Κουτρούμπας Αθανάσιος 03116073
Αραπίδης Μανώλης 03116071

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
1η ΑΣΚΗΣΗ

Παραλληλοποίηση αλγορίθμων σε πολυπύρηνες αρχιτεκτονικές κοινής μνήμης

Ακ. έτος 2020-2021, 9ο Εξάμηνο, Σχολή ΗΜ&ΜΥ

1.4 ΖΗΤΟΥΜΕΝΑ

1.4.1

Εξοικειωθείτε με τον τρόπο μεταγλώττισης και υποβολής εργασιών στις συστοιχίες.

Εξηγούμε αναλυτικά τα βήματα μεταγλώττισης και υποβολής των εργασιών στο **1.4.3**.

1.4.2

Αναπτύξτε παράλληλο πρόγραμμα στο μοντέλο κοινού χώρου διευθύνσεων (shared address space) με τη χρήση του OpenMP.

Το κομμάτι του κώδικα που έχει το μεγαλύτερο φορτίο φαίνεται παρακάτω πως το παραλληλοποιήσαμε:

```
for ( t = 0 ; t < T ; t++ ) {
      #pragma omp parallel for schedule(static) private(j, nbrs)
            for ( i = 1 ; i < N-1 ; i++ )
                  for ( j = 1 ; j < N-1 ; j++ ) {
                        nbrs = previous[i+1][j+1] + previous[i+1][j] + \
                        previous[i+1][j-1] + previous[i][j-1] + \
                        previous[i][j+1] + previous[i-1][j-1] + \
                        previous[i-1][j] + previous[i-1][j+1];
                  if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                        current[i][j]=1;
                  else
                        current[i][j]=0;
}
```

Παρατηρήσαμε ότι τα δύο εσωτερικά loops είναι πλήρως ανεξάρτητα, άρα και παραλληλοποιήσιμα. Αυτό συμβαίνει, διότι οι τιμές στα κελιά current, εξαρτώνται μόνο από τα γειτονικά τους κελιά στο πίνακα previous, και δεν υπάρχει καμία εξάρτηση μεταξύ τους.

Επίσης, δεν υπάρχουν race conditions, διότι το κάθε thread θα γράφει σε ξεχωριστή γραμμή του κοινού πίνακα και έχουμε δηλώσει τις μεταβλητές `nbrs` και `j` ως private ώστε να μην επηρεάζονται οι τιμές τους μεταξύ των threads.

1.4.3

Πραγματοποιήστε μετρήσεις επίδοσης σε έναν από τους κόμβους της συστοιχίας των clones για 1,2,4,6,8 πυρήνες και μεγέθη ταμπλώ 64×64, 1024×1024 και 4096×4096 (σε όλες τις περιπτώσεις τρέξτε το παιχνίδι για 1000 γενιές).

Για την πραγματοποίηση των μετρήσεων, χρησιμοποιούμε για την μεταγλώττιση το script `make_on_queue.sh` το οποίο φαίνεται παρακάτω:

```
#!/bin/bash

## Give the Job a descriptive name
# PBS -N make_omp_Game_Of_Life

## Output and error files
# PBS -e make_omp_Game_Of_Life.err
# PBS -o make_omp_Game_Of_Life.out

## How many machines should we get?
# PBS -l nodes=1:ppn=1

##How long should the job run for?
# PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab07/a1/src/Game_Of_Life
make
```

Για την εκτέλεση των μετρήσεων για κάθε συνδυασμό πυρήνων και μεγεθών ταμπλώ, χρησιμοποιούμε το script `run_on_queue.sh`, το οποίο φαίνεται παρακάτω. Στην ουσία, ορίζουμε ένα πίνακα για τα threads και έναν για τα μεγέθη των ταμπλό. Έπειτα για κάθε size και κάθε thread, κάνουμε export τα threads που θέλουμε μέσω της environment variable `OMP_NUM_THREADS` και τρέχουμε το εκτελέσιμο πρόγραμμα.

```
# !/bin/bash

## Give the Job a descriptive name
# PBS -N run_omp_Game_Of_Life

## Output and error files
# PBS -o run_omp_Game_Of_Life.out
# PBS -e run_omp_Game_Of_Life.err

## How many machines should we get?
# PBS -l nodes=1:ppn=8

##How long should the job run for?
# PBS -l walltime=00:01:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab07/a1/src/Game_Of_Life

FILE=metrics.txt
nthreads=( 1 2 4 6 8 )
sizes=( 64 1024 4096)

if [ -f "$FILE" ]; then
      rm $FILE
fi
for size in "${sizes[@]}";
do
      for nthread in "${nthreads[@]}";
      do
            export OMP_NUM_THREADS=${nthread};
            echo -n "Threads ${nthread} " >> $FILE
            ./Game_Of_Life ${size} 1000 >> $FILE
      done
            echo >> $FILE
done
```

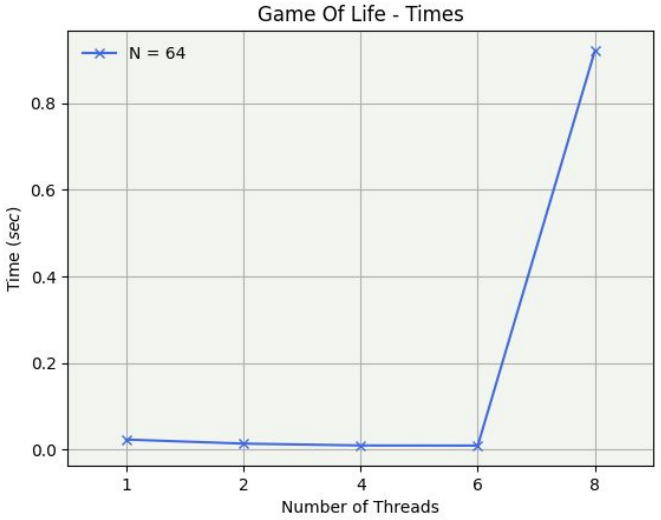
Τα αποτελέσματα αποθηκεύονται στο αρχείο `metrics.txt` και φαίνονται παρακάτω:

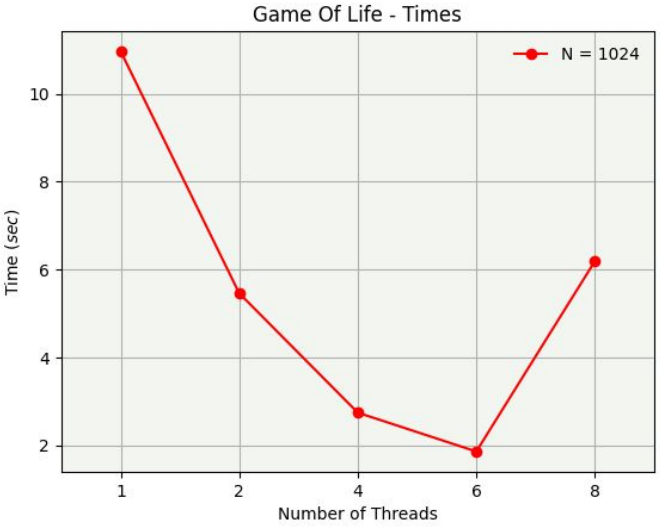
```
Threads 1 GameOfLife: Size 64 Steps 1000 Time 0.023125
Threads 2 GameOfLife: Size 64 Steps 1000 Time 0.013745
Threads 4 GameOfLife: Size 64 Steps 1000 Time 0.009466
Threads 6 GameOfLife: Size 64 Steps 1000 Time 0.009147
Threads 8 GameOfLife: Size 64 Steps 1000 Time 0.921568

Threads 1 GameOfLife: Size 1024 Steps 1000 Time 10.965569
Threads 2 GameOfLife: Size 1024 Steps 1000 Time 5.458503
Threads 4 GameOfLife: Size 1024 Steps 1000 Time 2.741920
Threads 6 GameOfLife: Size 1024 Steps 1000 Time 1.856764
Threads 8 GameOfLife: Size 1024 Steps 1000 Time 6.187554

Threads 1 GameOfLife: Size 4096 Steps 1000 Time 175.871665
Threads 2 GameOfLife: Size 4096 Steps 1000 Time 88.288967
Threads 4 GameOfLife: Size 4096 Steps 1000 Time 44.582814
Threads 6 GameOfLife: Size 4096 Steps 1000 Time 34.739006
Threads 8 GameOfLife: Size 4096 Steps 1000 Time 41.281937
```

