

# Image Histogramming and Connected Components Labeling in Parallel

*Soumya Das*

E-mail: [soumya@winlab.rutgers.edu](mailto:soumya@winlab.rutgers.edu)

December 19, 2003

*(Submitted as partial fulfillment of course requirement in Introductions to Parallel and Distributed Computing (ECE 16:332:566) to Dr. Manish Parashar, Associate Professor, Rutgers, The State University of New Jersey)*

## Introduction:

In this project, two useful primitives in image processing algorithms, histogramming and connected components labeling have been implemented in parallel. An application of histogramming is histogram normalization which flattens the histogram to improve the contrast of an image by “spreading out” colors that are too clumped together for human visual distinction. Connected components labeling (CCL) of an image is a fundamental step in the segmentation process and consists in identifying and labeling the separate different regions of interest of the image. Applications of CCL are in the fields of image understanding, volume visualization, character recognition, geometric modeling and computer vision. The connected component can also be applied to the several computational physics problems such as percolation and various Monte Carlo algorithms for computing the spin models of magnets such as the two-dimensional Ising spin model.

## Part I: Image Histogramming

Given an  $m \times n$  image with  $k$  grey levels, the problem is to compute an array  $H [0,1..k-1]$  such that  $H [i]$  is equal to the number of pixels in the image with grey level  $i$ . The sequence of operations is described below:

- The MATLAB file `image_matrix.m` reads an image (for e.g. `woman.jpg`), converts it into a grayscale image if its not a grayscale image and writes the pixel values in an output file (for e.g. data file `woman.txt`).
- The C program (`imghist.c` using MPI and `imhhistt.c` using threads) reads the pixel values from the input file, computes the global image histogram and writes the histogram matrix in an output file (`hist.dat` for `imghist.c` and `hist1.dat` for `imhhistt.c`).
- The MATLAB program `plothist.m` plots the histogram for the image.

In order to compute the image histogram in parallel, the divide and conquer technique has been adopted. The whole image is divided into sub images and each sub image is given to a processor (when using message passing paradigm) or thread (in case of shared memory paradigm). Each processor or thread then calculates the local histogram and after that the

local histograms are merged together (by adding the corresponding entries of local histograms) to form the global histogram. Partitioning simply divides the original problem into similar problems of smaller dimensions. In this problem, tasks to individual threads or processors have been distributed statically because the time required is dependent on the task size. So if threads or processors are given almost equal amount of work, then it is expected that the idle time of each thread or processor can be minimized without having to go for dynamic task assignment.

The sequential implementation of the problem scans the image from left to right and from top to bottom and if the current pixel has gray level  $k$ , it increments  $H[k]$  by one. The divide and conquer technique partitions the problem into several smaller and similar problems, with each processor or thread computing a portion of the original image. Thus the computations can take place in parallel; however the final merging task is essentially sequential at least for shared memory programming. The local histograms are computed in the same way as the sequential implementation described above. Finally the locally histograms are merged together one by one by adding the corresponding entries of the local histograms. For example, in the  $i$ th merging step (when using shared memory paradigm) we have,

$$global\_hist[k] = global\_hist[k] + local\_hist\_i[k];$$

where  $global\_hist[k]$  is the total number of pixels so far with gray level  $k$  and  $local\_hist\_i[k]$  is the number of pixels with gray level  $k$  in the  $i$ th sub-image.

While using message passing paradigm, MPI\_Reduce is used to merge the local histograms. However, it was observed that MPI is not very suitable for this problem due to the enormous communication that is needed between processors both during the initial data distribution part and final histogram merging part. Much better results are obtained using threads (shared memory paradigm). In MPI implementation, the cost of communication far offsets the gain in computation time using multiple processors. Particularly with large values of  $M$  and  $N$  ( $M$ =number of rows,  $N$ =Number of columns), performance degrades rapidly for MPI.

The following is a sample run using MPI and threads.

