CMSC 180 Laboratory: Exer 1 Part 2

Aron Resty Ramillano

March 2023

1 Introduction

In this week's laboratory, we explored how to optimize the interpolation program that we have just made in the Python Language. Because of significant inefficiencies, I converted my existing Python code and algorithm into a more efficient C-language counterpart.

2 Objectives

This week's goal is to optimize and gauge the performance of our algorithm and pseudo-code by running some tests that will push its calculating limits, both of the program and the computer. **Documentation** of the results is also required.

3 Methodology

First, we programmed the algorithm with brute force such that we get results with a program that runs fairly inefficiently. But with the help of translation of Python to C code, we are able to cut down the rune time of our code with a range of roughly from 400+ seconds, down to just 10 seconds at maximum.

Second, we have made some changes with when the "bounding points" were fetched from the matrix as we can see that it is repeatedly used for a long amount of time and a number of iterations, this means we can save some memory by not needlessly fetching them all the time.

Third, we have also made our code a bit more modular, where we can use some functions that can be useful for future applications, be called again *easily* for other algorithms that we will be needed when newer problems arise and faced.

4 Results and Discussion

4.1 Research Question 1

What do you think is the complexity of interpolating the point's elevation of an nxn square matrix with given/randomized values at grid points divisible by 10?

My answer above would be n^2 because if we analyze the code, there are nested loops involved, two levels, using the i and j integers to loop over the whole 2d matrix and interpolate the results, which are all running at constant time since they are all arithmetic operations, except for the traverasal of the 2d matrix.

Even though we already have points at coordinates divisible by 10, it is negligible in our pseudocode since we still have to check where we are currently in the matrix and what is the area of the four quadrants of a selected point to interpolate.

This table reflects the assumed complexity that we answered in Research Question 1, on how our algortihm slows down exponentially based on how many inputs are given. The highest operations performed by our algorithm with $n = 20000$ is up to $400,000,000$.

Here is a snippet of the code that is responsible for the majority of the computation:

```
/**
 * It takes a 2D array of floats, and interpolates the values in between
 * the points that are already defined
 *
 * @param M The matrix of elevations
 * @param n the size of the matrix
 */
void terrain_inter(float **M, int n)
{
    for (int i = 0; i < n; i++)
    {
        int min_x = getMin(i);
        int max_x = getMax(i);
        for (int j = 0; j < n; j++)
        {
            if (!((i % 10 == 0) && (j % 10 == 0)))
            {
                int min_y = getMin(j);
                int max_y = getMax(j);

                int area_d = (abs(min_x - i) * abs(min_y - j));
                int area_c = (abs(max_x - i) * abs(min_y - j));
                int area_b = (abs(min_x - i) * abs(max_y - j));
                int area_a = (abs(max_x - i) * abs(max_y - j));

                float elev_a = M[min_x][min_y];
                float elev_b = M[max_x][min_y];
                float elev_c = M[min_x][max_y];
                float elev_d = M[max_x][max_y];

                float elevation = ((area_a * elev_a) + (area_b * elev_b) +
                                   (area_c * elev_c) + (area_d * elev_d))
                                   / (float)(area_a + area_b + area_c + area_d);

                M[i][j] = elevation;
            }
        }
    }
}
```

n	Time Elapsed			Average Runtime (Seconds)	Complexity*
	Run 1	Run 2	Run 3		
100	0.00018	0.00017	0.00017	0.000173	10,000
200	0.00071	0.00070	0.00072	0.00071	40,000
300	0.00158	0.00156	0.00156	0.001567	90,000
400	0.00276	0.00276	0.00277	0.002763	160,000
500	0.00435	0.00440	0.00432	0.004357	250,000
600	0.00619	0.00620	0.00621	0.005587	360,000
700	0.00967	0.00847	0.00844	0.00886	490,000
800	0.01133	0.01102	0.01109	0.011147	640,000
900	0.01410	0.01401	0.01403	0.014047	810,000
1000	0.01795	0.01743	0.01721	0.01753	1,000,000
2000	0.07414	0.07089	0.07161	0.072213	4,000,000
3000	0.16152	0.16322	0.16246	0.1624	9,000,000
4000	0.32216	0.28960	0.29062	0.300793	16,000,000
8000	1.19128	1.16259	1.17755	1.17714	64,000,000
16000	4.78671	4.77117	4.81887	4.79225	256,000,000
20000	7.46443	7.48044	7.54000	7.494957	400,000,000

