A Parallel Rasterizing Framework

Abhishek Yenpure
Garett Roberts

June 9, 2017

Executive Summary and Motivation

There are many ways to render an image given an input dataset. viz. Ray Tracing, Rasterizing, Volume Rendering. Among all of these approaches the one that is most deployed in case of real time(gaming, etc.) applications is Rasterizing. Ray Tracing and Volume Rendering are use in scientific visualization and in case where the images can be pre rendered to use later(animations). Since rasterization needs to be done in real time, it is necessary that we make it as fast as possible, and hence we choose to parallelize the algorithm for rasterizing.

A rasterizer is a program which works on polygons of a dataset individually and renders an output image by appliying the scan line algorith on each polygon. The scanline algorithm first tries to find the bounds of the pixel rows and colums to begin which are determined by the vertices of the polygon. For each pixel that occurs in the above calculated bounds, the algorithm calculates the normals, color and the depth values. In case of overlapping polygons, the depth values determine what color and normal values are to be deposited to the pixel.

We started by writing a serial rasterizer. We decided to parallelize it with OpenMP, Cilk, and TBB to compare the differnces between all of them. We focused mostly on the main rendering function and the scan line algorithm to parallelize on. We decided on these two parts of the algorithm because of the impact it had on the program when analysizing with a profiler. (see pages below)

Proccess of Parallelizing

Serial Optimizations

The first thing we did before we embarked on actually parallelizing this project was to optimize our serial code in specified regions. Typically, when optimizing code you look for a certian number of things. The following is what we focused on.

- Cache misses, when the code recalls data that will more than likley no longer be in the cache at lower level memory, it can cause our program to take a lot longer than desired or expected. We focused on trying to keep memory used more frequently local.
- Over allocated variables, for example when variables are created in a for loop. Not only does this increase the amount of memory, but it increases cache reads/writes as well. This can lead to an excruciating dip in preformance. Typically to solve this problem we would try to define the variables and allocate them as soon as possible to reduce the frequency of creation and writing to variables.
- Passed parameter optimization, this is when a funcion is passed a copy or a pointer when not needed. The problem that this creates is the speed at which items can be passed can bottleneck preformance enough to sometimes lead to a noticble difference. Our method of reducing such a problem is to either pass references instead of pointers, eliminate them from being passed at all, or to pass a pointer/reference instead of a whole copy of the data.

Work Stealing vs. Work Sharing

Work-sharing a parallel construct in which a program shares part of the data between different processors or threads in able to execute the program. There are plenty of different patterns to achieve the desired output. Where

work stealing is different is that it will take any work from a thread if it is idle in order to not waste it's processing power.

Summary of optimization

OpenMP

OpenMP follows a work sharing approach for scheduling tasks in parallel, which is not very efficient at load balancing. The program can only be as fast as the slowest thread in the system

Our solution to solve this problem was to try multiple different methods. The first method we attempted was tasks, which we were hoping that the threads would have more data per thread, so that there would be less idle time per thread. This proved to be of minimal success. The next attempt was to try chuncking the data within the for loops. Chunking is the proccess of setting a set size per thread/processor to do a set size of work. This proved to improve preformance, but only minimally. Lastly, we tried to increase the stacksize that each thread could have. This was also with the hope that there would be less idle time per thread and allow for the threads to do more evenly distribute work, this also minimally impacted preformence.

CilkPlus and TBB

Cilk and TBB actually preformed great in our rasterizer. We actually had an incredible increase in speedup(almost linear) evident from our results while parallelizing our code with these two languages. We focused on the main functions and the scan line algorithm implementation. This speedup is good because Cilk and TBB use work stealing model to schedule it's threads.

Results

Serial

Figure 1. is the snippet of the profiling of the serial rasterizer. We were not too concerned with the times for the I/O part of the program. Hence, you'll notice the functions GetTriangles and WriteImage pop us in all the profiling results which will follow. The main objective we had was to reduce the time taken by the core rendering funtion which is the core of rendering work.

