



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

**ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
9ο εξάμηνο ΗΜΜΥ, ακαδημαϊκό έτος 2024-25**

Ομάδα: parlab38

Παναγιώτης Μπέλσης: el20874

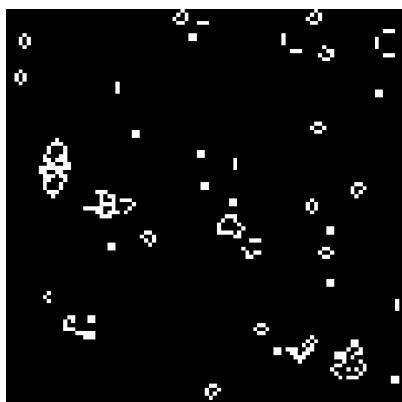
Γεώργιος Θεοδοσίου: el20109

Φίλιππος Μαρντιροσιάν: el20034

ΑΣΚΗΣΗ 1

Conway's Game of Life

- Αν ένα κελί είναι ζωντανό και έχει λιγότερους από 2 γείτονες πεθαίνει από μοναξιά.
- Αν ένα κελί είναι ζωντανό και έχει περισσότερους από 3 γείτονες πεθαίνει λόγω υπερπληθυσμού.
- Αν ένα κελί είναι ζωντανό και έχει 2 ή 3 γείτονες επιβιώνει μέχρι την επόμενη γενιά.
- Αν ένα κελί είναι νεκρό και έχει ακριβώς 3 γείτονες γίνεται ζωντανό (λόγω αναπαραγωγής).



Αρχικά τρέξαμε το πρόγραμμα *Game_Of_Life.c* τοπικά στον υπολογιστή (σειριακά) για έναν πίνακα 100x100 και μέσω του imagemagick παρατηρήσαμε το gif που προέκυψε.

Έπειτα για να επιτύχουμε την παραλληλοποίηση με τη χρήση του OpenMP τροποποιήσαμε το αρχικό αρχείο κώδικα (*Game_Of_Life.c*) εισάγοντας την παρακάτω γραμμή:

```
#pragma omp parallel for private(j, nbrs)
```

Με αυτόν τον τρόπο επιτυγχάνουμε την παραλληλοποίηση του `for()` loop, ώστε κάθε επανάληψη του loop να διανεμηθεί στα διαθέσιμα νήματα.

Επιπλέον, διασφαλίζουμε ότι κάθε νήμα θα έχει το δικό του αντίγραφο των μεταβλητών `j` και `nbrs`, έτσι ώστε να αποφευχθούν τα race conditions μεταξύ των threads.

```
gettimeofday(&ts,NULL);
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for private(j, nbrs) // Parallelize this loop
    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;
}
```

Έπειτα δημιουργήσαμε το παρακάτω Makefile

Makefile:

```
all: omp_gol

omp_gol: Game_Of_Life.c
        gcc -O3 -fopenmp -o GameOfLife Game_Of_Life.c

clean:
        rm GameOfLife
        rm performance_results.txt
```

Δημιουργία script : *make_on_queue.sh*

make_on_queue.sh:

```
#!/bin/bash

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

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

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

## How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Load necessary modules
module load openmp

## Navigate to the directory with your compiled program
cd /home/parallel/parlab38/a1

## Compile the code
make
```

Με τη χρήση του script *make_on_queue.sh* κάνουμε compile τον κώδικα και χρησιμοποιούμε OpenMP για παράλληλη επεξεργασία. Ζητάμε συγκεκριμένους πόρους (node=1, ppn=1), θέτουμε χρονικά όρια (10min) και δημιουργούμε τα αρχεία *game_of_life_performance.out* και *game_of_life_performance.err*

Δημιουργία script : *run_on_queue.sh*

run_on_queue.sh:

```
#!/bin/bash

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

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

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

## How long should the job run for?
#PBS -l walltime=00:20:00

## Start
## Load necessary modules
module load openmp

## Navigate to the directory with your compiled program
cd /home/parallel/parlab38/al

## Define board sizes and thread counts
board_sizes=(64 1024 4096)
thread_counts=(1 2 4 6 8)

## Loop over thread counts and board sizes
for N in "${board_sizes[@]}"; do
    for T in "${thread_counts[@]}"; do
        echo "Running Game of Life with size ${N}x${N} using ${T} threads for 1000 generations." >> performance_results.txt

        ## Set the number of threads
        export OMP_NUM_THREADS=$T

        ## Run the Game of Life executable and pass parameters
        ./GameOfLife $N 1000 >> performance_results.txt 2>&1 # Capture stderr as well
    done
done

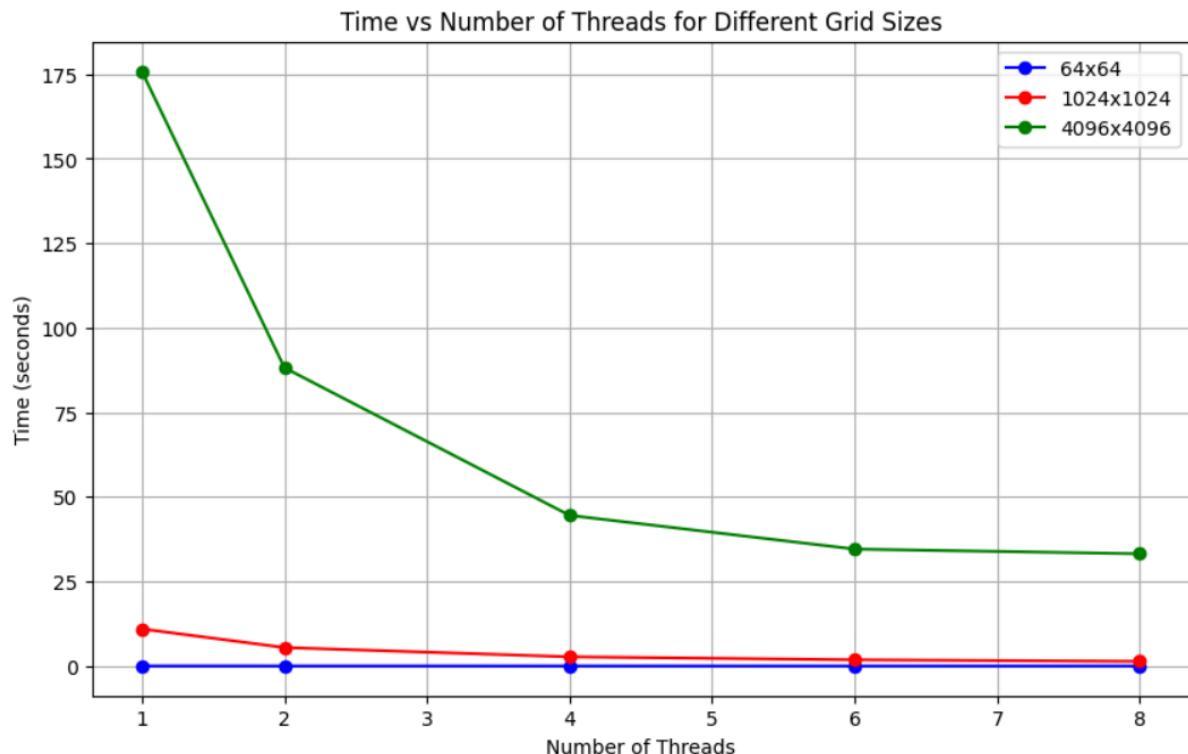
echo "Performance measurements complete. Results are saved in performance_results.txt." >> performance_results.txt
```

Με τη δημιουργία του script *run_on_queue.sh* θέτουμε χρονικό όριο run (20min), και με τη χρήση του OpenMp τρέχουμε παράλληλα (1 node σε 8 πυρήνες).
Το σύνολο των δεδομένων αποθηκεύονται στο *performance_results.txt* για 1,2,4,6,8 πυρήνες και μεγέθη ταμπλό 64x64, 1024x1024 και 4096x4096 (σε όλες τις περιπτώσεις τρέξαμε το παιχνίδι για 1000 γενιές)

Οι μετρήσεις που προκύπτουν είναι οι εξής:

	Grid Size	Threads	Time (seconds)
0	64x64	1	0.023139
1	64x64	2	0.013584
2	64x64	4	0.009996
3	64x64	6	0.008964
4	64x64	8	0.009662
5	1024x1024	1	10.964652
6	1024x1024	2	5.460825
7	1024x1024	4	2.723434
8	1024x1024	6	1.829067
9	1024x1024	8	1.375127
10	4096x4096	1	175.859194
11	4096x4096	2	88.213551
12	4096x4096	4	44.549267
13	4096x4096	6	34.578071
14	4096x4096	8	33.196358

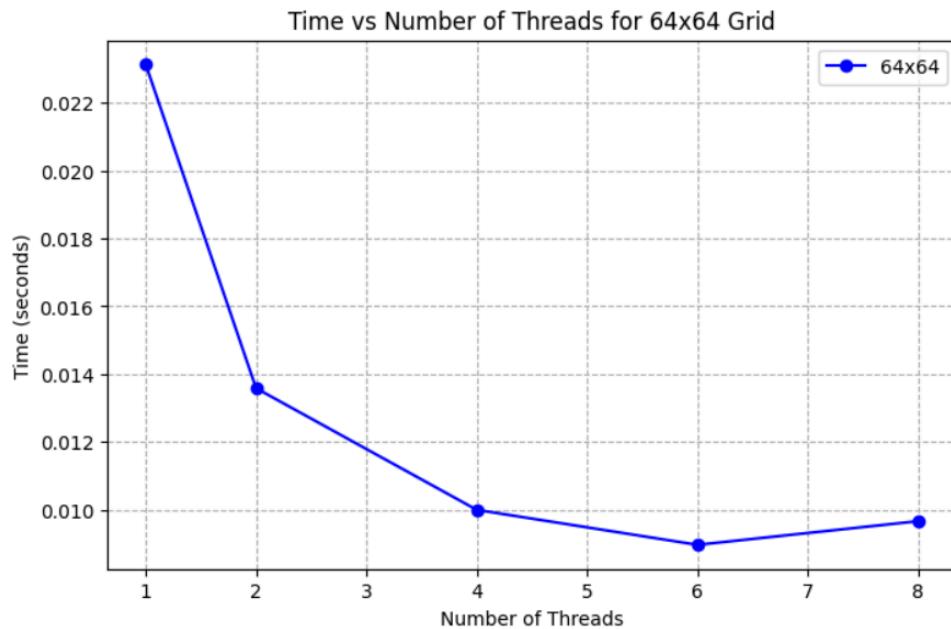
Έπειτα, τα κάναμε plot με χρήση της βιβλιοθήκης matplotlib της python:



Αρχικά, παρατηρούμε πως το μέγεθος ταμπλό (64x64, 1024x1024 και 4096x4096) επηρεάζει σε μεγάλο βαθμό τον χρόνο εκτέλεσης του προγράμματος. Προφανώς, όσο μεγαλύτερο το μέγεθος ταμπλό, τόσο μεγαλύτερος ο χρόνος εκτέλεσης.

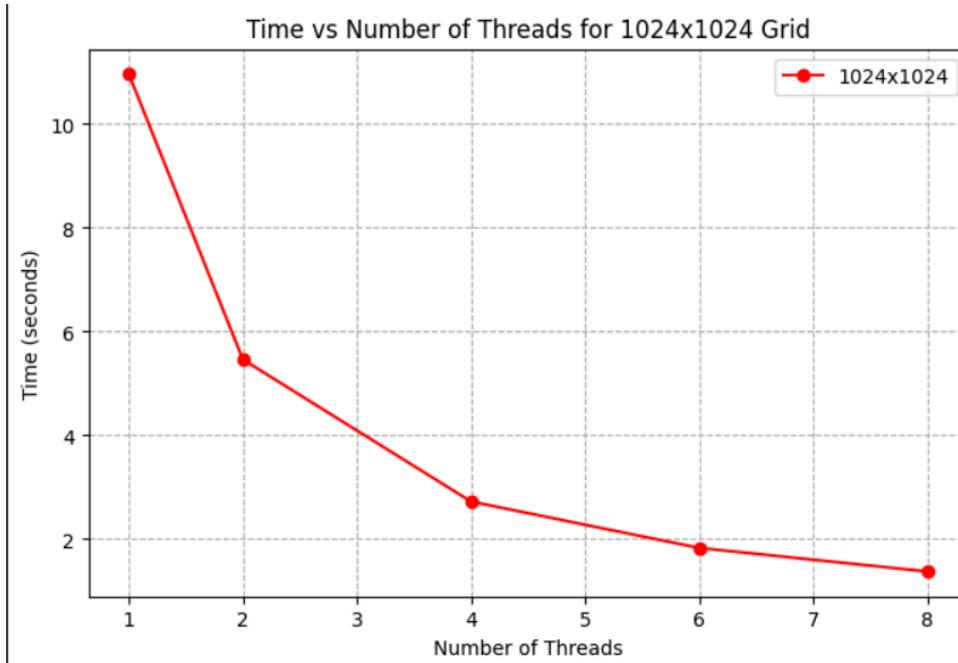
Η απόδοση, ακόμη, βελτιώνεται σημαντικά όταν αυξάνεται ο αριθμός των πυρήνων. Ειδικότερα, παρατηρούμε ότι σε γενικές γραμμές, όταν διπλασιάζεται ο αριθμός των πυρήνων, τότε υποδιπλασιάζεται ο χρόνος εκτέλεσης του προγράμματος.

Ας κοιτάξουμε λίγο πιο αναλυτικά την κάθε καμπύλη ξεχωριστά:

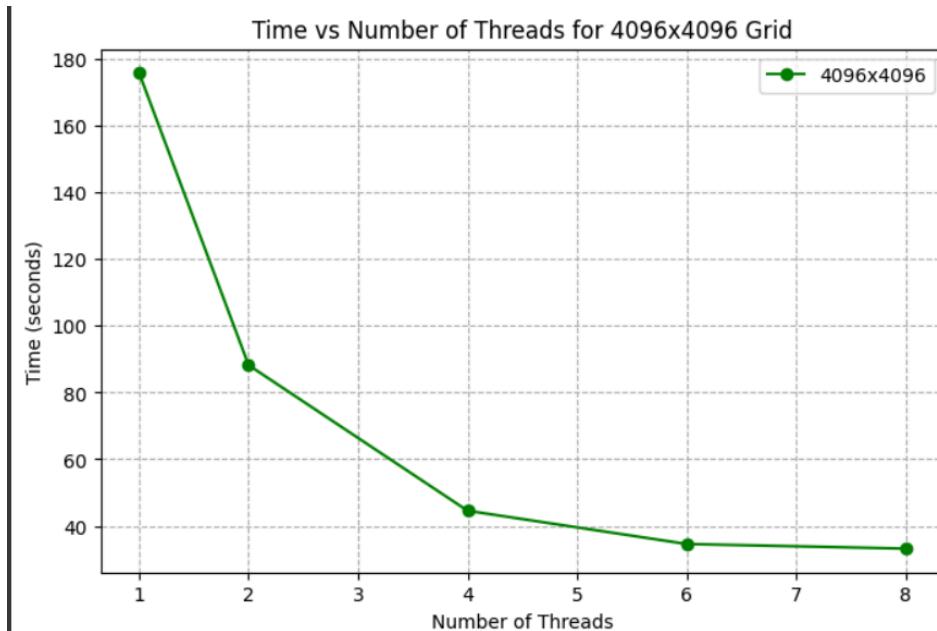


Τα συμπεράσματα είναι παρόμοια με προηγουμένως.

Η μόνη διαφορά είναι πως μεταξύ 6 και 8 πυρήνων, ο χρόνος εκτέλεσης χειροτερεύει ελαφρώς. Αυτό μπορεί να οφείλεται στο επιπλέον κόστος διαχείρισης των threads ή στην αναποτελεσματικότητα χειρισμού ενός μικρού προβλήματος (64x64 grid) με περισσότερους πυρήνες από όσους χρειάζονται.



Τα συμπεράσματα είναι ίδια με προηγουμένως.



Τα συμπεράσματα είναι ίδια με προηγουμένως.

ΑΣΚΗΣΗ 2.1 - Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Στην 2η άσκηση κληθήκαμε να διαχωρίσουμε N αντικείμενα σε k clusters (ομάδες) ακολουθώντας αρχικά τον αλγόριθμο **k-means**.

Στον φάκελο που μας δίνεται για την άσκηση /a2 πρόσβαση σε 3 αρχεία κώδικα με 3 διαφορετικές εκδοχές του k-means αλγορίθμου (sequential, omp_naive, omp_reduction).

Για **sequential εκτέλεση** του αλγορίθμου για διάφορα clusters προκύπτουν οι παρακάτω χρόνοι:

Για 3 clusters :

kmeans_seq -c 3 -s 100 -n 2 -t 0.001 -l 20
(total = 4.8994s)

Για 4 clusters :

kmeans_seq -c 4 -s 100 -n 2 -t 0.001 -l 20
(total = 4.4554s)

Για 16 clusters :

kmeans_seq -s 256 -n 8 -c 16 -l 10
(total = 9.09s)

Για **Omp_naive εκτέλεση**

Αλλάζαμε το αρχείο *omp_naive_kmeans.c* έτσι ώστε να παραλληλοποιήσουμε την εκτέλεση του αλγορίθμου kmeans με τη μέθοδο naive.

Αλλαγές που πραγματοποιήσαμε στον κώδικα του αρχείου : *omp_naive_kmeans.c*

```
do {  
    // before each loop, set cluster data to 0  
    for (i=0; i<numClusters; i++) {  
        for (j=0; j<numCoords; j++)  
            newClusters[i*numCoords + j] = 0.0;  
        newClusterSize[i] = 0;  
    }  
  
    delta = 0.0;  
  
    /*  
     * TODO: Detect parallelizable region and use appropriate OpenMP pragmas  
     */  
  
    #pragma omp parallel for private(index, j)  
    for (i=0; i<numObjs; i++) {
```

```

        // find the array index of nearest cluster center
        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords],
clusters);

        // if membership changes, increase delta by 1
        //maybe atomic here !!!
        if (membership[i] != index)
            delta += 1.0;

        // assign the membership to object i
        membership[i] = index;

        // update new cluster centers : sum of objects located within
        /*
         * TODO: protect update on shared "newClusterSize" array
         */
        #pragma omp atomic
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
        /*
         * TODO: protect update on shared "newClusters" array
         */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }

    // average the sum and replace old cluster centers with newClusters
    for (i=0; i<numClusters; i++) {
        if (newClusterSize[i] > 0) {
            for (j=0; j<numCoords; j++) {
                clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
newClusterSize[i];
            }
        }
    }
}

```

Έπειτα τροποποιήσαμε το script *run_on_queue.sh* έτσι ώστε να τρέξουμε το *omp_naive_kmeans.c* στον sandman για nthreads (1,2,4,8,16,32,64).

```

#!/bin/bash

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

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

## How many machines should we get?
#PBS -l nodes=1:ppn=64

##How long should the job run for?
#PBS -l walltime=00:10:00

```

```

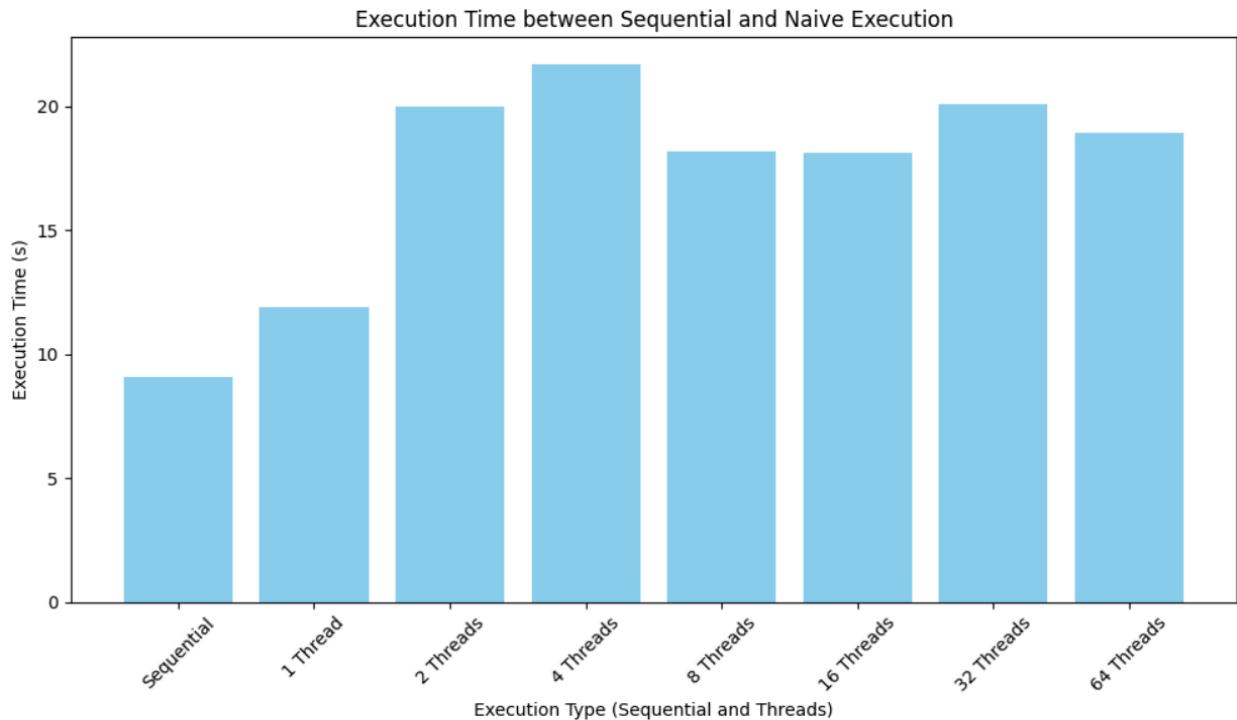
## Start
## Run make in the src folder (modify properly)

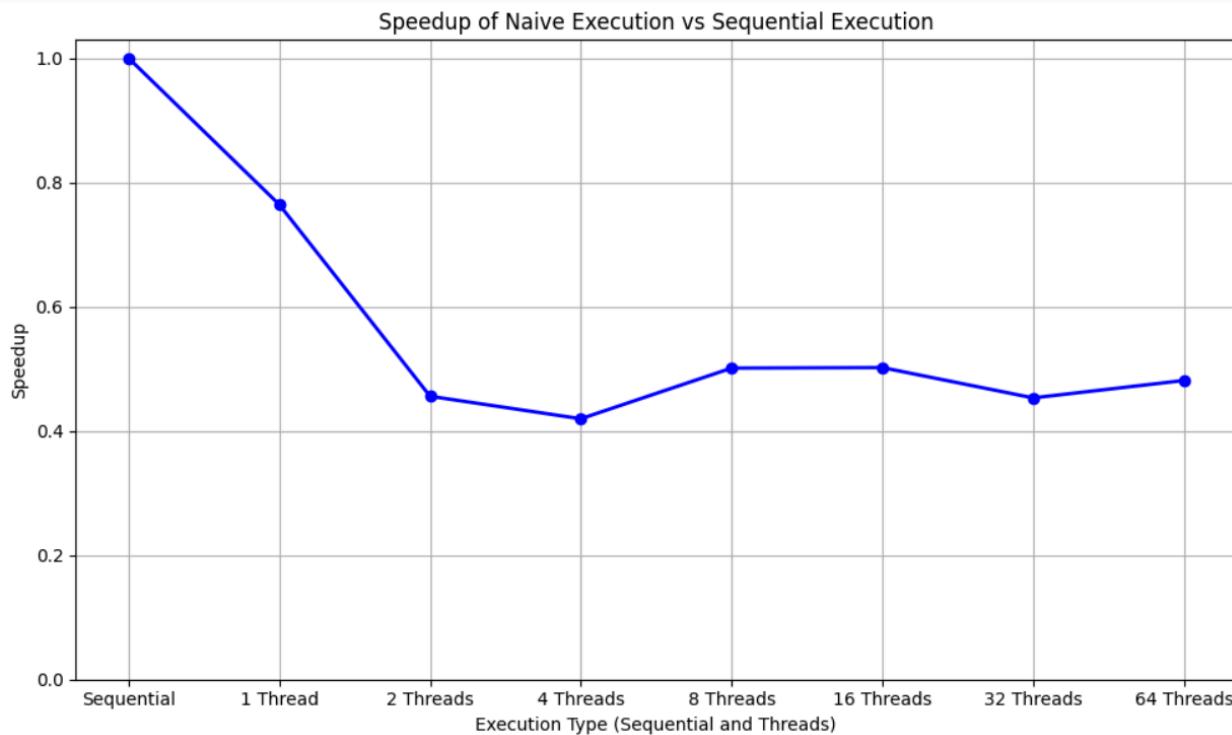
module load openmp
cd /home/parallel/parlab38/a2/kmeans

for threads in 1 2 4 8 16 32 64
do
    export OMP_NUM_THREADS=$threads
    export GOMP_CPU_AFFINITY="0-$((threads - 1))"
    echo "Running with $threads threads and CPU affinity set to $GOMP_CPU_AFFINITY"
./kmeans_omp_naive -s 256 -n 1 -c 4 -l 10
done

```

Από την παραπάνω εκτέλεση προκύπτει το ακόλουθο barplot που δείχνει ποιά περίπτωση είχε τον καλύτερο χρόνο:





Παρατηρούμε ότι η sequential εκτέλεση είναι πιο γρήγορη από την παράλληλη.

Στην παραλληλοποίηση βλέπουμε ότι για 8 και 16 threads έχουμε την καλύτερη επίδοση.

Όταν έχουμε λιγότερα threads (2 ή 4) απαιτείται μεγάλο overhead για πρόσβαση στη κοινή μνήμη, ενώ για περισσότερα threads (32 ή 64) ο μεγάλος κατακερματισμός της εργασίας οδηγεί στην περιορισμένη αξιοποίηση των caches και απαιτείται περισσότερος χρόνος για πρόσβαση στη μνήμη.

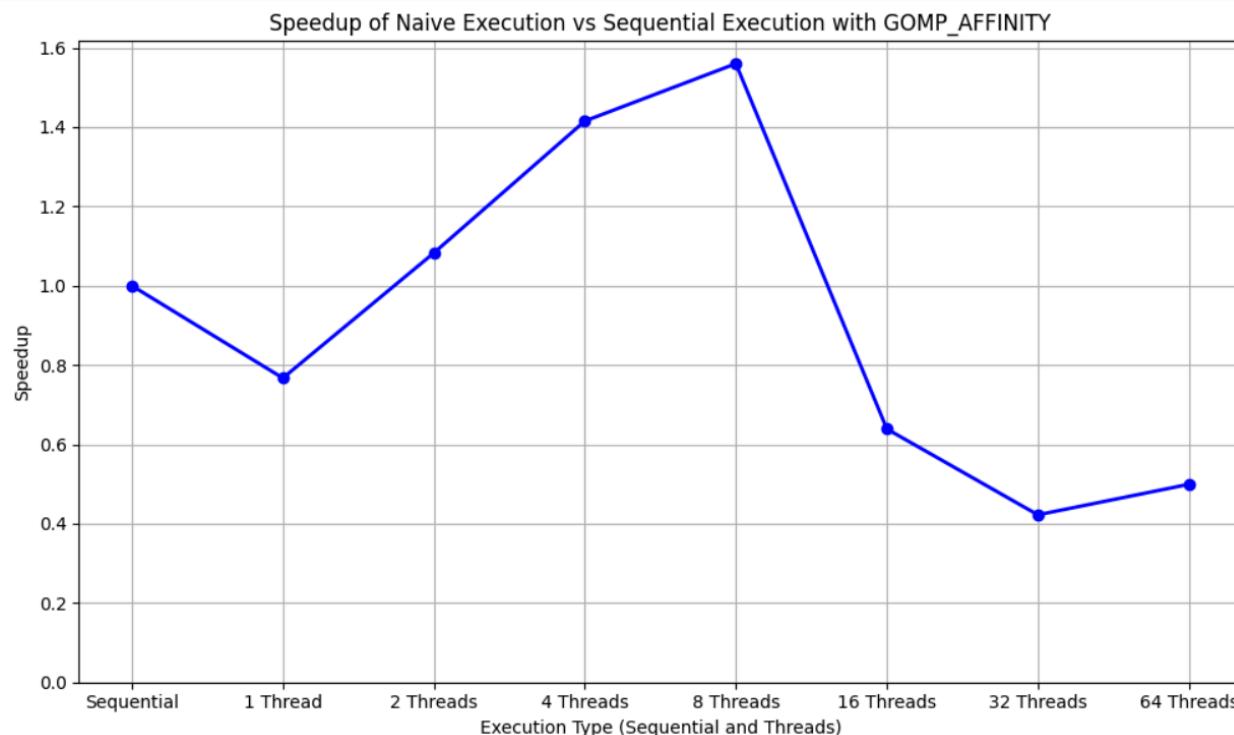
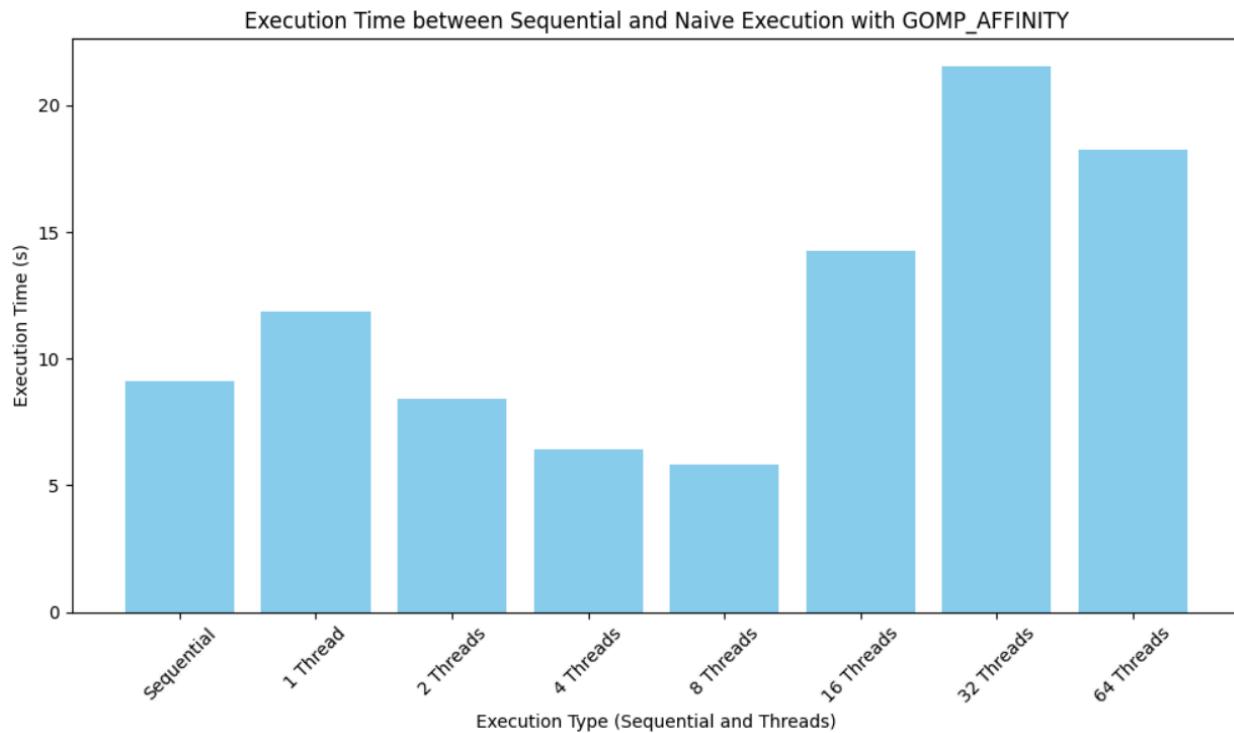
Ο Naive δεν είναι η πιο ενδειγμένη μέθοδος εκτέλεσης, καθώς όλα τα threads μοιράζονται τις ίδιες μεταβλητές γεγονός που προκαλεί race conditions. Για να αντιμετωπιστούν τα race conditions δηλώνουμε τις συγκεκριμένες μεταβλητές ως 'atomic' έτσι ώστε να μπορεί μόνο ένα νήμα κάθε φορά να αλλάζει την συγκεκριμένη μεταβλητή. Ωστόσο με αυτόν τον τρόπο προκύπτουν μεγαλύτερες καθυστερήσεις οι οποίες εξαλείφονται με τη μέθοδο reduction όπως θα δούμε παρακάτω.

Στη συνέχεια χρησιμοποιήσαμε την μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY.

Προσθέσαμε στο αρχείο `run_on_queue.sh` το παρακάτω:

```
export GOMP_CPU_AFFINITY="0-$((threads - 1))"
```

Και επαναλάβαμε τις μετρήσεις οπότε προέκυψαν τα παρακάτω διαγράμματα:



Πλέον, παρατηρούμε ότι όσο αυξάνεται ο αριθμός των threads (2,4,8) έχουμε καλύτερη επίδοση, ωστόσο για μεγάλο αριθμό threads (16,32,64) έχουμε παρόμοια αποτελέσματα με προηγουμένως. Το γεγονός αυτό οφείλεται στο **thread binding** δηλαδή τα thread προσδένονται σε συγκεκριμένους πυρήνες για όλη την εκτέλεση.

Για Omp_reduction εκτέλεση

Αλλάξαμε το αρχείο *omp_reduction_kmeans.c* έτσι ώστε να παραλληλοποιήσουμε την εκτέλεση του αλγορίθμου kmeans με τη μέθοδο reduction.

Αλλαγές που πραγματοποιήσαμε στον κώδικα :

omp_reduction_kmeans.c

```
timing = wtime();
do {
    // before each loop, set cluster data to 0
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++)
            newClusters[i*numCoords + j] = 0.0;
        newClusterSize[i] = 0;
    }

    delta = 0.0;

    /*
     * TODO: Initialize local cluster data to zero (separate for each thread)
     */

    for (k=0; k<nthreads; k++) {
        for (i=0; i<numClusters; i++) {
            for (j=0; j<numCoords; j++)
                local_newClusters[k][i*numCoords + j] = 0.0;
            local_newClusterSize[k][i] = 0;
        }
    }

    #pragma omp parallel for private(index, j)
    for (i=0; i<numObjs; i++)
    {
        // find the array index of nearest cluster center
        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords],
clusters);

        // if membership changes, increase delta by 1
        if (membership[i] != index)
            delta += 1.0;

        // assign the membership to object i
        membership[i] = index;

        // update new cluster centers : sum of all objects located within (average will
be performed later)
        /*
         * TODO: Collect cluster data in local arrays (local to each thread)
         * Replace global arrays with local per-thread
         */
        int threadid = omp_get_thread_num();
        local_newClusterSize[threadid][index]++;
        for (j=0; j<numCoords; j++)
            local_newClusters[threadid][index*numCoords + j] += objects[i*numCoords
+ j];
    }
}
```

```

//local_newClusters[threadid][index*numCoords + j] += 1;

//printf("Inside parallel for, i: %d\n", i);
//fflush(stdout);
}

/*
 * TODO: Reduction of cluster data from local arrays to shared.
 * This operation will be performed by one thread
 */

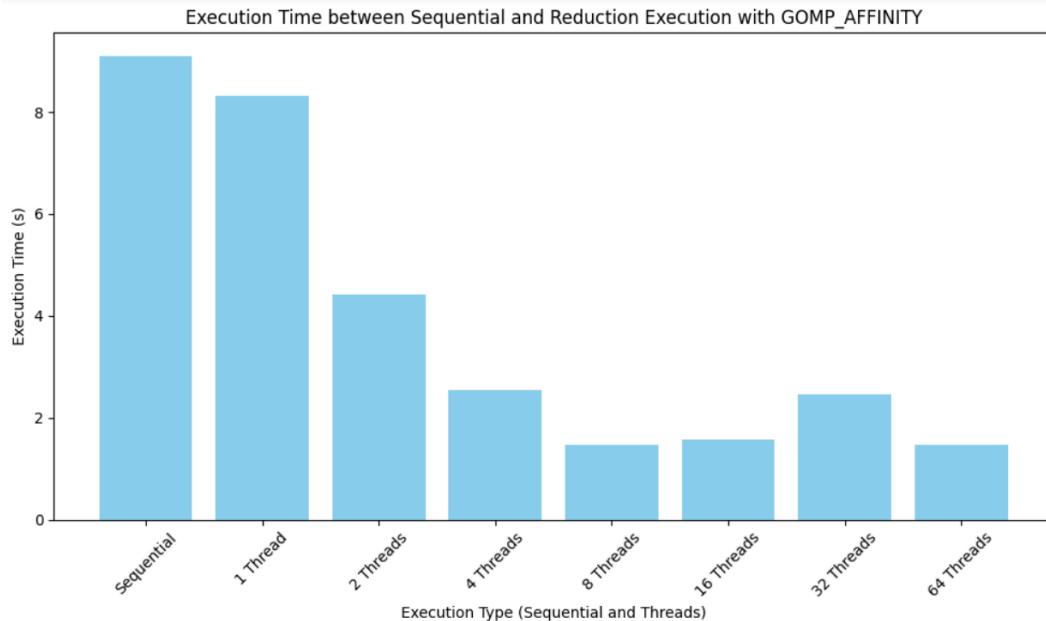
for (k=0; k<nthreads; k++) {
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            newClusters[i*numCoords + j] += local_newClusters[k][i*numCoords + j];
        }
        newClusterSize[i] += local_newClusterSize[k][i];
    }
}

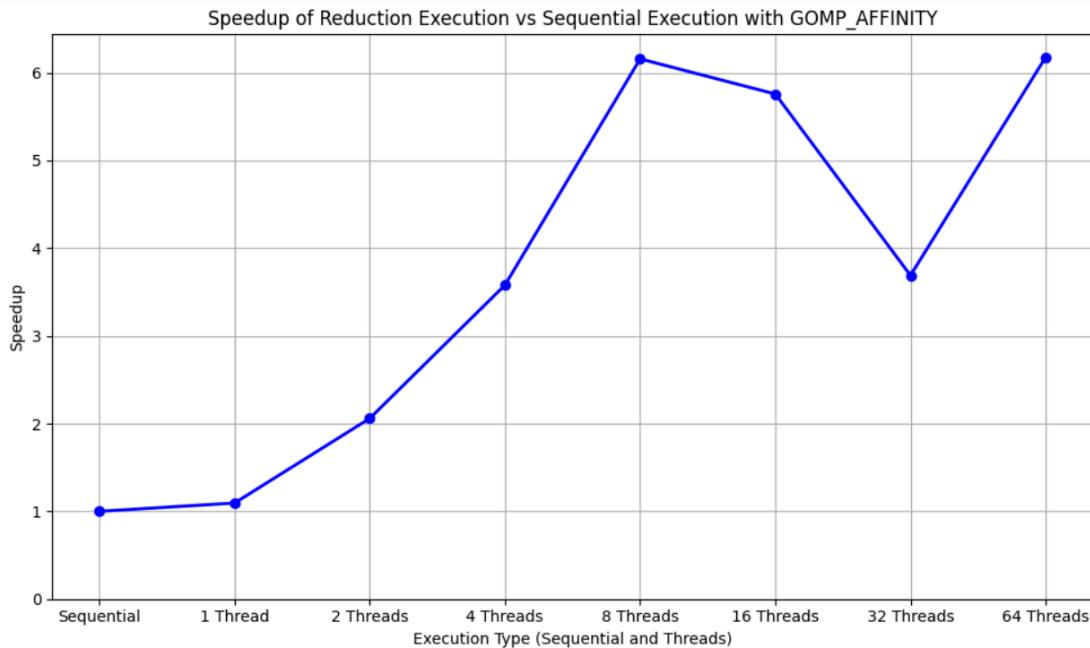
// average the sum and replace old cluster centers with newClusters
for (i=0; i<numClusters; i++) {
    if (newClusterSize[i] > 0) {
        for (j=0; j<numCoords; j++) {
            clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
newClusterSize[i];
        }
    }
}

```

Φτιάχνουμε πάλι το *script run_on_queue.sh* έτσι ώστε να τρέξουμε το *omp_reduction_kmeans.c* στον sandman για nthreads (1,2,4,8,16,32,64).

To plot που προκύπτει από την εκτέλεση του reduction:

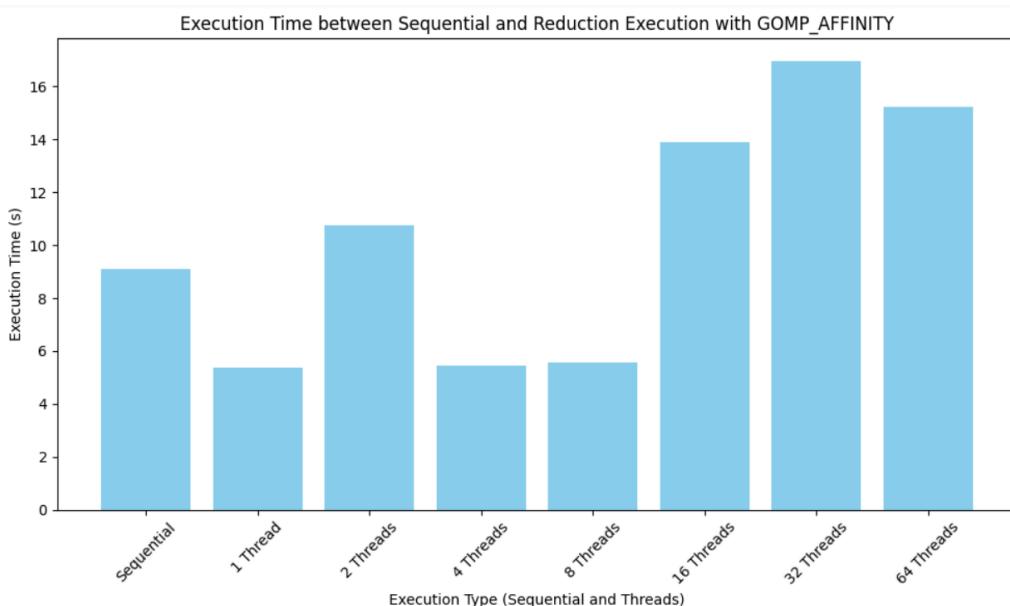


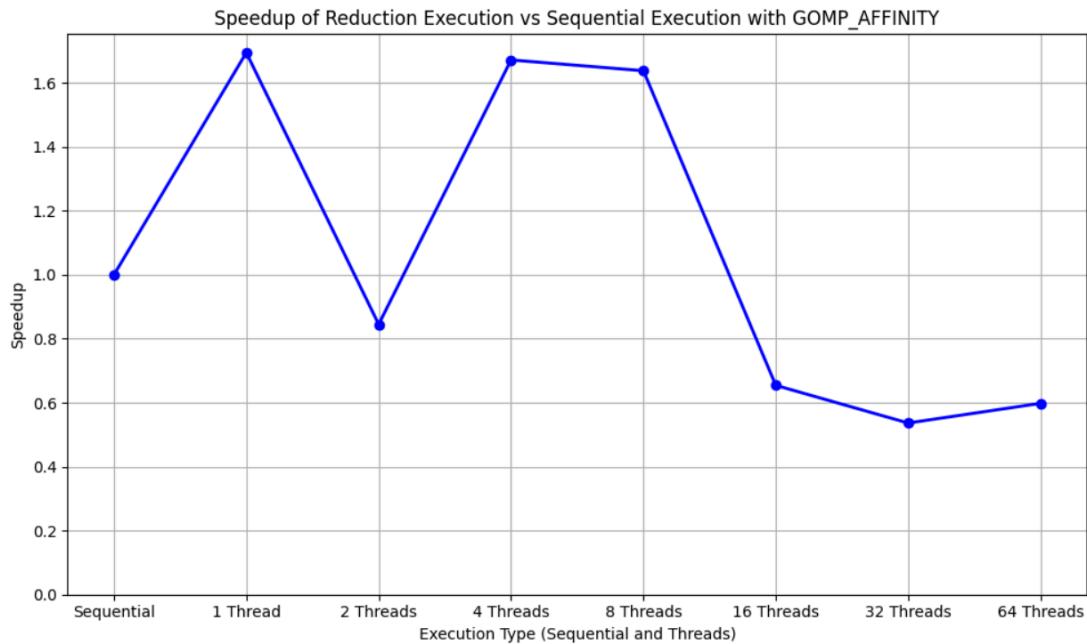


Σε σύγκριση με την προηγούμενη έκδοση έχουμε αρκετά καλύτερα αποτελέσματα.

Παρατηρούμε ότι με την αύξηση των threads μειώνεται ο συνολικός χρόνος, καθώς κατανέμονται καλύτερα οι πόροι. Σε αυτή τη περίπτωση, κάθε thread έχει το δικό του αντίγραφο των μεταβλητών (local πίνακες) τις οποίες ενημερώνει (κάθε thread παράλληλα τις δικές του μεταβλητές). Στο τέλος, όλα τα threads ενημερώνουν τις global μεταβλητές. Έτσι ελαχιστοποιούμε τις προσβάσεις σε shared δεδομένα και αποφεύγονται οι καθυστερήσεις που προέκυπταν προηγουμένως.

Παρακάτω φαίνονται τα διαγράμματα για το configuration: {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}





Παρατηρούμε ότι σε σχέση με το προηγούμενο configuration, ενώ έχουμε μικρότερο input, έχουμε σαφώς χειρότερα αποτελέσματα.

Το γεγονός αυτό οφείλεται στο **first-touch policy** του Linux και στα φαινόμενα **false-sharing**.

Στο first-touch policy όταν ένα thread έχει πρόσβαση για πρώτη φορά σε μια σελίδα μνήμης τότε αυτή ανήκει στο συγκεκριμένο thread και τα δεδομένα της σελίδας τοποθετούνται στη μνήμη τοπικά στο thread.

Επιθυμούμε κάθε thread να έχει πρόσβαση κοντά στους local πίνακες (local_newClusters) που διαχειρίζεται. Αυτό μπορεί να επιτευχθεί με χρήση malloc αντί για calloc, καθώς και την προσθήκη ενός parallel for.

Ταυτόχρονα, λόγω του μικρού input έχουμε φαινόμενα false-sharing, καθώς στην ίδια γραμμή της cache μπορεί να έχουμε γραμμές διαφορετικών local πινάκων. Έτσι, όταν ένα thread επεξεργάζεται μια συγκεκριμένη γραμμή της cache και πραγματοποιεί αλλαγές στα δεδομένα, τότε η συγκεκριμένη γραμμή γίνεται invalidate.

ΑΣΚΗΣΗ 2.2 - Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Στο 2o σκέλος της άσκησης μας ζητείται να υλοποιήσουμε τον αλγόριθμο **FW (Floyd-Warshall)** για την εύρεση ελάχιστων αποστάσεων για κάθε ζεύγος κόμβων σε έναν γράφο.

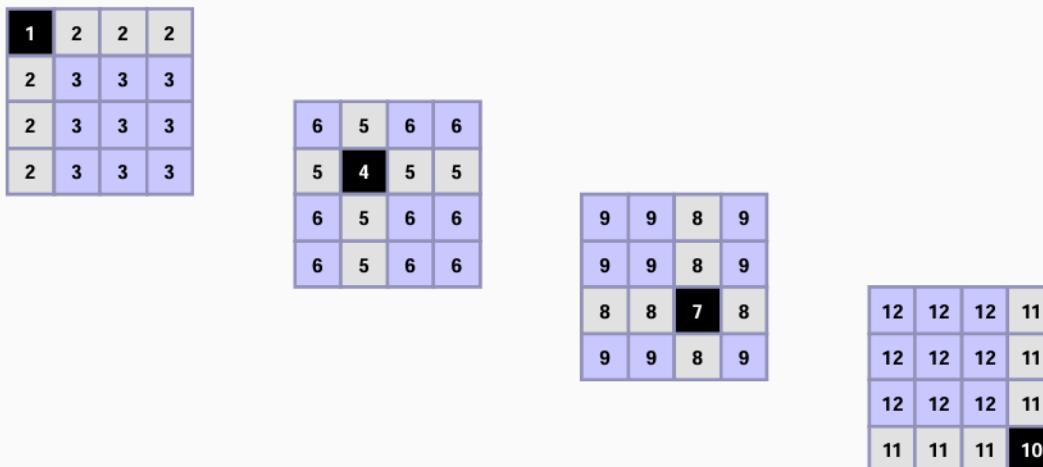
Μας δίνονται 3 εκδοχές του αλγορίθμου FW: fw.c, fw_tiled.c, fw_sr.c με βάση το paper J.-S. Park, M. Penner, and V. K. Prasanna, “Optimizing Graph Algorithms for Improved Cache Performance,” IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 9, SEPTEMBER 2004.

Κάναμε παραλληλοποίηση του κλασικού αλγορίθμου fw αλλά και του recursive και του tiled fw.

Recursive main idea :

FWR (A, B, C) if (base case) FWI (A, B, C) else FWR (A ₀₀ , B ₀₀ , C ₀₀); FWR (A ₀₁ , B ₀₀ , C ₀₁); FWR (A ₁₀ , B ₁₀ , C ₀₀); FWR (A ₁₁ , B ₁₀ , C ₀₁); FWR (A ₁₁ , B ₁₀ , C ₀₁); FWR (A ₁₀ , B ₁₀ , C ₀₀); FWR (A ₀₁ , B ₀₀ , C ₀₁); FWR (A ₀₀ , B ₀₀ , C ₀₀);	FWI (A, B, C) for (k=0; k<N; k++) for (i=0; i<N; i++) for (j=0; j<N; j++) A[i][j] = min(A[i][j], B[i][k]+C[k][j]);
--	--

Tiled main idea:



Εξήγηση: Η εκτέλεση του tiled αποτελείται από 3 phases

Στο πρώτο phase εκτελούμε FW στο κουτί k*k (μαύρο κουτί)

Στο δεύτερο phase εκτελούμε FW στα κουτιά πάνω, κατω, δεξιά και αριστερά από το μαύρο κουτί.

Στο τρίτο phase εκτελούμε FW στα εναπομείναντα κουτιά (μπλέ).

Έχουμε το παρακάτω script *run_on_queue.sh*, το οποίο αλλάζουμε αναλόγως το εκτελέσιμο που θέλουμε να τρέξουμε (*./fw* ή *./fw_sr* ή *./fw_tiled*). Προφανώς, κάθε φορά αλλάζουμε και τα δεδομένα εισόδου (όπως τα μεγέθη πινάκων, τα μεγέθη των block, και τα μεγέθη των tile):

Script *run_on_queue.sh*

```
#!/bin/bash

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

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

## How many machines should we get?
#PBS -l nodes=1:ppn=64

##How long should the job run for?
#PBS -l walltime=00:20:00

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

module load openmp
cd /home/parallel/parlab38/a2/FW

nthreads=( 1 2 4 8 16 32 64 )
sizes=( 1024 2048 4096 )
block=( 256 )

for BSIZE in "${block[@]}";
do
    for size in "${sizes[@]}";
    do
        echo "-----"
        for nthread in "${nthreads[@]}";
        do
            echo
            export OMP_NUM_THREADS=${nthread};
            echo "Running with $nthread threads"
            ## echo "Running with $size size"
            ## echo "Running with $BSIZE bsize"
            ./fw_sr ${size} ${BSIZE}
        done
    done
done
## ./fw <SIZE>
## ./fw_sr <SIZE> <BSIZE>
## ./fw_tiled <SIZE> <BSIZE>
```

Αλλαγές που κάναμε στον κώδικα *fw_sr.c* για την παραλληλοποίηση του :

```
/*
 * Recursive implementation of the Floyd-Warshall algorithm.
 * command line arguments: N, B
 * N = size of graph
 * B = size of submatrix when recursion stops
 * works only for N, B = 2^k
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "util.h"
#include <omp.h>

inline int min(int a, int b);

void FW_SR (int **A, int arow, int acol,
            int **B, int brow, int bcol,
            int **C, int crow, int ccol,
            int myN, int bsize);

int main(int argc, char **argv)
{
    int **A;
    int i,j;
    struct timeval t1, t2;
    double time;
    int B=16;
    int N=1024;

    if (argc !=3){
        fprintf(stdout, "Usage %s N B \n", argv[0]);
        exit(0);
    }

    N=atoi(argv[1]);
    B=atoi(argv[2]);

    A = (int **) malloc(N*sizeof(int *));
    for(i=0; i<N; i++) A[i] = (int *) malloc(N*sizeof(int));

    graph_init_random(A,-1,N,128*N);

    gettimeofday(&t1,0);
    #pragma omp parallel
    {
        #pragma omp single
        FW_SR(A,0,0, A,0,0,A,0,0,N,B);
    }
    gettimeofday(&t2,0);

    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)/1000000;
    printf("FW_SR,%d,%d,%4f\n", N, B, time);

    /*
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
    */
}
```

```

        return 0;
    }

    inline int min(int a, int b)
    {
        if(a<=b) return a;
        else return b;
    }

    void FW_SR (int **A, int arow, int acol,
                int **B, int brow, int bcol,
                int **C, int crow, int ccol,
                int myN, int bsize)
    {
        int k,i,j;

        /*
         * The base case (when recursion stops) is not allowed to be edited!
         * What you can do is try different block sizes.
         */
        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 {
                // #pragma omp parallel
                //{
                    // #pragma omp single nowait
                    //{
                        FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

                        #pragma omp task shared(A,B,C) if (0)
                        {
                            #pragma omp task shared(A,B,C)
                                FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2,
bsize);
                                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 shared(A,B,C) if (0)
                        {
                            #pragma omp task shared(A,B,C)
                                FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
ccol, myN/2, bsize);
                                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);
                    //}
                }
            }
        }
    }
}

```

```
// }  
}  
}
```

Επεξήγηση αλλαγών:

Αρχικά, το #parallel for δεν θα ήταν κατάλληλο εδώ, επειδή δεν μπορεί να διαχειριστεί την αναδρομική δομή των κλήσεων. Τα tasks, αντιθέτως, επιτρέπουν την αναδρομική κλήση να παραλληλοποιηθεί δυναμικά.

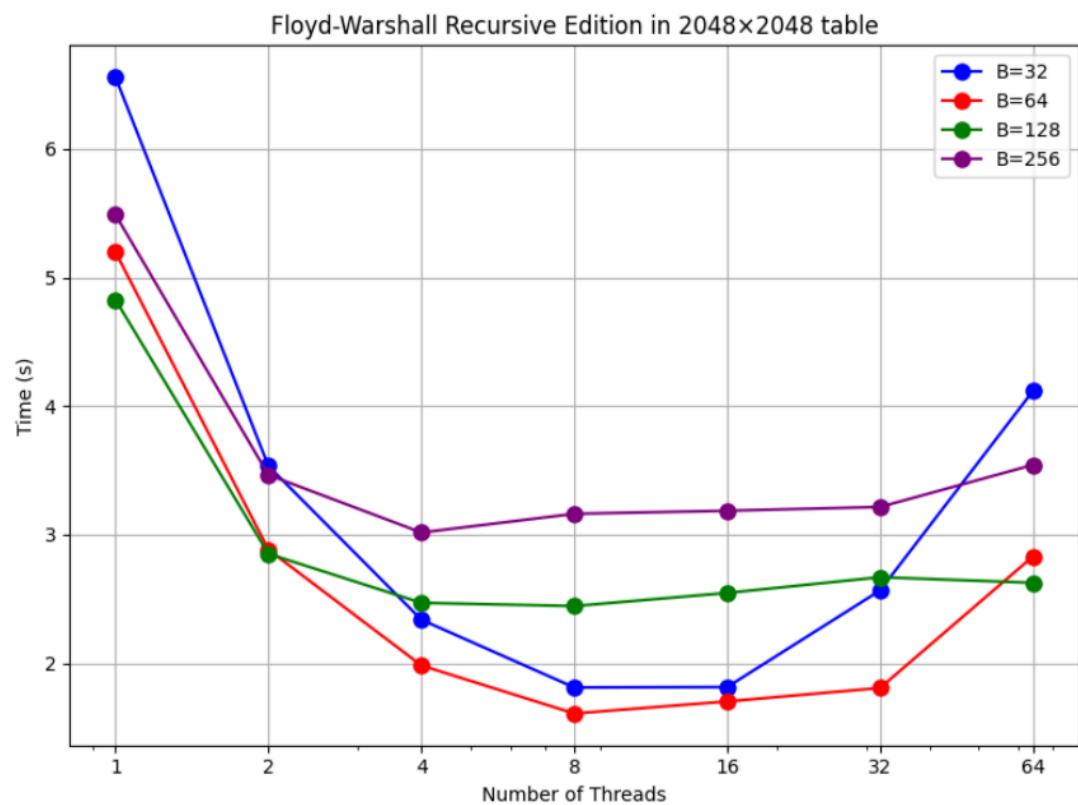
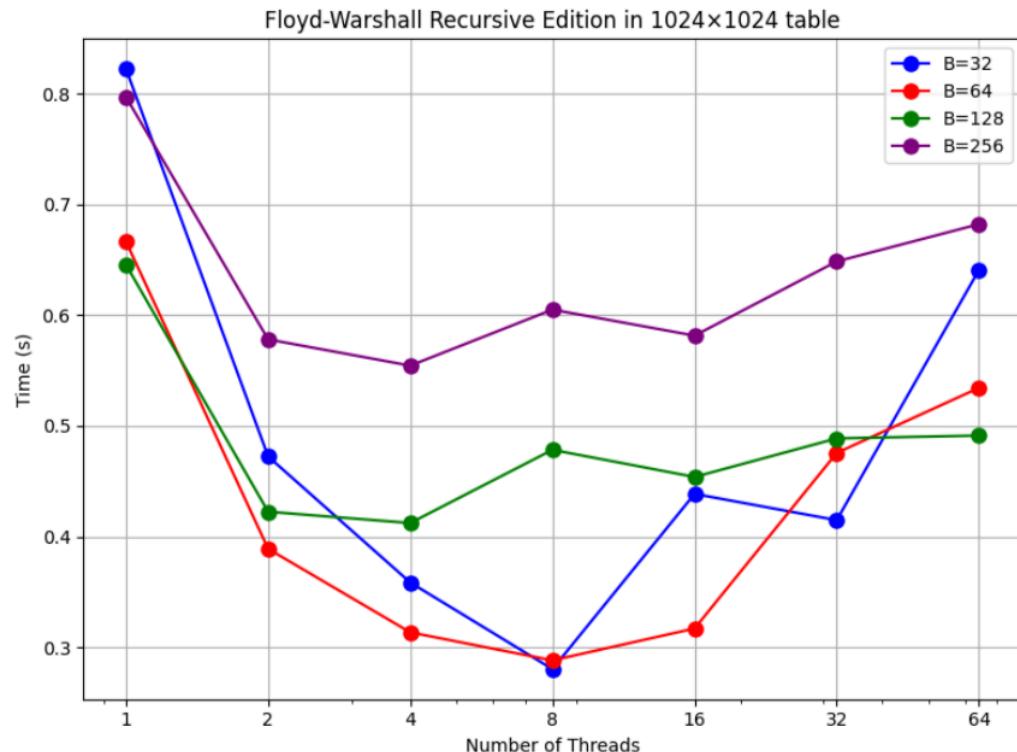
Επιπλέον, παρατηρήσαμε ότι οι αναδρομικές κλήσεις που είναι δυνατόν να παραλληλοποιηθούν είναι οι εξής:

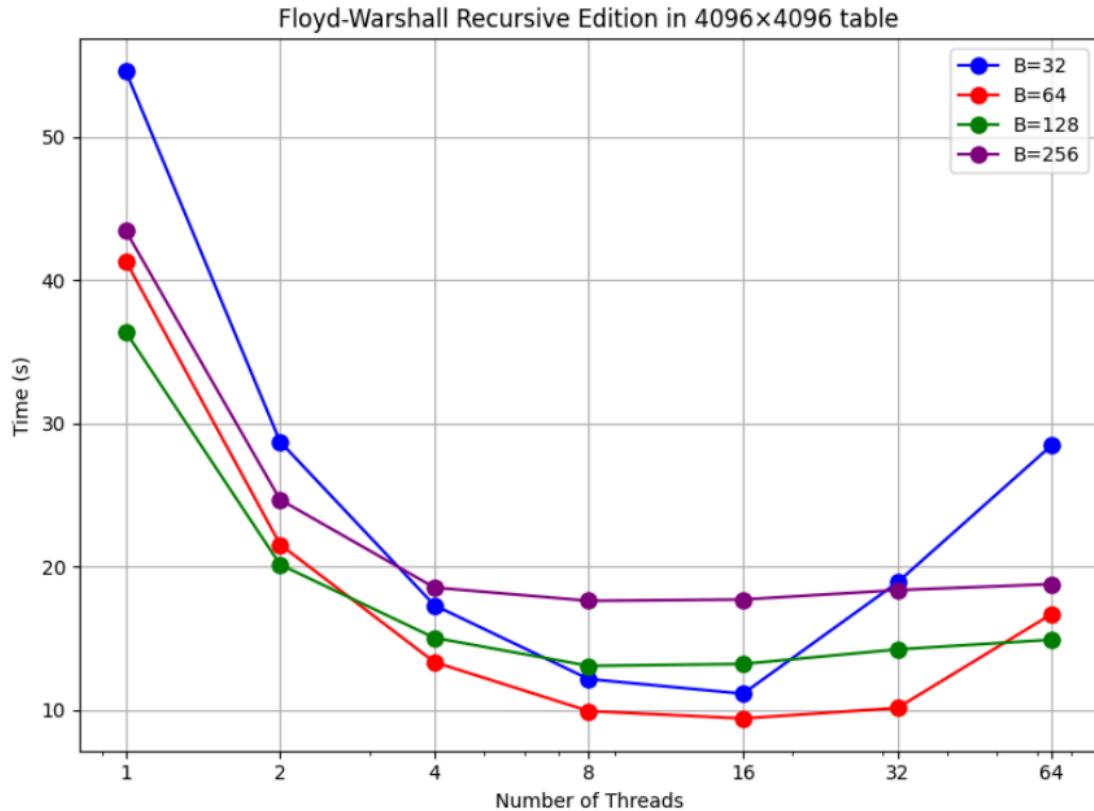
```
FWR (A, B, C)  
if (base case)  
    FWI (A, B, C)  
  
else  
    FWR (A00, B00, C00);      1  
  
    FWR (A01, B00, C01);      2  
    FWR (A10, B10, C00);  
  
    FWR (A11, B10, C01);      3  
    FWR (A11, B10, C01);      4  
  
    FWR (A10, B10, C00);      5  
    FWR (A01, B00, C01);  
  
    FWR (A00, B00, C00);      6
```

δηλαδή η 2η μαζί με την 3η, όπως και η 6η μαζί με την 7η. Μόνο σε αυτές τις περιπτώσεις οι πίνακες είναι ανεξάρτητοι μεταξύ τους, κι επομένως ο υπολογισμός μπορεί να γίνει ταυτόχρονα. Στις υπόλοιπες περιπτώσεις, δεν ισχύει το παραπάνω, κι επομένως τις αφήνουμε να εκτελεστούν σειριακά.

Στον κώδικα, επομένως, προσθέσαμε #pragma omp task shared(A,B,C) πριν την 2η και πριν την 3η (όπως και πριν την 6η και την 7η) αναδρομική κλήση, και μετά από αυτές προσθέσαμε #pragma omp taskwait για να εξασφαλίσουμε ότι τα αποτελέσματα αυτών των υπολογισμών είναι έτοιμα προτού συνεχίσουμε στις επόμενες αναδρομικές κλήσεις.

Plot για το *fw_rc.c* (με παραλληλοποίηση)





Παρατηρούμε ότι η καλύτερη περίπτωση (για το **4096x4096**) είναι για **block size = 64** και για **16 threads**, οπότε και έχουμε χρόνο λίγο κάτω από 10 sec.

ΠΡΟΑΙΡΕΤΙΚΑ - BONUS:

Παραλληλοποιήσαμε και την tiled υλοποίηση του αλγορίθμου Floyd-Warshall.
Αλλαγές που κάναμε στον κώδικα *fw_tiled.c* για την παραλληλοποίηση του :

fw_tiled.c

```
/*
 * Tiled version of the Floyd-Warshall algorithm.
 * command-line arguments: N, B
 * N = size of graph
 * B = size of tile
 * works only when N is a multiple of B
 */
#include <stdio.h>
```

```

#include <stdlib.h>
#include <sys/time.h>
#include "util.h"
#include <omp.h>

inline int min(int a, int b);
inline void FW(int **A, int K, int J, int N);

int main(int argc, char **argv)
{
    int **A;
    int i,j,k;
    struct timeval t1, t2;
    double time;
    int B=64;
    int N=1024;

    if (argc != 3){
        fprintf(stdout, "Usage %s N B\n", argv[0]);
        exit(0);
    }

    N=atoi(argv[1]);
    B=atoi(argv[2]);

    A=(int **)malloc(N*sizeof(int *));
    for(i=0; i<N; i++)A[i]=(int *)malloc(N*sizeof(int));

    graph_init_random(A,-1,N,128*N);

    gettimeofday(&t1,0);

    //Iterates over each diagonal block (tile) of size BxB in the matrix
    #pragma omp parallel shared(A, N, B) private(i, j, k)
    for(k=0;k<N;k+=B){
        //      #pragma omp single{
        FW(A,k,k,B);
        //}

        #pragma omp barrier
        //Updates each tile to the left of the diagonal tile in the current row
        #pragma omp for
        for(i=0; i<k; i+=B){
            FW(A,k,i,k,B); //left
            FW(A,k,k,i,B); //above
        }

        //Updates each tile to the right and below of the diagonal tile in the current row
        #pragma omp for
        for(i=k+B; i<N; i+=B){
            FW(A,k,i,k,B); //right
            FW(A,k,k,i,B); //below
        }
        /*

        //Updates each tile above the diagonal tile in the current column
        for(j=0; j<k; j+=B)
            FW(A,k,k,j,B) #pragma omp for collapse(2);
    }
}

```

```

//Updates each tile below the diagonal tile in the current column
for(j=k+B; j<N; j+=B)
    FW(A,k,k,j,B);
*/

#pragma omp barrier

/*
//Updates each block above and to the left of the diagonal tile
for(i=0; i<k; i+=B)
    for(j=0; j<k; j+=B)
        FW(A,k,i,j,B);

//Updates each block above and to the right of the diagonal tile
for(i=0; i<k; i+=B)
    for(j=k+B; j<N; j+=B)
        FW(A,k,i,j,B);

//Updates each block below and to the left of the diagonal tile
for(i=k+B; i<N; i+=B)
    for(j=0; j<k; j+=B)
        FW(A,k,i,j,B);

//Updates each block below and to the right of the diagonal tile
for(i=k+B; i<N; i+=B)
    for(j=k+B; j<N; j+=B)
        FW(A,k,i,j,B);
*/

#pragma omp for collapse(2)
for (i = 0; i < N; i += B) {
    for (j = 0; j < N; j += B) {
        if (i != k && j != k) {
            FW(A, k, i, j, B); // Update outer tile
        }
    }
}
#pragma omp barrier

}

gettimeofday(&t2,0);

time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)/1000000;
printf("FW_TILED,%d,%d,%4f\n", N,B,time);

/*
for(i=0; i<N; i++)
    for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);

*/
return 0;
}

inline int min(int a, int b)
{
    if(a<=b) return a;
    else return b;
}

```

```
inline void FW(int **A, int K, int I, int J, int N)
{
    int i,j,k;

    for(k=K; k<K+N; k++)
        for(i=I; i<I+N; i++)
            for(j=J; j<J+N; j++)
                A[i][j]=min(A[i][j], A[i][k]+A[k][j]);
}
```

Επεξήγηση αλλαγών:

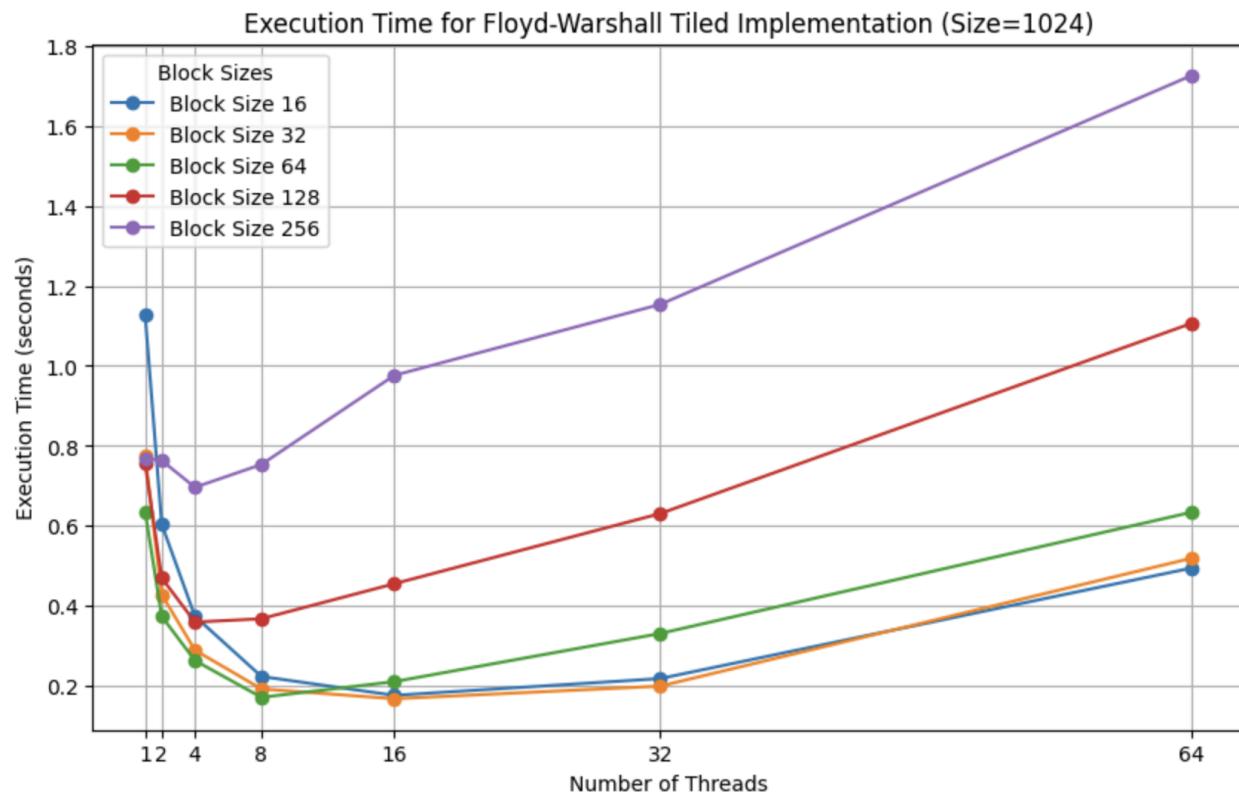
Για τα tiles που βρίσκονται πάνω και αριστερά από το διαγώνιο tile, η ενημέρωση γίνεται μέσω ενός for loop. Τα συγκεκριμένα tiles, όμως, μπορούν να παραλληλοποιηθούν και να υπολογιστούν ταυτόχρονα. Το ίδιο συμβαίνει και με τα κάτω και δεξιά tiles. Επίσης, με τη χρήση του #pragma omp for collapse(2) επιτρέπουμε στα νήματα να εκτελούν ανεξάρτητα την ενημέρωση των εξωτερικών tiles (αυτά που ενημερώνονται στο τρίτο phase). Τέλος, χρησιμοποιώντας #pragma omp barrier διασφαλίζουμε ότι όλες οι ενημερώσεις έχουν ολοκληρωθεί προτού τα νήματα προχωρήσουν στην επόμενη φάση.

Για να σιγουρευτούμε ότι το πρόγραμμά μας κάνει σωστά την παραλληλοποίηση, τρέξαμε το αρχικό fw_tiled (δεξιά στήλη) και την δικιά μας παραλληλοποιημένη έκδοση (αριστερή στήλη) για μικρό μέγεθος πίνακα (16x16) και μικρό μέγεθος tile (4x4) , και παρατηρήσαμε τα αποτελέσματα για τυχόν διαφορές λόγω πιθανών race conditions:

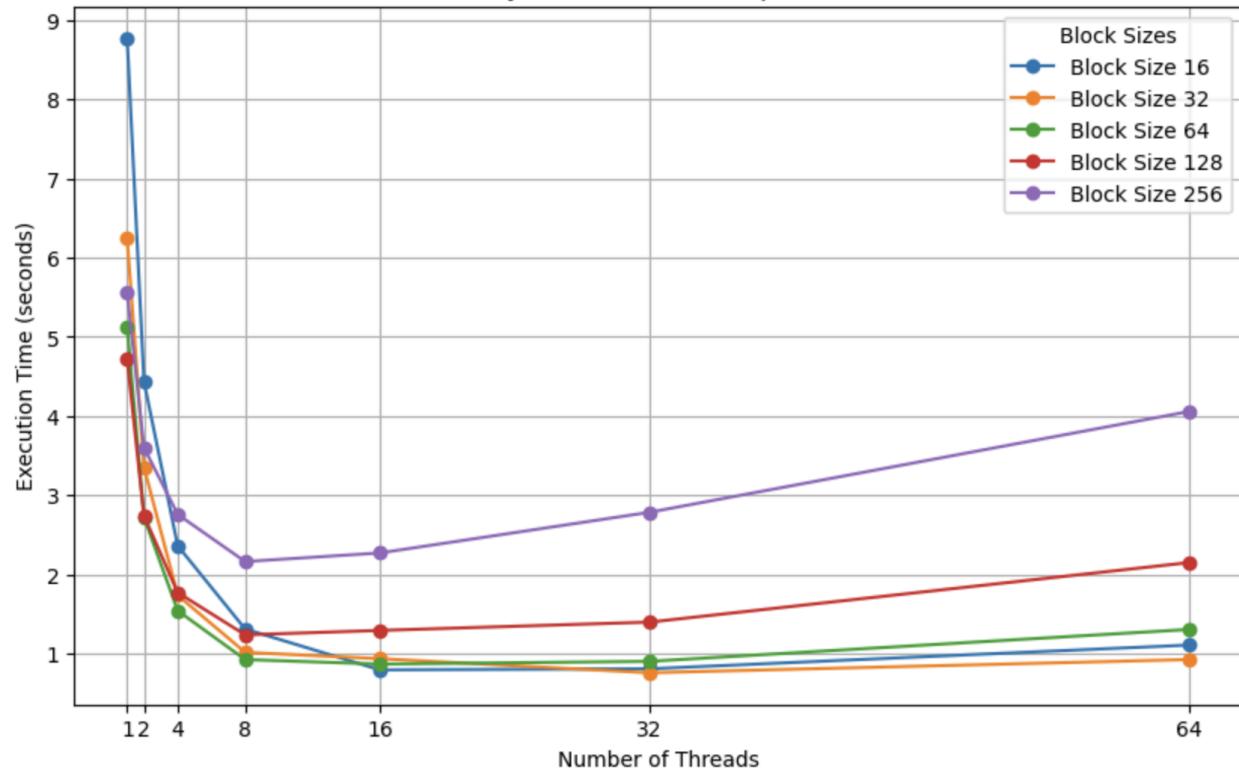
parlab38@scirouter: ~/a2/FW	parlab38@scirouter: ~/a2/FW
Running with 4 threads	Running with 1 threads
FW_TILED,16,4,0.0002	FW_TILED,16,4,0.0000
0	0
68956	68956
34224	34224
182105	182105
108483	108483
15862	15862
46055	46055
60446	60446
384693	384693
354851	354851
129460	129460
218436	218436
98471	98471
197871	197871
101154	101154
222006	222006
92454	92454
0	0
126678	126678
274559	274559
87443	87443
108316	108316
138509	138509
143309	143309
477147	477147
364313	364313
212323	212323
181660	181660
190925	190925
290325	290325
193608	193608
314460	314460
127186	127186
34732	34732
0	0
251740	251740
122175	122175
143048	143048
173241	173241
178041	178041
377297	377297
320627	320627
247055	247055
216392	216392
225657	225657
270655	270655
66930	66930
349192	349192
243582	243582
155651	155651
120919	120919
0	0
238571	238571
145950	145950
64003	64003
208160	208160
205737	205737
334559	334559
277174	277174

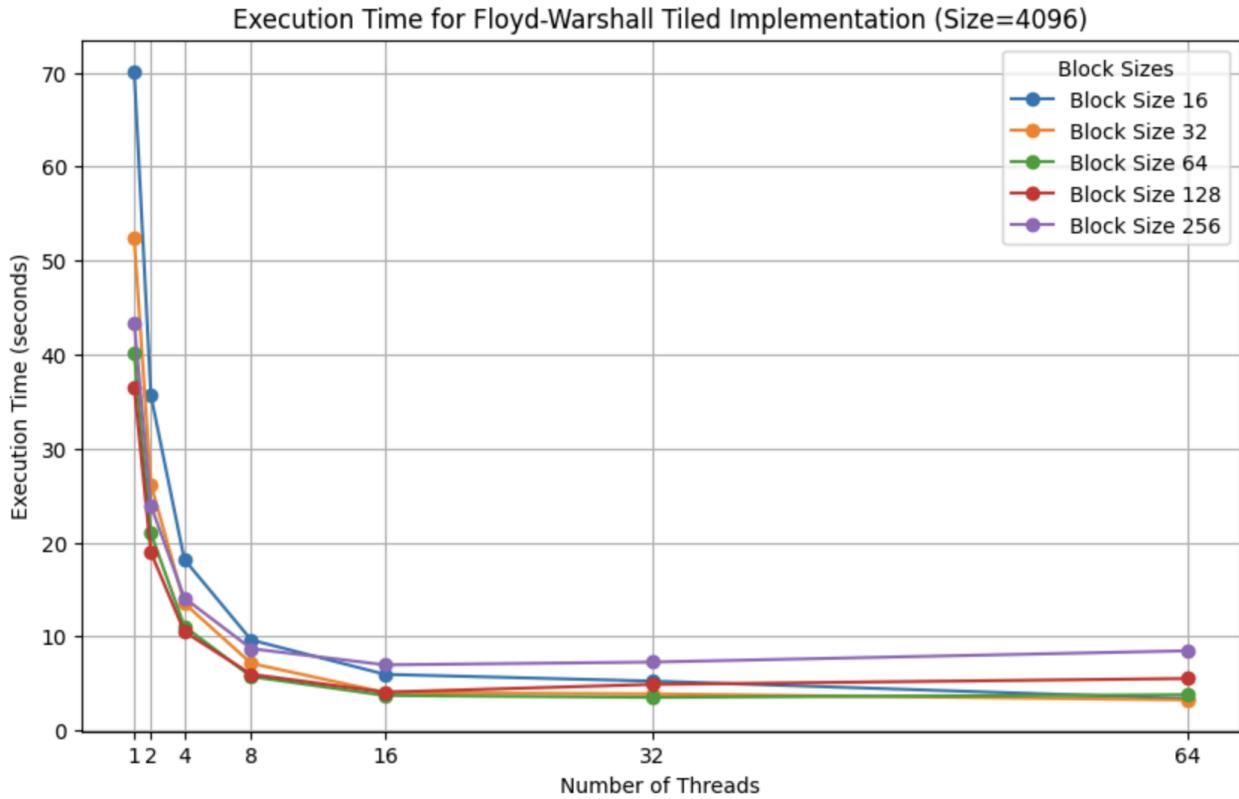
Τα αποτελέσματα, είναι πανομοιότυπα.

Plot via to *fw_tiled.c*



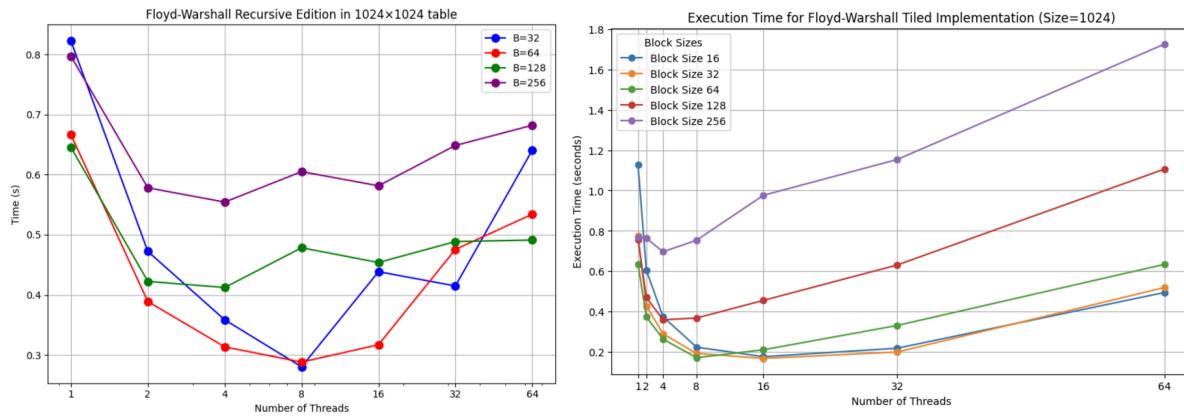
Execution Time for Floyd-Warshall Tiled Implementation (Size=2048)





Παρακάτω τοποθετούμε τα διαγράμματα της **recursive** υλοποίησης (αριστερά) και της **tiled** υλοποίησης (δεξιά) δίπλα δίπλα για να δούμε καλύτερα τις διαφορές:

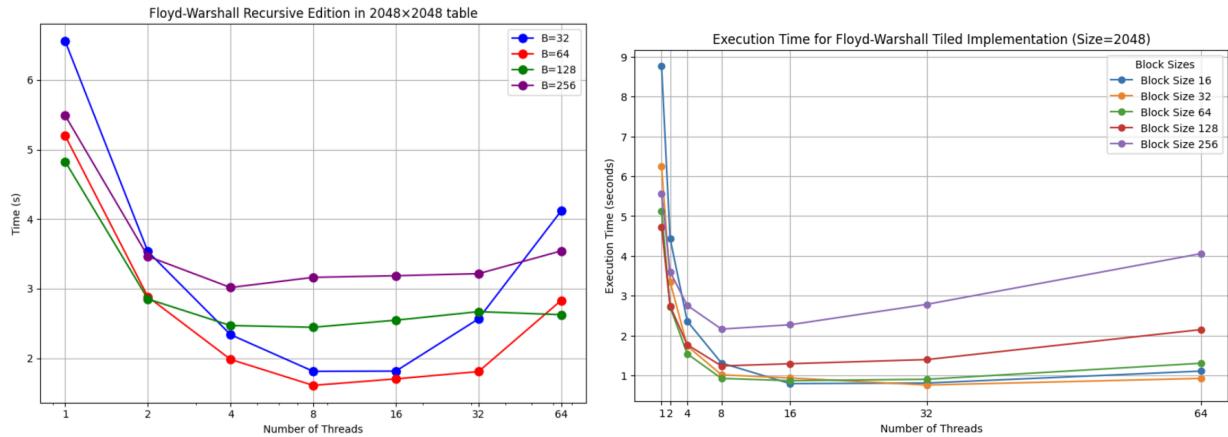
Για Size = 1024:



Εδώ παρατηρούμε πως στην recursive υλοποίηση οι καλύτεροι (μικρότεροι) χρόνοι παρουσιάζονται για $nthreads = 8$ στη γενική περίπτωση κι έπειτα χειροτερεύουν.

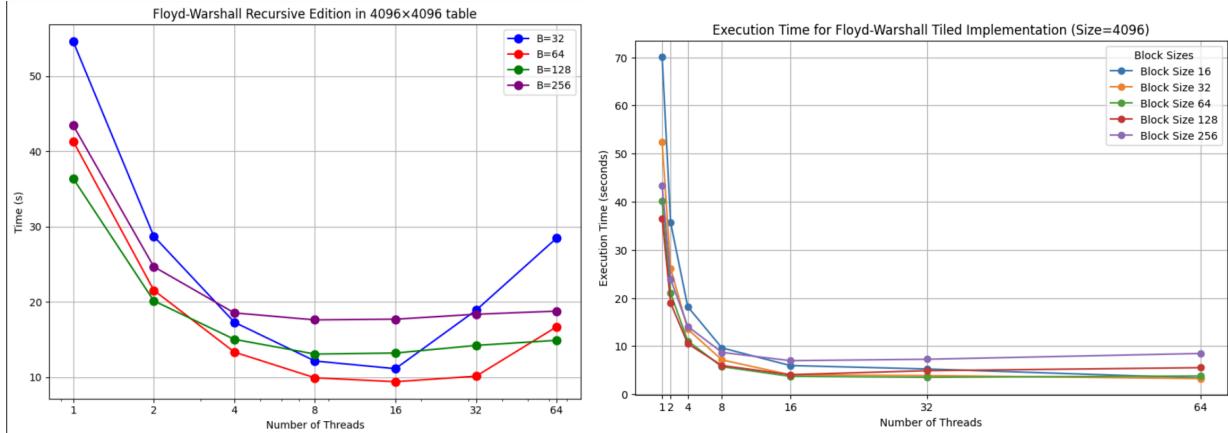
Επίσης, παρατηρούμε πως στην tiled υλοποίηση, για μεγάλα tile sizes, ο χρόνος χειροτερεύει (αυξάνεται) σημαντικά με την αύξηση των threads.

Για Size = 2048:



Εδώ παρατηρούμε μια πιο μεγάλη σταθερότητα στα αποτελέσματα σε σχέση με προηγουμένως, τόσο στην recursive υλοποίηση όσο και στην tiled. Επίσης, για μια ακόμη φορά, η tiled υλοποίηση έχει σημαντικά καλύτερους χρόνους από ό,τι η recursive.

Για Size = 4096:



Εδώ παρατηρούμε για μια ακόμη φορά πιο σταθερά αποτελέσματα και στις δύο υλοποιήσεις. Επίσης, παρατηρούμε **πολλαπλάσια καλύτερους (μικρότερους) χρόνους** στην περίπτωση της **tiled** υλοποίησης.

ΑΣΚΗΣΗ 2.1 - Αμοιβαίος Αποκλεισμός - Κλειδώματα

Παρακάτω φαίνονται οι μετρήσεις για κάθε είδος lock με το configuration:

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

Τα locks που χρησιμοποιήθηκαν είναι:

- nosync_lock
- pthread_mutex_lock
- pthread_spin_lock
- tas_lock
- ttas_lock
- array_lock
- clh_lock

Τα διάφορα κλειδώματα χρησιμοποιήθηκαν στο αρχείο **omp_lock_kmeans.c** όπου τρέξαμε τον αλγόριθμο kmeans. Τα locks προστατεύουν τα κρίσιμα σημεία του κώδικα για την αποφυγή race conditions.

Συγκρίνουμε την απόδοση μεταξύ της χρήσης του **OpenMP directive** (#pragma omp critical) και των **Locks** για να βρούμε την καλύτερη μέθοδο προστασίας του κρίσιμου τμήματος του kmeans.

Τροποποίηση του **script run_on_queue.sh** για την εκτέλεση του kmeans με διαφορετικά lock_mechanisms και με συγκεκριμένα configurations {Size, Coords, Clusters, Loops}.

```
#!/bin/bash

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

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

## How many machines should we get?
#PBS -l nodes=1:ppn=64

## How long should the job run for?
#PBS -l walltime=00:30:00

## Start
## Load OpenMP module and navigate to the directory
module load openmp
cd /home/parallel/parlab38/a2/a2_1

## List of lock mechanisms
lock_mechanisms=(
    "kmeans_omp_nosync_lock"
    "kmeans_omp_pthread_mutex_lock"
    "kmeans_omp_pthread_spin_lock"
    "kmeans_omp_tas_lock"
    "kmeans_omp_ttas_lock"
```

```

    "kmeans_omp_array_lock"
    "kmeans_omp_clh_lock"
)

## Loop through lock mechanisms and threads
for lock in "${lock_mechanisms[@]}"
do
    # Create unique output and error files for each mechanism
    out_file="run_${lock}.out"
    err_file="run_${lock}.err"

    for threads in 1 2 4 8 16 32 64
    do
        export OMP_NUM_THREADS=$threads
        export GOMP_CPU_AFFINITY="0-$((threads - 1))"
        echo "Running $lock with $threads threads and CPU affinity set to $GOMP_CPU_AFFINITY" >> $out_file
        ./$lock -s 32 -n 16 -c 32 -l 10 >> $out_file 2>> $err_file
    done
done
done

```

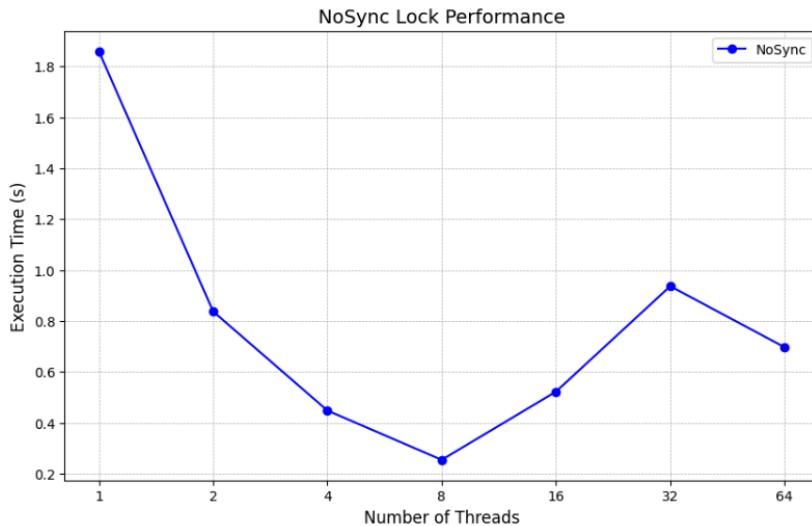
Τα αρχεία που προέκυψαν αφού τα τρέξαμε στον sandman:

```

parlab38@scirouter:~/a2/a2_1$ ls
file_io.c           main.c           run_kmeans.err
file_io.o           main.o           run_kmeans_general.err
kmeans.h            Makefile         run_kmeans_general.out
kmeans_omp_array_lock Makefile.old   run_kmeans_omp_array_lock.err
kmeans_omp_clh_lock make_kmeans.err run_kmeans_omp_array_lock.out
kmeans_omp_critical make_kmeans.out run_kmeans_omp_clh_lock.err
kmeans_omp_naive   make_on_queue.sh run_kmeans_omp_clh_lock.out
kmeans_omp_nosync_lock omp_critical_kmeans.c run_kmeans_omp_nosync_lock.err
kmeans_omp_pthread_mutex_lock  omp_critical_kmeans.o run_kmeans_omp_nosync_lock.out
kmeans_omp_pthread_spin_lock   omp_lock_kmeans.c  run_kmeans_omp_pthread_mutex_lock.err
kmeans_omp_tas_lock          omp_lock_kmeans.o  run_kmeans_omp_pthread_mutex_lock.out
kmeans_omp_ttas_lock         omp_naive_kmeans.c run_kmeans_omp_pthread_spin_lock.err
locks                  omp_naive_kmeans.o  run_kmeans_omp_pthread_spin_lock.out
run_kmeans_omp_tas_lock.err
run_kmeans_omp_tas_lock.out
run_kmeans_omp_ttas_lock.err
run_kmeans_omp_ttas_lock.out
run_kmeans.out
run_on_queue2.sh
run_on_queue.sh
seq_kmeans.c
util.c
util.o

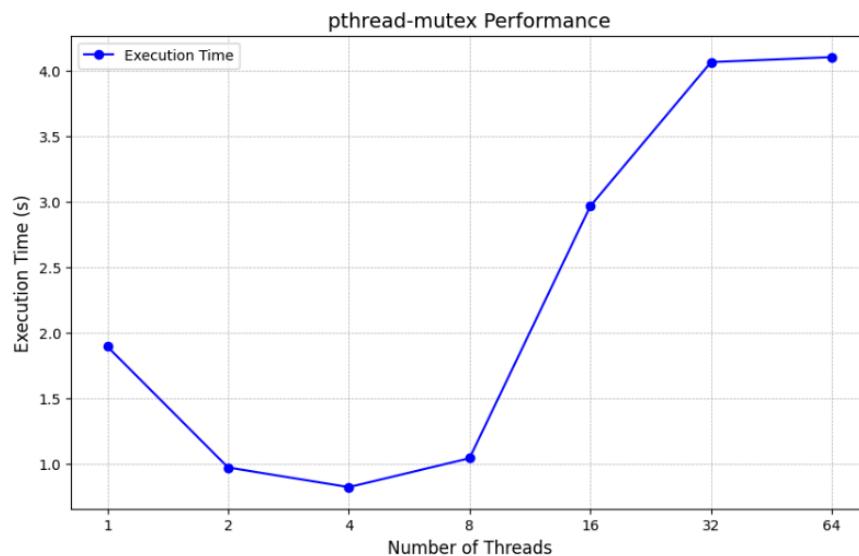
```

Nosync lock:



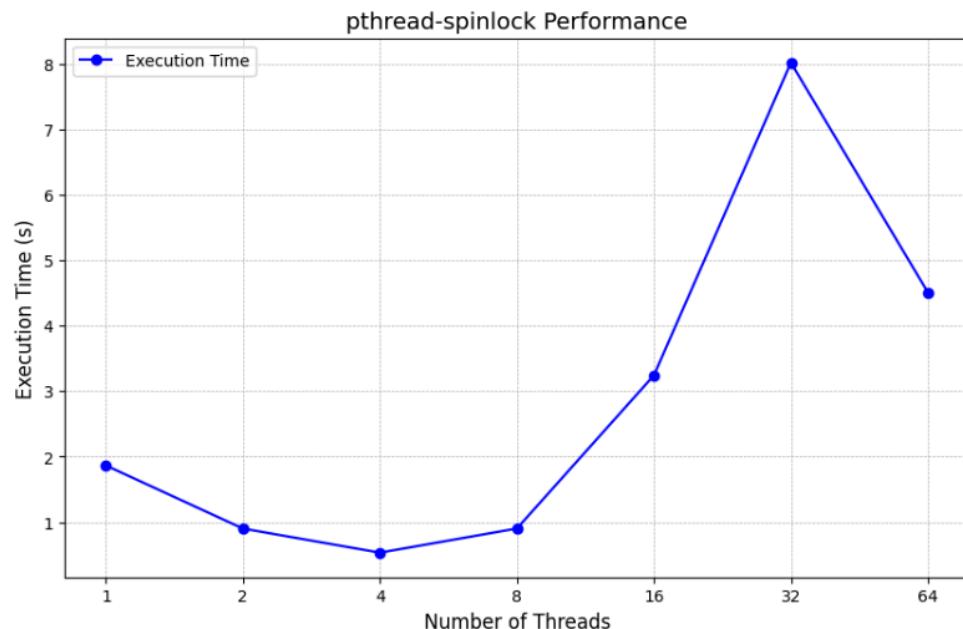
Το Nosync δεν χρησιμοποιεί κανένα συγχρονισμό για την προστασία του συγκεκριμένου τμήματος, οπότε είναι και πιο γρήγορο όταν δεν υπάρχει ανάγκη συγχρονισμού.

Pthread Mutex Lock:



Το Mutex (Mutual Exclusion) εξασφαλίζει ότι μόνο ένα νήμα μπορεί να αποκτήσει πρόσβαση σε έναν κοινό τρήμα ανά πάσα στιγμή. Όταν ένα νημά αποκτά το mutex, τα υπόλοιπα νημάτα πρέπει να περιμένουν μέχρι το mutex να απελευθερωθεί. Με αυτό τον τρόπο αποφεύγονται τα race conditions.

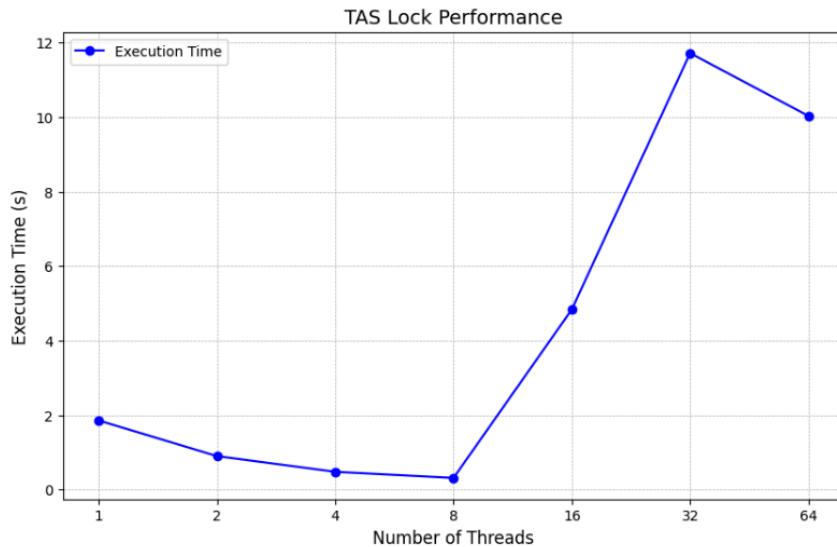
Pthread Spin Lock:



Το spinlock είναι ένα τύπος κλειδώματος όπου το νήμα που προσπαθεί να αποκτήσει το κλείδωμα "γυρίζει" (spin - συνήθως με while loop) συνεχώς και ελέγχει αν το κλείδωμα είναι ελεύθερο. Όταν το κλείδωμα είναι ελεύθερο, το νημά το αποκτά. Αυτή η προσέγγιση μπορεί να είναι πολύ γρήγορη αν οι συγκρούσεις είναι σπάνιες, αλλά αν υπάρχουν πολλές συγκρούσεις, τα

νήματα που περιστρέφονται συνεχώς καταναλώνουν CPU χωρίς να προχωρούν στην εκτέλεση, κάτι που μπορεί να οδηγήσει σε σπασάλη πόρων.

TAS Lock:

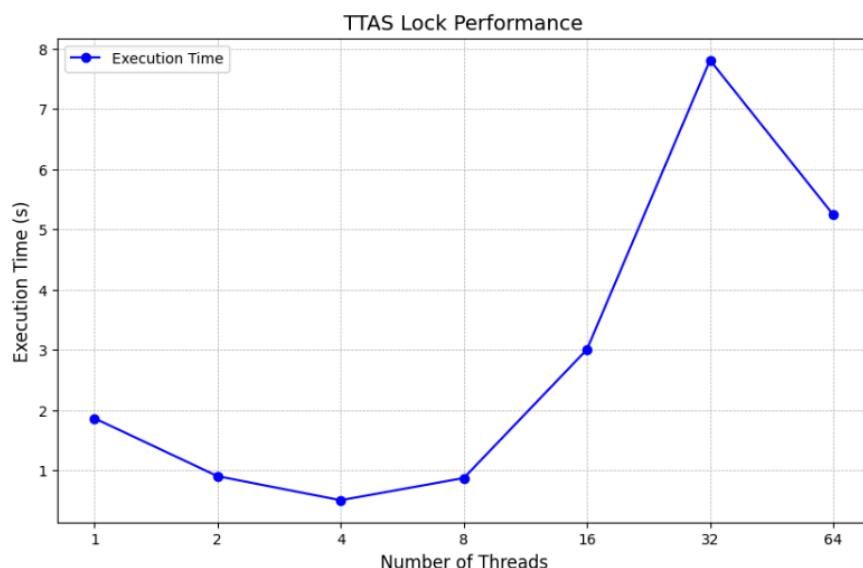


Το TAS Lock βασίζεται στη λειτουργία test and set.

Όταν ένα νήμα θέλει να αποκτήσει το lock, αρχικά ελέγχει αν η μεταβλητή του κλειδώματος είναι ελεύθερη (0) ή κατειλημμένη (1). Αν είναι ελεύθερη, την θέτει σε κατειλημμένη (0) και αποκτά πρόσβαση.

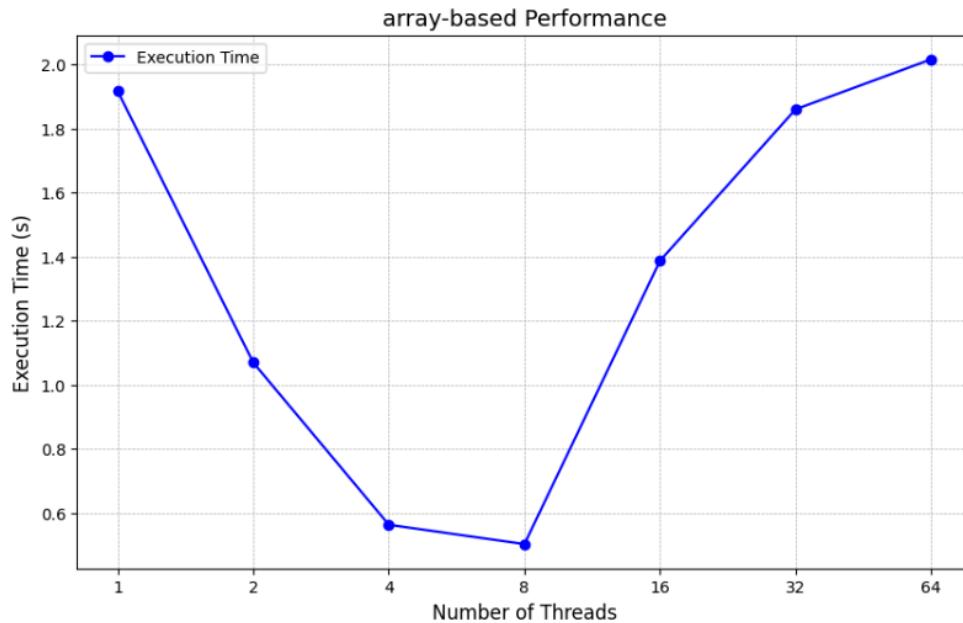
Αν είναι ήδη κατειλημμένη, το νήμα συνεχίζει να εκτελεί την ίδια λειτουργία, μέχρι να γίνει unlock. Έτσι, όταν έχουμε μεγάλο ανταγωνισμό δεν έχουμε καλή απόδοση, καθώς πραγματοποιούνται διαρκώς έλεγχοι από όλα τα νήματα.

TTAS Lock:



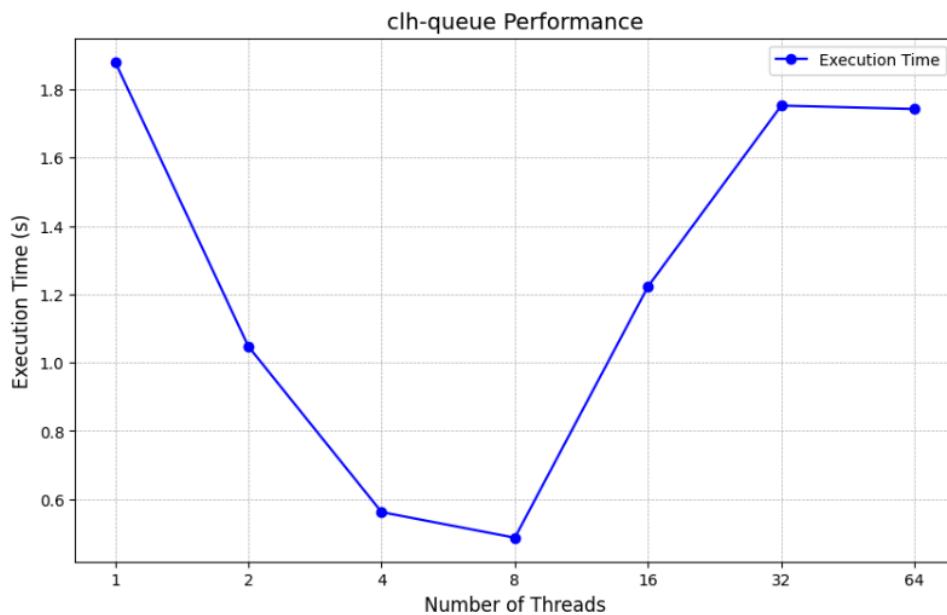
To TTAS Lock είναι μια παραλλαγή του TAS Lock, όπου το νήμα διαβάζει την κατάσταση του lock και έπειτα εκτελεί το test and set. Το γεγονός αυτό το καθιστά πιο αποδοτικό από το TAS.

Array Lock:



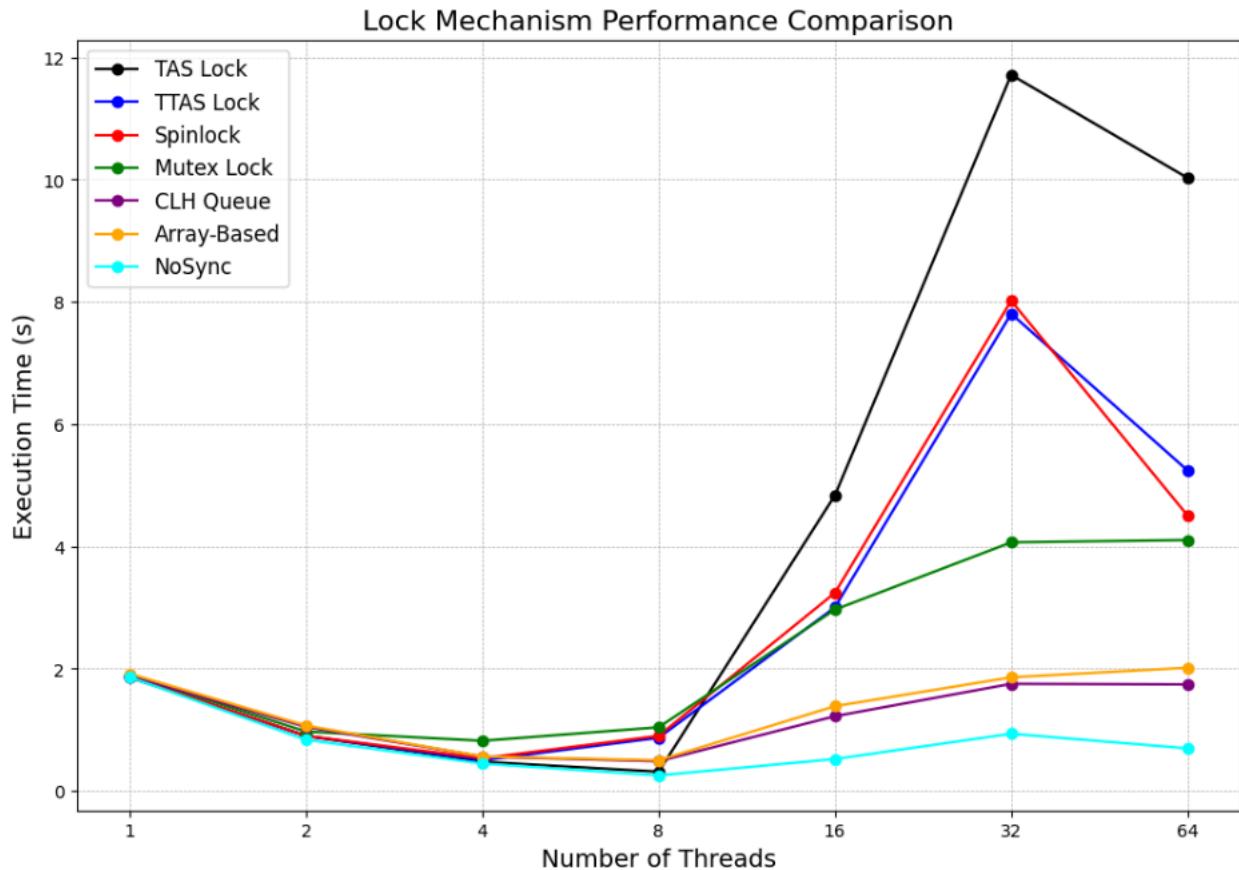
Το Array lock χρησιμοποιεί έναν πίνακα για να οργανώσει τη διαχείριση των νημάτων που προσπαθούν να αποκτήσουν το lock. Είναι πιο αποδοτικό από άλλα lock mechanisms για περιπτώσεις με πολλές ομάδες νημάτων, ωστόσο απαιτεί περισσότερη μνήμη για να αποθηκεύσει τις πληροφορίες των νημάτων και τη διαχείριση των locks.

CLH Lock:



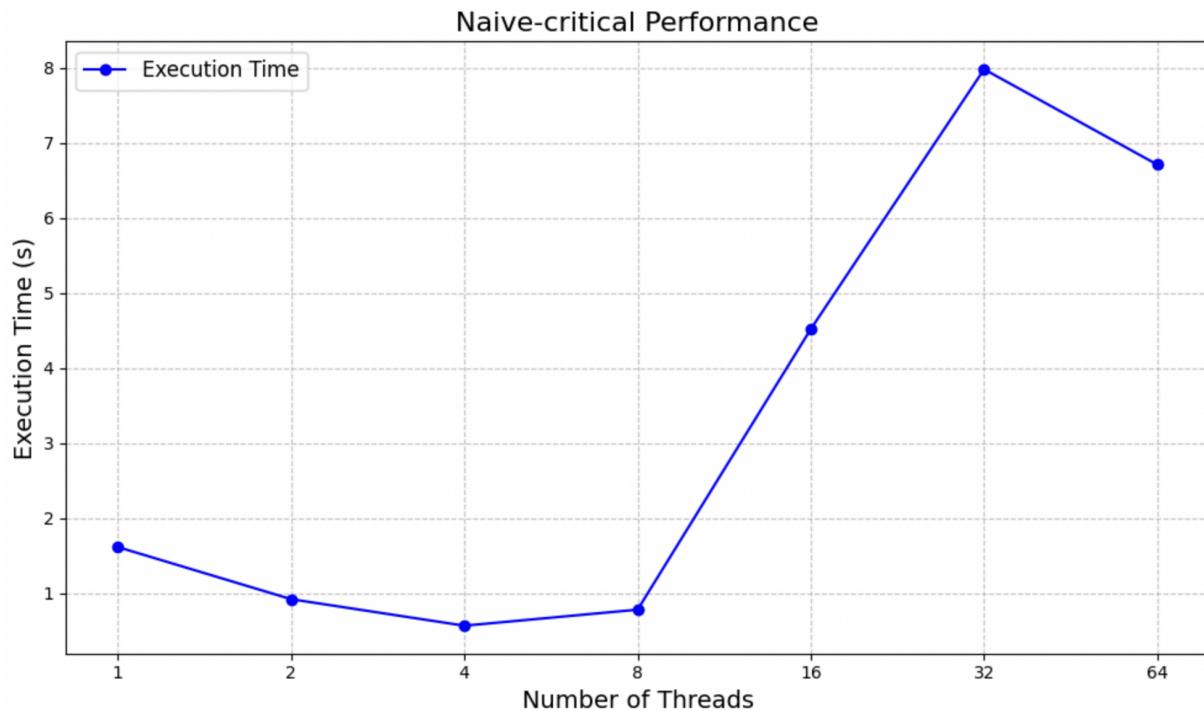
Το CLH lock χρησιμοποιεί μία ουρά για να διαχειριστεί τα νήματα που προσπαθούν να αποκτήσουν το lock. Κάθε νήμα δημιουργεί ένα κόμβο στην ουρά και περιμένει τη σειρά του για να αποκτήσει το κλείδωμα. Το μεγάλο πλεονέκτημα αυτού του τύπου είναι ότι τα νήματα που περιμένουν δεν καταναλώνουν CPU (δεν "γυρίζουν" συνεχώς), με αποτέλεσμα μεγαλύτερη αποδοτικότητα όταν υπάρχει αυξημένη κίνηση.

Σύγκριση των διαφόρων μηχανισμών lock:



Παρατηρούμε από το παραπάνω συγκεντρωτικό διάγραμμα ότι την καλύτερη επίδοση την έχει το **NoSync**, το οποίο όμως δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα. Χρησιμοποιούμε το NoSync σαν άνω όριο για την αξιολόγηση των υπόλοιπων locks.

Συνεπώς, το καλύτερο lock που παρέχει σωστά αποτελέσματα είναι το **CLH Lock** του οποίου η υλοποίηση στηρίζεται στη χρήση μιας συνδεδεμένης λίστας.



Παραπάνω παρατηρούμε, επίσης, τα αποτελέσματα για την υλοποίηση με το OpenMP directive `#pragma omp critical` για την προστασία του κρίσιμου τμήματος. Ο καλύτερος χρόνος (χαμηλότερος) με αυτή την υλοποίηση είναι $t = 0.5703\text{s}$ για `threads = 4`.

ΑΣΚΗΣΗ 2.3 - Ταυτόχρονες Δομές Δεδομένων

Σε αυτό το ερώτημα κληθήκαμε να συγκρίνουμε την επίδοση των παρακάτω ταυτόχρονων υλοποιήσεων μιας απλά συνδεδεμένης ταξινομημένης λίστας:

- Coarse-grain locking
- Fine-grain locking
- Optimistic synchronization
- Lazy synchronization
- Non-blocking synchronization
- Serial implementation

Με την εντολή **make** δημιουργούμε 6 εκτελέσιμα αρχεία που υλοποιούν τα παραπάνω.

Με τη δημιουργία ενός script **run_on_queue.sh** χρησιμοποιούμε κάθε ένα από τα 6 εκτελέσιμα που έχουμε και βάζουμε 4 ορίσματα στο καθένα:

Το εύρος των κλειδιών που θα χρησιμοποιηθούν από τα νήματα,

Το ποσοστό των λειτουργιών που θα είναι αναζητήσεις (**contains()**), εισαγωγές (**add()**) και διαγραφές (**remove()**).

Διαφορετικές τιμές για τις παραμέτρους των εκτελέσιμων:

- Αριθμός νημάτων: 1, 2, 4, 8, 16, 32, 64, 128 **
- Μέγεθος λίστας: 1024, 8192
- Ποσοστό λειτουργιών: 100-0-0, 80-10-10, 20-40-40, 0-50-50. Ο πρώτος αριθμός υποδηλώνει το ποσοστό αναζητήσεων και οι επόμενοι δύο το ποσοστό εισαγωγών και διαγραφών αντίστοιχα.

(** για την περίπτωση 64 threads έχουμε χρησιμοποιήσει hyperthreading, ενώ για την περίπτωση 128 έχουμε χρησιμοποιήσει oversubscription.)

run_on_queue.sh

```
#!/bin/bash

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

## How many machines should we get?
#PBS -l nodes=1:ppn=128

## How long should the job run for?
#PBS -l walltime=02:00:00

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

module load openmp
```

```

cd /home/parallel/parlab38/a2/a2_3/conc_ll

# Define the parameters for the experiments

# Number of threads
THREAD_COUNTS=(1 2 4 8 16 32 64 128)

# List sizes
LIST_SIZES=(1024 8192)

# Operation ratios: search-insert-delete
RATION_RATIOS=("100 0 0" "80 10 10" "20 40 40" "0 50 50")

EXECUTABLES=("x.cgi" "x.fgl" "x.nb" "x.lazy" "x.opt")

# Iterate over all executables
for executable in "${EXECUTABLES[@]}"
do
    # Define an output file for each executable
    output_file="${executable}_output.out"

    # Iterate over all combinations of list size and operation ratio
    for list_size in "${LIST_SIZES[@]}"
    do
        for ratio in "${RATION_RATIOS[@]}"
        do
            for threads in "${THREAD_COUNTS[@]}"
            do
                # Define MT_CONF based on the number of threads
                if [ "$threads" -le 32 ]; then
                    export MT_CONF=$((seq -s, 0 $((threads - 1))))
                elif [ "$threads" -le 64 ]; then
                    export MT_CONF=$((seq -s, 0 31),$(seq -s, 0 $((threads - 33))))
                else
                    export MT_CONF=$((seq -s, 0 31),$(seq -s, 0 31),$(seq -s, 0 31),$(seq -s, 0 31))
                fi

                # Extract the operation ratios
                read -r search insert delete <<< "$ratio"

                # Log the current configuration to the output file for each executable
                echo "Running $executable with $threads threads, list size $list_size, ratio $ratio" >>
$output_file

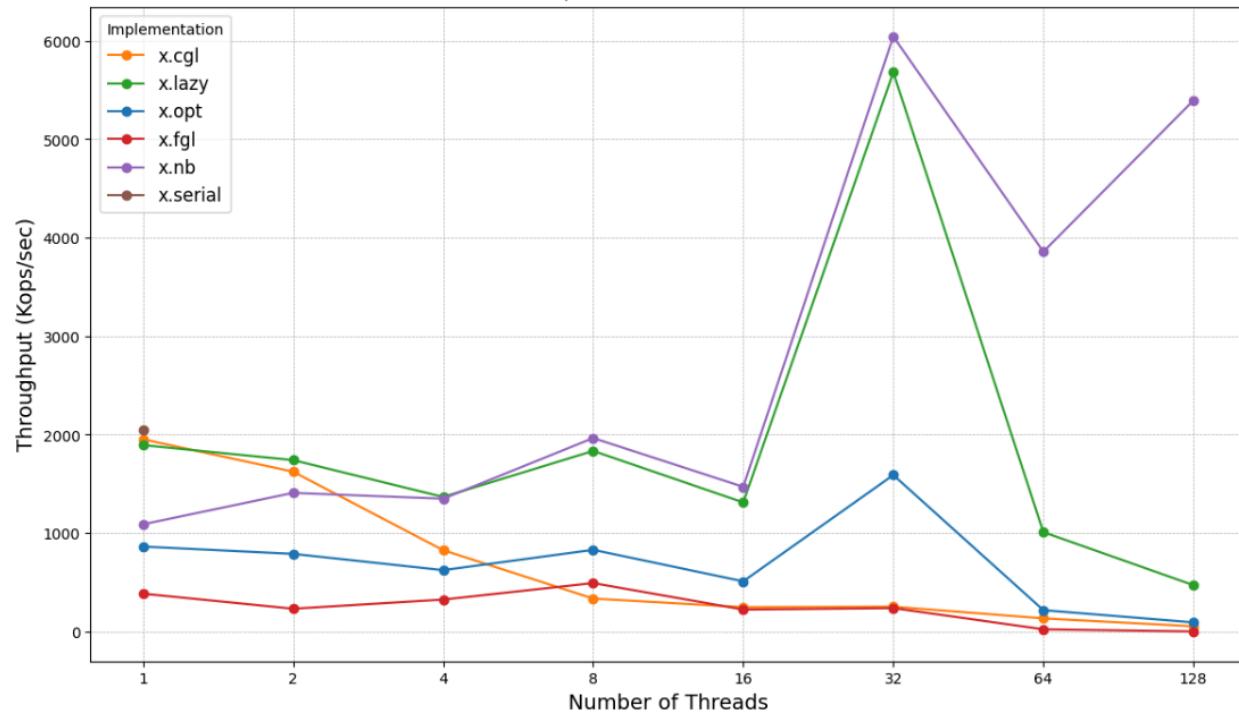
                # Ensure the arguments are passed correctly to the program
                ./"$executable" $list_size $search $insert $delete >> $output_file 2>&1
            done
        done
    done
done

```

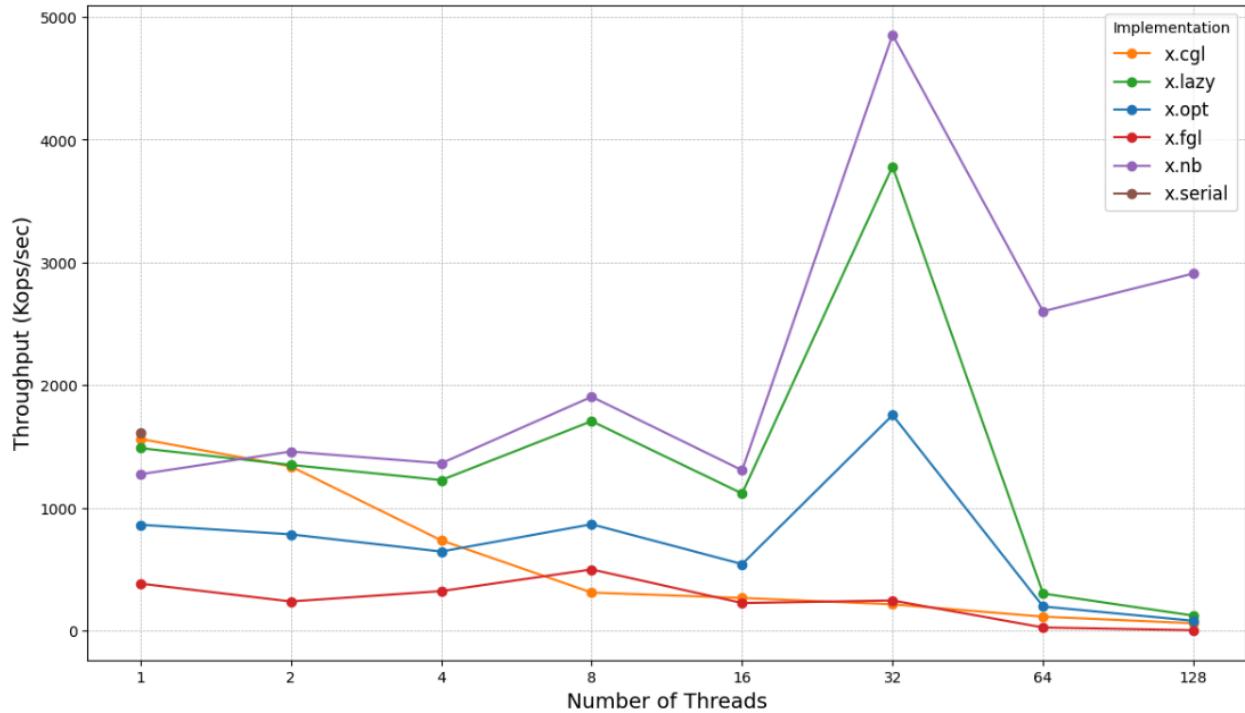
For List Size = 1024:



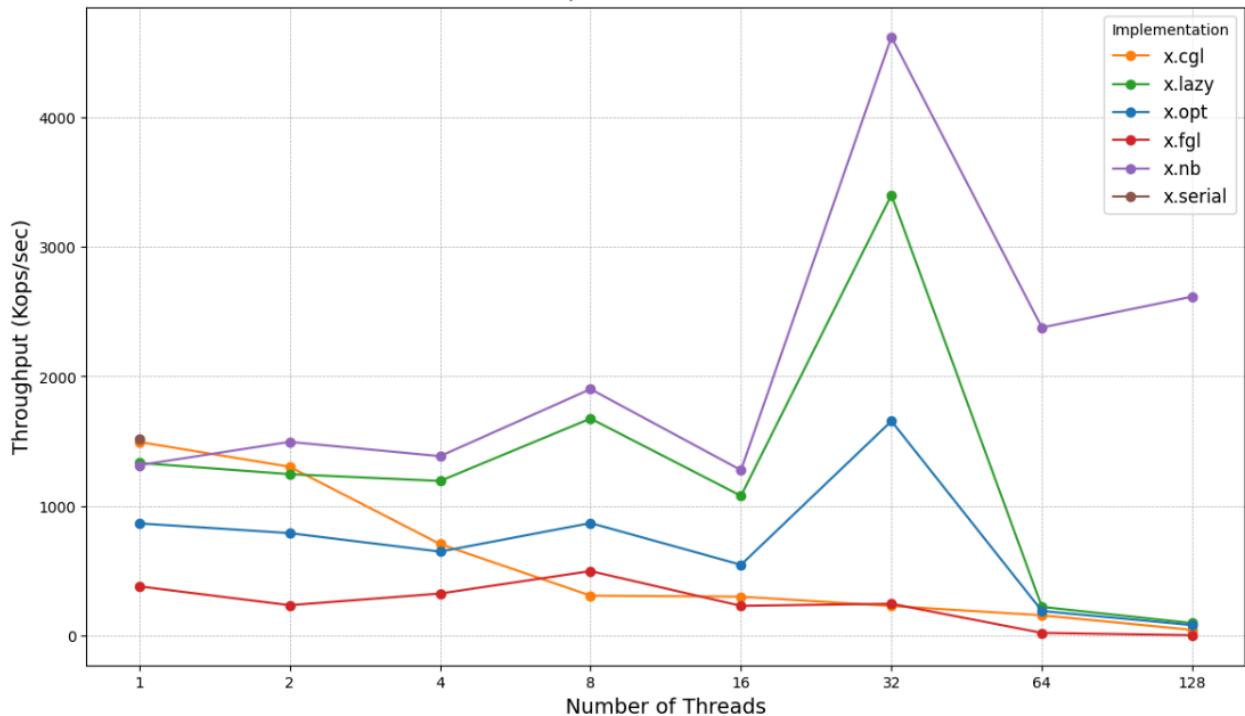
List Size = 1024, Search 80 - Insert 10 - Delete 10



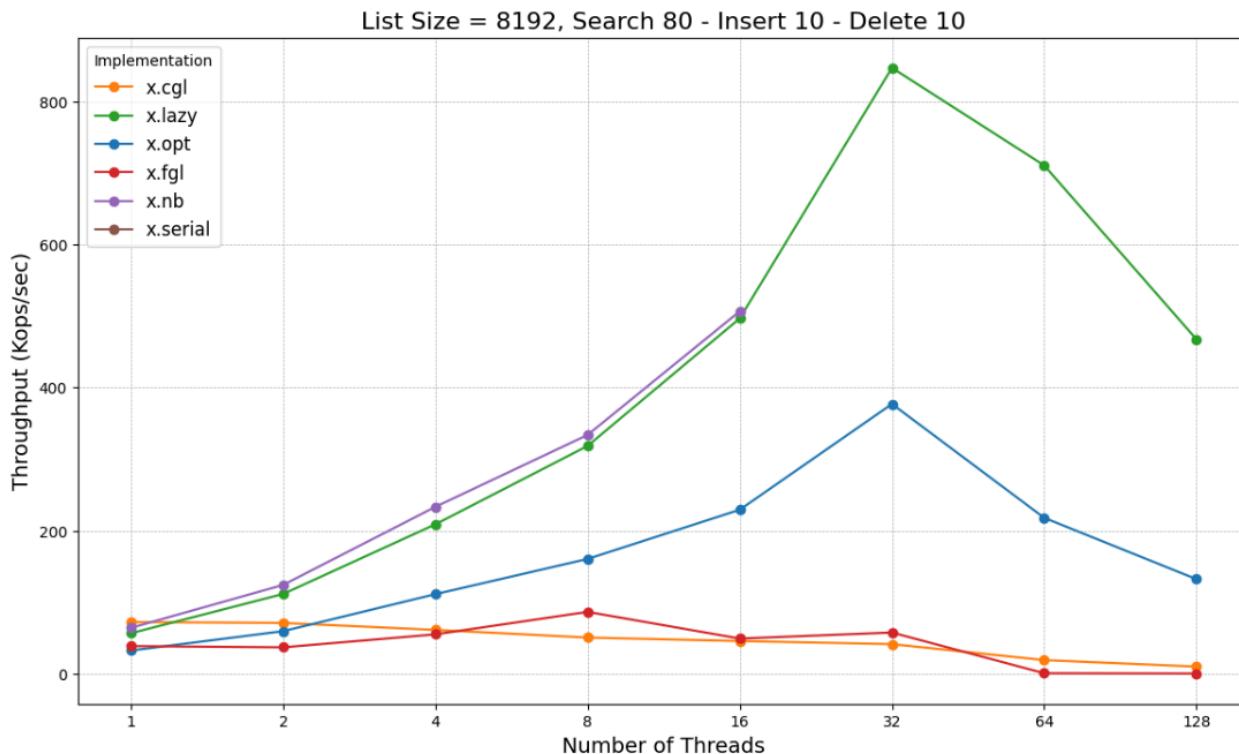
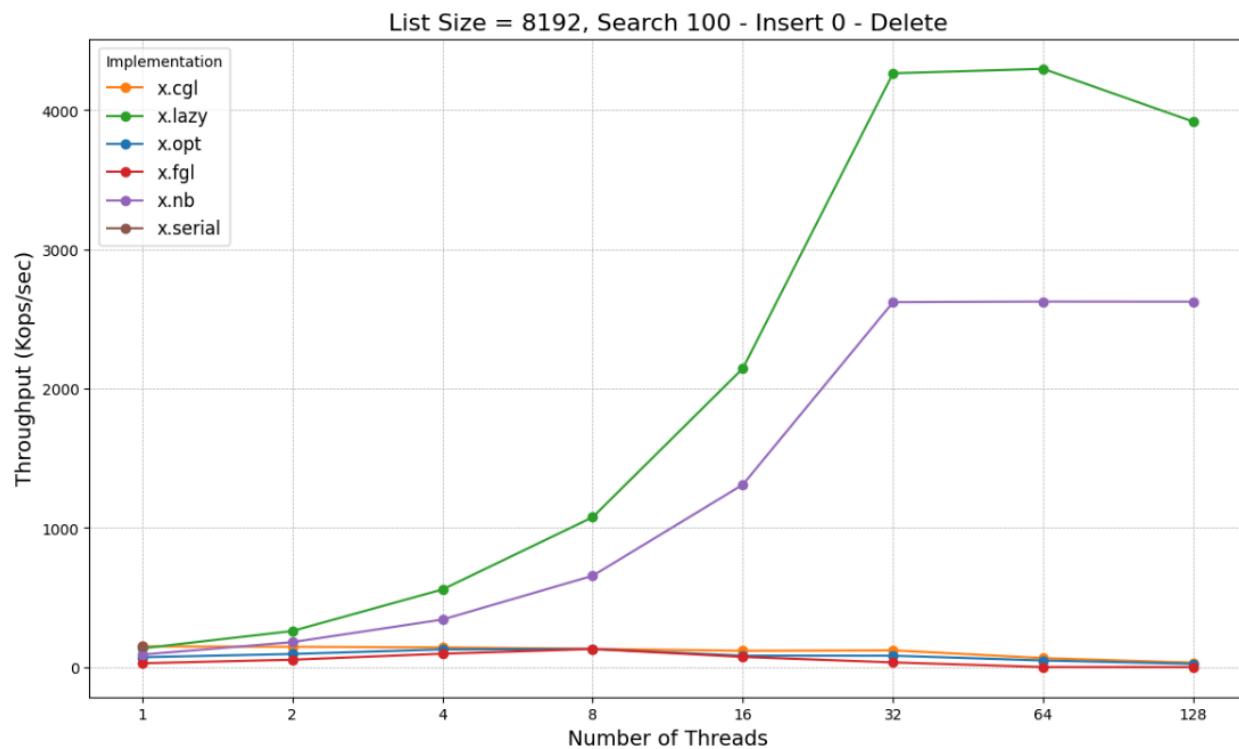
List Size = 1024, Search 20 - Insert 40 - Delete 40

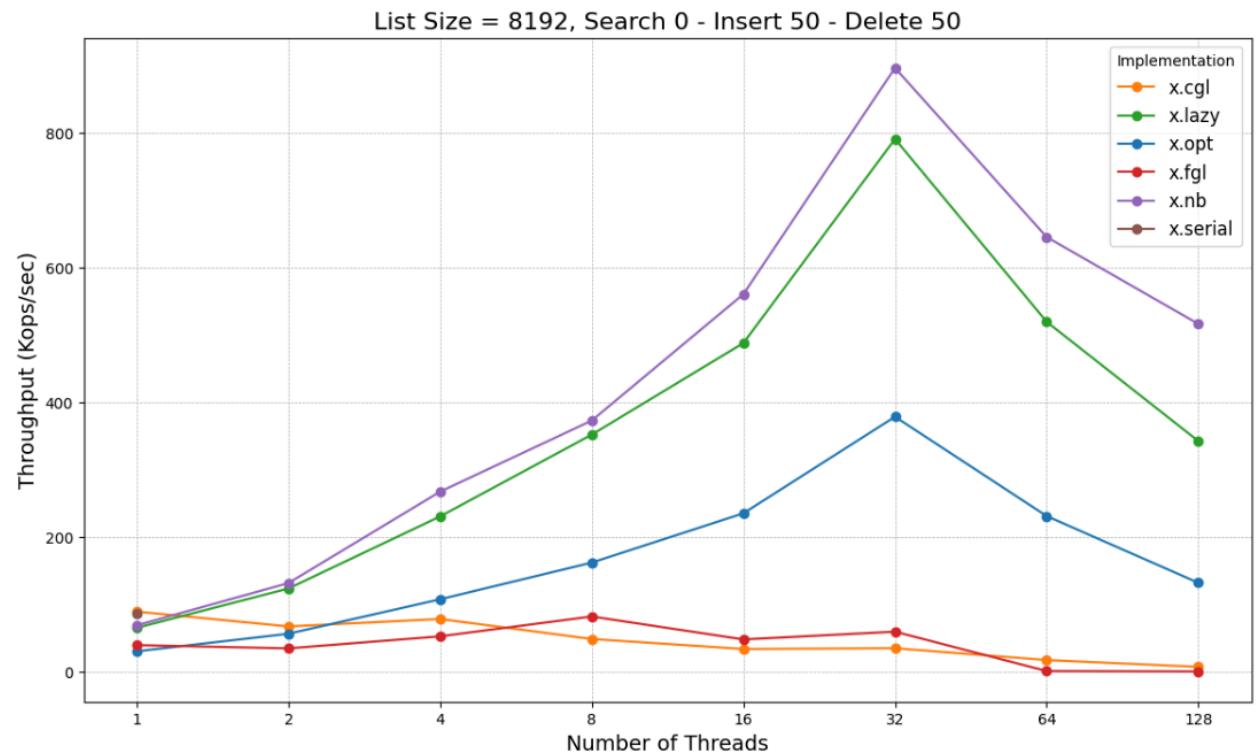
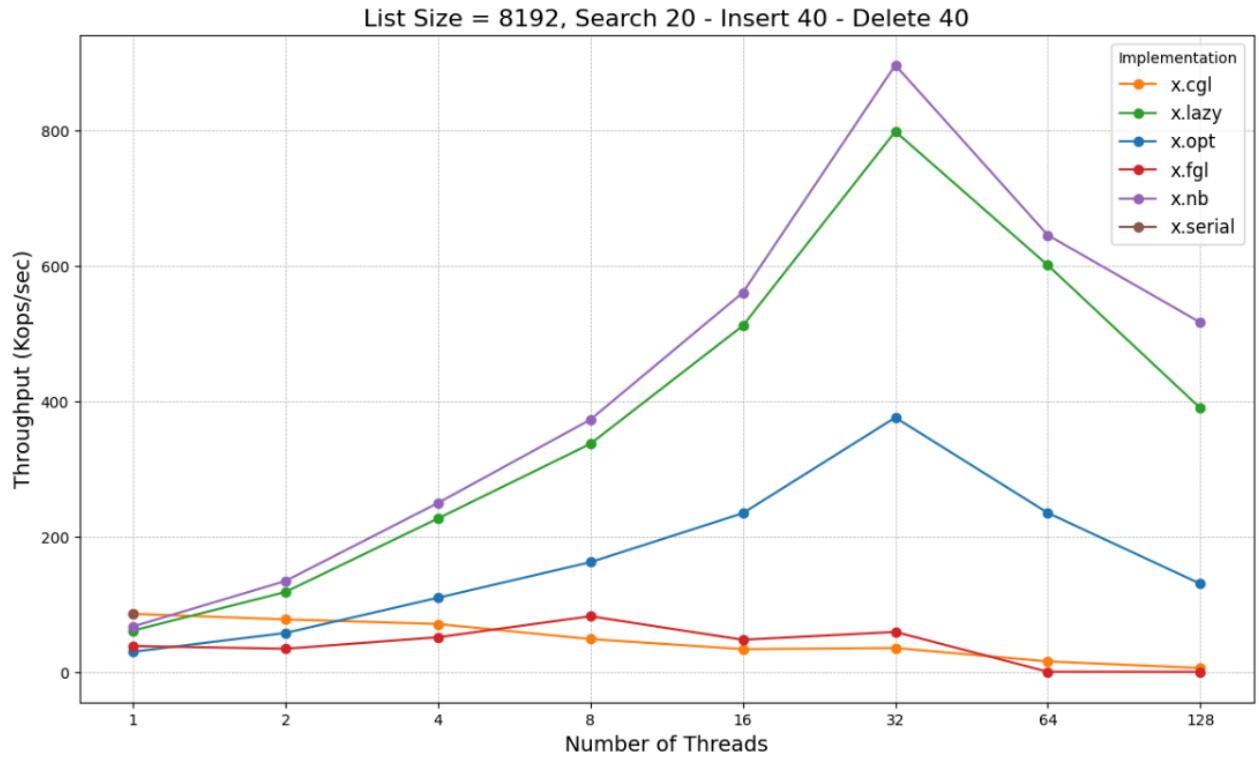


List Size = 1024, Search 0 - Insert 50 - Delete 50



Για List Size = 8192:





Παρατηρήσεις:

Coarse-grain locking (cgl):

Σε αυτή τη μέθοδο έχουμε ένα lock για ολόκληρη τη λίστα. Προκειμένου ένα thread να κάνει add(), remove(), contain() διεκδικεί το lock, το οποίο κάθε φορά χρησιμοποιεί ένα thread. Στο coarse-grain locking παρατηρούμε χαμηλότερη επίδοση, καθώς κάθε thread είναι πιθανό να έχει μεγάλο χρόνο αναμονής μέχρι να χρησιμοποιήσει το lock.

Fine-grain locking (fgl):

Το fgl είναι ουσιαστικά μια βελτίωση της μεθόδου cgl. Αντί για ένα ενιαίο, γενικό κλείδωμα ολόκληρης της λίστας, χρησιμοποιούνται πολλά μικρότερα κλειδώματα. Κάθε κόμβος μιας της λίστας μπορεί να έχει το δικό του κλείδωμα. Επιπλέον, επιτρέπεται πολλά threads να έχουν πρόσβαση στη δομή και να τροποποιούν διαφορετικά τμήματα ταυτόχρονα, οδηγώντας έτσι σε καλύτερη χρήση των πόρων. Στο fgl έχουμε καλύτερο performance σε σχέση με το cgl που κλειδώνει όλη τη λίστα, αλλά σε σχέση με άλλους αλγορίθμους είναι χειρότερο για τον λόγο ότι έχει δύσκολη υλοποίηση, υψηλό κόστος κτήσης και απελευθέρωσης των locks.

Optimistic synchronization:

Σε αυτή τη μέθοδο δεν χρησιμοποιούμε καθόλου locks μέχρι να βρούμε τον κόμβο-στόχο (διατρέχουμε τη λίστα μέχρι να τον βρούμε). Όταν τον βρούμε κλειδώνουμε τον συγκεκριμένο κόμβο καθώς και τον προηγούμενο του και ελέγχουμε αν ο προηγούμενος κόμβος (pred) είναι προσβάσιμος από την αρχή της δομής, και αν ο επόμενος του προηγούμενου είναι ο τρέχοντας (pred.next == curr).

Εάν δεν ισχύει κάποιο από τα δύο, τότε κάνουμε unlock και προσπαθούμε πάλι από την αρχή. Γενικά αυτή η μέθοδος είναι πιο αποδοτική από τις προηγούμενες, καθώς στη μέση περίπτωση χρησιμοποιούμε λιγότερα locks. Ο optimistic συγχρονισμός λειτουργεί καλύτερα από το fine-grain συγχρονισμό αν το κόστος να διατρέξουμε τη δομή 2 φορές χωρίς κλειδώματα είναι χαμηλότερο από το να την διατρέξουμε μία φορά με κλειδώματα. Επιπλέον, η validate διατρέχει τη λίστα από την αρχή, γεγονός που θέλουμε να το αποφύγουμε.

Lazy synchronization:

Η διαφορά της συγκεκριμένης μεθόδου είναι η προσθήκη μιας boolean μεταβλητής (marked) που δείχνει αν ο κόμβος βρίσκεται στη λίστα ή έχει διαγραφεί. Η contains() διατρέχει τη λίστα χωρίς να κλειδώνει και ελέγχει και την επιπλέον boolean μεταβλητή.

Η `add()` διατρέχει τη λίστα, κλειδώνει και ελέγχει τη συνέπεια της δομής καλώντας τη `validate`, η οποία δεν διατρέχει τη λίστα από την αρχή αλλά κάνει τοπικούς ελέγχους στους κόμβους `pred` και `curr`.

Σε αυτή τη μέθοδο, η `remove()` λειτουργεί σε 2 βήματα:

Το 1ο βήμα αφαιρεί λογικά τον κόμβο (θέτοντας τη `boolean` μεταβλητή σε `false`)

Το 2ο βήμα αφαιρεί φυσικά τον κόμβο επαναθέτοντας τους δείκτες.

Έτσι, παρατηρούμε επιτάχυνση στις αφαιρέσεις, καθώς πραγματοποιούνται πρώτα οι λογικές αφαιρέσεις που είναι πιο γρήγορες και οι φυσικές αφαιρέσεις (που είναι πιο αργές) πραγματοποιούνται μετέπειτα.

Ένα μειονέκτημα της συγκεκριμένης μεθόδου είναι ότι στις `remove()` και `add()` τα `threads` είναι πιθανό να καθυστερούν σημαντικά.

Non-blocking synchronization (nb):

Η ιδέα εδώ είναι να απαλείψουμε πλήρως τα `locks` και από τις 3 μεθόδους της δομής, `add()`, `remove()` και `contains()`. Έτσι, αποφεύγουμε καταστάσεις κατά τις οποίες η αποτυχία ενός νήματος που κρατάει ένα κλείδωμα θα οδηγήσει σε αποτυχία όλης της εφαρμογής (robustness). Στην υλοποίηση `non-blocking`, χρησιμοποιούνται τεχνικές όπως η ατομική ενημέρωση πεδίων (π.χ., μέσω της `Compare-and-Swap`), ώστε να αποτραπούν συνθήκες συναγωνισμού.

Συγκεκριμένα, τα πεδία `marked` και `next` της δομής αντιμετωπίζονται ως μία ατομική μονάδα, όπου οποιαδήποτε προσπάθεια ενημέρωσης του πεδίου `next` όταν το `marked` είναι `true` (δηλαδή ο κόμβος έχει διαγραφεί) θα αποτύχει. Επίσης, η αφαίρεση ενός κλειδιού περιλαμβάνει την ενημέρωση του πεδίου `marked`, ενώ η ενημέρωση των διευθύνσεων πραγματοποιείται μόνο μία φορά, ως σχεδιαστική επιλογή. Ακόμη, κάθε νήμα που διατρέχει τη λίστα μέσω των λειτουργιών `add()` και `remove()` αφαιρεί φυσικά τους κόμβους που συναντά και είναι διαγραμμένοι. Τέλος, αν η ατομική ενημέρωση αποτύχει, το νήμα επαναλαμβάνει την προσπάθεια ξεκινώντας ξανά από την αρχή της λίστας.

Όσον αφορά τις ιδιότητες προόδου, το **lock-free** εγγυάται ότι τουλάχιστον ένα νήμα θα κάνει πρόοδο, ενώ το **wait-free** διασφαλίζει ότι όλα τα νήματα ολοκληρώνουν τις λειτουργίες τους σε πεπερασμένο χρόνο.

Η τεχνική **non-blocking synchronization**, ως πιο αποδοτική μέθοδος, προσφέρει καλύτερο throughput σε σχέση με προηγούμενες τεχνικές.

Η παραπάνω περιγραφή των διαφορετικών μεθόδων συγχρονισμού φαίνεται και στα διαγράμματα. Σε όλες τις περιπτώσεις (και για τα 2 sizes και για όλα τα `threads`) παρατηρούμε ότι το **Coarse-grain locking** και το **Fine-grain locking** έχουν χαμηλή επίδοση. Αντίθετα, το **Lazy synchronization** και το **Non-blocking synchronization** έχουν πολύ υψηλότερη επίδοση, ειδικά όσο αυξάνονται τα `threads` (η καλύτερη επίδοση παρατηρείται για 32 `threads`).

Επιπλέον, παρατηρούμε ότι στην περίπτωση όπου έχουμε 100 searches, 0 insert και 0 delete, το **lazy** είναι καλύτερο από το **non-blocking**, ενώ στις περιπτώσεις όπου έχουμε 20-40-40 και 0-50-50 το **lazy** έχει χειρότερη επίδοση από το **non-blocking**, γεγονός που επιβεβαιώνει αυτά που αναλύσαμε παραπάνω. (όσο αυξάνονται οι κλήσεις στις μεθόδους add() και remove() τόσο πέφτει η απόδοση του lazy synchronization).

ΑΣΚΗΣΗ 3 - Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Στην 3η άσκηση κληθήκαμε να αναπτύξουμε και να βελτιστοποιήσουμε τον αλγόριθμο K-means σε GPU της NVIDIA, αξιολογώντας κάθε φορά τις επιδόσεις από το τελικό αποτέλεσμα του παράλληλου προγράμματος.

Naive version

Σε αυτή την έκδοση αναθέτουμε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι του αλγορίθμου: τον υπολογισμό των nearest clusters σε κάθε iteration. Συμπληρώνουμε όλα τα TODO στο πρόγραμμα που μας δίνεται: cuda_kmeans_naive.cu

Τα TODO που συμπληρώθηκαν με την σειρά είναι:

1. Υπορουτίνες get_tid() και euclid_dist_2():

```
__device__ int get_tid() {

    return blockIdx.x * blockDim.x + threadIdx.x;
    /* TODO: Calculate 1-Dim global ID of a thread */
}

/* square of Euclid distance between two multi-dimensional points */
__host__ __device__ inline static
double euclid_dist_2(int numCoords,
                     int numObjs,
                     int numClusters,
                     double *objects,      // [numObjs][numCoords]
                     double *clusters,     // [numClusters][numCoords]
                     int objectId,
                     int clusterId) {
    int i;
    double ans = 0.0;
```

```

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

```

Οι δύο προηγούμενες υπορουτίνες καλούνται από τον kernel `find_nearest_cluster()`:

```

__global__ static
void find_nearest_cluster(int numCoords,
                           int numObjs,
                           int numClusters,
                           double *objects,           // [numObjs][numCoords]
                           double *deviceClusters,   // [numClusters][numCoords]
                           int *deviceMembership,    // [numObjs]
                           double *devdelta) {

    /* 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;
        double dist, min_dist;

        /* 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);
        }
    }
}

```

```
    deviceMembership[tid] = index;
}
}
```

2. Μέσα στη συνάρτηση kmeans_gpu() συμπληρώνουμε τα TODO τα οποία είναι: Υπολογισμός κατάλληλου gridSize:

```
const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
numObjs;
const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
const unsigned int clusterBlockSharedDataSize = 0;
```

Αναθέτουμε σε κάθε thread της GPU ένα object, ώστε να υπολογίσει την απόστασή του από τα clusters και να βρει το κοντινότερό του, στο οποίο θα ανήκει στο επόμενο iteration. Άρα το grid που δημιουργούμε θα έχει τις παραπάνω διαστάσεις, όπου το (numThreadsPerClusterBlock - 1) αποτελεί το padding.

Αντιγραφή του πίνακα clusters στον πίνακα deviceClusters που έχει γίνει cudaMalloc:

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

Τέλος, αντιγράφουμε στην CPU τα αποτελέσματα που υπολόγισε ο kernel find_nearest_cluster():

```
/* TODO: Copy deviceMembership to membership */
checkCuda(cudaMemcpy(membership, deviceMembership, sizeof(int) * numObjs,
cudaMemcpyDeviceToHost));

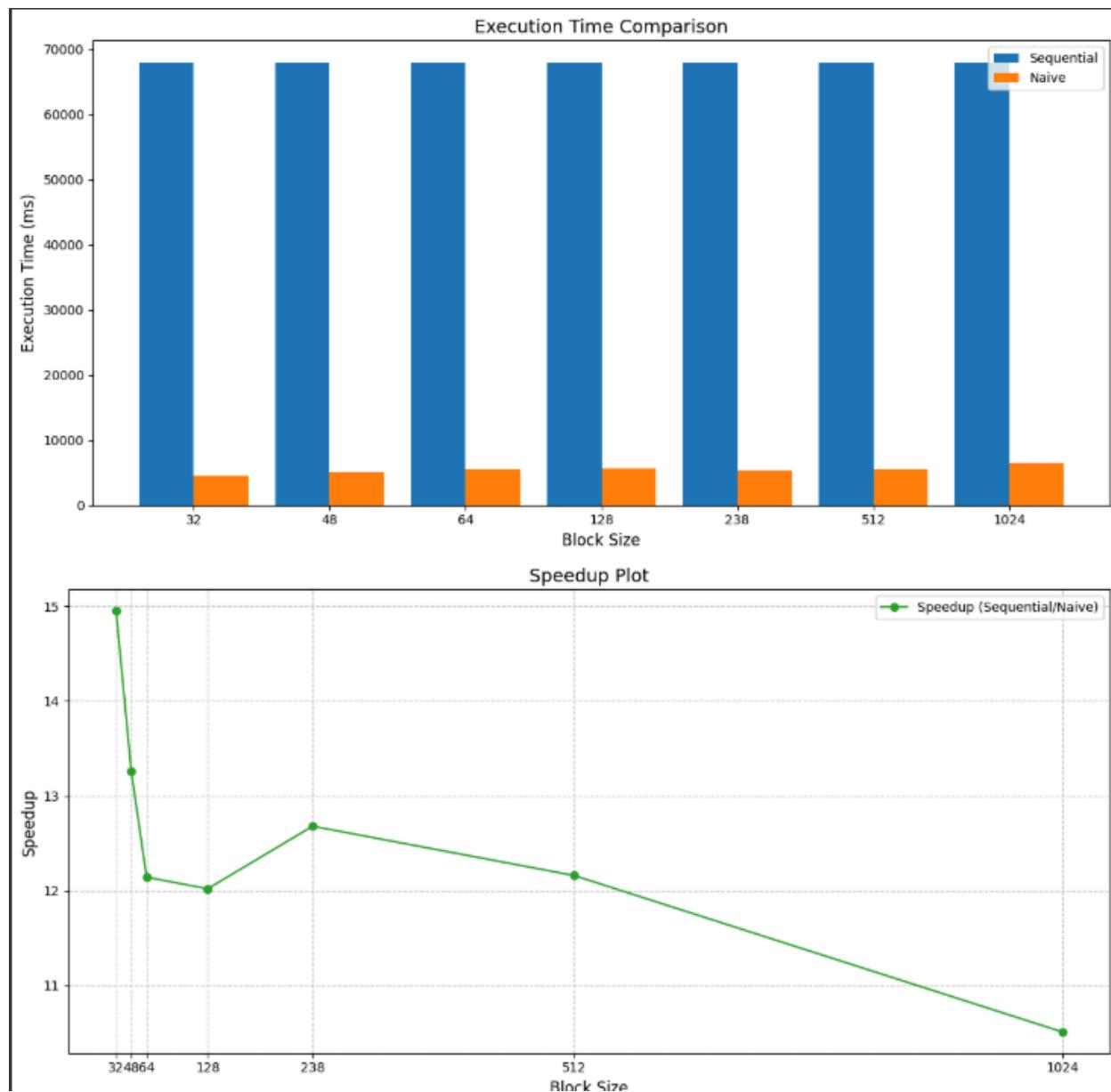
/* TODO: Copy dev_delta_ptr to &delta */
checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
```

Στην Naive υλοποίηση, η αποθήκευση των δεδομένων στους πίνακες που περνάμε ως παραμέτρους στον kernel γίνεται με row-based τρόπο, κάτι που θα δούμε αργότερα ότι δεν μας βοηθάει να εκμεταλλευτούμε το πολύ υψηλό bandwidth της GPU. Ως αποτέλεσμα αυτού δημιουργούνται καθυστερήσεις. Παρ' ολα αυτά, αναθέτουμε σε κάθε object 1 thread, κάτι το οποίο είναι πολύ πιο αποδοτικό σε σχέση με μια sequential εκτέλεση.

Πραγματοποιούμε μετρήσεις για την Naive version και την σειριακή έκδοση που μας δίνεται για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block_size = {32, 48, 64, 128, 238, 512, 1024}

Με βάση τον συνολικό χρόνο σε msec (για 10 loops) προκύπτει το ακόλουθο πινακάκι.

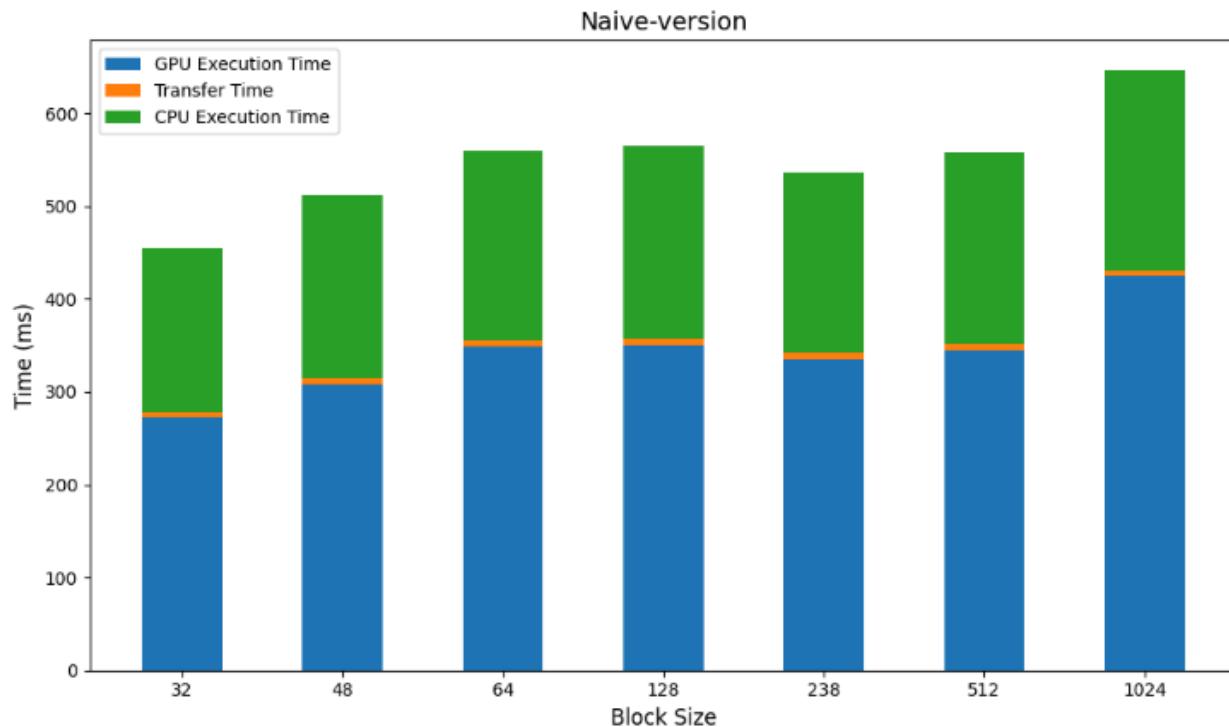
BlockSize	32	48	64	128	238	512	1024
Sequential	67927	67927	67927	67927	67927	67927	67927
Naive	4543	5124	5594	5651	5357	5586	6463



Τα παραπάνω διαγράμματα είναι:

- 1) διάγραμμα χρόνου εκτέλεσης που προκύπτει (x-axis = sequential και block_size, y-axis = time)
- 2) speedup plot σε σχέση με τη sequential (x-axis = block_size, y-axis = seq_time/time)

Η παράλληλη Naive version σε σχέση με την sequential είναι πολύ πιο γρήγορη και πετυχαίνουμε speedup ~11-15. Το Block Size φαίνεται να μην επηρεάζει ιδιαίτερα.



Στο παραπάνω stacked bar plot φαίνεται ότι αυτό που μας καθυστερεί στην naive version είναι ο χρόνος εκτέλεσης της CPU, καθώς είναι σχεδόν διπλάσιος σε σχέση με το άθροισμα του GPU execution time και του transfer time.

Παρατηρήσεις:

Για τα πειράματα μας χρησιμοποιούμε την GPU TeslaV100-SXM2-32GB η οποία διαθέτει 80 SMs (Streaming Multiprocessors).

Η υλοποίηση που έχουμε κάνει παραπάνω για τον kmeans δεν είναι ιδανική, καθώς δεν εκμεταλλεύμαστε πλήρως τις δυνατότητες της GPU και αναθέτουμε και στη CPU υπολογιστικά βαρύ κομμάτι (το update των νέων cluster centers). Επιπλέον το naive version δεν χρησιμοποιεί shared memory και οδηγεί σε χαμηλή επίδοση ανεξαρτήτως block size.

Οι χρόνοι μεταφοράς δεδομένων παραμένουν σχεδόν σταθεροί σε όλα τα block sizes (≈ 6.5 ms), κάτι που είναι αναμενόμενο, αφού εξαρτώνται κυρίως από το μέγεθος των δεδομένων και όχι από το block size.

Σκοπός μας για να πετύχουμε τα καλύτερα αποτελέσματα είναι να έχουμε επαρκώς μεγάλο block size ώστε να “γεμίζουμε” κάθε φορά το SM και να μην αφήνουμε ανεκμετάλλευτους πόρους.

Για block size=48 θα περιμέναμε κανονικά να μην έχει καλή απόδοση διότι δεν είναι δύναμη του 2 και έχουμε underutilization. Στην πραγματικότητα όμως η GPU έχει half warps με 16 threads και για αυτό φαίνεται να έχει από τις καλύτερες αποδόσεις.

Για block size = 1024 παρατηρείται μια μικρή αύξηση του χρόνου εκτέλεσης, η οποία πιθανώς να οφείλεται είτε σε έντονο ανταγωνισμό κατά την εκτέλεση της atomic εντολής είτε στο φαινόμενο του register spilling, δηλαδή όταν ορισμένα threads δεν βρίσκουν διαθέσιμους registers και αναγκάζονται να χρησιμοποιήσουν την πιο αργή local memory.

Transpose version

Σε αυτήν την version καλούμαστε να αλλάξουμε τη δομή των δεδομένων από row-based σε column-based. Δηλαδή πλέον ένα object / cluster δεν θα είναι μια γραμμή αλλά μια στήλη του αντίστοιχου πίνακα.

1. Η νέα euclid_dist_2_transpose():

```
/* square of Euclid distance between two multi-dimensional points using column-base format */
__host__ __device__ inline static
double euclid_dist_2_transpose(int numCoords,
                               int numObjs,
                               int numClusters,
                               double *objects,      // [numCoords][numObjs]
                               double *clusters,     // [numCoords][numClusters]
                               int objectId,
                               int clusterId) {
    int i;
    double ans = 0.0;

    /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
    clusters, but for column-base format!!! */
    for (int i = 0; i < numCoords; i++) {
        double diff = objects[i * numObjs + objectId] - clusters[i * numClusters + clusterId];
        ans += diff * diff;
    }      //arr[row][col] = arr[col * numRows + row]
```

```

    return (ans);
}

```

2. Ορίζουμε τους απαιτούμενους buffers:

```

/* TODO: Transpose dims */
double **dimObjects = (double**)calloc_2d(numCoords, numObjs, sizeof(double)); // ->
[numCoords][numObjs]
double **dimClusters = (double**)calloc_2d(numCoords, numClusters, sizeof(double)); // ->
[numCoords][numClusters]
double **newClusters = (double**)calloc_2d(numCoords, numClusters, sizeof(double)); // ->
[numCoords][numClusters]

```

Αλλάζουμε τα objects σε column-based indexing:

```

// 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];
    }
}

```

```

const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ?
blockSize : numObjs;
const unsigned int numClusterBlocks = (numObjs +
numThreadsPerClusterBlock - 1)/ numThreadsPerClusterBlock; /* TODO:
Calculate Grid size, e.g. number of blocks. */

```

Τέλος, μετατρέπουμε τα clusters στην προηγούμενη μορφή

```

/*TODO: Update clusters using dimClusters. Be carefull of layout!!!
clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */

for (i = 0; i < numClusters; i++) {           // Iterate over rows of the original layout
    for (j = 0; j < numCoords; j++) { // Iterate over columns of the original layout
        clusters[i*numCoords + j] = dimClusters[j][i];
    }
}

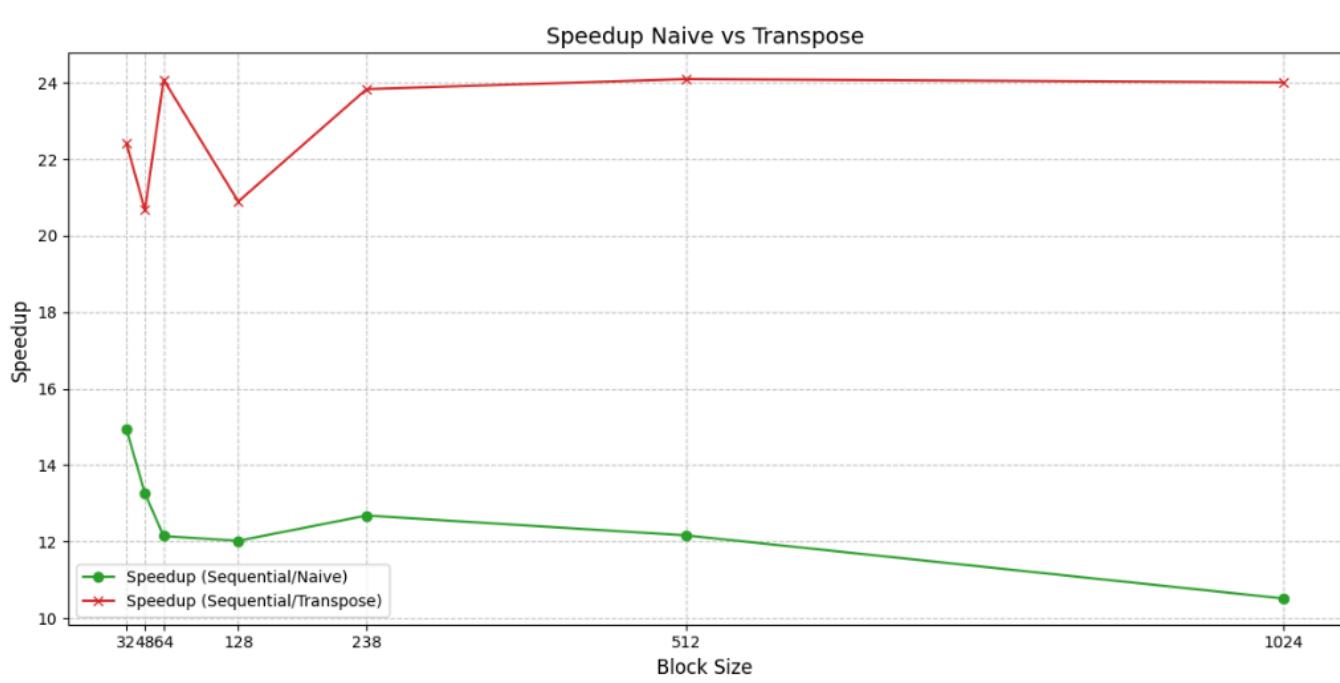
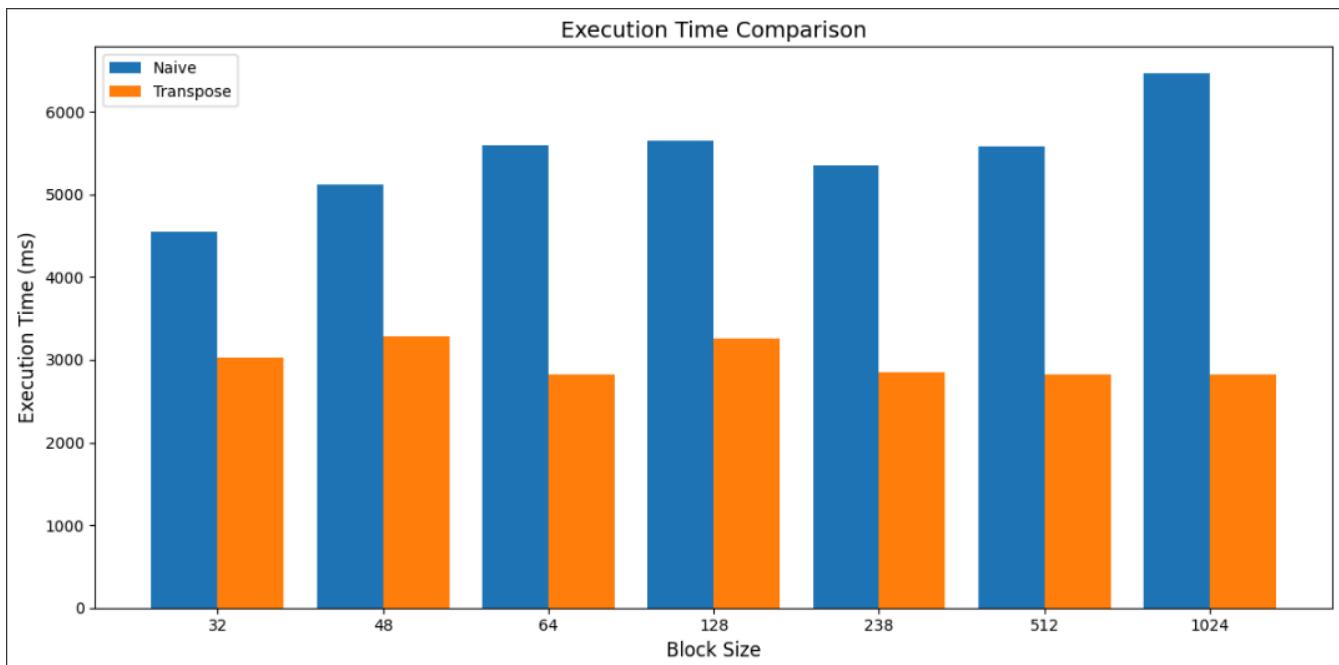
```

Αξιολόγηση επίδοσης σε σύγκριση και με την transpose version:

{Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block_size = {32, 48, 64, 128, 238, 512, 1024}

Με βάση το total σε msec (για 10 loops) προκύπτει το ακόλουθο πινακάκι.

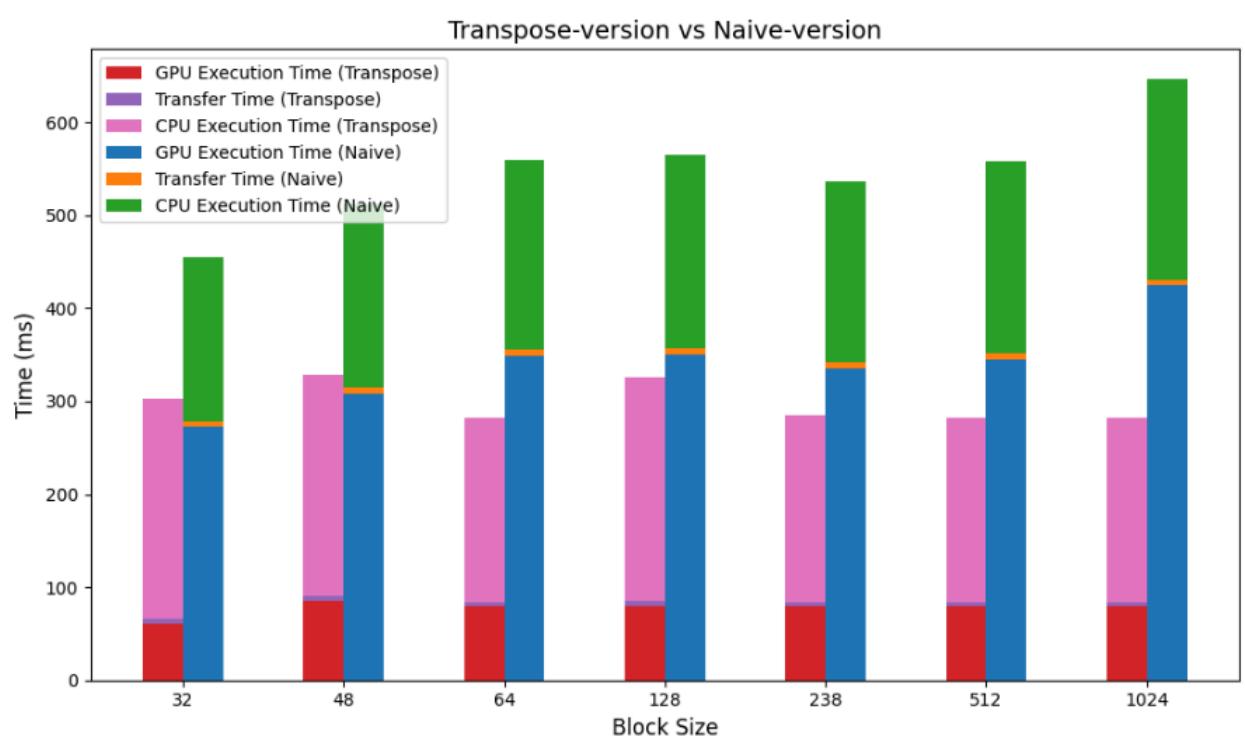
BlockSize	32	48	64	128	238	512	1024
Sequential	67927	67927	67927	67927	67927	67927	67927
Naive	4543	5124	5594	5651	5357	5586	6463
Transpose	3029	3285	2821	3251	2850	2819	2829



1)

Η GPU οργανώνει τις προσβάσεις στην καθολική μνήμη ώστε να μειώσει τον συνολικό χρόνο που απαιτείται για αυτές. Σε κάθε block, οι ταυτόχρονες προσβάσεις μνήμης από τα threads του block συγχωνεύονται σε μία κοινή πρόσβαση, εξυπηρετώντας όλα τα threads. Η καθολική μνήμη είναι οργανωμένη σε τμήματα συγκεκριμένου μεγέθους (bytes), και η αποδοτικότητα αυτής της συγχώνευσης εξαρτάται από την απόσταση των φυσικών διευθύνσεων που προσπελαύνουν τα threads. Γι' αυτόν τον λόγο, αναμένεται σημαντική βελτίωση στην απόδοση καθώς αυξάνεται το μέγεθος των blocks.

Παρόλα αυτά στο διάγραμμα του transpose δεν παρατηρούμε σταθερά ανοδική πορεία, καθώς έχουμε ένα spike για block size=64.



Η transpose version είναι πιο γρήγορη από τη naive. Αυτό έχει να κάνει με τη μεγιστοποίηση του bandwidth την οποία πετυχαίνει με την αλλαγή σε column-based από row-based που είχαμε στην naive version. Η βασική διαφορά συμβαίνει εκεί που τα objects (δηλαδη threads) υπολογίζουν τις αποστάσεις από τα clusters.

Όταν ένα thread κάνει access στη μνήμη γίνεται ένα transaction το οποίο θα στείλει τα ζητούμενα δεδομένα. Εάν μέσα στο half-warp (16 threads) τα γειτονικά threads χρησιμοποιούν δεδομένα που βρίσκονται μέσα στο transaction τότε θα τα λάβουν από το ίδιο transaction και θα έχουμε μεγιστοποίηση του bandwidth.

Αναπαράσταση πινάκων (objects/clusters), x_i , y_i coords του i -οστου object/cluster.

Naive version

```
x1 , y1  
x2 , y2  
x3 , y3  
.....  
x15 , y15  
x16 , y16
```

Transpose version

```
x1 , x2 , x3 , ..... , x15 , x16  
y1 , y2 , y3 , ..... , y15 , y16
```

Με την naive έκδοση σε κάθε transaction από τη μνήμη θα πάρνουμε x1,y1,x2,y2 ...x8,y8 διότι θα φέρει από τη μνήμη 128bytes δεδομένων / sizeof(double) = 16 στοιχεία. Τα y1, y2 ..y8 που πήραμε από τη μνήμη μαζί με τα x μας είναι για την ώρα περιττά. Μετά το στοιχείο x8 θα θέλαμε να έχουμε το x9 ... x16. Με την naive έκδοση εκμεταλλευόμαστε το μισό bandwidth. Με την transpose έκδοση εκμεταλλευόμαστε όλο το bandwidth.

Shared version

Στην συγκεκριμένη υλοποίηση παίρνουμε την προηγούμενη καλύτερη μας (transpose) και για επιπλέον βελτίωση βάζουμε τον πίνακα clusters στην πολύ γρήγορη shared memory του block. Διατηρούμε λοιπόν την column-based υλοποίηση.

Τα threads του block αντιγράφουν τα δεδομένα των clusters στον shared πίνακα

```
__global__ static  
void find_nearest_cluster(int numCoords,  
                           int numObjs,  
                           int numClusters,  
                           double *objects,           // [numCoords][numObjs]  
                           double *deviceClusters,    // [numCoords][numClusters]  
                           int *deviceMembership,     // [numObjs]  
                           double *devdelta) {  
    extern __shared__ double shmemClusters[];  
  
    /* TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.  
     * BEWARE: Make sure operations is complete before any thread continues... */  
    for(int i=threadIdx.x; i<numCoords*numClusters; i+=blockDim.x){  
        shmemClusters[i]=deviceClusters[i];  
    }  
    __syncthreads(); //ensures that threads are in sync  
  
    /* Get the global ID of the thread. */  
    int tid = get_tid();
```

Ορισμός της shared memory

```
/* Define the shared memory needed per block.
```

```

- BEWARE: We can overrun our shared memory here if there are too many
clusters or too many coordinates!
- This can lead to occupancy problems or even inability to run.
- Your exercise implementation is not requested to account for that (e.g. always
assume deviceClusters fit in shmemClusters */
//to allaksame to apo katw apo -1
const unsigned int clusterBlockSharedDataSize = numClusters * numCoords * sizeof(double);

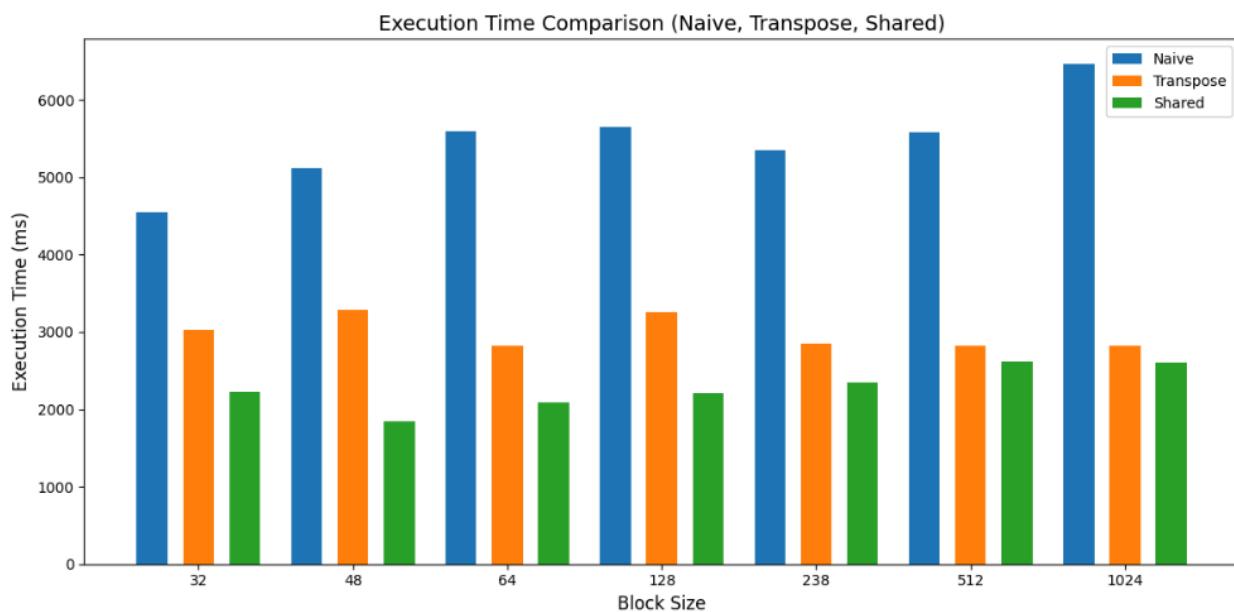
```

Αξιολόγηση επίδοσης σε σύγκριση και με την transpose version:

{Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block_size = {32, 48, 64, 128, 238, 512, 1024}

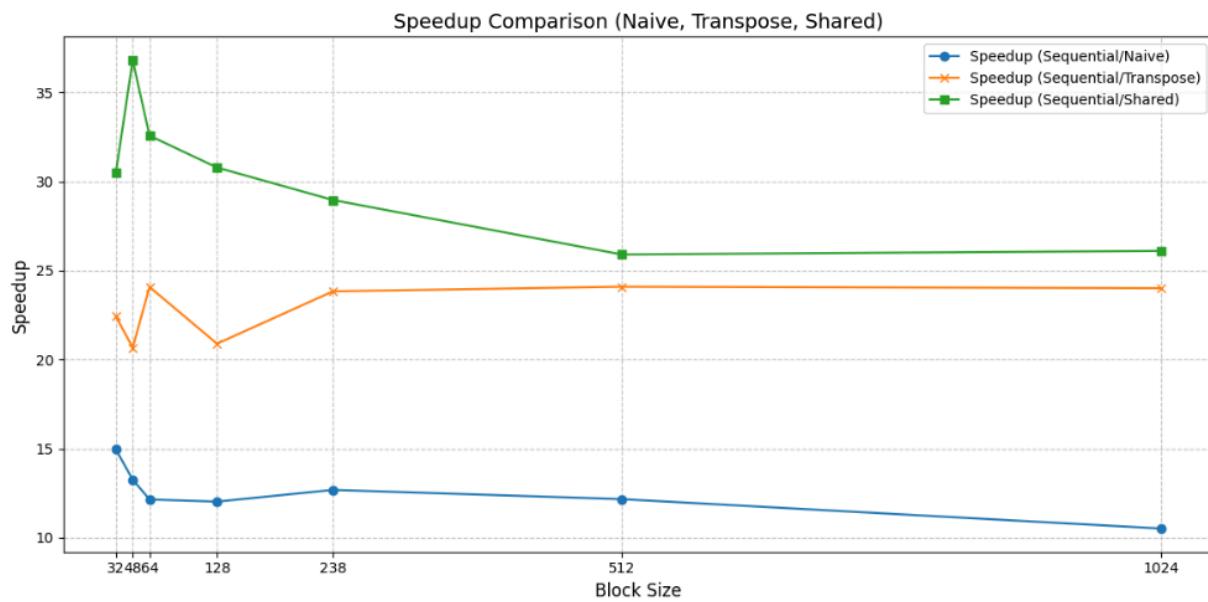
Με βάση το total σε msec (για 10 loops) προκύπτει το ακόλουθο πινακάκι.

blockSize	32	48	64	128	238	512	1024
Sequential	67927	67927	67927	67927	67927	67927	67927
Naive	4543	5124	5594	5651	5357	5586	6463
Transpose	3029	3285	2821	3251	2850	2819	2829
Shared	2228	1845	2085	2206	2345	2622	2602



Παρατηρούμε πολύ καλύτερη επίδοση σε σχέση με τις προηγούμενες υλοποιήσεις. Για να το πετύχουμε αυτό χρησιμοποιούμε την πολύ γρήγορη on-chip shared memory του block για την

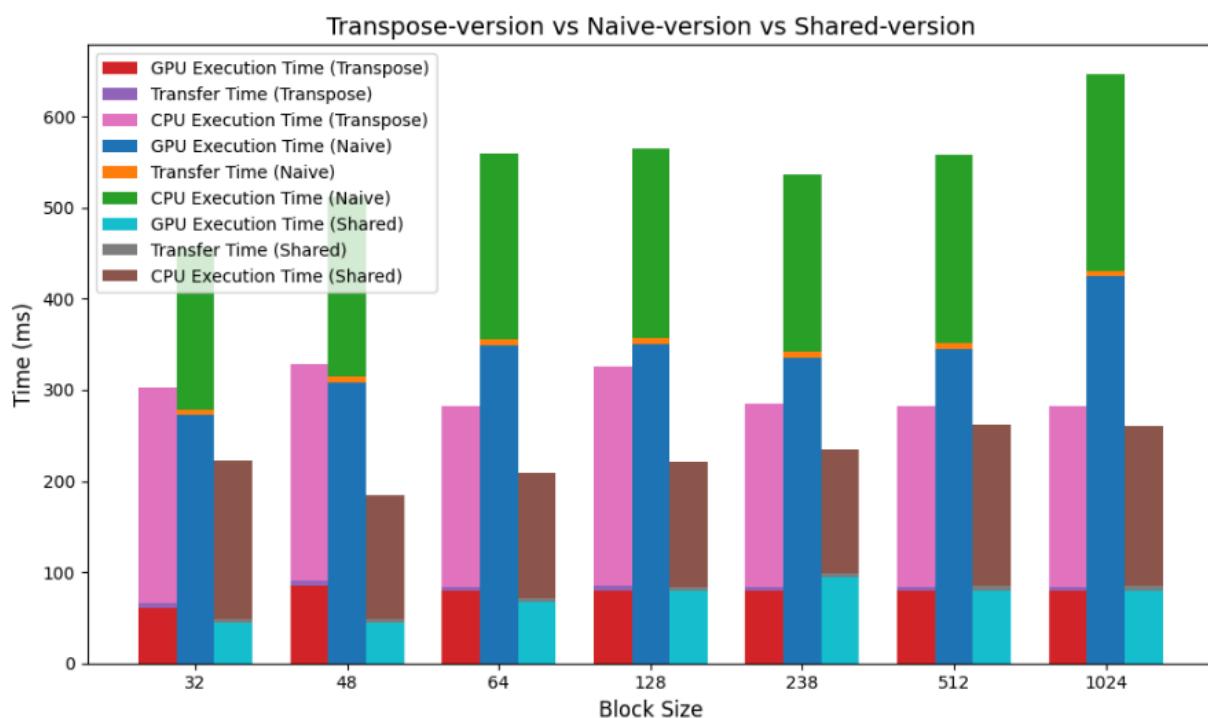
αποθήκευση των clusters. Όλα τα threads ενός block έχουν πρόσβαση στην shared memory.



To blockSize = 48 ήταν το χειρότερο στην transpose έκδοση, όμως τώρα είναι το καλύτερο.

Το 48 αν και δεν είναι δύναμη του 2, επειδή η GPU χρησιμοποιεί half-warps(16 threads) αποδίδει πολύ καλά.

Το 1024 δεσμεύει πολύ μεγάλο μέρος της L1 cache, με αποτέλεσμα η GPU να μην έχει αρκετή cache για αποδοτικές προσπελάσεις μνήμης. Οπότε γίνονται περισσότερες προσπελάσεις στην global memory.



Στην shared υλοποίηση παρατηρούμε γενικά ότι με την αύξηση του block size πέφτει η επίδοση. Αυτό συμβαίνει διότι όταν αυξάνονται τα threads, αυξάνονται και τα αιτήματα για διαφορετικές θέσεις μνήμης στην τοπική shared memory. Συνεπώς έχουμε bank conflicts και τελικά έχουμε καθυστερήσεις στις προσβάσεις μνήμης.

Οι αλλαγές των περιεχομένων του κάθε thread φαίνονται μόνο στο block του και όχι σε όλα όπως πρίν που γινόταν αποθήκευση στη global memory. Συνεπώς, το μέγεθος του block είναι αυτό που ορίζει πόσα threads θα βλέπουν τον κάθε ξεχωριστό πίνακα shmemClusters.

Σύγκριση υλοποιήσεων / bottleneck Analysis

1) Για την naive υλοποίηση έχουμε

Naive-version

```
blocksize 32
-> t_cpu_avg = 176.482987 ms
-> t_gpu_avg = 272.174644 ms
-> t_transfers_avg = 5.636191 ms
blocksize 64
-> t_cpu_avg = 203.509259 ms
-> t_gpu_avg = 349.405670 ms
-> t_transfers_avg = 6.568527 ms
blocksize 128
-> t_cpu_avg = 208.687568 ms
-> t_gpu_avg = 349.749398 ms
-> t_transfers_avg = 6.674457 ms
blocksize 238
-> t_cpu_avg = 193.935227 ms
-> t_gpu_avg = 335.266519 ms
-> t_transfers_avg = 6.586337 ms
blocksize 512
-> t_cpu_avg = 207.725930 ms
-> t_gpu_avg = 344.298935 ms
-> t_transfers_avg = 6.656742 ms
blocksize 1024
-> t_cpu_avg = 215.336776 ms
-> t_gpu_avg = 424.410510 ms
-> t_transfers_avg = 6.573272 ms
```

Παρατηρούμε ότι το μεγαλύτερο bottleneck που εμποδίζει την συνολική επίδοση του iterative μέρους είναι οι υπολογισμοί που δημιουργούνται στη CPU.

Transpose-version

```

blocksize 32
-> t_cpu_avg = 237.025905 ms
-> t_gpu_avg = 60.471106 ms
-> t_transfers_avg = 5.404449 ms
blocksize 48
-> t_cpu_avg = 238.299513 ms
-> t_gpu_avg = 85.328317 ms
-> t_transfers_avg = 4.900002 ms
blocksize 64
-> t_cpu_avg = 197.741079 ms
-> t_gpu_avg = 79.644990 ms
-> t_transfers_avg = 4.749608 ms
blocksize 128
-> t_cpu_avg = 240.136647 ms
-> t_gpu_avg = 79.647708 ms
-> t_transfers_avg = 5.338645 ms
blocksize 238
-> t_cpu_avg = 200.547862 ms
-> t_gpu_avg = 79.733968 ms
-> t_transfers_avg = 4.783440 ms
blocksize 512
-> t_cpu_avg = 197.791409 ms
-> t_gpu_avg = 79.394603 ms
-> t_transfers_avg = 4.706979 ms
blocksize 1024
-> t_cpu_avg = 198.841953 ms
-> t_gpu_avg = 79.301882 ms
-> t_transfers_avg = 4.753327 ms

```

Όπως και πριν, παρατηρούμε ότι η CPU έχει αναλάβει βαρύ uppoloγιστικά κομμάτι και μας κάνει bottleneck την συνολική επίδοση. Το update clusters γίνεται ακόμα στη CPU, ενώ μπορεί να γίνει στη GPU (το κάνουμε στην all gpu υλοποίηση).

Shared-version

```

blocksize 32
-> t_cpu_avg = 174.304366 ms
-> t_gpu_avg = 43.855047 ms
-> t_transfers_avg = 4.704738 ms
blocksize 48
-> t_cpu_avg = 136.000395 ms
-> t_gpu_avg = 43.823361 ms
-> t_transfers_avg = 4.745317 ms
blocksize 64
-> t_cpu_avg = 136.837482 ms
-> t_gpu_avg = 67.027926 ms
-> t_transfers_avg = 4.702067 ms
blocksize 128
-> t_cpu_avg = 136.416841 ms
-> t_gpu_avg = 79.520679 ms
-> t_transfers_avg = 4.751945 ms

```

```

blocksize 238
-> t_cpu_avg = 135.491872 ms
-> t_gpu_avg = 94.392729 ms
-> t_transfers_avg = 4.686093 ms
blocksize 512
-> t_cpu_avg = 176.985025 ms
-> t_gpu_avg = 80.042052 ms
-> t_transfers_avg = 5.239320 ms
blocksize 1024
-> t_cpu_avg = 175.034523 ms
-> t_gpu_avg = 80.076003 ms
-> t_transfers_avg = 5.175591 ms

```

Τα αποτελέσματα είναι παρόμοια με πριν.

2)

Old Configuration: {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block_size = {32, 48, 64, 128, 238, 512, 1024}

Με βάση το total σε msec (για 10 loops) προκύπτει το ακόλουθο πινακάκι.

blockSize	32	48	64	128	238	512	1024
Sequential	67927	67927	67927	67927	67927	67927	67927
Naive	4543	5124	5594	5651	5357	5586	6463
Transpose	3029	3285	2821	3251	2850	2819	2829
Shared	2228	1845	2085	2206	2345	2622	2602

New Configuration: {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} για {32, 48, 64, 128, 238, 512, 1024}

Με βάση το total σε msec (για 10 loops) προκύπτει το ακόλουθο πινακάκι.

blockSize	32	48	64	128	238	512	1024
Sequential	124709	124709	124709	124709	124709	124709	124709
Naive	4761	4779	4711	4726	4799	4721	4761
Transpose	4292	4346	4285	4283	3547	4306	4281
Shared	3812	3474	3023	3666	3558	2926	2924

Naive-version

```
blocksize 32
-> t_cpu_avg = 360.575914 ms
-> t_gpu_avg = 38.208771 ms
-> t_transfers_avg = 77.383041 ms
blocksize 48
-> t_cpu_avg = 361.321712 ms
-> t_gpu_avg = 34.626293 ms
-> t_transfers_avg = 81.952453 ms
blocksize 64
-> t_cpu_avg = 347.262645 ms
-> t_gpu_avg = 33.284545 ms
-> t_transfers_avg = 90.593362 ms
blocksize 128
-> t_cpu_avg = 340.255499 ms
-> t_gpu_avg = 32.033420 ms
-> t_transfers_avg = 100.398850 ms
blocksize 238
-> t_cpu_avg = 341.733909 ms
-> t_gpu_avg = 37.504148 ms
-> t_transfers_avg = 100.737238 ms
blocksize 512
-> t_cpu_avg = 360.588121 ms
-> t_gpu_avg = 33.250475 ms
-> t_transfers_avg = 78.277111 ms
blocksize 1024
-> t_cpu_avg = 362.199783 ms
-> t_gpu_avg = 32.660174 ms
-> t_transfers_avg = 81.244993 ms
```

Transpose-version

```
blocksize 32
-> t_cpu_avg = 310.638213 ms
-> t_gpu_avg = 35.341501 ms
-> t_transfers_avg = 83.238673 ms
blocksize 48
-> t_cpu_avg = 313.648009 ms
-> t_gpu_avg = 37.037778 ms
-> t_transfers_avg = 83.989620 ms
blocksize 64
-> t_cpu_avg = 307.564926 ms
-> t_gpu_avg = 33.047009 ms
-> t_transfers_avg = 87.912607 ms
blocksize 128
-> t_cpu_avg = 316.046047 ms
-> t_gpu_avg = 31.740117 ms
```

```
-> t_transfers_avg = 80.540419 ms

blocksize 238
-> t_cpu_avg = 247.614884 ms
-> t_gpu_avg = 31.984043 ms
-> t_transfers_avg = 75.170946 ms

blocksize 512
-> t_cpu_avg = 315.057755 ms
-> t_gpu_avg = 33.325601 ms
-> t_transfers_avg = 82.215047 ms

blocksize 1024
-> t_cpu_avg = 313.777018 ms
-> t_gpu_avg = 31.463695 ms
-> t_transfers_avg = 82.951832 ms
```

Shared-version

```
blocksize 32
-> t_cpu_avg = 260.281873 ms
-> t_gpu_avg = 32.943702 ms
-> t_transfers_avg = 88.045931 ms

blocksize 48
-> t_cpu_avg = 236.715698 ms
-> t_gpu_avg = 34.250712 ms
-> t_transfers_avg = 76.465821 ms

blocksize 64
-> t_cpu_avg = 199.391389 ms
-> t_gpu_avg = 29.218411 ms
-> t_transfers_avg = 73.715258 ms

blocksize 128
-> t_cpu_avg = 249.402857 ms
-> t_gpu_avg = 30.774975 ms
-> t_transfers_avg = 86.451554 ms

blocksize 238
-> t_cpu_avg = 243.287754 ms
-> t_gpu_avg = 30.015802 ms
-> t_transfers_avg = 82.575369 ms

blocksize 512
-> t_cpu_avg = 191.193175 ms
-> t_gpu_avg = 27.875233 ms
-> t_transfers_avg = 73.534846 ms

blocksize 1024
-> t_cpu_avg = 190.321517 ms
-> t_gpu_avg = 28.663158 ms
-> t_transfers_avg = 73.466444 ms
```

To gpu_avg έχει μειωθεί για όλες τις υλοποιήσεις σε σχέση με το παλιό configuration.

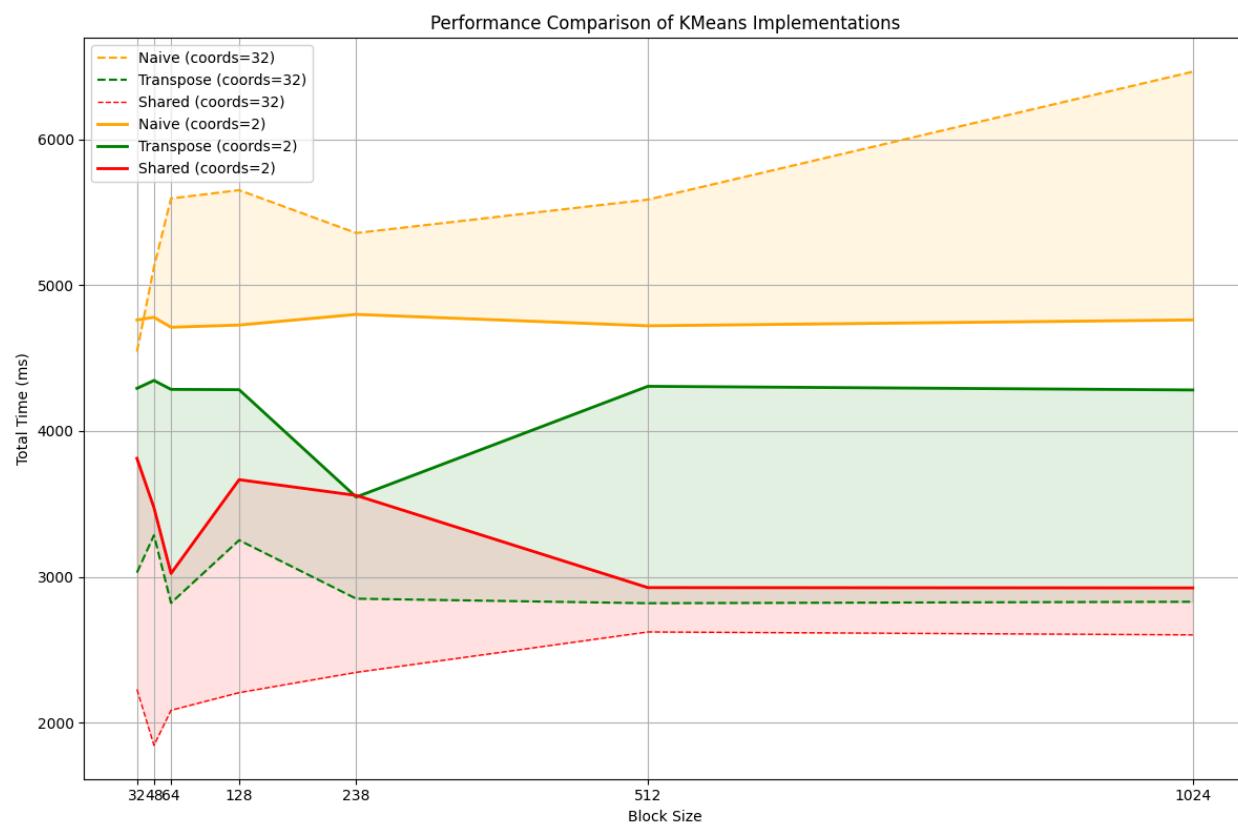
To cpu_avg έχει αυξηθεί για όλες τις υλοποιήσεις σε σχέση με το παλιό configuration.

To transfers_avg έχει αυξηθεί κατα πολύ (~80ms) σε σχέση με παλιά (~5ms).

Γενικά σε όλους τους χρόνους (για όλα τα block) παρατηρούνται μεγαλύτεροι χρόνοι σε όλες τις υλοποιήσεις εκτός από την naive.

Η μεγαλύτερη διαφορά παλιού-καινούργιου configuration είναι ο χρόνος transfers_avg που από περίπου 5ms, τώρα πλέον παίρνει περίπου 80ms.

2)



Πρώτο configuration: Coords=32 → numCoords*numClusters = 32*64= 2048 threads

Δεύτερο configuration: Coords=2 → numCoords*numClusters = 2*64= 128 threads

Για μεγαλύτερο numCoords πετυχαίνουμε μικρότερες καθυστερήσεις διότι είναι πολύ λιγότερο πιθανό δύο atomic εντολές να θέλουν να κλειδώσουν την ίδια θέση μνήμης. Για Coords=2 σε σχέση με Coords=32 παρατηρούμε πολύ μεγαλύτερα transfer time όπως φαίνεται από τους παραπάνω πίνακες.

Παρατηρούμε ότι η παρούσα shared υλοποίηση δεν είναι κατάλληλη για την επίλυση του

kmeans για arbitrary (δηλαδή τυχαία) configurations, όπως πχ στην περίπτωση που ο αριθμός των συντεταγμένων είναι πολύ μικρός (coords = 2).

Full-Offload (All-GPU) version

Στις προηγούμενες versions είχαμε παρατηρήσει ότι το μεγαλύτερο bottleneck γίνεται στους υπολογισμούς που τρέχουν στη CPU. Το κομμάτι που υπολογίζοταν στη CPU ήταν ο υπολογισμός των νέων κέντρων, οπότε θα αναθέσουμε αυτό τον υπολογισμό στην GPU με τη βοήθεια μιας συνάρτησης update_centroids().

Η ανανεωμένη συνάρτηση find_nearest_cluster() με 2 παραπάνω ορίσματα για τον υπολογισμό των πινάκων devicenewClusterSize και devicenewClusters.

```
__global__ static
void find_nearest_cluster(int numCoords,
                           int numObjs,
                           int numClusters,
                           double *deviceobjects,           // [numCoords][numObjs]
/*
                                     TODO: If you choose to do (some of) the new centroid calculation
here, you will need some extra parameters here (from "update_centroids").
*/
                           int *devicenewClusterSize,      // added this , [numClusters]
                           double *devicenewClusters,     // added this ,
                           [numCoords][numClusters]
                           double *deviceClusters,       // [numCoords][numClusters]
                           int *deviceMembership,        // [numObjs]
                           double *devdelta) {
extern __shared__ double shmemClusters[];

/* TODO: copy me from shared version... */
for(int i=threadIdx.x; i<numCoords*numClusters; i+=blockDim.x){
    shmemClusters[i]=deviceClusters[i];
}
__syncthreads(); //ensures that threads are in sync

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

/* TODO: copy me from shared version... */
if (tid<numObjs) {
    int index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2() with correct objectId/clusterId using clusters
in shmem*/
```

```

    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
shmemClusters, tid, 0);
    for (i = 1; i < numClusters; i++) {
        /* TODO: call dist = euclid_dist_2_(...) with correct objectId/clusterId using
clusters in shmem*/
        dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
shmemClusters, 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;

    /* TODO: additional steps for calculating new centroids in GPU? */
    atomicAdd(&devicenewClusterSize[index] , 1);

    for (int j=0; j<numCoords; j++)
        atomicAdd(&devicenewClusters[j*numClusters + index] , deviceobjects[j*numObjs +
tid]);
    }
}

```

Η παρακάτω συνάρτηση είναι υπεύθυνη για τον υπολογισμό των νέων κέντρων.

```

__global__ static
void update_centroids(int numCoords,
                      int numClusters,
                      int *devicenewClusterSize,           // [numClusters]
                      double *devicenewClusters,         // [numCoords][numClusters]
                      double *deviceClusters)          // [numCoords][numClusters])
{
    /* TODO: additional steps for calculating new centroids in GPU? */
    int tid = get_tid();
    int cluster = tid%numClusters;
    int clusterSize;

    if(tid<numClusters*numCoords) {
        clusterSize = devicenewClusterSize[cluster];
        if(clusterSize>0)
            deviceClusters[tid] = devicenewClusters[tid]/clusterSize;
        devicenewClusters[tid]=0;//extra
    }
    __syncthreads();//extra
}

```

```

        devicenewClusterSize[cluster] = 0; //extra
    }
}

```

Βάζουμε τα έξιτρα 2 ορίσματα της `find_nearest_cluster()` στον ορισμό της

```

/* TODO: change invocation if extra parameters needed*/
find_nearest_cluster
    <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
    (numCoords, numObjs, numClusters,
     deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters,
     deviceMembership, dev_delta_ptr);
}

```

Θεωρούμε ότι κάθε thread θα αντιπροσωπεύει μία διάσταση ενός cluster.

```

const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ?
blockSize : numCoords * numClusters; /* TODO: can use different blocksize here if deemed
better */
const unsigned int update_centroids_dim_sz =
((numCoords*numClusters-1)/update_centroids_block_sz) + 1; /* TODO: calculate dim for
"update_centroids" */

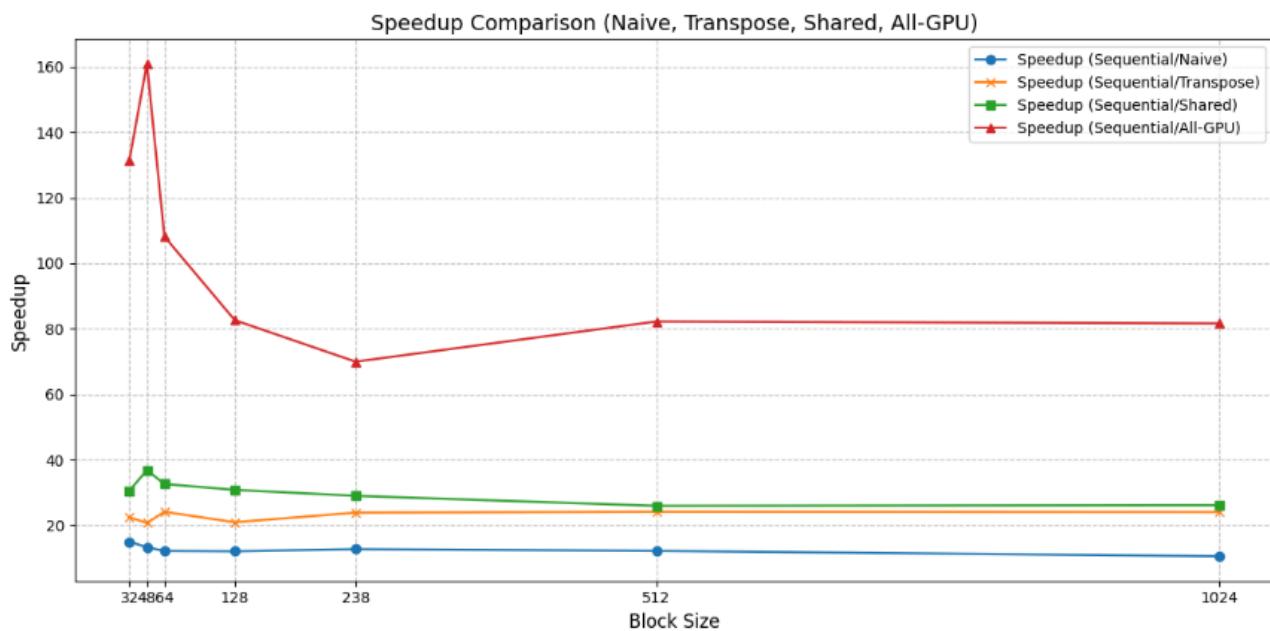
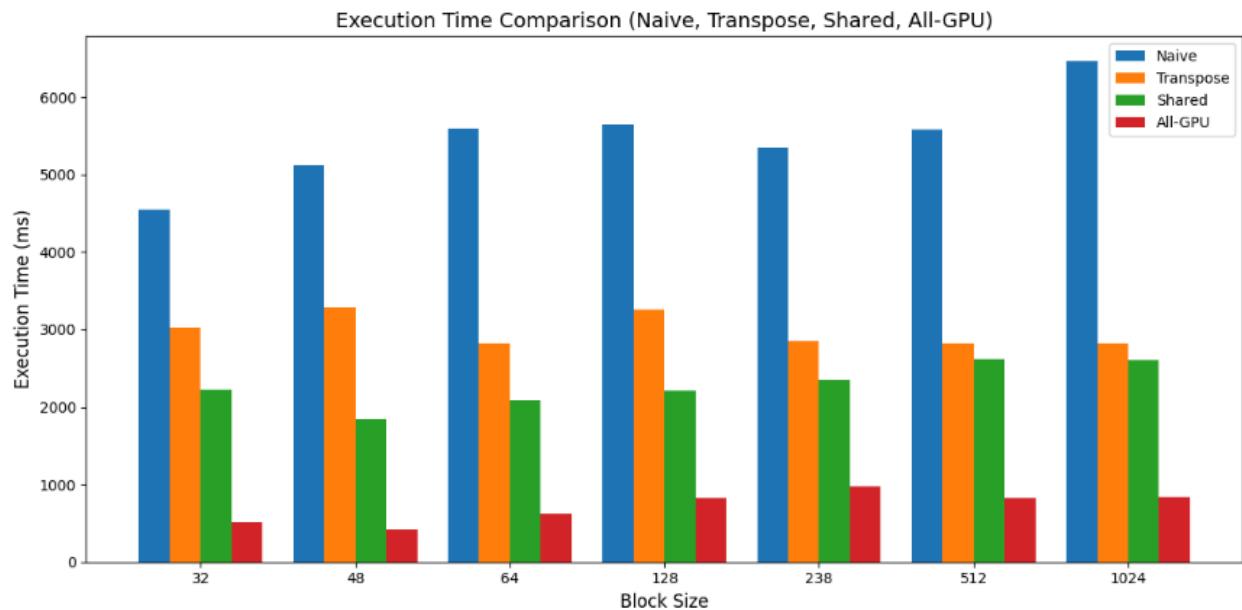
timing_gpu = wtime();
/* TODO: use dim for "update_centroids" and fire it*/
update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
    (numCoords, numClusters, devicenewClusterSize, devicenewClusters,
     deviceClusters);
}

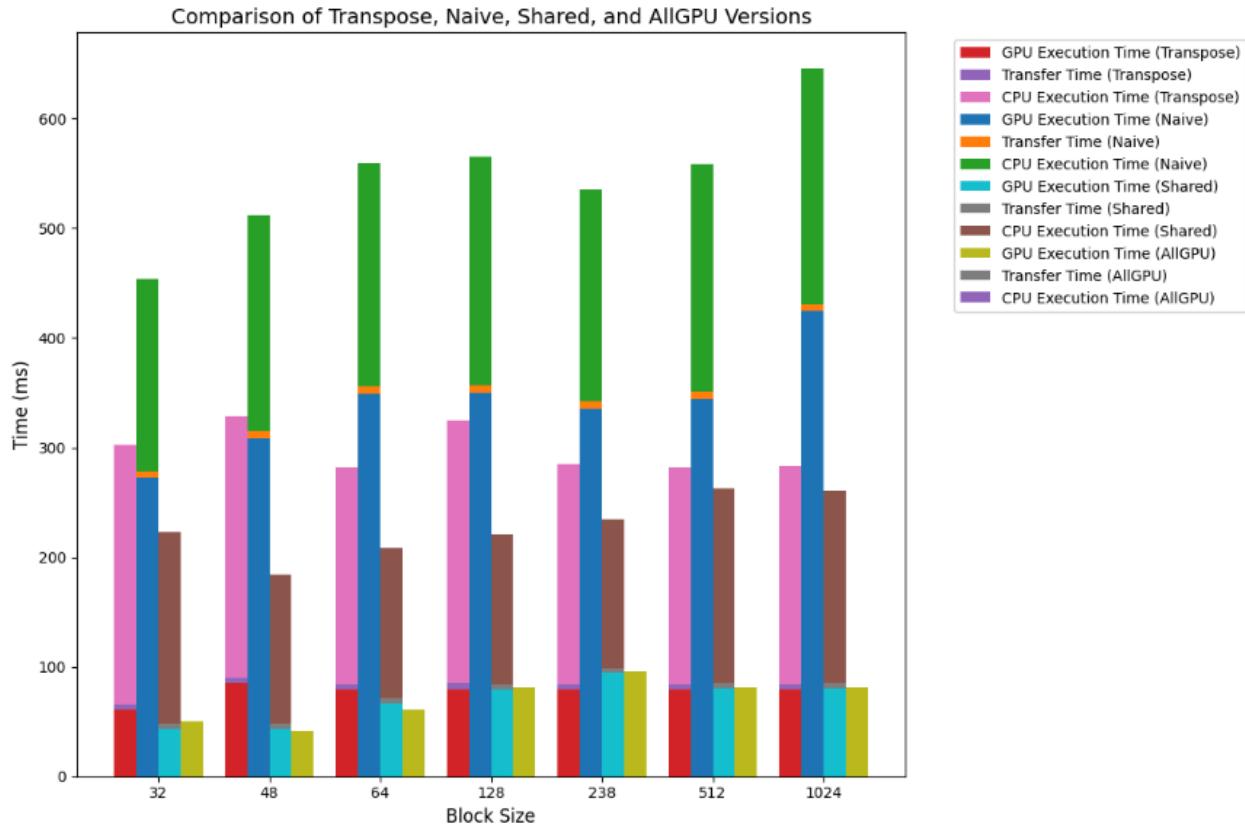
```

Configuration: {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block_size = {32, 48, 64, 128, 238, 512, 1024}

Με βάση το total σε msec (για 10loops) προκύπτει το ακόλουθο πινακάκι.

blockSize	32	48	64	128	238	512	1024
Sequential	67927	67927	67927	67927	67927	67927	67927
Naive	4543	5124	5594	5651	5357	5586	6463
Transpose	3029	3285	2821	3251	2850	2819	2829
Shared	2228	1845	2085	2206	2345	2622	2602
All-gpu	517	422	628	822	971	826	832

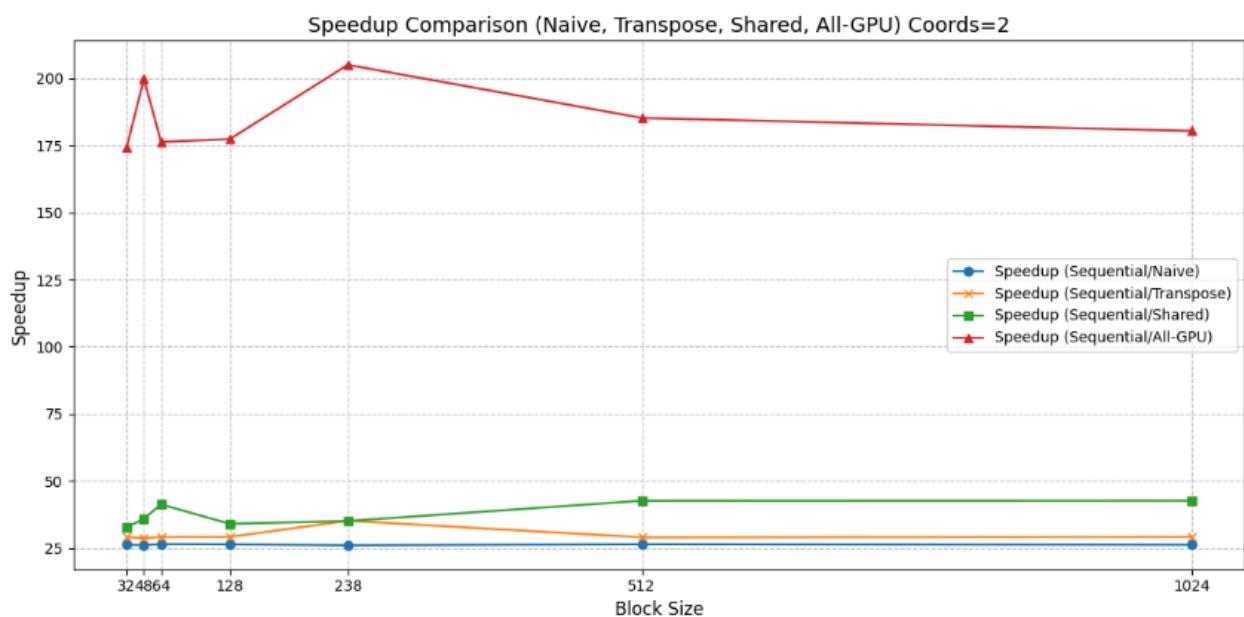
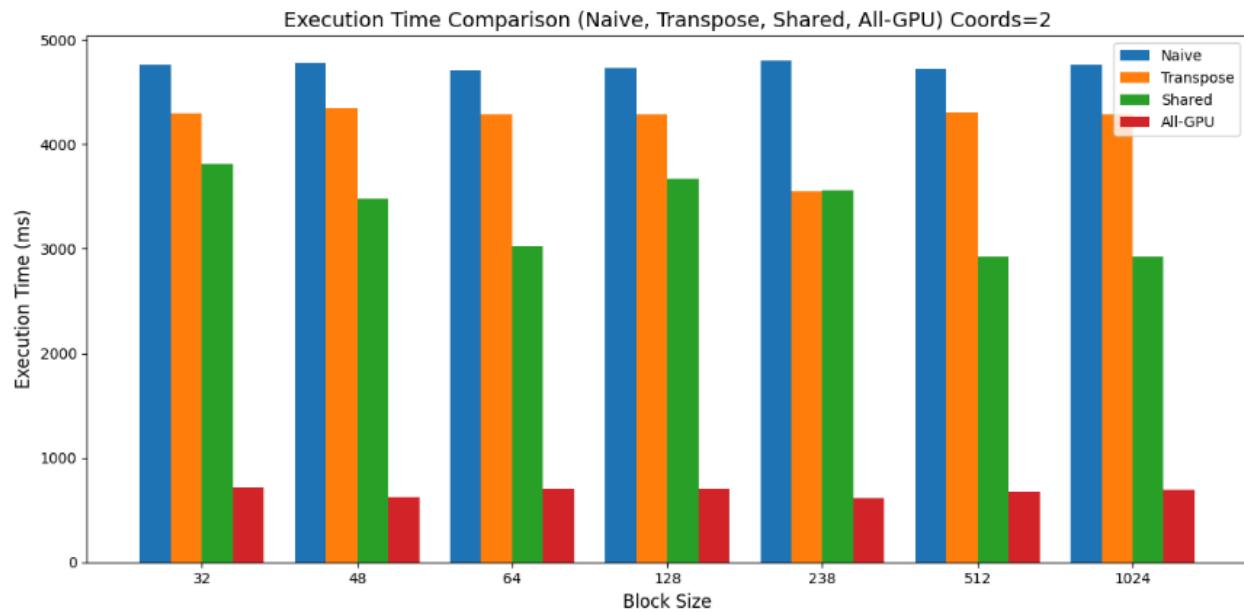


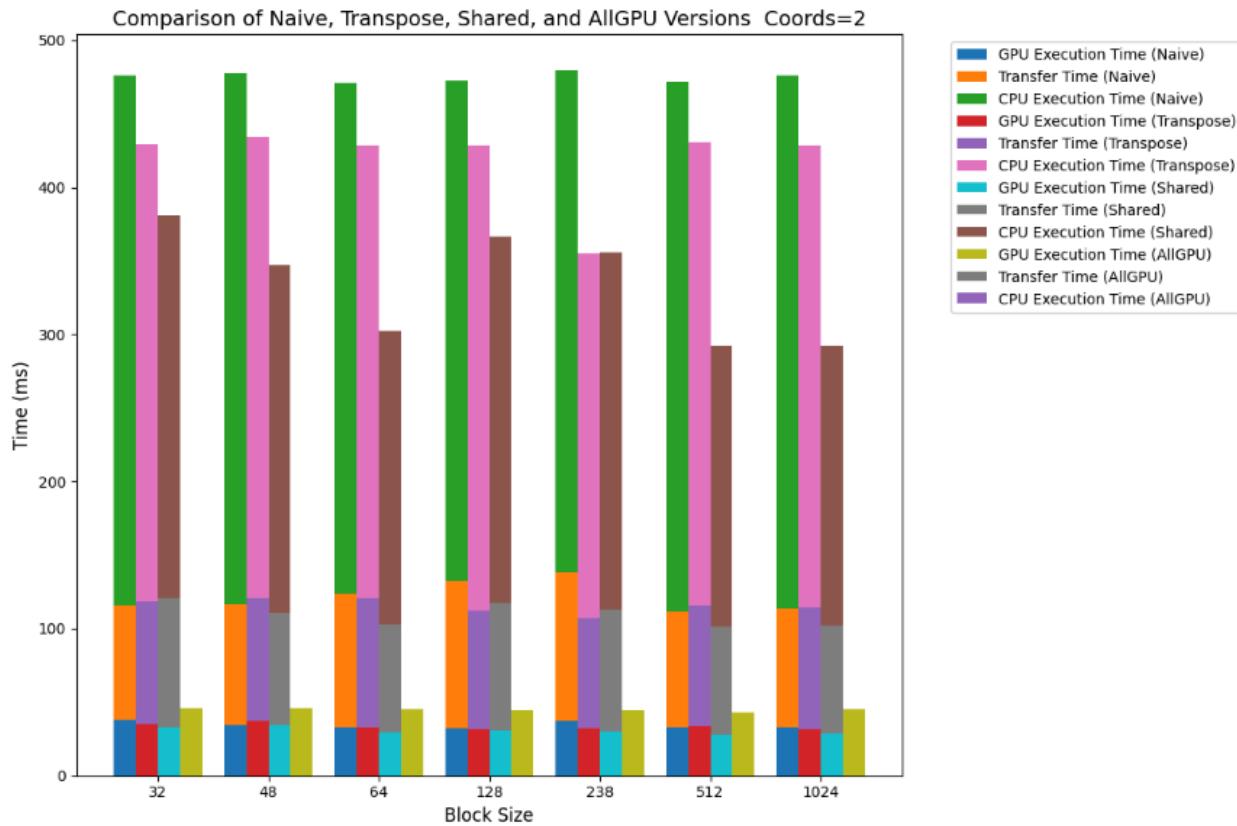


Configuration: {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} για {32, 48, 64, 128, 238, 512, 1024}

Με βάση το total σε msec (για 10loops) προκύπτει το ακόλουθο πινακάκι.

blockSize	32	48	64	128	238	512	1024
Sequential	124709	124709	124709	124709	124709	124709	124709
Naive	4761	4779	4711	4726	4799	4721	4761
Transpose	4292	4346	4285	4283	3547	4306	4281
Shared	3812	3474	3023	3666	3558	2926	2924
All-gpu	715	624	707	703	608	673	691





2)

To blocksize καθορίζει τον αριθμό των threads που περιλαμβάνονται σε κάθε block του `find_nearest_cluster` και, κατ' επέκταση, τα threads που μοιράζονται τη shared memory σε κάθε block.

Στο πρώτο configuration, επηρεάζει τον αριθμό των threads στα blocks του `update_centroids`, καθώς εκεί ισχύει ότι `numCoords * numClusters = 2048`.

Στο δεύτερο configuration, όπου `numCoords * numClusters = 128`, το μέγιστο μέγεθος των blocks του δεύτερου kernel φτάνει τα 2048 threads.

3)

Γνωρίζουμε ότι οι GPU είναι κατάλληλες για εφαρμογές που απαιτούν τεράστιο αριθμό threads τα οποία θα κάνουν μικρές και όχι ιδιαίτερα έξυπνες δουλειές. Ωστόσο όπως προαναφέραμε το `update_centroids` τρέχει παράγοντας μόνο `numCoords*numClusters` threads, στο πρώτο config αυτό ισούται με 32×64 . Οι τρεις προηγούμενες versions είχαν καθυστερήσεις στο κομμάτι της CPU αλλά και των transfer time. Πλέον, αφού δουλεύουμε μόνο στην GPU καταργούμε και τα δύο αυτά bottlenecks.

Το κομμάτι του `find_nearest_cluster` περιμένουμε να απαιτεί περισσότερο χρόνο στην all gpu

version αφού αυξήσαμε τις πράξεις που κάνει κάθε thread ενώ σε κάποια σημεία προσθέσαμε atomic εντολές.

Το κομμάτι του update_centroids όπως απαιτεί και αυτό κάποιο χρόνο όμως το tradeoff μεταξύ του update_centroids και της CPU είναι τεράστιο.

4)

Πρώτο configuration: Coords=32 → numCoords*numClusters = 32*64= 2048 threads

Δεύτερο configuration: Coords=2 → numCoords*numClusters = 2*64= 128 threads

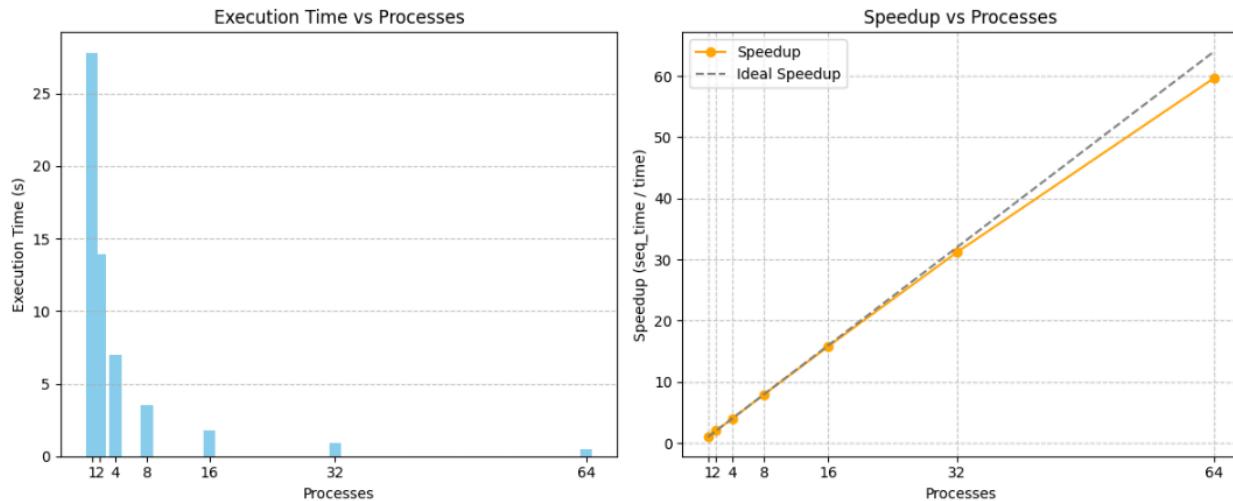
Για μεγαλύτερο numCoords πετυχαίνουμε μικρότερες καθυστερήσεις διότι είναι πολύ λιγότερο πιθανό δύο atomic εντολές να θέλουν να κλειδώσουν την ίδια θέση μνήμης. Για Coords=2 σε σχέση με Coords=32 παρατηρούμε πολύ μεγαλύτερα transfer time, όπως φαίνεται και στα παραπάνω διαγράμματα (γκρι, πορτοκαλί, μοβ μπάρες).

ΑΣΚΗΣΗ 4 - Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

4.1)

Σε αυτό το ερώτημα κληθήκαμε να υλοποιήσουμε άλλη μια φορά τον αλγόριθμο k-means παράλληλα, αλλά αυτή τη φορά με τη χρήση MPI.

MPI

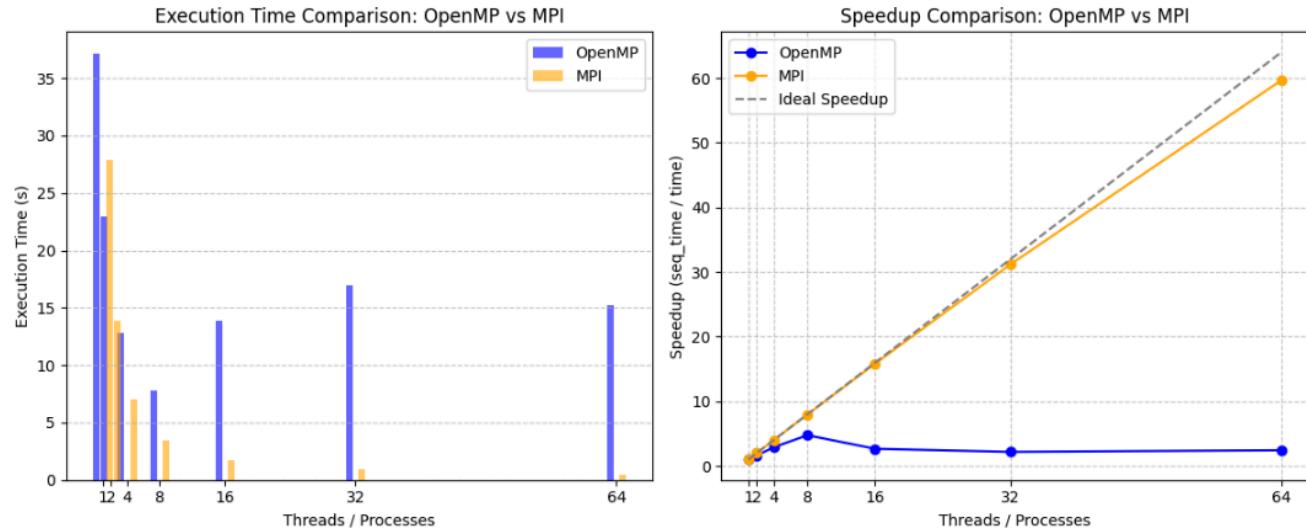


Παρατηρήσεις:

Έχουμε, σημαντική μείωση του χρόνου εκτέλεσης καθώς αυξάνονται οι διεργασίες. Το speedup ακολουθεί σχεδόν γραμμική αύξηση έως 16 διεργασίες, αλλά για 32, 64 διεργασίες αποκλίνει από την ιδανική κλιμάκωση λόγω overhead επικοινωνίας και συγχρονισμού.

BONUS

Σύγκριση MPI με OpenMp



Παρατηρούμε ότι η MPI υλοποίηση είναι πιο αποδοτική από την OpenMp αφού έχουμε μικρότερους χρόνους εκτέλεσης και μεγαλύτερα speedup. Η OpenMP υλοποίηση αρχίζει να επιβραδύνεται μετά τα 8 threads, πιθανώς λόγω αυξημένου contention στη μνήμη και overhead ενώ η MPI συνεχίζει να κλιμακώνεται καλά μέχρι 64 διεργασίες.

Τελικά συμπεραίνουμε ότι η MPI υλοποίηση κλιμακώνει καλύτερα, καθώς μειώνει το κόστος επικοινωνίας χρησιμοποιώντας διεργασίες αντί για threads, ενώ η OpenMp υλοποίηση είναι έχει πλεονεκτήματα για μικρό αριθμό threads, καθώς δεν απαιτεί επικοινωνία μέσω δικτύου ή διαμοιραζόμενης μνήμης.

4.2)

Κάθε διεργασία λαμβάνει ένα τμήμα (block) του αρχικού πίνακα, το οποίο αντιγράφει στον τοπικό της πίνακα. Η αντιγραφή ξεκινά από τη θέση [1][1], αφήνοντας κενή την πρώτη γραμμή και στήλη, ώστε να υπάρχει χώρος για τα δεδομένα που θα ανταλλάξουν οι γειτονικές διεργασίες. Για να επιτευχθεί αυτό, χρησιμοποιούμε τα custom datatypes του MPI.

Η κατανομή των δεδομένων από τη διεργασία 0 πραγματοποιείται μέσω:

```
MPI_Barrier(CART_COMM);
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block, &(u_current[1][1]),
1, local_block, 0, CART_COMM);
```

Για κάθε block υπολογίζουμε τα όρια των γραμμών και των στηλών που θα χρησιμοποιηθούν στους υπολογισμούς. Αυτά τα όρια εξαρτώνται από τη θέση του block στον πίνακα και από το αν χρησιμοποιείται padding.

Τα δεδομένα που πρέπει να σταλούν αποθηκεύονται στους αντίστοιχους buffers πριν την αποστολή τους στις γειτονικές διεργασίες:

```
// Copy data to send buffers
for (i = 0; i < local[0]; i++) {
    right_send[i] = u_current[i+1][local[1]];
    left_send[i] = u_current[i+1][1];
}

for (i = 0; i < local[1]; i++) {
    up_send[i] = u_current[1][i+1];
    down_send[i] = u_current[local[0]][i+1];
}
```

Η υλοποίηση της επικοινωνίας μεταξύ των διεργασιών έχει γίνει μέσω των MPI Isend και MPI Irecv. Κάθε διεργασία επικοινωνεί μόνο με τα γειτονικά blocks, αν υπάρχουν, στα οποία στέλνει τα δεδομένα.

Για να παρακολουθείται ο αριθμός των αποστολών και λήψεων που βρίσκονται σε εξέλιξη, χρησιμοποιείται ένας μετρητής (counter). Αυτός αυξάνεται κάθε φορά που εκτελείται μια αποστολή ή λήψη, ώστε στο τέλος να γνωρίζουμε πόσες συνολικά επικοινωνίες πρέπει να περιμένουμε να ολοκληρωθούν (με το MPI_Waitall που θα χρησιμοποιήσουμε παρακάτω).

```
if (north != -1) {
    // Send north
    MPI_Isend(up_send,local[1],MPI_DOUBLE,north,north,CART_COMM,
&send_requests[counter]);
    MPI_Irecv(up_receive,local[1],MPI_DOUBLE,north,rank,CART_COMM,
&receive_requests[counter]);
    counter++;
}
if (south != -1) {
    // Send south
}
```

```

        MPI_Isend(down_send,local[1],MPI_DOUBLE,south,south,CART_COMM,
&send_requests[counter]);
        MPI_Irecv(down_receive,local[1],MPI_DOUBLE,south,rank,CART_COMM,
&receive_requests[counter]);
        counter++;
    }
    if (west != -1) {
        // Send west
        MPI_Isend(left_send,local[0],MPI_DOUBLE,west,west,CART_COMM,
&send_requests[counter]);
        MPI_Irecv(left_receive,local[0],MPI_DOUBLE,west,rank,CART_COMM,
&receive_requests[counter]);
        counter++;
    }
    if (east != -1) {
        // Send east
        MPI_Isend(right_send,local[0],MPI_DOUBLE,east,east,CART_COMM,
&send_requests[counter]);
        MPI_Irecv(right_receive,local[0],MPI_DOUBLE,east,rank,CART_COMM,
&receive_requests[counter]);
        counter++;
    }
    MPI_Waitall(counter,receive_requests,MPI_STATUSES_IGNORE);
    MPI_Waitall(counter,send_requests,MPI_STATUSES_IGNORE);

```

Τελικά, τα δεδομένα που ελήφθησαν από τους γείτονες αντιγράφονται στους τοπικούς πίνακες και εκτελούνται οι υπολογισμοί.

Έλεγχος σύγκλισης

Για να ελέγξουμε αν το σύστημα έχει φτάσει σε σταθερή κατάσταση (δηλαδή αν η λύση έχει συγκλίνει), χρησιμοποιούμε τη συνάρτηση converge() από το αρχείο utils.h.

Αν η σύγκλιση επιτευχθεί τοπικά (σε μία διεργασία), αυτή επιστρέφει 1, διαφορετικά επιστρέφει 0.

Στη συνέχεια, χρησιμοποιούμε MPI_Allreduce με τη λειτουργία MPI_MIN, ώστε να συγκεντρώσουμε τα αποτελέσματα όλων των διεργασιών.

Αν όλες οι διεργασίες επιστρέψουν 1, σημαίνει ότι το σύστημα έχει συγκλίνει. Μόνο τότε η διαδικασία ολοκληρώνεται.

Αν έστω και μία διεργασία επιστρέψει 0, η εκτέλεση συνεχίζεται.

```

// MPI_Barrier(CART_COMM);
#ifndef TEST_CONV
if (t%C==0) {
    gettimeofday(&tss,NULL);
    //*****TODO*****
    /*Test convergence*/
    // Local conv check, converged = 1 if conv else converged = 0
    converged = converge(u_previous, u_current, i_min, i_max-1, j_min, j_max-1);
    MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN, CART_COMM);
}

```

```

    gettimeofday(&tsf,NULL);
    tconv+=(tsf.tv_sec-tss.tv_sec)+(tsf.tv_usec-tss.tv_usec)*0.000001;

}

#endif
// Just for debug on no conv test
// Error still exists even with barrier here
// MPI_Barrier(CART_COMM);

```

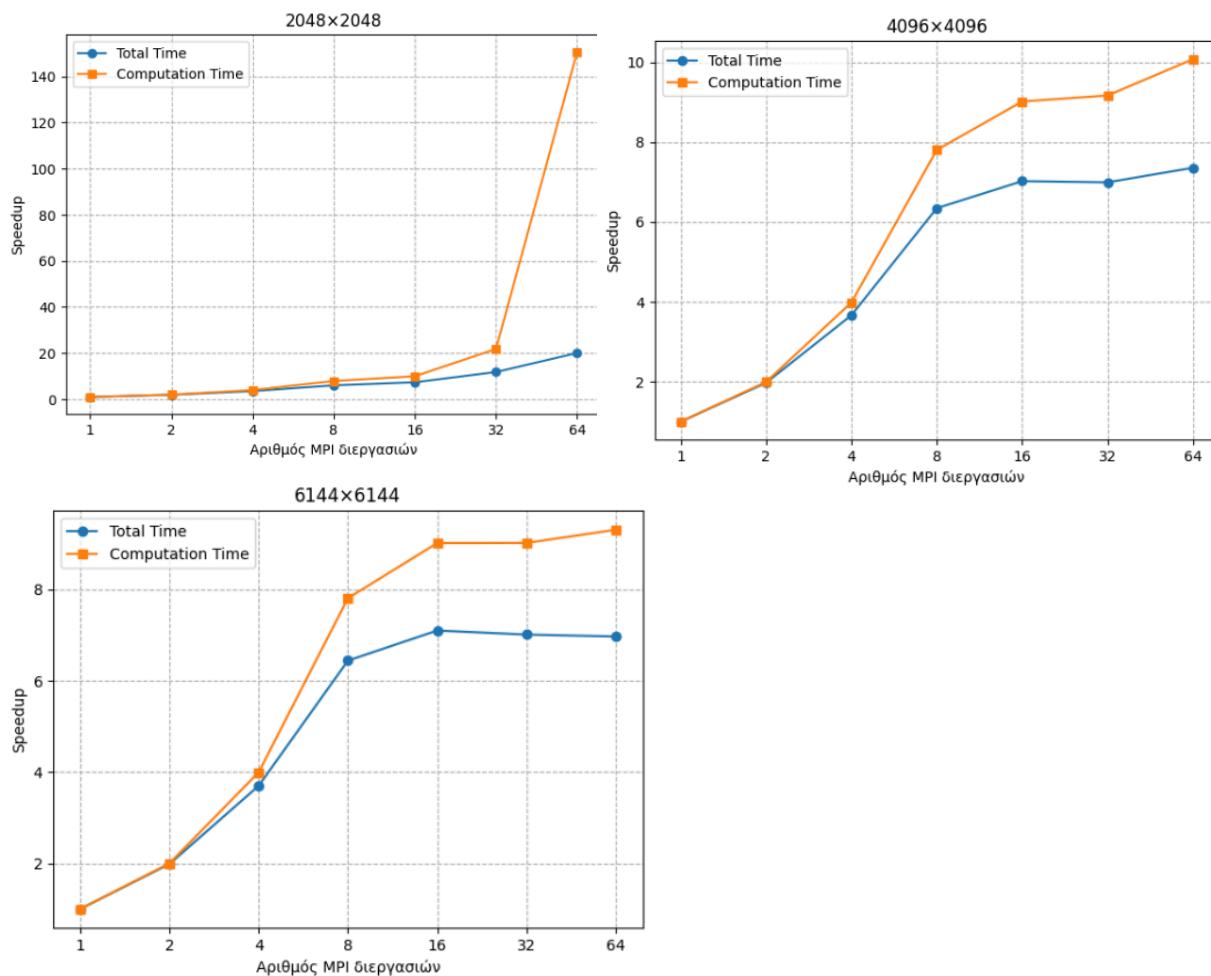
Αφού ολοκληρωθούν όλοι οι υπολογισμοί, τα δεδομένα από τα τοπικά blocks συγκεντρώνονται ξανά στον αρχικό πίνακα U μέσω της συνάρτησης MPI_Gatherv, ώστε να έχουμε το τελικό αποτέλεσμα σε μία ενιαία μορφή.

```

MPI_Gatherv(&u_current[1][1], 1, local_block, &U[0][0], scattercounts, scatteroffset,
global_block, 0, CART_COMM);

```

a)



2048x2048

Παρατηρούμε ότι το speedup του συνολικού χρόνου είναι σχετικά χαμηλό, ενώ το speedup του χρόνου υπολογισμών αυξάνεται σημαντικά για 64 διεργασίες.

4096×4096

Παρατηρούμε πιο σταθερή αύξηση του speedup και για τους δύο χρόνους. Ο υπολογιστικός χρόνος κλιμακώνεται καλύτερα σε σχέση με τον συνολικό χρόνο. Για 32 και 64 διεργασίες παρατηρούμε μείωση του συνολικού speedup, πιθανώς λόγω overhead επικοινωνίας.

6144×6144

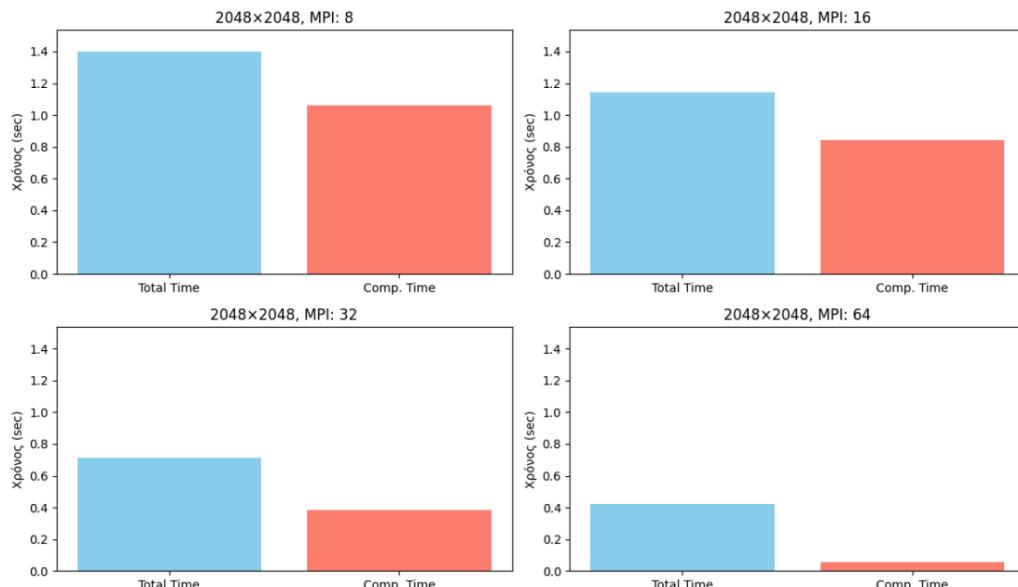
Το speedup του συνολικού χρόνου εκτέλεσης και του χρόνου υπολογισμών έχουν παρόμοια συμπεριφορά με την περίπτωση του 4096×4096. Ωστόσο, παρατηρούμε ότι το speedup σταθεροποιείται μετά από ένα σημείο, γεγονός που δείχνει ότι το κόστος επικοινωνίας υπερτερεί σε σχέση με το όφελος από την περαιτέρω αύξηση των διεργασιών.

β) Διαγράμματα barplots:

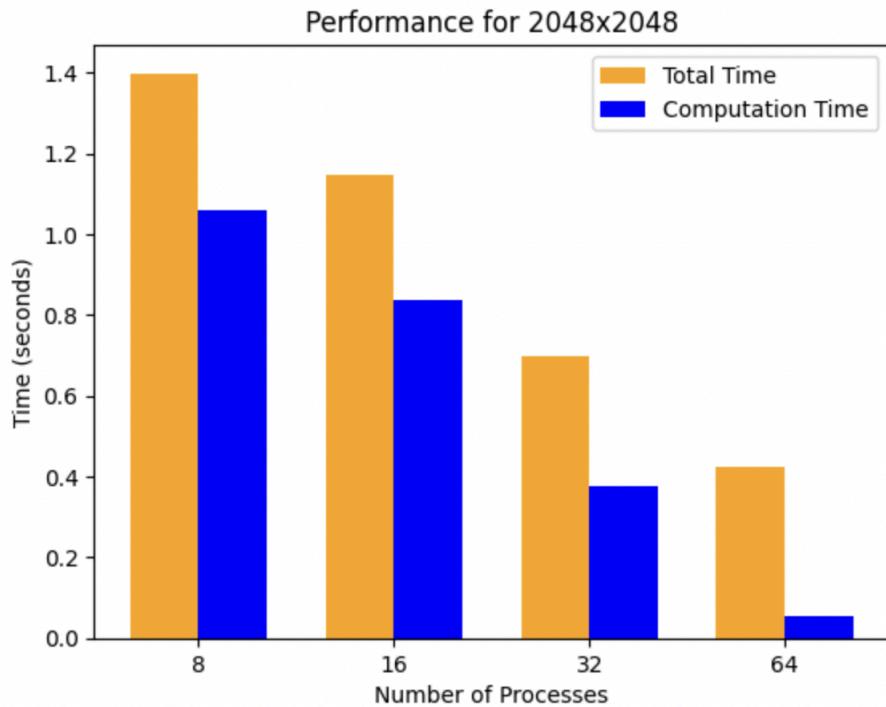
Τα παρακάτω διαγράμματα είναι όλα για την μέθοδο Jacobi.

Επικεντρωνόμαστε λίγο περισσότερο στους συνολικούς χρόνους (Total Time) και στους χρόνους υπολογισμού (Computation Time) από 8 έως και 64 διεργασίες.

Για μέγεθος πίνακα 2048x2048:



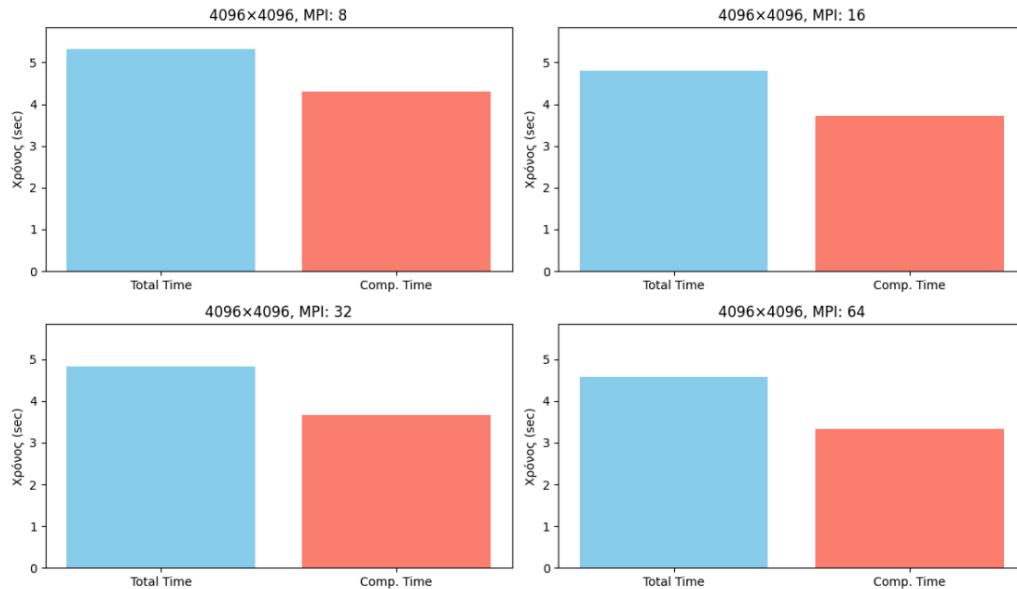
ή όλα σε κοινό διάγραμμα:



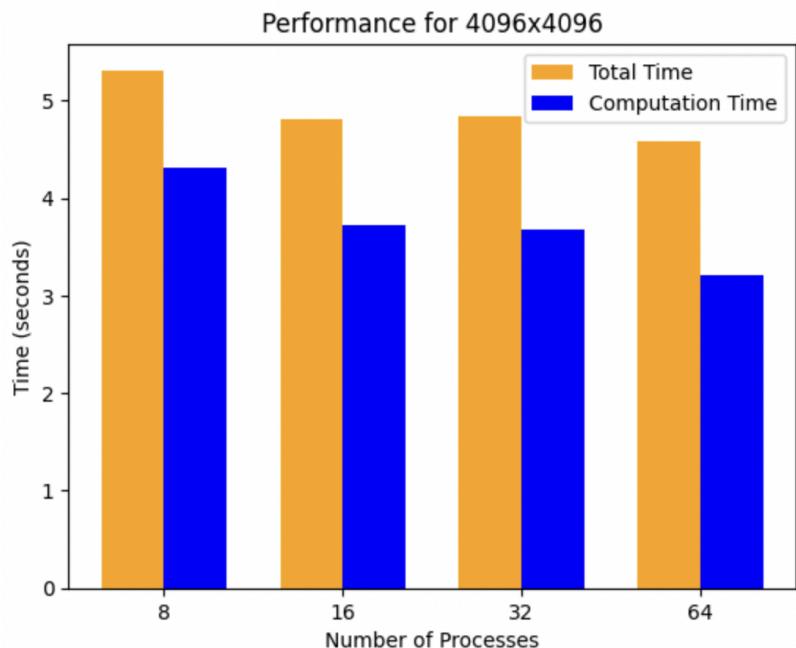
Με μια πιο “κοντινή” ματιά στους συνολικούς χρόνους, αλλά και στους χρόνους υπολογισμού, για 8, 16, 32, 64 διεργασίες, παρατηρούμε, ότι το computation time μειώνεται δραματικά σε κάθε διπλασιασμό του αριθμού των processes. Αξιοσημείωτη είναι η μεγάλη πτώση του χρόνου υπολογισμού από 32 σε 64 διεργασίες.

Ο συνολικός χρόνος μειώνεται και αυτός σε ικανοποιητικό βαθμό με την αύξηση των processes.

Για μέγεθος πίνακα 4096x4096:

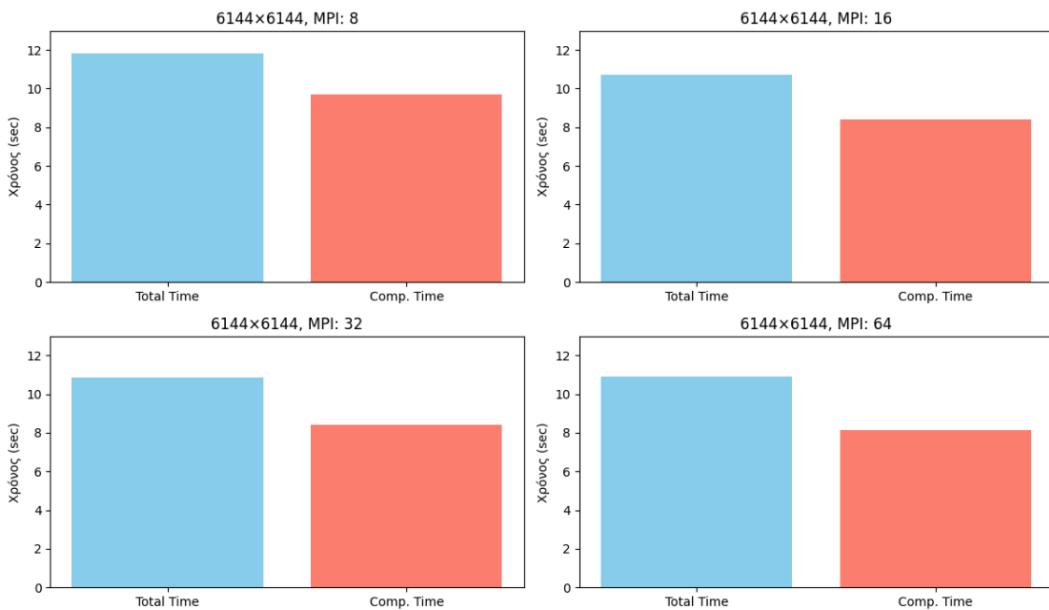


ή όλα σε κοινό διάγραμμα:

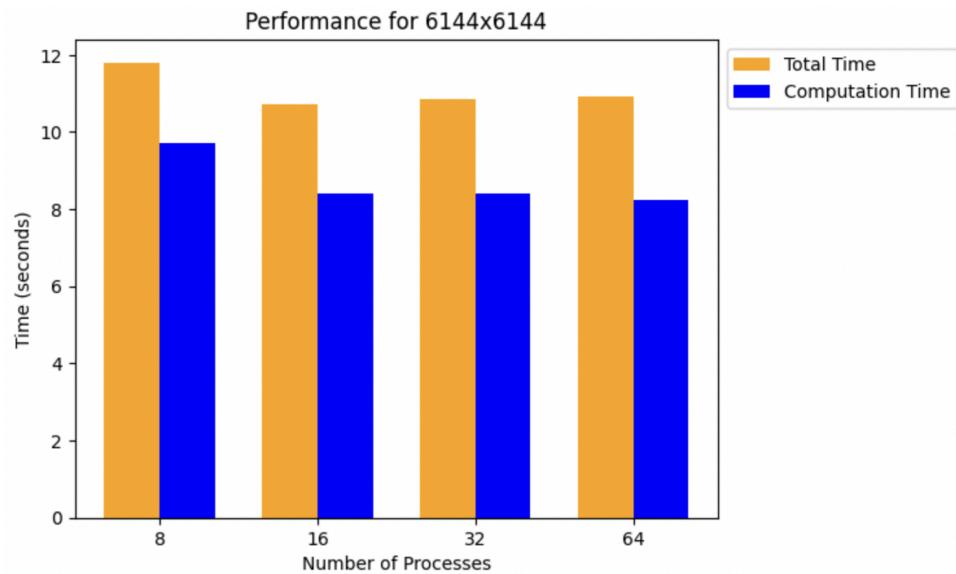


Οι συνολικοί χρόνοι (όπως και οι χρόνοι υπολογισμών) σε γενικές γραμμές παραμένουν σταθεροί, ανεξαρτήτως του αριθμού των processes. Σε αντίθεση με πριν, παρατηρούμε μια μικρή μείωση στο computation time στην αλλαγή από 32 σε 64 διεργασίες.

Για μέγεθος πίνακα 6144x6144:



ή όλα σε κοινό διάγραμμα:



Εδώ παρατηρούμε έναν “κορεσμό” των χρόνων, τόσο των συνολικών όσο και των χρόνων υπολογισμών. Οι χρόνοι για αριθμό διεργασιών από 16 και πάνω δεν μειώνονται σχεδόν καθόλου.