ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
DEMOCRITUS UNIVERSITY OF THRACE

Department of Electrical & Computer Engineering

Laboratory of Computer Architecture & High-Performance Systems

Matrix multiplication using Strassen's Algorithm with MPI

Mazarakis Periklis A.M. 57595

Papadopoulos Aristeidis A.M. 57576

Tsapekos Theodoros A.M. 57681

Academic Year 2021-2022

Theoretical background of Strassen's Algorithm

Strassen's algorithm is a way of multiplying matrices faster than the vanilla method of multiplication. Where the conventional multiplication has an algorithm complexity of $O(n^3)$, Strassen's method has $O(n^{2.80})$. The key of this methos is that instead of 8 recursive algorithms, it utilizes only 7. The difference might not seem like much with a first look, but for large matrices dimensions a lot of calculation time can be saved. Though, in order for this to happen, a lot more summations and subtractions must be calculated. Strassen's algorithm consists of four basic steps:

1. Each of the input matrices A, B and output matrix C is divided into $^n/_2 \times {}^n/_2$ sub-matrices.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

2. 10 new matrices S_1, S_2…,S_{10}, are created, each one of size $^n/_2 \times {}^n/_2$ and equal to the addition or subtraction of the two input matrices of step one. Each S_i matrix is computed as follows:

$$S1 = B12 - B22 \qquad S6 = B11 + B22$$
$$S2 = A11 + A12 \qquad S7 = A12 - A22$$
$$S3 = A21 + A22 \qquad S8 = B21 + B22$$
$$S4 = B21 - B11 \qquad S9 = A11 - A21$$
$$S5 = A11 + A22 \qquad S10 = B11 + B12$$

3. Using the submatrices created in step one, and the matrices from step two, seven new matrices P_1,P_2…,P_7 are created recursively, each one with dimensions $^n/_2 \times {}^n/_2$. These matrices are calculated using the following equations:

$$P_1 = A_{11} * S_1 = A_{11} * B_{12} - A_{11} * B_{22}$$

$$P_2 = S_2 * B_{22} = A_{11} * B_{22} + A_{12} * B_{22}$$

$$P_3 = S_3 * B_{11} = A_{21} * B_{11} + A_{22} * B_{11}$$

$$P_4 = A_{22} * S_4 = A_{22} * B_{21} - A_{22} * B_{11}$$

$$P_5 = S_5 * S_6 = A_{11} * B_{11} - A_{11} * B_{22} + A_{22} * B_{11} + A_{22} * B_{22}$$

$$P_6 = S_7 * S_8 = A_{12} * B_{21} - A_{12} * B_{22} - A_{22} * B_{21} - A_{22} * B_{22}$$

$$P_7 = S_9 * S_{10} = A_{11} * B_{11} - A_{11} * B_{12} - A_{21} * B_{11} - A_{21} * B_{12}$$

4. Lastly, the desired C_{11},C_{12}, C_{21},C_{22} submatrices are calculated, from adding/subtracting P_i matrices. Each C_{11},C_{12}, C_{21},C_{22} is calculated as follows:

