Matthew Silva

Professor Alvarez

CSC 415

19 April 2018

Assignment 2: Parallel Solution

My parallel solution for Assignment 2 works by first creating a copy of the raster. Creating a complete copy with _TIFFmemcpy removes the dependency of calculating values based on the raster and then writing those values to the raster. The program then decomposes the work of computing filtered pixels. Decomposition happens in bands, with each thread handling its fair share of the number of rows that need processing (e.g. the first three and last three rows would not need processing with a 7 by 7 filter). The task is decomposed into sequential (not interleaved) rows. If the rows do not divide evenly by the number of threads, then the last thread may handle fewer rows than the rest (the implementation will overestimate the number of rows handled by each thread in the case of uneven division). The threads are then launched with all of their required parameters, such as starting row index, ending row index, specifications of the raster, a pointer to the raster, a pointer to the copy of the raster, and a pointer to the filter. The threads are fully free of dependencies and will compute the pixels they are responsible for and place these filtered pixels back into the raster as they are calculated. All of this happens in parallel. The threads then join at the end so that execution does not return to main before the image is completed.

I progressed through several iterations of my parallel solution before arriving at the almost fully parallel solution I submitted. At first, I did not realize that the entire task could be

parallelized. Instead of creating a copy of the raster to resolve dependencies, I copied newly calculated pixels into an array of ‘outRaster’ arrays, with one passed to each thread. My first improvement was to pass them all the same array, since they would all write to their own part of that same array. However, at first, the threads would not copy this data back to the raster themselves, ending their task after calculating and storing the pixels into the communal outRaster. The filter_image_par function would then use the ourRaster array filled with newly computed filtered pixels in parallel by the threads to write back to the raster sequentially. This means that the entire task of writing the computed pixels back to the raster would not be parallelized.

The next iteration of my parallel solution used barriers to synchronize the writing operation for the threads. I wanted to be able to write back to the raster in parallel, but my implementation was not able to write back to the raster in parallel without proper synchronization. My threads read from the raster to compute new pixels. If, by chance, one thread wrote its last lines to the raster before the thread below it had finished calculating its first line of values, then the pixels computed would be based off of the filtered pixels written by the thread above instead of the original raster pixels. This unsynchronized change in the raster’s data would result in incorrect pixel calculations. I used barriers to synchronize this writing and parallelize it. The threads would compute the values of their respective rows and store them into the outRaster. Then, the threads would reach a barrier and have to wait for all threads to finish computing based off of the unchanged raster before they start writing. This barrier allowed the threads to write to the raster in parallel without creating dependencies in access to the shared resource of the raster. The writing operation was then parallelized, but required that each thread

finish before proceeding to the next task in parallel. I knew that the program could truly be completed in parallel independently since there should not be data dependencies that prevent progress in a problem like this.

To fully parallelize the program, the raster could no longer be a shared resource in terms of computing new pixels. The only way to do this was to create a copy of the raster, passing this copy to every thread to keep as an unchanged copy of the data in raster to use for calculations. The threads now read from the copy of the raster to compute new pixels and write directly to the raster. I hope that the overhead of copying the raster sequentially using a _TIFFmemcpy is not significant. I wonder if the task of copying the raster could be parallelized using pointer arithmetic to copy certain parts of the raster in each thread.

The AVX implementation computed 8 pixels at a time. It would take the first index of a group of 8 pixels and perform the filtering operation on that pixel. However, for each of the filtering operations on the 3 by 3 or 7 by 7 (or other filter size) around of the pixel, it loads in the equivalent pixel for that pixel and the next 7 pixels into a 256 bit AVX register. The filter's value for that position is also loaded 8 times consecutively into an AVX register. The multiplication is then completed in parallel for 8 filtering operations at once. I did not implement this but the sum of these pixel-filter products should stay in registers and be accumulated across all 9 or 49 (or other filter length) pixel-filter combinations required for one pixel. My implementation was not ideal for filtering images in which the number of columns to handle is not a multiple of the register length for floating points (8). The unevenly dividing columns of every row are handled sequentially, so a small portion of every row is computed without AVX instructions. I feel that the best way to do this would have been to handle only the last few rows (>8) sequentially.

However, this would involve taking sets of 8 values from two rows at once sometimes, which I think would be very bad for the AVX register loading instructions that require consecutive memory values. My AVX implementation was very slow compared to my other implementation because I spent most of my time trying to optimize my parallel solution.

The most significant speedups I achieved in this assignment were from parallelizing my results writing operations and from code motion and use of common subexpressions. Parallelizing the process of writing back to raster strengthened the readability of my code, while the code motion and especially the common subexpressions made the code very difficult to read. The code motion helped to speed up the code by about 16% alone. The speedups of the parallel solution over the sequential baseline are recorded in the following chart. The program was run in seawulf to generate data to calculate the speedups.

Image Filter Combination	4 Thread Speedup	8 Thread Speedup	16 Thread Speedup	32 Thread Speedup	48 Thread Speedup
earth-2048.tif / Gaussian-blur	3.670	6.472	11.015	10.648	10.528
earth-2048.tif / Identity-7	3.816	7.027	11.762	12.140	12.718
earth-8192.tif / Gaussian-blur	3.667	6.503	10.938	11.401	12.200
earth-8192.tif / Identity-7	3.869	6.997	11.991	12.067	13.170

I think that the reduction in speedup figures for higher numbers of threads for earth-2048 with the Gaussian-blur filter was because of the large overhead for thread creation in comparison to

the smaller problem size. It is notable that the speedups increased with more threads for all other trials, all of which were larger problems. I notice that the speedups also seem to approach 16, which would suggest that the maximum number of instances of parallel execution on Seawulf is 16. However, in class we discussed that seawulf has something like 24 cores with 2 hardware threads per core. As such, I would think that I would achieve close to a 48-fold speedup running with 48 threads. Hopefully we can discuss why the results seem to be skewed.

Overall, I think that my solution achieved acceptable speedups. Almost all of the problem is parallelized. Now, calculating and writing pixels is completed fully in parallel, which is a big improvement over my original solutions. According to Amdahl's law, we should be able to continue to speed up the program even further by allocating more threads on the appropriate hardware, since almost all of the work is completed in parallel. The only operations completed sequentially are the copying of the raster, the computation of the areas to be worked on by each thread (the decomposition of the task), and the creation of the threads (the assignment of the decomposed subtasks).