Assessing effects of Parallel Programming on Total Viewshed Algorithms

Duncan Gans

October 30, 2017

1 Introduction/Background

The total viewshed problem is a staple of GIS algorithms. A viewshed is all the points that can be seen from one spot. Although the past lab dealt with tackling this problem using a naive method, it can also be solved using radial sweep, using horizons, and likely other algorithms. For this lab, however, we are doing a total viewshed. This will determine the total amounts of points visible at every point in the grid. This can help us find which point is the most visible, which, like a single viewshed, has tons of applications. For this project we will be using set1.asc, a grid style map. Computing a viewshed for set1.asc without using parallel programming took 135 minutes, or just over 2 hours.

2 OpenMP and Parallel Programming

As computer scientists, two hours was too slow. But, by using OMP's parallel programming libraries, I was able to shrink the running time by huge amounts. While the absence of parallel programming led to runtimes of over three hours, using parallel programming I was able to get the runtime down to under four minutes. OpenMP made this possible.

In essence, parallel programming is the practice of breaking a block of code into chunks and having multiple sections of a processor do each individual block of code. By partitioning the work, and having the computer do the sections concurrently, you can save ridiculous amounts of time.

```
for (i = 0; i < grid->rows * grid->cols; i++)
{
  grid->total_view_shed[i] = createViewshed(grid, i/grid->cols, i%grid->cols);
}
```

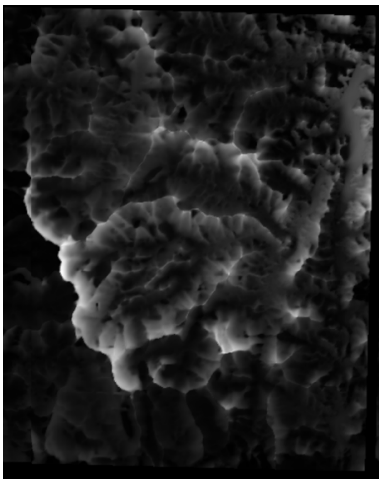

Figure 1: This viewshed illustrates the total viewshed from set1.asc.

This loop is asking to be parallelized. Instead of having each section work consecutively, why not just run all the sections at once. None of the sections rely on previous sections, so there shouldn't be any huge problems. Because of this simplicity, was very easy to parallelize it. To do so, I just modified it to be:

```
#pragma omp parallel
{
  #pragma omp for schedule(dynamic)
  for (i = 0; i < grid->rows * grid->cols; i++)
  {
    grid->total_view_shed[i] = createViewshed(grid, i/grid->cols, i%grid->cols);
  }
}
```

The first part of this, pragma omp parallel, tells the computer it will be doing parallel programming for the section denoted by the brackets. The second part, pragma omp for schedule(dynamic), tells the computer to parallelize the for loop. From here, all you have to do is follow the statement up with a for loop, and it will break it into sections for you and run it. The schedule(dynamic) part tells that it should schedule the chunks of the for loop dynamically. Using this scheduling method is particularly important when the amount of work in each iteration of the for loop could be different. In this block of code, occasionally it will come across a location that has no data value. When this happens it simply returns a zero. Because this makes the runtime vary considerably, it makes sense to allocate the chunks dynamically instead of statically.

3 Results: Parallelization and Speedup

To test the effects of parallelization, we ran multiple experiments on the Bowdoin server moosehead. To test the effectiveness of more parallelization we ran the above parallel program on computers with varying cores from one core to 40. To run these experiments, we used the function below:

```
hpcsub -l excl=true -l 10g=true -pe smp {number of cores}
-cmd viewshed_parallel set1.asc results.asc
```

This ensured that the 10g computers were being exclusively and that it would be as fast as possible. The table and graph illustrate how increasing number of cores decreased the overall run time.

As can be seen, with more cores, the runtime is significantly decreased. With a total of 40 cores, it is possible to actually bring the run time down to just under four seconds. However, as can be seen in figure two, the runtime increase isn't quite linear. While initally doubling the number of cores more than halves the run time, using 40 cores only speeds up the process by a factor of around 33. This shows that the growth is logarithmic rather than linear, which we would expect. Increasing the amount of cores ultimately leads to diminishing returns. However, as is clear, parallel programming is a great way to minimize run time easily and effectively.

Number of Cores	Runtime (minutes)
1	135
2	60
4	32.5
8	17
12	14.6
16	10
20	8
24	6.7
32	5
40	4

Table 1: Runtime Table.

