

# Συστήματα Παράλληλης Επεξεργασίας

Ομάδα 20

Κυριακή Ζιώγα: 03120632

Άρης Μαρκογιαννάκης: 03120085

Αθανάσιος-Παφαίλ Ρεμούνδος: 03119820

Χειμερινό Εξάμηνο 2024-2025

## Contents

<b>1 Εξοικείωση με το περιβάλλον προγραμματισμού</b>	<b>2</b>
1.1 Conway's Game of Life . . . . .	2
1.2 Αποτελέσματα + Bonus . . . . .	2
<b>2 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης</b>	<b>8</b>
2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means . . . . .	8
2.1.1 Shared clusters . . . . .	8
2.1.2 Copied clusters and reduce + Bonus . . . . .	11
2.1.3 Αμοιβαίος Αποκλεισμός - Κλειδώματα . . . . .	15
2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall . . . . .	18
2.2.1 Recursive Floyd-Warshall . . . . .	18
2.2.2 Bonus: Tiled Floyd-Warshall . . . . .	21
2.3 Ταυτόχρονες Δομές Δεδομένων . . . . .	24
<b>3 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών</b>	<b>26</b>
3.1 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means . . . . .	26
3.1.1 Naive version . . . . .	26
3.1.2 Transpose version . . . . .	29
3.1.3 Shared version . . . . .	30
3.1.4 Σύγκριση υλοποίησεων / bottleneck Analysis + Bonus-1 . . . . .	32
3.1.5 Full-Offload (All-GPU) version . . . . .	34
3.1.6 Bonus 2: Delta reduction (All-GPU) version . . . . .	38
<b>4 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης</b>	<b>43</b>
4.1 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means . . . . .	43
4.1.1 Υλοποίηση σε MPI . . . . .	43
4.1.2 Αποτελέσματα Μετρήσεων και Σχολιασμός . . . . .	44
4.1.3 Bonus: Σύγκριση με OpenMP Υλοποίηση . . . . .	45
4.2 Διάδοση θερμότητας σε δύο διαστάσεις . . . . .	46
4.2.1 Μεδοδος Jacobi . . . . .	46
4.2.2 (Bonus) Μέθοδος Gauss-Seidel SOR . . . . .	47
4.2.3 (Bonus) Μέθοδος Red-Black SOR . . . . .	47
4.2.4 Μετρήσεις με έλεγχο σύγκλισης . . . . .	48
4.2.5 Μετρήσεις χωρίς έλεγχο σύγκλισης . . . . .	49

# 1 Εξοικείωση με το περιβάλλον προγραμματισμού

## 1.1 Conway's Game of Life

Παρακάτω φαίνεται το κύριος μέρος της υλοποίησης:

```
for ( t = 0 ; t < T ; t++ ) {  
    #pragma omp parallel for shared(N, previous, current) private(i, j, nbrs)  
    for ( i = 1 ; i < N-1 ; i++ )  
        for ( j = 1 ; j < N-1 ; j++ ) {  
            nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \  
            + previous[i][j-1] + previous[i][j+1] \  
            + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];  
            if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )  
                current[i][j]=1;  
            else  
                current[i][j]=0;  
        }  
  
    #ifdef OUTPUT  
    print_to_pgm(current, N, t+1);  
    #endif  
    //Swap current array with previous array  
    swap=current;  
    current=previous;  
    previous=swap;  
}
```

Διαπιστώνουμε ότι η παραλληλοποίηση του εξωτερικού βρόχου (πρώτο for) είναι ανέφικτη, καθώς κάθε επανάληψη εξαρτάται από τα αποτελέσματα της προηγούμενης. Ωστόσο, σε κάθε δεδομένη χρονική στιγμή, ο υπολογισμός για κάθε κελί βασίζεται μόνο στα γειτονικά κελιά της ίδιας χρονικής στιγμής. Αυτό μας επιτρέπει να εφαρμόσουμε παραλληλισμό ανά γραμμή, αναθέτοντας διαφορετικές γραμμές σε ξεχωριστά νήματα. Η προσέγγιση αυτή ακολουθεί μια λογική επικεντρωμένη στα δεδομένα, καθώς κατανέμουμε τα δεδομένα μεταξύ των νημάτων.

Για την υλοποίηση της παραλληλοποίησης χρησιμοποιήθηκε το OpenMP, με την ακόλουθη εντολή:

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

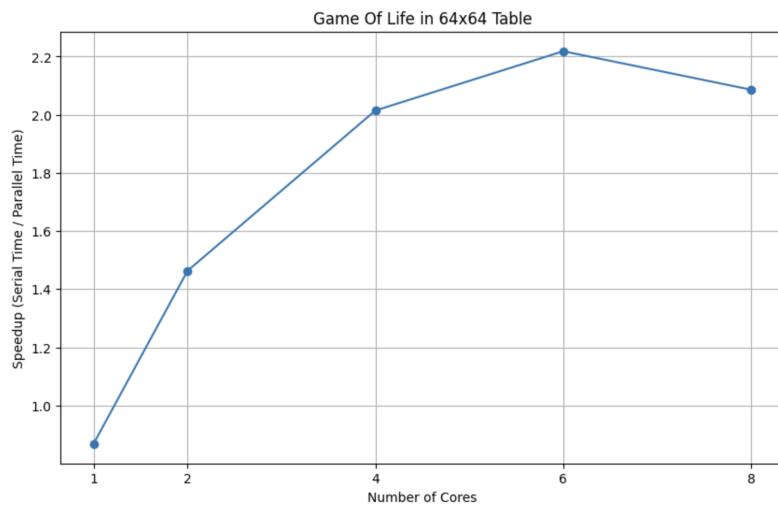
Αυτή η εντολή τοποθετείται ακριβώς πριν από τον δεύτερο βρόχο, επιτρέποντας έτσι την παράλληλη επεξεργασία των γραμμών του πίνακα.

## 1.2 Αποτελέσματα + Bonus

Σε όλες τις περιπτώσεις τρέχουμε το παιχνίδι για 1000 γενιές.

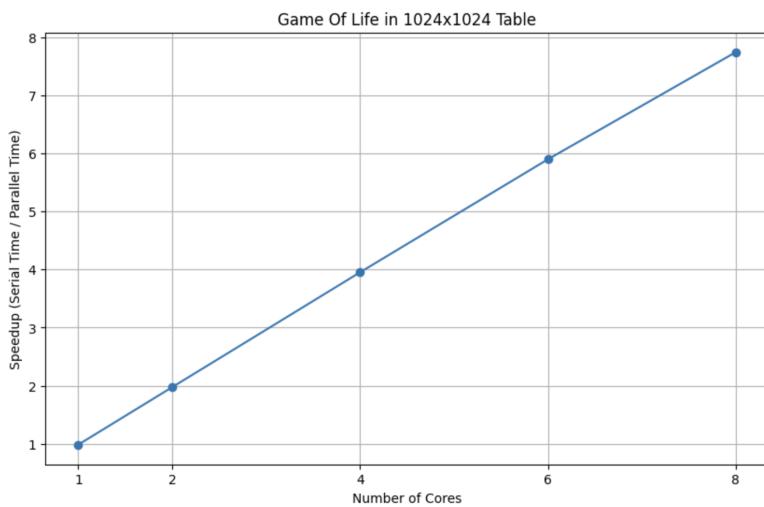
Cores \ Table Size	64 x 64	1024 x 1024	4096 x 4096
1	0.023423	10.988670	176.319766
2	0.013900	5.467626	88.485673
4	0.010096	2.729348	45.766579
6	0.009168	1.828424	43.741650
8	0.009750	1.392650	43.087714

- 64x64



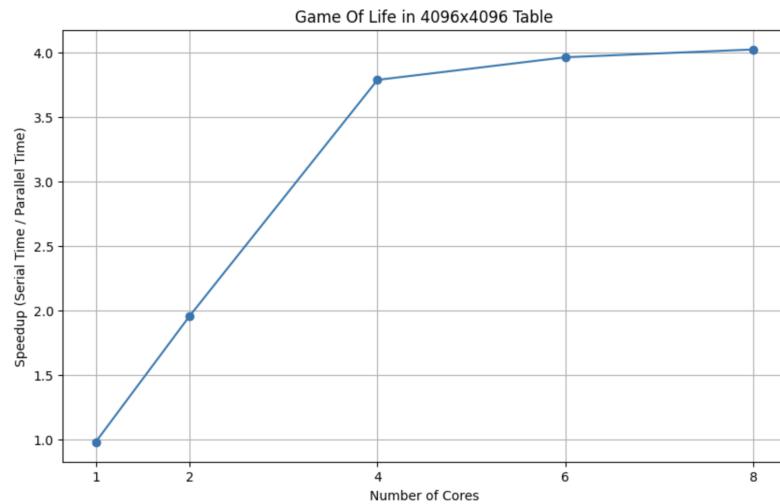
Βλέπουμε μεγάλη επιτάχυνση όταν μετακινούμαστε από 1 σε 2 πυρήνες, λίγο μικρότερη από 2 σε 4 και ακόμα πιο μικρή όταν πηγαίνουμε σε 6. Πέρα από τους 6 πυρήνες, η απόδοση πέφτει, γεγονός που υποδηλώνει ότι για ένα τόσο μικρό μέγευμος, το γενικό κόστος της παραλληλοποίησης υπερβαίνει των πλεονεκτημάτων όταν το πλήθος των πυρήνων είναι μεγαλύτερο από 6. Συνεπώς, η κλιμάκωση σταματάει στους 6 πυρήνες.

- 1024x1024



Αυτό το μέγεθος grid παρουσιάζει την καλύτερη κλιμάκωση μεταξύ των τριών μεγεθών. Υπάρχει σημαντική βελτίωση από 1 έως 2 πυρήνες και συνεχίζει να κλιμακώνεται καλά μέχρι τους 8 πυρήνες. Η επιτάχυνση είναι σχεδόν γραμμική μέχρι τους 4 πυρήνες και στη συνέχεια αρχίζει να μειώνεται, αλλά εξακολουθεί να παρουσιάζει βελτίωση μέχρι τους 8 πυρήνες.

- 4096x4096



Εδώ υπάρχει καλή κλιμάκωση από 1 έως 4 πυρήνες. Ωστόσο, μετά τους 4 πυρήνες, υπάρχει ελάχιστη βελτίωση. Αυτό πιθανώς οφείλεται στους περιορισμούς του εύρους ζώνης της μνήμης και στις επιδράσεις της χρυφής μνήμης. Δημιουργήσαμε λοιπόν το αρχείο *test.sh* το οποίο φαίνεται παρακάτω:

```
#!/bin/bash
## Give the Job a descriptive name
#PBS -N test

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

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

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

```
lscpu
```

```
echo "\n\nTest is completed successfully\n\n"
```

```
parlab20@scirouter:~/a1$ cat test.sh.o568694
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):              2
NUMA node(s):           1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                 15
Model name:             Intel(R) Xeon(R) CPU          E5335 @ 2.00GHz
Stepping:               7
CPU MHz:                2000.101
BogoMIPS:               4036.76
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               4096K ←
NUMA node0 CPU(s):      0-7
\n\nTest is completed successfully\n\n
parlab20@scirouter:~/a1$
```

Μέσω της εντολής lscpu μας παρέχεται πληροφορία σχετικά με τα χαρακτηριστικά της αρχιτεκτονικής των ερυ και των μηχανημάτων που τρέζαμε το πρόγραμμα μας. Οπότε για array size ίσο με 1024 όπως στην προηγούμενη περίπτωση το μέγεθος των δεδομένων είναι  $1024 \cdot 1024 \cdot 4$ (επειδή integers) = 4194304 bytes που ισοδυναμούν με 4096K, δηλαδή χωράνε ακριβώς στην L2 cache. Αντίθετα, σε αυτή την περίπτωση έχουμε  $4096 \cdot 4096 \cdot 4 > 4096K$  που δεν χωράνε οπότε πρέπει να πραγματοποιηθούν περισσότερες από μία προσβάσεις στη μνήμη.

Συνεπώς, το grid 1024x1024 παρουσιάζει την καλύτερη συνολική κλιμάκωση. Αυτό το μέγεθος είναι αρκετά μεγάλο ώστε να επωφεληθεί από τον παραλληλισμό, αλλά αρκετά μικρό ώστε να χωράει καλά στην κρυφή μνήμη cache για τα περισσότερα συστήματα. Το grid 64x64 είναι πολύ μικρό για να αξιοποιήσει αποτελεσματικά πολλούς πυρήνες, με την επιβάρυνση παραλληλισμού να γίνεται σημαντική. Τέλος, το grid 4096x4096 είναι αρκετά μεγάλο για να επωφεληθεί από την παραλληλοποίηση, αλλά προσκρούει σε όρια εύρους ζώνης μνήμης, εμποδίζοντας τη γραμμική κλιμάκωση πέραν των 4 πυρήνων.

**Bonus:** Τέλος υλοποιήσαμε δύο αρχικοποιήσεις του ταξιπλώ που οδηγούν σε ενδιαφέροντα οπτικά αποτελέσματα. Τα αντίστοιχα gifs βρίσκονται στους ακόλουθους συνδέσμους: pulsar.gif, blinker.gif. Παρακάτω παραθέτουμε και τους αντίστοιχους κώδικες.

```
void init_pulsar(int ** array1, int ** array2, int N) {
    if (N < 17) { // Pulsar needs at least 17x17 grid to fit.
        fprintf(stderr, "Grid size too small for Pulsar. Use N >= 17.\n");
        exit(-1);
    }

    int mid = N / 2; // Center of the grid

    // Coordinates relative to the center for one quadrant of the pulsar.
    int pulsar_pattern[12][2] = {
        {0, 2}, {0, 3}, {0, 4},
        {5, 2}, {5, 3}, {5, 4},
        {2, 0}, {3, 0}, {4, 0},
        {2, 5}, {3, 5}, {4, 5}
```

```

};

int i;
// Set the pulsar pattern in all four quadrants symmetrically.
for ( i = 0; i < 12; i++) {
    int x = pulsar_pattern[i][0];
    int y = pulsar_pattern[i][1];

    // Upper-left quadrant
    array1[mid - x][mid - y] = 1;
    array2[mid - x][mid - y] = 1;

    // Upper-right quadrant
    array1[mid - x][mid + y] = 1;
    array2[mid - x][mid + y] = 1;

    // Lower-left quadrant
    array1[mid + x][mid - y] = 1;
    array2[mid + x][mid - y] = 1;

    // Lower-right quadrant
    array1[mid + x][mid + y] = 1;
    array2[mid + x][mid + y] = 1;
}

}

void init_blinker(int ** array1, int ** array2, int N) {
if (N < 5) { // Blinker needs at least 5x5 grid to fit.
    fprintf(stderr, "Grid size too small for Blinker. Use N >= 5.\n");
    exit(-1);
}

int mid = N / 2; // Center of the grid

// Set the Blinker pattern in the middle
array1[mid][mid - 1] = 1;
array1[mid][mid] = 1;
array1[mid][mid + 1] = 1;

// Initialize the previous array the same way
array2[mid][mid - 1] = 1;
array2[mid][mid] = 1;
array2[mid][mid + 1] = 1;
}

```

## 2 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

### 2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

#### 2.1.1 Shared clusters

1. Έχουμε υλοποιήσει δύο τρόπους παραλληλοποίησης για αυτή την έκδοση του αλγορίθμου. Στην πρώτη υλοποίηση θεωρούμε την μεταβλητή delta ως μοιραζόμενη (shared) μεταξύ των διεργασιών και χρησιμοποιούμε το εξής directive:

```
#pragma omp atomic
```

για τον συγχρονισμό των threads. Στην δεύτερη υλοποίηση θεωρούμε την μεταβλητή delta ως private οπότε κάθε nόμα υπολογίζει τοπικά τη τιμή του delta και στο τέλος γίνεται reduction. Αυτό πραγματοποείται με το ακόλουθο directive:

```
#pragma omp parallel for private(i, j, index) reduction(+:delta)
```

Παρακάτω φαίνεται ο αντίστοιχος κώδικας.

```
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(i, j, index) reduction(+:delta)
    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) {
            // #pragma omp atomic
            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
     */
```

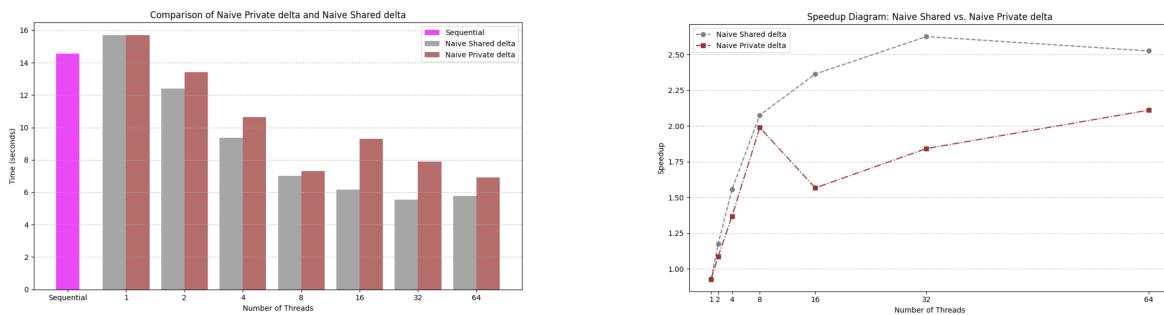
```

#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];
        }
    }
}

```

Με κατάλληλη αφαίρεση σχολίων μπορούμε να εναλλασόμαστε μεταξύ των δύο υλοποιήσεων.



Παρατηρούμε ότι η υλοποίηση με το reduction delta κλιμακώνει καλύτερα, έχει καλύτερο speedup όπως φαίνεται και από το διάγραμμα με μέγιστη τιμή μεγαλύτερη από 2.5. Σε αντίθεση με την υλοποίηση με το reduction delta, στην υλοποίηση με το shared delta πετυχαίνουμε χειρότερη κλιμάκωση κάτι που είναι αναμενόμενο καθώς η υλοποίηση αυτή έχει περαιτέρω καθυστέρηση λόγω ανάγκης συγχρονισμού των νημάτων. Σε κάθε περίπτωση και οι δύο υλοποιήσεις έχουν καλύτερο χρόνο από τον σειριακό εκτός από την περίπτωση που χρησιμοποιούμε μόνο ένα νήμα. Παρόλα αυτά, ο χρόνος δεν μειώθηκε αισθητά - μέγιστο συνολικό speedup 2.5.

2. Το GOMP\_CPU\_AFFINITY είναι μια μεταβλητή περιβάλλοντος που χρησιμοποιείται στην υλοποίηση του OpenMP από το GNU για να ελέγχει τη δέσμευση (binding) των threads σε συγκεκριμένους πυρήνες CPU. Με τη ρύθμιση αυτής της μεταβλητής, μπορούμε να καθορίσουμε ποιοι πυρήνες CPU θα χρησιμοποιηθούν από κάθε thread, κάτι που μπορεί να βελτιώσει την απόδοση μειώνοντας τις αλλαγές πλαισίου (context switching) και βελτιστοποιώντας τη χρήση της προσωρινής μνήμης (cache). Η σύνταξη αυτής της μεταβλητής περιλαμβάνει λίστες αριθμών πυρήνων, οι οποίες μπορεί να είναι μεμονωμένοι αριθμοί, εύρη ή εύρη με βήμα (stride).

Για να αποφασίσουμε πως θα ενώσουμε τα νήματα με τους πηρύνες τρέξαμε την εντολή `lscpu` στον `sandman` ώστε να πάρουμε περισσότερες πληροφορίες για το μηχάνημα. Το αποτέλεσμα της εντολής αυτής φαίνεται παρακάτω.

Τροποποιούμε, λοιπόν, το σκριπτάκι `run_on_queue.sh` ώστε να λαμβάνει υπόψην του το cpu affinity ως εξής:

```

# Define the CPU lists per NUMA node for GOMP_CPU_AFFINITY
numa_node_cpus=(

    "0-7,32-39"      # NUMA node 0
    "8-15,40-47"      # NUMA node 1
    "16-23,48-55"     # NUMA node 2
    "24-31,56-63"     # NUMA node 3
)

```

```

parLab2@psctrouter:-/az/kmeans/test/out$ cat lscpu.out
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                64
On-line CPU(s) list:  0-63
Thread(s) per core:   2
Core(s) per socket:   8
Socket(s):             4
NUMA node(s):          4
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Model name:            Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz
Stepping:              7
CPU MHz:               1400.000
CPU max MHz:           2201.0000
CPU min MHz:           1200.0000
BogoMIPS:              4400.42
Virtualization:        VT-x
L1 cache:              32K
L1.5 cache:            32K
L2 cache:              256K
L3 cache:              16384K
NUMA node0 CPU(s):    0-7,32-39
NUMA node1 CPU(s):    8-15,40-47
NUMA node2 CPU(s):    16-23,48-55
NUMA node3 CPU(s):    24-31,56-63
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
                     ss ht tm pbe syscall mtrr pdpe1gb rdtsvp lm constant_tsc arch_perfmon pebs bts rep_good nopl xttopology nonstop_tsc aperfmpref pml
                     pmlnudq dtes64 monitor ds_cpl vnx snx est tm2 ssse3 cx16 xtrp pdcm pcld dca ssse4_1 ssse4_2 xzepc popcnt tsc_deadline_timer aes
                     xsave avx lahf_lm eph kaiser tpr_shadow vnm flflexpriority ept vpid xsaveopt dtherm ida arat pin pts
parLab2@psctrouter:-/az/kmeans/test/out$ []

```

```

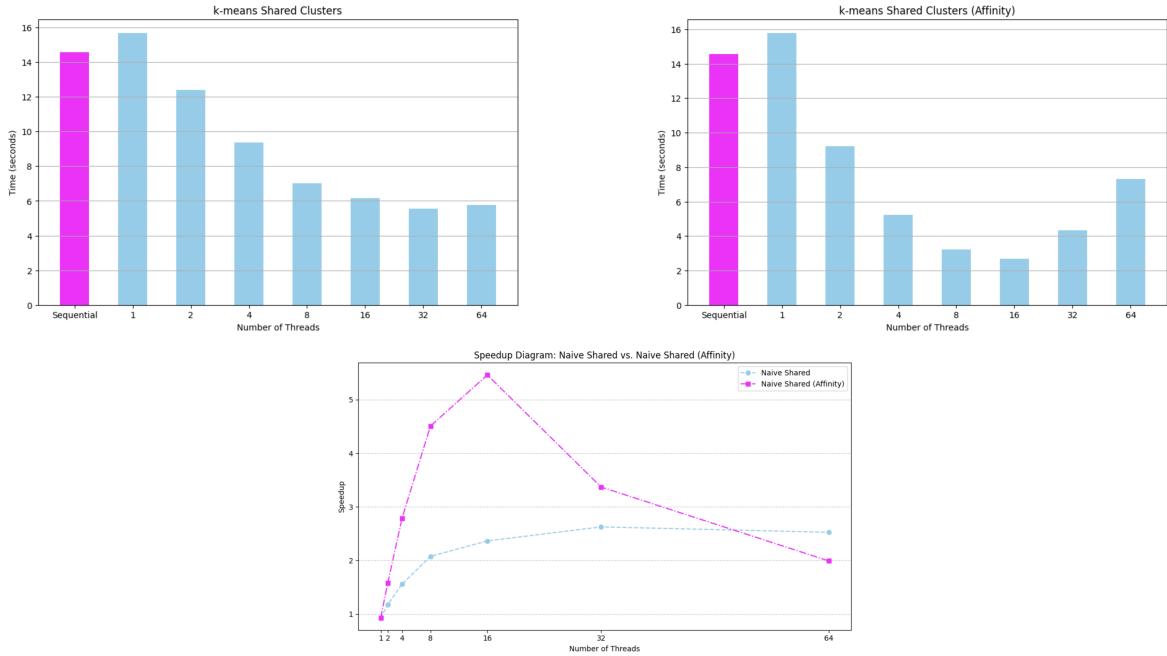
# Loop through each core count
for cores in "${core_counts[@]}"
do
    # Set the number of OpenMP threads
    export OMP_NUM_THREADS=$cores

