



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Συστήματα Παράλληλης Επεξεργασίας

Εξαμηνιαία Αναφορά

Ομάδα: parlab08
Στέφανος Γιαννακόπουλος (03120829)
Παρασκευή Κασιούμη (03120122)
Κωνσταντίνα Παπία (03120075)

2024-25

Άσκηση 1

Εξοικείωση με το περιβάλλον προγραμματισμού

Στην άσκηση 1, παραλληλοποιήσαμε το πρόγραμμα Παιχνίδι της Ζωής (*Conway's Game of Life*) στο μοντέλο κοινού χώρου διευθύνσεων (shared address space) με τη χρήση του OpenMP και πραγματοποιήσαμε μετρήσεις επίδοσης για τα μεγέθη πυρήνων 1,2,4,6,8 και τα μεγέθη ταμπλώ 64×64 , 1024×1024 και 4096×4096 . Σε όλες τις περιπτώσεις τρέξαμε το πρόγραμμα για 1000 γενιές.

Η παραλληλοποίηση υλοποιήθηκε στο ακόλουθο σημείο του προγράμματος:

```
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for shared(N, previous, current) private(nbros, i, j)
    for ( i = 1 ; i < N-1 ; i++ )
        for ( j = 1 ; j < N-1 ; j++ ) {
            nbros = 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 ( nbros == 3 || ( previous[i][j]+nbros ==3 ) )
                current[i][j]=1;
            else
                current[i][j]=0;
        }
}
```

Συγκεκριμένα, προσθέσαμε την γραμμή:

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

Με αυτό το directive, το for loop `for (i = 1 ; i < N-1 ; i++) {...}` εκτελείται παράλληλα από διαφορετικά threads. Οι μεταβλητές `previous` και `current` που αναπαριστούν το ταμπλώ, αλλά και η μεταβλητή `N`, με την οποία ορίζουμε τη διάσταση του ταμπλώ, είναι κοινές (shared) για όλα τα threads. Η μεταβλητή `i` ως index του παραλληλοποιούμενου for loop είναι εξ' ορισμού `private` για κάθε νήμα. Ορίζουμε, επιπλέον, και τον δείκτη `j` ως `private` μεταβλητή κάθε νήματος, ώστε να αποφευχθούν race conditions και η διάσχιση - ανάγνωση του ταμπλώ `previous` να πραγματοποιηθεί ορθά και ανεξάρτητα από κάθε νήμα. Θέτοντας το `j` ως `private` μεταβλητή κάθε νήματος, αποφεύγουμε και τον κίνδυνο race conditions κατά την εγγραφή στον πίνακα `current`. Τέλος, και η μεταβλητή `nbros` πρέπει να οριστεί ως `private` για κάθε νήμα, διότι σε κάθε iteration πραγματοποιούμε εγγραφή σε αυτήν τη μεταβλητή, ώστε να μετράμε το πλήθος νεκρών ή ζωντανών γειτόνων κάθε σημείου (i, j) του ταμπλώ.

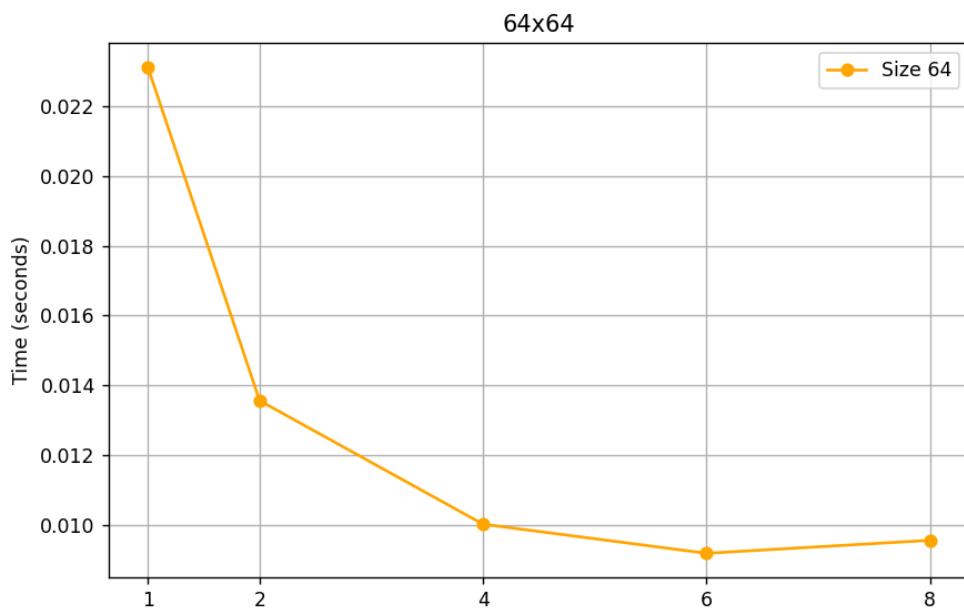
Το πλήθος των νημάτων που θα δημιουργηθούν ορίζεται με τη χρήση environment variable στο terminal. Για παράδειγμα, γράφοντας το command `export OMP_NUM_THREADS=4`, ορίζουμε ότι τα threads που θα δημιουργηθούν θα είναι 4.

Οι χρόνοι εκτέλεσης που προέκυψαν είναι οι ακόλουθοι:

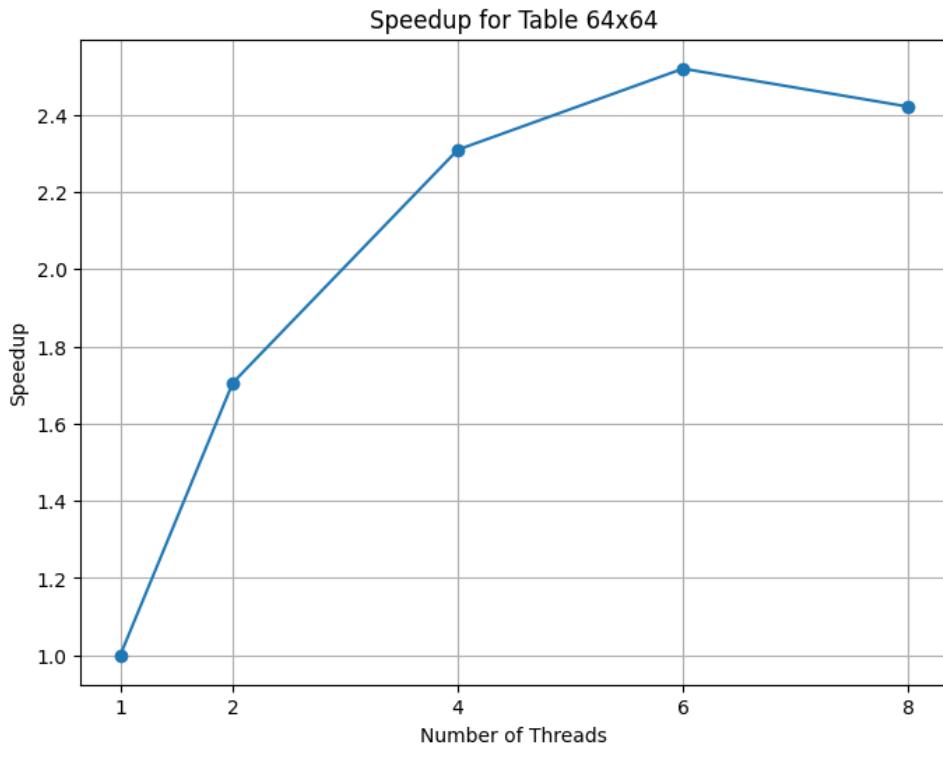
```
parlab08@scirouter:~/2024-2025/a1$ ls
Game_Of_Life    get_page_size.c      make_Game_Of_Life.out  run_game_of_life.out
Game_Of_Life.c  Makefile           make_on_queue.sh     run_on_queue.sh
get_page_size   make_Game_Of_Life.err  run_game_of_life.err
parlab08@scirouter:~/2024-2025/a1$ cat run_game_of_life.out
Number of threads: 1 - GameOfLife: Size 64 Steps 1000 Time 0.023119
Number of threads: 2 - GameOfLife: Size 64 Steps 1000 Time 0.013556
Number of threads: 4 - GameOfLife: Size 64 Steps 1000 Time 0.010012
Number of threads: 6 - GameOfLife: Size 64 Steps 1000 Time 0.009178
Number of threads: 8 - GameOfLife: Size 64 Steps 1000 Time 0.009551
Number of threads: 1 - GameOfLife: Size 1024 Steps 1000 Time 10.966436
Number of threads: 2 - GameOfLife: Size 1024 Steps 1000 Time 5.458709
Number of threads: 4 - GameOfLife: Size 1024 Steps 1000 Time 2.723174
Number of threads: 6 - GameOfLife: Size 1024 Steps 1000 Time 1.830095
Number of threads: 8 - GameOfLife: Size 1024 Steps 1000 Time 1.378764
Number of threads: 1 - GameOfLife: Size 4096 Steps 1000 Time 175.893582
Number of threads: 2 - GameOfLife: Size 4096 Steps 1000 Time 88.260482
Number of threads: 4 - GameOfLife: Size 4096 Steps 1000 Time 45.897376
Number of threads: 6 - GameOfLife: Size 4096 Steps 1000 Time 43.695293
Number of threads: 8 - GameOfLife: Size 4096 Steps 1000 Time 43.235918
parlab08@scirouter:~/2024-2025/a1$
```

Τα αποτελέσματα των μετρήσεων αυτών παρουσιάζονται στα ακόλουθα διαγράμματα που δείχνουν τον χρόνο εκτέλεσης του παραλληλοποιημένου προγράμματος συναρτήσει του πλήθους των νημάτων (Threads).

- Διάγραμμα χρόνου εκτέλεσης προγράμματος συναρτήσει των πυρήνων (1.1) και διάγραμμα speedup του προγράμματος συναρτήσει των πυρήνων (1.2) για ταμπλώ μεγέθους 64×64



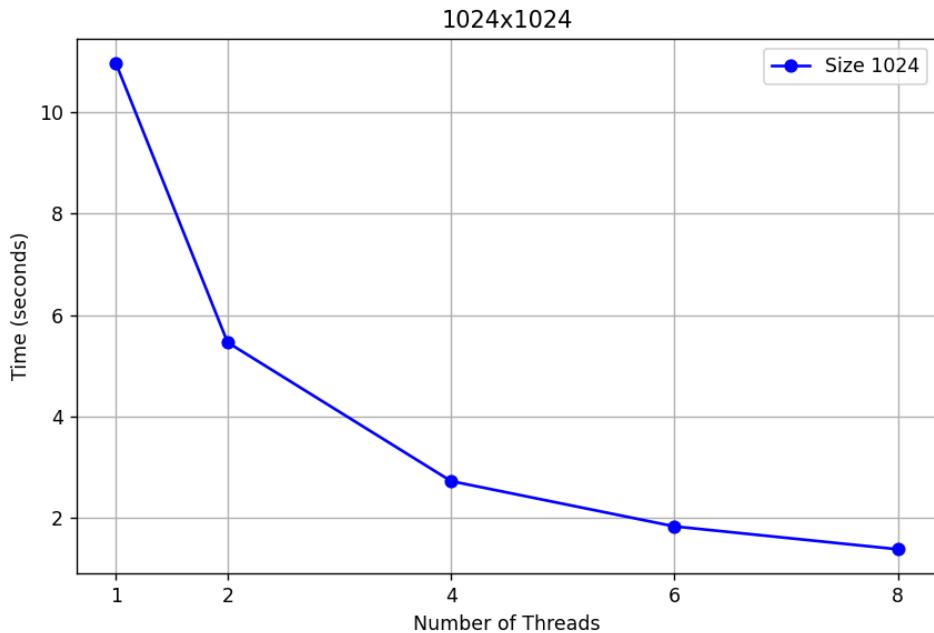
Διάγραμμα 1.1



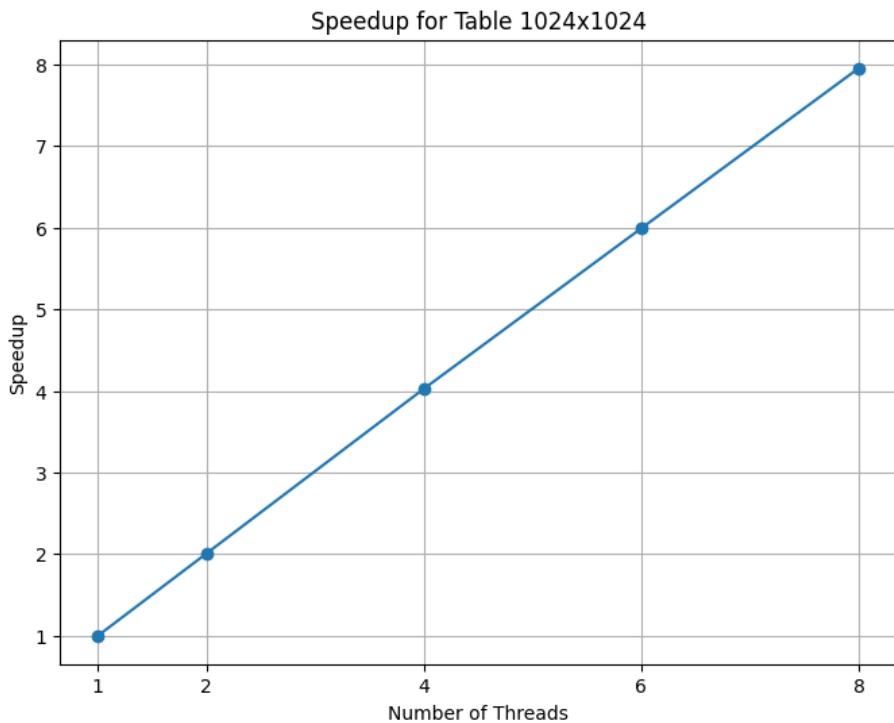
Διάγραμμα 1.2

Στο Διάγραμμα 1.1, παρατηρούμε ότι με την προσθήκη περισσότερων threads, μειώνεται ο χρόνος εκτέλεσης του προγράμματος. Ωστόσο, όταν αυξάνουμε τα threads σε πάνω από 6, διαπιστώνουμε ότι ο χρόνος εκτέλεσης αυξάνεται και το πρόγραμμά μας πλέον δεν κλιμακώνει. Σε ότι αφορά το speedup (Διάγραμμα 1.2), παρατηρούμε ότι αυξάνεται με μικρό βαθμό, ενώ για 6 threads και πάνω μένει σχεδόν σταθερό. Αυτό συμβαίνει, διότι το μέγεθος του ταμπλώ είναι αρκετά μικρό, με αποτέλεσμα το κόστος παραλληλοποίησης (όπως ο συγχρονισμός και η επικοινωνία μεταξύ των νημάτων) να υπερβαίνει το όφελος που προκύπτει από την παραλληλοποίηση. Ως αποτέλεσμα, η βελτίωση στην απόδοση είναι περιορισμένη, καθώς το κόστος της παράλληλης εκτέλεσης επηρεάζει σημαντικά τη συνολική ταχύτητα του προγράμματος.

- Διάγραμμα χρόνου εκτέλεσης προγράμματος συναρτήσει των πυρήνων (1.3) και διάγραμμα speedup του προγράμματος συναρτήσει των νημάτων (1.4) για ταμπλώ μεγέθους 1024×1024



Διάγραμμα 1.3

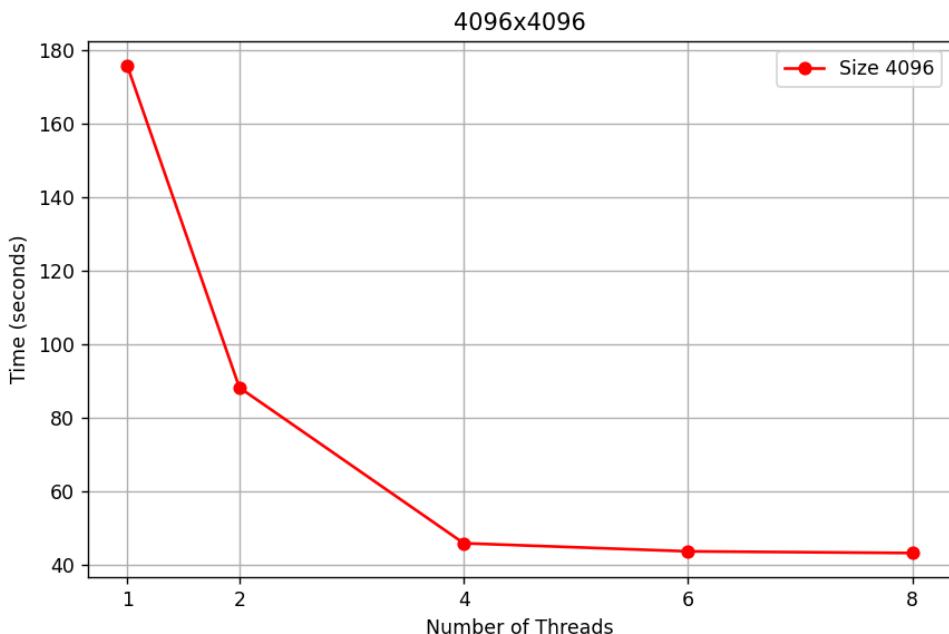


Διάγραμμα 1.4

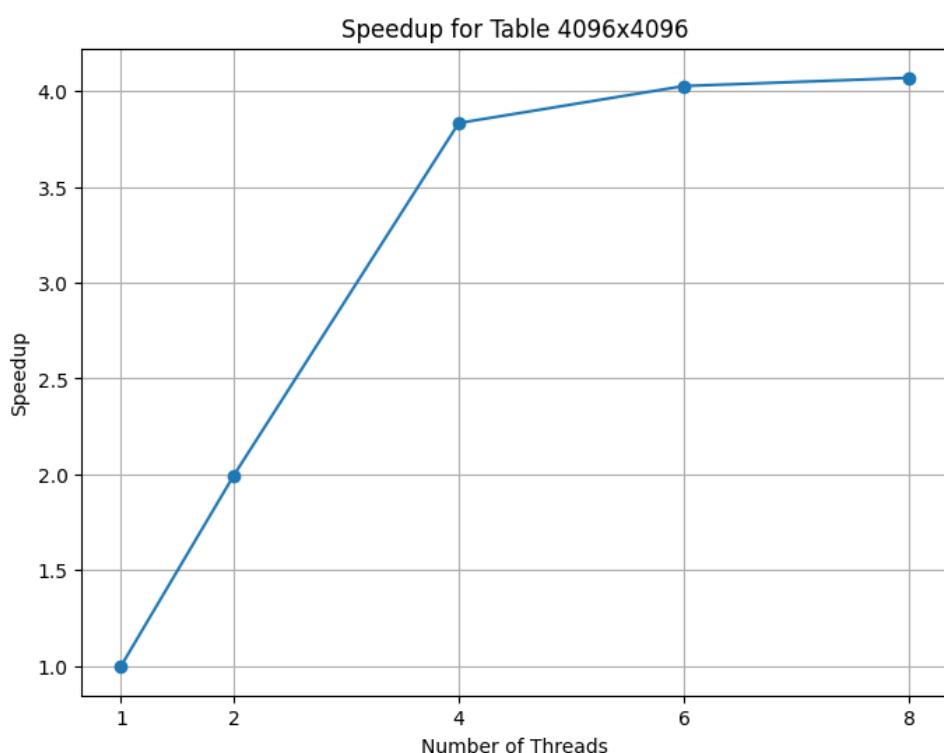
Σε ότι αφορά το Διάγραμμα 1.3, παρατηρούμε ότι ο χρόνος εκτέλεσης του προγράμματος μειώνεται με την προσθήκη περισσότερων threads, καθώς για το συγκεκριμένο μέγεθος ταμπλώ το overhead παραλληλοποίησης δεν επηρεάζει σημαντικά το συνολικό χρόνο εκτέλεσης του αλγορίθμου. Επομένως, εκμεταλλευόμαστε πλήρως την παραλληλοποίηση. Παράλληλα, ως προς το Διάγραμμα 1.4, το speedup είναι ιδανικό (καθώς είναι γραμμικό). Παρατηρούμε ότι, όταν αυξάνεται ο αριθμός των threads, προκύπτει μεγαλύτερο speedup.

Αυτό συμβαίνει, επειδή το μέγεθος ταμπλώ είναι αρκετά μεγάλο ώστε να επιτρέπει τη βέλτιστη κατανομή των σελίδων στη cache, διασφαλίζοντας ότι η επεξεργασία από τους πυρήνες εκμεταλλεύεται αποδοτικά τη διαθέσιμη μνήμη. Συγχρόνως, το μέγεθος του ταμπλώ είναι τέτοιο που το κόστος παραλληλοποίησης (συγχρονισμός, επικοινωνία) είναι αμελητέο σε σχέση με το όφελος της παραλληλοποίησης.

- Διάγραμμα χρόνου εκτέλεσης προγράμματος συναρτήσει των πυρήνων (1.5) και διάγραμμα speedup του προγράμματος συναρτήσει των νημάτων (1.6) για ταμπλώ μεγέθους 4096×4096



Διάγραμμα 1.5



Διάγραμμα 1.6

Για ταμπλώ μεγέθους 4096×4096 , στο Διάγραμμα 1.5, παρατηρούμε ότι με την προσθήκη περισσότερων threads, μειώνεται ο χρόνος εκτέλεσης του προγράμματος. Ωστόσο, μόλις προσθέσουμε περισσότερα από 4 threads, ο χρόνος εκτέλεσης παραμένει σχεδόν σταθερός και το πρόγραμμά μας δεν κλιμακώνει περαιτέρω. Αυτό συμβαίνει, λόγω της αυξημένης συμφόρησης στο διάδρομο μνήμης, σύμφωνα με το νόμο του Amdahl. Τέλος, σε ότι αφορά το speedup (Διάγραμμα 1.6), για τα πρώτα 4 threads είναι ιδανικό (linear). Ωστόσο, όταν χρησιμοποιούνται περισσότερα από 4 threads παρατηρείται scalability break. Αυτό προκύπτει, επειδή τα δεδομένα δεν χωρούν πλέον σε ένα block μνήμης με αποτέλεσμα, ο συνολικός αριθμός των blocks να αυξάνεται λόγω του μεγέθους του ταμπλώ. Η αύξηση αυτή, δημιουργεί συμφόρηση στο διάδρομο μνήμης όπως αναφέραμε και παραπάνω.

Άσκηση 2

Παραλληλοποίηση και Βελτιστοποίηση Αλγορίθμων σε Αρχιτεκτονικές Κοινής Μνήμη

Άσκηση 2.1: Παραλληλοποίηση και Βελτιστοποίηση του Αλγορίθμου k-means

Στην άσκηση 2.1, αναπτύξαμε δύο παράλληλες εκδόσεις του αλγορίθμου K-means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με τη χρήση του OpenMP. Για τον σκοπό αυτό, δοκιμάσαμε τις εξής βελτιώσεις υλοποίησης παράλληλου κώδικα, την υλοποίηση με Shared Clusters και την υλοποίηση με Copied Clusters και Reduce.

Shared Clusters:

1. Παραλληλοποίηση της εκδοχής του κώδικα Shared Clusters στο σημείο που φαίνεται παρακάτω

Σε αυτό το ερώτημα, θέλουμε να παραλληλοποιήσουμε το σειριακό κώδικα που μας έχει δοθεί προσθέτοντας κατάλληλες εντολές συγχρονισμού, ώστε να γίνει ορθά η παράλληλη ενημέρωση των πινάκων newClusterSize και newClusters. Παρακάτω φαίνεται το τμήμα του κώδικα που τροποποιήσαμε:

```
#pragma omp parallel for default(shared) private(index, i, j)

for (i=0; i<numObjs; i++) {
    // find the array index of nearest cluster center
    // ... index is private
    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
     */

    // has to be atomic because two or more threads could assign the index variable with the same cluster ...
    #pragma omp atomic
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        /*
         * TODO: protect update on shared "newClusters" array
         */
        // also needs to be operated in an atomic manner
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
}
```

Πιο συγκεκριμένα, με το directive `#pragma omp parallel for` ορίζουμε ότι το for loop `for (i=0; i<numObjs; i++)` θα εκτελεστεί παράλληλα από διαφορετικά νήματα. Στην ίδια γραμμή ορίζουμε και ποιες μεταβλητές θα είναι shared μεταξύ των νημάτων και ποιες μεταβλητές θα είναι private για κάθε νήμα. Η μεταβλητή `i` είναι εξ' ορισμού private για κάθε νήμα, ως ο δείκτης του παραλληλοποιούμενου for loop. Private θα είναι, επίσης, και η μεταβλητή `index`, διότι στη μεταβλητή αυτή κάθε νήμα γράφει προσωρινά το id του cluster που προέκυψε από τη συνάρτηση `find_nearest_cluster()`. Η μεταβλητή `delta` είναι shared μεταξύ των νημάτων, αλλά είναι πιθανό περισσότερα του ενός νήματος να προσπαθήσουν να την ενημερώσουν ταυτόχρονα. Γι' αυτό το λόγο, προστατεύουμε την πράξη `delta += 1` με τη χρήση atomic operation. Έπειτα, ο πίνακας `membership` πρέπει να είναι shared, για να μπορούν όλα τα νήματα να τον ενημερώνουν και επειδή θέλουμε οι ενημερώσεις αυτές να διατηρηθούν και μετά το πέρας εκτέλεσης του παράλληλου τμήματος κώδικα. Επίσης, σημειώνουμε ότι αφού

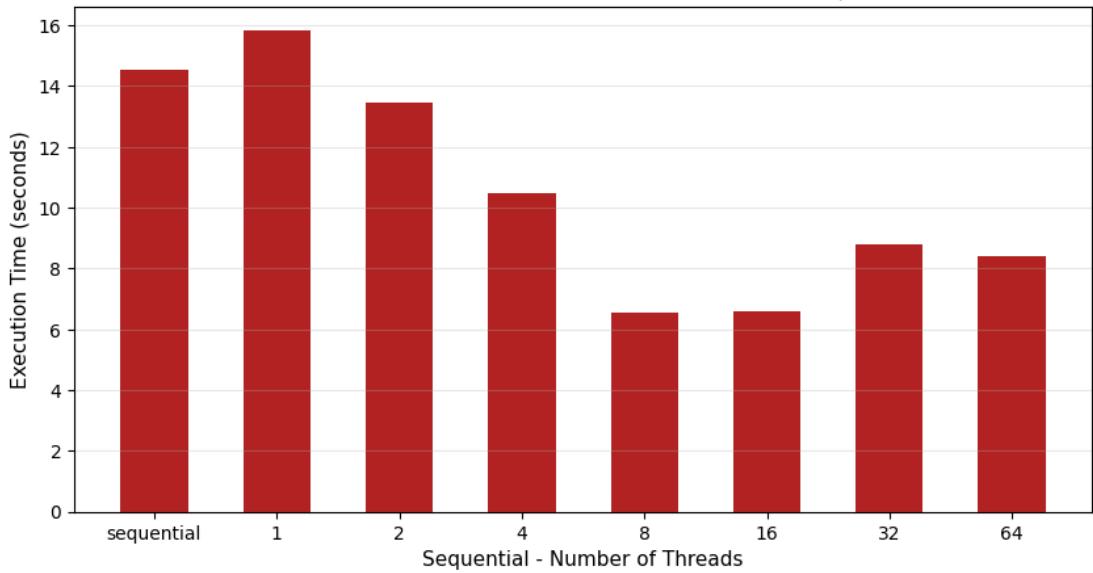
κάθε iteration του for loop εκτελείται μονάχα μία φορά και από ένα μόνο νήμα, τότε η μεταβλητή i δε θα λάβει ποτέ την ίδια τιμή για δύο ή περισσότερα νήματα ταυτόχρονα και, άρα, δε θα προκύψουν race conditions κατά την εγγραφή στη θέση μνήμης $\text{membership}[i]$. Η μεταβλητή j πρέπει να είναι private για κάθε νήμα, ώστε κάθε νήμα να μπορεί να την αυξάνει απομονωμένα από τα υπόλοιπα νήματα και να εκτελεί ορθά το for loop $\text{for } (j=0; j < \text{numCoords}; j++)$. Τέλος, οι πίνακες newClusterSize και newClusters είναι shared και, γι' αυτό το λόγο, προς αποφυγή race conditions κατά την ενημέρωσή των, πρέπει να τις προστατεύσουμε με atomic operations.

Εκτελούμε τον κώδικα για το configuration $\{\text{Size}, \text{Coords}, \text{Clusters}, \text{Loops}\} = \{256, 16, 32, 10\}$, για $\text{threads} = \{1, 2, 4, 8, 16, 32, 64\}$ στο μηχάνημα sandman. Προκύπτουν τα παρακάτω διαγράμματα:

- Barplot διάγραμμα χρόνου εκτέλεσης προγράμματος Naive K-means (2.1.1.α) και αντίστοιχο διάγραμμα speedup (2.1.1.β)

Naive K-means (Shared Clusters)

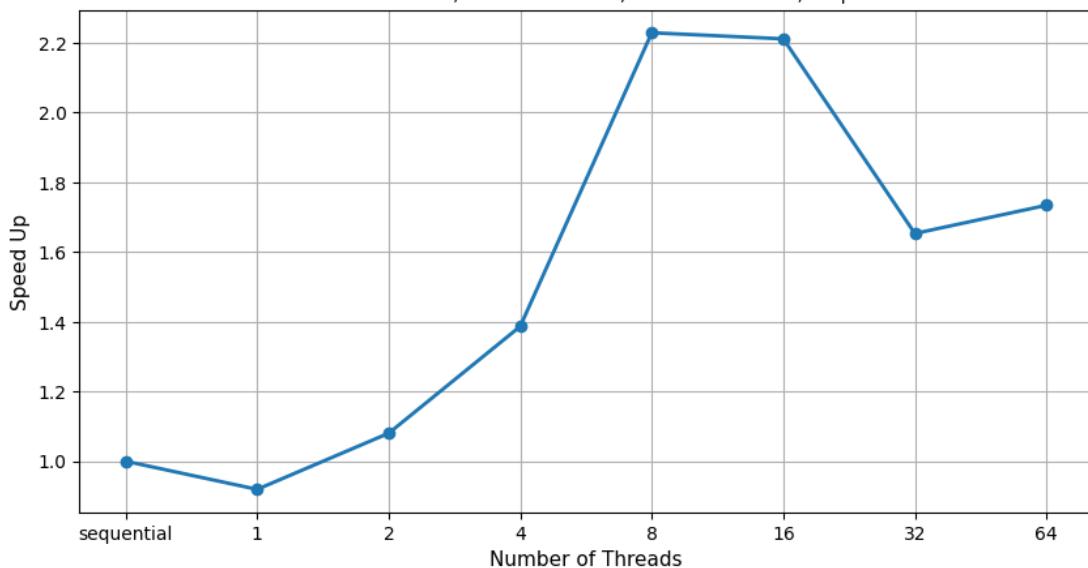
ClusterSize = 256MB, NumCoords = 16, NumClusters = 32, Loops = 10



Shared Clusters: Διάγραμμα 2.1.1.α

Naive K-means (Shared Clusters)

ClusterSize = 256MB, NumCoords = 16, NumClusters = 32, Loops = 10



Shared Clusters: Διάγραμμα 2.1.1.β

Από τα διαγράμματα παρατηρούμε ότι:

Στο **Διάγραμμα εκτέλεσης χρόνου (2.1.1.α)**, ο χρόνος εκτέλεσης μειώνεται καθώς αυξάνεται ο αριθμός των νημάτων, μέχρι και τα 8 νήματα. Από τα 16 νήματα και πάνω, ο χρόνος εκτέλεσης αρχίζει να αυξάνεται, κάτι που δείχνει ότι ο αλγόριθμος δεν επιταχύνεται περαιτέρω με περισσότερα νήματα.

Σε ότι αφορά το **Διάγραμμα 2.1.1.β**, καταλήγουμε σε παρόμοια συμπεράσματα. Το speedup αυξάνεται μέχρι και τα 8 νήματα, δείχνοντας ότι η πολυνηματική επεξεργασία είναι αποδοτική μέχρι αυτό το σημείο. Από τα 16 νήματα και μετά, η επιτάχυνση παραμένει σχεδόν σταθερή ή μειώνεται ελαφρώς.

Επομένως, όταν συγκρίνουμε τις χρονικές επιδόσεις του παράλληλου και του σειριακού προγράμματος δεν παρατηρούμε σημαντική βελτίωση στο παράλληλο πρόγραμμα. Αυτό συμβαίνει, διότι, για πολλά νήματα, το κόστος χρήσης atomic operations για το συγχρονισμό γίνεται σημαντικότερο από τα οφέλη που προσφέρει η παραλληλοποίηση. Οι κρίσιμες περιοχές που έχουν προστατευθεί από atomic operations είναι τρεις. Σε κάθε iteration του for loop, ένα νήμα επισκέπτεται πάντα τις δύο κρίσιμες περιοχές που αφορούν την ενημέρωση των newClusterSize και newClusters. Συνεπώς, στα σημεία αυτά, θα συγκεντρώνονται συχνά πολλά νήματα, τα οποία θα περιμένουν το ένα το άλλο για να εισέλθουν στην κρίσιμη περιοχή. Έτσι, δημιουργούνται σημεία συμφόρησης και προκαλούνται καθυστερήσεις.

Ένας ακόμη λόγος που δεν παρατηρούμε σημαντική βελτίωση στην επίδοση μπορεί να οφείλεται στις μετακινήσεις των νημάτων μεταξύ διαφορετικών πυρήνων. Όταν τα νήματα μετακινούνται σε άλλους πυρήνες μετά από ένα context switch, συχνά χρειάζεται να επαναφορτώσουν δεδομένα από τη μνήμη, γεγονός που αυξάνει τις προσπελάσεις στη μνήμη και επιβραδύνει την εκτέλεση του προγράμματος.

Συμπερασματικά, η αύξηση του πλήθους των νημάτων βελτιώνει την επίδοση του προγράμματος, συντηρητικά και μέχρι ένα σημείο, ύστερα από το οποίο τα κέρδη της παραλληλοποίησης αρχίζουν να μειώνονται.

Δοκιμάσαμε, επιπλέον, να παραλληλοποιήσουμε και το for loop που φαίνεται παρακάτω και υπολογίζει τους μέσους όρους και τα νέα κέντρα των clusters:

```
// 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];
        }
    }
}
```

Ωστόσο, δεν παρατηρήσαμε σημαντική διαφορά στο χρόνο εκτέλεσης του παράλληλου προγράμματός μας. Ισως αυτό να οφείλεται στο γεγονός ότι το for loop αυτό έχει χαμηλότερο operational intensity από το προηγούμενο και εκτελείται και για λιγότερα iterations. Άρα, η σειριακή εκτέλεση αυτού του for loop δεν είναι τόσο δαπανηρή, για να αξίζει να το παραλληλοποιήσουμε.

2. Χρήση της μεταβλητής περιβάλλοντος GOMP_CPU_AFFINITY

Μέσω της μεταβλητής περιβάλλοντος GOMP_CPU_AFFINITY, ο προγραμματιστής μπορεί να προσδέσει ένα συγκεκριμένο νήμα σε ένα συγκεκριμένο πυρήνα καθ' όλη τη διάρκεια εκτέλεσης του προγράμματος (thread binding). Η μεταβλητή αυτή χρησιμοποιείται συχνά σε εφαρμογές OpenMP. Θα την χρησιμοποιήσουμε και εμείς σε αυτό το ερώτημα και θα εκτελέσουμε ξανά τον αλγόριθμο που αναπτύξαμε στο προηγούμενο ερώτημα,ώστε να καταγράψουμε τις πιθανές διαφορές που θα παρατηρήσουμε.

Η εκτέλεση του προγράμματος θα γίνει στο μηχάνημα sandman. Οπότε, για να μπορέσουμε να προσδέσουμε τα νήματα στους κατάλληλους πυρήνες, πρέπει, πρώτα, να γνωρίζουμε πώς είναι οργανωμένοι οι πυρήνες στο μηχάνημα. Γι' αυτό το σκοπό, γράφουμε ένα script και εκτελούμε την εντολή lscpu στο sandman. Παρακάτω φαίνονται κάποια από τα αποτελέσματα που λαμβάνουμε:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	64
On-line CPU(s) list:	0-63
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	4
NUMA node(s):	4
Vendor ID:	GenuineIntel
CPU family:	6

NUMA node0 CPU(s):	0-7,32-39
NUMA node1 CPU(s):	8-15,40-47
NUMA node2 CPU(s):	16-23,48-55
NUMA node3 CPU(s):	24-31,56-63

Βλέπουμε ότι το sandman αποτελείται από 4 NUMA nodes. Κάθε ένα NUMA node διαθέτει 8 πυρήνες και κάθε πυρήνας μπορεί να φιλοξενήσει μέχρι και 2 νήματα. Οπότε, συνολικά, το sandman μπορεί να φιλοξενήσει μέχρι και 64 νήματα. Παρατηρούμε και τα ids που έχουν οι επεξεργαστές που διαθέτει το sandman. Λαμβάνοντας όλα αυτά υπόψιν, γράφουμε το παρακάτω command, ώστε να προσδέσουμε κάθε νήμα σε έναν συγκεκριμένο επεξεργαστή:

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

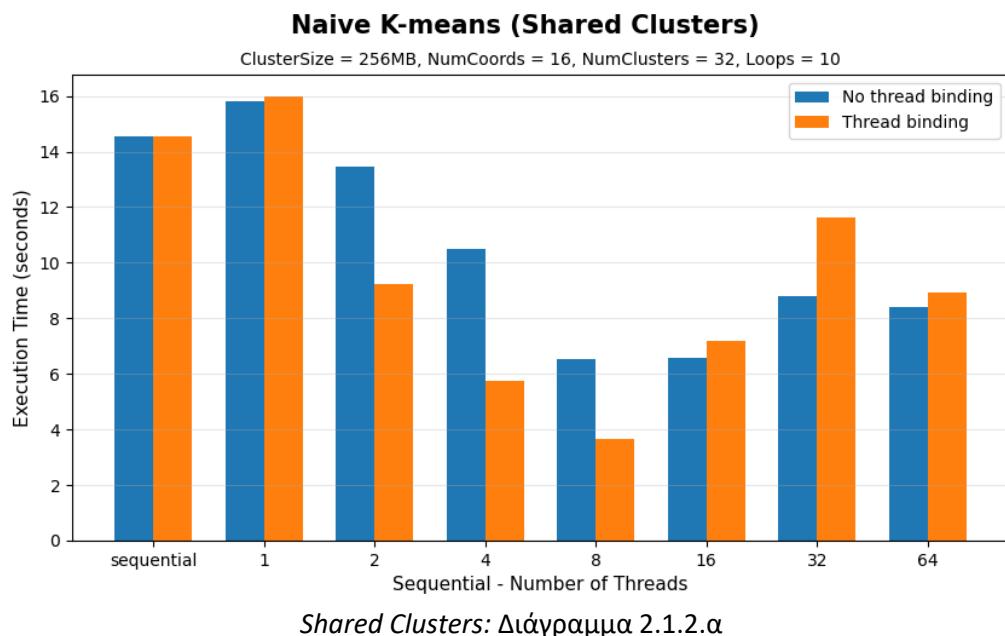
Η μεταβλητή \$nthrds φέρει το πλήθος των νημάτων που χρησιμοποιούνται. Και, έτσι, για παράδειγμα, αν εκτελέσουμε το πρόγραμμά μας για 4 νήματα, τότε:

- \$nthrds = 4
- GOMP_CPU_AFFINITY="0-3"

Επομένως, το πρώτο νήμα θα προσδεθεί στον επεξεργαστή με id = 0, το δεύτερο νήμα θα προσδεθεί στον επεξεργαστή με id = 1 κ.ο.κ.

