Documentation

Title: Nonphotorealistic rendering by openCV on Hadoop
Auther: LeiYu, HshiFei
Advisor: Dr.Null

Introduction

Background

Non-photorealistic rendering is a new technique in computer graphic area. In the early time, most of computer graphic search is focused on photorealism, which is reproducing an image as realistically as possible. In contrast to photorealism, non-photorealistic rendering is a process that try to produce an image looks like cartoon, painting by certain rendering algorithms. Non-photorealistic rendering is widely used in different fields, such as scientific visualization, architecture illustration, movies and so on. However, as general, a large sequence of frames require a lot of times and computations. Therefore, we hereby build a Hadoop cluster for computation to decrease computing cost. Hadoop is an open-source framework used for storing data and running application on clusters of commodity hardware. The core of Hadoop contains Hadoop Distributed File System (HDFS), which is a system that stores data across many machines without prior organization and MapReduce programming model, which is a parallel processing software framework. It splits files into large blocks and distributes them across nodes in a cluster, then transfers packaged code into nodes to process the data in parallel.
In our project, we built Hadoop cluster on our own computers and have one master computer and two slave computers. We built it via Java API to create HDFS on several machines and use map-reduce mechanism to assign task for computation. We also developed an image processing application by JavaCV, which provides utility classes to make developing functionality easier on Java platform in computer vision field, such as OpenCV.

Project's Goal

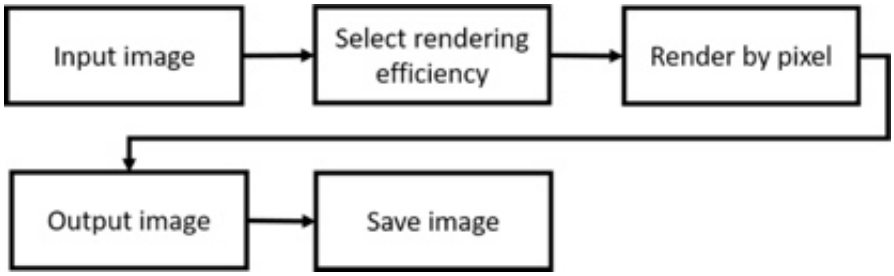
As above mentions, rendering a large sequence or a huge size of image requires many times and computation. Thus, the overall goal of this project is developing an image processing application based on Hadoop and get output images in real time.
First, in order to achieve the overall goal, we have to find a functional algorithm to do image rendering. In this phase, we need to figure out what efficiencies we want to develop, understand the concepts of these efficiencies and how to implement programs for them.
Second, we must build a Hadoop cluster on multiple machines. We need to study knowledge of Hadoop, install Hadoop on our own computers and virtual machines and successfully run simple application on it. We also need some Python and Shell script to configure the system.
At the end, we have to run a map-reduce application on Hadoop cluster and run our image processing

application on it. In this part, we have to realize the concept of map-reduce and think how to run our application via it. Then, we must figure out how to store an image in text file and how to split it.

Image Rendering Algorithm Desgin

We can separate this project in two level, one is base level which is our Hadoop cluster and another one is application level which is our image rendering application.

In application level, we tried to develop six rendering efficiencies, vintage, sketch, comic, cartoon, oil-painting and lomo, by JavaCV. Because of data of image stored in text files, we decided to render each pixel of image respectively. The entire working flow of the application is:



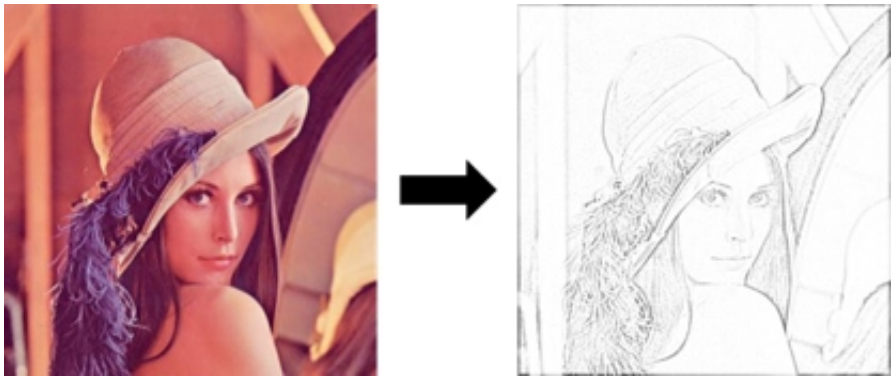
Vintage Style

Vintage filter can create a beautiful, nostalgic photo. In order to making our photo more old-fashion, we have to change colors more yellow and gray. We modify each pixel by the below formula:

Sketch Style

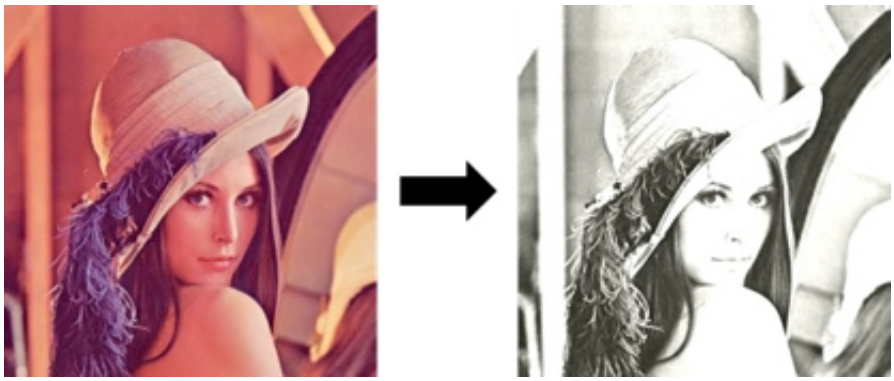
Sketch is a freehand drawing that is executed rapidly and is not usually intended as a finished work. We use the following steps to implement sketch filter.

