Parallelization of Image Segmentation Algorithms

Shu Jiang
University of Florida

Abstract

With the rapid developments of higher resolution imaging systems, larger image data are produced. To process the increasing image data with conventional methods, the processing time increases tremendously. Image segmentation is one of the many image processing algorithms, and it is widely used in medical imaging (i.e. find tumor in MRI), robotic vision (i.e. vision-based navigation), and face recognition. New faster image processing techniques are needed to keep up with the ever increasing image data size. This paper investigates the parallelization of image segmentation techniques: Watershed Transform and K-Means Clustering algorithms. Lastly, K-Means Clustering is combined with Watershed Transform to address the over-segmentation issue of the Watershed algorithm.

Introduction

Image segmentation is one of the many image processing algorithms. It is used mainly to reduce the original image data content for further processing. Image segmentation basically partitions the input image domain into regions, and each region contains pixels with a certain similar property with respect to each other within the region. Image segmentation is widely used in many applications such as medical imaging, robotic vision, and face recognition. Many algorithms have been developed to implement image segmentation; these include K-Means Clustering, Histogram-based, Region Growing, Graph Partitioning, Watershed Transform, Neural Networks, and etc.

With the development of higher resolution imaging technology, the sizes of image data are growing rapidly, and thus require longer processing time if conventional image segmentation method is used. This paper presents parallelization of conventional image segmentation algorithms to address the issue of greater processing demand.

Two image segmentation algorithms are examined and parallelized; they are Watershed Transform and K-Means Clustering. Watershed is chosen because it is widely used and studied, whereas K-Means is chosen because of its simplicity. Watershed Transform partitions an image into valleys of pixels with brinks that are shared by adjacent valleys; these brinks are called watershed lines. K-Means Clustering divides the image data set into K subsets of pixels according to some characteristics of individual pixel.

Watershed Transform is a popular image segmentation algorithm, especially in medical image analysis; however, it has a drawback: over-segmentation. Over-segmentation occurs when the image has many tiny valleys, which cause the Watershed algorithm to over-partition the image. The resulted image thus contains a dense collection of regions that might distort important features in the original image. Depending on the application and image size, over-segmentation might not be a problem in some cases. This paper addresses the over-segmentation issue of the conventional Watershed algorithm by combining it with K-Means Clustering algorithm [2].

Related Work

Image segmentation algorithms with various implementations have been investigated by many authors. Saegusa and Maruyama implemented a K-Means Clustering algorithm on FPGA with custom designed processing circuit units; four such units were used. Saegusa and Maruyama first divided the image to four parts and stored them into four external memory banks, and then four pixels were fed simultaneously to the four processing units to be processed in parallel, Figure-1 [1].

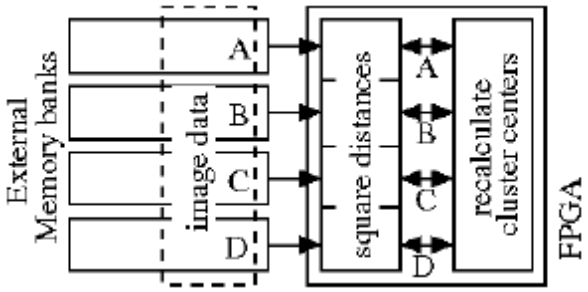


Figure-1: K-Means Clustering on FPGA [1]

In [2], a parallel Watershed Transform was implemented on the cellular neural network (CNN) universal machine. It showed that due to CNN's massively parallel array computing, the Watershed Transform can be parallelized without considering problems such as synchronization, communication, and load balancing. In [3], Moga, Bieniek, and Burkhardt introduced a divide-and-conquer parallel implementation of the Watershed Transform based on rain-falling and hill-climbing simulations. A detailed explanation of the Watershed Transform applied to image segmentation is also presented in [3]. In [4], authors presented a parallel and pipeline implementation of the Watershed Transform on FPGA.

In [5], the method of combining K-Means and Watershed algorithms was introduced to address Watershed's over-segmentation issue. It showed that the resulted images of the proposed segmentation method have 92% few partitions than the images produced by the Watershed alone.

Watershed Transform

Many implementations of the Watershed Transform had been developed [7]. This paper implements the Watershed Transform by immersion for grayscale images. Watershed by immersion was introduced in 1991 by Vincent and Soille [6], Figure-2. Each pixel in a grayscale image can have a value of 0-255, with 0 representing black, 255 for white, and values in between for shades of gray. To understand the idea of Watershed by immersion, imagine a landscape with catchment basins being immersed in a lake and a hole is pierced at the local minimum of each catchment basin. Water will fill up the basins start with their local minima. Dams are built at points where water coming from different basins would meet. When water reaches the highest altitude in the landscape, the immersion process is stopped. At a result, the landscape is partitioned into regions separated by dams, called watershed lines or watersheds [7]. Grayscale images can be imagined as such landscapes made of pixels with various altitudes (0-255). A binary image is produced by the Watershed Transform, 1

(black) is assigned to dams, or watersheds, and 0 (white) assigned to regions surrounded by dams.

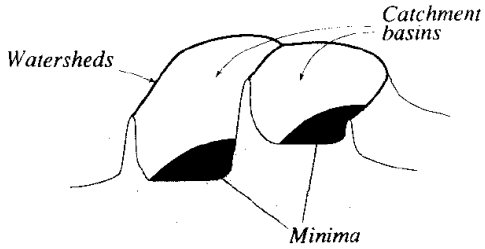


Figure-2: Watersheds, Minima, Basins [6]

Sequential Watershed Transform

The sequential Watershed algorithm implemented in this paper consists of following steps:

1. Read in the image file (.txt) and store image data in a matrix array
2. Find the minimum and maximum altitudes (pixel values) of the input grayscale image
3. Initialize the output image, each pixel in the output image is assigned to the constant INIT (-1)
4. Start with the minimum altitude, assign new distinct labels to each of these minimum, a label is any integer greater than 0
5. Exam each pixel at the next higher altitude
    a. If none of its neighbors is labeled (-1), label it as a new local minimum by assigning a new label to it (any integer > 0)

-1	-1	-1
-1	?	-1
-1	-1	-1

⇒

-1	-1	-1
-1	2	-1
-1	-1	-1

Figure-3

    b. If its neighbors are labeled and all neighbors have the same label, label it as a basin pixel by assigning the label of its neighbors to it

10	10	-1
-1	?	10
-1	-1	10

⇒

10	10	-1
-1	20	10
-1	-1	10

Figure-4

    c. If its neighbors are labeled and neighbors have different labels, label it as a watershed point

5	5	5
-1	?	5
7	-1	5

⇒

5	5	5
-1	13	5
7	-1	5

Figure-5

6. Repeat 3 until the maximum altitude is reached, all pixels in the maximum altitude are labels as watershed points

7. Produce the binary image by giving all watershed points the value 0 (black) and all non-watershed points the value 1 (white)
8. Write the binary output image to a file (.txt)

Steps 4 to 6 are the immersion process. In step 5, each pixel's neighbors are read to determine the final value for the pixel. However, the number of neighbors read can be varied. When all 8 neighbors are read, it is called 8-connectivity; and 4-connectivity for 4 neighbors, Figure-6. The implementation of Watershed for this paper uses 8-connectivity. 4-connectivity has the potential of reducing computation, and communication when the algorithm is parallelized. Lastly, edge pixels of the image are ignored for ease of implementation. The pseudo code for the sequential Watershed Transform implementation is shown below in Figure-7.