Εκτελούμε το πρόγραμμά μας, πάλι το configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10} για threads = {1, 2, 4, 8, 16, 32, 64}. Παρακάτω φαίνονται τα διαγράμματα που προκύπτουν:

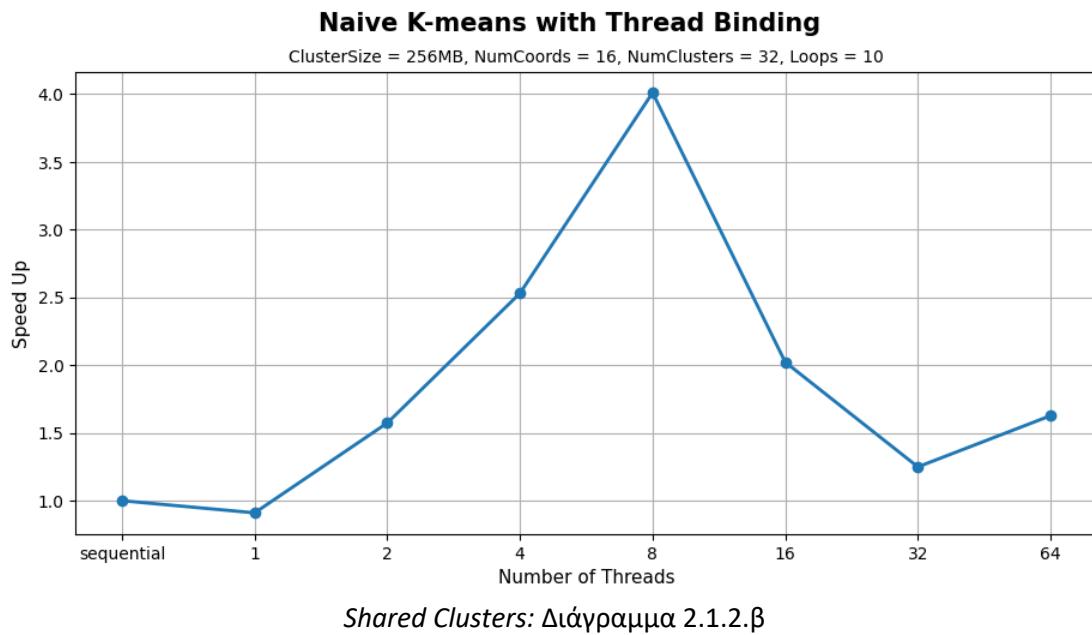
- **Barplot διάγραμμα χρόνου εκτέλεσης.** Παρουσιάζει τη σύγκριση του χρόνου εκτέλεσης του αλγορίθμου K-means με και χωρίς thread binding, για διάφορους αριθμούς νημάτων (2.1.2.α) και αντίστοιχο **διάγραμμα speedup** (2.1.2.β)



Στο **Διάγραμμα εκτέλεσης χρόνου (2.1.2.α)**, σε ότι αφορά τις εκτελέσεις με thread binding (πορτοκαλί ράβδοι), παρατηρούμε ότι ο χρόνος εκτέλεσης είναι χαμηλότερος σε σύγκριση με την εκδοχή χωρίς binding μέχρι και τη χρήση 8 νημάτων, όπου και παρατηρείται ο βέλτιστος χρόνος. Αυτό συμβαίνει επειδή, μέχρι τα 8 νήματα, τα νήματα εκτελούνται στο ίδιο NUMA node και επομένως, η πρόσβαση στη μνήμη είναι ταχύτερη λόγω καλύτερης τοπικότητας των δεδομένων.

Όταν ο αριθμός των νημάτων ξεπεράσει τα 8, τα επιπλέον νήματα πρέπει να εκτελεστούν σε διαφορετικά NUMA nodes. Αυτό συμβαίνει, διότι, όπως είδαμε και προηγουμένως με την εντολή Iscpu, το NUMA node0 διαθέτει τους επεξεργαστές με ids 0 – 7 και 32 – 39, ενώ οι επεξεργαστές 8 – 15 βρίσκονται στο NUMA node1. Συνεπώς, αφού τα επόμενα νήματα θα βρίσκονται σε διαφορετικά NUMA nodes, θα αυξάνεται η καθυστέρηση πρόσβασης στη μνήμη, μειώνοντας τα οφέλη από την αύξηση των νημάτων.

Επομένως, βλέπουμε ότι, πράγματι, η μετακίνηση των νημάτων μεταξύ των πυρήνων λόγω του context switching και του I/O επιβράδυνε το πρόγραμμά μας στο προηγούμενο ερώτημα. Τώρα που προσδέσαμε κάθε νήμα σε ένα συγκεκριμένο πυρήνα, διατηρούμε το cache locality και μειώνουμε τις προσβάσεις στη μνήμη, επειδή τα δεδομένα που χρησιμοποιεί το νήμα παραμένουν τοπικά στην cache του συγκεκριμένου πυρήνα, αποφεύγοντας την απώλεια δεδομένων από την cache που θα συνέβαινε αν το νήμα μεταφερόταν σε άλλον πυρήνα. Γι' αυτό το λόγο, το πρόγραμμά μας παρουσιάζει καλύτερη επίδοση, όταν χρησιμοποιούμε thread binding.



Το **Διάγραμμα Speedup (2.1.2.β)** δείχνει ότι η χρήση του **thread binding** βελτιώνει την απόδοση του αλγορίθμου K-means για μικρό έως μέτριο αριθμό νημάτων. Η μέγιστη επιτάχυνση παρατηρείται στα 8 νήματα, όπου ο παραλληλισμός αξιοποιείται πλήρως. Ωστόσο, καθώς αυξάνεται ο αριθμός των νημάτων πέρα από αυτό το σημείο, το speedup μειώνεται λόγω του αυξημένου overhead συγχρονισμού και της κατακερματισμένης εργασίας ανά νήμα. Συνολικά, το thread binding προσφέρει οφέλη στην επίδοση, αλλά αυτά περιορίζονται όταν χρησιμοποιείται πολύ μεγάλος αριθμός νημάτων.

Copied Clusters and Reduction:

1. Παραλληλοποιήστε τον αλγορίθμου k-means με τη χρήση copied clusters και reduction, αγνοώντας τα hints για false sharing.

Σε αυτό το ερώτημα, θα χρησιμοποιήσουμε τοπικούς (copied) πίνακες για κάθε νήμα, ώστε να μην απαιτείται συγχρονισμός κατά την εγγραφή – ενημέρωση. Οι τοπικοί πίνακες που θα

χρησιμοποιήσουμε θα ονομάζονται local_newClusterSize και local_newClusters. Έπειτα, τα αποτελέσματα αυτών των τοπικών πινάκων θα συνδυάζονται (reduction) στους πίνακες newClusterSize και newClusters αντιστοίχως.

Παρακάτω φαίνονται τα νέα τμήματα κώδικα που προσθέτουμε:

```

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

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

int thread_id;

#pragma omp parallel for reduction(+:delta) default(shared) private(i,index,j,thread_id)
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; // due to reductive parallel for it will be manipulated as an "atomic" variable ...

    // 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
     */
    thread_id = omp_get_thread_num();
    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
 */

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

```

Όπως και στο προηγούμενο ερώτημα, το for loop `for (i=0; i<numObjs; i++)` θα εκτελεστεί παράλληλα από διαφορετικά νήματα. Για τους ίδιους λόγους που αναλύσαμε και στο προηγούμενο ερώτημα οι μεταβλητές i, index και j πρέπει να είναι private για κάθε νήμα. Σε αυτό το ερώτημα, όμως, ορίζουμε και μία νέα μεταβλητή, το `thread_id`, στην οποία κάθε νήμα θα αποθηκεύει το id του μέσω της εντολής `thread_id = omp_get_thread_num();`. Για την αποφυγή race conditions κατά την εγγραφή αυτή, ορίζουμε τη μεταβλητή `thread_id` ως private για κάθε νήμα. Η μεταβλητή `delta` ορίζεται μέσα στο reduction list και, άρα, κάθε νήμα θα διαθέτει ένα αντίγραφο αυτής, ώστε να μπορεί να την αυξάνει κατά 1, απομονωμένα από τα υπόλοιπα νήματα. Στο τέλος του παραλληλοποιούμενου for loop, θα

συγκεντρώνονται όλα τα αντίγραφα της μεταβλητής `delta` και θα αθροίζονται, ώστε να προκύπτει η τελική τιμή της `delta`. Όλες οι υπόλοιπες μεταβλητές είναι `shared`, ακόμη και οι τοπικοί πίνακες `local_newClusterSize` και `local_newClusters`. Κατά την εγγραφή, όμως, στους τοπικούς πίνακες δεν υπάρχει κίνδυνος για race conditions. Οι τοπικοί πίνακες διαθέτουν πλήθος γραμμών ίσο με το πλήθος των νημάτων. Οπότε, κάθε νήμα διαθέτει τη δική του γραμμή στους τοπικούς πίνακες και γράφει μονάχα σε αυτήν. Η γραμμή που διαθέτει το νήμα έχει δείκτη ίσο με το `thread_id`, το οποίο είναι και μοναδικό για κάθε νήμα.

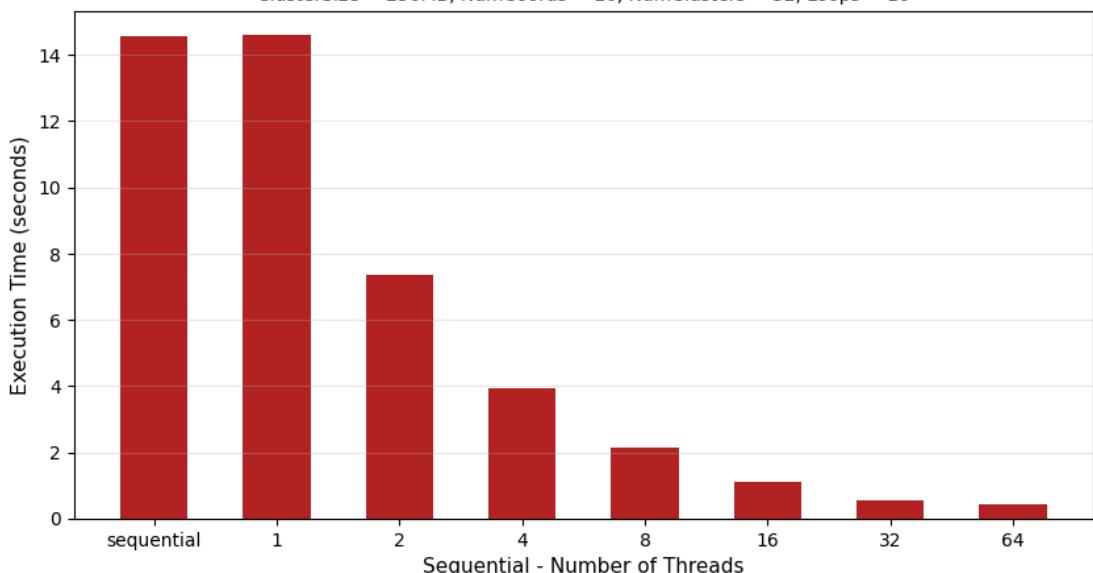
Έπειτα, το main thread συγκεντρώνει τα μερικά αποτελέσματα που έχουν αποθηκευτεί στους τοπικούς πίνακες `local_newClusterSize` και `local_newClusters` και τα αθροίζει κατάλληλα, ώστε να προκύψουν οι τελικές τιμές των `shared` πινάκων `newClusterSize` και `newClusters`. Και, με αυτόν τον τρόπο, πραγματοποιείται το ζητούμενο reduction.

Εκτελούμε το πρόγραμμά μας για το configuration `{Size, Coords, Clusters, Loops} = {256, 16, 32, 10}`, για `threads = {1, 2, 4, 8, 16, 32, 64}` στο μηχάνημα sandman. Σημειώνουμε ότι για την εκτέλεση του προγράμματος χρησιμοποιούμε την τεχνική `thread – binding` που είδαμε και στο προηγούμενο ερώτημα (`GOMP_CPU_AFFINITY`).

Παρακάτω φαίνονται το **διάγραμμα εκτέλεσης χρόνου (2.1.1.α)** και το **διάγραμμα επιτάχυνσης (speedup) (2.1.1.β)** που προκύπτουν:

K-means - Copied Clusters and Reduction

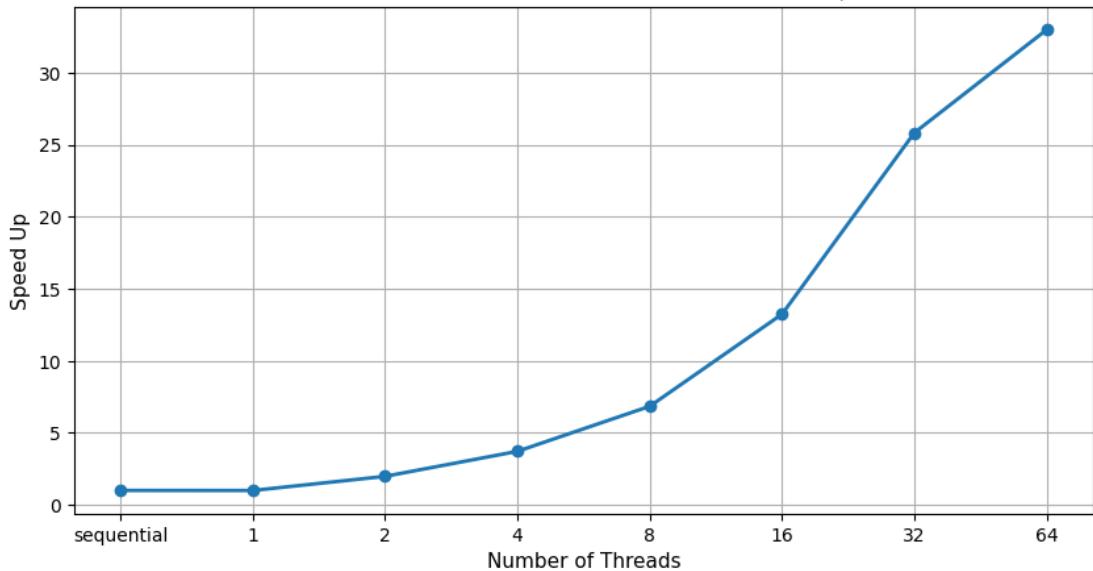
ClusterSize = 256MB, NumCoords = 16, NumClusters = 32, Loops = 10



Copied Clusters and Reduction: Διάγραμμα 2.1.1.α

K-means - Copied Clusters and Reduction

ClusterSize = 256MB, NumCoords = 16, NumClusters = 32, Loops = 10



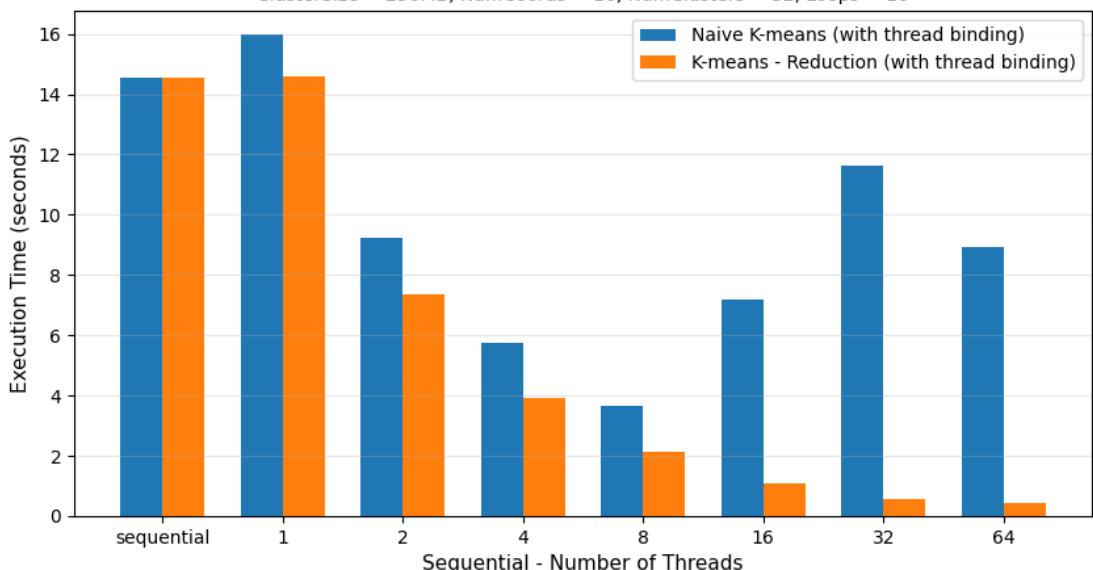
Copied Clusters and Reduction: Διάγραμμα 2.1.1.β

Μελετώντας το διάγραμμα 2.1.1α, παρατηρούμε ότι όταν αυξάνεται το πλήθος των νημάτων, μειώνεται αισθητά ο χρόνος εκτέλεσης του προγράμματός μας. Στο ίδιο συμπέρασμα καταλήγουμε, μελετώντας και το διάγραμμα επιτάχυνσης 2.1.1β, καθώς το speedup είναι σχεδόν γραμμικό μέχρι και τα 16 νήματα. Επομένως, η προσθήκη περισσότερων νημάτων βελτιώνει την επίδοση του προγράμματός μας και, άρα, το πρόγραμμά μας κλιμακώνει.

Αν συγκρίνουμε τα διαγράμματά μας με αυτά που λάβαμε στο ερώτημα του k – means που χρησιμοποιεί shared clusters, παρατηρούμε ότι η νέα έκδοση του αλγορίθμου παρουσιάζει σημαντική βελτίωση στην επίδοση. Χάριν ευκολίας, συλλέξαμε τις μετρήσεις των δύο ερωτημάτων σε ένα κοινό διάγραμμα, το οποίο φαίνεται παρακάτω και δείχνει ξεκάθαρα πόσο καλύτερη επίδοση εμφανίζει ο αλγόριθμος που χρησιμοποιεί copied clusters και reduction:

K-means

ClusterSize = 256MB, NumCoords = 16, NumClusters = 32, Loops = 10

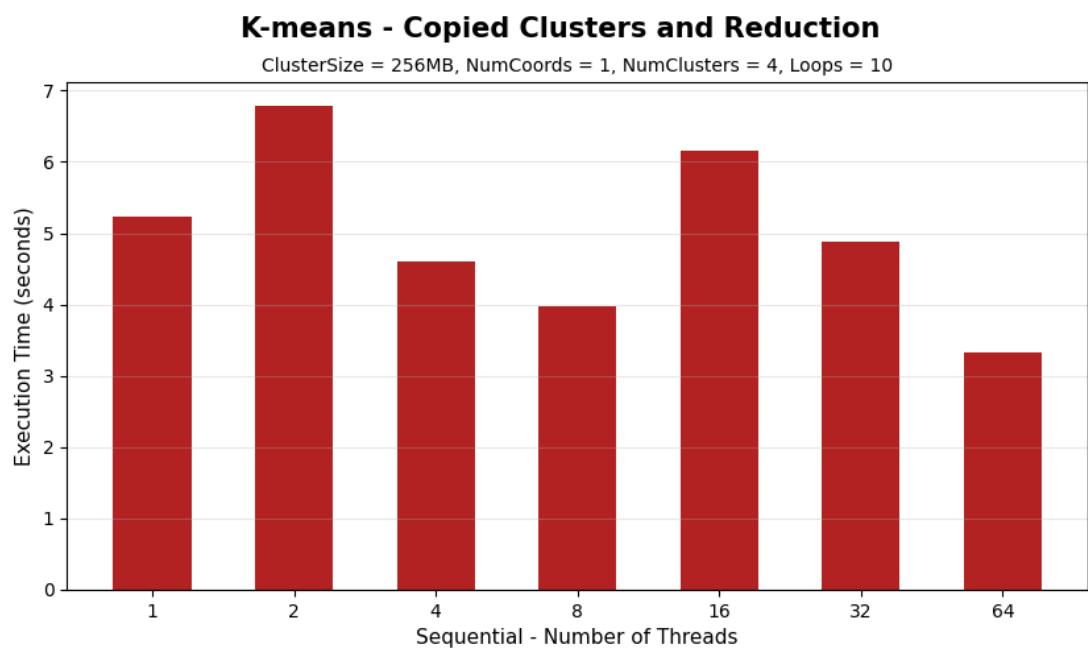


Σύγκριση χρόνων εκτέλεσης copied και shared clusters: Διάγραμμα 2.1.1.γ

Στο προηγούμενο ερώτημα χρησιμοποιούσαμε shared clusters, με αποτέλεσμα να υπάρχει η ανάγκη συγχρονισμού προς αποφυγή race conditions κατά την εγγραφή στα shared clusters από τα διάφορα νήματα. Ο συγχρονισμός αυτός πραγματοποιούταν με τη χρήση atomic operations και απ' ότι φαίνεται το κόστος του συγχρονισμού ήταν σημαντικότερο από τα οφέλη που προσφέρει η παραλληλοποίηση. Αντιθέτως, στο παρόν ερώτημα, χρησιμοποιούμε copied clusters για κάθε νήμα και απαλλασσόμαστε από τον κίνδυνο των race conditions και την ανάγκη συχρονισμού. Κάποια από τα μειονεκτήματα της νέας έκδοσης, όμως, είναι το ότι απαιτεί περισσότερη μνήμη για τη δημιουργία των copied clusters για κάθε νήμα και το ότι, στο τέλος του παράλληλου κώδικα, πρέπει να συνδυαστούν τα μερικά αποτελέσματα (reduction) από ένα νήμα, ώστε να προκύψουν τα τελικά μας αποτελέσματα. Ωστόσο, οι περιορισμοί αυτοί δε φαίνεται να επηρεάζουν αρνητικά την επίδοση του προγράμματος και, συνεπώς, η υλοποίηση με copied clusters είναι προτιμότερη για τη συγκεκριμένη εφαρμογή που εξετάζουμε.

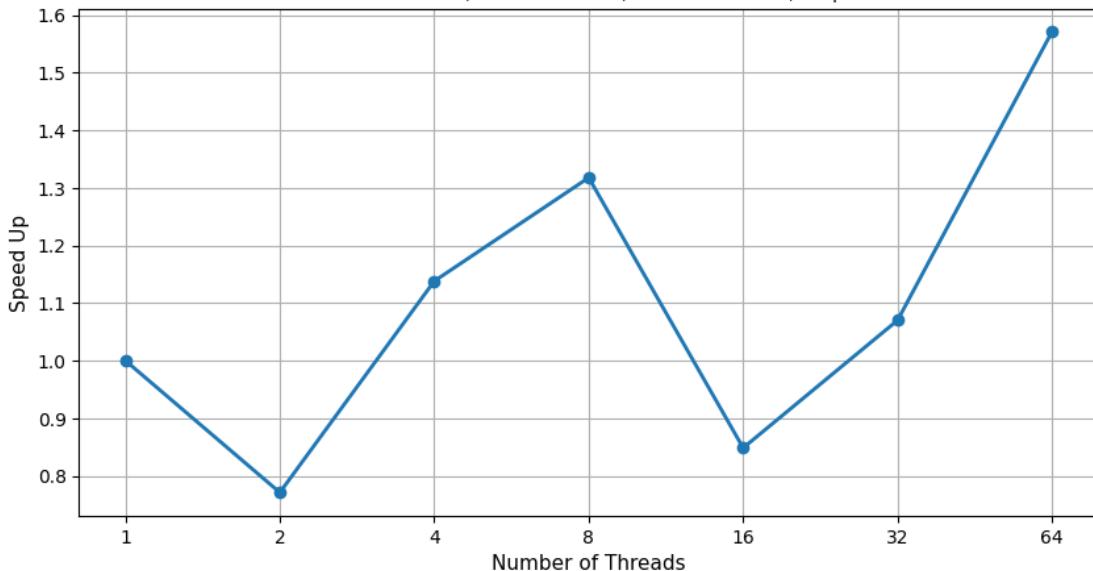
2. Εκτέλεση κώδικα με νέο configuration/ First – Touch/ False – Sharing

Σε αυτό το ερώτημα, θα εκτελέσουμε πάλι το k – means που χρησιμοποιεί copied clusters και reduction. Όμως, αυτή τη φορά θα χρησιμοποιήσουμε ένα νέο configuration: {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}. Συλλέξαμε τις μετρήσεις μας και δημιουργήσαμε το Διάγραμμα εκτέλεσης χρόνου (2.1.2.α) και το Διάγραμμα επιτάχυνσης (speedup) (2.1.2β):



K-means - Copied Clusters and Reduction

ClusterSize = 256MB, NumCoords = 1, NumClusters = 4, Loops = 10



Copied Clusters and Reduction: Διάγραμμα 2.1.2.β

Μελετώντας το διάγραμμα 2.1.2α, παρατηρούμε ότι όταν αυξάνονται τα νήματα, τότε δε δε μειώνεται πάντα ο χρόνος εκτέλεσης και αν κάποιες φορές μειώνεται, τότε η μείωση είναι ελαφριά και σε καμία περίπτωση τόσο αισθητή όσο ήταν στο προηγούμενο ερώτημα. Ύστερα, εξετάζοντας το διάγραμμα 2.1.2β, φαίνεται και πιο ξεκάθαρα ότι η παρουσία πολλών νημάτων δεν επιταχύνει το πρόγραμμά μας. Η μέγιστη επιτάχυνση που παρατηρούμε είναι για τα 64 νήματα και είναι περίπου ίση με 1.6 sec/sec, ενώ με το προηγούμενο configuration η μέγιστη επιτάχυνση που σημειώσαμε ήταν πάλι για 64 νήματα και περίπου ίση με 32 sec/sec. Είναι φανερό ότι τώρα, σε αντίθεση με πριν, το πρόγραμμά μας δεν κλιμακώνει. Η συμπεριφορά αυτή οφείλεται στο φαινόμενο false sharing, το οποίο, με τη σειρά του, είναι αποτέλεσμα της πολιτικής first – touch του Linux.

Σύμφωνα με την πολιτική first – touch του Linux, μία σελίδα μνήμης εκχωρείται στη μνήμη του κόμβου NUMA που βρίσκεται πιο κοντά στο νήμα που θα προσπελάσει τη σελίδα αυτή για πρώτη φορά. Η πολιτική αυτή έχει πολλά πλεονεκτήματα, ιδιαίτερα για προγράμματα σειριακής εκτέλεσης, καθώς βελτιώνει την τοπικότητα και την απόδοση της μνήμης, μειώνοντας τις καθυστερήσεις πρόσβασης.

Ωστόσο, η πολιτική first – touch μπορεί να φανεί περιοριστική κατά την εκτέλεση προγραμμάτων παράλληλης επεξεργασίας. Πιο συγκεκριμένα, τι συμβαίνει όταν σε ένα παράλληλο πρόγραμμα η προσπέλαση και αρχικοποίηση των δεδομένων γίνεται κυρίως από ένα μόνο νήμα; Τότε, όλα τα δεδομένα του προγράμματος καταλήγουν στη μνήμη ενός μόνο κόμβου NUMA. Κατά συνέπεια, για πολλά νήματα οι χρόνοι πρόσβασης στη μνήμη αυξάνονται. Διαφορετικές - ανεξάρτητες μεταβλητές αποθηκεύονται στο ίδιο cache line. Πολλά νήματα προσπελαύνουν τις διαφορετικές αυτές μεταβλητές που βρίσκονται στο ίδιο cache line, με αποτέλεσμα να προκαλούνται περιττές ενημερώσεις cache και, τελικά, συμφόρηση στο δίαυλο της μνήμης. Το φαινόμενο αυτό ονομάζεται false sharing.

Μία συχνή λύση για το φαινόμενο του false sharing είναι η παραλληλοποίηση των αρχικοποιήσεων δεδομένων, έτσι ώστε κάθε νήμα να έχει ένα τμήμα των δεδομένων σε κοντινή του μνήμη και, έτσι, δε θα δημιουργείται συμφόρηση στο δίαυλο μνήμης και, άρα, το πρόγραμμα θα εκτελείται ταχύτερα.

Στο συγκεκριμένο πρόγραμμα που μελετάμε αποφασίσαμε να παραλληλοποίησουμε την αρχικοποίηση των πινάκων local_newClusterSize και local_newClusters, έτσι ώστε κάθε νήμα να διαθέτει σε κοντινή του μνήμη τη γραμμή των πινάκων που χρησιμοποιεί. Όπως εξηγήσαμε και παραπάνω, κάθε νήμα έχει τη δική του γραμμή στους τοπικούς πίνακες και γράφει μονάχα σε αυτήν. Άρα, αναμένουμε ότι οι χρόνοι πρόσβασης στους τοπικούς πίνακες θα μειωθούν για κάθε νήμα και, συνεπώς, θα μειωθεί και ο χρόνος εκτέλεσης του προγράμματος. Παραλληλοποιούμε, επίσης, και την αρχικοποίηση των shared πινάκων newClusterSize και newClusters, με αποτέλεσμα κάθε νήμα να αποθηκεύει κάποια από τα στοιχεία των πινάκων αυτών σε κοντινή του μνήμη. Κάθε νήμα διαβάζει δεδομένα από τους shared αυτούς πίνακες και, άρα, αναμένουμε ότι και αυτή η τροποποίηση του κώδικα θα αποδειχθεί οφέλιμη. Παρακάτω φαίνονται ακριβώς οι αλλαγές που πραγματοποιήσαμε:

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

// before each loop, set cluster data to 0
#pragma omp parallel for default(shared) private(i,j)
for (i=0; i<numClusters; i++) {
    for (j=0; j<numCoords; j++)
        newClusters[i*numCoords + j] = 0.0;
    newClusterSize[i] = 0;
}

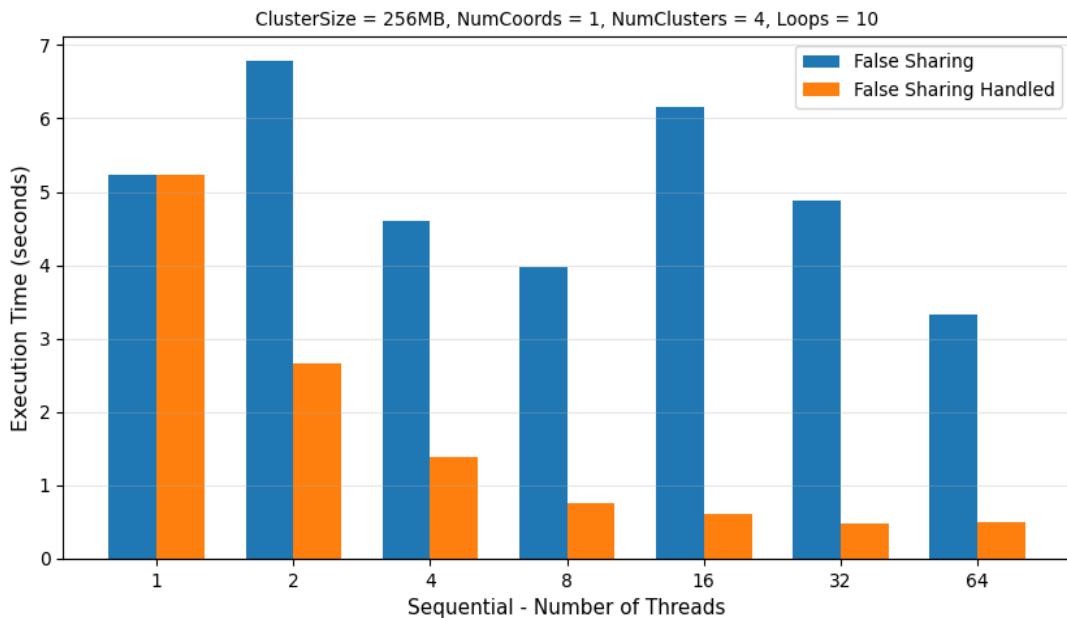
delta = 0.0;

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

#pragma omp parallel for default(shared) private(i,j,k)
for (i=0; i<nthreads; i++) {
    for (j=0; j<numClusters; j++){
        for (k=0;k<numCoords;k++)
            local_newClusters[i][j*numCoords + k] = 0.0;
        local_newClusterSize[i][j] = 0;
    }
}
```

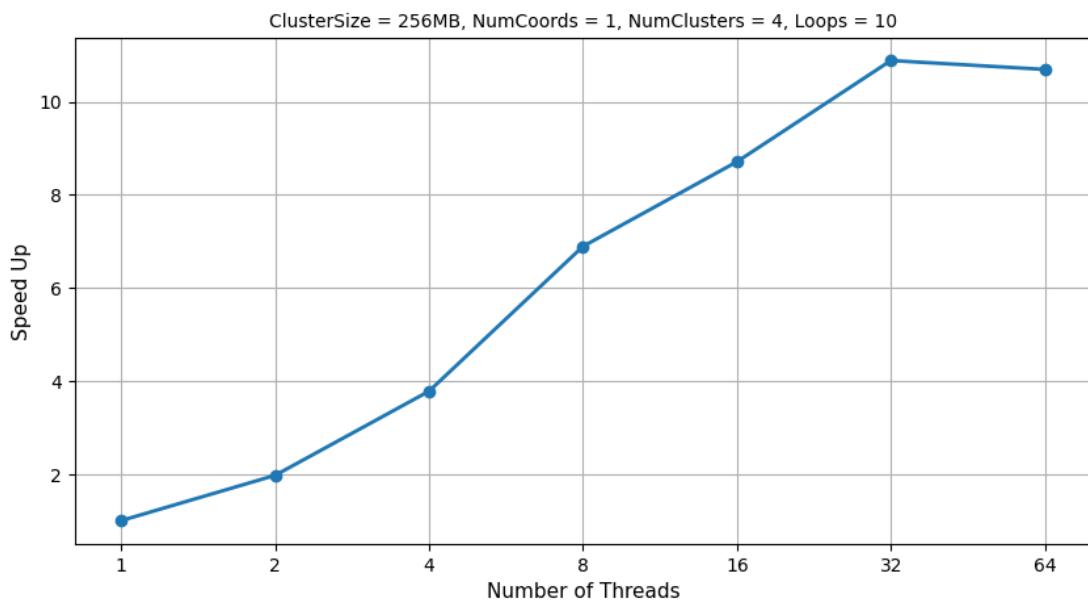
Εκτελούμε το νέο πρόγραμμα για το νέο configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}. Παρακάτω φαίνονται το νέο διάγραμμα χρόνου εκτέλεσης **2.1.2γ** που προκύπτει, καθώς και το νέο διάγραμμα επιτάχυνσης (**speedup**) **2.1.2δ**.

K-means



Σύγκριση χρόνων εκτέλεσης για copied clusters and reduction: Διάγραμμα 2.1.2.γ

K-means - Reduction - False Sharing Handled



No false sharing - copied clusters and reduction - Speedup: Διάγραμμα 2.1.2.δ

Στο διάγραμμα 2.1.2γ, παρατηρούμε ότι οι χρόνοι εκτέλεσης είναι αισθητά μικρότεροι από αυτούς που λάβαμε αρχικά για το νέο configuration. Συνεπώς, πράγματι, το φαινόμενο του false sharing παρεμπόδιζε την ταχεία εκτέλεση του προγράμματός μας. Μας απασχόλησε το φαινόμενο του false sharing πρώτη φορά τώρα και όχι στο προηγούμενο configuration, διότι τώρα το πλήθος των συντεταγμένων και των clusters ήταν μικρά (ίσο με 1 και 4 αντιστοίχως). Κατά συνέπεια, με την προηγούμενη υλοποίηση, οι τοπικοί πίνακες, που αρχικοποιούνταν από το main thread, βρίσκονταν σε πολύ κοντινές θέσεις μνήμης και τα διαφορετικά νήματα που τις προσπέλαυναν προκαλούσαν συμφόρηση στο δίσυλο της μνήμης. Όμως, με την αλλαγή που κάναμε τώρα, το πρόγραμμά έχει βελτιωθεί σημαντικά. Μελετώντας το διάγραμμα 2.1.2δ, παρατηρούμε ότι το πρόγραμμά μας κλιμακώνει σχεδόν γραμμικά μέχρι τα 32 νήματα.

Βέλτιστος χρόνος που πετύχαμε:

Είναι για 32 threads 0.4810s

```
dataset_size = 256.00 MB      numObjs = 33554432      numCoords = 1      numClusters = 4
Initial cluster centers:
clusters[0] =    0.00
clusters[1] =   2.22
clusters[2] =   4.43
clusters[3] =   6.66

OpenMP Kmeans - Reduction      (number of threads: 32)
          completed loop 10
          nloops = 10  (total =  0.4810s)  (per loop =  0.0481s)

Final cluster centers:
clusters[0] =   1.08
clusters[1] =   3.34
clusters[2] =   5.84
clusters[3] =   8.58
```

3. NUMA Aware Allocation

Έχοντας μελετήσει την πολιτική first - touch του Linux και το φαινόμενο false sharing από το προηγούμενο ερώτημα, θέλουμε να βελτιώσουμε περισσότερο την επίδοση του προγράμματός μας. Γι' αυτό το λόγο, αποφασίζουμε να παραλληλοποιήσουμε την αρχικοποίηση των objects, η οποία πραγματοποιείται στο αρχείο file_io.c και να καταστήσουμε το πρόγραμμά μας πιο NUMA aware. Με αυτόν τον τρόπο, το νήμα που θα αρχικοποιεί πρώτο κάποιο object θα αποθηκεύει το object σε κοντινό του κόμβο NUMA. Έτσι, όλα τα νήματα θα διαθέτουν σε κοντινή τους μνήμη κάποια από τα objects και όταν θα χρειαστεί, αργότερα, να τα προσπελάσουν κατά την εκτέλεση του αλγορίθμου k – means, οι χρόνοι πρόσβασης στη μνήμη θα είναι μικρότεροι. Επομένως, αναμένουμε ότι το πρόγραμμά μας θα εκτελείται ταχύτερα.

Κρίσιμα σημεία του κώδικα που αλλάξαμε, ώστε να έχουμε Numa aware allocation:

File_io.c:

```
#pragma omp parallel for private(j) schedule(static)
for (i = 0; i < numObjs; i++)
{
    unsigned int seed = i + omp_get_thread_num(); // Unique seed per thread
    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]);
    }
}
```

Ο κώδικας μετατράπηκε σε NUMA aware καθώς εκμεταλλεύεται την πολιτική first touch. Όπως περιγράψαμε νωρίτερα, παραλληλοποιώντας το for loop, κάθε νήμα αρχικοποιεί κάποια από τα objects και, συνεπώς, τα αποθηκεύει σε κοντινό κόμβο.

omp_reduction_NUMA_aware.c:

```
#pragma omp parallel private(i, j)
{
    int tid = omp_get_thread_num();
    local_newClusterSize[tid] = (int *) calloc(numClusters, sizeof(int));
    local_newClusters[tid] = (double *) calloc(numClusters * numCoords, sizeof(double));

    // Initialize each thread's local clusters with "first-touch"
    #pragma omp for schedule(static)
    for (i = 0; i < numClusters; i++) {
        for (j = 0; j < numCoords; j++)
            local_newClusters[tid][i * numCoords + j] = 0.0;
        local_newClusterSize[tid][i] = 0;
    }
}
```

A) αλλαγή στον κώδικα

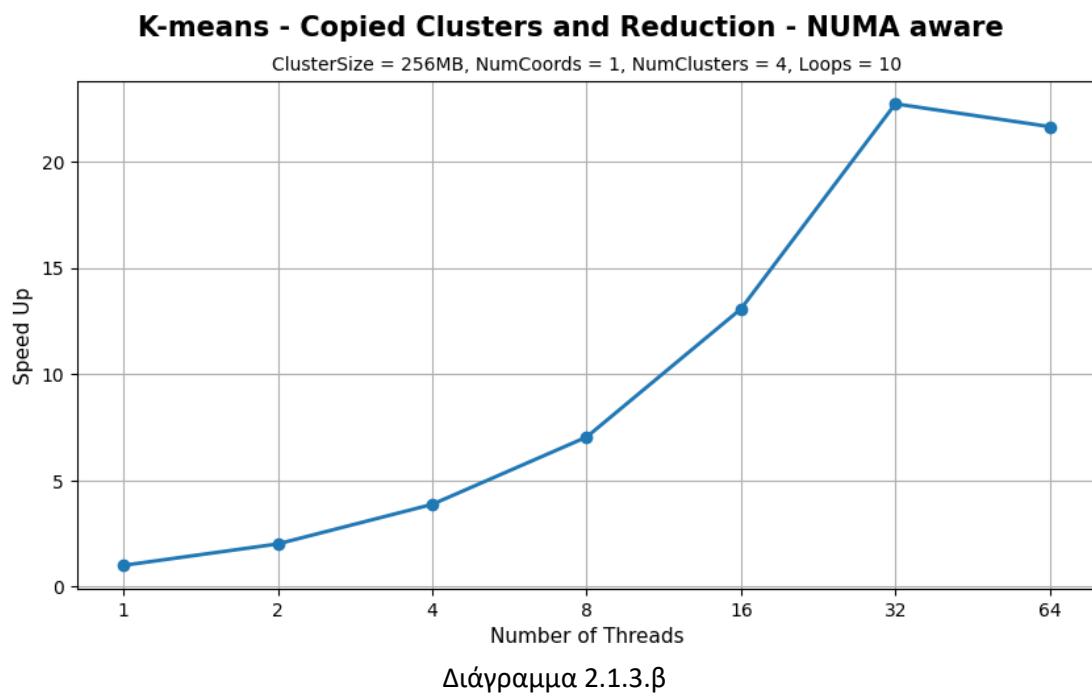
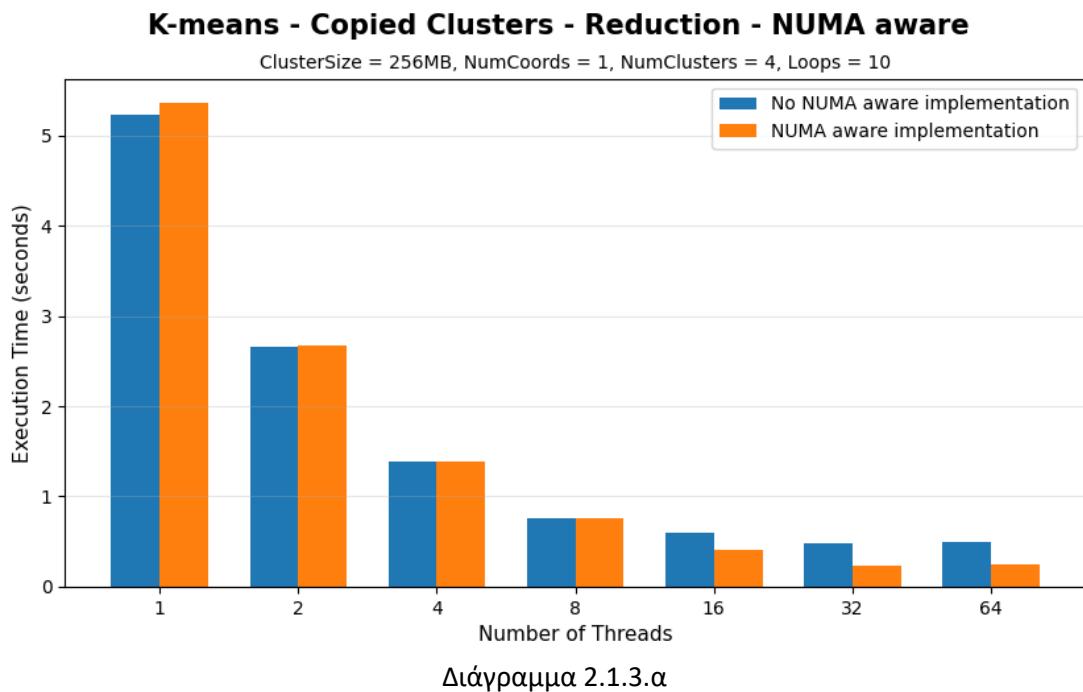
Αρχικοποίηση των τοπικών πινάκων local_newClusterSize και local_newClusters ανά νήμα: Κάθε νήμα εκτελεί την πρώτη προσπέλαση (first-touch) των δεδομένων local_newClusterSize και local_newClusters που του αντιστοιχούν. Αυτό αναγκάζει το λειτουργικό σύστημα να τοποθετήσει αυτά τα δεδομένα στον τοπικό NUMA κόμβο του νήματος, βελτιώνοντας την τοπικότητα της μνήμης.

