550 Programming Assignment 2

Brody Vogel, Jinghao Yan

4/20/2018

Introduction

When multiplying matrices, the algorithm given by Strassen is asymptotically faster than the conventional method. This, though, only holds true when multiplying matrices of a certain size or larger; it is more efficient to multiply smaller matrices using the conventional method. In what follows, we try to find that *crossover point*, that is, the point below which it is better to use the conventional algorithm.

Analytic Estimation of Crossover Point

We're looking for a point, c, such that above c it is more efficient to run the regular version of Strassen's algorithm, and below c more efficient to use conventional matrix multiplication. For the sake of estimation, we can assume Strassen's algorithm is always used to multiply two $n \times n$, square matrices, where n is a power of 2. We want our algorithm, then, to run Strassen's algorithm on $n \times n$ matrices until it reaches a point in the recursion where the dimensions of the matrices are $n \leq c$, at which point it should use conventional matrix multiplication to compute matrix products. We can find the equation for such an implementation by unrolling a version of Strassen's recurrence with an arbitrary c; here, we use $c = 4$. (For the case of $n = c = 4$, we must also use the fact that the number of simple arithmetic operations in a convetional multiplication of $n \times n$ matrices is $n^2(2n - 1)$)

$T(4) = 4^2(2(4) - 1) = 16 \times 7 = 112$

The next level, T(8), requires 7 multiplications and 18 additions or subtractions of 4×4 matrices, or:

$T(8) = 7[4^2(2(4) - 1)] + 18 \times 16 = 1072$

and so on:

$T(16) = 7[7[4^2(2(4) - 1)] + 18 \times 16] + 18 \times 64 = 8656$

...

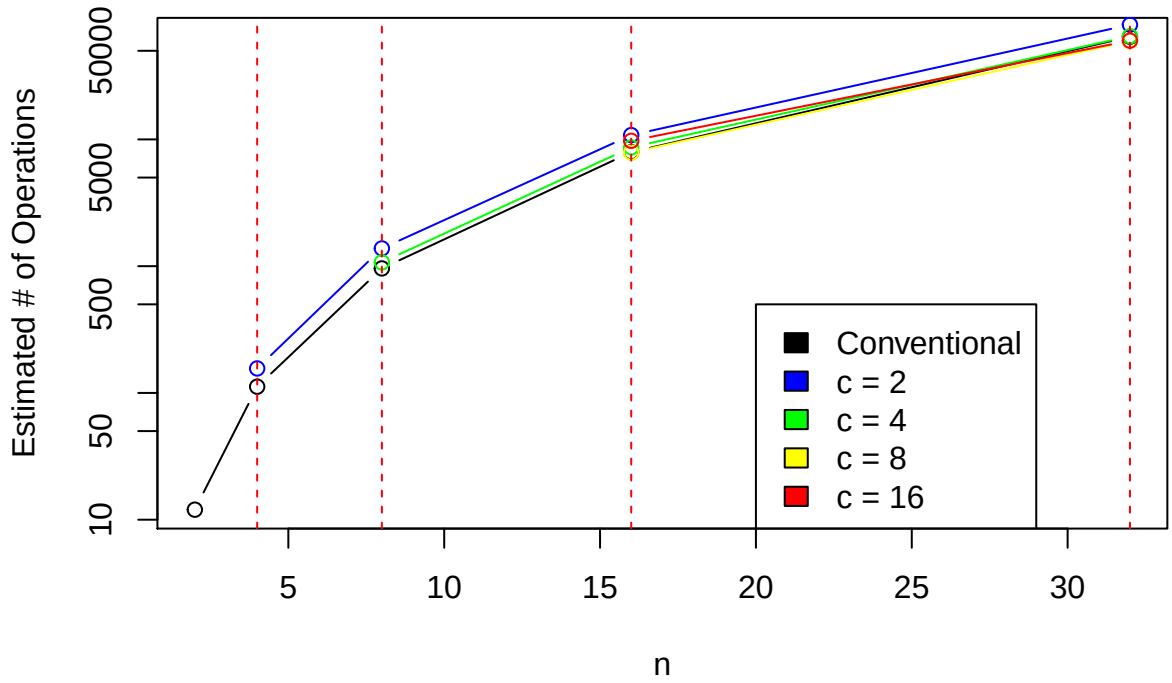
$T(n) = 7^{log_2(\frac{n}{x})}[(4^2(2(4) - 1))] + \sum_{i=1}^{log_2(\frac{n}{x})}(7^{i-1} \times 18 \times (\frac{n}{2^i})^2)$