1.Convert RGB image to grayscale image
2.Get invert color of grayscale image
3.Blur inverted image by Gaussian Blur algorithm
4.Combine blurred image and grayed image



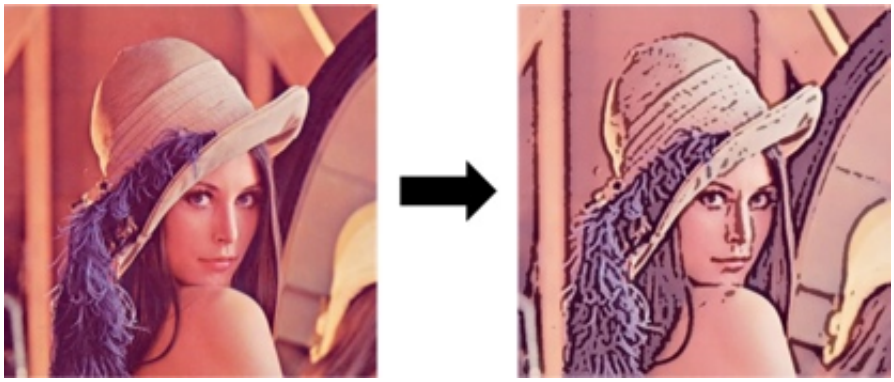
Comic Style

Comic is our childhood memory. In that time, most of comic is black and white. Thus, we develop this filter by the below steps and formula:
1.Get new R, G, B value by following formula:
2.Calculate this pixel's gray value
3.Modify values of R, G, B by following formula:



Cartoon Style

Cartoon filter is an artistic style which make image more satire, caricature, and humor. The implementation steps are show below:
1.Transform RGB image to grayscale
2.Enhance edges of grayed image by setting adaptive threshold
3.Filter colors of original image by bilateral filter
4.Combine enhanced image and filtered image



Oil-painting

Oil-painting is a process of painting with pigments with drying oil as binder. Oil-painting looks blurry, atomized. The following steps the idea to develop this filter efficiency.
1.Initial a brush which contains three parameters, brush length, brush width and brush theta.
2.Map each net cell to image's pixel and obtain the color
3.Set brush by following formula:

```
point1.x= i+brush.BrushLength/2*cos(brush.BrushTheta)+
brush.BrushWidth/2*sin(brush.BrushTheta)
point1.y=j-brush.BrushLength/2*sin(brush.BrushTheta)+
brush.BrushWidth/2*cos(brush.BrushTheta)
point2.x=i-brush.BrushLength/2*cos(brush.BrushTheta)
brush.BrushWidth/2*sin(brush.BrushTheta)
point2.y=j+brush.BrushLength/2*sin(brush.BrushTheta)-
brush.BrushWidth/2*cos(brush.BrushTheta)
```

4.Render each pixel by brush

LOMO Style

Lomography represents a photography experience which is free and unbounded. The characteristics of Lomography are blur, reach color. We develop this efficiency by below steps:
1.Lighten image and exclude colors
2.Blur lightened and excluded image by Gaussian blur
3.Add dark corner in blurred image

Hadoop Cluster Build

In order to implement our computation, a hadoop cluster is required. We hereby build a hadoop cluster of

3 local machines, 2 virtual machines of ubuntu 12.04 and 1 raspberrypi of its own linux os.

