Empirical Analysis of Matrix Multiplication

Aditya Kamat-001383483

Akash Srivastava-001388877

Pooja Pokharna-001388119

TABLE OF CONTENTS

1. Introduction 1
2. Strassen’s Algorithm 3
   1. Algorithm and it’s framework……………………………………………………………………………………….4
   2. Winograd’s Algorithm…………………………………………………………………………………………………….5
3. Deciding breakpoint of algorithm’s 8
4. Comparison between algorithm’s……………………………………………………………………………………….12
   1. Processing Time…………………………………………………………………………………………………………………13
   2. Parallel and sequential implementation……………………………………………………………………14
   3. Block Matrix Multiplication………………………………………………………………………………………….14
   4. Classic matrix multiplication with thread…………………………………………………………………15
5. Optimized matrix algorithm…………………………………………………………………………………………………15
6. Implementation of classic Matrix multiplication in c……………………………………………………17
7. Summary and conclusion……………………………………………………………………………………………………..18
8. References…………………………………………………………………………………………………………………………………18
9. contributions…………………………………………………………………………………………………………………………..18
10. ***Introduction:***

The multiplication of two matrices is one of the most basic operations of linear algebra

and has provided an important focus in the search for methods to speed up

the computation.

There are many algorithms that are being generated based on the matrix multiplication algorithms like routines etc. Thus, any speedup in matrix multiplication can increase the performance of a varied variety of mathematical algorithms.

Below is the description of some algorithms:

A. **Brute force inner product multiplication**: This method of multiplying two matrices performs the multiplication of values in each row of the Matrix A against each column of the Matrix B. This multiplication is in the asymptotic complexity of O (n3).

B. **Divide-and-Conquer** with brute force method Here, both the matrices are split up into four submatrices each. Each of these submatrices are further split up, recursively, till submatrices of the required dimension is received. After the base case of recursion is reached, the corresponding submatrices are multiplied using brute-force method, and are assembled back together, resulting in the product matrix.

C. **Strassen’s method** Both of the above-mentioned algorithms perform 8 individual multiplications and 4 additions. Strassen’s method uses 7 multiplications, and therefore has a slightly lesser asymptotic complexity of the order O (n2.807) as compared to the normal or classic matrix multiplication method.

tm(n) = nlog7

ta(n) = 6nlog7 − 6n2

where tm(n) and ta(n) respectively denote the number of multiplications and the number of additions; all logarithms in this report are in base 2, so we will simply write log instead of log2.

All the above-mentioned algorithms are compared against factors such as: Parallel vs sequential computing and block size. The dimensions considered varied from 2×2 matrices to 4096×4096, in powers of two. Square matrices are used in this work.

People have invested time in speeding up practical implementations of matrix multiplication and they have concentrated on the well-known standard algorithm (classic matrix multiplication).

They have spent very less time towards the investigation of alternative algorithms

whose asymptotic complexity is less than the (n3). Operations required by the conventional

algorithm to multiply m x m matrices is 12 i.e. 8 multiplications and 4 additions. One such algorithm is Strassen's algorithm, introduced in 1969, which has complexity (nlog (7)), where lg(x) denotes the base 2 logarithm of x and log (7) is (2.807). There are other algorithms as well which reduced the complexity even closer to the lower bound of (n2) e.g. coppersmith, Wingard, etc.., but so far none of these have lent themselves to practical implementations for typical-size matrices, say of order 10,000 or less.

Strassen's algorithm has the erroneous assumptions that it is not efficient

for matrix sizes that are seen in practice, and that it is unstable. It is well known that Strassen's algorithm can only be applied on a square matrix whose order is a power of two, but issues arise for matrices that are non-square or those having odd dimensions.

1. ***Strassen's Algorithm:***

we have reviewed many important aspects of Strassen's matrix multiplication

algorithm for matrices of arbitrary size. As mentioned above, that Strassen's algorithm can only be applied on a square matrix whose order is a power of two, but issues arise for matrices that are non-square or those having odd dimensions. We have tried to demonstrate

the working of Strassen’s algorithm through both practical and theoretical implementation.

