2022

ΗΜΜΥ, ΡΟΗ Υ, 9ο εξάμηνο: Συστήματα Παράλληλης Επεξεργασίας

1Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

ΟΜΑΔΑ 14 : ΠΑΝΑΓΙΩΤΗΣ ΝΤΑΓΚΑΣ, ΟΡΦΕΑΣ ΦΙΛΙΠΠΟΛΟΥΛΟΣ, ΜΑΡΙΑ-ΗΩΣ ΓΛΑΡΟΥ

1 Σκοπός της Άσκησης

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

2 Conway's Game of Life

Υλοποίηση Παραλληλοποίησης

Η κατάσταση κάθε κελιού καθορίζεται από εκείνη του ιδίου και των γειτόνων του σύμφωνα με τους κανόνες που περιγράφονται στην εκφώνηση. Έτσι με δεδομένη την σειριακή υλοποίηση του παιχνιδιού μπορούμε να παραλληλοποιήσουμε την διαδικασία ενημέρωσής της κατάστασης του κάθε κελιού με την βοήθεια του OpenMP. Συγκεκριμένα, χρησιμοποιήσαμε την εντολή:

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

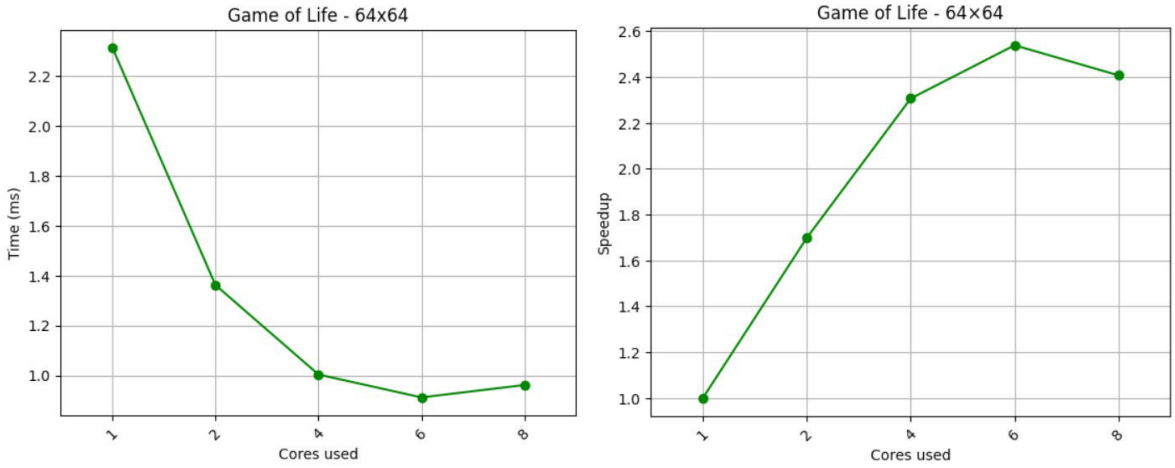
Προσθέτοντας την εντολή αυτή στον κώδικα που υλοποιεί σειριακά το παιχνίδι της ζωής, όπως φαίνεται στο παρακάτω screenshot απαιτούμε την παραλληλοποίηση του εξωτερικού βρόγχου που διασχίζει το grid και φροντίζουμε τα νήματα να μοιράζονται τις μεταβλητές N, previous, current εφόσον μπορούν τα νήματα να έχουν από κοινού πρόσβαση όλα τα νήματα στους πίνακες με τα προηγούμενα και τα τωρινά δεδομένα αλλά όχι τα i, j, nbrs δεδομένου ότι κάθε thread κρατάει και χειρίζεται τους δικούς του δείκτες για την κίνηση στους πίνακες και γράφει στην δική του μεταβλητή nbrs.

```
gettimeofday(&ts,NULL);
for ( t = 0 ; t < T ; t++ ) {
        #pragma omp parallel for shared(previous, current, N) private(i,j,nbrs)
        for ( i = 1 ; i < N-1 ; i++ )
                for ( j = 1 ; j < N-1 ; j++ ) {
                        nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
                               + previous[i][j-1] + previous[i][j+1] \
                               + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
                        if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                                current[i][j]=1;
                        else
                                current[i][j]=0;
                }

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

}
```

Αποτελέσματα

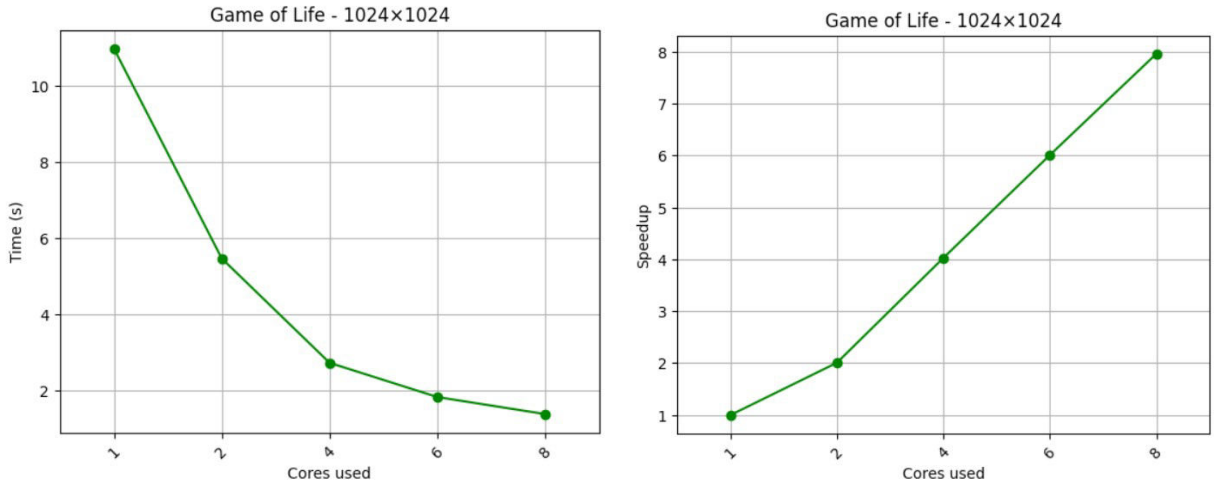
Για να ερμηνεύσουμε τα αποτελέσματα οπτικοποιήσαμε τις μετρήσεις με χρήση διαγραμμάτων. Συγκεκριμένα αναπαριστούμε τον χρόνο και το speedup του παράλληλου προγράμματος για ένα μόνο thread σε σχέση με εκείνο με πολλαπλά threads (δηλαδή πλοτάρουμε τον λόγο του χρόνου της παράλληλης υλοποίησης με ένα μόνο thread προς εκείνη με πολλαπλά).

- Grid 64 x 64



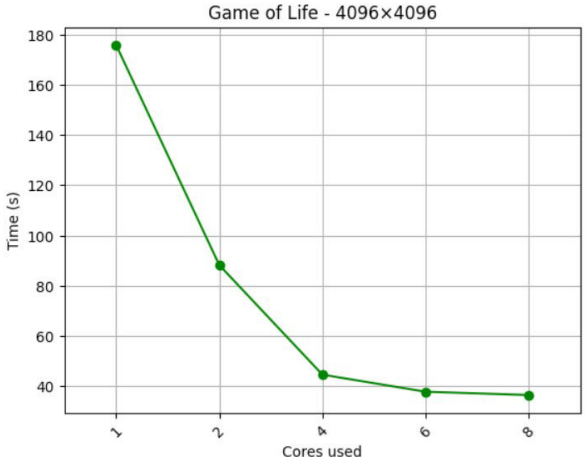
Παρατηρούμε ότι ενώ το speedup σημειώνει σημαντική αύξηση αρχικά, όταν αυξάνουμε τα νήματα από 4 σε 6 η αλλαγή δεν είναι μεγάλη ενώ από 6 σε 8 αυξάνεται ο χρόνος εκτέλεσης. Αυτή η αύξηση ερμηνεύεται από το overhead που προκύπτει από την δημιουργία τέτοιου αριθμού threads, ο οποίος δεν χρειάζεται δεδομένου ότι το grid είναι σχετικά μικρό (64 x 64) και επομένως και λίγες οι γραμμές που παραλληλοποιούμε. Για το μέγεθος αυτό μάλλον δεν μας κερδίζουμε σε χρόνο χρησιμοποιώντας περισσότερα των 6 threads και ανάλογα με τα διαθέσιμα resources ίσως αξίζει να αρκεστούμε στα 4 threads.

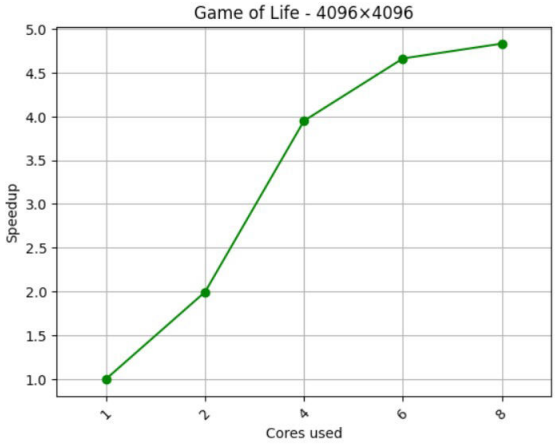
- Grid 1024 x 1024



Παρατηρούμε ότι το speedup είναι σημαντικό όσο αυξάνουμε τον αριθμό των χρησιμοποιούμενων threads. Μάλιστα διπλασιάζοντας των αριθμό των threads διπλασιάζεται και το speedup σε σχέση με την χρήση ενός μόνο thread που είναι η καλύτερη δυνατή επιτάχυνση που θα μπορούσαμε να περιμένουμε με την παραλληλοποίηση.

- Grid 4096 x 4096





Παρατηρούμε ότι έχουμε επιτάχυνση όσο αυξάνουμε τον αριθμό των threads. Όμως όσο μεγαλύτερη είναι η τιμή του πλήθους των threads στην οποία αναφερόμαστε τόσο μικρότερο είναι το speedup ου πετυχαίνουμε. Ο χρόνος χρησιμοποιώντας 6 και 8 threads βλέπουμε ότι δεν διαφοροποιείται παρά πολύ λίγο. Αυτό συμβαίνει καθώς το κάθε thread πρέπει να φέρει πολλά δεδομένα (4096 bytes) δεδομένου ότι κάθε thread αναλαμβάνει μία γραμμή του grid) στην cache με αποτέλεσμα να προκαλούνται πολλά misses στα επόμενα threads τα οποία καθυστερούν την εκτέλεση του προγράμματος.

Σημείωση: Για να επιβεβαιώσουμε τους ισχυρισμούς μας για το speedup όταν κάθε νήμα χειρίζεται μεγάλο πλήθος δεδομένων δοκιμάσαμε και μεγαλύτερο grid οπότε παρατηρήσαμε ότι η μεταβολή του χρόνου για τις διάφορες τιμές threads είναι παρόμοια.

- Σύγκριση αποτελεσμάτων υλοποίησης παράλληλου προγράμματος με 1 thread με το σειριακό πρόγραμμα

Παρατίθενται στον παρακάτω πίνακα οι τιμές των μετρήσεων.

Χρόνος (sec)			
Υλοποίηση\μέγεθος grid	64	1024	4096
Σειριακή	0.020356	10.809425	173.575566
Παράλληλη	0.023138	10.970185	175.962971

Μετρώντας τον χρόνο της σειριακής εκτέλεσης του προγράμματος παρατηρούμε λοιπόν ότι είναι ελάχιστα λιγότερος από εκείνον για την παράλληλη εκτέλεση του προγράμματος με ένα thread. Σε καμία περίπτωση δεν έχουμε παραλληλοποίηση των υπολογισμών του παιχνιδιού όμως στην

περίπτωση που δουλεύουμε με τις βιβλιοθήκες για τα νήματα έχουμε ένα overhead για την δημιουργία του thread που αναλαμβάνει τους υπολογισμούς που εξηγεί τα αποτελέσματα που προέκυψαν.

2Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

1 Σκοπός της Άσκησης

Στόχος της άσκησης είναι η ανάπτυξη δύο παράλληλων εκδόσεων του αλγορίθμου K-means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με τη χρήση του προγραμματιστικού εργαλείου OpenMP. Θα δοκιμάσουμε 2 βελτιώσεις υλοποίησης παράλληλου κώδικα και θα αξιολογήσουμε τις τελικές τους επιδόσεις.

Shared Cluster

Όσον αφορά την υλοποίηση με shared cluster αρχικά χρειάστηκε να παραλληλοποιήσουμε την βασική for loop (μέσα στο do{} while{} των εποχών) για τα objects όπως δείχνει το παρακάτω κομμάτι κώδικα:

(Σημείωση: Παρατηρήσαμε πως χωρίς το reduction στο delta οι επιδόσεις πέφτουν αρκετά, καθώς υπάρχει true sharing μεταξύ των threads και επιβαρύνεται ο χρόνος εκτέλεσης).

```
#pragma omp parallel for shared(objects,clusters,membership,newClusterSize,newClusters) private(i, index, j) firstprivate(numObjs, numCoords, numClusters) reduction(+:delta)
for (i=0; i<numObjs; i++) {
```

Επιπροσθέτως, προκειμένου να αποφύγουμε τα race conditions πάνω στα shared variables newClusterSize και clusters, χρειάζεται να κάνουμε atomic τις εντολές οι οποίες γράφουν πάνω σε αυτές, άρα:

```
#pragma omp atomic
    newClusterSize[index]++;
```

και

```
#pragma omp atomic
   newClusters[index*numCoords + j] += objects[i*numCoords + j];
```

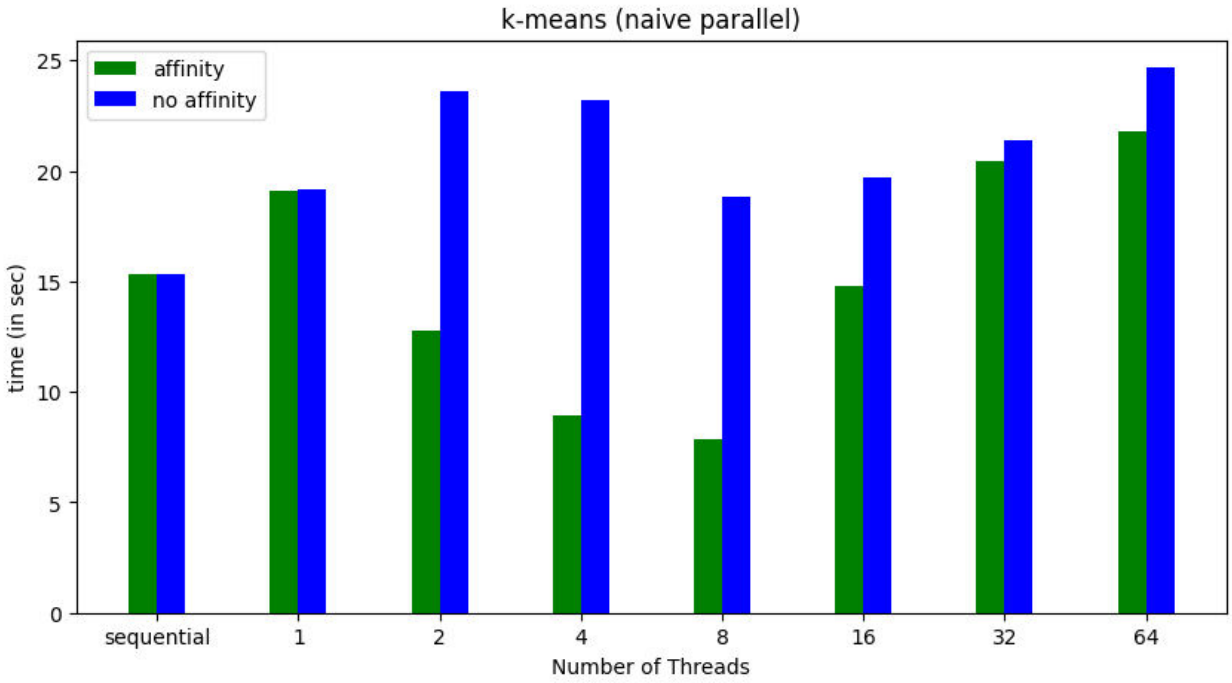
Τον παραπάνω κώδικα τον τρέξαμε με τα εξής:

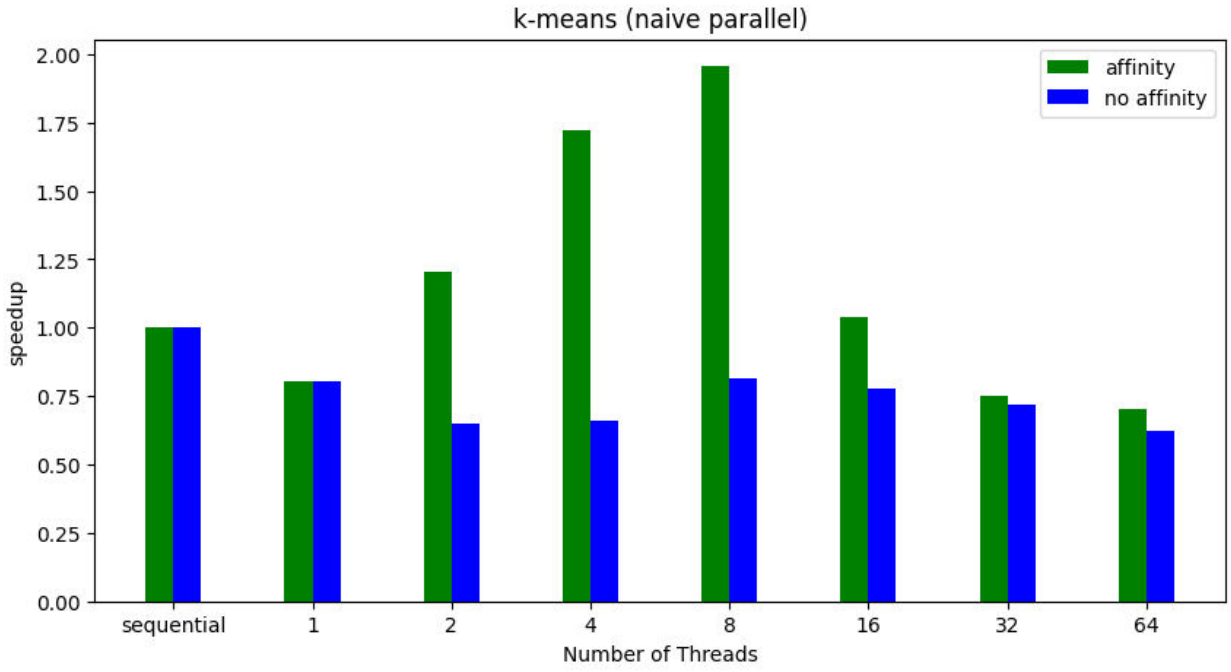
{Size, Coords, Clusters, Loops} = {256, 16, 16, 10}
threads = {1, 2, 4, 8, 16, 32, 64}

στο μηχάνημα sandman.

Στη συνέχεια χρησιμοποιήσαμε τη μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY προκειμένου να προσδέσουμε τα threads στα σε συγκεκριμένους πυρήνες κατά τη διάρκεια εκτέλεσης και ξανατρέξαμε το κώδικα.

Τέλος, τα barplots που προέκυψαν είναι τα εξής:


k-means (naive parallel)
affinity
no affinity
time (in sec)
0
5
10
15
20
25
sequential
1
2
4
8
16
32
64
Number of Threads


k-means (naive parallel)
affinity
no affinity
speedup
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
sequential
1
2
4
8
16
32
64
Number of Threads

Σχολιασμός αποτελεσμάτων:

Παρατηρούμε ότι **χωρίς το affinity** δεν έχουμε καθόλου καλά αποτελέσματα (χειρότερα από το σειριακό) και αυτό οφείλεται στο γεγονός ότι μεταξύ των διαφορετικών εποχών τα threads μπορεί να γίνουν scheduled σε διαφορετικούς πυρήνες με αποτέλεσμα να πρέπει να ξαναφέρουν στην cache τους τα δεδομένα που χρειάζονται (από τον πίνακα των objects) και σε μία NUMA αρχιτεκτονική όπως εδώ αυτό μπορεί να γίνει πολύ χρονοβόρο. Επίσης παρατηρούμε ότι μεταξύ των threads {1, 2, 4} και μεταξύ των threads {8, 16, 32, 64} ο χρόνος αυξάνεται (αντίστοιχα το speedup μειώνεται) σε ένα μικρό βαθμό και αυτό οφείλεται στο γεγονός ότι όσο αυξάνονται τα threads τόσο περισσότερο περιμένουν να γράψουν πάνω στις μεταβλητές newClusterSize και clusters (θυμίζουμε ότι το γράψιμο πάνω στις μεταβλητές αυτές γίνεται ατομικά). Τέλος παρατηρούμε ότι για 1 thread έχουμε λίγο χειρότερα αποτελέσματα από το σειριακό και αυτό οφείλεται στο overhead που έχουμε για τη δημιουργία του thread.

Ωστόσο, παρατηρούμε ότι **με affinity** τα αποτελέσματα είναι πολύ καλύτερα καθώς λύνουμε το παραπάνω πρόβλημα (δηλαδή το κομμάτι του πίνακα των objects που χρειάζονται κάθε φορά τα threads θα το βρουν στην cache τους). Αναλυτικότερα, για threads 2, 4, 8 πετυχαίνουμε καλύτερα αποτελέσματα από το σειριακό (και μάλιστα στη περίπτωση των 8 threads πετυχαίνουμε πολύ καλύτερα αποτελέσματα με speedup ~ 1.9), όμως για threads 32, 64 πετυχαίνουμε χειρότερους χρόνους από το σειριακό (ωστόσο, καλύτερους από το χωρίς affinity). Αυτό οφείλεται στο γεγονός ότι τα threads περιμένουν πολύ μέχρι να γράψουν στις μεταβλητές newClusterSize και clusters, δηλαδή το overhead των atomic εντολών είναι αρκετά έντονο. Με βάση τα παραπάνω, στη συγκεκριμένη υλοποίηση μπορεί να πει κανείς πως τα 8 threads είναι η “χρυσή τομή” που λύνει το πρόβλημα με τη μνήμη αλλά δεν έχει τόσο μεγάλο overhead από τις ατομικές εντολές.

Τέλος και στις 2 περιπτώσεις αντιμετωπίζουμε το πρόβλημα του false sharing μεταξύ των threads, γεγονός που αυξάνει το χρόνο εκτέλεσης, αλλά αυτά θα συζητηθεί εκτενώς στα παρακάτω ερωτήματα.

Copied Clusters and Reduce

Στη συγκεκριμένη υλοποίηση κάθε thread έχει το δικό του πίνακα newClusterSize και newClusters (με τα ονόματα local_newClusterSize και local_newClusters αντίστοιχα) προκειμένου να αποφύγουμε την καθυστέρηση που προκύπτει λόγω των ατομικών εντολών.

1ο ερώτημα:

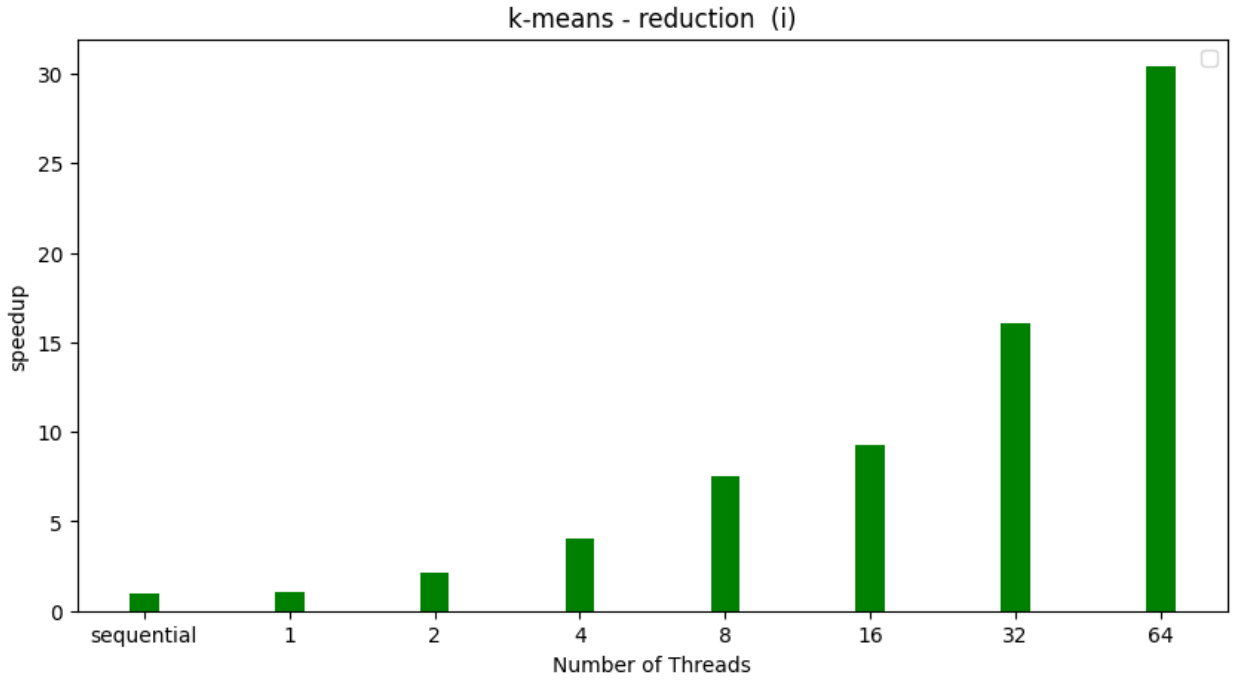
Στο συγκεκριμένο ερώτημα παραλληλοποιούμε με τον ίδιο τρόπο τον αλγόριθμο όπως και στο *Shared Cluster* μόνο που εδώ **δεν** χρειαζόμαστε τα atomics γιατί κάθε thread γράφει στους δικούς στου πίνακες και στο τέλος κάνουμε το reduce των αποτελεσμάτων (από ένα thread).

Το reduce των αποτελεσμάτων γίνεται ως εξής:

```
for(k = 0; k < nthreads; k++){
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            newClusters[i*numCoords+j] += local_newClusters[k][i*numCoords+j];
        }

        newClusterSize[i] += local_newClusterSize[k][i];

    }
}
```

Σε γενικές γραμμές περιμένουμε καλύτερα αποτελέσματα, ωστόσο αυτό θα φανεί από τα παρακάτω.

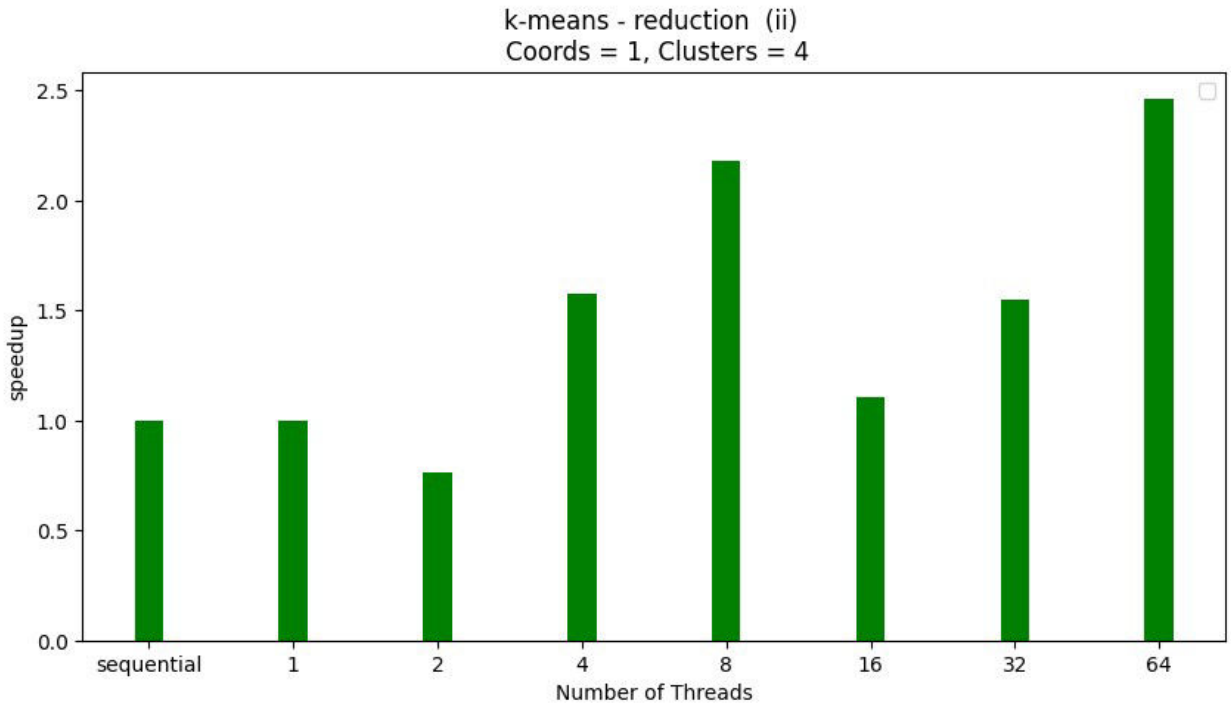
Παρακάτω είναι τα barplots του speedup και του execution time για threads = {1, 2, 4, 8, 16, 32, 64} και {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}:

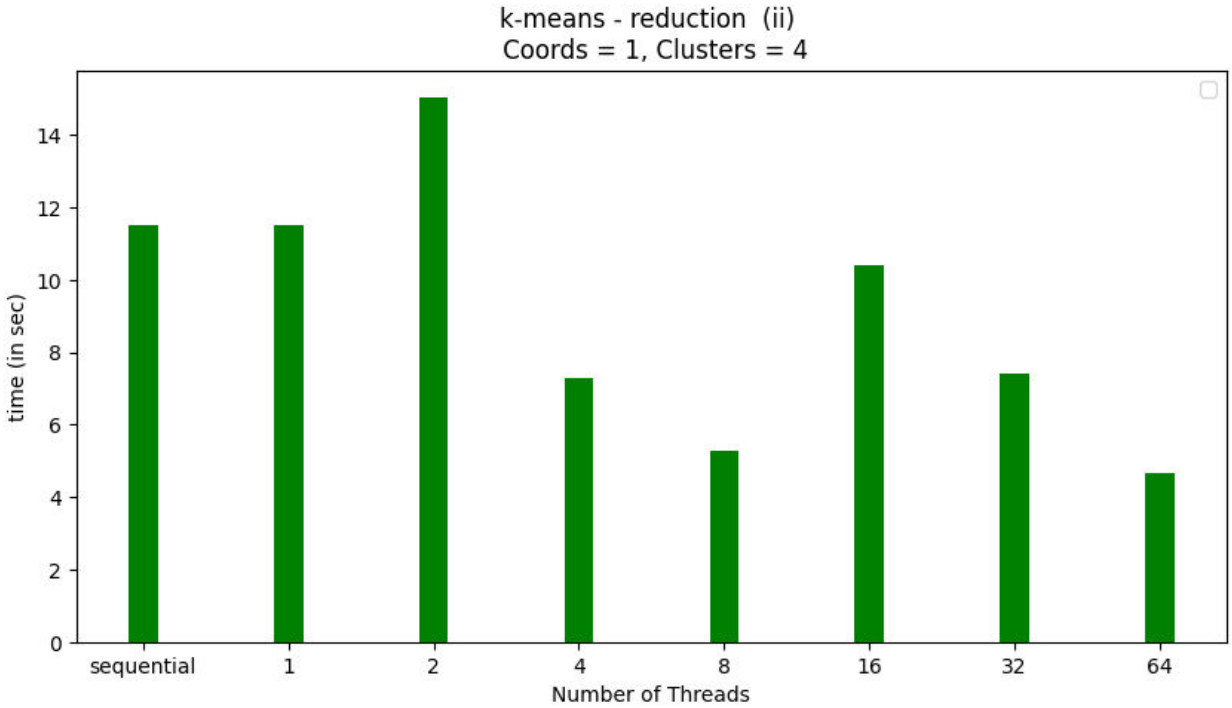


Όπως είναι εμφανές μόλις από τα 2 threads πετυχαίνουμε speedup ~ 2, ενώ όσο αυξάνονται παρατηρούμε γενικά ότι αυξάνεται και το speedup φτάνοντας σε τιμή 30 για 64 threads. Με την παραλληλοποίηση και το reduction λοιπόν το πρόγραμμα κάνει πολύ καλό scale. Τα αποτελέσματα είναι πολύ ικανοποιητικά. Αυτό είναι λογικό καθώς κάθε thread γράφει στα δικά του δεδομένα οπότε γλυτώνουμε τον χρόνο που απαιτεί ο συντονισμός της ανανέωσης (atomic εντολή) της δομής newClusters που χρειαζόταν όταν ήταν διαμοιραζόμενη μεταξύ των threads.

2ο ερώτημα:

Με configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} έχουμε τα εξής αποτελέσματα:





Στα παραπάνω διαγράμματα παρατηρούμε ότι για λίγα threads, 4 και 8, πετυχαίνουμε speedup 1.75 και 2.3 αντίστοιχα. Ωστόσο από την επίδοση όσο τα threads αυξάνονται παρατηρούμε ότι δεν κάνει κάνει καλό scale. Αυτό οφείλεται στο γεγονός ότι οι πίνακες local_newClusterSize και local_newClusters μικραίνουν πολύ με αποτέλεσμα πολλές σειρές να χωράνε σε ένα και μόνο cache line που αυτό συνεπάγεται άμεσα έντονο φαινόμενο false sharing μεταξύ των threads και άρα αρκετά αυξημένη καθυστέρηση λόγω cache coherence πρωτόκολλων.

Επιπροσθέτως και στις 2 περιπτώσεις ({Size, Coords, Clusters, Loops} = {256, 1, 4, 10} και {Size, Coords, Clusters, Loops} = {256, 10, 10, 10}) το first touch το κάνει το thread που κάνει το calloc για κάθε θέση των local πινάκων με αποτέλεσμα οι πίνακες αυτοί να τοποθετηθούν "κοντά" σε αυτό (δηλαδή στη μνήμη του αντίστοιχου NUMA node). Αυτό προσθέτει ακόμα μία καθυστέρηση καθώς κάθε thread θα βρει πιθανώς τα δεδομένα που χρειάζεται, τη πρώτη φορά, σε μνήμη απομακρυσμένου NUMA node.

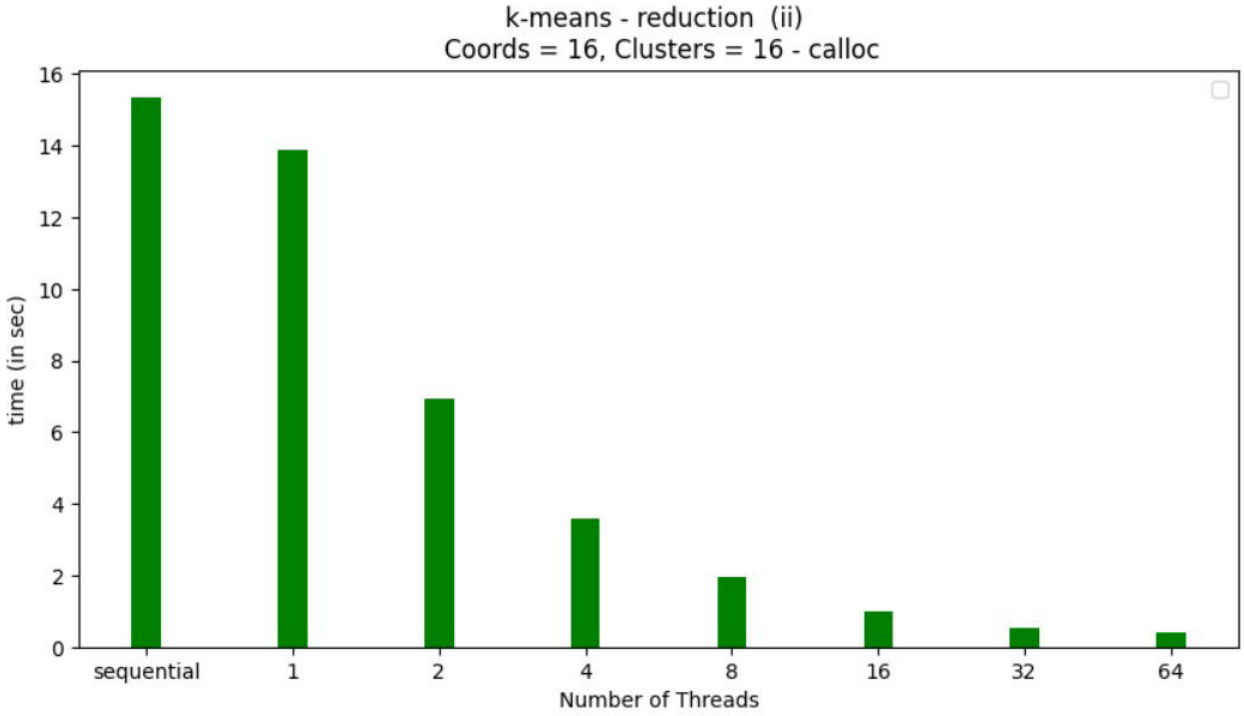
Για να λύσουμε και τα 2 παραπάνω προβλήματα παραλληλοποιούμε το for loop που κάνει τα calloc των local πινάκων. Κατά αυτό το τρόπο πετυχαίνουμε τόσο κάθε θέση (που αντιστοιχεί σε κάθε thread) των local πινάκων να αποθηκευτεί σε ξεχωριστό page (άρα δεν έχουμε πλέον false sharing) όσο και να γίνει το first touch της κάθε θέσης από το thread που τη χρησιμοποιεί με αποτέλεσμα να αποθηκευτεί (η κάθε θέση) στη μνήμη του NUMA node που "φιλοξενεί" το αντίστοιχο thread. (Θεωρούμε πως το calloc κάνει touch τους πίνακες με μηδενικά).

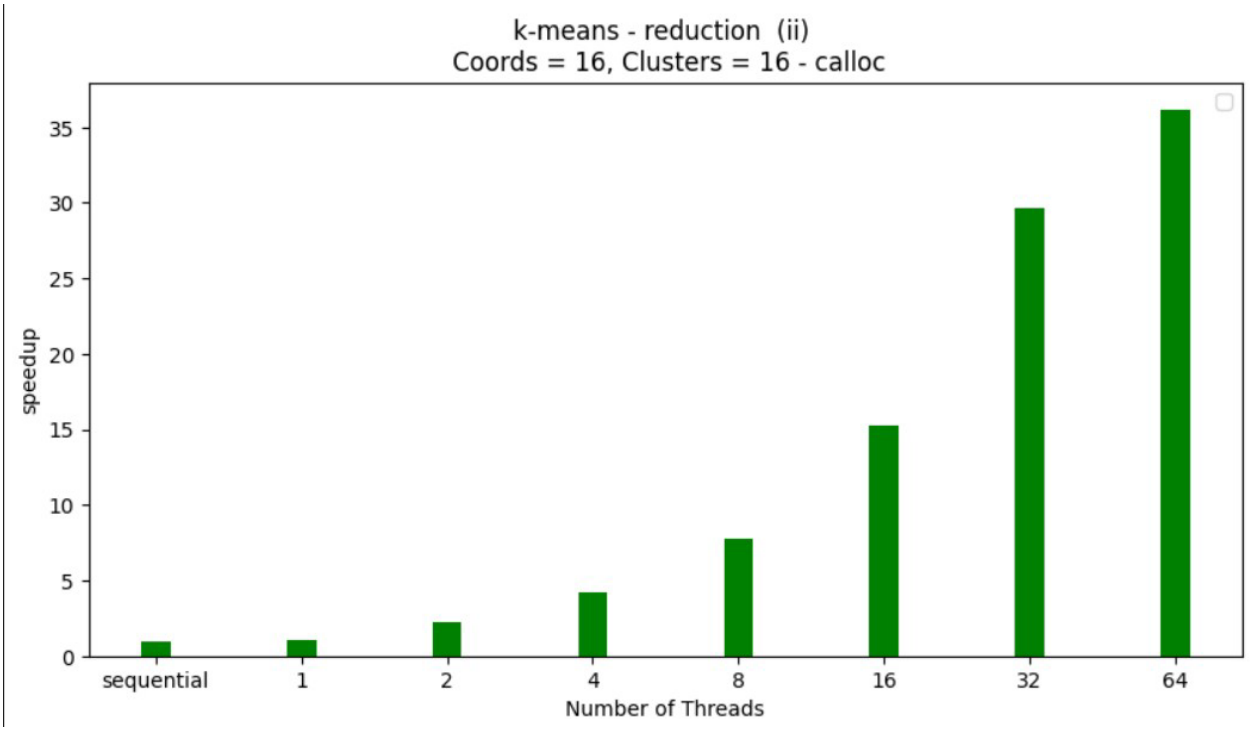
Δηλαδή κάναμε:

```
#pragma omp parallel for private(k)
for (k=0; k<nthreads; k++)
{
    local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters, sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords, sizeof(**local_newClusters));
}
```

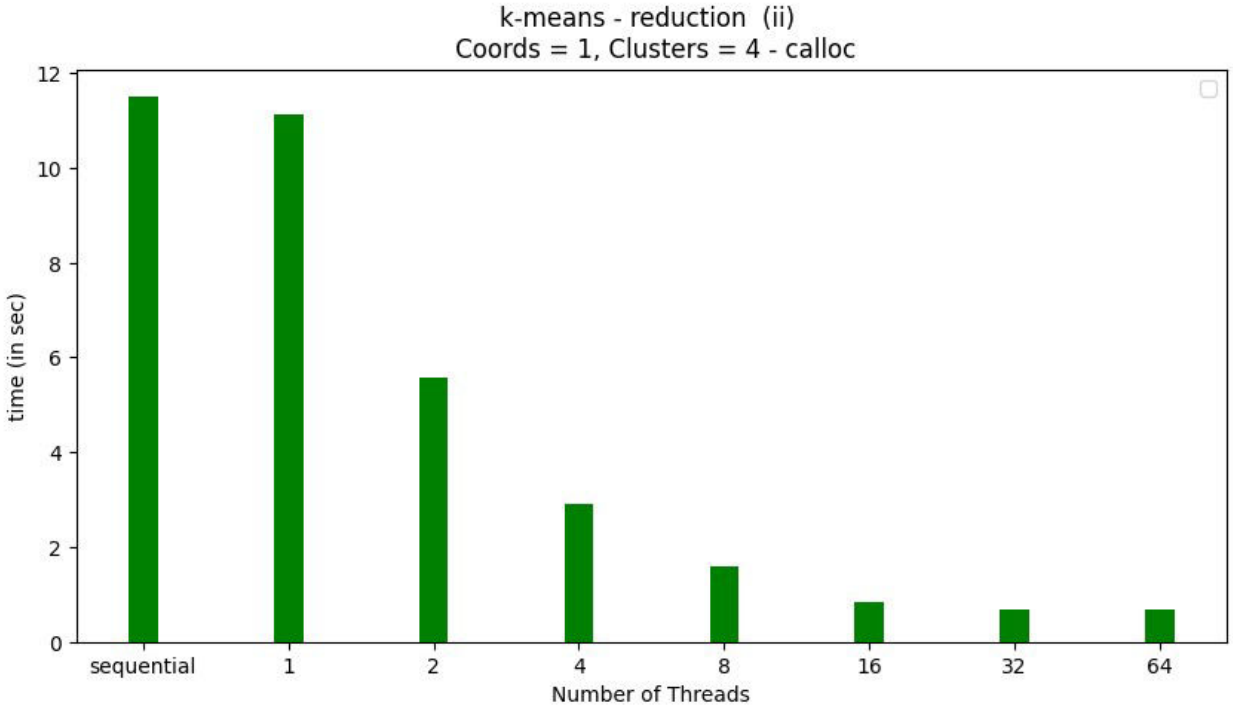
Τα αποτελέσματα είναι τα εξής:

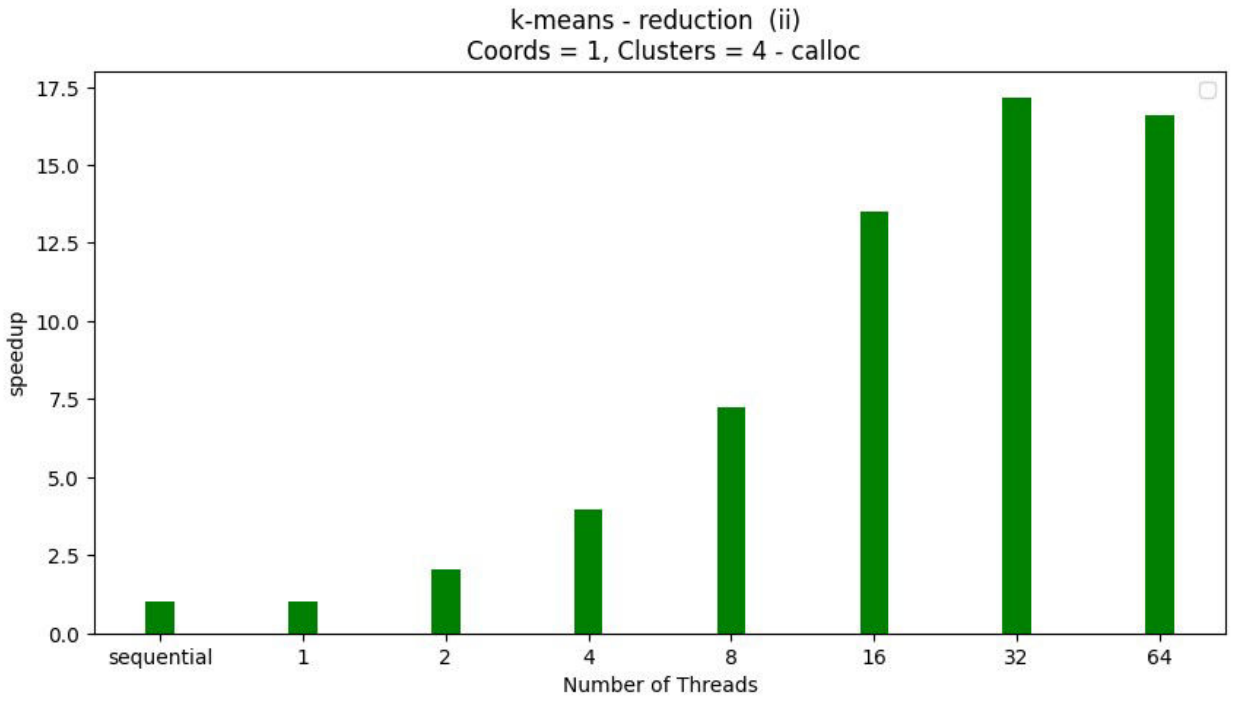
για configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}





και για configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}





Όσον αφορά το configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}, παρατηρούμε αρκετά βελτιωμένες επιδόσεις σε όλα τα threads και ελαφρώς καλύτερο scale. Πιο συγκεκριμένα αυξήθηκε το speedup ως εξής:

16 threads 10 → 14

32 threads 18 → 32
64 threads 29 → 36

Όσον αφορά το configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}, που είχε πολύ κακές επιδόσεις όσον αφορά το scaling, παρατηρούμε και εδώ αρκετά βελτιωμένες επιδόσεις, καθώς λύνουμε το κύριο πρόβλημα το οποίο είναι το false sharing. Πιο συγκεκριμένα αυξήθηκε το speedup ως εξής:

4 threads 1.6 → 4
8 threads 2.3 → 7.5
16 threads 1.2 → 13.5
32 threads 1.6 → 16.5
64 threads 2.5 → 16

Παρατηρούμε, τελικά ότι η παραλληλοποίηση του calloc βελτιώνει κατά πολύ τις επιδόσεις του μικρού configuration και το πρόγραμμα κάνει scale πολύ καλύτερα. Αυτό συμβαίνει, αφού λύσαμε το πρόβλημα του false sharing και επίσης, πλέον το κάθε thread έχει το κομμάτι του local clusters που χρειάζεται στο NUMA node που είναι πιο κοντά του και επομένως, τα accesses στην cache, αλλά και στην μνήμη γίνονται πολύ πιο γρήγορα και δεν υπάρχει overhead από άσκοπη επικοινωνία.

(**<u>ΣΗΜΕΙΩΣΗ</u>**: Σε κάποια γραφήματα η εκτέλεση με 1 thread φαίνεται να είναι λίγο πιο γρήγορη από την σειριακή έκδοση. Αυτό συμβαίνει, γιατί σε διαφορετικές εκτελέσεις μπορεί να υπάρχουν μικρές διαφορές στους χρόνους).

3ο ερώτημα:

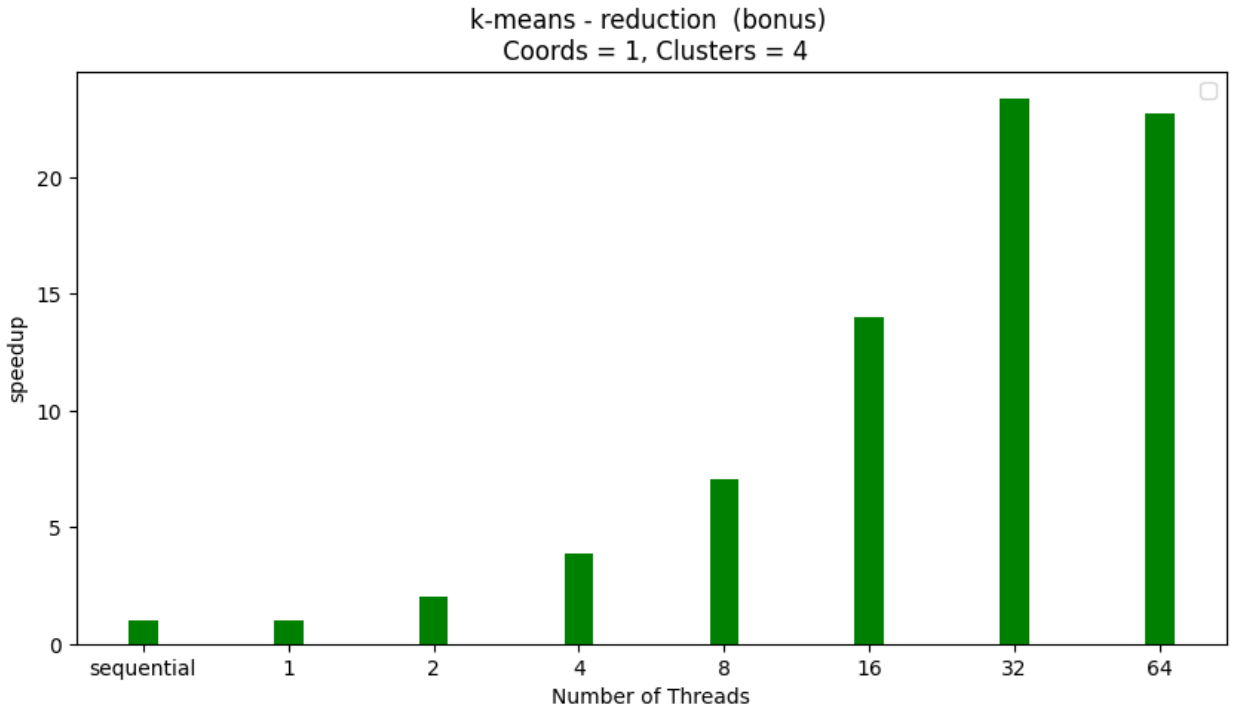
Όσον αφορά τον πίνακα των objects έχουμε και εδώ το πρόβλημα του first touch. Για το λόγο αυτό παραλληλοποιησαμε την αρχικοποίηση του πίνακα objects στο αρχείο file_io.c. Πιο συγκεκριμένα:

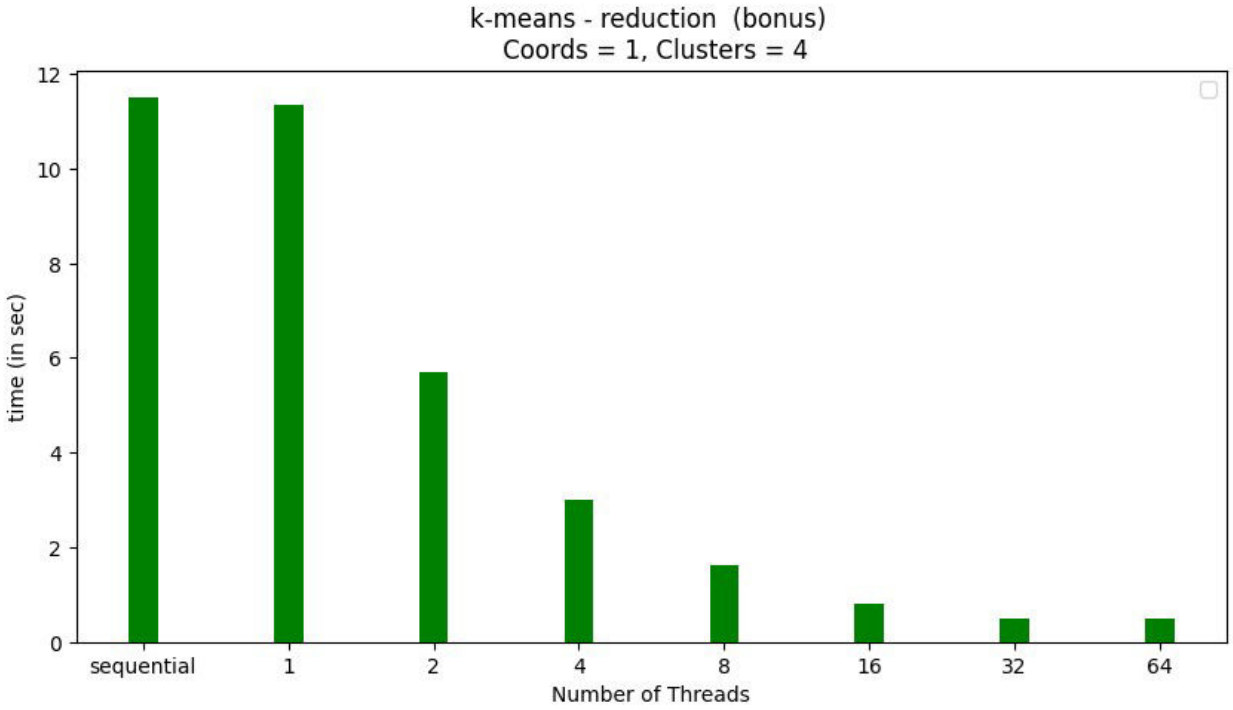
```
#pragma omp parallel for private(i, j)
for (i=0; i<numObjs; i++)
{
    unsigned int seed = i;
    for (j=0; j<numCoords; j++)
    {
        objects[i*numCoords + j] = (rand_r(&seed) / ((float) RAND_MAX)) * val_range;
        if (_debug && i == 0)
            printf("object[i=%ld][j=%ld]=%f\n",i,j,objects[i*numCoords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

Κατά αυτό το τρόπο κάθε thread φέρνει στην μνήμη του NUMA node που το “φιλοξενεί” τα δεδομένα του πίνακα objects που θα χρειαστεί. Σημαντικό ρόλο παίζει εδώ το γεγονός ότι έχουμε κάνει malloc για τον πίνακα objects (και όχι calloc) με αποτέλεσμα να μην γίνεται τότε touch ο πίνακας των objects.

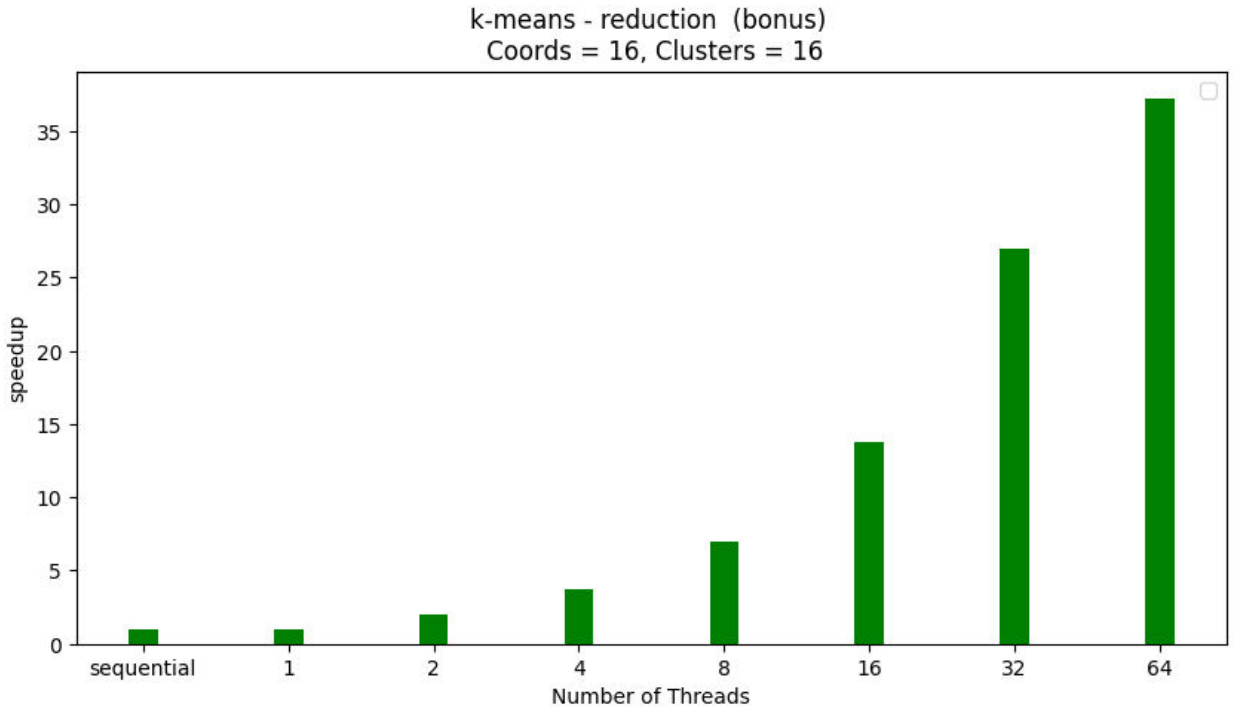
Τα αποτελέσματα με τη παραπάνω υλοποίηση (έχοντας παραλληλοποιήσει και το calloc των local πινάκων) είναι:

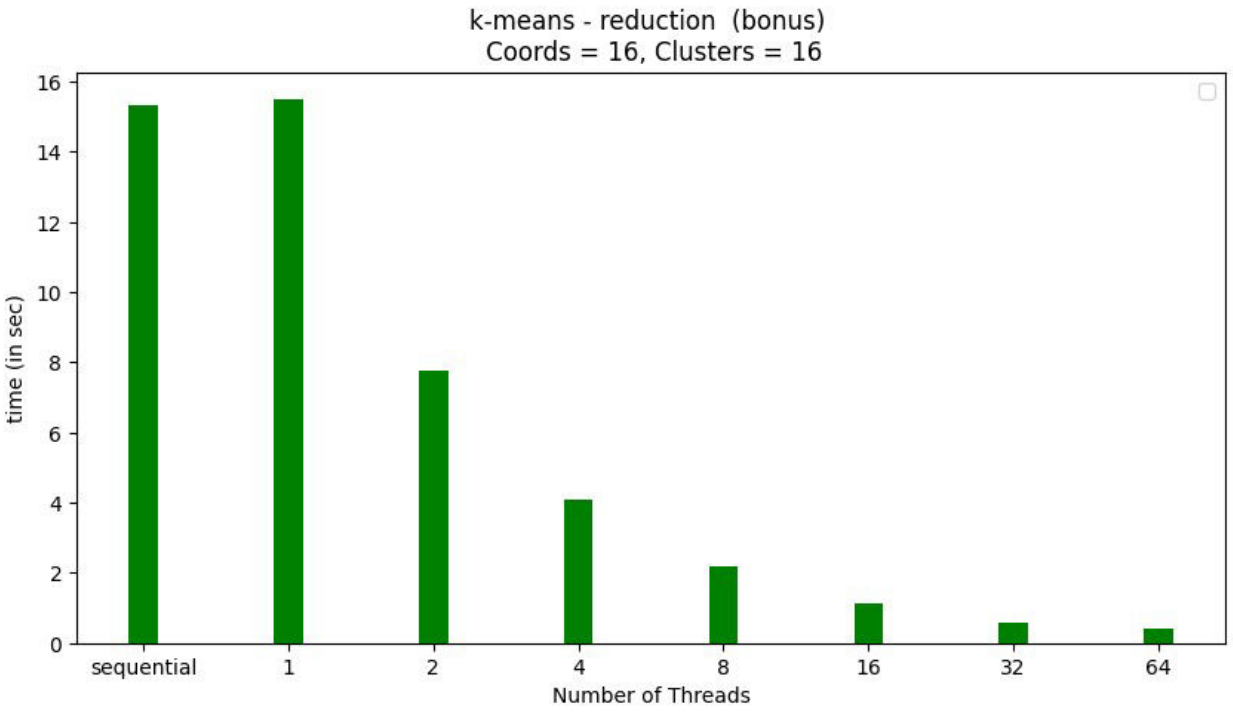
- Στο configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}





- Στο configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} :

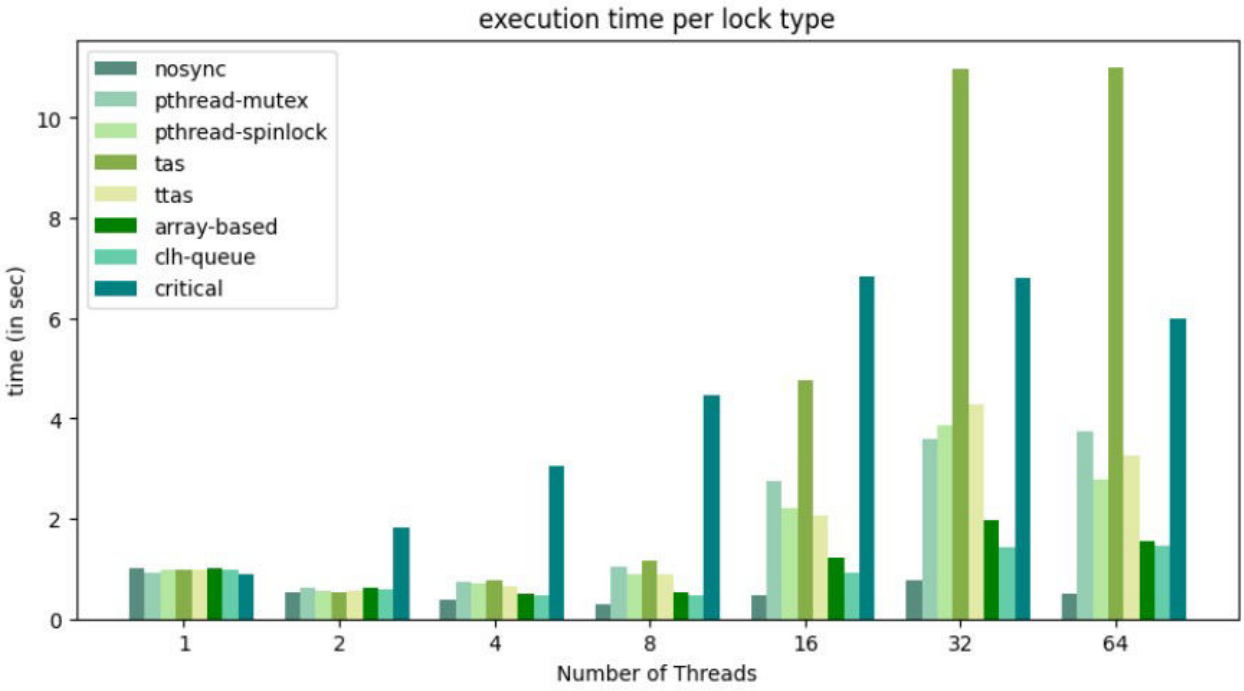

k-means - reduction (bonus)
Coords = 16, Clusters = 16
speedup
0
5
10
15
20
25
30
35
sequential
1
2
4
8
16
32
64
Number of Threads


k-means - reduction (bonus)
Coords = 16, Clusters = 16
time (in sec)
0
2
4
6
8
10
12
14
16
sequential
1
2
4
8
16
32
64
Number of Threads

Η βελτίωση που πετυχαίνουμε στο speedup δεν είναι σημαντική που εξηγείται από το γεγονός ότι προσπαθούμε να μειώσουμε τον χρόνο που απαιτείται για να φέρει κάθε thread τα δεδομένα που χρειάζεται στην cache του numa node όπου φιλοξενείται. Στις επόμενες εποχές πιθανότατα κάθε thread θα βρει τα δεδομένα που χρειάζεται στην cache οπότε το γεγονός ότι τα δεδομένα πιθανώς βρίσκονταν σε μνήμη απομακρυσμένου numa node δεν επηρεάζει χρονικά την εκτέλεση. Επομένως στο configuration {16, 16} δεν υπάρχει αισθητή βελτίωση (ελάχιστα στα 64 threads), καθώς οι προσβάσεις στον πίνακα objects είναι πολύ πιο χρονοβόρες, αφού είναι πολλές, από τον χρόνο που κερδίζουμε έχοντας φέρει το κάθε κομμάτι του πίνακα objects στο κοντινότερο NUMA node. Ωστόσο, στο configuration {1, 4} παρατηρούμε μια πιο αισθητή βελτίωση στο speedup (για 32, 64 threads έχουμε speedup >= 20), καθώς ο πίνακας objects είναι μικρός και ο χρόνος μεταφοράς των δεδομένων από τον NUMA node στην cache είναι συγκρίσιμος με τον συνολικό χρόνο προσβάσεων.

Όσον αφορά το bottleneck σε κάθε configuration, όταν έχουμε πολλά δεδομένα (configuration {16, 16}) πρόκειται για το πλήθος των threads που μπορεί να υποστηρίξει το sandman ενώ στο μικρό configuration είναι η μνήμη. Αναλυτικότερα, το sandman υποστηρίζει συνολικά 64 threads (έχει 4 nodes με 8 cores που υποστηρίζουν 2 threads) οπότε μπορούμε στην παραλληλοποίηση να εργάζονται μέχρι 64 threads. Όσο αυξάνουμε το πλήθος τους πετυχαίνουμε καλύτερα αποτελέσματα. Περιμένουμε ότι θα είχαμε ακόμα μικρότερους χρόνους σε μηχάνημα που υποστηρίζει μεγαλύτερο πλήθος threads. Στο μικρό configuration είναι θέματα μνήμης που δεν μας επιτρέπουν να μειώσουμε σημαντικά τον χρόνο για αυτό και κάθε προσπάθειά μας να τα λύσουμε έχει τόσο ικανοποιητικά αποτελέσματα.

Αμοιβαίος Αποκλεισμός – Κλειδώματα



Το nosync χρησιμοποιείται για να έχουμε ένα baseline στη σύγκριση των χρόνων.

Αρχικά, το mutex της pthread γενικά δεν δίνει καλούς χρόνους. Αυτό συμβαίνει, καθώς είναι υλοποιημένο με sleep στα threads που σημαίνει πως ο χρόνος που παραμένουν κοιμισμένα μαζί με το overhead που προστίθεται για το sleep/wake επιβαρύνουν την εκτέλεση.

Το spinlock της pthread υλοποιείται ως εξής:

```
  do
    {
      /* The lock is contended and we need to wait.  Going straight back
         to cmpxchg is not a good idea on many targets as that will force
         expensive memory synchronizations among processors and penalize other
         running threads.
         There is no technical reason for throwing in a CAS every now and then,
         and so far we have no evidence that it can improve performance.
         If that would be the case, we have to adjust other spin-waiting loops
         elsewhere, too!
         Thus we use relaxed MO reads until we observe the lock to not be
         acquired anymore.  */
      do
        {
          /* TODO Back-off.  */

          atomic_spin_nop ();

          val = atomic_load_relaxed (lock);
        }
      while (val != 0);

      /* We need acquire memory order here for the same reason as mentioned
         for the first try to lock the spinlock.  */
    }
  while (!atomic_compare_exchange_weak_acquire (lock, &val, 1));

  return 0;
}
```

(https://codebrowser.dev/glibc/glibc/nptl/pthread_spin_lock.c.html)

Γενικά το spinlock κάνει scale καλύτερα από τα mutex, tas, ttas κλειδώματα. Ωστόσο, είναι πολύ κοντά στην επίδοση με το ttas και παρουσιάζουν παρόμοια συμπεριφορά, γιατί έχουν παρόμοια υλοποίηση, ωστόσο το spinlock της pthread χρησιμοποιεί κάποιες βελτιστοποιημένες εντολές assembly για πιο

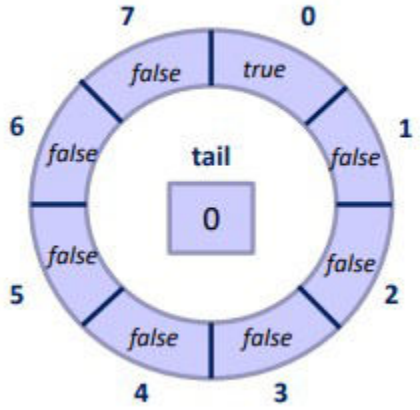
γρήγορες προσβάσεις στη μνήμη (πχ. Atomic_load_relaxed). Αναλυτικότερα θα εξηγηθεί η επίδοση του στον σχολιασμό του ttas.

Το tas κλείδωμα κάνει το χειρότερο scale από όλα τα κλειδώματα. Αυτό συμβαίνει, καθώς γράφει συνεχώς ατομικά μία μεταβλητή που είναι διαμοιραζόμενη μεταξύ όλων των threads και επομένως ενεργοποιείται συνεχώς το cache coherence protocol με αποτέλεσμα να υπάρχει μεγάλη κίνηση στο bus και έτσι και μεγάλο latency.

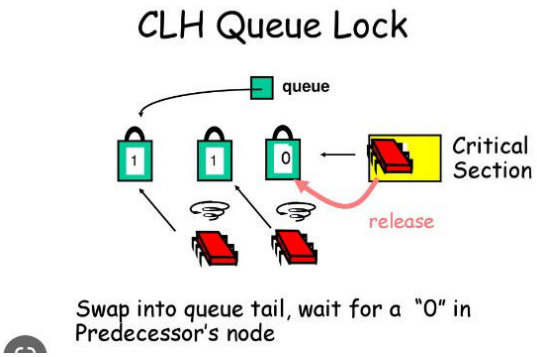
Από την άλλη, το ttas έχει καλύτερη επίδοση από το tas, καθώς στην υλοποίηση του, πρώτα ελέγχει αν η μεταβλητή-κλειδί είναι ελεύθερη πριν προσπαθήσει να την γράψει. Επομένως, κάνει τοπικά reads στην μεταβλητή και δεν χρειάζεται να δημιουργήσει αχρείαστη κίνηση στο bus επιβαρύνοντας τον χρόνο εκτέλεσης. Δοκιμάζει να γράψει τη μεταβλητή μόνο όταν ελευθερωθεί.

Τα κλειδώματα με το καλύτερο scaling είναι τα array-based και clh queue locks.

Array-Based concept:



CLH queue concept:



Αυτά τα 2 locks έχουν τις καλύτερες επιδόσεις, καθώς κάθε thread που θέλει να μπει στο κρίσιμο τμήμα θα ενεργοποιήσει το πρωτόκολλο cache coherence μία φορά, μόνο όταν έρθει η σειρά του να μπει. Αυτό συμβαίνει, γιατί τα threads περιμένουν με μία προκαθορισμένη σειρά να μπουν στο κρίσιμο τμήμα και μπαίνουν μόνο όταν ειδοποιηθούν από τον predecessor τους. Στο array-based αυτό θα γίνει όταν ο προηγούμενος κάνει true το flag του (βλ. σήμα), ενώ στο CLH όταν ο προηγούμενος κάνει False το δικό του flag. Ο CLH κάνει spin στο flag του predecessor του στην λίστα και περιμένει να γίνει False. Επομένως και στις 2 υλοποιήσεις το κάθε thread ελέγχει μία τοπική μεταβλητή, είτε δική του (array) είτε του predecessor του (clh). Αυτό, επίσης, εξασφαλίζει πως το cache coherence πρωτόκολλο θα ενεργοποιηθεί τις ελάχιστες δυνατές φορές και επομένως δεν θα υπάρχει αχρείαστη κίνηση στο bus. Επιπλέον, βλέπουμε στον κώδικα που μας δόθηκε που έχει γίνει και padding στις θέσεις του πίνακα/λίστας με βάση την cache line, ώστε να αποφευχθεί το false sharing.

Array-Based:

```
struct lock_struct {
        volatile char *flag;
        char padding1[64 - sizeof(char *)];
        unsigned long long tail;
        char padding2[64 - sizeof(unsigned long long)];
        int size;
};
```

CLH:

```
typedef struct {
        volatile char locked; /* FALSE or TRUE. */
//      char padding[63];
        char padding[1023];
} clh_node_t;
```

Αυτές οι 2 υλοποιήσεις είναι starvation free, δηλαδή προσδίδουν μια προτεραιότητα FIFO στα threads που προσπαθούν να πάρουν το κλειδί και εξασφαλίζουν πως όλα τα threads θα εξυπηρετηθούν κάποια στιγμή.

Η διαφορά μεταξύ των 2 είναι πως το clh απαιτεί σημαντικά λιγότερη μνήμη, καθώς υπάρχουν στη μνήμη κάθε φορά όσα nodes όσα και τα threads που θέλουν να μπουν στο κρίσιμο τμήμα, ενώ στο array-based το μέγεθος είναι προκαθορισμένο.

Παρατηρούμε πως το critical της OMP έχει τη $2^{η}$ χειρότερη επίδοση, αφού είναι καλύτερο μόνο από το tas. Γι' αυτό και στην προηγούμενη άσκηση το πρόγραμμα έκανε πολύ κακό scale μετά τα 8 threads.

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Σκοπός της άσκησης

Στόχος της άσκησης είναι να εξοικιωθείτε με τη χρήση των OpenMP tasks παράλληλοποιώντας τον αλγόριθμο Floyd-Warshall για αρχιτεκτονικές κοινής μνήμης.

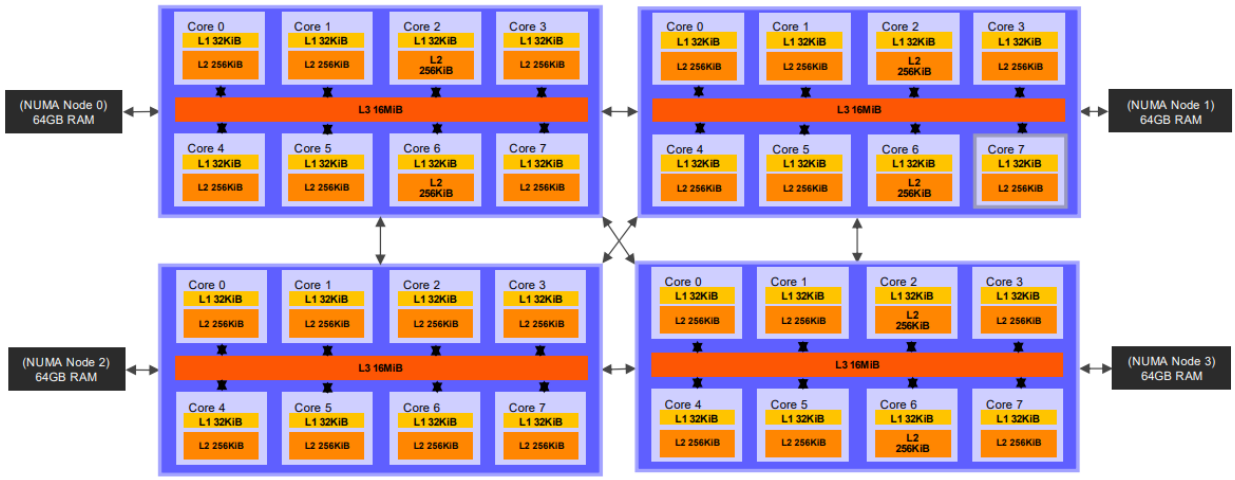
Ζητούμενο : Παραλληλοποίηση recursive αλγορίθμου

Στο ερώτημα αυτό παραλληλοποιήσαμε τον recursive αλγόριθμο χρησιμοποιώντας τα tasks που μας παρέχει το OpenMP. Η υλοποίησή μας φαίνεται παρακάτω:

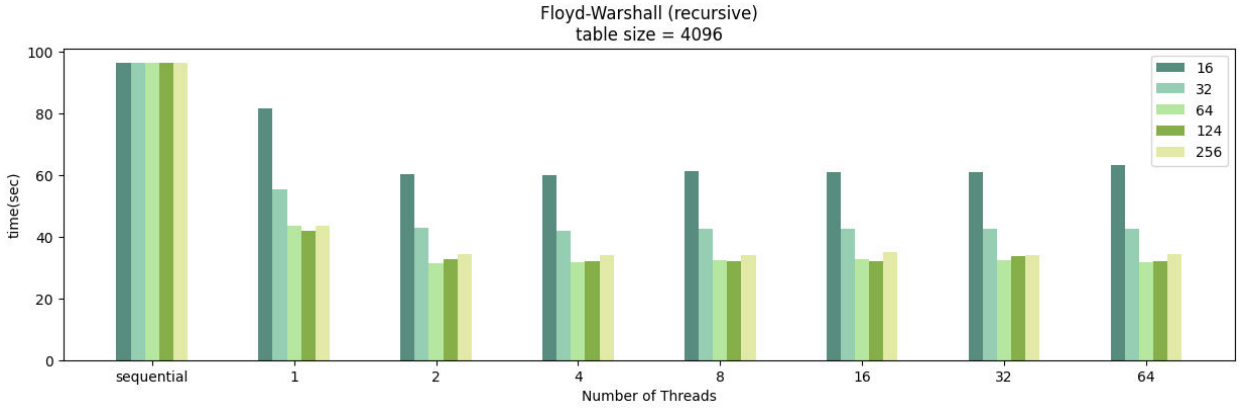
```
#pragma omp parallel
{
    #pragma omp single
    {
       FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
       #pragma omp task
       FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
       #pragma omp task
       FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
       #pragma omp taskwait
       FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize);
       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 task
       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
       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
       FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
    }
}
```

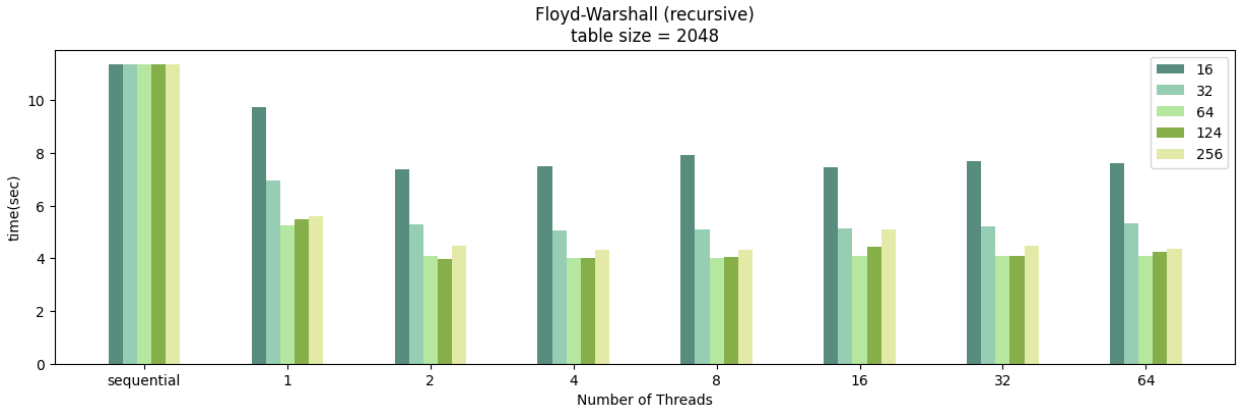
Ο πίνακας χωρίζεται σε 4 ίδιου μεγέθους υποπίνακες (A00, A01, A10, A11) όπως απεικονίζεται στο παραπάνω σχήμα. Μπορούμε να τρέχουμε παράλληλα τα FW_SR που κάνουν υπολογισμούς στους δύο διαγώνιους υποπίνακες (A01, A10) καθώς δεν χρειάζονται οι υπολογισμοί του ενός για εκείνους του άλλου. Στον παραπάνω κώδικα υπολογίζονται τα στοιχεία του A00, στην συνέχεια δημιουργούνται δύο tasks ένα για καθένα από τα A01 και A10 και **αφού** αυτά ολοκληρώσουν τους υπολογισμούς (εξασφαλίζεται αυτό με το taskwait) υπολογίζεται το A11, πάλι δημιουργούνται δύο tasks για τα A10 και A01 και τέλος αφού ολοκληρωθούν αυτοί οι υπολογισμοί (taskwait) υπολογίζεται το A00.

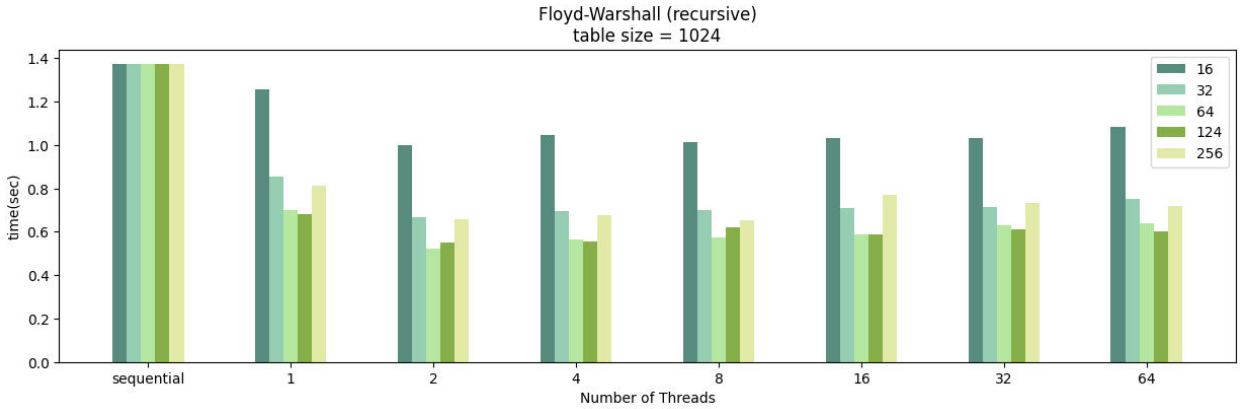
Ο υπολογισμός για κάθε ένας από τους παραπάνω υποπίνακες γίνεται αναδρομικά μέχρι να φτάσουμε το base case (μέγεθος υποπίνακα block size). Δοκιμάσαμε διαφορετικά block sizes: 32, 64, 124, 256. Στο παρακάτω σχήμα συνοψίζονται οι γνώσεις μας για την χωρητικότητα των μνημών του μηχανήματος sandman όπου τρέχουμε το πρόγραμμά μας:



Με βάση τις πληροφορίες αυτές, για block size = 32 όλα τα δεδομένα του base case χωράνε στην L1 cache ενώ για 64 και 124 χωράνε στην L2 (το 256 είναι πολύ μεγάλο και χωράει ολόκληρο μόνο στην L3). Περιμένουμε ότι θα έχουμε την καλύτερη επίδοση όταν τα δεδομένα για τον υπολογισμό του base case του αναδρομικού αλγορίθμου χωράνε στην L1 είτε στην L2 (η L3 είναι διαμοιραζόμενη μεταξύ των cores ενός numa node).







Τα αποτελέσματα που προκύπτουν απεικονίζονται στα παραπάνω διαγράμματα.

Παρατηρούμε ότι για μέγεθος πίνακα:

- 1024 x 1024 πετυχαίνουμε το καλύτερο αποτέλεσμα για 2 threads και block size = 64
- 2048 x 2048 πετυχαίνουμε το καλύτερο αποτέλεσμα για 2 threads και block size = 124
- 4096 x 4096 πετυχαίνουμε το καλύτερο αποτέλεσμα για 2 threads και block size = 64

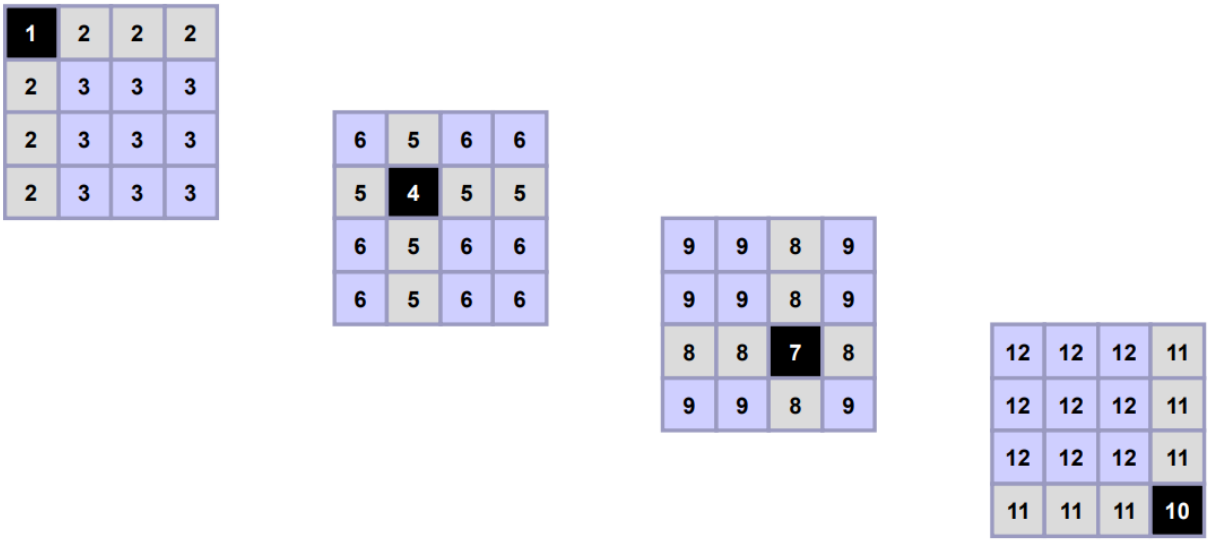
Παρατηρούμε ότι σε κάθε περίπτωση για block size = 124 ή 64 έχουμε τα καλύτερα αποτελέσματα. Γενικά θέλουμε το base case να αρκετά μεγάλο ώστε να γίνονται όσο το δυνατόν περισσότεροι υπολογισμοί παράλληλα με parallel for αλλά όχι τόσο μεγάλο που να μην χωράνε τα στοιχεία του base case υποπίνακα στις L1 ή L2. Επίσης, όσον αφορά το πλήθος των threads παρατηρούμε ότι πετυχαίνουμε τους μικρότερους χρόνους για 2 threads. Για μεγαλύτερο πλήθος threads, οι χρόνοι

εκτέλεσης παραμένουν σχεδόν ίδιοι (έχουμε μία ελαφρύ αύξηση), δηλαδή δεν φαίνεται να κλιμακώνει. Δεν γνωρίζουμε για ποιο λόγο εμφανίζει ο αλγόριθμος αυτή την συμπεριφορά όμως σε κάθε επανάληψη δημιουργούνται δύο νέα παράλληλα tasks (αυτά που εκτελούν τον υπολογισμό των στοιχείων των δύο διαγώνιων μπλοκ σε κάθε αναδρομική κλήση). Ο αλγόριθμος αναδρομικα αυξάνει τον αριθμό των παράλληλων tasks και θεωρητικά θα έπρεπε να επωφελείται από την ύπαρξη περισσότερων threads, κάτι που δεν παρατηρήσαμε (βλέπε τα παραπάνω διαγράμματα).

Για μέγεθος πίνακα 1024 x 1024 παρατηρούμε ότι το speedup που πετυχαίνουμε είναι κάπως μικρότερο από αυτό που έχουμε για μεγαλύτερο μέγεθος πίνακα. Αυτό είναι λογικό καθώς όσο λιγότερους υπολογισμούς έχουμε τόσο πιο αισθητό είναι το overhead της δημιουργίας των tasks, του συγχρονισμού των threads και της κατανομής του φόρτου εργασίας σε αυτά (task queue).

Bonus Ερώτημα: Παραλληλοποίηση tiled αλγορίθμου

Ο αλγόριθμος αυτός σπάει τον αρχικό πίνακα σε υποπίνακες το μέγεθος των οποίων προσδιορίζεται από το block size. Για κάθε υποπίνακα που ανήκει στην διαγώνιο (βλέπε σχήμα παρακάτω) υπολογίζει τις ελάχιστες αποστάσεις. Στην συνέχεια αφού τις υπολογίσει, προχωράει σε υπολογισμούς των γκρι υποπινάκων τους οποίους χρησιμοποιεί για να υπολογίσει τους μοβ.



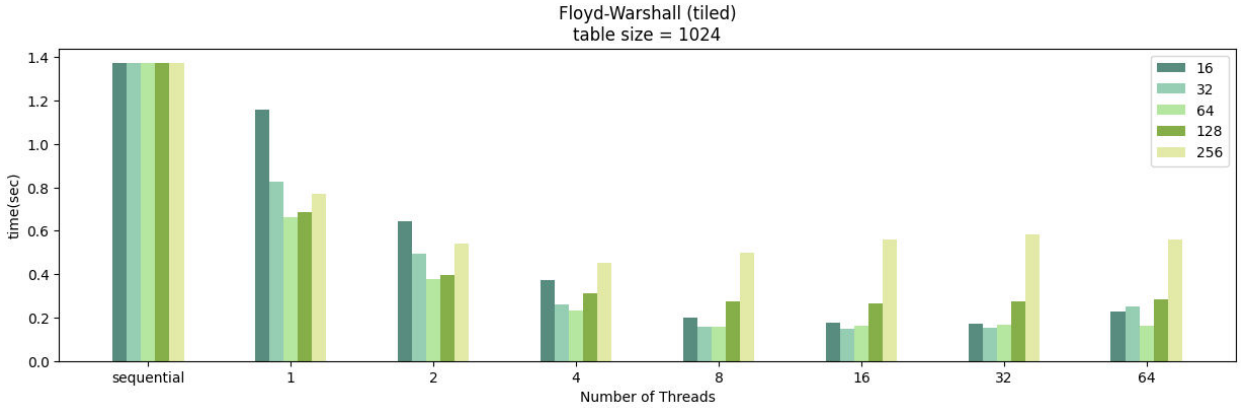
Για το ερώτημα αυτό παραλληλοποιήσαμε τον υπολογισμό αυτό με χρήση parallel for όπως φαίνεται παρακάτω.

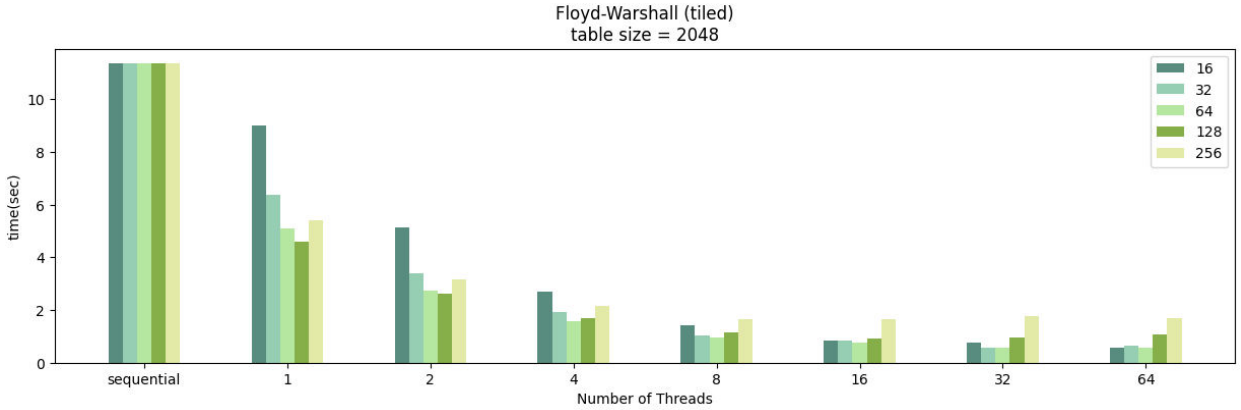
```
for(k=0;k<N;k+=B){
        FW(A,k,k,k,B);
        #pragma omp parallel
        {
                #pragma omp for private(i) nowait
                for(i=0; i<k; i+=B)
                        FW(A,k,i,k,B);

                #pragma omp for private(i) nowait
                for(i=k+B; i<N; i+=B)
                        FW(A,k,i,k,B);

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

                #pragma omp for private(j)
                for(j=k+B; j<N; j+=B)
                        FW(A,k,k,j,B);

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

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

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

                #pragma omp for private(i, j) nowait
                for(i=k+B; i<N; i+=B)
                        //#pragma omp private(j) nowait
                        for(j=k+B; j<N; j+=B)
                                FW(A,k,i,j,B);
        }
}
```

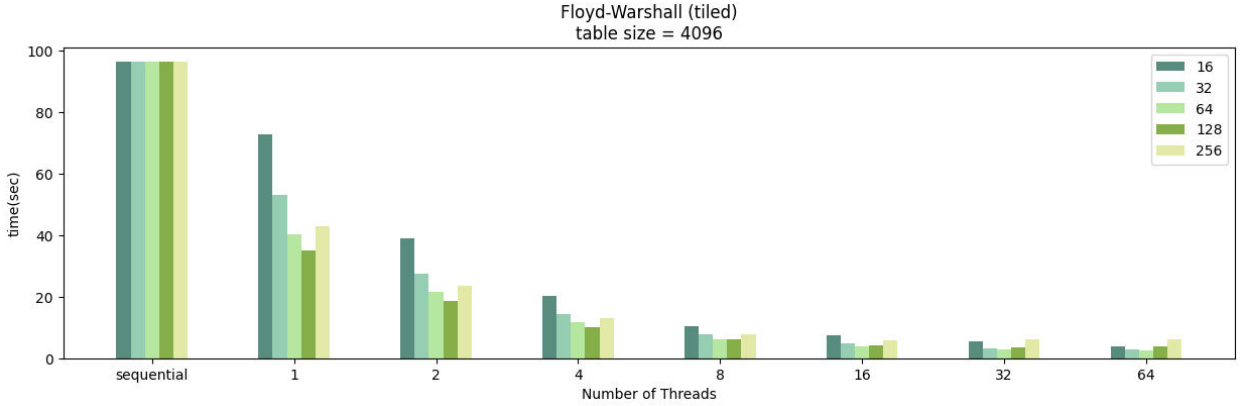
implicit barrier

Το 1ο loop υπολογίζει τις αποστάσεις στο μαύρο υποπίνακα, τα επόμενα 2 στα γκρι και τα τελευταία 4 loop στα μοβ. Το parallel for θέτει implicit barrier το οποίο καθυστερεί την εκτέλεση αχρείαστα καθώς το 2ο και το 3ο loop μπορούν να εκτελούνται παράλληλα (δεν υπάρχει εξάρτηση δεδομένων μεταξύ τους). Αφού ολοκληρωθούν (δεν βάζουμε nowait στο 3ο loop) μπορούν τα 4 τελευταία loop να εκτελεστούν ταυτόχρονα.

Τα διαγράμματα που συνοψίζουν τα αποτελέσματα που πετύχαμε παρατίθενται παρακάτω.







Παρατηρούμε ότι για μέγεθος πίνακα:

- 1024 x 1024 πετυχαίνουμε το καλύτερο αποτέλεσμα για 64 threads και block size = 64
- 2048 x 2048 πετυχαίνουμε το καλύτερο αποτέλεσμα για 64 threads και block size = 64
- 4096 x 4096 πετυχαίνουμε το καλύτερο αποτέλεσμα για 64 threads και block size = 64

Εδώ σε αντίθεση με την recursive υλοποίηση η αύξηση του πλήθους των threads μειώνει σημαντικά τον χρόνο εκτέλεσης (για 64 έχουμε και στις 3 περιπτώσεις τα καλύτερα αποτελέσματα), δηλαδή έχουμε πολύ ικανοποιητικό scaling ειδικά μέχρι τα 32 threads.

Το tiled πετυχαίνει πολύ καλύτερους χρόνους συγκριτικά με το recursive γεγονός που εξηγείται εν μέρη από την χρήση parallel for αντί για tasks (το οποίο είναι σημαντικά γρηγορότερο). Επίσης, στα 4

τελευταία for loops (διπλά) κάθε thread υπολογίζει μία γραμμή μιας περιοχής (βλέπε παρακάτω σχήμα) οπότε εκμεταλλευόμαστε καλύτερα το locality των δεδομένων. Να σημειώσουμε ότι πειραματιστήκαμε και με παραλληλοποίηση του εσωτερικού loop των 4 τελευταίων διπλών loop και παρατηρήσαμε ότι αυξήθηκε ο χρόνος εκτέλεσης, γεγονός που επιβεβαιώνει την σημασία του data locality στην μείωση του χρόνου εκτέλεσης.

2.3 Ταυτόχρονες δομές δεδομένων

Για 128 threads έχουμε 2 threads ανά core που ανταγωνίζονται για τα resources (κυρίως cache) οπότε μειώνεται το throughput για όλες τις υλοποιήσεις.

Coarse-grain locking:

Παρατηρούμε ότι όσο αυξάνονται τα χρησιμοποιούμενα threads το throughput μειώνεται (για όλα τα sizes και όλα τα workloads). Αυτό είναι λογικό καθώς δεν είναι scalable το κλείδωμα ολόκληρης της δομής (σε οποιοδήποτε operation) όταν ενδιαφέρει η τροποποίηση μόνο ενός κομματιού της. Όσο περισσότερα είναι τα threads τόσο περισσότερο εμφανής γίνεται η αχρείαστη αποκλειστικότητα ολόκληρης της δομής που δίνεται σε ένα thread για να κάνει το operation του.

Fine-Grain Locking:

Το throughput που πετυχαίνει είναι παρόμοιο για όλα τα workloads. Αυξάνοντας το μέγεθος της λίστας καθώς γίνεται πιο χρονοβόρα η αναζήτηση στοιχείου αυξάνεται ο χρόνος εκτέλεσης. Επίσης, καλύτερο throughput έχουμε για 8 και 64 threads αν η διαφοροποίηση ανάλογα με το πλήθος threads στο throughput δεν είναι σημαντική. Σε σχέση με το coarse-grain μέχρι και τα 8 threads το throughput του είναι μικρότερο καθώς είναι μεγάλο το overhead των πολλαπλών lock και unlock που κάνει για να καταφέρει να κλειδώσει μόνο το κομμάτι της λίστας που τον ενδιαφέρει. Για μεγαλύτερο πλήθος threads το locking ολόκληρης της δομής που κάνει το coarse grain το καθιστά υποδεέστερο του fine-grain locking.

Optimistic synchronization:

Παρατηρούμε ότι με εξαίρεση το workload 100/0/0 (μόνο contains operations) και για τα δύο list sizes λόγω του **local locking** καταφέρνει να κάνει scale καλά σε περισσότερα threads : το throughput αυξάνεται συνεχώς μέχρι τα 128 threads που ξαναπέφτει ειδικά για το μικρότερο size (οι εναλλαγές επηρεάζουν περισσότερο όταν έχουμε μικρότερη λίστα). Από 8 threads το throughput που πετυχαίνει είναι πολύ καλύτερο σε σχέση με τις προηγούμενες υλοποιήσεις.

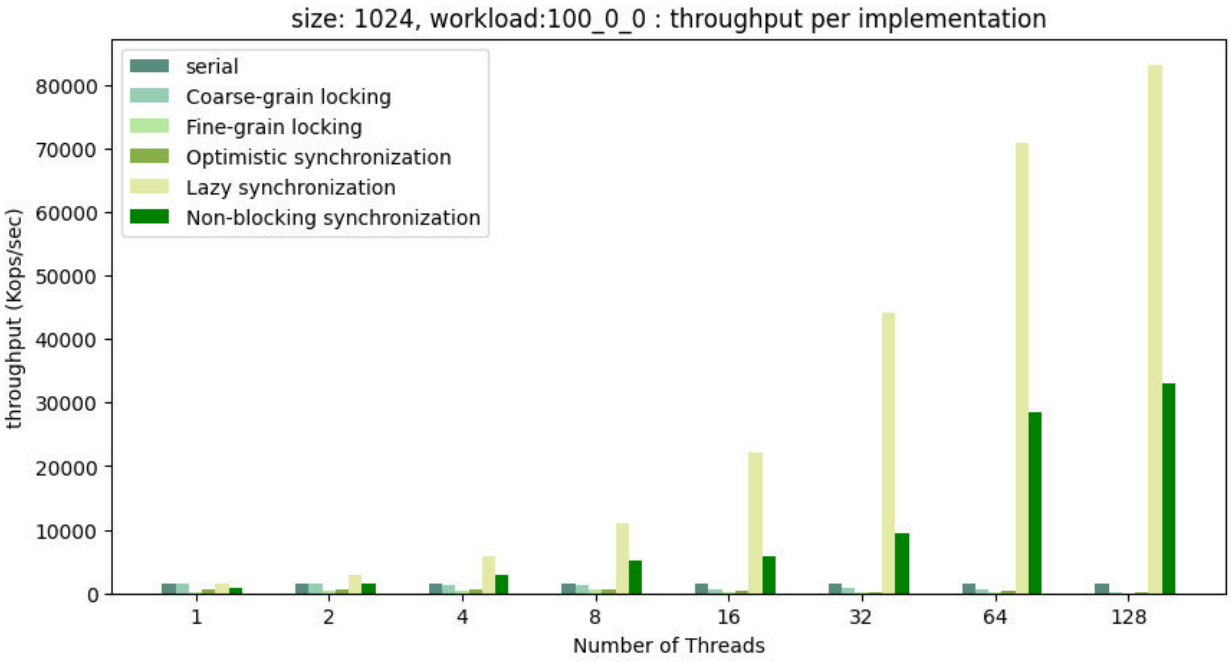
Lazy synchronization:

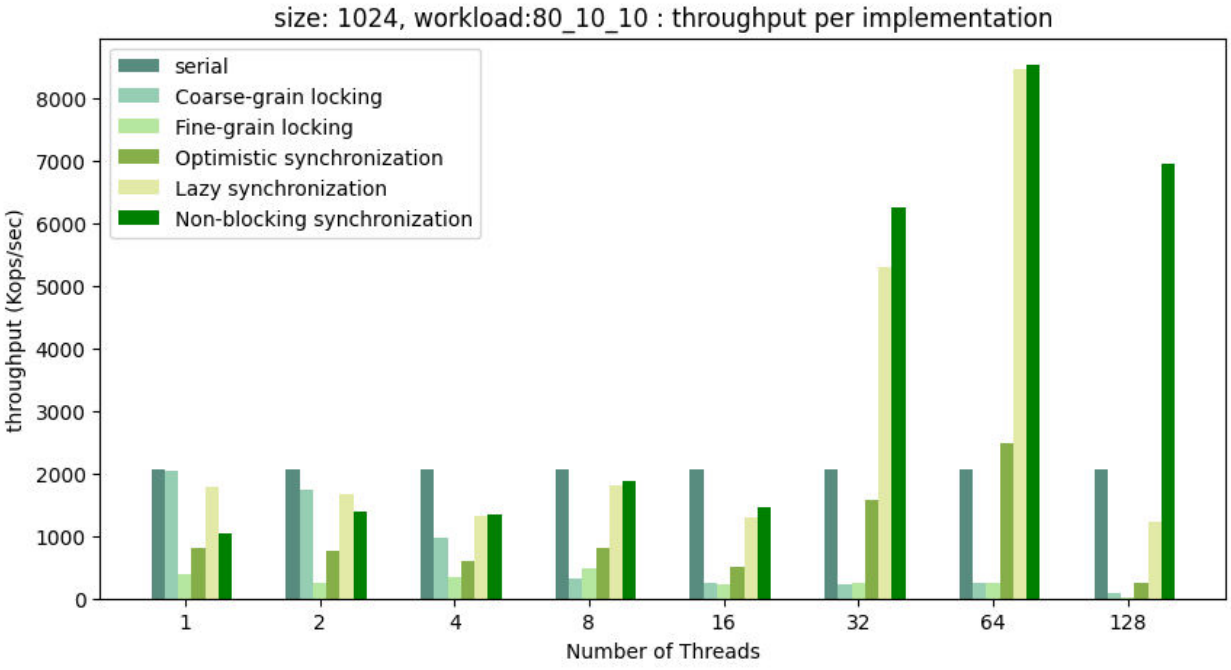
Παρατηρούμε ότι και για τα δύο list sizes και για όλα τα workloads κάνει πολύ καλό scale στα περισσότερα threads (και αυτό από 1 μέχρι 64) με μεγάλη αύξηση να σημειώνεται από τα 16 στα 64. Όπως και το optimistic κάνει local locking για να κάνει add και remove όμως σε αντίθεση με το optimistic χωρίς επαναδιάσχιση της λίστας από την αρχή, με local checks στο validation. Για αυτό και στο workload 0/50/50 (που έχει μόνο add και remove operations) έχει καλύτερη επίδοση σε σχέση με το optimistic. Επίσης, η contains δεν γίνεται με locking που είναι εμφανές από την ανωτερότητα της επίδοσης στο workload 100/0/0 (μόνο contains operations).

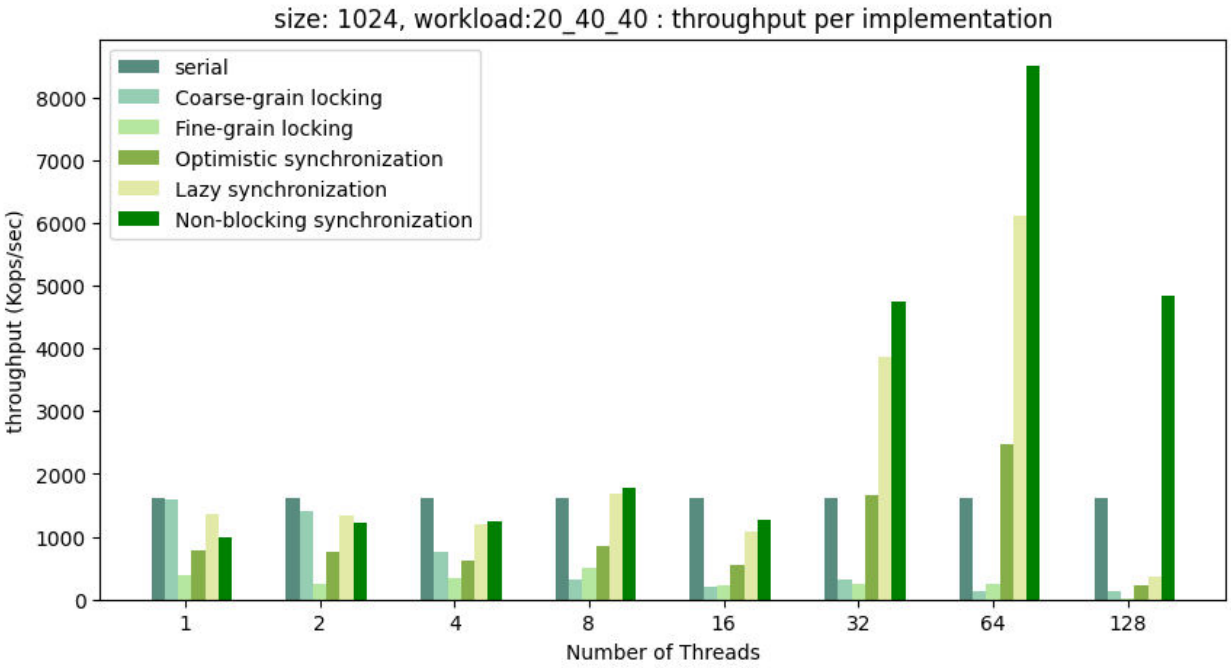
Non-blocking synchronization:

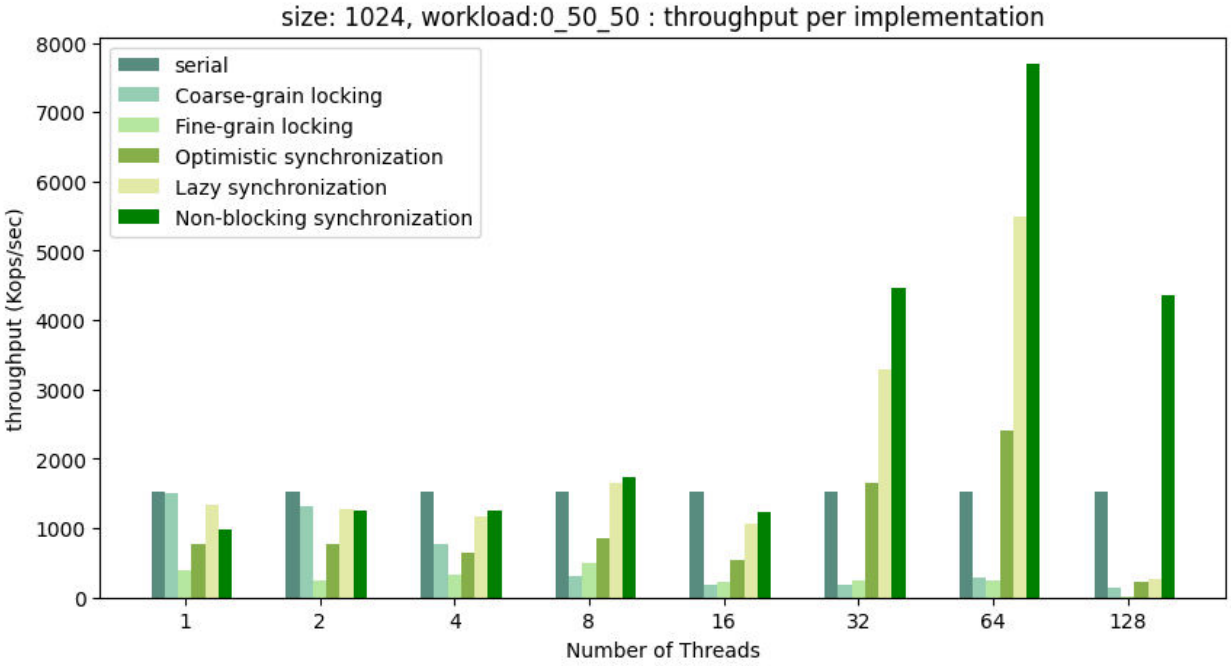
Και πάλι παρατηρούμε ότι κάνει πολύ καλό scaling. Δεν υπάρχουν κλειδώματα και χρησιμοποιεί ατομικές εντολές μόνο εκεί που χρειάζεται. Στα 128 threads μπορεί πάλι να πέφτει από τα 64 αλλά πετυχαίνει πολύ καλύτερο throughput σε σχέση με όλες τις προηγούμενες υλοποιήσεις. Για το workload 100/0/0 το throughput είναι χαμηλότερο μόνο από εκείνο του lazy synchronization και αυτό γιατί στην contains γίνεται ατομικά ο έλεγχος για την εύρεση και την ύπαρξη του στοιχείου αφού έχει διασχίσει lock-free την λίστα μέχρι να το βρει.

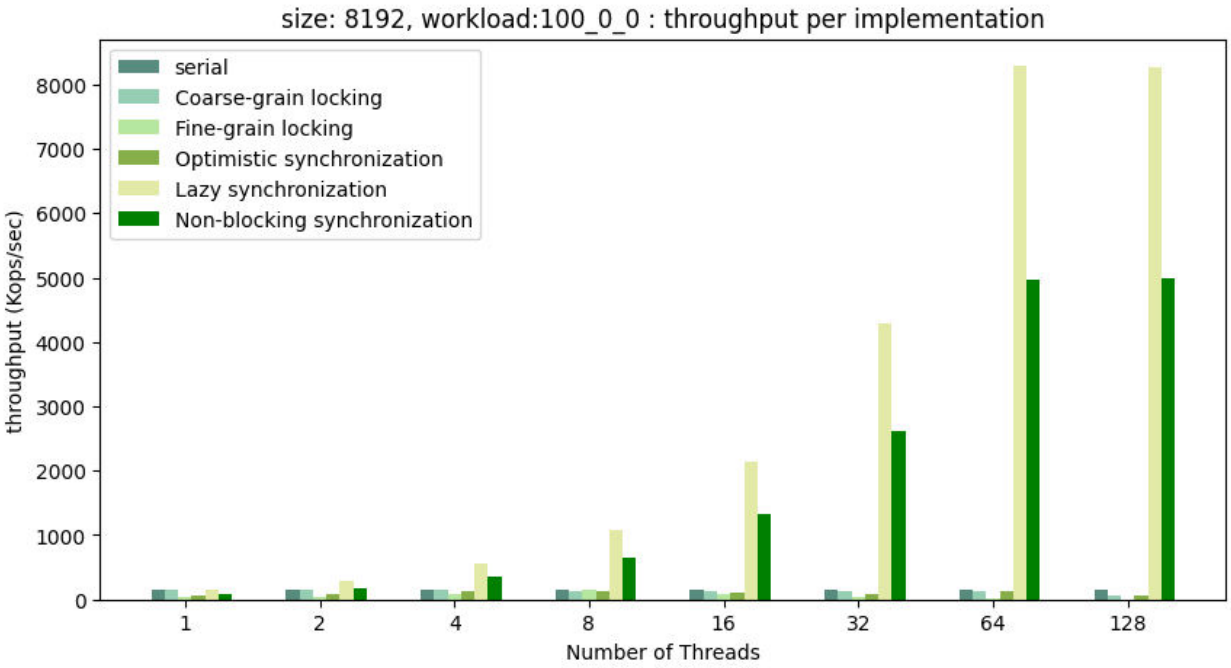
```
public boolean contains(Tt item) {
  boolean marked;
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key)
    curr = curr.next;
  Node succ = curr.next.get(marked);
  return (curr.key == key && !marked[0])
}
```

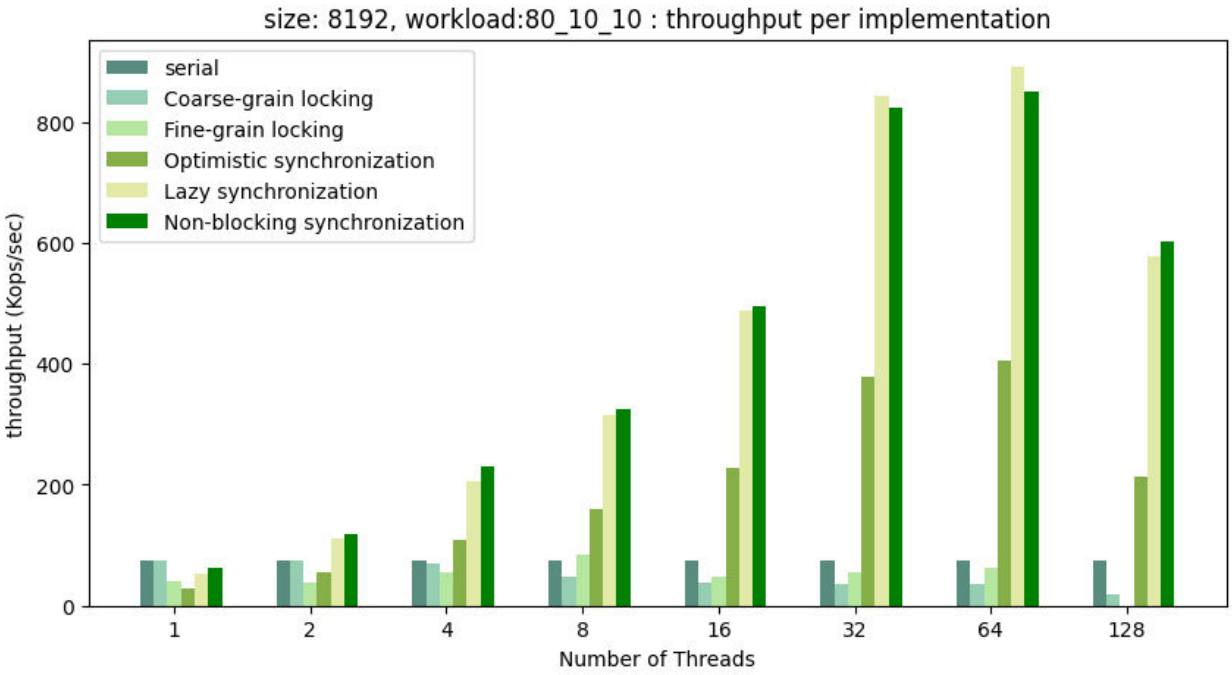

size: 1024, workload:100_0_0 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
10000
20000
30000
40000
50000
60000
70000
80000
1
2
4
8
16
32
64
128
Number of Threads

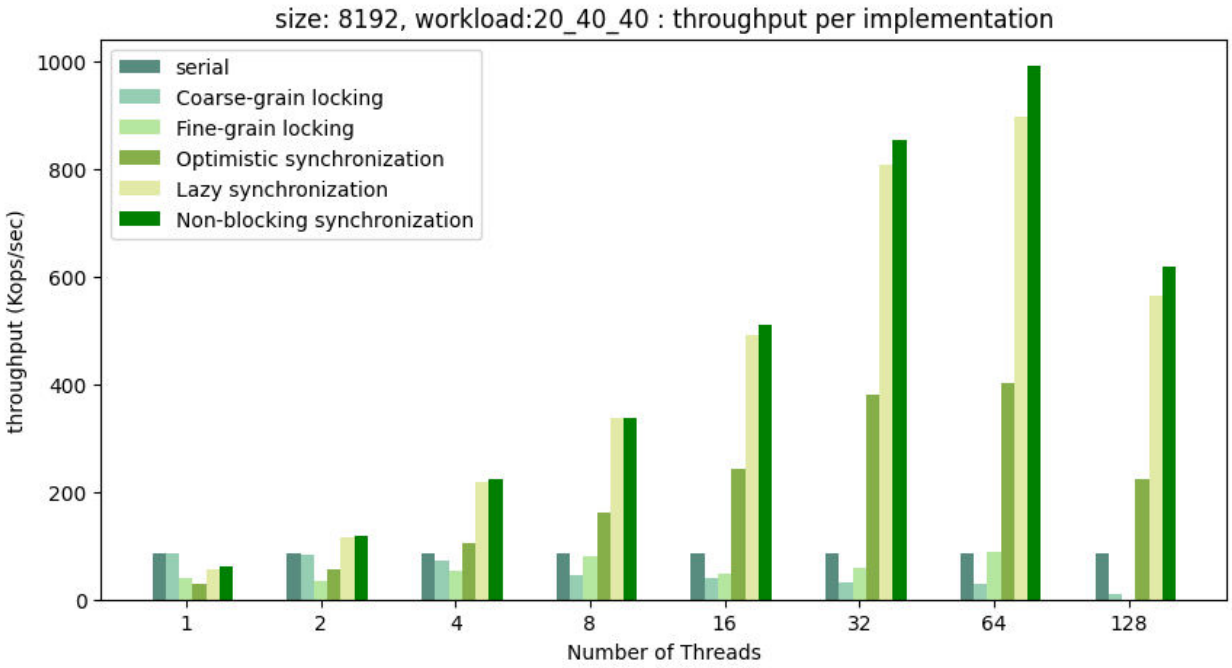

size: 1024, workload:80_10_10 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
1000
2000
3000
4000
5000
6000
7000
8000
1
2
4
8
16
32
64
128
Number of Threads

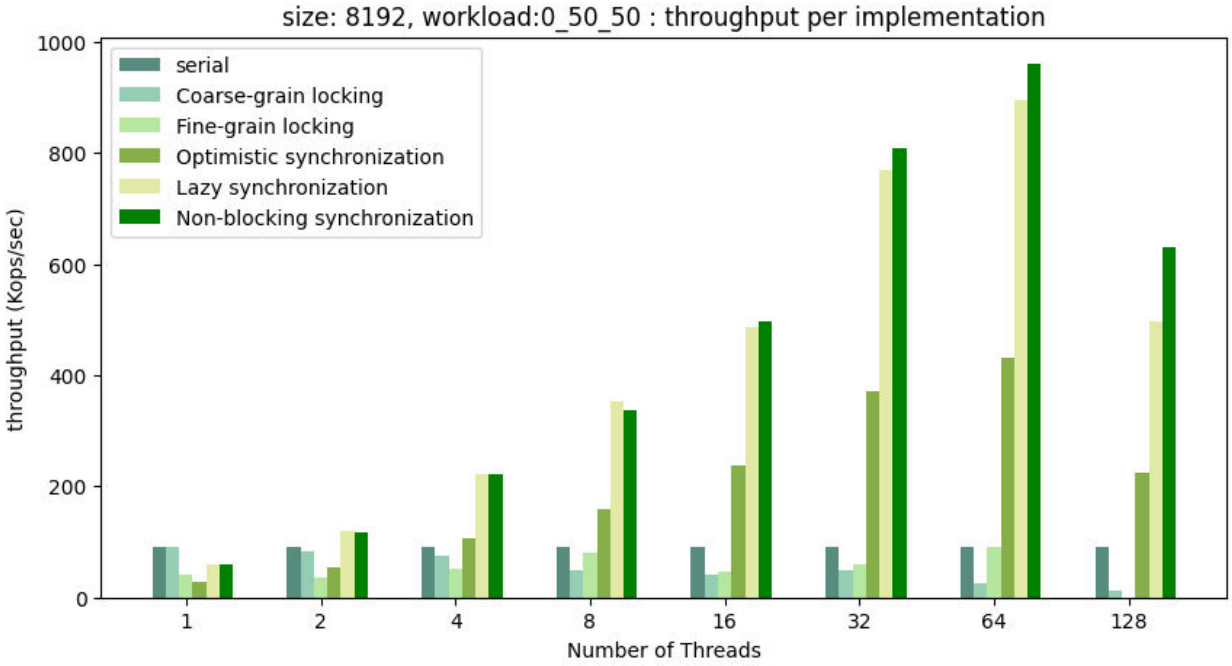

size: 1024, workload:20_40_40 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
1000
2000
3000
4000
5000
6000
7000
8000
1
2
4
8
16
32
64
128
Number of Threads


size: 1024, workload:0_50_50 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
1000
2000
3000
4000
5000
6000
7000
8000
1
2
4
8
16
32
64
128
Number of Threads


size: 8192, workload:100_0_0 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
1000
2000
3000
4000
5000
6000
7000
8000
1
2
4
8
16
32
64
128
Number of Threads


size: 8192, workload:80_10_10 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
200
400
600
800
1
2
4
8
16
32
64
128
Number of Threads


size: 8192, workload:20_40_40 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
200
400
600
800
1000
1
2
4
8
16
32
64
128
Number of Threads


size: 8192, workload:0_50_50 : throughput per implementation
serial
Coarse-grain locking
Fine-grain locking
Optimistic synchronization
Lazy synchronization
Non-blocking synchronization
throughput (Kops/sec)
0
200
400
600
800
1000
1
2
4
8
16
32
64
128
Number of Threads

3. Παραλληλοποίηση αλγορίθμων σε επεξεργαστές γραφικών

3.1 Παραλληλοποίηση του αλγορίθμου K-means

Naive version

Αρχικά υλοποιήσαμε τη συνάρτηση get_tid() η οποία επιστρέφει το global ID ενός thread που την καλεί ως εξής:

```
__device__ int get_tid(){
        return blockDim.x*blockIdx.x + threadIdx.x;
```

Εφόσον τα threads μοιράζονται σε blocks χρησιμοποιούμε το ID του block που βρίσκεται το thread πολλαπλασιάζοντας με το μέγεθος ενός block για να πάρουμε το εύρος των ID που συμπεριλαμβάνει το συγκεκριμένο block και εκεί προσθέτουμε το local ID του thread.

Στη συνέχεια υλοποιούμε την euclid_dist_2 με την ίδια λογική που είχαμε δει στη CPU υλοποίηση του K-means (το μόνο που αλλάζει είναι τα ορίσματα):

```
float euclid_dist_2(int    numCoords,
                    int    numObjs,
                    int    numClusters,
                    float *objects,     // [numObjs][numCoords]
                    float *clusters,    // [numClusters][numCoords]
                    int    objectId,
                    int    clusterId)

{
    int i;
    float ans=0.0;

        /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem
=clusterId from clusters*/
    for(i=0; i<numCoords; i++){
        ans += (objects[objectId*numCoords + i] - clusters[clusterId*numCoords +
 i]) * (objects[objectId*numCoords + i] - clusters[clusterId*numCoords + i]);
    }
```

Στη συνέχεια υλοποιούμε την find_nearest_cluster. Η παραλληλοποίηση γίνεται σε αντιστοιχία με τον αριθμό των objects. Για αρχή λαμβάνουμε το global ID του thread που την καλεί και αν είναι μικρότερο του αριθμού των Objects τότε εισέρχεται στη διαδικασία υπολογισμού:

```
    /* Get the global ID of the thread. */
int tid = get_tid();

    /* TODO: Maybe something is missing here... should all threads run this? */
if (tid < numObjs) {
    int   index, i;
    float dist, min_dist;
```

Μετά το κάθε thread καλεί για το δικό της object (που αντιστοιχεί στο global tid) την euclid_dist_2 για το 1^{ο} cluster για αρχή και μετά με μία λούπα ελέγχονται οι αποστάσεις από όλα τα clusters προκειμένου να βρεθεί η ελάχιστη και ανανεώνεται ο πίνακας deviceMembership:

```
    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
    min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters, tid, 0);
    for (i=1; i<numClusters; i++) {
        /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
        dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters, tid, i);

        /* no need square root */
        if (dist < min_dist) { /* find the min and its array index */
            min_dist = dist;
            index    = i;
        }
    }

    if (deviceMembership[tid] != index) {
        /* TODO: Maybe something is missing here... is this write safe? */
        atomicAdd(devdelta, 1.0);
    }

    /* assign the deviceMembership to object objectId */
    deviceMembership[tid] = index;
  }
}
```

Επίσης, σε περίπτωση που αλλάξει το membership ενός object τότε αυξάνεται ατομικά η τιμή της μεταβλητής devdelta.

Προχωρώντας στην kmeans_gpu συμπληρώνουμε τις αρχικοποιήσεις, τις αντιγραφές δεδομένων στη GPU. Για αρχή υπολογίζουμε τον αριθμό των blocks που θα χρησιμοποιήσουμε με σκοπό να έχουμε περίπου όσα threads όσα είναι και τα Objects, ώστε να μην υπάρχουν threads που απλώς θα περιμένουν, αλλά και να μην υπάρχουν threads που θα χειριστούν πάνω από 1 object:

```
const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize)? blockSize: numObjs;
const unsigned int numClusterBlocks = (numObjs + (numThreadsPerClusterBlock-1))/numThreadsPerClusterBlock;
```

Επομένως ο αριθμός των blocks είναι ο αριθμός των objects (συν τον αριθμό threads ανά block – 1 για λόγους padding) διά των αριθμό threads ανά block.

Κατά τη διάρκεια κάθε iteration υπολογισμού των clusters πριν καλέσουμε την find_nearest_clusters αντιγράφουμε στην αντίστοιχη δομή των clusters στη GPU τα νέα κέντρα των cluster που υπολογίστηκαν στο τέλος του προηγούμενου iteration:

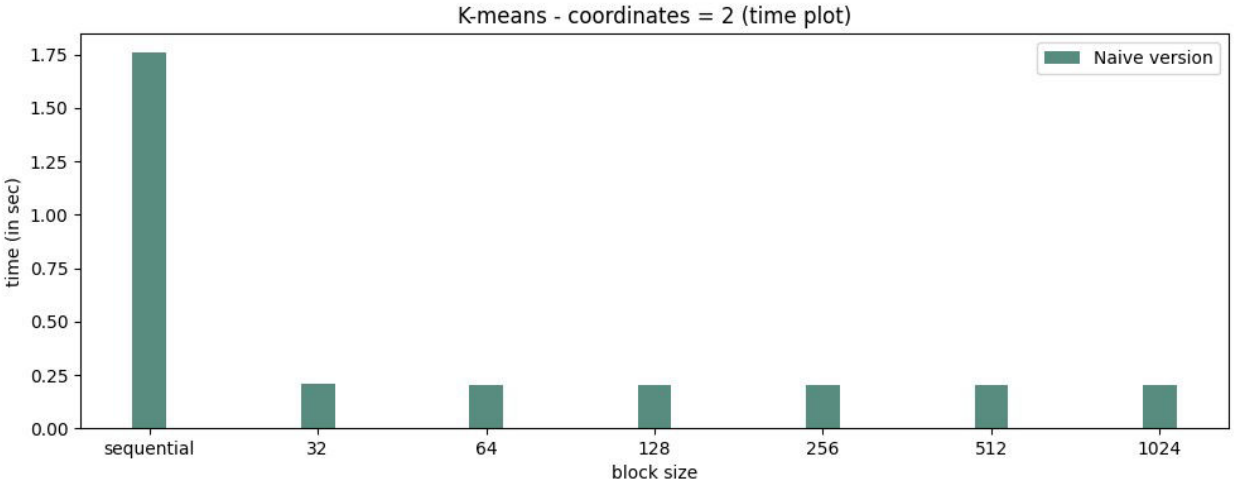
```
/* TODO: Copy clusters to deviceClusters*/
checkCuda(cudaMemcpy(deviceClusters, clusters,
          numClusters*numCoords*sizeof(float), cudaMemcpyHostToDevice));
```

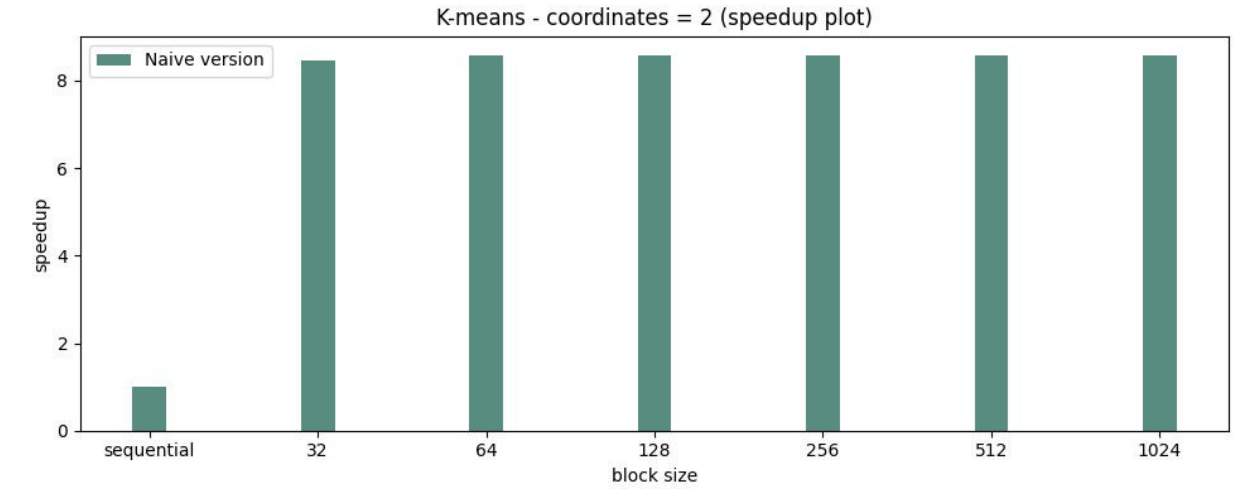
Μετά την εκτέλεση της find_nearest_cluster στην GPU (το εξασφαλίζουμε με την cudaDeviceSynchronize()) αντιγράφουμε από την GPU στον Host τον πίνακα membership και το delta για τον έλεγχο της σύγκλισης.

```
checkCuda(cudaMemcpy(membership, deviceMembership, numObjs*sizeof(int), cudaMemcpyDeviceToHost));

/* TODO: Copy dev_delta_ptr to &delta*/
checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(float), cudaMemcpyDeviceToHost));
```

Τέλος, υπολογίζονται εκ νέου τα κέντρα των clusters και συνεχίζει η εκτέλεση στο επόμενο iteration ή τερματίζει.

Αξιολόγηση Επίδοσης Naive υλοποίησης:





Βλέπουμε πως η εκτέλεση του αλγορίθμου επιταχύνεται περίπου 8.5x με την naive υλοποίηση στην GPU σε όλα τα block sizes. Όπως ήταν αναμενόμενο έχουμε σημαντικό speed-up σε σχέση με την σειριακή εκτέλεση, αφού ο υπολογισμός των nearest clusters γίνεται παράλληλα για κάθε object από ένα thread. Ωστόσο, φαίνεται πως στην naive υλοποίηση το μέγεθος του block δεν παίζει σημαντικό ρόλο στο speed-up, καθώς ο K-means είναι memory-bound και είναι περιορισμένος από το μέγεθος των δεδομένων στην μνήμη και όχι από τον αριθμό των υπολογισμών πάνω σε κάθε κομμάτι μνήμης και επομένως το μέγεθος του block δεν έχει μεγάλη σημασία. Ωστόσο, πρέπει να προσπαθούμε να πετυχαίνουμε υψηλό occupancy της GPU για να έχουμε καλά αποτελέσματα.

Transpose version

Για την transpose έκδοση κάναμε μερικές αλλαγές στον κώδικα της naive έκδοσης. Για αρχή στην euclid_dist_2, εφόσον πλέον οι πίνακες που χρησιμοποιούμε είναι ανεστραμμένοι, για να πάρουμε τις

συντεταγμένες ενός object πρέπει να ακολουθήσουμε την εξής λογική: λαμβάνουμε μία συντεταγμένη χρησιμοποιώντας το objectId και τον αριθμό της συντεταγμένης και για την επόμενη συντεταγμένη προχωράμε στον πίνακα κατά numObjs θέσεις για να πάρουμε την επόμενη συντεταγμένη:

```
    for(i=0; i<numCoords; i++){
        ans += (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]) * (objec
ts[i*numObjs + objectId] - clusters[i*numClusters + clusterId]);
    }
```

Επομένως οι αποστάσεις υπολογίζονται με τον παραπάνω τρόπο.

Δεν κάνουμε κάποια αλλαγή στην find_nearest_cluster (ίδια με την naive έκδοση).

Στην kmeans_gpu, αρχικά αντιστρέφουμε το allocation των πινάκων των objects, clusters και newclusters:

```
    /* TODO: Transpose dims */
    float  **dimObjects = (float **) calloc_2d(numCoords, numObjs, sizeof(float)); //cal
d(...) -> [numCoords][numObjs]
    float  **dimClusters = (float **) calloc_2d(numCoords, numClusters, sizeof(float));
lloc_2d(...) -> [numCoords][numClusters]
    float  **newClusters = (float **) calloc_2d(numCoords, numClusters, sizeof(float));
lloc_2d(...) -> [numCoords][numClusters]
```

Όπως βλέπουμε πλέον οι πίνακες είναι της μορφής numCoords x numObjs και στη συνέχεια η αντιγραφή των objects γίνεται με ανεστραμμένη λογική, όπου κάθε γραμμή είναι μια συντεταγμένη και κάθε στήλη είναι ένα object:

```
//  TODO: Copy objects given in [numObjs][numCoords] layout to new
//  [numCoords][numObjs] layout
for(i = 0; i < numObjs; i++){
    for(j = 0; j < numCoords; j++){
        dimObjects[j][i] = objects[i*numCoords + j];
    }
}
```

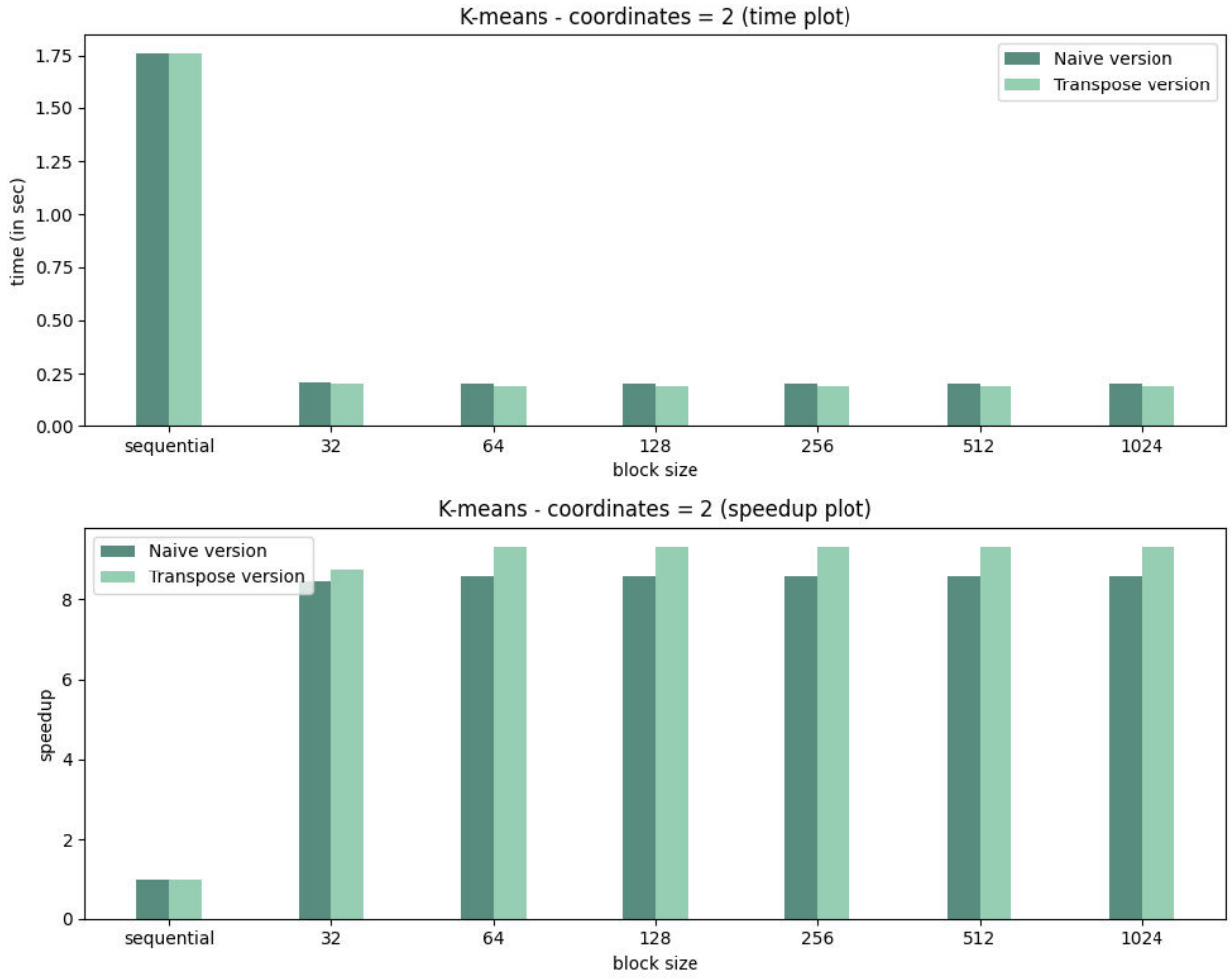
Μετά, κατά τη διάρκεια της αντιγραφής των clusters στην GPU (deviceClusters), πλέον αντιγράφουμε χρησιμοποιώντας το dimClusters[0], καθώς έχουμε δισδιάστατο πίνακα:

```
    /* TODO: Copy clusters to deviceClusters*/
    checkCuda(cudaMemcpy(deviceClusters, dimClusters[0], numClusters*numCoords*sizeof(flo
at), cudaMemcpyHostToDevice));
```

Και τέλος φροντίζουμε να αντιγράψουμε τα τελικά clusters από το dimClusters με μη ανάστροφο τρόπο για να είναι στη σωστή μορφή:

```
    /*TODO: Update clusters using dimClusters. Be carefull of layout!!! clusters[numClusters]
[numCoords] vs dimClusters[numCoords][numClusters] */
    for(i = 0; i < numCoords; i++){
        for(j = 0; j < numClusters; j++){
            clusters[j*numCoords + i] = dimClusters[i][j];
        }
    }
```

Αξιολόγηση επίδοσης:



Παρατηρούμε πως το speed up βελτιώνεται στην transpose υλοποίηση, καθώς φτάνει μέχρι και περίπου x10. Το block size αυτή τη φορά φαίνεται να παίζει ρόλο μόνο όταν έχουμε size = 32, όπου

έχουμε χειρότερη επίδοση από τα υπόλοιπα μεγέθη (x8.5). Αυτό πιθανότατα συμβαίνει, λόγω των προσβάσεων στην μνήμη, αφού με size = 32 δεν αξιοποιείται καλά το throughput.

Η διαφορά στην επίδοση παρατηρείται, λόγω των προσβάσεων σε συνεχόμενες θέσεις στην μνήμη των threads. Τα κοντινά threads υπολογίζουν κοντινά objects με αποτέλεσμα να υπάρχει locality. Στην naive έκδοση τα threads δεν κάνουν access γειτονικές θέσεις μνήμης και για αυτό το λόγο τα accesses δεν μπορούν να γίνουν coalesced (δηλαδή δεν μπορούν να batch-αριστούν σε λίγα accesses). Στη transpose version ωστόσο, κάθε thread, μία δεδομένη χρονική στιγμή, κάνει access ένα coordinate από ένα object και τα γειτονικά του thread (πχ με αυτά που ανήκουν στο ίδιο warp) θα διαβάσουν, την ίδια χρονική στιγμή, το ίδιο coordinate γειτονικών objects. Έτσι πλέον τα threads κάνουν access γειτονικές θέσεις μνήμης και τα memory accesess είναι coalesced.

Στην naive έκδοση, ένα thread θα κάνει μια πρόσβαση στην μνήμη και θα φέρει όλες τις συντεταγμένες ενός object και τα διπλανά του objects με όλες τις συντεταγμένες τους για όσο μέγεθος επιτρέπει το throughput. Ωστόσο, είναι πιο αποτελεσματικό στην transpose έκδοση τα threads να φέρνουν μόνο την συγκεκριμένη συντεταγμένη που χρειάζονται ανά iteration και μαζί να φέρουν την συγκεκριμένη συντεταγμένη περισσότερων objects την οποία χρειάζονται ταυτόχρονα τα υπόλοιπα threads του block εκείνη τη στιγμή της εκτέλεσής τους.

Shared version

Για την shared έκδοση τροποποιούμε τον κώδικα της transpose στην find_nearest_cluster και στην kmeans_gpu. Στην find_nearest_cluster αρχικοποιούμε έναν shared πίνακα για τα clusters αντιγράφοντας στην shared memory τα clusters για γρηγορότερη πρόσβαση:

```
    if(threadIdx.x < numClusters){
        int i;
        for(i = 0; i < numCoords; i++){
                shmemClusters[i*numClusters + threadIdx.x] = deviceClusters[i*numClusters + t
hreadIdx.x];
        }
    }

    __syncthreads();
```

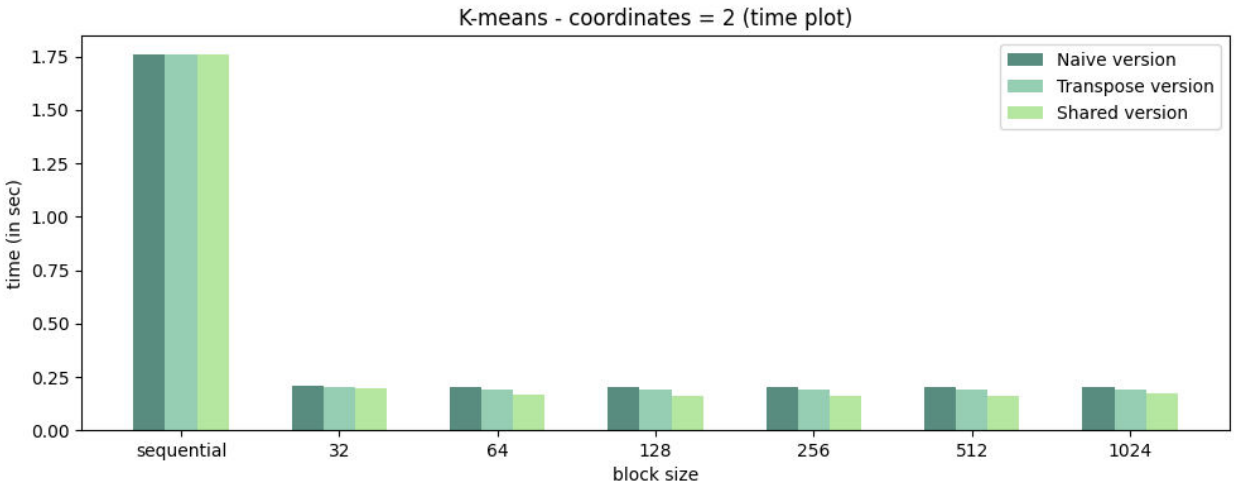
Παραλληλοποιούμε τη διαδικασία αφήνοντας numClusters threads ανά block να αντιγράψουν από ένα cluster στην shared memory και έτσι πλέον υπάρχουν όλα τα clusters στην shared memory κάθε block και μετά χρησιμοποιούμε __syncthreads() πριν αρχίσουμε τον υπολογισμό, προκειμένου να είμαστε σίγουροι πως η αντιγραφή έχει τελειώσει.

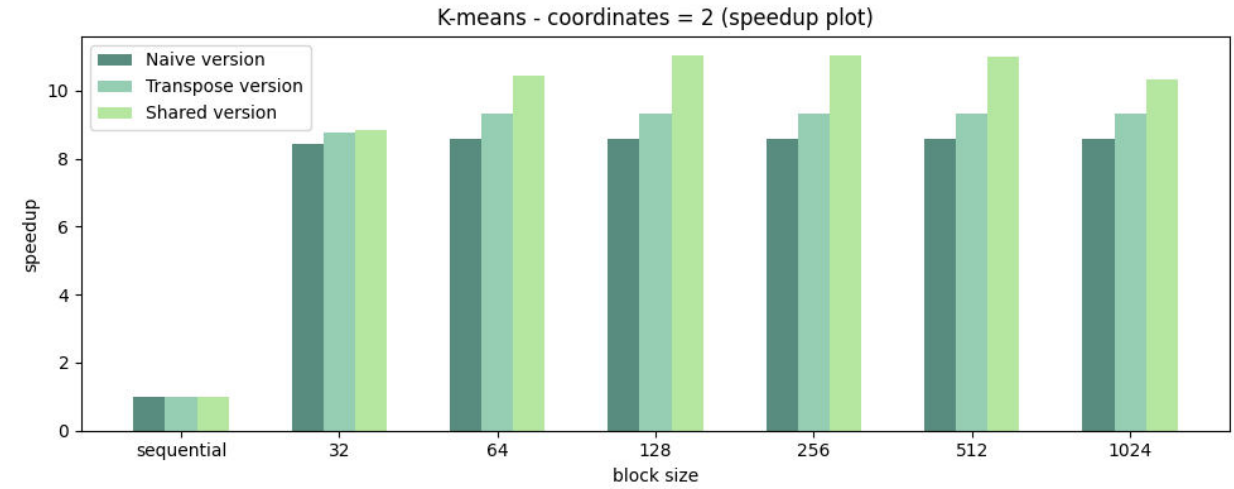
Στη συνέχεια, η διαδικασία υπολογισμού ακολουθεί τη λογική της transpose έκδοσης απλώς γίνεται πάνω στα shmemClusters.

Στην kmeans_gpu προσθέτουμε τώρα και ένα μέγεθος από Shared Data Size που θα ζητήσουμε από την GPU όταν κάνουμε fire τον kernel:

```
const unsigned int clusterBlockSharedDataSize = sizeof(float)*numClusters*numCoords;
```

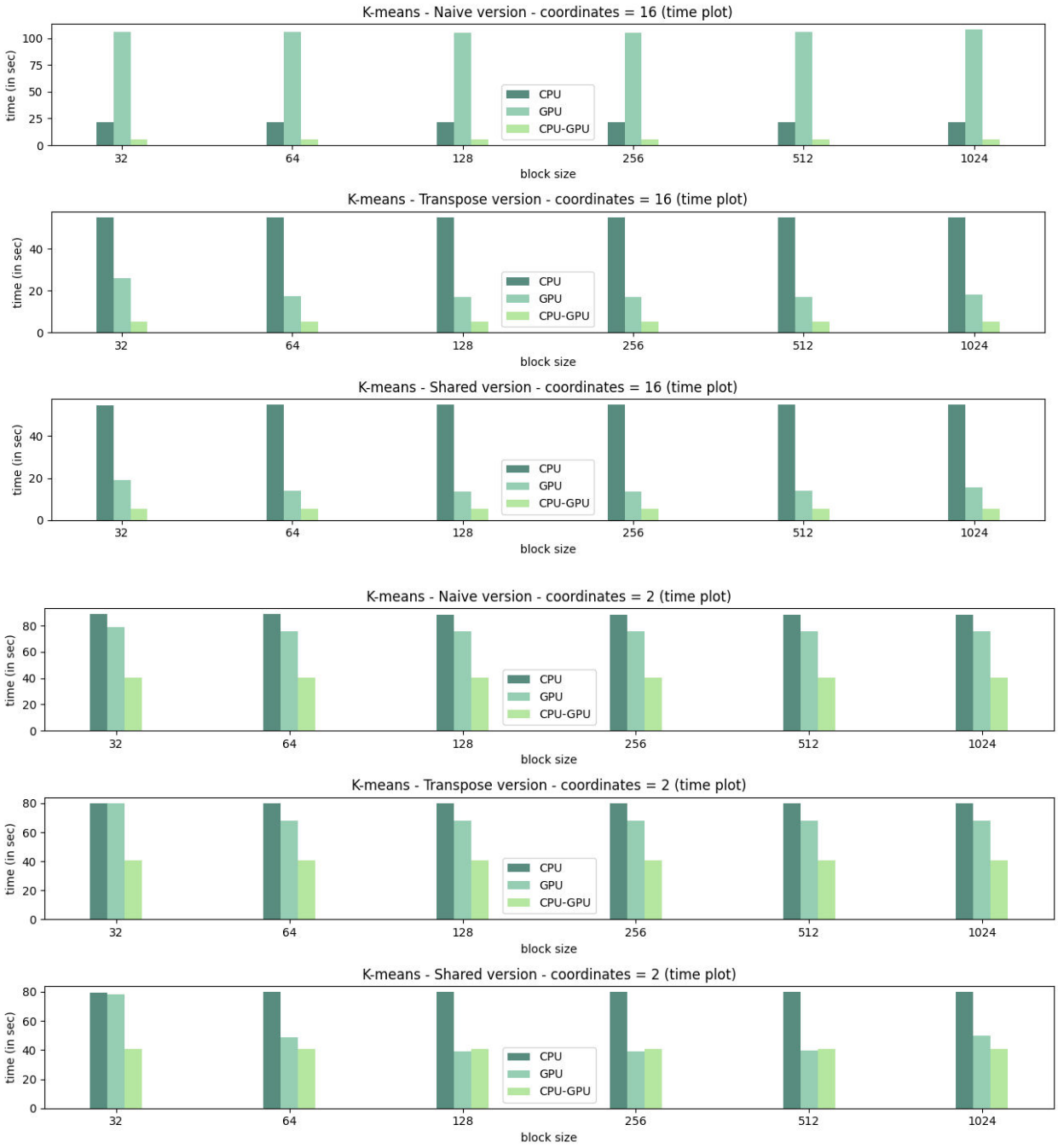
<u>Αξιολόγηση επίδοσης:</u>





Παρατηρούμε πως με τη χρήση της shared memory της GPU φτάνουμε μέχρι και x11.6 speed up, όπως βλέπουμε στα διαγράμματα. Ωστόσο, αυτή η επίδοση επιτυγχάνεται κυρίως για block sizes = 128, 256, 512, ενώ στα υπόλοιπα sizes βλέπουμε χαμηλότερη επίδοση. Στα μικρά μεγέθη (32, 64) ο αριθμός των blocks είναι μεγάλος, με αποτέλεσμα να χρειάζεται να κάνουμε allocate (δηλαδή να βρει χώρο στη shared memory η GPU) πολλές φορές τα ίδια δεδομένα στην shared μνήμη του ίδιου SM από διαφορετικά blocks (τα οποία έχουν δρομολογηθεί στον ίδιο SM) και έτσι να έχουμε υψηλό memory usage. Αντίθετα, για μέγεθος 1024 έχουμε πολλά threads που προσπαθούν να έχουν πρόσβαση στην shared μνήμη ταυτόχρονα και αυτό μπορεί να προκαλέσει conflicts για resources στην shared memory.

Σύγκριση υλοποιήσεων / bottleneck Analysis

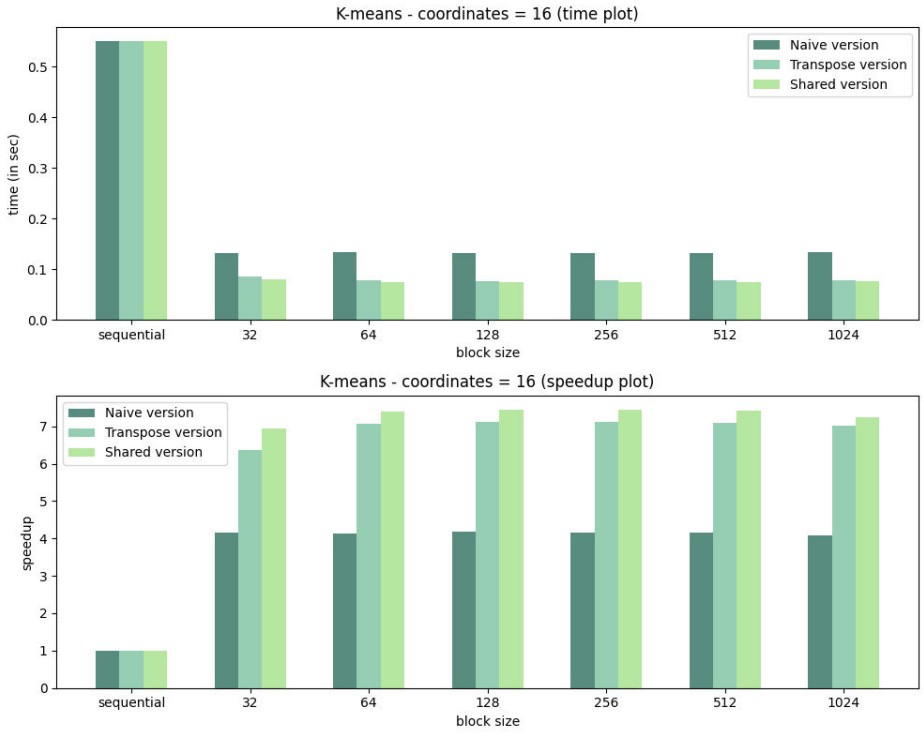


1 Ο K-means δεν είναι «καλός» αλγόριθμος για GPU γιατί είναι memory-bound και όχι computation-bound. Όπως μπορούμε να δούμε και από τα παραπάνω διαγράμματα, οι υλοποιήσεις transpose και shared καταφέρνουν να μειώσουν σημαντικά τον χρόνο εκτέλεσης στην GPU (τον χρόνο στην find_nearest_cluster) σε σχέση με την naive. Ο shared κάνει αποδοτικότερα τους υπολογισμούς στην GPU σε σχέση με τον transpose εκμεταλλευόμενος την shared μνήμη ανά block. Όμως, παραμένει σημαντικός ο χρόνος για τον υπολογισμό των νέων clusters (γίνεται στην CPU, αφού ολοκληρωθούν οι υπολογισμοί στην GPU) που είναι bottleneck. Επίσης, για πολλά objects ένα ακόμα bottleneck είναι ο χρόνος για την μεταφορά

δεδομένων μεταξύ GPU και CPU. Αυτό είναι λογικό καθώς για περισσότερα objects ο πίνακας deviceMemberhip, τον οποίο πρέπει να τον γυρίσουμε στη CPU, είναι μεγαλύτερος. Αυτό περιμένουμε να λυθεί στην All GPU υλοποίηση καθώς εκεί η μεταφορά δεδομένων αφορά μόνο τα Clusters. Ένα ακόμα bottleneck είναι ο χρόνος υπολογισμών στην GPU. Αυτός επηρεάζεται σημαντικά από τον τρόπο πρόσβασης στην μνήμη (π.χ. μπορούμε να δούμε ότι για 16 coordinates και λιγότερα objects ο χρόνος για τον K-means είναι λιγότερος από εκείνον για 2 coordinates και περισσότερα objects).

2 Το συνολικό speed up συγκριτικά με το προηγούμενο config πέφτει (μέγιστο = x7.5) και δεν είναι καλή υλοποίηση για arbitrary configurations. Στην shared υλοποίηση θέλουμε στην shared μνήμη του κάθε block να χωράνε όλα τα clusters. Αν οι διαστάσεις είναι μεγάλες δεν είναι αρκετή η μνήμη και θα χρειαστεί για κάθε coordinate να φέρουμε τα δεδομένα (συγκεκριμένο coordinate για όλα τα clusters) σε πολλαπλά batches, διαδικασία που καθυστερεί τους υπολογισμούς. Επίσης, μειώνοντας τον αριθμό των objects μειώνονται τα threads που χρησιμοποιούμε δεδομένου ότι έχουμε ένα thread να αντιστοιχεί σε κάθε object επομένως μειώνεται ο παραλληλισμός που πετυχαίνουμε. Στην δική μας περίπτωση επειδή όταν αυξάνουμε τα coordinates μειώνουμε τον αριθμό των objects του dataset, έχουμε το πρόβλημα utilization της GPU (χαμηλό occupancy) λόγω χρήσης μειωμένου πλήθους threads.

Παρακάτω παραθέτουμε και σε μορφή διαγραμμάτων τα αποτελέσματα:



Bonus: Full-Offload (All-GPU) version

Για την Full-offload έκδοση χρησιμοποιούμε δύο νέους πίνακες devicenewClusters και devicenewClusterSize. Στον πρώτο, τα thread στην GPU γράφουν ανά cluster το άθροισμα (των τιμών των συντεταγμένων) των objects που υπολογίστηκε (στην GPU) ότι ανήκουν σε αυτό. Στον 2^{ο}, το πλήθος των objects που υπολογίστηκε ότι ανήκουν σε κάθε cluster στην εκάστοτε επανάληψη.

Αρχικά αρχικοποιούμε τους παραπάνω πίνακες με τις διαστάσεις που φαίνονται παρακάτω:

```
checkCuda(cudaMalloc(&devicenewClusters, numClusters*numCoords*sizeof(float)));
checkCuda(cudaMalloc(&devicenewClusterSize, numClusters*sizeof(int)));
```

Παραθέτουμε την δομή της while παρακάτω:

```
do {
    timing_internal = wtime();
    checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(float)));
    checkCuda(cudaMemset(devicenewClusters, 0.0, numClusters * numCoords * sizeof(float)));
    checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));

    find_nearest_cluster
        <<< numClusterBlocks, numThreadsPerClusterBlock, 2*clusterBlockSharedDataSize >>>
        (numCoords, numObjs, numClusters,
         deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters, deviceMembership, dev_delta_ptr);

    cudaDeviceSynchronize(); checkLastCudaError();
    const unsigned int update_centroids_block_sz = (blockSize > numCoords*numClusters) ? blockSize : numCoords*numClusters;
    const unsigned int update_centroids_dim_sz = 1;
    update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
        (numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters, numClusterBlocks);

    checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(float), cudaMemcpyDeviceToHost));
    cudaDeviceSynchronize(); checkLastCudaError();

    delta /= numObjs;
    loop++;
    timing_internal = wtime() - timing_internal;
    if ( timing_internal < timer_min) timer_min = timing_internal;
    if ( timing_internal > timer_max) timer_max = timing_internal;

    } while (delta > threshold && loop < loop_threshold);
```

Στη συνέχεια παραθέτουμε την find_nearest_cluster όπου αφού κάθε thread υπολογίσει το πλησιέστερο cluster από το object στο οποίο αντιστοιχεί, ενημερώνει το άθροισμα των objects του cluster αυτού καθώς επίσης προσθέτει 1 στο πίνακα deviceNewClusterSize. Αυτά τα πετυχαίνει με την ατομική εντολή atomicAdd.

```
/* additional steps for calculating new centroids in GPU */
int offset = numClusters*numCoords;
for(i = 0; i < numCoords; i++){
        atomicAdd(&devicenewClusters[i*numClusters + index], deviceobjects[i*numObjs + tid]);
}

atomicAdd(&devicenewClustersSize[index], 1);
```

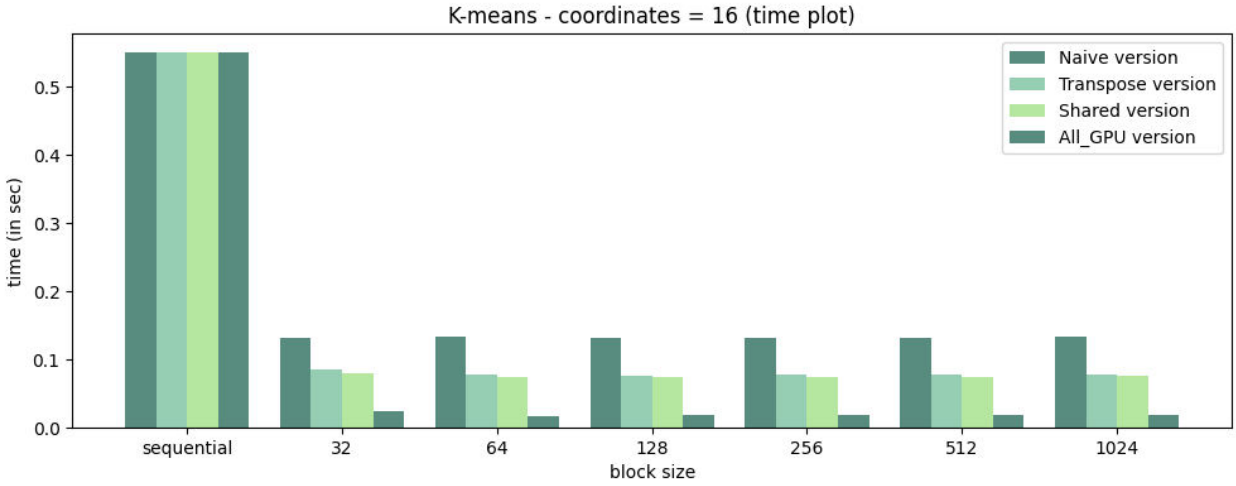
Αφού ολοκληρωθούν οι υπολογισμοί για όλα τα objects στους πίνακες devicenewClusters και devicenewClusterSize, πράγμα που εξασφαλίζουμε με την εντολή cudaDeviceSynchronize() μετά την κλήση της find_nearest_clusters καλούμε τον update_centroids kernel. Ο τελευταίος υπολογίζει τα νέα clusters χρησιμοποιώντας τα δεδομένα στους πίνακες devicenewClusters και devicenewClusterSize όπως φαίνεται στο παρακάτω κώδικα:

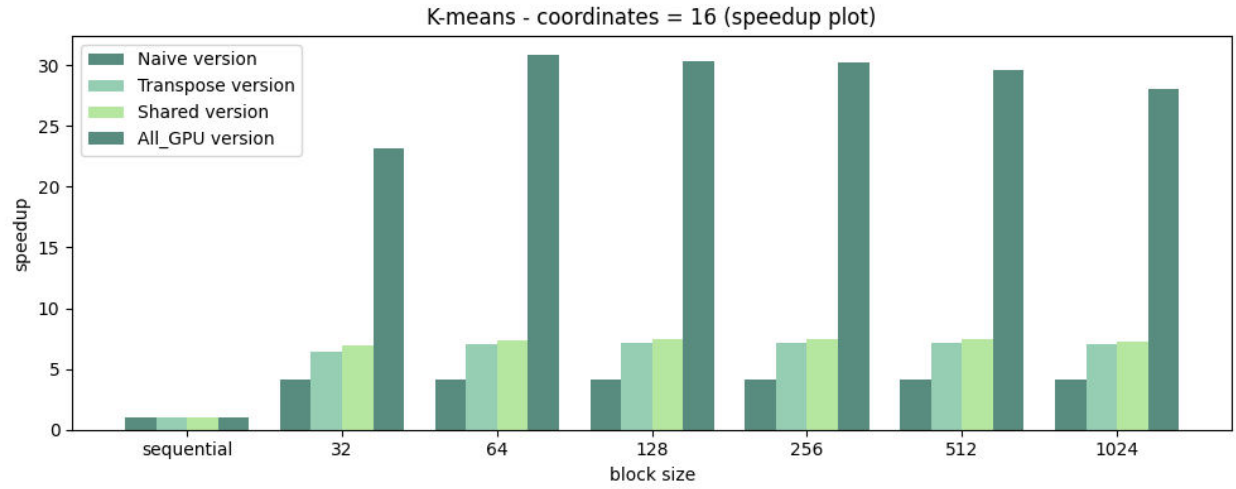
```
__global__ static
void update_centroids(int numCoords,
                      int numClusters,
                      int *devicenewClusterSize,
                      float *devicenewClusters,
                      float *deviceClusters,
                      int grid_size)
{
    int local_tid = threadIdx.x;
    int i;
    if(local_tid < numClusters){

        int temp = devicenewClusterSize[local_tid];
        for(i = 0; i < numCoords; i++){
            deviceClusters[i*numClusters + local_tid] = devicenewClusters[i*numClusters + local_tid]/temp;
        }
    }

}
```

Κάθε thread αναλαμβάνει ένα cluster.

Στην αρχή κάθε while loop επαναθέτουμε στο 0 όλες τις θέσεις των πινάκων πίνακες devicenewClusters και devicenewClusterSize ώστε να είναι σωστοί οι υπολογισμοί μας.

Παρακάτω παρουσιάζουμε και σε μορφή διαγραμμάτων τα αποτελέσματα:


K-means - coordinates = 16 (time plot)
Naive version
Transpose version
Shared version
All_GPU version
time (in sec)
0.0
0.1
0.2
0.3
0.4
0.5
sequential
32
64
128
256
512
1024
block size


K-means - coordinates = 16 (speedup plot)
Naive version
Transpose version
Shared version
All_GPU version
speedup
0
5
10
15
20
25
30
sequential
32
64
128
256
512
1024
block size

1 Παρατηρούμε ότι τη All-GPU version πετυχαίνει πολύ καλύτερους χρόνους, ειδικά για το configuration με τα 16 coordinates ({256, 16, 16, 10}). Συγκεκριμένα, στο configuration {256, 16, 16, 10} πετυχαίνουμε speedup ~ 30 ενώ στο {256, 2, 16, 10} ~ 14.

2 Όσον αφορά το block size, παρατηρούμε ότι με μικρότερο block size πετυχαίνουμε καλύτερο speedup, ειδικά για το configuration με τα 16 coordinates. Συγκεκριμένα στο configuration με τα 16 coordinates καλύτερο speedup έχουμε για block_size = 64 και με τα 2 για block_size = 32. Επειδή για μεγάλο block size έχουμε πολλά threads ανά block που θέλουν πρόσβαση στην κοινή μνήμη η επίδοση πέφτει. Για το μικρό block size (32) και τα 16 coordinates καθώς έχουμε περισσότερα δεδομένα που θέλουμε να υπάρχουν στην shared μνήμη ανά block, χρειάζεται να κάνουμε allocate (δηλαδή να βρει χώρο στη shared memory η GPU) πολλές φορές τα ίδια δεδομένα στην shared μνήμη του ίδιου SM από διαφορετικά blocks (τα οποία έχουν δρομολογηθεί στον ίδιο SM) που δεν είναι αποδοτικό.

3 Η update centroids εκτελεί μόνο μία διαίρεση για κάθε cluster. Κάθε thread αναλαμβάνει μόνο ένα cluster και δεν χρειάζεται να κάνει πολλούς υπολογισμούς πάνω στα δεδομένα επομένως δεν είναι κατάλληλη για GPUs. Ωστόσο, προσπαθήσαμε να τη παραλληλοποίησουμε με το βέλτιστο τρόπο.

Ένα αρνητικό στην υλοποίηση all-gpu είναι ότι ο πίνακας των δεδομένων για όλα τα clusters (devicenewClusters, devicenewClusterSize) ανανεώνεται ατομικά από κάθε thread.

Σε σχέση με τις προηγούμενες υλοποιήσεις, δεν μεταφέρουμε τα δεδομένα μεταξύ GPU-CPU σε κάθε iteration. Μόνο μία φορά στην αρχή (CPU -> GPU) και μία φορά στο τέλος για να πάρουμε στην CPU τα αποτελέσματα που είναι στην GPU. Επίσης, παραλληλοποιούμε ένα κομμάτι (υπολογισμό των νέων clusters) που στις προηγούμενες υλοποιήσεις εκτελούταν σειριακά στην CPU. Ακόμα και αν η παραλληλοποίηση δεν είναι πολύ καλή, συγκριτικά μειώνει περαιτέρω τον χρόνο εκτέλεσης.

4 Όπως προαναφέρθηκε στο configuration με τα 16 coordinates έχουμε πολύ καλύτερη βελτίωση σε σχέση με τις προηγούμενες υλοποιήσεις και πετυχαίνουμε τελικά το 2πλάσιο speedup σε σχέση με εκείνο του configuration με τα 2 coordinates. Στα 16 coordinates, έχουμε λιγότερα objects και επομένως συνολικά οι προσβάσεις στην μνήμη είναι λιγότερες και παράλληλα περισσότερους υπολογισμούς ανά object (ανά thread δηλαδή). Αυτό είναι καλή πρακτική για GPU υλοποιήσεις. **Επίσης, με τα λιγότερα objects έχουμε προφανώς λιγότερα AtomicAdds (λιγότερα +1 στο devicenewClusterSize) και έτσι μειώνεται ο χρόνος.**