ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ροή Υ : Συστήματα Παράλληλης Επεξεργασίας

2η Άσκηση (Τελική Αναφορά)

Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου Floyd-Warshall σε Αρχιτεκτονικές Κοινής Μνήμης

Κριθαρούλας Διονύσιος 03117875

Ομάδα:Parlab18

Εξάμηνο: 9ο

Πίνακας περιεχομένων

Εισαγωγή

Η συγκεκριμένη εργαστηριακή άσκηση έχει ως στόχο την ανάπτυξη διαφορετικών παράλληλων εκδόσεων του αλγορίθμου FloydWarshall, την αξιολόγηση της παραγωγικότητας (productivity) ανάπτυξης παράλληλου κώδικα και την τελική επίδοση του παράλληλου προγράμματος, επιλέγοντας ένα από τα δύο προγραμματιστικά εργαλεία για αρχιτεκτονικής κοινής μνήμης: OpenMP ή Threading Building Blocks (TBBs).

Ερώτημα 1 (Έχουν προστεθεί ορισμένα σχόλια σε σχέση με την ενδιάμεση αναφορά)

Ο αλγόριθμος Floyd-Warshall(FW) υπολογίζει τα ελάχιστα μονοπάτια ανάμεσα σε όλα τα ζεύγη κορυφών ενός γράφου (all-pairs shortest path). Για τον σκοπό αυτό χρησιμοποιεί έναν πίνακα γειτνίασης A ο οποίος αναπαριστά τον κατευθυνόμενο γράφο του προβλήματος. Πιο συγκεκριμένα ο πίνακας A είναι δισδιάστατος και σε κάθε θέση του i,j περιέχει το βάρος της ακμής (μπορεί να είναι και αρνητικό) που ενώνει τους κόμβους i,j (εάν η ακμή αυτή υπάρχει).

Για τον αλγόριθμο αυτό ,πέρα από την standard έκδοση του, μας δίνονται και επιπλέον 2 υλοποιήσεις, μια recursive και μια tiled οι οποίες προσπαθούν να αξιοποιήσουν καλύτερα την κρυφή μνήμη. Στην συνέχεια αναλύεται η κάθε έκδοση ξεχωριστά μαζί με σχεδιαστικές επιλογές , επιλογές υλοποίησης της παραλληλοποίησης αλλά και το πως αυτές οι επιλογές ενδέχεται να επηρεάσουν την απόδοση του αλγορίθμου.

Standard Έκδοση του Αλγορίθμου

Σύμφωνα με την Standard του έκδοση ο αλγόριθμος του Floyd-Warshall(FW) έχει ως εξής:

```
for (k=0; k<N; k++)
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         A[i][j] = min(A[i][j], A[i][k]+A[k][j]);
```

Όπως φαίνεται και από τον κώδικα η λογική της standard προσέγγισης είναι πως σε κάθε χρονικό βήμα k υπολογίζεται για όλα τα ζεύγη κόμβων i,j το συντομότερο

μονοπάτι από τον κόμβο i στον κόμβο j το οποίο περνάει από τους κόμβους 0 μέχρι k (όχι κατ' ανάγκη όλους).

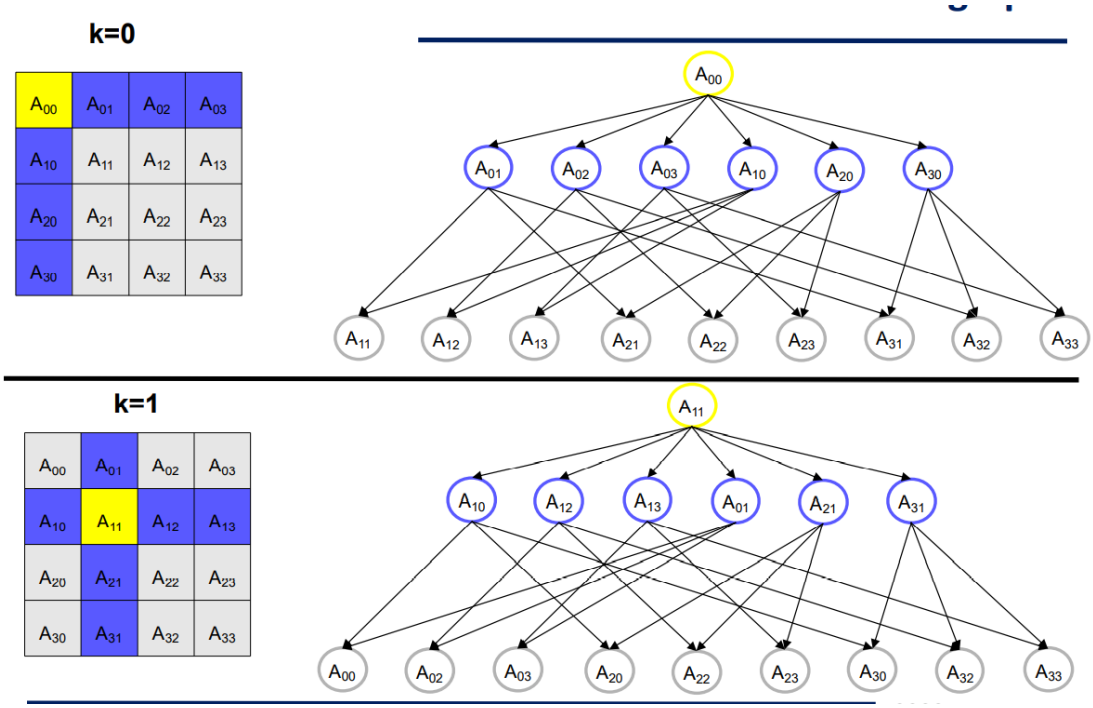
Κατανομή των Υπολογισμών

Αρχικά λοιπόν πρέπει να αποφασίσουμε πως θα κατανείμουμε τους υπολογισμούς μας. Η επιλογή μίας task centric προσέγγισης θα μας βοηθήσει να κατανοήσουμε καλύτερα την λειτουργία του αλγορίθμου και να μπορέσουμε να δούμε τις δυνατότητες παραλληλισμού του. Έτσι λοιπόν επιλέγουμε τον ακόλουθο ορισμό:

1 task: Ο υπολογισμός ενός στοιχείου A[i][j] του πίνακα γειτνίασης A για ένα συγκεκριμένο χρονικό βήμα k.

Ορισμός ορθής σειράς εκτέλεσης

Κάνοντας την παραπάνω επιλογή προκύπτει το ακόλουθο task graph για τον αλγόριθμο μας θεωρώντας ότι έχουμε έναν πίνακα γειτνίασης A μεγέθους 4x4:



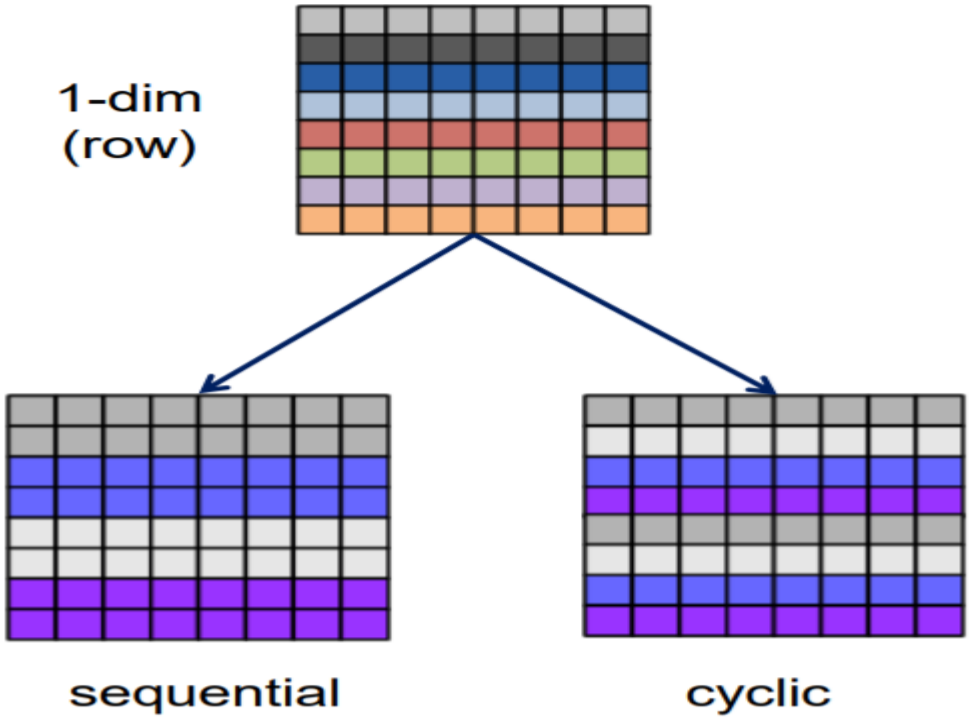
Σχόλιο: Το παραπάνω task-graph μας δίνεται στην παρουσίαση της εργαστηριακής άσκησης.

Σύμφωνα με το παραπάνω task graph λοιπόν παρατηρούμε ότι **δεν υπάρχει η δυνατότητα παραλληλοποίησης ως προς τα χρονικά βήματα k**. Αυτό οφείλεται στο γεγονός πως σε κάθε χρονικό βήμα k κάθε στοιχείο του πίνακα γειτνίασης A προκειμένου να υπολογίσει την νέα τιμή του , χρειάζεται σίγουρα την τιμή που απέκτησε με το πέρας του προηγούμενου χρονικού βήματος k-1. Επομένως δεν μπορεί να παραλληλοποιηθεί ο συγκεκριμένος αλγόριθμος ως προς τα χρονικά βήματα k. Αντίθετα παρατηρούμε ότι υπάρχει δυνατότητα παραλληλοποίησης ως προς τους δείκτες i,j καθώς κάθε χρονική στιγμή k το στοιχείο A[i,j] χρειάζεται τα στοιχεία A[i][j] , A[i][k] και A[k][j] που έχουν υπολογισθεί όμως την προηγούμενη χρονική στιγμή k-1.

Παρατήρηση: Παρατηρώντας τις στοιχειώδεις πράξεις του κώδικα βλέπουμε ότι στην πραγματικότητα αποτελείται από περισσότερες εξαρτήσεις από αυτές που παρουσιάζονται στον παραπάνω γράφο εξαρτήσεων (task graph). Ο γράφος αυτός θεωρούμε ότι προέρχεται από την γνώση μας ότι ο παραπάνω αλγόριθμος γράφεται ισοδύναμα:
$A^k[i][j] = \min(A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j])$, όπου $A^k[i][j]$ αναφέρεται στην χρονική στιγμή k ενώ $A^{k-1}[i][j]$, $A^{k-1}[i][k]$, $A^{k-1}[k][j]$ αναφέρεται στην χρονική στιγμή k-1.

Ανάθεση εργασιών σε οντότητες εκτέλεσης

Μια πρώτη επιλογή για την ανάθεση των εργασιών σε οντότητες εκτέλεσης είναι κάθε νήμα να αναλάβει τον υπολογισμό των στοιχείων ορισμένων γραμμών του πίνακα A για ένα χρονικό βήμα k. Η επιλογή αυτή είναι συμβατή με την ύπαρξη της παραλληλίας που εντοπίσαμε στο προηγούμενο βήμα. Η ανάθεση αυτή μπορεί να γίνει στατικά. Με άλλα λόγια κάθε νήμα αναλαμβάνει έναν συγκεκριμένο και προκαθορισμένο αριθμό γραμμών του πίνακα A (chunk) και εκτελεί όλους τους υπολογισμούς που σχετίζονται με τα στοιχεία των συγκεκριμένων γραμμών του πίνακα. Στην περίπτωση που επιλέξουμε την στατική ανάθεση έχουμε δύο υποπεριπτώσεις. Είτε κυκλικά κάθε νήμα να αναλαμβάνει μια γραμμή του πίνακα A είτε συνεχόμενα το ίδιο νήμα να αναλάβει έναν αριθμό από γραμμές του πίνακα. Οι δύο υποπεριπτώσεις φαίνονται στην συνέχεια σχηματικά:



Πέρα από την στατική ανάθεση των εργασιών σε νήματα υπάρχει και η δυνατότητα της δυναμικής ανάθεσης όπου χωρίζουμε τις γραμμές του πίνακα σε chunks (=σύνολο γραμμών) και κάθε νήμα με το που τελειώσει την εκτέλεση μίας εργασίας (στην προκειμένη περίπτωση εκτέλεση των υπολογισμών για όλα τα στοιχεία ενός chunk του πίνακα Α) ζητάει δυναμικά να του δοθεί η επόμενη προς εκτέλεση εργασία. Η επιλογή αυτή φαίνεται ότι δεν θα προσφέρει κάποια βελτίωση στην επίδοση του αλγορίθμου καθώς οι υπολογισμοί που γίνονται για κάθε chunk του πίνακα Α έχουν το ίδιο κόστος (έχουμε ισοκατανομή του φορτίου μεταξύ των εργασιών). Αντίθετα η δυναμική ανάθεση θα οδηγήσει σε επιβάρυνση προκαλώντας μεγαλύτερο overhead στο runtime σύστημα το οποίο πρέπει να αναθέτει κατά την διάρκεια της εκτέλεσης του προγράμματος εργασίες στα νήματα. Το overhead αυτό θα έχει επίπτωση και στον χρόνο εκτέλεσης του προγράμματος.

Παρόμοια αποτελέσματα θεωρούμε ότι θα επιφέρει επίσης η δυναμική ανάθεση chunks σε νήματα με την διαφορά τώρα ότι η τιμή του chunk δεν είναι σταθερή αλλά ορίζεται ανάλογα με τον ελεύθερο αριθμό των νημάτων. Επομένως θα έχουμε την μείωση των chunks καθώς προχωρούν οι επαναλήψεις. Και αυτή η περίπτωση αποτελεί ένα είδος δυναμικής ανάθεσης η οποία για τους ίδιους λόγους δεν αναμένεται να επιφέρει κάποιο αποτέλεσμα.

Μια τελείως διαφορετική οπτική θα ήταν αντί να χωρίσουμε σε γραμμές τον πίνακα Α να τον χωρίζαμε σε block συγκεκριμένου size. Έτσι μία εργασία θα αφορά πλέον τον υπολογισμό των τιμών του πίνακα Α για το συγκεκριμένο block τον οποίο θα αναλαμβάνουν πλέον τα νήματα εκτέλεσης (μια τέτοια υλοποίηση μπορεί να γίνει με threading building blocks). Η οπτική αυτή υποθέτουμε ότι θα δημιουργήσει πρόβλημα στην τοπικότητα των δεδομένων. Όπως αναφέραμε και προηγούμενος κάθε στοιχείο A[i][j] για να υπολογίσει την νέα του τιμή σε ένα χρονικό βήμα k χρειάζεται τις τιμές A[i][k] και A[k][j] (πέρα από την δικιά του τιμή) στο προηγούμενο χρονικό βήμα. Με την μέθοδο χωρισμού σε γραμμές λοιπόν ήμασταν σίγουροι ότι τουλάχιστον κάθε νήμα θα είχε το δεδομένο A[i][k] (σε ορισμένες περιπτώσεις μπορεί να έχει και το A[k][j].) Αντίθετα ο χωρισμός σε block μπορεί να οδηγήσει σε κάποιες επαναλήψεις τα νήματα να μην έχουν ούτε το A[i][k] ούτε το A[k][j] με αποτέλεσμα να αυξάνονται οι απαιτήσεις για επικοινωνία. Αναμένεται λοιπόν η οπτική αυτή να μην επιφέρει κάποια βελτίωση στον χρόνο εκτέλεσης σε σχέση με την προηγούμενη.

Παρατήρηση : Ο συγκεκριμένος αλγόριθμος (standard υλοποίηση) παρόλο που αναμένουμε να παρουσιάσει μείωση του χρόνου εκτέλεσης με την παραλληλοποίηση του θεωρούμε πως δεν θα κλιμακώνει, ειδικά όσο αυξάνουμε το μέγεθος του πίνακα Α. Κάτι τέτοιο οφείλεται στο γεγονός πως καθώς αυξάνεται το μέγεθος του πίνακα Α, αυτός παύει πλέον να χωράει στην cache. Έτσι σε κάθε χρονικό βήμα k απαιτείται η μεταφορά του πίνακα Α από την κύρια μνήμη στην cache (για να είμαστε ακριβείς ένα κομμάτι του πίνακα και όχι όλος ο πίνακας καθώς ένα κομμάτι του θα παραμένει στην cache). Η μεταφορά αυτή δημιουργεί μεγάλη επιβάρυνση στον χρόνο εκτέλεσης του αλγορίθμου αποτελώντας ένα άνω όριο (bottleneck) στην κλιμάκωση. Αν το σκεφτούμε και με το μοντέλο roofline θα δούμε ότι:

- Ο αλγόριθμος εκτελεί συνολικά N^3 πράξεις.
- Εάν ο πίνακας Α χωράει στην cache τότε έχουμε N^2 δεδομένα (τον φέρνουμε μία φορά από την κύρια μνήμη και τέλος).
- Επομένως το Operational Intensity είναι N^3/N^2 = N flop/byte. Παρατηρούμε λοιπόν ότι προσεγγιστικά σε κάθε byte δεδομένων εκτελούνται N πράξεις κάτι που δείχνει ότι ο αλγόριθμος έχει μεγάλο χώρο για παραλληλία.
- Αντίθετα αν ο πίνακας Α δεν χωράει στην cache θεωρούμε ότι σε κάθε χρονικό βήμα θα πρέπει να έρχεται ολόκληρος από την κύρια μνήμη (προσεγγιστικά όπως αναφέραμε και προηγουμένως). Έτσι θα έχουμε συνολικά $N*N^2 = N^3$ δεδομένα. Επομένως το Operational Intensity γίνεται πλέον N^3/N^3=1flop/byte. Παρατηρούμε ότι είναι μιας τάξης μεγέθους μικρότερο από το προηγούμενο μην δίνοντας πλέον μεγάλες δυνατότητες για μείωση του χρόνου εκτέλεσης μέσω της παραλληλίας καθώς μεγάλος χρόνος του προγράμματος πλέον καταναλώνεται στην μεταφορά των δεδομένων από την κύρια μνήμη (Νόμος του Amdal).

Όπως λοιπόν αναλύσαμε η standard υλοποίηση είναι memory bound. Μία υλοποίηση η οποία βελτιώνει το πρόβλημα αυτό αξιοποιώντας καλύτερα την κρυφή μνήμη είναι η recursive υλοποίηση του αλγορίθμου η οποία αναλύεται στην συνέχεια.

Recursive Υλοποίηση

Η λογική του κώδικα που υλοποιεί η σειριακή recursive υλοποίηση του αλγορίθμου Floyd-Warshall είναι η ακόλουθη:

```
FWR (A, B, C)
  if (base case)
    FWI (A, B, C)
  else
    FWR (A00, B00, C00);
    FWR (A01, B00, C01);
    FWR (A10, B10, C00);
    FWR (A11, B10, C01);
    FWR (A11, B10, C01);
    FWR (A10, B10, C00);
    FWR (A01, B00, C01);
    FWR (A00, B00, C00);
```

```
FWI (A, B, C)
for (k=0; k<N; k++)
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         A[i][j] = min(A[i][j], B[i][k]+C[k][j]);
```

Σύμφωνα με το παραπάνω πρόγραμμα παρατηρούμε ότι καλείται αρχικά η συνάρτηση FWR(A,A,A) όπου A ο πίνακας γειτνίασης. Εφόσον ο πίνακας A δεν έχει το μέγεθος BxB που έχουμε ορίσει εμείς (base case) η συνάρτηση FWR καλείται αναδρομικά 8 φορές παίρνοντας κάθε φορά ως ορίσματα τρία από τα 4 τετραγωνικά block A_{11} , A_{12} , A_{13} , A_{14} στα οποία έχει χωριστεί ο αρχικός πίνακας A. Η σειρά με την οποία καλείται αναδρομικά (παίρνοντας κάθε φορά τα κατάλληλα ορίσματα) είναι σημαντική για την ορθότητα του αλγορίθμου. Όταν πλέον φτάσουμε σε ένα στάδιο της αναδρομής στο οποίο το υπό-block του πίνακα A έχει διαστάσεις BxB καλείται η συνάρτηση FWI η οποία εκτελεί την γνωστή standard υλοποίηση του αλγορίθμου που αναλύσαμε νωρίτερα.

