International Journal of Integrated Engineering, Vol. 10 No. 1 (2018) p. 74-84
© Penerbit UTHM
DOI: https://doi.org/10.30880/ijie.xx.xx.xxxx.xx.xxxx

Parallel Processing of Image Segmentation Data Using Hadoop

M. Nishat Akhtar[1], Junita Mohamad Saleh[2*], C. Grelck[3]

[1]School of Aerospace Engineering
Universiti Sains Malaysia, 14300, Nibong Tebal, Penang, Malaysia
[2*]School of Electrical and Electronics Enginnering
Universiti Sains Malaysia, 14300, Nibong Tebal, Penang, Malaysia
[3]Informatics Institute, University of Amsterdam,
Science Park 904, Amsterdam, The Netherlands

Received 30 November 2017; Accepted 19 February 2018, Available online 30 April 2018

Abstract: The use of sequential programming is slowly getting replaced by distributed and parallel computing which is widely being used in computing industries to handle tasks with big data and various high-end computing applications comprising of huge image and video data banks. Moreover, image processing using parallel computation is also gaining momentum in today's technological era. Nowadays researchers are coming up with various methodologies to tackle high scale image processing applications by implementing parallel computing methodologies to carry out the specified image processing application task and simultaneously checking its performance against sequential programming. At the same time there are constraints on what can be done to maximize the task performance using high end multi-core CPU's with advanced buses and interconnects that offer high bandwidth with low system latency. It is to be noted that there is no availability of standardized image processing task which can be used to evaluate a single node system. In this paper, we propose an efficient parallel processing algorithm to perform the task of image segmentation with the foremost aim to analyze the threshold of data size at which the proposed method outperforms sequential programming method in terms of task execution time by analyzing the distribution of average CPU cores usage and its threads over the execution time. The proposed methodology could be useful for researchers, as it can perform multiple image segmentation in parallel, which can save a lot of time of the user. For the purpose of comparison, we also implemented the same image segmentation task using sequential method of programming in an integrated development environment platform.

Keywords: Parallel computation, Image segmentation, Hadoop, HIPI, Map-reduce, Input split

1. Introduction

In order to process terabytes and petabytes of data, the Map-reduce framework proposed by Google is a viable model. Moreover, Map-reduce framework is now a key feature of Hadoop which is an open source software framework for writing applications using parallel programming technique [1]. Map-reduce have become highly popular over a wide range of applications for intense data analysis. The performance of Map-reduce depends upon many factors i.e. network configuration of the cluster (which determines whether it is single node or multi-node cluster), controllable parameter in the Hadoop framework (setting the split size of number of mapping and reducing for task distribution). It is considered to be very essential to tune the Hadoop framework so as to achieve maximum performance. Besides this, performance also depends upon the system/machine/nodes configuration i.e., multi-core CPU with high frequency will definitely give a better performance than single-core CPU with lower frequency.

Nowadays, high scale image analysis done using distributed and parallel computing is widely being recognized across the industrial and research field. This type of image analysis is also used for video data, which continuously generates sequential images and related data which includes associated time and frame information. Since video cameras are also used in surveillance application which leads to the generation of huge image datasets. Therefore there arises a great need to come up with a solution which can analyze these huge image data files in parallel. Moreover, in order to process multiple image files, sequential programming could become time consuming when the size of the dataset expands. Hadoop Image Processing Interface (HIPI) is considered to be an essential API for analyzing bundle of images in parallel [2]. The advantages of distributed and parallel processing of large image database using HIPI API of Hadoop framework should be taken into consideration. Furthermore, if the computational resources can be acquired easily and cheaply, then HIPI is most suitable for handling large image database in an economic manner.

The foremost contribution of this manuscript is to implement parallel image segmentation using Map-reduce technique with HIPI to analyze the threshold of the data size at which the proposed parallel image segmentation

*Corresponding author: jms@usm.my
2018 UTHM Publisher. All right reserved.
penerbit.uthm.edu.my/ojs/index.php/ijie

method outperforms the same image segmentation task performed using sequential programming. For our proposed experiment, we are not dealing with any benchmark performance evaluation by taking multiple nodes to process terabytes and petabytes of data. Instead we are emphasizing on single node to analyze its optimum performance.

The second task is to illustrate the performance of the proposed parallel image segmentation method in terms of task mapping and task reducing for job/task completion.

The rest of the paper is organized as follows; Section 2 gives a brief overview of related works done by researchers in the field. Section 3 gives the methodology of how parallel image segmentation is implemented with Hadoop Map-reduce algorithm. Finally, section 4 and section 5 present the results and conclusion, respectively.

2. Related Works

Over the years multiple image segmentation algorithms have been used to analyze images. Nowadays a wide range of algorithms is being used to carry out the process of image segmentation as texture is an essential feature which reflects important information about an image surface. The aim of image segmentation is to cluster the entire pixels into specified salient image regions, i.e. regions corresponding to individual objects, surfaces or natural part of objects. It is an essential process of object recognition, image compression, image database lookup and occlusion boundary estimation within stereo or motion system.

Researchers these days are dealing with the problem of over segmentation of images which ultimately leads to inaccurate results and therefore leaves a room for enhancing this problem [3]. The basic image properties dealt with image segmentation are its dissimilarity and similarity. Sharp changes in the intensity of image causes dissimilarity whereas similarity corresponds to the process of combining and matching the pixels with the neighboring one based on its gray level pixel value match. Some of the widely recognized techniques to implement image segmentation are; Otsu's threshold method for automated image segmentation [4], region growing and region merging technique [5], edge detection method [6], watershed transformation [7] and histogram thresholding based algorithms [8]. Amongst all the techniques Otsu's method is a widely renowned method to carry out the process of image segmentation. Since it is an automated process, therefore it is easier to be applied on bulk image data simultaneously. Since we are dealing with multiple image datasets, therefore it is appropriate to use Open Source Computer Vision (OpenCV) library and it is also to be noted that Otsu's threshold technique has high degree of compatibility with OpenCV. Furthermore, OpenCV has the capability to exploit high degree of parallelism due to its available rich set of libraries. These scenarios make the condition more favorable for parallel image processing in an efficient manner.

For parallel image processing platform, HIPI is an extensive Application Program Interface (API) which is only compatible with Hadoop Map-Reduce framework [2]. HIPI has full potential to accommodate high throughput image processing using Hadoop Map-Reduce algorithm which can be implemented on a cluster of nodes. Hadoop has its own file system for data storage which is called Hadoop Distributed File System (HDFS) and HIPI facilitates the solution to store big image data on HDFS for efficient data processing. Moreover, HIPI provides integration with OpenCV, which is the most popular open-source library to carryout high end image processing tasks [2].

Processing big image datasets using Principal Component Analysis (PCA) was also very much known in terms of classical method for which Cadima and Jolliffe [9] came up with an efficient method to interpret big datasets using PCA by reducing the dimensionality of the image datasets and at the same time increasing its interpreting ability. However, the disadvantage of PCA is that even the simplest invariance is unable to be captured by this process unless the information is provided explicitly by the training data [10]. Moreover, any covariance matrix is also difficult to be evaluated in a precise manner [10]. If suppose the intensity level of an image falls evenly outside the range of levels in the background region , then threshold techniques is highly compatible to be applied as it analyses the image on the basis of the local pixel information. If a principal component analysis is done on any image, then segmentation process is the foremost step which needs to be applied. In order to carry out the process of image segmentation, the images are split into multiple blocks i.e. each pixel can be represented by a block which contains its neighboring pixel. This is due to the fact that most of the points in any high intensity image are spatially coherent with their neighboring pixel point. PCA is used to analyze the variation of patterns in any image. It expresses the pattern data in such a way that it highlights their similarities and differences. As discussed in the above paragraph, that in order to apply PCA, image data are to be divided into blocks so as to analyze the image. The foremost focus of the researchers in this field these days are that how to process this image data blocks in parallel. This study would in turn give a new paradigm of benchmark study in the area of cloud computing and big data.

In order to evaluate Map-Reduce systems, Sangroya et al. [11] developed a MRBS benchmark. MRBS in turn provides five sub-bench marks to analyze several application domains and a broad range of execution conditions [11]. For the purpose of parallel image processing, Slabaugh et al. [12] used Open Multi-Processing (OpenMP) technique to apply image transformation, image morphology and median filtering. Using OpenMP technique, multiple thread(s) of CPU cores were brought to use to increase the level of parallelism. From their conducted experiment, image processing (image transformation, image morphology and median

filtering) performed using OpenMP multi threaded technique emphatically outperformed image processing performed using single threaded technique. According to the research of Kang et al. [13] on performance comparison of OpenMP, Message passing Interface (MPI) and Map-Reduce programming for computing all-pairs-shortest-path problem, OpenMP gave the best results followed by MPI and Map-Reduce. As per their research, OpenMP is considered to be the de facto standard model for shared memory systems, MPI is the de facto model for distributed processing system and Map-Reduce is the de facto standard framework for high end data processing. The disadvantage of OpenMP is that it only runs in shared memory computers and requires a compiler that supports OpenMP. However, MPI can be implemented on both shared and distributed memory architectures. On the other hand, the MPI performance is limited by the bandwidth of communication network between the nodes.

3. Methodology

The proposed parallel image segmentation program is designed using HIPI which is made compatible to Hadoop 2.6.1 version. The most interesting feature of HIPI is its integration with OpenCV, which is a prominent open source library comprising of various computer vision algorithms. In addition to this HIPI has the capability to be deployed on the cluster of nodes. Storage of large collection of image is an issue; however HIPI sorts out this problem by using the memory storage of HDFS and makes it accessible for efficient distributed image processing.

For the proposed image segmentation program designed using HIPI, the input data should be in HIB format which stands for HIPI Image Bundle and is the foremost representation of image collection on HDFS. HIB exploits the Hadoop Map-Reduce feature to its maximum which is designed to support the efficient processing of large flat files. The HIB class provides the basic functions for reading, writing and concatenating HIB files. In order to create HIB, the HIPI distribution comprises of several useful tools [26], which also includes a Map-Reduce program that has the capability to build up a HIB from multiple images downloaded from the internet directly.

The initial process which HIPI uses to filter out the images is known as culling process. The process of culling is based on various user defined conditions i.e. spatial dimension or resolution associated to image meta-data. The culler class extends the Map-Reduce framework and enables the culling process through HIB runtime mode prior to its delivery to the Mapper in a complete decoded format.

Soon after the culling process, the image are assigned to the individual Mapper's in order to implement the map task so as to attempt to maximize the data locality, which sends the Map-Reduce code to each data node in the cluster. The following Section 3.1 elaborates the design consideration for building HIPI on Hadoop framework.

3.1 Design Consideration

For any Map-Reduce task, its execution time determines its performance. The factors which influence the performance of Map-Reduce task are uniform data distribution, input split size parameter, number of Map-Reduce task along with resource utilization of the node. These factors are discussed in the following sub-sections.

3.1.1 Uniform Data Distribution

Once the Mapper receives the HIB file with a specific input key-value pair, then it transforms that input key-value pair to a group of intermediate key-value pair. Soon after, the obtained intermediate key-value pair is shuffled and is passed to the Reducer where they are consumed. Moreover, this distribution of key-value pair to the Reducer can either be skewed or even. An even balanced load can reduce the task execution task execution time drastically by deploying all Reducers to complete the job at the same time. However, in order to achieve this, the chunk of HIB file has to be in even division which is not attained easily, therefore there is some room left for the skewed load, where most of the Reducers finish up the task quickly whereas some of them take a little longer time. The uniform data distribution is considered to be an important parameter for the designed image segmentation program.

