Strassen's Report

The purpose of this experiment was to implement an efficient version of Strassen's algorithm, such that it would outperform the standard algorithm (for matrices of sufficiently large dimension). My aim was to minimize the experimental crossover point for Strassen's algorithm – i.e., the dimension below which it is optimal to use the standard algorithm to compute matrix products. My program produced the following results. Runtimes are approximate and measured in seconds. Crossover points are also approximate.

dim=	600	800	993[1]	1000	1600
standard	3.5	13.5	36.9	43.5	162
strassen (modified)	2.5	5.1	11.0	10.2	39
crossover	70	50-55	70	90	70

As the table shows, the modified implementation of Strassen's algorithm outperformed the standard algorithm across all dimensions tested, and this difference in performance grew as the dimension grew. For *dim* > 1000, Strassen outperformed standard by a factor of about 4. In fact, Strassen (using a crossover point of 70) reliably outperformed the standard algorithm for matrices as small as *dim* = 100.

The optimal crossover point varied with the size of *dim*, but was reliably between 50 and 90. 70 seemed to be the most consistent optimum.

This crossover point is surprisingly low. The theoretical crossover (assuming that all scalar operation cost 1 unit and other operations cost nothing), can be calculated as follows. Standard algorithm takes 2(n^3) – n^2 operations, while Strassen's algorithm takes 7(n^(lg 7)) – 6(4^(lg n)) operations (for powers of 2). These equations can be derived from plotting the number of operations taken by each algorithm at n=2, n=4, n=8, etc. Setting them equal to each other, we obtain an n_0 of roughly 2^9 = 512.

The experimental crossover of 70 is surprising, then. The most likely explanation for this disparity is that in the standard algorithm, the CPU must access and copy elements from a very large chunk of memory – as such, relatively more work must be done to "jump" to each appropriate value as the inner for-loop iterates. In the implementation of Strassen, meanwhile, operations are done on

[1] This dimension was chosen to test the efficiency of "padding" in the Strassen implementation. Odd-dimensioned submatrices are padded with one column and one row of zeroes, to allow further subdividing. The dimension of 993 was specifically chosen because it would require multiple iterations of padding – at dim=993, dim=497, dim=249, etc. As the results show, padding slowed the program only very slightly.

relatively small chunks of memory. The CPU may find these smaller blocks of memory more workable in a practical sense, and runtime is reduced due to quicker access within these chunks.

My program utilized a number of optimizations to obtain this degree of efficiency in Strassen's algorithm – in fact, my original, straightforward implementation only barely outperformed the standard. Optimizations largely focused on reducing redundant calculations. The two most significant optimizations are detailed below.

1. At each round of recursion (call to "strassen" in the program code), subvalues p1...p7 were calculated exactly once. Subvalues p1...p5 were passed to each "product submatrix" c0...c03 as necessary. Subvalues p6 and p7 were calculated within c00 and c03, respectively, as these two subvalues were not needed by any other of the c00...c03. This method sped up the program tremendously, when compared to a method wherein the p1...p5 are calculated individually within the c00...c03, leading to numerous redundant calculations.
2. Each sum and difference submatrix needed for p1...p7(such as (A+D), (F-H), etc.) was calculated in one for-loop and allocated as one matrix. This produced much better performance when compared to a naive method of allocating two matrices and finding their sum or difference.

In addition to these major optimizations, I made an effort to optimize for-loops by pulling out redundant calculations and minimizing memory access/allocation. For example, using a temporary variable "sum" (rather than directly incrementing C[i][j] at each iteration) substantially sped up the standard algorithm.

I believe that I managed space effectively in my implementation, however, improvements can be made. Most notably, my program takes the input file and allocates its values into two matrices A and B, stored on the heap. In practice, it may be faster for the overall program (outside of the multiplication algorithm specifically) to perform operations directly on the values of the input file, or else only extracting values as needed on-the-fly. For large matrices where overall runtime is a priority, such a "lazy" method may provide better results.