ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
DEMOCRITUS UNIVERSITY OF THRACE

Τμήμα Ηλεκτρολόγων και Μηχανικών Υπολογιστών

Εργαστήριο Αρχιτεκτονικής Υπολογιστών και Συστημάτων Υψηλών Επιδόσεων

Πολλαπλασιασμός πινάκων με τον αλγόριθμο του Strassen σε περιβάλλον MPI.

Μαζαράκης Περικλής Α.Μ. 57595

Παπαδόπουλος Αριστείδης Α.Μ. 57576

Τσαπέκος Θεόδωρος Α.Μ. 57681

Ακαδημαϊκό έτος 2021-2022

Θεωρητικό υπόβαθρο του σειριακού αλγορίθμου

Ο αλγόριθμος του Strassen είναι ένας αλγόριθμος για τον γρηγορότερο πολλαπλασιασμό δύο πινάκων σε σχέση με τον συμβατικό αλγόριθμο πολλαπλασιασμού τους. Εκεί που ο συμβατικός πολλαπλασιασμός έχει πολυπλοκότητα $O(n^3)$ ο αλγόριθμος του Strassen έχει $O(n^{2.80})$. Το κλειδί του αλγορίθμου είναι πως αντί για 8 αναδρομικούς πολλαπλασιασμούς, κάνει μόνο 7, πράγμα που μπορεί να μη φαίνεται σημαντικό με την πρώτη ματιά, αλλά για μεγάλες διαστάσεις πινάκων εξοικονομεί σημαντική ποσότητα χρόνου. Για να συμβεί κάτι τέτοιο όμως, χρειάζεται να πραγματοποιηθούν πολύ περισσότερες προσθαφαιρέσεις εις βάρος του ενός πολλαπλασιασμού. Ο αλγόριθμος του Strassen απαρτίζεται από 4 βασικά βήματα:

1. Οι πίνακες εισόδου A και B και ο πίνακας εξόδου C διαιρούνται σε $^{n}/_{2} \times {}^{n}/_{2}$ υποπίνακες.
2. Δημιουργούνται 10 πίνακες S_1, S_2…,S_{10}, κάθε ένας από τους οποίους είναι $^{n}/_{2} \times {}^{n}/_{2}$ και είναι το άθροισμα ή η διαφορά των 2 πινάκων από το 1ο βήμα.
3. Χρησιμοποιώντας τους υποπίνακες του 1ου και 2ου βήματος, δημιουργούνται αναδρομικά 7 νέοι πίνακες P_1,P_2…,P_7. Και αυτοί οι πίνακες είναι της μορφής $^{n}/_{2} \times {}^{n}/_{2}$.
4. Τέλος, υπολογίζονται οι επιθυμητοί υποπίνακες C_{11},C_{12}, C_{21},C_{22} που αποτελούν τον πίνακα C, από προσθαφαιρέσεις συνδυασμών των P_i πινάκων.

Έστω, λοιπόν, ο πολλαπλασιασμός δύο πινάκων A και B που έχουν ως αποτέλεσμα έναν πίνακα C γίνεται με τη βοήθεια των πινάκων που θα ακολουθήσουν:

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

Οι 10 πίνακες του 2ου βήματος:

$S_1 = B_{12} - B_{22}$ $\quad$ $S_6 = B_{11} + B_{22}$

$S_2 = A_{11} + A_{12}$ $\quad$ $S_7 = A_{12} - A_{22}$

$S_3 = A_{21} + A_{22}$ $\quad$ $S_8 = B_{21} + B_{22}$

$S_4 = B_{21} - B_{11}$ $\quad$ $S_9 = A_{11} - A_{21}$

$S_5 = A_{11} + A_{22}$ $\quad$ $S_{10} = B_{11} + B_{12}$

Οι 7 πολλαπλασιασμοί του 3ου βήματος:

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

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

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

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

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

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

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

Έτσι, καταλήγουμε στους υποπίνακες που αποτελούν τον πίνακα C του 4ου βήματος:

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

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

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

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

Ουσιαστικά, ο αλγόριθμος οδηγεί στον κατακερματισμό των πινάκων φτάνοντας σε απλά γινόμενα στοιχείων και στη συνέχεια με κατάλληλους συνδυασμούς τους, δημιουργεί τους υποπίνακες C_{11}, C_{12}, C_{21}, C_{22}.