#### **Sample run of imghist.c (using MPI)**

```
discover> mpirun -np 4 imghist
Number of processors=4
ROOT processor reading data from input file .....
ROOT processor broadcasting data to other processors
ImagePixel array: number of rows=1485; number of columns=1100; Gray levels 0-255
Merging local histograms to the global histogram
Image Histogram data written to hist.dat file
Total time taken = 2.570091 seconds
```

**Sample run of imghist.c (using threads)**

```
discover> ./imghist woman.txt 4
Reading data from input file woman.txt .....
Number of threads = 4
Thread id=0: step=275
colstart=0: colend=274
Thread id=1: step=275
colstart=275: colend=549
Thread id=2: step=275
colstart=550: colend=824
Thread id=3: step=275
colstart=825: colend=1099
clock start=7400000, clock end = 7520000
CLOCKS_PER_SEC=1000000, elapsed time =0.120000 seconds
```

Image Histogram data written to hist1.dat file

The following observations were made when the programs were run with different images. The number of rows, number of columns and time taken for computing the global histogram are tabulated below.

Shared memory using 4 threads:

Number of rows=M	Number of columns=N	Elapsed time (seconds)
1131	741	0.060000
868	1160	0.070000
1071	1580	0.110000
1485	1100	0.120000
1680	2240	0.270000

MPI using 4 processors:

Number of rows=M	Number of columns=N	Elapsed time (seconds)
1131	741	1.622046
868	1160	2.019544
1071	1580	2.769893
1485	1100	2.955591
1680	2240	5.823097