    # Choose the appropriate NUMA node CPUs based on the core count
    if [ $cores -le 16 ]; then
        export GOMP_CPU_AFFINITY="${numa_node_cpus[0]}"
    elif [ $cores -le 32 ]; then
        export GOMP_CPU_AFFINITY="${numa_node_cpus[0]},${numa_node_cpus[1]}"
    elif [ $cores -le 48 ]; then
        export GOMP_CPU_AFFINITY="${numa_node_cpus[0]},${numa_node_cpus[1]},"
        ${numa_node_cpus[2]}"
    else
        export GOMP_CPU_AFFINITY="${numa_node_cpus[0]},${numa_node_cpus[1]},"
        ${numa_node_cpus[2]},"
        ${numa_node_cpus[3]}"
    fi
done

```

Συνεπώς, όταν το τρέχουμε με 16 πυρήνες ή λιγότερους τα βάζουμε να τρέξουν όλα στον numa node 0. Όταν το τρέχουμε από 17 πυρήνες έως 32 τότε τα βάζουμε να τρέξουν στους numa node 0 και 1. Παρόμοια διαδικασία ακολουθούμε και για τα υπόλοιπα πλήθη πυρήνων όπως φαίνεται στον παραπάνω κώδικα.

Παρακάτω φαίνονται τα αντίστοιχα διαγράμματα:



Στην υλοποίηση με affinity παρατηρούμε εώς και πενταπλάσια επίσοδη σε σχέση τη σειριακή. Μέχρι και 32 πυρήνες ο χρόνος είναι αισθητά καλύτερος με την κορύφωση να είναι στους 8 πυρήνες σε αντίθεση με την υλοποίηση χωρίς affinity. Η μόνη περίπτωση στην οποία η υλοποίηση χωρίς affinity είναι καλύτερη από αυτήν με affinity είναι όταν έχουμε 64 πυρήνες. Αυτό συμβαίνει, όπως αναφέραμε και παραπάνω, λόγω λιγότερων context switches και καλύτερης χρήσης της cache.

### 2.1.2 Copied clusters and reduce + Bonus

- Η δεύτερη και πιο εξελιγμένη έκδοση του κώδικα ακολουθεί μια εντελώς διαφορετική φιλοσοφία από την προηγούμενη. Συγκεκριμένα χρησιμοποιούμε τοπικούς (copied) πίνακες local\_newClusters και local\_newClusterSize για κάθε νήμα, ώστε να μπορεί να γίνεται η εγγραφή χωρίς συγχρονισμό. Μετά το πέρας των τοπικών υπολογισμών τα αποτελέσματα των τοπικών αυτών πινάκων συνδυάζονται (reduction) στους πίνακες newClusters και newClusterSize.

```
#pragma omp parallel
{
    int thread_num = omp_get_thread_num();

    /*
     * TODO: Initialize local cluster data to zero (separate for each thread)
     */
    for (i = 0; i < numClusters; i++) {
        for (j = 0; j < numCoords; j++)
            local_newClusters[thread_num][i * numCoords + j] = 0.0;
        local_newClusterSize[thread_num][i] = 0;
    }

    #pragma omp for private(i, j, index) reduction(+:delta)
    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;
    }
}
```

```

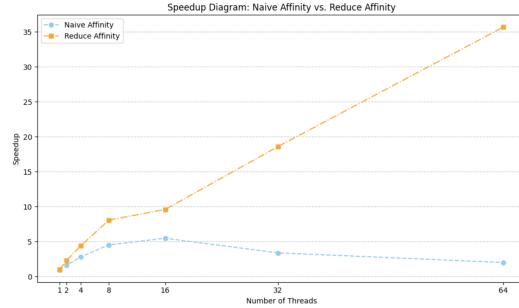
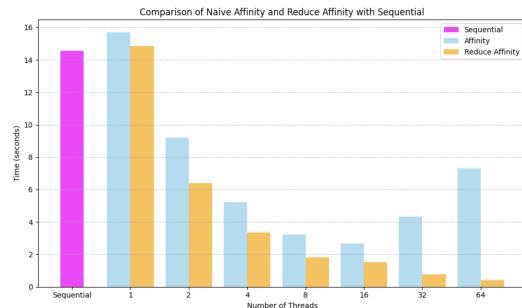
// assign the membership to object i
membership[i] = index;

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

/*
 * TODO: Reduction of cluster data from local arrays to shared.
 * This operation will be performed by one thread
*/
#pragma omp single
for (k = 0; k < nthreads; k++) {
    for (i = 0; i < numClusters; i++) {
        newClusterSize[i] += local_newClusterSize[k][i];
        for (j = 0; j < numCoords; j++)
            newClusters[i * numCoords + j] += local_newClusters[k][i * numCoords + j];
    }
}
}

```

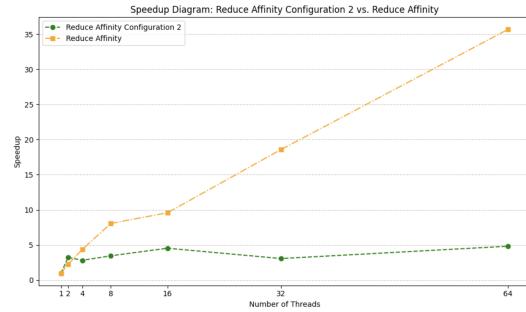
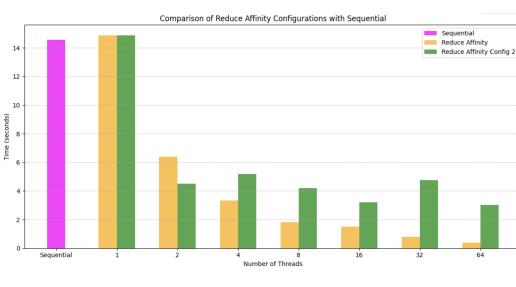
Παρακάτω φαίνονται τα αντίστοιχα διαγράμματα.



Συγκρίντας αυτή την έκδοση του κώδικα με την προηγούμενη φαίνεται ότι η χρήση μοιραζόμενων μεταβλητών newClusters και newClusterSize και η χρήση ατομικών εντολών - για τον συγχρονισμό των εγγραφών μεταξύ των νημάτων - επιφέρει σημαντικό κόστος από άποψη χρόνου. Σε αντίθεση με την προηγούμενη έκδοση, η νέα υλοποίηση επιτυγχάνει πολύ καλύτερη κλιμάκωση - με κορυφαία επίδοση αυτή των 64 πυρήνων.

2. Δοκιμάζουμε το ακόλουθο configuration Size, Coords, Clusters, Loops = {256, 1, 4, 10}, και για να συγκρίνουμε το scalability με το προηγούμενο {Size, Coords, Clusters, Loops} = {256, 16, 32, 10} δημιουργήσαμε τα παρακάτω διαγράμματα.

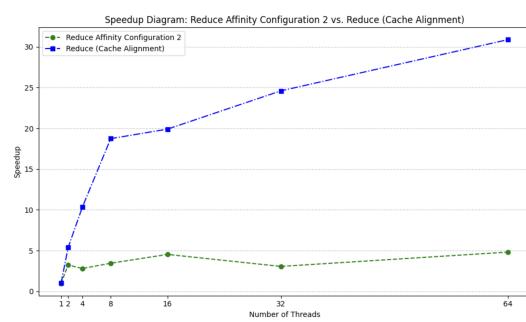
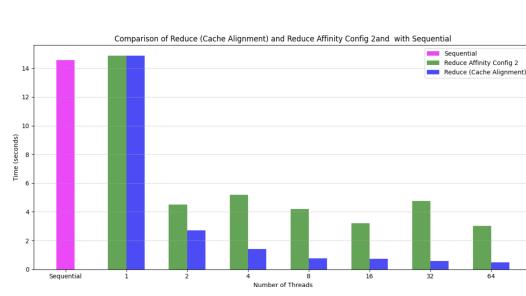
Παρατηρούμε ότι συγκριτικά με την προηγούμενη έκδοση έχουμε πολύ χαμηλότερη κλιμάκωσιμότητα και επίδοση. Αυτό το γεγονός υπονθέτουμε ότι οφείλεται στο ότι το thread που κάνει initialize τους copied πίνακες localNewClusters τους κάνει allocate στην κοντινότερη σε αυτόν cache. Συγκεκριμένα, το φαινόμενο αυτό αντιστοιχεί στην πολιτική First-Touch όπου αποφεύγεται το False Sharing. Παρακάτω φαίνεται το αντίστοιχο κομμάτι κώδικα:



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

**Πολιτική First-Touch.** Η πολιτική First-Touch είναι μια στρατηγική κατανομής μνήμης, όπου τα δεδομένα κατανέμονται στη μονάδα επεξεργασίας που τα προσπελαύνει πρώτη. Αυτή η προσέγγιση βελτιστοποιεί την τοπικότητα των δεδομένων διασφαλίζοντας ότι τα δεδομένα βρίσκονται στον κόμβο που τα χρειάζεται πρώτος, μειώνοντας ενδεχομένως το κόστος μεταφοράς δεδομένων σε κατανεμημένα συστήματα. Ευθυγραμμίζοντας τα δεδομένα με τη μονάδα που τα χρησιμοποιεί πρώτη, μπορεί να επιτευχθεί καλύτερη αξιοποίηση της μνήμης cache και μείωση της καθυστέρησης.

**False Sharing.** Το False Sharing είναι ένα μοτίβο χρήσης που μειώνει την απόδοση σε συστήματα με κατανεμημένες και συνεκτικές cache. Προκύπτει όταν πολλαπλοί επεξεργαστές ενημερώνουν ξεχωριστά στοιχεία μέσα στην ίδια γραμμή cache, οδηγώντας σε άσκοπη ακύρωση ολόκληρης της γραμμής cache. Αυτή η ακύρωση αναγκάζει τους επεξεργαστές να ανακτούν τη νεότερη έκδοση της γραμμής από τη μνήμη, ακόμα κι αν τα στοιχεία που προσπελάστηκαν δεν έχουν τροποποιηθεί. Το αποτέλεσμα είναι η αύξηση της κυκλοφορίας στον διασύνδεσμο και του γενικού φόρτου, γεγονός που μπορεί να υποβαθμίσει σημαντικά την απόδοση και την κλιμάκωση των παράλληλων εφαρμογών.



Τα ίδια αποτελέσματα θα μπορούσαμε να πετύχουμε εφαρμόζοντας padding. Αρχικά, παρατηρούμε ότι στο δεύτερο configuration έχουμε 4 clusters και 1 διάσταση και ταυτόχρονα βλέπουμε στον κώδικα το εξής:

```
double * local_newClusters[nthreads]; // [nthreads][numClusters][numCoords]
```

Αυτό σημαίνει ότι `local_newClusters[0]` έχει μέγεθος 16 bytes με  $4(\text{Clusters}) \cdot 1(\text{Coords}) \cdot 1(\text{double})$ . Αφού κάθε `double` έχει μέγεθος 8 bytes, το συνολικό size ισούται με 32 bytes. Για να μάθουμε το μέγεθος της cache line τρέξαμε την εντολή `getconf -a — grep CACHE` η οποία μας έβγαλε τα παρακάτω αποτέλεσμα:

LEVEL1_ICACHE_SIZE	32768
LEVEL1_ICACHE_ASSOC	8
LEVEL1_ICACHE_LINESIZE	64
LEVEL1_DCACHE_SIZE	32768
LEVEL1_DCACHE_ASSOC	8
LEVEL1_DCACHE_LINESIZE	64
LEVEL2_CACHE_SIZE	262144
LEVEL2_CACHE_ASSOC	8
LEVEL2_CACHE_LINESIZE	64
LEVEL3_CACHE_SIZE	16777216
LEVEL3_CACHE_ASSOC	16
LEVEL3_CACHE_LINESIZE	64
LEVEL4_CACHE_SIZE	0
LEVEL4_CACHE_ASSOC	0
LEVEL4_CACHE_LINESIZE	0

Παρατηρούμε ότι η LEVEL1\_DCACHE\_LINESIZE είναι 64 bytes. Συνεπώς, σε μία cache line χωράνε 2 τοπικές μεταβλητές local\_newClusters. Αυτό έχει ως αποτέλεσμα δύο νήματα να μοιράζονται την ίδια cache line - το ένα όμως ασχολείται πρακτικά με το πρώτο μισό ενώ το δεύτερο με το άλλο μισό. Έτσι, όταν ένα νήμα πάει να γράψει στη μεταβλητή local\_newClusters όταν κάνει το αντιστοιχο νήμα που μοιράζεται την ίδια cache line invalid - κάτι που αυξάνει την καυνυστέρηση καθώς στην επόμενη επανάληψη το δεύτερο νήμα χρειάζεται να πάει στη μνήμη. Αυτό όμως μπορούσε μακριά νήματα να αποφευχθεί αν κάθε νήμα το αντιστοιχούσαμε σε μία διαφορετική cache line. Συνεπώς, αφού έχουμε μέχρι 64 νήματα για να το πετύχουμε αυτό όμως πρέπει κάθε νήμα να απέχει από τα υπόλοιπα  $31 \cdot 64 + 32 = 2016$  bytes. Οπότε εφαρμόζοντας padding 2016 bytes πετυχαίνουμε το ίδιο αποτέλεσμα που πετύχαμε και παραπάνω.

(Bonus) 3. Δοκιμάζουμε να βελτιώσουμε την επίδοση του προγράμματος, λαμβάνοντας υπόψη τα NUMA χαρακτηριστικά του μηχανήματος και το διαμοιρασμό του πίνακα objects στα memory nodes, ο οποίος αρχικοποιείται στο αρχείο "file\_io.c". Συγκεκριμένα, χρησιμοποιούμε την First-Touch πολιτική που αναφέραμε παραπάνω ώστε κάθε νήμα να φέρνει τα objects στη δική του κοντινότερη cache. Παρακάτω παραθέτουμε τον κώδικα του αρχείου "file\_io.c".

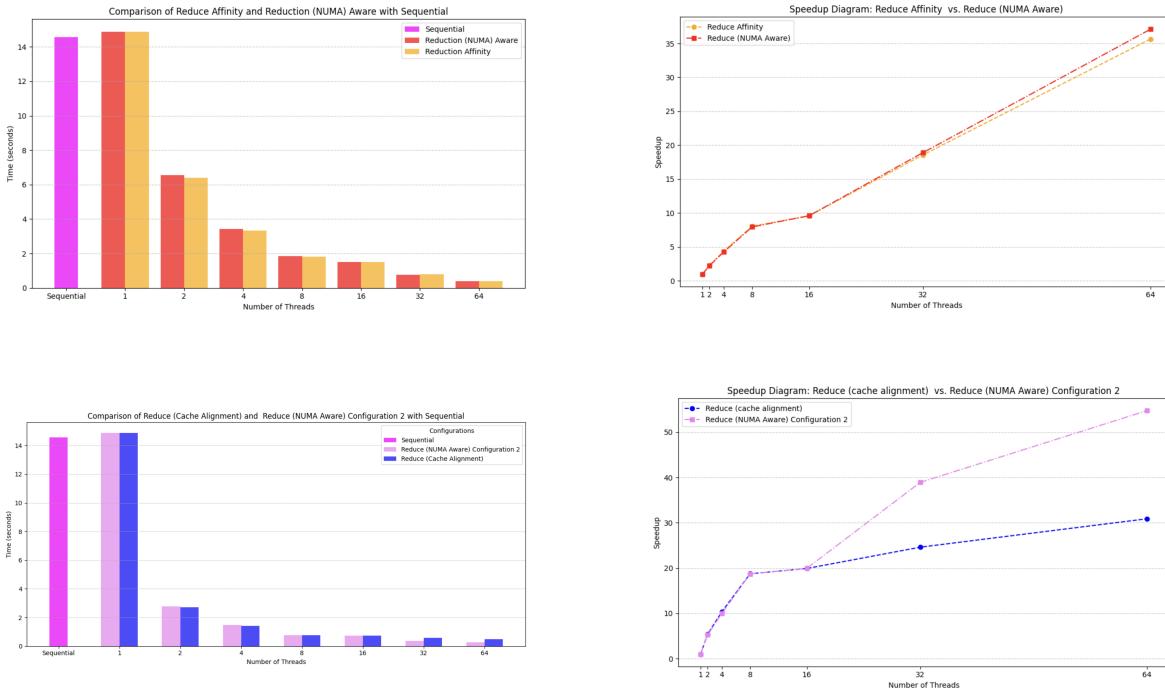
```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();

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

Αξίζει να σημειωθεί ότι χρειάζεται να μοιράσουμε ίσο αριθμό από objects σε κάθε νήμα - κάτι που πετυχαίνουμε με το παρακάτω directive:

```
#pragma omp for schedule(static)
```

Παρακάτω φαίνονται τα αντίστοιχα διαγράμματα.



Παρατηρούμε ότι η NUMA Aware τεχνική έχει μεγαλύτερη απόδοση στο configuration 2 όπου πετυχαίνει κλιμάκωση μεγαλύτερη από 50.

### 2.1.3 Αμοιβαίος Αποκλεισμός - Κλειδώματα

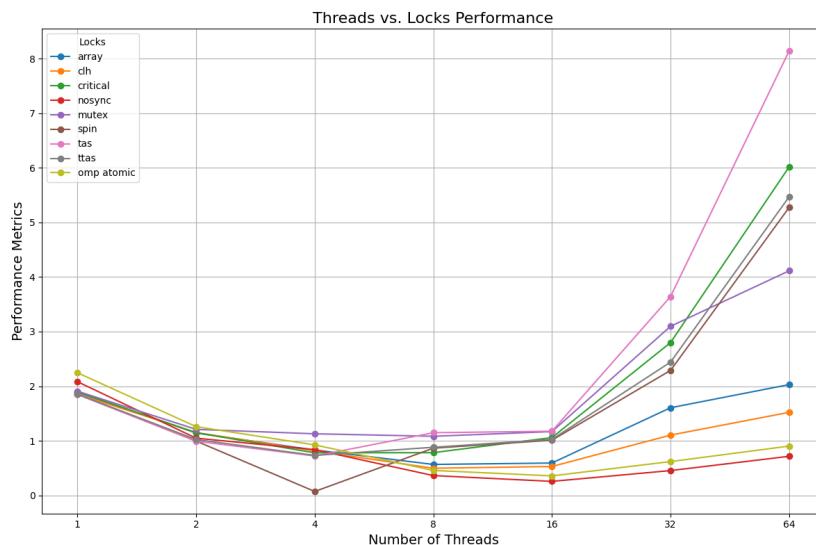
1. Στην υλοποίηση "shared-clusters" του αλγορίθμου K-means, πολλαπλά νήματα ενημερώνουν ταυτόχρονα μια κοινόχρηστη δομή δεδομένων που ονομάζεται newClusters. Για να διασφαλιστεί η ορθότητα, απαιτείται αμοιβαίος αποκλεισμός όταν τα νήματα έχουν πρόσβαση σε αυτό το κρίσιμο τμήμα. Διάφορες υλοποιήσεις κλειδωμάτων παρέχουν αυτόν τον αμοιβαίο αποκλεισμό, καθεμία με διαφορετικά χαρακτηριστικά απόδοσης. Συγχρίνουμε διάφορους τύπους κλειδωμάτων—nosync\_lock, pthread\_mutex\_lock, pthread\_spin\_lock, tas\_lock, ttas\_lock, array\_lock, clh\_lock, omp critical και omp atomic—για να κατανοήσουμε τον αντίκτυπό τους στην απόδοση σε ένα πολυνηματικό περιβάλλον. Συγκεντρώνουμε 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 στο μηχάνημα sandman (εφαρμόζοντας thread-binding).

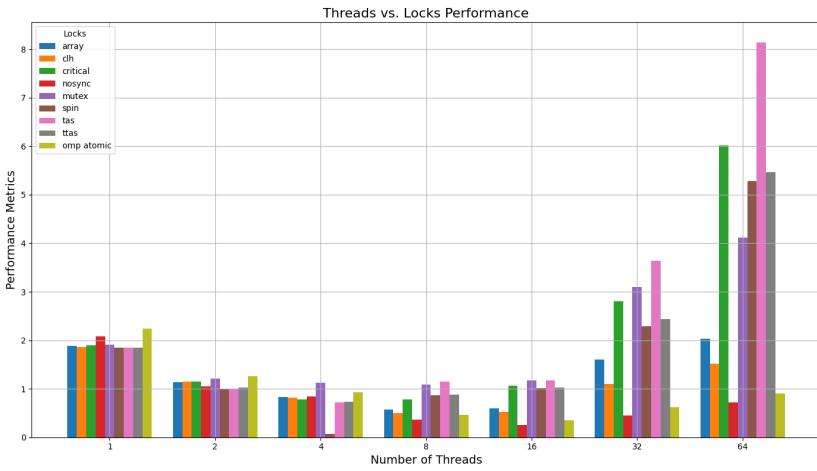
**Λειτουργία κάθε κλειδώματος και ο αντίκτυπός της στην απόδοση.**

- Το nosync\_lock δεν παρέχει κανέναν αμοιβαίο αποκλεισμό. Τα νήματα έχουν πρόσβαση στον κοινόχρηστο πόρο χωρίς κανέναν συγχρονισμό, με αποτέλεσμα εσφαλμένα αποτελέσματα λόγω race conditions. Συνεπώς, παρουσιάζει τους χαμηλότερους χρόνους εκτέλεσης επειδή δεν υπάρχει επιβάρυνση συγχρονισμού, παρουσιάζοντας καλή κλιμάκωση καθώς αυξάνεται ο αριθμός των νημάτων.
- Το pthread\_mutex\_lock είναι ένα κλείδωμα mutex που παρέχεται από τη βιβλιοθήκη Pthreads, εξασφαλίζοντας αμοιβαίο αποκλεισμό επιτρέποντας μόνο σε ένα νήμα να κατέχει το κλείδωμα κάθε φορά. Τα κλειδώματα mutex περιλαμβάνουν κλήσεις συστήματος και μπορούν να προκαλέσουν αλλαγές περιβάλλοντος, οδηγώντας σε μεγαλύτερη επιβάρυνση. Αποδίδουν καλά με λιγότερα νήματα, αλλά υποφέρουν καθώς αυξάνεται ο αριθμός των νημάτων λόγω αυξημένης διεκδίκησης και αποκλεισμού.
- Το pthread\_spin\_lock είναι ένα κλείδωμα περιστροφής από τη βιβλιοθήκη Pthreads, όπου τα νήματα που προσπαθούν να αποκτήσουν το κλείδωμα θα περιμένουν ενεργά (περιστροφή) μέχρι να γίνει διαθέσιμο. Αποδίδει καλύτερα από τα mutex όταν ο χρόνος κράτησης του κλειδώματος είναι μικρός, επειδή αποφεύγει αλλαγές περιβάλλοντος. Ωστόσο, με πολλά νήματα, η περιστροφή μπορεί να σπαταλά κάπλους CPU, ειδικά αν το κλείδωμα είναι περιζήτητο.

- Το `tas_lock` (Κλείδωμα Test-and-Set) είναι ένα απλό κλείδωμα περιστροφής που χρησιμοποιεί την ατομική εντολή test-and-set για να αποκτήσει το κλείδωμα. Το κλείδωμα αρχικοποιείται σε μια τιμή που υποδηλώνει ότι είναι ελεύθερο (συνήθως 0). Όταν ένα νήμα θέλει να έχει πρόσβαση στο κρίσιμο τμήμα εκτελεί την εντολή test-and-set στη μεταβλητή του κλειδώματος. Αν η προηγούμενη τιμή ήταν 0 (ελεύθερο), η εντολή επιστρέφει 0 και θέτει τη μεταβλητή σε 1, υποδηλώνοντας ότι το κλείδωμα έχει αποκτηθεί. Αν η προηγούμενη τιμή ήταν 1 (κατειλημένο), η εντολή επιστρέφει 1, και το νήμα πρέπει να επαναλάβει τη διαδικασία. Αν το νήμα δεν καταφέρει να αποκτήσει το κλείδωμα, εισέρχεται σε έναν βρόχο αναμονής, εκτελώντας συνεχώς την εντολή test-and-set μέχρι το κλείδωμα να απελευθερωθεί. Όταν το νήμα ολοκληρώσει την εργασία του στον κρίσιμο τμήμα, θέτει τη μεταβλητή του κλειδώματος πίσω σε 0, επιτρέποντας σε άλλα νήματα να το αποκτήσουν. Το συγκεκριμένο κλείδωμα προκαλεί υψηλή διεκδίκηση στο memory bus λόγω συνεχών λειτουργιών test-and-set, οδηγώντας σε υποβάθμιση της απόδοσης με περισσότερα νήματα.
- Το `ttas_lock` (Κλείδωμα Test-and-Test-and-Set) βελτιώνει το `tas_lock`. Τα νήματα διαβάζουν πρώτα την τιμή του κλειδώματος πριν προσπαθήσουν να το αποκτήσουν, μειώνοντας την κίνηση στο memory bus. Αποδίδει καλύτερα από το `tas_lock` υπό υψηλή διεκδίκηση, αλλά εξακολουθεί να αντιμετωπίζει προβλήματα απόδοσης καθώς αυξάνεται ο αριθμός των νημάτων.
- Το `array_lock` (Κλείδωμα Βασισμένο σε Πίνακα) είναι ένα κλείδωμα βασισμένο σε ουρά όπου κάθε νήμα περιστρέφεται σε μια ξεχωριστή θέση πίνακα, μειώνοντας τη διεκδίκηση σε κοινόχρηστες θέσεις μνήμης. Παρέχει δικαιοσύνη και μειώνει την κίνηση της συνοχής cache, αποδίδοντας καλά έως έναν μέτριο αριθμό νημάτων, αλλά μπορεί να επιφέρει επιβάρυνση λόγω διαχείρισης του πίνακα.
- Το `c1h_lock` (Κλείδωμα CLH) είναι ένα κλείδωμα περιστροφής βασισμένο σε ουρά που χρησιμοποιεί μια συνδεδεμένη λίστα. Κάθε νήμα περιστρέφεται σε έναν κόμβο, μειώνοντας τη διεκδίκηση και την ακύρωση της cache. Κλιμακώνεται καλύτερα από τα απλά κλειδώματα περιστροφής και διατηρεί καλή απόδοση ακόμη και καθώς αυξάνεται ο αριθμός των νημάτων λόγω μειωμένης διεκδίκησης.
- Η οδηγία `omp critical` στο OpenMP διασφαλίζει ότι μόνο ένα νήμα εκτελεί τον κώδικα που περικλείεται κάθε φορά. Είναι απλή στη χρήση αλλά μπορεί να γίνει στενωπός με πολλά νήματα λόγω σειριοποίησης, οδηγώντας σε αυξημένους χρόνους εκτέλεσης σε υψηλότερους αριθμούς νημάτων.
- Η οδηγία `omp atomic` διασφαλίζει ότι μια συγκεκριμένη λειτουργία μνήμης εκτελείται ατομικά. Έχει λιγότερη επιβάρυνση από το `omp critical` επειδή λειτουργεί σε επίπεδο εντολής. Κλιμακώνεται καλύτερα με περισσότερα νήματα αλλά περιορίζεται σε απλές λειτουργίες.

Παρακάτω φαίνονται οι χρόνοι εκτέλεσης:





### Ανάλυση Απόδοσης.

- Από τα παραπάνω διαγράμματα παρατηρούμε ότι ο nosync\_lock δείχνει την καλύτερη απόδοση σχεδόν σε όλους τους αριθμούς νημάτων επειδή δεν υπάρχει επιβάρυνση συγχρονισμού. Ο χρόνος εκτέλεσης μειώνεται καθώς αυξάνεται ο αριθμός των νημάτων - μέχρι τα 16 νήματα ενώ μετά έχουμε μια μικρή αύξηση - υποδεικνύοντας καλή κλιμάκωση, αλλά τα αποτελέσματα είναι εσφαλμένα λόγω race conditions.
- Το omp atomic παρουσιάζει καλή κλιμάκωση έως 16 νήματα, διατηρώντας χαμηλούς χρόνους εκτέλεσης. Η απόδοση αρχίζει να υποβαθμίζεται πέρα από 32 νήματα αλλά παραμένει καλύτερη από τα υπόλοιπα κλειδώματα, υποδεικνύοντας ότι οι ατομικές λειτουργίες κλιμακώνονται καλά για απλές ενημερώσεις.
- Τα clh\_lock και array\_lock αποδίδουν καλύτερα από τα απλά κλειδώματα περιστροφής (tas\_lock, ttas\_lock) σε υψηλότερους αριθμούς νημάτων. Το clh\_lock υπερέχει συνεχώς του array\_lock στα 32 και 64 νήματα, υποδεικνύοντας καλύτερη κλιμάκωση υπό υψηλή διεκδίκηση. Η φύση τους που βασίζεται σε ουρά μειώνει τη διεκδίκηση και την ακύρωση της cache, οδηγώντας σε βελτιωμένη απόδοση.
- Η οδηγία omp critical αποδίδει επαρκώς έως 4 νήματα, αλλά ο χρόνος εκτέλεσης αυξάνεται σημαντικά με περισσότερα από 8 νήματα, υποδεικνύοντας ότι το κρίσιμο τμήμα γίνεται στενωπός λόγω σειριοποίησης. Στα 64 νήματα, ο χρόνος εκτέλεσης είναι ο υψηλότερος μεταξύ όλων των υλοποιήσεων, επισημαίνοντας την κακή κλιμάκωση.
- Το pthread\_mutex\_lock δείχνει λογική απόδοση έως 8 νήματα. Ο χρόνος εκτέλεσης αυξάνεται απότομα πέρα από 16 νήματα, αντανακλώντας επιβάρυνση από αλλαγές περιβάλλοντος και αποκλεισμό, καθιστώντας το μη ιδανικό για υψηλούς αριθμούς νημάτων σε σενάρια λεπτόκοκκου κλειδώματος.
- Το pthread\_spin\_lock αποδίδει καλά έως 4 νήματα. Ο χρόνος εκτέλεσης αυξάνεται με περισσότερα νήματα αλλά όχι τόσο δραστικά όσο τα tas\_lock ή ttas\_lock. Η φύση της ενεργής αναμονής οδηγεί σε σπατάλη κύκλων CPU υπό υψηλή διεκδίκηση.
- Τα tas\_lock και ttas\_lock δείχνουν αυξημένους χρόνους εκτέλεσης καθώς αυξάνεται ο αριθμός των νημάτων. Το ttas\_lock αποδίδει ελαφρώς καλύτερα από το tas\_lock, ειδικά σε υψηλότερους αριθμούς νημάτων, λόγω μειωμένης κίνησης στο memory bus. Ωστόσο, και τα δύο υποφέρουν από διεκδίκηση και δεν είναι κατάλληλα για σενάρια με πολλά νήματα.

Η λειτουργία κάθε κλειδώματος επηρεάζει άμεσα την απόδοσή του. Τα κλειδώματα που ελαχιστοποιούν τη διεκδίκηση σε κοινόχρηστη μνήμη (όπως τα clh\_lock και array\_lock) αποδίδουν καλύτερα υπό υψηλή σύγχρονη πρόσβαση. Οι ελαφριές μηχανισμοί συγχρονισμού (omp atomic) είναι πλεονεκτικοί όταν το κρίσιμο τμήμα περιλαμβάνει απλές λειτουργίες. Τα βαριά κλειδώματα που περιλαμβάνουν αποκλεισμό ή κλήσεις συστήματος (pthread\_mutex\_lock) δεν κλιμακώνουν καλά με αυξημένους αριθμούς νημάτων.

## 2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

### 2.2.1 Recursive Floyd-Warshall

Σε αυτό το ερώτημα υλοποιούμε μια παράλληλη έκδοση του recursive αλγορίθμου Floyd-Warshall χρησιμοποιώντας OpenMP tasks και πραγματοποιούμε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για threads = 1, 2, 4, 8, 16, 32, 64 στο μηχάνημα sandman. Παρακάτω φαίνεται ο κώδικας τα αντίστοιχα διαγράμματα.

```
// FW Recursive Algorithm

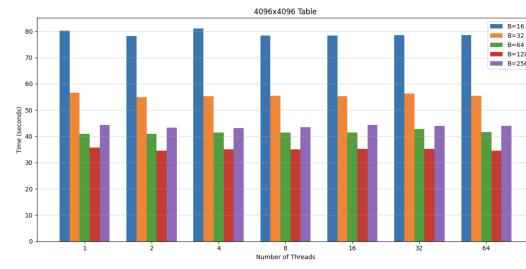
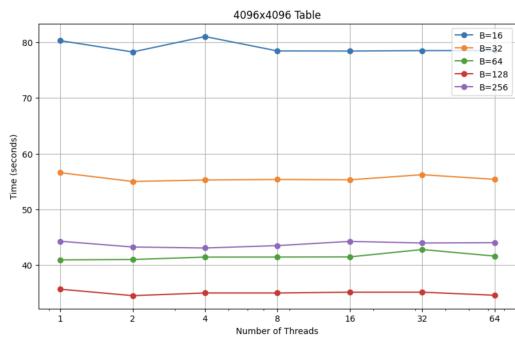
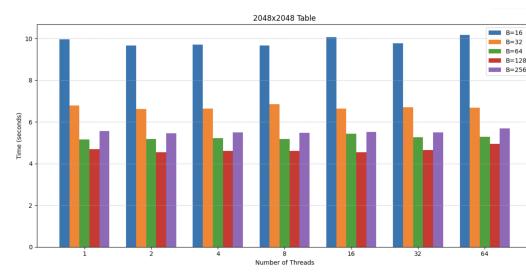
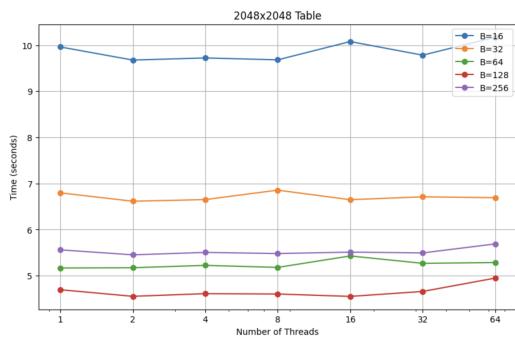
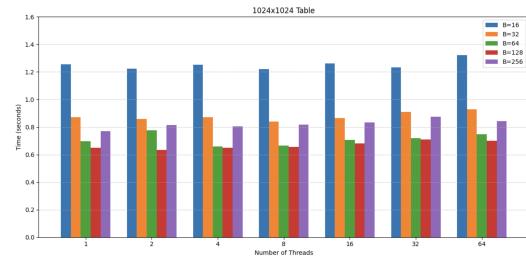
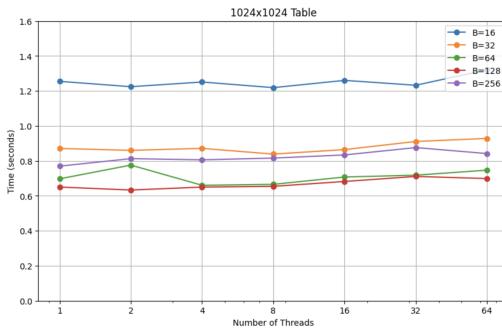
#pragma omp parallel
{
    #pragma omp single
    {
        FW_SR(A, arow, acol, B, brow, bcol, C, crow, ccol, myN / 2, bsize);

        #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);
        }

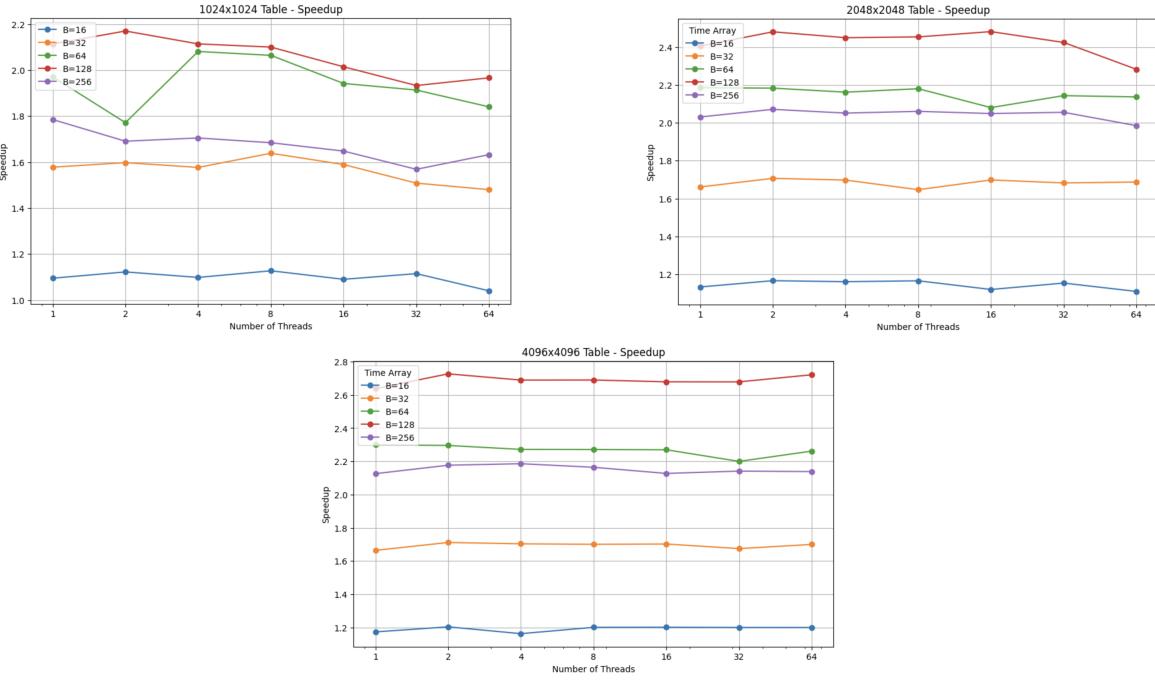
        #pragma omp taskwait
        FW_SR(A, arow + myN / 2, acol + myN / 2, B, brow + myN / 2, bcol, C, crow, ccol + myN / 2,
               myN / 2, bsize);
        FW_SR(A, arow + myN / 2, acol + myN / 2, B, brow + myN / 2, bcol + myN / 2, C, crow + myN / 2,
               ccol + myN / 2, myN / 2, bsize);

        #pragma omp task shared (A, B, C)
        {
            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);
        }

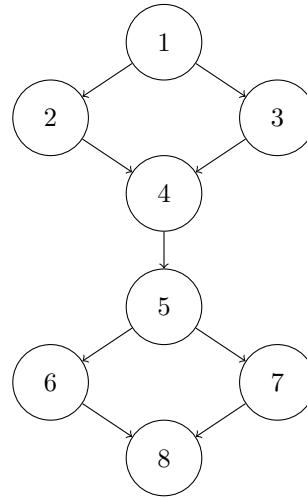
        #pragma omp taskwait
        FW_SR(A, arow, acol, B, brow, bcol + myN / 2, C, crow + myN / 2, ccol, myN / 2, bsize);
    }
}
```



Παρακάτω φαίνονται τα αντίστοιχα Speedup διαγράμματα:



Η κλιμάκωση του προγράμματος μας δεν είναι καλή, γεγονός που περιμέναμε καθώς λόγω εξαρτήσεων έχουμε τη δυνατότητα να παραλληλοποιήσουμε μόνο τις αναδραμικές κλήσεις 2,3 και 6,7 όπως φαίνεται και από το ακόλουθο task graph. Παρατηρούμε επίσης, ότι συνήθως η καλύτερη απόδοση επιτυγχάνεται με δύο πυρήνες ανεξαρτήτως του μεγέθους του block και του table, κάτι που εξηγείται πάλι από το γεγονός ότι αφού μπορούν να παραλληλοποιηθούν μέχρι δύο αναδραμικές κλήσεις περισσότεροι πυρήνες δεν θα μας προσφέρουν ιδιαίτερη επιτάχυνση - καθώς δεν υπάρχουν tasks να τους αξιοποιήσουν. Συνεπώς, είναι αναμενόμενο ότι το πρόγραμμα μας δεν κλιμακώνει καλά για περισσότερα από δύο threads - δεν μπορούμε να εκμεταλλευτούμε περαιτέρω παραλληλισμό και ταυτόχρονα έχουμε περισσότερο overhead λόγω της δημιουργίας και της επικοινωνίας μεγαλύτερου αριθμού threads.



Αξίζει να σημειωθεί ότι και στα 3 μεγέθη table το block size 16 έχει το μεγαλύτερο χρόνο εκτέλεσης ενώ για block size ίσο με 128 πετυχαίνουμε την καλύτερη απόδοση.

### 2.2.2 Bonus: Tiled Floyd-Warshall

Παρακάτω υλοποιούμε παράλληλη έκδοση του tiled αλγορίθμου Floyd-Warshall και συγκεντρώνουμε μετρήσεις για όμοια μεγέθη πινάκων. Παρατηρούμε ότι οι επιδόσεις είναι αυξημένες συγκριτικά με τη recursive έκδοση.

```
// FW Tiled

for(k=0;k<N;k+=B){
    FW(A,k,k,k,B);

    #pragma omp parallel
    {
        // Row blocks
        #pragma omp for schedule(static) nowait
        for (i = 0; i < k; i += B) {
            FW(A, k, i, k, B);
        }

        #pragma omp for schedule(static) nowait
        for (i = k + B; i < N; i += B) {
            FW(A, k, i, k, B);
        }

        // Column blocks
        #pragma omp for schedule(static) nowait
        for (j = 0; j < k; j += B) {
            FW(A, k, k, j, B);
        }

        #pragma omp for schedule(static) nowait
        for (j = k + B; j < N; j += B) {
            FW(A, k, k, j, B);
        }
    }

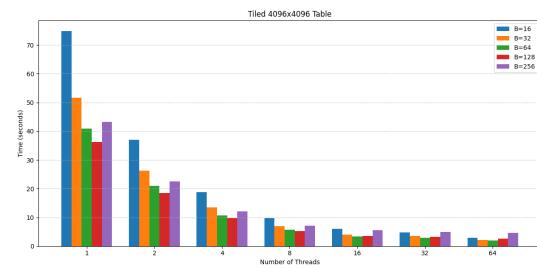
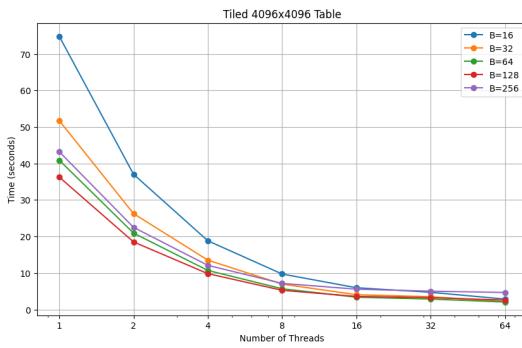
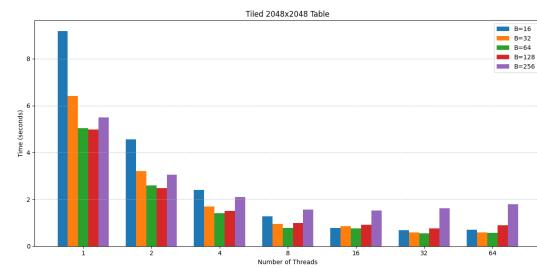
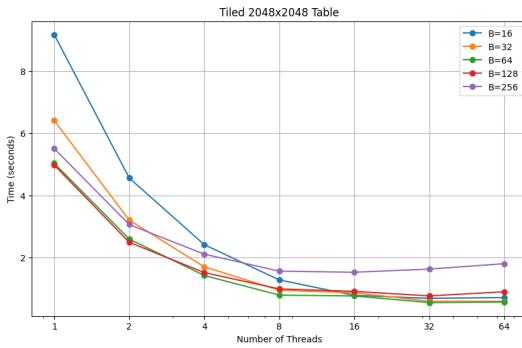
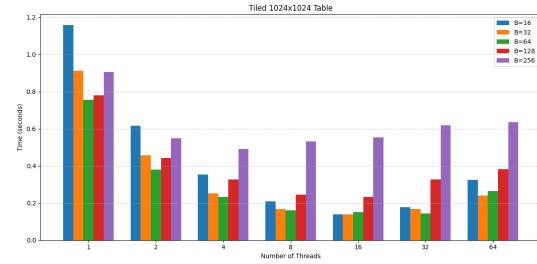
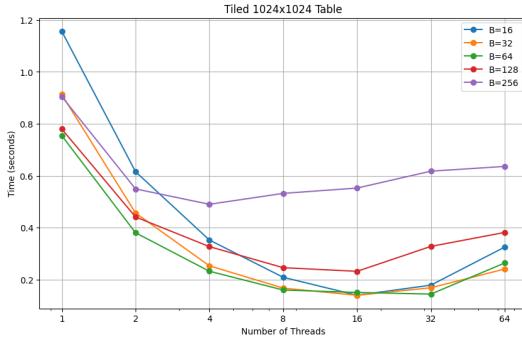
    #pragma omp parallel
    {
        #pragma omp for collapse(2) schedule(static) nowait
        for(i=0; i<k; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for collapse(2) schedule(static) nowait
        for(i=0; i<k; i+=B)
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);

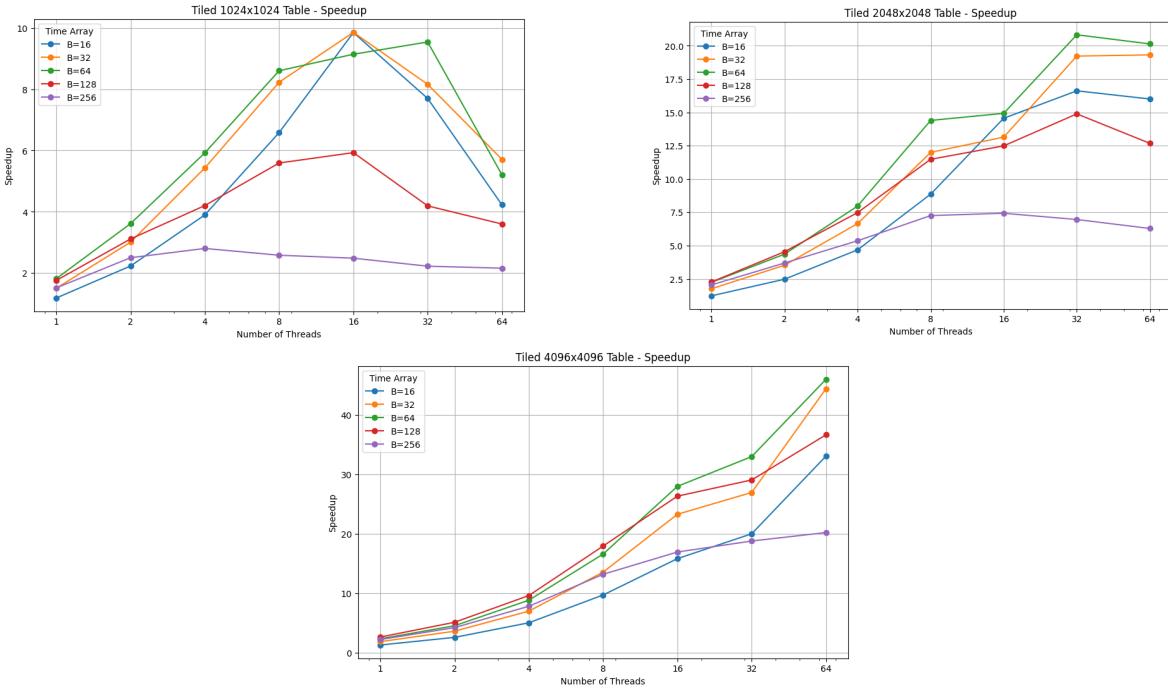
        #pragma omp for collapse(2) schedule(static) nowait
        for(i=k+B; i<N; i+=B)
            for(j=0; j<k; j+=B)
                FW(A,k,i,j,B);

        #pragma omp for collapse(2) schedule(static) nowait
        for(i=k+B; i<N; i+=B)
            for(j=k+B; j<N; j+=B)
                FW(A,k,i,j,B);
    }
}
}
```

}



Παρακάτω φαίνονται τα αντίστοιχα Speedup διαγράμματα:



Ο καλύτερος χρόνος που πετύχαμε στο  $4096 \times 4096$  Table ήταν 2.0465 seconds με 64 threads και block size 64.

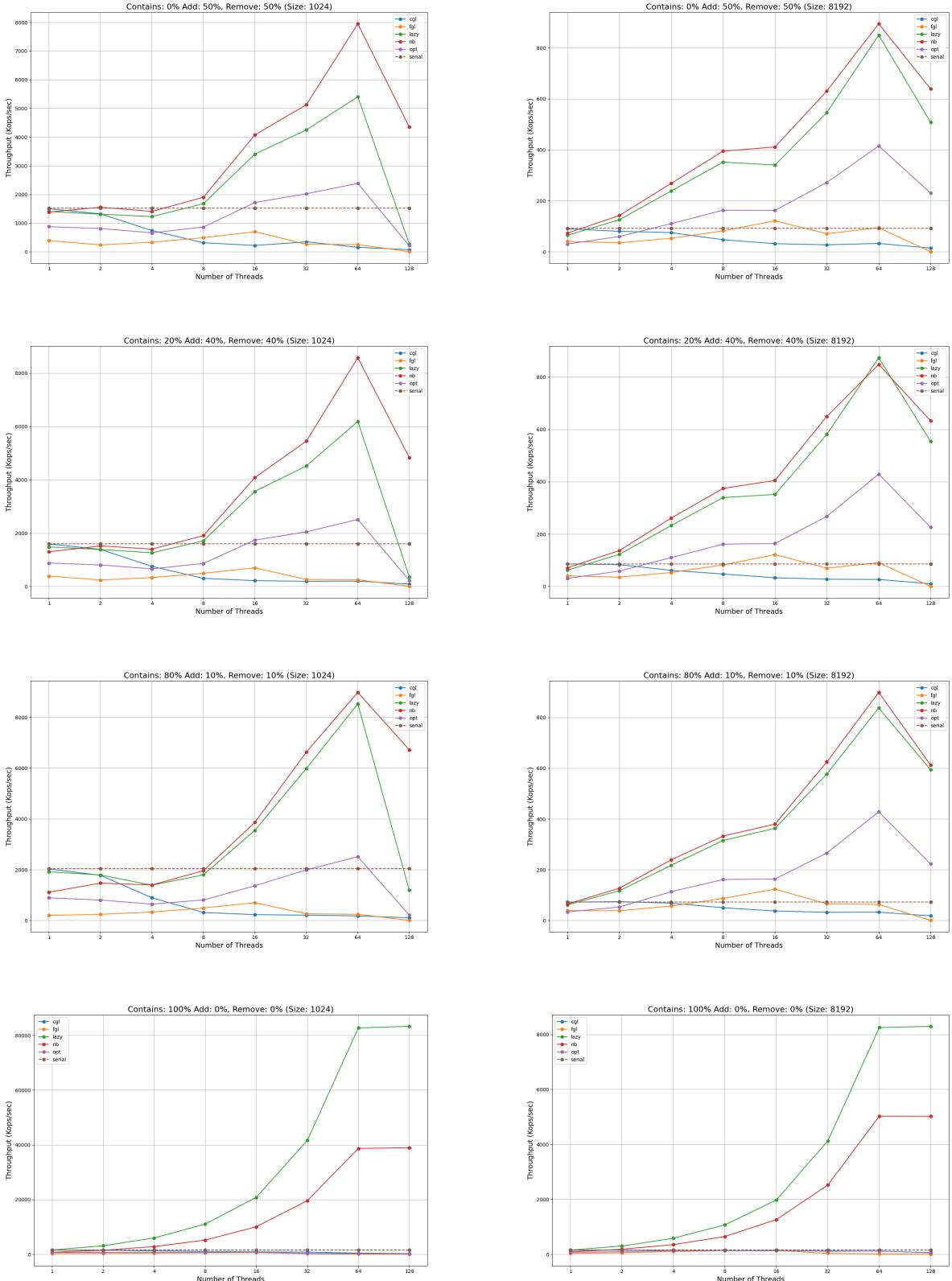
Τα αποτελέσματα υποδεικνύουν την αποδοτικότητα του tiled αλγορίθμου Floyd-Warshall, αναδεικνύοντας τη σημασία της ισορροπίας μεταξύ υπολογιστικού φόρτου, memory access patterns και παραλληλισμού. Η tiled εκδοχή του αλγορίθμου κλιμακώνει πολύ καλύτερα σε σχέση με την recursive.

Για μικρότερα μεγέθυνα πίνακα ( $N=1024$ ), η βέλτιστη απόδοση παρατηρείται με μέτριο μέγεθος tile ( $B=64$ ) και αυξημένο αριθμό νημάτων, επιτυγχάνοντας ελάχιστους χρόνους εκτέλεσης (π.χ., 0.1441s με 64 νήματα). Αυτή η αποδοτικότητα οφείλεται στη βελτιωμένη χρήση της κρυφής μνήμης μέσω της tiled στρατηγικής, η οποία διατηρεί μικρότερα υποπλέγματα σε γρήγορη μνήμη, μειώνοντας τις καθυστερήσεις λόγω ελλείψεων cache. Ωστόσο, τα οφέλη της παραλληλοποίησης μειώνονται με υπερβολικό αριθμό νημάτων (π.χ.,  $B=16, 64$  νήματα, 0.3252s), πιθανόν λόγω συμφόρησης στους κοινόχρηστους πόρους και αυξημένου κόστους διαχείρισης νημάτων. Μικρότερα tiles γενικά αποδίδουν καλύτερα σε μικρότερους πίνακες, καθώς προσαρμόζονται καλύτερα στην cache, ενώ τα μεγαλύτερα tiles είναι λιγότερο αποτελεσματικά στις ίδιες περιπτώσεις.

Για μεγαλύτερους πίνακες ( $N=4096$ ), η επίδραση του μεγέθους του tile γίνεται πιο έντονη. Παρόλο που τα μικρότερα tiles ( $B=64$ ) συνεχίζουν να αποδίδουν καλύτερα με υψηλότερους αριθμούς νημάτων (π.χ., 2.0465s με 64 νήματα), τα μεγαλύτερα tiles ( $B=256$ ) δυσκολεύονται λόγω αυξημένων απαιτήσεων memory bandwidth και μειωμένης αποδοτικότητας της cache.

Ενδιαφέρον είναι ότι, καθώς αυξάνεται ο αριθμός των νημάτων, τα οφέλη περιορίζονται περισσότερο στα μεγαλύτερα tiles, όπου η συμφόρηση και το κόστος συγχρονισμού αντισταθμίζουν τα οφέλη της παραλληλοποίησης. Τα αποτελέσματα αυτά υπογραμμίζουν την ανάγκη για προσεκτική ρύθμιση του μεγέθους των tiles και του αριθμού των νημάτων, λαμβάνοντας υπόψη τα χαρακτηριστικά του υλικού και το μέγεθος του προβλήματος.

## 2.3 Ταυτόχρονες Δομές Δεδομένων



- **Coarse-grain locking.** Η CGL χρησιμοποιεί ένα μοναδικό κλείδωμα για ολόκληρη τη δομή δεδομένων, εξασφαλίζοντας ότι μόνο ένα νήμα μπορεί να προσπελάσει ή να τροποποιήσει τη δομή μία χρονική στιγμή. Σε όλες τις εκτελέσεις, η CGL παρουσιάζει μείωση της απόδοσης καθώς

αυξάνεται ο αριθμός των νημάτων. Για παράδειγμα, στην εκτέλεση "Contains: 0% Add: 50%, Remove: 50% (Size: 1024)", η απόδοση πέφτει από  $\sim 1500$  Kops/sec με 1 νήμα σε  $\sim 73$  Kops/sec με 128 νήματα. Το βασικό πρόβλημα είναι ο ανταγωνισμός κλειδώματος. Καθώς περισσότερα νήματα προσπαθούν να αποκτήσουν το μοναδικό κλείδωμα, ξοδεύουν περισσότερο χρόνο αναμονής, οδηγώντας σε μείωση της συνολικής απόδοσης. Συνεπώς, η CGL δεν κλιμακώνεται καλά με την αύξηση των νημάτων λόγω της σειριακής πρόσβασης που επιβάλλεται από το ενιαίο κλείδωμα.

- **Fine-grain locking.** Το FGL χρησιμοποιεί πολλαπλά κλειδώματα, καθένα από τα οποία προστατεύει ένα υποσύνολο της δομής δεδομένων, επιτρέποντας μεγαλύτερη ταυτόχρονη χρήση σε σύγκριση με το CGL. Σε ορισμένες εκτελέσεις, η FGL παρουσιάζει μέτρια επεκτασιμότητα. Σε όλα τα διαγράμματα, η απόδοση αρχικά μειώνεται, αλλά στη συνέχεια σταθεροποιείται ή αυξάνεται ελαφρώς πριν πέσει κατακόρυφα σε υψηλούς αριθμούς νημάτων. Γενικά, η FGL υπερτερεί έναντι της CGL μειώνοντας τον ανταγωνισμό κλειδώματος. Ουστόσο, η απόδοσή της μπορεί ακόμα να υποβαθμιστεί σε συνθήκες υψηλού ανταγωνισμού ή με πολλά νήματα. Συνεπώς, αν και καλύτερη από την CGL, η FGL εισάγει μεγαλύτερη πολυπλοκότητα στη διαχείριση κλειδώματος, η οποία μπορεί μερικές φορές να αντισταθμίσει τα κέρδη της σε απόδοση.
- **Optimistic synchronization.** Ο Optimistic συγχρονισμός υποθέτει ότι οι συγχρούσεις μεταξύ των νημάτων είναι σπάνιες και επιτρέπει στα νήματα να προχωρούν χωρίς κλειδώματα, επικυρώνοντας τις αλλαγές πριν από τη δέσμευση. Σε ορισμένες περιπτώσεις, όπως την εκτέλεση "80% Add: 10%, Remove: 10% (Size: 8192)", η απόδοση αυξάνεται με τα νήματα αλλά τελικά πέφτει καθώς ο ανταγωνισμός γίνεται σημαντικός. Ο optimistic συγχρονισμός λειτουργεί καλύτερα από τον FGL συγχρονισμό καθώς το κόστος να διατρέξουμε τη δομή δύο φορές χωρίς κλειδώματα είναι χαμηλότερο από το να την διατρέξουμε μία φορά με κλειδώματα. Συνεπώς, ο optimistic συγχρονισμός είναι αποτελεσματικός σε περιβάλλοντα με χαμηλό ανταγωνισμό, αλλά μπορεί να δυσκολευτεί καθώς αυξάνονται τα επίπεδα ταυτόχρονης χρήσης, οδηγώντας σε αυξημένη επιβάρυνση επικύρωσης και μειωμένη απόδοση.
- **Lazy synchronization.** Ο lazy συγχρονισμός επιτρέπει στα νήματα να εκτελούν λειτουργίες με τρόπο που ελαχιστοποιεί το χρόνο χράτησης κλειδώματος και αναβάλλει ορισμένες ενημερώσεις για να ενισχύσει την ταυτόχρονη εκτέλεση. Ο lazy συγχρονισμός επιδεικνύει σταθερά υψηλή απόδοση καθώς αυξάνεται ο αριθμός των νημάτων μέχρι έναν συγκεκριμένο αριθμό νημάτων - στη περίπτωση μας μέχρι 64 νήματα. Συνεπώς, με την ελαχιστοποίηση της διάρκειας και της συχνότητας απόκτησης κλειδώματος, ο lazy συγχρονισμός μειώνει τον ανταγωνισμό και ενισχύει τον παραλληλισμό.
- **Non-blocking synchronization.** Οι τεχνικές συγχρονισμού NB, όπως οι αλγόριθμοι χωρίς κλείδωμα ή χωρίς αναμονή, επιτρέπουν στα νήματα να λειτουργούν σε κοινά δεδομένα χωρίς τους παραδοσιακούς μηχανισμούς κλειδώματος. Ο συγχρονισμός NB παρουσιάζει την υψηλότερη απόδοση σε όλες τις εκτελέσεις, ξεπερνώντας συχνά σημαντικά άλλες μεθόδους. Συγκεκριμένα, ο NB αποφεύγει εντελώς τον ανταγωνισμό, επιτρέποντας στα νήματα να προχωρούν χωρίς αναμονή. Παρά την έλλειψη κλειδωμάτων, οι μέθοδοι NB διασφαλίζουν τη συνέπεια των δεδομένων μέσω ατομικών πράξεων και προσεκτικού σχεδιασμού αλγορίθμων. Συνεπώς, ο nb συγχρονισμός προσφέρει ανώτερη κλιμάκωση και απόδοση, ιδίως σε περιβάλλοντα υψηλής ταυτόχρονης λειτουργίας, αν και μπορεί να απαιτεί πιο σύνθετες στρατηγικές υλοποίησης.
- **Serial execution.** Η σειριακή εκτέλεση επεξεργάζεται όλες τις λειτουργίες διαδοχικά χωρίς καμία ταυτόχρονη εκτέλεση, χρησιμεύοντας ως βάση σύγκρισης. Σε όλες τις εκτελέσεις, η σειριακή εκτέλεση διατηρεί σταθερή απόδοση ανεξάρτητα από τον αριθμό των νημάτων, καθώς μόνο ένα νήμα εκτελεί ενεργά λειτουργίες ανά πάσα στιγμή. Συνεπώς, η σειριακή εκτέλεση εξασφαλίζει την ορθότητα χωρίς επιβάρυνση συγχρονισμού, δεν αξιοποιεί τα πλεονεκτήματα των πολλαπλών νημάτων, με αποτέλεσμα χαμηλότερη απόδοση σε περιβάλλοντα ταυτόχρονης εκτέλεσης.

Συμπερασματικά, σε περιβάλλοντα υψηλού συγχρονισμού ο nb συγχρονισμός και ο lazy συγχρονισμός είναι οι πιο αποτελεσματικοί, παρέχοντας υψηλή απόδοση και επεκτασιμότητα. Επειτα, σε περιβάλλοντα όπου οι συγχρούσεις είναι σπάνιες κατάλληλος είναι και ο optimistic συγχρονισμός.

### 3 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

#### 3.1 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

##### 3.1.1 Naive version

Σε αυτή την έκδοση όλα αναθέσουμε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι του αλγορίθμου: τον υπολογισμό των nearest clusters σε κάθε iteration.

Τλοποιούμε την συνάρτηση `get_tid` όπως φαίνεται παρακάτω:

```
__device__ int get_tid() {
    return threadIdx.x + blockIdx.x * blockDim.x; /* TODO: copy me from naive version... */
}
```

Για τον υπολογισμό της ευκλείδιας απόστασης χρησιμοποιούμε την παρακάτω συνάρτηση:

```
__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;
    for (int i = 0; i < numCoords; i++)
        ans += (objects[objectId * numCoords + i] - clusters[clusterId * numCoords + i]) *
               (objects[objectId * numCoords + i] - clusters[clusterId * numCoords + i]);
    return (ans);
```

Για τον υπολογισμό των nearest clusters σε κάθε iteration χρησιμοποιούμε την παρακάτω συνάρτηση:

```
__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();

    if (tid < numObjs) { // Ensure the thread operates within bounds
        int index, i;
        double dist, min_dist;

        /* find the cluster id that has min distance to object */
        index = 0;

        min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters, tid, 0);

        for (i = 1; i < numClusters; i++) {
            dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters, tid, i);
            if (dist < min_dist)
```

```

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

if (deviceMembership[tid] != index) {
    //(*devdelta) += 1.0; possible race condition
    atomicAdd(devdelta, 1.0); // surely not optimal performance
}

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

```

Πραγματοποιύμε τις ζητούμενες μεταφορές δεδομένων σε κάθε iteration του αλγορίθμου, ώστε να γίνει δυνατή η εκτέλεση του υπόλοιπου kmeans (νέα cluster centers) στην CPU μετά από κάθε GPU kernel invocation, όπως φαίνεται παρακάτω:

```

checkCuda(cudaMemcpy(deviceClusters, clusters,
                     numClusters * numCoords * sizeof(double), cudaMemcpyHostToDevice));
checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
checkCuda(cudaMemcpy(membership, deviceMembership,
                     numObjs * sizeof(int), cudaMemcpyDeviceToHost));
checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
                     sizeof(double), cudaMemcpyDeviceToHost));

```

Τηλογίζουμε το grid size όπως φαίνεται παρακάτω:

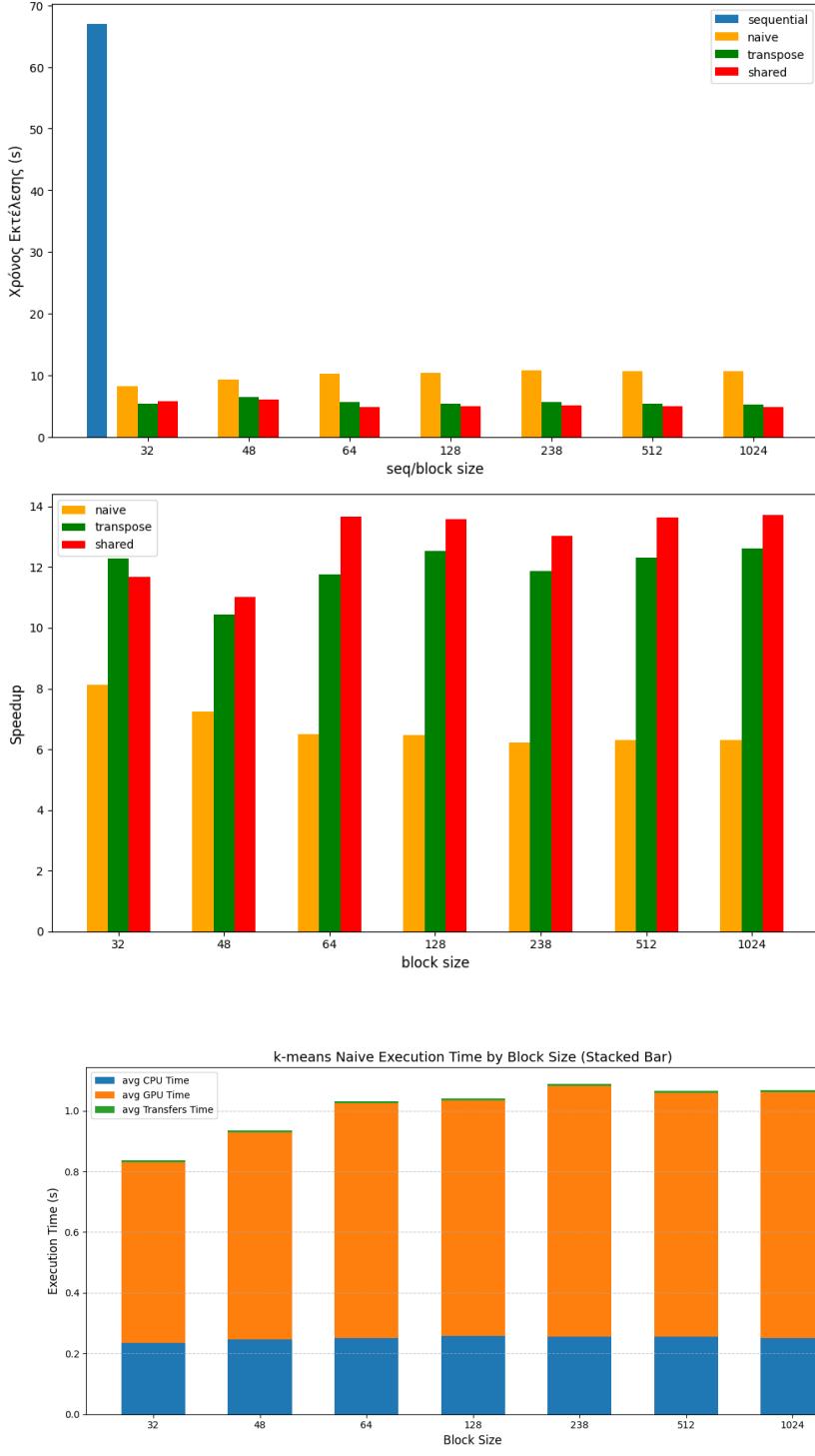
```

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

```

1. Παρακάτω φαίνονται τα διαγράμματα που ζητούνται για για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block\_size = {32, 48, 64, 128, 238, 512, 1024}.

Figure 1: Execution Time and Speedup barplots



Ο χρόνος εκτέλεσης φαίνεται να αυξάνεται (και αντίστοιχα το speedup να μειώνεται) ή να παραμένει σχετικά σταθερός καιώς αυξάνεται το μέγεθος του block size. Αυτό υποδεικνύει ότι η απόδοση δεν κλιμακώνεται γραμμικά με το μέγεθος του block, πιθανόν λόγω περιορισμών στη μνήμη, αφού ο K-means είναι ένας αλγόριθμος που χαρακτηρίζεται ως memory-bound, δηλαδή εκτελούνται σχετικά λίγες υπολογιστικές πράξεις σε μεγάλο όγκο δεδομένων. Αυτό έχει ως αποτέλεσμα η απόδοση να περιορίζεται χυρίως από τις καθυστερήσεις που προκαλούνται κατά τη φόρτωση δεδομένων από τη μνήμη, γεγονός που συχνά υπερκαλύπτει το όφελος που προκύπτει από την ταυτόχρονη εκτέλεση πράξεων από τα νήματα της GPU. Το διάγραμμα stacked bar παρουσιάζει τη διάσπαση του συνολικού χρόνου σε τρεις βασικές κατηγορίες: χρόνο GPU, χρόνο μεταφορών και χρόνο υπολογισμών στην CPU. Παρατηρούμε ότι ο χρόνος GPU είναι

σταθερά ο μεγαλύτερος, καταλαμβάνοντας το μεγαλύτερο μέρος του συνολικού χρόνου, γεγονός που δείχνει ότι η GPU εκτελεί την κύρια υπολογιστική εργασία. Ο χρόνος CPU παραμένει σχεδόν σταθερός, υποδηλώνοντας ότι ο ρόλος της CPU περιορίζεται σε δευτερεύουσες λειτουργίες, όπως η ενημέρωση των κέντρων των clusters. Τέλος, ο χρόνος μεταφοράς δεδομένων (πράσινο) είναι πολύ μικρός σε όλες τις περιπτώσεις, κάτι που υποδεικνύει ότι οι μεταφορές δεδομένων δεν αποτελούν σημαντικό περιορισμό για αυτή την υλοποίηση. Η συνολική κατανομή δείχνει ότι η απόδοση επηρεάζεται κυρίως από την GPU, ενώ οι υπόλοιποι παράγοντες έχουν μικρότερη συμβολή.

2. Όπως φαίνεται και από το διάγραμμα του speedup η naïve υλοποίηση του kmeans είναι 6-8 φορές γρηγορότερη από την sequential εκδοχή του αλγορίθμου. Παρόλα αυτά η naïve υλοποίηση δεν είναι ο ιδανικός πηρύνας για GPUs. Αν και χρησιμοποιεί τα νήματα για τον υπολογισμό των αποστάσεων και το atomicAdd για την αποφυγή αγώνων δεδομένων, υπάρχουν αρκετές βελτιώσεις που μπορούν να γίνουν. Η χρήση της shared memory θα μπορούσε να μειώσει την καθυστέρηση πρόσβασης στη μνήμη, ενώ η βελτιστοποίηση των προσβάσεων στη μνήμη μέσω coalesced patterns θα βελτίωνε την απόδοση. Επιπλέον, η εξάλειψη της εξάρτησης από atomicAdd και η καλύτερη κατανομή εργασιών μεταξύ των νημάτων θα ενίσχυαν τον παραλληλισμό και την αποδοτικότητα.
3. Για μικρά block-sizes (π.χ., 32, 48, 64), παρατηρείται αυξημένος χρόνος εκτέλεσης λόγω του ότι δεν απασχολούνται πλήρως οι διαθέσιμοι πυρήνες της GPU, οδηγώντας σε χαμηλή αποδοτικότητα. Καθώς το block-size αυξάνεται (π.χ., 128), η απόδοση σταθεροποιείται, καθώς η GPU επιτυγχάνει καλύτερη κατανομή εργασιών. Ωστόσο, για πολύ μεγάλα block-sizes (π.χ., 512, 1024), υπάρχει μια μικρή επιδείνωση, η οποία μπορεί να αποδοθεί σε αυξημένο κόστος συγχρονισμού και περιορισμούς στους διαθέσιμους πόρους, όπως registers και shared memory. Συνολικά, η συμπεριφορά αυτή δείχνει ότι η υλοποίηση πιθανώς είναι memory-bound, καθώς η πρόσβαση στη μνήμη αποτελεί περιοριστικό παράγοντα.

### 3.1.2 Transpose version

Σε αυτή την έκδοση αλλάζουμε την δομή των δεδομένων που έχουν διαστάσεις (e.g. objects & clusters) από row-based σε column-based indexing. Αυτό σημαίνει πως πρέπει όλοι οι σχετικοί πίνακες να γίνουν transpose με τη χρήση buffers όπου αναγράφεται.

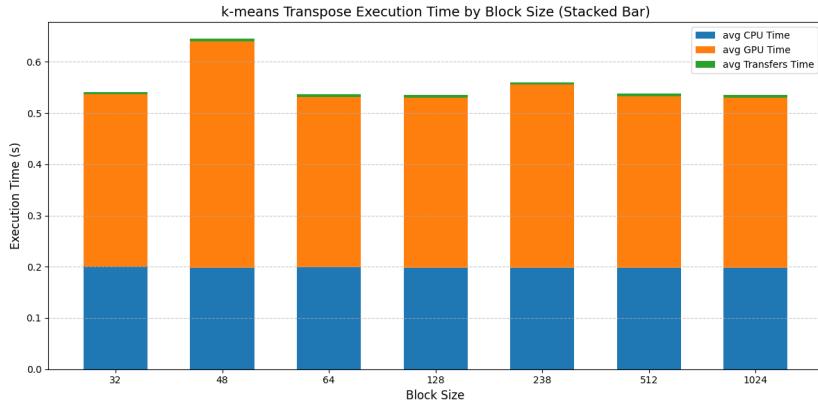
Για τον υπολογισμό της ευχλείδιας απόστασης αυτή την φορά χρησιμοποιούμε την παρακάτω συνάρτηση:

```
_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;
    for (int i = 0; i < numCoords; i++)
        ans += (objects[i * numCoords + objectId] - clusters[i * numCoords + clusterId]) *
               (objects[i * numCoords + objectId] - clusters[i * numCoords + clusterId]);
    return (ans);
}
```

Παρακάτω φαίνεται ο τρόπος με τον οποίο αλλάζουμε την δομή των δεδομένων που έχουν διαστάσεις (e.g. objects & clusters) από row-based σε column-based indexing:

```
double **dimObjects = (double **) calloc_2d(numCoords, numObjs, sizeof(double));
double **dimClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
double **newClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
```

1. Τα διαγράμματα για τον χρόνο εκτέλεσης και το speedup φαίνονται στο Figure 1 και παρακάτω φαίνεται το stacked barplot που ζητείται.



H transpose version παρουσιάζει σημαντικά βελτιωμένη επίδοση σε σχέση με τη naive έκδοση για όλες τις τιμές του block\_size. Οι χρόνοι εκτέλεσης για την transpose είναι αισθητά χαμηλότεροι, υποδεικνύοντας ότι η βελτιστοποίηση που επιτυγχάνεται μέσω της αναδιάταξης δεδομένων αυξάνει την αποδοτικότητα της πρόσβασης στη μνήμη. Συγκεκριμένα, οι χρόνοι εκτέλεσης για την transpose παραμένουν αρκετά σταθεροί, γεγονός που δείχνει ότι το block\_size επηρεάζει λιγότερο αυτή την έκδοση συγκριτικά με τη naive.

Στην transpose version, το block\_size έχει μικρότερη επίδραση στην απόδοση συγκριτικά με τη naive, υποδεικνύοντας ότι η βελτιστοποίηση στη διάταξη δεδομένων επιτρέπει καλύτερη διαχείριση του bandwidth της μνήμης και της χρήσης των blocks. Ωστόσο, παρατηρούνται μικρές διακυμάνσεις (π.χ., στο block\_size = 48 ή block\_size = 238), που μπορεί να οφείλονται σε ζητήματα ευθυγράμμισης δεδομένων ή χρήση shared memory στη GPU. Συνολικά, η transpose version αποδίδει καλύτερα, ανεξάρτητα από το block\_size.

Από το stacked barplot φαίνεται επίσης ότι η naive και η transpose έχουν περίπου ίδιο CPU time ενώ η transpose έχει σημαντικά λιγότερο GPU time. Αυτό υποδηλώνει ότι η transpose έκδοση αξιοποιεί πιο αποδοτικά την υπολογιστική ισχύ της GPU, πιθανότατα μέσω της βελτιστοποιημένης πρόσβασης στη μνήμη, μειώνοντας τους χρόνους εκτέλεσης των υπολογισμών. Η βελτίωση αυτή μπορεί να αποδούνε στην αναδιάταξη των δεδομένων, η οποία μειώνει τις μη αποδοτικές προσπελάσεις στη μνήμη (memory coalescing) και βελτιώνει την αξιοπόίηση των threads της GPU. Επιπλέον, το γεγονός ότι ο χρόνος CPU παραμένει σχεδόν αμετάβλητος μεταξύ των δύο εκδόσεων δείχνει ότι οι άλλαγές έγιναν κυρίως στο GPU kernel, χωρίς να επηρεαστούν οι λειτουργίες που εκτελούνται στη CPU.

2. Η διαφορά στην απόδοση μεταξύ των δύο εκδόσεων οφείλεται στον τρόπο με τον οποίο οι δύο δομές δεδομένων επηρεάζουν την πρόσβαση στη μνήμη και την εκμετάλλευση της τοπικότητας της μνήμης από τα threads. Στην πρώτη έκδοση (row-based format), κάθε thread διαβάζει δεδομένα που βρίσκονται σε διαφορετικές γραμμές της μνήμης. Αυτό σημαίνει ότι τα διαδοχικά threads προσπελαύνουν στοιχεία που δεν είναι διαδοχικά τοποθετημένα στη μνήμη. Με αποτέλεσμα να μην εκμεταλλεύονται τη δυνατότητα coalesced memory access των GPUs, όπου τα threads μιας warp μπορούν να διαβάσουν συνεχόμενα δεδομένα με μια ενιαία πρόσβαση στη μνήμη. Αντίθετα, στη δεύτερη έκδοση (column-based format), τα threads διαβάζουν στοιχεία που βρίσκονται στην ίδια στήλη, άρα σε συνεχόμενες θέσεις μνήμης. Έτσι, τα διαδοχικά threads ενός warp προσπελαύνουν δεδομένα που είναι κοντά στη μνήμη, με αποτέλεσμα να εκμεταλλεύονται καλύτερα την coalesced memory access. Αυτό μειώνει σημαντικά τον αριθμό των ανεξάρτητων προσβάσεων στη μνήμη και βελτιώνει την απόδοση. Επιπλέον, οι GPUs είναι σχεδιασμένες να λειτουργούν βέλτιστα όταν οι προσβάσεις στη μνήμη είναι ευθυγραμμισμένες και συνεχόμενες, κάτι που επιτυγχάνεται στην έκδοση με τη μετατέση (transpose), αλλά όχι στην row-based έκδοση.

### 3.1.3 Shared version

Σε αυτή την έκδοση τοποθετούμε τα clusters στην διαμοιραζόμενη μνήμη της GPU ώστε να μπορούν τα στοιχεία κάθε block να τα κάνουν access πιο γρήγορα.

Παρακάτω φαίνεται πως ορίζουμε το μέγεθος της διαμοιραζόμενης μνήμης που χρειάζεται η υλοποίησή μας:

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

Για τον υπολογισμό των nearest clusters σε κάθε iteration, στην συγκεκριμένη έκδοση, χρησιμοποιούμε

την παρακάτω συνάρτηση:

```
_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[];

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

for (int i = threadIdx.x; i < numClusters * numCoords; i += blockDim.x) {
    shmemClusters[i] = deviceClusters[i];
}
__syncthreads();

if (tid < numObjs) {
    int index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;

    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters, tid,
                                       i);

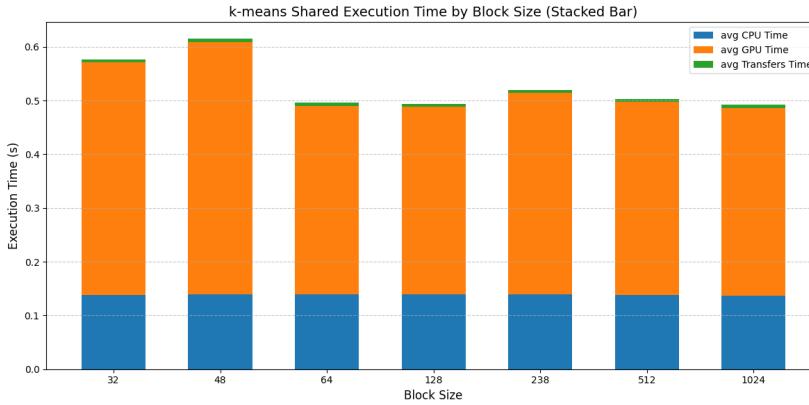
    for (i = 1; i < numClusters; i++) {
        dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters, tid, i);

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

    if (deviceMembership[tid] != index) {
        atomicAdd(devdelta, 1.0);
    }

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

1. Τα διαγράμματα για τον χρόνο εκτέλεσης και το speedup φαίνονται στο Figure 1 και παρακάτω φαίνεται το stacked barplot που ζητείται.



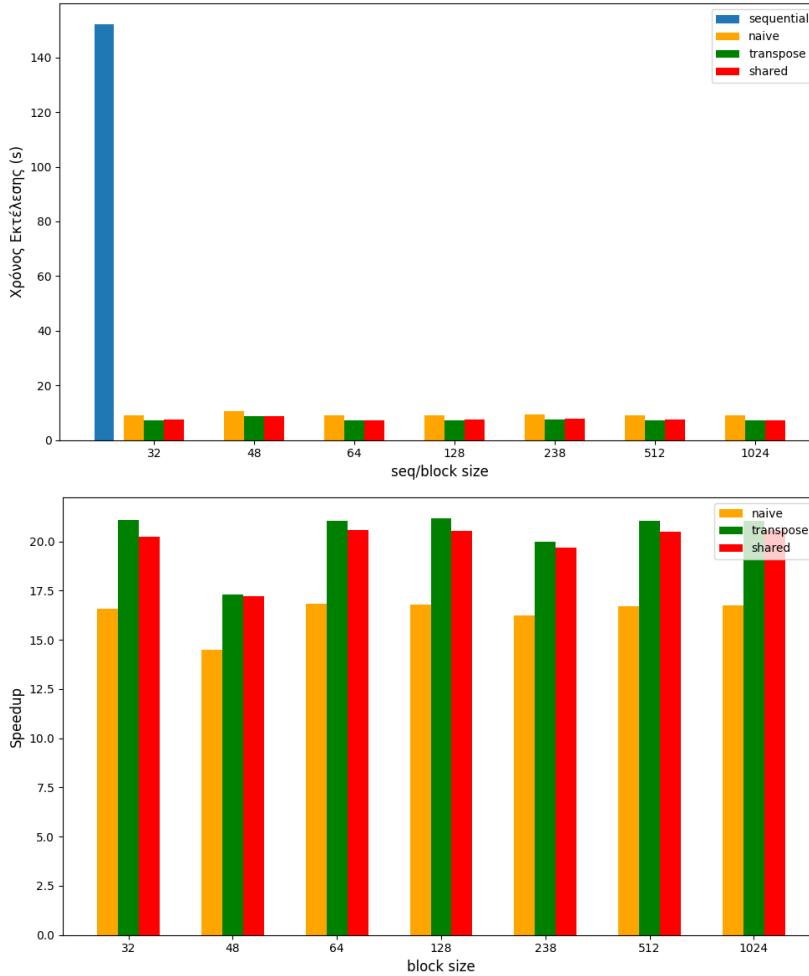
H shared memory ékdoosi πaφouσiaζei βeλtiωménη epiðoisoη se σchésoη me tñn naivē kai tñn transpose ékdoosi, ópωs φañnetai apó tñs metrήsies. H shared memory aξiopoiieί tñn taχyterη, χaυmhlήs kauñusstérøsøs mññmη pñi eñvai koiñj se óla tñs threads evñs block, muiñnonatas σhmanatiká tñ kóstoñs tñw prospelásøwøs stñ global memory. Sñnoliká, η chrήsøtη shared memory meiñwnei tñw xroñous GPU computation, oðhgyñntas se βeλtiωménη suñolikó xroñu ektélesøsøs kauñwøs eχei stañherá kauñteørous xroñou s se σchésoη me tñn naivē kai suñxñá xzeperná tñn transpose, eidiñká gya megálæs tñmés block size. Pañatøroñme akómyt òti me tñn aúñsøtø tñ block size o xroñu ektélesøsøs meiñwnetai, enw pañatøroñme kai pñløi kápøia peaks stñ xroñu ektélesøsøs gya block\_size = {48,238} ta ópøia ñen eñvai ñuñaméis tñw 2 kai suñepwøs η xeiñotøerø autñ epiðoisoη mporøi na oñfélætai se ññtñmata suñugrámmitøs ñeðoménwø h chrήsøtη shared memory stñ GPU.

To block size pñiñe xriñvmo rólo stñ shared memory ékdoosi, kauñwøs kauñorízei tñw ariñmuñ tñw threads pñi suñeragázontai gya na ñiaxepriñtouñ apoteleñsamatiká tñ shared memory. An tñ block size eñvai pñløi mikró, η shared memory ñen aξiopoiieítai plñhrøs, enw an eñvai pñløi megálø, mporøi na aúñhñeñ η suñuñrøsøtø stñw pñrøs tñs GPU. Gia tñ suñgækeménñ ulopoiñsø, oñ bñlitiñtøs tñmés tñ block size bñsokontai stñ meñsáia pñriñxø (64-128), ópou eñiñorrøpeñtai η chrήsøtø tñw threads kai η shared memory. Apó tñ stacked barplot pañatøroñme òti η shared memory ektélesøtø tñ algoriñmuñ eχei tñ mikróterø CPU time apó tñs állæs ñuñ ektélesøsøs kai aúñménñ GPU time se σchésoη me tñn transpose ektélesøtø. Autñ oñfélærøi stñ gennonñs òti xriñmopotioñtøs tñ shared clusters aúñanouñme tñ xroñu pñi pñrññeñ tñ pñrgøramma stñw GPU. Kai pñløi tñ average transfers time eñvai ařketañ mikróterø apó tñ avg GPU time kai avg CPU time.

### 3.1.4 Σúγkriñtø ułoñpoññsewø / bottleneck Analysis + Bonus-1

1. Me bñsøtø tñw xeñwariñtouñ xroñismouñ (GPU part, CPU-GPU transfers, CPU part) kai tñ ñeðoménwø apó tñ stacked barplots, φañnetai òti o xýriñs pñriñrøstikó pñrøgøntas stñtñ apóðoisoñ tñ iteratiñve mñrøs eñvai o xroñuñ GPU. H GPU ektélesøtø tñ xýriñs mñrøs tñw upoloyisumwø, kai oñpøiañdþpote kauñusstérøsøtø stñ mññmñ h stñ suñgærøniñmø tñw threads eptñrøzæi ámøsa tñn apóðoisoñ. Η chrήsøtη shared memory stñ shared memory meiñwnei tñs kauñusstérøsøtøs pñrøsøbasøsøtø stñ mññmñ, allá aúñanøi elaqñrøs tñ suñolikó xroñu GPU lógya tñs eptñløññ pñløsøkotøtøs stñ ñiaxepriñtø tñ shared memory. O xroñuñ CPU pñrøaméniñ stñtñ apóðoisoñ tñ iteratiñve mñrøs eptñrøzæi kauñwøs pñriñrøsøtøs ñeðoménwø s ñeñtøreñouñsøs leitouñrgyøs, ópωs η enñmérøwañtø tñw kéntrøwø tñw clusters. Oñ metapøorøs ñeðoménwø eñvai epiñsøtø añhñantøs se σchésoη me tñw állæs xroñouñs, káti pñi ñeñxñeñ òti ñen apoteleñsøtø pñriñrøsøtø. Suñepwøs, η suñolikó apóðoisoñ tñ iteratiñve mñrøs eptñrøzæi kauñwøs apó tñn apoteleñsamatikótøtø tñ GPU stñ xeñwariñmø tñw ñeðoménwø kai tñn ektélesøtø tñw upoloyisumwø.

2. Pañakátañ φañnetai tñ ñiaxgrámata pñi ññtñmata gya gya tñ configuration {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} kai gya block\_size = {32, 48, 64, 128, 238, 512, 1024}.



Για το πείραμα με numCoords = 2, παρατηρούμε ότι οι χρόνοι εκτέλεσης είναι γενικά υψηλότεροι σε σύγκριση με το numCoords = 32, ειδικά για τις υλοποιήσεις naive και transpose. Συγκεκριμένα, η προσέγγιση naive παρουσιάζει μικρές διακυμάνσεις μεταξύ των διαφορετικών μεγεθών block, ενώ η transpose και η shared φαίνεται να έχουν πιο σταυρερή απόδοση, με την trasnpose να πετυχαίνει τις καλύτερες επιδόσεις συνολικά, αλλά με ελαφρώς υψηλότερες τιμές σε σχέση με την περίπτωση του numCoords = 32. Εντυπωσιακή είναι η μεγάλη αύξηση του χρόνου για την σειριακή υλοποίηση (sequential) στο numCoords = 2 σε σχέση με το numCoords = 32, γεγονός που υποδεικνύει ότι το κόστος επεξεργασίας αυξάνεται δυσανάλογα όταν ο αριθμός των συντεταγμένων είναι χαμηλός. Αυτό πιθανώς οφείλεται στον αυξημένο χρόνο επικοινωνίας ή στην μικρότερη δυνατότητα εκμετάλλευσης της παράλληλης επεξεργασίας από το hardware.

Η shared υλοποίηση παρουσιάζει περιορισμούς όταν πρόκειται για arbitrary configurations. Ένας βασικός περιορισμός είναι η εξάρτηση από το μέγεθος της shared memory που είναι διαθέσιμη στη συσκευή CUDA. Όπως επισημαίνεται και στον κώδικα, αν το μέγεθος της μνήμης που απαιτείται για τα clusters υπερβαίνει τη διαθέσιμη μνήμη shared ανά block, η εκτέλεση μπορεί να αποτύχει ή να παρουσιάσει προβλήματα απόδοσης. Επίσης, η προσέγγιση δεν λαμβάνει υπόψη την πιθανότητα αυξημένων αριθμών clusters ή συντεταγμένων που μπορεί να οδηγήσουν σε υπέρβαση της μνήμης shared.

**(Bonus-1) 3.** Η συνάρτηση cudaOccupancyMaxPotentialBlockSize είναι μια βοηθητική συνάρτηση που παρέχεται από το CUDA Runtime API για βοηθήσει στον προσδιορισμό του βέλτιστου μέγεθους block και grid για τη μέγιστη αξιοποίηση του GPU occupancy για μια συγκεκριμένη kernel function. Το occupancy αναφέρεται στην αναλογία των ενεργών warps προς τον μέγιστο αριθμό warps που υποστηρίζονται σε έναν multiprocessor, κάτι που μπορεί να επηρεάσει την αποδοτικότητα και την απόδοση της εκτέλεσης του kernel.

Με τον παρακάτω κώδικα καλούμε την cudaOccupancyMaxPotentialBlockSize για να μας δώσει τις προτεινόμενες τιμές blockSize και minGridSize για την kernel function find\_nearest\_cluster:

```

int minGridSize, blockSize;
checkCuda(cudaOccupancyMaxPotentialBlockSize(
    &minGridSize, // Minimum grid size
    &blockSize, // Optimal block size
    find_nearest_cluster, // Kernel function
    0, // Dynamic shared memory size per block (0 if none)
    0 // Maximum number of blocks (0 for default)
));
printf("Optimal block size: %d\n", blockSize);
printf("Suggested minimum grid size: %d\n", minGridSize);
printf("Final grid size: %d\n", numClusterBlocks);

```

Τα αποτελέσματα που πήραμε φαίνονται παρακάτω:

Table 1: Configuration: {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}

	blockSize	best blockSize from experiments	minGridSize	numClusterBlocks
Naive	1024	32	20	4096
Transpose	1024	1024	20	4096
Shared	1024	1024	20	4096
All-GPU	1024	512	20	4096
All-GPU+delta reduction	1024	512	20	4096

Table 2: Configuration: {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10}

	blockSize	best blockSize from experiments	minGridSize	numClusterBlocks
Naive	1024	64	20	65536
Transpose	1024	128	20	65536
Shared	1024	1024	20	65536
All-GPU	1024	512	20	65536
All-GPU+delta reduction	1024	512	20	65536

Η συνάρτηση cudaOccupancyMaxPotentialBlockSize προτείνει σταθερά το μέγιστο blockSize των 1024, καθώς αυτό εξασφαλίζει τη μέγιστη αξιοποίηση του GPU occupancy. Αυτό συμβαίνει επειδή το occupancy, δηλαδή η αναλογία ενεργών warps προς τον μέγιστο αριθμό υποστηριζόμενων warps σε έναν multiprocessor, μεγιστοποιείται όταν χρησιμοποιείται ο μέγιστος αριθμός threads ανά block. Ωστόσο, στις πειραματικές μετρήσεις φαίνεται ότι η βέλτιστη απόδοση συχνά προκύπτει από μικρότερα block sizes, πιθανώς λόγω μειωμένης συμφόρησης στους καταχωρητές και τη shared memory. Όσον αφορά το gridSize, η τιμή του εξαρτάται από το μέγεθος του dataset και διαμορφώνεται έτσι ώστε να καλύπτει επαρκώς το σύνολο των αντικειμένων (numObjs). Στην προκειμένη περίπτωση, η επιλογή του gridSize εξασφαλίζει ότι υπάρχει πλήρης κάλυψη του dataset, με τη χρήση επαρκών blocks για την επεξεργασία όλων των αντικειμένων.

### 3.1.5 Full-Offload (All-GPU) version

Σε αυτήν την έκδοση υλοποιούμε ολόκληρο τον kmeans στην GPU, δηλαδή στην προηγούμενη shared προσθέτουμε και τον υπολογισμό των νέων cluster centroid (update\_centroids) στην GPU.

Για τον υπολογισμό των nearest clusters σε κάθε iteration χρησιμοποιούμε την παρακάτω συνάρτηση:

```

__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
   */

    int *devicenewClusterSize, double *devicenewClusters,
    double *deviceClusters,      // [numCoords][numClusters]
    int *deviceMembership,       // [numObjs]
    double *devdelta) {

extern __shared__ double shmemClusters[];

/* TODO: copy me from shared version... */

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

int i;
for (i = threadIdx.x; i < numClusters * numCoords; i += blockDim.x) {
    shmemClusters[i] = deviceClusters[i];
}

__syncthreads();

/* TODO: copy me from shared version... */
if (tid < numObjs) {

/* TODO: copy me from shared version... */

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

```

Η συνάρτηση που χρησιμοποιούμε για τον υπολογισμό των νέων cluster centroid στην GPU φαίνεται παρακάτω:

```

__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 totalThreads = gridDim.x * blockDim.x;

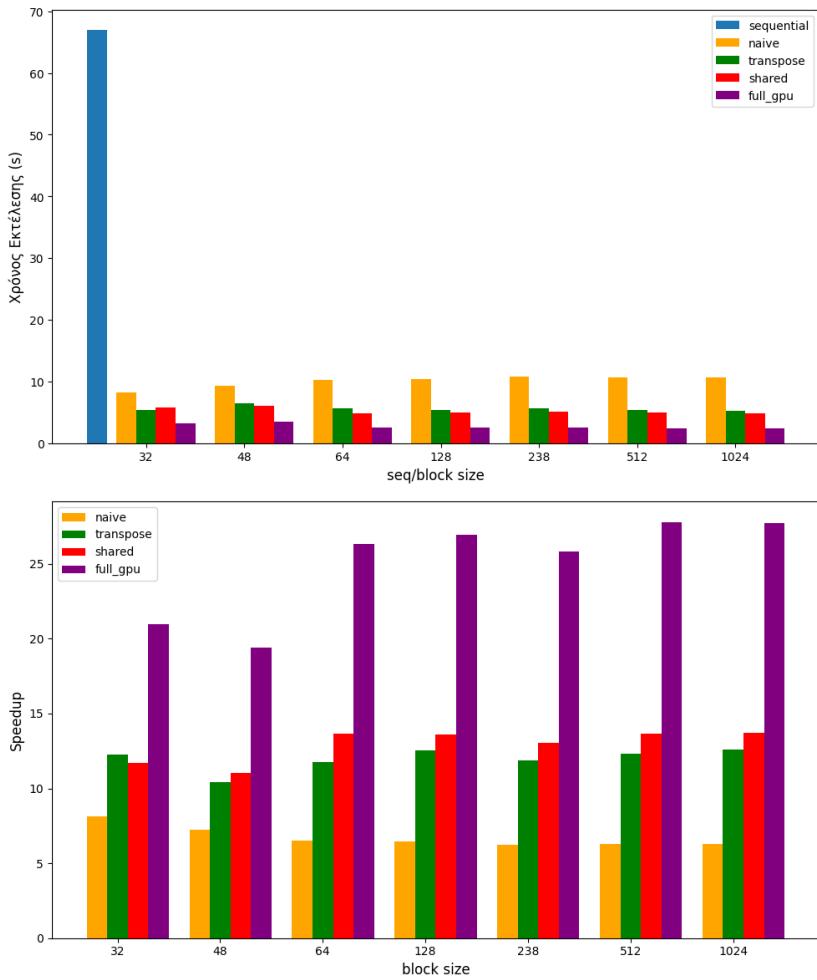
// Update centroids
for (int i = tid; i < numCoords * numClusters; i += totalThreads) {
    int clusterId = i % numClusters;
    int coordId = i / numClusters;

    if (devicenewClusterSize[clusterId] > 0) {
        deviceClusters[i] = devicenewClusters[i] / devicenewClusterSize[clusterId];
    }
}

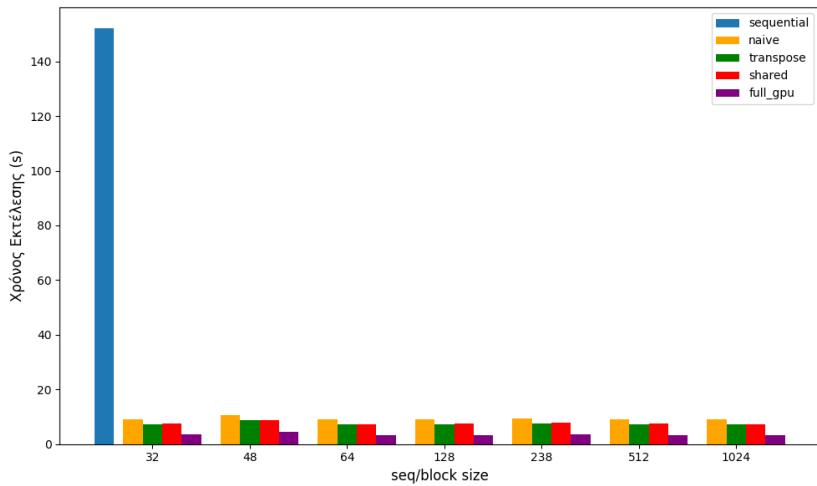
// Reset new clusters and sizes
for (int i = tid; i < numCoords * numClusters; i += totalThreads) {
    devicenewClusters[i] = 0.0;
}
for (int i = tid; i < numClusters; i += totalThreads) {
    devicenewClusterSize[i] = 0;
}
}
}
}
}

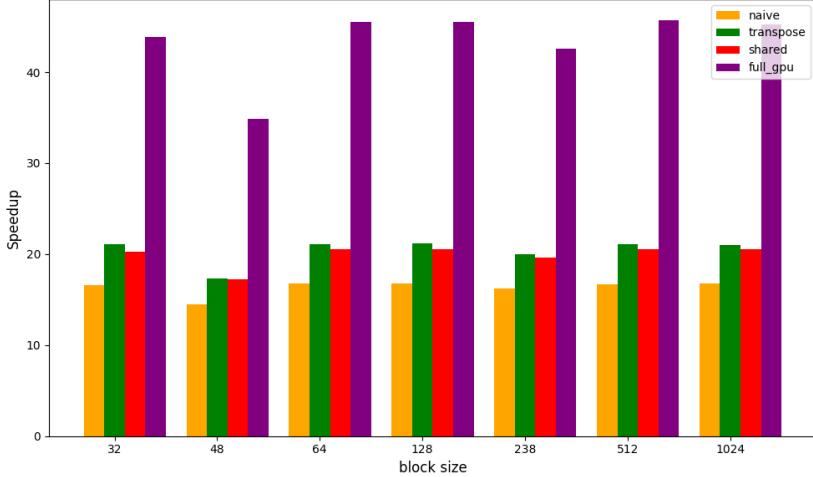
```

- Παρακάτω φαίνονται τα διαγράμματα για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block\_size = {32, 48, 64, 128, 238, 512, 1024}



Παρακάτω φαίνονται τα διαγράμματα για το configuration  $\{\text{Size}, \text{Coords}, \text{Clusters}, \text{Loops}\} = \{1024, 2, 64, 10\}$  και για  $\text{block\_size} = \{32, 48, 64, 128, 238, 512, 1024\}$





Η all-GPU version παρουσιάζει σημαντική βελτίωση στην επίδοση σε σχέση με τις άλλες υλοποιήσεις (naive, transpose, shared) σε όλες τις περιπτώσεις block sizes και configurations. Αυτό οφείλεται στην πλήρη χρήση του παράλληλου υπολογισμού στην GPU για όλες τις φάσεις του αλγορίθμου. Τα διαγράμματα speedup δείχνουν ότι η all-GPU version επιτυγχάνει σημαντικά υψηλότερο speedup, ειδικά για το configuration με numCoords = 32, όπου η μείωση του overhead από τη μεταφορά δεδομένων και η καλύτερη διαχείριση των πόρων της GPU παίζουν μεγαλύτερο ρόλο. Αντίθετα, οι παλιότερες εκδόσεις περιορίζονταν είτε από τη μεταφορά δεδομένων είτε από το μικρότερο βαθμό παράλληλης εκμετάλλευσης.

2. To block\_size παίζει διαφορετικό ρόλο ανάλογα με το configuration. Σε μεγαλύτερα block sizes (π.χ. 512 ή 1024), παρατηρείται μικρή μείωση του χρόνου εκτέλεσης για την all-GPU version, ειδικά στο configuration με numCoords = 32, κανός αυξάνεται η παράλληλη χρήση των πόρων της GPU. Ωστόσο, για configurations με μικρό αριθμό συντεταγμένων (numCoords = 2), η επίδραση του block\_size είναι μικρότερη λόγω του περιορισμένου αριθμού υπολογισμών ανά thread. Γενικά, το block\_size επηρεάζει την απόδοση κυρίως μέσω της κατανομής threads και της χρήσης της shared memory.

3. Η update\_centroids είναι λιγότερο κατάλληλη για GPUs λόγω της σχετικής ασυμμετρίας στον υπολογισμό (δηλ. λίγοι υπολογισμοί ανά thread) και της εξάρτησης από atomics. Ωστόσο, στην all-GPU version, αυτό το εμπόδιο μειώνεται μέσω της πιο αποδοτικής διαχείρισης των atomics και της κατανομής εργασιών στη GPU. Σε σχέση με τις παλιές εκδόσεις, η διαφορά στην επίδοση οφείλεται κυρίως στο ότι αποφεύγονται οι μεταφορές δεδομένων μεταξύ CPU και GPU, οι οποίες αποτελούσαν σημαντικό bottleneck στις naive και transpose εκδόσεις.

4. Η διαφορά μεταξύ των δύο configurations έγκειται στον αριθμό των συντεταγμένων (numCoords) και στον αριθμό των υπολογισμών που απαιτούνται ανά thread. Στο configuration με numCoords = 32, υπάρχει μεγαλύτερο computational load ανά thread, οδηγώντας σε καλύτερη εκμετάλλευση των πόρων της GPU. Αντίθετα, στο configuration με numCoords = 2, το χαμηλότερο computational load περιορίζει τη συνολική απόδοση της GPU, και η σχετική διαφορά μεταξύ των εκδόσεων είναι μικρότερη. Ωστόσο, ακόμα και σε αυτό το configuration, η all-GPU version υπερέχει λόγω της πλήρους αξιοποίησης της GPU για όλους τους υπολογισμούς.

### 3.1.6 Bonus 2: Delta reduction (All-GPU) version

Σε αυτή την έκδοση του αλγορίθμου όλα υπολογίσουμε το delta με reduction (αντί για global atomics!). Ακολουθούμε τις οδηγίες στην `find_nearest_cluster` για να υλοποιήσουμε δεντρικό reduction σε κάθε block.

Παρακάτω φαίνεται το μέγεθος της `shmem` του πυρήνα σε αυτή την έκδοση του αλγορίθμου:

```
const unsigned int clusterBlockSharedDataSize = (numClusters * numCoords * sizeof(double)) +
                                              (numThreadsPerClusterBlock * sizeof(double));
```

Παρακάτω φαίνεται η `find_nearest_cluster` που χρησιμοποιήθηκε σε αυτή την έκδοση του αλγορίθμου:

```

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

extern __shared__ double shmem_total[];
double *shmemClusters = shmem_total;
double* deltas_buffer = &shmem_total[numClusters * numCoords];
int local_tid = threadIdx.x;

/* TODO: copy me from shared version... */

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

int i;
for (i = threadIdx.x; i < numClusters * numCoords; i += blockDim.x) {
    shmemClusters[i] = deviceClusters[i];
}

__syncthreads();

// Initialize the delta reduction buffer
deltas_buffer[local_tid] = 0.0;

/* TODO: copy me from shared version... */
if (tid < numObjs) {

    /* TODO: copy me from shared version... */
    int index, i;
    double dist, min_dist;

    index = 0;

    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
                                       shmemClusters, tid, 0);

    for (i = 1; i < numClusters; i++) {
        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;
        }
    }
}

```

```

/* TODO: Replacing (*devdelta)+= 1.0; with reduction:
   - each thread updates the single element of delta_reduce_buff
   corresponding to its local id (threadIdx.x) -> 1.0 if membership changes, otherwise 0.
   - Then, ensuring delta_reduce_buff is fully updated, its contents must be summed in
   delta_reduce_buff[0]
   either by one thread (lower perf) or with a tree-based reduction (similar to dot reduction
   example in slides)
   - Finally, delta_reduce_buff[0] (local value in block) must be added to devdelta (global
   delta value), ensuring write dependencies!
*/
/* TODO: additional steps for calculating new centroids in GPU? */

// Check if membership changes
if (deviceMembership[tid] != index) {
    deltas_buffer[local_tid] = 1.0; // Mark delta for this thread
    deviceMembership[tid] = index; // Update membership
}

// Update cluster sizes and centroids
atomicAdd(&devicenewClusterSize[index], 1);
for (int coord = 0; coord < numCoords; coord++) {
    atomicAdd(&devicenewClusters[coord * numClusters + index],
              deviceobjects[coord * numObjs + tid]);
}

// Ensure all threads have updated deltas_buffer
__syncthreads();

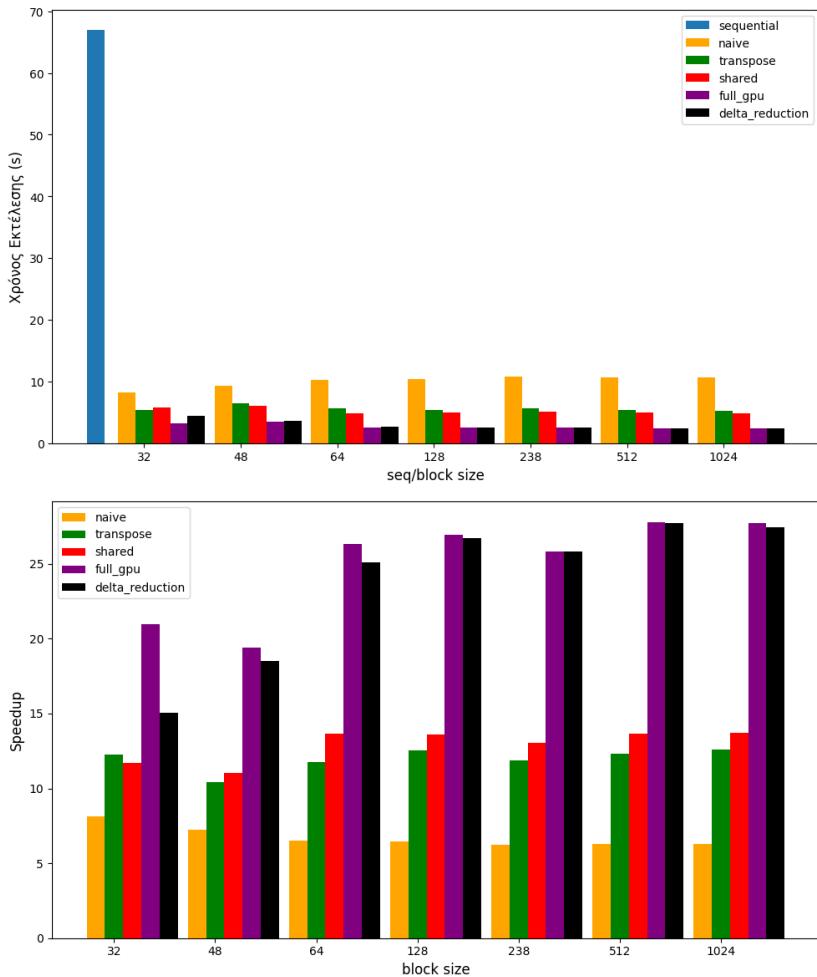
// Perform reduction on deltas_buffer to compute block's contribution to delta
for (int offset = blockDim.x / 2; offset > 0; offset /= 2) {
    if (local_tid < offset) {
        deltas_buffer[local_tid] += deltas_buffer[local_tid + offset];
    }
    __syncthreads();
}

// Add block's delta contribution to global delta
if (local_tid == 0) {
    atomicAdd(devdelta, deltas_buffer[0]);
}
}

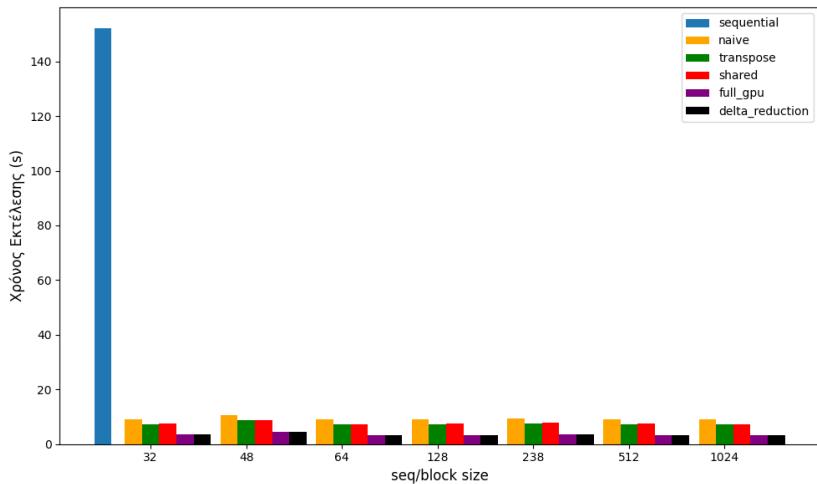
}

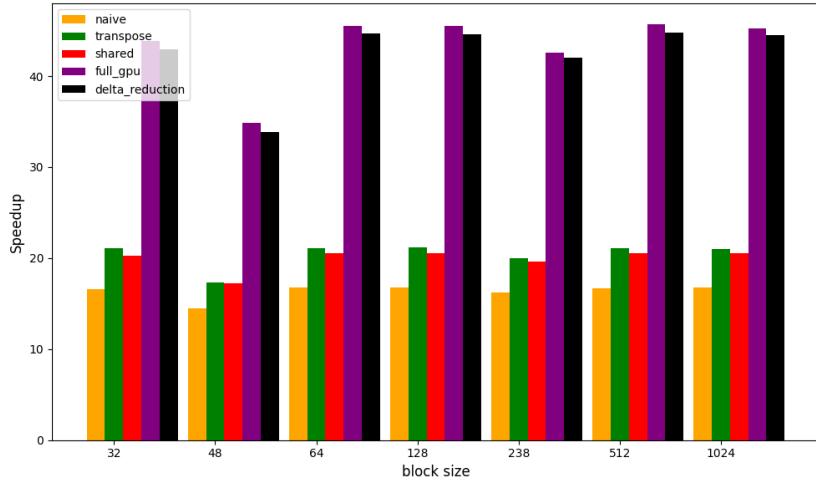
```

- Παρακάτω φαίνονται τα διαγράμματα για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block\_size ={32, 48, 64, 128, 238, 512, 1024}



Παρακάτω φαίνονται τα διαγράμματα για το configuration {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} και για block\_size = {32, 48, 64, 128, 238, 512, 1024}





Η έκδοση "all-gpu delta reduction" παρουσιάζει μικρές αποκλίσεις σε σχέση με την "all-gpu" έκδοση, με οριακά χειρότερες επιδόσεις στις περισσότερες περιπτώσεις. Αυτό πιθανότατα οφείλεται στη χρήση του reduction για τον υπολογισμό του delta, η οποία εισάγει κάποιο overhead λόγω των συγχρονισμών και της πολυπλοκότητας της μεθόδου. Παρόλο που η μείωση βελτιώνει τη συνοχή της ενημέρωσης του delta, η αυξημένη πολυπλοκότητα μπορεί να αναιρέι μέρος του οφέλους από τον ταχύτερο υπολογισμό.

2. Το block\_size επηρεάζει σημαντικά την απόδοση επειδή καθορίζει τον αριθμό των threads που μπορούν να συνεργαστούν για τη μείωση στο shared memory. Στην "delta reduction" έκδοση, μεγαλύτερα block sizes επιτρέπουν τη χρήση περισσότερων threads για τη μείωση του delta, γεγονός που οδηγεί σε καλύτερη αξιοποίηση της parallelism των blocks. Καθώς αυξάνεται το block size, η διαδικασία μείωσης γίνεται πιο αποδοτική, αφού περισσότερα threads μπορούν να εκτελούν ταυτόχρονα τα βήματα της μείωσης και να συγκεντρώνουν αποτελεσματικά τα δεδομένα στο shared memory, πριν γίνει η τελική ενημέρωση του delta στη global memory. Έτσι, για μεγάλα block sizes, η μείωση επιτυγχάνει υψηλότερη απόδοση, καθώς μεώνονται οι συγχρούσεις μνήμης και το latency της global memory.

## 4 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

### 4.1 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

#### 4.1.1 Υλοποίηση σε MPI

Αρχικά στο αρχείο `file_io.c` συμπληρώνουμε τα κόμματα που λείπουν στην συνάρτηση `dataset_generation()` όπως φαίνεται παρακάτω:

Τυπολογίζουμε τον αριθμό των objects που κάθε διεργασία θα εξετάσει. Ο αριθμός των objects δεν είναι απαραίτητα ίσος για κάθε διεργασία, εάν το `numObjs` δεν είναι πολλαπλάσιο του `size`.

```
*rank_numObjs = numObjs / size + (rank < (numObjs % size) ? 1 : 0);
```

Για να διανεμηθούν τα δεδομένα στις διεργασίες, η διεργασία 0 αρχικά υπολογίζει δύο πίνακες:

- `sendcounts[size]`: Καθορίζει πόσα στοιχεία αποστέλλονται σε κάθε διεργασία.
- `displs[size]`: Καθορίζει τις θέσεις εκκίνησης των δεδομένων που θα αποσταλούν.

```
int sendcounts[size], displs[size];
if (rank == 0) {
    objects = (double *) malloc(numObjs * numCoords * sizeof(double));
    int offset = 0;
    for (i = 0; i < size; i++) {
        sendcounts[i] = (numObjs / size + (i < (numObjs % size) ? 1 : 0)) * numCoords;
        displs[i] = offset;
        offset += sendcounts[i];
    }
}
```

Κάνουμε Broadcast τους πίνακες `sendcounts` και `displs` στις άλλες διεργασίες:

```
MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);
```

Κάνουμε scatter τα κατάλληλα objects σε κάθε διεργασία:

```
MPI_Scatterv(objects, sendcounts, displs, MPI_DOUBLE, rank_objects,
              (*rank_numObjs) * numCoords, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Έπειτα στο αρχείο `kmeans.c` συμπληρώνουμε τα κομμάτια που λείπουν από την συνάρτηση `kmeans` όπως φαίνεται παρακάτω: Κάθε διεργασία επεξεργάζεται ένα υποσύνολο δεδομένων και υπολογίζει τοπικά τις συντεταγμένες των νέων κέντρων clusters. Αυτά συγχωνεύονται για να υπολογιστεί το τελικό αποτέλεσμα όπως φαίνεται παρακάτω:

```
MPI_Allreduce(rank_newClusters, newClusters, numClusters * numCoords, MPI_DOUBLE, MPI_SUM,
               MPI_COMM_WORLD);
MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT, MPI_SUM,
               MPI_COMM_WORLD);
```

Παρόμοια με παραπάνω κάνουμε reduction για να υπολογίσουμε την `delta_variable` που χρησιμοποιείται για τον έλεγχο σύγκλισης του αλγορίθμου. Ο αλγόριθμος επαναλαμβάνεται μέχρι να επιτευχθεί σύγκλιση, δηλαδή όταν ένα ποσοστό των σημείων σταματήσει να αλλάζει συστάδα (ή οταν κάνουμε `#loops == loop_threshold`).

```
MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Έπειτα στο αρχείο `main.c` συμπληρώνουμε τα κομμάτια που λείπουν από την συνάρτηση `main()` όπως φαίνεται παρακάτω:

Κάνουμε Broadcast τις αρχικές θέσεις των clusters σε όλες τις διεργασίες:

```
MPI_Bcast(clusters, numClusters * numCoords, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Καλούμε την συνάρτηση που τρέχει τον αλγόριθμο k-means:

```
kmeans(objects, numCoords, rank_numObjs, numClusters, threshold, loop_threshold,
       membership, clusters);
```

Για να γίνουν gather τα δεδομένα από τις διεργασίες, η διεργασία 0 αρχικά υπολογίζει δύο πίνακες:

- sendcounts[size]: Καθορίζει πόσα στοιχεία γίνονται gathered από κάθε διεργασία.
- displs[size]: Καθορίζει τις θέσεις εκκίνησης των δεδομένων που θα γίνουν gathered.

```
int recvcounts[size], displs[size];
if (rank == 0) {
    int total = 0;
    for (i = 0; i < size; i++) {
        recvcounts[i] = (numObjs / size) + (i < (numObjs % size) ? 1 : 0);
        displs[i] = total;
        total += recvcounts[i];
    }
}
```

Κάνουμε Broadcast τους πίνακες recvcounts και displs στις άλλες διεργασίες:

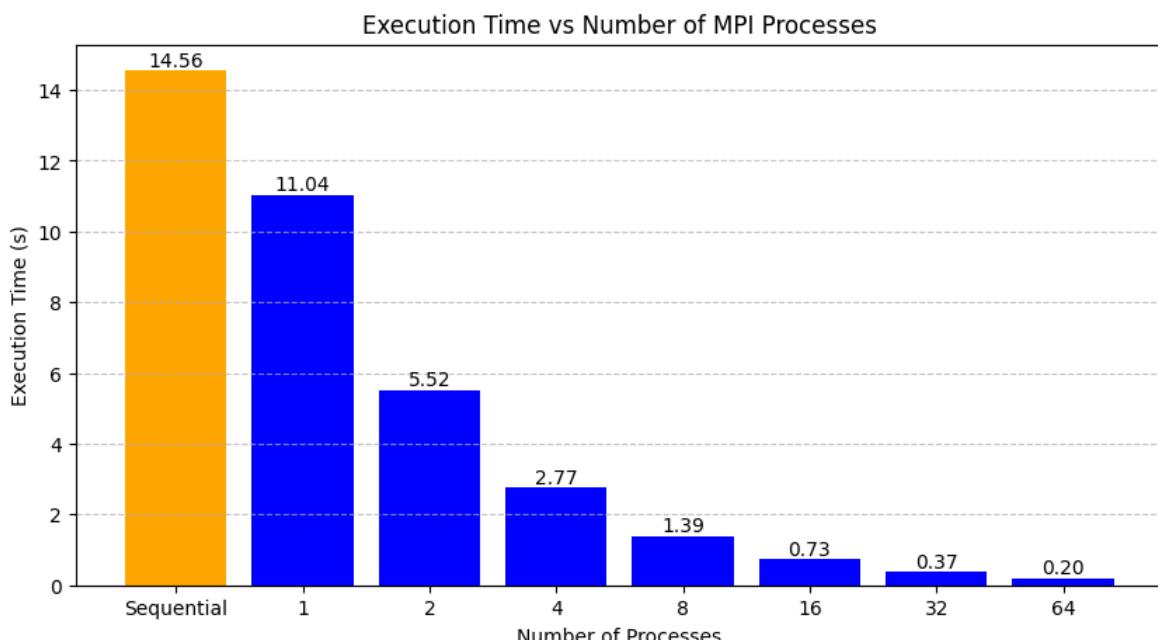
```
MPI_Bcast(recvcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);
```

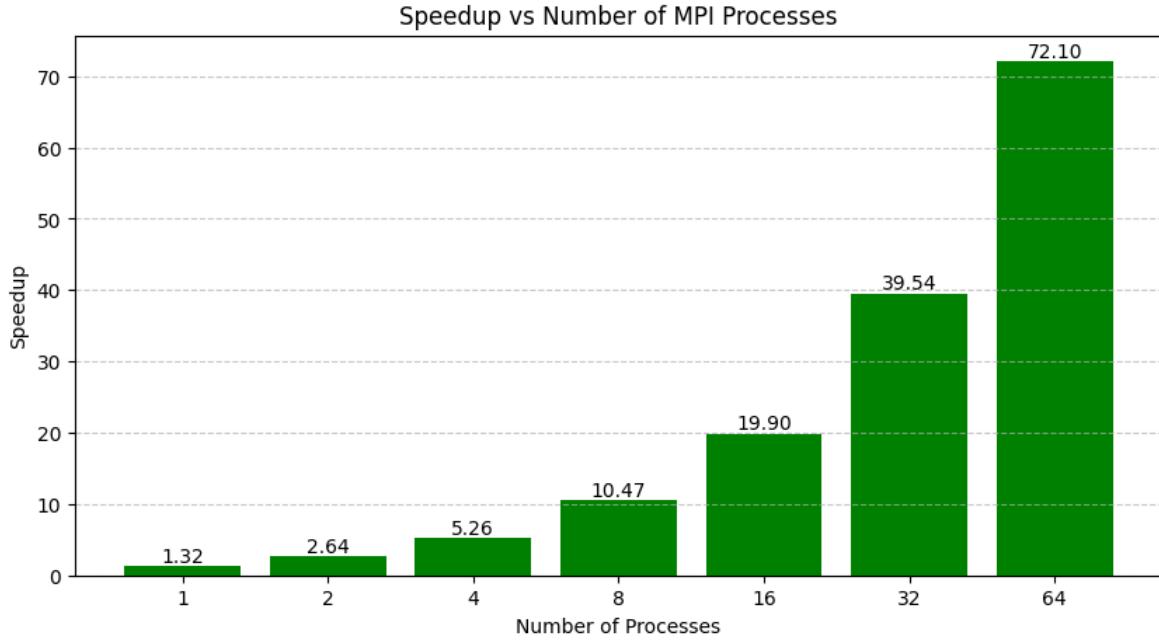
Κάνουμε Gather τις πληροφορίες για το membership των στοιχείων από κάθε διεργασία:

```
MPI_Gatherv(membership, rank_numObjs, MPI_INT,
             tot_membership, recvcounts, displs, MPI_INT,
             0, MPI_COMM_WORLD);
```

#### 4.1.2 Αποτελέσματα Μετρήσεων και Σχολιασμός

Παρακάτω φαίνονται τα barplots χρόνου εκτέλεσης και speedup για configuration Size, Coords, Clusters, Loops = {256, 16, 32, 10} για 1, 2, 4, 8, 16, 32 και 64 MPI διεργασίες στα clones:

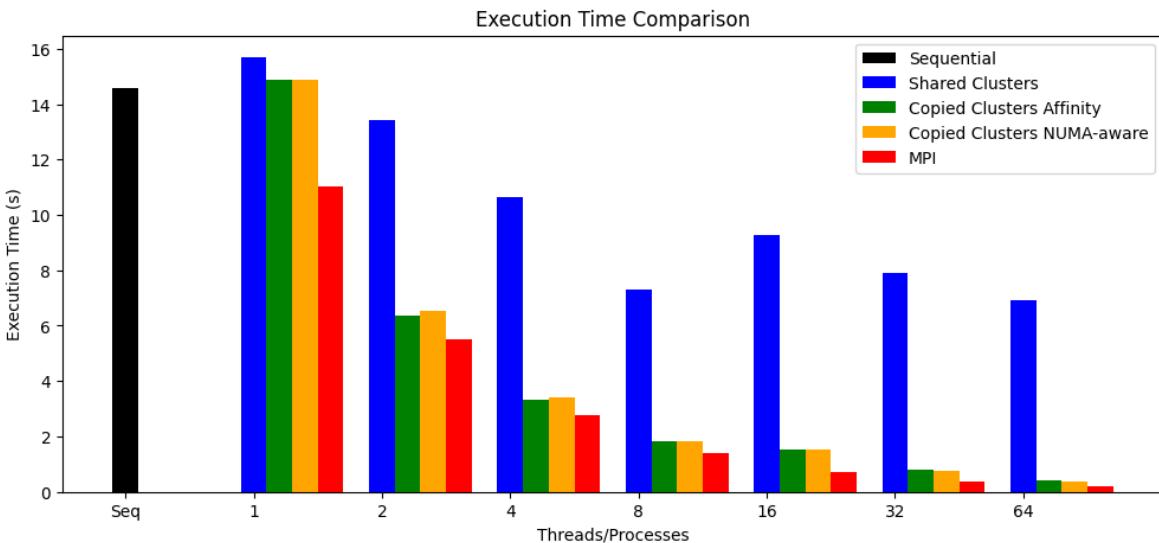


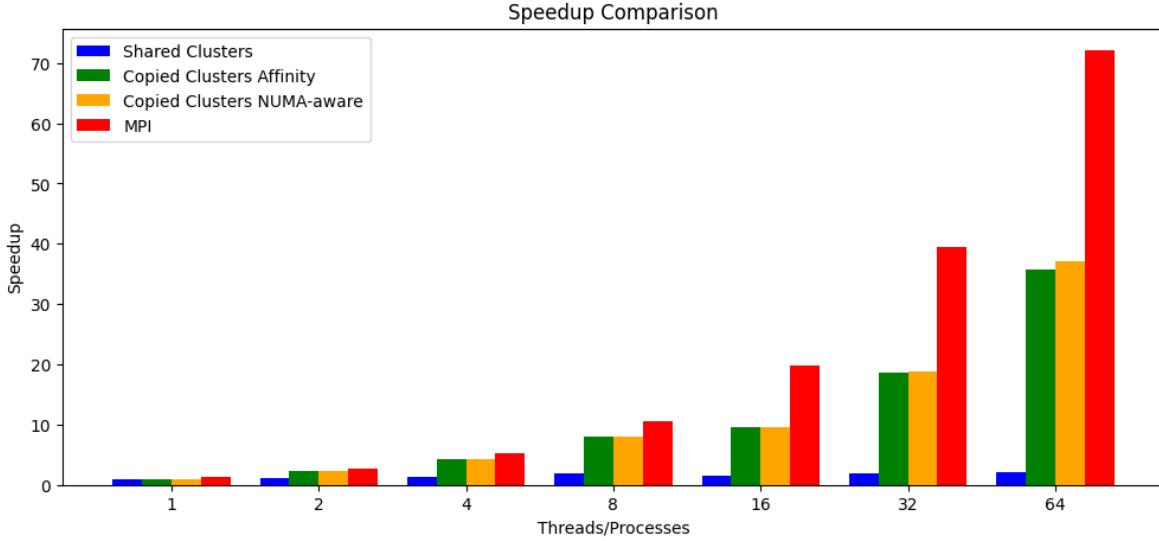


Η MPI υλοποίηση παρουσιάζει εξαιρετική απόδοση. Παρατηρούμε ότι μέχρι και τα 32 processes το speedup είναι σχεδόν γραμμικό, γεγονός που δείχνει ότι η παράλληλη επεξεργασία κλιμακώνεται πολύ αποδοτικά. Ωστόσο, στα 64 processes, παρόλο που η επιτάχυνση παραμένει σημαντική, δεν είναι πλέον γραμμική. Αυτό πιθανότατα οφείλεται στα overheads που προκύπτουν από την αυξημένη επιχοινωνία μεταξύ των διεργασιών, καθώς και σε πιθανή ανισορροπία στον καταμερισμό του φορτίου.

#### 4.1.3 Bonus: Σύγκριση με OpenMP Υλοποίηση

Παρακάτω συγκρίνουμε τις υλοποιήσεις με OpenMP με την υλοποίηση με MPI:





Η υλοποίηση με MPI παρουσιάζει σημαντικά μικρότερους χρόνους εκτέλεσης σε σχέση με την OpenMP, ειδικά καθώς αυξάνεται ο αριθμός των πυρήνων. Για 64 πυρήνες, η MPI υλοποίηση επιτυγχάνει τον χαμηλότερο χρόνο, ξεπερνώντας ακόμα και την πιο αποδοτική OpenMP προσέγγιση με NUMA-Aware allocation. Αυτό υποδηλώνει ότι η προσέγγιση κατανεμημένης μνήμης της MPI κλιμακώνεται καλύτερα από την OpenMP, ιδιαίτερα όταν ο αριθμός των πυρήνων αυξάνεται σημαντικά. Η OpenMP, ακόμη και με NUMA-aware κατανομή, επηρεάζεται περισσότερο από τα context switches και τον περιορισμένο έλεγχο της τοποθέτησης των δεδομένων στη μνήμη. Ωστόσο, η χρήση της MPI απαιτεί υψηλότερο overhead λόγω επικοινωνίας μεταξύ διεργασιών, κάτι που μπορεί να είναι αποδοτικό σε μεγάλο αριθμό κόμβων, αλλά όχι πάντα σε single-node συστήματα όπου η OpenMP μπορεί να είναι πιο αποδοτική, κάτι που δεν παρατηρήθηκε όμως στα πειταματικά αποτελέσματα.

## 4.2 Διάδοση Θερμότητας σε δύο διαστάσεις

Στόχος μας είναι να αναπτύξουμε παράλληλο πρόγραμμα στο μοντέλο ανταλλαγής μηνυμάτων με την βοήθεια της βιβλιοθήκης MPI. Στο directory `/home/parallel/parlab20/a4/heat_transfer/mpi` του scirouter βρίσκονται οι υλοποίησεις των παραχάτων ωγορίζμων. Στο αρχείο `mpi_skeleton.c` μας δίνεται σκελετός υλοποίησης σε MPI, στον οποίο καλούμαστε να συμπληρώσουμε τον κώδικάς μας. Συγκεκριμένα βασιζόμενοι στο `mpi_skeleton.c` έχουμε φτιάξει 3 αρχεία τα `mpi_jacobi.c`, `mpi_gauss_seidel_sor.c` και `mpi_red_black_sor.c`, τα οποία υλοποιούν σε MPI τις μεθόδους Jacobi, Gauss-Seidel SOR και Red-Black SOR αντίστοιχα.

### 4.2.1 Μεδοδος Jacobi

Ο κύριος υπολογιστικός πυρήνας της μεθόδου Jacobi αποτελείται από δύο βασικά μέρη: την επικοινωνία μεταξύ των διεργασιών και τον υπολογισμό των νέων τιμών της μήτρας. Αρχικά, πριν από κάθε νέο υπολογισμό, οι διεργασίες ανταλλάσσουν δεδομένα με τους γειτονικούς τους κόμβους ώστε να ενημερώσουν τα σύνορά τους. Η ανταλλαγή γίνεται με τη χρήση της συνάρτησης MPI\_Sendrecv, η οποία επιτρέπει ταυτόχρονη αποστολή και λήψη δεδομένων, αποφεύγοντας έτσι τον κίνδυνο αδιεξόδου (deadlock). Κάθε διεργασία στέλνει τις τιμές των όγκων της μήτρας της στους γειτονικούς κόμβους και λαμβάνει αντίστοιχα τις συνοριακές τιμές από αυτούς. Ειδικότερα, αν υπάρχει γειτονικός κόμβος προς τα πάνω (βόρειος γείτονας), η διεργασία στέλνει την πρώτη εσωτερική της γραμμή και λαμβάνει τη γραμμή φάντασμα από τον γείτονά της. Αν υπάρχει γειτονικός κόμβος προς τα κάτω (νότιος γείτονας), τότε στέλνει την τελευταία εσωτερική της γραμμή και λαμβάνει τη γραμμή φάντασμα από τον νότιο γείτονα. Παρόμοια, αν υπάρχει ανατολικός γείτονας, η διεργασία στέλνει την τελευταία εσωτερική στήλη της και λαμβάνει τη στήλη φάντασμα από τον ανατολικό γείτονα, ενώ αν υπάρχει δυτικός γείτονας, τότε στέλνει την πρώτη εσωτερική της στήλη και λαμβάνει την αντίστοιχη φάντασμα. Αν κάποιος από αυτούς τους γείτονες δεν υπάρχει, δηλαδή αν η διεργασία βρίσκεται στα όρια του πλέγματος των διεργασιών, η αποστολή και λήψη δεδομένων παραλείπεται. Αφού ολοκληρωθεί η επικοινωνία, κάθε διεργασία προχωρά στον υπολογισμό

των νέων τιμών της μήτρας της. Ο υπολογισμός βασίζεται στην κλασική εξίσωση του Jacobi, όπου κάθε στοιχείο της μήτρας αντικαθίσταται από τον μέσο όρο των τεσσάρων γειτονικών του τιμών. Η επανάληψη του υπολογισμού γίνεται μέσα σε ένα διπλό βρόχο for, όπου κάθε στοιχείο (i, j) ανανεώνεται σύμφωνα με τις τιμές των γειτονικών κελιών της προηγούμενης επανάληψης. Τα όρια του υπολογισμού έχουν καθοριστεί ώστε να αποφεύγονται προσπελάσεις σε μη διαυλέσμιες θέσεις, κάτι που είναι ιδιαίτερα σημαντικό για τις διεργασίες που βρίσκονται στα όρια του πλέγματος. Η εναλλαγή μεταξύ της προηγούμενης και της τρέχουσας μήτρας επιτυγχάνεται μέσω εναλλαγής δεικτών, ώστε να μην απαιτείται η αντιγραφή των δεδομένων και να μειώνεται το κόστος εκτέλεσης. Για την ακριβή μέτρηση της απόδοσης, χρησιμοποιούνται χρονόμετρα. Πριν ξεκινήσει ο κύριος υπολογιστικός βρόχος, αρχικοποιείται ένας χρονομετρητής ώστε να καταγράψει τον συνολικό χρόνο εκτέλεσης. Επιπλέον, πριν από την έναρξη του υπολογισμού, καταγράφεται ο χρόνος έναρξης των υπολογισμών, και μόλις ολοκληρωθεί ο υπολογισμός των νέων τιμών, καταγράφεται ο χρόνος λήξης και αποθηκεύεται η διαφορά στο tcomp, που αντιπροσωπεύει το συνολικό χρόνο υπολογισμού. Με αυτόν τον τρόπο, είναι δυνατή η ακριβής ανάλυση της απόδοσης του αλγορίθμου και η αξιολόγηση του κόστους επικοινωνίας σε σχέση με τον καθαρό υπολογισμό. Ο σχεδιασμός του υπολογιστικού πυρήνα ακολουθεί μια τυπική στρατηγική βελτιστοποίησης παραλλήλων αλγορίθμων, όπου διαχωρίζεται η επικοινωνία από τον υπολογισμό, η ανταλλαγή δεδομένων γίνεται με αποδοτικό τρόπο μέσω MPI\_Sendrecv, και η χρήση δεικτών για την εναλλαγή των πινάκων μειώνει την περιττή χρήση μνήμης. Έτσι, η υλοποίηση είναι κλιμακώσιμη και αποδοτική για μεγάλο αριθμό διεργασιών και μεγάλες διαστάσεις μήτρας.

#### 4.2.2 (Bonus) Μέθοδος Gauss-Seidel SOR

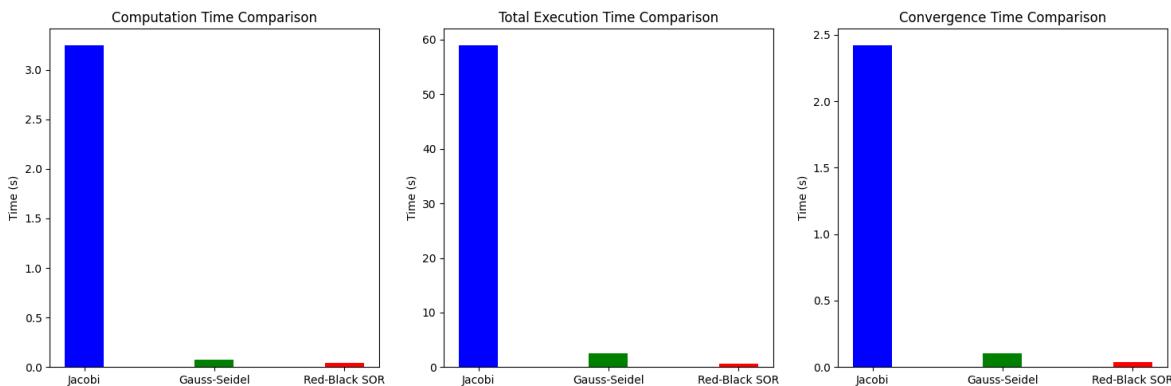
Ο υπολογιστικός πυρήνας της μεθόδου Gauss-Seidel SOR (Successive Over-Relaxation) παρουσιάζει σημαντικές διαφορές σε σχέση με τη μέθοδο Jacobi, τόσο στη διαδικασία επικοινωνίας όσο και στον τρόπο ενημέρωσης των τιμών του πλέγματος. Ενώ στη μέθοδο Jacobi κάθε σημείο του πλέγματος ενημερώνεται αποκλειστικά με βάση τις τιμές της προηγούμενης επανάληψης, στη Gauss-Seidel SOR οι τιμές ενημερώνονται διαδοχικά, χρησιμοποιώντας ήδη υπολογισμένες νέες τιμές από την τρέχουσα επανάληψη. Η επικοινωνία μεταξύ των διεργασιών είναι πιο περίπλοκη, καθώς η μέθοδος απαιτεί διαδοχική ενημέρωση των τιμών του πλέγματος, γεγονός που δημιουργεί εξαρτήσεις μεταξύ των υπολογισμών διαφορετικών διεργασιών. Στην αρχή κάθε επανάληψης, οι διεργασίες ανταλλάσσουν δεδομένα με τους γειτονικούς κόμβους. Συγκεκριμένα, η βόρεια διεργασία λαμβάνει ενημερωμένες τιμές από τη γειτονική της και στέλνει τα δικά της δεδομένα, ενώ οι νότιοι και ανατολικοί γείτονες απλώς λαμβάνουν τις ενημερωμένες τιμές από την προηγούμενη επανάληψη. Η ανταλλαγή των δεδομένων προς τα δυτικά γίνεται μέσω της MPI\_Sendrecv, παρόμοια με τη μέθοδο Jacobi. Μετά την ολοκλήρωση του υπολογισμού της τοπικής μήτρας, οι διεργασίες πρέπει να εξασφαλίσουν ότι οι γειτονικές τους ενημερώνονται σωστά. Εδώ η επικοινωνία με τις νότιες και ανατολικές διεργασίες συμβαίνει στο τέλος κάθε επανάληψης, καθώς οι τιμές αυτών των περιοχών μόλις έχουν ενημερωθεί και πρέπει να σταλούν. Οι επικοινωνίες αυτές γίνονται με απλά MPI\_Send αντί για MPI\_Sendrecv, επειδή η αποστολή γίνεται στο τέλος του υπολογισμού και δεν χρειάζεται να προηγηθεί λήψη.

#### 4.2.3 (Bonus) Μέθοδος Red-Black SOR

Ο υπολογιστικός πυρήνας της μεθόδου Red-Black SOR (Successive Over-Relaxation) αποτελεί μια βελτιστοποιημένη παραλλαγή της Gauss-Seidel SOR, η οποία βασίζεται στην εναλλασσόμενη ενημέρωση των τιμών του πλέγματος χρησιμοποιώντας έναν διαχωρισμό σε δύο φάσεις: την Red Phase και την Black Phase. Αυτή η τεχνική χωρίζει το πλέγμα σε δύο ομάδες σημείων, που αντιστοιχούν σε ένα μοτίβο τύπου "σκακιέρας", και εξασφαλίζει ότι κάθε σημείο ενημερώνεται χρησιμοποιώντας ήδη υπολογισμένες τιμές της τρέχουσας επανάληψης. Στην αρχή κάθε επανάληψης, όπως και στις προηγούμενες μεθόδους, γίνεται ανταλλαγή δεδομένων με τις γειτονικές διεργασίες, ώστε κάθε υποπεριοχή να έχει πρόσβαση στις επικαρποιημένες τιμές των γειτόνων της. Η επικοινωνία αυτή υλοποιείται μέσω MPI\_Sendrecv, όπου κάθε διεργασία στέλνει τα δεδομένα των άκρων της προς τις γειτονικές διεργασίες και λαμβάνει αντίστοιχα τις συνοριακές τιμές τους. Αυτή η ανταλλαγή πραγματοποιείται πριν από κάθε φάση (Red και Black), ώστε να διασφαλιστεί ότι κάθε διεργασία έχει τις σωστές τιμές των γειτονικών κελιών πριν εκτελέσει τις ενημερώσεις. Η Red Phase ξεκινά αμέσως μετά την πρώτη ανταλλαγή δεδομένων και περιλαμβάνει τον υπολογισμό των τιμών για τα σημεία του πλέγματος που βρίσκονται σε θέσεις όπου η ανθροιστική συντεταγμένη (i+j) είναι άρτια. Αυτή η επιλογή διαχωρίζει τα σημεία που υπολογίζονται στη συγκεκριμένη φάση από εκείνα που θα ενημερωθούν στη δεύτερη φάση. Ο τύπος ενημέρωσης των τιμών

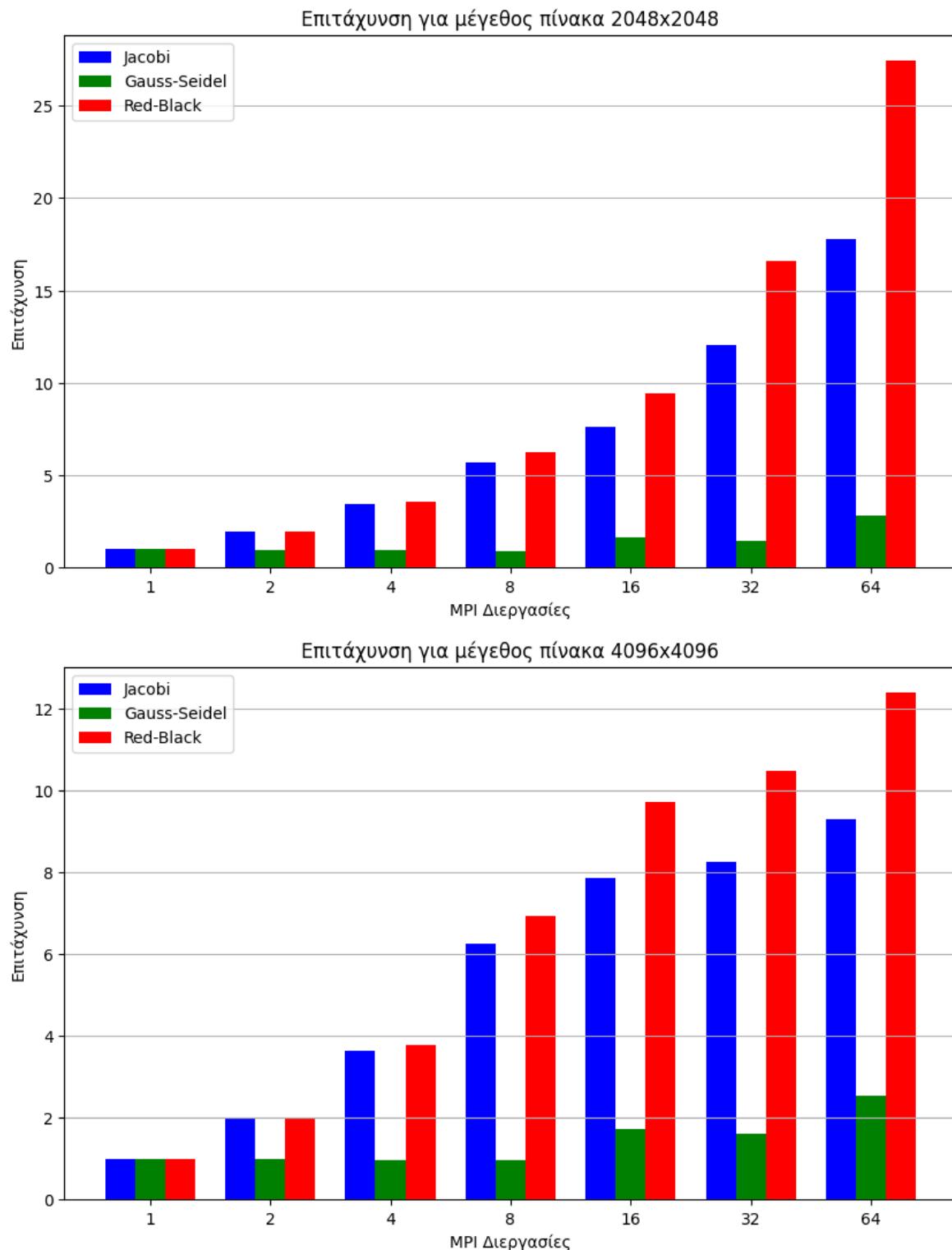
βασίζεται στη μέθοδο SOR, όπου κάθε νέο στοιχείο υπολογίζεται χρησιμοποιώντας ένα συνδυασμό της παλιάς του τιμής και του μέσου όρου των τεσσάρων γειτόνων του. Ο υπολογισμός αυτός περιλαμβάνει τον συντελεστή χαλάρωσης (omega) που ελέγχει την ταχύτητα σύγκλισης, με αποτέλεσμα η μέθοδος να συγχλίνει γρηγορότερα από τη Jacobi, υπό την προϋπόθεση σωστής επιλογής του  $\omega$ . Μετά την ολοκλήρωση της Red Phase, ακολουθεί μια νέα ανταλλαγή δεδομένων μεταξύ των διεργασιών, ώστε οι τιμές που μόλις ενημερώθηκαν να είναι διαθέσιμες στις γειτονικές υποπεριοχές πριν ξεκινήσει η Black Phase. Η διαδικασία επικοινωνίας είναι πανομοιότυπη με εκείνη που εκτελέστηκε πριν από τη Red Phase, αλλά αυτή τη φορά τα δεδομένα που αποστέλλονται περιλαμβάνουν τις ενημερωμένες τιμές της Red Phase. Η Black Phase εκτελείται στη συνέχεια, ενημερώνοντας τα σημεία του πλέγματος των οποίων το άθροισμα των συντεταγμένων  $(i+j)$  είναι περιττό. Επειδή η ενημέρωση των κόκκινων σημείων έχει ήδη ολοκληρωθεί και διαμοιραστεί, τα μαύρα σημεία μπορούν να χρησιμοποιούν τις νέες τιμές των γειτονικών τους κόκκινων σημείων χωρίς καθυστέρηση. Ο υπολογισμός εδώ είναι παρόμοιος με εκείνον της Red Phase, με τη διαφορά ότι αντί να χρησιμοποιούνται αποκλειστικά οι τιμές του `u_previous`, γίνεται και χρήση του `u_current`, που περιλαμβάνει τις πρόσφατα υπολογισμένες τιμές της Red Phase.

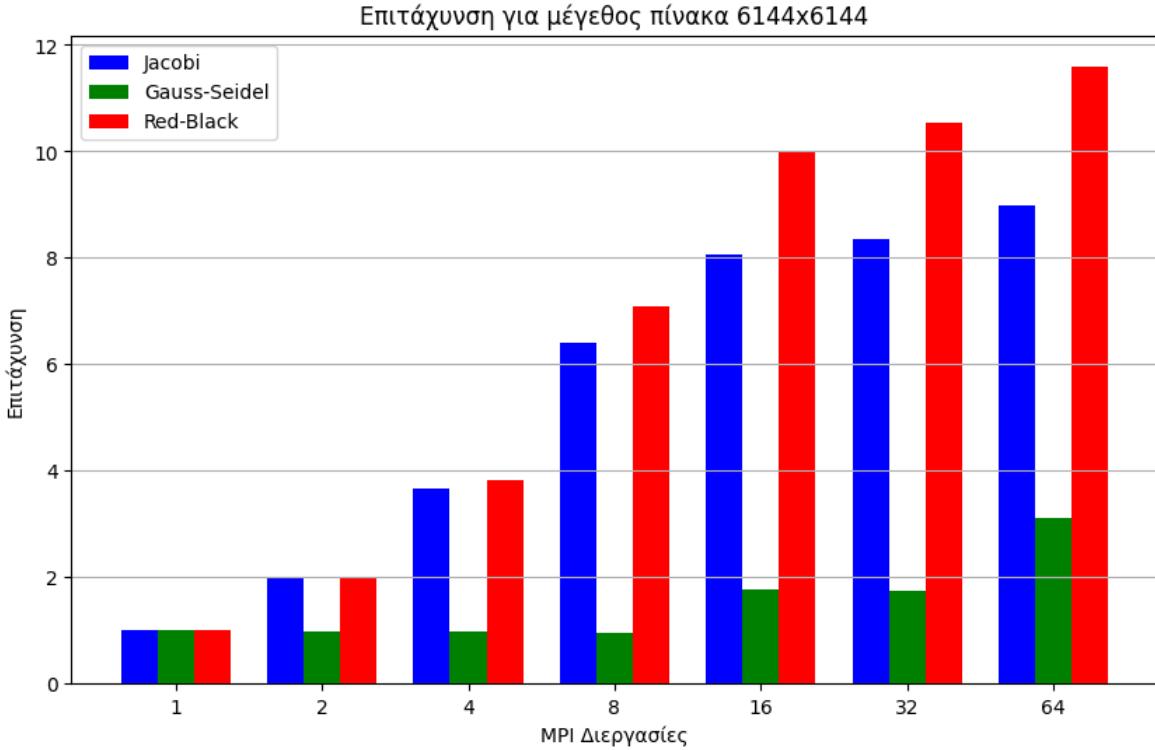
#### 4.2.4 Μετρήσεις με έλεγχο σύγκλισης



Από τα διαγράμματα παρατηρούμε ότι η μέθοδος Red-Black SOR υπερέχει σημαντικά έναντι των άλλων, τόσο σε χρόνο υπολογισμού όσο και σε συνολικό χρόνο εκτέλεσης και χρόνο σύγκλισης. Η Gauss-Seidel είναι επίσης πολύ πιο γρήγορη από τη Jacobi, αλλά παραμένει πιο αργή από την Red-Black SOR. Η Jacobi εμφανίζει τον μεγαλύτερο συνολικό χρόνο, γεγονός που την καθιστά λιγότερο αποδοτική σε ένα σύστημα κατανεμημένης μνήμης. Συνεπώς, η βέλτιστη επιλογή για την επίλυση του προβλήματος είναι η μέθοδος Red-Black SOR, καθώς εξασφαλίζει ταχύτερη σύγκλιση και μικρότερο συνολικό χρόνο εκτέλεσης.

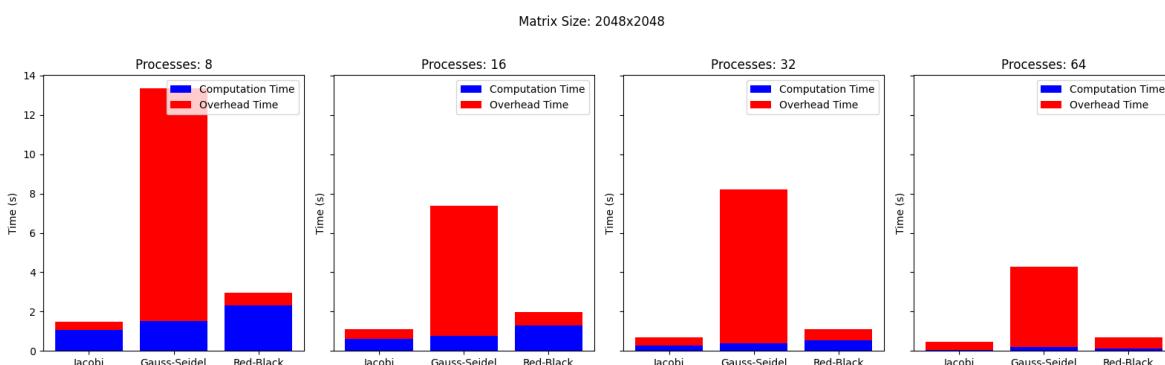
#### 4.2.5 Μετρήσεις χωρίς έλεγχο σύγκλισης

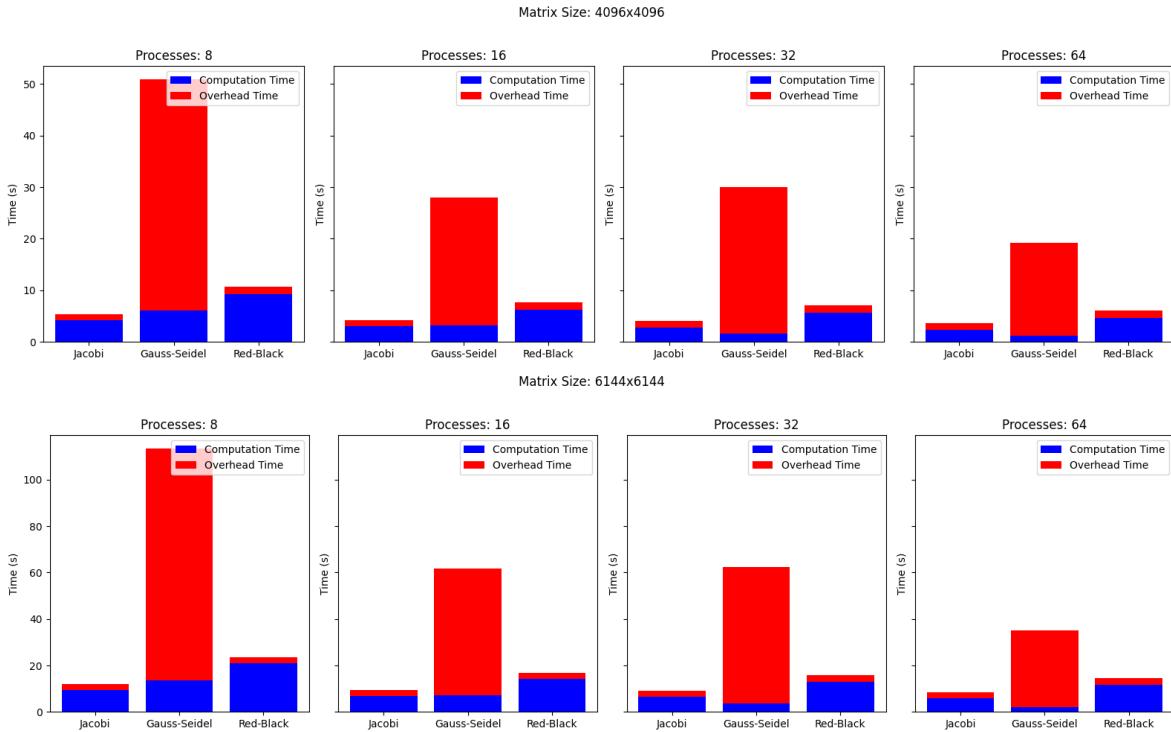




Από τα διαγράμματα επιτάχυνσης για κάθε μέγεθος πίνακα:

- Η μέθοδος Red-Black SOR εμφανίζει τον υψηλότερο βαθμό επιτάχυνσης για όλες τις τιμές MPI διεργασιών, ιδιαίτερα όταν ο αριθμός διεργασιών αυξάνεται πάνω από 8. Στα 64 cores, η μέθοδος αυτή παρουσιάζει τη μεγαλύτερη επιτάχυνση, κάτι που την καθιστά πιο κλιμακώσιμη.
- Η Jacobi έχει επίσης πολύ καλή επιτάχυνση και ακολουθεί τη Red-Black SOR, αλλά εμφανίζει μικρότερη απόδοση για πολύ υψηλό αριθμό MPI διεργασιών (64). Αυτό πιθανώς οφείλεται στο αυξημένο synchronization overhead.
- Η Gauss-Seidel παρουσιάζει τη χειρότερη επιτάχυνση, με τη βελτίωση να είναι σημαντικά μικρότερη σε σύγκριση με τις άλλες δύο μεθόδους. Η υπερβολική επικοινωνία και εξάρτηση μεταξύ των υπολογισμών κάθε διεργασίας φαίνεται να επηρεάζουν αρνητικά την απόδοση.
- Το μεγαλύτερο μέγεθος πίνακα (6144x6144) ευνοεί την καλύτερη κλιμάκωση. Αυτό φαίνεται από την πιο ομαλή αύξηση της επιτάχυνσης, ειδικά για τη Red-Black SOR και τη Jacobi. Αντίθετα, για μικρότερα μεγέθη (2048x2048), η αύξηση της επιτάχυνσης δεν είναι τόσο απότομη, κάτι που σημαίνει ότι το πρόβλημα δεν είναι αρκετά μεγάλο για να εκμεταλλευτεί πλήρως τους υπολογιστικούς πόρους.





Από τα διαγράμματα των χρόνων εκτέλεσης για τα μεγέθη 2048x2048, 4096x4096, και 6144x6144, παρατηρούνται τα εξής:

- Η μέθοδος Gauss-Seidel εμφανίζει τον μεγαλύτερο χρόνο εκτέλεσης λόγω υψηλού overhead. Αυτό είναι ιδιαίτερα εμφανές για τις περιπτώσεις με μικρότερο αριθμό διεργασιών, όπου το κόκκινο τμήμα (overhead time) κυριαρχεί στα διαγράμματα μπάρας. Καθώς αυξάνονται οι MPI διεργασίες, το overhead μειώνεται, αλλά παραμένει σημαντικότερο σε σχέση με τις άλλες δύο μεθόδους.
- Οι μέθοδοι Jacobi και Red-Black SOR παρουσιάζουν χαμηλότερους χρόνους εκτέλεσης σε σύγκριση με τη Gauss-Seidel. Η Red-Black SOR φαίνεται να έχει μικρότερο overhead από τη Gauss-Seidel, αλλά μεγαλύτερο από τη Jacobi.
- Η Jacobi είναι η πιο ισορροπημένη μέθοδος όσον αφορά τον υπολογιστικό χρόνο και το overhead, διατηρώντας σταθερά χαμηλό το κόκκινο τμήμα στα διαγράμματα, πράγμα που σημαίνει ότι η επικοινωνία και το synchronization έχουν μικρότερη επίδραση.
- Ο χρόνος εκτέλεσης μειώνεται καθώς αυξάνονται οι διεργασίες, αλλά η μείωση αυτή δεν είναι γραμμική για όλες τις μεθόδους. Η απόδοση της Gauss-Seidel βελτιώνεται, αλλά εξακολουθεί να υστερεί λόγω overhead.