Harvard University
Computer Science 124

Programming Assignment 2
Due March 25, 2016
Curren Iyer and Paul Lisker

1 Code Structure & Optimization

Our implementation of modified Strassen's algorithm (where rather than a base case of matrices of size 1, a crossover n_0 is defined) was completed using the language C. As such, we had specific calls to mallocs and used pointers throughout our code. Ultimately, this allowed us to minimize the memory allocations in our impementation and recurse nicely within already allocated data structures.

Our code relies on three malloc'ed matrices outside of the first call to the Strassen function. These matrices are M_1 and M_2, the two matrices to be multiplied, and M_3, the answer matrix, where $M_1 M_2 = M_3$. These matrices were malloc'ed within our code using int**.

Our Strassen function took the following eight arguments:

1. Dimension (dimension of the input matrices)
2. int** pointer to M_1
3. int** pointer to M_2
4. A's starting row index
5. A's starting column index
6. B's starting row index
7. B's starting column index
8. int** pointer to M_3

Strassen relies on seven products ($P_1, \ldots, P_7$) which, when multiplied creatively, give us the quadrants of the product of two original matrices. However, to minimize malloc calls within our Strassen function, each P was calculated and then saved to its corresponding quadrant(s) in the answer matrix, adding it or subtracting it according to Strassen's formula. For example, after P_4 was calculated, it was added (using += operator within the appropriate matrix addition helper function) to the bottom left quadrant of the answer matrix as well as the top left quadrant.

Crucially, by calculating each P in successive order and adding it into the corresponding quadrant in the final matrix, the only malloc's necessary were a maximum of three: (1) the matrix memory allocation that represented the P being calculated, (2 3) the two matrix memory allocation for the two matrices that were multiplied together to arrive at P. These malloc's were for matrices of dimension $d/2$, since a quadrant will have half the dimension of the input matrix.

Let's look at an example for the sake of clarity: to calculate P_4, quadrant E was subtracted from G. This was done with the aid of a helper function that subtracted two matrices of dimension d and saved them to a third matrix of size d. However, though the matrices were of size d, rather than copying the quadrant out of the original matrix into memory allocations corresponding to matrices of dimension d, the matrix subtraction helper function that we wrote took as arguments the dimension, the pointers to the three matrices, as well as the indexes that corresponded to the starting row and column (the top left value) of where the matrices to be subtracted are, as well as the indexes corresponding to the starting row and column of where the answer should be saved. As such, by tracking indexes, given matrix M_1, the answer to $B - D$ could be obtained by passing in to the subtraction function the pointer to M_1 twice, but specifying the starting row (0) and column $(0 + \frac{d}{2})$ to refer to quadrant B, and the starting row $(0 + \frac{d}{2})$ and column $(0 + \frac{d}{2})$ to refer to quadrant D. Since we also pass the dimension of the smaller matrices, given the starting indexes and their dimension, adding or subtracting the correct matrix was doable.

Then, to continue the example, the answer to $E - G$, having been saved to a matrix, was then recursively passed into a new call of the Strassen function, with the other matrix passed in being D (or rather, M_1, with the indices adjusted to point to the D quadrant), with P_4 being passed in as the answer matrix.

Thus, in this way, by always passing in the dimension of the matrixes to be multiplied and a pointer to the initial M_1 or M_2 matrices, or one of the three malloc'ed matrices within Strassen, we were able to optimize Strassen to be much faster with fewer malloc calls. (We had originally coded Strassen to have 9 malloc calls: one for each P, and two more sub-matrices that were used to calculate each P.)

1.1 Padding

Because Strassen continually halves the original two input matrices into smaller and smaller sub-problems (i.e. sub-matrices), it makes sense that the algorithm works for all n by n matrices where nd is a power of 2. Therefore, in the case where n is not a power of two, we initially thought it would be best to try to convert a matrix with non-power-of-two dimensions into one that did have power-of-two dimensions, such as by padding the matrix with enough extra rows and columns of zeroes until it had power-of-two dimensions. However, we quickly saw the inefficiency of our algorithm, particularly when n is just 1 more than a power of two - for instance if $= 1025$, we would have to pad the matrix to a dimension of 2048, a very costly memory allocation!

