Scalable Image Processing Using Publish/Subscribe

Shruti Padamata Kanu Sahai
CS710-Project, Spring 2014
Georgia Institute Of Technology

ABSTRACT

With recent improvements in camera performance and the spread of low-priced and lightweight video cameras, a large amount of video data and image data is generated, and stored in database form. At the same time, there are limits on what can be done to improve the performance of single computers to make them able to process large-scale information, such as in video analysis and image processing. Therefore, an important research topic is how to perform parallel distributed processing of an image database by using the computational resources in a cloud environment. At present, the Apache Hadoop distribution for open-source cloud computing is available which allows parallel processing of data in a distributed environment using the MapReduce programming model. The implementation and analysis of this kind of a system has been well researched over the recent past. We, as a part of our project for the course CS – 7210 (Distributed Computing) at Georgia Institute of Technology, have tried an alternative approach to deal with this problem using a publish/subscribe system – Scribe – implemented on top of a Pastry routing substrate. In this paper, we will describe, in depth, how scalable image processing can be achieved utilizing a publish/subscribe system. In particular, we will analyze the problem of performing face-recognition to query for a target image in the image database stored across multiple nodes in different data-centers, utilizing the Scribe infrastructure.

1. INTRODUCTION

With the spread of cloud computing and network techniques and equipment in recent years, a large amount of image data from variety of applications like social-networking and image-sharing applications is being collected every day. The need for analysis techniques to take advantage of useful information that can be extracted from such data sets is thus, increasing. The Photos application is one of Facebook's most popular features [7]. Till date, users have uploaded over 15 billion photos which makes Facebook the biggest photo sharing website. For each uploaded photo, Facebook generates and stores four images of different sizes, which translates to a total of 60 billion images and 1.5PB of storage. The current growth rate is 220 million new photos per week, which translates to 25TB of additional storage consumed weekly. At the peak there are 550,000 images served per second. Video data, that includes the information about the sequential images/frames, encoded in a different manner, is also increasing at rapid rates and has even higher storage requirements. Today, because video cameras are set up to perform surveillance of moving objects such as pedestrians and vehicles, a large amount of video data is generated, and stored in database form. There is a huge potential for analyzing this data and deriving useful information. For this purpose, parallel processing techniques that can be implemented in a parallel and distributed fashion, to work on huge amounts of data, possible spread across multiple sites, is an important area of research.

When considering operations such as search and other types of analysis of images stored in a database, there are limits on what can be done to improve the performance of single computers to make them able to process large-scale information. Therefore, the advantages of parallel distributed processing of an image database by using the computational resources of a cloud computing environment should be considered. In addition, if computational resources can be secured easily and relatively inexpensively, then cloud computing is suitable for handling large image databases at low cost.

In the recent past, Hadoop, as a mechanism for processing large amounts of data using parallel and distributed computing is being considered a promising approach. For reasons such as ease of programming, using the functional programming approach of MapReduce on the Hadoop system, applications that can process data across multiple machines in a distributed environment are being developed. But, Hadoop's implementation of MapReduce has its own limitations.

Hadoop's performance is closely tied to its task scheduler, which implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers. In practice, the homogeneity assumptions do not always hold. An especially compelling environment where Hadoop's scheduler is inadequate is a virtualized data center. Researchers also expect heterogeneous environments to become common in private data centers, as organizations often own multiple generations of hardware, and data centers are starting to use virtualization to simplify management and consolidate servers. Matei Zaharia et al., in their work [8], have observed that Hadoop's homogeneity assumptions lead to incorrect and often excessive speculative execution in heterogeneous environments, and can even degrade performance below that obtained with speculation disabled. Moreover, Hadoop uses HDFS (Hadoop Distributed File System) as one of its underlying layer which involves breaking up a large file into chunks for storage. This kind of a storage system will not be very efficient for image processing involving face recognition. Also, if the size of images stored is very large then movement of data across the nodes for processing will have a toll on the performance of the system.