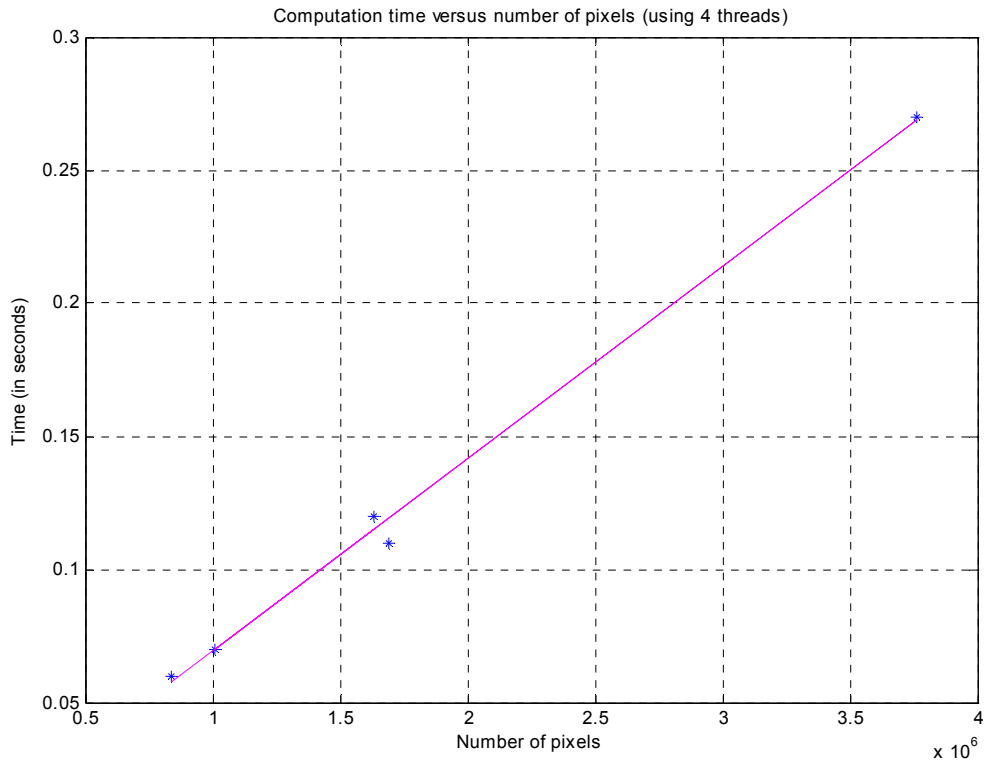


Fig 1: Computation time versus number of pixels for shared memory programming

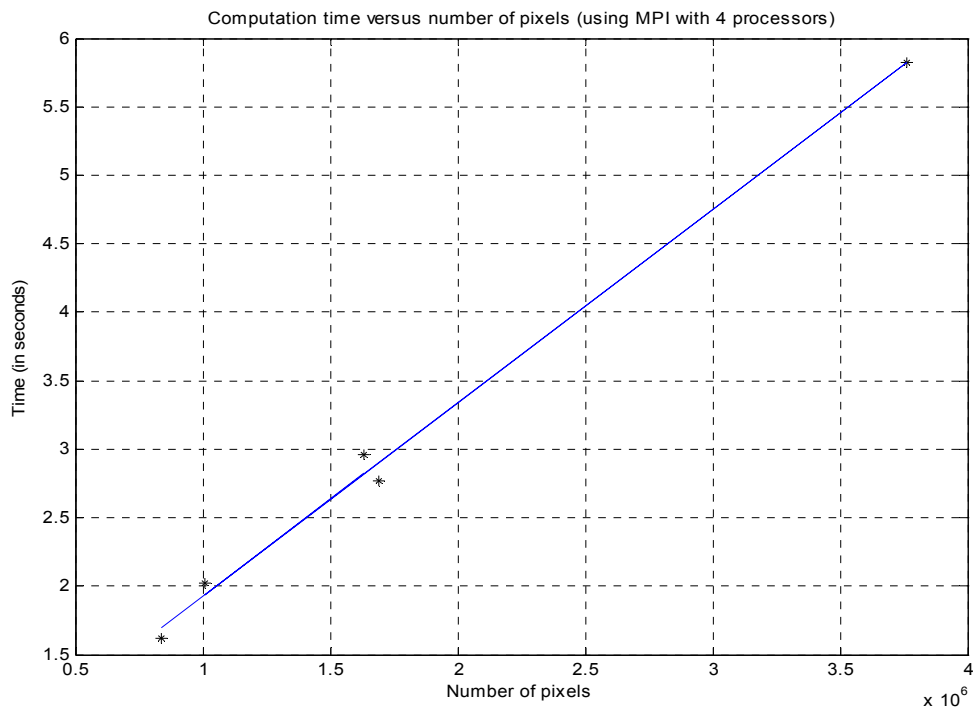


Fig 2: Computation time versus number of pixels when using MPI with 4 processors

From the above graphs, it can be seen that for both shared memory programming and message passing, the computation time increases with the number of pixels almost linearly. However, the time taken when using message passing is much more than when using shared memory. The reason is that as number of pixels increases, the cost of communication far outweighs the cost of computation. In MPI implementation, time is measured from the instant when the root processor starts broadcasting the pixel values to all the processors, so that the individual processors can start working on their portion of data in parallel till the point when all the local histograms have been merged to form the global histogram. Instead of broadcasting all the pixel values to all the processors, the root processor can send to individual processors their portion of data only. However, even then the time taken is much more than what is needed when shared memory paradigm was used.

From the above graphs, the computation time per pixel can be computed as follows:

***Computation time per pixel = slope of the fitted straight line***

Thus computation time per pixel when using shared memory is  $7.24 \times 10^{-8}$  seconds, while computation time per pixel when using message passing is  $1.44 \times 10^{-6}$  seconds.

**Note: In the graphs, computation time refers to the total time i.e. the combined computation time and communication time if any.**

The following observations were made when the programs were run with the image flag.jpg. The times taken for computing the global histogram for different number of threads/processors are tabulated below:

Shared memory (number of rows=1131, number of columns=741):

Number of threads	Elapsed time (seconds)
2	0.050000
3	0.050000
4	0.040000
5	0.060000

Message passing (number of rows=1131, number of columns=741):

Number of processors	Elapsed time (seconds)
2	0.813986
3	1.131303
4	1.300856
5	2.075196

