ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

Άσκηση 1: Παραλληλοποίηση αλγορίθμων σε πολυπύρηνες αρχιτεκτονικές κοινής μνήμης

Χειμερινό εξάμηνο 2019-20 - Ροή Υ

Αντωνιάδης, Παναγιώτης
el15009@central.ntua.gr

Μπαζώτης, Νικόλαος
el15739@central.ntua.gr

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers."

– Gene Amdahl, *American computer architect*

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 Περιγραφή

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

Παρακάτω παρατίθεται η κύρια επανάληψη της σειριακής υλοποίησης που μας δίνεται:

```
for ( t = 0 ; t < T ; t++ ) {
    for ( i = 1 ; i < N-1 ; i++ ){
        for ( j = 1 ; j < N-1 ; j++ ) {
            // Compute number of neighbors
            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 ) )
                // Cell is alive
                current[i][j]=1;
            else
                // Cell is dead
                current[i][j]=0;
            }
    }
    // Swap current array with previous array
    swap=current;
    current=previous;
    previous=swap;
}
```

Αφού μετά από κάθε χρονική στιγμή ανανεώνουμε όλο το ταμπλό δεν μπορούμε να παραλληλοποιήσουμε την διαδικασία ως προς τις χρονικές στιγμές. Όμως, το γεγονός ότι η ανανέωση ενός κελιού εξαρτάται μόνο από τα γειτονικά του μας δίνει την δυνατότητα να παραλληλοποιήσουμε την διαδιακασία της ανανέωσης του N×N πίνακα. Χρησιμοποώντας το OpenMP, προσθέτουμε στην γραμμή 2 την εξής εντολή:

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

με η οποία ουσιαστικά ζητάμε να παραλληλοποιηθούν οι εσωτερικοί βρόγχοι με όσα νήματα έχουν οριστεί (μεταβλητή περιβάλλοντος OMP_NUM_THREADS). Οι μεταβλητή N και οι δύο δείκτες μοιράζονται από τα νήματα ενώ οι μεταβλητές i, j, nbrs είναι ατομικές για το κάθε νήμα.

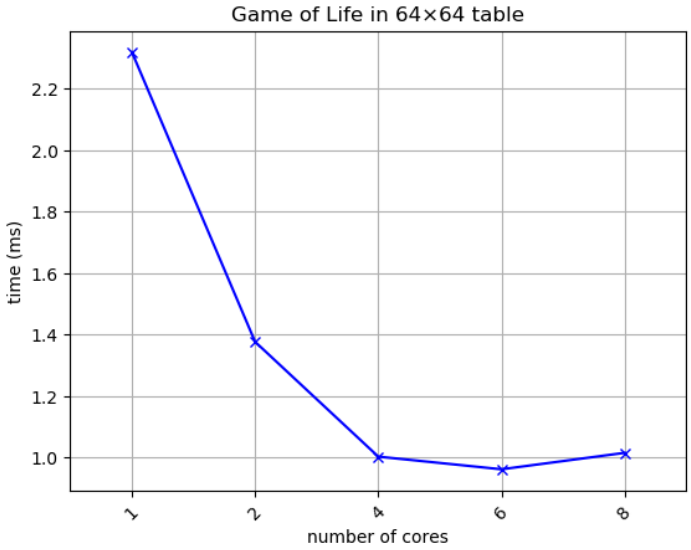
1.4 Αποτελέσματα

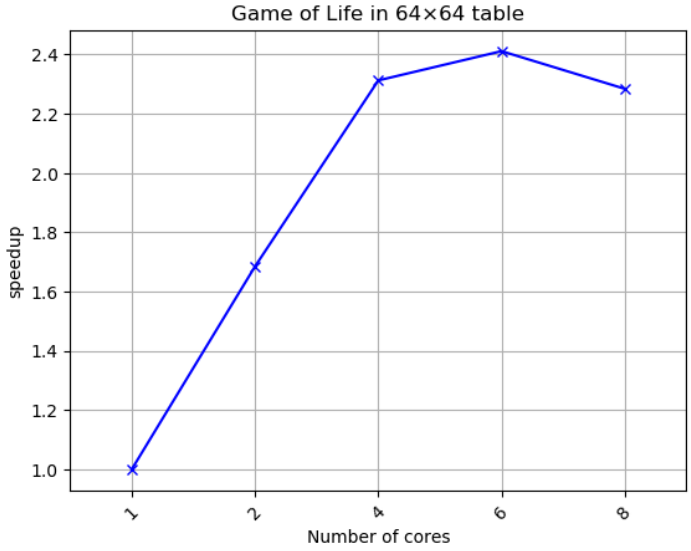
Όλες οι μετρήσεις για το Game of Life πραγματοποιήθηκαν για 1000 γενιές. Οι παράμετροι οι οποίοι μεταβάλλαμε κάθε φορά ήταν το μέγεθος του ταμπλό και ο αριθμός των πυρήνων και όλοι οι συνδυασμοί συνοψίζονται στον παρακάτω πίνακα (χρόνος σε second):

Cores \ Table Size	64×64	1024×1024	4096×4096
1	0.023186	10.962964	175.840471
2	0.013763	5.455029	88.314822
4	0.010027	2.722563	44.508848
6	0.009616	1.830053	36.922346
8	0.010151	1.377850	36.343052

Για να ερμηνεύσουμε τα παραπάνω αποτελέσματα, κατασκευάσαμε για κάθε μέγεθος ταμπλό δύο διαγράμματα όπου παριστάνουν την μεταβολή του χρόνου και του speedup καθώς αυξάνεται ο αριθμός των πυρήνων. Να σημειωθεί ότι ως speedup ορίζεται ο λόγος του χρόνου του σειριακού προγράμματος προς τον χρόνο του παράλληλου ($S = \frac{T_s}{T_p}$). Να σημειωθεί, ότι προφανώς αυτό που επιδιώκουμε όταν παραλληλοποιούμε ένα πρόγραμμα είναι να μειώσουμε τον χρόνο εκτέλεσης. Ιδανικά, θέλουμε η προσθήκη περισσότερων πυρήνων να μειώνει με ανάλογο τρόπο τον χρόνο εκτέλεσης του προγράμματος και ώστε $S = p$. Ωστόσο, αυτό δεν συμβαίνει διότι τα νήματα κάποια στιγμή θα επικοινωνήσουν για να ανταλλάξουν δεδομένα, όπως επίσης, η δημιουργία και ο τερματισμός των νημάτων απαιτεί κάποιο χρόνο που σε ένα σειριακό πρόγραμμα δεν θα χρειαζόταν.

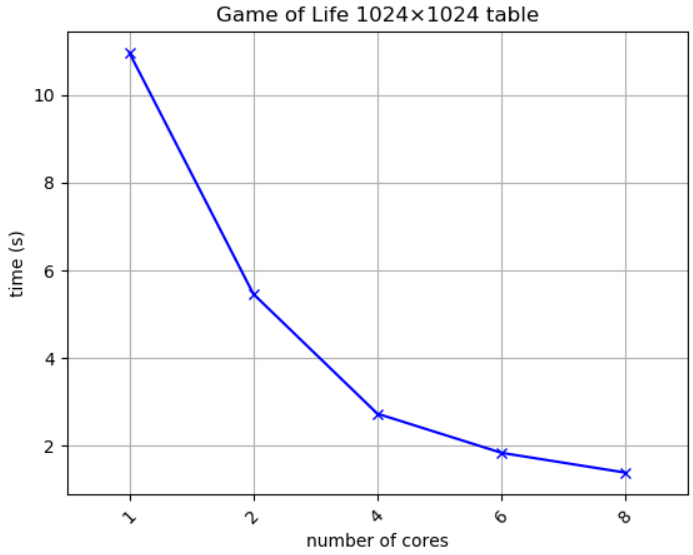
- **64×64**

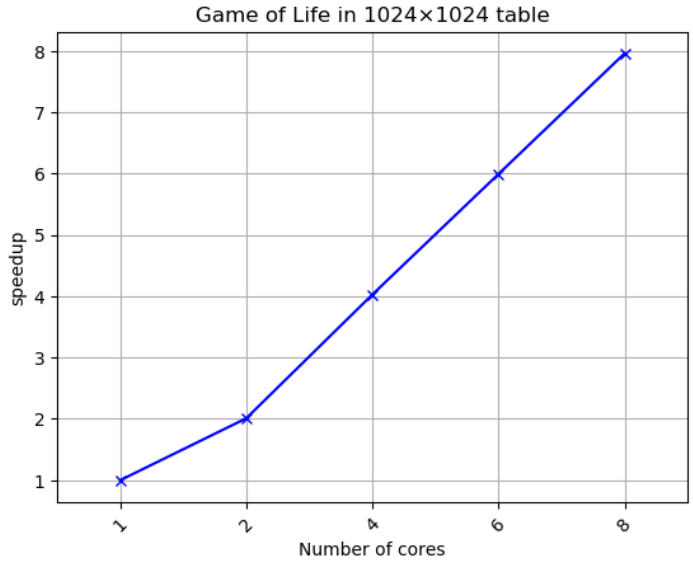




Σχολιασμός: Η μείωση του χρόνου δεν αυξάνεται ανάλογα με τον αριθμό των πυρήνων. Ειδικότερα, για 4 και 6 πυρήνες έχουμε ελάχιστη μείωση ενώ για 8 πυρήνες ο χρόνος εκτέλεσης αυξάνεται. Αυτό συμβαίνει, διότι το ταμπλό είναι μικρό και ο συνολικός χρόνος εκτέλεσης ακόμα και του σειριακού προγράμματος είναι μικρός. Έτσι, από ένα σημείο και μετά η δημιουργία των νημάτων και η επικοινωνία μεταξύ τους για την ανταλλαγή δεδομένων σπαταλά περισσότερο χρόνο από αυτόν που κερδίζουμε εκτελώντας το πρόγραμμα παράλληλα.

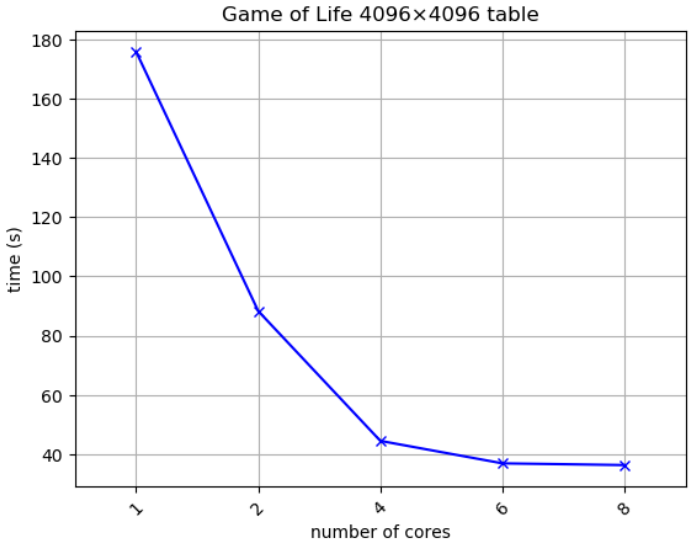
- **1024×1024**

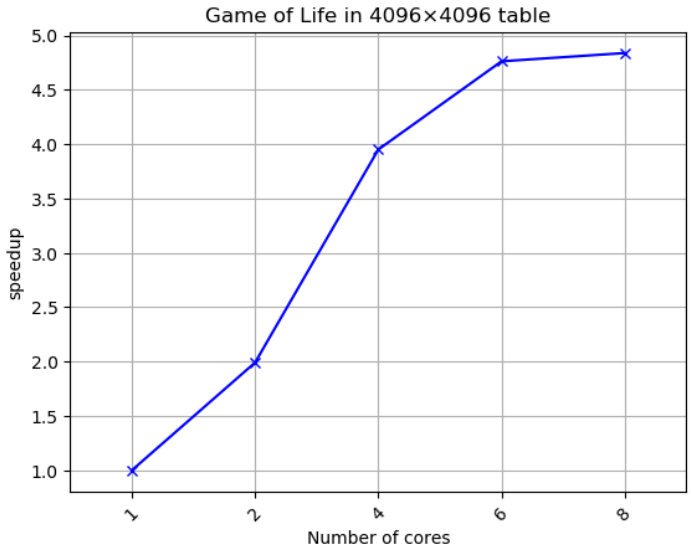




Σχολιασμός: Παρατηρούμε ότι το 1024×1024 ταμπλό αποτελεί μία πολύ καλή περίπτωση παραλληλισμού. Καθώς αυξάνουμε τον αριθμό των πυρήνων ο χρόνος πέφτει στο μισό και το speedup διπλασιάζεται. Αυτό συμβαίνει, διότι στο ταμπλό ο χρόνος που χάνεται για την επικοινωνία των νημάτων είναι μικρός σε σχέση με τον χρόνο που κερδίζουμε για την παραλληλοποίηση.

- **4096×4096**





Σχολιασμός: Εδώ, μέχρι και τους 4 πυρήνες έχουμε μείωση ανάλογη του αριθμού των πυρήνων που χρησιμοποιούμε. Στη συνέχεια, ελάχιστη αύξηση. Αυτό πιθανόν συμβαίνει γιατί το μεγάλο μέγεθος του ταμπλό δημιουργεί πολλούς γείτονες για κάθε νήμα με τους οποίους πρέπει να επικοινωνήσει. Ο χρόνος δημιουργίας και τερματισμού των νημάτων είναι αμελητέος, όμως ο χρόνος επικοινωνίας τους επηρεάζει πολύ μειώνοντας το τελικό speedup.

Συνολικά, παρατηρούμε ότι στο μικρό ταμπλό ο χρόνος δημιουργίας των νημάτων εμπόδιζε την επίτευξου μεγάλης μείωσης. Στο μεγάλο ταμπλό, το εμπόδιο ήταν η συνεχώς επικοινωνία των νημάτων (ίσως και η συμφόρηση στο διάδρομο της μνήμης μιας και έχουμε αρχιτεκτονική κοινής μνήμης). Στο μεσαίο μέγεθος του ταμπλό, πετυχαίνουμε μία ισορροπία εκμεταλλεύοντας στο μέγιστο τους πυρήνους που διαθέτουμε κάθε φορά.

1.5 Ειδικές Αρχικοποιήσεις

Θα συμπληρωθεί στην τελική αναφορά.

2 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου Floyd-Warshall σε αρχιτεκτονικές κοινής μνήμης

2.1 Σκοπός την ενότητας

Η ενότητα αυτή αποτελεί το βασικό κορμό της άσκησης. Στόχος της ενότητας είναι να αναπτύξουμε διαφορετικές παράλληλες εκδόσεις του αλγορίθμου Floyd-Warshall, να αξιολογήσουμε 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).

2.2 Ο αλγόριθμος Floyd-Warshall

Ο αλγόριθμος των Floyd-Warshall (FW) υπολογίζει τα ελάχιστα μονοπάτια ανάμεσα σε όλα τα ζεύγη των N κόμβων ενός γράφου (all-pairs shortest path). Θεωρώντας το γράφο αποθηκευμένο στον πίνακα γειτνίασης A, ο αλγόριθμος 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]);
```

Εκτός από την standard έκδοση του αλγορίθμου, έχουν προταθεί άλλες δύο εκδόσεις, μία αναδρομική (recursive) και μία tiled, προκειμένου να αξιοποιείται καλύτερα η κρυφή μνήμη [1].

2.3 Σχεδιασμός παραλληλοποίησης

2.3.1 Αρχική έκδοση

Ο αλγόριθμος απαιτεί τον υπολογισμό των τιμών A_{ij} για N χρονικά βήματα (ο κώδικας παρουσιάστηκε παραπάνω).

1. **Κατανομή Υπολογισμών**

   Θεωρούμε ως ένα task τον υπολογισμό της τιμής A_{ij} για ένα συγκεκριμένο χρονικό βήμα.

2. **Ορισμός ορθής σειράς εκτέλεσης μέσω task graph**

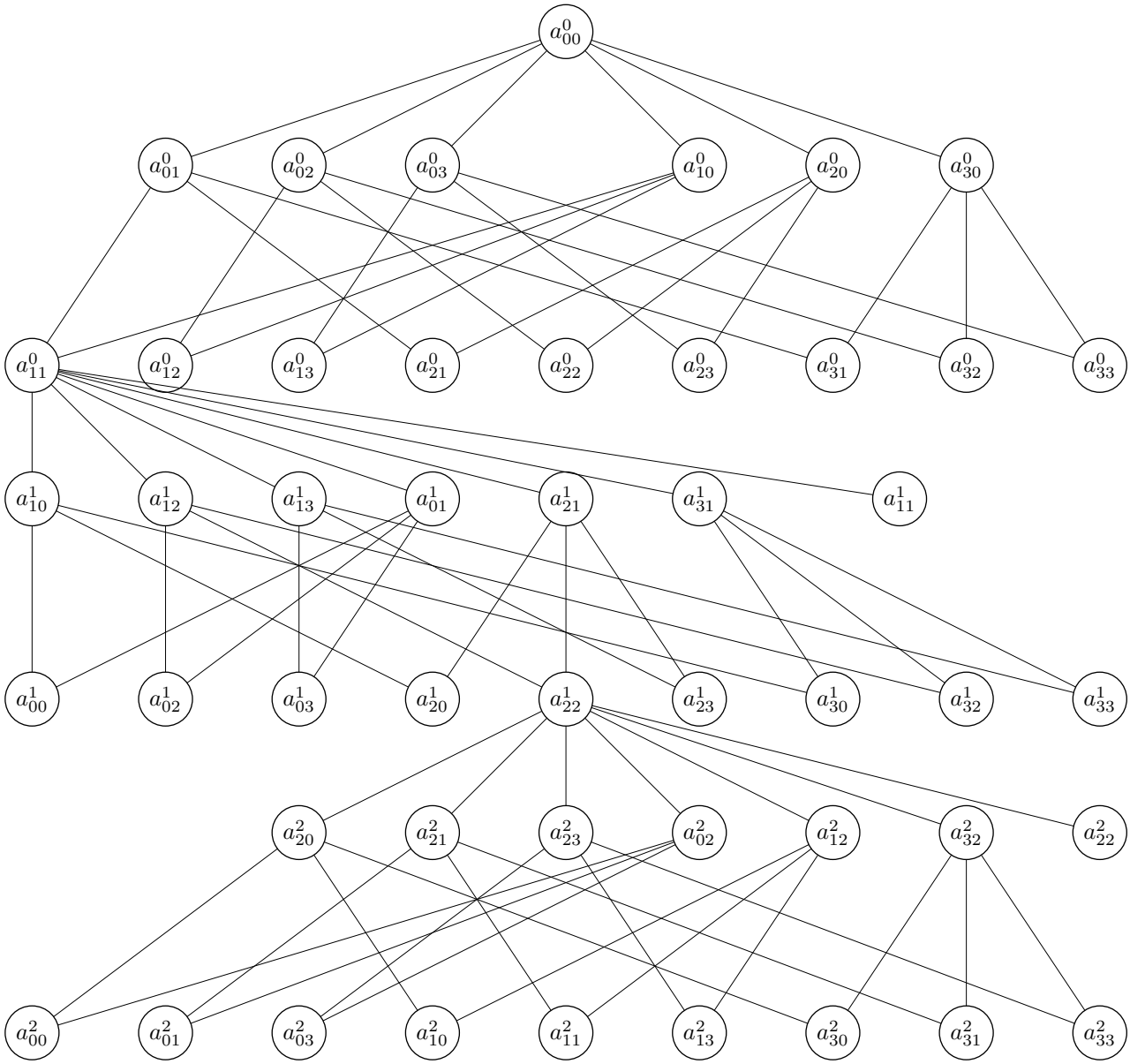
   Με τον συμβολισμό A^k_{ij}, αναφερόμαστε στην τιμή του κελιού A_{ij} την χρονική στιγμή k. Παρατηρώντας τον αλγόριθμο, βλέπουμε ότι μια συγκεκριμένη χρονική στιγμή k:

   - Η τιμή A^k_{kk} πρέπει να περιμένει μόνο την προηγούμενη τιμή της A^{k-1}_{kk}.
   - Οι τιμές A^k_{ik} και A^k_{ki} πρέπει επιπλέον να περιμένουν και το A^k_{kk}.
   - Οι υπόλοιπες τιμές μπορούν να υπολογιστούν μόνο όταν έχουν υπολογιστεί και το A^k_{ki} και το A^k_{ik}.

   Αν θεωρήσουμε ότι έχουμε τον παρακάτω 4×4 πίνακα (συμβολίζουμε εδώ με a τις τιμές του πίνακα, ενώ στις άλλες υλοποιήσεις θα συμβολίζουμε με A ολόκληρα blocks του πίνακα).:

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}

Προκύπτει το παρακάτω task graph:



Σημείωση: Προφανώς, στον παραπάνω γράφο, έχουμε άλλη μία επανάληψη που ξεκινάει από το a_{33}^2 για να υπολογιστούν οι τελικές τιμές και ακολουθεί το ίδιο μοτίβο.

2.3.2 Αναδρομική έκδοση

Ο αναδρομικός αλγόριθμος είναι ο εξής:

```
FWR (A, B, C)
    if (base case)
        FWI (A, B, C)
    else
        FWR (A11, B11, C11);
        FWR (A12, B11, C12);
        FWR (A21, B21, C11);
        FWR (A22, B21, C12);
        FWR (A22, B21, C12);
        FWR (A21, B21, C11);
        FWR (A12, B11, C12);
        FWR (A11, B11, C11);


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]);
```

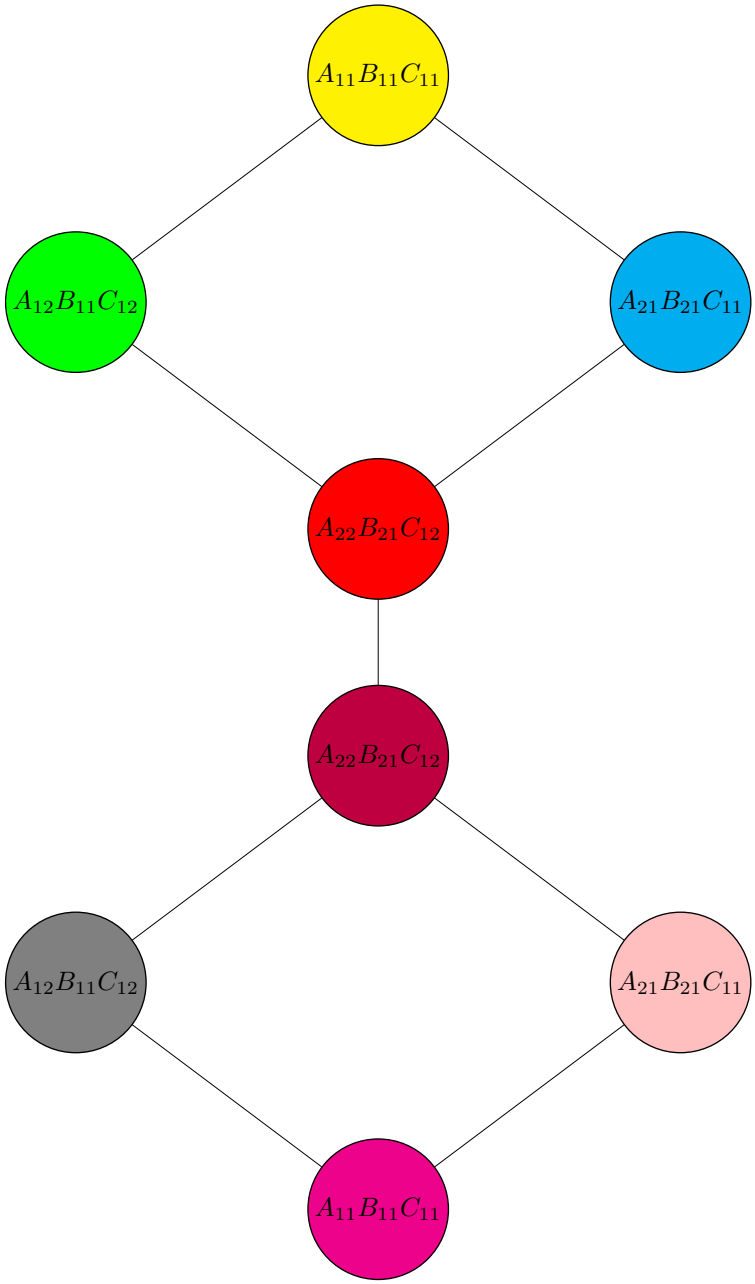
1. **Κατανομή Υπολογισμών**

   Η συνάρτηση FWI θα κληθεί όταν οι πίνακες είναι αρκετά μικροί ώστε να χωρέσουν στην cache. Μπορούμε να θεωρήσουμε ως ένα task την εκτέλεση της συνάρτησης FWI, αλλά αρχικά για λόγους παρουσιάσης του task graph θεωρούμε ως task τον υπολογισμό της συνάρτησης FWR στο πρώτο επίπεδο της ανάδρομης. Αφού η λειτουργία είναι αναδρομική, ο γράφος επεκτείνεται αναδρομικά όπως θα δείξουμε παρακάτω.

2. **Ορισμός ορθής σειράς εκτέλεσης μέσω task graph**

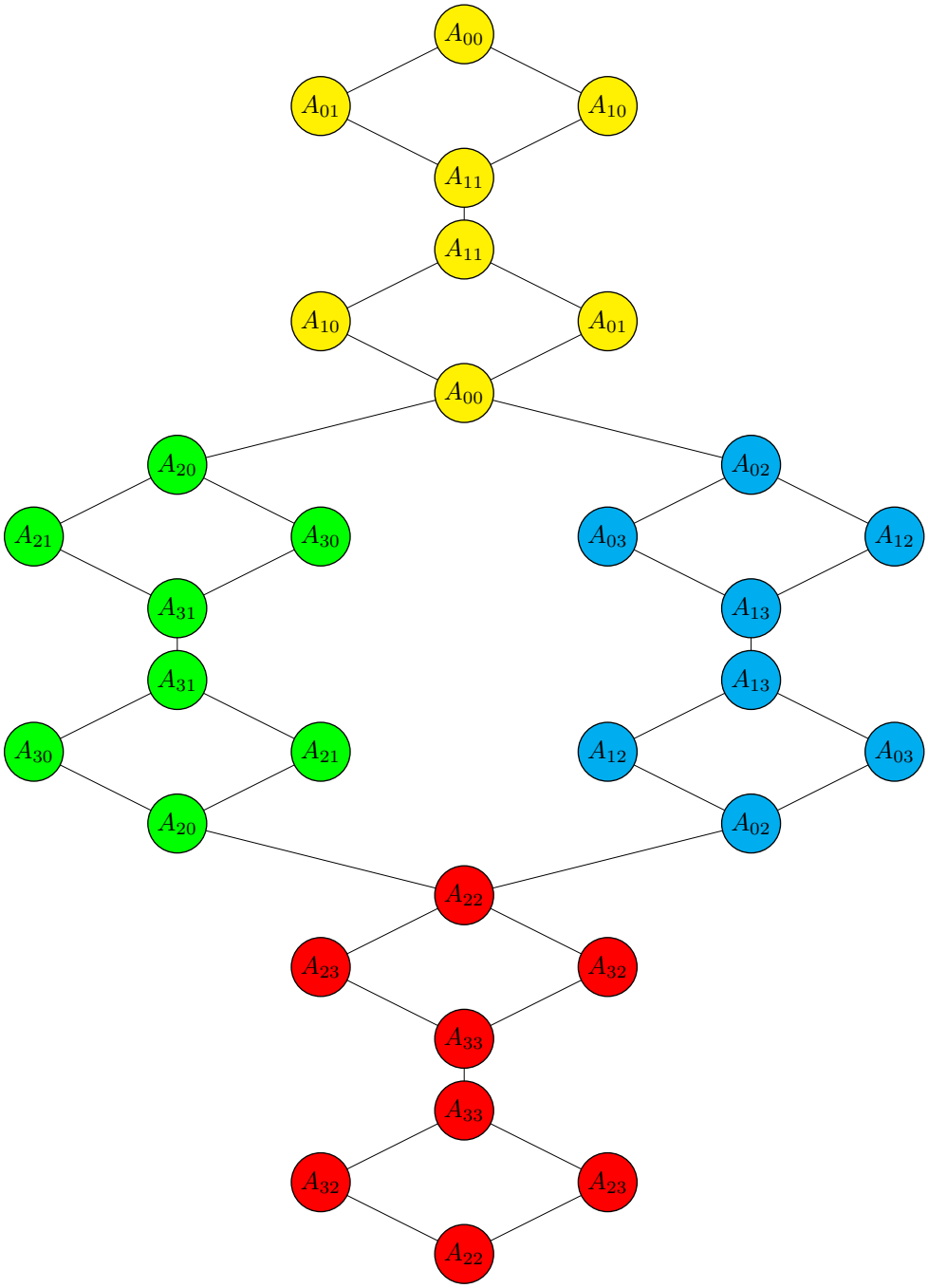
   Συμβολίζουμε ως $A_{i_1 j_1} B_{i_2 j_2} C_{i_3 j_3}$ την κλήση της συνάρτηση FWR() στο 1ο επίπεδο της αναδρομής στα αντίστοιχα block του πίνακα, σύμφωνα μα το παρακάτω:

A_{11}	A_{12}
A_{21}	A_{22}



Παρακάτω, γίνεται μία προσπάθεια να θεωρήσουμε ως task τον υπολογισμό της FWI στην περίπτωση όπου έχουμε 8 κόμβους και στην cache χωράει block 2×2, με αποτέλεσμα η αναδρομή να έχει τρία επίπεδα. Έχουμε δηλαδή τον παρακάτω πίνακα όπου κάθε A_{ij} είναι ένα block 2×2. Έχει διατηρηθεί ο ίδιος χρωματικός κώδικας έτσι ώστε να γίνει αντιληπτό σε ποιο task του 1ου γράφου αντιστοιχούν τα tasks του δεύτερου. Παρουσιάζονται οι 4 πρώτες κλήσεις της αναδρομής.

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}


A_{00}
A_{01}
A_{10}
A_{11}
A_{11}
A_{10}
A_{01}
A_{00}
A_{20}
A_{02}
A_{21}
A_{30}
A_{03}
A_{12}
A_{31}
A_{13}
A_{31}
A_{13}
A_{30}
A_{21}
A_{12}
A_{03}
A_{20}
A_{02}
A_{22}
A_{23}
A_{32}
A_{33}
A_{33}
A_{32}
A_{23}
A_{22}

2.3.3 Tiled έκδοση

Η tiled έκδοση του αλγορίθμου, αντίστοιχα με την recursive, στοχεύει στο να εφαρμόσει τον κλασσικό αλγόριθμο σε blocks του πίνακα (tiles εδώ). Έτσι, εφαρμόζει επαναληπτικά τις παρακάτω 3 φάσεις:

- Εφαρμογή FW στο tile A_{kk}
- Εφαρμογή FW στα tiles A_{ik} και A_{ki} χρησιμοποιώντας το A_{kk}.
- Εφαρμογή FW στα υπόλοιπα tiles A_{ij} χρησιμοποιώντας κάθε φορά τα A_{ik} και A_{ki}.

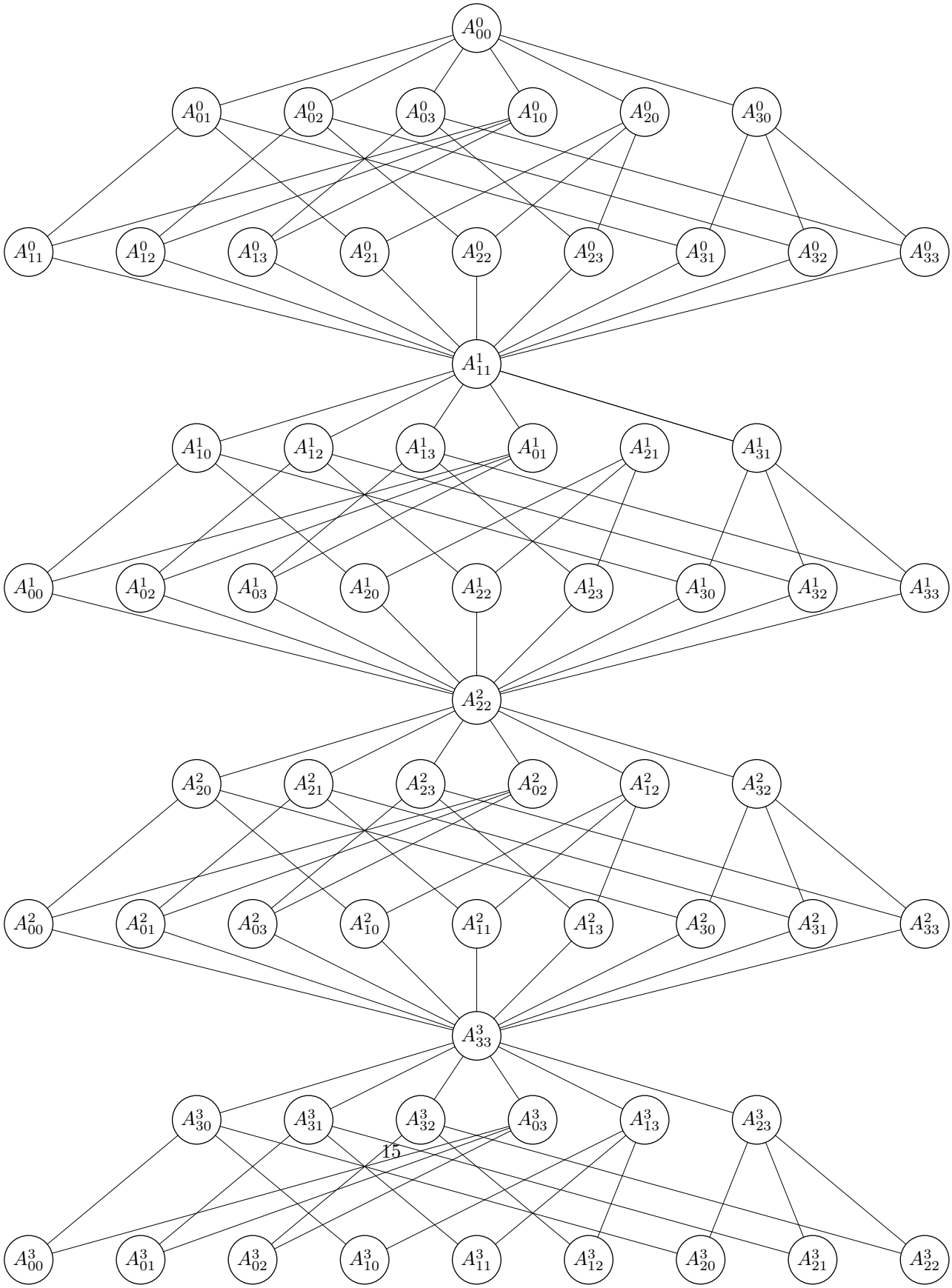
1. **Κατανομή Υπολογισμών**

   Θεωρούμε ως ένα task την εφαρμογή του κλασσικού Floyd-Warshall αλγορίθμου σε ένα tile.

2. **Ορισμός ορθής σειράς εκτέλεσης μέσω task graph**

   Παρουσιάζουμε το task graph για ένα πίνακα όπου $\frac{\textbf{\textit{table size}}}{\textbf{\textit{tile size}}} = 4$. Συνεπώς, ο πίνακας των tiles είναι ο εξής:

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}


A^0_{00}
A^0_{01} A^0_{02} A^0_{03} A^0_{10} A^0_{20} A^0_{30}
A^0_{11} A^0_{12} A^0_{13} A^0_{21} A^0_{22} A^0_{23} A^0_{31} A^0_{32} A^0_{33}
A^1_{11}
A^1_{10} A^1_{12} A^1_{13} A^1_{01} A^1_{21} A^1_{31}
A^1_{00} A^1_{02} A^1_{03} A^1_{20} A^1_{22} A^1_{23} A^1_{30} A^1_{32} A^1_{33}
A^2_{22}
A^2_{20} A^2_{21} A^2_{23} A^2_{02} A^2_{12} A^2_{32}
A^2_{00} A^2_{01} A^2_{03} A^2_{10} A^2_{11} A^2_{13} A^2_{30} A^2_{31} A^2_{33}
A^3_{33}
A^3_{30} A^3_{31} A^3_{32} A^3_{03} A^3_{13} A^3_{23}
A^3_{00} A^3_{01} A^3_{02} A^3_{10} A^3_{11} A^3_{12} A^3_{20} A^3_{21} A^3_{22}

Αναφορές

[1] J.-S. Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 769–782, 2004.

[2] "Σημειώσεις του μαθήματος."