* 1. ***Algorithm and its Framework:***

The classic or standard matrix multiplication algorithm for multiplying two m x m matrices requires m3 scalar multiplications and m3-m2 scalar additions for a total arithmetic operation count of 2m3-m2. In general terms, the classic or standard matrix multiplication algorithm for multiplying two 2 x 2 matrices requires 8 i.e. (23) scalar multiplications and 23-22 i.e. 4 scalar additions for a total arithmetic operation count of 2(2)3-(2)2 i.e. 2(8)-4=12.

Strassen's introduced an algorithm, for square matrices, which is based on a clever way of multiplying 2x2 matrices using 7 multiplications and 18 additions/subtractions. Thus, if A and B are both m x m matrices where m is even, we partition A and B into 2 x 2 block matrices, each block of size m=2 m=2. Then the product C = AB is again partitioned in the same way.

=,

The above matrix can be computed using seven products-

P1 = (A11 + A22) (B11 + B22);

P2 = (A21 + A22) B11;

P3 = A11(B12 - B22);

P4 = A22(B21 - B11);

P5 = (A11 + A12) B22;

P6 = (A21 - A11) (B11 + B12);

P7 = (A12 - A22) (B21 + B22)

and observing that the resultant matrix will be C11 = P1 + P4 - P5 + P7, C12 = P3 + P5, C21 = P2 + P4, and C22 = P1 + P3 - P2 + P6.

Now, to compute the products Pi; i = 1; 2; …. . , 7 using the classic matrix multiplication algorithm, we can calculate the total number of arithmetic operations required by this single application of Strassen's technique as 7(2(m/2)3 - (m/2)2 ) + 18(m/2)2 = (7/4)m3 + (11/4)m2.

The ratio of this operation count to operation required by classic matrix multiplication is—

which is approx 7/8 as m gets large, this implies that for large matrices one level of Strassen's construction produces a 12.5% improvement over classic matrix multiplication.

Strassen's algorithm can continue, applying this construction recursively to compute each Pi ; i = 1; 2; : : : 7, and so on. The recursion can be carried as far as possible, and then terminated at the bottom level when the multiplications are scalar. Hence, we can assume that the simplest version of Strassen's algorithm requires matrix order m = 2d.

To compute matrix C by Strassen's method with full recursion as described above, Let, T(m) be the arithmetic operation count required we have-

T (m) = 7T (m/2) + 18((m/2)2 )=

Thus, compared to the standard method, Strassen's technique provides an asymptotically faster algorithm for multiplying matrices.

Hopcroft and Kerr showed that at least seven multiplications are required by any algorithm for multiplying 2x2 matrices. Thus, a recursive algorithm based on 2x2 matrix multiplication cannot be asymptotically faster than Strassen's algorithm. It is still an open problem whether a faster algorithm can be obtained using 3x3 matrices. Even though Strassen's algorithm leads to an asymptotic improvement there is substantial overhead due to the extra additions and subtractions. If we try to reduce the number of additions it will not lead to an asymptotic improvement, but it may provide a practical improvement. Probert proved that the minimal number of additions and subtractions used by any bilinear algorithm for multiplying 2x2 matrices with seven multiplications is 15.

A bilinear algorithm for computing the product C = AB, consists of four stages: (1) form linear combinations of the elements of A, (2) form linear combinations of the elements of B, (3) form products where the first operand is from (1) and the second operand is from (2), (4) form linear combinations of elements obtained from (3).

* 1. ***Winograd's Algorithm:***

He has taken the base as Strassen’s algorithm and developed a new algorithm by reducing the number of additions. His algorithm required 7 multiplications and minimal number of 15 additions/subtractions.

* + 1. ***Computation:*** Stages (1) and (2) of the algorithm compute-

S1 = A21 + A22;

S2 = S1 - A11 = A21 + A22 - A11;

S3 = A11 - A21;

S4 = A12 - S2 = A12 - A21 - A22 + A11;

T1 = B12 - B11;

T2 = B22 - T1 = B22 - B12 + B11;

T3 = B22 - B12;

T4 = B21 - T2 = B21 - B22 +B12-B11.