Having established the need for efficient ways of conducting parallel image processing, we define one particular use case which demands scalable and efficient parallel processing of image database and build our application to address that. With the need to store huge amounts of data which is growing by the day, big companies tend to have multiple data-centers which are geographically distributed across the globe. If we take an example of Facebook again, then Facebook also has its data-centers located in US, China and many other countries. The data stored at the data-center at a specific location is most likely to contain the data pertaining to users residing in that location due to locality principles. Now, in this setting, suppose if we want to judge the popularity of a person (for example, a celebrity) at a location, then we may want to know how many users in that area have uploaded an image of the person. The number of photos of the person found at data-centers in different locations can indicate the popularity of the person in those respective locations. Performing such a search for a target object or face in all the images contained in a database

, using offline processing techniques like MapReduce would require enormous amounts of data movement. Instead, we develop a simple user-library to perform the search by distributing the processing function to the nodes containing the actual data, to avoid such data movements.

In our project, we use a Java-CV based face recognition program which, given a training set and a target image, can determine the label of the image, based on the labels of the images in training set. This program can be used in our framework to recognize the faces stored at a node from the training set and match them against that of the celebrity's face. For simplicity of implementation, we assume that the database of images stored at the nodes of the data-centers is already pre-processed in a way that faces of all the persons have been extracted from the images and stored separately with a mapping to the actual photos. Such processing can be easily performed using other image processing techniques and face detection and extraction.

Since Scribe can be used for efficient multicast to a large number of groups and nodes, we decided to leverage it to perform parallel processing on the data stored at the leaf nodes by dynamic invocation of executable code that can be multicasted on the tree. In Scribe multicast, the message is replicated in the internal nodes of the tree and passed onto the multiple children. We utilize this functionality to deploy an image procesing function at each node in the tree to perform the processing images at that node. Further, to aggregate the results obtained from such processing at multiple nodes and derive results from the entire data, we extended the model to a map-reduce kind of framework. Thus, the data obtained from the processing on individual nodes can be aggregated in a reduce phase, to obtain more meaningful system-wide results. To keep the processing functions simple, we restricted the data.

All the nodes which subscribe to the 'map' topic will receive the processing function to be executed on the data in the form of a multicasted message. After the mapping phase is finished, the mapper nodes subscribe to different trees for the reduction of different keys. Hence this leads to formation of multiple scribe trees for different keys thereby leading to load-balancing.

The rest of the report is organized as follows. In section 2, we will describe and analyze the related work. In section 3, we provide some background on Scribe, that provides a multicast infrastructure for the nodes running our application and the underlying Pastry routing substrate, that is used for building Scribe. In section 4, we will give an in depth description of the design of our application and how the pastry and scribe infrastructure is utilized to perform map-reduce kind of processing on image databases. In section 5, we will discuss the shortcomings of our approach and scope for improvement. In section 6, we will end with a conclusion.

2. RELATED WORK

There have been many implementations of parallel image processing systems, be it in a multi-core architecture [3] or in a cluster computing environment. In earlier times, most of the parallel image processing has been done in the context of segmenting a very large image and processing it parallely. In [5], the authors developed a system to obtain results for the parallelization of high level image processing algorithms, namely for active contour and modal analysis methods. On the other hand, we have image processing systems which operate on a database of images stored across a cluster of computers. MapReduce is a popular choice for this kind of parallel processing. HIPI [2] is a library for Hadoop's MapReduce framework that provides an API for performing image processing tasks in a distributed computing environment. [9] uses MapReduce for video processing in distributed environments. As individual cluster computers often cannot satisfy the computational demands of emerging problems in huge amount of image data, it was natural for people to extend their approaches to multiple clusters. [10] takes into consideration a multi cluster environment for computationally efficient processing of hyperspectral image cubes.

In Web-Scale Computer Vision using MapReduce for Multimedia Data Mining [11], Brandyn White et al. present a case study of classifying and clustering billions of regular images using MapReduce. No mention is made of average image dimensions or any issues with not being able to process certain images because of memory limitations. However, a way of pre-processing images for use in a sliding window approach for object recognition is described. Therefore one can assume that in this approach, the size of images is not an issue, because the pre-processing phase cuts everything into a manageable size.