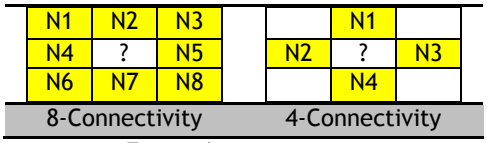


Figure-6: connectivity

```
#define INIT            -1
#define WATERSHED   0

int inputImage[m][n];        //storage input image matrix
int outputImage[m][n];       //storage output image matrix

main()
{
   read input image file, inputImage[m][n] = image file;

   find minimum and maximum altitudes of  the input image;

   initialize the output image, outputImage[m][n] = INIT;

   for(altitude <= the maximum altitude)
   {
      If(inputImage[m][n] == altitude)
      {
         If(no labeled neighbors)
            outputImage[m][n] = a new label;
         if(same labeled neighbors)
            outputImage[m][n] = neighbors' label;
         if(different labeled neighbors)
            outputImage[m][n] =  WATERSHED;
      }
      altitude++;
   }

   If(outputImage[m][n] != WATERSEHD)    //output binary image
      outputImage[m][n] = 1;

   write output image to file, image file = outputImage[m][n];
}
```

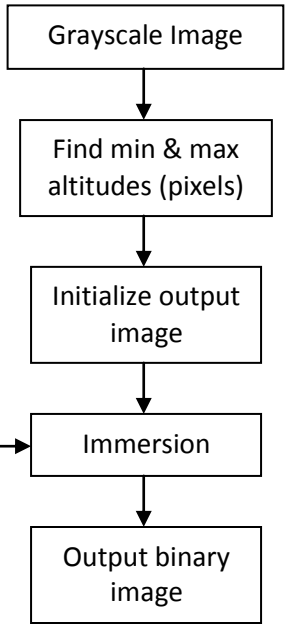
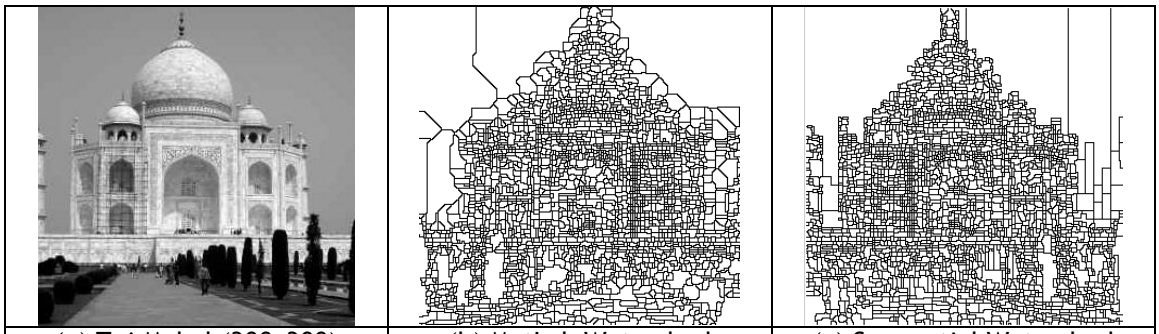


Figure-7: Pseudo code for sequential Watershed Transform with block diagram

The result of the Watershed implemented for this paper is shown here in Figire-5(c). A grayscale image of Taj Mahal (300x300) is segmented. The result of MatLab implementation is also show here in Figure-5(b) for comparison purpose.



(a) Taj Mahal (300x300) | (b) MatLab Watershed | (c) Sequential Watershed

Figure-8: MatLab Watershed vs. implemented sequential Watershed

Parallel Watershed Transform

There are two major classifications of current parallel implementations of the Watershed Transform: domain decomposition and functional decomposition [7]. Domain decomposition divides the image into sub-images and distributes them across processors, and each processor uses the sequential algorithm to process its assigned sub-image. Domain decomposition is more portable than functional decomposition in the sense that one decomposition result of an image can be used for many algorithms if they are processing the same image, thus domain decomposition is used for this paper. Two dominant parallel programming models exist: message-passing programming model and shared-memory programming model. In message-passing model, tasks and data are assigned to processors, and processors interact with each other by initiating explicit communication calls (i.e. MPI_Send() and MPI_Receive()). In shared-memory model, all processor shared a common memory, and processors interact with each other by simply reading and writing to the shared memory space. Shared-memory programming model is used for this paper. Lastly, Unified Parallel C (UPC) is used to write parallel programs for both image segmentation algorithms.

Parallelization of the Watershed algorithm is done by dividing the input image matrix into p strips of sub-matrices (p = # of processors), and each processor applying Watershed to one sub-matrix, Figur-9. In other words, every processor runs the same program but processes different data. Data allocation is done statically before run time. The reason for dividing the image into strips of columns is that the number of rows is usually smaller than the number of columns (i.e. 480x640), thus frequency of communication is a little smaller than dividing the image into slices of rows.

Figure-9: Image data domain decomposition

Among the steps described previous for the sequential Watershed, only steps 3 to 6 are parallelized. Steps 1 and 8, reading from and writing to an image file, require parallel I/O [8]. Steps 1 and 8 are not parallelized because their execution times are assumed to be negligible compare to immersion time, and the complexity of parallel I/O render the effort not worthwhile. Step 2, finding the minimum and maximum altitudes of the image, is not parallelized because two global shared variables are used to store the minimum and maximum altitudes, and locks are needed to prevent more than one processors trying to write to the same memory location at the same time; locks would introduce overhead that might be worse than execute the step sequentially. Barriers are also used in the parallel program to synchronize all processors and making sure no one gets too far ahead of the others. Barriers introduce overhead naturally, but they are necessary to ensure the correctness of the final result.

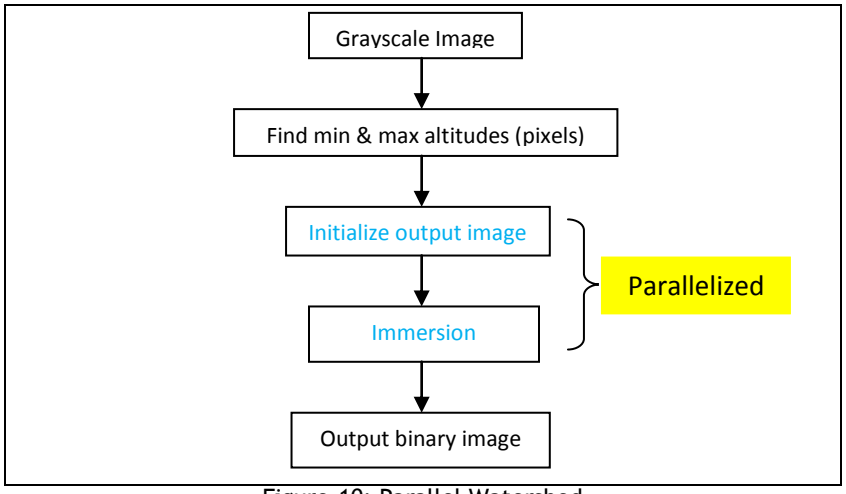
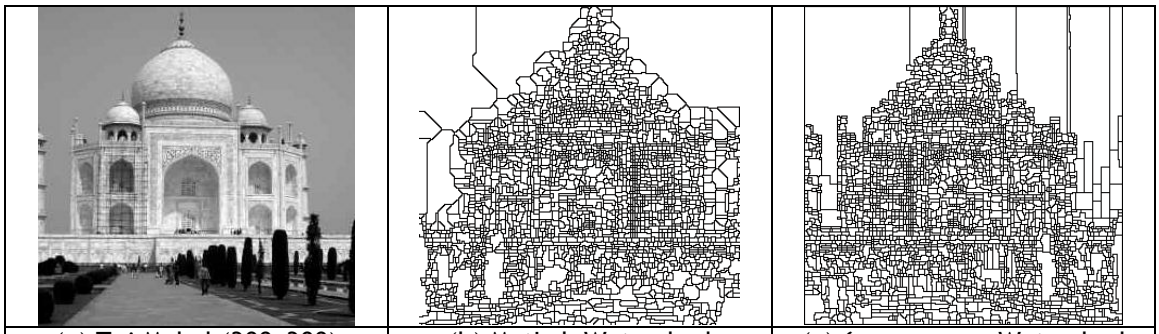


