COLLEGE OF ENGINEERING TRIVANDRUM

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

CST 306
ALGORITHM ANALYSIS AND DESIGN
Matrix Multiplication Analysis
Assignment - 1

Demel C John
CSE 24 Batch 2
Roll no 22
TVE20CS042

Matrix Multiplication Analysis

1 Task

Implement three algorithms: – Iterative Matrix Multiplication algorithm. – Divide-and-Conquer Matrix Multiplication algorithm. – Strassen's Algorithm. Assume that all matrix multiplication operations will take two square n×n matrices as input. In implementing these algorithms you are only allowed to use the following language features: – Integer variables – Integer arrays of any dimensionality. – Addition and subtraction of integers. – Multiplication of integers. – Control structures of your programming language (loops and conditionals) as well as function calls and user-defined functions.

You are not allowed to use any matrix arithmetics operations or any vector arith- metics operations. In addition, implement a testing mechanism that allows you to collect the information about the run time of your implementation. The testing mechanism, at a minimum, shall consist of the following:

– A procedure for generating a random n × n matrix with $n^2 integervaluesinit$.

– Archive any generated matrices (this is needed to make sure that your algorithms are tested on the same set of matrix multiplication operations).

– Find the execution time of each of the matrix multiplication algorithms you im- plement.

– A procedure that takes as input a set of experimental parameters, performs the desired run-time experiment (see below), and collects the run-time data.

– Save the results of the experiment in a way that can be used for further analysis Performing Experiments

With the three implementations of the Matrix Multiplication algorithms, and with the test mechanism you build to test them, you perform a run-time experiment designed to understand how the performance of the algorithms changes with the increase in the input size.

The experiment should have the following parameters:

(a) Matrix sizes. Select between 10 and 20 different matrix sizes starting with some- thing relatively small (e.g., n = 10, and ending with a reasonably large matrix size that still can fit comfortably in RAM (e.g., n = 1000). Matrix sizes is the first parameter of your experiment.

(b) Repetitions. For each matrix size, you generate a number of pairs of matrices to be multiplied using your algorithm implementations. This is controlled by the second parameter called Repeats. We want Repeats > 1, because we want to convince ourselves that the runtime of your implementation observed for a specific pair of matrices of size n×n is consistent with the run-times observed on all other pairs of matrices of the same size.

(c) Measurement. For each matrix size and for each repetition step, generate two matrices of the appropriate size and compute their product using all three of your methods. Find the running time for each computation. For each matrix size, aggre- gate the results by computing the average running time, and standard deviation of the running time for each of the three algorithm implementations.

(d) Reporting. For each algorithm, you should obtain a set of average run-times for each input size considered. Plot these runtime numbers vs input size. If possible, fit a curve to them to see if they follow the expected algorithm run-times.

(e) Analysis. Observe the results and answer the following questions. – What is the observed run-time behavior for each algorithm? – Which algorithm is the fastest in your experiment? Which is the slowest? – What are the asymptotic trends for each algorithm? Can you observe or predict if Strassen's Algorithm overtakes one or both of the remaining ones at some point?

(f) Report. Prepare a properly formatted, and presentable report describing your work. The report shall contain the precise description of the experiment your im- plemented, and the experimental results both in tabular and in graphical form, and it shall contain your analysis.

2 Matrix Multiplication Algorithms

2.1 Iterative method

The iterative method, also known as the traditional method or the naive method, is a straightforward approach to matrix multiplication. In this method, each element of the resulting matrix is computed by multiplying the corresponding elements of the input matrices and summing them up.The iterative method is simple to implement and easy to understand.However, it can be computationally expensive, especially for large matrices, as it involves a time complexity of O(`n^3`), where n is the dimension of the matrices.This method performs all the necessary multiplications and additions explicitly, without any optimizations or shortcuts.