Ο παραπάνω αλγόριθμος, παρά το γεγονός ότι έχει ελάχιστα χαμηλότερη υπολογιστική πολυπλοκότητα, εμφανίζει τα εξής μειονεκτήματα:

- Οι δύο πίνακες πρέπει να είναι τετραγωνικοί.
- Λόγω της αναδρομικής φύσης του, απαιτείται μεγάλη ποσότητα μνήμης.
- Για πίνακες μικρών διαστάσεων, εμφανίζει χειρότερα αποτελέσματα από τον κανονικό πολλαπλασιασμό πινάκων. Προκειμένου να καρπωθούν τα οφέλη του αλγορίθμου, πρέπει οι πίνακες να έχουν διαστάσεις 1000x1000 και πάνω. Για μικρότερους πίνακες ο κλασικός πολλαπλασιασμός τείνει να είναι γρηγορότερος.
- Στην πράξη, ο αλγόριθμος αυτός είναι πιο αποδοτικός από ένα όριο και πάνω, το οποίο εξαρτάται από την υλοποίηση του και το υλικό στο οποίο εκτελείται αυτή.
- Η μείωση των πράξεων έχει ως συνέπεια την μειωμένη ευστάθεια του αλγορίθμου.
- Αυξάνοντας την συνθετότητα της υλοποίησης μπορούν να ξεπεραστούν μερικά από τα προαναφερθέντα μειονεκτήματα.

Σειριακή Υλοποίηση

Η παραπάνω διαδικασία υλοποιείται ως εξής:

```
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;
}
```

Αρχικά, λοιπόν, γίνεται η ανάθεση της μνήμης στους πίνακες A και B. Στη συνέχεια, οι A και B διαιρούνται σε υποπίνακες με διάσταση $^{n}/_{2} \times {}^{n}/_{2}$ και ακριβώς από κάτω γίνεται άλλη

μια ανάθεση μνήμης στους πίνακες S_1, S_2…,S_{10} και ο υπολογισμός τους βάση των τύπων που προαναφέρθηκαν. Για τον υπολογισμό των P_1,P_2…,P_7, υπάρχει αναδρομή αφού για τον υπολογισμό του κάθε P_i επικαλείται η συνάρτηση **Strassen**, την οποία διατρέχουμε όλη αυτή την ώρα. Εφόσον υπολογιστούν τα P_i καθαρίζεται η αχρείαστη μνήμη και δημιουργείται εκ νέου χώρος για τους πίνακες C_{11}, C_{12}, C_{21}, C_{22} όπου και υπολογίζονται βάση του 4^{ου} βήματος και στη συνέχεια αναθέτονται στην κατάλληλη θέση για τη δημιουργία του τελικού πίνακα C.

Τρέχοντας τον παραπάνω αλγόριθμο, για N = 4, προκειμένου να γίνει κατανοητό αν βγάζει τα ίδια αποτελέσματα ο αλγόριθμος Strassen, εμφανίζονται τα ακόλουθα:

```
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
```

Όπου η υλοποίηση του αλγορίθμου φαίνεται δουλεύει σωστά. Φυσικά, με μεγαλύτερες διαστάσεις πινάκων, δεν μπορεί να εκτυπωθεί το τελικό αποτέλεσμα.

Παράλληλη Υλοποίηση

Υπάρχουν πολλοί τρόποι να εφαρμοστεί η παραλληλία στον αλγόριθμο του Strassen, ωστόσο εξερευνήθηκαν δύο. Αρχικά, πρέπει να αποσταλούν οι αρχικοί πίνακες από τον root στους υπόλοιπους κόμβους. Στην συνέχεια, όλοι (root + υπόλοιποι κόμβοι) χωρίζουν και δημιουργούν τους υποπίνακες $A_{i,j}$, $B_{i,j}$. Έπειτα, σπάει ο υπολογισμός των P_i , ανάλογα με το πλήθος των κόμβων που χρησιμοποιούνται και ο κάθε κόμβος υπολογίζει από έναν τέτοιο πίνακα. Εν συνεχεία, στέλνονται πίσω στον root τα P_i και αυτός υπολογίζει τους υποπίνακες $C_{i,j}$ και άρα το τελικό αποτέλεσμα. Δυνητικά, θα ήταν δυνατή επαυξημένη παραλληλία στην υλοποίηση αυτή, με τρόπο τέτοιο ώστε και οι υποπίνακες $A_{i,j}$ να σπάνε σε υποπίνακες ανάλογα με το μέγεθος τους κ.ο.κ., όμως τότε θα αύξανε υπερβολικά η συνθετότητα του κώδικα.

