Roya Salei

October 29, 2020

CS 3310

Project 1 : Matrix multiplication

Theory

In this project, I examined the running time of 3 different algorithms for performing matrix multiplication and compared them to their theorical time complexity estimation. Theoretically, programs with similar input and output but a different number of operations do not perform equally in terms of time and space complexity. Programs with least number of basic operations must perform faster than programs with larger number of basic operations.

Procedure

The following hardware and software were used:

1. A Dell laptop with:
   a. Windows 10 operating system
   b. 12 GB of RAM (11.9 GB usable)
   c. 917 GB hard drive with 347 GB free strorage
2. Visual Studio IDE
3. C++ programming language

Experiment Setup:

1. Data Set:
   Per instruction for this project, all matrices were set to be square (n x n) with sizes varying in powers of two. The sizes were explicitly set to include the following:
   n = 2, 4, 8, 16, 32, 64, 256
   To remove any discrepancies of the input data, a set of random matrices with different sizes were generated using a function (`matrix_builder()`*) for all given sizes (n). Data was saved into a CSV file called matrices.csv. The random numbers were generated between 0 and 100. This function only ran one time to generate the data.

Then, another function (`read_matrices_from_file()`) was implemented to read the CSV file and fill two identical matrices, for every set of data. Furthermore, those matrices and the third matrix (which was used for the result of multiplication) were passed to our three experimental algorithms as arguments.

2. Implementation

   Three different algorithms were implemented using three different functions with the following function headers:

   ```
   void multiply_iterative(int** a, int** b, int** r, int n);
   void multiply_divide_conquer(int** a, int** b, int** r, int n);
   void multiply_strassen(int** a, int** b, int** r, int n);
   ```

   Three generated matrices from the CSV file (as mentioned in Data Set section, above) were passed to all three algorithms including the size of that matrix. (Please see main.cpp)

   Additionally, to remove the long compilation of the header files, a file for all headers were created in a separate file (stdafx.h) and was included in the main.cpp:

   ```
   include "stdafx.h"
   ```

3. Testing Plans

   To increase the statistical power of the analysis, each algorithm ran 20 times for every single data set and recorded the actual time in another CSV file (times.csv). To represent the running time of each algorithm, the average run time over 20 repetitions were used.

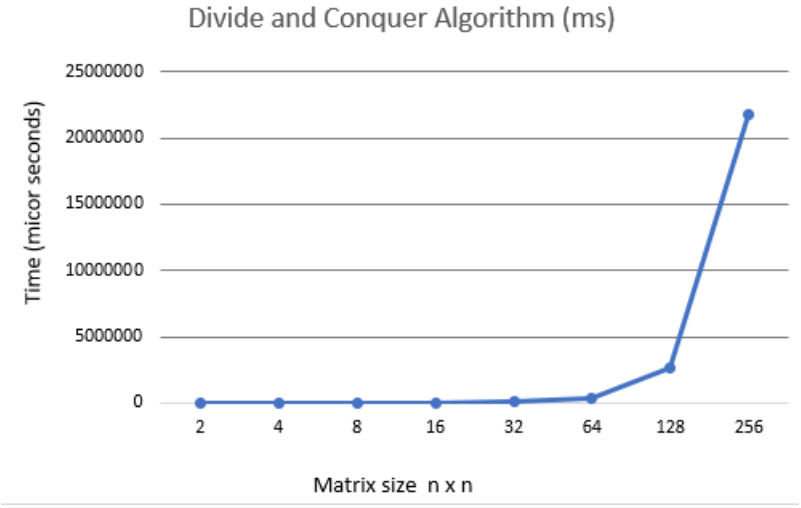
Results

Table 1 shows the result of the running time (converted from microseconds to seconds) for all three algorithms. As evident in table 1, the iterative algorithm is much faster than the other two algorithms which utilize the divide and conquer approach to further chunk the matrices into smaller matrices.

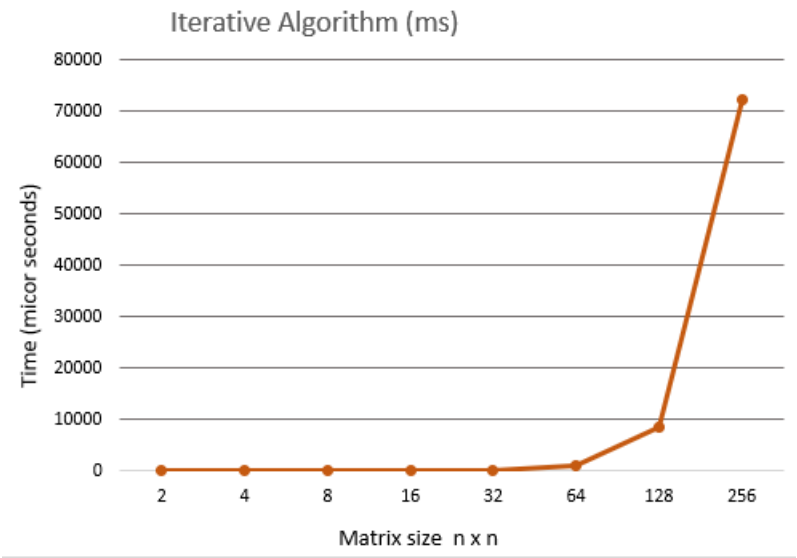
However, between Strassen algorithm and Divide and Conquer algorithm, as expected the Strassen algorithm clearly shows a better performance, simply because Strassen algorithm has one less recursive call.

Table 1Algorithm average run time for matrices with different sizes

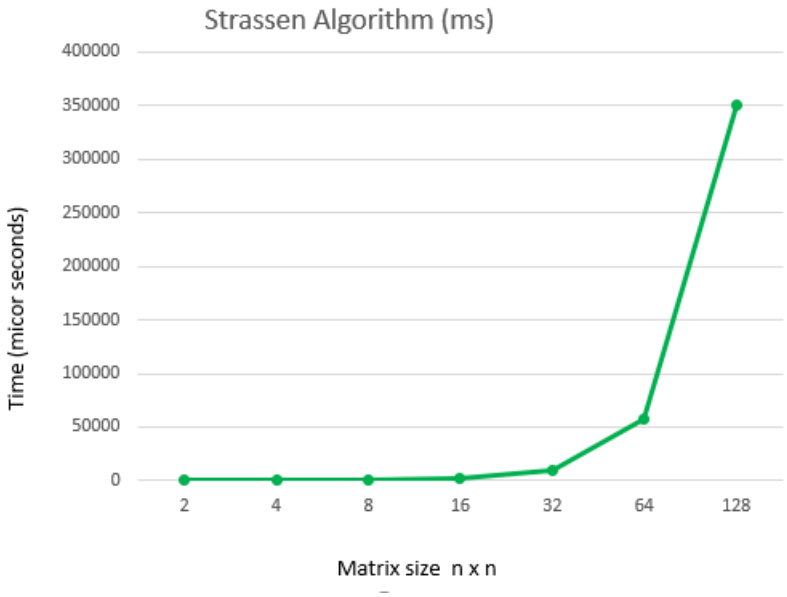
Size n	Iterative Algorithm (s)	Strassen Algorithm (s)	Divide and Conquer Algorithm (s)
2	0.00000115	0.000002	0.00001425
4	0.00000105	0.00007325	0.00011735
8	0.00000735	0.00049675	0.0007498
16	0.0000487	0.001420375	0.00933285
32	0.0002756	0.009645625	0.04816985
64	0.00109955	0.057997725	0.3264064
128	0.00872185	0.349915525	2.659183
256	0.0723601	computer crashed	21.76907



Graph 1 Shows running time (micor second) vs matrix size for Divide and Conquer Algorithm



Graph 2 Shows running time (micor-second) vs matrix size for Iterative Algorithm



Graph 3 Shows running time (micor-second) vs matrix size for Strassen Algorithm

Constraints of the project

Due to the computing power of my laptop and hard drive fragmentation unfortunately I couldn't perform higher matrix multiplication and computer continuously crashed even at 256 x256 matrices.

Conclusion

As shown in table 2, Strassen algorithm uses 7 recursive calls compared to Divide and Conquer algorithm which uses 8 recursive calls.

Table 2: Recurrence Relation for Strassen vs Divide and Conquer Algorithm

Divide and Conquer Algorithm	T(N) = 8 T ($n/2$) + O (n ^2)
Strassen Algorithm	T(N) = 7 T ($n/2$) + O (n ^2)

Even though Strassen algorithm performance, theoretically beats the other two algorithm (O ($n^{\lg 7}$) ≈ O ($n^{2.81}$) compared to O(n^3) for the Iterative or divide and conquer algorithm), there are many other factors in play when performing such algorithms. Such as space complexity and the demand that a divide and conquer places on computer memory. Divide and conquer is implemented using recursion. Because it uses stack and memory intensively it will become dependent on to the system that is being used. In the case of my experiment my computer was extremely limited in such resources and as a result it started failing even at low matrix sizes such as 256.

Glossary

* I did not use CamelCase for function names, for two reasons: it was not required by the project instructions and for me the names that include underscore (_) are more readable.