



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

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

Ομάδα 25

Γεώργιος Γιαννακόπουλος

AM: 03120828

Όλγα Κοντακιώτη

AM: 03120058

Αποστολία Χρυσοβαλάντου Σκέντζου

AM: 03120054

1^η Εργαστηριακή Άσκηση

Εξοικείωση με το περιβάλλον προγραμματισμού

Εισαγωγή

Στην παρούσα εργαστηριακή άσκηση, θα εξετάσουμε το πρόβλημα του Conway's Game of Life, ένα κλασικό παράδειγμα κυψελωτού αυτομάτου, όπου κάθε κελί ενός πίνακα $n \times n$ μπορεί να βρίσκεται σε δύο καταστάσεις: «ζωντανό» ή «νεκρό». Η κατάσταση κάθε κελιού στην επόμενη χρονική στιγμή καθορίζεται από την κατάσταση των 8 γειτονικών του κελιών (βλ.

1	2	3			
4		5			
6	7	8			

διπλανό σχήμα), σύμφωνα με προκαθορισμένους κανόνες. Το συγκεκριμένο πρόβλημα είναι ιδανικό για την επίδειξη παραλληλοποίησης, καθώς η κατάσταση κάθε κελιού μπορεί να υπολογιστεί ανεξάρτητα, επιτρέποντας τη χρήση πολλαπλών threads για την επιτάχυνση της εκτέλεσης.

Στόχος μας είναι να μελετήσουμε την επίδραση της παραλληλοποίησης στην απόδοση του προγράμματος, εξετάζοντας τον χρόνο εκτέλεσης με διαφορετικό αριθμό threads και διαφορετικά μεγέθη πίνακα. Συγκεκριμένα, θα συγκρίνουμε την απόδοση του παραλληλοποιημένου προγράμματος με τον θεωρητικά ιδανικό

χρόνο εκτέλεσης, ο οποίος προκύπτει από τη χρήση P threads. Παράλληλα, θα εξετάσουμε την επίδραση του μεγέθους του προβλήματος στις δυνατότητες βελτιστοποίησης μέσω παραλληλοποίησης.

Συγκεκριμένα θα χρησιμοποιήσουμε την παρακάτω παραμετροποίηση:

- Μέγεθος πίνακα προβλήματος: **64 x 64, 1024 x 1024, 4096 x 4096**
- Αριθμός νημάτων (threads): **1, 2, 3, 4, 5, 6, 7, 8**

Για κάθε τέτοια παραμετροποίηση θα χρησιμοποιήσουμε **αριθμό χρονικών βημάτων ίσο με 1000**.

Παρατήρηση:

Μας ζητήθηκε να παρατηρήσουμε τα αποτελέσματα για αριθμούς threads: 1, 2, 4, 6, 8, αλλά θελήσαμε να δοκιμάσουμε λίγο περισσότερες παραμετροποιήσεις για την διεξαγωγή των συμπερασμάτων μας.

Μετρικές για την διεξαγωγή συμπερασμάτων:

Για κάθε παραμετροποίηση που θα εξετάσουμε θα παίρνουμε ως ουτρυτ τον χρόνο εκτέλεσης του βασικού block υπολογισμού του προβλήματος, δηλαδή την περιήγηση για όλες τις χρονικές στιγμές πάνω στον πίνακα.

Θα συγκρίνουμε τον χρόνο εκτέλεσης αυτής της παραλληλοποίησης με τον ιδανικό χρόνο εκτέλεσης ενός παραλληλοποιημένου προγράμματος που χρησιμοποιεί P πυρήνες για την εκτέλεση του:

$$T_{p_{ideal}} = \frac{T_{serial}}{P}$$

Παραλληλοποίηση προγράμματος:

Για να καταφέρουμε να παραλληλοποιήσουμε το πρόγραμμα που μας δόθηκε, θα χρησιμοποιήσουμε την εντολή **#pragma omp parallel for**, εντός του δοσμένου προγράμματος Game_Of_Life.c, όπως φαίνεται παρακάτω:

```

for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for private(j, nbrs) // Parallelize the row loop
    for ( i = 1 ; i < N-1 ; i++ )
        for ( j = 1 ; j < N-1 ; j++ ) {
            nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
                + previous[i][j-1] + previous[i][j+1] \
                + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
            if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                current[i][j]=1;
            else
                current[i][j]=0;
        }
}

```

Η παραλληλοποίηση έγινε ως προς τις γραμμές, αφού παρατηρούμε πως τα κελιά αυτά μπορούν να υπολογίσουν την νέα κατάστασή τους ανεξάρτητα από το αποτέλεσμα της παραπάνω ή παρακάτω γραμμής. Αντίστοιχα, θα μπορούσαμε να έχουμε επιλέξει και τις στήλες (σημειώνεται, όμως, ότι δεν έχει νόημα να παραλληλοποιήσουμε και τις δύο διαστάσεις ταυτόχρονα).

Χρησιμοποιούμε τις μεταβλητές `j` και `nbrs` ως `private` για κάθε `thread`, εξασφαλίζοντας ότι κάθε `thread` έχει το δικό του αντίγραφο αυτών των μεταβλητών και αποφεύγει έτσι `race conditions`. Αυτό αποκλείει την περίπτωση δύο ή περισσότερα `threads` να προσπελάσουν ταυτόχρονα την ίδια θέση μνήμης, οδηγώντας σε απρόβλεπτη συμπεριφορά κατά την ανάγνωση ή τροποποίηση των τιμών. Με αυτόν τον τρόπο, διασφαλίζεται ότι κάθε `thread` μπορεί να διαχειρίζεται ανεξάρτητα τη σάρωση των στηλών.

Outputs:

Τα outputs που πήραμε παρουσιασμένα σαν χρόνοι εκτέλεσης (σε sec):

#νημάτων/μέγεθος πίνακα	64 x 64	1024 x 1024	4096 x 4096
serial	0.020366	10.790454	173.278309
1	0.023118	10.968904	175.880846
2	0.013572	5.460309	88.251924
3	0.011263	3.625454	59.159467
4	0.010126	2.722881	45.907959
5	0.009314	2.190667	46.522953
6	0.009595	1.829894	43.540697
7	0.009430	1.569943	43.124358
8	0.009467	1.378128	42.996053

Παρατήρηση:

Για τα παρακάτω διαγράμματα θα χρειαστεί να υπολογίσουμε τον ιδανικό χρόνο παραλληλοποίησης που όπως προαναφέραμε είναι $T_{p_{ideal}} = T_{serial} / P$.

Για τον σειριακό χρόνο εκτέλεσης θα μπορούσαμε να χρησιμοποιήσουμε είτε τα αποτελέσματα που θα παίρναμε από την σειριακή εκτέλεση του προγράμματος αφαιρώντας την εντολή παραλληλοποίησης `#pragma omp parallel for` (τα αποτελέσματα αυτά φαίνονται στην πρώτη γραμμή του παραπάνω πίνακα), είτε να μην αφαιρέσουμε την εντολή παραλληλοποίησης και να τρέξουμε για αριθμό νημάτων $p = 1$, που στην ουσία υπονοεί σειριακή εκτέλεση (τα αποτελέσματα φαίνονται στην δεύτερη γραμμή του παραπάνω πίνακα).

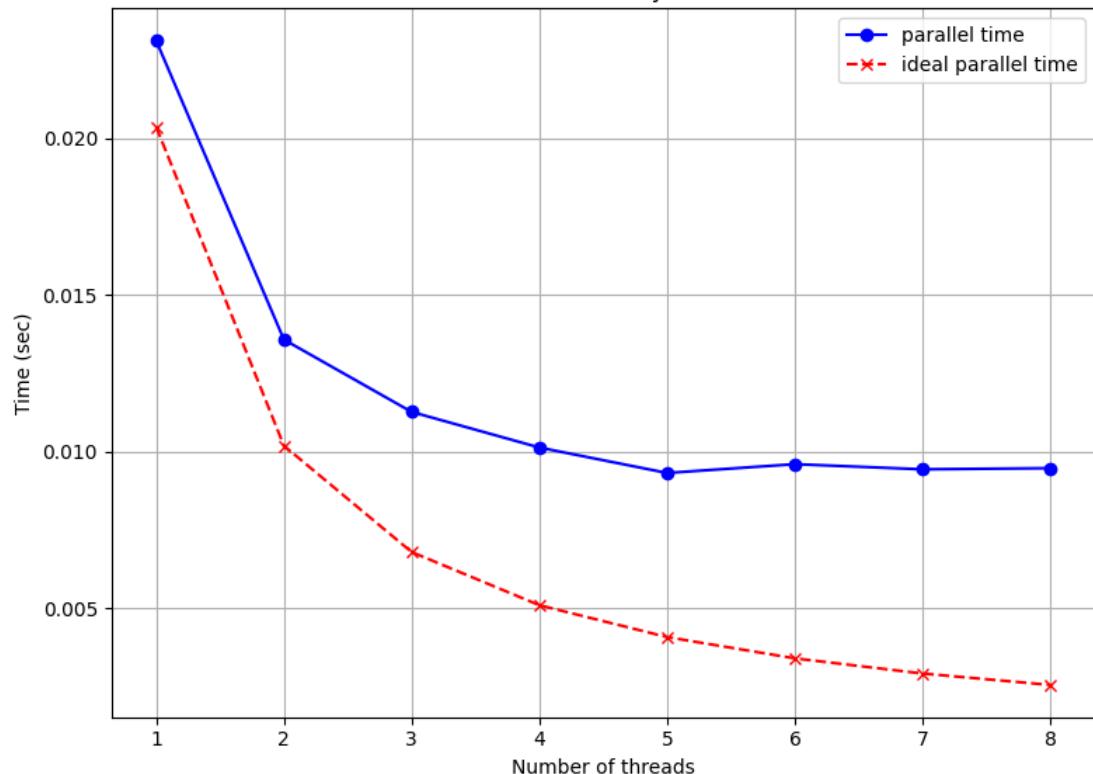
Όταν χρησιμοποιείται η εντολή `#pragma omp parallel for` σε ένα πρόγραμμα, ακόμη και αν ο αριθμός των νημάτων είναι $p = 1$, υπάρχει ένα μικρό υπολογιστικό κόστος (*overhead*) λόγω της υποδομής που απαιτείται για την παραλληλοποίηση. Αυτό ενδεχομένως περιλαμβάνει τη δημιουργία δομών δεδομένων για την υποστήριξη παράλληλων εκτελέσεων, τον σχεδιασμό της κατανομής των εργασιών στα νήματα, καθώς και τον ενδεχόμενο συγχρονισμό τους. Παρά το γεγονός ότι χρησιμοποιείται μόνο ένα νήμα, το σύστημα εκτέλεσης προετοιμάζει τον κώδικα με τρόπο που να είναι συμβατός με πολλαπλές διεργασίες, επιβαρύνοντας την εκτέλεση με αυτό το *overhead*.

Επομένως, στην περίπτωση που η εντολή απουσιάζει και έχουμε κατεξοχήν σειριακή εκτέλεση δεν θα επιβαρυνθεί η εκτέλεση από τα πρόσθετα βήματα και θα πάρουμε γρηγορότερο σειριακό πρόβλημα γεγονός που επιβεβαιώνεται και από τις μετρήσεις μας.

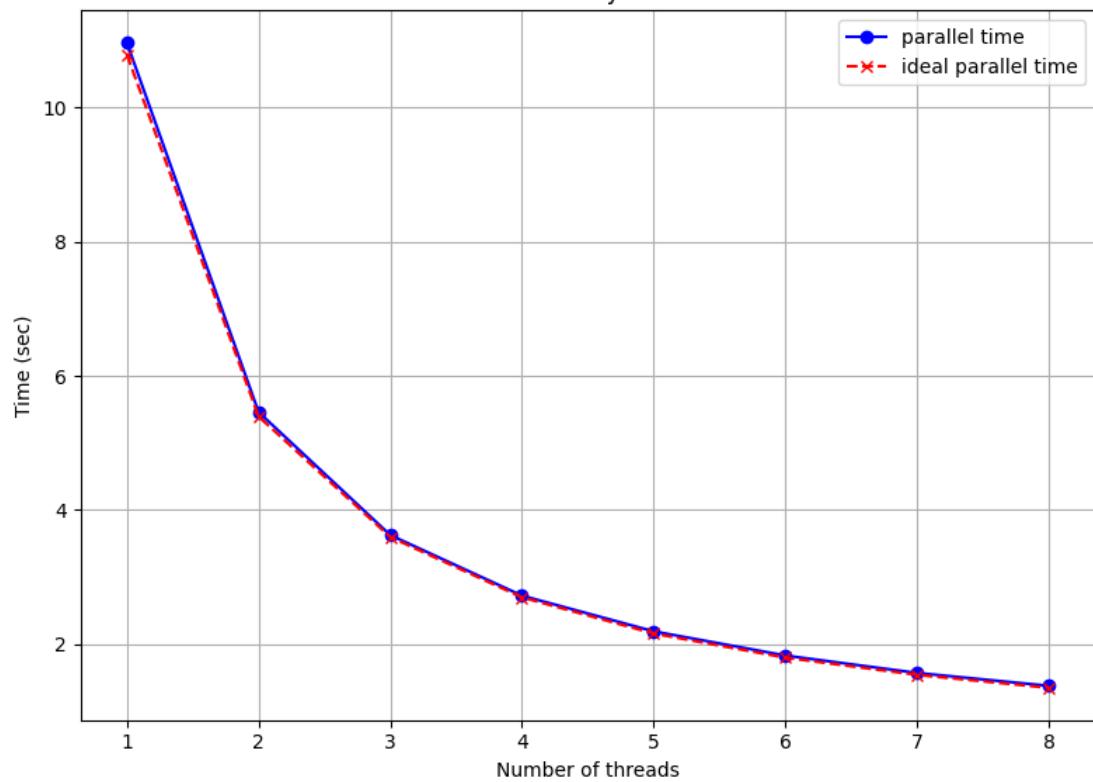
Για τους παραπάνω λόγους θεωρούμε σωστότερη την προσέγγιση για τους χρόνους ενός σειριακού προγράμματος να χρησιμοποιήσουμε τα αποτελέσματα της πρώτης γραμμής, που αντιστοιχεί στην εκτέλεση δίχως την χρήση της εντολής `#pragma omp parallel for`.

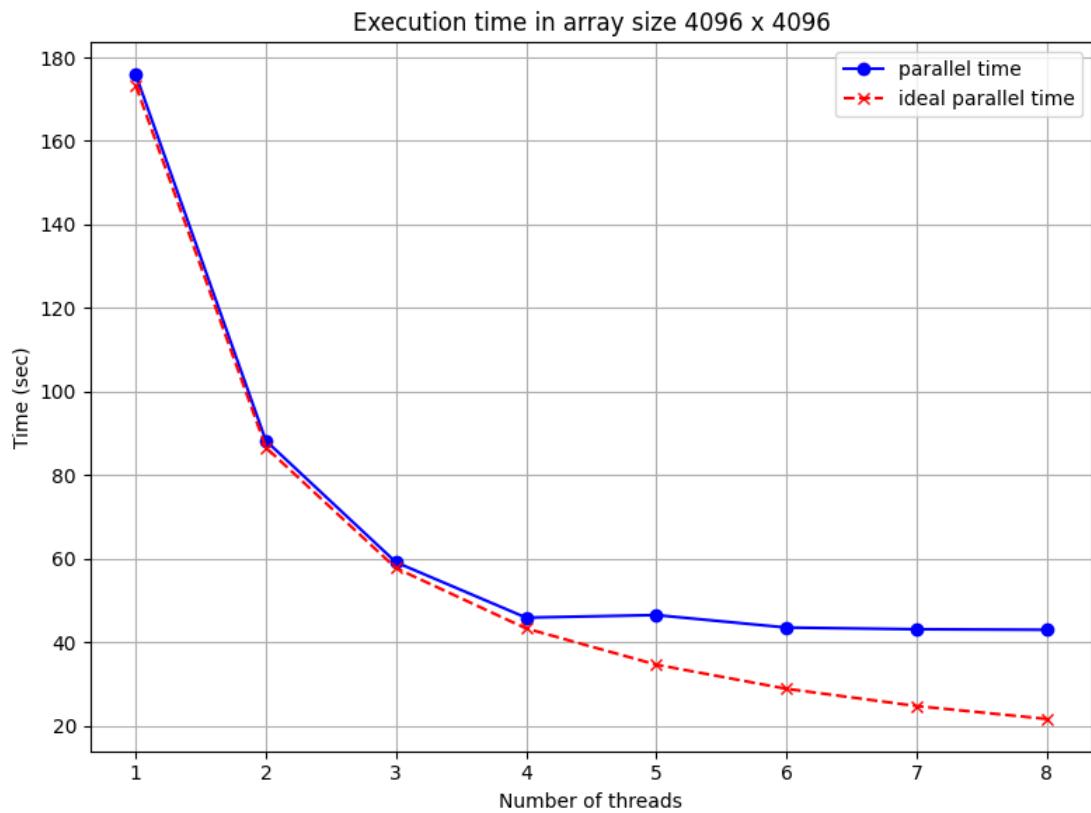
Παρουσιάζουμε τώρα τα αποτελέσματα και σε γραφική απεικόνιση, βλέποντας για κάθε μέγεθος πίνακα την καμπύλη της ιδανικής γραμμικής παραλληλοποίησης και της πραγματικής που πήραμε σύμφωνα με τις παραπάνω τιμές:

Execution time in array size 64 x 64

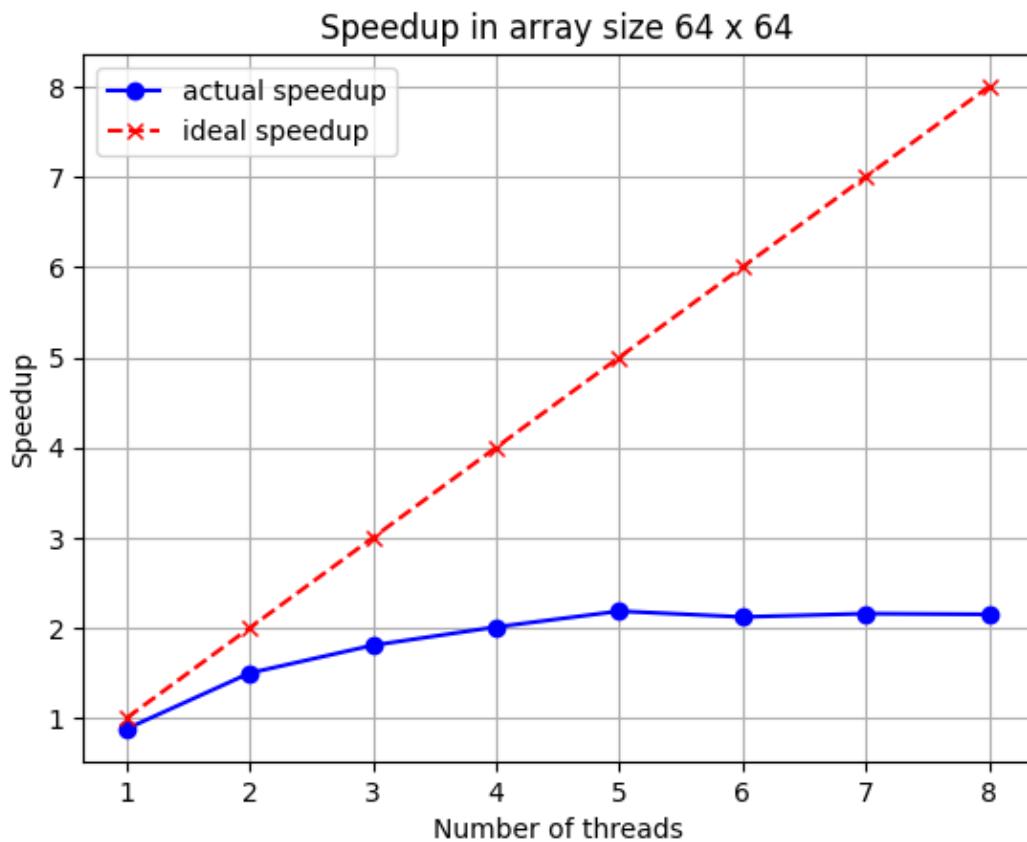


Execution time in array size 1024 x 1024

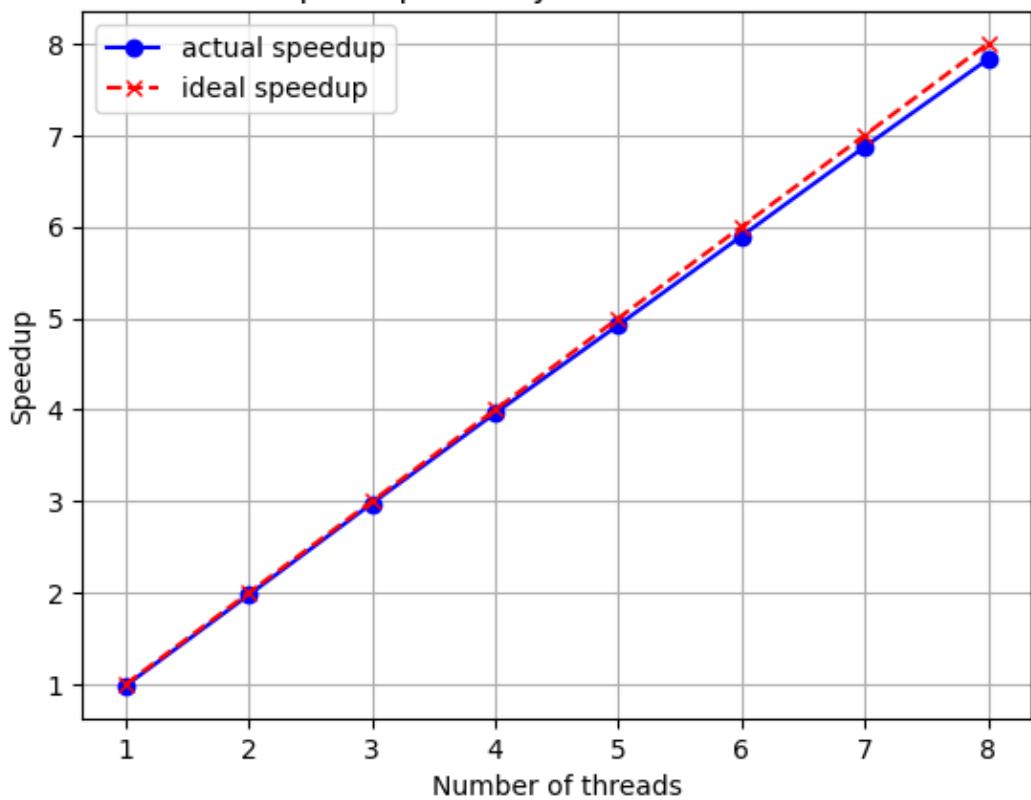




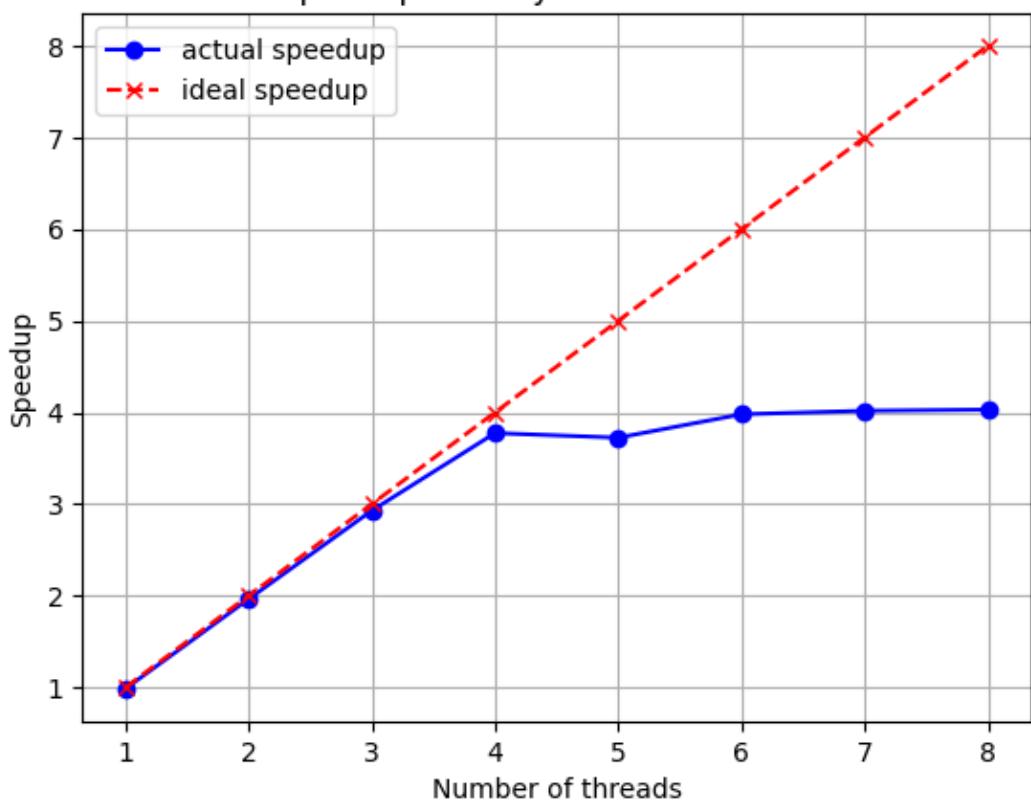
Ακολουθούν τα διαγράμματα που απεικονίζουν την επιτάχυνση (speedup):



Speedup in array size 1024×1024



Speedup in array size 4096×4096



Σχολιασμός και συμπεράσματα:

Παρατηρούμε ότι για μικρά grids (π.χ. 64×64), η αύξηση του αριθμού των threads μειώνει τον χρόνο εκτέλεσης έως ένα συγκεκριμένο σημείο, από το οποίο και έπειτα ο χρόνος παραμένει σχεδόν σταθερός. Η συνολική απόδοση απέχει από την ιδανική παραλληλοποίηση. Αυτό οφείλεται κυρίως στο κόστος δημιουργίας και συγχρονισμού των threads. Ειδικότερα, το κόστος για τη δημιουργία των threads είναι συγκρίσιμο με το έργο που εκτελεί ο κώδικας μας, επηρεαζόντας την απόδοσή του, ενώ ο συγχρονισμός των νημάτων, ο οποίος εξαρτάται από τον αριθμό τους, προσθέτει επιπλέον κόστος. Σε μεγαλύτερα grids (π.χ. 4096×4096), παρατηρούμε ότι η απόδοση είναι σχεδόν ιδανική με μέχρι 4 threads, καθώς το speedup αυξάνεται αναλογικά. Όμως, μετά από αυτό το σημείο, η περαιτέρω αύξηση των threads δεν οδηγεί σε αντίστοιχη μείωση του χρόνου. Αυτή η υποβάθμιση («σπάσιμο») της παραλληλοποίησης οφείλεται στο γεγονός ότι τα δεδομένα δεν χωρούν όλα στην cache, γεγονός που προκαλεί συμφόρηση στο δίστημα μνήμης. Αντίθετα, για ενδιάμεσα μεγέθη grid (π.χ. 1024×1024), η παραλληλοποίηση είναι πολύ πιο αποδοτική, προσεγγίζοντας την ιδανική απόδοση χωρίς να εμφανίζονται τα παραπάνω προβλήματα.

Συνοψίζοντας, παρατηρούμε ότι η παραλληλοποίηση είναι αποδοτική και επιφέρει σημαντικές βελτιώσεις στον χρόνο εκτέλεσης, ιδίως για μεσαία και μεγάλα μεγέθη προβλημάτων. Ωστόσο, υπάρχουν σημεία όπου το κόστος των threads και η χρήση της μνήμης περιορίζουν τις δυνατότητες περαιτέρω βελτιστοποίησης. Η βελτιστοποίηση της απόδοσης θα μπορούσε να ενισχυθεί περαιτέρω με τη χρήση τεχνικών για την αποφυγή bottlenecks μνήμης ή με την εκμετάλλευση πιο προχωρημένων μοντέλων παραλληλοποίησης.

Κώδικας που υλοποιήθηκε:

- **make_on_queue.sh**: Το πρόγραμμα για την μεταγλώττιση του Game_Of_Life.c που καλεί την MakeFile

```
parlab25@scirouter:~/ex1$ cat make_on_queue.sh
#!/bin/bash

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

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

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

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

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

module load openmp
cd /home/parallel/parlab25/ex1/
make
```

- **Makefile**

```
parlab25@scirouter:~/ex1$ cat Makefile
all: omp_Game_Of_Life

omp_Game_Of_Life: Game_Of_Life.c
    gcc -O3 -fopenmp -o omp_Game_Of_Life Game_Of_Life.c

clean:
    rm omp_Game_Of_Life
```

- **run_on_queue.sh**: Script για να τρέχουμε το πρόγραμμα Game_Of_Life.c για τις παραμετροποιήσεις που θέλουμε

```
parlab25@scirouter:~/ex1$ cat run_on_queue.sh
## give the Job a descriptive name
#PBS -N run_omp_Game_Of_Life

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

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

## How long should the job run for?
#PBS -l walltime=00:40:00

## Start
## Load OpenMP module and move to the working directory
module load openmp
cd /home/parallel/parlab25/ex1/

## Create a new directory for the output files
OUTPUT_DIR="game_of_life_output_serial"
mkdir -p $OUTPUT_DIR

time ./omp_Game_Of_Life 4096 1000 > $OUTPUT_DIR/run_omp_Game_Of_Life_ser4096.out 2> $OUTPUT_
DIR/run_omp_Game_Of_Life_ser4096.err

## Loop over thread counts
##for i in {1..8}
##do
##    export OMP_NUM_THREADS=$i
##    echo "Running with OMP_NUM_THREADS=$i"

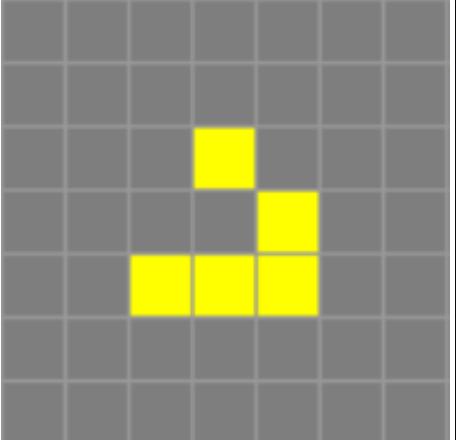
##    Redirect output and error for each thread count into the new directory
##    ##time ./omp_Game_Of_Life 4096 1000 > $OUTPUT_DIR/run_omp_Game_Of_Life_4096_$i.out 2
##    > $OUTPUT_DIR/run_omp_Game_Of_Life_4096_$i.err
##done
```

Bonus:

Για την παραγωγή οπτικών αποτελεσμάτων, προσθέσαμε το flag -DOUTPUT στη Makefile και τροποποιήσαμε τη συνάρτηση αρχικοποίησης init_random στο αρχείο Game_Of_Life.c.

Επιλέξαμε αρχικά την αρχικοποίηση τύπου “glider”, η οποία δημιουργεί ένα pattern που κινείται διαγώνια στο grid.

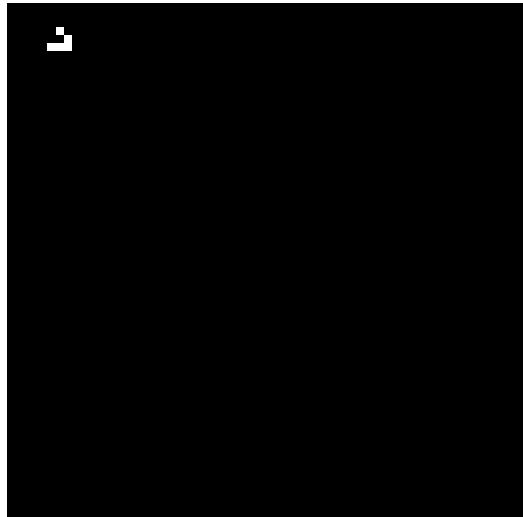
Αρχικοποίηση:



```
// Glider
/* void init_random(int **array1, int **array2, int N) {
    int x = 5, y = 5;
    array1[x][y] = 1;
    array1[x][y+1] = 1;
    array1[x-2][y+1] = 1;
    array1[x][y+2] = 1;
    array1[x-1][y+2] = 1;

    // Copy the same pattern into the second array (array2)
    array2[x][y] = 1;
    array2[x][y+1] = 1;
    array2[x-2][y+1] = 1;
    array2[x][y+2] = 1;
    array2[x-1][y+2] = 1;
} */
```

Αποτέλεσμα:

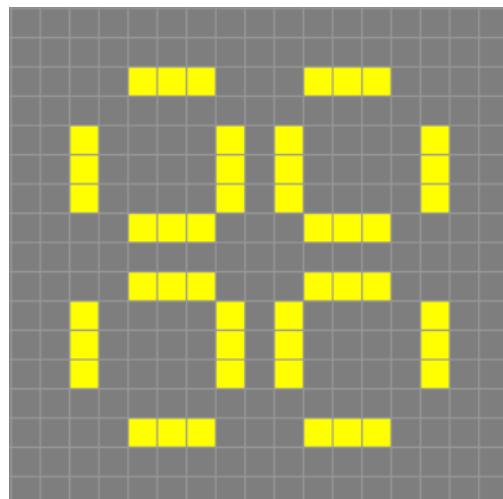


Το βίντεο είναι διαθέσιμο στον παρακάτω σύνδεσμο:

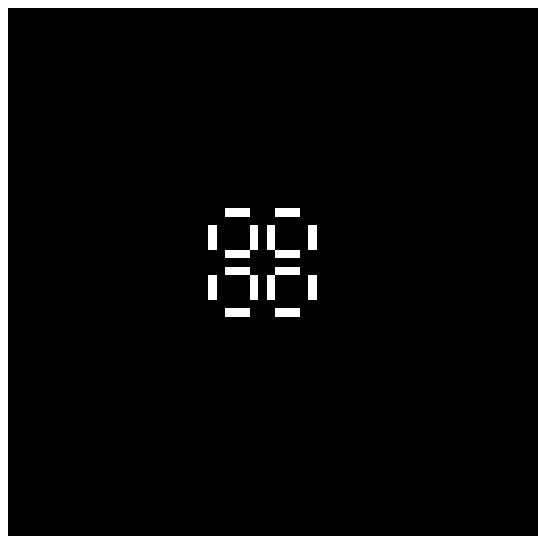
[https://drive.google.com/file/d/1rtpepduaY-
PCyZ4g3IYAQToGsvPteHln/view?usp=sharing](https://drive.google.com/file/d/1rtpepduaY-PCyZ4g3IYAQToGsvPteHln/view?usp=sharing)

Στη συνέχεια, δοκιμάσαμε την αρχικοποίηση τύπου “pulsar”, η οποία δημιουργεί ένα pattern που λειτουργεί ως ταλαντωτής με περίοδο 3. Δηλαδή, το σχήμα του επαναλαμβάνεται κάθε 3 time steps, ενώ παραμένει σταθερό μεταξύ των επαναλήψεων.

Αρχικοποίηση:



Αποτέλεσμα:



To βίντεο είναι διαθέσιμο στον παρακάτω σύνδεσμο:

https://drive.google.com/file/d/1JMEqifxiyy5YjBHjzRV9DEWWo_W3OkkP/view?usp=sharing

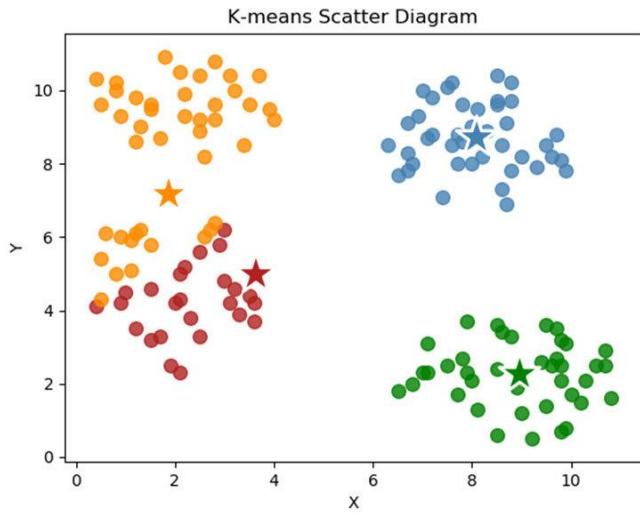
Σημειώνεται ότι για όλες τις περιπτώσεις χρησιμοποιήσαμε 8 threads, ενώ επιλέξαμε grid size 64 και 200 time steps.

2^η Εργαστηριακή Άσκηση

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Ο αλγόριθμος *k-means* είναι ένας διαδεδομένος αλγορίθμικός τρόπος ομαδοποίησης δεδομένων (clustering), που χρησιμοποιείται ευρέως σε επιστημονικές εφαρμογές, όπως η μηχανική μάθηση και η ανάλυση μεγάλων δεδομένων. Στόχος του είναι να διαχωρίσει ένα σύνολο δεδομένων σε k ομάδες (clusters), προσδιορίζοντας κεντρικά σημεία (centroids) για κάθε ομάδα. Ο αλγόριθμος λειτουργεί επαναληπτικά: αρχικά ορίζει τυχαία τα k centroids και κατόπιν εκχωρεί κάθε σημείο δεδομένων στο πλησιέστερο centroid. Σε κάθε επανάληψη, ενημερώνονται τα centroids σύμφωνα με τις νέες ομάδες δεδομένων, μέχρις ότου η διαδικασία συγκλίνει.



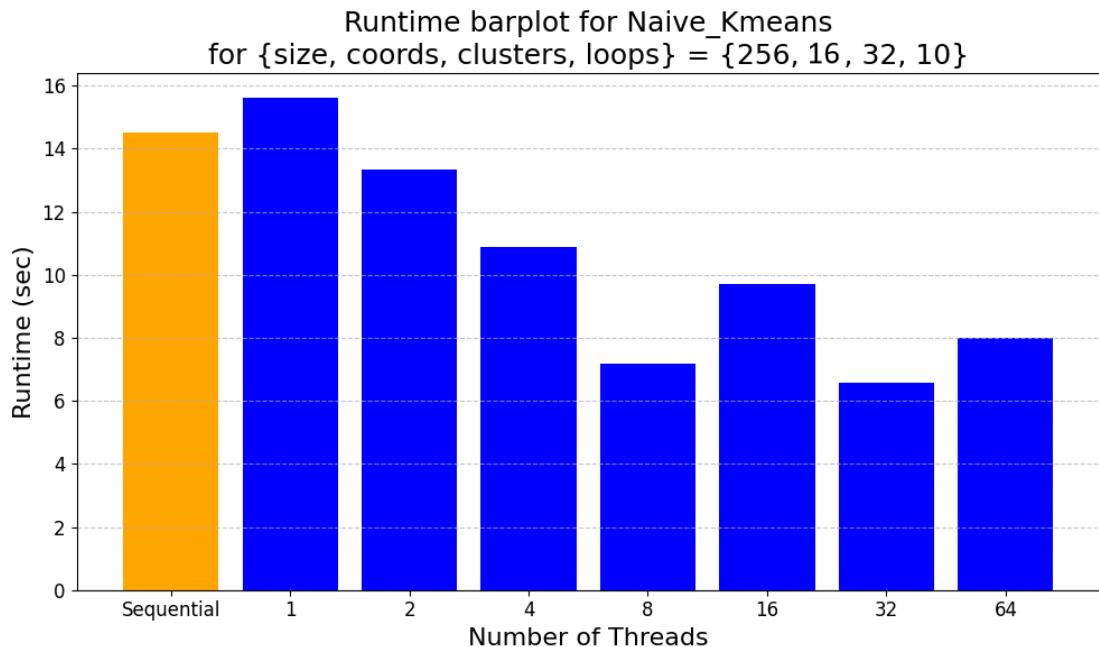
Στη συνέχεια, θα μελετήσουμε την παραλληλοποίηση του αλγορίθμου με δύο διαφορετικές προσεγγίσεις. Στην πρώτη προσέγγιση, οι πίνακες newClusters και newClusterSize, που κρατούν πληροφορίες για τις σχηματιζόμενες ομάδες, θα είναι κοινόχρηστοι μεταξύ των threads, διασφαλίζοντας έτσι την πρόσβαση στα κοινά δεδομένα. Στη δεύτερη προσέγγιση, κάθε thread θα διαθέτει ένα τοπικό αντίγραφο αυτών των δεδομένων και θα δουλεύει ανεξάρτητα, ενώ στο τέλος τα τοπικά αποτελέσματα θα συνδυάζονται μέσω της διαδικασίας reduction.

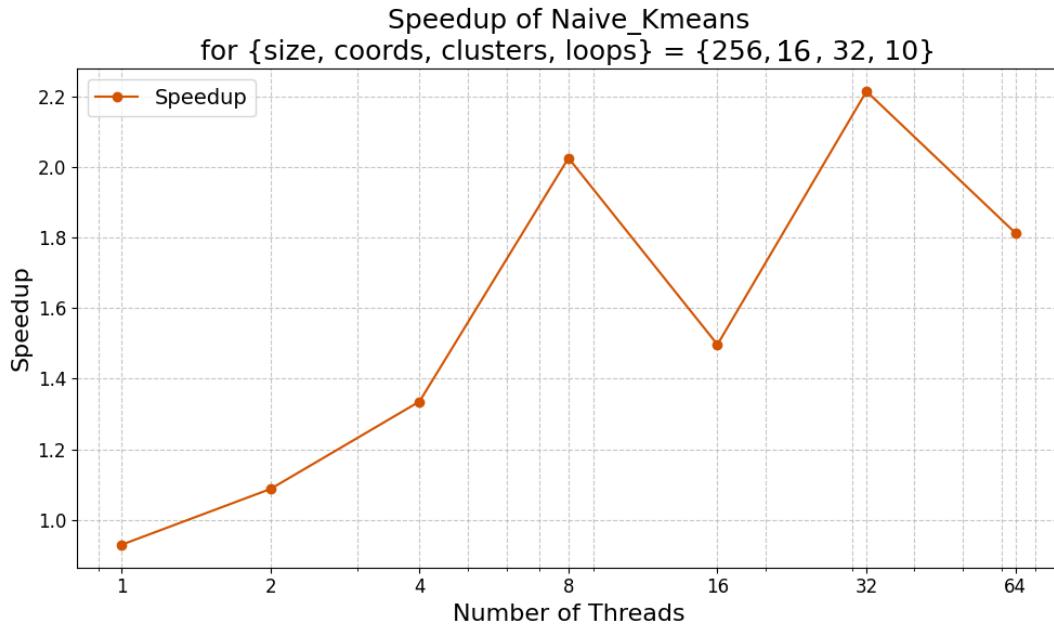
1^η υλοποίηση: Shared Clusters

Όπως αναφέρθηκε, σε αυτήν την υλοποίηση τα κρίσιμα δεδομένα είναι κοινόχρηστα (*shared*) μεταξύ των threads. Ο παραλληλισμός έγκειται στον υπολογισμό της ευκλείδειας απόστασης μεταξύ κάθε σημείου και των τρεχόντων κέντρων των clusters, για τον οποίο χρησιμοποιούμε την εντολή **#pragma omp parallel for**. Προκειμένου να αποφύγουμε το *race condition*, εξασφαλίζουμε την ατομική ενημέρωση των κοινών πινάκων *newClusters* και *newClusterSize* μέσω της εντολής **#pragma omp atomic**. Η διαχείριση της μεταβλητής *delta*, που καθορίζει τη σύγκλιση του αλγορίθμου, γίνεται αντιγράφοντας τη μεταβλητή σε κάθε thread και συνδυάζοντας τα αποτελέσματα σε παγκόσμια τιμή με την εντολή *reduction* του OpenMP.

Στη συνέχεια, θα παρουσιάσουμε τις μετρήσεις εκτέλεσης της παραπάνω παραλληλοποιημένης εκδοχής του αλγορίθμου και θα αναλύσουμε τα σχετικά συμπεράσματά μας. Το configuration που θα χρησιμοποιηθεί για τις μετρήσεις έχει τις παραμέτρους **{size, coords, clusters, loops} = {256, 16, 32, 10}** και θα δοκιμαστεί με διαφορετικό αριθμό *threads* **{1, 2, 4, 8, 16, 32, 64}**.

Παρουσιάζουμε τα διαγράμματα για το runtime και το speedup:





Σχολιασμός και συμπεράσματα:

Στη συγκεκριμένη υλοποίηση, παρατηρούμε ότι ο παραλληλισμός δεν βελτιώνει σημαντικά τις επιδόσεις. Παρά τη μείωση του χρόνου εκτέλεσης σε σχέση με το σειριακό πρόγραμμα, το speedup παραμένει χαμηλό, κάτω από 2.5, ακόμη και με την αύξηση του αριθμού των threads. Ενώ αρχικά καταγράφεται κάποια κλιμάκωση μέχρι τα 8 threads, μετά από αυτό το σημείο η απόδοση παρουσιάζει μη ομοιόμορφη συμπεριφορά και κακή κλιμάκωση.

Για τη διασφάλιση της ορθότητας των αποτελεσμάτων του αλγορίθμου, χρησιμοποιήσαμε την εντολή `#pragma omp atomic` ώστε να επιτευχθεί σειριακή εγγραφή στους κοινόχρηστους πίνακες από κάθε νήμα. Ωστόσο, η σειριοποίηση αυτή, που είναι απαραίτητη λόγω των πολλαπλών προσβάσεων στους πίνακες από κάθε νήμα, περιορίζει σημαντικά την απόδοση του παράλληλου προγράμματος, καθώς δημιουργεί ένα σημείο συμφόρησης. Η επίδραση αυτής της συμφόρησης είναι ιδιαίτερα εμφανής για μεγαλύτερο αριθμό threads όπως είναι εμφανές στο παραπάνω διάγραμμα.

Τέλος, η απόδοση επιβαρύνεται από τα *migrations* που επιβάλλονται στα threads από το λειτουργικό σύστημα. Συγκεκριμένα, το λειτουργικό σύστημα στο *content switch* μπορεί να επιβάλλει στα threads να τρέξουν σε διαφορετικούς φυσικούς πυρήνες από αυτούς που αρχικά έτρεχαν. Κατά τη διάρκεια αυτών των αλλαγών, τα threads χάνουν την τοπική τους *cache*, καθώς μετακινούνται σε άλλους πυρήνες. Ως αποτέλεσμα, συχνές προσβάσεις σε πιο απομακρυσμένη μνήμη

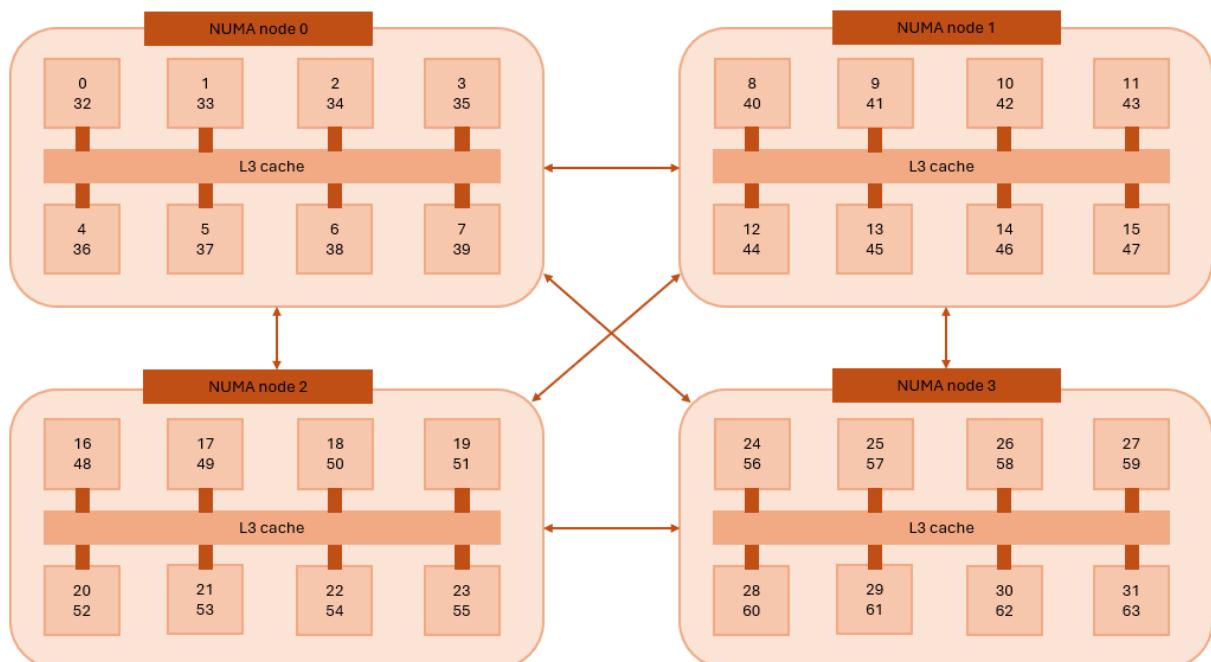
γίνονται αναπόφευκτες, αυξάνοντας την κίνηση στον δίσκο μνήμης και δημιουργώντας επιπρόσθετες καθυστερήσεις.

Για να λύσουμε το τελευταίο πρόβλημα του *migration* μπορούμε να χρησιμοποιήσουμε την μεταβλητή περιβάλλοντος **GOMP_CPU_AFFINITY**.

Η μεταβλητή περιβάλλοντος **GOMP_CPU_AFFINITY**

Η μεταβλητή περιβάλλοντος **GOMP_CPU_AFFINITY** της βιβλιοθήκης **GNU OpenMP** (GOMP) επιτρέπει τον έλεγχο της τοποθέτησης των threads σε συγκεκριμένους πυρήνες επεξεργαστή (CPU affinity). Ορίζοντας την GOMP_CPU_AFFINITY, μπορούμε να καθορίσουμε σε ποιους πυρήνες θα εκτελούνται τα threads, βελτιστοποιώντας τη χρήση της τοπικής μνήμης cache και μειώνοντας την ανάγκη για μετακινήσεις μεταξύ των πυρήνων. Αυτή η δυνατότητα είναι ιδιαίτερα χρήσιμη για τη βελτίωση της απόδοσης σε πολυπύρηνα συστήματα, καθώς μειώνει το overhead που προκαλείται από τον συγχρονισμό και τη μεταφορά δεδομένων, εξασφαλίζοντας καλύτερη αξιοποίηση των πόρων του επεξεργαστή.

Με την εντολή **lscpu** είδαμε την αντιστοίχιση των λογικών πυρήνων με τους φυσικούς, καθώς το σύστημα έχει 32 physical cores και υποστηρίζει hyperthreading επιπέδου 2, άρα έχουμε 64 logical cores. Προκειμένου να είναι σαφής η παραμετροποίηση που θα παρουσιάσουμε αργότερα για την μεταβλητή περιβάλλοντος, παρουσιάζουμε σχηματικά τις αντιστοιχίσεις των ονομάτων των threads πάνω στο σύστημά μας:



Θα δοκιμάσουμε να «δέσουμε» τα threads στους επεξεργαστές με διαφορετικές προσεγγίσεις, αξιοποιώντας ή αγνοώντας το hyperthreading, ώστε να παρατηρήσουμε την επίδρασή του στην απόδοση. Στην πρώτη προσέγγιση, κάθε νέο thread θα δεσμεύεται στο ίδιο φυσικό πυρήνα (physical core), όπου αυτό είναι δυνατόν, αξιοποιώντας πλήρως το hyperthreading. Στη δεύτερη προσέγγιση, κάθε thread θα ανατίθεται σε διαφορετικό φυσικό πυρήνα, αποφεύγοντας το hyperthreading. Στην τρίτη προσέγγιση, θα εκτελέσουμε τον αλγόριθμο με το βέλτιστο configuration που προέκυψε από τις δύο προηγούμενες δοκιμές και θα μετρήσουμε τους αντίστοιχους χρόνους εκτέλεσης.

Συγκεκριμένα, οι εντολές περιβάλλοντος που θα χρησιμοποιήσουμε για κάθε εκτέλεση θα είναι οι παρακάτω:

1^η εκτέλεση (Avoid hyperthreading)

```
## Loop over the thread counts
for threads in "${thread_counts[@]}"; do
    export OMP_NUM_THREADS=$threads

    ## Define GOMP_CPU_AFFINITY based on the number of threads
    case $threads in
        1)
            export GOMP_CPU_AFFINITY="0"
            ;;
        2)
            export GOMP_CPU_AFFINITY="0 1"
            ;;
        4)
            export GOMP_CPU_AFFINITY="0-3"
            ;;
        8)
            export GOMP_CPU_AFFINITY="0-7"
            ;;
        16)
            export GOMP_CPU_AFFINITY="0-15"
            ;;
        32)
            export GOMP_CPU_AFFINITY="0-31"
            ;;
        64)
            export GOMP_CPU_AFFINITY="0-63"
            ;;
    esac
```

2^η εκτέλεση (Use hyperthreading)

```
## Loop over the thread counts
for threads in "${thread_counts[@]}"; do
    export OMP_NUM_THREADS=$threads

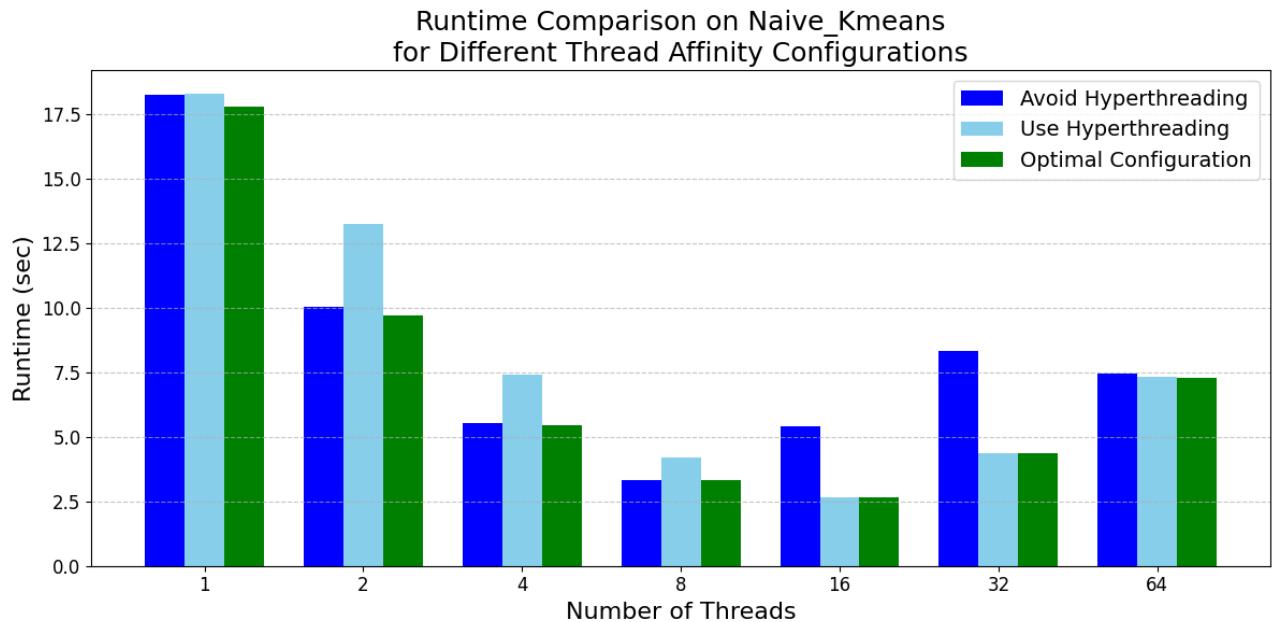
    ## Define GOMP_CPU_AFFINITY based on the number of threads
    case $threads in
        1)
            export GOMP_CPU_AFFINITY="0"
            ;;
        2)
            export GOMP_CPU_AFFINITY="0 32"
            ;;
        4)
            export GOMP_CPU_AFFINITY="0 1 32 33"
            ;;
        8)
            export GOMP_CPU_AFFINITY="0-3 32-35"
            ;;
        16)
            export GOMP_CPU_AFFINITY="0-7 32-39"
            ;;
        32)
            export GOMP_CPU_AFFINITY="0-15 32-47"
            ;;
        64)
            export GOMP_CPU_AFFINITY="0-63"
            ;;
    esac
```

3^η εκτέλεση (Optimal configuration)

```
## Loop over the thread counts
for threads in "${thread_counts[@]}"; do
    export OMP_NUM_THREADS=$threads

    ## Define GOMP_CPU_AFFINITY based on the number of threads
    case $threads in
        1)
            export GOMP_CPU_AFFINITY="0"
            ;;
        2)
            export GOMP_CPU_AFFINITY="0 1"
            ;;
        4)
            export GOMP_CPU_AFFINITY="0-3"
            ;;
        8)
            export GOMP_CPU_AFFINITY="0-7"
            ;;
        16)
            export GOMP_CPU_AFFINITY="0-7 32-39"
            ;;
        32)
            export GOMP_CPU_AFFINITY="0-15 32-47"
            ;;
        64)
            export GOMP_CPU_AFFINITY="0-63"
            ;;
    esac
```

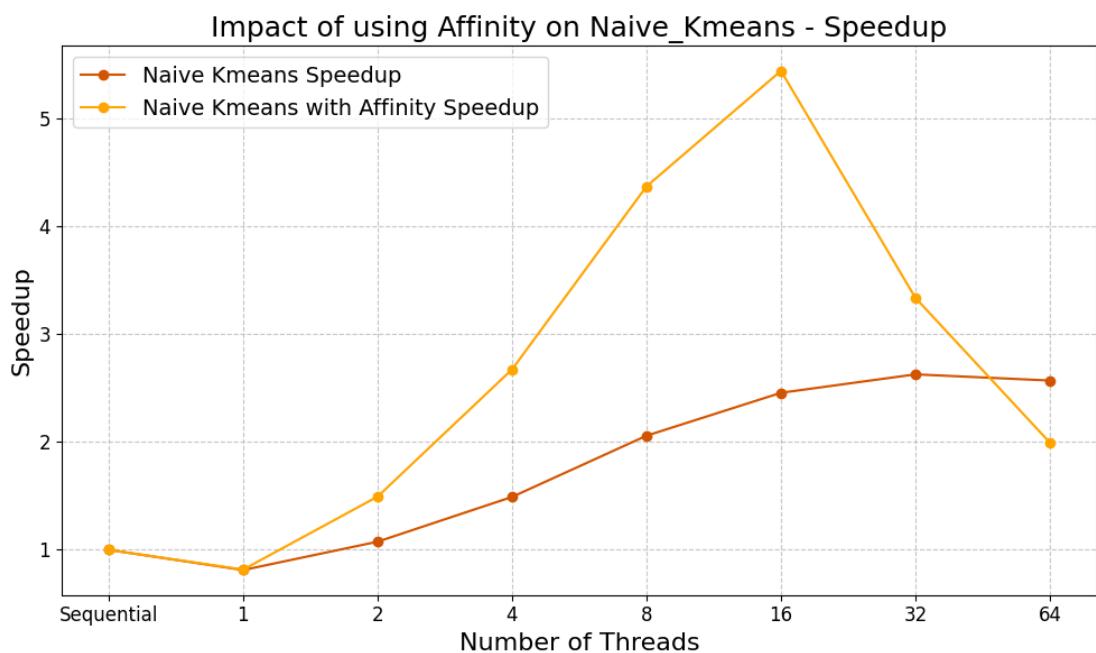
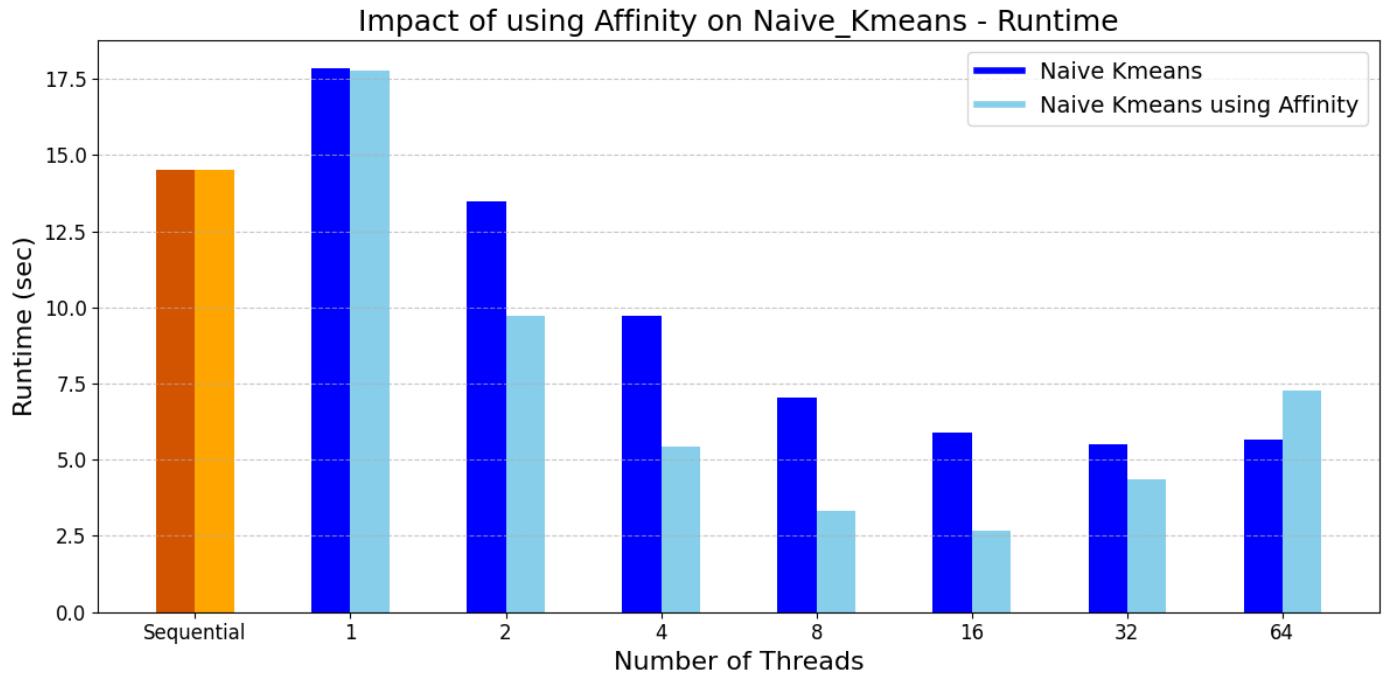
Στη συνέχεια, θα παρουσιάσουμε το διάγραμμα που απεικονίζει το χρόνο εκτέλεσης (runtime) για τις τρεις παραπάνω περιπτώσεις.