<u>Κατανομή των υπολογισμών</u>

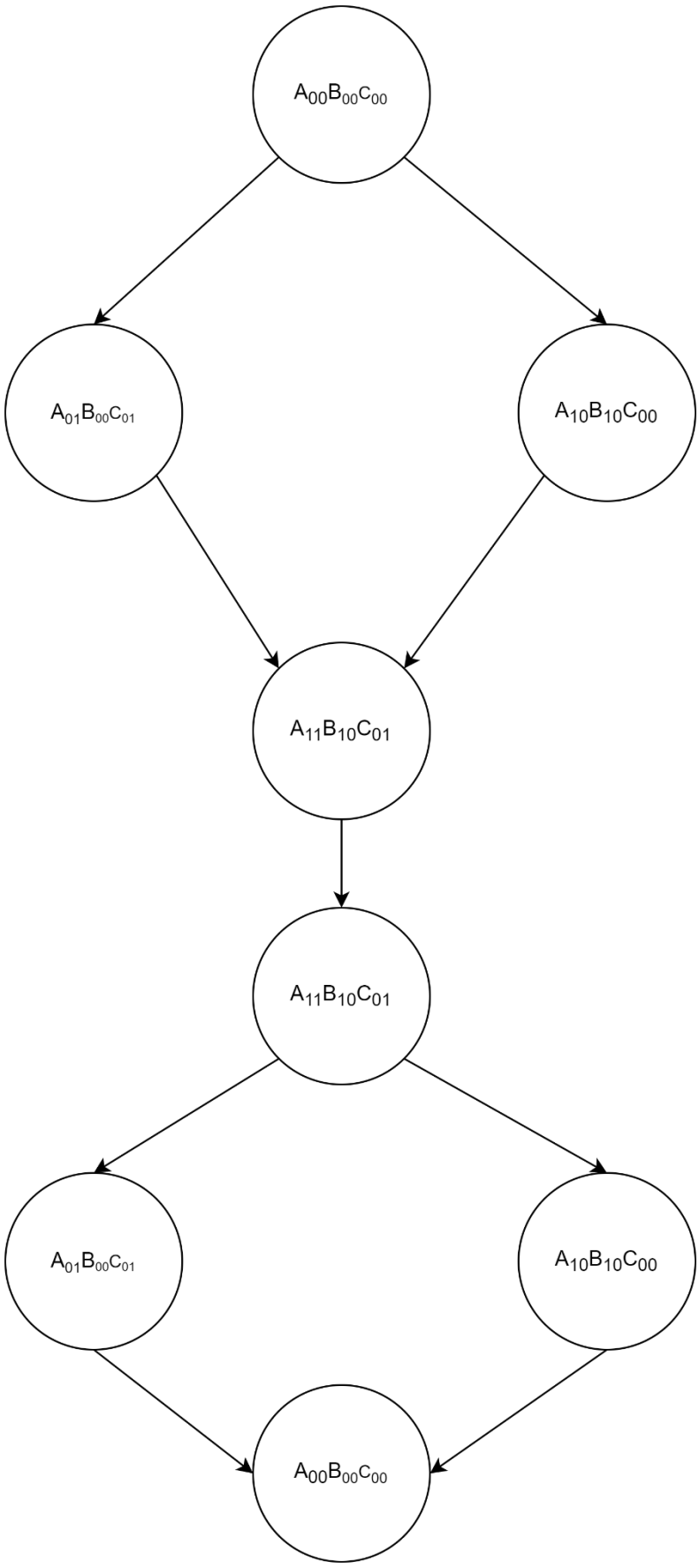
Προκειμένου να κατανοήσουμε την λογική του αλγορίθμου αλλά και τις εξαρτήσεις που εμφανίζονται θα χρησιμοποιήσουμε και εδώ μία ανάλυση που βασίζεται σε μία task centric προσέγγιση. Ως πρώτη προσέγγιση θεωρούμε:

1 task = υπολογισμός της FWR στο πρώτο στάδιο της αναδρομής

Με βάση την επιλογή αυτή και θεωρώντας τον πίνακα Α αρχικά χωρισμένο σε 4 τετραγωνικά block όπως στην συνέχεια:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$$

προκύπτει το ακόλουθο task-graph:


$A_{00}B_{00}C_{00}$
$A_{01}B_{00}C_{01}$
$A_{10}B_{10}C_{00}$
$A_{11}B_{10}C_{01}$
$A_{11}B_{10}C_{01}$
$A_{01}B_{00}C_{01}$
$A_{10}B_{10}C_{00}$
$A_{00}B_{00}C_{00}$

Παρατήρηση

Αν παρατηρήσουμε τον αλγόριθμο θα δούμε ότι όταν φτάσει η ώρα να καλεστεί αναδρομικά η συνάρτηση FWR(A_{01},B_{00},C_{01}) αυτή θα πρέπει να περιμένει να εκτελεστεί πρώτα η FWR(A_{00},B_{00},C_{00}) που καλείται ακριβώς προηγουμένως καθώς χρειάζεται τα αποτελέσματα που θα έχουν υπολογισθεί για το block B_{00}. Έτσι λοιπόν παρατηρείται μια εξάρτηση μεταξύ αυτών των δύο αναδρομικών κλίσεων. Η λογική αυτή οδήγησε στο task-graph που είδαμε προηγουμένως.

Παρατηρώντας λοιπόν το παραπάνω task graph βλέπουμε ότι υπάρχει η δυνατότητα παραλληλοποίησης μεταξύ των εργασιών FWR(A_{01},B_{00},C_{01}), FWR(A_{10},B_{10},C_{00}) και τις δύο φορές που καλούνται η μία μετά την άλλη.

Αν θέλουμε να δούμε με περισσότερη λεπτομέρεια την σειρά εκτέλεσης και της εξαρτήσεις του προγράμματος μπορούμε να ορίσουμε ως task το εξής:

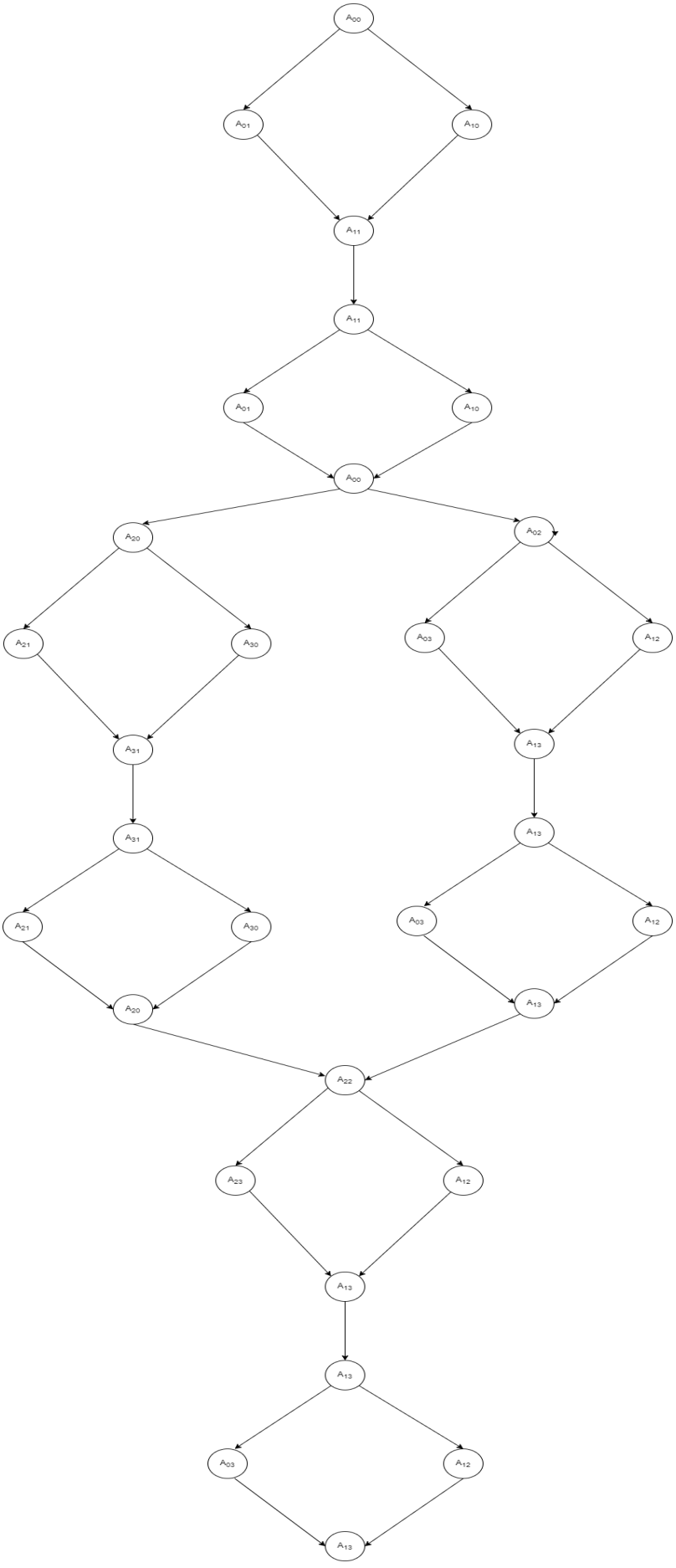
1 task = υπολογισμός της FWI

Προκειμένου να σχεδιάσουμε το task-graph θεωρούμε τον ακόλουθο πίνακα A:

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix},$$

όπου A_{ij} ένα block του πίνακα A το οποίο έχει μέγεθος BxB.

Το task graph για τον παραπάνω πίνακα είναι εύκολο να σχεδιαστεί από το task-graph που σχεδιάσαμε προηγουμένως καθώς εφόσον ο αλγόριθμος είναι αναδρομικός κάθε κόμβος του προηγούμενου task-graph μπορεί να αναλυθεί σε ένα «υπό-task graph» με την ίδια δομή όπως το αρχικό. Επομένως έχουμε το ακόλουθου task graph για αυτήν την προσέγγιση:


A00
A01
A10
A11
A11
A01
A10
A00
A20
A02
A21
A30
A03
A12
A31
A13
A31
A13
A21
A30
A03
A12
A20
A13
A22
A23
A12
A13
A13
A03
A12
A13

Παρατηρούμε ότι και σε αυτήν την περίπτωση εμφανίζονται οι δυνατότητες παραλληλίας που σχολιάσαμε και στο αρχικό task-graph. Επιπλέον το μέγιστο speedup είναι 32/18 = 1,77

Tiled Υλοποίηση

Σύμφωνα με τον κώδικα που μας δίνεται για την σειριακή tiled υλοποίηση του αλγορίθμου του FW μπορούμε να παρατηρήσουμε ότι ο αλγόριθμος λειτουργεί ως εξής:

- Χωρίζουμε τον αρχικό πίνακα γειτνίασης A μεγέθους NxN σε tiles(blocks) μεγέθους BxB. Έτσι πλέον παίρνουμε έναν νέο πίνακα A' όπου κάθε στοιχείο του θεωρούμε έναν block μεγέθους BxB.
- Σε κάθε χρονικό βήμα k εκτελούμε τα εξής:
    - Εφαρμόζουμε την standard υλοποίηση του αλγορίθμου FW στο block A'_{kk}.
    - Εφαρμόζουμε ξεχωριστά την standard υλοποίηση του αλγορίθμου FW σε όλα τα block A'_{ik} ,όπου το i διατρέχει όλες τις γραμμές του νέου πίνακα A', και σε όλα τα block A'_{kj} ,όπου το j διατρέχει όλες τις στήλες του νέου πίνακα A'. Για τον υπολογισμό των στοιχείων αυτών χρησιμοποιούμε τον πίνακα A'_{kk} που υπολογίσαμε στο προηγούμενο βήμα.
    - Εφαρμόζουμε ξεχωριστά την standard υλοποίηση του αλγορίθμου FW σε όλα τα υπόλοιπα στοιχεία A'_{ij} του πίνακα A' χρησιμοποιώντας όμως τους αντίστοιχους πίνακες A'_{ik} και A'_{kj} που υπολογίσαμε στο προηγούμενο βήμα.

<u>Κατανομή υπολογισμών</u>

Στο συγκεκριμένο αλγόριθμο θεωρούμε και πάλι μια task centric προσέγγιση όπου:

1 task = Η εφαρμογή του standard FW σε ένα tile του πίνακα A

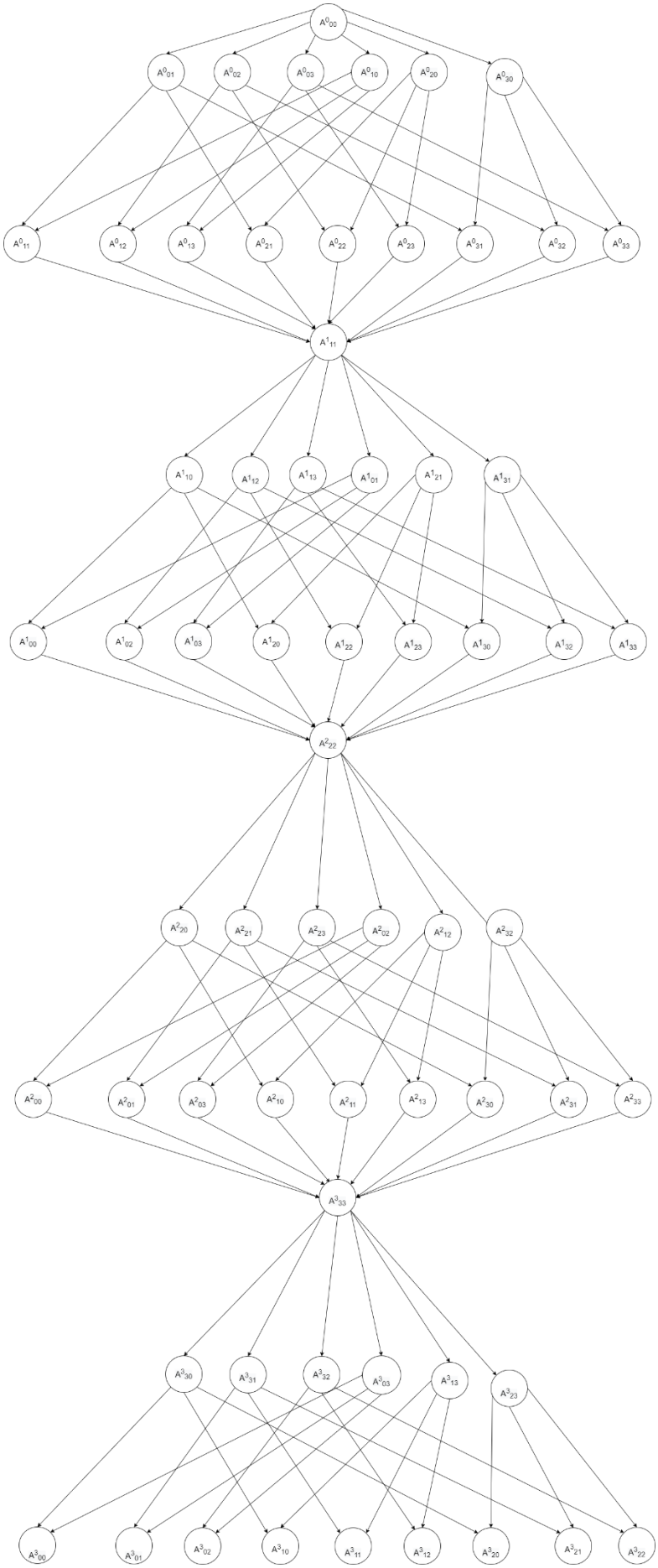
<u>Ορισμός Ορθής Σειράς Εκτέλεσης</u>

Θεωρούμε ότι ο πίνακας Α έχει την ακόλουθη μορφή:

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix},$$

όπου A_{ij} ένα συγκεκριμένο tile του πίνακα Α.

Με βάση λοιπόν τις παρατηρήσεις που έγιναν προηγουμένως και τον ορισμό του task που επιλέξαμε προκύπτει το ακόλουθο task-graph:


A⁰00
A⁰01 A⁰02 A⁰03 A⁰10 A⁰20 A⁰30
A⁰11 A⁰12 A⁰13 A⁰21 A⁰22 A⁰23 A⁰31 A⁰32 A⁰33
A¹11
A¹10 A¹12 A¹13 A¹01 A¹21 A¹31
A¹00 A¹02 A¹03 A¹20 A¹22 A¹23 A¹30 A¹32 A¹33
A²22
A²20 A²21 A²23 A²02 A²12 A²32
A²00 A²01 A²03 A²10 A²11 A²13 A²30 A²31 A²33
A³33
A³30 A³31 A³32 A³03 A³13 A³23
A³00 A³01 A³02 A³10 A³11 A³12 A³20 A³21 A³22

Παρατηρώντας και το task graph επαληθεύονται οι προηγούμενες παρατηρήσεις, ότι δηλαδή σε κάθε χρονικό βήμα k τα A_{ik} και A_{kj} υπολογίζονται ανεξάρτητα μεταξύ τους χρησιμοποιώντας όμως το A_{kk} που υπολογίζεται πρώτο. Τέλος υπολογίζονται όλα τα υπόλοιπα A_{ij} το καθένα από τα οποία εξαρτώνται από τα A_{ik} και A_{kj} που υπολογίσαμε προηγουμένως.

Παρατηρούμε λοιπόν ότι η παραλληλία εντοπίζεται στα εξής δύο σημεία:

- Στον υπολογισμό των A_{ik} και A_{kj} για ένα χρονικό βήμα k.
- Στον υπολογισμό των A_{ij} για ένα χρονικό βήμα k.

Παρατήρηση: Σύμφωνα με το Task Graph το μέγιστο speedup που μπορεί να επιτευχθεί είναι 64/12 =5,33. Έτσι η tiled υλοποίηση ως πρώτη ματιά εμφανίζει μεγαλύτερες δυνατότητες παραλληλοποίησης και κλιμάκωσης από την αντίστοιχη recursive.

Σχόλιο

Όπως εξηγήσαμε η standard υλοποίηση του FW είναι memory bound. Η tiled υλοποίηση χωρίζοντας τον πίνακα σε blocks και εφαρμόζοντας τον αλγόριθμο του FW εσωτερικά έχει ως στόχο να αυξήσει το temporal locality (locality time), δηλαδή το locality που παρατηρείται όταν κάποια δεδομένα χρησιμοποιούνται πολλές φορές. Ο τρόπος της tiled υλοποίησης να καταφέρει να εκμεταλλευτεί το temporal locality είναι να μειώσει το reuse time, δηλαδή όταν τα ίδια δεδομένα χρησιμοποιούνται πολλές φορές να χρησιμοποιούνται σε σύντομα χρονικά διαστήματα ώστε να έχουν μεγαλύτερη πιθανότητα να βρίσκονται ακόμα στην cache.

Στον αλγόριθμο του FW πιο συγκεκριμένα για ένα χρονικό βήμα k το στοιχείο A[k][j] θα χρησιμοποιηθεί από όλα τα στοιχεία που βρίσκονται στην στήλη j του πίνακα A. Εκτελώντας όμως τον standard αλγόριθμο για ολόκληρο τον πίνακα A το στοιχείο A[k][j] θα ξαναχρησιμοποηθεί όταν θα έρθει η ώρα του υπολογισμού του A[i+1][j] σε σχέση με το A[i][j] το οποίο το ξαναχρησιμοποίησε. Η tiled υλοποίηση λοιπόν χωρίζοντας τον πίνακα A σε blocks και εφαρμόζοντας τον FW εσωτερικά προσπαθεί ακριβώς να εκμεταλλευτεί αυτήν την επαναχρησιμοποίηση των δεδομένων για να μειώσει τον χρόνο εκτέλεσης του αλγορίθμου.

Βέβαια προκειμένου η tiled υλοποίηση να είναι ορθή υπάρχουν εξαρτήσεις οι οποίες πρέπει να ικανοποιούνται και οι οποίες αναλύθηκαν προηγουμένως.

Καθοριστικό ρόλο στην απόδοση της tiled υλοποίησης φαίνεται να παίζει η επιλογή της παραμέτρου B που καθορίζει το μέγεθος του tile(block). Καταλαβαίνουμε ότι χρειάζεται να επιλέξουμε σχετικά μικρό B έτσι ώστε το block μας να εκμεταλλευτούμε καλύτερα την τοπικότητα των δεδομένων αλλά και σχετικά μεγάλο ώστε να μην παρουσιαστεί overhead στους υπολογισμούς εξαιτίας του μεγάλου αριθμού των blocks.

Παρόμοιες παρατηρήσεις μπορούν να γίνουν και για την recursive υλοποίηση. Και σε αυτήν την περίπτωση προσπαθούμε να εκμεταλλευτούμε το locality των δεδομένων. Θέλουμε τιμές ορίου B οι οποίες να μην είναι σχετικά αλλά όχι και πολύ μικρές γιατί τότε θα έχουμε overhead καθώς οι αναδρομές θα είναι πολύ βαθιές.

Ερώτημα 2

Αρχικές μετρήσεις για τις τρείς σειριακές υλοποιήσεις:

Αρχικά παραθέτουμε τις μετρήσεις που έγιναν για τις τρείς σειριακές υλοποιήσεις προκειμένου να δούμε την συμπεριφορά της κάθε υλοποίησης. Έτσι έχουμε:

- *Για την Standard υλοποίηση:*

Μέγεθος πίνακα A	Χρόνος εκτέλεσης (sec)
1024x1024	1.3486
2048x2048	11.2916
4096x4096	93.7918

Παρατηρούμε ότι η αύξηση του μεγέθους του πίνακα A δημιουργεί πολύ μεγάλη αύξηση στον χρόνο εκτέλεσης του αλγορίθμου η οποία δεν είναι ανάλογη της πολυπλοκότητας του. Το γεγονός αυτό επαληθεύει την παρατήρηση μας πως η standard υλοποίηση του αλγορίθμου είναι memory bound.

