PROJECT - HISTOGRAM EQUALIZATION

Rahul Rudradevan
UCI Student ID: 28688203
rrudrade@uci.edu

EECS 220
DSP Architecture
University of California, Irvine

Spring Quarter, 2012

Abstract

The objective of this project is to experience parallel programming and measure the performance improvement of using a Multicore system. In addition to performance improvement, analysis of sharing and passing data between cores and control overhead will be discussed.
In this project, we look at code parallelization, synchronization, shared data structures, and load balancing.

Contents

Chapter 1

Histogram Equalization

1.1 About

This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.

The method is useful in images with backgrounds and foregrounds that are both bright or both dark. In particular, the method can lead to better views of bone structure in x-ray images, and to better detail in photographs that are over or under-exposed. A key advantage of the method is that it is a fairly straightforward technique and an invertible operator. So in theory, if the histogram equalization function is known, then the original histogram can be recovered. The calculation is not computationally intensive. A disadvantage of the method is that it is indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal. [1]

The algorithm contains of 3 significant steps, the first is to calculate the histogram for the array. This counts the number of times a pixel value occurs in the input array. The next step is calculating the cumulative distribution function (cdf), using the histogram created above. And the final step involves calculating the equalized values using cdf calculated above.

Of these 3 steps the first and the last can be done in a fully distributed manner. But the second step involves passing of values between threads therefore contains significant control overhead.

1.2 Program

1.2.1 Thread Data

The following information is passed to each thread:

```
typedef struct str_thdata
{
  int index;
  int size;
  int num_threads;
  int *pixel_value;
  int *cdf;
  int *hist;
  unsigned char *in_image;
  unsigned char *out_image;
} thdata;
```

Here *index* denotes the thread index, using this index the thread identifies the part of the array that it needs to work on. The *size* represents the total size of the input i.e. "rows * columns". *num_threads* gives the total number of threads in the system. *pixel_value* counts the number of times a pixel appears in the input array. *cdf* gives the cumulative distribution of intensities from the histogram created using *pixel_value*. The final equalized value is calculated and stored into *hist*. *in_image* contains the input that is passed into the program array input and *out_image* gives the final values to be written to the output file.

1.2.2 Building the parallel version

Since the steps 1 and 3 are can be run in parallel we can calculate counts of pixel intensities and generate equalization values in parallel.

- Step 1: We divide the input array by the number of threads, each thread updates the count for its section of the array. Here there is a chance of a race condition happening. We therefore introduce a mutex associated with each pixel value from 0 to 255. Therefore a thread can only update a pixel value if there is no other thread currently in the process of updating it.
- Step 2: This step works in a propagate manner i.e. the value from one step needs to be forwarded to the next. We therefore make use of a pipe that forwards values from one thread to the next. A thread waits till it gets the value from the previous thread for it to begin processing.
- Step 3: Calculating the equalization values requires no synchronization at all. All threads work independently on different parts of the *hist* array, thus helping us realize total parallelism.

Now since these steps need to be executed one after the other, we need to make sure that only when all the threads are done with a particular step we start with the next step. For this we make use of **barriers** that enforces synchronization at every step in the program.

Code

The code can be run using the following syntax:

```
Syntax: ./hist.o [number_of_threads] [filename]
```

1.3 Running hist.o

1.3.1 Measurements on lido.eecs.uci.edu

All the tests below were done on the input file **IrvinePark.tiff**. Here the Unix utility *time* was used to measure the time taken by the program:

```
rahul@solaris:~/project$ time ./hist.o 32 IrvinePark.tiff
Number of threads: 32

real    0m0.382s
user    0m3.917s
sys     0m0.093s
rahul@solaris:~/project$
```