A description of a MapReduce-based approach for nearest-neighbor clustering by Liu Ting et al. Is presented in Clustering Billions of Images with Large Scale Nearest Neighbor Search [12]. This report focuses more on the technicalities of adapting a spill-tree based approach for use on multiple machines. Also, a way for compressing image information into smaller feature vectors is described. With regards to this thesis, again the focus is not so much on processing the images to attain some other result than something intermediate to be used in search and clustering.

To the best of our knowledge, no major work has been done on deploying parallel image processing using publish/subscribe systems.

3. BACKGROUND

In this section, we will dive into the concepts of scalable wide area peer to peer object location and routing substrate – Pastry and the functionality of Scribe on top of Pastry. Lastly, we will present some face-recognition techniques that are implemented in OpenCV and JavaCV libraries.

3.1 Pastry

Pastry [13] is a generic peer-to-peer object location and routing scheme, based on a self-organizing overlay network of nodes connected to the Internet. Pastry is intended as general substrate for the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems. Several application have been built on top of Pastry to date, including a global, persistent storage utility called PAST and the scalable publish/subscribe system of SCRIBE. Pastry routes messages to the node whose nodeId is numerically closest to the given key. To support the routing procedure, each node maintains a Leaf set, Routing table and Neighborhood set. Pastry also takes into account the locality properties of the nodes.

3.2 Scribe

Scribe [14] is a scalable application-level multicast infrastructure built on top of Pastry. Nodes can create, send messages to, and join many groups. Scribe can support simultaneously a large numbers of groups with a wide range of group sizes, and a high rate of membership turnover. Scribe system consists of a network of Pastry nodes, where each node runs the Scribe application software. Each group has a unique groupId. The Scribe node with a nodeId numerically closest to the groupId acts as the rendezvous point for the associated group or the root of the multicast tree created for the group.

3.3 Face Recognition

Automatic face recognition is all about extracting the meaningful features from an image, such as lines, edges or angles, putting them into a useful representation and performing some kind of classification on them. Face recognition based on the geometric features of a face is probably the most intuitive approach to face recognition.

OpenCV (Open Source Computer Vision) is a popular computer vision library started by Intel in 1999. The cross-platform library sets its focus on real-time image processing. OpenCV 2.4 comes with a FaceRecognizer class for face recognition. The currently available algorithms are:

- Eigenfaces
- Fisherfaces
- Local Binary Patterns Histograms

Eigenfaces method takes a holistic approach to face recognition: A facial image is a point from a high dimensional image space and a lower-dimensional representation is found, where classification becomes easy. The Principal Component Analysis (PCA), which is the core of the Eigenfaces method, finds a linear combination of features that maximizes the total variance in data. While this is clearly a powerful way to represent data, it doesn't consider any classes and so a lot of discriminative information may be lost when throwing components away. The Linear Discriminant Analysis performs a class-specific dimensionality reduction and was invented by the great statistician Sir R. A. Fisher. Hence, this method came to be known as Fisherfaces. Another approach is Local Binary Patterns. Its basic idea is to summarize the local structure in an image by comparing each pixel with its neighborhood. Take a pixel as center and threshold its neighbors against. If the intensity of the center pixel is greater-equal its neighbor, then denote it with 1 and 0 if not. In our implementation, we use JavaCV which provides Java wrapper functions around the above mentioned functions.

4. DESIGN

In this section, we will extensively describe the design of our framework and application. We build a library that can be used in a Scribe application, to perform distributed processing on the data available on the Scribe tree nodes, using dynamic invocation of user-specified processing functions that are multicasted on the Scribe trees. The library provides a programming interface similar to that of Hadoop MapReduce. The programmer can thus perform two-phase computations that can be structured as map and reduce functions. As noted in [1], a wide-range of applications can be implemented in this fashion. The dynamic invocation of functions on data present on the application nodes eliminates the need for moving the data for executing MapReduce kind of jobs. This can be a powerful tool for data intensive applications like many of the image and video processing applications. We implement a simple face recognition application to illustrate the usage of the library.

