Report for the final project
“Parallel flood fill”

By Kirill Aristarkhov and Brandon Katz for CS140 S22

Introduction

Our program implements the bucket fill function that is included in many image editing services. There hasn’t been any previous discussion of the problem in class so we were learning as we progressed. Upon passing input of image template and points to start filling from, the program would fill each of the regions in parallel. The number of threads used (passed in the input file) corresponds to how many regions of the image can be concurrently filled. The program can either randomly generate a template from scratch, using Pthreads to generate lines that build enclosed regions, or it can read a premade image and color it as instructed.

Methods

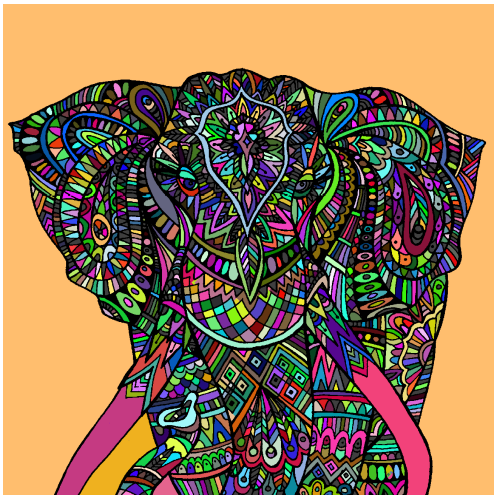
When running the code, the user can choose to either randomly generate a grid-like image with regions to fill, or pass the program a premade black and white image. The user can then choose the number of regions to fill, and whether to fill from randomly generated starting points or from a predetermined list of points that can be passed from a file or command line. The 2nd option allows for most precise results and best consistency for the sake of comparing timing. The user chooses how many threads to run with. With 1 thread, the regions are filled in serially. With more than 1 thread, each thread checks a global variable to see if there are still points to fill from and the regions are filled in parallel.

To create a blank matrix as the first step of generating a random template, we used openMP with a collapsed for loop to assign white values to each index of the image. We then used pthreads to draw the lines on the blank image. We discuss this more in the results section.

We used Pthreads in order to parallelize the filling of discrete regions in the image.
Filling a premade image: A strictly black and white .ppm file can be passed to the program to be colored. The user can choose to allow each thread to generate a random starting point within the image to start coloring with a random color while the number of regions colored is less than the entered count. With this option there is potential for undefined behavior when multiple threads randomly start coloring within the same enclosed region, and this happens more often when the image contains large regions. Normally threads stop coloring in a direction once they reach a black boundary and they do not color pixels that they have already colored with their own value. We encountered issues where multiple threads would continuously fight over coloring the same region and prevent the program from terminating. This occurrence is evident below:

Where there is a single region, and all threads are fighting over the coloring of the one region. In an attempt to resolve this, we defined a function that orders threads to forfeit a region if they encounter a nonblack nonwhite color that is not their own. Though this helps the program to terminate, it leads to some regions being multicolored in a way that depends on the order of which threads arrive and what rank their color is. To achieve a more definite output, it is better to pass a predefined list of points to start from to the program, and threads atomically pop values from a global queue while it is not empty, terminating when there are none left.

We originally attempted to achieve parallel filling of single regions but it was difficult to achieve in a way that would avoid redundancy and provide speedup from serial implementation. Trying to synchronize threads to not encroach on each other's territory within a region led to memory complications and we decided to abandon this goal in favor of coloring multiple regions instead, which we also decided would generate a more visually appealing image.



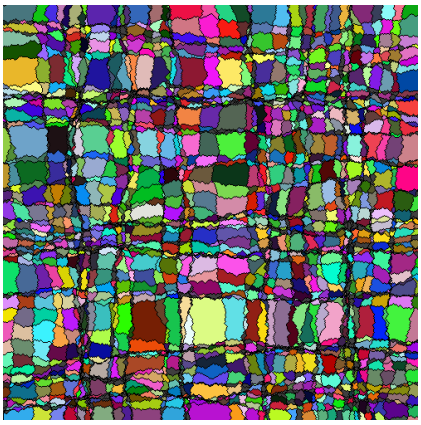
With regards to individual responsibilities, we completed the entire project using pair programming. As for the timeline, we don't remember particular dates but the work progressed in the following order:

- Figuring out image formats and file i/o in c++
- Exploring PPM format and drawing random gridlines on a blank matrix
- Parallelizing the previous code with Pthreads and OpenMP
- Trying one version of ange-lines based flood fill, turned out to be inefficient
- Writing the spreading center flood fill algorithm
- Setting coordinate choosing functionality, parallelizing flood fill for multiple regions with Pthreads
- Adding image coloring functionality
- Testing and debugging