```
// Reset local cluster data in a NUMA-aware way
#pragma omp parallel private(i, j, k)
{
    int tid = omp_get_thread_num();
    for (i = 0; i < numClusters; i++) {
        local_newClusterSize[tid][i] = 0;
        for (j = 0; j < numCoords; j++)
            local_newClusters[tid][i * numCoords + j] = 0.0;
    }
}
```

B) αλλαγή στον κώδικα

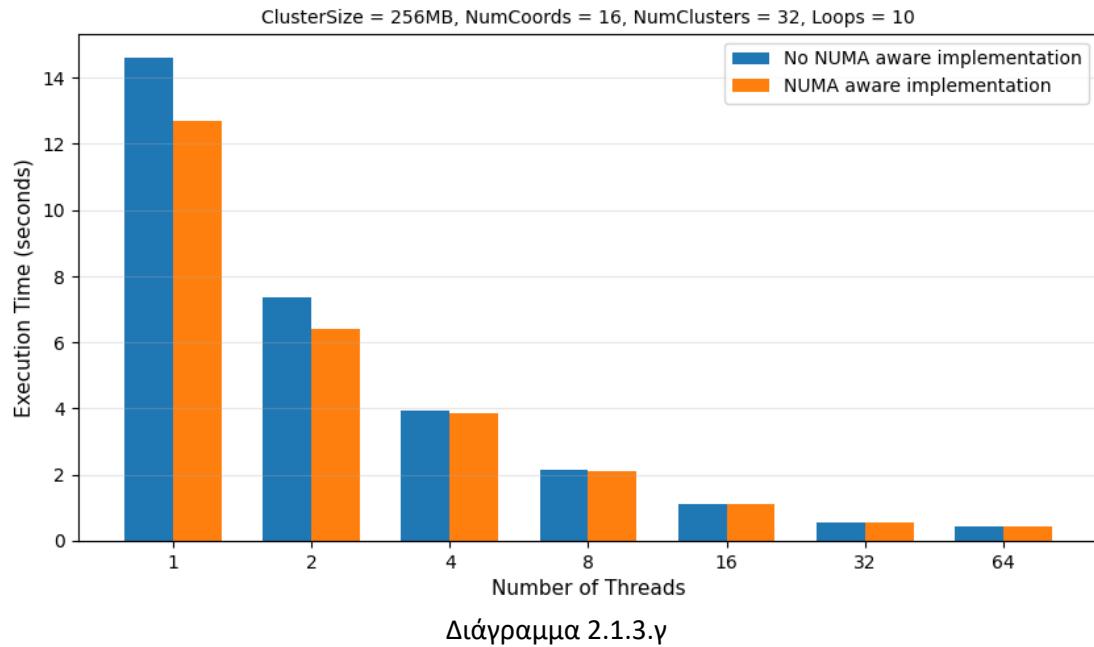
Επαναφορά των τοπικών πινάκων local_newClusterSize και local_newClusters πριν από κάθε loop. Εδώ, πριν από κάθε loop, τα δεδομένα επαναφέρονται από το κάθε νήμα, διατηρώντας την τοπική τους τοποθέτηση στον NUMA κόμβο. Κάθε νήμα χειρίζεται τα δεδομένα του, αποφεύγοντας προσπελάσεις σε απομακρυσμένους κόμβους.

Configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}:



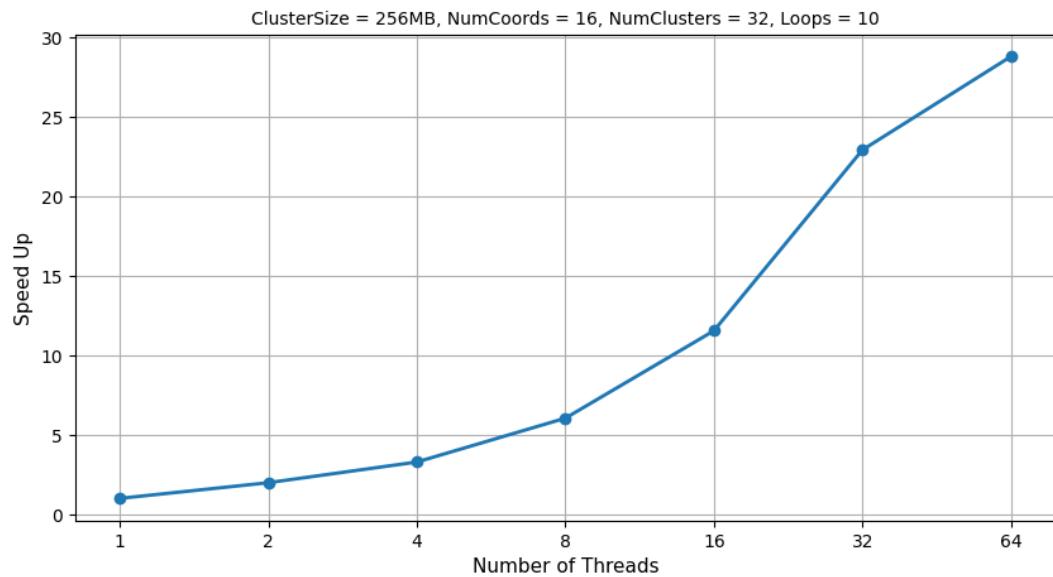
Configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}:

K-means - Copied Clusters - Reduction - NUMA aware



Διάγραμμα 2.1.3.γ

K-means - Copied Clusters and Reduction - NUMA aware



Διάγραμμα 2.1.3.δ

Σχολιασμός:

Η βελτίωση της απόδοσης του αλγορίθμου K-means με αξιοποίηση των χαρακτηριστικών NUMA του μηχανήματος και της πολιτικής first-touch στο Linux, επιτρέπει την κατανομή των δεδομένων objects στους κατάλληλους NUMA nodes, μειώνοντας τις καθυστερήσεις λόγω απομακρυσμένων προσβάσεων. Αυτό επιτυγχάνεται προσπελάζοντας τα δεδομένα αρχικά από τον πυρήνα όπου θα εκτελεστεί ο κύριος φόρτος εργασίας για κάθε νήμα.

Για το configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}, το κυρίαρχο πρόβλημα είναι το false sharing, το οποίο περιορίζει την απόδοση. Η NUMA-aware κατανομή βελτιώνει

την τοπικότητα της μνήμης, αλλά οι καθυστερήσεις από τις ενημερώσεις cache λόγω false sharing εξακολουθούν να επηρεάζουν την απόδοση.

Αντίθετα, στο configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, όπου υπάρχουν περισσότερες συντεταγμένες και clusters, το bottleneck σχετίζεται περισσότερο με τη διαχείριση των μεγάλων δεδομένων. Η NUMA-aware κατανομή παίζει καθοριστικό ρόλο εδώ, βελτιώνοντας αισθητά την απόδοση μέσω τοπικής πρόσβασης στη μνήμη και ελαχιστοποιώντας το overhead των απομακρυσμένων προσβάσεων. Η πολιτική first-touch βοηθά στην κλιμάκωση με πολλά νήματα, καθώς κάθε νήμα έχει γρήγορη πρόσβαση στα δεδομένα του στον τοπικό κόμβο.

Συνολικά, η NUMA-aware κατανομή είναι ιδιαίτερα αποδοτική, αλλά το κύριο bottleneck διαφέρει: στο πρώτο configuration είναι το false sharing, ενώ στο δεύτερο η διαχείριση τοπικότητας της NUMA μνήμης και η κατανομή μεγάλων δεδομένων.

Άσκηση 2.2: Εξοικείωση με τη χρήση των OpenMP tasks παράλληλοποιώντας τον αλγόριθμο Floyd-Warshall για αρχιτεκτονικές κοινής μνήμης.

1. Recursive FW

Σε αυτήν την άσκηση, υλοποιήσαμε μια παράλληλη έκδοση του recursive αλγορίθμου Floyd-Warshall χρησιμοποιώντας OpenMP tasks και πραγματοποιήσαμε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχάνημα sandman.

Παρακάτω φαίνονται τα τμήματα του κώδικα που τροποποιήσαμε:

```
// Starting the parallel region here to include all recursive calls within a single parallel region
#pragma omp parallel
{
    #pragma omp single
    {
        FW_SR(A, 0, 0, A, 0, 0, A, 0, 0, N, B);
    }
}
```

Αρχικά, στο main πρόγραμμα χρησιμοποιούμε την οδηγία `#pragma omp parallel` για να δηλώσουμε την έναρξη της παράλληλης περιοχής. Στην περιοχή αυτή, τα νήματα που δημιουργούνται εκτελούν παράλληλα τον κώδικα που ακολουθεί. Με την οδηγία `#pragma omp single` δηλώνουμε ότι το κομμάτι του κώδικα που ακολουθεί πρέπει να εκτελεστεί από ένα μόνο νήμα, ενώ τα υπόλοιπα νήματα είτε περιμένουν είτε εκτελούν άλλες εργασίες. Στην περίπτωσή μας, η οδηγία αυτή χρησιμοποιείται για να εξασφαλίσει ότι μόνο ένα νήμα θα ξεκινήσει την εκτέλεση της συνάρτησης FW_SR. Έπειτα, οι αναδρομικές κλήσεις μπορούν να δημιουργήσουν νέα tasks για να εκτελεστούν παράλληλα από τα υπόλοιπα νήματα. Με αυτόν τον τρόπο, αποφεύγεται η πολλαπλή εκτέλεση της αρχικής κλήσης και η εκτέλεση παραμένει συντονισμένη.

```

} else {
    // Recursive tasks for matrix computation with task dependency management
    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
    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);

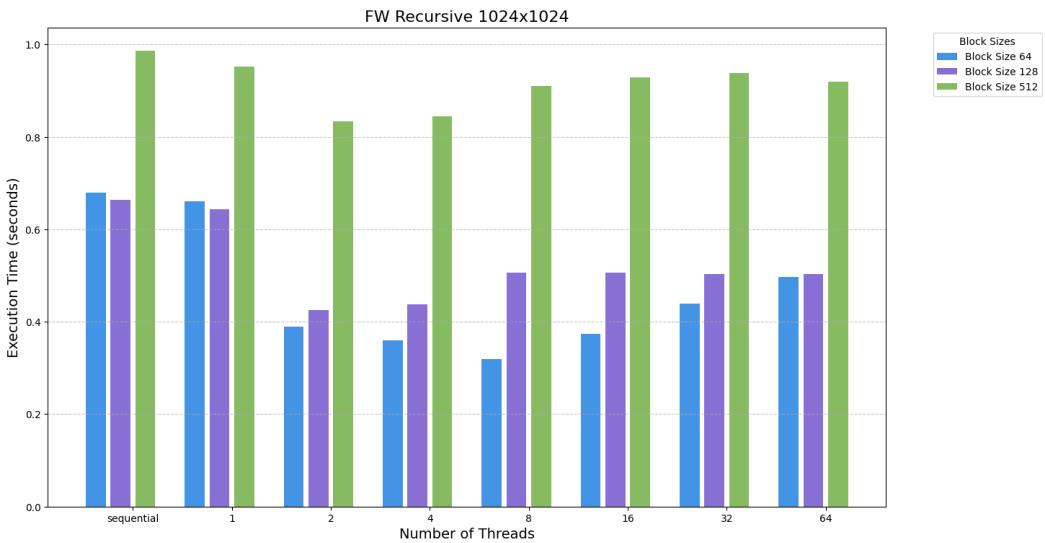
    #pragma omp task
    FW_SR(A, arow, acol + myN / 2, B, brow, bcol + myN / 2, C, crow + myN / 2, ccol + myN / 2, myN / 2, bsize);

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

```

Ακολούθως, τροποποιήσαμε το μέρος του κώδικα που ορίζεται η συνάρτηση FW_SR για την περίπτωση της αναδρομής. Σε αυτήν την περίπτωση, το πρόβλημα διασπάται αναδρομικά σε μικρότερα υποπροβλήματα όταν το μέγεθος του υποπίνακα myN είναι μεγαλύτερο από το bsize. Συγκεκριμένα, για κάθε υποπρόβλημα τα **tasks** δημιουργούνται με τη χρήση της οδηγίας `#pragma omp task` και εκτελούνται ταυτόχρονα από τα διαθέσιμα threads. Με την οδηγία `#pragma omp taskwait`, εξασφαλίζεται η σωστή σειρά εκτέλεσης, διατηρώντας τη σωστή σειρά και αποφεύγοντας προβλήματα εξαρτήσεων δεδομένων. Αυτή η διαχείριση διασφαλίζει ότι οι αναδρομικές κλήσεις συντονίζονται σωστά, βελτιώνοντας την απόδοση μέσω παράλληλης εκτέλεσης.

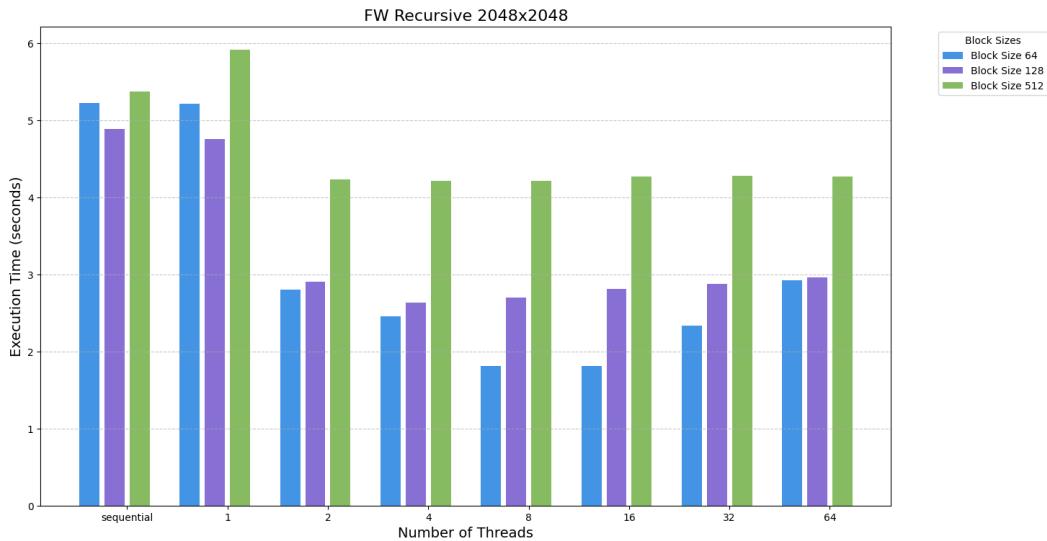
Από την εκτέλεση του αλγορίθμου **Floyd-Warshall σε αναδρομική (recursive) μορφή** προέκυψαν τα εξής 3 barplot διαγράμματα χρόνου εκτελεσης:



Διάγραμμα 2.2.1.α

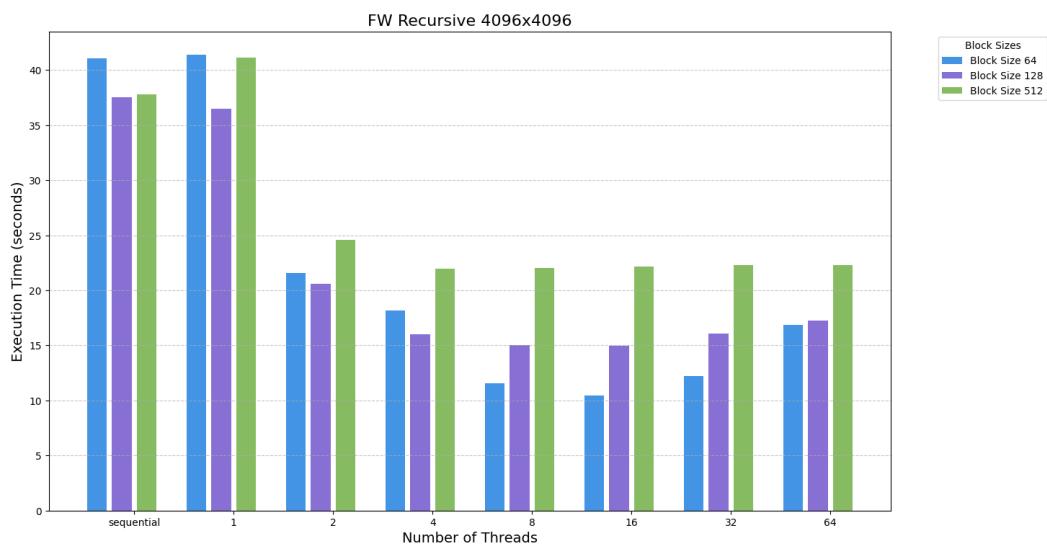
Σε ότι αφορά το **Διάγραμμα 2.2.1.α**, για μέγεθος πίνακα **1024x1024**, παρατηρούμε ότι ο χρόνος εκτέλεσης μειώνεται όσο αυξάνονται τα νήματα μέχρι να φτάσουν τα 8. Για περισσότερα των 8 νημάτων, η απόδοση επιδεινώνεται και ο χρόνος αυξάνεται ξανά. Η αύξηση του χρόνου οφείλεται σε υπερβολικό overhead από την διαχείριση πολλών νημάτων αλλά και στο γεγονός ότι το μικρό μέγεθος του πίνακα περιορίζει τη δυνατότητα αποτελεσματικής κατανομής των εργασιών σε πολλά threads. Επιπλέον, σημαντικό ρόλο

στην απόδοση του αλγορίθμου παρατηρούμε ότι διαδραματίζει και το block size. Ειδικότερα, φανερό είναι ότι το block size 64 είναι πιο αποδοτικό για μικρότερο μέγεθος πίνακα.



Διάγραμμα 2.2.1.β

Στο παραπάνω διάγραμμα (2.2.1.β), για μέγεθος πίνακα **2048x2048**, παρατηρούμε ότι υπάρχει σημαντική μείωση του χρόνου εκτέλεσης με την αύξηση των threads έως τα 16. Αυτό μας δείχνει ότι για αυτό το μέγεθος πίνακα, η παραλληλοποίηση είναι πιο αποτελεσματική, καθώς υπάρχει μεγαλύτερο περιθώριο για εκτέλεση πολλαπλών tasks ταυτόχρονα. Για περισσότερα των 16 νημάτων, η απόδοση επιδεινώνεται και ο χρόνος αυξάνεται ξανά για τους λόγους που αναφέρθηκαν και προηγουμένως. Επιπλέον, σε ότι αφορά το block size, παρατηρούμε περισσότερες διαφορές μεταξύ των block sizes όσο το μέγεθος του πίνακα μεγαλώνει. Το κατάλληλο block size εξαρτάται από το μέγεθος του πίνακα και την αρχιτεκτονική του συστήματος.



Διάγραμμα 2.2.1.γ

Στο διάγραμμα για μέγεθος πίνακα **4096x4096** (2.2.1.γ), παρατηρούμε σημαντική βελτίωση στο χρόνο εκτέλεσης για πολλά threads. Για το μεγαλύτερο μέγεθος πίνακα, η χρήση 16 threads είναι πιο αποτελεσματική. Για μεγαλύτερο αριθμό threads, πάνω από 16,

παρατηρείται μικρή αύξηση του χρόνου εκτέλεσης και επομένως μικρή επιδείνωση στην απόδοση. Αυτό μπορεί να οφείλεται στο νόμο του **Amdahl**, όπου η αύξηση του αριθμού των threads δεν οδηγεί πάντα σε ανάλογη μείωση του χρόνου εκτέλεσης, καθώς κάποια τμήματα του αλγορίθμου ενδέχεται να παραμένουν σειριακά. Ταυτόχρονα, σε μεγαλύτερους πίνακες, η αύξηση των threads ενδέχεται να έχει λιγότερη επίδραση στην μείωση του χρόνου, καθώς οι υπολογιστικοί περιορισμοί και η επικοινωνία μεταξύ των threads γίνονται πιο επιβαρυντικοί. Τέλος, για μεγαλύτερους πίνακες, όπως ο 4096x4096, το πλεονέκτημα των μικρότερων block sizes μειώνεται και η επίδοση αρχίζει να εξαρτάται περισσότερο από την αποτελεσματικότητα του συστήματος στην παραλληλοποίηση και στη διαχείριση μνήμης. Σε αυτές τις περιπτώσεις, ο χρόνος εκτέλεσης φαίνεται να είναι περισσότερο επηρεασμένος από το μέγεθος του πίνακα παρά από το block size.

Σημειώνουμε ότι ο καλύτερος χρόνος που πετύχαμε για τον πίνακα 4096x4096 ήταν ίσος με 10.43sec και προέκυψε για 16 νήματα και block size = 64.

Συνοψίζοντας, παρατηρούμε ότι για όλα τα μεγέθη πινάκων και blocks που δοκιμάσαμε η επίδοση του recursive FW δε βελτιώνεται αισθητά και για πολλά νήματα το πρόγραμμά μας δεν κλιμακώνει. Αυτό συμβαίνει, διότι τα tasks εμφανίζουν εξαρτήσεις μεταξύ τους και μας υποχρεώνουν να εισάγουμε στον κώδικα συχνά taskwaits, τα οποία, ουσιαστικά, σειριοποιούν την εκτέλεση του κώδικα. Στον αλγόριθμο k – means είχαμε πετύχει πολύ καλύτερο scalability απ' ότι τώρα. Η χρήση omp parallel for φαίνεται πως προσφέρει μεγαλύτερα οφέλη παραλληλισμού σε σχέση με τη χρήση omp tasks.

2. Tiled FW

Στη συνέχεια, υλοποιήσαμε μια παράλληλη έκδοση του tiled αλγορίθμου Floyd-Warshall χρησιμοποιώντας OpenMP tasks και πραγματοποιήσαμε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για threads = {1, 2, 4, 8, 16, 32, 64}.

Παρακάτω φαίνονται τα τμήματα του κώδικα που τροποποιήσαμε:

```
#pragma omp parallel default(shared) private(i, j)
{
    #pragma omp single
    {
        for(k=0;k<N;k+=B){
            FW(A,k,k,k,B);

            for(i=0; i<k; i+=B)
                #pragma omp task
                FW(A,k,i,k,B);

            for(i=k+B; i<N; i+=B)
                #pragma omp task
                FW(A,k,i,k,B);

            for(j=0; j<k; j+=B)
                #pragma omp task
                FW(A,k,k,j,B);

            for(j=k+B; j<N; j+=B)
                #pragma omp task
                FW(A,k,k,j,B);

            #pragma omp taskwait

            for(i=0; i<k; i+=B)
                #pragma omp task
                for(j=0; j<k; j+=B)
                    FW(A,k,i,j,B);

            for(i=0; i<k; i+=B)
                #pragma omp task
                for(j=k+B; j<N; j+=B)
                    FW(A,k,i,j,B);

            for(i=k+B; i<N; i+=B)
                #pragma omp task
                for(j=0; j<k; j+=B)
```

```

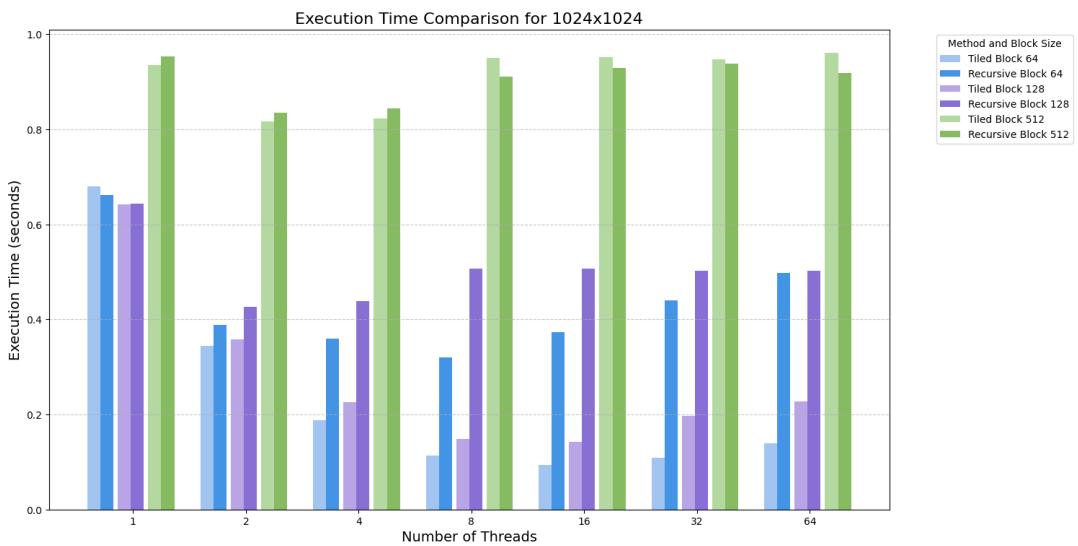
FW(A,k,i,j,B);

for(i=k+B; i<N; i+=B)
    #pragma omp task
    for(j=k+B; j<N; j+=B)
        FW(A,k,i,j,B);