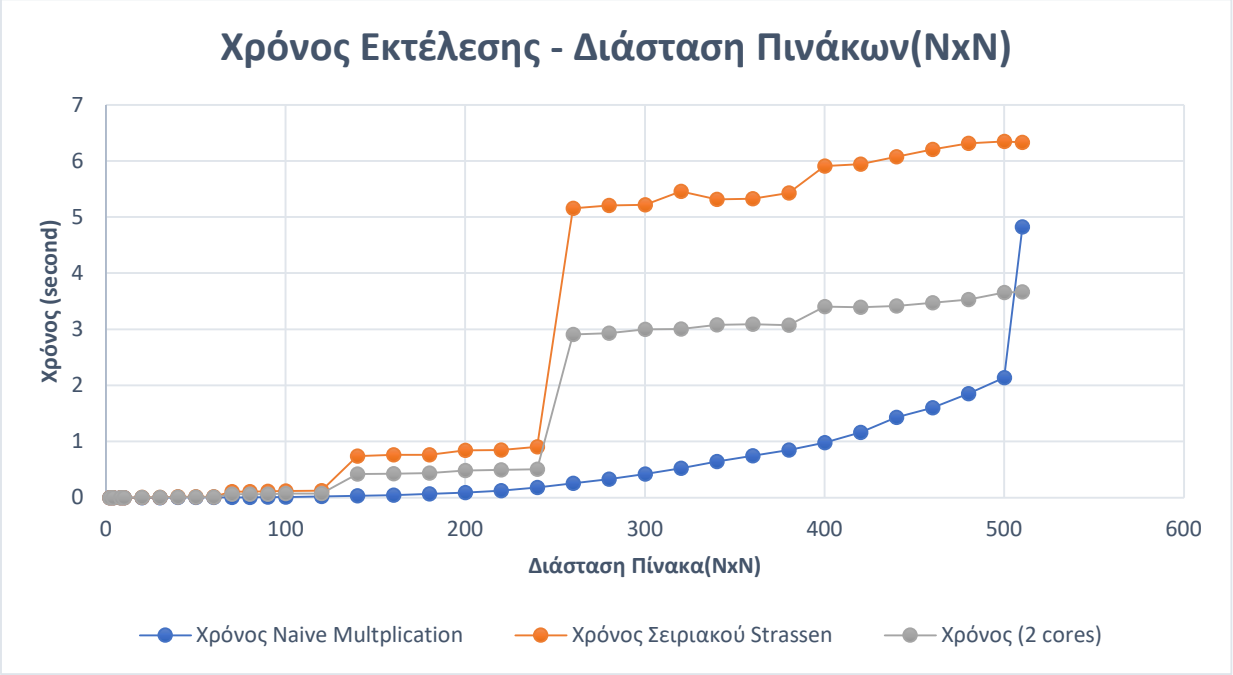
Η 2^{η} υλοποίηση που εξερευνήθηκε, ακολουθεί τα παραπάνω βήματα, αλλά αφού υπολογιστούν τα P_i, τέσσερις από τους διαθέσιμους κόμβους (ή 2, αν δεν υπάρχουν 4) υπολογίζουν ο καθένας τους από ένα $C_{i,j}$ (ή από 2) και στην συνέχεια μαζεύει ο root αυτούς τους υποπίνακες και συνθέτει τον πίνακα του τελικού αποτελέσματος. Ωστόσο,

αυτή η εκδοχή δεν χρησιμοποιήθηκε, διότι το κόστος της επικοινωνίας υπερνικούσε τα οφέλη που κερδίζονταν από την κατάτμηση του υπολογισμού του πίνακα C.