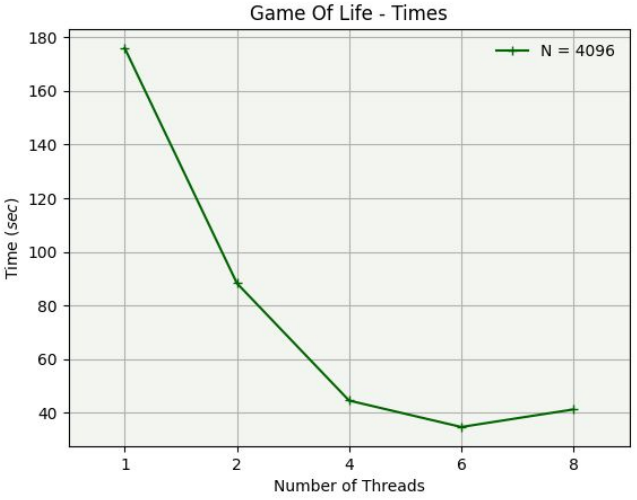
1.4.4

Συγκεντρώστε τα αποτελέσματά σας σε γραφήματα για το χρόνο και το speedup, καθώς και τα σχόλιά σας, στην ενδιάμεση αναφορά.

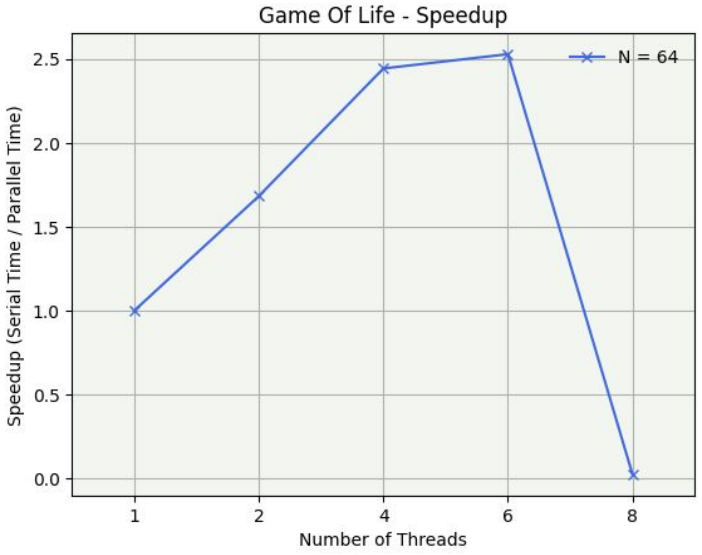
Διαγράμματα Χρόνου

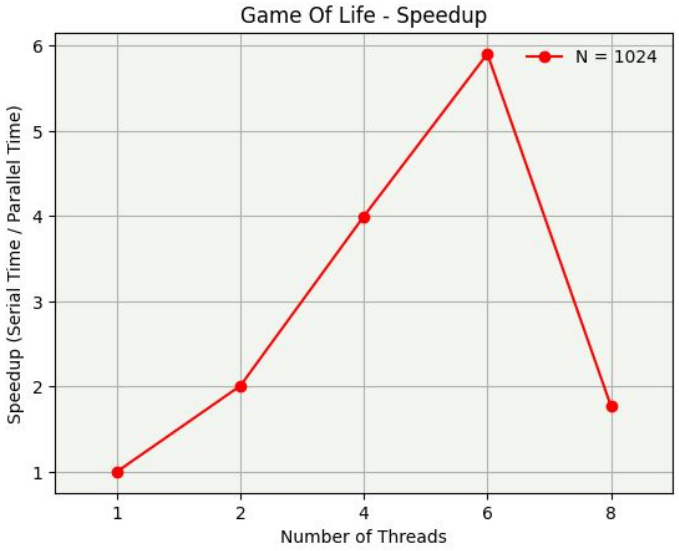


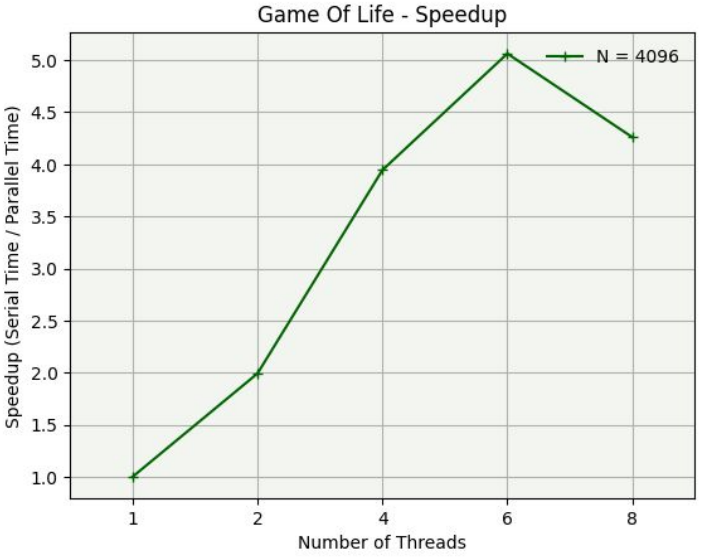




Διαγράμματα Speedup







Συμπεράσματα

Παρατηρούμε ότι το πρόγραμμα μας κλιμακώνει σωστά μέχρι τους 6 πυρήνες, ενώ για τους 8 πυρήνες παρατηρούμε ότι έχουμε μια απότομη αύξηση στο χρόνο εκτέλεσης, για όλα τα μεγέθη ταμπλό.

Για αυξανόμενα μεγέθη ταμπλό, παρατηρούμε ότι η παραπάνω συμπεριφορά έχει σταδιακά μικρότερη επίδραση.
π.χ. για N=64, ο χρόνος εκτέλεσης στα 8 threads είναι κατά πολύ μεγαλύτερος και από τον χρόνο σε 1 thread, ενώ για N=4096 δεν έχουμε σημαντική διαφοροποίηση από ότι για 6 threads.

Μία πιθανή εξήγηση είναι ότι το κόστος του χρονοπρογραμματισμού των παραπάνω threads είναι μεγαλύτερο από το όφελος της περαιτέρω παραλληλοποίησης.

Τέλος, για την βέλτιστη εκτέλεση του αλγορίθμου θα επιλέγαμε 6 threads, καθώς δεν παρατηρούμε κάποια κλιμάκωση πέρα από αυτό το σημείο.

2.4 ΖΗΤΟΥΜΕΝΑ

2.4.1

Ανακαλύψτε τον παραλληλισμό του αλγορίθμου σε κάθε έκδοση και σχεδιάστε την παραλληλοποίησή του. Περιγράψτε τη στρατηγική παραλληλοποίησης στην ενδιάμεση αναφορά.

Κλασσικός Αλγόριθμος:
Για κάθε k:

- Τα στοιχεία A[i][k], A[k][j] (που αποτελούν την μπλε γραμμή στα σχήματα) δεν αλλάζουν τιμή,διότι:
  - $A[i][k] = min(A[i][k], A[k][k] + A[i][k]) = A[i][k]$
  - $A[k][j] = min(A[k][j], A[k][k] + A[k][j]) = A[k][j]$
- Άρα, μπορούμε να παραλληλοποιήσουμε πλήρως τα δύο εσωτερικά loop για το i, j.

Στον κώδικα με τα directives του OpenMP, θα είναι έτσι:

```
for(k=0;k<N;k++)
 #pragma omp parallel for private(j)
      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]);
```

Recursive Αλγόριθμος:
Επειδή ο FW είναι memory bound και για μεγάλα N δεν χωράει ολόκληρος στην cache, υπάρχουν οι εναλλακτικές υλοποιήσεις του αλγορίθμου. Μία από αυτές είναι και ο αναδρομικός αλγόριθμος FW, ο οποίος αναλύεται παρακάτω:

Ο Αναδρομικός αλγόριθμος δέχεται ως **είσοδο**:

- 3 πίνακες (`A, B, C`) ίδιου μεγέθους.
    - Αρχικά μέγεθος = N
- Την γραμμή και την στήλη του πρώτου στοιχείου, για κάθε ένα από τους τρεις πίνακες (`arow, acol, brow, bcol, crow, ccol`).
    - Αρχικά ολα row = 0, col = 0
- Το μέγεθος του υποπίνακα προς επεξεργασία (`myN`).
    - Αρχικά μέγεθος = N
- Το μέγεθος του υποπίνακα για το οποίο θα σταματήσει η αναδρομή (`bsize`), το base case της αναδρομής δηλαδή.

