Terrain Viewshed Computation and Speedup with Parallel Programming

Ethan Zhou and Jake Adicoff

1. Introduction

In this project, we look at the effects of parallel computing for solving viewshed problem. In the viewshed problem, we begin with a terrain represented in an elevation grid, where each cell in the grid represents an $n \times n$ area in the terrain and contains the height at that point in space. When we compute viewshed, we choose a viewpoint in this elevation grid, and determine the set of all other visible cells in the grid. The viewshed output is a grid that corresponds to the original elevation grid. If a point *i,j* in the elevation grid is determined to be visible, then the point *i,j* in the output is a 1, and if this point is not visible, the value in the output is 0. We parallelized our algorithm in an attempt to speed up the runtime of the viewshed computation

2. Our Approach

This problem is an excellent candidate for parallel computing, since the data used to determine if a point is visible remains completely static during runtime, and the exact same computation is executed for every point in the grid. We can thus simultaneously determine if multiple points are visible without the risk of overwriting necessary data or repeating computations. The outermost two *for* loops of our code iterate through all points in the grid, one by one, and determines if some point, with index *i,j* is visible. Using Open MP's *collapse* function, we are able to compress both loops into one, which allows easy multithreading over multiple loops. Since we are essentially working with a single loop, the parallelization is straightforward. For each thread (specified by the user), viewshed computations for pairs *i,j* are

executed and our viewshed output is updated. With our parallelized code, we are then able to test the speedup that we get given a specified number of threads. We did experiments on two large elevation grids: `usadem2.asc` and `portland_me.asc`. Both are very large grids, and with only one thread, they take 93 and 218 seconds respectively. The long runtime allows us to accurately measure speedup when using more threads. We tested these files using 1, 2, 4, 8, 16, 20, 50, and 100 threads. With these numbers of threads, we would expect to see approximately linear speedup in the beginning, but since a computer has limited cores, we would also expect to see this improvement drop off as the number of threads gets large. We used Bowdoin's Dover server to run our code, which has 24 cores. Therefore, we expect to see very high speedup when using up to 24 cores, but after that we expect diminishing returns for runtime.

3. Results

Overall, the results somewhat match our predictions. The runtime does not actually decrease linearly with thread count, probably because of the way we update our viewshed array. Each thread needs to modify the same viewshed array, possibly accessing the same cache lines. This may result in false sharing, where while each thread executes independent calculations, some instructions require access to cache lines being used by other threads. When this occurs, additional instructions are required to update the conflicting cache line so that no data corruption can occur. Instead, we see an exponential decrease. This fits somewhat with our prediction, as increasing thread count quickly cuts down runtime up around the number of cores in the computer, and only shows diminishing returns past this point.

portland_me.asc		usadem2.asc	
Threads	Runtime	Threads	Runtime
1	92.5	1	217.9
2	76.2	2	149.1
4	63	4	116
8	37.9	8	60.7
16	28.2	16	41.7
20	19.8	20	37
50	12.7	50	20.9
100	9.3	100	14.5

Figure 1: Number of threads and runtimes for both elevation grids.

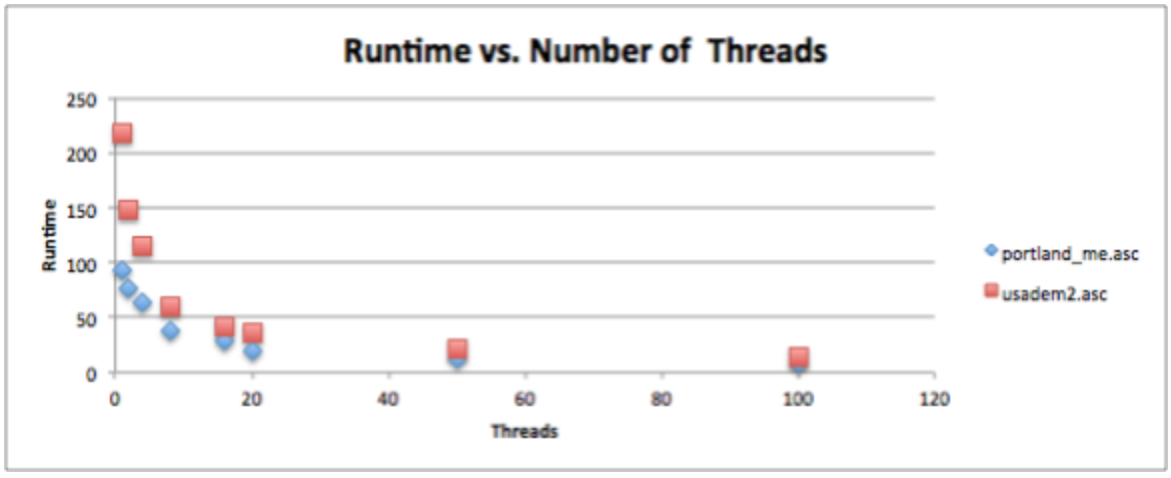

Figure 2: Number of threads vs. runtimes for both elevation grids. (linear x-axis)

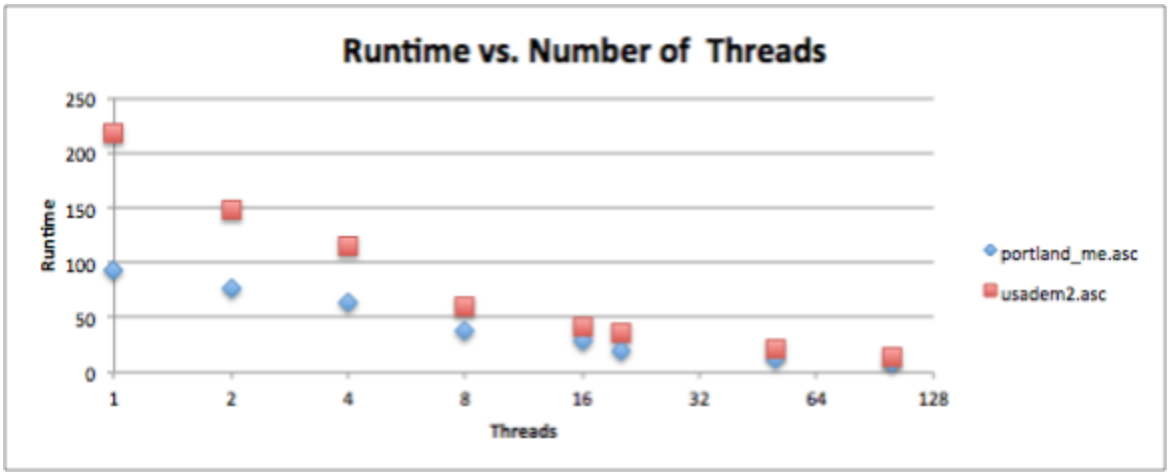

Figure 3: Number of threads vs. runtimes for both elevation grids. (logarithmic x-axis)

5. Conclusion

Multithreading the viewshed problem is very effective in reducing the runtime. Runtime decreases exponentially with number of threads, such that the reduction is significant when the number of threads is comparable to number of cores, and the reduction slows down past this point. To optimize multithreaded performance, it is necessary to avoid all possibility of false sharing, so that no thread operations would negatively affect operations on other threads.