View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Procedia Computer Science
Volume 29, 2014, Pages 2306–2314
ICCS 2014. 14th International Conference on Computational Science

Productivity frameworks in big data image processing computations - creating photographic mosaics with Hadoop and Scalding

Piotr Szul and Tomasz Bednarz
CSIRO, Australia
piotr.szul@csiro.au, tomasz.bednarz@csiro.au

Abstract
In the last decade, Hadoop has become a de-facto standard framework for big data processing in the industry. Although Hadoop today is primarily applied to textual data, it can be also used to process binary data including images. A number of frameworks have been developed to increase productivity of developing Hadoop based solutions. This paper demonstrates how such a framework (Scalding) can be used to create a concise and efficient solution to a big data image-processing problem of creating photographic mosaics and compares it to a Hadoop API based implementation.

Keywords: image processing, big data, productivity frameworks, hadoop

1 Introduction

In the last decade, big data analytics have become a major decision support factor in the industry and a number of technologies have been developed to both enable and streamline processing of huge volumes and variety of data. The most ubiquitous amongst them is Apache Hadoop [1] - an open-source software framework for storage and large scale processing of data-sets on clusters of commodity hardware that can reliably scale to thousands of nodes and petabytes of data. Around the Hadoop core grew an ecosystem of technologies including among others: workflows (e.g., Oozie [2]), databases (e.g., HBase [3]), machine learning engines (e.g., Mahout [4]) log analysis tools (e.g., Apache Flume [4]) and productivity frameworks.

Hadoop uses MapReduce [6] as its core computational paradigm. This model allows to define a computation in a way that is easy do distribute across a big data cluster, but is based on relatively low level concepts of mappers and reducers. This makes it difficult to apply to complex transformations that require multiple MapReduce steps depending on each other results.

A number of frameworks have been developed to allow easier definition of such complex data processing pipelines, and to increase productivity of developing MapReduce applications. They isolate developers from low-level MapReduce jobs and instead provide flow like abstraction for

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014
© The Authors. Published by Elsevier B.V.

doi:10.1016/j.procs.2014.05.215

computations. The most popular productivity frameworks of this type include: Apache Pig [6], Cascading [4], and Crunch [8].

Some of these frameworks are written in programming languages that support functional programming paradigm and attempt to provide even more abstract functional-like approach to expressing Hadoop computations that closely resembles native functional constructs of the language. These frameworks include Scala-based Scoobi [6], Scrunch and Scalding [7] and Clojure based Cascalog [8] (the latter two are wrappers on Cascading).

Scala [13] based frameworks are most numerous due to the relatively high adoption of Scala and its versatility. Although they differ in significantly in how they internally convert abstract pipelines into an actual sequence of MapReduce jobs, the APIs they expose are relatively similar. For the purpose of this paper, we selected Scalding as it is the most mature framework.

Scalding [7] is a Scala library that makes it easy to specify Hadoop MapReduce jobs. Scalding is built on top of Cascading [4], a Java library that abstracts away low-level Hadoop details. Scalding provides a domain specific language to make MapReduce computations look very similar to Scala's collection API.

Although Hadoop is predominantly used for processing of textual data, there are numerous examples of using Hadoop for processing binary data including images in astronomy [9], life sciences [10] and other areas [11] [16] [18]. In many of these applications however Hadoop is used to run trivial embarrassingly parallel jobs (map-only) that apply the same independent transformation to a set of input images ([10], [11]).

This paper explores how the functional productivity framework such as Scalding can be applied to build a non-trivial image-processing pipeline of creating a photo mosaic which consists of two dependent jobs (a map-reduce and an map-only job), and run it using Hadoop on a large set of input images. It also compares performance and productivity of the Scalding implementation to a baseline version implemented with standard Hadoop API.

The rest of the paper is organized as follows: section 2 describes the two different approaches of creating mosaics including the problem statement. The experimental algorithms, settings, and results are described in section 3. The final section concludes the paper along with the potential future works.

2 Creating Mosaics: MapReduce and Scalding

A photographic mosaic is a picture (usually a photograph) that is divided into (usually equally sized) rectangular sections, each of which is replaced with another photograph that matches the target photo [12].