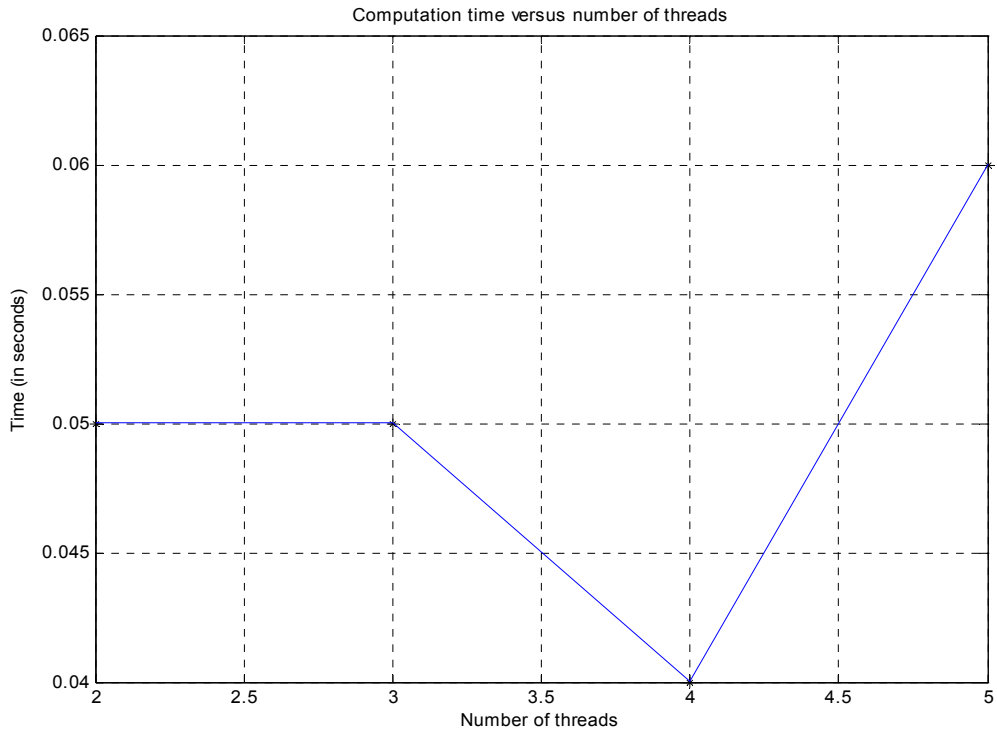


Fig 3: Computation time versus number of threads for shared memory programming

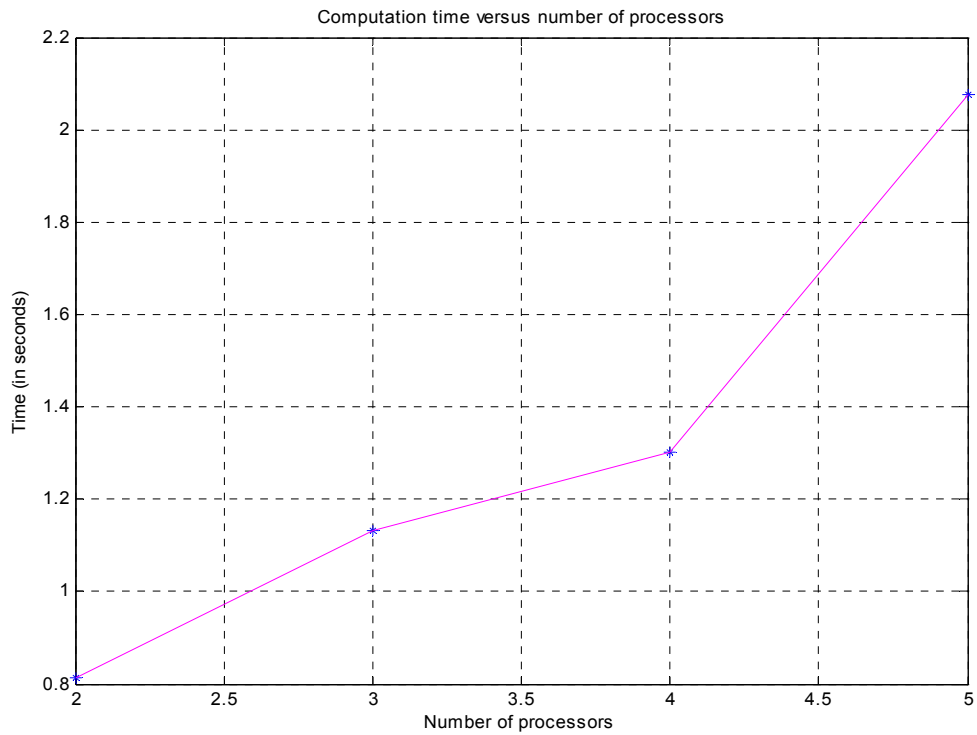


Fig 4: Computation time versus number of processors for message passing

From the above graphs, it can be seen that there is no speedup achieved when using message passing paradigm. The reason as already been explained, is because of the cost of communication dominating over the cost of computation. Thus increasing the number of processors does not seem to have an effect on the time taken; in fact the time taken increases with increasing number of processors. The same observations were made for other images also. When using shared memory, there seems to be some speedup but consistent speedup