2.2 Divide and Conquer method

The divide and conquer method is a widely used algorithmic technique, and it is also applicable to matrix multiplication. In the context of matrix multiplication, the divide and conquer approach involves breaking down the given matrices into smaller submatrices, performing multiplications on these submatrices recursively, and then combining the results to obtain the final matrix product.One advantage of the divide and conquer approach is that it can lead to improved algorithmic efficiency, especially for large matrices. By breaking the problem into smaller subproblems and leveraging the inherent parallelism, divide and conquer algorithms can potentially reduce the time complexity of matrix multiplication. However, it is important to note that the actual performance improvement may vary depending on the specific implementation and the characteristics of the matrices involved.It involves a time complexity of O(`n^3`).

2.3 Strassen's method

Strassen's method is an innovative algorithm for matrix multiplication that reduces the number of required scalar multiplications compared to traditional methods. It is based on the principle of divide and conquer, but with a unique twist.The key idea behind Strassen's method is to divide the input matrices into smaller submatrices of equal size, typically of order n/2. These submatrices are then used to construct new matrices through a set of mathematical operations, which are combined to obtain the final matrix product.The breakthrough innovation of Strassen's method lies in reducing the number of scalar multiplications required. While the standard matrix multiplication algorithm involves eight scalar multiplications for each element of the resulting matrix, Strassen's method only requires seven. This reduction in scalar multiplications contributes to improved computational efficiency, especially for large matrices.However, it is important to note that Strassen's method also introduces additional matrix additions and subtractions, which can impact the overall performance. For small matrices, the overhead of these additional operations may outweigh the benefits gained from the reduced scalar multiplications. As a result, Strassen's method is typically more efficient for larger matrices, where the reduction in scalar multiplications becomes more significant.Tt involves a time complexity less than O(`n^3`).

3 Experiment Procedure

The procedure involved a main driver function which captures the time of three multiplication algorithm and stores in the csv file .The result stored in the data.csv file is fetched and analysis is done. The main driver function calls iterative method which iteratively calculates the product matrix using brute force and gives the run time to main function. The divide and conquer uses divide and conquer method to find the product by dividing each matrix to 4 matrices and find product. The strassen's method uses divide and conquer approach and find using a tricky method which helps to reduce the time complexity.The time of each algorithm is fetched to testing function and plot points in the graph and compares the curves. Also the mean, variance and standard deviation is found to analysis the multiplication.

4 Result

The result of the analysis shows that the time taken by iterative method is much smaller than other two methods. This is due to the high overhead by function calls.The divide and conquer method uses many functions which increases the overhead due to jumping of control.The iterative if uses functions will get comparable values. The divide and conquer and strassen method have comparable values. The strassen pad with zeroes to bring the matrix to order 2 power n to 2 power n which makes the strassen method to have a function which appears as a step by step function. From the graph also, the change in time can be observed.

4.1 Subresults

The complexity of divide and conquer is almost like O(`n^3`) and the time complexity of strassen is less in the points where there is same n's vales for divide and conquer, so time complexity is less than that of divide and conquer. Time complexity of iterative matrix appears low.So iterative is fastest and divide and conquer is slowest.From the observed trend, strassen will beat divide and conquer when n is between 150 to 200.

Method	Mean	Variance	Standard Deviation
Iterative	7.157	402.805	20.070
Divide and Conquer	56.011	26867.172	163.912
Strassen	69.098	26254.683	162.033

n	iterativeTime	divConTime	strassenTime
2	1.550000160932541e-05	4.2599996959324926e-05	8.84000000951346e-05
2	2.5499997718725353e-05	6.779999966965988e-05	0.0001123000018258579
2	1.3100001524435356e-05	3.9200000173877925e-05	0.0001027999696649611
2	1.3999997463542968e-05	5.610000152955763e-05	8.969999908003956e-05
2	1.979999797185883e-05	5.6499997299397364e-05	7.439999899361283e-05
2	1.0399999155197293e-05	8.620000153314322e-05	7.330000153160654e-05
4	6.420000136131421e-05	0.0003359000002092216	0.00041820000114967115
4	6.7299999500392e-05	0.00037480000173673034	0.0004867999996349681
4	7.479999840143137e-05	0.0003913999971700832	0.0004195000001345761
4	9.59999997576233e-05	0.0003423999987717252	0.00043559999903663993
4	0.00011799999992945231	0.00040169999920181	0.0004893000004813075
4	0.000139899999339832	0.00044549999802256934	0.00046380000276258215
8	0.0003071000028285198	0.0022791999981564004	0.0029495999988284893
8	0.00031549999766866677	0.002248400000098627	0.002976799998577917
15	0.0018799999998009298	0.01359519999823533	0.02237679999619834
15	0.0018744000008155126	0.013837300000886898	0.026595600000291597
15	0.0018321999996260274	0.015810599998076214	0.023211599997011945
15	0.00181220000013127	0.014313800002128119	0.02274109999780194
16	0.002265600000100676	0.020637599998735823	0.02391200000056415
16	0.0028511000018625055	0.01951050000207033	0.022506799999973737
27	0.012244099998497404	0.07943199999863282	0.17291329999716254
27	0.011612500002229353	0.07927619999827584	0.1597397000005003
27	0.01054929999736487	0.07374650000201655	0.1752572000004875
27	0.01109569999971427	0.07872800000041025	0.16916689999925438
32	0.017681099998299032	0.1767809999983001	0.1774586999999883
32	0.017737699999997858	0.20931760000166832	0.1686431000016455
41	0.040558699998655356	0.24774150000303052	1.1958190000004834
41	0.03780939999705879	0.23593090000213124	1.1942983000008098
42	0.045311099998798454	0.27570360000026994	1.302647699998488
42	0.045847500001400476	0.3006901000007929	1.2154970000010508
64	0.19181559999924502	1.3683511999988696	1.2836276999987604
64	0.15202249999856576	1.2153894000002765	1.2556041000025289
71	0.21514839999872493	1.5253824999999779	9.274046599999565
71	0.19258609999815235	1.5031350999997812	9.589613500000269
123	1.1999259999975038	18.36726009999984	17.700281800000084
123	2.5509918999996444	9.994183700000576	8.459497400002874
128	1.242340900000272	10.28295000000071	8.47225520000211
128	1.228714700002456	10.21605609999824	8.13409369999863
173	2.9640620999998646	17.95296690000032	57.72717150000244
173	2.964736999998422	17.728057099997386	57.9146662000021
152	2.0484934999985853	13.05601260000185	58.14310420000038
152	2.0150214999994205	13.137088700001186	58.39566489999925
256	9.568696999998792	117.87967040000149	74.17168380000294
256	9.807013700003154	93.14452899999742	58.756607400002395
379	31.665183400000387	241.8349199000004	544.8924805999995
379	41.28531699999803	276.5534828000018	518.911039999999
355	27.087586600002396	173.52046070000142	474.1569023999982
355	31.13967650000268	196.368979400002	519.6904472999995
512	93.79022020000048	693.0294885000003	432.8167016999978
512	96.27824959999998	889.9918118400016	529.0640416999995

4.2 Plot

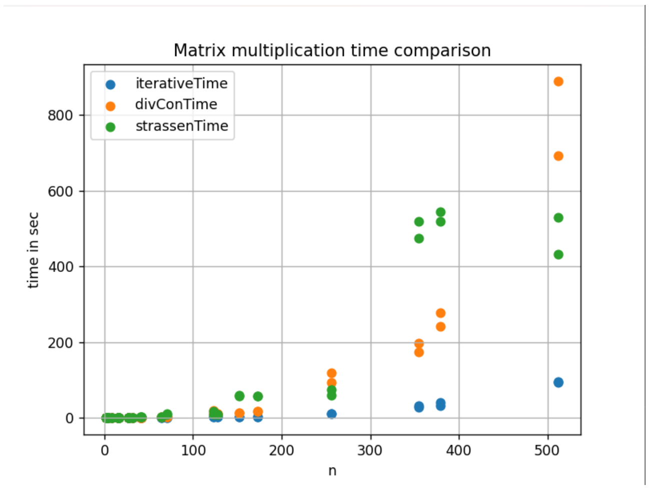

Figure 4.1: Output