



**Εθνικό Μετσόβιο Πολυτεχνείο Σχολή
Ηλεκτρολόγων Μηχ. και Μηχανικών
Υπολογιστών Εργαστήριο Υπολογιστικών
Συστημάτων**

**Συστήματα Παράλληλης Επεξεργασίας 9ο Εξάμηνο
Ακαδημαϊκό έτος: 2025-2026**

**Ομάδα : parlab16
Γεώργιος Κοκορομύτης / 03120012
Οδυσσέας-Αρθούρος-Ρήγας Τσουκνίδας / 03120043
Κωνσταντίνος Φέζος / 03118076**

1. Εξοικείωση με το περιβάλλον προγραμματισμού - Game of Life

Στην άσκηση αυτή μας ζητείται να παραλληλοποιήσουμε το γνωστό “παιχνίδι” Conway's Game of Life. Μας δίνεται έτοιμος ο σειριακός κώδικας από το εργαστήριο και εμείς πρέπει να χρησιμοποιήσουμε το OpenMP για τη παραλληλοποίηση.

Για λόγους ευκολίας αλλά και αποφυγής σφαλμάτων, θα κάνουμε την μεταγλώττιση με την χρήση του παρακάτω Makefile:

```
all: game

game: Game_Of_Life.c
    gcc -O3 -fopenmp -o game_of_life Game_Of_Life.c

clean:
    rm game_of_life
```

Έπειτα στον C κώδικα παρατηρούμε ότι υπάρχει ένα 3πλο nested loop (γενιές, οριζόντιος άξονας, κάθετος άξονας). Προφανώς δεν μπορούμε να παραλληλοποιήσουμε τις γενιές καθώς η κάθε μία εξαρτάται από την προηγούμενη, αλλά μπορούμε να παραλληλοποιήσουμε τον υπολογισμό για κάθε γραμμή του grid. Κρατάμε ως private τις μεταβλητές για το j και τον τρέχον γείτονα (nbrs) καθώς το κάθε νήμα εκτελεί το j-loop για διαφορετικό i. Παρόλα αυτά διατηρούμε κοινούς τον previous πίνακα (μόνο διαβάζουν από εκεί τα νήματα) καθώς και τον current (κάθε νήμα γράφει σε διαφορετική γραμμή του).

```
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for schedule(static) private(j, nbrs) shared(previous, current, N)
    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;
        }
}
```

Για την μεταγλώττιση θα χρησιμοποιήσουμε το παρακάτω script:

```
#!/bin/bash
```

```

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

## Output and error files
#PBS -o output_files/make.out
#PBS -e output_files/make.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/parlab16/game_of_life
make clean
make

```

Με το παραπάνω script ορίζουμε:

- Το όνομα του Job μας
- Το αρχεία στα οποία θα γραφτεί το standard output και error και τα οποία θα δούμε αργότερα για να βγάλουμε τα συμπεράσματα μας
- Τον αριθμό των nodes, καθώς και τον cores ανά node (εδώ 1 core σε 1 node αφού θέλουμε απλά να μεταγλωτίσουμε τον κώδικα)
- Το άνω χρονικό όριο για την εκτέλεση του Job μας
- Το τι θέλουμε να εκτελεστεί στην συστοιχία. Εδώ πάμε δηλαδή στο directory που βρίσκεται το Makefile και εκεί κάνουμε “make”

Μπορούμε τώρα να υποβάλλουμε το script στην ουρά parlab με το command: **qsub -q parlab make_on_queue.sh**, το οποίο μας επιστρέφει το όνομα του job μας στον torque: **59586.localhost**

Παρατηρούμε στην συνέχεια πως έχουμε πλέον δημιουργηθεί 2 αρχεία το **make.err** (που είναι άδειο) και το **make.out**:

```

rm game_of_life
gcc -O3 -fopenmp -o game_of_life Game_Of_Life.c

```

Καταλαβαίνοντας ότι η μεταγλώττιση εκτελέστηκε σωστά (βλέπουμε και το executable στο directory μας), προχωράμε στο script για να τρέξουμε το executable αυτό στην συστοιχία:

```

#!/bin/bash

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

## Output and error files
#PBS -o output_files/run_k_x.out
#PBS -e output_files/run_k_x.err

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

##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/parlab16/game_of_life
export OMP_NUM_THREADS=k
./game_of_life x y

```

Ανάλογα με τα πόσα cores θέλουμε στο node (λόγω ότι το OpenMP είναι με shared memory, θα περιοριστούμε σε ένα μόνο node) θα αλλάζουμε την τιμή του k. Θέτουμε επίσης τον αριθμό των threads ίσο με το ppn. Τέλος, ορίζουμε τα arguments για το executable με το x να προσδιορίζει το Array Size και το y τον αριθμό των γενιών.

Υποβάλλουμε το script με την εντολή **qsub -q parlab run_on_queue.sh** και αφού τελειώσει και ελέγχουμε πως το .err αρχείο είναι άδειο, βλέπουμε στο .out αρχείο output της μορφής:

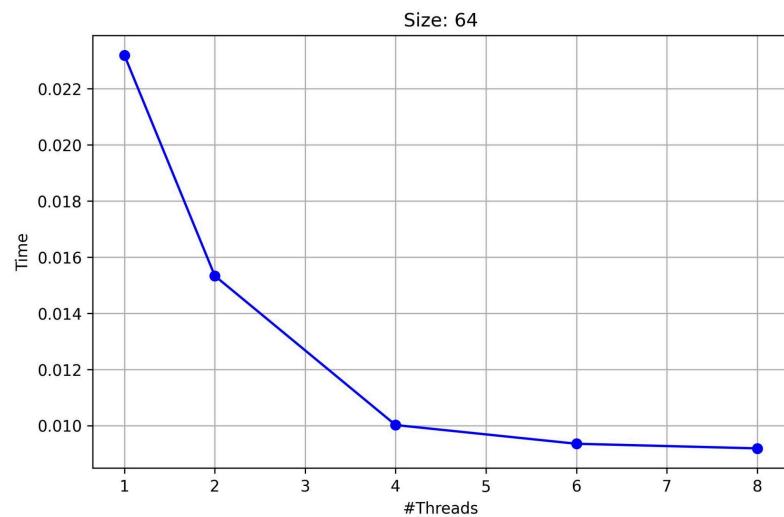
GameOfLife: Size x Steps y Time time_value

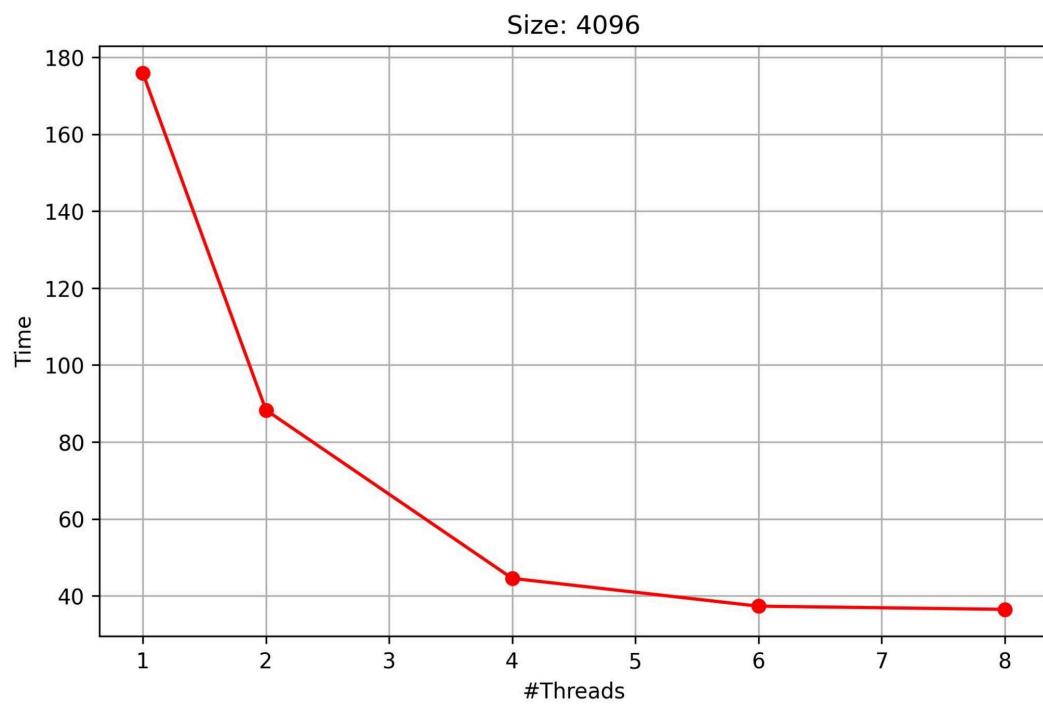
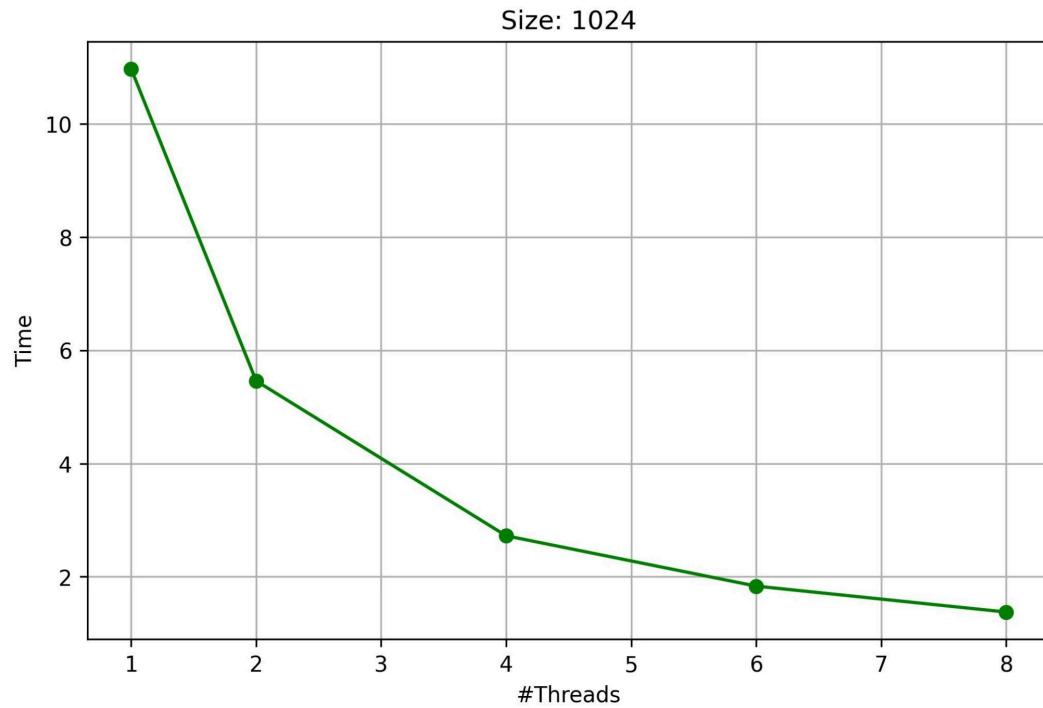
Παρακάτω φαίνονται οι χρόνοι για 1000 γενιές και διαφορετικό αριθμό cores και διαφορετικά array sizes

#cores/Array Size	64	1024	4096
1	0.023195	10.969780	175.931732

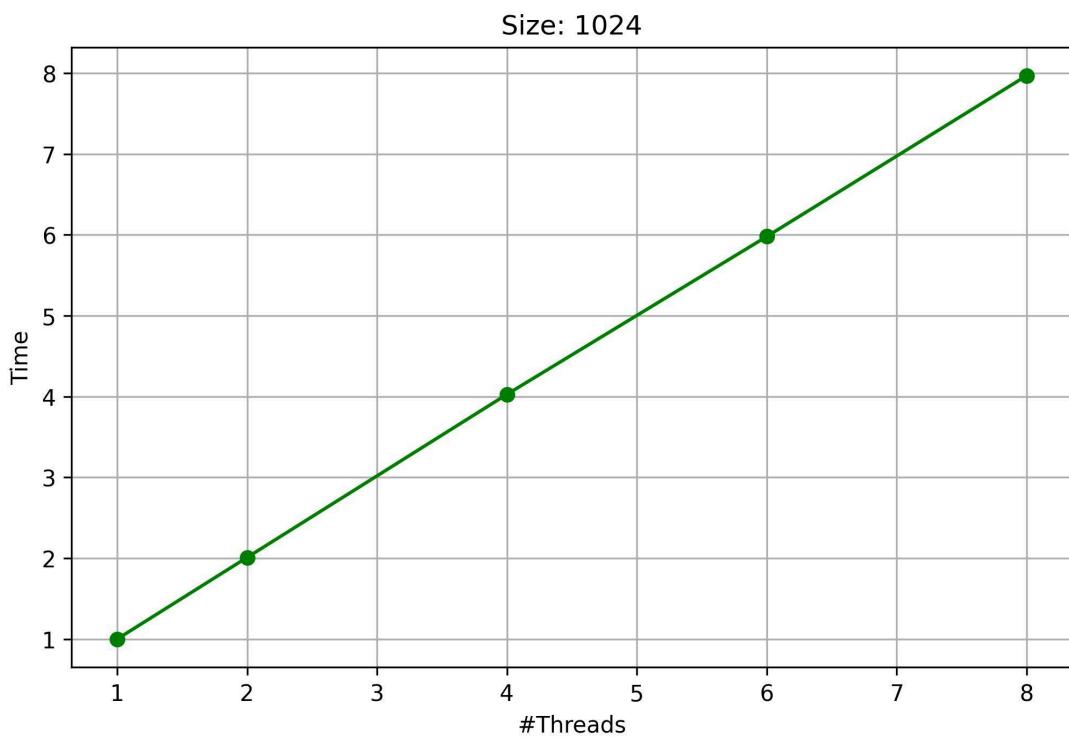
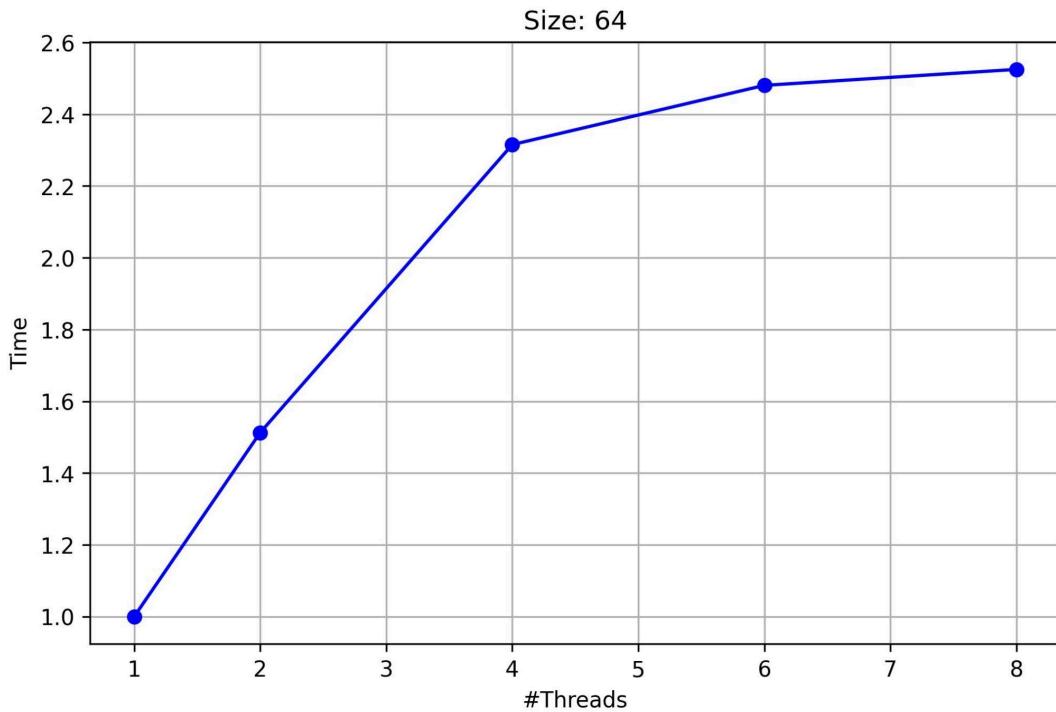
2	0.015333	5.458615	88.303914
4	0.010018	2.723742	44.549190
6	0.009351	1.834500	37.325994
8	0.009186	1.376445	36.491348

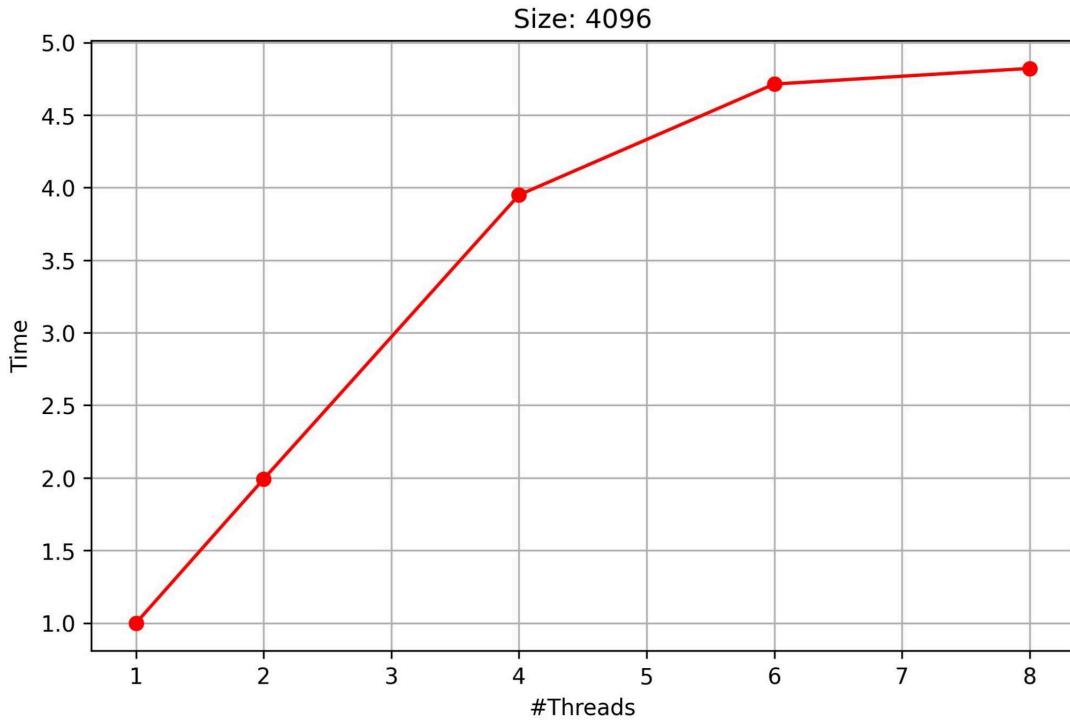
Διαγράμματα Χρόνου





Διαγράμματα Speedup





Σχολιασμός Αποτελεσμάτων

Στο 64×64 grid βλέπουμε ότι έχουμε speedup, αλλά πως αυτό δεν κλιμακώνει και πολύ καλά προσθέτοντας επεξεργαστές. Δεδομένου ότι το μέγεθος του grid είναι αρκετά μικρό ώστε να χωράει στην cache, δεν ευθύνεται η μνήμη για την κακή κλιμάκωση, αλλά το overhead για την δημιουργία και την διαχείριση των επιπλέον νημάτων καθώς και “barriers” των γενιών, διότι τα νήματα αφού ολοκληρώσουν τον υπολογισμό μίας γραμμής πρέπει να περιμένουν και όλα τα υπόλοιπα να τελειώσουν, πριν προχωρήσουν όλα στην επόμενη γενιά.

Στο 1024×1024 grid (που χωράει επίσης στην cache) παρατηρούμε ότι η κλιμάκωση είναι πολύ καλύτερη και έχουμε linear (perfect) speedup, δηλαδή σε ρ επεξεργαστές έχουμε speedup ίσο με ρ. Η εμφανής βελτίωση σε σχέση με το 64×64 οφείλεται στον νόμο του Amdahl. Το σειριακό τμήμα του προγράμματος παραμένει σταθερό, όμως το συνολικό πλήθος των υπολογισμών στα loops αυξάνεται σημαντικά. Ως αποτέλεσμα, το f μειώνεται και τα περιθώρια παραλληλοποίησης αυξάνονται.

Ο λόγος που χάνουμε την καλή κλιμάκωση στο 4096×4096 grid, οφείλεται στο γεγονός ότι πλέον δεν χωράει ολόκληρος ο πίνακας στην cache, άρα πλέον χάνεται χρόνος και σε προσπελάσεις στην μνήμη - η εφαρμογή μας είναι memory bandwidth bound.

Ορθότητα

Τέλος πρέπει να σημειωθεί ότι με την παραπάνω διαδικασία δεν ελέγχουμε αν το παραλληλοποιημένο πρόγραμμα είναι ορθό (δηλαδή παράγει το ίδιο αποτέλεσμα με το σειριακό). Για να το ελέγχουμε αυτό θα προσθέσουμε το flag -DOUTPUT στο compilation και μετά θα ελέγχουμε με το command diff κατά πόσο τα .gif αρχεία των παράλληλων είναι identical με αυτό του σειριακού. Είναι σημαντικό να καταλάβουμε όμως πως αυτό δεν μπορεί να γίνει ταυτόχρονα με την μέτρηση χρόνου, καθώς η διαδικασία παραγωγής output έχει αρκετό overhead, το οποίο μάλιστα παρατηρήσαμε ότι αυξάνεται στην παραλληλοποίηση.

2. Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου kmeans

Σκοπός της συγκεκριμένης άσκησης είναι να παραλληλοποιήσουμε με 2 διαφορετικούς τρόπους τον αλγόριθμο K-means, σε προγραμματιστικό μοντέλο κοινού χώρου διευθύνσεων, με την χρήση του εργαλείου OpenMP.

Ο αλγόριθμος αυτός είναι αλγόριθμος ομαδοποίησης (clustering) που ταξινομεί N αντικείμενα σε k μη επικαλυπτόμενες ομάδες/συστάδες. Παρότι είναι αρκετά απλός, αποτελεί ένα από τα πιο χρήσιμα και διαδεδομένα εργαλεία για την υλοποίηση clustering στη μηχανική μάθηση.

Σε επίπεδο ψευδοκώδικα μπορεί να περιγραφεί ως εξής:

```
until convergence (or fixed loops)
    for each object
        find nearest cluster
    for each cluster
        calculate new cluster center coordinates.
```

Εδώ καλούμαστε να υλοποιήσουμε 2 διαφορετικές παραλληλοποιήσεις, μία “απλοϊκή” με shared clusters - στην οποία πρέπει να εισάγουμε κατάλληλες εντολές συγχρονισμού για να γίνει ορθά η ενημέρωση του πίνακα newClusters, στον οποίο αποθηκεύονται οι νέες υπολογιζόμενες συντεταγμένες των κέντρων των συστάδων - καθώς και μία πιο εξελιγμένη, η οποία διατηρεί τοπικούς πίνακες για κάθε thread (αποφεύγοντας έτσι το overhead του συγχρονισμού) και συνδυάζοντας στο τέλος τα αποτελέσματα με reduction.

Ζητούμενο 1 - Shared Clusters

1.

Στον κώδικα που μας παρέχεται από το εργαστήριο μπορούμε να δούμε στην σειριακή υλοποίηση του αλγορίθμου το βασικό σώμα του αλγορίθμου:

```

do {
    // before each loop, set cluster data to 0
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++)
            newClusters[i*numCoords + j] = 0.0;
        newClusterSize[i] = 0;
    }
    delta = 0.0;
    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 objects located within
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            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];
            }
        }
    }
    // Get fraction of objects whose membership changed during this loop. This is used as a
convergence criterion.
    delta /= numObjs;
    loop++;
    printf("\r\tcompleted loop %d", loop);
    fflush(stdout);
} while (delta > threshold && loop < loop_threshold);

```

Μπορούμε εύκολα να δούμε εδώ πως το for loop που διατρέχει όλα τα αντικείμενα, υπολογίζει το νέο nearest cluster για κάθε ένα και κάνει assign το σημείο στο νέο cluster μπορεί να παραλληλοποιηθεί. Αυτό συμβαίνει καθώς αφού κάθε iteration του αφορά ένα διαφορετικό αντικείμενο, και συνεπώς είναι ανεξάρτητο από τα υπόλοιπα. Αντίστοιχα, μπορεί να παραλληλοποιηθεί το for loop που βρίσκει τα νέα κέντρα των cluster, αντικαθιστώντας τα παλιά.

Έτσι, χρησιμοποιούμε το parallel for του OpenMP, ορίζοντας κατάλληλα το scheduling policy (static) καθώς και σε κάθε loop της μεταβλητές που θα είναι shared σε όλα τα threads και αυτές που θα είναι private.

Σε αυτό το σημείο συνειδητοποιούμε ότι ο κώδικας παρουσιάζει ένα πρόβλημα: διαφορετικά threads ενδέχεται να γράφουν στις ίδιες θέσεις μνήμης, είτε στην μεταβλητή delta είτε σε θέσεις των πινάκων newClusterSize και newCluster, καθώς διαφορετικά αντικείμενα μπορεί να αντιστοιχίζονται στο ίδιο cluster. Αυτό δημιουργεί τον κίνδυνο εμφάνισης race conditions, τον οποίο μπορούμε να αντιμετωπίσουμε με την εισαγωγή του **#pragma omp atomic** πριν από τις εγγραφές αυτές.

Εδώ να σημειωθεί πως το **omp atomic** είναι πολύ πιο αποδοτικό από το **omp critical**, επειδή στις περισσότερες υλοποιήσεις χρησιμοποιεί hardware atomic instructions της ISA και μόνο αν δεν είναι διαθέσιμες, χρησιμοποιεί software lock. Το critical, από την άλλη, χρησιμοποιεί πάντα runtime lock και άρα είναι βαρύτερο.

Ο τελικός κώδικας είναι ο παρακάτω:

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

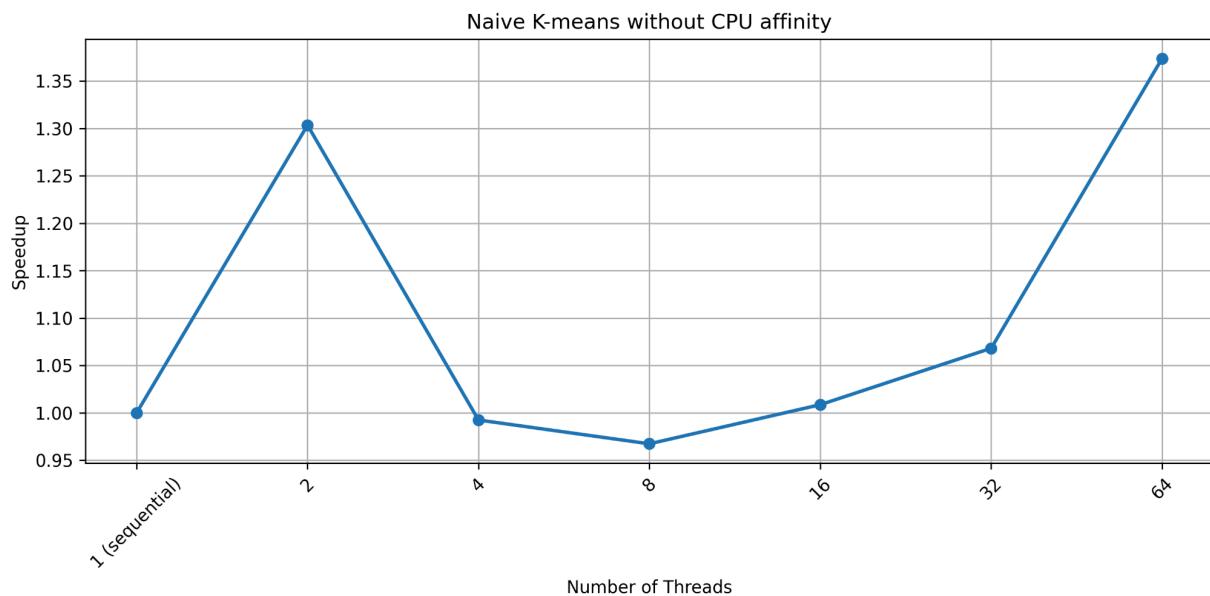
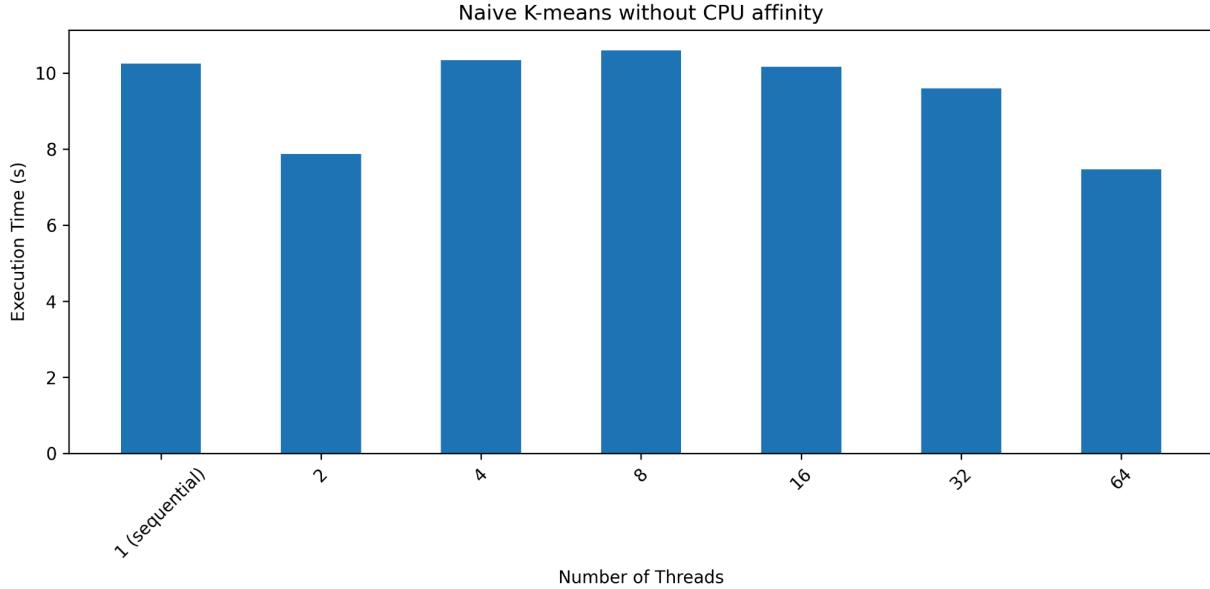
```

/*
 * 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];
}
#pragma omp parallel for schedule(static) private(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];
        }
    }
    // Get fraction of objects whose membership changed during this
loop. This is used as a convergence criterion.
    delta /= numObjs;
    loop++;
    printf("\r\tcompleted loop %d", loop);
    fflush(stdout);
} while (delta > threshold && loop < loop_threshold);

```

Στην συνέχεια θα πάμε να τρέξουμε τον αλγόριθμο μας στο μηχάνημα sandman, για configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, και για αριθμό threads = {1,2,4,8,16,32,64}. Ελέγχουμε προφανώς πως οι παραλληλοποιημένες εκτελέσεις δίνουν το ίδιο αποτέλεσμα με την σειριακή εκτέλεση.

Τα διαγράμματα χρόνου εκτέλεσης και speedup είναι τα παρακάτω:



Βλέπουμε δηλαδή ότι το speedup είναι πολύ κακό. Έχουμε sequential χρόνο 10.25 seconds. και με 64 threads (32 φυσικοί πυρήνες) καταφέρνουμε να ρίξουμε τον χρόνο μόνο στο 7.46 seconds. Ο λόγος που συμβαίνει αυτό είναι ότι το overhead που εισάγει ο συγχρονισμός (τα omp atomic) είναι τόσο μεγάλος, που ακυρώνει σε μεγάλο βαθμό τα οφέλη της παραλληλοποίησης - για 4 και 8 threads καταλήγουμε να έχουμε slowdown!!

Στο επομένω ζητούμενο θα δούμε πως να χρησιμοποιήσουμε το reduction για να υλοποιήσουμε μία αποδοτική παραλληλοποίηση.

2.

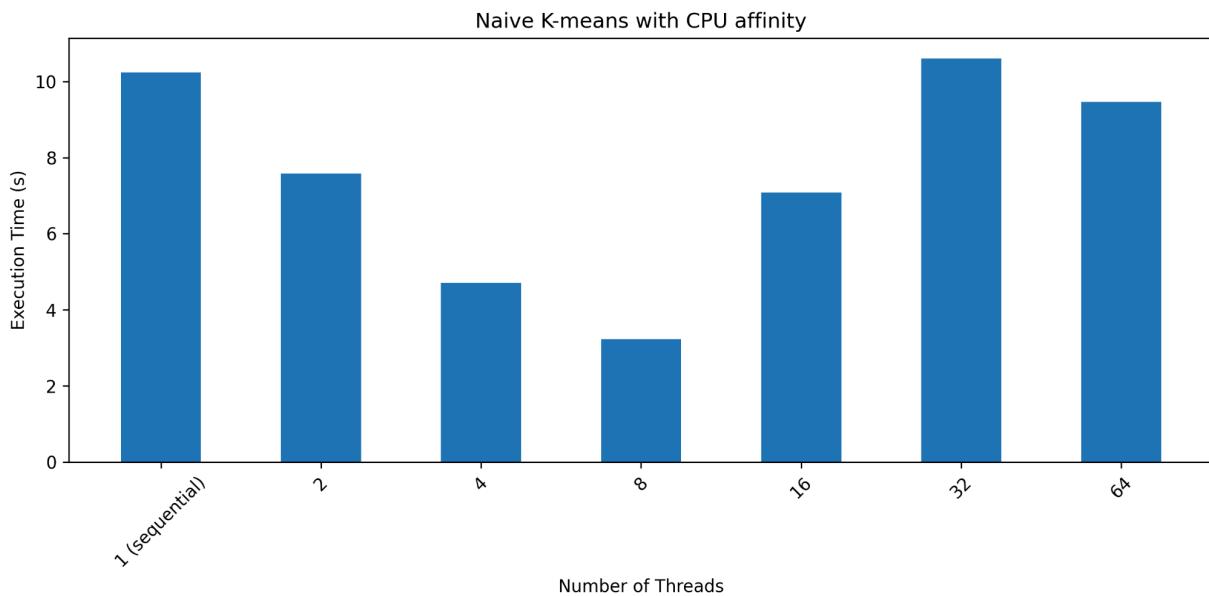
Το GOMP_CPU_AFFINITY είναι μια μεταβλητή περιβάλλοντος της υλοποίησης OpenMP της, η οποία καθορίζει σε ποιά cores επιτρέπεται να δεθούν τα threads που δημιουργούνται (επιτρέποντας το thread binding). Αυτό μπορεί να βελτιώσει την απόδοση αυξάνοντας την τοπικότητα των δεδομένων στις cache.

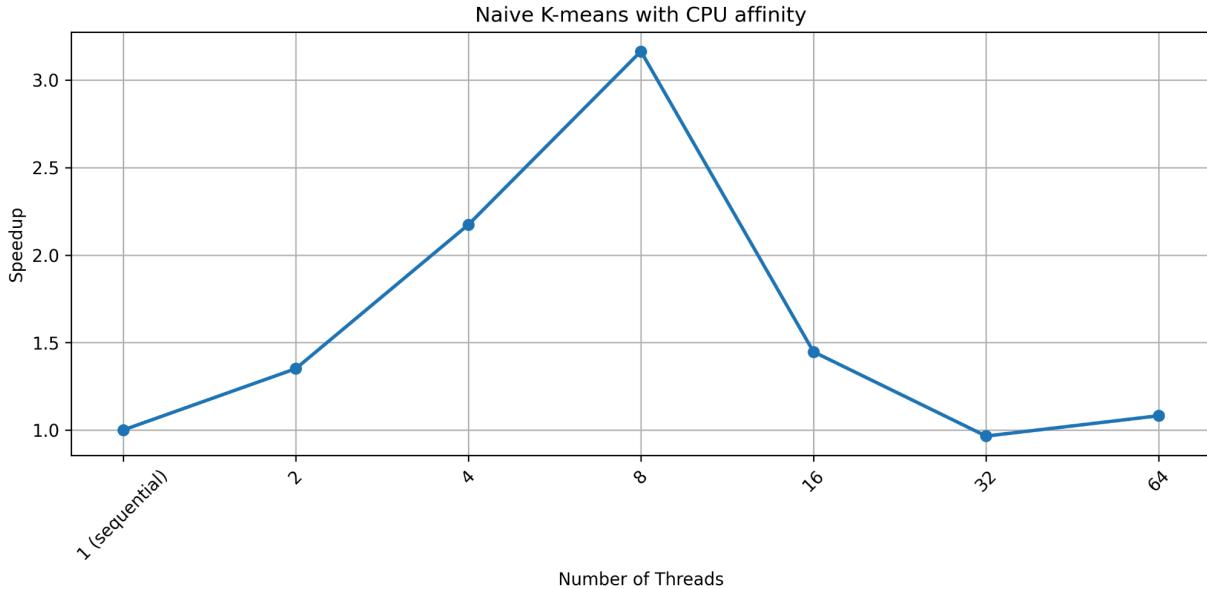
Έτσι πριν επαναλάβουμε τις μετρήσεις, αφού κάνουμε export των αριθμών των threads που θέλουμε, θα προσθέσουμε και την παρακάτω γραμμή για να αποτρέψουμε το thread migration.

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

Δηλαδή για παράδειγμα για 32 threads θα κάνουμε export 0-31.

Τα διαγράμματα χρόνου εκτέλεσης και speedup είναι τα παρακάτω:





Βλέπουμε πως τα αποτελέσματα εδώ είναι εμφανώς καλύτερα από τα προηγούμενα (χωρίς το CPU Affinity). Μέχρι τα 8 threads έχουμε βελτίωση καθώς αν και συνεχίζουμε να έχουμε το overhead του συγχρονισμού, πλέον με το thread binding αξιοποιούμε καλύτερα τις cache του συστήματος. Μετά τα 8 threads όμως τα νήματα αυξάνονται τόσο που ο συγχρονισμός κάνει dominate την παραλληλοποίηση, και επιστρέφουμε σε χρόνους εκτέλεσης παρόμοιους (ή και χειρότερους) με αυτούς που είχαμε πριν χρησιμοποιήσουμε το CPU Affinity.

Ακόμα, χαλάει την επίδοση και το γεγονός ότι έχουμε αγνοήσει τα NUMA χαρακτηριστικά του συστήματος. Συγκεκριμένα το allocation αλλά και η αρχικοποίηση των πινάκων newClusters και newClusterSize γίνονται στο σειριακό κομμάτι του κώδικα, και άρα από το master thread. Έτσι οι πίνακες αυτοί βρίσκονται στην τοπική μνήμη του CPU socket που ανήκει το master thread και ο χρόνος προσπέλασης σε αυτήν των threads των άλλων sockets είναι μεγάλος. Περισσότερα όμως για τα NUMA χαρακτηριστικά θα τα δούμε στο επόμενο ζητούμενο.

Τέλος, να σημειωθεί πως είναι αναμενόμενο να μην έχουμε μεγάλη βελτίωση από τα 32 στα 64 threads καθώς στην μετάβαση αυτή οι φυσικοί πυρήνες δεν αυξάνονται αλλά πάμε από τους 32 φυσικούς πυρήνες με 32 threads, σε πάλι 32 φυσικούς πυρήνες, αλλά με 64 threads (hyper-threading).

Zητούμενο 2 - Copied Clusters & Reduce

1.

Έχοντας δει πως το overhead του συγχρονισμού είναι τόσο μεγάλο που ακυρώνει στην πράξη τα οφέλη της παραλληλοποίησης, θα προσπαθήσουμε να τον αποφύγουμε.

Αυτό θα το κάνουμε διατηρώντας τοπικά αντίγραφα σε κάθε thread για τις κοινές μεταβλητές (delta) και τους κοινούς πίνακες (newClusters και newClusterSize) και στο τέλος θα φροντίσουμε να εφαρμόσουμε κατάλληλο reduction.

Για την μεταβλητή delta μπορούμε να χρησιμοποιήσουμε το reduction clause του OpenMP:

```
#pragma omp parallel reduction(+:delta)
```

To reduction για τους 2 πίνακες δεν μπορεί να γίνει τόσο απλά, αλλά πρέπει να γράψουμε λίγες γραμμές κώδικα. Πρέπει αρχικά να κάνουμε το allocation και αρχικοποίηση της μνήμη για το κάθε thread:

```
for (k=0; k<nthreads; k++)
{
    local_newClusterSize[k] = (typeof(*local_newClusterSize)) malloc(numClusters,
sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) malloc(numClusters *
numCoords, sizeof(**local_newClusters));
}
```

Και έπειτα, μέσα στην παράλληλη περιοχή, κάθε νήμα πρέπει σε κάθε επανάληψη του εξωτερικού βρόχου να μηδενίζει τους τοπικούς του πίνακες και να εξασφαλίζει ότι όλες οι εγγραφές γίνονται στη δική του τοπική μνήμη. Στο τέλος κάθε iteration, ένα μόνο νήμα (με **omp single**) πρέπει να προσπελαύνει διαδοχικά όλους τους τοπικούς πίνακες και να αθροίζει τα περιεχόμενά τους, πραγματοποιώντας έτσι το αντίστοιχο reduction.