- *Για την recursive υλοποίηση:*

Για πίνακα A 1024x1024:

Μέγεθος block B	Χρόνος εκτέλεσης (sec)
32x32	0.8398
64x64	0.6801
128x128	0.6847
256x256	0.7867
512x512	0.9630

Για πίνακα Α 2048x2048:

Μέγεθος block B	Χρόνος εκτέλεσης (sec)
32x32	6.5355
64x64	5.1868
128x128	4.7446
256x256	5.4706
512x512	5.9261

Για πίνακα Α 4096x4096:

Μέγεθος block B	Χρόνος εκτέλεσης (sec)
32x32	54.3782
64x64	41.2659
128x128	36.5424
256x256	43.4749
512x512	41.1402

Παρατηρήσεις:

- Παρατηρούμε ότι και για τα τρία μεγέθη του πίνακα Α η recursive υλοποίηση είναι πιο γρήγορη από την standard. Η μεγαλύτερη διαφορά στον χρόνο εκτέλεσης παρατηρείται βέβαια για μέγεθος πίνακα 4096x4096. Το γεγονός αυτό δείχνει την χρησιμότητα της recursive υλοποίησης στην προσπάθεια να καταπολεμήσει την μεγάλη αύξηση του χρόνου που παρατηρείται σε περιπτώσεις που ο πίνακας Α δεν χωράει εξ ολοκλήρου στην cache (καλύτερη εκμετάλλευσή της).

- Για μέγεθος πίνακα Α 1024x1024 παρατηρούμε ότι η καλύτερη τιμή του Β είναι 64 καθώς για μικρότερες και μεγαλύτερες τιμές ο χρόνος εκτέλεσης είναι επίσης μεγαλύτερος. Όμως για μεγέθη πίνακα Α 2048x2048 και 4096x4096 παρατηρούμε ότι η ιδανική τιμή για το Β είναι 128. Το γεγονός αυτό οφείλεται στο ότι αυξάνοντας το μέγεθος του πίνακα Α η αναδρομή γίνεται πιο βαθιά αν κρατήσουμε το Β σταθερό. Η αύξηση αυτή της αναδρομής δημιουργεί overhead στον χρόνο εκτέλεσης του αλγορίθμου.

- *Για την tiled υλοποίηση:*

Για πίνακα Α 1024x1024:

Μέγεθος block B	Χρόνος εκτέλεσης (sec)
32x32	0.8050
64x64	0.6511
128x128	0.6595
256x256	0.8039
512x512	0.9822

Για πίνακα Α 2048x2048:

Μέγεθος block B	Χρόνος εκτέλεσης (sec)
32x32	6.1631
64x64	4.9861
128x128	4.6889
256x256	5.3903
512x512	5.9349

Για πίνακα Α 4096x4096:

Μέγεθος block B	Χρόνος εκτέλεσης (sec)
32x32	50.9325
64x64	39.8292
128x128	35.5852
256x256	43.2379
512x512	41.0922

<u>Παρατηρήσεις:</u>

- Όπως ακριβώς η recursive έτσι και για την tiled υλοποίηση παρατηρούμε ότι και για τα τρία μεγέθη του πίνακα Α είναι πιο γρήγορη από την standard. Η μεγαλύτερη διαφορά στον χρόνο εκτέλεσης παρατηρείται βέβαια για μέγεθος πίνακα 4096x4096.
- Για μέγεθος πίνακα Α 1024x1024 παρατηρούμε ότι η καλύτερη τιμή του Β είναι 64 καθώς για μικρότερες και μεγαλύτερες τιμές ο χρόνος εκτέλεσης είναι επίσης μεγαλύτερος. Όμως για μεγέθη πίνακα Α 2048x2048 και 4096x4096 παρατηρούμε ότι η ιδανική τιμή για το Β είναι 128. Το γεγονός αυτό οφείλεται στο ότι αυξάνοντας το μέγεθος του πίνακα Α το φορτίο των πράξεων γίνεται όλο και μεγαλύτερο κρατώντας το Β σταθερό (στην προκειμένη περίπτωση 64). Η αύξηση αυτή της υπολογιστικής απαίτησης δημιουργεί overhead στον χρόνο εκτέλεσης αυξάνοντας τον τελικά.

- Η recursive με την tiled υλοποίηση δείχνουν να κυμαίνονται πάνω κάτω στους ίδιους χρόνους εκτέλεσης.

Υλοποίηση παράλληλων προγραμμάτων για κάθε μία από τις τρείς εκδόσεις του αλγορίθμου.

Στην συνέχεια παραθέτουμε τις αρχικές μας παράλληλες υλοποιήσεις για κάθε μία από τις τρείς εκδόσεις του αλγορίθμου FW καθώς και τους χρόνους που προέκυψαν μετά την εκτέλεση τους στο μηχάνημα sandman. Παράλληλα παραθέτουμε ορισμένα διαγράμματα αλλά και παρατηρήσεις που θα μας βοηθήσουν στην κατανόηση των παραπάνω μετρήσεων.

Standard Υλοποιήση

Όπως αναφέραμε και προηγουμένως στην standard υλοποίηση η παραλληλοποίηση παρατηρείται στον υπολογισμό των στοιχείων A[i][j] του πίνακα A για ένα χρονικό βήμα k.

Η πρώτη λοιπόν παράλληλη υλοποίηση βασίστηκε στην βιβλιοθήκη Openmp και πιο συγκεκριμένα στην δομή parallel-for η οποία χρησιμοποιείται για την παραλληλοποίηση βρόγχων. Στην συνέχεια παρατίθενται διαφορετικές παράλληλες υλοποιήσεις με χρήση του parallel for της Openmp:

Με χρήση του openmp και πιο συγκεκριμένα του parallel for

1. **parallel for schedule(static)**

Η πρώτη υλοποίηση βασίζεται στην στατική ανάθεση των υπολογισμών των γραμμών του πίνακα γειτνίασης A σε νήματα εκτέλεσης. Μάλιστα η ανάθεση αυτή γίνεται στατικά και διαδοχικά (sequential).

Σχόλιο Ο όρος sequential εξηγήθηκε στις αρχικές παρατηρήσεις που κάναμε για την standard υλοποίηση του αλγορίθμου.

Η παραπάνω υλοποίηση επιτυγχάνεται με την εντολή **#pragma omp for schedule(static)** ακριβώς πριν το loop που διατρέχει τις γραμμές του πίνακα A. Ο κώδικας που υλοποιεί την παραπάνω λογική δίνεται στην συνέχεια:

```
for(k=0;k<N;k++){
   #pragma omp parallel shared(N,A,k) private(i,j)
   {
           #pragma omp for schedule(static)
           for(i=0; i<N; i++)
                   for(j=0; j<N; j++)
                           A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
   }
}
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.1434
2 threads	1.1102
4 threads	0.6248
8 threads	0.3535
16 threads	0.3440
32 threads	0.2182
64 threads	0.2609

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	17.5175
2 threads	8.5555
4 threads	4.4134
8 threads	2.4930
16 threads	1.8630
32 threads	1.3817
64 threads	0.9845

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	143.2983
2 threads	82.5778
4 threads	37.6248
8 threads	23.4741
16 threads	23.0245
32 threads	46.1663

64 threads	17.6361

2. **parallel for schedule(static,1)**

Η δεύτερη υλοποίηση βασίζεται και πάλι στην στατική ανάθεση των υπολογισμών των γραμμών του πίνακα γειτνίασης A σε νήματα εκτέλεσης. Όμως αυτήν την φορά η ανάθεση γίνεται κυκλικά (cyclic).

Σχόλιο: Ο όρος cyclic εξηγήθηκε στις αρχικές παρατηρήσεις που κάναμε για την standard υλοποίηση του αλγορίθμου.

Η παραπάνω υλοποίηση επιτυγχάνεται με την εντολή **#pragma omp for schedule(static,1)** ακριβώς πριν το loop που διατρέχει τις γραμμές του πίνακα A. Ο κώδικας που υλοποιεί την παραπάνω λογική δίνεται στην συνέχεια:

```
for(k=0;k<N;k++){
   #pragma omp parallel shared(N,A,k) private(i,j)
   {
           #pragma omp for schedule(static,1)
           for(i=0; i<N; i++)
                   for(j=0; j<N; j++)
                           A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
   }
}
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα A 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα A 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.1427
2 threads	1.8166
4 threads	1.0497
8 threads	0.5576
16 threads	0.4039
32 threads	0.3214
64 threads	0.4081

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	18.0004
2 threads	11.3669
4 threads	5.9262
8 threads	3.0070
16 threads	2.2874
32 threads	1.6661
64 threads	1.3616

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	143.3273
2 threads	105.9412
4 threads	47.2544
8 threads	29.6275
16 threads	33.1125
32 threads	48.7490
64 threads	24.6148

3. parallel for schedule(dynamic)

Η τρίτη υλοποίηση αφορά την δυναμική ανάθεση των υπολογισμών των γραμμών του πίνακα γειτνίασης Α σε νήματα εκτέλεσης. Την δυναμική αυτή ανάθεση την αναλαμβάνει το σύστημα εκτέλεσης (run time system) της openmp. Ο κώδικας που υλοποιεί την παραπάνω υλοποίηση είναι ο ακόλουθος:

```
for(k=0;k<N;k++){
   #pragma omp parallel shared(N,A,k) private(i,j)
   {
           #pragma omp for schedule(dynamic)
           for(i=1; i<N; i++)
                   for(j=0; j<N; j++)
                           A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
   }
}
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.0500
2 threads	1.7085
4 threads	0.9839
8 threads	0.5458
16 threads	0.5076
32 threads	0.4845
64 threads	0.7277

Για πίνακα Α 2048x2048:

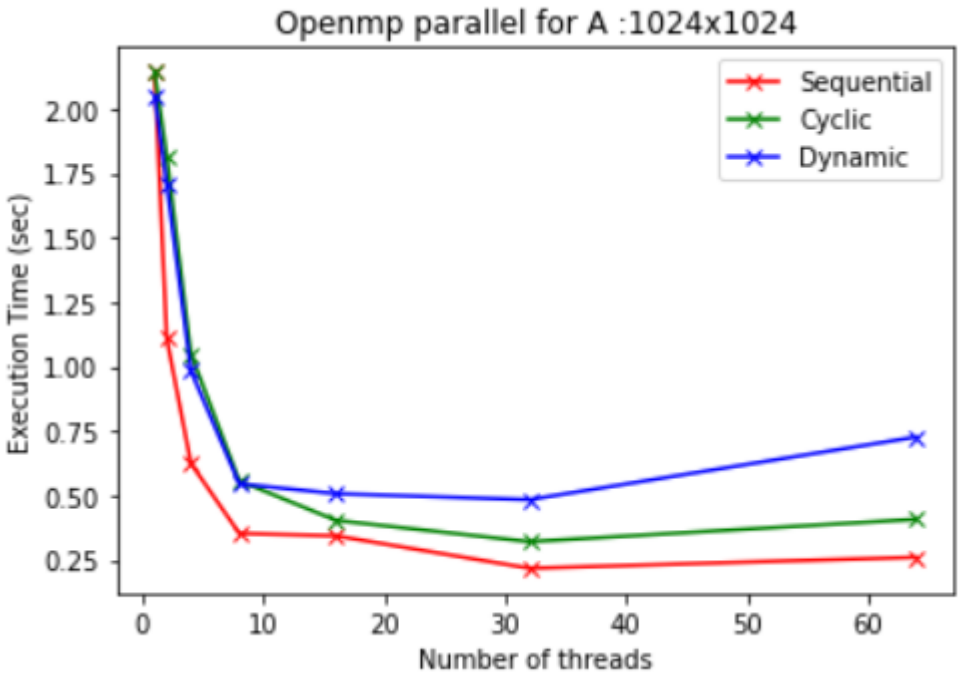
Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	16.3809
2 threads	10.9144
4 threads	6.3524
8 threads	3.7650
16 threads	3.0143
32 threads	2.5285
64 threads	3.0307

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	134.4312
2 threads	80.6257
4 threads	43.4977
8 threads	33.4386
16 threads	33.4387
32 threads	33.6602
64 threads	35.2971

Στην συνέχεια παραθέτουμε ορισμένα διαγράμματα τα οποία μας δείχνουν τον χρόνο εκτέλεσης της κάθε μίας από τις τρείς υλοποιήσεις για κάθε ένα από τα μεγέθη του πίνακα Α:

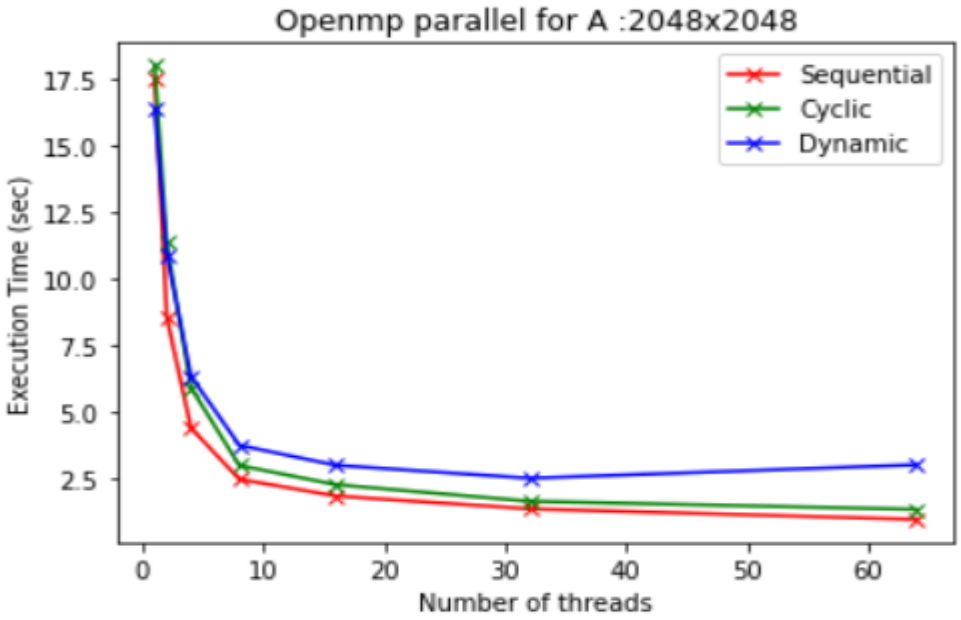
A:1024x1024



Με βάση το παραπάνω διάγραμμα κάνουμε τις ακόλουθες παρατηρήσεις:

- Καλύτερη υλοποίηση για μέγεθος πίνακα 1024x1024 είναι η static-sequential υλοποίηση. Η υλοποίηση αυτή πετυχαίνει καλύτερους χρόνους από τις άλλες δύο.
- Καμία από τις τρεις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads).

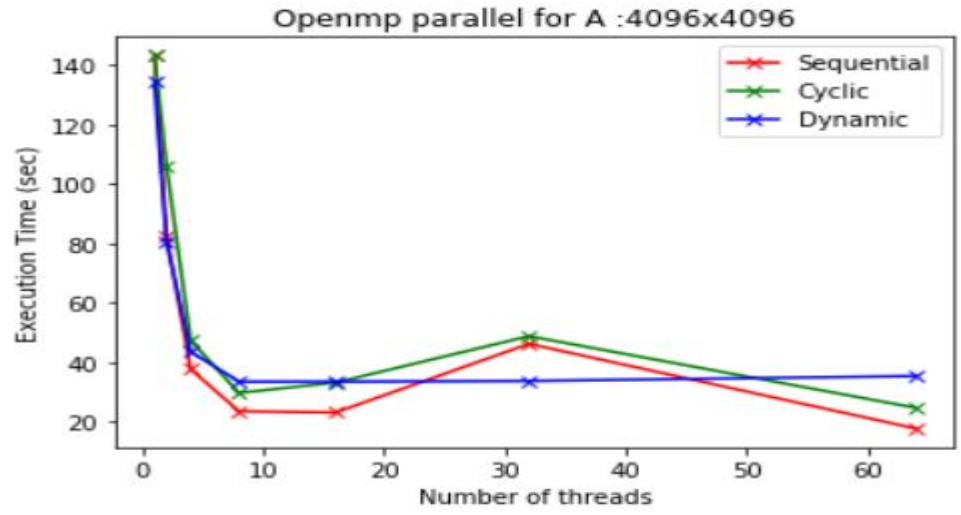
A:2048x2048



Με βάση το παραπάνω διάγραμμα κάνουμε τις ακόλουθες παρατηρήσεις:

- Καλύτερη υλοποίηση για μέγεθος πίνακα 2048x2048 εξακολουθεί να είναι η static-sequential υλοποίηση. Η υλοποίηση αυτή πετυχαίνει καλύτερους χρόνους από τις άλλες δύο.
- Καμία από τις τρεις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads).

A:4096x4096



Με βάση το παραπάνω διάγραμμα κάνουμε τις ακόλουθες παρατηρήσεις:

- Καλύτερη υλοποίηση για μέγεθος πίνακα 4096x4096 εξακολουθεί να είναι η static-sequential υλοποίηση. Στην περίπτωση όμως αυτή φαίνεται να έχουμε μία σημαντική διαφορά. Η δυναμική υλοποίηση φαίνεται να είναι καλύτερη για 32 threads.
- Καμία από τις τρεις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads).

Γενικές Παρατηρήσεις

- Η static-sequential υλοποίηση φαίνεται πως είναι καλύτερη από τις άλλες δύο και για τα τρία μεγέθη του πίνακα Α.
- Η dynamic υλοποίηση φαίνεται να είναι η χειρότερη από τις τρείς υλοποιήσεις. Κάτι τέτοιο επαληθεύει και την αρχική μας πρόβλεψη. Καθώς ο υπολογισμός των στοιχείων μιας γραμμής του πίνακα δεν παρουσιάζει ανισοκατανομή φορτίου η δυναμική υλοποίηση δεν έχει κάποια ιδιαίτερη χρησιμότητα . Αντίθετα δημιουργεί overhead στο σύστημα εκτέλεσης το οποίο πρέπει να αναλάβει την δυναμική αυτή ανάθεση.
- Η μόνη περίπτωση όπου η dynamic υλοποίηση φαίνεται να είναι καλύτερη από την static είναι για μέγεθος πίνακα A:4096x4096 και 32 threads.
- Όλες οι υλοποιήσεις έχουν πολύ κακή κλιμάκωση. Κάτι τέτοιο οφείλεται στους περιορισμούς τις standard υλοποίησης (memory bound) τους οποίους έχουμε αναλύσει και στις αρχικές μας παρατηρήσεις.

Σημείωση: Σημειώνουμε ότι χρησιμοποιήθηκε και η τεχνική #pragma omp for collapse(2) η οποία μας δίνει την δυνατότητα να παραλληλοποιήσουμε μαζί τους δύο συνεχόμενους βρόγχους (βρόγχος που διατρέχει τις γραμμές του πίνακα , βρόγχος που διατρέχει τις στήλες του πίνακα). Η τεχνική αυτή όμως δεν έδωσε κάποια καλύτερα αποτελέσματα σε σχέση με τις προηγούμενες.

Από τις τρείς λοιπόν υλοποιήσεις φαίνεται να επιλέγουμε την static – sequential ως γενικά καλύτερη.

Πέρα όμως από τη χρήση του Openmp και πιο συγκεκριμένα του parallel for η standard υλοποίηση του FW μπορεί να παραλληλοποιηθεί και με χρήση των TBBs. Τα TBBs διαθέτουν και αυτά το δικό τους tbb::parallel_for. Μέσα σε αυτό το tbb::parallel_for δημιουργείται ένα αντικείμενο το οποίο περιγράφει τον αρχικό χώρο των επαναλήψεων (tbb::blocked_range , tbb::blocked_range2d). Η ανώνυμη

συνάρτηση που ακολουθεί περιγράφει τι «δουλειά» θα γίνει σε οποιονδήποτε υποχώρο επαναλήψεων του βρόγχου.

Χρησιμοποιώντας λοιπόν την παραπάνω δομή έχουμε τις ακόλουθες υλοποιήσεις:

Με χρήση TBBs

1. Παραλληλοποίηση ανά γραμμή

Σύμφωνα με την υλοποίηση αυτή χωρίζουμε τον αρχικό χώρο των N γραμμών του πίνακα A σε chunks (κάθε chunk είναι μία γραμμή του πίνακα A) και κάθε task είναι η εκτέλεση των υπολογισμών που αφορούν το συγκεκριμένο chunk. Ο κώδικας που υλοποιεί τα παραπάνω είναι ο ακόλουθος:

```
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k++){
   tbb::parallel_for(
           tbb::blocked_range<size_t>(0,N),
           [=](const tbb::blocked_range<size_t>& r){
                   for(size_t i = r.begin(); i!=r.end(); i++){
                           for(int j=0; j<N; j++){
                                   A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
                           }
                   }
           }
           );
   }
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα A 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα A 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.2023
2 threads	1.1250
4 threads	0.7292
8 threads	0.4520
16 threads	0.3238
32 threads	0.3866
64 threads	0.4307

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	17.8267
2 threads	8.6678
4 threads	5.1061
8 threads	3.0566
16 threads	2.4811
32 threads	2.2622
64 threads	2.3493

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	144.8747
2 threads	76.9135
4 threads	40.5428
8 threads	29.6198
16 threads	29.6847
32 threads	30.0915
64 threads	32.0071