Σε κάθε κλήση της αναδρομικής συνάρτησης γίνεται έλεγχος για το ποιά περίπτωση θα εκτελεστεί.

- Αν το μέγεθος του υποπίνακα που δίνεται, είναι μικρότερο ή ίσο του base case, τότε:
- **`if (myN <= bsize)`**:
    - Για κάθε χρονικό βήμα **`k`**, υπολογίζεται για κάθε ζεύγος κόμβων **`i-j`**, αν υπάρχει συντομότερο μονοπάτι από τον **`i`** προς τον **`j`**, περνώντας από τον **`k`**.
    - Όλοι αυτοί οι υπολογισμοί γίνεται σχετικά με τις θέσεις (σειρά, στήλη) των πρώτων στοιχείων των υποπινάκων που έχουν δοθεί σε αυτήν την αναδρομική κλήση.
- Αλλιώς αν το μέγεθος του υποπίνακα που δίνεται, είναι μεγαλύτερο του base case, τότε:
- **`if (myN > bsize):`**
    - Γίνονται 8 αναδρομικές κλήσεις, όπου υποδιπλασιάζεται το μέγεθος του πίνακα (**`myN / 2`**), και στις αλλάζουν οι θέσεις των πρώτων στοιχείων των τριών πινάκων σύμφωνα με την σειρά που υποδεικνύει ο αλγόριθμος.
        - Στις πρώτες 4 κλήσεις, στέλνονται οι υποπίνακες που αντιστοιχούν σε:
            - Κλήσεις A: A_{00}, A_{01}, A_{10}, A_{11} υποπίνακα
            - Κλήσεις B: B_{00}, B_{00}, B_{10}, B_{10} υποπίνακα
            - Κλήσεις C: C_{00}, C_{01}, C_{00}, C_{01} υποπίνακα
        - Στις τελευταίες 4 κλήσεις, γίνονται οι ίδιες κλήσεις με αντίστροφη σειρά.
            - Κλήσεις A: A_{11}, A_{10}, A_{01}, A_{00} υποπίνακα
            - Κλήσεις B: B_{10}, B_{10}, B_{00}, B_{00} υποπίνακα
            - Κλήσεις C: C_{01}, C_{00}, C_{01}, C_{00} υποπίνακα
            - Κλήσεις:

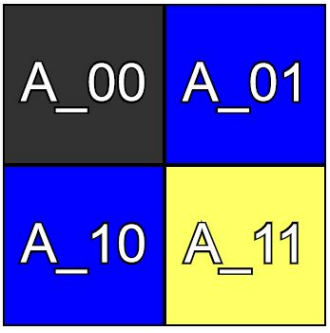
Οι **εξαρτήσεις** που υπάρχουν είναι:

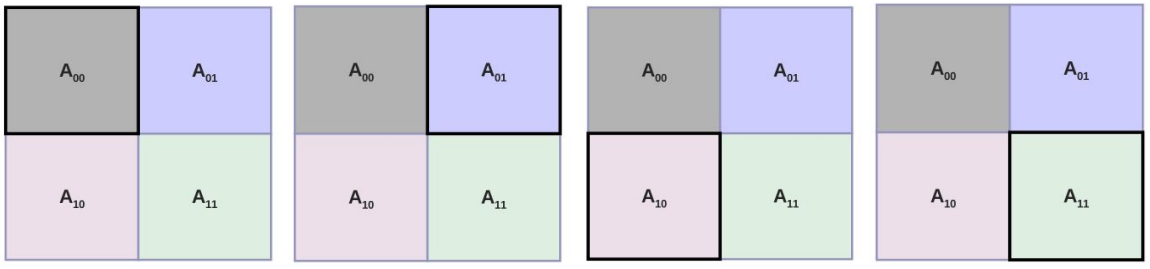
- Το μαύρο block εξαρτάται μόνο από τον εαυτό του.
    - Τα κελιά του μαύρου block, εξαρτάται μόνο από τα κελιά του μαύρου block.
- Τα block των μπλε γραμμών και στηλών είναι ανεξάρτητα μεταξύ τους (εξαρτώνται μόνο από το μαύρο block).
    - Τα κελιά του κάθε μπλε block, εξαρτώνται μόνο από τα κελιά του μαύρου block και τα κελιά του ίδιου του μπλε block.
- Τα κίτρινα tiles είναι ανεξάρτητα μεταξύ τους (εξαρτώνται μόνο από τα μπλε block του σταυρού).
    - Τα κελιά του κάθε κίτρινου block, εξαρτώνται μόνο από τα κελιά των προβολών τους στα μπλε block και του ίδιου του κίτρινου block. (π.χ. Φαίνονται αυτές οι εξαρτήσεις στο παρακάτω παράδειγμα, όπου υποθέτουμε πίνακας με 2x2 block).

Σχετικά με τις αναδρομικές κλήσεις παρατηρούμε, ότι:
Έστω ότι έχουμε τις αναδρομικές κλήσεις με τον παρακάτω πίνακα 2x2, ώστε οι μόνες κλήσεις που θα γίνουν είναι οι 8 που περιγράψαμε (για απλότητα).

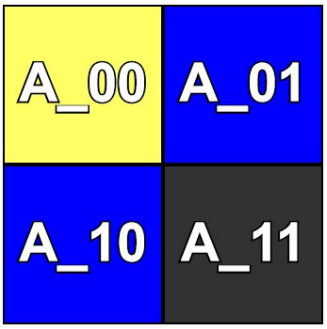
Για τις πρώτες 4 κλήσεις, θέλουμε να εκτελεστεί πρώτα:

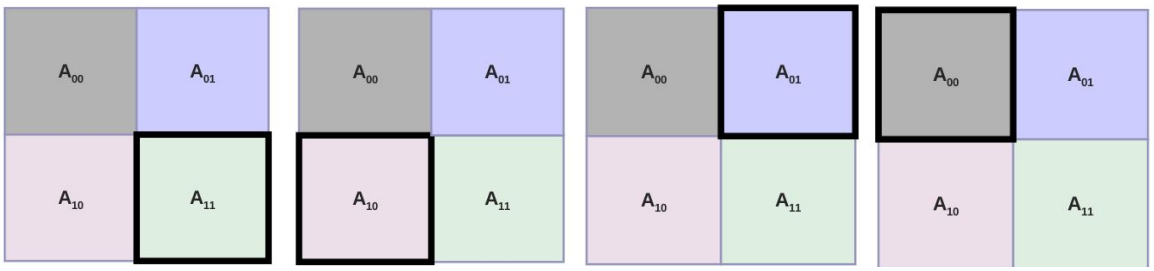
- Το A_{00} , το οποίο εξαρτάται μόνο από τον εαυτό του, για αυτό και το καλούμε με τα B_{00}, C_{00} .
- Το A_{01} , το οποίο εξαρτάται μόνο του A_{00} και τον εαυτό του, για αυτό και το καλούμε με τα B_{00}, C_{01} .
- Το A_{10} , το οποίο εξαρτάται μόνο του A_{00} και τον εαυτό του, για αυτό και το καλούμε με τα B_{10}, C_{00} .
- Το A_{11} , το οποίο εξαρτάται μόνο του A_{00} και τον εαυτό του, για αυτό και το καλούμε με τα B_{10}, C_{01} .





Αντίστοιχα, για τις 4 κλήσεις με την αντίστροφη σειρά, ο Α καλείται με τους Β και C, όπως γράψαμε πάνω ανάλογα με τις εξαρτήσεις του σταυρού.





