Project 2: Shared Memory Multithreading Programing

Daniel Woldegiorgis

October 22, 2019

1 "Blur" Filter

In order to blur each pixel, we use the following formula to obtain a weighted average of itself and its neighbors.

$$output(x,y) = \frac{\sum_{i=-(r-1)}^{r-1}\sum_{j=-(r-1)}^{r-1} input(x+i,y+i)(r-|i|)(r-|j|)}{\sum_{i=-(r-1)}^{r-1}\sum_{j=-(r-1)}^{r-1}(r-|i|)(r-|j|)}$$

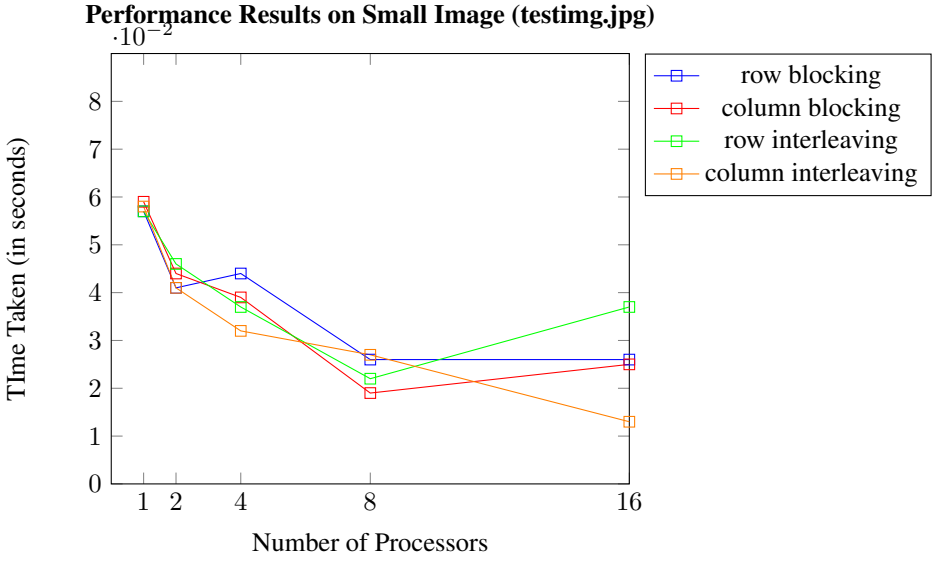
As suggested, I precomputed the weights and final division factor to reduce the amount of work done while looping through individual pixels of the image.

The results obtained through the row blocking, column blocking, row interleaving and column interleaving methods using 1, 2, 4, 5, and 16 processors is tabulated below and then graphed. This is divided into three cases based on the size of the input image that was specified.

Small Image Time Test Results (in seconds)				
Num. Threads	Row Blocking	Col Blocking	Row Interleaving	Col Interleaving
1	0.057	0.059	0.057	0.058
2	0.041	0.044	0.046	0.041
4	0.044	0.039	0.037	0.032
8	0.026	0.019	0.022	0.027
16	0.026	0.025	0.037	0.013
Medium Image Time Test Results (in seconds)				
Num. Threads	Row Blocking	Col Blocking	Row Interleaving	Col Interleaving
1	23.745	22.352	23.650	22.476
2	11.948	11.249	11.704	11.150
4	6.043	5.704	5.951	5.595
8	3.184	2.915	2.938	2.830
16	2.624	1.749	2.101	1.573