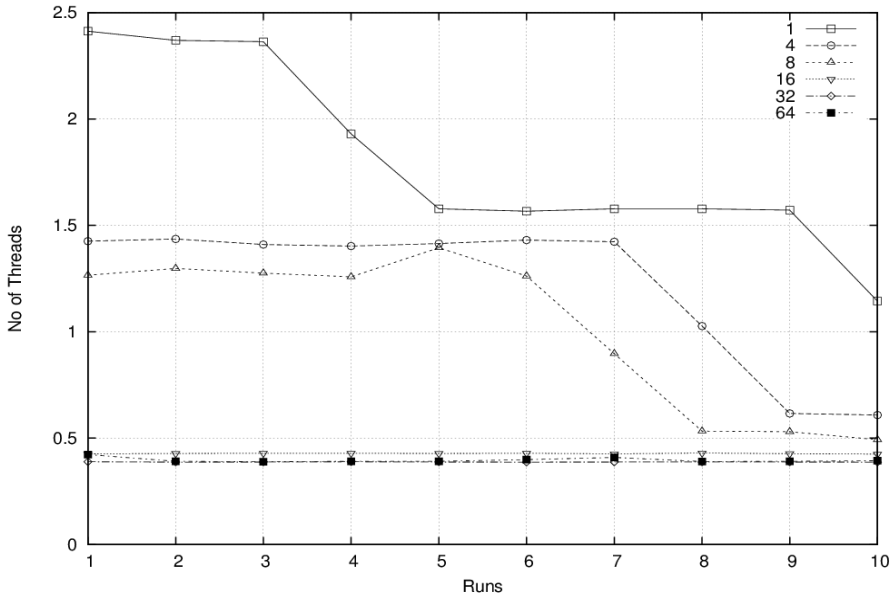
We sample the times for 10 runs of the program using image size of 9980928. The following table lists the measurements on lido.eecs.uci.edu:

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	2.413	2.370	2.364	1.930	1.578	1.567	1.578	1.578	1.572	1.144
4	1.426	1.436	1.410	1.403	1.415	1.431	1.423	1.027	0.616	0.609
8	1.266	1.298	1.276	1.258	1.395	1.262	0.897	0.533	0.530	0.493
16	0.426	0.428	0.429	0.429	0.428	0.429	0.426	0.430	0.427	0.425
32	0.389	0.387	0.388	0.389	0.388	0.387	0.388	0.388	0.388	0.387
64	0.423	0.391	0.389	0.391	0.391	0.399	0.409	0.390	0.391	0.394

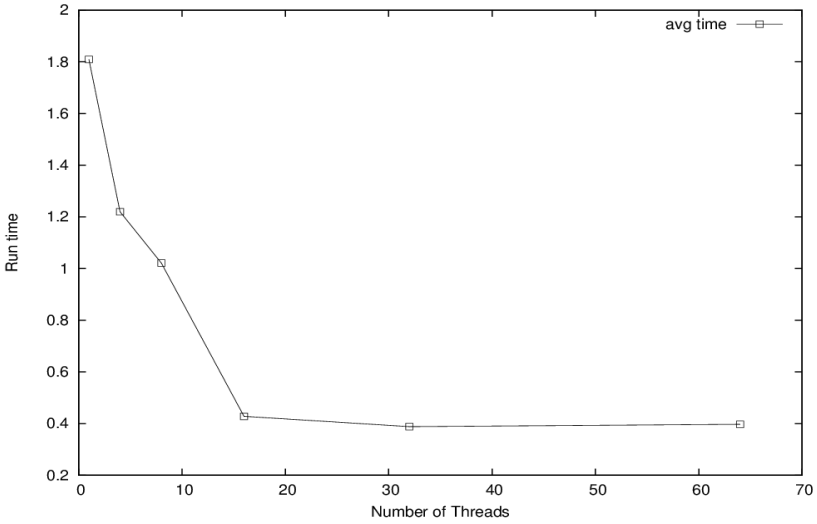
The average time and the data processed by each thread is given below:

Threads	Average	Data for each thread
1	1.809	9980928
4	1.220	2495232
8	1.021	1247616
16	0.428	623808
32	0.388	311904
64	0.397	155952

These statistics can be better understood using the following graphs. First we look at the graph corresponding to the first table. One trend we can identify right away is that with each successive run the runtime decreases. The reason for this is because the first time the program is run, the cache is cold and therefore does not contain the input file resulting in a lot of misses. With every subsequent run the cache hit rate increases thus increasing the overall performance of the program.



From the graph of average runtimes we can see that as we increase the number of threads, the time taken by the program keeps decreasing and hits a minimum at 0.338s. Increasing the number of threads does not improve performance as this time is governed by Amdahl's law i.e. the sequential code (computing *cdf*) in the program serves as the bottleneck that limits the amount of parallelism that can be extracted out of the program. Therefore best performance is achieved using 16 threads or more (32 performs the best).



1.4 Conclusion

In order to extract maximum performance from lido.eecs.uci.edu we have set the number of threads to 16 or more. The program was tested for 3 other images too and the same trend was observed.

References

[1] , http://en.wikipedia.org/wiki/Histogram_equalization

[2] http://tinyurl.com/6m5mmw9

[3] https://eee.uci.edu/12s/18418

[4] https://computing.llnl.gov/tutorials/pthreads/

[5] http://softpixel.com/~cwright/programming/threads/threads.c.php2

[6] http://pages.cs.wisc.edu/~travitch/pthreads_primer.html