**CS 473 Project Summary & Analysis**

For the past two weeks our team worked on an implementation of Strassen’s algorithm in C++. The main idea of the algorithm is to multiply together two matrices by only using seven multiplications instead of the standard eight. The brute force, or “Highschool Multiplication” uses eight multiplications to find the result. The Strassen method 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 two rows or columns.

The algorithm first pads any inputted matrix with zeros in order to be able to split the matrix into at least four sub-matrices, one for each quadrant. The splitting continues until the sub-matrices are only 2x2. The Strassen algorithm uses only seven multiplication, and then computes the final values by using addition and subtraction on the seven multiplied values. The merge part of the algorithm begins when four computed matrixes combine into a larger matrix. This process utilizes the same multiplications, additions, and subtractions as it does for the base case. The merging continues until only one matrix remains. This matrix will be the size of the original matrix.

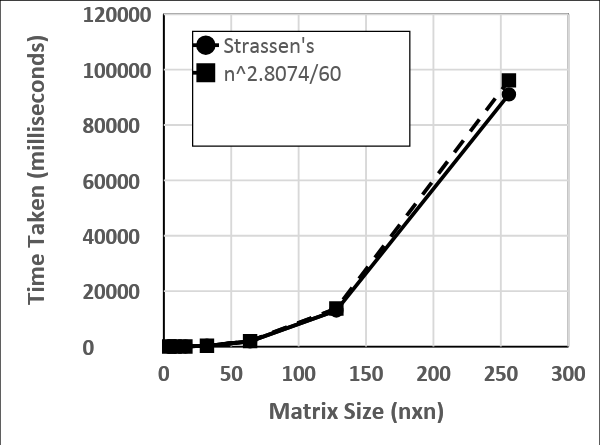
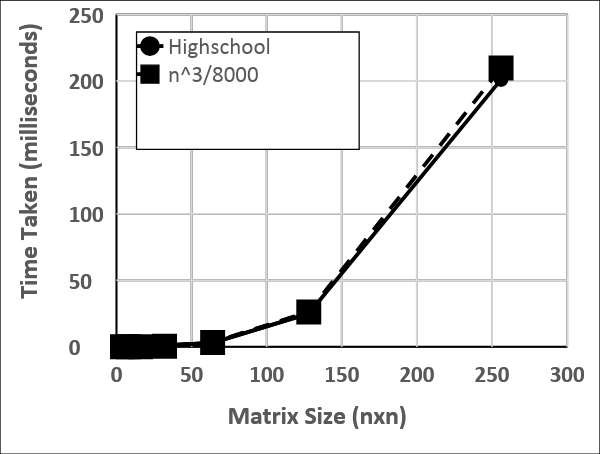
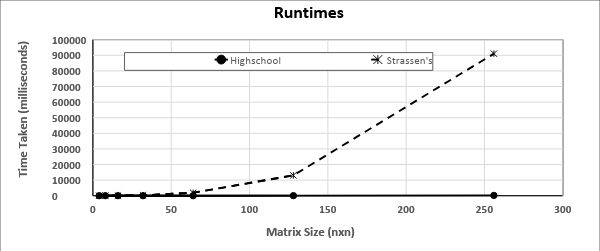
For testing the program’s outputs and algorithm analysis, we generated random floating-point numbers up to three decimal places. The numbers were created by first generating a uniform floating-point distribution and then randomly selecting values from the distribution object. Tested input matrix sizes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

Initial developments included the implementation of “high school” matrix multiplication. The value for each element in the output matrix was calculated by finding the sum of products in the matching row/column pairs between the two multiplied matrices. A triply-nested for loop was used. It alternated between rows of the first matrix and columns of the second for every entry. This executed in exactly n^3 time, where n is the total number of rows of the square input matrix.

We also followed OOP (object-oriented programming) design principles. A Matrix class was created in order to facilitate easy access to add, subtract, partition, and print matrices. 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 Strassen’s algorithm required. To see the full code please explore the accompanying implementation and header files.

We began analyzing both matrix multiplication methods for their overall time complexity shortly after. The testing machine was a personal desktop computer with the following specifications:

* Processor: Intel® Core I7-3770 CPU @ 3.40 GHz 4 Cores, 8 Logical Processors
* RAM: 16 GB
* Operating System: Microsoft Windows 10 Professional
* Architecture: x64 bit

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 produced the following data:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Matrix Size** | **4x4** | **8x8** | **16x16** | **32x32** | **64x64** | **128x128** | **256x256** |
| “High School” runtime (millisec) | 1< | 1< | 1< | 1< | 3 | 25 | 201 |
| Strassen’s  runtime (millisec) | 1 | 5 | 37 | 268 | 1882 | 13057 | 91059 |

Key findings:

Our implementation of the Strassen’s algorithm was unable to overtake brute force “high school” matrix multiplication as n grew larger up to n = 256. We can clearly observe this in the topmost graph on page 2, where Strassen’s algorithm runtime seemingly grows faster than “high school” matrix multiplication. Theoretically Strassen’s algorithm should have allowed fewer computations from the very beginning. However, we see that this is not the case. Strassen’s runtime was approximately 522 times slower than “high school” multiplication for 128x128 matrix. Upon research into the possible causes for this inconsistency between the expected and observed values, various stack overflow answers point to the fact that “the absence of a cut-off point”, and “number density” are the primary reasons regular matrix multiplication seems more effective.

On the positive side, the factor by which regular matrix multiplication is faster decreases as n increases. From matrix sizes 64x64 to 512x512 Strassen algorithm went from being around 627 times slower to being around 327 times slower (512x512 not included in graph for clarity). This trend leads us to believe that eventually Strassen’s algorithm would overtake “high school” multiplication, but only on much larger matrix sizes.

To improve our implementation of Strassen algorithm a proper cut off point would have to be established where regular multiplication would take over (so that 2D vectors won’t have to be copied up to 1x1 size) and also test our approach with more matrixes that had more digits/precision than a regular double.

Both of us would also agree that allocating some time to work side by side on the project rather than remotely would have also benefited the project by creating a more productive environment. Also, having class time to work on this project is often more productive than time out of class, especially since the professor is right there to answer questions.