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

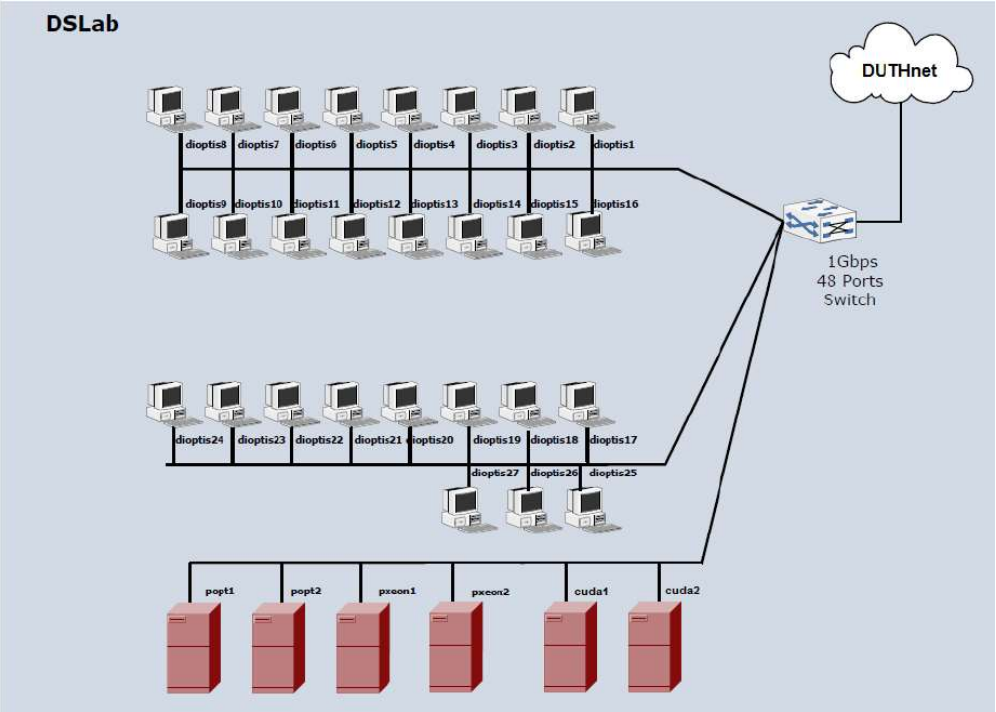
$$C_{22} = P_5 + P_1 - P_3 + P_7$$

Essentially, this method of computation leads to the fragmentation of the original input matrices, and with its recursive nature, results in simple element multiplication. Afterwards, with proper combination of these simple elements, the C_{11}, C_{12}, C_{21}, C_{22} submatrices are created.

The aforementioned algorithm, while having just a little less computational complexity, has some major drawbacks such as:

- The two input matrices must be square matrices.
- Due to the recursive nature of the algorithm, large RAM memory is required.
- For matrices with relatively small dimensions, this method is worse than the vanilla method. In order to reap the rewards of Strassen's algorithm, the input matrices dimensions must be near 1000x1000 or more. For smaller matrices(in terms of dimensions) the vanilla method is faster.
- Actually, this algorithm is more efficient after a threshold has been reached. This threshold is dependent on the algorithm implementation and the hardware the implementation runs on.
- The reduced mathematical operations could lead to less stability of the algorithm.
- Increasing the complexity of the algorithm's implementation, some of these drawbacks could be overcome.

Network Layout



The nodes used can be seen above, with the names “dioptis” and each of them has a 2GB RAM memory.

Serial Implementation

Strassen’s algorithm serial implementation is as follows:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
// Use of MPI here, because MPI_Wtime is more accurate than time functions from
// <time.h>
// N indicates the matrices dimensions, e.g. N=120 means matrices of 120x120
#define N 120
// Dynamically allocate memory using the following function
float **mem_alloc(int size){
    int i;
    float **array = (float **) malloc (size * sizeof (float *));
    for (i=0; i< size; i++) array[i]=malloc(size*sizeof(float));
    return array;
}
// Matrix Initialization Function
float **create_matrix(int size){
    // Due to the use of 2d arrays, int pointer is needed. Therefore,
    // using malloc, memory size is being allocated, and then the matrix is
    // randomly initiated
    float **array = mem_alloc(size);
    int i,j;
    for (i = 0; i < size; i++){
        for (j =0; j<size; j++) array[i][j] = rand() % 10; // normalize values
    }
    return array;
}
```

After the necessary libraries are imported, the above two functions are defined. The first one, allocates dynamically memory for 2D matrices, while the second one randomly fills the array elements of the desirable size.

