15-418 Final Project Report

William Qian, Joseph Gnehm

December 10, 2019

1 Project URL

The url for our project is `https://geejoseph.github.io/qian-gnehm/`, our code is hosted at `https://github.com/geejoseph/qian-gnehm`.

2 Summary

For our Final Project, we decided to conduct an exploration into parallelizing image segmentation. More specifically, we analyzed the graph-based reduction sweep image segmentation technique described by Farias, Marroquim and Clua in (Farias 2013). Our project involved building the codebase from scratch, including a sequential version, several iterations in CUDA for the GPU, and an OpenMP implementation, geared towards the GHC machines. Our analysis suggests moderate speedup gains with both parallel implementations, where the speedups improve (or stabilize) as the size of an image increases. We do however note some drawbacks to our parallelization techniques, including synchronization burdens as well as moderate hyperparameter tuning which may potentially limit further parallelization attempts as well as reduce the generalizability of the implementations.

3 Background

Image segmentation is the process of splitting an image into its distinct components. Used primarily for a variety of computer vision tasks, such as medical imaging or object recognition, many flavors of image segmentation are widely available. The class of graph based algorithms, represents the

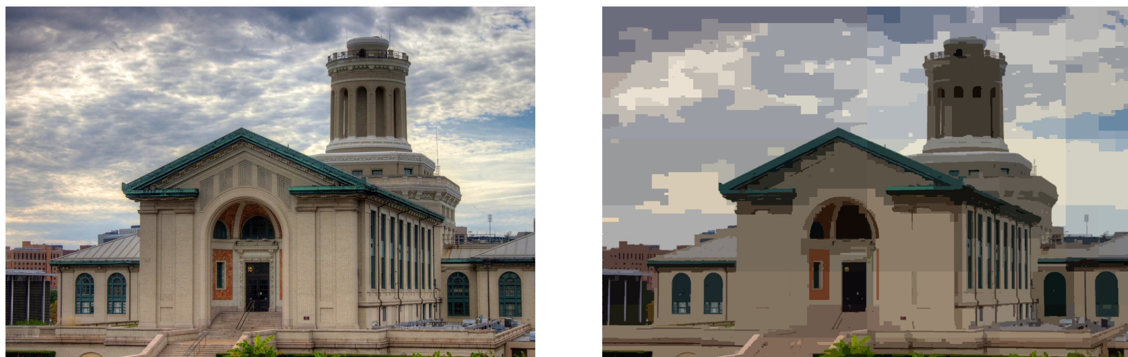


Figure 1: Hamerschlag and Image Segmented Hamerschlag!

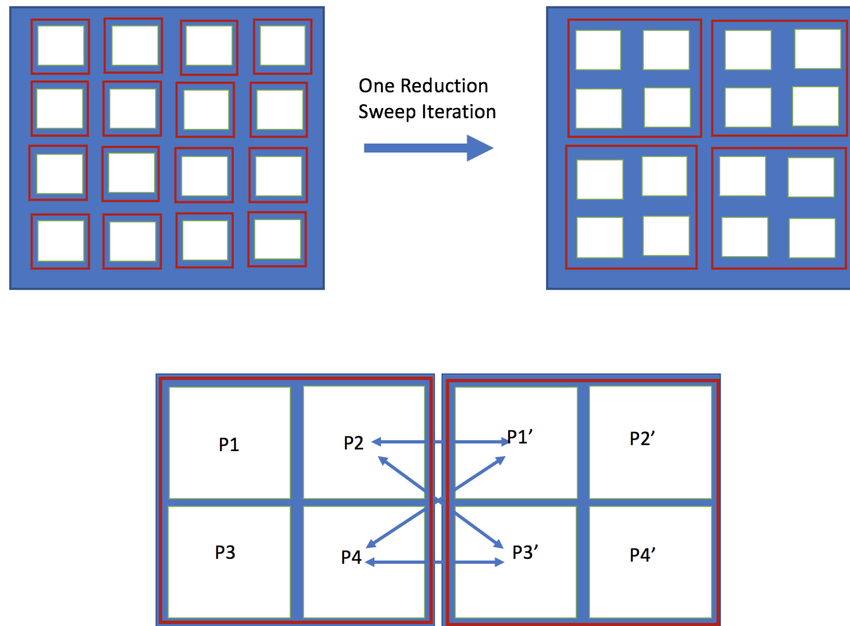


Figure 2: Top: One pass of reduction Sweep, red outline indicates region, white are pixels. Bot: Process Merging two Regions, arrows show all comparisons/merges that need to be done, we sweep by doing all rows first, for all regions, and then all diagonals

image as a graph, and creates segments based of how the edges are defined. One very popular algorithm, reduces image segmentation to determining the minimum spanning tree, and is often solved using parallelized Boruvka's algorithm. We instead focus on another technique, coined the reduction-sweep algorithm, that enables very interesting parallel opportunities.