Figure-10: Parallel Watershed

Experiments and Results

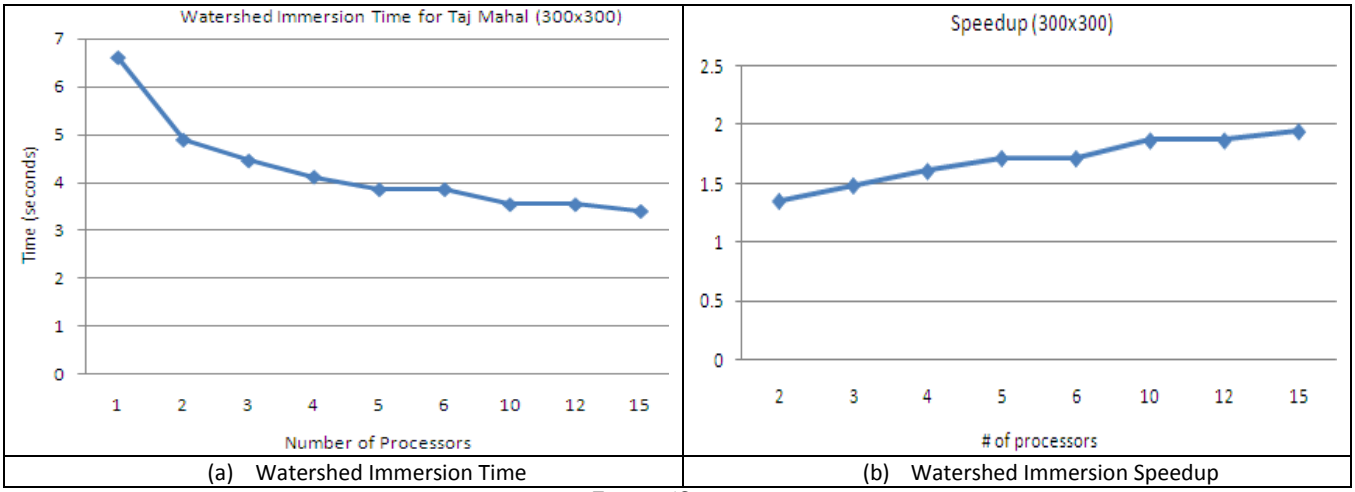
The Marvel machine in the HCS lab is used to conduct experiments to quantify the performance of the parallel algorithms. Marvel has 8 AMD Opteron 880 Dual-core processors (2.4GHz), which give a total of 16 processors. The communication among processors is supported by the HyperTransport link. Marvel also has a 32GB of shared-memory.

A 300x300 grayscale image, Figure-11(a), is first used to test the performance of the parallel Watershed algorithm. The segmented image, Figure-11(c), is produced when 6 processors are used. The execution times of the program for various numbers of processors are also measured to quantify the speedup. As discussed previously, not all parts of the parallel program is being executed in parallel, thus only execution time and speedup of the (parallelized) immersion process of the Watershed algorithm are measured, Figure-12(a). The resulted speedup is not as good as expected and it flattens out as the number of processor increases. In conclusion, the implementation of the parallel Watershed algorithm is not scalable. A larger image (1236x1500) is also tested, and the result produced gives the same conclusion, Figure-13.



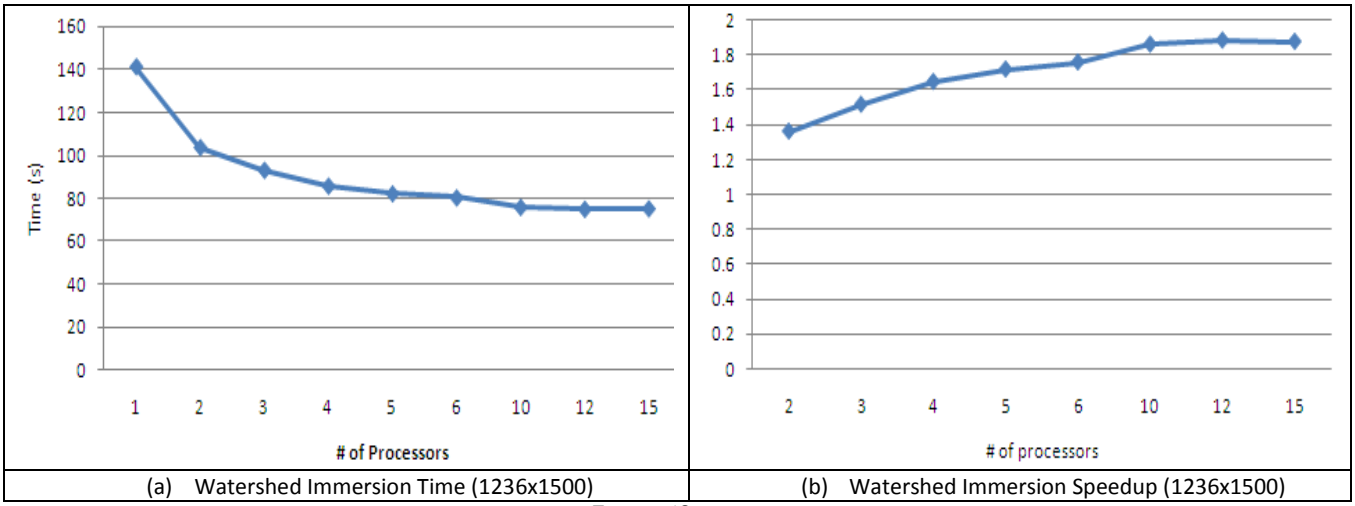
(a) Taj Mahal (300x300) | (b) MatLab Watershed | (c) 6-processors Watershed

Figure-11



(a) Watershed Immersion Time | (b) Watershed Immersion Speedup

Figure-12



(a) Watershed Immersion Time (1236x1500) | (b) Watershed Immersion Speedup (1236x1500)

Figure-13

The performance of the parallel Watershed algorithm is analyzed by the Parallel Performance Wizard [9] to look for possible optimization opportunities. Figure-14 shows the profile metrics pie chart produced by PPW for the parallel Watershed running on 5 processors. Figure-14 also justifies the earlier assumption made during parallelization that the immersion time would dominate the execution time of the whole program. Based on Amdahl's law, in

order to speed up the algorithm, the obvious choice would be to speed up the immersion process of Watershed.