    #pragma omp taskwait
}
}
}

```

Από την εκτέλεση του αλγορίθμου Floyd-Warshall σε **tiled μορφή**, δημιουργήθηκαν τα ακόλουθα 3 barplot διαγράμματα που **απεικονίζουν** και **συγκρίνουν** τους χρόνους **εκτέλεσης** του **recursive αλγορίθμου** με τον **tiled αλγόριθμο**.



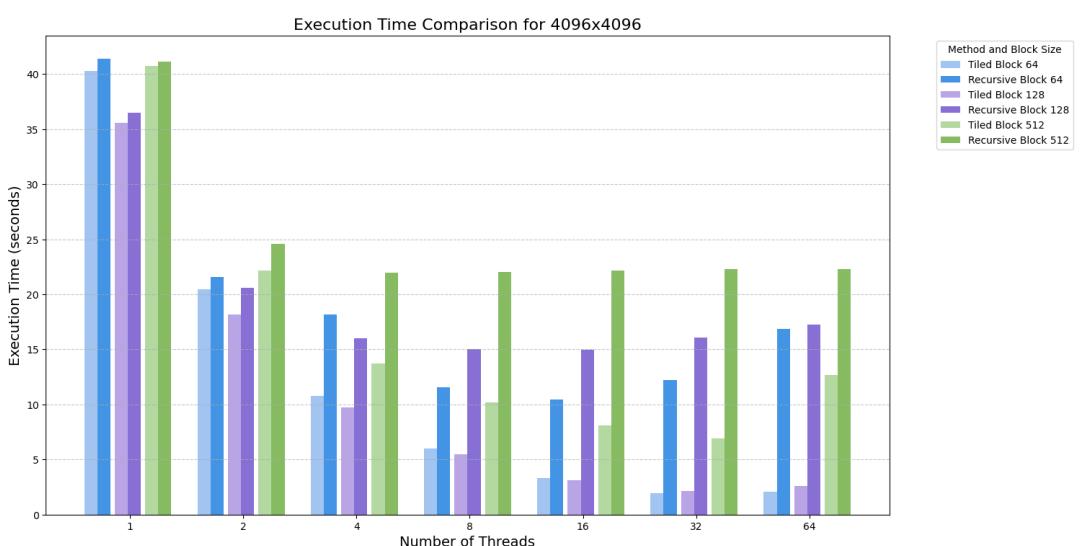
Διάγραμμα 2.2.2.α

Στο **Διάγραμμα 2.2.2.α**, για το μέγεθος του πίνακα **1024x1024**, παρατηρούμε ότι ο αλγόριθμος **Tiled** επιτυγχάνει σημαντική υπεροχή, ιδιαίτερα για μικρότερα block sizes και χαμηλότερους αριθμούς threads. Αυτό συμβαίνει, καθώς εκμεταλλεύεται καλύτερα τη μνήμη και την τοπικότητα των δεδομένων, μειώνοντας τις καθυστερήσεις στις μεταφορές δεδομένων. Παρόλα αυτά, η απόδοσή του μειώνεται καθώς ο αριθμός των νημάτων αυξάνεται. Συγκεκριμένα για 8 και περισσότερα threads και block size 512, ο χρόνος εκτέλεσης του Tiled είναι υψηλότερος από αυτόν του Recursive. Ο αλγόριθμος **Recursive** παραμένει, γενικά, πιο σταθερός στην απόδοσή του, αν και λιγότερο αποδοτικός σε σχέση με τον Tiled στα μικρότερα block sizes.



Διάγραμμα 2.2.2.β

Σε ότι αφορά το **Διάγραμμα 2.2.2.β**, για μέγεθος πίνακα **2048x2048**, παρατηρούμε ότι ο αλγόριθμος Tiled είναι πιο γρήγορος από τον Recursive για όλους τους αριθμούς νημάτων και τα block sizes. Ο χρόνος εκτέλεσης του αλγορίθμου Tiled μειώνεται για τα block sizes 64 και 128, φτάνοντας στην καλύτερη απόδοση για 32 threads και μετά παρουσιάζει ξανά μικρή αύξηση.



Διάγραμμα 2.2.2.γ

Τέλος, στο παραπάνω διάγραμμα **(2.2.2.γ)**, για μέγεθος πίνακα **4096x4096**, παρατηρούμε ότι ο αλγόριθμος Tiled παραμένει πολύ πιο γρήγορος σε σχέση με τον Recursive για όλους τους αριθμούς νημάτων και τα block sizes.

Σημειώνουμε ότι ο καλύτερος χρόνος για μέγεθος πίνακα 4096x4096 που επιτεύχθηκε ήταν ίσος με 1,95sec και προέκυψε για 32 threads και block size 64.

Παρατηρούμε, επίσης, ότι για όλα τα μεγέθη πινάκων και blocks που δοκιμάσαμε, η αύξηση των νημάτων δε βελτιώνει σημαντικά την επίδοση του προγράμματος μας. Μάλιστα, για πολλά νήματα, οι χρόνοι εκτέλεσης παραμένουν σταθεροί και το πρόγραμμά μας δεν κλιμακώνει πλέον. Για μία ακόμη φορά, βλέπουμε ότι η χρήση tasks δεν προσφέρει σημαντικά οφέλη κατά την παραλληλοποίηση. Ωστόσο, οι χρόνοι εκτέλεσης που

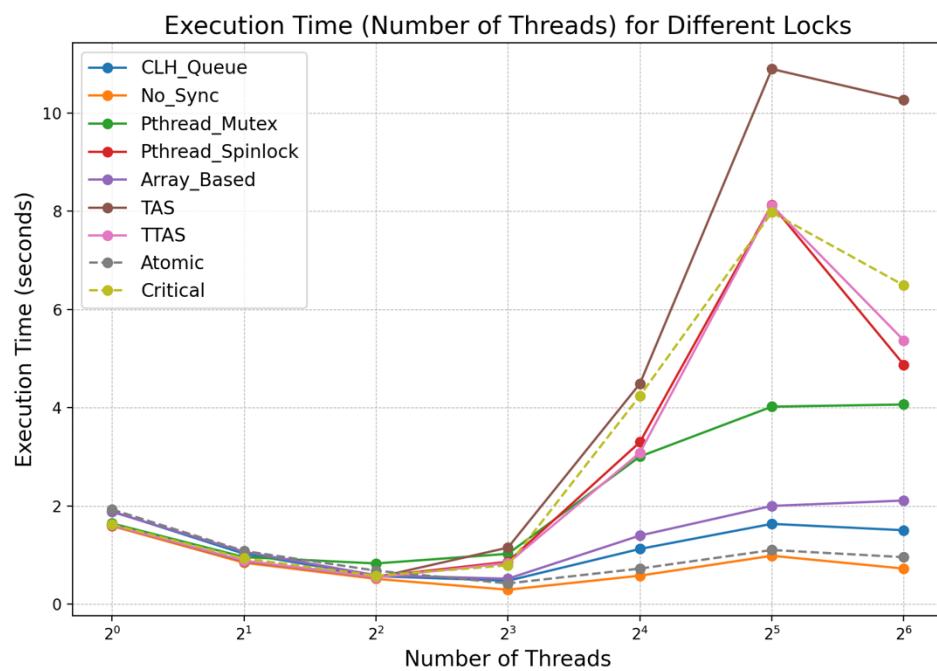
πετυχαίνουμε τώρα είναι αισθητά μικρότεροι από τους χρόνους εκτέλεσης που προέκυψαν κατά το recursive FW.

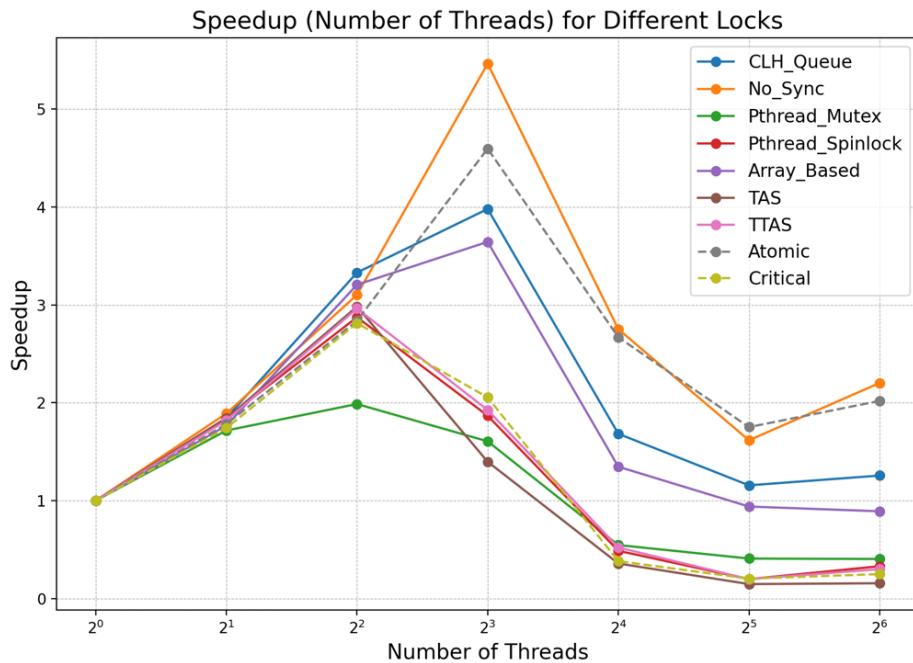
Άσκηση 3 (κομμάτι της Άσκησης 2)

Αμοιβαίος Αποκλεισμός - Κλειδώματα και Ταυτόχρονες Δομές Δεδομένων

1. Σύγκριση Κλειδωμάτων:

Αρχικά, παραθέτουμε τα συνολικά διαγράμματα χρόνου και speedup:





Στα διαγράμματα χρόνου εκτέλεσης και speedup που παρουσιάζονται για τα διάφορα κλειδώματα, παρατηρούμε ότι τα καλύτερα αποτελέσματα επιτυγχάνονται με τα κλειδώματα **CLH**, **Array** και **OMP Atomic**. Το κρίσιμο τμήμα περιλαμβάνει μόνο πράξεις προσθήκης, γεγονός που ευνοεί τη χρήση ατομικών εντολών. Οι ατομικές εντολές παρακάμπτουν τη διαδικασία κλειδώματος, εκτελώντας απευθείας τις λειτουργίες στον επεξεργαστή, επιτυγχάνοντας μεγαλύτερη απόδοση και καθιστώντας για το συγκεκριμένο κρίσιμο τμήμα το OMP Atomic ως τον πιο γρήγορο μηχανισμό.

Στη συνέχεια, γίνεται αναλυτική αξιολόγηση των κλειδωμάτων σε ζεύγη, βασισμένη στην ομοιότητα των χαρακτηριστικών τους.

Nosync Lock:

Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα. Ωστόσο, θα χρησιμοποιηθεί ως άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων.

A) Pthread_Mutex vs Pthread_Spinlock:

Pthread_Mutex_Lock:

Το pthread_mutex είναι ένας μηχανισμός αμοιβαίου αποκλεισμού που παρέχεται από τη βιβλιοθήκη POSIX Threads. Βασίζεται σε μια ατομική μεταβλητή που δηλώνει αν το κλείδωμα είναι κατειλημμένο ή διαθέσιμο. Όταν ένα νήμα προσπαθεί να αποκτήσει το κλείδωμα και αυτό είναι κατειλημμένο, τίθεται σε κατάσταση ύπνου μέχρι να απελευθερωθεί.

Τα mutexes είναι κατάλληλα για μεγάλα κρίσιμα τμήματα κώδικα, καθώς αποφεύγουν την περιττή κατανάλωση πόρων της CPU μέσω του sleep. Ωστόσο, η διαχείριση τους μέσω του λειτουργικού συστήματος αυξάνει το κόστος, καθιστώντας τα λιγότερο αποδοτικά για μικρά κρίσιμα τμήματα, όπου προτιμώνται πιο ελαφριές μέθοδοι, όπως τα spinlocks.

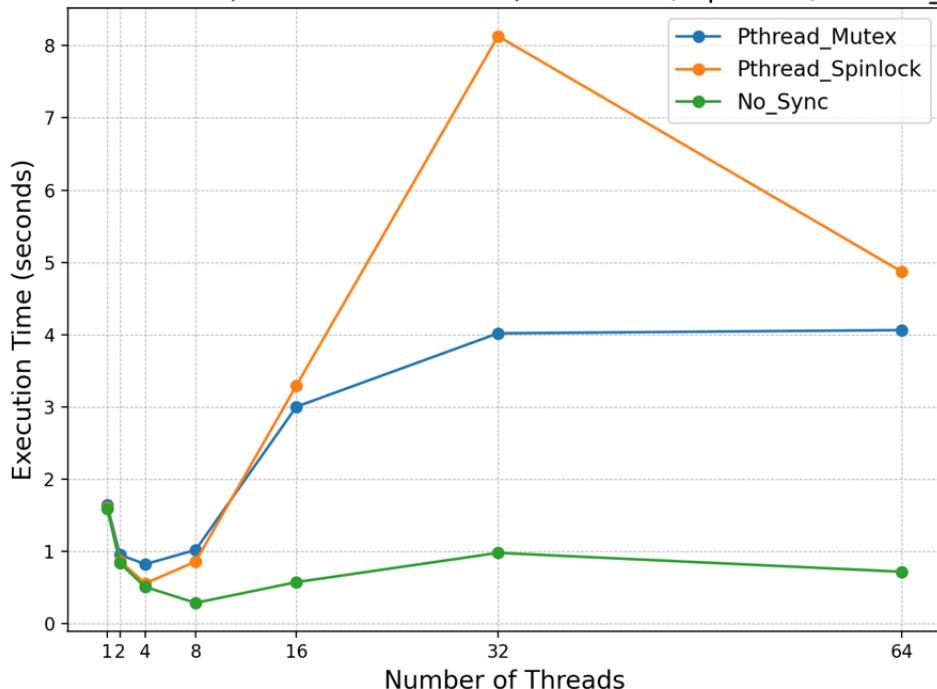
Συμπερασματικά, τα mutexes είναι αξιόπιστα για συγχρονισμό, αλλά η χρήση τους πρέπει να γίνεται με κριτήριο το μέγεθος του κρίσιμου τμήματος και το κόστος διαχείρισης.

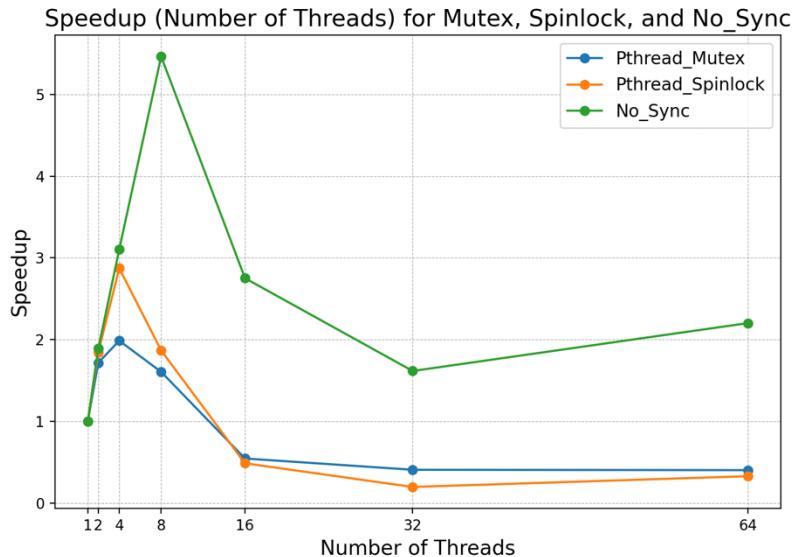
Pthread_Spinlock:

Το `pthread_spinlock` αποτελεί έναν τύπο κλειδώματος που παρέχεται από τη βιβλιοθήκη POSIX Threads, σχεδιασμένο για να εξασφαλίζει αμοιβαίο αποκλεισμό. Βασίζεται σε μια ατομική μεταβλητή που υποδεικνύει την κατάσταση του κλειδώματος (ανοιχτό ή κλειστό). Σε περίπτωση που ένα νήμα προσπαθήσει να αποκτήσει το κλείδωμα και αυτό δεν είναι διαθέσιμο, το νήμα παραμένει ενεργό σε κατάσταση busy-waiting, εκτελώντας συνεχείς ελέγχους για την απελευθέρωσή του.

Η συνεχής αυτή αναμονή καταναλώνει πόρους της CPU, γεγονός που καθιστά τα spinlocks κατάλληλα κυρίως για σύντομα κρίσιμα τμήματα κώδικα, όπου η καθυστέρηση από τη διαχείριση του λειτουργικού συστήματος (όπως στα mutexes) θα ήταν πιο δαπανηρή από την ενεργή αναμονή.

Execution Time (Number of Threads) for Mutex, Spinlock, and No_Sync





Η αύξηση του αριθμού των νημάτων από 1 σε 2 και στη συνέχεια σε 4 οδηγεί σε μείωση του χρόνου εκτέλεσης και βελτίωση του speedup, αξιοποιώντας αποτελεσματικά τις δυνατότητες του παραλληλισμού. Ωστόσο, η περαιτέρω αύξηση των νημάτων προκαλεί αντίστροφη επίδραση, καθώς ο χρόνος εκτέλεσης αυξάνεται και το speedup μειώνεται.

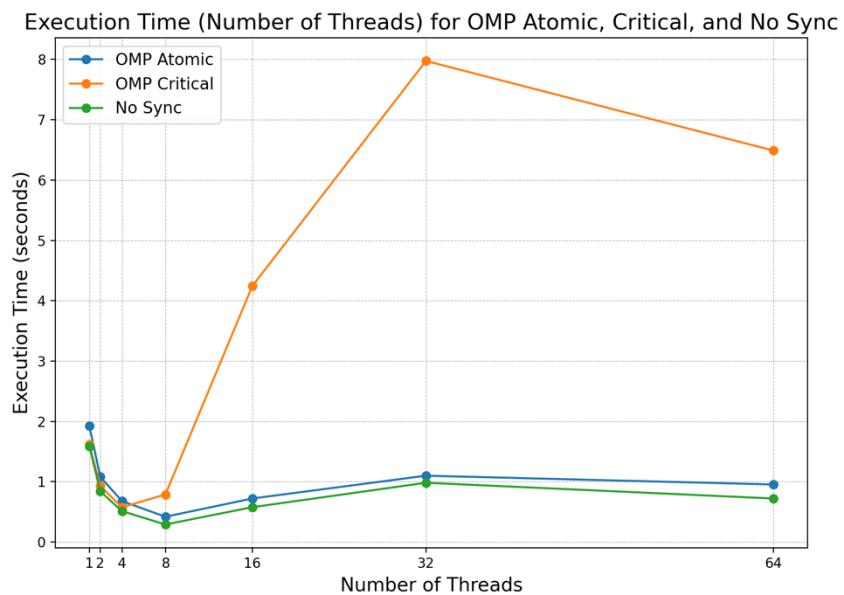
Αυτό συμβαίνει επειδή, όσο αυξάνεται ο αριθμός των νημάτων, εντείνεται ο ανταγωνισμός για την πρόσβαση στο κλείδωμα, με αποτέλεσμα μεγαλύτερες καθυστερήσεις λόγω της αναμονής. Στην περίπτωση του spinlock, η συνεχής περιστροφή (spinning) επιβαρύνει την κατανάλωση πόρων της CPU, ενώ στα mutex, η διαδικασία "κούμησης" και "αφύπνισης" των νημάτων από το λειτουργικό σύστημα δημιουργεί πρόσθετο κόστος. Αυτοί οι περιορισμοί υπονομεύουν την αποδοτικότητα της πολυνηματικής εκτέλεσης, με την υπερβολική αύξηση των νημάτων να εξαλείφει τελικά τα οφέλη του παραλληλισμού.

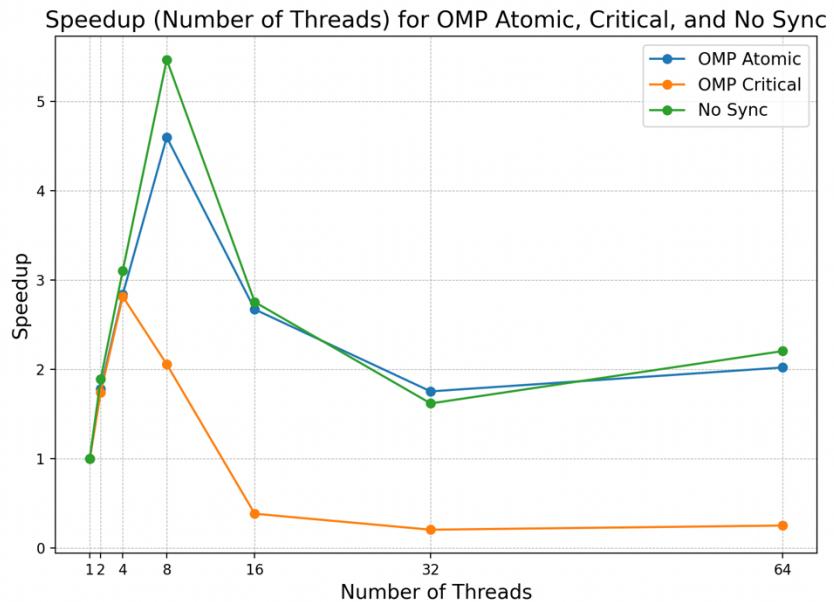
Παρόλο που στην θεωρία θα περιμέναμε λόγω της φύσης του κρίσιμου τμήματος που έχουμε, το execution time για spin lock να είναι σε κάθε περίπτωση μικρότερο, στην πράξη παρατηρούμε ότι σε κάποιες περιπτώσεις γίνεται το αντίθετο. Όταν έχουμε μικρό κρίσιμο τμήμα και αυξημένο αριθμό νημάτων, έχουμε "καλύτερη" επίδοση στο mutex, καθώς ο χρόνος εκτέλεσης εξαρτάται και από παράγοντες όπως η κατανομή εργασιών όπου σε περιπτώσεις όπου τα νήματα εκτελούν διαφορετικές εργασίες, η χρήση mutex μπορεί να οδηγήσει σε καλύτερη απόδοση. Αυτό συμβαίνει, καθώς η CPU μπορεί να χρησιμοποιηθεί για άλλες εργασίες κατά την αναμονή, αντί να σπαταλάται σε busy waiting. Ακόμα, όταν το κρίσιμο τμήμα είναι αρκετά μικρό, το κόστος του mutex (ύπνου και αφύπνισης) μπορεί να είναι συγκρίσιμο με το κόστος του spin, ειδικά αν οι υπολογιστικοί πόροι δεν είναι υπερφορτωμένοι. Σε αυτή την περίπτωση, το mutex lock έχει καλύτερη συμπεριφορά λόγω της αποφυγής "φαινομένων συμφόρησης" στην CPU. Τέλος, ο επεξεργαστής και η αρχιτεκτονική του συστήματος μπορούν να επηρεάσουν τα αποτελέσματα. Εάν το σύστημα διαθέτει αποτελεσματικό μηχανισμό διαχείρισης ύπνου και αφύπνισης, τότε το mutex lock μπορεί να είναι πιο αποδοτικό από το spinlock σε ορισμένες περιπτώσεις.

B) Omp Atomic vs Omp Critical:

OMP Atomic: Το OMP Atomic αποτελεί την υλοποίηση ατομικών εντολών στο OpenMP. Οι ατομικές εντολές πραγματοποιούνται απευθείας στον επεξεργαστή, χωρίς την ανάγκη παρέμβασης του λειτουργικού συστήματος. Αυτό έχει ως αποτέλεσμα μικρότερο κόστος και ταχύτερη εκτέλεση, ειδικά για απλές λειτουργίες όπως οι προσθέσεις ή οι αφαιρέσεις. Λόγω της ελαφρότητας αυτής της μεθόδου, το OMP Atomic είναι κατάλληλο για μικρά κρίσιμα τμήματα κώδικα που δεν απαιτούν σύνθετο συγχρονισμό.

OMP Critical Section: Το OMP Critical Section είναι η υλοποίηση κρίσιμου τμήματος στο OpenMP. Επιτρέπει τη διαχείριση του αμοιβαίου αποκλεισμού, εξασφαλίζοντας ότι μόνο ένα νήμα μπορεί να εκτελέσει το κρίσιμο τμήμα κάθε φορά. Παρόλο που προσφέρει μεγαλύτερη ευελιξία συγκριτικά με τις ατομικές εντολές, το κόστος του είναι υψηλότερο, καθώς ενδέχεται να προκαλέσει σημαντική συμφόρηση όταν χρησιμοποιείται σε περιβάλλοντα με μεγάλο αριθμό νημάτων. Για το λόγο αυτό, η χρήση του συνίσταται σε μεγαλύτερα ή πιο σύνθετα κρίσιμα τμήματα κώδικα όπου απαιτείται αυστηρότερος συγχρονισμός.





Το OMP Critical παρουσιάζει σημαντική αύξηση στον χρόνο εκτέλεσης καθώς αυξάνεται ο αριθμός των νημάτων, σε αντίθεση με το OMP Atomic, το οποίο εμφανίζει σχεδόν σταθερούς χρόνους με ελάχιστες αυξήσεις. Αυτό οφείλεται στο γεγονός ότι το OMP Critical υλοποιείται μέσω κλειδώματος, με αποτέλεσμα, όσο περισσότερα νήματα συμμετέχουν, τόσο μεγαλύτερος να είναι ο ανταγωνισμός για την απόκτηση του κλειδώματος. Αυτό οδηγεί σε αυξημένες καθυστερήσεις λόγω της αναμονής.

Από την άλλη πλευρά, το OMP Atomic βασίζεται σε ατομικές εντολές που υποστηρίζονται απευθείας από το hardware. Αυτές οι εντολές εκτελούνται χωρίς τη μεσολάβηση κλειδώματος ή του λειτουργικού συστήματος, γεγονός που μειώνει σημαντικά τις καθυστερήσεις. Ως αποτέλεσμα, το OMP Atomic είναι πιο αποδοτικό για μικρά κρίσιμα τμήματα, ειδικά σε περιβάλλοντα με μεγάλο αριθμό νημάτων.

C) TAS vs TTAS locks:

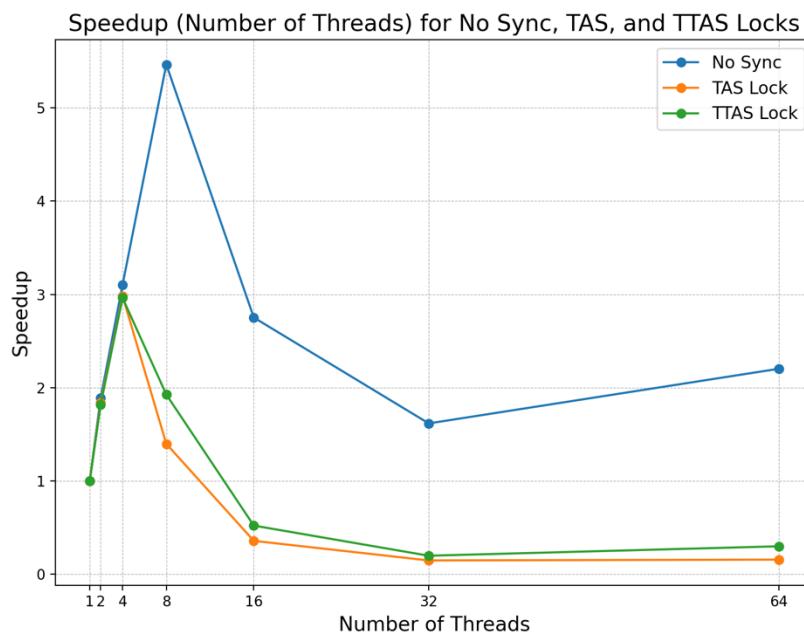
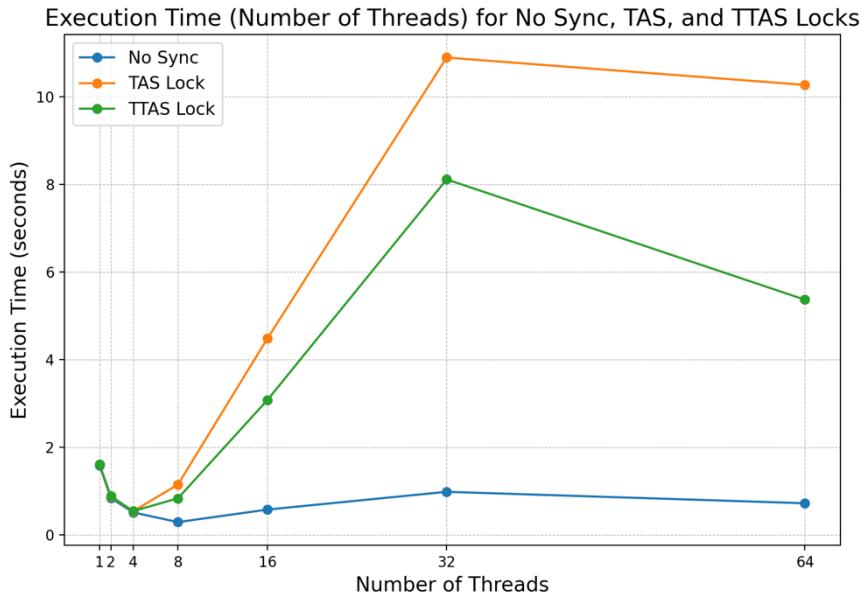
Test and Set Lock:

Το TAS Lock είναι ένας απλός αλγόριθμος συγχρονισμού που χρησιμοποιεί την ατομική εντολή test-and-set. Κάθε νήμα προσπαθεί συνεχώς να αποκτήσει το κλείδωμα, εκτελώντας busy-waiting μέχρι το κλείδωμα να απελευθερωθεί. Παρόλο που είναι εύκολο στην υλοποίηση, η συνεχής προσπάθεια απόκτησης του κλειδώματος δημιουργεί έντονη συμφόρηση στον διάδρομο δεδομένων, ειδικά σε περιβάλλοντα με μεγάλο αριθμό νημάτων. Το TAS είναι κατάλληλο μόνο για μικρά κρίσιμα τμήματα, όπου ο χρόνος κατοχής του κλειδώματος είναι ελάχιστος.

Test and Test and Set Lock:

Το TTAS Lock είναι μια βελτιωμένη εκδοχή του TAS, σχεδιασμένη για να μειώνει τη συμφόρηση στον διάδρομο δεδομένων. Πριν επιχειρήσει να αποκτήσει το κλείδωμα, ένα νήμα ελέγχει πρώτα τοπικά αν το κλείδωμα είναι ελεύθερο. Μόνο αν το κλείδωμα φαίνεται διαθέσιμο, εκτελεί την εντολή test-and-set. Αυτή η προσέγγιση μειώνει τη συχνότητα

ενημερώσεων της μνήμης, βελτιώνοντας την αποδοτικότητα σε περιβάλλοντα υψηλού παραλληλισμού. Παρά τη βελτίωση, εξακολουθεί να βασίζεται στο busy-waiting, καταναλώνοντας πόρους της CPU όταν το κλείδωμα δεν είναι διαθέσιμο.



Καθώς αυξάνεται ο αριθμός των νημάτων, το TAS παρουσιάζει χαμηλότερη απόδοση και περιορισμένη κλιμάκωση σε σύγκριση με το TTAS. Το TAS βασίζεται στην ατομική λειτουργία test-and-set, η οποία προσπαθεί να αποκτήσει το κλείδωμα τροποποιώντας τη μεταβλητή state. Ακόμα και αν η προσπάθεια αποτύχει, προκαλεί cache invalidation σε όλα τα νήματα που περιμένουν, κάτι που αυξάνει τη συμφόρηση στον διάδρομο δεδομένων, ιδιαίτερα όταν υπάρχει μεγάλος αριθμός νημάτων.

Από την άλλη πλευρά, το TTAS λειτουργεί πιο αποδοτικά, καθώς πριν επιχειρήσει να αλλάξει την κατάσταση της μεταβλητής state, ελέγχει επανειλημμένα την τρέχουσα τιμή της για να βεβαιωθεί ότι το κλείδωμα είναι ελεύθερο. Μόνο όταν το κλείδωμα είναι διαθέσιμο εκτελεί την ατομική εντολή test-and-set. Αυτός ο τρόπος λειτουργίας μειώνει δραστικά τα cache invalidations, καθώς τα νήματα αποφεύγουν να τροποποιήσουν την κατάσταση χωρίς λόγο, οδηγώντας σε καλύτερη απόδοση και λιγότερη συμφόρηση.

Παρόλο που έχουν διαφορετικά χαρακτηριστικά, και οι δύο τύποι κλειδώματος μοιράζονται ένα κοινό μειονέκτημα: την έλλειψη "δικαιοσύνης". Αυτό σημαίνει ότι υπάρχει κίνδυνος κάποιο νήμα να μην καταφέρει ποτέ να αποκτήσει πρόσβαση στο κρίσιμο τμήμα, ένα φαινόμενο γνωστό ως starvation, το οποίο γίνεται πιο έντονο σε συνθήκες υψηλού φόρτου.

Συμπερασματικά, το TTAS προσφέρει σημαντική βελτίωση στην αποδοτικότητα σε σχέση με το TAS σε πολυνηματικά περιβάλλοντα, καθώς αποφεύγει περιπτές ενημερώσεις της cache. Ωστόσο, η έλλειψη δικαιοσύνης περιορίζει τη χρήση τους σε εφαρμογές όπου το starvation δεν αποτελεί σημαντικό πρόβλημα.

D) Array vs CLH locks:

Array Lock:

Το Array Lock είναι μια μέθοδος συγχρονισμού που βασίζεται σε μια κοινή ατομική μεταβλητή tail και έναν πίνακα flag, ο οποίος μοιράζεται μεταξύ όλων των νημάτων. Κάθε στοιχείο του πίνακα αντιστοιχεί σε ένα συγκεκριμένο νήμα. Όταν ένα νήμα επιθυμεί να αποκτήσει το κλείδωμα, αυξάνει την τιμή της μεταβλητής tail και ελέγχει τη θέση που του αντιστοιχεί στον πίνακα. Αν η τιμή στη συγκεκριμένη θέση είναι true, το νήμα αποκτά το κλείδωμα. Εάν όχι, εισέρχεται σε κατάσταση busy-waiting και περιμένει μέχρι η τιμή να αλλάξει σε true.

Κατά την απελευθέρωση του κλειδώματος, το νήμα θέτει τη δική του θέση στον πίνακα flag σε false και ενεργοποιεί την επόμενη θέση θέτοντάς την σε true. Αυτό διασφαλίζει ότι το δικαίωμα εισόδου στο κρίσιμο τμήμα περνάει στο επόμενο νήμα με βάση τη σειρά εισαγωγής, υλοποιώντας μια πολιτική First Come First Served (FCFS).

Το Array Lock ξεχωρίζει για την αποφυγή συμφόρησης στον διάδρομο δεδομένων, καθώς κάθε νήμα κάνει spin στη δική του ξεχωριστή θέση μνήμης αντί να τροποποιεί συνεχώς μια κοινή μεταβλητή. Αυτό μειώνει τα cache invalidations, βελτιώνοντας την απόδοση σε συστήματα με πολλούς πυρήνες. Επιπλέον, ο σχεδιασμός του εξασφαλίζει δικαιοσύνη, καθώς τα νήματα εξυπηρετούνται με τη σειρά που αιτούνται το κλείδωμα, αποτρέποντας φαινόμενα starvation.

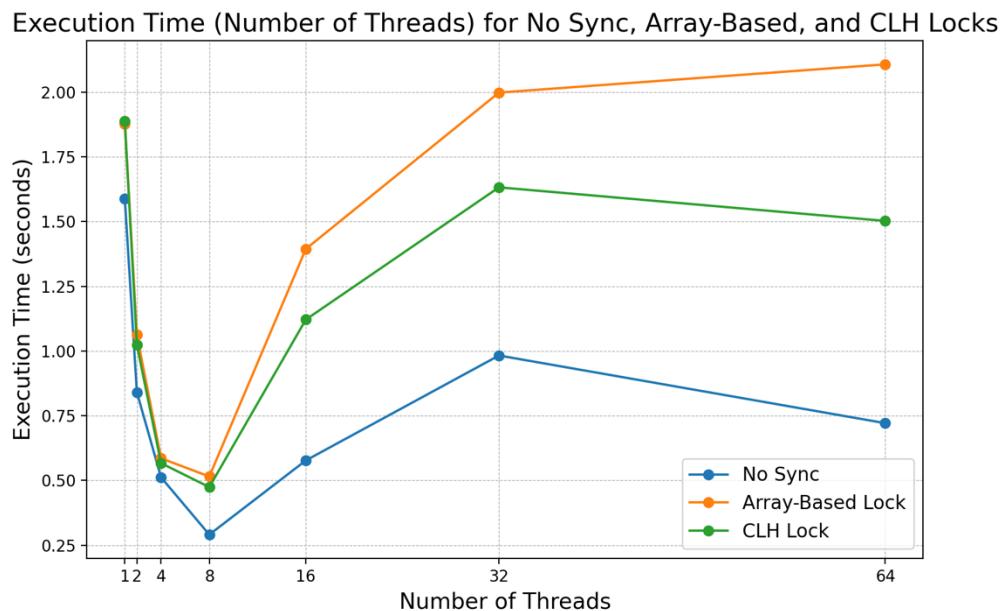
CLH Lock:

Το CLH Lock βασίζεται σε μια δομή συνδεδεμένης λίστας, όπου κάθε νήμα έχει έναν δικό του κόμβο καθώς και έναν αναφοράς στον προηγούμενο κόμβο του στη λίστα (predecessor). Μια κοινή ατομική μεταβλητή, γνωστή ως tail, χρησιμεύει ως δείκτης στον τελευταίο κόμβο που προστέθηκε στη λίστα.

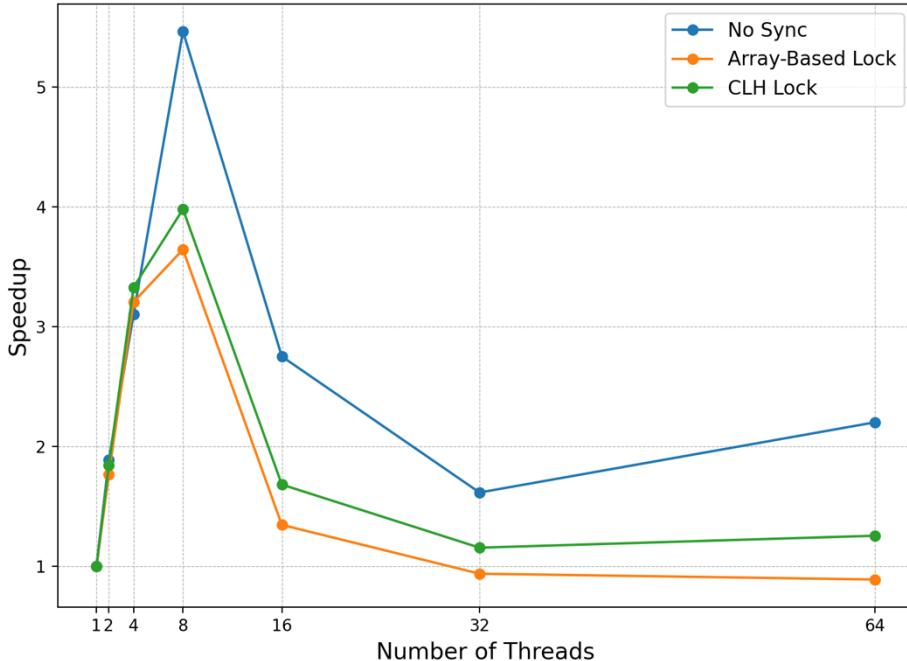
Όταν ένα νήμα επιθυμεί να αποκτήσει το κλείδωμα, δημιουργεί τον δικό του κόμβο και θέτει το πεδίο locked του κόμβου του σε true. Στη συνέχεια, προσθέτει τον κόμβο του στο τέλος της λίστας, ενημερώνοντας τη μεταβλητή tail. Από εκείνη τη στιγμή, το νήμα αρχίζει να ελέγχει (spin-waiting) την κατάσταση του πεδίου locked του κόμβου που προηγείται στη λίστα, δηλαδή του predecessor. Όταν το πεδίο locked του προηγούμενου κόμβου αλλάξει σε false, το νήμα αποκτά το κλείδωμα και μπορεί να εισέλθει στο κρίσιμο τμήμα.

Για να απελευθερώσει το κλείδωμα, το νήμα ενημερώνει το πεδίο `locked` του δικού του κόμβου σε `false`. Αυτό επιτρέπει στο επόμενο νήμα στη λίστα να παρατηρήσει την αλλαγή και να αποκτήσει το κλείδωμα. Με αυτόν τον τρόπο, εξασφαλίζεται η σειρά εξυπηρέτησης First Come First Served (FCFS), η οποία αποτρέπει φαινόμενα starvation, καθώς κάθε νήμα εξυπηρετείται με τη σειρά που ζητάει το κλείδωμα.

Ένα σημαντικό πλεονέκτημα του CLH Lock είναι ότι μειώνει τη συμφόρηση στον διάδρομο δεδομένων, καθώς κάθε νήμα κάνει spin μόνο πάνω στη μεταβλητή του προηγούμενου κόμβου και όχι σε μια κοινή μεταβλητή. Αυτό σημαίνει ότι τα spin-waiting νήματα αλληλεπιδρούν με διαφορετικές τοποθεσίες μνήμης, μειώνοντας τα cache invalidations και αυξάνοντας την απόδοση σε περιβάλλοντα με πολλούς πυρήνες.



Speedup (Number of Threads) for No Sync, Array-Based, and CLH Locks



Καθώς αυξάνεται ο αριθμός των νημάτων πέρα από τα 8, παρατηρείται αύξηση του χρόνου εκτέλεσης και στα δύο κλειδώματα, CLH και Array. Στην περίπτωση του CLH Lock, όταν ένα νήμα απελευθερώνει το κλείδωμα, ενημερώνεται η cache του επόμενου νήματος (successor), καθώς αυτό εκτελεί συνεχώς spin πάνω στο πεδίο locked του προηγούμενου κόμβου (predecessor). Αυτό προκαλεί cache invalidation, επηρεάζοντας την απόδοση.

Στο Array Lock, η χρήση ενός κοινόχρονου πίνακα flag δημιουργεί το πρόβλημα του false sharing, όπου διαφορετικά νήματα επεξεργάζονται θέσεις του πίνακα που βρίσκονται στην ίδια cache line, οδηγώντας σε συχνά cache invalidations. Για να μετριαστεί αυτό, εφαρμόζεται padding, δηλαδή προστίθενται κενά διαστήματα ανάμεσα στις θέσεις του πίνακα. Παρόλο που αυτό μειώνει τη συχνότητα των cache invalidations, το πρόβλημα δεν εξαλείφεται εντελώς.

Το CLH Lock εμφανίζει καλύτερη απόδοση για περισσότερα από 8 νήματα, διότι κάθε νήμα διατηρεί τον κόμβο του τοπικά στη μνήμη του, ενώ στο Array Lock ο πίνακας flag είναι κοινόχρονος από όλα τα νήματα. Αυτή η κοινή χρήση του πίνακα οδηγεί σε μεγαλύτερη συμφόρηση στον διάδρομο δεδομένων, μειώνοντας την αποδοτικότητα του Array Lock όταν αυξάνεται ο αριθμός των νημάτων.

Συνοπτικά, η τοπική διαχείριση μνήμης στο CLH Lock προσφέρει πλεονεκτήματα σε σενάρια υψηλού παραλληλισμού, ενώ το Array Lock είναι πιο επιρρεπές σε προβλήματα συμφόρησης, περιορίζοντας την απόδοσή του σε περιβάλλοντα με πολλά νήματα.

2. Ταυτόχρονες Δομές Δεδομένων:

Σκοπός του συγκεκριμένου ερωτήματος είναι η μελέτη διαφόρων ταυτόχρονων υλοποιήσεων μιας απλά συνδεδεμένης λίστας και συγκεκριμένα η αξιολόγηση της επίδοσής τους κάτω από διαφορετικές συνθήκες.

Αρχικά, εκτελέσαμε την σειριακή έκδοση του κώδικα και έπειτα τις ταυτόχρονες υλοποιήσεις μίας απλά συνδεδεμένης ταξινομημένης λίστας. Τα αποτελέσματα των εκτελέσεων αυτών παρουσιάζονται στα διαγράμματα (διαγράμματα throughput per thread count) που προέκυψαν. Για όλες τις εκδοχές του κώδικα χρησιμοποιήσαμε τις ακόλουθες τιμές των παραμέτρων. Συγκεκριμένα, για τα νήματα οι τιμές που χρησιμοποιήσαμε είναι οι: 1, 2, 4, 8, 16, 32, 64, 128, για το μέγεθος της λίστας: 1024, 8192 και για το ποσοστό λειτουργιών: 100-0-0, 80-10-10, 20-40-40, 0-50-50, όπου ο πρώτος αριθμός υποδηλώνει το ποσοστό αναζητήσεων, ο επόμενος το ποσοστό εισαγωγών και ο τελευταίος το ποσοστό διαγραφών.

Serial

Η εκτέλεση της λίστας στη σειριακή της εκδοχή χαρακτηρίζεται από την απουσία οποιουδήποτε κλειδώματος ή πολυπλοκότητας συγχρονισμού. Αυτό σημαίνει ότι προκύπτει μέγιστη απόδοση για μικρό αριθμό λειτουργιών ή αναζητήσεων, καθώς δεν υπάρχει overhead συγχρονισμού. Ωστόσο, δεν παρέχει δυνατότητα παράλληλης εκτέλεσης, επομένως για εφαρμογές με μεγάλο αριθμό νημάτων δεν είναι εξίσου αποδοτική.

Στον ακόλουθο πίνακα παρουσιάζονται τα αποτελέσματα της σειριακής έκδοσης του κώδικα:

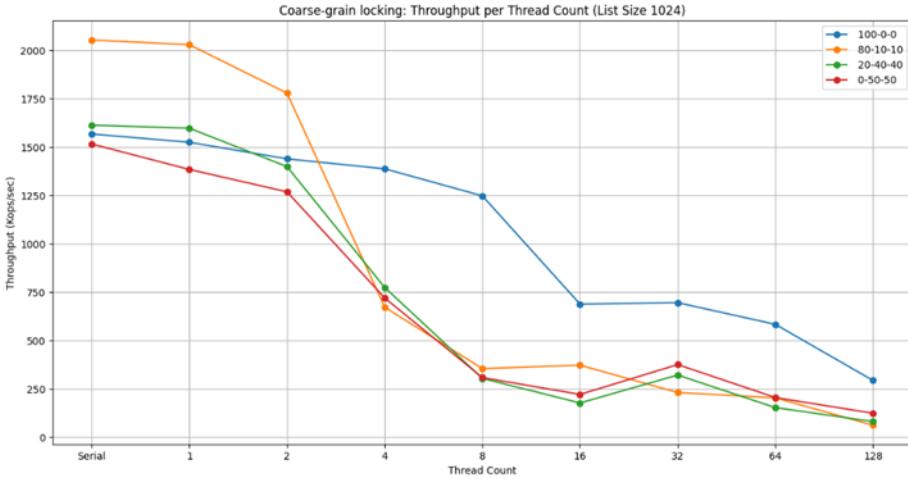
Μέγεθος Λίστας	Ποσοστό Λειτουργιών	Throughput (Kops/sec)
1024	100-0-0	1567.09
	80-10-10	2052.58
	20-40-40	1612.64
	0-50-50	1515.39
8192	100-0-0	152.73
	80-10-10	74.98
	20-40-40	79.73
	0-50-50	91.33

Για μέγεθος λίστας 1024 παρατηρούμε ότι η σειριακή υλοποίηση είναι αποτελεσματική και εμφανίζει throughput της τάξης των 1500-2000 Kops/sec με μέγιστο το 2052.58 για ποσοστό λειτουργιών 80-10-10. Για μεγαλύτερο μέγεθος λίστας (8192) παρατηρούμε μια καθολική σημαντική μείωση του throughput.

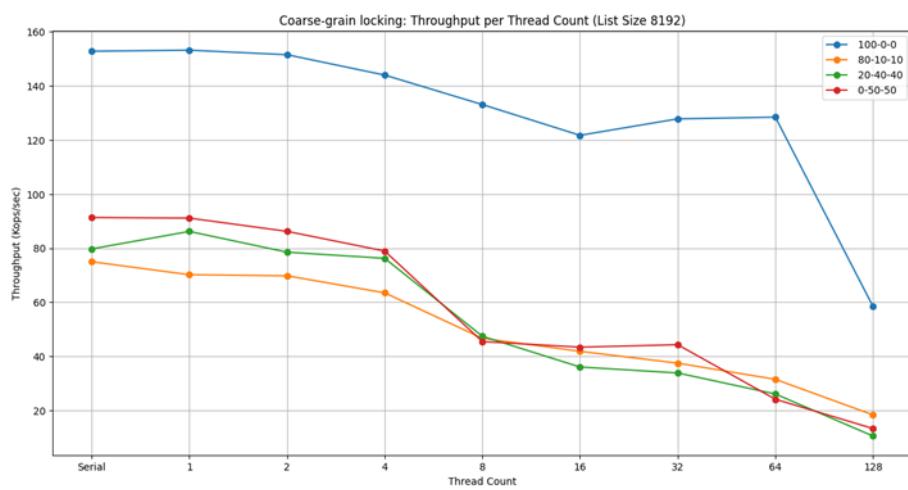
Coarse-grain locking

Το Coarse-grain locking χαρακτηρίζεται από την εύκολη υλοποίησή του καθώς, γίνεται χρήση ενός κλειδώματος για όλη τη δομή. Ωστόσο, το γεγονός αυτό, περιορίζει την παράλληλη πρόσβαση των νημάτων στη δομή.