Ο τελικός κώδικας είναι ο παρακάτω:

```
do {
    delta = 0.0;
    /*
     * TODO: Initialize local cluster data to zero (separate for each thread)
     */
    #pragma omp parallel private(i, index, j, k) reduction(+:delta)
    {

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

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

```

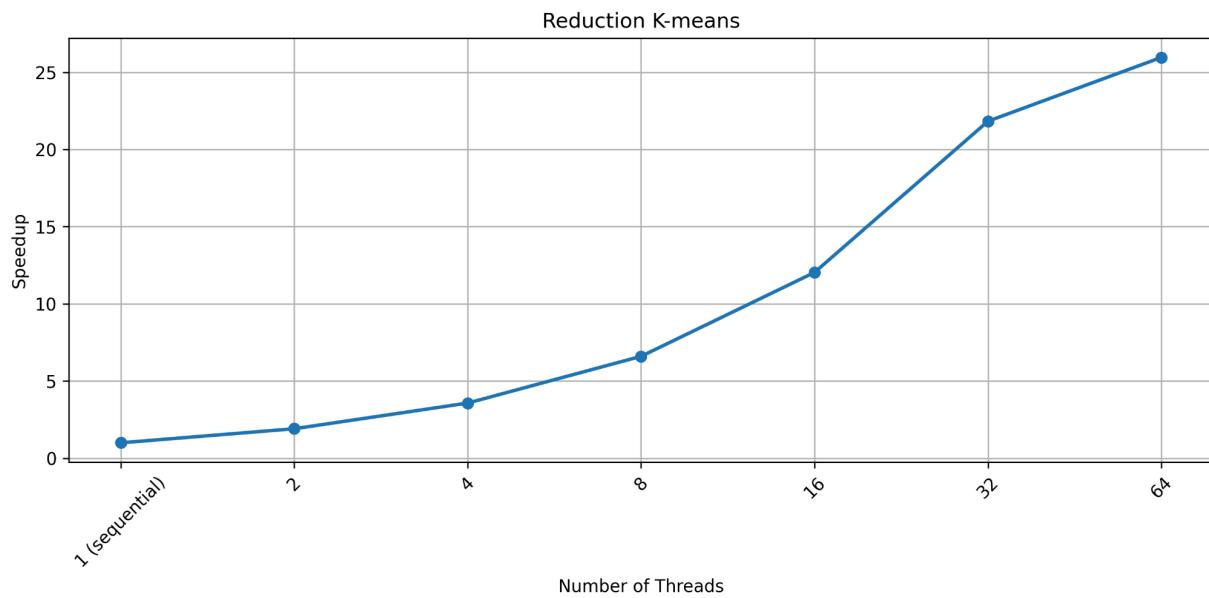
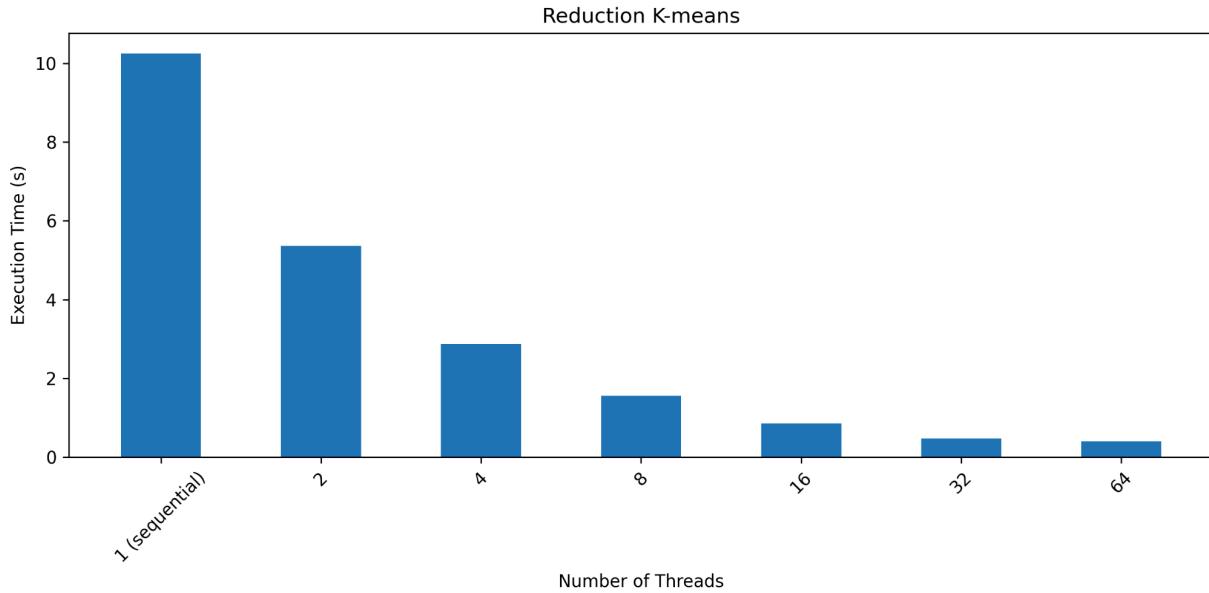
    // if membership changes, increase delta by 1
    if (membership[i] != index)
        delta += 1.0;

    // 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
     */
    local_newClusterSize[thread_id][index]++;
    for (j=0; j<numCoords; j++)
        local_newClusters[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
 */
#pragma omp single
{
    for (i=0; i<numClusters; i++) {
        newClusterSize[i] = 0;
        for(k=0; k<numCoords; k++)
            newClusters[i*numCoords + k] = 0.0;
        for (j=0; j<nthreads; j++) {
            newClusterSize[i] += local_newClusterSize[j][i];
            for (k=0; k<numCoords; k++)
                newClusters[i*numCoords + k] +=
local_newClusters[j][i*numCoords+k];
        }
    }
}
#pragma omp for schedule(static)
// 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];
        }
    }
}
// Get fraction of objects whose membership changed during this loop. This is used as a
convergence criterion.
    delta /= numObjs;
    loop++;
    printf("\r\tcompleted loop %d", loop);
    fflush(stdout);
} while (delta > threshold && loop < loop_threshold);

```

Εκτελούμε λοιπόν την παραπάνω παραληλοποιημένη εκδοχή του αλγορίθμου, χρησιμοποιώντας τα ίδια configurations όπως και πριν. Τα αποτελέσματα συνοψίζονται στα παρακάτω διαγράμματα χρόνου εκτέλεσης και speedup:

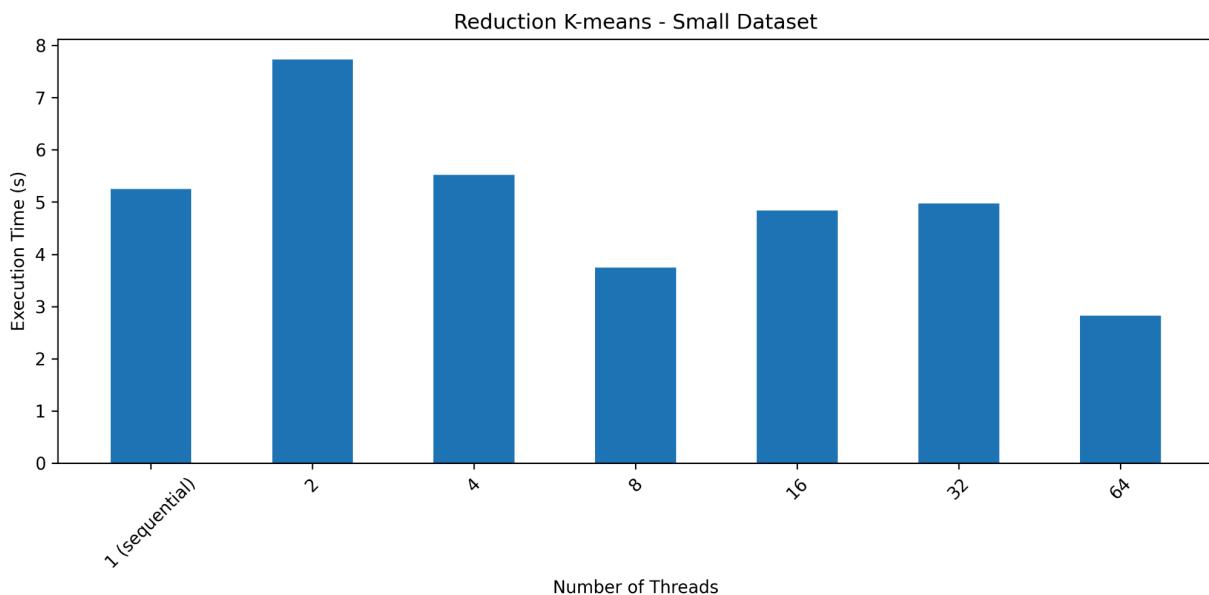


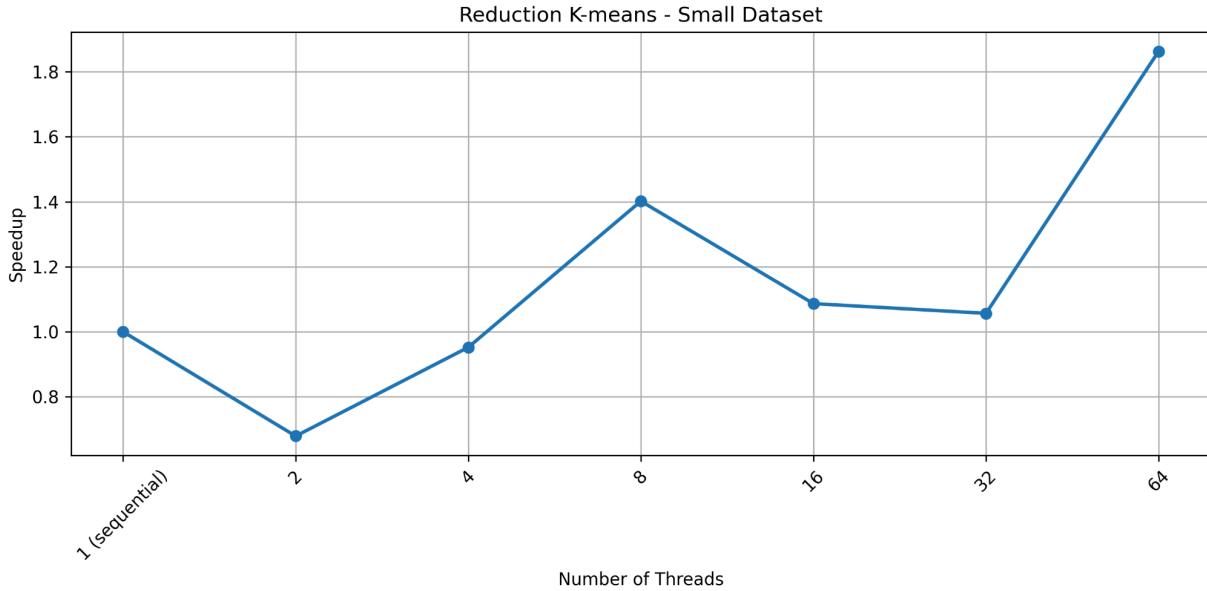
Πλέον εδώ βλέπουμε πως το πρόγραμμα μας κλιμακώνει αρκετά καλά, αν και προφανώς δεν έχουμε perfect speedup. Έχουμε μείωση του χρόνου σε κάθε αύξηση των threads, και καταφέραμε να μειώσουμε τον χρόνο εκτέλεσης από τα 10.2548 seconds (1 thread) στα 0.3949 seconds (64 threads). Έχουμε δηλαδή μέγιστο speedup περίπου ίσο με 25.96, και καταλαβαίνουμε στην πράξη πόσο επιβλαβείς για την επίδοση ήταν το overhead του συγχρονισμού.

2.

Εδώ θα επαναλάβουμε τις μετρήσεις, αλλά για νέο configuration όπου το κάθε object έχει μόνο 1 συντεταγμένη (σημεία στον x άξονα), και τα επιθυμητή clusters είναι 4 (<{Size, Coords, Clusters, Loops} = {256, 1, 4, 10}).

Τα διαγράμματα χρόνου εκτέλεσης και speedup είναι τα παρακάτω:





Εδώ παρατηρούμε πως το πρόγραμμα μας δεν κλιμακώνει καλά, και έχει ως και slowdown στα 2 threads. Ο λόγος που αυτό το μικρότερο configuration έχει πολύ χαμηλότερη επίδοση από το μεγαλύτερο στον ίδιο κώδικα μπορεί να αναζητηθεί σε 2 πράγματα: στην first touch πολιτική του Linux, και στα φαινόμενα false sharing.

First Touch:

Στο Linux το allocation της μνήμης με malloc δεν γίνεται την ώρα που τρέχει η malloc, αλλά μόνο όταν προσπελάζεται πρώτη φορά μια από τις ζητούμενες σελίδες μνήμης. Έτσι σε ένα NUMA σύστημα - σαν το sandman - η φυσική μνήμη δεσμεύεται στο NUMA node της CPU που πρώτη προσπελάζει μια σελίδα μνήμης.

Στον παραπάνω κώδικα η δέσμευση μνήμης γίνεται έξω από το βασικό loop του k-means, έξω από την παράλληλη περιοχή, και με calloc, συνεπώς όλες οι σελίδες της allocated μνήμης προσπελάζονται αμέσως (λόγω του calloc) και δεσμεύονται στο NUMA node του CPU socket στο οποίο ανήκει το master thread. Και έτσι τα threads που βρίσκονται στα άλλα 3 CPU sockets του μηχανήματος έχουν μεγάλο χρόνο προσπέλασης στην private τους μνήμη.

False Sharing:

Το άλλο πρόβλημα που εμφανίζεται εδώ, είναι το φαινόμενο του false sharing. Αν και έχουμε υλοποιήσει τους πίνακες newClusterSize και newClusters να είναι ξεχωριστοί για κάθε thread, το μικρό μέγεθος του κάθε object (1 συντεταγμένη) καθιστά τον ξεχωριστό χώρο του κάθε thread τόσο μικρό, που είναι πιθανό να «συνωστιστούν» διαφορετικοί private per-thread buffers σε ένα μοναδικό cache line. Συγκεκριμένα, ο χώρος που δεσμεύει κάθε thread για τον πίνακα newClusters είναι numClusters*numCoords*sizeof(double)=32 bytes, ενώ για τον πίνακα newClusters είναι numClusters*sizeof(int) = 16 bytes.

Συνεπώς, αν και έχουμε χωρίσει λογικά τα δεδομένα και το κάθε thread γράφει σε διαφορετικές θέσεις μνήμης, το γεγονός ότι αυτές βρίσκονται στην ίδια cache line σημαίνει ότι οι εγγραφές σε αυτές προκαλούν cache coherence invalidations. Δηλαδή, γράφοντας ένα thread μια τιμή, κάνει invalidate όλες τις άλλες μεταβλητές άλλων threads που βρίσκονται στο ίδιο cache line, αναγκάζοντας τα να ξανα-προσπελάζουν την μνήμη συχνά, καταστρέφοντας έτσι την επίδοση. Αυτό το “ping pong” των έγκυρων δεδομένων από cache σε cache, είναι το φαινόμενο false sharing, το οποίο το έχουμε δει και πιο αναλυτικά στο θέμα του MESI στο μάθημα των Προηγμένων Θεμάτων Αρχιτεκτονικής.

Ο λόγος που στο προηγούμενο configuration (16 συντεταγμένες, 32 clusters) δεν είχαμε κακή κλιμάκωση είναι πως οι πίνακες για το κάθε thread ήταν τόσο μεγάλοι που κατά βάση η μνήμη του κάθε thread δεν βρισκόταν σε κοινά cache lines με τις μνήμες άλλων thread, και συνεπώς δεν παρατηρούνταν μείωση επίδοσης λόγω false sharing.

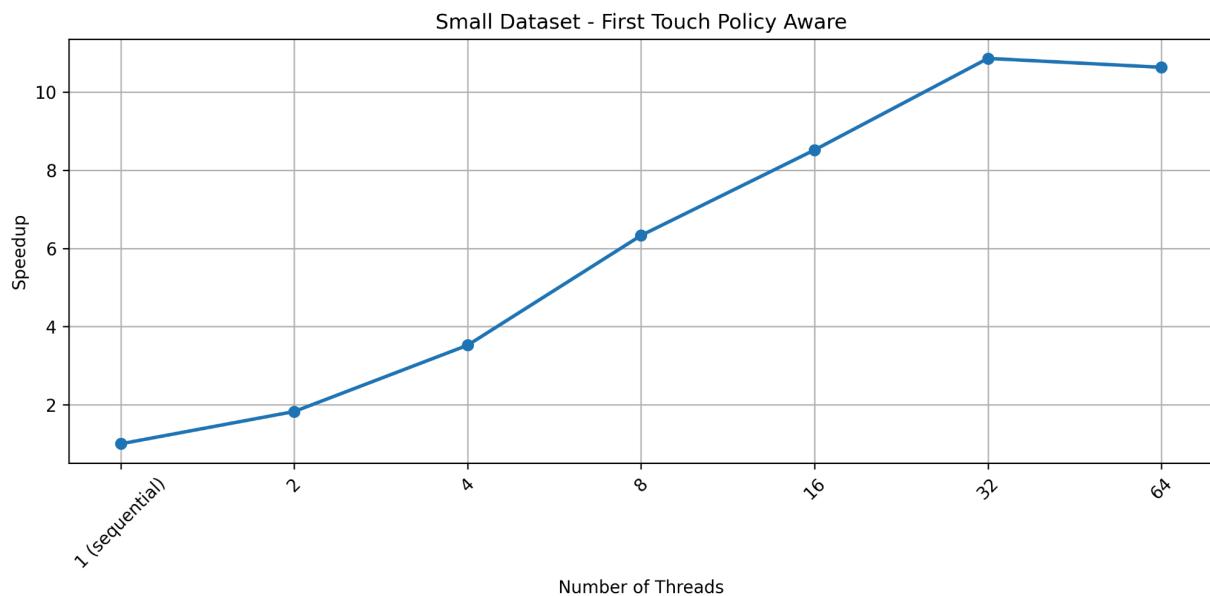
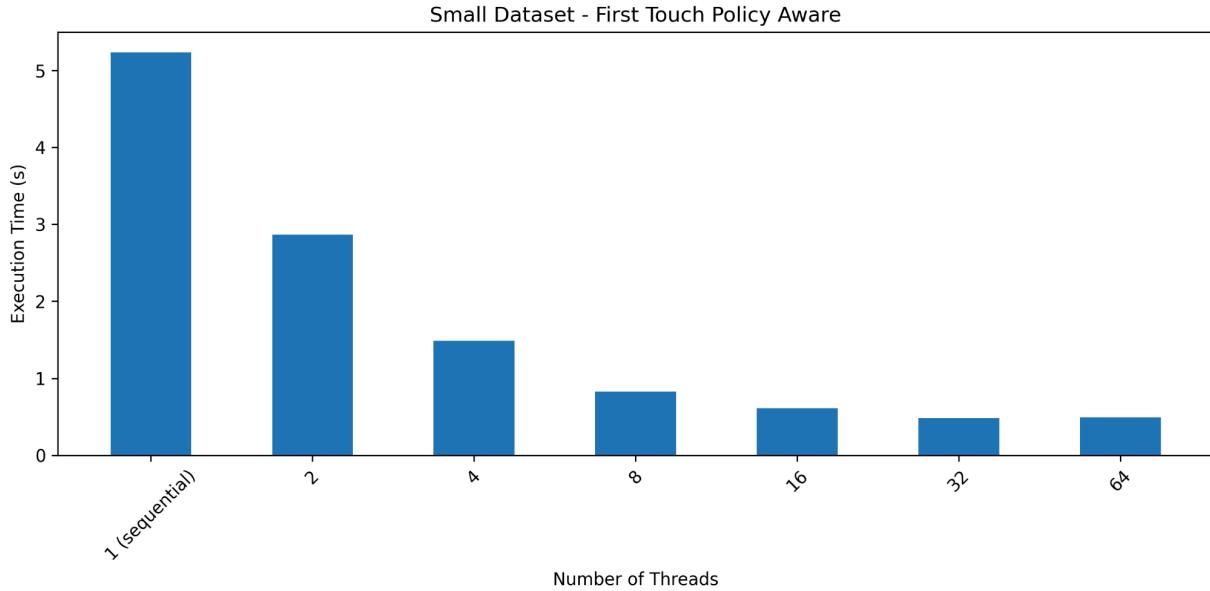
Λύση:

Για να λύσουμε τα παραπάνω θέμα παραλληλοποιήσουμε και το allocation, ώστε το first touch της κάθε σελίδας να γίνεται από το thread που θα χρησιμοποιήσει εν τέλει την μνήμη αυτή. Ο κώδικας με την προσθήκη του OpenMP directive είναι ο παρακάτω:

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

Βέβαια η παραπάνω λύση δεν λύνει απολύτως τα φαινόμενα false sharing - αυτό θα γινόταν πιο αποδοτικά με κάποιου είδους padding. Παρόλα αυτά, τα μετριάζει καθώς μειώνει και το κόστος της επαναπροσπέλασης της μνήμης όταν έχουμε cache coherence invalidations.

Παίρνουμε λοιπόν με τον νέο κώδικα μετρήσεις και ο καλύτερος χρόνος που καταφέρνουμε είναι **0.4818 seconds**. Τα διαγράμματα χρόνου εκτέλεσης και speedup είναι τα παρακάτω:



3.

Στο προηγούμενο ερώτημα εστιάσαμε στη βελτίωση της επίδοσης λαμβάνοντας υπόψη το που γίνεται το allocation της private μνήμης κάθε thread στο NUMA σύστημα που χρησιμοποιούμε. Ωστόσο, τα NUMA χαρακτηριστικά επηρεάζουν και τα ίδια τα δεδομένα. Καθώς σε κάθε επανάληψη του κύριου loop κάθε thread πρέπει να διαβάζει τις συντεταγμένες ενός object, είναι σημαντικό το dataset να είναι κατανεμημένο ισομερώς στα NUMA memory nodes και όχι συγκεντρωμένο στη μνήμη ενός μόνο CPU socket.

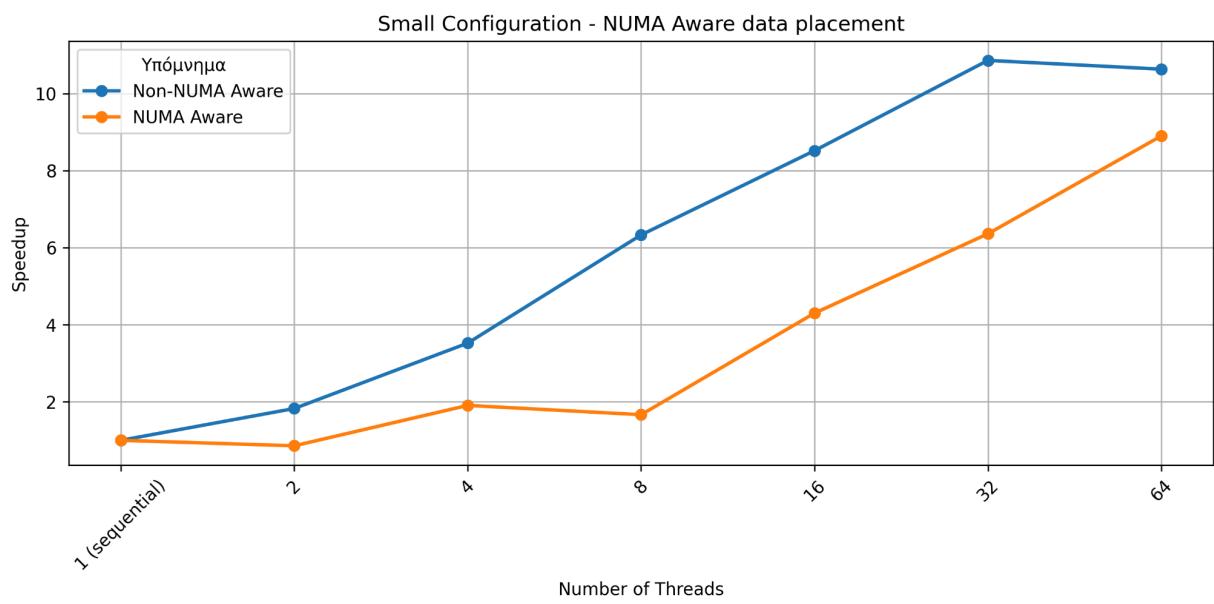
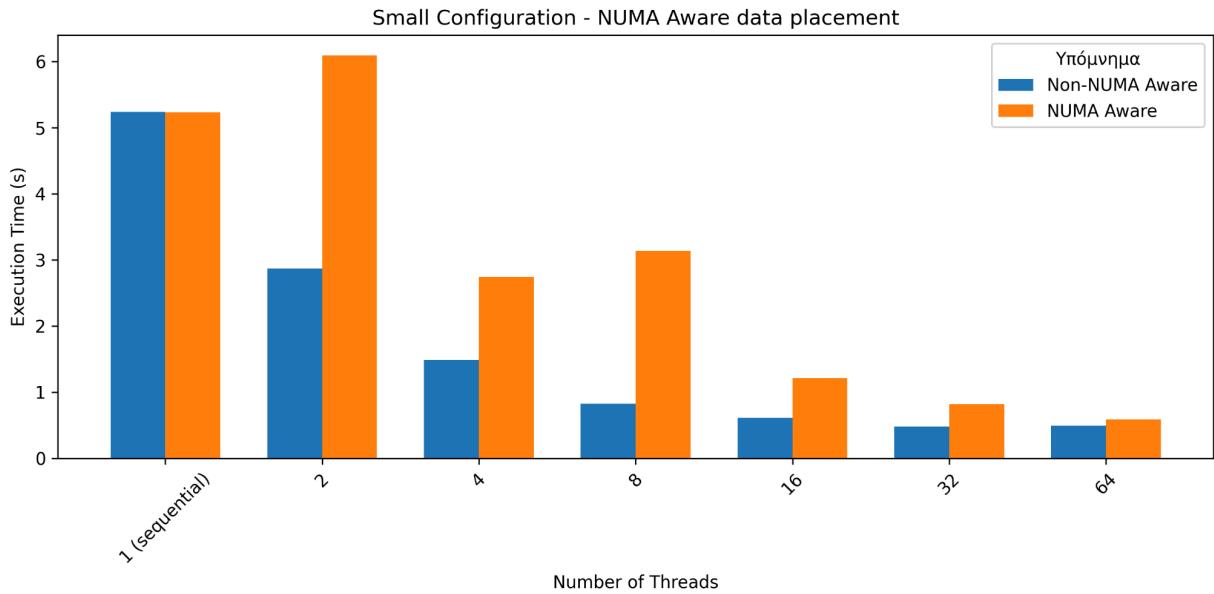
Για να το πετύχουμε αυτό, τροποποιούμε το αρχείο file_io.c, όπου γίνεται η αρχικοποίηση του dataset. Εκεί η μνήμη δεσμεύεται με malloc, και το first-touch πραγματοποιείται μέσα σε ένα loop που αρχικοποιεί τις συντεταγμένες με τυχαίες τιμές. Παραλληλοποιώντας αυτό το loop, κατανέμουμε τις σελίδες μνήμης του dataset στα NUMA nodes, “κοντά” στα threads που θα τις προσπελάσουν αργότερα. Ο σχετικός κώδικας είναι ο εξής:

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

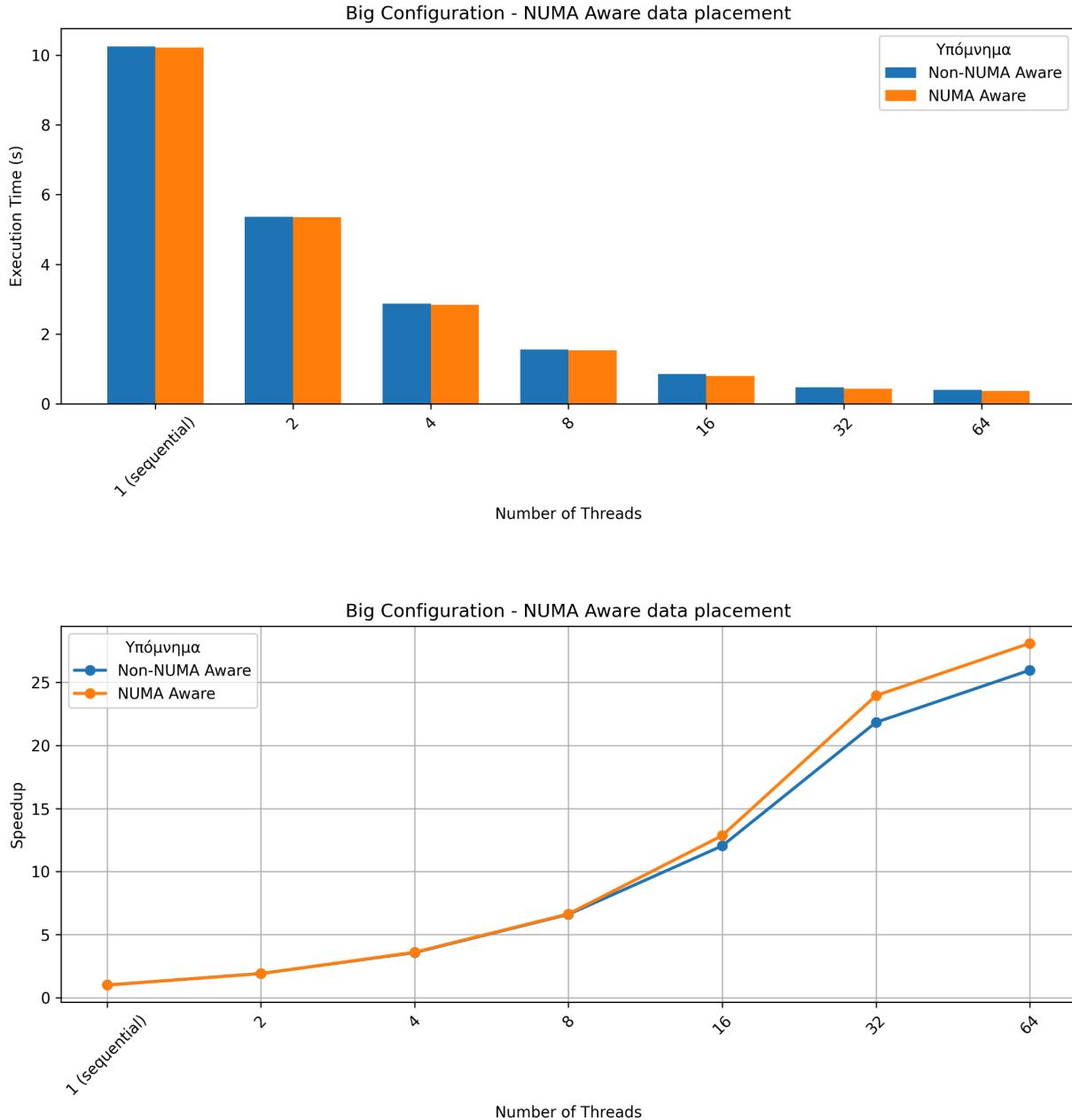
Να σημειωθεί εδώ, πως πρέπει εδώ να ενημερώσουμε και το Makefile ώστε να χρησιμοποιούμε τα OMPFLAGS και στη μεταγλώττιση του file_io.c, καθώς και να προσθέσουμε το flag -fopenmp στα LDFLAGS, καθώς πλέον πάνω από ένα object file χρησιμοποιεί OpenMP.

Παίρνουμε με τον ανανεωμένο κώδικα μετρήσεις και για το “μικρό” και για το “μεγάλο” configuration και τα διαγράμματα είναι τα παρακάτω:

“Μικρό” Configuration



“Μεγάλο” Configuration



Παρατηρούμε συνεπώς πως οι αλλαγές που κάναμε άφησαν απαράλλαχτη την επίδοση στο “μεγάλο” configuration (ελάχιστα καλύτερο speedup) ενώ χειροτέρεψαν τα πράγματα στο “μικρό” configuration.

Η μικρή αλλά υπαρκτή βελτίωση στο “μεγάλο” configuration είναι απολύτως αναμενόμενη. Με το NUMA-aware data placement, μεγάλο μέρος των προσβάσεων στα objects γίνεται πλέον από το NUMA memory node που αντιστοιχεί στο CPU socket του κάθε thread, μειώνοντας έτσι

το latency των memory accesses. Ωστόσο, το συνολικό κέρδος παραμένει περιορισμένο, διότι το k-means σε αυτή τη ρύθμιση είναι μόνο μερικώς memory-bound: σημαντικό τμήμα του συνολικού χρόνου δαπανάται σε υπολογισμούς (find nearest cluster) και σε προσπελάσεις/εγγραφές στους πίνακες newClusters και newClusterSize. Αυτά τα τμήματα δεν επηρεάζονται από το NUMA-aware placement, με αποτέλεσμα η συνολική βελτίωση να είναι σχετικά μικρή.

Για το “μικρό” configuration από την άλλη, το dataset είναι τόσο μικρό ώστε χωράει πρακτικά ολόκληρο στην cache, συνεπώς, το NUMA-aware allocation του δεν προσφέρει ουσιαστικά πλεονεκτήματα. Πριν την αλλαγή, κάθε thread χρειαζόταν να προσπελάσει μόνο μία φορά ένα NUMA memory node (ενδεχομένως “μακριά” μνήμη, δηλαδή μνήμη που αντιστοιχεί σε διαφορετικό CPU socket) για να φέρει, μέσα από λίγες προσπελάσεις (καθώς το dataset χωράει σε ελάχιστες cache lines), ολόκληρο τον πίνακα objects στην cache. Από εκεί και πέρα, ο πίνακας παρέμενε στην τοπική cache μέχρι της εκτέλεσης του αλγορίθμου.

Με το NUMA-aware data placement, ο πίνακας objects κατανέμεται πλέον σε ΟΛΑ τα NUMA memory nodes. Έτσι, κάθε thread χρειάζεται να προσπελάσει δεδομένα που βρίσκονται σε ΟΛΑ τα NUMA nodes προκειμένου να φέρει ολόκληρο το dataset στην cache του. Παρότι ένα από αυτά τα nodes είναι “κοντινό”, στην πράξη αντί για λίγες μόνο πιθανώς απομακρυσμένες προσβάσεις, το thread καταλήγει να πραγματοποιεί πολλαπλές απομακρυσμένες προσπελάσεις σε τρία διαφορετικά NUMA nodes (που αντιστοιχούν στα 3 άλλα CPU sockets).

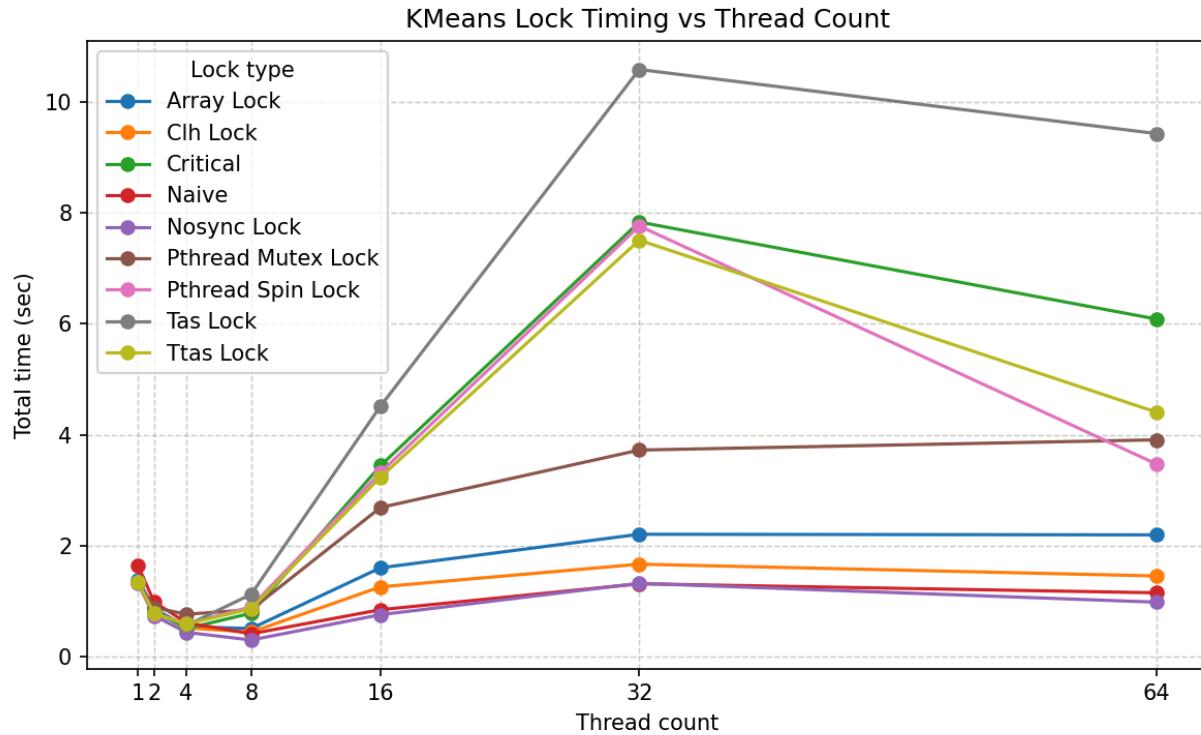
Γενικά, το bottleneck στο “μικρό” configuration οφείλεται κυρίως στα φαινόμενα false sharing, τα οποία εξακολουθούν βέβαια να υπάρχουν. Η τοποθέτηση της private μνήμης κάθε thread “κοντά” του μειώνει μεν το κόστος των cache coherence invalidations, όπως είδαμε και παραπάνω, αλλά δεν εξαλείφει το πρόβλημα. Επιπλέον, σημαντικό ρόλο παίζει και το latency της μνήμης για την αρχική προσπέλαση του dataset, έστω κι αν αυτή γίνεται ουσιαστικά μόνο μία φορά ανά thread.

Ζητούμενο 3 - Αμοιβαίος Αποκλεισμός - Κλειδώματα

Όπως είδαμε παραπάνω, ο αλγόριθμος k-means περιέχει σημεία στα οποία απαιτείται συγχρονισμός μεταξύ των threads. Στο συγκεκριμένο ζητούμενο καλούμαστε να αξιολογήσουμε διάφορους μηχανισμούς κλειδώματος που μπορούν να χρησιμοποιηθούν για την εξασφάλιση αμοιβαίου αποκλεισμού κατά την ενημέρωση του πίνακα newClusters στη “shared cluster” υλοποίηση του αλγορίθμου.

Για τον σκοπό αυτό θα εκτελέσουμε τον αλγόριθμο στο μηχάνημα sandman με τις παραμέτρους {Size, Coords, Clusters, Loops} = {32, 16, 32, 10} και για πλήθος νημάτων threads = {1, 2, 4, 8, 16, 32, 64}, και για 8 διαφορετικά είδη locks (συμπεριλαμβανομένης και της περίπτωσης χωρίς καθόλου συγχρονισμό, η οποία λειτουργεί ως κατώτατο χρονικό όριο για τη σύγκριση των υπολογίτων).

Τα αποτελέσματα παρουσιάζονται στο παρακάτω διάγραμμα:



Σχολιασμός Αποτελεσμάτων:

nosync_lock:

Στην υλοποίηση nosync δεν πραγματοποιείται κανένας μηχανισμός συγχρονισμού, και επομένως δεν εξασφαλίζεται αμοιβαίος αποκλεισμός και άρα ούτε ορθότητα. Παρ' όλα αυτά, τη συμπεριλαμβάνουμε στις μετρήσεις ως κατώτατο χρονικό όριο, δηλαδή ως το ανώτατο δυνατό όριο επίδοσης. Παρότι το nosync εμφανίζεται πράγματι ταχύτερο από τις υπόλοιπες υλοποιήσεις που προσφέρουν σωστό συγχρονισμό, παρατηρούμε ότι και αυτό δεν κλιμακώνει αρκετά καθώς αυξάνεται ο αριθμός των νημάτων. Αυτό οφείλεται στο ότι οι προσβάσεις στη μνήμη (αναγνώσεις και εγγραφές) εξακολουθούν να είναι ατομικές σε επίπεδο hardware. Δηλαδή, δεν θα διαβάσουμε για παράδειγμα ποτέ 32bits από μία παλιά τιμή και 32bits από μία νέα. Έτσι οι εγγραφές και οι αναγνώσεις δημιουργούν έντονη συμφόρηση στον δίσυλο μνήμης, με αποτέλεσμα τη μειωμένη κλιμάκωση.

naive:

Ο κώδικας εδώ χρησιμοποιεί atomic hardware instructions ως μηχανισμούς συγχρονισμού. Προφανώς, αυτό το είδος κλειδώματος είναι γρηγορότερο από οποιοδήποτε software-based κλειδωμα, όπως είναι τα υπόλοιπα του πειράματος.

pthread_mutex_lock:

Το pthread_mutex_lock είναι ένας μηχανισμός αμοιβαίου αποκλεισμού που παρέχεται από τα POSIX threads. Όταν ένα thread επιχειρήσει να αποκτήσει ένα mutex το οποίο είναι ήδη κατειλημμένο, μπλοκάρει και τίθεται σε κατάσταση αναμονής έως ότου το mutex απελευθερωθεί. Το πλεονέκτημα αυτού του μηχανισμού είναι ότι αποφεύγεται το busy-waiting, καθώς η αναμονή δεν καταναλώνει CPU κύκλους, ενώ το μειονέκτημα του είναι πως απαιτείται context switch που εισάγει επιπλέον καθυστερήσεις. Για τον λόγο αυτό, τα mutex locks είναι καταλληλότερα όταν η αναμονή αναμένεται να είναι σχετικά μεγάλη.

pthread_spin_lock:

Το pthread_spin_lock είναι άλλο ένα κλείδωμα που προσφέρουν τα POSIX threads, που βασίζεται όμως σε busy-waiting μηχανισμό. Ένα thread που αποτυγχάνει να αποκτήσει το lock συνεχίζει να ξαναπροσπαθεί μέχρι να το αποκτήσει. Αυτός ο μηχανισμός μπορεί να είναι ταχύτερος όταν η διάρκεια κατοχής του lock είναι εξαιρετικά μικρή, αλλά επιβαρύνει σημαντικά την CPU.

tas_lock:

Το tas lock (test-and-set lock) αποτελεί ένα είδος spinlock όπου σε κάθε επανάληψη, κάθε νήμα εκτελεί μια ατομική εντολή εγγραφής (atomic-write) στον δίσκο μνήμης, διεκδικώντας επιθετικά το κρίσιμο τμήμα. Παρατηρώντας το διάγραμμα, η επίδοση του tas_lock (γκρι γραμμή) είναι η χειρότερη όλων καθώς αυξάνονται τα νήματα, παρουσιάζοντας εκθετική αύξηση του χρόνου εκτέλεσης. Αυτό συμβαίνει διότι η συνεχιζόμενη εκτέλεση ατομικών εγγραφών, ακόμη και όταν το lock δεν έχει απελευθερωθεί, προκαλεί τεράστια συμφόρηση στον δίσκο συστήματος (bus contention) και συνεχή ακύρωση των γραμμών της κρυφής μνήμης (cache-ping-pong/false-sharing) όλων των επεξεργαστών, καθιστώντας το πρακτικά άχρηστο σε συνθήκες υψηλού ανταγωνισμού.

ttas_lock:

Το ttas_lock (test-and-test-and-set) αποτελεί μια βελτιστοποίηση του απλού TAS, όπου το νήμα διαβάζει πρώτα την τιμή του lock και επιχειρεί ατομική εγγραφή μόνο αν ανιχνεύσει ότι αυτό είναι ελεύθερο. Στο γράφημα παρατηρούμε ότι αυτή η τεχνική προσφέρει σαφώς καλύτερη κλιμάκωση σε σχέση με το απλό TAS, καθώς μειώνει δραστικά την κίνηση στον δίσκο όσο τα νήματα βρίσκονται σε αναμονή. Ωστόσο, εξακολουθεί να υστερεί σημαντικά σε σχέση με τα queue-based locks (όπως το Array και το CLH) σε μεγάλο αριθμό νημάτων. Αυτό οφείλεται στο φαινόμενο της "αγέλης" (thundering herd problem): τη στιγμή που το lock απελευθερώνεται, όλα

τα νήματα που περίμεναν αντιλαμβάνονται ταυτόχρονα την αλλαγή και επιτίθενται μαζικά με ατομικές εντολές, προκαλώντας στιγμιαία αλλά έντονη συμφόρηση.

array_lock:

Το array_lock ανήκει στην κατηγορία των queue-based locks, όπου κάθε νήμα καταλαμβάνει μια μοναδική θέση σε έναν κυκλικό πίνακα και περιμένει (spin-άρει) αποκλειστικά σε αυτή τη θέση μέχρι να έρθει η σειρά του. Τα αποτελέσματα στο διάγραμμα (μπλε γραμμή) είναι εντυπωσιακά, καθώς ο χρόνος εκτέλεσης παραμένει σχεδόν σταθερός ανεξάρτητα από τον αριθμό των νημάτων. Αυτό οφείλεται στο γεγονός ότι εξαλείφεται πλήρως ο ανταγωνισμός για την ίδια διεύθυνση μνήμης. Κάθε νήμα ελέγχει μια διαφορετική μεταβλητή (με την προϋπόθεση σωστού padding για αποφυγή του false sharing), επιτυγχάνοντας έτσι Deterministic Fairness (FIFO) και ελαχιστοποιώντας την κίνηση στον δίσυλο μνήμης.

clh_lock:

Το clh_lock είναι επίσης ένα queue-based lock, το οποίο όμως υλοποιεί μια νοητή ουρά (linked list logic) όπου κάθε νήμα περιστρέφεται ελέγχοντας την κατάσταση του προκατόχου του (predecessor). Όπως και το array_lock, έτσι και το clh επιδεικνύει σταθερότητα σε υψηλό αριθμό νημάτων. Το πλεονέκτημά του έγκειται στην αποδοτικότερη διαχείριση μνήμης σε σχέση με το array lock, καθώς δεν απαιτεί προκαθορισμένο μέγεθος πίνακα για το μέγιστο πλήθος νημάτων. Η στρατηγική του "local spinning" σε δεδομένα που βρίσκονται πιθανότατα ήδη στην cache του επεξεργαστή, το καθιστά μία από τις πιο αποδοτικές λύσεις για συστήματα shared memory με υψηλό contention.

critical:

Η ονομασία critical αναφέρεται πιθανότατα σε υψηλού επιπέδου δομές αμοιβαίου αποκλεισμού, όπως το **#pragma omp critical** του OpenMP. Παρόλο που προσφέρει ευκολία στον προγραμματισμό, το διάγραμμα αποκαλύπτει ότι η επίδοσή του είναι φτωχή καθώς αυξάνεται ο παραλληλισμός, ακολουθώντας παρόμοια αυξητική τάση με τα απλά spinlocks. Αυτό συμβαίνει διότι, εσωτερικά, τέτοιες δομές συχνά υλοποιούνται με γενικού σκοπού κλειδώματα που δεν είναι βελτιστοποιημένα για συγκεκριμένα μοτίβα πρόσβασης ή NUMA αρχιτεκτονικές, εισάγοντας σημαντικό overhead διαχείρισης και αποτυγχάνοντας να κλιμακώσουν γραμμικά σε περιβάλλοντα με έντονο ανταγωνισμό πόρων.

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Εισαγωγή

Στο συγκεκριμένο ζητούμενο, κληθήκαμε να παραλληλοποιήσουμε τον αλγόριθμο *Floyd-Warshall*. Ο αλγόριθμος Floyd–Warshall υπολογίζει τις συντομότερες διαδρομές μεταξύ όλων των ζευγών κόμβων σε έναν γράφο βαρών, χρησιμοποιώντας λογική δυναμικού

προγραμματισμού. Η “διασημότερη” υλοποίηση του αλγορίθμου χρησιμοποιεί τρεις φωλιασμένους βρόχους for. Στον πίνακα A αποθηκεύεται, αρχικά, ο πίνακας γειτνίασης (ή οι αρχικές αποστάσεις) του γράφου και στη συνέχεια, για κάθε k από 0 έως N-1, εξετάζεται ο κόμβος k ως πιθανός ενδιάμεσος κόμβος, για κάθε μονοπάτι από τον κόμβο i στον j. Συγκεκριμένα, εφαρμόζεται η σχέση:

$A[i][j] = \min(A[i][j], A[i][k] + A[k][j])$ για κάθε $i, j, k < N$. Ο πλήρης κώδικας είναι:

```
for (k=0; k<N; k++)
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i][j] = min(A[i][j], A[i][k]+A[k][j]);
```

Η παραπάνω υλοποίηση του Floyd-Warshall μπορεί να παραλληλοποιηθεί για σταθερό k (ο εξωτερικός βρόχος ΔΕΝ μπορεί να παραλληλοποιηθεί, καθώς κάθε επανάληψη k εξαρτάται από τα αποτελέσματα της προηγούμενης εκτέλεσης του βρόχου). Μία ιδέα είναι να χρησιμοποιήσουμε, όπως στο ζητούμενο με την παραλληλοποίηση του kmeans, parallel for. Μια πρόχειρη υλοποίηση είναι η ακόλουθη:

```
for (k = 0; k < N; k++) {
    #pragma omp parallel for private(j)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
}
```

Ωστόσο, η παραλληλοποίηση του επαναληπτικού αλγορίθμου δεν προτείνεται. Για μεγάλους πίνακες A, τα στοιχεία δεν χωράνε στην cache, οδηγώντας σε πολλαπλά cache misses λόγω κακού spatial locality. Συγκεκριμένα, με τον τρόπο που διατρέχουν οι βρόχοι τον πίνακα A, τα στοιχεία $A[i][k]$ και $A[k][j]$ θα προσπελαύνονται με σχεδόν random access τρόπο. Το overhead αυτό θα υπερκαλύπτει την επιτάχυνση που προσδίδει η παραλληλοποίηση, με αποτέλεσμα να υπάρχει μικρό scalability.

Γι' αυτό, εκμεταλλευόμαστε την αναδρομική “φύση” του αλγορίθμου, ώστε να εκτελέσουμε τις απαραίτητες πράξεις διατρέχοντας τον πίνακα A που αναπαριστά το γράφημα ανά τοπικά blocks.

Ανασκόπηση αναδρομικού ορισμού αλγορίθμου:

Ορίζουμε τον πίνακα αποστάσεων $D[k][i][j] =$ το μήκος του συντομότερου μονοπατιού από i στο j, στο οποίο επιτρέπονται να χρησιμοποιηθούν ως ενδιάμεσοι κόμβοι στοιχεία μόνο από το σύνολο {0,1,2,...,k}.

Ο αναδρομικός ορισμός του αλγορίθμου, λοιπόν, γράφεται:

$$D[k][i][j] = \min(D[k-1][i][j], D[k-1][i][k] + D[k-1][k][j])$$

δηλαδή, μεταξύ των κορυφών i, j :

- είτε το καλύτερο μονοπάτι **δεν χρησιμοποιεί** τον k ως ενδιάμεσο
- είτε το καλύτερο μονοπάτι **χρησιμοποιεί** τον k , οπότε σπάει σε δύο τμήματα: $i \rightarrow k$ και $k \rightarrow j$.

Με άλλα λόγια ο Floyd–Warshall είναι ένας αναδρομικά ορισμένος αλγόριθμος όπου το state $D[k][i][j]$ σημαίνει “η καλύτερη απόσταση από i σε j επιτρέποντας μόνο ενδιάμεσους κόμβους από το $\{0\dots k\}$ ”, και ο αναδρομικός τύπος συγκρίνει το μονοπάτι που δεν περνά από τον k με αυτό που περνά.

Η ιδέα του optimization πηγάζει από τη διαίρεση του state space των επιτρεπόμενων ενδιάμεσων κόμβων K . Το σύνολο $\{0\dots N-1\}$ χωρίζεται σε δύο ισομεγέθη υποσύνολα $K0$ και $K1$ και υπολογίζουμε πρώτα όλες τις αποστάσεις επιτρέποντας ως intermediates μόνο το $K1$, και στη συνέχεια επιτρέποντας intermediates το $K2$. Η δομή του Floyd–Warshall διατηρείται αυτούσια, αλλά η ενημέρωση $A[i][j] = \min(A[i][j], A[i][k] + A[k][j])$ εφαρμόζεται σε επίπεδο blocks, με αποτέλεσμα σημαντικά καλύτερη εκμετάλλευση της cache και περισσότερες ευκαιρίες για παραλληλισμό μέσω OpenMP tasks.

Παραθέτουμε το σχήμα που διαιρεί τον πίνακα A σε 4 blocks και το state space σε δύο υποσύνολα $K0$ και $K1$. Ο πίνακας μπορεί αναδρομικά να διαιρεθεί σε οποιοδήποτε αριθμό blocks = 2^k , όπου k ακέραιος μεγαλύτερος του 1.



Μπορούμε να σκεφτούμε τον παραπάνω διαχωρισμό ως εξής:

Διαιρούμε το αρχικό γράφημα σε δύο υπογραφήματα G_0 και G_1 . Το block A_{xy} αναπαριστά τις μεταβάσεις που γίνονται από κόμβους μέσα στο υπογράφημα G_x σε κόμβους στο υπογράφημα G_y . Προφανώς, αν $x=y$, αναφερόμαστε σε μεταβάσεις εντός του ίδιου υπογραφήματος.

Αφού θεωρήσαμε διαίρεση του state-space σε K_0 και K_1 , πρέπει να ελέγχουμε τις διακριτές περιπτώσεις που το ενδιάμεσο σημείο ενός μονοπατιού i,j είναι μέρος του G_0 (στο state K_0) και του G_1 (στο state K_1) αντίστοιχα.

Άρα για τα blocks $A_{00}, A_{01}, A_{10}, A_{11}$ ελέγχουμε αν τα σημεία του K_0 , σε περίπτωση που χρησιμοποιηθούν ως ενδιάμεσα, δίνουν βέλτιστο μονοπάτι. Αναδρομικά γράφουμε:

```
//A00,B00,C00
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);

//A01,B00,C01
FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);

//A10,B10,C00
FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);

//A11,B10,C01
FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
```

```
myN/2, bsize);
```

και όμοια για τα σημεία του state K1:

```
//A11,B11,C11
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);

//A10,B11,C10
FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol,
myN/2, bsize);

//A01,B01,C11
FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
myN/2, bsize);

//A00, B01, C10
FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
```

To base case της αναδρομικής συνάρτησης εκτελείται όταν έχουμε N μικρότερο του block size B που έχουμε θέσει. Τότε σημαίνει ότι η κλήση βρίσκεται μέσα στο ελάχιστου μεγέθους block που έχουμε ορίσει και εκεί εκτελούμε τον απλό επαναληπτικό αλγόριθμο Floyd-Warshall (με πολύ καλύτερο cache utilization για κατάλληλο B εφόσον το block είναι αρκούντως μικρό για να χωράει όλο στην cache).

Στη συνέχεια, παραλληλοποιήσαμε την εκτέλεση της αναδρομικής συνάρτησης του Floyd-Warshall, με χρήση των OpenMP tasks. Κάθε αναδρομική κλήση μπορεί να θεωρηθεί ένα ξεχωριστό task και εξετάσαμε ποια από αυτά μπορούν να εκτελεστούν ταυτόχρονα.

Κατ'αρχάς, όπως και στην υλοποίηση με for, ο αλγόριθμος δεν μπορεί να παραλληλοποιηθεί ως προς K, καθώς η κάθε κατάσταση του state space εξαρτάται από τον υπολογισμό των υπολοίπων καταστάσεων. Αναγκαστικά, θα πρέπει οι ομάδες των task που αφορούν την κάθε κατάσταση να εκτελεστούν σειριακά.

Από την άλλη, κάποια tasks μέσα στο κάθε state space μπορούν να παραλληλοποιηθούν. Προσέχοντας τις αναδρομικές κλήσεις και τα data dependencies μεταξύ τους, καταλήξαμε ότι είναι ορθό να παραλληλοποιηθούν οι αναζητήσεις του ενδιάμεσου σημείου της κάθε κατάστασης για αφετηρία και τερματισμό που ανήκουν σε διαφορετικά υπογραφήματα. Με άλλα λόγια, παραλληλοποιήσαμε τις αναδρομικές κλήσεις:

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

καθώς ούτε γράφουν στην ίδια περιοχή μνήμης ούτε διαβάζουν από περιοχή που γράφει η άλλη κλήση.

Ως pivot, πριν την εκτέλεση των συγκεκριμένων κλήσεων, θα εκτελεστεί η κλήση:

```
//A00,B00,C00
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
```

ώστε τα βέλτιστα μονοπάτια εντός του πρώτου υπογραφήματος να έχουν γίνει resolve πριν εξεταστούν μεταβάσεις στο άλλο υπογράφημα.

Μετά το πέρας της εκτέλεσης της παράλληλης περιοχής, εκτελούμε την κλήση:

```
//A11,B10,C01
FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
myN/2, bsize);
```

Η παραπάνω κλήση, αν και εξετάζει το ίδιο state space, δεν μπορεί να παραλληλοποιηθεί με τις υπόλοιπες καθώς χρειάζεται να διαβάσει τα δεδομένα σε περιοχές μνήμης που γράφουν οι δύο προηγούμενες διεργασίες.

Κατ' απόλυτη αντιστοιχία, υλοποιούνται και οι αναδρομικές κλήσεις του δεύτερου state space K1, απλά με pivot το resolution εντός του αντίστοιχου υπογραφήματος.

Καταληκτικά, ο τελικός **παραλληλοποιημένος κώδικας** με χρήση OpenMP tasks είναι:

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

    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
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
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);
}
}
}

```

Εδώ να σημειωθεί πως όταν προσθέτουμε το clause if(0) δεν δημιουργείται το task, (καθώς το 0 αντιστοιχεί σε false) αλλά αντίθετα το thread εκτελεί απευθείας τον κώδικα. Ο λόγος που δεν αφαιρούμε το OpenMP directive τελείως είναι στιλιστικός.

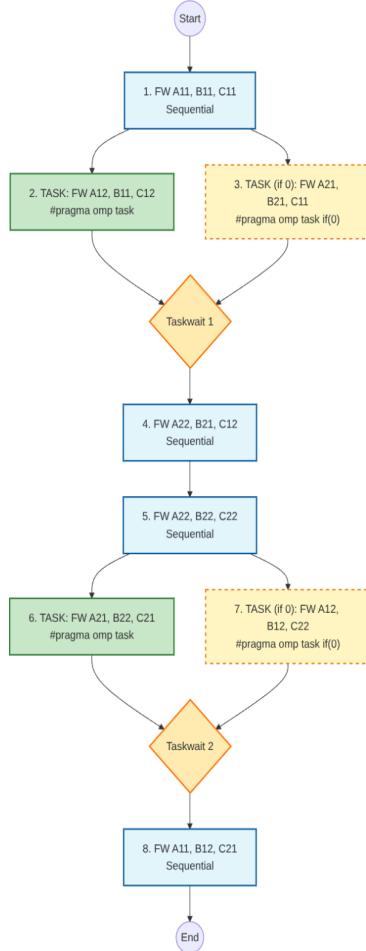
Και η κλήση της παραπάνω συνάρτησης από την main πρέπει να γίνει σε παράλληλη περιοχή (ώστε να δημιουργηθούν τα νήματα) αλλά από μόνο ένα thread (ώστε να κληθεί μία φορά η συνάρτηση):

```

#pragma omp parallel
#pragma omp single
FW_SR(A,0,0, A,0,0,A,0,0,N,B);

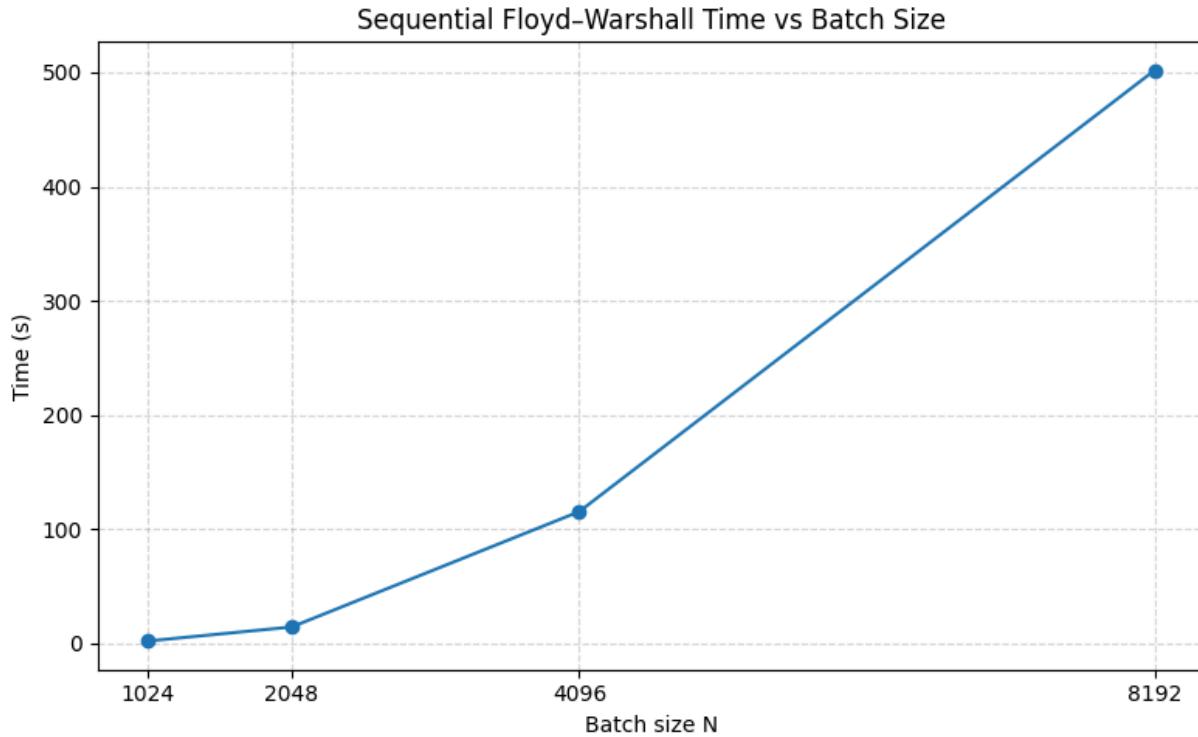
```

Και το **task graph** που προκύπτει, κατά την εκτέλεση:



Αποτελέσματα μετρήσεων:

Αρχικά, εκτελέσαμε τον δοθέντα σειριακό αλγορίθμου Floyd-Warshall στον υπολογιστή sandman του εργαστηρίου και λάβαμε τα ακόλουθα αποτελέσματα:

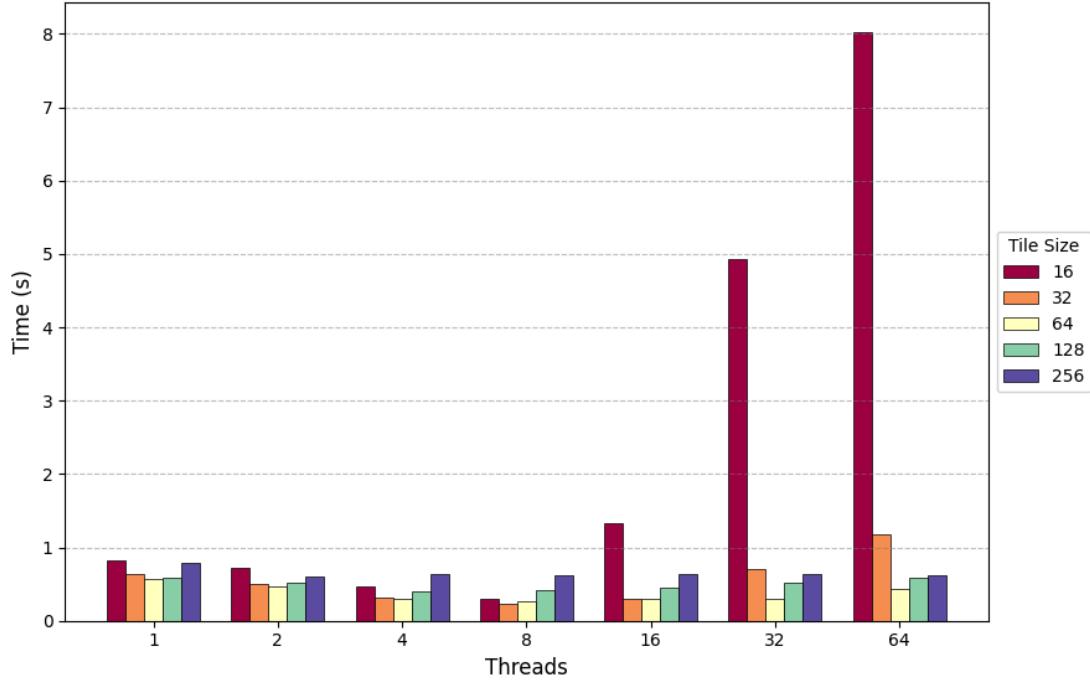


Στη συνέχεια, εκτελέσαμε και τον παραλληλοποιημένο κώδικα, για διάφορες τιμές του batch size N, base case block size B, σε διαφορετικά parallel thread configurations.

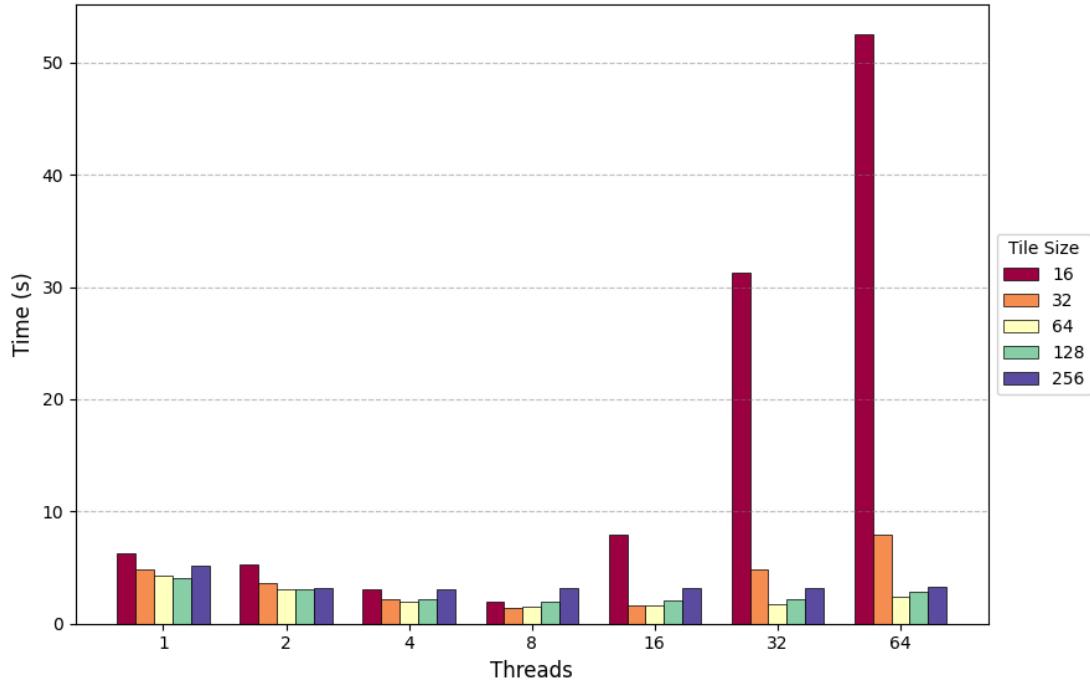
Αξιολογήσαμε την απόδοση του αλγορίθμου για μεγέθη N = 1024, 2048 και 8192 και για μεγέθη tile B = 16, 32, 64, 128, 256. Οι χρόνοι μετρώνται καθώς αυξάνεται ο αριθμός των νημάτων εκτέλεσης: από 1 έως 64 νήματα.

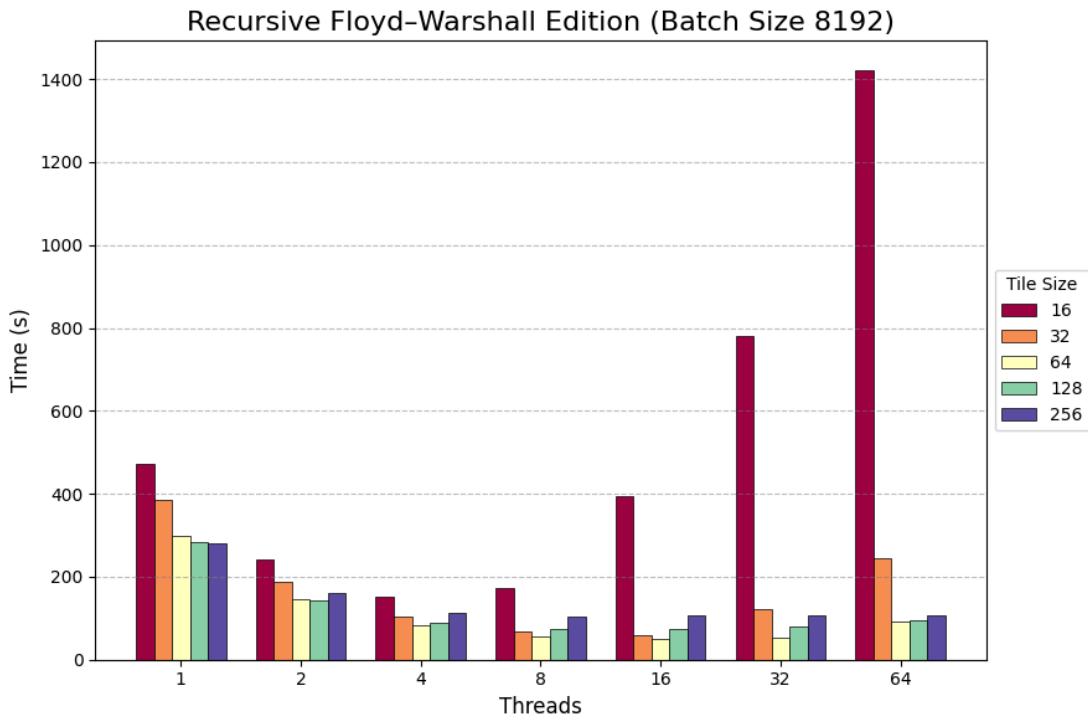
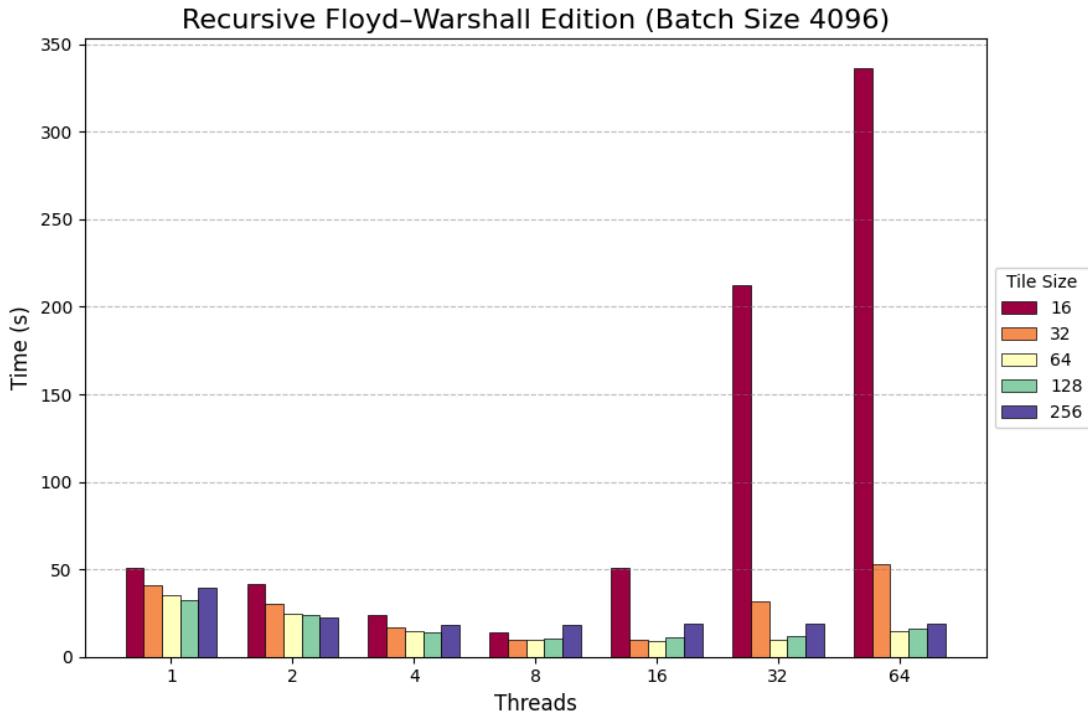
Λάβαμε τα εξής αποτελέσματα:

Recursive Floyd-Warshall Edition (Batch Size 1024)



Recursive Floyd-Warshall Edition (Batch Size 2048)





Είναι σαφές ότι οι χρόνοι της παραλληλοποιημένης έκδοσης είναι αρκετά καλύτεροι της απλής σειριακής εκτέλεσης, στην πλειοψηφία των configurations.

Τα αποτελέσματα και στα τρία μεγέθη δείχνουν ξεκάθαρα τα ακόλουθα φαινόμενα:

- Για μικρά block sizes, ο πίνακας χωρίζεται σε πάρα πολλά πολύ μικρά υποπροβλήματα. Στην αναδρομική υλοποίηση του Floyd–Warshall αυτό μεταφράζεται σε **τεράστιο πλήθος OpenMP tasks**, όπου κάθε task εκτελεί ελάχιστο υπολογισμό. Το κόστος δημιουργίας και προγραμματισμού (scheduling) αυτών των tasks από το runtime γίνεται μεγαλύτερο από τον ίδιο τον υπολογισμό, με αποτέλεσμα σημαντικά επιβραδύνσεις καθώς αυξάνεται ο αριθμός των νημάτων.
- Επιπλέον, τα πολύ μικρά blocks οδηγούν σε κακή spatial locality. Εφόσον κάθε block έχει μηδαμινό μέγεθος, πολύ μικρότερο από το μέγεθος ενός cache line, μεγάλο μέρος των δεδομένων που φέρνει στη cache η CPU δεν χρειάζονται για το συγκεκριμένο task-thread. Ας μην ξεχνάμε ότι στη C οι πίνακες είναι row major, δηλαδή αποθηκεύονται στη μνήμη κατά γραμμές, πράγμα που σημαίνει ότι στην cache θα έρχονται δεδομένα που δεν θα αξιοποιήσει καν το συγκεκριμένο thread, αφού θα βρίσκονται εκτός του block που επεξεργάζεται. Προκύπτει, επομένως, το ίδιο πρόβλημα που έχουμε στην υλοποίηση του Floyd–Warshall με for loops: φέρνουμε στην cache line blocks τα οποία έχουν λίγα χρήσιμα δεδομένα, με τα υπόλοιπα δεδομένα να χρειαστεί να ξαναέρθουν από τη μνήμη, αφού μέχρι να χρειαστούν το παλιό block που τα περιείχε θα έχει αντικατασταθεί.
- Για την κατάρρευση της απόδοσης σε εκτελέσεις με μικρό block size και πολλά threads, σε σχέση με τη σειριακή εκτέλεση και τη χρήση λίγων νημάτων, οφείλεται το cache coherence overhead. Πολλά threads εργάζονται σε γειτονικά στοιχεία που ανήκουν στην ίδια γραμμή cache, με αποτέλεσμα να υπάρχει έντονο φαινόμενο false sharing και cache-line ping-pong μεταξύ των πυρήνων (σαν τα MESI invalidations), με αποτέλεσμα μεγάλες καθυστερήσεις στο memory subsystem. Έτσι, η χρήση μικρού B όχι μόνο αυξάνει τα cache misses, αλλά προκαλεί και σημαντικό coherence traffic όταν αυξάνεται ο αριθμός των threads.
- Συνεπώς, παρατηρούμε ότι σε κάθε πείραμα, το μικρότερο block size που δοκιμάσαμε ($B=16$) είχε τη χειρότερη επίδοση (και σε μερικές περιπτώσεις, με τεράστια διαφορά, όπως για threads περισσότερα των 16, όπου η απόδοση καταρρέει λόγω πολλαπλών coherence misses και υπερφόρτωσης του scheduler)
- Η βέλτιστη συμπεριφορά παρατηρήθηκε για tiles μεσαίου μεγέθους, και ειδικά για $B=64$. Στην περίπτωση αυτή, συμπεραίνουμε ότι το κάθε block χωράει στις cache L1/L2 και κερδίζουμε αρκετά λόγω άριστου spatial locality.
- Για μεγαλύτερα tiles, όπως για $B=256$, η επίδοση ξεκινά να εμφανίζει σημάδια επιδείνωσης. Είναι σαφώς καλύτερη από τις περιπτώσεις πολύ μικρών tile sizes, αλλά εικάζουμε ότι η καθυστέρηση οφείλεται πάλι σε κακό cache locality (τα blocks δεν χωράνε όλα στην cache, που οδηγεί σε misses κατά την εκτέλεση του base case for loop). Μια άλλη παρατήρηση είναι ότι τα προγράμματα με μεγάλα tile sizes δεν κλιμακώνουν καλά με την αύξηση των νημάτων, ακόμα και σε περιπτώσεις που οι αντίστοιχες εκτελέσεις με μικρότερο B έδωσαν αισθητά καλύτερα αποτελέσματα. Αυτό ενδεχομένως να οφείλεται στο ότι μεγάλο B συνεπάγεται λίγα blocks, και άρα λιγότερα tasks για κάθε thread, με ενδεχόμενο κάποια να μένουν άπραγα. Το βλέπουμε καθαρά

στα διαγράμματα: το $B=256$ δεν καταρρέει όπως το $B=16$, αλλά δεν επιταχύνει πάνω από 8 threads.

- Σημαντικές είναι και οι παρατηρήσεις που κάναμε για τα διάφορα μεγέθη N που δοκιμάστηκαν. Συγκεκριμένα, με αύξηση του batch size, ο πίνακας γίνεται όλο και περισσότερο memory-bound, με αποτέλεσμα το block size να γίνεται ολοένα και πιο καθοριστικός παράγοντας για την επίδοση του αλγορίθμου, λόγω της επίδρασής του στην cache locality. Για μικρά block sizes, το working set χωρίζεται, λόγω του μεγέθους του, σε υπερβολικά πολλά sub-states, και η διαδικασία αυτή προσδίδει τεράστιο overhead (υπενθυμίζουμε ότι το working set του Floyd-Warshall, όντας τετραγωνικός πίνακας, έχει τάξη μεγέθους $O(N^2)$).
- Καταληκτικά, διαπιστώνουμε το εξής: το sweet-spot για κάθε batch size N είναι block size=64 ή 128, ώστε να επιτυγχάνεται το βέλτιστο locality, και η βέλτιστη επίδοση παρατηρείται στα 8 ή και 16 νήματα (ανάλογα το block size). Προφανώς, αν καλούμασταν να επιλέξουμε, θα προτιμούσαμε τα 8 λόγω του φθηνότερου κόστους για τη χρήση τους.

2.3 Ταυτόχρονες Δομές Δεδομένων

Εδώ καλούμαστε να μελετήσουμε διαφορετικές τεχνικές συγχρονισμού. Συγκεκριμένα μας δίνονται 5 διαφορετικές ταυτόχρονες υλοποιησεις μιας απλής συνδεδεμένης λίστας:

- Coarse-grain locking (**CGL**)
- Fine-grain locking (**FGL**)
- Optimistic Synchronization (**OPT**)
- Lazy Synchronization (**LAZY**)
- Non-blocking Synchronization (**NB**)

Θα πάρουμε μετρήσεις για διάφορους αριθμούς νημάτων, μέγεθος λίστας καθώς και ποσοστό λειτουργιών. Η κάθε εκτέλεση είναι προγραμματισμένη να διαρκεί 10 δευτερόλεπτα, και αυτό που μετράμε είναι το Throughput (σε Kops/sec).

Τα διαφορετικά configurations που δοκιμάζουμε για κάθε είδος ταυτόχρονης υλοποίησης είναι τα παρακάτω:

- Αριθμός Threads: 1, 2, 4, 8, 16, 32, 64, 128
- Μέγεθος Λίστας: 1024, 8192
- Ποσοστά Λειτουργιών: 100-0-0, 80-10-10, 20-40-40, 0-50-50 (ο πρώτος αριθμός υποδηλώνει το ποσοστό αναζητήσεων, ο δεύτερος το ποσοστό εισαγωγών, και ο τρίτος το ποσοστό διαγραφών)

Θα μετρήσουμε επίσης την απόδοση της σειριακής υλοποίησης της απλής συνδεδεμένης λίστας, για τα ίδια μεγέθη λίστας και τα ίδια ποσοστά λειτουργιών. Με αυτόν τον τρόπο θα διαθέτουμε ένα σημείο αναφοράς για σύγκριση με τις παράλληλες υλοποιήσεις.

Το μηχάνημα στο οποίο εκτελούμε τις μετρήσεις, το sandman, διαθέτει 32 φυσικούς πυρήνες. Μέχρι τώρα δοκιμάζαμε έως και 64 threads, κάνοντας χρήση του hyperthreading. Με αυτόν τον τρόπο κάθε φυσικός πυρήνας εμφανίζεται ως δύο λογικοί πυρήνες, μοιράζοντας τις εντολές των δύο νημάτων στους κοινούς πόρους, ανάλογα με το ποιοι από αυτούς είναι διαθέσιμοι σε κάθε κύκλο ρολογιού.

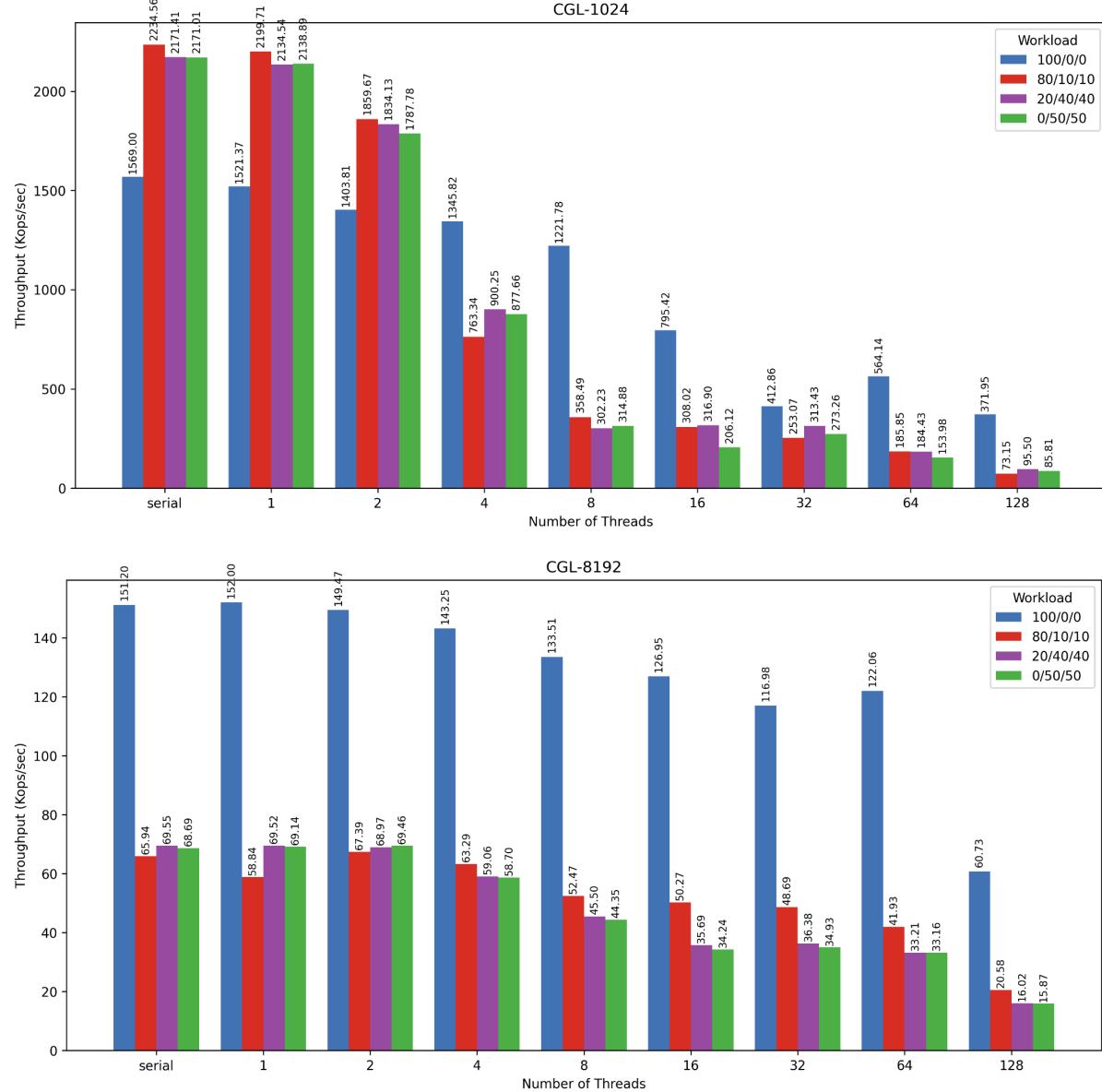
Πλέον θα αξιοποιήσουμε και το oversubscription, δηλαδή θα εκτελούμε περισσότερα νήματα από τους διαθέσιμους λογικούς πυρήνες του sandman, φτάνοντας συνολικά τα 128 threads. Σε αντίθεση με το hyperthreading (όπου κάθε φυσικός πυρήνας μπορεί να εξυπηρετεί ταυτόχρονα δύο λογικά νήματα αξιοποιώντας την αρχιτεκτονική του) το oversubscription δεν προσφέρει επιπλέον πραγματικό παραλληλισμό. Αντίθετα, όταν υπάρχουν περισσότερα threads από λογικούς πυρήνες, το λειτουργικό αναγκάζεται να κάνει context switching και έτσι κάθε στιγμή τρέχουν μόνο τόσα threads όσα οι λογικοί πυρήνες, ενώ τα υπόλοιπα περιμένουν. Αυτό μπορεί να είναι ωφέλιμο όταν υπάρχουν καθυστερήσεις λόγω αδράνειας (spinlocks, sleeps) ή λόγω έντονου I/O. Αντίθετα, το overhead του context switching είναι επιβλαβές όταν ο υπολογισμός είναι computation-intensive.

Coarse Grain Locking:

Το coarse-grain locking αποτελεί την απλούστερη τεχνική συγχρονισμού για ταυτόχρονες δομές δεδομένων, όπου ολόκληρη η δομή προστατεύεται από ένα και μοναδικό κλείδωμα. Το βασικό πλεονέκτημα της υλοποίησης αυτής είναι η προγραμματιστική ευκολία. Άλλα έτσι μειώνεται ο παραλληλισμός, καθώς για παράδειγμα δεν γίνεται ταυτόχρονα ένα νήμα να εισάγει έναν κόμβο στην αρχή της λίστας, ενώ ένα άλλο να διαγράφει έναν από το τέλος της λίστας. Επίσης η υλοποίηση αυτή δεν είναι robust, καθώς όσο αυξάνεται ο αριθμός των νημάτων αυξάνεται και ο ανταγωνισμός, και η αποτυχία ενός μόνο νήματος που “κρατά” κάποιο lock μπορεί να είναι καταστροφική. Και πράγματι παρατηρούμε πως σχεδόν σε κάθε διπλασιασμό των νημάτων, το throughput μειώνεται, καθιστώντας την σειριακή εκτέλεση καλύτερη από την προσπάθεια παραλληλοποίησης.

Ακόμα, βλέπουμε πως η πτώση του throughput στη λίστα μεγέθους 8192 είναι μικρότερη σε σχέση με τη λίστα μεγέθους 1024. Αυτό οφείλεται στο ότι, όσο μεγαλύτερη είναι η λίστα, τόσο περισσότερο χρόνο απαιτεί κάθε πράξη (contains/add/remove) για να ολοκληρωθεί. Έτσι, παρότι το global lock εξακολουθεί να περιορίζει τη ρυθμαπόδοση, τα νήματα προσπαθούν να το αποκτήσουν πολύ πιο αραία, αφού κάθε λειτουργία διαρκεί περισσότερο. Ως αποτέλεσμα, η συνολική συμφόρηση στο lock είναι μικρότερη και η αύξηση του αριθμού των νημάτων υποβαθμίζει τη ρυθμαπόδοση πιο ήπια σε σχέση με τη μικρή λίστα, όπου οι λειτουργίες είναι πολύ γρηγορότερες.

Τέλος, παρατηρούμε πως όταν το ποσοστό των “contains” είναι μεγαλύτερο σε σχέση με τις εισαγωγές και διαγραφές, η απόδοση είναι καλύτερη, προφανώς διότι όταν ένα νήμα δεν κάνει add ή remove ενός κόμβου, χρησιμοποιεί για λιγότερη ώρα το lock.

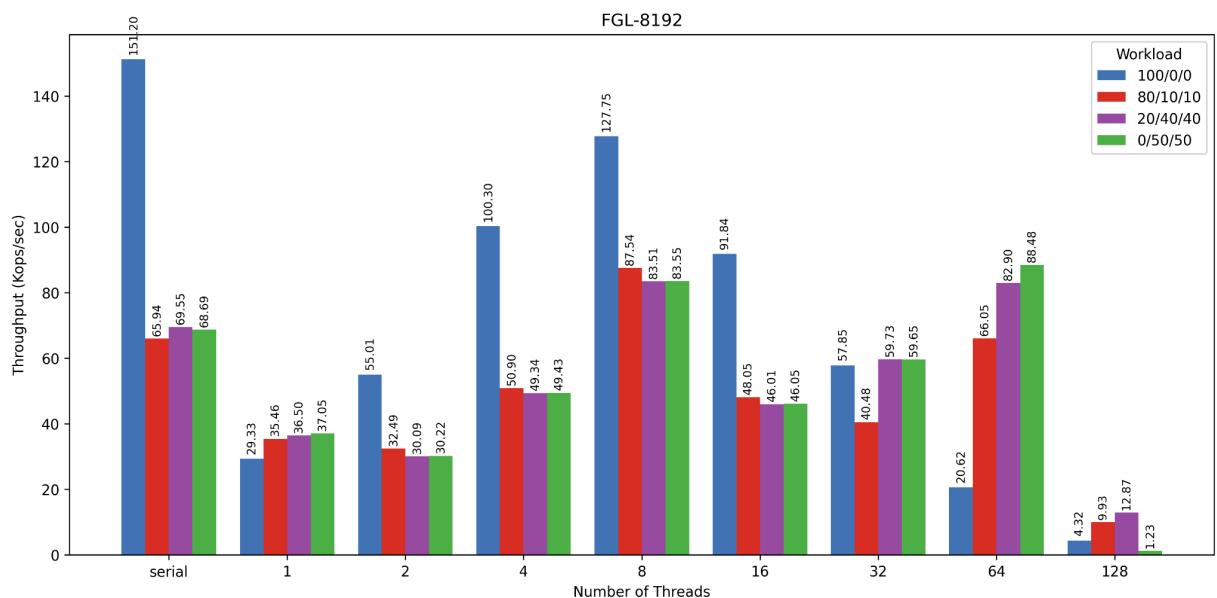
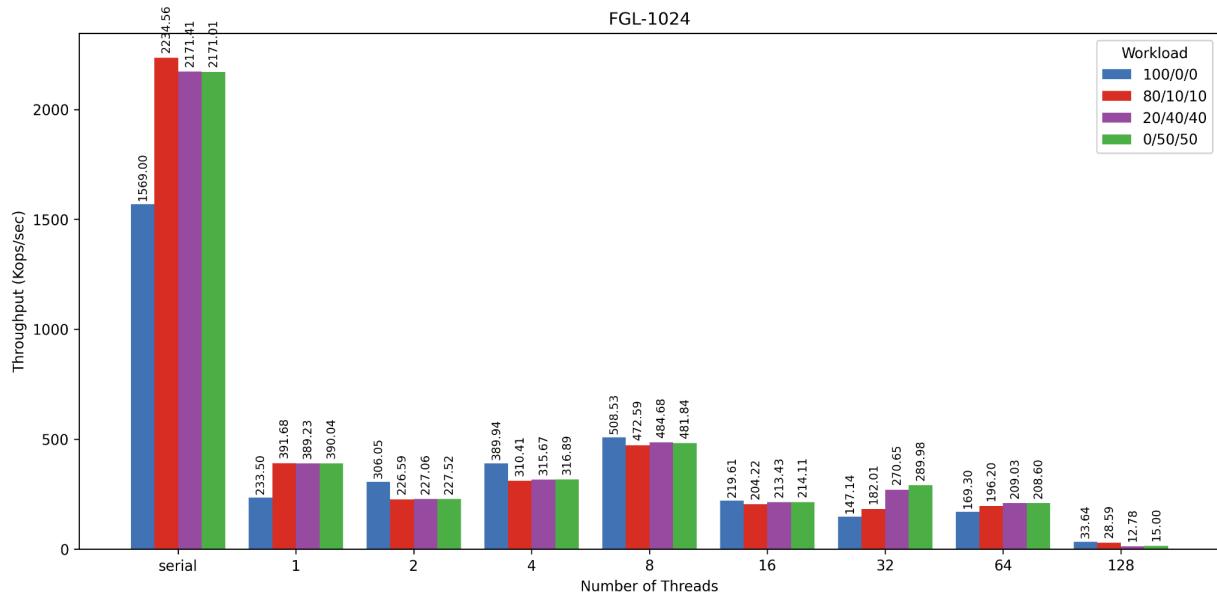


Fine-Grain Locking:

Στο fine-grain locking η λίστα προστατεύεται με πολλά μικρά κλειδώματα, ένα σε κάθε κόμβο, και η πρόσβαση γίνεται με την τεχνική hand-over-hand locking. Αυτό σημαίνει ότι το νήμα κλειδώνει τον επόμενο κόμβο πριν αποδεσμεύσει τον προηγούμενο, προχωρώντας βήμα-βήμα μέσα στη λίστα. Έτσι, υλοποιείται ο αμοιβαίος αποκλεισμός, ενώ ταυτόχρονα επιτρέπεται σε πολλά νήματα να εκτελούν ανεξάρτητες λειτουργίες σε διαφορετικά σημεία της λίστας (αν και συνεχίζουν αυτά να μπλοκάρουν άλλα νήματα που θέλουν να προχωρήσουν σε μετέπειτα κόμβους της λίστας). Ούτε εδώ έχουμε robustness προφανώς αφού συνεχίζουμε να χρησιμοποιούμε locks, αλλά πλέον έχουμε μειώσει και την προγραμματιστική ευκολία.

Όσο για την απόδοση, από την μία το fine grain locking επιτυγχάνει πολύ καλύτερη κλιμάκωση και υψηλότερο throughput όσο αυξάνεται ο αριθμός των νημάτων, αλλά κατά βάση παραμένει χειρότερο από την σειριακή εκτέλεση. Εξαίρεση αποτελεούν μερικά configuration όπως για παράδειγμα τα 8 threads στην λίστα μεγέθους 8192 για ποσοστά λειτουργιών 80-10-10. Αντίθετα, στο μικρό μέγεθος λίστας, το throughput είναι ΠΟΛΥ χειρότερο από το σειριακό για κάθε ποσοστό λειτουργιών και για κάθε αριθμό νημάτων.

Τέλος, παρατηρούμε πως το over-subscription εδώ έχει χείριστη απόδοση και για τα 2 μεγέθη λίστας, καθώς πέρα από το lock contention έχουμε και υπερβολικό overhead από τα context switches.

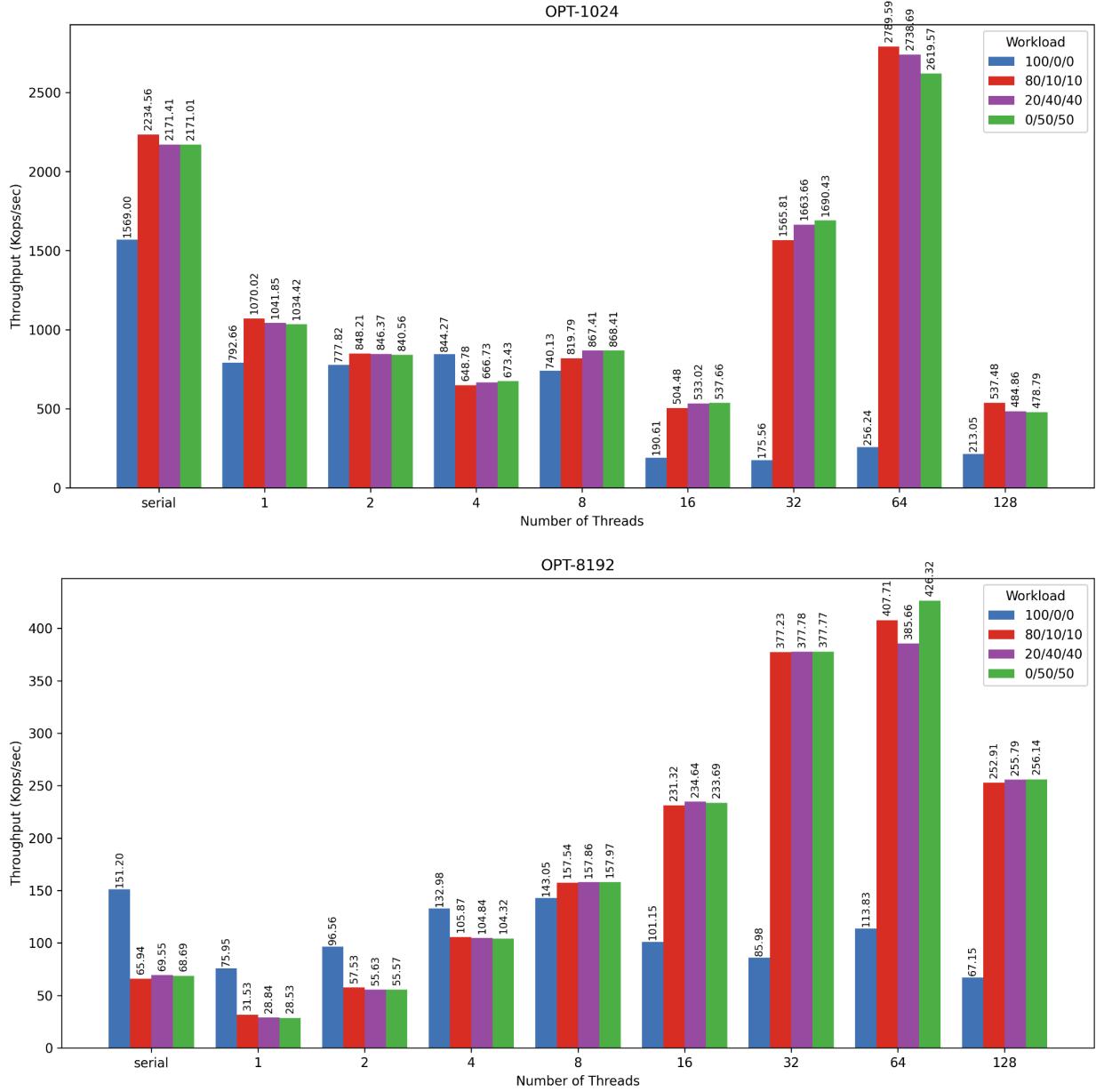


Optimistic Synchronization:

To optimistic synchronization επιχειρεί να μειώσει το κόστος του fine-grain locking βασιζόμενο στην αισιόδοξη υπόθεση ότι οι συγκρούσεις μεταξύ νημάτων είναι σχετικά σπάνιες. Κατά την εισαγωγή ή διαγραφή, το νήμα διατρέχει τη λίστα χωρίς κλειδώματα, εντοπίζοντας πρώτα τους κόμβους που το ενδιαφέρουν (pred, curr). Μόνο όταν φτάσει στη σωστή θέση επιχειρεί να αποκτήσει τα απαραίτητα κλειδώματα στους δύο κόμβους. Αφού αποκτήσει τα locks, πραγματοποιεί έναν έλεγχο συνέπειας (validation) για να επιβεβαιώσει ότι η δομή δεν έχει αλλάξει από άλλο νήμα. Συγκεκριμένα, ελέγχει ότι οι κόμβοι εξακολουθούν να υπάρχουν και να είναι διαδοχικοί. Αν ο έλεγχος αποτύχει, τα locks απελευθερώνονται και η λειτουργία επαναλαμβάνεται. Καταλαβαίνουμε συνεπώς πως η υλοποίηση αυτή είναι καλύτερη από το fine-grain εφόσον το κόστος να διατρέξουμε την λίστα 2 φορές χωρίς κλειδώματα, είναι χαμηλότερο από το να την διασχίσουμε μία φορά με hand-over-hand locking.

Πλέον έχουμε την πρώτη υλοποίηση που έχει καλή κλιμάκωση και καταφέρνει καλύτερες επιδόσεις από την σειριακή εκτέλεση. Συγκεκριμένα, για την λίστα μεγέθους 1024, για αριθμό νημάτων ως 32 το contention των locks είναι μεγαλύτερο από το όφελος της παραλληλοποίησης, αλλά στα 64 threads η παραλληλοποίηση “νικάει” και το throughput γίνεται μέγιστο. Εξαίρεση αποτελεί η εκτέλεση με 100% λειτουργίες contains. Εκεί, η σειριακή υλοποίηση συνεχίζει να είναι καλύτερη, καθώς απλά προσπελάσεται η λίστα, ενώ στο optimistic synchronization έχουμε και overhead από τα validations (που εδώ θα είναι πάντα true). Κακό είναι και το performance των 128 threads.

Στην λίστα μεγέθους 8192, που όπως είδαμε και προηγουμένως το μεγάλο της μέγεθος σημαίνει ότι ο ανταγωνισμός για τα locks είναι μικρότερος, ήδη από τα 4 threads η απόδοση γίνεται καλύτερη από την σειριακή εκτέλεση, με εξαίρεση προφανώς την λειτουργία με 100% contains. Αξιοσημείωτο είναι πως το oversubscription εδώ αποδίδει. Όχι όσο τα 64 threads, αλλά η ρυθμαπόδοση του βρίσκεται στο επίπεδο των 16 threads - υψηλότερα από το serial δηλαδή.