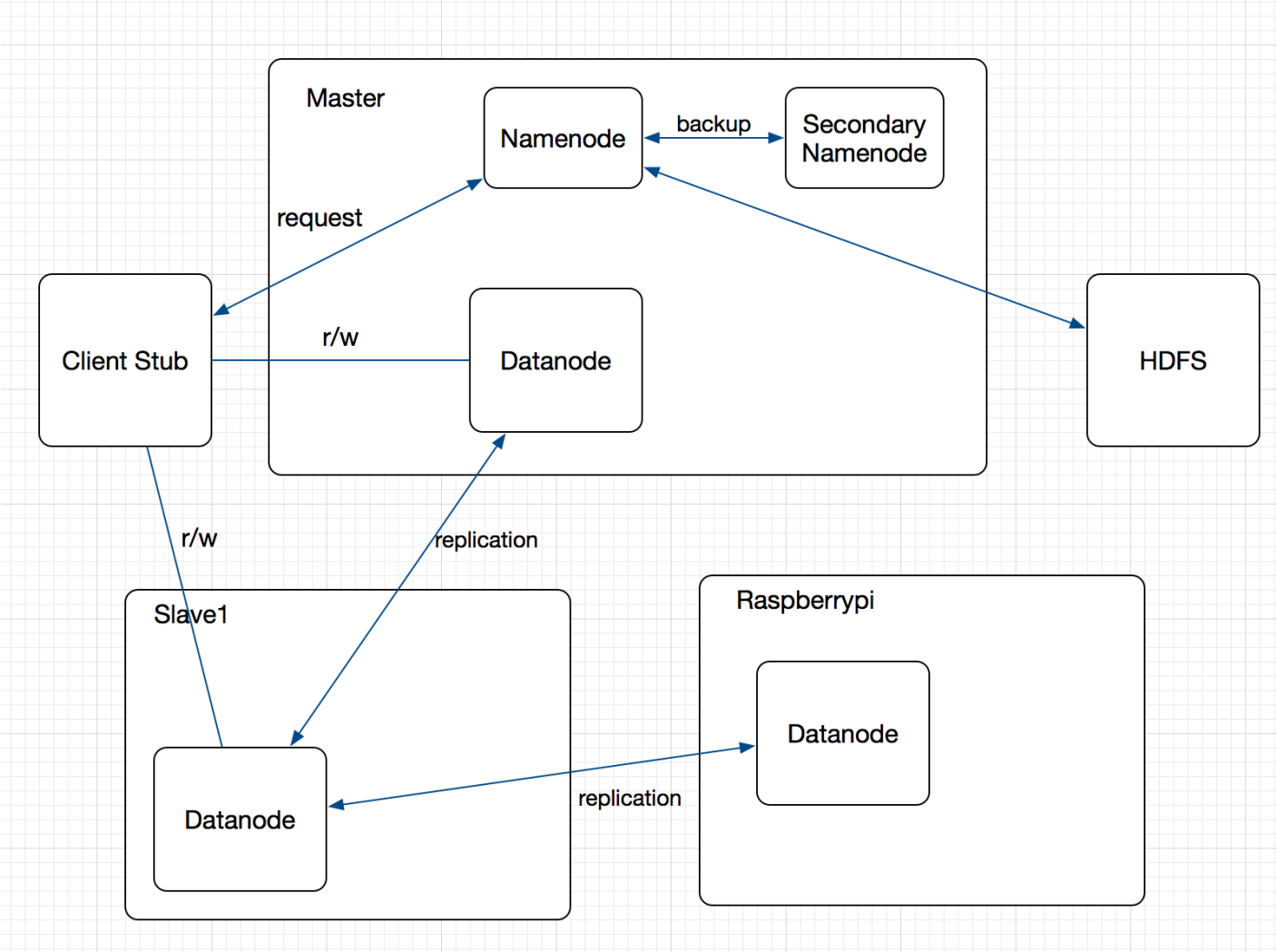
Cluster Design

	OS	Hadoop Roles
Master	ubuntu 12.04	Namenode, Datanode, ResourceManager, NodeManager, SecondaryNamenode
Slave1	ubuntu 12.04	Datanode, NodeManager
Raspberrypi	raspbian	Datanode, NodeManager

Table 1. Cluster Cast

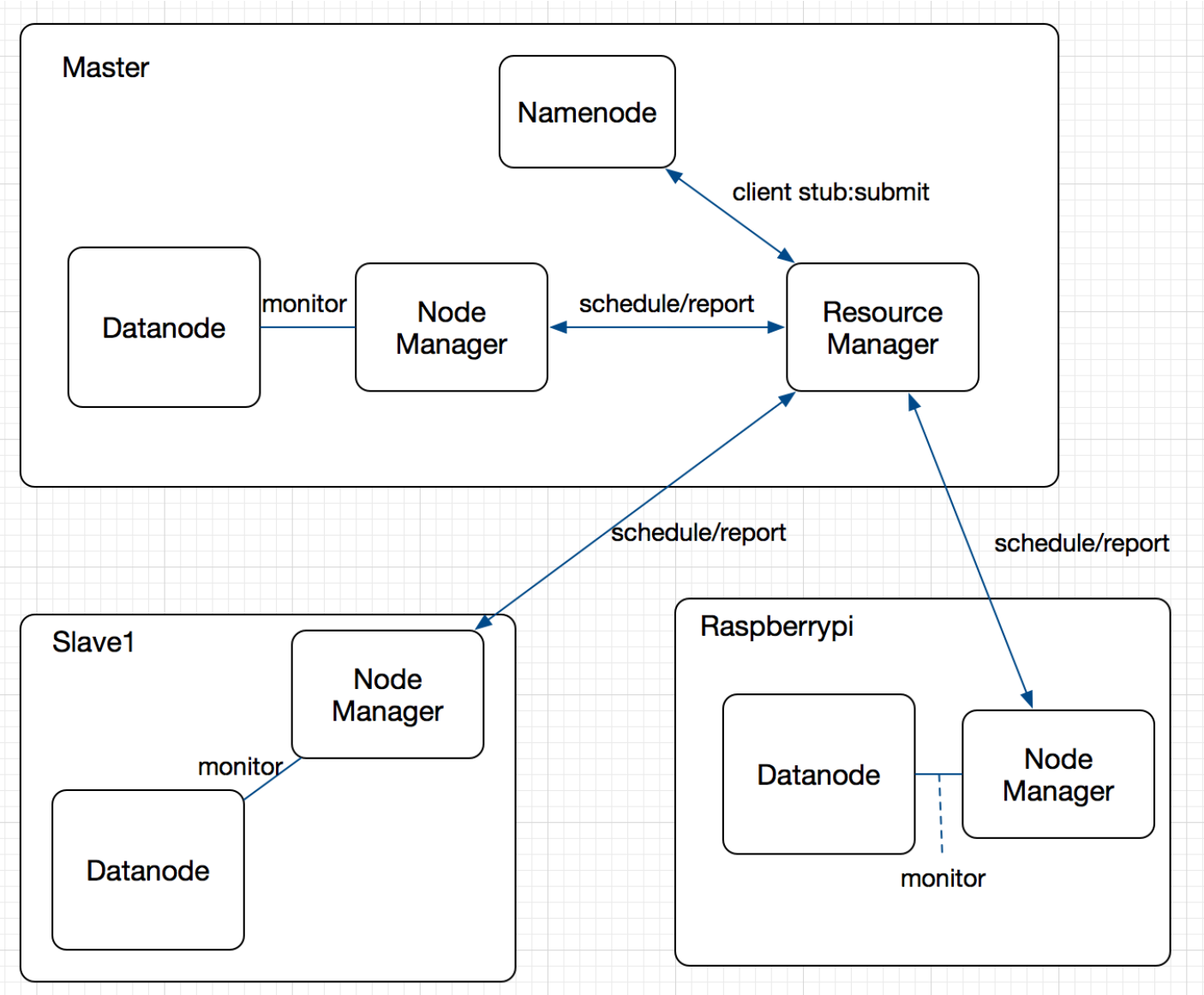
Design Issues:
Usually in order to keep the safety of Namenode, a standalone machine is used exclusively for the Namenode. However in our simple case, with 3 Datanodes in the cluster, the ResourceManager can schedule the task on each machine, making full use of the cluster.
SecondaryNamenode should usually be set up on another machine other than the Namenode. However in our simple case, as the local network is over a low-bandwidth wifi, we just set it up on the same machine of the Namenode.