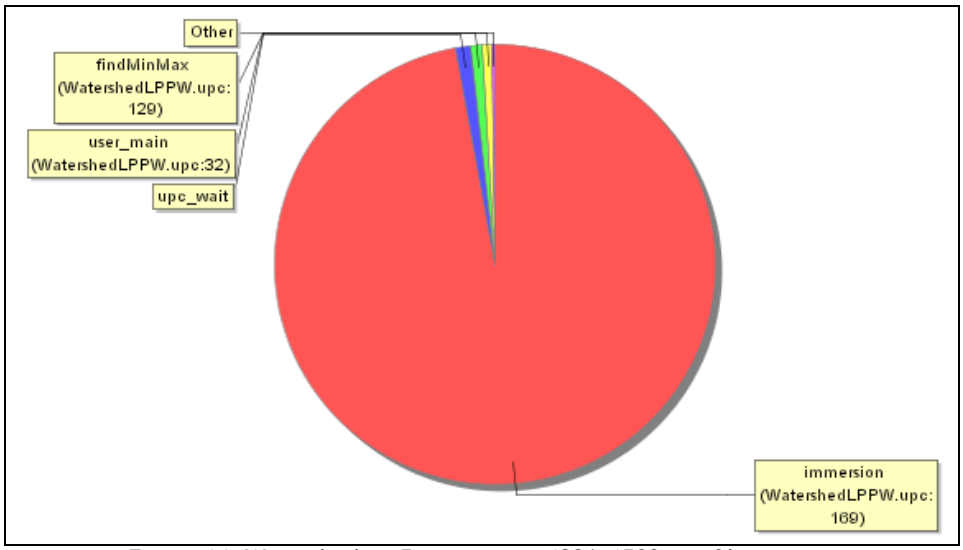


Figure-14: Watershed on 5 processors (1236x1500) profile metrics

One approach to reduce the execution time of the immersion process is to reduce the number of neighbors being accessed for each pixel (step 5). The new immersion process is implemented using 4-connectivity (4 neighbors are access) instead of 8-connectivity in the original immersion process. However, the results produced by the Watershed with 4-connecitivity show that the execution time of the immersion process is not reduced, Figure-15; this result is not expected. Since the new immersion process did not improve the performance of the Watershed algorithm, it is discarded.

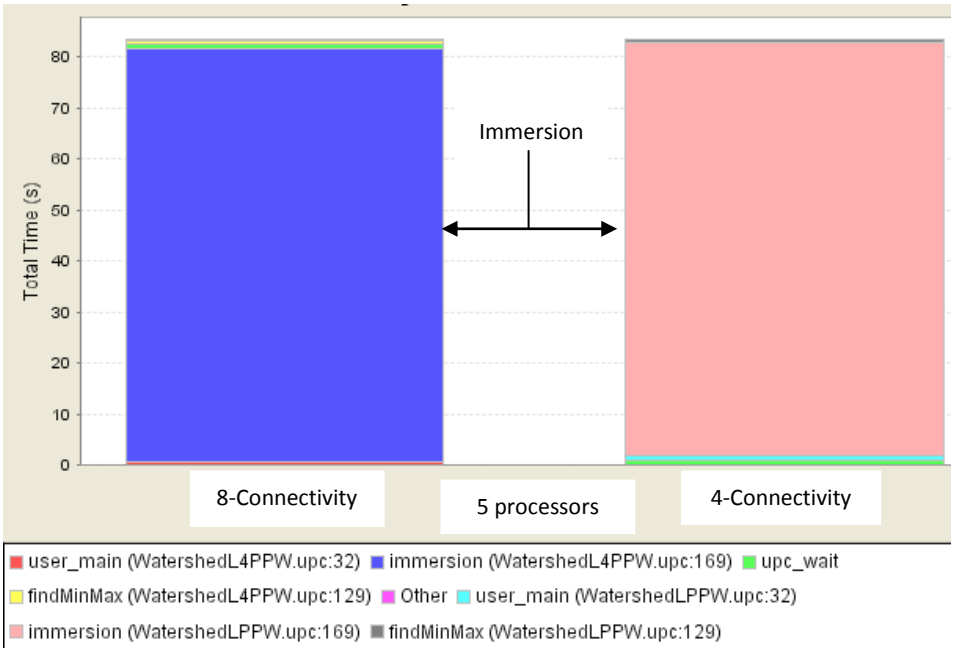


Figure-15: 8-connectivity vs. 4-connectivity

Another approach can be used to reduce the execution time of Watershed is to hide remote accesses with computation by pre-fetching ghost zones [8], Figure-16. For the Watershed algorithm with 8-connectivity, each pixel in the ghost zone would require three remote accesses of neighbor pixels. However, this approach is not implemented after the realization that remote access is not the cause of non-scalability of the Watershed algorithm; the immersion time is not reduced even if remote neighbor pixels are not accessed.

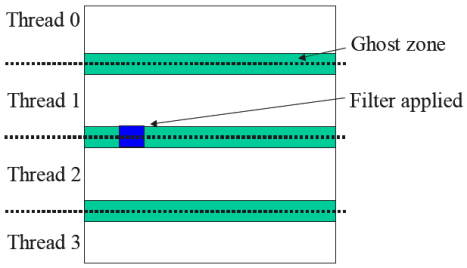


Figure-16: Ghost zone optimization [8]

K-Means Clustering Algorithm

K-Means Clustering algorithm clusters the image data set into k subsets of pixels; each subset has a center value which is the average of all pixels in the subset, thus k subsets resulted in k centers total. A pixel is grouped into a subset by first calculating the distances between the pixel and each center, and then the pixel is grouped into the subset that has the closest center. After one pass through the image, error is calculated, and the clustering process is stopped when the error converged to a value; error is the sum of the squared distances between all pixels in a subset and the subset's center.

Sequential K-Means Clustering Algorithm

The sequential K-Means Clustering algorithm implemented for grayscale images consists of following steps:

1. Read in the image file (.txt) and store image data in a matrix array
2. Find the minimum and maximum pixel values of the input image
3. Initialized K centers with the results from step 2
4. For each pixel
    i. calculate its distance to each center
    ii. cluster the pixel into the subset that has closest center
5. Calculate new K centers; each new K center is the average of all pixel values in its subset
6. Calculate new error, the sum of the squared distances between all pixels in a subset and the subset's center
7. Repeat steps 4 to 6 until error converged to a value
8. Write the output image to an image file (.txt)

The pseudo code for the sequential K-Means Clustering implementation is shown below in Figure-17.

```
#define K
int inputImage[m][n];        //storage input image matrix
int outputImage[m][n];       //storage output image matrix

main()
{
   read input image file, inputImage[m][n] = image file;
   findMinMaxPixels();
   initialize centers, center[K];

   while(error is not converged)
   {
      for(each pixel inputImage[m][n])
      {
          distance = |inputImage[m][n] – center[K]|;
          outputImage[m][n] = closest center[K];
      }

      new center[k] =                                    ;

      error =                                  ∑∑(input;
   }

   write output image to file, image file = outputImage[m][n];
}
```

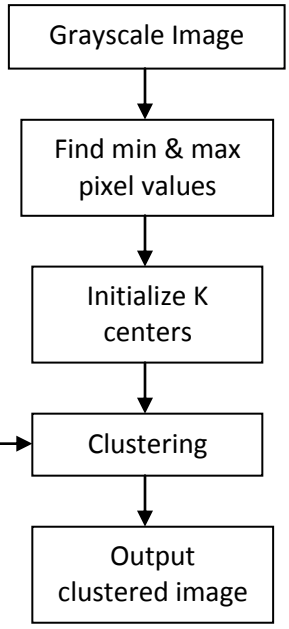
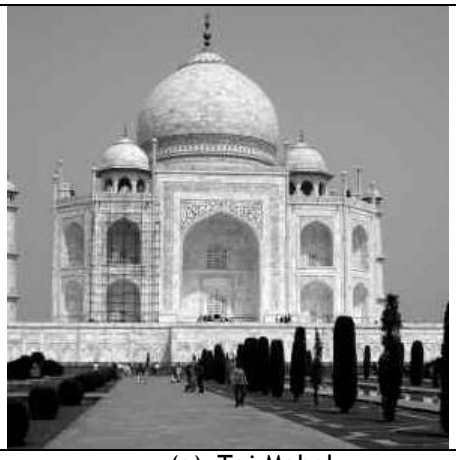
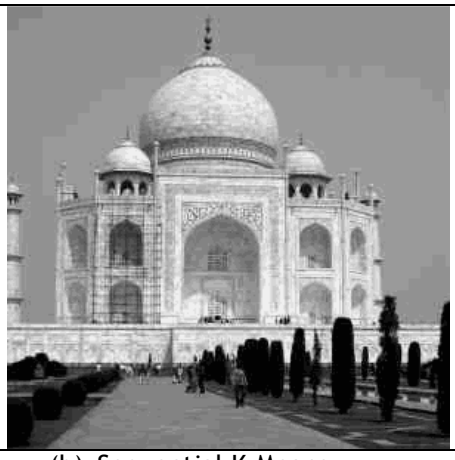