Or, for c in general:

$T(n) = 7^{log_2(\frac{n}{x})}[(c^2(2(c) - 1))] + \sum_{i=1}^{log_2(\frac{n}{x})}(7^{i-1} \times 18 \times (\frac{n}{2^i})^2)$

(Of course, this formula only holds when both n and x are powers of 2 - we thought that gave a good enough estimation).

Now, to find such a c, we need to find the value such that inserting it into T(n) for c results in our algorithm requiring fewer simple arithmetic operations than the conventional $n^2(2n-1)$, for all values $n > c$ (below which the value from our estimated formula should be undefined). From the plot and dataframe below, we see this is at about $c = 8$ or $c = 16$ (with $c = 8$ and $n = 16$, performing the modified algorithm is essentially the same as performing conventional matrix multiplication on 16×16 matrices - 7872 operations vs 7936; $c = 16$ is therefore a safer cutpoint and the one we went with in our Python program).




```
##   dimension  conv   c_2   c_4   c_8  c_16
## 1         2    12    NA    NA    NA    NA
## 2         4   112   156    NA    NA    NA
## 3         8   960  1380  1072    NA    NA
## 4        16  7936 10812  8656  7872    NA
## 5        32 64512 80292 65200 59712 60160
```

Optimizations

0-Padding

To use Strassen's algorithm on two $n \times n$ matrices where n is odd, we went with a 0-padding approach, and ended up trying two different methods.

The first 0-padding method we tried was adding rows and columns until the $n \times n$ matrices became $m \times m$ matrices, where $n < m = 2^x$ for some x. The idea, here, was that each split in Strassen's recursion would be a clean power of 2, down to the base case. This does technically produce accurate results if the zeroes are then stripped appropriately (which we verified through testing), but it becomes very slow with large n. The periodic massive spikes in runtimes of this implementation of Strassen's algorithm for odd, square matrices gave away its poor performance; it would, for instance, take about 10 seconds for $n \leq 32$, and then jump to about 30 seconds for $32 < n \leq 64$. This is because this implementation adds *up to n-2* extra rows and

columns of 0s to each matrix, which quickly becomes an unmanageable bottleneck as n gets moderately large. Thus, we had to throw this idea out.

We realized shortly after giving up on our previous attempt at 0-padding that, when we transform the $n \times n$ matrices into $m \times m$ matrices inside the recursion, m need not be a power of 2; m must only be *divisible by* 2 (again, verified through testing). The idea this time was that, as long as m is divisible by 2, eventually along the recursion it will hit at or below the cutpoint, which we know our algorithm correctly handles (because it then just performs conventional matrix multiplication). Or, put simply, in contrast to our previous attempt, rather than adding *all* the 0s we could ever need (by making m a power of 2), this implementation only adds 0s when we need them to perform multiplications along the recursion. Before each application of Strassen's algorithm down the recursion tree, then, our implementation checks if n is divisible by 2; if it isn't, the algorithm transforms the matrices into $m \times m$ matrices where $m = n + 1$ by adding one row and column of 0s to each matrix. Now, rather than - as in our previous attempt - adding exponentially more arithmetic operations to our implementation (the majority of which were on 0s), our program adds at most one row and column of 0s at each level of recursion. This cut the runtime down *significantly*; for instance, on an $n \times n$ matrix where $n = 128$, this cut the runtime from roughly 6 minutes down to less than a minute. Ultimately, this was the best we could do - in terms of 0-padding - to ensure our implementation accurately handles $n \times n$ matrices where n is odd.