Stage (3) computes the seven products

P1 = A11B11

P2=A12B21

P3 = S1T1;

P4 = S2T2;

P5 = S3T3;

P6 = S4B22;

P7 = A22T4;

and stage (4) computes

U1 = P1 + P2;

U2 = P1 + P4;

U3 = U2 + P5;

U4 = U3 + P7;

U5 = U3 + P3;

U6 = U2 + P3;

U7 = U6 + P6:

Which concludes-

C11 = U1,

C12 = U7,

C21 = U4, and

C22 = U5

Complexity—

tm(n) = nlog7

ta(n) = 5nlog7 – 5n2

Now, we may consider the complexity of each of the three algorithms (classic, Strassen and Winograd) as the sum of the number of multiplications tm(n) and the number of additions ta(n):

Tclassic(n) = 2n3 − n2

TStrassen(n) = 7nlog7 − 6n2

TWino(n) = 6nlog7 − 5n2

One key element for obtaining an efficient implementation of Strassen's algorithm is to stop the recursions early and switching it to the classic algorithm when Strassen's construction no longer leads to an improvement. The test used to determine whether to apply another level of recursion is called the breakpoint. Finally, for matrices with odd dimensions, some technique must be applied to make the dimensions even e.g. padding (generally involve either adding or deleting rows and/or columns to/from the original matrix), then apply Strassen's algorithm to the altered matrix, and then correct the results. For now, we assume that matrix dimensions are always even.

To calculate the actual performance of Strassen's algorithm on real machines we cannot rely only on operation (multiplication and additions or subtractions) count alone. While revealing asymptotic behavior, it can be a poor predictor of actual cost on specific problem sizes and machines for it does not consider memory access patterns, possible data reuse, and differences in speed between different arithmetic operations.

There are three different models for estimating the computing time of Strassen’s algorithm. These are based on three different assumptions for the costs M (m; k; n) and G (m; n). The first and simplest model stays with operation count, thus setting M (m; k; n) = 2mkn - mn and G (m; n) = mn. Where, M (m; k; n) be the cost of multiplying an m x k matrix by a k x n matrix using the classic matrix multiplication algorithm and G (m; n) be the cost of adding or subtracting two m x n matrices.

Our second model represents a simple attempt to differentiate between the compute rates for matrix addition/subtraction and multiplication on actual machines. M (m; k; n) and G (m; n) should be, cubic and quadratic functions of their parameters respectively. Thus, we use µmkn for M, and ƴmn for G. The coefficients µ and ƴ and can be chosen based on the running rates of a particular algorithm on given machine.

Even the second model is lacking detail since it does not consider different shapes of matrices and effects which may be prevailing for small matrices. Thus, our third model includes low order terms, using a full cubic polynomial in m, k, and n for M, and a quadratic polynomial in m and n for G. The second model is an asymptotic estimate of the third model. In the third model,

M (m, k, n) = µ7mkn + µ6mk + µ5mn + µ4kn + µ3m + µ2k + µ1n + µ0

and

G (m, n) = ƴ3mn + ƴ2m + ƴ1n + ƴ0

The coefficients µi , i = 0; 1; : : : ; 7 and ƴj , j = 0; 1; 2; 3, can be obtained empirically for a given implementation of the operation on a given computer by using a least squares of the polynomials M (m; k; n) and G(m; n) to measured the computation time of an algorithm.

If A and B are of size 2dm x 2d k and 2d k x 2d n, respectively, then Strassen's algorithm can be called recursively d times. If we choose to stop the recursion after these d steps of application of Strassen's algorithm, so that the standard algorithm is used to multiply the resulting m x k and k x n matrices, then the cost of Strassen's algorithm in terms of M and G is—

W (2dm, 2dk, 2dn) = 7d (2mkn-mn) + (7d - 4d) (4mk+ 4kn + 7mn)/3

To calculate based on the given computer—

W (2dm, 2dk, 2dn) = µ 7d (2mkn-mn) + ƴ (7d - 4d) (4mk+ 4kn + 7mn)/3

For the square matrix case (m = k =n) which simplifies our equation—

W (2dm) = 7d (2(m)3 - (m)2) + 5(m)2 (7d - 4d).

and, if Strassen's original version of the algorithm had been used in the above equation then –

S (2dm) = 7d (2(m)3 - (m)2) + 6(m)2 (7d - 4d).

Here, we have obtained closed form empirical expression for operation count costs of both the original and the Winograd variant of Strassen's algorithm which will be used to estimate the performance of our implementation.

1. ***Deciding Breakpoint of Algorithm’s:***

As mentioned above, stopping the Strassen recursions early and performing the remaining matrix multiplications using the classic algorithm. Using the operation count model, we present theoretical and empirical techniques for determining the optimal stopping point based on the above equations.

