**CS 473 Project Summary**

For the last two weeks our team worked on the implementation of Strassen’s algorithm in C++. The main function of the algorithm is to multiply together two matrices, doing so with 7 multiplication operations rather than 8 as in simple brute force approach. This concept was discussed extensively in chapter 5 lectures on divide and conquer algorithms. Strassen’s multiplication fits into the divide and conquer classification because of the way it handles the multiplication of arrays of sizes larger than 2 rows or columns. The algorithm first pads any inputted matrix with zeros in order to be able to split the matrix into at least 4 sub matrices, one for each corner. The splitting continues until a 2x2 matrix is reached. Using a series of equations, the algorithm computes seven Strassen constants utilizing only seven multiplications and final computes the output values by using the constants in more summation/subtraction operations. The merge part of the algorithm begins when a computed matrix is used along with 3 other computed matrixes to combine into an even larger matrix utilizing the same functions as for the constants. The merging occurs continues until the final 4 submatrices are combined into the final multiplicated output.

For the testing the program’s outputs and algorithm analysis (below) we generated random floating-point numbers up to 3rd decimal place. The numbers were created by first generating a uniform floating-point distribution and then randomly selecting values from the distribution object. Tested matrix sizes ranged from 4x4 matrices where 16 values were generated to 256x256 matrices where 65536 random doubles were created for each matrix in the multiplication operation. An option to accept user inputted matrix values was also implemented but wasn’t used for analysis.

Initial developments include the implementation of “high school” matrix multiplication where the final value for each element in the output matrix was calculated by finding the sum of products in matching row - column pairs between the two multiplied matrixes. A triply nested for loop that alternated rows in the first matrix and columns in the second for each entry of the output matrix was used to perform the entire operation with the basic operation executing exactly n^3 where n was the total number of columns or rows an input square matrix had.

From the beginning we also invested some time follow OOP design principles. A *Matrix* class was created in order to facilitate easy access to add, add subtract, partition, and print functionalities. A separate *Strassen* class separated logic of the main algorithm from the rest of the code to improve readability and maintainability of the intricate set of equations Streassen’s algorithm required. To see the full code please explore the accompanying implementation and header files.

By the end of Decemeber 12th, our output was consistent across all matrix sizes. The outputs can be seen in attached “outputs.txt” file in the same directory as this file.

We began analyzing both matrix multiplication methods for their overall time complexity shortly after. The testing machines consisted of several lab computers with the following specifications…

Proccessor:  
RAM:  
Operating System:  
Architecture:  
MORE:

Running both the “high school” and Strassen algorithms on matrix sizes of 4, 8, 16, 32, 64 , 128 and 256 for 5 times and taking the average of the five calculated run times the following data was obtained…

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Matrix Size** | **4x4** | **8x8** | **32x32** | **64x64** | **128x128** | **256x256** |
| “High School” runtime (sec) |  |  |  |  |  |  |
| Strassen  runtime (sec) |  |  |  |  |  |  |

Graphs:

Key findings: (answer last bullet in prompt) (wwhcih algorithm faster - in accordance to theory or not - for which inputs) (quatatative evealuation - starssen was so and so much faster)