Ακολουθούν τα διαγράμματα Throughput per Thread Count για μεγέθη λίστας 1024 και 8192:



Σε ότι αφορά το παραπάνω διάγραμμα, παρατηρούμε ότι για μικρό αριθμό νημάτων, η απόδοση είναι αρκετά καλή για όλα τα ποσοστά λειτουργιών. Ωστόσο, καθώς αυξάνεται ο αριθμός των νημάτων, το throughput μειώνεται σημαντικά. Αυτό συμβαίνει διότι, όσο αυξάνεται ο αριθμός των νημάτων, όλο και περισσότερα από αυτά προσπαθούν να αποκτήσουν πρόσβαση στη λίστα μέσω του ίδιου κλειδώματος (contention). Το γεγονός αυτό οδηγεί σε κατάσταση αναμονής για τα περισσότερα νήματα, δημιουργώντας καθυστερήσεις που μειώνουν το συνολικό throughput. Η καλύτερη απόδοση παρατηρείται για το ποσοστό λειτουργιών 100-0-0, το οποίο περιλαμβάνει μόνο αναζητήσεις. Οι αναζητήσεις είναι πιο αποδοτικές, καθώς δεν υπάρχει η ανάγκη αποκλειστικής πρόσβασης. Αντίθετα, τα ποσοστά λειτουργιών που περιλαμβάνουν πολλές εισαγωγές ή διαγραφές, όπως 0-50-50, έχουν σημαντικά μειωμένο throughput καθώς απαιτούν αποκλειστική πρόσβαση στη λίστα, αυξάνοντας τις καθυστερήσεις λόγω της συχνής σύγκρουσης για το ίδιο κλείδωμα. Σε σχέση με το Serial, το Coarse-grain locking αποδίδει το ίδιο καλά μόνο όταν υπάρχει μόνο 1 νήμα για όλα τα ποσοστά λειτουργιών. Αντίθετα, όταν ο αριθμός των νημάτων αυξάνεται, το contention στα κλειδώματα δημιουργεί σημαντική καθυστέρηση, μειώνοντας το throughput σε επίπεδα χαμηλότερα από το Serial.



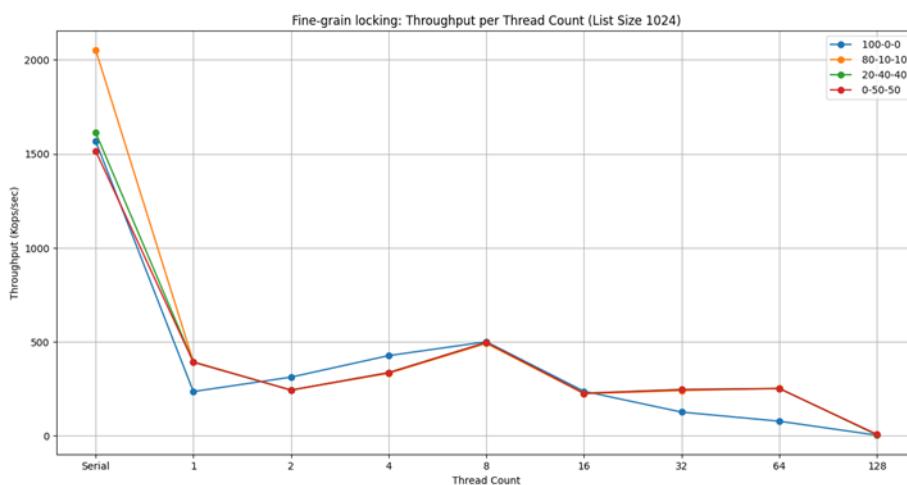
Στο διάγραμμα για μέγεθος λίστας 8192, παρατηρούμε παρόμοια πτώση με προηγουμένως στο throughput με την αύξηση των νημάτων να είναι πιο έντονη λόγω του μεγαλύτερου

μεγέθους λίστας. Συγκεκριμένα, η καλύτερη απόδοση παρατηρείται για ακόμα μία φορά στο ποσοστό λειτουργίας 100-0-0, με την πτώση στο throughput να είναι πιο έντονη λόγω του overhead που συνεπάγεται η λίστα μεγαλύτερου μεγέθους. Για τα ποσοστά λειτουργίας με πολλές αλλαγές (εισαγωγές-διαγραφές) (80-10-10, 20-40-40, 0-50-50) παρατηρούμε ότι έχουν χαμηλή απόδοση, ειδικά για υψηλούς αριθμούς νημάτων, αφού το μεγάλο μέγεθος λίστας αυξάνει τις καθυστερήσεις και μειώνει το throughput.

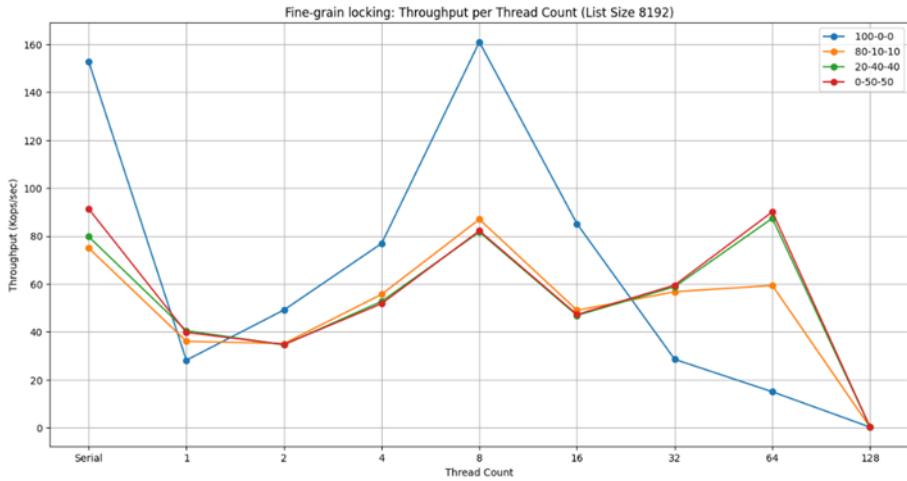
Fine-grain locking

Το Fine-Grain Locking βελτιώνει τη δυνατότητα παράλληλης πρόσβασης, επιτρέποντας σε διαφορετικά τμήματα της λίστας να κλειδωθούν ανεξάρτητα. Ωστόσο, η πολυπλοκότητα και το overhead των πολλαπλών κλειδωμάτων επηρεάζουν την απόδοση.

Ακολουθούν τα διαγράμματα Throughput per Thread Count για μεγέθη λίστας 1024 και 8192:



Σε ότι αφορά το παραπάνω διάγραμμα, παρατηρούμε ότι η μέγιστη απόδοση εμφανίζεται για 8 νήματα. Αυτό οφείλεται στο ότι η χρήση πολλαπλών κλειδωμάτων του Fine-grain locking επιτρέπει την ταυτόχρονη πρόσβαση σε διαφορετικά τμήματα της λίστας, μειώνοντας τις καθυστερήσεις. Ωστόσο, πέρα από τα 8 νήματα, το overhead του συγχρονισμού αυξάνεται, με αποτέλεσμα να μειώνεται η απόδοση. Στα ποσοστά λειτουργιών με ενημερώσεις και διαγραφές (80-10-10, 20-40-40, 0-50-50), παρατηρείται αυξημένη απόδοση σε σύγκριση με το Coarse-grain, αφού η δυνατότητα παράλληλης πρόσβασης επιτρέπει την εκτέλεση πολλών αλλαγών ταυτόχρονα. Αντίθετα, στο ποσοστό 100-0-0, η απόδοση είναι χαμηλότερη από το Coarse-grain, καθώς οι αναζητήσεις δεν επωφελούνται ιδιαίτερα από τα πολλαπλά κλειδώματα, ενώ το overhead του Fine-grain locking μειώνει την απόδοση.



Στο διάγραμμα για μέγεθος λίστας 8192, παρατηρούμε παρόμοια συμπεριφορά με το διάγραμμα για το μέγεθος λίστας 1024, ωστόσο το throughput μειώνεται ακόμα περισσότερο για υψηλό αριθμό νημάτων. Το μεγάλο μέγεθος λίστας αυξάνει την πολυπλοκότητα της διαχείρισης των πολλαπλών κλειδωμάτων, με αποτέλεσμα μεγαλύτερο overhead. Παρόλα αυτά, τα ποσοστά λειτουργιών με αλλαγές (80-10-10, 20-40-40, 0-50-50) διατηρούν σχετικά καλή απόδοση σε σύγκριση με το Coarse-grain. Για ποσοστά λειτουργίας (100-0-0, 80-10-10), που περιλαμβάνουν κυρίως αναζητήσεις, η υψηλότερη τιμή προκύπτει για αριθμό νημάτων ίσο με 8, ενώ σε ποσοστά λειτουργιών που περιλαμβάνουν πολλές αλλαγές στη λίστα (20-40-40, 0-50-50) για αριθμό νημάτων 64. Αυτό οφείλεται στην ικανότητα του Fine-grain locking να επιτρέπει την παράλληλη πρόσβαση σε διαφορετικά τμήματα της λίστας, περιορίζοντας το contention. Ωστόσο, στις υπόλοιπες τιμές των νημάτων, το overhead του συγχρονισμού και η πολυπλοκότητα του Fine-grain locking οδηγούν σε μείωση της απόδοσης.

Optimistic Synchronization

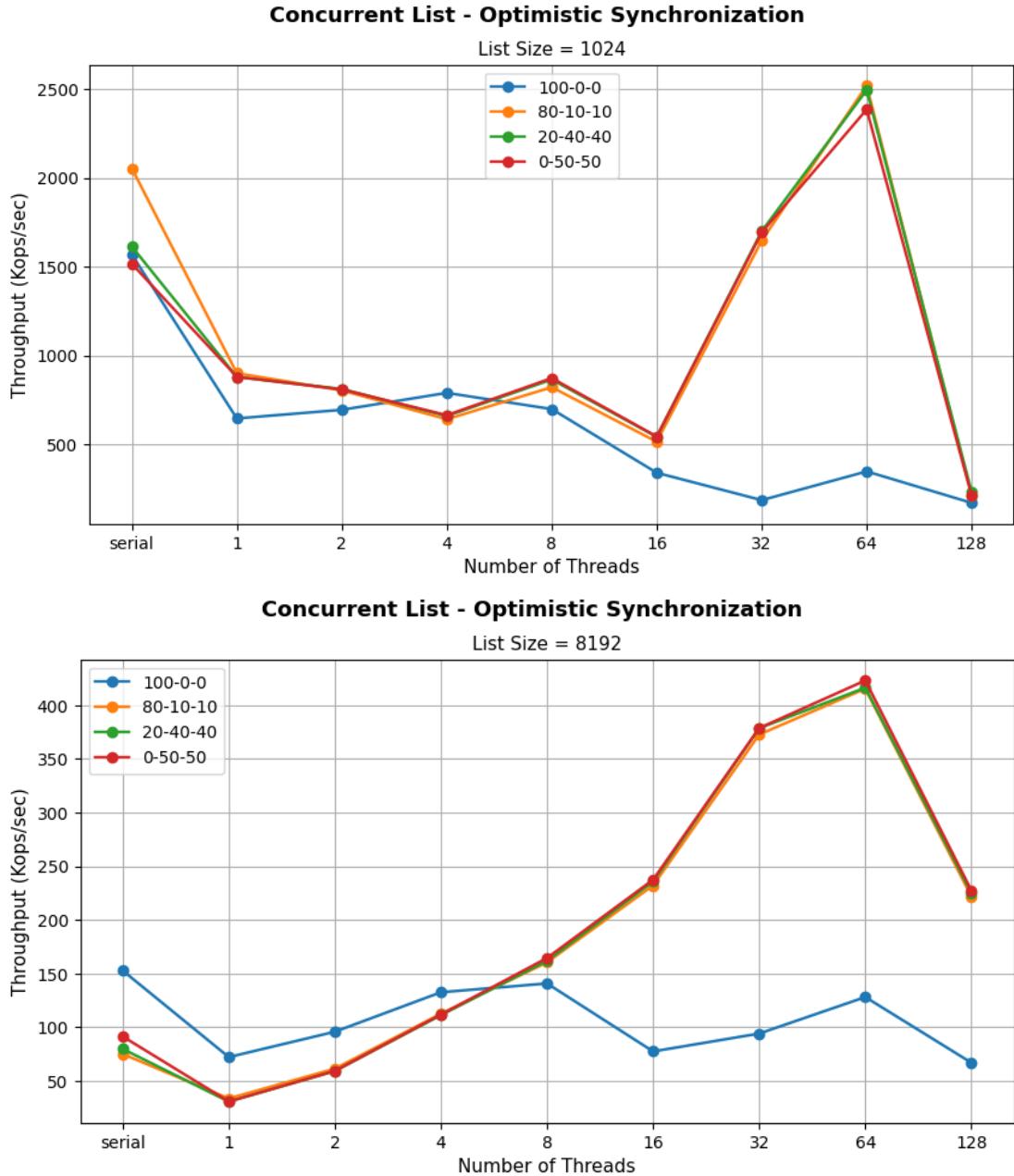
Όπως είδαμε και προηγουμένως, το fine – grain locking παρουσιάζει καλύτερη επίδοση από το coarse – grain locking. Ωστόσο, ακόμη και σε αυτήν τη νέα υλοποίηση υπάρχουν αρκετά κλειδώματα, με αποτέλεσμα, πολλές φορές, νήματα που προσπελαύνουν διακριτά μέρη της λίστας να αδυνατούν να εργαστούν ταυτόχρονα. Σε αυτό το υποερώτημα, θα εξετάσουμε μία νέα βελτιωμένη υλοποίηση, η οποία χρησιμοποιεί optimistic synchronization.

Κατά το optimistic synchronization, διασχίζουμε τη λίστα χωρίς να χρησιμοποιούμε κανένα κλείδωμα. Μόλις εντοπίσουμε τους κόμβους που μας ενδιαφέρουν, τότε κλειδώνουμε μονάχα αυτούς και πραγματοποιούμε τη λειτουργία που επιθυμούμε (contains(), add() ή remove()). Έπειτα, όμως, πρέπει να βεβαιωθούμε ότι η λίστα παραμένει συνεπής και ότι δεν παρουσιάστηκαν conflicts με άλλα νήματα που επιτελούσαν άλλες λειτουργίες στη λίστα. Γι' αυτό το λόγο, διατρέχουμε άλλη μία φορά τη λίστα από την αρχή, μέσω της ειδικής συνάρτησης validate(), και σε περίπτωση ασυνέπειας, επαναλαμβάνουμε τη διαδικασία που περιγράφηκε.

Επομένως, κατά το optimistic synchronization χρησιμοποιούμε λιγότερα κλειδώματα απ' ότι στο fine – grain locking. Αναμένουμε, επομένως, ότι το optimistic synchronization θα παρουσιάζει καλύτερη επίδοση από το fine – grain locking, εφόσον, επιπλέον, η διάσχιση της

λίστας δύο φορές χωρίς τη χρήση κλειδωμάτων αποδειχθεί ταχύτερη από τη διάσχιση της λίστας μία φορά και με χρήση κλειδωμάτων.

Εκτελούμε τον κώδικα που μας έχει δοθεί και με τις παραμέτρους που μας ζητούνται. Στα παρακάτω διαγράμματα, απεικονίζουμε το ρυθμό λειτουργιών του προγράμματος στη μονάδα του χρόνου συναρτήσει του πλήθους των νημάτων, για τις διάφορες παραμέτρους που χρησιμοποιήσαμε:



Μελετώντας και τα δύο διαγράμματα, παρατηρούμε ότι το optimistic synchronization παρουσιάζει υψηλότερο throughput, και συνεπώς καλύτερη επίδοση, από το fine – grain locking για τις ίδιες παραμέτρους. Αυτό είναι λογικό, διότι, όπως εξηγήσαμε και νωρίτερα, οι δύο διασχίσεις δίχως κλειδώματα που πραγματοποιεί ο αλγόριθμος του optimistic synchronization είναι ταχύτερες από τη μία διάσχιση με κλειδώματα που πραγματοποιεί το fine – grain locking.

Εξετάζοντας το πρώτο διάγραμμα, παρατηρούμε συχνά scalability breaks, ενώ για το ποσοστό λειτουργιών “100 – 0 – 0”, το πρόγραμμά μας δεν κλιμακώνει. Ωστόσο, για τα

υπόλοιπα ποσοστά λειτουργιών, επιτυγχάνεται μέγιστο throughput για πλήθος νημάτων ίσο με 64. Τα συχνά scalability breaks μπορεί να οφείλονται στα πολλά conflicts που παρατηρούνται μεταξύ των νημάτων, τα οποία οδηγούν σε ασυνέπειες και σε αυξημένο πλήθος διασχίσεων της λίστας και επαναλήψεων των λειτουργιών. Παρ' όλα αυτά, το optimistic synchronization παραμένει καλύτερο σε επίδοση από το fine – grain και coarse – grain locking.

Μελετώντας το δεύτερο διάγραμμα, παρατηρούμε ότι για ποσοστά λειτουργιών “80 – 10 – 10”, “20 – 40 – 40” και “0 – 50 – 50”, το πρόγραμμά μας κλιμακώνει, αν και παρουσιάζει scalability break στα 128 νήματα. Η βελτίωση αυτή μπορεί να οφείλεται στο γεγονός ότι, τώρα, χρησιμοποιούμε μεγαλύτερο μέγεθος λίστας και, άρα, ελαττώνεται η πιθανότητα εμφάνισης conflicts μεταξύ των διαφορετικών νημάτων. Ωστόσο, επειδή, τώρα, η λίστα είναι μεγαλύτερη, η διάσχισή της είναι πιο χρονοβόρα, με αποτέλεσμα το throughput να είναι μικρότερο από το throughput του πρώτου διαγράμματος. Σημειώνουμε ότι, και πάλι, για το ποσοστό λειτουργιών “100 – 0 – 0”, το πρόγραμμά μας δεν κλιμακώνει.

Lazy Synchronization

Όπως είδαμε και νωρίτερα, η χρήση optimistic synchronization αυξάνει σημαντικά την επίδοση του προγράμματός μας, αλλά, παρ' όλα αυτά, παρουσιάζει κάποια μειονεκτήματα. Ενδεικτικά, η συνάρτηση validate(), μέσω της οποίας ελέγχουμε τη συνέπεια της λίστας μας, διατρέχει τη λίστα από την αρχή, κάτι που είναι αρκετά χρονοβόρο. Επίσης, χρησιμοποιούμε και αρκετά κλειδώματα και, κυρίως, η συνάρτηση contains() ίσως να μπορούσε να υλοποιηθεί χωρίς κανένα κλείδωμα. Η λύση σε αυτούς τους προβληματισμούς δίνεται από το lazy synchronization, το οποίο αποτελεί μετεξέλιξη του optimistic synchronization.

Στο lazy synchronization, κάθε κόμβος της λίστας διαθέτει ένα επιπλέον πεδίο, ένα dirty bit. Το dirty bit είναι μία boolean μεταβλητή, η οποία μας υποδεικνύει εάν ο κόμβος αυτός βρίσκεται στη λίστα ή εάν έχει διαγραφεί. Συνεπώς, η συνάρτηση contains() διατρέχει τη λίστα χωρίς τη χρήση κλειδωμάτων και ελέγχει την τιμή του dirty bit. Η συνάρτηση add() διατρέχει τη λίστα, επίσης, χωρίς τη χρήση κλειδωμάτων και, μόλις εντοπίσει τους κόμβους που χρειάζεται, κλειδώνει μονάχα αυτούς και τους τροποποιεί κατάλληλα. Έπειτα, η συνάρτηση add() πρέπει να βεβαιωθεί ότι η λίστα παραμένει συνεπής και, γι' αυτό το λόγο, καλείται η συνάρτηση validate(). Η συνάρτηση validate() διατρέχει τη λίστα από την αρχή, αλλά, σε αντίθεση με το optimistic synchronization, πραγματοποιεί τοπικούς ελέγχους στους κόμβους pred και curr. Τέλος, η συνάρτηση remove() πραγματοποιείται σε δύο βήματα (lazy). Αρχικά, η remove() διασχίζει τη λίστα χωρίς τη χρήση κλειδωμάτων και μόλις εντοπίσει τους κόμβους που χρειάζεται, κλειδώνει μονάχα αυτούς και εκτελεί τη διαδικασία διαγραφής σε δύο βήματα:

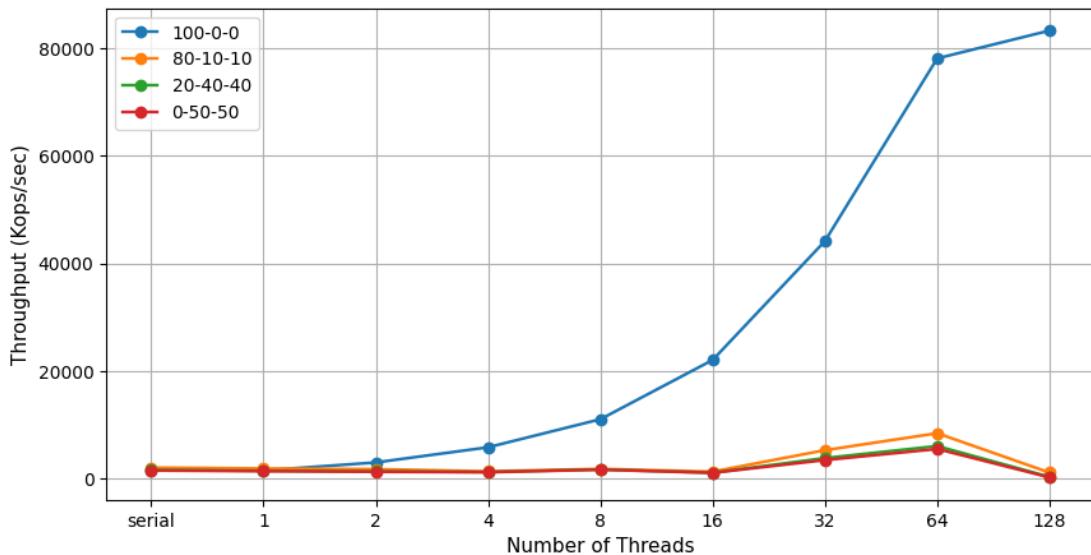
- αρχικά, θέτει το dirty bit του υπό διαγραφή κόμβου ίσο με true (λογική διαγραφή) και,
- ύστερα, διαγράφει φυσικά τον κόμβο, επαναθέτοντας τους κατάλληλους δείκτες.

Κατά τη remove(), πρέπει, και πάλι, να εξασφαλιστεί ότι η λίστα παραμένει συνεπής και γι' αυτό το λόγο, καλείται η συνάρτηση validate().

Αφού μελετήσαμε πώς λειτουργεί το lazy synchronization, είμαστε έτοιμοι να εκτελέσουμε τις προσομοιώσεις, σύμφωνα με τις οδηγίες της εκφώνησης. Στα παρακάτω διαγράμματα, απεικονίζουμε το throughput του προγράμματός μας συναρτήσει του πλήθους των νημάτων:

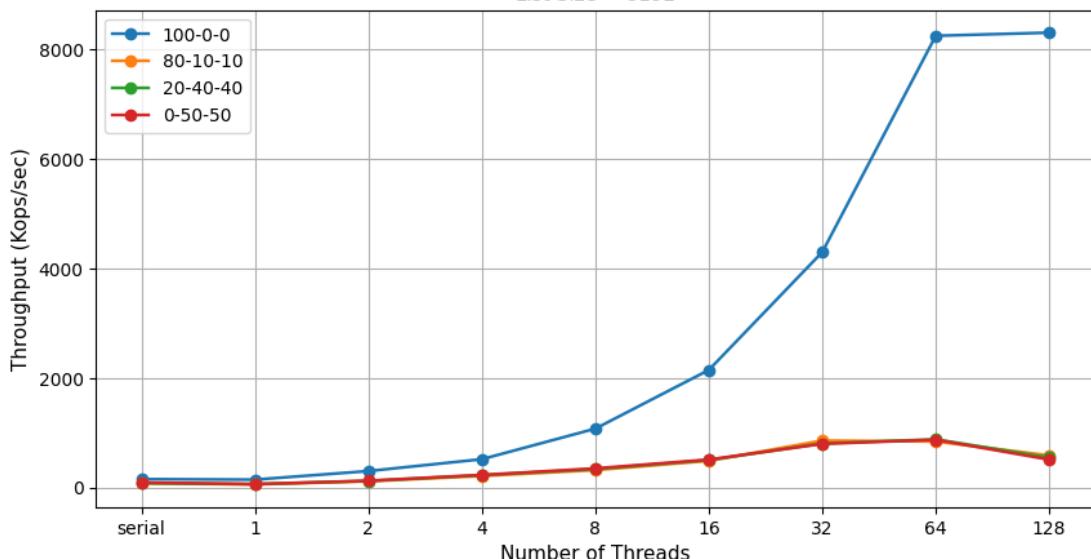
Concurrent List - Lazy Synchronisation

List Size = 1024



Concurrent List - Lazy Synchronisation

List Size = 8192

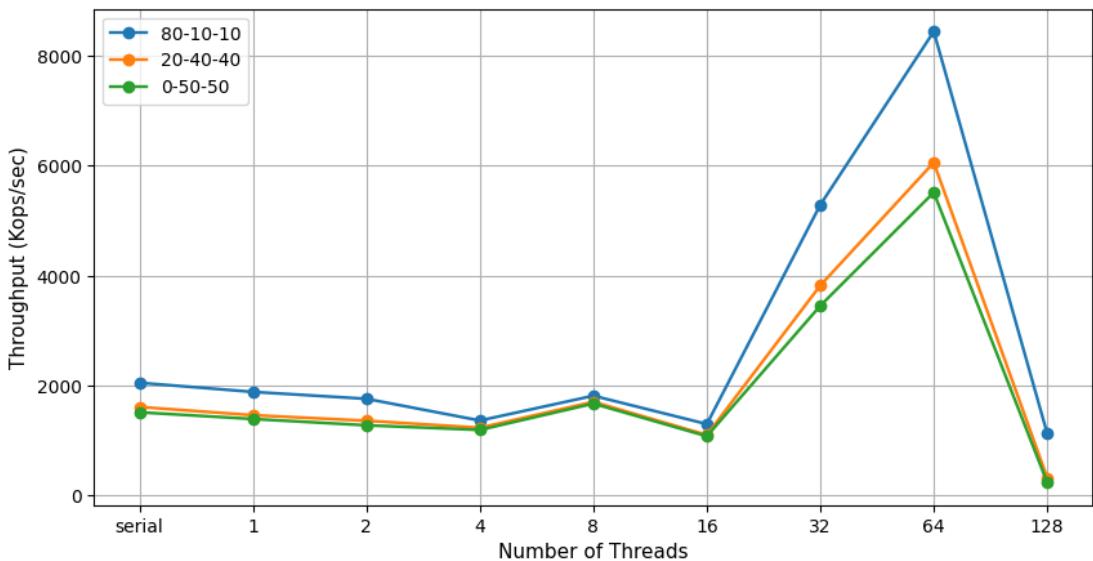


Αρχικά, και για τα δύο διαγράμματα, θα μελετήσουμε το ποσοστό λειτουργιών “100 – 0 – 0”. Το throughput, τώρα, είναι πολύ μεγαλύτερο απ’ ότι ήταν στο optimistic synchronization. Υπενθυμίζουμε ότι το ποσοστό λειτουργιών “100 – 0 – 0” πραγματοποιεί μονάχα αναζητήσεις – contains() στη λίστα. Στο lazy optimization, η συνάρτηση contains() δε χρησιμοποιεί κανένα κλείδωμα και δεν καλεί τη συνάρτηση validate() για έλεγχο συνέπειας της λίστας. Συνεπώς, τώρα, η συνάρτηση contains() εκτελείται πολύ ταχύτερα απ’ ότι στο optimistic synchronization. Παρατηρούμε, επίσης, ότι η προσθήκη περισσότερων νημάτων, αυξάνει το throughput του προγράμματος και, άρα, το πρόγραμμά μας κλιμακώνει για ποσοστό λειτουργιών “100 – 0 – 0”.

Επειδή, όμως, το throughput για το ποσοστό λειτουργιών “100 – 0 – 0” είναι πολύ μεγάλο, δεν μπορούμε να παρατηρήσουμε τι συμβαίνει με τα υπόλοιπα ποσοστά λειτουργιών, όταν τα απεικονίζουμε όλα σε ένα κοινό σύστημα αξόνων. Γι’ αυτό το λόγο, επανασχεδιάζουμε τα διαγράμματα που φαίνονται παραπάνω, αλλά αυτή τη φορά, δε θα απεικονίσουμε το ποσοστό λειτουργιών “100 – 0 – 0”. Οπότε, παράγουμε τα παρακάτω διαγράμματα:

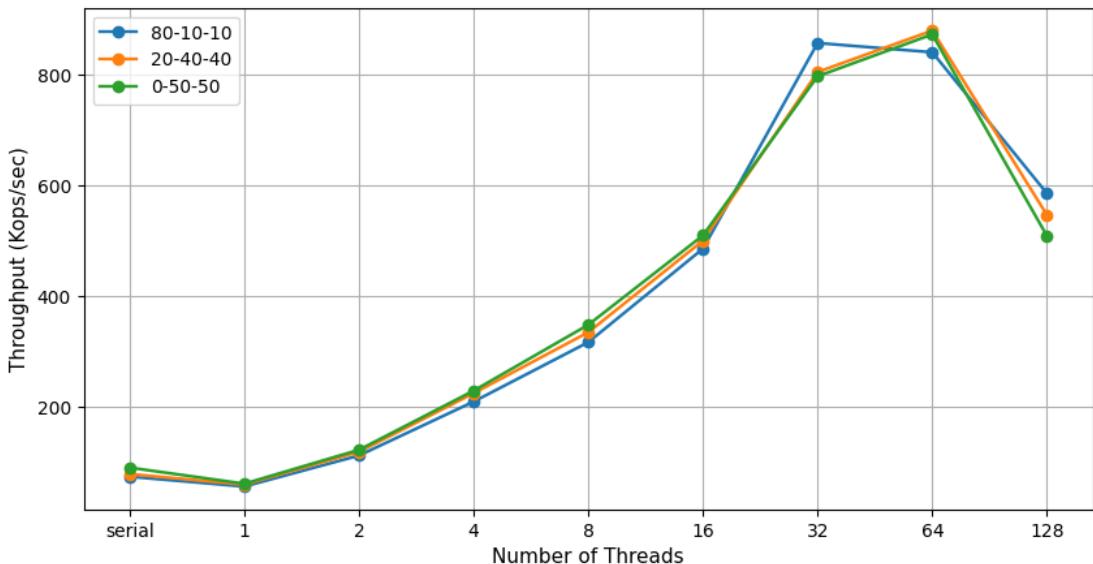
Concurrent List - Lazy Synchronisation

List Size = 1024



Concurrent List - Lazy Synchronisation

List Size = 8192



Και στα δύο διαγράμματα, πετυχαίνουμε throughput πολύ μεγαλύτερο απ' ότι με optimistic synchronization, όπως και αναμέναμε. Αυτό συμβαίνει, διότι, πλέον, η συνάρτηση validate() εκτελείται ταχύτερα, καθώς πραγματοποιεί τοπικούς ελέγχους μόνο στους κόμβους pred και curr και όχι σε όλη τη λίστα.

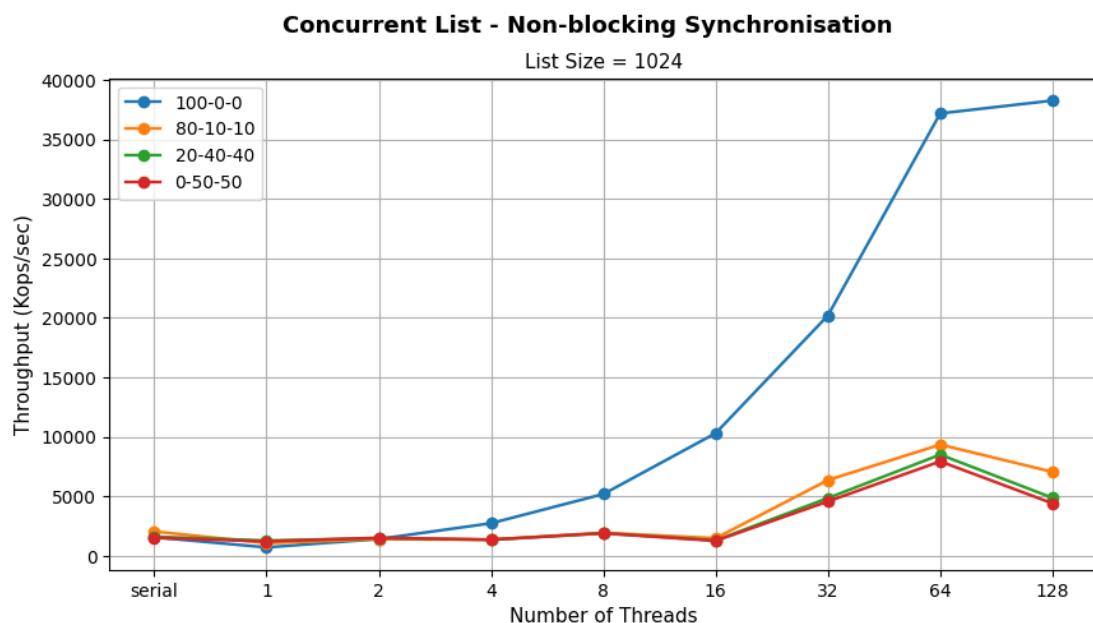
Αν μελετήσουμε πιο προσεκτικά το πρώτο διάγραμμα, παρατηρούμε ότι το πρόγραμμά μας δεν κλιμακώνει, καθώς παρουσιάζονται συχνά scalability breaks. Αυτό συμβαίνει, διότι είναι πιθανό να παρουσιάζονται πολλά conflicts μεταξύ των διάφορων νημάτων, τα οποία οδηγούν σε ασυνέπειες και υποχρεώνουν το πρόγραμμά μας να προσπαθήσει πάλι να πραγματοποιήσει τις λειτουργίες add() ή remove(). Αντιθέτως, στο δεύτερο διάγραμμα, εργαζόμαστε με μία μεγαλύτερη λίστα και παρατηρούμε ότι το πρόγραμμά μας κλιμακώνει. Αυτό συμβαίνει, διότι η λίστα έχει μεγαλύτερο μήκος και, κατά συνέπεια, μειώνεται η πιθανότητα εμφάνισης conflicts μεταξύ διαφορετικών λειτουργιών από διαφορετικά νήματα.

Σημειώνουμε, επίσης, ότι το throughput της μεγάλης λίστας είναι πάντα μικρότερο από το throughput της μικρότερης λίστας, διότι απαιτείται περισσότερος χρόνος για να διασχίσουμε μία μεγάλη λίστα απ' ότι μία μικρότερη.

Non – blocking Syncrhonization

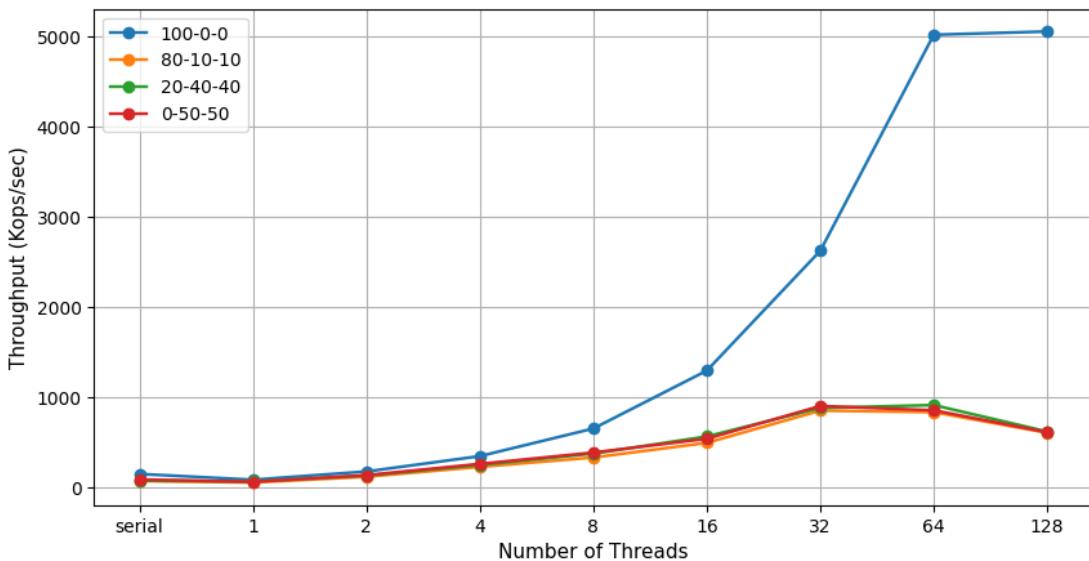
Όπως έχουμε ήδη διαπιστώσει, η χρήση κλειδωμάτων, αν και πολλές φορές απαραίτητη για το συγχρονισμό των διάφορων λειτουργιών, συχνά προσθέτει ανεπιθύμητες καθυστερήσεις στο πρόγραμμά μας. Επομένως, μία άλλη προσέγγιση στο πρόβλημα που μελετάμε θα ήταν η υλοποίηση απλά συνδεδεμένης λίστας που επιτρέπει non – blocking synchronization. Δηλαδή, τώρα, δε θα χρησιμοποιήσουμε πουθενά κανένα κλείδωμα. Για να μπορέσει να συμβεί αυτό, και πάλι, κάθε κόμβος της λίστας θα διαθέτει ένα επιπλέον πεδίο, μία boolean μεταβλητή που ονομάζουμε marked bit. Επομένως, σε κάθε κόμβο, τα πεδία marked και next θα τα χειριζόμαστε ως ατομικές μονάδες και οποιαδήποτε ενημέρωση πραγματοποιείται σε αυτά θα πρέπει να γίνεται ατομικά. Αν προσπαθήσουμε να ενημερώσουμε το πεδίο next, ενώ το marked είναι true (ο κόμβος δεν υπάρχει), τότε η προσπάθεια αυτή θα θεωρείται αποτυχημένη. Συνεπώς, η συνάρτηση remove(), για να διαγράψει έναν κόμβο, θα θέτει ατομικά το marked bit ίσο με true και θα πραγματοποιεί μία μόνο απόπειρα να ενημερώσει κατάλληλα τους δείκτες. Κάθε νήμα που διατρέχει τη λίστα για οποιαδήποτε από τις τρεις λειτουργίες (add(), remove(), contains()) θα διαγράφει φυσικά όποιον κόμβο συναντά και διαθέτει marked bit ίσο με true. Σημειώνουμε ότι σε περίπτωση αποτυχημένης ατομικής ενημέρωσης, οδηγούμαστε σε επαναπροσάθεια, διατρέχοντας από την αρχή τη λίστα.

Αφού εξηγήσαμε σύντομα πώς λειτουργεί το non – blocking synchronization, τώρα μπορούμε να εκτελέσουμε τις προσομοιώσεις που μας έχουν ζητηθεί και να εξετάσουμε αν η νέα υλοποίηση είναι καλύτερη από το lazy synchronization, το οποίο μέχρι τώρα μας είχε δώσει τα καλύτερα αποτελέσματα. Στα παρακάτω διαγράμματα απεικονίζουμε το throughput συναρτήσει του πλήθους των νημάτων για τις διάφορες προσομοιώσεις που εκτελούμε:



Concurrent List - Non-blocking Synchronisation

List Size = 8192

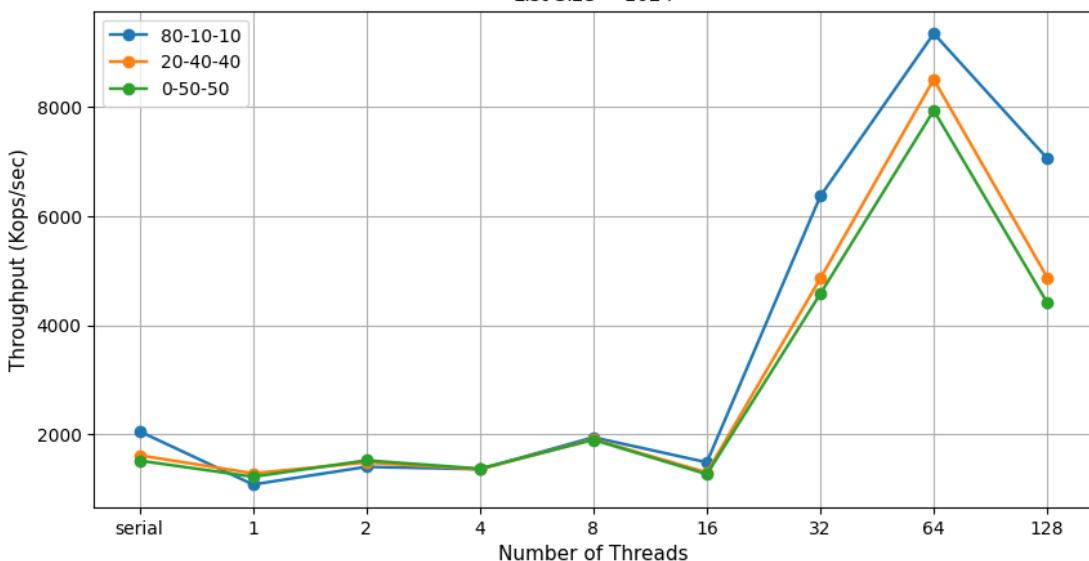


Αρχικά, και για τα δύο διαγράμματα, θα μελετήσουμε το ποσοστό λειτουργιών “100 – 0 – 0”, κατά το οποίο εκτελείται μόνο η συνάρτηση `contains()` στη λίστα μας. Παρατηρούμε ότι, τώρα, το throughput είναι χαμηλότερο απ' ότι στο lazy synchronization. Αυτό συμβαίνει, διότι, στο lazy synchronization, η συνάρτηση `contains()` δε χρησιμοποιούσε κανένα κλείδωμα, ενώ, στο non – blocking synchronization, η `contains()` χρησιμοποιεί atomic operations και μόλις συναντίσει κάποιον κόμβο που διαθέτει το `marked` ίσο με `true`, τότε η `contains()` προσπαθεί να διαγράψει τον κόμβο και φυσικά. Επομένως, η `contains()` του non – blocking synchronization είναι πιο κοστοβόρα από την αντίστοιχη συνάρτηση του lazy synchronization. Ωστόσο, το throughput που λαμβάνουμε για το ποσοστό λειτουργιών “100 – 0 – 0” παραμένει μεγαλύτερο απ' ότι στις υπόλοιπες υλοποιήσεις που είδαμε.

Όπως και προηγουμένως έτσι και τώρα, το throughput του ποσοστού λειτουργιών “100 – 0 – 0” είναι πολύ μεγαλύτερο από το throughput των υπόλοιπων ποσοστών λειτουργιών και αυτό δε μας επιτρέπει να παρατηρήσουμε τι συμβαίνει με τα υπόλοιπα ποσοστά λειτουργιών. Γι' αυτό το λόγο, απομονώνουμε τα υπόλοιπα ποσοστά λειτουργιών σε ξεχωριστά διαγράμματα, όπως φαίνεται παρακάτω:

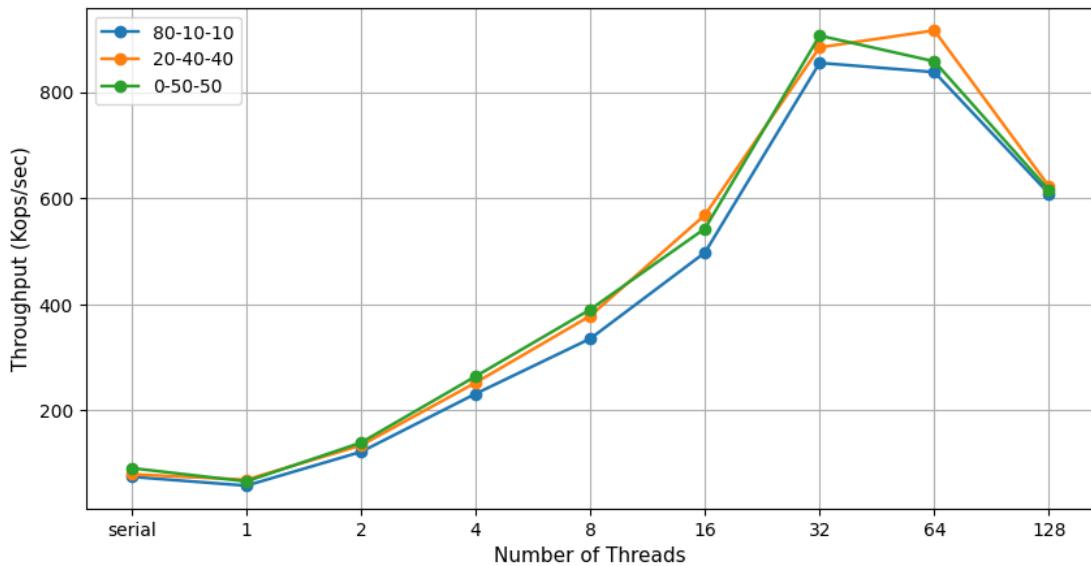
Concurrent List - Non-blocking Synchronisation

List Size = 1024



Concurrent List - Non-blocking Synchronisation

List Size = 8192



Μελετώντας τα δύο διαγράμματα, παρατηρούμε ότι το throughput είναι ελαφρώς μεγαλύτερο από το throughput που λάβαμε στο lazy synchronization. Αυτό συμβαίνει, διότι οι συναρτήσεις `add()` και `remove()`, πλέον, δε χρησιμοποιούν κλειδώματα, αντιθέτως χρησιμοποιούν atomic operations. Παρατηρούμε, και πάλι, ότι στο πρώτο διάγραμμα, το πρόγραμμά μας παρουσιάζει αρκετά scalability breaks, διότι, πιθανώς, παρατηρούνται πολλές αποτυχημένες απόπειρες ατομικών ενημερώσεων που οδηγούν το πρόγραμμά μας σε επαναπροσπάθειες, επιβραδύνοντας το. Αντιθέτως, στο δεύτερο διάγραμμα, όπου χρησιμοποιούμε μεγαλύτερη λίστα, δεν παρατηρείται το φαινόμενο αυτό και το πρόγραμμά μας κλιμακώνει. Σημειώνουμε, επίσης, ότι το throughput στο δεύτερο διάγραμμα είναι πάντα μικρότερο απ' ότι στο πρώτο διάγραμμα, διότι στο δεύτερο διάγραμμα εργαζόμαστε με λίστα μεγαλύτερου μήκους και η διάσχισή της είναι πιο χρονοβόρα.

Άσκηση 3

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Ο στόχος αυτής της άσκησης ήταν η ανάπτυξη, παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου **K-means** μέσω του εργαλείου **CUDA**, ώστε να αξιοποιηθεί η επεξεργαστική ισχύς των καρτών γραφικών NVIDIA.

Naive Version

Στην **naive** έκδοση που υλοποιήθηκε, η GPU ανέλαβε την εκτέλεση του πιο υπολογιστικά απαιτητικού μέρους του αλγορίθμου: τον υπολογισμό των κοντινότερων clusters (nearest clusters) για κάθε σημείο δεδομένων σε κάθε επανάληψη του αλγορίθμου.

Για την υλοποίηση της παράλληλης έκδοσης, ξεκινήσαμε με τον σχεδιασμό και την ανάπτυξη του πυρήνα `find_nearest_cluster`. Ο συγκεκριμένος πυρήνας αναλαμβάνει να υπολογίσει το κοντινότερο cluster για κάθε σημείο δεδομένων και υλοποιήθηκε για εκτέλεση στην GPU. Παράλληλα, αναπτύχθηκαν οι απαραίτητες υπορουτίνες, όπως η `euclid_dist_2` για τον υπολογισμό της ευκλείδειας απόστασης μεταξύ σημείων και η `get_tid`, που εξασφαλίζει την κατάλληλη κατανομή των δεδομένων στα threads της GPU. Οι υπορουτίνες αυτές επέτρεψαν την αποδοτική και σωστή εκτέλεση των υπολογισμών σε πολλαπλά threads ταυτόχρονα.

Η διαχείριση των δεδομένων μεταξύ CPU και GPU ήταν κρίσιμη για την επιτυχία της υλοποίησης. Σε κάθε επανάληψη του αλγορίθμου, πραγματοποιήθηκαν δύο βασικές μεταφορές δεδομένων: από τη CPU προς την GPU (Host to Device) και από την GPU προς την CPU (Device to Host). Η πρώτη μεταφορά περιλάμβανε το dataset και τα clusters, ώστε η GPU να εκτελέσει τους απαιτούμενους υπολογισμούς. Η δεύτερη μεταφορά επέστρεφε τα αποτελέσματα στην CPU, ώστε να ενημερωθούν τα κέντρα των clusters.

Τα σημεία του κώδικα στα οποία έγιναν αλλαγές φαίνονται παρακάτω:

- `__device__ int get_tid()`