```
float **Strassen(float **matrixA,float **matrixB,int n){
        // This function implements the Strassen Multiplication Algorithm.
        // The input arrays are being split into 4 submatrices each and then the
        // Strassen coefficients (Pi,i=1..7) are being computed recursively using
        // this function, until the result is the result of a simple Multiplication
        // between two numbers. So in every recursion the arrays are being reduced
        // until they become 2x2 matrices.Finally, the result submatrices (Ci,j) are
        // concatenated floato one.
        float **P1,**P2,**P3,**P4,**P5,**P6,**P7;
        float **C11,**C12,**C21,**C22;
        float **S1,**S2,**S3,**S4,**S5,**S6,**S7,**S8,**S9,**S10;
        float **A11,**A12,**A21,**A22,**B11,**B12,**B21,**B22;
    int i,j;
    float ** res = mem_alloc(n);
    int new_n = n/2;
    if(n>1) {
                A11 = mem_alloc(new_n);
                A12 = mem_alloc(new_n);
                A21 = mem_alloc(new_n);
                A22 = mem_alloc(new_n);
                B11 = mem_alloc(new_n);
                B12 = mem_alloc(new_n);
                B21 = mem_alloc(new_n);
            B22 = mem_alloc(new_n);
            for (i = 0; i < new_n; i++) {
                for(j = 0; j<new_n; j++){
                        A11[i][j] = matrixA[i][j];
                A12[i][j] = matrixA[i][j + new_n];
                A21[i][j] = matrixA[i + new_n][j];
                A22[i][j] = matrixA[i + new_n][j + new_n];
                B11[i][j] = matrixB[i][j];
                B12[i][j] = matrixB[i][j + new_n];
                B21[i][j] = matrixB[i + new_n][j];
                B22[i][j] = matrixB[i + new_n][j + new_n];
                    }
            }
            S1 = mem_alloc(new_n);
            S2 = mem_alloc(new_n);
            S3 = mem_alloc(new_n);
            S4 = mem_alloc(new_n);
            S5 = mem_alloc(new_n);
            S6 = mem_alloc(new_n);
            S7 = mem_alloc(new_n);
            S8 = mem_alloc(new_n);
            S9 = mem_alloc(new_n);
            S10 = mem_alloc(new_n);
            for (i=0;i<new_n;i++){
                for(j=0;j<new_n;j++){
                    S1[i][j] = B12[i][j] - B22[i][j];
                    S2[i][j] = A11[i][j] + A12[i][j];
                    S3[i][j] = A21[i][j] + A22[i][j];
                    S4[i][j] = B21[i][j] - B11[i][j];
                    S5[i][j] = A11[i][j] + A22[i][j];
                    S6[i][j] = B11[i][j] + B22[i][j];
                    S7[i][j] = A12[i][j] - A22[i][j];
                    S8[i][j] = B21[i][j] + B22[i][j];
                    S9[i][j] = A11[i][j] - A21[i][j];
                    S10[i][j] = B11[i][j] + B12[i][j];
                }
            }
            P1 = Strassen(A11,S1,new_n);
            P2 = Strassen(S2,B22,new_n);
            P3 = Strassen(S3,B11,new_n);
            P4 = Strassen(A22,S4,new_n);
            P5 = Strassen(S5,S6,new_n);
            P6 = Strassen(S7,S8,new_n);
            P7 = Strassen(S9,S10,new_n);
            //free unneeded memory space
            free(A11);free(A12);free(A21);free(A22);free(B11);free(B12);free(B21);free(B22);
            free(S1);free(S2);free(S3);free(S4);free(S5);free(S6);free(S7);free(S8);free(S9);free(S10);
            C11 = mem_alloc(new_n);
            C12 = mem_alloc(new_n);
            C21 = mem_alloc(new_n);
            C22 = mem_alloc(new_n);
            for(i=0;i<new_n;i++){
                for(j=0;j<new_n;j++){
                        C11[i][j] = P5[i][j] + P4[i][j] - P2[i][j] + P6[i][j];
                        C12[i][j] = P1[i][j] + P2[i][j];
                        C21[i][j] = P3[i][j] + P4[i][j];
                        C22[i][j] = P5[i][j] + P1[i][j] - P3[i][j] - P7[i][j];
                }
            }
            // Free unneeded space
```

```
            free(P1);free(P2);free(P3);free(P4);free(P5);free(P6);free(P7);
          for (i=0;i<new_n;i++){
            for(j=0;j<new_n;j++){
              res[i][j] = C11[i][j];
              res[i][j+new_n] = C12[i][j];
              res[new_n+i][j] = C21[i][j];
              res[new_n+i][new_n+j] = C22[i][j];}
            }
          }
    else {res[0][0] = matrixA[0][0]*matrixB[0][0];}
        return res;
}
int i,j,k;
int main(int argc, char *argv[]) {
          float **C = mem_alloc(N);
          int new_n = N/2;
          MPI_Init(&argc,&argv);
        srand(0);
        //Initiate matrices
        printf("Matrices Initiation \n");
        float **A = create_matrix(N);
          float **B = create_matrix(N);
        printf("Normal Computation\n");
          // Timer for normal computation
          double start_time_normal = MPI_Wtime();
        for(i=0;i<N;i++){
          for(j=0;j<N;j++){
            for(k=0;k<N;k++) C[i][j] += A[i][k]*B[k][j];
          }
        }
          double end_time_normal = MPI_Wtime(); // Timer for normal computation
          printf("Time for Normal Computation is %f \n",end_time_normal - start_time_normal);
          printf("Strassen Computation\n");
          free(C);
          double one_core_start = MPI_Wtime(); // Timer for serial strassen computation
      float **C_strassen = Strassen(A,B,N);
          double one_core_end = MPI_Wtime(); // Timer for serial strassen computation
          printf("Time for Serial Strassen Computation is %f\n", one_core_end - one_core_start);
          // Free memory space
          free(A);
          free(B);
          free(C_strassen);
          MPI_Finalize();
}
```

Afterwards, the Strassen algorithm is implemented. The input matrices A,B are divided into submatrices of dimensions $^{n}/_{2} \times {}^{n}/_{2}$ and then memory is allocated for the matrices S_1, S_2…,S_{10} . After their calculation, P_1,P_2…,P_7 follow, using recursively the same function. After P_i are computed, memory is freed and allocated for submatrices C_{11}, C_{12}, C_{21}, C_{22}, which are then computed based on step four equations and placed properly in the final output matrix C.

Running the above code, for N=4, meaning that the matrices are 4x4, one can understand if the method procures the correct results. Its output is the following:

```
Normal Computation
C using naive computation is:
  185.000031    129.000000    151.000031    54.000000
  127.000031    87.000000    105.000031    42.000000
  33.000031    11.000000    13.000032    12.000000
  70.000031    40.000000    32.000031    6.000000
Strassen Computation
C using Strassen is:
  185.000000    129.000000    151.000000    54.000000
  127.000000    87.000000    105.000000    42.000000
  33.000000    11.000000    13.000000    12.000000
  70.000000    40.000000    32.000000    6.000000
```

Parallel Implementation

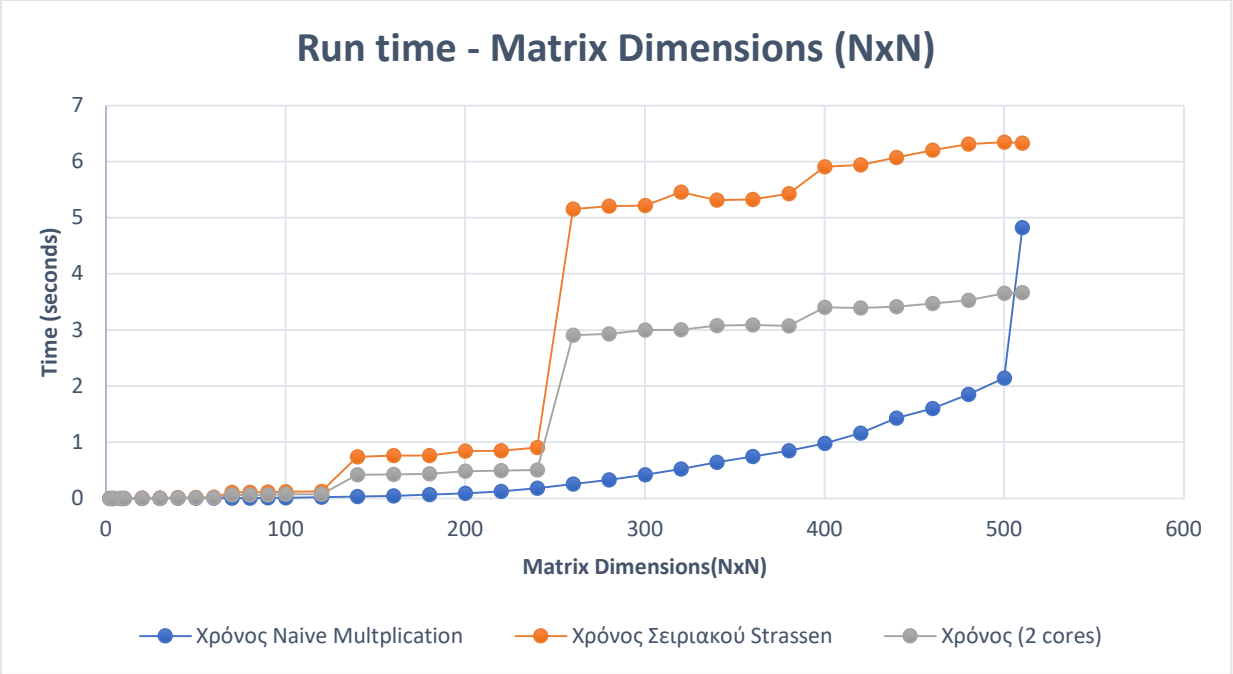
There are many different methods that someone can use to implement Strassen's algorithm in parallel. For the purposes of this project, two methods were researched. First of all, the original input matrices must be sent to every other node in the network by the root node. Then, every node creates the submatrices $A_{i,j}$, $B_{i,j}$ and the calculation of P_i , is broken up between the number of nodes used, so that each node computes of one such matrix. After this process has been completed, the results are sent back to the root node, where the submatrices $C_{i,j}$ are calculated, i.e., the end result. Potentially, it could be possible for increased parallelism in a way that the submatrices $A_{i,j}$, $B_{i,j}$ are broken up into submatrices depending on their size, but then the complexity of the code would increase by a lot.

The second method we tried, follows the aforementioned steps, but after the computation of P_i, four of the available nodes (or two if four aren't available) compute of one(or two) $C_{i,j}$ each and then root collects all those submatrices and constructs the C matrix. Alas, this method wasn't used, because the communication overhead would overcome the gains of Strassen's Algorithm.