Έτσι, χρησιμοποιήθηκε η 1η υλοποίηση που αναλύθηκε, εξετάζοντας διάφορους πίνακες με μέγεθος έως και 510x510. Έπειτα από αυτό το μέγεθος, λόγω της τεράστιας απαίτησης σε μνήμη, εμφανίζονταν σφάλμα και ήταν αδύνατη η συνέχεια, για την δεδομένη υλοποίηση. Ιδανικά, πίνακες με διάσταση ίση με κάποια δύναμη του 2, θα βοηθήσει στην ισομερή κατανομή του φόρτου στους επεξεργαστές, χωρίς να είναι απαραίτητη η προϋπόθεση αυτή για να χρησιμοποιηθεί ο αλγόριθμος. Στον παρακάτω πίνακα απεικονίζονται οι χρόνοι εκτέλεσης (σε second) για διάφορες τιμές πυρήνων και μεγέθους πινάκων.

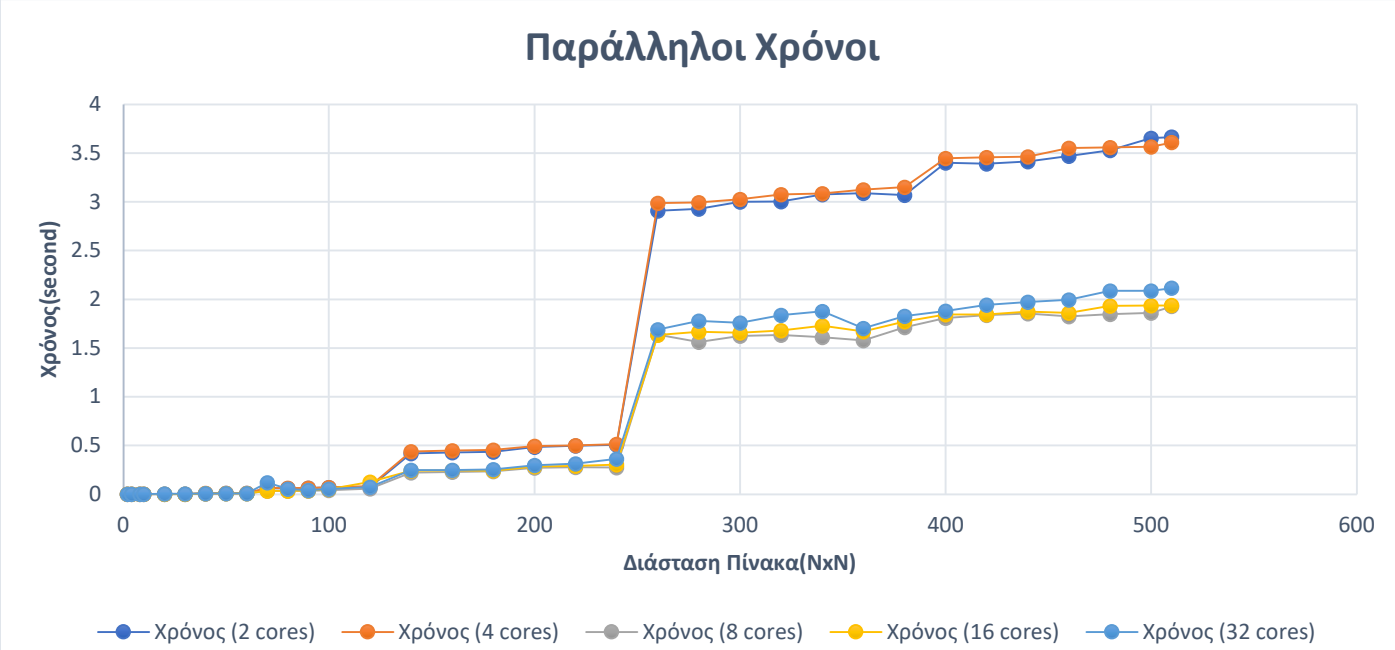
	Πλήθος Πυρήνων					
Διάσταση πινάκων	1	2	4	8	16	32
50	0.017081	0.009886	0.010999	0.006487	0.008033	0.008059
100	0.119569	0.069142	0.071619	0.039046	0.04884	0.052908
200	0.842056	0.484121	0.49316	0.270716	0.282562	0.298439
300	5.215494	2.99921	3.025402	1.625625	1.65834	1.75937
400	5.90756	3.40188	3.447827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