Για την παραλληλοποίηση, παρατηρούμε ότι:

- Οι κλήσεις των A_{01} , A_{10} υποπινάκων (μπλε block) είναι ανεξάρτητες μεταξύ τους, καθώς η μόνη τους εξάρτηση αφορά τα ίδια τα block και το εκάστοτε μαύρο block (είτε το A_{00} είτε το A_{11}).
- Οπότε αυτές οι κλήσεις είναι παραλληλοποιήσιμες και τις παραλληλοποιούμε μέσω tasks.
- Χρησιμοποιούμε τα κατάλληλα directives (του OpenMP) στον παρακάτω ψευδοκώδικα, για την χρήση των tasks και των tasks wait, ώστε να εξασφαλίσουμε την σειριοποιησιμότητα του προγράμματος.

```
FWR (A, B, C)
if (base case)
    FWI (A, B, C)
else
    FWR (A_00, B_00, C_00);

    #pragma omp task
    FWR (A_01, B_00, C_01);
    #pragma omp task
    FWR (A_10, B_10, C_00);
    #pragma omp taskwait

    FWR (A_11, B_10, C_01);
    FWR (A_11, B_10, C_01);

    #pragma omp task
    FWR (A_10, B_10, C_00);
    #pragma omp task
    FWR (A_01, B_00, C_01);
    #pragma omp taskwait

    FWR (A_00, B_00, C_00);
```

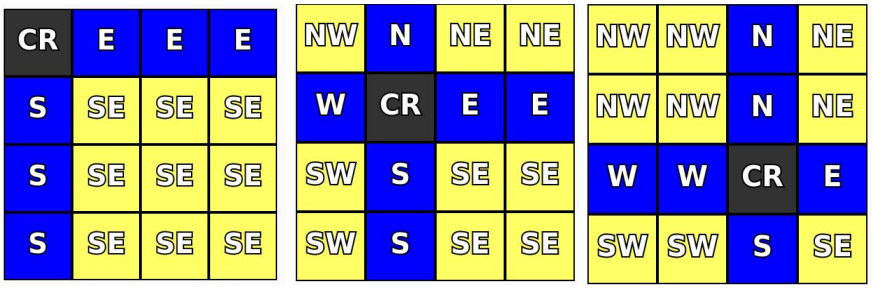
Tiled Αλγόριθμος:
Η τεχνική του tiling, στηρίζεται στην υψηλότερη επαναχρησιμοποίηση κώδικα μέσα στον κώδικα σε loops. Ο tiled αλγόριθμος του FW αναλύεται παρακάτω:

Ο κεντρικός αλγόριθμος:
Τρέχει από **0** έως **N** (μέγεθος πίνακα), και με βήμα **B** (μέγεθος tile, $N = p \cdot B,\ p \in \mathbb{N}$, διαιρεί τον πίνακα σε tiles μεγέθους **B**).

- Κατά την **k**-οστή block επανάληψη, ο αλγόριθμος:
    - Καλεί την συνάρτηση FW, για ανανέωση της τιμής του **k**-οστού διαγώνιου tile (CR - black tile).
    - Καλεί την συνάρτηση FW, για ανανέωση της τιμής των υπόλοιπων tiles που ανήκουν στην στήλη του **k**-οστού block (N, S - blue tiles).
    - Καλεί την συνάρτηση FW, για ανανέωση της τιμής των υπόλοιπων tiles που ανήκουν στην γραμμή του **k**-οστού block (E, W - blue tiles).
    - Καλεί την συνάρτηση FW, για ανανέωση της τιμής των υπόλοιπων tiles του πίνακα (NE, NW, SE, SW - yellow tiles).
        - Πρώτα των NW tiles.
        - Μετά των NE tiles.
        - Μετά των SW tiles.
        - Τέλος των SE tiles.

Η συνάρτηση `FW()` δέχεται ως **είσοδο**:

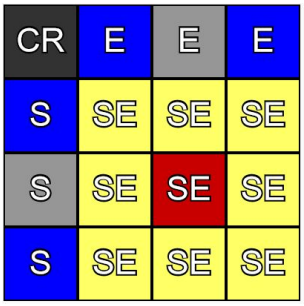
- Τον δισδιάστατο Πίνακα (`A`).
    - Καλείται πάντα με τον πίνακα εισόδου `A`.
- Η τιμή `k` για το εξωτερικότερο loop (`K`).
    - Καλείται κάθε φορά με την δεδομένη τιμή του `k`.
- Η τιμή `i` για το εσωτερικό loop (`I`).
    - Καλείται κάθε φορά με τις δεδομένες τιμές του `i` (για N tiles) ή `i+B` (για S tiles).
- Η τιμή `j` για το εσωτερικότερο loop (`J`).
    - Καλείται κάθε φορά με τις δεδομένες τιμές του `j` (για W tiles) ή `j+B` (για E tiles).
- Το μέγεθος του υποπίνακα προς επεξεργασία (`N`).
    - Καλείται πάντα με το μέγεθος του tile **B**.



Παράδειγμα εκτέλεσης αλγορίθμου για πίνακα με 4x4 tiles.

Παρατηρούμε, ότι:

- Το μαύρο tile εξαρτάται μόνο από τον εαυτό του.
    - Τα κελιά του μαύρου tile, εξαρτάται μόνο από τα κελιά του μαύρου tile.
- Τα tiles των μπλε γραμμών και στηλών είναι ανεξάρτητα μεταξύ τους (εξαρτώνται μόνο από το μαύρο tile).
    - Τα κελιά του κάθε μπλε tile, εξαρτώνται μόνο από τα κελιά του μαύρου tile και τα κελιά του ίδιου του μπλε tile.
- Τα κίτρινα tiles είναι ανεξάρτητα μεταξύ τους (εξαρτώνται μόνο από τα μπλε tiles του σταυρού).
    - Τα κελιά του κάθε κίτρινου tile, εξαρτώνται μόνο από τα κελιά των προβολών τους στα μπλε tiles και του ίδιου του κίτρινου tile. (π.χ. Φαίνονται αυτές οι εξαρτήσεις στο παρακάτω παράδειγμα, όπου υποθέτουμε πίνακας με 4x4 tiles).



Η παραλληλοποίηση του αλγορίθμου θα γίνει με tasks, και τα παράλληλα κομμάτια είναι:

- Ανανέωση της τιμής του CR, **k**-οστού διαγώνιου tile.

- (`#pragma omp task`)
    - Ανανέωση της τιμής των N tiles που ανήκουν στην στήλη του **k**-οστού block.
- (`#pragma omp task`)
    - Ανανέωση της τιμής των S tiles που ανήκουν στη στήλη του **k**-οστού block.
- (`#pragma omp task`)
    - Ανανέωση της τιμής των E tiles που ανήκουν στην γραμμή του **k**-οστού block.
- (`#pragma omp task`)
    - Ανανέωση της τιμής των W tiles που ανήκουν στη γραμμή του **k**-οστού block.

- (`#pragma omp taskwait`)

- (`#pragma omp task`)
    - Ανανέωση της τιμής των NE tiles του πίνακα.
- (`#pragma omp task`)
    - Ανανέωση της τιμής των NW tiles του πίνακα.
- (`#pragma omp task`)
    - Ανανέωση της τιμής των SW tiles του πίνακα.
- (`#pragma omp task`)
    - Ανανέωση της τιμής των SE tiles του πίνακα.

- (`#pragma omp taskwait`)

2.4.2

Υλοποιήστε παράλληλες εκδόσεις του αλγορίθμου Floyd Warshall (τουλάχιστον μία για κάθε σειριακή έκδοση) χρησιμοποιώντας το OpenMP ή τα Threading Building Blocks και πραγματοποιήστε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για 1, 2, 4, 8,16, 32 και 64 threads στο μηχάνημα sandman.

Κλασσικός Αλγόριθμος:
Το κομμάτι του κώδικα που έχει το μεγαλύτερο φορτίο φαίνεται παρακάτω πως το παραλληλοποιήσαμε:

```
for(k=0;k<N;k++)
    #pragma omp parallel for private(j)
      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]);
```

Στην ουσία είναι ο ίδιος που είχαμε και στην ενδιάμεση αναφορά για το ερώτημα 2.4.1.