One level of Strassen's algorithm requires 7 multiplications and 15 additions/subtractions. Thus, applying the algorithm to 2x2 matrices requires 22 arithmetic operations (25 for the original version), while classic matrix multiplication only requires 8 multiplications and 4 additions total 12 operations. Thus, we see that for small matrices, the extra additions/subtractions can outweigh the benefit of the reduced number of multiplications. Furthermore, this affects the algorithm even when applied to larger matrices, for if the recursions are carried to the bottom (scalar) level very large matrices are needed before Strassen's algorithm becomes faster than classic matrix multiplication.

For example, if we use to compute the arithmetic operation count cost of Strassen's algorithm with full recursion for various square matrix sizes by setting m = 2 and varying d, in equation

W (2dm) = 7d (2(m)3 - (m)2) + 5(m)2 (7d - 4d).

we obtain the results shown in Table1.

|  |  |  |  |
| --- | --- | --- | --- |
| **2d** | **Classic MM** | **Strassen (Winograd) (Full Recursion)** | **Strassen (Original)**  **(Full Recursion)** |
| 2 | 12 | 22 | 25 |
| 8 | 960 | 1,738 | 2,017 |
| 16 | 7,936 | 13,126 | 15,271 |
| 32 | 64,512 | 95,722 | 111,505 |
| 128 | 4,177,920 | 4,859,338 | 5,666,497 |
| 256 | 33,488,896 | 34,261,126 | 39,960,391 |
| 512 | 268,173,312 | 240,810,922 | 280,902,385 |
| 1024 | 2,146,435,072 | 1,689,608,614 | 1,971,035,287 |
| 2048 | 17,175,674,880 | 11,842,988,938 | 13,816,121,377 |

Table 1: *Operation Count Costs for 2d X 2d Matrices, Full Recursion.*

Observe that the smallest square power-of-2 matrix order for which Strassen's algorithm has a lower operation count than regular classic matrix multiplication is 2048, or 1024 for the original version.

The key to not continue the Strassen recursions past the point where it is more efficient to use regular matrix multiplication. To determine the optimal point, or breakpoint, at which to switch from Strassen's algorithm to regular classic matrix multiplication we need to consider when one application of Strassen's recursion leads to an improvement over completing the remaining required matrix multiplications using the standard algorithm. Therefore, the optimal breakpoint is independent of overall matrix size and this small change dramatically improves the practical performance of Strassen's algorithm.

We will use the empirical procedure to determine the breakpoint of the algorithm. We again assuming that matrix dimensions are always even. We need to characterize the set of positive integers (m; k; n) such that using the standard algorithm alone is less costly than applying one level of Strassen's recursion followed by the standard method—

This is equivalent to finding the solutions to the inequality—

M (m, k, n) ≤ 7M (m/2, k/2, n/2) + 4G (m/2, k/2) + 4G (k/2, n/2) + 7G (m/2, n/2)

The set of solutions to this inequality will be the values for m, k, and n for which we should switch to the standard algorithm.

Then using the classic matrix multiplication, the equation will become—

mkn ≤ 4(mk + kn + mn)

On solving the above inequality—divide both sides by mkn—

1 ≤ 4(1/n + 1/m + 1/k)

First, we examine the case of square matrices where m = k = n. In this case we obtain m ≤ 12.

1 ≤ 4(1/m + 1/m + 1/m)

↓

1 ≤ 4(3/m)

↓

1 ≤ 12/m

↓

m ≤ 12

In other words, the operation count is reduced, using Strassen's algorithm, when applied to matrices of order greater than 12, and not reduced when applied to matrices of order less than or equal to 12.

We came to conclusion that, for the case so far considered, instead of recursing Strassen’s algorithm all the way down to matrices of dimension 1 (scalars), we should instead switch to regular classic matrix multiplication whenever the remaining matrix multiplications involve matrices whose order is 12 or less.