4.1 Terminology

Based on the kind of job assigned or performed by a Scribe node, we refer to them using one of the following categories, through the rest of the paper.

- Mapper nodes – All the nodes which store data relevant to the application will subscribe to a 'Map' topic so as to be in one multicast group. In our case, these will be the nodes that store the image data for a particular application.
- Map tree root – Root of the multicast tree formed by 'Map' topic. This node may or may not be a subscriber of the topic.
- Reducer nodes – These are responsible for carrying out reduce tasks. There will be multiple nodes designated as reducers depending upon the number of partitions of the intermediate keys.
- Reduce tree root – All the reducer nodes will subscribe to a 'Reduce' topic to be in the same multicast group. This node will act as a rendezvous point for the reducer nodes.

By virtue of all mapper nodes being a part of the same multicast group, the root of map tree can be used to multicast the map job to all the mapper nodes. Mapper nodes, after completing their tasks, may send the intermediate key-value pairs to the reducer nodes which in turn after the reduction of values for all the keys may send the final output to the root of the reduce tree.

4.2 Mapper and Reducer classes

Hadoop's implementation of MapReduce provides the user with Mapper and Reducer classes in the user-library which can be extended by users to override the functions required, commonly the map and reduce functions. In a similar manner, we have designed two classes, namely PastryMapper and PastryReducer which are analogous to the aforementioned classes. These classes are designed to store information about the state of the tasks, the intermediate key-value pairs received and the user-defined map/reduce functions at a node. The usage of these attributes will be explained in detail in later sections.

4.3 Workflow of the MapReduce job

The MapReduce job on a particular scribe tree can be started by any Pastry node in the network. The application at that node needs to have the implementation code for the functions that are to be dynamically invoked at the mapper and reducer nodes. All the nodes containing the data of interest, .i.e., the image files pertaining to a particular application initially subscribe to a scribe topic. On job invocation, the user-defined code is first passed to the root of this tree (, which is also assumed to the map tree in our implementation), in a packaged form. The root then propagates the code to all the members of the tree. These nodes will contain the actual data to be processed. Hence, they invoke the map functions on each file or image.

The intermediate keys from the different nodes may contain common keys. The set of unique keys is obtained by transferring all the unique keys from the mapper nodes to the map-tree root. The root also partitions the keys and assigns topic names for the reducer trees corresponding to each partition. It passes this information back to the mapper nodes. Mappers subscribe to the reducer topics corresponding to their keys and transfer the intermediate data to the nodes of the respective trees. These nodes act as reducers and invoke the reduce function on each set of intermediate values. Further to write the final output on a single node, all the reducer nodes subscribe to a topic called 'reduce' and pass the results to the root of that tree, which finally writes the output to a file on its disk.

5. IMPLEMENTATION

The interface made available to the user of the application is similar to the Hadoop's MapReduce implementation. The user needs to define a setup() function to declare the input and output key and value data-types. Next, he/she needs to define the map and reduce functions extending the PastryMapper and PastryReducer classes respectively. All this user code is then archived in a jar file and stored on the node which needs to instantiate the job. Following steps are involved in the execution of a MapReduce job on the publish/subscribe infrastructure:

- All the nodes in the system will subscribe to a 'Map' topic and a 'Reduce' topic. Subscription to is not necessarily required, but reduces the code complexity.
- The node containing the user code will make a call to job_start() function to initiate the job and send the function containing jar to the root of the map tree as well as the reduce tree for the function to be deployed. This message is sent as a direct pastry message to both the roots. Figure1 shows that state of the nodes where the initiator for the job sends the code package to the map and reduce tree roots. Note that the reduce topic has no members at this point and the reducer nodes are not designated.
- Map tree root and the reduce tree root upon receiving the jar will then instantiate a job object each containing three classes : PastryMapper, PastryReducer and Temporary Store.
- Map tree root multicasts the user code containing jar to all the mapper nodes in the form of a scribe message. It retains a copy of the Pastry message itself. This is also included in Figure 1.
- Once all the mapper nodes receive the jar from the root, they instantiate a job object themselves and initialize only the mapper instances. All the mapper nodes then proceed to carry out the map function provided by the user.
- Once the map task on a node is finished, the node sends a list of all the input keys it has to the map root tree. Map root tree keeps on accumulating all the keys from all the mapper nodes and hence forms a list of all the unique input keys present in the system which need to be reduced. For the reduction of each key, a separate topic is created, for all the mapper nodes to send the corresponding data to the key to the respective reducer nodes. So, basically reducer node of a key is the root of the topic created for the key.
- The map tree root thus creates a hash map of keys and corresponding topics (stored in a class called KeyTopicMap) and multicasts it to all the mapper nodes. Once the mapper nodes receive this hash map, they create and subscribe to all the topics of the keys.

