

Συστήματα Παράλληλης Επεξεργασίας

1η Εργαστηριακή Ασκηση

Κακούρης Δημήτριος el19019

Κλήμου Ανθή el19033

Κοσμάς Θωμάς el19845

1.2 Conway's Game of Life

Προκειμένου να παραλληλοποιήσουμε το Game Of Life εντοπίζουμε τα σημεία του κώδικα τα οποία επιδέχονται παραλληλοποίηση. Το Game Of Life υπολογίζει το state του κάθε κελιού του grid σύμφωνα με τους κανόνες που δίνονται στην εκφώνηση. Κρατάμε δυο states το προηγούμενο και το τωρινό grid, το προηγούμενο αφορά τη στιγμή που απεικονίζουμε το grid ώστε να υπολογίσουμε το τωρινό, δηλαδή το τωρινό αφορά την t+1 στιγμή και το προηγούμενο την t.

Προκειμένου να κάνουμε χρήση του API της OpenMP, κάνουμε `#include <omp.h>` και μεταγλωτίζουμε κατάλληλα το πρόγραμμα με το flag `-fopenmp`.

Παραθέτουμε το `make_on_queue.sh` το οποίο χρησιμοποιούμε για να υποβάλλουμε στη συστοιχία το job που αφορά το make:

```
parlab09@scirouter:~/ex1_dimitris$ cat make_on_queue.sh
#!/bin/bash

## Give the Job a descriptive name
#PBS -N makejob

## Output and error files
#PBS -o makejob.out
#PBS -e makejob.err

## How many machines should we get?
#PBS -l nodes=1

## Start
## Run make in the src folder (modify properly)

mkdir -p ${HOME}/tmp
export TMPDIR=${HOME}/tmp

module load openmp
cd /home/parallel/parlab09/ex1_dimitris/
make

rm -r ${HOME}/tmp
```

Παραθέτουμε το run_on_queue.sh το οποίο χρησιμοποιούμε για να υποβάλλουμε στη συστοιχία το job που αφορά την εκτέλεση:

Εδώ βλέπουμε συγκεκριμένα το bash script για 64x64 grid, έχουμε φτιάξει άλλα δυο για 1024x1024 και 4096x4096.

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_omp_game

## Output and error files
#PBS -o run_omp_game_64.out
#PBS -e run_omp_game_64.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

##How long should the job run for?
#PBS -l walltime=00:05:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab09/ex1_dimitris

for threads in 1 2 4 6 8
do
    module load openmp
    export OMP_NUM_THREADS=${threads}
    ./Game_Of_Life 64 1000
done
```

Επιλέγουμε να παραλληλοποιήσουμε το πρώτο loop δηλαδή το loop που αφορά τις γραμμές του grid.

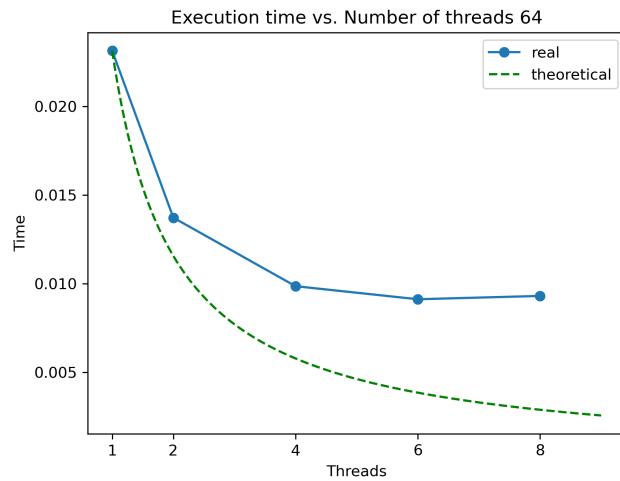
Επιλέγουμε να κάνουμε private τις μεταβλητές nbrs, i, j. Διαφορετικά θα οδηγηθούμε σε race conditions.

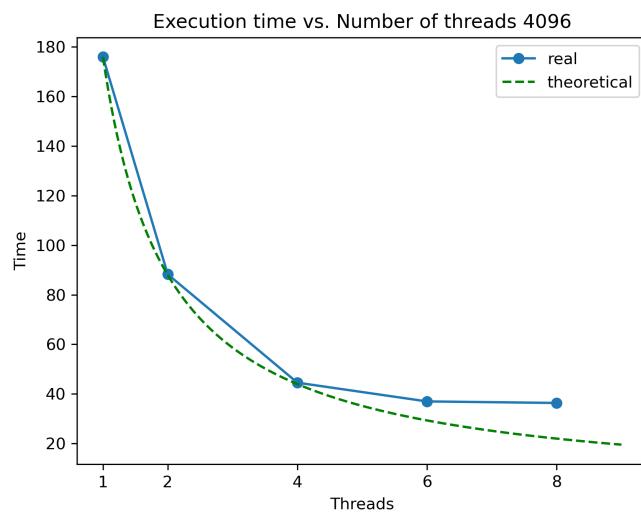
Γράφουμε επίσης ρητά ότι θέλουμε να είναι shared το previous και current state.

```
/*Game of Life*/
gettimeofday(&ts,NULL);
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for private(nbrs,i,j) shared (previous,current)
    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;
        }
}
```

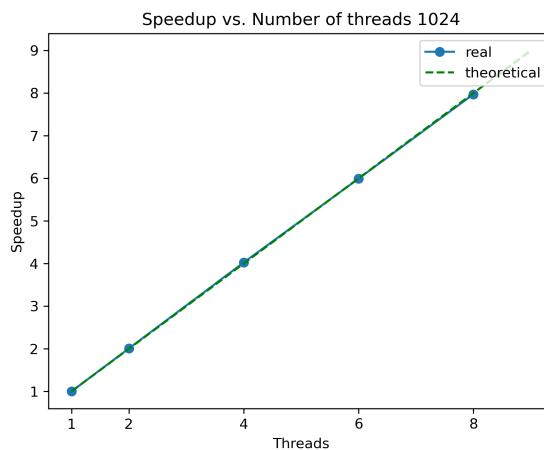
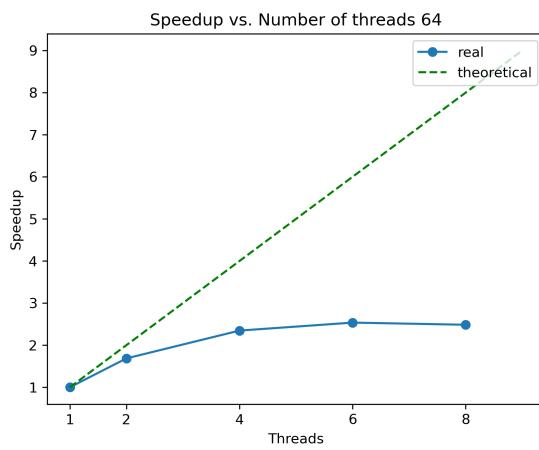
Εκτελούμε για 1000 εποχές το Game Of Life για 1,2,4,6,8 cores και μεγέθη ταμπλώ 64x64, 1024x1024 και 4096x4096.

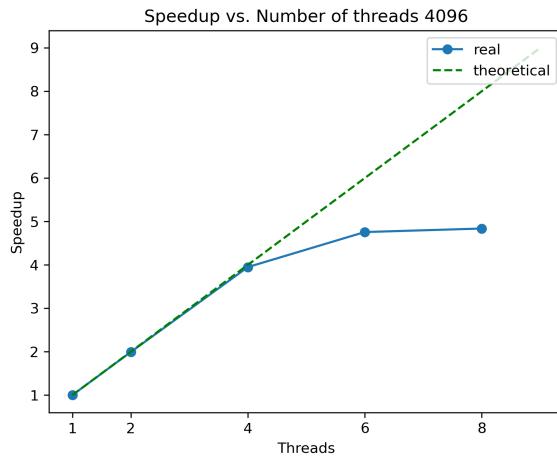
Έτσι παίρνουμε τα παρακάτω διαγράμματα χρόνου εκτέλεσης:





Αντίστοιχα τα διαγράμματα Speedup:





Παρατηρήσεις:

Στην ιδεατή περίπτωση θα περιμέναμε με αύξηση των threads, λόγω της δυνατότητας παραλληλοποίησης που μας προσφέρει η OpenMP, να οδηγεί σε αντίστοιχο speedup του χρόνου εκτέλεσης

Παρατηρούμε πώς το linear speedup το έχουμε στη περίπτωση του grid size 1024 x 1024.
Στις άλλες περιπτώσεις:

- **Περίπτωση 64 x 64:**
Στη περίπτωση αυτή η προσθήκη threads δε κλιμακώνει γραμμικά, είναι μικρό το μέγεθος του grid και το overhead που μας δίνει το communication των threads δυσχεραίνει τη κλιμάκωση.
- **Περίπτωση 4096 x 4096:**
Στη περίπτωση πάλι δεν έχουμε γραμμική κλιμάκωση, είναι μεγάλο το μέγεθος του grid η συνεχής πρόσβαση στη μνήμη δυσχεραίνει τη κλιμάκωση, λόγω μη αποδοτικής χρήσης της cache. Παρατηρούμε εδώ πιο έντονα το γεγονός ότι το Game Of Life είναι memory bound.

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Shared Clusters:

Η πρώτη υλοποίηση που καλούμαστε να συμπληρώσουμε είναι η naive υλοποίηση.

Ζητείται το configuration:

$$\{\text{Size}, \text{Coords}, \text{Clusters}, \text{Loops}\} = \{256, 16, 16, 10\}$$

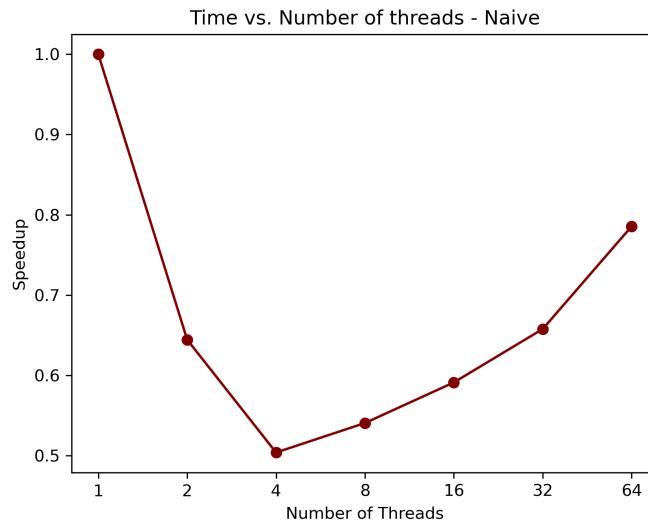
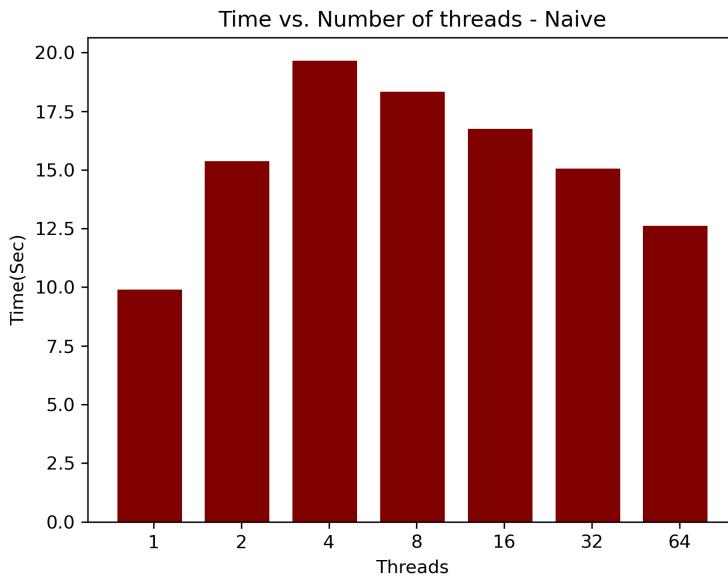
Καλούμαστε να παραλληλοποιήσουμε το πιο υπολογιστικά βαρύ κομμάτι του κώδικα που είναι η εύρεση του κοντινότερου cluster για κάθε σημείο του dataset. Ο υπολογισμός γίνεται με τη συνάρτηση `find_nearest_cluster()`.