Results

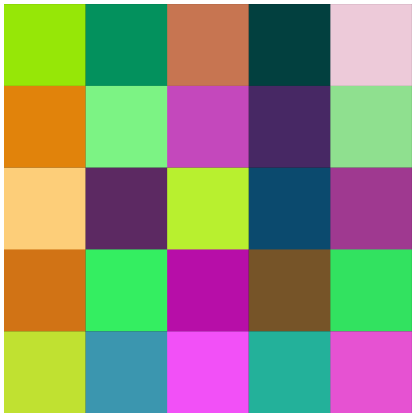
We first tested the timing speedup of generating grid templates, and we used Pthreads and openMP to parallelize this. As we increased the thread count, the total number of line traces to generate was divided among the threads and we observed expected continuous speedup since there is no locking overhead that requires threads to wait. It is perfectly fine for threads to reach race conditions on a single index of the matrix because this is how enclosed regions are generated.
We did not report this time however because our initial plan of using these grids was discarded as we instead focused more on testing the time on premade images to avoid randomness.

Among many other tests we did involving randomly generated starting points, we determined that these 2 following tests would be best for demonstrating the parallel speedup of our program. We previously wanted to test on extremely large samples of filling regions by generating 100,000+ random points and letting recoloring happen freely, except the threads fighting over regions led to this method of testing not being viable as the results were nondeterministic. This is what the result of these tests looked like:



We tested on pre-generated 5000x5000 images, one with a total of 2500 evenly distributed regions and another with only 25, and passing a set of points such that each region is filled exactly once. Roughly the same number of total pixels is colored in each image.

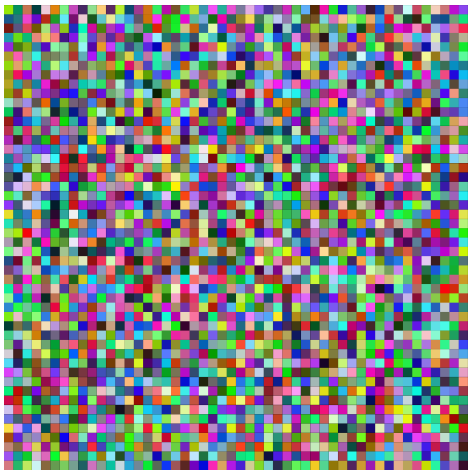
5000x5000 pixels, 25 regions:



Number of threads	Execution time	Speedup	Efficiency
1	2.778 seconds		
2	1.457 seconds	$S_2 = 1.907$	$E_2 = 0.954$
4	0.819 seconds	$S_4 = 3.392$	$E_4 = 0.848$
8	0.501 seconds	$S_8 = 5.545$	$E_8 = 0.693$
16	0.530 seconds	$S_{16} = 5.242$	$E_{16} = 0.327$

We see that for this image size we start to lose speedup after 8 threads because there isn't a big enough workload to balance the overhead of a large number of threads synchronizing on popping starting points from the global queue. Though the image sizes in both of these tests are the same, the second test has a larger quantity of black lines and thus a few thousand less pixels are filled. This explains why the first test runs slightly slower than the second test when both are executed with 1 thread.

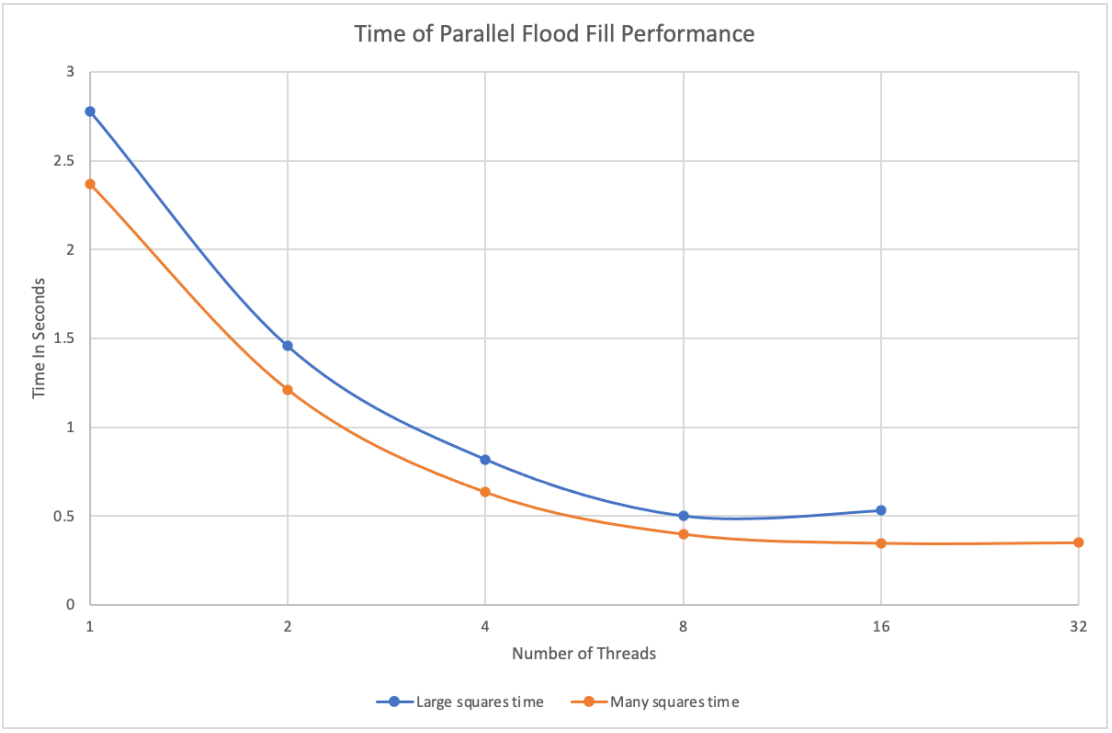
5000x5000 pixels, 2500 regions:

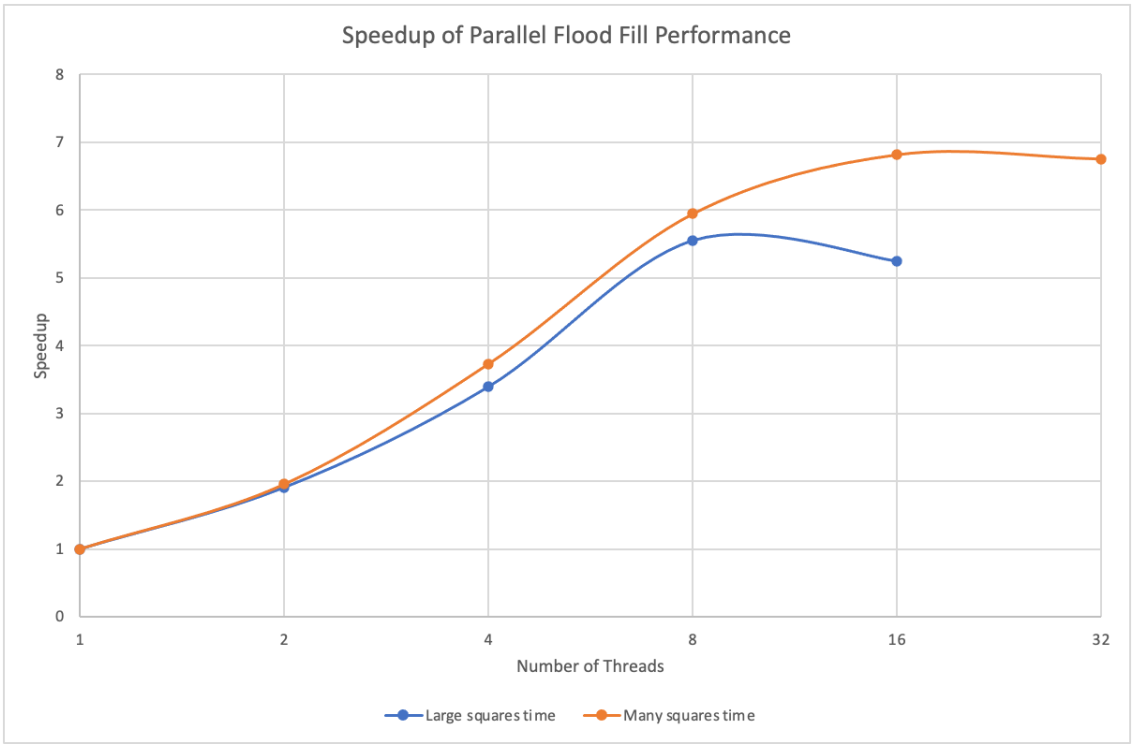


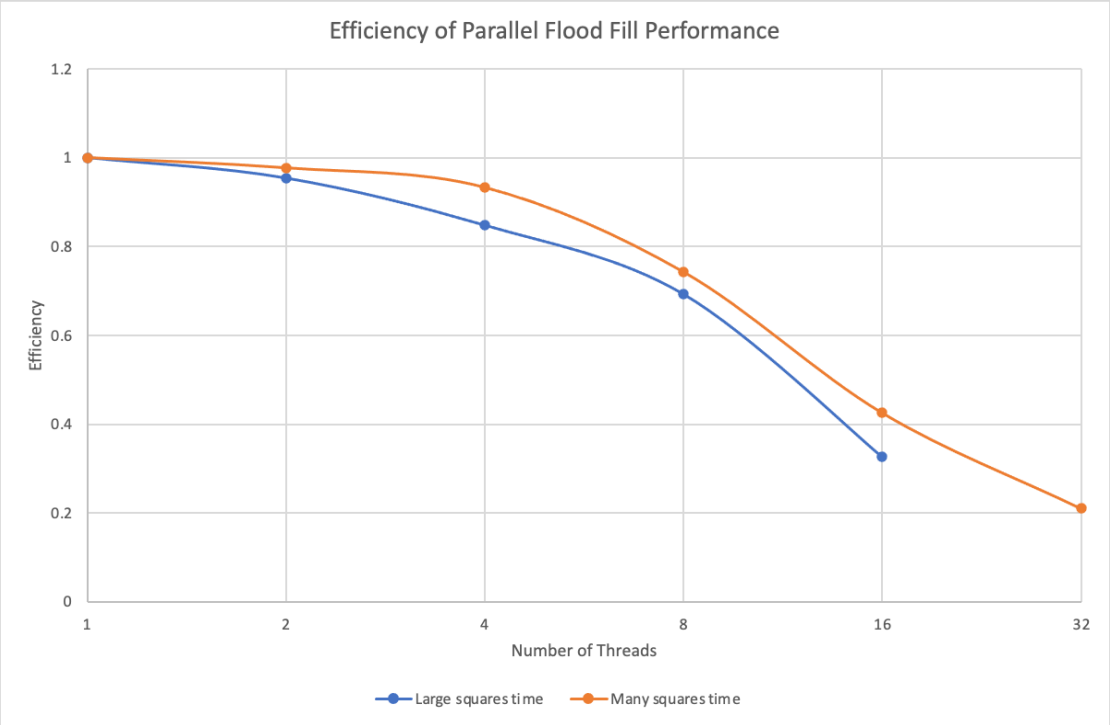
Number of threads	Execution time	Speedup	Efficiency
1	2.369 seconds		
2	1.212 seconds	$S_2 = 1.955$	$E_2 = 0.977$
4	0.635 seconds	$S_4 = 3.730$	$E_4 = 0.933$
8	0.399 seconds	$S_8 = 5.942$	$E_8 = 0.743$
16	0.348 seconds	$S_{16} = 6.814$	$E_{16} = 0.426$
32	0.351 seconds	$S_{32} = 6.751$	$E_{32} = 0.211$

Here we see that the efficiency for 16 threads is .426 compared to .327 that was observed on the test with fewer colorable regions.

When running the program with many threads, we are able to observe better speedup from serial execution when there are many discrete regions to fill. This allows the workload to be more optimally distributed among threads. We were able to run up to 32 threads before seeing overhead slowdown, unlike the slowdown we observed at 16 threads in the previous example. Our solution is weakly scalable because we must increase the problem size in order to maintain efficiency while increasing the number of threads used. When we split the 5000x5000 image into more regions, it allows more threads to be working at once without interfering with each other. Unfortunately, this also comes with a drawback, as filling smaller regions leads threads to finish more frequently and build up more overhead locking time. Despite this, we still observed faster execution times for the image that had more regions.
Had we instead chosen to statically assign blocks of work to each thread before launching them, it seems that we would have been able to have greater efficiency for larger numbers of threads. However, that solution did not make sense in the contexts of the other modes of coloring that we attempted to implement during our project, so we did not get the chance to test this.
If we were to further increase the size of the image and the number of fillable regions however, we would be able to still see significant speedup for a larger increase of thread quantity. Unfortunately, the biggest time cost of our program is the file i/o operation and not the parallel procedure. This file size already takes a long time to render from c++ so we decided to stop here.


Time of Parallel Flood Fill Performance
Time In Seconds
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
32
Number of Threads
Large squares time
Many squares time


Speedup of Parallel Flood Fill Performance
Speedup
8
7
6
5
4
3
2
1
0
1
2
4
8
16
32
Number of Threads
Large squares time
Many squares time



Conclusion

We were able to successfully speed up the creation of a random template and coloring of an image using Pthreads to fill unique regions in parallel. While we observe additional speedup as we increment the number of threads, the efficiency is most remarkable at 2 threads. We are aware of areas that we could potentially improve our implementation and we believe that with more experimentation we could have improved efficiency for higher thread quantities, but for the sake of simply reducing total time taken we are happy with the results we observed.

Note: In order to view the input and output images, you must have an image viewing app such as gimp or preview to open the .ppm files, and we used this to convert them to .png images for the presentation

Computer used for testing:

Running on csil-13:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	12
On-line CPU(s) list:	0-11
Thread(s) per core:	2
Core(s) per socket:	6
Socket(s):	1
CPU family:	6
CPU MHz:	3100.000
Model name:	Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz