

# 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  $N \times N$ ). 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 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;
```

```

    }

#define 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

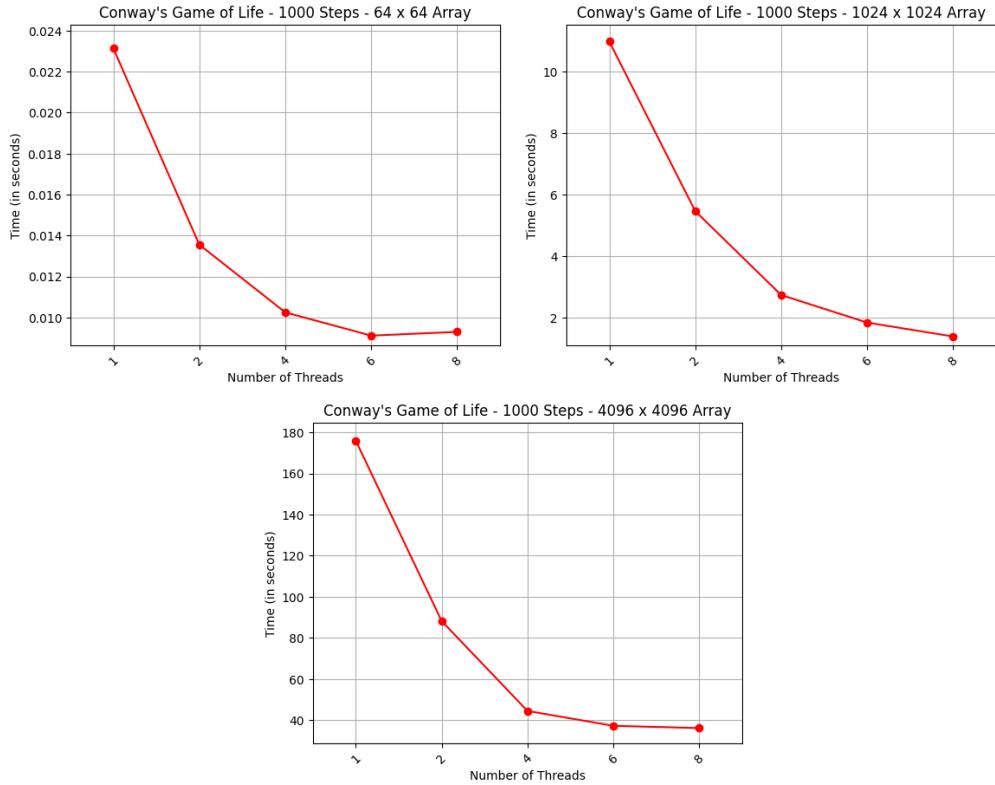


Figure 2: Χρόνος Εκτέλεσης Game of Life

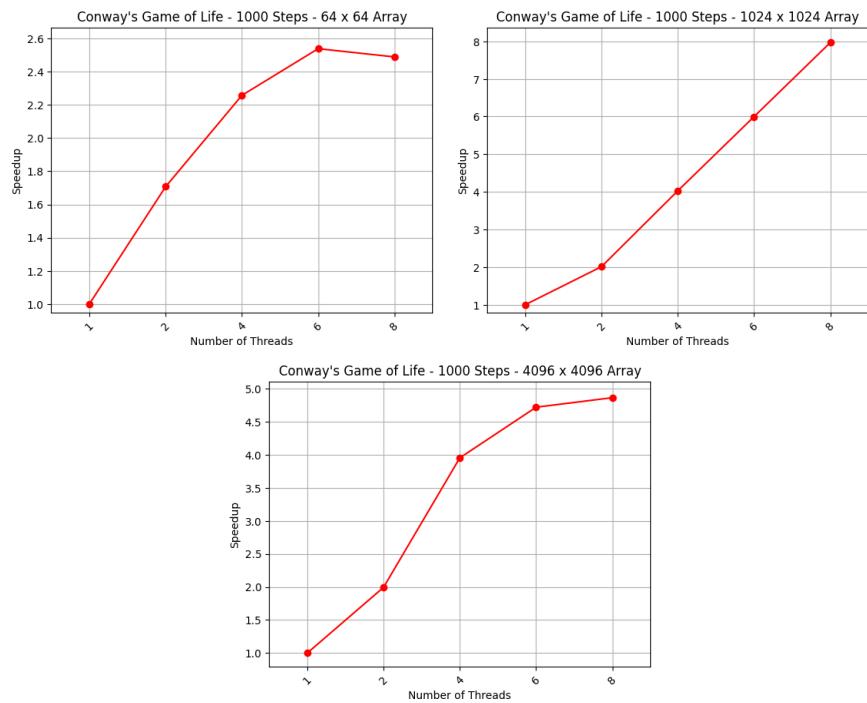


Figure 3: Speedup Game of Life

## 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 δι-αμοιραζόμενα 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];
}
```

---

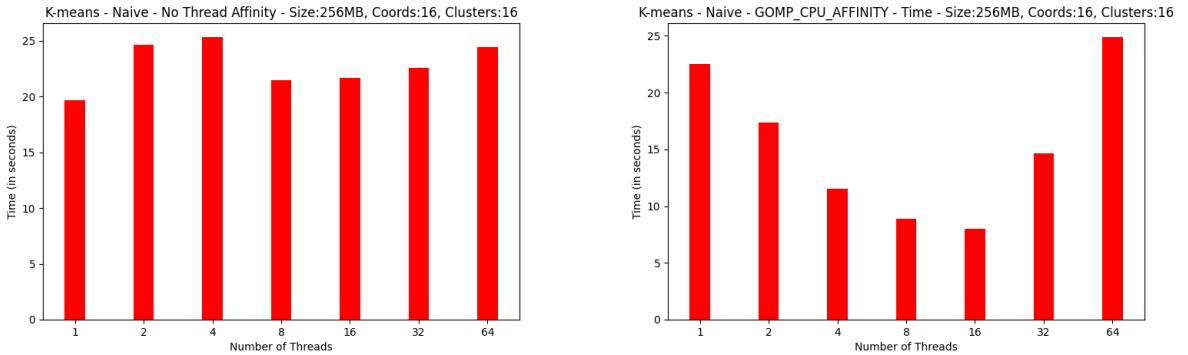


Figure 4: Χρόνος Εκτέλεσης K-Means Naive

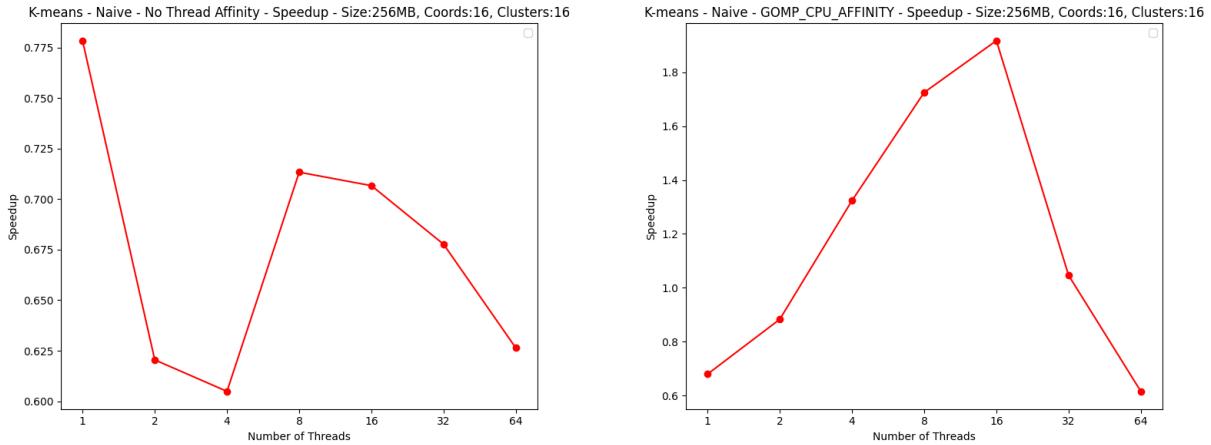


Figure 5: Speedup K-Means Naive

**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 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 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.

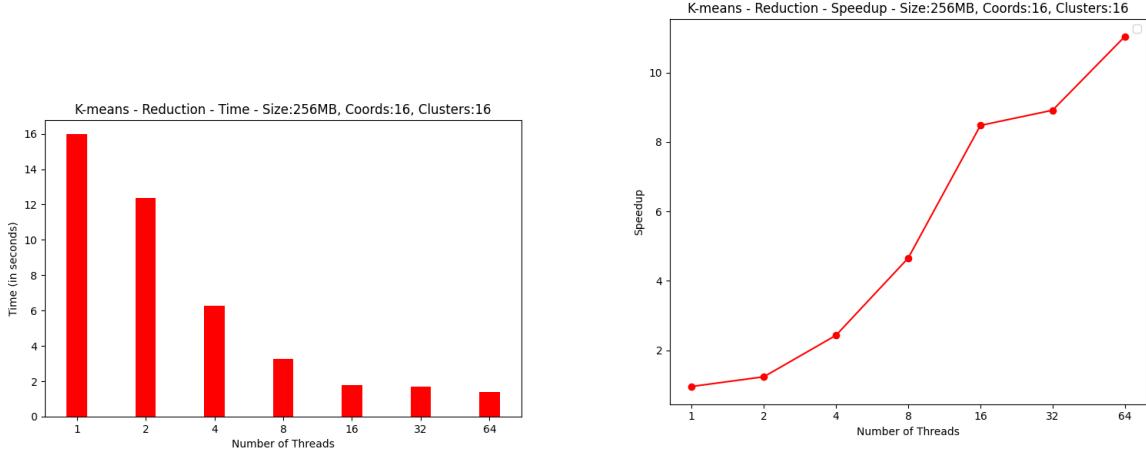


Figure 6: Γραφήματα K-Means Reduction - Simple

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 δουλεύμε 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 configuartion,  $\{\text{Size}, \text{Coords}, \text{Clusters}, \text{Loops}\} = \{256, 1, 4, 10\}$ . Πλέον αναφερόμαστε σε αυτό το configuration ως "Hard".

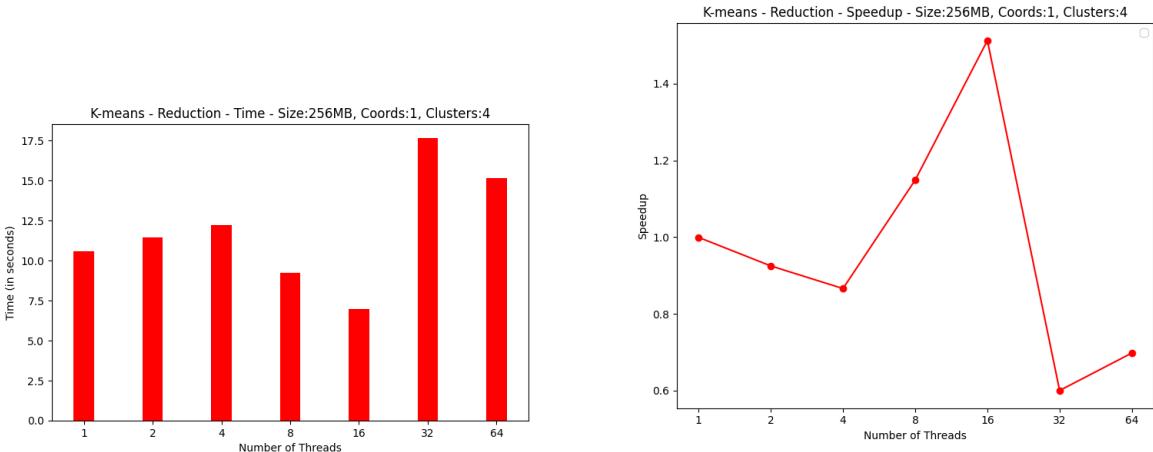


Figure 7: Γραφήματα K-Means Reduction - Hard

To configuartion 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 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));
}
```

---

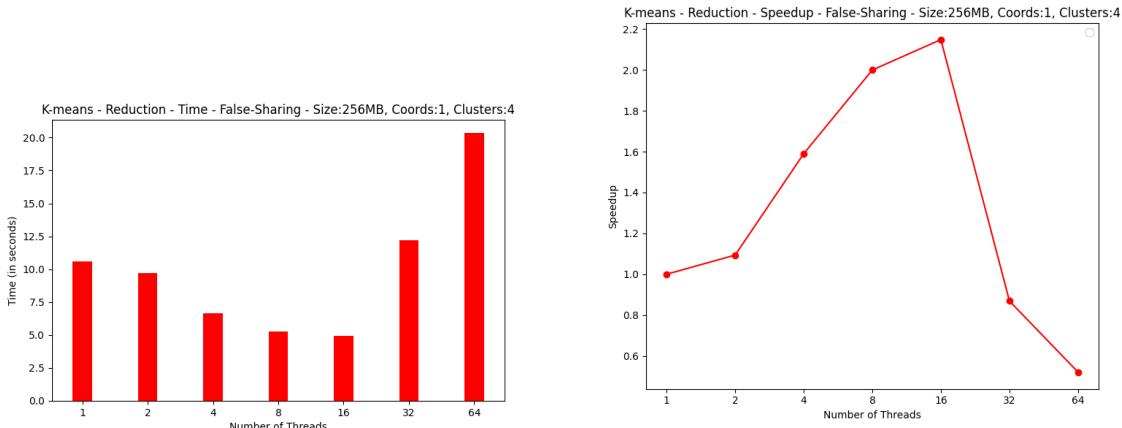


Figure 8: Γραφήματα K-Means Reduction - Hard - False Sharing Aware

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. Συγχριτικά 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 ανάγγωση 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.

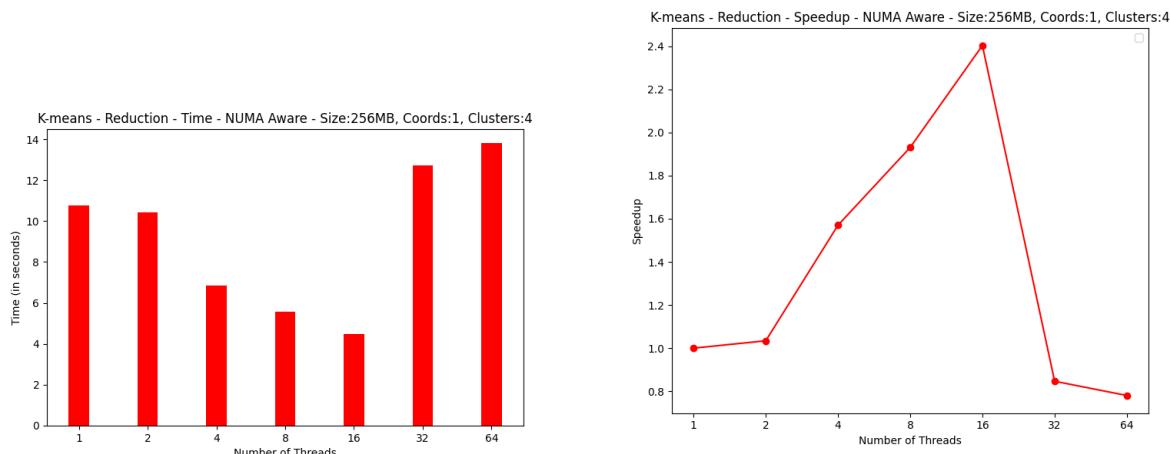


Figure 9: Γραφήματα K-Means Reduction - Hard - NUMA Aware

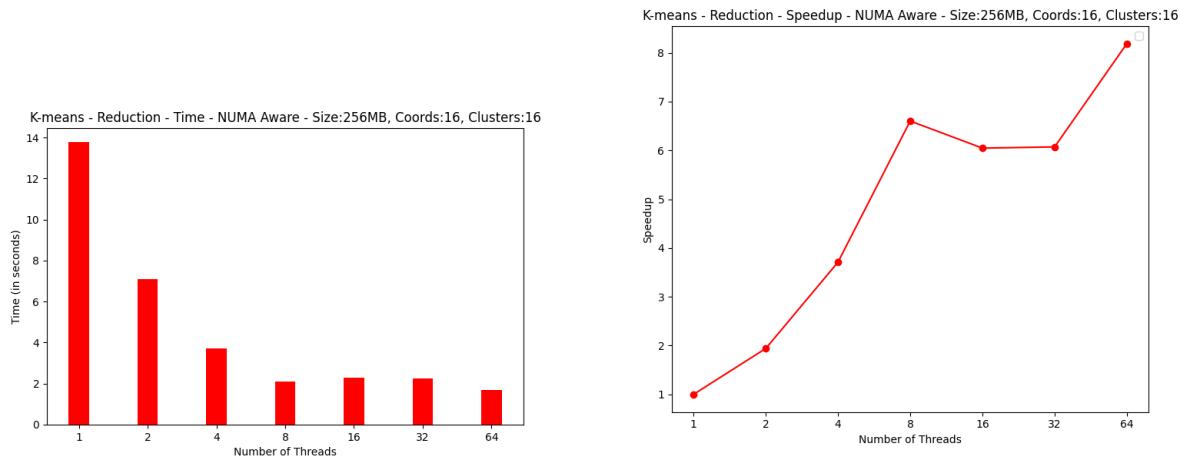


Figure 10: Γραφήματα K-Means Reduction - NUMA Aware

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 αφιθμό νημάτων. Πάλι όμως, βγαίνουμε 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]);

```

---

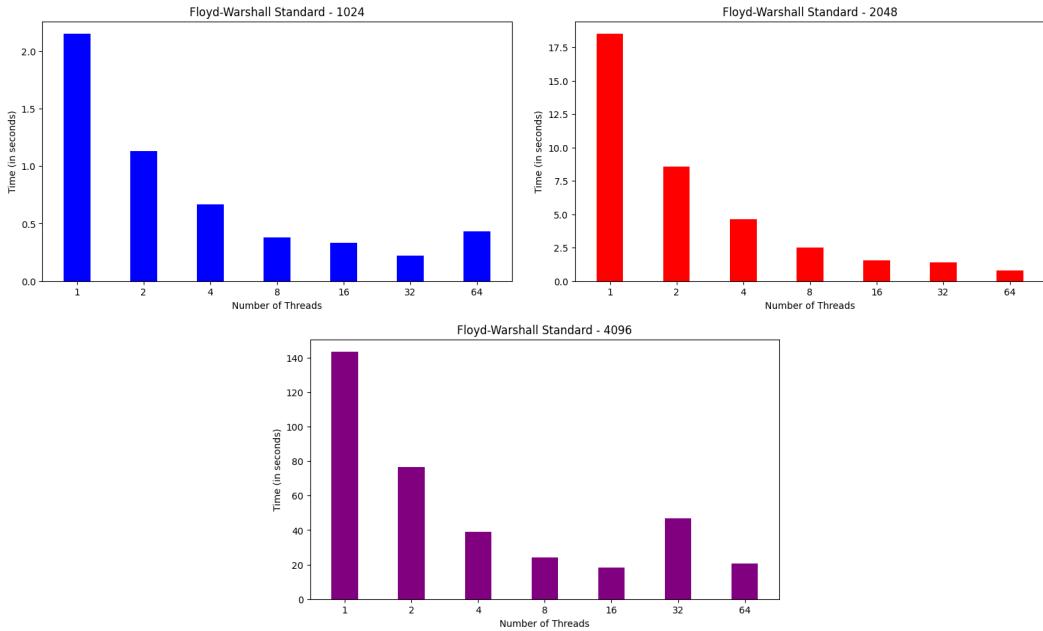


Figure 11: Χρόνος Εκτέλεσης FW Standard

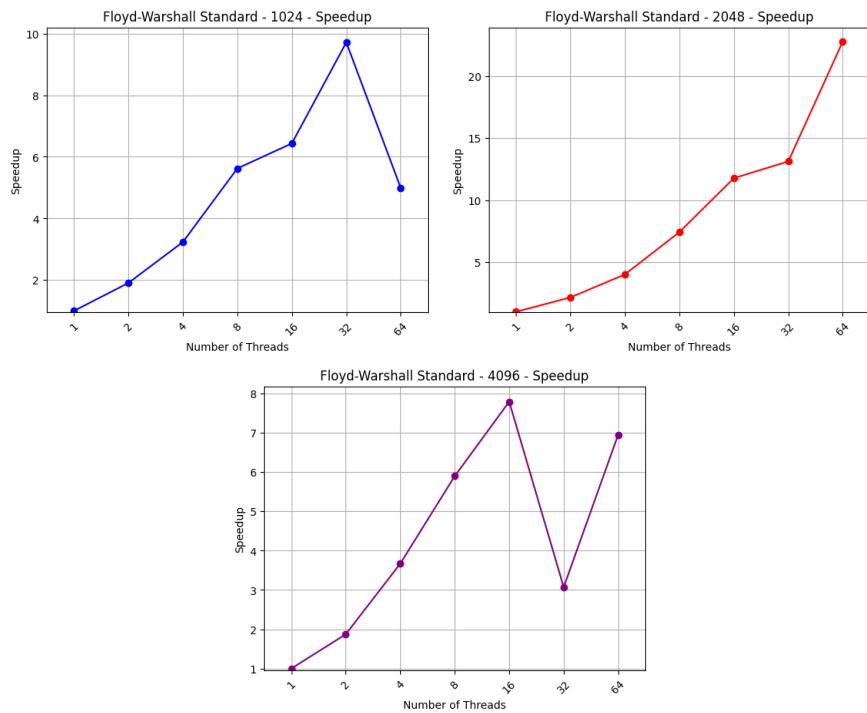


Figure 12: Speedup FW Standard

## 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 ακλιμάκωσης είναι ολόιδια με το πρόβλημα του Game Of Life. Για μικρούς πίνακες, υπάρχει μεγάλη δυνατότητα για ακλιμάκωση, 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 ακλιμάκωση.

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);

```

---

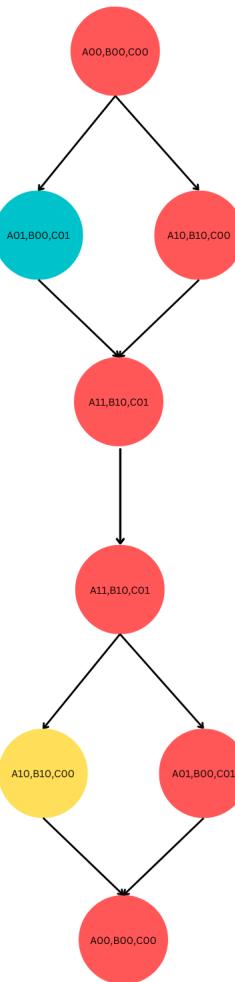


Figure 13: FW Recursive Task Graph - 1st iteration

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

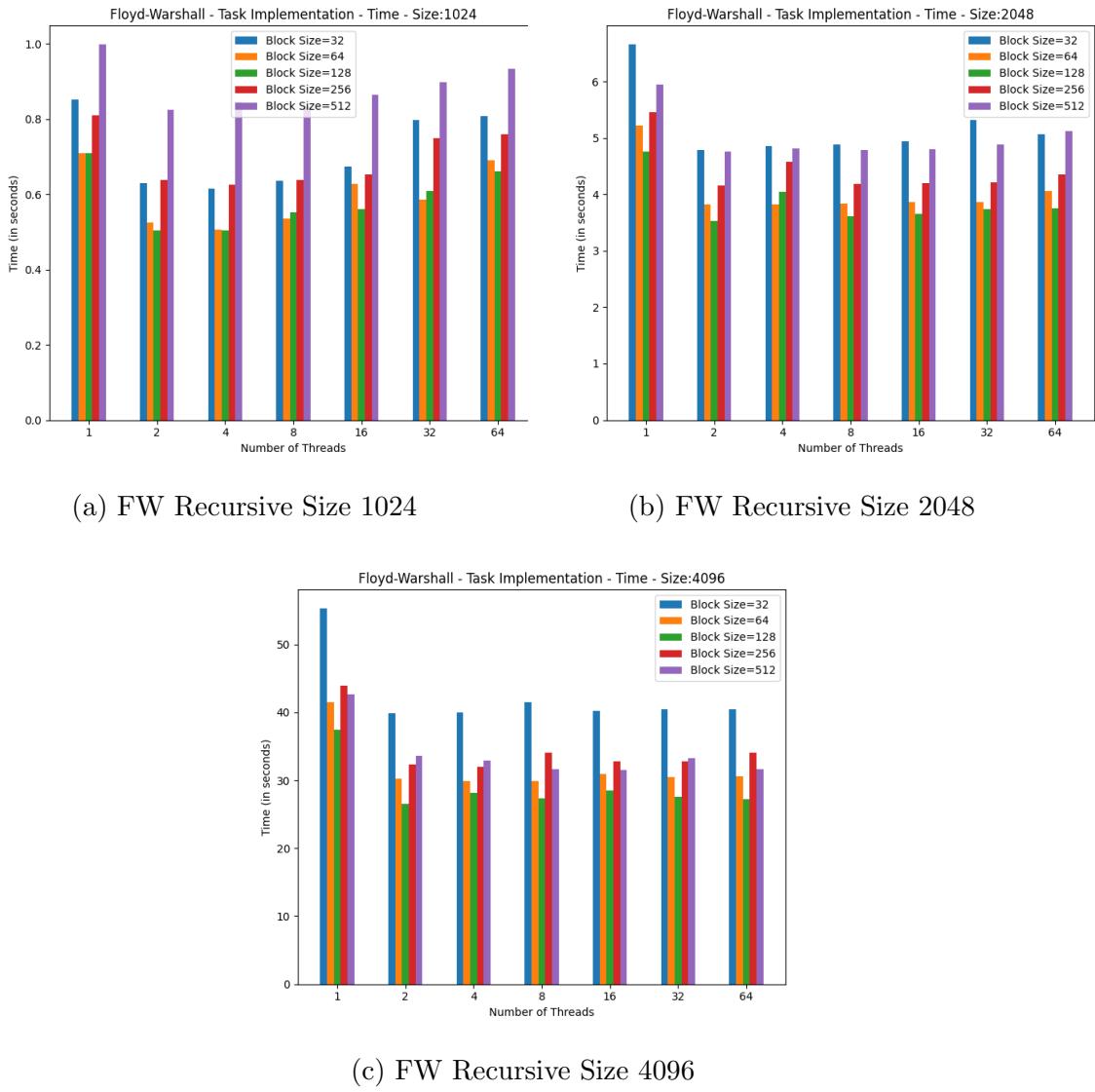


Figure 14: Χρόνος Εκτέλεσης FW Recursive

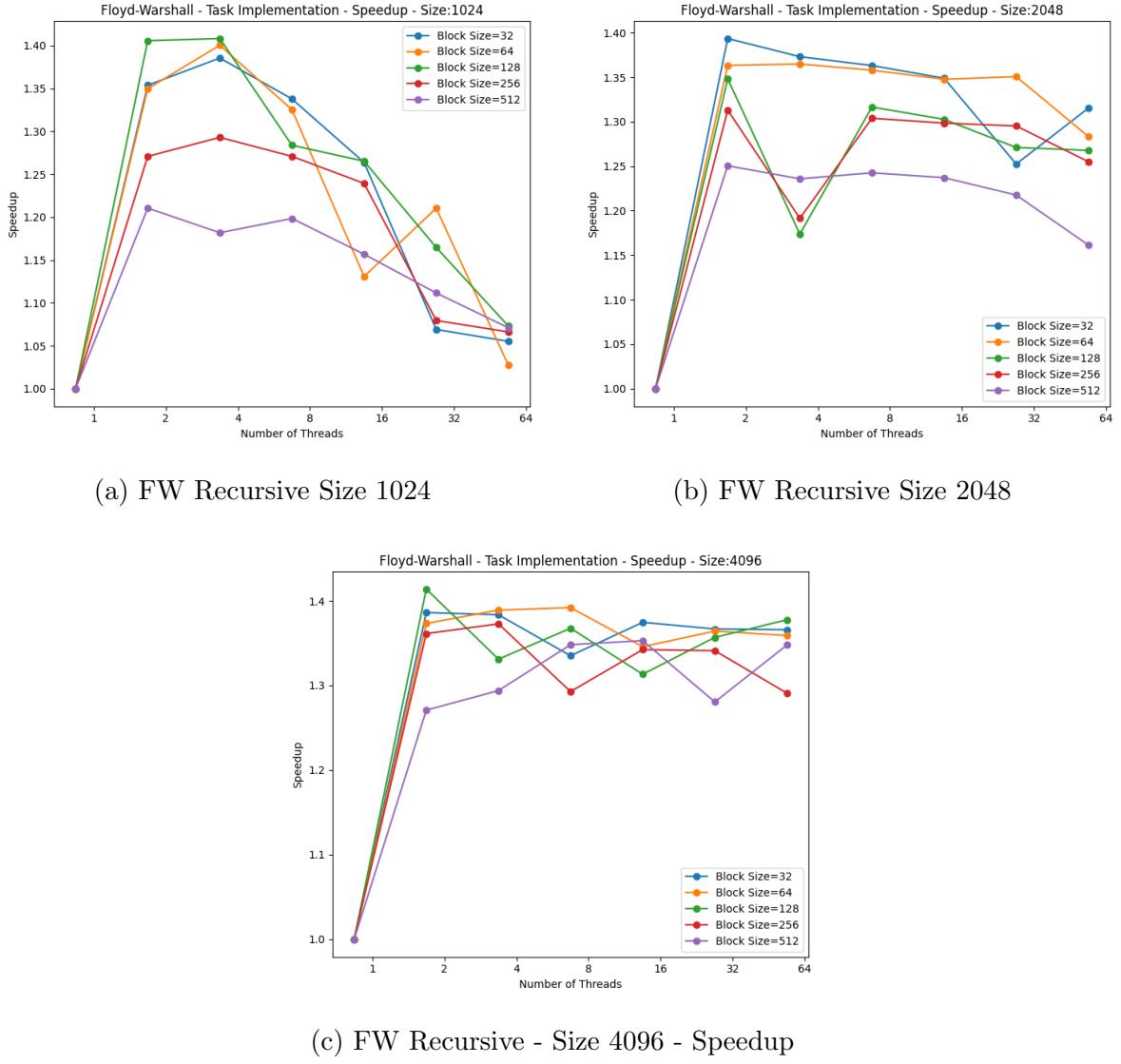


Figure 15: Speedup FW Recursive

### 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 ύστοντας ένα environmental variable, το **OMP\_NESTED** σε **TRUE**.

Η λύση αυτή δεν επαρκεί καθώς μετά μπορεί να επιφέρει αντίθετα αποτελέσματα από τα επιμυητά. Επιτρέποντας την εκτέλεση φωλιασμένων παράλληλων περιοχών, υπάρχει ο κίνδυνος να καλέσουμε πολλά νήματα για την δημιουργία των task άλλα 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:

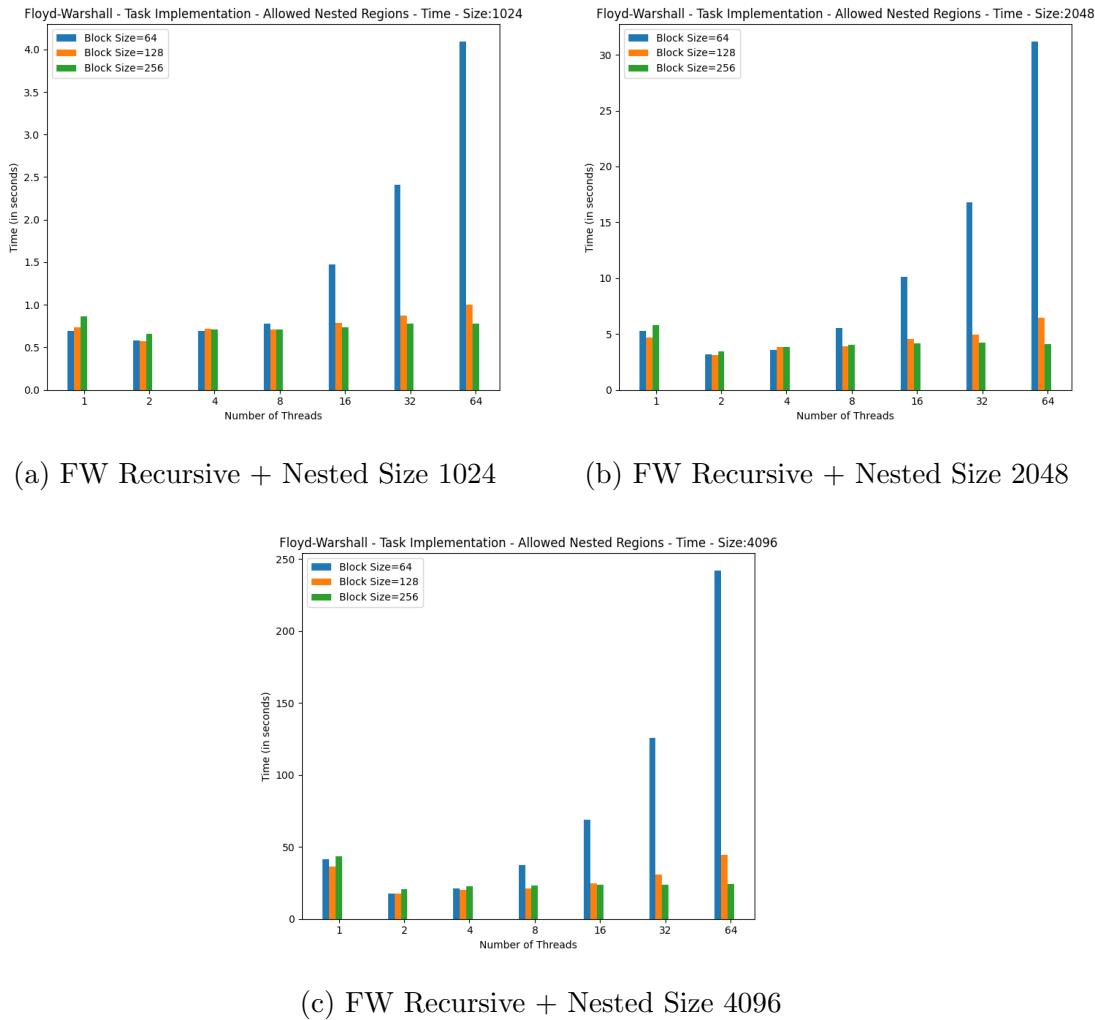


Figure 16: Χρόνος Εκτέλεσης FW Recursive/Nested

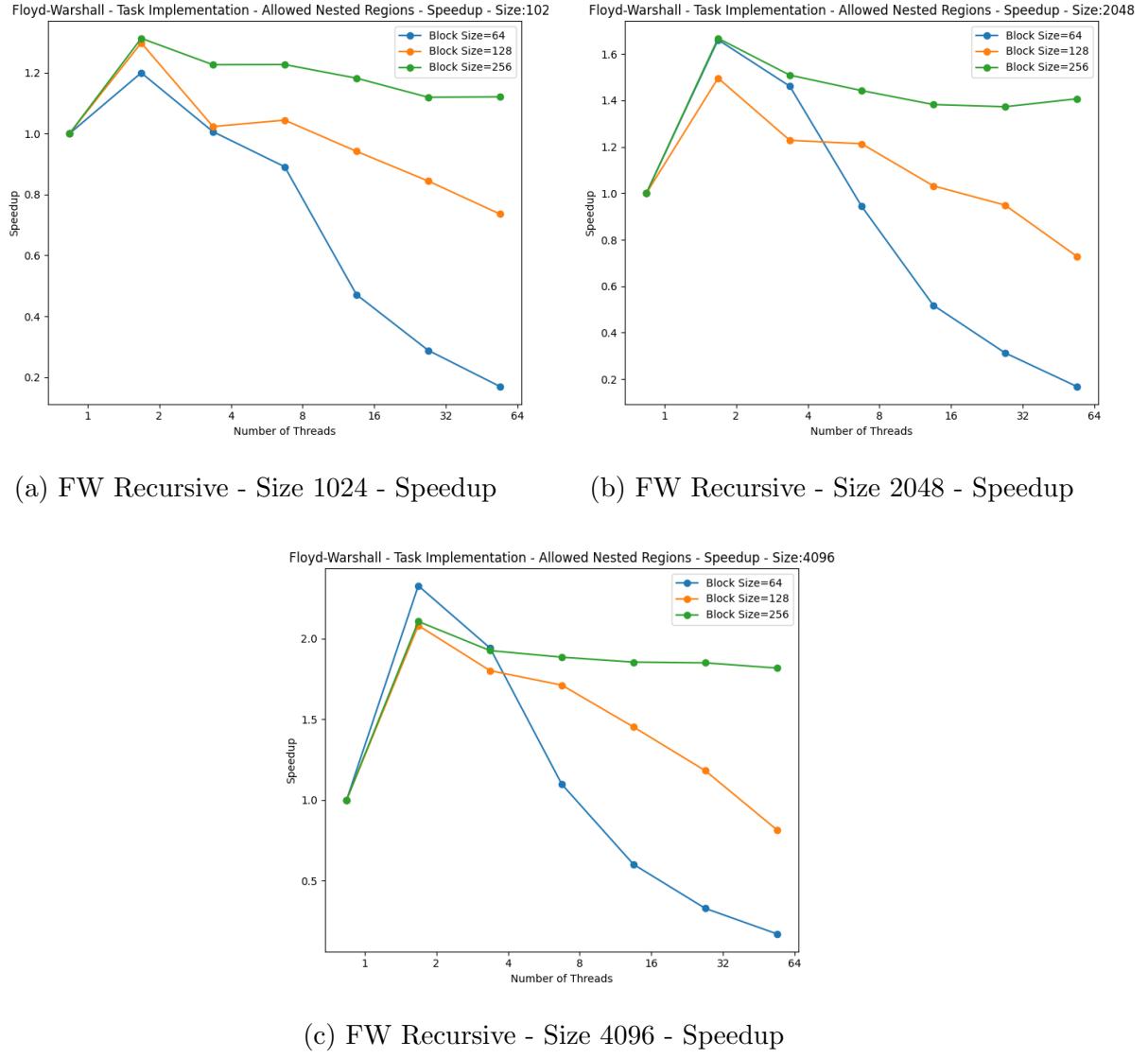


Figure 17: Speedup FW Recursive/Nested

### 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.

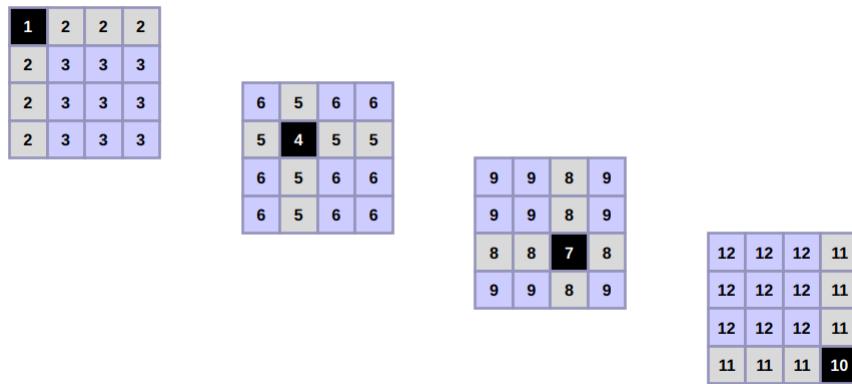
Η κατάλληλη επιλογή block size εξαρτάται κάθε φορά από τον επεξεργαστή που χρησι-

μοποιούμε. Σημαντικό να μην χρησιμοποιηθούν μικρά block sizes στην έκδοση που επιτρέπονται οι φωλιασμένες παράλληλες περιοχές γιατί γρήγορα υπερφορτώνεται το pool από 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).

Τλοποιήσαμε 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 τρέζουν 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.

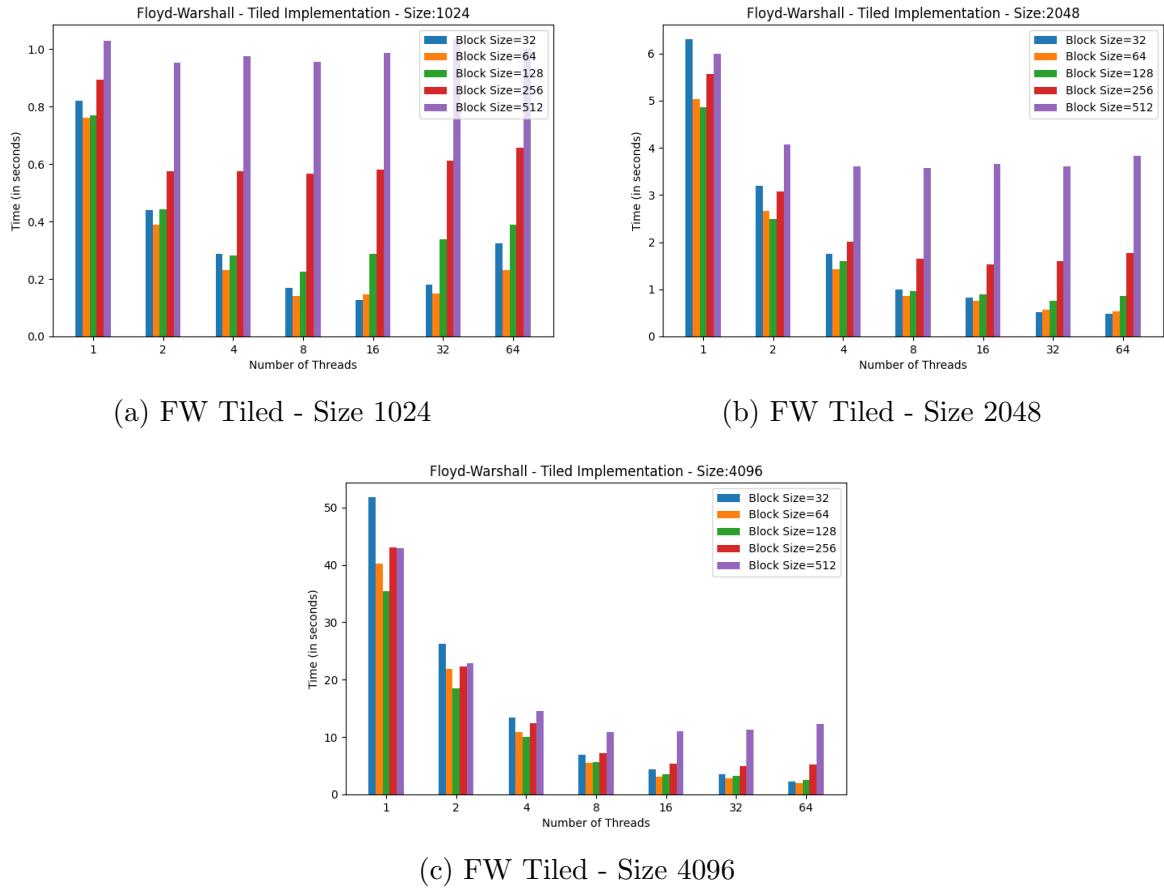


Figure 19: Χρόνος Εκτέλεσης FW Tiled

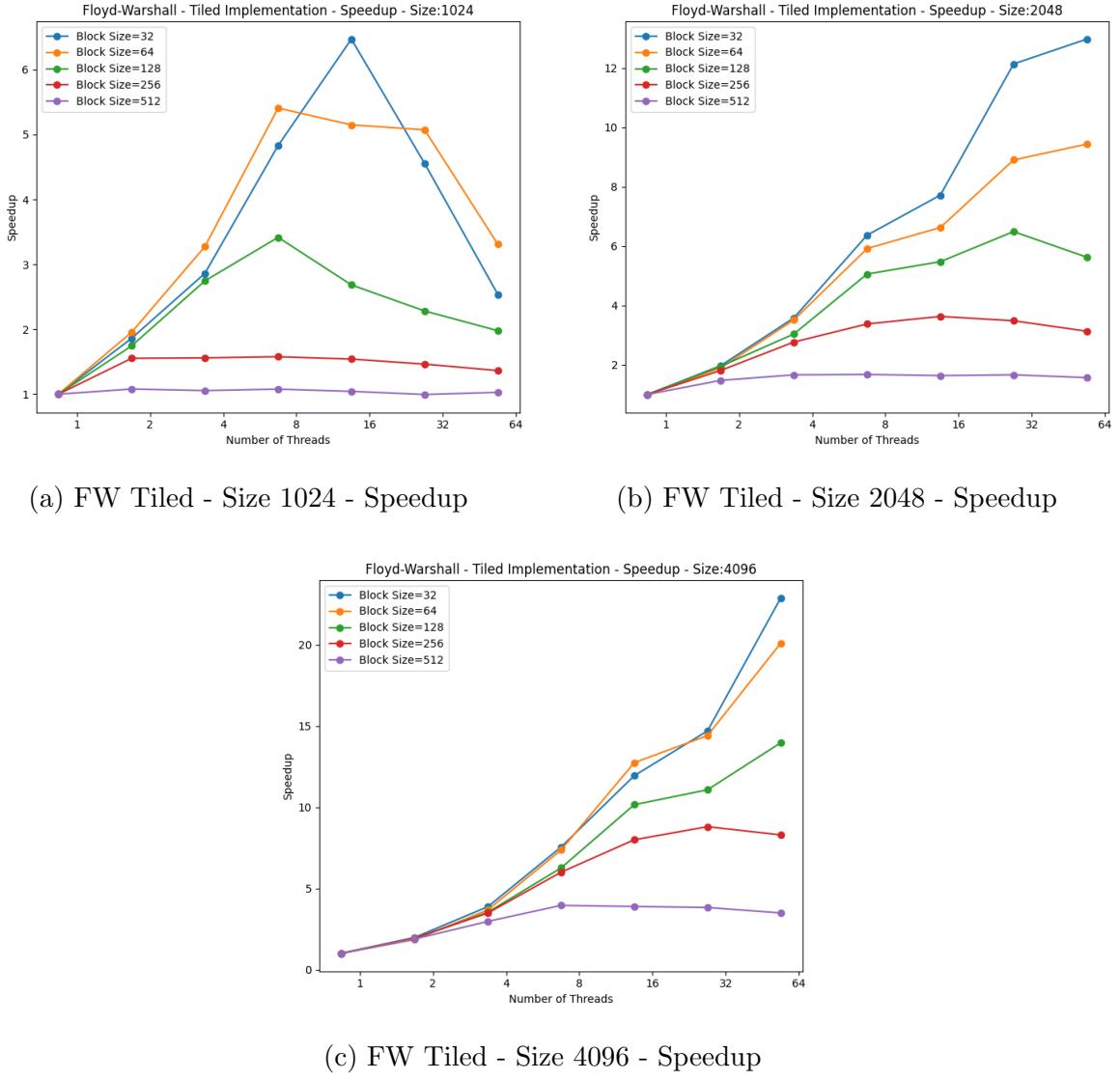


Figure 20: Speedup FW Tiled

### 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. Συγχριτικά 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.

H 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.

### 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 κλειδώματα

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.

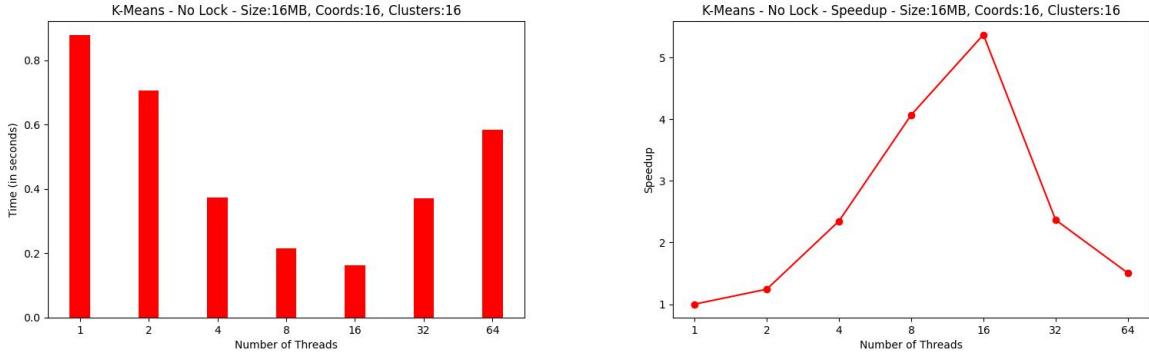


Figure 21: K-Means Naive - No Sync

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

To 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.

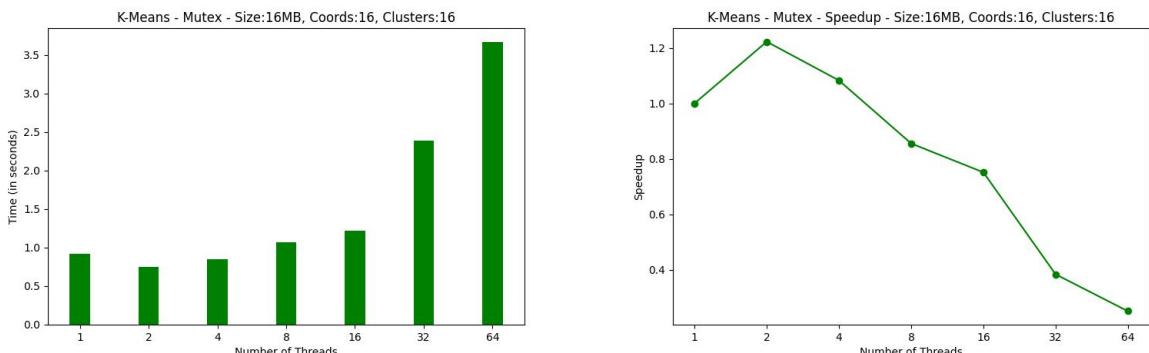


Figure 22: K-Means Naive - PThread - Mutex

Φαίνεται ότι ο χρόνος εκτέλεσης επηρεάζεται πολύ από την χρήση του κλειδώματος. Το overhead που προσθέτει το συνεχόμενο κλειδωμα/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 μπει 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).

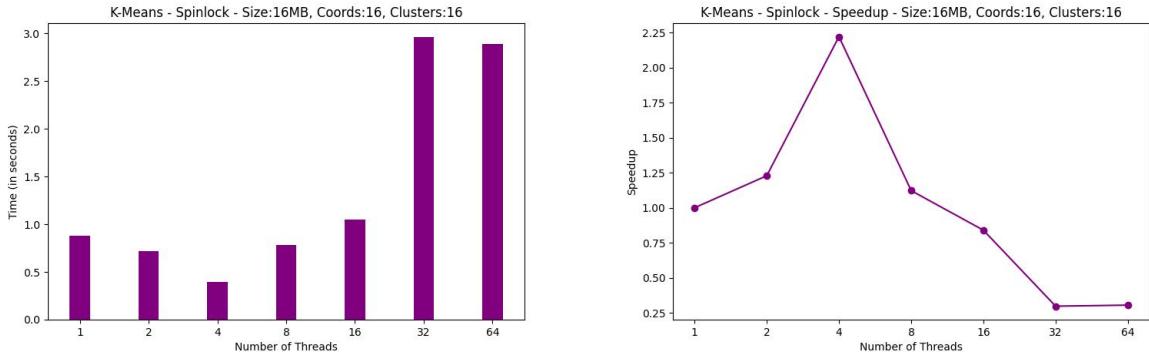


Figure 23: K-Means Naive - PThread - Spinlock

Παρατηρούμε πολύ καλύτερη απόδοση συγκριτικά με το κλειδωμα με mutex. Η απουσία του context switching για το κλειδωμα και ξεκλείδωμα και το γεγονός ότι το χρίσιμο 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

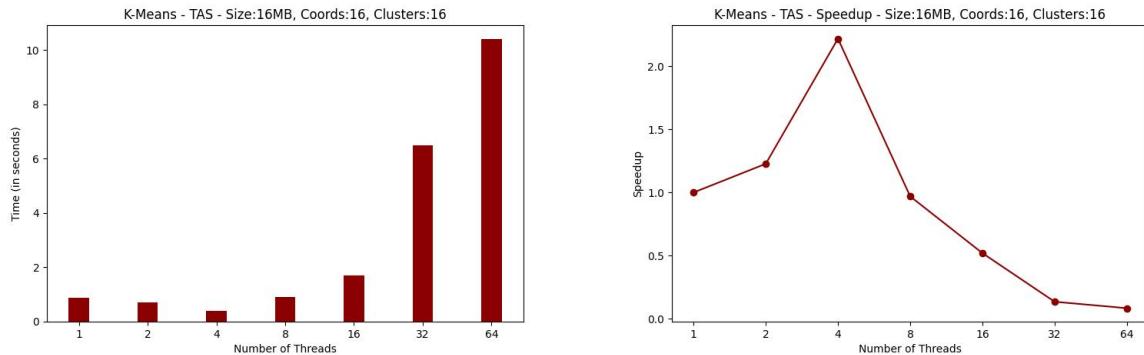


Figure 24: K-Means Naive - TAS

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, εκτελόμεις 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 χρειαστεί 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 κλειδωμα.

### 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 αλλάζει 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.

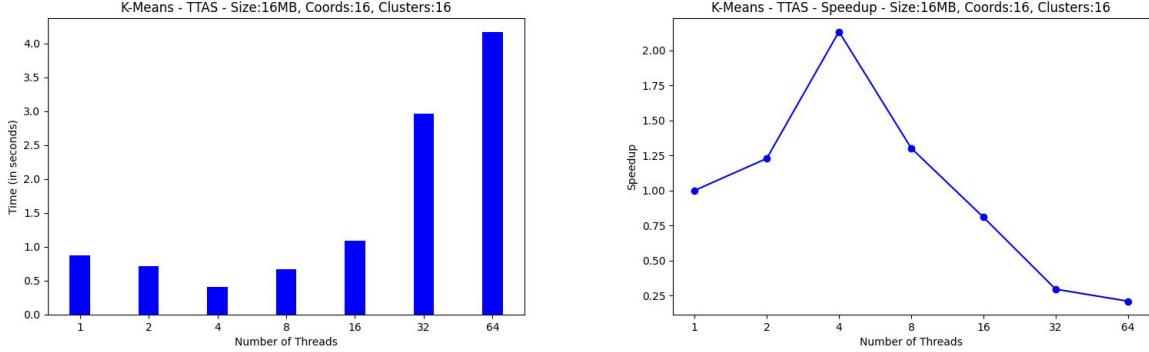


Figure 25: K-Means Naive - TTAS

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 δίσκο 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.

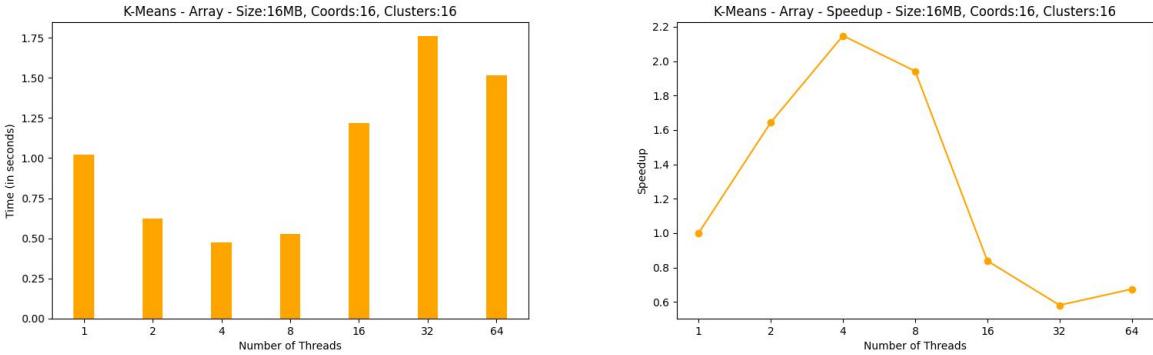


Figure 26: K-Means Naive - Array Based Lock

Ο συγκεκριμένος τύπος κλειδώματος είναι ο πρώτος που επιχειρεί να είναι NUMA/cache coherent aware. Η τιμή που δείχνει την ύση του κάθε nόματος (mySlotIndex) είναι τοπική για κάθε nόμα. Δηλαδή, τα nόματα κάνουν spin σε μια ύση μνήμης που είναι τοπική. Έτσι, δεν προκαλείται coherence traffic χωρίς λόγο. Αν και ο boolean πίνακας είναι διαμοιραζόμενος, τα nόματα δεν ”παλεύουν” να αποκτήσουν πρόσβαση σε αυτή την ύση μνήμης καθώς κάνουν spin στις τοπικές ύσεις 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 nόματα 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 ως ουρά, εκτελείται first-come-first-served σειρά εκτέλεσης.

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

### 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 nόματα κάνουν spin τοπικά. Αντί για ουρά, χρησιμοποιεί μια σχεδίαση linked list. Το κάθε nόμα έχει τοπικά ένα QNode που δείχνει την κατάσταση του. Αν η boolean τιμή locked είναι true, τότε το nόμα είτε έχει έλεγχο του lock ή περιμένει για αυτόν. Αν είναι false, έχει αφήσει το lock. Κάθε nόμα αναφέρεται στο node του προηγούμενου nόματος 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 ύση του πίνακα που κοιτάζει την κατάσταση του nόματος, το κάθε nόμα 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.

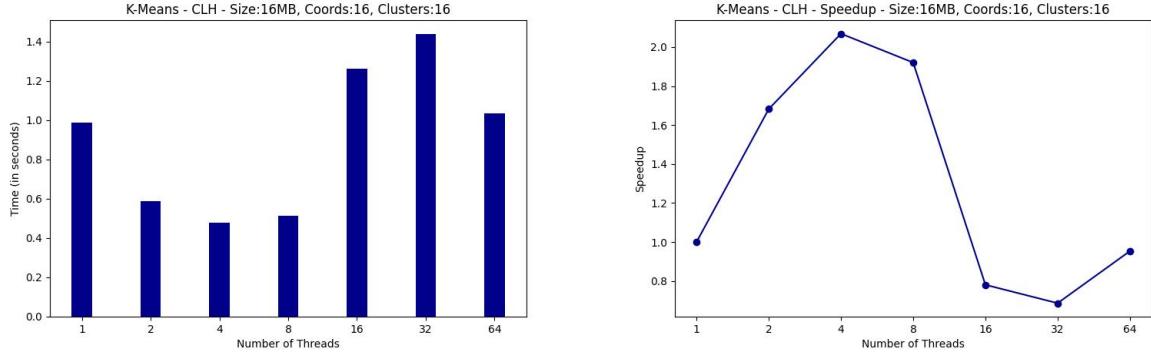


Figure 27: K-Means Naive - CLH Lock

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

## 3.2 Mutual Exclusion με χρήση critical/atomic pragma

Το **#pragma omp critical** δηλώνει ότι η παραχάτω περιοχή θα πρέπει να εκτελείται από ένα μόνο νήμα κάθε φορά. Δηλαδή, εφαρμόζεται mutual exclusion για αυτή την περιοχή. Προσφέρει μια πολύ ευκολότερη λύση για τον προγραμματιστή στο να εφαρμόσει mutual exclusion όμως δεν προσφέρει έλεγχο για τον τύπο του κλειδώματος.

Το OpenMP επιλέγει (συνήθως) από μόνο του τι είδους κλειδώματα να χρησιμοποιήσει. Υπάρχει και η επιλογή ο προγραμματιστής να δώσει ένα "hint" στο directive αν γνωρίζει από πριν την συμπεριφορά των νημάτων. Τα hints είναι τα εξής:

**Uncontended:** Αν περιμένουμε λίγα νήματα να προσπαθήσουν ταυτόχρονα να αποκτήσουν το lock.

**Contented:** Αν περιμένουμε πολλά νήματα να προσπαθήσουν ταυτόχρονα να αποκτήσουν το lock.

**Speculative:** Ο προγραμματιστής προτείνει να χρησιμοποιηθούν τεχνικές όπως transactional memory (speculative techniques).

**Non-Speculative:** Ακριβώς το αντίθετο με το προηγούμενο.

Τα μη-αντίθετα hints μεταξύ τους μπορούν να συνδιαστούν κίολας.

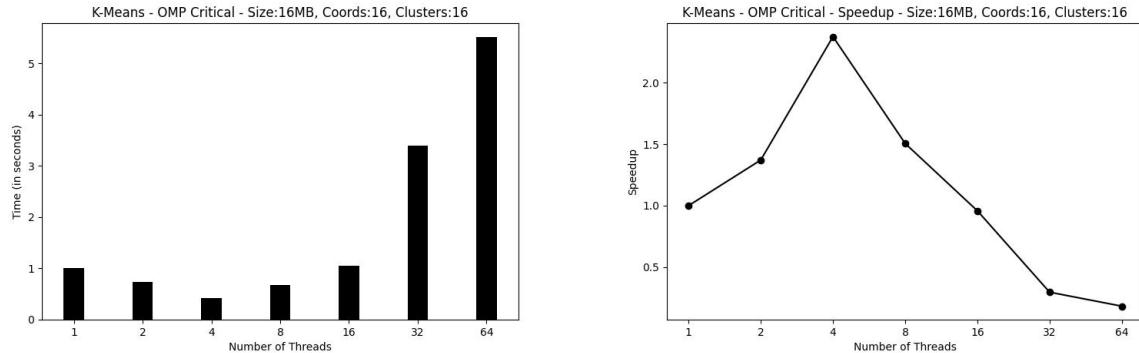


Figure 28: K-Means Naive - Critical Directive

Τα συμπεράσματα είναι παρόμοια με άλλες υλοποιήσεις lock. Το directive δεν κάνει scale με καλό τρόπο, την βέλτιστη απόδοση την έχει στα 2 νήματα. Για μεγαλύτερο αριθμό νημάτων, η απόδοση είναι μεταξύ του TAS και TTAS.

### Best Time - OMP Critical

0.4248s - 4 Threads

### Critical OR Atomic

Το atomic construct διαβεβαιώνει ότι η ακριβώς επόμενη εντολή γίνεται ατομικά. Δεν χρησιμοποιείται για structured block σαν αυτό που συναντάμε σε αυτό το πρόβλημα. Επειδή όμως εδώ το critical section αποτελείται μόνο από increments, μπορούμε να χρησιμοποιήσουμε δύο atomic directives, όπως κάναμε και στην προηγούμενη άσκηση.

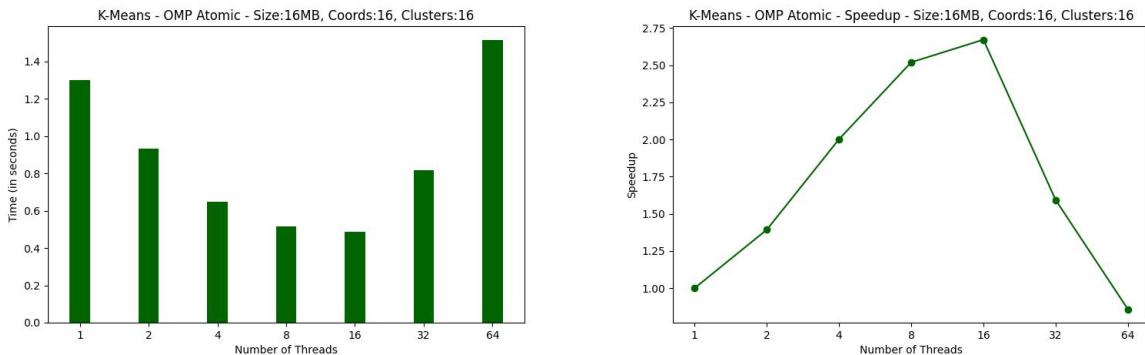


Figure 29: K-Means Naive - Atomic Directive

### Best Time - OMP Atomic

0.4867s - 16 Threads

Συγχρίνοντας την χρήση του critical και του atomic, καλύτερο χρόνο πετυχαίνει το critical για μικρότερο αριθμό νημάτων. Αν και το overhead για το critical είναι μεγαλύτερο από το overhead του atomic, το pragma omp atomic παρουσιάζεται περισσότερες φορές στο critical section ενώ το omp critical μόνο 1 φορά.

Σημαντικότερο design choice από το ποιο lock πρέπει να επιλέξει ένας προγραμματιστής είναι και τι μέθοδο θα χρησιμοποιήσει για τον συγχρονισμό, έίτε αυτό είναι συμβατικά locks, atomic operations, speculative methods ή reduction. Για παράδειγμα, για τον αλγόριθμο K-means, η version του reduction κλιμάκωνε με καλύτερο συμπεριφορά.

Γενικά, σαν τελικό συμπέρασμα για τα διαφορετικά κλειδώματα, αυτή η υλοποίηση του αλγορίθμου K-Means, δεν κλιμάκωνε πολύ καλά για μεγάλο αριθμό νημάτων. Τον καλύτερο χρόνο πετυχαίνει το TAS για μικρό αριθμό νημάτων όμως. Για τόσο λίγους "εργάτες", είναι πολύ σημαντικό το lock που θα χρησιμοποιήσουμε είναι να έχει μικρό overhead.

Το συμπέρασμα αυτό είναι αρκετά διαφορετικό για μεγάλο αριθμό νημάτων. Οι πιο εξειδικευμένες λύσεις όπως το Array Lock και το CLH Lock παρουσιάζουν πολύ πιο ισορροπημένα αποτελέσματα. Κύριος λόγος είναι η architectural aware υλοποίηση τους. Η εφαρμογή τους σε διαφορετικά προβλήματα που κλιμάκωνουν καλύτερα για περισσότερα νήματα θα έδειχναν πιο καθαρά την ανωτερότητα τους συγχριτικά με τα άλλα locks. Κύριο μειονέκτημα τους είναι το μεγαλύτερο overhead που έχουν και η λίγο μεγαλύτερη ανάγκη για μνήμη.

## All Times

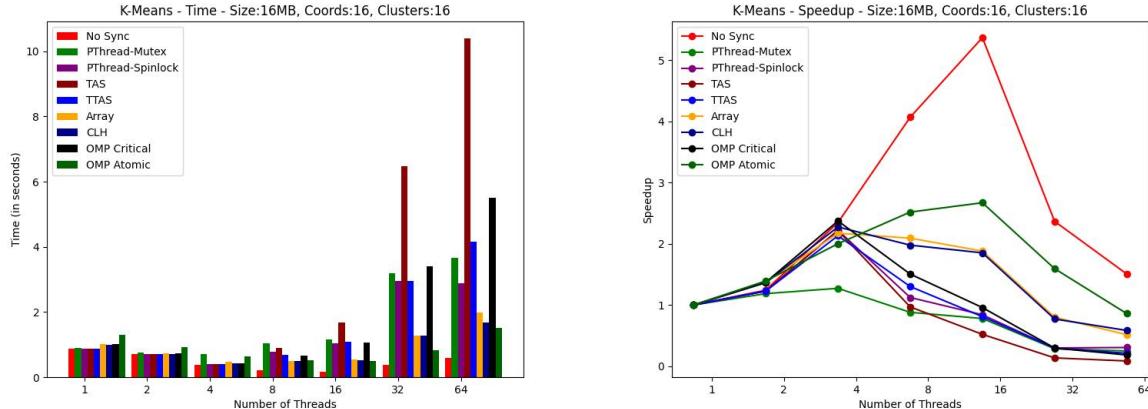


Figure 30: K-Means Naive - All Locks

## 4 Άσκηση 4 - Ταυτόχρονες Δομές Δεδομένων

Σκοπός της συγκεκριμένης άσκησης είναι η μελέτη και η αξιολόγηση της επίδοσης μιας ταυτόχρονης δομής δεδομένων. Η δομή αυτή είναι μια απλά συνδεδεμένη ταξινομημένη λίστα. Εκτελώντας πειράματα, βγαίνουμε σε συμπεράσματα για το πως οι διαφορετικοί τρόποι συγχρονισμού αποφέρουν και διαφορετικά αποτελέσματα.

## Δομή Συνδεδεμένης Λίστας

Οι δομές της απλά συνδεδεμένης λίστας είναι:

```
typedef struct ll_node {
    int key;
    struct ll_node *next;
} ll_node_t;

struct linked_list {
    ll_node_t *head;
};
```

Ένα set αποτελείται από μια συνδεδεμένη λίστα κόμβων (linked list of nodes). Το περιεχόμενο του κόμβου το αγνωστόμε καθώς δεν επηρεάζει τις μετρήσεις μας. Το αναγνωριστικό του κάθε κόμβου είναι ο αριθμός key. Ο πρώτος και ο τελευταίος κόμβος έχουν ιδιαίτερη σημασία και ονομάζονται head και tail αντίστοιχα. Ακολουθούμε τις εξής συμβάσεις για τα πειράματά μας:

- Τα head, tail nodes είναι σταθερά, δεν αφαιρούνται ούτε προστίθονται. Αποκαλούνται sentinel nodes.
- Η λίστα είναι ταξινομημένη βάσει κλειδιού, τα κλειδία είναι unique.
- Οι παρακάτω μέθοδοι ακολουθούν τους 2 αυτούς κανόνες. Π.χ. η μέθοδος add() δεν θα προσθέσει κόμβο με κλειδί ίδιο με ήδη υπάρχοντα κόμβο.
- Ένα αντικείμενο ανήκει στην λίστα μόνο και μόνο αν υπάρχει μονοπάτι που το βρίσκει έχοντας ως αφετηρία τον κόμβο head.

Οι μέθοδοι που υποστηρίζονται από αυτή την δομή είναι:

- **contains(x):** Επιστρέφει True μόνο και μόνο αν η λίστα περιέχει το x.
- **add(x):** Προσθέτει το x στην λίστα και επιστρέφει True μόνο αν το x δεν υπήρχε ήδη στην λίστα.
- **remove(x):** Αφαιρεί το x από την λίστα και επιστρέφει True μονο αν το x υπήρχε στην λίστα (και διαγράφηκε).

Μελετάμε την συμπεριφορά των παρακάτω τρόπων συγχρονισμού για διαφορετικό αριθμό νημάτων {1, 2, 4, 8, 16, 32, 64, 128}, για διαφορετικό μέγεθος λίστας {1024, 8192} και για διαφορετικό ποσοστό λειτουργιών {100/0/0, 80/10/10, 20/40/40, 0/50/50}. Για το ποσοστό λειτουργιών, ο πρώτος αριθμός δηλώνει ποσοστό αναζητήσεων, ο δεύτερος εισαγωγών και ο τρίτος διαγραφών.

## Επεξήγηση των Workloads

Είναι πολύ πιο ρεαλιστικό σε μια εφαρμογή να υπάρχουν περισσότερες κλήσεις τις μεθόδου `contains()` παρά των `add()` και `remove()`. Τα πρώτα 2 ενδέχομενα καλύπτουν αυτές τις ρεαλιστικές χρήσεις. Τα workloads 20/40/40 και 0/50/50 είναι πιο ακραία και πιο σπάνια σε κλασσικές εφαρμογές.

### Serial

Η σειριακή εκτέλεση προσφέρεται μόνο για λόγους benchmarking. Μέσω αυτών των αποτελεσμάτων, κρίνουμε και τις υπόλοιπες υλοποιήσεις. Οι μετρήσεις στον πίνακα αφορούν Throughput, δηλαδή πόσα Operations έχανε η κάθε υλοποίηση σε ένα δευτερόλεπτο. Συγκεκριμένα, ο οριθμός εκφράζει Kops/sec (Kilo-operations/second).

Workload Size	100/0/0	80/10/10	20/40/40	0/50/50
1024	1566.15	1897.13	1612.85	1410.06
8192	151.82	69.55	81.33	65.60

### Concurrent Linked List

Έχοντας δει scalable spin locks στην προηγούμενη άσκηση, τα χρησιμοποιούμε για να επιχειρήσουμε να δημιουργήσουμε scalable ταυτόχρονες δομές δεδομένων. Το lock που χρησιμοποιούμε σε όλες τις υλοποιήσεις που χρησιμοποιούν lock, είναι το pthread Spinlock.

Γνωρίζοντας τα αποτελέσματα από πριν, για να είναι πιο καθαρά τα διαγράμματα, χωρίζουμε τους τύπους του κλειδώματος σε low και all performers. Οι μετρικές που χρησιμοποιούμε για να δούμε ποια υλοποίηση είναι καλύτερη είναι το Throughput και το "Speedup".

Εδώ, δεν αναφερόμαστε στον χρόνο, άρα ο οριθμός του Speedup είναι διαφορετικός. Για να δούμε πόσο καλύτερη (ή χειρότερη!) είναι η παράλληλη υλοποίηση από την σειριακή, χρησιμοποιούμε τον τύπο  $Speedup = \frac{Throughput(Parallel)}{Throughput(Serial)}$ .

Για κάθε τύπο συγχρονισμού, επεξηγούμε το πως πετυχαίνουμε την παράλληλη εκτέλεση και πως εκτελούνται οι 3 προαναφερόμενοι μέθοδοι.

## 4.1 Low Performers

Οι υλοποιήσεις που θεωρούμε "Low Performers" είναι οι παρακάτω. Ο λόγος που τους θεωρούμε χαμηλών επιδόσεων είναι γιατί οι υλοποιήσεις αυτές είναι απλοϊκές και έχουν ξεκάθαρα προβλήματα που εμφανίζονται στο θεωρητικό επίπεδο. Με βάση αυτά τα προβλήματα, γνωρίζουμε από πριν την ανικανότητά τους να κλιμακώσουν.

### 4.1.1 Coarse-grain Locking

Η απλούστερη προσπάθεια για ταυτόχρονη συνδεδεμένη λίστα. Υπάρχει ένα lock για όλη την δομή και κάθε φορά που ένα νήμα θέλει να εφαρμόσει μια μέθοδο, περιμένει μέχρι να αποκτήσει το lock και μετά εκτελεί την λειτουργία του κανονικά. Η μόνη προσθήκη που πρέπει να κάνουμε στην δομή είναι ένα spinlock και η κάθε μέθοδος να κάνει lock στο κάλεσμά της και unlock στο τέλος της.

Το προφανές συμπέρασμα είναι ότι αν και η τεχνική αυτή λειτουργεί πλήρως, η απόδοσή της είναι πάντα όσο η σειριακή ή και χειρότερη λόγω του overhead του κλειδώματος.

#### 4.1.2 Fine-grain Locking

Εφόσον η συνδεδεμένη λίστα αποτελείται από ένα πεπερασμένο πλήθος κόμβων, μπορούμε αντί να έχουμε ένα lock για όλη την δομή, να έχουμε πολλαπλά locks, συγχεκριμένα ένα για κάθε κόμβο. Αυτό το καταφέρνουμε τοποθετώντας ένα spinlock σε κάθε node της συνδεδεμένης λίστας.

Πλέον, για να υπάρχει ασφάλεια στην χρήση των μεθόδων από τα νήματα, πρέπει να υπάρξει ένα σύστημα hand-over-hand locking. Για να κλειδώσει ο curr κόμβος πρέπει να είναι ήδη κλειδωμένος ο pred κόμβος. Αυτό το κλείδωμα και ξεκλείδωμα πρέπει πάντα να γίνεται με αυτή την ακολουθία.

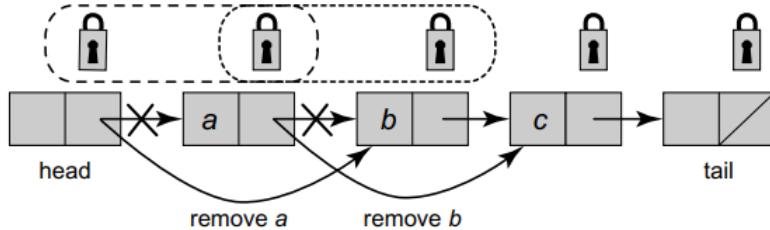


Figure 31: Fine-grained Locking - Remove method - Hand-over-Hand locking

Για την διαγραφή, πρέπει να κλειδωθεί ο κόμβος που πρόκεται να κλειδωθεί για να μην δράσει κάποια άλλη λειτουργία πάνω του και επίσης πρέπει να κλειδωθεί ο προηγούμενος κόμβος καθώς αυτός θα πραγματοποιήσει την διαγραφή, αλλάζοντας την διεύθυνση next.

Η διάσχιση της λίστας πρέπει αποκλειστικά να γίνεται με αυτό το σύστημα lock coupling. Συνέπεια αυτού είναι ότι αν ένα στοιχείο κλειδώσει στις αρχές της λίστας, απαγορεύεται από άλλα νήματα να επεξεργαστούν αργότερα στοιχεία που βρίσκονται πιο μετά στην λίστα.

Το fine-grained locking μπορεί να δεχτεί πολλές βελτιώσεις. Είναι πολύ πιθανό κάποια νήματα να περιμένουν πολύ ώρα να αποκτήσουν πρόσβαση στην δομή σε τελείως διαφορετικά σημεία από αυτά που επεξεργάζονται άλλα νήματα. Για αυτό ευθύνεται η ανάγκη για hand-over-hand locking.

#### 4.1.3 Optimistic synchronization

Στην "αισιόδοξη" υλοποίηση, περιμένουμε να έχουμε πολύ μικρές πιθανότητες κάτι να πάει στραβά. Αναζητούμε χωρίς να κλειδώνουμε τα στοιχεία αλλά με το που βρούμε τους κόμβους που μας ενδιαφέρουν τότε να επαληθεύουμε ότι οι κλειδωμένοι κόμβοι είναι οι σωστοί. Αν λόγω κακού συγχρονισμού κλειδώνουμε τους λάθους κόμβους, τότε απελευθερώνουμε τις κλειδαριές και ξαναπροσπαθούμε.

Άρα, αποδεχόμαστε το ρίσκο αλλά δεν αφήνουμε την δομή πλήρως ασυνεπή. Προσθέτουμε την μέθοδο **validate()**. Η μέθοδος αυτή ελέγχει αν τα δύο στοιχεία (pred,curr) βρίσκονται ακόμα στην δομή. Με βάση τις παραπάνω συμβάσεις αυτό σημαίνει:

- Ο pred κόμβος να είναι προβιβάσιμος από το head της δομής
- Να ισχύει ( $\text{pred.next} == \text{curr}$ , δηλαδή ο επόμενος του προηγούμενου είναι ο τρέχοντας)

Πλέον, για να τρέξουμε τις μεθόδους, ακολουθούμε την εξής λογική:

Δεν κλειδώνουμε όσο διατρέχουμε την λίστα. Όταν φτάσουμε στο σημείο που θέλουμε να εκτελέσουμε ενέργεια, τότε κλειδώνουμε τον προηγούμενο ακριβώς κόμβο και τον κόμβο που θέλουμε να εκτελέσουμε ενέργεια. Αφού γίνει το κλείδωμα, τρέχουμε την validate(pred, curr). Αν αποτύχει, ξαναξεκινάμε την διαδικασία. Αν πετύχει, εκτελούμε την ενέργεια που θέλαμε. Τέλος, ξεκλειδώνουμε πάλι με την ίδια σειρά, πρώτα τον προηγούμενο κόμβο και μετά τον τωρινό.

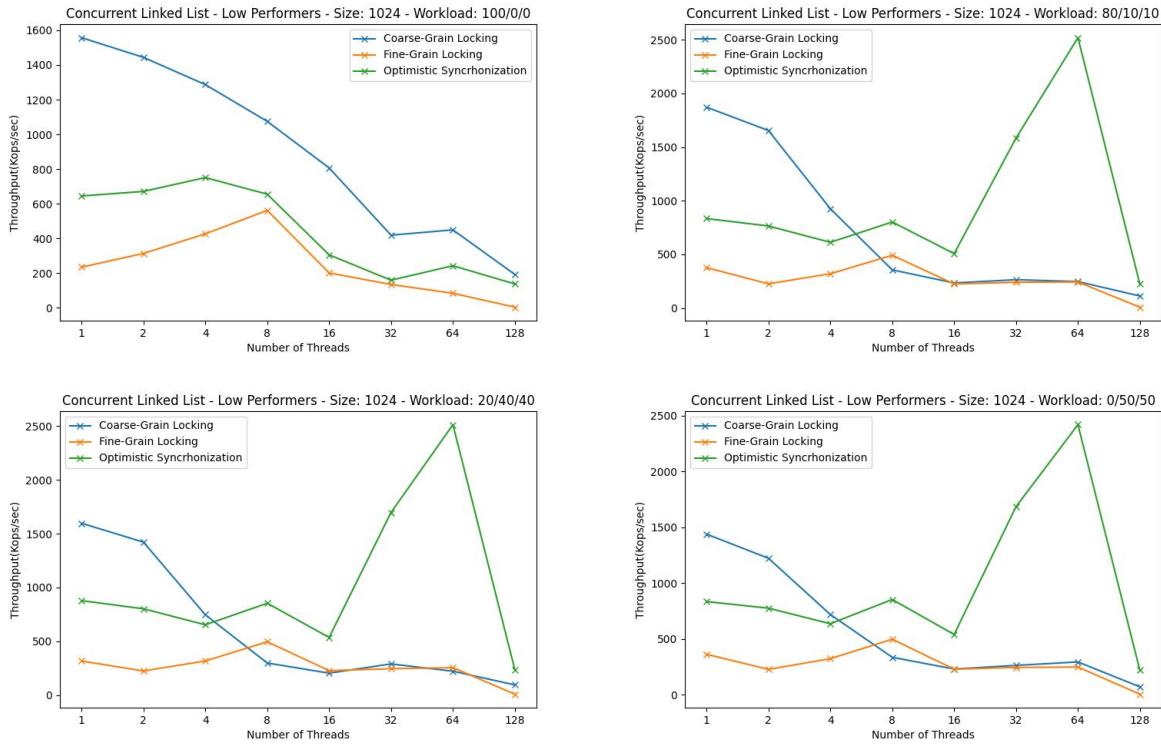


Figure 32: Concurrent Linked List - Throughput - Size 1024 - Low Performers

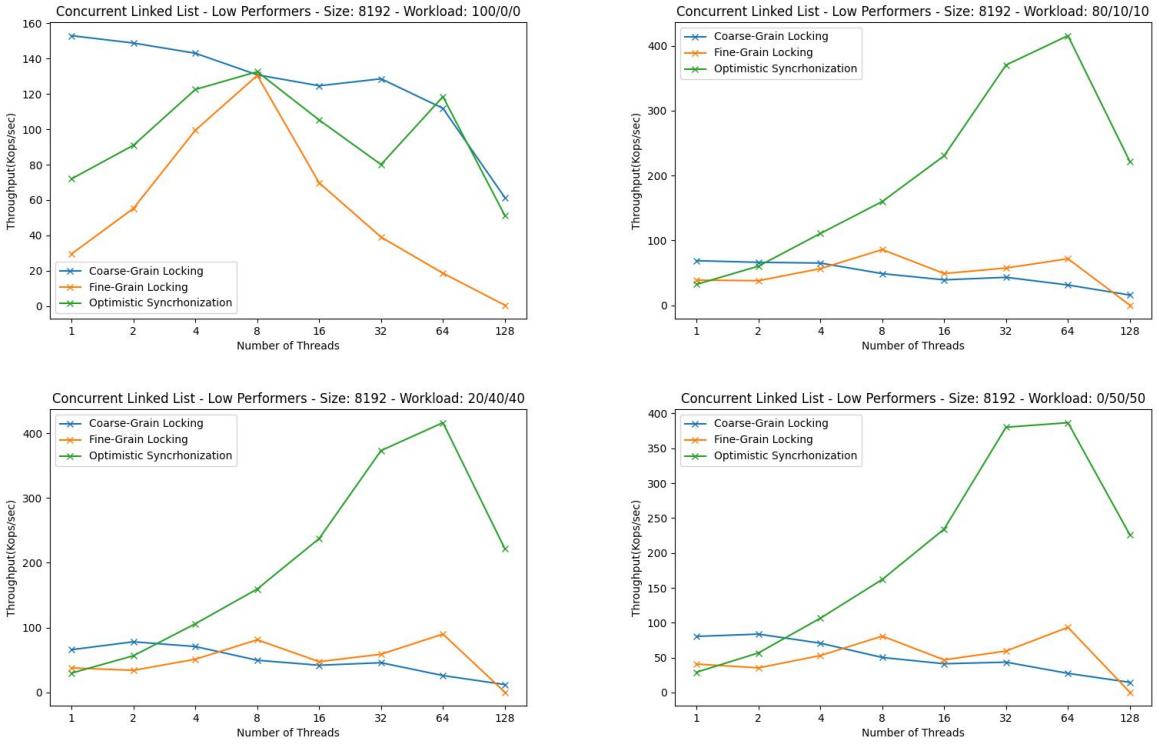


Figure 33: Concurrent Linked List - Throughput - Size 8192 - Low Performers

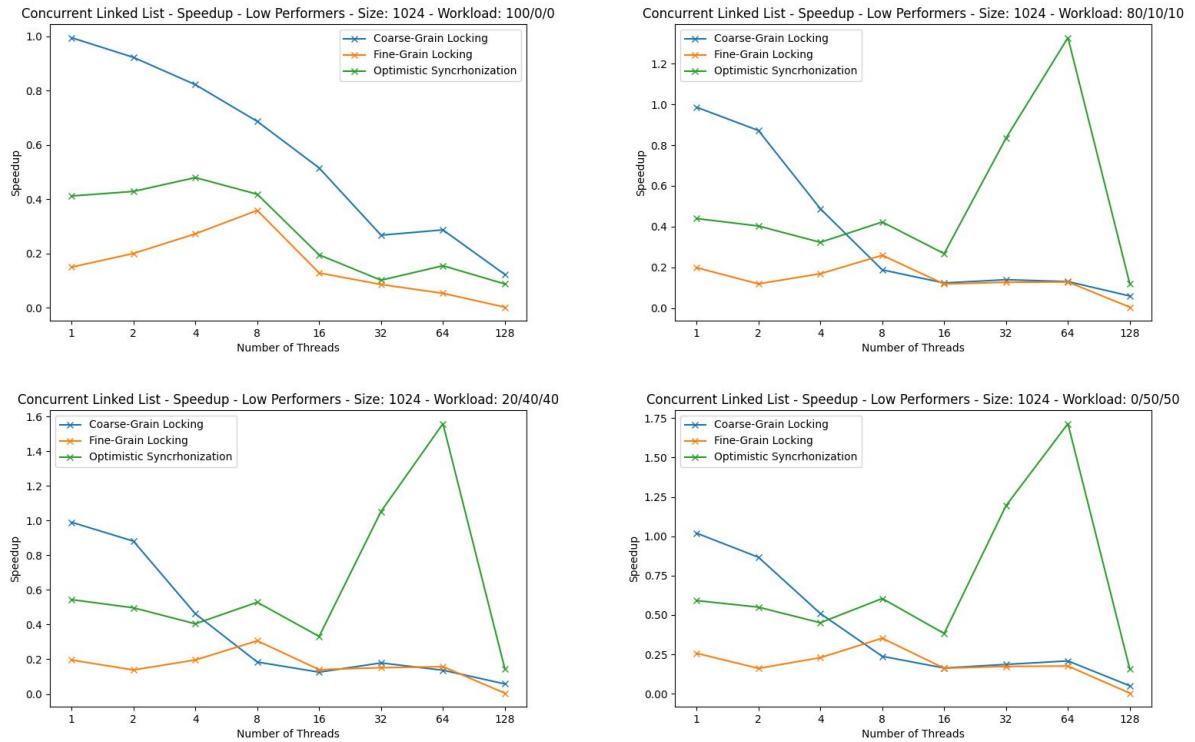


Figure 34: Concurrent Linked List - Speedup - Size 1024 - Low Performers

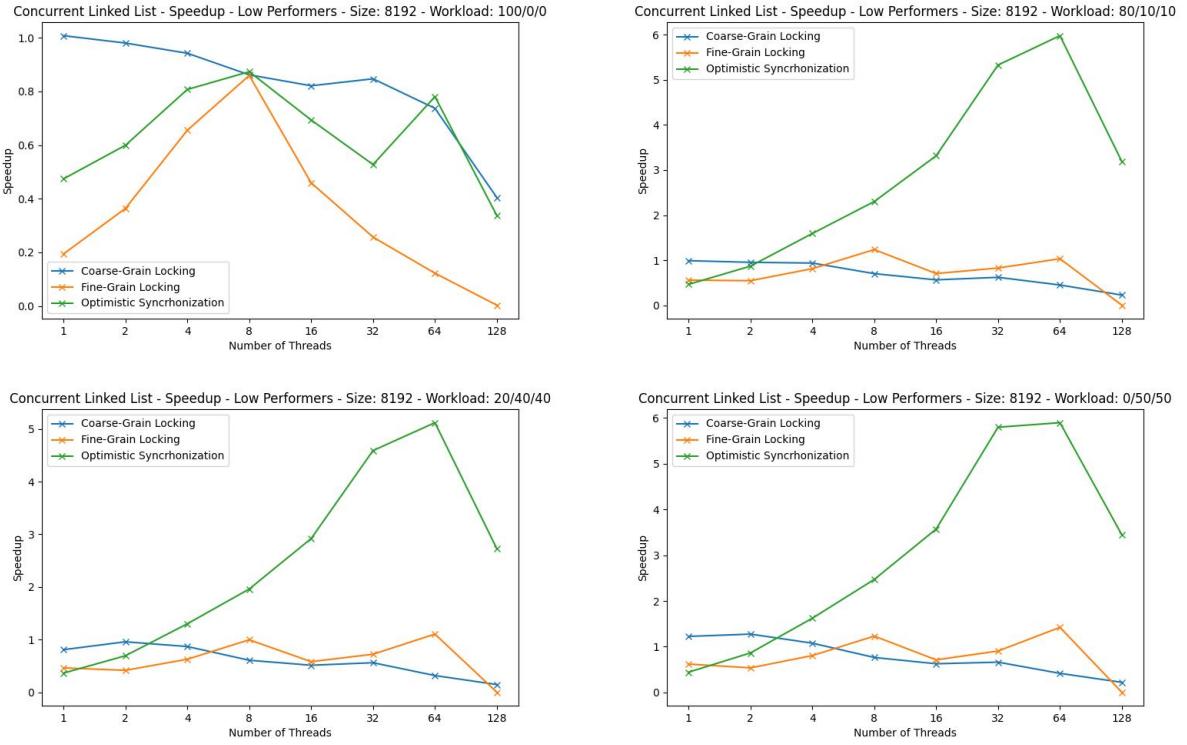


Figure 35: Concurrent Linked List - Speedup - Size 8192 - Low Performers

## Low Performers - Συμπεράσματα

### Coarse-Grained Synchronization

Παρατηρώντας τα γραφήματα, βγαίνουμε στο συμπέρασμα ότι για οποιοδήποτε συνδυασμό λειτουργιών ή και μεγέθους λίστας, η απόδοση του είναι είτε ίση με την σειριακή ή χειρότερη, ανεξαρτήτως workload ή μεγέθους λίστας. Το συμπέρασμα αυτό είναι προφανές καθώς η υλοποίηση της coarse-grained λίστας είναι η σειριακή με την προσθήκη ενός lock ώστε να υπάρχει mutual exclusion μεταξύ νημάτων.

### Fine-Grained Synchronization

Τα συμπεράσματα είναι αρκετά παρόμοια και για την fine-grained υλοποίηση. Οποιαδήποτε λειτουργία απαιτεί να διατρέξουμε την λίστα από το head της λίστας. Η αναζήτηση αυτή γίνεται πάντα με χρήση του hand-over-hand locking. Δηλαδή, έχουμε πολλαπλά κλειδώματα και ξεκλειδώματα για οποιαδήποτε λειτουργία. Πάλι, η απόδοση είναι πολύ χειρότερη από την σειριακή ανεξαρτήτως χαρακτηριστικών μεγέθους/workload.

Σε θέματα ταυτοχρονισμού, η υλοποίηση δεν προσφέρει πολλά. Αν κάποιο νήμα κλειδώσει lock νωρίς στην λίστα, τα υπόλοιπα νήματα δεν μπορούν να προχωρήσουν μετά από αυτό το σημείο λόγω της αναγκαιότητας για το hand-over-hand locking.

## Optimistic Synchronization

Στην "αισιόδοξη" υλοποίηση παρατηρούμε τις πρώτες βελτιώσεις στην απόδοση. Για μικρές λίστες, απαιτούνται πολλά νήματα, τουλάχιστον 32, ώστε το throughput να είναι μεγαλύτερο από το σειριακό. Οι επιδόσεις περιορίζονται από το γεγονός ότι η validate() αναγκάζεται να διασχίζει όλη την λίστα μέχρι το σημείο ενδιαφέροντος και από τα conflicts που μπορεί να προκύπτουν, είτε απόκτησης lock ή synchronization conflict.

Για μεγάλες λίστες, η απόδοση κλιμακώνει αρκετά γραμμικά, μέχρι τα πρώτα 32 νήματα, μετά η επίδοση είτε αυξάνεται ελάχιστα ή καταρέει. Ενδιαφέρουσα σημείωση ότι η optimistic υλοποίηση δεν έχει καλές επιδόσεις στο workload 100/0/0 αν και η μέθοδος contains() είναι η πιο "ελαφριά" από τις 3 και θα έπρεπε τουλάχιστον να αποδίδει όπως τα άλλα workloads στην μεγάλη λίστα.

## 4.2 High Performers

Οι υλοποιήσεις που θεωρούμε "High Performers" είναι το lazy και το non-blocking synchronization. Μέχρι στιγμής, η κάθε υλοποίηση προσπαθεί να διορθώσει τα σημαντικότερα προβλήματα της προηγούμενης. Θα δούμε ότι και οι δύο εκδοχές έχουν πολύ παρόμοιες επιδόσεις.

### 4.2.1 Lazy synchronization

Ο "τεμπέλικος" συγχρονισμός είναι το επόμενο βήμα βελτιστοποίησης. Σε κάθε κόμβο, προσθέτουμε μια boolean τιμή με το όνομα "marked". Η τιμή αυτή δείχνει αν ένας κόμβος βρίσκεται (λογικά) στην λίστα. Αν η τιμή marked είναι True, τότε ο κόμβος αυτός έχει αφαιρεθεί λογικά από την λίστα. Ο αλγόριθμος θεωρεί ότι οποιοδήποτε κόμβος δεν είναι marked, είναι και valid στην λίστα.

Η validate() ανανεώνεται ελέγχοντας πλέον και αν κάποιος κόμβος είναι marked. Δεν χρειάζεται να διατρέχει όλη την λίστα.

Πλέον η contains() δεν θα βρεθεί ποτέ σε conflict παρόμοια με αυτά που θα βρισκόταν στην optimistic υλοποίηση, είναι πλήρως wait-free. Επίσης, η add() και η remove(), αν και κάνουν block, διατρέχουν την λίστα μόνο μια φορά, λόγω της ανανεωμένης validate(). Η add(), διατρέχει την λίστα, κλειδώνει τον predecessor του κόμβου που πρόκεται να προσθέσει, και εκτελεί την ενέργειά της. Η remove(), πρώτα κάνει mark τον κόμβο που πρόκεται να αφαιρέσει και μετά αλλάζει τον δείκτη του predecessor του στόχου ώστε να έχει ορθότητα (και συνέχεια) η λίστα.

Την "ψυσική" αφαίρεση του κόμβου από την μνήμη την κάνει είτε το σύστημα εκτέλεσης (garbage collector) ή ο προγραμματιστής αν γράφει σε κάποια γλώσσα που ο έλεγχος της μνήμης γίνεται χειροκίνητα. Την υλική αυτή αφαίρεση μπορούμε να την κάνουμε ανά batches, και όχι κάθε φορά που αφαιρούμε λογικά έναν κόμβο, δηλαδή έχουμε έλεγχο για το πότε θα τρέξουμε αυτή την, μπορεί χρονοβόρα, ενέργεια.

Οι βελτιώσεις αυτές καθιστούν την lazy υλοποίηση πολύ ελκυστική λύση. Περιμένουμε να δούμε πολύ καλύτερα αποτελέσματα, ειδικά σε workloads που έχουν μεγάλο ποσοστό contains(), καθώς αυτή η μέθοδος είναι πλέον wait-free.

#### 4.2.2 Non-blocking synchronization

Οι τελικές βελτιστοποιήσεις που μπορούμε να κάνουμε είναι να αφαιρέσουμε πλήρως την ανάγκη για locks. Η μέθοδος `contains()` ήδη λειτουργεί χωρίς κλειδώματα και δεν χρειάζεται να κάνουμε κάποιες αλλαγές σε αυτή.

Θα προσπεράσουμε αρχετές λεπτομέρειες για το πως λειτουργεί ο non-blocking συγχρονισμός καθώς είναι εξαιρετικά περίπλοκος προγραμματιστικά.

Οι μέθοδοι `add()` και `remove()` όμως πρέπει να δεχτούν αλλαγές. Η πρώτη αλλαγή είναι να θεωρήσουμε την διεύθυνση `next` και την τιμή `marked` του κάμβου ως μια ”μονή ατομική μονάδα”, δηλαδή, οποιαδήποτε προσπάθεια να αλλάξει η τιμή `next` όταν η τιμή `marked` είναι `True`, θα αποτύχει.

Σε αντίθεση με την `lazy` υλοποίηση που η φυσική αφαίρεση μπορούσε να γίνει σε δική μας κλήση, η non-blocking λίστα ακολουθεί διαφορετική λογική. Οι `add()` και `remove()`, όσο διατρέχουν την λίστα, ελέγχουν για `marked` κόμβους. Αν βρουν `marked` κόμβο, εκτελούν μια μέθοδο `compareAndSet()`, με σκοπό την αφαίρεση αυτού του κόμβου. Η συμπεριφορά αυτή είναι κοινή και για τις 2 μεθόδους. Και οι 2 μέθοδοι χρησιμοποιούν ατομική μέθοδο `compare-and-swap` για να εκτελέσουν τις ενέργειές τους. Αν για οποιοδήποτε λόγο η `remove()` δεν καταφέρει να αφαιρέσει φυσικά τον κόμβο, δεν ξαναπροσπαθεί γιατί θα επιχειρήσουν και τα επόμενα νήματα όσο διατρέχουν την λίστα.

Με αυτές τις αλλαγές, επιτυχώς υλοποιήσαμε μια λίστα η οποία δεν έχει ανάγκη από lock για να προσφέρει safe concurrency. Η απαλειφή των κλειδαριών επίσης προσφέρει και μια πιο ανθεκτική δομή η οποία δεν κινδυνεύει από αποτυχία σε ενδεχόμενο ”θανάτου” νήματος που κρατάει lock.

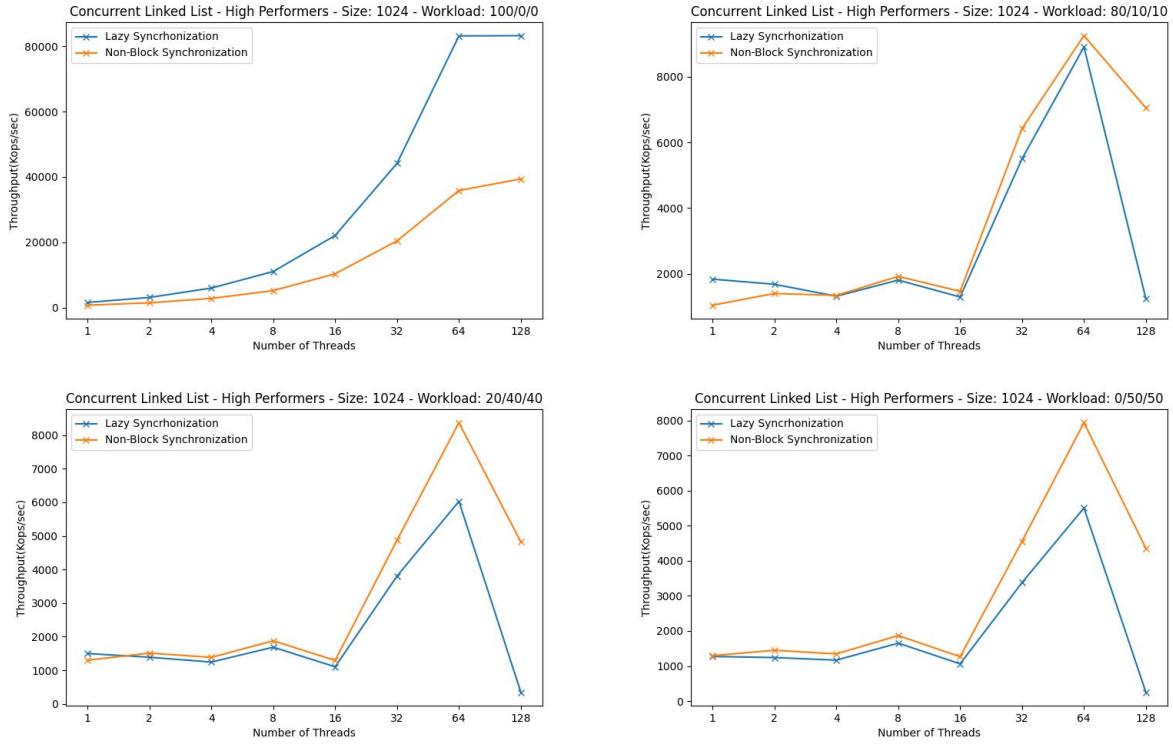


Figure 36: Concurrent Linked List - Throughput - Size 1024 - High Performers

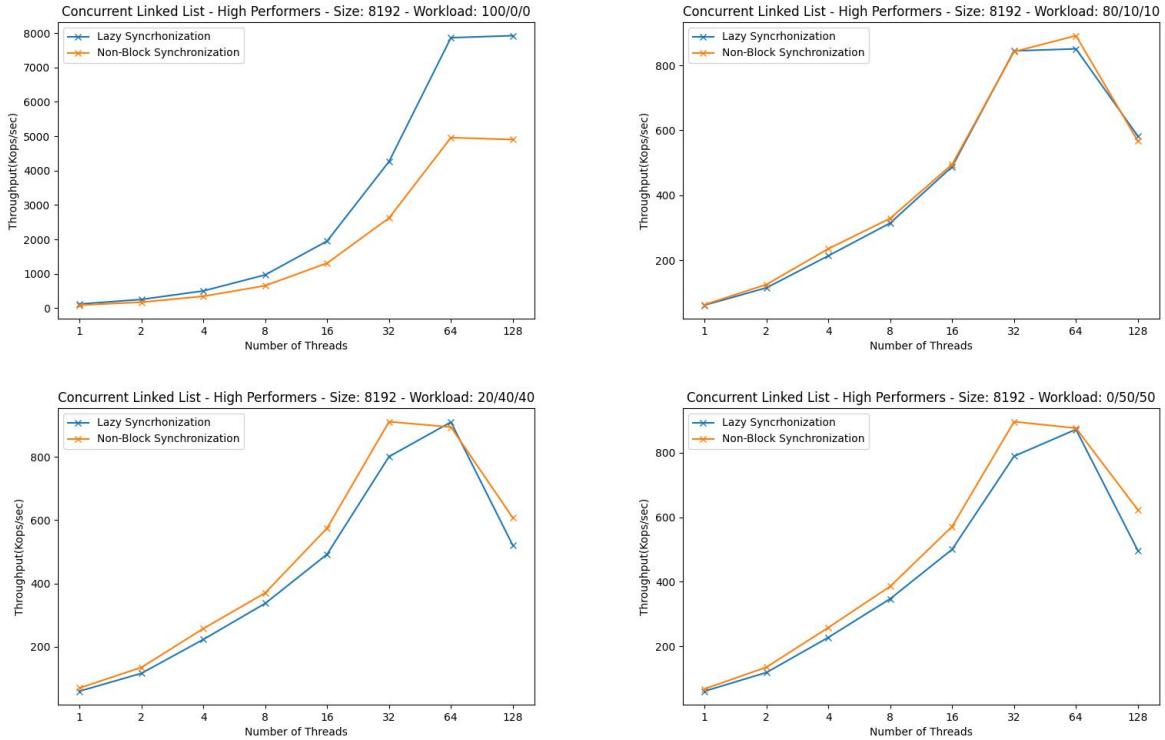


Figure 37: Concurrent Linked List - Throughput - Size 8192 - High Performers

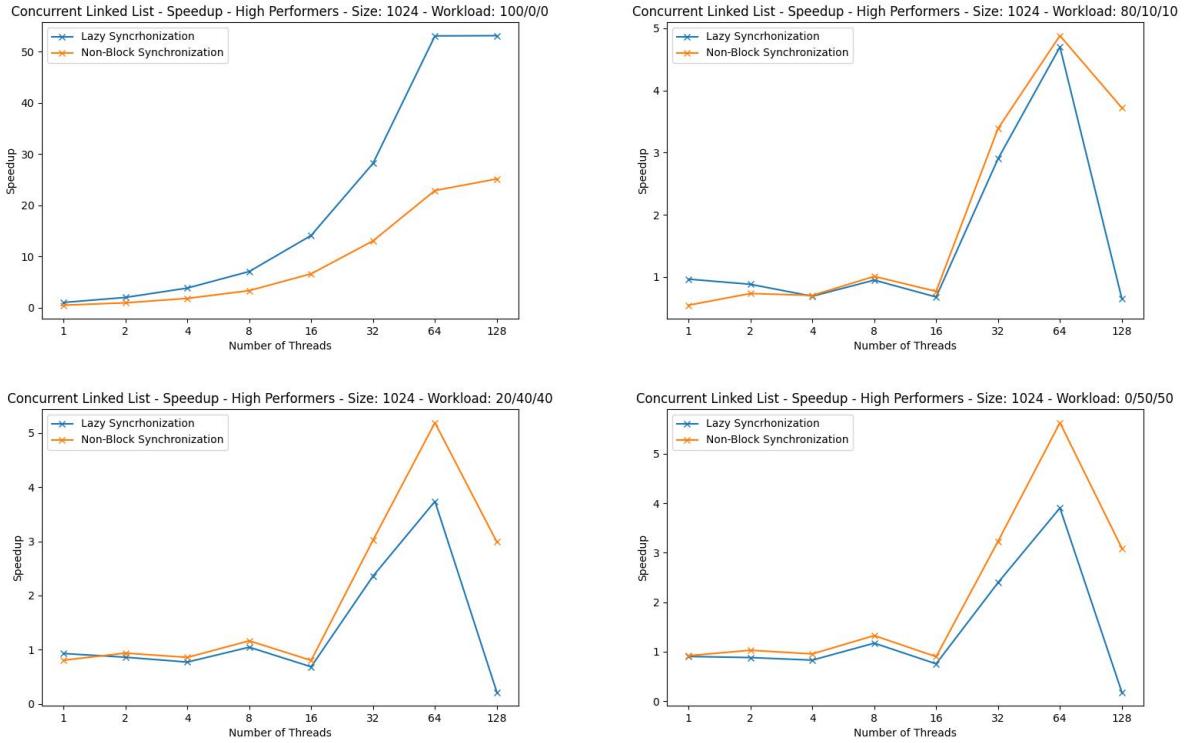


Figure 38: Concurrent Linked List - Speedup - Size 1024 - High Performers

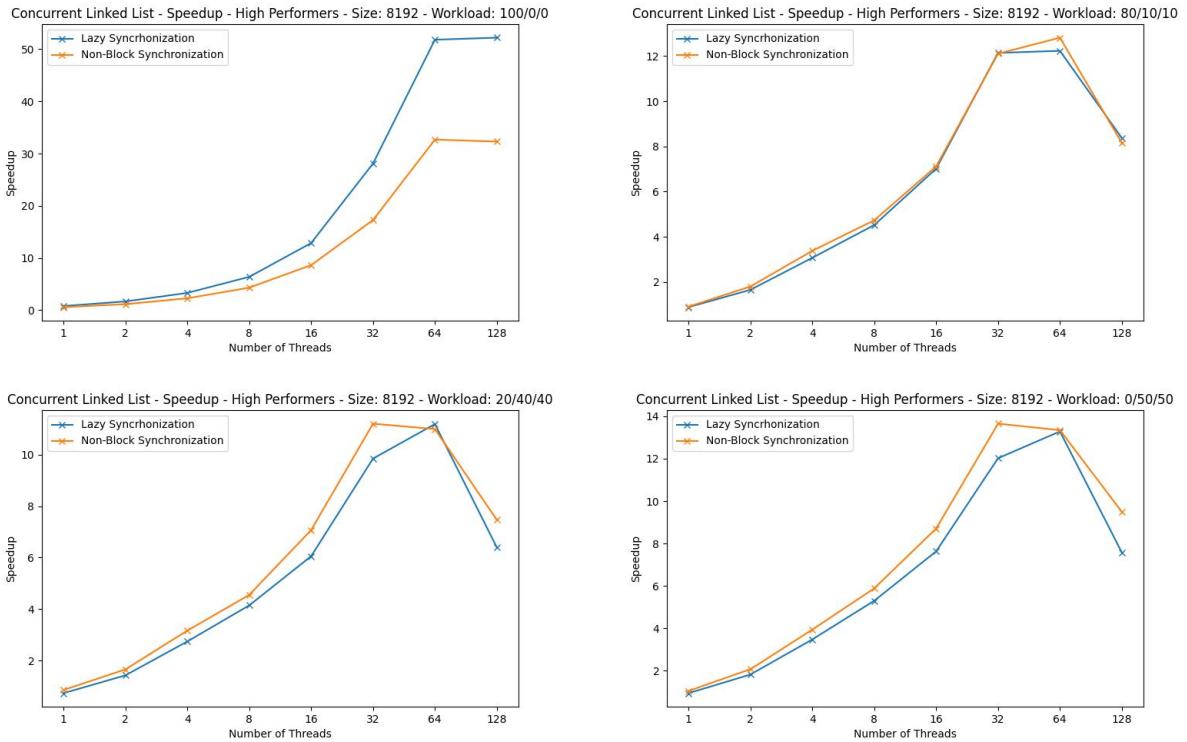


Figure 39: Concurrent Linked List - Speedup - Size 8192 - High Performers

## High Performers - Συμπεράσματα

### Workload

Για το πρώτο workload που αποτελείται μόνο από contains(), η κλιμάκωση της υλοποίησης είναι σχεδόν ακριβώς γραμμική για τα πρώτα 64 νήματα. Οι παράμετροι που μελετάμε αυτό το workload είναι λίγο ”υπερβολικά ιδανικοί” καθώς δεν υπάρχουν marked κόμβοι στο πρόβλημα. Αν και οι μέθοδοι contains() είναι ολόιδιοι στην θεωρία, στις συγκεκριμένες υλοποιήσεις η non-block εκδοχή ελέγχει και αν ο δεξιά κόμβος είναι marked ή όχι. Αυτός ο έξτρα έλεγχος που κάνει ευθύνεται και για την μεγάλη διαφορά που έχουν στο throughput αυτές οι 2 εκδοχές στα περισσότερα νήματα. Το μέγιστο speedup είναι 55 για την lazy και 26 για την non-blocking.

Για τα υπόλοιπα workloads που περιέχουν και add()/remove(), τα αποτελέσματα είναι πάλι αρκετά κοντά μεταξύ τους. Για την μεγάλη λίστα, οι 2 εκδοχές είναι πρακτικά πανομοιότυπες. Μέχρι τα 16 νήματα, η απόδοση είναι κοντά στην σειριακή. Αυξάνοντας περαιτέρω τα νήματα, το throughput των λιστών ”εκτοξεύεται” απευθείας, διπλασιάζοντας με κάθε περαιτέρω διπλασιασμό των νημάτων. Αυτό ισχύει μέχρι τα 128 νήματα που καταφέρει η απόδοση αφού δεν μπορεί να προσφέρει άλλους πυρήνες το σύστημα. Πρέπει πλέον να δρομολογούνται 128 λογικά νήματα σε 64 φυσικά νήματα. Αυτό το έξτρα overhead καταστρέφει την απόδοση, ειδικά για την lazy λίστα αφού θα αυξάνονται και τα conflicts για απόκτηση των κλειδαριών.

**Σημείωση:** Αν και στην θεωρία η μέθοδος contains() δεν καθαρίζει κόμβους στην non-blocking λίστα, στην υλοποίηση που μας δίνεται αυτό δεν ισχύει. Η συνάρτηση list\_search είναι κοινή και για τις 3 μεθόδους και εκτελεί έλεγχο για καθαρισμό marked κόμβων.

### Size

Το μέγεθος της λίστας έχει αρκετά σημαντικό ρόλο στην απόδοση καθώς μεγαλύτερη λίστα σημαίνει λιγότερα conflicts, τουλάχιστον για τις συγκεκριμένες λίστες αφού η διάσχιση είναι wait-free. Για μικρές λίστες, η non-blocking λίστα έχει ένα προβάδισμα στην απόδοση ενώ στην μεγαλύτερη λίστα, είναι πολύ κοντά οι αποδόσεις. Το μικρό αυτό προβάδισμα το αποκτάει η non-blocking καθώς η lazy λίστα έχει ακόμα το ζήτημα των lock conflict. Το φαινόμενο αυτό είναι πολύ πιο σπάνιο στην μεγάλη λίστα.

Αν και η lazy λίστα έχει το overhead του κλειδώματος, η non-blocking λίστα έχει το overhead του compare-and-set. Η lazy λίστα δεν καθιστά σύγουρη την πρόοδο των νημάτων καθώς οι μέθοδοι add() και remove() είναι τύπου blocking αλλά δεν απαιτεί από κόμβο να έχει ατομική αναφορά ή να κάνει ατομικό έλεγχο μεταβλητών που ”κοστίζει” σε απόδοση. Περαιτέρω, η non-blocking λίστα απαιτεί να κάνει έλεγχο και διαγραφή των marked κόμβων όσο διατρέχει την λίστα, ενέργεια που μπορεί να προκαλέσει conflict μεταξύ νημάτων και να αναγκάσει restart της διαδικασίας.

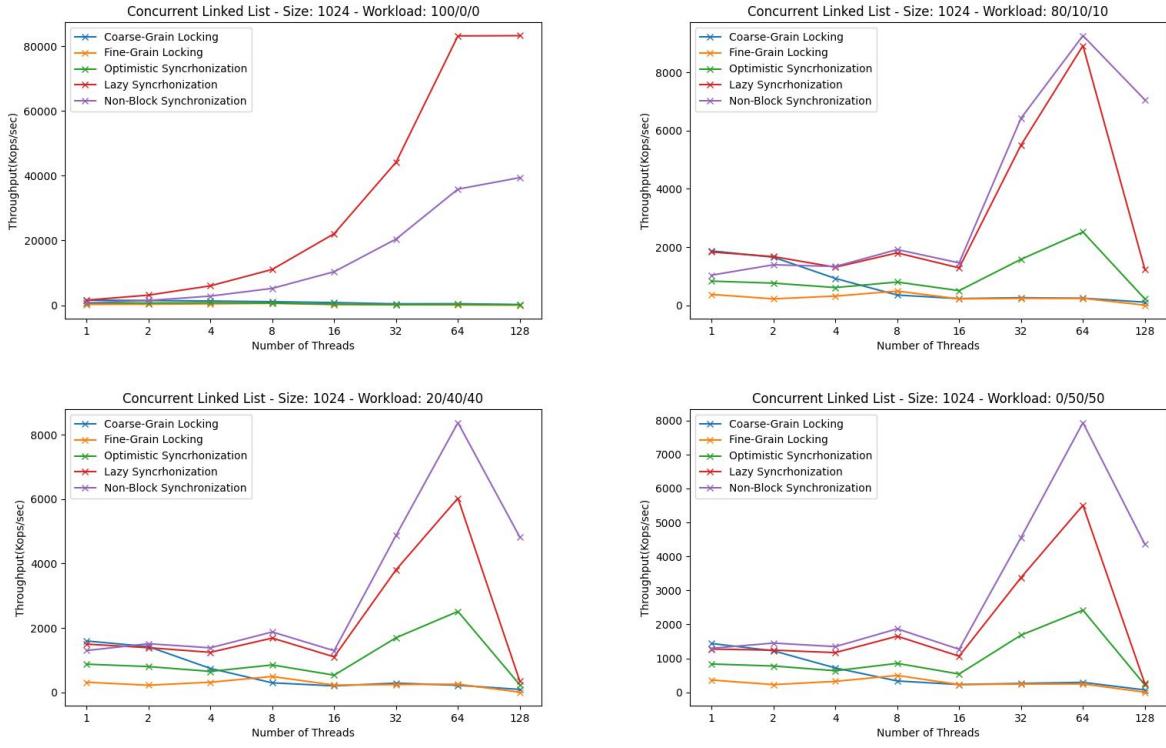


Figure 40: Concurrent Linked List - Throughput - Size 1024 - All Synchronization Techniques

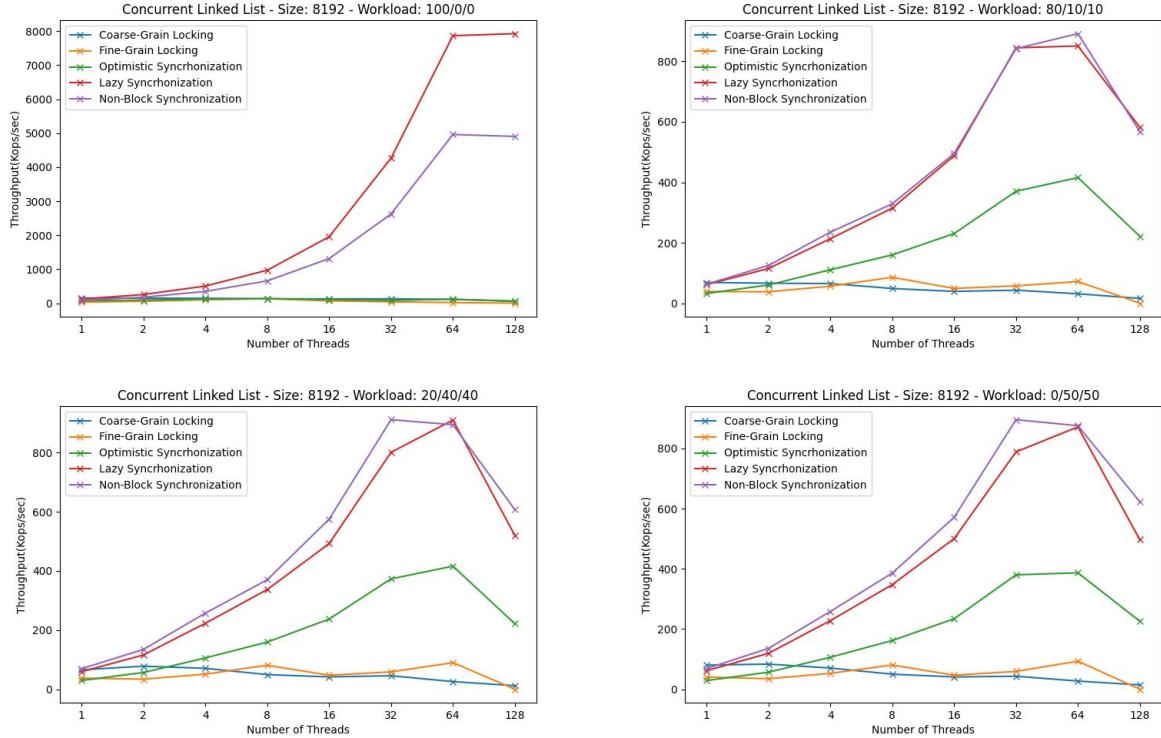


Figure 41: Concurrent Linked List - Throughput - Size 8192 - All Synchronization Techniques

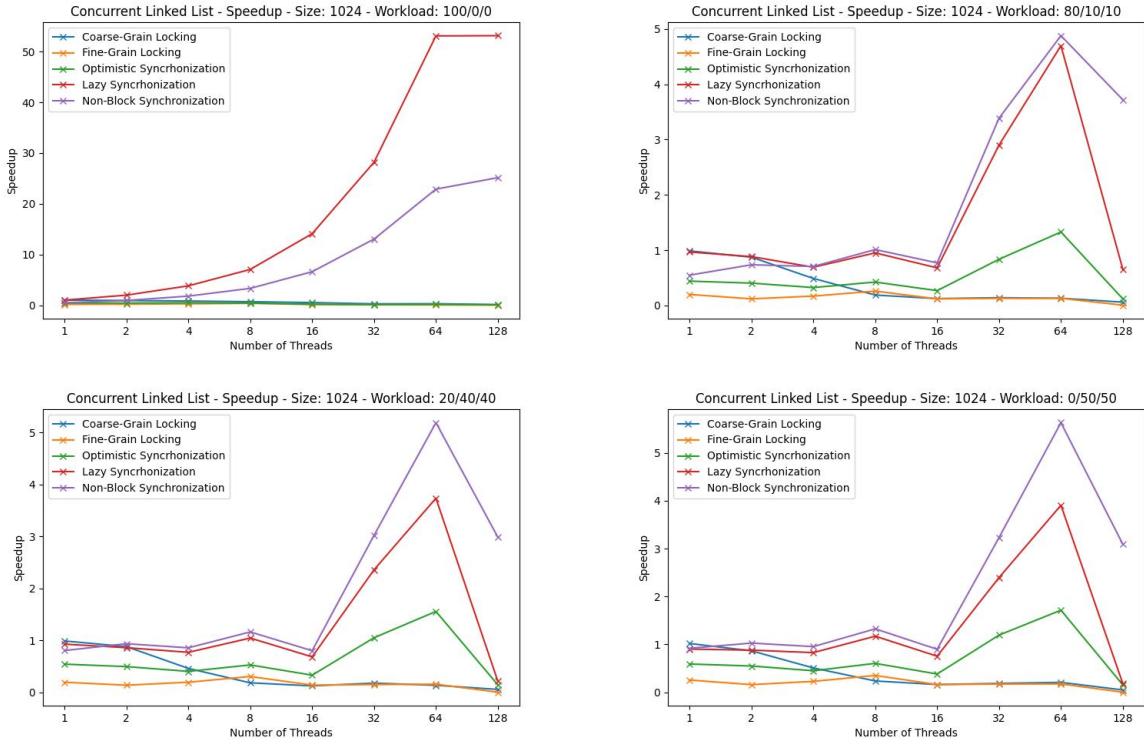


Figure 42: Concurrent Linked List - Speedup - Size 1024 - All Synchronization Techniques

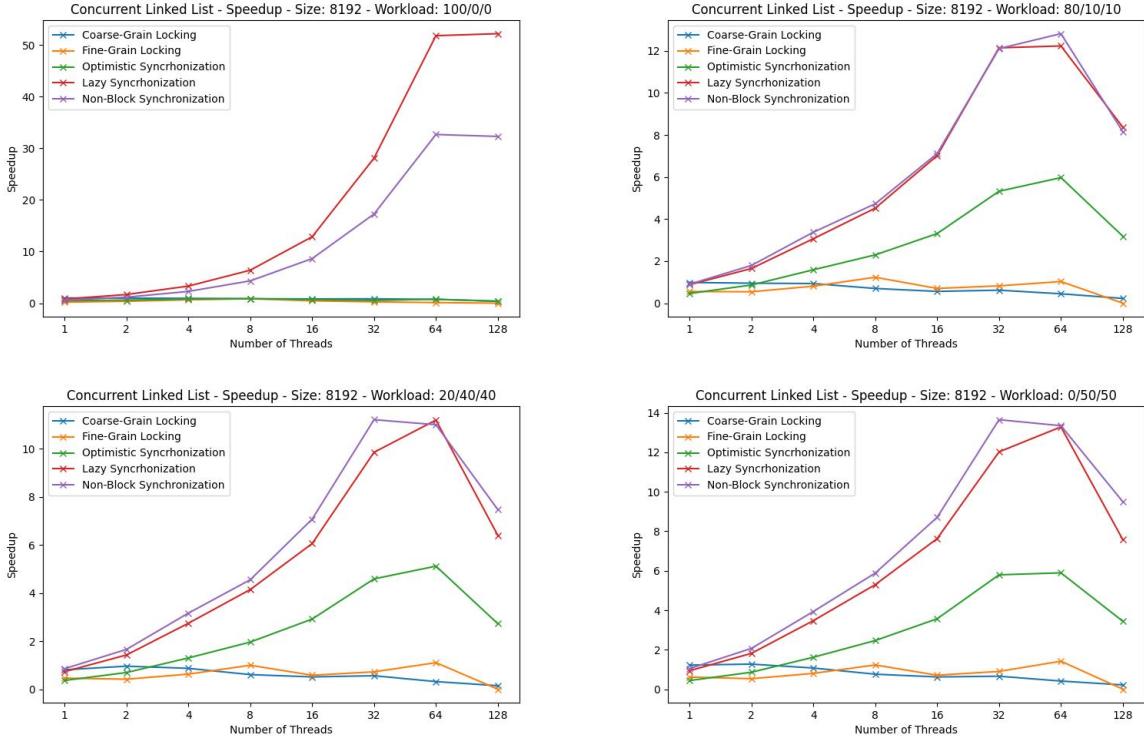


Figure 43: Concurrent Linked List - Speedup - Size 8192 - All Synchronization Techniques

## Τελικά Συμπεράσματα

Οι coarse-grained, fine-grained και optimistic synchronization λίστες έχουν αρκετά θεωρητικά προβλήματα που παρουσιάζονται και στα αποτελέσματα των πειραμάτων. Οι δύο πρώτες δεν καταφέρνουν να ξεπεράσουν τις επιδόσεις της σειριακής δομής ενώ η τρίτη έχει μέτριες επιδόσεις, ειδικά σε μικρές λίστες. Σε μεγάλες λίστες, τα συμπεράσματα είναι λίγο διαφορετικά για την optimistic λίστα, με κάποιες μικρές προοπτικές για κλιμάκωση.

Σε αντίθεση, η lazy και η non-blocking λίστα είναι πιο εκλεπτυσμένες λύσεις. Δεν μπορούμε να πούμε ότι η non-blocking είναι βελτίωση της lazy καθώς η μία μπορεί να αντικαταστήσει την άλλη, αναλόγως την εφαρμογή. Η non-blocking είναι robust λύση χωρίς locks αλλά με μεγαλύτερο performance drawback λόγω των συνεχόμενων compare-and-set. Αντιθέτως, η lazy έχει locks στις μεθόδους add()/remove() όμως είναι πιο γρήγορη σε workload που αποτελούνται μόνο από contains().

**Σημείωση:** Πιστεύουμε ότι και η non-blocking θα είχε ίδια απόδοση με την lazy στο 100/0/0 workload αν δεν αναγκάζοταν να διαγράψει τα marked nodes, όπως κάνει στην δοσμένη υλοποίηση.

‘=’

## 5 Άσκηση 5 - Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Αντικείμενο μελέτης της συγκεκριμένης άσκησης ήταν η εκτέλεση του αλγορίθμου K-means σε αρχιτεκτονική GPU. Μελετάμε διαφορετικές τεχνικές με τις οποίες μπορούμε να δομήσουμε ένα πρόγραμμα που θα εκτελεστεί σε GPU και πως μπορούμε να αξιοποιήσουμε όλο το υλικό για να μειώσουμε τον χρόνο εκτέλεσης, κρατώντας την GPU συνεχώς busy.

### 5.1 Τεχνικές υλοποίησης αλγορίθμων που εκτελούνται σε αρχιτεκτονικές GPU

Στον προγραμματισμό με χρήση επιταχυντών, ο κλασσικός επεξεργαστής συνεχίζει να εκτελεί σημαντικό έργο, αρχικοποιώντας και μεταφέροντας δεδομένα.

Ο χρόνος σειριακής εκτέλεσης:

- {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} είναι ίσος με 18446.2ms
- {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} είναι ίσος με 5484.8ms

**Σημείωση:** Όλος ο κώδικας της άσκησης βρίσκεται σε ξεχωριστό .zip αρχείο στο Helios της ομάδας.

### 5.2 K-Means Naive

Η πρώτη υλοποίηση είναι και η πιο απλοϊκή σε σκέψη. Δρομολογούμε όσα νήματα όσα και τα στοιχεία του πίνακα στον οποίο εκτελούμε τον αλγόριθμο K-means. Κάθε νήμα, με συγκεκριμένο thread ID, βρίσκει την κοντινότερη απόσταση από κάποιο κέντρο καλώντας την συνάρτηση euclid\_dist\_2. Τέλος, αυξάνουμε το μέγεθος membership του κέντρου αυτού.

Οι διαστάσεις του πίνακα objects, είναι αριθμός\_αντικειμένων x αριθμός\_συντεταγμένων.

Εφόσον τα warps, δηλαδή οι ομάδες των νημάτων, είναι πολλαπλάσια των 32, κάνουμε έναν έλεγχο ορίων ώστε αν δρομολογηθούν νήματα με thread ID μεγαλύτερο από τον αριθμό των αντικειμένων στον πίνακα, να μην δράσουν ώστε να μην επιχειρήσουν να αναζητήσουν στοιχείο σε διεύθυνση μνήμης που δεν έχει γίνει allocated.

Μεταξύ κάθε iteration, μεταφέρουμε από την CPU στην GPU τους πίνακες των κέτρων (Clusters) και από την GPU στην CPU τον πίνακα membership, ώστε να γίνει η ανανέωση των κέτρων των cluster. Η CPU αναλαμβάνει την ανανέωση αυτή.

## K-Means Naive - Αποτελέσματα

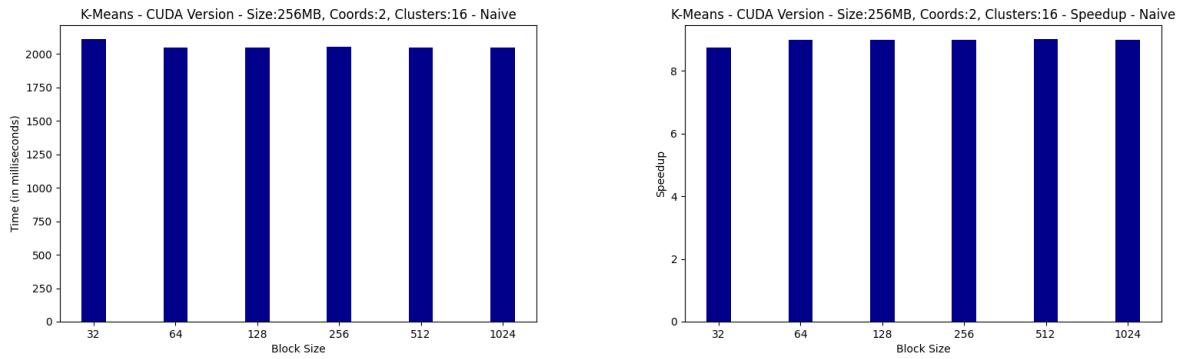


Figure 44: K-Means Naive - GPU Edition

## K-Means Naive - Συμπεράσματα

Η επίδοση της παράλληλης έκδοσης είναι εμφανώς καλύτερη από την σειριακή. Δεδομένου την απλότητα της υλοποίησης, το speedup είναι αρκετά αξιοσημείωτο. Περαιτέρω, οι παράλληλες εκδόσεις με GPU έχουν το έξτρα overhead της μεταφοράς δεδομένων. Οι μετρήσεις που κάναμε λαμβάνουν υπόψιν τους το φανόμενο αυτό, ώστε να είναι δίκαιη η σύγκριση με την σειριακή υλοποίηση.

To block size δεν φαίνεται να επηρεάζει τα αποτελέσματα σε αυτή την υλοποίηση. Με βάση το CUDA Occupancy Calculator, προτιμούμε block size ίσο με 128 καθώς είναι το πιο ισορροπημένο που δίνει occupancy ίσο με 100%.

**Σημείωση:** Ο υπολογισμός του occupancy έγινε με βάση το αρχείο spreadsheet της NVIDIA. Δεν είναι πλήρως ορθό χριτήριο για την επιλογή του block size. Πιο μοντέρνοι - application specific τρόποι υπολογισμού του occupancy είναι προτιμότερες επιλογές.

## K-Means Naive - Best Time

2051.098108ms - 128 Block Size

## 5.3 K-Means Transpose

Το πρώτο βήμα προς την βελτιστοποίηση είναι να ακολουθήσουμε μια πιο architectural aware προσέγγιση. Το προφανές πρόβλημα της naive υλοποίησης είναι η κακή πρόσβαση στην μνήμη.

Εδώ επιχειρούμε να λύσουμε αυτό το πρόβλημα δίνοντας παραπάνω σημασία στο memory coalescing. Η υλοποίηση αυτή είναι ίδια με την σημαντική διαφορά ότι αλλάζουμε την λίστα των δεδομένων objects και clusters από row-based σε column-based indexing. Ανανεώνουμε και την συνάρτηση εύρεσης απόστασης μεταξύ 2 σημείων σε column-based δεικτοδότηση.

## K-Means Transpose - Αποτελέσματα

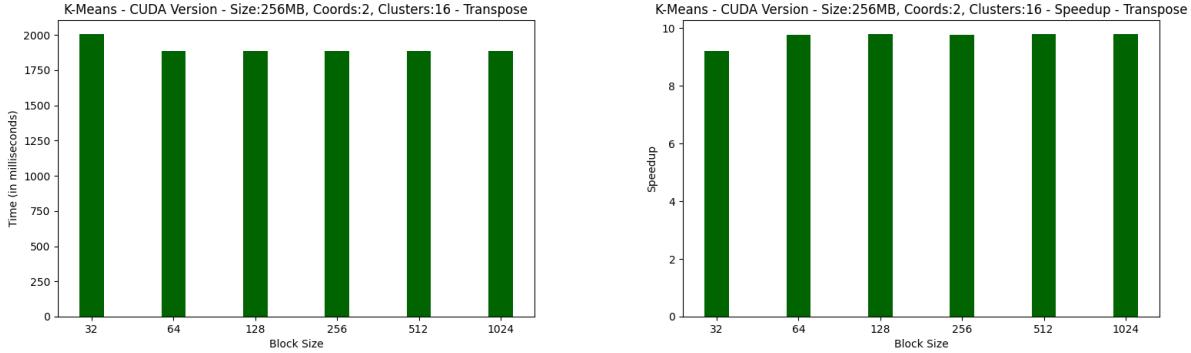


Figure 45: K-Means Transpose - GPU Edition

## K-Means Transpose - Συμπεράσματα

Παρατηρούμε αρκετά καλύτερα αποτελέσματα από την Naive έκδοση. Η αλλαγή στην συμπεριφορά του προγράμματος είναι αρκετά μικρή αλλά είναι επίσης πιο architectural aware. Στην προηγούμενη έκδοση, η πρόσβαση στην μνήμη ήταν συνεχόμενη αλλά μη ευθυγραμμισμένη. Για μια δεδομένη σειρά νημάτων που έκανε πρόσβαση στον πίνακα Clusters (με διαστάσεις x:numClusters, y:numCoords), η πρώτη θέση μνήμης που έκανε πρόσβαση το κάθε νήμα ήταν μια σειρά μωριά. Άρα, γινόντουσαν πολλαπλά memory transactions για να υπολογιστούν οι αποστάσεις. Το ίδιο πρόβλημα παρουσιάζοταν και στον πίνακα των αντικειμένων (Objects, με διαστάσεις x:numObjs, y:numCoords).

Το πρόβλημα αυτό λύνεται κάνοντας τις προσβάσεις στην μνήμη συνεχόμενες KAI ευθυγραμμισμένες. Αυτό πετυχαίνεται κάνοντας transpose τους προαναφερόμενους πίνακες.

Το block size πάλι δεν φαίνεται να επηρεάζει σημαντικά τα αποτελέσματα. Με εξαίρεση το blockSize=32, οι υπόλοιπες επιλογές παρουσιάζουν πρακτικά ολόιδια αποτελέσματα. Επιλέγουμε ως βέλτιστο το blockSize=128, για τον ίδιο λόγο που εξηγήσαμε και στην Naive έκδοση (occupancy).

## K-Means Transpose - Best Time

1883.369207ms - 128 Block Size

## 5.4 K-Means Shared Memory

Το τελευταίο άμεσο βήμα βελτιστοποίησης είναι να εκμεταλευτούμε παραπάνω το ίδιο το υλικό. Οι **Streaming Multiprocessors** της κάρτας γραφικών περιέχουν shared memory. Η ειδική

αυτή μνήμη, δίνει την δυνατότητα στα νήματα ίδιων block να προσπελαύνουν δεδομένα πιο γρήγορα. Στην υλοποίηση αυτή, τοποθετούμε τον πίνακα cluster, δηλαδή τον πίνακα που περιέχει τις συντεταγμένες των κέντρων των cluster, στην διαμοιραζόμενη μνήμη. Η υπόλοιπη υλοποίηση είναι ολόδια με την Transpose.

## K-Means Shared Memory - Αποτελέσματα

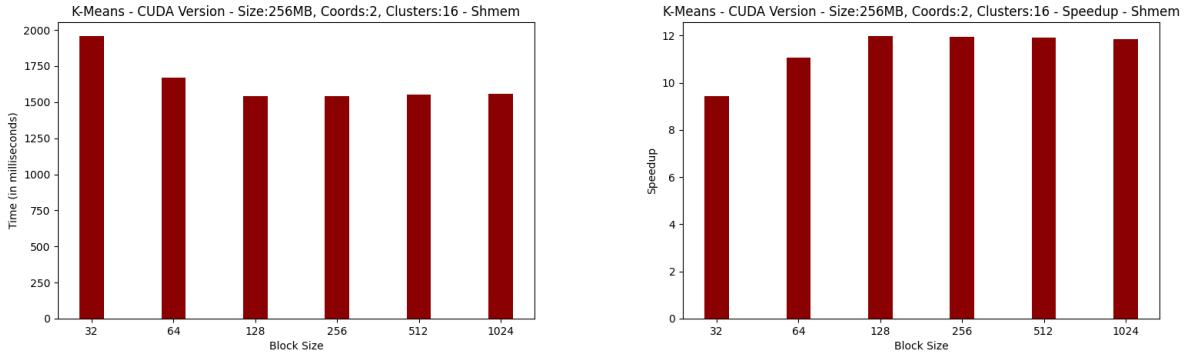


Figure 46: K-Means Shared - GPU Edition

## K-Means Shared Memory - Συμπεράσματα

Τα αποτελέσματα είναι τα βέλτιστα συγχριτικά με τις 3 υλοποιήσεις. Η shared memory είναι μια πιο ελεγχόμενη μορφή μνήμης cache, στην οποία μπορούμε αναλόγως το πρόβλημα, να τοποθετήσουμε και διαφορετικά δεδομένα, συγκεκριμένα αυτά που χρησιμοποιούνται πιο συχνά. Σημαντικό είναι να τοποθετούμε τα δεδομένα με σωστό τρόπο ώστε να αποφευχθούν τα bank conflicts.

Η μείωση του χρόνου είναι λογική αφού χρησιμοποιούμε μνήμη πιο πάνω στην ιεραρχία από αυτήν που χρησιμοποιούσαμε στην Transpose/Naive έκδοση. Πλέον, οι προσπελάσεις γίνονται γρηγορότερα. Η μοιραζόμενη μνήμη είναι κοινή για νήματα που βρίσκονται στο ίδιο block. Με βάση το occupancy calculator, και στα 2 configurations που μελετάμε, γίνεται υψηλή χρήση των SM.

To block size συνεχίζει να μην παίζει κάποιο ρόλο στην εκτέλεση του προγράμματος, βέλτιστο είναι πάλι οτιδήποτε πάνω ή ίσο με 128 νήματα.

## K-Means Shared Memory - Best Time

1541.312933ms - 128 Block Size

## 5.5 Σύγκριση υλοποιήσεων / bottleneck Analysis / μελέτη άλλων configurations

Από τις 3 υλοποιήσεις, λογικό η καλύτερη να είναι αυτή που αξιοποιεί και την μοιραζόμενη μνήμη αλλά και το memory coalescing, δηλαδή, η Shared Memory version. Οι χρόνοι εκτέλεσεις και το speedup παρουσιάζονται σε κοινό διάγραμμα και για τις 3 εκδόσεις παρακάτω.

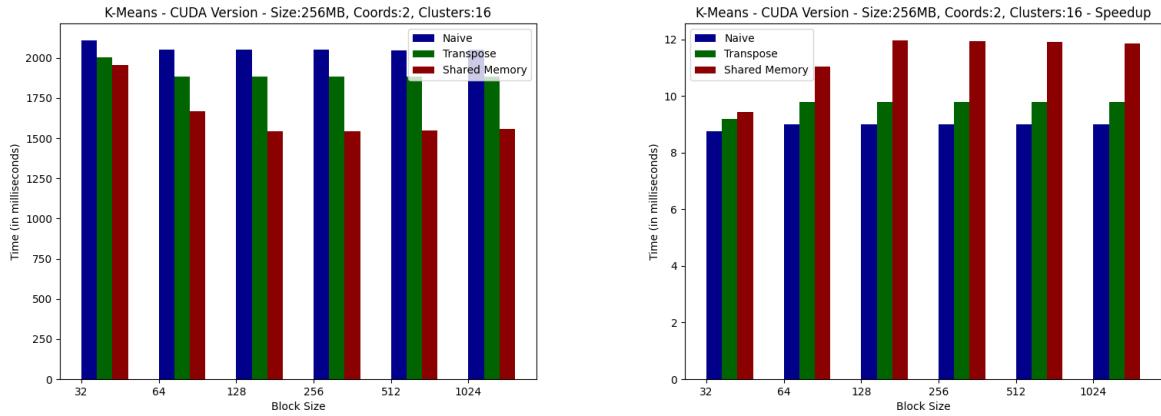


Figure 47: K-Means - GPU Edition Size - numCoords=2

## Bottleneck Analysis

Προσθέτουμε 3 ζεχωριστά timers (CPU, Transfer, GPU) για να μελετήσουμε πιο μέρος του συστήματος απαιτεί τον περισσότερο χρόνο. Οι πίνακες είναι κατά μέσο όρο για όλα τα block sizes που μελετάμε και σε millisecond.

Type \ Timers	CPU Timer	Transfer Timer	GPU Timer
Type			
Naive	887.971	406.153	765.745
Transpose	797.969	406.419	701.031
Shared Memory	796.91	406.115	433.498

Το μέγευθος της μεταφοράς των δεδομένων είναι ίδιο και για τις 3 υλοποιήσεις και δεν μπορούμε να επέμβουμε για να βελτιωθεί με κάποιον τρόπο. Είναι αναγκαίο κακό του προγραμματισμού ετερογενών συστημάτων.

Διαδοχικά, μειώνεται ο χρόνος εκτέλεσης του μέρους που εκτελείται από την GPU καθώς βελτιώνονται οι υλοποιήσεις.

Το μέρος που εκτελείται στον επεξεργαστή είναι περίπου σταθερό, παρατηρούμε όμως μια μικρή μείωση, κοντά στα 10%, μεταξύ της Naive έκδοσης και των Transpose/Shared Memory. Αυτή δικαιολογείται γιατί πλέον ο υπολογισμός των νέων κέντρων γίνεται με βάση τον transposed πίνακα, που είναι ελάχιστα πιο cache-friendly (τουλάχιστον για το συγκεκριμένο configuration).

Το κυριότερο bottleneck είναι η εκτέλεση στον CPU και η μεταφορά των δεδομένων. Ο συνολικός χρόνος εκτέλεσης μπορεί να μειωθεί περαιτέρω αν μεταφερθεί όλη η εκτέλεση του αλγορίθμου στην GPU. Έτσι, περιορίζεται δραστικά η συνεχόμενη μεταφορά δεδομένων μεταξύ host και device. Επίσης, αξιοποιούμε το γεγονός ότι μπορεί, έστω στοιχειωδώς, να παραλληλοποιηθεί το task εύρεσης νέων κέντρων.

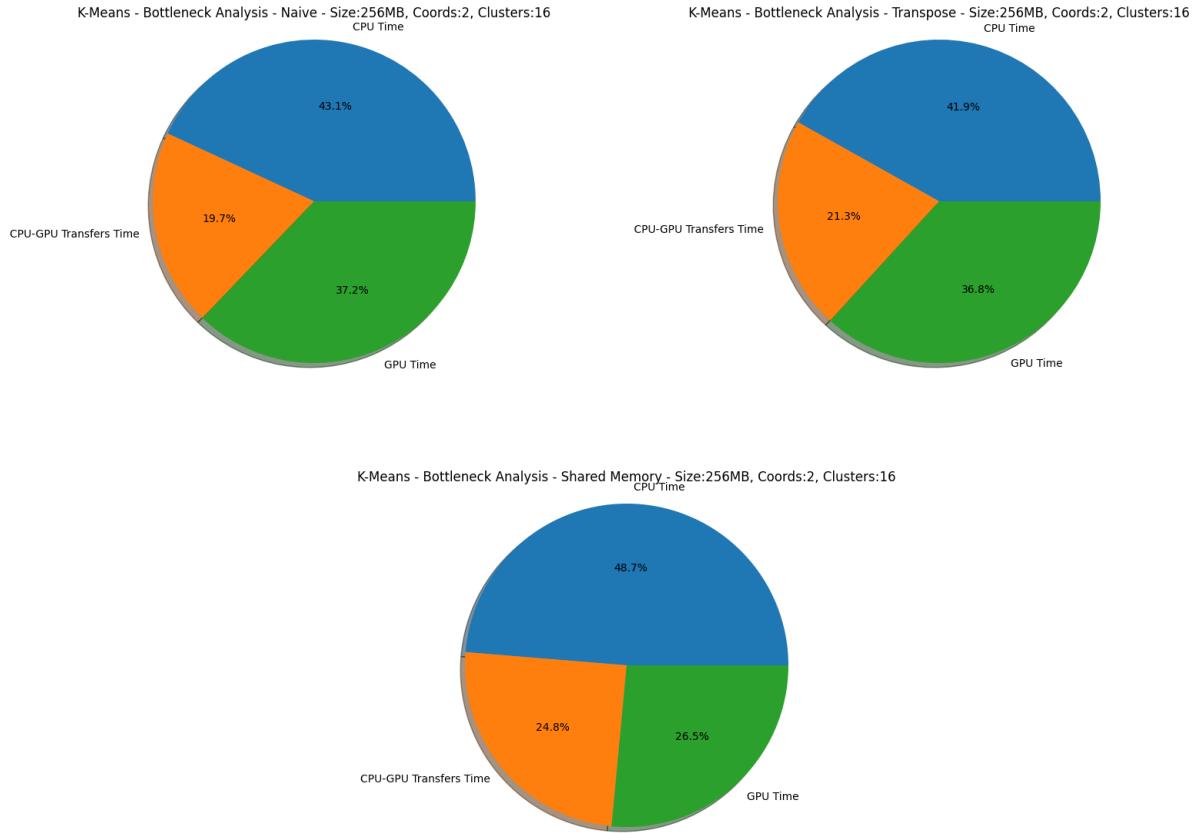


Figure 48: K-Means - Bottleneck Analysis - numCoords=2

### Μελέτη Άλλων Configuration

Δημιουργούμε τα γραφήματα των ίδιων υλοποιήσεων για το configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}. Τα "σφάλματα" στην σχεδίαση του Naive αλγορίθμου είναι πιο ξεκάθαρα εδώ. Το γεγονός ότι δεν εκμεταλέυεται την ικανότητα των GPU για memory coalescing και, αντιθέτως, απαιτούνται πολλαπλά transactions για τον υπολογισμό των αποστάσεων, περιορίζει πολύ την επίδοσή της.

Η transpose έκδοση είναι πάνω από 5 φορές καλύτερη στο GPU μέρος αλλά 2 φορές χειρότερη στο CPU μέρος. Η χειροτέρευση του CPU μέρους εξηγείται από το γεγονός ότι υπολογίζουμε τα νέα κέντρα με column-based δεικτοδότηση η οποία δεν είναι τόσο cache-friendly (τουλάχιστον στο συγκεκριμένο configuration).

Το κόστος της μεταφοράς είναι πάλι κοινό για όλες τις μεταφορές. Η shared έκδοση μπορεί να βελτιωθεί στο κομμάτι που αναλαμβάνει η CPU. Μια καλή λύση θα ήταν να ξαναγίνεται transpose ο πίνακας πριν μεταφερθεί στην CPU, καθώς η GPU μπορεί να κάνει αυτό το transpose πολύ πιο σύντομα.

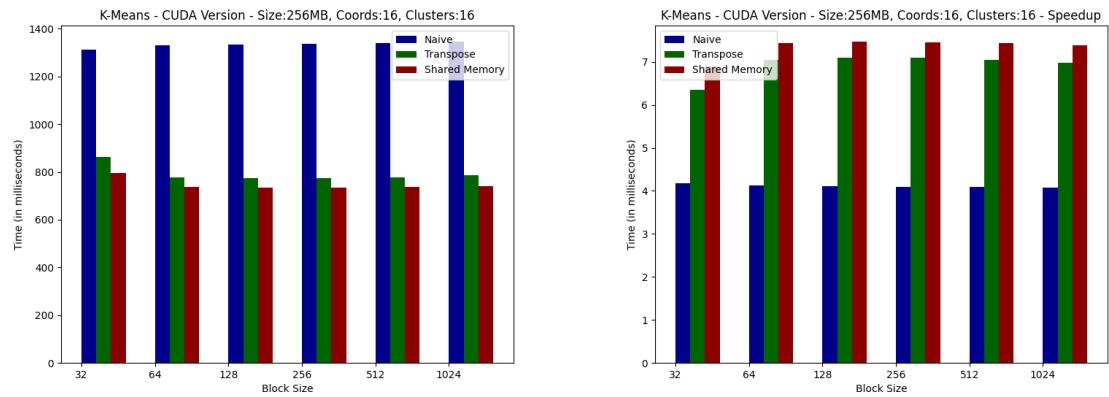


Figure 49: K-Means - GPU Edition Size - numCoords=16

Type \ Timers	CPU Timer	Transfer Timer	GPU Timer
Naive	214.842	53.018	1065.877
Transpose	550.324	53.009	188.571
Shared Memory	549.727	53.006	144.105

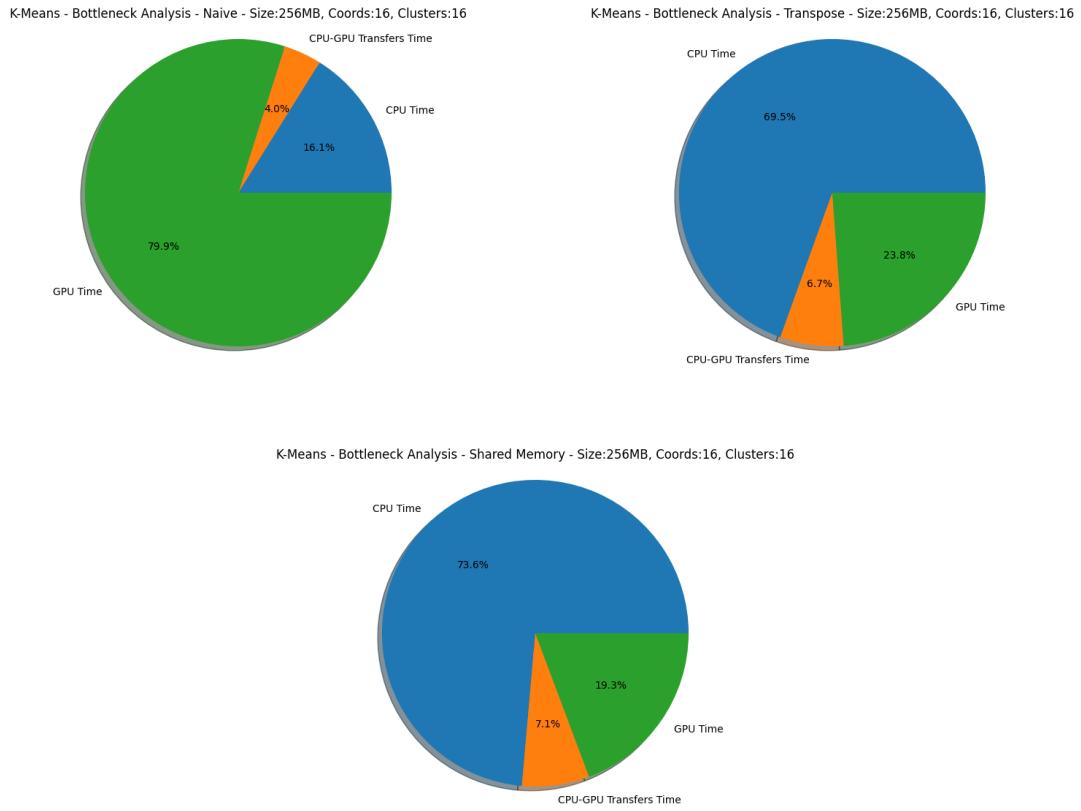


Figure 50: K-Means - Bottleneck Analysis - numCoords=16

## 5.6 K-Means All GPU

Ακόμα work in progress...

# 6 Άσκηση 6 - Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

Κέντρο της συγκεκριμένης άσκησης ήταν η επίλυση του προβλήματος της διάδοσης της θερμότητας και συγκεκριμένα 3 από τις μεθόδους/αλγορίθμους που χρησιμοποιούνται για την επίλυση μερικών διαφορικών εξισώσεων.

Η επίλυση της εξίσωσης διάδοσης της θερμότητας θα γίνει σε αρχιτεκτονικές κατανεμημένης μνήμης με χρήση προγραμματιστικού μοντέλου MPI.

### 6.1 Περιγραφή των μεθόδων

Κάθε μέθοδος εκτελείται πάνω σε έναν 2D-πίνακα διαστάσεων MxN. Διατρέχουμε ολόκληρο τον πίνακα, στοιχείο-στοιχείο για να υπολογίσουμε την τρέχουσα τιμή του πίνακα.

#### 6.1.1 Μέθοδος Jacobi

Η πιο απλή μέθοδος επίλυσης της διαφορικής εξίσωσης μετάδοσης θερμότητας. Για κάθε σημείο, παίρνουμε κατά μέσο όρο το άθροισμα των 4 γειτονικών σημείων. Αυτή η διαδικασία επαναλαμβάνεται ”χρονικά” μέχρι να συγκλίνει η μέθοδος.

---

```
void Jacobi(double ** u_previous, double ** u_current, int X_min, int X_max, int Y_min, int Y_max) {
    int i,j;
    for (i=X_min;i<X_max;i++)
        for (j=Y_min;j<Y_max;j++)
            u_current[i][j]=(u_previous[i-1][j]+u_previous[i+1][j]+u_previous[i][j-1]
                +u_previous[i][j+1])/4.0;
}
```

---

#### 6.1.2 Μέθοδος Gauss-Seidel με χρήση Successive Over-Relaxation

Προχωράμε στην χρήση της μεθόδου successive over-relaxation, η οποία χρησιμοποιείται για να αυξήσουμε τον ρυθμό σύγκλισης σε iterative διεργασίες.

Εκμεταλευόμαστε το γεγονός ότι σε κάθε σημείο, οι γείτονες σε θέση west,north έχουν ήδη υπολογίσει την τιμή τους για την παρούσα χρονική στιγμή. Άρα, αντί να χρησιμοποιήσουμε και από τους 4 γείτονες τις τιμές της προηγούμενης χρονικής στιγμής, χρησιμοποιούμε την τιμή της παρούσας χρονικής στιγμής όπου είναι δυνατό. Αυτό, μαζί με την χρήση του relaxation factor omega, προσφέρει πολύ καλύτερη επίδοση στο θέμα της σύγκλισης.

---

```
void GaussSeidel(double ** u_previous, double ** u_current, int X_min, int X_max, int Y_min, int Y_max, double omega) {
```

```

int i,j;
for (i=X_min;i<X_max;i++)
    for (j=Y_min;j<Y_max;j++)
        u_current[i][j]=u_previous[i][j]+(u_current[i-1][j]+u_previous[i+1][j]
            +u_current[i][j-1]+u_previous[i][j+1]-4*u_previous[i][j])*omega/4.0;
}

```

---

### 6.1.3 Μέθοδος Red-Black ordering με χρήση Successive Over-Relaxation

Η μέθοδος αυτή εκτελείται σε 2 φάσεις. Στην πρώτη φάση, υπολογίζουμε τις τιμές του πίνακα που βρίσκονται σε άρτια θέση (Update red phase) χρησιμοποιώντας τις γειτονικές τιμές της προηγούμενης χρονικής στιγμής.

Στην επόμενη φάση (Update black phase) υπολογίζουμε τις τιμές του πίνακα που βρίσκονται σε περιττή θέση χρησιμοποιώντας τις γειτονικές τιμές της παρούσας χρονικής στιγμής. Αυτό είναι εφικτό αφού όλοι οι γείτονες μιας περιττής θέσης (black) θα είναι τιμές που βρίσκονται σε άρτιες θέσεις (red).

Η μέθοδος αυτή προσφέρει καλύτερο ρυθμό σύγχλισης συγχριτικά με την Gauss-Seidel αλλά έχει 2 στάδια υπολογισμού αντί για 1.

```

void RedSOR(double ** u_previous, double ** u_current, int X_min, int X_max, int
Y_min, int Y_max, double omega) {
    int i,j;
    for (i=X_min;i<X_max;i++)
        for (j=Y_min;j<Y_max;j++)
            if ((i+j)%2==0)
                u_current[i][j]=u_previous[i][j]+(omega/4.0)*(u_previous[i-1][j]
                    +u_previous[i+1][j]+u_previous[i][j-1]+u_previous[i][j+1]-4*u_previous[i][j]);
}

void BlackSOR(double ** u_previous, double ** u_current, int X_min, int X_max,
int Y_min, int Y_max, double omega) {
    int i,j;
    for (i=X_min;i<X_max;i++)
        for (j=Y_min;j<Y_max;j++)
            if ((i+j)%2==1)
                u_current[i][j]=u_previous[i][j]+(omega/4.0)*(u_current[i-1][j]
                    +u_current[i+1][j]+u_current[i][j-1]+u_current[i][j+1]-4*u_previous[i][j]);
}

```

---

## 6.2 Data Parallelism Pattern - Single Program Multiple Data

Για τις παράλληλες υλοποιήσεις που δημιουργήσαμε, ακολουθήσαμε την τεχνική της παραλληλοποίησης δεδομένων. Συγκεκριμένα, ακολουθήσαμε την προσέγγιση Single Program Multiple Data (SPMD). Στην προσέγγιση αυτή, όλες οι διεργασίες εκτελούν το ίδιο πρόγραμμα, αλλά υπ-

άρχουν διαφορές στην επικοινωνία και στα δεδομένα που επεξεργάζεται ο καθένας αναλόγως κάποιου χαρακτηριστικού ID που έχει κάθε διεργασία. Η τεχνική αυτή έχει προοπτικές για πολύ καλή κλιμάκωση σε αρχιτεκτονικές κατανεμημένης μνήμης με χρήση προγραμματιστικού μοντέλου MPI.

Η κύρια διεργασία, με PID=0 είναι αυτή η οποία κάνει τον διαμοιρασμό των δεδομένων στις υπόλοιπες διεργασίες και αργότερα λαμβάνει τα τελικά αποτελέσματα από όλους.

Για να ξεκινήσουμε την διαδικασία του διαμοιρασμού, χωρίζουμε τον αρχικό πίνακα σε Grid, βάσει των διαθέσιμων διεργασιών που έχουμε σε κάθε εκτέλεση. Τις διαστάσεις αυτού του grid τις ορίζουμε παραμετρικά από τις εισόδους Px,Py κατά την εκτέλεση του προγράμματος. Το γινόμενο Px\*Py πρέπει πάντα να είναι ίσο με των αριθμό MPI διεργασιών που έχουμε ορίσει.

Αν μπορούμε να διαιρέσουμε τέλεια την διάσταση globalX του πίνακα με τον αριθμό Px τότε η διάσταση localX του τοπικού block κάθε διεργασίας είναι ίσο με:

$$localX = \frac{globalX}{Px}$$

Αντίστοιχα βρίσκουμε την διάσταση localY του τοπικού block

$$localY = \frac{globalY}{Py}$$

Αν οι διαστάσεις globalX, globalY δεν διαιρούνται τέλεια τότε προσθέτουμε padding στον global πίνακα για ευκολία στον διαμοιρασμό. Θέλει ιδιαίτερη προσοχή αυτή η περίπτωση γιατί κατά την διάρκεια του υπολογισμού, οι διεργασίες που κατέχουν εν τέλει dummy δεδομένα padding, ΔΕΝ πρέπει να τα λάβουν υπόψιν τους στους υπολογισμούς.

Κάθε διεργασία λαμβάνει ένα υπό-block, διαστάσεων localX x localY. Ο διαμοιρασμός αυτός γίνεται από την διεργασία master, με την χρήση της εντολής **MPI\_Scatterv** και φαίνεται καλύτερα στο παρακάτω σχήμα.

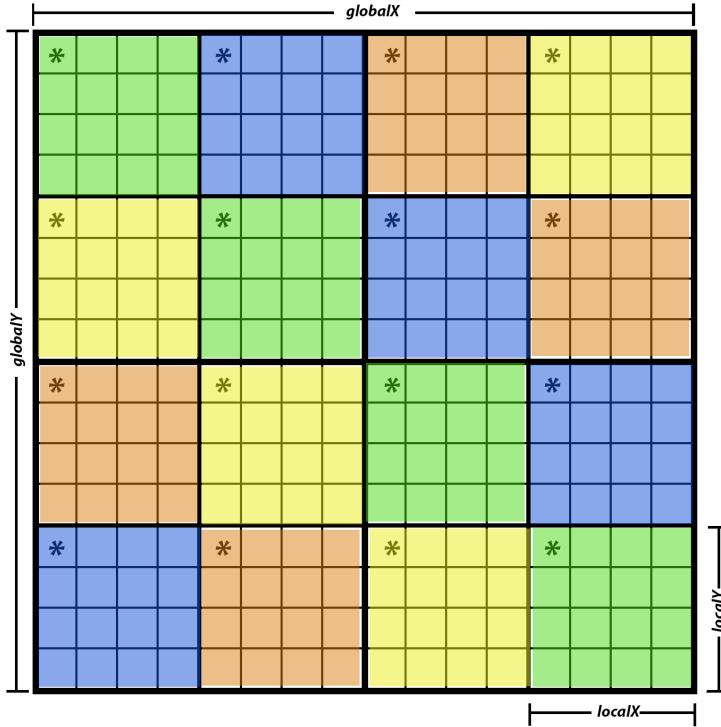


Figure 51: Global and Local Blocks (16x16 Global - Px=4, Py=4)

Οι αστερίσκοι δείχνουν τα σημεία στα οποία ξεκινάει ο διαμοιρασμός δεδομένων (scatter-points).

Η διαδικασία αυτή επαναλαμβάνεται ανάποδα, με χρήση της συνάρτησης **MPI\_Gatherv**, στο τέλος της εκτέλεσης του προγράμματος ώστε η master διεργασία να λάβει τα αποτελέσματα πίσω στον global πίνακα.

### 6.3 Ζητήματα επικοινωνίας μεταξύ διεργασιών

Έχοντας πλέον χωρίσει τα δεδομένα σε ένα δισδιάστατο πλέγμα, προκύπτει το πρόβλημα επικοινωνίας μεταξύ διεργασιών καθώς οι συνοριακές τιμές των κελιών χρειάζονται δεδομένα που ανήκουν σε άλλη διεργασία για να εκτελέσουν τους υπολογισμούς τους. Η επικοινωνία αυτή αλλάζει από μέθοδο σε μέθοδο.

Η κάθε διεργασία εκτελεί υπολογισμούς σε δικό της block, συγκεκριμένων διαστάσεων.

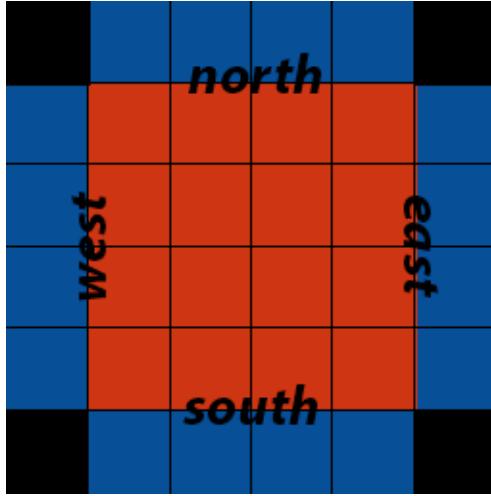


Figure 52: Heat Transfer Methods - 4x4 Local Block

Κάθε block, έχει πραγματικές διαστάσεις  $N \times N$ , αλλά έχουμε δεσμένους στην μνήμη διαστάσεις  $(N+2) \times (N+2)$ . Οι έξτρα ψέσεις μνήμης αυτές είναι εκεί που θα στείλουν οι γειτονικές διεργασίες τα δεδομένα που χρειάζεται η κάθε διεργασία για τους υπολογισμούς του local block του.

Στο παραπάνω σχήμα, οι πραγματικές διαστάσεις του block παρουσιάζονται με κόκκινο χρώμα ενώ οι "βοηθητικές" ψέσεις παρουσιάζονται με μπλε χρώμα. Τα μαύρα κουτάκια είναι δυναμικά δεσμευμένα, αλλά δεν χρησιμοποιούνται στο πρόβλημά μας.

Παρατηρούμε ότι επόμενη διαδικασία είναι η κάθε διεργασία να γνωρίζει ποιες διαδικασίες είναι οι γειτονικές του ώστε να γίνει σωστή ανταλλαγή δεδομένων.

Ο διαχωρισμός και η αποστολή δεδομένων στο OpenMPI μπορεί να γίνει με αρκετούς τρόπους. Αξιοποιήσαμε την ύπαρξη των Cartesian Communicator. Η κάθε διεργασία ανήκει σε συγκεκριμένη ψέση στο δισδιάστατο πλέγμα, έχει ειδικό PID ίσο με το άθροισμα  $x+y$  όπου  $(x,y)$  οι διαστάσεις του στον Cartesian Communicator.

$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$
$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$
$(2,0)$	$(2,1)$	$(2,2)$	$(2,3)$
$(3,0)$	$(3,1)$	$(3,2)$	$(3,3)$

Figure 53: Heat Transfer Methods - Cartesian Communicator ( $P_x=P_y=4$ )

Με χρήση της εντολής **MPI\_Cart\_shift**, η κάθε διεργασία μπορεί να βρεί τους west,east,north

και south γείτονές της. Έτσι, η κάθε διεργασία γνωρίζει πλέον τους γείτονες τους ή γνωρίζουν αν είναι συνοριακές διεργασίες, δηλαδή αν βρίσκονται στα όχρα του global πίνακα.

### Επικοινωνία στην μέθοδο Jacobi

Για την μέθοδο Jacobi η επικοινωνία είναι αρχετά απλή. Κάθε διεργασία πρέπει να λάβει είτε μια στήλη πίνακα (από τους west, east γείτονες) ή μια γραμμή (από τους north, south γείτονες). Αντίστοιχα, η ίδια διεργασία πρέπει να στείλει τις γραμμές/στήλες που έχουν ανάγκη οι γειτονικές διεργασίες.

Χρησιμοποιώντας το παραπάνω σχήμα που παρουσιάζουμε το local block, κάθε διεργασία πρέπει να αποστέλλει στους αντίστοιχους γείτονές του τις κόκκινες στήλες/γραμμές και να λάβει από αυτούς στις αντίστοιχες μπλε θέσεις τα δικά τους δεδομένα.

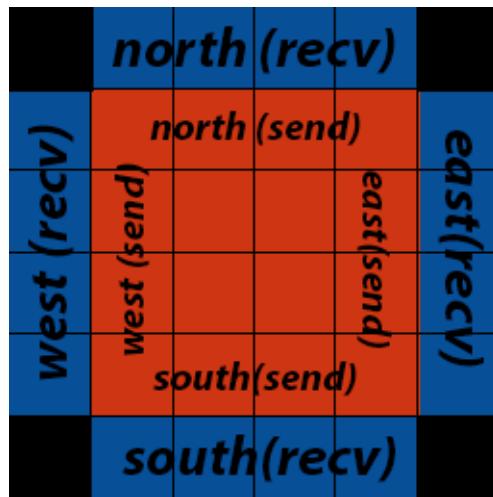


Figure 54: Jacobi Communication - Each process sends to each neighbours the "send" row/-column and receives the "recv" row/column

Εφόσον υπάρχει αυτή η σχέση αποστολής/λήψης μηνύματος μεταξύ των γειτονικών διεργασιών, χρησιμοποιούμε την εντολή **MPI\_Sendrecv**. Οι οριακές διεργασίες που δεν έχουν συγκεκριμένους γείτονες δεν παίρνουν μέρος στις επιμέρους επικοινωνίες. Η οποιαδήποτε διεργασία θα στείλει στην γειτονική της και μετά θα μπλοκάρει μέχρι να λάβει από αυτή δεδομένα.

### Επικοινωνία στην μέθοδο Gauss-Seidel

Η επικοινωνία για την μέθοδο Gauss-Seidel είναι αρχετά διαφορετική καθώς πλέον γίνεται σε 2 φάσεις αντί για 1. Κάθε διεργασία, έχει ως dependency τιμές του πίνακα από τους αριστερούς και πάνω γείτονες (west,north).

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Figure 55: Heat Transfer Methods - Gauss-Seidel Communication ( $P_x = P_y = 4$ )

Στο παραπάνω σχήμα φαίνεται με ποια σειρά μπορεί να γίνει η εκτέλεση της μεθόδου Gauss-Seidel. Οι διεργασίες με ίδιο χρώμα σημαίνει ότι μπορούν να τρέξουν ταυτόχρονα. Η πρώτη διεργασία που μπορεί να ξεκινήσει τον υπολογισμό είναι  $(0,0)$  αφού δεν έχει κάποιο dependency από δεδομένα. Αφού υπολογιστεί όλο το block της  $(0,0)$  τότε μπορεί να στείλει την οριακή east στήλη και south γραμμή ώστε να ξεκινήσουν τον υπολογισμό οι  $(1,0)$  και  $(0,1)$ . Το μοτίβο αυτό επαναλαμβάνεται μέχρι να υπολογιστεί το τελευταίο block.

Σε αντίθεση με την Jacobi, χρησιμοποιούμε τις non-blocking εκδοχές των send,recv. Στην εκτέλεση αυτών, επιστρέφεται ένα request στον χρήστη. Μαζέυοντας όλα τα request για κάθε send/recv εντολή που χρησιμοποιεί η κάθε διεργασία για επικοινωνία με τους γείτονες, περιμένει στην ολοκλήρωση της επικοινωνίας στην εντολή **MPI\_Waitall**.

Άρα, οι διεργασίες που δεν έχουν κάποιο dependency, δηλαδή δεν περιμένουν στο request των receive, είναι ελεύθερα να εκτελέσουν τους υπολογισμούς τους.

Η διαδικασία επικοινωνίας του Gauss-Seidel είναι σίγουρα πιο περίπλοκη και χρονοβόρα από την Jacobi αλλά το αποτέλεσμα είναι ένας πολύ γρηγορότερος ρυθμός σύγκλισης.

### Επικοινωνία στην μέθοδο Red-Black ordering

Η επικοινωνία για την μέθοδο Red-Black είναι πολύ παρόμοια με αυτή του Jacobi. Η συγχεκριμένη, έχει 2 φάσεις υπολογισμού και 2 φάσεις επικοινωνίας. Κάνουμε ανταλλαγή δεδομένων όπως κάναμε και στην Jacobi, αποστέλοντας τιμές του πίνακα της προηγούμενης χρονικής στιγμής και μετά εκτελούμε την φάση υπολογισμού Update Red. Για την 2η φάση, κάνουμε την ίδια διαδικασία επικοινωνίας, αυτή την φορά για τον πίνακα της παρούσας χρονικής στιγμής και μετά εκτελούμε την φάση υπολογισμού Update Black. Είναι σαν να εκτελούμε 2 φορές την μέθοδο Jacobi.

Η μέθοδος Red-Black έχει το μεγαλύτερο overhead και από τους 3 αλγορίθμους, αλλά έχει και τον καλύτερο ρυθμό σύγκλισης.

## 6.4 Μετρήσεις με έλεγχο σύγκλισης

Κάποιες σύντομες σημειώσεις:

- Ο χρόνος εκτέλεσης που υπολογίζουμε, είναι πάντα ο μέγιστος μεταξύ των διεργασιών
- Ο χρόνος επικοινωνίας είναι ο μέσος όρος μεταξύ της επικοινωνίας ΌΛΩΝ των διεργασιών
- Ο χρόνος σύγκλισης είναι επίσης ο μέγιστος μεταξύ όλων

Για κάθε αλγόριθμο που υλοποιήσαμε, κάναμε τις απαραίτητες μετρήσεις σε πίνακα μεγέθους 1024x1024, χρησιμοποιώντας 64 διεργασίες MPI. Εξετάσαμε 3 διαφορετικούς συνδυασμούς για το grid size: 8x8, 16x4 και 4x16. Θα δούμε άμεσα ότι έχει πολύ μεγάλη σημασία η επιλογή του σωστού block size.

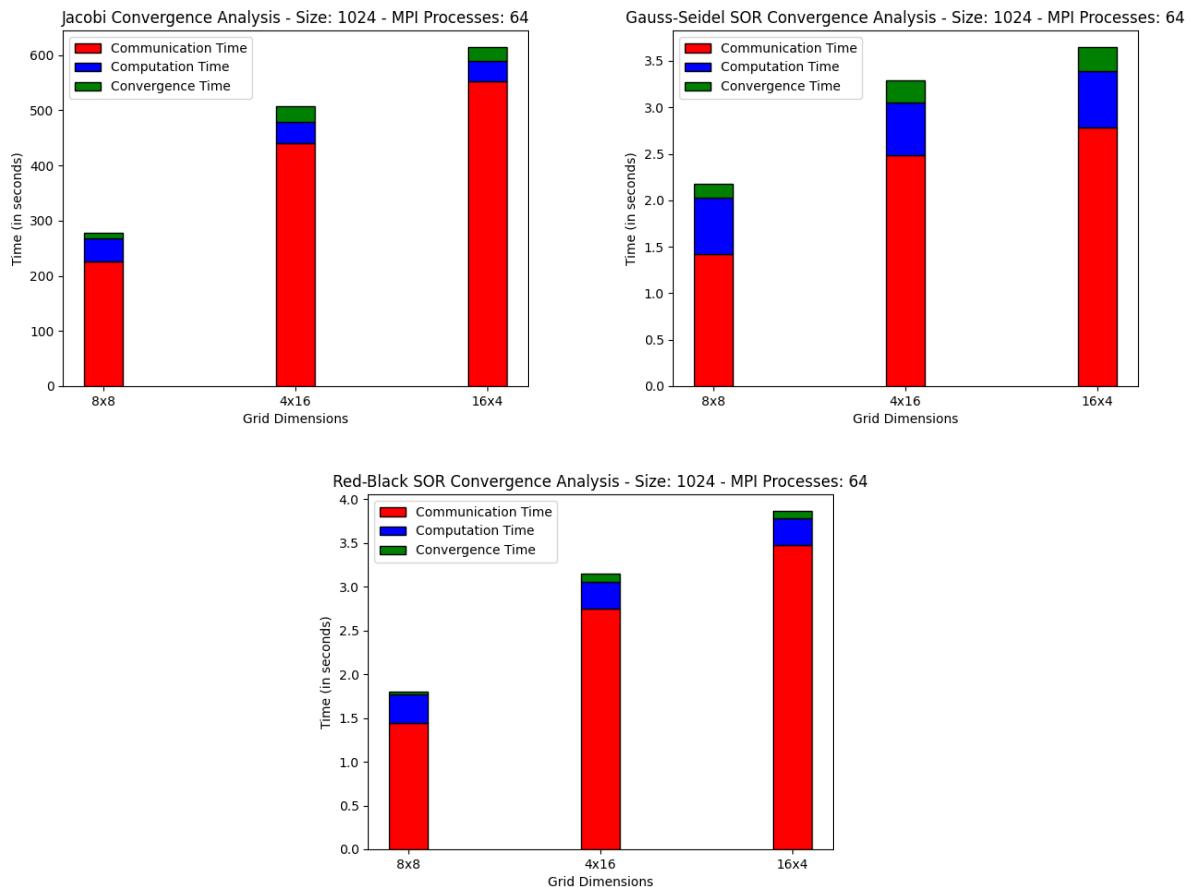


Figure 56: Heat Transfer Methods - Convergence Comparisso

Method \ Timers	Communication	Computation	Convergence	Total	Iterations
Jacobi	225.811780	40.527238	9.415624	270.395009	798200
Gauss-Seidel SOR	1.402382	0.604060	0.142405	2.109383	3200
Red-Black SOR	1.453022	0.327437	0.030029	1.829416	2500

## **Συμπεράσματα**

Η επιλογή του 8x8 grid size είναι ζεκάνθαρη καθώς τα αποτελέσματα για 16x4 ή 4x16 είναι πολύ χειρότερα χωρίς να επηρεάζει κάπως αλλιώς την ποιότητα του προγράμματος. Στα επόμενα πειράματα, χωρίς έλεγχο σύγκλισης, επιλέγουμε grid size 8x8 για τις εκτελέσεις με 64 διεργασίες.

### **Jacobi**

Ο Jacobi, καθώς είναι και η πιο naïve λύση μαθηματικά αλλά και προγραμματιστικά, εχεί και την χειρότερη επίδοση. Το μεγαλύτερο μειονέκτημά του είναι ο απίστευτα αργός ρυθμός σύγκλισης του. Χρειάζονται 250 φορές περισσότερα iterations συγχριτκά με τον Gauss Seidel και 320 φορές περισσότερα συγχριτκά με τον Red-Black ordering.

### **Gauss-Seidel SOR/Red-Black SOR**

Οι δύο αυτοί μέθοδοι ανήκουν στο ίδιο group καθώς οι επιδόσεις τους είναι πολύ παρόμοιες. Θεωρητικά, όπως θα δούμε και αργότερα, ο Gauss-Seidel έχει μικρότερο χρόνο υπολογισμού καθώς γίνεται μόνο σε ένα μέρος του αλγορίθμου, σε αντίθεση με τον Red-Black που γίνεται σε 2 φάσεις. Όμως, ο Red-Black έχει το πλεονέκτημα ότι συγκλίνει πιο γρήγορα.

Τελικά, θα επιλέγαμε την μέθοδο Red-Black ordering για την επίλυση προβλημάτων που **ΑΠΑΙΤΟΥΝ** σύγκλιση.

## **6.5 Μετρήσεις χωρίς έλεγχο σύγκλισης - Ποιοτικός έλεγχος scaling**

Για κάθε αλγόριθμο που υλοποιήσαμε, λαμβάνουμε μετρήσεις για μεγέθη πινάκων {2048x2048, 4096x4096, 6144x6144} και για {1,2,4,8,16,32,64} διεργασίες MPI. Πλέον, δεν απαιτούμε σύγκλιση των μενόδων, αλλά τρέχουμε και τους 3 αλγορίθμους για ακριβώς 256 iterations. Με αυτές τις μετρήσεις θέλουμε να δούμε, ποιοτικά, πόσο καλά κλιμακώνουν οι μέθοδοι αυτοί.

### 6.5.1 Speedup

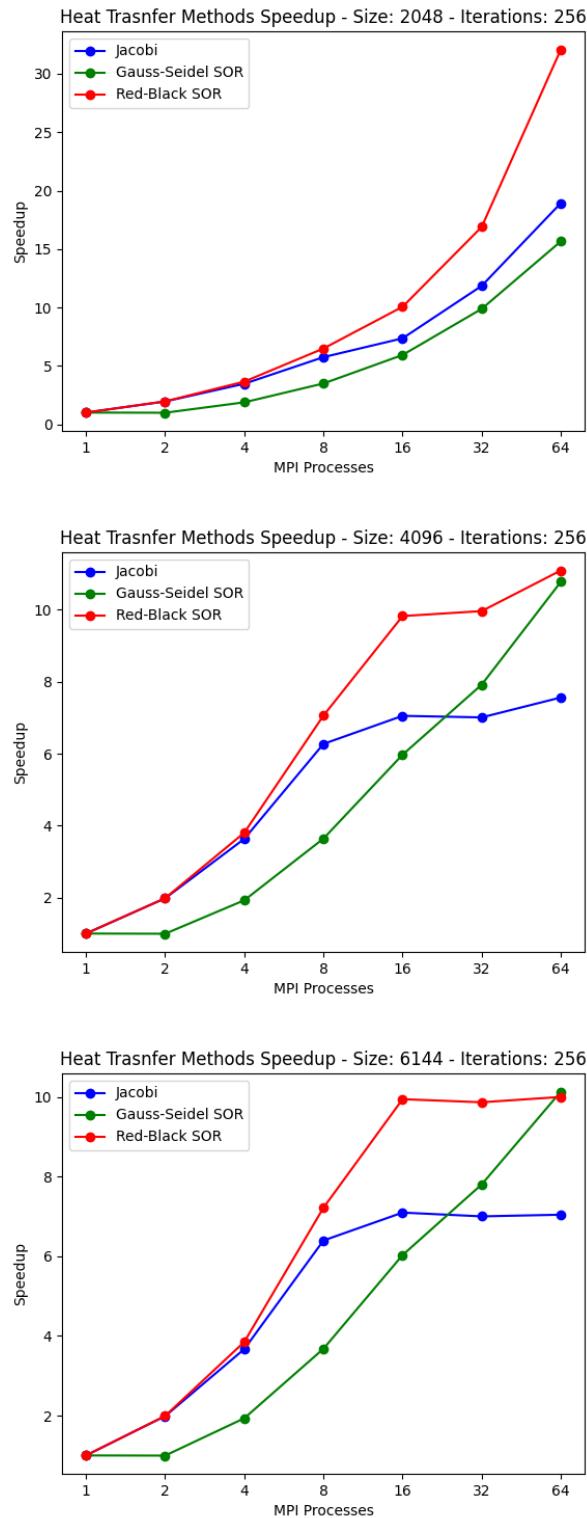


Figure 57: Heat Transfer Methods - Speedup

### 6.5.2 Ανάλυση Χρόνου Εκτέλεσης

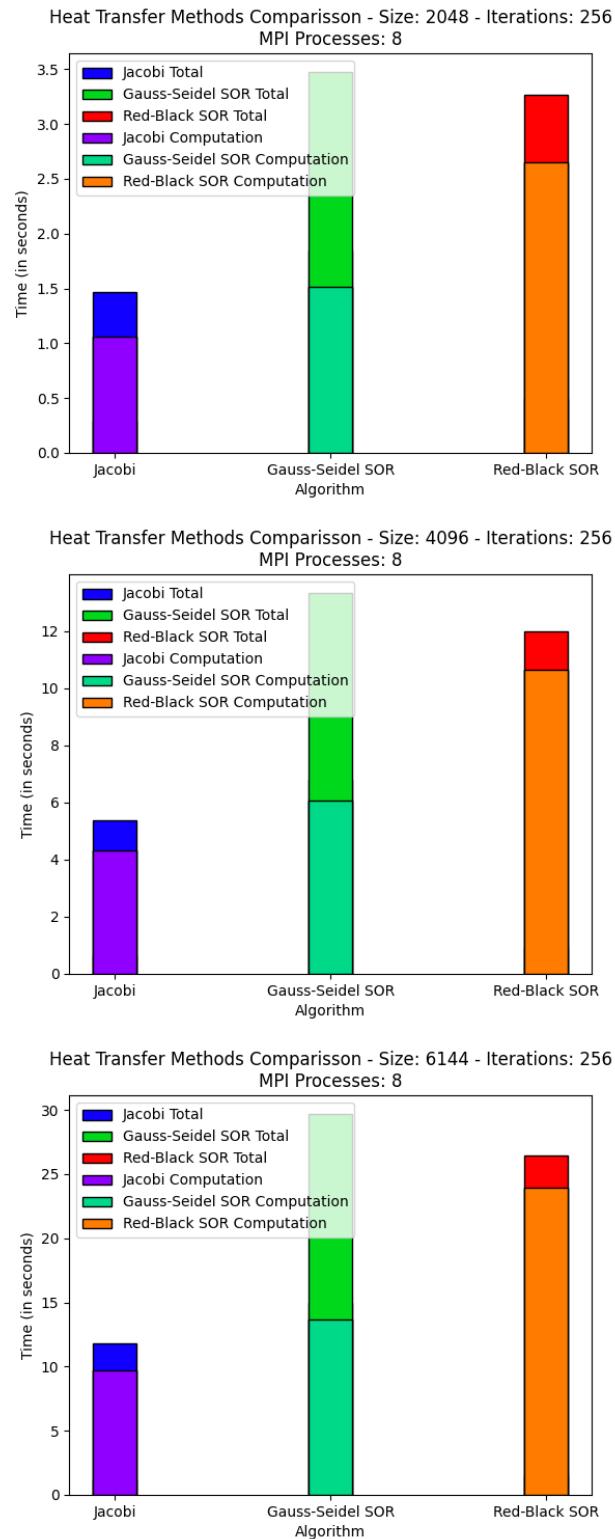


Figure 58: Heat Transfer Methods - Time Comparisson - MPI Processes: 8

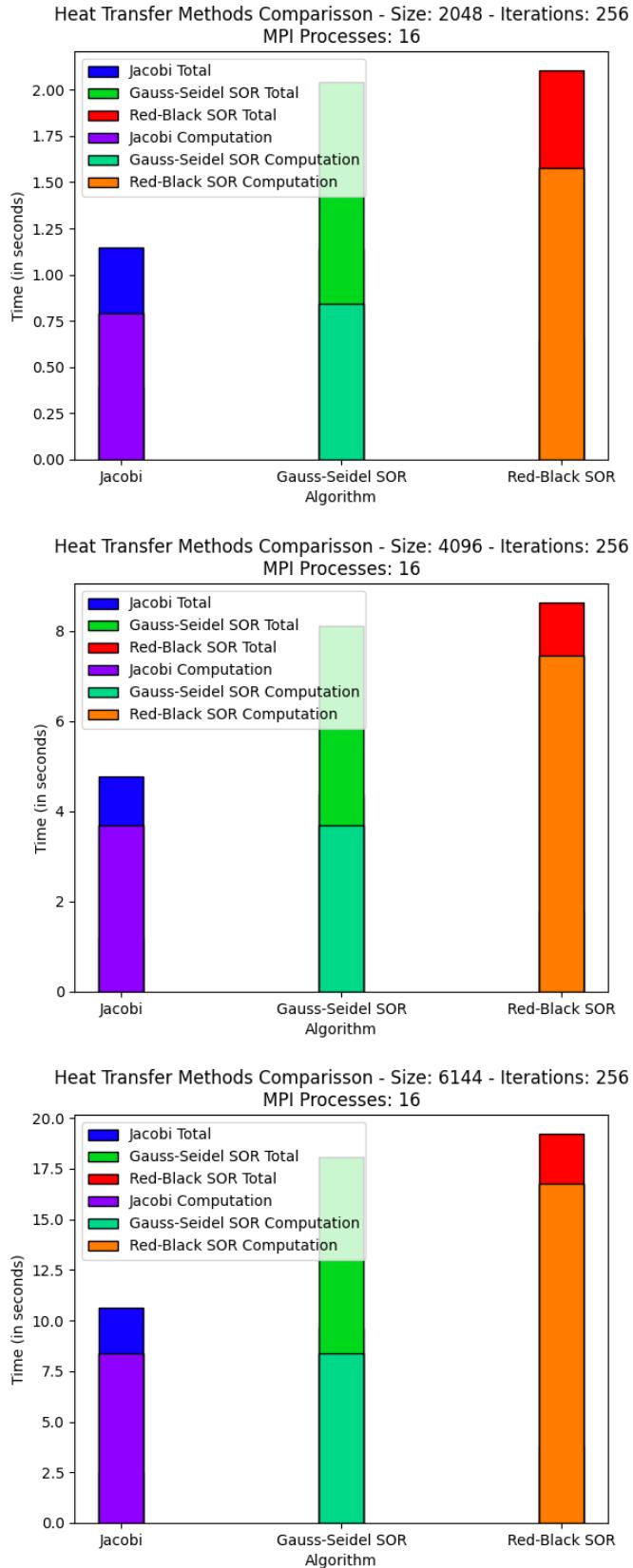


Figure 59: Heat Transfer Methods - Time Comparisson - MPI Processes: 16

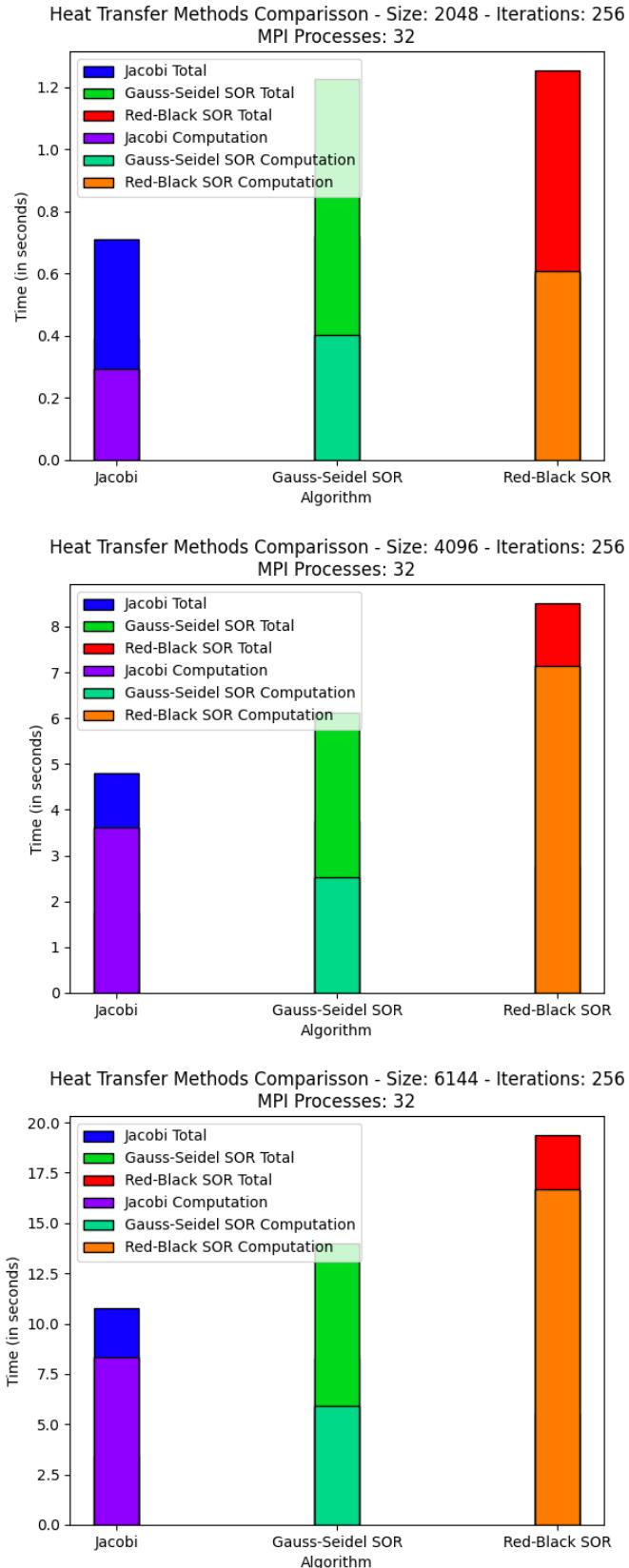


Figure 60: Heat Transfer Methods - Time Comparisson - MPI Processes: 32

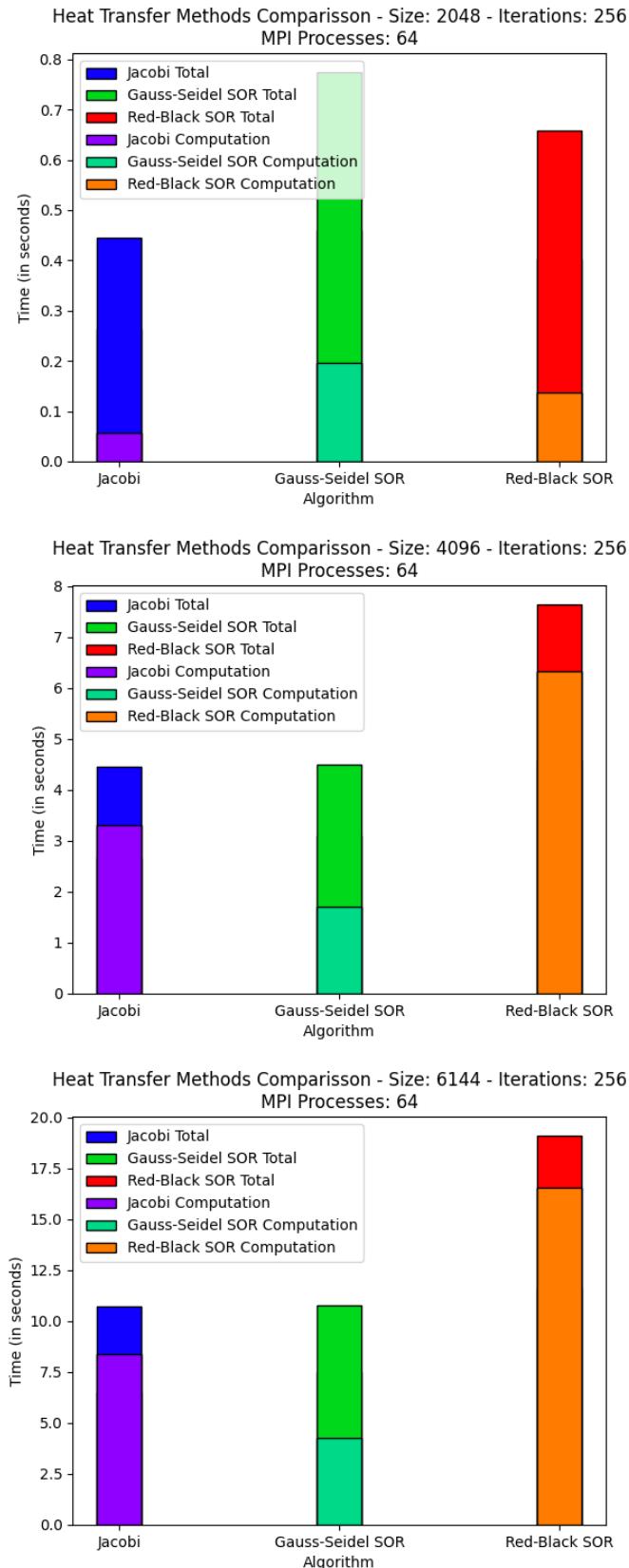
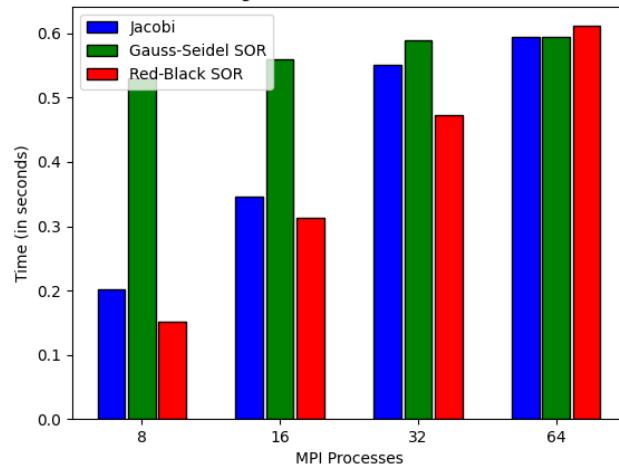
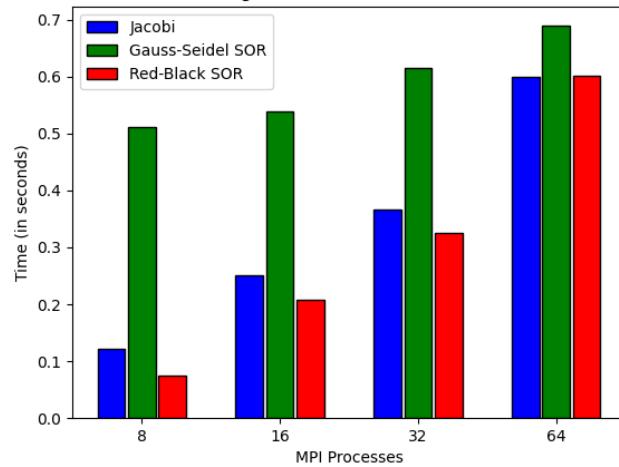


Figure 61: Heat Transfer Methods - Time Comparisson - MPI Processes: 64

Communication Percentage (over total time) - Size: 2048 - Iterations: 256



Communication Percentage (over total time) - Size: 4096 - Iterations: 256



Communication Percentage (over total time) - Size: 6144 - Iterations: 256

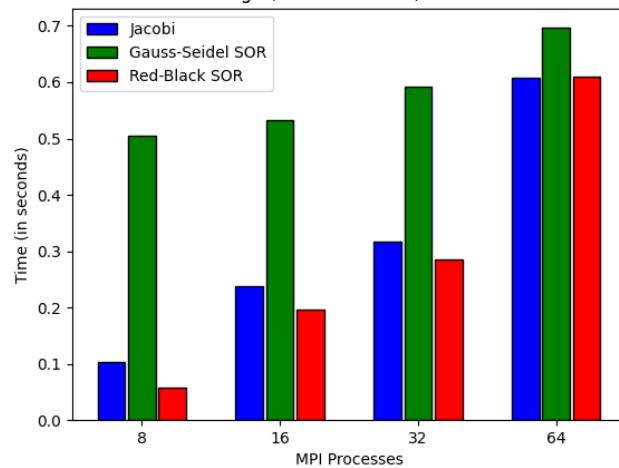


Figure 62: Heat Transfer Methods - Communication Analysis

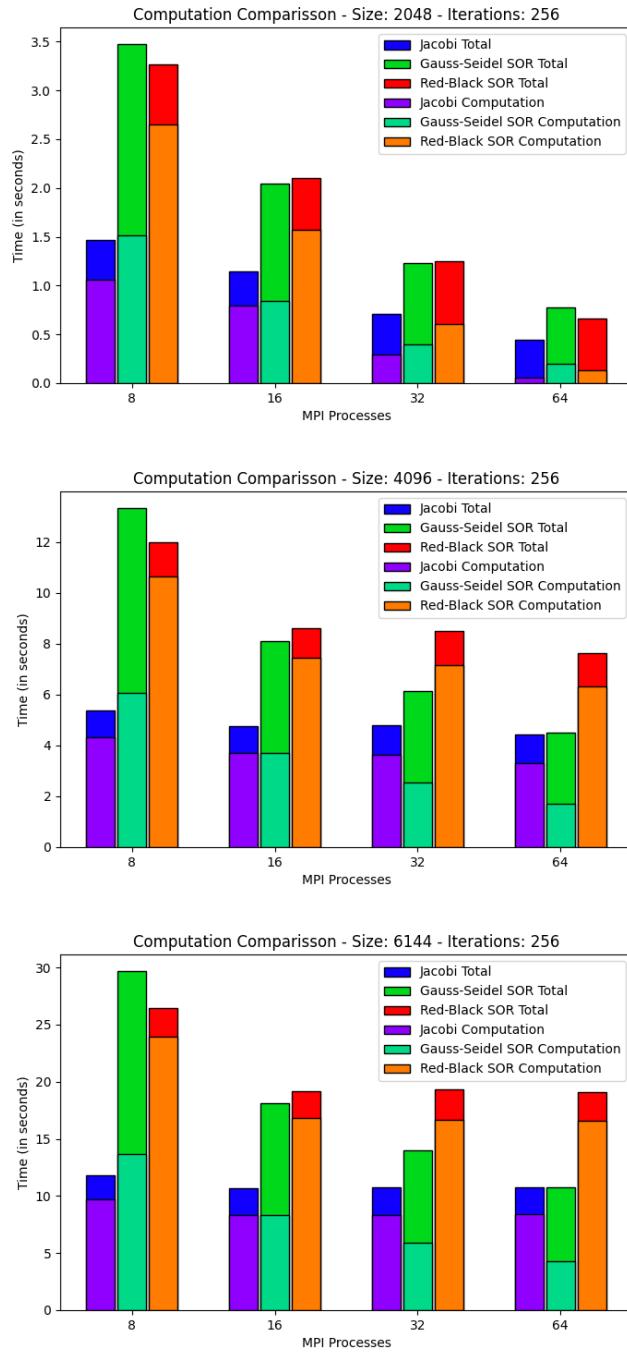


Figure 63: Heat Transfer Methods - Computation Analysis

## Συμπεράσματα

### Jacobi

Η μέθοδος Jacobi αποτελεί την γρηγορότερη, σε χρόνο εκτέλεση, μέθοδο. Συμπέρασμα εύνοητο καθώς έχει το μικρότερο overhead επικοινωνίας και μόνο μία φάση computation.

Για μικρούς πίνακες φαίνεται να κλιμακώνει με καλό ρυθμό καθώς συνεχώς αυξάνεται το

speedup του με την συνεχή αύξηση των διεργασιών. Για μεσαίους και μεγάλους πίνακες όμως, παρατηρούμε άλλα συμπεράσματα. Μετά τις 8 διεργασίες, περαιτέρω αύξηση των πόρων δεν σημαίνει και βελτίωση της επίδοσης. Δηλαδή, η κλιμάκωση "σπαεί" στις 8 διεργασίες.

## Gauss-Seidel

Η μέθοδος Gauss-Seidel έχει σίγουρα τον αργότερο ρυθμό εκτέλεσης από όλες τις μεθόδους. Αν και οι 3 μέθοδοι έχουν κάποιου είδους blocking επικοινωνία που σταματάει την εκτέλεση, η Gauss-Seidel περιορίζεται αρκετά περισσότερο. Έχει το μεγαλυτερό ποσοστό επικοινωνίας συγχριτκά με τις υπόλοιπες.

Για μικρούς πίνακες, η μέθοδος κλιμακώνει με καλό ρυθμό, αλλά όχι τόσο καλά όσο οι υπόλοιπες. Για μεσαίους και μεγάλους πίνακες όμως, η μέθοδος φαίνεται να κλιμακώνει πολύ καλύτερα από τις άλλες. Το διάγραμμα speedup δεν δείχνει κάποιου είδους stalling στην αύξηση του speedup. Δεν υπάρχει κάποιο σημείο που "σπάει" η κλιμάκωση της μεθόδου.

## Red-Black ordering

Η μέθοδος Red-Black ordering έχει επίσης αρκετά υψηλό overhead αφού έχει 2 φάσεις επικοινωνίας και 2 φάσεις υπολογισμού. Το ποσοστό επικοινωνίας αυξάνεται με την συνεχή αύξηση των διεργασιών και δεν μένει στάσιμο, κάτι που περιμέναμε καθώς θυμίζει πολύ την επικοινωνία της μεθόδου Jacobi.

Τα συμπεράσματα περί κλιμάκωσης είναι πολύ παρόμοια με αυτά της Jacobi. Για μικρά μεγέθη πινάκων, φαίνεται να κλιμακώνει καλύτερα από τις άλλες 2, μην ξεχνάμε όμως ότι έχει και τον μεγαλύτερο χρόνο εκτέλεσης και από τις 3. Για μεσαία και μεγάλα μεγέθη πινάκων, κλιμακώνει καλά μέχρι τις 16 διεργασίες και μετά "σπάει".

## 6.6 Τελικά Συμπεράσματα

Από τις 3 μεθόδους, καλύτερες ενδείξεις κλιμάκωσης παρουσιάζει η Gauss-Seidel. Αν υπόθεσουμε ότι οι περισσότερες εκτελέσεις ατυών των μεθόδων θα είναι με έλεγχο σύγκλισης, η Gauss-Seidel προσφέρει τα περισσότερα πλεονεκτήματα: και καλύτερη κλιμάκωση αλλά και πολύ καλό ρυθμό σύγκλισης.

Η μέθοδος Jacobi δεν προτιμάτε καθώς και σε σταθερό αριθμό iteration, αν και είναι πιο γρήγορος σε χρόνο εκτέλεσης, το αποτέλεσμα που θα δώσει δεν θα είναι ποιοτικό, αφού έχει πολύ χειρότερη ρυθμό σύγκλισης. Η Red-Black έχει τον καλύτερο ρυθμό σύγκλισης αλλά δεν κλιμακώνει όσο καλά όσο η Gauss-Seidel.

Με τους δοσμένους πόρους, βέλτιση επιλογή θα ήταν η Red-Black για εκτελέσεις με έλεγχο σύγκλισης και η Gauss-Seidel για εκτελέσεις σταθερού αριθμού iteration. Σε υποθετικό σενάριο περισσότερων διεργασιών MPI, θα επιλέγαμε την Gauss-Seidel καθώς με τα συγκεκριμένα πορίσματα, θα είχε καλύτερες επιδόσεις.

**Σημείωση:** Υπάρχει και μία ακόμη βελτιστοποίηση που μπορεί να γίνει. Στην μέθοδο Jacobi, και στην πρώτη φάση υπολογισμού της Red-Black, οι διεργασίες μπλοκάρουν μέχρι να λάβουν τα απαραίτητα δεδομένα από τις γειτονικές διεργασίες. Μπορεί να γίνει η εξής βελτίωση: Οι send/recv να γίνονται non-blocking, να εκτελούνται οι υπολογισμοί εκεί που μπορούν να γίνουν (δηλαδή στο εσωτερικό του block και όχι στις άκρες του) και μετά να

γίνουν οι υπολογισμοί στα άκρα του local block όταν σιγουρευτούμε ότι έχουν τελειώσει οι μεταφορές δεδομένων.