Σχολιασμός και συμπεράσματα:

Παρατηρούμε ότι για μικρό αριθμό *threads* (1, 2, 4, 8), τα καλύτερα αποτελέσματα επιτυγχάνονται όταν τα *threads* κατανέμονται σε διαφορετικούς πυρήνες εντός του ίδιου NUMA node (π.χ. NUMA Node 0). Αυτό διατηρεί την πρόσβαση στη μνήμη τοπική και αξιοποιεί τη δυνατότητα κοινής χρήσης της L3 cache. Για 16 και 32 *threads*, η βέλτιστη απόδοση επιτυγχάνεται μέσω της αξιοποίησης του hyperthreading, δηλαδή δεν απόδίδεται σε κάθε *thread* ένας dedicated core, αλλά η πρόσδεση των *threads* πραγματοποιείται σε όσο το δυνατόν λιγότερα NUMA nodes (NUMA Node 0 για 16 *threads* και NUMA Nodes 0 και 1 για 32). Αυτή η παρατήρηση φανερώνει ότι στον συγκεκριμένο αλγόριθμο και για το εν λόγω configuration, προτεραιότητα έχει η εγγύτητα των δεδομένων (*locality*). Με άλλα λόγια, η τοπική (*local*) μνήμη υπερτερεί έναντι της επεξεργαστικής και υπολογιστικής ισχύος, το οποίο είναι αναμενόμενο καθώς εκτελούμε μία απλή πράξη σε μία πληθώρα δεδομένων.

Θα παρουσιάσουμε στη συνέχεια διαγράμματα για να συγκρίνουμε την optimal affinity εκτέλεση με την προηγούμενη υλοποίηση naive_kmeans.



Σχολιασμός και συμπεράσματα:

Η πρόσδεση των *threads* σε συγκεκριμένους πυρήνες οδηγεί σε σημαντικά καλύτερη κλιμακωσιμότητα μέχρι και τα 16 *threads*, ενώ και μέχρι και τα 32 *threads* καταφέραμε να μειώσουμε τον χρόνο εκτέλεσης, χωρίς να αποφύγουμε ωστόσο το *overhead* της προσπέλασης στη μνήμη από τον μεγάλο αριθμό *threads*.

Η καλύτερη αυτή συμπεριφορά ήταν αναμενόμενη με βάση την προηγούμενη ανάλυση.

Ο καλύτερος χρόνος που επιτεύχθηκε ήταν 2.67 sec με 16 threads, οδηγώντας σε ένα speedup λίγο πάνω από το 5.

Κώδικας που υλοποιήθηκε:

Τέλος παραθέτουμε την υλοποίηση του παράλληλου προγράμματος kmeans_naive:

```
/*
#pragma omp parallel for shared(newClusters, newClusterSize) 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];
        }
    }
}
```

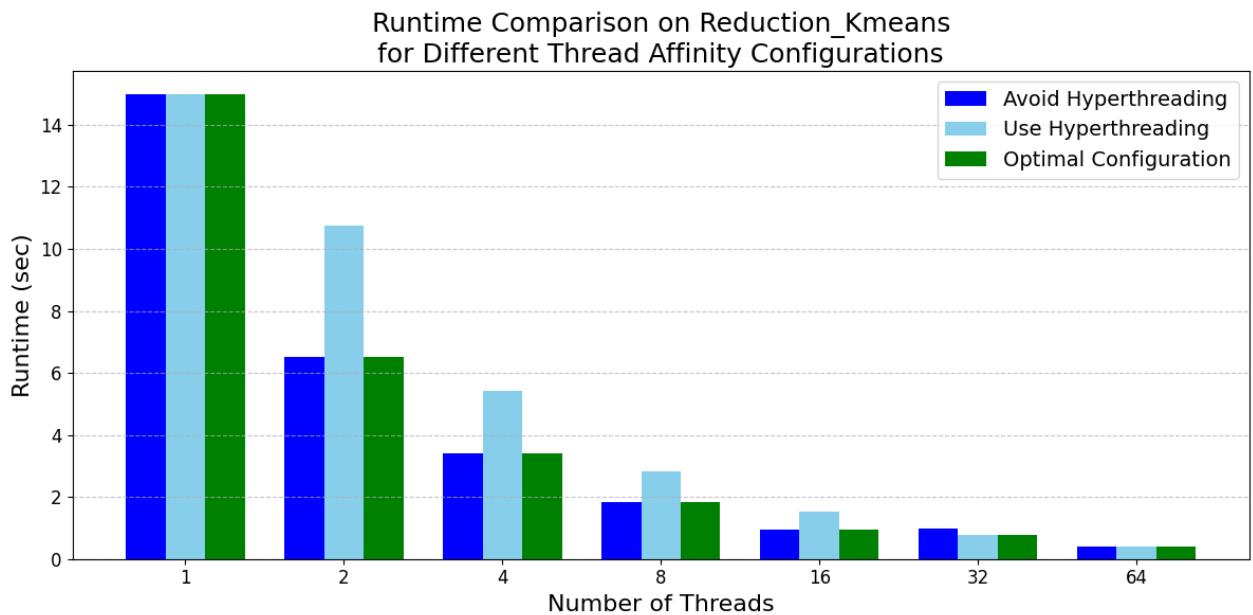
Στον παραπάνω κώδικα γίνεται φανερό πως χρησιμοποιήσαμε για την παραλληλοποίηση την εντολή **#pragma omp parallel for** και για την ορθότητα του προγράμματος την εντολή **#pragma omp atomic**.

(Σημειώνεται πως παραθέσαμε μόνο τον κώδικα στον οποίο προσαρμόσαμε τις εντολές παραλληλοποίησης και όχι όλο τον κώδικα του αλγορίθμου).

2^η υλοποίηση: Copied Clusters and Reduce

Θα παραλληλοποιήσουμε τώρα τον αλγόριθμο $k\text{-means}$ με μία νέα προσέγγιση, όπου κάθε νήμα θα διατηρεί τοπικά αντίγραφα των πινάκων *newClusters* και *newClusterSize*, αντί να είναι κοινόχρηστα. Όπως και στην προηγούμενη προσέγγιση, η παράλληλη εκτέλεση θα εφαρμοστεί στη διαδικασία υπολογισμού των αποστάσεων κάθε σημείου από τα κέντρα των clusters. Στη συνέχεια, τα τοπικά αποτελέσματα θα συνδυάζονται μέσω χειροκίνητης διαδικασίας *reduction*, καθώς το OpenMP δεν υποστηρίζει αυτόματο *reduction* για σύνθετες πράξεις σε πίνακες. Για τον σκοπό αυτό, εξασφαλίζουμε ότι ένα μόνο νήμα θα αναλάβει τη σύνθεση των αποτελεσμάτων, διατηρώντας την ορθότητα και την αποδοτικότητα στον τελικό υπολογισμό.

Θα ακολουθήσουμε την ίδια μεθοδολογία διερεύνησης όπως και προηγουμένως, ώστε να εντοπίσουμε το βέλτιστο ταίριασμα των threads και να αξιοποιήσουμε την δυνατότητα του hyperthreading στους επεξεργαστές. Στο διάγραμμα χρόνων εκτέλεσης που παρουσιάζεται, οι πρώτες δύο εκτελέσεις ακολουθούν την ίδια παραμετροποίηση: η πρώτη αποφεύγει το hyperthreading, ενώ η δεύτερη το αξιοποιεί πλήρως. Η τρίτη εκτέλεση απεικονίζει την καλύτερη δυνατή ρύθμιση (optimal) που προέκυψε από τις προηγούμενες δοκιμές, επιτυγχάνοντας τον ελάχιστο χρόνο εκτέλεσης.



To optimal configuration που προέκυψε είναι το:

```
## Loop over the thread counts
for threads in "${thread_counts[@]}"; do
    export OMP_NUM_THREADS=$threads

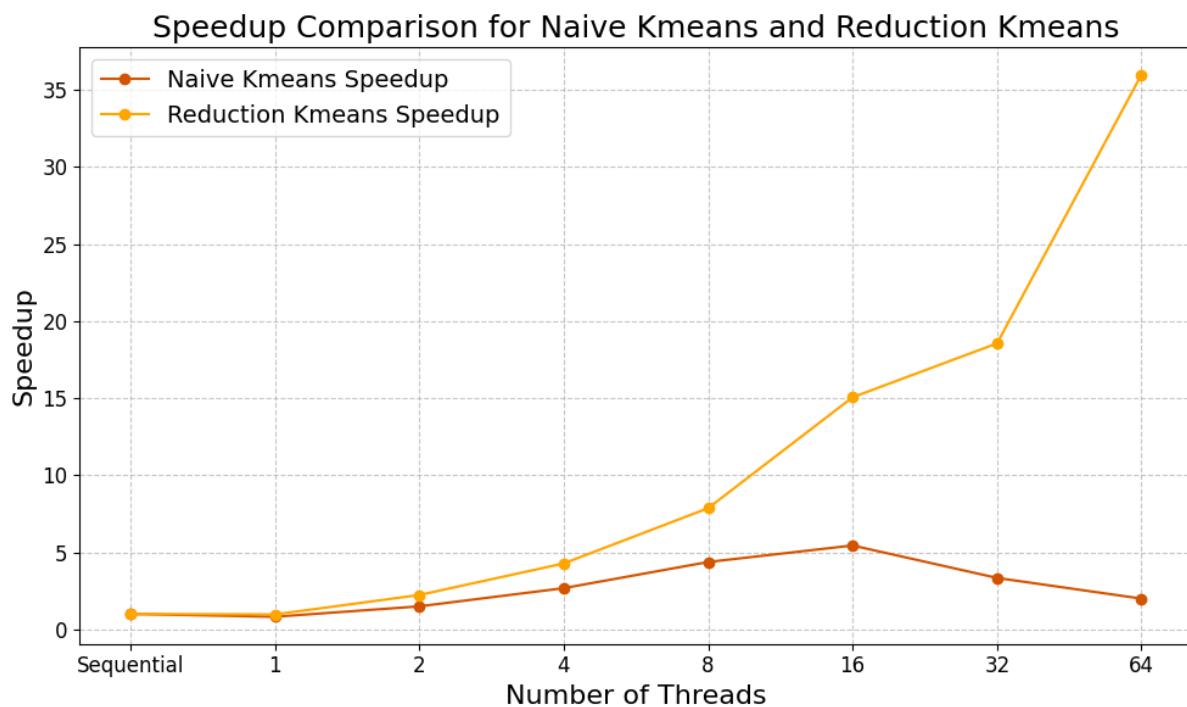
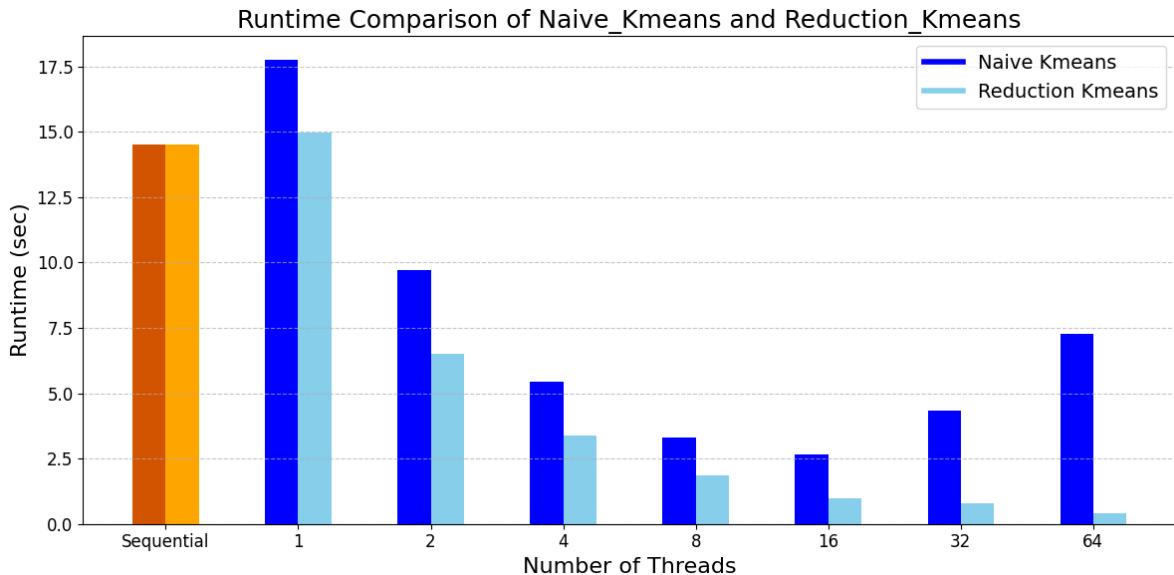
    ## Define GOMP_CPU_AFFINITY based on the number of threads
    case $threads in
        1)
            export GOMP_CPU_AFFINITY="0"
            ;;
        2)
            export GOMP_CPU_AFFINITY="0 1"
            ;;
        4)
            export GOMP_CPU_AFFINITY="0-3"
            ;;
        8)
            export GOMP_CPU_AFFINITY="0-7"
            ;;
        16)
            export GOMP_CPU_AFFINITY="0-15"

            ##echo "Running with $threads threads and affinity $GOMP_CPU_AFFINITY" > Test.out
            ;;
        32)
            export GOMP_CPU_AFFINITY="0-15 32-47"
            ;;
        64)
            export GOMP_CPU_AFFINITY="0-63"
            ;;
    esac
```

Σχολιασμός και συμπεράσματα:

Παρατηρούμε ότι, σε αντίθεση με προηγουμένως, για 16 *threads* αποδίδει καλύτερα η χρήση δύο NUMA nodes (0 και 1) αντί για τη χρήση μόνο του πρώτου *node* με αξιοποίηση του *hyperthreading*. Αυτό υποδεικνύει ότι, σε αυτό το στάδιο της παραλληλοποίησης, η προτεραιότητα δίνεται στην αξιοποίηση των φυσικών πόρων του συστήματος – υποδηλώνοντας ότι το *task* είναι περισσότερο *compute-bound* συγκριτικά με πριν. Η επιλογή των δύο NUMA nodes επιτρέπει στα *threads* να χρησιμοποιούν περισσότερους φυσικούς πυρήνες, περιορίζοντας την ανάγκη για *hyperthreading* και μεγιστοποιώντας την υπολογιστική ισχύ, η οποία κρίνεται πιο αποδοτική για την τρέχουσα υλοποίηση. Αυτό ενδεχομένως οφείλεται στο γεγονός ότι η αλλαγή από *shared* σε *copied* δεδομένα σημαίνει ότι κάθε *thread* δουλεύει αποκλειστικά στη δική του διάσταση, μειώνοντας τις προσπελάσεις κοινής μνήμης και ελαχιστοποιώντας τις καθυστερήσεις λόγω συγχρονισμού. Έτσι, η επιβάρυνση μετατοπίζεται περισσότερο στο κομμάτι των υπολογισμών, όπου η εκμετάλλευση φυσικών πυρήνων από διαφορετικά NUMA nodes αποδίδει καλύτερα από την υπερ-εκμετάλλευση ενός μόνο *node* με *hyperthreading*.

Θα συγκρίνουμε τώρα την optimal εκτέλεση που πήραμε για το *shared_clusters* με εκείνη του *copied_clusters*.



Σχολιασμός και συμπεράσματα:

Παρατηρούμε ότι η επίδοση βελτιώνεται σημαντικά με τη χρήση τοπικών αντιγράφων δεδομένων για όλα τα πλήθη *threads*. Ο χρόνος εκτέλεσης μειώνεται σχεδόν σταθερά, υποδιπλασιαζόμενος κάθε φορά που διπλασιάζεται ο αριθμός των *threads*, ενώ το speedup παραμένει σχεδόν γραμμικό μέχρι τα 16 *threads*. Αυτή η βελτίωση οφείλεται στο γεγονός ότι δεν απαιτείται πλέον συγχρονισμός μεταξύ των *threads*, καθώς ο συγχρονισμός και η χρήση *atomic* λειτουργιών, που

ήταν απαραίτητα στη *naive* υλοποίηση, προσέθεταν σημαντικό *overhead*, επιβαρύνοντας την επίδοση.

Είναι σαφές ότι ο υπολογισμός των αποστάσεων που εκτελεί κάθε *thread* δεν εξαρτάται από τα αποτελέσματα των υπολογισμών των άλλων *threads*, αλλά μόνο από τα κέντρα των *clusters*, όπως είχαν υπολογιστεί στην προηγούμενη επανάληψη. Συνεπώς, είναι βέλτιστο το κάθε *thread* να εργάζεται σε τοπικά αντίγραφα δεδομένων, αποφεύγοντας το *overhead* του συγχρονισμού και των *atomic* ενημερώσεων στους κοινόχρηστους πίνακες. Αυτή η προσέγγιση μειώνει τον χρόνο εκτέλεσης και βελτιώνει την αποδοτικότητα του αλγορίθμου.

Ο καλύτερος χρόνος που καταγράψαμε μειώνεται από 2.67 sec που είχαμε στην *naive* υλοποίηση σε 0.4 sec για 64 *threads*.

Κώδικας που υλοποιήθηκε:

Παραθέτουμε εδώ τον κώδικα που αναπτύξαμε για την παραλληλοποίηση του *omp_reduction_kmeans*:

```

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

#pragma omp parallel for shared(newClusters, newClusterSize) private(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 later)
    /*
     * TODO: Collect cluster data in local arrays (Local to each thread)
     * Replace global arrays with local per-thread
     */
    int thread_num = omp_get_thread_num();
    int nthreads = omp_get_num_threads(); // Get total number of threads
    // Dynamic mapping logic
    int mapped_thread_id;

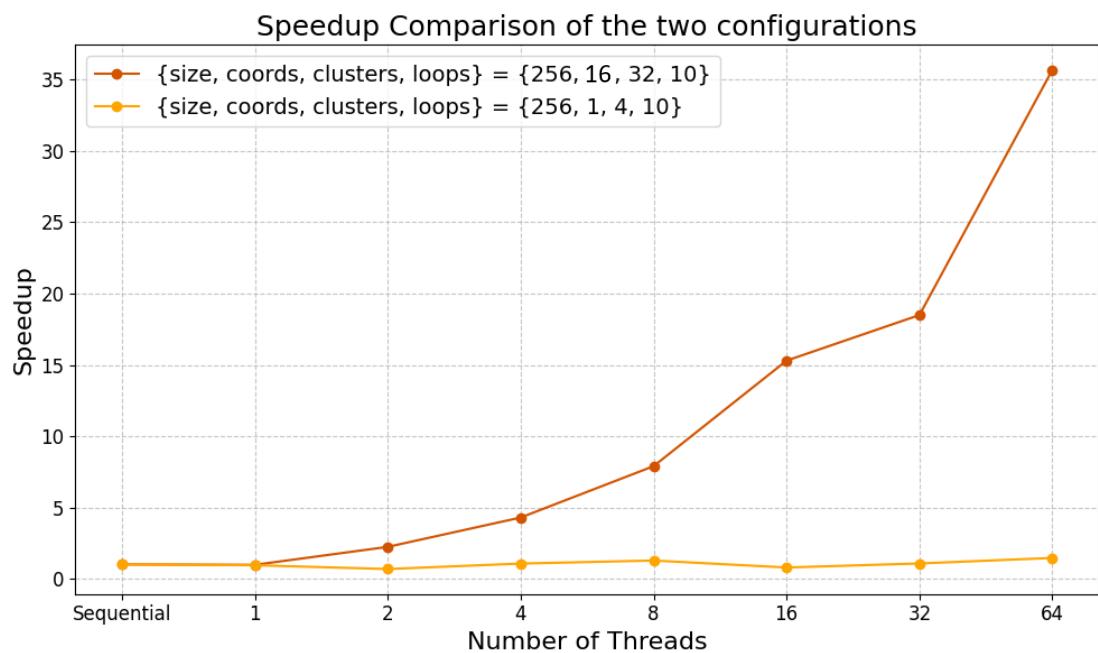
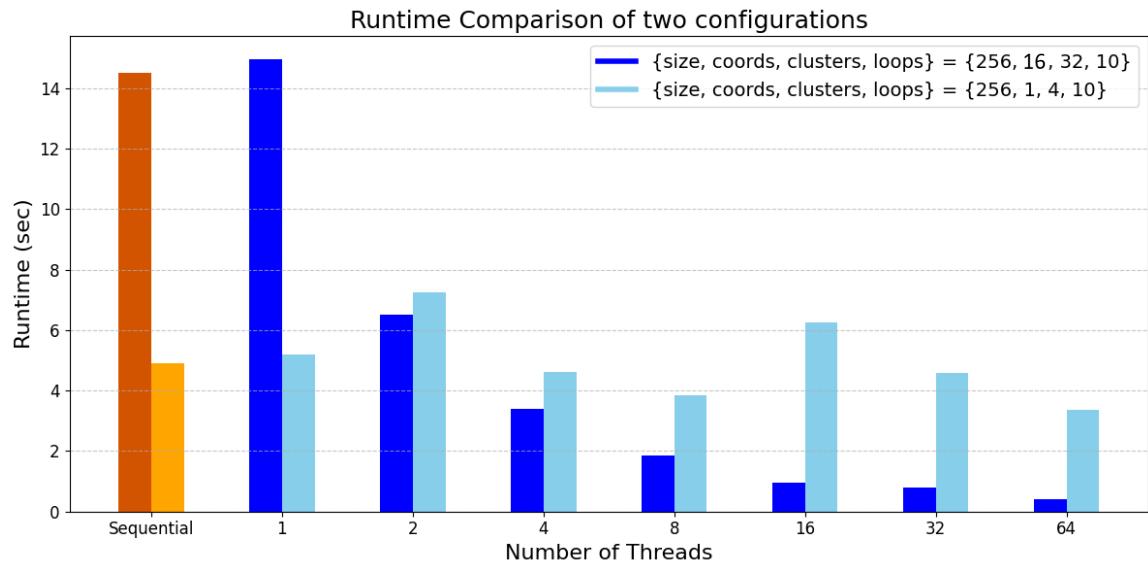
    if (nthreads == 16) {
        mapped_thread_id=thread_num;
        //mapped_thread_id = (thread_num < 8) ? thread_num : thread_num - 24;
        //printf("\r\n\t thread after mapping is %d", mapped_thread_id);
    } else if (nthreads == 32) {
        mapped_thread_id = thread_num;
        //mapped_thread_id = (thread_num < 16) ? thread_num : thread_num - 16;
    } else {
        mapped_thread_id = thread_num; // Default mapping for other cases
    }

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

```

Θα δοκιμάσουμε ένα διαφορετικό configuration στην ίδια υλοποίηση, με τις παραμέτρους: $\{Size, Coords, Clusters, Loops\} = \{256, 1, 4, 10\}$. Σε αυτή την περίπτωση, μειώνοντας τον αριθμό των ομάδων και των διαστάσεων, το κάθε δεδομένο θα είναι σημαντικά μικρότερο. Στη συνέχεια, θα συγκρίνουμε το scalability που επιτυγχάνεται για τις δύο αυτές παραμετροποιήσεις, προκειμένου να αξιολογήσουμε την επίδραση των αλλαγών αυτών στην απόδοση του αλγορίθμου.¹⁶



Σχολιασμός και συμπεράσματα:

Παρατηρούμε ότι με το μικρότερο configuration, η απόδοση του προγράμματος μειώθηκε σημαντικά, διαταράσσοντας την κλιμακωσιμότητα που είχε παρατηρηθεί προηγουμένως. Αυτή η υποβάθμιση της απόδοσης οφείλεται σε αναποτελεσματική διαχείριση της cache, η οποία επηρεάζεται από φαινόμενα όπως το **false sharing** και η **first touch** πολιτική. Τα φαινόμενα αυτά, που θα αναλύσουμε παρακάτω, προκαλούν αυξημένες καθυστερήσεις στη μνήμη, επιβαρύνοντας την επίδοση του προγράμματος και περιορίζοντας την αποδοτικότητα του παραλληλισμού.

False Sharing

Το φαινόμενο του *false sharing* προκύπτει όταν πολλαπλά threads προσπελαύνουν ξεχωριστές μεταβλητές που βρίσκονται στο ίδιο cache line (μονάδα δεδομένων στην cache), με τουλάχιστον ένα από τα threads να πραγματοποιεί εγγραφή. Παρά το ότι τα threads δεν μοιράζονται άμεσα δεδομένα, η εγγύτητά τους στο ίδιο cache line προκαλεί περιττό συγχρονισμό και καθυστερήσεις λόγω της ανάγκης για ενημέρωση της cache. Αυτό μειώνει την απόδοση σε πολυπύρηνα συστήματα, καθώς οι cache των επεξεργαστών πρέπει να συγχρονίζονται συνεχώς.

First Touch Policy

Η πολιτική *First Touch* στα Linux αφορά τον τρόπο κατανομής της φυσικής μνήμης σε αρχιτεκτονικές Non-Uniform Memory Access (NUMA). Σύμφωνα με αυτήν την πολιτική, όταν μια σελίδα μνήμης προσπελαστεί για πρώτη φορά από ένα thread, η μνήμη δεσμεύεται στο κοντινότερο NUMA node, μειώνοντας έτσι την απόσταση που απαιτείται για τις μελλοντικές προσπελάσεις. Η *First Touch* επιτρέπει πιο αποδοτική χρήση της μνήμης σε πολυπύρηνα συστήματα με NUMA, βελτιώνοντας τον χρόνο πρόσβασης και την απόδοση των εφαρμογών που απαιτούν εντατική χρήση μνήμης.

Θα προσπαθήσουμε τώρα να λύσουμε τα παραπάνω προβλήματα για να βελτιώσουμε την επίδοση της παραλληλοποίησης.

- **First Touch**

Στην προηγούμενη υλοποίηση της παραλληλοποίησης, η αρχικοποίηση των τοπικών πινάκων *local_NewClusters* και *local_newClusterSize* γινόταν από ένα μόνο νήμα, το οποίο δημιουργούσε όλα τα δεδομένα. Σύμφωνα με την πολιτική *First Touch*, αυτό είχε ως αποτέλεσμα η μνήμη για τους πίνακες αυτούς να δεσμεύεται κοντά στον επεξεργαστή του νήματος που έκανε την αρχικοποίηση. Ωστόσο, καθώς αυτά τα δεδομένα έπρεπε να επεξεργαστούν αργότερα από άλλα νήματα σε διαφορετικούς επεξεργαστές, η τοποθέτησή τους στη μνήμη παρέμενε "απομακρυσμένη" από αυτούς τους επεξεργαστές.

Αυτός ο τρόπος προσπέλασης δημιουργεί αυξημένο *latency* στην πρόσβαση στα δεδομένα, καθώς τα άλλα νήματα αναγκάζονται να προσπελάσουν δεδομένα που βρίσκονται εκτός της τοπικής τους *cache*, αυξάνοντας έτσι τον χρόνο που απαιτείται για κάθε προσπέλαση μνήμης. Ειδικά σε αρχιτεκτονικές NUMA (Non-Uniform Memory Access), όπου η απόσταση και ο χρόνος πρόσβασης στα δεδομένα εξαρτώνται από τη φυσική τοποθεσία τους, η απόδοση επηρεάζεται σημαντικά. Μια πιο αποδοτική προσέγγιση θα ήταν να αναθέσουμε σε κάθε νήμα να αρχικοποιεί τα δικά του τοπικά δεδομένα, εξασφαλίζοντας ότι αυτά θα βρίσκονται κοντά στον επεξεργαστή που θα τα χρησιμοποιήσει. Με αυτόν τον τρόπο, θα μειωθεί το *latency* και θα βελτιστοποιηθεί η χρήση της τοπικής *cache*, ενισχύοντας την αποδοτικότητα της παράλληλης επεξεργασίας.

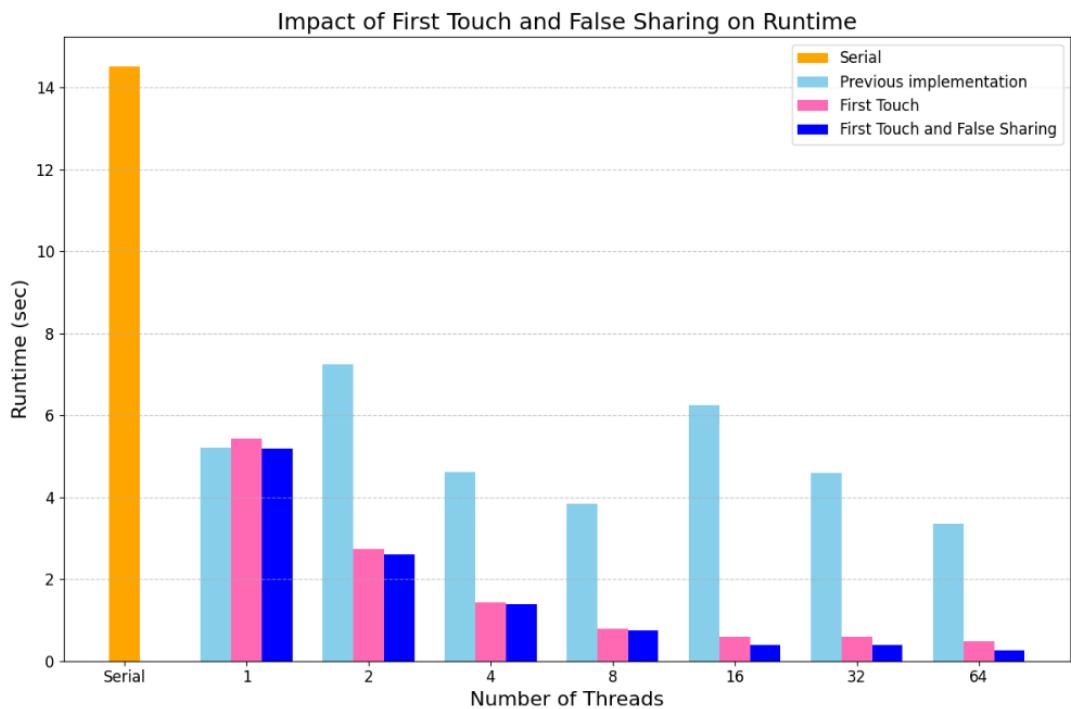
Για να αντιμετωπιστεί αυτό το πρόβλημα, θα εφαρμόσουμε παράλληλη αρχικοποίηση των πινάκων, με τη χρήση της εντολής **calloc** έναντι της **malloc**, ώστε κάθε επεξεργαστής να αναλαμβάνει την αρχικοποίηση του τμήματος του πίνακα που θα επεξεργαστεί. Με αυτόν τον τρόπο, η μνήμη για κάθε τμήμα θα δεσμεύεται κοντά στον αντίστοιχο επεξεργαστή και αναμένουμε βελτίωση της επίδοσης.

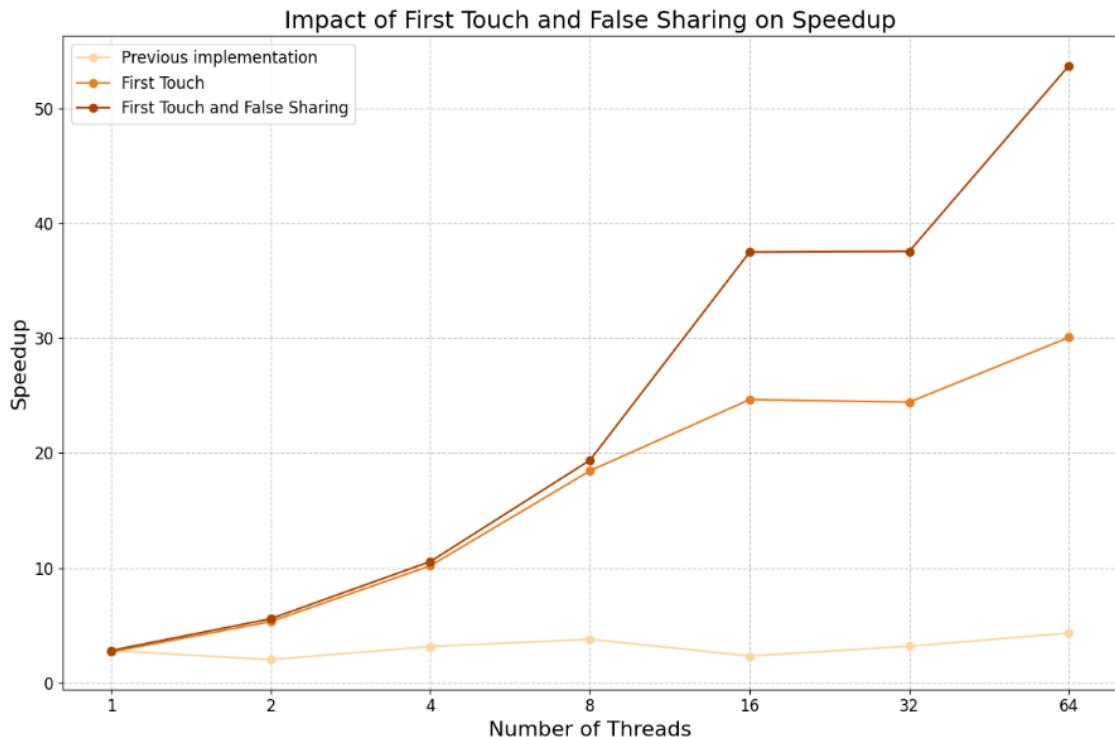
- **False sharing**

Με το μικρότερο configuration, τα δεδομένα που επεξεργάζονται τα νήματα αποτελούν πίνακα διαστάσεων 4x1, ο οποίος περιέχει δεδομένα τύπου double (8 bytes). Κάθε γραμμή του πίνακα καταλαμβάνει 32 bytes. Ωστόσο, δεδομένου ότι το μέγεθος γραμμής *cache* (cache line) στο σύστημά μας είναι 64 bytes, οι τοπικοί πίνακες δύο διαφορετικών threads αποθηκεύονται στην ίδια γραμμή *cache*. Αυτό οδηγεί σε φαινόμενο **false sharing**, καθώς η παράλληλη πρόσβαση των threads στα δεδομένα της ίδιας γραμμής *cache* προκαλεί συχνά invalidations και αυξημένο *latency*, με αποτέλεσμα τη σημαντική επιβάρυνση της απόδοσης του προγράμματος.

Για να αντιμετωπιστεί αυτό, χρησιμοποιούμε ευθυγράμμιση και padding. Η ευθυγράμμιση επιτυγχάνεται με τη χρήση της εντολής **posix_memalign**, η οποία διασφαλίζει ότι κάθε τοπικός πίνακας ξεκινά σε διεύθυνση πολλαπλάσιο του μεγέθους γραμμής cache (CACHE_LINE_SIZE). Παράλληλα, το padding εφαρμόζεται με την προσθήκη επιπλέον μνήμης μέσω κατάλληλου υπολογισμού, με τη χρήση της συνάρτησης **get_padded_size**, ώστε οι πίνακες διαφορετικών threads να βρίσκονται σε ξεχωριστές γραμμές cache. Ο συνδυασμός αυτών των τεχνικών μειώνει τα φαινόμενα false sharing, ελαχιστοποιεί τα cache misses και επιτρέπει τη βέλτιστη αξιοποίηση της μνήμης, βελτιώνοντας την απόδοση και την κλιμάκωση του προγράμματος.

Παρακάτω παρουσιάζονται τα διαγράμματα του χρόνου εκτέλεσης και του speedup, τα οποία καταγράψαμε συγκρίνοντας τρεις διαφορετικές περιπτώσεις: την αρχική επίδοση που παρατηρήθηκε, την επίδοση μετά την αντιμετώπιση του φαινομένου first touch, καθώς και την επίδοση μετά την επίλυση τόσο του first touch όσο και του false sharing.





Σχολιασμός και συμπεράσματα:

Από τα παραπάνω διαγράμματα, γίνεται φανερό ότι οι βελτιστοποιήσεις που εφαρμόσαμε συμβάλλουν ουσιαστικά στην κλιμακωσιμότητα της παραλληλοποίησης. Στο πρώτο διάγραμμα, παρατηρούμε ότι η αντιμετώπιση του φαινομένου **first touch** είναι καθοριστική για τη μείωση του χρόνου εκτέλεσης, μειώνοντάς τον από 3.3632 δευτερόλεπτα για 64 threads (χωρίς βελτιστοποίηση) σε 0.4826 δευτερόλεπτα, δηλαδή περίπου 7 φορές μικρότερο χρόνο. Αυτή η βελτίωση είναι αναμενόμενη, καθώς η συνεχής πρόσβαση σε μακρινή μνήμη αποτελούσε σημαντικό overhead, το οποίο εξαλείφθηκε με την τοπική κατανομή των δεδομένων στη μνήμη.

Η περαιτέρω βελτιστοποίηση με την αντιμετώπιση του **false sharing** βελτιώνει ακόμη περισσότερο την απόδοση, μειώνοντάς τον χρόνο εκτέλεσης στα 0.27 δευτερόλεπτα για 64 threads, σχεδόν στο μισό από τον χρόνο που παρατηρήθηκε με την αντιμετώπιση μόνο του *first touch*. Μετά από αυτές τις βελτιστοποιήσεις, παρατηρούμε ικανοποιητική κλιμακωσιμότητα, με συνεχή βελτίωση του χρόνου εκτέλεσης καθώς αυξάνεται ο αριθμός των threads. Συνολικά, οι βελτιστοποιήσεις αυτές ενισχύουν την αποδοτικότητα του παραλληλισμού, ελαχιστοποιώντας το overhead από τη μνήμη και βελτιώνοντας την αξιοποίηση των διαθέσιμων υπολογιστικών πόρων.

Κώδικας που υλοποιήθηκε:

```
#define CACHE_LINE_SIZE 64

// Helper function to calculate padded size
size_t get_padded_size(size_t base_size, size_t element_size) {
    size_t total_size = base_size * element_size;
    size_t padding = CACHE_LINE_SIZE - (total_size % CACHE_LINE_SIZE);
    return total_size + padding;
}
```

```
#pragma omp parallel for
for (k=0; k<nthreads; k++) //initializing local arrays for each thread
{
    //first touch
    //local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters, sizeof(**local_newClusterSize));
    //local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords, sizeof(**local_newClusters));

    //first touch and false sharing
    // Padding to avoid false sharing
    size_t padded_size = get_padded_size(numClusters, sizeof(int));
    posix_memalign((void **)&local_newClusterSize[k], CACHE_LINE_SIZE, padded_size);

    // Padding for the clusters array as well
    size_t padded_clusters_size = get_padded_size(numClusters * numCoords, sizeof(double));
    posix_memalign((void **)&local_newClusters[k], CACHE_LINE_SIZE, padded_clusters_size);

    // Initialize memory with zeros
    memset(local_newClusterSize[k], 0, padded_size);
    memset(local_newClusters[k], 0, padded_clusters_size);
}
```

Bonus: NUMA-aware allocation

Στο configuration {Size, Coords, Clusters, Loops}={256, 1, 4, 10} υλοποιήσαμε την NUMA-aware κατανομή μνήμης για τα πίνακα objects. Αυτό επιτεύχθηκε τροποποιώντας το αρχείο file_io.c ώστε ο βρόχος που αρχικοποιεί τα πίνακα με τυχαίες τιμές να παραλληλοποιείται ως εξής:

```
double * dataset_generation(int numObjs, int numCoords)
{
    double * objects = malloc(numObjs * numCoords * sizeof(*objects));
    double val_range = 10.0;
    long i,j;
    // Parallelize the outer loop for NUMA-aware first-touch placement.
    #pragma omp parallel for schedule(static)
    for (i=0; i < numObjs; i++) {
        unsigned int seed = i; // Each object gets a unique seed
        long offset = i * numCoords;

        // Vectorize the inner loop to improve throughput
        #pragma omp simd
        for (j=0; j < numCoords; j++) {
            objects[offset + j] = (rand_r(&seed) / ((double) RAND_MAX)) * val_range;
        }
    }

    return objects;
}
```

Για να επιτευχθεί η βέλτιστη κατανομή της μνήμης σύμφωνα με την πολιτική first-touch σε NUMA περιβάλλον, χρησιμοποιούμε το `#pragma omp parallel for schedule(static)` στον εξωτερικό βρόχο. Με αυτόν τον τρόπο, οι επαναλήψεις του βρόχου διανέμονται στα διαθέσιμα νήματα με στατική κατανομή, πράγμα που σημαίνει ότι κάθε νήμα αναλαμβάνει ένα συνεχές τμήμα των επαναλήψεων και, κατά συνέπεια, της μνήμης που αφορά αυτά τα δεδομένα. Αυτό εξασφαλίζει ότι κάθε νήμα "αγγίζει" πρώτα τα δεδομένα που θα χρησιμοποιήσει, οδηγώντας σε τοπική τοποθέτηση της μνήμης και μειώνοντας έτσι την καθυστέρηση πρόσβασης.

Μέσα στον εξωτερικό βρόχο, ορίζεται το seed ως η τιμή του `i`, έτσι ώστε κάθε αντικείμενο να έχει το δικό του ξεχωριστό seed, εξασφαλίζοντας thread-safe γενιά τυχαίων αριθμών μέσω της συνάρτησης `rand_r`. Στη συνέχεια, χρησιμοποιείται ο εσωτερικός βρόχος για την επανάληψη επί των συντεταγμένων του αντικειμένου, ο οποίος έχει επισημανθεί με το `#pragma omp simd`. Αυτή η οδηγία δίνει το σήμα στον compiler ότι οι πράξεις μέσα στον βρόχο μπορούν να γίνουν vectorized, δηλαδή να εκτελούνται παράλληλα σε επίπεδο SIMD, γεγονός που βελτιώνει σημαντικά την απόδοση των υπολογισμών.

To static scheduling και η χρήση SIMD (vectorization) έχουν γίνει για να επιτευχθεί μέγιστη απόδοση. To static scheduling εξασφαλίζει ομοιόμορφη κατανομή των iterations και τοπική πρόσβαση στη μνήμη (NUMA-aware), ενώ το SIMD στον ταυτόχρονο και γρήγορο υπολογισμό των τιμών, μειώνοντας το συνολικό χρόνο εκτέλεσης της συνάρτησης.

Με αυτήν την παραλληλοποίηση, κάθε νήμα "αγγίζει" το τμήμα της μνήμης που θα χρησιμοποιήσει αργότερα. Σύμφωνα με την πολιτική **first-touch** του Linux, η μνήμη που αρχικοποιείται από ένα νήμα τοποθετείται στο NUMA node που είναι πιο κοντά σε αυτό το νήμα. Αυτό οδηγεί σε μειωμένη καθυστέρηση πρόσβασης (lower memory latency) και, κατά συνέπεια, σε βελτίωση της απόδοσης του προγράμματος, ιδιαίτερα σε συστήματα με πολλαπλούς NUMA nodes.

Στο configuration {256, 1, 4, 10} έχουμε 33.554.432 αντικείμενα με μία συντεταγμένη το καθένα, οπότε ο συνολικός αριθμός στοιχείων είναι 33.554.432, ενώ στο configuration {256, 16, 32, 10} έχουμε 2.097.152 αντικείμενα με 16 συντεταγμένες το καθένα, που επίσης οδηγεί σε συνολικά 33.554.432 στοιχεία. Παρόλο που ο συνολικός αριθμός των στοιχείων που αρχικοποιούνται είναι ίδιος στις δύο διαμορφώσεις και επομένως ο χρόνος για την αρχικοποίηση δεν διαφέρει σημαντικά, το κύριο ζήτημα έγκειται στο υπολογιστικό φορτίο του αλγορίθμου kmeans. Συγκεκριμένα, στο πρώτο configuration, για κάθε iteration υπολογίζονται 33.554.432 αντικείμενα επί 4 clusters επί 1 συντεταγμένη, πράγμα που αντιστοιχεί περίπου σε 134.217.728 υπολογισμούς, ενώ στο δεύτερο configuration, για κάθε

iteration υπολογίζονται 2.097.152 αντικείμενα επί 32 clusters επί 16 συντεταγμένες, δηλαδή περίπου 1.073.741.824 υπολογισμοί (περίπου 8 φορές περισσότερους υπολογισμούς). Επομένως, ενώ η NUMA-aware allocation βελτιώνει την πρόσβαση στη μνήμη σε αμφότερα τα configurations, το κύριο bottleneck στην απόδοση παραμένει η κεντρική φάση του kmeans, όπου στο configuration {256, 16, 32, 10} ο αυξημένος αριθμός clusters και οι περισσότερες συντεταγμένες οδηγούν σε σημαντικά μεγαλύτερο υπολογιστικό φόρτο.

Αυτό μπορούμε να παρατηρήσουμε, και είναι και λογικό, είναι ότι γενικά μεταξύ των 2 configurations, το μεγαλύτερο κάνει πολύ καλύτερα scale από το μικρό, δεδομένης της παραλληλοποίησης που έχουμε υλοποιήσει. Όπως είπαμε και πιο πάνω, στο μεγάλο configuration οι υπολογισμοί μοιράζονται περισσότερο μεταξύ των 2 loops, σε αντίθεση με τον μικρό configuration, πράγμα που εκμεταλλεύμαστε με την χρήση SIMD γιατί μπορούμε και έχουμε έναν «τύπο nested παραλληλοποίησης», που ένα απλό parallel for δεν πετυχαίνει.

Συνολικά, και οι δύο διαμορφώσεις επωφελούνται από τη σωστή κατανομή της μνήμης, αλλά η διαφορά στην απόδοση οφείλεται κυρίως στο γεγονός ότι στο configuration {256, 16, 32, 10} η υπολογιστική πολυπλοκότητα του αλγορίθμου είναι πολύ υψηλότερη, καθιστώντας τον υπολογισμό των αποστάσεων και την ενημέρωση των κέντρων το κυρίαρχο bottleneck της εφαρμογής.

Δυστυχώς καθώς από τις 14/2 έως και τις 16/2 ο sandman δεν ήταν σε λειτουργία (task deferred) δεν μπορούσαμε να κάνουμε ολοκληρωμένες μετρήσεις και αντίστοιχα γραφήματα για να τεκμηριώσουμε την θέση μας. Αντ' αυτού, περιοριστήκαμε στην χρήση της γραμμής parlab που μας επιτρέπει το τρέξιμο μέχρι 8 νήματα σε 1 node. Οι χρόνοι βγαίνουν μεγαλύτεροι από τα προηγούμενα ερωτήματα, πράγμα που είναι λογικό, καθώς μιλάμε για άλλο μηχάνημα, με άλλη αρχιτεκτονική και δυναμική. Ωστόσο, καλύπτει τις ανάγκες του ερωτήματος καθώς το προσεγγίζουμε συγκριτικά με τους χρόνους πριν και μετά το NUMA Awareness.

Config {256,1,4,10}

Threads	Before	After
1	11.9634s	12.4319s
2	5.9888s	6.2160s
4	3.0010s	3.1177s
8	1.5187s	1.5672s

Config {256,16,32,10}

Threads	Before	After
1	30.3455s	26.5482s
2	15.2228s	13.2689s
4	7.6085s	6.6476s
8	3.8132s	3.3300s

Παρατηρήσεις:

Ο χρόνος για το μικρό Config βλέπουμε ότι γίνεται λίγο μεγαλύτερος. Το περιμέναμε γιατί το επιπρόσθετο overhead από τις παραλληλοποιήσεις και το vectorization δεν αποδίδει στο συγκεκριμένο configuration λόγω της διαδικασίας υπολογισμού που περιγράψαμε πριν, όπου το υπολογιστικό έργο είναι μικρό, αλλά οφείλεται και στο ότι δεν τρέχουμε στον sandman, όπου ενδεχομένως να είχαμε καλύτερο αποτέλεσμα. Γενικά η υλοποίηση με parallel for static και SIMD εξυπηρετεί σε περισσότερους υπολογισμούς.

Επιπλέον, στην υλοποίηση του NUMA awareness συγκαταλέγονται και τα προαναφερθέντα στα προηγούμενα ερωτήματα περί first-touch policy, false sharing (padding) καθώς και η χρήση affinity. Αν συμπεριλάβουμε όλα αυτά θα δούμε ότι το τελικό συνολικό speedup θα ήταν ιδιαίτερα καλύτερο.

Τέλος, μερικές ενδεικτικές τιμές που είχαμε τρέξιμο παλιότερα, όσο ο sandman λειτουργούσε ακόμα, πρόκειται για χρόνους με παραλληλοποίηση μόνο στο εξωτερικό loop, χωρίς static και SIMD implementation. Παρατηρείται πολύ ελαφριά κλιμάκωση καθώς το data generation δεν είναι το κύριο bottleneck. Με εισαγωγή των μεθόδων που εκτελέσαμε και πριν θα βλέπαμε ακόμα μεγαλύτερη βελτίωση.

Threads	Config {256,1,4,10}	Config {256,16,32,10}
1	5.1884s	12.8472s
2	2.6079s	6.4804s
4	1.3638s	3.3686s
8	0.7517s	1.8228s
16	0.3856s	0.9237s
32	0.3719s	0.7704s
64	0.2673s	0.4085s

Κώδικας που υλοποιήθηκε: (ΠΑΛΙΟΣ)

```
#pragma omp parallel for private(j) // parallelize the for loop
for (i=0; i<numObjs; i++)
{
    unsigned int seed = i;
    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]);
        }
    }
}
```

Σημειώνεται αυτός είναι ο κώδικας που είχαμε τρέξει στις προηγούμενες μετρήσεις στον *sandman* που αναφέρεται στην ανάλυσή μας (μετρήσεις προηγούμενου πίνακα).

Αμοιβαίος Αποκλεισμός - Κλειδώματα

Στην παρούσα μελέτη θα εξετάσουμε την επίδοση διαφορετικών κλειδωμάτων για την προστασία της ενημέρωσης του πίνακα newClusters. Συγκεκριμένα, θα συγκρίνουμε τις επιδόσεις των παρακάτω κλειδωμάτων:

- ✓ nosync_lock
- ✓ pthread_mutex_lock
- ✓ pthread_spin_lock
- ✓ tas_lock (Test-And-Set Lock)
- ✓ ttas_lock (Test-and-Test-and-Set Lock)
- ✓ array_lock
- ✓ clh_lock (Craig, Landin, and Hagersten Lock)
- ✓ #pragma omp atomic
- ✓ #pragma omp critical

Θα παρουσιάσουμε λίγες πληροφορίες για κάθε είδος κλειδώματος, έτσι ώστε να είναι πιο σαφής η ανάλυση αργότερα.

No_sync Lock

Αυτή η υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό, με αποτέλεσμα τα νήματα να μην συγχρονίζονται και η ενημέρωση του πίνακα newClusters να μην πραγματοποιείται σωστά. Θα χρησιμοποιήσουμε τη μέθοδο ως σημείο αναφοράς για το ανώτερο όριο ταχύτητας χωρίς συγχρονισμό.

Pthread_mutex_lock

Είναι βασικό κλείδωμα της βιβλιοθήκης Pthreads. Ένα νήμα καλώντας τη συνάρτηση pthread_mutex_lock αποκτά ένα κλείδωμα, εφόσον δεν το έχει κάποιο άλλο thread. Όλα τα υπόλοιπα threads βρίσκονται σε αναμονή μέχρι να ελευθερωθεί το κλείδωμα.

Pthread_spin_lock

Είναι και αυτό ένα κλείδωμα που παρέχεται στη βιβλιοθήκη Pthreads. Υλοποιεί αμοιβαίο αποκλεισμό (mutual exclusion) μέσω ενεργής αναμονής (busy waiting). Είναι ιδανικό για περιπτώσεις όπου ο χρόνος αναμονής για το κλείδωμα αναμένεται να είναι πολύ μικρός. Αντί να θέτει το νήμα σε κατάσταση αναμονής όπως τα pthread_mutex_lock, το spinlock διατηρεί το νήμα ενεργό και επαναλαμβάνει (spins) την προσπάθεια απόκτησης του κλειδώματος μέχρι να το αποκτήσει. Δεν χρειάζεται επικοινωνία δηλαδή μεταξύ των νημάτων για να καταλάβουν εάν ελευθερώθηκε η κρίσιμη περιοχή επειδή κάθε νήμα ελέγχει συνεχώς την κατάστασή της όσο δεν βρίσκεται στο κρίσιμο τμήμα.

Tas_lock (Test-and-Set Lock)

Είναι ένας απλός μηχανισμός κλειδώματος που υλοποιεί αμοιβαίο αποκλεισμό (mutual exclusion) χρησιμοποιώντας την ατομική λειτουργία Test-And-Set που παρέχεται από το υλικό. Η λειτουργία Test-And-Set ελέγχει την τιμή δύο μεταβλητών state και test. Η μεταβλητή state ορίζει την τρέχουσα κατάσταση, δηλαδή εάν ένα νήμα θέλει να κλειδώσει το κρίσιμο τμήμα, και η μεταβλητή test κρατάει την προηγούμενη κατάσταση, δηλαδή εάν η περιοχή προηγουμένως ήταν κλειδωμένη ή ελεύθερη. Εάν ένα νήμα θέσει state = 1 (lock) και βρει την μεταβλητή test = 0 (unlocked) θα κλειδώσει το κρίσιμο τμήμα αλλιώς αν βρει test = 1 (locked) εισέρχεται σε βρόχο επαναλαμβάνοντας τη λειτουργία (busy waiting). Είναι αντιληπτό πως επειδή οι μεταβλητές είναι κοινές, για μεγάλο αριθμό νημάτων θα υποφέρει από κίνηση στον δίαυλο μνήμης λόγω του cache coherence πρωτοκόλλου.

Ttas_lock (Test-and-Test-and-Set Lock)

To ttas_lock (Test-And-Test-And-Set Lock) είναι μια βελτίωση του απλού tas_lock. Πριν ένα νήμα επιχειρήσει να αποκτήσει το κλείδωμα μέσω της ατομικής λειτουργίας Test-And-Set, ελέγχει πρώτα την τρέχουσα κατάσταση του κλειδώματος διαβάζοντας την τιμή της μεταβλητής state. Αν το κλείδωμα φαίνεται ελεύθερο (state=0), τότε καλεί την ατομική λειτουργία Test-And-Set για να το αποκτήσει, όπως περιγράφηκε προηγουμένως. Αν το κλείδωμα είναι

κατειλημμένο (state=1), το νήμα επαναλαμβάνει τον έλεγχο της μεταβλητής state χωρίς να επιχειρεί συνεχώς να κλειδώσει την περιοχή, μειώνοντας έτσι την πρόσβαση στη μνήμη.

Array_lock

Το **array_lock** είναι ένας τύπος κλειδώματος που βασίζεται στη χρήση ενός πίνακα (array) για να επιτυγχάνει δίκαιη και τακτική πρόσβαση των νημάτων σε έναν κοινόχρηστο πόρο. Κάθε νήμα αποκτά έναν μοναδικό δείκτη θέσης στον πίνακα κατά την είσοδο στο κλείδωμα. Τα νήματα τοποθετούνται σε μια σειρά αναμονής (queue) μέσα στον πίνακα και αναμένουν ενεργά (busy waiting) μέχρι να έρθει η σειρά τους να αποκτήσουν το κλείδωμα. Το κλείδωμα περνάει στο επόμενο νήμα με βάση την κυκλική σειρά στον πίνακα, δηλαδή κάθε νήμα ξεκλειδώνει τον επόμενο του όταν βγαίνει από την κρίσιμη περιοχή.

CLh_lock (Craig, Landin and Hagersten Lock)

Το **CLH Lock** είναι ένας μηχανισμός κλειδώματος βασισμένος σε ουρά που χρησιμοποιείται για τον συγχρονισμό πολλαπλών νημάτων ώστε να διασφαλιστεί ότι μόνο ένα νήμα έχει πρόσβαση σε έναν κοινό πόρο τη φορά. Κάθε νήμα δημιουργεί έναν "κόμβο" (node) και τοποθετείται σε μια λογική ουρά, περιμένοντας τη σειρά του για να αποκτήσει το lock. Το νήμα που περιμένει παρακολουθεί την κατάσταση του προηγούμενου κόμβου αντί να ελέγχει συνεχώς την κοινή μνήμη, μέχρι το προηγούμενο νήμα να απελευθερώσει το lock. Με αυτόν τον τρόπο, κάθε νήμα γνωρίζει πότε μπορεί να αποκτήσει το lock, εξασφαλίζοντας αμοιβαίο αποκλεισμό και δικαιοσύνη στην πρόσβαση. Αυτή η διαδικασία μειώνει την ταυτόχρονη επιβάρυνση της μνήμης, αφού η αναμονή γίνεται τοπικά.