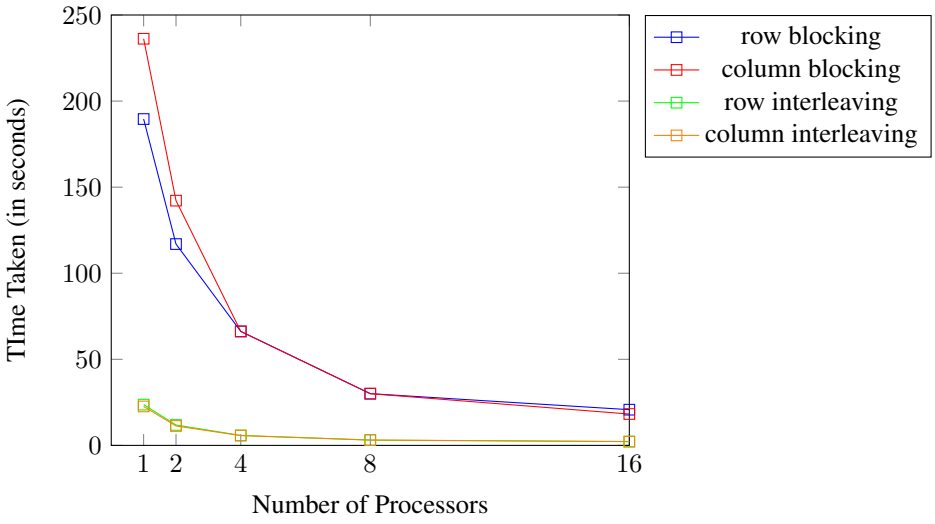
Large Image Time Test Results (in seconds)				
Num. Threads	Row Blocking	Col Blocking	Row Interleaving	Col Interleaving
1	930.619	590.135	931.420	879.747
2	317.369	367.725	317.369	438.944
4	184.908	148.146	234.528	219.438
8	93.275	30.046	116.704	111.415
16	50.698	93.099	94.508	89.185

Performance Results on Small Image (testimg.jpg)



Small Image Interpretation: We can see that given a relatively small image, the performance as we use more and more processors is generally getting better for all four parallelizing mechanisms. However, the comparative performance of each of the four threads is very disarrayed and fluctuating as we continue to increase the number of processors. It is therefore hard to definitively claim whether one of them is better than the rest for an image of this size.

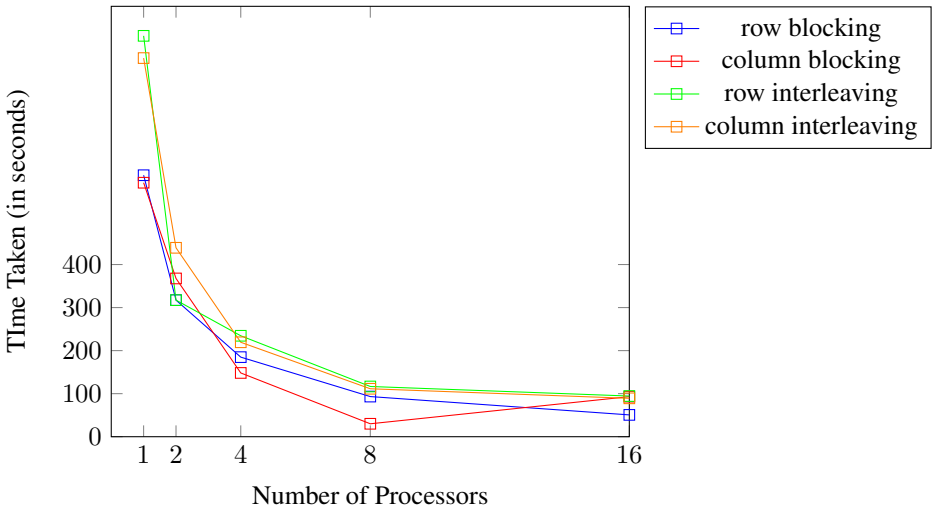
Performance Results on Medium Image (fall.jpg)



Medium Image Interpretation: Similar to the small image results above, the time taken to finish a task continues to

diminish as we continue to use more and more processors. On the other hand, however, given an image of medium size, row and column interleaving outperform row and column blocking. This could perhaps be explained by the fact that interleaving naturally lends itself to load balancing because of the cyclic nature of the assignment of data to specific threads.

Performance Results on Large Image (gorge.jpg)



Large Image Interpretation: We can see that given a relatively large image, the performance as we use more and more processors is generally getting better for all four parallelizing mechanisms. Eventhough, the performance of each of the four cases is a little disarrayed in a manner that's similar to the small image case, we can note that row blocking is consistently low compared to the other parallelizing mechanisms. Therefore, part2 is implemented using the row blocking parallelizing mechanism.

2 Histogram Analysis

Here we will be accumulating four different histogram values: a red histogram (0-255), green histogram(0-255), blue histogram(0-255), and a sum histogram(R+G+B=0-765) for each pixel in a given input image.

2.1 Part A

For this section, I allocated the four histograms as a shared memory object protected by locks in 1 of 3 ways: locks for each histogram bucket, locks for groups of 32 histogram buckets, or locks on entire histograms.

