Συστήματα Παράλληλης Επεξεργασίας

Ομάδα 6
Βασίλειος Βρεττός - el18126,
Ανδρέας Βατίστας - el18020

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 cslab 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 x 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 64x64, 1024x1024 και 4096x4096.

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

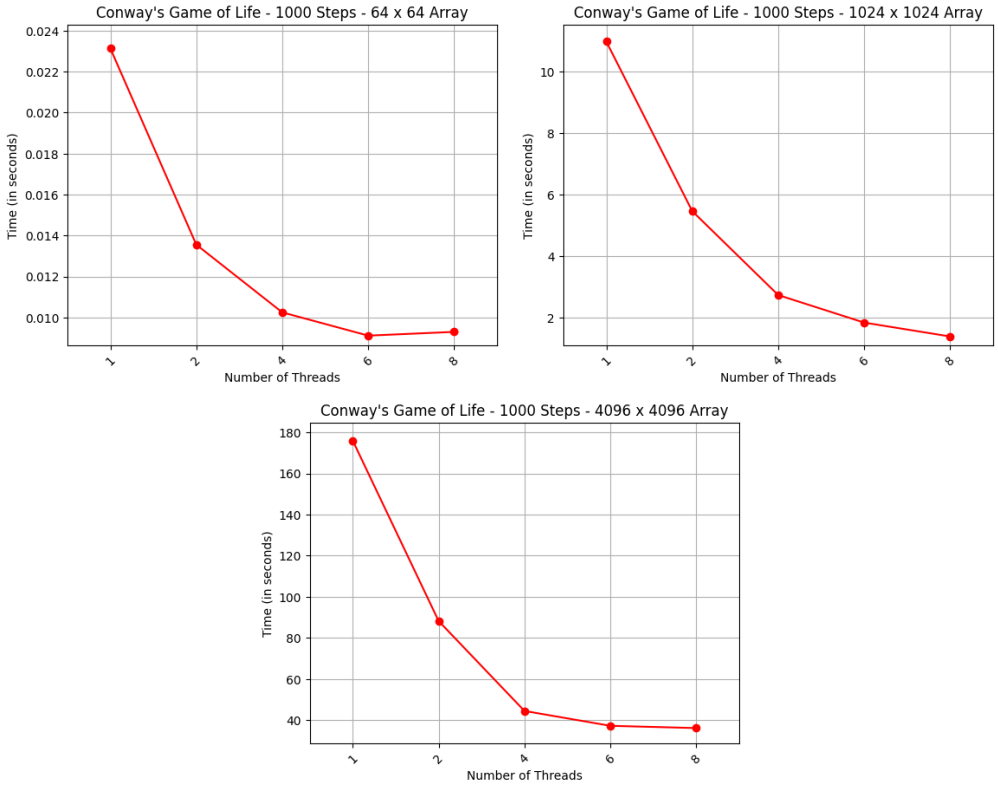


Figure 2: Χρόνος Εκτέλεσης Game of Life

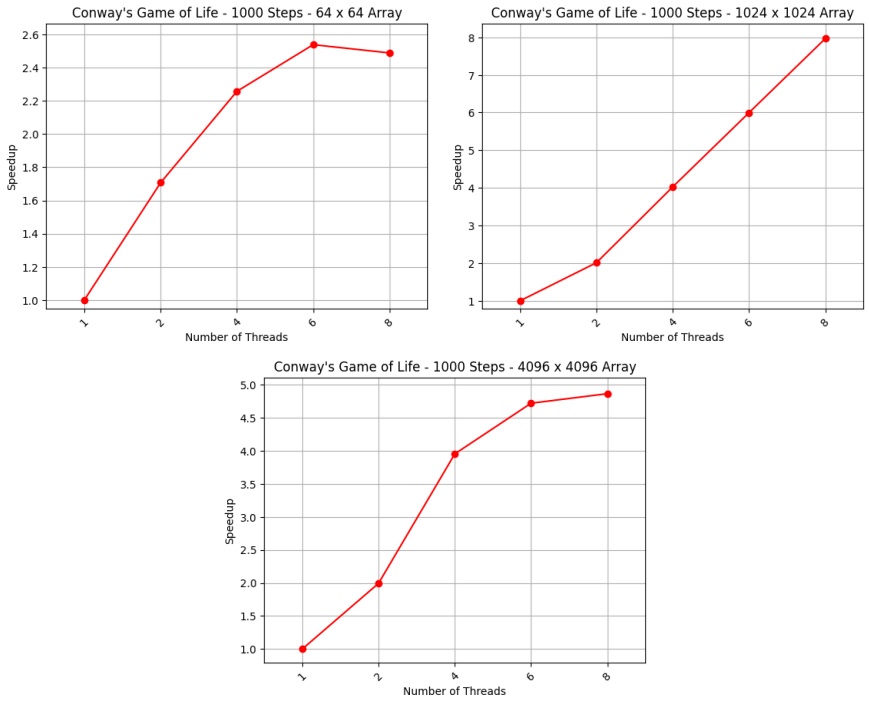


Figure 3: Speedup Game of Life

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 διαμοιρασμού (cache invalidation e.t.c.). Στο τέλος, κάθε νήμα ενημερώνει το master thread με τα δικα του αποτελέσματα (reduction).

2.1.1 K-means - Shared Clusters

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

Τα κλειδώματα 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];
}
```

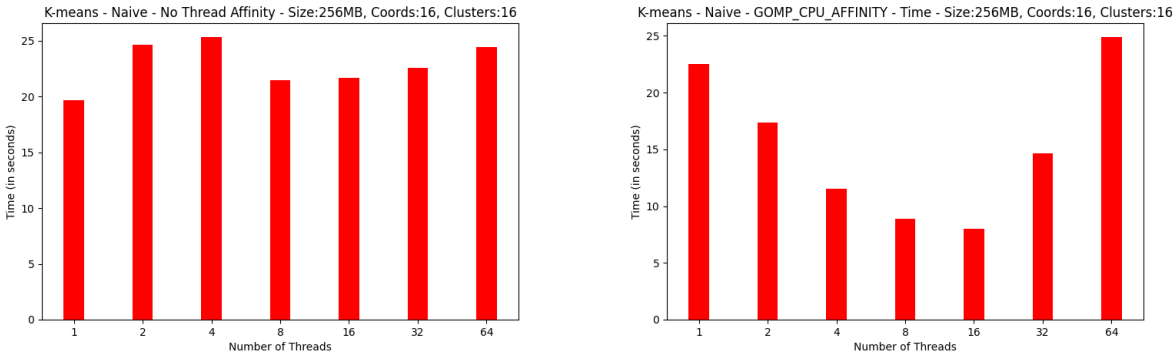


Figure 4: Χρόνος Εκτέλεσης K-Means Naive

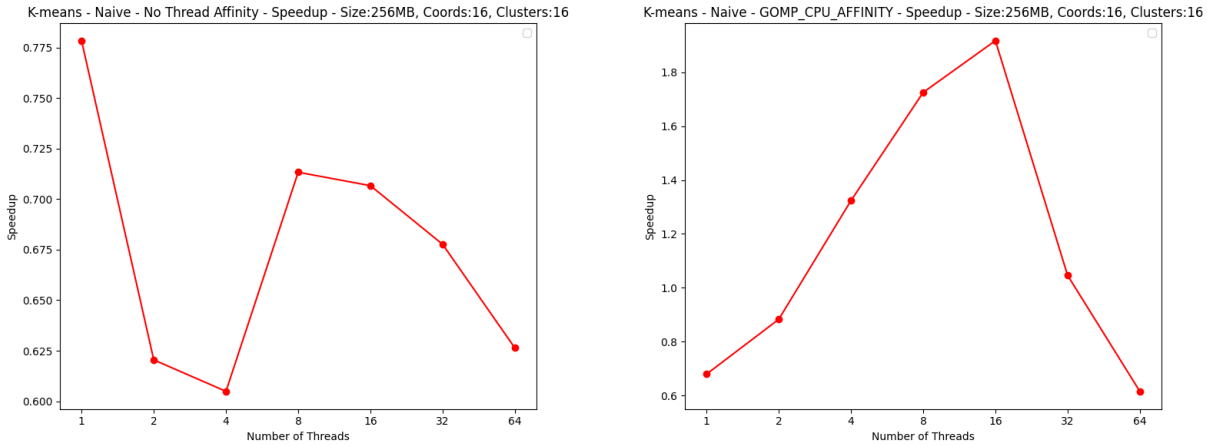


Figure 5: Speedup K-Means Naive

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 μεταβλήτης 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 τακτική 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.

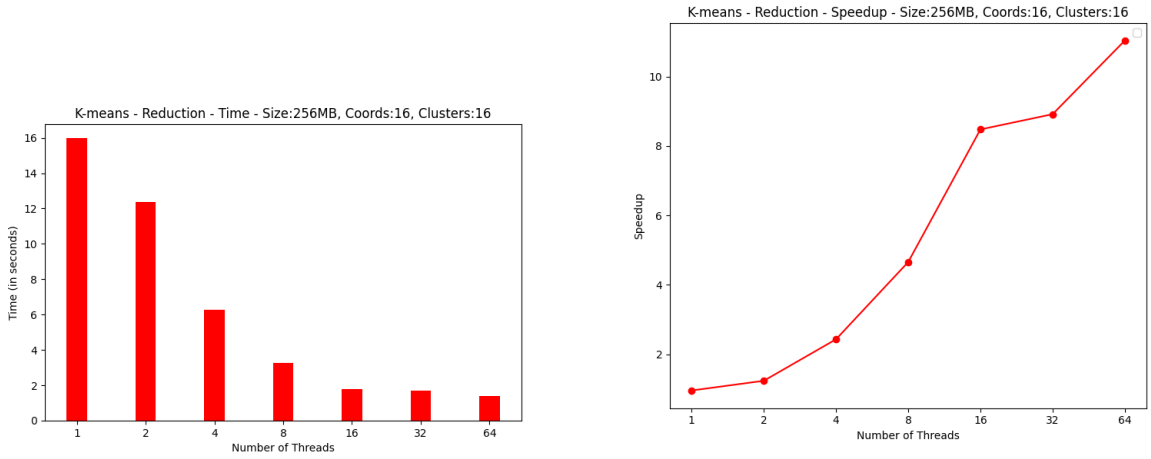


Figure 6: Γραφήματα K-Means Reduction - Simple

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

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

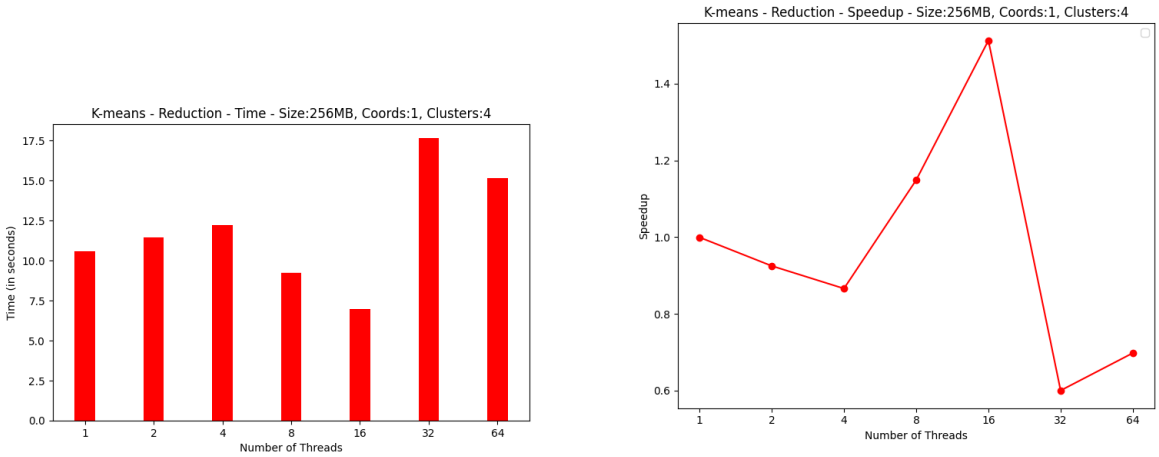


Figure 7: Γραφήματα K-Means Reduction - Hard

Το configuartion αυτό έχει πολλά περισσότερα δεδομένα και μεγαλύτερο πίνακα objects, ο οποίος διαβάζεται από όλα τα νήματα. Αν 2 ή παραπάνω νήματα επεξεργζάονται **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 γίνόταν 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));
}
```

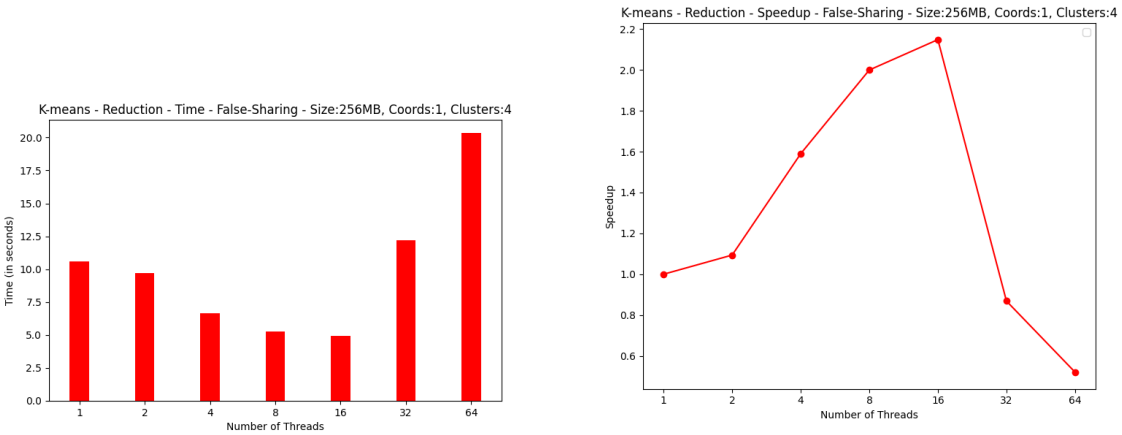


Figure 8: Γραφήματα K-Means Reduction - Hard - False Sharing Aware

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.

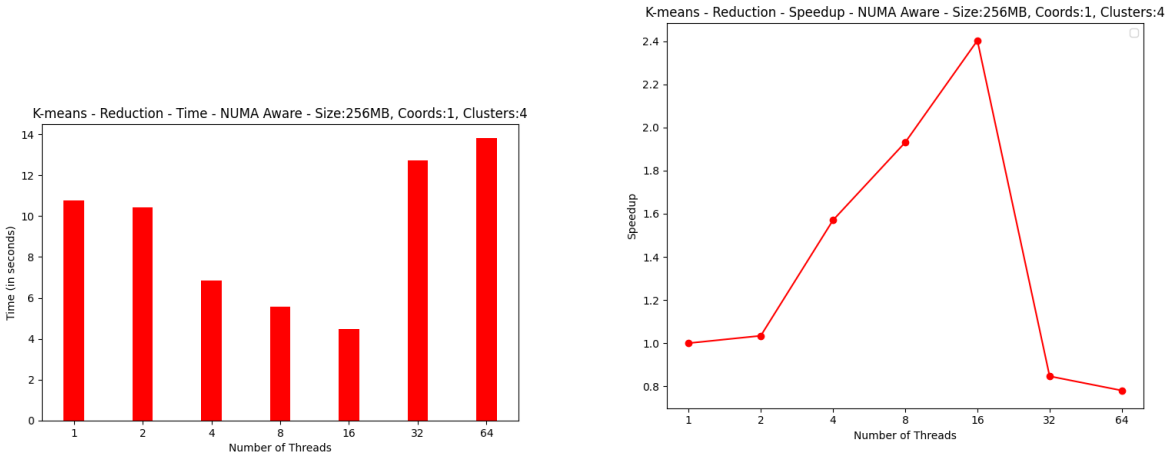
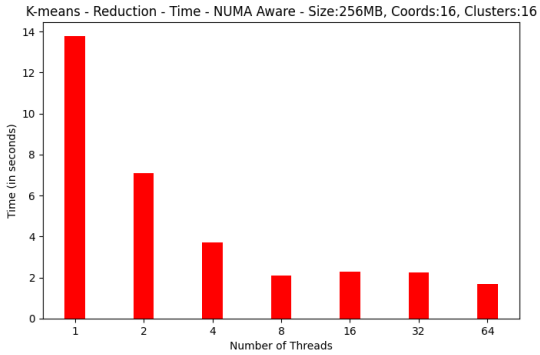


Figure 9: Γραφήματα K-Means Reduction - Hard - NUMA Aware



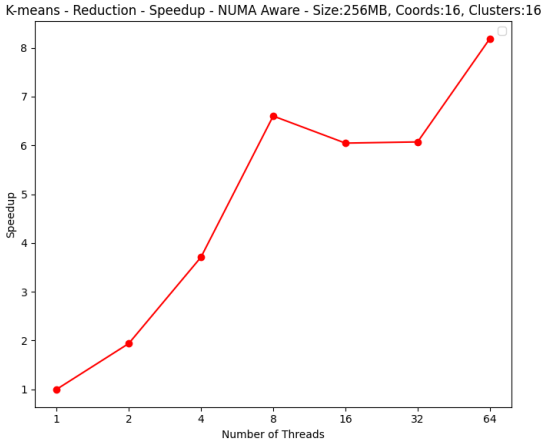


Figure 10: Γραφήματα K-Means Reduction - NUMA Aware

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]);
```

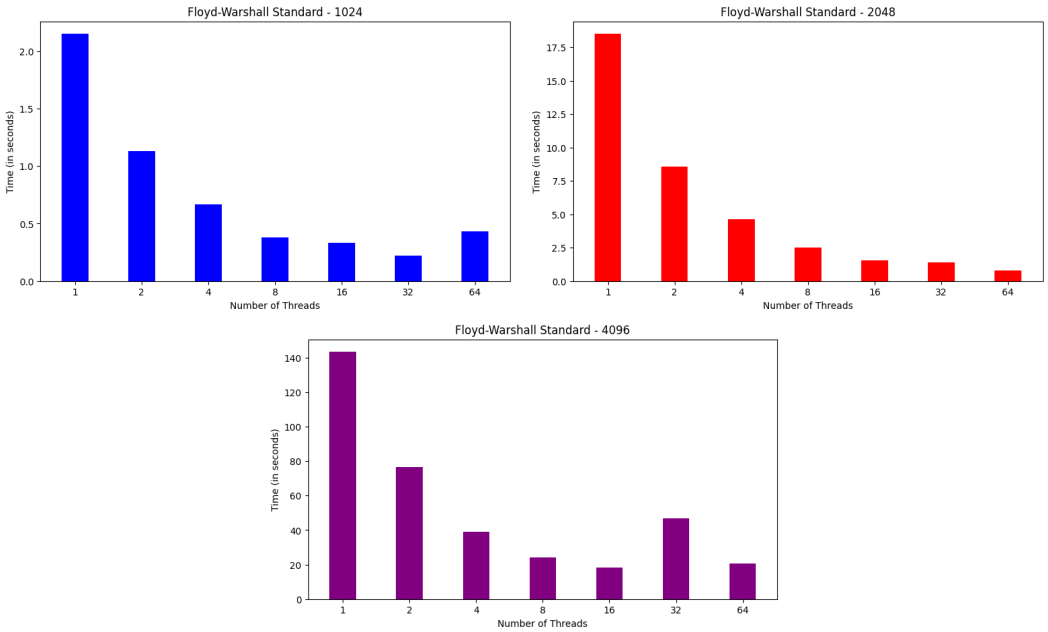


Figure 11: Χρόνος Εκτέλεσης FW Standard

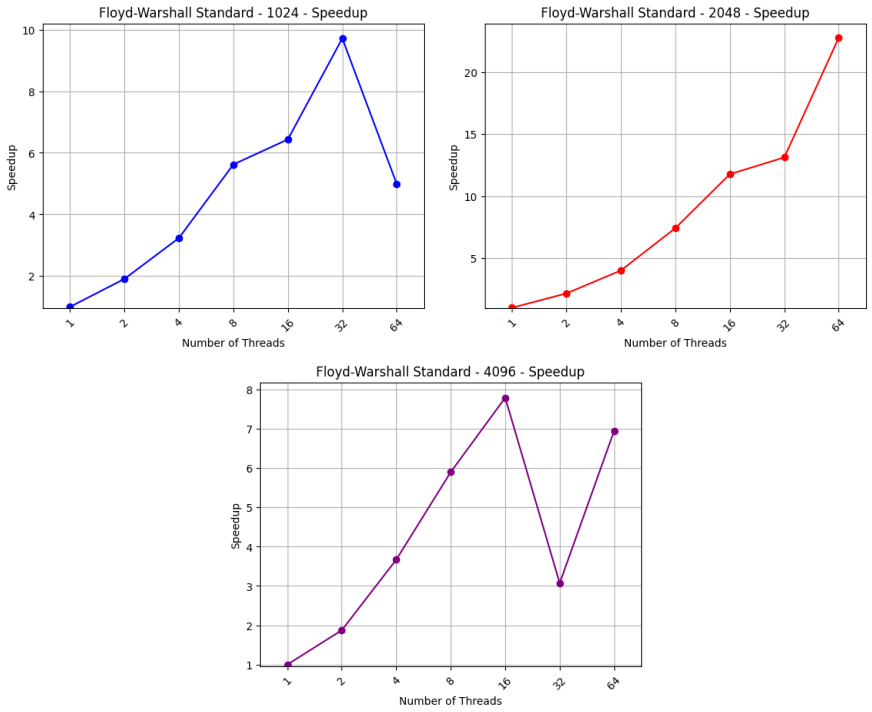


Figure 12: Speedup FW Standard

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 πέρι κλιμάκωσης είναι ολόιδια με το πρόβλημα του Game Of Life. Για μικρούς πίνακες, υπάρχει μεγάλη δυνατότητα για κλιμάκωση, αλλά αυξάνοτας 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 εμας, 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);
```

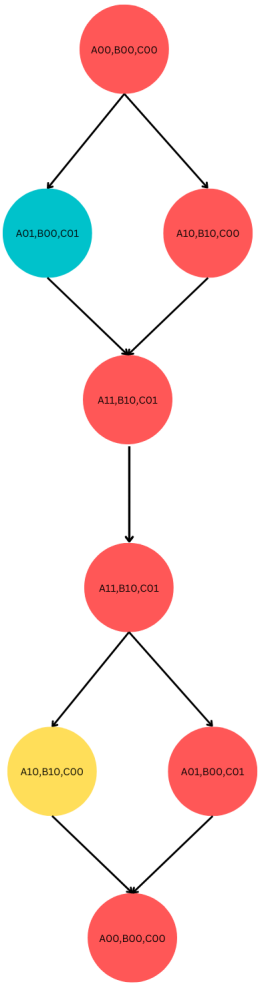
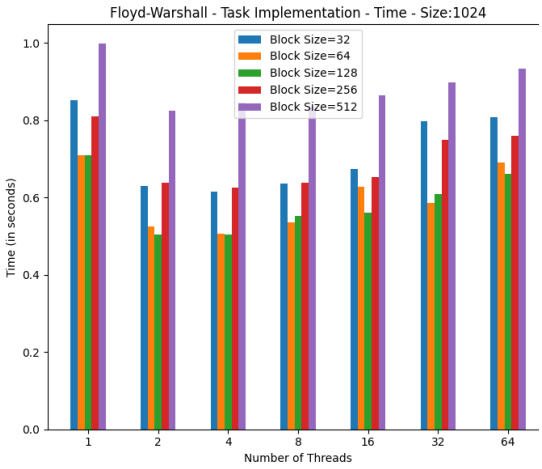
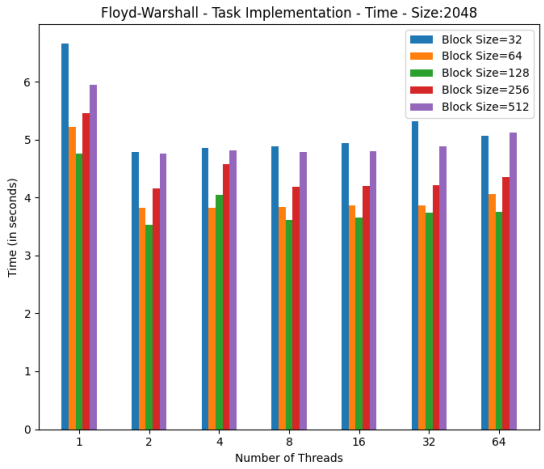


Figure 13: FW Recursive Task Graph - 1st iteration

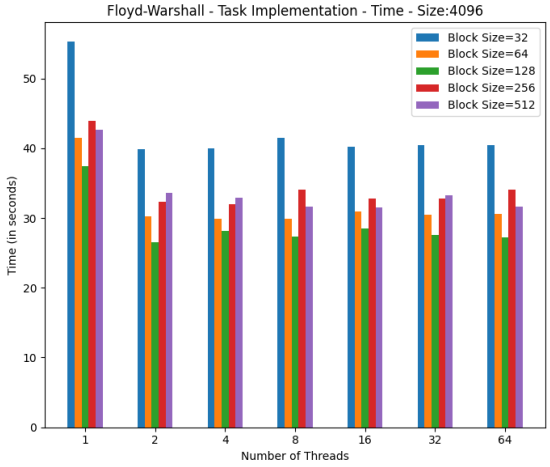
Η παραλληλοποίηση αυτής της έκδοσης γίνεται με χρήση task, ώστε να μπορεί να είναι αναδρομική.



(a) FW Recursive Size 1024

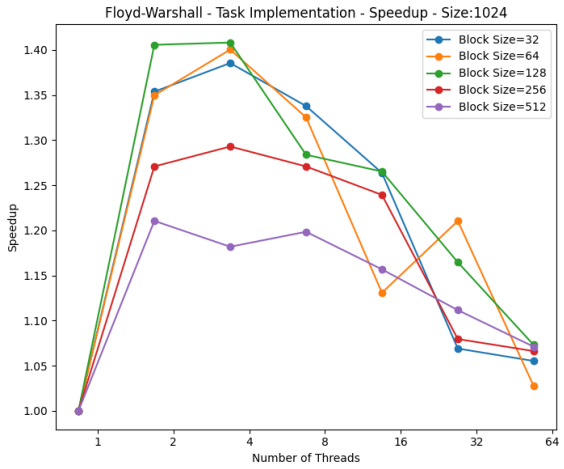


(b) FW Recursive Size 2048

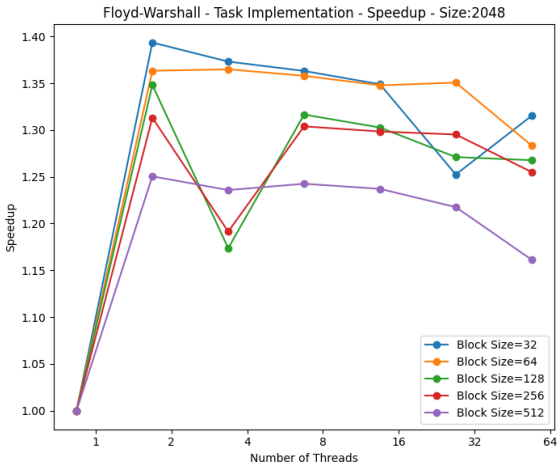


(c) FW Recursive Size 4096

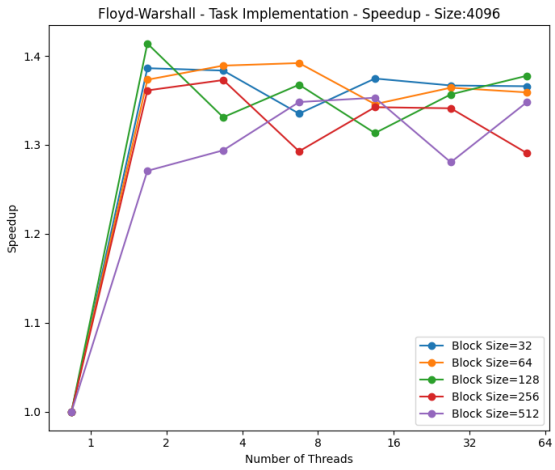
Figure 14: Χρόνος Εκτέλεσης FW Recursive



(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 παραλληλοποίησή 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.

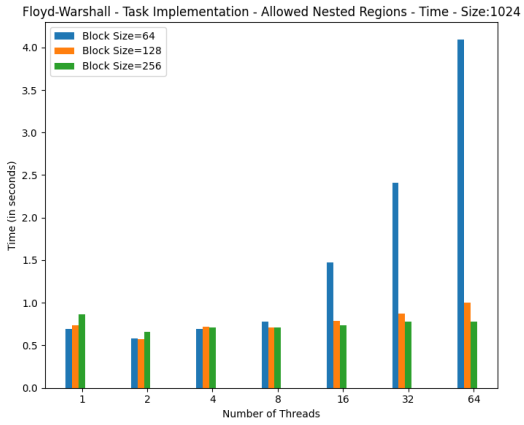
Η λύση για το παραπάνω είναι να αλλάξουμε την συμπεριφορά του OpenMP. Γίνεται να προσπεράσουμε αυτόν τον περιορισμό θέτοντας ένα environmental variable, το **OMP_NESTED** σε **TRUE**.

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.

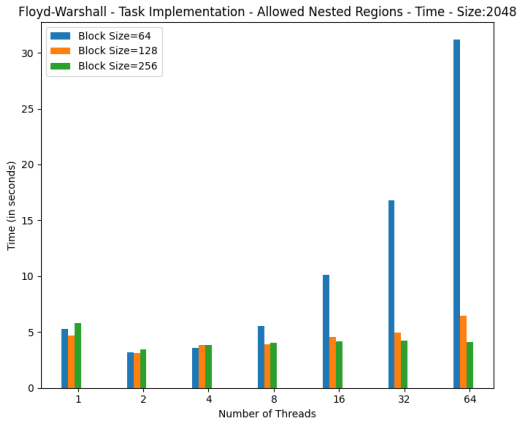
Χρειάζονται περαιτέρω περιορισμοί. Αυτοί δίνονται από το OpenMP με την μορφή μεταβλητών περιβάλλοντος.

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.

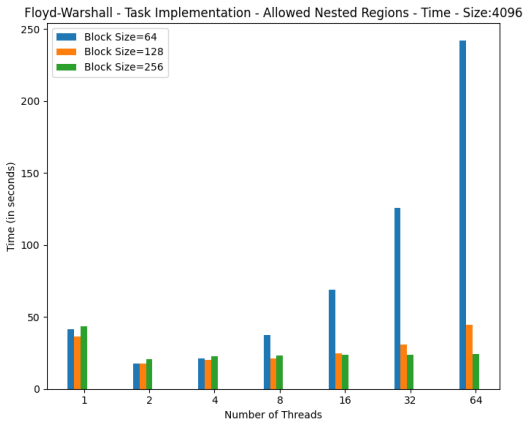
Εφαρμόζοντας τα παραπάνω, βγάζουμε τα παρακάτω αποτελέσματα:



(a) FW Recursive + Nested Size 1024

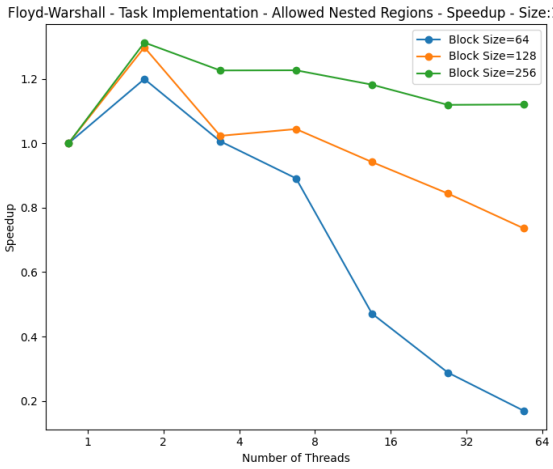


(b) FW Recursive + Nested Size 2048

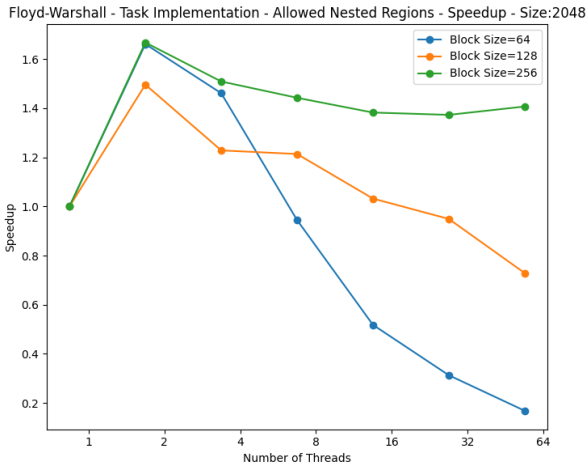


(c) FW Recursive + Nested Size 4096

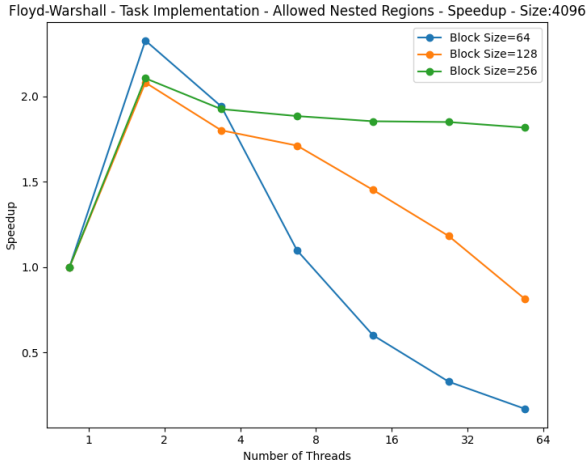
Figure 16: Χρόνος Εκτέλεσης FW Recursive/Nested



(a) FW Recursive - Size 1024 - Speedup



(b) FW Recursive - Size 2048 - Speedup



(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.

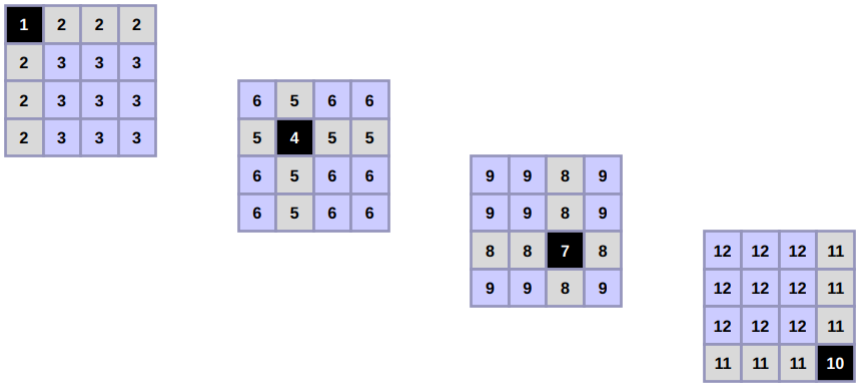
Η κατάλληλη επιλογή block size εξαρτάται κάθε φορά από τον επεξεργαστή που χρησι-

μοποιούμε. 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 Παραλληλισμού FW Tiled



Σε κάθε βήμα, υπολογίζουμε πρώτα το διαγώνιο στοιχείο του πίνακα (k-οστο), 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 παραλληλισμό βρόγχων και μια με παραλληλισμό βρόγχων και tasks.

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

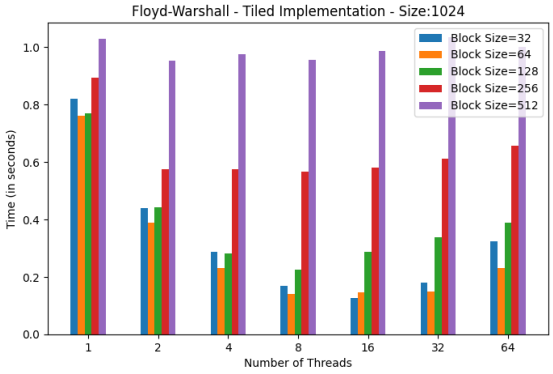
Κώδικας FW Tiled

```
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);
```

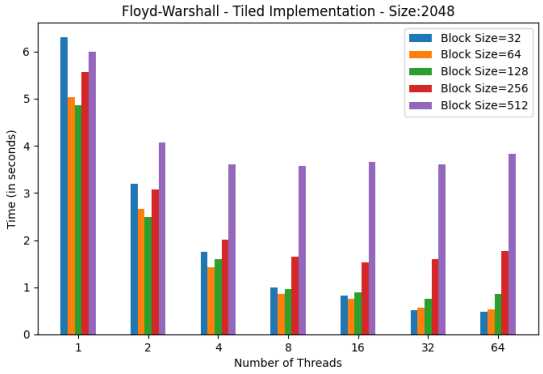
```
      #pragma omp for nowait
      for(j=0; j<k; j+=B)
         FW(A,k,k,j,B);

      #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.

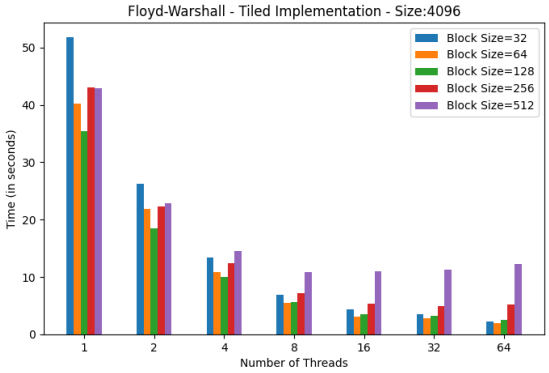
Η επιλογή collapse(2) χρησιμοποιείται καθώς χωρίς αυτήν, ο 2ος βρόγχος θα έτρεχε μονονηματικά, από το νήμα που εκτελεί το for loop. Με το συγκεκριμένο, μπορούν να χρησιμοποιηθούν όλα τα νήματα του συστήματος για να παραλληλοποιηθεί ο τέλεια φωλιασμένος βρόγχος.



(a) FW Tiled - Size 1024

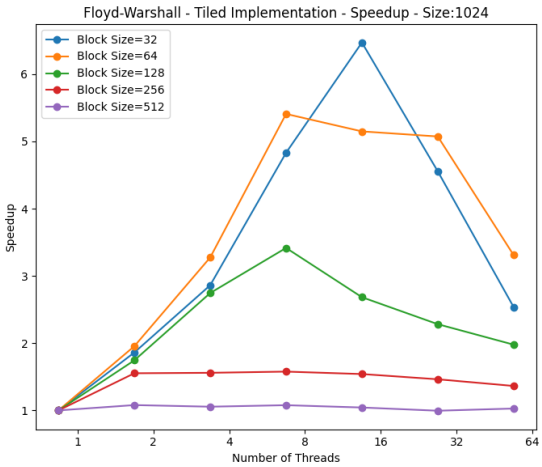


(b) FW Tiled - Size 2048

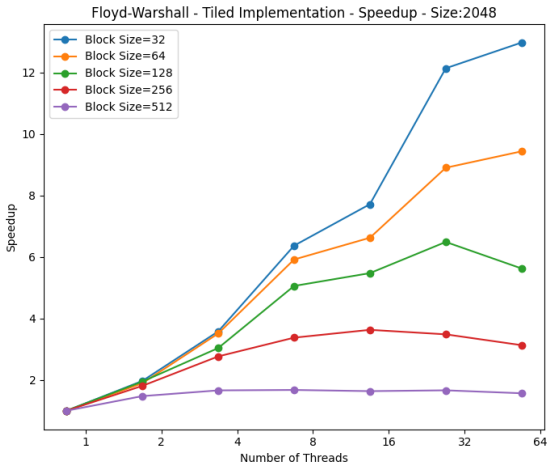


(c) FW Tiled - Size 4096

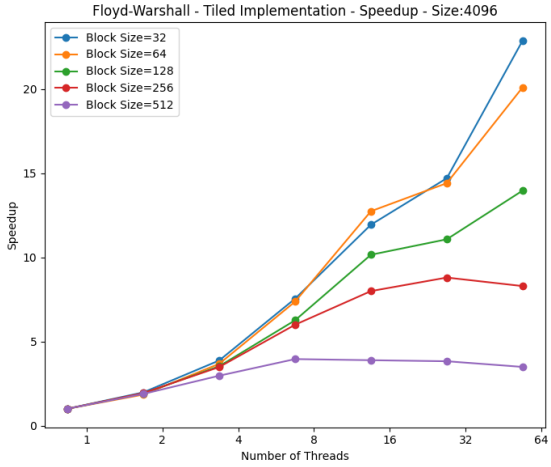
Figure 19: Χρόνος Εκτέλεσης FW Tiled



(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 έκοδση 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.

3 Άσκηση 3 - Αμοιβαίος Αποκλεισμός/Κλειδώματα

Σε αυτή την άσκηση, επιστρέφουμε πίσω στην ανάλυση της παραλληλοποίησης του αλγορίθμου K-means. Στην naive έκδοση, υπάρχει το ζήτημα της ανανέωσης των μεταβλητών newClusters και newClustersSize, οι οποίες δείχνουν τις μεταβολές των κέντρων των Cluster και το μέγεθος αυτών. Εφόσον λειτουργούμε με διαμοιραζόμενα δεδομένα, η ανανέωση αυτών των μεταβλητών πρέπει να γίνει ατομικά, δηλαδή από κάθε νήμα ξεχωριστά.

Το κομμάτι κώδικα στο οποίο γίνεται αυτή η ανανέωση είναι το critical section του προβλήματος.

K-Means Naive - Critical Section

```
// update new cluster centers : sum of objects located within
lock_acquire(lock);
newClusterSize[index]++;
for (j=0; j<numCoords; j++){
    newClusters[index*numCoords + j] += objects[i*numCoords + j];
}
lock_release(lock);
```

Η δημιουργία αυτού του χώρου mutual exclusion μπορεί να γίνει είτε με χρήση locks ή με χρήση των pragma του OpenMP **#pragma omp atomic/critical**

3.1 Mutual Exclusion με χρήση κλειδαριών

Στην ανάλυση που θα κάνουμε, πρέπει να πάρουμε υπόψιν τα NUMA χαρακτηριστικά της αρχιτεκτονικής που χρησιμοποιούμε. Όλα τα κλειδώματα που θα αναλύσουμε έχουν μειονέκτημα ότι μοιράζονται το entity του lock. Η ανανέωση της τιμής του lock θα προκαλέσει cache invalidation και θα εφαρμοστεί το MESI πρωτόκολλο για να υπάρξει συνάφεια μνήμης. Εφόσον έχουμε και NUMA χαρακτηριστικά αλλά και cache coherence, την αρχιτεκτονική αυτή την αποκαλούμε cc-NUMA.

Τα locks που θα εξετάσουμε είναι τα εξής:

Pthread - Mutex

Pthread - Spinlock

Test-and-Set (TAS)

Test-Test-and-Set (TTAS)

Array Based Lock

CLH Lock

Παρουσιάζουμε και την εκδοχή χωρίς κλειδώματα για λόγους σύγκρισης.

3.1.1 Εκτέλεση χωρίς κλείδωμα

Η εκτέλεση χωρίς κλειδώματα θα προσφέρει λάθος αποτελέσματα καθώς κανείς δεν μας καθιστά σίγουρο ότι δεν μπορούν 2 νήματα ή παραπάνω να επεξεργαστούν την ίδια μεταβλητή και κάποιο να "ακυρώσει" την δράση ενός άλλου νήματος.

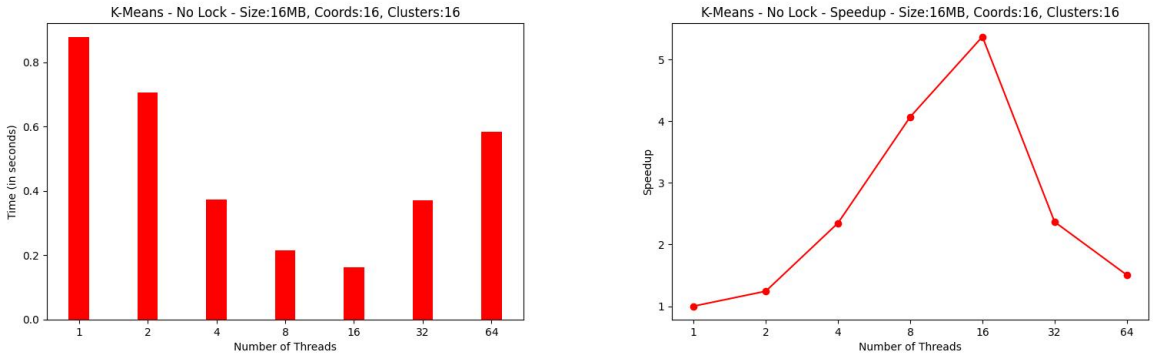


Figure 21: K-Means Naive - No Sync

Περιμένουμε ο βέλτιστος χρόνος εκτέλεσης να είναι από αυτή την εκδοχή αφού δεν υπάρχει το overhead του locking.

Best Time - No Sync

0.1635s - 16 Threads

3.1.2 PThread - Mutex

Το mutex είναι από τις πιο απλές μορφές mutual exclusion lock. Το συγκεκριμένο lock έχει μόνο 2 καταστάσεις, locked και unlocked. Μόνο ένα νήμα επιτρέπεται να κατέχει το lock κάθε στιγμή. Αν ένα νήμα προσπαθήσει να αποκτήσει ένα ήδη κλειδωμένο lock, το νήμα θα γίνει blocked μέχρι να απελευθερωθεί το lock.

Όταν απελευθερωθεί, ένα από τα νήματα που περιμένουν θα επιλεχθεί να μπει στο κρίσιμο τμήμα του προγράμματος.

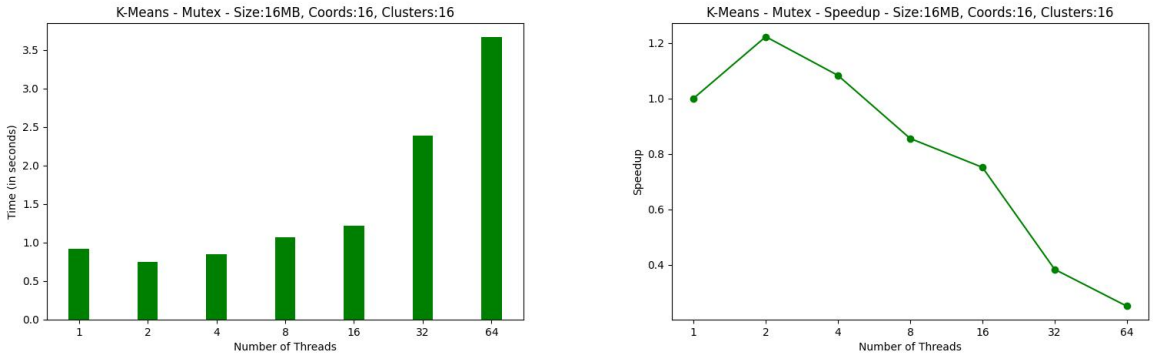


Figure 22: K-Means Naive - PThread - Mutex

Φαίνεται ότι ο χρόνος εκτέλεσης επηρεάζεται πολύ από την χρήση του κλειδώματος. Το overhead που προσθέτει το συνεχόμενο κλείδωμα/ξεκλείδωμα μαζί με τα συνεχόμενα MESI invalidations καθιστούν το mutex lock μια κακή επιλογή όταν έχουμε σύστημα με πολλούς πυρήνες, πολλά MESI nodes και συχνά κλειδώματα.

Άλλος συντελεστής που επηρεάζει την απόδοση των mutex είναι ότι απαιτείται context switch για τις λειτουργίες του. Η χρησιμότητα τους φαίνεται σε άλλα προγράμματα με μεγαλύτερα critical sections. Επίσης, είναι επιτρεπτό ένα νήμα να πέσει για ύπνο όσο περιμένει το lock, δεν θα προκαλέσει deadlock.

Best Time - PThread - Mutex

0.7506s - 2 Threads

3.1.3 PThread - Spinlock

Τα spinlock διαφέρουν αρκετά από τα mutexes. Αντί να γίνει blocked το νήμα που προσπαθεί να πάρει το ήδη κλειδωμένο lock, θα μπεί σε busy-wait loop μέχρι να απελευθερωθεί το lock. Αυτό σημαίνει ότι δεν γίνεται context switch στην λειτουργία τους. Η υλοποίηση της βιβλιοθήκης pthread χρησιμοποιεί είτε εντολή atomic_exchange (αν υποστηρίζεται από το υλικό) ή μια weak εντολή compare-and-swap (αν δεν υποστηρίζεται το atomic_exchange).

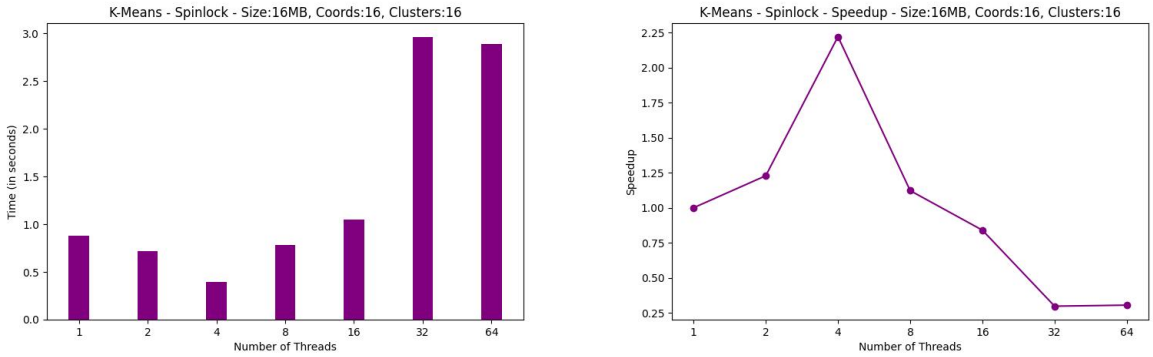


Figure 23: K-Means Naive - PThread - Spinlock

Παρατηρούμε πολύ καλύτερη απόδοση συγκριτικά με το κλείδωμα με mutex. Η απουσία του context switching για το κλείδωμα και ξεκλείδωμα και το γεγονός ότι το κρίσιμο μονοπάτι του συγκεκριμένου προβλήματος είναι μικρό ευνοεί σημαντικά τα spinlock.

Για απλά κλειδώματα στα οποία τα νήματα δεν χρειάζεται να περιμένουν πολύ ώρα για να λάβουν το lock, τα spinlock είναι μια καλή επιλογή.

Best Time - PThread - Spinlock

0.3960s - 4 Threads

3.1.4 Test-and-Set (TAS)

Τα κλειδώματα Test-and-Set ακολουθούν την εξής λογική:

- Έστω η μέθοδος "Test", η οποία ελέγχει αν το lock έχει παρθεί από κάποιο thread. Αν η κλειδαριά είναι ελεύθερη, η "Test" επιστρέφει False. Αν όχι, επιστρέφει True.

- Το νήμα που καλεί την "Test", όχι μόνο διαβάζει την τιμή του lock, αλλά αννανεώνει κιόλας το lock με την τιμή που μόλις διάβασε

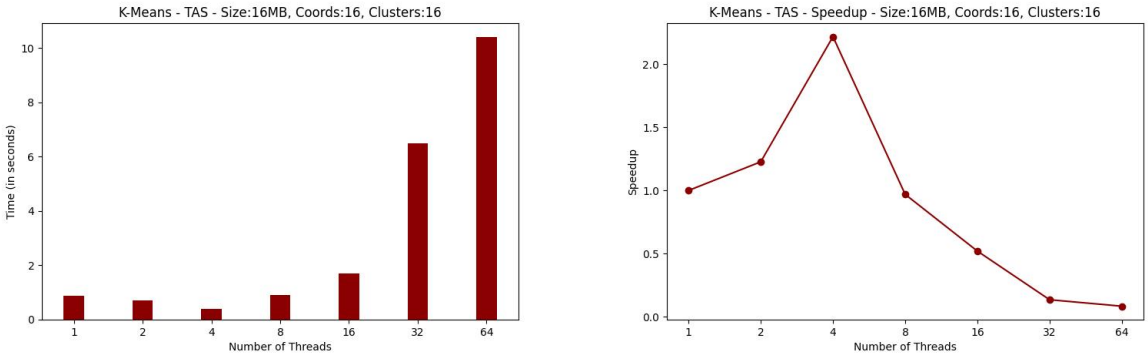


Figure 24: K-Means Naive - TAS

Για μικρό αριθμό νημάτων, παρατηρούμε ότι τα κλειδώματα TAS έχουν επιτρεπτή απόδοση. Το overhead τους είναι πολύ μικρό καθώς απλώς εκτελούν έναν ατομικό έλεγχο και ανανέωση μεταβλητής.

Η κατάσταση αυτή αλλάζει σημαντικά όταν αυξάνουμε τον αριθμό νημάτων. Εδώ, αναφερόμαστε πάλι στα NUMA χαρακτηριστικά των επεξεργαστών που χρησιμοποιούμε, καθώς και στην ανάγκη για cache coherence. Με τα TAS κλειδώματα, εκτελούμε περιττά cache invalidations ανανεώνοντας συνεχώς την τιμή του lock καθώς κάνουμε τον έλεγχο. Έχουμε υπερβολική "κίνηση" στον δίαυλο μνήμης των επεξεργαστών. Επίσης, ειδικά στην περίπτωση που χρησιμοποιούμε 64 νήματα, υπάρχουν και οι ακραίες περιπτώσεις που πρέπει 2 νήματα που απέχουν την μεγαλύτερη δυνατή απόσταση μεταξύ τους να χρειαστέι να κάνουν Modify (και άρα να γίνουν Invalidated τα υπόλοιπα). Αυτή η μεταφορά δεδομένων από τα πιο μακρινά NUMA nodes καταστρέφει την απόδοση.

Τα κλειδώματα TAS έχουν την χειρότερη απόδοση από όλα τα κλειδώματα. Το "σφάλμα" τους είναι ξεκάθαρο και επιλύνεται από το επόμενο κλείδωμα.

Best Time - Test-and-Set

0.3960s - 4 Threads

3.1.5 Test-Test-and-Set (TTAS)

Σε αντίθεση με το TAS κλείδωμα, το TTAS κλείδωμα **ΔΕΝ** αλλάζει την τιμή του lock αν το βρει κλειδωμένο, αλλά το αφήνει **LOCKED**. Γίνεται συνεχόμενα έλεγχος για αν το lock είναι ελεύθερο από κάποιο νήμα αλλά αυτό δεν σημαίνει ότι με το που αλλάξει η τιμή σε **UNLOCKED** το νήμα αυτό θα πάρει το lock. Για αυτό, χρειάζεται να τρέξουμε άλλη μια φορά

την μέθοδο getAndSet(true) (δηλαδή την "μέθοδο" "Test") για να πάρει το lock. Αν αποτύχει η συγκεκριμένη μέθοδος και βρει πάλι κλειδωμένο το lock, σημαίνει ότι κάποιο άλλο νήμα πρόλαβε να παραλάβει την κλειδαριά, και το νήμα που προσπάθησε ξανά τρέχει συνεχόμενους ελέγχους.

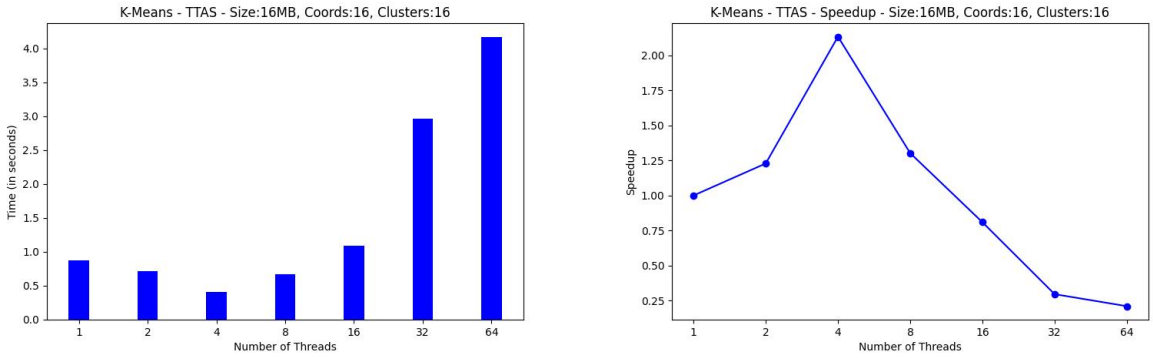


Figure 25: K-Means Naive - TTAS

Το overhead των TTAS κλειδωμάτων είναι ελάχιστα μεγαλύτερο από τα TAS κλειδώματα και άρα σε μικρό αριθμό νημάτων, τα TTAS είναι ελάχιστα χειρότερα από τα TAS.

Χρησιμοποιώντας περισσότερα νήματα, παρατηρούμε ότι η απόδοση είναι 2 φορές καλύτερη από τα TAS κλειδώματα. Ακόμα όμως και με αυτή την αλλαγή, η επίδοση των TTAS κλειδωμάτων, δεν είναι και η καλύτερη.

Όταν πολλά νήματα "παλέυουν" για την απόκτηση μιας κλειδαριάς, λέμε ότι υπάρχει μεγάλος ανταγωνισμός. Αν συχνά ένα νήμα αποτυγχάνει να αποκτήσει το lock έχοντας όμως ήδη περάσει το πρώτο "Test" στάδιο, πρέπει για λίγο χρόνο να σταματήσει να προσπαθεί να αποκτησει το lock γιατί έτσι μειώνεται η κινητικότητα στην δίαυλο μνήμης. Την λογική αυτή ακολουθούν τα επόμενα κλειδώματα.

Best Time - Test-Test-and-Set

0.4121s - 4 Threads

3.1.6 Array Based Lock

Το συγκεκριμένο κλείδωμα υλοποιείται ως ένα queue. Τα νήματα μοιράζονται μια μεταβλητή, την ουρά, η οποία δείχνει ποιος κατέχει την κλειδαριά. Για να αποκτήσει ένα νήμα το lock, αυξάνει κατά 1 την μεταβλητή της ουράς. Το μέγεθος αυτό δείχνει το ποιο νήμα έχει τον έλεγχο. Χρησιμοποιώντας αυτόν τον αριθμό ως index, μια boolean τιμή δείχνει αν ένα lock είναι ελεύθερο ή δεσμευμένο. Αν είναι true, τότε επιτρέπεται το νήμα να πάρει το lock.

Τα νήματα κάνουν spin μέχρι το slot τους να γίνει true. Για να απελευθερωθεί το lock, το κάθε νήμα καθιστά την δική τους θέση ως false και κάνουν set την επόμενη θέση σε true.

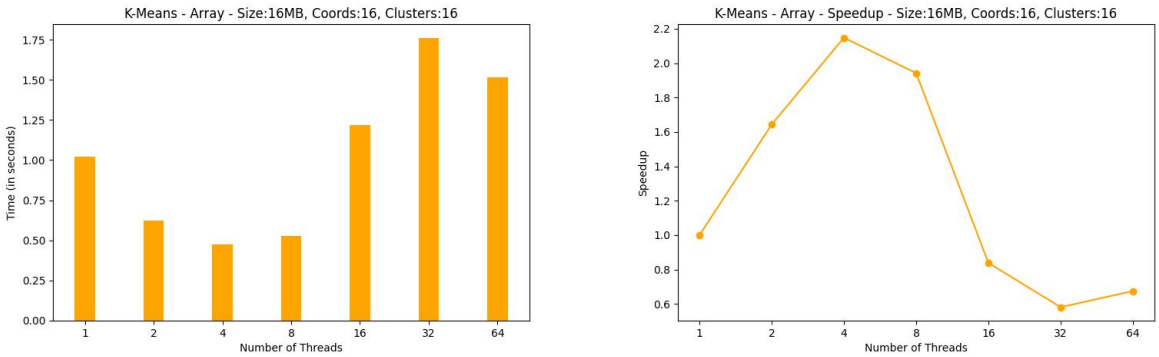


Figure 26: K-Means Naive - Array Based Lock

Ο συγκεκριμένος τύπος κλειδώματος είναι ο πρώτος που επιχειρεί να είναι NUMA/cache coherent aware. Η τιμή που δείχνει την θέση του κάθε νήματος (mySlotIndex) είναι τοπική για κάθε νήμα. Δηλαδή, τα νήματα κάνουν spin σε μια θέση μνήμης που είναι τοπική. Έτσι, δεν προκαλείται coherence traffic χωρίς λόγο. Αν και ο boolean πίνακας είναι διαμοιραζόμενος, τα νήματα δεν "παλεύουν" να αποκτήσουν πρόσβαση σε αυτή την θέση μνήμης καθώς κάνουν spin στις τοπικές θέσεις τους, μειώνοντας έτσι και το invalidation traffic.

Το array lock έχει το μειονέκτημα ότι πρέπει να προσέχουμε για φαινόμενα false sharing. Εφόσον ο boolean πίνακας που δείχνει την κατάσταση του lock είναι διαμοιραζόμενος, θα υπάρχει σχεδόν σίγουρα false sharing. False sharing σημαίνει ότι 2 ή περισσότερα νήματα μοιράζονται ίδια cache lines αλλά επεξεργάζονται διαφορετικές μεταβλητές. Η ευκολότερη λύση είναι να χρησιμοποιήσουμε padding.

Άλλο ένα πλεονέκτημα είναι ότι είναι πολύ δίκαιο lock και δεν παρατηρείται starvation. Λόγω της σχεδίασης του lock ώς ουρά, εκτελείται first-come-first-served σειρά εκτέλεσης.

Παρατηρούμε πολύ καλύτερους χρόνους για αριθμό νημάτων πάνω από 8, τουλάχιστον συγκριτικά με τις προηγούμενες υλοποιήσεις. Πάλι όμως, το πρόβλημα δεν κλιμακώνει για πάνω από 8 νήματα.

Best Time - Array Based Lock

0.4755s - 4 Threads

3.1.7 CLH Lock

Η τελευταία βελτίωση που θα μελετήσουμε είναι για να μειωθεί η ανάγκη για μνήμη που απαιτεί το array lock.

Το CLH lock ακολουθεί την λογική του array lock, δηλαδή ότι τα νήματα κάνουν spin τοπικά. Αντί για ουρά, χρησιμοποιεί μια σχεδίαση linked list. Το κάθε νήμα έχει τοπικά ένα QNode που δείχνει την κατάσταση του. Αν η boolean τιμή locked είναι true, τότε το νήμα είτε έχει έλεγχο του lock ή περιμένει για αυτόν. Αν είναι false, έχει αφήσει το lock. Κάθε νήμα αναφέρεται στο node του προηγούμενου νήματος με μία τοπική μεταβλητή.

Για να γίνει locked ή unlocked η κλειδαριά, χρησιμοποιείτε παρόμοια λογική με το array lock, απλά αυτή την φορά αντί να κάνουμε increment την θέση του πίνακα που κοιτάζει την κατάσταση του νήματος, το κάθε νήμα γίνεται locked από μόνο του και μετά θέτει το node

του ως το tail του queue. Για να απελευθερωθεί το lock, θέτει τον εαυτό του ως unlocked και χρησιμοποιεί το node του προηγούμενου νήματος ως το νέο node του για τα επόμενα κλειδώματα.

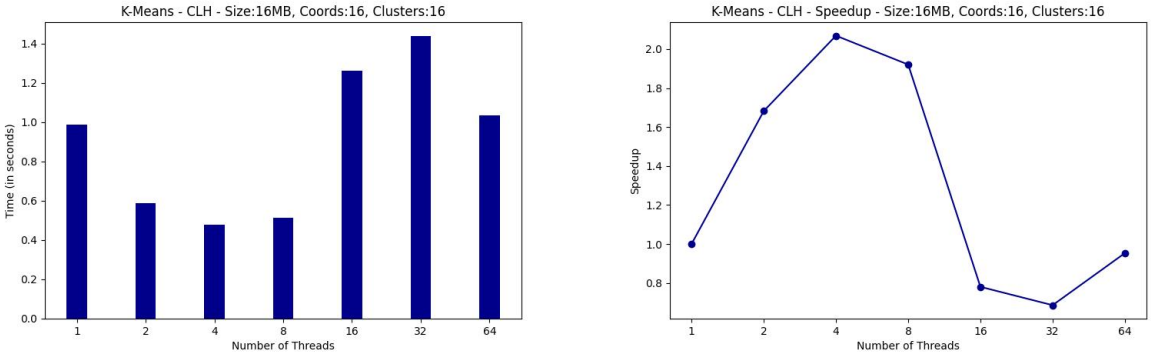


Figure 27: K-Means Naive - CLH Lock

Η προγραμματιστική δυσκολία του CLH Lock είναι ξεκάθαρη, με εμφανή αποτελέσματα όμως. Έχει την πιο ισορροπημένη απόδοση συγκριτικά με όλες τις υλοποιήσεις. Ακόμα και σε μεγάλο αριθμό νημάτων, δεν μειώνεται η απόδοση του σε μεγάλο βαθμό. Πάλι όμως, δεν παρατηρείται κλιμάκωση για πάνω από 8 νήματα.

Best Time - CLH Lock

0.4768s - 4 Threads

All Times

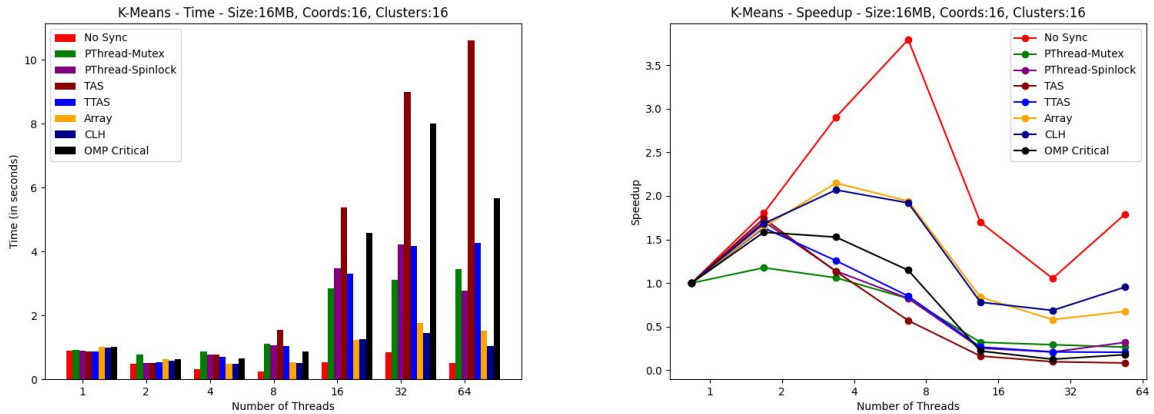


Figure 28: K-Means Naive - All Locks

3.2 Mutual Exclusion με χρήση critical/atomic pragma

Είχα πάει σε πρόγραμμα Erasmus+ και δεν πρόλαβα να κάνω αυτό το ερώτημα καθώς είχε πέσει ο sandman την ώρα που το έστελνα. Θα το παραδώσω μαζί με το επόμενο ερώτημα, χίλια συγγνώμη.