```
89      */
90
91     #pragma omp parallel for shared(newClusters, objects, membership,clusters,newClusterSize,numCoords,delta) private(i,j,index)
92     for (i=0; i<numObjs; i++) {
93         // find the array index of nearest cluster center
94         index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
95
96         // if membership changes, increase delta by 1
97         if (membership[i] != index)
98             #pragma omp atomic
99             delta += 1.0;
100
101         // assign the membership to object i
102         membership[i] = index;
103
104         // update new cluster centers : sum of objects located within
105         /*
106         * TODO: protect update on shared "newClusterSize" array
107         */
108         #pragma omp atomic
109         newClusterSize[index]++;
110
111         for (j=0; j<numCoords; j++)
112             /*
113             * TODO: protect update on shared "newClusters" array
114             */
115             #pragma omp atomic
116             newClusters[index*numCoords + j] += objects[i*numCoords + j];
117
118     }
119
120 }
```

Στη παραπάνω υλοποίηση παραλληλοποιούμε το loop με `#pragma omp parallel for`.

Κλειδώνουμε τη με ατομική εντολή τη μεταβλητή `delta` όταν τα threads επιθυμούν να αυξήσουν τον αριθμό του `count` των αλλαγών στο `membership`, ενώ κλειδώνουμε με ατομική εντολή και το πίνακα μεγέθους των `clusters`, το `newClusterSize` και τον πίνακα που έχει τα κέντρα των `clusters`, το `newClusters`. Η ατομική εντολή προσδιορίζεται από το `#pragma omp atomic`. Παρατηρούμε στο διάγραμμα χρόνου εκτέλεσης προς τον αριθμό των νημάτων τα παρακάτω αποτελέσματα. Για τα benchmarks χρησιμοποιούμε το μηχάνημα `sandman` που μας δίνει τη δυνατότητα να χρησιμοποιήσουμε 64 threads.

Παρατηρούμε πως φαίνεται η σειριακή υλοποίηση είναι η γρηγορότερη εκ των υπολοίπων, αυτό οφείλεται στο ανταγωνισμό για το lock των atomic instructions που έχουμε στο κώδικα, συνολικά γίνονται πολλές προσβάσεις στους πίνακες `newClusters` και `newClusterSize`. Όταν κλειδώνει η πρόσβαση από ένα νήμα, κλειδώνει όλος ο πίνακας για τα άλλα νήματα οπότε έχουμε συνωστισμό. Επίσης ο χρόνος εκτέλεσης χειροτερεύει και από φαινόμενα όπως ότι τα δεδομένα τα οποία θέλει ο κάθε επεξεργαστής δεν είναι pinned στη cache αυτού.

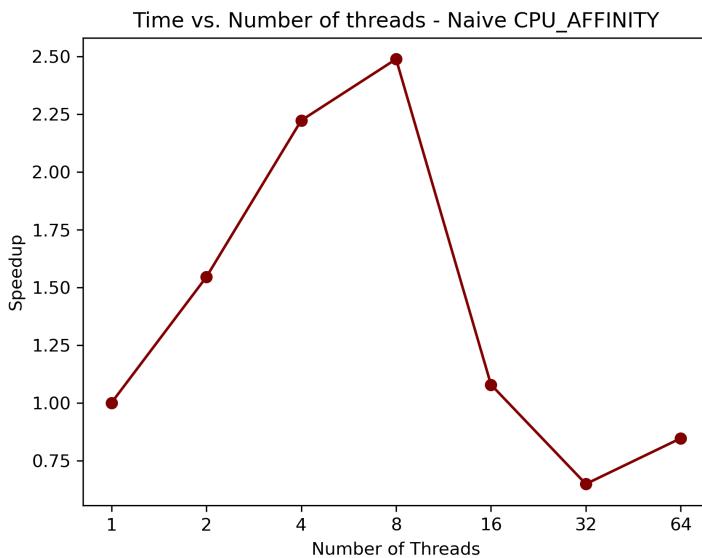
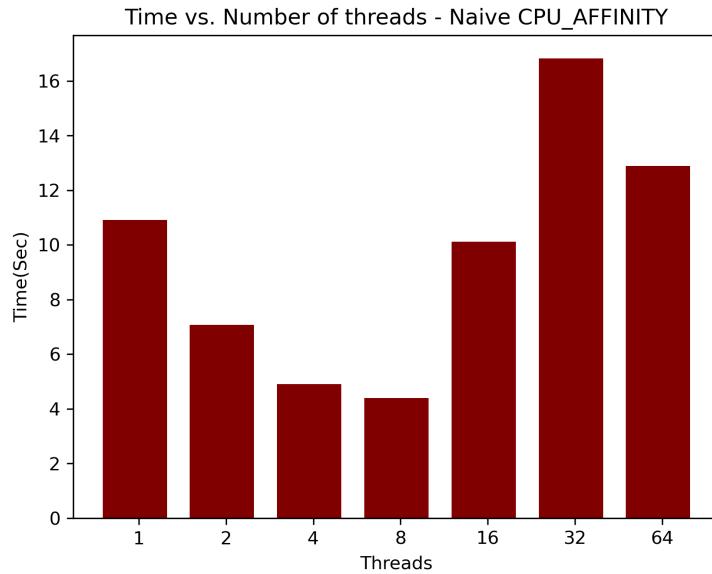


Το ζήτημα του pinning των δεδομένων το αντιμετωπίζουμε με την επιλογή GOMP_CPU_AFFINITY:

```
for i in 1 2 4 8 16 32 64
do
    export OMP_NUM_THREADS=${i}
    export GOMP_CPU_AFFINITY="0-63"
```

Έτσι προσδένουμε τα δεδομένα του κάθε επεξεργαστή στη cache του, βελτιώνοντας το locality των δεδομένων. Συγκεκριμένα στους επεξεργαστές 0 έως 31. Παρατηρούμε όπως είναι αναμενόμενο καλύτερους χρόνους και βελτίωση σε χρόνο στις παράλληλες

υλοποιήσεις σε σχέση με τη σειριακή. Χάνονται αυτά τα πλεονεκτήματα από 16 threads και μετά λόγω αυξημένου contention. Καλύτερη επίδοση παρατηρούμε στα 8 threads.



Copied and Reduced Clusters:

Ορίζουμε τα τοπικά clusters που θα χρησιμοποιηθούν για να μαζέψουν τα επιμέρους αριθμητικά αποτελέσματα για την ανανέωση των “γενικών” clusters, όπως φαίνεται στο

πρώτο TODO. Στη συνέχεια υλοποιούμε τη παραλληλοποίηση της εύρεσης των κοντινότερων clusters.

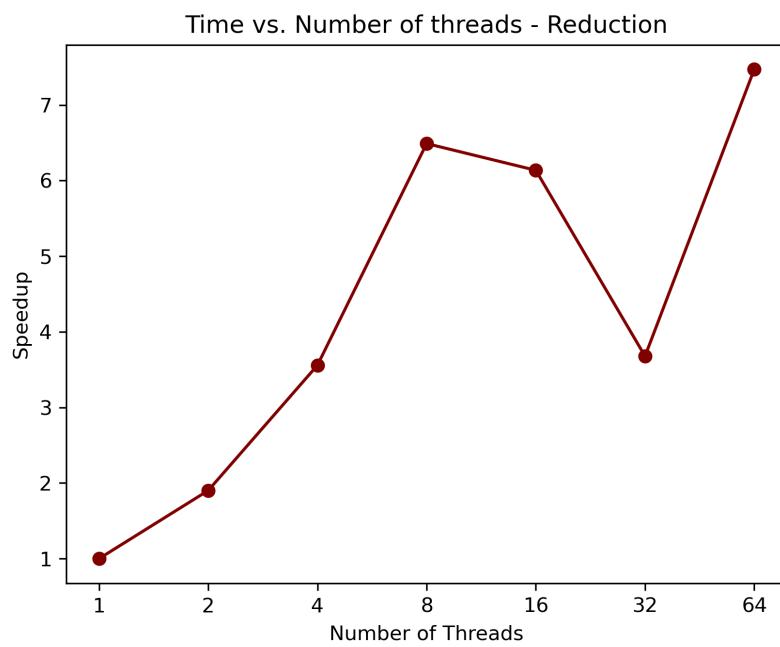
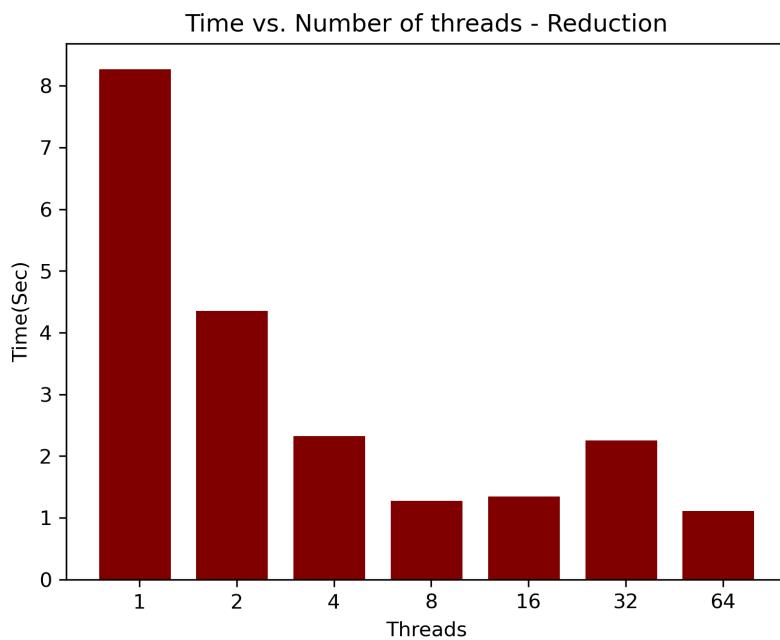
```
04  /*
05  * TODO: Initialize local cluster data to zero (separate for each thread)
06  */
07  for(k=0; k<nthreads; k++){
08      for (i=0; i<numClusters; i++) {
09          for (j=0; j<numCoords; j++)
10              local_newClusters[k][i*numCoords + j] = 0.0;
11          local_newClusterSize[k][i] = 0;
12      }
13  }
14 #pragma omp parallel shared(local_newClusters, objects,delta,membership,clusters,local_newClusterSize,numCoords) private(i,j,index)
15 {
16 #pragma omp for
17 for (i=0; i<numObjs; i++)
18 {
19     // find the array index of nearest cluster center
20     index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
21
22     // if membership changes, increase delta by 1
23     if (membership[i] != index)
24         #pragma omp atomic
25         delta += 1.0;
26
27     // assign the membership to object i
28     membership[i] = index;

```

Αντι να έχουμε ένα global array για clusters όπως στη naive υλοποίηση και να δημιουργείται ένα bottleneck λόγω των προσβάσεων που επιθυμούν να πραγματοποιήσουν τα threads και να ανανέωσουν τόσο το newClusters όσο και το newClusterSize καθώς τότε κλειδώνεται όλος ο πίνακας τώρα το κάθε thread γράφει σε δικά του τοπικά arrays και το master thread στο τέλος μαζεύει αποτελέσματα με #pragma omp master. Έτσι προκύπτει και η υλοποίηση του reduction.

```
29
30     // update new cluster centers : sum of all objects located within (average will be performed later)
31     /*
32     * TODO: Collect cluster data in local arrays (local to each thread)
33     *       Replace global arrays with local per-thread
34     */
35     int tid = omp_get_thread_num();
36     local_newClusterSize[tid][index]++;
37     for (j=0; j<numCoords; j++)
38         local_newClusters[tid][index*numCoords + j] += objects[i*numCoords + j];
39
40 }
41
42 /*
43 * TODO: Reduction of cluster data from local arrays to shared.
44 *       This operation will be performed by one thread
45 */
46 #pragma omp barrier
47 #pragma omp master
48 {
49     for (k=1; k<nthreads; k++)
50     {
51         for (i=0; i<numClusters; i++)
52         {
53             for (j=0; j<numCoords; j++)
54                 local_newClusters[0][i*numCoords + j] += local_newClusters[k][i*numCoords + j];
55         }
56     }
57     local_newClusterSize[0][i] += local_newClusterSize[k][i];
58 }
59
60 }
```

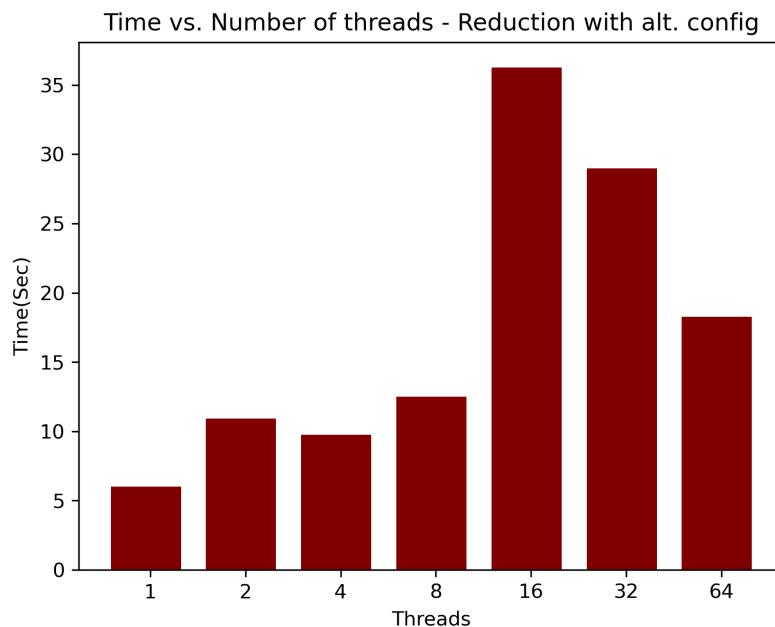
Στο reduction implementation έχουμε σχεδόν γραμμικό speedup μεχρι και 8 threads, απο 16 και μετα δεν παρατηρείται βελτίωση στο χρόνο (ακόμα υπάρχει και χειροτέρευση για 64 threads).

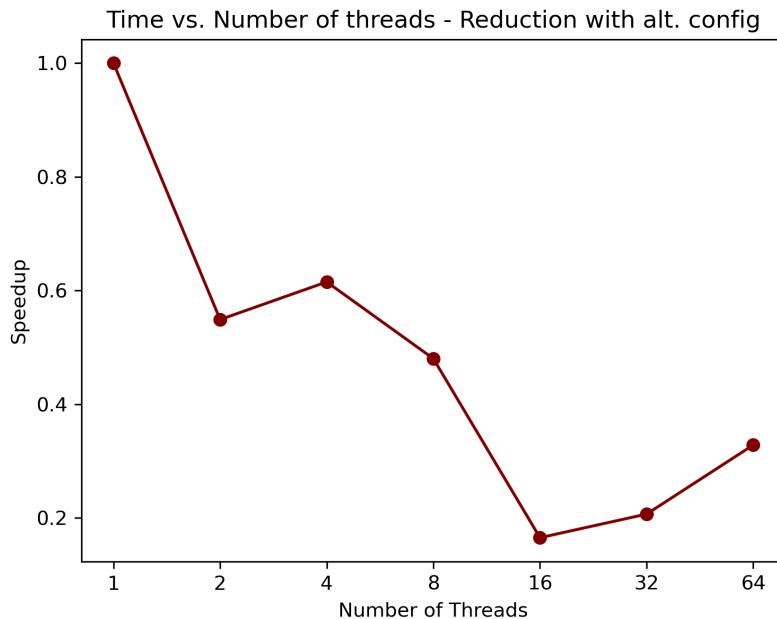


Ζητείται να συγκρίνουμε και με μια εναλλακτική αρχικοποίηση:

$$\{\text{Size, Coords, Clusters, Loops}\} = \{256, 1, 4, 10\}$$

Παρατηρούμε στη παραμετροποίηση αυτή πως ο καλύτερος χρόνος προκύπτει από τη σειριακή υλοποίηση. Διαπιστώνουμε ότι οι χρόνοι εκτέλεσης και η κλιμάκωση δεν είναι ικανοποιητική. Αυτό οφείλεται στον μικρό αριθμό των clusters, ο οποίος οδηγεί στο να βρίσκονται περισσότερες γραμμές των πινάκων local_newClusters και local_newClusterSize στην ίδια περιοχή της cache μνήμης. Αποτέλεσμα αυτού είναι το φαινόμενο του False Sharing, κατά το οποίο η εγγραφή δεδομένων από ένα νήμα συμπίπτει χωρικά με αυτές των άλλων νημάτων, καθώς γράφονται στην ίδια cache line. Σε επόμενη εγγραφή τα νήματα πρέπει να κάνουν invalidate τη cache line που θέλουν να γράψουν και ύστερα να φορτώσουν εκ νέου τα δικά τους δεδομένα. Όλο αυτό δημιουργεί αργοπορία λόγω της μετακίνησης δεδομένων από κύρια μνήμη και ανάποδα, που χειροτερεύει με τον αριθμό των νημάτων. Ο καλύτερος χρόνος που πετύχαμε είναι 5.79 sec στη σειριακή υλοποίηση.





Για να αποφύγουμε αυτό το φαινόμενο, θα προσθέσουμε μηδενικά στο τέλος κάθε τμήματος των πινάκων που αντιστοιχεί σε κάποιο thread (padding), ώστε να συμπληρωθεί το επιθυμητό μήκος cache line. Έτσι, αποφεύγουμε να επεξεργάζονται το ίδιο cache line δύο ή περισσότερα threads. Αυτό θα γίνει κατά το allocation:

Η πολιτική "first touch" στα συστήματα NUMA ορίζει ότι η μνήμη που ανατίθεται σε ένα νήμα προέρχεται από την περιοχή μνήμης που είναι πιο κοντά στον επεξεργαστή που εκτελεί αυτό νήμα για πρώτη φορά. Αυτό βελτιστοποιεί την απόδοση, καθώς ελαχιστοποιείται η καθυστέρηση πρόσβασης στη μνήμη. Έχουμε θέσει όπως είχε ζητηθεί `GOMP_CPU_AFFINITY='0-63'` για τα ερωτήματα του reduction και ετσι τα νήματα προσδένονται στις CPU που πρωτοέφεραν τα δεδομένα τους. Τώρα όλα τα νήματα που θα χρειάζονται το δεδομένο αυτό δε θα το πηγαινοφερνούν μεταξύ της RAM των nodes που εκείνα βρίσκονται και αυτής της αρχικής ανάθεσης.

Το μεγαλύτερο ίσως optimization είναι η παραλληλοποίηση του allocation των `local_newClusters` και `local_newClusterSize`, ετσι το κάθε νήμα κάνει `allocate blocks` με private τρόπο, επομένως μειώνεται αρκετά η πιθανότητα να πέσουν blocks από δύο ξεχωριστά νήματα στην ίδια Cache Line. Τα παραπάνω τα υλοποιούμε με `#pragma omp parallel for private (k)`.

```

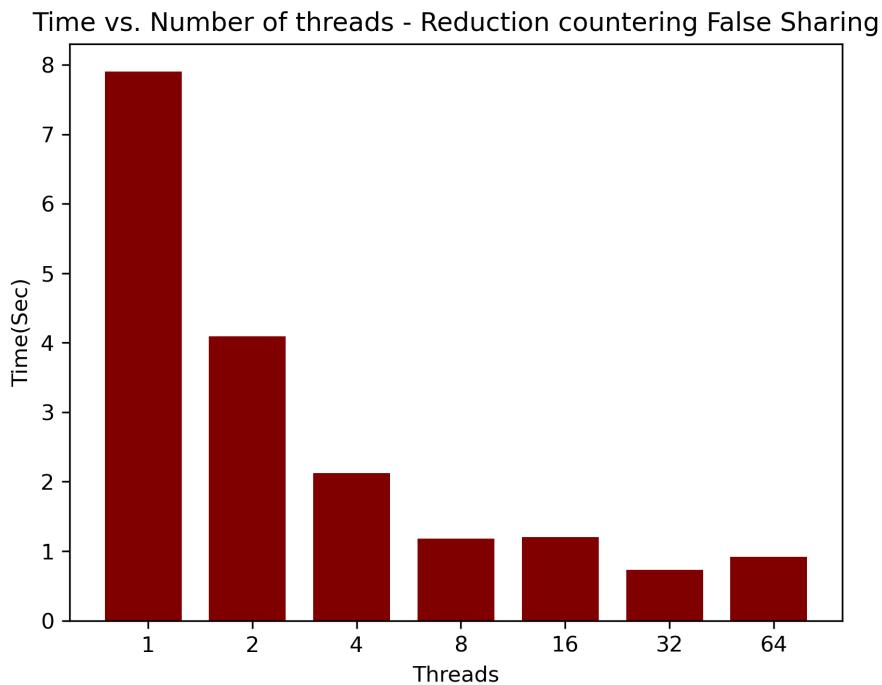
#define CACHE_LINE_SIZE 64
#define PADDING_INT ((numClusters/(CACHE_LINE_SIZE/sizeof(int))+1)*sizeof(int) - numClusters)
#define PADDING_DOUBLE (((numClusters*numCoords)/(CACHE_LINE_SIZE/sizeof(double))+1)*sizeof(double) - numClusters*numCoords)

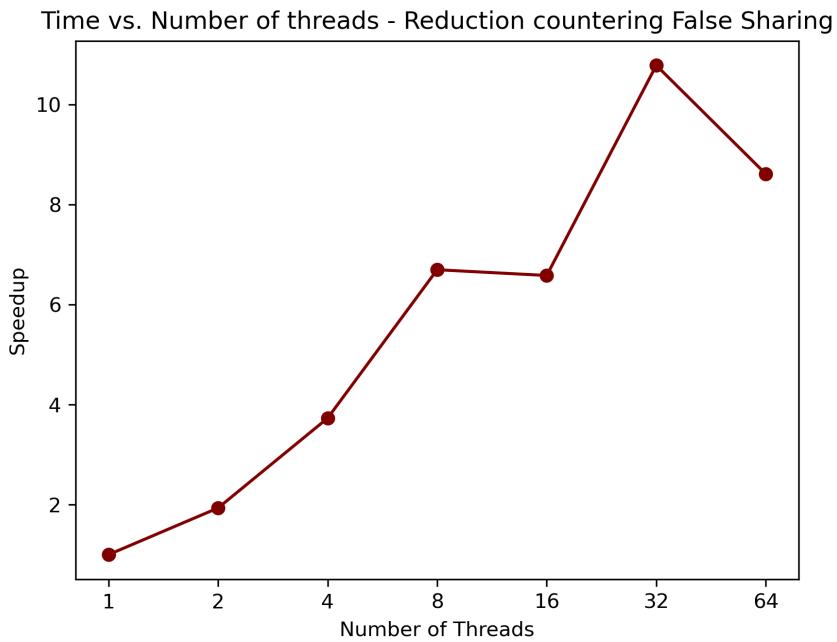
// Each thread calculates new centers using a private space. After that, thread 0 does an array reduction on them.
int * local_newClusterSize[nthreads]; // [nthreads][numClusters]
double * local_newClusters[nthreads]; // [nthreads][numClusters][numCoords]

/*
 * Hint for false-sharing
 * This is noticed when numCoords is low (and neighboring local_newClusters exist close to each other).
 * Allocate local cluster data with a "first-touch" policy.
 */
// Initialize local (per-thread) arrays (and later collect result on global arrays)
#pragma omp parallel for private (k)
for (k=0; k<nthreads; k++)
{
    local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters+PADDING_INT, sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters*numCoords+PADDING_DOUBLE, sizeof(**local_newClusters));
}

```

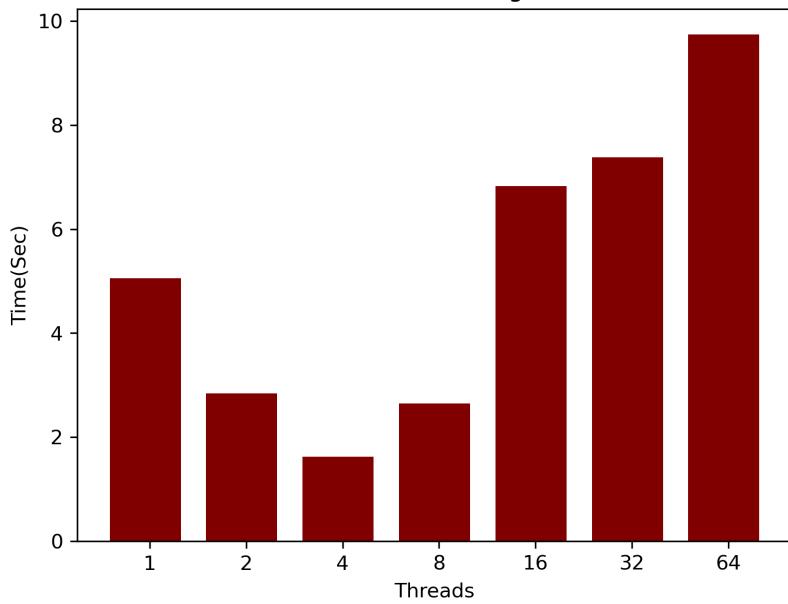
Παρατηρούμε πως και στις δύο περιπτώσεις έχουμε αισθητά βελτιωμένο χρόνο εκτέλεσης:



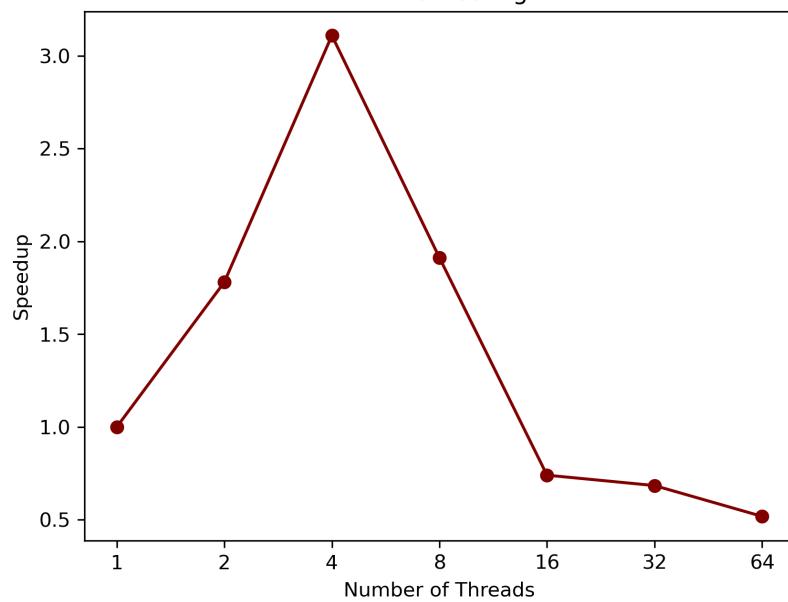


Ειδικά στη περίπτωση του alternative configuration επειδή το μέγεθος της cache line είναι 64 bytes, συμπεραίνουμε πως ο πίνακας των localClusters για ένα νήμα χωράει ακριβώς στη μισή cache line. Με συνέπεια το ακριβώς διπλανό νήμα να φέρνει το δικό του localClusters στην ίδια γραμμή κάνοντας συνεχώς invalidate τα άλλα αντίγραφα των cache lines και ξαναφορτώνοντας με τα δικά του δεδομένα. Με το padding όμως αναγκάζουμε κάθε επόμενο localClusters να πάει στην από κάτω cache line και η διαφορά είναι εμφανής.

Time vs. Number of threads - Reduction countering False Sharing
with alt.config



Time vs. Number of threads - Reduction countering False Sharing
with alt.config



2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

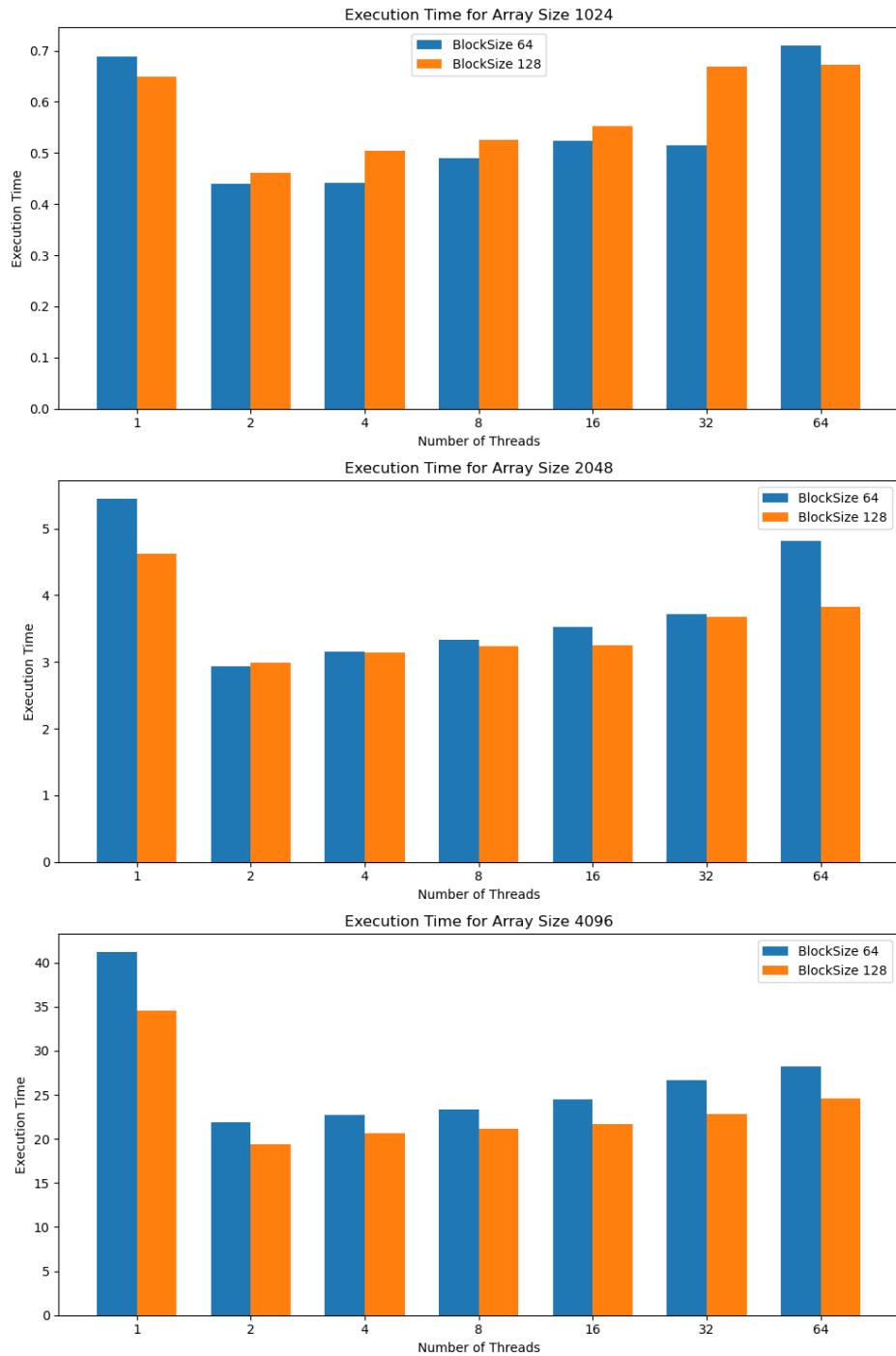
Σε αυτό το ερώτημα παραλληλοποιούμε τον αλγόριθμο fw_sr.c και πραγματοποιήσουμε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχάνημα sandman.

```
if(myN<=bsize)
    for(k=0; k<myN; k++)
        for(i=0; i<myN; i++)
            for(j=0; j<myN; j++)
                A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
else {
    FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
    #pragma omp parallel
    {
        #pragma omp single
        {
            #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 parallel
    {
        #pragma omp single
        {
            #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);
}
```

Παραλληλοποιούμε τον κώδικα με χρήση task. Βλέπουμε πως λόγω των εξαρτήσεων μπορούμε να παραλληλοποιήσουμε μόνο τις {2,3} και {6,7} αναδρομικές κλήσεις της συνάρτησης FW_SR.

Δοκιμάζουμε και μετράμε χρόνο εκτέλεσης για N = {1024,2048,4096} και B = {64,128} με Threads= {1,2,4,8,16,32,64}

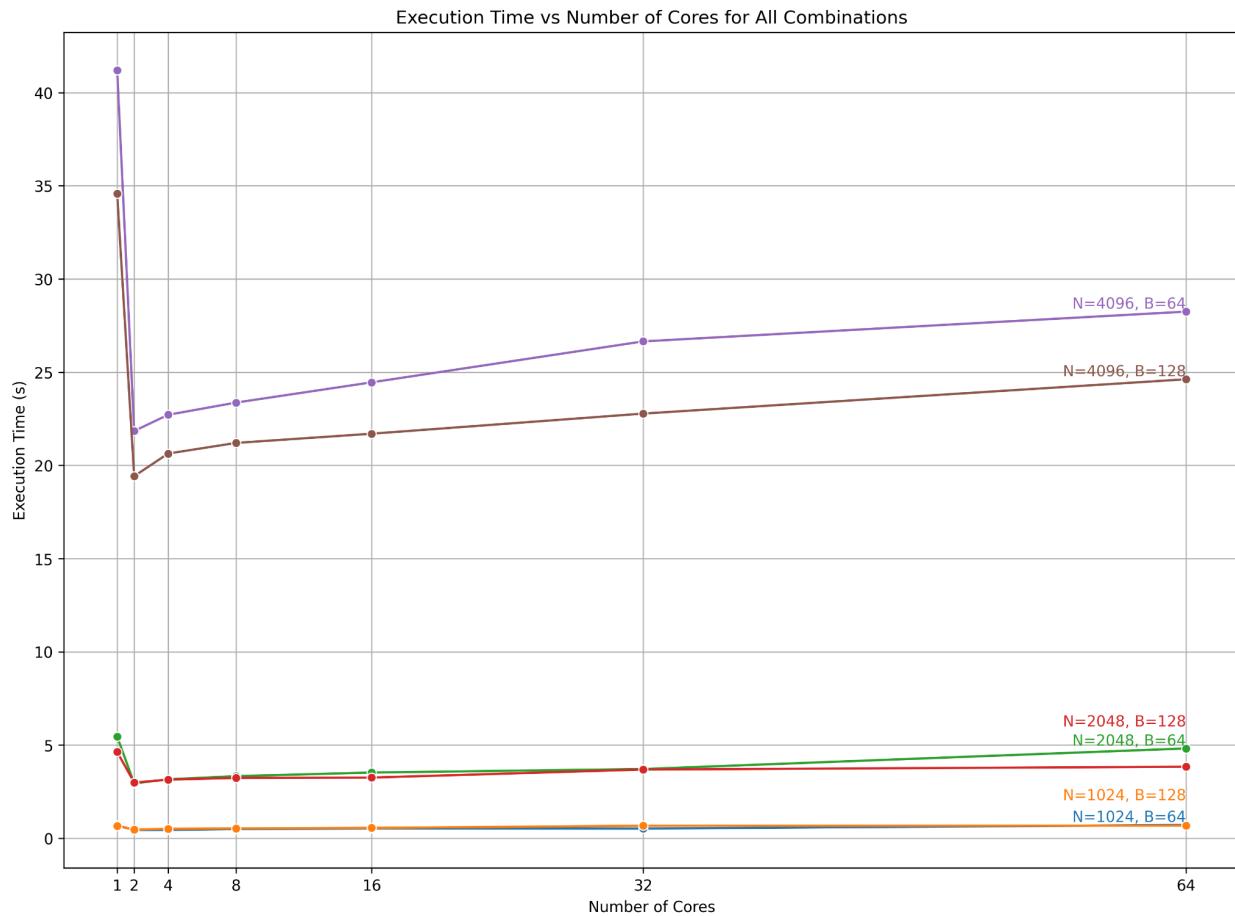
Παρακάτω λαμβάνουμε τα bar plots για καθε array configuration:



Για 64 threads έχουμε εμφανώς χειρότερη απόδοση ως προς τη κλιμάκωση μέχρι και τα 32 threads, αυτό συμβαίνει διότι λόγω του hyperthreading, το hyperthreading επιχειρεί να έχει δύο

pipelines στον ίδιο πυρήνα και να εκμεταλλευτεί το ότι οι δύο αυτές διεργασίες δε θα θελουν ταυτόχρονα τους ίδιους πόρους τη CPU και έτσι μπορούμε να εκμεταλλευτούμε τους νεκρούς αυτούς χρόνους(π.χ cache misses). Στη προκειμένη όμως έχουμε threads στον ίδιο πυρήνα να δουλεύουν στο ίδιο block και επομένως το overhead μειώνει κατά πολύ την απόδοση.

Αναπαριστούμε επίσης σε ένα συνολικό plot τα execution times:

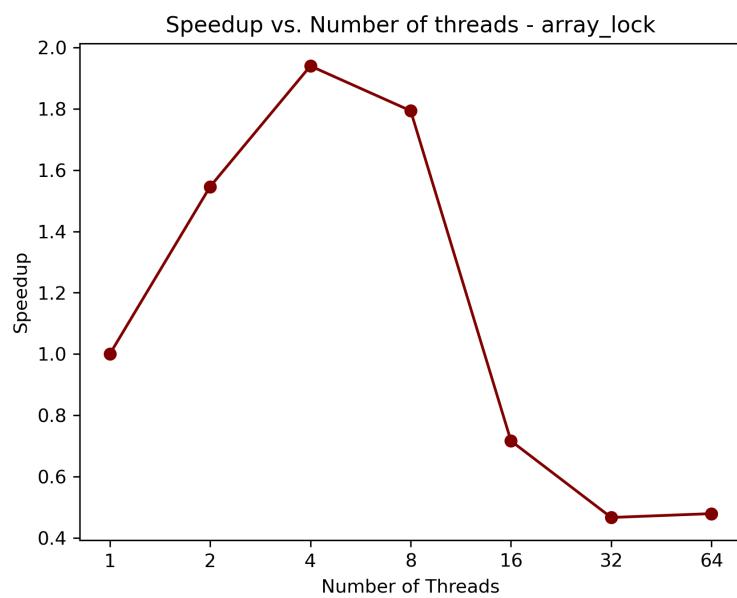
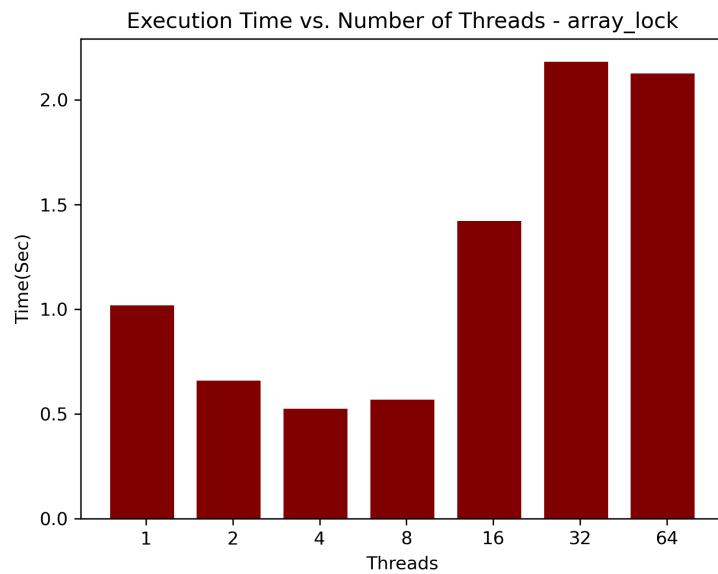


Παρατηρούμε πως βέλτιστη επίδοση έχουμε πάντα στους δύο πυρήνες ανεξαρτήτως των παραμέτρων N και B . Αυτό οφείλεται στο ότι μπορούμε να παραλληλοποιήσουμε μεχρι δύο αναδρομικές κλήσεις την φορά, άρα παραπάνω πυρήνες δεν θα δώσουν περαιτέρω επιτόχυνση καθώς δεν υπάρχουν tasks να τους αξιοποιήσουν.

Επίσης παρατηρούμε πως στα μικρά N το B δεν έχει σημαντικό ρόλο σε αντίθεση με το $N = 4096$ όπου το $B=128$ είναι εμφανώς καλύτερο.

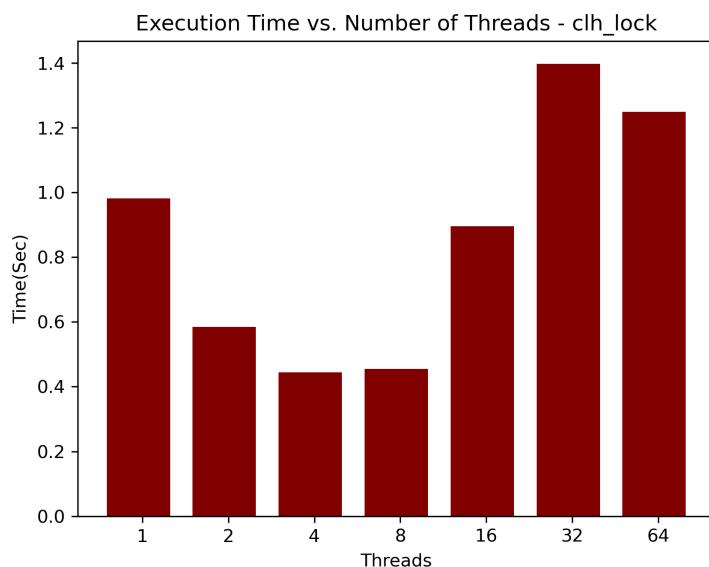
3.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means - Locks

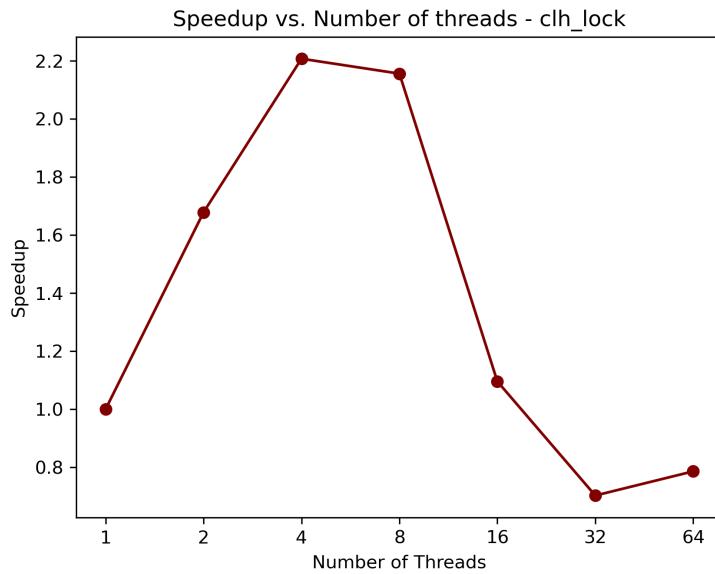
- Array Lock:



Με το array lock δημιουργείται μια λίστα με μέγεθος ίσο με τον αριθμό των threads. Στο τέλος της λίστας υπάρχει ένας counter, ο οποίος αρχικά έχει τιμή μηδέν και ορίζει το τέλος της λίστας. Επιπλέον, το στοιχείο lock[0] αρχικά έχει την τιμή true, ώστε να επιτρέπεται στο πρώτο thread που θα προσπαθήσει να εκτελέσει να τα καταφέρει. Κάθε φορά που ένα thread θέλει να εκτελέσει, λαμβάνει την τιμή του μετρητή και τον αυξάνει κατά ένα. Αναμένει μέχρι το αντίστοιχο στοιχείο του στον πίνακα lock (έστω i) να γίνει lock[i] = true. Όταν αυτό συμβεί, εκτελεί το κρίσιμο τμήμα του κώδικα. Μετά την ολοκλήρωση του κρίσιμου τμήματος, θέτει το lock[i] = false και το lock[(i+1)%size] = true, επιτρέποντας έτσι στην επόμενη διαδικασία να προχωρήσει. Έτσι μειώνεται το contention στο bus γιατί κάθε thread διαβάζει μία ξεχωριστή διεύθυνση μνήμης. Αναμενόμενο λοιπόν είναι και το αποτέλεσμα του χρόνου εκτέλεσης που παρατηρούμε. Από τα διαγράμματα παρατηρούμε πως το array lock δίνει αθροιστικά λιγότερο χρόνο. Μετά τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ από το contention για το lock).

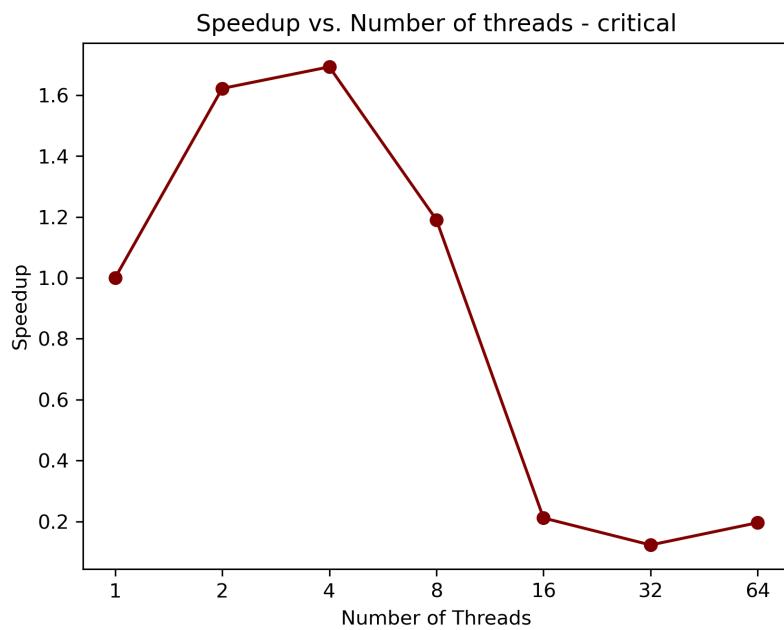
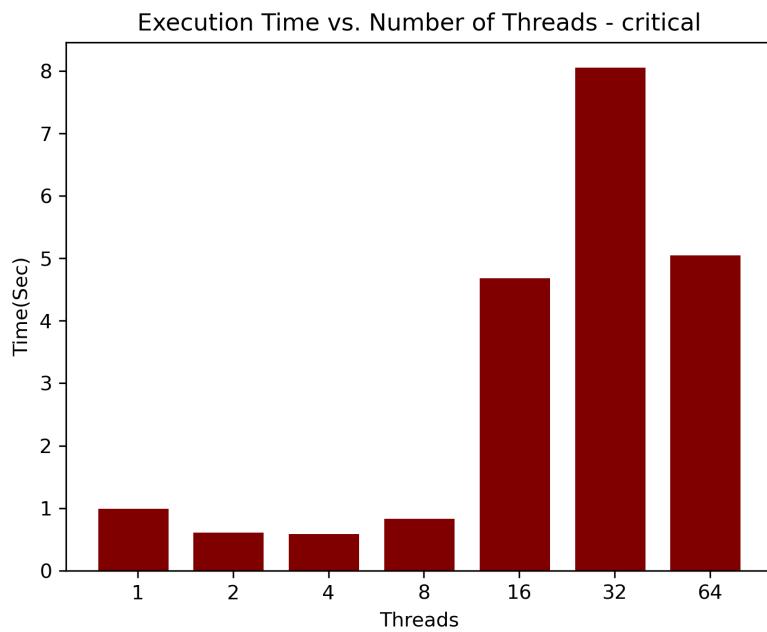
- CLH Lock:





Το clh lock υλοποιεί μια συνδεδεμένη λίστα στην οποία κάθε νήμα που περιμένει να μπεί στο κρίσιμο τμήμα δημιουργεί ένα νέο κόμβο στο τέλος της λίστας και θέτει το flag του σε false (δείχνοντας πως θέλει να πάρει το lock). Έτσι κάθε νήμα κάνει spin το flag του προγόνου του. Το clh lock είναι αποδοτικότερο κλείδωμα ιδιαίτερα για critical sections που δε διαρκούν μεγάλο χρονικό διάστημα διότι είναι τύπου spin lock και σε αυτές τις περιπτώσεις είναι αποδοτικότερο να κάνει spin σε μία μεταβλητή από το να βάζουμε σε sleep τα νήματα και να ξανα ρυπνάμε. Το clh lock μας δίνει τους καλύτερους χρόνους εκτέλεσης, λόγω του spin σε δικιά του διεύθυνση για κάθε νήμα και της μείωσης των Cache Invalidations καθώς γίνεται invalidate μόνο η cache line του επόμενου στη σειράς νήματος πετυχαίνουμε τους καλύτερους χρόνους.

- Critical:

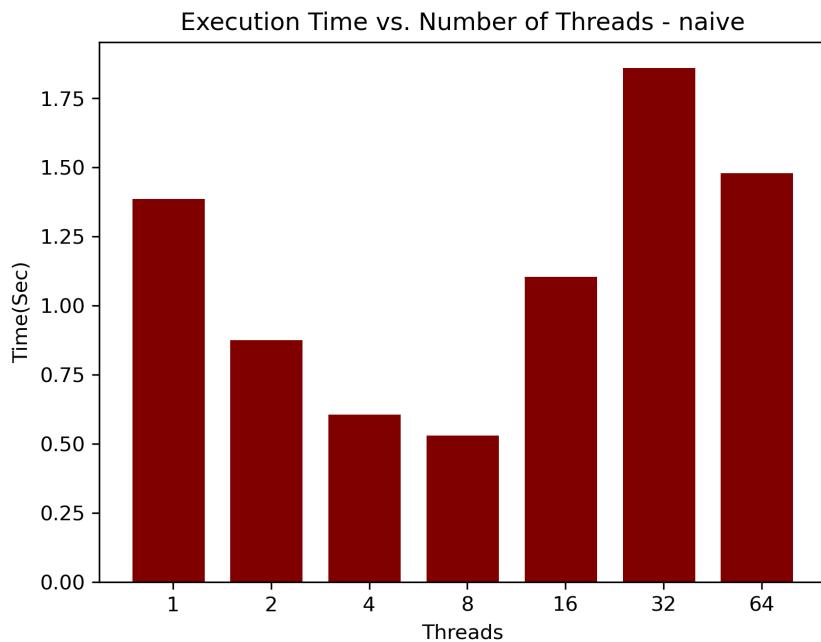


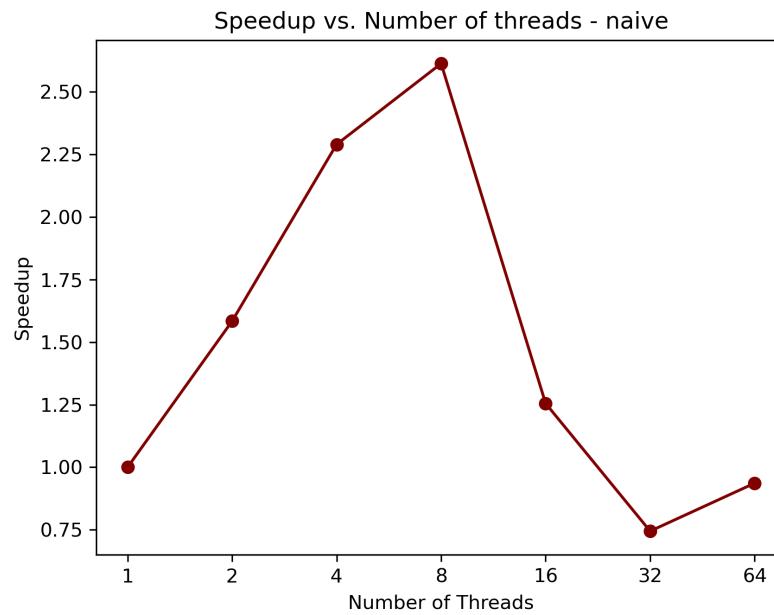
Στην `critical` υλοποίηση ο συγχρονισμός γίνεται από το API της OpenMP, συγκεκριμένα παρεμβάλει πριν και μετά από το `critical` section την εξής “εντολή”:

```
#pragma omp critical
```

```
#pragma omp critical
{
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++){
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }
}
```

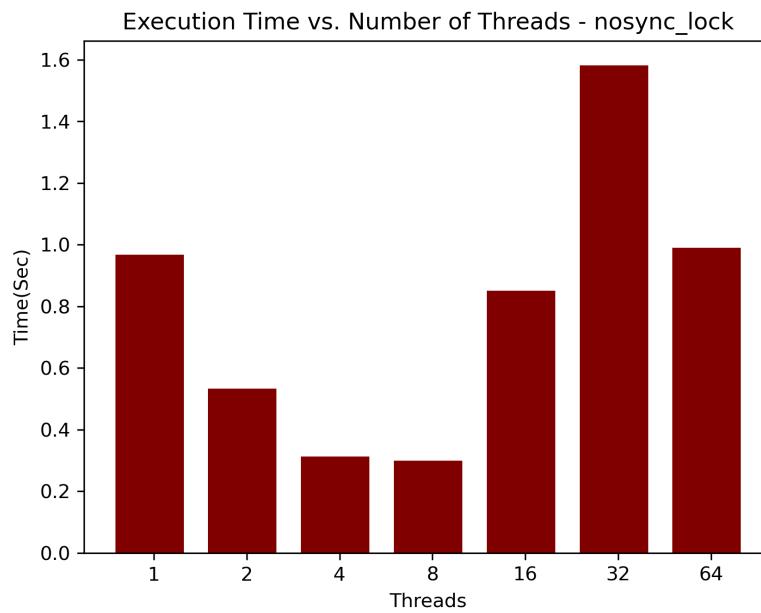
- Naive:

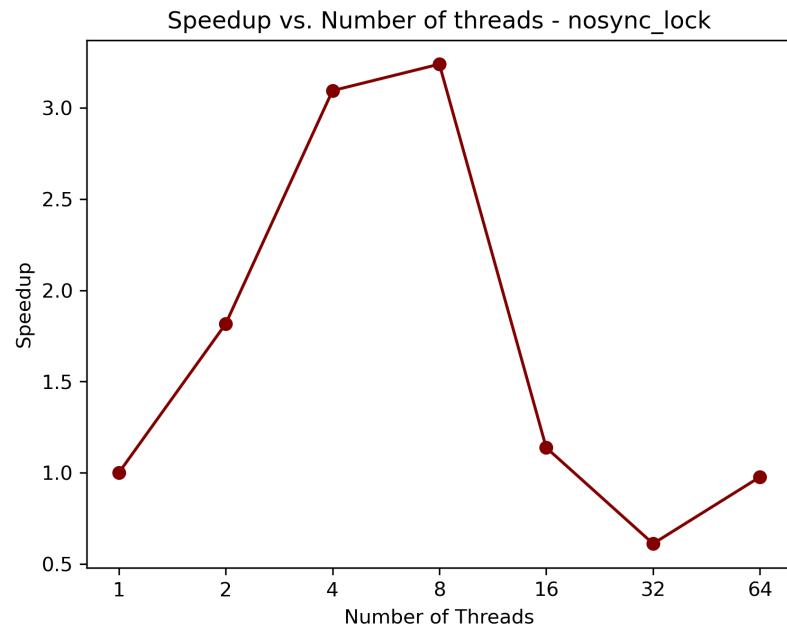




Στη Naive υλοποίηση χρησιμοποιούμε `#pragma omp atomic`, πριν τη πρόσβαση στο `newClusters` και `newClusterSize` ώστε να εξασφαλίσουμε atomic access σε αυτά.

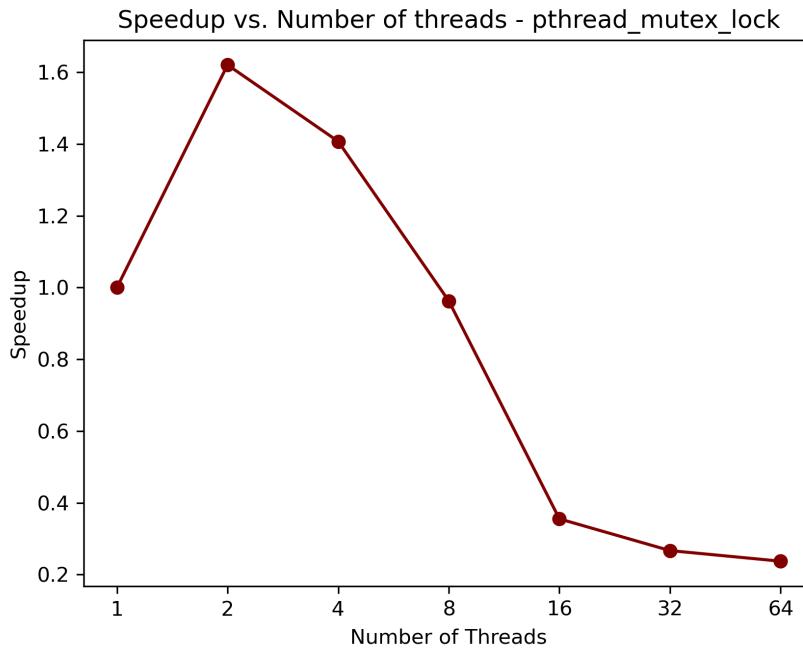
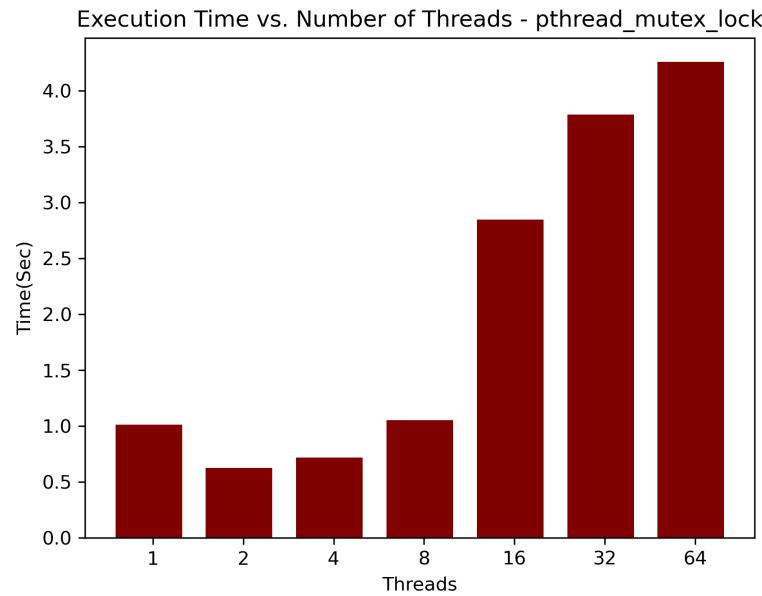
- NoSync Lock:





Το nosync lock όπως είναι αναμενόμενο έχει το λιγότερο συνολικό χρόνο για κάθε configuration νημάτων, δε φροντίζει για ατομικό access στα newClusters και newClusterSize και επομένως το clustering που κάνει είναι λάθος.

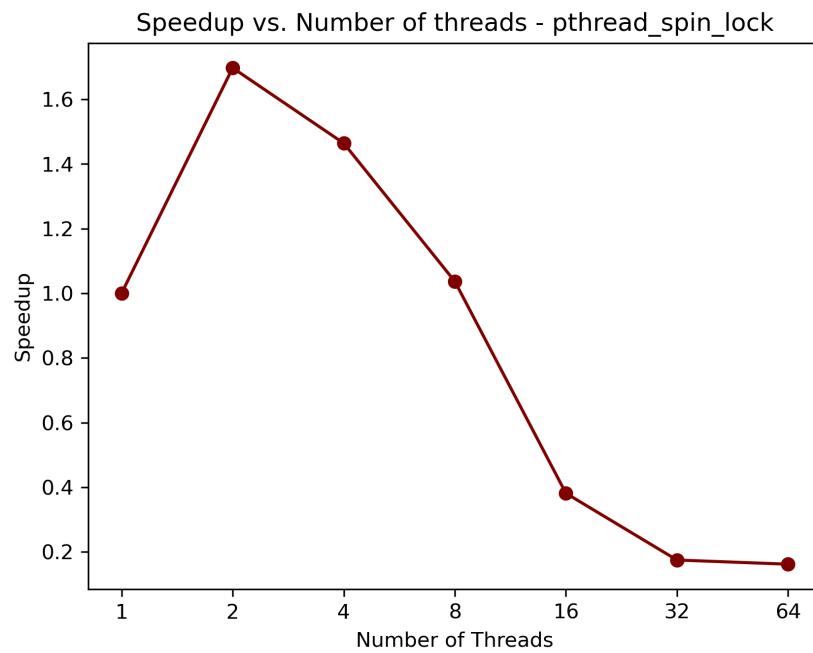
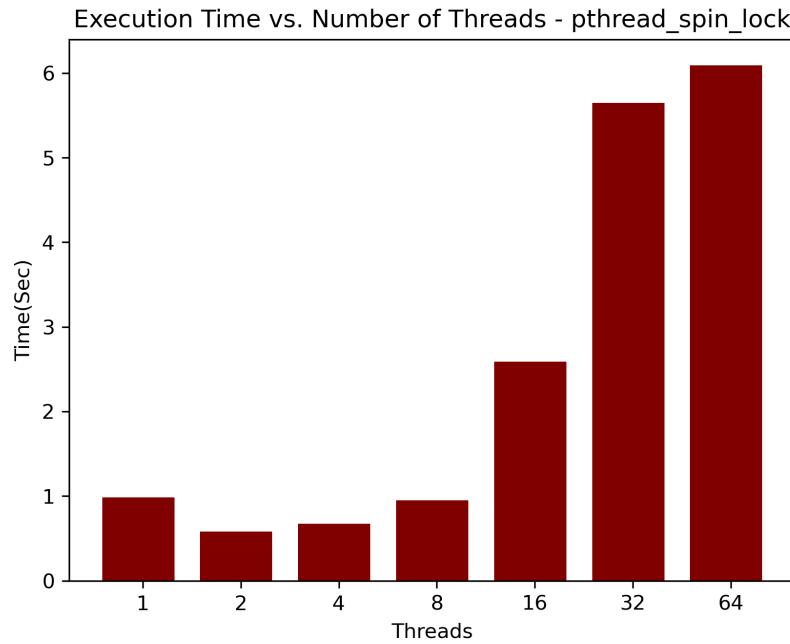
- Pthread Mutex Lock:



Στην υλοποίηση αυτή κάθε νήμα περιμένει να πάρει το lock κάνοντας κλήση στη συνάρτηση `acquire_lock()`, αν το mutex δεν είναι ελεύθερο το νήμα περιμένει (`sleep`) μέχρι να είναι ελεύθερο. Ξανά όπως και στις άλλες υλοποιήσεις μετα τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του

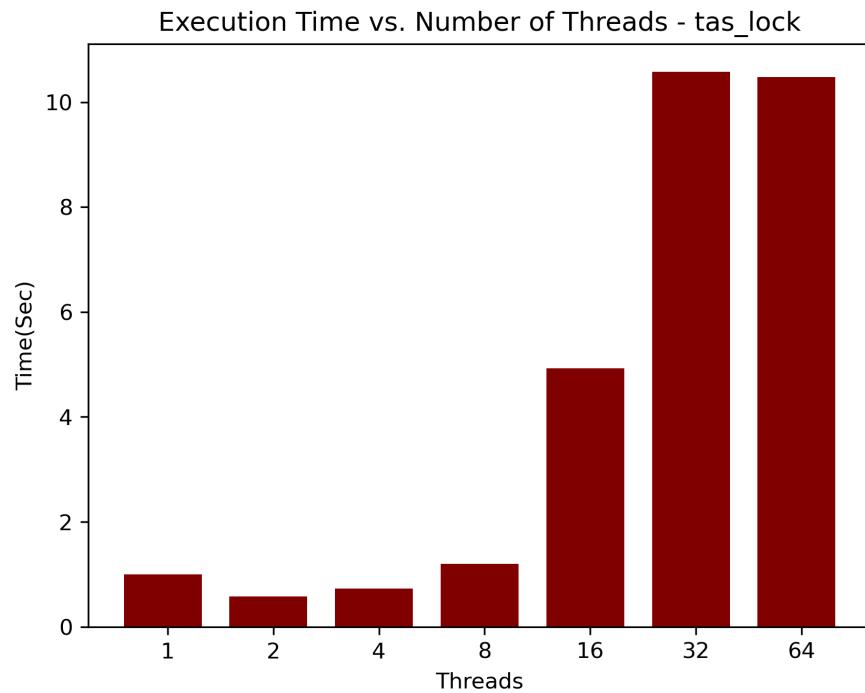
πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ από το contention για το lock).

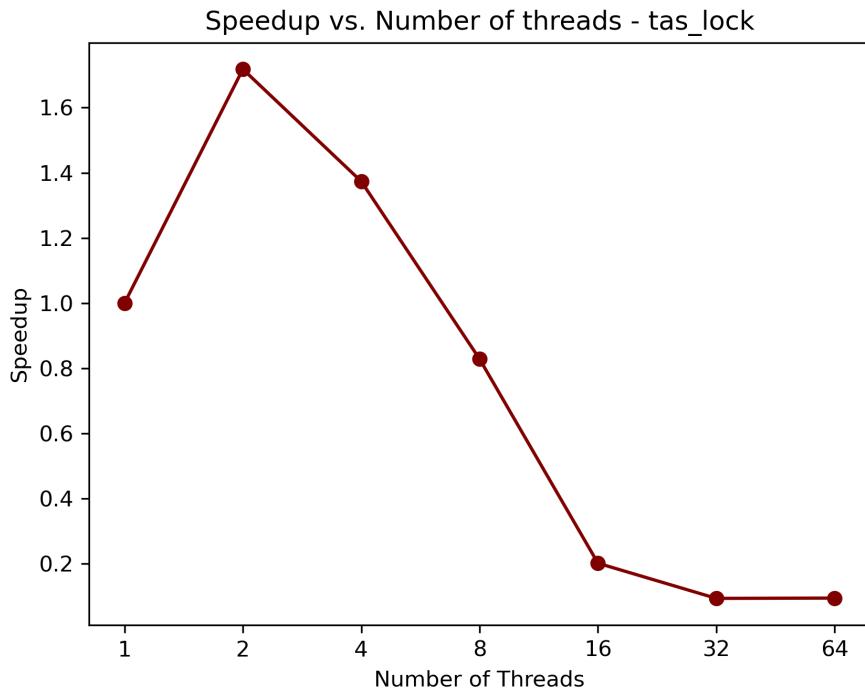
- Pthread Spin Lock:



Σε αυτή την υλοποίηση το thread προσπαθεί να πάρει το lock κάνοντας συνέχεια spin (τσεκάρει συνεχώς τη διαθεσιμότητα του lock) μέχρι να καταφέρει να το πάρει. Δεν εξασφαλίζεται κάποια σειριοποίηση αν και στη προκειμένη δε μας ενδιαφέρει. Ξανα όπως και στις άλλες υλοποιήσεις μετα τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ από το contention για το lock).

- TAS Lock





Η υλοποίηση αυτή βασίζεται στη λογική Test and Set, το κάθε νήμα με χρήση ατομικής εντολής `__sync_lock_test_and_set` γράφει `locked` στο state του lock και επιστρέφει πίσω την παλιά τιμή του state, αν δεν ηταν `unlocked` συνεχίζει το while loop.

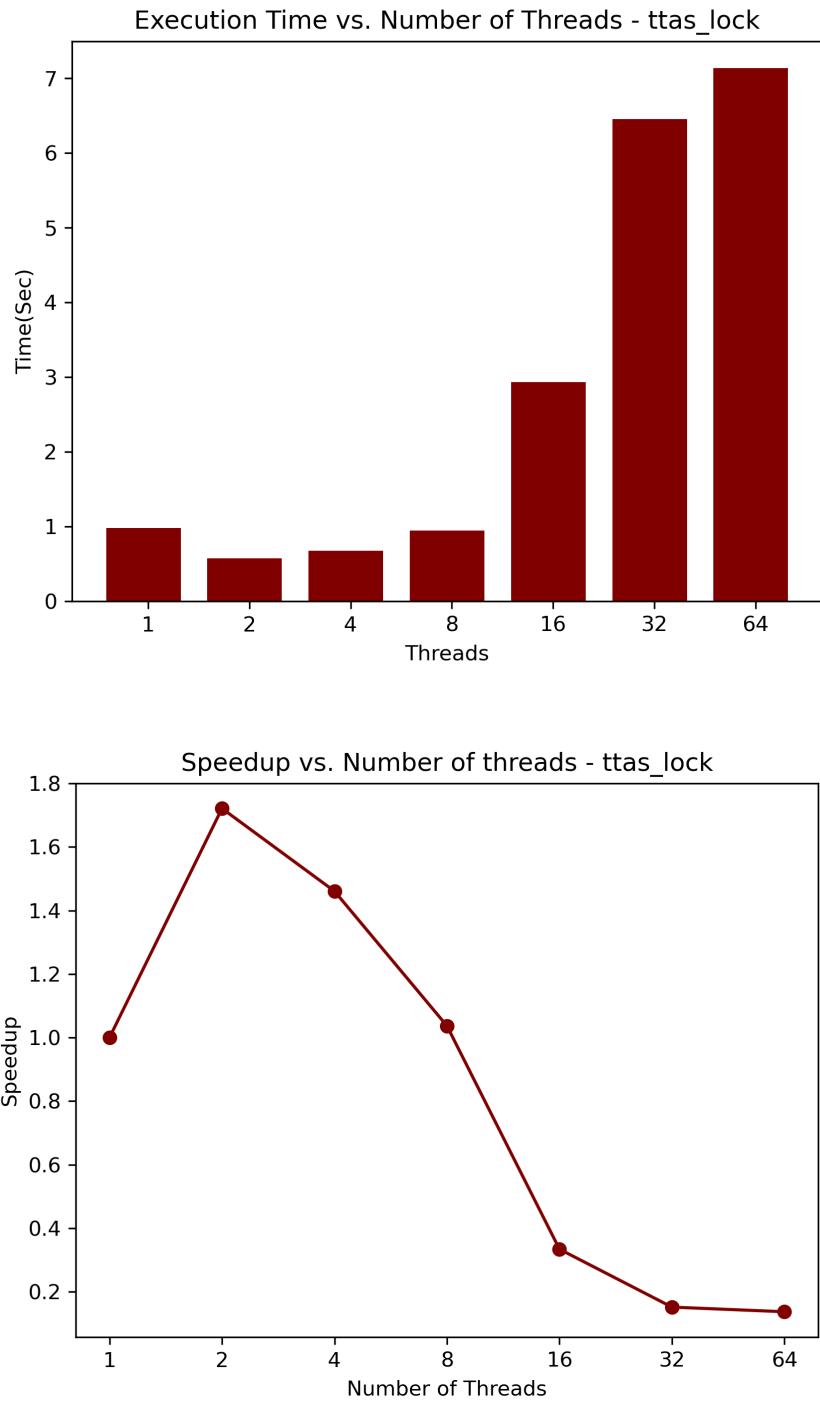
The `while` loop uses the `__sync_lock_test_and_set` function to attempt to acquire the lock. This function is a built-in function provided by GCC for atomic operations. It takes two arguments: a pointer to a memory location and a new value. The function atomically sets the value at the given memory location to the new value and returns the old value.

```
void lock_acquire(lock_t *lock)
{
    lock_t *l = lock;

    while (__sync_lock_test_and_set(&l->state, LOCKED) == LOCKED)
        /* do nothing */ ;
}
```

Ξανα όπως και στις άλλες υλοποιήσεις μετα τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ από το contention για το lock).

- TTAS Lock:



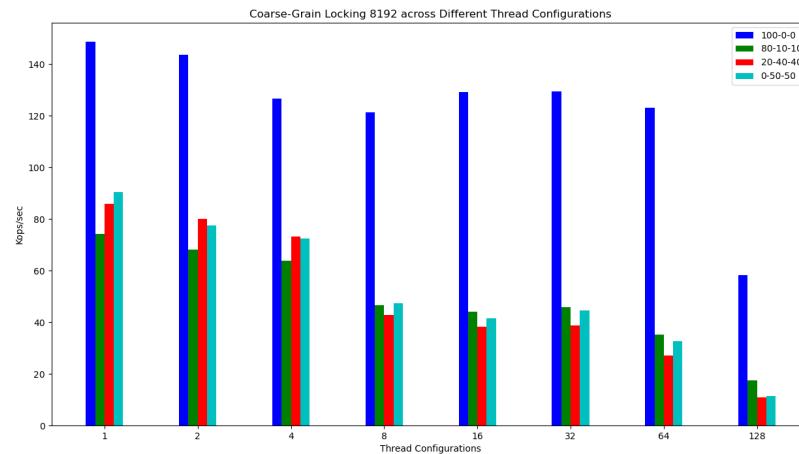
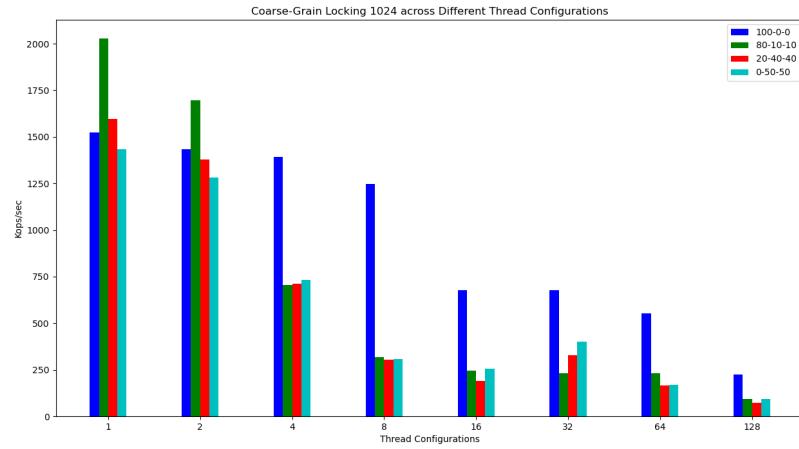
Σε αντίθεση με το tas_lock, το ttas_lock προσπαθεί να αποκτήσει το κλείδωμα μόνο όταν διαπιστώσει ότι το κλείδωμα είναι ελεύθερο. Αυτό σημαίνει ότι η ατομική εντολή που γράφει στην κοινόχρηστη μνήμη εκτελείται λιγότερες φορές, άρα έχουμε και λιγότερη κίνηση στο bus. Επομένως, αναμένουμε να δούμε βελτίωση της απόδοσης σε σχέση με πριν. Κάτι που παρατηρούμε καθώς σε όλα τα threads βλέπουμε μειωμένο χρόνο εκτέλεσης.

```
void lock_acquire(lock_t *lock)
{
    lock_t *l = lock;

    do {
        while (l->state == LOCKED)
            /* do nothing */ ;
        } while (__sync_lock_test_and_set(&l->state, LOCKED) == LOCKED);
}
```

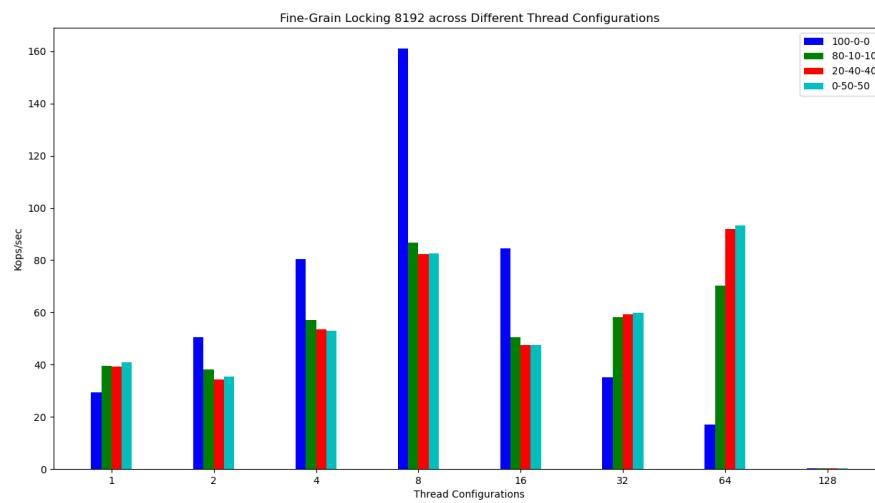
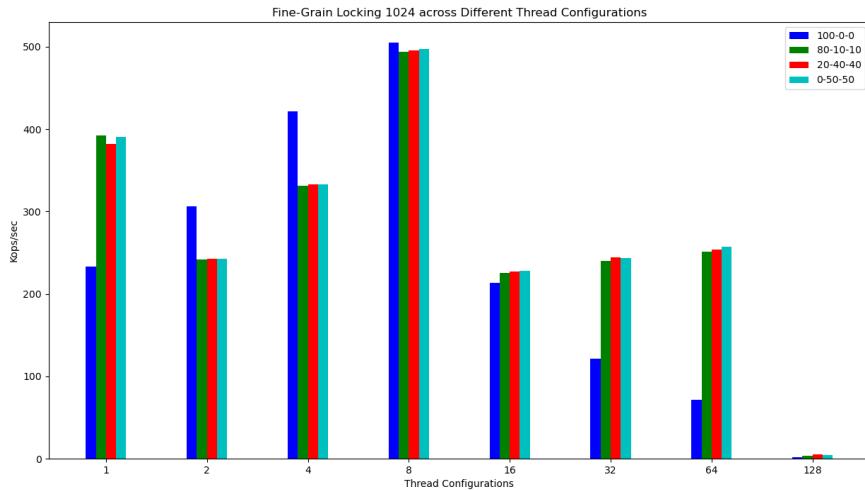
2.3 Ταυτόχρονες Δομές Δεδομένων

- Coarse-Grain Locking



Στο coarse-grain locking για καθε operation στη λίστα κλειδώνεται όλη η λίστα. Αυτό οδηγεί ,όπως είναι αναμενόμενο, στο να μειώνονται τα operations ανά second όσο περισσότερα threads εμπλέκουμε, τόσο στο μέγεθος 1024 όσο και στο 8192.

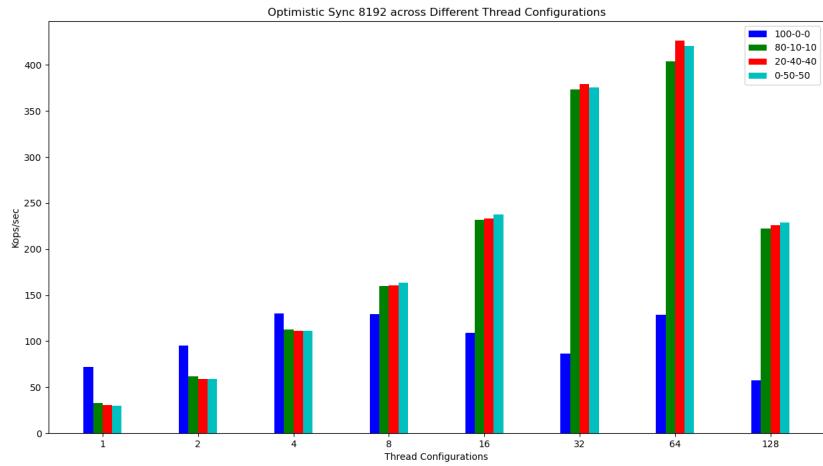
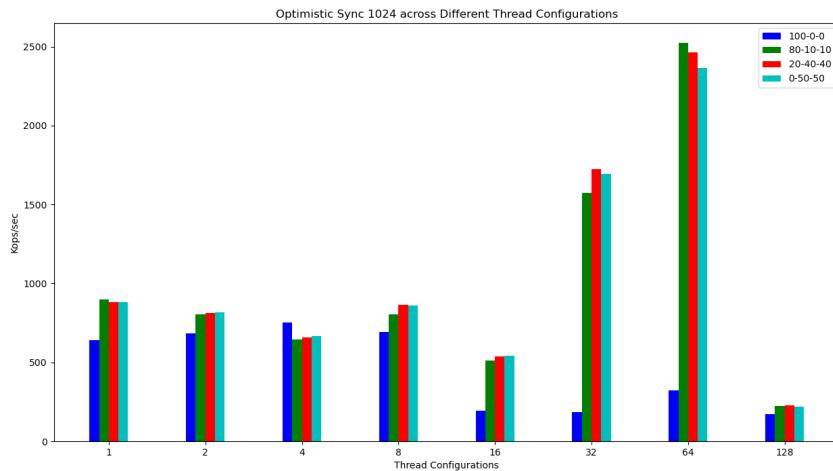
- **Fine-Grain Locking**



Στο fine grain locking δε κλειδώνει όλη τη λίστα καθε thread που θέλει να πραγματοποιήσει ένα operation, αλλα κλειδώνει σε επίπεδο node. Ένα νήμα πρέπει να πάρει το lock του πριν μπορέσει να το διαβάσει ή να εκτελέσει μια πράξη όπως εισαγωγή ή διαγραφή. Όπως είναι αναμενόμενο η απόδοση είναι ανοδική με αύξηση των threads. Ωστόσο χειροτερεύει όταν έχουμε μεγάλο ποσοστό αναζητήσεων, λόγω του hand-over-hand locking τα thread που θα

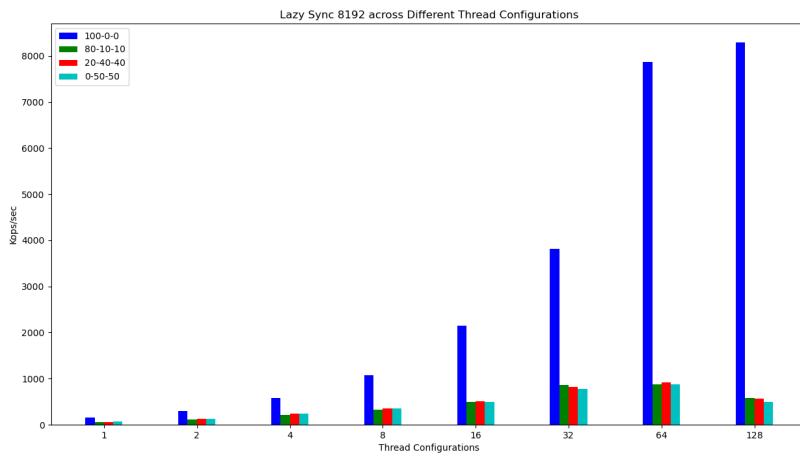
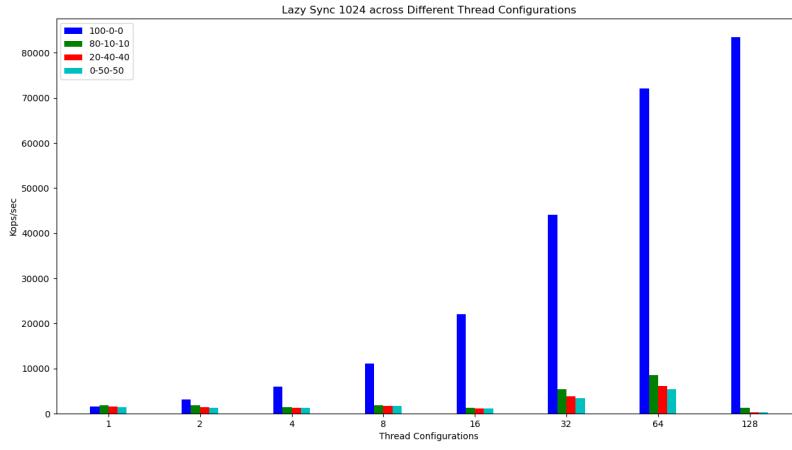
θέλουν να διαβάσουν πέρα από το locked node από ένα άλλο thread θα δημιουργούν “κίνηση” πίσω από αυτό, με αποτέλεσμα να σπαταλούν ώρα σε αναμονή παρά στη πράξη τους.

- **Optimistic Synchronization**



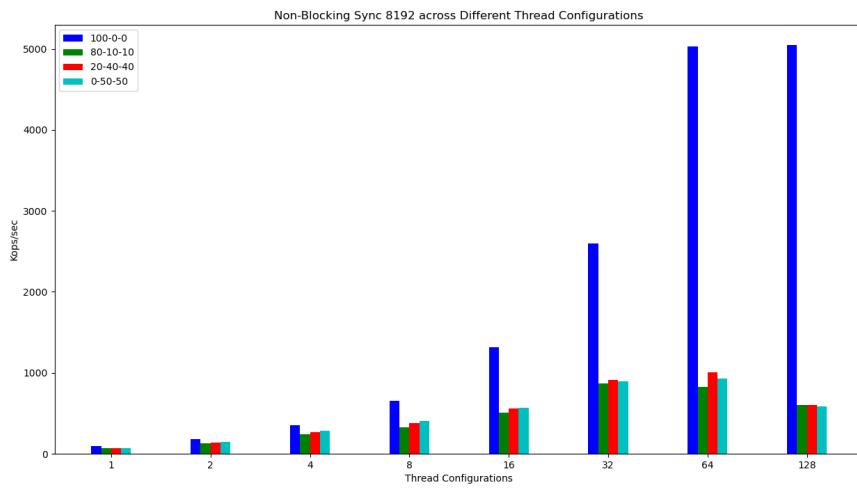
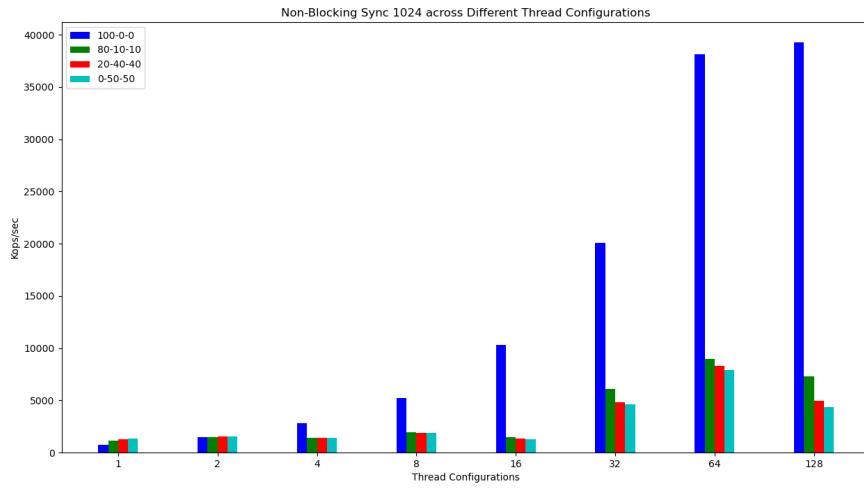
Το optimistic synchronization βασίζεται στην ιδέα ότι η σύγκρουση μεταξύ νημάτων δεν είναι συχνή, με αυτό το σκεπτικό κάθε νήμα αρχικά δουλεύει χωρίς locks και εκτελεί τα operations και μόνο στο τέλος κανει validate οτι δεν υπήρξε σύγκρουση με operation άλλου thread. Η διαφορά στην απόδοση συγκριτικά με άλλα locking schemas είναι πιο εμφανής στα configurations που περιλαμβάνουν εισαγωγές/διαγραφές. Ένα σημαντικό μειονέκτημα είναι ότι είναι πιθανό το starvation καθώς σε συστήματα που έχουν μεγάλη κίνηση είναι πιθανό τα νήματα να αποτυγχάνουν να κάνουν validate και να ξαναπροσπαθούν ξανα και ξανα.

- Lazy Synchronization



Στο Lazy synchronization προσθέτουμε στη δομή μία boolean μεταβλητή που δείχνει αν ο κόμβος βρίσκεται στη λίστα ή έχει διαγραφεί από αυτή. Η κύρια διαφορά σε σχέση με το optimistic synchronization είναι ότι η μέθοδος contains δε χρειάζεται να διατρέξει την λίστα από την αρχή σε περίπτωση που το validate αποτύχει. Γενικά παρατηρούμε πως πετυχαίνουμε “θετικό” scaling καθώς ανεβαίνουμε τα thread configurations.

- Non-Blocking Synchronization



To non-blocking synchronization αποτελεί τη λύση στην χρήση lock της lazy synchronization στα insert και delete tasks. Το overhead των locks απαλείφεται και αντικαθιστάται με reruns στην περίπτωση που δεν ισχύει η τωρινή κατάσταση κάποιου εκ των εξεταζόμενων κόμβων. Το μεγαλύτερο μειονέκτημα αυτής της στρατηγικής είναι η επιβάρυνση της find. Αυτή, κατά τη διάσχιση της λίστας, κάνει έναν έλεγχο σε κάθε κόμβο, ώστε να δει αν είναι marked. Αυτό επιτυγχάνεται συγχωνεύοντας την boolean που αναφέραμε στην Lazy υλοποίηση με το πεδίο next του κόμβου, το οποίο δείχνει στο επόμενο στοιχείο της λίστας. Έτσι, η boolean αυτή λειτουργεί τώρα ως flag που επιτρέπει την ενημέρωση του πεδίου next του κόμβου, εφόσον αυτός υπάρχει λογικά. Έχουμε πάλι “θετικό” scaling μεχρι και τα 64 threads.