3.1.2 Input Split Size

The split size divides the files into multiple blocks according to its block size. The Map-Reduce job submitter generates the number of splits which is equal to the number of block size of the file. For any given data size, the number of time the Mapper and the Reducer function is called can be determined by the size of the intermediate key-value pair. The proposed parallel image segmentation technique provides support for the three parameters as follows; key size, value size and the total number of key-value pairs. These parameters altogether can determine the total data to be processed from each specific map and moreover, it also determines the total data size to be shuffled. According to "Hadoop, the Definitive Guide", the default value of the maximum split size is the maximum value that can be represented by a java `long` data type.

The configuration parameters i.e., mapred.min.split.size and mapred.max.split.size are used to define the minimum and the maximum split size of the input data. The final split size could be calculated using the formula:

```
max(mapred.min.split.size,min(mapred.
max.split.size, HDFS.block.size))
```

By default:

```
mapred.min.split.size<HDFS.block.size<
mapred.max.split.size
```

It is also noted from the Hadoop log files that for every size of image data, if:

```
Number of allotted Mapper=number of
map task launched=1
```

Then the input split size for each Mapper is same as the input file size.

Similarly, if:

```
Number of allotted Mapper=number of
map task launched= 1+n
```

Then the input split size for each Mapper is *1/(1+n)* of the input file size.

Therefore, it can be said that the split size is equivalent to the block size. However, it only has an effect when split size is lesser than the block size.

3.1.3 Mapping and Reducing

Google introduced the Map-Reduce framework in order to allow a distributed processing on multiple clusters [1]. Unlike other distributed processing framework, where the data are pushed to specific nodes that belong to a particular cluster for processing, the Map-Reduce system follows a different approach [1]. In this case, the data are distributed among the nodes and the tasks are pushed to the particular node that stores the data. Map-Reduce framework is a two step process and is based on the concept of key, value pair $\langle k, v\rangle$. The Map function or the Mapper takes one pair of data with a type in single data domain as input $\langle k^{x,in}, v^{x,in}\rangle$ and returns a list of pairs as output in different domain which could be written as:

$(k_1^{x,out}, v_1^{x,out}), (k_1^{x,out}, v_{(M-1)}^{x,out}), \ldots\ldots\ldots\ldots, (k_n^{x,out}, v_2^{x,out}), (k_n^{x,out}, v_M^{x,out})$

The key emitted by the Mapper is not unique, therefore the Reducer which is also known as the Reducer function, groups up the values together for each Mapper domain. This could be written as:

$(k_1^{x,out}, [v_1^{x,out}, \ldots\ldots\ldots\ldots, v_{(M-1)}^{x,out}]) \rightarrow (k_1^{y,out}, v_1^{y,out})$

Depending on the implementation of the Map-Reduce framework, the Reducer could also produce multiple key, value pairs as output. Thus, the function of Map-Reduce framework is to transform a list of (key, value) pairs into a list of values. This model is different from the typical functional programming of Map-Reduce combination, which can only accept a list of arbitrary values and returns just one single value that altogether combines the values returned by the Mapper.
Section 3.2 will elaborate on how to build HIPI on Hadoop framework.

3.2 Building HIPI on Hadoop

1. Setting up of Java

HIPI is composed in Java and has been tried with Java 7 and 8. Java version has to be checked using the following command:

```
java -version
```

2. Setup Hadoop

Unzip Hadoop using tar command: `tar -xvzf hadoop-2.6.1.tar.gz`
HIPI works with a standard establishment of the Apache Hadoop Distributed File System (HDFS) and Map-Reduce. HIPI has been tried with Hadoop version 2.6.1.
The verification of the main Hadoop script has to be checked from the path using the following command:

```
which Hadoop
```

3. Setup Gradle

The HIPI distribution utilizes the Gradle construct automation framework to organize package and compilation assembly. HIPI has been implemented with Gradle adaptation version 2.5.
Introduce Gradle on the Hadoop framework by checking the path using the command:

```
which gradle
```

4. Introduce HIPI

For the proposed research, HIPI has been downloaded from GitHub. The most ideal approach to get the most recent version of HIPI is by cloning the official GitHub repository (GitHub, 2008) and building it alongside the majority of the tools required to execute the framework. The following command is used to clone the GitHub repository:

```
git-clone
git@github.com:uvagfx/hipi.git
```

The git clone command construct the HIPI Library and its associated tools.
In order to build the HIPI library along with all of its associated tools, simply run gradle from the HIPI root directory. Fig. 1 shows the detailed steps to demonstrate how HIPI library is built using Gradle.

1. *Change directory to HIPI*
2. *Issue Gradle build command*
3. *Java compiler gets built*
4. *Java process resources gets built*
5. *HIPI tools gets built*
6. *Finish building the HIPI library along with all tools and examples.*
7. *BUILD SUCCESSFUL*

Fig. 1 Building up of HIPI library

The following Section 3.3 demonstrates the designing of Map-Reduce algorithm to implement parallel image segmentation.

3.3 Image Segmentation on Hadoop Framework

Thresholding is considered to be an important technique for image segmentation which has got potential to identify and extract the target portion of an image from its actual background on the principal of distribution of gray levels in an image object. According to Otsu's method, an image is considered to be a two-dimensional grayscale intensity function which contains N pixels including gray levels ranging from 1 to L [4]. As per Otsu's analysis, the number of pixels having gray level i is denoted by f_i. Therefore the probability function (P_i) of gray level i in an image could be written as [4]:

$$P_i = f_i / N \tag{1}$$

For the analysis of bi-level thresholding of an image, the pixels could be divided into two classes C_1 and C_2 respectively. C_1 consists of first tier of gray level (1........,t) and C_2 consists of second tier of gray level (t_{+1}............,L). Therefore, the gray level probability distribution for the two classes could be written as:

$$C_1: P_1 / \omega_1(t) .... P_t / \omega_1(t) \tag{2}$$

And $C_2: P_{t+1} / \omega_2(t), P_{t+2} / \omega_2(t), ..., P_L / \omega_2(t)$ (3)

Where $\omega_1(t) = \sum^{t}_{i=1} P_i$ and $\omega_2(t) = \sum^{L}_{i=t+1} P_i$

Otsu's method could also be applied for M number of classes assuming that there are M-1 thresholds, $\{t_1, t_2............, t_{M-1}\}$ which divide the original image into M classes: C_1 for [1......,t_1], C_2 for [t_1+1........,t_2].......,C_i for [t_{i-1}+1.........,t_i] and C_m for [t_{M-1}+1..........,L].