Recursive Αλγόριθμος:
Το κομμάτι του κώδικα που έχει το μεγαλύτερο φορτίο φαίνεται παρακάτω πως το παραλληλοποιήσαμε:

```
#pragma omp parallel
{
  #pragma omp single
  {
    FW_SR(A, arow, acol, B, brow, bcol, C, crow, ccol, myN/2, bsize);

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

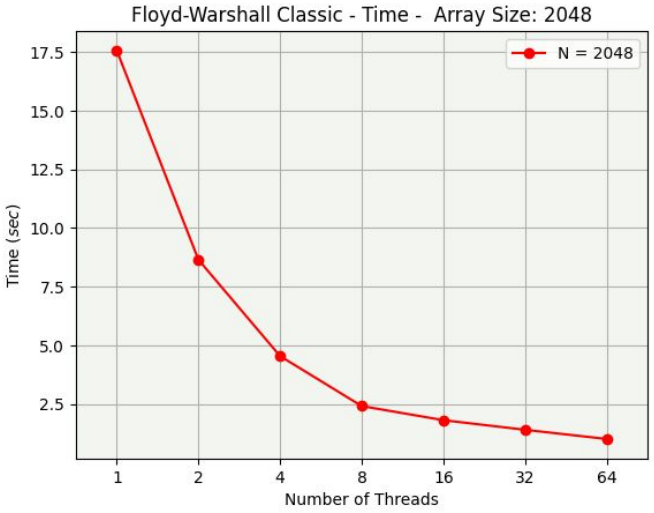
Αυτό είναι το κομμάτι κώδικα που συμβαίνει όταν δεν πέφτουμε στο base case της αναδρομής.

Tiled Αλγόριθμος:
Το κομμάτι του κώδικα που έχει το μεγαλύτερο φορτίο φαίνεται παρακάτω πως το παραλληλοποιήσαμε:

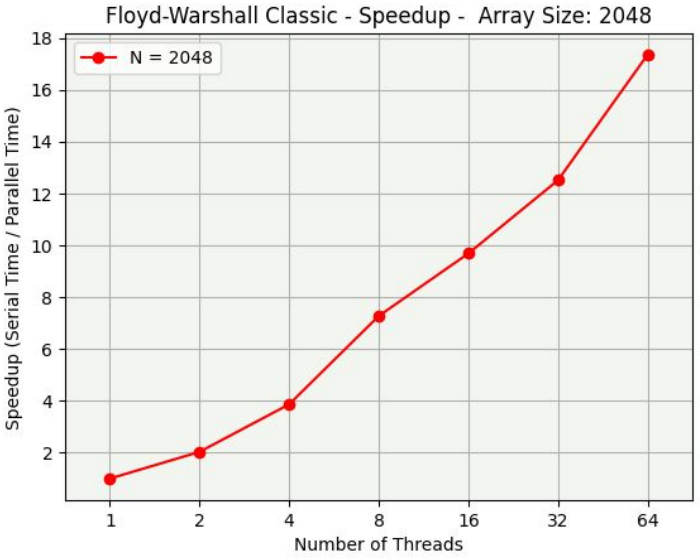
```
for(k=0;k<N;k+=B) {
#pragma omp parallel
	{
		#pragma omp single
		{
			FW(A,k,k,k,B);	// CR tile

			for(i=0; i<k; i+=B)   // N tiles
			    #pragma omp task firstprivate(i)
			    FW(A,k,i,k,B);

			for(i=k+B; i<N; i+=B) // S tiles
			    #pragma omp task firstprivate(i)
			     FW(A,k,i,k,B);

			for(j=0; j<k; j+=B)   // W tiles
			    #pragma omp task firstprivate(j)
			    FW(A,k,k,j,B);

			for(j=k+B; j<N; j+=B) // E tiles
			    #pragma omp task firstprivate(j)
			    FW(A,k,k,j,B);

			 #pragma omp taskwait

			for(i=0; i<k; i+=B)   // NW tiles
			    for(j=0; j<k; j+=B)
			        #pragma omp task firstprivate(i, j)
			        FW(A,k,i,j,B);

			for(i=0; i<k; i+=B)   // NE tiles
			    for(j=k+B; j<N; j+=B)
			        #pragma omp task firstprivate(i, j)
			        FW(A,k,i,j,B);

			for(i=k+B; i<N; i+=B) // SW tiles
			    for(j=0; j<k; j+=B)
			        #pragma omp task firstprivate(i, j)
			        FW(A,k,i,j,B);