Functionality of the Cluster



Graph1. HDFS r/w Functionnality

The HDFS system of our cluster works as the graph above. Upon client request, the Namenode will handle the file system metadata registry, asking the HDFS for approval. As Namenode succeeds in returning the permission reply(with the actual data location), the client itself will access the correct machine where data locates and implement r/w.
Data replication is maintained between Datanodes.



Graph2. Job Commit Functionality

ResourceManger plays as a global coordinator for the mapreduce task in the hadoop cluster. The brief graph shows the basical idea of our cluster's job commit system the the ResourceManger automatically assign the task to available machine(NodeManager). And the NodeManager will monitor the implementation condition of the local task and report to ResourceManager.
Upon local failure or success, the report will make the ResourceManager reassign the failed task or record the successful task.
Actually in hadoop 2.0 there are more roles in the background, like AppMaster and Container, but here just for simplicity, we don't think of talking about it.

XML Configuration

Some basic configurations are set in xml files in hadoop root directory which is automatically read when

we launch the cluster.

XML path for hadoop: `$HADOOP_HOME/etc/hadoop/$`

```
//core-site.xml
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://master:54310</value>
</property>
```

core-site.xml sets the main entry of the hdfs access, we assign the name field `fs.defaultFS` by `hdfs://master:54310`, so any hdfs requests will be accessed through this address.

```
//hdfs-site.xml
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file:/home/hadoop/app/name</value>
</property>
<property>
  <name>dfs.replication</name>
  <value>2</value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file:/home/hadoop/app/data</value>
</property>
<property>
  <name>dfs.namenode.secondary.http-address</name>
  <value>master:50090</value>
</property>
```

hdfs-site.xml deals with all hdfs related settings.
`dfs.namenode.name.dir` sets the local path for metadata managed by Namenode.
`dfs.datanode.data.dir` sets the local path for actual data managed by DataNode.
And we also set the replica of data by 2 so each data will be copied in to one more backup.
And we also open the 50090 port on master for SecondaryNamenode access.

```
//mapred-site.xml
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
<property>
    <name>mapreduce.jobhistory.address</name>
    <value>master:10020</value>
</property>
<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>master:19888</value>
</property>
<property>
    <name>mapreduce.jobtracker.address</name>
    <value>master:9001</value>
</property>
```

mapred-site.xml deals with all mapreduce related settings.
We use yarn framework for resource monioring by web application.

```
//yarn-site.xml
<property>
    <name>yarn.resourcemanager.hostname</name>
    <value>master</value>
</property>
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>

<property>
    <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

In yarn-site.xml, we set the nodemanager's aux-services by `org.apache.hadoop.mapred.ShuffleHandler`, so there will be a default shuffle for our mapreduce task as we do not make any programming on the shuffle algorithm.

```
//slaves
master
slave1
raspberrypi
```

The slaves file in the path names all the machines running datanode.

Local Settings

For different machines in the cluster to access each other without problems via ssh. We need make same settings locally on each machine.

```
#/etc/host
192.168.1.149 master
192.168.1.102 raspberrypi
192.168.1.103 slave1
```

We should edit the host name for each ip in our local network. Here is just an assignment of ip address for my machines.
(For virtual machines, bridged network should be used so that they can be assigned an ip address in the same network as the host)

```
#~/.profile
export HADOOP_HOME=/home/hadoop/hadoop
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
export PATH="/opt/gradle-3.4.1/bin:$JAVA_HOME/bin:$HADOOP_HOME/bin:$PATH"
```

And we also need to add some path in the system environment for hadoop to run.

Cluster Initialization

We hereby run the cluster by hadoop shell command.

```
$ hadoop namenode -format
```

This command just initialize the hdfs, formatting both the namenode and datanode.