In order to implement image segmentation using Hadoop, the image file in a bundled form is converted into HIPI format with HIB extension before it becomes the part of main configuration files for mapping and reducing. Once the image file is successfully converted to the OpenCV compatible format (Mat), then the image file is passed to the Mapper so as to enable the task distribution to the java threads. Before processing the image data, the Mapper ensures that the images are in the single channel format (grayscale format). To smoothen up the images, the Gaussian blur is applied after setting the Kernel size. Once the Gaussian blur is applied to the images, the Region of Interest (ROI) boundary is set so as to apply the Otsu's threshold. After applying the Otsu's threshold, the ROI pixels are stored in a variable before it is passed to the Reducer. Fig. 2 represents the functioning of the Mapper.

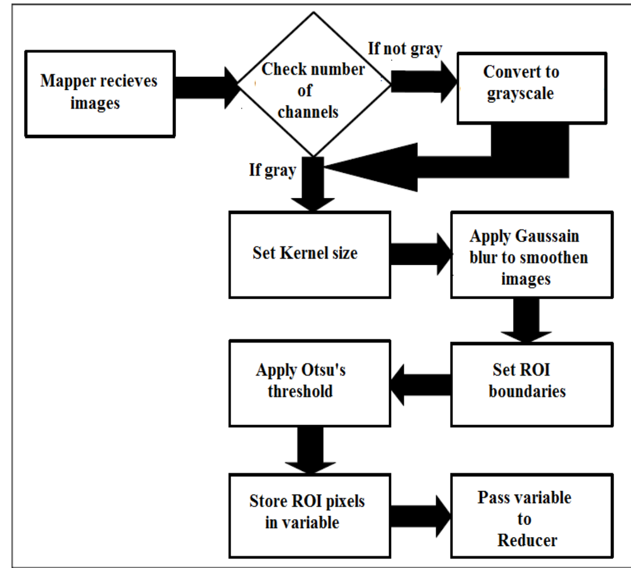

Fig. 2 Illustration of Mapper

From Fig. 2, it could be inferred that once the Mapper is done with the ROI pixels storage in the variable, then the variable is passed to the Reducer. In Fig. 3, the functioning of the Reducer is shown.

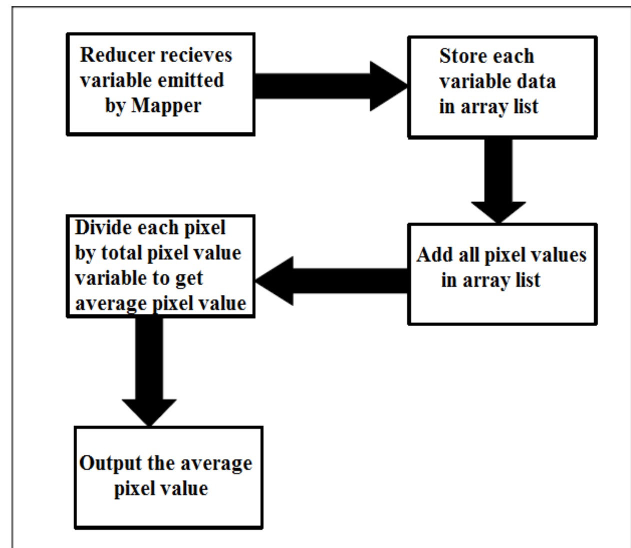

Fig. 3 Illustration of Reducer

The Reducer receives the variable and stores it in an array list and sums up all the pixel value of the ROI in that particular array. In order to get the average pixel value of the ROI from all the segmented images, the Reducer divides the total summed up pixel value from the total number of variable passed by the Mapper.
Fig. 4 and Fig. 5 show the pseudocode of Mapper and Reducer respectively to design parallel image segmentation algorithm:

1. *Input image .JPG format*
2. *Covert image to HIB.Dat*
3. *Pass image to Mapper<HipiImageHeader, FloatImage, IntWritable, IntWritable> //IntWritable is a HIPI data type*
4. *Get image resolution*
5. *Check image channel*
   if(! GRAYSCALE)
   covert to grayscale
6. *Set kernel size (width x height pixels) for Gaussian blur parameter*
7. *Apply Gaussian blur*
   opencv_imgproc.GaussianBlur(SourceImage, TargetImage, size, (0,0)); //(0,0 is the anchor point)
8. *Apply OTSU's threshold*
   opencv_imgproc.threshold(SourceImage, TargetImage, 0, 255, opencv_imgproc.THRESH_OTSU);
9. *Count non-zero pixels*
10. *Emit Resultant image Mat to Reducer*
    context.write(new IntWritable(1),new IntWritable (non_zero pixels));

Fig. 4 Pseudocode for Mapper

1. *Reducer receives image Reducer<IntWritable, IntWritable, IntWritable, Text>*
2. *Initialize a counter and iterate over IntWritable/int records from Mapper*
3. *Check the count of total image samples to determine the average pixel value // Emit output of job which will be written to HDFS context.write(key, new Text(result));*
4. *Output the resultant average pixel value*

Fig. 5 Pseudocode for Reducer

4. Results

The technologies used in the methodology section are scalable and can be used on a cluster of machines. However, the experiments are not performed on a cluster but on a single quadcore machine with 3.40 GHz clock frequency and 8 GB RAM running on Ubuntu 14.04-Linux 64 bits (used for both parallel and sequential mode) to test the single node performance and the version of Hadoop used is 2.6.1. The image datasets are taken from CVonline image database which is commonly used by researchers for downloading the image datasets [14]. Our aim is to analyze the performance of the proposed image segmentation method to investigate the threshold data size point at which it outperforms sequential programming mode in terms of task execution time using multiple threads of CPU cores. If initially, we can achieve the optimum performance in the single node, then it will be easier for us to replicate it on the cluster of machines to process bigger datasets.