Rather than blindly pad to the next power of two at the cost of poor memory usage, we decided to pad in a slightly different manner. First, given an input dimension n, we would continually take one half of n, rounding up when necessary (via the ceiling function, so either $\frac{n}{2}$ or $\frac{n+1}{2}$), until the value remaining was below our crossover point (in this case 32 - more on how we derived this value later). Then, while this value was below our original dimension we would continue doubling this new rounded up value, with the final product as the dimension of the padded matrix. As an example, let us again consider the case where we multiply matrices with dimension $n = 1025$. Taking half of 1025 yields 512.5, which would be rounded up to 513 due to the ceiling. Taking half again yields 256.5, rounded up to 257; then 128.5 (129), 64.5 (65), 32.5 (33), and finally 17. Since 17 is indeed less than 32, we would then begin doubling 17 until we reach a value greater than or equal to 1025: first to 34, then to 68, then 136, then 272, then 544, and finally 1088, which is greater than 1025. Thus the input matrices would first be padded with zeroes to dimension 1088, and then we would proceed to run Strassen's algorithm. This process ensures that we are more efficient with memory allocation - allocating for a 1088 by 1088 matrix is a much greater improvement over allocating for a 2048 by 2048 one, as in the previous method. Mathematically, the padding function takes approximately $log(\frac{n}{n_0})$ divisions (i.e. we keep halving with ceiling until we reach the crossover n_0) and $log(\frac{n}{n_0})$ multiplications. Therefore, if we assume the cost of arithmetic operations is 1, we can approximate the running time of our padding function to be $2log(\frac{n}{n_0})$.

1.2 Caching

As suggested by the hint in the assignment spec, we sought to optimize our algorithm even further by improving the caching performance of the standard multiplication algorithm. Originally, we implemented the matrix multiplication such that, given two input matrices a and b and a result matrix c, the algorithm iterates through the rows (i) and then columns (j), then incremented a counter (k) (we'll call this order IJK) such that

$$c[i][j] += a[i][k] * b[k][j]$$

However, when we changed the order of the for loops such that we iterated through the counter first, then the rows then columns (we'll call this order KIJ), the algorithm run time decreased by about 80%, from approximately 21000 to just over 4000!

2 Determining the Crossover n_0

The goal of this assignment was to determine, both analytically and empirically, the crossover point at which the dimension of the matrices being multiplied is some n_0 such that it is more efficient to use Strassen's algorithm in combination with standard matrix multiplication rather than just the latter alone. We will begin with the theoretical approach, approximating the crossover point mathematically.

2.1 Analytical Results

To determine the crossover point analytically, we will first have to consider the recurrences of the two matrix multiplication algorithms. We will begin with the standard algorithm. Let us consider the multiplication of two 2 by 2 matrices, as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a*e+b*g & a*f+b*h \\ c*e+d*g & c*f+d*h \end{pmatrix}$$

From the above example, we can describe the running time by keeping track of the number of operations, including multiplications and additions, as follows (where m represents multiplications and a represents additions):

$$\begin{pmatrix} 2m, 1a & 2m, 1a \\ 2m, 1a & 2m, 1a \end{pmatrix}$$

Summing these, we count 8 multiplications and 4 additions. To find the recurrence, we can track the number of multiplications and additions for $n \geq 2$, the results of which are shown in the below table:

n	multiplications	additions
2	$2 \times (2 \times 2) = 8$	$1 \times (2 \times 2) = 4$
3	$3 \times (3 \times 3) = 27$	$2 \times (3 \times 3) = 18$
4	$4 \times (4 \times 4) = 64$	$3 \times (4 \times 4) = 48$
$\vdots$	$\vdots$	$\vdots$
n	$n \times n^2 = n^3$	$(n-1) \times n^2 = n^3 - n^2$

Therefore for n by n matrices, where $n \geq 2$, the standard matrix multiplication involves n^3 multiplications and $n^3 - n^2$ additions. Given that we are making the assumption that arithmetic operations (including multiplication and addition) have a cost of 1, we can approximate the recurrence for standard matrix multiplication as follows:

$$\begin{aligned} T(n) &= 1(n^3) + 1(n^3 - n^2) && (1) \\ &= 2n^3 - n^2 && (2) \end{aligned}$$

Now let us determine the recurrence for Strassen's algorithm. Using a similar method of tracking the number of operations (and assuming that each arithmetic one has an associated cost of 1), we see that Strassen's algorithm uses a divide-and-conquer approach that involves 7 multiplications of sub-matrices with dimension $\frac{n}{2}$ (known as $P_1 \dots P_7$ as seen in the Divide-and-Conquer lecture notes). Additionally, the algorithm involves 10 recombinations (i.e. additions and/or subtractions) to construct these sub-matrices (again with size $\frac{n}{2}$), and 8 more to combine them to form the components of the resulting product matrix. Therefore we can write the recurrence for Strassen's algorithm as follows:

$$\begin{aligned} T(n) &= 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 && (3) \\ &= 7T(\frac{n}{2}) + \frac{9}{2}n^2 && (4) \end{aligned}$$

