20sdc2

Part One: Image Blurring

I've created and tested 4 separate, parallelized methodologies for blurring an input image. These methods are based on row-major or column-major processing, and block or interleaved processing. In theory, row-major block processing should be the fastest method, as the sequential data accesses should have improved locality. For similar reasons, we would expect column-major interleaved processing to be the slowest method due in part to higher data fragmentation.

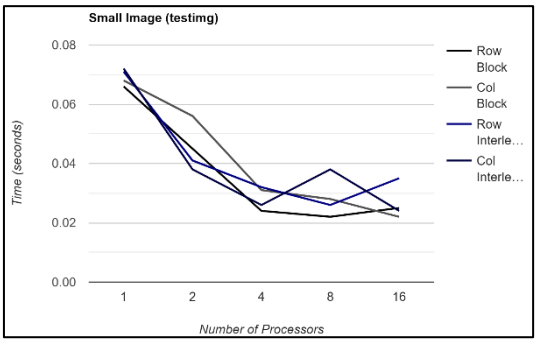

Figure 1 – Small Image Test

In practice, however, we have observed that there is no statistically significant difference in processing times between the four methodologies (Figures 1-3, Appendix). There are two potential explanations for the observed trends. First, we must acknowledge that the caches we're operating on are large and can easily store all the data for our executing program and the input image we're processing. Due to our ability to cache all our work, we would expect little to no impact from our methodologically differential localities. Another explanation is that the GNU compiler is very good, and when we compile with the highest level of optimization, the compiler will optimize-out poor array access paradigms in favor of more efficient methods.

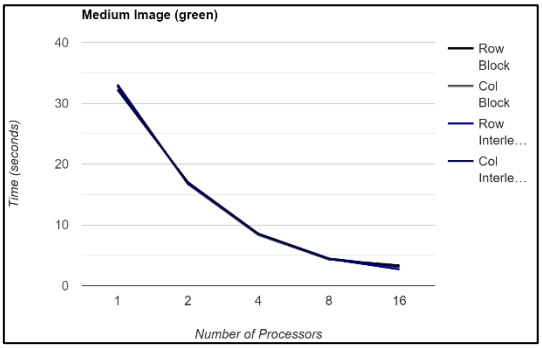

Figure 2 – Medium Image Test

I also expected that when using an increased number of processors, the execution time of our program would decrease. This indeed held true (Figures 1-3 below, Appendix), but the time reduction was not perfectly inversely linear with the number of processors. This is clearly due at least in part to the overhead costs associated with using more pthreads. Another reason may be that our program didn't have exclusive access to the number of processors it was logically using (due to other users running tests, background processes, etc.), and thus may have needed to share processors to appear to be running with true parallelization.

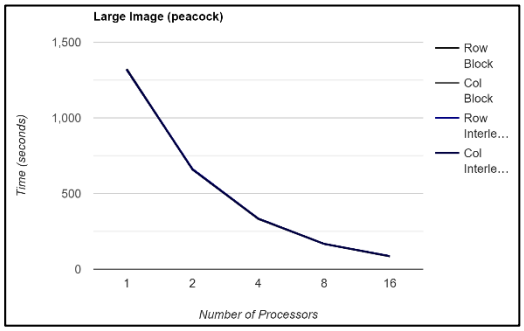

Figure 1 – Large Image Test

It is also observed that the overhead costs of pthreads must be relatively fixed, as larger images enjoy more stable benefits from the parallelized work.

Part Two: Histogram Analysis

In this section, I've successfully counted the occurrence of different pixel values in an input image using 5 different methodologies. The 3 initial methods are to create histogram-wide locks that each process must acquire before being able to increment counts, the next is creating "bucket" locks[1] where sets of pixel values share locks, and finally is individual locks, where each pixel value has a unique lock that must be acquired before incrementing the corresponding histogram. Figure 4 plots these results with respect to time and number of processors on a large image (peacock.jpg). As expected, with one processor there is no difference in results between the three initial methodologies, because there are no lock conflicts when only one processor is ever trying to access a given piece of shared data. Further, as the number of processors increases, we see that the programs which ran with more unique locks performed better than the one large histogram-wide lock. This is an obvious result, as the one lock method will lead to increasing numbers of access conflicts as all processors are trying to acquire the same lock simultaneously, while the "many locks" approach minimizes lock conflicts. The more locks we have, however, the more overhead we have, so these results become increasingly stable and true as both the size of the image and the number of processors grows.

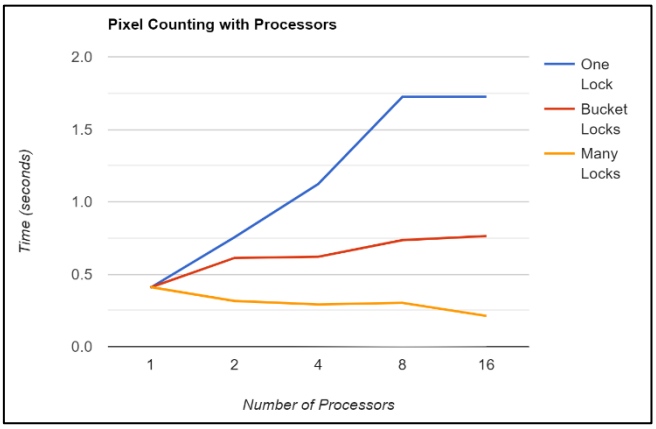

Figure 4 – Three Lock Methods

Next, I implemented "local" histograms, and then sequentially reduced the local results of each processor to one output. This eliminates the need for locks, as well as other "unseen" overhead factors (e.g. cache coherence and consistency) associated with using shared memory. These results are shown on Figure 5, where we see that the local histogram approach far outpaced the many locks approach, which was itself the best method from the three mentioned above. Again, this result is reasonable as we avoid a lot of unnecessary overhead from locks and shared memory. The local histogram approach experienced little to no speedup with an increased number of processors. This could be due to several different factors. First, and most likely, is that the limited agent of the program is likely the sequential reduce, which will take a relatively fixed time regardless of the number of processors we use. Next, any time we save from parallelizing

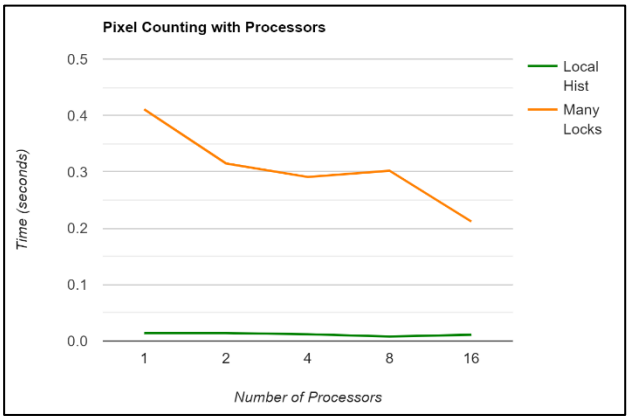

Figure 5 – Local Histogram Method

[1] My buckets are creating by using the mod function, instead of using blocked ranges, as I expect the decreased locality of pixel colors corresponding to the same lock to result in less dead-time waiting for a lock.

the work could be offset by the increased overhead of using more threads. Finally, with such small runtimes, using wall-clock time as our reference may just be too unreliable and noisy, even on a large image such as peacock.

The final addition to this section was to remove any safety mechanisms (e.g. locks or private histograms) from the program, and have each thread try accessing shared memory simultaneously. As expected, this leads to some pixel miscounting, as multiple threads could simultaneously get the current count, increment it, and return it, which would erase multiple counts (e.g. if 8 processors all took the value 4, incremented it, and all returned the value 5, then we would be missing 7 increments). Further, the general trend was for the number of miscounts to increase as the number of processors increased. Again, this is expected in a typical image as there are higher likelihoods for collisions when more threads are accessing the same shared data. Raw data outputs for miscounted pixels are included in the appendix, while Figure 6 shows the percent of missed pixels when counting. These results were derived from two images, peacock and monochrome-test. Monochrome is an image where every pixel has the exact same value, while peacock is a more "typical" photo. As such, the expectation was that there would be far more resource access conflict in the monochrome photo than in the typical photo. We ultimately see this as true in the figure.

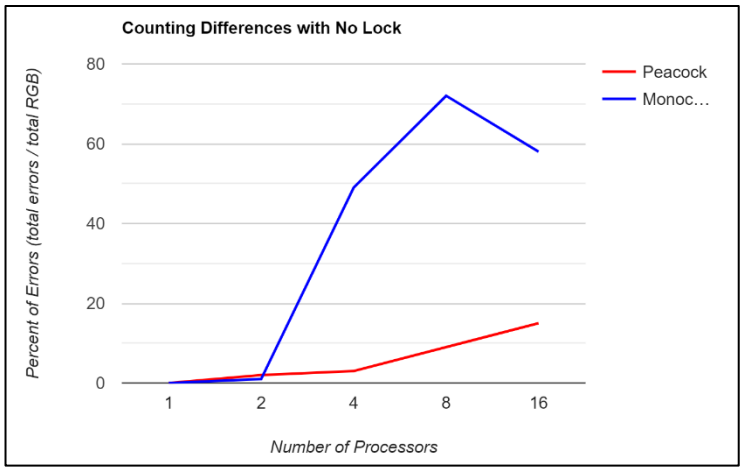

Figure 6 – No lock error rates

Finally, we look at the time of all the different methodologies in Figure 7. As expected, we see that the no lock method is faster than the three locking methods. Unexpectedly, however, is that the local histogram method is even faster than the no lock method, without being inaccurate like the no lock method is. This may be due to the fact that the local histogram method doesn't have to deal with "invisible" overheads related to maintaining cache consistency and coherence, which the no lock method will have to do by virtue of accessing shared memory, even if it does so unsafely (i.e. without locks).

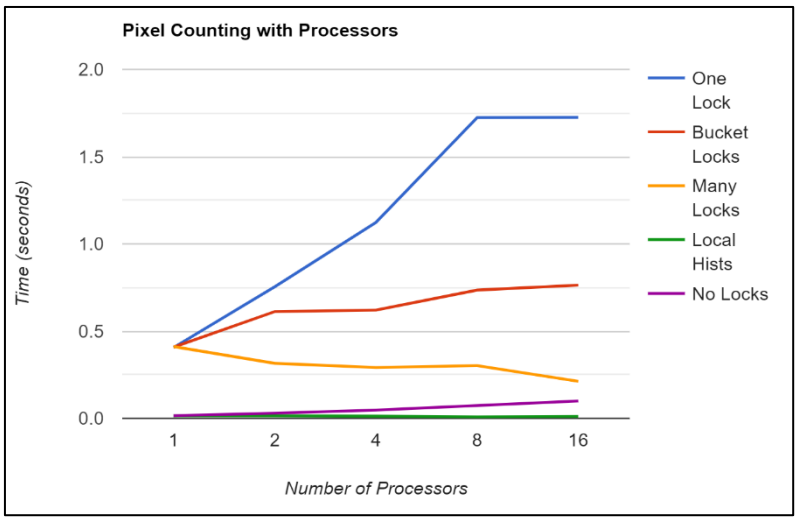

Figure 7 – Comparing all 5 methods

Appendix

Image Size								
Small Image			Medium Image			Large Image		
Processors	Methodology	Time (seconds)	Processors	Methodology	Time (seconds)	Processors	Methodology	Time (seconds)
1	Column, Block	0.068	1	Column, Block	32.834	1	Column, Block	1316.842
1	Column, Interleaved	0.072	1	Column, Interleaved	32.229	1	Column, Interleaved	1317.536
1	Row, Block	0.066	1	Row, Block	33.085	1	Row, Block	1323.655
1	Row, Interleaved	0.071	1	Row, Interleaved	32.787	1	Row, Interleaved	1319.16
2	Column, Block	0.056	2	Column, Block	16.682	2	Column, Block	659.756
2	Column, Interleaved	0.038	2	Column, Interleaved	17.012	2	Column, Interleaved	657.934
2	Row, Block	0.045	2	Row, Block	16.688	2	Row, Block	658.163
2	Row, Interleaved	0.041	2	Row, Interleaved	16.844	2	Row, Interleaved	662.83
4	Column, Block	0.031	4	Column, Block	8.318	4	Column, Block	331.388
4	Column, Interleaved	0.026	4	Column, Interleaved	8.529	4	Column, Interleaved	334.317
4	Row, Block	0.024	4	Row, Block	8.343	4	Row, Block	331.693
4	Row, Interleaved	0.032	4	Row, Interleaved	8.485	4	Row, Interleaved	334.785
8	Column, Block	0.028	8	Column, Block	4.338	8	Column, Block	167.286
8	Column, Interleaved	0.038	8	Column, Interleaved	4.451	8	Column, Interleaved	167.546
8	Row, Block	0.022	8	Row, Block	4.295	8	Row, Block	166.468
8	Row, Interleaved	0.026	8	Row, Interleaved	4.394	8	Row, Interleaved	167.258
16	Column, Block	0.022	16	Column, Block	3.045	16	Column, Block	83.831
16	Column, Interleaved	0.024	16	Column, Interleaved	3.137	16	Column, Interleaved	85.961
16	Row, Block	0.025	16	Row, Block	3.311	16	Row, Block	84.029
16	Row, Interleaved	0.035	16	Row, Interleaved	2.666	16	Row, Interleaved	86.641

Appendix Table – Part One Runtimes

Type	Num Procs	Time (s)
One lock	1	0.406
One lock	2	0.754
One lock	4	1.123
One lock	8	1.724
One lock	16	1.725
Bucket locks	1	0.408
Bucket locks	2	0.612
Bucket locks	4	0.62
Bucket locks	8	0.735
Bucket locks	16	0.763
Many locks	1	0.411
Many locks	2	0.315
Many locks	4	0.291
Many locks	8	0.302
Many locks	16	0.212
Local Hists	1	0.014
Local Hists	2	0.014
Local Hists	4	0.012
Local Hists	8	0.008
Local Hists	16	0.011
No Locks	1	0.015
No Locks	2	0.029
No Locks	4	0.047
No Locks	8	0.073
No Locks	16	0.099

Appendix Table – Part Two Runtimes

Processors	Peacock Pixels Off	Monochrome Pixels Off	Peacock Percent Error	Monochrome Percent Error
1	0	0	0%	0%
2	294189	34941	2%	1%
4	372431	3361759	3%	49%
8	1374157	4972220	9%	72%
16	2192120	4025476	15%	58%

Appendix Table – Part Two No Lock Errors