OpenMP

Figure 2. provides the profiling results for OpenMP. With OpenMP we notices that it spends a lot of CPU time in OpenMP specific routines. this has our algorithm inside, but OpenMP in general has a lot of overhead of thread scheduling.

CilkPlus

Figure 3. provides the profiling results for CilkPlus. With Cilk we got the best profiling results. The work done by the Cilk threads had very low overhead and gave pretty consistent results across multiple runs.

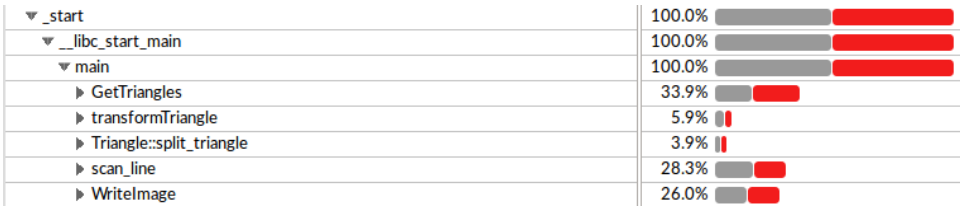


Figure 1: Performance results for Serial version

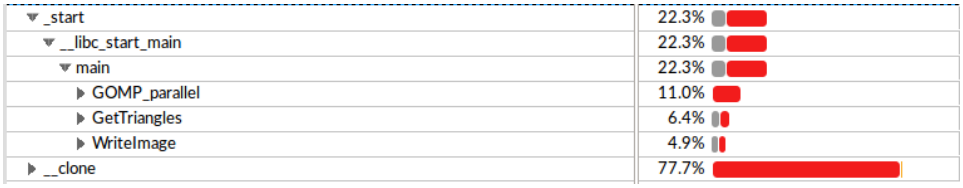


Figure 2: Performance results for OpenMP version

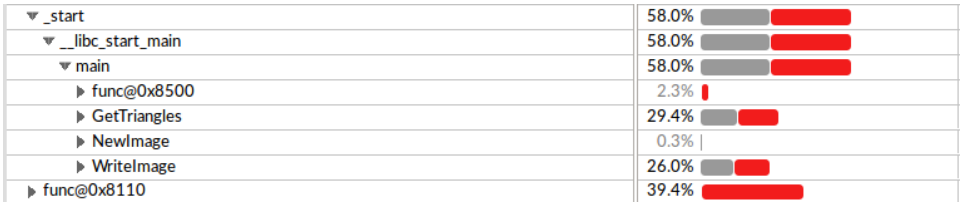


Figure 3: Performance results for Cilk version

TBB

Figure 4. provides the profiling results for TBB. With TBB it's the same story as Cilk, However profiling tree was so huge that it is almost unreadble.