To prove this, we can compute the operation count cost of Strassen's algorithm with breakpoint at order 12 for various square power-of-2 order matrix sizes by setting m = 8 and varying d, thus obtaining the results shown in Table 2 from equation-

W (2dm) = 7d (2(m)3 - (m)2) + 5(m)2 (7d - 4d).

|  |  |  |  |
| --- | --- | --- | --- |
| **8\*2d** | **Classic MM** | **Strassen (Winograd) (Breakpoint  = 12)** | **Strassen (Original)**  **(Full Recursion)** |
| 8 | 960 | 960 | 1,738 |
| 16 | 7,936 | 7,680 | 13,126 |
| 32 | 64,512 | 57,600 | 95,722 |
| 128 | 4,177,920 | 2,991,360 | 4,859,338 |
| 256 | 33,488,896 | 21,185,280 | 34,261,126 |
| 512 | 268,173,312 | 149,280,000 | 240,810,922 |
| 1024 | 2,146,435,072 | 1,048,892,160 | 1,689,608,614 |
| 2048 | 17,175,674,880 | 7,357,973,760 | 11,842,988,938 |

Table 2: Operation Count Costs for 8\*2d x 8\*2d Matrices, Cutoff= 12.

***4.Comparison between Algorithm’s:***

***4.1 Processing time:***

Below is the Avg. processing time of all the algorithms when we run it 100 times—

In order to do comparison in terms of running time for different algorithms, we are calculating this time in the following manner.

* Every algorithm has been given its own function.
* We are storing the running time for every function in a variable.
* The time calculated and stored in this variable has been calculated using System.nanoTime() function.
* All these functions are inside a loop which runs for 20 times. So, eventually, we are summing up the time required for individual functions for 20 times.
* Average of the time is then calculated for every function and then time is calculated in milliseconds
* We are running the loop for 20 times in order to deal with the following fluctuations
* We have observed that, whenever we calculate the time required for running any algorithm, it is not the same, if we run the same algorithm again. There is a change in time in terms of few milliseconds for larger matrices and few nanoseconds for small matrices. Sometimes, the time calculated is larger than the previous run and sometimes, the time calculated is smaller than the previous loop. Therefore, it becomes necessary to run these algorithmic functions again and again in a loop and then calculate the average time, which will be more accurate one.
* The reason behind the changes in time for any algorithmic function is mostly due to the processing which is carried internally in the system.
* We have considered the following algorithms and calculated their time after running them for 100 consecutive times.
* Classic Matrix Multiplication
* Block Matrix Multiplication
* Classic Matrix Multiplication with threading
* Divide and Conquer Matrix Multiplication
* Strassens Matrix Multiplication
* Here, we have introduced two new implementations of algorithms. The one is Block Matrix Multiplication and the second one is Classic Matrix multiplication using threading

These algorithms have been executed for matrix size beginning from N=2 to N = 1024

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2d** | **Classic MM**  **(Time in milliseconds)** | **Strassen**  **(Time in milliseconds)** | **Divide & Conquer (Time in milliseconds)** | **Block**  **(Time in milliseconds)** | **Classic with Thread**  **(Time in milliseconds)** |
| 16 | 0 | 0 | 2 | 0 | 1 |
| 32 | 0 | 2 | 11 | 0 | 1 |
| 64 | 0 | 13 | 73 | 0 | 2 |
| 128 | 3 | 114 | 721 | 3 | 8 |
| 256 | 27 | 783 | 5481 | 23 | 47 |
| 512 | 264 | 5233 | 42008 | 237 | 278 |

Table 3: Comparison between different algorithms based on processing time.