Στο ακόλουθο γράφημα, αποτυπώνεται ο χρόνος εκτέλεσης για διάφορα μεγέθη πινάκων, χρησιμοποιώντας τον απλό πολλαπλασιασμό, τον αλγόριθμο Strassen σε έναν κόμβο και σε δύο(δηλαδή παράλληλα).



Σύμφωνα με το παραπάνω διάγραμμα, και έχοντας ως δεδομένο ότι η μέγιστη διάσταση πινάκων που μπορούσαν να δοκιμαστούν ήταν 510x510, φαίνεται πως ο απλός πολλαπλασιασμός (naive) είναι πιο αποδοτικός από τον σειριακό Strassen, αλλά φαίνεται

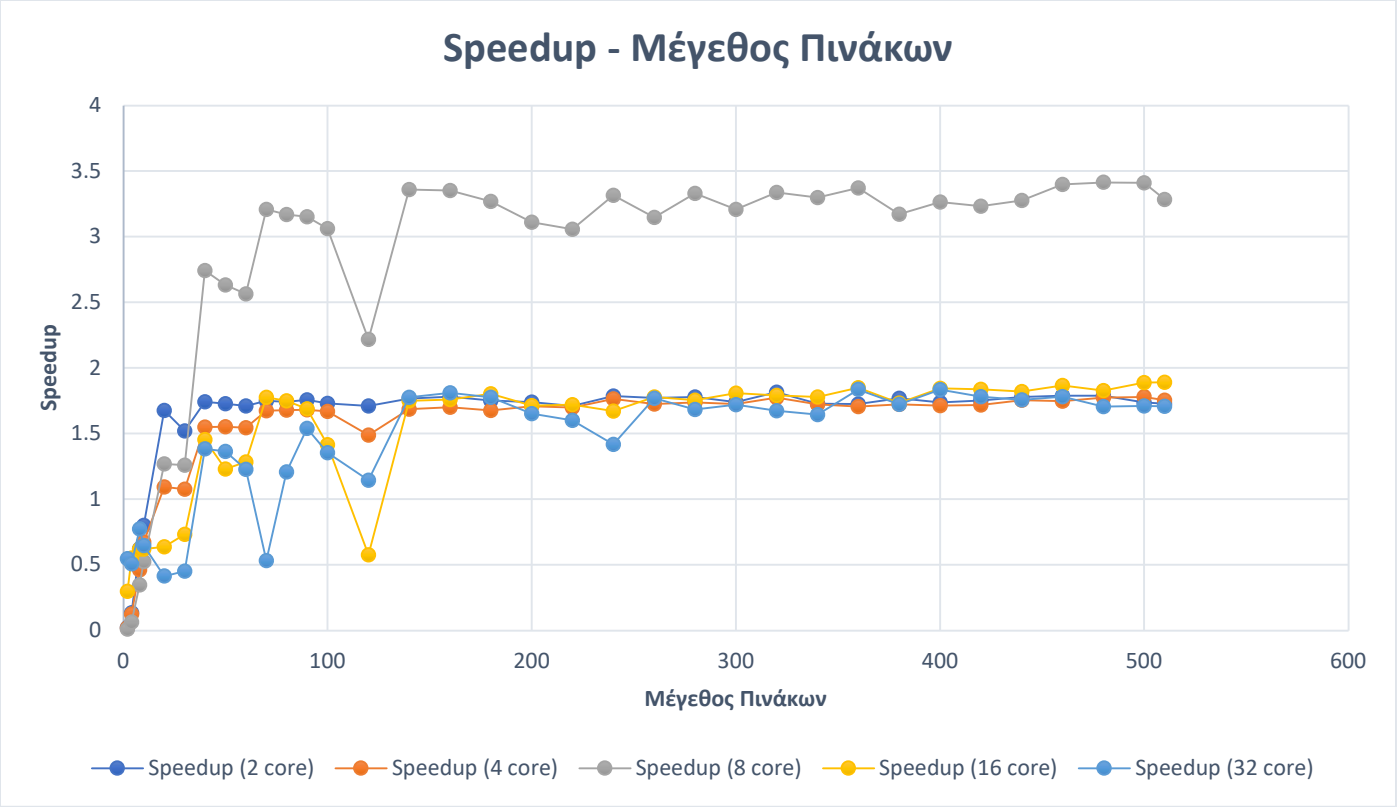
πως τείνει να τον ξεπεράσει μετά από την τιμή N=510. Επιπλέον, ο παράλληλος αλγόριθμος με χρήση δύο πυρήνων είναι πιο γρήγορος για αυτήν την διάσταση, επομένως μπορεί να γίνει η υπόθεση πως με πίνακες μεγαλύτερων διαστάσεων, ο αλγόριθμος είναι πιο γρήγορος από τον απλό πολλαπλασιασμό. Στο επόμενο γράφημα παρατηρείται η εξέλιξη των χρόνων υπολογισμού του παράλληλου αλγορίθμου, για 2,4,8,16,32 υπολογιστικούς κόμβους.



