.out.println("Time Taken: "+ (end-start)/(1000*60) +
                            "minutes");
      }

}
```

A quick Tutorial on multithreaded programming and concurrency in Java is available at http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

Hadoop Sequence Files

Hadoop excels in processing large files. The default HDFS block size is about 64 MB. On the other end of the spectrum, image files that are a few KB in size will not work very well as input to Hadoop programs. Having a large number of small files also puts additional strain on the HDFS name node as it has to index all the files in RAM. One way to get around this is to merge all the images as a single sequence file.

Hadoop has an input file type that is useful in such circumstances called *SequenceFile*. A sequence file consists of sequence of key-value pairs. In our case, the key will be the file name and the value will be the file contents. For the purpose of this project, you may use a SequenceFile to package the individual JPEG frames of the video. Once you have processed all the frames in Hadoop, the output will be a bunch of SequenceFiles.

A SequenceFile generator progam (**tar-to-seq.jar**) is located in **/afs/qatar.cmu.edu/course/15/319/workspace/tools/.** You will need to create a TAR of all the input JPEG files first and then use the program. Use the following commands from within the directory that contains all the JPEG files:

```
tar –cf video.tar *.jpg
java -jar tar-to-seq.jar video.tar video.seq
```

This file can now be uploaded to your HDFS store and you may use it as the input to your Hadoop program. Your Hadoop program will produce SequenceFiles as an output as well.

You may use the provided ***SeqToFile*** java program to unpack the output sequence files and obtain the individual JPEG images.

```
hadoop jar seqtofile.jar /user/hadoop/blur/out/part-00000
```

You may repeat this command for each output file that you obtain from Hadoop.

Page 103 in the textbook "*Hadoop-The Definitve Guide*" discusses SequenceFiles in detail.

Adding an External JAR Library to your Hadoop Project

In order to run a Hadoop job on your cloud, the Hadoop code is packaged into a JAR and distributed among the nodes in your cloud. If you are using an external library, it must be included in the Buildpath and packaged into the final Hadoop JAR file. To do this in eclipse:

1. Place your external library JAR file(s) into your project folder in directory *lib*
2. Place another copy of the JAR file(s) outside your project folder
3. In your project properties, add an External JAR to your build path and point it to the JAR files(s) located outside your project folder.
4. Export your project as a JAR and make sure you include the *lib* folder which contain your external JARs

You may not be able to run such code directly from Eclipse. If you face any such trouble, try running the Hadoop job from command line using the command *hadoop jar*.