Heng Zhou
Hongyang Zheng

Programming Assignment 1

ANLY550
March 29, 2019

Theoretical Analysis

First calculate the runtime for traditional matrix multiplication
The product of two $n*n$ matrices A and B is a $n*n$ matrix C. We know that $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$ for $i, j = 1, 2, ..., n$. For each c_{ij}, there are n multiplications and $n-1$ additions. Since we need to compute $n * n = n^2$ entries for C, we require $n^2 * n$ multiplications and $n^2(n-1)$ additions, then the total operation count is $n^2(n+n-1) = 2n^3 - n^2$.

Second calculate the runtime for Strassen's matrix multiplication
From previous lectures, we know that Strassen's algorithm divides each $n * n$ matrix into 4 $\frac{n}{2} * \frac{n}{2}$ matrices and then performs multiplications/additions/subtractions on each $\frac{n}{2} * \frac{n}{2}$ matrix for each recurrence. Strassen's algorithm uses 7 multiplications and 18 additions/subtractions per recurrence. Since the multiplication is performed on $\frac{n}{2} * \frac{n}{2}$ matrix, the size $= \frac{n}{2}$. The additions/subtractions is also performed on $\frac{n}{2} * \frac{n}{2}$ matrix, since for every entry, we need to perform the additions/subtractions, the operation count is $(\frac{n}{2})^2$. Therefore the total time is $T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$.

Find the cross-over point
From above, we get $T_c(n) = 2n^3 - n^2$ for traditional matrix multiplication and $T_s(n) = 7(\frac{n}{2}) + 18(\frac{n}{2})^2$ for Strassen's algorithm. Let us combine these two equations, then we get the runtime for a modified Strassen's algorithm such that after $n \leq n_0$, we switch from Strassen's algorithm to the traditional matrix multiplication:

$$T(n) = \begin{cases} T(\frac{n}{2}) + 18(\frac{n}{2})^2, & n > n_0 \\ T_c(\frac{n}{2}), & \text{otherwise} \end{cases}$$

Since we only want to switch when Strassen's algorithm starts becoming slower than the traditional way, we need to find the optimal n_0. Let set the two equations equal:

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

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

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

Since $\frac{n}{2}$ requires ceilings we need to discuss different situations for n is even and n is odd.

When n is even
We can directly use the size $\frac{n}{2}$, by solving the above equation, we get: $n_0 = 15$. Therefore, when $n = 16$, the Strassen's algorithm starts to perform better than conventional matrix

multiplication.

When n is odd
Let $n = 2k+1$, and then plug $2k+1$ into above equation, then we get $k = 18.085$, $n_0 = 37.17$, therefore $n_0 = 38$. As a result, when $n = 39$, the Strassen's algorithm starts to perform better than conventional matrix multiplication.

Experimental Analysis

Description of functions in code
The code is shown in $strassen - product.py$.

The code with time calculation is shown in $strassen - time.py$.

$strassen - even$ is a function to determine whether the dimension of passed matrix is even, if it is, then calculate matrix multiplication using variant of Strassen's algorithm in this function. In this function ,we initialize each new sub-matrices with size $\frac{n}{2}$ and divide each matrix into 4 sub-matrices. They are $a_{11}, a_{12}, a_{21}, a_{22}, b_{11}, b_{12}, b_{21}, b_{22}$. Then we can compute p_1 to p_7, the result matrix C can be obtained, which equals A times B.

$strassen - odd$ is a function to pad one row and one column with 0s to the passed matrix when its dimension is odd, then return to even function to calculate matrix multiplication.

$traditional - matrix - product$ is a function used to calculate matrix multiplication by using conventional method, which is called when encountered CrossOverPoint.

The helper *add* and *subtract* are used to compute addition and subtraction between matrices.

Deal with odd dimension matrix
When the number of dimension is the power of 2, we can easily apply variant of Strassen's algorithm directly, but when it is not, we must have to change it into a matrix with power of 2 dimensions. At first, we tried to pad the matrix with 0s to make it has nearest power of 2 dimensions. For example, if we have a matrix with 131×131, then we pad 0s to make it a 256×256 matrix, which is 2^8. However, this method is not efficient enough, therefore, we came up with another method. We first check if the dimension of passed matrix is even instead of the power of two, if it is, then perform variant of Strassen's algorithm; if it is not, we pad only one row and one column with 0s to make it is even. Now, suppose we have even dimension matrices now, but the dimensions are not power of two, which means it would become odd at some point. So we modified the code to make it recursively check and

pad, which guaranteed that we can calculate the multiplications of matrices with variant of Strassen's algorithm whenever encountered odd or even.