The reduction sweep algorithm represents every pixel as a node, and is connected to every neighboring pixel with an edge. The edge value represents the euclidean distance between the two pixel RGB values, and if the distance is sufficiently small, we can go ahead and merge the two pixels. Instead of explicitly combining the vertices, we instead keep track of which segment every vertex belongs to, where initially, every pixel belongs to its own segment. When we merge, we assign the new cumulative segment the weighted average of the two segment's RGB values, where the initial larger segment represents the new merged segment.

Note that this particular interpretation leads to some potential avenues of parallelization. Intuitively, two segments that are very far away from one another should be able to be computed simultaneously. We formalize this intuition through our reduction sweep. We describe it as follows: We do a series of passes on the image, where we eventually reduce the entire image into one region, where a regions partition up the pixels and implies that the pixels within it have all been compared and sufficiently merged. Every phase doubles the width and height of the region, and so we would have 1 by 1, then 2 by 2, 4 y 4 and so on. During each reduction phase, we would effectively combine 4 quadrants to generate the new regions, were each quadrant is merged by joining on the edge pixels of every quadrant. The process of combining the the quadrants, across rows, cols and diagonals is known as the 'sweeping', and the order of sweeping produces different segmentations but does not introduce concurrency bugs. It should be noted however, during a merge of pairwise quadrants, only 1 thread can touch that pair, as every pixel in the quadrants are susceptible to

multiple reads/writes. Finally, we should note that this algorithm was introduced by the reduction sweep image segmentation paper referred to below.

For our project, we have broken down image segmentation to several phases. First, we apply a gaussian blur with a sigma value of 0.8 in order to smooth out the image for processing. We then apply our reduction sweep algorithm to find the segments, and finally update the pixels to reflect their new color. For us, we are interested in looking at the speedup of the reduction sweep phase in particular. We note that updating the pixel colors is inherently tied to the data structures used within reduction-sweep, and so we include it in the parallel speedup.

An interesting part of this project is the actual quality and difference between the results produced by different algorithms. The algorithm in the paper produces noticeably more "square" regions and ours does also. We ignore the final step the authors take in merging regions that are too small size-wise because this is similar to a reduction sweep but completely sequential. This seems to be the main purpose behind gaussian blurring and the post-processing step too. Neither one of these steps is parallelized in the paper.

We now present both our CUDA and OpenMP implementations.

4 Approach

4.1 Sequential

Before describing the specific parallel approaches, we will first formalize some of the Background information. In particular, we use three primary datastructures. First, we have a Pixels datastructure, which holds the RGB information for each pixel. Then we have a next datastructure, which is a union find like datastructure that contains pointers to other pixel indexes that eventually reaches the "leader" pixel which represents the segment. Note that leader pixels are distinguished as they point to themselves. Finally, we also have a size datastructure that stores the size of the segments at the leader pixels index.

The sequential version was heavily inspired by the algorithm presented in the reference paper. In particular, upon testing some of the other sweep orders, we determined that row, rowDiagonal, col and colDiagonal sweep order provides the best image segmentation, and performance changes across the different orders were negligible. The following parallel implementations were constructed based off of our own judgement however.

4.2 CUDA

As the subsection suggests, we've provided cuda implementation, targeting the GHC machines. We ultimately constructed three attempts at a CUDA solution, each one trying to address some of the particular downsides of the previous iteration. Our first implementation, we attempt a direct, naive translation from the sequential version to a data parallel one. This was met with several challenges. First, the sequential implementation used 2D vector arrays for the pixel,next and size datastructure. To simplify the CUDA implementation, we opted to use 1D arrays and thus refactored the sequential implementation to maintain consistency for comparison. The main challenge of the naive implementation, is the natural complexity of the reduce sweep function. In particular, every sweep attempt (ie row sweep,col sweep, diagonal sweeps) required different grid and block Dimensions. Every block would look after a square of pixels, in particular, we initially set every block to lock after 16 by 16 squares of pixels, partitioning the image, until the region values exceed these values at which point we use the region dimensions as our block size. On the other hand, the number of threads in a block is dictated by the number of threads that can act

independently in the block itself. As an example, given the 16 by 16 block with a bunch of 1 by 1 regions and sweeping by row, the number of threads would be 16 * 8, where every even column would be able to act independently. In contrast if we had 2 by 2 regions, then we would only have 8 * 4 threads, since all every thread must handle all row edges in its region, so it must do 2 comparisons. This implementation makes sure that every thread we spawn will do meaningful work. Aside from the constant resizing of the parameters, after every sweep we must make sure to synchronize the function calls, to avoid any potential races. In regards to updating the pixels, we can easily do so, by letting every pixel simultaneously access the next array, until it finds the leader. We do a small optimization where once we find the leader, we automatically point the initial pointer to the leader, to reduce the chain we would need to traverse. We did experiment with potentially updating the entire chain for every search in next, but quickly realized that the chain length is upper bounded by O(logn) where n is number of pixels, and so the added complexity was not worth the work of reducing the entire chain. Since updating the pixel values was embarrassingly parallel, we simply kept it throughout our CUDA attempts.