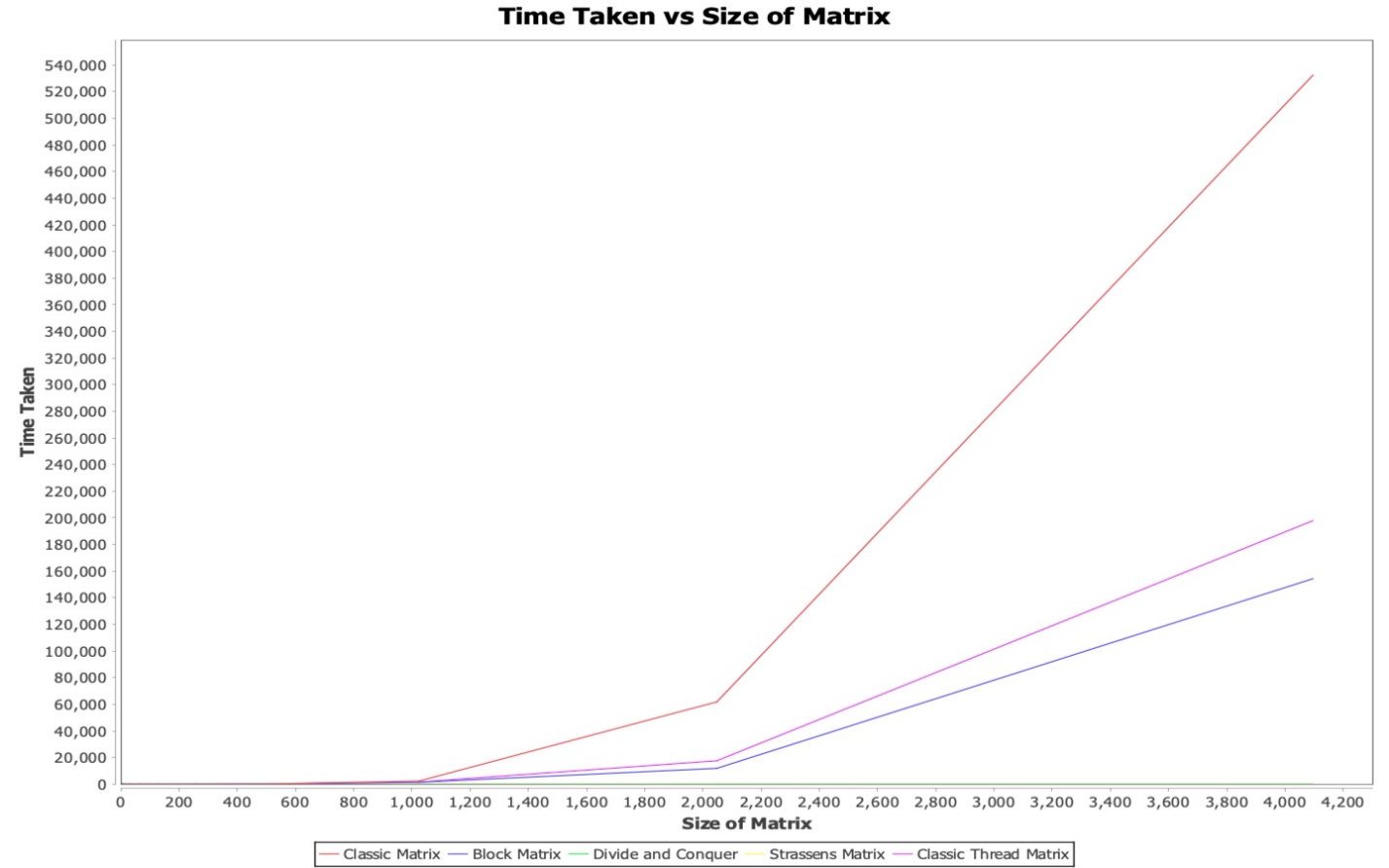
Below figure is the graphical representation of the above table—

A screenshot of a social media post

Description automatically generated

***4.2 Parallel and sequential Implementation—***

Below graph is the comparison between classic matrix multiplication with thread implementation and block size matrix implementation.



***4.3*** ***Block Matrix Multiplication***

This is a variation on normal matrix multiplication wherein you divide the matrix into smaller sub-matrices and then calculate those matrices individually. The advantage of this method is that you will keep the values that you need to do the matrix calculation in the cache longer. This will speed up the calculations.

That is why this type of matrix multiplication is often referred to as block matrix multiplication.

The performance of this matrix multiplication will vary as per the size of block.