```
$ ~/hadoop/sbin/start-all.sh
```

And we launch all the necessary processes of hadoop.

Upon successful launch of hadoop, we can use web interface to monitor and track the implementation of jobs.

Web Interfaces:
Jobhistroy: `master:19888`
Nodescondition: `master:8088`
Logs: `master:50070`

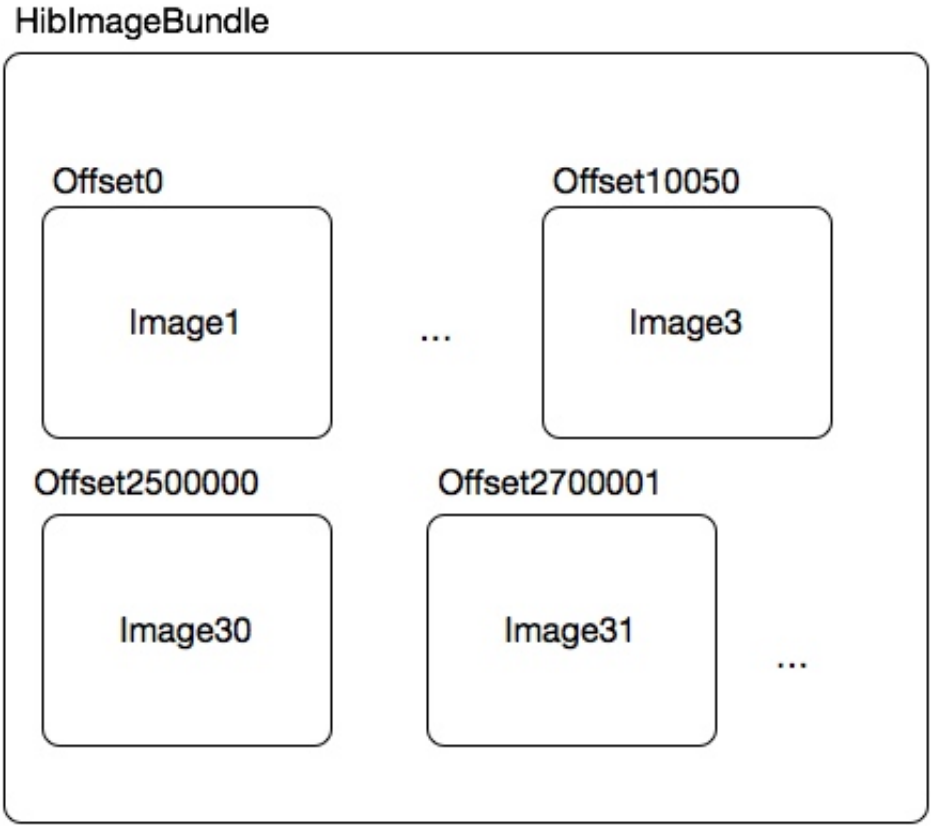
Mapreduce Design

In order to run our application on Hadoop, we need to rewrite the locally one-machine based program into a mapreduce program, where we need to define a new input and output formats of mapreduce.

InputFormat

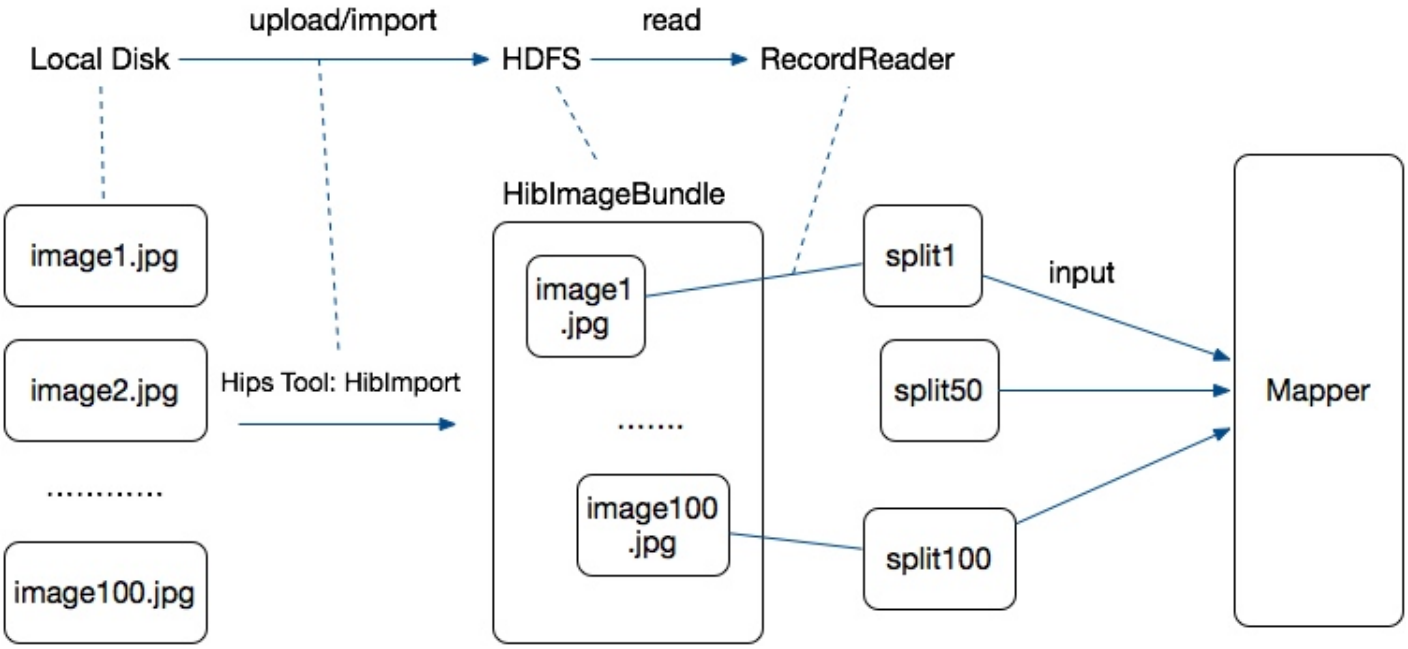
For image input, we use a 3rd party library which is famed for image processing on hadoop.
We use its ImageBundle datastructure for image storing on HDFS.
Note: usually hadoop is good for large data computation where large data means one large files because many small files will increase the cost of I/O.

ImageBundle data structure is as below:



All input images are combined together into one large file, and the location(offset) of each image is stored as well as the metadata. So for hadoop input we just send the HibImageBundle to the RecordReader and the override function with get one exactly one image for each split for each map task.

The input logic for our case is as below:

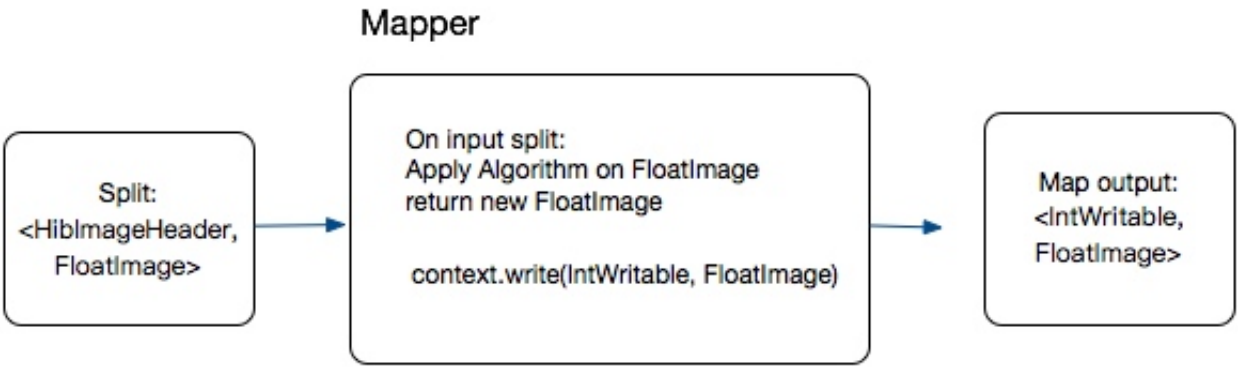


We use a hipi tool, HibImport to upload the several images in local disk to a single large file in HDFS. And the Record Reader will read it into splits for map tasks.

Mapper

Each map task takes an input of one split and make some computation on it and output the key,value pair for reducer tasks.
I/O Specification for Mapper:
Input: < HipiImageHeader, FloatImage >
Output: < IntWritable, FloatImage >
Where HipiImageHeader is the metadata of image and FloatImage is the actual data of the image.
And we apply the image rendering algorithms we showed before on the FloatImage, and assign each processed image with an unique IntWritable number.

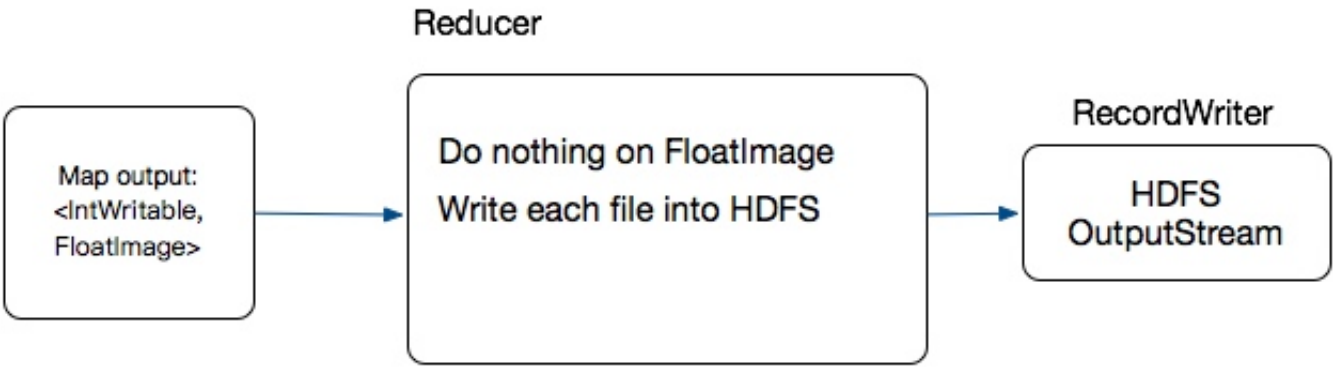
The workflow is showed below:



Reducer

Reducer takes each output of map tasks, resolves it and writes into the outputstream of HDFS.
Here as we just assign each output of map tasks a unique IntWritable, so the iterator in Reducer will only have one image element which is going to be written.

The Reducer in our case works in this way:



OutputFormat

Hadoop's default outputFormat only deals with texts, so in order to write images into HDFS we need write in binary format.

We rewrite the binary output util class, BinaryOutputFormat in Hipi into the MultiImageOutputFormat to output multiple images.