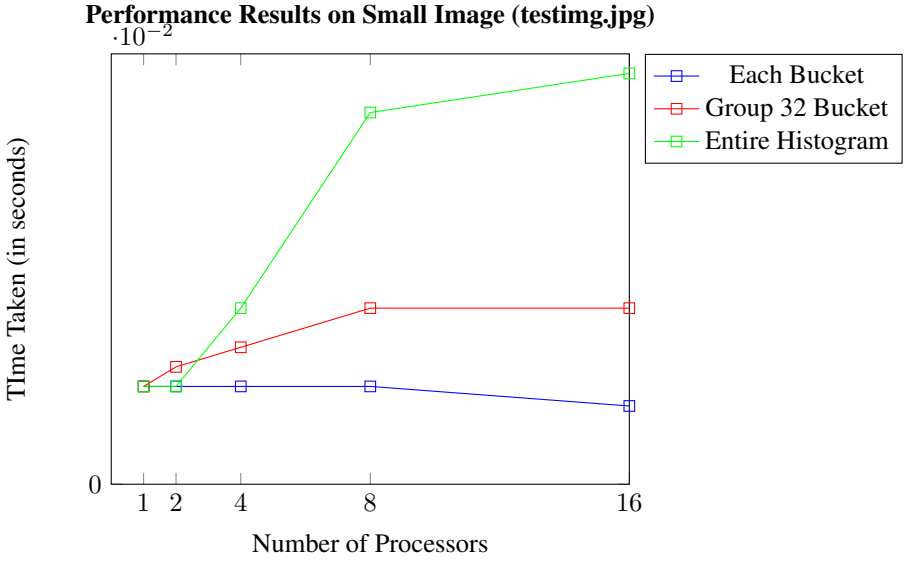
Below are the tabular results obtained when running the row blocking version of the problem on 1, 2, 4, 8, and 16 processors using the three locking mechanisms described above. These are followed by line graphs to ease comparisons among them.

Small Image Time Test Results using Row Blocking (in seconds)			
Num. Threads	Each Bucket	Group 32 Buckets	Entire Histogram
1	0.005	0.005	0.005
2	0.005	0.006	0.005
4	0.005	0.007	0.009
8	0.005	0.009	0.019
16	0.004	0.009	0.021

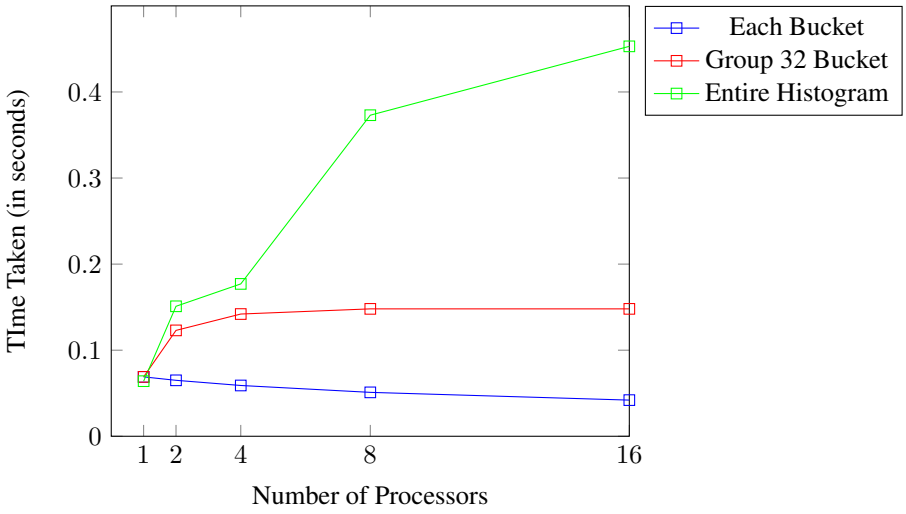
Medium Image Time Test Results using Row Blocking (in seconds)			
Num. Threads	Each Bucket	Group 32 Buckets	Entire Histogram
1	0.069	0.069	0.064
2	0.065	0.123	0.151
4	0.059	0.142	0.177
8	0.051	0.148	0.373
16	0.042	0.148	0.453

Large Image Time Test Results using Row Blocking (in seconds)			
Num. Threads	Each Bucket	Group 32 Buckets	Entire Histogram
1	0.417	0.420	0.386
2	0.330	0.528	1.134
4	0.272	0.668	0.930
8	0.230	0.712	2.411
16	0.217	0.690	2.762