For this study, the tasks were run for image segmentation comprising of 100 MB, 200 MB, 250 MB, 257 MB, 260 MB, 300 MB, 350 MB, 400 MB, 450 MB and 500 MB image dataset using Hadoop distributed mode with HIPI and OpenCV sequential. This should provide a clear understanding on the execution time of parallel programming mode and sequential programming mode. The platform used to implement OpenCV sequential mode is Visual Studio Integrated Development Environment 2010 version.
Fig. 6 shows the results of task execution time for the image datasets with varying size ranging from 100 MB to 500 MB to determine the data size threshold at which proposed image segmentation algorithm using HIPI outperforms OpenCV sequential programming. The graph in Fig. 7 summarizes the maximum CPU cores usage attained by different size of image datasets.

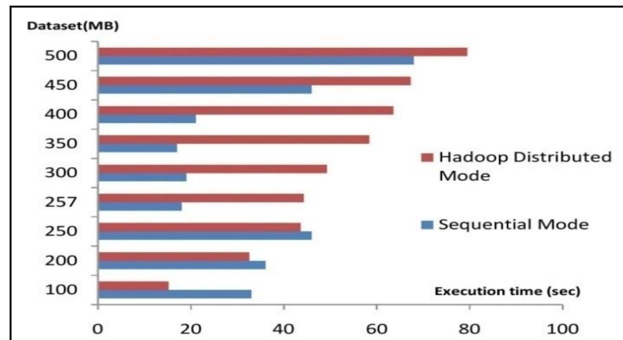

Fig. 6 Task execution time for different image datasets

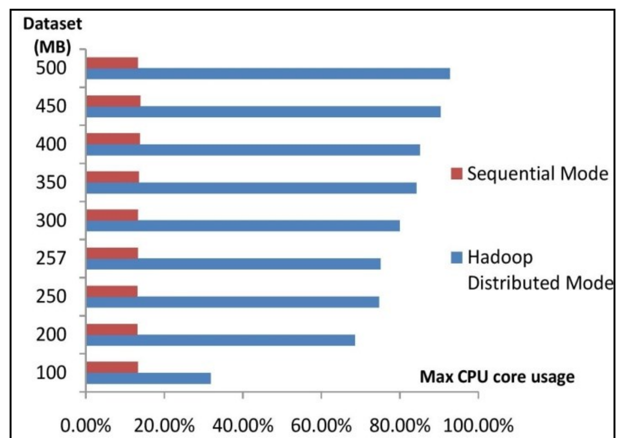

Fig. 7 Maximum CPU core usage for different image datasets

In terms of task execution time for smaller size of image datasets i.e., less than 257 MB, OpenCV sequential mode performs better than Hadoop distributed mode which could be observed clearly from Fig. 6. This is due to the fact that the master thread worker takes a fixed amount of time slot to process each chunk of image data sequentially without any thread context-switching where as in case of Hadoop distributed mode, the factor of split size comes into consideration which causes delay in task execution. The input split size set for the proposed experiment is 128 MB. This is the most compatible split size with Hadoop 2.6.1 version to obtain the optimum output [1]. For smaller datasets, i.e., lesser than 257 MB, Hadoop spawns either 1 or 2 mapping task. If the dataset size is lesser than 128 MB, then Hadoop spawns only 1 map task and if the dataset size is more than 128 MB, then Hadoop spawns 2 map tasks.

Unsurprisingly, image segmentation done using sequential programming has a relatively stable CPU core usage which averages around 13% over the entire execution. However, a theoretical CPU core usage is of 14%. The 1% difference is due to the I/O disk usage operation. It is also to be noted that the image segmentation implemented sequentially is totally cache bound. However, if the application wants to access the memory that is not in the cache then it might have to compete with the other memory access of numerous cores an if the application wants to write to the memory location, then there might arise a cache eviction(s) for other cores.

To analyze our study further in terms of CPU core usage, and task execution time, we will use 50 MB to 500 MB image dataset.

4.1 Evaluating the Performance of 100 MB and 500 MB Image Data on Task Execution Time and CPU Core Usage

Firstly, the performance result using 50MB image dataset to evaluate the impact of CPU cores usage along with task execution time of Map-Reduce job is shown. This evaluation was done using BytesWritable data type and a constant key-value pair size of 1 KB. Analysis of the cores usage along with different segment of task execution time is done. As per Fig. 7, the maximum CPU core usage attained for 50 MB image dataset is 27.01%. Fig. 8 shows the distribution of CPU cores usage over various time segments for the implementation of 50 MB to 500 MB size of image dataset.

It is clear from the Fig. 8 that the maximum CPU cores usage for 50 MB image dataset is attained at the 11th second which is the middle value of the total task execution time.

