CS 124 Programming Assignment 2: Spring 2021

Your name(s) (up to two): Ziyuan Zhao, Addison Zhang

Collaborators:

No. of late days used on previous psets: 3

No. of late days used after including this pset: 3

Cross-over Point:

Analytical: For the conventional algorithm, multiplying two n by n matrices require $n-1$ additions (or subtractions) and n multiplications (or divisions) of real numbers for each one of the n^2 elements in the resulting matrix, thus it has a cost of $(2n-1)n^2 = 2n^3 - n^2$ operations. For Strassen's algorithm, it takes 7 multiplications on matrices with size $\lceil \frac{n}{2} \rceil * \lceil \frac{n}{2} \rceil$ and 18 additions on matrices with size $\lceil \frac{n}{2} \rceil * \lceil \frac{n}{2} \rceil$. We have the recurrence relation:

$$T(n) = 7T(\lceil \frac{n}{2} \rceil) + 18(\lceil \frac{n}{2} \rceil)^2$$

Case 1: n is even, $\lceil \frac{n}{2} \rceil = \frac{n}{2}$. Then the cross-over happens when the cost of the traditional algorithm is the same as using Strassen's algorithm for only one level and then use the conventional multiplication, namely n_0, the cross-over point, is given by $2n_0^3 - n_0^2 = 7T(\frac{n_0}{2}) + 18(\frac{n_0}{2})^2$, where $T(\frac{n_0}{2})$, the base case, is calculated time used by the conventional algorithm. We solve this

$$2n_0^3 - n_0^2 = 7(2(\frac{n_0}{2})^3 - (\frac{n_0}{2})^2) + 18(\frac{n_0}{2})^2$$

$$2n_0^3 - \frac{7}{4}n_0^3 = -\frac{7}{4}n_0^2 + \frac{18}{4}n_0^2 + n_0^2$$
$$n_0 = 15$$

to get $n_0 = 15$ under the assumption that n is even.

We now prove that this is indeed the crossover point, i.e., for any $n < n_0$, applying Strassen's algorithm uses more time and for any $n > n_0$, applying Strassen's algorithm uses more time too!

Proof: For any $n < n_0$, we rearrange the equations above to get the time difference between applying Strassen for one level before using conventional multiplication and applying the conventional multiplication directly:

$$\Delta T = -\frac{7}{4}n^2 + \frac{18}{4}n^2 + n^2 - 2n^3 + \frac{7}{4}n^3 = \frac{1}{4}n^2(15-n)$$

so it's straightforward to see that when $n < 15$, $\Delta T > 0$. But then by induction, applying Strassen for more than one levels starting below n_0 will uses even more time.

For any $n > n_0 = 15$, the above equation gives $\Delta T < 0$, which says that applying Strassen's algorithm for one level starting from above n_0 will save time compared to direct matrix multiplication, then by induction, we save more time by recursively applying Strassen's algorithm until we hit $n = n_0$, when applying Strassen's is neither wasting or saving time.

Therefore, we conclude that n_0 is indeed the crossover point. The proof also applies to the second case where n is odd, discussed below.

Case 2: n is odd, $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$. Using the same logic as above, we have $2n^3 - n^2 \geq 7T(\frac{n+1}{2}) + 18(\frac{n+1}{2})^2$. Substituting we get:

$$2n^3 - n^2 \geq 7(2(\frac{n+1}{2})^3 - (\frac{n+1}{2})^2) + 18(\frac{n+1}{2})^2$$

$$2n^3 - n^2 \geq \frac{7}{4}(n+1)^3 + \frac{11}{4}(n+1)^2$$
$$n \geq 37.1699$$

We have $n0 = 38$ for n that is odd.

Experiment: To find the cross over point experimentally, we tested various dimensions (n), each with n_0 values from 2 to $\lceil \frac{n}{2} \rceil$, and recorded the n_0 that gave us the fastest run time in the table below. For simplicity, we used random Boolean matrix with equal probability of assigning 0 or 1 to each entry because this speeds up our process of checking the correctness of the outputs. This certainly matters as in reality we may want to store float value entries (although not asked by the assignment) in the matrix (for example, calculate Markov chains), then the running time will be longer than what we present here, but should be bounded by certain constant factor. For completeness we also test random matrices with equal weights of generating $0, -1, 1$. To scan for the crossover point, we used flag $= 1$ in our code.