			for(i=k+B; i<N; i+=B) // SE tiles
			     for(j=k+B; j<N; j+=B)
			        #pragma omp task firstprivate(i, j)
			        FW(A,k,i,j,B);
		}
	}
}
```

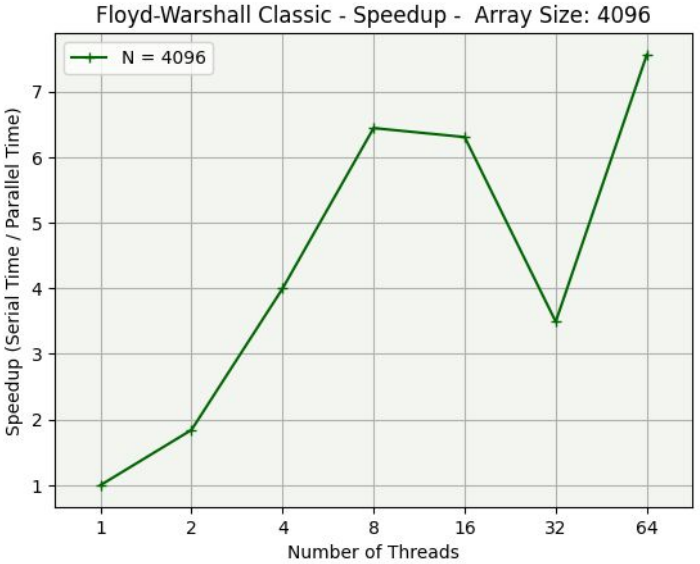
Χρησιμοποιούμε `firstprivate (i), firstprivate (j), firstprivate (i, j)`, γιατί οι μεταβλητές `i, j` χρησιμοποιούνται ταυτόχρονα από άλλες παράλληλες περιοχές οπότε πρέπει να δηλωθούν ως ιδιωτικές αλλιώς υπάρχει περίπτωση να πάρουν λάθος τιμές.

Διαγράμματα Χρόνου Κανονικού FW

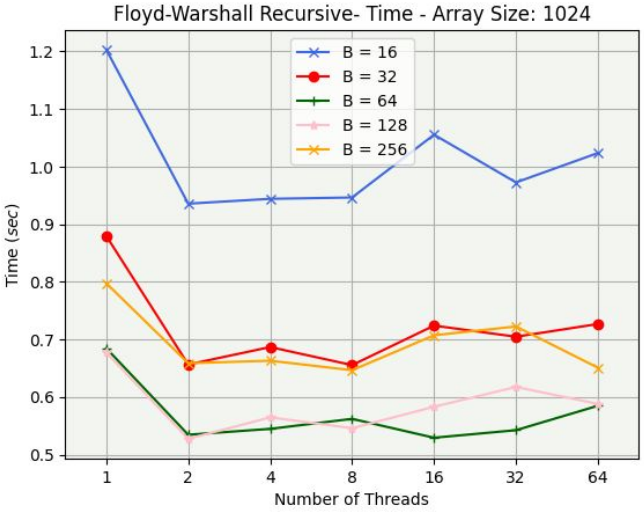


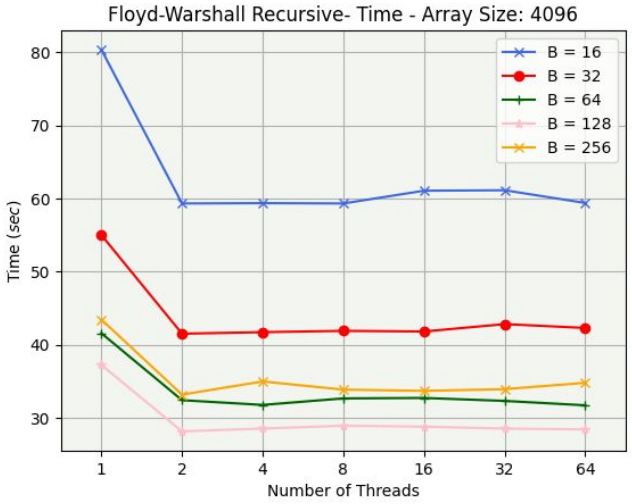
Διαγράμματα Speedup Κανονικού FW



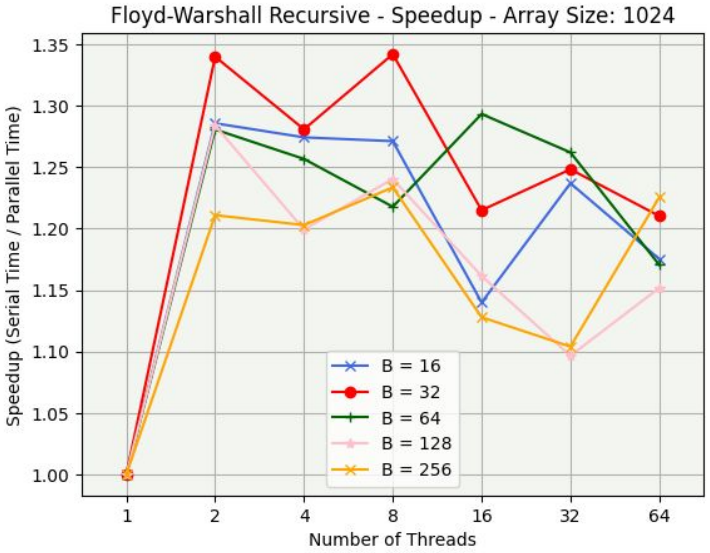


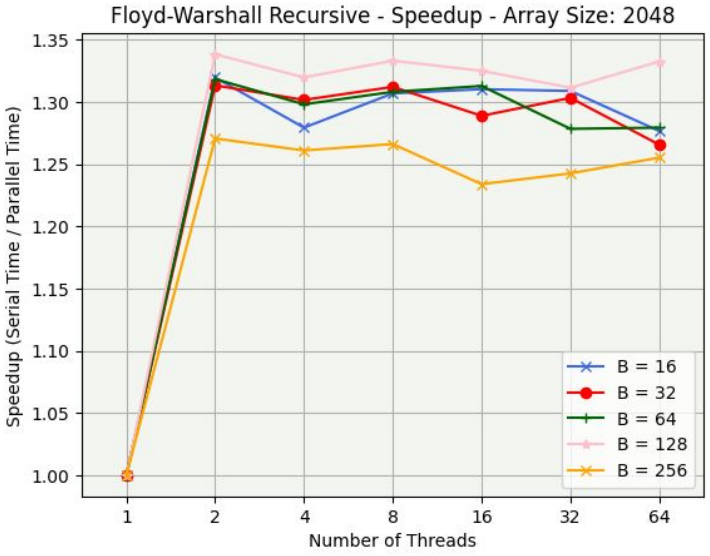
Διαγράμματα Χρόνου Recursive FW

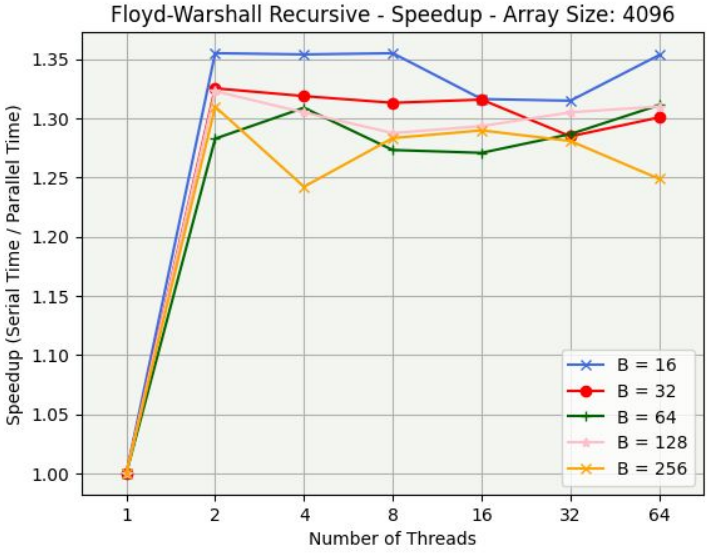




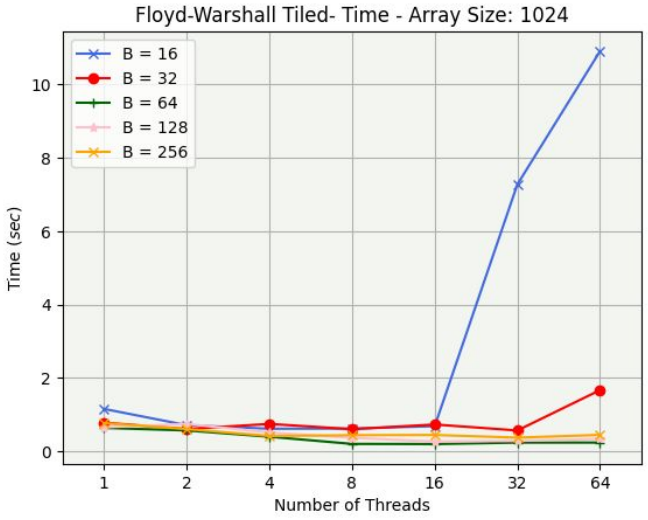
Διαγράμματα Speedup Recursive FW

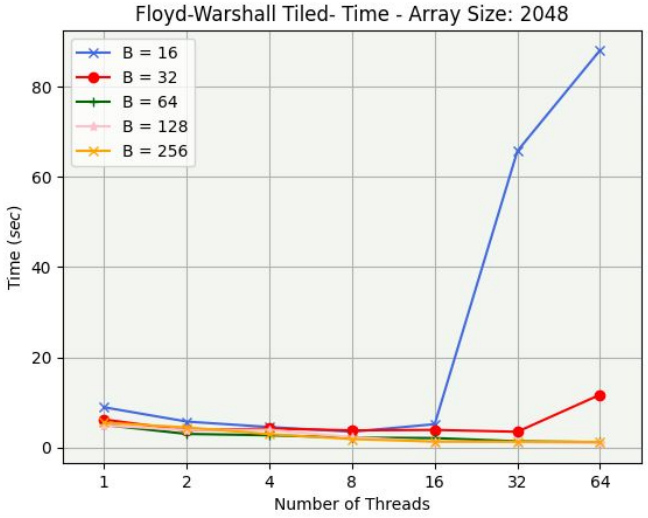


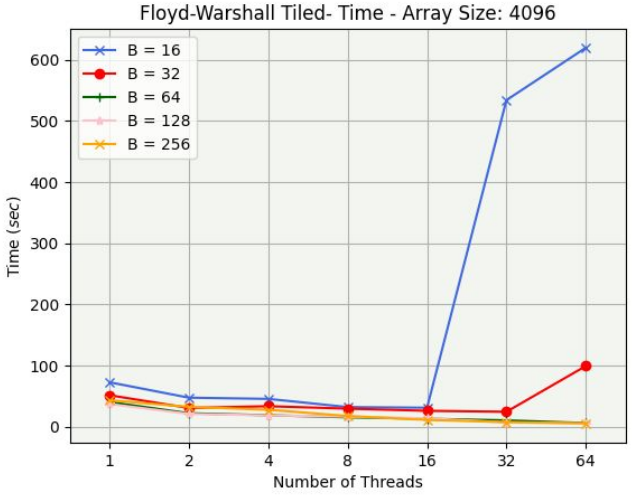




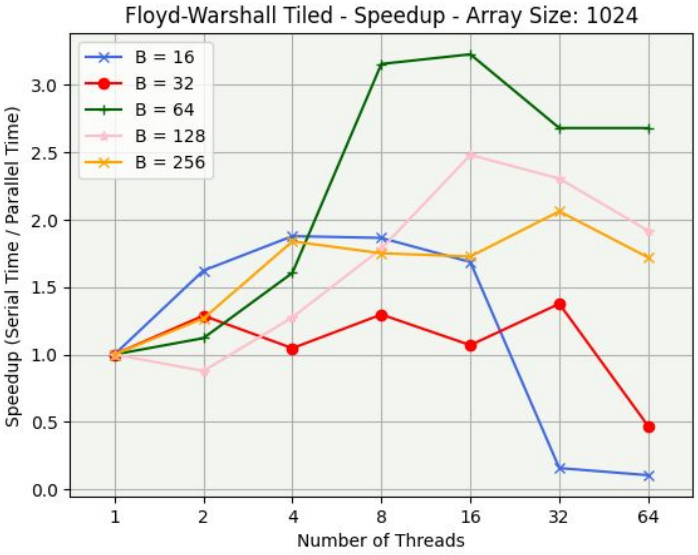
Διαγράμματα Χρόνου Tiled FW

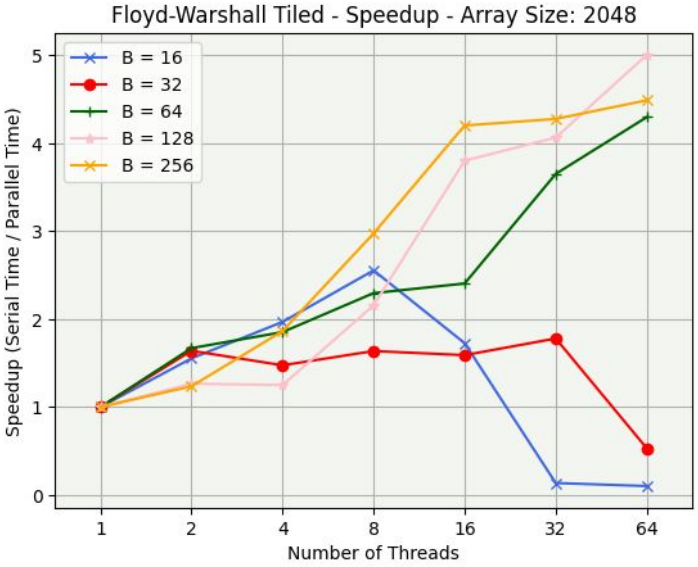






Διαγράμματα Speedup Tiled FW





Συμπεράσματα

Για την παραλληλοποίηση του **κανονικού αλγόριθμου**, παρατηρούμε ότι έχουμε πολύ καλή κλιμακωσιμότητα μέχρι τα 8 threads (σχεδόν υποδιπλασιασμός του χρόνου), για κάθε μέγεθος πίνακα. Όμως μετά τα 8 threads, η επίδοση του χρόνου μένει σχετικά στάσιμη, και δεν βελτιώνεται με τον διπλασιασμό των threads. Αυτό φανερώνει πρόβλημα κλιμάκωσης, για αυτό και έχουν αναπτυχθεί οι καλύτερες εκδόσεις του αλγορίθμου.

Παρατηρούμε ότι για την παραλληλοποίηση του **αναδρομικού αλγορίθμου**, δεν έχουμε κλιμάκωση όσον αφορά την αύξηση των threads, πέρα από την αρχική αύξηση $1 \rightarrow 2$. Όμως παρατηρήσαμε ότι για διαφορετικές τιμές του blok_size πετυχαίνουμε καλύτερη αριθμητικά απόδοση. Βλέπουμε ότι για υψηλά B (B = 64, B = 128), έχουμε τους καλύτερους χρόνους, για όλα τα μεγέθη πίνακα.

Δεν γνωρίζουμε για ποιο λόγο εμφανίζει ο αλγόριθμος αυτή την συμπεριφορά αλλα γνωρίζουμε ότι σε κάθε επανάληψη δημιουργούνται δύο νέα παράλληλα tasks (διαγώνια μπλοκ). Οπότε ο αλγόριθμος αναδρομικα αυξάνει τον αριθμό των παράλληλων tasks και θεωρητικά θα έπρεπε να επωφελείται από την ύπαρξη περισσότερων threads. Το παραπάνω όμως δεν το παρατήρησαμε στις μετρήσεις μας.