2.Παραλληλοποίηση ανά γραμμή με ορισμό κατωφλίου (grainsize)

Η υλοποίηση αυτή χωρίζει και πάλι τον αρχικό χώρο των Ν γραμμών του πίνακα σε chunks θέτοντας όμως και ένα κατώφλι που καθορίζει την δουλειά που θα αναλάβει κάθε task. Το κατώφλι αυτό ορίζεται να είναι ίσο με g = N / Number_of_threads, όπου Ν ο αριθμός των γραμμών του πίνακα Α και Number_of_theads ο αριθμός των νημάτων που θα εκτελέσουν τον συγκεκριμένο παράλληλο κώδικα. Ο κώδικας που υλοποιεί την συγκεκριμένη εκδοχή είναι ο ακόλουθος:

```
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k++){
   tbb::parallel_for(
           tbb::blocked_range<size_t>(0,N,N/nthreads),
           [=](const tbb::blocked_range<size_t>& r){
                   for(size_t i = r.begin(); i!=r.end(); i++){
                           for(int j=0; j<N; j++){
                                   A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
                           }
                   }
           }
           );
   }
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.1538
2 threads	1.1305
4 threads	0.9023
8 threads	0.3558
16 threads	0.3911
32 threads	0.4012
64 threads	0.6103

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	20.4069
2 threads	8.6243
4 threads	6.4539
8 threads	3.7947
16 threads	3.4044
32 threads	2.7111
64 threads	3.4004

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	157.3772
2 threads	82.4547
4 threads	49.4511
8 threads	56.0066
16 threads	56.3295
32 threads	48.4027
64 threads	32.8591

3. Παραλληλοποίση χρησιμοποιώντας affinity partitioner

Η υλοποίηση αυτή λειτουργεί με τον ίδιο τρόπο όπως και η πρώτη με την διαφορά της χρήσης του affinity partitioner. Το affinity partitioner χωρίζει τις γραμμές του πίνακα Α με τέτοιο τρόπο ώστε να εκμεταλλευτεί όσο καλύτερα γίνεται το cache locality. Ο κώδικας που υλοποιεί την συγκεκριμένη εκδοχή είναι ο ακόλουθος:

```
tbb::affinity_partitioner ap;
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k++){
   tbb::parallel_for(
          tbb::blocked_range<size_t>(0,N),
          [=](const tbb::blocked_range<size_t>& r){
                 for(size_t i = r.begin(); i!=r.end(); i++){
                        for(int j=0; j<N; j++){
                               A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
                        }
                 }
          },ap
          );
   }
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.1546
2 threads	1.1293
4 threads	0.5980
8 threads	0.3447
16 threads	0.3209
32 threads	0.2497
64 threads	0.3500

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	17.5263
2 threads	8.7524
4 threads	4.5107
8 threads	2.3513
16 threads	1.4133

32 threads	1.3847
64 threads	1.7032

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	142.9925
2 threads	76.8002
4 threads	37.0079
8 threads	22.3243
16 threads	19.5363
32 threads	20.2388
64 threads	23.7715

4. Παραλληλοποίηση ανά γραμμή χρησιμοποιώντας auto-partitioner

Η διαφορά της συγκεκριμένης υλοποίησης είναι η χρήση auto-partitioner το οποίο επιλέγει πιο πρέπει να είναι το σωστό grainsize για τα tasks προσπαθώντας να ελαχιστοποιήσει το range-splitting για να εξασφαλίσει καλό load-balancing. Ο κώδικας που υλοποιεί την συγκεκριμένη υλοποίηση είναι ο ακόλουθος:

```
tbb::auto_partitioner ap;
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k++){
   tbb::parallel_for(
           tbb::blocked_range<size_t>(0,N),
           [=](const tbb::blocked_range<size_t>& r){
                   for(size_t i = r.begin(); i!=r.end(); i++){
                           for(int j=0; j<N; j++){
                                   A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
                           }
                   }
           },ap
           );
   }
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.1813
2 threads	1.1078
4 threads	0.7372
8 threads	0.4444

16 threads	0.3824
32 threads	0.3892
64 threads	0.4648

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	17.5799
2 threads	8.6317
4 threads	5.0473
8 threads	2.9786
16 threads	2.3380
32 threads	2.4270
64 threads	2.4907

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	144.1395
2 threads	77.6645
4 threads	39.8316
8 threads	29.6057
16 threads	29.0330
32 threads	29.8412
64 threads	31.4291

5. Παραλληλοποίηση ανά block

Η υλοποίηση αυτή χωρίζει τον πίνακα Α μεγέθους NxN σε blocks και κάθε task αναφέρεται στον υπολογισμό των στοιχείων του συγκεκριμένου block. Ο κώδικας που υλοποιεί την παραπάνω εκδοχή είναι ο ακόλουθος:

```
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k++){
   tbb::parallel_for(
           tbb::blocked_range2d<size_t>(0,N,0,N),
           [=](const tbb::blocked_range2d<size_t>& r){
                   for(size_t i = r.rows().begin(); i!=r.rows().end(); i++){
                           for(size_t j = r.cols().begin(); j!=r.cols().end(); j++){
                                   A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
                           }
                   }
           }
           );
   }
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	1.9598
2 threads	1.1438
4 threads	1.4595
8 threads	0.7651
16 threads	0.5640
32 threads	0.7331
64 threads	0.8960

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	16.3755
2 threads	8.0173
4 threads	9.2827
8 threads	5.3714
16 threads	4.0764
32 threads	4.1823
64 threads	3.9667

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	132.5806
2 threads	81.3102
4 threads	59.4953
8 threads	40.6889
16 threads	36.9445
32 threads	37.5133
64 threads	38.2370

6. Παραλληλοποίση ανά block με ορισμό κατωφλίου (grainsize)

Η υλοποίηση αυτή χωρίζει τον πίνακα Α μεγέθους NxN σε blocks και κάθε task αναφέρεται στον υπολογισμό των στοιχείων του συγκεκριμένου block. Η διαφορά είναι πως ορίζει ως κατώφλι για το μέγεθος του block το g = N / Number_of_threads όπως και στην περίπτωση της παραλληλοποίησης ανά γραμμή με ορισμό κατωφλίου. Ο κώδικας που υλοποιεί την παραπάνω εκδοχή είναι ο ακόλουθος:

```
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k++){
   tbb::parallel_for(
           tbb::blocked_range2d<size_t>(0,N,N/nthreads,0,N,N/nthreads),
           [=](const tbb::blocked_range2d<size_t>& r){
                   for(size_t i = r.rows().begin(); i!=r.rows().end(); i++){
                           for(size_t j = r.cols().begin(); j!=r.cols().end(); j++){
                                   A[i][j]=min(A[i][j], A[i][k] + A[k][j]);
                           }
                   }
           }
           );
   }
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	1.8628
2 threads	2.0705
4 threads	1.4545
8 threads	0.7342
16 threads	0.5767
32 threads	0.7087
64 threads	0.8064

Για πίνακα Α 2048x2048:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	15.1314
2 threads	7.8577
4 threads	8.7812
8 threads	5.8888
16 threads	5.0269
32 threads	3.7534
64 threads	3.6809

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	124.1772
2 threads	83.8298
4 threads	60.7343
8 threads	51.6938
16 threads	47.6624
32 threads	41.5344
64 threads	38.8209

7. Παραλληλοποίση χρησιμοποιώντας cahce_aligned_allocator

Η συγκεκριμένη υλοποίηση λειτουργεί με τον ίδιο τρόπο όπως και η υλοποίηση 3 (χρήση affinity partioner). Η διαφορά της είναι πως αντί να δεσμεύσει μνήμη χρησιμοποιώντας την malloc χρησιμοποιεί την κλάση cache_aligned_allocator των TBBs. Το cache_aligned_allocator δεσμεύει μνήμη στα όρια των cache lines προσπαθώντας να αποφύγει το False Sharing. Το False Sharing συμβαίνει όταν λογικά διαχωρίσιμα αντικείμενα καταλαμβάνουν την ίδια Αυτό μπορεί να μειώσει την επίδοση αν πολλά threads προσπαθούν να έχουν πρόσβαση στα αντικείμενα αυτά ταυτόχρονα. Στην συνέχεια παραθέτουμε το κομμάτι της δέσμευσης μνήμης που διαφοροποιεί την συγκεκριμένη υλοποίηση από την περίπτωση 3:

```
tbb::task_scheduler_init init(nthreads);

tbb::parallel_for(
                    tbb::blocked_range<int>(0,N), [=](const tbb::blocked_range<int>& r)
                    {
                            for(int i=r.begin(); i<r.end(); i++)
                                    A[i] = tbb::cache_aligned_allocator<int>().allocate(N);
                    });
```

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads και για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για πίνακα Α 1024x1024:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	2.1652
2 threads	1.1198
4 threads	0.5922
8 threads	0.3275

16 threads	0.3128
32 threads	0.2302
64 threads	0.2851

Για πίνακα Α 2048x2048:

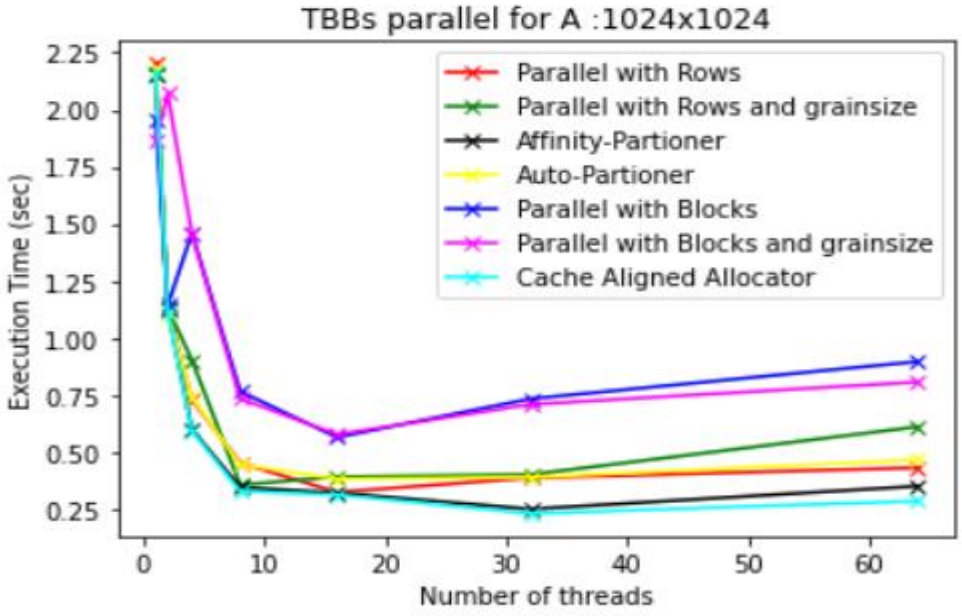
Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	18.1192
2 threads	8.6834
4 threads	4.4938
8 threads	3.0871
16 threads	2.3755
32 threads	1.5715
64 threads	1.3037

Για πίνακα Α 4096x4096:

Αριθμός νημάτων	Χρόνος εκτέλεσης
1 thread	143.8747
2 threads	76.6531
4 threads	39.6960
8 threads	25.5068
16 threads	23.4627
32 threads	21.7298
64 threads	23.6590

Στην συνέχεια παραθέτουμε ορισμένα διαγράμματα τα οποία μας δείχνουν τον χρόνο εκτέλεσης της κάθε μίας από τις 7 υλοποιήσεις για κάθε ένα από τα μεγέθη του πίνακα Α:

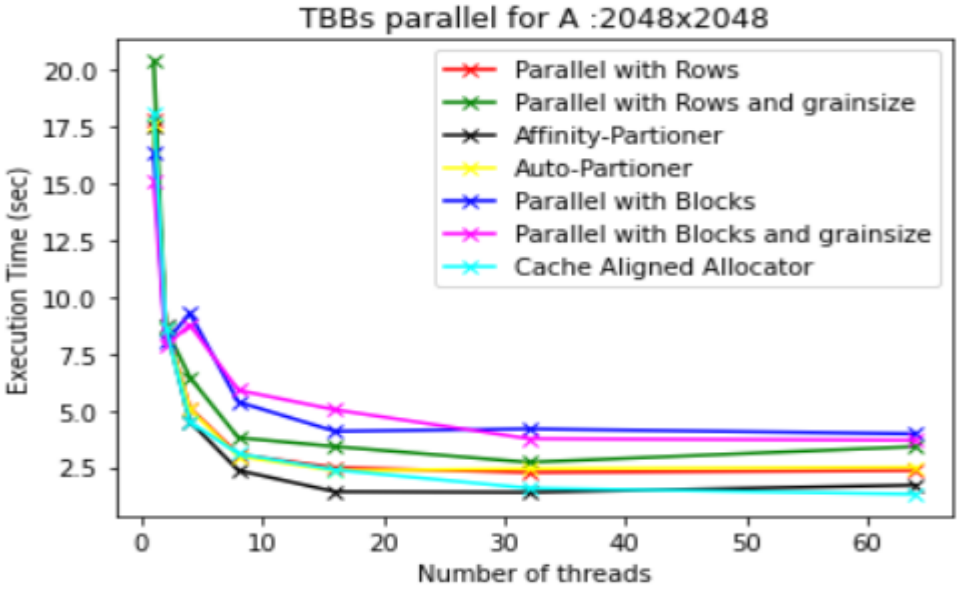
A:1024x1024



Με βάση το παραπάνω διάγραμμα κάνουμε τις ακόλουθες παρατηρήσεις:

- Καλύτερη υλοποίηση για μέγεθος πίνακα 1024x1024 είναι η Affinity-Partitioner αλλά και η Cache-Aligned-Allocator. Οι υλοποιήσεις αυτές πετυχαίνουν καλύτερους χρόνους από τι υπόλοιπες.
- Καμία από τις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads).

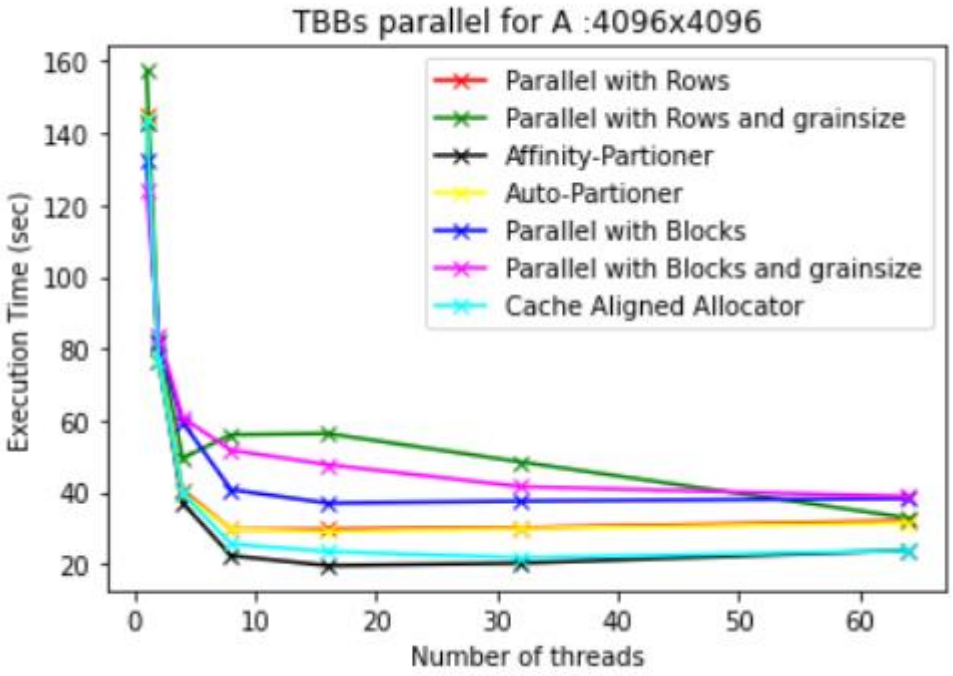
A:2048x2048



Με βάση το παραπάνω διάγραμμα κάνουμε τις ακόλουθες παρατηρήσεις:

- Καλύτερη υλοποίηση για μέγεθος πίνακα 2048x2048 είναι η Affinity-Partitioner αλλά και η Cache-Aligned-Allocator. Οι υλοποιήσεις αυτές πετυχαίνουν καλύτερους χρόνους από τι υπόλοιπες.
- Καμία από τις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads).

A:4096x4096



Με βάση το παραπάνω διάγραμμα κάνουμε τις ακόλουθες παρατηρήσεις:

- Καλύτερη υλοποίηση για μέγεθος πίνακα 2048x2048 είναι η Affinity-Partitioner αλλά και η Cache-Aligned-Allocator. Οι υλοποιήσεις αυτές πετυχαίνουν καλύτερους χρόνους από τι υπόλοιπες.
- Καμία από τις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads).

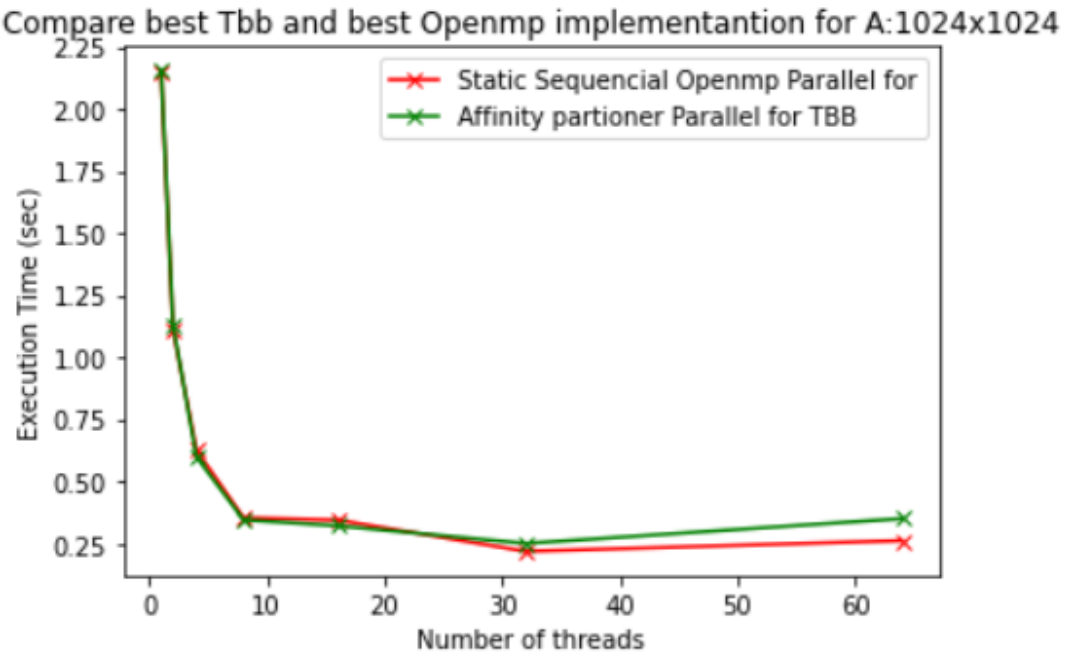
Γενικές Παρατηρήσεις

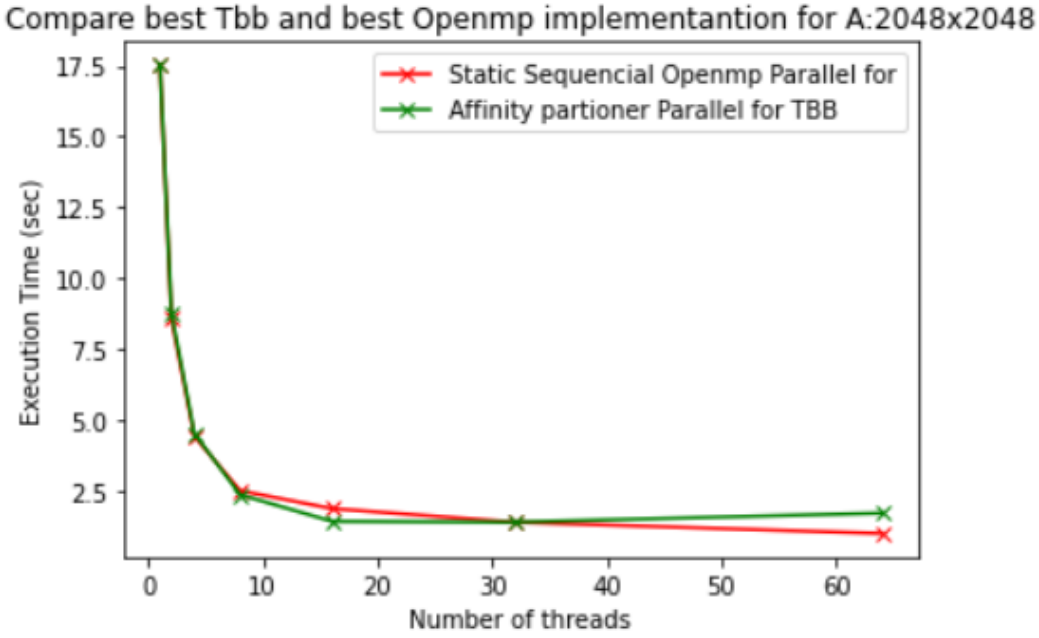
- Η υλοποίηση που χρησιμοποιεί Affinity-Partitioner φαίνεται να παρουσιάζει τα καλύτερα αποτελέσματα. Κάτι τέτοιο φαίνεται να είναι λογικό καθώς η βελτιστοποίηση του cache locality είναι ένα στοιχείο που παίζει σημαντικό ρόλο στην βελτιστοποίηση του αλγορίθμου μας όπως έχουμε εξηγήσει και στις αρχικές παρατηρήσεις.

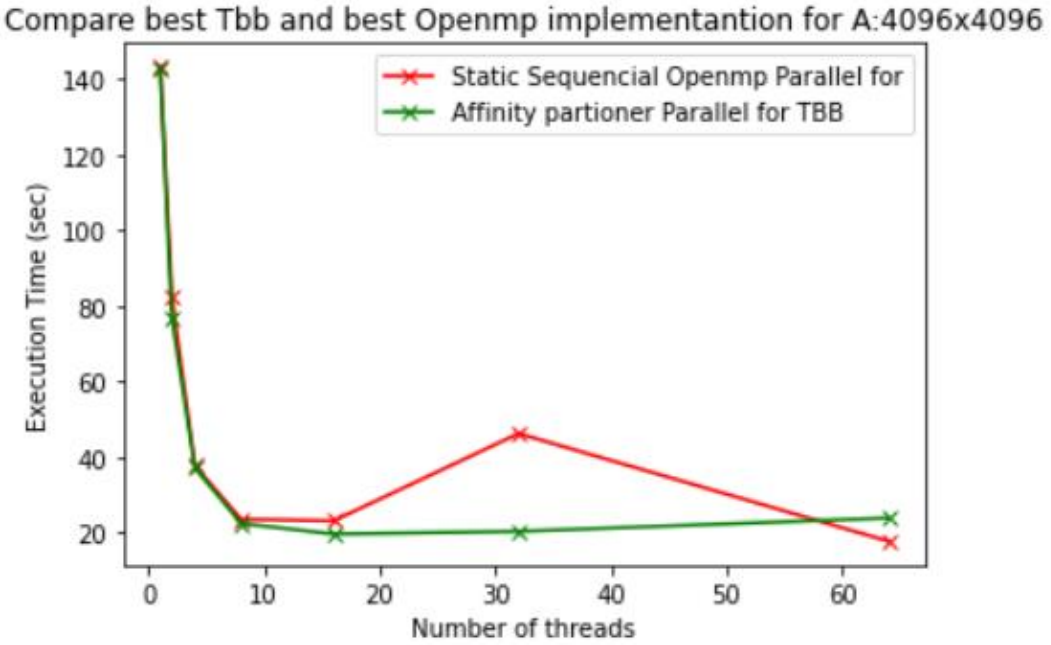
- Οι υλοποιήσεις που χωρίζουν τον πίνακα A σε blocks φαίνεται να είναι οι χειρότερες. Κάτι τέτοιο επαληθεύει και τις αρχικές μας παρατηρήσεις. Μία τέτοια υλοποίηση δημιουργεί πρόβλημα στην τοπικότητα των δεδομένων. Κάθε στοιχείο A[i][j] για να υπολογίσει την νέα του τιμή σε ένα χρονικό βήμα k χρειάζεται τις τιμές A[i][k] και A[k][j] (πέρα από την δικιά του τιμή) στο προηγούμενο χρονικό βήμα. Ο χωρισμός σε block μπορεί να οδηγήσει σε κάποιες επαναλήψεις τα νήματα να μην έχουν ούτε το A[i][k] ούτε το A[k][j] με αποτέλεσμα να αυξάνονται οι απαιτήσεις για επικοινωνία και άρα ο χρόνος εκτέλεσης.
- Η προσθήκη του cache-aligned-allocator δεν φαίνεται να επιδρά θετικά (ίσως παρατηρείται ελάχιστη μείωση σε ορισμένες περιπτώσεις) στον χρόνο εκτέλεσης.
- Όλες οι υλοποιήσεις έχουν πολύ κακή κλιμάκωση. Κάτι τέτοιο οφείλεται στους περιορισμούς τις standard υλοποίησης (memory bound) τους οποίους έχουμε αναλύσει και στις αρχικές μας παρατηρήσεις.

Η καλύτερη λοιπόν παράλληλη υλοποίηση του standard FW με χρήση TBBs φαίνεται να είναι αυτή που χρησιμοποιεί το affinity partioner.

Τέλος παρουσιάζεται ένα διάγραμμα για κάθε μέγεθος του πίνακα A που **συγκρίνει την καλύτερη υλοποίηση της Openmp (static sequential parallel for) και την καλύτερη υλοποίηση των TBBs (affinity-partioner):**







Παρατηρούμε ότι και οι δύο υλοποιήσεις παρουσιάζουν παρόμοια συμπεριφορά για όλα τα μεγέθη του πίνακα Α. Η μόνη διαφορά παρατηρείται για Α:4096x4096 και 32 threads που όπως είδαμε η openmp υλοποίηση παρουσιάζει αρκετά χειρότερη συμπεριφορά. Βέβαια η static parallel for με openmp παρουσιάζει γενικά χαμηλότερους χρόνους για 64 threads και καλύτερη κλιμάκωση. Άρα επιλέγουμε την στατική parallel for με openmp.

Recursive Υλοποίηση

Με βάση την σχεδίαση των παράλληλων προγραμμάτων που αναλύσαμε η παραλληλοποίηση στην recursive υλοποίηση παρατηρείται στον υπολογισμό των αναδρομικών κλίσεων FWR(A_{01},B_{00},C_{01}), FWR(A_{10},B_{10},C_{00}) οι οποίες καλούνται συνεχόμενα μέσα στο πρόγραμμα (μάλιστα καλούνται δύο φορές).

Μία πρώτη παράλληλη υλοποίηση έγινε με δημιουργία task dependency graph μέσω της openmp. Έτσι έχουμε την ακόλουθη υλοποίηση:

Task dependency graph με Openmp

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
void FW_SR (int **A, int arow, int acol,
            int **B, int brow, int bcol,
            int **C, int crow, int ccol,
            int myN, int bsize)
{
     int k,i,j;

     if(myN<=bsize)
        for(k=0; k<myN; k++)
           for(i=0; i<myN; i++)
              for(j=0; j<myN; j++)
                 A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
     else {
        FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
        #pragma omp parallel
        {
                #pragma omp single
                {
                        #pragma omp task shared(A,B,C)
                                FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
                        FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
                        #pragma omp taskwait
                }
        }
        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize);
        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
        #pragma omp parallel
        {
                #pragma omp single
                {
                        #pragma omp task shared(A,B,C)
                                FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
                        FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
                        #pragma omp taskwait
                }
        }
        FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
     }
}
```