#pragma omp atomic

Είναι μια οδηγία του OpenMP που χρησιμοποιείται για να εξασφαλίσει την ατομικότητα (atomicity) μιας συγκεκριμένης πράξης σε ένα πρόγραμμα που εκτελείται σε περιβάλλον πολλαπλών νημάτων. Το #pragma omp atomic εφαρμόζεται σε μία μόνο πράξη, όπως ανάθεση ή ενημέρωση μεταβλητής. Χρησιμοποιεί υποστήριξη από το hardware (π.χ., atomic instructions) ή από τον μεταγλωττιστή για να διασφαλίσει ότι η πράξη εκτελείται με ατομικότητα, δηλαδή χωρίς να μπορεί να διακοπεί από άλλο νήμα.

#pragma omp critical

Όταν ένα νήμα εισέρχεται σε μια κρίσιμη περιοχή, όλα τα υπόλοιπα νήματα που προσπαθούν να μπουν στην ίδια κρίσιμη περιοχή περιμένουν μέχρι να βγει το πρώτο νήμα. Είναι ιδανικό για προστασία πολύπλοκων τμημάτων κώδικα που δεν μπορούν να καλυφθούν από την οδηγία #pragma omp atomic.

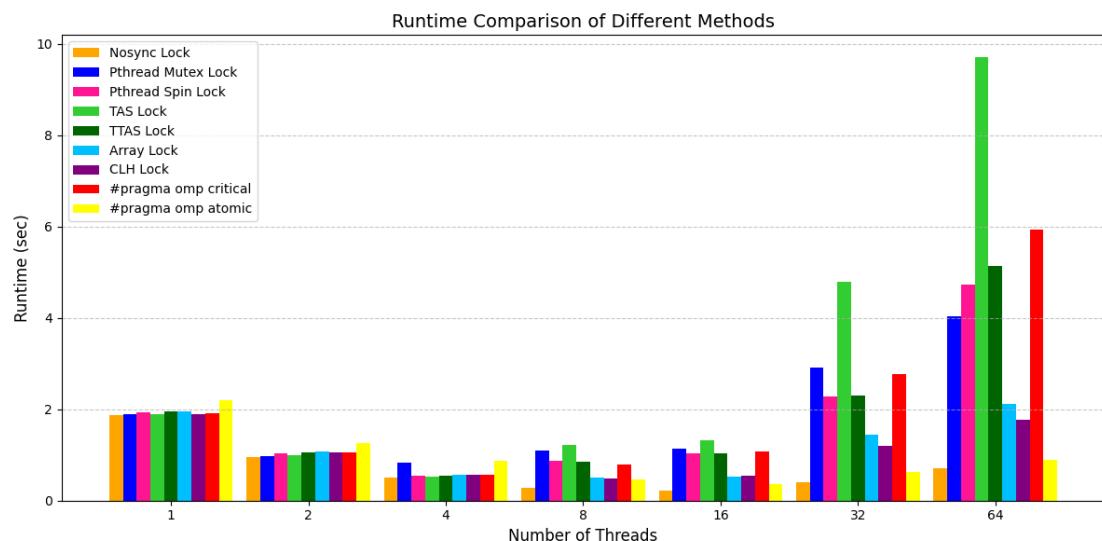
Η εκτέλεση θα γίνει χρησιμοποιώντας την υλοποίηση του αλγορίθμου kmeans για shared data με το εξής configuration:

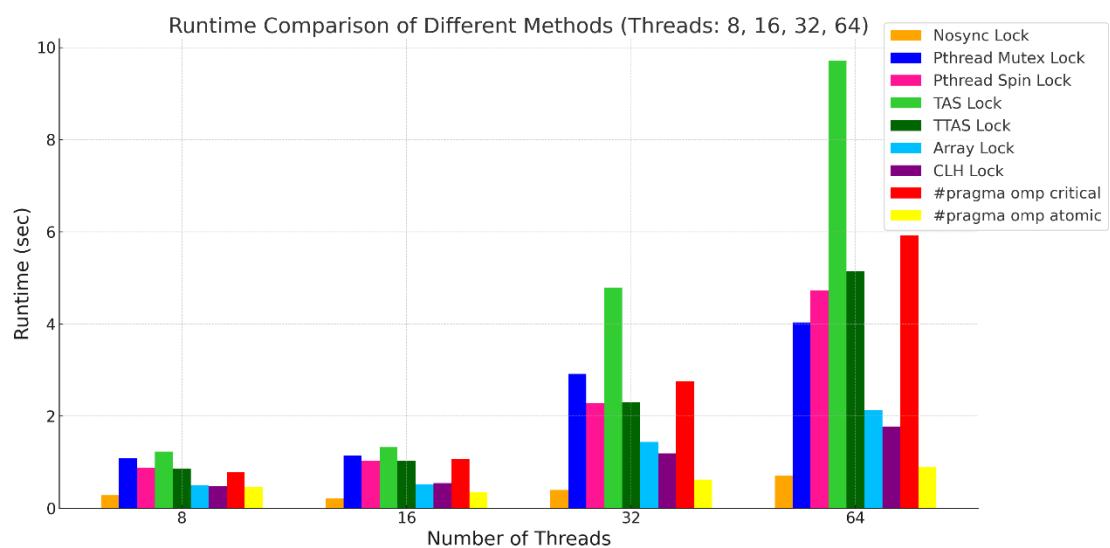
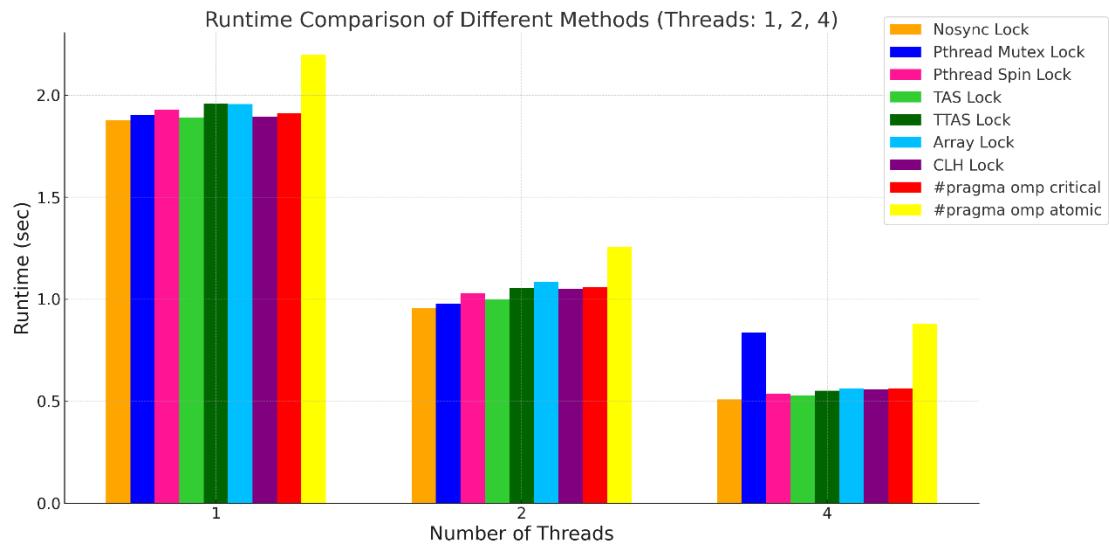
{Size, Coords, Clusters, Loops} = {32, 16, 32, 10}

Θα ληφθούν μετρήσεις για διαφορετικό αριθμό νημάτων:

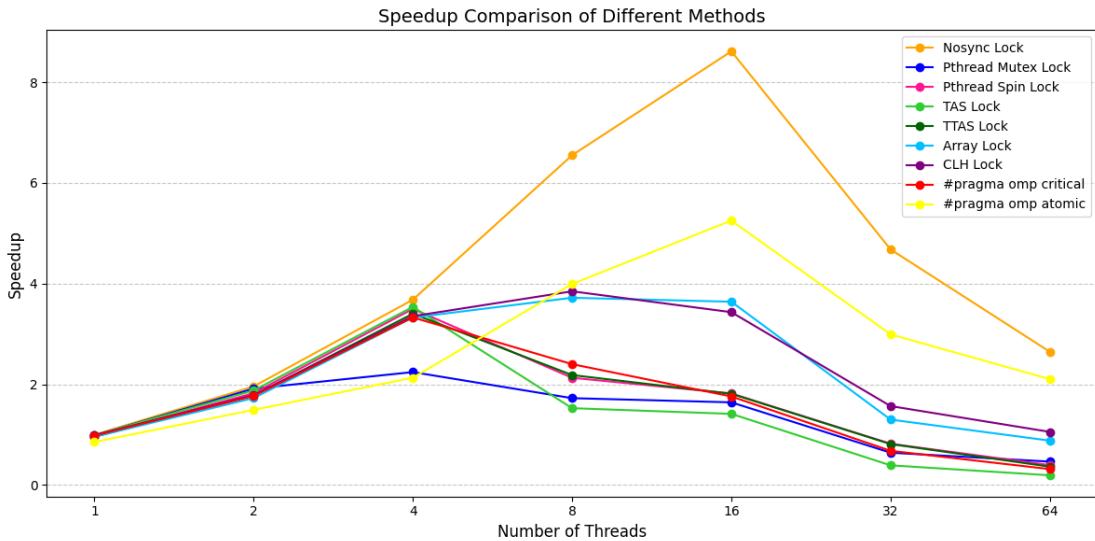
nthreads = {1, 2, 4, 8, 16, 32, 64}

Παραθέτουμε ένα ενιαίο διάγραμμα που παρουσιάζει τους χρόνους εκτέλεσης για όλες τις περιπτώσεις. Επιπλέον, για λόγους ευκρίνειας, παραθέτουμε και δύο ξεχωριστά διαγράμματα: το πρώτο για 1, 2 και 4 threads και το δεύτερο για 8, 16, 32 και 64 threads.





Ακολουθεί το διάγραμμα που παρουσιάζει το speedup για κάθε περίπτωση, επιτρέποντας τη σύγκριση της απόδοσης σε σχέση με την εκτέλεση δίχως συχρονισμό.



Σχολιασμός και συμπεράσματα:

Για μικρό αριθμό νημάτων (π.χ. 1, 2 ή 4 νήματα), η επίδοση των περισσότερων κλειδωμάτων είναι παρόμοια και πλησιάζει την απόδοση του **Nosync Lock**, καθώς οι συγκρούσεις στο κοινό κρίσιμο τμήμα είναι σπάνιες. Σε αυτήν την περίπτωση, το **#pragma omp atomic** έχει την χειρότερη επίδοση, κυρίως λόγω του overhead που προκαλούν οι ατομικές λειτουργίες αφού απαιτείται συγχρονισμός στο επίπεδο του επεξεργαστή και της μνήμης. Παρόλο που το **#pragma omp atomic** έχει σχεδιαστεί για την ασφαλή ατομική τροποποίηση της μνήμης, η συχνή αλληλεπίδραση με τη μνήμη και ο έλεγχος του αν το κρίσιμο τμήμα είναι ελεύθερο δημιουργούν επιπλέον καθυστέρηση. Αυτή η αναγκαία παρακολούθηση και το επιπλέον κόστος αναμονής το καθιστούν λιγότερο αποδοτικό σε σχέση με άλλες πιο απλές μεθόδους συγχρονισμού, οι οποίες δεν έχουν το ίδιο overhead και λειτουργούν πιο αποδοτικά σε μικρούς αριθμούς νημάτων.

Για μεγαλύτερο αριθμό νημάτων (π.χ. 16, 32, 64 νήματα), το **#pragma omp atomic** αρχίζει να παρουσιάζει σημαντική βελτίωση σε σχέση με τα υπόλοιπα κλειδώματα. Αυτό συμβαίνει επειδή, καθώς αυξάνεται ο αριθμός των νημάτων, η ανάγκη για συγχρονισμό εντείνεται, και το **#pragma omp atomic** μειώνει τον ανταγωνισμό για το κοινό κρίσιμο τμήμα επιτρέποντας ατομικές ενημερώσεις με λιγότερο overhead. Αντίθετα, τα απλά κλειδώματα όπως το **pthread mutex** και **spin lock** ή τα **TAS lock** και **TTAS lock** δεν αποδίδουν το ίδιο καλά, καθώς δημιουργείται αυξημένη κυκλοφορία δεδομένων μεταξύ των νημάτων και του συστήματος μνήμης, με αποτέλεσμα την αύξηση των καθυστερήσεων.

Ειδικότερα, τα **TAS Lock** και **TTAS Lock**, παρά τη θεωρητική τους αποδοτικότητα, αρχίζουν να αντιμετωπίζουν προβλήματα καθώς αυξάνεται ο αριθμός των

νημάτων. Αυτό συμβαίνει επειδή δημιουργούν μεγάλο ανταγωνισμό για τη μνήμη (*cache coherence*), και όσο περισσότερα τα νήματα, τόσο μεγαλύτερες οι καθυστερήσεις στην απόκτηση του κλειδώματος. Ωστόσο, το **TTAS Lock** αποδίδει καλύτερα από το **TAS Lock** σε μεγάλες κλίμακες, λόγω της "προληπτικής" λειτουργίας του, όπου η τιμή του κλειδώματος (*state*) ελέγχεται πρώτα, μειώνοντας έτσι τον αριθμό των αλληλεπιδράσεων με τη μνήμη και τις καθυστερήσεις. Αντίθετα, στο **TAS Lock** τα νήματα, μη γνωρίζοντας την τρέχουσα κατάσταση του κλειδώματος, συνεχώς προσπαθούν να κλειδώσουν την κρίσιμη περιοχή, αυξάνοντας την πίεση στη μνήμη και τον χρόνο καθυστέρησης, ιδίως σε περιβάλλοντα με πολλά νήματα.

Όσον αφορά το **pthread_mutex_lock**, η απόδοση επιδεινώνεται όσο αυξάνεται ο αριθμός των νημάτων, κυρίως λόγω της επιβάρυνσης του λειτουργικού συστήματος, το οποίο πρέπει να διαχειριστεί την αναμονή και την αφύπνιση των νημάτων. Κάθε φορά που το *lock* είναι κατειλημμένο, τα νήματα μπαίνουν σε κατάσταση αναμονής και το λειτουργικό σύστημα είναι υπεύθυνο να τα "ξυπνήσει" όταν το *lock* γίνει διαθέσιμο, γεγονός που δημιουργεί σημαντικό overhead από την διαχείριση αυτής της διαδικασίας.

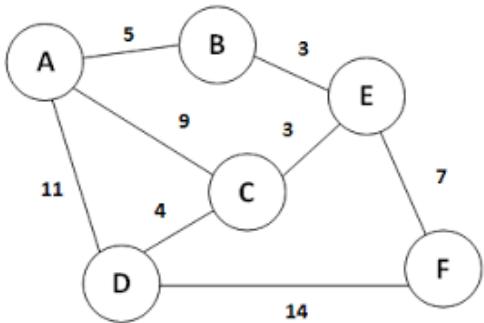
Ακόμη, στο **pthread_spin_lock** η επιβάρυνση προέρχεται από το *busy waiting*, όπου τα νήματα "σπινάρουν" συνεχώς διεκδικώντας το *lock*. Καθώς αυξάνεται ο αριθμός των νημάτων, η CPU καταναλώνεται όλο και περισσότερο από νήματα που περιμένουν ενεργά, χωρίς να παράγουν χρήσιμο έργο, κάτι που οδηγεί σε σημαντική πτώση της απόδοσης λόγω υπερβολικής κατανάλωσης των πόρων της CPU.

Τα πιο εξελιγμένα κλειδώματα, όπως το **Array Lock** και το **CLH Lock**, αποδίδουν καλύτερα σε τέτοιες καταστάσεις, διότι χρησιμοποιούν ειδικές δομές δεδομένων (πίνακες ή λίστες) που επιτρέπουν στα νήματα να εργάζονται πιο ανεξάρτητα, μειώνοντας τις συγκρούσεις στη μνήμη και εξασφαλίζοντας μεγαλύτερη αποδοτικότητα.

Τέλος, το **#pragma omp critical**, αν και αποδοτικό για μικρό αριθμό νημάτων, υπολείπεται καθώς αυξάνονται τα νήματα. Αυτό συμβαίνει επειδή απαιτεί από τα threads να περιμένουν τη σειρά τους για να εκτελέσουν το κρίσιμο τμήμα, προκαλώντας καθυστερήσεις καθώς ο ανταγωνισμός αυξάνεται. Σε αντίθεση με το **#pragma omp atomic**, το οποίο επιτρέπει ατομικές ενημερώσεις χωρίς αποκλεισμό του κρίσιμου τμήματος, το **#pragma omp critical** δημιουργεί μεγαλύτερο overhead και δεν κλιμακώνει καλά για μεγάλο αριθμό νημάτων.

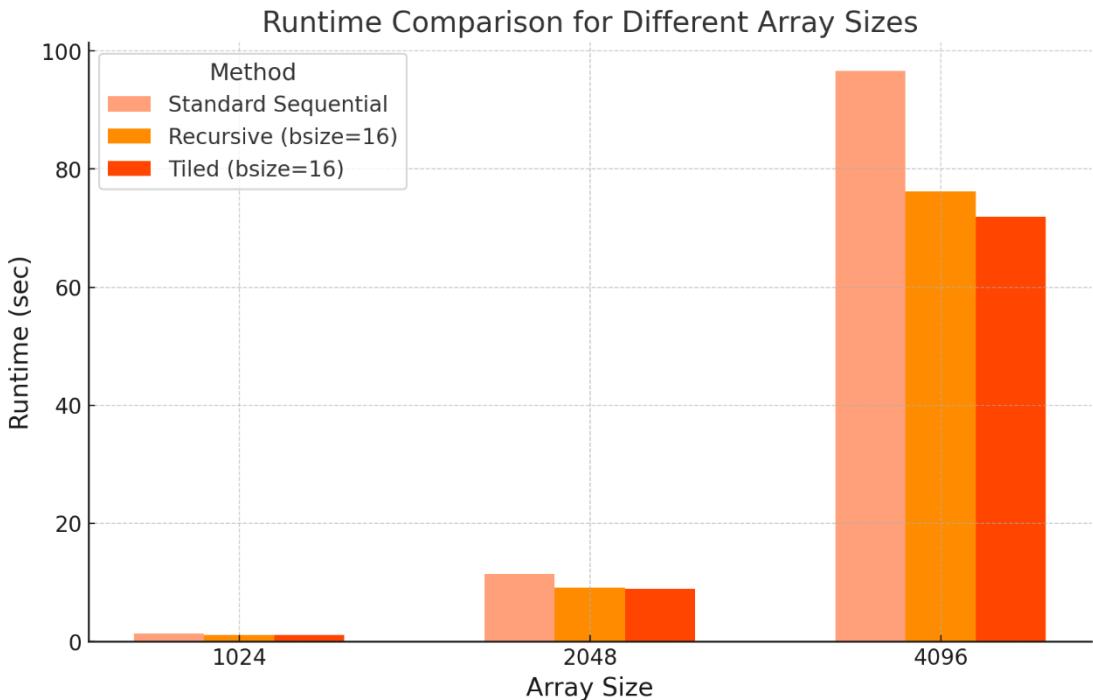
Από το διάγραμμα speedup γίνεται περισσότερο εμφανής η παραπάνω ανάλυση και η ανάδειξη του αποδοτικότερου τύπου κλειδώματος.

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

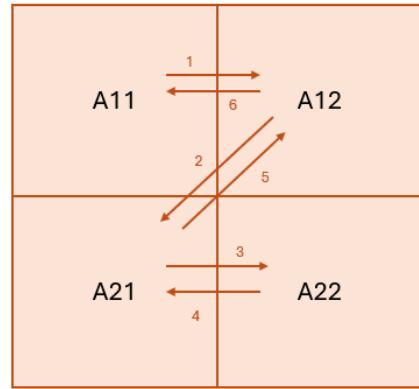


Ο αλγόριθμος *Floyd-Warshall* είναι μια κλασική δυναμική μέθοδος προγραμματισμού που χρησιμοποιείται για την εύρεση των συντομότερων διαδρομών ανάμεσα σε όλα τα ζεύγη κόμβων σε γράφο όπου κάθε ακμή έχει ένα βάρος. Ο αλγόριθμος λειτουργεί επαναληπτικά, ανανεώνοντας τις αποστάσεις μεταξύ κόμβων μέσω ενός ενδιάμεσου κόμβου κάθε φορά, και χρησιμοποιεί έναν πίνακα για την αποθήκευση των ελαχίστων αποστάσεων. Με πολυπλοκότητα $O(n^3)$, ο *Floyd-Warshall* είναι αποδοτικός για γραφήματα μεσαίου μεγέθους και υποστηρίζει τόσο θετικά όσο και αρνητικά βάρη, αρκεί να μην υπάρχουν αρνητικοί κύκλοι.

Πέρα από την κλασική έκδοση του αλγορίθμου, έχουν αναπτυχθεί δύο επιπλέον παραλλαγές: η αναδρομική (recursive) και η tiled. Από τη σύγκριση των σειριακών χρόνων εκτέλεσης για διάφορα μεγέθη πινάκων, διαπιστώνεται ότι η tiled έκδοση είναι η ταχύτερη από τις τρεις, όπως καταδεικνύουν τα παρακάτω διαγράμματα.



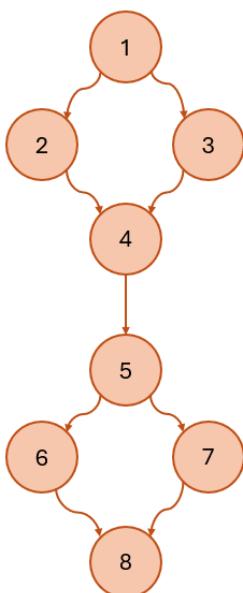
Για να παραλληλοποιήσουμε την αναδρομική υλοποίηση του αλγορίθμου *Floyd-Warshall*, χρησιμοποιούμε *tasks* του OpenMP, τα οποία είναι ιδανικά για την παραλληλοποίηση αναδρομικών συναρτήσεων. Αρχικά, εξετάζουμε την αναδρομική δομή του αλγορίθμου για να εντοπίσουμε τις εξαρτήσεις μεταξύ των αναδρομικών κλήσεων, εξασφαλίζοντας έτσι τη σωστή ακολουθία εκτέλεσης.



Η αναδρομή διασπά τον συνολικό πίνακα γειτνίασης σε τέσσερις υποπίνακες σε κάθε βήμα, και εκτελούνται συνολικά οκτώ αναδρομικές κλήσεις για την επεξεργασία των υποπινάκων. Η διαδικασία συνεχίζεται έως ότου το μέγεθος των υποπινάκων φτάσει ένα καθορισμένο όριο (*b_size*), στο οποίο σημείο εφαρμόζεται ο κλασικός αλγόριθμος *Floyd-Warshall* για την επεξεργασία αυτών των μικρότερων υποπινάκων.

Με την προσέγγιση αυτή, επιτυγχάνουμε την ορθότητα της αναδρομής, αφού εξετάζουμε εξαντλητικά όλες τις δυνατές περιπτώσεις και επικαιροποιήσεις των αποστάσεων στον πίνακα γειτνίασης. Η χρήση των OpenMP *tasks* επιτρέπει σε κάθε αναδρομικό βήμα να εκτελείται ανεξάρτητα από τα υπόλοιπα, επιτυγχάνοντας αποδοτικότερη παράλληλη επεξεργασία και βελτιστοποιώντας την τοπικότητα της μνήμης.

Εξετάζοντας λοιπόν τις οκτώ αναδρομικές κλήσεις συμπεραίνουμε τον παρακάτω γράφο εξαρτήσεων:



Όπου οι αναδρομικές κλήσεις είναι:

1) FWR (A11, B11, C11)

2) FWR (A12, B11, C12)

3) FWR (A21, B21, C11)

parallel

4) FWR (A22, B21, C12)

5) FWR (A22, B22, C22)

6) FWR (A21, B22, C21)

parallel

7) FWR (A12, B12, C22)

8) FWR (A11, B12, C21)

Παρατηρούμε ότι υπάρχει περιορισμένη δυνατότητα παραλληλοποίησης λόγω των εξαρτήσεων μεταξύ των tasks, με το μέγιστο speedup να υπολογίζεται θεωρητικά ως 8/6.

Θα παρουσιάσουμε μετρήσεις για διαφορετικά μεγέθη πινάκων γειτνίασης και διαφορετικό αριθμό threads που εκτελούν το παράλληλο πρόγραμμα. Η επίδοση επηρεάζεται τόσο από το μέγεθος του πίνακα όσο και από τον αριθμό των threads, αλλά σημαντικό ρόλο παίζει επίσης και η επιλογή του ορίου αναδρομής, *b_size*. Επομένως, θα παραμετροποιήσουμε και αυτήν την τιμή για να εντοπίσουμε το βέλτιστο configuration.

Συγκεκριμένα, θα δοκιμάσουμε τις παρακάτω παραμετροποιήσεις:

- **Nthreads: {1, 2, 4, 8, 16, 32, 64}**
- **Array_size: {1024x1024, 2048x2048, 4096x4096}**
- **B_size: {16, 32, 64, 128, 256}**

Με αυτές τις παραμετροποιήσεις, θα αξιολογήσουμε την επίδραση κάθε συνδυασμού στην επίδοση του αλγορίθμου και θα προσδιορίσουμε το βέλτιστο *b_size* για κάθε συνδυασμό μεγέθους πίνακα και αριθμού threads.

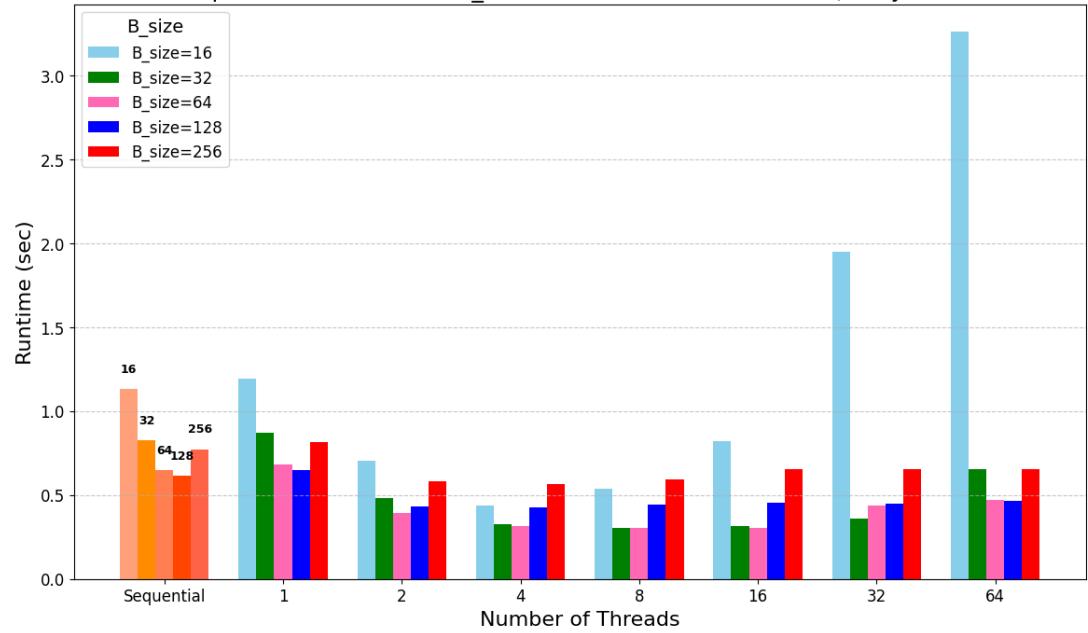
Παρουσιάζουμε για αρχή του χρόνους εκτέλεσης για την σειριακή εκτέλεση του αλγορίθμου Floyd-Warshall, για να τις έχουμε ως τιμές αναφοράς:

Παρατηρούμε πως στο σειριακό πρόγραμμα έχουμε πολύ κακή κλιμάκωση και όσο αυξάνουμε το μέγεθος των πινάκων αυξάνεται σημαντικά ο χρόνος εκτέλεσης.

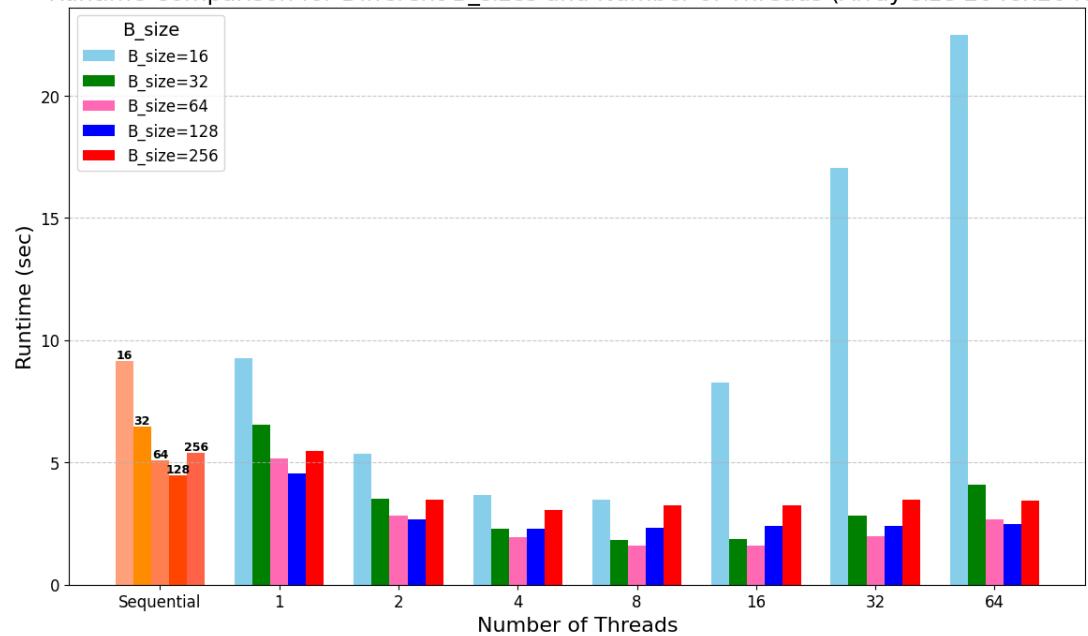
Θα παρατηρήσουμε λοιπόν την επίδραση τόσο της αναδρομικής υλοποίησης όσο και του παραλληλισμού.

Για κάθε array size τώρα θα πάρουμε το διάγραμμα για τους χρόνους εκτέλεσης:

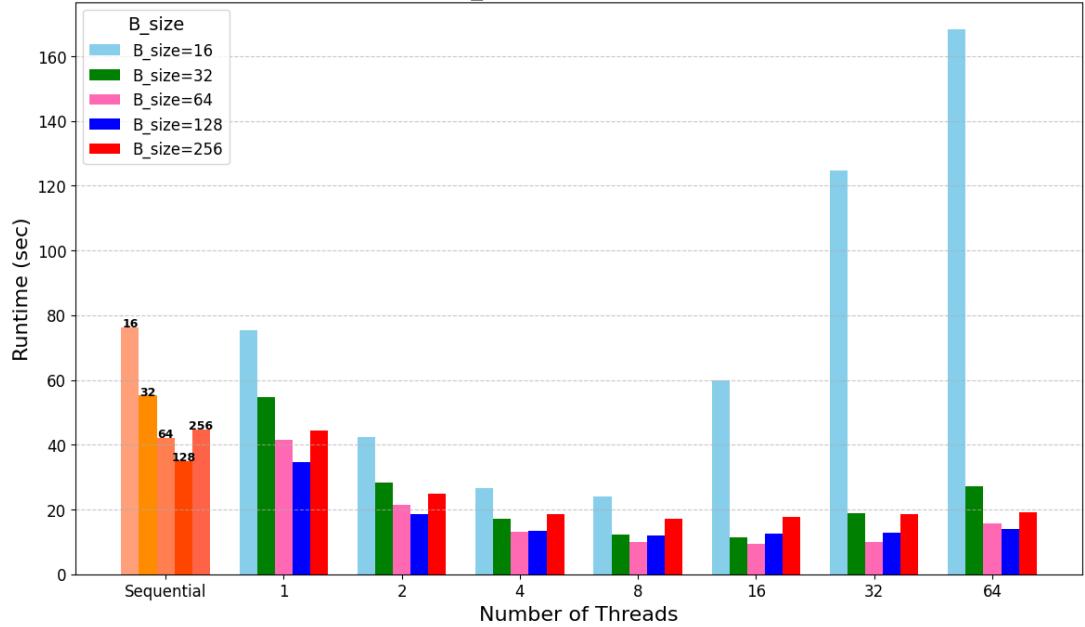
Runtime Comparison for Different B_sizes and Number of Threads (Array size 1024x1024)



Runtime Comparison for Different B_sizes and Number of Threads (Array size 2048x2048)



Runtime Comparison for Different B_sizes and Number of Threads (Array size 4096x4096)



Σχολιασμός και συμπεράσματα:

Συγκρίνοντας τους χρόνους εκτέλεσης της κλασικής και της αναδρομικής υλοποίησης του *Floyd-Warshall*, διαπιστώνουμε ότι η αναδρομική έκδοση είναι σταθερά ταχύτερη, ανεξάρτητα από το *block size*. Αυτό οφείλεται στο ότι η αναδρομική προσέγγιση εκμεταλλεύεται την τοπικότητα των δεδομένων, μειώνοντας το κόστος πρόσβασης στη μνήμη. Μέσω της διαίρεσης του πίνακα σε μικρότερα υποτμήματα και της εκτέλεσης του αλγορίθμου σε αυτά, η αναδρομική υλοποίηση επιτρέπει την επεξεργασία να γίνεται κυρίως στην *cache*, περιορίζοντας τις αργές προσπελάσεις στην κύρια μνήμη και βελτιώνοντας τον συνολικό χρόνο εκτέλεσης σε σχέση με την κλασική μέθοδο.

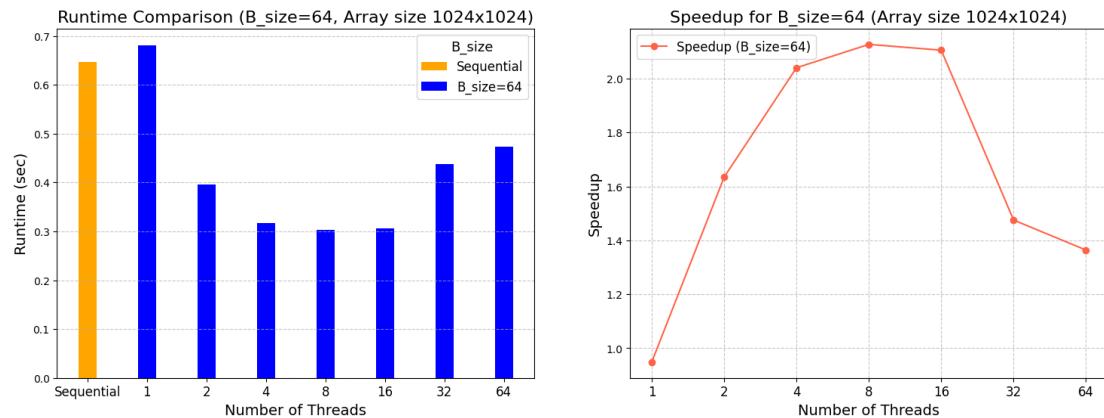
Μια αξιοσημείωτη παρατήρηση αφορά την κακή απόδοση που παρατηρείται σε όλα τα μεγέθη πινάκων για μικρό *block size* (*b_size=16*) και υψηλό αριθμό *threads* (*nthreads=32* και *nthreads=64*). Αυτό το φαινόμενο οφείλεται στο γεγονός ότι το *overhead* της δημιουργίας *tasks* και του συγχρονισμού των *threads* δεν αντισταθμίζεται από το μικρό μέγεθος των υποπινάκων, ενώ το σπάσιμο του προβλήματος σε πολύ μικρούς υποπίνακες επιβαρύνει τον αλγόριθμο με πολλαπλές και συχνές προσπελάσεις στη μνήμη.

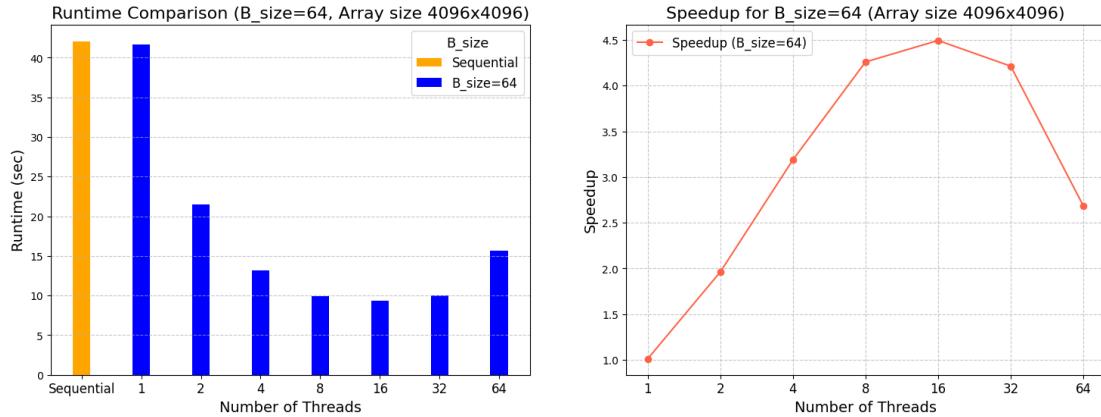
To *block size* (*b_size*) παίζει καθοριστικό ρόλο στην απόδοση του παράλληλου προγράμματος. Τόσο οι πολύ μικρές όσο και οι πολύ μεγάλες τιμές του *b_size* δεν αποδίδουν βέλτιστα. Σε μικρές τιμές (π.χ., *b_size=16*), τα *threads* διαχειρίζονται υποπίνακες που οδηγούν σε αυξημένα *cache misses* και αυξημένο *overhead* λόγω συχνών προσβάσεων στη μνήμη, δημιουργώντας συμφόρηση. Αντίστοιχα, για

μεγάλες τιμές (π.χ., $b_size=256$), τα δεδομένα που πρέπει να επεξεργαστούν τα $threads$ δεν χωρούν στην $cache$, με αποτέλεσμα να παρατηρούνται και πάλι $cache$ misses.

Παρατηρούμε ότι για όλα τα μεγέθη πίνακα, η βέλτιστη απόδοση επιτυγχάνεται για $block$ sizes 64 και 128. Στην περίπτωση του πίνακα μεγέθους $1024x1024$, παρατηρείται καλή απόδοση και για $b_size=32$, κάτι που είναι αναμενόμενο λόγω του μικρότερου μεγέθους του προβλήματος.

Συνοψίζοντας, τα παραπάνω διαγράμματα μας επιτρέπουν να εξαγάγουμε συμπεράσματα για την ιδανική επιλογή του αριθμού των $threads$. Για να φανούν καλύτερα τα αποτελέσματα, θα επαναλάβουμε τα διαγράμματα εστιάζοντας στις βέλτιστες τιμές του $block$ size που παρατηρήσαμε παραπάνω.





Από τα παραπάνω διαγράμματα, παρατηρούμε ότι κανένα configuration δεν καταφέρνει να επιτύχει σημαντικό speedup. Αυτό το αποτέλεσμα είναι αναμενόμενο, όπως φάνηκε και από τη δομή του task graph που παρουσιάστηκε προηγουμένως, καθώς οι δυνατότητες παραλληλοποίησης του αλγορίθμου είναι περιορισμένες. Η αναδρομική διαδικασία μπορεί να διαχωριστεί σε 8 tasks, ωστόσο υπάρχουν μόνο δύο σημεία όπου είναι δυνατός ο πραγματικός παραλληλισμός. Αυτή η απαίτηση για σειριοποίηση επιβάλλει ένα ανώτατο όριο στο speedup, όπως περιγράφεται και από τον νόμο του Amdahl.

Όσον αφορά την επιλογή του κατάλληλου αριθμού threads για την εκτέλεση, παρατηρούμε ότι η κλιμάκωση είναι σχετικά καλή μέχρι τα 8 ή 16 threads. Πέρα από αυτό το όριο, η αύξηση του αριθμού των threads δεν βελτιώνει την απόδοση και μπορεί ακόμη και να την επιδεινώσει. Αυτό οφείλεται στο ότι ο αλγόριθμος είναι *memory-bound*, με αποτέλεσμα η απόδοση να περιορίζεται από την αυξημένη ανάγκη για πρόσβαση στη μνήμη. Με άλλα λόγια, η επιπλέον υπολογιστική ισχύς δεν βελτιώνει την απόδοση μετά από ένα σημείο, καθώς δημιουργείται συμφόρηση στον δίαυλο μνήμης, περιορίζοντας την αποδοτικότητα της παραλληλοποίησης.

Σημειώνεται πως οι καλύτεροι χρόνοι εκτέλεσης που πήραμε για κάθε μέγεθος εισόδου είναι:

- 1024×1024 : 0.3037 sec για block size 64 και εκτέλεση με 8 threads
- 2048×2048 : 1.6078 sec για block size 64 και εκτέλεση με 8 threads
- 4096×4096 : 9.3617 sec για block size 64 και εκτέλεση με 16 threads

Τέλος, μεγαλύτερο speedup επιτυγχάνεται για μεγαλύτερο μέγεθος πίνακα, με το μέγιστο να φτάνει περίπου το 4.5 για μέγεθος 4096. Αυτό είναι μεγαλύτερο από το θεωρητικά υπολογισμένο (~ 1.33)

Η παρατηρούμενη απόκλιση μεταξύ του θεωρητικού speedup (8/6) και του πραγματικού, που είναι σημαντικά υψηλότερο, οφείλεται σε διάφορους

παράγοντες. Πέρα από τις απλοποιήσεις του θεωρητικού μοντέλου, η πραγματική υλοποίηση αξιοποιεί βελτιώσεις όπως η δυναμική κατανομή των tasks μέσω OpenMP tasking, η βέλτιστη εκμετάλλευση της ιεραρχίας μνήμης και η μείωση του overhead επικοινωνίας. Επιπλέον, η χρήση του if(0) στις OpenMP tasks παροτρύνει την άμεση εκτέλεσή τους, διασφαλίζοντας ότι προγραμματίζονται δυναμικά στο task queue και εκτελούνται αποδοτικά από διαθέσιμα νήματα. Αυτό οδηγεί σε βέλτιστη κατανομή του φόρτου, μειωμένο load imbalance και καλύτερη αξιοποίηση του memory bandwidth, επιτρέποντας μεγαλύτερη επικαλυπτόμενη εκτέλεση εργασιών. Ως αποτέλεσμα, η πραγματική ταχύτητα εκτέλεσης είναι σημαντικά βελτιωμένη σε σχέση με τις θεωρητικές εκτιμήσεις. Η απόκλιση αυτή ενδεχομένως επηρεάζεται από την αναδρομική φύση του αλγορίθμου.

Κώδικας που υλοποιήθηκε:

Ο κώδικας που υλοποιήσαμε για την δημιουργία και διαχείριση των tasks είναι ο παρακάτω:

```
#pragma omp parallel
{
    #pragma omp single //one thread will create the tasks
    FW_SR(A,0,0, A,0,0,A,0,0,N,B);
}

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

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

    #pragma omp task if(0)
    {
        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);

    #pragma omp task if(0)
    {
        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);
}
```

Χρησιμοποιούμε τους πίνακες A, B, και C ως shared για να αποφύγουμε την επιπλέον κατανάλωση μνήμης και τις περιττές αντιγραφές δεδομένων. Εφόσον τα threads εργάζονται σε ανεξάρτητα τμήματα των πινάκων, δεν προκύπτουν *race conditions*, καθιστώντας την κοινή χρήση των πινάκων αποδοτική και βέλτιστη. Έτσι, κάθε task έχει άμεση πρόσβαση στα δεδομένα που χρειάζεται χωρίς να απαιτείται η δημιουργία νέων τοπικών αντιγράφων, κάτι που θα επιβάρυνε σημαντικά τη μνήμη.

Επιπλέον, η χρήση της οδηγίας `if(0)` στα tasks επιτρέπει την επιλεκτική παράλληλη εκτέλεση στην αναδρομική υλοποίηση του αλγορίθμου. Με το `if(0)`, συγκεκριμένα tasks εκτελούνται άμεσα στο τρέχον thread, αντί να δημιουργηθεί ανεξάρτητο task για αυτά. Αυτό είναι χρήσιμο για τα μικρά υποπροβλήματα, όπου το κόστος δημιουργίας και διαχείρισης νέου task (overhead) υπερβαίνει τα οφέλη της παράλληλης εκτέλεσης. Έτσι, ο παραλληλισμός εφαρμόζεται μόνο στα μεγαλύτερα υποπροβλήματα, όπου είναι ουσιαστικά επωφελής, ενώ τα μικρότερα εκτελούνται σειριακά, εξοικονομώντας χρόνο και βελτιστοποιώντας τη συνολική απόδοση του αλγορίθμου.

Παραλληλοποίηση tiled έκδοσης του αλγορίθμου Floyd-Warshall

Σε αυτή την ενότητα, θα παραλληλοποιήσουμε μια διαφορετική υλοποίηση του αλγορίθμου Floyd-Warshall, βασισμένη στην τεχνική της διαίρεσης του πίνακα σε τμήματα (tiles). Αντί να αναλύουμε το πρόβλημα αναδρομικά σε μικρότερα υποπροβλήματα, όπως σε άλλες μεθόδους, εδώ διασπάμε τον πίνακα εξαρχής σε σταθερού μεγέθους τμήματα. Κάθε tile αντιπροσωπεύει ένα υποσύνολο του πίνακα, το οποίο υπολογίζεται ξεχωριστά.

Η επεξεργασία των tiles γίνεται με τρόπο που να σέβεται τις εξαρτήσεις δεδομένων, διασφαλίζοντας ότι κάθε υπολογισμός βασίζεται σε ενημερωμένα δεδομένα από προηγούμενα tiles.

Θα πάρουμε τα ίδια διαγράμματα που παραθέσαμε πριν και θα συγκρίνουμε τα αποτελέσματά μας με εκείνα της προηγούμενης υλοποίησης βγάζοντας συμπεράσματα για την επίδοση των δύο μεθόδων.

Κώδικας που υλοποιήθηκε:

```

#pragma omp parallel
{
    #pragma omp for schedule(static)
    for (i = 0; i < N; i += B) {
        if (i != k) {
            FW(A, k, i, k, B); // Process column
            FW(A, k, k, i, B); // Process row
        }
    }

    // Phase 3: Process remaining blocks
    #pragma omp for collapse(2) schedule(static)
    for (i = 0; i < N; i += B) {
        for (j = 0; j < N; j += B) {
            if (i != k && j != k) {
                FW(A, k, i, j, B);
            }
        }
    }
}

gettimeofday(&t2, 0);

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

// Free memory
for (i = 0; i < N; i++) {
    free(A[i]);
}
free(A);

return 0;
}

inline int min(int a, int b)
{
    return (a <= b) ? a : b;
}

```

1	2	2	2
2	3	3	3
2	3	3	3
2	3	3	3

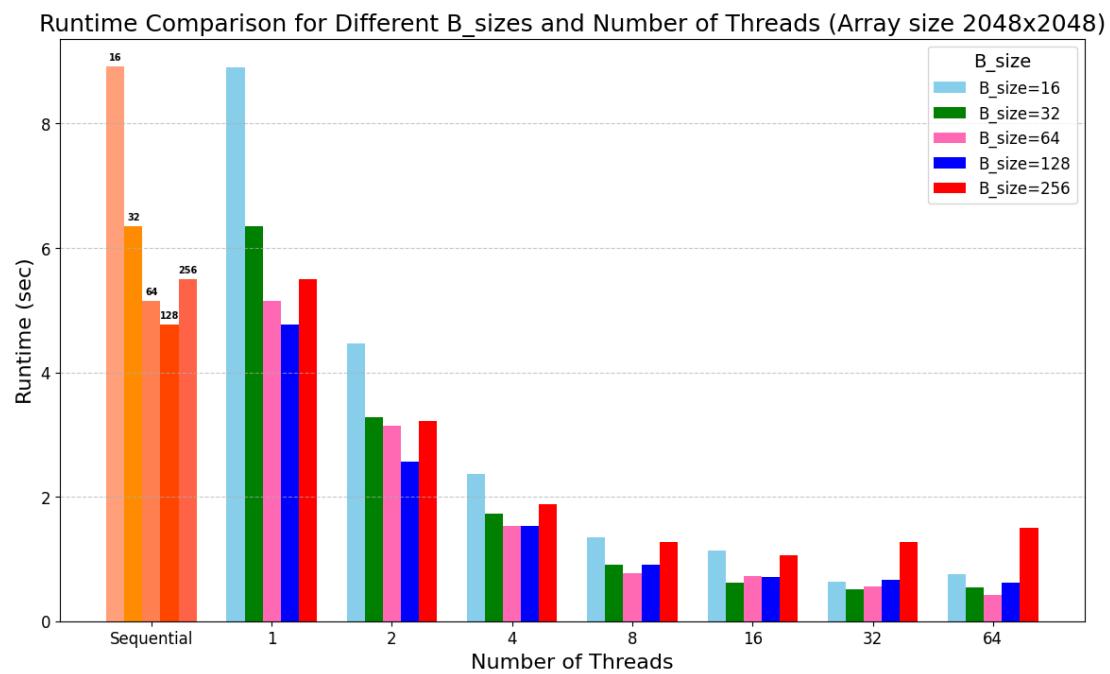
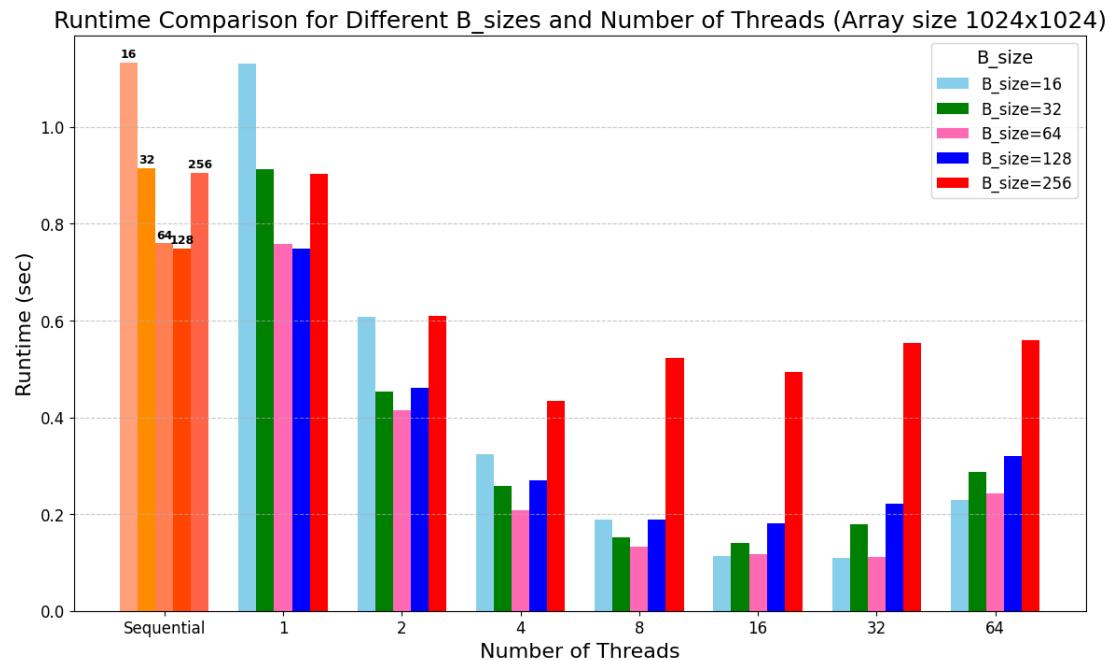
H tiled υλοποίηση του Floyd-Warshall έχει νόημα να παραλληλοποιηθεί στις φάσεις 2 και 3, όπου στη 2 υπολογίζονται τα μπλοκ της στήλης (i,k) και της γραμμής (k,j) και στην 3 τα υπόλοιπα μπλοκ (i,j) εκτός των προηγουμένων και του διαγώνιου που υπολογίζεται στην φάση 1.

Για την παραλληλοποίηση στην φάση 2 δημιουργούμε μία παράλληλη περιοχή στην οποία χρησιμοποιούμε την εντολή `for schedule(static)`. Η στατική κατανομή (static) μοιράζει εξίσου τις επαναλήψεις σε όλα τα threads, μειώνοντας το overhead του runtime scheduler (το κόστος συγχρονισμού που απαιτείται για τη διαχείριση των threads). Τα μπλοκ στηλών και γραμμών είναι ανεξάρτητα και μπορούν να εκτελεστούν ταυτόχρονα.

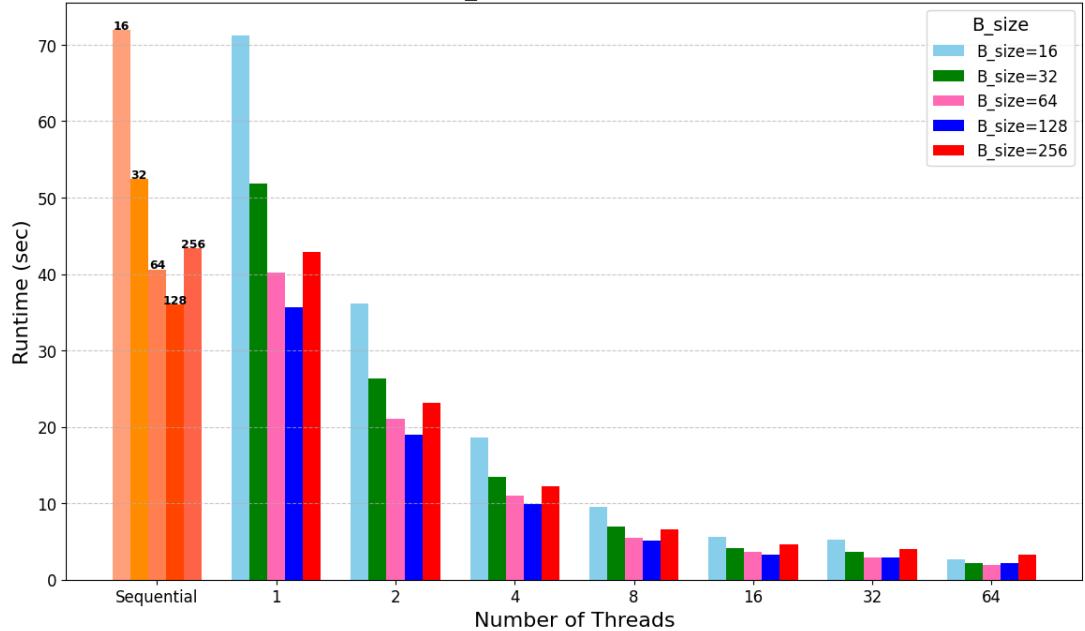
Για την φάση 3, εκτός της εντολής `schedule(static)` προσθέτουμε και το `collapse(2)`, του οποίου η λειτουργία είναι να ενοποιήσει τα δύο ακόλουθα loops σε ένα με αποτέλεσμα το OpenMP να εκτελεί τις επαναλήψεις και των δύο σε ένα

ενιαίο σύνολο. Αυτό το προτιμούμε γιατί για μεγάλες τιμές του N θα παρατηρήσουμε μεγάλο optimization το οποίο οφείλεται σε καλύτερη κατανομή εργασίας λόγω του collapse(2).

Από την εκτέλεση του κώδικα για διάφορα block sizes προκύπτουν τα παρακάτω αποτελέσματα:



Runtime Comparison for Different B_sizes and Number of Threads (Array size 4096x4096)



Σχολιασμός και συμπεράσματα:

Όπως μπορούμε να παρατηρήσουμε ο κώδικας αυτός αντιμετωπίζει πολύ αποτελεσματικά τους πίνακες μεγάλου μεγέθους σε σύγκριση με τους πιο μικρούς. Εμφανίζεται καλύτερη κλιμάκωση με το μέγεθος της εισόδου. Ο αλγόριθμος επωφελείται από την κατανομή σε blocks, διατηρώντας τον χρόνο εκτέλεσης χαμηλότερο ακόμα και για $N=4096$.

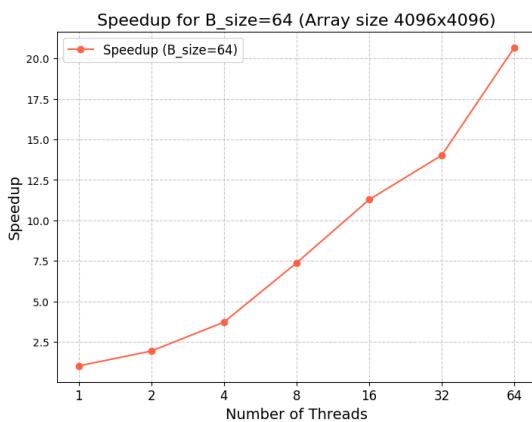
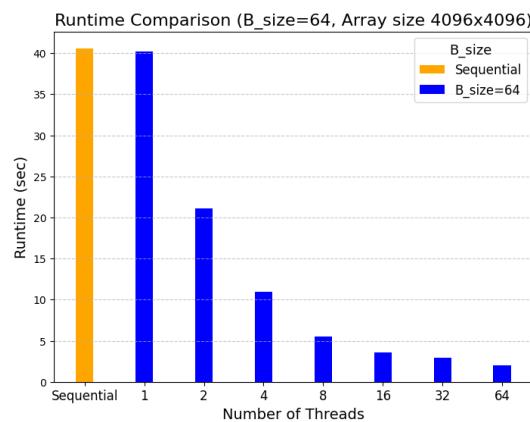
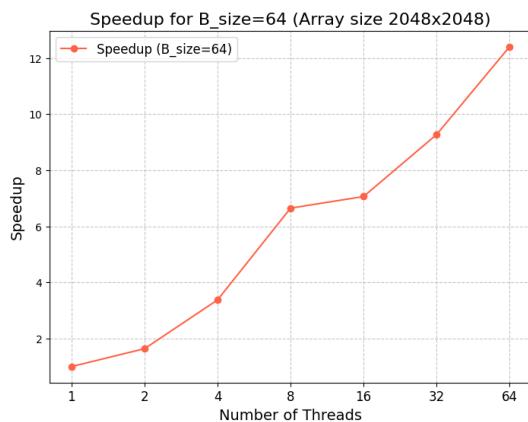
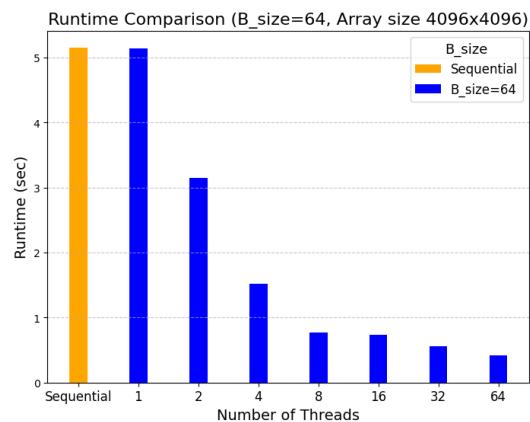
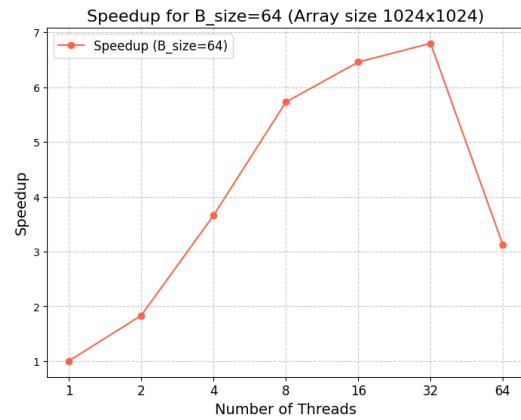
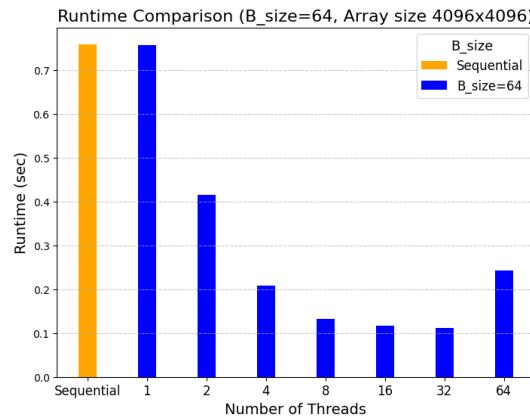
Σε αυτή τη φάση να σημειωθεί ότι το βέλτιστο block size για τον tiled κώδικα που παρατηρείται φαίνεται να είναι το $b_size=64$. Φαίνεται να αξιοποιεί πλήρως την cache και μειώνει τις καθυστερήσεις μνήμης, με τους χρόνους να μειώνονται δραστικά ακόμα και με πολλά threads.

