Συστήματα Παράλληλης Επεξεργασίας

Άσκηση 1 - Παραλληλοποίηση αλγορίθμων σε πολυπύρηνες αρχιτεκτονικές κοινής μνήμης

Χειμερινό Εξάμηνο 2020-2021

Ενδιάμεση Αναφορά

Ηλιάδης Θρασύβουλος - 03115761
Πέππας Αθανάσιος - 03115749

1. Conway's Game of Life

1.1 Σκοπός της ενότητας

Σκοπός της συγκεκριμένης ενότητας της άσκησης είναι η εξοικείωση με τις υποδομές του εργαστηρίου (πρόσβαση στα συστήματα, μεταγλώττιση προγραμμάτων, υποβολή εργασιών κλπ) μέσα από την παραλληλοποίηση ενός απλού προβλήματος σε αρχιτεκτονικές κοινής μνήμης.

1.2 Το Παιχνίδι της Ζωής

Το *Παιχνίδι της Ζωής (Conway's Game of Life)* λαμβάνει χώρα σε ένα ταμπλό με κελιά δύο διαστάσεων. Το περιεχόμενο κάθε κελιού μπορεί να είναι γεμάτο (alive) ή κενό (dead), αντικατοπτρίζοντας την ύπαρξη ή όχι ζωντανού οργανισμού σε αυτό, και μπορεί να μεταβεί από τη μία κατάσταση στην άλλη μία φορά εντός συγκεκριμένου χρονικού διαστήματος. Σε κάθε βήμα (χρονικό διάστημα), κάθε κελί εξετάζει την κατάστασή του και αυτή των γειτόνων του (δεξιά, αριστερά, πάνω, κάτω και διαγώνια) και ακολουθεί τους παρακάτω κανόνες για να ενημερώσει την κατάστασή του:

• Αν ένα κελί είναι ζωντανό και έχει λιγότερους από 2 γείτονες πεθαίνει από μοναξιά.
• Αν ένα κελί είναι ζωντανό και έχει περισσότερους από 3 γείτονες πεθαίνει λόγω υπερπληθυσμού.
• Αν ένα κελί είναι ζωντανό και έχει 2 ή 3 γείτονες επιβιώνει μέχρι την επόμενη γενιά.
• Αν ένα κελί είναι νεκρό και έχει ακριβώς 3 γείτονες γίνεται ζωντανό (λόγω αναπαραγωγής).

1.3 Υλοποίηση

Μας δίνεται η σειριακή έκδοση του αλγορίθμου.

```
for ( t = 0 ; t < T ; t++ ) {
        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;
            }
        #ifdef OUTPUT
        print_to_pgm(current, N, t+1);
        #endif
        //Swap current array with previous array
        swap=current;
        current=previous;
        previous=swap;
}
```

Παρατειρούμε πως δε μπορούμε να κάνουμε καποια παραλληλοποίηση στο πρώτο for καθώς καθε επανάληψη χρειάζεται δεδομένα από την προηγούμενη. Από την άλλη όταν βρισκόμαστε σε μια δεδομένη χρονική στιγμή ο υπολογισμός για κάθε κελή απαιτεί τα δεδομένα από τα κελιά των γειτώνων της ίδιας χρονικής στιγμής, συνεπώς μπορούμε να παραλληλοποιήσουμε ανα γραμμή και να δώσουμε σε διαφορετικά threads τον υπολογισμό διαφορετικών γραμμών. Η λογική αυτή είναι μια data centric λογική καθώς μοιράζουμε τα δεδομένα μεταξύ των threads.

Η παραλληλοποίηση έγινε με το OpenMP και συγκεκριμένα με την παρακάτων εντολή

```
#pragma omp parallel for shared(N, previous, current) private(i, j, nbrs)
```

Η εντολή αυτή πρέπει να γραφεί ακριβώς πάνω από το δεύτερο for ώστε να παραλληλοποιήσουμε τις γραμμές του ταμπλο.

Τέλος με το παρακάτω bash script εκτελούμε το πρόγραμμα για 1,2,4,6,8 πυρήνες

```
#!/bin/bash

##Job name
#PBS -N run_omp_gol

## Out, error files
#PBS -o rgol.out
#PBS -e rgol.err

##Number of machines
#PBS -l nodes=1:ppn=8

##Run time
#PBS -l walltime=00:30:00

module load openmp
cd /home/parallel/parlab16/a1/conway/parallel

for thr in 1 2 4 6 8
do
  export OMP_NUM_THREADS=$thr
  ##export OMP_DYNAMIC=TRUE
  ./omp_gol 64 1000
  ./omp_gol 1024 1000
  ./omp_gol 4096 1000
done
```

1.4 Αποτελέσματα

Όλες οι μετρήσεις έγιναν για 1000 γενιές και για 1,2,4,6,8 πυρήνες και μεγέθη τετραγωνικού ταμπλό 64, 1024, 4096. Παρακάτω φαίνονται συνοπτικά οι χρόνοι:

Cores	64x64	1024x1024	4096x4096
1	0,023133	10,962866	175,865034
2	0.013579	5.461324	88.379539
4	0.009120	2.740984	44.598599
6	0.008836	1.830631	34.987049
8	0.451345	6.114917	41.152464
Σειρακό	0,020351	10,790288	173,263631

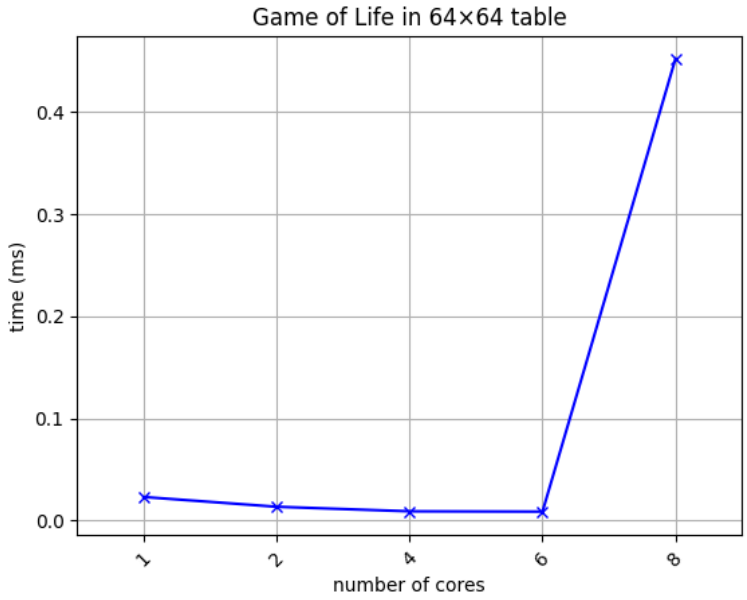
Με βάση τα αποτελέσματα αυτά κατασκευάσαμε από δύο γραφήματα για κάθε περίπτωση ταμπλό. Ένα γράφημα για τον χρόνο εκτέλεσης και ένα γράφημα για το *Speedup* το οποίο δίνεται από τον παρακάτω τύπο

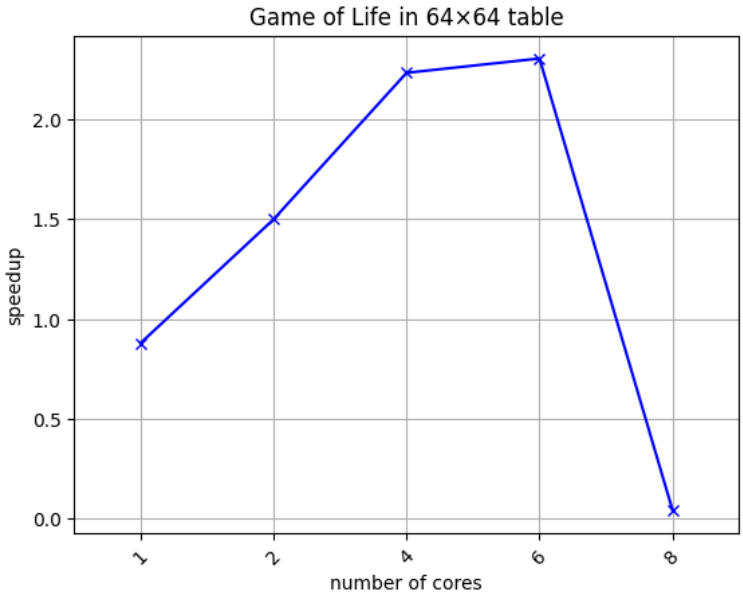
$$S = \frac{T_s}{T_p}$$

όπου T_s ο χρόνος του σειριακού προγράμματος και T_p ο χρόνος του αντίστοιχου παράλληλου.

Αυτό που μας δίνει το *S* είναι το πόσες φορές πιο γρήγορο είναι το παράλληλο πρόγραμμα από το σειριακό. Ουσιαστικά εμείς θέλουμε να μειώνεται ο χρόνος εκτέλεσης και αυτό να γίνεται ανάλογα με την αύξηση των πυρήνων που χρησιμοποιούμε, δηλαδή το S να αυξάνεται όσο αυξάνονται και οι πυρήνες. Όμως όπως θα δούμε και παρακάτω, όταν έχουμε ενα τόσο απλό πρόβλημα που ακόμα και το σειριακό είναι αρκετά γρήγορο, η παραλληλοποίηση μπορεί να μας δημιουργήσει πρόβλημα.

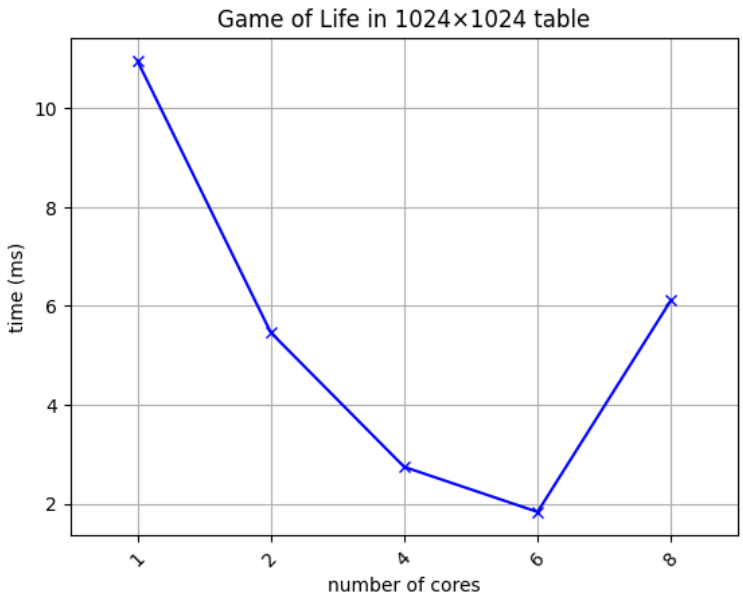
- $64x64$

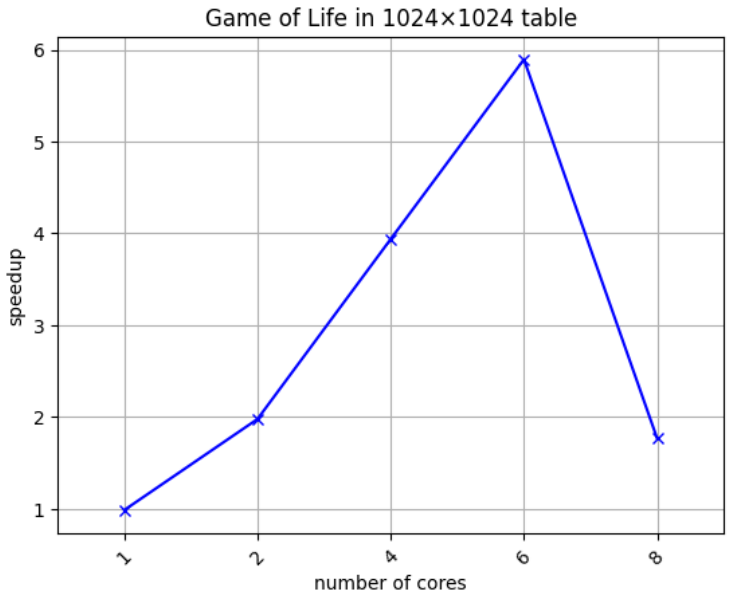




Βλέπουμε πως ο χρόνος εκτέλεσης όσο οι πυρήνες αυξάνονται μειώνεται. Αυτό όμως μεχρι τους 6, διότι όταν φτάνουμε στους 8 βλέπουμε πως ο χρόνος εκτέλεσης όχι απλώς δεν εχει μειωθεί ανάλογα με την αύξηση των πυρήνων αλλά είναι μικρότερος του σειρακού. Αυτό οφείλεται από τη μία στο γεγονός πως έχουμε αρκετά μικρό ταμπλό ώστε η παραλληλοποίηση να είναι χρήσιμη και από την άλλη, ο χρόνος που απαιτείται από το σύστημα για τη δημιουργία και των συγχρονισμό των threads είναι μεγάλος ώστε να μας δίνει εν τέλει πιο αργό πρόγραμμα.

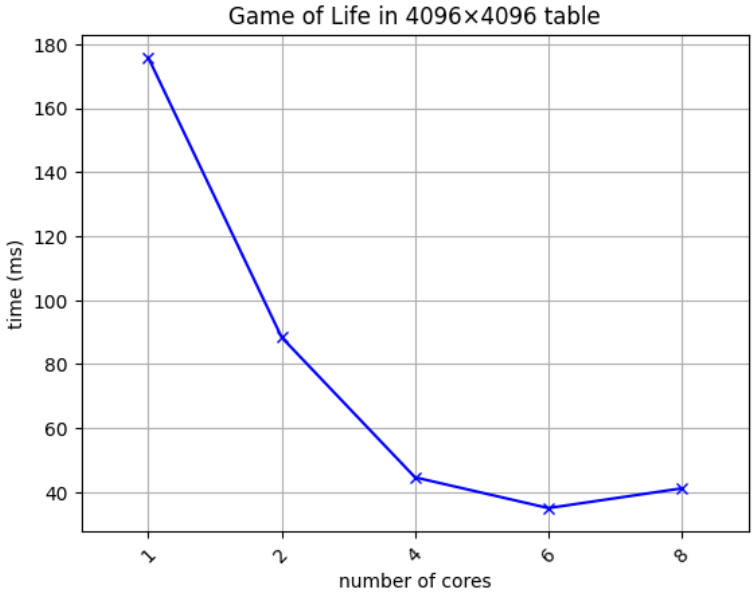
- $1024x1024$

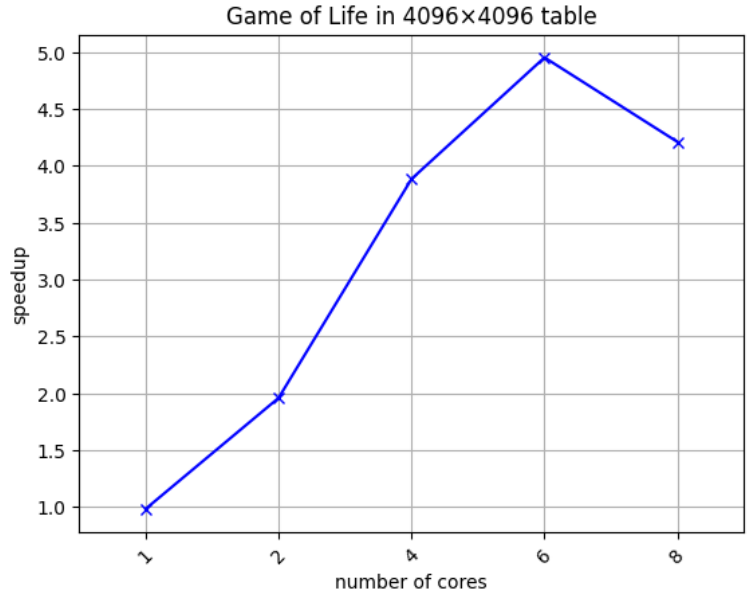




Εδώ βλέπουμε πάλι πως έχουμε το ίδιοφαινόμενο όπως και στην προηγούμενη περίπτωση όμως με τη διαφορά πως μέχρι τους 6 πυρήνες έχουμε πολύ καλή παραλληλοποίηση καθώς όσο αυξάνονται οι πυρήνες αυξάνεται και το S. Όμως και πάλι στους 8 πυρήνες έχουμε αύξηση του χρόνου εκτέλεσης αυτή τη φορά όμως όχι περισσότερο από το σειρακό.

- $4096x4096$





Τέλος για το μέγεθος ταμπλό 4096 έχουμε μια καλύτερη εικόνα παραλληλοποίησης καθως το S αυξάνεται όσο αυξάνονται και οι πυρήνες και στην περίπτωση των 8 ναι μεν έχουμε μείωση αλλά σε καμιά περίπτωση σαν τις 2 προηγούμενες περιπτώσεις. Αυτό οφείλεται στο γεγονός πως ο χρόνος δημιουργίας και συγχρονισμού των threads δεν είναι πλέον συγκρίσιμος με την εκτέλεση του κρίσιμου κομματιου του αλγορίθμου λόγω του μεγέθους του ταμπλό.

2. Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου FloydWarshall σε αρχιτεκτονικές κοινής μνήμης

2.1 Σκοπός της ενότητας

Η ενότητα αυτή αποτελεί το βασικό κορμό της άσκησης. Στόχος της ενότητας είναι να αναπτύξετε διαφορετικές παράλληλες εκδόσεις του αλγορίθμου FloydWarshall, να αξιολογήσετε την παραγωγικότητα (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).

Στα πλαίσια της αναφοράς αυτής ασχολούμαστε με 3 σειριακές εκδοχές του αλγορίθμου:

- Standard
- Recursive και
- Tiled

Για κάθε μία από τις παραπάνω θα προτείνουμε έναν τρόπο σκέψης που οδηγεί στην παράλληλη εκτέλεση.

Σε όλες τις εκδοχές λόγω της φύσης του αλγορίθμου χρειάζεται να υπολογιστεί ο πίνακας A_k *πριν τον υπολογισμό του πίνακα* A_{k-1}

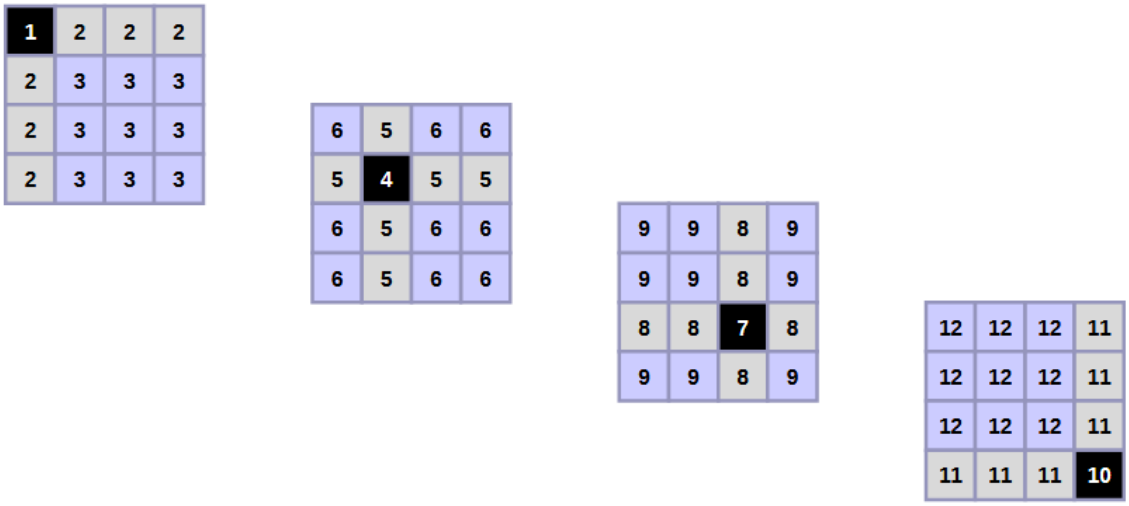
Standard έκδοση

Με το γράφο γειτνίασης στον πίνακα A μπορούμε να θεωρήσουμε τον αλγόριθμο 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]);
        }
    }
}
```

Όπου το **k** αναφέρεται στο χρονικό βήμα του αλγορίθμου και τα **i-j** τα ζευγάρια κόμβων.

Εάν οπτικοποιήσουμε τον πίνακα γειτνίασης A ως εξής, βλέπουμε ότι τα τετράγωνα που είναι στην ίδια στήλη και την ίδια γραμμή με το A[i][j] μπορούν να υπολογίζονται ταυτόχρονα, αφού δεν υπάρχει εξάρτηση μεταξύ τους παρά μόνο με το χρονικό βήμα k. Οπότε μπορούμε να πούμε πως οι λούπες i και j μπορούν να εκτελεστούν παράλληλα.



Recursive έκδοση

Θα φανταζόμαστε και εδώ πως χωρίζουμε τον πίνακα γειτνίασης ως εξής:

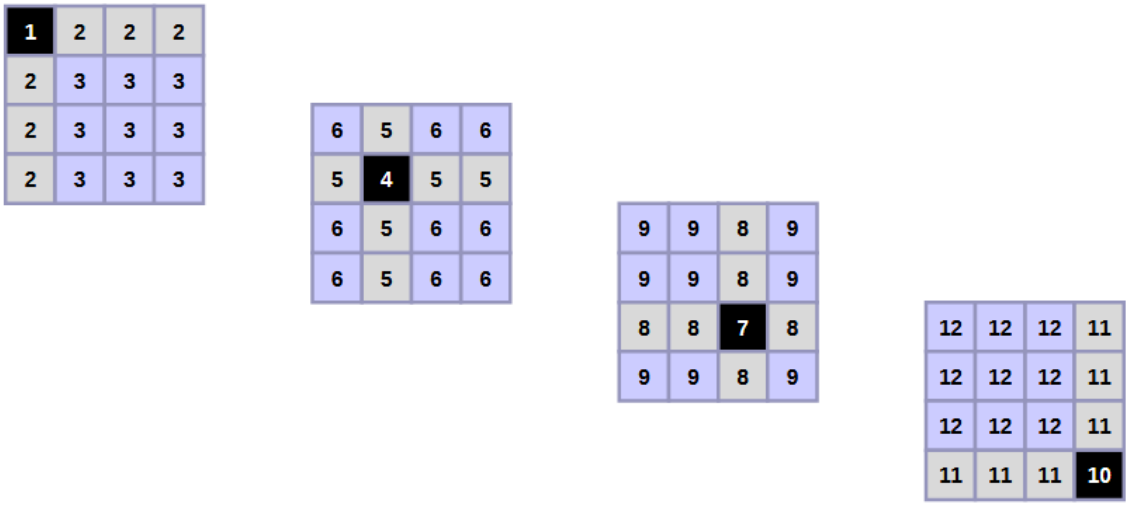
Ισχύει ότι ο πίνακας A~00~ πρέπει να υπολογιστεί πριν από τον A~11~. Οι πίνακες A~01~ και A~10~ όμως είναι ανεξάρτητοι και άρα μπορούν να επεξεργαστούν ταυτόχρονα.

Η εκτέλεση του αλγορίθμου μοιάζει:

```
FWR (A, B, C)
    if (base case)
        FWI (A, B, C)
    else
        FWR (A00, B00, C00);
        FWR (A01, B00, C01); // Αυτές οι κλήσεις μπορούν να εκτελεστούν
παράλληλα!
        FWR (A10, B10, C00); // Αυτές οι κλήσεις μπορούν να εκτελεστούν
παράλληλα!
        FWR (A11, B10, C01);
        FWR (A11, B10, C01);
        FWR (A10, B10, C00); // Και αυτές οι κλήσεις μπορούν να εκτελεστούν
παράλληλα!
        FWR (A01, B00, C01); // Και αυτές οι κλήσεις μπορούν να εκτελεστούν
παράλληλα!
        FWR (A00, B00, C00);
```

Tiled έκδοση

Ο σειριακός αλγόριθμος tiled δουλεύει ως εξής: Χωρίζει σε υποπίνακες το πρόβλημα έστω μεγέθους `size`. Σε κάθε βήμα **k** αφού ανανεωθεί το A[i][j] στοιχείο, ανανεώνονται όλα τα στοιχεία στην ίδια στήλη και γραμμή, και στο τέλος ανανεώνει όλα τα υπόλοιπα στοιχεία του tile.



Βλέποντας την εξέλιξη του αλγόριθμου όπως παραπάνω, αρχίζουμε να σκεφτόμαστε τα πράγματα που θα μπορούσαν να τρέχουν παράλληλα. Όταν ο αλγόριθμος εξετάζει το tile 1, τότε σίγουρα δεν μπορούμε να πειράξουμε τα tiles της ομάδας 3 γιατί υπάρχει εξάρτηση. Μπορούμε όμως να υπολογίσουμε *παράλληλα* τα tiles της ομάδας 2, για τα οποία δεν υπάρχει εξάρτηση να μας κρατά πίσω

Αντίστοιχα στα υπόλοιπα βήματα, τα tiles της ίδιας στήλης και γραμμής μπορούν να επεξεργάζονται παράλληλα, ενώ μόλις τελειώσει η επεξεργασία και γίνει κάποιος συγχρονισμός στα δεδομένα τους, μπορούν να επεξεργαστούν και πάλι παράλληλα τα υπόλοιπα (διαγώνια) tiles του πίνακα.

Θα μπορούσαμε να δώσουμε έναν ψευδωκώδικα όπως:

```
for (k = 0; k < matrix_size; k++){
    FW(init_tile)
    for (tile in same row or column){
        new task FW(tile)
    sync_data()  \\ some kind of synchronization
    }
    for (tile in rest of tiles){
        new task FW(tile)
    }
    sync_data()  \\ some kind of synchronization
}
```

Καλή μας τύχη στο κομμάτι της υλοποίησης!