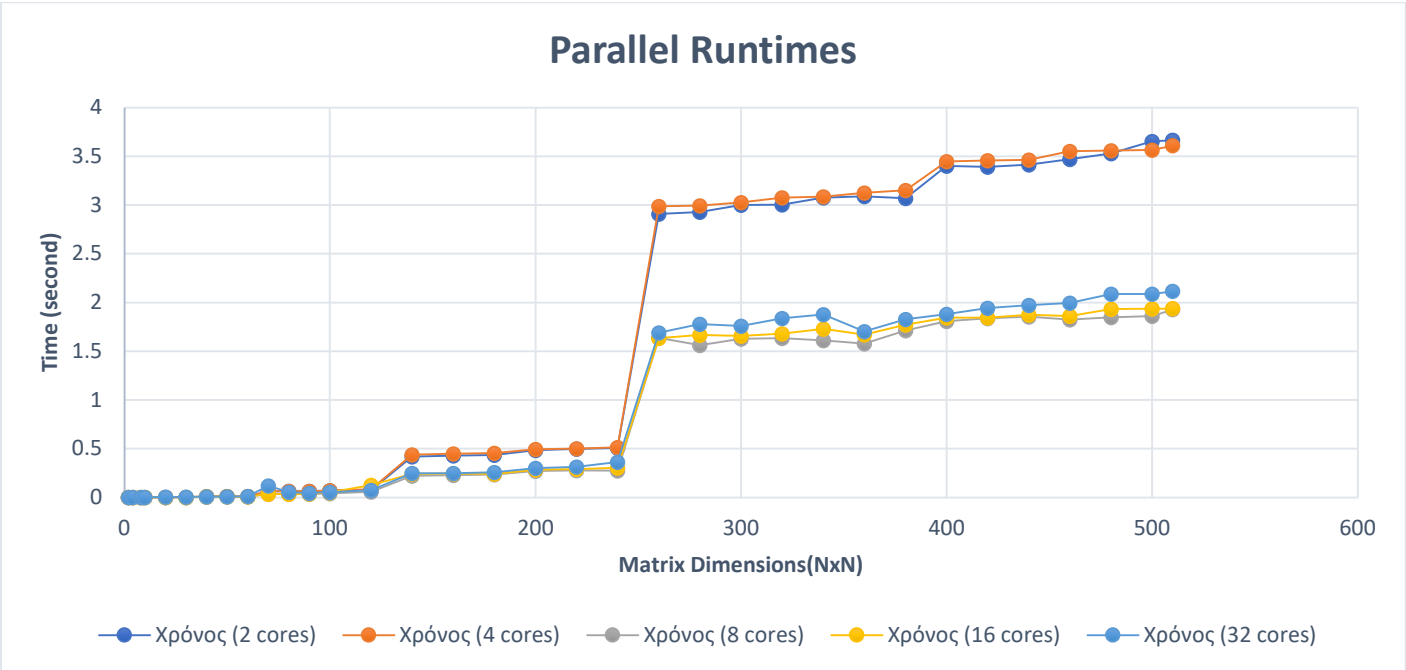
So, using the first method, we examined different sized matrices up to 510x510. After those dimensions, due to large memory requirements, the program crashed, which made it impossible to continue, at least for this specific implementation. Ideally, matrices with dimensions equal to a power of 2, helps divide equally the computational cost to the available processors, without this statement being necessary to run the algorithm. In the following matrix the run time (in seconds) for different number of processors and different matrix sizes.

	Number of nodes/processors					
Matrices Dimension	1	2	4	8	16	32
50	0.017081	0.009886	0.010999	0.006487	0.008033	0.008059
100	0.119569	0.069142	0.071619	0.039046	0.04884	0.052908
200	0.842056	0.484121	0.49316	0.270716	0.282562	0.298439
300	5.215494	2.99921	3.025402	1.625625	1.65834	1.75937
400	5.90756	3.40188	3.447827	1.809301	1.844279	1.878658
510	6.331387	3.667094	3.611091	1.928006	1.938972	2.115468

In the following plot, the runtime can be seen for different matrices dimensions, using vanilla multiplication, Strassen in one processor and in two. With the blue line, the naive multiplication can be seen, while the orange line displays the time of Strassen's algorithm ran in one node, and finally the gray line is the runtime of Strassen's algorithm in two nodes.