Με βάση το παραπάνω σχήμα, συμπεραίνεται πως με την χρήση έως και 8 πυρήνων επιταχύνεται το πρόγραμμα, όμως με την χρήση παραπάνω παραμένει σε ίδια επίπεδα. Αυτό πιθανόν οφείλεται σε δύο παράγοντες, ότι χρειάζεται παραπάνω διαστάσεις για να χρησιμοποιηθούν ή ότι πιθανόν η υλοποίηση δεν μπορεί να τους αξιοποιήσει όπως πρέπει. Στον επόμενο πίνακα, αποτυπώνεται η επιτάχυνση (speedup) και πάλι συναρτήσει της διάστασης του προβλήματος και του πλήθους των πυρήνων.

	Πλήθος Πυρήνων				
Διάσταση πινάκων	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

Ενώ χρησιμοποιώντας περισσότερα σημεία, αποτυπώνεται γραφικά ως εξής:



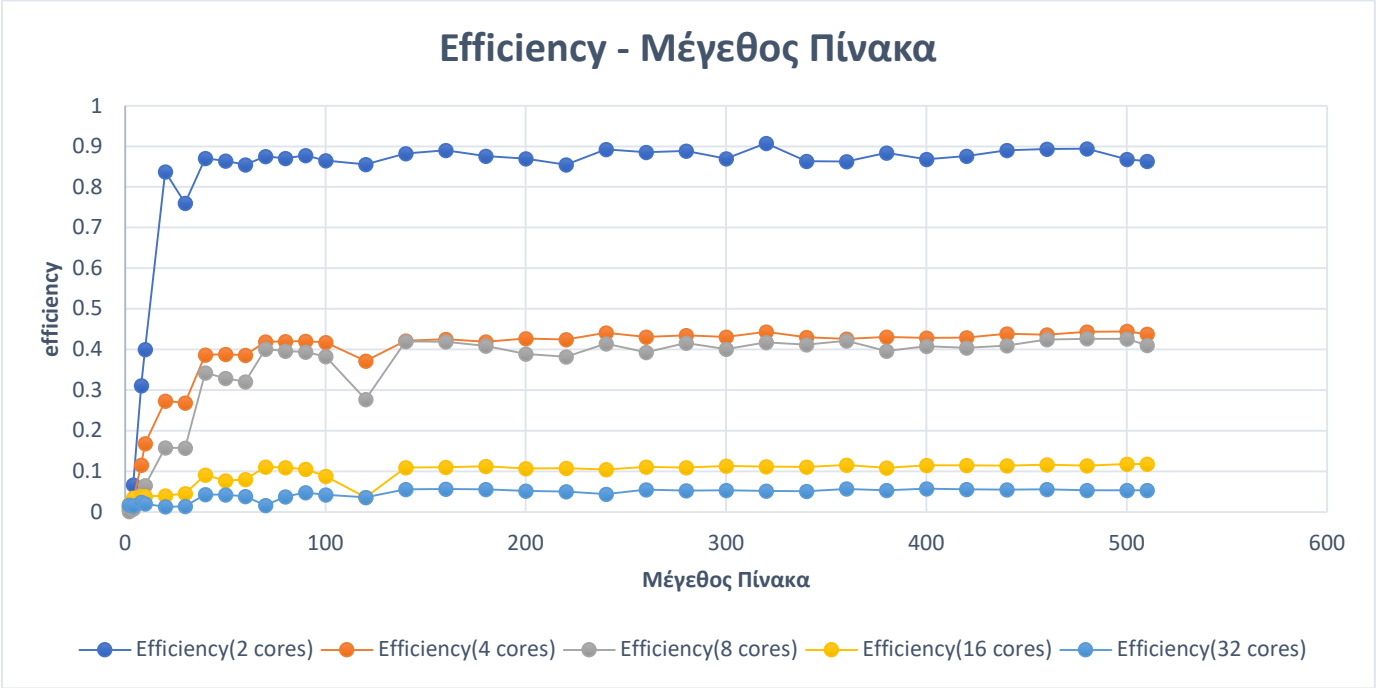
Με βάση τα παραπάνω, η μέγιστη επιτάχυνση επιτυγχάνεται για 8 πυρήνες, ενώ, λόγω της υλοποίησης του κώδικα, για περισσότερο από 8 πυρήνες, επειδή δεν αξιοποιούνται κατάλληλα από τον κώδικα. Με διαφορετική υλοποίηση, μπορούν να αξιοποιηθούν καλύτερα, αλλά χάνεται η ευστοχία της συγκεκριμένης υλοποίησης.