A mosaic is created by splitting the source image into a grid of smaller sub-images and then finding the best match for each sub-image in a usually large set of library images. The best match functions can range from average color comparison, through histogram comparison to pixel-by-pixel comparison of the images. It is essentially an image search problem of finding closest matches for a number of input images in a set of library images and it becomes a big data problem as the latter one grows.

VideoMosaic Project [11] applied Hadoop to distributed creation of photographic mosaics from frames of vide files. In this project however, the distribution was achieved by independent processing of every video frame with the actual mosaic creation being centralised (all mapper nodes had a full copy of all libray images and a Hadoop map-only job maps each video frame to its corresponding mosaic).

In the next section we are going to present a MapReduce approach and a flow approach for generating photographic mosaics.

2.1 MapReduce approach

Mosaic creation can be relatively easily expressed in the MapReduce paradigm. In the map phase, for each image of the library set, we calculate its distance (or sameness) with all the grid cells of the input image. The result of the map phase can be visualized as heat map with green cells representing a better fit (one heat map per one reference image) as shown in Figure 1.

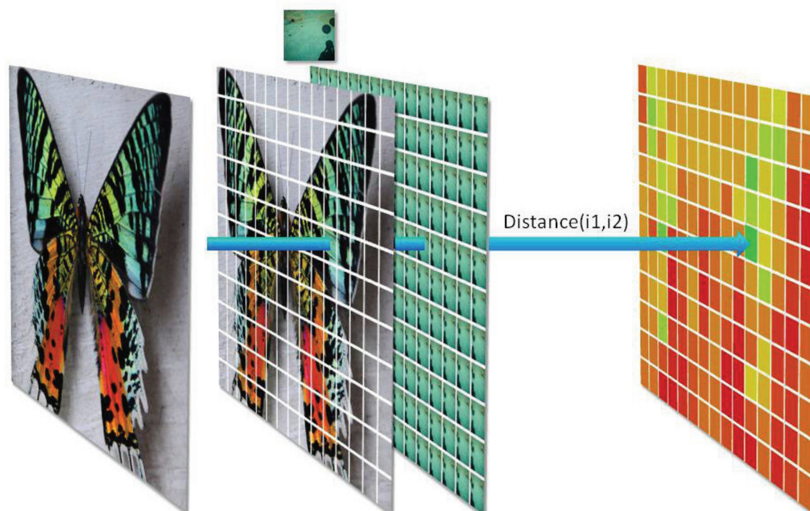

Figure 1: Map phase of mosaic creation

In the reduce phase, we combine all the heat maps and for each grid cell we find the image with the best fit (the greenest one in the heat map) and choose it as the replacement (see Figure 2).

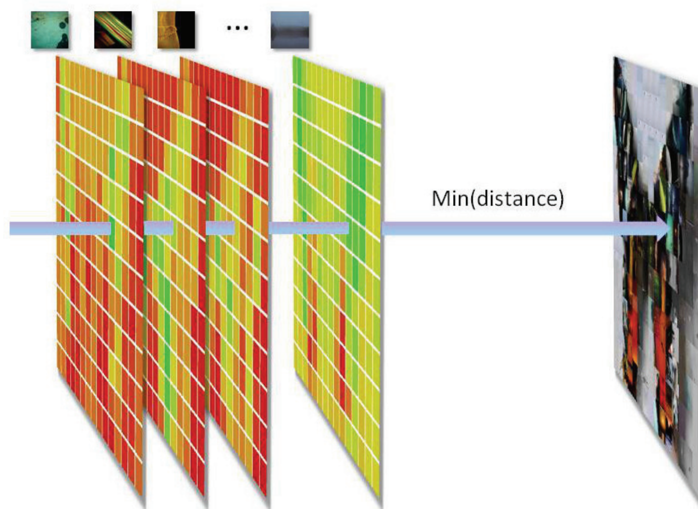

Figure 2: The reduce phase of mosaic creation

This algorithm can be expressed by the following functional (Scala-like) pseudo-code.