Figure-17: Pseudo code for K-Means Clustering algorithm with block diagram

The clustered image of K-Means for a grayscale image is not as visually obvious as a clustered color image, Figure-18. For a color image, K-Means reduces its original number of colors to K colors. For a grayscale image, K-Means reduces its original 256 possible levels down to K levels. As a result, K-Means effectively reduces the information content of an image while preserving its important features.



(a) Taj Mahal



(b) Sequential K-Means

Figure-18: K-Means clustered images

Parallel K-Means Clustering Algorithm

The implementation of the parallel K-Means Clustering algorithm uses the same parallelization approach (discussed previously) used for Watershed algorithm, data domain decomposition. The input image is again being partitioned into p (# of processors) strips and distribute over p processors, Figure-9. Only steps 4 to 7 (clustering) of the sequential K-Means algorithm are parallelized because they dominate the total program execution time, Figure-19. In step 6 of the sequential K-Means, the sum of all pixels in a subset and the total number of pixels are used to calculate new centers. Since pixels belong to a subset might be distributed across the image (which is distributed across processors), shared memory locations are allocated to store those values. Any processor can update the sum of a subset by adding a pixel to it, thus lock is used to prevent processors from writing to the same memory location.

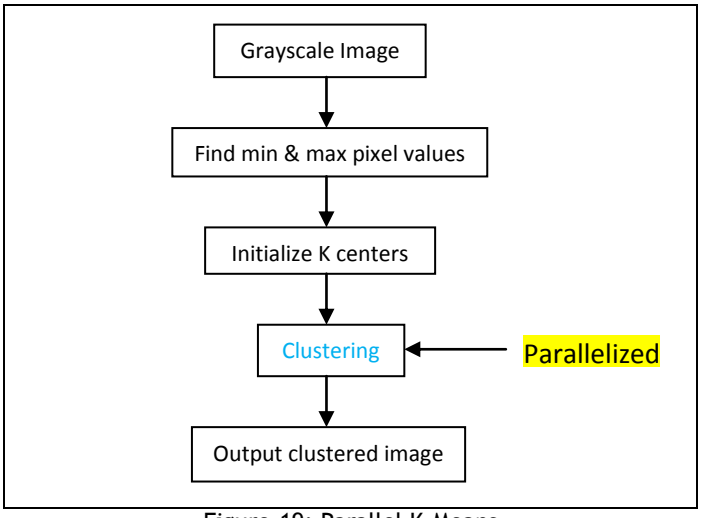
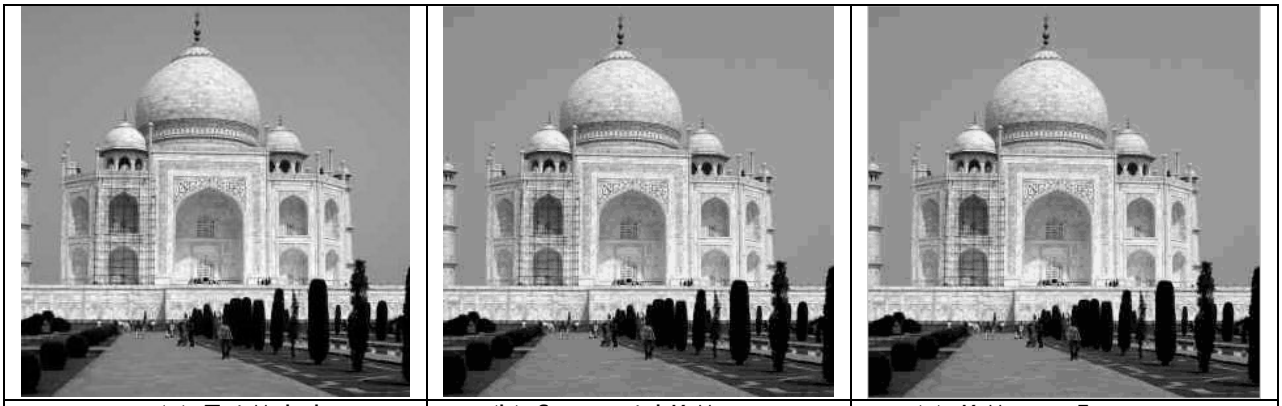


Figure-19: Parallel K-Means

Experiments and Results

The Marvel machine is also used to conduct experiments and quantify the performance of parallel K-Means algorithm. The images produced by both sequential and parallel K-Means are almost the same, Figure-20.



(a) Taj Mahal	(b) Sequential K-Means	(c) K-Means, 5 processors

Figure-20: Sequential and parallel K-Means

The performance of the parallel K-Means is shown below in Figure-21; while the clustering process appears to have a linear speedup, the whole program run time does not. The pie charts in Figure-22 also shows that while the clustering take up most of the total program execution time in the sequential case; overheads are significant when 5 processors are used. These overheads are locks (upc_lock & upc_unlock) and synchronization (upc_wait). In order to improve the performance of the parallel K-Means algorithm, it is necessary to reduce the use of locks. Synchronization is a side effect of locks, thus reducing locks would effectively reduce wait time as well. Scalability is also an issue. When more processors are used, more contention would result because more processors would try to write to the same memory location simultaneously. The overhead would dominate even if the clustering time shows a linear speedup.

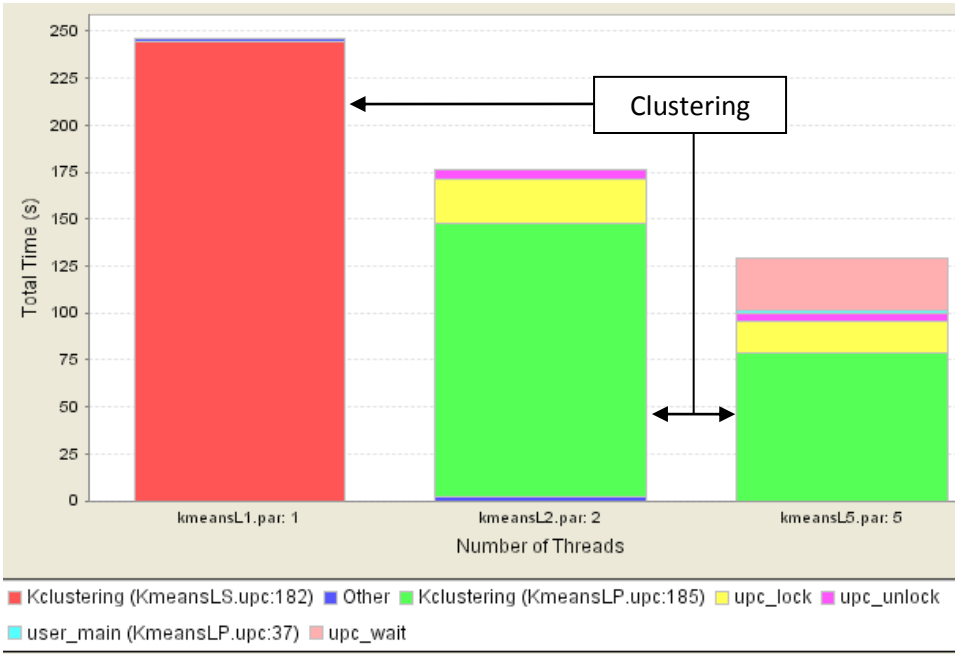


Figure-21: Performance analysis of sequential and parallel K-Means

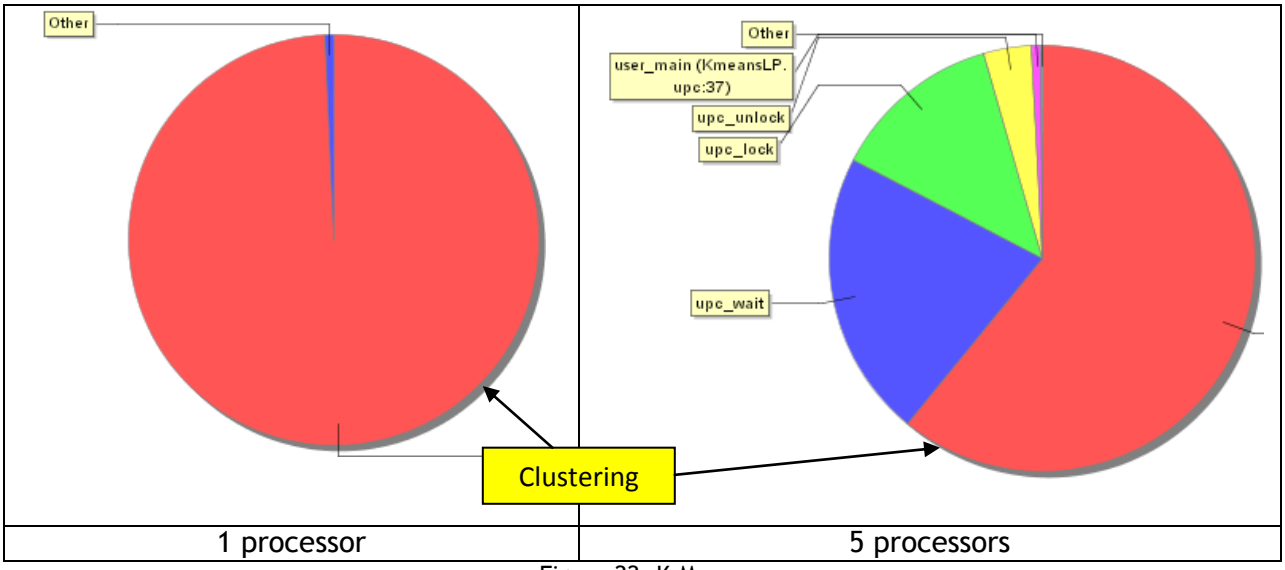
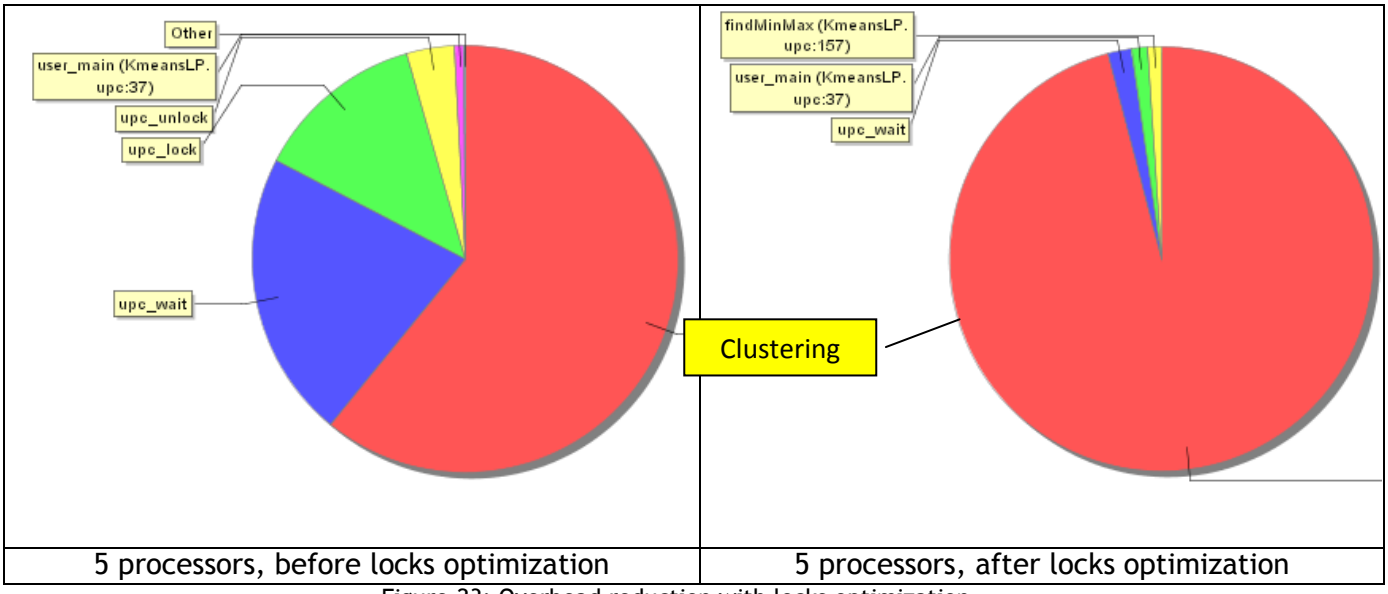


Figure-22: K-Means

The use of locks is reduced by instead of updating shared variables when each pixel is accessed, local variables are used and the accumulated local results are updated to the shared variable after all pixels are processed. The results obtained after the optimization show a significant reduction in overhead. Figure-23 compares the overhead with and without lock optimization with the same number of processors. The synchronization overhead indeed is significantly reduced as well with the reduction in lock overhead.



5 processors, before locks optimization	5 processors, after locks optimization

Figure-23: Overhead reduction with locks optimization

Performance analysis of the optimized parallel K-Means shows good scalability and almost negligible overhead, Figure-24. Also, close to linear speedup is achieved by the optimized parallel K-Means, Figure-25.