Table 1: n vs Average Runtime vs Complexity

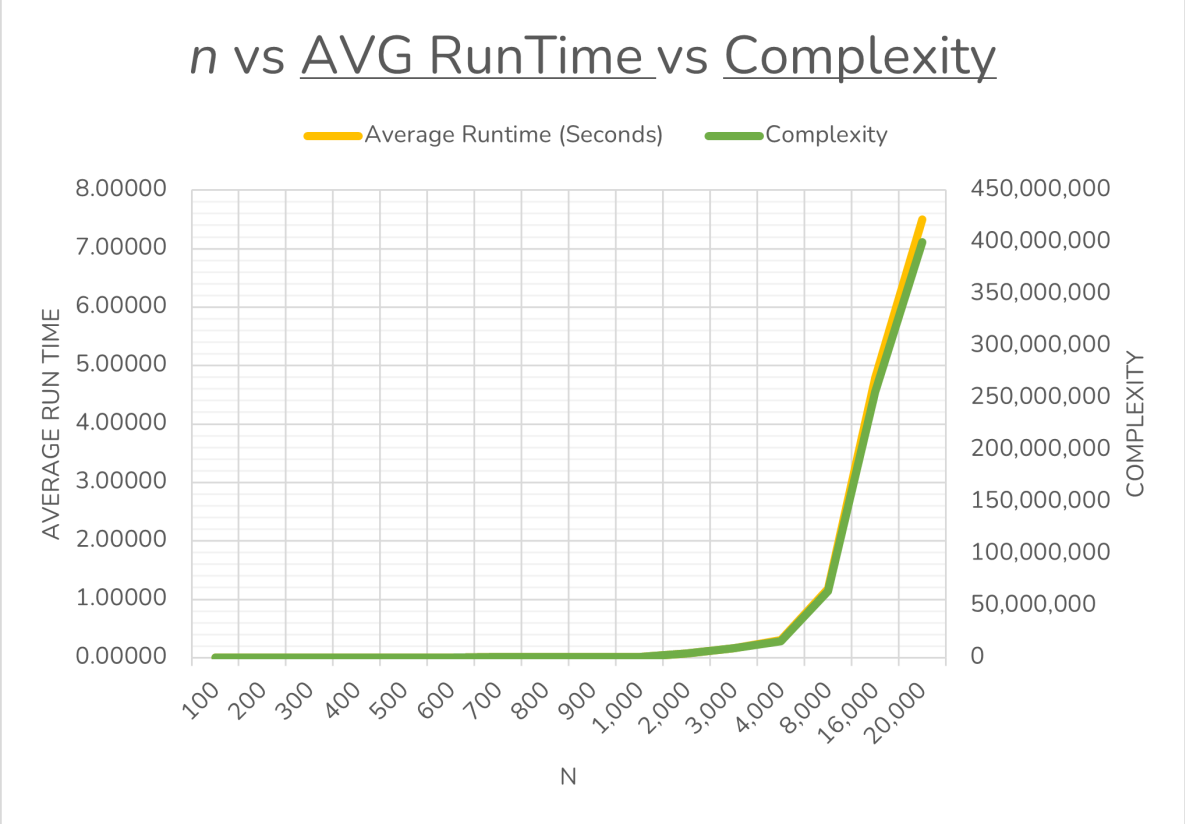

Figure 1: Line graph representation of the table above

4.2 Research Question2

Do the two lines agree, at leastin the form? If not, provide an explanation why so?

Yes, they do agree in terms of form when referenced using n. We can see that both line graphs follow an exponential trend since, it is what we have calculated and assumed in the first place: n^2.

4.3 Research Question 3

Discuss ways on how we can make it better (lower average runtime) without using any extra processors or cores (notice that the word "ways" is in plural form).

If we employ multithreading in our program by utilizing the available cores in our system to do various tasks in concurrent fashion, we can maximize the available pipelines that compute for specific points in our matrix in a much faster way, rather than having it calculate one by one. Multithreading is multiple stream of processes in our system that does concurrent computing of various tasks. This can cut down on processing times and make our algorithms run more efficiently. Running programs parallel is better, sometimes, than serially.

5 Conclusion

This exploration of the capabilities of our machine in computing and interpolating points using given points in the matrices shows a smaller picture of the reality of what is happening in our computers in the modern age. Our phones, laptops, servers are doing various tasks in multi-tasking fashion that we do not even realize its doing many things at once, with the help of multi-threading. Supercomputers utilize the millions of cores available to them to calculate complex genetics, server data transfers, financial institutions, astronomers, researchers, and other various stuff that requires a LARGE amount of computing power. Multithreading, and Parallel Computing will help make things more efficient for us users to unlock new possibilities in what we can create and calculate. My method has some large downsides, but that is where things started back then in the advent of computing. it was only very recently that we really started to maximize the power of multi-cores and threading to improve our throughput in almost everything we do in computers.

6 List of Collaborators

- Lyco Sheen Lacuesta - Assistance in C code and General Inquiries about the Course, LateX assistance.
- Kenneth Renz Tegrado - Analyzing and Explanation of various C code.

7 References

- GeeksforGeeks. (2022, December 21). Multidimensional Arrays in C / C++. GeeksforGeeks. Retrieved March 6, 2023, from https://www.geeksforgeeks.org/multidimensional-arrays-c-cpp/
- GeeksforGeeks. (2023, February 20). How to dynamically allocate a 2D array in C? GeeksforGeeks. Retrieved March 6, 2023, from https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/
- How to return a 2D array from a function in C? Stack Overflow. Retrieved March 6, 2023, from https://stackoverflow.com/questions/5201708/how-to-return-a-2d-array-from-a-function-in-c
- GeeksforGeeks. (2023, January 6). Multithreading in C. GeeksforGeeks. Retrieved March 7, 2023, from https://www.geeksforgeeks.org/multithreading-in-c/

8 Appendices