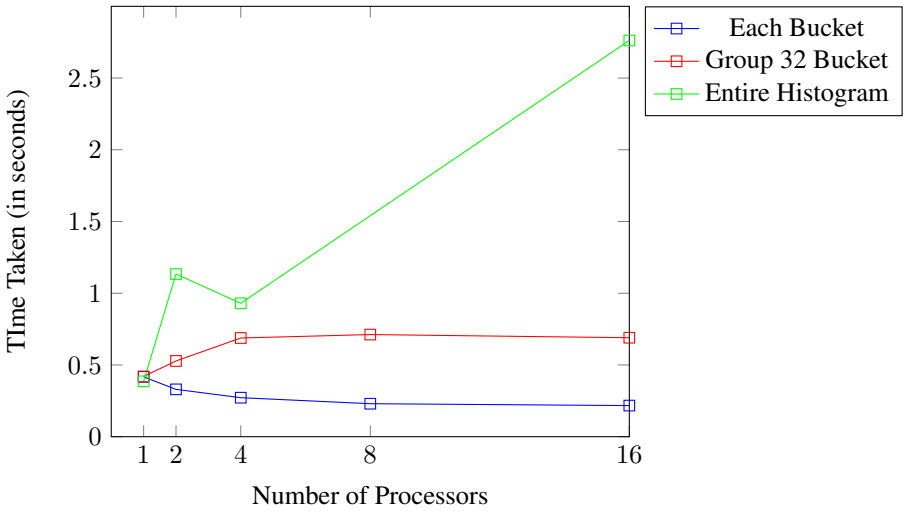
Performance Results on Small Image (testimg.jpg)



Performance Results on Medium Image (fall.jpg)



Performance Results on Large Image (gorge.jpg)



Interpretation: We can see that given either a small, medium or large image, locking each histogram bucket outperforms blocking by locking groups of 32 buckets which in turn outperforms blocking entire histograms.

One thing to note here is that a specific lock is acquired when one of the shared rHist,gHist,bHist or sHist array entries are about to be updated (incremented).

Therefore, one of the clear disadvantages of locking entire histograms when doing an update to just one pixel entry of that histogram is that other threads will have to wait for this update to happen before they can obtain the lock and update it themselves. Since this serialization happens frequently, it affects the performance of this locking mechanism. Similarly, even though it is better to lock particular sections of the array that are about to be updated, leaving the rest of the array unlocked incase other threads want to update them, it is still going to be locking entries that are not going to be updated and this affects its performance.

Lastly, locking only the histogram bucket that is about to be updated gives high performance because the likelihood of threads waiting for other threads to release the lock on a specific entry among 256 is less.

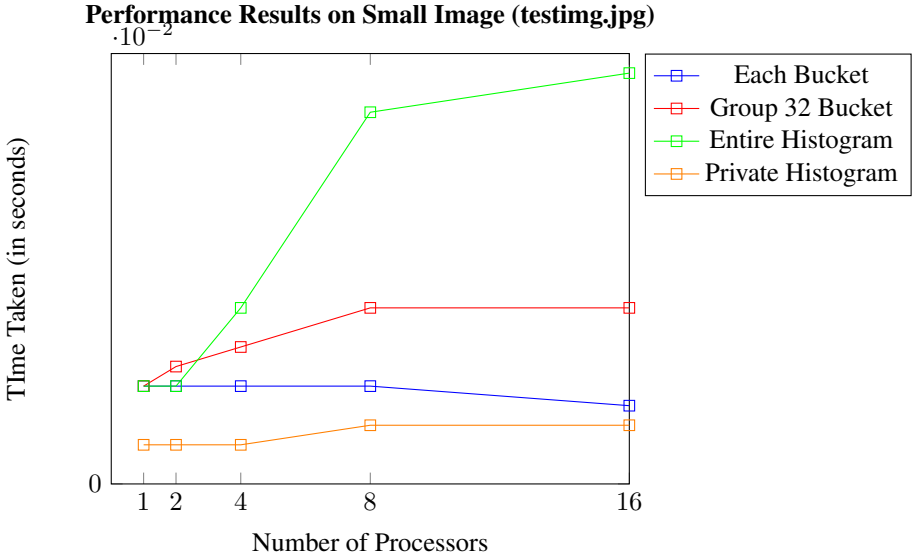
2.2 Part B

For this section I allocated private histograms for each processor with a final reduction to one set of histograms at the end. There is no locking in this section.

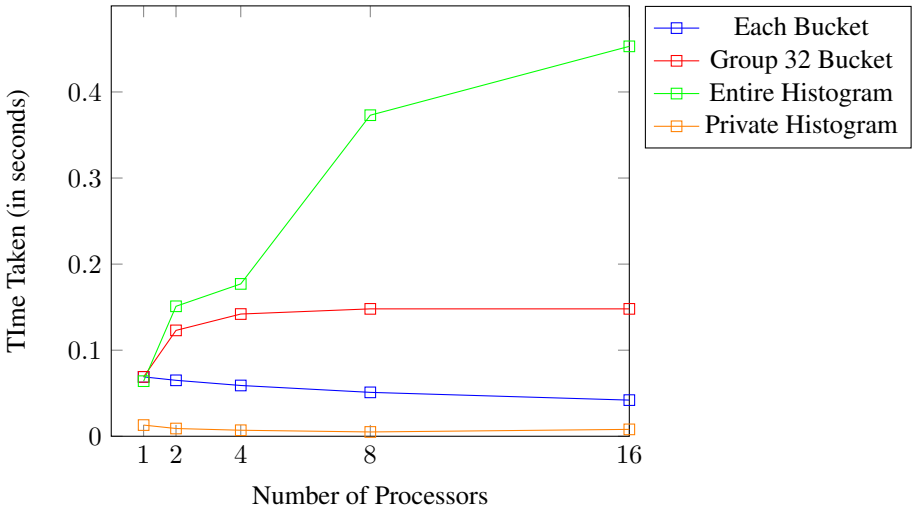
The tabular results obtained when running the row blocking parallelization on 1 ,2, 3, 8, and 16 processors is given below. These are then followed by line graphs that plot the results from part 2A and add an extra orange line for the performance of 2B to ease in comparing their performances.

Time Test Results using Row Blocking (in seconds)			
Num. Threads	Small Image	Medium Image	Large Image
1	0.002	0.013	0.073
2	0.002	0.009	0.042
4	0.002	0.007	0.024
8	0.003	0.005	0.022
16	0.003	0.008	0.015

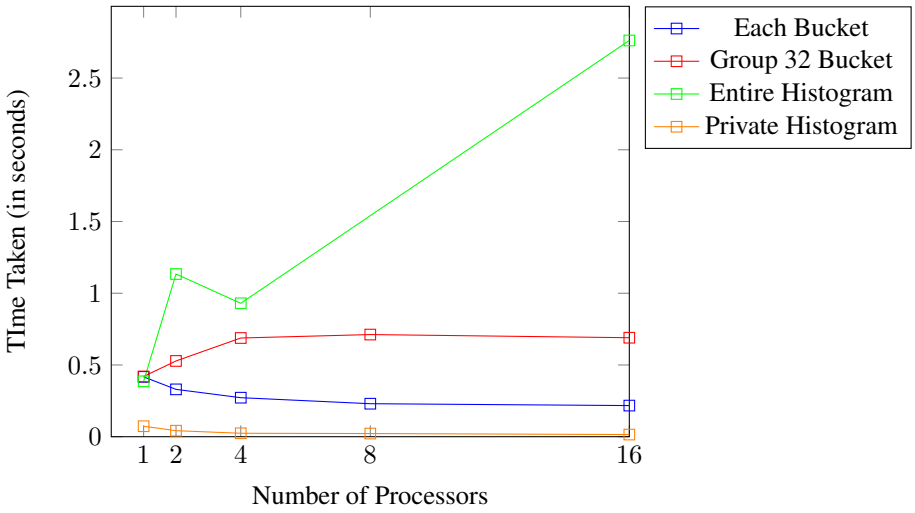
Performance Results on Small Image (testimg.jpg)



Performance Results on Medium Image (fall.jpg)



Performance Results on Large Image (gorge.jpg)



We can see that the orange line for part 2B is outperforming all possible lock combinations of part 2A, so this helps to explain the serializing nature of the use of locks and its effects on performance. So, if there is a possibility of doing the computation accurately without the use of locks, this suggests that such an approach would be better than using locks as the main synchronization primitive.

2.3 Part C

Here, we notice the effects of race conditions that exist in the program that become apparent when the locks are removed from Part A.

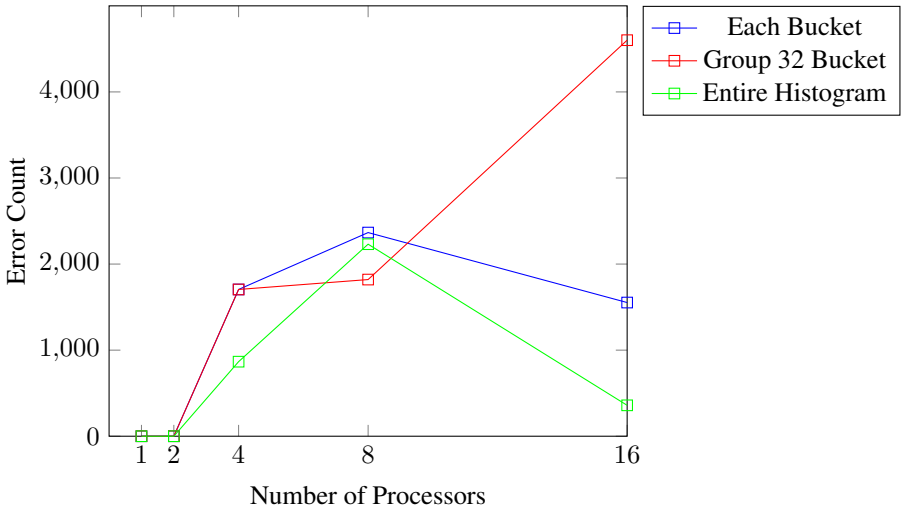
The valdiff errors that were obtained when running the row blocking parallelization in cs338-part2c.c on 1 ,2, 3, 8, and 16 processors is given below. These are then followed by line graphs that plot these errors against the number of active threads during it's execution.

Small Image Valdiff Total Error Results using Row Blocking			
Num. Threads	Each Bucket	Group 32 Buckets	Entire Histogram
1	0	0	0
2	0	0	0
4	1705	1705	866
8	2366	1820	2231
16	1553	4601	359

Medium Image Valdiff Results using Row Blocking			
Num. Threads	Each Bucket	Group 32 Buckets	Entire Histogram
1	0	0	0
2	12888	13237	18386
4	117796	116867	99437
8	138995	180442	141142
16	313299	253798	215180

Large Image Valdiff Results using Row Blocking			
Num. Threads	Each Bucket	Group 32 Buckets	Entire Histogram
1	0	0	0
2	132730	136615	147502
4	490783	480662	385622
8	1527279	1152687	1469185
16	2811987	2587319	2748313

Valdiff Error Results on Small Image (testimg.jpg)

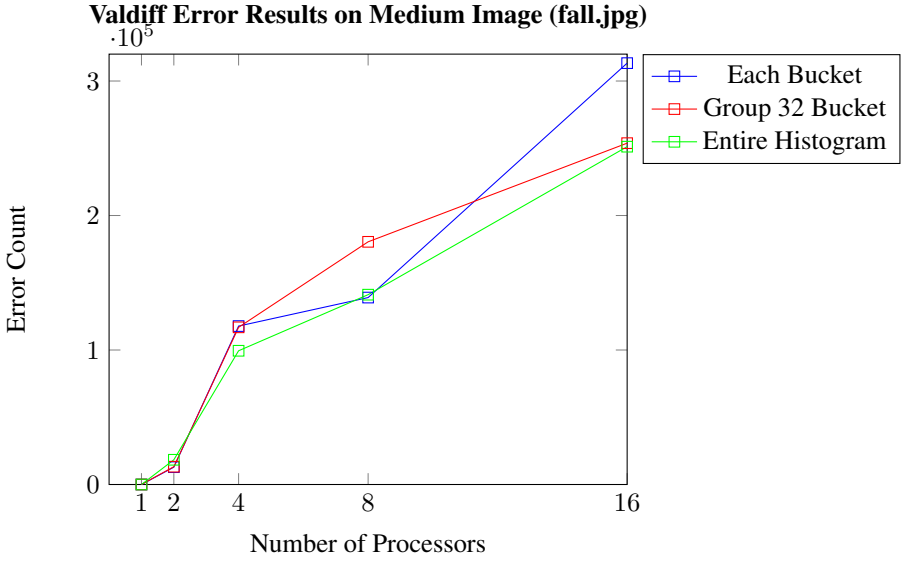


Given a smaller image we can see that different runs can result in vastly different number of errors because of the sporadic nature of the outcomes of race conditions. Generally though, the likelihood of not getting errors seems to

decrease as the number of processors increases as seen across multiple different runs during testing.

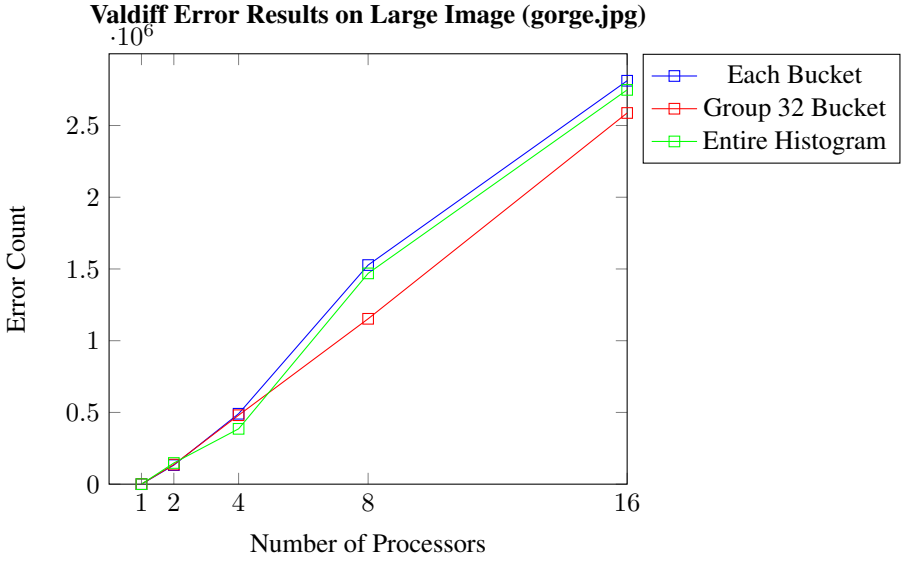
Note here that in the cases where race conditions have had effect, the recorded rHist will always be less than its intended value.

Valdiff Error Results on Medium Image (fall.jpg)



We can see that as the image size continues to increase, there is more consistency in the upward spike of the number of errors as the number of processors increases. This can be explained by the fact that there are now more competitors trying to update a given entry in the histogram.

Valdiff Error Results on Large Image (gorge.jpg)



We can now see that there is an even more consistent upward trajectory in the number of errors as the number of processors continues to increase. This can be the result of increased competition as explained above.

Therefore, we can generally say that as the image size grows, the number of errors that we expect to have also continues to grow because there is increased opportunity for errors to occur as the running time continues to get delayed.

2.4 Steps taken to Parallelize.

I first started out with getting a serial version of part1 using the row blocking, column blocking, row interleaving and column interleaving mechanisms. After this I started working on the their parallelized versions using pthread. At first I had implemented it such that the last thread would have to work on some of the stray rows at the end that are left over because of integer truncation at the beginning of the doubly nested for loops that loop over the image using indices i and j. Then, I adjusted it to use the ceil function so that all the rows share approximately equal number of rows (± 1).

In the beginning, I was simply doing all of the computation of both the weights and division factor inside of the nested as it was iterating through the image using the doubly nested for loop. Later, I pre-calculated both the weights and division factor before this loop and reduced the work done inside the loop body. In order to enable this, I made sure that pixels within radius pixels of the edges of the image were corrected for (subtracted) during the accumulation of the numerator and denominator.

In addition, since obtaining the numerator for the red, green and blue pixels are independent operations that do not interfere with each other, I chose to have execute them during a single iteration instead of doing multiple calculations.

For part 2A, I tried to reduce the amount of instructions that are executed inside of mutex locks and made sure that I am locking the histograms only when a thread is about to update the shared histogram.

For part 2B, I allocated private histograms using a histogram struct and had each thread make local updates to each respective rHist, gHist, bHist or sHist entry and then finally return the struct back to the main thread where a final reduction was performed as each one was being joined back into the main thread. This results were then written out to an output file. There were no locks used for this part.

For part 2C, I removed the USE_MUTEX_LOCKS directive and then performed analysis on the valdiff outputs as compared to the expected correct output.