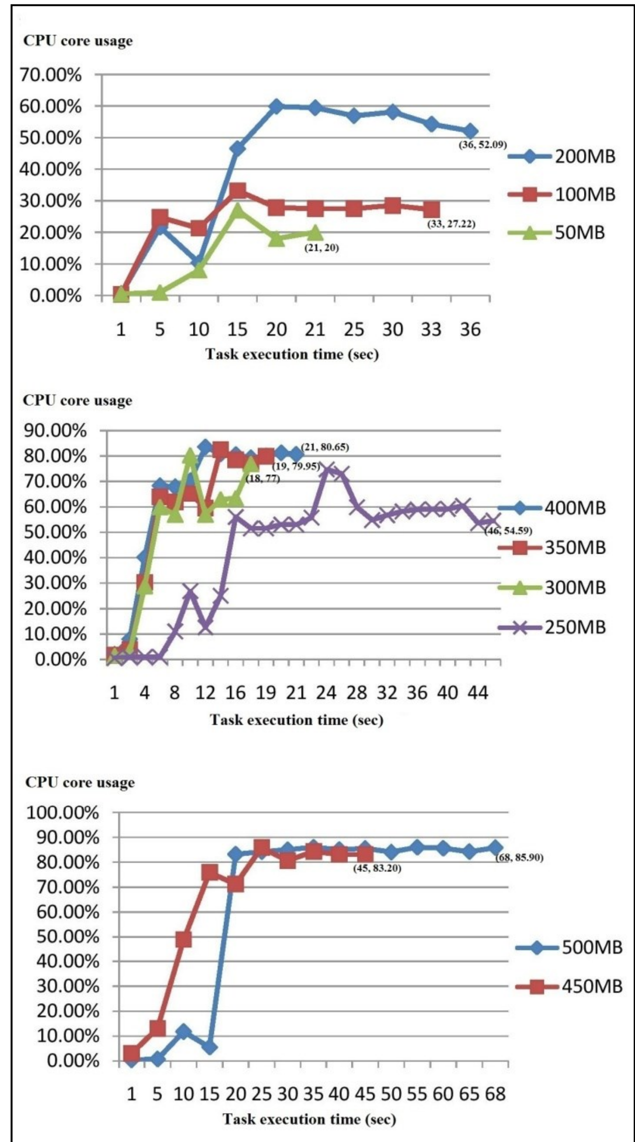

Fig. 8 Distribution of CPU cores usage over time for 50 MB to 500 MB image dataset in Hadoop distributed mode

Smaller datasets which are lesser than 100 MB are unable to exploit the multiple cores threads due to the fact that the split size set for the block for the proposed experiment is 128 MB. Therefore, for the implementation of 50 MB image dataset, the number of input split(s) and the number of spawned map task is only 1. In addition to this, not even half of the threads of the single Hadoop block are allotted to execute the job, as a result of which multiple CPU cores threads are unable to get harnessed. From Fig. 7, it could be observed that, the maximum CPU cores usage value attained for 100 MB image dataset is 33.23%. As per Fig. 8, which shows the CPU cores usage distribution for 100 MB image dataset over various time segments, it could be observed from the graph that the maximum CPU cores usage is attained at the 15th second which is again almost the middle value of the total task execution time. There is an additional 6.22% increment in the CPU cores usage for 100 MB image data if compared with 50 MB image data. It is worth to be noted that for 100 MB image data, majority of the threads of the 128 MB block is put into action to

execute the job. However, since the size of the image dataset does not cross 128 MB, therefore, the number of input split and number of spawned map task is only 1.

Now let us focus on the 200 MB image dataset, from Fig. 7, it could be observed that the maximum CPU cores usage attained for 200 MB image dataset is 59.90%. Moreover, from the graph in Fig. 8, the CPU cores usage distribution of 200 MB image dataset, it could be observed that there is a wide gap between the maximum CPU cores usage between 100 MB image data and 200 MB image data. Since the 200 MB image dataset is greater than the single block size of 128 MB. Therefore, 200 MB image dataset is divided into 2 Hadoop blocks, as a result of which 75 % of the Hadoop threads in total of two blocks together are allotted to complete the task execution of 200 MB image dataset as per the Hadoop log files record. For 200 MB image dataset, the number of split and the spawned map task is 2 since it is allotted 2 blocks. It is worth to be noted from the graph in Fig. 8, that for the size of 200 MB image dataset, maximum CPU cores utilization is achieved at 20th second which again lies at the middle of the total task execution time. It is worth to be noted that for all the three image datasets, i.e., 50 MB, 100 MB and 200 MB, the maximum CPU cores utilization is achieved at the middle of the task execution time.

4.2 Image Datasets Between 200 MB and 500 MB

This section will analyze the CPU cores usage distribution for 250 MB, 300 MB, 350 MB, 400 MB, 450 MB and 500 MB image datasets. At first the CPU cores usage distribution for the size of 250 MB image dataset would be discussed. From the graph in Fig. 7, the maximum CPU cores usage attained for the size of 250 MB image dataset is 74.69% and its associated distribution of CPU cores utilization could be observed from Fig. 8.

The 250 MB image dataset totally gets divided into 2 Hadoop blocks, each comprising of 128 MB. Therefore, the total number of input splits and the number of spawned map task is equal to 2. Since 250 MB image dataset is almost divided exactly into 2 Hadoop blocks, therefore, all the Hadoop threads of two blocks are utilized to execute the job as a result of which CPU cores usage is more than 50% at most of the time interval segments post 15th second. For 250 MB image data also, it is observed from the graph in Fig. 8, that the maximum CPU cores usage of 74.69% is attained at the 25th second which again lies near the middle value of the total task execution time. Now let us come to the 300 MB image dataset. From the graph in Fig. 7 it could be observed that the maximum CPU cores usage attained for 300 MB image dataset is 80.34%. The 300 MB image dataset totally gets divided into 3 Hadoop blocks of 128 MB each.

Therefore, the number of splits and the number of spawned map task is equal to 3 due to which Hadoop threads from 3 blocks are brought to action to execute the job as results of which the CPU cores are utilized up to 80% if compared to the maximum CPU cores utilization for 250 MB image dataset which was 74.69%. This shows that there is an increment of 5.31% of CPU cores usage when image dataset is divided into 3 Hadoop blocks. For 300 MB image data also, the maximum CPU cores usage is attained at the 10th second which again lies near middle value of the total task execution time as observed from Fig. 8, and after attaining the maximum CPU cores usage, there is a stable CPU cores utilization of more than 60 %.

