

Συστήματα Παράλληλης Επεξεργασίας

Ομάδα 6

Βασίλειος Βρεττός - el18126,

Ανδρέας Βατίστας - el18020,

Αλέξανδρος Τσάφος - el18211

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Εθνικό
Μετσόβιο Πολυτεχνείο

1 Άσκηση 1 - Εξοικείωση με το περιβάλλον προγραμματισμού

Η συγκεκριμένη άσκηση είναι εισαγωγική με σκοπό την εξοικείωση μας με τα μηχανήματα του eslab στα οποία εκτελούμε τις εργασίες. Για να δοκιμάσουμε την κατανόησή μας, ζητήθηκε να παραλληλοποιήσουμε το Conway's Game of Life, ένα απλό παιχνίδι το οποίο "παίζεται" πάνω σε ένα ταμπλό (στην δική μας περίπτωση, διαστάσεων $N \times N$). Το παιχνίδι τρέχει για K γύρους (χρονικά διαστήματα). Το κάθε σημείο του ταμπλό έχει 2 καταστάσεις, είτε ζωντανό ή νεκρό. Οι 2 κύριοι κανόνες του παιχνιδιού είναι:

1. Αν ένα ζωντανό σημείο έχει περισσότερους από 3 γείτονες ή λιγότερους από 2, τότε γίνεται νεκρό στον επόμενο γύρο. Αν έχει ακριβώς 2 ή 3, παραμένει ζωντανό.
2. Αν ένα νεκρό σημείο έχει ακριβώς 3 γείτονες, γίνεται ζωντανό στον επόμενο γύρο

Τρέχουμε το παραπάνω σενάριο για 1000 γενιές (γύρους) σε πίνακες διαστάσεων 64×64 , 1024×1024 και 4096×4096 .

Ο κώδικας ο οποίος ζητείται να παραλληλοποιηθεί είναι ο εξής.

```
for ( t = 0 ; t < T ; t++ ) {  
    #pragma omp parallel for shared (previous, current) private (i,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;
    }

#ifdef OUTPUT
    print_to_pgm(current, N, t+1);
#endif
    //Swap current array with previous array
    swap=current;
    current=previous;
    previous=swap;
}

```

Το πρώτο for-loop, το οποίο είναι η αλλαγή των γενιών, δεν έχει νόημα να παραλληλοποιηθεί αφού χρειαζόμαστε την τιμή της προηγούμενης γενιάς για να υπολογίσουμε τις καταστάσεις των κελιών για την επόμενη γενιά.

Η δική μας προσθήκη στον κώδικα είναι η γραμμή.

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

Το συγκεκριμένο compiler directive (pragma) είναι χαρακτηριστικό του OpenMP. Είναι μια “οδηγία” προς τον μεταγλωττιστή η οποία του ζητάει να παραλληλοποιηθούν τα εσωτερικά for-loops με νήματα ορισμένα από ένα environmental variable. Στο συγκεκριμένο directive, επίσης ζητάμε:

1. Οι δείκτες previous, current να μοιράζονται μεταξύ των νημάτων. Η διάσταση N μοιράζεται by default.
2. Οι μεταβλητές i,j,nbrs να είναι ξεχωριστές για κάθε νήμα.

Την απόδοση του πολυνηματισμού θα εξετάσουμε με χρήση την μετρική του χρόνου, και κατά συνέπεια, του speedup $S = \frac{T_s}{T_p}$.

```

GameOfLife: Size 64 Steps 1000 Time 0.023134 Threads: 1
GameOfLife: Size 64 Steps 1000 Time 0.013553 Threads: 2
GameOfLife: Size 64 Steps 1000 Time 0.010247 Threads: 4
GameOfLife: Size 64 Steps 1000 Time 0.009112 Threads: 6
GameOfLife: Size 64 Steps 1000 Time 0.009294 Threads: 8
GameOfLife: Size 1024 Steps 1000 Time 10.969249 Threads: 1
GameOfLife: Size 1024 Steps 1000 Time 5.452860 Threads: 2
GameOfLife: Size 1024 Steps 1000 Time 2.724507 Threads: 4
GameOfLife: Size 1024 Steps 1000 Time 1.828835 Threads: 6
GameOfLife: Size 1024 Steps 1000 Time 1.376873 Threads: 8
GameOfLife: Size 4096 Steps 1000 Time 175.911689 Threads: 1
GameOfLife: Size 4096 Steps 1000 Time 88.237651 Threads: 2
GameOfLife: Size 4096 Steps 1000 Time 44.518161 Threads: 4
GameOfLife: Size 4096 Steps 1000 Time 37.291715 Threads: 6
GameOfLife: Size 4096 Steps 1000 Time 36.185633 Threads: 8

```

Figure 1: Έξοδος Game of Life

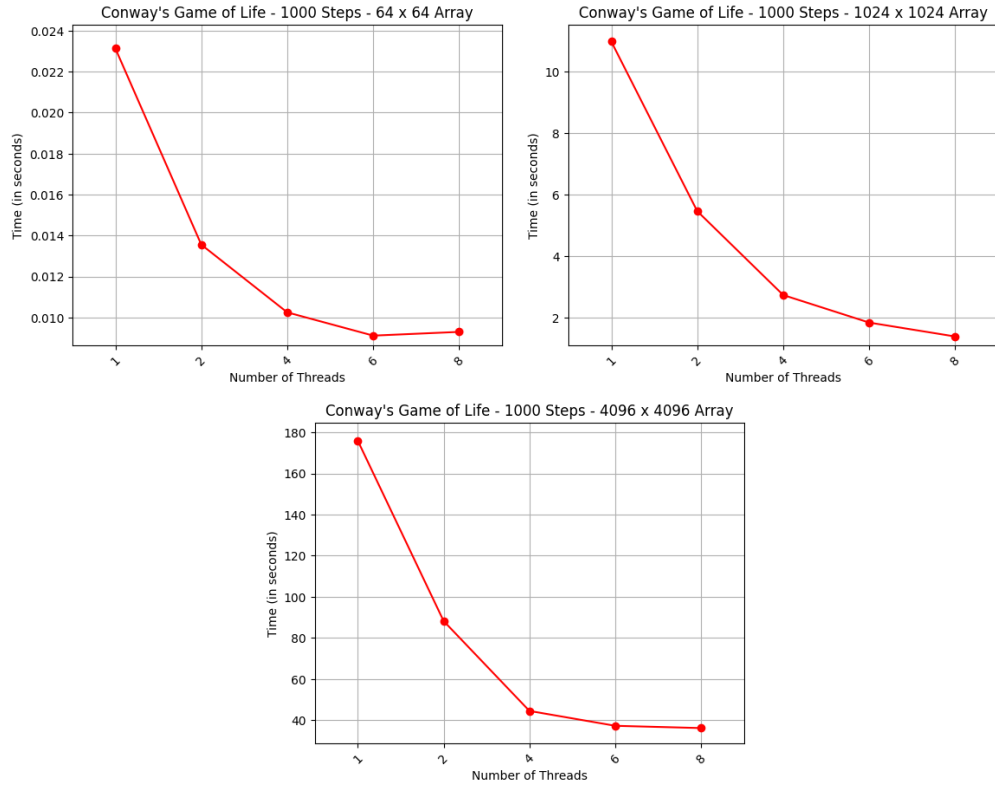


Figure 2: Χρόνος Εκτέλεσης Game of Life

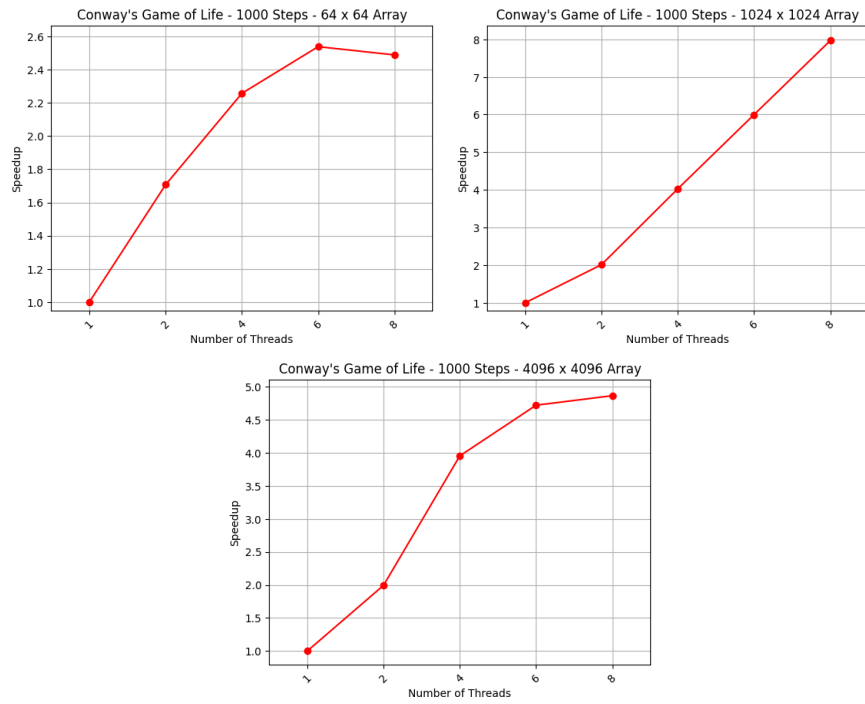


Figure 3: Speedup Game of Life

Συμπεράσματα

Για διαστάσεις 64x64

Παρατηρούμε ότι η μείωση του χρόνου δεν είναι ακριβώς ανάλογη των αριθμών των νημάτων. Από 1 σε 2 νήματα ή από 2 σε 4 νήματα βλέπουμε σημαντική βελτίωση της απόδοσης. Η εναλλαγή από 4 σε 6 νήματα προσφέρει πολύ μικρότερη αύξηση απόδοσης. Τέλος, από 6 σε 8 νήματα παρατηρούμε **ΑΥΞΗΣΗ** του χρόνου εκτέλεσης (μηδαμινή).

Η συγκεκριμένη αύξηση οφείλεται στο overhead που υπάρχει με την χρήση πολυνηματισμού σε μια διεργασία (γέννηση νημάτων, επικοινωνία κ.λ.π.). Ο συγκεκριμένος πίνακας είναι τόσο μικρός που η παραλληλοποίησή του περαιτέρω δεν βγάζει νόημα. Αν είχαμε την δυνατότητα να εξετάσουμε μεγαλύτερο αριθμό νημάτων, πολύ πιθανό να βλέπαμε περαιτέρω αύξηση του χρόνου εκτέλεσης

Για διαστάσεις 1024x1024

Παρατηρούμε ότι το speedup τείνει να είναι πλήρως γραμμικό. Ο χρόνος υποδιπλασιάζεται με κάθε διπλασιασμό νημάτων. Το συγκεκριμένο ταμπλό φαίνεται ιδανικό για παραλληλοποίηση. Η επικοινωνία μεταξύ νημάτων δεν περιορίζει την απόδοσή τους

Για διαστάσεις 4096x4096

Στους πρώτους 2 διπλασιασμούς των threads, βλέπουμε γραμμική μείωση του χρόνου (υποδιπλασιασμός) και κατά συνέπεια γραμμική αύξηση του speedup. Η λογική αυτή σταματάει να ισχύει για περαιτέρω αύξηση των νημάτων. Το συγκεκριμένο φαινόμενο δικαιολογείται από την ύπαρξη μεγάλου φόρτου (συμφόρησης) στον διάδρομο μνήμης. Υπάρχει συχνός διαμοιρασμός δεδομένων μεταξύ νημάτων καθώς αυξάνονται οι τιμές που πρέπει να ελεγχθούν για να υπολογιστούν οι γείτονες.

2 Άσκηση 2 - Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

Στην συγκεκριμένη άσκηση, σκοπός είναι η παραλληλοποίηση 2 διαφορετικών εκδόσεων των αλγορίθμων K-means και Floyd-Warshall. Χρησιμοποιούμε πιο "εξειδικευμένες" τεχνικές στο OpenMP και για πρώτη φορά συναντάμε και την ιδέα του task based parallelism.

2.1 Αλγόριθμος K-means

Ο αλγόριθμος k-means διαχωρίζει N αντικείμενα σε k μη επικαλυπτόμενες ομάδες. Ο παρακάτω ψευδοκώδικας περιγράφει τον αλγόριθμο.

```
until convergence (or fixed loops)
  for each object
    find nearest cluster
  for each cluster
    calculate new cluster center coordinates.
```

Στο συγκεκριμένο πρόβλημα μελετάμε 2 διαφορετικές εκδόσεις. Στην 1η, ο πίνακας των συντεταγμένων μοιράζεται από τα νήματα (shared cluster). Στην 2η υλοποίηση, δίνουμε στο κάθε νήμα μια τοπική έκδοση του πίνακα. Έτσι, τα νήματα μπορούν να κάνουν πράξεις χωρίς να υπάρχουν προβλήματα διαμοίρασμού (cache invalidation e.t.c.). Στο τέλος, κάθε νήμα ενημερώνει το master thread με τα δικά του αποτελέσματα (reduction).

2.1.1 K-means - Shared Clusters

Το μειονέκτημα της συγκεκριμένης υλοποίησης είναι ότι εφόσον λειτουργούμε πάνω σε διανοιζόμενα δεδομένα, υπάρχει ανάγκη η ανανέωση των shared variables **newClusterSize** και **newClusters** να γίνεται ατομικά.

Τα κλειδιά για την ανανέωση των μεταλητών έγινε με την χρήση του **atomic** pragma.

Το πρόγραμμα λήγει για δοσμένα iterations ή μέχρι να συγκλίνει. Τα παρακάτω δεδομένα είναι για τον συνδυασμό {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}.

```
#pragma omp parallel for default(shared) shared(newClusterSize, newClusters)
    private(i, j, index)
    for (i=0; i<numObjs; i++){
        /*
         *
        */
        #pragma omp atomic
        newClusterSize[index]++;
        /*
         *
        */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }
```

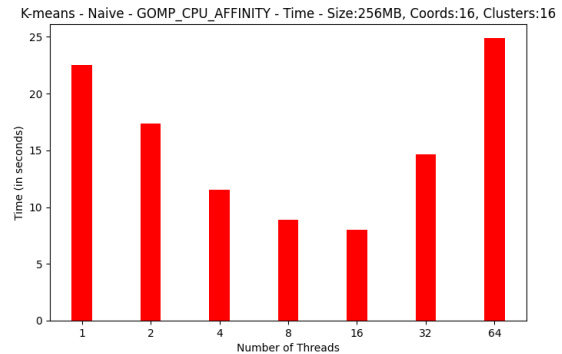
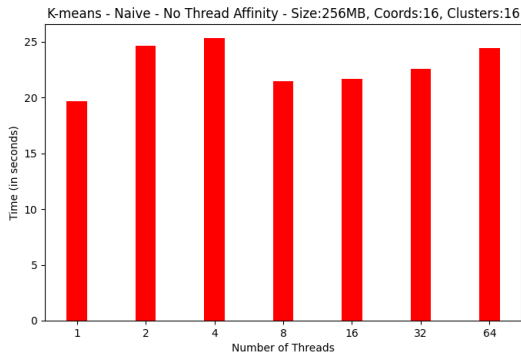


Figure 4: Χρόνος Εκτέλεσης K-Means Naive

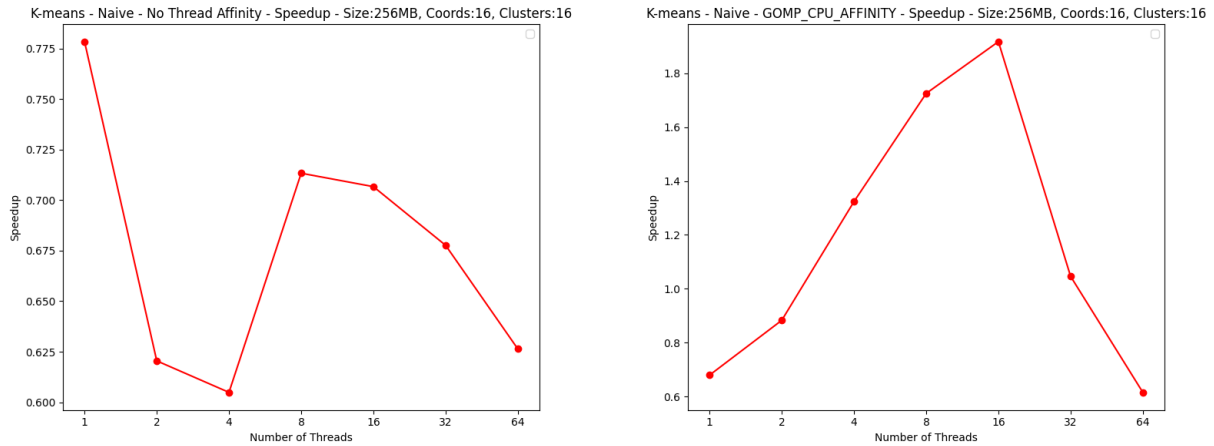


Figure 5: Speedup K-Means Naive

Σημείωση: Για το speedup θεωρούμε σειριακό χρόνο εκτέλεσης την έκδοση ΧΩΡΙΣ κλειδώματα. Εδώ το speedup με 1 νήμα είναι μικρότερο της μονάδας γιατί έχουμε κλειδώματα.

Το άμεσο συμπέρασμα είναι ότι το πρόγραμμα δεν βελτιώνεται με χρήση παραπάνω νημάτων. Συγκεκριμένα, η αύξηση των διαθέσιμων νημάτων αυξάνει τον χρόνο εκτέλεσης.

Η ύπαρξη κλειδωμάτων και η συχνή αλλαγή διαμοιραζόμενων μεταβλητών σημαίνει ότι χάνεται πολύς χρόνος σε συγχρονισμό και σε θέματα συνάφειας μνήμης.

Σαν προσπάθεια να βελτιώσουμε τα αποτελέσματα, χρησιμοποιήσαμε την μεταβλητή περιβάλλοντος `GOMP_CPU_AFFINITY`. Η λειτουργία αυτής της μεταβλητής είναι ότι προσδένει τα "λογικά" νήματα σε hardware νήματα των επεξεργαστών βάσει μιας ακολουθίας. Γνωρίζοντας ότι οι επεξεργαστές του Sandman θεωρούν ότι π.χ. στο Physical Core 0 υπάρχουν τα Logical Cores (0,32), επιλέξαμε τα νήματα να προσδένονται ακολουθιακά στους φυσικούς πυρήνες των επεξεργαστών.

Παρατηρήσαμε βελτίωση των χρόνων για τα πρώτα 32 νήματα. Μετά από αυτό το σημείο, τα πλέον καινούργια νήματα θα προσδένονται στα logical cores (Hyperthreading) και δεν θα δούμε κάποια μείωση χρόνου εκτέλεσης. Προσδένοντας νήματα σε συγκεκριμένους πυρήνες αποτρέπει το Λειτουργικό Σύστημα από το να αλλάζει θέση τα νήματα (λόγω χρονοδρομολόγησης) και να απαιτείται μεταφορά cache lines από πυρήνα σε πυρήνα.

Το thread affinity εξασφαλίζει τα νήματα να βρίσκονται όσο πιο κοντά γίνεται στα δεδομένα που επεξεργάζονται. Πραγματοποιούν μια πιο "NUMA Aware" εκτέλεση του προγράμματος. Οι συγκεκριμένες βελτιστοποιήσεις είναι low level στην φύση τους και αλλάζει η ταχτική που πρέπει να ακολουθήσουμε αναλόγως το πρόβλημα. Στην συγκεκριμένη υλοποίηση, είδαμε βέλτιστη απόδοση έχοντας πάντα νήματα στα πιο κοντινά NUMA nodes μεταξύ τους.

2.1.2 K-means - Copied Clusters and Reduce

Η έκδοση αυτή καταργεί το πρόβλημα των κλειδωμάτων και των διαμοιραζόμενων δεδομένων. Έχοντας δει τα πλεονεκτήματα του thread affinity στην προηγούμενη έκδοση, συνεχίζουμε την εφαρμογή του και εδώ.

Αρχικά παρουσιάζονται τα δεδομένα από την εκτέλεση έχοντας αγνοήσει τα tips για την αποφυγή του false-sharing.

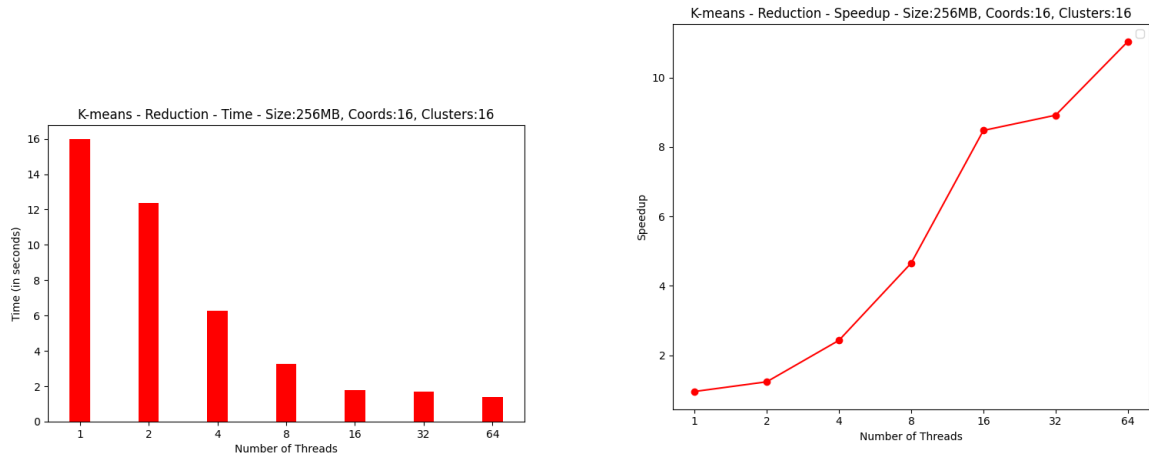


Figure 6: Γραφήματα K-Means Reduction - Simple

Η συγκεκριμένη έκδοση επιτυγχάνει πολύ καλύτερη επίδοση από την Shared Cluster. Χάνεται το μεγάλο overhead του συγχρονισμού που είχε η προηγούμενη. Το κυριότερο μειονέκτημα όμως είναι η μεγαλύτερη ανάγκη από μνήμη αφού δουλεύουμε με copied δεδομένα. Η ανάγκη για μνήμη είναι πολλαπλάσια της αρχικής έκδοσης, αναλόγως τα πόσα νήματα χρησιμοποιούμε. Για μεγαλύτερα προβλήματα μπορεί να ήταν και απαγορευτική η χρήση πολλών νημάτων.

Εξετάζοντας τα γραφήματα, παρατηρούμε σχεδόν γραμμικό speedup. Άρα, η έκδοση αυτή έχει πολύ ικανή δυνατότητα για παραλληλοποίηση.

Προχωράμε σε επόμενο configuration, {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}. Πλέον αναφερόμαστε σε αυτό το configuration ως "Hard".

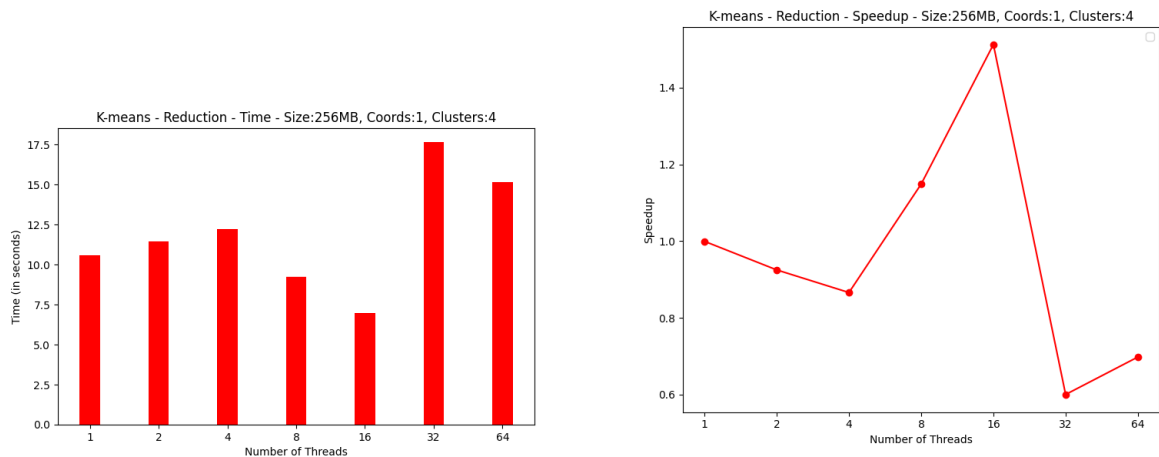


Figure 7: Γραφήματα K-Means Reduction - Hard

Το configuration αυτό έχει πολλά περισσότερα δεδομένα και μεγαλύτερο πίνακα objects, ο οποίος διαβάζεται από όλα τα νήματα. Αν 2 ή παραπάνω νήματα επεξεργάζονται **ΔΙΑΦΟΡΕΤΙΚΑ** δεδομένα στην ίδια Cache Line τότε έχουμε φαινόμενο false sharing.

Πλέον, πρέπει να βρούμε πιο έξυπνη λύση για να αποφύγουμε τέτοια προβλήματα. Τα Linux λειτουργούν με κάτι που ονομάζεται **First Touch Placement Policy**. Δηλαδή, όταν ένα νήμα καταχωρεί μνήμη (και αρχικοποιεί), η θέση αυτής της μνήμης θα είναι όσο πιο κοντά στο νήμα γίνεται. Η πολιτική αυτή είναι ορθότατη για σειριακές εφαρμογές.

Στην δική μας υλοποίηση, η καταχώρηση της μνήμης γινόταν από ένα νήμα, πριν μπούμε σε παράλληλη περιοχή. Για να γίνει το πρόβλημα περισσότερο **Operating System Aware**, καταχωρούμε την μνήμη πλέον ανά νήμα.

```
#pragma omp parallel private(k)
{
    k = omp_get_thread_num();
    local_newClusterSize[k] = (typeof(*local_newClusterSize))
        calloc(numClusters, sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters *
        numCoords, sizeof(**local_newClusters));
}
/*
*
*/
#pragma omp parallel default(shared) private(k)
{
    k = omp_get_thread_num();
    memset(local_newClusterSize[k], 0, numClusters * sizeof(float));
    memset(local_newClusters[k], 0, numClusters * numCoords * sizeof(float));
}
```

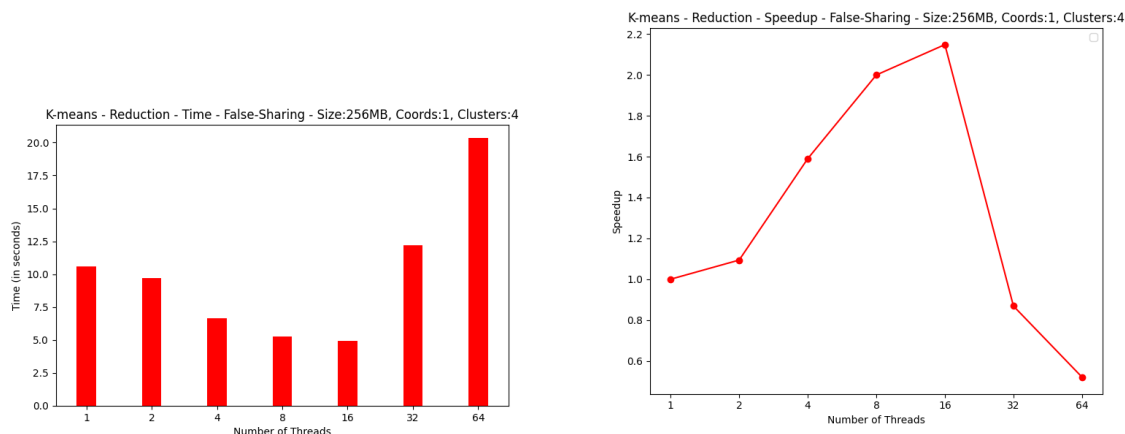


Figure 8: Γραφήματα K-Means Reduction - Hard - False Sharing Aware

Βελτιώνοντας την τοπικότητα των δεδομένων στα εκάστοτε νήματα λύνει κάποια από τα

προβλήματα του false sharing για το συγκεκριμένο πρόβλημα, αλλά τα αποτελέσματα δεν θεωρούνται ικανοποιητικά. Συγκριτικά με πριν, τα αποτελέσματα είναι αρκετά καλύτερα, καθώς βλέπουμε βελτίωση με αύξηση των νημάτων σε αντίθεση με πριν. Χρειάζεται όμως περισσότερη παρέμβαση από εμάς για να βελτιωθεί ο χρόνος.

Ο καλύτερος χρόνος που καταφέρνουμε είναι 4.9294s για 16 νήματα. Παραπάνω από 16 νήματα, παρατηρούμε χειροτέρευση της απόδοσης.

Η παρέμβαση αυτή θα γίνει κάνοντας ακόμα καλύτερα καταχώρηση των δεδομένων στα νήματα. Ο πίνακας objects είναι κρίσιμο μέρος του προβλήματος αφού γίνεται ανάγνωση μέρους αυτού, ξεχωριστό από όλα τα νήματα. Έχουμε ήδη λύσει το πρόβλημα με την τοπικότητα των local πινάκων.

Η τελευταία βελτιστοποίηση είναι η αρχικοποίηση του πίνακα objects ακριβώς με τον ίδιο τρόπο που έγινε και η αρχικοποίηση των τοπικών πινάκων, δηλαδή σε παράλληλη περιοχή, ανά νήμα.

```
#pragma omp parallel for private(i, j)
for (i=0; i<numObjs; i++)
{
    unsigned int seed = i;
    for (j=0; j<numCoords; j++)
    {
        objects[i*numCoords + j] = (rand_r(&seed) / ((float) RAND_MAX)) *
            val_range;
        if (_debug && i == 0)
            printf("object[i=%ld][j=%ld]=%f\n",i,j,objects[i*numCoords + j]);
    }
}
```

Το κάθε νήμα θα κάνει αρχικοποίηση του slice του οποίου τα δεδομένα θα επεξεργαστεί αργότερα.

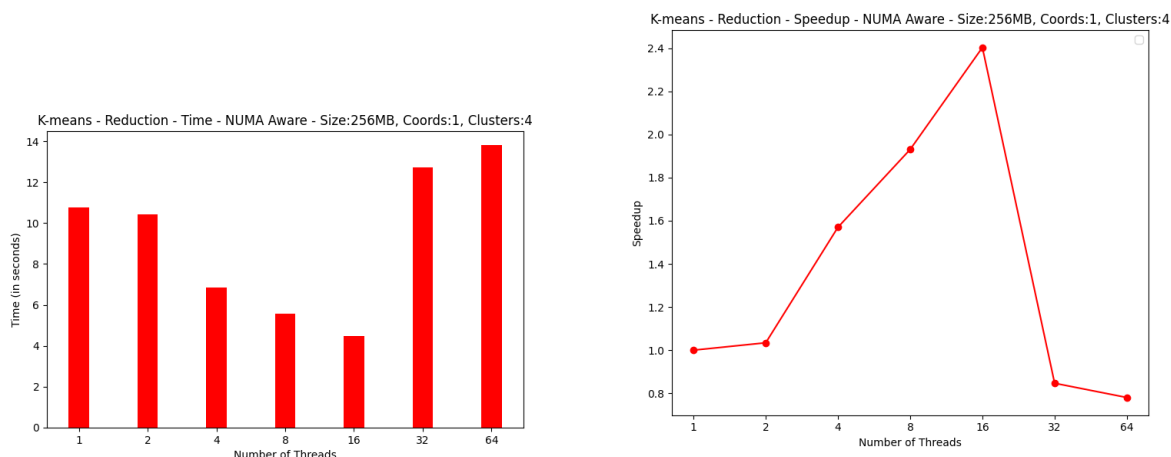


Figure 9: Γραφήματα K-Means Reduction - Hard - NUMA Aware

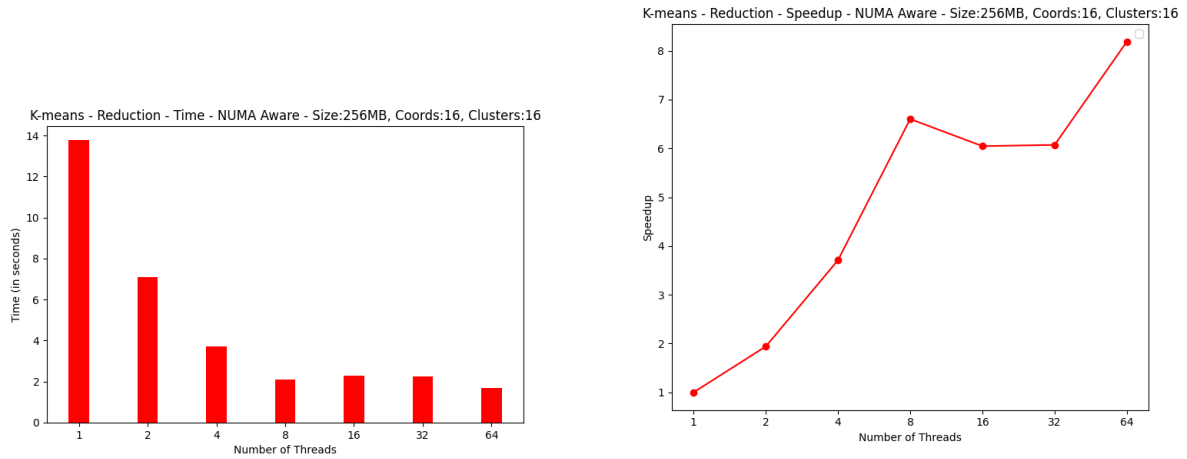


Figure 10: Γραφήματα K-Means Reduction - NUMA Aware

Η βελτίωση που υπέστει η "δύσκολη" περίπτωση είναι μικρή αλλά αξιοσημείωτη. Στο συγκεκριμένο σύστημα, δεν υπάρχει λόγος να χρησιμοποιήσουμε πάνω από 16 νήματα για την εκτέλεση του προγράμματος. Η πιο εύκολη περίπτωση ευνοήθηκε περισσότερο από τις αλλαγές, ειδικά για μικρότερο αριθμό νημάτων. Πάλι όμως, βγαινουμε στο συμπέρασμα ότι δεν υπάρχει λόγος να χρησιμοποιηθούν πάνω από 8 ή 16 νήματα γιατί η διαφορά στον χρόνο είναι μηδαμινή.

Best Times - Reduction

{Size, Coords, Clusters, Loops} = {256, 1, 4, 10}: 4.4818s - 16 Threads
 {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}: 1.6809s - 64 Threads

2.2 Αλγόριθμος Floyd-Warshall

Έστω πίνακας "γειτνίασης" A. Ο κώδικας σε C για τον αλγόριθμο Floyd-Warshall είναι ο 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]);
```

Το κομμάτι το οποίο μπορεί να παραλληλοποιηθεί είναι οι 2 εσωτερικά-φωλιασμένοι βρόγχοι.

2.2.1 Floyd-Warshall - Standard Edition

Η standard έκδοση του αλγορίθμου (σε C με OpenMP) είναι ως εξής:

```
for(k=0; k<N; k++)
  #pragma omp parallel for shared(A) private(i,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]);

```

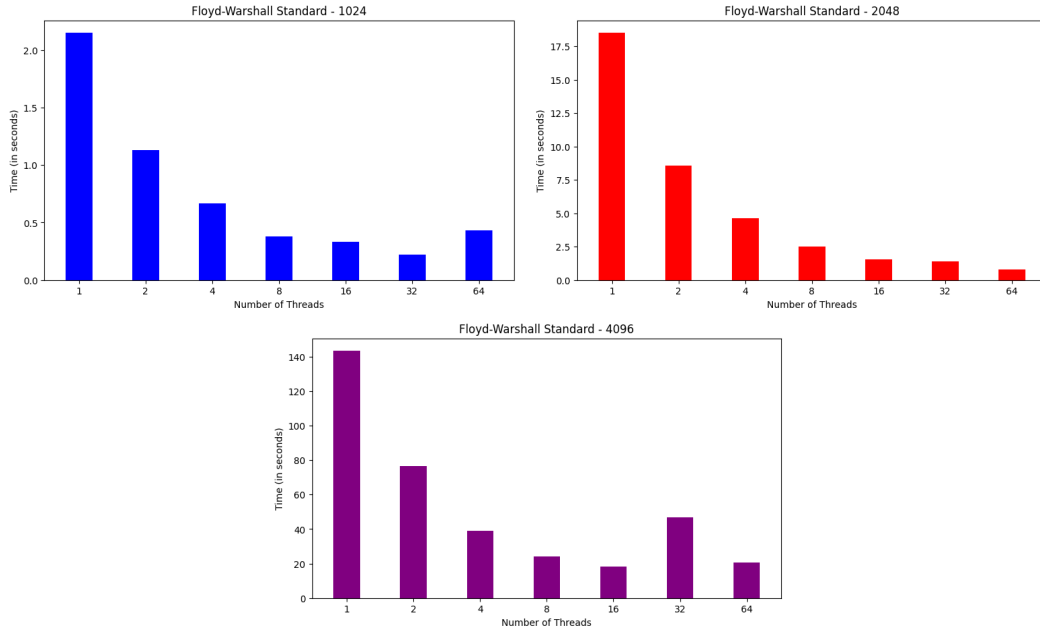


Figure 11: Χρόνος Εκτέλεσης FW Standard

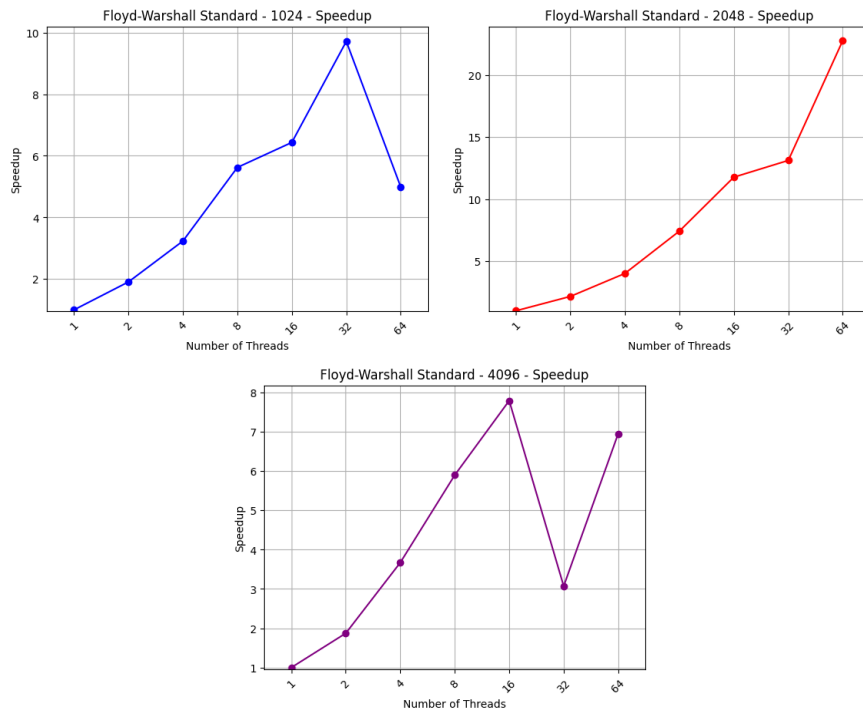


Figure 12: Speedup FW Standard

Best Times - Standard

Size 1024: 0.2213s - 32 Threads

Size 2048: 0.8121s - 64 Threads

Size 4096: 18.4082s - 16 Threads

Παρατηρούμε ότι τα συμπεράσματα περί κλιμάκωσης είναι ολόιδια με το πρόβλημα του Game Of Life. Για μικρούς πίνακες, υπάρχει μεγάλη δυνατότητα για κλιμάκωση, αλλά αυξάνοντας τα νήματα, το overhead που εισάγεται λόγω της επικοινωνίας μεταξύ τους περιορίζει την αύξηση της απόδοσης.

Για μεσαίους πίνακες, οι οποίοι χωράνε στην cache των πυρήνων, έχουμε ακόμα μεγαλύτερη δυνατότητα για κλιμάκωση.

Για μεγάλους πίνακες, το πρόβλημα γίνεται πιο περίπλοκο καθώς πλέον δεν χωράει ο πίνακας στις cache μνήμες του επεξεργαστή. Αυξάνονται οι ανάγκες για μεταφορά cache lines. Άρα πλέον το πρόβλημα είναι memory bound.

2.2.2 Floyd-Warshall - Recursive/Task Based

Το νόημα της συγκεκριμένης έκδοσης του αλγορίθμου είναι να τρέχει αναδρομικά μέχρι να συναντήσει πίνακα μεγέθους, επιλεγμένο από εμάς, βολικό για τον επεξεργαστή. Το μέγεθος αυτό ονομάζουμε block size. Δηλαδή, προσπαθούμε να χωρέσουμε ολόκληρο τον πίνακα στην cache. Παρακάτω παρουσιάζουμε το task graph της συνάρτησης αυτής για την πρώτη φορά που καλείται.

```
// Task 1: FWR(A00, B00, C00)
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

// Task 2: FWR(A01, B00, C01) and FWR(A10, B10, C00)
#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);
#pragma omp task shared(A,B,C)
    FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
#pragma omp taskwait

// Task 3: FWR(A11, B10, C01)
#pragma omp task shared(A,B,C)
    FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
        myN/2, bsize);
#pragma omp taskwait

// Task 4: FWR(A11, B10, C01)
#pragma omp task shared(A,B,C)
    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 taskwait

// Task 5: FWR(A10, B10, C00) and FWR(A01, B00, C01)
```

```

#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);
#pragma omp task shared(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);
#pragma omp taskwait

// Task 6: FWR(A00, B00, C00)
FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);

```

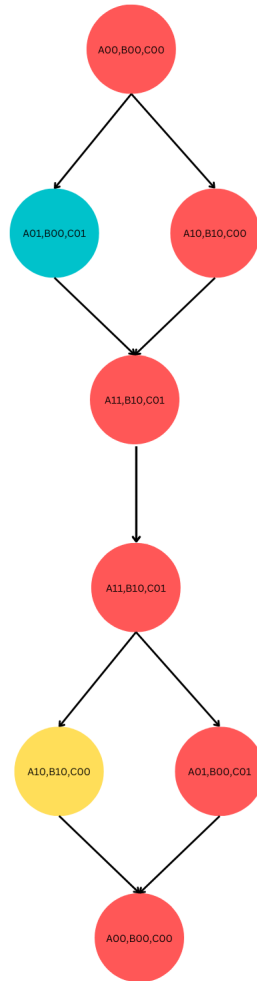
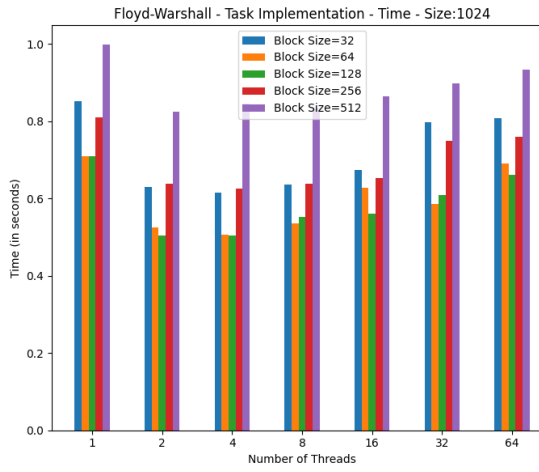
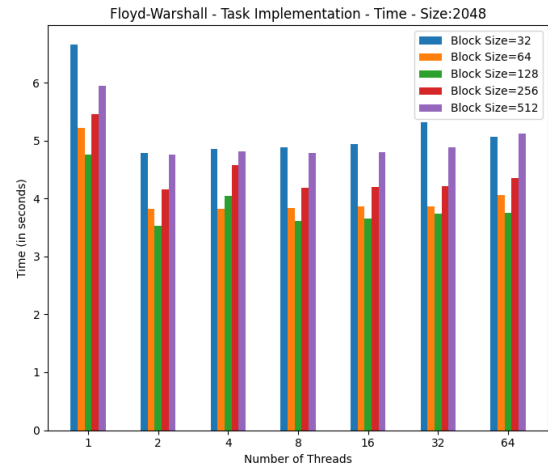


Figure 13: FW Recursive Task Graph - 1st iteration

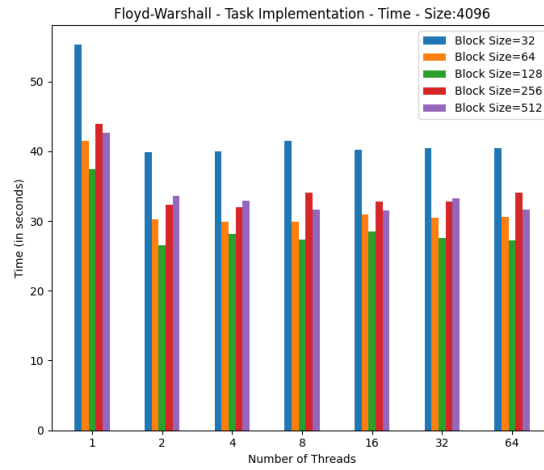
Η παραλληλοποίηση αυτής της έκδοσης γίνεται με χρήση task, ώστε να μπορεί να είναι αναδρομική.



(a) FW Recursive Size 1024

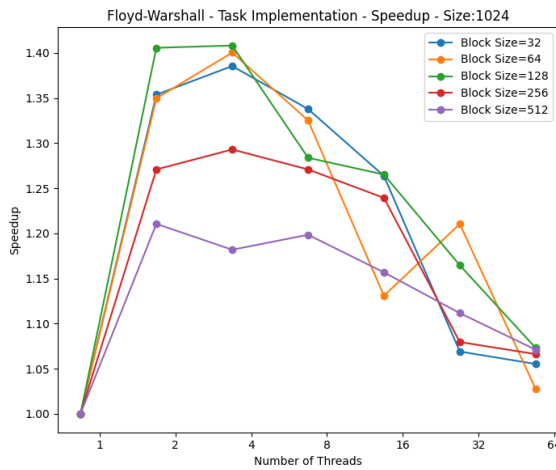


(b) FW Recursive Size 2048

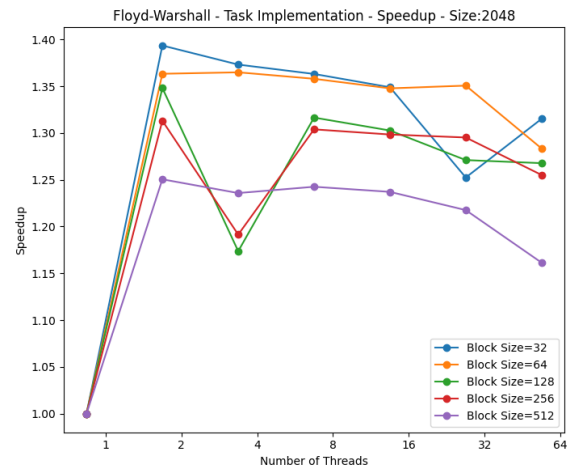


(c) FW Recursive Size 4096

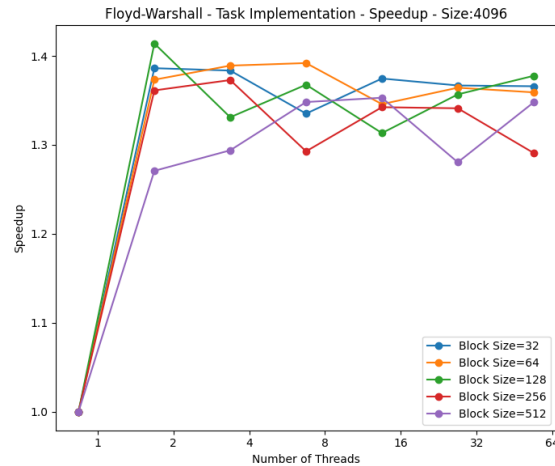
Figure 14: Χρόνος Εκτέλεσης FW Recursive



(a) FW Recursive Size 1024



(b) FW Recursive Size 2048



(c) FW Recursive - Size 4096 - Speedup

Figure 15: Speedup FW Recursive

Best Times - Task - No Nested

Size 1024: 0.5034s - 4 Threads - Block Size 128

Size 2048: 3.5248s - 4 Threads - Block Size 128

Size 4096: 26.4761s - 2 Threads - Block Size 128

Η πρώτη παρατήρηση είναι ότι το speedup δεν ξεπερνάει το 1.4. Σύμφωνα και με το task graph, η μέγιστη παραλληλοποίηση γίνεται με 2 νήματα σε κάθε επίπεδο αναδρομής. Ως προεπιλογή, το OpenMP **ΔΕΝ** επιτρέπει φωλιασμένη παραλληλοποίηση. Όταν ένα νήμα εισέρχεται σε παράλληλη περιοχή μέσα σε άλλη παράλληλη περιοχή, τότε δεν καλεί παραπάνω νήματα για την εκτέλεση αυτής της περιοχής. Άρα υπάρχουν περιοχές στο πρόγραμμα που εκτελούνται με το πολύ 1 εργάτη.

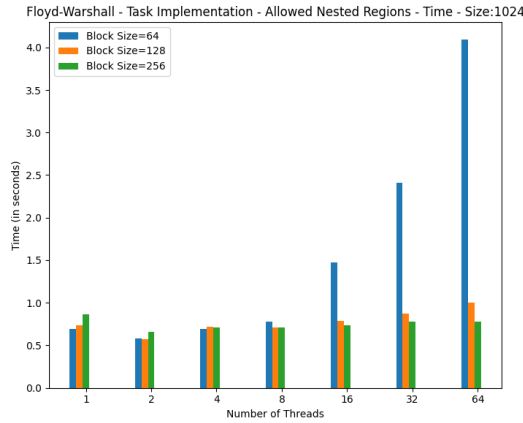
Η λύση για το παραπάνω είναι να αλλάζουμε την συμπεριφορά του OpenMP. Γίνεται να προσπεράσουμε αυτόν τον περιορισμό θέτοντας ένα environmental variable, το **OMP_NESTED** σε **TRUE**.

Η λύση αυτή δεν επαρκεί καθώς μετά μπορεί να επιφέρει αντίθετα αποτελέσματα από τα επιθυμητά. Επιτρέποντας την εκτέλεση φωλιασμένων παράλληλων περιοχών, υπάρχει ο κίνδυνος να καλέσουμε πολλά νήματα για την δημιουργία των task αλλά να μην υπάρχουν διαθέσιμα νήματα για την ίδια την εκτέλεση των πράξεων.

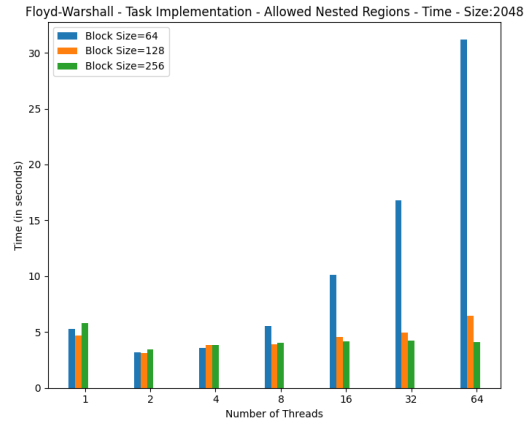
Χρειάζονται περαιτέρω περιορισμοί. Αυτοί δίνονται από το OpenMP με την μορφή μεταβλητών περιβάλλοντος.

Οι **SUNW_MP_MAX_POOL_THREADS** και **SUNW_MP_MAX_NESTED_LEVELS** περιορίζουν τον αριθμό των διαθέσιμων νημάτων/δούλων (δηλαδή όλων εκτός του νήματος αφέντη) και το μέγιστο βάθος που επιτρέπονται από τα νήματα να καλούν άλλα νήματα αντίστοιχα.

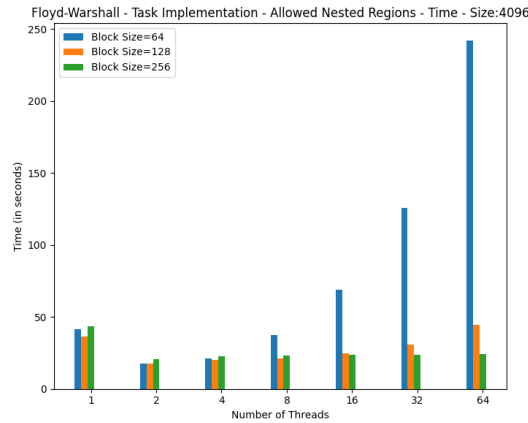
Εφαρμόζοντας τα παραπάνω, βγάζουμε τα παρακάτω αποτελέσματα:



(a) FW Recursive + Nested Size 1024



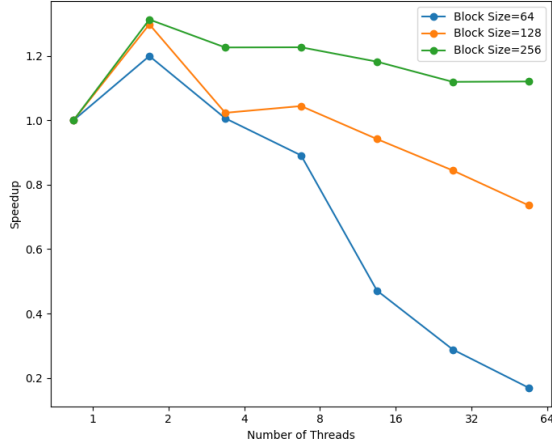
(b) FW Recursive + Nested Size 2048



(c) FW Recursive + Nested Size 4096

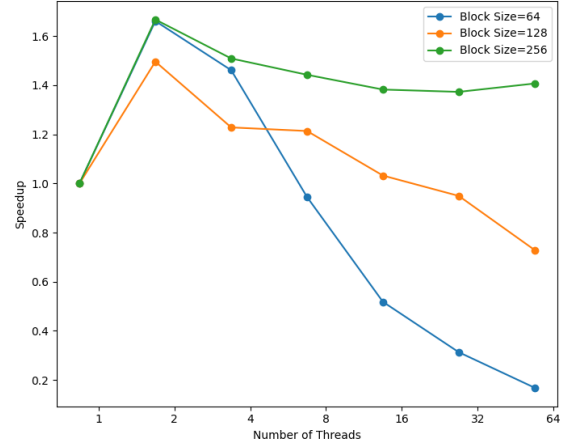
Figure 16: Χρόνος Εκτέλεσης FW Recursive/Nested

Floyd-Warshall - Task Implementation - Allowed Nested Regions - Speedup - Size:102



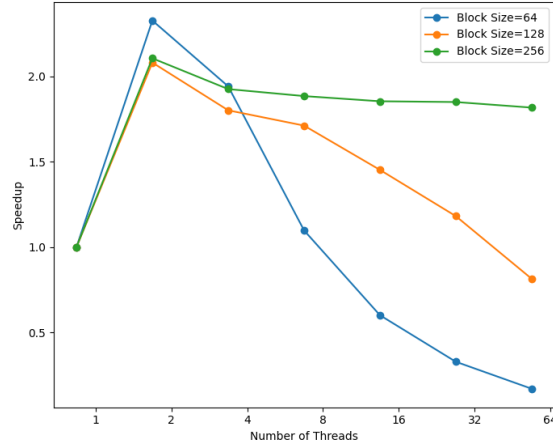
(a) FW Recursive - Size 1024 - Speedup

Floyd-Warshall - Task Implementation - Allowed Nested Regions - Speedup - Size:2048



(b) FW Recursive - Size 2048 - Speedup

Floyd-Warshall - Task Implementation - Allowed Nested Regions - Speedup - Size:4096



(c) FW Recursive - Size 4096 - Speedup

Figure 17: Speedup FW Recursive/Nested

Best Times - Task - With Nesting

Size 1024: 0.5688s - 2 Threads - Block Size 128

Size 2048: 3.1347s - 4 Threads - Block Size 128

Size 4096: 17.3402s - 2 Threads - Block Size 128

Σημείωση: Για το speedup θεωρούμε σειριακό χρόνο εκτέλεσης για κάθε έκδοση τον χρόνο εκτέλεσης με ένα νήμα.

Και οι 2 εκδόσεις του recursive αλγορίθμου δεν φτάνουν σε απόδοση τον standard αλγόριθμο, σε ζήτημα scaling. Χρειάζεται περισσότερη μελέτη από το μέρος μας για να φτάσουμε καλύτερες επιδόσεις, κυρίως με καλύτερη χρήση των μεταβλητών περιβάλλοντος της OpenMP.

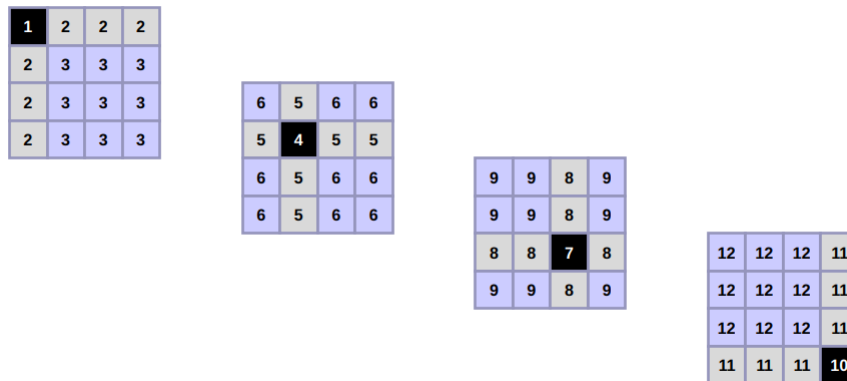
Η κατάλληλη επιλογή block size εξαρτάται κάθε φορά από τον επεξεργαστή που χρησι-

μπορούμε. Σημαντικό να μην χρησιμοποιηθούν μικρά block sizes στην έκδοση που επιτρέπονται οι φωλιασμένες παράλληλες περιοχές γιατί γρήγορα υπερφορτώνεται το pool απο task όμως δεν υπάρχουν διαθέσιμα νήματα να δεχθούν task.

2.2.3 Floyd-Warshall - Tiled

Στην συγκεκριμένη έκδοση, εκμεταλλευόμαστε το παρακάτω σχήμα/ακολουθία πράξεων.

Figure 18: Βήματα Παράλληλισμού FW Tiled



Σε κάθε βήμα, υπολογίζουμε πρώτα το διαγώνιο στοιχείο του πίνακα (k-οστό), μετά υπολογίζουμε τα στοιχεία της ίδιας γραμμής και στήλης και μετά τα υπόλοιπα στοιχεία του πίνακα. Αυτές οι 3 ενέργειες πρέπει να γίνουν διαδοχικά μεταξύ τους με αυτή την σειρά. Αυτό σημαίνει ότι οι 2 τελευταίες ενέργειες μπορούν να γίνουν ταυτόχρονα (μεταξύ τους).

Υλοποιήσαμε 2 διαφορετικές εκδόσεις, μία με παράλληλισμό βρόγχων και μια με παράλληλισμό βρόγχων και tasks.

Τα αποτελέσματα που παρουσιάζουμε είναι από την έκδοση με τον παράλληλισμό βρόγχων.

Κώδικας FW Tiled

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

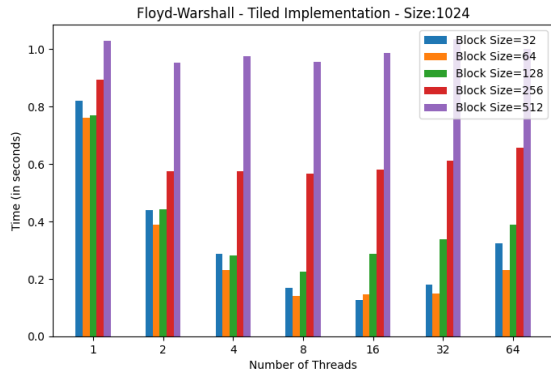
    #pragma omp for collapse(2) nowait
    for(i=k+B; i<N; i+=B)
        for(j=0; j<k; j+=B)
            FW(A,k,i,j,B);

    #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);
}
}

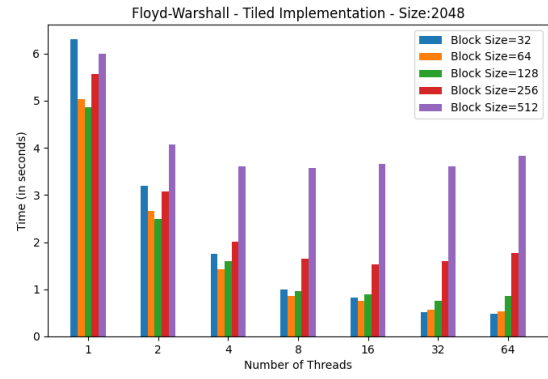
```

Η επιλογή `nowait` χρησιμοποιείται αφού μεταξύ τους οι βρόγχοι της κάθε παράλληλης περιοχής μπορούν να τρέξουν ταυτόχρονα.

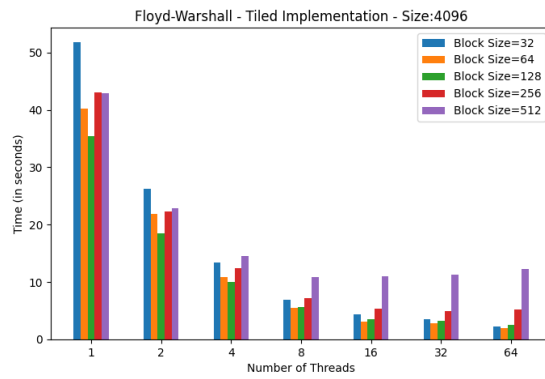
Η επιλογή `collapse(2)` χρησιμοποιείται καθώς χωρίς αυτήν, ο 2ος βρόγχος θα έτρεχε μονονηματικά, από το νήμα που εκτελεί το `for` loop. Με το συγκεκριμένο, μπορούν να χρησιμοποιηθούν όλα τα νήματα του συστήματος για να παραλληλοποιηθεί ο τέλει φωλιασμένος βρόγχος.



(a) FW Tiled - Size 1024

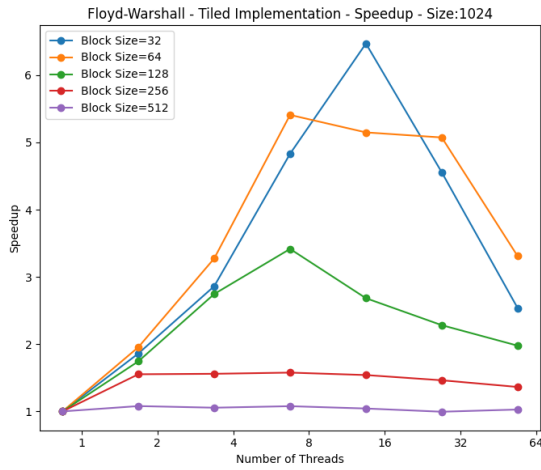


(b) FW Tiled - Size 2048

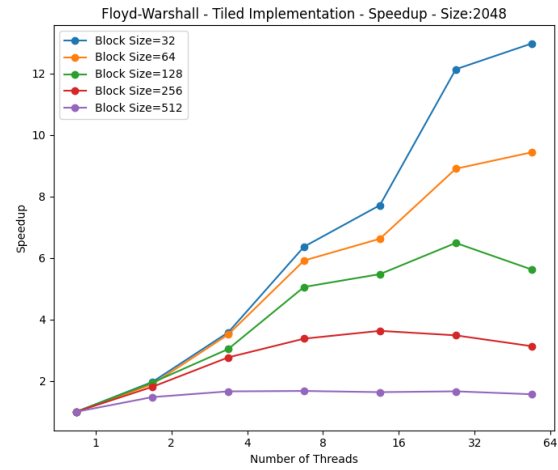


(c) FW Tiled - Size 4096

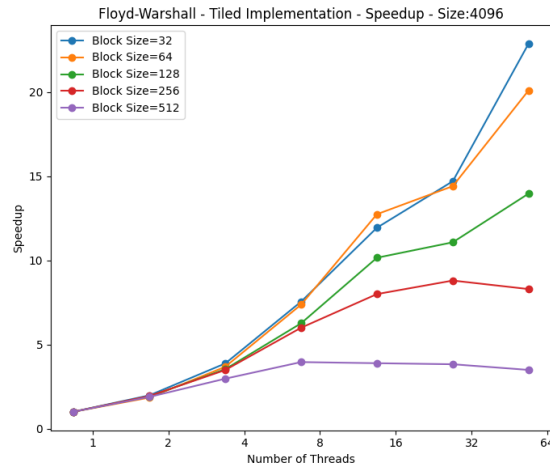
Figure 19: Χρόνος Εκτέλεσης FW Tiled



(a) FW Tiled - Size 1024 - Speedup



(b) FW Tiled - Size 2048 - Speedup



(c) FW Tiled - Size 4096 - Speedup

Figure 20: Speedup FW Tiled

Best Times - Tiled - ParFor

Size 1024: 0.1267s - 16 Threads - Block Size 32

Size 2048: 0.4858s - 64 Threads - Block Size 32

Size 4096: 1.9965s - 64 Threads - Block Size 64

Οι χρόνοι εκτέλεσης είναι πλέον πολύ καλύτεροι και από τις 2 εκδόσεις. Συγκριτικά με την αρχική έκδοση, όπου ο καλύτερος χρόνος ήταν 20.6s (με 64 νήματα) για τον πίνακα μεγέθους 4096, πλέον βέλτιστος χρόνος είναι 1.99s για block size=64.

Η Tiled έκδοση φαίνεται να κλιμακώνει πολύ καλύτερα από την standard για μεγάλους πίνακες. Για μεσαίους και μικρούς, τα αποτελέσματα είναι αρκετά παρόμοια. Είναι καλύτερη επιλογή από άποψη αξιοποίησης πόρων καθώς ακόμα και με μικρότερο αριθμό νημάτων, πετυχαί-

νουμε πολύ καλύτερους χρόνους.