Για την παραλληλοποίηση του **tiled αλγόριθμου**, αρχικά βλέπουμε ότι για B = 16 έχουμε την χειρότερη επίδοση (και ακόμα χειρότερη για B = 8, αλλά δεν το συμπεριλάβαμε καν στην αναφορά), το οποίο κρύβει και την κλιμακωσιμότητα για τα άλλα μεγέθη B. Στα διαγράμματα speedup, βλέπουμε καλύτερα ότι σε σχέση με τον recursive αλγόριθμο οι χρόνοι μειώνονται όσο αυξάνεται ο αριθμός των threads. Καλύτερη κλιμάκωση προσφέρουν οι μεγαλύτερες τιμές του B (όπως B = 64, 128, 256 αντίστοιχα), αλλά αριθμητικά μικρότερους χρόνους πετυχαίνουμε για μικρότερου μεγέθους Block Size (όπως B = 32).

Συγκριτικά, σε σχέση με την κλιμάκωση αλλά και τον χρόνο ο καλύτερος αλγόριθμος είναι ο tiled, όπως φάνηκε από τις μετρήσεις.

2.4.3

Αξιοποιήστε τις παρατηρήσεις σας και τις μετρήσεις σας από τις πρώτες υλοποιήσεις, την κατανόηση της αρχιτεκτονικής του μηχανήματος sandman, τις δομές παράλληλου προγραμματισμού που προσφέρει κάθε εργαλείο παραλληλοποίησης και τις δυνατότητες του μεταγλωττιστή για να βελτιώσετε την παράλληλη επίδοση του αλγορίθμου Floyd Warshall.

Επιλέξαμε να βελτιώσουμε τον κώδικα του tiled αλγόριθμου, καθώς ο recursive δεν κλιμάκωνε ιδιαίτερα. Δοκιμάσαμε να αλλάξουμε την υλοποίηση με τα tasks και να δοκιμάσουμε με τα parallel for, κάτι το οποίο βελτίωση αρκετά την επίδοση μας.

Έπειτα δοκιμάσαμε να βάλουμε `nowait`, διότι κάποια threads θα περίμεναν να τελειώσει το loop για να συνεχίσουν στα υπόλοιπα. Αυτό βελτίωσε κάπως την επίδοση, αλλά όχι δραματικά, όπως και η δυναμική χρονοδρομολόγηση των threads αντί για στατική.

Κάτι που βελτίωσε αρκετά την επίδοση ήταν η χρήση του `collapse` construct, το οποίο χρησιμοποιήσαμε στα διπλά loop. Στην ουσία, αυτό το construct, δηλώνει ότι θα γίνει μια συγχώνευση του εσωτερικού loop με το εξωτερικό, εφόσων τα όρια των loop είναι ανεξάρτητα, ώστε όταν οι επαναλήψεις του εξωτερικού loop είναι μικρότερες από τα διαθέσιμα thread, να μην περιμένουν, αλλά να συνεχίζουν στις επόμενες επαναλήψεις.

Επίσης μια άλλη αλλαγή που κάναμε στην προσπάθεια να βελτιώσουμε το locality των αναφορών, ήταν να αλλάξουμε την σειρά που επεξεργαζόμαστε τα tiles, ώστε κάποια από τα δεδομένα να βρίσκονται ήδη στην cache για τα επόμενα.
Π.χ. Βάλαμε πρώτα να υπολογίζονται τα North και West tiles, και αμέσως μετά τα NW tiles, ώστε να αυξήσουμε την πιθανότητα να πάρουν τα δεδομένα κατευθείαν από την cache και να μειώσουμε τα misses.
Φροντίσαμε να μην επηρεάσουμε την ορθότητα του προγράμματος, προσέχοντας την σειρά που θα βάλουμε την επεξεργασία των tiles, όπως φαίνεται παρακάτω.

Tiled Optimal Αλγόριθμος:
Το κομμάτι του κώδικα που έχει το μεγαλύτερο φορτίο φαίνεται παρακάτω πως το παραλληλοποιήσαμε:

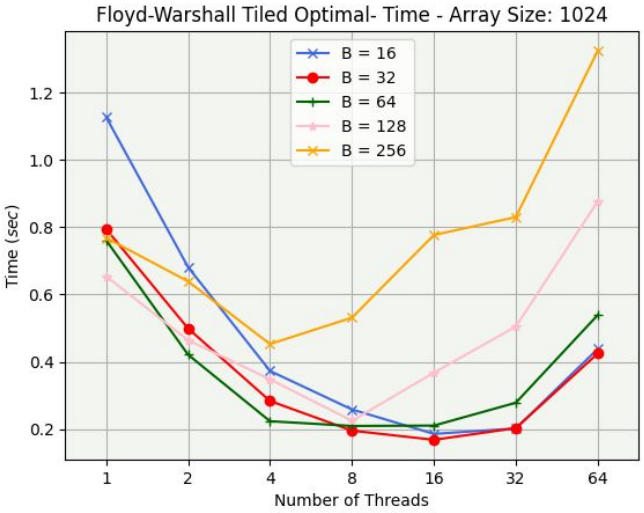
```
for(k=0;k<N;k+=B)
{
    #pragma omp parallel
    {
        FW(A,k,k,k,B);        // CR tile