Τέλος, ακολουθούν τα αποτελέσματα για την αποδοτικότητα, πρώτα με την μορφή πίνακα για μερικές τιμές και μετά με την μορφή γραφήματος.

	Πλήθος Πυρήνων				
Διάσταση πινάκων	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

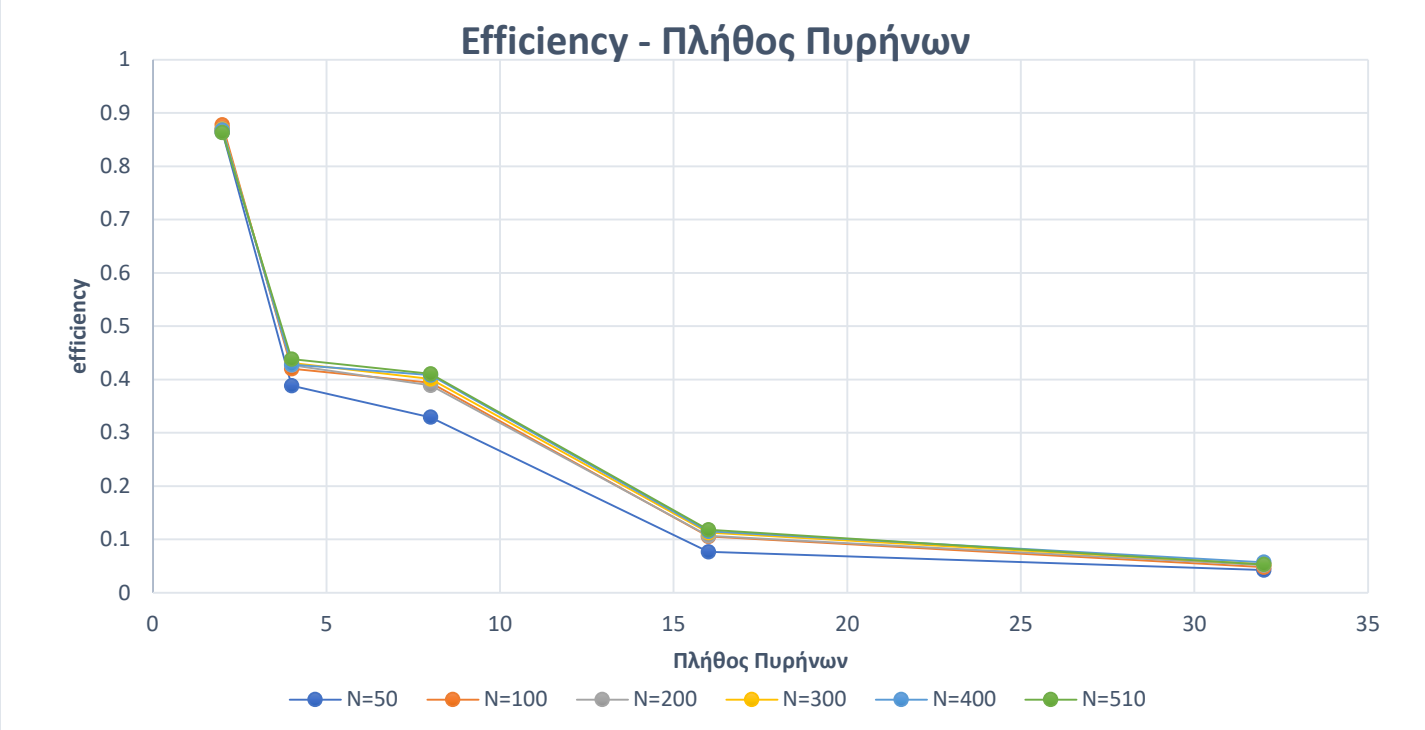
Σύμφωνα με τα παραπάνω νούμερα, παρατηρούμε πως όσο αυξάνονται οι πυρήνες, με αντίστοιχη αύξηση στις διαστάσεις του πίνακα , μειώνεται η αποδοτικότητα της συγκεκριμένης υλοποίησης, διότι πλέον πολλοί πυρήνες υπολογίζουν λιγότερα πράγματα και άρα παραμένουν αδρανής για περισσότερο χρονικό διάστημα. Επιπλέον, επειδή τα P_i στέλνονται στον root για να υπολογιστούν οι υποπίνακες $C_{i,j}$, οι υπόλοιποι πυρήνες είναι