During the naive implementation, we quickly made the observation that within a block, the threads would only interact with datastructure indices that were within the block. As a result, seeing how shared memory provided good speedup gains in assignment 2, we decided to try to integrate shared memory into our solution. With this idea, we now split our CUDA solution in two. The first part utilizes the shared memory, effectively, calculating the regions until it reaches 64 by 64, and the second phase is identical to the first attempt, except we start the region size of at 64 by 64. There were some unique challenges with shared memory, namely that in the first phase, we now have 32 by 32 blocks, with 1024 threads a piece. Each thread is responsible for grabbing some value from the datastructures, and putting it into shared memory. In order to correctly put them into shared memory, we also need a special mapping from the original datastructures, to the shared memory datastructure, which can be calculated by the blockIds and ThreadIds (note this will be important later). For the reduce-sweep phase wrt to the shared memory section, the implementation is similar to the first attempt, except as the regions start growing bigger, more and more threads begin to sit idle while very few do work. As will be seen, the results of this implementation were mixed.

Finally, we further optimized our solution, by taking direct inspiration from the sequential version. More specifically, this solution is equivalent to the first attempt, except as the region grows increasingly large, we instead calculate these iterations with the sequential solution code. The intuition is that, rather than having the overhead of calculating the new dimensions and spinning up the threads, running it sequential could potentially be faster. However, instead of mindlessly copying from kernel memory to host memory in order to run the sequential version, we instead spin up a kernel function with 1 block and 1 thread, that executes the sequential code, thus reducing the copying overhead.

Note that we also considered some other smaller optimizations, such has having a transposed version of the datastructure, so that when we try to access down the original column, it becomes cacheable, but eventually rejected the idea, because of the overhead of having to copy over the data, and maintain it.

4.3 OpenMP

The basic approach with OpenMP was to start with the sequential code and then make it as amenable as possible to OpenMP. The first question was what to parallelize, since the authors of the paper we started with do not actually explain what they do for parallelization. Eventually we figured out that the columns and rows of the sweeps can be done in parallel, since they af-

fect the same regions. Somewhat counter-intuitively, it is the columns of the horizontal sweeps (and first diagonal sweeps) and the rows of the vertical sweeps (and second diagonal sweeps) that can be parallelized. If you do it the other way, the image is not correct and there are race conditions. Incidentally, we figured this out the hard way. This is because we have to respect the potential region boundaries when parallelizing, if a thread touches a region at all then it has to be the only thread that touches that region in the sweep. Here is a helpful figure from the paper with an annotated version side by side. The annotated version shows each thread as a blue line.

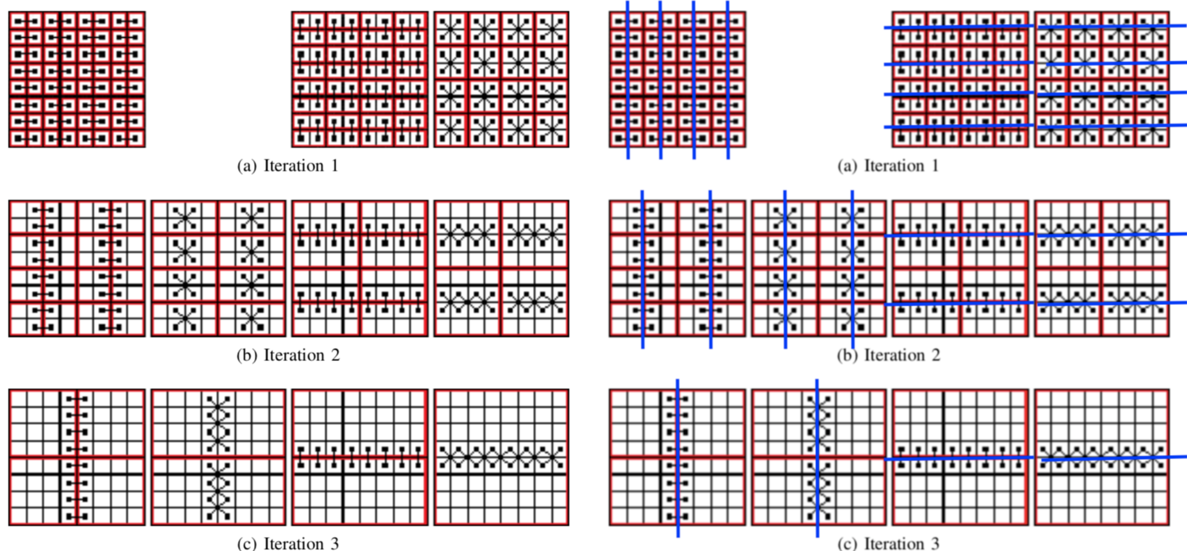


Fig. 7. Illustrated example of the Reduction Sweeps performed on an 8x8 image. Each iteration performs horizontal, diagonal, vertical, and (again) diagonal verifications. The exception is Iteration 1, which skips the first diagonal sweep due to the algorithm's formulation.

Fig. 7. Illustrated example of the Reduction Sweeps performed on an 8x8 image. Each iteration performs horizontal, diagonal, vertical, and (again) diagonal verifications. The exception is Iteration 1, which skips the first diagonal sweep due to the algorithm's formulation.

From then we had to modify the code so that the sweeps were done as these rows or columns and not in their original order. It also became clear that like with the CUDA implementation, eventually the last iterations would become more expensive since they are just one or two columns. We expected that playing with the scheduling in OpenMP might be able to give us a good speedup, with threads taking multiple columns or dividing them up. It was also encouraging that this algorithm already divides the image up into columns and rows - so hopefully the workload would be pretty even across threads.

We made sure with this implementation that through all the changes the images produced were exactly the same, so this implementation wasn't cheating in any way. We verified that OpenMP was using all the threads available to it (even by default). We were not sure whether to expect that the last two sweeps in each iteration would be much faster, since they were working horizontally in memory and may be able to use the cache better. We did not expect that the iterations would be unbalanced between sweeps, since they are very similar and our images are not biased in any particular direction (e.g. all images of trees with the segments being vertical).

As we will see in results, the total number of iterations very closely tracks the log of the largest dimension of the image. So the first few iterations are very important.

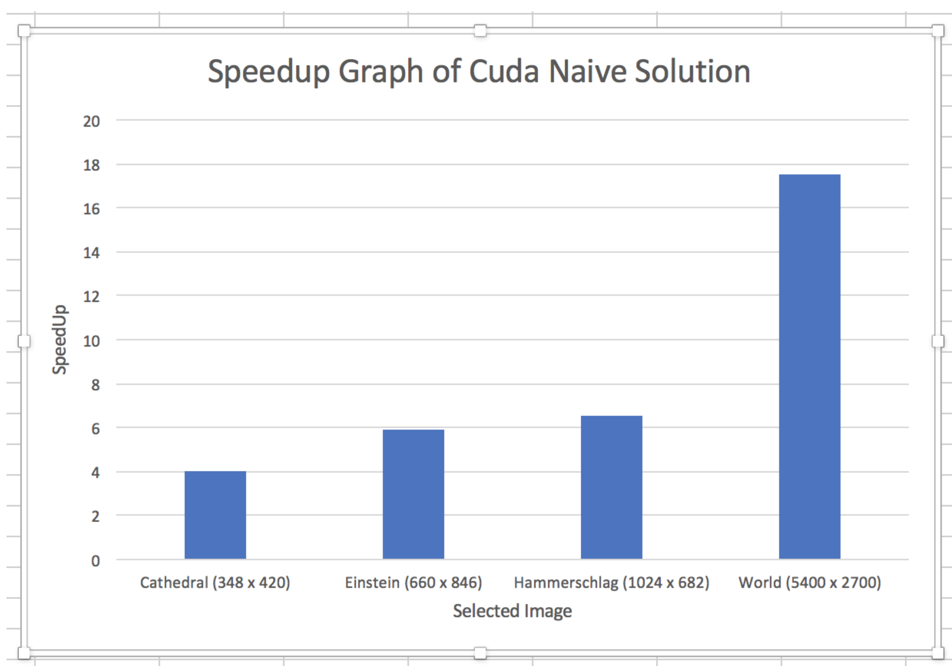
5 Results

The sequential code was run on a 3.2 GHz Intel Core i7 Processor, while the CUDA code was run on an Nvidia GeForce GTX 1080 GPU. The OpenMP code took advantage of 8 of the cores used by the sequential version, since the GHC machines have eight 3.2 GHz Intel Core i7 processors. These machines also have x2 hyperthreading so a total of 16 threads (see specs here).

5.1 CUDA

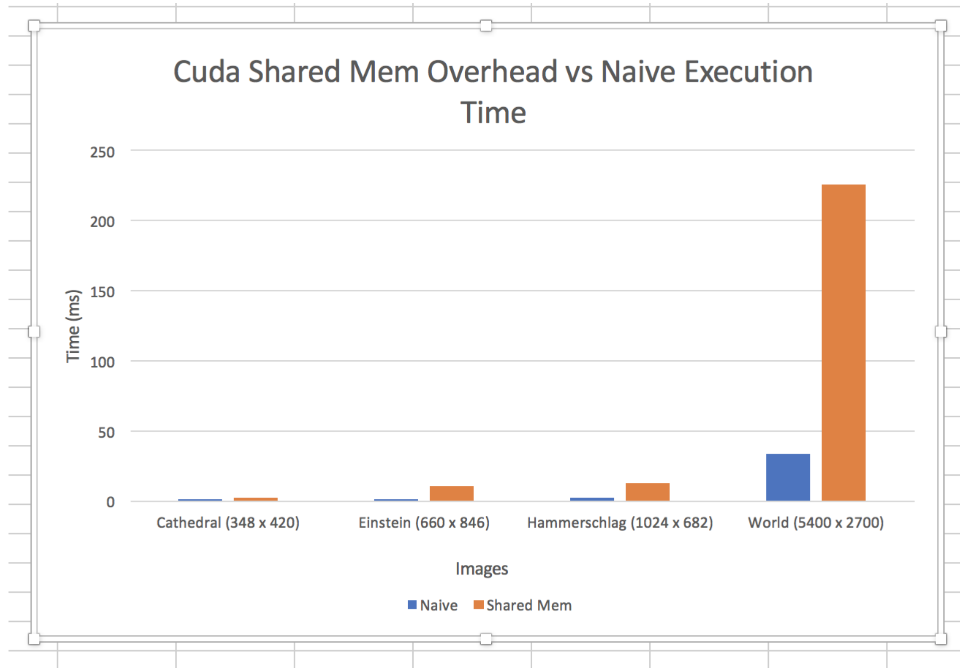
In order to determine performance, we timed the execution of reduce sweep plus pixel update with the sequential version and parallel version, except we excluding the malloc and copying overhead for the parallel version when looking at speedup directly. We do however consider this overhead later on. In order to gain a better look at the internals, we also measure the time for every reduce phase iteration, shared memory phase execution time, and sequential code execution time for the third attempt in order to draw comparisons between sequential version, and the multiple CUDA implementations.

We tested on a images with varying dimensions, ranging from 348px by 420px to 5400px by 2700px.



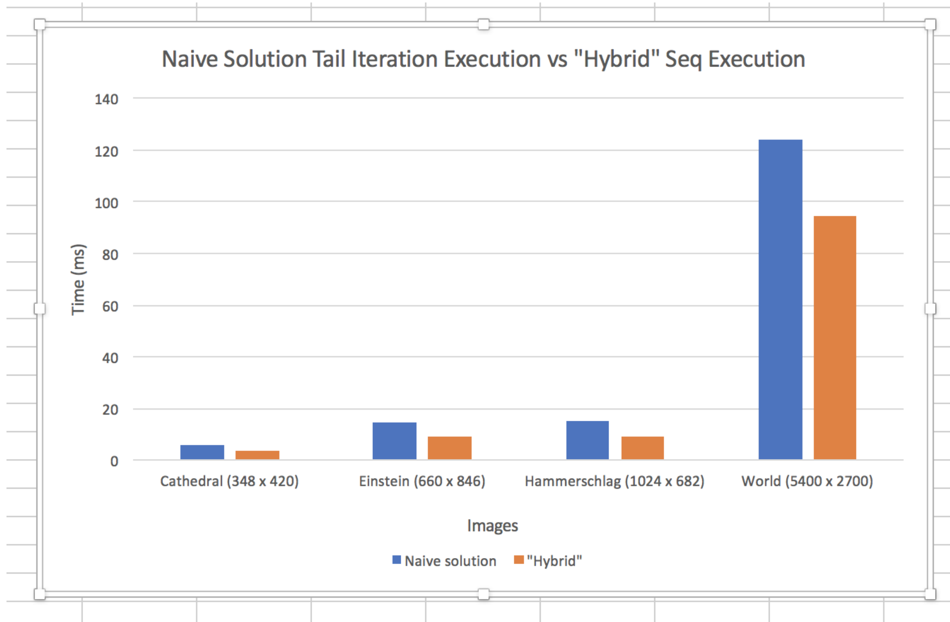
Below shows the speedup of the naive cuda approach compared to sequential approach on images of varying size. As can be seen, the speedups differ drastically, from around 4x on our smallest images to around 17x on the largest one. We considered this speedup satisfactory, particularly the 15x speedups and believe it aligns with the nature of our algorithm. By design, the early reduction phases contain very small region sizes, which implies that each independent thread is required to do less work overall, and there exists more opportunity for independent threads as there are more regions overall. The effect of parallelization is amplified with larger images rather than smaller ones, as more work is needed in these earlier reduction phases as the image simply requires more partitioning into blocks, and so parallelization can be easily abused. However, as the reduction

phase iterations get higher, there is less opportunity to have independent threads working, and each thread will need to do substantially more work, since the regions are naturally larger. Finally, another factor that limits the parallel speedup, is the necessity of having barriers between sweep phases in order to maintain correctness. The existence of synchronization objects inherently creates overhead, but it is speculated not to be a major bottleneck in our case. The primary reason for this is because since our every thread in our solution is given meaningful work, and all regions are the same size, every thread has approximately the same amount of work. Therefore work seems to be distributed relatively evenly and so the barriers simply exist as reinforcement rather than large overhead.



Moving on to the shared memory approach. It was surprising to see it perform so poorly. Our figure describes the execution time for the shared memory section, which covers reduction-sweeps from 1 by 1 to 32 by 32 using shared memory, compared to the total execution of the naive solution on reduction sweeps from 1 by 1 to 32 by 32. As the size of the image dimensions increase, the shared memory overhead increases substantially faster then the naive solutions execution. After analyzing the work load of the shared memory phase, we note that the primary differences between the two implementations are the need to load and unload the shared memory and a mapping is required to correctly point the shared memory to the actual memory's location. We empirically tested, and determined that the actual load and unloading of the shared memory contributes approximately 1 ms of overhead at most. As a result we look at the mapping. We determine that the mapping involves modulus and division operations, both which are relatively arithmetic intensive. We also note that this mapping is called in tandem with our find function, which gets constantly called by our interface. With these two points in mind, it seems reasonable that the potential performance gains of shared memory if overshadowed by the necessary mapping that maintains its correctness.

Above is a figure of the iteration times of the sequential version compared to the naive parallel solution for an image. Note that this behaviour was similar across all our tested images. As mentioned earlier, the initial iterations are extremely expensive for the sequential version, and



extremely cheap for the Cuda version. However, we also see the reverse behaviour emerge in the later iterations. This can be attributed to the fact that the overhead of maintaining the parallel solution exceeds the perceived benefit of parallelization. We hoped to decrease the tail iterations by introducing our pseudo hybrid solution. To some extent, the solution managed to reduce the overall runtime as shown in the graph. The serial execution of the last couple iterations are less than the Naive solution across the board. The only issue is that it requires some hyperparamater tuning for when to start serializing execution, which could be somewhat arbitrary.

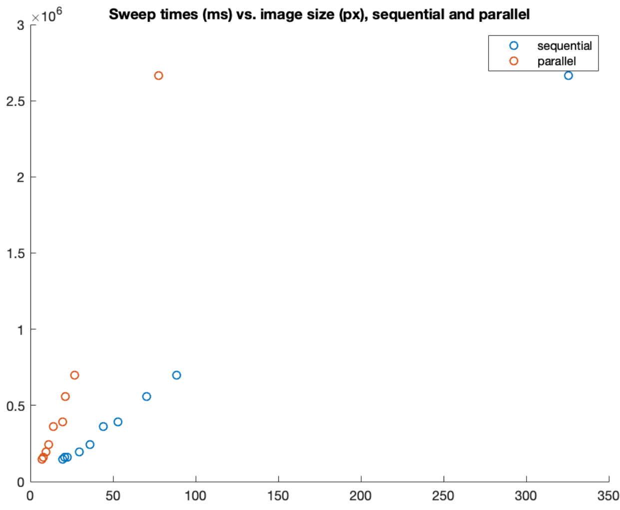
Overall, we found Cuda to be moderately successful, the high speedups provided for large image

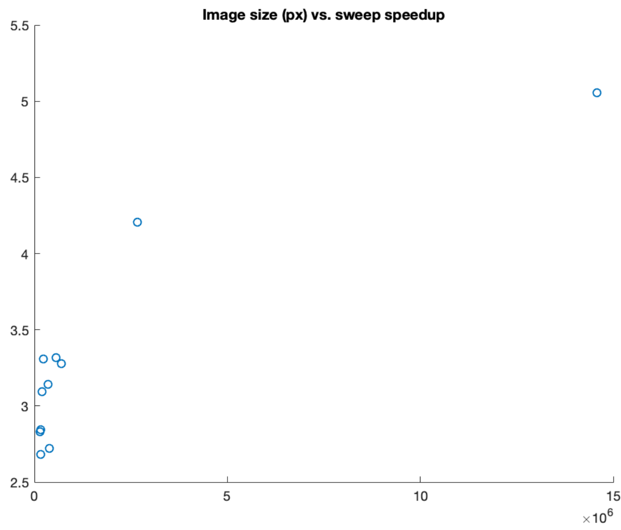
is a potential draw to the approach, while it is difficult to increase speedup on lower images by the nature of the algorithm. It should be noted however, that the cuda mallocing and copying appear to have a relatively large overhead, which is something to consider when running the entire image segmentation pipeline.

5.2 OpenMP

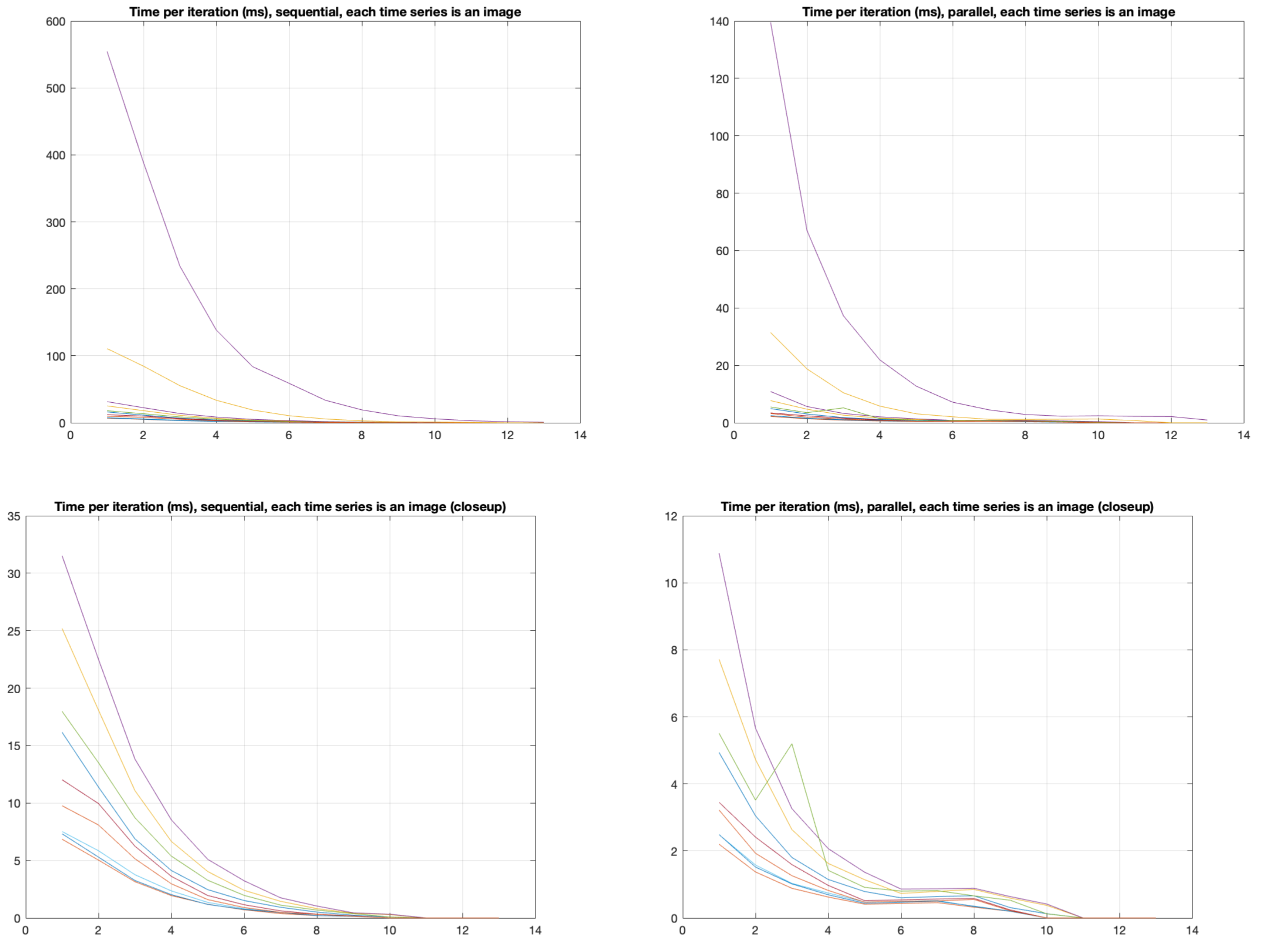
Our main results with OpenMP are an at least 4x speedup. It seems as though as the images get even larger, we may get a higher asymptotic speedup going to about 6x, however we decided to focus by optimizing on the smaller images because they run much faster and we had a problem running even the largest image in this case. But the algorithm should continue scaling. The highest speedup we can expect here is 8x because we have 8 processors and there isn't a lot of empty time while running each row or col, so we don't expect the hyperthreading to help too much. But because of overhead it seems reasonable that the curve is this gentle. For OpenMP we took timings from 11 images of various sizes and compositions. Here are graphs for the times of both and overall speedup:

Looking more closely, the two big components are the sweep time and the update time. The sweep time is what we are focusing on with the algorithm. We see a good speedup here:





This shows even reaching a 5x speedup with the largest image. It was also interesting to see how the total time each iteration took tailed off in both the sequential and parallel cases.



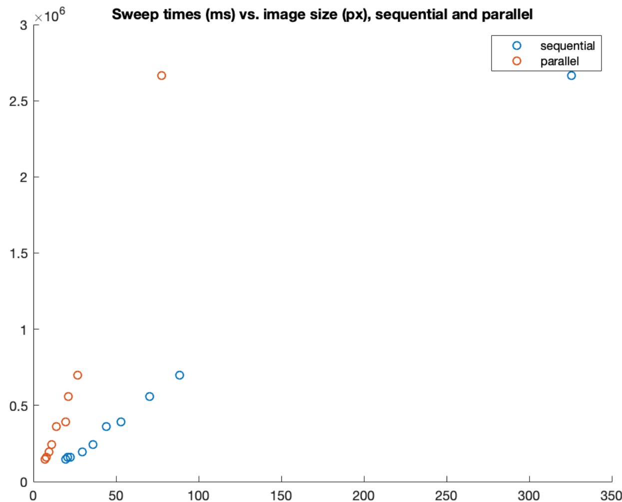
We can see the sequential (left) and parallel (right) are very similar, having the same number of total iterations, but the time of the parallel iterations decreases more quickly at the beginning. However like with CUDA we get a significant bump at the end with parallel because of the overhead.

Trying to control number of threads didn't help, it was best just to let OpenMP always use all the threads. This makes sense since all the sweeps are synchronized between themselves.

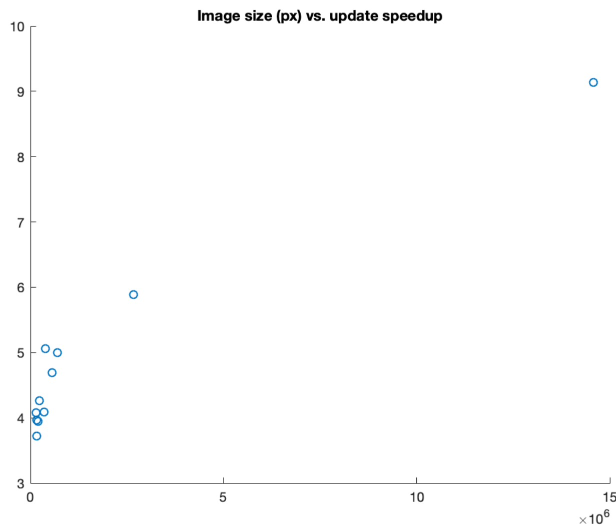
One surprise to use was that more in-depth scheduling strategies did not help much. The basic dynamic strategy with chunk sizes of 1 worked very well, it was basically the same as the static strategy. We attempted to use the dynamic strategy with larger chunks or chunks inversely proportional to the number of threads, but this did not help, it seemed to just add overhead. Using the guided strategy was even worse, probably for the same reason. This is interesting because the authors reported that they thought the random nature of the finds in the code would unbalance the tasks. But perhaps having a whole column or row evened this out.

It also did not seem like there was one kind of sweep that was very inefficient or thrashing the cache. It seems to take much less time than the other sweeps. To check this we looked at the second iteration of the algorithm for all the pictures. We used the second iteration because the first iteration doesn't have the first diagonal sweep. This also should show the benefits or drawbacks of any caching that will happen in the rest of the iterations. But, to our surprise, the first diagonal sweep still seems to take very little (even constant-seeming) time. This could be because it has to check fewer regions or is redundant. It could also be a bug, but we have checked it against the

pseudocode from the paper, and there don't seem to be any obvious discrepancies in the image. We thought it could be that the first sweep loads a bunch of the pixels into the cache or prepares them, but the first diagonal sweep continues to take a very short time. It does help a little, but not much. Swapping the rows or columns, although creating an incorrect image and lots of race conditions, also actually made the program go slower. So this more solidly cements our choice.



The update part of the time was a good control, because it showed normal scaling and what you would expect from the theoretical speedup, since it has no strange dependencies, it just tries to update every pixel in the image. We can see it here:



There may still be more to explore with this OpenMP implementation. In particular, it is possible to parallelize even further into the rows and cols, since there are parts that exclusively cover some regions. Since the chunking seemed to work best with a chunk size of 1, this more fine-grained parallelism may be very promising. To do this we would have to rewrite the algorithm to parallelize over these "region areas". We also did not do much hyperparameter tuning with OpenMP in the inter-

ests of keeping the results we got general. This seemed to work well, but it may also be interesting to investigate changing the value of `t`, or how similar the color of two regions has to be to merge them.

For further analysis, it may be helpful to look at the total time used by the threads during the sweeps using the clock function.

5.3 Possible ways forward

We may be able to try different data structures, although the find structure seemed to be very helpful in this case we never measured how much time it actually saved.

In general, there are three assumptions we could relax going forward to potentially get a much higher speedup.

For the purposes of image segmentation to be used in the real world, it may be helpful to trade off speed for accuracy. So we could not verify all the edges in the image, potentially making some regions smaller, but also not having to cover all the pixels.

We could also stop assuming determinism, and maybe using basic locks allow the threads to interact, or allow the first thread to merge two regions. This could let us parallelize more widely across rows or columns, although the output image would differ each time.

Finally, it may also help if we could use just a little bit of prior knowledge to help with scheduling. Then we may be able to assign more threads to certain areas of the picture in advance, so that they are balanced. This may be even more important if we were to try testing out much larger images, where it seems like we would get the speedup we want.

6 References

Farias, Ricardo & Marroquim, Ricardo & Clua, Esteban. (2013). Parallel Image Segmentation Using Reduction-Sweeps on Multicore Processors and GPUs. Brazilian Symposium of Computer Graphic and Image Processing. 139-146. 10.1109/SIBGRAPI.2013.28.
`https://www.cs.cmu.edu/~418/doc/openmp.pdf`
`http://www.cs.cmu.edu/~418/lectures/05_progperf1.pdf`
`http://jakascorner.com/blog/2016/06/omp-for-scheduling.html`
`https://gcc.gnu.org/onlinedocs/libgomp/Runtime-Library-Routines.html`
`http://www.cs.cmu.edu/~418/assignment_writeups/asst3/assignment3-f19.pdf`

7 List of work and distribution of total credit

Sequential implementation: 80% William Qian, 20% Joseph Gnehm
CUDA implementation: William Qian
OpenMP implementation: Joseph Gnehm