```
__device__ int get_tid() {
    return blockDim.x*blockIdx.x+threadIdx.x; /* TODO: Calculate 1-Dim
                                                global ID of a thread */
}
```

- `double euclid_dist_2_transpose()`

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

    /* TODO: Calculate the euclid_dist of elem=objectId of objects from
       elem=clusterId from clusters*/

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

    return (ans);
}

...
```

- `void find_nearest_cluster()`

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

/* TODO: Maybe something is missing here... should all threads run
this? */

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

Σε αυτό το σημείο κάθε thread αναλαμβάνει να επεξεργαστεί το δικό του αντικείμενο (που αντιστοιχεί στο global thread ID) και αρχικά υπολογίζει την απόσταση του αντικειμένου από το πρώτο cluster χρησιμοποιώντας τη συνάρτηση euclid_dist_2. Στη συνέχεια, μέσα από μία επαναληπτική διαδικασία, ελέγχει τις αποστάσεις του αντικειμένου από όλα τα clusters για να βρει τη μικρότερη απόσταση. Όταν εντοπιστεί το cluster με την ελάχιστη απόσταση, ενημερώνεται ο πίνακας deviceMembership. Αν η ανάθεση του αντικειμένου σε cluster αλλάζει (δηλαδή, το αντικείμενο μετακινηθεί σε νέο cluster), αυξάνεται ατομικά η τιμή της μεταβλητής devdelta.

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

/* TODO: call min_dist = euclid_dist_2(...) with correct
   objectId/clusterId */
min_dist =
euclid_dist_2(numCoords,numObjs,numClusters,objects,deviceClusters,tid,0);

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

if (deviceMembership[tid] != index) {
    /* TODO: Maybe something is missing here...is this write safe? */

    atomicAdd(devdelta,1.0);
}

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

Ακολούθως, στην kmeans_gru, πραγματοποιούμε τις αρχικοποιήσεις και τις μεταφορές δεδομένων προς τη μνήμη της GPU.

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

Σε κάθε επανάληψη, πριν καλέσουμε τη συνάρτηση `find_nearest_clusters`, μεταφέρουμε τα ενημερωμένα κέντρα των clusters, που υπολογίστηκαν στην προηγούμενη επανάληψη, στη GPU. Αφού ολοκληρώθει η εκτέλεση της `find_nearest_clusters` στη GPU, επιστρέφουμε στον Host τα δεδομένα του πίνακα `membership` και την τιμή του `delta` για να ελέγχουμε αν έχει επιτευχθεί σύγκλιση.

Τέλος, τα κέντρα των clusters ανανεώνονται και αποφασίζεται αν θα ξεκινήσει η επόμενη επανάληψη ή αν η διαδικασία θα τερματιστεί.

```
/* GPU part: calculate new memberships */

timing_transfers = wtime();
/* TODO: Copy clusters to deviceClusters
checkCuda(cudaMemcpy(...));
checkCuda(cudaMemcpy(deviceClusters, clusters,
numClusters*numCoords*sizeof(double), cudaMemcpyHostToDevice));

transfers_time += wtime() - timing_transfers;

checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));

timing_transfers = wtime();
/* TODO: Copy deviceMembership to membership
checkCuda(cudaMemcpy(...));
checkCuda(cudaMemcpy(membership, deviceMembership,
numObjs*sizeof(int), cudaMemcpyDeviceToHost));

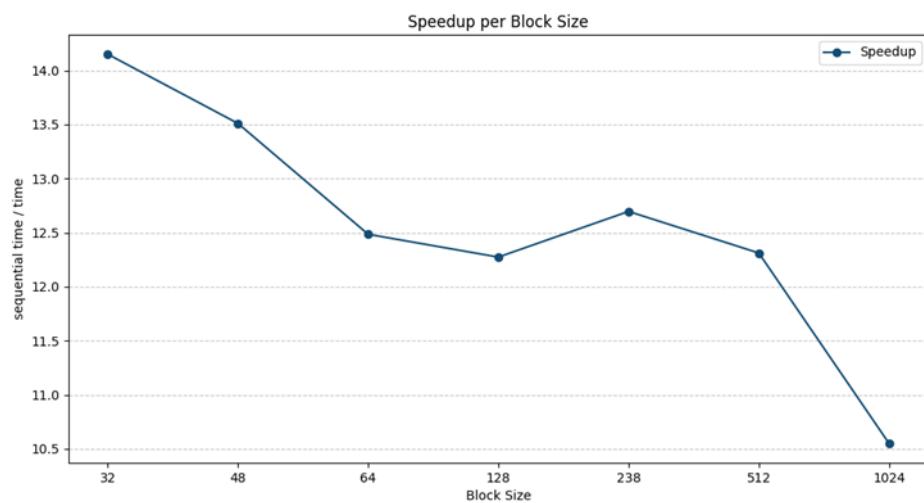
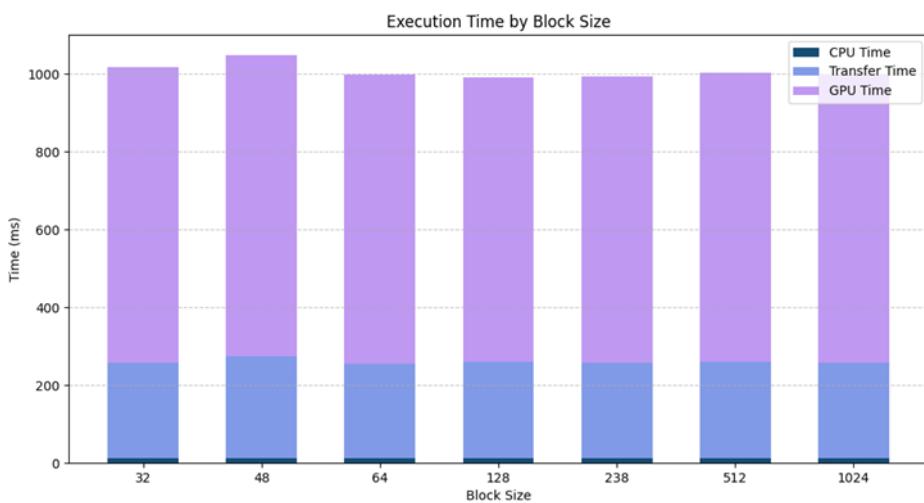
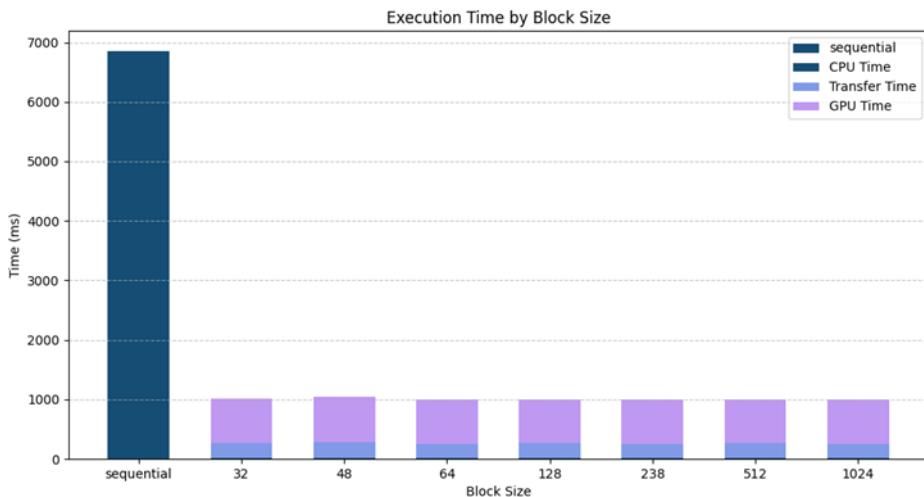
/* TODO: Copy dev_delta_ptr to &delta
checkCuda(cudaMemcpy(...));

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

transfers_time += wtime() - timing_transfers;
...

return;
}
```

1. Για την αξιολόγηση της απόδοσης του αλγορίθμου, πραγματοποιήθηκαν μετρήσεις για τη naive version και τη σειριακή έκδοση χρησιμοποιώντας το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και διάφορες τιμές του μεγέθους των blocks (`block_size` = {32, 48, 64, 128, 238, 512, 1024}). Τα αποτελέσματα που προέκυψαν απεικονίστηκαν στα ακόλουθα τρία διαγράμματα.



Το πρώτο είναι ένα barplot διάγραμμα χρόνου εκτέλεσης, στο οποίο ο άξονας x περιλαμβάνει τη σειριακή έκδοση και τις διαφορετικές τιμές `block_size`, ενώ ο άξονας y παρουσιάζει τον συνολικό χρόνο εκτέλεσης. Οι μπάρες του διαγράμματος χωρίστηκαν σε τρία επίπεδα, που αντιπροσώπευαν τον χρόνο εκτέλεσης στην GPU, τον χρόνο μεταφορών δεδομένων και τον χρόνο εκτέλεσης υπολογισμών στην CPU. Το δεύτερο διάγραμμα είναι το ίδιο με το πρώτο χωρίς όμως την μπάρα για το `sequential` για να απεικονίζονται καλύτερα τα αποτελέσματα.

Το τρίτο διάγραμμα είναι ένα speedup plot, όπου ο άξονας x περιλάμβανε το block_size και ο άξονας y παρουσίαζε τη σχέση seq_time / time, δηλαδή το κέρδος από την παράλληλη υλοποίηση.

2. Η παράλληλη έκδοση του αλγορίθμου **K-means** παρουσίασε σαφή βελτίωση απόδοσης σε σύγκριση με τη σειριακή υλοποίηση, ιδιαίτερα για μεγαλύτερα dataset και αριθμό clusters.

Ο σημαντικότερος παράγοντας που συνεισφέρει στη βελτίωση αυτή είναι η εκτεταμένη παραλληλία που παρέχεται από τη GPU, καθώς η επεξεργασία των δεδομένων και ο υπολογισμός των κοντινότερων clusters εκτελούνται ταυτόχρονα σε πολλαπλά threads.

Ωστόσο, η επίδοση επηρεάζεται από το γεγονός ότι ο αλγόριθμος **K-means** περιλαμβάνει ενδιάμεσες φάσεις που είναι σειριακές ή εκτελούνται καλύτερα στην CPU. Συγκεκριμένα, η ενημέρωση των cluster centers στην CPU, που απαιτεί συγχρονισμό δεδομένων μεταξύ GPU και CPU, περιορίζει το συνολικό speedup. Επιπλέον, η μεταφορά δεδομένων μεταξύ CPU και GPU (Host to Device και Device to Host) εισάγει επιπλέον overhead, γεγονός που μειώνει περαιτέρω την αποτελεσματικότητα

Ο K-means δεν είναι γενικά κατάλληλος για παράλληλη εκτέλεση σε GPUs, διότι περιέχει σημεία που περιορίζουν την πλήρη αξιοποίηση της παραλληλίας. Τα σημεία αυτά, όπως προαναφέρθηκαν, είναι οι σειριακές φάσεις και οι επαναλαμβανόμενες μεταφορές δεδομένων.

3. Η επίδοση του αλγορίθμου δεν φαίνεται να επηρεάζεται ιδιαίτερα από το μέγεθος των blocks (block_size). Αυτό συμβαίνει επειδή ο K-means είναι memory-bound, δηλαδή η απόδοσή του εξαρτάται περισσότερο από τη μεταφορά δεδομένων στη μνήμη παρά από τους υπολογισμούς ανά block.

Transpose Version

Σε αυτήν την εκδοχή, θα τροποποιήσουμε τις δομές δεδομένων objects και clusters. Πιο συγκεκριμένα, θα μετατρέψουμε το indexing των δομών αυτών από row – based σε column – based. Γι' αυτό το λόγο, οι σχετικοί πίνακες πρέπει να γίνουν transpose με τη χρήση βοηθητικών buffers. Ανοίγουμε το αρχείο cuda_kmeans_transpose.cu, που μας έχει δοθεί, και πραγματοποιούμε τις απαραίτητες τροποποιήσεις και προσθήκες, σύμφωνα με τις υποδείξεις των νέων **TODO**. Παρακάτω φαίνονται αναλυτικά τα βήματα που ακολουθούμε:

- `__device__ int get_tid()`

Η συνάρτηση αυτή έχει ακριβώς το ίδιο περιεχόμενο που είχε και στη παίνε εκδοχή. Οπότε, η υλοποίησή της έχει δοθεί στο προηγούμενο τμήμα του κεφαλαίου αυτού.

- `double euclid_dist_2_transpose()`

Η συνάρτηση αυτή υπολογίζει και επιστρέφει την ευκλείδεια απόσταση μεταξύ ενός object και ενός cluster. Πρέπει να λάβουμε υπόψιν μας ότι, πλέον, τα ορίσματα objects και clusters είναι μονοδιάστατοι πίνακες με column – based indexing:

```

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

    /* TODO: 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++) {
        to_square = objects[numObjs*i+objectId]-clusters[numClusters*i+clusterId];
        ans += to_square*to_square;
    }
    return (ans);
}

```

- void find_nearest_cluster()

Δε χρειάζεται να πραγματοποιήσουμε κάποια αλλαγή σε αυτήν τη συνάρτηση. Το σώμα της έχει υλοποιηθεί ήδη στο τμήμα της naive εκδοχής του k – means.

- void kmeans_gpu()

Αρχικά, πρέπει να ορίσουμε και να αρχικοποιήσουμε τους buffers που θα περιέχουν τα objects και τα clusters του προβλήματός μας. Προσέχουμε ώστε οι buffers να έχουν column – based indexing:

```

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

```

Όμως, η συνάρτηση void kmeans_gpu() καλείται με όρισμα τον πίνακα double *objects, ο οποίος είναι row – based indexed. Γι' αυτό το λόγο, πρέπει να αντιγράψουμε τα περιεχόμενα αυτού του πίνακα στους column – based indexed buffers που δημιουργήσαμε πριν λίγο:

```

// TODO: Copy objects given in [numObjs][numCoords] layout to new
// [numCoords][numObjs] layout
for (i=0; i<numObjs; i++) {
    for (j=0; j<numCoords; j++) {
        dimObjects[j][i] = objects[i*numCoords+j];
    }
}

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

```

Υστερα, πρέπει να ορίσουμε το μέγεθος του CUDA grid που θα χρησιμοποιήσουμε. Πιο συγκεκριμένα, όπως είδαμε και στη naive εκδοχή, θέλουμε να δημιουργήσουμε τόσα CUDA threads όσο και το πλήθος των objects, ώστε κάθε CUDA thread να αναλαμβάνει ένα object και να βρίσκει σε ποιο cluster ανήκει αυτό. Με άλλα λόγια, θέλουμε να δημιουργήσουμε CUDA threads με πλήθος τουλάχιστον ίσο με numObjs. Μας έχει δοθεί ήδη το μέγεθος που θα έχει ένα CUDA thread block, όπως φαίνεται και παρακάτω:

```
const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize : numObjs;
```

Άρα, για να έχουμε, τελικά, τόσα CUDA threads όσο και το πλήθος των objects, το μέγεθος του grid πρέπει να είναι ίσο με:

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

ακριβώς όπως το είχαμε υπολογίσει και στη naïve εκδοχή.

Στη συνέχεια, προσθέτουμε τις απαραίτητες μεταφορές δεδομένων από τον host (CPU) στο device (GPU) και αντίστροφα:

```
/* TODO: Copy clusters to deviceClusters */
checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
                      numClusters * numCoords * sizeof(double),
                      cudaMemcpyHostToDevice));

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

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

Η συνάρτηση void kmeans_gpu() καλείται με όρισμα τον πίνακα double *clusters, ο οποίος είναι row – based indexed. Στο τέλος της συνάρτησης, ο πίνακας αυτός πρέπει να περιέχει τα τελικά – οριστικά κέντρα των clusters. Ωστόσο, εμείς, κατά την εκτέλεση του κώδικα, αποθηκεύσαμε τα κέντρα των clusters στο δισδιάστατο πίνακα dimClusters, ο οποίος είναι column – based indexed. Άρα, τώρα, πρέπει να αντιγράψουμε τα περιεχόμενα του πίνακα dimClusters στον πίνακα clusters, διατρέχοντας προσεκτικά τους πίνακες και χρησιμοποιώντας το κατάλληλο indexing:

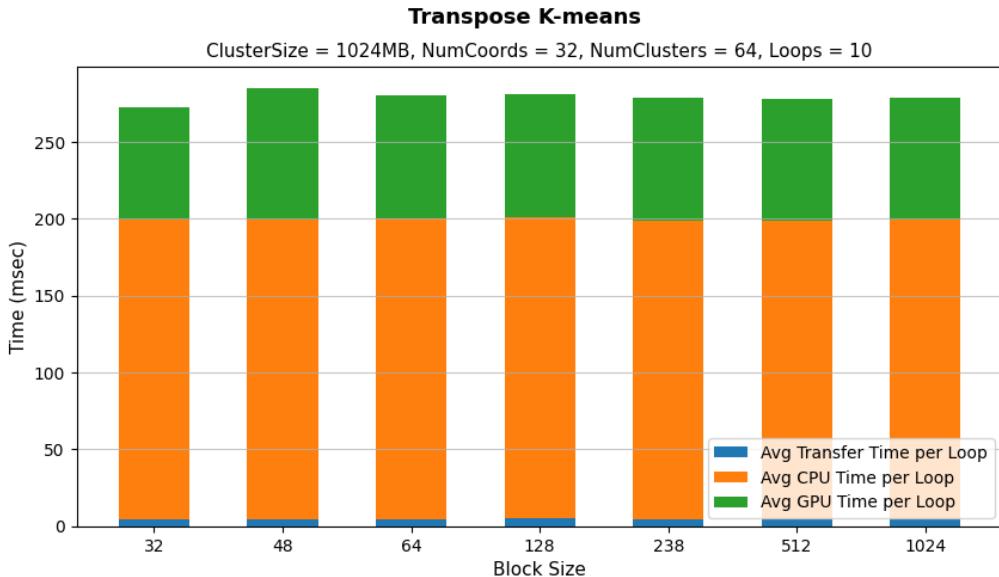
```
/*TODO: Update clusters using dimClusters. Be carefull of layout!!!
   clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */
for (i = 0; i < numClusters; i++) {
    for (j = 0; j < numCoords; j++) {
        clusters[i*numCoords + j] = dimClusters[j][i];
    }
}
```

Έτσι, έχουμε ολοκληρώσει τον κώδικα του transpose version του k – means και είμαστε έτοιμοι να τον μεταγλωττίσουμε. Αρχικά, μεταγλωττίζουμε τον κώδικα χρησιμοποιώντας το flag -DVALIDATE, ώστε να ελέγχουμε την ορθότητα του. Τρέχουμε το παραγόμενο εκτελέσιμο και διαπιστώνουμε ότι ο κώδικας μας είναι ορθός. Έπειτα, μεταγλωττίζουμε ξανά τον κώδικα μας, αλλά αυτήν τη φορά δε χρησιμοποιούμε το flag -DVALIDATE. Θα λάβουμε μετρήσεις σχετικά με την επίδοση του transpose version για διάφορα block sizes = {32, 48, 64, 128, 238, 512, 1024} και για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}, όπως κάναμε και προηγουμένως με το naïve version.

Καθώς χρησιμοποιούμε 10 loops, ο αλγόριθμος k – means θα εκτελεστεί αρκετές φορές, το πολύ 10. Για κάθε εκτέλεση – loop του k – means, διαθέτουμε μετρήσεις που μας δίνουν:

- το μέσο χρόνο μεταφοράς δεδομένων από το host (CPU) στο device (GPU) και αντίστροφα,
- το μέσο χρόνο εκτέλεσης στον επεξεργαστή (CPU) και
- το μέσο χρόνο εκτέλεσης στην κάρτα γραφικών (GPU).

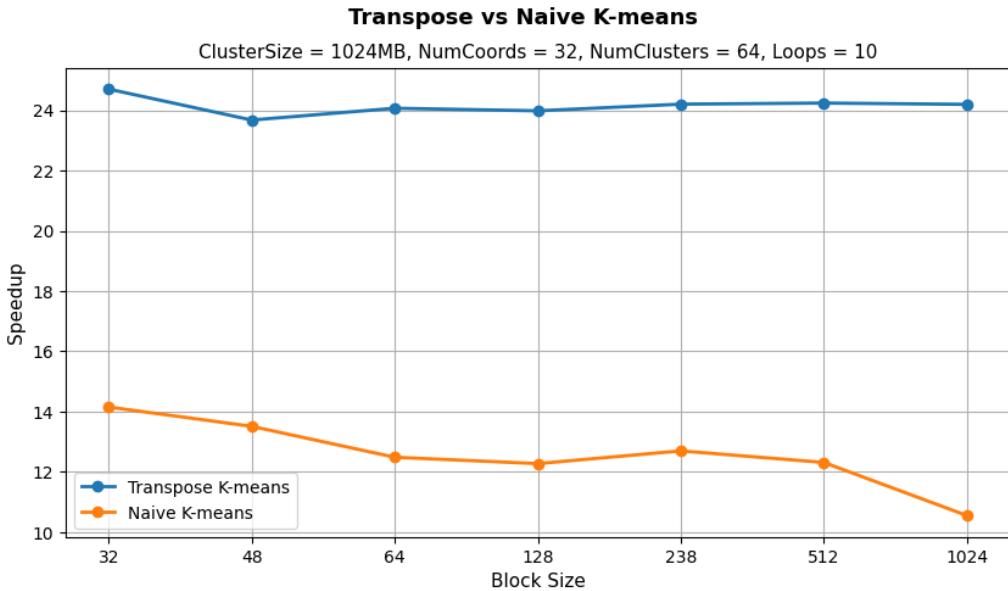
Απεικονίζουμε όλες αυτές τις μετρήσεις συναρτήσει του block size, στο stacked bar plot που φαίνεται και παρακάτω:



Αρχικά, παρατηρούμε ότι, ανεξαρτήτως του block size, ο μέσος χρόνος μεταφοράς δεδομένων και ο μέσος χρόνος εκτέλεσης στον επεξεργαστή παραμένουν σταθεροί και αυτό είναι απολύτως λογικό. Ο χρόνος μεταφοράς δεδομένων παραμένει σταθερός, διότι, ανεξαρτήτως του block size, το μέγεθος των δεδομένων παραμένει σταθερό και το δίκτυο που συνδέει τον επεξεργαστή με την κάρτα γραφικών παραμένει αμετάβλητο. Ο χρόνος εκτέλεσης στον επεξεργαστή παραμένει, επίσης, σταθερός, διότι ο επεξεργαστής εκτελεί έναν σειριακό κώδικα που διαχειρίζεται πάντα συγκεκριμένο όγκο δεδομένων και εκτελεί πάντα συγκεκριμένο αριθμό πράξεων – operations.

Όσον αφορά το μέσο χρόνο εκτέλεσης στην κάρτα γραφικών, παρατηρούμε ότι και αυτός είναι σχετικά σταθερός και γίνεται μέγιστος για block size = 48. Βλέπουμε, λοιπόν, και, πάλι, ότι το block size δεν επηρεάζει το χρόνο εκτέλεσης στην GPU. Η αύξηση του block size, η οποία συνεπάγεται αύξηση του occupancy των SMs της GPU, δεν επιταχύνει το πρόγραμμά μας. Αυτό συμβαίνει, διότι ο αλγόριθμος k – means είναι memory – bound. Δηλαδή, απαιτεί πολλές προσβάσεις στη μνήμη, ενώ για κάθε δεδομένο που ανασύρεται από τη μνήμη, δεν πραγματοποιούνται πολλές πράξεις – operations, με αποτέλεσμα να μην κρύβονται τα memory latencies.

Σημειώνουμε ότι ο μέσος χρόνος εκτέλεσης ανά loop του σειριακού k – means είναι περίπου ίσος με 6.85 sec, πολύ μεγαλύτερος από το χειρότερο χρόνο εκτέλεσης που εμφανίζεται στο παραπάνω διάγραμμα. Γι' αυτό το λόγο, επιλέξαμε να μην απεικονίσουμε το χρόνο εκτέλεσης του σειριακού αλγορίθμου. Αν απεικονίζαμε στο ίδιο διάγραμμα και το χρόνο εκτέλεσης του σειριακού προγράμματος, τότε δε θα μπορούσαμε να διακρίνουμε τις μικροδιαφορές που παρουσιάζονται στις μετρήσεις μας για τα διαφορετικά block sizes. Η σύγκριση της επίδοσης του παράλληλου κώδικα με την επίδοση του σειριακού μπορεί να γίνει και μέσα από το speedup plot (seq_time/parallel_time). Παρακάτω φαίνονται το speedup της transpose εκδοχής, αλλά και της naive εκδοχής σε ένα κοινό διάγραμμα.



Αρχικά, παρατηρούμε ότι το transpose version είναι περίπου 24 φορές ταχύτερο από τον σειριακό κώδικα του k – means. Παρατηρούμε, επίσης, ότι το speedup παραμένει σχετικά σταθερό, ανεξαρτήτως του block size και το πρόγραμμά μας δεν κλιμακώνει περαιτέρω. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος k – means είναι memory – bound, όπως εξηγήσαμε και νωρίτερα.

Παρατηρούμε, επίσης, ότι το transpose version είναι αρκετά ταχύτερο από το naive version. Η μετατροπή των δομών δεδομένων από row – based indexed σε column – based indexed, τελικά, ήταν σημαντική. Μετά από αυτήν τη μετατροπή, ο πίνακας objects περιέχει στην i – οστή γραμμή του την i – οστή συντεταγμένη κάθε αντικειμένου. Και, άρα, τα αντικείμενα τοποθετούνται σε συνεχόμενες θέσεις της καθολικής μνήμης της GPU βάσει αυτού του σχηματισμού. Αυτό είναι ιδιαίτερα βοηθητικό, διότι κάθε νήμα αναλαμβάνει ένα object και, άρα, συχνά, κοντινά νήματα θα διαβάζουν την ίδια συντεταγμένη, αλλά για διαφορετικά objects. Όπως γνωρίζουμε και από τη θεωρία, η καθολική μνήμη της GPU χωρίζεται σε segments. Μόλις κάποιο νήμα διαβάσει ένα word αυτού του segment, τότε το segment αυτό γίνεται διαθέσιμο στο half – warp του νήματος για την ίδια εντολή. Επομένως, όταν κάποιο νήμα διαβάζει την i – οστή συντεταγμένη ενός object, τότε τα νήματα του half – warp θα λαμβάνουν την i – οστή συντεταγμένη των δικών τους αντικειμένων, χωρίς να χρειαστεί να προσπελάσουν τη μνήμη. Και έτσι, μειώνονται οι καθυστερήσεις λόγω προσβάσεων στη μνήμη, με αποτέλεσμα το transpose version είναι ταχύτερο από το naive version.

Shared Version

Σε αυτήν τη νέα εκδοχή του αλγορίθμου k – means, θέλουμε κάθε CUDA thread block να διαθέτει ένα αντίγραφο του πίνακα clusters σε τοπική μνήμη, ορατή μονάχα σε όλα τα νήματα του συγκεκριμένου CUDA thread block. Ανοίγουμε, λοιπόν, το αρχείο `cuda_kmeans_shared.cu` και πραγματοποιούμε τις απαραίτητες τροποποιήσεις και προσθήκες, σύμφωνα με τις υποδείξεις των νέων **TODO**. Παρακάτω φαίνονται αναλυτικά τα βήματα που ακολουθούμε:

- `double euclid_dist_2_transpose()`

Δε χρειάζεται να πραγματοποιήσουμε κάποια τροποποίηση σε αυτήν τη συνάρτηση. Αντιγράφουμε το σώμα της από την αντίστοιχη συνάρτηση του transpose version.

- `void find_nearest_cluster()`

Σε αυτήν τη συνάρτηση, πρέπει, τώρα, για κάθε CUDA thread block, να πραγματοποιείται η αντιγραφή του πίνακα clusters σε τοπική διαχειριζόμενη μνήμη. Παρακάτω φαίνεται ο τροποποιημένος κώδικας της συνάρτησης για αυτό το ερώτημα:

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

    /* TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
       BEWARE: Make sure operations is complete before any thread continues... */
    int i, j;
    int local_tid = threadIdx.x;

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

    __syncthreads();

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

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

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

    for (i = 1; i < numClusters; i++) {
        /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId using clusters in shmem*/
        dist =
            euclid_dist_2_transpose(numCoords,numObjs,numClusters,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) {
        /* TODO: Maybe something is missing here... is this write safe? */
        atomicAdd(devdelta, 1.0);
    }

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

Όπως βλέπουμε, για κάθε block, ορίζουμε μία νέα μεταβλητή, τον double πίνακα shmemClusters. Σε αυτόν τον πίνακα, θέλουμε να αντιγράψουμε τον πίνακα deviceClusters. Όπως έχουμε ήδη εξηγήσει, ο πίνακας shmemClusters αποθηκεύεται σε μνήμη κοντά στο CUDA thread block και όχι στην καθολική μνήμη της GPU. Κατά συνέπεια, το περιεχόμενο του πίνακα αυτού είναι ορατό μόνο από τα νήματα του block. Επομένως, σε ένα block, πρέπει τα νήματα αυτού του block να πραγματοποιήσουν την αντιγραφή. Γι' αυτό το λόγο, στο nested for loop, όπου πραγματοποιείται η αντιγραφή, τα νήματα εισέρχονται βάσει του local id που διαθέτουν εντός του block τους και όχι του global id που διαθέτουν

εντός του grid. Σε ένα block, κάθε νήμα με id local_tid, αναλαμβάνει να αντιγράψει τις στήλες και, μέχρι να διασχίσει όλες τις στήλες του πίνακα deviceClusters. Με αυτόν τον τρόπο, βεβαιωνόμαστε ότι, σε κάθε block, αρχικοποιείται επιτυχώς ο πίνακας shmemClusters, ενώ αποφεύγουμε και την εμφάνιση race conditions.

Κάθε νήμα που ολοκληρώνει την αντιγραφή των στηλών που ανέλαβε, δεν πρέπει να προχωρήσει απευθείας σε υπολογισμούς. Αντιθέτως, πρέπει να περιμένει να ολοκληρωθεί η διαδικασία της αντιγραφής από όλα τα νήματα, ώστε, όταν, αργότερα, θα ξεκινήσει τους υπολογισμούς, να διαβάζει τα σωστά δεδομένα που χρειάζεται. Εισάγοντας την εντολή `_syncthreads();` αμέσως μετά το nested for loop, βεβαιωνόμαστε ότι όλα τα νήματα περιμένουν να ολοκληρωθεί η διαδικασία της αντιγραφής και, έπειτα, προχωρούν σε υπολογισμούς.

Το σώμα του `if (tid < numObjs) {}` παραμένει ίδιο με αυτό που είχαμε δει στο naive version, μόνο που όταν καλούμε την `euclid_dist_2_transpose()`, περνάμε, πλέον, σαν όρισμα τον πίνακα `shmemClusters` και όχι τον πίνακα `deviceClusters`.

- `void kmeans_gpu()`

Όπως και στο transpose version, ορίζουμε τους buffers, όπου θα αποθηκεύσουμε τα `objects` και `clusters` με column – based indexing:

```
/* TODO: Copy me from transpose version*/
//calloc_2d(... -> [numCoords][numObjs]
double **dimObjects = (double**) calloc_2d(numCoords, numObjs, sizeof(double));
//calloc_2d(... -> [numCoords][numClusters]
double **dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
//calloc_2d(... -> [numCoords][numClusters]
double **newClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
```

Έπειτα, αντιγράφουμε στον buffer `dimObjects` τα περιεχόμενα του πίνακα `objects`, όπως ακριβώς είχαμε κάνει και στο transpose version:

```
/* TODO: Copy me from transpose version*/
for (i = 0; i < numObjs; i++) {
    for(j = 0; j < numCoords; j++) {
        dimObjects[j][i] = objects[i*numCoords+j];
    }
}
```

Ύστερα, πρέπει να ορίσουμε το μέγεθος του CUDA grid που θα χρησιμοποιήσουμε. Για τους λόγους που έχουμε αναφέρει τόσο στο naive version όσο και στο transpose version, το μέγεθος του grid είναι το ακόλουθο:

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

Τώρα, πρέπει να ορίσουμε το μέγεθος της διαμοιραζόμενης μνήμης κάθε block. Όπως είδαμε και παραπάνω, κάθε block διαθέτει στην τοπική του μνήμη ένα αντίγραφο του πίνακα `clusters`. Άρα, για τη διαμοιραζόμενη μνήμη κάθε block, αρκεί να δεσμεύσουμε όσα bytes καταλαμβάνει ο πίνακας `clusters`:

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

Στη συνέχεια, προσθέτουμε τις απαραίτητες μεταφορές δεδομένων από τον host (CPU) στο device (GPU) και αντίστροφα:

```
/* TODO: Copy clusters to deviceClusters */
checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
                     numClusters * numCoords * sizeof(double),
                     cudaMemcpyHostToDevice));

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

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

Τέλος, όπως είδαμε αναλυτικότερα στο transpose version, πρέπει να αντιγράψουμε τα στοιχεία του buffer `dimClusters` στον πίνακα `clusters`:

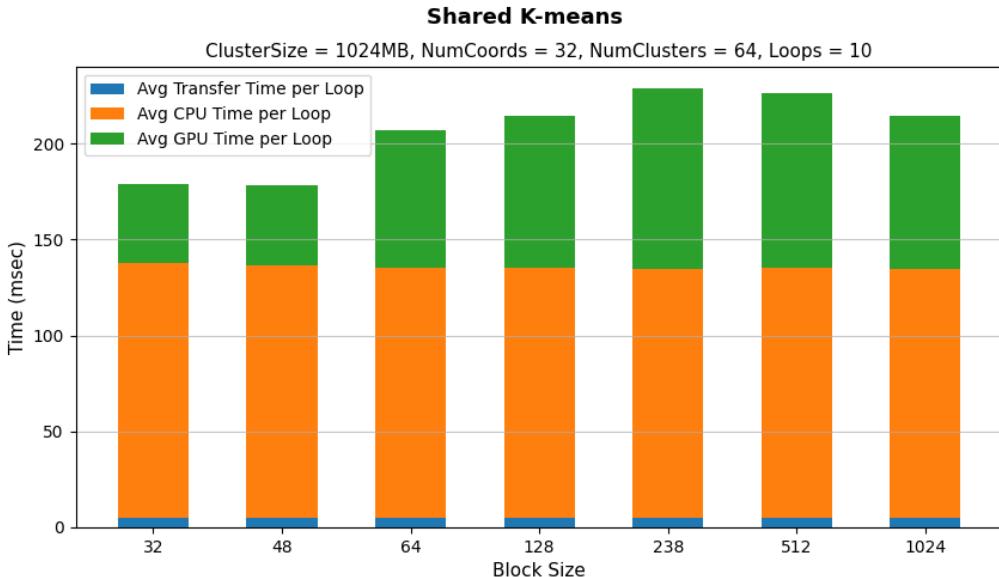
```
/*TODO: Update clusters using dimClusters. Be carefull of layout!!!
   clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */
for (i = 0; i < numClusters; i++) {
    for (j = 0; j < numCoords; j++) {
        clusters[i*numCoords + j] = dimClusters[j][i];
    }
}
```

Οπότε, ολοκληρώσαμε τις τροποποιήσεις. Επόμενο βήμα είναι να ελέγξουμε την ορθότητα του κώδικα που γράψαμε. Γι' αυτό το σκοπό, μεταγλωττίζουμε τον κώδικα χρησιμοποιώντας το flag -DVALIDATE και εκτελούμε το παραγόμενο εκτελέσιμο. Διαπιστώνουμε ότι ο κώδικας μας είναι ορθός. Επομένως, μεταγλωττίζουμε ξανά τον κώδικα μας, αλλά χωρίς τη χρήση του flag -DVALIDATE. Θα λάβουμε μετρήσεις σχετικά με την επίδοση του transpose version για διάφορα block sizes = {32, 48, 64, 128, 238, 512, 1024} και για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}, όπως κάναμε και προηγουμένως με το transpose version.