ανεκμετάλλευτοι από το πρόγραμμα. Ενώ το efficiency σε σχέση με το μέγεθος του πίνακα δίνεται από:



Συμπεράσματα

Η παραλληλοποίηση του συγκεκριμένου αλγορίθμου δεν είναι εύκολη, σε σχέση με τον συμβατικό πολλαπλασιασμό. Δεν υπάρχει συγκεκριμένη υλοποίηση που να αποδίδει το ίδιο ικανοποιητικά σε κάθε περίπτωση, καθώς τα αποτελέσματα του αλγορίθμου εξαρτώνται από πολλούς παράγοντες(π.χ. μέγεθος πινάκων, πλήθος επεξεργαστών, μέγεθος διαθέσιμης μνήμης, υλοποίηση κώδικα κ.α.). Γενικότερα πάντως, ο αλγόριθμος του Strassen, όπως έχει ήδη αναφερθεί, αποδίδει για πίνακες με διαστάσεις μεγαλύτερες από 1000. Επιπλέον, βασιζόμενοι στο εξής γράφημα:



Μπορούμε να πούμε, πως ο αλγόριθμος δεν είναι επεκτάσιμος, καθώς με την προσθήκη παραπάνω κόμβων, η αποδοτικότητα μειώνεται. Ειδικότερα, αυτή η διαπίστωση ισχύει για την παρούσα υλοποίηση, που δεν αξιοποιεί στο μέγιστο τους πλεονάζοντες πυρήνες/κόμβους.

Εκτιμούμε πως ο παραπάνω αλγόριθμος θα έχει περισσότερες δυσκολίες αν υλοποιηθεί στο OpenMP, καθώς η μνήμη είναι μία, και λόγω της αναδρομικής φύσης του αλγορίθμου απαιτείται συχνή προσπέλαση αυτής, καθώς και μεγάλο μέγεθος. Η συχνή προσπέλαση μπορεί να προκαλέσει συνθήκες ανταγωνισμού μεταξύ των νημάτων.

Βιβλιογραφία

- Διαφάνειες μαθήματος «Παράλληλοι αλγόριθμοι & Υπολογιστική Πολυπλοκότητα» από κ. Ηρακλή Σπηλιώτη
- 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