Choice of input number and dimension

In order to test our algorithm, we first generated the input integers for the matrix in the range $[-10, 10]$. (We also tried some other types of number such as float numbers or some very large numbers, but their runtime is similar, so the type of numbers does not really matter here.) We set CrossOverPoint from 1 to dimension of each original matrix, so the whole algorithm would run at every value of CrossOverPoint. When the dimension of sub-matrix is below CrossOverPoint, the algorithm would calculate the multiplication of sub-matrices by conventional algorithm; when it is above, it would keep using Strassen's algorithm. Therefore the whole algorithm will recursively check this condition and perform the combination of conventional algorithm and Strassen's perfectly. We didn't try CrossOverPoint with value larger than the dimension of original matrix, since under this situation, the whole algorithm would run conventional algorithm directly, which couldn't detect the real optimal point and give us a valuable conclusion. In this problem, we tried different dimensions, both even and odd, power of 2 and not power of 2 from $[1, 300]$, they are $[64, 105, 126, 128, 129, 256]$.

Results under each dimension

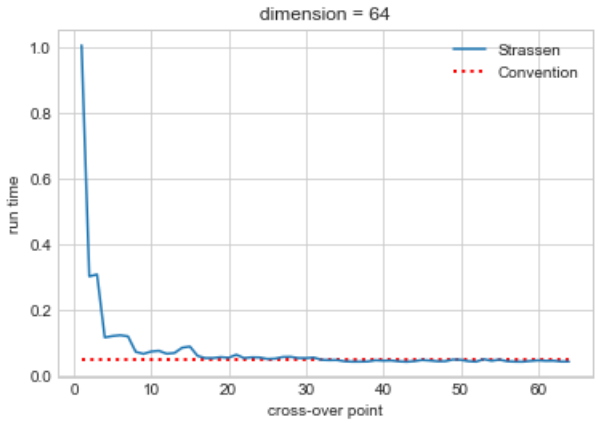

Figure 1: N=64

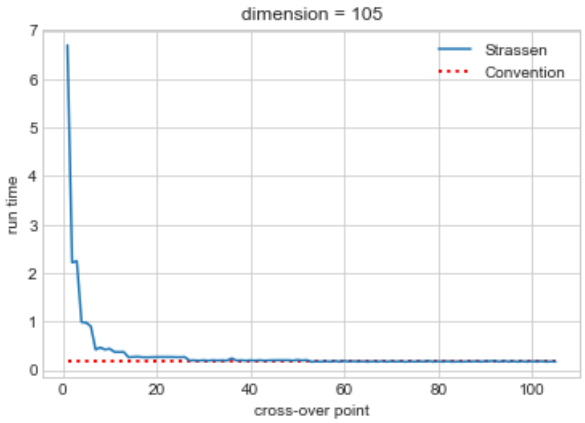

Figure 2: N=105

We tried 6 dimensions in the program, recording run time under both variant of Strassen's algorithm with different CrossOverPoint and conventional algorithm. The figures here visualized the process of each dimension. The blue line indicates the run time for variant of Strassen's and red line represents the run time only using conventional algorithm.

As we can see from the graph, at first the run time decreases sharply. This is because at

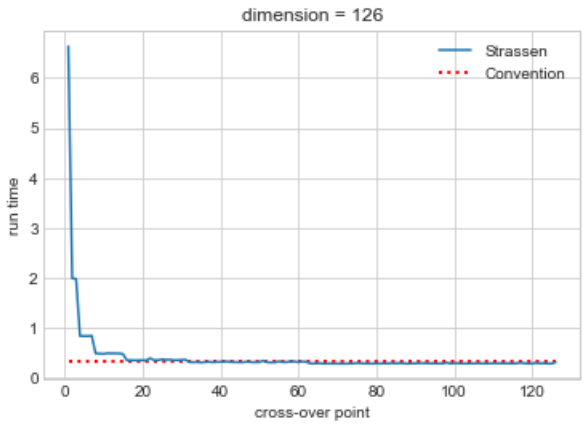

Figure 3: N=126

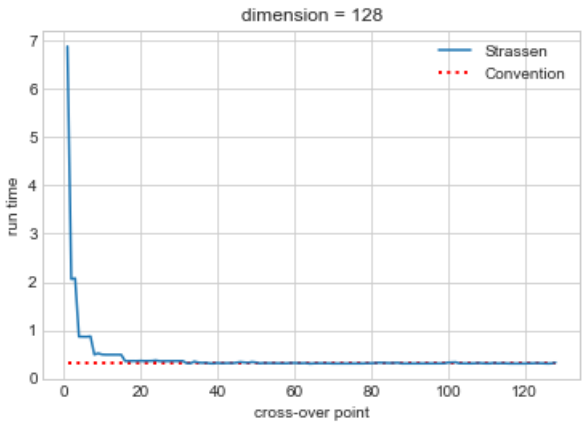

Figure 4: N=128

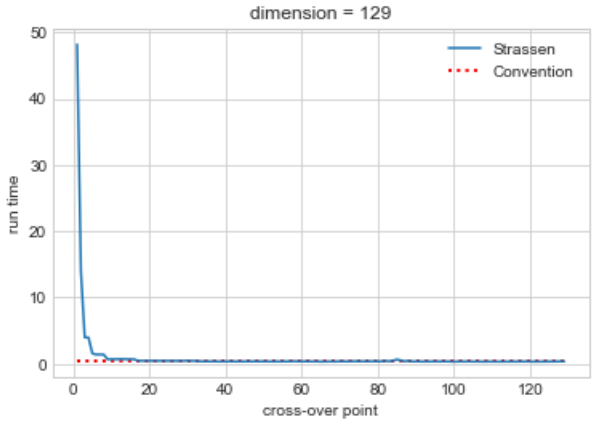

Figure 5: N=129

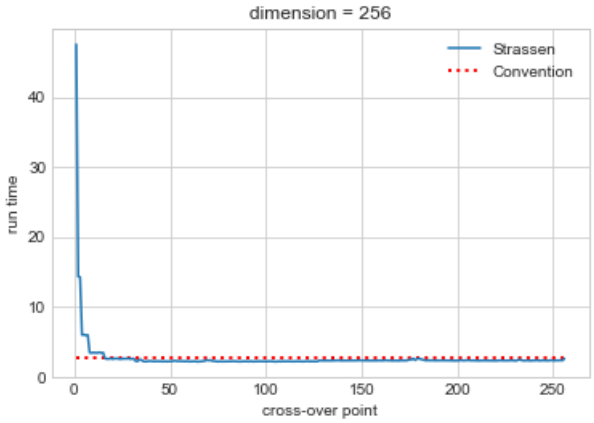

Figure 6: N=256

Dimension	CrossOverPoint Theoretical	Minimum Runtime	CrossOverPoint Experimental
64	15	0.04027009	43
105	38	0.172886133	65
126	15	0.282181024	72
128	15	0.306087019	64
129	38	0.31235671	107
256	15	2.12689805	105

Figure 7: Results Table

first we set the cross-over point very low, and then the algorithm does more recurrences of Strassen's algorithm than the ideal situation, therefore it causes longer run time. As the cross-over point increases more closely to the dimension of the matrix, the run time does not

change much and close to the run time of using conventional method, because the variant of Strassen's algorithm will turn into the conventional method at the very beginning.

Figure.7 is a table containing 4 columns, column one represents dimensions; column two indicates theoretical CrossOverPoint given by analysis in the first section; column three contains minimum run time for each dimension; and column four represents the CrossOverPoint that gives minimum run time. Our CrossOverPoint in program for dimensions $64, 105, 126, 128, 129, 256$ are $n = 43, n = 65, n = 64, n = 107, n = 105$, which are significantly different from theoretical CrossOverPoint from analysis before (We first assumed that we would get points around 16 and 38). In our implementation, taking the example for dimension 128 and 129, the odd case runs just a little slower than the even case. Since they have similar size of dimension, size does not matter a lot here. The difference of run time is because every time the odd case encountered odd dimension, it has to pad first and then runs as an even case, until reaches the CrossOverPoint. The padding procedure is also the reason for the CrossOverPoint of n=129 is much higher than n=128, because when CrossOverPoint is high, odd case would run with fewer steps of padding and run in conventional algorithm earlier, which decreases run time. Now, take a look at n=126 and n=128, both of them have even dimensions, 128 is the power of 2 while 126 is not. CrossOverPoint of n=126 is 72, much higher than n=128, which is the same as we expected. When 126/2, the dimension becomes odd number 63 and needs to pad to make it eligible for Stassen's, so it also needs to run in conventional algorithm earlier. Our highest run time is when dimension is 256, it is normal because when size of dimension increases, the steps of dividing, addition, subtraction and grouping results would increase significantly.

Conclusion

From previous parts, we can conclude that the type of input number(integer/float/large number) does not matter significantly, while the size of the matrix dimension can have impact on the run time and CrossOverPoint. For example, whether the dimension is even; if it is even, whether it is the power of 2. Our CrossOverPoint from experimental results are much larger than theoretical ones. It may depend on various factors like the implementation methods, the hardware, memory access speed, cache consideration and hyper-threading considerations. What is more, in theory, we assume that the addition/subtracting/multiplying/dividing or other single arithmetical operation cost 1, but in reality, it may be different and dynamic.

The difficulties arose during our investigation is that at first, we want to calculate the run time based on the average of five trails. However, when we implemented this approach, we found that it will costs too much time especially when the dimension becomes larger than 128. So we didn't adopt this method at the end for this assignment.