        #pragma omp for nowait schedule(dynamic)
        for(i=0; i<k; i+=B)   // N tiles
            FW(A,k,i,k,B);

        #pragma omp for nowait schedule(dynamic)
        for(j=0; j<k; j+=B)   // W tiles
            FW(A,k,k,j,B);
```

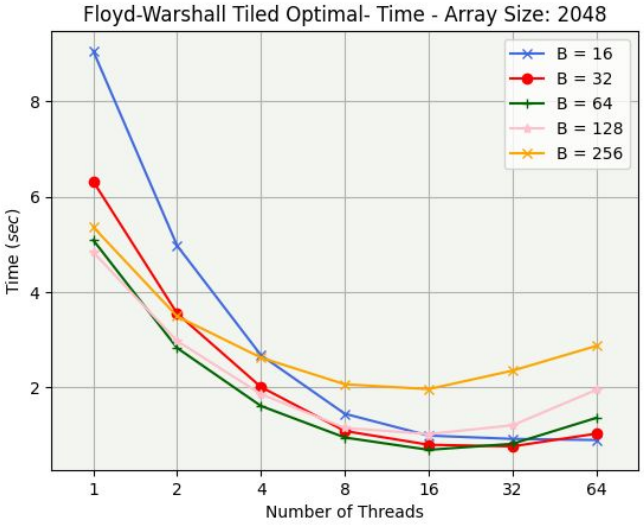
```
        #pragma omp for private(j) nowait collapse(2) schedule(dynamic)
        for(i=0; i<k; i+=B)   // NW tiles
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for nowait schedule(dynamic)
        for(j=k+B; j<N; j+=B) // E tiles
            FW(A,k,k,j,B);

        #pragma omp for private(j) nowait collapse(2) schedule(dynamic)
        for(i=0; i<k; i+=B)   // NE tiles
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for nowait schedule(dynamic)
        for(i=k+B; i<N; i+=B) // S tiles
            FW(A,k,i,k,B);

        #pragma omp for private(j) nowait collapse(2) schedule(dynamic)
        for(i=k+B; i<N; i+=B) // SW tiles
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for private(j) nowait collapse(2) schedule(dynamic)
        for(i=k+B; i<N; i+=B) // SE tiles
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);
    }
}
```

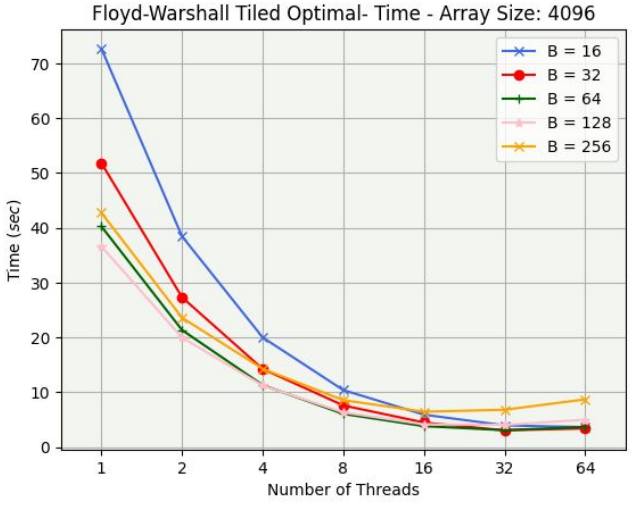
2.4.4

Πραγματοποιήστε μετρήσεις για τις καλύτερες παράλληλες εκδόσεις, για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096, για 1, 2, 4, 8, 16, 32 και 64 threads στο μηχάνημα sandman.

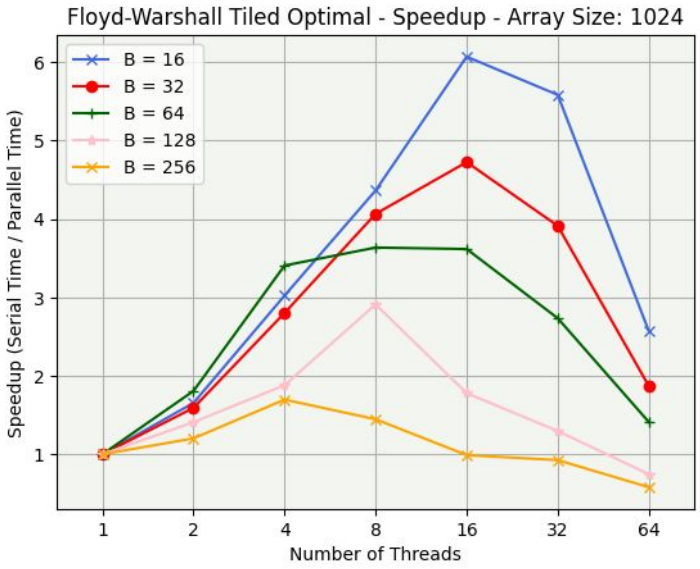
Διαγράμματα Χρόνου Tiled Optimal

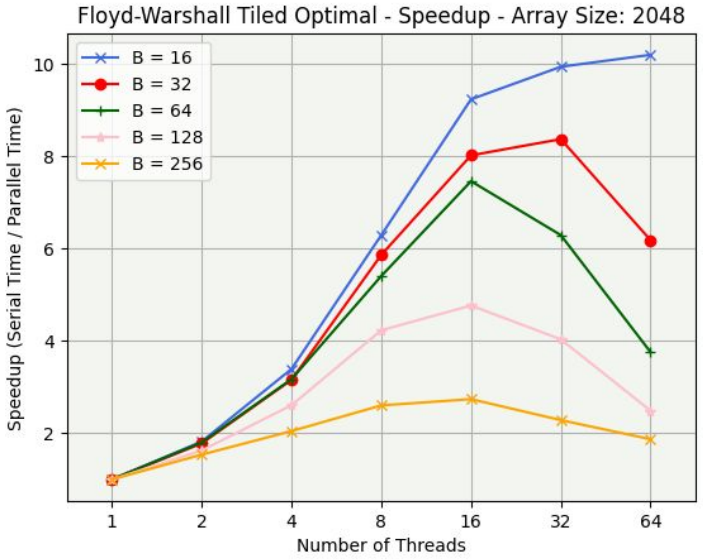






Διαγράμματα Speedup Tiled Optimal





Συμπεράσματα

Παρατηρούμε, ότι για μικρά μεγέθη πίνακα, τα μεγάλα B δεν κλιμακώνουν καθόλου καλά (B = 128, B = 256). Όσο όμως μεγαλώνει το μέγεθος του πίνακα, οι διαφορές μεταξύ του B γίνονται όλο και πιο δυσδιάκριτες.
Σχετικά με την κλιμακωσιμότητα του προγράμματος, παρατηρούμε είναι σχετικά καλή μέχρι τα 16 threads. Για παραπάνω threads ή δεν υπάρχει κάποια αισθητή βελτίωση για μεγάλα μεγέθη πίνακα, ή γίνεται και χειρότερη για τα μικρότερα μεγέθη.

Παρατηρώντας τα speedup διαγράμματα, βλέπουμε ότι για B = 16, έχουμε την καλύτερη κλιμάκωση, και όσο αυξάνεται το B, χειροτερεύει.

Όμως τους μικρότερους αθροιστικά χρόνους, για το μεγάλο μέγεθος πίνακα, τους πήραμε για B = 32, ή B = 16.

Εν τέλει ύστερα από τις βελτιστοποιήσεις του tiled, οι χρόνοι μας μειώθηκαν αρκετά αλλά σε καμία περίπτωση τάξη μεγέθους.

2.4.5

Περιγράψτε τις καλύτερες παράλληλες εκδόσεις σας και παρουσιάστε τις μετρήσεις σας στην τελική αναφορά. Αναφέρετε τον καλύτερο χρόνο που επιτύχατε!

Το καλύτερο χρόνο τον πετύχαμε στον Optimal Tiled αλγόριθμο, για B=32 και με χρόνο t=2.9631 seconds.