Lazy Synchronization:

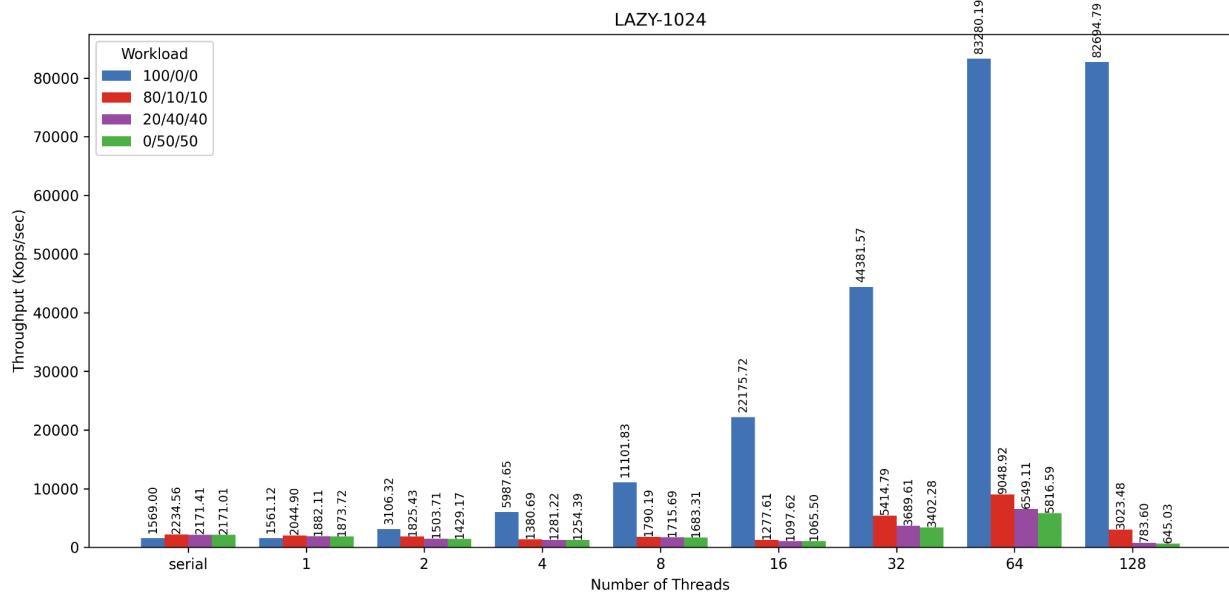
Το lazy synchronization βασίζεται στην ίδια λογική με το optimistic synchronization, αλλά βελτιώνει περαιτέρω την απόδοση μειώνοντας το κόστος των κλειδωμάτων και του validation. Η βασική ιδέα είναι ότι οι κόμβοι δεν αφαιρούνται άμεσα από τη δομή: πρώτα “αφαιρούνται” λογικά μέσω μιας boolean μεταβλητής, και η φυσική τους αφαίρεση γίνεται αργότερα από ένα νήμα που θα τους προσπελάσει. Η μέθοδος contains εκτελείται πλήρως χωρίς κλειδώματα, διατρέχει τη λίστα και ελέγχει μόνο την boolean μεταβλητή για να δει αν ένας κόμβος είναι έγκυρος. Η add και η remove διατρέχουν τη λίστα χωρίς locking, αλλά κλειδώνουν μόνο τους κόμβους pred και curr όταν εντοπίσουν το σωστό σημείο. Το validation, σε αντίθεση με την optimistic υλοποίηση,

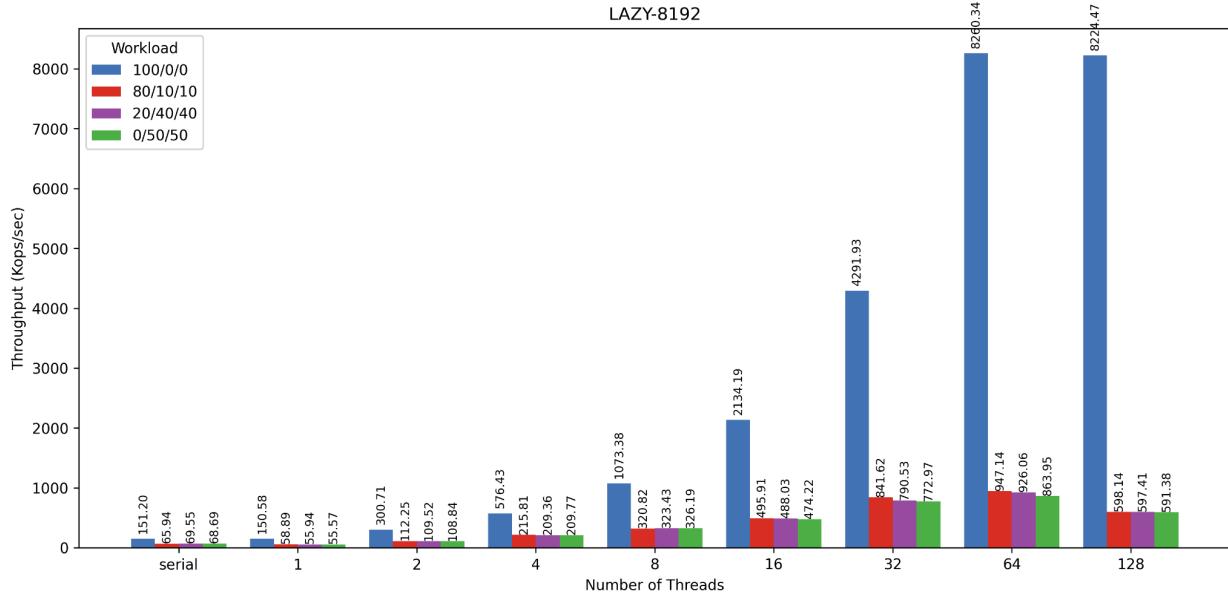
δεν απαιτεί να ξαναδιατρέξουμε τη λίστα από την αρχή - αντίθετα, γίνονται τοπικοί έλεγχοι συνέπειας μόνο στους δύο εμπλεκόμενους κόμβους.

Πλέον είναι σαφές ότι το lazy synchronization επιτυγχάνει πολύ καλή επίδοση και κλιμάκωση. Στη “μικρή” λίστα, οι εκτελέσεις με 100% contains παρουσιάζουν ήδη από τα 2 νήματα αύξηση της ρυθμαπόδοσης σε σχέση με τη σειριακή υλοποίηση, ενώ για όλα τα υπόλοιπα workloads τα 32 και 64 threads ξεπερνούν αρκετά την απόδοση του serial.

Στη “μεγάλη” λίστα τα αποτελέσματα είναι ακόμη καλύτερα. Για όλα τα ποσοστά λειτουργιών, η παράλληλη εκτέλεση δίνει μεγαλύτερο throughput από τη σειριακή ήδη από τα 2 threads.

Τέλος, το πιο αξιοσημείωτο εύρημα είναι η εξαιρετικά καλή απόδοση του oversubscription στα 128 threads. Τα 128 threads αποδίδουν καλύτερα από τα 32 και μόνο ελάχιστα χειρότερα από τα 64 threads. Παρά την ενδιαφέρουσα αυτή συμπεριφορά όμως, το oversubscription εξακολουθεί να μην αποτελεί πρακτικό ή αποδοτικό μοντέλο εκτέλεσης.





Non-Blocking Synchronization:

Το non-blocking synchronization αποτελεί το τελικό και πιο προηγμένο στάδιο βελτιστοποίησης των ταυτόχρονων λιστών, καθώς εξαλείφει πλήρως τη χρήση κλειδωμάτων από όλες τις βασικές λειτουργίες. Σε αντίθεση με τις προηγούμενες μεθόδους, η συνέπεια της δομής εξασφαλίζεται αποκλειστικά μέσω ατομικών ενεργειών του επεξεργαστή - μέσω της εντολής Compare-and-Swap (CAS). Η βασική ιδέα είναι ότι τα κρίσιμα πεδία ενός κόμβου (next address και το bit marked) συνδυάζονται σε μία ατομική μονάδα ώστε οποιαδήποτε ενημέρωση να μπορεί να ελεγχθεί και να εφαρμοστεί με μία CAS.

Οι 4 βασικές συναρτήσεις που χρησιμοποιούνται είναι οι contains, find, remove και add.

Η contains είναι πλέον wait free καθώς δεν έχει κλειδώματα, και όλα τα νήματα που την εκτελούν ταυτόχρονα προοδεύουν.

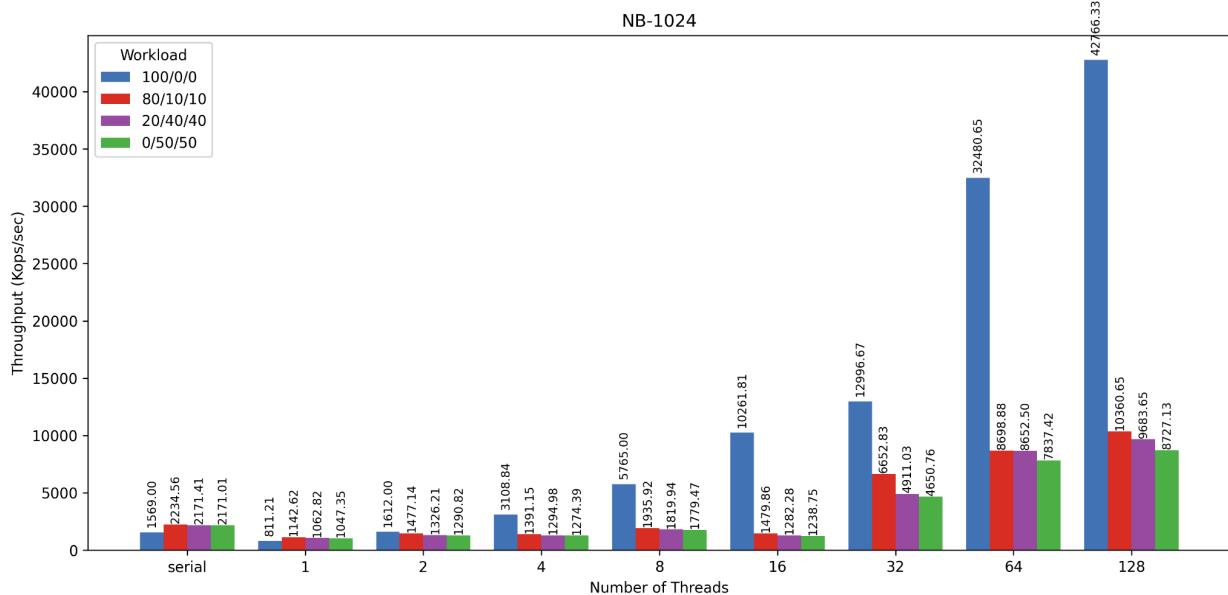
Η add δημιουργεί και αρχικοποιεί τον νέο κόμβο, και προσπαθεί συνεχόμενα να εισάγει τον νέο κόμβο μέχρι να τα καταφέρει (η εισαγωγή γίνεται απλά με μία Compare and Set λειτουργία).

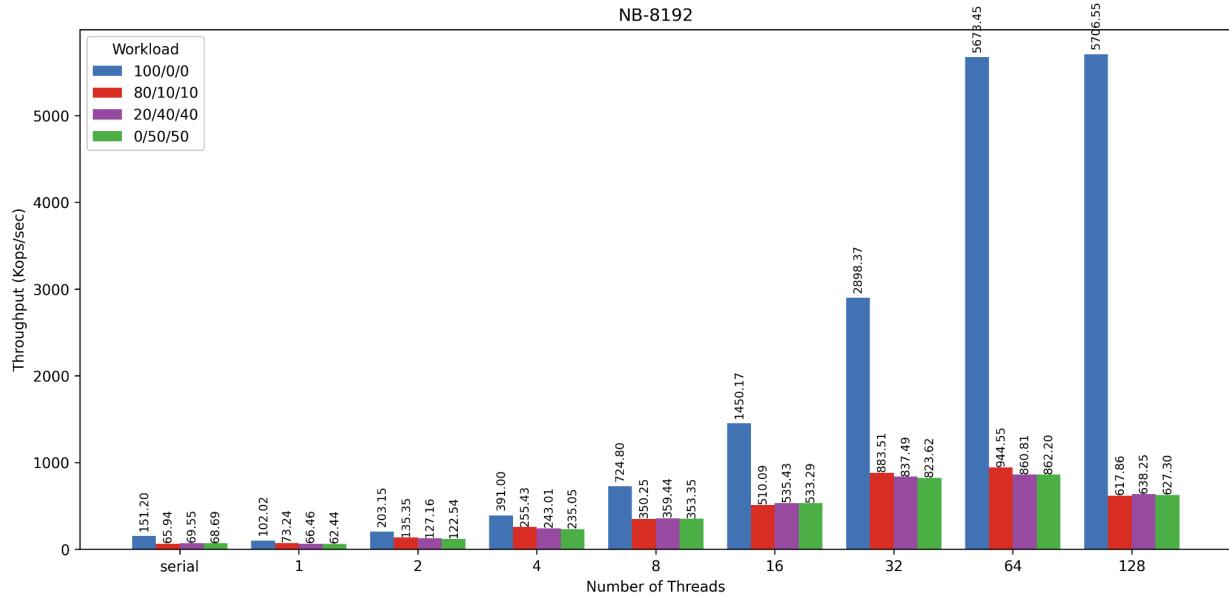
Για την διαγραφή ενός κόμβου, καλείται αρχικά η συνάρτηση find η οποία διατρέχει την λίστα μέχρι να βρει ένα στοιχεία μεγαλύτερο από το ζητούμενο. Παράλληλα όμως, αν βρει ένα στοιχεία που έχει σβηστεί λογικά θα προσπαθήσει να το σβήσει και φυσικά, και αν αποτύχει θα ξαναρχίσει από την αρχή. Αφού ολοκληρωθεί καλείται remove η οποία προσπαθεί μέχρι να καταφέρει να αφαιρέσει λογικά τον κόμβο, και αφού τα καταφέρει κάνει και μία προσπάθεια να τον αφαιρέσει και φυσικά. Αν αποτύχει, δεν ξανα-προσπαθεί αφού μία find για ένα επόμενο remove θα πραγματοποιήσει την φυσική αυτή διαγραφή.

Με μια πρώτη ματιά, η non-blocking υλοποίηση μπορεί να φαίνεται αντιπαραγωγική. Για παράδειγμα, ένα νήμα που θέλει να διαγράψει έναν κόμβο πρέπει πρώτα να καλέσει τη find, η οποία όχι μόνο καθαρίζει λογικά διαγραμμένους κόμβους που συναντά, αλλά ενδέχεται και να αναγκαστεί να ξαναξεκινήσει ολόκληρη τη διάσχιση της λίστας εάν αποτύχει. Ωστόσο, αυτό το επιπλέον κόστος είναι ένα απόλυτα λογικό τίμημα για το βασικό πλεονέκτημα της non-blocking προσέγγισης: την πλήρη απουσία κλειδωμάτων. Η απουσία των locks εξασφαλίζει και robustness, αφού πλέον δεν υπάρχει ο κίνδυνος να αποτύχει ολόκληρη η εφαρμογή λόγω ενός νήματος που απέτυχε καθώς κρατούσε ένα lock.

Πειραματικά, γίνεται ξεκάθαρο ότι η non-blocking υλοποίηση επιτυγχάνει εξαιρετικό throughput, ξεπερνώντας τη σειριακή εκτέλεση σχεδόν σε κάθε configuration. Στη “μικρή” λίστα, η απόδοση αρχίζει να υπερτερεί σημαντικά από τα 32 threads και πάνω, ενώ στη “μεγάλη” λίστα το κέρδος εμφανίζεται ακόμη νωρίτερα, ήδη από τα 16 threads. Και βλέπουμε κιόλας την ιδιαίτερα υψηλή ρυθμαπόδοση στις εκτελέσεις που έχουμε 100% λειτουργίες contains, αφού αυτές είναι, όπως είπαμε προηγουμένως, wait-free.

Παρόλα αυτά, πρέπει να επισημανθεί πως το non-blocking έχει ένα βασικό μειονέκτημα. Πολύ δύσκολο προγραμματισμό (ενδεχομένως ακόμα και αδύνατο σε άλλες δομές).





3. Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Εισαγωγή στον προγραμματισμό με GPU

Οι κάρτες γραφικών είναι συσκευές σχεδιασμένες με γνώμονα την μεγιστοποίηση του throughput κατά την εκτέλεση κώδικα σε αυτές. Σε αντίθεση με τους επεξεργαστές (CPUs), που υλοποιούν αυστηρότερα μοντέλα μνήμης για τη διατήρηση της συνέπειας των προγραμμάτων και της ροής ελέγχου, οι GPUs δίνουν την ελευθερία ασταμάτητης παράλληλης εκτέλεσης κώδικα, χωρίς την επιβολή ακολουθιακής συνέπειας από το hardware. Αποτελούνται από χιλιάδες μικρότερους και απλούς πυρήνες, οι οποίοι μπορούν να εκτελούν ταυτόχρονα την ίδια εντολή σε διαφορετικά δεδομένα. Αυτή η αρχιτεκτονική τις καθιστά ιδανικές για προβλήματα που χαρακτηρίζονται από παραλληλισμό δεδομένων, όπως είναι η επεξεργασία εικόνας, οι γραμμικές άλγεβρες και, φυσικά, αλγόριθμοι μηχανικής μάθησης όπως ο K-means, που θα μας απασχολήσει στην παρούσα άσκηση.

Στο προγραμματιστικό μοντέλο CUDA, η βασική μονάδα εκτέλεσης είναι το **νήμα (thread)**. Τα νήματα οργανώνονται σε ομάδες που ονομάζονται **blocks**, και τα blocks με τη σειρά τους σχηματίζουν ένα **πλέγμα (grid)**. Όταν καλούμε μια συνάρτηση που εκτελείται στην κάρτα γραφικών, η οποία ονομάζεται **πυρήνας (kernel)**, καθορίζουμε πόσα blocks και πόσα threads ανά block θα ενεργοποιηθούν. Σε επίπεδο υλικού, τα threads εκτελούνται σε ομάδες των 32, που ονομάζονται **warps**. Όλα τα threads ενός warp εκτελούν την ίδια εντολή ταυτόχρονα (αρχιτεκτονική SIMT - Single Instruction, Multiple Threads). Αυτό σημαίνει ότι αν τα threads ενός

warp αποκλίνουν λόγω κάποιου branch, η εκτέλεση ουσιαστικά “σειριοποιείται”, καθώς για παράδειγμα, θα πρέπει πρώτα να εκτελεστούν τα threads σε δεδομένα για τα οποία το branch είναι taken, και μετά αυτά για τα οποία το branch είναι not-taken.

Ένα επιπρόσθετο κρίσιμο στοιχείο για την επίδοση στις GPUs είναι η ιεραρχία μνήμης. Στο ανώτερο επίπεδο υπάρχει η **global memory**, η οποία είναι μεγάλης χωρητικότητας αλλά αργή στην προσπέλαση. Για να επιτευχθεί υψηλή απόδοση, είναι ζωτικής σημασίας η πρόσβαση στην καθολική μνήμη να γίνεται με συγκεκριμένο τρόπο (memory coalescing), όπου γειτονικά threads διαβάζουν γειτονικές θέσεις μνήμης. Επιπλέον, υπάρχει η **διαμοιραζόμενη μνήμη (shared memory)**, η οποία είναι πολύ μικρότερη αλλά, ως on-chip memory, είναι εξαιρετικά γρήγορη, επιτρέποντας την ταχύτατη ανταλλαγή δεδομένων μεταξύ των threads του ίδιου block. Η βελτιστοποίηση ενός αλγορίθμου σε CUDA συχνά ανάγεται στη βελτιστη διαχείριση αυτών των τύπων μνήμης και στην ελαχιστοποίηση της επικοινωνίας μεταξύ της CPU (Host) και της GPU (Device), καθώς ο δίσαυλος επικοινωνίας PCIe έχει περιορισμένο εύρος ζώνης.

Εισαγωγή στην Άσκηση

Στη συγκεκριμένη άσκηση κληθήκαμε να παραλληλοποιήσουμε τον αλγόριθμο k-means σε επεξεργαστές γραφικών. Τα πειράματα έγιναν με προοδευτική μετάβαση από μια σειριακή εκτέλεση στην CPU σε μια πλήρως επιταχυνόμενη εκτέλεση στην GPU, αναλύοντας τα bottlenecks σε κάθε στάδιο και εφαρμόζοντας τεχνικές βελτιστοποίησης μνήμης και υπολογισμών.

Η πρώτη εκτέλεση ήταν η σειριακή (όλος ο υπολογισμός του kmeans στη CPU). Στη συνέχεια, υλοποιήσαμε την “naive” εκδοχή, όπου η κάρτα γραφικών ανέλαβε το υπολογιστικά βαρύ κομμάτι του αλγορίθμου, δηλαδή τον υπολογισμό των nearest clusters σε κάθε iteration. Έπειτα βελτιστοποίησαμε ως προς την πρόσβαση του κώδικα σε δομές μνήμης, χρησιμοποιώντας μετασχηματισμούς για καλύτερο data locality (transpose version) και αξιοποιώντας κατώτερου ιεραρχικού επιπέδου μνήμη (shared memory). Τέλος, υλοποιήσαμε την πλήρως offloaded έκδοση του αλγορίθμου στη GPU (υπολογισμός nearest clusters και νέων κέντρων αποκλειστικά στη GPU), με δύο περαιτέρω εκδοχές με χρήση reduction.

Ανάλυση του δοσμένου codebase

Πριν προχωρήσουμε στην αναλυτική επεξήγηση κάθε ζητουμένου, αξίζει να κάνουμε μια επισκόπηση του κώδικα που ήδη μας δίνεται, ώστε να υπάρχει πλήρης σαφήνεια σχετικά με τη ροή της εκτέλεσης του αλγορίθμου. Κατ’αρχάς, το entrypoint του προγράμματος Cuda είναι το αρχείο `main_gru.cu`. Το αρχείο αυτό εξυπηρετεί τρεις σκοπούς: να δημιουργηθεί το dataset πάνω στο οποίο θα εκτελεστεί ο αλγόριθμος (μέσω της συνάρτησης `dataset_generation` που ορίζεται στο βοηθητικό αρχείο `file_io.c`), να ξεκινήσει ο βασικός υπολογισμός στη GPU (μέσω της κλήσης του σωστού kernel `kmeans_gru` το οποίο υποβάλουμε στην ουρά) και να ελεγχθεί η ορθότητα του εκάστοτε πυρήνα, κατά βούληση, μέσω της προσθήκης της σημαίας `VALIDATE_FLAG=-DVALIDATE` κατά τη μεταγλώτιση. Παρομοίως, το entrypoint για τη σειριακή εκτέλεση στη cpu είναι το `main_seq.c`.

Αξιοσημείωτες είναι και οι υλοποιήσεις μερικών βιοθητικών συναρτήσεων σε επίπεδο kernel που μας ζητήθηκαν και χρησίμευσαν σε όλες τις υλοποιήσεις μας. Η σημαντικότερη ίσως από αυτές είναι η `get_tid()`, η οποία υπολογίζει το μοναδικό αναγνωριστικό (global ID) κάθε νήματος (thread) σε ένα μονοδιάστατο πλέγμα εκτέλεσης CUDA.

Η συνάρτηση αξιοποιεί τις ενσωματωμένες μεταβλητές της CUDA:

- `blockIdx.x`: το αναγνωριστικό του block στο οποίο ανήκει το νήμα
- `blockDim.x`: τον αριθμό των νημάτων ανά block,
- `threadIdx.x`: τη θέση του νήματος μέσα στο block.

Η υλοποίηση της συνάρτησης είναι η παρακάτω:

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

Μία άλλη βιοθητική συνάρτηση που χρησιμοποιήθηκε εκτενώς είναι αυτή για τον υπολογισμό της ευκλείδιας απόστασης δύο σημείων του χώρου. Ειδικότερα, χρησιμοποιήθηκαν δύο εκδοχές: η `euclid_dist_2()` και η `euclid_dist_2_transpose()`, με την πρώτη να χρησιμοποιεί την αρχική μορφή του πίνακα `objects` (σε row-major διάταξη) ενώ η δεύτερη υλοποιεί τον ίδιο μηχανισμό αλλά για την περίπτωση που τα δεδομένα είναι σε column-major διάταξη. Και οι δύο συναρτήσεις υπολογίζουν το τετράγωνο της ευκλείδιας απόστασης μεταξύ ενός αντικειμένου και ενός cluster center σε πολυδιάστατο χώρο. Να σημειωθεί ότι αποφεύγεται ο υπολογισμός της τετραγωνικής ρίζας, καθώς δεν μας ενδιαφέρει η ακριβής τιμή της απόστασης αλλά μόνο η σύγκριση της με άλλες αποστάσεις. Εφόσον η τετραγωνική ρίζα είναι γνησίως μονότονη συνάρτηση, η διάταξη των αποστάσεων διατηρείται και συνεπώς η σύγκριση μπορεί να γίνει ισοδύναμα χρησιμοποιώντας τα τετράγωνα των αποστάσεων. Αυτό μειώνει το υπολογιστικό κόστος χωρίς να επηρεάζει το αποτέλεσμα. Στα αντίστοιχα ερωτήματα θα αναλυθούν εκτενώς οι δύο αυτές υλοποιήσεις.

Ζητούμενο 1 - Naive Version

Στο ζητούμενο αυτό καλούμαστε να συμπληρώσουμε τον κώδικα που μας δίνεται από το εργαστήριο, αναθέτοντας στην κάρτα γραφικών το υπολογιστικά βαρύ κομμάτι του αλγορίθμου, δηλαδή τον υπολογισμό των ευκλείδειων αποστάσεων κάθε αντικειμένου από τα κέντρα των συστάδων και η εύρεση του πλησιέστερου cluster για το καθένα.

Συμπληρώνουμε τον κώδικα για την συνάρτηση `euclid_dist_2()`, η οποία υπολογίζει το τετράγωνο της ευκλείδειας απόστασης μεταξύ ενός αντικειμένου και ενός cluster center σε πολυδιάστατο χώρο. Είναι δηλωμένη ως `__host__` και `__device__`, ώστε να μπορεί να κληθεί τόσο στην CPU όσο και στη GPU.

Ο κώδικας της συνάρτησης είναι ο παρακάτω:

```

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

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

    return (ans);
}

```

Έπειτα, προχωράμε στον κώδικα του kernel **find_nearest_cluster** που θα εκτελείται στην κάρτα γραφικών.

Αρχικά πρέπει να διασφαλίσουμε ότι κάθε νήμα αντιστοιχεί σε έγκυρο αντικείμενο. Επειδή συνήθως εκτελούνται περισσότερα νήματα από όσα είναι τα object, ελέγχουμε αν **tid < numObjs** ώστε μόνο τα κατάλληλα νήματα να εκτελέσουν τον υπολογισμό.

Έπειτα χρησιμοποιούμε κατάλληλα την συνάρτηση **euclid_dist_2()** για να βρούμε το cluster που έχει την μικρότερη απόσταση από το τρέχον αντικείμενο.

Τέλος, για όσα αντικείμενα αλλάζουν το κοντινότερο cluster στο συγκεκριμένο iteration, πρέπει να αυξηθεί ο μετρητής **devdelta**, ο οποίος χρησιμοποιείται για τον έλεγχο σύγκλισης του αλγορίθμου. Επειδή όμως πολλά νήματα μπορεί να προσπαθήσουν ταυτόχρονα να ενημερώσουν αυτή τη μεταβλητή, προκύπτει race condition. Για τον λόγο αυτό χρησιμοποιείται η συνάρτηση **atomicAdd()**, η οποία εξασφαλίζει ορθή ενημέρωση της τιμής. Ωστόσο, πρέπει να σημειωθεί ότι η **atomicAdd()** εφαρμόζεται σε επίπεδο ολόκληρης της συσκευής και όχι μόνο σε επίπεδο **block**. Αυτό σημαίνει ότι όλα τα νήματα από όλα τα **blocks** συγχρονίζονται πάνω στην ίδια μεταβλητή, γεγονός που αυξάνει τον συγχρονισμό και μπορεί να δημιουργήσει συμφόρηση.

```

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

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

    // Maybe we have more threads than actual objects
    if (tid < numObjs) {
        int index, i;
        double dist, min_dist;

        /* find the cluster id that has min distance to object */
        index = 0;
        // We need to calculate the distance from cluster 0 - our starting point
        min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects,
                               deviceClusters, tid, 0);

        for (i = 1; i < numClusters; i++) {
            dist = euclid_dist_2(numCoords, numObjs, numClusters, objects,
                               deviceClusters, tid, i);
            /* no need square root */
            if (dist < min_dist) { /* find the min and its array index */
                min_dist = dist;
                index = i;
            }
        }
    }

    if (deviceMembership[tid] != index) {
        // This addition must be done atomically
        atomicAdd(devdelta, 1.0);
    }

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

```

Τέλος, θα συμπληρώσουμε μερικά σημαντικά σημεία στην συνάρτηση **kmeans_gpu**, η οποία καλείται από **main_gpu.cu**:

Στον δοθέντα κώδικα έχει ήδη υπολογιστεί το block size του kernel στην μεταβλητή **numThreadsPerClusterBlock**, και μας μένει να υπολογίσουμε και το grid size, με τον παρακάτω κώδικα:

```
const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock
- 1)/ numThreadsPerClusterBlock;
```

Έτσι εξασφαλίζουμε ότι δημιουργούνται αρκετά blocks ώστε να αντιστοιχηθεί τουλάχιστον ένα νήμα σε κάθε αντικείμενο. Η πρόσθεση του **numThreadsPerClusterBlock - 1** πριν από τη διαίρεση επιτρέπει την ορθή στρογγυλοποίηση προς τα πάνω, καλύπτοντας και την περίπτωση όπου ο αριθμός των αντικειμένων δεν είναι πολλαπλάσιος του μεγέθους του block.

Έχει ήδη υλοποιηθεί από το εργαστήριο η δέσμευση της απαραίτητης μνήμης με την χρήση της **cudaMalloc()**, καθώς και η αντιγραφή των πινάκων **object** και **membership** με χρήση της συνάρτησης **cudaMemcpy()**, που γίνονται στην αρχή του K-means αλγορίθμου.

Εμείς, σε κάθε iteration του αλγορίθμου θα φροντίσουμε τις παρακάτω μεταφορές δεδομένων μεταξύ host και device:

- clusters (CPU) to deviceClusters(GPU):

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

Και αφού κληθεί ο kernel (και η CPU περιμένει την ολοκλήρωση εκτέλεσης του με την χρήση της **cudaDeviceSynchronize()**, θα μεταφέρουμε από την GPU στη CPU τα εξής:

- deviceMembership (GPU) to membership (CPU)
- dev_delta_ptr (GPU) to delta (CPU)

Ο κώδικας είναι ο παρακάτω:

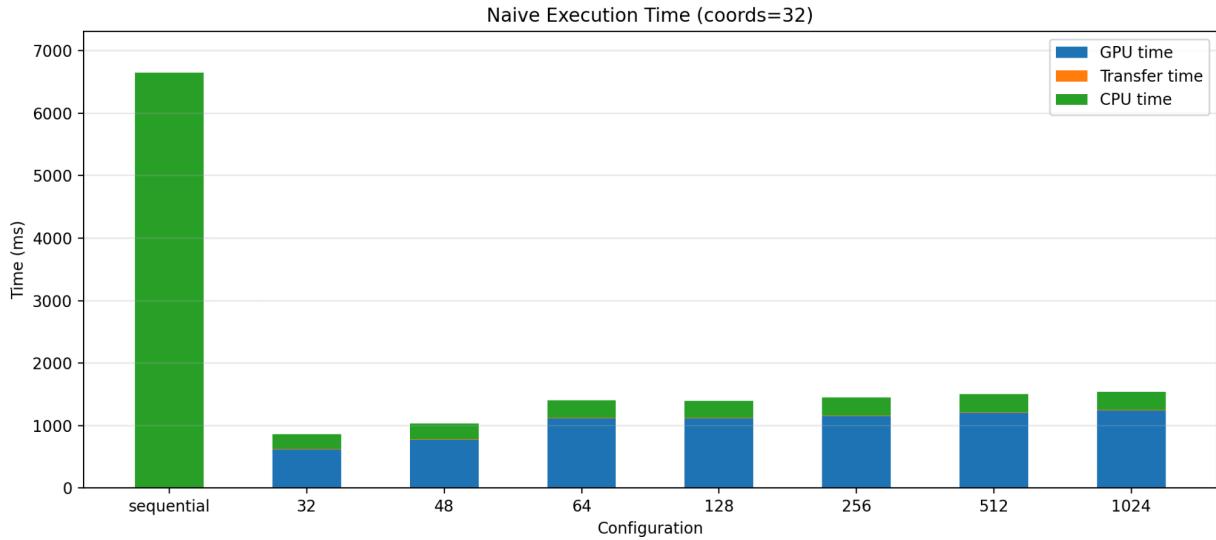
```
checkCuda(cudaMemcpy(membership, deviceMembership, numObjs *
sizeof(int), cudaMemcpyDeviceToHost));
```

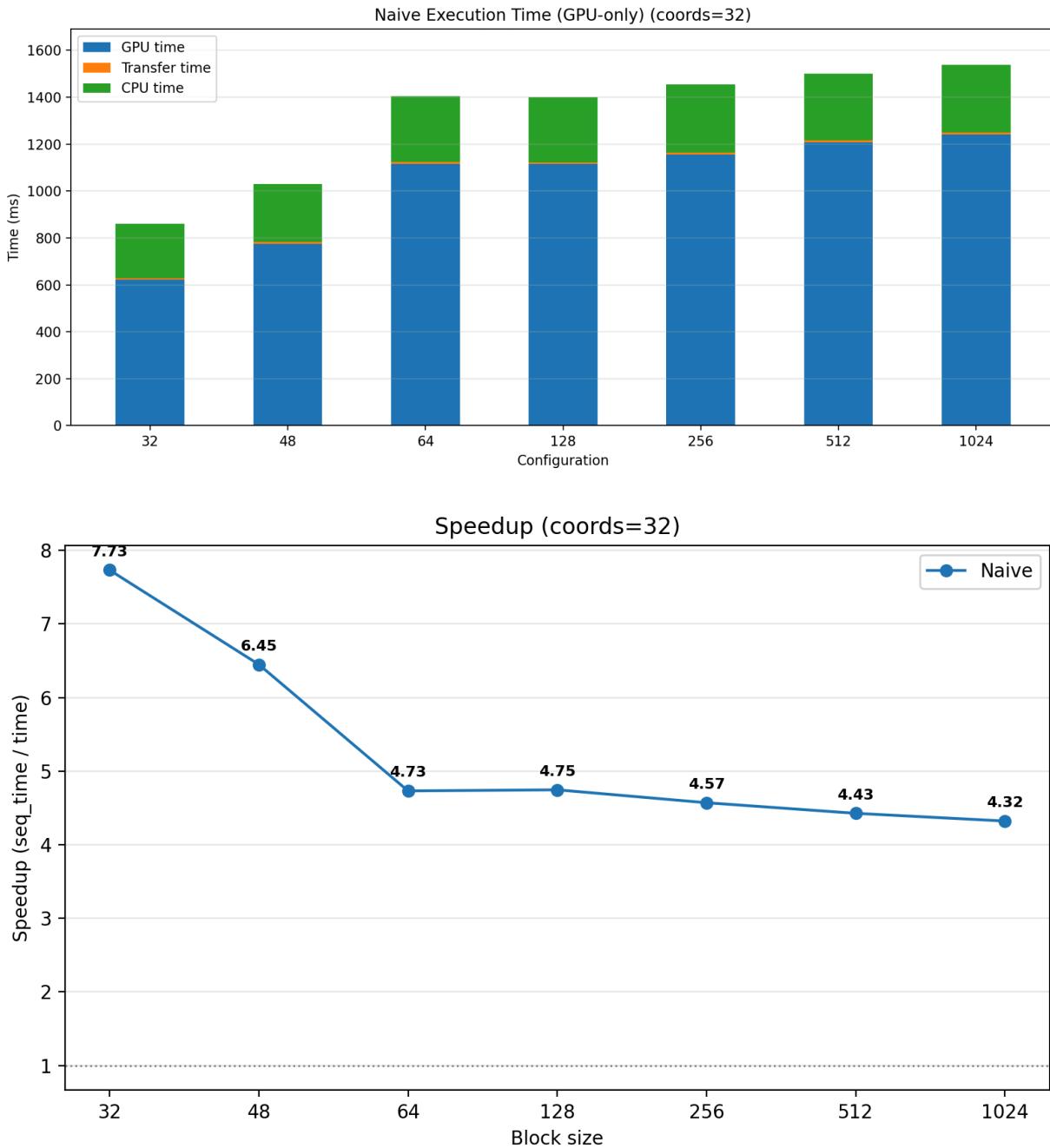
```
checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double),
cudaMemcpyDeviceToHost));
```

Έχοντας λοιπόν υλοποιήσει τη naïve προσέγγιση, θα πάρουμε μετρήσεις και για αυτή και για την σειριακή εκδοχή, ώστε να μπορέσουμε να συγκρίνουμε την επίδοση τους. Οι μετρήσεις θα πραγματοποιηθούν για το παρακάτω configuration:

$\{\text{Size, Coords, Clusters, Loops}\} = \{1024, 32, 64, 10\}$ και για block size= {32, 48, 64, 128, 238, 512, 1024}.

Τα διαγράμματα χρόνου εκτέλεσης (με και χωρίς τη σειριακή) καθώς και αυτά του αντίστοιχου speedup βρίσκονται παρακάτω:





Παρατηρούμε συνεπώς, ότι η παράλληλη έκδοση είναι σαφώς ταχύτερη από τη σειριακή, γιατί το πιο «βαρύ» κομμάτι του k-means (υπολογισμός αποστάσεων και επιλογή nearest cluster για κάθε object) παραλληλοποιείται στη GPU.

Ωστόσο, η συγκεκριμένη υλοποίηση δεν είναι ιδανική για GPU, επειδή δεν κρατάει όλη τη ροή στη συσκευή: σε κάθε iteration αντιγράφει αρκετά δεδομένα μεταξύ host και device, και κάνει την ενημέρωση των cluster centers στη CPU με πλήρη πέρασμα από όλα τα δεδομένα. Επιπλέον, η

ενημέρωση του devdelta με global atomicAdd εισάγει contention. Άρα, παρότι υπάρχει σημαντικό κέρδος, αυτό περιορίζεται από data transfers, CPU computation και συγχρονισμούς.

Επιπλέον, με βάση τα διαγράμματα, παρατηρούμε ότι η καλύτερη επίδοση επιτυγχάνεται για μικρά block sizes (για 32 δηλαδή), ενώ όσο το block size αυξάνεται, ο συνολικός χρόνος εκτέλεσης αυξάνει και το speedup μειώνεται σταδιακά.

Αυτό οφείλεται στο ότι στη συγκεκριμένη naivε υλοποίηση κάθε thread εκτελεί αρκετή δουλειά (υπολογισμό αποστάσεων για όλα τα clusters), ενώ τα δεδομένα προσπελαύνονται κυρίως από global memory, δημιουργώντας thrashing στον Memory Controller και στις caches όταν πολλά threads (από το ίδιο block) ζητάνε ταυτόχρονα δεδομένα. Με μεγαλύτερα block sizes αυξάνεται η πίεση σε πόρους (registers, shared memory, occupancy), με αποτέλεσμα λιγότερα ενεργά warps ανά SM και χαμηλότερη αξιοποίηση της GPU. Επιπλέον, δεν υπάρχει ουσιαστικό όφελος από μεγαλύτερα blocks, αφού δεν αξιοποιείται shared memory ούτε υπάρχει συνεργασία μεταξύ threads του ίδιου block.

Συνεπώς, μικρότερα block sizes οδηγούν σε καλύτερη απόδοση λόγω καλύτερου scheduling και μεγαλύτερου παραλληλισμού, ενώ μεγαλύτερα block sizes αυξάνουν το overhead χωρίς να βελτιώνουν τη μνήμη ή τον υπολογισμό. Αυτό επιβεβαιώνει ότι η συγκεκριμένη υλοποίηση δεν είναι βελτιστοποιημένη για την αρχιτεκτονική της GPU και δεν εκμεταλλεύεται αποτελεσματικά το block-level parallelism.

Ζητούμενο 2 - Transpose Version

Για να αντιμετωπιστεί το πρόβλημα της κακής χρήσης του εύρους ζώνης μνήμης της Naive έκδοσης, προχωρήσαμε στην Transpose έκδοση. Η βασική αλλαγή εδώ είναι η αναδιοργάνωση της δομής των δεδομένων στη μνήμη. Αντί να αποθηκεύουμε τα δεδομένα ανά αντικείμενο (Row-Major), τα αποθηκεύουμε ανά διάσταση (Column-Major). Δηλαδή, όλες οι συντεταγμένες 'x' όλων των αντικειμένων αποθηκεύονται συνεχόμενα, μετά όλες οι συντεταγμένες 'y' κ.ο.κ. Αυτή η αλλαγή έχει δραματική επίδραση στην απόδοση του προγράμματος. Πλέον, όταν το νήμα 0 διαβάζει την πρώτη συντεταγμένη του αντικειμένου 0 και το νήμα 1 διαβάζει την πρώτη συντεταγμένη του αντικειμένου 1, οι διευθύνσεις μνήμης που προσπελαύνουν είναι γειτονικές. Αυτό επιτρέπει στην GPU να ομαδοποιήσει αυτά τα αιτήματα σε μία και μοναδική συναλλαγή μνήμης.

Δεδομένου ότι η κάθε συναλλαγή γίνεται ανά warp(s) (warp = 32 νήματα), όλα τα νήματα αυτά θα λάβουν με μία πρόσβαση στην global memory όλους τους αριθμούς για να εκτελέσουν τις πράξεις που τους έχουν ανατεθεί. Συνεπώς, τα παράλληλα νήματα που εκτελούν την ίδια εντολή δεν θα σπαταλούν το εύρος ζώνης της GPU εκτελώντας περιπτές συναλλαγές με τη global memory της GPU, όπως γινόταν στη naivε εκδοχή του αλγορίθμου.

Παρακάτω εξηγούμε τις αλλαγές που εφαρμόσαμε στον πηγαίο κώδικα:

Σε πρώτη φάση, προσαρμόστηκε η συνάρτηση υπολογισμού της απόστασης, ώστε να υποστηρίζει τον πίνακα objects σε column-major μορφή. Η κύρια διαφορά με την naive έκδοση είναι:

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

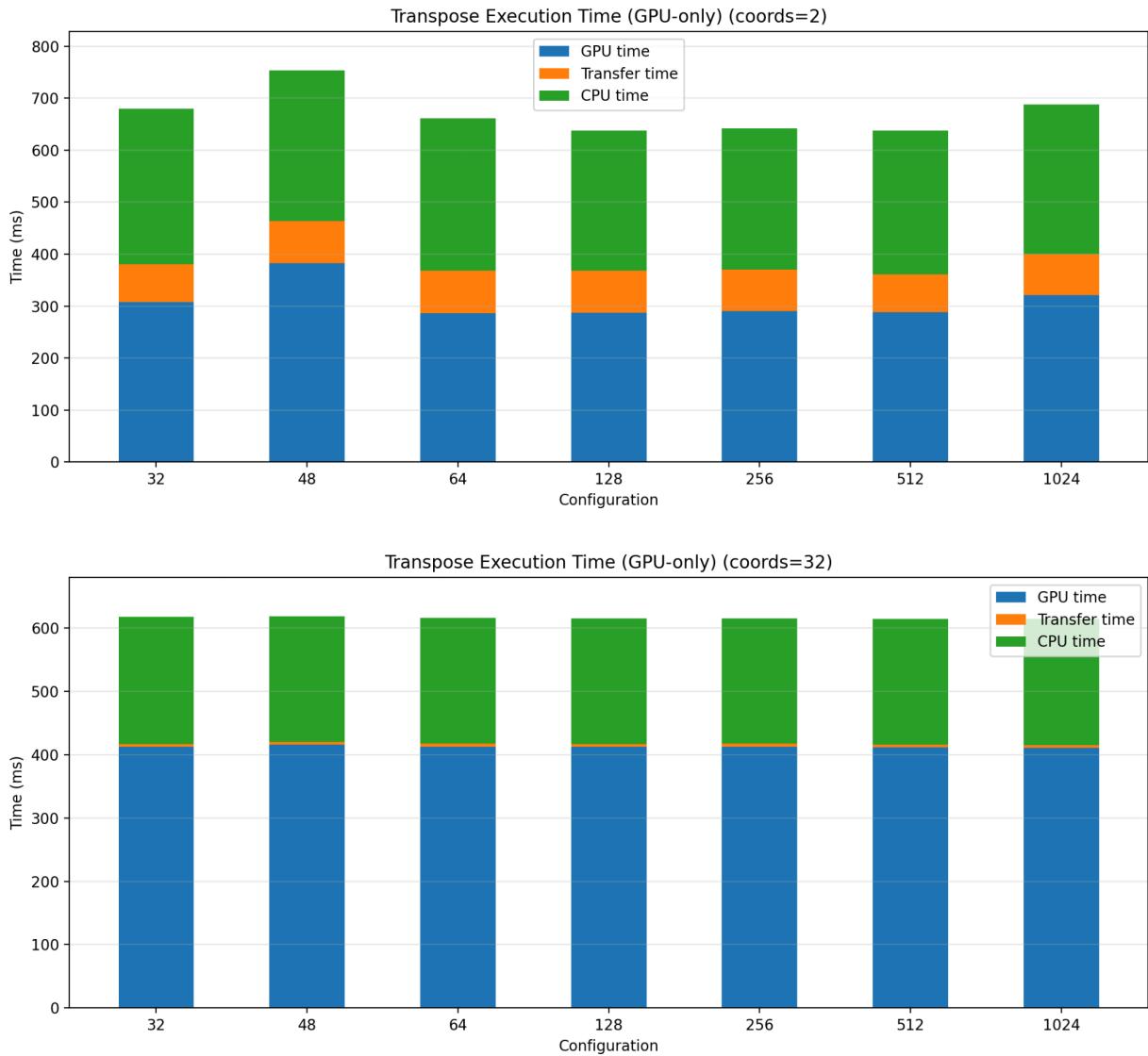
Παρατηρούμε ότι σε αυτό το σημείο οι πίνακες objects και clusters διατρέχονται διαφορετικά. Αυτό συμβαίνει γιατί στην τρέχουσα υλοποίηση, όλες οι i-οστές συντεταγμένες όλων των αντικειμένων του dataset αποθηκεύονται μαζί.

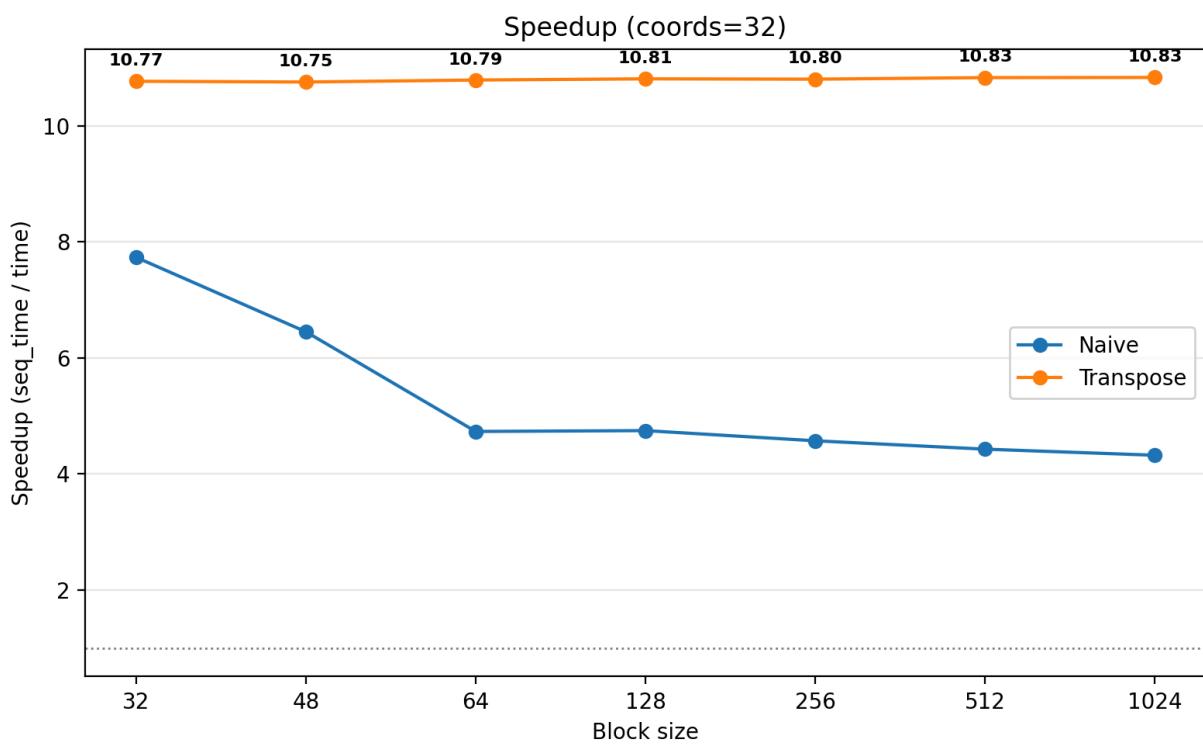
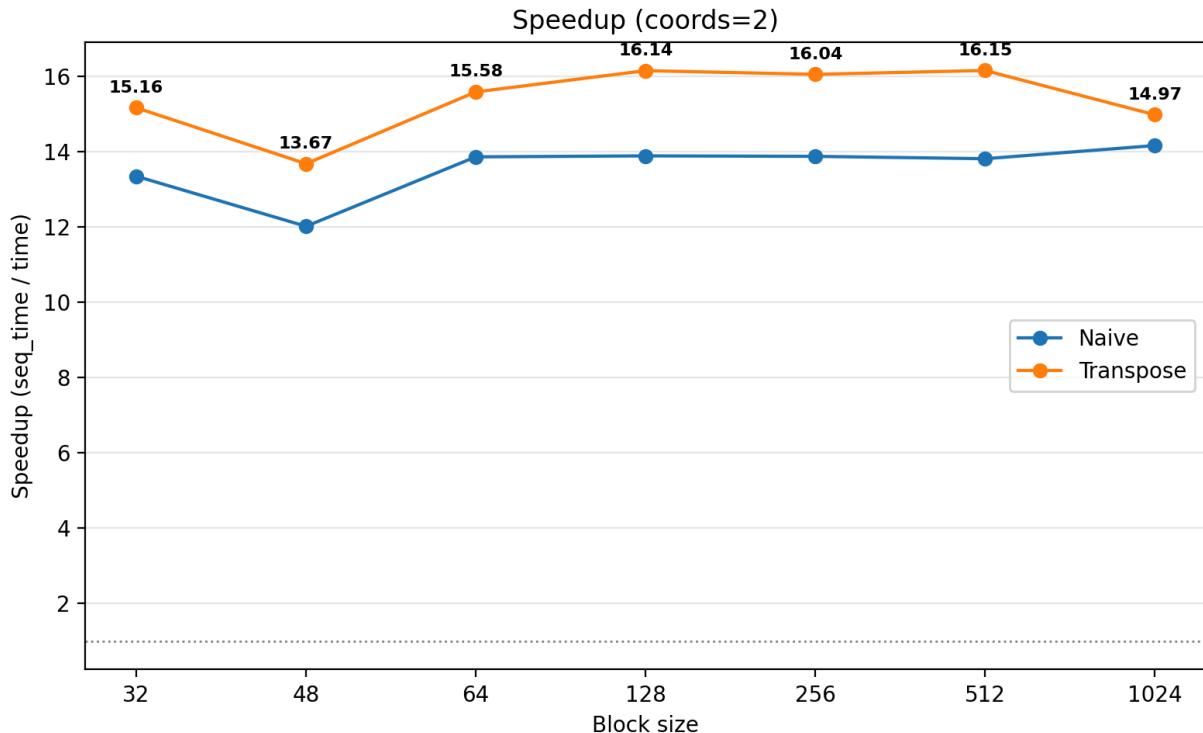
Στη συνέχεια, στη C συνάρτηση που τρέχει στον host (CPU), μετατρέπουμε κατάλληλα τον πίνακα objects προτού τον εισάγουμε στη GPU για υπολογισμούς:

```
for (i = 0; i < numObjs; i++) {
    for (j = 0; j < numCoords; j++) {
        dimObjects[j][i] = objects[i * numCoords + j];
    }
}

for (i = 0; i < numCoords; i++) {
    for (j = 0; j < numClusters; j++) {
        dimClusters[i][j] = dimObjects[i][j];
    }
}
```

Παρακάτω φαίνονται και τα αποτελέσματα των μετρήσεων που λάβαμε:





Τα αποτελέσματα επιβεβαιώνουν τη θεωρία, καθώς ο χρόνος εκτέλεσης μειώθηκε αρκετά, προσφέροντας επιτάχυνση 2 φορές σε σχέση με την Naive έκδοση. Αυτή η βελτίωση αποδεικνύει ότι ο αλγόριθμος ήταν περιορισμένος από την ταχύτητα ανάγνωσης της μνήμης (memory bandwidth bound).

Ζητούμενο 3: Shared Version

Στο επόμενο στάδιο βελτιστοποίησης αξιοποιήσαμε τη Shared Memory, τη χαμηλής καθυστέρησης on-chip-memory της GPU. Εξηγούμε γιατί η χρήση του τύπου αυτού μνήμης έχει τεράστια σημασία όχι μόνο για την υλοποίησή μας, αλλά γενικότερα στην εκτέλεση κώδικα σε GPU.

Η Shared Memory έχει τάξεις μεγέθους μικρότερο latency από την Global Memory και προσφέρει πολύ υψηλό bandwidth. Λειτουργεί ουσιαστικά ως μια "προγραμματιζόμενη cache", δηλαδή κρυφή, on-chip μνήμη ταχύτατης προσπέλασης που διαχειρίζεται ρητά ο προγραμματιστής, και είναι ορατή μόνο στα νήματα που ανήκουν στο ίδιο thread block. Συνεπώς, είναι πολύ χρήσιμη για την αποθήκευση δεδομένων που χρειάζονται από κοινού τα νήματα σε αρκετά σημεία της εκτέλεσης.

Στη δική μας περίπτωση, που ο στόχος μας είναι να βρούμε την απόσταση κάθε αντικειμένου από όλα τα κέντρα των clusters, ο πίνακας που αναπαριστά τα κέντρα αποτελεί κοινό παράγοντα για όλα τα νήματα και, επιπρόσθετα, τα περιεχόμενά του διαβάζονται επανειλημένα από κάθε νήμα, καθώς για κάθε cluster και για κάθε συντεταγμένη απαιτούνται προσπελάσεις $\approx numClusters \cdot numCoords$. Είναι λοιπόν περιττό να εκτελούμε transactions με την global memory σε τόσο μεγάλη συχνότητα, για δεδομένα τα οποία, όπως ανακαλύψαμε, είναι αρκούντως μικρά για να αποθηκευτούν στη shared memory. Άρα, η χρήση της κρίνεται αναγκαία για την επίτευξη περαιτέρω speedup.

Η φόρτωση δεδομένων στη shared memory γίνεται στην αρχή της εκτέλεσης του kernel. Τα threads του κάθε block συνεργάζονται για να μεταφέρουν τα κέντρα από τη global memory στη shared memory. Κάθε thread αναλαμβάνει να φορτώσει ένα μικρό κομμάτι των συνολικών δεδομένων, όπως φαίνεται στον παρακάτω κώδικα:

```

int i;
extern __shared__ double shmemClusters[];

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

/* Use this so all threads wait until copy is done */
__syncthreads();

```

Εκτός της αντιγραφής των κέντρων των clusters στη shared memory, πολύ σημαντική είναι και η χρήση της συνάρτησης `__syncthreads()`. Η μέθοδος αυτή είναι ένα είδος “φράγματος”, που απαγορεύει στα νήματα να προχωρήσουν προτού ολοκληρωθεί η αντιγραφή των κέντρων. Αν δεν χρησιμοποιούταν, τότε είναι πολύ πιθανό να καταλήγαμε σε λάθος αποτελέσματα (`undefined behavior`) αφού μερικά νήματα δεν θα “έβλεπαν” τα τελικά δεδομένα της shared memory, ενώ παράλληλα θα είχαν προχωρήσει ήδη σε υπολογισμούς αλλά με ελλιπή στοιχεία.

Ο έλεγχος για την επάρκεια χωρητικότητας της shared memory ώστε να αποθηκεύσουμε τα κέντρα των clusters γίνεται ως εξής:

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

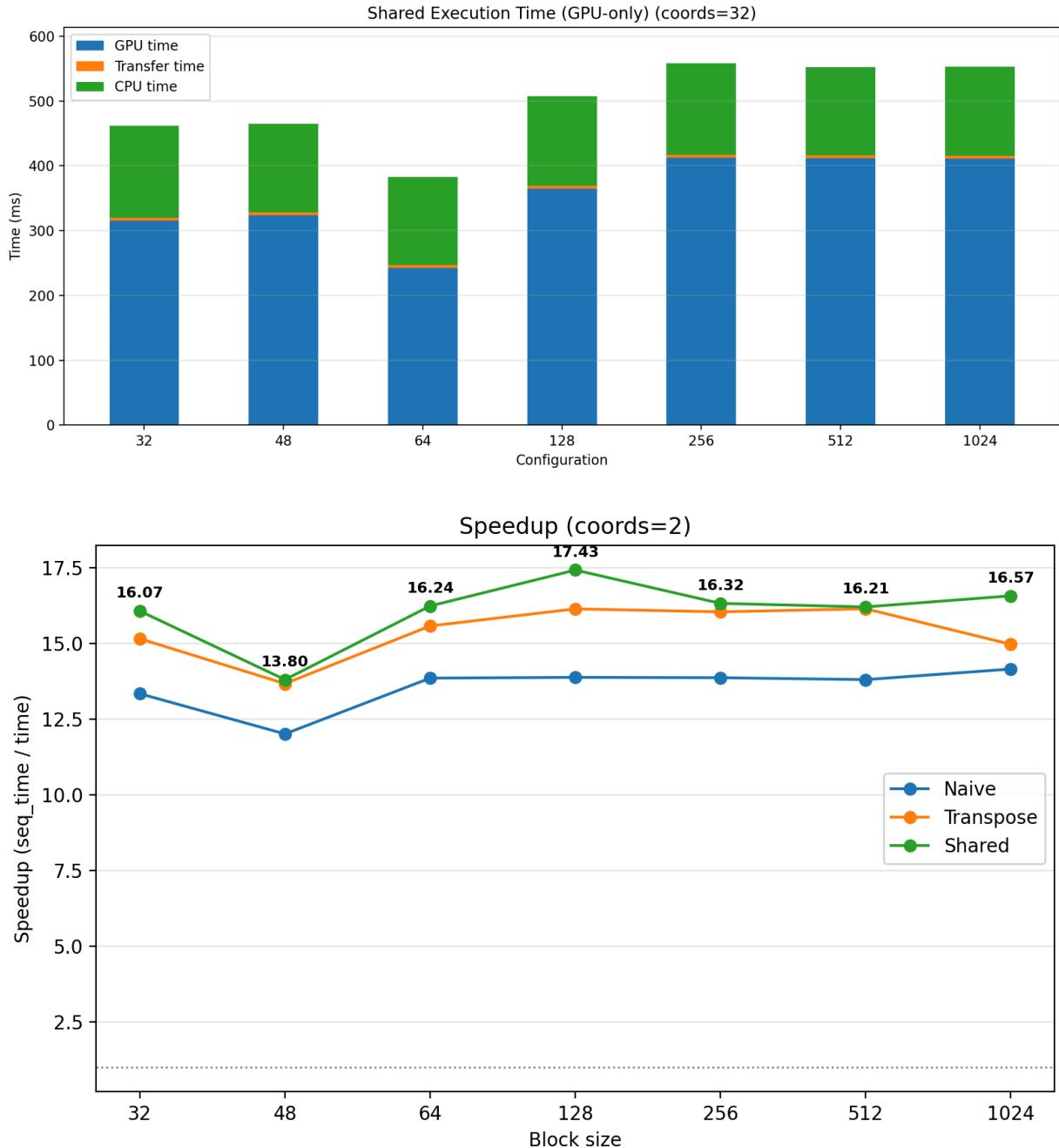
cudaDeviceProp deviceProp;
int deviceNum;
cudaGetDevice(&deviceNum);
cudaGetDeviceProperties(&deviceProp, deviceNum);

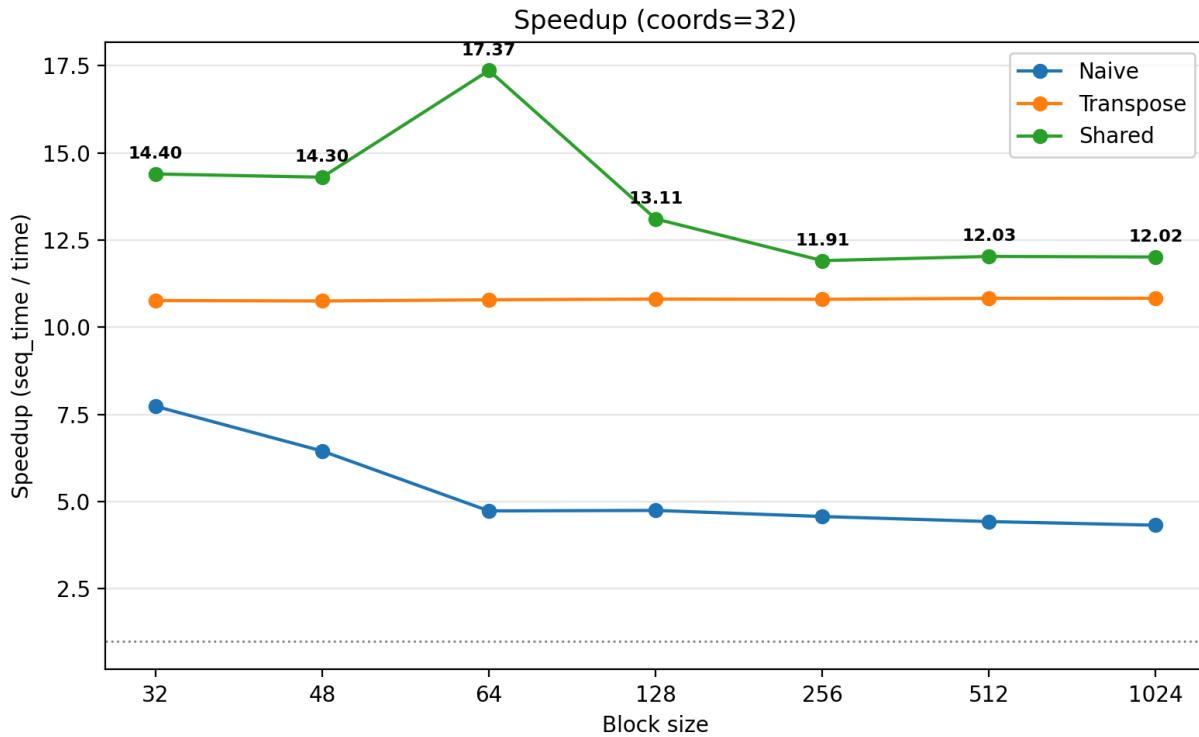
if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock) {
    error("Your CUDA hardware has insufficient block shared memory
to hold all cluster centroids\n");
}
```

Το υπόλοιπο της υλοποίησης δεν διαφοροποιείται από την transpose εκδοχή. Είναι επίσης αρκετά σημαντικό να τονιστεί ότι, από την εκφώνηση της άσκησης, μας δινόταν ότι τα clusters χωρούσαν στην shared memory και επομένως δεν απαιτήθηκε η εφαρμογή blocking/tiling για τα δεδομένα αυτά.

Παρακάτω φαίνονται τα διαγράμματα με τα αποτελέσματα των μετρήσεων των τειραμάτων που εκτελέσαμε, για τα ζητούμενα configurations:







Η επιτάχυνση προέρχεται από τη δραστική μείωση της κίνησης προς την κύρια μνήμη, αυξάνοντας την υπολογιστική ένταση (Operation Intensity) του αλγορίθμου. Σύμφωνα με το **Roofline Model**, ο αλγόριθμος K-means είναι εγγενώς memory-bound. Με τη χρήση Shared Memory, τα clusters διαβάζονται από την global memory μόνο μία φορά ανά block, αντί για μία φορά ανά thread. Αυτό μειώνει δραστικά τον όγκο των bytes που μεταφέρονται, μετακινώντας το "λειτουργικό σημείο" του πυρήνα δεξιά στην καμπύλη Roofline και απελευθερώνοντας πολύτιμο bandwidth για να διαβαστούν πιο γρήγορα τα μοναδικά δεδομένα του κάθε αντικειμένου.

Πιο συγκεκριμένα, για την περίπτωση των 32 συντεταγμένων, τα μικρότερα block sizes (32, 48, 64) παρουσιάζουν πιο εμφανές speedup σε σχέση με configurations με μεγαλύτερα block sizes. Η διαφοροποίηση αυτή οφείλεται κυρίως στον τρόπο με τον οποίο το block size επηρεάζει το occupancy και το latency hiding της GPU. Block sizes που είναι πολλαπλάσια του μεγέθους του warp (32) επιτρέπουν καλύτερη αξιοποίηση των execution units, ενώ πολύ μεγάλα block sizes μπορούν να περιορίσουν το occupancy (λόγω εξάντλησης shared memory ή registers), μειώνοντας την ικανότητα της GPU να "κρύβει" τις καθυστερήσεις πρόσβασης στη μνήμη.

Το μέγιστο speedup εμφανίζεται για block size = 64. Το συγκεκριμένο configuration αποτελεί το "sweet spot" για το πείραμα, διότι παρέχει επαρκή αριθμό threads ανά block για αποδοτική εκτέλεση αλλά και μείωση του κλάσματος του latency ως προς τον συνολικό χρόνο. Διατηρεί επίσης το occupancy ψηλά ώστε να μεγιστοποιείται το memory throughput που μπορεί να επιτύχει η GPU.

Από την άλλη πλευρά, για το configuration όπου coords = 2, το όφελος από τη χρήση της shared memory είναι περιορισμένο. Παρόλο που ο αλγόριθμος παραμένει bandwidth-bound, το

Operation Intensity δεν βελτιώνεται ουσιαστικά. Το κόστος των προσπελάσεων στα centroids δεν αποτελεί το κυρίαρχο bottleneck, καθώς τα δεδομένα αυτά είναι μικρά και εξυπηρετούνται ήδη αποτελεσματικά από τις L1/L2 caches της GPU. Κατά συνέπεια, η μείωση των προσπελάσεων στη global memory μέσω της shared memory δεν οδηγεί σε αισθητή βελτίωση σε σχέση με την transpose έκδοση. Αντίθετα, για coords = 32, οι επαναλαμβανόμενες προσπελάσεις στα centroids κυριαρχούν στο memory traffic, και η διαχείρισή τους μέσω shared memory μετατρέπει τη διαδικασία σε πιο αποδοτική, επιτρέποντας στον πυρήνα να εκμεταλλευτεί καλύτερα το διαθέσιμο peak performance.

Bottleneck Analysis

Για όλα τα παραπάνω πειράματα, προκύπτει ότι το iterative μέρος δεν περιορίζεται μόνο από έναν παράγοντα, αλλά από διαφορετικό bottleneck ανά configuration.

Για **coords = 2** (Shared και Transpose πειράματα), ο συνολικός χρόνος ανά επανάληψη μοιράζεται κυρίως μεταξύ GPU χρόνου και CPU χρόνου, ενώ τα CPU to GPU transfers αποτελούν μη αμελητέο αλλά δευτερεύον κόστος. Συγκεκριμένα, το πρωτοκαλί χωρίο που αποτυπώνει τις μεταφορές αυτές παραμένει σταθερά ορατό σε όλα τα block sizes, καθώς οι συντεταγμένες είναι λίγες, άρα πέφτει το computational complexity, άρα γίνεται περισσότερο εμφανής ο χρόνος απασχόλησης του διαύλου επικοινωνίας host-device. Παράλληλα, ο CPU χρόνος παραμένει σημαντικός επειδή το update των centroids γίνεται σειριακά στον host. Επομένως, για coords=2 το bottleneck είναι μικτό, με πρόσθετο overhead από host-device transfers που περιορίζει το τελικό speedup.

Για **coords = 32**, η εικόνα αλλάζει: στο transpose version ο GPU χρόνος κυριαρχεί σχεδόν πλήρως (οι μεταφορές είναι πρακτικά αμελητέες στο σύνολο), ενώ στη Shared παρατηρείται ότι το GPU time μειώνεται αισθητά για συγκεκριμένα block sizes (ιδίως στο 64), γεγονός που δείχνει ότι το bottleneck βρίσκεται κυρίως στην on-device συμπεριφορά μνήμης/υπολογισμού και ότι η caching των centroids στη shared μπορεί να μειώσει το global memory traffic όταν αυτό είναι κυρίαρχο. Σε κάθε περίπτωση, για coords=32 το iterative loop είναι **device-bound** (κυρίως GPU-bound), με τον CPU χρόνο να παραμένει δεύτερος αλλά όχι καθοριστικός, και με τα transfers να μην αποτελούν πλέον τον κύριο περιορισμό.

Zητούμενο 4: All-GPU Version

Μέχρι το σημείο αυτό, ο έλεγχος της σύγκλισης του αλγορίθμου k-means παρέμενε στην CPU. Συγκεκριμένα, σε κάθε επανάληψή του, η CPU εκκινούσε τον πυρήνα της GPU για την ανάθεση των αντικειμένων σε clusters, ανέμενε την ολοκλήρωση του kernel, υπολόγιζε τα νέα κέντρα και στη συνέχεια τα μετέφερε εκ νέου στη GPU. Η προσέγγιση αυτή εισήγαγε ένα σημαντικό σημείο συμφόρησης, καθώς κάθε επανάληψη απαιτούσε πολλαπλές μεταφορές δεδομένων μέσω του διαύλου PCIe, ο οποίος παρουσιάζει πολύ υψηλότερη καθυστέρηση και χαμηλότερο εύρος ζώνης σε σχέση με την επικοινωνία εσωτερικά της GPU.

Στην έκδοση **All-GPU**, ολόκληρη η διαδικασία ενημέρωσης των κέντρων μεταφέρθηκε στην GPU. Η CPU περιορίζεται πλέον αποκλειστικά στον αρχικό συγχρονισμό και στην εκκίνηση της επανάληψης, χωρίς να εμπλέκεται στον υπολογισμό των κέντρων ή στον έλεγχο σύγκλισης.

Παραθέτουμε τις αλλαγές που υπέστη ο κώδικας του host, σε σχέση με τη shared εκδοχή, για να υποστηρίξει full-offload του kmeans στην GPU:

```
// ALL-GPU VERSION (Host Code)
do {
    // 0. Reset accumulators on GPU
    checkCuda(cudaMemset(devicenewClusterSize, 0, ...));
    checkCuda(cudaMemset(devicenewClusters, 0, ...));

    // 1. Launch Assignment Kernel (No Memcpy needed!)
    find_nearest_cluster<<<...>>>(...);

    // 2. Launch UPDATE Kernel on GPU (New!)
    update_centroids<<<...>>>(...);

    // 3. Launch AVERAGE Kernel on GPU (New!)
    average_centroids<<<...>>>(...);

    // 4. Copy ONLY delta back (CHEAP!)
    checkCuda(cudaMemcpy(&delta, dev_delta_ptr, ... DeviceToHost));

    // CPU does nothing but check delta
} while (delta > threshold && loop < loop_threshold);
```

Συγκεκριμένα, αφαιρέσαμε όλες τις εντολές τύπου:

```
checkCuda(cudaMemcpy(deviceClusters, dimClusters[0], ...HostToDevice));
checkCuda(cudaMemcpy(membership,...DeviceToHost));
```

που αφορούν στην αντιγραφή δεδομένων από τη CPU στη GPU, και αντιστρόφως. Οι αντιγραφές αυτές ήταν ιδιαίτερα κοστοβόρες αφού θα έπρεπε να γίνεται συνεχής ανταλλαγή δεδομένων τόσο εντός της GPU, όσο και μεταξύ του PCIe bus. Η μεταφορά πινάκων τέτοιου μεγέθους προσέδιδε τεράστιες καθυστερήσεις στα προγράμματά μας, καθιστώντας τα λιγότερο αποδοτικά.

Η υλοποίηση ανασχεδιάστηκε σε τρεις διακριτούς πυρήνες που εκτελούνται σειριακά σε κάθε επανάληψη του kmeans, κρατώντας όλα τα δεδομένα (objects, membership, newClusters)

μόνιμα στην VRAM. Η CPU περιορίζεται πλέον σε ρόλο ελεγκτή, εκκινώντας τους kernels και διαβάζοντας μόνο μία τιμή (το delta) για τον έλεγχο τερματισμού.

Συγκεκριμένα η ροή εκτέλεσης είναι:

- Kernel 1 (find_nearest_cluster): Κάθε νήμα αναλαμβάνει ένα αντικείμενο, υπολογίζει την απόστασή του από όλα τα κέντρα (τα οποία βρίσκονται cached στη Shared Memory για ταχύτατη προσπέλαση) και βρίσκει το πλησιέστερο. Το νήμα γράφει το index του νέου cluster στον πίνακα deviceMembership και στη συνέχεια αποδεσμεύεται. Επιλέξαμε αυτού του είδους την υλοποίηση για καθαρά λόγους σαφήνειας, καθώς έτσι διατηρούμε το kernel "καθαρό" και εστιασμένο μόνο στους υπολογισμούς αποστάσεων. Παραθέτουμε και την υλοποίησή μας:

```
// --- ---
/* Runs in parallel on the GPU to find the nearest cluster for each
object */
/* NO REDUCTION HERE - JUST ASSIGNMENT */
__global__ static
void find_nearest_cluster(int numCoords,
    int numObjs,
    int numClusters,
    double *objects,           // [numCoords][numObjs]
    double *deviceClusters,    // [numCoords][numClusters]
    int *deviceMembership,     // [numObjs] ** to be
calculated in this kernel*
    double *devdelta)
{
    extern __shared__ double shmemClusters[];

    /* Copy deviceClusters to shmemClusters so they can be accessed
faster. */
    int i;
    for (i = threadIdx.x; i < numClusters * numCoords; i +=
blockDim.x) {
        shmemClusters[i] = deviceClusters[i];
    }
    /* Use this so all threads wait until copy is done */
    __syncthreads();

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

```

/* Run only threads that are tied to a kmeans object */
if (tid < numObjs) {
    int index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* Let cluster 0 be the initial minimum distance */
    min_dist = euclid_dist_2_transpose(numCoords, numObjs,
numClusters, objects, shmemClusters, tid, 0);

    /* iterate over all clusters to find the nearest */
    for (i = 1; i < numClusters; i++) {
        dist = euclid_dist_2_transpose(numCoords, numObjs,
numClusters, objects, shmemClusters, tid, i);
        /* no need square root */
        if (dist < min_dist) { /* find the min and its array index
*/
            min_dist = dist;
            index = i;
        }
    }
    if (deviceMembership[tid] != index) {
        atomicAdd(devdelta, 1.0);
    }

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

    // NO REDUCTION (ATOMIC ADDs) HERE!
}
}

```

- Kernel 2 (update_centroids): Ένας νέος πυρήνας που αναλαμβάνει τον υπολογισμό των νέων κέντρων με βάση τα νέα memberships. Κάθε νήμα διαβάζει το membership που υπολογίστηκε προηγουμένως και χρησιμοποιεί atomic operations (atomicAdd) για να προσθέσει τις συντεταγμένες του αντικειμένου του στα καθολικά αθροίσματα

(devicenewClusters) στην Global Memory. Είναι αξιοσημείωτο ότι, παρόλο που τα atomics στην Global Memory έχουν κόστος λόγω των συγκρούσεων, αυτό είναι αμελητέο μπροστά στο κόστος της μεταφοράς των δεδομένων στον CPU, όπως θα αποδειχθεί και παρακάτω με τα αποτελέσματα των μετρήσεων.

```
/*
 * Separate kernel to update centroids.
 * Since we didn't do reduction in the previous step, we must iterate
 over ALL objects here
 * to calculate the new centers. This is likely very slow on the GPU
 without optimization,
 * but this is the "No Reduction" baseline.
 *
 */
__global__ static
void update_centroids(int numCoords,
                      int numObjs,
                      int numClusters,
                      double *objects,           // [numCoords][numObjs]
                      int *membership,          // [numObjs]
                      int *devicenewClusterSize, // [numClusters]
                      double *devicenewClusters) // [numCoords][numClusters]
{
    int tid = get_tid();

    // Each thread takes an object and adds it to the appropriate new
    cluster
    // effectively performing the reduction here instead of the
    previous kernel.
    if (tid < numObjs) {
        int index = membership[tid];

        atomicAdd(&devicenewClusterSize[index], 1);
        for(int i=0; i < numCoords; ++i) {
            atomicAdd(&devicenewClusters[i * numClusters + index],
                      objects[i*numObjs + tid]);
        }
    }
}
```

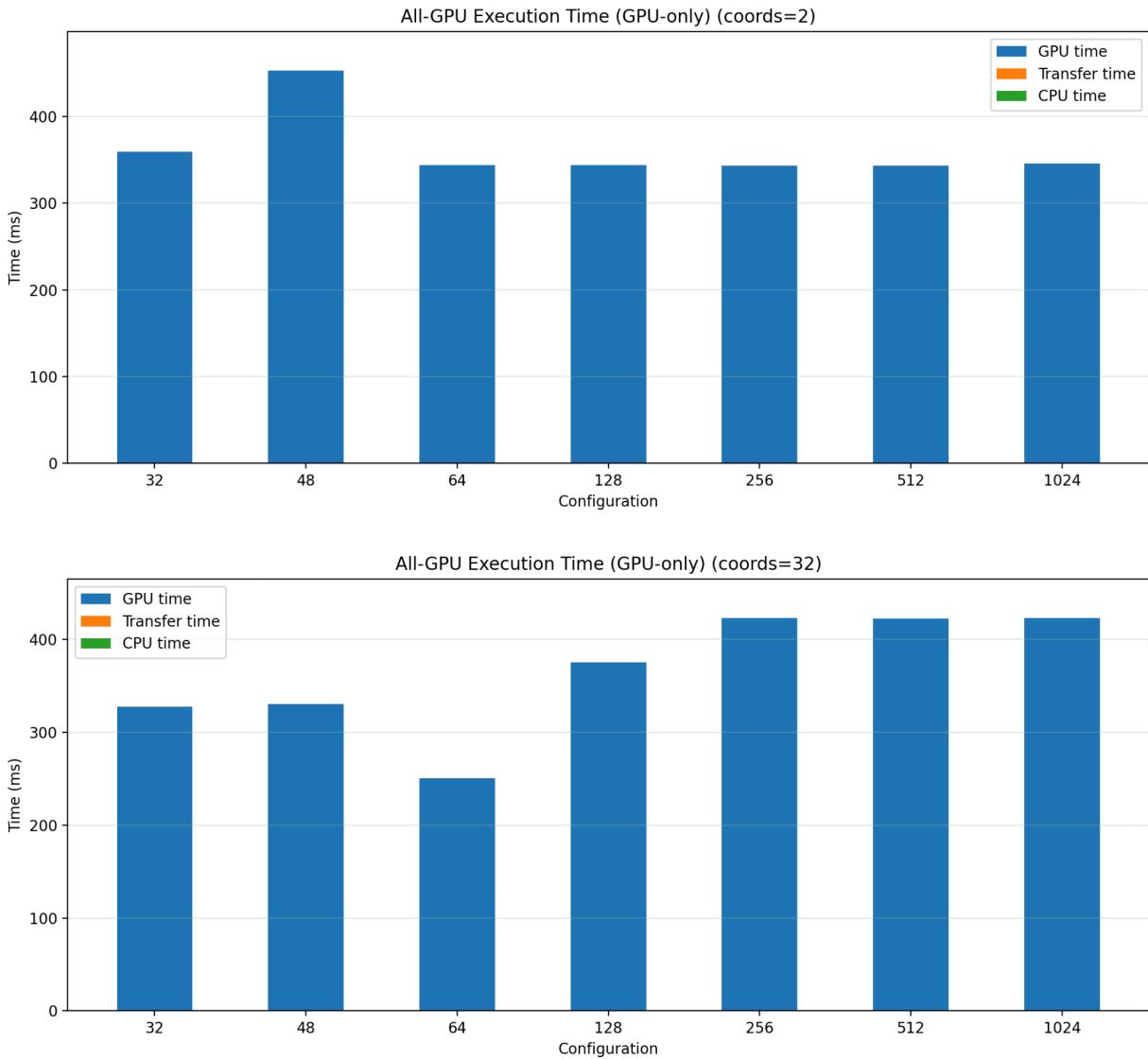
- Kernel 3 (average_centroids): Ένας lightweight πυρήνας που διαιρεί τα αθροίσματα με το πλήθος των μελών κάθε cluster για να βρει τους τελικούς μέσους όρους.

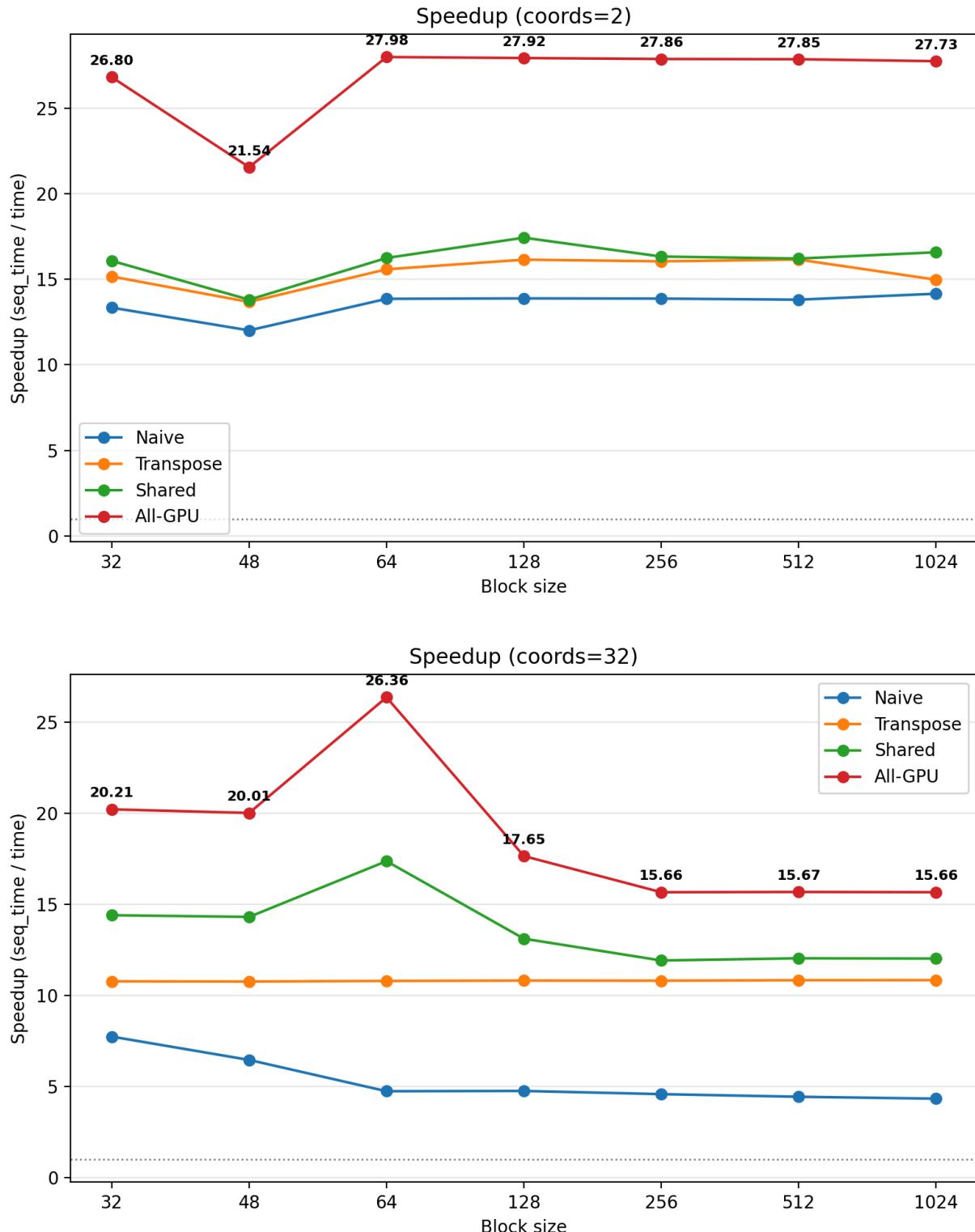
```
__global__ static
void average_centroids(int numCoords,
    int numClusters,
    int *devicenewClusterSize,           // [numClusters]
    double *devicenewClusters,          //
[numCoords][numClusters]
    double *deviceClusters)           // [numCoords][numClusters]
{
    int tid = get_tid();
    int total = numCoords * numClusters;

    if (tid < total) {
        int coord = tid / numClusters;
        int cluster = tid % numClusters;
        int count = devicenewClusterSize[cluster];

        if (count > 0) {
            deviceClusters[coord * numClusters + cluster] =
                devicenewClusters[coord * numClusters + cluster] /
count;
        }
    }
}
```

Τα διαγράμματα που προέκυψαν από τις μετρήσεις μας είναι τα εξής:





Σχολιασμός-Παρατηρήσεις:

- Στο configuration {1024, 2, 64, 10}, όπου κάθε αντικείμενο περιγράφεται από μόλις δύο συντεταγμένες, η all-GPU έκδοση παρουσιάζει εντυπωσιακή βελτίωση της επίδοσης, με speedup που προσεγγίζει και σταθεροποιείται γύρω στο 27-28x για block sizes μεγαλύτερα ή ίσα του 64. Αντίθετως, οι naive, transpose και shared εκδόσεις εμφανίζουν σαφώς χαμηλότερα speedup, της τάξης των 13-14x για τη naive και 15-17x για τις transpose και shared. Παρότι οι τελευταίες δύο εκδόσεις βελτιώνουν σημαντικά την τοπικότητα πρόσβασης στη μνήμη και μειώνουν το κόστος ανά kernel μέσω coalesced accesses και caching των clusters στη shared memory, καμία δεν κατορθώνει να ξεπεράσει ένα συγκεκριμένο όριο επίδοσης. Η συμπεριφορά αυτή εξηγείται από το γεγονός ότι, σε κάθε επανάληψη του αλγορίθμου, απαιτείται μεταφορά του πίνακα membership από τη GPU στον CPU για τον υπολογισμό των νέων κέντρων των clusters και στη συνέχεια μεταφορά των ενημερωμένων κέντρων πίσω στη GPU. Το κόστος αυτών των μεταφορών μέσω του διαύλου PCIe είναι ιδιαίτερα υψηλό σε σχέση με τον χρόνο εκτέλεσης των kernels και λειτουργεί ως κυρίαρχο bottleneck που περιορίζει το συνολικό speedup ανεξαρτήτως του πόσο αποδοτικός είναι ο υπολογισμός στην GPU.
- Η παραπάνω παρατήρηση ερμηνεύεται άμεσα μέσω του νόμου του Amdahl. Ακόμη και αν το τμήμα του κώδικα που εκτελείται στην GPU επιταχυνθεί σημαντικά, το σειριακό ή μη επιταχυνόμενο τμήμα, το οποίο στην προκειμένη περίπτωση αντιστοιχεί στις μεταφορές δεδομένων CPU-GPU και στον υπολογισμό των νέων clusters στον CPU, θέτει ένα άνω φράγμα στο συνολικό speedup. Στις naive, transpose και shared εκδόσεις, το ποσοστό του χρόνου που καταναλώνεται σε αυτές τις μεταφορές παραμένει σημαντικό, με αποτέλεσμα το συνολικό speedup να κορέννυται σχετικά νωρίς. Η all-GPU έκδοση, αντίθετα, εξαλείφει σχεδόν πλήρως αυτό το σειριακό τμήμα, διατηρώντας όλα τα δεδομένα (objects, membership και clusters) μόνιμα στη VRAM και περιορίζοντας τον ρόλο του CPU σε έναν ελαφρύ έλεγχο σύγκλισης μέσω της ανάγνωσης μιας μόνο μεταβλητής (delta).
- Παρόμοια συμπεράσματα προκύπτουν και από την ανάλυση με βάση το Roofline model. Στο configuration με coords = 2, ο αλγόριθμος χαρακτηρίζεται από χαμηλή υπολογιστική ένταση καθώς για κάθε αντικείμενο εκτελούνται λίγες πράξεις ανά byte δεδομένων που προσπελαύνεται. Σε αυτή την περίπτωση, η απόδοση περιορίζεται κυρίως από το memory bandwidth και, στις παλιότερες εκδόσεις, ακόμη περισσότερο από το bandwidth και το latency του PCIe. Αν και το update_centroids kernel παραμένει memory-bandwidth-bound, το θεωρητικό “ταβάνι” του Roofline model ανεβαίνει σημαντικά, καθώς η πρόσβαση στη μνήμη γίνεται πλέον μέσω της πολύ ταχύτερης global και L2 μνήμης της GPU και όχι μέσω του διαύλου PCIe.
- Η εικόνα διαφοροποιείται στο δεύτερο configuration {1024, 32, 64, 10}, όπου κάθε αντικείμενο περιγράφεται από 32 συντεταγμένες. Σε αυτή την περίπτωση, παρατηρείται ότι το speedup της all-GPU έκδοσης μειώνεται, με μέγιστη επίδοση για block size γύρω στα 64 threads και σαφή πτώση για μεγαλύτερα block sizes. Παρότι η all-GPU παραμένει η ταχύτερη υλοποίηση, το πλεονέκτημά της έναντι των shared και transpose εκδόσεων περιορίζεται αισθητά. Η βασική αιτία είναι ότι η αύξηση του αριθμού των

συντεταγμένων αυξάνει σημαντικά το υπολογιστικό φορτίο ανά αντικείμενο, δηλαδή τον αριθμό των πράξεων που απαιτούνται για τον υπολογισμό των αποστάσεων. Ως αποτέλεσμα, ο αλγόριθμος αποκτά υψηλότερη υπολογιστική ένταση και μετακινείται, σύμφωνα με το Roofline model, προς μια περισσότερο compute-bound περιοχή.

- Η επίδραση του block size παρατηρείται και στα δύο configuration, περισσότερο όμως όταν coords=32. Η πτώση της επίδοσης για block size ίσο με 48 threads οφείλεται στο γεγονός ότι το μέγεθος αυτό δεν είναι πολλαπλάσιο του warp size των 32 threads, οδηγώντας σε κακή χρήση των warps και αυξημένο control overhead. Για block sizes ίσα ή μεγαλύτερα των 64 threads, η GPU φαίνεται να φτάνει σε κατάσταση κορεσμού των διαθέσιμων SMs, με αποτέλεσμα η επίδοση να σταθεροποιείται.

Καταληκτικά, η all-GPU υλοποίηση μετατοπίζει τα bottleneck σύμφωνα τόσο με τον νόμο του Amdahl όσο και με το Roofline model, επιτρέποντας σημαντικά υψηλότερο speedup.

Ζητούμενο 5: Reduction

Ο στόχος των βελτιστοποιήσεων με χρήση τεχνικών reduction είναι να εξαλείψουν τα ακριβά atomic operations στην Global Memory. Όταν πολλά threads προσπαθούν να ενημερώσουν ταυτόχρονα την ίδια μνήμη (π.χ. dev_delta ή τα centroids), δημιουργείται συμφόρηση, καθώς οι εντολές ατομικής ενημέρωσης σειριοποιούνται από το hardware της GPU, με αποτέλεσμα να εισάγεται καθυστέρηση στα προγράμματά μας.

Ακολουθήσαμε τρεις στρατηγικές για την υλοποίηση reduction σε διάφορα σημεία:

Partial centroid reduction: Η ιδέα μας ήταν αντί κάθε thread να γράφει κατευθείαν στην Global Memory, τα threads ενός block να συνεργάζονται για τον υπολογισμό των αθροισμάτων των centroids. Τα μερικά αθροίσματα υπολογίζονται στη shared memory του block, χρησιμοποιώντας atomics σε αυτή που είναι πολύ γρηγορότερα. Στο τέλος, μόνο ένα ή λίγα threads ανά block προσθέτουν αυτά τα μερικά αθροίσματα στον τελικό global πίνακα. Παρακάτω φαίνεται η κύρια ιδέα σε κώδικα C:

```

if (use_shmem_partials) {
    atomicAdd(&shmemNewClusterSize[index], 1);
    for (i = 0; i < numCoords; i++) {
        atomicAdd(&shmemNewClusters[i * numClusters + index], objects[i *
numObjs + tid]);
    }
}
/* ... */
if (use_shmem_partials) {
    __syncthreads();
}

```

```

for (i = threadIdx.x; i < numClusters; i += blockDim.x) {
    atomicAdd(&devicenewClusterSize[i], shmemNewClusterSize[i]);
}
for (i = threadIdx.x; i < numClusters * numCoords; i += blockDim.x) {
    atomicAdd(&devicenewClusters[i], shmemNewClusters[i]);
}
}
}

```

Delta reduction: Η προσέγγιση αυτή στοχεύει στη βελτιστοποίηση εγγραφής πάνω στη μεταβλητή delta (πόσα αντικείμενα άλλαξαν cluster), η οποία ενημερώνεται από κάθε thread που αλλάζει cluster. Αντί κάθε thread να τρέχει `atomicAdd(devdelta, 1.0)`, κάτι που δημιουργεί πολύ υψηλό contention, χρησιμοποιούμε δενδρική αναγωγή (tree-based reduction) εντός του block στη Shared Memory. Στο τέλος, μόνο το thread 0 κάθε block κάνει `atomicAdd` στην global memory.

```

/* Each thread writes its own delta in shared memory */
delta_reduce_buff[threadIdx.x] = local_delta;
__syncthreads();

/* Tree-based reduction in shmem */
for (int offset = blockDim.x / 2; offset > 0; offset >>= 1) {
    if (threadIdx.x < offset) {
        delta_reduce_buff[threadIdx.x] += delta_reduce_buff[threadIdx.x +
offset];
    }
    __syncthreads();
}
/* only thread 0 updates global memory */
if (threadIdx.x == 0) {
    atomicAdd(devdelta, delta_reduce_buff[0]);
}

```

All reduction: Στην προσέγγιση αυτή, κάθε block γράφει τα δικά του αποτελέσματα (μερικά αθροίσματα centroids, μερικό delta) σε δικό του ξεχωριστό χώρο στην Global Memory (devicenewClustersBlocks, devdeltaBlocks). Δεν υπάρχει race condition, άρα δεν χρειάζονται atomics.

```

/* --- Phase 1: Each block writes its own partial sums to Global Memory (No
Global Atomics) --- */

```

```

/* Wait for all threads in block to update shared memory counts */
__syncthreads();

/* Calculate base index for this block's data in the global per-block array
*/
/* devicenewClustersBlocks has size: [numBlocks][numCoords][numClusters] */
int block_offset = blockIdx.x * (numClusters * numCoords);

/* Transfers from Shared Memory to Global Memory */
/* Each thread writes a portion of the shared memory buffer to the block's
dedicated global slot */
for (i = threadIdx.x; i < numClusters * numCoords; i += blockDim.x) {
    devicenewClustersBlocks[block_offset + i] = shmemClusters[i];
}

/* Same logic for cluster sizes per block */
for (i = threadIdx.x; i < numClusters; i += blockDim.x) {
    devicenewClusterSizeBlocks[blockIdx.x * numClusters + i] =
shmemClusterSizes[i];
}

```

Στη συνέχεια, ένας δεύτερος πυρήνας αναλαμβάνει να εκτέλεσει το reduction, δηλαδή να αθροίσει τα αποτελέσματα όλων των blocks.

```

/* --- Phase 2: Separate Kernel to reduce block results --- */

template <typename T>
__global__ static
void reduce_blocks(const T *block_data, T *out, int numBlocks, int
elemsPerBlock) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= elemsPerBlock) {
        return;
    }

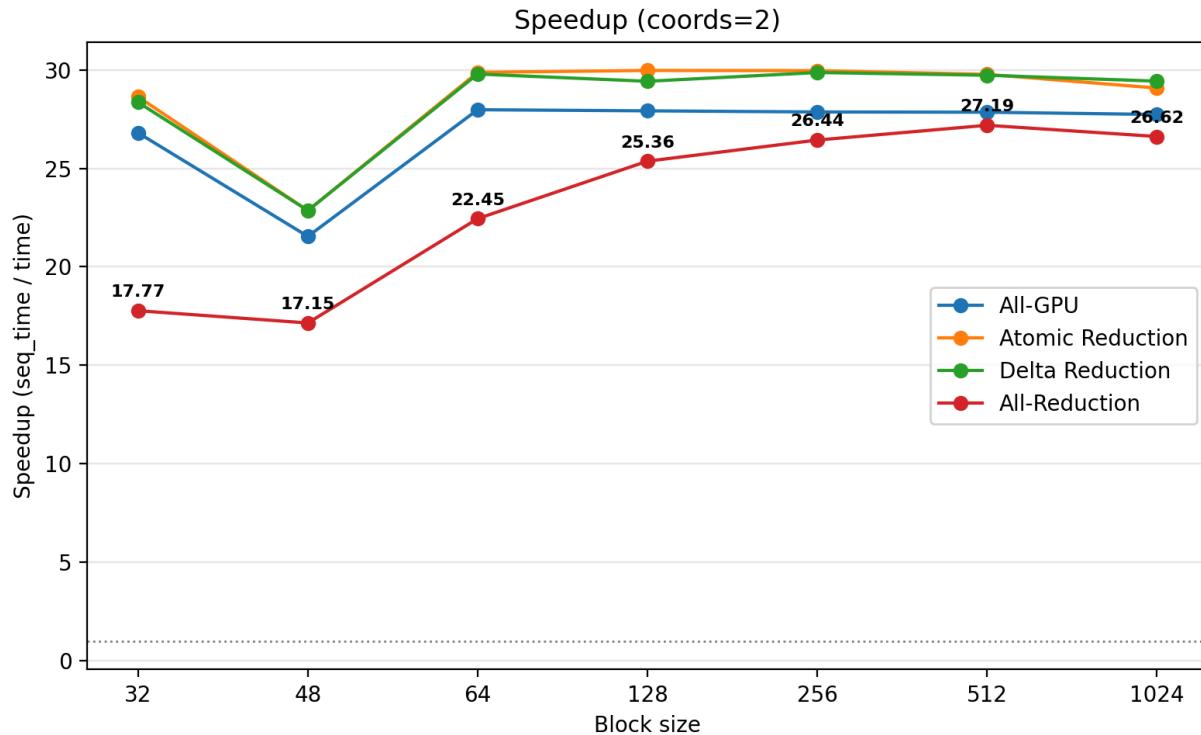
    T sum = 0;
    /* Iterate over all blocks to sum their contributions */
    for (int b = 0; b < numBlocks; ++b) {
        sum += block_data[b * elemsPerBlock + idx];
    }
}
```

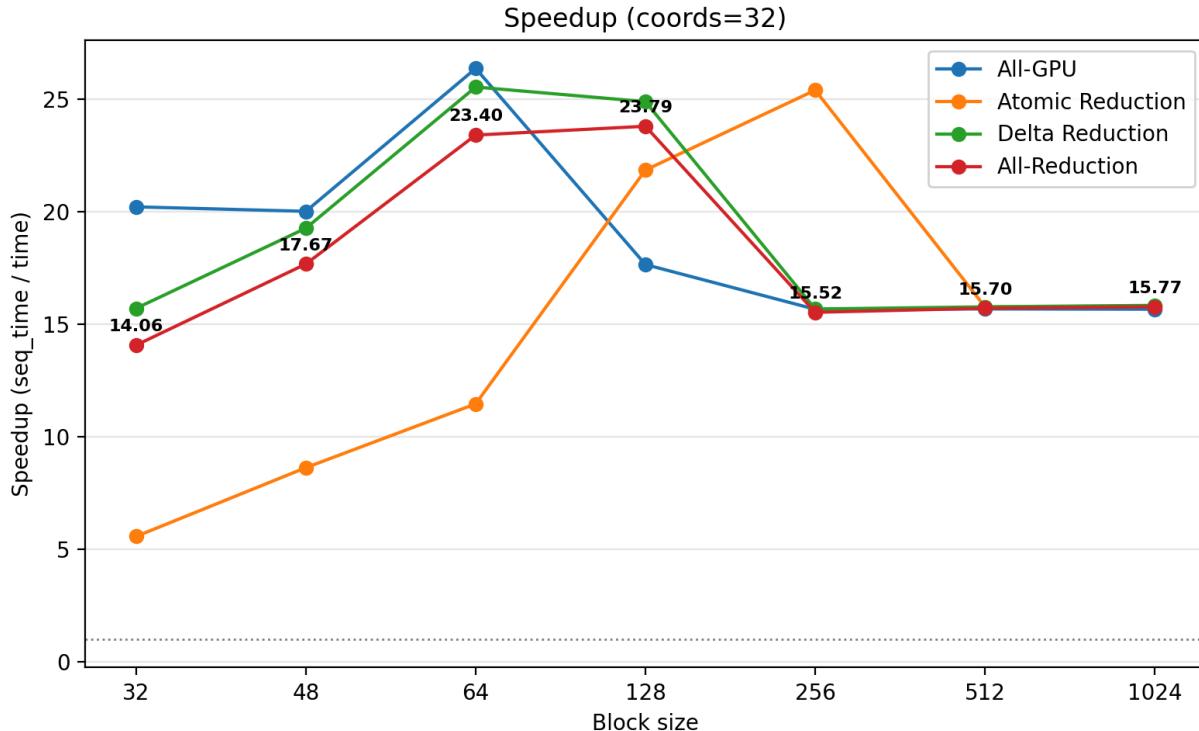
```

    out[idx] = sum;
}

```

Παρακάτω φαίνονται τα διαγράμματα επιτάχυνσης, όπου συγκρίνονται οι μέθοδοι reduction με την All-GPU εκδοχή ως baseline:





Παρατηρήσεις

- Για coords = 2, οι atomic και delta reduction μέθοδοι φαίνονται να είναι καλύτερες από την all-gpu εκδοχή, ειδικά για μεγάλα block sizes. Η All-Reduction, από την άλλη, είναι συστηματικά χειρότερη, αν και με μικρή διαφορά. Η αιτία που δεν είδαμε βελτίωση με την all-reduction εκδοχή είναι γιατί γράφει ένα αντίγραφο των αποτελεσμάτων ανά block στην Global Memory, προσθέτοντας περιττό memory traffic, αλλά και overhead εκκίνησης δεύτερου kernel για την εκτέλεση του reduction (που επιβάλλεται για την ολοκλήρωση του αλγορίθμου). Εκτός αυτού, όλοι οι μέθοδοι reduction προσθέτουν πολυπλοκότητα (κώδικα διαχείρισης shared memory, reduction loops), άρα και επιπλέον υπολογιστικό κόστος, για να λύσουμε ένα πρόβλημα “contention” που για τόσες λίγες συντεταγμένες είναι οριακά ανύπαρκτο.
- Για coords = 32, παρατηρείται το εξής ενδιαφέρον φαινόμενο: στα μικρά block sizes (<64) οι απλούστερες μέθοδοι (all-GPU, delta reduction) έχουν το μεγαλύτερο speedup, ενώ η atomic reduction έχει μεν αύξουσα πορεία, αλλά ξεκινά από πάρα πολύ χαμηλά επίπεδα! Αυτό συμβαίνει γιατί με coords=32 και αρκετά clusters, ο χώρος που δεσμεύει η atomic reduction στη Shared Memory για τα μερικά αθροίσματα είναι σημαντικός. Επιπλέον, υπάρχει ένα σταθερό διαχειριστικό κόστος ανά block για τον μηδενισμό και την τελική αντιγραφή αυτών των αθροίσμάτων στην global memory. Όταν το block size είναι μικρό (π.χ. 32 threads), αυτός ο φόρτος μοιράζεται σε ελάχιστα threads, με αποτέλεσμα κάθε thread να ξοδεύει δυσανάλογα μεγάλο μέρος του χρόνου του σε shared memory operations, αντί για τον "ωφέλιμο" υπολογισμό αποστάσεων. Καθώς το block size αυξάνεται (π.χ. στα 256 threads που έχουμε το μέγιστο speedup που είδαμε για το συγκεκριμένο dataset μεταξύ όλων των benchmarks), η ίδια ποσότητα

διαχειριστικής εργασίας μοιράζεται σε 8-πλάσιο αριθμό threads, οδηγώντας σε δραστική μείωση του overhead ανά thread. Τότε, και μόνο τότε, η μέθοδος atomic reduction καταφέρνει να αναδείξει το πλεονέκτημά της (μείωση του contention στην global memory), ξεπερνώντας σε απόδοση τις απλούστερες μεθόδους.

- Για block sizes > 256, παρατηρείται ότι όλα τα προγράμματα έχουν σχεδόν τον ίδιο χρόνο. Αυτό πιθανώς συμβαίνει διότι η ίδια η GPU είναι το bottleneck, καθώς τα μεγάλα block sizes έχουν απαιτήσεις για πολλούς registers και SMs (Streaming Multiprocessors), με αποτέλεσμα αυτοί να “γεμίζουν”. Επομένως, δεν χωράνε ταυτόχρονα πολλά ενεργά blocks, και τα προγράμματα γίνονται bounded από την χωρητικότητα των SMs.
- Η All-reduction μέθοδος πάλι δεν καταφέρνει να “κερδίσει” και προφανώς αυτό οφείλεται στο τεράστιο κόστος σε Global Memory bandwidth που απαιτεί. Αυτή η τεράστια ποσότητα εγγραφών στην αργή Global Memory προκαλεί συμφόρηση στο memory bus, οπότε ξαναδημιουργείται το πρόβλημα που προσπαθήσαμε να λύσουμε, καθώς στην περίπτωση αυτή έχουμε χειρότερο performance από τα atomic operations.

4. Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένες μνήμης

4.1 Άλλη μία παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-Means

Εδώ καλούμαστε να παραλληλοποιήσουμε άλλη μία φορά τον αλγόριθμο kmeans, αυτή τη φορά με την χρήση του MPI. Δηλαδή θα υλοποιήσουμε παραλληλοποίηση σε αρχιτεκτονική κατανεμημένες μνήμης, όπου η επικοινωνία μεταξύ διαφορετικών διεργασιών γίνεται με ανταλλαγή μηνυμάτων.

Από το εργαστήριο μας δίνεται template κώδικα, και κατάλληλο Makefile. Θα ξεκινήσουμε με το αρχείο **main.c**, συμπληρώνοντας τον κώδικα.

Καταρχάς, όπως και σε κάθε πρόγραμμα MPI, απαιτείται η αρχικοποίηση του περιβάλλοντος MPI με τη συνάρτηση **MPI_Init**, καθώς και ο προσδιορισμός του αναγνωριστικού κάθε διεργασίας (rank) και του συνολικού πλήθους των διεργασιών μέσω των συναρτήσεων **MPI_Comm_rank** και **MPI_Comm_size** αντίστοιχα. Οι πληροφορίες αυτές είναι απαραίτητες για τον σωστό καταμερισμό των δεδομένων και τον συντονισμό της επικοινωνίας μεταξύ των διεργασιών κατά την εκτέλεση του αλγορίθμου.

```

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

```

Έπειτα υπολογίζουμε τον αριθμό των objects από το μέγεθος του dataset, ελέγχουμε ότι είναι τουλάχιστον ίσος με τον αριθμό των clusters και δημιουργούμε το dataset παράλληλα με MPI (περισσότερα για αυτό αργότερα, στο αρχείο file_io.c). Στη συνέχεια, μόνο μία διεργασία (επιλέγουμε αυτή με rank ίσο με 0) αρχικοποιεί τα κέντρα των clusters επιλέγοντας τα πρώτα δεδομένα ως αρχικά κέντρα.

Στο σημείο, αυτό πρέπει να κάνουμε broadcast τα αρχικά cluster centres σε όλα τα ranks. Αυτό το πετυχαίνουμε με την χρήση της συνάρτησης MPI_Bcast. Τα ορίσματα που δέχεται η συνάρτηση σε σειρά είναι:

- Pointer στα δεδομένα που θα γίνουν broadcast
- Πλήθος στοιχείων που θέλουμε να γίνουν broadcast
- MPI Datatype των στοιχείων
- Rank διεργασίας που πραγματοποιεί το broadcast
- MPI Communicator

```

// Rank 0 broadcasts clusters array to everyone else
MPI_Bcast(clusters, numClusters * numCoords, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

```

Έπειτα, αφού δεσμεύουμε μνήμη για τους πίνακες membership και tot_membership, οι οποίοι αποθηκεύουν αντίστοιχα το cluster στο οποίο ανήκει κάθε αντικείμενο σε επίπεδο διεργασίας και συνολικά για όλες τις διεργασίες, προχωράμε στην εκτέλεση του βασικού υπολογιστικού βρόχου του αλγορίθμου.

Καλούμε τη συνάρτηση kmeans, δίνοντας προσοχή στο ότι κάθε διεργασία MPI επεξεργάζεται μόνο το δικό της υποσύνολο δεδομένων (rank_numObjs) και όχι το σύνολο των αντικειμένων (numObjs).

```

// Each MPI process executed K-Means only for a subset of all objects
// rank_numObjs out of the total numObjs
kmeans(objects, numCoords, rank_numObjs, numClusters, threshold,
loop_threshold, membership, clusters);

```

Μετά την ολοκλήρωση του υπολογιστικού μέρους, κάθε διεργασία διαθέτει έναν τοπικό πίνακα membership μεγέθους rank_numObjs, ο οποίος περιέχει για κάθε αντικείμενο του δικού της τμήματος το id του cluster στο οποίο ανήκει. Πρέπει συνεπώς να συλλεχθούν όλα αυτά τα τοπικά κομμάτια σε έναν συνολικό πίνακα tot_membership στον rank 0. Επειδή ο καταμερισμός

των αντικειμένων δεν είναι πάντα απολύτως ισομερής (όταν `numObjs % size != 0`, κάποιοι `ranks` θα έχουν 1 αντικείμενο παραπάνω), δεν μπορούμε να χρησιμοποιήσουμε απλό **`MPI_Gather()`** (ίδιο `sendcount` από όλους). Χρειαζόμαστε την συνάρτηση **`MPI_Gatherv()`**, που επιτρέπει διαφορετικό πλήθος στοιχείων από κάθε διεργασία, αρκεί ο `root` να γνωρίζει πόσα θα λάβει από κάθε `rank` και σε ποια θέση θα τα τοποθετήσει.

Στον παρακάτω κώδικα, ο `rank 0` υπολογίζει δύο βοηθητικούς πίνακες: **`recvcounts`** και **`displs`**. Ο `recvcounts` δηλώνει πόσα στοιχεία θα παραλάβει ο `root` από κάθε διεργασία, ενώ ο `displs` δηλώνει το offset μέσα στον `tot_membership` από όπου θα γραφτούν τα δεδομένα της κάθε διεργασίας. Ο υπολογισμός των πινάκων `recvcounts` και `displs` βασίζεται στον ίδιο τρόπο καταμερισμού των αντικειμένων που χρησιμοποιείται σε όλο την εφαρμογή. Το σύνολο των αντικειμένων μοιράζεται όσο το δυνατόν ισομερώς μεταξύ των διεργασιών: κάθε `rank` επεξεργάζεται έναν βασικό αριθμό αντικειμένων, ενώ τυχόν επιπλέον αντικείμενα κατανέμονται στις πρώτες διεργασίες. Με αυτή τη λογική καθορίζεται αφενός πόσα στοιχεία θα στείλει κάθε διεργασία στον `root` και αφετέρου η διαδοχική θέση στην οποία θα τοποθετηθούν τα δεδομένα της μέσα στον συνολικό πίνακα `tot_membership`. Η ίδια ακριβώς κατανομή εφαρμόζεται και στο `file_io.c` για τον υπολογισμό του `rank_numObjs`, εξασφαλίζοντας συνέπεια μεταξύ τοπικής επεξεργασίας και τελικής συλλογής των αποτελεσμάτων.

```

// Each rank initially receives base = numObjs / size objects.
// The remaining objects (rest = numObjs % size) are distributed to
the first rest ranks.
int base = numObjs / size;
int rest = numObjs % size;
int i; // Loop index
int acc=0; // Accumulator
displs[0] = 0; // Rank 0 writes at the beginning of the buffer
for (i=0; i<rest; i++) {
    recvcounts[i] = base + 1;
    displs[i+1] = acc + base + 1;
    acc += base + 1;
}
for (i=rest; i<size; i++) {
    recvcounts[i] = base;
    if (i != size - 1) {
        displs[i+1] = acc + base;
        acc += base;
    }
}

```

Στην συνέχεια, οι πίνακες `recvcounts` και `displs` που υπολογίζονται στον rank 0 μεταδίδονται σε όλες τις διεργασίες με χρήση **`MPI_Bcast`**, ώστε όλοι οι ranks να έχουν κοινή γνώση της παγκόσμιας διάταξης των δεδομένων πριν από τη φάση της συλλογής.

Η συλλογή των αποτελεσμάτων πραγματοποιείται με τη συνάρτηση **`MPI_Gatherv`**, η οποία επιτρέπει σε κάθε διεργασία να αποστείλει διαφορετικό αριθμό στοιχείων. Συγκεκριμένα, κάθε rank στέλνει τον τοπικό του πίνακα `membership` μεγέθους `rank_numObjs` (`sendbuf`, `sendcount`, `sendtype`), ενώ ο root τα συγκεντρώνει στον πίνακα `tot_membership` (`recvbuf`). Οι πίνακες `recvcounts` και `displs` καθορίζουν αντίστοιχα πόσα στοιχεία λαμβάνονται από κάθε διεργασία και σε ποια θέση του `recvbuf` θα τοποθετηθούν, το **`MPI_INT`** δηλώνει τον τύπο των δεδομένων, ο **rank 0** είναι ο **root** της συλλογής και το **`MPI_COMM_WORLD`** ο communicator στον οποίο πραγματοποιείται η λειτουργία.

```

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

// Rank 0 gathers all the membership information (all ranks run this
// code)
MPI_Gatherv(membership, rank_numObjs, MPI_INT, tot_membership,
            recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);

```

Τέλος, στον rank 0 δίνεται η δυνατότητα προαιρετικού ελέγχου των αποτελεσμάτων σε debug mode, ακολουθεί η αποδέσμευση της δυναμικά δεσμευμένης μνήμης και η ομαλή τερματική φάση του MPI προγράμματος με κλήση της **`MPI_Finalize()`**.

Στην συνέχεια θα πάμε στο αρχείο `kmeans.c`, να προσθέσουμε κατάλληλο κώδικα, ώστε να παραλληλοποιηθεί και ο κύριος υπολογισμός της εφαρμογής μας.

Ο υπολογιστικός πυρήνας του αλγορίθμου παραμένει ουσιαστικά ίδιος με τον σειριακό K-means που έχουμε δει στο εργαστήριο: κάθε διεργασία αναθέτει τα τοπικά της αντικείμενα στο πλησιέστερο cluster, ενημερώνει τα τοπικά αθροίσματα των κέντρων και μετρά πόσα αντικείμενα άλλαξαν cluster. Η βασική διαφορά στην MPI εκδοχή είναι ότι οι ενημερώσεις αυτές γίνονται τοπικά σε κάθε rank και στη συνέχεια απαιτείται επικοινωνία ώστε να προκύψουν τα συνολικά δεδομένα για όλα τα αντικείμενα.

Το πρώτο σημείο όπου εισάγεται MPI είναι η συνένωση των τοπικών μερικών αποτελεσμάτων των clusters. Κάθε διεργασία υπολογίζει τους πίνακες `rank_newClusters` και `rank_newClusterSize`, που αφορούν μόνο το δικό της υποσύνολο δεδομένων. Με τη χρήση της **`MPI_Allreduce`** τα τοπικά αυτά αθροίσματα συνδυάζονται σε καθολικούς πίνακες (`newClusters`, `newClusterSize`) που είναι διαθέσιμοι σε όλες τις διεργασίες. Στη συγκεκριμένη κλήση, τα βασικά

arguments είναι: ο τοπικός πίνακας αποστολής (sendbuf), ο πίνακας υποδοχής των συνολικών τιμών (recvbuf), το πλήθος των στοιχείων, ο τύπος δεδομένων (MPI_INT ή MPI_DOUBLE), το reduction operation (MPI_SUM) και ο communicator (MPI_COMM_WORLD). Με αυτόν τον τρόπο όλες οι διεργασίες μπορούν να υπολογίσουν τα νέα κέντρα clusters με βάση το συνολικό dataset.

Το δεύτερο σημείο MPI αφορά τον έλεγχο σύγκλισης. Κάθε rank υπολογίζει τοπικά τη μεταβλητή rank_delta, δηλαδή πόσα από τα δικά του αντικείμενα άλλαξαν cluster στο τρέχον iteration. Και εδώ χρησιμοποιείται **MPI_Allreduce** με operation **MPI_SUM**, ώστε να προκύψει το συνολικό delta για όλα τα αντικείμενα. Η τιμή αυτή χρησιμοποιείται ως κοινό κριτήριο σύγκλισης από όλες τις διεργασίες.

Ο συνολικός κώδικας είναι ο παρακάτω:

```

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

    rank_delta = 0.0;

    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 rank_delta by 1
        if (membership[i] != index)
            rank_delta += 1.0;

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

        // update new cluster centers : sum of objects located within
        rank_newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            rank_newClusters[index*numCoords + j] += objects[i*numCoords
+ j];
    }
}

```

```

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

// 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];
        }
    }
}

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

// Get fraction of objects whose membership changed during this
loop. This is used as a convergence criterion.
delta /= numObjs;

loop++;
//printf("\r\tcompleted loop %d", loop);
//fflush(stdout);
} while (delta > threshold && loop < loop_threshold);

```

Τέλος, για να μπορεί να γίνει παράλληλα ο υπολογισμός, πρέπει να έχει δημιουργηθεί και το dataset παράλληλα. Για αυτό θα τροποποιήσουμε κατάλληλα και τον κώδικα στο αρχείο file_io.c και συγκεκριμένα την συνάρτηση dataset_generation που καλείται στην main.

Στη συνάρτηση αυτή αρχικοποιείται εκ νέου το MPI περιβάλλον σε επίπεδο διεργασίας, και γίνεται χρήση των **MPI_Comm_rank** και **MPI_Comm_size** ώστε κάθε rank να γνωρίζει τον ρόλο του και το συνολικό πλήθος διεργασιών. Στη συνέχεια υπολογίζεται ο αριθμός αντικειμένων που θα επεξεργαστεί κάθε διεργασία (**rank_numObjs**) με ισομερή κατανομή, όπου κάθε rank λαμβάνει έναν βασικό αριθμό αντικειμένων και τυχόν υπόλοιπα κατανέμονται στις πρώτες διεργασίες. Δηλαδή με παρόμοιο τρόπο με αυτόν που είδαμε προηγουμένως στην main.

Με την ίδια λογική υπολογίζονται και οι πίνακες **sendcounts** και **displs**, οι οποίοι καθορίζουν πόσα στοιχεία δεδομένων θα σταλούν σε κάθε rank και από ποια θέση του συνολικού πίνακα θα ξεκινήσει η αποστολή. Οι πίνακες αυτοί υπολογίζονται στον rank 0 και διαδίδονται σε όλες τις

διεργασίες με **MPI_Bcast**, ώστε να υπάρχει κοινή γνώση της κατανομής πριν τη φάση διαμοιρασμού των δεδομένων.

```

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

/*
 * TODO: Calculate number of objects that each rank will examine
 (*rank_numObjs)
 */
// Each rank initially receives base = numObjs / size objects.
// The remaining objects (rest = numObjs % size) are distributed to the
first rest ranks.
int base = numObjs / size;
int rest = numObjs % size;
if (rank < rest) {
    *rank_numObjs = base + 1;
}
else {
    *rank_numObjs = base;
}

/* allocate space for objects[][] and read all objects */
int sendcounts[size], displs[size];
if (rank == 0) {
    objects = (typeof(objects)) malloc(numObjs * numCoords *
sizeof(*objects));

    int i; // Loop index
    int acc=0; // Accumulator (number of Objects, not number of doubles)
    displs[0] = 0; // Rank 0 takes data at the beginning of the buffer
    for (i=0; i<rest; i++) {
        sendcounts[i] = (base + 1)*numCoords;
        displs[i+1] = (acc + base + 1)*numCoords;
        acc += (base + 1);
    }
    for (i=rest; i<size; i++) {
        sendcounts[i] = base*numCoords;
        if (i != size - 1) {
            displs[i+1] = (acc + base)*numCoords;
            acc += base;
        }
    }
}

```

```

    }

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

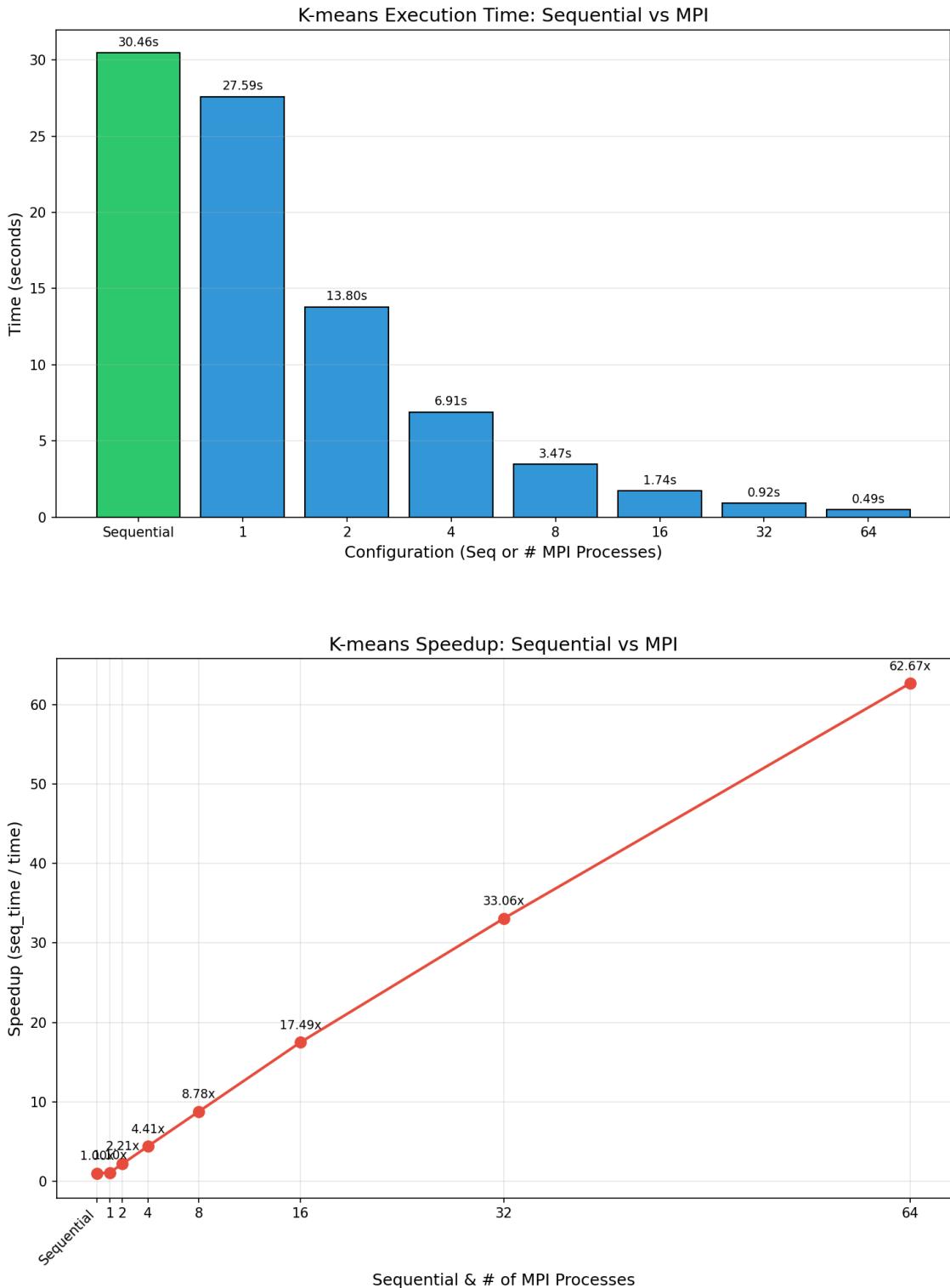
```

Στο σημείο αυτό κάθε διεργασία δεσμεύει μνήμη μόνο για το τοπικό της τμήμα του dataset (rank_objects), σύμφωνα με τον αριθμό αντικειμένων που της έχει ανατεθεί. Η δημιουργία των δεδομένων πραγματοποιείται αποκλειστικά στον rank 0, ο οποίος αρχικοποιεί τον συνολικό πίνακα αντικειμένων με τυχαίες τιμές και τον χρησιμοποιεί ως πηγή για τη διανομή των δεδομένων.

Η διανομή των αντικειμένων στις διεργασίες γίνεται με τη χρήση της **MPI_Scatterv**, η οποία επιλέγεται επειδή επιτρέπει διαφορετικό πλήθος δεδομένων προς κάθε rank. Ο πίνακας **sendcounts** καθορίζει πόσα στοιχεία αποστέλλονται σε κάθε διεργασία, ενώ ο **displs** ορίζει τη θέση έναρξης των δεδομένων μέσα στον συνολικό πίνακα. Κάθε rank λαμβάνει τα δεδομένα του στον τοπικό πίνακα **rank_objects**, καθιστώντας δυνατή την παράλληλη εκτέλεση του αλγορίθμου πάνω σε διαφορετικά υποσύνολα του dataset.

Έχοντας ολοκληρώσει τον κώδικα, θα τρέξουμε συγκεκριμένα πειράματα με configuration: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10} και για 1, 2, 4, 8, 16, 32 και 64 MPI διεργασίες.

Τα παρακάτω διαγράμματα απεικονίζουν τον χρόνο εκτέλεσης της σειριακής υλοποίησης του K-means καθώς και της παράλληλης MPI εκδοχής για διαφορετικό πλήθος MPI διεργασιών, ενώ παράλληλα παρουσιάζεται και το αντίστοιχο speedup που επιτυγχάνεται μέσω της παραλληλοποίησης.



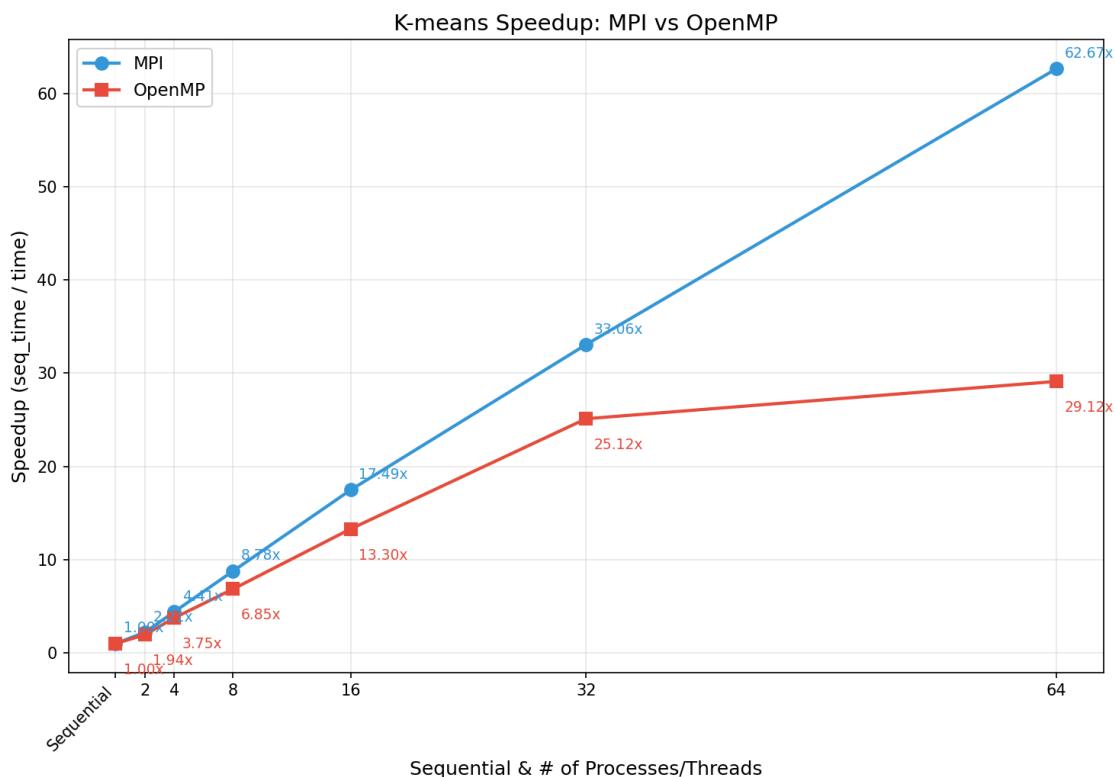
Από το διάγραμμα του speedup φαίνεται ότι η MPI υλοποίηση του K-means παρουσιάζει πολύ καλή κλιμάκωση. Για αριθμό διεργασιών από 2 έως 32 το speedup είναι μεγαλύτερο από το ιδανικό (superlinear), ενώ στα 64 MPI processes παραμένει πολύ κοντά στο ιδανικό x64, με μια μικρή μόνο απόκλιση προς τα κάτω. Συνολικά, τα αποτελέσματα δείχνουν ότι η

παραλληλοποίηση με MPI είναι ιδιαίτερα αποδοτική και αξιοποιεί αποτελεσματικά την αύξηση του αριθμού των διεργασιών.

Σύγκριση Με OpenMP

Στο σημείο αυτό συγκρίνουμε την απόδοση της παράλληλης υλοποίησης του K-means με χρήση MPI με την αντίστοιχη υλοποίηση που αναπτύχθηκε σε προηγούμενη εργαστηριακή άσκηση με OpenMP. Η σύγκριση βασίζεται στους χρόνους εκτέλεσης που μετρήθηκαν για τα δύο μοντέλα παράλληλου προγραμματισμού, με στόχο να αξιολογηθεί η αποδοτικότητα και η κλιμάκωση κάθε προσέγγισης.

Στο παρακάτω διάγραμμα παρουσιάζεται η σύγκριση του speedup της υλοποίησης του K-means με MPI και OpenMP για διαφορετικό αριθμό διεργασιών και νημάτων αντίστοιχα.



Από τα αποτελέσματα φαίνεται ότι και οι δύο προσεγγίσεις επιτυγχάνουν σημαντική επιτάχυνση σε σχέση με τη σειριακή εκτέλεση, ωστόσο η MPI υλοποίηση εμφανίζει σαφώς καλύτερη κλιμάκωση όσο αυξάνεται ο βαθμός παραλληλισμού. Για μικρό αριθμό πόρων οι διαφορές είναι περιορισμένες, αλλά σε μεγαλύτερες διαμορφώσεις η MPI υπερέχει αισθητά, φτάνοντας σε πολύ υψηλότερα επίπεδα speedup σε σχέση με την OpenMP υλοποίηση. Αυτό δείχνει ότι, για το συγκεκριμένο πρόβλημα και περιβάλλον εκτέλεσης, το μοντέλο κατανεμημένης μνήμης αξιοποιεί αποτελεσματικότερα τον διαθέσιμο παραλληλισμό σε μεγάλη κλίμακα.

Παρόλα αυτά, πρέπει να σημειωθεί ότι οι μετρήσεις για τις δύο υλοποιήσεις πραγματοποιήθηκαν σε διαφορετικά συστήματα εκτέλεσης, με διαφορετικά χαρακτηριστικά υλικού και περιβάλλοντος. Για τον λόγο αυτό, τα αποτελέσματα δεν μπορούν να συγκριθούν άμεσα ως προς τους απόλυτους χρόνους και θα πρέπει να ερμηνευθούν με επιφύλαξη. Η σύγκριση βασίζεται κυρίως στη συμπεριφορά του speedup και στις τάσεις κλιμάκωσης, οι οποίες δίνουν μια ενδεικτική εικόνα της αποδοτικότητας κάθε μοντέλου παράλληλου προγραμματισμού, χωρίς όμως να επιτρέπουν οριστικά συμπεράσματα για την απόλυτη υπεροχή της μίας προσέγγισης έναντι της άλλης.

4.2 Διάδοση Θερμότητας σε 2 διαστάσεις

Στην άσκηση αυτή, καλούμαστε να μελετήσουμε το πρόβλημα της διάδοσης θερμότητας σε δύο διαστάσεις, ένα κλασικό πρόβλημα αριθμητικής επίλυσης μερικών διαφορικών εξισώσεων, το οποίο εμφανίζεται σε πλήθος επιστημονικών εφαρμογών. Για την επίλυσή του χρησιμοποιούνται τρεις ευρέως διαδεδομένοι υπολογιστικοί πυρήνες: η μέθοδος *Jacobi*, η μέθοδος *Gauss-Seidel* με *Successive Over-Relaxation (SOR)* και η μέθοδος *Red-Black SOR*. Στόχοι της άσκησης είναι ο εντοπισμός του παραλληλισμού των αλγορίθμων, η υλοποίησή τους σε αρχιτεκτονικές κατανεμημένης μνήμης με τη χρήση του μοντέλου ανταλλαγής μηνυμάτων (MPI), καθώς και η αξιολόγηση της επίδοσής τους μέσω μετρήσεων και συγκρίσεων.

Το πρόβλημα της διάδοσης θερμότητας ανήκει στην κατηγορία των προβλημάτων *stencil*, όπου η τιμή κάθε σημείου του πλέγματος σε μια δεδομένη χρονική στιγμή υπολογίζεται αποκλειστικά με βάση τις τιμές των γειτονικών σημείων του στην προηγούμενη χρονική επανάληψη. Επομένως, οι υπολογισμοί εντός της ίδιας χρονικής στιγμής είναι ανεξάρτητοι μεταξύ τους, γεγονός που καθιστά το πρόβλημα ιδιαίτερα κατάλληλο για παραλληλοποίηση.

Παραλληλοποίηση με πυρήνα Jacobi

Αρχικά, όπως και σε κάθε MPI πρόγραμμα, πρέπει η κάθε διεργασία να αρχικοποιήσει το MPI, και να μάθει τα αναγνωριστικό του από τον communicator:

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Για την ευκολότερη διαχείριση της δισδιάστατης αποσύνθεσης του υπολογιστικού χωρίου, χρησιμοποιείται καρτεσιανός communicator του MPI. Η χρήση καρτεσιανής τοπολογίας επιτρέπει την άμεση και συστηματική αντιστοίχιση των διεργασιών σε ένα λογικό πλέγμα δύο διαστάσεων, το οποίο αντιστοιχεί φυσικά στη γεωμετρία του προβλήματος και διευκολύνει τον εντοπισμό των γειτονικών διεργασιών που απαιτούνται για την ανταλλαγή των ghost cells.

Στον παρακάτω κώδικα, δημιουργείται ένας μη περιοδικός δισδιάστατος καρτεσιανός communicator (**`MPI_Cart_create`**), σύμφωνα με τις διαστάσεις του πλέγματος διεργασιών. Στη συνέχεια, με τη συνάρτηση **`MPI_Cart_coords`**, κάθε διεργασία υπολογίζει τις συντεταγμένες της μέσα στο καρτεσιανό πλέγμα, οι οποίες χρησιμοποιούνται αργότερα για τον εντοπισμό των γειτόνων της και την ορθή επικοινωνία κατά την εκτέλεση του αλγορίθμου Jacobi.

```

MPI_Comm CART_COMM;           //CART_COMM: the new 2D-cartesian
communicator
int periods[2]={0,0};         //periods={0,0}: the 2D-grid is non-periodic
int rank_grid[2];            //rank_grid: the position of each process on
the new communicator

MPI_Cart_create(MPI_COMM_WORLD,2,grid,periods,0,&CART_COMM);
MPI_Cart_coords(CART_COMM,rank,2,rank_grid);

```

Στη συνέχεια, υπολογίζονται οι διαστάσεις των τοπικών subdomains που αναλαμβάνει κάθε MPI διεργασία, με βάση τις διαστάσεις του συνολικού υπολογιστικού χωρίου και του πλέγματος διεργασιών. Σε περίπτωση που το χωρίο δεν μπορεί να κατανεμηθεί ισομερώς, εφαρμόζεται padding ώστε οι διαστάσεις του να είναι πολλαπλάσιες του αριθμού των διεργασιών ανά διάσταση. Έπειτα, η διεργασία με rank 0 δεσμεύει και αρχικοποιεί το καθολικό grid, ενώ κάθε διεργασία δεσμεύει τις αντίστοιχες τοπικές υποπεριοχές μνήμης, προσθέτοντας επιπλέον γραμμές και στήλες για τη διαχείριση των ghost cells που απαιτούνται για την επικοινωνία μεταξύ γειτονικών διεργασιών.

Σε αυτό το στάδιο πρέπει να γίνει η διανομή του καθολικού δισδιάστατου υπολογιστικού χωρίου από τη διεργασία με rank 0 προς όλες τις υπόλοιπες διεργασίες. Για τον σκοπό αυτό ορίζονται προηγουμένως κατάλληλοι MPI derived datatypes, οι οποίοι περιγράφουν με ακρίβεια τα τοπικά δισδιάστατα υποχωρία που αντιστοιχούν σε κάθε διεργασία. Η χρήση παραγώγων τύπων δεδομένων επιτρέπει την αποδοτική μεταφορά ολόκληρων υποπλεγμάτων του πίνακα χωρίς την ανάγκη χειροκίνητης αντιγραφής ή αναδιάταξης δεδομένων, διευκολύνοντας έτσι την παραλληλοποίηση του αλγορίθμου.

Πιο συγκεκριμένα, χρησιμοποιείται η συνάρτηση **`MPI_Type_vector`** για τον ορισμό δισδιάστατων μπλοκ δεδομένων, καθορίζοντας τον αριθμό των γραμμών (count), το μήκος κάθε γραμμής (**`blocklength`**) και το βήμα (**`stride`**) μεταξύ διαδοχικών γραμμών στη μνήμη. Στη συνέχεια, η **`MPI_Type_create_resized`** χρησιμοποιείται ώστε να προσαρμοστεί σωστά το memory extent του νέου τύπου δεδομένων, επιτρέποντας τη σωστή στοιχίση των υποπλεγμάτων κατά τη χρήση συλλογικών συναρτήσεων όπως η `MPI_Scatterv` και η `MPI_Gatherv`. Τέλος, με την **`MPI_Type_commit`** οριστικοποιούνται οι νέοι τύποι δεδομένων, ώστε να μπορούν να χρησιμοποιηθούν με ασφάλεια στις επικοινωνίες μεταξύ των διεργασιών.

```

//----Datatype definition for the 2D-subdomain on the global
matrix----/
MPI_Datatype global_block;
MPI_Type_vector(local[0],local[1],global_padded[1],MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&global_block);
MPI_Type_commit(&global_block);

//----Datatype definition for the 2D-subdomain on the local matrix----/
MPI_Datatype local_block;
MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
MPI_Type_commit(&local_block);

```

Έπειτα, η διεργασία με rank 0 υπολογίζει τη διάταξη των τοπικών υποχωρίων μέσα στο καθολικό πλέγμα, καθορίζοντας για κάθε διεργασία το πλήθος και τη θέση των μπλοκ δεδομένων που θα διανεμηθούν. Οι πληροφορίες αυτές χρησιμοποιούνται από τις συλλογικές συναρτήσεις επικοινωνίας της MPI, ώστε να επιτευχθεί η σωστή και αποδοτική κατανομή του δισδιάστατου χωρίου σε όλα τα παράλληλα νήματα εκτέλεσης.

Σε αυτό το στάδιο πραγματοποιείται η αρχική κατανομή του καθολικού υπολογιστικού χωρίου στις επιμέρους διεργασίες. Κάθε διεργασία λαμβάνει το αντίστοιχο τοπικό υποχώριο του πίνακα, το οποίο θα χρησιμοποιηθεί τόσο για τον υπολογισμό της μεθόδου Jacobi όσο και για την ανταλλαγή οριακών δεδομένων με τις γειτονικές διεργασίες. Παράλληλα, διασφαλίζεται ότι οι τοπικοί πίνακες είναι σωστά αρχικοποιημένοι και περιλαμβάνουν τον απαραίτητο χώρο για ghost cells, οι οποίες είναι κρίσιμες για την ορθή εκτέλεση του stencil υπολογισμού.

Η διανομή των δεδομένων υλοποιείται με τη συλλογική συνάρτηση **MPI_Scatterv**, η οποία επιτρέπει την αποστολή διαφορετικών τμημάτων του καθολικού πίνακα σε κάθε διεργασία, βάσει προκαθορισμένων μετρητών και μετατοπίσεων. Ως buffer αποστολής χρησιμοποιείται ο καθολικός πίνακας μόνο από τη διεργασία με rank 0, ενώ οι υπόλοιπες διεργασίες λαμβάνουν τα δεδομένα τους στις εσωτερικές θέσεις των τοπικών πινάκων, ώστε να διατηρηθούν τα ghost cells αμετάβλητα. Η ίδια διαδικασία εφαρμόζεται τόσο για τον πίνακα της προηγούμενης όσο και της τρέχουσας επανάληψης, εξασφαλίζοντας τη σωστή αρχικοποίηση πριν την έναρξη της επαναληπτικής διαδικασίας.

```

// We initialize u_current and u_previous
init2d(u_previous, local[0] + 2, local[1] + 2);
init2d(u_current, local[0] + 2, local[1] + 2);

// Only rank 0 (the sender) needs to have sendbuf
double* U0_ptr;

```

```

if (rank == 0) {
    U0_ptr=&(U[0][0]);
}
else {
    U0_ptr= NULL;
}
// We send to u_previous[1][1] and not [0][0] because those are ghost
cells
MPI_Scatterv(U0_ptr, scattercounts, scatteroffset, global_block,
&(u_previous[1][1]), 1, local_block,
0, MPI_COMM_WORLD);
MPI_Scatterv(U0_ptr, scattercounts, scatteroffset, global_block,
&(u_current[1][1]), 1, local_block,
0, MPI_COMM_WORLD);

// After Scatterv we don't need the global array so far
if (rank==0)
    free2d(U);

```

Για την ανταλλαγή οριακών δεδομένων απαιτείται η αποστολή τόσο γραμμών όσο και στηλών των τοπικών πινάκων. Επειδή στη C οι στήλες δεν είναι αποθηκευμένες συνεχόμενα στη μνήμη (οι πίνακες είναι row-major), ορίζεται ένας derived datatype που περιγράφει μια στήλη του πίνακα.

Ο τύπος column επιτρέπει την αποδοτική επικοινωνία στηλών μεταξύ γειτονικών διεργασιών χωρίς τη χρήση ενδιάμεσων buffers. Για την υλοποίηση χρησιμοποιούμε τις συναρτήσεις MPI_Datatype, MPI_Type_vector, MPI_Type_create_resized, MPI_Type_commit, όπως προηγουμένως.

```

MPI_Datatype column;
MPI_Type_vector(local[0], 1, local[1] + 2, MPI_DOUBLE, &dummy);
MPI_Type_create_resized(dummy, 0, sizeof(double), &column);
MPI_Type_commit(&column);

```

Στο σημείο αυτό κάθε διεργασία προσδιορίζει τις γειτονικές διεργασίες με τις οποίες θα ανταλλάσσει οριακά δεδομένα κατά την εκτέλεση του stencil υπολογισμού. Η αναγνώριση των γειτόνων είναι απαραίτητη ώστε κάθε υποχώριο να λαμβάνει τις σωστές τιμές στα ghost cells του από τα γειτονικά υποχωρία, λαμβάνοντας υπόψη ότι οι διεργασίες στα όρια του πλέγματος δεν διαθέτουν γείτονες προς όλες τις κατευθύνσεις.

Ο εντοπισμός των γειτόνων υλοποιείται με τη συνάρτηση **MPI_Cart_shift**, η οποία, με βάση τον καρτεσιανό communicator, επιστρέφει τα ranks των διεργασιών που βρίσκονται σε θετική και αρνητική μετατόπιση κατά μήκος κάθε διάστασης. Εφόσον το πλέγμα διεργασιών είναι μη περιοδικό, για τις διεργασίες που βρίσκονται στα άκρα του πλέγματος η συνάρτηση επιστρέφει την τιμή **MPI_PROC_NULL**, η οποία χρησιμοποιείται στη συνέχεια για την ασφαλή παράλειψη επικοινωνίας με ανύπαρκτους γείτονες.

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

// Shift by 1 step in x direction (find north and south neighbor) and y
// direction (find west and east neighbor)
// MPI_Cart_shift will return MPI_PROC_NULL for non-existing neighbors
MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);
```

Σε αυτό το σημείο ορίζονται τα όρια επανάληψης για κάθε διεργασία, ώστε οι υπολογισμοί να εκτελούνται μόνο στα έγκυρα εσωτερικά στοιχεία του τοπικού υποχωρίου. Τα αρχικά όρια εξαιρούν τις ghost cells, ενώ στη συνέχεια προσαρμόζονται ανάλογα με τη θέση της διεργασίας στο πλέγμα και την ύπαρξη ή μη γειτονικών διεργασιών.

Συγκεκριμένα, όταν μια διεργασία βρίσκεται σε οριακή θέση (π.χ. χωρίς north, south, west ή east γείτονα), τα αντίστοιχα όρια επανάληψης μετατοπίζονται κατάλληλα ώστε να αποφεύγεται ο υπολογισμός σε ghost cells, οριακά σημεία ή στοιχεία που ανήκουν στο padding του καθολικού πίνακα. Με αυτόν τον τρόπο ο κώδικας καλύπτει ομοιόμορφα όλες τις περιπτώσεις διεργασιών, τόσο εσωτερικές όσο και οριακές, διασφαλίζοντας τη σωστή εκτέλεση του αλγορίθμου Jacobi.

```
//---Define the iteration ranges per process----//
int i_min,i_max,j_min,j_max;
// 0 and local[0]/[1] are ghost cells
i_min = 1;
i_max = local[0] + 1;
if (north == MPI_PROC_NULL) {
    i_min = 2; // ghost cell + boundary
}
if (south == MPI_PROC_NULL) {
    i_max -= (global_padded[0] - global[0]) + 1;
}

j_min = 1;
j_max = local[1] + 1;
if (west == MPI_PROC_NULL) {
```

```

        j_min = 2;    // ghost cell + boundary
    }
    if (east == MPI_PROC_NULL) {
        j_max -= (global_padded[1] - global[1]) + 1;
    }
}

```

Ο υπολογιστικός πυρήνας υλοποιείται ως επαναληπτικός βρόχος Jacobi. Όταν είναι ενεργοποιημένος ο έλεγχος σύγκλισης (**TEST_CONV**), ο αλγόριθμος Jacobi εκτελείται μέχρι να επιτευχθεί καθολική σύγκλιση όλων των διεργασιών. Αντίθετα, χωρίς έλεγχο σύγκλισης, η εκτέλεση γίνεται για προκαθορισμένο αριθμό επαναλήψεων (256), ανεξάρτητα από τη σύγκλιση της λύσης.

Σε κάθε επανάληψη γίνεται πρώτα εναλλαγή δεικτών μεταξύ **u_previous** και **u_current**, ώστε ο νέος υπολογισμός να γράφεται σε ξεχωριστό πίνακα από αυτόν που χρησιμοποιείται για ανάγνωση. Στη συνέχεια πραγματοποιείται ανταλλαγή οριακών τιμών (ghost cells) με τους τέσσερις γείτονες: στέλνεται/λαμβάνεται η πάνω και κάτω γραμμή, ενώ για τις αριστερή/δεξιά στήλη χρησιμοποιείται ο παράγωγος τύπος column. Η επικοινωνία γίνεται με **MPI_Sendrecv** ώστε να πραγματοποιείται ταυτόχρονα αποστολή και λήψη χωρίς κίνδυνο deadlock, ενώ στις διεργασίες που δεν έχουν γείτονα (π.χ. **MPI_PROC_NULL**) η επικοινωνία παραλείπεται.

Αφού ενημερωθούν τα ghost cells, εκτελείται το τοπικό βήμα Jacobi καλώντας **Jacobi(u_previous, u_current, i_min, i_max, j_min, j_max)**, η οποία υπολογίζει τις νέες τιμές μόνο στο εσωτερικό εύρος δεικτών που έχει καθοριστεί. Επιπροσθέτως, όταν είναι ενεργό το **TEST_CONV**, κάθε 100 επαναλήψεις γίνεται έλεγχος σύγκλισης στο τοπικό υποχώριο μέσω της συνάρτησης **converge()**, που μας δίνεται από το εργαστήριο στο αρχείο **utils.c**.

Ο τοπικός έλεγχος σύγκλισης κάθε διεργασίας συνδυάζεται σε καθολικό επίπεδο με τη συλλογική συνάρτηση **MPI_Allreduce**, η οποία εκτελεί ταυτόχρονα reduction και διάχυση του αποτελέσματος σε όλες τις διεργασίες. Με τη χρήση του τελεστή **MPI_MIN**, η καθολική σύγκλιση επιτυγχάνεται μόνο όταν όλες οι διεργασίες έχουν συγκλίνει, καθώς η ύπαρξη έστω μίας μη συγκλίνουσας διεργασίας αρκεί για να συνεχιστεί η επαναληπτική διαδικασία.

Για τη χρονομέτρηση του υπολογιστικού πυρήνα χρησιμοποιούνται τρία ζευγάρια μετρήσεων **gettimeofday**. Το πρώτο (**tts**, **ttf**) μετρά τον συνολικό χρόνο του πυρήνα, δηλαδή από την είσοδο στον επαναληπτικό βρόχο μέχρι την έξοδο. Το δεύτερο (**tcs**, **tcf**) περικλείει αποκλειστικά την κλήση της **Jacobi()** και χρησιμοποιείται για τη συσσώρευση του καθαρού χρόνου υπολογισμών (**tcomp**). Τέλος, όταν είναι ενεργό το **TEST_CONV**, το τρίτο ζευγάρι (**tcvs**, **tcvf**) χρονομετρεί τη φάση ελέγχου σύγκλισης (κλήση **converge()** και την αντίστοιχη συλλογική επικοινωνία), ενημερώνοντας το **tconv**.

Μετά το τέλος των επαναλήψεων, κάθε διεργασία διαθέτει τους δικούς της χρόνους και στη συνέχεια εφαρμόζεται **MPI_Reduce** με τελεστή **MPI_MAX** προς τη διεργασία 0 για να ληφθεί ο

μέγιστος συνολικός χρόνος (**total_time**) και ο μέγιστος χρόνος υπολογισμών (**comp_time**) ανάμεσα σε όλες τις διεργασίες. Η επιλογή του μέγιστου είναι κατάλληλη, γιατί σε περιβάλλον MPI ο συνολικός χρόνος εκτέλεσης καθορίζεται από τη διεργασία που ολοκληρώνει τελευταία.

Ο συνολικός κώδικας του υπολογιστικού πυρήνα βρίσκεται παρακάτω:

```
//----Computational core----//
gettimeofday(&tt, NULL);
#ifndef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
#undef T
#define T 256
for (t=0;t<T;t++) {
#endif
    /*Compute and Communicate*/
    /*Add appropriate timers for computation*/
    swap = u_previous;
    u_previous = u_current;
    u_current = swap;
    // Send to your north neighbor your top row and receive its bottom
row
    if (north != MPI_PROC_NULL) {
        MPI_Sendrecv(&u_previous[1][1], local[1], MPI_DOUBLE, north, 0,
&u_previous[0][1], local[1],
        MPI_DOUBLE, north, 0, MPI_COMM_WORLD,&status);
    }
    // Send to your south neighbor your bottom row and receive its top
row
    if (south != MPI_PROC_NULL) {
        MPI_Sendrecv(&u_previous[local[0]][1], local[1], MPI_DOUBLE,
south, 0, &u_previous[local[0] + 1][1], local[1],
        MPI_DOUBLE, south, 0, MPI_COMM_WORLD, &status);
    }
    // Send to your east neighbor your right-most column and receive its
left-most column
    if (east != MPI_PROC_NULL) {
        MPI_Sendrecv(&u_previous[1][local[1]], 1, column, east, 0,
&u_previous[1][local[1] + 1], 1,
        column, east, 0, MPI_COMM_WORLD, &status);
    }
    // Send to your west neighbor your left-most column and receive its
right-most column
```

```

    if (west != MPI_PROC_NULL) {
        MPI_Sendrecv(&u_previous[1][1], 1, column, west, 0,
&u_previous[1][0], 1,
                column, west, 0, MPI_COMM_WORLD, &status);
    }
    gettimeofday(&tcs, NULL);
    Jacobi(u_previous, u_current, i_min, i_max, j_min, j_max);
    gettimeofday(&tcf, NULL);
    // Update total computation time
    tcomp += (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec - tcs.tv_usec) *
0.000001;
    #ifdef TEST_CONV
    if (t%C==0) {
        /*Test convergence*/
        gettimeofday(&t cvs, NULL);
        converged = converge(u_previous, u_current, i_min, i_max, j_min,
j_max);
        // Check whether every process converged (that's why we use
MPI_MIN, so if even one hasn't converged we keep going)
        MPI_Allreduce(&converged, &global_converged, 1, MPI_INT,
MPI_MIN, MPI_COMM_WORLD);
        gettimeofday(&t cvf, NULL);
        // Update total converge time
        tconv += (tcvf.tv_sec - tcvs.tv_sec) + (tcvf.tv_usec -
tcvs.tv_usec) * 0.000001;
    }
    #endif
}
gettimeofday(&ttf,NULL);

ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;

MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

```

Μετά την ολοκλήρωση του υπολογιστικού πυρήνα, τα τοπικά υποχωρία όλων των διεργασιών συγκεντρώνονται εκ νέου στον καθολικό πίνακα από τη διεργασία με rank 0. Η διεργασία αυτή δεσμεύει τον απαιτούμενο χώρο για τον καθολικό πίνακα, ενώ οι υπόλοιπες διεργασίες συμμετέχουν μόνο ως αποστολείς δεδομένων.

Η συγκέντρωση των δεδομένων υλοποιείται με τη συλλογική συνάρτηση **MPI_Gatherv**, η οποία επιτρέπει τη συλλογή των τοπικών πινάκων με βάση τα προκαθορισμένα offsets και counts που

αντιστοιχούν στη δισδιάστατη αποσύνθεση του χωρίου. Τα δεδομένα αποστέλλονται από τις εσωτερικές θέσεις των τοπικών πινάκων ([1][1]), ώστε να εξαιρούνται τα ghost cells και να ανασυντίθεται σωστά ο καθολικός πίνακας.

```
//----Rank 0 gathers local matrices back to the global matrix----//
if (rank==0) {
    U=allocate2d(global_padded[0],global_padded[1]);
    U0_ptr = &(U[0][0]);
}
else {
    U0_ptr = NULL;
}
// Pointer to [1][1] because [0][0] is ghost cell
MPI_Gatherv(&u_current[1][1], 1, local_block, U0_ptr, scattercounts,
scatteroffset, global_block, 0, MPI_COMM_WORLD);
```

Τέλος, μετά την ολοκλήρωση της παράλληλης εκτέλεσης, η διεργασία με rank 0 αναλαμβάνει την συλλογή και παρουσίαση των αποτελεσμάτων. Εκτυπώνονται βασικές πληροφορίες για την εκτέλεση του αλγορίθμου και οι χρόνοι εκτέλεσης. Προαιρετικά, όταν είναι ενεργοποιημένο το flag **PRINT_RESULTS** αποθηκεύεται το τελικό αποτέλεσμα σε αρχείο, ενώ όταν είναι ενεργοποιημένο το **TEST_CONV** καταγράφεται και ο συνολικός χρόνος σύγκλισης στο standard output. Προτού επιστρέψει η main, καλείται η **MPI_Finalize()**, η οποία τερματίζει ομαλά το περιβάλλον MPI, απελευθερώνοντας τους πόρους επικοινωνίας και σηματοδοτώντας την ολοκλήρωση της παράλληλης εκτέλεσης του προγράμματος.

Παραλληλοποίηση με πυρήνα Gauss-Seidel SOR

Η παράλληλη υλοποίηση της μεθόδου Gauss-Seidel με Successive Over-Relaxation (SOR) ακολουθεί την ίδια γενική λογική με τον Jacobi ως προς τη διάσπαση του δισδιάστατου χωρίου σε υποπεριοχές και την ανταλλαγή ghost cells με τους 4 γείτονες κόμβους (north/south/east/west). Η βασική διαφορά έγγυται στον τρόπο ενημέρωσης των κελιών μέσα σε κάθε επανάληψη: στον Gauss-Seidel, οι νέες τιμές χρησιμοποιούνται άμεσα καθώς σαρώνουμε το πλέγμα, γεγονός που εισάγει εξαρτήσεις δεδομένων εντός της ίδιας επανάληψης, σε αντίθεση με τον Jacobi, όπου ο πυρήνας ενημέρωσης ήταν πλήρως “explicit” (για κάθε κελί (i,j) υπολογίζαμε τη νέα τιμή γράφοντας στο u_current[i][j] χρησιμοποιώντας μόνο τιμές της προηγούμενης επανάληψης από τον u_previous). Η σκέψη πίσω από αυτή την αλγορίθμική βελτίωση είναι να εκμεταλλευτούμε τις most-up-to-date τιμές που έχουν ενημερωθεί στην ίδια επανάληψη και ταυτόχρονα να εφαρμόσουμε over-relaxation, με σκοπό την επιτάχυνση της σύγκλισης.

Για την υλοποίηση, εφαρμόσαμε πληθώρα αλλαγών στον κώδικα που χρησιμοποιούσε πυρήνα Jacobi, ώστε πλέον να γίνεται σύγκλιση με τον πυρήνα Gauss-Seidel.

Αρχικά, υλοποιήσαμε τον ίδιο τον πυρήνα ως μια συνάρτηση C:

```
void GaussSeidelSOR(double **u_previous, double **u_current, int X_min, int
X_max, int Y_min, int Y_max, double omega) {
    int i, j;
    for (i = X_min; i < X_max; i++) {
        for (j = Y_min; j < Y_max; j++) {
            u_current[i][j] = u_previous[i][j]
                + (u_current[i - 1][j] + u_previous[i + 1][j]
                + u_current[i][j - 1] + u_previous[i][j +
1])
                - 4.0 * u_previous[i][j])
                * omega / 4.0;
        }
    }
}
```

Επειδή ο πυρήνας Gauss-Seidel απαιτεί και ένα relaxation factor ω , το αρχικοποιήσαμε στην παρακάτω τιμή:

```
omega = 2.0 / (1 + sin(3.14 / global[0]));
```

Η ανταλλαγή μηνυμάτων στον Gauss-Seidel SOR γίνεται με τον ίδιο τρόπο όπως και στη μέθοδο Jacobi, χρησιμοποιώντας ανταλλαγή ghost cells με τους τέσσερις γειτονικούς κόμβους μέσω **`MPI_Sendrecv`**. Συγκεκριμένα, αποστέλλονται και λαμβάνονται οι πάνω/κάτω γραμμές ως συνεχόμενα τμήματα μνήμης (`MPI_DOUBLE`), ενώ για τις αριστερές/δεξιές στήλες χρησιμοποιείται ο ίδιος παράγωγος τύπος δεδομένων (`datatype column`) ώστε να μεταφέρονται αποδοτικά μη συνεχόμενα στοιχεία λόγω row-major διάταξης στη C. Η μοναδική διαφοροποίηση σε σχέση με τον Jacobi δεν αφορά στην ίδια την επικοινωνία, αλλά στην επεξεργασία των δεδομένων μετά το halo exchange. Επειδή ο Gauss-Seidel kernel αξιοποιεί και τιμές του `u_current` (ήδη ενημερωμένες μέσα στην ίδια επανάληψη), αντιγράφονται οι ghost boundary τιμές από τον `u_previous` στον `u_current` για να εξασφαλιστεί ότι τα συνοριακά δεδομένα που χρησιμοποιούνται στον υπολογισμό παραμένουν συνεπή με την προηγούμενη επανάληψη.

```
if (north != MPI_PROC_NULL) {
    MPI_Sendrecv(&u_previous[1][1], local[1], MPI_DOUBLE, north, 0,
&u_previous[0][1], local[1], MPI_DOUBLE,
```

```

        north, 0, MPI_COMM_WORLD, &status);
}
if (south != MPI_PROC_NULL) {
    MPI_Sendrecv(&u_previous[local[0]][1], local[1], MPI_DOUBLE, south, 0,
&u_previous[local[0] + 1][1],
                local[1], MPI_DOUBLE, south, 0, MPI_COMM_WORLD, &status);
}

if (east != MPI_PROC_NULL) {
    MPI_Sendrecv(&u_previous[1][local[1]], 1, column, east, 0,
&u_previous[1][local[1] + 1], 1, column, east,
                0, MPI_COMM_WORLD, &status);
}
if (west != MPI_PROC_NULL) {
    MPI_Sendrecv(&u_previous[1][1], 1, column, west, 0, &u_previous[1][0],
1, column, west, 0,
                MPI_COMM_WORLD, &status);
}

for (i = 0; i < local[0] + 2; i++) {
    u_current[i][0] = u_previous[i][0];
    u_current[i][local[1] + 1] = u_previous[i][local[1] + 1];
}
for (j = 0; j < local[1] + 2; j++) {
    u_current[0][j] = u_previous[0][j];
    u_current[local[0] + 1][j] = u_previous[local[0] + 1][j];
}

gettimeofday(&tcs, NULL);
GaussSeidelSOR(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;

```

Παραλληλοποίηση με πυρήνα Red-Black SOR

Η μέθοδος Red-Black-SOR είναι μια παραλλαγή του Gauss-Seidel που στοχεύει να συνδυάσει τα πλεονεκτήματα της ταχύτερης σύγκλισης του Gauss-Seidel-SOR με μια πιο “καθαρή” παραλληλοποίηση. Η βασική ιδέα είναι ο χρωματισμός του πλέγματος με λογική σκακιέρας, δηλαδή όλοι οι γείτονες north, south, west, east έχουν διαφορετικό χρώμα από το τρέχον κελί.

Μαθηματικά, κάθε κελί (i,j) χαρακτηρίζεται ως κόκκινο ή μαύρο ανάλογα με την ιδιότητα $(i+j) \ mod \ 2$. Έτσι, κάθε κόκκινο κελί θα ενημερώνεται μόνο από μαύρα κελιά-γείτονες, και αντιστρόφως, επιτρέποντας ανεξάρτητη ενημέρωση των κελιών κάθε χρωματικής ομάδας. Συνεπώς, τα δύο γκρουπ κελιών θα ενημερώνονται εναλλάξ και θα λαμβάνουν τις ήδη ενημερωμένες τιμές από την ομάδα του άλλου χρώματος.

Με τη λογική αυτή, η υλοποίηση με Red-Black-SOR, μέσα σε μια επανάληψη, αξιοποιεί “νεότερες” τιμές σε σχέση με τον Jacobi και, σε περιβάλλον MPI, μπορεί να μειώσει το κόστος επικοινωνίας μεταξύ γειτονικών κόμβων. Συγκεκριμένα, η επανάληψη χωρίζεται σε δύο διαδοχικές φάσεις. Αρχικά, υπολογίζονται οι νέες τιμές των κόκκινων κελιών και, στη συνέχεια, αφού έχει προηγηθεί η κατάλληλη ανταλλαγή οριακών δεδομένων στα ghost cells, υπολογίζονται οι νέες τιμές των μαύρων κελιών χρησιμοποιώντας πλέον τις most-up-to-date τιμές των κόκκινων γειτόνων. Με αυτόν τον τρόπο, το δεύτερο βήμα μιας επανάληψης, που είναι η ανανέωση των μαύρων κελιών, δεν βασίζεται αποκλειστικά σε δεδομένα της προηγούμενης επανάληψης, αλλά ενσωματώνει πληροφορία που προέκυψε μέσα στο ίδιο iteration, άρα η διάδοση της πληροφορίας γίνεται πιο γρήγορα.

Σε σύγκριση με τον Gauss-Seidel, η Red-Black διάσπαση έχει ένα επιπλέον πλεονέκτημα ως προς την παραλληλοποίηση. Στον κλασικό Gauss-Seidel η ενημέρωση γίνεται με σειριακή σάρωση και μόνο ένα μέρος των γειτονικών τιμών είναι “up-to-date”, ενώ οι υπόλοιπες παραμένουν από την προηγούμενη επανάληψη. Στην MPI υλοποίηση αυτό ενισχύεται από το γεγονός ότι οι πιο πρόσφατες τιμές δεν μεταφέρονται απαραίτητα άμεσα μεταξύ διαφορετικών υποχωρίων εντός της ίδιας επανάληψης. Αντίθετα, στο Red-Black-SOR η εξάρτηση οργανώνεται σε δύο καθαρά στάδια (red-black), και με την ανταλλαγή ghost cells ανάμεσα στα δύο υπο-βήματα εξασφαλίζεται ότι οι ενημερωμένες τιμές είναι διαθέσιμες και στα boundaries των υποχωρίων πριν από το update του άλλου γκρουπ κελιών. Έτσι, η “νεότερη” πληροφορία αξιοποιείται πιο ομοιόμορφα στο καθολικό πλέγμα, κάτι που συνήθως οδηγεί σε καλύτερη συμπεριφορά σύγκλισης.

Όσον αφορά την υλοποίηση με τον Red-Black πυρήνα, οι διαφοροποιήσεις σε σχέση με τη Jacobi υλοποίηση αφορούν στο computational core, όπου πλέον ο Red-Black χωρίζει κάθε iteration σε 2 kernel invocations (red και black) και 2 halo exchanges, για την λήψη τιμών από τα γειτονικά ranks.

Πιο συγκεκριμένα, έχουμε:

- ανταλλαγή του `u_previous` πριν το RedSOR (ίδιο μοτίβο με Jacobi)
- ανταλλαγή του `u_current` μετά το RedSOR, αλλά πριν το BlackSOR
- επιτυγχάνεται εν τέλει η χρήση νεότερων τιμών στην εκτέλεση του BlackSOR, όπου οι γείτονες διαβάζονται από `u_current` και όχι `u_previous`.

```
/* code inside for loop - computational core */
```

```
swap = u_previous;
```

```

u_previous = u_current;
u_current = swap;

MPI_Sendrecv(... u_previous ...);

for (i = 0; i < local[0] + 2; i++) {
    u_current[i][0] = u_previous[i][0];
    u_current[i][local[1] + 1] = u_previous[i][local[1] + 1];
}
for (j = 0; j < local[1] + 2; j++) {
    u_current[0][j] = u_previous[0][j];
    u_current[local[0] + 1][j] = u_previous[local[0] + 1][j];
}

gettmeofday(&tcs, NULL);

/* Red phase */
RedSOR(u_previous, u_current, i_min, i_max, j_min, j_max, omega);

MPI_Sendrecv(... u_current ... tag=1 ...);

/* Black phase (with updated values) */
BlackSOR(u_previous, u_current, i_min, i_max, j_min, j_max, omega);

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

```

Παρατηρούμε ότι σε αυτή την περίπτωση οι timers (μετρούν το computation time) περικλείουν περισσότερη χρήσιμη δουλειά, πράγμα το οποίο θα αποτυπωθεί και στα διαγράμματα των αποτελεσμάτων.

Παραθέτουμε και τις υλοποιήσεις των πυρήνων:

Red sweep:

```

void RedSOR(double **u_previous, double **u_current,
            int X_min, int X_max, int Y_min, int Y_max,
            double omega)
{
    int i, j;
    for (i = X_min; i < X_max; i++) {

```

```

    for (j = Y_min; j < Y_max; j++) {
        if ((i + j) % 2 == 0) {
            u_current[i][j] =
                u_previous[i][j]
                + (omega / 4.0) *
                    u_previous[i - 1][j] +
                    u_previous[i + 1][j] +
                    u_previous[i][j - 1] +
                    u_previous[i][j + 1]
                    - 4.0 * u_previous[i][j]
            );
        }
    }
}

```

Black sweep:

```
void BlackSOR(double **u_previous, double **u_current,
              int X_min, int X_max, int Y_min, int Y_max,
              double omega)
{
    int i, j;
    for (i = X_min; i < X_max; i++) {
        for (j = Y_min; j < Y_max; j++) {
            if ((i + j) % 2 == 1) {
                u_current[i][j] =
                    u_previous[i][j]
                    + (omega / 4.0) * (
                        u_current[i - 1][j] +
                        u_current[i + 1][j] +
                        u_current[i][j - 1] +
                        u_current[i][j + 1]
                        - 4.0 * u_previous[i][j]
                    );
            }
        }
    }
}
```

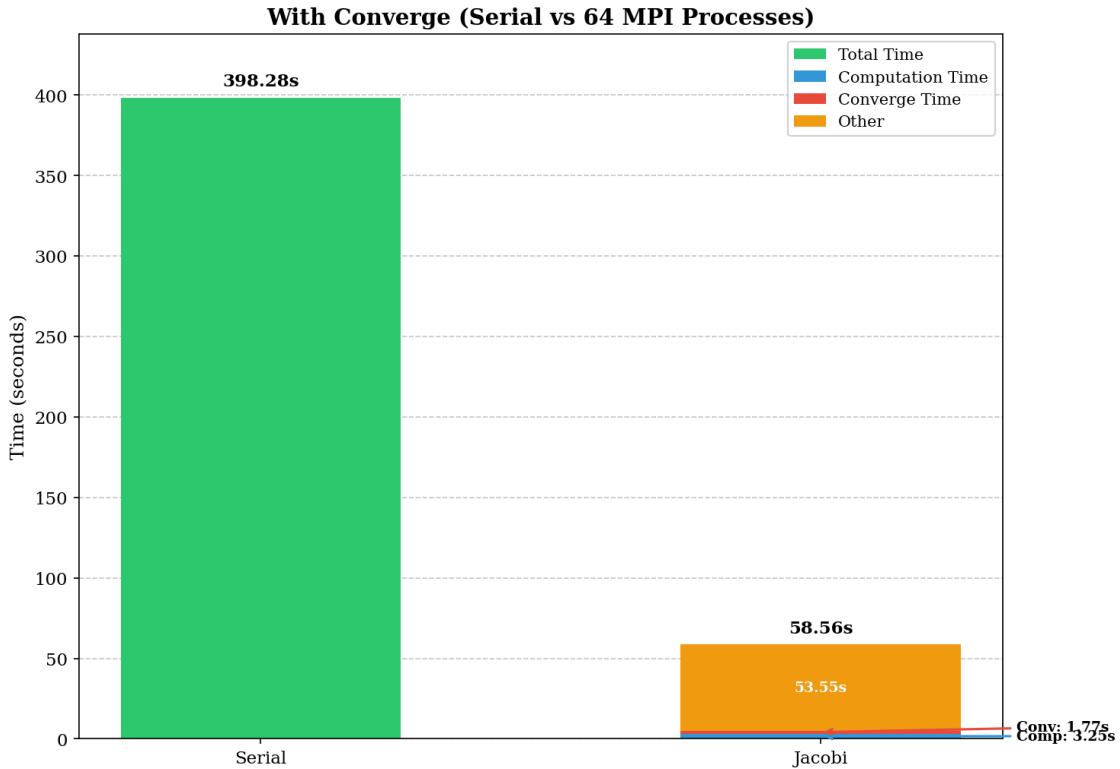
Περιβάλλον Εκτέλεσης

Για τη μεταγλώττιση και εκτέλεση των παράλληλων προγραμμάτων MPI απαιτείται η κατάλληλη ρύθμιση του περιβάλλοντος μέσω του συστήματος modules. Συγκεκριμένα, πριν τη μεταγλώττιση φορτώνεται το module του MPI (**openmpi/1.8.3**), το οποίο παρέχει τον απαραίτητο compiler wrapper και τις βιβλιοθήκες για την ορθή δημιουργία των εκτελέσιμων αρχείων. Η μεταγλώττιση πραγματοποιείται με ενεργοποιημένες βελτιστοποιήσεις (-O3), ώστε να επιτευχθεί καλύτερη απόδοση του υπολογιστικού πυρήνα.

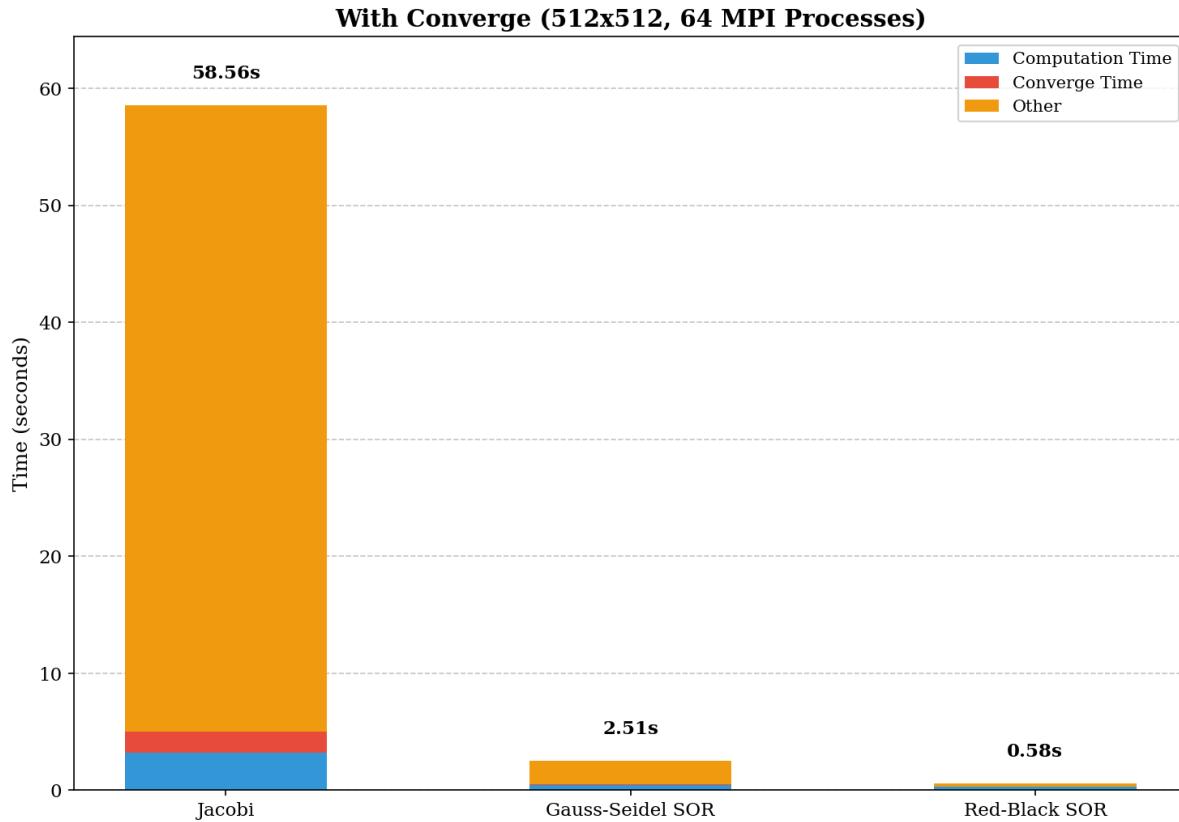
Κατά την εκτέλεση των προγραμμάτων χρησιμοποιείται η εντολή **mpirun** με το argument **--mca btl tcp,self**, ώστε να ρυθμιστεί ρητά η point-to-point μετακίνηση των δεδομένων μέσα στο δίκτυο. Με τη ρύθμιση αυτή αποφεύγεται η χρήση του shared-memory Byte Transfer Layer (sm BTL), το οποίο ενεργοποιείται για την επικοινωνία διεργασιών που εκτελούνται στον ίδιο κόμβο. Η χρήση του sm BTL προκαλεί καθυστερήσεις σε περιβάλλοντα όπου το directory που χρησιμοποιεί βρίσκεται σε network filesystem, όπως συμβαίνει στους κόμβους clones. Για τον λόγο αυτό, η επικοινωνία πραγματοποιείται αποκλειστικά μέσω TCP, εξασφαλίζοντας πιο σταθερή και αξιόπιστη συμπεριφορά κατά τη λήψη των μετρήσεων επίδοσης.

Μετρήσεις με έλεγχο σύγκλισης

Στο πλαίσιο της αξιολόγησης της παράλληλης υλοποίησης του υπολογισμού διάδοσης θερμότητας σε 2 διαστάσεις, πραγματοποιήθηκαν μετρήσεις με ενεργό έλεγχο σύγκλισης για μέγεθος πίνακα 512 x 512. Οι μετρήσεις εκτελέστηκαν τόσο για τη σειριακή υλοποίηση όσο και για τις παράλληλες υλοποίησεις σε περιβάλλον MPI με 64 διεργασίες. Για κάθε περίπτωση καταγράφηκαν ο συνολικός χρόνος εκτέλεσης, ο καθαρός χρόνος υπολογισμών, καθώς και ο χρόνος που αφιερώνεται στον έλεγχο σύγκλισης, με στόχο την κατανόηση της επίδρασης της παραλληλοποίησης και της επικοινωνίας στην τελική επίδοση.



Από το παραπάνω διάγραμμα, βλέπουμε ξεκάθαρα ότι η παράλληλη υλοποίηση Jacobi με 64 MPI διεργασίες μειώνει σημαντικά τον συνολικό χρόνο εκτέλεσης σε σχέση με τη σειριακή εκδοχή. Ο χρόνος καθαρών υπολογισμών είναι ελάχιστος, ενώ παρατηρείται επιβάρυνση από την επικοινωνία και τον έλεγχο σύγκλισης. Ωστόσο, η διαφορά στην ταχύτητα σύγκλισης με τη χρήση των δύο SOR αλγορίθμων ήταν ακόμα πιο εμφανής:



Η **Jacobi** υλοποίηση είναι μακράν η χειρότερη με συνολικό χρόνος στα 58.56s. Το μεγαλύτερο μέρος είναι του χρόνου εκτέλεσης είναι γενικό overhead επικοινωνίας-MPI και όχι καθαρός υπολογισμός. Αυτό είναι αναμενόμενο γιατί η Jacobi συνήθως χρειάζεται πολύ περισσότερες επαναλήψεις για να φτάσει στο ίδιο κριτήριο σύγκλισης, άρα “πληρώνει” πολλές φορές τα communication/sync κόστη.

Η **Gauss-Seidel SOR** έρχεται να βελτιώσει αισθητά το convergence time στα 2.51s συνολικά. Παρότι κάθε επανάληψη μπορεί να είναι ακριβότερη από Jacobi, χρειάζεται λιγότερες επαναλήψεις για σύγκλιση, άρα μειώνει δραστικά το συνολικό communication που απαιτείται.

Τέλος, η **Red-Black SOR** είναι η καλύτερη από τις υλοποιήσεις μας. Παρά το ότι η συγκεκριμένη υλοποίηση έχει περισσότερα στάδια ανά iteration (red + black), τα οποία την καθιστούν πιο compute-intensive και εισάγουν περαιτέρω overhead επικοινωνίας, φαίνεται να έχει τον ελάχιστο χρόνο σύγκλισης επειδή απαιτεί ακόμη λιγότερες συνολικές επαναλήψεις.

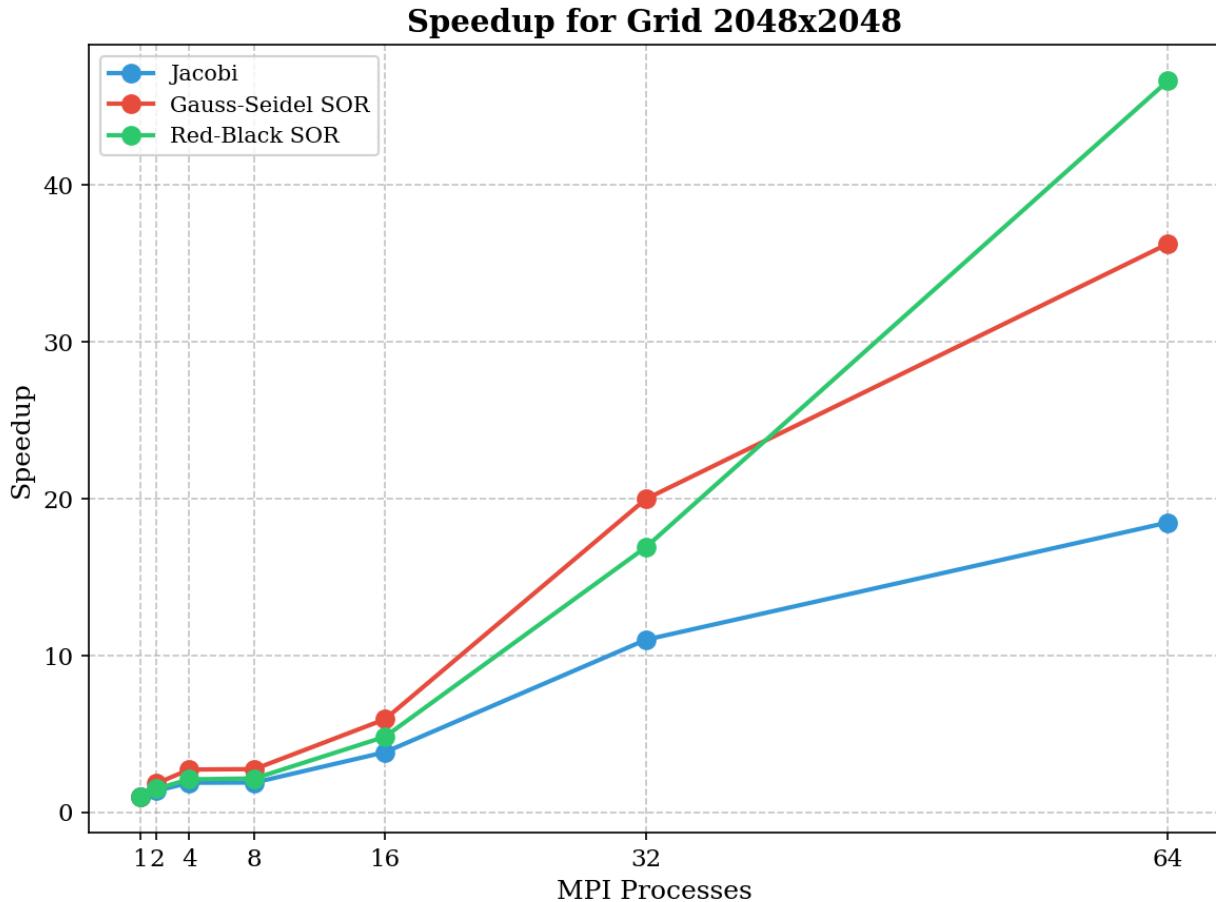
Συμπεραίνουμε λοιπόν ότι για να έχουμε την γρηγορότερη δυνατή σύγκλιση θα επιλέγαμε την Red-Black SOR υλοποίηση, η οποία φαίνεται να κάνει τη διαφορά στο συγκεκριμένο configuration του προβλήματος διάδοσης θερμότητας.

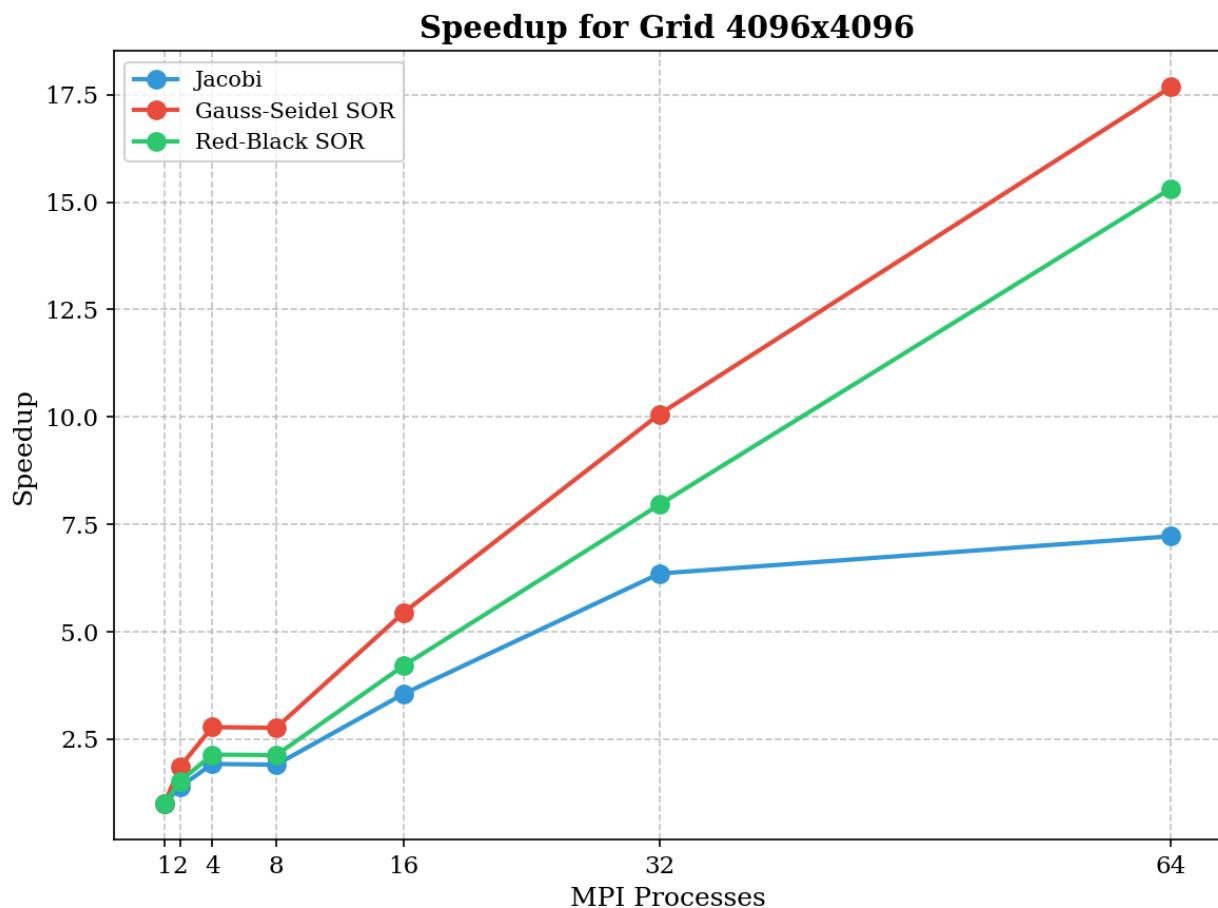
Μετρήσεις χωρίς έλεγχο σύγκλισης

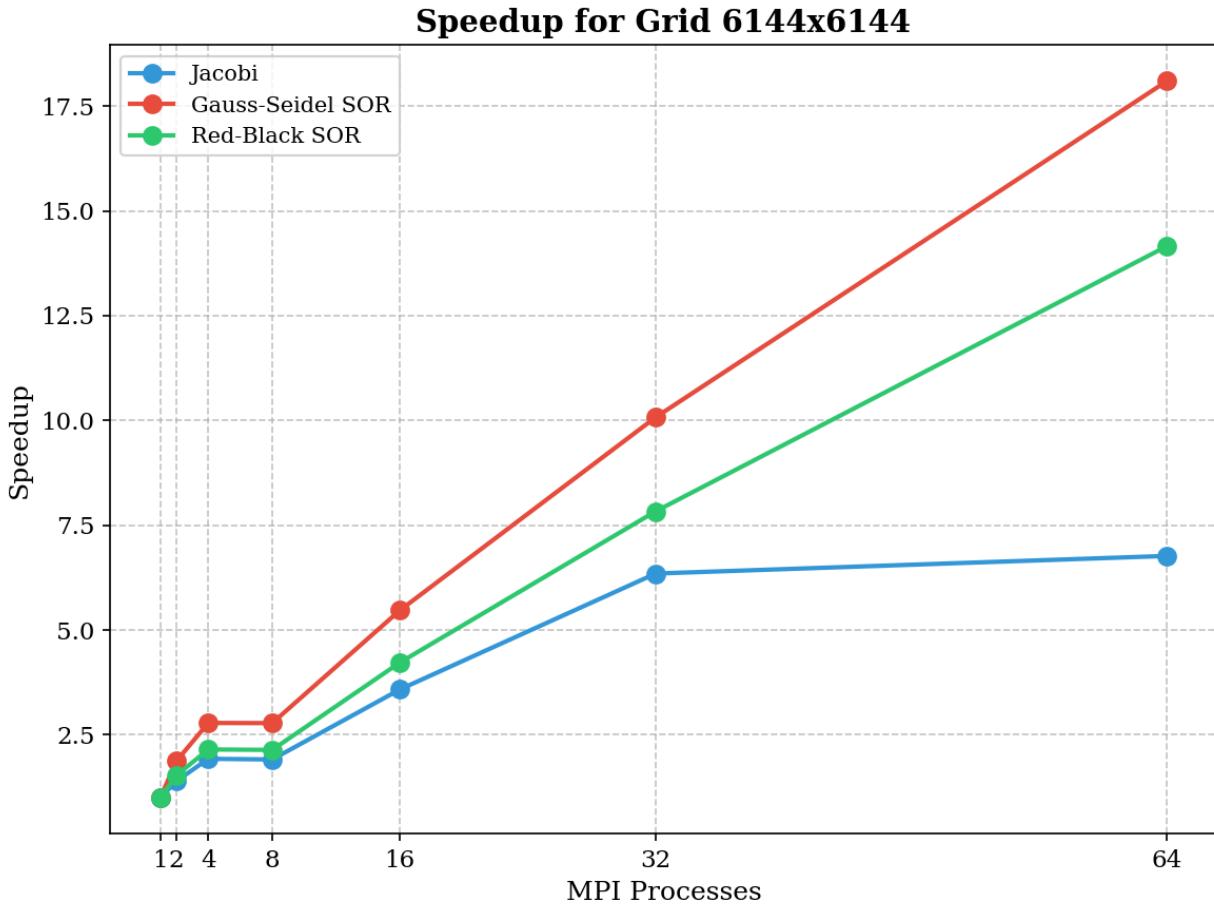
Στη συνέχεια πραγματοποιήθηκαν μετρήσεις επίδοσης χωρίς έλεγχο σύγκλισης, με σκοπό την αξιολόγηση της κλιμακωσιμότητας της παράλληλης υλοποίησης. Ο αλγόριθμος εκτελέστηκε για σταθερό αριθμό επαναλήψεων ($T=256$), ώστε όλες οι εκτελέσεις να είναι άμεσα συγκρίσιμες. Οι

μετρήσεις πραγματοποιήθηκαν για μεγέθη πινάκων 2048x2048, 4096x4096 και 6144x6144, και για διαφορετικό αριθμό MPI διεργασιών (1, 2, 4, 8, 16, 32 και 64). Για κάθε περίπτωση καταγράφηκαν ο συνολικός χρόνος εκτέλεσης και ο καθαρός χρόνος υπολογισμών, επιτρέποντας την ανάλυση της επίδρασης της παραλληλοποίησης και της επικοινωνίας στην απόδοση του εκάστοτε αλγορίθμου.

Παραθέτουμε διάγραμμα επιτάχυνσης για κάθε μέγεθος πίνακα, που αποτυπώνει τις επιδόσεις καθενός εκ των τριών εξεταζόμενων αλγορίθμων:







Από τα διαγράμματα επιτάχυνσης, παρατηρείται ότι και οι τρεις παράλληλες υλοποιήσεις (Jacobi, Gauss-Seidel SOR, Red-Black SOR) παρουσιάζουν βελτίωση της επίδοσης καθώς αυξάνεται ο αριθμός των MPI διεργασιών (Strong Scaling).

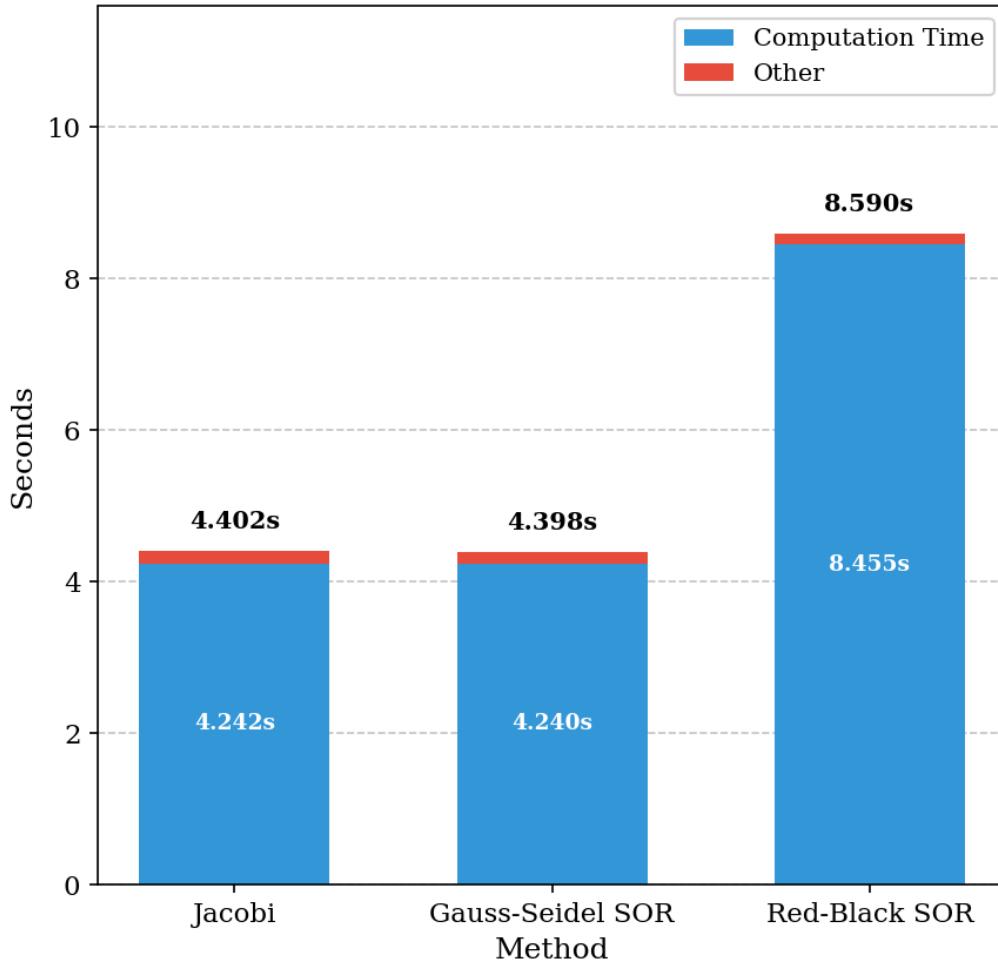
Συγκριτικά ανά μέγεθος προβλήματος, διαπιστώνεται ότι για τα μεγαλύτερα μεγέθη πίνακα (π.χ. 6144x6144) η επιτάχυνση τείνει να διατηρείται πιο κοντά στην ιδανική γραμμική αύξηση σε σχέση με τα μικρότερα μεγέθη (2048x2048). Αυτό συμβαίνει διότι στα μεγάλα προβλήματα ο λόγος των υπολογισμών προς την MPI-επικοινωνία είναι μεγαλύτερος, με κάθε διεργασία να έχει επαρκή όγκο υπολογιστικού φόρτου που υπερκαλύπτει το κόστος της ανταλλαγής μηνυμάτων. Αντίθετα, στα μικρότερα μεγέθη, καθώς αυξάνονται οι διεργασίες, το overhead της επικοινωνίας και του συγχρονισμού κυριαρχεί νωρίτερα, οδηγώντας σε ταχύτερο κορεσμό της καμπύλης επιτάχυνσης.

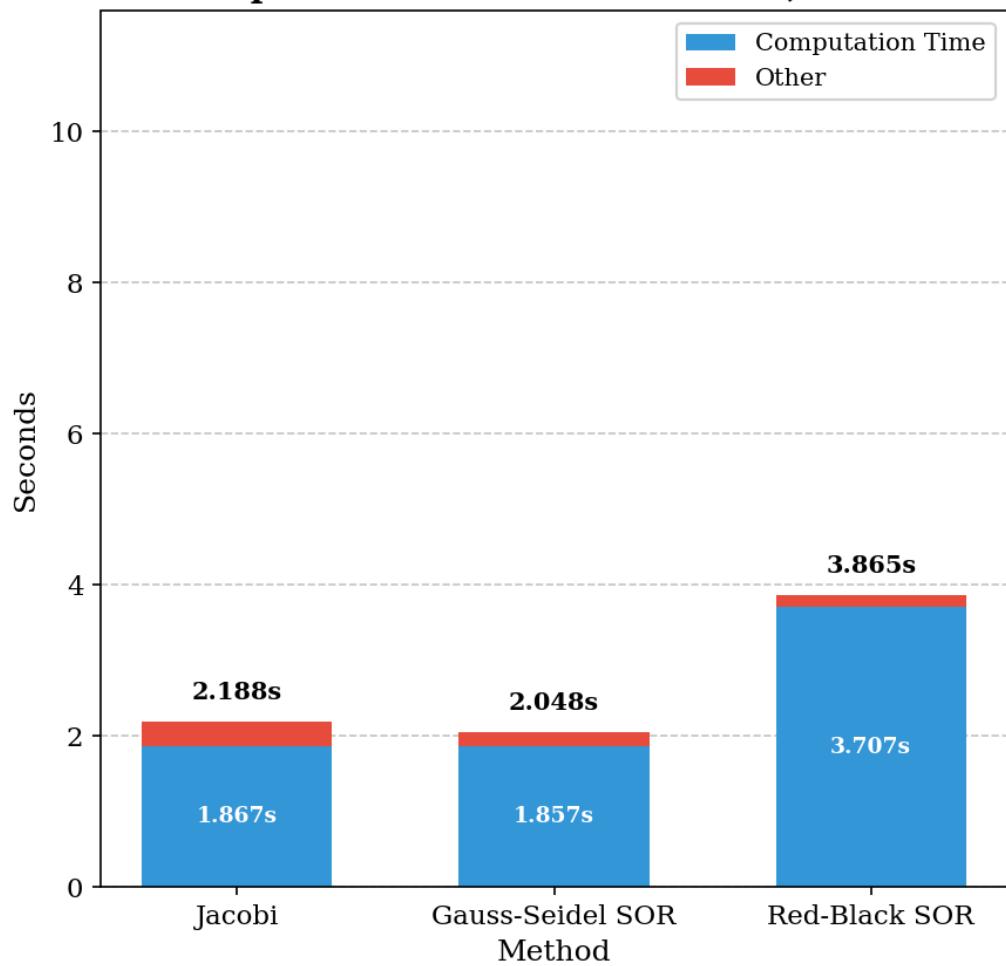
Μεταξύ των μεθόδων, η Jacobi εμφανίζει συχνά την πιο "καθαρή" κλιμάκωση λόγω της απλότητάς της (μία φάση επικοινωνίας). Η Red-Black SOR, παρόλο που αποτελεί αποδοτική μέθοδο ως προς τη σύγκλιση, φαίνεται να εμφανίζει ελαφρώς χαμηλότερο speedup στις πολλές διεργασίες, διότι απαιτεί δύο φάσεις συγχρονισμού και δύο ανταλλαγές δεδομένων (halo exchanges) ανά επανάληψη, αυξάνοντας την επιρροή του latency του δικτύου στην υλοποίηση.

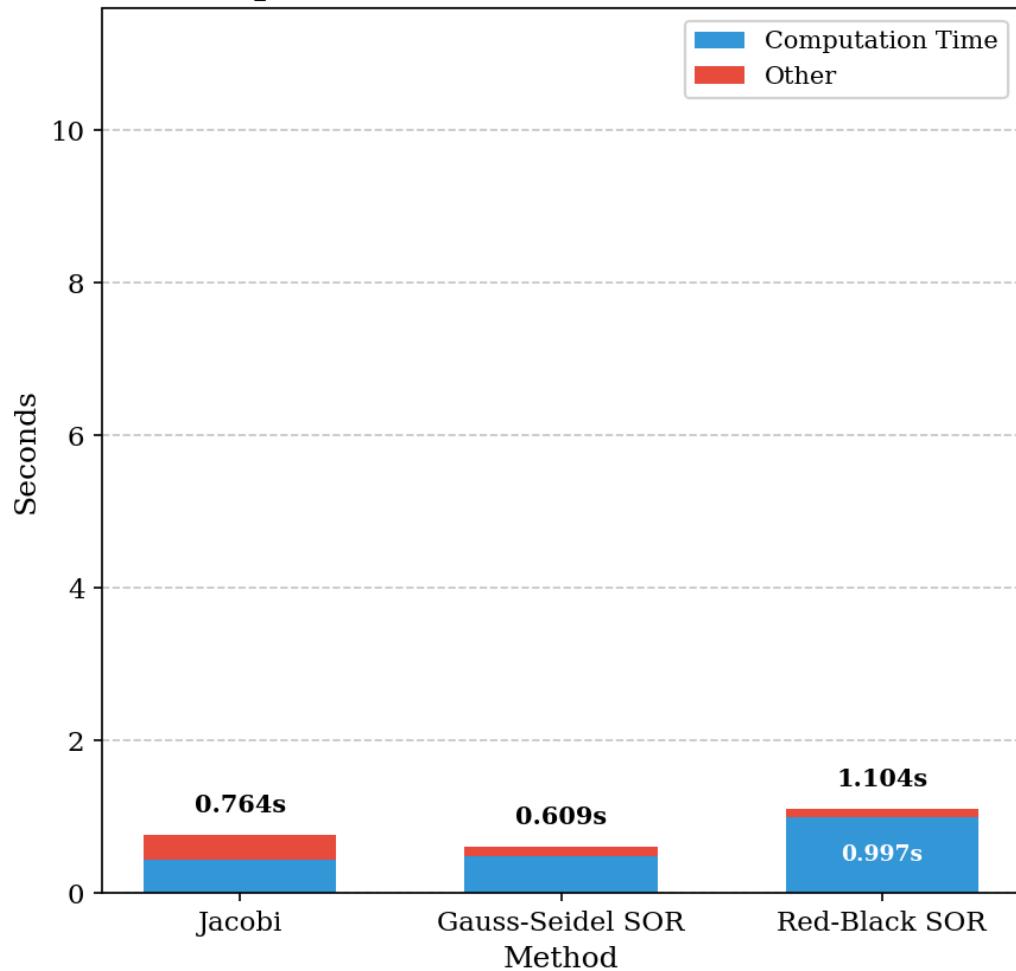
Στη συνέχεια παραθέτουμε και τα barplots που αποτυπώνουν τους συνολικούς χρόνους εκτέλεσης και τους καθαρούς χρόνους υπολογισμών για σταθερό αριθμό επαναλήψεων.

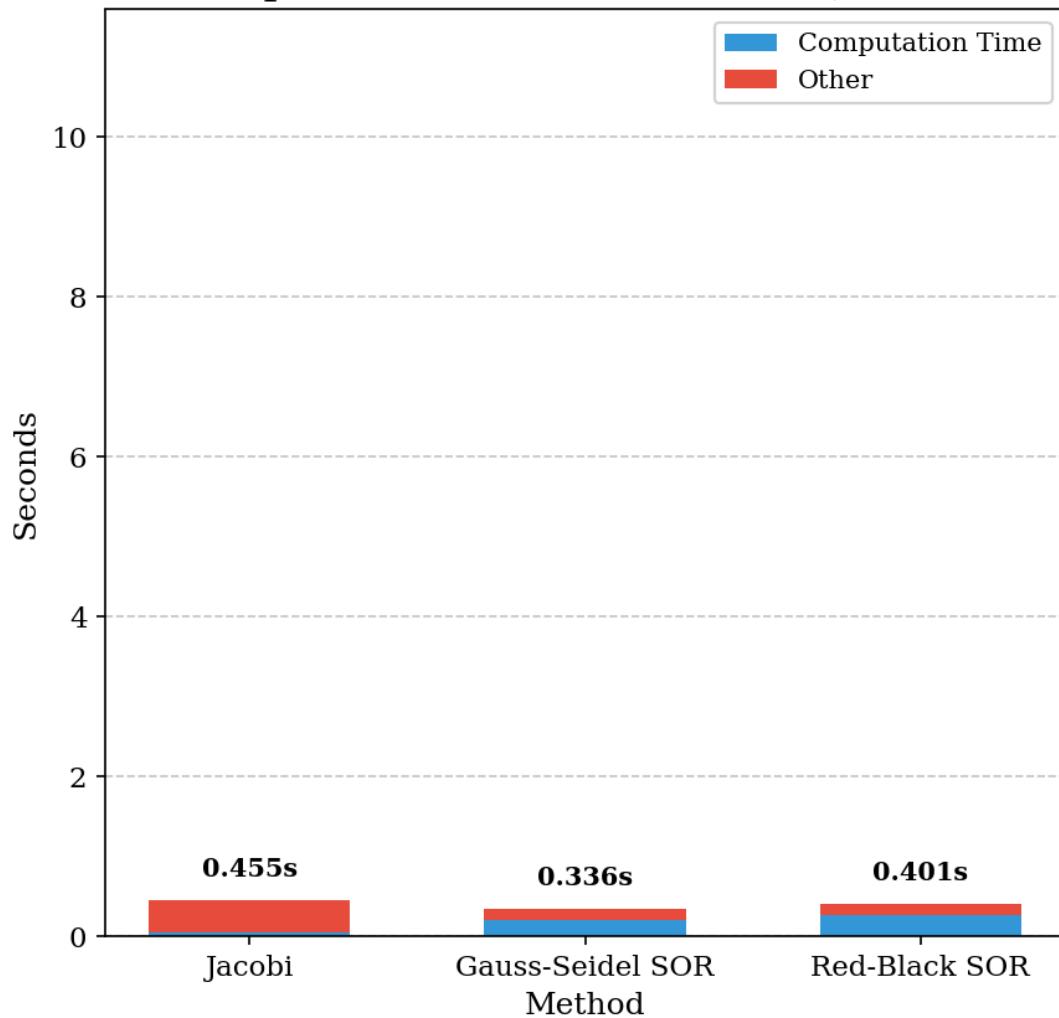
Για μέγεθος πίνακα 2048x2048:

Total vs Computation Time - Size = 2048, MPI Proc. = 8

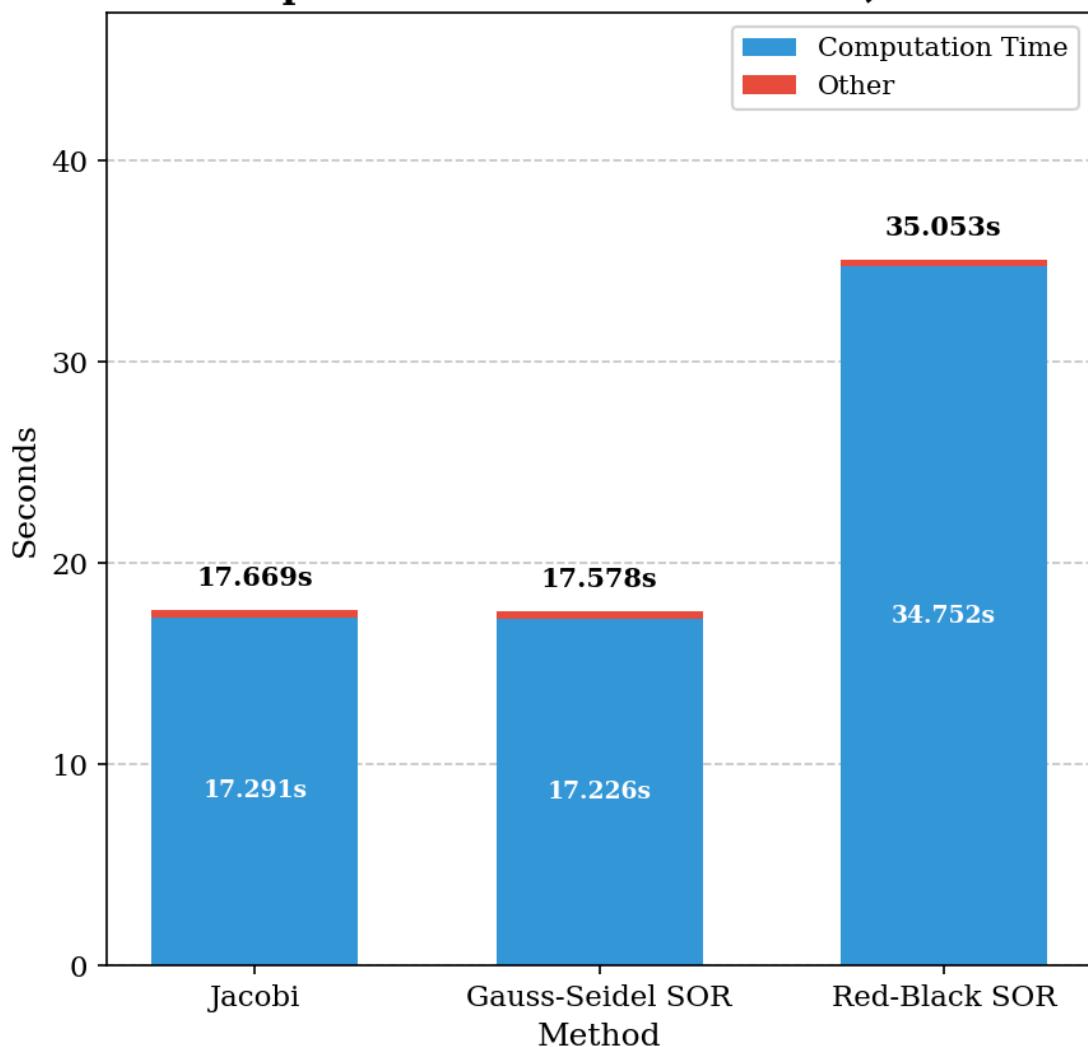


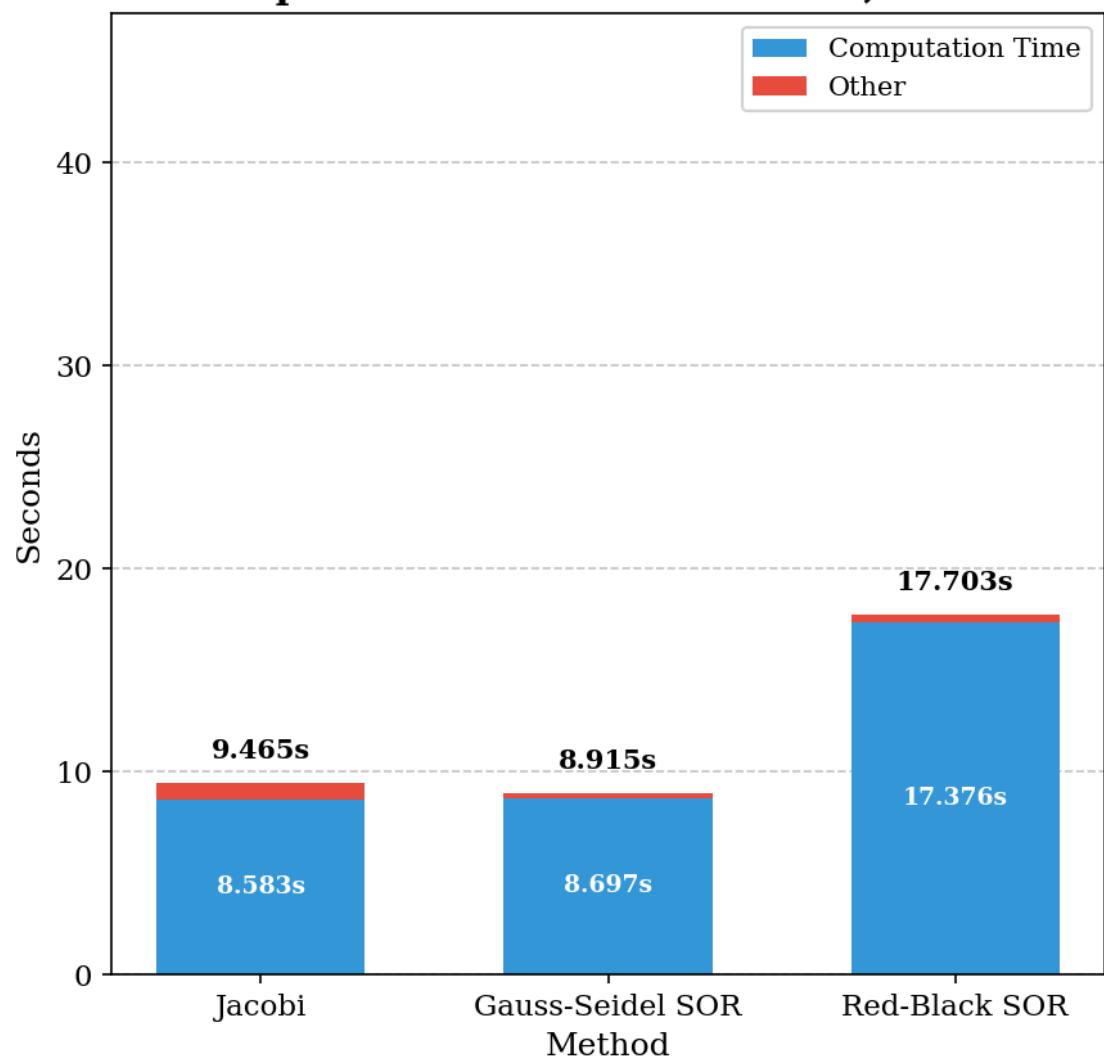
Total vs Computation Time - Size = 2048, MPI Proc. = 16

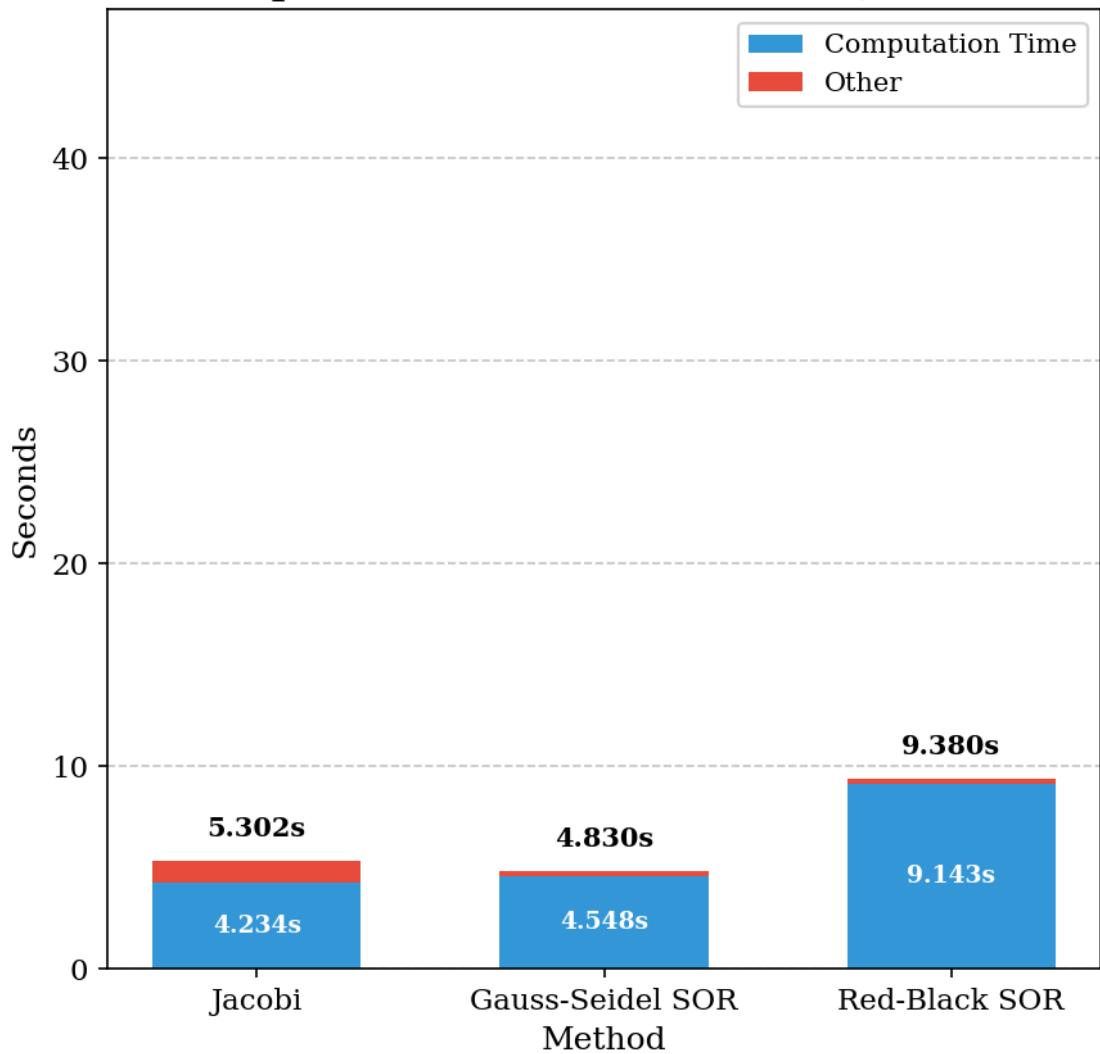
Total vs Computation Time - Size = 2048, MPI Proc. = 32

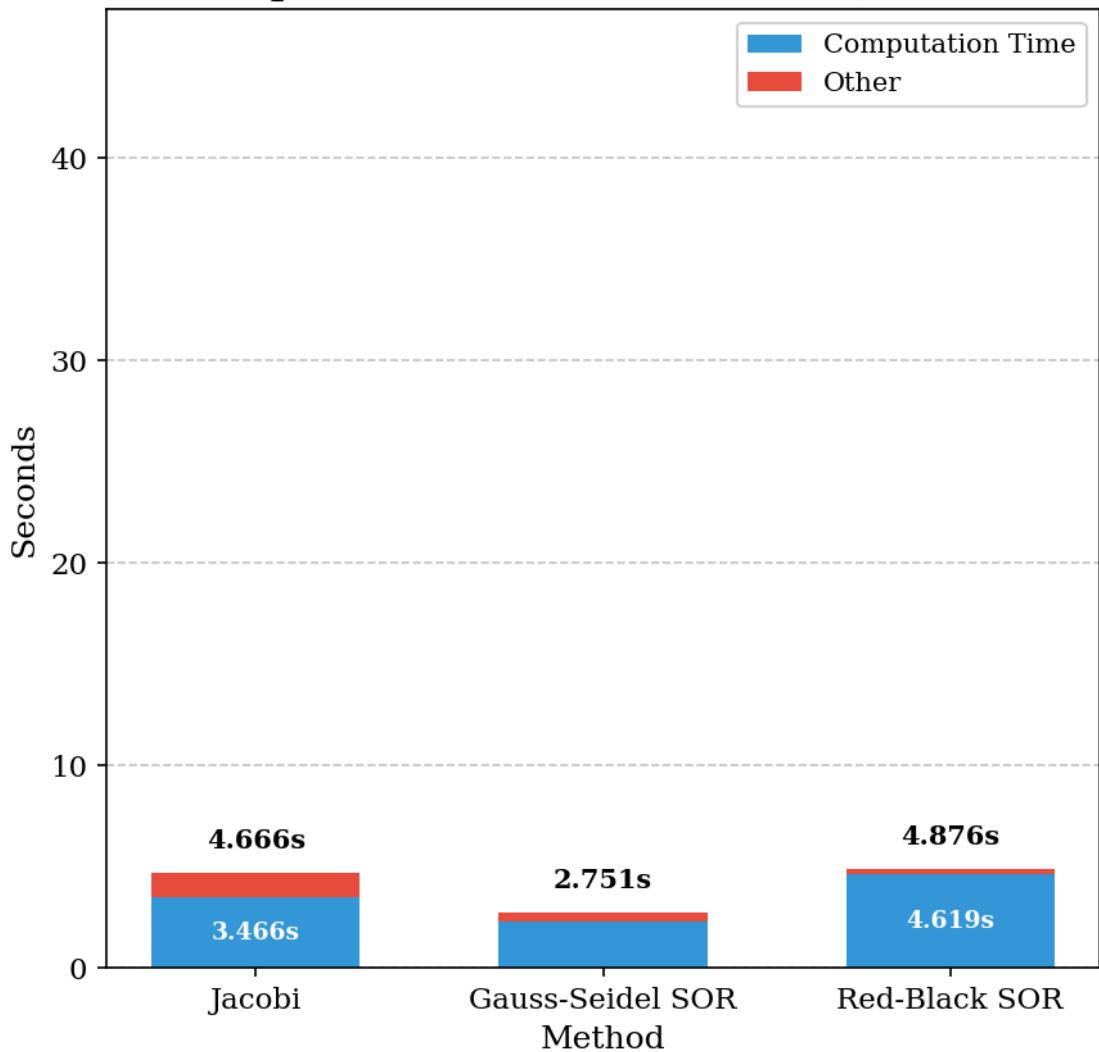
Total vs Computation Time - Size = 2048, MPI Proc. = 64

Για μέγεθος πίνακα 4096x4096:

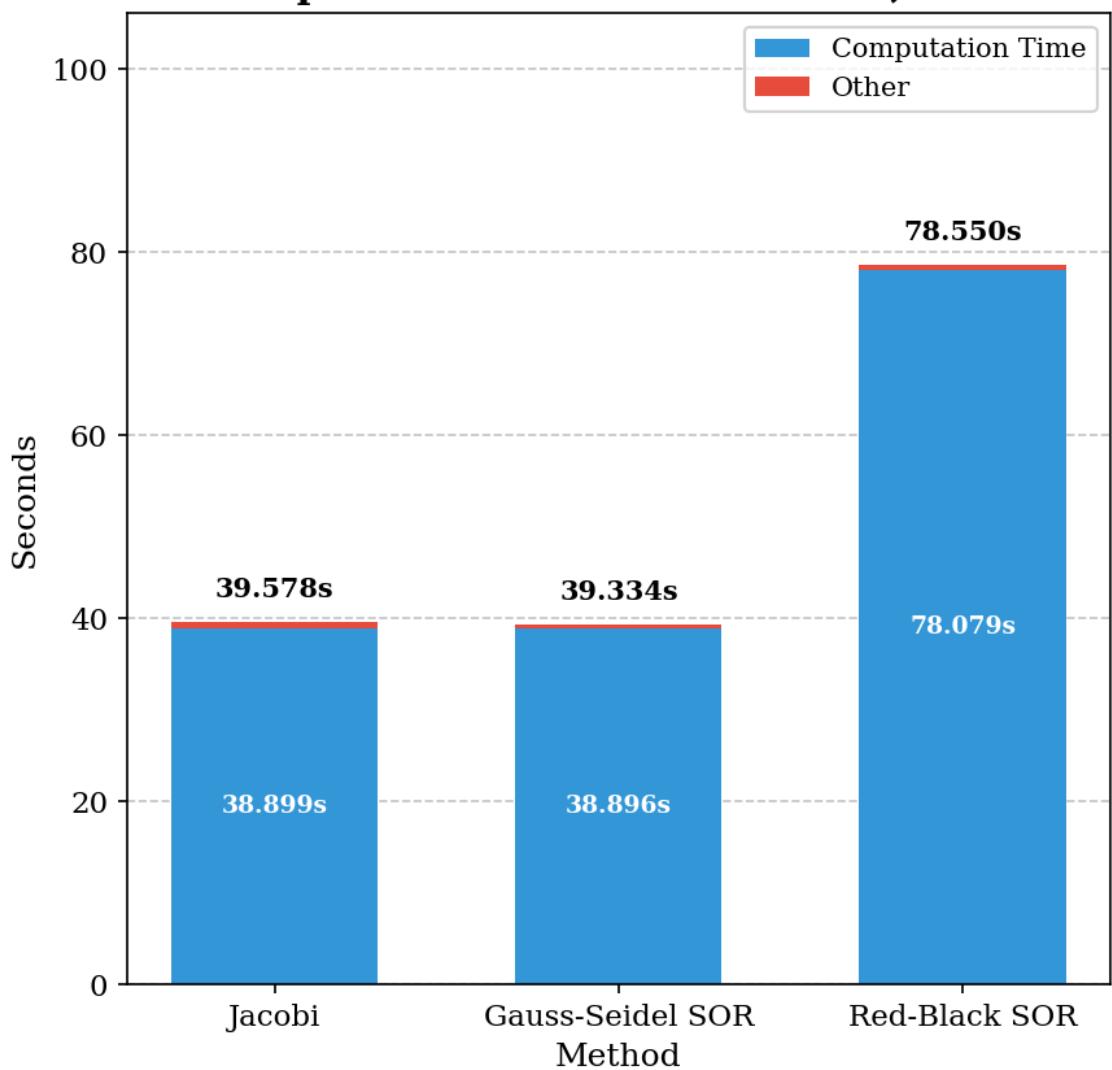
Total vs Computation Time - Size = 4096, MPI Proc. = 8

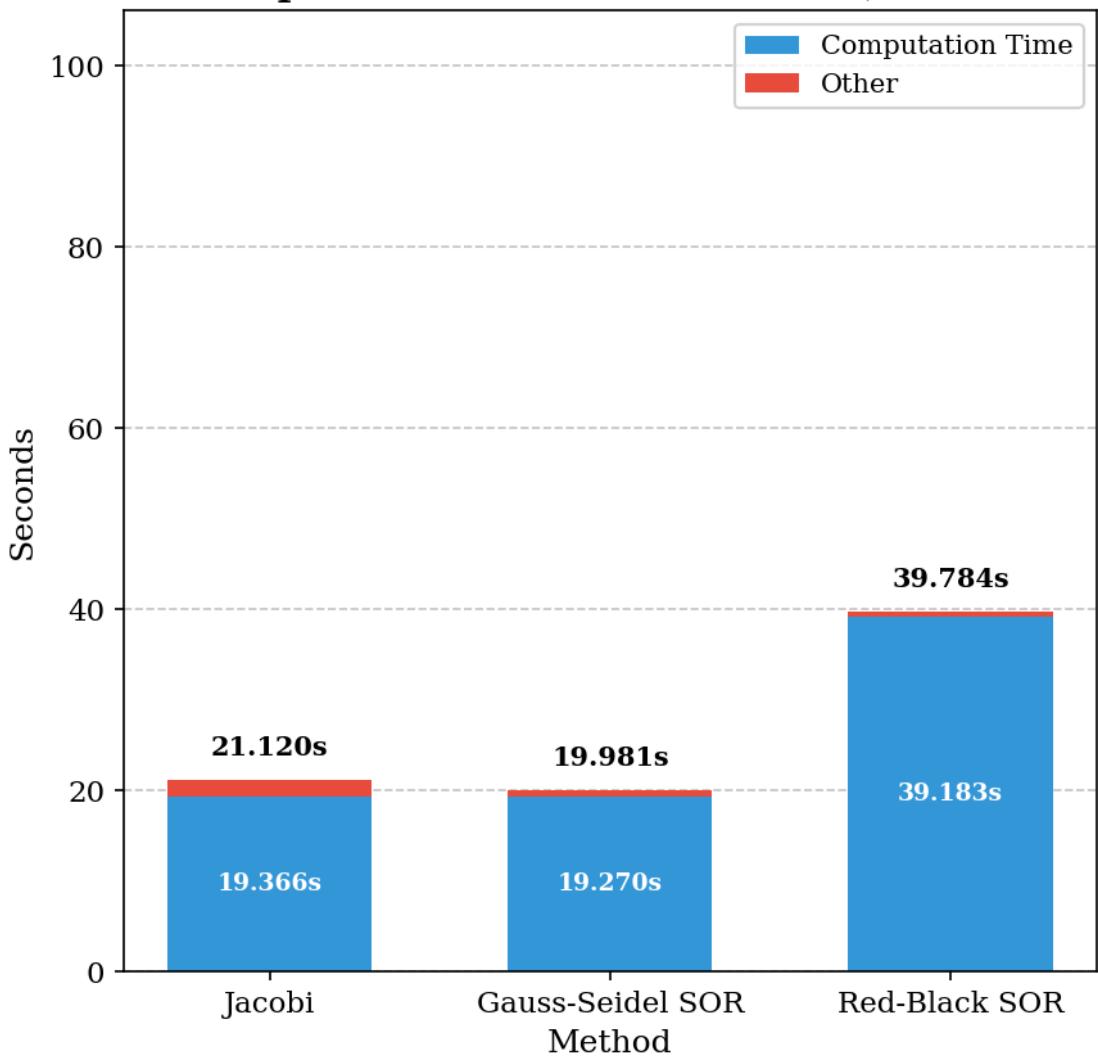
Total vs Computation Time - Size = 4096, MPI Proc. = 16

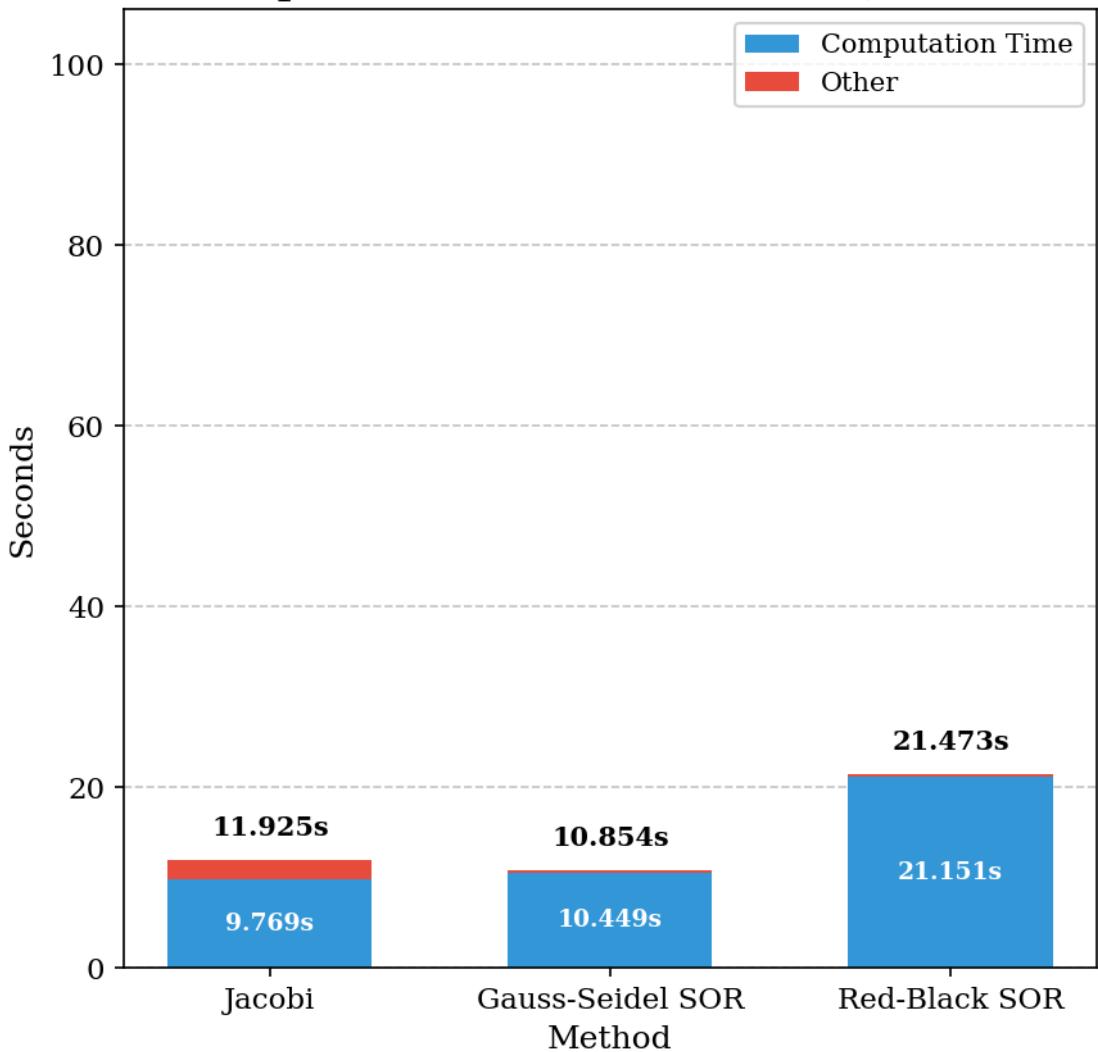
Total vs Computation Time - Size = 4096, MPI Proc. = 32

Total vs Computation Time - Size = 4096, MPI Proc. = 64

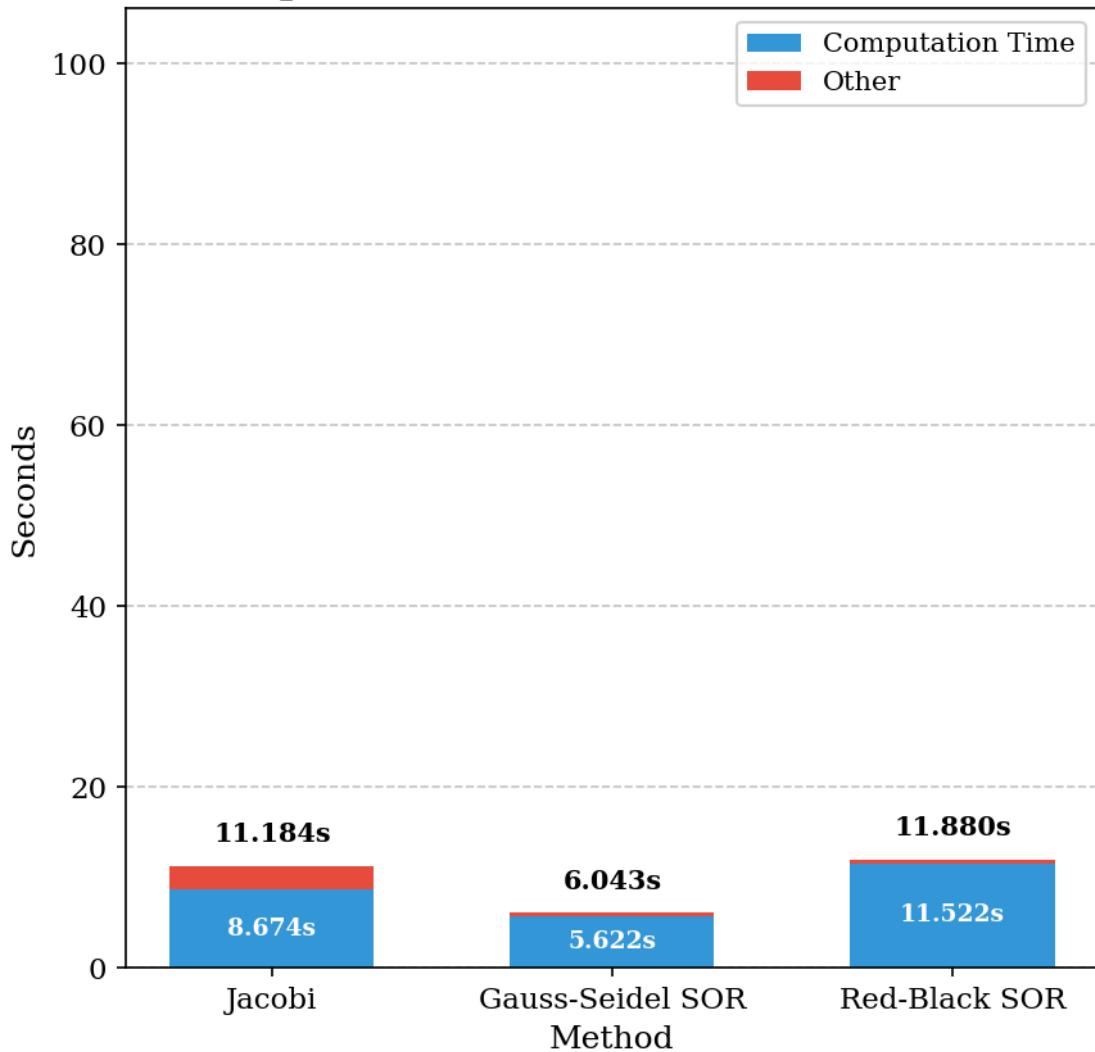
Για μέγεθος πίνακα 6144x6144:

Total vs Computation Time - Size = 6144, MPI Proc. = 8

Total vs Computation Time - Size = 6144, MPI Proc. = 16

Total vs Computation Time - Size = 6144, MPI Proc. = 32

Total vs Computation Time - Size = 6144, MPI Proc. = 64



Στα παραπάνω barplots παρατηρείται ότι με την αύξηση του αριθμού των MPI διεργασιών ο καθαρός χρόνος υπολογισμών μειώνεται, ενώ ο χρόνος που αντιστοιχεί σε επικοινωνία και συγχρονισμό αυξάνεται. Καθώς μεταβαίνουμε από τις 8 στις 64 διεργασίες, παρατηρούμε ότι το "κόκκινο" τμήμα (Other) καταλαμβάνει ολοένα και μεγαλύτερο ποσοστό του συνολικού χρόνου, επιβεβαιώνοντας ότι η εκτέλεση καθίσταται σταδιακά communication-bound.

Παρατηρούμε ωστόσο, σε αντίθεση με το πείραμα που ελέγχαμε τη σύγκλιση των προγραμμάτων, ότι η μέθοδος η οποία έχει συστηματικά τον μικρότερο συνολικό χρόνο για το σταθερό πλήθος επαναλήψεων είναι η Jacobi. Αυτό συμβαίνει καθώς ο υπολογιστικός της πυρήνας είναι απλούστερος και απαιτεί μόνο μία ανταλλαγή δεδομένων ανά iteration, με αποτέλεσμα σε εκτέλεση για τον ίδιο αριθμό επαναλήψεων να είναι η καλύτερη μέθοδος. Αντίθετα, η Red-Black SOR εμφανίζει τους υψηλότερους χρόνους στο συγκεκριμένο σενάριο,

καθώς κάθε επανάληψη αποτελείται από δύο υπολογιστικά υπο-βήματα (Red και Black) και δύο διακριτές διαδικασίες επικοινωνίας, αυξάνοντας το συνολικό overhead.

Τέλος, η Gauss-Seidel SOR κινείται συνήθως ανάμεσα στις δύο άλλες μεθόδους ως προς τον χρόνο ανά επανάληψη. Έχει παραπλήσιο μοτίβο επικοινωνίας με την Jacobi, αλλά ελαφρώς πιο σύνθετο υπολογιστικό πυρήνα, γεγονός που αντανακλάται στους χρόνους.

Συνοψίζοντας, σε πειράματα σταθερών επαναλήψεων, η Jacobi είναι η γρηγορότερη, αφού έχει τον ελάχιστο χρόνο εκτέλεσης ανά iteration. Ωστόσο, όπως έδειξε η ανάλυση με σύγκλιση, η Red-Black SOR υπερτερεί τελικά σε πραγματικό χρόνο λύσης διότι, παρόλο που είναι πιο αργή ανά επανάληψη, απαιτεί δραματικά λιγότερες επαναλήψεις για να συγκλίνει σε λύση.