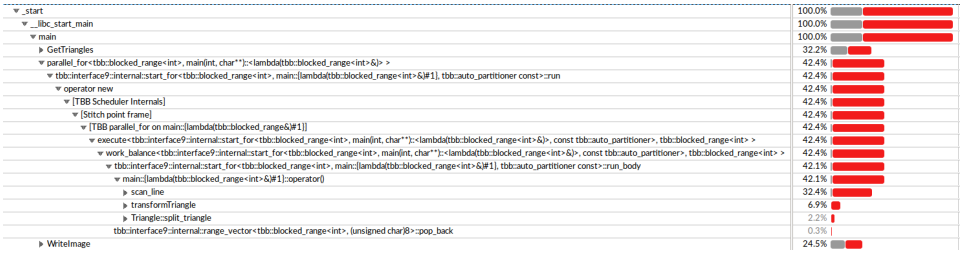


Figure 4: Performance results for TBB version

Conclusion

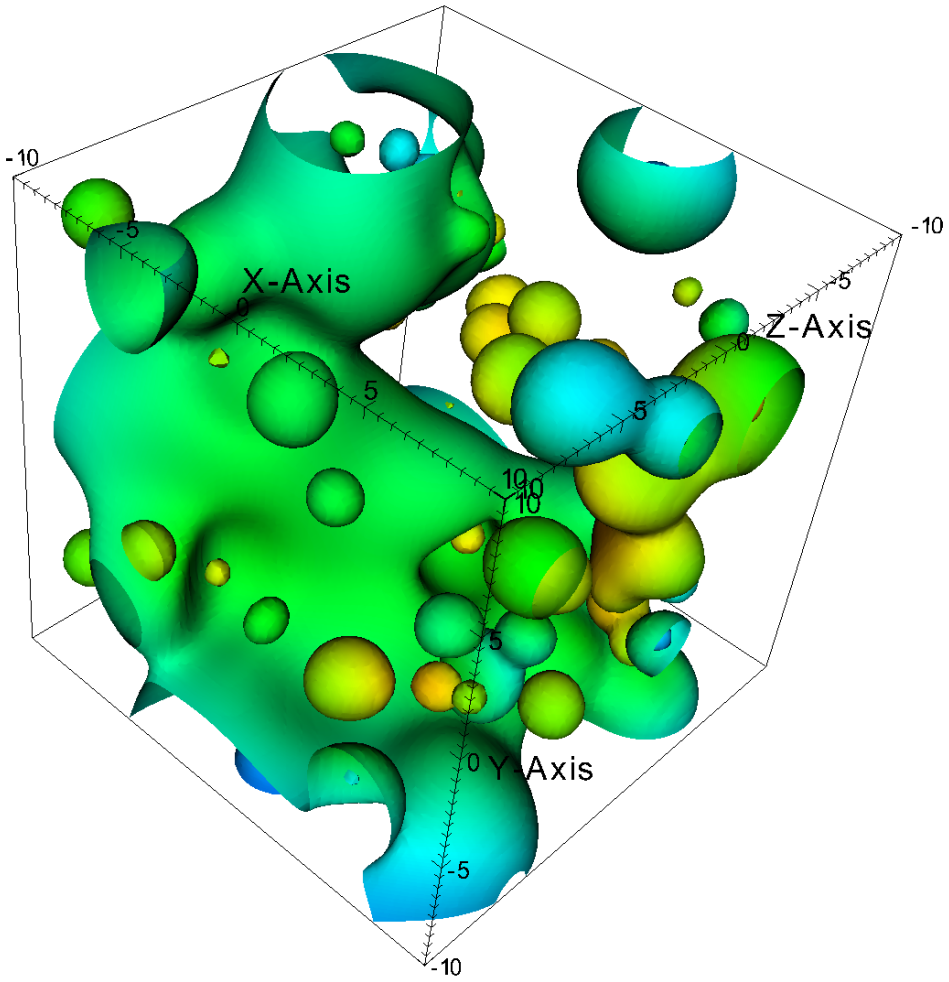
Results				
Datasets	Serial	OpenMP	CilkPlus	TBB
HardyGlobal Approximation (170,939 polygons, 4 cores)	0.46693	0.220864	0.157116	0.136108
8 cores	-	0.120751	0.0850136	0.0818198
Tornado Approximation (5,317,225 polygons, 4 cores)	9.55054	3.08153	2.46171	2.47374
8 cores	-	1.70896	1.24497	1.27784

As you can see from our results, the speed of our program grows almost linearly. Which is a nice case for a parallelism. The linear speed up stays consistent within both sized datasets.

We learned quite a bit through this project. The first was about -O3 flags and that they don't guarantee steps to be proccessed in the same order. We also learned about work-stealing and work-sharing and how they can effect preformence and speedup.
Our rasterizer optimization proved to be a proficent optimization for paralellization which scales well.

Datasets

Hardy's Global Approximation of the noise dataset from VitIt
170,939 polygons



Tornado dataset by Leigh Orf
6,317,225 polygons