Όπως παρατηρούμε η παραπάνω υλοποίηση λειτουργεί ως εξής:

- Το αρχικό νήμα εκτέλεσης εκτελεί την FWR(A_{00} , B_{10} , C_{00})
- Στην συνέχεια ,και αφού τελειώσει η προηγούμενη εκτέλεση, δημιουργείται μία παράλληλη περιοχή με την χρήση του #pragma omp parallel. Με την εντολή αυτή δημιουργείται αυτόματα ένα barrier στο τέλος της παράλληλης περιοχής που η εντολή ορίζει. Έτσι πρέπει να ολοκληρωθεί πρώτα η εκτέλεση της παράλληλης περιοχής για να συνεχίσει η εκτέλεση του κώδικα.

- Μέσα στην παράλληλη περιοχή αρχικά έχουμε την εντολή #pragma omp single. Αυτή φροντίζει μόνο ένα νήμα (το κύριο νήμα εκτέλεσης) να μπορεί να εισέλθει μέσα σε αυτήν.
- Το κύριο νήμα εισέρχεται και δημιουργεί ένα task με την εντολή #pragma omp task. Το task αφορά την εκτέλεση του FWR(A_{01} , B_{10} , C_{01}). Το task αυτό μεταφέρεται σε μία ουρά όπου θα εκτελεστεί από κάποιο νήμα.
- Παράλληλα το κύριο νήμα εκτέλεσης εκτελεί την FWR((A_{10} , B_1 , C_{00}).
- Έπειτα χρειάζεται να κάνει #pragma omp taskwait καθώς χρειάζεται να περιμένει να ολοκληρωθεί η εκτέλεση του task που δημιούργησε αμέσως πριν προκειμένου να εξέλθει από την παράλληλη περιοχή.
- Αφού ολοκληρωθεί η εκτέλεση της παράλληλης περιοχής το κύριο νήμα εκτέλεσης εκτελεί την FWR(A_{11},B_{10},C_{01}).
- Έπειτα για τις επόμενες τρεις εντολές η διαδικασία είναι ίδια με αυτήν που περιγράψαμε προηγουμένως (οι εντολές αυτές όπως ξέρουμε είναι οι ίδιες με τις προηγούμενες και εκτελούνται με την αντίστροφη σειρά (βλέπε task-graph στην σχεδίαση που προηγήθηκε)).

Όπως καταλαβαίνουμε ο παραπάνω κώδικας σέβεται τις εξαρτήσεις που παρατηρήθηκαν στην αρχική σχεδίαση της παραλληλοποίησης.

Σημείωση: Για λόγους ευκολίας παρουσιάζεται μόνο το κομμάτι του κώδικα στο οποίο υπάρχει η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads , για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096 καθώς και για μέγεθος block Β 32 , 64 , 128 , 256 , 512. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	0.8566
2	32x32	0.5014
4	32x32	0.5466
8	32x32	0.5683
16	32x32	0.6059
32	32x32	0.6667
64	32x32	0.9446
1	64x64	0.6954
2	64x64	0.4409
4	64x64	0.4766
8	64x64	0.4934
16	64x64	0.5039
32	64x64	0.5659
64	64x64	0.6543

1	128x128	0.6783
2	128x128	0.5405
4	128x128	0.5598
8	128x128	0.5763
16	128x128	0.5761
32	128x128	0.6543
64	128x128	0.7187
1	256x256	0.8087
2	256x256	0.6099
4	256x256	0.5847
8	256x256	0.6784
16	256x256	0.6785
32	256x256	0.7419
64	256x256	0.8540
1	512x512	0.9677
2	512x512	0.8088
4	512x512	0.8342
8	512x512	0.8809
16	512x512	0.8493
32	512x512	0.9910
64	512x512	1.0116

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	6.8650
2	32x32	3.5977
4	32x32	3.7900
8	32x32	3.9359
16	32x32	4.0795
32	32x32	4.3123
64	32x32	5.3447
1	64x64	5.3376
2	64x64	2.9300
4	64x64	3.0740
8	64x64	3.2028
16	64x64	3.3431
32	64x64	3.7543
64	64x64	4.0833

1	128x128	4.8407
2	128x128	3.0874
4	128x128	3.3084
8	128x128	3.3172
16	128x128	3.4932
32	128x128	3.7868
64	128x128	4.1087
1	256x256	5.5881
2	256x256	3.6092
4	256x256	3.7013
8	256x256	3.8353
16	256x256	3.7489
32	256x256	4.0764
64	256x256	4.5881
1	512x512	6.0310
2	512x512	4.3868
4	512x512	4.4271
8	512x512	4.4699
16	512x512	4.5148
32	512x512	4.6181
64	512x512	4.9523

Για Α 4096 x 4096

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	55.6393
2	32x32	28.7684
4	32x32	29.5881
8	32x32	30.2662
16	32x32	30.2761
32	32x32	31.3311
64	32x32	35.0064
1	64x64	41.6049
2	64x64	22.2180
4	64x64	22.4831
8	64x64	22.5182
16	64x64	23.9750
32	64x64	25.8715
64	64x64	26.6888
1	128x128	36.0514

2	128x128	21.1406
4	128x128	22.1722
8	128x128	22.4019
16	128x128	23.2522
32	128x128	23.8560
64	128x128	25.2030
1	256x256	43.8691
2	256x256	25.5690
4	256x256	25.9677
8	256x256	26.2579
16	256x256	26.6738
32	256x256	27.6759
64	256x256	29.3488
1	512x512	41.2800
2	512x512	26.2359
4	512x512	26.7033
8	512x512	26.7577
16	512x512	26.5646
32	512x512	27.7943
64	512x512	27.4183

Στην συνέχεια παραθέτουμε δύο υλοποιήσεις οι οποίες κάνουν χρήση των TBBs

1. Task-groups

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

tbb::task_group g;

g.run([arow,acol,brow,bcol,crow,ccol,myN,bsize,&A,&B,&C] {
        FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
});

g.run([arow,acol,brow,bcol,crow,ccol,myN,bsize,&A,&B,&C] {
        FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
} );
g.wait();

FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize);
FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);

g.run([arow,acol,brow,bcol,crow,ccol,myN,bsize,&A,&B,&C] {
        FW_SR(A,arow+myN/2, acol,B,brow+myN/2, ccol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
} );
g.run([arow,acol,brow,bcol,crow,ccol,myN,bsize,&A,&B,&C] {
        FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
} );
g.wait();

FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
```

Ο συγκεκριμένος κώδικας όπως και ο κώδικας που προηγήθηκε έχουν παρόμοια λογική. Στο συγκεκριμένο κώδικα δημιουργούμε ομάδες από tasks (task-groups) στα οποία έχουμε δύο βασικές λειτουργίες. Με την συνάρτηση run τοποθετούμε ένα task στην ουρά προς εκτέλεση από κάποιο thread και με την συνάρτηση wait περιμένουμε την ολοκλήρωση όλων των προηγούμενων tasks του συγκεκριμένου task group. Η λογική με την οποία δημιουργήθηκε το task-group του συγκεκριμένου κώδικα αλλά και η λογική με την οποία επιτευχθεί ο συγχρονισμός των tasks είναι ίδια με την πρώτη υλοποίηση και δεν περιγράφεται αναλυτικά. Επισημαίνουμε ότι οι πίνακες περνούν ως ορίσματα by reference για να έχουμε ελάχιστη καθυστέρηση.

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads , για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096 καθώς και για μέγεθος block Β 32 , 64 , 128 , 256 , 512. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	0.8404
2	32x32	0.4859
4	32x32	0.2946
8	32x32	0.3392
16	32x32	0.4139
32	32x32	0.5141
64	32x32	0.6123
1	64x64	0.6733
2	64x64	0.4232
4	64x64	0.3286
8	64x64	0.3655
16	64x64	0.4135
32	64x64	0.4329
64	64x64	0.4923
1	128x128	0.7311
2	128x128	0.4706
4	128x128	0.4089
8	128x128	0.4069
16	128x128	0.4609
32	128x128	0.4444
64	128x128	0.4674

1	256x256	0.8133
2	256x256	0.5967
4	256x256	0.5861
8	256x256	0.5866
16	256x256	0.6499
32	256x256	0.5572
64	256x256	0.6267
1	512x512	0.9642
2	512x512	0.8316
4	512x512	0.8985
8	512x512	0.9013
16	512x512	0.8213
32	512x512	0.8371
64	512x512	0.8128

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	7.0022
2	32x32	3.4509
4	32x32	2.6067
8	32x32	2.1086
16	32x32	2.4550
32	32x32	2.7367
64	32x32	3.5525
1	64x64	6.0402
2	64x64	3.0278
4	64x64	2.3835
8	64x64	2.1021
16	64x64	2.4733
32	64x64	2.5212
64	64x64	2.7506
1	128x128	4.7043
2	128x128	2.9642
4	128x128	2.4071
8	128x128	2.3114
16	128x128	2.3826
32	128x128	2.6981
64	128x128	2.7267

1	256x256	5.4105
2	256x256	3.7569
4	256x256	3.1333
8	256x256	3.0655
16	256x256	3.0689
32	256x256	3.0921
64	256x256	3.0723
1	512x512	5.9361
2	512x512	4.2290
4	512x512	4.1830
8	512x512	4.1315
16	512x512	4.1086
32	512x512	4.4124
64	512x512	4.1766

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	51.4411
2	32x32	26.4184
4	32x32	19.1573
8	32x32	15.2454
16	32x32	15.9476
32	32x32	17.2369
64	32x32	23.0422
1	64x64	40.0472
2	64x64	21.6410
4	64x64	15.5934
8	64x64	13.0899
16	64x64	14.3052
32	64x64	15.4328
64	64x64	16.4846
1	128x128	35.2175
2	128x128	20.6297
4	128x128	15.5485
8	128x128	13.3635
16	128x128	14.1422
32	128x128	14.8700

64	128x128	16.0368
1	256x256	42.8374
2	256x256	24.1891
4	256x256	19.1055
8	256x256	18.2041
16	256x256	18.2389
32	256x256	19.4284
64	256x256	18.6591
1	512x512	40.9874
2	512x512	24.5595
4	512x512	21.7695
8	512x512	21.6280
16	512x512	21.7499
32	512x512	22.0966
64	512x512	21.7804

2.Flow graph

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
else {
   tbb::flow::graph g;

   tbb::flow::continue_node <tbb::flow::continue_msg> A11(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize); }
    );
   tbb::flow::continue_node <tbb::flow::continue_msg> A12(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
   } );
   tbb::flow::continue_node <tbb::flow::continue_msg> A21(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
   } );
   tbb::flow::continue_node <tbb::flow::continue_msg> A22(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize);
   });
   tbb::flow::continue_node <tbb::flow::continue_msg> A22_(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
   });
   tbb::flow::continue_node <tbb::flow::continue_msg> A21_(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow+myN/2, acol,B,brow+myN/2, ccol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
   } );
   tbb::flow::continue_node <tbb::flow::continue_msg> A12_(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
   } );
   tbb::flow::continue_node <tbb::flow::continue_msg> A11_(g,[=] (const tbb::flow::continue_msg &)
   {
           FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
   });
```

```
make_edge(A11,A12);
make_edge(A11,A21);
make_edge(A12,A22);
make_edge(A21,A22);
make_edge(A22,A22_);
make_edge(A22_,A12_);
make_edge(A22_,A21_);
make_edge(A12_,A11_);
make_edge(A21_,A11_);

A11.try_put(tbb::flow::continue_msg());
g.wait_for_all();
```

Ο συγκεκριμένος κώδικας δημιουργεί απευθείας τους κόμβους, δηλαδή τα tasks, και τις ακμές, δηλαδή τις εξαρτήσεις μεταξύ των tasks, υλοποιώντας έτσι ένα task graph. δίνοντας στο σύστημα εκτέλεσης την ευθύνη να τους διαμοιράσει σε threads. Οι ακμές δημιουργούνται με την εντολή make_edge η οποία δέχεται ως ορίσματα τους δύο κόμβους – tasks που έχουν εξάρτηση και πρέπει να «ενωθούν» με μία ακμή. Το Flow-Graph δίνει μεγαλύτερη ευελιξία σε σχέση με τα task-groups αφού μπορεί να περιγράψει οποιονδήποτε γράφο εξαρτήσεων όσο περίπλοκος και αν είναι καθώς απλά δημιουργεί κόμβους και ακμές.

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads , για μέγεθος πίνακα A 1024x1024, 2048x2048, 4096x4096 καθώς και για μέγεθος block B 32 , 64 , 128 , 256 , 512. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	0.7927
2	32x32	0.4603
4	32x32	0.4409
8	32x32	0.4128
16	32x32	0.5385
32	32x32	0.6206
64	32x32	0.7269
1	64x64	0.7426
2	64x64	0.4418
4	64x64	0.3635
8	64x64	0.4065
16	64x64	0.4080
32	64x64	0.4918
64	64x64	0.5716
1	128x128	0.7420
2	128x128	0.5007
4	128x128	0.4513
8	128x128	0.4641
16	128x128	0.4345
32	128x128	0.5083
64	128x128	0.5000
1	256x256	0.8409
2	256x256	0.5876
4	256x256	0.5664
8	256x256	0.6150
16	256x256	0.6110
32	256x256	0.6602
64	256x256	0.6462
1	512x512	0.9634
2	512x512	0.8158
4	512x512	0.8345
8	512x512	0.8498
16	512x512	0.8181
32	512x512	0.8122
64	512x512	0.8809

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	6.1457
2	32x32	3.3884
4	32x32	3.8508
8	32x32	3.4101
16	32x32	3.6368
32	32x32	3.7961
64	32x32	4.8725
1	64x64	5.0816
2	64x64	2.6962
4	64x64	2.7462
8	64x64	2.8693
16	64x64	3.3290
32	64x64	3.3070
64	64x64	3.4376
1	128x128	4.7094
2	128x128	2.7034
4	128x128	2.7191
8	128x128	2.8404
16	128x128	2.9632
32	128x128	3.1185
64	128x128	2.8232
1	256x256	5.5100
2	256x256	3.3800
4	256x256	3.1213
8	256x256	3.2828
16	256x256	3.0189
32	256x256	3.1970
64	256x256	3.3254
1	512x512	5.9138
2	512x512	4.2531
4	512x512	4.1341
8	512x512	4.1463
16	512x512	4.1765
32	512x512	4.1671
64	512x512	4.2239

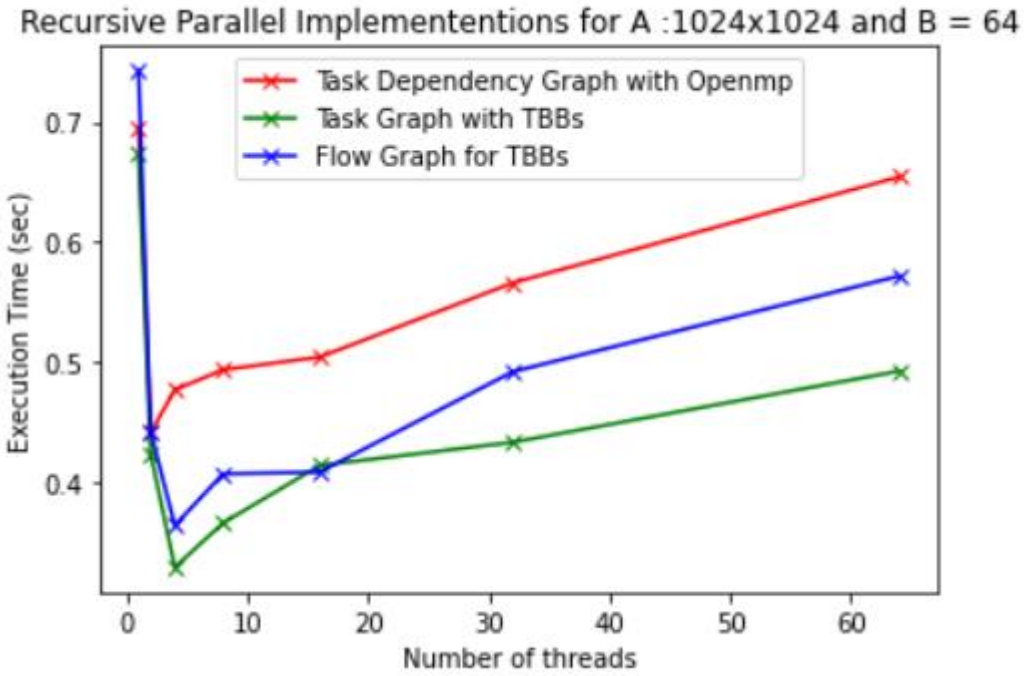
Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	49.7788
2	32x32	25.8892
4	32x32	27.2942
8	32x32	25.2554
16	32x32	27.8798
32	32x32	26.1864
64	32x32	37.2939
1	64x64	38.9658
2	64x64	20.4024
4	64x64	19.1677
8	64x64	20.1898
16	64x64	24.6152
32	64x64	24.5331
64	64x64	24.8533
1	128x128	35.3638
2	128x128	18.9084
4	128x128	18.4570
8	128x128	19.6258
16	128x128	21.7358
32	128x128	20.0909
64	128x128	22.4573
1	256x256	43.3490
2	256x256	24.3988
4	256x256	19.8441
8	256x256	20.8541
16	256x256	21.8641
32	256x256	20.6811
64	256x256	20.8531
1	512x512	40.8977
2	512x512	24.9819
4	512x512	22.0859
8	512x512	21.9837
16	512x512	21.9303
32	512x512	22.0246
64	512x512	21.9751

Στην συνέχεια παραθέτουμε ορισμένα διαγράμματα τα οποία μας δείχνουν τον χρόνο εκτέλεσης της κάθε μίας από τις 3 προηγούμενες υλοποιήσεις για κάθε ένα από τα μεγέθη του πίνακα Α. Το μέγεθος του Β επιλέχθηκε να είναι 64 προκειμένου να γίνουν οι συγκρίσεις. Όπως παρατηρήσαμε από τις μετρήσεις το 64 είναι το νούμερο όπου γενικά παρατηρούνται οι καλύτερες μετρήσεις του Β. Σε κάποιες περιπτώσεις βέβαια για μέγεθος πίνακα Α:2048x2048 και Α:4096x4096 παρατηρήθηκε ότι η καλύτερη τιμή του Β είναι 128. Κάτι τέτοιο έρχεται σε συμφωνία με την αρχική μας παρατήρηση για την τιμή του Β που κάναμε στην σειριακή recursive υλοποίηση.

Τα διαγράμματα είναι τα ακόλουθα:

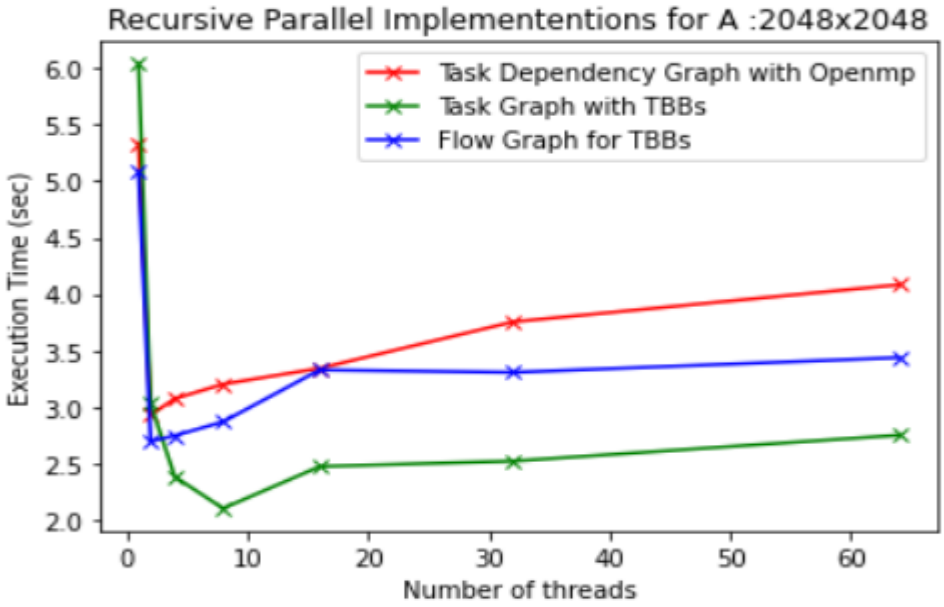
A:1024x1024 – B:64x64



<u>Παρατηρήσεις</u>

- Καλύτερη υλοποίηση για μέγεθος πίνακα 1024x1024 είναι η Task-Graph with TBBs.
- Καμία από τις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads). Μάλιστα όπως παρατηρούμε αυξάνοντας τον αριθμό των νημάτων σε πάνω 4 υπάρχει και αύξηση του χρόνου εκτέλεσης. Βλέπουμε λοιπόν ότι οι recursive παράλληλες υλοποιήσεις παρουσιάζουν πολύ κακή κλιμάκωση.

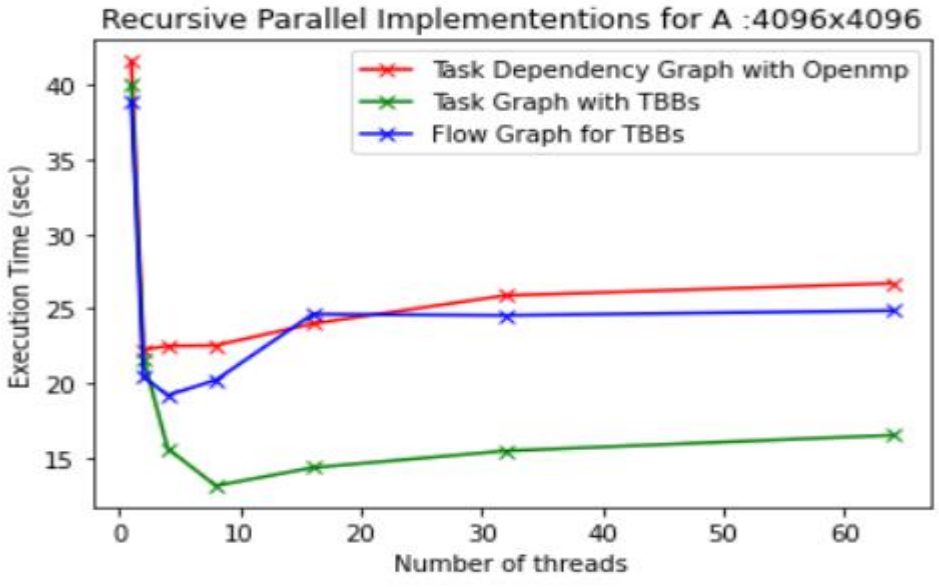
A:2048x4048 – B:64x64



Παρατηρήσεις

- Καλύτερη υλοποίηση για μέγεθος πίνακα 2048x4048 συνεχίζει να είναι η Task-Graph with TBBs.
- Καμία από τις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads). Μάλιστα όπως παρατηρούμε αυξάνοντας τον αριθμό των νημάτων σε πάνω 4 υπάρχει και αύξηση του χρόνου εκτέλεσης. Βλέπουμε λοιπόν ότι οι recursive παράλληλες υλοποιήσεις παρουσιάζουν πολύ κακή κλιμάκωση.

A:4096x4096 – B:64x64



Παρατηρήσεις

- Καλύτερη υλοποίηση για μέγεθος πίνακα 4096x4096 συνεχίζει να είναι η Task-Graph with TBBs.
- Καμία από τις υλοποιήσεις δεν έχει καλό speedup (ανάλογη μείωση του χρόνου εκτέλεσης με την αύξηση των threads). Μάλιστα όπως παρατηρούμε αυξάνοντας τον αριθμό των νημάτων σε πάνω 4 ή 8 υπάρχει και αύξηση του χρόνου εκτέλεσης. Βλέπουμε λοιπόν ότι οι recursive παράλληλες υλοποιήσεις παρουσιάζουν πολύ κακή κλιμάκωση.

Γενικές Παρατηρήσεις

- Η Task Groups with TBBs υλοποίηση είναι η καλύτερη και για τα τρία μεγέθη του πίνακα Α.
- Η Flow graph with TBBs υλοποίηση παρουσιάζει χειρότερη συμπεριφορά από την Task-Graph with TBBs. Αυτό ίσως οφείλεται στο μεγαλύτερο overhead που προκαλεί η δημιουργία του Flow-graph. Ακριβώς εξαιτίας της ευελιξίας του και τις απλότητας υλοποίησης του το flow-graph δημιουργείται στατικά και έπειτα ξεκινάει η εκτέλεση του. Στην συγκεκριμένη περίπτωση που ο γράφος των εξαρτήσεων είναι σχετικά απλός και μπορεί να υλοποιηθεί από τα task-groups η χρήση του flow-graph φαίνεται να επιβαρύνει την εκτέλεση του προγράμματος.
- Η Task Dependency Graph με Openmp φαίνεται να είναι η χειρότερη από τις τρείς υλοποιήσεις. Έτσι φαίνεται πως η δημιουργία του γράφου εξαρτήσεων με χρήση των TBBs δίνει καλύτερα αποτελέσματα από ότι με χρήση της Openmp.
- Όλες οι υλοποιήσεις έχουν πολύ κακή κλιμάκωση. Μάλιστα σε ορισμένες περιπτώσεις έχουμε και αύξηση του χρόνου εκτέλεσης με την αύξηση των πυρήνων. Κάτι τέτοιο έρχεται σε συμφωνία με τις αρχικές μας παρατηρήσεις όπου παρατηρήσαμε την όχι και τόσο μεγάλη δυνατότητα παραλληλίας που παρέχει η recursive υλοποίηση (δες μέγιστο speedup που υπολογίσθηκε από τον γράφο εξαρτήσεων κατά την διάρκεια της σχεδίασης της παραλληλίας.)

Tiled Υλοποίηση

Με βάση την σχεδίαση που κάναμε η παραλληλοποίηση στην tiled υλοποίηση παρατηρείται στα εξής:

- Στον υπολογισμό των A_{ik} και A_{kj} για ένα χρονικό βήμα k.
- Στον υπολογισμό των υπόλοιπων A_{ij} (πέρα από τα block του σταυρού) για ένα χρονικό βήμα k.

Με βάση την παραλληλία αυτή δημιουργήθηκαν οι παρακάτω παράλληλες υλοποιήσεις οι οποίες αναλύονται στην συνέχεια:

Με χρήση των TBBs

1.Task-group

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k+=B){
   FW(A,k,k,k,B);

   tbb::task_group g;

   for(i=0; i<k; i+=B)
      g.run( [k,i,B,&A] { FW(A,k,i,k,B); } );

   for(i=k+B; i<N; i+=B)
      g.run( [k,i,B,&A] { FW(A,k,i,k,B); } );

   for(j=0; j<k; j+=B)
      g.run( [k,j,B,&A] { FW(A,k,k,j,B); } );

   for(j=k+B; j<N; j+=B)
      g.run( [k,j,B,&A] { FW(A,k,k,j,B); } );

   g.wait();

   for(i=0; i<k; i+=B)
      for(j=0; j<k; j+=B)
         g.run( [k,i,j,B,&A] { FW(A,k,i,j,B); } );

   for(i=0; i<k; i+=B)
      for(j=k+B; j<N; j+=B)
         g.run( [k,i,j,B,&A] { FW(A,k,i,j,B); } );

   for(i=k+B; i<N; i+=B)
      for(j=0; j<k; j+=B)
         g.run( [k,i,j,B,&A] { FW(A,k,i,j,B); } );

   for(i=k+B; i<N; i+=B)
      for(j=k+B; j<N; j+=B)
         g.run( [k,i,j,B,&A] { FW(A,k,i,j,B); } );
   g.wait();
}
```

Όπως παρατηρούμε ο παραπάνω κώδικας λειτουργεί ως εξής:

- Αρχικά υπολογίζεται από το κύριο νήμα εκτέλεσης το διαγώνιο block A_{kk}.
- Έπειτα για κάθε block του σταυρού (A_{ik} και A_{kj}) δημιουργείται ένα task το οποίο μεταφέρεται στην ουρά εκτέλεσης για να εκτελεστεί από τα threads.
- Έπειτα το κύριο νήμα εκτέλεσης περιμένει των υπολογισμό όλων των block του σταυρού προκειμένου να συνεχίσει την εκτέλεση του (εντολή g.wait()).

- Αφού ολοκληρωθεί ο υπολογισμός για όλα τα block του σταυρού δημιουργείται ένα task για κάθε ένα από τα υπόλοιπα block και τοποθετείται στην ουρά εκτέλεσης για να εκτελεσθεί από τα threads.
- Το κύριο νήμα εκτέλεσης περιμένει να ολοκληρωθεί ο υπολογισμός για όλα τα υπόλοιπα block ώστε να ξεκινήσει από την αρχή την ίδια διαδικασία για το επόμενο χρονικό βήμα k.

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads, για μέγεθος πίνακα A 1024x1024, 2048x2048, 4096x4096 καθώς και για μέγεθος block B 32 , 64 , 128. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για A 1024 x 1024:

Αριθμός νημάτων	Μέγεθος B	Χρόνος Εκτέλεσης (sec)
1	32x32	0.7703
2	32x32	0.4282
4	32x32	0.5661
8	32x32	0.4554
16	32x32	0.3173
32	32x32	0.2411
64	32x32	0.3940
1	64x64	0.7306
2	64x64	0.3977
4	64x64	0.3945
8	64x64	0.1717
16	64x64	0.1412
32	64x64	0.2203
64	64x64	0.2123
1	128x128	0.7593
2	128x128	0.4231
4	128x128	0.3471
8	128x128	0.2118
16	128x128	0.2530
32	128x128	0.2213
64	128x128	0.2687
1	256x256	0.8704
2	256x256	0.5081
4	256x256	0.4092
8	256x256	0.3286
16	256x256	0.4238
32	256x256	0.3737

64	256x256	0.3615
1	512x512	0.9399
2	512x512	0.8383
4	512x512	0.8288
8	512x512	0.8951
16	512x512	0.8827
32	512x512	0.8755
64	512x512	0.8515

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	6.1109
2	32x32	3.1291
4	32x32	4.5285
8	32x32	3.8971
16	32x32	3.2599
32	32x32	3.1388
64	32x32	2.4705
1	64x64	4.9605
2	64x64	2.5403
4	64x64	2.4856
8	64x64	1,8444
16	64x64	1.2490
32	64x64	1.1596
64	64x64	1.1165
1	128x128	4.7870
2	128x128	2.4853
4	128x128	2.4605
8	128x128	1.8020
16	128x128	1.1349
32	128x128	1.0107
64	128x128	0.9859
1	256x256	5.4899
2	256x256	2.9417
4	256x256	2.3002
8	256x256	1.6369
16	256x256	1.3551
32	256x256	1.1786
64	256x256	1.2663

1	512x512	5.9351
2	512x512	3.6408
4	512x512	2.5330
8	512x512	2.3417
16	512x512	2.2518
32	512x512	2.5887
64	512x512	2.4940

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	50.5895
2	32x32	25.595
4	32x32	38.4020
8	32x32	32.9890
16	32x32	28.6368
32	32x32	18.9066
64	32x32	12.5939
1	64x64	39.2869
2	64x64	20.1086
4	64x64	13.3264
8	64x64	18.3174
16	64x64	13.3626
32	64x64	9.6986
64	64x64	5.8841
1	128x128	35.7565
2	128x128	18.1348
4	128x128	19.4232
8	128x128	14.5768
16	128x128	10.2503
32	128x128	7.1732
64	128x128	4.8367
1	256x256	43.4059
2	256x256	22.3831
4	256x256	18.1368
8	256x256	13.6973
16	256x256	9.4827
32	256x256	6.8570
64	256x256	5.5312

1	512x512	41.8591
2	512x512	22.0091
4	512x512	16.7767
8	512x512	11.5339
16	512x512	8.8518
32	512x512	9.9555
64	512x512	11.4955

2.Flow-graph

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
tbb::task_scheduler_init init(nthreads);
for(k=0;k<N;k+=B){
   tbb::flow::graph g;
   tbb::flow::continue_node <tbb::flow::continue_msg> node1(g,[=] (const tbb::flow::continue_msg &)
           {
                   FW(A,k,k,k,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node2(g,[=] (const tbb::flow::continue_msg &)
           {
                   for(int i=0; i<k; i+=B) FW(A,k,i,k,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node3(g,[=] (const tbb::flow::continue_msg &)
           {
                    for(int i=k+B; i<N; i+=B) FW(A,k,i,k,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node4(g,[=] (const tbb::flow::continue_msg &)
           {
                   for(int j=0; j<k; j+=B) FW(A,k,k,j,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node5(g,[=] (const tbb::flow::continue_msg &)
           {
                   for(int j=k+B; j<N; j+=B) FW(A,k,k,j,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node6(g,[=] (const tbb::flow::continue_msg &)
           {
                   for(int i=0; i<k; i+=B) for(int j=0; j<k; j+=B) FW(A,k,i,j,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node7(g,[=] (const tbb::flow::continue_msg &)
           {
                   for(int i=0; i<k; i+=B) for(int j=k+B; j<N; j+=B) FW(A,k,i,j,B);
           } );

   tbb::flow::continue_node <tbb::flow::continue_msg> node8(g,[=] (const tbb::flow::continue_msg &)
           {
                    for(int i=k+B; i<N; i+=B) for(int j=0; j<k; j+=B) FW(A,k,i,j,B);
           } );
```

```
tbb::flow::continue_node <tbb::flow::continue_msg> node9(g,[=] (const tbb::flow::continue_msg &)
        {
                for(int i=k+B; i<N; i+=B) for(int j=k+B; j<N; j+=B) FW(A,k,i,j,B);
        } );

make_edge(node1,node2);
make_edge(node1,node3);
make_edge(node1,node4);
make_edge(node1,node5);
make_edge(node2,node6);
make_edge(node4,node6);
make_edge(node2,node7);
make_edge(node5,node7);
make_edge(node3,node8);
make_edge(node4,node8);
make_edge(node3,node8);
make_edge(node5,node8);

node1.try_put(tbb::flow::continue_msg());
g.wait_for_all();
```

Ο κώδικας αυτός δημιουργεί απευθείας τον γράφο των εξαρτήσεων που δείξαμε στην σχεδίαση.