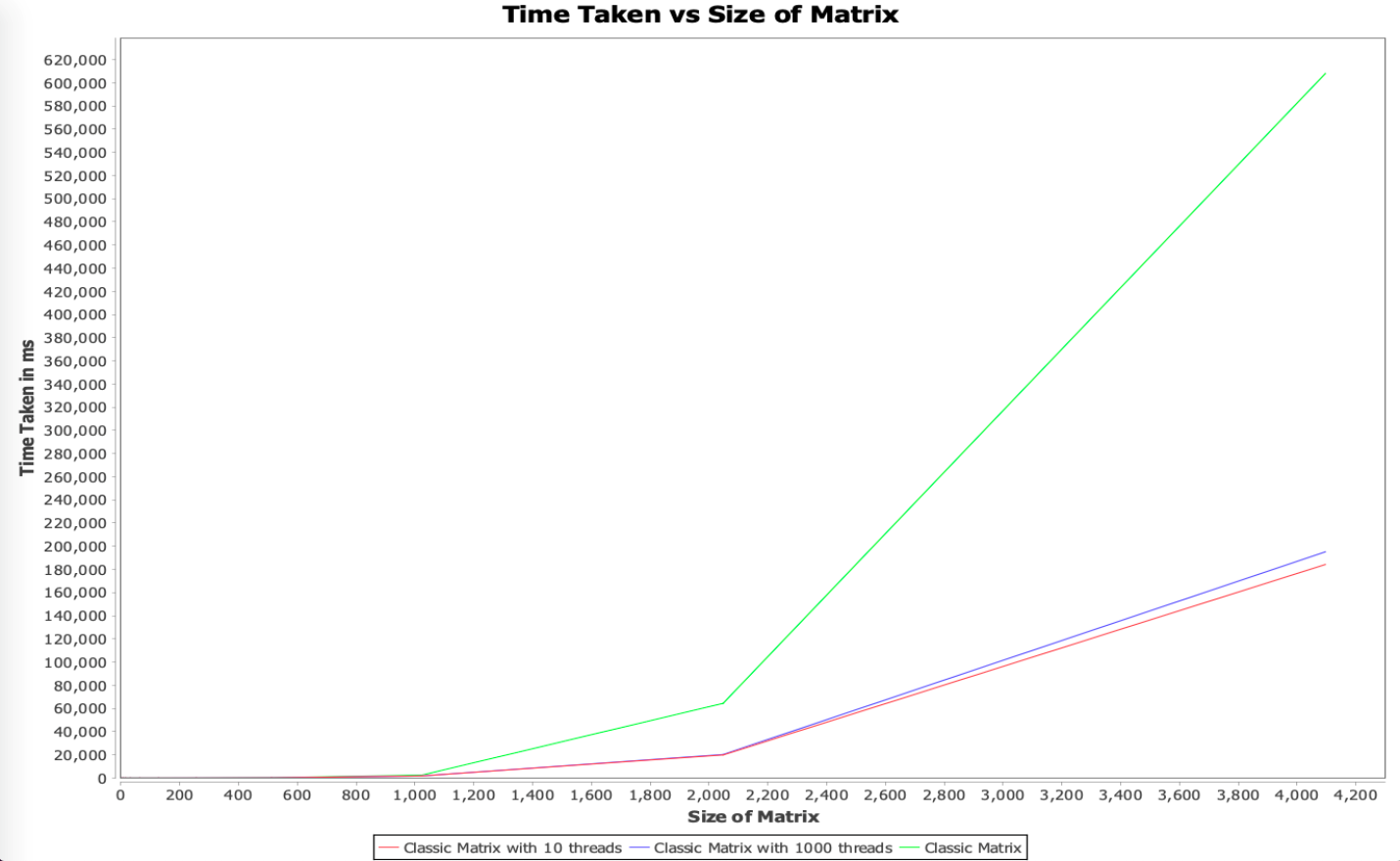
As per the above graph, block matrix multiplication performs well than classic matrix multiplication

***4.4*** ***Classic matrix multiplication with thread***

We can clearly see that classic matrix multiplication is the best to obtain product of 2 matrices, but this is not true for large matrices. To optimize matrix multiplication, we have implemented the concept of multithreading. Multithreading allows us to calculate the value of the elements for the product matrix simultaneously.

For multithreading implementation, we have used Java’s Executor Framework. Performance of the program improved for large matrix multiplication as compared to the non-threaded implementation.

Other observations include that the performance of the classic multiplication improves with less amount of threads rather than having a large number of threads working simultaneously. For example, if we use 10 threads rather than 1000 threads for multiplying the matrices, the implementation with 10 simultaneous threads out performs the implementation with a 1000 simultaneous thread on most occasions. Hence, the implementation with 10 threads seem to be the most optimized.