Matrix Implementation

In our first try at the problem, we designed our *Stras()* and matrix addition and subtraction functions to take dimesion, d, and a string, *line*, as inputs. This worked pretty well and was accurate, but as n grew its memory usage slowed our script almost to a halt. This was because, for each level of the recursion, the algorithms implemented and filled in two matrices from scratch every time a matrix addition, multiplication, or subtraction was needed. So, to multiply two $n \times n$ matrices, the algorithm had to consider and store in memory hundreds of sub-matrices; it also had to consider and store in memory all the necessary strings to make these sub-matrices. This didn't make much of a difference for small-ish n (< 32), but above that the runtime grew exponentially.

To fix this issue, we separted the string-reading from the matrix operations in our program. We designed a new function, *make_matrices()*, that takes in a dimension, d, and a string, *entry*, and returns two $d \times d$ matrices. With this change, when multiplying two $n \times n$ matrices, our program can generate the various sub-matrices it needs along the recursion for multiplications, additions, and subtractions with indexes into the original $n \times n$ matrices. This, then, means the program only needs to remember the original string and two matrices, and a series of indexes that represent the necessary sub-matrices. Once we did this, our runtimes for larger n were massively reduced; this eventually gave us our optimal program performance.

Results

Below are our results, with the crossed-over results denoted with a *. The cutpoint is on the y-axis, the dimensions of the matrices being multiplied on the x-axis, and the elements of the table represent the average time, in seconds, of three runs of Strassen's with those specifications. Our experimental results are roughly in-line with our analytical estimates. From the table, we can see that, for $c = 4$, we never found a crossover point; that is, it was always faster to just use conventional multiplication. We think this is because, by the time the program finishes indexing back to the original string and matrices, it's probably had to add more simple operations than it saves by performing Strassen's. For $c = 8$, our modified implementation of Strassen's algorithm became more efficient only when $n = 512$. And then, with $c = 16$, the crossover didn't technically happen until $n = 64$, but it was very close with $n = 32$, which is exactly where our estimation

predicted it should crossover. Finally, with $c = 32$, the crossover happened at $n = 64$, which is precisely what our estimation predicted.

C	n = 4	n = 8	n = 16	n = 32	n = 64	n = 128	n = 256	n = 355	n = 512
Conv.	.000077	.000390	.002786	.022410	.190265	1.446821	11.555824	32.145350	1:31.322604
4	N/A	.000750	.006025	.050115	.391159	2.619214	18.047293	1:16.651814	2:06.283741
8	N/A	N/A	.004018	.033015	.238634	1.650528	12.265202	44.137502	1.27.042247*
16	N/A	N/A	N/A	.024476	.189257*	1.388178*	9.963639*	31.220998*	1:11.381031*
32	N/A	N/A	N/A	N/A	.176725*	1.296108*	9.323453*	26.365574*	1:06.647091*

So, our program doesn't perform *exactly* like our estimation says it should - it runs a bit slower. We'd expect, with $c = 8$, the crossover point to be very close to 16, but that wasn't necessarily true in our testing. The runtime of our algorithm with $c = 8$ for $n \geq 16$ was *close* to that of the conventional algorithm, but was never actually faster until $n = 512$. With $c = 16$, then, our estimation would lead us to expect our modified implementation of Strassen's to run noticeably faster than the conventional method for $n \geq 32$, but this was only actually true for $n \geq 64$ (although it *was* very close for $n = 32$). Hence, there may be some things that are slowing our program down slightly, but not that we could find.

In conclusion, then, our experimental results didn't match up perfectly with our analytic estimations - but they were close. We think that, if we could cut down on the number of matrix indexes our program requires, the results would be identical; as it stands, these indexes seem to add extra operations to our implementaion of Strassen's. Of course, you don't need to worry about indexing into data structures when running Strassen's by hand (or in estimating the number of arithmetic operations it requires), and so we're fairly confident this accounts for the differences. It would seem, though, that any said difference between our estimations and experimental results become negligible once the cutpoint reaches 16 or 32, which seems appropriate and isn't far from what we'd expect.