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads , για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096 καθώς και για μέγεθος block Β 32 , 64 , 128 , 256 , 512. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	0.5470
2	32x32	0.4093
4	32x32	0.3667
8	32x32	0.3687
16	32x32	03941
32	32x32	0.3859
64	32x32	0.3874
1	64x64	0.5137
2	64x64	0.3747
4	64x64	0.3388
8	64x64	0.3578
16	64x64	0.3721
32	64x64	0.3502
64	64x64	0.3991
1	128x128	0.5778
2	128x128	0.4223
4	128x128	0.3757
8	128x128	0.3923
16	128x128	0.3933
32	128x128	0.4166
64	128x128	0.3923
1	256x256	0.6744
2	256x256	0.5442
4	256x256	0.5157
8	256x256	0.5524
16	256x256	0.5716
32	256x256	0.5123
64	256x256	0.5134
1	512x512	0.9034
2	512x512	0.7727
4	512x512	0.7774

8	512x512	0.7804
16	512x512	0.8019
32	512x512	0.8131
64	512x512	0.7972

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	4.0048
2	32x32	2.8986
4	32x32	2.4556
8	32x32	2.4866
16	32x32	2.5278
32	32x32	2.5843
64	32x32	2.5467
1	64x64	3.3006
2	64x64	2.4544
4	64x64	2.1563
8	64x64	2.1881
16	64x64	2.2991
32	64x64	2.3363
64	64x64	2.3757
1	128x128	3.3624
2	128x128	2.5664
4	128x128	2.2314
8	128x128	2.1851
16	128x128	2.3163
32	128x128	2.2464
64	128x128	2.2578
1	256x256	4.1771
2	256x256	2.9417
4	256x256	2.6844
8	256x256	2.7116
16	256x256	2.8341
32	256x256	2.6852
64	256x256	2.8541
1	512x512	5.0987
2	512x512	3.9368
4	512x512	3.5747
8	512x512	3.5422

16	512x512	3.5634
32	512x512	3.6647
64	512x512	3.5626

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	32x32	50.5895
2	32x32	25.595
4	32x32	38.4020
8	32x32	32.9890
16	32x32	28.6368
32	32x32	18.9066
64	32x32	12.5939
1	64x64	31.7764
2	64x64	23.8498
4	64x64	18.6119
8	64x64	19.2848
16	64x64	19.3208
32	64x64	19.3541
64	64x64	19.2581
1	128x128	25.7023
2	128x128	19.1042
4	128x128	15.2168
8	128x128	15.7492
16	128x128	15.6858
32	128x128	16.1533
64	128x128	16.0925
1	256x256	24.7407
2	256x256	17.8363
4	256x256	14.2277
8	256x256	14.3748
16	256x256	14.6841
32	256x256	14.4214
64	256x256	14.5351
1	512x512	31.2599
2	512x512	22.3274
4	512x512	19.5011
8	512x512	19.2422
16	512x512	18.7140

32	512x512	18.7891
64	512x512	18.8790

Με χρήση της Openmp

1. Parallel-for Static

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
 for(k=0;k<N;k+=B){
    FW(A,k,k,k,B);

#pragma omp parallel shared(A,B,N) private(i,j)
{
    #pragma omp for schedule(static)
    for(i=0; i<k; i+=B)
            FW(A,k,i,k,B);

    #pragma omp for schedule(static)
    for(i=k+B; i<N; i+=B)
            FW(A,k,i,k,B);

    #pragma omp for schedule(static)
    for(j=0; j<k; j+=B)
            FW(A,k,k,j,B);

    #pragma omp for schedule(static)
    for(j=k+B; j<N; j+=B)
            FW(A,k,k,j,B);


    #pragma omp for schedule(static)
    for(i=0; i<k; i+=B)
            for(j=0; j<k; j+=B)
                    FW(A,k,i,j,B);


    #pragma omp for schedule(static)
    for(i=0; i<k; i+=B)
            for(j=k+B; j<N; j+=B)
                    FW(A,k,i,j,B);

    #pragma omp for schedule(static)
    for(i=k+B; i<N; i+=B)
            for(j=0; j<k; j+=B)
                    FW(A,k,i,j,B);

    #pragma omp for schedule(static)
    for(i=k+B; i<N; i+=B)
            for(j=k+B; j<N; j+=B)
                    FW(A,k,i,j,B);
 }
}
```

Ο συγκεκριμένος κώδικας βασίζεται στην δομή parallel for της βιβλιοθήκης openmp και παραλληλοποιεί τις εξής εκτελέσεις:

- Όλα τα block της στήλης k που βρίσκονται πριν από το block A_{kk} εκτελούνται παράλληλα.
- Όλα τα block της στήλης k που βρίσκονται μετά από το block A_{kk} εκτελούνται παράλληλα
- Όλα τα block της γραμμής k που βρίσκονται πριν από το block A_{kk} εκτελούνται παράλληλα.
- Όλα τα block της γραμμής k που βρίσκονται μετά από το block A_{kk} εκτελούνται παράλληλα.
- Όλα τα υπόλοιπα block μπορούν να εκτελεσθούν παράλληλα.

Παρατήρηση: Το block A_{kk} *έχει* υπολογισθεί έξω από τις παράλληλες εκτελέσεις.

Οι παραπάνω παράλληλες εκτελέσεις περιέχονται μέσα σε μία παράλληλη περιοχή στο τέλος της οποίας υπονοείται επίσης barrier το οποίος περιμένει όλα τα νήματα εκτέλεσης να τελειώσουν προκειμένου ο αλγόριθμος να περάσει στο επόμενο χρονικό βήμα k.

Μεταξύ αυτών των παράλληλων εκτελέσεων υπονοείται barrier το οποίο περιμένει όλα τα νήματα να τελειώσουν την μία παράλληλη εκτέλεση για να περάσουν στην επόμενη. Το barrier αυτό υπονοείται εξαιτίας του #pragma omp for schedule(static). Επιπλέον η ανάθεση των υπολογισμών στα threads γίνεται στατικά εξαιτίας του schedule(static).

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads, για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Στις μετρήσεις που ακολουθούν παρουσιάζονται τα αποτελέσματα για μέγεθος Β 64. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	0.6799
2	64x64	0.3870
4	64x64	0.2665
8	64x64	0.1772
16	64x64	0.2040
32	64x64	0.2673
64	64x64	0.3441

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	5.1952
2	64x64	2.6962
4	64x64	1.6294
8	64x64	1.0164
16	64x64	0.8398
32	64x64	0.7165
64	64x64	0.8730

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	40.5202
2	64x64	21.0799
4	64x64	11.2450
8	64x64	6.2612
16	64x64	4.3408
32	64x64	3.6807
64	64x64	3.5471

2.Parallel-for Cyclic

```
for(k=0;k<N;k+=B){
    FW(A,k,k,k,B);

    #pragma omp parallel shared(A,B,k)
    {
            #pragma omp for schedule(static,1)
            for(i=0; i<k; i+=B)
                    FW(A,k,i,k,B);

            #pragma omp for schedule(static,1)
            for(i=k+B; i<N; i+=B)
                    FW(A,k,i,k,B);

            #pragma omp for schedule(static,1)
            for(j=0; j<k; j+=B)
                    FW(A,k,k,j,B);

            #pragma omp for schedule(static,1)
                    for(j=k+B; j<N; j+=B)
                            FW(A,k,k,j,B);



            #pragma omp for schedule(static,1)
            for(i=0; i<k; i+=B)
                    for(j=0; j<k; j+=B)
                            FW(A,k,i,j,B);


            #pragma omp for schedule(static,1)
            for(i=1; i<k; i+=B)
                    for(j=k+B; j<N; j+=B)
                            FW(A,k,i,j,B);

            #pragma omp for schedule(static,1)
            for(i=k+B; i<N; i+=B)
                    for(j=0; j<k; j+=B)
                            FW(A,k,i,j,B);

            #pragma omp for schedule(static,1)
            for(i=k+B; i<N; i+=B)
                    for(j=k+B; j<N; j+=B)
                            FW(A,k,i,j,B);
    }
}
```

Η συγκεκριμένη υλοποίηση δεν διαφέρει σε τίποτα από την προηγούμενη παρά μόνο στο γεγονός ότι χρησιμοποιεί την εντολή #pragma omp for schedule(static,1) αντί για #pragma omp for schedule(static).

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads, για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Στις μετρήσεις που ακολουθούν παρουσιάζονται τα αποτελέσματα για μέγεθος Β 64. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	0.6822
2	64x64	0.3769
4	64x64	0.2393
8	64x64	0.1958
16	64x64	0.2010
32	64x64	0.2354
64	64x64	0.2784

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	5.1955
2	64x64	2.7713
4	64x64	1.6199
8	64x64	0.9741
16	64x64	0.8689
32	64x64	0.8674
64	64x64	0.9695

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	40.4949
2	64x64	21.4786
4	64x64	11.2875
8	64x64	6.1446
16	64x64	4.4709
32	64x64	3.8077
64	64x64	3.6673

3. Parallel-for με χρήση nowait

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
for(k=0;k<N;k+=B){
    FW(A,k,k,k,B);

    #pragma omp parallel shared(A,B,k)
    {
            #pragma omp for nowait
            for(i=0; i<k; i+=B)
                    FW(A,k,i,k,B);

            #pragma omp for nowait
            for(i=k+B; i<N; i+=B)
                    FW(A,k,i,k,B);

            #pragma omp for nowait
            for(j=0; j<k; j+=B)
                    FW(A,k,k,j,B);

            #pragma omp for nowait
                    for(j=k+B; j<N; j+=B)
                            FW(A,k,k,j,B);
    }

    #pragma omp parallel shared(A,B,k)
    {

            #pragma omp for nowait
            for(i=0; i<k; i+=B)
                    for(j=0; j<k; j+=B)
                            FW(A,k,i,j,B);


            #pragma omp for nowait
            for(i=0; i<k; i+=B)
                    for(j=k+B; j<N; j+=B)
                            FW(A,k,i,j,B);

            #pragma omp for  nowait
            for(i=k+B; i<N; i+=B)
                    for(j=0; j<k; j+=B)
                            FW(A,k,i,j,B);

            #pragma omp for  nowait
            for(i=k+B; i<N; i+=B)
                    for(j=k+B; j<N; j+=B)
                            FW(A,k,i,j,B);
```

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η συγκεκριμένη υλοποίηση βασίζεται στις ακόλουθες παραλληλίες:

- Όλα τα block της στήλης k που βρίσκονται πριν από το block A_{kk}, όλα τα block της στήλης k που βρίσκονται μετά από το block A_{kk} , όλα τα block της γραμμής

k που βρίσκονται πριν από το block A_{kk} καθώς και όλα τα block της γραμμής k που βρίσκονται μετά από το block A_{kk} μπορούν να υπολογισθούν παράλληλα.

- Όλα τα υπόλοιπα block μπορούν να εκτελεσθούν παράλληλα.

Το γεγονός λοιπόν ότι όλα τα block του «σταυρού» μπορούν να εκτελεστούν παράλληλα προσπαθεί να εκμεταλλευτεί η συγκεκριμένη υλοποίηση προσθέτοντας μετά την εντολή #pragma omp for schedule(static) το nowait. Έτσι για παράδειγμα το κάθε νήμα δεν χρειάζεται να περιμένει τα υπόλοιπα νήματα να τελειώσουν τον υπολογισμό των blocks της στήλης k που βρίσκονται πριν από το block A_{kk} και μπορούν να συνεχίσουν στους επόμενους υπολογισμούς.

Με το nowait προσπαθούμε λοιπόν να απαλλάξουμε το κώδικα μας από τα περιττά barriers που φέρνει μαζί του το κάθε #pragma omp for.

Βέβαια με την χρήση του nowait δημιουργείται η ανάγκη να προστεθούν εντολές που διασφαλίζουν ότι όλα τα νήματα θα ολοκληρώσουν πρώτα τον υπολογισμών των block του σταυρού για να προχωρήσουν στον υπολογισμό των υπόλοιπων block.

Για τον λόγο αυτό ο υπολογισμός των στοιχείων του σταυρού και ο υπολογισμός των υπόλοιπων στοιχείων περιέχονται σε δύο διαφορετικές παράλληλες περιοχές. Έτσι δημιουργείται ένα barrier το οποίο δημιουργεί τον απαραίτητο συγχρονισμό ώστε να τηρούνται οι εξαρτήσεις.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads , για μέγεθος πίνακα A 1024x1024, 2048x2048, 4096x4096 καθώς και για μέγεθος block B = 64 . Τα αποτελέσματα φαίνονται στην συνέχεια:

Για A 1024 x 1024:

Αριθμός νημάτων	Μέγεθος B	Χρόνος Εκτέλεσης (sec)
1	64x64	0.6773
2	64x64	0.4961
4	64x64	0.2460
8	64x64	0.1621
16	64x64	0.1266
32	64x64	0.1504
64	64x64	0.1951

Για A 2048 x 2048:

Αριθμός νημάτων	Μέγεθος B	Χρόνος Εκτέλεσης (sec)
1	64x64	5.2026
2	64x64	2.8683
4	64x64	1.5771
8	64x64	0.9378
16	64x64	0.5970

32	64x64	0.4635
64	64x64	0.5694

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	40.5148
2	64x64	22.7669
4	64x64	11.5559
8	64x64	6.2192
16	64x64	3.8149
32	64x64	2.3310
64	64x64	1.7766

3. Parallel-for με χρήση nowait και collapse(2)

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
for(k=0;k<N;k+=B){
   FW(A,k,k,k,B);

   #pragma omp parallel shared(A,B,k)
   {
           #pragma omp for nowait
           for(i=0; i<k; i+=B)
                   FW(A,k,i,k,B);

           #pragma omp for nowait
           for(i=k+B; i<N; i+=B)
                   FW(A,k,i,k,B);

           #pragma omp for nowait
           for(j=0; j<k; j+=B)
                   FW(A,k,k,j,B);

           #pragma omp for nowait
                   for(j=k+B; j<N; j+=B)
                           FW(A,k,k,j,B);
   }

   #pragma omp parallel shared(A,B,k)
   {

           #pragma omp for collapse(2) nowait
           for(i=0; i<k; i+=B)
                   for(j=0; j<k; j+=B)
                           FW(A,k,i,j,B);


           #pragma omp for collapse(2) nowait
           for(i=0; i<k; i+=B)
                   for(j=k+B; j<N; j+=B)
                           FW(A,k,i,j,B);

           #pragma omp for collapse(2) nowait
           for(i=k+B; i<N; i+=B)
                   for(j=0; j<k; j+=B)
                           FW(A,k,i,j,B);
```

```
#pragma omp for collapse(2) nowait
for(i=k+B; i<N; i+=B)
        for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);
```

Η συγκεκριμένη υλοποίηση δεν διαφέρει σε τίποτα με την προηγούμενη με την διαφορά ότι στους διπλούς βρόγχους έχει προστεθεί η εντολή collapse(2) η οποία ενώνει τον βρόγχο για να τον παραλληλοποιήσει.

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads, για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Στις μετρήσεις που ακολουθούν παρουσιάζονται τα αποτελέσματα για μέγεθος Β 64. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	0.6781
2	64x64	0.3738
4	64x64	0.2219
8	64x64	0.1545
16	64x64	0.1369
32	64x64	0.2145
64	64x64	0.2433

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	5.1664
2	64x64	2.6177
4	64x64	1.4155
8	64x64	0.8351
16	64x64	0.6990
32	64x64	0.5977
64	64x64	0.6216

Για Α 4096 x 4096:

Αριθμός νημάτων	Μέγεθος B	Χρόνος Εκτέλεσης (sec)
1	64x64	40.2495
2	64x64	21.1699
4	64x64	10.6977
8	64x64	5.7521
16	64x64	3.3269
32	64x64	2.7837
64	64x64	1.9843

5. Parallel-for με ένωση των loops

Ο κώδικας της συγκεκριμένης υλοποίησης είναι ο ακόλουθος:

```
for(k=0;k<N;k+=B){
    FW(A,k,k,k,B);

    #pragma omp parallel shared(A,B,k) private(i,j)
    {
            #pragma omp for schedule(static)
            for(i=0; i<N; i+=B){
                    if(i!=k){
                            FW(A,k,i,k,B);
                    }
            }

            #pragma omp for schedule(static)
            for(j=0; j<N; j+=B){
                    if(j!=k){
                            FW(A,k,k,j,B);
                    }
            }



            #pragma omp for schedule(static)
            for(i=0; i<N; i+=B){
                    if(i!=k){
                            for(j=0; j<N; j+=B){
                                    if(j!=k){
                                            FW(A,k,i,j,B);
                                    }
                            }
                    }
            }