To find the crossover point, we want to determine the point at which it is more efficient to switch from the standard matrix multiplication to Strassen's algorithm rather than just running the standard one. This will involve considering the scenario where the next recursive call in Strassen will yield the standard algorithm (rather than another call to Strassen) and compare that to simply using the standard algorithm from the beginning. We can describe the recurrence for Strassen's under this condition as follows:

$$T(n) = 7T(\frac{n}{2}) + \frac{9}{2}n^2 \tag{5}$$

Substituting the equation for the standard algorithm for the recursive call:

$$\begin{align} &= 7(2(\frac{n}{2})^3 - (\frac{n}{2})^2) + \frac{9}{2}n^2 \tag{6}\\ &= 7(\frac{1}{4}n^3 - \frac{1}{4}n^2) + \frac{9}{2}n^2 \tag{7}\\ &= \frac{7}{4}n^3 - \frac{7}{4}n^2 + \frac{9}{2}n^2 \tag{8}\\ T(n) &= \frac{7}{4}n^3 + \frac{11}{4}n^2 \tag{9} \end{align}$$

To find the crossover n_0, we set this result equal to the standard algorithm's recurrence to find an intersection point:

$$\begin{align} \frac{7}{4}n^3 + \frac{11}{4}n^2 &= 2n^3 - n^2 \tag{10}\\ \frac{1}{4}n^3 &= \frac{15}{4}n^2 \tag{11}\\ n &= 15 \tag{12}\\ \tag{13} \end{align}$$

Thus we arrive at a crossover value of $n_0 = 15$ (in theory, because Strassen normally deals with powers of two, we could round this up to 16).

Note that this recurrence does not account for all n, as our algorithm involves some padding beforehand to account for certain values of n. In the case where n is a power of two, no adjustments have to be made, as Strassen's algorithm will continually split the matrices into sub-matrices with dimension $\frac{n}{2}$ until it reaches the crossover, at which point it uses the standard algorithm. We must then consider the case where n is not a power of two. This itself splits into two sub-cases: dimension n where the odd factor is less than the crossover, and dimension n where the odd factor is

greater than the crossover.
For the first case, let us consider $n = 52$ and $n_0 = 16$. In accordance with our padding, we take half of n until we reach a value less than n_0; in this case, we reach 13. From there, we double the value while it is less than our original value. In this case, doubling 13 twice again yields 52, meaning there is no additional padding necessary. Therefore the algorithm works with the same amount of overhead as in the case of n being a power of two.

For the second case, let us consider $n = 99$ and $n_0 = 16$. This time, since n is odd, we can't return $\frac{n}{2}$, but instead add 1 before dividing by two (i.e. $\frac{n+1}{2}$, which is equivalent to taking the ceiling of our division), resulting in 50. Repeating this process yields a value of 13, which is less than 15. Doubling accordingly yields 104, meaning we have to pad our input matrices beforehand before running Strassen's algorithm. Because of this additional overhead, we expect the crossover to be slightly larger for these values of n.

2.2 Empirical Results

Determining the optimal crossover point experimentally involves finding the value of n_0 such that the running time of the Strassen/standard hybrid is minimized. As such, when our program was run with flag 6, a for loop was executed where Strassen was run with successively greater crossover points (specifically, each iteration had a crossover that was double that of before). This necessitated the creation of a new Strassen function where the crossover value was passed in to the function so that it could change with every iteration of the for loop. While running, our code used the clock function (from C's time.h library) to track the duration of Strassen's algorithm with the respective crossover point.

After experimentation with various different matrix sizes (from as small as 10 to as large as 2048), we ultimately determined that for the vast majority of them, 32 was the optimal crossover point—particularly, for every dimension that was a power of 2, that was the optimal crossover point. However, for certain matrices (rare, but occasionally) the ideal crossover point was 64, and for others (more rare), the crossover point was 16. However, in these cases, a crossover of 32 was very close in time, too.

A smaller crossover point suggests that Strassen is more efficient, whereas a larger crossover point suggests that standard multiplication is more efficient. In the cases where the crossover point was larger, this could be attributed to the standard algorithm being more efficient than usual thanks to our caching optimization, and/or the Strassen algorithm being slightly slower than usual due to significant padding.