Algorithm 1: Functional version of photographic mosaic creation

```
 1:  val sourceImage      // Input image
 2:  val libraryImages    // Collection of library images
 3:  val sizeX, sizeY     // Dimensions of the mosaic grid
 4:
 5:  sourceGrid = (for (x ←(0 until sizeX) ; y ← (0 until sizeY)) yield (x,y))
 6:          . map { p → (p, sourceImage.subimage(p)) }          // Split source image
 7:  mosaicGrid = libraryImages
 8:          .flatmap { i →sourceGrid.map{ (p, subimg) →(p, distance(i, subimg),i) }}
 9:          .groupBy(p)
10:          .reduce { ((d1,i1),(d2,i2)) → if (d1<d2) i1 else i2 }
11:  mosaicGrid.foreach { (p, i) → mosaicImage.subimage(p) = i }    // Render mosaic
```

The **flatMap** operation on *libraryImages* in line 8 corresponds to the map phase of MapReduce, the **groupBy** operation in line 9 to the sort-and-shuffle phase and **reduce** operation[*] in line 10 to the reduce phase.

2.2 Scalding Approach

The MapReduce approach described in the previous section is not very efficient for Hadoop execution, as the output of the map phase (that needs to be written to disk and then copied to reducers) is in the form of *(position, distance, image)* {for all positions in the *sourceGrid* and all images in the *libraryImages*} and thus its size would be approximately *numberOfCells(sourceGrid)* bigger than the initial set of the library images.

To ameliorate this problem, the map phase can output a short unique image identifier rather than the full image, and after the reduce phase, these identifiers can be used to retrieve actual images for every grid cell. This however requires an extra scan of the library image set to perform the retrieval.

In addition, we may want to put some constraints on the images used for mosaic creation (e.g., only full color images and those, which dimensions are larger than the mosaic cell), which some of the library images may not satisfy (e.g., gray-scale images) and thus need to be filtered out before we proceed to the cell distance calculation.

Here is where productivity frameworks, such as Scalding, start showing their value by providing higher level abstraction and shielding the developer from the actual MapReduce jobs required to perform a computation. Scalding model[†] represents a computation as flows of tuples from sources to sinks through pipelines composed of tuple-processing operations. The sources and sinks read and write data in various formats from and to a file system. Scalding provides a number of high level operations such as *map*, *filter*, *groupBy* and *join* that can be used to compose processing pipelines.

The conceptual flow for building photographic mosaics is depicted in Figure 3. The input stream of tuples *(id, image)* is first filtered and then transformed by the pipeline to produce *(pos, min-id)* tuples, which are subsequently joined with the original stream and written as *(pos, image)* to the output. The flow does not have any explicit notion of MapReduce phases but is actually transformed by Scalding into a pipeline of two MapReduce jobs, which we will be referring to as: *BestMatchesSelection* step and *Join* step.

[*] The reduce operation is somewhat simplified here as the groupBy operation in Scala produces a Map of Lists. Please see the source code of name.pszul.mosaic.CreateMosaicFunctional class for the actual implementation.

[†] This model is actually defined by Cascading and Scalding provides Scala based DSL to simplify construction of the flows.

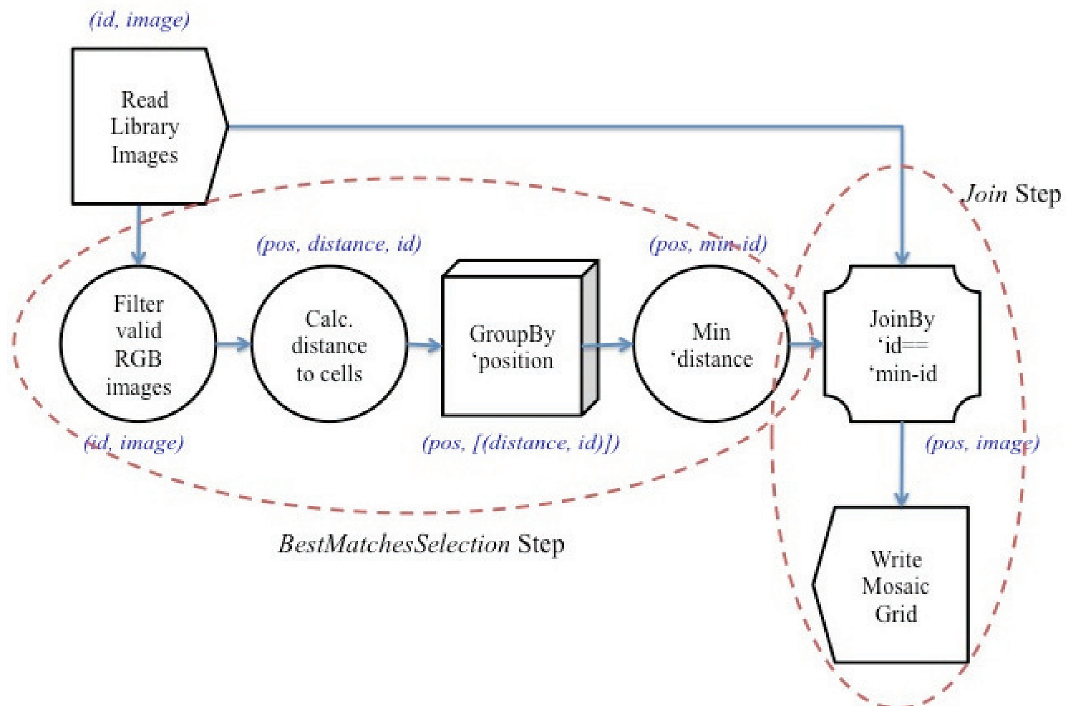

Figure 3: Scalding flow for creating photographic mosaics

3 Experimental Results

The mosaic-building algorithm was implemented in two versions:

1. *MapReduce*(Hadoop): A series of MapReduce jobs (corresponding to *BestMatchesSelection* and *Join* steps) implemented in Java with standard Hadoop API operating on datasets located in the HDFS;
2. *Scalding*(Hadoop): Scalding flow operating on datasets located in the HDFS;

The source code of the programs together with test datasets is available at: https://bitbucket.org/piotrszul/scalding-mosaic. (Please refer to the README file for more information on how to compile the programs and extract the datasets). Image processing operations were implemented using standard Java AWT image API (*java.awt.image.BufferedImage*) and Java Image Filters library [13]. Both *MapReduce* and *Scalding* versions use map side joins in the *Join* step. The *Hadoop* version is using a combiner in the *BestMatchesSelection* step to improve its performance by reducing the number of records passed between the map and reduce phases of the job.

Both *MapReduce* and *Scalding* implementations have been used to generate mosaics for datasets ranging from 9MB to 6GB on a 6-node Hadoop cluster. Each node in the cluster had a single 500GB SAS HDD, 4 x Intel(R) Xeon(R) CPU E5-2660 2.20GHz CPU (4 cores with HT each) and 64GB of RAM, running Ubuntu 12.04 with Claudera CDH4.2.0 (hadoop-0.20). The cluster was configured with 180 map tasks slots, the HDFS block size of 64MB. The default split size of 64MB was used, which allowed for all the map tasks to be executed in parallel even for the biggest dataset.

Each dataset consisted of 75x75 JPEG encoded images, harvested from Flicker and combined into Hadoop sequence files of type [Text, BytesWriteable]. Image URLs were used as sequence keys and

binary representation of the images as its values. Other characteristics of the datasets and the results of running both implementations are presented in Table 1.

Data set	# of images	Avg size [kB]	Total size [GB]	Avg time to process [s]		# map tasks
				MapReduce	Scalding	
img-set-tiny	5679	1.7	0.0092	134.8	111.0	1
img-set-small	27921	11.5	0.3	146.0	128.5	5
img-set-med	111645	11.5	1.2	147.0	133.3	20
img-set-big	557436	11.5	6.1	257.0	220.0	100

Table 1: Results of executing *MapReduce* and *Scalding* versions of mosaic creation

The results show that both *MapReduce* and *Scalding* versions scale very well with the execution time remaining essentially constant within the considered range of dataset sizes (see: Figure 4). The *Scalding* version however is on average 13% faster than the *MapReduce* version. Job statistics show that the *Scalding* version generates substantially less map output records from the *BestMatchesSelection* step - 114K compared to 632M produced by the *Hadoop* version. The combiner in the *Hadoop* version reduces the number of records sent to the reduce phase to 3M, but that is still about 30 times more than what the *Scalding* version outputs. Although Scalding does not use combiners, it optimizes map-side evaluation with Partial Aggregation [14] to achieve similar effect (i.e., reduce the map output size). In the case of mosaic creation, Partial Aggregation performs better than a combiner, which results in better performance of the *Scalding* version.

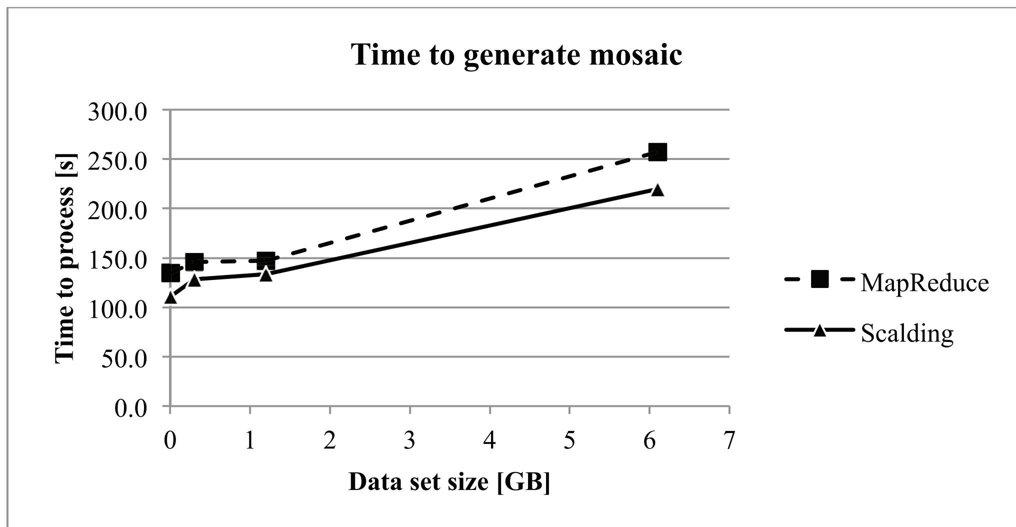

Figure 4: Comparison of *MapReduce* vs. *Scalding* execution times

To compare productivity of developing Hadoop applications with Scalding and Hadoop API, we use the lines of code (LOC) metrics as an approximate measure of the development effort. The results are presented in Table 2 and show that the *Scalding* implementation is three times smaller than the corresponding Hadoop API implementation (*MapReduce*).

Task	#Lines of Code	
	Scala/Scalding	Java/MapReduce
Job setup	8	33
Flow setup	0	12
Split image to grid	8	18
Select best matches	16	51
Join	3	35
Others[‡]	21	31
Total	56	180

Table 2: Lines of code for *MapReduce* and *Scalding* implementation

Figure 5 compares LOC for *Scalding* and *MapReduce* versions in different categories ordered by the productivity gain (*Scalding* LOC as the percentage of *MapReduce* LOC). Because of the implicit flows in Scalding, the biggest gain is achieved in the Flow setup category with no explicit code required in *Scalding* to setup or control the execution of the pipeline. The *join* operations provided by Scalding significantly reduce the effort to implement the *Join* step. Interestingly, the code for splitting the input image into a grid of cells is also approximately 3 times shorter in the Scalding implementation even though it is independent from the Scalding or Hadoop MapReduce API, which can be contributed to the higher productivity of Scala compared to Java.

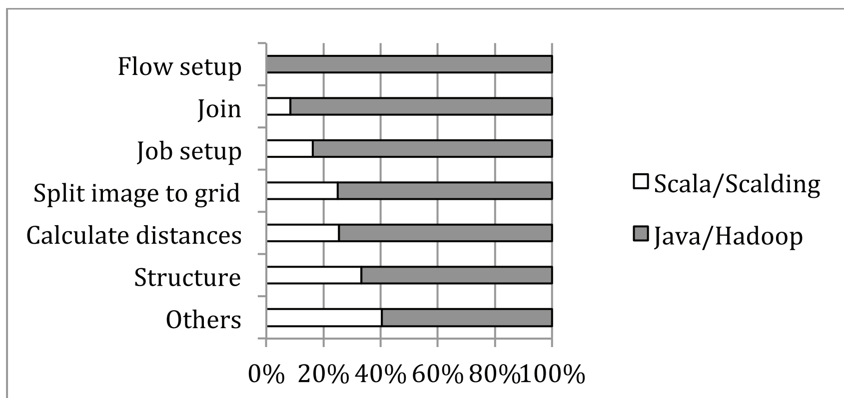

Figure 5: Detailed comparison of LOC for Scalding and MapReduce implementation

Overall the results show that the *Scalding* version of the mosaic creation application is not only requires one third of the code of the *MapReduce* version, but is also about 10% faster then the latter one. Although it is certainly possible to optimize the *MapReduce* version to be at least as good as the *Scalding* one in terms of performance, the cost would be increased complexity that would cause further degradation of development productivity. The results also show that Hadoop is a viable platform for distributed image processing.

[‡] That category includes packages declaration and imports

4 Conclusions

This paper presents an application of a Hadoop productivity framework Scalding for solving a big data image processing problem of creating photographic mosaic. The high-level flow paradigm insulates the developer from the Hadoop MapReduce model and allows for a natural expression of the solution pipeline with minimal plumbing code. Even though the mosaic creation pipeline is relatively simple, it benefits from the high level operation provided by Scalding (e.g., groupBy or join) and more complex pipelines can be productively built using the same approach.

The focus of the work presented in this paper was to demonstrate applicability of productivity frameworks for building image processing pipelines and as such the actual performance of the implementation was a secondary concern. Further work will concentrate on the execution aspects including: code optimization, cluster tuning, integration with high performance image processing libraries (e.g., OpenCV) and execution on a Hadoop cluster with GPU support.

References

[1] The Apache Software Foundation, "Apache Hadoop," 2007. [Online]. Available: http://hadoop.apache.org/,.

[2] The Apache Software Foundation, "Apache Oozie Workflow Scheduler for Hadoop," 2011. [Online]. Available: http://oozie.apache.org/.

[3] The Apache Software Foundation, "Apache HBase," [Online]. Available: http://hbase.apache.org/.

[4] The Apache Software Foundation, "Apache Mahout," [Online]. Available: http://mahout.apache.org/.

[5] The Apache Software Foundation, "Apache Flume," 2009. [Online]. Available: http://flume.apache.org/.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters ," *Communications of the ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[7] The Apache Software Foundation, “Apache Pig,” 2007. [Online]. Available: https://pig.apache.org/.

[8] cascading.org, "Cascading: Application Platform for Enterprise Big Data," 2007. [Online]. Available: http://www.cascading.org.

[9] The Apache Software Foundation, "Crunch," 2013. [Online]. Available: http://crunch.apache.org/.

[10] NICTA, "Soobi," 2011. [Online]. Available: http://www.opennicta.com/home/scoobi.

[11] Twitter, "Scalding," 2011. [Online]. Available: https://github.com/twitter/scalding.

[12] cascalog.org, "Cascalog:Fully-featured data processing and querying library for Clojure or Java.," 2010. [Online]. Available: http://cascalog.org/.

[13] K. Wiley, A. Connolly, S. Krughoff, J. Gardner, M. Balazinska, B. Howe, Y. Kwon and Y. Bu, "Astronomical Image Processing with Hadoop," in *Astronomical Data Analysis Software and Systems XX. ASP Conference Proceedings*, 2011.

[14] S. Chen, T. Bednarz, P. Szul, D. Wang, Y. Arzhaeva, N. Burdett, A. Khassapov, J. Zic, S. Nepal, T. Gurevey and J. Taylor, "Galaxy + Hadoop: Toward a Collaborative and Scalable Image Processing Toolbox in Cloud," in *ICSOC Workshop 2013*, 2013.

[15] B. Harris and J. George, "VideoMosaic Project," 2006. [Online]. Available:

http://www.intelegance.net/video/videomosaic.shtml.

[16] Wikpedia, "Photographic mosaic," [Online]. Available: http://en.wikipedia.org/wiki/Photographic_mosaic.

[17] JH Labs, “JH Labs - Java Image Processing,” [Online]. Available: http://www.jhlabs.com/ip/filters/index.html.

[18] cascading.org, "Partial Aggregation instead of Combiners," [Online]. Available: http://docs.cascading.org/cascading/2.0/userguide/htmlsingle/#N21472.

[19] Scala, "The Scala Programming Language," 2002. [Online]. Available: http://www.scala-lang.org/.

[20] L. Liu, C. Sweeney, S. Arietta and J. Lawrence, “HIPI - Hadoop Image Processing Interface,” [Online]. Available: http://hipi.cs.virginia.edu/about.html.

[21] The Apache Software Foundation, "Scrunch - A Scala Wrapper for the Apache Crunch Java API," 2013. [Online]. Available: http://crunch.apache.org/scrunch.html.

[22] M. H. Almeer, "Cloud Hadoop Map Reduce For Remote Sensing Image Analysis," *Journal of Emerging Trends in Computing and Information Sciences,* vol. 3, no. 4, pp. 637-644, 2012.