***5- Optimized Matrix Algorithm***

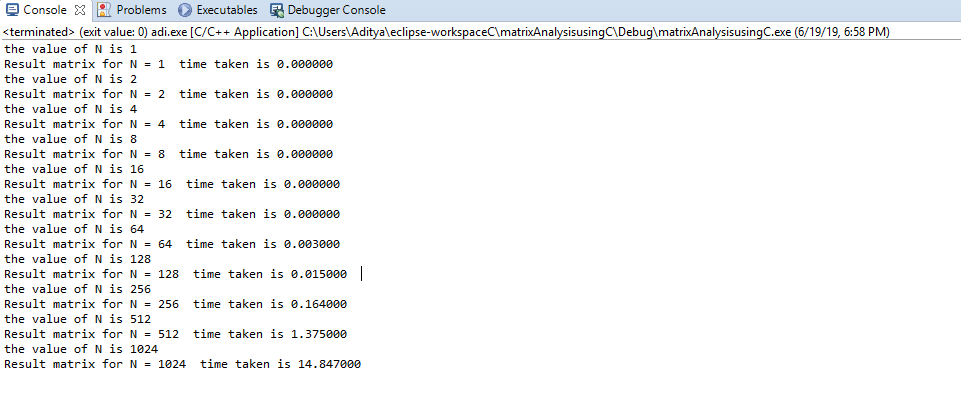
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2d** | **Classic MM**  **(Time in milliseconds)** | **Strassen**  **(Time in milliseconds)** | **Block**  **(Time in milliseconds)** | **Classic with Thread**  **(Time in milliseconds)** |
| 16 | 0 | 1 | 0 | 7 |
| 32 | 1 | 6 | 1 | 7 |
| 64 | 0 | 13 | 9 | 12 |
| 128 | 3 | 205 | 5 | 55 |
| 256 | 27 | 783 | 24 | 97 |
| 512 | 392 | 9795 | 450 | 273 |
| 1024 | 6200 | 69093 | 2782 | 2000 |

After comparing all the different available algorithms, the most efficient algorithms were Classic Matrix multiplication with threading and Block Matrix Multiplication

For N having very less values (lesser than 32), classic Matrix multiplication took lesser time than other matrix multiplication techniques.

For N >= 4096, Strassens Matrix Multiplication will be executed. Even in the case of execution of strassen Matrix Multiplication, we have considered one more case of Optimization in it. Strassen Matrix, divides itself into sub matrices and runs recursively. For Matrices, larger than 4096, once the sub matrices size is less than 4096, then we are again running block matrix for optimizing the algorithm.

***6- Implementation of Classic Matrix Multiplication in C***

******

|  |  |  |
| --- | --- | --- |
| **2d** | **Classic MM in Java**  **(Time in milliseconds)** | **Classic MM in C**  **(Time in milliseconds)** |
| 16 | 0 | 0 |
| 32 | 0 | 0 |
| 64 | 0 | 3 |
| 128 | 3 | 15 |
| 256 | 27 | 164 |
| 512 | 392 | 1375 |
| 1024 | 6200 | 14847 |

In case of implementation of code in C, the internal thread gets suspended due to improper memory allocation. In Java, the memory allocation and garbage collection is handled internally but that’s not the case in C. Hence, while carrying out any multiplication, it is necessary to allocate memory using malloc function and then free up the allocated space using free function.

After comparing the results of implementation of code in C and Java, it is found that the time required for implementation in C is very large in comparison with Java

***7- Summary and Conclusion***

As per our analysis, we can say that classic matrix multiplication is very efficient for smaller values of N (N <=32)

It was difficult to choose between block matrix multiplication and classic matrix multiplication with threads as both outperformed each other on several executions. After running the algorithms for particular period of time and considering the average, we can say that classic matrix multiplication using threading performs the best for N < = 4096.

Due to processing power constraints, it is not possible to calculate matrix multiplication for values of N > 4096 but we are using strassen’s matrix multiplication with a combination of block and threading matrix for optimum performance.

Use of Java has been better than C in terms of performance as Java takes lesser time to compute than C.

***8- References***

1. <https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm>
2. <https://en.wikipedia.org/wiki/Strassen_algorithm>
3. <https://www.youtube.com/watch?v=0oJyNmEbS4w>
4. <https://www.geeksforgeeks.org/strassens-matrix-multiplication/>
5. <https://www.geeksforgeeks.org/multiplication-matrix-using-pthreads/>
6. <https://www.geeksforgeeks.org/calloc-versus-malloc/>

***9-Contributions***

Aditya Kamat: 33.333%

Akash Srivastava: 33.33%

Pooja Pokharna: 33.33%