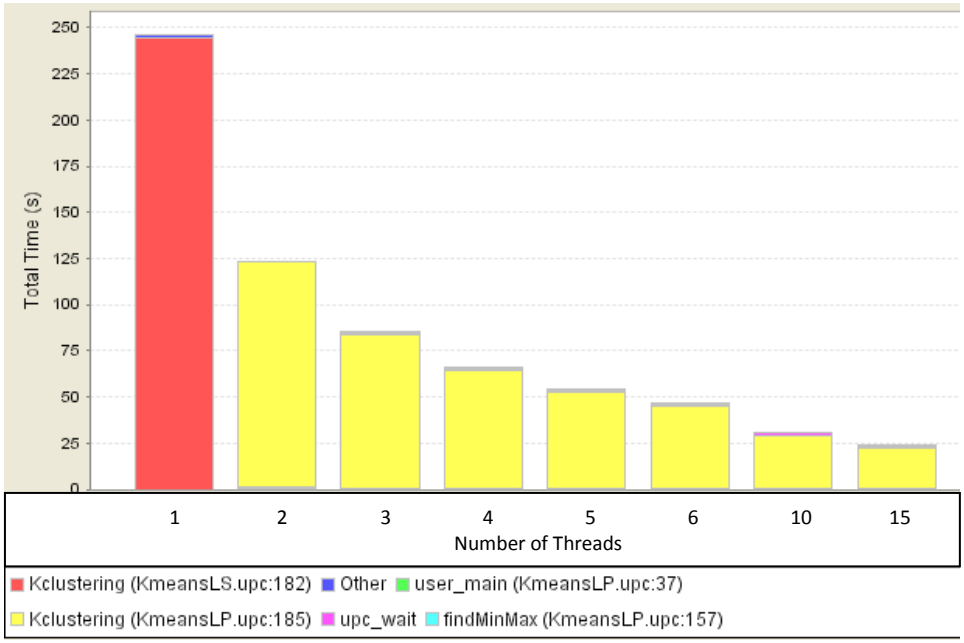
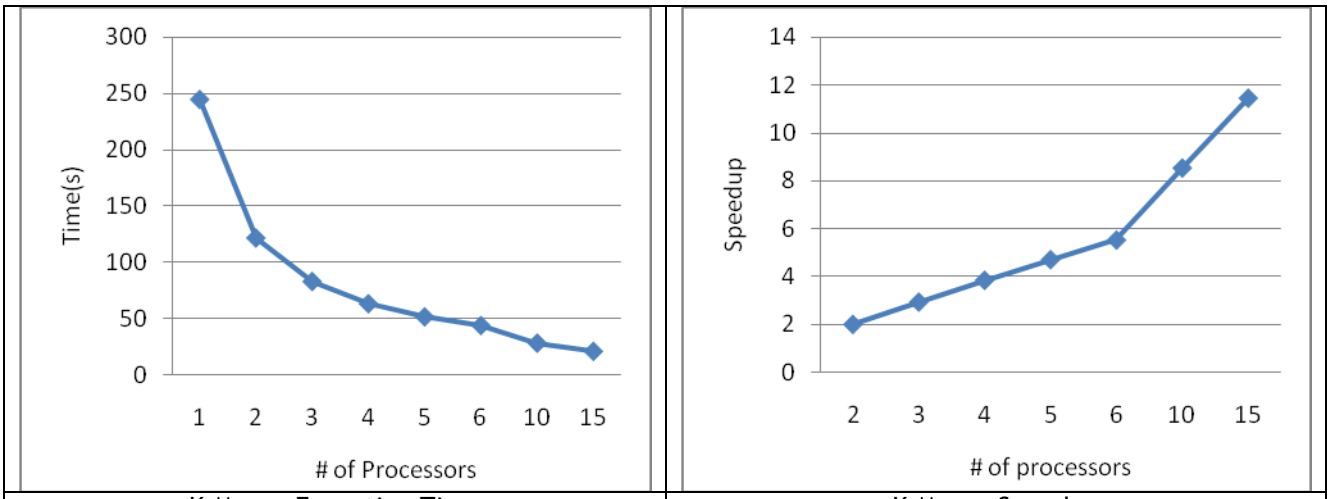


Figure-24: Optimized K-Means

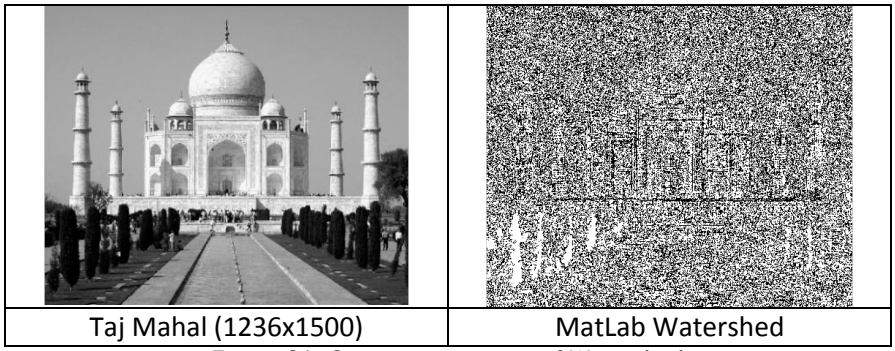


K-Means Execution Time | K-Means Speedup

Figure-25: K-Means Speedup

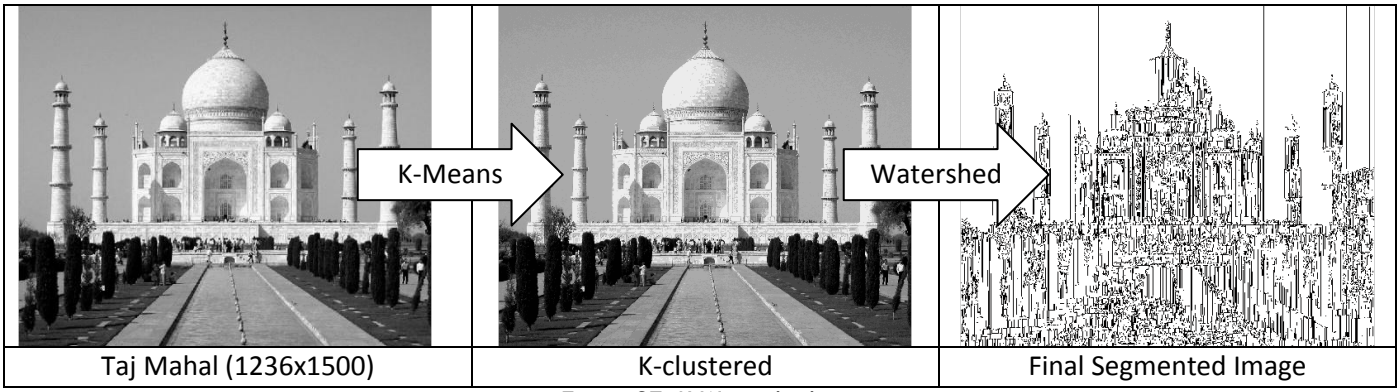
K-Watershed (K-Means Clustering + Watershed Transform)

The over-segmentation issue of Watershed is more apparent when it is applied to large images, Figure-26. However, the issue of over-segmentation can be addressed by combining K-Means with Watershed. K-Means is first applied to the grayscale image to reduce its gray levels, and then Watershed is applied to the K-clustered image to produce a final segmented image, Figure-27.



Taj Mahal (1236x1500) | MatLab Watershed

Figure-26: Over-segmentation of Watershed



Taj Mahal (1236x1500) | K-clustered | Final Segmented Image

Figure-27: K-Watershed

The performance of K-Watershed can be predicted from previous performance analyses of K-Means and Watershed algorithms. The K-Means part would have a near linear speedup, whereas the Watershed part would have a poor speedup, and Watershed would become the bottleneck for the speedup of K-Watershed. The performance analysis of K-Watershed shows that it is indeed the case, Figures 28 &29.