Similarly for 350 MB image dataset, the maximum CPU cores utilization is 82.47% which could be observed from the graph in Fig. 7. For 350 MB image dataset, the maximum CPU cores usage is attained at the 14th second after which there is a stable CPU cores usage of more than 75% as shown in Fig. 8. The 350 MB image dataset again gets divided into 3 Hadoop blocks of 128 MB each. Therefore, the total number of input split and the total number of spawned map task is equal to 3. Therefore, all the threads of the first two Hadoop blocks and approximately 75% threads of the third block is utilized to execute the 350 MB image segmentation job in parallel using PAA deployed on Hadoop framework as per the Hadoop log files record.

For the 400 MB image data, it could be observed from the graph in Fig. 7 that the maximum CPU cores usage attained is 83.58%. It could also be observed from the graph in Fig. 8 that for 400 MB image dataset, the maximum CPU cores usage is attained at the 12th second which lies again near the mid-point of the total task execution time and then after a stable CPU cores usage of more than 80% is observed till the finish time. For 400 MB image dataset, the number of split size and the number of spawned map task is equal to 4 which clearly specifies that the 400 MB image dataset is divided into 4 blocks. Therefore, 100% threads of the first three blocks and less than 20% threads of the fourth block are utilized to execute the job as per the Hadoop log files record. Since minimum number of threads from the fourth block is used, therefore the difference in the maximum CPU cores usage is not much if compared to 350 MB image dataset.

Now let us move to the 450 MB image dataset. From the graph in Fig. 7, it could be observed that the maximum CPU cores usage attained is 85.92 %. From the graph in Fig. 8, it could be observed that the maximum CPU cores usage for 450 MB image dataset is attained at 25th second and after attaining the maximum value there is a stable CPU cores usage of more than 82%. The 450 MB image dataset totally gets divided into 4 Hadoop blocks of 128 MB each as a result of which 100% thread usage of the first three blocks and more than 50% thread usage of the fourth block is done to execute the job of image segmentation in parallel using PAA as per the Hadoop log files record.

Similarly for 500 MB image dataset, the maximum CPU cores usage attained is 86.06% which could be observed from the graph in Fig. 7. Since the Hadoop block size set for the proposed experiment using Hadoop framework is 128 MB, therefore, the 500 MB image dataset is divided

into 4 Hadoop blocks, as a result of which 100% threads of the first three blocks and more than 80% threads of the fourth block is allotted to execute the job as per the Hadoop log files record. It is worth to be noted from the graph in Fig. 8, that after attaining the maximum CPU cores usage at 25th second, there is a stable CPU cores usage of around 85% till the job finishes at 68th second.

4.3 CPU Cores Usage Analysis for Image Segmentation Using Sequential Programming

In this section, the analysis of the CPU core usage for the implementation of image segmentation using sequential programming mode along with different segment of task execution time is done.

The graphs in Fig. 9 show the distribution of CPU cores usage for the execution of various sizes of image datasets using sequential programming. It could be observed from all the graphs that there is a stable CPU core usage of 13%-14% for all the size of image dataset due to the fact that sequential programming does not take multi-cores usage into consideration. The initial spike like trend in all the graphs arises in sequential implementation only when the degree of Input-Output bound process increases. It is also worth to be noted that if the users want to leverage on a lower end machines to carry out image processing tasks with lower size of image dataset, then the sequential computing is preferable over parallel computing.