was not noticed when using more number of threads. The reason may be attributed to the number of CPUs available on the system i.e. 2. Since this is a computationally intensive problem, the worker threads are almost busy throughout and so increasing number of threads do not achieve significant or any speedup.

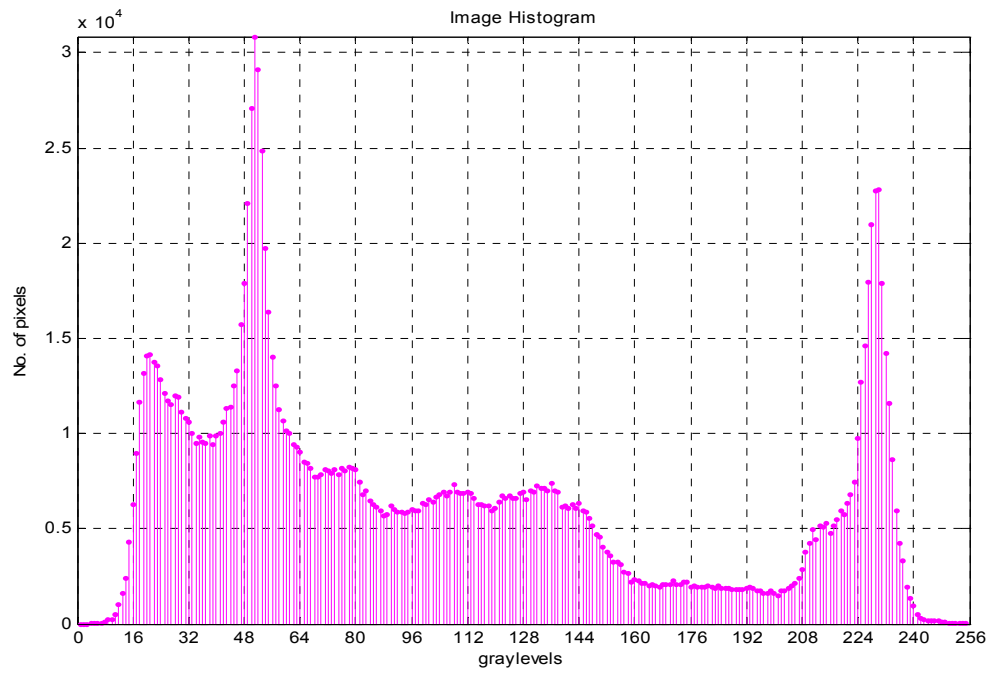
The following figure shows the original RGB image, the next figure shows the transformed grayscale image which has been used for computing the image histogram. The image histogram is shown in the following figure. Then the image histograms obtained by shared memory programming and message passing have been compared and it is seen that the histograms obtained from the two methods are identical, as expected.



*Fig 5: Original RGB Image*

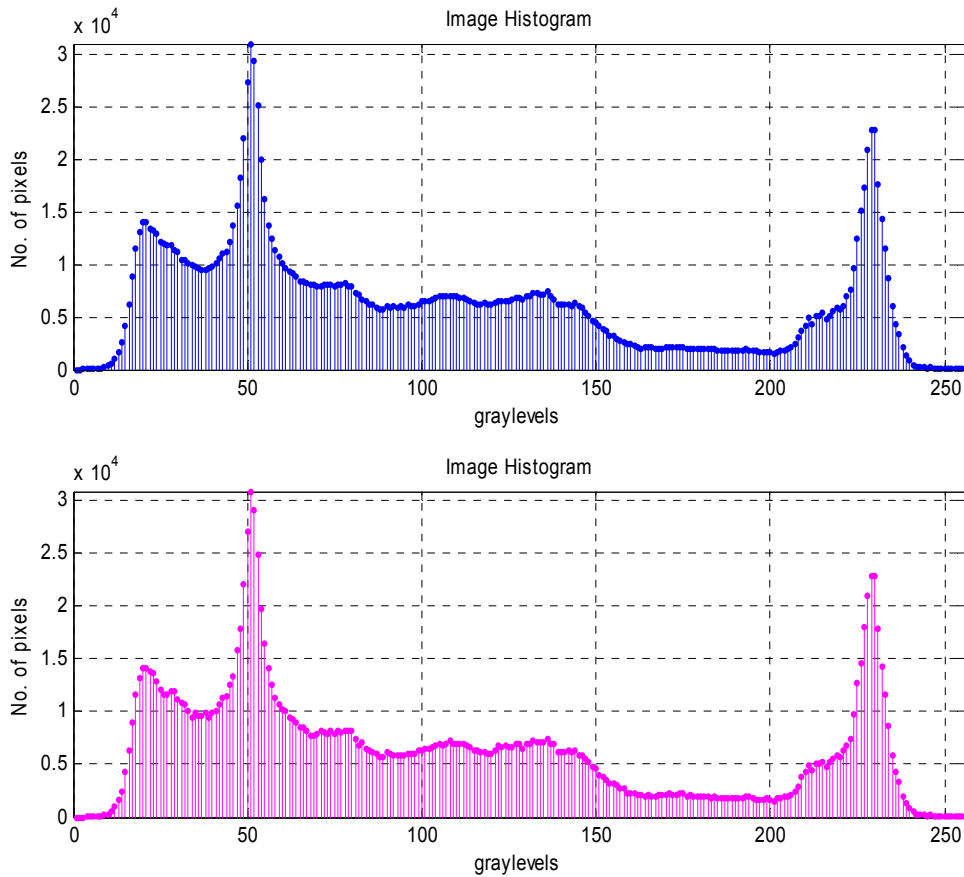


*Fig 6: Grayscale Image*



*Fig 7: Image Histogram of the grayscale Image*





*Fig 8: Image Histogram of the grayscale image by (i) shared memory programming and (ii) message passing*

## Part II: Connected Components Labeling

Detection of connected components between pixels in binary images (which have only two possible gray levels) is a fundamental step in segmentation of image objects and regions. The original algorithm was developed by Rosenfeld and Pfaltz in 1966. A connected component in an image is a set of connected pixels that share a specific property,  $V$ . Two pixels,  $p$  and  $q$ , are connected if there is a path from  $p$  to  $q$  of pixels with property  $V$ . A path is an ordered sequence of pixels such that any two adjacent pixels in the sequence are neighbors. Constructing a connected component consists of growing sets of pixels that are connected and have the same value of a property. This could be accomplished by first finding a pixel with a given property value, then looking at all its neighbors, labeling each that has the same value as being in the same component, and so on.

In the first part, it has been seen that message passing paradigm is not particularly suited for computations like this due to the large overhead associated with communication. So the connected components labeling has been implemented using threads only. The connected components labeling in parallel has been implemented using divide and conquer technique. In this case, after connected components labeling by each thread is over, the local connected component labels have to be merged together. However, there may be pixels which are connected but have been given separate labels as they have been processed by different threads. So during merging, some of the component labels may have to be computed again by finding the equivalence of labels of connected pixels.

The sequence of operations is described below:

- The MATLAB file `image_label.m` reads an image, converts it into a binary image and writes the pixel values in an output file.
- The C program (`imglbl.c`) reads the pixel values from the input file, computes the connected components label locally, then merges the locally computed connected components label by the equivalence resolution and writes the global connected components label in an output file.

**Pixel Connectivity:** A pixel  $p$  at coordinate  $(x, y)$  has four direct neighbors,  $N_4(p)$  and four diagonal neighbors  $N_D(p)$ . To establish connectivity between pixels of 1s in a binary image, three type of connectivity for pixels  $p$  and  $q$  can be considered

- 4 connectivity-connected if  $q$  is in  $N_4(p)$
- 8 connectivity- connected if  $q$  is in the union of  $N_4(p)$  and  $N_D(p)$ .

In this problem, 4 connectivity has been considered. For the purpose of simplifying the problem, binary image instead of grayscale image has been considered. In the case of grayscale images, since the number of grayscales can in the range of 0-255, pixels are said to be connected if their values are within a range.

The whole image is divided into sub images and each sub image is given to a thread. Each thread calculates the local connected components label and after that the connected components labels are merged. In this case also, tasks to individual threads have been distributed statically because the time required is dependent on the task size. So if threads are given almost equal amount of work, then it is expected that the idle time of each thread can be minimized without having to go for dynamic task assignment.

The neighbors of a pixel  $p$  are the pixels to the N, S, W and E of that pixel. The sequential implementation of the problem scans the image from left to right and from top to bottom and if the current pixel is 0, it moves to the next pixel. If  $p$  is 1 and the pixels to the N and W are 0,  $p$  is assigned a new label. If only one of the 2 neighboring pixels (N, W) is not 0 then  $p$  is assigned that label. If both the neighbors have non zero values,  $p$  can be assigned any one of the labels. The divide and conquer technique partitions the problem into several smaller and similar problems. Thus the computations can take place

in parallel; however the final merging task is more complicated than the previous part as has already been explained.

The following is a sample run using threads.

**Sample run of imglbl.c (using threads)**

```
discover> ./imglbl diana.txt 4
```

```
Reading data from input file diana.txt .....
Number of threads = 4
Image is 641 X 1533 pixels
Thread id=1: colstart=383, colend=765
Thread id=0: colstart=0, colend=382
Thread id=3: colstart=1149, colend=1532
Thread id=2: colstart=766, colend=1148
clock start=380000, clock end = 490000
CLOCKS_PER_SEC=1000000, elapsed time =22.010000 seconds
```

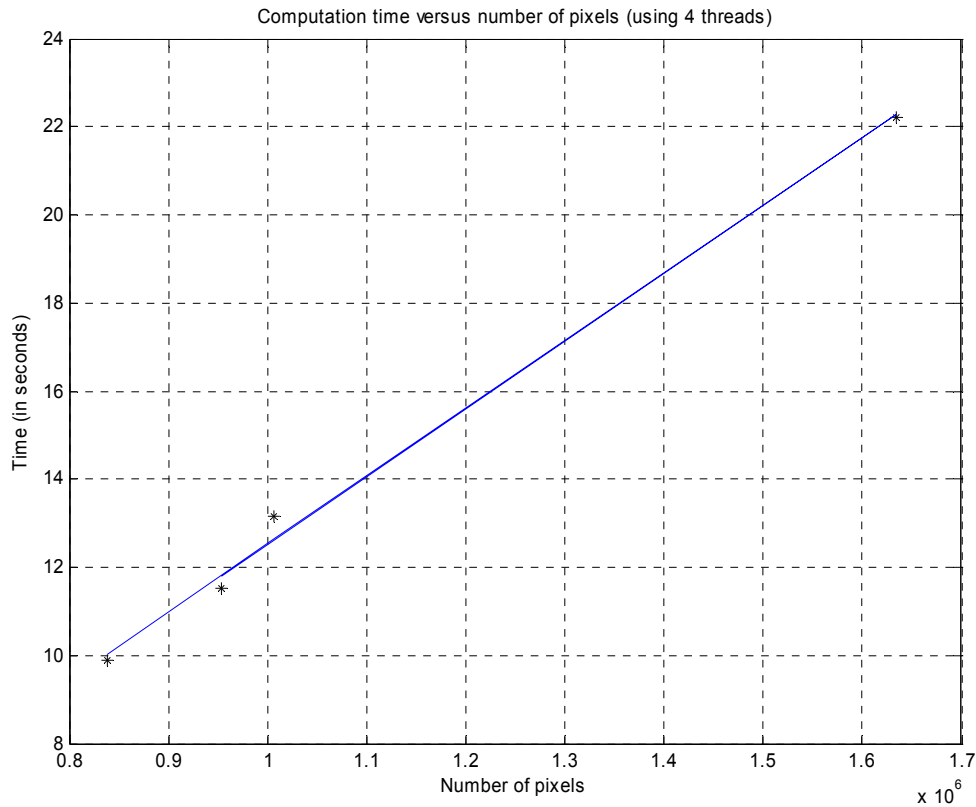
Image component labels written to lbl.txt file

The following observations were made when the programs were run with different images. The number of rows, number of columns and time taken for computing the global histogram are tabulated below.

Shared memory using 4 threads:

Number of rows=M	Number of columns=N	Elapsed time (seconds)
1131	741	9.890000
868	1160	13.150000
1485	1100	22.220000
641	1533	11.510000

**Resolving Equivalence:** Initially, when the connected component labels are assigned locally by each thread, the labels are taken from a shared counter. If a new label is required the associated thread would lock the global counter, increment it, take the incremented value as the new label and finally unlock the global counter. This ensures that all the component labels assigned by the different threads are unique. There are other ways to ensure this but this is easy to implement using a mutex lock. During the merging stage, only the border regions between the sub images processed by different threads need to be examined for finding equivalent labels. If the pixels in these regions have the same pixel value and their labels are different from that of their immediate neighbors to their north (implying that they are new sets of equivalent labels), then the two labels are equivalent, the latter condition preventing duplicate entries of equivalent labels. After all



*Fig 9:* Computation time of connected component labeling versus number of pixels when using 4 threads

the equivalent labels have been resolved, the final global connected components label are decided. This completes the task of connected components labeling of the whole image.

### **Conclusion:**

It was observed in both the image histogramming and connected components labeling problems that significant speedup is not achieved while using threads because of reasons already discussed. On the other hand, the use of MPI resulted in even worse performance due to the large communication overhead. However, it became obvious that shared memory architecture offers the best solution to the problem in hand. Currently many researchers have been working on how to reduce the complexity of equivalence resolution in order to achieve better results. Due to time constraints, a relatively simple but original equivalence resolution (explained in the previous paragraph) has been implemented. A number of images of varying sizes were considered and it was evident that in both the cases, the total number of pixels and elapsed time have an almost perfect linear relationship; the time increasing for increasing number of pixels.

## **Acknowledgements:**

I wish to thank Professor. Manish Parashar for his helpful suggestions and comments that helped me to narrow down on the project goals and create a well-defined problem statement. This also made it feasible for me to valuable insight into the area.

## **References:**

- [Aya] D. Ayala, J. Rodriguez, A. Aguilera, “Connected Component Labeling Based on the EVM Model”.
- [Bau] A. Baumker, W. Dittrich, “Parallel Algorithms for Image Processing: Practical Algorithms with Experiments (Extended Abstract)”.
- [Paa] J. Park, C. G. Looney, H. Chen, “Fast Connected Component Labeling Algorithm Using a Divide and Conquer Technique”.
- [Bad] D. A. Bader, J. JaJa, “Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study (Extended Abstract)”.
- [Aln] H. M. Alnuweiri, V. K. Prasanna, “Parallel Architectures and Algorithms for Image Component Labeling”.
- [Gre] J. Greiner, “A Comparison of Data-Parallel Algorithms for Connected Components”.
- [Wil] Barry Wilkinson and Michael Allen, “Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Prentice Hall, 1999.