Για μικρότερα block sizes παρουσιάζονται βελτιωμένοι χρόνοι, ειδικά με λίγα threads (π.χ., 1, 2, 4). Η χρήση μικρών blocks αξιοποιεί την τοπικότητα. Παρόλα αυτά, για υψηλά threads (>16), το μικρό μέγεθος των blocks δημιουργεί overhead λόγω συγχρονισμού.

Για μεγαλύτερα block sizes η απόδοση μειώνεται (π.χ., block size 256), επειδή τα blocks είναι πολύ μεγάλα για να χωρέσουν στην cache, οδηγώντας σε cache misses.

Επίσης, παρατηρούμε ότι αρχικά η μείωση του χρόνου είναι πιο απότομη μέχρι και τα 4 threads, ωστόσο και για τα μεγαλύτερα έχουμε, φθίνουσα μεν, αλλά ικανοποιητική κλιμάκωση.

Στα παρακάτω διαγράμματα μπορούμε να μελετήσουμε το speedup για το βέλτιστο block size (64):



Για τον πίνακα 4096, παρατηρείται συνεχής αύξηση του speedup καθώς αυξάνονται τα νήματα, φτάνοντας σε πολύ υψηλές τιμές για 64 νήματα. Λιγότερα έντονα αλλά όμοια αποτελέσματα έχουμε και για τον πίνακα μεγέθους 2048. Ωστόσο, στον μικρότερο πίνακα 1024 παρατηρούμε ότι το speedup παρουσιάζει σημαντική αύξηση μέχρι τα 8 threads, μέχρι τα 32 έχει μικρές αυξήσεις ενώ για τα 64 έχουμε σημαντική πτώση του. Αυτή η πτώση πιθανότατα οφείλεται σε υπερβολικό overhead από τη διαχείριση των νημάτων σε σχέση με το μικρό μέγεθος του πίνακα, που περιορίζει την αποδοτικότητα.

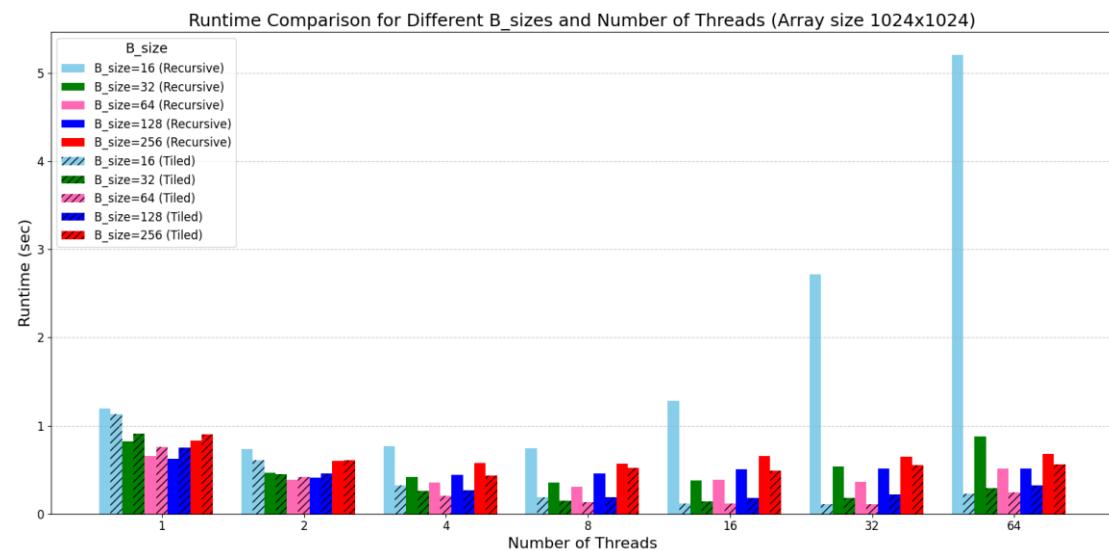
Εδώ μπορούμε να επιβεβαιώσουμε ότι η tiled υλοποίηση λειτουργεί αποδοτικά για πίνακες μεγάλους μεγέθους, ενώ σε μικρότερα μεγέθη για μεγάλο αριθμό threads κωλυσιεργεί λόγω του overhead που προκαλεί η διαχείριση των νημάτων.

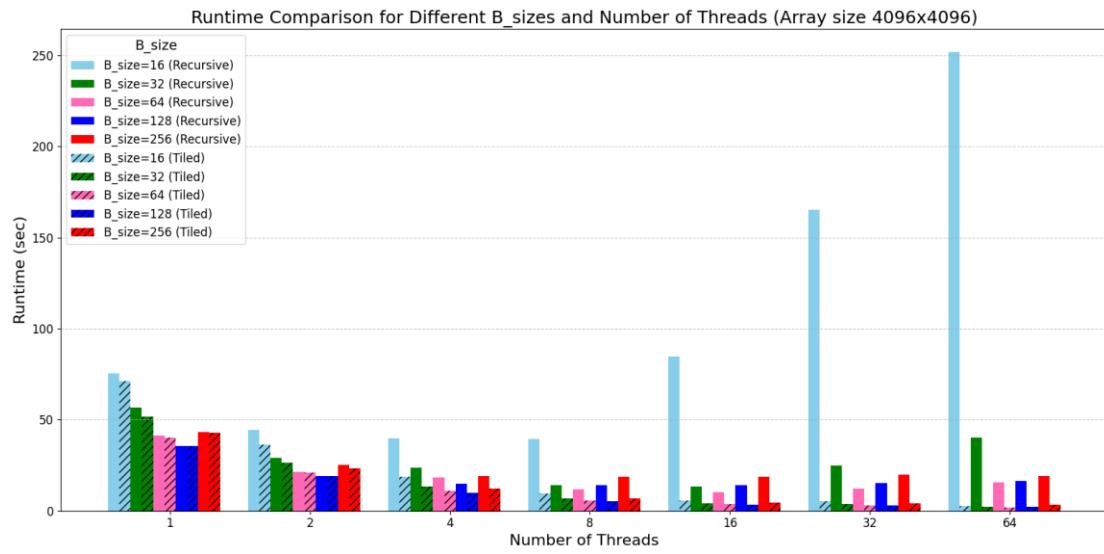
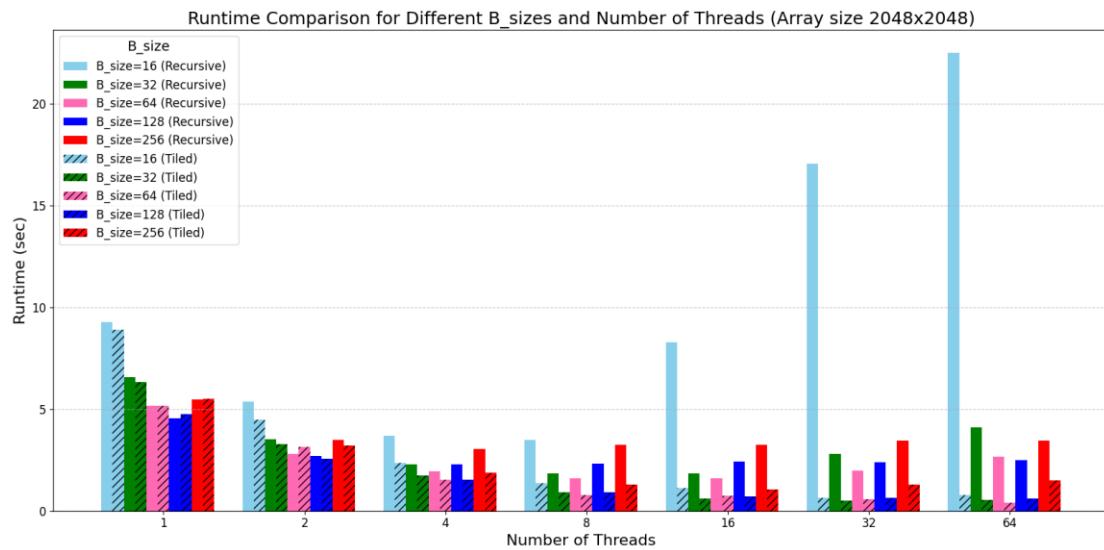
Σημειώνεται πως οι καλύτεροι χρόνοι εκτέλεσης που πήραμε για κάθε μέγεθος εισόδου είναι:

- 1024 x 1024: 0.1117 sec για block size 64 και εκτέλεση με 32 threads
- 2048 x 2048: 0.4157 sec για block size 64 και εκτέλεση με 64 threads
- 4096 x 4096: 1.9624 sec για block size 64 και εκτέλεση με 64 threads

και μέγιστο speedup >20.

Για να συγκρίνουμε με μεγαλύτερη ευκολία τον tiled κώδικα με τον recursive, δημιουργήσαμε τα παρακάτω διαγράμματα.





Σχολιασμός και συμπεράσματα:

Παρατηρώντας τα παραπάνω διαγράμματα αρχικά βλέπουμε την τεράστια διαφορά που έχουν οι 2 υλοποίησεις για τα μικρά block sizes. Η Tiled υλοποίηση υπερέχει συνολικά της recursive. Το block size 64 προσφέρει την καλύτερη ισορροπία μεταξύ τοπικότητας και αποδοτικής χρήσης της cache. Η κλιμάκωση με τον αριθμό των threads είναι ομαλή, με μικρή πτώση για >32 threads. Η Recursive εκδοχή από την άλλη είναι αποτελεσματική μόνο για μικρό αριθμό threads και block sizes. Η απόδοση μειώνεται απότομα με την αύξηση των threads λόγω έλλειψης τοπικότητας και αυξημένου synchronization. Ενώ η recursive εκδοχή φαίνεται να φτάνει τον βέλτιστο χρόνο στα 8 threads, η tiled αξιοποιεί πλήρως την κατανομή σε περισσότερα threads, πετυχαίνοντας ακόμα μικρότερους χρόνους για τα 64 threads.

Από όλα τα παραπάνω καταλήγουμε στην εξής υπόθεση: Είναι πιθανό πως η υλοποίηση ενός υβριδικού αλγορίθμου που θα συνδυάζει την recursive και την tiled προσέγγιση θα μπορούσε να προσφέρει ένα συνολικά βέλτιστο, γρήγορο και αποδοτικό παράλληλο αλγόριθμο. Η tiled προσέγγιση αποδίδει εξαιρετικά όταν η τοπικότητα της μνήμης είναι κρίσιμη, όπως σε μεγάλα datasets ή σε περιπτώσεις όπου το μέγεθος των blocks χωρά στην cache. Για μεγαλύτερα προβλήματα, η tiled μέθοδος θα μπορούσε να χρησιμοποιηθεί για να χωρίσει το πρόβλημα σε μεγαλύτερα blocks που εκμεταλλεύονται την cache. Στη συνέχεια, εντός αυτών των blocks, η recursive μέθοδος θα μπορούσε να εφαρμοστεί για να επεξεργαστεί τα δεδομένα χωρίς να δημιουργείται overhead από περιττή διαχείριση tiles.

Σε μικρότερα προβλήματα, η recursive μέθοδος μπορεί να είναι πιο αποδοτική επειδή μειώνει το overhead που εισάγει η tiled διαχείριση (π.χ., καταμερισμός tiles, συγχρονισμός threads). Η recursive μέθοδος μπορεί να χρησιμοποιηθεί σε αρχικά στάδια ή σε μικρότερα blocks, εξοικονομώντας χρόνο διαχείρισης. Όταν τα blocks φτάνουν σε μεγαλύτερο μέγεθος, η tiled μέθοδος μπορεί να αναλάβει για αποδοτική εκτέλεση.

Επί της ουσίας μια τέτοια υλοποίηση θα είχε ως αποτέλεσμα να εκμεταλλευτούμε τα πλεονεκτήματα και των 2 μεθόδων ώστε να έχουμε τον βέλτιστο αλγόριθμο.

2.3 Ταυτόχρονες Δομές Δεδομένων

Στο παρόν τμήμα, θα εξετάσουμε την επίδοση ορισμένων τεχνικών συγχρονισμού για μια απλά συνδεδεμένη λίστα υπό διαφορετικές συνθήκες. Συγκεκριμένα, θα συγκρίνουμε τις παρακάτω υλοποιήσεις:

- ✓ Coarse-grain locking
- ✓ Fine-grain locking
- ✓ Optimistic synchronization
- ✓ Lazy synchronization
- ✓ Non-blocking synchronization

Κάθε τεχνική θα περιγραφεί συνοπτικά, με στόχο την καλύτερη ερμηνεία των αποτελεσμάτων.

Coarse-grain locking

Το coarse-grain locking χρησιμοποιεί ένα ενιαίο κλείδωμα για ολόκληρη τη δομή δεδομένων, εξασφαλίζοντας τη σειριακή πρόσβαση από τα νήματα. Αυτή η προσέγγιση είναι απλή στην υλοποίηση, καθιστώντας την ιδανική για περιπτώσεις

όπου η πολυτπλοκότητα της συγχρονισμένης διαχείρισης πρέπει να μειωθεί. Ωστόσο, περιορίζει σημαντικά την παράλληλη απόδοση, καθώς μόνο ένα νήμα μπορεί να αποκτήσει πρόσβαση στη δομή σε κάθε λειτουργία, γεγονός που μπορεί να οδηγήσει σε σημαντικές καθυστερήσεις όταν υπάρχουν πολλά νήματα που ανταγωνίζονται για πόρους.

Fine-grain locking

To fine-grain locking χρησιμοποιεί πολλαπλά, ξεχωριστά κλειδώματα για διαφορετικά τμήματα της δομής δεδομένων. Συγκεκριμένα, εφαρμόζεται το hand-over-hand locking για την ασφαλή επεξεργασία της συνδεδεμένης λίστας: κάθε νήμα κλειδώνει τόσο τον τρέχοντα κόμβο όσο και τον επόμενο, και απελευθερώνει κάθε κόμβο μόλις προχωρήσει στον επόμενο. Με αυτόν τον τρόπο, διασφαλίζεται ότι κανένα άλλο νήμα δεν θα κάνει αλλαγές σε κρίσιμους κόμβους κατά τη διάρκεια των λειτουργιών, εξασφαλίζοντας έτσι την αναμενόμενη συμπεριφορά της λίστας. Αυτή η μέθοδος επιτρέπει ταυτόχρονες λειτουργίες σε ανεξάρτητα τμήματα της δομής, βελτιώνοντας την παράλληλη απόδοση. Ωστόσο, είναι σημαντικά πιο δύσκολη στην υλοποίηση και το κόστος απόκτησης και απελευθέρωσης πολλών κλειδωμάτων μπορεί να είναι υψηλό, ειδικά για πιο σύνθετες δομές δεδομένων.

Optimistic synchronization

To optimistic synchronization στοχεύει στη μείωση της χρήσης κλειδωμάτων, ενισχύοντας την απόδοση. Κατά την εισαγωγή ή διαγραφή ενός στοιχείου, πραγματοποιείται αρχικά μια αναζήτηση της κατάλληλης θέσης χωρίς τη χρήση hand-over-hand locking που είχαμε προηγουμένως. Εάν το στοιχείο εντοπιστεί, το νήμα αποκτά κλείδωμα και, στην «παγωμένη» πια περιοχή, επαληθεύει ότι η κατάσταση της δομής παραμένει αμετάβλητη πριν προχωρήσει στην επεξεργασία, δηλαδή εάν ο επόμενος κόμβος στη λίστα είναι αυτός που αναμένει. Εάν ο έλεγχος αυτός αποτύχει τότε γίνεται επαναληπτική προσπάθεια μέχρι την επιτυχία. Στην αναζήτηση ενός στοιχείου, όπως αναφέρθηκε, δεν χρησιμοποιούνται κλειδώματα. Το βασικό overhead αυτής της μεθόδου προέρχεται από την επαλήθευση (validate()), καθώς σε περίπτωση αποτυχίας, το νήμα πρέπει να ξεκινήσει πάλι από την αρχή της λίστας για να βεβαιωθεί ότι η δομή είναι όπως αναμενόταν. Παρόλο που αυτή η προσέγγιση προσθέτει ένα επιπλέον κόστος στην επαναληπτική διαδικασία επαλήθευσης, σε περιπτώσεις όπου οι αλλαγές στη δομή είναι σπάνιες, η συνολική απόδοση βελτιώνεται σημαντικά λόγω της μείωσης της χρήσης κλειδωμάτων.

Lazy synchronization

Το lazy synchronization εισάγει μια "τεμπέλικη" προσέγγιση στη διαχείριση των κλειδωμάτων, προσπαθώντας να μειώσει το overhead της προηγούμενης μεθόδου. Αυτή η προσέγγιση διαχωρίζει τις εργασίες σε δύο φάσεις: την ελαφριά φάση και την βαριά φάση. Σε κάθε κόμβο της λίστας προστίθεται ένα bit πληροφοριών που υποδεικνύει αν ο κόμβος έχει διαγραφεί ή όχι. Στην ελαφριά φάση, πραγματοποιείται η λογική διαγραφή ενός κόμβου απλά αλλάζοντας την τιμή αυτού του bit. Αυτή η φάση εκτελείται άμεσα και με τον απαραίτητο συγχρονισμό. Στη βαριά φάση, γίνεται η φυσική διαγραφή του κόμβου μέσω της αναδιάταξης των δεικτών, και αυτό μπορεί να πραγματοποιηθεί αργότερα, όταν το σύστημα έχει διαθέσιμους πόρους. Αυτή η μεθοδολογία μειώνει τον χρόνο που απαιτείται για την εκτέλεση κρίσιμων τμημάτων, καθώς η επαλήθευση (`validate()`) μπορεί να γίνει απλά ελέγχοντας τα bit των δύο γειτονικών κόμβων, αντί να διασχίζουμε όλη τη λίστα από την αρχή. Αυτό μειώνει σημαντικά το overhead και βελτιώνει την απόδοση. Ωστόσο, η προσέγγιση αυτή απαιτεί σωστή διαχείριση για να αποφευχθούν προβλήματα όπως η συσσώρευση περιττών κόμβων, που θα μπορούσαν να μειώσουν την αποδοτικότητα της δομής δεδομένων αν παραμείνουν για μεγάλο χρονικό διάστημα.

Non-blocking synchronization

Το non-blocking synchronization αποφεύγει εντελώς τη χρήση κλειδωμάτων, βασιζόμενο στις ατομικές εντολές που προσφέρει το υλικό, όπως οι εντολές Test-and-Set (TAS) και Compare-and-Swap (CAS). Αυτή η μέθοδος επιτρέπει στα νήματα να εκτελούν λειτουργίες ταυτόχρονα χωρίς να μπλοκάρουν το ένα το άλλο, εξαλείφοντας το overhead που σχετίζεται με τα κλειδώματα. Παρότι είναι εξαιρετικά αποδοτική τεχνική σε περιβάλλοντα με υψηλή ταυτόχρονη δραστηριότητα, η υλοποίησή της είναι συνήθως πιο περίπλοκη και απαιτεί βαθιά κατανόηση του υλικού και της συμπεριφοράς των ατομικών εντολών.

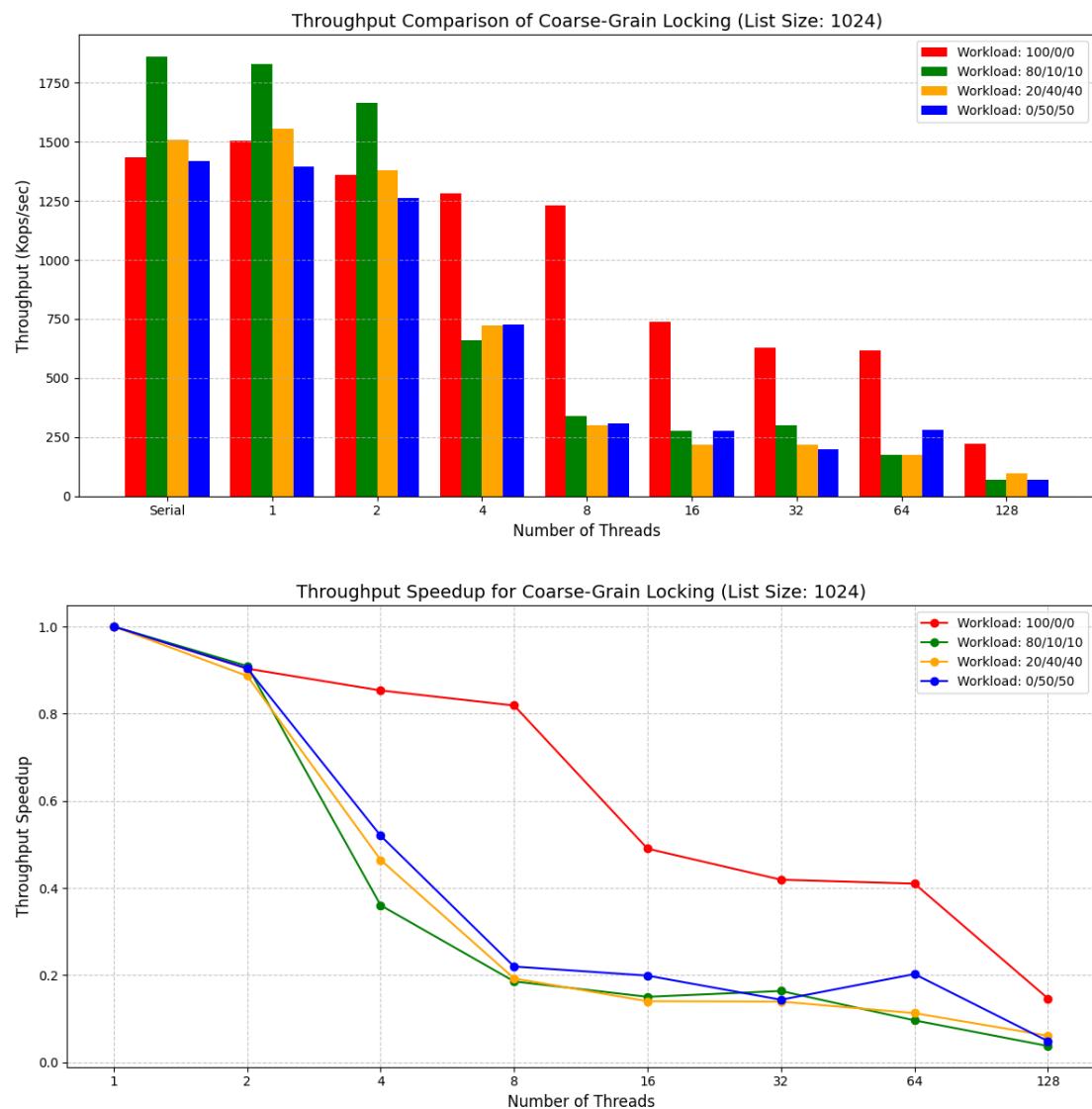
Η εκτέλεση πραγματοποιείται με την παρακάτω παραμετροποίηση:

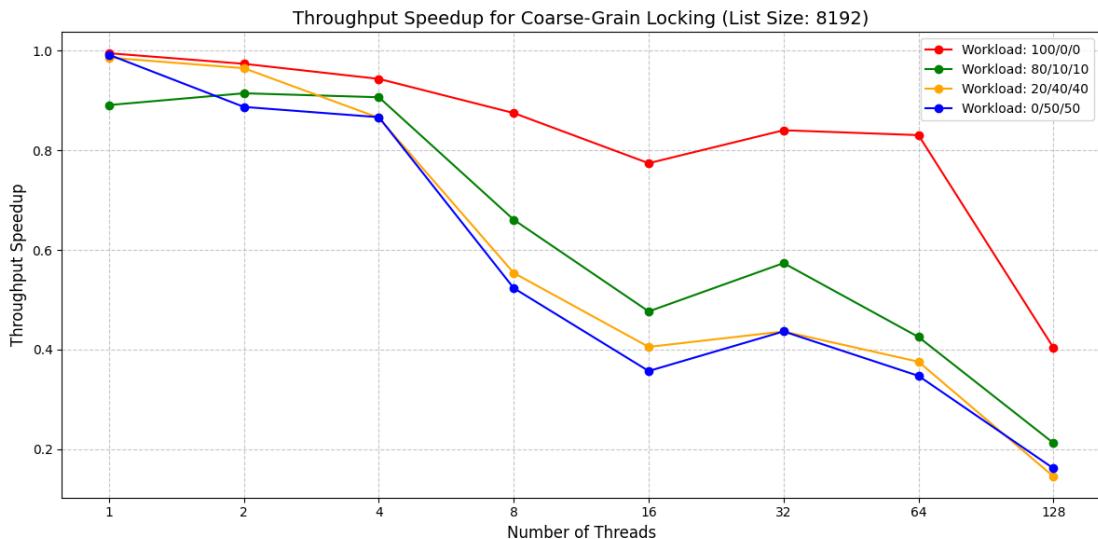
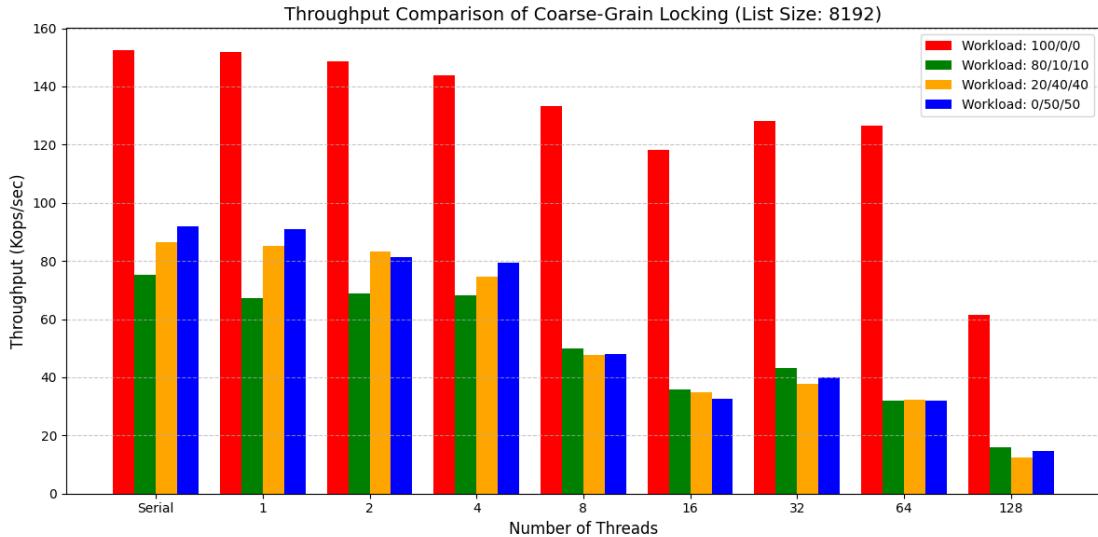
- Αριθμός νημάτων (threads): **1, 2, 4, 8, 16, 32, 64, 128.**
Κάθε φορά ρυθμίζουμε κατάλληλα τη μεταβλητή `MT_CONF`, ώστε τα νήματα να κατανέμονται σε διαδοχικούς φυσικούς πυρήνες. Για 64 και 128 threads, όπου οι 32 διαθέσιμοι φυσικοί πυρήνες δεν επαρκούν, αξιοποιείται το *hyperthreading* όπου κάθε φυσικός πυρήνας τρέχει 2 νήματα και *oversubscription* με κάθε φυσικό πυρήνα να εκτελεί 4 νήματα αντίστοιχα.
- Μέγεθος λίστας: **1024, 8192.**
- Ποσοστό λειτουργιών (αναζητήσεις-εισαγωγές-διαγραφές): **100-0-0, 80-10-10, 20-40-40, 0-50-50.**

Ακολουθούν τα διαγράμματα του throughput (kops/sec) και του throughput speedup ως συνάρτηση του αριθμού των threads για κάθε ποσοστό λειτουργιών, προκειμένου να εξετάσουμε πώς η απόδοση επηρεάζεται από τον συγχρονισμό σε σχέση με τα διαφορετικά workloads. Το throughput speedup υπολογίζεται ως ο λόγος του throughput για την ταυτόχρονη υλοποίηση προς το throughput για τη σειριακή έκδοση. Παρουσιάζουμε ένα διάγραμμα για κάθε μέγεθος λίστας και κάθε τεχνική συγχρονισμού.

Σημειώνεται ότι η σειριακή έκδοση εκτελέστηκε μόνο με 1 thread, αφού η δομή δίχως κανένα είδους συγχρονισμού δεν είναι *thread safe*, και άρα η οποιαδήποτε προσπάθεια παραληλισμού θα είχε μη αναμενόμενη συμπεριφορά.

Coarse-grain locking





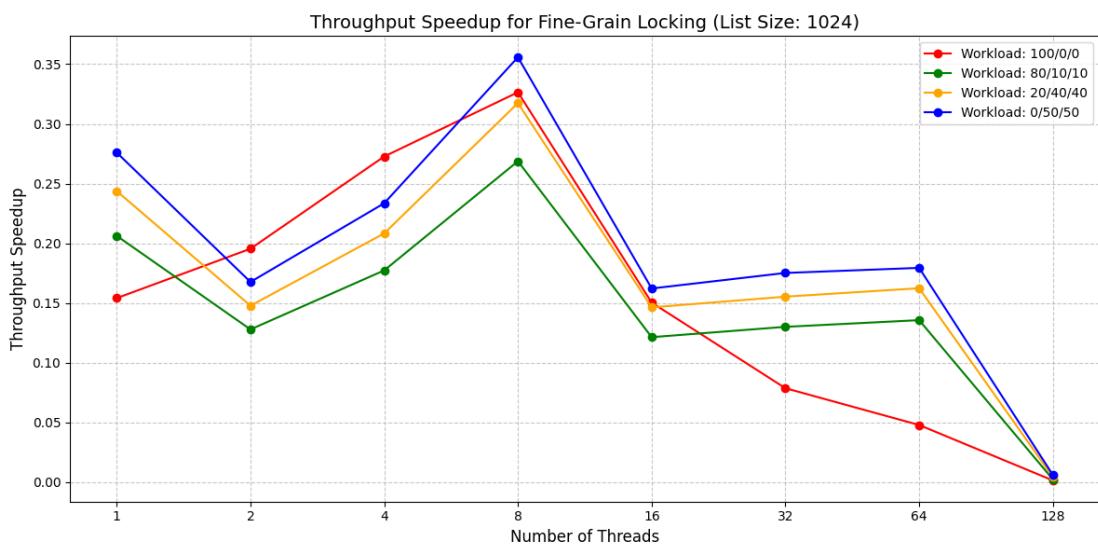
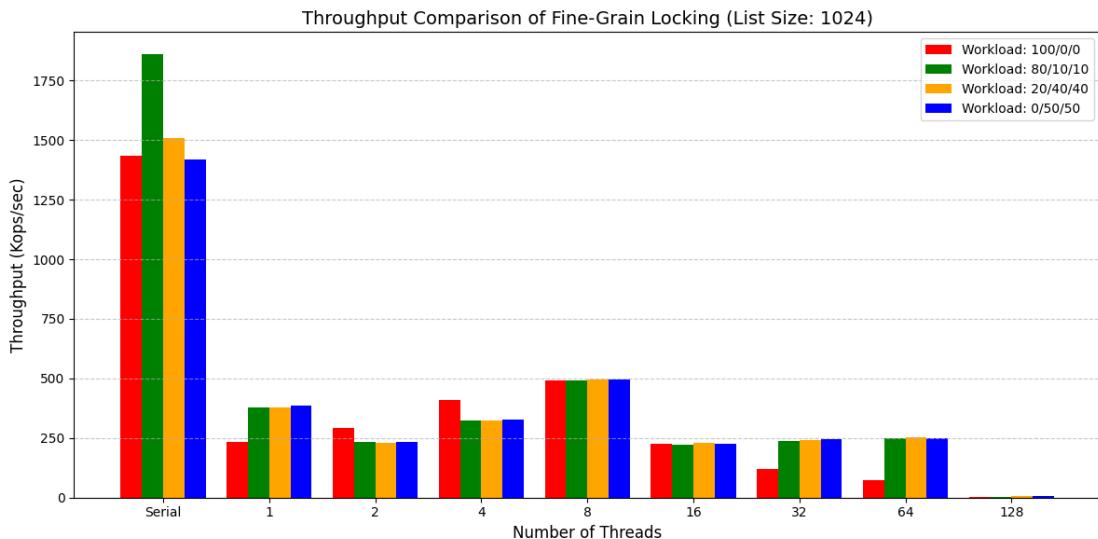
Στην περίπτωση του coarse-grain locking, όπου ολόκληρη η δομή κλειδώνεται με ένα ενιαίο κλείδωμα για κάθε λειτουργία, το throughput είναι εντυπωσιακά υψηλότερο όταν ο αριθμός των νημάτων είναι μικρός. Αυτή η συμπεριφορά είναι φυσιολογική, καθώς κάθε λειτουργία αποκτά αποκλειστική πρόσβαση στη δομή, οδηγώντας σε σειριοποίηση των λειτουργιών. Όταν αυξάνεται ο αριθμός των νημάτων, παρατηρούμε ότι το μόνο που αλλάζει είναι το κόστος του συγχρονισμού, αφού η εργασία εξακολουθεί να εκτελείται κατά βάση σειριακά λόγω του αποκλειστικού κλειδώματος. Το overhead του συγχρονισμού γίνεται εμφανέστερο με μεγάλο αριθμό νημάτων, όπου η απόδοση πέφτει δραματικά.

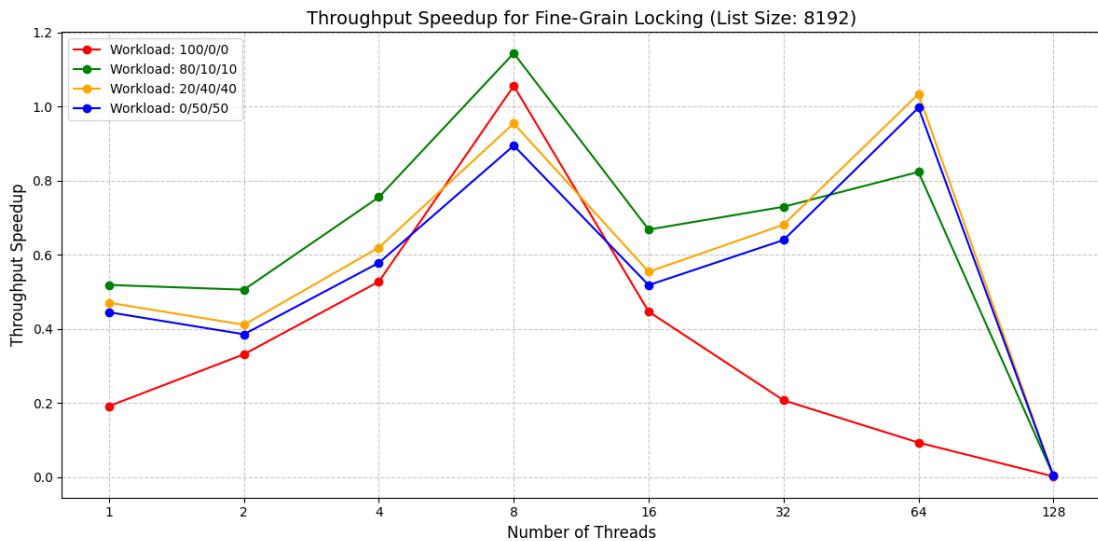
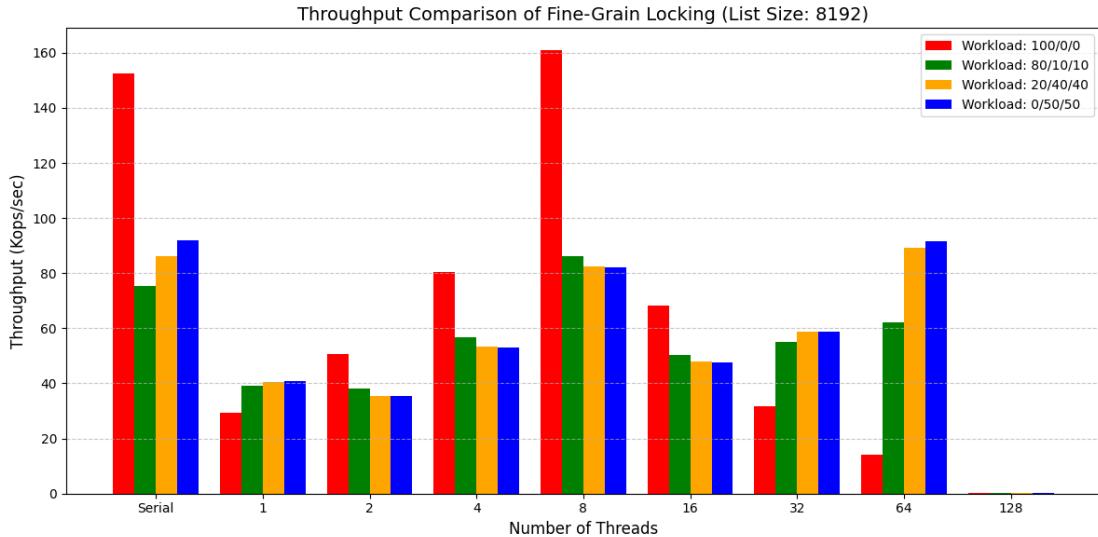
Επιπλέον, σε κανένα σενάριο δεν παρατηρείται throughput μεγαλύτερο από αυτό της σειριακής εκτέλεσης, ανεξαρτήτως του μεγέθους της λίστας, καθώς η σειριοποίηση που προκαλείται από το ενιαίο κλείδωμα περιορίζει την ταυτόχρονη εκτέλεση. Είναι επίσης εμφανές ότι το throughput μειώνεται σημαντικά όταν το workload περιλαμβάνει insert και delete operations, γεγονός που οφείλεται στο

αυξημένο κόστος χρόνου που απαιτούν αυτές οι λειτουργίες λόγω των μεταβολών στη δομή.

Όπως αναμενόταν, το throughput είναι μεγαλύτερο για μικρότερα μεγέθη λίστας, καθώς η επεξεργασία μικρότερης δομής είναι ταχύτερη και το κλείδωμα παραμένει ενεργό για μικρότερο χρονικό διάστημα, επιτρέποντας πιο γρήγορη απελευθέρωση της δομής για άλλες λειτουργίες. Αντίθετα, για μεγαλύτερες λίστες, η επεξεργασία είναι πιο χρονοβόρα, και η δομή παραμένει κλειδωμένη για περισσότερο χρόνο, αποκλείοντας ταυτόχρονες λειτουργίες και μειώνοντας έτσι την απόδοση.

Fine-grain locking





Στην περίπτωση του fine-grain locking, για μικρό μέγεθος λίστας παρατηρούμε ότι το throughput καταφέρνει να εξισορροπηθεί μεταξύ των διαφορετικών workloads. Ωστόσο, το hand-over-hand locking προσθέτει σημαντικό overhead, το οποίο μας εμποδίζει να ξεπεράσουμε το throughput της σειριακής εκτέλεσης. Το overhead αυτό γίνεται ιδιαίτερα εμφανές στο workload που τρέχει μόνο αναζητήσεις (100/0/0), όπου το throughput είναι σημαντικά χαμηλότερο σε σύγκριση με τη σειριακή εκτέλεση, καθώς η έλλειψη συγχρονισμού σε αυτό το σενάριο επιτρέπει υψηλότερη απόδοση.

Καθώς ο αριθμός των νημάτων αυξάνεται, το throughput μειώνεται, καθώς όλο και περισσότερα νήματα βρίσκονται "παγωμένα" εξαιτίας των κλειδωμάτων που δημιουργούνται σε κάποιο σημείο της λίστας μέσω του hand-over-hand locking. Ωστόσο, σε αντίθεση με το coarse-grain locking, το fine-grain locking επιτυγχάνει καλύτερο throughput για μεγαλύτερο μέγεθος λίστας. Αυτό είναι αναμενόμενο, καθώς η μεγαλύτερη λίστα προσφέρει μεγαλύτερη πιθανότητα τα διαφορετικά

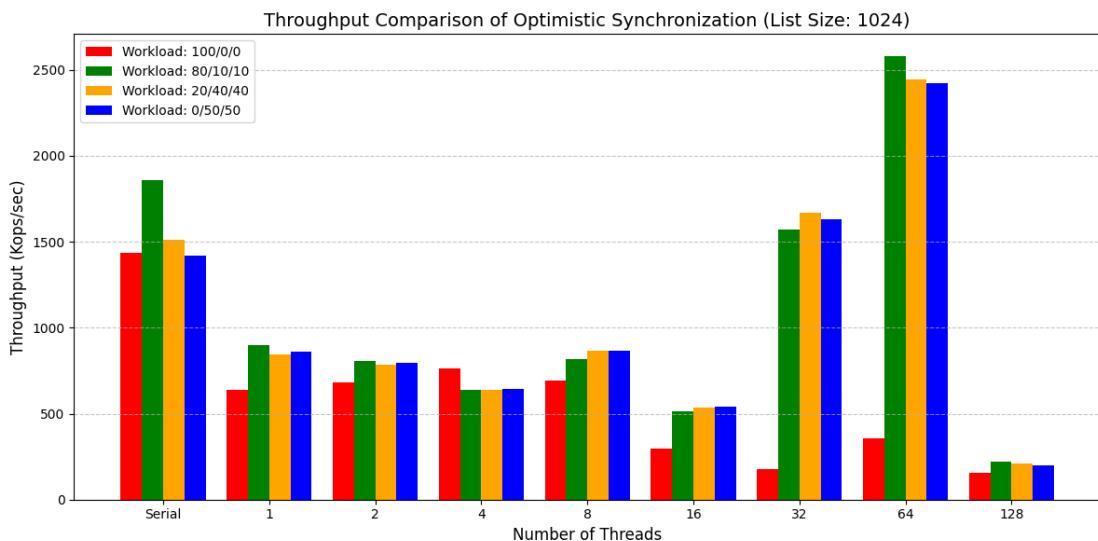
νήματα να δουλεύουν σε απομακρυσμένα σημεία της λίστας, ελαχιστοποιώντας τις συγκρούσεις και επιτρέποντας καλύτερη ταυτόχρονη πρόσβαση.

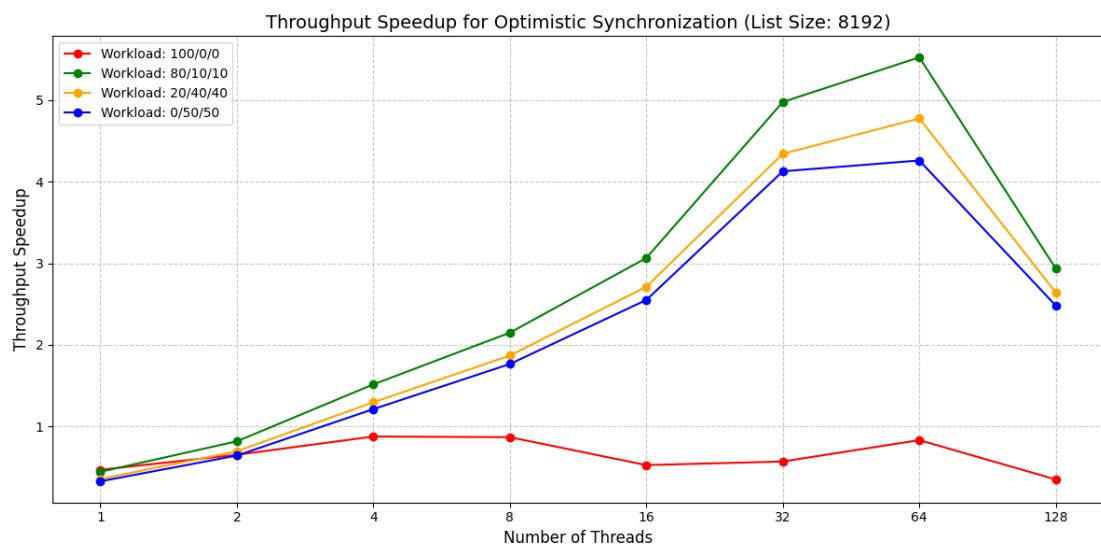
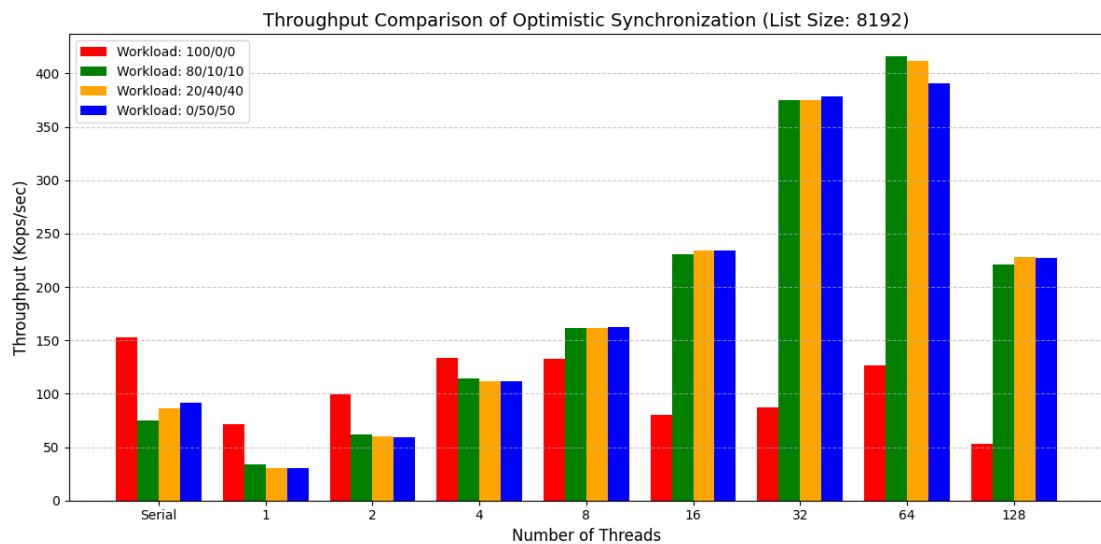
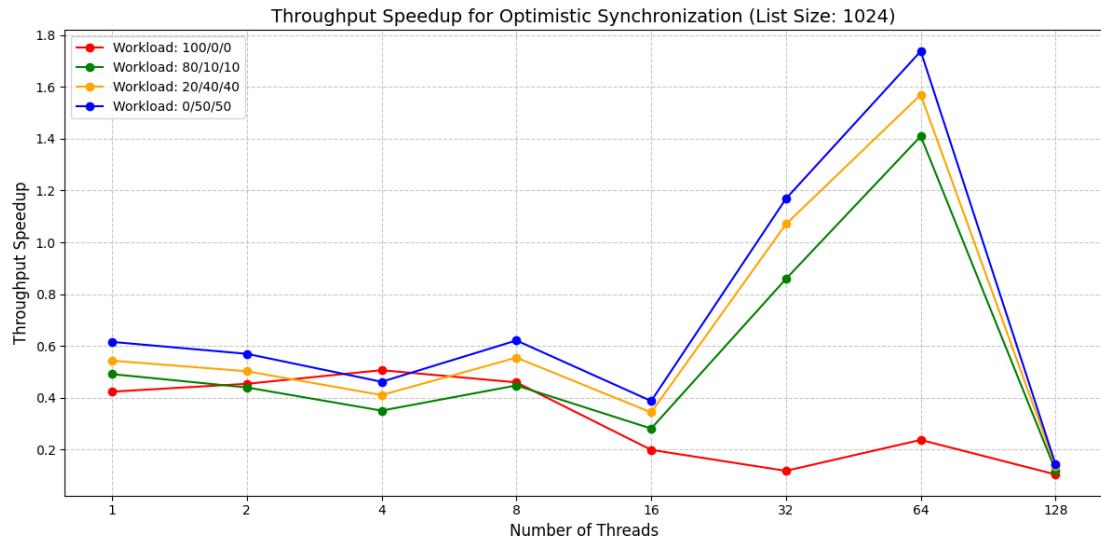
Στο μεγαλύτερο μέγεθος λίστας, παρατηρούμε για πρώτη φορά ότι το throughput ξεπερνά τη σειριακή εκτέλεση για 8 νήματα, κάτι που δείχνει την ικανότητα του fine-grain locking να εκμεταλλευτεί αποτελεσματικά τον παραλληλισμό όταν οι συνθήκες είναι ευνοϊκές. Ωστόσο, πέρα από αυτό το σημείο, η κλιμάκωση δεν είναι καλή, καθώς το throughput δεν βελτιώνεται σημαντικά με περισσότερα νήματα.

Ειδικότερα, στην περίπτωση του oversubscription με 128 νήματα, το throughput πέφτει δραματικά, με τιμές τόσο χαμηλές που είναι δύσκολο να διακριθούν στο διάγραμμα. Αυτό οφείλεται στο γεγονός ότι το κέρδος από τον παραλληλισμό δεν μπορεί να αντισταθμίσει το κόστος του context switching που επιβάλλεται από το λειτουργικό σύστημα, το οποίο προσπαθεί να διαχειριστεί πολλά περισσότερα νήματα από τους διαθέσιμους πυρήνες. Το συνεχές context switching οδηγεί σε σπατάλη των πόρων του συστήματος και σε σημαντική μείωση του throughput.

Συνοψίζοντας, το fine-grain locking προσφέρει καλύτερη ταυτόχρονη εκτέλεση και αποδοτικότητα για μεγαλύτερες λίστες και σε καταστάσεις με μικρό αριθμό νημάτων. Παρ' όλα αυτά, όταν ο αριθμός των νημάτων αυξάνεται, το overhead από τον συγχρονισμό και το context switching γίνεται ο κυρίαρχος παράγοντας που επιβαρύνει την απόδοση, ιδιαίτερα σε περιπτώσεις oversubscription.

Optimistic synchronization





To optimistic synchronization επιδεικνύει σημαντική βελτίωση στην κλιμάκωση του throughput μέχρι τα 64 νήματα. Η απόδοση βελτιώνεται διότι η τεχνική αυτή επιτρέπει στα νήματα να εκτελούν αναζητήσεις χωρίς να χρειάζονται άμεσα

κλειδώματα, μειώνοντας έτσι τις συγκρούσεις. Όταν όμως ο αριθμός των νημάτων υπερβαίνει τους διαθέσιμους πόρους του συστήματος (128 νήματα), παρατηρείται σημαντική μείωση του throughput λόγω του oversubscription, όπου το κόστος του context switching γίνεται υψηλό, μειώνοντας την απόδοση του συστήματος.

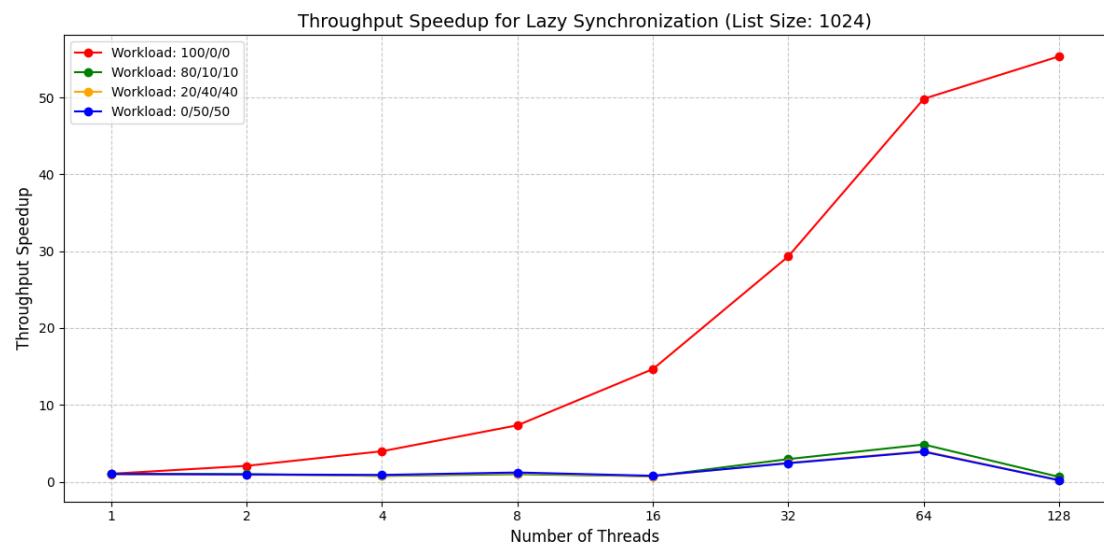
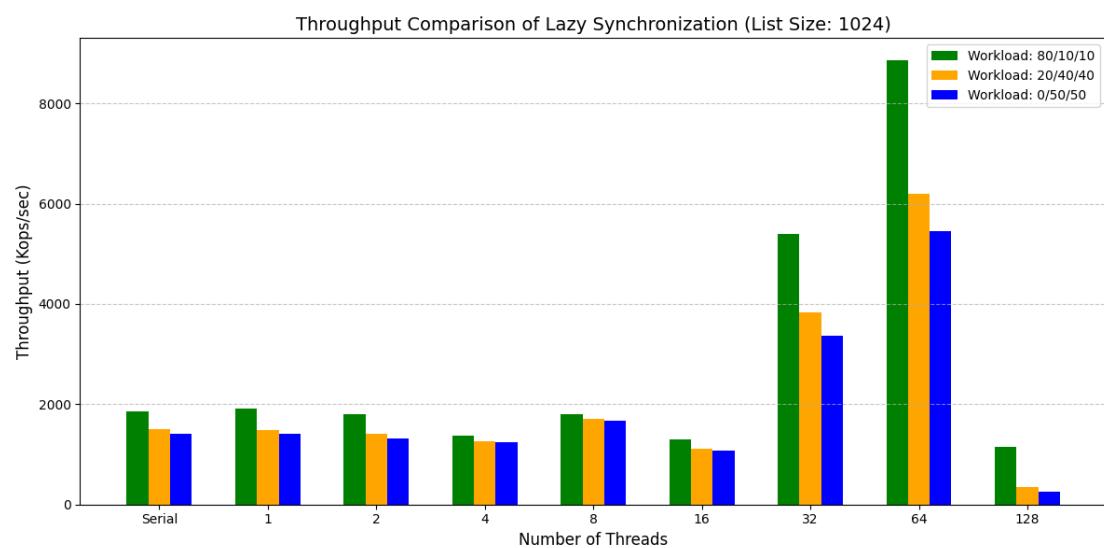
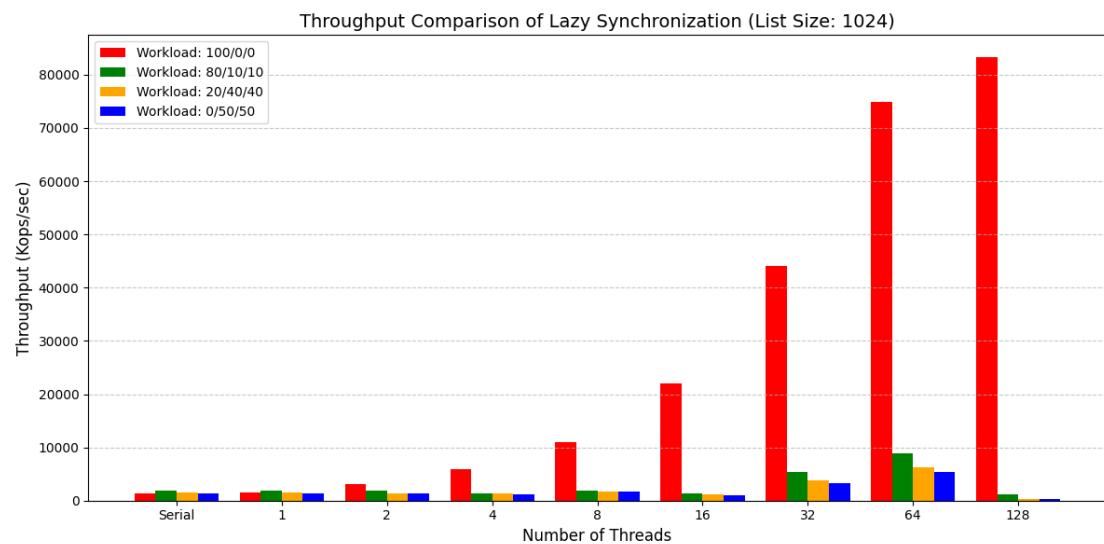
Το μέγεθος της λίστας έχει επίσης καθοριστική επίδραση στην απόδοση. Σε λίστα μεγέθους 8192, το optimistic synchronization επιτρέπει στα νήματα να εργάζονται σε πιο απομακρυσμένα τμήματα της λίστας, μειώνοντας τις συγκρούσεις και επιτυγχάνοντας καλύτερη ταυτόχρονη εκτέλεση. Αντίθετα, σε μικρότερη λίστα (1024), τα νήματα έχουν μεγαλύτερη πιθανότητα να δουλεύουν κοντά το ένα στο άλλο, με αποτέλεσμα να αυξάνεται το κόστος συγχρονισμού και το throughput να μην βελτιώνεται ανάλογα. Ακόμα, το κόστος της validate() για μεγαλύτερες λίστες είναι πιο αισθητό, αφού τις διατρέχει από την αρχή.

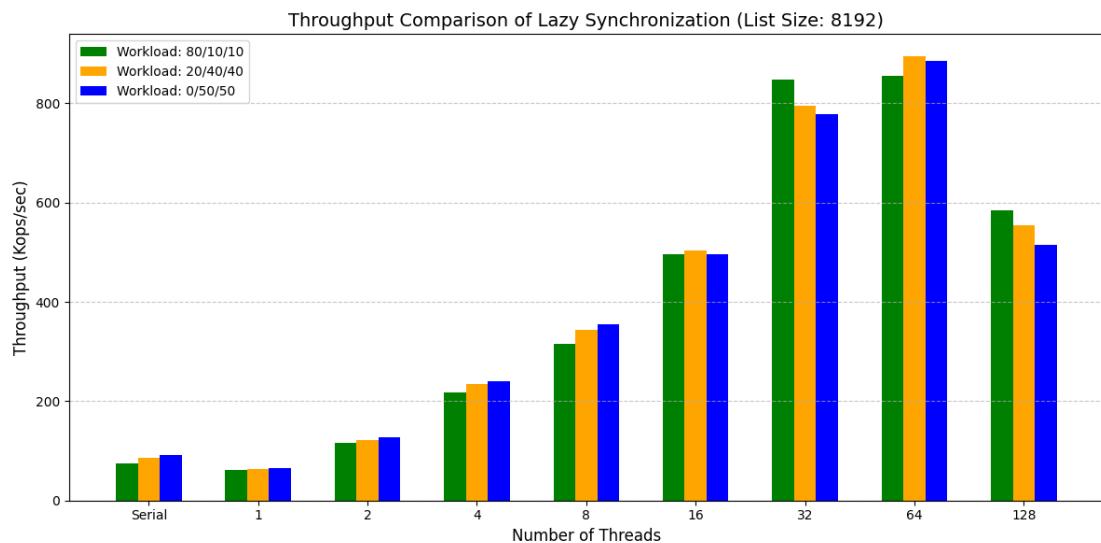
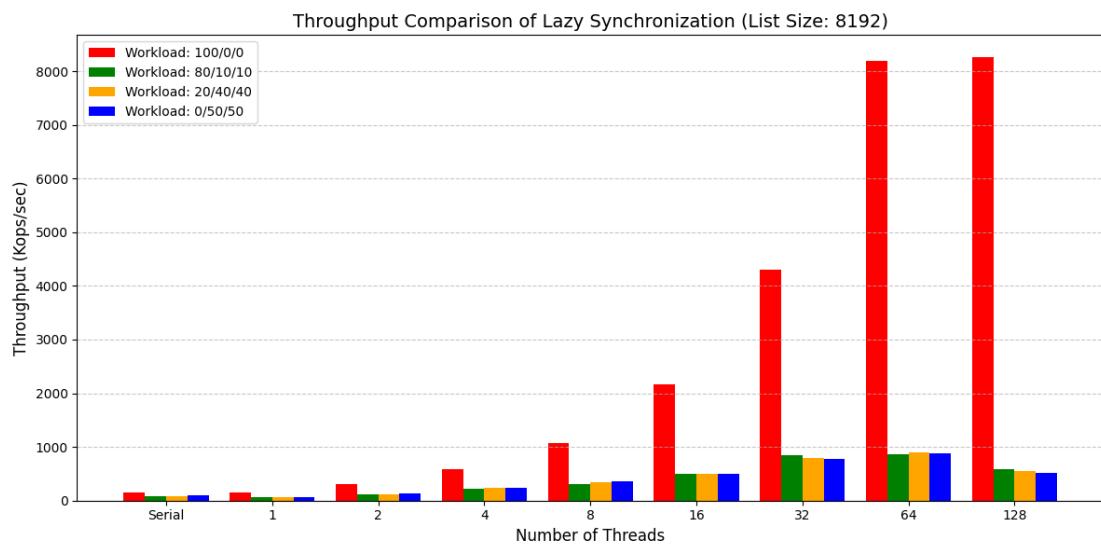
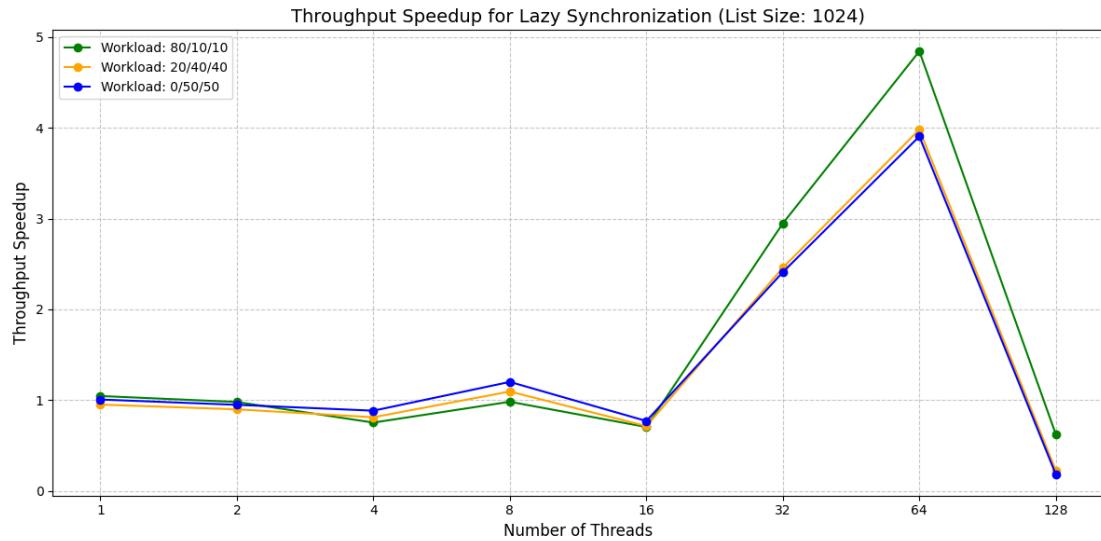
Στα workloads με μόνο αναζητήσεις (100/0/0), το throughput δεν κλιμακώνει καλά λόγω του κόστους της validate(), η οποία διατρέχει τη λίστα για να επαληθεύσει την ακεραιότητά της. Αυτό το overhead καθιστά τη σειριακή εκτέλεση συχνά πιο αποδοτική. Από την άλλη πλευρά, στα workloads που περιλαμβάνουν τροποποιήσεις (όπως 80/10/10, 20/40/40, 0/50/50), το optimistic synchronization καταφέρνει να βελτιώσει το throughput καθώς αποφεύγεται η άμεση χρήση αποκλειστικών κλειδωμάτων, επιτρέποντας στα νήματα να εργάζονται πιο παράλληλα.

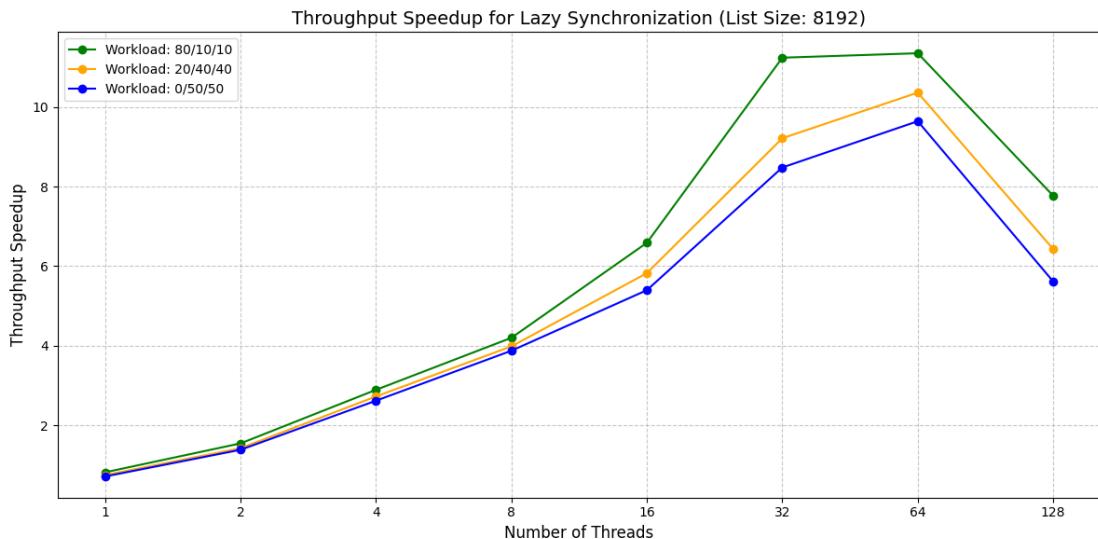
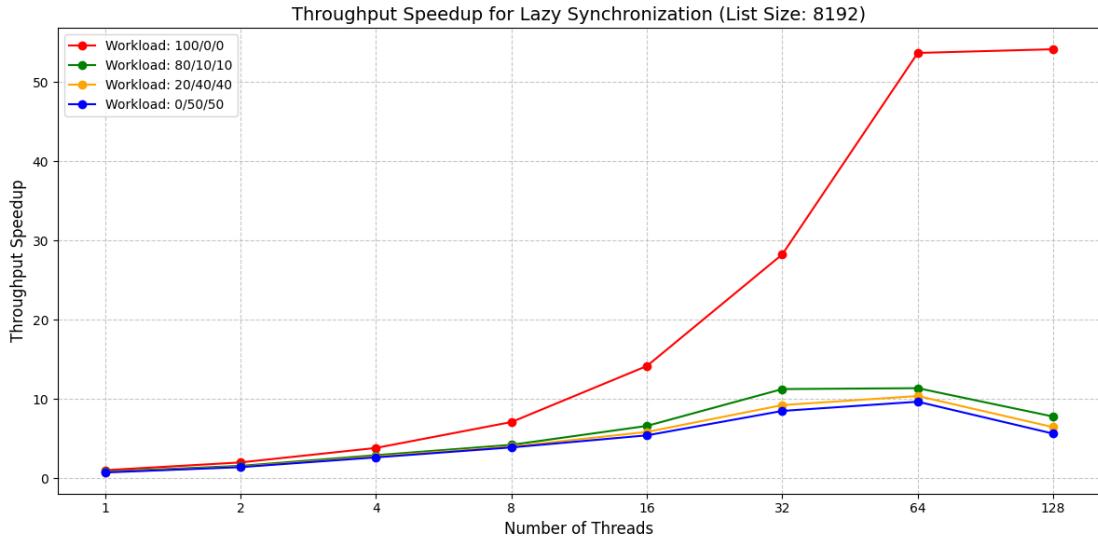
Συνολικά, το optimistic synchronization είναι αποδοτικό για μέτριο αριθμό νημάτων και μεγαλύτερες λίστες, όπου οι συγκρούσεις μειώνονται και επιτυγχάνεται καλύτερη ταυτόχρονη πρόσβαση. Ωστόσο, η απόδοσή του περιορίζεται από το overhead της validate() σε workloads με μόνο αναγνώσεις, ενώ σε περιπτώσεις oversubscription το υψηλό κόστος του context switching επηρεάζει αρνητικά την απόδοση. Είναι μια καλή επιλογή για σενάρια με ισορροπημένο αριθμό νημάτων και κατάλληλη αναλογία αναζητήσεων και τροποποιήσεων, προσφέροντας καλύτερο παραλληλισμό από άλλες πιο συγκεντρωτικές τεχνικές κλειδώματος.

Lazy synchronization

Στα παρακάτω διαγράμματα, καθώς το throughput στο πρώτο workload είναι αισθητά υψηλότερο σε σύγκριση με τα υπόλοιπα, παραθέτουμε για λόγους σαφήνειας ένα επιπλέον διάγραμμα που περιλαμβάνει μόνο τα τρία τελευταία workloads, ώστε να διευκολυνθεί η διεξαγωγή πιο ακριβών συμπερασμάτων.







Η πρώτη σαφής βελτίωση που παρατηρούμε σε σχέση με τις προηγούμενες υλοποιήσεις συγχρονισμού είναι η σημαντικά αυξημένη τιμή του throughput στο πρώτο workload που περιλαμβάνει μόνο αναζητήσεις (100/0/0). Αυτή η βελτίωση αναδεικνύει τη μείωση του κόστους της διαδικασίας validate(), η οποία πλέον δεν χρειάζεται να διατρέχει τη λίστα από την αρχή. Αντίθετα, μπορεί να ελέγχει την τιμή δύο bit, κάτι που απλοποιεί την επαλήθευση της ακεραιότητας και μειώνει δραματικά το κόστος του συγχρονισμού στις αναζητήσεις.

Όσον αφορά τα υπόλοιπα workloads που περιλαμβάνουν τροποποιήσεις, όπως εισαγωγές και διαγραφές, παρατηρούμε ότι το lazy synchronization καταφέρνει να διατηρήσει καλή απόδοση καθώς αυξάνεται ο αριθμός των νημάτων. Η χρήση "τεμπέλικης" διαχείρισης, δηλαδή η διάκριση των λειτουργιών σε λογικές και φυσικές τροποποιήσεις, μειώνει το κόστος των κλειδωμάτων στα κρίσιμα τμήματα της λίστας. Τα νήματα εκτελούν τις τροποποιήσεις τους πιο αποτελεσματικά, χωρίς να εμποδίζονται υπερβολικά από τις διαδικασίες συγχρονισμού. Αυτό εδηγεί τη σημαντική αύξηση του throughput, ειδικά όταν ο

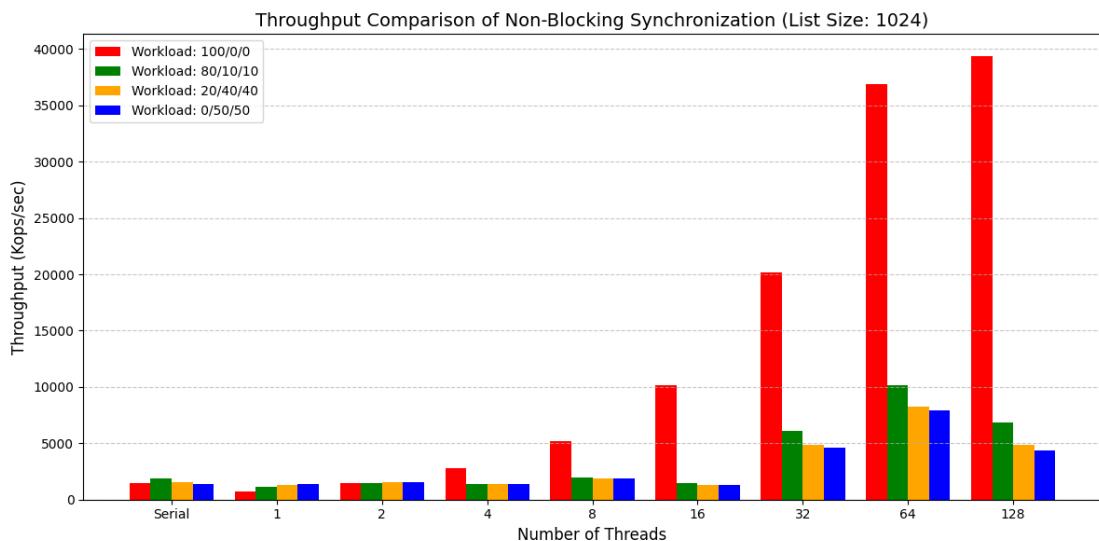
αριθμός των νημάτων φτάνει τα 32 και 64, όπου η αποδοτικότητα διατηρείται σε υψηλά επίπεδα.

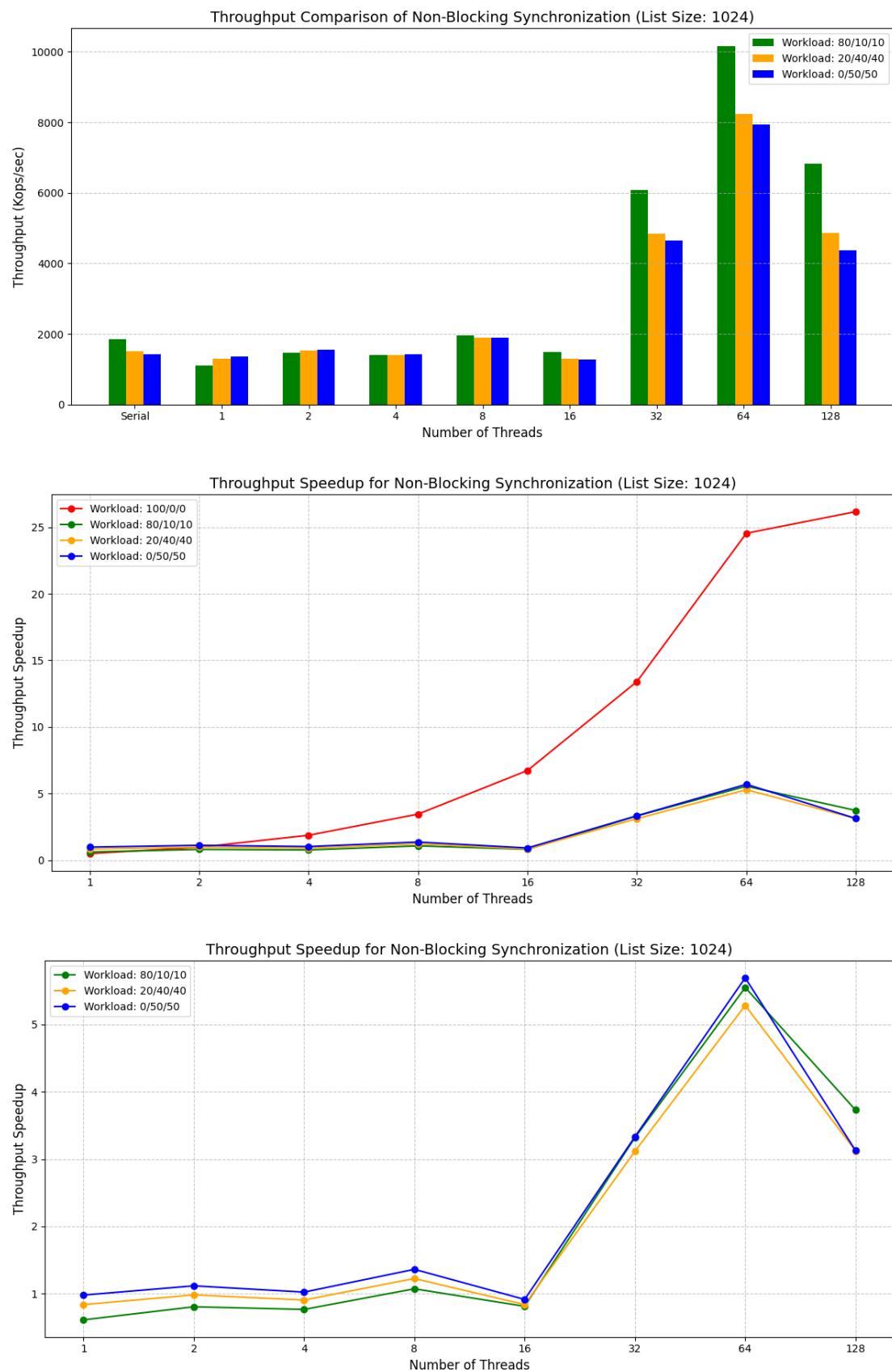
Σε συνθήκες oversubscription με 128 νήματα, το throughput μειώνεται αισθητά, λόγω του αυξημένου κόστους του context switching, όπως έχουμε ήδη αναφέρει.

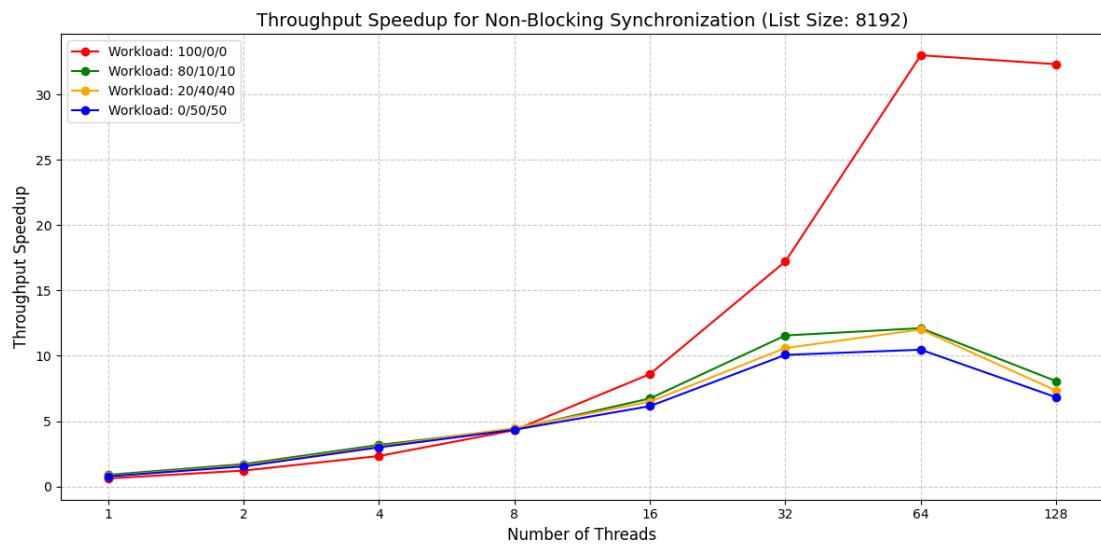
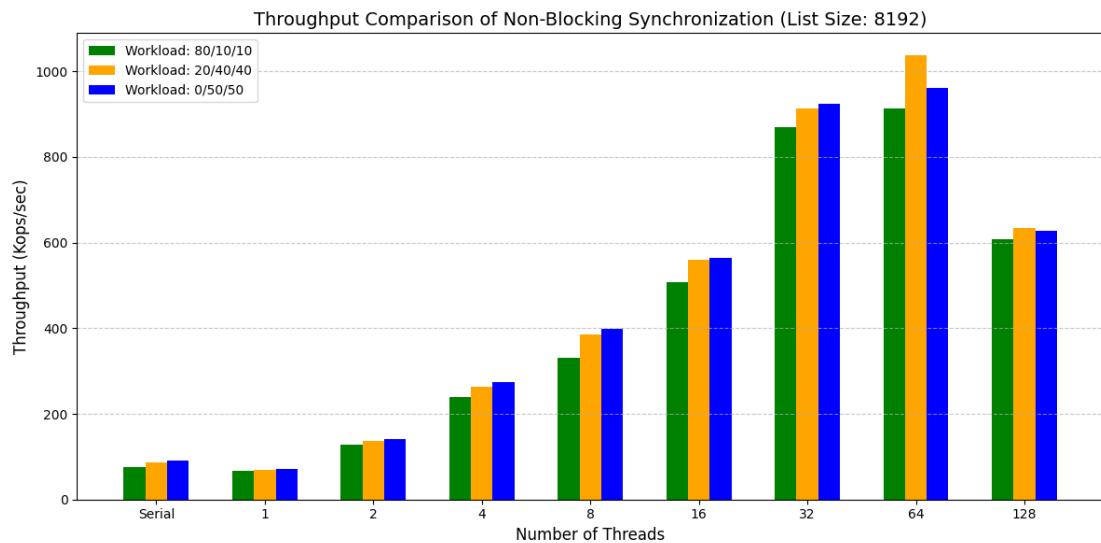
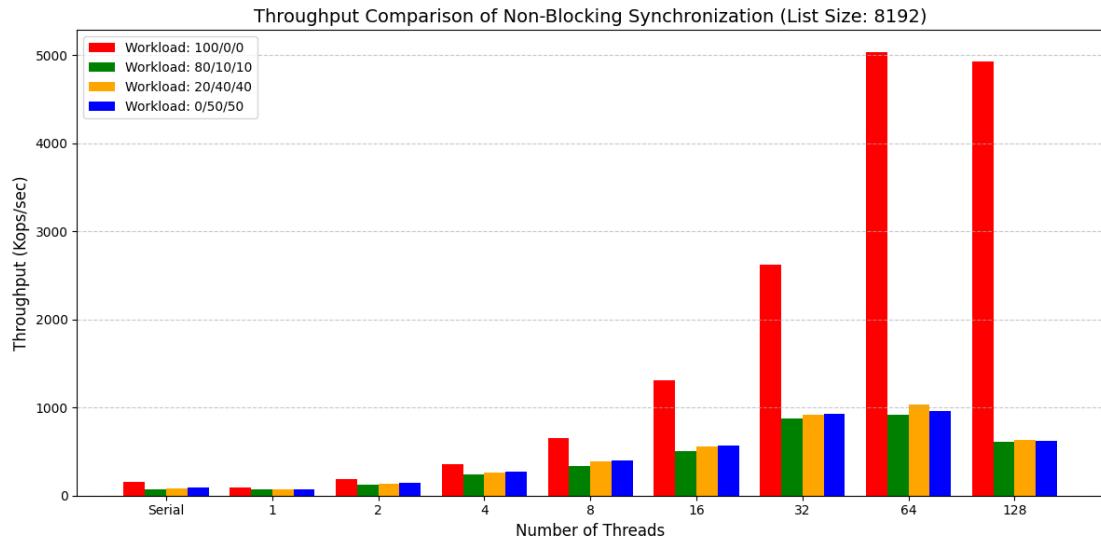
Συνοψίζοντας, το lazy synchronization καταφέρνει να βελτιώσει σημαντικά την απόδοση σε workloads και καταφέρνει να αυξήσει αισθητά το throughput σε σχέση με την σειριακή εκτέλεση παρουσιάζοντας ικανοποιητική κλιμάκωση για μεγάλο αριθμό νημάτων.

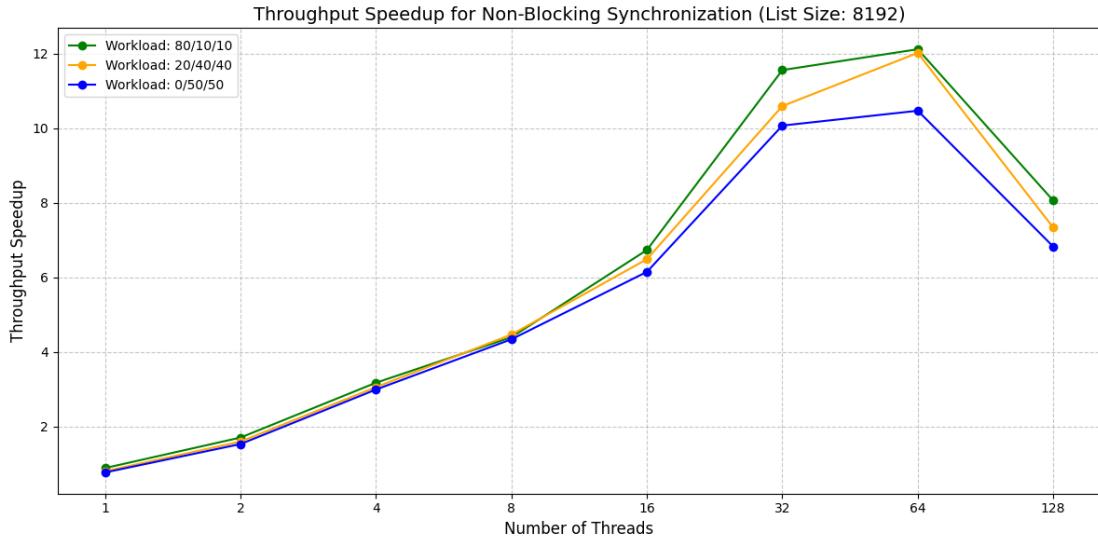
Non-blocking synchronization

Στα παρακάτω διαγράμματα, καθώς το throughput στο πρώτο workload είναι αισθητά υψηλότερο σε σύγκριση με τα υπόλοιπα, παραθέτουμε για λόγους σαφήνειας ένα επιπλέον διάγραμμα που περιλαμβάνει μόνο τα τρία τελευταία workloads, ώστε να διευκολυνθεί η διεξαγωγή πιο ακριβών συμπερασμάτων.









Σε αυτή την υλοποίηση επιδιώκουμε να μειώσουμε το overhead από τη χρήση κλειδωμάτων, αξιοποιώντας ατομικές εντολές που προσφέρονται από το υλικό. Παρατηρούμε ότι η συγκεκριμένη υλοποίηση κλιμακώνει ικανοποιητικά για όλα τα workloads καθώς αυξάνεται ο αριθμός των νημάτων, επιτρέποντας την αξιοποίηση του παραλληλισμού χωρίς το κόστος που συνεπάγεται ο παραδοσιακός συγχρονισμός.

Στα workloads που περιλαμβάνουν μόνο αναζητήσεις, το throughput εμφανίζεται σημαντικά αυξημένο, καθώς η δομή της λίστας παραμένει σταθερή χωρίς τροποποιήσεις. Αυτό εξαλείφει το κόστος από συγκρούσεις και την ανάγκη για επαλήθευση της ακεραιότητας, επιτρέποντας στα νήματα να διατρέχουν τη λίστα χωρίς παρεμβολές. Η απουσία τροποποιήσεων κάνει τις αναζητήσεις εξαιρετικά αποδοτικές, διευκολύνοντας την πλήρη εκμετάλλευση του παραλληλισμού. Ωστόσο, σε σύγκριση με την προηγούμενη υλοποίηση, παρατηρούμε χαμηλότερο throughput στις αναζητήσεις. Αυτό συμβαίνει επειδή κατά τη διαδικασία της αναζήτησης μπορεί να βρεθεί κάποιος κόμβος που είναι προγραμματισμένος να αφαιρεθεί, οδηγώντας σε διαδικασίες αφαίρεσης που επηρεάζουν την απόδοση.

Για workloads με τροποποιήσεις, η απόδοση βελτιώνεται καθώς αυξάνεται ο αριθμός των νημάτων, αλλά φτάνει σε ένα σημείο όπου το overhead από τις αποτυχημένες προσπάθειες τροποποίησης και η συμφόρηση μειώνουν την αποτελεσματικότητα. Οι ατομικές εντολές βελτιώνουν την απόδοση σε σχέση με τα πλήρη κλειδώματα, αλλά η απόδοσή τους εξαρτάται από τη συχνότητα των συγκρούσεων, ειδικά όταν υπάρχουν πολλές ταυτόχρονες τροποποιήσεις σε μεγάλες λίστες. Ο ανταγωνισμός για τους ίδιους κόμβους οδηγεί σε μεγαλύτερο κόστος σε περιπτώσεις έντονου συγχρονισμού.

Για μικρό αριθμό νημάτων, οι μικρότερες τιμές throughput οφείλονται στο αρχικό overhead των ατομικών εντολών, το οποίο δεν αντισταθμίζεται από το κέρδος της παράλληλης εκτέλεσης. Όσο αυξάνεται ο αριθμός των νημάτων, το κέρδος από την

tautóchronη πρόσβαση αρχίζει να υπερνικά το κόστος διαχείρισης των ατομικών εντολών, βελτιώνοντας σταδιακά την απόδοση.

Τελικά συμπεράσματα:

Θα παραθέσουμε τώρα κάποια τελικά συμπεράσματα για να γίνει πιο σαφής η επίδραση της κάθε παραμετροποίησης που εξετάσαμε στην επίδοση.

✓ **List size**

Είναι εμφανές από τα παραπάνω διαγράμματα, αλλά πολύ περισσότερο από το διάγραμμα που παραθέτουμε παρακάτω, πως πετυχαίνουμε σημαντικά καλύτερο throughput για μικρότερες λίστες. Αυτό το συμπέρασμα είναι απολύτως λογικό, καθώς λειτουργίες επεξεργασίας και περιήγησης είναι εγγενώς πιο γρήγορες σε λίστες με λιγότερους κόμβους. Εκτός αυτού, οι μικρότερες λίστες αξιοποιούν πιο αποδοτικά την τοπικότητα της μνήμης (cache locality), καθώς ένα μεγαλύτερο ποσοστό της δομής μπορεί να αποθηκευτεί στην cache της CPU. Αντίθετα, στις μεγαλύτερες λίστες, η ανάγκη συχνής προσπέλασης της κύριας μνήμης (main memory) λόγω περιορισμένης cache οδηγεί σε υψηλότερη καθυστέρηση (latency) και συνεπώς μειωμένο throughput. Έτσι, η μονάδα χρόνου αξιοποιείται πιο αποτελεσματικά για μεγαλύτερο αριθμό λειτουργιών σε μικρότερες λίστες.

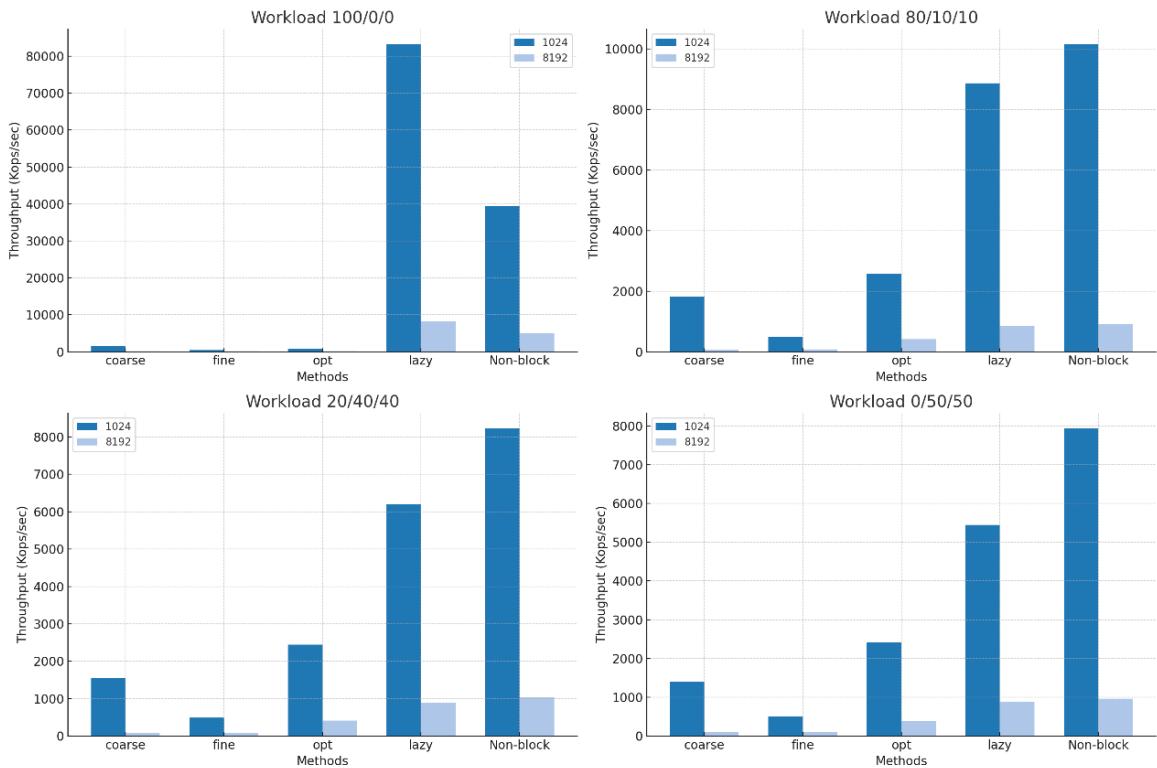
✓ **Workloads**

Το είδος των λειτουργιών που πραγματοποιούνται επηρεάζει καθοριστικά το throughput. Όταν έχουμε μόνο read λειτουργίες, επιτυγχάνουμε υψηλότερο throughput στις περισσότερες περιπτώσεις, καθώς δεν μεταβάλλεται η δομή της λίστας. Όσο αυξάνονται οι write λειτουργίες (εισαγωγές και διαγραφές), τόσο μειώνεται το throughput λόγω του πρόσθετου χρόνου που απαιτείται για την τροποποίηση της λίστας και των αποτυχημένων προσπαθειών συγχρονισμού που προκαλούνται από τις αλλαγές της δομής από άλλα νήματα. Επιπλέον, οι μέθοδοι συγχρονισμού με επαναληπτικές προσπάθειες παρουσιάζουν σημαντικές δυσκολίες σε workloads με υψηλά ποσοστά εγγραφών.

✓ **Synchronization method**

Στο διάγραμμα που παραθέτουμε παρακάτω έχουμε συμπεριλάβει για κάθε μέθοδο το καλύτερο throughput που πετύχαμε χρησιμοποιώντας κάποιο αριθμό νημάτων, για κάθε μέγεθος λίστας και κάθε είδος workload.

Throughput Comparison for Different Methods and Workloads (1024 vs 8192)



Είναι φανερό ότι η μέθοδος που χρησιμοποιούμε επηρεάζει σημαντικά το throughput. Στην ανάλυσή μας παρουσιάσαμε τις μεθόδους με τέτοια σειρά, ώστε κάθε νέα μέθοδος να επιλύει τα προβλήματα της προηγούμενης, κάτι που αποτυπώνεται και στο διάγραμμα. Η μόνη μέθοδος που παρατηρούμε μείωση στο throughput είναι η fine-grain locking και αυτό οφείλεται στο υψηλό κόστος του hand-over-hand locking που απαιτείται για τη διεκπεραίωση κάθε λειτουργίας.

Οι τελευταίες μέθοδοι, lazy synchronization και non-blocking synchronization, αξιοποιούν βελτιστοποιήσεις που τους επιτρέπουν να πετυχαίνουν τις καλύτερες επιδόσεις. Συνολικά, βλέπουμε πως η μέθοδος non-blocking υπερέχει, αποδεικνύοντας ότι η απαλοιφή των κλειδωμάτων συμβάλλει σημαντικά στη βελτίωση της απόδοσης. Η μόνη περίπτωση όπου η μέθοδος lazy synchronization υπερισχύει είναι όταν το workload περιλαμβάνει μόνο αναζητήσεις (λειτουργία contains()), καθώς στην περίπτωση αυτή δεν χρησιμοποιούνται κλειδώματα, και το σύστημα δεν έχει επιπλέον overhead.

Για να γίνει ακόμη πιο εμφανής η σύγκριση των μεθόδων, θα παρουσιάσουμε και στη μορφή πίνακα την επί τις εκατό βελτίωση που έχει η κάθε μέθοδος σε σχέση με την προηγούμενή της.

Ποσοστά βελτίωσης για list_size=1024

Method/ Workload	Fine-grain Compared to Coarse-grain	Optimistic Compared to Fine-grain	Lazy Compared to Optimistic	Non-blocking Compared to Lazy
100/0/0	↓ 67.35%	↑ 55.12%	↑ 10832.58%	↓ 52.74%
80/10/10	↓ 73.13%	↑ 424.59%	↑ 243.51%	↑ 14.62%
20/40/40	↓ 68.26%	↑ 394.50%	↑ 153.62%	↑ 32.75%
0/50/50	↓ 64.42%	↑ 388.36%	↑ 124.73%	↑ 45.80%

Ποσοστά βελτίωσης για list_size=8192

Method/ Workload	Fine-grain Compared to Coarse-grain	Optimistic Compared to Fine-grain	Lazy Compared to Optimistic	Non-blocking Compared to Lazy
100/0/0	↑ 6.05%	↓ 16.86%	↑ 6073.66%	↓ 39.04%
80/10/10	↑ 25.00%	↑ 383.03%	↑ 105.60%	↑ 6.70%
20/40/40	↑ 4.77%	↑ 361.98%	↑ 117.11%	↑ 15.95%
0/50/50	↑ 0.46%	↑ 327.33%	↑ 126.47%	↑ 8.53%

3^η Εργαστηριακή Άσκηση

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Οι κάρτες γραφικών είναι σχεδιασμένες για προβλήματα με υψηλό παραλληλισμό, πολλά δεδομένα και πολλούς υπολογισμούς ανά δεδομένο. Στόχος τους είναι η επιτάχυνση data-parallel και compute-intensive εφαρμογών, χρησιμοποιώντας περισσότερους αλλά απλούστερους πυρήνες. Η αρχιτεκτονική τους περιλαμβάνει μείωση της κρυφής μνήμης και απλή χρήση της, ενώ το hardware thread scheduling επιτρέπει την αποτελεσματική διαχείριση νημάτων με σχεδόν μηδενικό κόστος συγχρονισμού. Επιπλέον, προσφέρουν υψηλό εύρος ζώνης στην κύρια μνήμη, με περιορισμένη όμως χωρητικότητα.

Στο πλαίσιο αυτό, στην παρούσα άσκηση θα μελετήσουμε και θα βελτιστοποιήσουμε τον αλγόριθμο K-means σε μια κάρτα γραφικών NVIDIA με τη χρήση του εργαλείου CUDA. Συγκεκριμένα, θα δημιουργήσουμε αρχικά μια υλοποίηση στην οποία θα αναθέσουμε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι του αλγορίθμου: τον υπολογισμό των nearest clusters σε κάθε επανάληψη. Στη συνέχεια, θα δοκιμάσουμε κάποιες συνήθεις βελτιώσεις επίδοσης για κάρτες γραφικών και θα αξιολογήσουμε την τελική απόδοση του παράλληλου προγράμματος.

Αναλυτικότερα, πρώτα θα τροποποιήσουμε τη δομή των δεδομένων από row-based σε column-based indexing, προκειμένου να βιοθήσουμε την GPU στη λήψη των δεδομένων από την DRAM της. Δεύτερον, θα τοποθετήσουμε τα clusters στη μοιραζόμενη μνήμη της GPU, ώστε τα στοιχεία κάθε block να έχουν ταχύτερη πρόσβαση σε αυτά. Τρίτον, θα υλοποιήσουμε ολόκληρο τον αλγόριθμο K-means στην GPU, δηλαδή θα προσθέσουμε εκεί και τον υπολογισμό των νέων cluster centroids, προσπαθώντας να εξοικονομήσουμε χρόνο στις πράξεις. Τέλος, θα υπολογίσουμε το delta με reduction αντί για global atomics, καθώς το atomicAdd είναι ακριβό όταν εκτελείται από πολλά threads.

Για να αξιολογήσουμε την κλιμακωσιμότητα της παραλληλοποίησης, θα συγκεντρώσουμε χρόνους εκτέλεσης για διάφορες τιμές block size στην GPU. Συγκεκριμένα, θα πραγματοποιήσουμε μετρήσεις για **bsize = {32, 48, 64, 128, 238, 512, 1024}**.

Ωστόσο, παρατηρούμε ότι ορισμένες από τις τιμές δεν είναι πολλαπλάσια του 32, γεγονός που δεν επιτρέπει τη βέλτιστη αξιοποίηση των πόρων της GPU. Η αρχιτεκτονική της GPU βασίζεται σε warps των 32 threads, τα οποία εκτελούνται

παράλληλα. Όταν επιλέγουμε bsize που δεν είναι πολλαπλάσιο του 32, η GPU δεσμεύει επιπλέον threads ώστε να συμπληρώσει πλήρη warps, αφήνοντας τα περιττά threads ανενεργά.

Ειδικές περιπτώσεις που οδηγούν σε αναποτελεσματική χρήση των threads:

- bsize = 48: Δημιουργούνται δύο warps, εκ των οποίων το δεύτερο έχει 16 ανενεργά threads.
- bsize = 238: Παράγονται 8 warps, αλλά το τελευταίο έχει 18 ανενεργά threads.

Αναμένουμε ότι αυτές οι τιμές του bsize θα προκαλέσουν ακανόνιστη κλιμάκωση, καθώς δεν εκμεταλλεύονται πλήρως τους υπολογιστικούς πόρους της GPU.

Ειδικότερα, αναμένουμε ότι το bsize = 64 θα έχει σημαντικά καλύτερη απόδοση σε σχέση με το bsize = 48, καθώς και τα δύο δεσμεύουν τον ίδιο αριθμό warps, αλλά στη δεύτερη περίπτωση ένα σημαντικό ποσοστό των threads παραμένει ανενεργό.

Από τα διαγράμματα που θα δημιουργήσουμε, θα είναι εμφανής η διαφορά στον χρόνο εκτέλεσης και στο speedup μεταξύ αυτών των περιπτώσεων, επιβεβαιώνοντας τη σημασία της σωστής επιλογής του block size για τη μέγιστη αξιοποίηση των πόρων της GPU.

1^η υλοποίηση: Naive version

Σε αυτήν την έκδοση του αλγορίθμου, αναθέτουμε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι, δηλαδή τον υπολογισμό των nearest clusters (τον εντοπισμό του κοντινότερου κέντρου για κάθε αντικείμενο) σε κάθε επανάληψη. Για να το πετύχουμε αυτό, αρχικά υλοποιήσαμε τη συνάρτηση που υπολογίζει το καθολικό ID ενός thread (global TID) σε μονοδιάστατο block και grid, το οποίο χρησιμοποιείται για την ανάθεση δεδομένων στα threads. Δεδομένου ότι το CUDA δεν παρέχει απευθείας αυτόν τον υπολογισμό, ορίσαμε το global TID ως:

$$\text{global TID} = \text{local_tid} + \text{blockDim.x} * \text{blockIdx.x}$$

όπου $\text{local_tid} = \text{threadIdx.x}$.

Στη συνέχεια, υλοποιήσαμε τη συνάρτηση euclid_dist_2, η οποία επιστρέφει την ευκλείδεια απόσταση στο τετράγωνο μεταξύ ενός αντικειμένου και ενός κέντρου. Αυτές οι δύο βοηθητικές συναρτήσεις χρησιμοποιούνται για την εκτέλεση στην GPU.

Κατόπιν, προχωρήσαμε στην υλοποίηση του πυρήνα find_nearest_cluster. Σε αυτόν, όλα τα threads για τα οποία ισχύει $\text{tid} < \text{numObjs}$, υπολογίζουν το ID του

κοντινότερου cluster για το αντίστοιχο αντικείμενο, αξιοποιώντας την euclid_dist_2 με objectId = tid και clusterId το κάθε cluster που εξετάζεται (i, όπου το i κυμαίνεται από 0 έως numClusters).

Για τη μεταβλητή devdelta, η οποία «κρατάει» τον αριθμό των αντικειμένων που αλλάζουν cluster σε κάθε επανάληψη (χρησιμοποιείται ως κριτήριο σύγκλισης συγκρινόμενη με ένα threshold), εφαρμόσαμε την εντολή atomicAdd ώστε να αποφεύγονται συγκρούσεις μεταξύ threads που προσπαθούν ταυτόχρονα να την ενημερώσουν.

Για τη διάσταση του grid, δηλαδή τον αριθμό των blocks, χρησιμοποιήσαμε τη σχέση:

$(\text{numObjs} + \text{numThreadsPerClusterBlock} - 1) / \text{numThreadsPerClusterBlock}$,

όπου numObjs είναι ο συνολικός αριθμός αντικειμένων και numThreadsPerClusterBlock ο αριθμός των threads ανά block, ο οποίος περιορίζεται είτε από το μέγεθος του block (blockSize) είτε από τον αριθμό των αντικειμένων (αν είναι μικρότερος από το blockSize). Ο όρος numThreadsPerClusterBlock – 1 προστίθεται για να εξασφαλιστεί σωστή στρογγυλοποίηση προς τα πάνω στη διαίρεση. Έτσι, όλα τα αντικείμενα κατανέμονται σε threads, ακόμα και αν η διαίρεση δεν είναι τέλεια, εξασφαλίζοντας ότι δεν θα υπάρχουν αχρησιμοποίητοι πόροι (threads) στην GPU.

Μέσα στο while loop, χρειάστηκε να πραγματοποιήσουμε τις απαραίτητες μεταφορές δεδομένων, καθώς ο προγραμματισμός για κάρτες γραφικών απαιτεί πρώτα την αντιγραφή των δεδομένων εισόδου από τη CPU στην GPU και, μετά την εκτέλεση των kernels, την αντιγραφή των αποτελεσμάτων από την GPU πίσω στη CPU. Για να μεταφέρουμε τα δεδομένα των clusters από τη CPU στην GPU, η οποία αναλαμβάνει τους υπολογισμούς, χρησιμοποιήσαμε την εντολή:

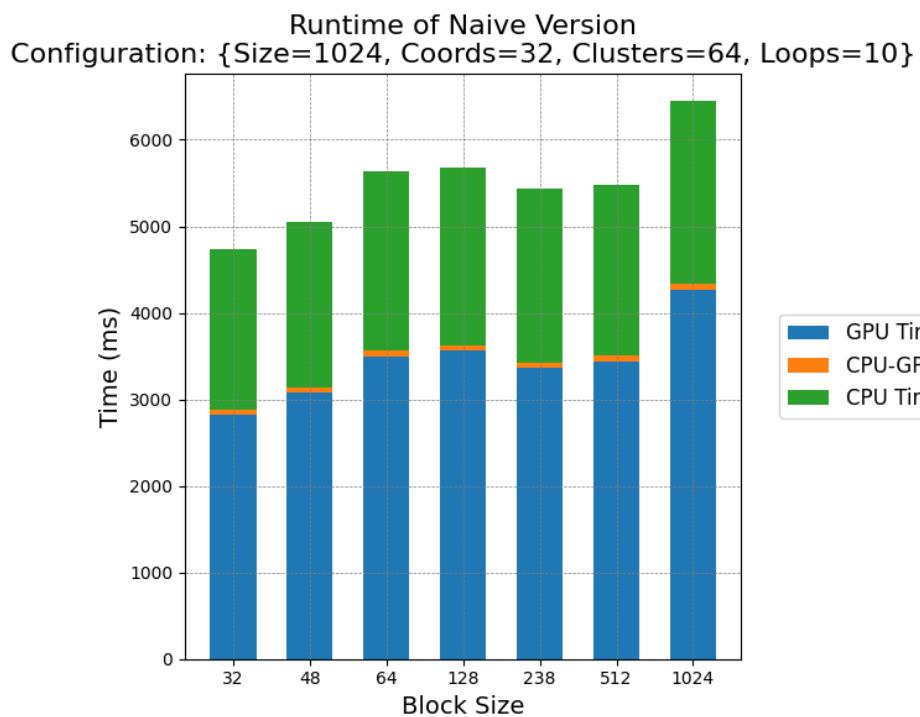
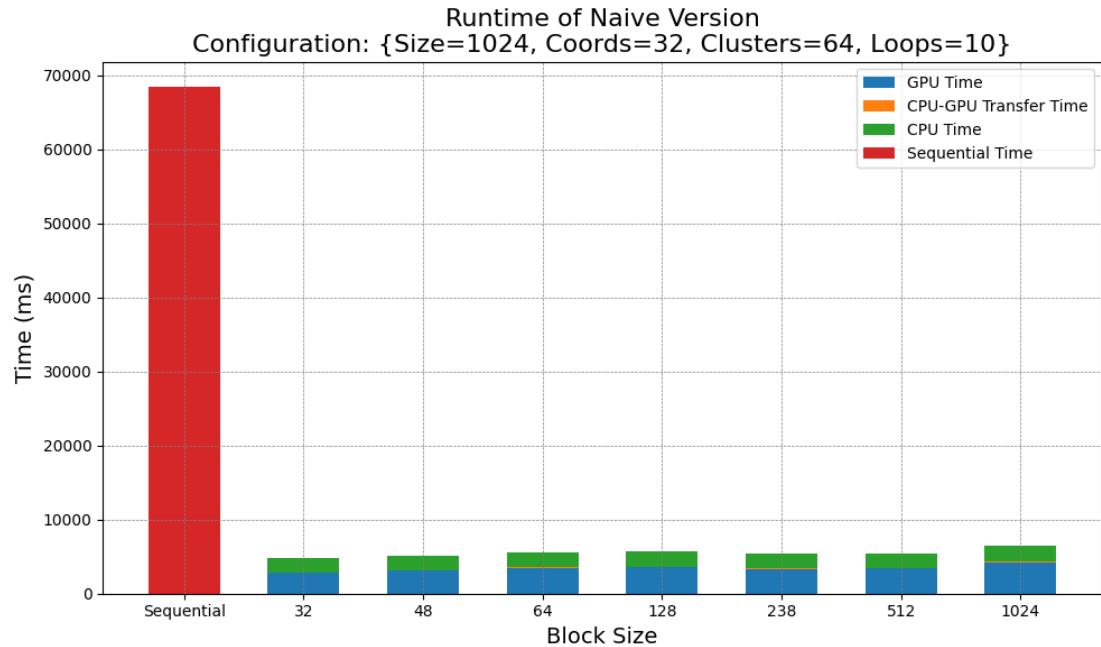
```
checkCuda(cudaMemcpy(deviceClusters, clusters, numClusters * numCoords * sizeof(double), cudaMemcpyHostToDevice)).
```

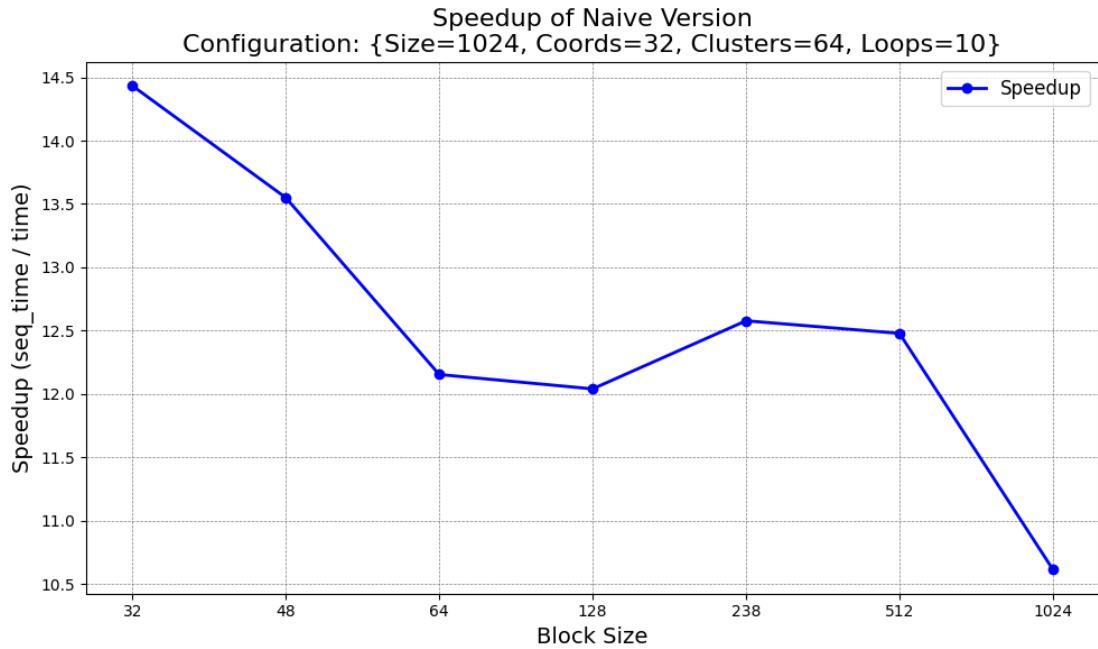
Μετά την εκτέλεση του find_nearest_cluster, πραγματοποιήσαμε τις αντίστοιχες μεταφορές δεδομένων πίσω στη CPU. Συγκεκριμένα, αντιγράψαμε τη μεταβλητή deviceMembership, η οποία περιέχει την πληροφορία σχετικά με το σε ποιο cluster ανήκει κάθε αντικείμενο, ώστε να υπολογιστούν στην CPU τα νέα κέντρα. Παράλληλα, μεταφέραμε και τη μεταβλητή dev_delta_ptr για να γίνει ο έλεγχος της σύγκλισης.

Παρακάτω παρουσιάζεται το bar plot διάγραμμα που απεικονίζει τον χρόνο εκτέλεσης για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}. Συμπεριλαμβάνονται δύο εκδοχές: ένα διάγραμμα που περιλαμβάνει τη σειριακή

έκδοση και ένα χωρίς αυτήν, για μεγαλύτερη ευκρίνεια. Χρησιμοποιούνται stacked bar plots με τρία διακριτά επίπεδα: τον χρόνο εκτέλεσης στην GPU, τον συνολικό χρόνο μεταφορών δεδομένων και τον χρόνο εκτέλεσης των υπόλοιπων υπολογισμών στη CPU.

Επιπλέον, παρατίθεται το αντίστοιχο διάγραμμα speedup, στο οποίο το speedup υπολογίζεται ως sequential time / total time.





Σχολιασμός και συμπεράσματα

Η παράλληλη έκδοση παρουσιάζει σημαντική επιτάχυνση σε σχέση με τη σειριακή, όπως καταδεικνύεται στα διαγράμματα. Το speedup ξεπερνά το 10x για όλα τα μεγέθη block, ενώ η σειριακή έκδοση απαιτεί πολύ περισσότερο χρόνο (περίπου 70 δευτερόλεπτα). Αυτό επιβεβαιώνει ότι ο υπολογισμός των nearest clusters μπορεί να αξιοποιήσει την παράλληλη επεξεργασία της GPU.

Ο αλγόριθμος K-means συχνά θεωρείται ιδανικός για GPUs, καθώς το πιο υπολογιστικά απαιτητικό μέρος του – η ανάθεση σημείων στα κοντινότερα clusters – μπορεί να γίνει ανεξάρτητα για κάθε σημείο. Επιπλέον, περιλαμβάνει επαναλαμβανόμενες αριθμητικές πράξεις (όπως ο υπολογισμός ευκλείδειας απόστασης) και χαρακτηρίζεται από πολλές επαναλήψεις με σταθερό μοτίβο.

Παρ’ όλα αυτά, η υλοποίησή μας δεν εκμεταλλεύεται πλήρως τις δυνατότητες της GPU. Πρώτον, η μνήμη δεν χρησιμοποιείται αποδοτικά. Οι προσβάσεις στη μνήμη είναι μη coalesced, μειώνοντας την απόδοση, ενώ η shared memory, που θα μπορούσε να αποθηκεύσει συχνά χρησιμοποιούμενα δεδομένα, όπως τα κέντρα των clusters, για ταχύτερη πρόσβαση, παραμένει ανεκμετάλλευτη. Επιπλέον, η παραλληλοποίηση του αλγορίθμου δεν είναι πλήρης, καθώς ο υπολογισμός των νέων κέντρων εκτελείται σειριακά.

Τα ζητήματα αυτά αποτελούν περιορισμούς της παρούσας έκδοσης και θα προσπαθήσουμε να τα αντιμετωπίσουμε στις επόμενες υλοποιήσεις.

Η επίδοση της παράλληλης έκδοσης επηρεάζεται από το μέγεθος του block size, όπως αποδεικνύεται από τα διαγράμματα χρόνου εκτέλεσης και speedup. Για μικρά block sizes (π.χ. 32 ή 48), η GPU αξιοποιείται πιο αποδοτικά, οδηγώντας σε υψηλότερο speedup και μικρότερους χρόνους εκτέλεσης. Αντίθετα καθώς αυξάνεται το block size (π.χ. σε 1024), η απόδοση μειώνεται.

Αυτή η συμπεριφορά μπορεί να αποδοθεί κυρίως στο global memory latency. Ο αλγόριθμος K-means, για κάθε σημείο δεδομένων, υπολογίζει την ευκλείδεια απόσταση από κάθε κέντρο, γεγονός που απαιτεί επαναλαμβανόμενες προσπελάσεις στη μνήμη για τη φόρτωση των δεδομένων σημείων και κέντρων. Χωρίς την αξιοποίηση της shared memory, η GPU αναγκάζεται να ανακτά αυτά τα δεδομένα από την global memory για κάθε υπολογισμό. Όταν το block size είναι μεγάλο, περισσότερα threads του ίδιου block προσπαθούν ταυτόχρονα να προσπελάσουν την global memory, προκαλώντας congestion και αυξημένο latency, ειδικά στην περίπτωσή μας που οι προσπελάσεις δεν είναι συντονισμένες και παραμένουν μη coalesced.

Επιπλέον, η αύξηση του block size μπορεί να οδηγήσει σε περισσότερες συγκρούσεις κατά την ενημέρωση κοινών μεταβλητών, όπως το dev_delta, εξαιτίας του κόστους της εντολής atomicAdd, το οποίο επιβαρύνει τη συνολική εκτέλεση. Σε εξαιρετικά μεγάλα block sizes υπάρχει επίσης ο κίνδυνος ο αριθμός των threads που εκτελούνται ταυτόχρονα να υπερβεί τις δυνατότητες της GPU, οδηγώντας σε υποεκμετάλλευση της παράλληλης επεξεργασίας.

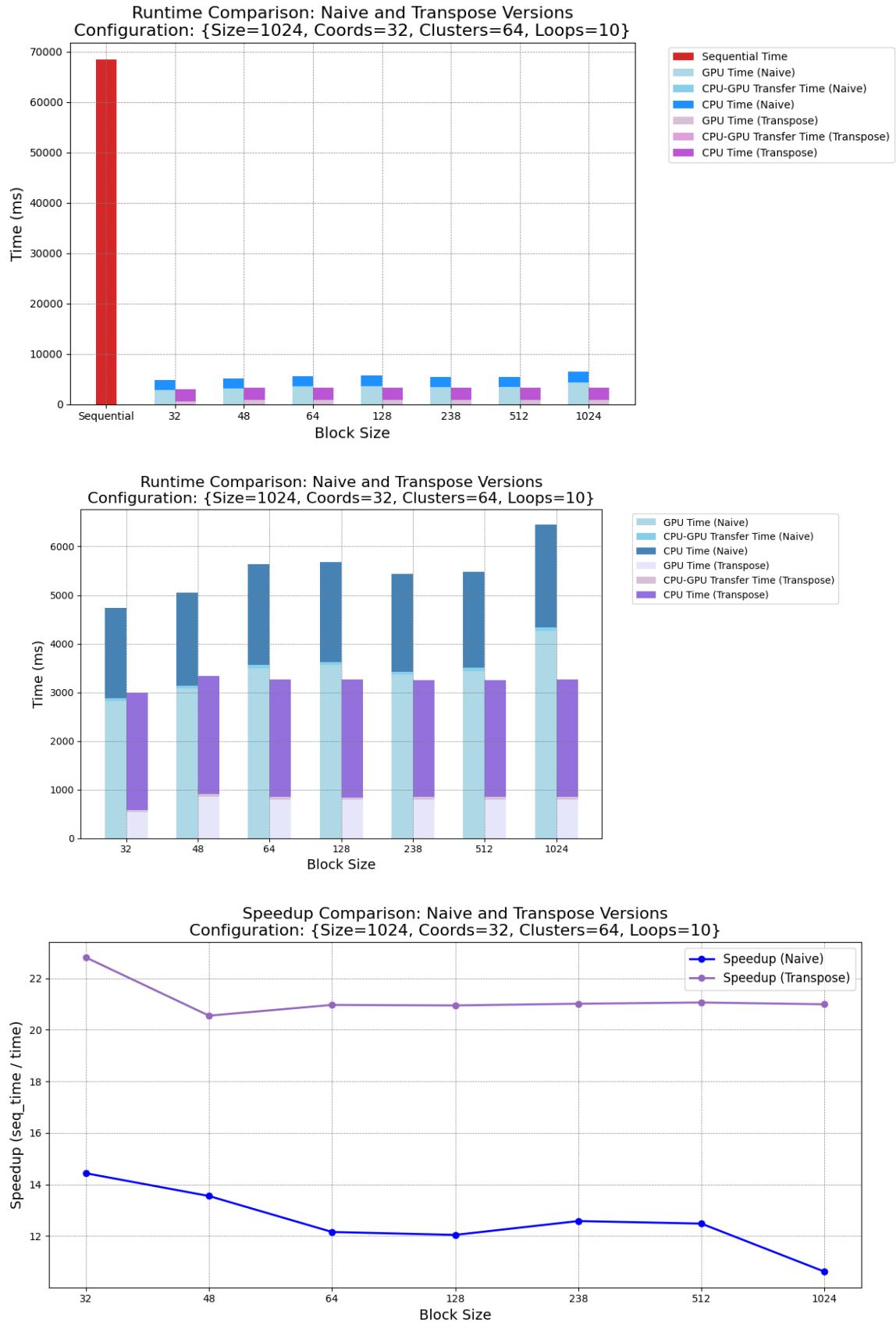
Αντίθετα, μικρότερα block sizes επιτρέπουν μια πιο αποτελεσματική κατανομή των threads και μια καλύτερη διαχείριση των προσπελάσεων στη μνήμη, με αποτέλεσμα υψηλότερο speedup και καλύτερη συνολική απόδοση.

Τέλος, αξίζει να σημειωθεί ότι οι μεταφορές δεδομένων δεν επηρεάζονται σημαντικά από το μέγεθος του block, διατηρώντας το κόστος τους σχετικά σταθερό.

2^η υλοποίηση: Transposed version

Στην transpose έκδοση, η δομή των δεδομένων αλλάζει από row-based σε column-based indexing, δηλαδή υλοποιείται ουσιαστικά όπως η naive, αλλά με την αντιστροφή του indexing στους πίνακες. Συγκεκριμένα, υλοποιήσαμε τη συνάρτηση euclid_dist_2_transpose, η οποία ακολουθεί την ίδια λογική με την naive έκδοση, αλλά προσπελάζει τους πίνακες με column-based format. Παράλληλα, φροντίσαμε για τη σωστή αρχικοποίηση των δεδομένων και την μετατροπή τους στην νέα μορφή.

Ακολουθούν τα αποτελέσματα που προέκυψαν από αυτήν την προσέγγιση.



Σχολιασμός και συμπεράσματα

Η transpose έκδοση παρουσιάζει σημαντική βελτίωση στην απόδοση σε σχέση με την naïve, όπως καταδεικνύεται από τα διαγράμματα speedup και χρόνου εκτέλεσης. Η κύρια αιτία αυτής της διαφοράς είναι ο τρόπος με τον οποίο τα threads προσπελαύνουν τη μνήμη. Στην naïve έκδοση, η διάταξη των δεδομένων είναι row-based, με αποτέλεσμα τα threads ενός warp (ομάδα 32 threads) να προσπελαύνουν μη συνεχόμενες περιοχές στη μνήμη. Συγκεκριμένα, κατά τον υπολογισμό της ευκλείδειας απόστασης, τα threads χρησιμοποιούν το ίδιο coordinate για διαφορετικά objects (αντίστοιχα και για τα clusters). Ωστόσο, στην row-based διάταξη, τα δεδομένα για κάθε object είναι τοποθετημένα το ένα δίπλα στο άλλο στη μνήμη. Αυτό σημαίνει ότι οι διαφορετικές συντεταγμένες του ίδιου object βρίσκονται σε συνεχόμενες θέσεις, ενώ οι ίδιες συντεταγμένες για διαφορετικά objects αποθηκεύονται σε μη συνεχόμενες. Αυτό οδηγεί σε αυξημένο latency λόγω των πολλαπλών memory transactions. Αντίθετα, στην transpose έκδοση, η διάταξη των δεδομένων είναι column-based, πράγμα που επιτρέπει στα threads ενός warp να προσπελάσουν δεδομένα σε συνεχόμενες θέσεις μνήμης, εκμεταλλευόμενα την coalesced memory access. Έτσι, η transpose έκδοση αξιοποιεί πιο αποδοτικά την αρχιτεκτονική της GPU, μειώνοντας σημαντικά το κόστος πρόσβασης στη μνήμη και βελτιώνοντας την απόδοση.

Παρ' όλα αυτά, αν και ο χρόνος εκτέλεσης στην GPU μειώνεται αισθητά, ο χρόνος στην CPU παρουσιάζει μια μικρή αύξηση. Αυτό συμβαίνει γιατί με την αντιστροφή των διαστάσεων των πινάκων καταστρέφεται το locality, το οποίο η CPU εκμεταλλεύόταν μέσω των caches της, με αποτέλεσμα να προκύπτει μια επιβάρυνση στην απόδοσή της.

Αναφορικά με τον ρόλο του block size, παρατηρούμε διαφορετική συμπεριφορά στις δύο εκδόσεις. Στην naïve έκδοση, η επίδραση του block size στην απόδοση είναι πιο έντονη, καθώς η μη coalesced πρόσβαση στη μνήμη μπορεί να επιδεινωθεί με την αύξηση του μεγέθους του block. Αντίθετα, στην transpose έκδοση, η βελτιωμένη διαχείριση της μνήμης οδηγεί σε πιο σταθερή απόδοση, ανεξαρτήτως του block size. Αυτό οφείλεται στο γεγονός ότι η πρόσβαση στη μνήμη παραμένει coalesced για κάθε μέγεθος block, μειώνοντας έτσι την ευαισθησία της απόδοσης στις παραμέτρους εκτέλεσης.

3^η υλοποίηση: Shared version

Σε αυτή την υλοποίηση θα εκμεταλλευτούμε την μοιραζόμενη μνήμη που προσφέρουν οι GPUs, η οποία είναι προσβάσιμη από όλα τα νήματα ενός block. Η μοιραζόμενη μνήμη λειτουργεί ως ένας γρήγορος προσωρινός χώρος αποθήκευσης, επιτρέποντας στα νήματα να μοιράζονται δεδομένα χωρίς την ανάγκη συνεχών προσβάσεων στη βασική μνήμη (global memory). Ως αποτέλεσμα, μειώνει τον δαπανούμενο χρόνο για ανάγνωση και εγγραφή δεδομένων και συνάμα βελτιστοποιεί την απόδοση των παραλληλων υπολογισμών.

Επιπλέον πλεονέκτημά της είναι το γεγονός ότι διευκολύνεται ο συγχρονισμός των νημάτων. Καθώς όλα τα νήματα ενός block έχουν ταυτόχρονη πρόσβαση στα δεδομένα στη shared memory, μπορούν να συντονίζουν τις ενέργειές τους πιο αποτελεσματικά. Αυτό είναι ιδιαίτερα σημαντικό σε εφαρμογές όπως ο αλγόριθμος K-Means, όπου τα νήματα χρειάζεται να ενημερώνουν κοινές μεταβλητές ή να μοιράζονται αποτελέσματα υπολογισμών. Η ταχεία πρόσβαση και η κοινή χρήση δεδομένων μειώνουν την ανάγκη για αργές συναρτήσεις

συγχρονισμού, επιτρέποντας στους υπολογισμούς να εκτελούνται πιο ομαλά και γρήγορα.

Επεξηγούμε στη συνέχεια τον κώδικα που υλοποιήσαμε γι' αυτή την έκδοση.

Αρχικά χρειάζεται να δηλώσουμε την μοιραζόμενη μνήμη ως εξής

`extern __shared__ double shmemClusters[];`

Στην συνέχεια πρέπει τα δεδομένα των κέντρων των clusters (device clusters) να αντιγραφούν στην GPU:

```
int local_tid = threadIdx.x;

for (int i = local_tid; i < numClusters; i += blockDim.x) {
    for(int j = 0; j < numCoords; j++) {
        shmemClusters[j * numClusters + i] = deviceClusters[j * numClusters + i];
    }
}
```

Χρησιμοποιούμε τα local_tids ώστε να κατανείμουμε την εργασία μεταξύ των threads. Αντί να αφήσουμε ένα νήμα να κάνει την αντιγραφή όλων των δεδομένων μόνο, χωρίζουμε την αντιγραφή τμηματικά σε πολλά νήματα, παραλληλοποιώντας την. Με τον πρώτο βρόχο (stride loop) εξασφαλίζουμε ότι κάθε thread, ξεκινώντας από το δικό του local tid, θα αντιγράψει περισσότερο από 1 στοιχείο (αν πληρείται η συνθήκη ότι τα clusters είναι περισσότερα από τον αριθμό νημάτων στο block). Με το nested loop για το εκάστοτε cluster κάθε νήμα αντιγράφει τις

συντεταγμένες. Έτσι κάθε νήμα αντιγράφει μέρος των δεδομένων στην μοιραζόμενη μνήμη αποτελεσματικά.

Για να προχωρήσουμε πρέπει να σιγουρευτούμε ότι όλες οι προηγούμενες εργασίες έχουν έρθει εις πέρας οπότε καλούμε σε συγχρονισμό μέσω της `_syncthreads()`.

Μετά τον συγχρονισμό, τα threads μπορούν να χρησιμοποιήσουν τα δεδομένα που βρίσκονται στη μοιραζόμενη μνήμη για τους υπολογισμούς απόστασης. Με την χρήση της `shmemClusters` πλέον τα δεδόμενα προέρχονται από την μοιραζόμενη μνήμη επομένως η πρόσβαση τους είναι πολύ πιο γρήγορη από την `global memory`.

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

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using clusters in shmem*/
    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters, tid, index);

    for (i = 1; i < numClusters; i++) {
        /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId using clusters in shmem*/
        dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters, tid, i);

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

Σημείωση: η συνθήκη `if (tid < numObjs)` είναι απαραίτητη γιατί δεν είναι απαραίτητο όλα τα threads να αντιστοιχούν σε αντικείμενα, οπότε η εκτέλεση από αυτά τα πλεονάζοντα νήματα θα ήταν ανούσια σπατάλη πόρων. Επιπλέον από κάτω ακολουθεί η πρόσθεση του `delta`, η οποία πρέπει να εκτελεστεί ατομικά για να αποφύγουμε τα race condition που θα υπάρξουν.

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

Στο host μέρος του κώδικα θα καθορίσουμε το μέγεθος της shared memory κατά την εκκίνηση του kernel μέσω της `eventoljcs`:

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

Αυτό το μέγεθος υπολογίζεται με βάση τον αριθμό των συντεταγμένων (`numCoords`) και τον αριθμό των clusters (`numClusters`), διασφαλίζοντας ότι υπάρχει αρκετή μνήμη για την αποθήκευση όλων των κέντρων των clusters σε column-based μορφή.

Αμέσως πριν την εκκίνηση του kernel θα βεβαιωθούμε ότι δεν υπερβαίνουμε την διαθέσιμη μνήμη της GPU

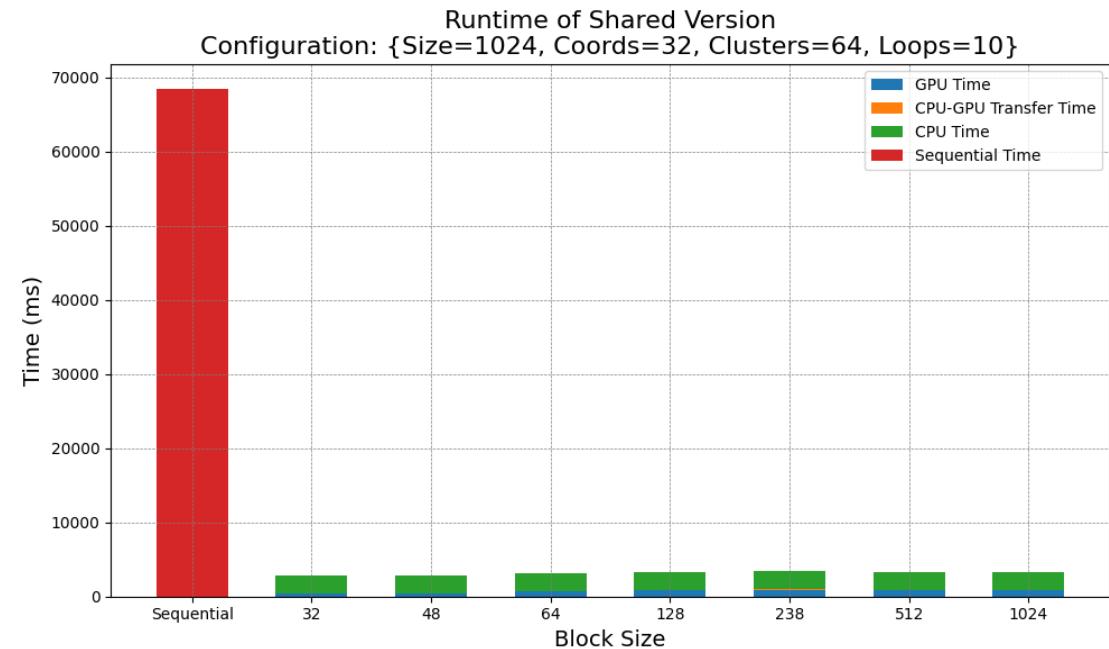
```
cudaDeviceProp deviceProp;
int deviceNum;
cudaGetDevice(&deviceNum);
cudaGetDeviceProperties(&deviceProp, deviceNum);

if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock) {
    error("Your CUDA hardware has insufficient block shared memory to hold all cluster centroids\n");
}
```

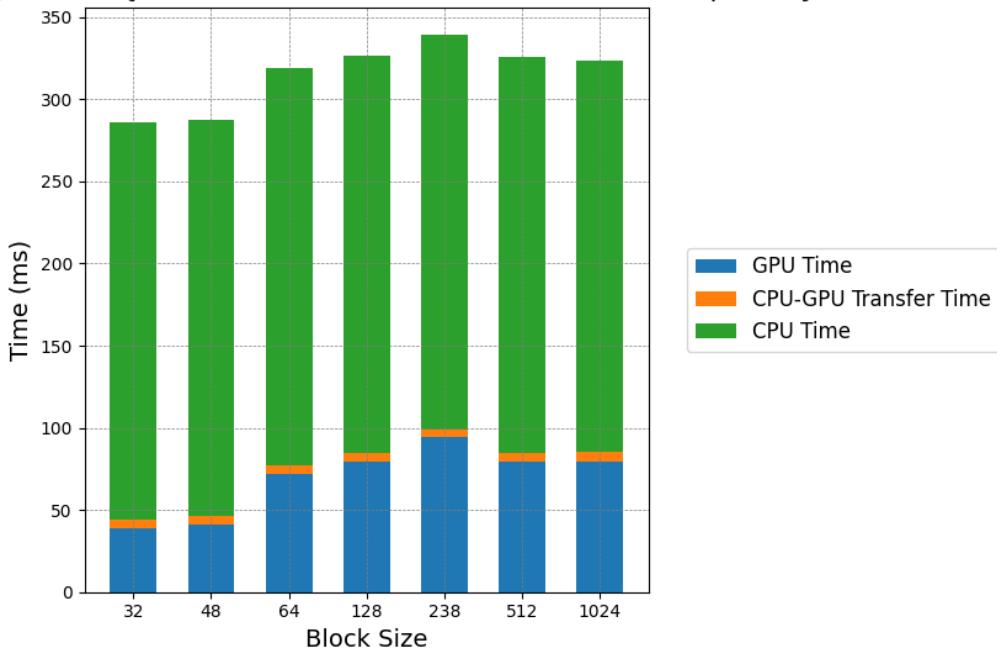
Τέλος, ο kernel εκκινείται:

```
find_nearest_cluster
<<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
    | | | (numCoords, numObjs, numClusters,
    | | | | deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
```

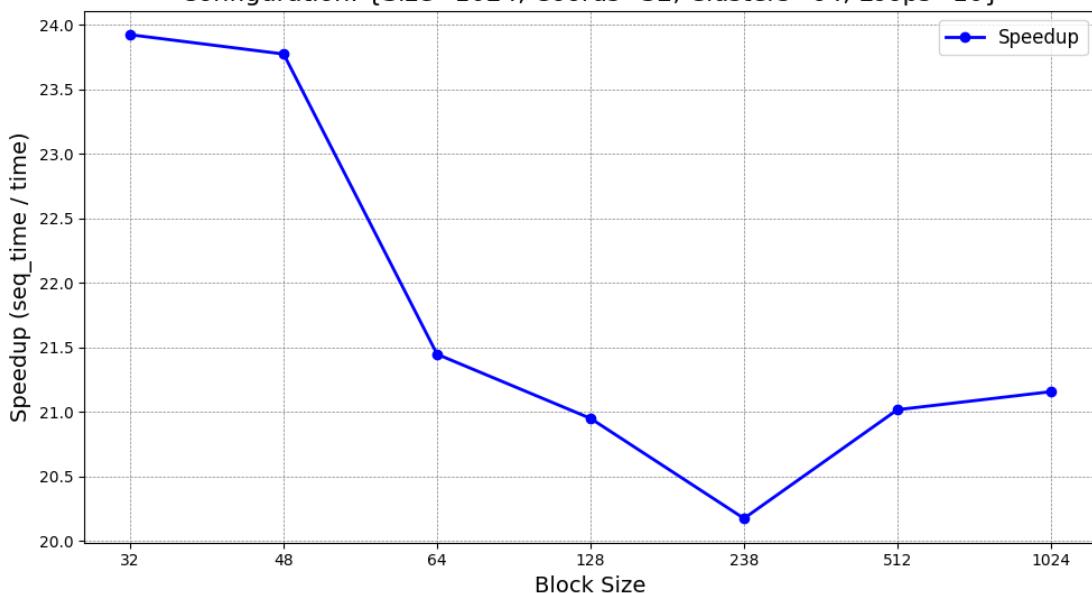
Παραθέτουμε τώρα τα διαγράμματα που πήραμε για τους χρόνους εκτέλεσης και το speedup σε αυτή την υλοποίηση:



Runtime of Shared Version
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



Speedup of Shared Version
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



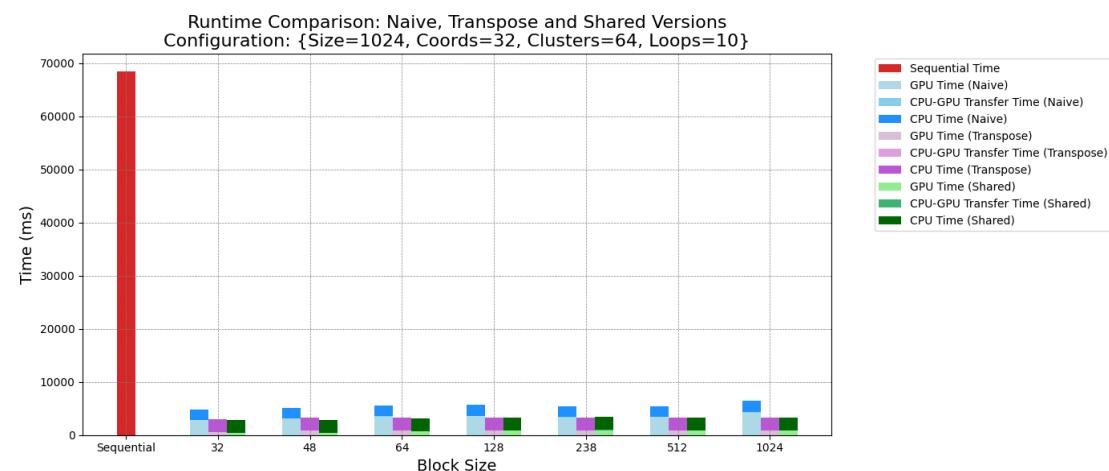
Παρατηρήσεις και συμπεράσματα:

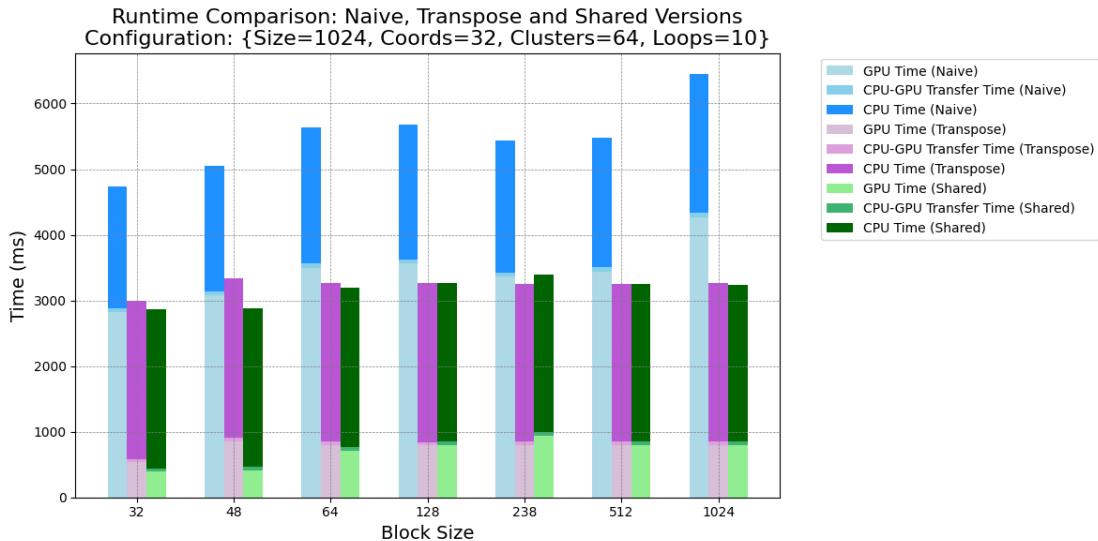
Από τα 2 πρώτα διαγράμματα μπορούμε να διακρίνουμε προφανώς ότι η shared version συνεχίζει να λειτουργεί καλύτερα από τον σειριακό κώδικα, ενώ βλέπουμε πως η **CPU** τείνει να καλύπτει το μεγαλύτερο ποσοστό του χρόνου (πράσινη στήλη), ενώ η **GPU** (μπλε στήλη) έχει σαφώς μικρότερο χρόνο, γεγονός που επιβεβαιώνει ότι το «βαρύ» υπολογιστικό κομμάτι τρέχει αρκετά γρήγορα στη **GPU**. Οι μεταφορές δεδομένων (πορτοκαλί) παραμένουν σχετικά μικρές, συνεπώς δεν αποτελούν βασικό bottleneck.

Από το 3ο διάγραμμα μπορούμε να αποφανθούμε ότι τα μικρά block size αποδίδουν το μέγιστο speedup (έως ~24x). Αυτό υποδηλώνει ότι εκεί επιτυγχάνεται ίσως μια πιο αποδοτική κατανομή των threads, καλύτερη ευθυγράμμιση στη μνήμη ή αξιοποίηση της αρχιτεκτονικής της GPU. Καθώς μεγαλώνει το block size (64, 128, 238), το speedup μειώνεται στην περιοχή του 20-21x. Αυτό μπορεί να οφείλεται είτε σε ελαφρώς μειωμένη occupancy της GPU, είτε σε αύξηση του overhead διαχείρισης για μεγαλύτερα blocks. Για block sizes 512 και 1024, το speedup ανεβαίνει ελαφρά, ωστόσο δεν φτάνει τα επίπεδα των 32 ή 48. Φαίνεται ότι η απόδοση ισορροπεί εκ νέου, υποδηλώνοντας πως για αυτή τη διάσταση προβλήματος δεν υπάρχει μία και μοναδική «βέλτιστη» τιμή block size αλλά αρκετά μεγέθη που οδηγούν σε συγκρίσιμα αποτελέσματα.

Μια ενδιαφέρουσα παρατήρηση στο διάγραμμα speedup είναι η απότομη μείωση που εμφανίζεται στο bsize = 238. Το φαινόμενο αυτό αποτυπώνει την επίδραση της μη βέλτιστης επιλογής block size στην απόδοση, καθώς η τιμή 238 δεν είναι πολλαπλάσιο του 32, οδηγώντας σε υποεκμετάλλευση των warps της GPU. Αν η εκτέλεση είχε πραγματοποιηθεί με bsize = 256, θα αναμέναμε μια ομαλότερη καμπύλη στο διάγραμμα speedup, καθώς η GPU θα αξιοποιούσε πλήρως τους διαθέσιμους υπολογιστικούς πόρους, μειώνοντας τον συνολικό χρόνο εκτέλεσης.

Για την ευκολότερη διεξαγωγή συμπερασμάτων θα παραθέσουμε διαγράμματα σύγκρισης με τις προηγούμενες υλοποιήσεις μας:





Σχολιασμός και συμπεράσματα

Αρχικά αν εξαιρέσουμε τα μπλοκ 128 και 512 που ο συνολικός χρόνος είναι ο ίδιος με του Transpose και του 238 που είναι ελαφρώς μεγαλύτερος του Transpose, σε κάθε άλλη περίπτωση η Shared version έχει τον καλύτερο χρόνο από κάθε άλλη έκδοση.

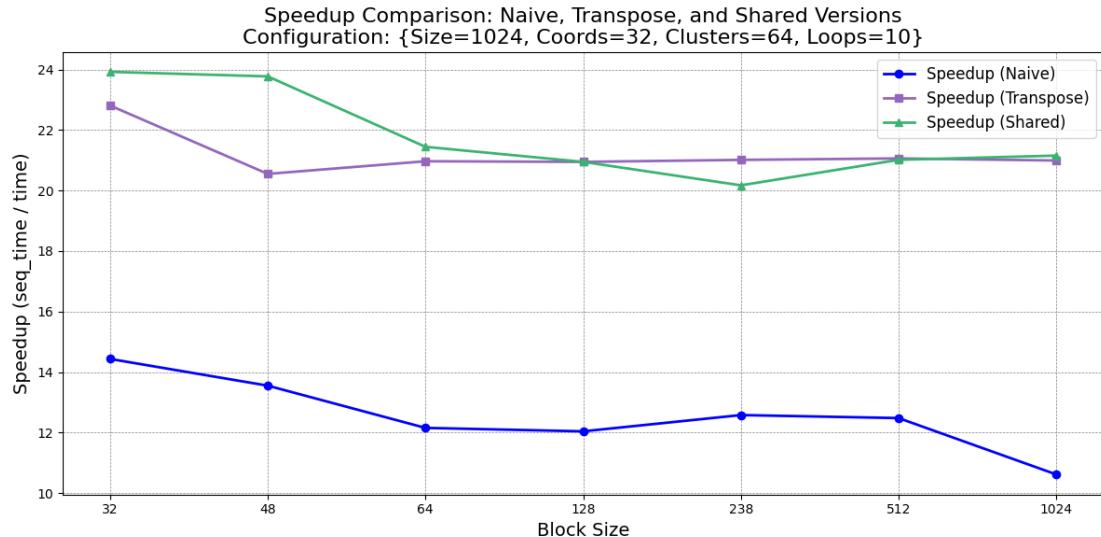
Συγκεκριμένα τώρα μπορούμε να δούμε ότι το **GPU Time (shared)** είναι αισθητά το **μικρότερο** μεταξύ των τριών εκδόσεων σε όλα τα block sizes. Η μεταφορά των κέντρων (clusters) και τμημάτων των δεδομένων στη **μοιραζόμενη μνήμη (shared memory)** μειώνει ακόμα περισσότερο τις καθυστερήσεις πρόσβασης στην global memory. Το **CPU Time (shared)** παραμένει το ίδιο (περίπου) με τον χρόνο του Transpose καθώς σε επίπεδο CPU δεν διαφέρει κάτι από την Transpose παρά μόνο η μεταφορά των δεδομένων και ο υπολογισμός τους στην μοιραζόμενη μνήμη.

Για **μικρότερα block sizes (32, 48, 64)**, οι αποκλίσεις είναι σαφείς: η Naive υστερεί σημαντικά, η Transpose βρίσκεται σε ενδιάμεσο επίπεδο, ενώ η Shared έχει τον καλύτερο χρόνο. Στα **μεγαλύτερα** αρχικά παραμένει στάσιμος και ύστερα συνεχίζει η Naive να αυξάνεται, ενώ η Transpose μοιάζει ανθεκτική σε αλλαγές στο block size και η Shared όπως προείπαμε για τα ενδιάμεσα αυξάνεται ενώ για τα μεγάλα ξανά μειώνεται ο χρόνος.

Οι **CPU-GPU Transfer Times** (απαλοί τόνοι για κάθε έκδοση) παραμένουν σχετικά χαμηλοί σε όλες τις περιπτώσεις.

Ένα πιθανό αίτιο για την καλύτερη επίδοση στα μικρότερα block sizes είναι ότι ο συγχρονισμός παραμένει πιο τοπικός: όταν έχουμε περισσότερα blocks με λιγότερα threads ανά block, η διαδικασία αντιγραφής του πίνακα deviceClusters στη shared memory γίνεται πιο παράλληλα και με λιγότερη επιβάρυνση

συγχρονισμού, ενώ, όσο μεγαλώνει το μέγεθος του block, μεγαλώνει και η ανάγκη για πιο global συγχρονισμό στη GPU.



Σχολιασμός και συμπεράσματα

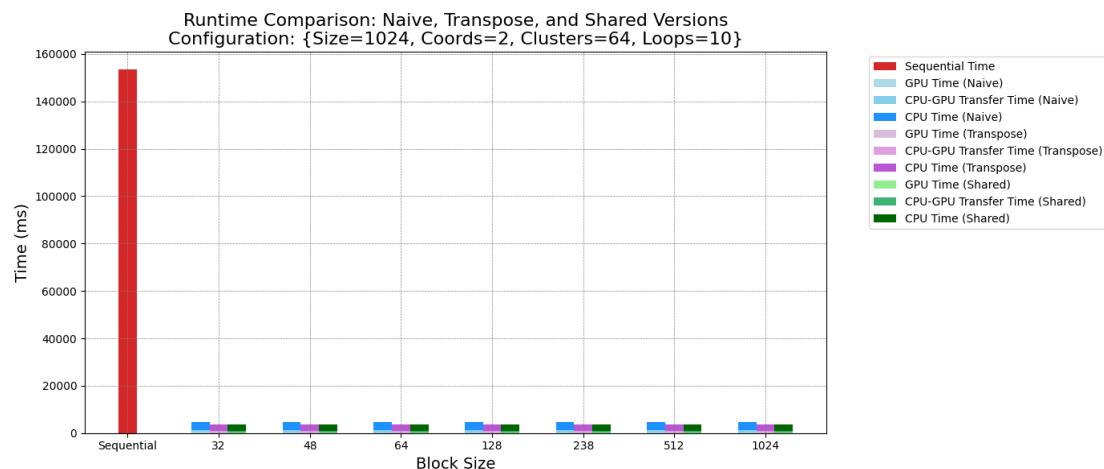
Τέλος, εδώ μπορούμε να επιβεβαιώσουμε ότι το speedup του Naïve μειώνεται με την αύξηση του μεγέθους του μπλοκ, ενώ του transpose για μικρά block sizes μειώνεται μεχρι να σταθεροποιηθεί για 64 και μεγαλύτερες τιμές. Η Shared λαμβάνει την ίδια ισορροπία με την Transpose πέραν της τιμής 238 που βρίσκεται κάτω από αυτήν. Αυτό μπορεί να συμβαίνει για αρκετούς λόγους, όπως ο κορεσμός της GPU για μεγάλο block size. Η GPU μπορεί να «γεμίσει» από άποψη occupancy, δηλαδή έχει αρκετά threads για να κρύβει τις καθυστερήσεις μνήμης, με την βελτίωση που προσφέρει να γίνεται λιγότερο προφανής αφού η Transpose version έχει ήδη καλά ευθυγραμμισμένα τα δεδομένα και εκτελείται αποτελεσματικά. Επιπλέον υπάρχει και ένα μικρό overhead που προκαλείται από την διαχείριση της κοινής μνήμης, που έχει ανάγκη από επιπλέον συγχρονισμούς (`__syncthreads()`). Όταν το block size αυξάνεται αρκετά, το όφελος του ταχύτερου access αρχίζει να αντισταθμίζεται από άλλους παράγοντες, με αποτέλεσμα η καμπύλη της Shared να «συναντά» αυτή της Transpose.

Άρα μετά από το block size 64 η Transpose έχει ήδη λύσει αρκετά ζητήματα ασύμφορων προσβάσεων, ενώ η Shared δεν κερδίζει τόσο παραπάνω σε σχέση με το overhead που συνεπάγεται. Έτσι οι δύο καμπύλες συγκλίνουν και έχουν παρόμοιο speedup.

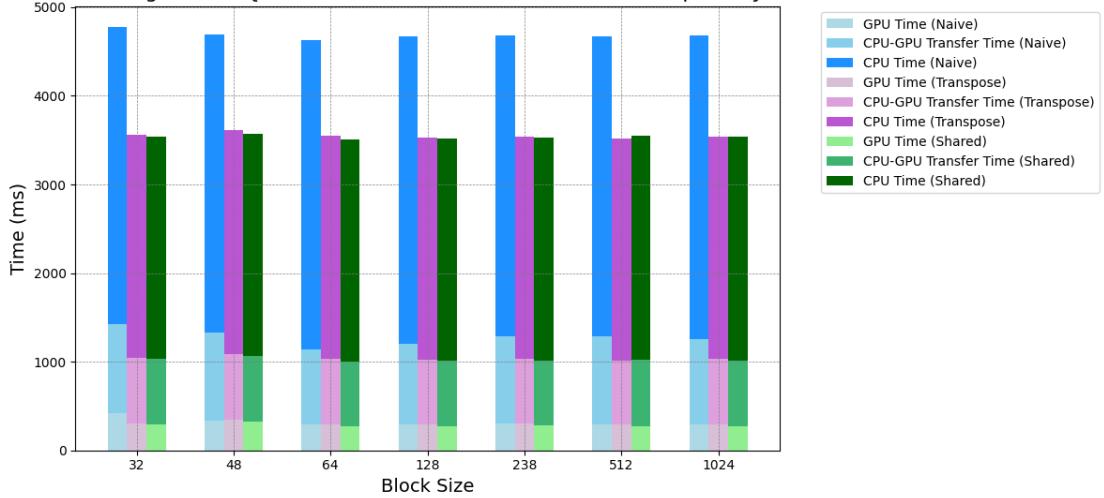
Σύγκριση υλοποιήσεων / bottleneck analysis

Με βάση τους δεχτικούς timer (GPU part, CPU-GPU transfers, CPU part) εντός του while loop, έχουμε καταλήξει στα παραπάνω διαγράμματα. Όπως μπορούμε να δούμε στο μέχρι πρότινο σενάριο, ο naive αλγόριθμος είχε πολύ μεγαλύτερο χρόνο στην GPU σε σχέση με τους αλλους 2, κάτι που όπως προείπαμε οφείλεται κυρίως λόγω των πολυάριθμων ασυνεχών προσβάσεων στη global memory (non coalesced accesses), οι οποίες οδηγούν σε περισσότερα, μικρότερα και αργά transactions που κοστίζουν αρκετά. Αντίθετα, ο Transpose βελτιώνει το μοτίβο των προσβάσεων με column-based διάταξη, ώστε η GPU να κάνει πιο coalesced αναγνώσεις, με λιγότερα και μεγαλύτερα transactions. Ωστόσο, εξακολουθεί να εξαρτάται από τη main memory σε κάθε επανάληψη. Η Shared υλοποίηση περιορίζει ακόμη περισσότερο αυτή την επιβάρυνση, καθώς μεταφέρει τα κέντρα των clusters (ή και τμήματα των δεδομένων) σε μοιραζόμενη μνήμη, μειώνοντας έτσι τα κόστη πρόσβασης στη global memory. Το αποτέλεσμα είναι εμφανώς χαμηλότερος GPU χρόνος στα περισσότερα block sizes, κυρίως για μικρότερα blocks όπου ο συγχρονισμός παραμένει πιο “τοπικός”. Για μεγαλύτερα blocks εμφανίζεται επιπρόσθετο overhead αντιγραφής-συγχρονισμού, που μπορεί προσωρινά να ανατρέψει αυτή την τάση, χωρίς όμως να αλλάζει τη βασική αρχή ότι η Shared κερδίζει χάρη στις ταχύτερες προσβάσεις και στη μείωση των αργών transactions.

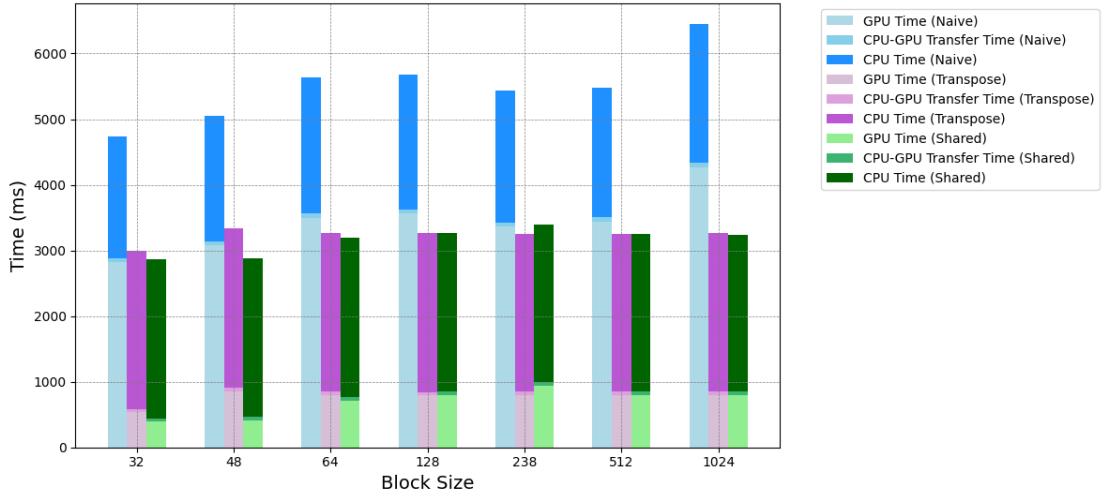
Παραθέτουμε στη συνέχεια τα αντίστοιχα διαγράμματα:



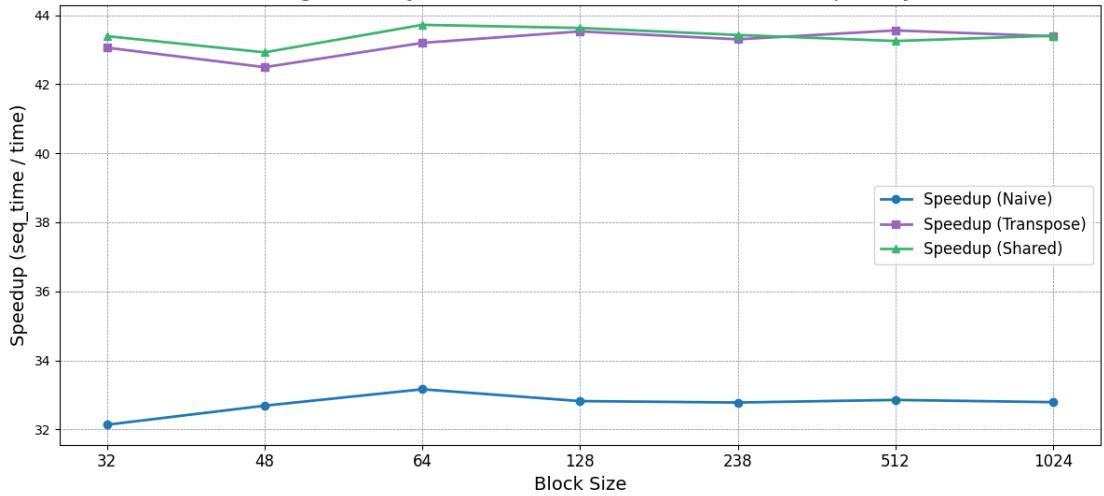
Runtime Comparison: Naive, Transpose, and Shared Versions
Configuration: {Size=1024, Coords=2, Clusters=64, Loops=10}



Runtime Comparison: Naive, Transpose and Shared Versions
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



Speedup Comparison: Naive, Transpose and Shared Versions
Configuration: {Size=1024, Coords=2, Clusters=64, Loops=10}



Σχολιασμός και συμπεράσματα

Δοκιμάζοντας πλέον το νέο configuration μειώνουμε τις διαστάσεις από 32 σε 2 και αμέσως μπορούμε να παρατηρήσουμε τα εξής:

Naive: Εξακολουθεί να έχει αυξημένο GPU Time σε σχέση με Transpose/Shared, αλλά η διαφορά είναι ότι πρώτον παραμένει σε κοντινές τιμές με μικρές διακυμάνσεις για τα διάφορα block sizes και δεύτερον έχει αυξηθεί κατά πολύ ο χρόνος της CPU.

Transpose και Shared: Σε ολα τα block sizes βλέπουμε ότι οι δύο εκδόσεις έρχονται **πολύ κοντά** σε χρόνους. Σε αυτό το configuration και οι 2 υλοποιήσεις διατηρούν σταθερά, ανάμεσα στα block sizes, σταθερούς χρόνος CPU, CPU-GPU Transfer και GPU.

Το ιδιαίτερο που παρατηρούμε στα διαγράμματα είναι ότι ο σειριακός χρόνος αυξήθηκε πάνω από 100% καθώς και στο speedup που κερδίζουμε λόγω της GPU. Επιπλέον στις 2 διαστάσεις έχουμε μεγαλύτερους συνολικούς χρόνους και στις υλοποιήσεις των Transpose και Shared, αύξηση που μοιάζει να οφείλεται στους σημαντικά αυξημένους χρόνους των CPU-GPU Transfers. Ο Naive παραμένει ίδιος (ενώ στις 32 διαστάσεις ο χρόνος αυξανόταν με το block size).

Εφόσον μειώσαμε τον αριθμό των διαστάσεων από 32 σε 2 και διατηρούμε σταθερό το μέγεθος του dataset στα 1024 MB, παρουσιάζεται αύξηση του αριθμού των αντικειμένων, άρα αυξημένοι υπολογισμοί, πράγμα που συνεπάγεται την αύξηση του σειριακού χρόνου. Στην πράξη, το πρόβλημα μετατοπίζεται από ένα περισσότερο **compute-bound** σενάριο (πολλές πράξεις πολλαπλασιασμού/πρόσθεσης για την απόσταση σε πολλές διαστάσεις) σε ένα περισσότερο **memory-bound** σενάριο, καθώς αυξάνεται ο αριθμός των αντικειμένων που πρέπει να φορτωθούν και να ενημερωθούν σε κάθε επανάληψη. Ενώ οι υπολογισμοί ανά αντικείμενο γίνονται πολύ απλούστεροι, το πλήθος των αντικειμένων (και άρα οι αναγκαίες προσβάσεις στη μνήμη) αυξάνεται, οπότε δεν υπάρχουν αρκετές, πιο απαιτητικές, πράξεις για να καλυφθούν οι επιπλέον καθυστερήσεις μνήμης. Να σημειωθεί ότι ακόμα και στη shared memory, αυτό το φαινόμενο δεν εξαλείφεται εντελώς, διότι όσο περισσότεροι είναι οι κύκλοι επικοινωνίας και ενημέρωσης δεδομένων, τόσο μεγαλώνει ο σειριακός χρόνος και οι επιμέρους καθυστερήσεις μεταφοράς, ιδιαίτερα όταν το dataset παραμένει σταθερά μεγάλο σε μέγεθος.

Πιο συγκεκριμένα όμως, αύξηση στον αριθμό των αντικειμένων που επεξεργαζόμαστε προκαλεί περισσότερες επαναλήψεις ενημέρωσης των clusters στην CPU, **με αποτέλεσμα να μεγαλώνει ο σειριακός χρόνος σε σχέση με την περίπτωση των 32 διαστάσεων**. Ταυτόχρονα, οι μεταφορές δεδομένων

από και προς τη GPU αυξάνονται, διότι ενώ οι συντεταγμένες είναι λιγότερες, ο συνολικός αριθμός των στοιχείων (δηλαδή των σημείων στο dataset) είναι σημαντικά μεγαλύτερος, με αποτέλεσμα να στέλνεται και να επιστρέφεται μεγαλύτερος όγκος πληροφορίας σε κάθε βήμα. Έτσι, ακόμα και οι Transpose και Shared υλοποιήσεις παρουσιάζουν τελικά υψηλότερους συνολικούς χρόνους, καθώς οι CPU-GPU μεταφορές και η σειριακή ενημέρωση των clusters κερδίζουν αναλογικά μεγαλύτερο βάρος στο συνολικό προφίλ εκτέλεσης. Βέβαια, να σημειωθεί πως οι υψηλότεροι χρόνοι δεν σημαίνουν χειρότερη επίδοση σε προβλήματα με μικρές διαστάσεις (το speedup έχει σχεδόν διπλασιαστεί άρα αντιθέτως επωφελούμαστε παραπάνω)

Η συγκεκριμένη υλοποίηση με χρήση shared memory μπορεί να προσφέρει σημαντική επιτάχυνση σε πολλές περιπτώσεις. Ωστόσο ο κώδικας στην φάση που βρίσκεται τώρα δεν κρίνεται κατάλληλος για χρήση σε arbitrary configurations. Αν το configuration έχει “μη βολικές” τιμές, για παράδειγμα έναν τεράστιο αριθμό clusters ή coordinates συνοδευόμενο από μεγάλο dataset, μπορεί να αυξήσει την δέσμευση μοιραζόμενης μνήμης σε επίπεδο τέτοιο που «μπλοκάρει» το να εκτελούνται ταυτόχρονα αρκετά blocks στον ίδιο SM, που σημαίνει ότι μπορεί να προκαλέσει υπερβολικά χαμηλό occupancy, δηλαδή να μειώσει τον αριθμό των threads που μπορούν να «φορτώνονται» ταυτόχρονα στη GPU. Το αποτέλεσμα θα ήταν πολύ μικρότερη αξιοποίηση της παράλληλης επεξεργασίας και αρα να μην κρύβονται αποτελεσματικά οι καθυστερήσεις μνήμης και να πέφτει σημαντικά η απόδοση της GPU.

Από την άλλη σε διατάξεις όπου ο αριθμός των διαστάσεων είναι μικρός ή το πρόβλημα παρουσιάζει ήδη ελαφρύ υπολογιστικό φορτίο, το όφελος της shared memory μπορεί να μην είναι το ίδιο εντυπωσιακό, αφού το επιπλέον κόστος διαχείρισής της και το overhead που συνοδεύουν οι απαραίτητοι συγχρονισμοί μπορεί να αντισταθμίζουν το κέρδος.

4^η υλοποίηση: Full-Offload (All-GPU) version

Σε αυτή την υλοποίηση, μεταφέρουμε ολόκληρο τον υπολογιστικό φόρτο στη GPU. Εκτός από τον υπολογισμό των αποστάσεων μεταξύ κάθε αντικειμένου και των κέντρων των clusters, και την ανακατανομή των σημείων στα clusters, η GPU αναλαμβάνει πλέον και τον υπολογισμό των νέων κέντρων σύμφωνα με τη νέα κατανομή. Αυτή η προσέγγιση μας απαλλάσσει από την ανάγκη μεταφοράς του πίνακα Membership από τη μνήμη της GPU στη μνήμη της CPU, προσφέροντας σημαντική βελτίωση στην επίδοση.

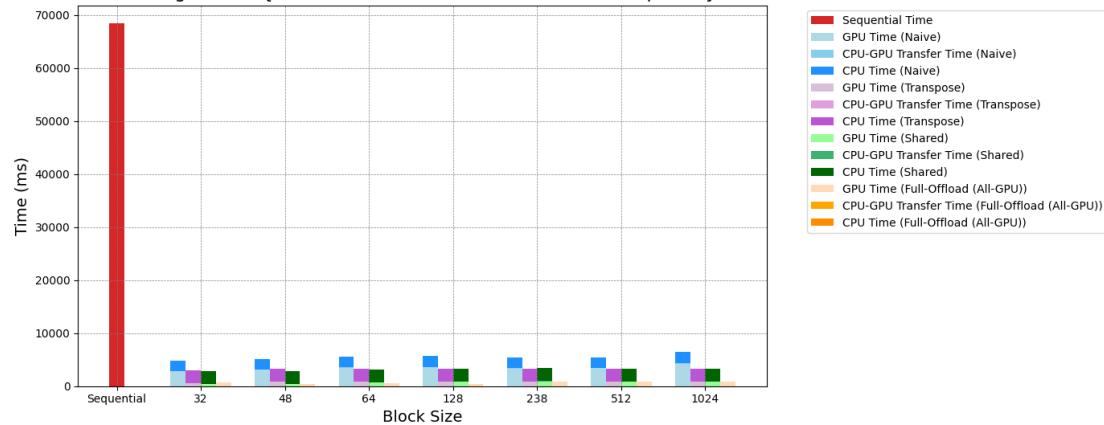
Για την επίτευξη αυτού, χρησιμοποιούμε δύο διαφορετικά kernels: το **find_nearest_cluster** και το **update_centroids**. Το πρώτο kernel παραμένει το ίδιο με τις προηγούμενες υλοποιήσεις, με την προσθήκη μιας νέας λειτουργίας: τον υπολογισμό των αθροισμάτων των συντεταγμένων των αντικειμένων που ανήκουν σε κάθε cluster. Καθώς το kernel αυτό ήδη υπολογίζει τη νέα κατανομή των αντικειμένων στα clusters, είναι φυσικό να αναλάβει και την άθροιση των συντεταγμένων τους.

Ωστόσο, η πρόσβαση στους πίνακες devicenewClusters και devicenewClusterSize μπορεί να δημιουργήσει **race conditions**, καθώς η κατανομή των αντικειμένων στα clusters είναι ακανόνιστη. Για να αντιμετωπιστεί το πρόβλημα, χρησιμοποιείται **shared memory** για τον συγχρονισμό των threads εντός ενός block, αποφεύγοντας την ανάγκη για global συγχρονισμό που θα υποβάθμιζε σημαντικά την απόδοση. Κάθε block χρησιμοποιεί προσωρινούς πίνακες (newCluster_temp και newClustersize_temp) στη shared memory για την αποθήκευση των μερικών αθροισμάτων των συντεταγμένων. Ο συγχρονισμός επιτυγχάνεται μέσω της εντολής **atomicAdd**, που διασφαλίζει την ατομικότητα των πράξεων στα δεδομένα της shared memory. Αφού ολοκληρωθούν τα τοπικά αθροίσματα, πολλά threads αναλαμβάνουν να ενημερώσουν ατομικά τα global δεδομένα, συνδυάζοντας τα μερικά αποτελέσματα από όλα τα blocks ώστε να προκύψουν οι τελικοί πίνακες.

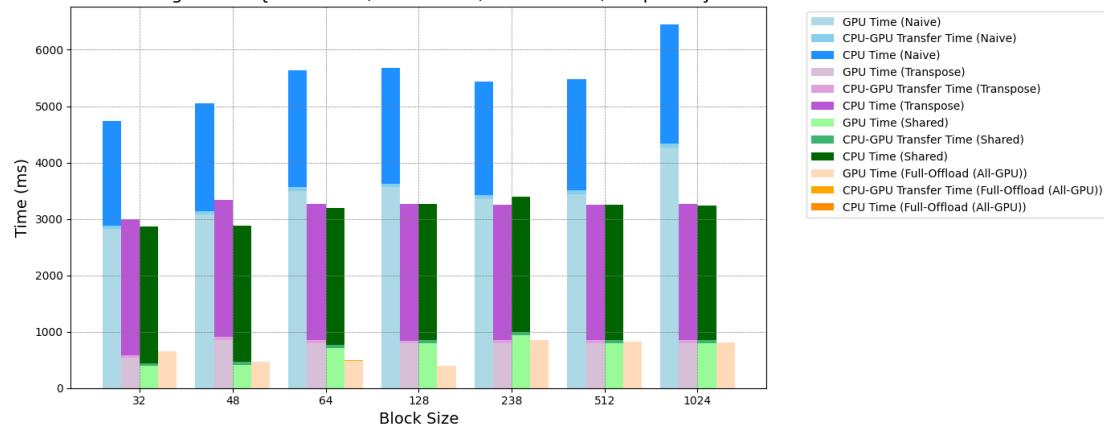
Το δεύτερο kernel, **update_centroids**, αναλαμβάνει την απλή λειτουργία του υπολογισμού του μέσου όρου των αθροισμάτων που προέκυψαν από το πρώτο kernel. Ο αριθμός των δεδομένων που επεξεργάζεται είναι της τάξης **#coordinates * #clusters**, ο οποίος είναι πολύ μικρότερος σε σχέση με τον αριθμό των αντικειμένων **#objects** που επεξεργάζεται το πρώτο kernel. Αυτό επιτρέπει την καλύτερη αξιοποίηση των πόρων της GPU. Αν χρησιμοποιούσαμε το ίδιο kernel και για την άθροιση και για την κανονικοποίηση των στοιχείων, θα δεσμεύαμε πόρους της τάξης **#objects**. Όμως, η δεύτερη λειτουργία (κανονικοποίηση) είναι υπολογιστικά πολύ πιο ελαφριά, και θα άφηνε πολλά threads ανενεργά, οδηγώντας σε σπατάλη πόρων.

Θα πάρουμε τώρα μετρήσεις για τα δύο configuration {1024, 32, 64, 10} και {1024, 2, 64, 10}. Στα διαγράμματα που ακολουθούν παραθέτουμε και τα δεδομένα εκτέλεσης από τις προηγούμενες υλοποιήσεις για να είναι ευκολότερη η σύγκριση.

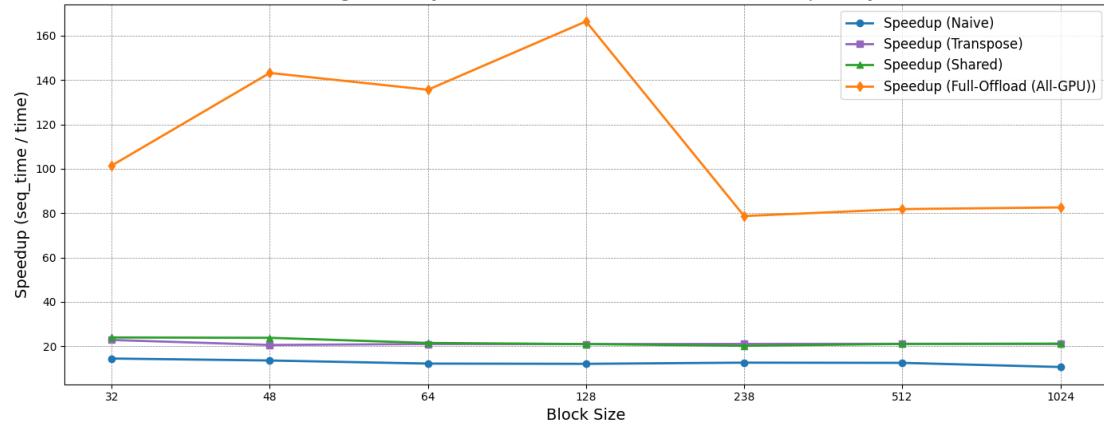
Runtime Comparison: Naive, Transpose, Shared and Full-Offload (All-GPU) Versions
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}

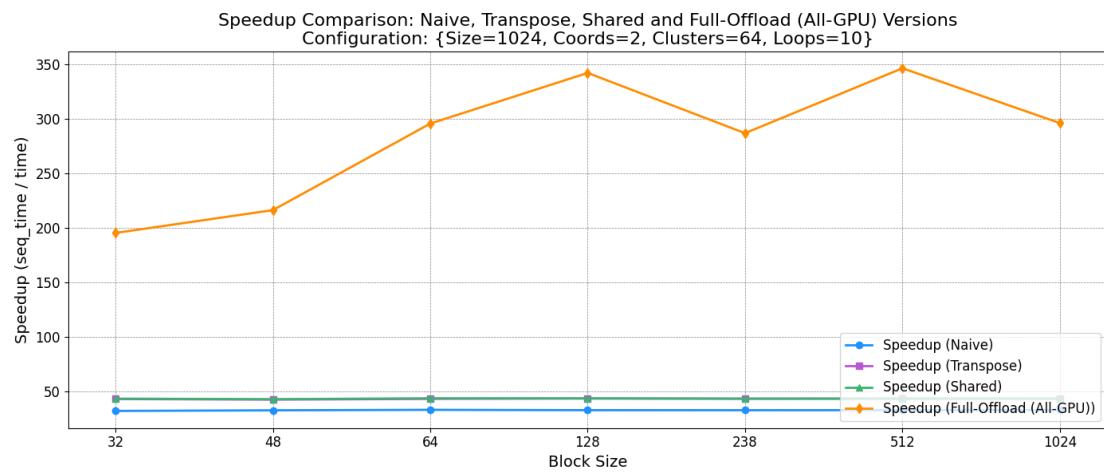
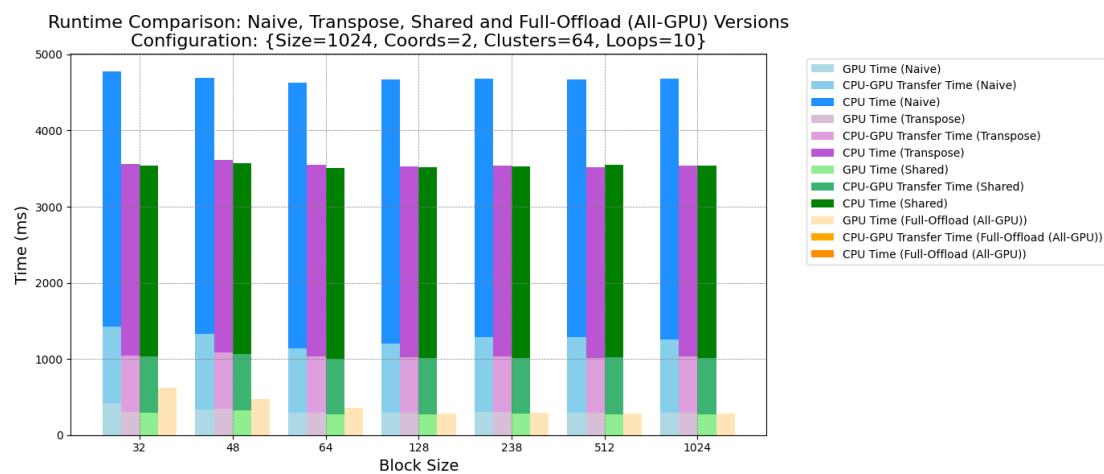
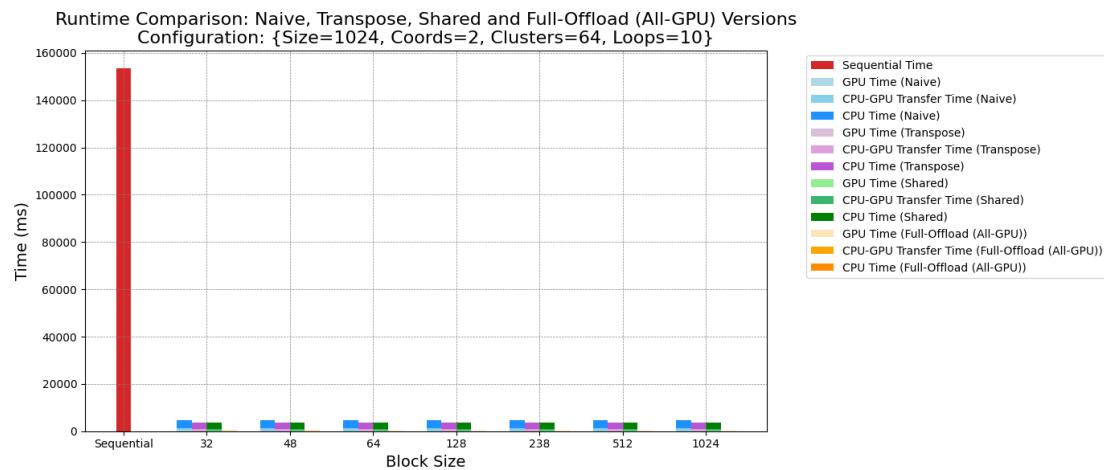


Runtime Comparison: Naive, Transpose, Shared and Full-Offload (All-GPU) Versions
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



Speedup Comparison: Naive, Transpose, Shared, and Full-Offload (All-GPU) Versions
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}





Σχολιασμός και συμπεράσματα

Από τα παραπάνω διαγράμματα χρόνου, είναι ξεκάθαρο πως η νέα υλοποίηση επιτυγχάνει σημαντική βελτίωση στην επίδοση. Όπως αναμενόταν, ο χρόνος εκτέλεσης της CPU είναι πρακτικά μηδενικός, ενώ ο χρόνος μεταφοράς δεδομένων από και προς την GPU είναι αμελητέος, καθώς πλέον μεταφέρεται

μόνο η τιμή του delta πίσω στην CPU. Αυτή η προσέγγιση επιτρέπει τον έλεγχο των επαναλήψεων του αλγορίθμου με βάση τη σύγκλιση, μειώνοντας τη γενική επικοινωνία. Επιπλέον, η βελτιστοποίηση στα δύο kernels επέτρεψε να μετατρέψουμε τους υπολογισμούς της GPU σε περισσότερο compute-bound διεργασίες, καθώς οι αθροίσεις των συντεταγμένων εκτελούνται τοπικά στην GPU, αυξάνοντας την αποδοτικότητα και μειώνοντας τις καθυστερήσεις.

Παρατηρούμε ότι με την All-GPU υλοποίηση επιτυγχάνουμε σημαντική βελτίωση στον χρόνο εκτέλεσης, της τάξης του ~81% για το πρώτο configuration και ~84% για το δεύτερο, συγκριτικά με την προηγούμενη υλοποίηση που βασιζόταν στη shared memory. Επιπλέον, τα διαγράμματα speedup αναδεικνύουν ακόμα πιο έντονα το κέρδος από τη νέα υλοποίηση, υπογραμμίζοντας τη σαφή υπεροχή της All-GPU προσέγγισης σε όρους επίδοσης, ειδικά για πιο απαιτητικά workloads και μεγαλύτερα block sizes.

Η εξάρτηση της απόδοσης του All-GPU implementation από το μέγεθος του block size δείχνει ότι η καλύτερη απόδοση επιτυγχάνεται για μεσαία block sizes, όπως 128. Σε μικρά block sizes (π.χ., 32), το overhead από την εκκίνηση πολλών threads υπερβαίνει τα οφέλη του παραλληλισμού, ενώ σε πολύ μεγάλα block sizes (π.χ., 1024), παρατηρείται συμφόρηση στους υπολογιστικούς πόρους της GPU, κάτι που οδηγεί σε μείωση της αποδοτικότητας. Συνολικά, το All-GPU implementation είναι πιο αποδοτικό από τις προηγούμενες υλοποιήσεις, επειδή ελαχιστοποιεί τη χρήση της CPU και επιτυγχάνει την καλύτερη κατανομή υπολογιστικού φόρτου στις GPUs. Σε αντίθεση με τις υλοποιήσεις Naive, Transpose, και Shared Memory, που παρουσιάζουν σημαντική εξάρτηση από την αλληλεπίδραση μεταξύ CPU και GPU και τις καθυστερήσεις μεταφοράς δεδομένων, το All-GPU καταφέρνει να ελαχιστοποιήσει τον χρόνο μεταφοράς και να εκμεταλλευτεί πλήρως τους πόρους της GPU. Αυτή η διαφοροποίηση του επιτρέπει να επιδεικνύει σταθερά υψηλότερο speedup σε σχέση με τις άλλες υλοποιήσεις, ιδιαίτερα σε κατάλληλα block sizes (64-128).

Η λειτουργία update_centroids (υπολογισμός των νέων κέντρων των clusters) δεν είναι κατάλληλη για GPU καθώς οι προσβάσεις στη μνήμη, δηλαδή η ενημέρωση των πινάκων newClusters και newClusterSize, είναι ακανόνιστες αντί για συνεχόμενες. Αυτό συμβαίνει επειδή η κατανομή των αντικειμένων (objects) στα clusters είναι ακανόνιστη, γεγονός που οδηγεί σε race conditions που δεν μπορούν εύκολα να αποφευχθούν. Με τον διαχωρισμό των δύο kernels καταφέραμε να εξισορροπήσουμε καλύτερα τις προσβάσεις στη μνήμη και να μειώσουμε τα bottlenecks, ιδιαίτερα στη δεύτερη περίπτωση που παρουσίαζε το μεγαλύτερο πρόβλημα. Έτσι, συνδυάσαμε πιο αποδοτικά τις πολλές μνήμες με περισσότερους υπολογισμούς.

Εάν, τέλος, συγκρίνουμε τα δύο configurations, παρατηρούμε ότι το πρώτο έχει μεγαλύτερη εξάρτηση από το block size. Όπως ήδη αναφέρθηκε, προσθέσαμε στη shared memory προσωρινούς πίνακες για newClusters και newClusterSize με διαστάσεις numCoords * numClusters και numClusters αντίστοιχα. Επομένως, στο πρώτο configuration, με περισσότερες συντεταγμένες (numCoords) και λιγότερα αντικείμενα (numObjs), η shared memory φορτώνεται περισσότερο για μεγάλα block sizes. Αυτό εξηγεί γιατί παρατηρείται πτώση του speedup στα block sizes 238, 512 και 1024, όπου η αυξημένη χρήση της shared memory περιορίζει την απόδοση.

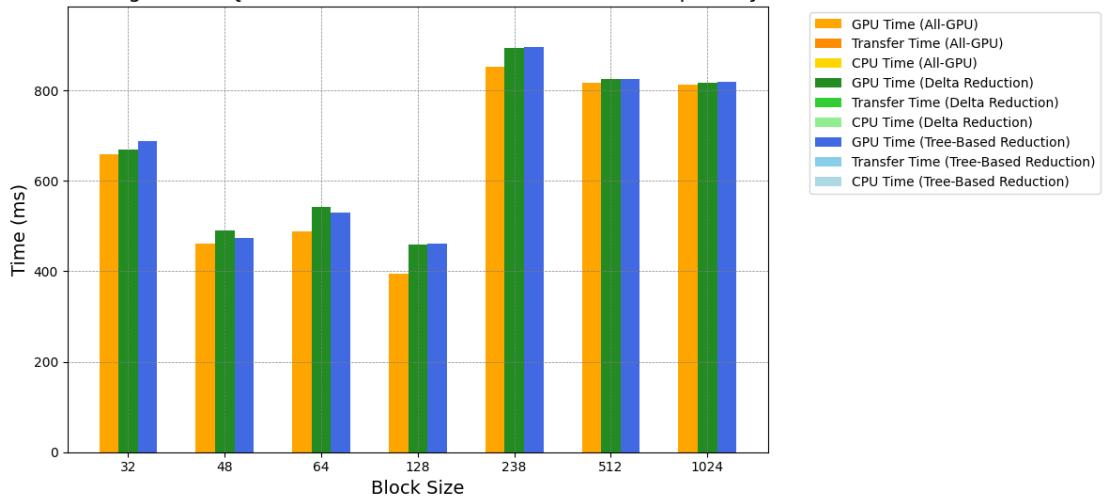
5^η υλοποίηση: Delta Reduction (All-GPU) version

Έως τώρα η αύξηση του delta από κάθε thread γινόταν με atomicAdd globally στη GPU κάτι που σαφώς επιβάρυνε την επίδοση. Ακολουθήσαμε λοιπόν ίδιο σκεπτικό με τους temporary πίνακες newClusters και newClustersize της προηγούμενης υλοποίησης και δεσμεύσαμε λίγο ακόμα χώρο στο shared memory για να υλοποιήσουμε block level reduction για το delta. Συγκεκριμένα, δεσμεύσαμε χώρο όσο ο αριθμός των threads σε κάθε Thread Block έτσι ώστε κάθε thread να ενημερώνει το delta κάνοντας πρόσβαση στον πίνακα με βάση το local id του. Καταφέραμε έτσι να παραλληλοποιήσουμε την ενημέρωση των delta μέσα στο ίδιο Block. Φυσικά, μετά χρειάστηκε σε κάθε block threads να αναλάβουν την πρόσθεση των στοιχείων του temporary πίνακα και τέλος, να προσθέσουν τις τιμές στη global μεταβλητή devdelta η οποία και επιστρέφεται στη cpu.

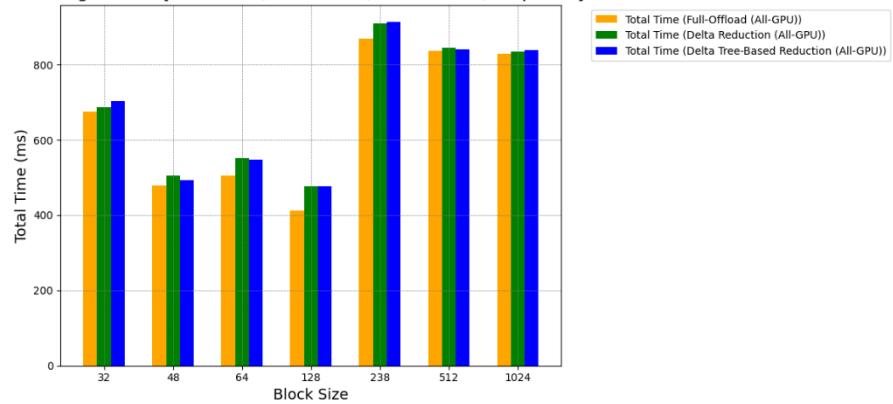
Δοκιμάσαμε και πήραμε μετρήσεις για δύο υλοποιήσεις. Στην πρώτη ένα thread ανά block αναλαμβάνει την πρόσθεση των τοπικών αποτελεσμάτων στη shared memory ενώ στην δεύτερη είναι ένα πιο αποδοτικό tree level reduction.

Ακολουθούν τα διαγράμματα με τους χρόνους εκτέλεσης και τα speedup για τα δύο configuration.

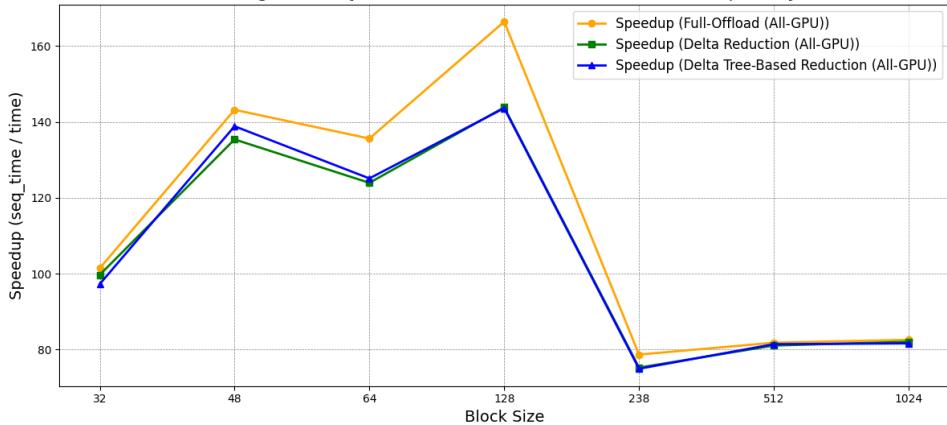
Runtime Comparison: All-GPU, Delta Reduction, and Tree-Based Reduction
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



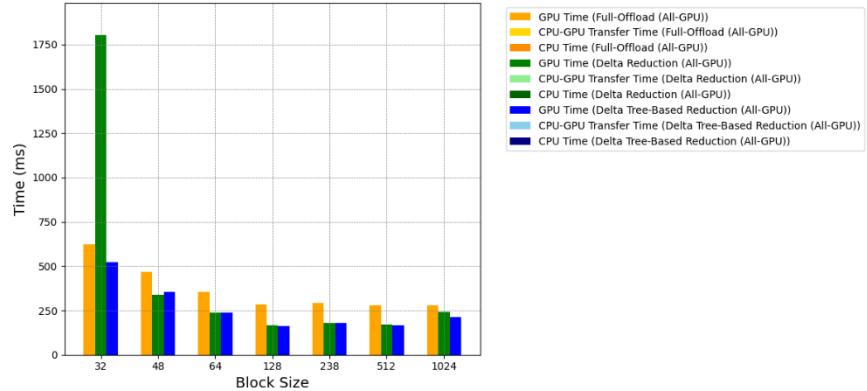
Runtime Comparison: Full-Offload (All-GPU), Delta Reduction (All-GPU) and Delta Tree-Based Reduction (All-GPU) Versions
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



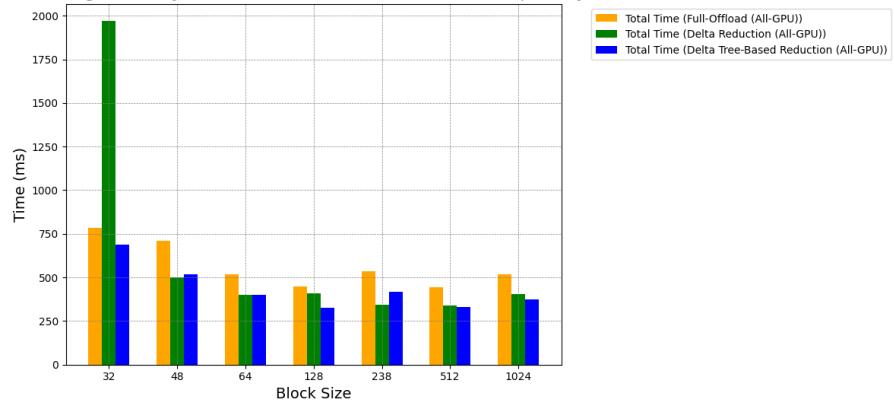
Speedup Comparison: Full-Offload (All-GPU), Delta Reduction (All-GPU) and Delta Tree-Based Reduction (All-GPU) Versions
Configuration: {Size=1024, Coords=32, Clusters=64, Loops=10}



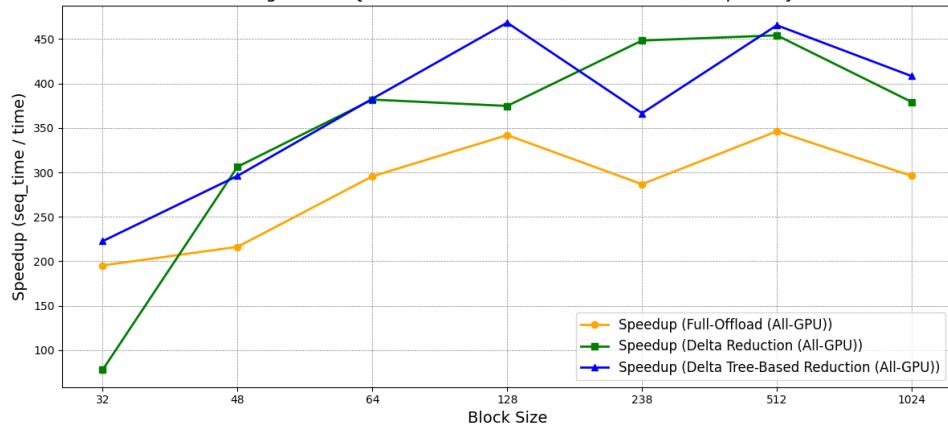
Runtime Comparison: Full-Offload (All-GPU), Delta Reduction (All-GPU) and Delta Tree-Based Reduction (All-GPU) Versions
 Configuration: {Size=1024, Coords=2, Clusters=64, Loops=10}



Runtime Comparison: Full-Offload (All-GPU), Delta Reduction (All-GPU) and Delta Tree-Based Reduction (All-GPU) Versions
 Configuration: {Size=1024, Coords=2, Clusters=64, Loops=10}



Speedup Comparison: Full-Offload (All-GPU), Delta Reduction (All-GPU) and Delta Tree-Based Reduction (All-GPU) Versions
 Configuration: {Size=1024, Coords=2, Clusters=64, Loops=10}



Σχολιασμός και συμπεράσματα

Η υλοποίηση της All-GPU Delta Reduction εμφανίζει ελαφρώς αυξημένους χρόνους σε σύγκριση με την απλή All-GPU έκδοση, ιδιαίτερα για μικρά block sizes. Αυτό οφείλεται στο γεγονός ότι το block-level reduction, το οποίο ανατίθεται σε ένα thread ανά block, είναι αρκετά υπολογιστικά απαιτητικό, προσθέτοντας επιπλέον φόρτο στο συνολικό χρόνο εκτέλεσης.

Όσον αφορά το tree-based reduction, στο πρώτο configuration (με περισσότερες συντεταγμένες), η χρήση του δεν βελτιώνει την επίδοση, καθώς η cache είναι ήδη σημαντικά επιβαρυμένη λόγω της αυξημένης χρήσης μνήμης. Ωστόσο, στο δεύτερο configuration (με λιγότερες συντεταγμένες), το tree-based reduction αποδεικνύεται πιο αποτελεσματικό, μειώνοντας τους χρόνους εκτέλεσης σε όλες σχεδόν τις περιπτώσεις. Αυτό υποδεικνύει ότι η αποτελεσματικότητα του tree-based reduction εξαρτάται από τη συνολική χρήση μνήμης και την κατανομή των δεδομένων.

Επιπλέον, παρατηρείται σαφής εξάρτηση από το block size, καθώς όσο αυξάνεται ο αριθμός των threads μέσα σε ένα block, απαιτείται περισσότερη shared memory για το reduction. Αυτό οδηγεί σε μεγαλύτερη επιβάρυνση για μεγάλα block sizes, επηρεάζοντας αρνητικά την επίδοση. Το αποτέλεσμα είναι η πτώση του speedup που παρατηρούμε για μεγαλύτερα block sizes, ιδιαίτερα στο πρώτο configuration όπου η χρήση της shared memory είναι ήδη αυξημένη.

Συνολικά, η All-GPU Delta Reduction προσφέρει οφέλη σε σενάρια με μικρότερο φόρτο μνήμης, αλλά η εξάρτηση από το block size και η επιβάρυνση της shared memory μπορεί να περιορίσει την απόδοσή της σε πιο απαιτητικά configurations.

4^η Εργαστηριακή Άσκηση

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

4.1 Παραλληλοποίηση αλγορίθμου kmeans σε αρχιτεκτονική κοινής μνήμης

Θα παραλληλοποιήσουμε τον αλγόριθμο KMeans χρησιμοποιώντας MPI, διαφοροποιώντας την υλοποίηση από προηγούμενες προσεγγίσεις, καθώς αυτή τη φορά οι διεργασίες δεν μοιράζονται κοινά δεδομένα στη μνήμη, αλλά εκτελούνται σε διαφορετικά μηχανήματα. Αυτό δημιουργεί την ανάγκη επικοινωνίας μεταξύ των διεργασιών, γεγονός που επηρεάζει την πολυπλοκότητα της παραλληλοποίησης. Η ανάλυσή μας ξεκινά με την κατανομή των δεδομένων στις διεργασίες.

Θα περιγράψουμε τη λογική του κώδικα που υλοποιήσαμε:

Η διεργασία με rank = 0 αρχικοποιεί τυχαία τα αρχικά κέντρα των clusters και τα διαμοιράζει στις υπόλοιπες διεργασίες μέσω της εντολής:

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

Στη συνέχεια, κάθε διεργασία αναλαμβάνει να υπολογίσει τις αποστάσεις μιας υποομάδας σημείων από τα κέντρα των clusters και να αποφασίσει σε ποιο cluster ανήκει κάθε σημείο. Η κατανομή των σημείων στις διεργασίες γίνεται ομοιόμορφα, προσπαθώντας να αποδώσουμε ίσο αριθμό σημείων σε κάθε διεργασία. Ωστόσο, αν ο συνολικός αριθμός των σημείων δεν είναι ακριβώς διαιρετός με τον αριθμό των διεργασιών, το υπόλοιπο ισομοιράζεται, με αποτέλεσμα κάποιες διεργασίες να λαμβάνουν ένα επιπλέον σημείο.

```
/*
 * TODO: Calculate number of objects that each rank will examine (*rank_numObjs)
 */
int base = numObjs / size;
int rem = numObjs % size;
if (rank < rem)
    *rank_numObjs = base + 1;
else
    *rank_numObjs = base;
```

Λόγω αυτής της ανισοκατανομής, χρησιμοποιούμε τις **MPI_Scatterv** και **MPI_Gatherv** για την κατανομή και τη συλλογή των δεδομένων. Προτού εκτελεστούν αυτές οι συναρτήσεις, υπολογίζουμε τις sendcounts και displs, ώστε κάθε διεργασία να γνωρίζει το εύρος των δεδομένων που θα επεξεργαστεί.

```

/*
 * TODO: Calculate sendcounts and displs, which will be used to scatter data to each rank.
 * Hint: sendcounts: number of elements sent to each rank
 *       displs: displacement of each rank's data
 */
int r, offset = 0;
for (r = 0; r < size; r++) {
    int count = (r < rem) ? base + 1 : base;
    sendcounts[r] = count * numCoords;
    displs[r] = offset;
    offset += sendcounts[r];
}

/*
 * TODO: Broadcast the sendcounts and displs arrays to other ranks
 */
MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);

```

Αφού κάθε διεργασία υπολογίσει τις νέες ομάδες των σημείων, με την εντολή MPI_Allreduce κάθε διεργασία λαμβάνει το άθροισμα των συντεταγμένων για κάθε cluster, καθώς και τον αριθμό των σημείων σε κάθε ομάδα, ώστε να υπολογίσει τα νέα κέντρα. Ο ίδιος μηχανισμός εφαρμόζεται και για τη μεταβλητή delta, η οποία συγκεντρώνεται μέσω MPI_Allreduce, ώστε όλες οι διεργασίες να γνωρίζουν αν έχει επιτευχθεί σύγκλιση.

```

/*
 * TODO: Perform reduction of cluster data (rank_newClusters, rank_newClusterSize) from local arrays to shared.
 */
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);

/*
 * TODO: Perform reduction from rank_delta variable to delta variable, that will be used for convergence check.
 */
MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

```

Μόλις ολοκληρωθεί η εκτέλεση του αλγορίθμου, οι τελικές ομάδες συγκεντρώνονται στη διεργασία με rank = 0 μέσω της MPI_Gatherv.

```

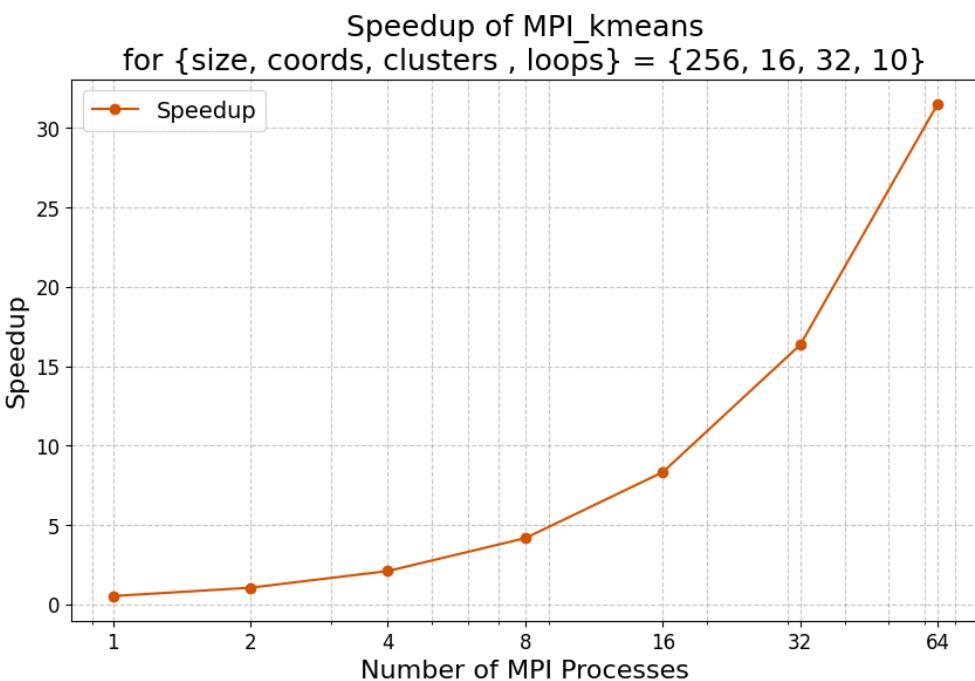
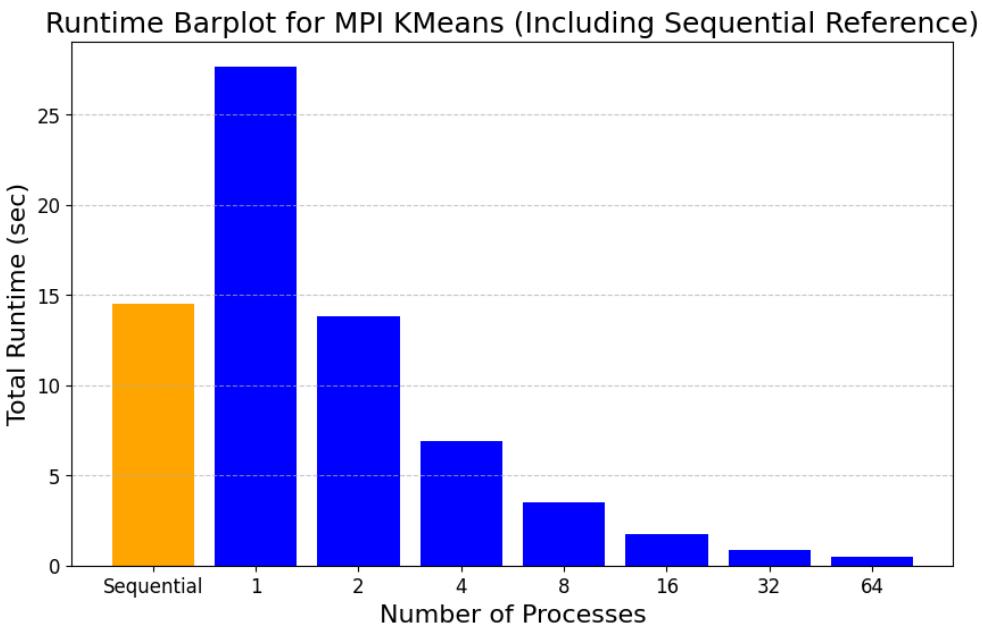
/*
 * TODO: Gather membership information from every rank. (hint: each rank may send different number of objects)
 */
MPI_Gatherv(membership, rank_numObjs, MPI_INT, tot_membership, recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);

```

Θα μετρήσουμε τον χρόνο εκτέλεσης του KMeans για το configuration:

{size, coords, clusters, loops} = {256, 16, 32, 10}, δοκιμάζοντας διαφορετικούς αριθμούς MPI διεργασιών {1, 2, 4, 8, 16, 32, 64}.

Στη συνέχεια, παρουσιάζουμε το barplot με τους χρόνους εκτέλεσης και το διάγραμμα speedup για την αξιολόγηση της απόδοσης της παράλληλης εκδοχής.



Σχολιασμός και συμπεράσματα

Από το barplot του χρόνου εκτέλεσης, παρατηρούμε ότι η αύξηση των διεργασιών μειώνει δραστικά τον χρόνο εκτέλεσης, επιβεβαιώνοντας την αποτελεσματικότητα της παράλληλης εκδοχής. Ωστόσο, η μείωση αυτή δεν είναι απολύτως γραμμική, γεγονός που υποδηλώνει την ύπαρξη επικοινωνιακών και συγχρονιστικών overheads. Ο MPI αλγόριθμος, όταν εκτελείται με μία μόνο διεργασία, παρουσιάζει χρόνο εκτέλεσης σχεδόν διπλάσιο από την καθαρά σειριακή υλοποίηση. Αυτό το αυξημένο overhead, ενώ μοιάζει απροσδόκητο αρχικά, δεδομένου ότι η υλοποίηση έχει σχεδιαστεί για περιβάλλοντα παράλληλης

επεξεργασίας και περιλαμβάνει όλες τις απαραίτητες κλήσεις για την αρχικοποίηση, τη διαχείριση και τον συγχρονισμό που απαιτούνται σε ένα σύστημα MPI, εν τέλει είναι λογικό. Ειδικότερα, η διαδικασία εκκίνησης και ο καθαρισμός του περιβάλλοντος, καθώς και οι συλλογικές λειτουργίες που εφαρμόζονται για τη διάδοση και τη συγχώνευση δεδομένων, ενεργοποιούνται ανεξάρτητα από το αν υπάρχει πραγματική ανταλλαγή μηνυμάτων. Επομένως, το γεγονός ότι ο χρόνος εκτέλεσης με μία μόνο διεργασία είναι σημαντικά υψηλότερος από αυτόν της σειριακής υλοποίησης αντικατοπτρίζει το κόστος των εσωτερικών διαδικασιών που υλοποιούνται για να υποστηριχθεί ο παράλληλος χαρακτήρας του αλγορίθμου. Άρα το αποτέλεσμα αποδίδεται στο ότι η υποδομή του MPI εισάγει ένα σταθερό overhead, το οποίο εμφανίζεται ακόμα και όταν δεν αξιοποιείται πλήρως ο παράλληλος μηχανισμός, καθώς ο σχεδιασμός του συστήματος προορίζεται για περιβάλλοντα πολλαπλών διεργασιών.

Το διάγραμμα speedup δείχνει ότι η επιτάχυνση αυξάνεται σημαντικά με τον αριθμό των διεργασιών, προσεγγίζοντας ιδανικές τιμές μέχρι ένα σημείο. Ωστόσο, παρατηρούμε ότι το speedup αρχίζει να αποκλίνει από την ιδανική κλιμάκωση σε πολύ μεγάλες τιμές διεργασιών, πιθανώς λόγω του αυξημένου κόστους επικοινωνίας. Συγκεκριμένα, για τους λόγους που αναφέραμε εκτενώς πριν, το speedup για 1 thread είναι κάτω του 1 (0.5372) που φανερώνει την επίδραση του overhead. Για 2 νήματα το speedup είναι πολύ κοντά στο 1, πράγμα που οφείλεται στο tradeoff κόστους αρχικοποίησης και επικοινωνίας με το όφελος της παραλληλοποίησης. Καθώς αυξάνεται ο αριθμός διεργασιών έως 8 και 16, βλέπουμε σημαντική βελτίωση στο speedup, που όμως παραμένει υπογραμμικό σε σχέση με το ιδανικό (π.χ. ιδανικά για 16 διεργασίες θα περιμέναμε speedup ίσο με 16, ενώ παρατηρούμε περίπου 8, πράγμα που υπογραμμίζει το ότι όσο πάει γίνεται και πιο σημαντικό το επικοινωνιακό και συγχρονιστικό overhead). Για 32 και 64 διεργασίες, ενώ το speedup αυξάνεται (17 και 32 αντίστοιχα), το ποσοστό απόδοσης σε σχέση με το ιδανικό μειώνεται αισθητά. Συγκεκριμένα, στα 64 πυρήνες καταλήγουμε περίπου το μισό speedup του ιδανικού, δείχνοντας ότι το αυξημένο overhead επικοινωνίας και οι επιβαρύνσεις λόγω συγχρονισμού αρχίζουν να κυριαρχούν και να περιορίζουν την κλιμάκωση.

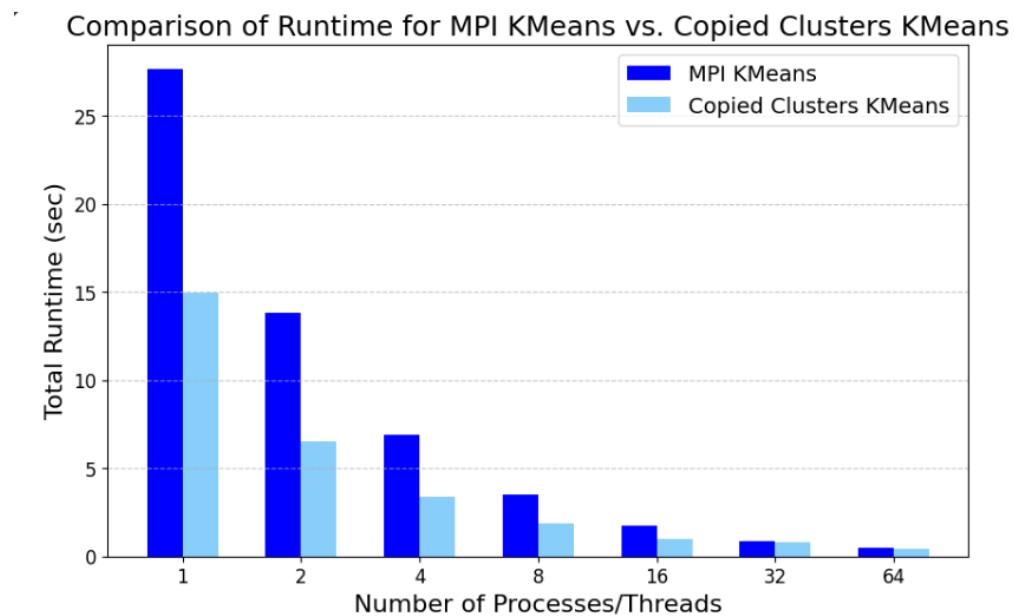
Συνεπώς, η παράλληλη εκδοχή είναι σαφώς ταχύτερη από τη σειριακή, ενώ η κλιμάκωση είναι καλή μέχρι ένα συγκεκριμένο όριο (περίπου 16-32 διεργασίες), πέρα από το οποίο το overhead αρχίζει να επηρεάζει την απόδοση.

Σύγκριση παραλληλοποίησης αλγορίθμου σε αρχιτεκτονική κοινής μνήμης και σε αρχιτεκτονική κατανεμημένης μνήμης.

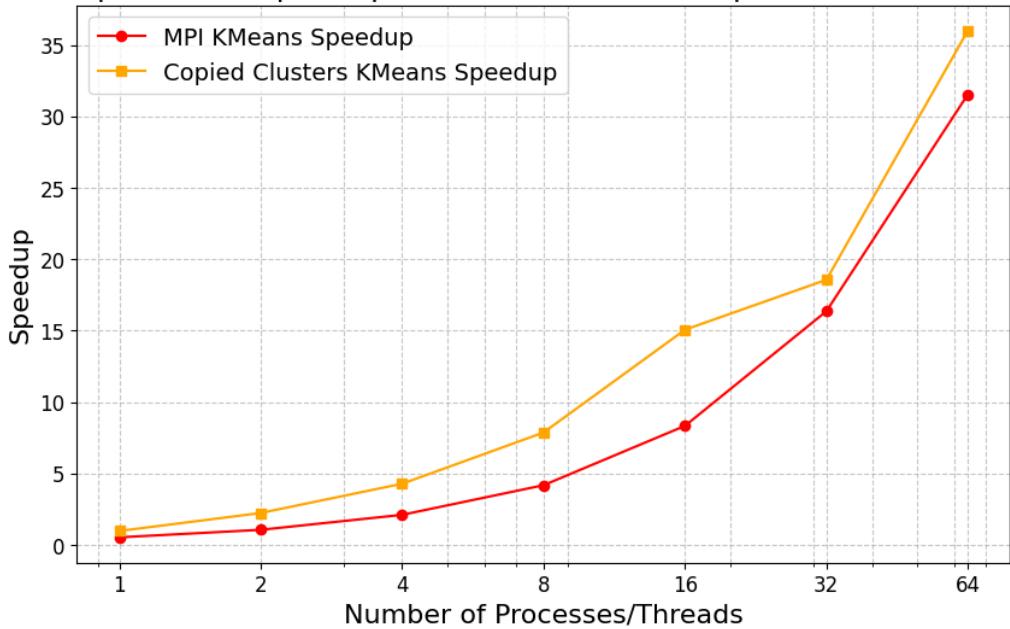
Σε αυτό το σημείο, είναι ενδιαφέρον να συγκρίνουμε την απόδοση της παραλληλοποίησης του αλγορίθμου KMeans σε δύο διαφορετικές αρχιτεκτονικές: κοινής μνήμης και κατανεμημένης μνήμης. Για τη σύγκριση αυτή, θα χρησιμοποιήσουμε:

- Την υλοποίηση **MPI του KMeans**, που αναπτύξαμε προηγουμένως για αρχιτεκτονική κατανεμημένης μνήμης.
- Την υλοποίηση **Copied_Clusters KMeans**, που είχαμε αναπτύξει νωρίτερα στην αναφορά και είχε αποδειχθεί πιο αποδοτική σε σχέση με την υλοποίηση Shared_Clusters για αρχιτεκτονική κοινής μνήμης.

Για να διευκολύνουμε την ανάλυση και εξαγωγή συμπερασμάτων, παρουσιάζουμε κοινά διαγράμματα χρόνου εκτέλεσης (barplot) και επιτάχυνσης (speedup) για τις δύο αυτές υλοποιήσεις. Μέσα από τη σύγκριση αυτή, θα αξιολογήσουμε τις διαφορές στην επίδοση των δύο αρχιτεκτονικών και τον αντίκτυπο της επικοινωνίας στη συνολική απόδοση της κατανεμημένης προσέγγισης.



Comparison of Speedup for MPI KMeans vs. Copied Clusters KMeans



Σχολιασμός και συμπεράσματα:

Η σύγκριση των δύο υλοποιήσεων του KMeans δείχνει ότι η Copied Clusters KMeans παρουσιάζει καλύτερη απόδοση σε σχέση με την MPI KMeans, τόσο στον χρόνο εκτέλεσης όσο και στο speedup. Από το barplot, παρατηρούμε ότι η Copied Clusters KMeans επιτυγχάνει χαμηλότερους χρόνους εκτέλεσης για όλους τους αριθμούς διεργασιών/νημάτων, γεγονός που δείχνει αποδοτικότερη χρήση των διαθέσιμων πόρων. Στο διάγραμμα speedup, η Copied Clusters KMeans έχει υψηλότερες τιμές επιτάχυνσης, υποδεικνύοντας καλύτερη κλιμάκωση, πιθανώς λόγω του μικρότερου overhead επικοινωνίας, καθώς βασίζεται σε κοινή μνήμη αντί για ανταλλαγή μηνυμάτων.

Αυτή η συμπεριφορά οφείλεται στο γεγονός ότι σε αρχιτεκτονικές κοινής μνήμης, όλες οι διεργασίες έχουν άμεση πρόσβαση στα ίδια δεδομένα, αποφεύγοντας το κόστος μεταφοράς των ενδιάμεσων αποτελεσμάτων μεταξύ των διεργασιών. Αντίθετα, σε αρχιτεκτονικές κατανεμημένης μνήμης, όπως η MPI υλοποίηση, η ανταλλαγή δεδομένων μέσω μηνυμάτων εισάγει σημαντικό overhead, επιβραδύνοντας την εκτέλεση. Παρόλο που στις αρχιτεκτονικές κοινής μνήμης προκύπτουν race conditions, λόγω πολλαπλών προσβάσεων σε κοινές θέσεις μνήμης, ο συγχρονισμός των διεργασιών μέσω μηχανισμών όπως locks και barriers έχει μικρότερο κόστος από την επικοινωνία μέσω μηνυμάτων στις κατανεμημένες υλοποιήσεις. Η υλοποίηση της Copied Clusters KMeans εκτελείται σε μηχανή με NUMA αρχιτεκτονική, η οποία διαθέτει 4 NUMA nodes, κάθε ένας με 16 νήματα (8 φυσικά νήματα συν 8 επιπλέον μέσω hyperthreading). Αυτή η

αρχιτεκτονική επιτρέπει την αποδοτική τοπική πρόσβαση στη μνήμη, μειώνοντας το latency των προσβάσεων σε σύγκριση με κατανεμημένες λύσεις όπου η επικοινωνία μεταξύ των μηχανημάτων είναι πιο δαπανηρή, αξιοποιώντας το cache locality. Παρά το γεγονός ότι η NUMA αρχιτεκτονική απαιτεί προσεκτική διαχείριση της τοπικότητας της μνήμης για τη βελτιστοποίηση της απόδοσης, το συνολικό κόστος επικοινωνίας και συγχρονισμού παραμένει σημαντικά μικρότερο από αυτό που παρατηρείται σε υλοποιήσεις που βασίζονται στην ανταλλαγή μηνυμάτων μέσω MPI. Τα πειραματικά αποτελέσματα δείχνουν ότι, στον KMeans, το κόστος επικοινωνίας στις κατανεμημένες υλοποιήσεις είναι μεγαλύτερο από το κόστος συγχρονισμού στις κοινές μνήμες. Συνεπώς, η Copied Clusters KMeans αποδεικνύεται πιο αποδοτική για την παραλληλοποίηση του αλγορίθμου.

4.2 Διάδοση Θερμότητας σε δύο διαστάσεις

Στη συνέχεια αυτής της άσκησης θα εξετάσουμε την επίδραση της παραλληλοποίησης σε αρχιτεκτονική κατανεμημένης μνήμης των διαφόρων υπολογιστικών πυρήνων για την επίλυση του προβλήματος διάδοσης της θερμότητας. Συγκεκριμένα, θα μελετήσουμε τρις διαδεδομένους υπολογιστικούς πυρήνες: Jacobi, Gauss-Seidel με Successive Over-Relaxation (SOR) και Red-Black με SOR.

Θα μελετήσουμε παρακάτω τις δυνατότητες παραλληλοποίησης κάθε υπολογιστικού πυρήνα και στη συνέχεια για κάθε πυρήνα θα πάρουμε μετρήσεις που θα μας βοηθήσουν να πάρουμε αποτελέσματα για την σύγκλιση των τριών μεθόδων και ύστερα μετρήσεις για τον χρόνο εκτέλεσης της παραλληλοποίημενης έκδοσης σε κάθε πυρήνα, αναδεικνύοντας την ταχύτερη τεχνική για δεδομένο αριθμό επαναλήψεων.

Αρχικά, θα παρουσιάσουμε με συντομία τον τρόπο παραλληλοποίησης του κάθε υπολογιστικού πυρήνα και ύστερα θα παρουσιάσουμε τα διαγράμματα σύγκρισης για τους χρόνους εκτέλεσης.

4.2.1 Υπολογιστικός πυρήνας Jacobi

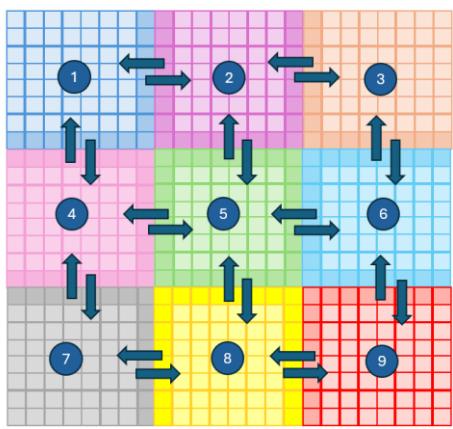
Παραθέτουμε εδώ τον κώδικα του πυρήνα Jacobi:

Μέθοδος Jacobi

```

for (t = 0; t < T && !converged; t++)
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            U[t+1][i][j]=(1/4)*(U[t][i-1][j]+U[t][i][j-1]
                                +U[t][i+1][j]+U[t][i][j+1]);
    converged=check_convergence(U[t+1],U[t]);

```



Παρατηρούμε πως η ανανέωση των κελιών σε μια χρονική στιγμή εξαρτάται μονάχα από τα τέσσερα γειτονικά κελιά της την προηγούμενη χρονική στιγμή. Συνεπώς μπορούμε να παραλληλοποιήσουμε την ανανέωση των κελιών μέσα σε μια χρονική στιγμή φροντίζοντας οι διεργασίες να στέλνουν τα “οριακά” τους κελιά στις γειτονικές διεργασίες και συνεπώς να λαμβάνουν από αυτές τα αντίστοιχα κελιά.

Είναι λοιπόν συνετό να χωρίσουμε τον αρχικό δισδιάστατο πίνακα σε δισδιάστατους υποπίνακες τους οποίους θα μοιράσουμε σε ανάμεσα στις διεργασίες. Δηλαδή ο διαχωρισμός των δεδομένων θα γίνει σε 2D blocks με sequential τρόπο. Με τον τρόπο αυτό διασφαλίζουμε την ελάχιστη απαραίτητη επικοινωνία μεταξύ των διεργασιών, αφού χρειάζεται η κάθε διεργασία να στείλει μονάχα τα συνοριακά της κελιά.

Οι απαραίτητες ανταλλαγές των κελιών φαίνονται στο παραπάνω σχήμα, τα πιο σκουρόχρωμα κελιά είναι αυτά που πρέπει να ανταλλαγούν μεταξύ των γειτονικών διεργασιών.

Θα αναλύσουμε τον κώδικα που αναπτύξαμε για την παραληλοποίηση του υπολογιστικού πυρήνα Jacobi.

Η κάθε διεργασία θα έχει δύο πίνακες `u_current` και `u_previous` τους οποίους χρησιμοποιεί για να κρατάει δεδομένα από την προηγούμενη επανάληψη και να αποθηκεύει τα δεδομένα της τρέχουσας επανάληψης του πυρήνα. Για κάθε `block` δεδομένων που αναλαμβάνει κάθε διεργασία δεσμεύουμε λίγο περισσότερο χώρο, τα `ghost cells`, στα οποία η κάθε διεργασία θα αποθηκεύει τα χρήσιμα για αυτή δεδομένα που λαμβάνει από τις γειτονικές της διεργασίες.

Τα αρχικά δεδομένα θα μοιραστούν ομοιόμορφα ανάμεσα στις διεργασίες και θα γίνει το απαραίτητο padding στις τελευταίες γραμμές και στήλες του συνολικού πίνακα για να καταφέρουν να μοιραστούν τα δεδομένα ισόποσα.

Η διεργασία με rank=0 θα μοιράσει κατάλληλα τα κελιά μέσω της εντολής MPI_Scatterv:

```
MPI_Scatterv(U_start, scattercounts, scatteroffset, global_block, &(u_current[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
```

Σημειώνεται πως χρησιμοποιούμε δείκτη για τον receive buffer ίσο με u_current[1][1] γιατί θέλουμε τα δεδομένα που θα διαδοθούν να αποθηκευτούν στα εσωτερικά κελιά και όχι στα ghost cells, ára ξεκινάμε από το index [1][1].

Επειδή, όπως επεξηγήσαμε παραπάνω η κάθε διεργασία θα στέλνει στις γειτονικές της μια σειρές και στήλες των υπολογισμένων κελιών της ορίσαμε κατάλληλα datatypes row και column για να διευκολύνουμε την επικοινωνία μεταξύ των διεργασιών.

```
// Row (for sending entire rows)
MPI_Datatype row;
MPI_Type_contiguous(local[1], MPI_DOUBLE, &row);
MPI_Type_commit(&row);

// Column (for sending entire columns)
MPI_Datatype col;
MPI_Type_vector(local[0], 1, local[1] + 2, MPI_DOUBLE, &dummy);
MPI_Type_create_resized(dummy, 0, sizeof(double), &col);
MPI_Type_commit(&col);
```

Η κάθε διεργασία θα χρειαστεί να επικοινωνήσει με το πολύ τέσσερις γειτονικές της, βόρεια, νότια, ανατολική και δυτική. Πριν ξεκινήσουμε την επικοινωνία για κάθε διεργασία ελέγχουμε ποιο από τους τέσσερις εν δυνάμει γείτονές της υπάρχουν (οι “ακριανές” διεργασίες δεν έχουν όλους τους γείτονες). Οι γείτονες διαπιστώνονται με την βοήθεια της εντολής MPI_Cart_shift, η οποία επιστρέφει MPI_PROC_NULL αν κάποιος γείτονας δεν υπάρχει.

```
int north, south, east, west;

MPI_Cart_shift(CART_COMM,1,1,&west,&east);
MPI_Cart_shift(CART_COMM,0,1,&north,&south);
```

Ακόμη χρειάζεται να προσδιορίσουμε τα όρια i_min i_max j_min j_max μέσα στα οποία η κάθε διεργασία θα εφαρμόσει την ανανέωση με τον πυρήνα jacobi. Συγκεκριμένα, μας ενδιαφέρει να μην εφαρμοστεί ανανέωση τόσο στα ghost cells που αναφέραμε πριν, αφού είναι κελιά που απλώς κουβαλούν πληροφορία από την προηγούμενη επανάληψη, αλλά και στα κελιά των αρχικών συνοριακών συνθηκών του προβλήματος της διάδοσης της θερμότητας, δηλαδή οι πρώτες και οι τελευταίες στήλες του αρχικού πίνακα.

```

// i (row) limits
if (north == MPI_PROC_NULL)
    i_min = 2; // Ghost cell + boundary
else
    i_min = 1;

if (south == MPI_PROC_NULL) {
    int pad = global_padded[0] - global[0]; // Padding calculation
    i_max = local[0] + 1 - pad - 1;
} else {
    i_max = local[0] + 1;
}

// j (column) limits
if (west == MPI_PROC_NULL)
    j_min = 2; // Ghost cell + boundary
else
    j_min = 1;

if (east == MPI_PROC_NULL) {
    int pad = global_padded[1] - global[1]; // Padding calculation
    j_max = local[1] + 1 - pad - 1;
} else {
    j_max = local[1] + 1;
}

```

Ύστερα η κάθε διεργασία θα στείλει στους τέσσερις γείτονές της, εφόσον αυτοί υπάρχουν, τις κατάλληλες γραμμές και στήλες. Συγκεκριμένα, η επικοινωνία έχει ως εξής:

- **Στον βόρειο γείτονα:** στέλνουμε την πρώτη γραμμή δεδομένων ($\&u_{previous}[1][1]$) και λαμβάνουμε από εκείνον δεδομένα που γράφουμε στην πρώτη γραμμή - ghost cells ($\&u_{previous}[0][1]$).
- **Στον νότιο γείτονα:** στέλνουμε την τελευταία γραμμή δεδομένων ($\&u_{previous}[local[0]][1]$) και λαμβάνουμε από εκείνον δεδομένα που γράφουμε στην τελευταία γραμμή - ghost cells ($\&u_{previous}[local[0]+1][1]$).
- **Στον ανατολικό γείτονα:** στέλνουμε την τελευταία στήλη δεδομένων ($\&u_{previous}[1][local[1]]$) και λαμβάνουμε από εκείνον δεδομένα που γράφουμε στην τελευταία στήλη - ghost cells ($\&u_{previous}[1][local[1]+1]$).
- **Στον δυτικό γείτονα:** στέλνουμε την πρώτη στήλη δεδομένων ($\&u_{previous}[1][1]$) και λαμβάνουμε από εκείνον δεδομένα που γράφουμε στην πρώτη στήλη - ghost cells ($\&u_{previous}[1][0]$).

Η επικοινωνία αυτή φαίνεται αναλυτικά στον παρακάτω κώδικα:

```

// Swap pointers
swap=u_previous;
u_previous=u_current;
u_current=swap;

// Start non-blocking communication
MPI_Request reqs_sent[4];
MPI_Request reqs_received[4];
MPI_Status stats[4];
int count_neighbours = 0;

// Communicate with north
if (north != MPI_PROC_NULL){
    MPI_Isend(&u_previous[1][1], 1, row, north, 0, MPI_COMM_WORLD, &reqs_sent[count_neighbours]);
    MPI_Irecv(&u_previous[0][1], 1, row, north, 0, MPI_COMM_WORLD, &reqs_received[count_neighbours]);
    count_neighbours++;
}

// Communicate with south
if (south != MPI_PROC_NULL){
    MPI_Isend(&u_previous[local[0][1]], 1, row, south, 0, MPI_COMM_WORLD, &reqs_sent[count_neighbours]);
    MPI_Irecv(&u_previous[local[0]+1][1], 1, row, south, 0, MPI_COMM_WORLD, &reqs_received[count_neighbours]);
    count_neighbours++;
}

// Communicate with east
if (east != MPI_PROC_NULL){
    MPI_Isend(&u_previous[1][local[1]], 1, col, east, 0, MPI_COMM_WORLD, &reqs_sent[count_neighbours]);
    MPI_Irecv(&u_previous[1][local[1]+1], 1, col, east, 0, MPI_COMM_WORLD, &reqs_received[count_neighbours]);
    count_neighbours++;
}

// Communicate with west
if (west != MPI_PROC_NULL){
    MPI_Isend(&u_previous[1][1], 1, col, west, 0, MPI_COMM_WORLD, &reqs_sent[count_neighbours]);
    MPI_Irecv(&u_previous[1][0], 1, col, west, 0, MPI_COMM_WORLD, &reqs_received[count_neighbours]);
    count_neighbours++;
}

MPI_Waitall(count_neighbours, reqs_received, stats);
MPI_Waitall(count_neighbours, reqs_sent, stats);

```

Χρησιμοποιούμε non blocking εντολές επικοινωνίας για να ελαχιστοποιήσουμε τον χρόνο εκτέλεσης. Στο τέλος προσθέτουμε barrier MPI_Waitall για να είμαστε σίγουροι πως όλοι έχουν στείλει και έχουν λάβει τα κατάλληλα δεδομένα πριν εφαρμόσουμε τον πυρήνα ανανέωσης Jacobi.

Μετά την επικοινωνία κρατάμε όλα τα δεδομένα στον πίνακα u_previous και ανανεώνουμε τον πίνακα u_current με βάση τον πυρήνα Jacobi.

```

gettimeofday(&tcs, NULL);
for (i=i_min;i<i_max;i++)
    for (j=j_min;j<j_max;j++)
        u_current[i][j]=(u_previous[i-1][j]+u_previous[i+1][j]+u_previous[i][j-1]+u_previous[i][j+1])/4.0;
gettimeofday(&tcf, NULL);
tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;

```

Αφού περάσουν ο κατάλληλος επαναλήψεων ή αφού έχει συγκλίνει ο αλγόριθμος όλοι οι πίνακες μαζεύονται μέσω της εντολής MPI_Gatherv στην διεργασία με rank = 0 η οποία αναλαμβάνει να τυπώσει και τα τελικά αποτελέσματα.

```

MPI_Gatherv(&u_current[1][1], 1, local_block, U_start, scattercounts, scatteroffset, global_block, 0, MPI_COMM_WORLD);

```

4.2.2 Υπολογιστικός πυρήνας Gauss-Seidel SOR

Μέθοδος Gauss-Seidel SOR

```
for (t = 0; t < T && !converged; t++)
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            U[t+1][i][j]=U[t][i][j]
                +(omega/4)*(U[t+1][i-1][j]+U[t+1][i][j-1]
                +U[t][i+1][j]+U[t][i][j+1]
                -4*U[t][i][j]);
    converged=check_convergence(U[t+1],U[t]);
```

Για να παραλληλοποιήσουμε αυτόν τον υπολογιστικό πυρήνα ακολουθούμε ακριβώς την ίδια στρατηγική που περιγράψαμε παραπάνω για την παραλληλοποίηση του Jacobi, μόνο που θα χρειαστεί να τροποποιήσουμε τα δεδομένα που ανταλλάσσουν οι γειτονικές διεργασίες. Συγκεκριμένα, βασική διαφορά είναι πως για την γρηγορότερη σύγκλιση ο βόρειος και ο δυτικός γείτονας κάθε διεργασίας θα στείλουν τα δεδομένα που υπολόγισαν μετά την ανανέωση. Θα στείλουν δηλαδή δεδομένα από τον πίνακα u_current και όχι μονάχα από τον u_previous.

Αναλυτικότερα, αρχικά πραγματοποιείται το swap των pointers. Στη συνέχεια, λαμβάνουμε την τιμή του u_previous από νότια και ανατολικά και του u_current από βόρεια και δυτικά, ενώ στέλνουμε τον u_previous βόρεια και δυτικά.

```
// Swap pointers
swap=u_previous;
u_previous=u_current;
u_current=swap;

// Declare MPI_Request and MPI_Status arrays
MPI_Request requests[4]; // 4 sends + 4 receives
MPI_Status statuses[4];
int request_count = 0; // How many requests we actually issue

// Start non-blocking communication
if (south != MPI_PROC_NULL) {
    MPI_Irecv(&u_previous[local[0] + 1][1], 1, row, south, 0, MPI_COMM_WORLD, &requests[request_count++]);
}
if (east != MPI_PROC_NULL) {
    MPI_Irecv(&u_previous[1][local[1] + 1], 1, col, east, 0, MPI_COMM_WORLD, &requests[request_count++]);
}

if (north != MPI_PROC_NULL) {
    MPI_Isend(&u_previous[1][1], 1, row, north, 0, MPI_COMM_WORLD, &requests[request_count++]);
}
if (west != MPI_PROC_NULL) {
    MPI_Isend(&u_previous[1][1], 1, col, west, 0, MPI_COMM_WORLD, &requests[request_count++]);
}

MPI_Waitall(request_count, requests, statuses);

MPI_Request requests2[2]; // Request the current north and current west
MPI_Status statuses2[2];
int request_count2 = 0;

if (north != MPI_PROC_NULL) {
    MPI_Irecv(&u_current[0][1], 1, row, north, 0, MPI_COMM_WORLD, &requests2[request_count2++]);
}
if (west != MPI_PROC_NULL) {
    MPI_Irecv(&u_current[1][0], 1, col, west, 0, MPI_COMM_WORLD, &requests2[request_count2++]);
}

MPI_Waitall(request_count2, requests2, statuses2);
```

Μετά την κλήση του MPI_Waitall, που διασφαλίζει την ολοκλήρωση της επικοινωνίας, ακολουθεί η εκτέλεση των υπολογισμών.

```
gettimeofday(&tc, NULL);
for (i = i_min; i < i_max; i++) {
    for (j = j_min; j < j_max; j++) {
        u_current[i][j] = u_previous[i][j] + (omega/4.0)*(u_current[i-1][j] + u_previous[i+1][j] + u_current[i][j-1] + u_previous[i][j+1] - 4 * u_previous[i][j]);
    }
}
gettimeofday(&tcf, NULL);
tcomp += (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec - tcs.tv_usec) * 0.000001;
```

Τέλος, στέλνουμε την τιμή του u_current νότια και ανατολικά.

```
if (south != MPI_PROC_NULL) {
    MPI_Isend(&u_current[local[0]][1], 1, row, south, 0, MPI_COMM_WORLD, &requests[0]);
}
if (east != MPI_PROC_NULL) {
    MPI_Isend(&u_current[1][local[1]], 1, col, east, 0, MPI_COMM_WORLD, &requests[1]);
}
```

4.2.3 Υπολογιστικός πυρήνας Red-Black SOR

Μέθοδος Red-Black SOR

```
for (t = 0; t < T && !converged; t++)
    //Red phase
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            if ((i+j)%2==0)
                U[t+1][i][j]=U[t][i][j]
                    +(omega/4)*(U[t][i-1][j]+U[t][i][j-1]
                    +U[t][i+1][j]+U[t][i][j+1]
                    -4*U[t][i][j]);
    //Black phase
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            if ((i+j)%2==1)
                U[t+1][i][j]=U[t][i][j]
                    +(omega/4)*(U[t+1][i-1][j]+U[t+1][i][j-1]
                    +U[t+1][i+1][j]+U[t+1][i][j+1]
                    -4*U[t][i][j]);
    converged=check_convergence(U[t+1],U[t]);
```

Η μέθοδος αυτή συνδυάζει τις δύο παραπάνω. Πάλι θα ακολουθήσουμε την βασική δομή που εξηγήσαμε για το Jacobi και αυτό που θα χρειαστεί να αλλάξει είναι η επικοινωνία μεταξύ των διεργασιών.

Αφού οι διεργασίες έχουν ανταλλάξει δεδομένα με τους τέσσερις γείτονες όπως περιγράφηκε στη μέθοδο Jacobi θα σώσουμε αυτές τις τιμές στον πίνακα u_previous και θα εφαρμόσουμε τον πυρήνα για τα “κόκκινα” κελιά και αφού ανανεωθούν οι τιμές τους οι διεργασίες θα επικοινωνήσουν πάλι όπως

προηγουμένως για να στείλουν τις ανανεωμένες τιμές πριν εφαρμοστεί ο πυρήνας για τα μαύρα κελιά.

Η επικοινωνία αυτή φαίνεται αναλυτικά στον παρακάτω κώδικα:

```

// Swap pointers
swap=u_previous;
u_previous=u_current;
u_current=swap;

// Start non-blocking communication
MPI_Request reqs_prev_sent[4];
MPI_Request reqs_curr_sent[4];
MPI_Request reqs_prev_received[4];
MPI_Request reqs_curr_received[4];
int count_prev_neighbours = 0;
int count_curr_neighbours = 0;

// Communicate with north
if (north != MPI_PROC_NULL){
    MPI_Isend(&u_previous[1][1], 1, row, north, 0, MPI_COMM_WORLD, &reqs_prev_sent[count_prev_neighbours]);
    MPI_Irecv(&u_previous[0][1], 1, row, north, 0, MPI_COMM_WORLD, &reqs_prev_received[count_prev_neighbours]);
    count_prev_neighbours++;
}

// Communicate with south
if (south != MPI_PROC_NULL){
    MPI_Isend(&u_previous[local[0]][1], 1, row, south, 0, MPI_COMM_WORLD, &reqs_prev_sent[count_prev_neighbours]);
    MPI_Irecv(&u_previous[local[0]+1][1], 1, row, south, 0, MPI_COMM_WORLD, &reqs_prev_received[count_prev_neighbours]);
    count_prev_neighbours++;
}

// Communicate with east
if (east != MPI_PROC_NULL){
    MPI_Isend(&u_previous[1][local[1]], 1, col, east, 0, MPI_COMM_WORLD, &reqs_prev_sent[count_prev_neighbours]);
    MPI_Irecv(&u_previous[1][local[1]+1], 1, col, east, 0, MPI_COMM_WORLD, &reqs_prev_received[count_prev_neighbours]);
    count_prev_neighbours++;
}

// Communicate with west
if (west != MPI_PROC_NULL){
    MPI_Isend(&u_previous[1][1], 1, col, west, 0, MPI_COMM_WORLD, &reqs_prev_sent[count_prev_neighbours]);
    MPI_Irecv(&u_previous[1][0], 1, col, west, 0, MPI_COMM_WORLD, &reqs_prev_received[count_prev_neighbours]);
    count_prev_neighbours++;
}

MPI_Waitall(count_prev_neighbours, reqs_prev_received, MPI_STATUSES_IGNORE);

gettimeofday(&tcs, NULL);

RedSOR(u_previous, u_current, i_min, i_max, j_min, j_max, omega);

gettimeofday(&tcf, NULL);
tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;

// Communicate with north
if (north != MPI_PROC_NULL){
    MPI_Isend(&u_current[1][1], 1, row, north, 0, MPI_COMM_WORLD, &reqs_curr_sent[count_curr_neighbours]);
    MPI_Irecv(&u_current[0][1], 1, row, north, 0, MPI_COMM_WORLD, &reqs_curr_received[count_curr_neighbours]);
    count_curr_neighbours++;
}

// Communicate with south
if (south != MPI_PROC_NULL){
    MPI_Isend(&u_current[local[0]][1], 1, row, south, 0, MPI_COMM_WORLD, &reqs_curr_sent[count_curr_neighbours]);
    MPI_Irecv(&u_current[local[0]+1][1], 1, row, south, 0, MPI_COMM_WORLD, &reqs_curr_received[count_curr_neighbours]);
    count_curr_neighbours++;
}

// Communicate with east
if (east != MPI_PROC_NULL){
    MPI_Isend(&u_current[1][local[1]], 1, col, east, 0, MPI_COMM_WORLD, &reqs_curr_sent[count_curr_neighbours]);
    MPI_Irecv(&u_current[1][local[1]+1], 1, col, east, 0, MPI_COMM_WORLD, &reqs_curr_received[count_curr_neighbours]);
    count_curr_neighbours++;
}

// Communicate with west
if (west != MPI_PROC_NULL){
    MPI_Isend(&u_current[1][1], 1, col, west, 0, MPI_COMM_WORLD, &reqs_curr_sent[count_curr_neighbours]);
    MPI_Irecv(&u_current[1][0], 1, col, west, 0, MPI_COMM_WORLD, &reqs_curr_received[count_curr_neighbours]);
    count_curr_neighbours++;
}

MPI_Waitall(count_curr_neighbours, reqs_curr_received, MPI_STATUSES_IGNORE);

gettimeofday(&tcs, NULL);

BlackSOR(u_previous, u_current, i_min, i_max, j_min, j_max, omega);

gettimeofday(&tcf, NULL);
tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*1e-6;

```

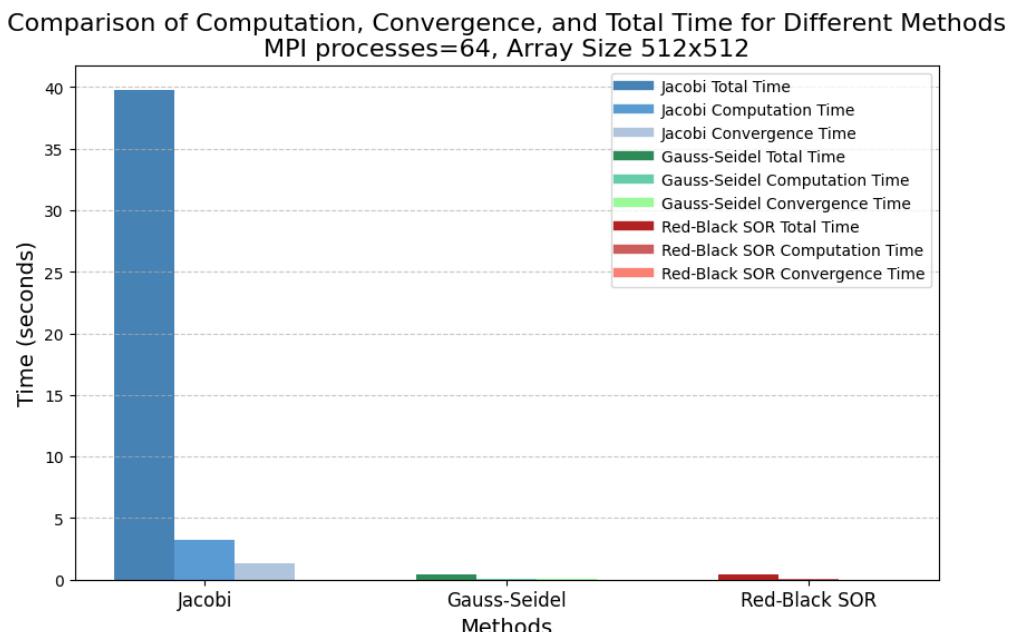
Μετρήσεις για έλεγχο σύγκλισης:

Θα παρουσιάσουμε τώρα διαγράμματα για τις επαναλήψεις που χρειάζεται ο κάθε υπολογιστικός πυρήνας για να συγκλίνει στο τελικό αποτέλεσμα, για ταμπλό μεγέθους 512x512 και 64 MPI διεργασίες. Σημειώνουμε ότι για τη μεταγλώττιση των προγραμμάτων χρησιμοποιήθηκε το επίπεδο βελτιστοποίησης -O3, ενώ κάθε πρόγραμμα εκτελέστηκε τρεις φορές και αξιοποιήσαμε τον μέσο όρο των χρόνων εκτέλεσης για την αξιολόγηση της απόδοσης.

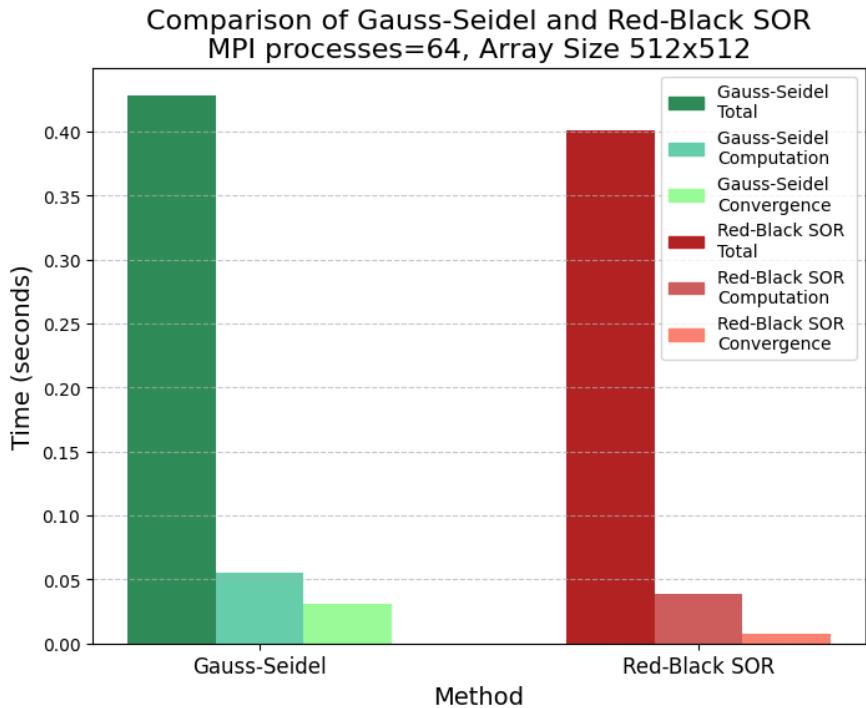
Έχουμε πάρει δύο διαφορετικές εκτελέσεις, η πρώτη εκτέλεση έχει γίνει εκτελώντας τον κώδικα με την εντολή **mpirun --mca btl tcp,self ...**, ενώ στην δεύτερη εκτέλεση χρησιμοποιούμε την εντολή **mpirun --mca btl tcp,self --map-by-node ...**

Η χρήση του **map-by-node** στην εκτέλεση του **mpirun** εξασφαλίζει ομοιόμορφη κατανομή των διεργασιών στους διαθέσιμους κόμβους, βελτιστοποιώντας την απόδοση. Αντί να συγκεντρώνονται πολλές διεργασίες σε έναν κόμβο, κατανέμονται κυκλικά, μειώνοντας τη συμφόρηση στην επικοινωνία και εξισορροπώντας το υπολογιστικό φορτίο. Αυτό οδηγεί σε καλύτερη αξιοποίηση των πόρων και συχνά σε μικρότερους χρόνους εκτέλεσης. Η στρατηγική αυτή είναι ιδιαίτερα χρήσιμη σε υπολογιστικά συστήματα πολλαπλών κόμβων, όπου η αποδοτική διαχείριση των διεργασιών επηρεάζει άμεσα την απόδοση.

α) Μετρήσεις σύγκλισης δίχως την χρήση της **map-by-node**



Για λόγους ευκρίνειας παρουσιάζουμε ξεχωριστό διάγραμμα για τις μεθόδους Gauss-Seidel SOR και Red-Black SOR, οι οποίες παρουσιάζουν συντρηπτικά καλύτερα αποτελέσματα από τη μέθοδο Jacobi:



Παραθέτουμε ακόμα και τις επαναλήψεις που χρειάστηκε κάθε μέθοδος μέχρι την σύγκλιση:

Μέθοδος	Επαναλήψεις
Jacobi	236001
Gauss-Seidel SOR	1501
Red-Black SOR	1201

Σχολιασμός και συμπεράσματα:

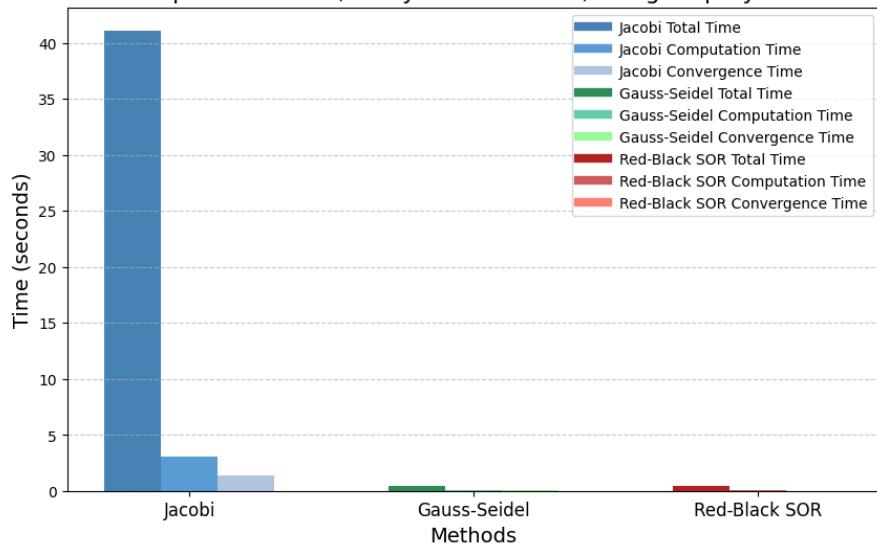
Παρατηρούμε, όπως αναμενόταν, ότι η μέθοδος Jacobi συγκλίνει σημαντικά πιο αργά. Αυτό οφείλεται στο γεγονός ότι, όπως περιγράψαμε προηγουμένως, η μέθοδος χρησιμοποιεί δεδομένα μόνο από την προηγούμενη επανάληψη, με αποτέλεσμα να απαιτούνται περισσότερες επαναλήψεις για τη σύγκλιση. Αντίθετα, η μέθοδος Gauss-Seidel SOR επιταχύνει τη διαδικασία αξιοποιώντας έναν έξυπνο μηχανισμό: τα κελιά των βόρειων και δυτικών γειτόνων έχουν ήδη υπολογιστεί στην τρέχουσα επανάληψη, επιτρέποντας τη χρήση αυτών των πιο πρόσφατων τιμών αντί των δεδομένων της προηγούμενης επανάληψης. Αυτό οδηγεί σε ταχύτερη σύγκλιση. Σύμφωνα με τις μετρήσεις μας, η μέθοδος Gauss-Seidel SOR

συγκλίνει 98.92% γρηγορότερα σε συνολικό χρόνο εκτέλεσης και απαιτεί 99.36% λιγότερες επαναλήψεις σε σχέση με τη μέθοδο Jacobi.

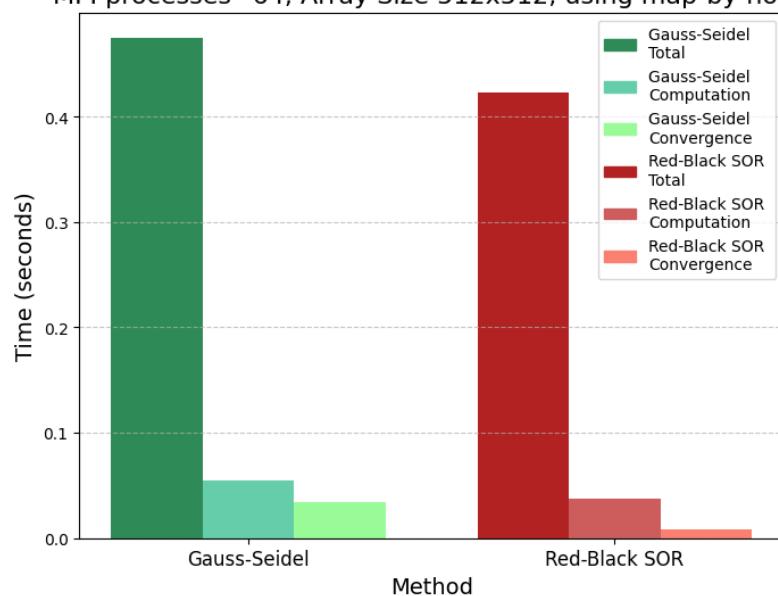
Η μέθοδος Red-Black SOR βελτιώνει περαιτέρω τη σύγκλιση, χωρίζοντας το πλέγμα σε δύο ομάδες κελιών, παρόμοια με μια "σκακιέρα", επιτρέποντας τον παράλληλο υπολογισμό τους. Έτσι, περισσότερες διεργασίες μπορούν να χρησιμοποιήσουν άμεσα τις πιο πρόσφατες υπολογισμένες τιμές της τρέχουσας επανάληψης, επιταχύνοντας περαιτέρω τη σύγκλιση σε σύγκριση με τη Gauss-Seidel SOR. Οι μετρήσεις μας δείχνουν ότι η Red-Black SOR επιτυγχάνει 6.33% ταχύτερη σύγκλιση στον συνολικό χρόνο εκτέλεσης και απαιτεί 19.99% λιγότερες επαναλήψεις σε σχέση με τη Gauss-Seidel SOR.

β) Μετρήσεις σύγκλισης χρησιμοποιώντας την map-by-node

Comparison of Computation, Convergence, and Total Time for Different Methods
MPI processes=64, Array Size 512x512, using map-by-node



Comparison of Gauss-Seidel and Red-Black SOR
MPI processes=64, Array Size 512x512, using map-by-node



Παραθέτουμε πάλι τους αριθμούς επαναλήψεων που καταγράψαμε για κάθε μέθοδο:

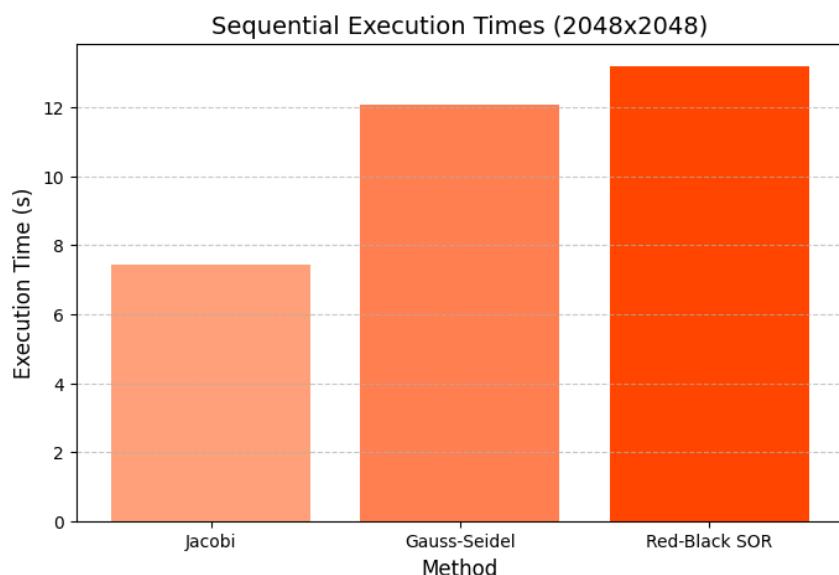
Μέθοδος	Επαναλήψεις
Jacobi	236001
Gauss-Seidel SOR	1501
Red-Black SOR	1201

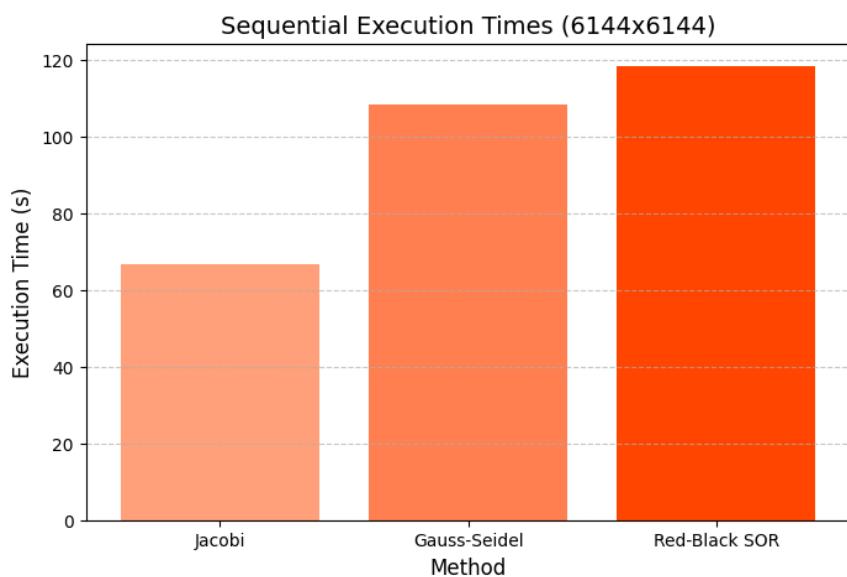
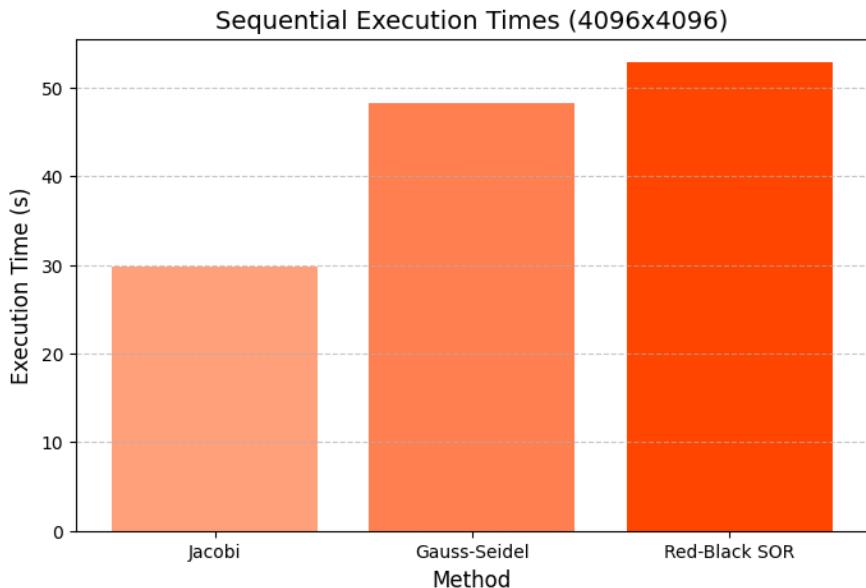
Σχολιασμός και συμπεράσματα:

Παρατηρούμε πως η εκτέλεση χρησιμοποιώντας την map-by-node δεν επηρεάζει την επίδοση των υπολογιστικών πυρήνων ως προς τη σύγκλιση.

Μετρήσεις χωρίς έλεγχο σύγκλισης

Αρχικά παρουσιάζουμε τους σειριακούς χρόνους εκτέλεσης για κάθε μέγεθος πίνακα που θα εξετάσουμε: 2048x2048, 4096x4096 και 6144x6144 για τους τρεις υπολογιστικούς πυρήνες:





Σχολιασμός:

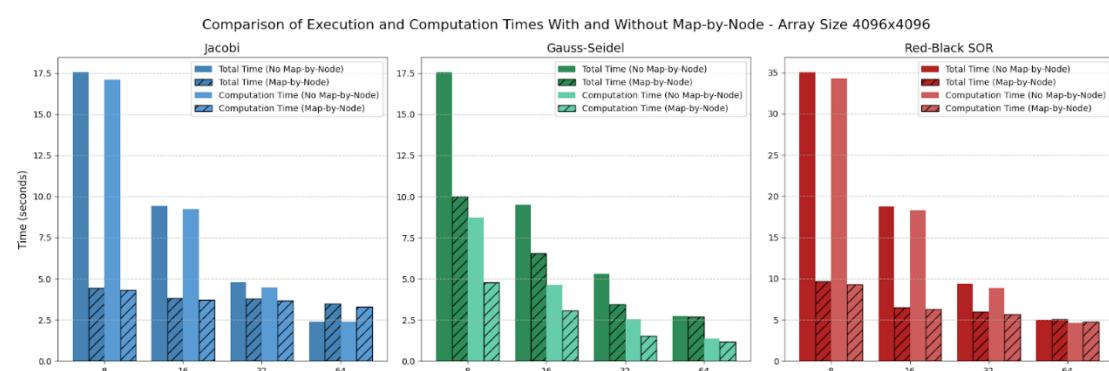
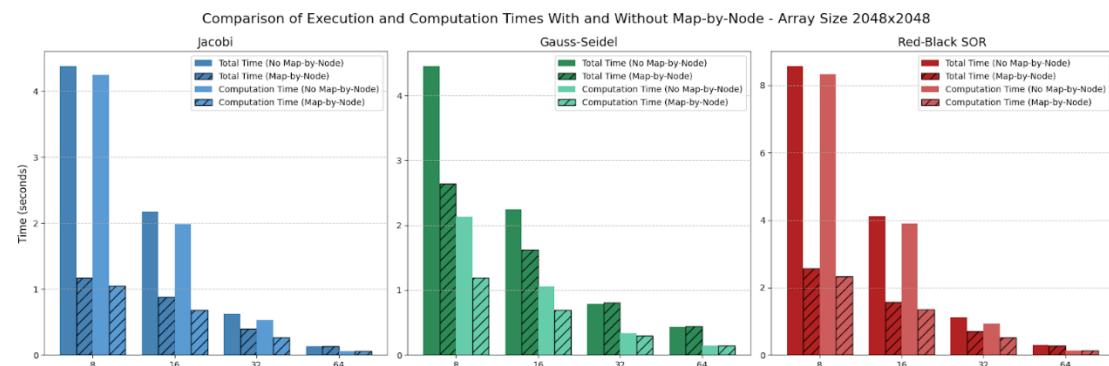
Από τις γραφικές παραστάσεις,, παρατηρούμε ότι ο Jacobi έχει τον μικρότερο χρόνο εκτέλεσης, γεγονός που υποδηλώνει ότι έχει τον ελαφρύτερο υπολογιστικό φορτίο ανά επανάληψη. Ο Gauss-Seidel είναι σημαντικά πιο αργός, γεγονός που υποδηλώνει ότι απαιτεί περισσότερους υπολογισμούς ανά βήμα. Η Red-Black SOR είναι η πιο αργή από όλες, πιθανότατα λόγω του πρόσθετου κόστους διαχωρισμού των στοιχείων και της επιπλέον υπολογιστικής επιβάρυνσης από τις ενημερώσεις στις υπολογιστικές θέσεις. Έτσι, ο συνολικός υπολογιστικός φόρτος αυξάνεται από Jacobi προς Red-Black SOR, καθιστώντας τις τελευταίες μεθόδους πιο απαιτητικές σε σειριακή εκτέλεση.

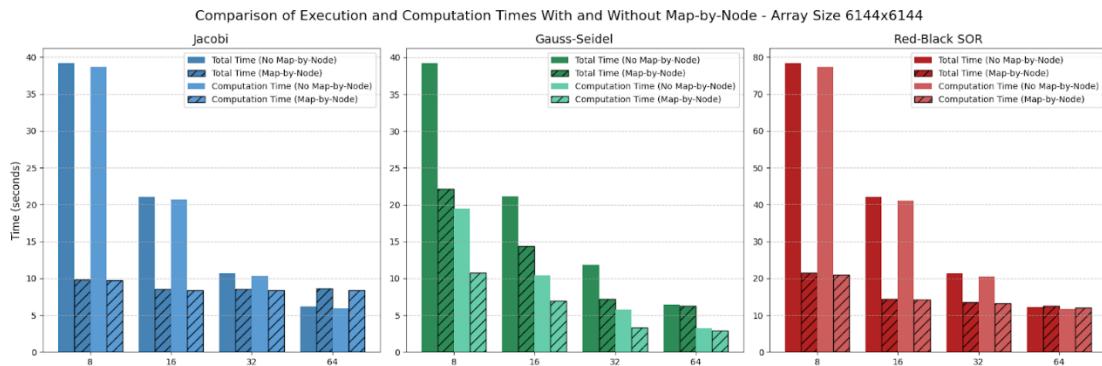
Ακολουθώντας την ίδια προσέγγιση που εφαρμόσαμε για τις μετρήσεις σύγκλισης, πραγματοποιήσαμε εκτελέσεις τόσο χωρίς όσο και με τη χρήση της επιλογής **map-by-node** για την εκτέλεση του παράλληλου κώδικα. Στόχος μας είναι να συγκρίνουμε την επίδραση της κατανομής των διεργασιών στην απόδοση.

Παραθέτουμε απευθείας τους χρόνους εκτέλεσης και για τις δύο περιπτώσεις, ώστε να αξιολογήσουμε ποια προσέγγιση συμβάλλει στη βελτιστοποίηση της παραλληλοποίησης.

Σε κάθε περίπτωση έχουμε εκτελέσει τους υπολογιστικούς πυρήνες για 256 επαναλήψεις και θα πάρουμε μετρήσεις **για μεγέθη πίνακα 2048x2048, 4096x4096 και 6144x6144**, ενώ θα εκτελούμε με διαφορετικό **αριθμό MPI διεργασιών {1, 2, 4, 8, 16, 32, 64}**.

Για λόγους εποπτείας στα παρακάτω διαγράμματα παρουσιάζουμε μετρήσεις μονάχα για αριθμό διεργασιών {8, 16, 32, 64} καθώς δεν εμποδίζει την αποτελεσματική διεξαγωγή συμπερασμάτων.





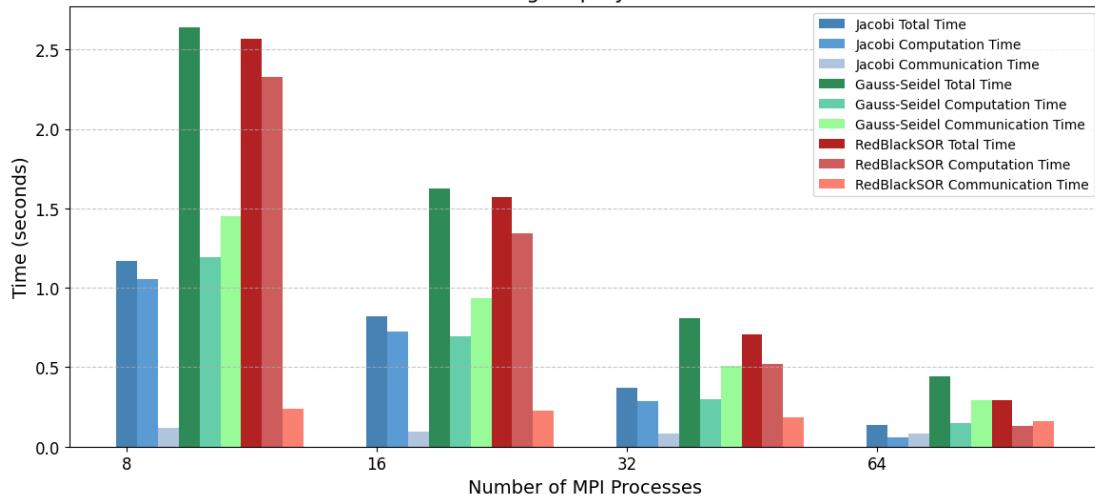
Σχολιασμός:

Παρατηρούμε πως η χρήση του map-by-node βελτιώνει σημαντικά την επίδοση, καθώς για όλα τα configurations που εξετάζουμε, η δεύτερη εκτέλεση είναι αισθητά ταχύτερη από την πρώτη. Αυτό συμβαίνει επειδή το map-by-node κατανέμει τις διεργασίες ομοιόμορφα στους φυσικούς κόμβους, αποφεύγοντας τον υπερκορεσμό συγκεκριμένων μονάδων υπολογισμού. Με αυτόν τον τρόπο, μειώνεται η συμφόρηση στη μνήμη και την επικοινωνία, επιτρέποντας καλύτερη εκμετάλλευση των διαθέσιμων υπολογιστικών πόρων και βελτιώνοντας την κλιμάκωση της εκτέλεσης.

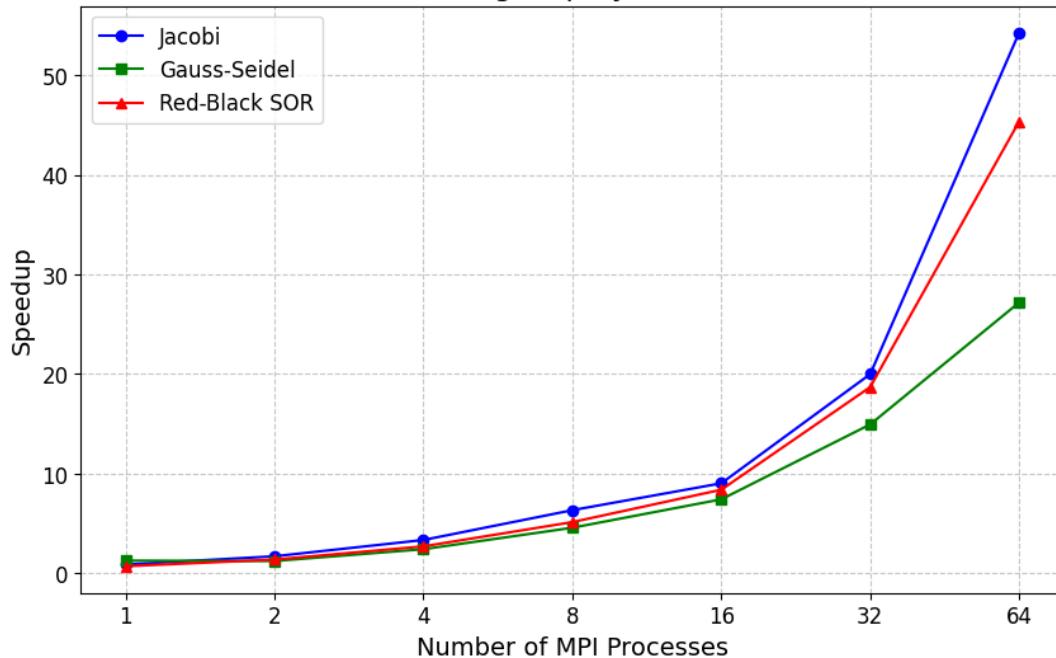
Θα παρουσιάσουμε τώρα αναλυτικά τα διαγράμματα για τον χρόνο εκτέλεσης και την επιτάχυνση από τις μετρήσεις που πήραμε εκτελώντας τα προγράμματα με την Map-by-node καθώς αποδείχθηκε αποτελεσματικότερη. Οι μετρήσεις που παρουσιάζονται είναι σύμφωνες με το configuration που περιγράφηκε παραπάνω.

Πάλι για λόγους ευκρίνειας για τα διαγράμματα χρόνου θα παρουσιάσουμε αποτελέσματα εκτέλεσης με 8, 16, 32 και 64 MPI διεργασίες. Ο χρόνος επικοινωνίας (communication time) που παρουσιάζεται στα διαγράμματα έχει υπολογιστεί ως η διαφορά του συνολικού χρόνου (total time) και του χρόνου υπολογισμών (computational time).

Comparison of Computation, Communication, and Total Time for Different Methods - 2048x2048 using map-by-node



Speedup Comparison for Different Methods - 2048x2048 using map-by-node



Σχολιασμός και συμπεράσματα:

Από το διάγραμμα χρόνου εκτέλεσης παρατηρούμε ότι ο χρόνος επικοινωνίας για τον Gauss-Seidel είναι δυσανάλογα μεγάλος σε όλες τις περιπτώσεις. Αυτό υποδεικνύει, όπως θα δούμε στην πορεία, ότι η μη συμμετρική επικοινωνία του Gauss-Seidel δημιουργεί σοβαρό bottleneck, με αποτέλεσμα οι διεργασίες στις «κάτω-δεξιές» θέσεις να αναγκάζονται να περιμένουν σημαντικά για την άφιξη των οριακών δεδομένων. Επιπλέον, για τους αλγορίθμους Jacobi και Red-Black παρατηρούμε ότι καθώς αυξάνεται ο αριθμός των πυρήνων, ο χρόνος υπολογισμού γίνεται όλο και μικρότερο ποσοστό του συνολικού χρόνου εκτέλεσης. Αυτό είναι λογικό, δεδομένου ότι η παραλληλοποίηση των

υπολογισμών βελτιώνεται, ενώ το μερίδιο του χρόνου επικοινωνίας είτε παραμένει σταθερό είτε αυξάνεται λόγω της επιπρόσθετης επικοινωνίας μεταξύ των διεργασιών. Τέλος, αξίζει να τονίσουμε ότι για σταθερό αριθμό επαναλήψεων ο Jacobi είναι ταχύτερος από όλους τους αλγορίθμους σε επίπεδο χρόνου ανά επανάληψη, όμως η υπεροχή των άλλων μεθόδων οφείλεται στον πολύ ταχύτερο ρυθμό σύγκλισης που επιτυγχάνουν, γεγονός που τους καθιστά πιο αποδοτικούς όταν ο συνολικός χρόνος έως τη σύγκλιση είναι κρίσιμος, γεγονός που αναλύσαμε στις προηγούμενες μετρήσεις μας.

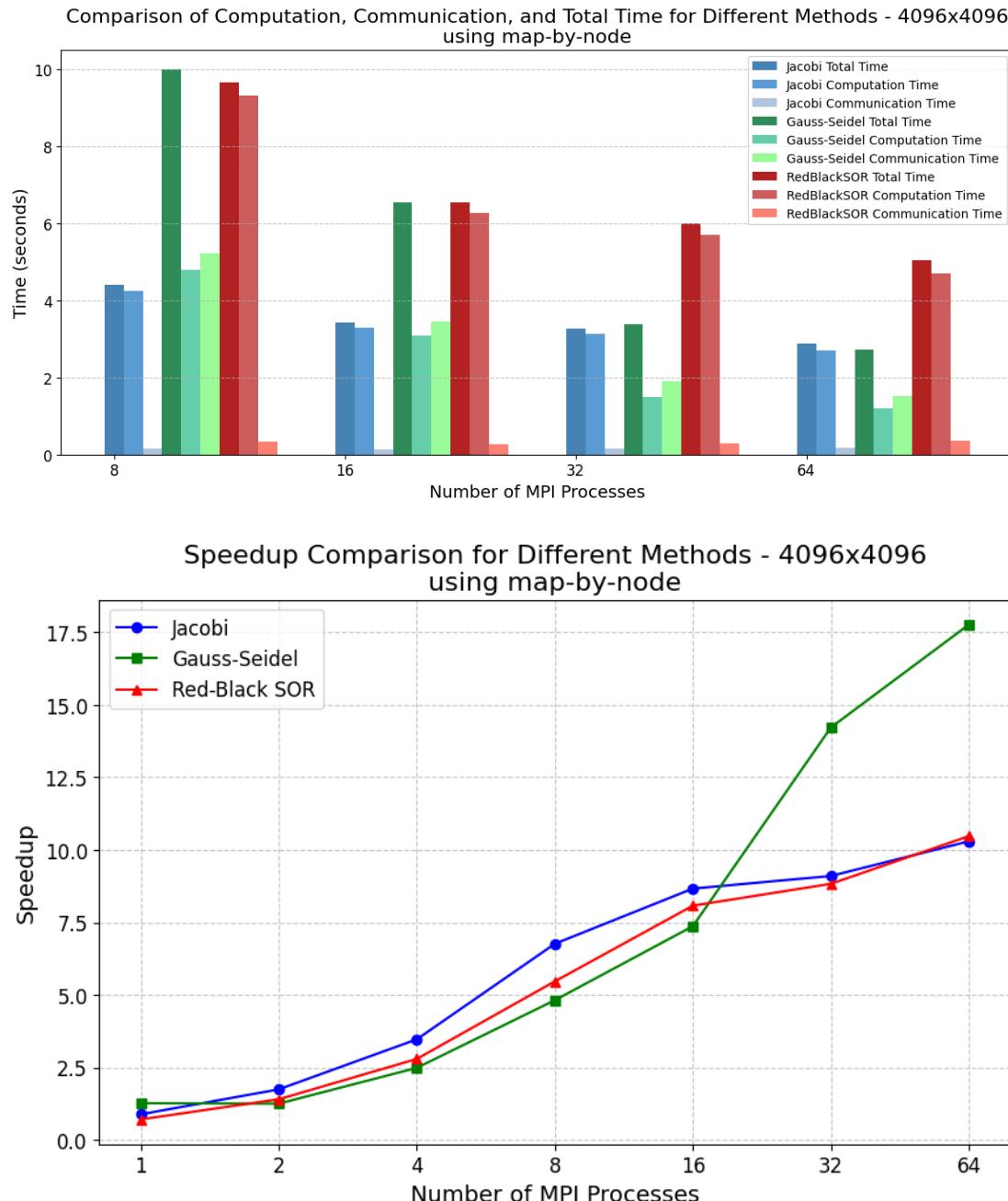
Από το διάγραμμα επιτάχυνσης παρατηρούμε ότι αρχικά όλες οι μέθοδοι επιταχύνονται καθώς αυξάνουμε τον αριθμό των cores, πράγμα που υποδηλώνει ότι το όφελος από τον παραλληλισμό υπερτερεί του κόστους της επικοινωνίας. Επίσης, παρατηρούμε ότι οι μέθοδοι Jacobi και Red-Black επιτυγχάνουν πολύ καλύτερο scaling σε σύγκριση με την Gauss-Seidel, η οποία παρουσιάζει σημαντικά πιο αργή επιτάχυνση καθώς προσθέτουμε cores. Αυτό μπορούμε να το αποδώσουμε στο γεγονός ότι, στις μεθόδους Jacobi και Red-Black, οι διεργασίες εκτελούν τις ανταλλαγές δεδομένων και τους υπολογισμούς σε συγχρονισμένες φάσεις, ενώ στην Gauss-Seidel οι αλυσιδωτές εξαρτήσεις (δηλαδή οι εξαρτήσεις κάθε κελιού από το κελί στα αριστερά και επάνω) δημιουργούν ασυμμετρία στην επικοινωνία.

Συγκεκριμένα, στη μέθοδο Jacobi, σε κάθε iteration όλες οι διεργασίες πρώτα ανταλλάσσουν δεδομένα με τους γείτονές τους και έπειτα μπαίνουν σε μια φάση υπολογισμού, διασφαλίζοντας έτσι χρονική συμμετρία και συγχρονισμό μεταξύ τους. Αντίστοιχα, στη μέθοδο Red-Black, αρχικά όλες οι διεργασίες ανταλλάσσουν δεδομένα του προηγούμενου time step, εκτελούν τους “κόκκινους” υπολογισμούς, ανταλλάσσουν δεδομένα του τρέχοντος time step και τέλος πραγματοποιούν τους “μαύρους” υπολογισμούς, γεγονός που εξασφαλίζει επίσης ομοιόμορφο συγχρονισμό.

Από την άλλη πλευρά, στη μέθοδο Gauss-Seidel οι εξαρτήσεις κάθε κελιού από το κελί στα αριστερά και επάνω οδηγούν σε ασυμμετρία κατά την επικοινωνία. Συγκεκριμένα, η διεργασία στην πάνω αριστερή γωνία πρέπει πρώτα να ολοκληρώσει τους υπολογισμούς της και να διαδώσει τα αποτελέσματά της προς τα δεξιά και προς τα κάτω, και στη συνέχεια κάθε επόμενη διεργασία αναγκάζεται να περιμένει να λάβει τα δεδομένα από την αριστερή και την επάνω διεργασία προτού ξεκινήσει τους δικούς της υπολογισμούς. Αυτό το αλυσιδωτό φαινόμενο καθυστερεί ιδιαίτερα τις διεργασίες στις κάτω δεξιές θέσεις, κάνοντας το κάθε iteration να διαρκεί περισσότερο σε σχέση με τις άλλες μεθόδους.

Συμπληρωματικά, αξίζει να σημειωθεί ότι οι ασυμμετρίες αυτές οδηγούν σε μειωμένη παράλληλη απόδοση στη Gauss-Seidel, καθώς οι διεργασίες καταλήγουν να περιμένουν τα αποτελέσματα των γειτονικών τους, γεγονός που

αυξάνει το συνολικό χρόνο κάθε iteration και μειώνει το scaling σε υψηλότερα πλήθη cores.



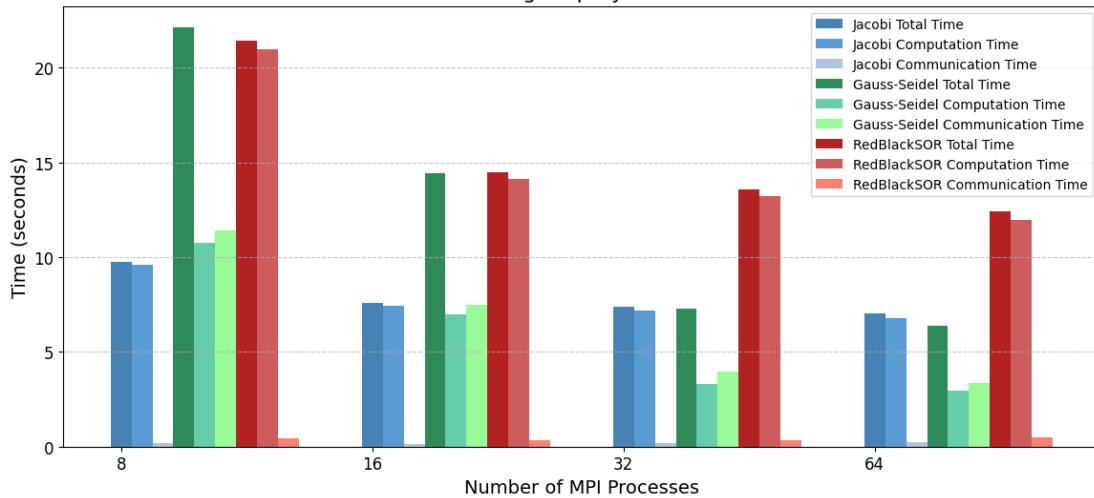
Σχολιασμός και συμπεράσματα:

Παρατηρώντας τα διαγράμματα χρόνου μπορούμε να επιβεβαιώσουμε την υπόθεση που κάναμε προηγουμένως. Συγκεκριμένα, βλέπουμε ότι το “μπλοκάρισμα” στο speedup για τους αλγορίθμους Jacobi και Red-Black οφείλεται κυρίως στον χρόνο υπολογισμών, ενώ ο χρόνος επικοινωνίας αποτελεί ένα πολύ μικρό ποσοστό του συνολικού χρόνου εκτέλεσης. Αυτή η παρατήρηση

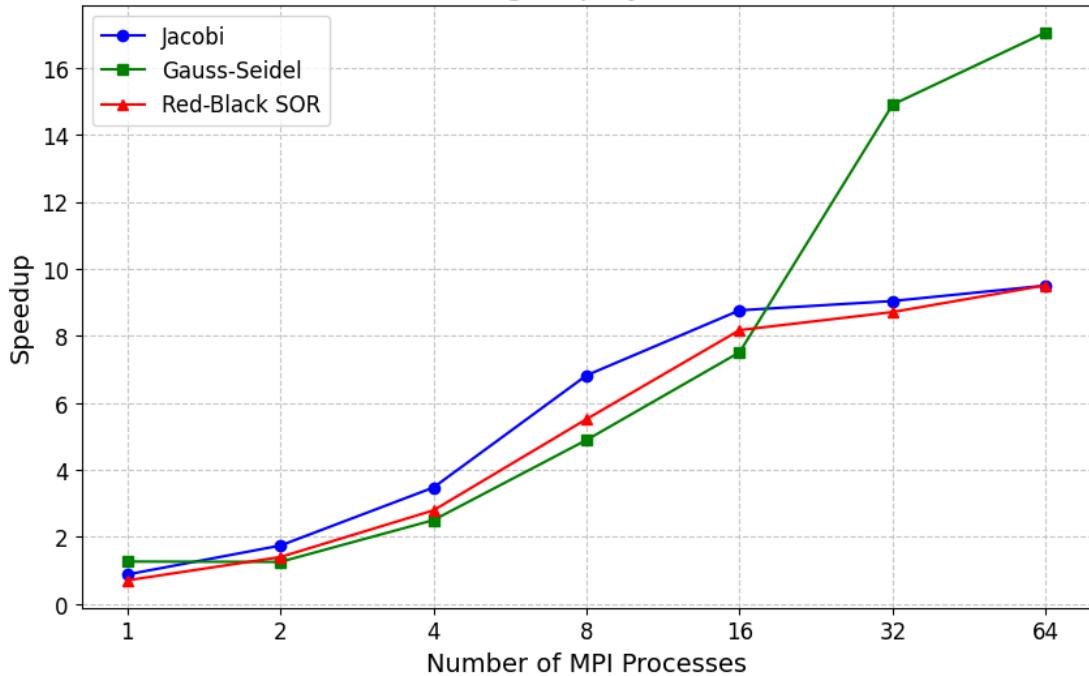
υποστηρίζει την ιδέα ότι οι καθυστερήσεις οφείλονται σε συγκρούσεις για την πρόσβαση στη Cache, ιδιαίτερα όταν ο αριθμός των threads αυξάνεται και τα δεδομένα δεν χωρούν πλήρως στην Cache κάθε CPU. Από την άλλη, ο Gauss-Seidel, στις περιπτώσεις των 32 και 64 threads, καταφέρνει να μειώσει σημαντικά τον χρόνο υπολογισμών του σε χαμηλότερα επίπεδα από τους άλλους δύο αλγορίθμους, αλλά ταυτόχρονα εμφανίζει σημαντική καθυστέρηση λόγω της επικοινωνίας, καθώς αρκετά threads πρέπει να περιμένουν να λάβουν τα δεδομένα των προηγούμενων υπολογισμών. Αξίζει να σημειωθεί ότι, παρά το υψηλότερο communication overhead, ο Gauss-Seidel καταφέρνει πρακτικά ίδιο χρόνο ανά επανάληψη με τον Jacobi για 32 και 64 processes, ο οποίος ήταν ταχύτερος ανά επανάληψη για τον πίνακα 2048x2048. Αυτό το αποτέλεσμα αναδεικνύει το δίλημμα μεταξύ της ταχύτητας των υπολογισμών και του κόστους επικοινωνίας στις παραλληλοποιημένες υλοποιήσεις: ενώ οι μέθοδοι Jacobi και Red-Black περιορίζονται από την επιβράδυνση λόγω των συγκρούσεων στην Cache, ο Gauss-Seidel εκμεταλλεύεται καλύτερα τον παραληλισμό για τον υπολογισμό, αν και υποφέρει από ασυμμετρία στην επικοινωνία.

Από το διάγραμμα επιτάχυνσης στο 4096x4096 grid παρατηρούμε αρκετά διαφορετική συμπεριφορά σε σχέση με το προηγούμενο configuration. Εδώ οι μέθοδοι Jacobi και Red-Black πρακτικά σταματούν να κάνουν scale μετά από 16 threads, ενώ πριν είχαν δυνατότητα scaling μέχρι τα 64. Αντίθετα, η μέθοδος Gauss-Seidel διατηρεί το scaling που παρατηρήθηκε προηγουμένως. Αυτή η διαφοροποιημένη συμπεριφορά οφείλεται πιθανώς σε επιβράδυνση λόγω των λειτουργιών πρόσβασης στη μνήμη, κυρίως όταν ενεργοποιούνται πολλαπλά threads σε κάθε CPU. Για τις μεθόδους Red-Black και Jacobi, το scaling σταματά στα 16 threads, διότι, δεδομένου ότι η ουρά parlab διαθέτει 8 nodes με 2 τετραπύρηνες CPUs ανά node, 8nodes x 8ppn, αυτό αντιστοιχεί στο μέγιστο πλήθος threads για το οποίο κάθε CPU λειτουργεί με ένα thread. Όταν χρησιμοποιούνται 32 και 64 threads, πολλαπλά cores κάθε CPU χρησιμοποιούνται ταυτόχρονα, προκαλώντας ανταγωνισμό για την Cache καθώς ένας μεγαλύτερος όγκος δεδομένων πρέπει να χωρέσει σε ένα μικρότερο χώρο μνήμης Cache. Από την άλλη πλευρά, στον Gauss-Seidel τα threads δεν εκτελούν τους υπολογισμούς τους εξ ολοκλήρου συγχρονισμένα, κάτι που μειώνει τις συνθήκες ανταγωνισμού για την Cache και επιτρέπει στο scaling να παραμείνει σταθερό ακόμη και για μεγαλύτερο grid.

Comparison of Computation, Communication, and Total Time for Different Methods - 6144x6144 using map-by-node



Speedup Comparison for Different Methods - 6144x6144 using map-by-node



Σχολιασμός και συμπεράσματα:

Το διάγραμμα Speedup για το συγκεκριμένο grid είναι μια παρόμοια έκδοση με αυτό που παρατηρήσαμε στην περίπτωση του 4096x4096, με την διαφορά ότι τώρα είναι πολύ πιο έντονα φανερό ότι δεν υπάρχει scale για το Jacobi και RedBlackSOR μετά τα 16 νήματα, ενώ και ο Gauss-Seidel δείχνει ότι παύει να κάνει scale στα 64 νήματα. Τα διαγράμματα χρόνου εκτέλεσης εμφανίζουν την ίδια γενική μορφή, με διαφορές μόνο στους απόλυτους χρόνους. Οι αλγόριθμοι δείχνουν συνεπή συμπεριφορά όπως και στο προηγούμενο μέγεθος grid. Αυτό επιβεβαιώνει τις προηγούμενες υποθέσεις μας ότι το “μπλοκάρισμα” στο Speedup

για τις μεθόδους Jacobi και Red-Black οφείλεται στις περιορισμένες επιδόσεις της Cache, φαινόμενο που δεν εμφανίζεται στην υλοποίηση της Gauss-Seidel. Δεδομένου ότι ο πίνακας είναι ακόμη μεγαλύτερος, είναι αναμενόμενο να παρατηρούμε το ίδιο φαινόμενο και σε αυτή την περίπτωση.

Εξάρτηση μεθόδων από το μέγεθος ταμπλό

Για καλύτερη εποπτεία των συμπερασμάτων μας θα παρουσιάσουμε συνοπτικά την εξάρτηση που διαπιστώσαμε να έχει η παράλληλη έκδοση κάθε μεθόδου από το μέγεθος του ταμπλό:

✓ **Μέθοδος Jacobi:**

Η μέθοδος Jacobi παρουσιάζει καλή παραλληλοποίηση, ιδιαίτερα για μικρά μεγέθη πινάκων, όπως φαίνεται στο πρώτο διάγραμμα. Ωστόσο, καθώς το μέγεθος του πλέγματος αυξάνεται, η επιτάχυνση σταματά να βελτιώνεται σημαντικά όταν ο αριθμός των διεργασιών μεγαλώνει. Αυτό οφείλεται στο υψηλό υπολογιστικό και επικοινωνιακό κόστος της μεθόδου, καθώς κάθε επανάληψη χρησιμοποιεί μόνο τιμές από την προηγούμενη, καθυστερώντας τη σύγκλιση και περιορίζοντας τα οφέλη από την αύξηση των διεργασιών σε μεγάλους πίνακες.

✓ **Μέθοδος Gauss-Seidel:**

Η μέθοδος Gauss-Seidel εμφανίζει μέτρια παράλληλη αποδοτικότητα σε μικρούς πίνακες, αλλά κλιμακώνεται πολύ καλύτερα σε μεγαλύτερα μεγέθη, όπως φαίνεται στα διαγράμματα για 4096×4096 και 6144×6144 . Αυτό συμβαίνει γιατί αξιοποιεί τις ήδη ενημερωμένες τιμές στην ίδια επανάληψη, μειώνοντας τον συνολικό αριθμό των επαναλήψεων. Ωστόσο, καθώς αυξάνεται ο αριθμός των διεργασιών, το κόστος επικοινωνίας μεγαλώνει, περιορίζοντας την επιτάχυνση, αν και εξακολουθεί να υπερέχει σε σχέση με τη Jacobi για μεγάλους πίνακες.

✓ **Μέθοδος Red-Black SOR:**

Η μέθοδος Red-Black SOR ακολουθεί παρόμοια τάση με τη Gauss-Seidel, επιδεικνύοντας καλή επιτάχυνση για μικρό αριθμό διεργασιών. Ωστόσο, για μεγαλύτερα μεγέθη πινάκων, η αύξηση της επιτάχυνσης είναι πιο περιορισμένη. Αυτό πιθανότατα οφείλεται στον διαχωρισμό του πλέγματος σε μαύρα και κόκκινα σημεία, γεγονός που εισάγει καθυστερήσεις συγχρονισμού και αυξάνει το κόστος επικοινωνίας. Παρόλο που εκμεταλλεύεται την παράλληλη εκτέλεση, η κλιμάκωσή της περιορίζεται για μεγάλα προβλήματα.