```
public class MultiImageOutputFormat extends BinaryOutputFormat<IntWritable, Float
Image>{
  @Override
  public RecordWriter<IntWritable, FloatImage> getRecordWriter(TaskAttemptContext
 context) throws IOException,InterruptedException{
  //set HDFS OutputStream
  }

  protected static class BinaryRecordWriter extends RecordWriter<IntWritable, Flo
atImage>{
    @Override
    public void write(IntWritable key, FloatImage value){
    //output Image
    }
  }
 }
```

Test

I/O specification:

```
FileInputFormat.setInputPaths(job, new Path("/user/hadoop/VideoSeq.hib"));
FileOutputFormat.setOutputPath(job, new Path("/project1/output"));
```

We use HibImport to upload the images into HDFS as an ImageBundle of 25 images, about 1.5MB.

The result is as below:

```
hadoop@master:~$ hadoop jar  ~/Desktop/test100/Nonphr.jar proj.Nonphr --libjars /
home/hadoop/workspace/HadoopCV/hipi-2.1.0.jar
17/05/01 08:18:25 INFO Configuration.deprecation: mapred.job.tracker is deprecate
d. Instead, use mapreduce.jobtracker.address
17/05/01 08:18:26 INFO client.RMProxy: Connecting to ResourceManager at master/19
2.168.1.149:8032
17/05/01 08:18:27 WARN mapreduce.JobResourceUploader: Hadoop command-line option
parsing not performed. Implement the Tool interface and execute your application
with ToolRunner to remedy this.
17/05/01 08:18:30 INFO input.FileInputFormat: Total input paths to process : 1
Spawned 1map tasks
17/05/01 08:18:31 INFO mapreduce.JobSubmitter: number of splits:1
17/05/01 08:18:32 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_149
3597690450_0001
17/05/01 08:18:34 INFO impl.YarnClientImpl: Submitted application application_149
3597690450_0001
17/05/01 08:18:34 INFO mapreduce.Job: The url to track the job: http://master:808
```

```
8/proxy/application_1493597690450_0001/
17/05/01 08:18:34 INFO mapreduce.Job: Running job: job_1493597690450_0001
17/05/01 08:19:05 INFO mapreduce.Job: Job job_1493597690450_0001 running in uber
mode : false
17/05/01 08:19:06 INFO mapreduce.Job:  map 0% reduce 0%
17/05/01 08:19:16 INFO mapreduce.Job:  map 100% reduce 0%
17/05/01 08:19:28 INFO mapreduce.Job:  map 100% reduce 75%
17/05/01 08:19:31 INFO mapreduce.Job:  map 100% reduce 77%
17/05/01 08:19:34 INFO mapreduce.Job:  map 100% reduce 80%
17/05/01 08:19:37 INFO mapreduce.Job:  map 100% reduce 83%
17/05/01 08:19:40 INFO mapreduce.Job:  map 100% reduce 85%
17/05/01 08:19:44 INFO mapreduce.Job:  map 100% reduce 87%
17/05/01 08:19:47 INFO mapreduce.Job:  map 100% reduce 89%
17/05/01 08:19:50 INFO mapreduce.Job:  map 100% reduce 92%
17/05/01 08:19:53 INFO mapreduce.Job:  map 100% reduce 95%
17/05/01 08:19:56 INFO mapreduce.Job:  map 100% reduce 97%
17/05/01 08:19:59 INFO mapreduce.Job:  map 100% reduce 99%
17/05/01 08:20:02 INFO mapreduce.Job:  map 100% reduce 100%
17/05/01 08:20:13 INFO mapreduce.Job: Job job_1493597690450_0001 completed succes
sfully
17/05/01 08:20:13 INFO mapreduce.Job: Counters: 49
    File System Counters
        FILE: Number of bytes read=69120881
        FILE: Number of bytes written=138479233
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=1481243
        HDFS: Number of bytes written=69120650
        HDFS: Number of read operations=6
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=1
        Launched reduce tasks=1
        Data-local map tasks=1
        Total time spent by all maps in occupied slots (ms)=29256
        Total time spent by all reduces in occupied slots (ms)=213620
        Total time spent by all map tasks (ms)=7314
        Total time spent by all reduce tasks (ms)=53405
        Total vcore-milliseconds taken by all map tasks=7314
        Total vcore-milliseconds taken by all reduce tasks=53405
        Total megabyte-milliseconds taken by all map tasks=7489536
        Total megabyte-milliseconds taken by all reduce tasks=54686720
    Map-Reduce Framework
        Map input records=25
        Map output records=25
        Map output bytes=69120750
        Map output materialized bytes=69120881
        Input split bytes=113
        Combine input records=0
```

```
        Combine output records=0
        Reduce input groups=25
        Reduce shuffle bytes=69120881
        Reduce input records=25
        Reduce output records=25
        Spilled Records=50
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=619
        CPU time spent (ms)=5300
        Physical memory (bytes) snapshot=397631488
        Virtual memory (bytes) snapshot=4449624064
        Total committed heap usage (bytes)=261107712
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=1481130
    File Output Format Counters
        Bytes Written=69120650
```

For more detailed output results, please refer to the result.txt files.

Evaluation

As each image is taken as exactly one split, the number of mapper is the same as the number of images. The Mapper and Reducer work correctly.

CPU time used is 5300ms, which seems very slow for computing only 25 images.
We also tried many times, the CPU time varies from 5 to 25 sec or so, which seems not proper for just computing images, as we only use 100milisecs or so for doing it on the local machine.
More proofs should be verified in the future work.

Fututure Works

Mapper Improvement

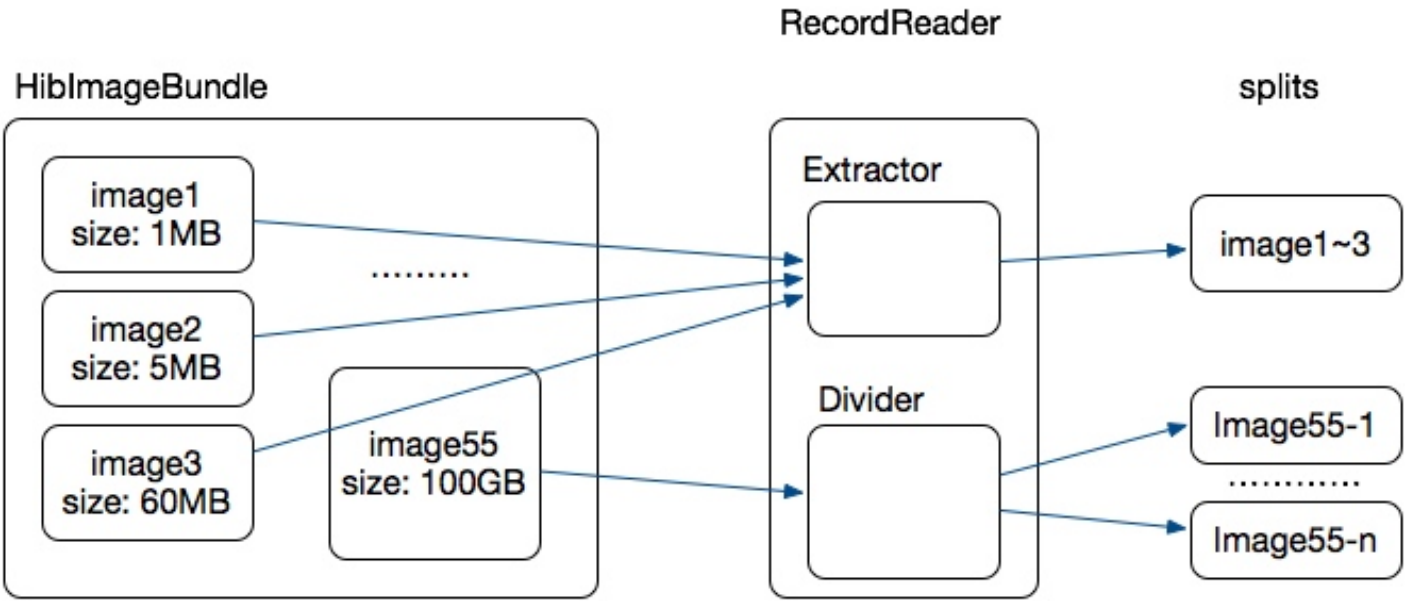
The way we used for the map input can be thoughtfully improved.
In our case, we take an ImageBundle for input, and the RecordReader for the input will read one image for each split for each Map task.

It may works well for small images which is commonly used in our daily life. But note that hadoop block is of 128MB size, which is a proper size for a split.

Usually a jpeg image is of 3-4MB, and a raw image can be 50MB or so.
So in order to make better perfomance, some combination could be used to input multiple images as a split for the map tasks.

On the other hand, for large images like satellite images(one image of GBs), one split can be so big that it should be divided further.

So we should introduce a new frame work for the Recordreader as below:

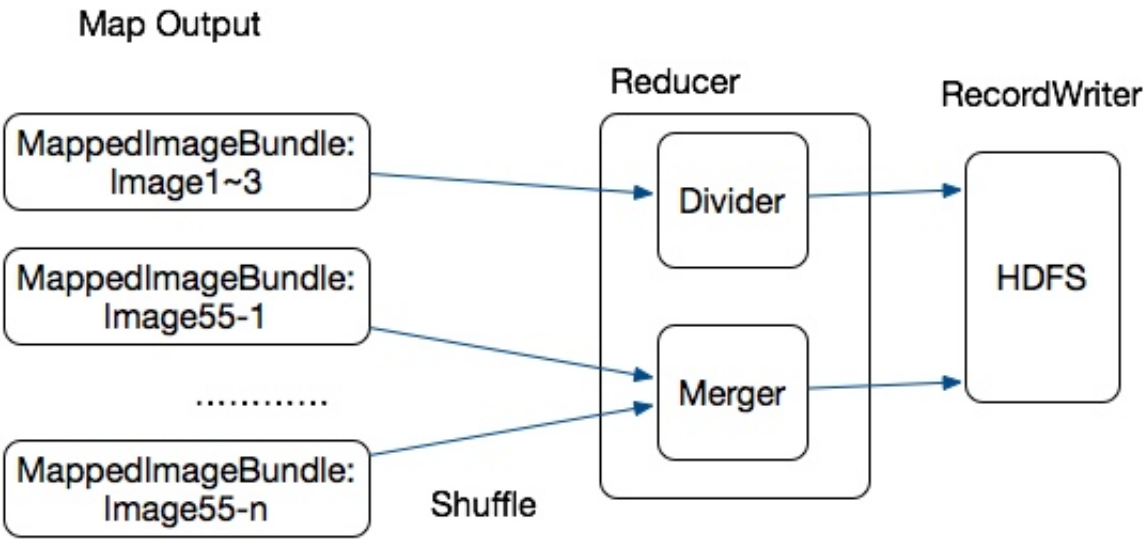


Upon reading the records in HibImageBundle, the RecordReader will detect the size of the image, if it is too small, it will read more images and the Extractor will create a new HibImageBundle for the images read. It the size of image is too large, the Divider will cut the image into parts with formalized naming and create an HibImageBundle, each containing one parts.

Reducer Improvement

The improvement for Reducer works as a reverse of the Mapper.
It takes the outputs of the Mapper, if the output is a sequence of one split large image, the Reducer will merge them together.
If the output is an ImageBundle containing multiple small images, the Reducer will divide them.
And the RecordWriter will write the images into HDFS.

The workflow of Reducer is as below:



Conclusion

Challenging Parts

The most challenging parts for our project is to run image processing on a hadoop cluster.
Few former works can be found online, except one 3rd-party library, hipi.
However hipi itself is not complete as it lacks community experience and the function is limited.
We had to rewrite the OutputFormat for hadoop to output image files on HDFS. And in order to using common image library like opencv in Java development, another 3rd-party lib, javaCV is also applied.
And rewriting a regular program into a mapreduce version requires many works on both knowing hadoop and image processing.

Pitfalls/Dealt

Our case does not rewrite anything about the shuffle algorithm. We just use the default shuffle for data transfer between Mapper and Reducer. For better performance, we should consider it.

We didn't use Reducer much, at least not doing any computation on Reducers. We just use Reducer to collect the outputs of Mappers and write them into HDFS.
In our case, it could add overhead of data transfer, as we could do it only on Mappers.
But for the future design, the Reducer is necessary, as there are Merges and Divisions for the images.

Referrence

hipi: http://hipi.cs.virginia.edu/index.html
hadoop api: http://hadoop.apache.org/docs/r2.7.3/api/index.html
javacv api: http://bytedeco.org/javacv/apidocs/