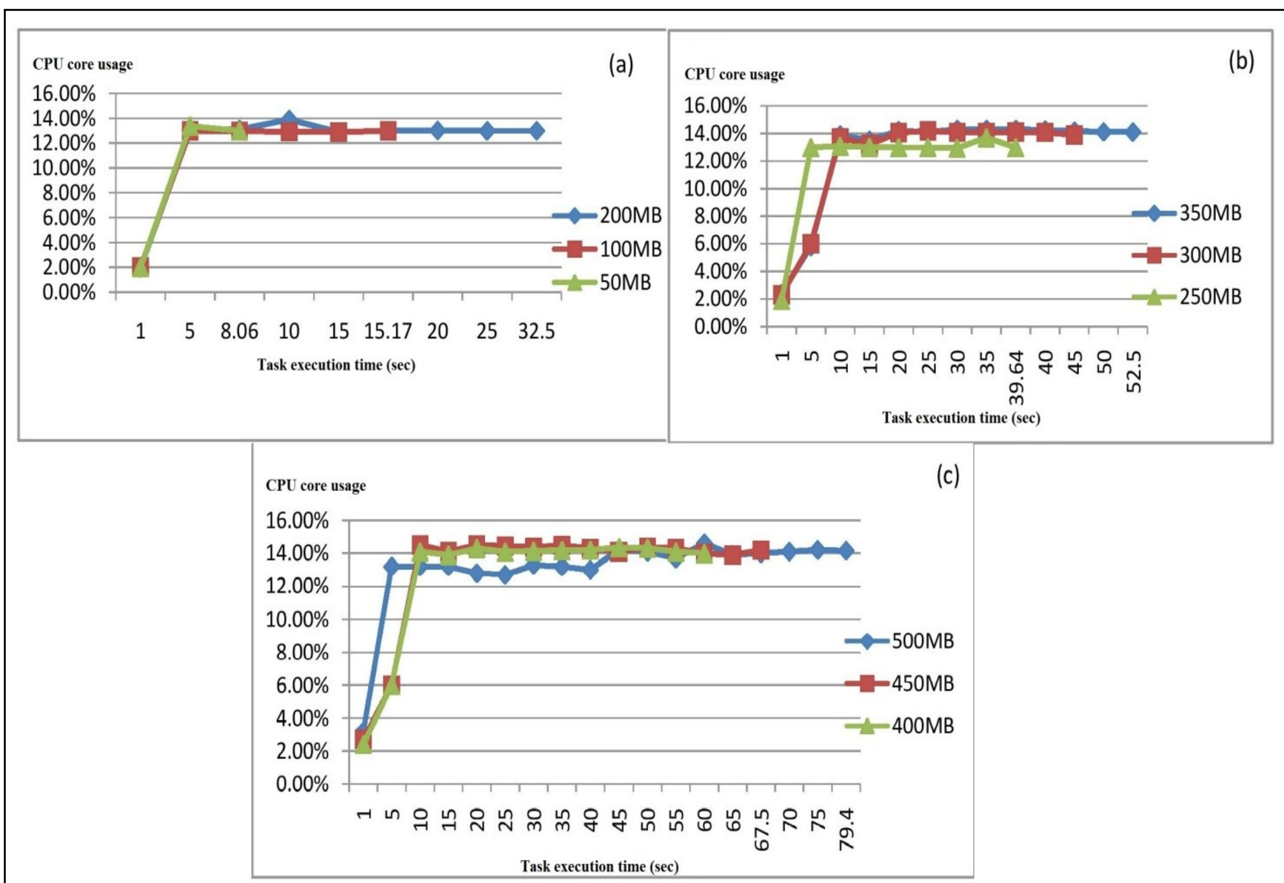

Fig. 9: Distribution of CPU cores usage for (a) 50 MB to 200 MB image dataset in sequential programming mode, (b) 250 MB to 350 MB image dataset in sequential programming mode, and (c) 400 MB to 500 MB image dataset in sequential programming mode

5 Conclusion

To evaluate the threshold of the data size at which the proposed parallel image segmentation method outperforms sequential programming method, we draw a comparison between parallel programming approach using Hadoop Map-Reduce distributed method and sequential programming approach using OpenCV. From the above results, it could be clearly inferred that for scaled-up image datasets, the proposed parallel image segmentation method tends to be far more superior compared to OpenCV sequential mode due to the fact that parallel image segmentation maximizes the CPU cores usage to increase the degree of task parallelization. However, it is also advisable to evaluate smaller image datasets (i.e. up to 250 MB) using sequential programming rather than going for parallel programming. Moreover, a uniform task mapping and reducing could be observed as the image dataset starts expanding. In addition to this, our focus is also on increasing the efficiency of a single node in terms of performance.

However, the conventional wisdom in academics and industry is to scale out using a cluster of commodity computer machines for better distribution of workloads rather than going for scaled-up systems by adding more resources to it. In our case, we have emphasized on task execution time, CPU core usage, input split size and task mapping and reducing for single node. However, the performance characteristics of Hadoop could be fundamentally different, if it is implemented on networked cluster of machines for which the resulting bandwidth/latency characteristics will have an important impact. As of now Hadoop is only compatible with Ethernet networks which follow TCP/IP protocol. Moreover, in order to increase the network throughput efficiency, Hadoop is working on InfiniBand too. In a nutshell, it could be clearly stated that in order to process small scale dataset (up to 250 MB), sequential processing could be effective if compared to parallel programming algorithms. However, since the technology is heading towards big data challenges, it is highly encouraged that programmers should try to adopt parallel programming method to process high scale data.

ACKNOWLEDGEMENT

The authors would like to acknowledge School of Aerospace Engineering, Universiti Sains Malaysia and the Institute of Postgraduate Studies (IPS), Universiti Sains Malaysia for the Global Fellowship [USM.IPS/USMGF(06/14)] financial support to carry out this research. This research is also supported by the School of Electrical and Electronic Engineering of Universiti Sains Malaysia and Informatics Institute, University of Amsterdam.

References

[1] T. White, *Hadoop: The definitive guide*: " O'Reilly Media, Inc.", 2012.

[2] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence, "HIPI: A Hadoop image processing interface for image-based Map-Reduce tasks," *Chris. University of Virginia,* vol. 2(1), pp. 1-5, 2011.

[3] B. Rajitha, A. Tiwari, and S. Agarwal, "Image segmentation and defect detection techniques using homogeneity," in *Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015 International Conference on*, 2015, pp. 678-683.

[4] N. Otsu, "A threshold selection method from gray-level histogram," *IEEE Transactions on System Man Cybernetics,* vol. 9, pp. 62-66, 1979.

[5] Ali, M., Siarry, P. and Pant, M., "Multi-level Image Thresholding Based on Hybrid Differential Evolution Algorithm. Application on Medical Images", *Metaheuristics for Medicine and Biology*, vol. 8(1), pp. 23-36, 2017.

[6] Costantini, L. and Nicolussi, R., "Performances evaluation of a novel Hadoop and Spark based system of image retrieval for huge collections",*Advances in Multimedia*, vol. 20(1), pp.11-16, 2015.

[7] Durad, H., Kazmi, W. and Akhtar, M.N., "Parallel lossless image compression using MPI", *VAWKUM Transactions on Computer Sciences*, vol. 4(2), pp.11-19, 2015.

[8] Firdousi, R. and Parveen, S.,"Local Thresholding Techniques in Image Binarization", *International Journal of Engineering And Computer Science*, vol. 3(3), pp.4062- 4065, 2014.

[9] J. Cadima and I. T. Jolliffe, "Loading and correlations in the interpretation of principle compenents," *Journal of Applied Statistics,* vol. 22, pp. 203-214, 1995.

[10] U. Demšar, P. Harris, C. Brunsdon, A. S. Fotheringham, and S. McLoone, "Principal component analysis on spatial data: An overview," *Annals of the Association of American Geographers,* vol. 103, pp. 106-128, 2013.

[11] A. Sangroya, S. Bouchenak, and D. Serrano, "Experience with benchmarking dependability and performance of Map-Reduce systems," *Performance Evaluation,* vol. 101, pp. 1-19, 2016.

[12] Slabaugh, Greg, Richard Boyes, and Xiaoyun Yang. "Multicore image processing with openmp [applications corner]." *IEEE Signal Processing Magazine* 27(2),134-138, 2010.

[13] Kang, Sol Ji, Sang Yeon Lee, and Keon Myung Lee. "Performance comparison of OpenMP, MPI, and Map-Reduce in Practical problems." *Advances in Multimedia,* vol.7, pp 1-7*,* 2015.

[14] CVonline Image Database [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm