Richard Idigo

Professor Young

CS 331

May 15th, 2015

**Project 1: Comparing Matrix Multiplication Algorithms**

Data Sets, Test Strategies and Explanation of Results

**Test Strategies**- While deciding on how to test and analyze all the algorithms against each other, I decided to have the algorithms continuously check constant data, in other words every element in the matrices were generated with the same exact technique. At first I thought of using a random generator to generate the values, but I thought that this may change the results of the experiment because the data wasn’t constant. I also wanted each algorithm to have the opportunity to execute on the same data that the others did. While testing I incorporated many print statements to check logic and correctness.

**Data Sets-** Because the project deals with matrix multiplication, I thought the most fitting data structure would be a two-dimensional array. I felt that out of all the data structures I could use, two-dimensional array mimicked the characteristics of a matrix the best. Following your directions, the values for the nxn matrices are powers of 2(2, 4, 8, 16...)After obtaining the row and column size from the user and creating two matrices, I populated Matrix A to have the first index contains 0 and the each element after increases by one. For Matrix b the first element is 1 and every element after again increases by 1. I did this so that each element in the arrays would be unique and for simple error detection for the multiplication operation and others.

For the project it was required to use matrix sizes 2^n. So I used array ranging from 4x4, 8x8, and 64x64 all the way through matrices of size 2048x2048, 211. When I tried to push for greater n values, I found that after compilation Eclipse would report an “out of memory” error of exception once it tried to execute the program. This particularly happened when I tried to incorporate matrices of size 16384 for the classic method. I reduced by a power and tried 8192, this brought about the same message. Finally reduced it to another power of 2 and tried 4096, this however brought about weird behavior from Eclipse. The main console did not appear to be doing anything as if it didn’t compile anything. On the contrary, when I checked the current processes running on my notebook I noticed that for at least 15 minutes Java ™ Platform was averaging about 25.6% of the overall CPU use. Also using the task manager tool I was able to see that Eclipse Luna was taking 51% of the memory, 526.5MB. After 20 minutes I thought about terminating the program but left it alone, eventually Eclipse started to display calculations and a time was finally given. The graph attached shows the behavior of the algorithms visually.

**Results and Explanation-** From running several tests on the algorithms with the alternating sizes of arrays, the result of the project proved something that was initially quite surprising. I thought the Strassen algorithm would be the fastest of the three for every size, but as my results and graphs show it gets less and less efficient when the size of an array is very large. Also I thought that the divide and conquer method would be faster than the classic; however it is the classic that is faster. As you can see in the graph attached, classic is a little faster than divide and conquer. From further research I found this is due to the fact that Strassens best performance takes place with “smaller” large arrays. However, when an array is very large, Strassens lessens in comparison to other algorithms for matrix multiplication.

Theoretical Complexity Comparisons

Theoretically the complexities for the three different algorithms for matrix multiplication used in this program differ from each other, with the exception of classical multiplication and the divide and conquer method.

The classical method uses a total of three for loops to conduct the multiplication of the matrices. Considering the three for loops it is intuitive why the complexity is O(n3). The divide and conquer method implemented first partitions each array into fourths, and then executes the multiplication with the newly created arrays among each other which then creates the four quadrants of the resulting array.

Though there is an initial splitting of the arrays, the complexity still turns out to be O(n3).

The Strassen's algorithm differs from both of the previous mentioned; however the difference is very slight. The complexity for the method is roughly O(n2.81). The slight speed difference is due to the recursion it incorporates.

Classical Matrix Multiplication Vs Divide-and-Conquer Matrix Multiplication

The classic method of matrix multiplication can be described by the following expression;

What this translates into is for the matrix A and matrix B with sub-elements Aij and, when multiplied together they create a matrix C with sub-elements Cij where Aij X Bij = Cij (for every element in matrix A and every element in matrix B). So using the two-dimensional arrays A and B, for loops are used to cycle through each element in both, and a for loop two keep track of place in the resulting array to prevent any out of bounds errors. I created a variable in one of the for loops to hold the value of Aij x Bij for a given iteration. The sum is then stored in the appropriate Cij in the matrix C. Using a total of for loops. Producing the complexity of *O(n3)*

The divide-and-conquer method can be visualized as the following diagram;

This diagram explains that when given a matrix B and matrix A with the intention of conducting multiplication between them, the result creates a matrix C that is the same size as the two matrices.

However, if we split matrix A and matrix B into fourths as shown by the diagram, expressions consisting of multiplication and addition can be used to find the sub-matrices of matrix C. For instance

C11(the first quadrant of the matrix) can be found by multiplying A11 and B11, then adding that result to the product of A12 and B21. To capture the behavior of this algorithm I first split arrays A and B using a set of for loops that copy each element selectively and places it in its corresponding array(A11, B11, etc). After the original arrays are partitioned, I created new arrays corresponding to the resulting matrix C (C11, C12). For each quadrant of C, I followed the expressions stated above using a multiplication and addition method I created for ease. Producing the complexity of *O(n3)*. Unlike the Strassen algorithm, neither the classic method nor divide and conquer method use recursion for the computation.

Strength and Constraints of Your Work

This project was created and tested with the following hardware and software;

|  |  |
| --- | --- |
| **Hardware**  Toshiba notebook-  -Intel(R) core i3-3120M CPU processor @ 2.50ghz  - ram 4GB (3.88 GB usable)  - 64-bit Operating system type  - 750GB Hard Drive (545 GB usable) | **Software**  -Operating System Windows 8.1  -IDE Eclipse Luna Version: Luna Service Release 1 (4.4.1)  Build id: 20140925-1800  Programming Language: Java |

Program Correctness

In terms of optimization I could of course start with optimizing the algorithms. According to an article I read, the Strassen algorithm’s efficiency depend on the implementation as well as the hardware used. So with that I could have also used better hardware such as an i5 Intel processor instead, or an architecture with more RAM and other hardware components. I also could have tried programming in another language such as Python as it is less “restrictive” than Java.

As far as time is considered I could have allowed a bigger window for the algorithms to conduct. For this experiment I made anywhere from 25 mins to 30 mins the cut-off point, however, I could have instead checked the algorithms with a window of maybe an hour. Also in my program I after I found the appropiate sub-matrices of C, I did not then join them back together. Meaning I didnt not fuse C11, C12, C21 and C22 into the matrix C. Essentially the answer is still found but just not reassembled.