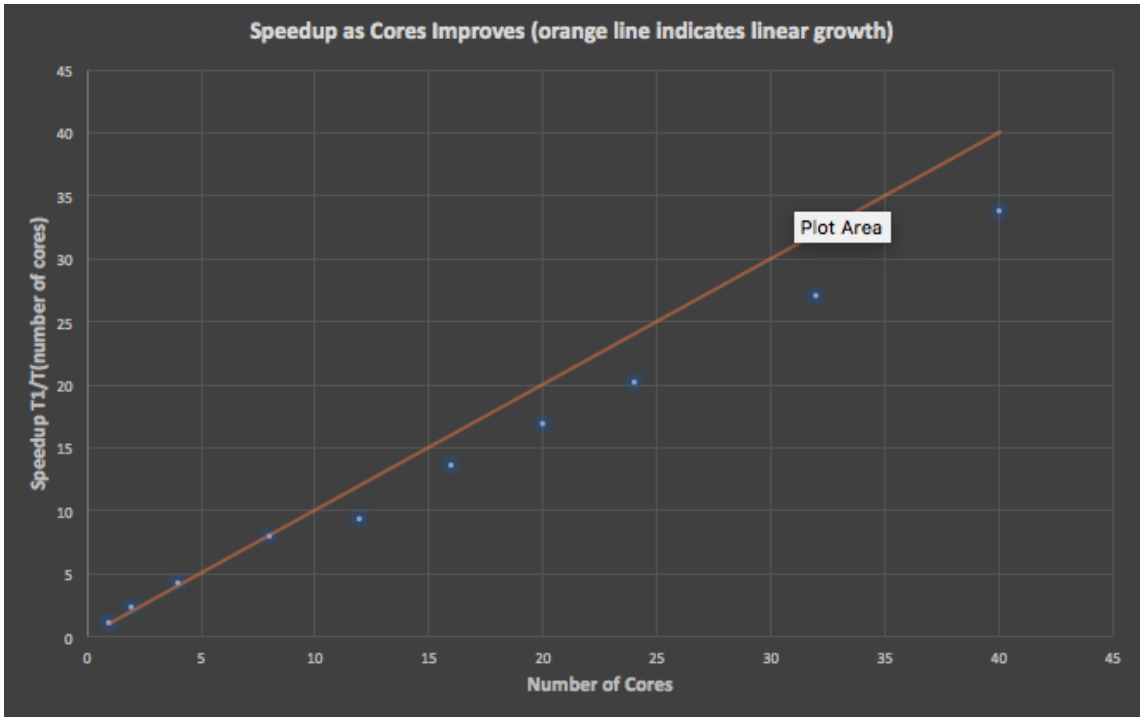

Figure 2: Illustration of speedup as cores increase.

4 Results: grid layout

From here I looked at the impact of using a blocked layout. To more easily and independently look at the effects, I used two separate functions. Although my first results showed that the blocked viewshed took more time than the row major viewshed, I quickly realized that it was because the for loops within total viewshed are different. Initially my row major viewshed went through the grid with one for loop, but the blocked viewshed went through the grid with two loops, one loop for the block, and one for the place within the block. In order to get actual comparable run times, I switched the row major viewshed to be two for loops, one for the row, and one for the column (this will be obvious in my code). By doing this I was able to get better results. Using this version of Row Major Viewshed, the run time was 208 minutes. Using my blocked method of calculating the viewshed, I was able to get to 147 minutes. This is a distinct improvement, that can be attributed to less cache misses. Although I did not do tests combining a blocked layout with parallel programming, I'd imagine we could get an even lower runtime than three minutes and fifty eight seconds.

5 Conclusion and Closing Thoughts

This was the most in depth programming assignment I've done in my Computer Science classes that dealt heavily with run times. It was a fun task to try to limit runtime in multiple different ways (through a blocked layout, and through parallel programming). Furthermore, running programs on the Bowdoin Servers was a fun additional aspect. It really gave me a huge idea of how much one is able to modify runtime with relatively small methods. Furthermore, up until I took GIS, everything ran quick enough that modifications to the code didn't impact the run time in a necessary way. Being forced to create a more efficient algorithm was frankly fun. As is mentioned in section 3 and 4, parallel programming proved to significantly reduce run time. Although adding additional cores yielded diminishing returns, it was still possible to significantly reduce run time. Furthermore, implementing a blocked layout made it possible to reduce run time without using any OMP. Overall this lab pushed me to think about CS in a completely new and enjoyable way.