}
```

Ο συγκεκριμένος κώδικας βασίζεται στο γεγονός πως για παράδειγμα ο υπολογισμός των block που βρίσκονται στην γραμμή k και πριν από το διαγώνιο στοιχείο A_{kk} και ο υπολογισμός των block που βρίσκονται που στην γραμμή k και πριν από το διαγώνιο στοιχείο A_{kk} μπορούν να υπολογισθούν σε ένα loop. Αυτό που απαιτείται είναι η προσθήκη μίας εντολής if-statement που θα εξασφαλίζει ότι δεν θα ξαναυπολογιστεί το διαγώνιο στοιχείο. Ίδια λογική ακολουθείται και για τα υπόλοιπα block.

Έπειτα η παραλληλοποίηση μπορεί να γίνει κανονικά όπως και στην περίπτωση 1.

Σημείωση: Για λόγους χώρου παρατέθηκε μόνο το κομμάτι του κώδικα που πραγματοποιήθηκε η παραλληλοποίηση.

Η παραπάνω υλοποίηση εκτελέστηκε για 1,2,4,8,16,32 και 64 threads, για μέγεθος πίνακα Α 1024x1024, 2048x2048, 4096x4096. Στις μετρήσεις που ακολουθούν παρουσιάζονται τα αποτελέσματα για μέγεθος Β 64. Τα αποτελέσματα φαίνονται στην συνέχεια:

Για Α 1024 x 1024:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	0.6739
2	64x64	0.3738
4	64x64	0.2054
8	64x64	0.1349
16	64x64	0.1357
32	64x64	0.1555
64	64x64	0.2046

Για Α 2048 x 2048:

Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	5.1988
2	64x64	2.6902
4	64x64	1.4170
8	64x64	0.8596
16	64x64	0.7513
32	64x64	0.5014
64	64x64	0.6960

Για Α 4096 x 4096:

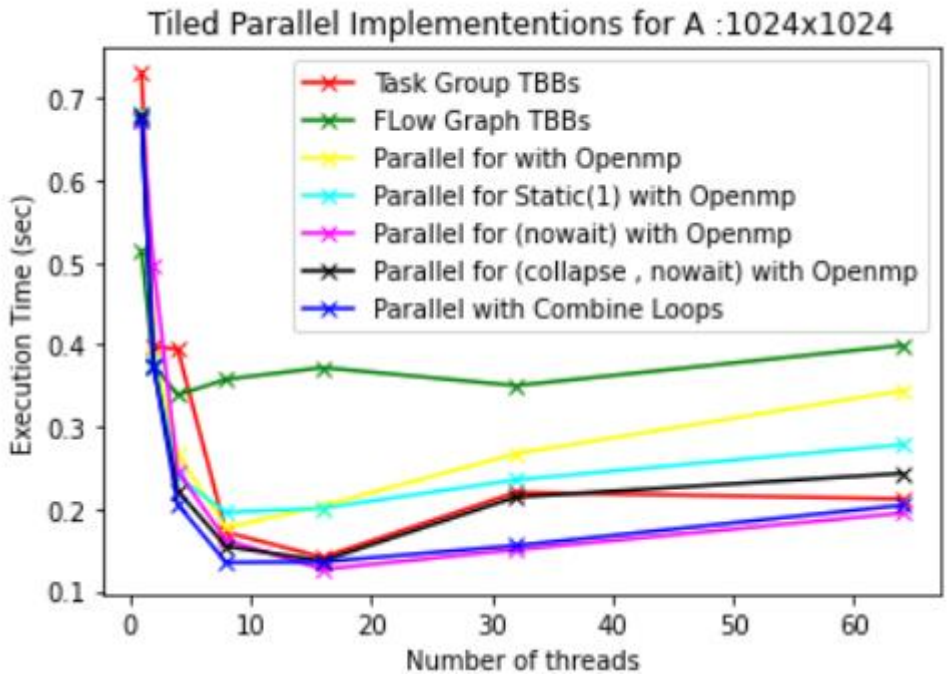
Αριθμός νημάτων	Μέγεθος Β	Χρόνος Εκτέλεσης (sec)
1	64x64	40.4476
2	64x64	21.0300
4	64x64	10.9523

8	64x64	5.5183
16	64x64	3.6551
32	64x64	2.8240
64	64x64	1.8088

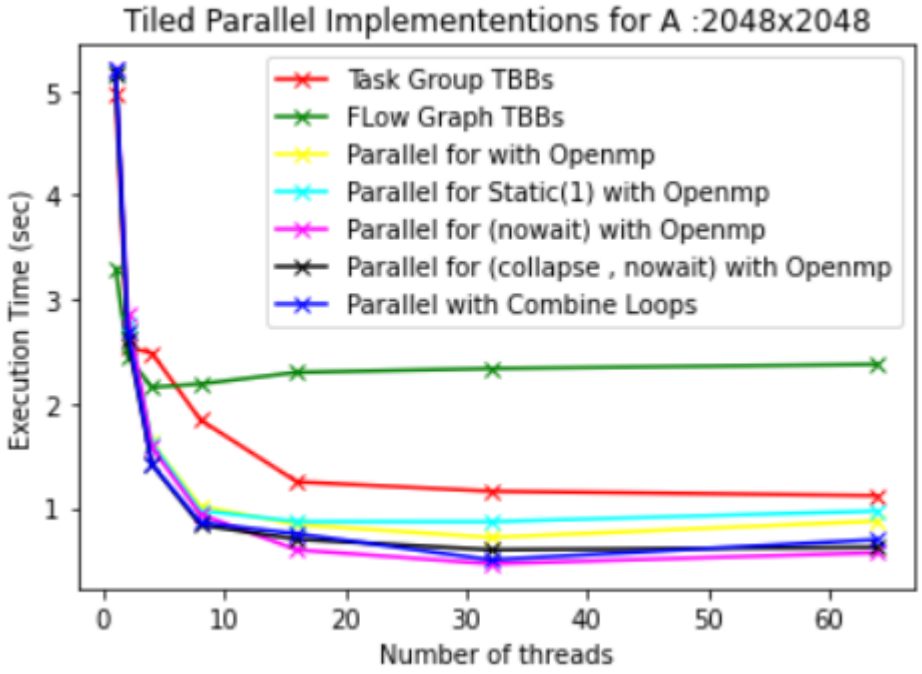
Στην συνέχεια παραθέτουμε ορισμένα διαγράμματα τα οποία μας δείχνουν τον χρόνο εκτέλεσης της κάθε μίας από τις 3 προηγούμενες υλοποιήσεις για κάθε ένα από τα μεγέθη του πίνακα Α. Το μέγεθος του Β επιλέχθηκε να είναι 64 προκειμένου να γίνουν οι συγκρίσεις. Όπως παρατηρήσαμε από τις μετρήσεις (κάποιες δεν εμφανίζονται στην αναφορά για λόγους χώρου) το 64 είναι το νούμερο όπου γενικά παρατηρούνται οι καλύτερες μετρήσεις του Β. Σε κάποιες περιπτώσεις βέβαια για μέγεθος πίνακα Α:2048x2048 και Α:4096x4096 παρατηρήθηκε ότι η καλύτερη τιμή του Β είναι 128. Κάτι τέτοιο έρχεται σε συμφωνία με την αρχική μας παρατήρηση για την τιμή του Β που κάναμε στην σειριακή tiled υλοποίηση.

Τα διαγράμματα είναι τα ακόλουθα:

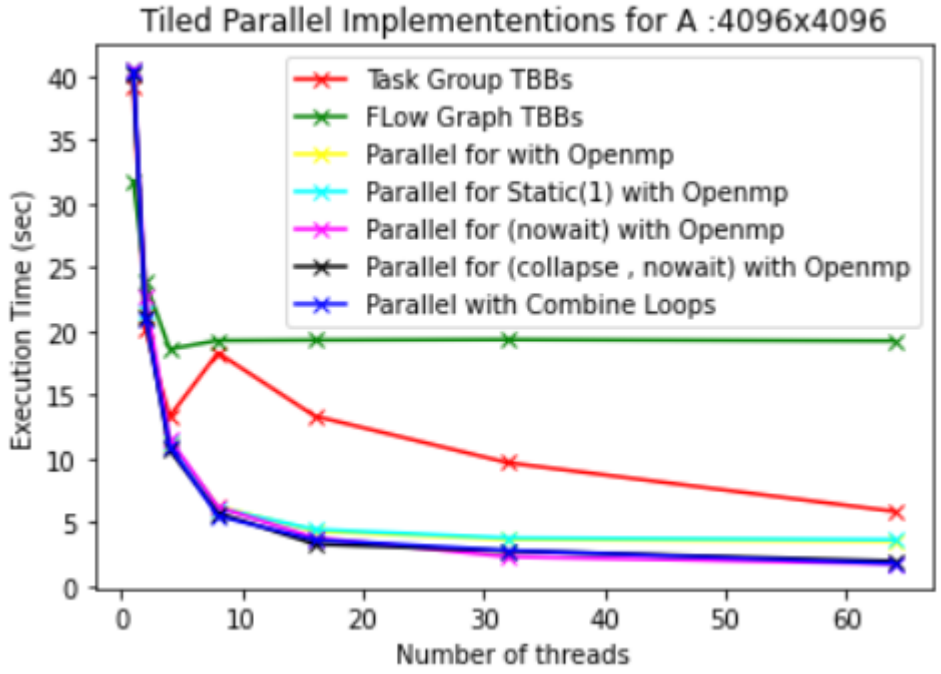
A:1024x1024:



A:2048x2048:



A:4096x409:



Παρατηρήσεις:

Από τις μετρήσεις και τα παραπάνω διαγράμματα μπορούμε να κάνουμε τις ακόλουθες παρατηρήσεις για τις παράλληλες εκδόσεις του tiled FW:

- Η χρήση των task groups των TBBs αποδείχτηκε καλύτερη από την χρήση του Flow Graph όπως έγινε και στην recursive υλοποίηση.
- Το Flow-Graph μέσω των TBBs φαίνεται να είναι η χειρότερη από όλες τις υλοποιήσεις.
- Ανάμεσα στις υλοποιήσεις με Openmp η απλή static και η static(1) φαίνεται πως είναι χειρότερες κάτι το οποίο παρατηρείται πιο έντονα όσο αυξάνεται το μέγεθος του πίνακα Α. Αυτό επαληθεύει και την παραπάνω παρατήρηση μας για το πως οι άλλες τρεις υλοποιήσεις με openmp εκμεταλλεύονται καλύτερα την παραλληλία. Από τις δύο αυτές υλοποιήσεις ίσως λίγο καλύτερη είναι η static χωρίς ωστόσο να παρουσιάζει κάποια σημαντική διαφορά.
- Οι υπόλοιπες τρεις υλοποιήσεις είναι αρκετά καλές. Η προσθήκη του collapse(2) δεν φαίνεται να δίνει κάτι επιπλέον όσο αναφορά την επίδοση του κώδικα. Επιπλέον και η τελευταία υλοποίηση παρουσιάζει λίγο χειρότερα αποτελέσματα. Έτσι καλύτερη θεωρούμε ότι είναι η υλοποίηση με parallel for και nowait της Openmp.
- Καθώς αυξάνεται το μέγεθος του πίνακα Α παρατηρείται καλύτερη κλιμάκωση . Έτσι για παράδειγμα για Α:4096x4096 υλοποίηση με parallel for και nowait της Openmp (αλλά και οι άλλες δύο που αναφέραμε στο προηγούμενο bullet) υποδιπλασιάζει τον χρόνο της ακόμα και με την αύξηση των νημάτων από 32 σε 64. Κάτι τέτοιο είναι λογικό καθώς για Β:64x64 και Α:1024x1024 δημιουργούνται συνολικά 16 blocks, για Β :64x6464 και Α:2048x2048 δημιουργούνται συνολικά 32 blocks ενώ για Β:64x6464 και Α:4096x4096 δημιουργούνται συνολικά 64 blocks. Έτσι η αύξηση του πίνακα Α για σταθερό Β αυξάνει και τις δυνατότητες παραλληλίας γεγονός που κάνει όλο και περισσότερα νήματα να είναι χρήσιμα. Για το λόγο αυτό η κλιμάκωση (ή αντίστοιχα το γραμμικό speedup) παρατηρούνται ακόμα και για 32 με 64 threads όταν το μέγεθος του πίνακα Α είναι 4096x4096.

Συγκριτικές Παρατηρήσεις μεταξύ των τριών εκδόσεων και της παραλληλίας τους

Ως γενικές παρατηρήσεις μπορούμε να πούμε τις ακόλουθες:

- Η recursive υλοποίηση ήταν η υλοποίηση που διέθετε την λιγότερη παραλληλία. Για τον λόγο αυτό το speedup των υλοποιήσεων δεν ήταν γραμμικό και οι υλοποιήσεις δεν κλιμάκωναν ικανοποιητικά ενώ παρατηρήθηκαν και περιπτώσεις που η αύξηση των threads πέρα από ένα σημείο οδηγούσε και σε αύξηση του χρόνου εκτέλεσης.
- Η standard υλοποίηση ήταν πιο παραλληλοποιήσιμη γεγονός που έδωσε και μεγαλύτερη κλιμάκωση αλλά και μεγαλύτερη πτώση του χρόνου εκτέλεσης.

Ο χρόνος εκτέλεσης δεν μπόρεσε όμως να μειωθεί περεταίρω εξαιτίας του γεγονότος πως η υλοποίηση αυτή είναι memory bound όπως έχουμε εξηγήσει.

- Η tiled υλοποίηση είχε την μεγαλύτερη δυνατότητα παραλληλίας καθώς συνδυάζει και τα καλά που δίνει το parallel for της openmp χωρίς να έχει κάποιο bottleneck μνήμης λόγο της καλής εκμετάλλευσης της cache και του data-localilty που προσφέρει. Η κλιμάκωση ήταν πολύ μεγαλύτερη από της άλλες περιπτώσεις ενώ ο χρόνος εκτέλεσης ακόμα και για μέγεθος πίνακα A:4096x4096 έπεσε κάτω από τα 2sec.

Καλύτερες παράλληλες υλοποιήσεις για την κάθε έκδοση και οι καλύτεροι χρόνοι εκτέλεσης που επιτεύχθηκαν.

Για την Standard Υλοποίηση:

- Static parallel for μέσω Openmp
- Για A:1024x1024 : 0.2182sec με 32 threads
- Για A:2048x2048 : 0.9845sec με 32 threads
- Για A:4096x4096 : 17.6361sec με 64 threads

Για την Recursive Υλοποίηση (B:64x64):

- Task Groups μέσω TBBs
- Για A:1024x1024 : 0.3286sec με 4 threads
- Για A:2048x2048 : 2.1021sec με 4 threads
- Για A:4096x4096 : 13.0899sec με 8 threads

Για την Tiled Υλοποίηση(B:64x64):

- Parallel for με nowait μέσω Openmp.
- Για A:1024x1024 : 0.1266sec με 16 threads
- Για A:2048x2048 : 0.4635sec με 32 threads
- Για A:4096x4096 : 1.7766sec με 64 threads

Τα παραπάνω αποτελέσματα μπορούν να μας δώσουν και τις ακόλουθες παρατηρήσεις:

- Η recursive όπως είπαμε και προηγουμένως εμφανίζει την μικρότερη παραλληλία. Οι ελάχιστες τιμές στους χρόνους εκτέλεσης παρατηρούνται για 4-8 threads. Αντίθετα η standard υλοποίηση έχει καλύτερη παραλληλία και γι αυτό οι ελάχιστες τιμές στους χρόνους εκτέλεσης είναι στα 16-32 threads.

Μάλιστα μέσω της παραλληλίας η standard γίνεται τελικά πιο γρήγορη για μέγεθος πίνακα Α 1024x1024 και 2048x2048 . Όμως το γεγονός ότι είναι memory bound φαίνεται για το μέγεθος πίνακα Α 4096x4096 όπου εκεί ο χρόνος της και μετά την παραλληλοποίηση είναι αρκετά μεγαλύτερος.

- Η tiled υλοποίηση παρουσιάζει την μεγαλύτερη παραλληλία και κλιμάκωση. Γι΄ αυτό και ο ελάχιστος χρόνος εκτέλεσης επιτυγχάνεται για 16,32,64 threads. Μάλιστα η αύξηση του μεγέθους του πίνακα Α δημιουργεί μεγαλύτερη παραλληλία (για το ίδιο Β όπως αναφέραμε και σε προηγούμενες παρατηρήσεις) και έτσι ο ελάχιστος χρόνος παρατηρείται για μεγαλύτερο αριθμό νημάτων.

Ερώτημα 3

Κατανόηση της Αρχιτεκτονικής του Μηχανήματος Sandman αλλά και των δυνατοτήτων του μεταγλωττιστή προκειμένου να μειώσουμε περαιτέρω τον χρόνο εκτέλεσης.

<u>Μηχάνημα Sandman</u>

Γνωρίζουμε ότι το μηχάνημα Sandman αποτελείται από 4 κόμβους. Κάθε κόμβος αποτελείται από 8 πυρήνες και σε κάθε πυρήνα τρέχουν δύο οντότητες εκτέλεσης (hardware threads) Συνολικά λοιπόν μπορούμε να χρησιμοποιήσουμε μέχρι 64 οντότητες εκτέλεσης (threads) προκειμένου να εκτελέσουμε τις παράλληλες εκδόσεις των προγραμμάτων μας.

Εφόσον η αρχιτεκτονική του μηχανήματος μας είναι NUMA (non-uniform memory access) κάθε κόμβος διαθέτει την δικιά του μνήμη RAM. Έτσι το κόστος ένος νήματος που τρέχει σε έναν κόμβο να αποκτήσει πρόσβαση σε στοιχεία που βρίσκονται στην του κόμβου αυτού είναι πολύ μικρότερο από το να αποκτήσει πρόσβαση σε στοιχεία που βρίσκονται σε μνήμη άλλου κόμβου.

Ο κάθε πυρήνας αποτελείται από μία L3-cache μεγέθους 16Mib = 2^4 Mib = $2^4 * 2^{20}$ Byte = 2^{24} Bytes . Παρατηρούμε λοιπόν ότι η μέγιστη χωρητικότητα της L3-cache του κάθε πυρήνα είναι 2^{24} bytes = 16.777.216 bytes = . Έτσι για το κάθε μέγεθος του πίνακα γειτνίασης Α έχουμε:

- Size_A *(4byte) = (1024x1024) * 4 = 1.048.576 * 4 = 4.194.304 bytes. Επομένως ο πίνακας Α με μέγεθος 1024x1024 χωράει εξολοκλήρου στην L3-cache ενός node.
- Size_A* (4byte) = (2048x2048) * 4 = 16.777.216 bytes. Επομένως ο πίνακας Α με μέγεθος 2048x2048 χωράει ακριβώς στην L3-cache ενός node.
- Size_A* (4byte) = (4096x4096) * 4 = 16.777.216 * 4 = 67.108.864 bytes. Επομένως ο πίνακας Α με μέγεθος 4096x4096 δεν χωράει ολόκληρος στην L3-cache ενός node.

Σχόλιο: Πολλαπλασιάζουμε το μέγεθος του πίνακα Α με 4 bytes ώστε να βρούμε τα συνολικά byte που είναι αποθηκευμένα στον πίνακα Α. Ο πολλαπλασιασμός γίνεται με 4 bytes καθώς κάθε στοιχείο του πίνακα Α είναι τύπου int το οποίο είναι 4 bytes.

Παρατήρηση: Το γεγονός ότι ο πίνακας Α μεγέθους 4096x4096 δεν χωράει ολόκληρος στην L3-cache εξηγεί και την μεγάλη αύξηση που παρατηρείται στον χρόνο εκτέλεσης της standard υλοποίησης του FW όταν ο πίνακας γειτνίασης Α έχει μέγεθος 4096x4096. Βέβαια υποθέτουμε πως και στην περίπτωση που ο πίνακας Α έχει μέγεθος 2048x2048 θα έχουμε κάποια misses καθώς ο πίνακας χωράει ακριβώς στην L3-cache. Όμως στην L3-cache θα αποθηκευτούν και μεταβλητές του προγράμματος ενδεχομένως και άλλα στοιχεία.

ΜΕ βάση την περιγραφή της αρχιτεκτονικής του μηχανήματος sandman παραθέτουμε δύο βελτιστοποιήσεις οι οποίες θα βοηθήσουν στην περεταίρω μείωση του χρόνου εκτέλεσης εκμεταλλευόμενες ακριβώς την αρχιτεκτονική αυτή. Έτσι έχουμε:

Threads Affinity

Μια λογική η οποία θα εκμεταλλευτεί καλύτερα το data locality είναι το threads affinity. Η γενική ιδέα πίσω από το threads affinity είναι το «δέσιμο των threads» με συγκριμένους πυρήνες σε όλη την διάρκεια που εκτελούν υπολογισμούς. Γενικά το λειτουργικό σύστημα πολλές φορές μεταφέρει τα threads βάζοντας τα να εκτελούνται σε διαφορετικούς πυρήνες. Κάτι τέτοιο το κάνει για να βελτιστοποιήσει το scheduling. Η μεταφορά όμως των threads σε διαφορετικούς πυρήνες διαφορετικών κόμβων μπορεί να οδηγήσει σε μη καλή εκμετάλλευση του data locality. Καθώς στον αλγόριθμο του Floyd-Warshall όπως έχουμε εξηγήσει τα νήματα επαναχρησιμοποιούν δεδομένα αν μεταφερθούν σε άλλον κόμβο δεν θα έχουν πλέον τα δεδομένα αυτά ούτε στην cache τους αλλά μπορεί και ούτε στην μνήμη του node τους. Έτσι ο χρόνος εκτέλεσης αυξάνεται σηματνικά.

Προκειμένου να πετύχουμε αυτό το δέσιμο θέτουμε την μεταβλητή περιβάλλοντος **OMP_PROC_BIND σε true**. Έτσι κάθε νήμα δένεται με τον πυρήνα στον οποίο ξεκινάει να τρέχει.

First Touch Policy

Μία δεύτερη βελτιστοποίηση η οποία έχει ως σκοπό δεδομένα που χρησιμοποιεί ένα συγεκριμένο thread να βρίσκονται όλο και πιο κοντά σε αυτό, δηλαδή στην μνήμη του node που εκτελείται, είναι η first touch policy. Η first touch policy λέει το εξής:

- Σε ποιανού node την μνήμη θα γίνει μεταφερθεί ο πίνακας Α δεν ορίζεται όταν ζητηθεί από το λειτουργικό σύστημα χώρος για τον πίνακα Α (malloc , memory allocation). Με την malloc απλά μας παρέχονται εικονικές διευθύνσεις από το λειτουργικό σύστημα.
- Η πρόσβαση σε φυσικές θέσεις μνήμης γίνεται αντίθετα όταν αρχικοποιηθεί ο πίνακας Α.

Στην περίπτωση της άσκησης όμως παρατηρούμε ότι στον αρχικό κώδικα ο πίνακας Α αρχικοποιείται από ένα thread (το master thread). Έτσι όλος ο πινακας Α μεταφέρεται στην μνήμη του node όπου εκτελείται το master thread. Κάτι τέτοιο δημιουργεί μεγαλύτερο κόστος σε threads που εκτελούνται εκτός του συγκεκριμένου node να αποκτήσουν πρόσβαση σε στοιχεία του πίνακα. Μία βελτιστοποίηση λοιπόν θα ήταν και η αρχικοποίηση του πίνακα Α να γίνεται και αυτή παράλληλα. Έτσι το κομμάτι του πίνανα Α που αρχικοποιεί ένα thread θα μεταφέρεται στην μνήμη του node που αυτό εκτελείται μειώνοντας έτσι και το κόστος πρόσβασης στα δεδομένα.

Η αρχικοποίηση του πίνακα Α μεταβλήθηκε στο αρχείο util.c για την standard και την tiled υλοποίηση και ο κώδικας είναι ο ακόλουθος:

```
void graph_init_random(int **adjm, int seed, int n,  int m)
{
     unsigned  int i, j;

     srand48(seed);
     #pragma omp parallel for schedule(static) shared(adjm,n) private(i,j)
     for(i=0; i<n; i++)
        for(j=0; j<n; j++)
           adjm[i][j] = abs((( int)lrand48()) % 1048576);

     for(i=0; i<n; i++)adjm[i][i]=0;
}
```

Ερώτημα 4

Χρησιμοποιώντας τις δύο παραπάνω βελτιστοποιήσεις ξαναμετρήσαμε τους χρόνους εκτέλεσης για τις καλύτερες παράλληλες υλοποιήσεις της standard και της tiled υλοποίησης. Τα αποτελέσματα που πήραμε είναι τα ακόλουθα:

Καλύτερες υλοποιήσεις για κάθε έκδοση:

Για την Standard Υλοποίηση:

- Static parallel for μέσω Openmp
- Για A:1024x1024 : 0.1503sec με 32 threads
- Για A:2048x2048 : 0.7732sec με 64 threads
- Για A:4096x4096 : 6.7358sec με 32 threads

Για την Tiled Υλοποίηση(B:64x64):

- Parallel for με nowait μέσω Openmp.
- Για A:1024x1024 : 0.0809sec με 32 threads
- Για A:2048x2048 : 0.27695sec με 32 threads
- Για A:4096x4096 : 1.3513sec με 64 threads

Για την Recursive Υλοποίηση(B:64x64):

- Στην recursive υλοποίηση δεν είδαμε κάποια βελτιστοποίηση να έχει οδηγήσει σε καλύτερα αποτελέσματα οπότε οι μετρήσεις είναι αυτές που παραθέσαμε προηγουμένως.

Σημείωση: Σε όλες τις μετρήσεις χρησιμοποιήθηκε το flag -03 του μεταγλωττιστή για βελτιστοποίηση.

Ερώτημα 5

Συνοψίζοντας οι καλύτερες παράλληλες εκδόσεις για κάθε αρχική έκδοση μετά από όλες τις βελτιστοποιήσεις είναι οι ακόλουθες:

Για την Standard Υλοποίηση:

- Static parallel for μέσω Openmp
- Για A:1024x1024 : 0.1503sec με 32 threads
- Για A:2048x2048 : 0.7732sec με 64 threads
- Για A:4096x4096 : 6.7358sec με 32 threads

Για την Recursive Υλοποίηση (B:64x64):

- Task Groups μέσω TBBs
- Για A:1024x1024 : 0.3286sec με 4 threads
- Για A:2048x2048 : 2.1021sec με 4 threads
- Για A:4096x4096 : 13.0899sec με 16 threads

Για την Tiled Υλοποίηση(B:64x64):

- Parallel for με nowait μέσω Openmp.
- Για A:1024x1024 : 0.0809sec με 32 threads
- Για A:2048x2048 : 0.27695sec με 32 threads
- Για A:4096x4096 : 1.3513sec με 64 threads

Ο καλύτερος χρόνος που επιτεύχθηκε για κάθε μέγεθος του πίνακα Α είναι ο ακόλουθος:

- **Για Α:1024x1024 : 0.0809sec από την Tiled παράλληλη έκδοση**
- **Για Α:2048x2048 : 0.27695sec από την Tiled παράλληλη έκδοση**
- **Για Α:4096x4096 : 1.3513sec από την Tiled παράλληλη έκδοση**