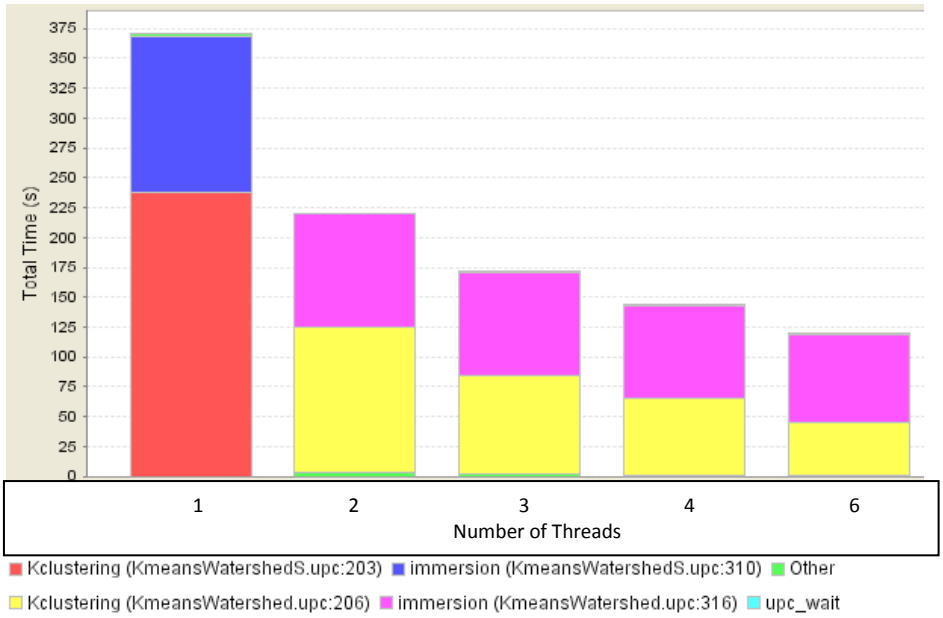
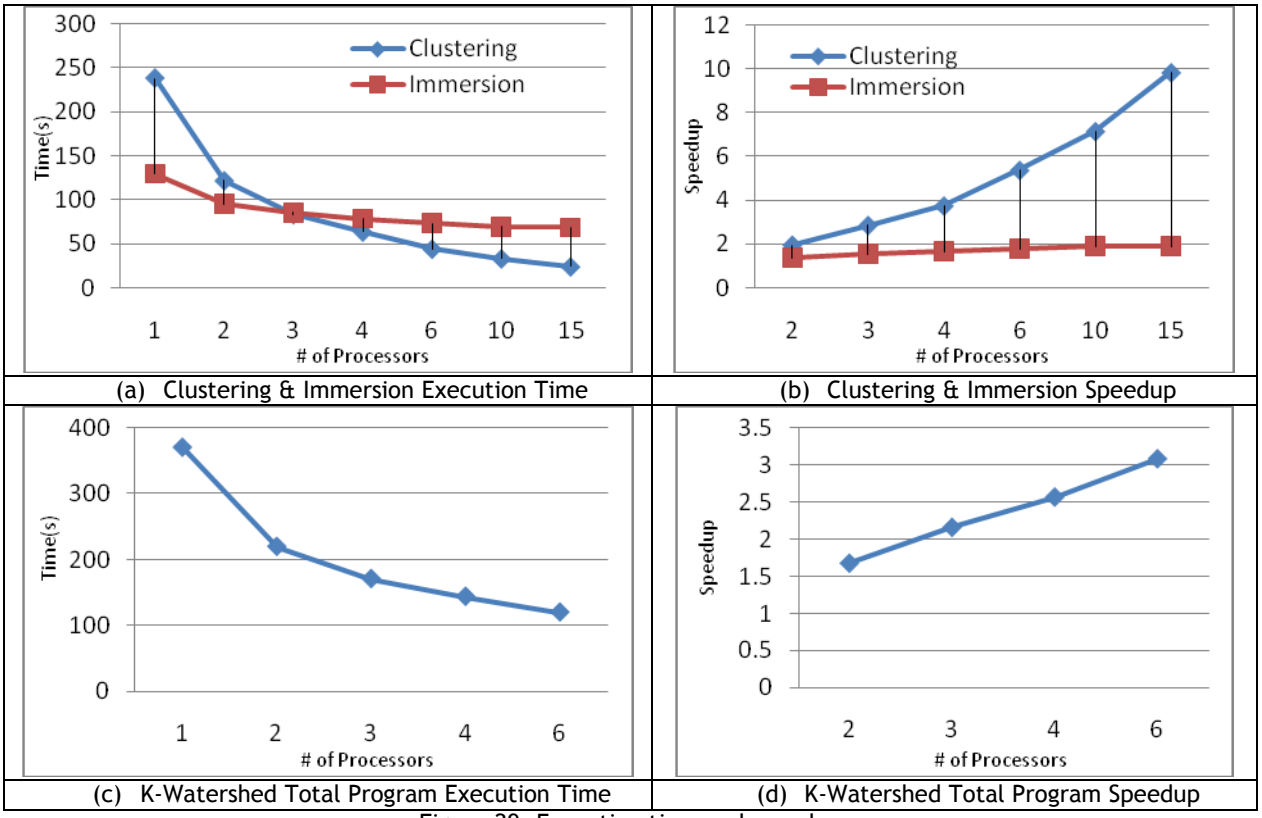


Figure-28: K-Watershed



(a) Clustering & Immersion Execution Time

(b) Clustering & Immersion Speedup

(c) K-Watershed Total Program Execution Time

(d) K-Watershed Total Program Speedup

Figure-29: Execution time and speedup

Conclusion

K-Means Clustering and Watershed Transform image segmentation algorithms have been studied and parallelized. The performances of these two parallel algorithms are also analyzed. Optimizations are also attempted for both algorithms when non-ideal performances are observed. Optimization for Watershed is attempted but without improvement in performance, whereas optimization for parallel K-Means improve its performance significantly. As a result, the parallel Watershed implementation exhibits poor scalability, while the parallel K-Means implementation achieves a close to linear speedup. Lastly, the over-segmentation problem of Watershed is addressed by combining K-Means and Watershed algorithms. The resulted algorithm, K-Watershed, inevitably inherits the performances of its parent algorithms. The speedup of K-Watershed is limited by the poor speedup of Watershed.

References

[1] T. Saegusa, T. Maruyama, "Real-Time Segmentation of Color Images Based on the K-Means Clustering on FPGA", International Conference on Field-Programmable Technology, 2007.
[2] S. Eom, V. Shin, B. Ahn, "Cellular Watersheds: A Parallel Implementation of the Watershed Transform on the CNN Universal Machine", ICICE Trans. Inf. & Syst., Vol.E90-D, No.4 April 2007.
[3] A. Moga, A. Bieniek, H. Burkhardt, "Parallel Watershed Transformation Algorithms for Image Segmentation", Parallel Computing 24, 1998.
[4] D. Trieu, T. Maruyama, "A Pipeline Implementation of a Watershed Algorithm on FPGA", International Field Programmable Logic and Applications, 2007.
[5] H.P. Ng, S.H. Ong, K.W.C Foong, P.S. Goh, W.L. Nowinski, "Medical Image Segmentation Using K-Means Clustering and Improved Watershed Algorithm", IEEE Southwest Symposium on Image Analysis and Interpretation, 2006.
[6] L. Vincent, P. Soille, "Watersheds in Digital Spaces: An Efficient Algorithm on Immersion Simulations", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 13, No. 6, June 1991.
[7] J. Roerdink, A. Meijster, "The Watersehd Transform: Definitions, Algorithms and Parallelization Strategies", Fundamenta Informaticae 41, 2001.
[8] S. Chauvin, P. Saha, F. Cantonnet, S. Annareddy, T. El-Ghazawi, "UPC Manual v1.2", High Performance Computing Laboratory, George Washington University.
[9] A. Leko, M. Billingsley, "Parallel Performance Wizard User Manual", High-Performance Computing & Simulation Research Lab, University of Florida, 2007.