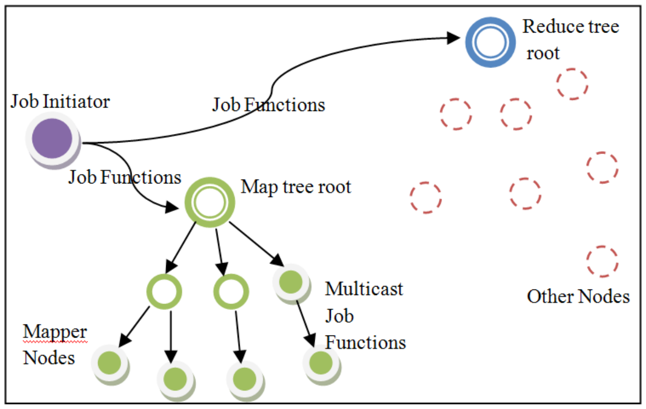

Figure 1: Invocation of the job from any node in the tree

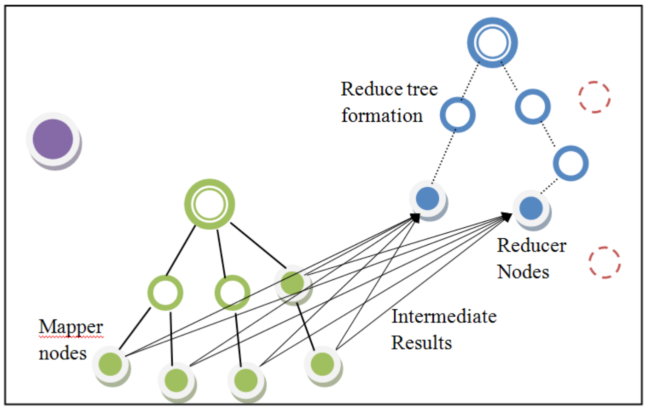

Figure 2: Mapper nodes send the intermediate results to reducer nodes

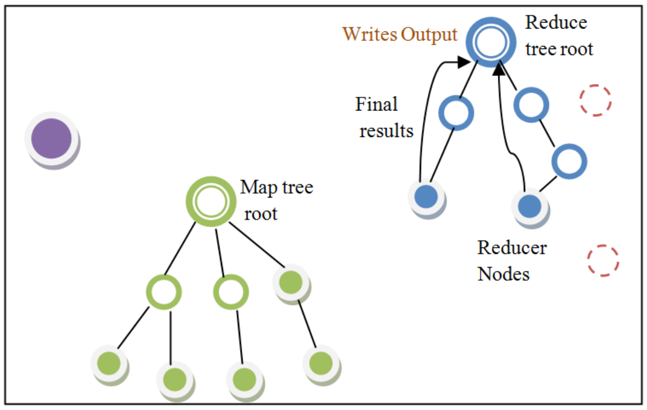

Figure 3: Reducer nodes send the final results to the reduce topic root node which will write the output to disk

- The mapper nodes iterate through all the unique keys of the map and send the list of intermediate values corresponding to a key to that respective key's reducer node. Thus, the shuffling of intermediate key-value pairs after the completion of the map phase is captured in the creation of multiple reduce trees. Figure 2 illustrates this shuffling of the keys sent from the mapper to the reducer nodes.
- The pastry node acting as a reducer is not aware of its responsibility before hand. Upon receiving a scribe message containing intermediate key value pairs, it initializes a job object instantiating just the temporary store. Here, the role of temporary store first comes into play. It acts as a container for all the intermediate key value pairs on the reducer node. All the reduce nodes, while accumulating the intermediate key value pairs, also subscribe to the 'Reduce' topic and notify the map tree root of the event. Map tree root requires this subscription acknowledgement from all the reducer nodes before multicasting the 'Reduce' function to all the reducer nodes.
- Once the map tree root receives notification from all the reducer nodes, it multicasts the reduce function in the 'Reduce' topic along with a list of node handles which let each reducer node know how many messages from mapper nodes should it expect and wait on. Reducer node on the receipt of this message from the root will initialize the **state** and **reducer** in the job object and then on will proceed with the reduction.
- Final output of each reduce task at a node which is stored in a context object is then finally sent to the root of the reduce tree where it is written out to a file on disk. This is shown in Figure 3.

The Pastry and Scribe library from FreePastry are implemented in Java. The other components required for the implementation of the application, like the face-recognition libraries are also available as Java libraries through JavaCV. Hence, we have implemented the entire application in Java. The code packages are passed between nodes as byte array objects contained in Pastry Messages or as Scribe Content. Java reflection API is used for dynamic invocation of the user-supplied functions.

Using, the above map-reduce kind of library, we have developed a simple face-recognition application to obtain the number of users at different data-centers who have stored a particular person's image. As mentioned in the Background section, JavaCV provides three kinds of Face-recognition functions. We have used the EigenFaces method, since it was performing well on the data-set that we have used for our testing.

The map phase of the application takes the name of one input image as ImageName and its location as a String. Here, location is the randomly chosen location among (United States, China and India), that we statically assigned to each location, for illustration purposes. This location is specified in a file in each node's directory. The map phase processes the image using the training set and predicts a label consistent with the labeling in the training set. We then, compare this label with that of the query image to determine whether it is the same face, and output a key-value pair with location as the key and a value of one, to indicate the count of the images at that location. To avoid processing the query or target image multiple times in each map invocation, we have hard-coded the target image label in the map function. This is similar to processing the case where the user application may pre-process the query image and provide the label to used for comparison.

The reduce phase aggregates the count for each location and produces the final output as the number of images of the target person present at each location or data-center.

6. EVALUATION

The source code attached with the project contains the implementation for the simple Face Recognition application that we have described previously. For implementing any such map-reduce kind of job, the user needs to provide the map and reduce functions to be used for the invocation by extending the PastryMapper and PastryReducer classes described in the previous section. The input and output formats for the map and reduce functions are currently limited to a small number of types: TextName for representing the name of a text-file, Text for representing text-file data, ImageName for representing names of image files and the default Jave Object types like String and Integer, all of which are Serializable.

The face-recognition application that we have implemented tries to identify a target face among the images present on the different nodes in the system. Along with the face-recognition function, the user needs to input a training set of images that are labeled with numeric identifiers. We have obtained the training set from the AT&T Facedatabase [15]. It contains ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement). We have split the image database into two sets, one set of around 100-120 images that are labeled with numbers, to be used as a training set and the rest of the images are split across the Scribe tree nodes, each containing 28-32 unlabeled images.

As described in the previous section, the map function takes each image in the node's local database as input and outputs the intermediate key-value pairs of location and value one, on finding a match with the target image. The reduce function aggregates these values to get the final count of matching images per location and writes the output to 'Reduce-Output.txt'.

We have evaluated the code and obtained some basic measurements of the time taken for propagation of the Job Functions from the map-tree root to the mapper nodes, the execution of the map phase per file and the for the execution of reduce phase per key. The results of the average, minimum and maximum time taken for each of these is shown in Table1. The measurements were performed on a Virtual Box installation, running Ubuntu 32bit, with 4GB base memory and 16GB virtual disk space. The Virtual Box was installed on a machine with Intel i5-3230M CPU, 64-bit Windows OS and 8GB physical memory. All the pastry nodes were run on a single JVM.

Measurement Parameter	Avg.	Min.	Max.
Time to propagate Job code to mapper nodes	117703 ms	0 ms	228068 ms
Map function execution time	702 ms	662	883
Reduce function execution time	662 us	14.6 us	1815 us

Table 1: Timing measurements

The timing for propagating the Code package shows a wide range of values which seem to be continuously increasing, and thus, the large difference between the minimum, average and maximum values. This is mostly due to running the nodes in a single JVM. We verified that the behavior is different when some of the nodes are run on a separate node. Also, the value of 0 for the minimum time might be because of the root being a part of the tree and sending a message to itself. Unfortunately, we could not complete all the changes to make the code run on separate JVMs to report the correct timings. The rest two parameters should be correct, as they will not be affected by the single JVM case.

7. ACKNOWLEDGMENTS

We would like to thank Professor Karsten Schwan, our instructor for the course Distributed Computing, for exposing us to the field of peer-to-peer computing and motivating and giving us the freedom to implement our own ideas in this area. We also express our gratitude to Liting Hu, our guide for the project, who has led us to the right resources and has always been available to provide us with her excellent feedback on our work.

8. REFERENCES

[1] D. Jeffrey, and G. Sanjay, MapReduce: Simplified data processing on large clusters. Paper presented at OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, USA, December 6 – 8, 2004.

[2] S. Chris, L. Liu, A. Sean, and L. Jason, HIPI: A hadoop image processing interface for image-based map reduce tasks, B.S. Thesis. University of Virginia, Department of Computer Science, 2011.

[3] Sanjay, S. Neeraj, S. Shiru, Image Processing Tasks using Parallel Computing in Multi core Architecture and its Applications in Medical Imaging, International Journal of Advanced Research in Computer and Communication Engineering Vol. 2, Issue 4, April 2013

[4] D. Bader, J. JaJa, D. Harwood, and L. S. Davis. Parallel algorithms for image enhanced and segmentation by region growing with an experimental study. Technical Report UMCP-CSD:CS-TR-3449, University of Maryland, May 1995.

[5] J. Barbosa, J. Tavares, and A. J. Padilha, "Parallel Image Processing System on a Cluster of Personal Computers", Vector and Parallel Processing , 4th International Conference , Porto, Portugal, June, 2000.

[6] J.G.E. Olk, P.P. Jonker: Parallel Image Processing Using Distributed Arrays of Buckets, Pattern recognition and Image Analysis, vol. 7, no. 1,pp.114-121,1997

[7] Doug Beaver , Sanjeev Kumar , Harry C. Li , Jason Sobel , Peter Vajgel, Finding a needle in Haystack: facebook's photo storage, Proceedings of the 9th USENIX conference on Operating systems design and implementation, p.1-8, October 04-06, 2010, Vancouver, BC, Canada

[8] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments, in: Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, San Diego, USA, Dec. 2008

[9] Muneto Yamamoto and Kunihi ko Kaneko, "Parallel Image Database Processing With MapReduce And Performance Evaluation In Pseudo Distributed Mode," International Journal of Electronic Commerce Studies, Vol.3, No.2, pp.211-228, 2012.

[10] F. Liu, F. J. Seinstra and A. Plaza, "Parallel hyperspectral image processing on distributed multicluster systems," Journal of Applied Remote Sensing, pp. 1-14, 2011.

[11] Brandyn White, Tom Yeh, Jimmy Lin, and Larry Davis. Web-scale computer vision using mapreduce for multimedia data mining. In Proceedings of the Tenth International Workshop on Multimedia Data Mining, MDMKDD '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM.

[12] Clustering Billions of Images with Large Scale Nearest Neighbor Search, 2007.

[13] Antony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany, Nov. 2001.

[14] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., AND ROWSTRON, A. 2002. SCRIBE: A large-scale and decentralized application-level multicast in frastructure. IEEE J. Sel. Areas Commun. 20, 8 (Oct.), 1489–1499.

[15] Face Recognition with OpenCV http://docs.opencv.org/modules/contrib/doc/facerec/facerec_tutorial.html#face-database

AT&T Facedatabase http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html