Καθώς χρησιμοποιούμε 10 loops, ο αλγόριθμος k – means θα εκτελεστεί αρκετές φορές, το πολύ 10. Για κάθε εκτέλεση – loop του k – means, διαθέτουμε μετρήσεις που μας δίνουν:

- το μέσο χρόνο μεταφοράς δεδομένων από το host (CPU) στο device (GPU) και αντίστροφα,
- το μέσο χρόνο εκτέλεσης στον επεξεργαστή (CPU) και
- το μέσο χρόνο εκτέλεσης στην κάρτα γραφικών (GPU).

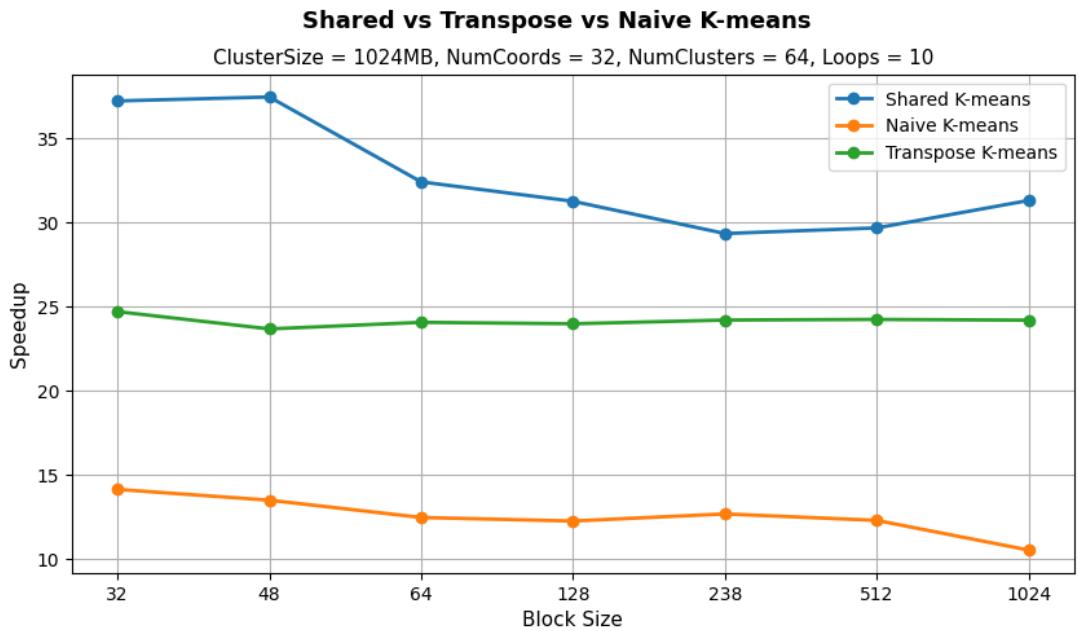
Απεικονίζουμε όλες αυτές τις μετρήσεις συναρτήσει του block size, στο stacked bar plot που φαίνεται και παρακάτω:



Αρχικά, παρατηρούμε και, πάλι, ότι ο χρόνος μεταφοράς δεδομένων και ο χρόνος εκτέλεσης στον επεξεργαστή παραμένουν σταθεροί και ανεξάρτητοι του block size, για τους λόγους που εξηγήσαμε και νωρίτερα, στα προηγούμενα versions.

Η μέτρηση που μεταβάλλεται συναρτήσει του block size είναι ο χρόνος εκτέλεσης στην κάρτα γραφικών. Παρατηρούμε ότι, για μικρό μέγεθος block, σημειώνονται χαμηλότεροι χρόνοι εκτέλεσης στη GPU. Καθώς αυξάνεται το block size, και, άρα, και το occupancy των SMs της GPU, βλέπουμε ότι ο χρόνος εκτέλεσης στη GPU αυξάνεται. Πιο συγκεκριμένα, όπως έχουμε ήδη εξηγήσει, στο shared version, κάθε block διαθέτει σε τοπική μνήμη ένα αντίγραφο του πίνακα clusters. Το αντίγραφο αυτό είναι ορατό μονάχα στα νήματα του block. Κατά συνέπεια, κάθε φορά που ένα νήμα προσπελάζει ένα cluster, δε χρειάζεται να καταφεύγει στην καθολική μνήμη της GPU. Αντιθέτως, καταφεύγει στην τοπική μνήμη του block, η οποία βρίσκεται πιο κοντά από ότι η καθολική μνήμη. Έτσι, μειώνονται οι χρόνοι πρόσβασης στη μνήμη και, άρα, βελτιώνεται η επίδοση του προγράμματος. Ωστόσο, όταν αυξάνεται το μέγεθος του block, παρατηρούμε ότι ελαττώνεται η επίδοση του shared version. Αυτό συμβαίνει, διότι όταν αυξάνεται το μέγεθος του block, αυξάνεται και το πλήθος των νημάτων εντός του block. Κατά συνέπεια, θα υπάρχουν περισσότερα νήματα, τα οποία θα προσπελάζουν διαφορετικά words στην τοπική διαμοιραζόμενη μνήμη του block. Επομένως, θα παρουσιάζονται πολλά αιτήματα για προσπέλαση διαφορετικών θέσεων μνήμης, ενώ το bandwidth της τοπικής μνήμης του block δε θα μπορεί να υποστηρίξει την παράλληλη εξυπηρέτηση όλων αυτών των αιτημάτων. Συνεπώς, θα παρουσιάζονται πολλά bank conflicts, με αποτέλεσμα, τελικά, οι χρόνοι πρόσβασης στη μνήμη να αυξάνονται, παρά να μειώνονται.

Στη συνέχεια, σχεδιάζουμε και το speedup διάγραμμα του shared version. Στο ίδιο διάγραμμα, προσθέτουμε και τα speedup διαγράμματα του naive και transpose version, ώστε να διαπιστώσουμε αν η επίδοση του shared είναι, πράγματι, καλύτερη από την επίδοση των δύο προηγούμενων versions.

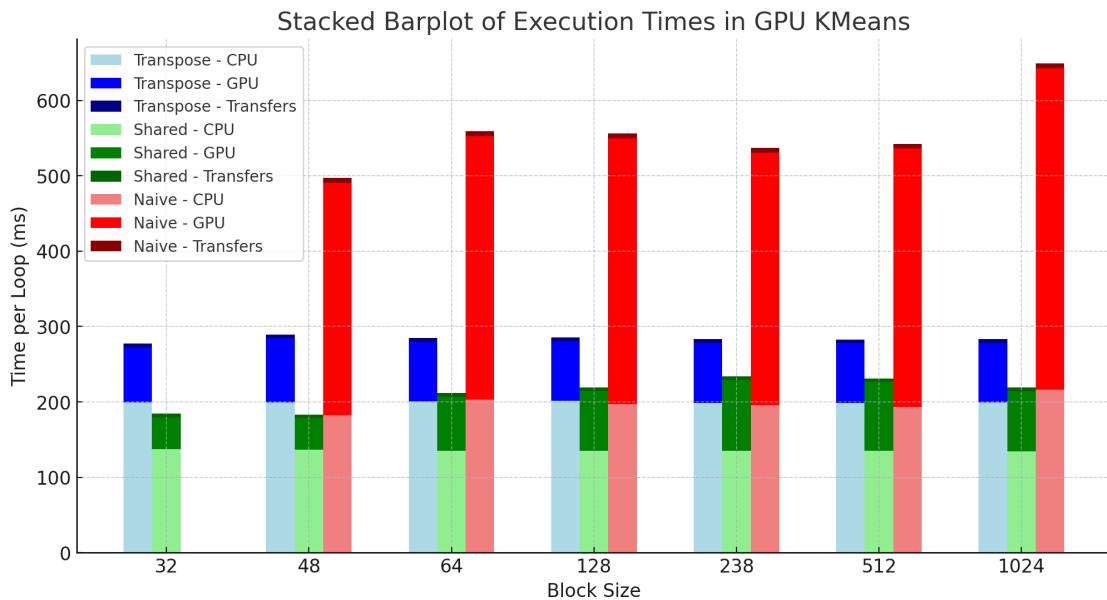


Αρχικά, βλέπουμε ότι το shared version είναι τουλάχιστον 30 φορές ταχύτερο του σειριακού k – means. Μάλιστα, στην καλύτερη μέτρηση που λάβαμε, ο shared k – means είναι 37 φορές ταχύτερος από τον σειριακό k – means. Βλέπουμε και, πάλι, ότι η αύξηση του block size, αν και συνεπάγεται αύξηση του occupancy των SMs της GPU, δεν προκαλεί μείωση στο χρόνο εκτέλεσης του προγράμματος μας. Άρα, το πρόγραμμά μας δεν κλιμακώνει.

Παρατηρούμε, επίσης, ότι το shared version εκτελείται πάντα σε λιγότερο χρόνο απ' ότι το naive και το transpose version. Άρα, τελικά, ακόμη και εάν για μεγάλα μεγέθη blocks, εμφανίζονται bank conflicts που επιβραδύνουν το shared version, αυτό σημειώνει, παρ' όλα αυτά, καλύτερη επίδοση από το transpose και το naive version.

Σύγκριση υλοποίησεων / bottleneck Analysis

1.



Με βάση το διάγραμμα που δείχνει τους χρόνους εκτέλεσης ανά iteration (CPU, GPU, μεταφορές), μπορούμε να αξιολογήσουμε τι περιορίζει τη συνολική απόδοση κάθε υλοποίησης.

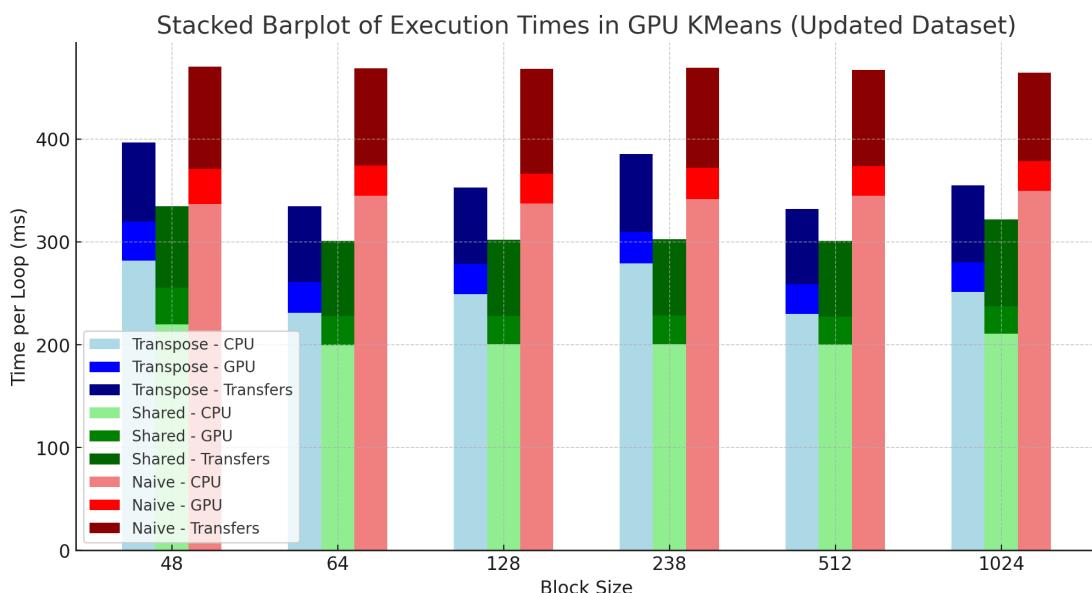
Η Naive GPU είναι η λιγότερο αποδοτική προσέγγιση, όπως φαίνεται από το γεγονός ότι παρουσιάζει σημαντικά μεγαλύτερο GPU execution time σε σχέση με τις άλλες δύο υλοποιήσεις. Ο κύριος λόγος για αυτό είναι η μη αποδοτική χρήση της μνήμης GPU, όπου πιθανώς γίνεται υπερβολική πρόσβαση στη global memory, οδηγώντας σε μεγάλες καθυστερήσεις λόγω memory latencies. Παράλληλα, η CPU συμμετέχει σημαντικά στον συνολικό χρόνο εκτέλεσης, γεγονός που δείχνει ότι υπάρχει περιττό synchronization μεταξύ CPU και GPU. Οι μεταφορές δεδομένων (CPU-GPU transfers) είναι μεγαλύτερες από τις άλλες υλοποιήσεις, κάτι που ενισχύει την υπόθεση ότι υπάρχει αυξημένη επικοινωνία μεταξύ CPU και GPU, περιορίζοντας την απόδοση. Συνολικά, η Naive GPU KMeans είναι GPU-bound, με την υψηλή χρήση της GPU να αποτελεί τον βασικό περιοριστικό παράγοντα.

Η Transpose GPU βελτιώνει αισθητά την εκτέλεση σε σχέση με τη Naive GPU KMeans, αλλά εξακολουθεί να έχει κάποια overheads. Στο γράφημα φαίνεται ότι ο GPU χρόνος έχει μειωθεί, κάτι που υποδηλώνει καλύτερη χρήση των memory access patterns, πιθανώς μέσω βελτιωμένων coalesced memory accesses ή αποφυγής περιττών global memory προσβάσεων. Παρ' όλα αυτά, η CPU εξακολουθεί να είναι σημαντικό bottleneck, κάτι που φαίνεται από την αυξημένη CPU συνεισφορά στον συνολικό χρόνο εκτέλεσης. Αυτό σημαίνει ότι υπάρχει υψηλό synchronization cost μεταξύ CPU και GPU, το οποίο θα μπορούσε να μειωθεί μέσω περαιτέρω optimization. Οι μεταφορές δεδομένων είναι ελαφρώς μειωμένες, γεγονός που δείχνει ότι η Transpose υλοποίηση εκμεταλλεύεται καλύτερα τη μνήμη και τις

μεταφορές δεδομένων από/προς τη GPU. Παρόλα αυτά, η συνολική απόδοση εξακολουθεί να επηρεάζεται από το CPU bottleneck.

Η Shared GPU είναι η πιο αποδοτική προσέγγιση, όπως φαίνεται και από το γράφημα, όπου τόσο ο GPU χρόνος όσο και ο CPU χρόνος είναι οι χαμηλότεροι από όλες τις υλοποιήσεις. Αυτό σημαίνει ότι η υλοποίηση αξιοποιεί βελτιστοποιημένη χρήση shared memory, αποφεύγοντας επαναλαμβανόμενες προσβάσεις στη global memory, μειώνοντας έτσι το latency. Επιπλέον, η CPU overhead είναι χαμηλή, κάτι που σημαίνει ότι υπάρχει καλύτερη ισορροπία φόρτου μεταξύ CPU και GPU. Οι μεταφορές δεδομένων είναι επίσης χαμηλές και σταθερές, γεγονός που δείχνει ότι η υλοποίηση δεν απαιτεί πολλές επιπλέον επικοινωνίες μεταξύ CPU και GPU. Το bottleneck σε αυτή την προσέγγιση φαίνεται να έχει ελαχιστοποιηθεί, με τον μόνο περιοριστικό παράγοντα να είναι η επεξεργαστική ισχύς της GPU.

2.



Η γενική κατανομή των χρόνων εκτέλεσης παραμένει σταθερή. Η Naive GPU εξακολουθεί να είναι η λιγότερο αποδοτική επιλογή, καθώς παρουσιάζει τον υψηλότερο CPU time και Transfer time, υποδηλώνοντας ότι η CPU εκτελεί μεγάλο μέρος του φόρτου εργασίας και υπάρχει αυξημένη επικοινωνία CPU-GPU. Η Transpose GPU μειώνει μερικώς το CPU overhead, αλλά εξακολουθεί να μην αξιοποιεί πλήρως τη GPU. Η Shared GPU παραμένει η πιο αποδοτική υλοποίηση, καθώς παρουσιάζει τον χαμηλότερο συνολικό χρόνο εκτέλεσης.

Όσον αφορά τις μεταβολές στην επίδοση ανά block size, παρατηρούμε ότι για μικρότερα block sizes (32-128), η Naive GPU είναι εξαιρετικά αργή, λόγω της αυξημένης επιβάρυνσης της CPU. Καθώς αυξάνεται το block size (238-1024), η συνολική επίδοση βελτιώνεται σταδιακά και για τις τρεις υλοποιήσεις. Η Shared GPU παρουσιάζει τη μικρότερη διακύμανση μεταξύ των διαφορετικών block sizes, γεγονός που δείχνει ότι η shared memory αξιοποιείται αποτελεσματικά, ανεξάρτητα από το μέγεθος του block size.

Ένα ακόμη σημαντικό στοιχείο είναι ότι η Shared GPU δεν παρουσιάζει μεγάλη ευαισθησία στις αλλαγές του block size. Οι χρόνοι GPU και CPU είναι αρκετά σταθεροί και η κύρια

διακύμανση προέρχεται από τον χρόνο μεταφοράς δεδομένων (CPU-GPU transfers), ο οποίος παραμένει μικρός και δεν επηρεάζει σημαντικά την συνολική απόδοση. Αυτό υποδηλώνει ότι η διαχείριση της μνήμης μέσω shared memory γίνεται αποδοτικά, χωρίς μεγάλες καθυστερήσεις στις memory accesses.

Η Shared GPU αποδεικνύεται η πιο αποδοτική επιλογή, καθώς αξιοποιεί βελτιστοποιημένη χρήση της GPU, μειώνοντας τον συνολικό χρόνο εκτέλεσης σε σχέση με άλλες υλοποιήσεις. Επιπλέον, παρουσιάζει σταθερή απόδοση ανεξάρτητα από το block size, γεγονός που υποδηλώνει ότι η μνήμη GPU χρησιμοποιείται αποδοτικά. Ένα ακόμη σημαντικό πλεονέκτημα είναι το χαμηλό synchronization overhead μεταξύ CPU και GPU, χάρη στη βελτιστοποιημένη χρήση της shared memory.

Ωστόσο, η καταλληλότητα της υλοποίησης για arbitrary configurations εξαρτάται από διάφορους παράγοντες. Σε datasets με πολύ υψηλή διάσταση (coords >> 2), η shared memory μπορεί να μην επαρκεί για την αποθήκευση όλων των δεδομένων, με αποτέλεσμα την αυξημένη χρήση της global memory και τη μείωση της απόδοσης. Επίσης, όταν ο αριθμός των clusters (numClusters) είναι μεγάλος, η διαχείριση της μνήμης γίνεται πιο περίπλοκη, καθώς αυξάνεται η ανάγκη για περισσότερο shared memory space. Τέλος, η κατανομή του workload μεταξύ CPU και GPU μπορεί να χρειαστεί προσαρμογές, καθώς η παρούσα υλοποίηση είναι βέλτιστη για συγκεκριμένες κατανομές δεδομένων και block sizes.

Full-Offload (All-GPU) version

Οι συναρτήσεις int get_tid() και double euclid_dist_2_transpose() παραμένουν ίδιες με τα προηγούμενα versions.

- void find_nearest_cluster():

Φορτώνουμε τα κεντροειδή στην shared memory. Για κάθε αντικείμενο, βρίσκουμε το cluster με τη μικρότερη ευκλείδεια απόσταση. Κρατάμε πόσα αντικείμενα μετακινήθηκαν (atomicAdd στη devdelta). Τέλος, υπολογίζουμε αθροίσματα για κάθε διάσταση/cluster (με atomicAdd στο devicenewClusters και στο devicenewClusterSize).

```

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

    /* TODO: copy me from shared version... */
    int i,j ;
    int local_tid = threadIdx.x;
    for(i = local_tid; i < numClusters; i += blockDim.x) {
        for(j = 0; j < numCoords; j++)
            shmemClusters[j*numClusters + i] = deviceClusters[j*numClusters + i];
    }

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

    // initialize the two added parameteres ...
    for (i = tid; i < numCoords * numClusters; i += gridDim.x * blockDim.x) {
        devicenewClusters[i] = 0.0;
    }

    for (i = tid; i < numClusters; i += gridDim.x * blockDim.x) {
        devicenewClusterSize[i] = 0;
    }

    __syncthreads();
}

```

```

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

    /* TODO: additional steps for calculating new centroids in GPU? */

    int index, i;
    double dist, min_dist;

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

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

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

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

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

    // we have to update the new cluster centers: sum of objects located within ...
    atomicAdd(&devicenewClusterSize[index],1);
    for(j=0;j<numCoords;j++){
        atomicAdd(&devicenewClusters[numClusters*j + index],deviceObjects[numObjs*j + tid]);
    }
}
}

```

- void update_centroids():

Συγκεκριμένα, πρώτα βρίσκουμε σε ποιο cluster αντιστοιχεί το tid με την πράξη tid % numClusters. Πλαιρούμε το μέγεθος αυτού του cluster από τον πίνακα devicenewClusterSize[cluster]. Εφόσον το μέγεθος (clusterSize) είναι μεγαλύτερο

από 0, διαιρούμε το άθροισμα που είχαμε μαζέψει (devicenewClusters[tid]) δια του πλήθους και αποθηκεύουμε το αποτέλεσμα στο deviceClusters[tid].

```
__global__ static
void update_centroids(int numCoords,
                      int numClusters,
                      int *devicenewClusterSize,           // [numClusters]
                      double *devicenewClusters,          // [numCoords][numClusters]
                      double *deviceClusters)           // [numCoords][numClusters])
{

    /* TODO: additional steps for calculating new centroids in GPU? */
    int tid = get_tid();
    int cluster = tid%numClusters;
    int clusterSize;

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

- void kmeans_gpu():

Αρχικά, μετατρέπουμε τα δεδομένα των αντικειμένων από τη μορφή [numObjs][numCoords] στη μορφή [numCoords][numObjs], ώστε να αξιοποιηθεί καλύτερα η τοπικότητα μνήμης στον υπολογισμό των αποστάσεων (συνάρτηση euclid_dist_2_transpose). Στη συνέχεια, επιλέγουμε την κατάλληλη γεωμετρία εκτέλεσης (blockSize, gridSize) και το μέγεθος της shared memory (clusterBlockSharedDataSize), τα οποία καθορίζουν πόσοι πόροι GPU θα χρησιμοποιηθούν και πώς θα φορτώνονται τα δεδομένα των κεντροειδών στην ταχύτερη shared memory.

Ο αλγόριθμος εκτελείται σε επαναλήψη, αρχικοποιούμε ένα μετρητή αλλαγών (dev_delta_ptr) στο μηδέν και καλούμε τον kernel find_nearest_cluster. Εκεί, για κάθε αντικείμενο, βρίσκουμε το πλησιέστερο cluster (υπολογίζοντας αποστάσεις από τα κέντρα) και προσθέτουμε τις συντεταγμένες του αντικειμένου στα μερικά αθροίσματα του αντίστοιχου cluster (devicenewClusters), καταγράφοντας ταυτόχρονα πόσα αντικείμενα έχουν «μετακινηθεί» (dev_delta_ptr). Αμέσως μετά αντιγράφουμε το dev_delta_ptr στην CPU, ώστε να γνωρίζουμε πόσα αντικείμενα άλλαξαν cluster.

Σε επόμενο βήμα, καλείται ο kernel update_centroids, ο οποίος διαιρεί τα αθροίσματα συντεταγμένων κάθε cluster με τον αριθμό των αντικειμένων που του αντιστοιχούν, υπολογίζοντας έτσι τα νέα κέντρα. Εάν το ποσοστό αντικειμένων που άλλαξαν cluster (delta / numObjs) είναι μικρότερο από ένα προκαθορισμένο όριο (threshold) ή αν έχουμε συμπληρώσει τον μέγιστο αριθμό επαναλήψεων (loop_threshold), η διαδικασία σταματά. Διαφορετικά, τα ανανεωμένα κέντρα αντιγράφονται πίσω στην GPU και ο αλγόριθμος επαναλαμβάνεται. Με το πέρας αυτών των βημάτων, έχουμε την τελική ανάθεση των αντικειμένων σε clusters καθώς και τις τελικές συντεταγμένες των κέντρων τους.

```

/* TODO: Copy me from transpose version*/
//calloc_2d(...) -> [numCoords][numObjs]
double **dimObjects = (double**) calloc_2d(numCoords, numObjs, sizeof(double));
//calloc_2d(...) -> [numCoords][numClusters]
double **dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
//calloc_2d(...) -> [numCoords][numClusters]
double **newClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));

printf("\n|-----Full-offload GPU Kmeans-----|\n\n");

/* TODO: Copy me from transpose version*/
for (i = 0; i < numObjs; i++) {
    for(j = 0; j < numCoords; j++) {
        dimObjects[j][i] = objects[i*numCoords+j];
    }
}

/* TODO: Calculate Grid size, e.g. number of blocks. */
const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock -1) / numThreadsPerClusterBlock;
/* Define the shared memory needed per block.
   - BEWARE: We can overrun our shared memory here if there are too many
   clusters or too many coordinates!
   - This can lead to occupancy problems or even inability to run.
   - Your exercise implementation is not requested to account for that (e.g. always assume deviceClusters fit in shmemClusters */
const unsigned int clusterBlockSharedDataSize = numClusters*numCoords*sizeof(double);

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

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

/* TODO: can use different blocksize here if deemed better */
const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ? blockSize : numCoords * numClusters;
/* TODO: calculate dim for "update_centroids" */
const unsigned int update_centroids_dim_sz = (numCoords*numClusters+update_centroids_block_sz-1)/update_centroids_block_sz;
timing_gpu = wtime();
// TODO: use dim for "update_centroids" and fire it
update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
    (numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);

```

Kernel find_nearest_cluster():

Κάθε thread αναλαμβάνει ένα αντικείμενο, υπολογίζει το πλησιέστερο cluster και κάνει atomicAdd για να αθροίσει τις συντεταγμένες του αντικειμένου στα μερικά αθροίσματα newClusters, καθώς και να αυξήσει τον μετρητή newClusterSize.

Μόλις τελειώσει αυτός ο kernel, έχουμε ολοκληρωμένη την «πρόχειρη» ενημέρωση: δηλαδή ξέρουμε ανά cluster πόσα αντικείμενα του αντιστοιχούν και το άθροισμα των διανυσμάτων τους.

Kernel update_centroids():

Κάθε thread αναλαμβάνει να διαιρέσει το άθροισμα των συντεταγμένων ενός cluster διάστασης με το πλήθος των αντικειμένων σε αυτό το cluster, ώστε να πάρουμε τη «μέση τιμή» (το νέο κέντρο).

Εδώ η κατανομή των threads συνήθως ακολουθεί το πλήθος numClusters * numCoords, που σημαίνει ότι κάθε thread μπορεί να ενημερώσει μία συνιστώσα ενός cluster.

Αυτή η προσέγγιση δουλεύει και είναι απλή, με ελάχιστο επιπλέον κόστος συγχρονισμού:

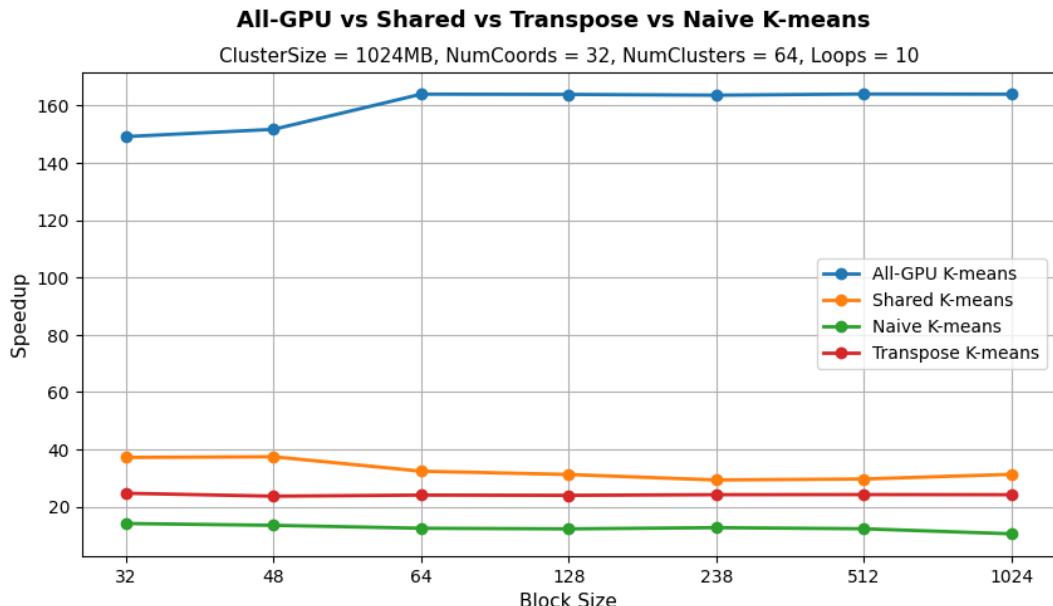
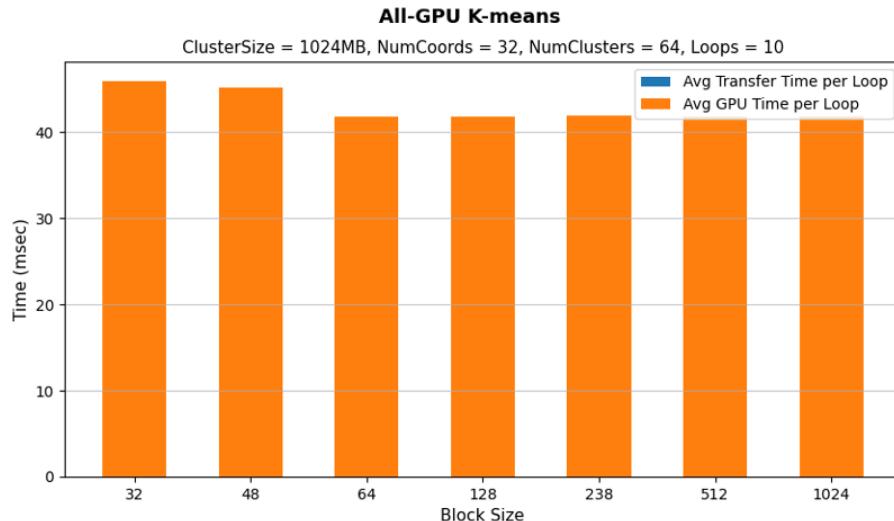
- Ο συγχρονισμός (global) γίνεται αυτόματα στο τέλος κάθε kernel.
- Τα threads του πρώτου kernel «πιάνουν» όλα τα αντικείμενα, ενώ τα threads του δεύτερου kernel «πιάνουν» όλες τις συνιστώσες των cluster (αποφεύγοντας σε

σημαντικό βαθμό τα ανενεργά threads, μιας και το numCoords * numClusters συνήθως έχει επαρκές μέγεθος).

Συνεπώς, η λύση αυτή (α) λειτουργεί σωστά, και (β) είναι αρκετά βολική γιατί ο διαχωρισμός σε δύο σαφείς kernels (assignment + update) καθιστά τον κώδικα ευκολότερο στον έλεγχο.

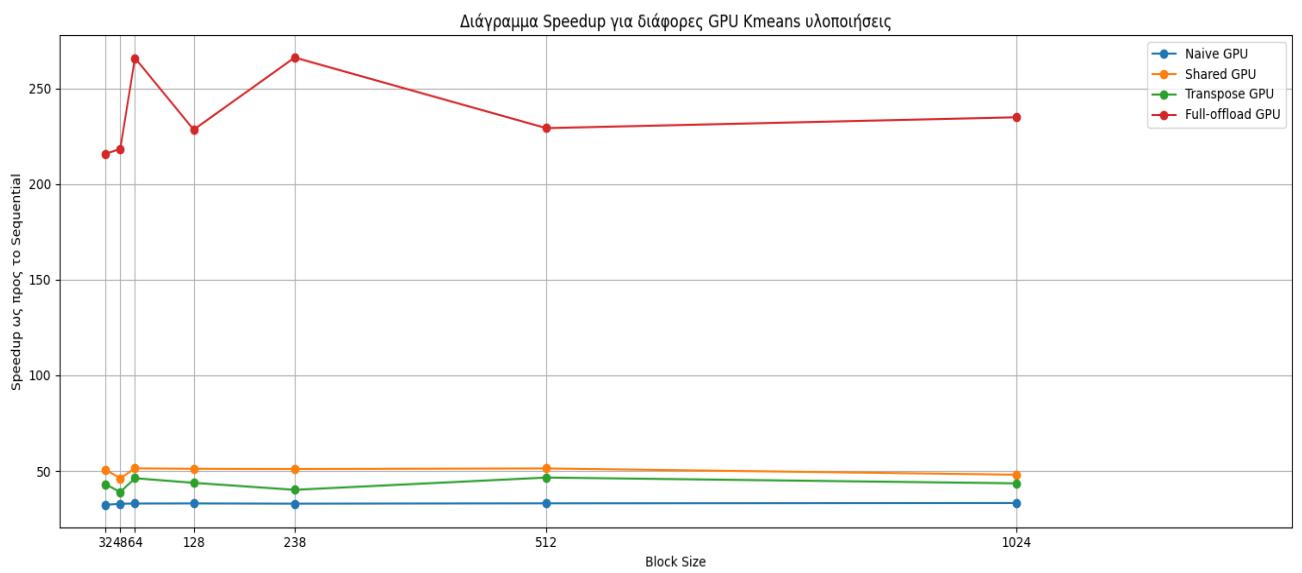
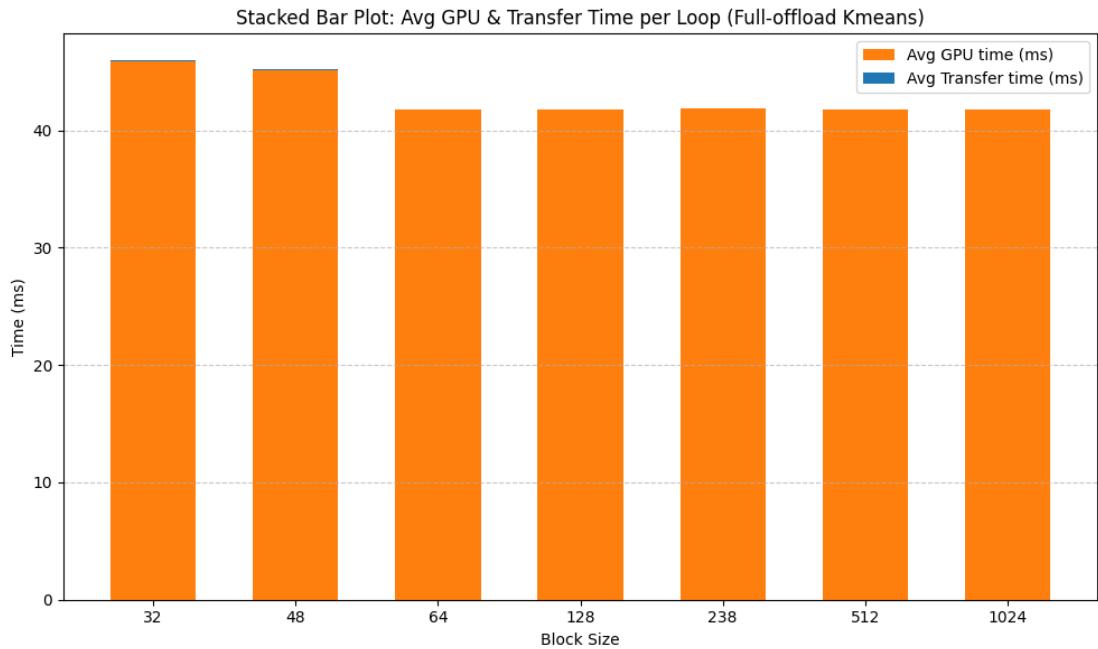
1. Πρώτο configuration {1024, 32, 64, 10}:

Ο χρόνος μεταφοράς δεδομένων είναι τόσο μικρός συγκριτικά με το χρόνο εκτέλεσης στην GPU που δε φαίνεται στο παρακάτω stacked barplot.



Δεύτερο configuration {1024, 2, 64, 10}:

Αντίστοιχα και σε αυτή την περίπτωση ο χρόνος μεταφοράς δεδομένων είναι τόσο μικρός συγκριτικά με το χρόνο εκτέλεσης στην GPU που δε φαίνεται στο παρακάτω stacked barplot.



Η Full-Offload GPU υλοποίηση φέρνει δραματική βελτίωση στην απόδοση του k-means σε σχέση με τις άλλες προσεγγίσεις, κυρίως για τους εξής λόγους:

- Εκτέλεση ολόκληρου του αλγορίθμου στην GPU

Η υπολογιστική ρουτίνα ανατίθεται εξ ολοκλήρου στην κάρτα γραφικών. Έτσι αποφεύγεται το συνεχές πέρα δώθε δεδομένων ανάμεσα σε CPU και GPU, που είναι εξαιρετικά χρονοβόρο. Ειδικά στο k-means, κάθε επανάληψη απαιτεί υπολογισμούς απόστασης για όλα τα σημεία και ενημέρωση κέντρων. Αν αυτά τα βήματα απαιτούσαν συνεχή μεταφορά μεγάλου όγκου δεδομένων (π.χ. χαρακτηριστικά σημείων) στην CPU για να επεξεργαστεί, θα

προσέθεταν μεγάλο overhead. Στην Full-Offload λύση, τα δεδομένα μένουν καρφωμένα στη GPU καθ' όλη τη διάρκεια της επεξεργασίας.

- Εκμετάλλευση της αρχιτεκτονικής της GPU

Η GPU διαθέτει πολλαπλούς πυρήνες (CUDA cores/Stream Multiprocessors) που τρέχουν χιλιάδες νήματα ταυτόχρονα. Με τη Full-Offload υλοποίηση, εκτελούμε τους υπολογισμούς εντελώς παράλληλα, αξιοποιώντας στο μέγιστο το πλήθος των νημάτων και το υψηλό throughput της μνήμης. Η οργάνωση των δεδομένων και των blocks μπορεί να γίνει με τρόπο που βελτιώνει τη συνεχή πρόσβαση στη μνήμη, ενώ δεν απαιτείται πια αντίστοιχος συγχρονισμός με την CPU.

- Απαλλαγή από CPU overhead και context switching

Σε υλοποιήσεις όπου κάθε επανάληψη «σπάει» μεταξύ CPU και GPU, συχνά παρατηρείται overhead από το context switching και την αναμονή των δύο μερών να ολοκληρώσουν επιμέρους εργασίες. Με την εξολοκλήρου εκτέλεση στην GPU, η CPU πλέον μένει σχεδόν αδρανής και δεν αποτελεί bottleneck στον κύκλο εργασιών. Αυτό μειώνει σημαντικά το συνολικό χρόνο.

- Συγκριτικά με άλλες GPU υλοποιήσεις (Naive, Shared, Transpose)

Στις Naive GPU προσεγγίσεις, αν και οι πυρήνες εκτελούνται στην GPU, συχνά επιστρέφουμε δεδομένα στην CPU σε κάθε βήμα ή δεν εκμεταλλεύμαστε πλήρως τη μνήμη της GPU (π.χ. περισσότερες ασυνεχείς προσβάσεις στη μνήμη, επιπλέον μεταφορές κ.λπ.).

Στις Shared ή Transpose προσεγγίσεις, μπορεί μεν να βελτιστοποιείται η προσπέλαση της μνήμης ή να αξιοποιείται shared memory, αλλά αν κάθε loop ολοκληρώνεται με επικοινωνία CPU-GPU, το κέρδος της βελτιστοποίησης μειώνεται δραστικά.

Αντιθέτως, στη Full-Offload, η επικοινωνία CPU-GPU περιορίζεται στην αρχική μεταφορά (φόρτωση) των δεδομένων και στην τελική μεταφορά των αποτελεσμάτων, γλιτώνοντας μεγάλο ποσό overhead.

2. Το `block_size` επηρεάζει άμεσα την ταχύτητα εκτέλεσης ενός GPU kernel, καθώς καθορίζει τον τρόπο με τον οποίο κατανέμονται τα νήματα στα Streaming Multiprocessors (SMs) και επομένως τον βαθμό παράλληλης εκτέλεσης (occupancy). Με σωστή επιλογή `block_size` επιτυγχάνεται μεγαλύτερη αξιοποίηση των πόρων της GPU, καθώς φορτώνονται περισσότερα πλήρη warps και μειώνεται ο χαμένος χρόνος από αδρανείς υπολογιστικές μονάδες. Παράλληλα, η μνήμη (global, shared) χρησιμοποιείται πιο αποτελεσματικά, αφού το `block_size` επηρεάζει την ποσότητα shared memory που δεσμεύεται ανά block και καθορίζει πόσα blocks μπορούν να τρέχουν ταυτόχρονα σε έναν SM.

Επιπλέον, ένα κατάλληλο `block_size` διευκολύνει την ομαδοποιημένη (coalesced) πρόσβαση στη μνήμη, αποφεύγοντας cache misses που κοστίζουν σε χρόνο. Όταν το `block_size` είναι πολύ μεγάλο, μπορεί να πέσει η occupancy λόγω έλλειψης shared memory, ενώ όταν είναι πολύ μικρό δεν αξιοποιείται πλήρως η παράλληλη ισχύς της GPU. Συνεπώς, η επιλογή `block_size` πρέπει να γίνεται με γνώμονα το μέγεθος του προβλήματος, την ποσότητα μνήμης που απαιτεί ο kernel και τα χαρακτηριστικά της συγκεκριμένης GPU, ώστε να μεγιστοποιείται η απόδοση στην εκτέλεση του αλγορίθμου (π.χ. k-means).

Στην περίπτωση μας, είδαμε ότι για την Full-Offload GPU υλοποίηση, τα block sizes 64 και 238 οδήγησαν στους χαμηλότερους συνολικούς χρόνους (περίπου 580–582 ms για 10 επαναλήψεις), γεγονός που δείχνει ότι εκεί η occupancy και η πρόσβαση στη μνήμη ήταν πιο αποδοτικές σε σχέση με άλλα μεγέθη.

3. Το update_centroids kernel, όπως υλοποιείται παραπάνω, είναι λειτουργικό για χρήση σε GPU. Αφού, το κάθε thread ενημερώνει (διαιρεί) ένα στοιχείο του πίνακα deviceClusters, αποφεύγοντας race conditions. Η προσέγγιση με 1D grid είναι απλή και κάνει σωστά τη δουλειά της.

Ωστόσο, δεν είναι απαραίτητα η πιο βελτιστοποιημένη λύση:

Η πρόσβαση σε devicenewClusterSize[cluster] από πολλούς threads ίσως μη γίνεται με τον πιο coalesced τρόπο. Για μεγάλο αριθμό clusters ή συντεταγμένων, θα μπορούσε να βοηθήσει η οργάνωση 2D grid/block ή η φόρτωση των τιμών clusterSize σε shared memory.

Από την προηγούμενη ανάλυση των bottlenecks (Naive, Transpose, Shared), ξέρουμε ότι:

Η Naive GPU μεταφέρει συνεχώς δεδομένα μεταξύ CPU-GPU και αφήνει μεγάλο φόρτο εργασίας στην CPU, ανεβάζοντας έτσι τον χρόνο CPU και τα transfer times.

Η Transpose GPU μειώνει μεν κάπως την επιβάρυνση της CPU, αλλά δεν εκμεταλλεύεται πλήρως τη GPU.

Η Shared GPU είναι η πιο αποδοτική από τις «μερικές» GPU υλοποιήσεις, αφού περιορίζει αρκετά τα transfers και χρησιμοποιεί αποτελεσματικά τη shared memory, μειώνοντας τον χρόνο πρόσβασης στη μνήμη.

Η All-GPU (Full-offload) έκδοση, κάνει κάτι διαφορετικό από τις άλλες:

Όλος ο αλγόριθμος τρέχει στη GPU: Η CPU πλέον δεν συμμετέχει στο υπολογιστικό κομμάτι. Μένουμε στη GPU καθ' όλη τη διάρκεια του k-means: Δεν υπάρχουν πια επαναλαμβανόμενες αντιγραφές δεδομένων ανά loop. Τα δεδομένα μεταφέρονται μία φορά στην GPU, υπολογίζονται όλα τα βήματα εκεί (ευρετήριο κοντινότερου κέντρου, ενημέρωση κέντρων) και μόνον στο τέλος επιστρέφουμε τα αποτελέσματα. Έτσι εξαλείφουμε το μεγαλύτερο μέρος του κόστους επικοινωνίας CPU-GPU. Καταργούμε το περίτλοκο synchronization: Εφόσον η CPU δεν παρεμβάλλεται ενεργά σε κάθε loop, μειώνεται το overhead από context switches ή από αναμονή ολοκλήρωσης των kernels για να τρέξει ο κώδικας CPU.

Συμπερασματικά, ακόμη κι αν το update_centroids δεν είναι υπερ-βελτιστοποιημένο, η απουσία μεταφοράς δεδομένων σε κάθε loop και η ελαχιστοποίηση του φόρτου στην CPU εξηγούν τη μεγάλη διαφορά στην απόδοση της Full-offload GPU έκδοσης συγκριτικά με τις παλιές υλοποιήσεις.

4. Η διαφορά βρίσκεται στην διαφορά του αριθμού των συντεταγμένων, όπου συναρτήσει των γραφημάτων, στη Full-Offload GPU υλοποίηση, παρατηρούμε μεγαλύτερο speedup όταν έχουμε μόνο 2 συντεταγμένες (coords=2) σε σχέση με την περίπτωση όπου έχουμε 32 συντεταγμένες (coords=32). Ο βασικός λόγος είναι πως, για το ίδιο συνολικό μέγεθος δεδομένων (π.χ. 1GB):

Περισσότερα αντικείμενα όταν οι συντεταγμένες είναι λίγες (coords=2)

Εφόσον κάθε σημείο έχει ελάχιστα χαρακτηριστικά (2 μόνο στοιχεία), για να γεμίσει το ίδιο 1GB χρειάζονται πολύ περισσότερα αντικείμενα. Η GPU, μοιράζει τον υπολογισμό όλων αυτών των σημείων σε χιλιάδες παράλληλα νήματα και μπορεί να ολοκληρώσει την ίδια δουλειά πολύ ταχύτερα. Αυτό οδηγεί σε πολύ υψηλό speedup.

Λιγότερα αντικείμενα όταν αυξάνονται οι συντεταγμένες (coords=32)

Όσο μεγαλώνει ο αριθμός των συντεταγμένων, τόσο βαρύτερο γίνεται κάθε σημείο (32 χαρακτηριστικά αντί για 2), οπότε ο συνολικός αριθμός των σημείων μειώνεται για να παραμείνουμε στο 1GB. Η GPU έχει να φορτώσει περισσότερα bytes ανά σημείο (γίνεται πιο “memory-bound” σε σχέση με την περίπτωση των 2 coords), πράγμα που περιορίζει κάπως το συγκριτικό της πλεονέκτημα. Ως αποτέλεσμα, το speedup παραμένει μεν μεγάλο, αλλά όχι όσο στην περίπτωση των 2 coords.

Bonus 2: Delta reduction (All-GPU) version

Σε αυτό το μέρος της άσκησης υλοποιήσαμε το delta με reduction. Συμπληρώσαμε την έκδοση του αλγορίθμου `cuda_kmeans_all_gpu_delta_reduction.cu` υλοποιώντας όλα τα νέα TODO και χρησιμοποιώντας τα κομμάτια από την *all_gpu version*. Παρακάτω απεικονίζονται οι αλλαγές που πραγματοποιήσαμε στον κώδικα (πολλές από τις αλλαγές έχουν ήδη παρουσιαστεί αναλυτικά σε προηγούμενες ενότητες):

- `get_tid()`: υπολογισμός μοναδικού ID κάθε thread. Χρησιμοποιείται για να προσδιορίσει ποιο αντικείμενο επεξεργάζεται κάθε thread.

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

- `euclid_dist_2_transpose()`: υπολογισμός της τετραγωνικής Ευκλείδειας απόστασης μεταξύ ενός αντικειμένου και ενός cluster