According to this plot and having in mind the max dimensions that we could use were 510x510, it seems that vanilla multiplication (naive) is more efficient than running Strassen's algorithm in one node, but there seems to be an incline to surpass it after N=510, making the vanilla multiplication inefficient after that point. Furthermore, the parallel algorithm using two cores is more efficient for these dimensions, so we suppose that for even larger matrices, Strassen's algorithm is in fact faster. In the next graph, we can observe the evolution of the algorithm's runtime for 2,4,8,16,32 nodes.

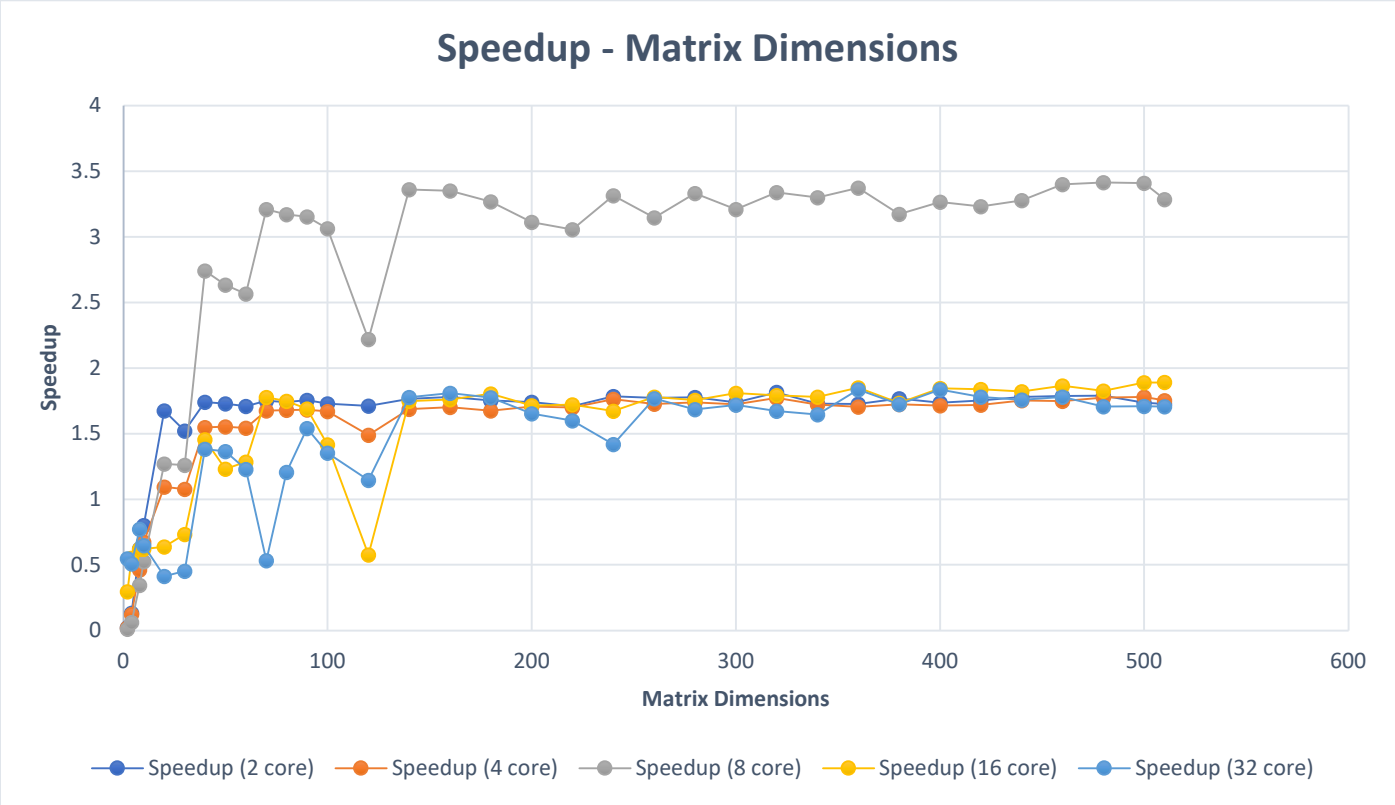


With the blue line, the runtime for two cores can be seen, while with the orange, gray, yellow and light blue the time for 4,8,16,32 nodes is displayed. Using the above plot, the program is in fact sped up, using up to 8 nodes. With more nodes, no speedup is achieved. This is contributed to two factors. First, the fact that it's a drawback of this implementation and second that in order to observe more speedup, larger dimensions are needed. In the

following matrix, speedup can be seen in connection with the matrix dimensions and the number of nodes.

	Number of nodes				
Matrix Dimensions	2	4	8	16	32
50	1.727796884	1.55295936	2.633112379	1.230673472	1.36480953
100	1.729325157	1.669515073	3.062259899	1.415683866	1.353651622
200	1.739350286	1.707470192	3.110477401	1.713326633	1.652464993
300	1.738955925	1.723901154	3.208300807	1.808561574	1.719593946
400	1.736557433	1.713415435	3.265106248	1.844558226	1.835260596
510	1.726540689	1.753316934	3.283904199	1.89125681	1.706993913

While using more datapoints, the following plot can be created:

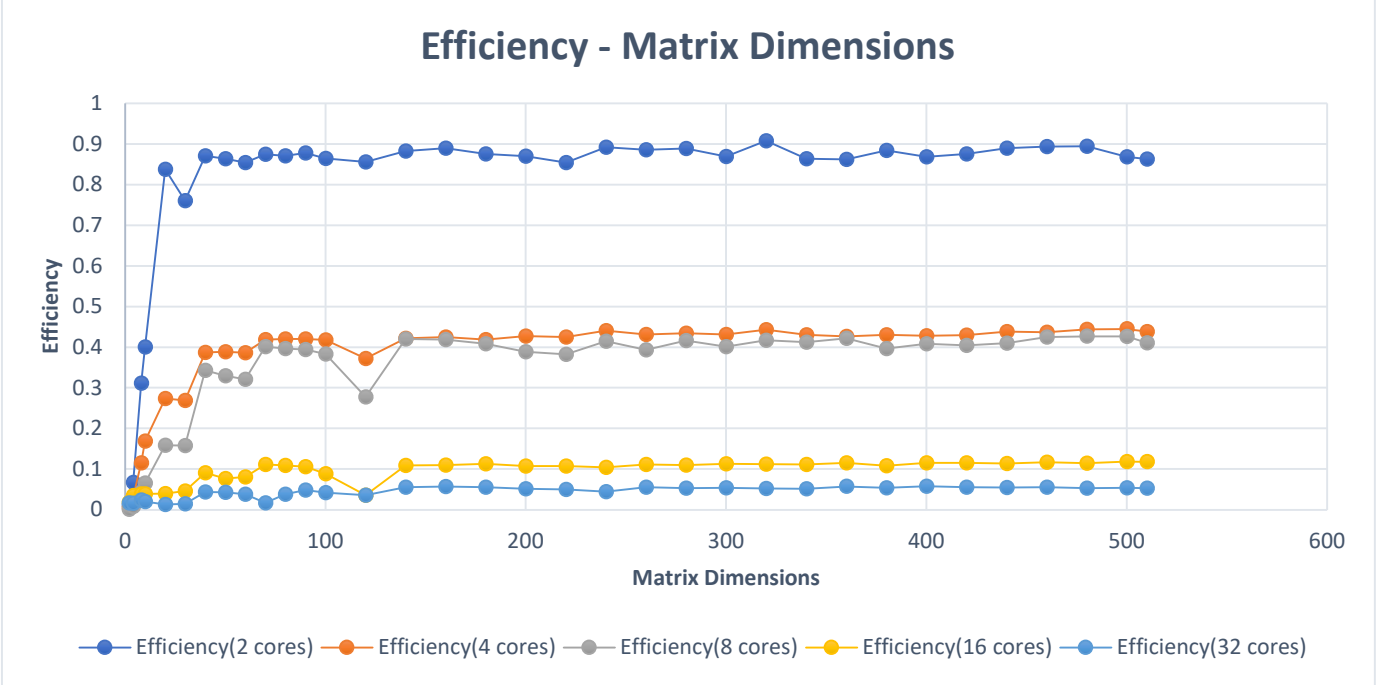