Interestingly, we have noticed that optimizing the traditional matrix multiplication using cache locality (which we will discuss in the next section of the report) in fact changes the behavior and convergence of the cross-over point especially for larger n. Thus we also record the result obtained before this optimization in the table below (n_0(pre opt) and time(ms) (pre opt)).

n	n_0	time(ms)	n_0 (pre opt)	time(ms) (pre opt)
4	2	0.025	2	0.022
5	3	0.092	3	0.108
8	4	0.019	4	0.018
9	4	0.149	4	0.141
16	8	0.02	8	0.024
17	7	0.179	7	0.183
32	16	0.059	16	0.059
33	15	0.263	12	0.267
64	32	0.131	32	0.227
65	27	0.550	33	0.636
128	64	0.636	50	1.774
129	41	1.518	42	2.275
256	128	3.356	74	11.593
257	77	5.78	76	11.542
512	138	27.556	100	85.453
513	130	31.56	104	86.907
1024	306	197.359	80	614.258
1025	312	222.449	136	582.011

Observations: It seems that as n grows, the optimal cross-over points for even and odd ns that are close to each other also become closer together. Overall, after optimizing, the cross-over point seems to be linear in n as n grows bigger and roughly appears to be $\frac{1}{4}$ of n. However before optimizing, the cross-over point seems to converge as n grows larger, which we found to be very interesting. By convergence, we mean that the numerical value of crossover point stabilizes as the matrix dimension becomes larger. This is in contrast with the behavior we observed for the optimized algorithm, where the crossover point grows proportionally to the matrix dimension, so their ratio instead seems to converge.

We also noted that the above mentioned observations are highly system and machine dependent. The data presented before came from one of our computers. But on another computer we have, where the code overall runs slower, we have observed a convergence of crossover point n_0 to 1/8 of matrix dimension when the dimension tested goes to as large as 4096 and stabilizes after that. Thus, it appears the crossover point may converge eventually at large matrix dimensions.

Finally, we want to remark that the results for the random matrices with entries from $\{0, -1, 1\}$ are very similar to what we discussed above, so the data and observations are omitted to avoid redundancy.

Optimization: We mostly optimized our algorithm in terms of space using realloc for padding and cache locality to improve the speed of the conventional matrix multiplication.

Realloc in Padding: Realloc reduces the space needed when the input matrix has odd dimension. Instead of padding the initial matrix up to dimension $2^k \geq n$, we use realloc to resize the memory block and then fill the additional rows and columns with zeros. This allows us to split matrices into even-dimensional submatrices at every recursive step, and the extra rows and columns of zero will not affect matrix multiplications or combining the resulting blocks into the original matrix as long as we pay attention when looping matrix indices. Yet, a curious and somewhat bizarre incident occurred when we tried to upload the code to Gradescope - the autograder reported that our code gave segmentation fault error. Franklyn has helped us to debug the issue. The code ran well on his PC but didn't work when he ran it on the remote server for grading. Since it will be hard to debug for running on that server, we decided to take out the realloc optimization and use the naive method of padding upto dimension 2^k instead for the Gradescope version of our code. All the rest implementations are the same as the code we used locally for measuring the running time and crossover point - we have also uploaded our code using realloc for reference under the file name `strassen_realloc.c`.

Cache Locality: The inspiration comes from the hint to "loop through the variables in the right order". We experimented with various ns and found that looping in the order (i, k, j) gives the best result. It in fact surprised us a lot when we found that the traditional algorithm for $n = 1024$ took only 4 seconds to run compared to 63 seconds before the algorithm, which really shows the power of this seemingly simple optimization. To understand the reason, we drew schematics of the matrices, and we realize two important points: 1. The result is not changed by changing the order of the loops indexed by i, j, k because we are simply adding the component-wise products to the corresponding location in the product matrix in a different order; 2. When using the order of (i, k, j) and the addition rule of $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$ to compute $C = AB$, for the innermost loop, we are scanning across columns of B at row k and computing the products, but then for the second innermost loop, we scan across columns of A at row i - note that in these two loops we read along rows of matrices A and B. Given that we store our matrices using a pointer to an array of pointers (two-star programmer [1]) to arrays representing rows, we do expect the code

[1] "A rating system for C-programmers. The more indirect your pointers are (i.e. the more "*" before your variables), the higher your reputation will be. No-star C-programmers are virtually non-existent, as virtually all non-trivial programs require use of pointers. Most are one-star programmers. In the old times, one would occasionally

to run faster when visiting matrices elements in row order because they are stored in contiguous memory blocks that can be more efficiently accessed from cache [1].

Triangle in random graphs:

We obtained results below using the modified Strassen code for random graphs with 1024 vertices (this is the case where flag=3 in our code), which are very close to the expected values:

p	Result	Expected
0.01	175	178.433
0.02	1423	1427.464
0.03	4923	4817.682
0.04	11851	11419.714
0.05	22512	22304.128

Note that we used the PCG random number generator [2] instead of the native RNG from C, because as suggested in our report for programming assignment 1, the native RNG might give weird patterns in generated pseudorandoms.

References

[1] M. Clarkson, N. Foster. *Course note - Memory and Locality.* https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec25-locality/lec25.html

[2] M.E. O'Neill. *PCG, A Family of Better Random Number Generators.* https://www.pcg-random.org/.

find a piece of code done by a three-star programmer and shiver with awe. Some people even claimed they'd seen three-star code with function pointers involved, on more than one level of indirection. Sounded as real as UFOs to me."