With the blue line, the runtime for two cores can be seen, while with the orange, gray, yellow and light blue the time for 4,8,16,32 nodes is displayed. It can be seen that maximum speedup is achieved using 8 nodes, and that for more than 8 nodes, more speedup can not be achieved, due to restrictions of this implementation. It is possible that a different implementation could utilize more efficiently the number of available nodes.

Finally, the results for efficiency can be viewed, firstly with some indicative values and then using more datapoints with a graph.

	Number of nodes				
Matrix Dimensions	2	4	8	16	32
50	0.863898442	0.38823984	0.329139047	0.076917092	0.042650298
100	0.864662578	0.417378768	0.382782487	0.088480242	0.042301613
200	0.869675143	0.426867548	0.388809675	0.107082915	0.051639531
300	0.869477963	0.430975289	0.401037601	0.113035098	0.053737311
400	0.868278716	0.428353859	0.408138281	0.115284889	0.057351894
510	0.863270344	0.438329233	0.410488025	0.118203551	0.05334356

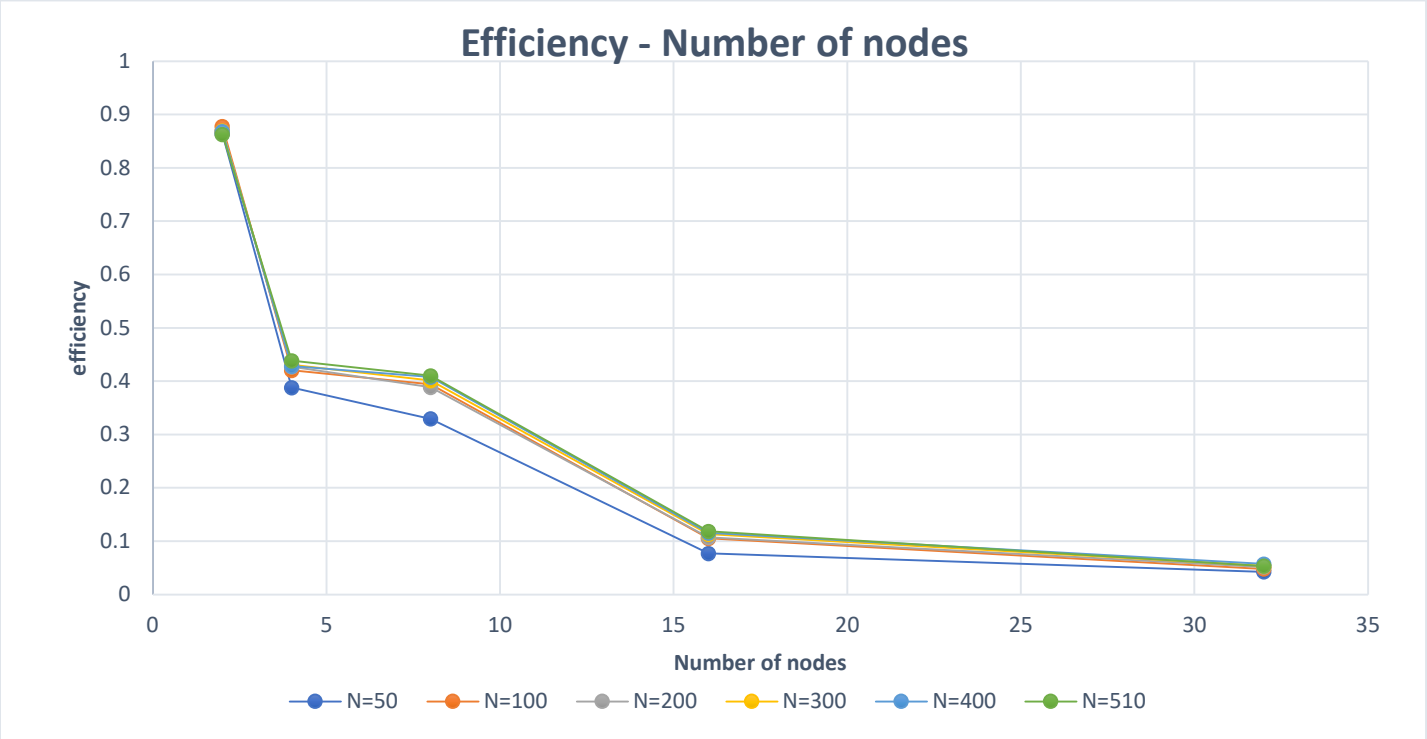
According to the numbers displayed above, we can observe that while the matrix dimension gets larger, and with an increase in the available nodes, the efficiency of this algorithm is reduced. This can be explained by the fact that as the nodes increase, they compute less and less matrices, so they stay inactive for larger periods of time. Furthermore, because P_i are sent to the root node, the rest of the nodes remain unutilized for the rest of the runtime, while they wait for root to calculate the submatrices $C_{i,j}$, οι υπόλοιποι πυρήνες είναι ανεκμετάλλευτοι από το πρόγραμμα. Finally, efficiency in connection with the matrix dimensions can be seen below:



Conclusions

Parallelizing Strassen's algorithm is not an easy task, Η παραλληλοποίηση του συγκεκριμένου αλγορίθμου δεν είναι εύκολη, compared with conventional multiplication. More than one implementations perform different in the same situations, as the algorithm results are dependent on lots of factors such as the matrices dimensions, the number of available cores, the hardware specifications, the code used etc. In general, it can be said

that Strassen's algorithm is better utilized using matrices of dimensions 1000x1000 and more. Furthermore, using the following plot:



We can say that the algorithm is not scalable, because with the increase of nodes, the efficiency is reduced. However, this statement applies for this specific implementation, as the number of cores are not efficiently utilized.

We expect that the Strassen's algorithm could be more difficult to implement using OpenMP, due to the algorithm's recursive nature and seeing as the memory used in OpenMP is shared and more frequently accessed. The frequent access could create the problem of data race condition.

References

- Lecture notes of «Parallel Algorithms & Computational Complexity» by mr. Iraklis Spiliotis
- Introduction to Parallel Programming, Peter Pacheco
- Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- https://en.wikipedia.org/wiki/Strassen_algorithm ,Wikipedia
- https://github.com/psakoglou/Strassen-Algorithm-Simulation-and-Asymptotic-Efficiency-CPP/blob/master/Strassen's%20Algorithm%20code/Strassen.hpp , Pavlos Sakoglou
- Parallel Implementation of Strassens Matrix Multiplication Algorithm, By Karthik Venkataramana Pemmaraju