```
/* TODO: calculate the euclid_dist of elem=objectId of objects from elem=clusterId from clusters, but for column-base format!!! */
double to_square = 0.0;

/* TODO: 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++) {
    to_square = objects[numObjs*i+objectId]-clusters[numClusters*i+clusterId];
    ans += to_square*to_square;
}
return (ans);
}
```

- `find_nearest_cluster()`: Υπολογισμός του πλησιέστερου cluster για κάθε σημείο δεδομένων. Αν το cluster του σημείου αλλάζει, το μετράει στη `devdelta`. Ενημέρωση των νέων μέσων όρων των clusters (`devicenewClusters`)

```
__global__ static
void find_nearest_cluster(int numCoords,
                           int numObjs,
                           int numClusters,
                           double *deviceobjects,           // [numCoords][numObjs]
                           /* ... */

                           /* TODO: If you choose to do (some of) the new centroid calculation here, you will need
                           some extra parameters here (from "update_centroids"). */

                           /* ... */

                           int * devicenewClusterSize,
                           // added these two parameters ...
                           double * devicenewClusters,
                           double *deviceClusters,        // [numCoords][numClusters]
                           int *deviceMembership,        // [numObjs]
                           double *devdelta) {

    extern __shared__ double shmemClusters[];
}
```

```

/* TODO: copy me from shared version... */
int i,j ;
int local_tid = threadIdx.x;
for(i=local_tid;i<numClusters;i+=blockDim.x){
    for(j=0;j<numCoords;j++) shmemClusters[j*numClusters + i] = deviceClusters[j*numClusters + i] ;
}

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

    /* TODO: additional steps for calculating new centroids in GPU? */

        int index, i;
        double dist, min_dist;

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

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

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

```

- Reduction στο partial_deltas[] για να υπολογιστεί το τελικό devdelta

```

extern __shared__ double partial_deltas[];
// initialise partial_deltas
partial_deltas[local_tid] = 0.0;

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

```

- Υπολογισμός των νέων clusters

```

/* TODO: additional steps for calculating new centroids in GPU? */
int tid = get_tid();
int cluster = tid%numClusters;
int clusterSize;

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

/*
 * TODO: copy me from transpose version*
double **dimObjects = (double**) calloc_2d(numCoords, numObjs, sizeof(double)); //calloc_2d(...)-> [numCoords][numObjs]
double **dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double)); //calloc_2d(...)-> [numCoords][numClusters]
double **newClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double)); //calloc_2d(...)-> [numCoords][numClusters]

printf("\n|-----Full-offload GPU Kmeans-----|\n\n");

/* TODO: Copy me from transpose version*/
for (i=0; i<numObjs; i++) {
    for(j=0; j<numCoords; j++) {
        dimObjects[j][i] = objects[i*numCoords+j];
    }
}

```

```

const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize : numObjs;
const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) / numThreadsPerClusterBlock; /* TODO: Calculate Grid size,
| e.g. number of blocks. */
/* Define the shared memory needed per block.
- BEWARE: We can overrun our shared memory here if there are too many
clusters or too many coordinates!
- This can lead to occupancy problems or even inability to run.
- Your exercise implementation is not requested to account for that (e.g. always assume deviceClusters fit in shmemClusters */
const unsigned int clusterBlockSharedDataSize = (numClusters*numCoords + numThreadsPerClusterBlock)*sizeof(double);

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

timing_transfers = wtime();
/* TODO: Copy dev_delta_ptr to &delta
checkCuda(cudaMemcpy(...));
checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
transfers_time += wtime() - timing_transfers;

const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ? blockSize : numCoords * numClusters; /* TODO:
can use different blocksize here if deemed better */
const unsigned int update_centroids_dim_sz = (numCoords*numClusters+update_centroids_block_sz-1)/update_centroids_block_sz; /* TODO:
calculate dim for "update_centroids" */
timing_gpu = wtime();
// TODO: use dim for "update_centroids" and fire it
update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
    (numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);

```

1. Ακολουθώντας τις οδηγίες για την `find_nearest_cluster`, εφαρμόζουμε δεντρικό reduction για τον υπολογισμό της `devdelta` ανά block. Για να επιτευχθεί αυτό, πρέπει να αυξήσουμε το μέγεθος της shared memory κατάλληλα, καθώς το reduction θα απαιτήσει περισσότερους πόρους μνήμης. Στον πυρήνα `find_nearest_cluster` η μεθοδολογία του δεντρικού reduction για τον υπολογισμό της `devdelta` αποτελείται από τον υπολογισμό της διαφοράς από τον μέσο όρο του cluster από κάθε thread και στη συνέχεια, τη χρήση δεντρικής (tree-based) μεθοδολογίας για να συγκεντρωθούν τα αποτελέσματα ανά block, με τη χρήση της `__syncthreads()` για συγχρονισμό των threads.

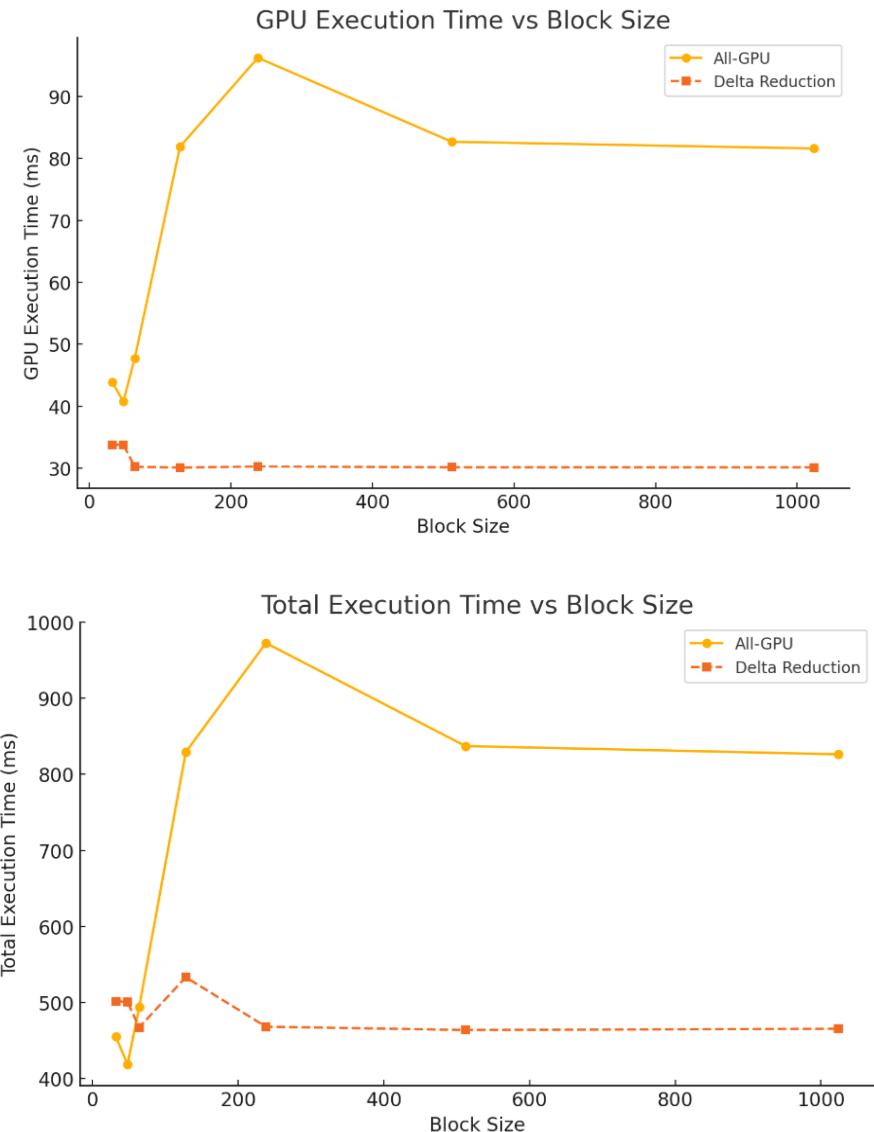
Αξιολόγηση της επίδοσης:

1. Επαναλάβαμε τις προηγούμενες μετρήσεις για την *all-gpu delta reduction version* και καταλήξαμε στα εξής συμπεράσματα:

Συγκρίνοντας την **All-GPU** και την **Delta Reduction** έκδοση του αλγορίθμου, παρατηρούμε διαφοροποιήσεις στην απόδοση που σχετίζονται κυρίως με τον τρόπο που διαχειρίζονται οι δύο εκδόσεις τη μνήμη και την εκτέλεση των υπολογισμών. Η **Delta Reduction** παρουσιάζει γενικά καλύτερη απόδοση σε μεγαλύτερα block sizes, καθώς η χρήση του tree reduction για τον υπολογισμό του delta μειώνει τη χρήση των global atomics. Με αυτόν τον τρόπο, βελτιώνεται και ο χρόνος εκτέλεσης. Συγκεκριμένα, στην All-GPU έκδοση με `block_size = 64`, ο χρόνος GPU είναι 47.72 ms, ενώ στη Delta Reduction είναι 30.22 ms για το ίδιο block size, δείχνοντας έτσι την αποτελεσματικότητα της delta reduction. Ωστόσο, η All-GPU έκδοση φαίνεται να είναι πιο αποδοτική σε μικρότερα block sizes, με το `block_size = 32` να έχει συνολικό χρόνο εκτέλεσης 454.99 ms, σε σύγκριση με τα 501.57 ms της Delta Reduction. Η αύξηση του `block_size` στην All-GPU έκδοση αυξάνει σημαντικά τους χρόνους εκτέλεσης, με την απόδοση να επιδεινώνεται, λόγω της υπερφόρτωσης της μνήμης και της χρήσης των global atomics για τη συγχώνευση των αποτελεσμάτων. Αντίθετα, στην Delta Reduction, οι μεγαλύτεροι χρόνοι εκτέλεσης σε μεγαλύτερα block sizes (`block_size = 238`) συνεχίζουν να είναι καλύτεροι από την αντίστοιχη All-GPU έκδοση. Η διαχείριση της μνήμης και η ταχύτερη εκτέλεση στην GPU στην Delta Reduction αποδίδεται στην καλύτερη αξιοποίηση της shared memory και στην αποφυγή των ακριβών global atomics.

2. Παίζει κάποιο διαφορετικό ρόλο το block_size και γιατί;

Από τα παραπάνω καταλήγουμε στο συμπέρασμα ότι η επιλογή του block_size έχει καθοριστικό ρόλο στην απόδοση, με μικρότερα block sizes να είναι πιο αποδοτικά στην All-GPU, ενώ τα μεγαλύτερα block sizes ευνοούν την Delta Reduction, αφού επιτυγχάνεται καλύτερη χρήση των πόρων της μνήμης και των GPU threads.



Παραπάνω, απεικονίζονται τα διαγράμματα που συγκρίνουν τον χρόνο εκτέλεσης στην GPU (`t_gpu_avg`) και τον συνολικό χρόνο εκτέλεσης (`total`) για τις εκδόσεις All-GPU και Delta Reduction, σε σχέση με το block size. Βλέπουμε ότι η Delta Reduction έχει μικρότερο GPU execution time, αλλά ο συνολικός χρόνος δεν είναι πάντα καλύτερος, κάτι που οφείλεται στους προαναφερθέντες παράγοντες.

Άσκηση 4

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

Ο σκοπός αυτής της άσκησης είναι η παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης με τη χρήση του προγραμματιστικού εργαλείου MPI.

4.1. Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου **K-means**

Αρχικά, πραγματοποιήσαμε τις ακόλουθες αλλαγές στον κώδικα:

file_io.c

- Υπολογισμός πόσων αντικειμένων θα επεξεργαστεί κάθε διεργασία (rank) στη διαδικασία της κατανομής εργασίας

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

- Υπολογισμός των πινάκων sendcounts και displs, οι οποίοι είναι απαραίτητοι για τη διασπορά των δεδομένων σε κάθε διεργασία

```
/*
 * TODO: calculate sendcounts and displs, which will be used to scatter data to each rank.
 * Hint: sendcounts: number of elements sent to each rank
 *       displs: displacement of each rank's data
 */
int quotient = numObjs/size;
int remainder = numObjs%size;
for (i = 0; i < size; i++) {
    if (i < remainder) sendcounts[i] = quotient + 1;
    else sendcounts[i] = quotient;
}
displs[0] = 0;

for (i = 1; i < size; i++) displs[i] = displs[i - 1] + sendcounts[i - 1];

}
```

- Broadcast (MPI_Bcast) των πινάκων sendcounts και displs σε όλες τις διεργασίες

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

- Scatter (MPI_Scatterv) τα αντικείμενα σε κάθε διεργασία

```
/*
 * TODO: Scatter objects to every rank. (hint: each rank may receive different number of objects)
 */
MPI_Scatterv(objects,sendcounts,displs,MPI_DOUBLE,rank_objects,*rank_numObjs,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

main.c

- Broadcast (MPI_Bcast) τα initial clusters σε όλους και έπειτα προσαρμογή του πλήθους των αντικειμένων για τα οποία θα δουλέψει κάθε διεργασία

```
/*
 * TODO: Broadcast initial cluster positions to all ranks
 */
MPI_Bcast(clusters,numClusters*numCoords,MPI_DOUBLE,0,MPI_COMM_WORLD);

// membership: the cluster id for each data object
membership = (int*) malloc(rank_numObjs * sizeof(int));
tot_membership = (int*) malloc(numObjs * sizeof(int));

// start the core computation
/*
 * TODO: Fix number of objects that this kmeans function call will process
 */
kmeans(objects, numCoords, rank_numObjs, numClusters, threshold, loop_threshold, membership, clusters);
```

- Υπολογισμός των arrays recvcounts και displs, τα οποία χρησιμοποιούνται για τη συλλογή δεδομένων από κάθε διεργασία

```
// Gather membership information from all ranks to tot_membership
int recvcounts[size], displs[size];
if (rank == 0) {
    /* TODO: Calculate recvcounts and displs, which will be used to gather data from each rank.
     * Hint: recvcounts: number of elements received from each rank
     *       displs: displacement of each rank's data
     */
    int quotient = numObjs/size;
    int remainder = numObjs%size;

    int i = 0 ;
    for(i; i<size;i++){
        if(i<remainder) recvcounts[i]=quotient+1;
        else recvcounts[i] = quotient;
    }

    displs[0]=0;
    for(i=1;i<size;i++) displs[i] = displs[i-1] + recvcounts[i-1];
}
```

- Μετάδοση των πινάκων recvcounts και displs σε όλες τις διεργασίες χρησιμοποιώντας MPI_Bcast και τέλος, συλλογή δεδομένων (MPI_Gatherv) από όλες τις διεργασίες

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

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

kmeans.c

- Reduction των δεδομένων των clusters από τις τοπικές μνήμες κάθε διεργασίας στην κοινή μνήμη

```
/*
 * TODO: Perform reduction of cluster data (rank_newClusters, rank_newClusterSize) from local arrays to shared.
 */
MPI_Allreduce(rank_newClusters,newClusters, numClusters * numCoords,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(rank_newclusterSize,newClusterSize,numClusters ,MPI_INT,MPI_SUM, MPI_COMM_WORLD);
```

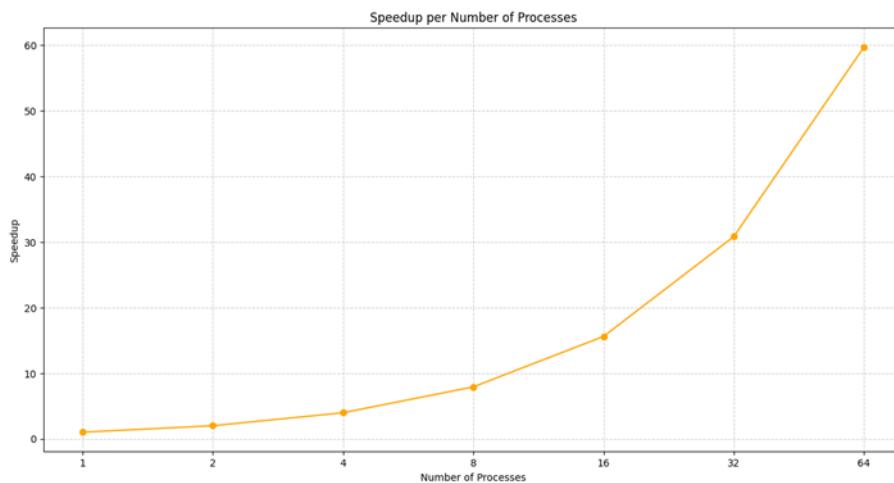
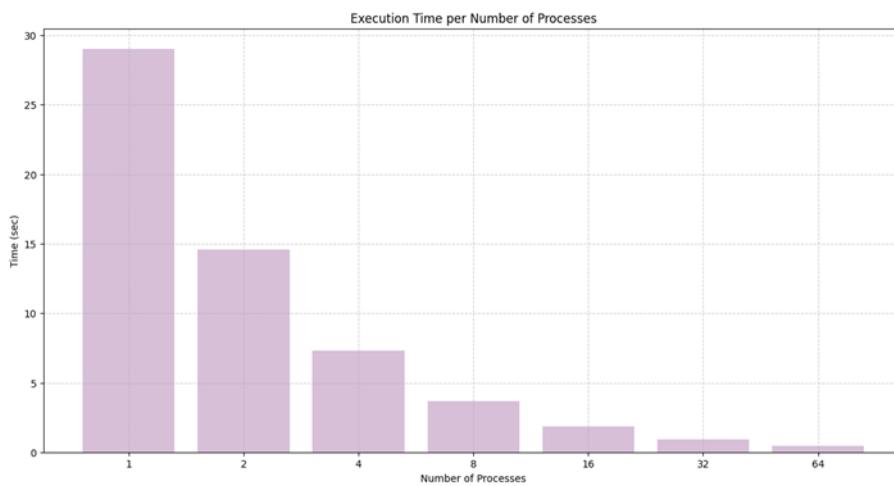
- Reduction των δεδομένων από τη μεταβλητή rank_delta σε μία κοινή μεταβλητή delta, που θα χρησιμοποιηθεί για τον έλεγχο της σύγκλισης

```

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

```

Έπειτα, συγκεντρώσαμε μετρήσεις για το configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10} για 1, 2, 4, 8, 16, 32 και 64 MPI διεργασίες στα clones και από τις μετρήσεις που προέκυψαν, δημιουργήσαμε τα διαγράμματα barplot για τον χρόνο εκτέλεσης (x-axis = sequential και processes, y-axis = time) και το αντίστοιχο speedup plot (x-axis = sequential και processes, y-axis = seq_time/time). Τα διαγράμματα ακολουθούν στη συνέχεια:



Σε ό,τι αφορά τα παραπάνω διαγράμματα, παρατηρούμε ότι η παράλληλη έκδοση του αλγορίθμου παρουσιάζει εξαιρετική κλιμακωσιμότητα. Συγκεκριμένα, ο χρόνος εκτέλεσης μειώνεται σημαντικά καθώς προστίθενται περισσότερες διεργασίες. Από 29.0096 δευτερόλεπτα με 1 διεργασία, ο χρόνος πέφτει στα 0.4856 δευτερόλεπτα με 64 διεργασίες, κάτι που υποδεικνύει ότι το σύστημα επωφελείται από την προσθήκη νέων διεργασιών χωρίς να υπάρχουν καθυστερήσεις επικοινωνίας ή συγχρονισμού που να επηρεάζουν την απόδοση.

Το speedup πλησιάζει πολύ το γραμμικό, υποδεικνύοντας ότι για κάθε επιπλέον διεργασία, ο χρόνος εκτέλεσης μειώνεται σχεδόν γραμμικά.

Αυτό επιτυγχάνεται κυρίως λόγω του γεγονότος ότι κατά την εκτέλεση της συνάρτησης `kmeans()`, κάθε διεργασία επεξεργάζεται ανεξάρτητα τα δικά της αντικείμενα χωρίς να απαιτεί επικοινωνία με άλλες διεργασίες. Η μόνη στιγμή που απαιτείται επικοινωνία είναι στο τέλος κάθε iteration, όταν γίνεται ένα συνολικό collective communication μέσω της λειτουργίας `Allreduce`, ώστε να ενημερωθούν οι κοινές δομές δεδομένων για όλα τα ranks.

Επιπλέον, το κόστος των αρχικών και τελικών επικοινωνιών είναι πολύ μικρό, καθώς αυτές οι διαδικασίες, όπως η κατανομή των αντικειμένων και το συνολικό `reduce` για το συνολικό membership, πραγματοποιούνται μόνο μία φορά στην αρχή και μία στο τέλος του αλγορίθμου. Επομένως, τα overheads επικοινωνίας δεν επιβαρύνουν σημαντικά τη συνολική απόδοση του αλγορίθμου.

Αυτό δείχνει ότι οι καθυστερήσεις συγχρονισμού και επικοινωνίας μεταξύ των διεργασιών είναι πολύ μικρές και δεν επιβαρύνουν την απόδοση, επιτρέποντας στην παράλληλη έκδοση του αλγορίθμου να πετύχει εξαιρετική απόδοση με ελάχιστα overheads.

- **Προαιρετικά (Bonus):** Σύγκριση των χρόνων εκτέλεσης που συγκεντρώσαμε για την MPI υλοποίηση σε σχέση με την OpenMP που υλοποιήσατε προηγουμένως.

Στη σύγκριση της MPI και OpenMP υλοποίησης, παρατηρούμε ότι η MPI έχει καλύτερη κλιμακωσιμότητα και αποδίδει καλύτερα όσο αυξάνουμε τον αριθμό των διεργασιών. Αν και η MPI παρουσιάζει μεγαλύτερο κόστος για λίγες διεργασίες λόγω της επικοινωνίας και του συγχρονισμού, γίνεται πιο αποδοτική όσο αυξάνονται οι διεργασίες, καθώς οι κόμβοι μπορούν να προστεθούν εύκολα και χωρίς περιορισμούς μνήμης. Αντίθετα, στην OpenMP η απόδοση περιορίζεται από την κοινή μνήμη και το contention που δημιουργείται με περισσότερους πυρήνες. Συνολικά, για συστήματα με μεγάλους αριθμούς διεργασιών, η MPI είναι η προτιμώμενη επιλογή λόγω της καλύτερης κλιμακωσιμότητας και των μικρότερων overheads στην επικοινωνία.

4.2 Διάδοση Θερμότητας σε Δύο Διαστάσεις

Σε αυτό το κεφάλαιο, θα μελετήσουμε την επίδοση τριών αλγορίθμων που χρησιμοποιούνται για την επίλυση του προβλήματος της διάδοσης θερμότητας σε δύο διαστάσεις. Πιο συγκεκριμένα, οι αλγόριθμοι που θα μελετήσουμε είναι:

- η μέθοδος Jacobi,
- η μέθοδος Gauss – Seidel με Successive Over – Relaxation (SOR) και
- η μέθοδος Red – Black με SOR.

Σκοπός μας είναι να παραλληλοποιήσουμε κάθε έναν από τους προαναφερόμενους αλγορίθμους σε αρχιτεκτονική κατανεμημένης μνήμης με μοντέλο ανταλλαγής μηνυμάτων.

Μέθοδος Jacobi

Σύμφωνα με την επαναληπτική μέθοδο Jacobi, η θερμότητα σε κάθε σημείο του πλέγματός μας τη χρονική στιγμή $t + 1$ θα είναι ίση με το μέσο όρο των θερμοτήτων που καταγράφηκαν τη χρονική στιγμή t σε 4 γειτονικά σημεία. Η εξίσωση που περιγράφει αυτήν τη σχέση είναι η ακόλουθη:

$$u_{x,y}^{t+1} = \frac{u_{x-1,y}^t + u_{x,y-1}^t + u_{x+1,y}^t + u_{x,y+1}^t}{4}$$

Συνεπώς, καταλαβαίνουμε ότι για τον υπολογισμό της θερμότητας ενός σημείου τη χρονική στιγμή $t + 1$, χρειαζόμαστε θερμότητες της χρονικής στιγμής t . Γι' αυτό το λόγο, θα

διατηρούμε δύο πλέγματα – πίνακες: ένα για την τρέχουσα χρονική στιγμή $t + 1$ και ένα για την προηγούμενη χρονική στιγμή t .

Ο αλγόριθμός μας θα τερματίζει, μόλις τα ομότιμα στοιχεία των δύο πινάκων για τις στιγμές $t + 1$ και t διαφέρουν ελάχιστα ή και καθόλου. Σε αυτήν την περίπτωση, ο αλγόριθμός μας έχει συγκλίνει. Ωστόσο, σε περίπτωση που δε θέλουμε να περιμένουμε μέχρι να συγκλίνει ο αλγόριθμος, θα έχουμε τη δυνατότητα να επιβάλλουμε στον αλγόριθμο να εκτελέσει ένα συγκεκριμένο πλήθος επαναλήψεων και, έπειτα, να τερματίσει, δίχως να πραγματοποιήσει κανέναν έλεγχο σύγκλισης.

Επόμενο βήμα είναι να αποφασίσουμε πώς θα μοιράσουμε τα δεδομένα μας σε κάθε μία MPI διεργασία. Αποφασίζουμε, λοιπόν, να οργανώσουμε τις διεργασίες σε ένα δισδιάστατο (καρτεσιανό) πλέγμα. Κάθε διεργασία αναλαμβάνει τους υπολογισμούς ενός χωρίου του πλέγματος θερμοτήτων, το οποίο χωρί ονήκει μόνο σε αυτήν. Προφανώς, για την ορθή διεξαγωγή των υπολογισμών, μία διεργασία μπορεί να χρειαστεί κάποιο δεδομένο που διαθέτει μία άλλη διεργασία και, έτσι, προκύπτει και η ανάγκη για επικοινωνία μεταξύ των διεργασιών μέσω ανταλλαγής μηνυμάτων.

Λαμβάνοντας όλα αυτά υπόψιν, είμαστε έτοιμοι να συμπληρώσουμε τον παράλληλο κώδικα που μας έχει δοθεί στο αρχείο `mpi_skeleton.c`.

Αρχικά, μεταβαίνουμε στην αρχή του κώδικα, στο σημείο όπου κάνουμε `include` τις κατάλληλες C βιβλιοθήκες και τα απαραίτητα header files. Εκεί, προσθέτουμε και τον ορισμό της μεταβλητής `TEST_CONV`, με τη βοήθεια της οποίας θα ενεργοποιούμε και θα απενεργοποιούμε τους ελέγχους σύγκλισης. Ορίζουμε, επίσης, και τον τύπο `bool`, ο οποίος δεν υπάρχει στη γλώσσα C, αλλά θα μας φανεί χρήσιμος στη συνέχεια. Προσθέτουμε και τη συνάρτηση `Jacobi()`, η οποία υλοποιεί την εξίσωση που είδαμε παραπάνω.

```
#define TEST_CONV
typedef enum { false, true } bool;

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

Όπως αναφέραμε και νωρίτερα, κάθε διεργασία θα πρέπει να διατηρεί δύο πίνακες: `u_previous` και `u_current` με τη θερμότητα των σημείων του δικού της χωρίου για τις χρονικές στιγμές t και $t + 1$. Σε κάθε έναν από αυτούς τους πίνακες, προσθέτουμε περιμετρικά ένα επιπλέον στρώμα δεδομένων, στο οποίο κάθε διεργασία θα αποθηκεύει τις γραμμές και τις στήλες που έλαβε έπειτα από επικοινωνία με γειτονικές διεργασίες. Αρχικοποιούμε τους πίνακες `u_previous` και `u_current`, γεμίζοντάς τους με 0. Έπειτα, χρησιμοποιώντας την εντολή `Scatterv()`, η διεργασία 0 μοιράζει κατάλληλα τα δεδομένα του πλέγματος θερμοτήτων σε κάθε διεργασία του καρτεσιανού communicator. Επομένως, ο πίνακας `u_current` κάθε διεργασίας περιέχει τις θερμότητες που αντιστοιχούν στο χωρίο της διεργασίας. Κάθε διεργασία αντιγράφει τα περιεχόμενα του πίνακα `u_current` στον πίνακα `u_previous`.

```

//----Rank 0 scatters the global matrix----//
//----Rank 0 scatters the global matrix----//
//*****TODO*****//

/*Fill your code here*/

/*Make sure u_current and u_previous are
 both initialized*/
zero2d(u_current,local[0]+2,local[1]+2);
zero2d(u_previous,local[0]+2,local[1]+2);

MPI_Scatterv(*U,scattercounts,scatteroffset,global_block,(u_current+local[1]*3),1,local_block,0,CART_COMM);

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

//*****

```

Όπως εξηγήσαμε και νωρίτερα, οι διεργασίες θα ανταλλάζουν μέσω μηνυμάτων τις περιμετρικές τους γραμμές και στήλες, ώστε να μπορεί να γίνει ορθά ο υπολογισμός των θερμοτήτων κάθε χωρίου από κάθε διεργασία. Για την ανταλλαγή μηνυμάτων, ορίζουμε νέους τύπους δεδομένων. Ορίζουμε τον τύπο row ως contiguous και τον τύπο column ως vector, διότι θέλουμε τα δεδομένα τύπου column να επέχουν μεταξύ τους κατά κάποιο stride.

```

//----Define datatypes or allocate buffers for message passing----//
//*****TODO*****//

/*Fill your code here*/

MPI_Datatype row;
MPI_Type_contiguous(local[1],MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&row);
MPI_Type_commit(&row);

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);

//*****

```

Στη συνέχεια, κάθε διεργασία πρέπει να βρει το rank των 4 γειτονικών διεργασιών με τις οποίες θα πρέπει να ανταλλάξει μηνύματα. Γι' αυτό το σκοπό, χρησιμοποιούμε την εντολή MPI_Cart_Shift(). Σημειώνουμε, όμως, ότι οι διεργασίες που βρίσκονται περιμετρικά του καρτεσιανού πλέγματος, δε θα έχουν 4 γείτονες, όπως οι διεργασίες εσωτερικά του πλέγματος. Αντιθέτως, οι διεργασίες περιμετρικά του καρτεσιανού πλέγματος, θα έχουν λιγότερους από 4 γείτονες. Σε αυτές τις περιπτώσεις, η εντολή MPI_Cart_Shift() θα επιστρέψει αρνητική τιμή, όταν δε βρίσκει ένα γείτονα. Γνωρίζοντας αυτήν την πληροφορία, είμαστε σε θέση να χειριστούμε κατάλληλα τις περιπτώσεις λιγότερων γειτόνων.

```

//----Find the 4 neighbors with which a process exchanges messages----//
//*****TODO*****//
int north, south, east, west;

/*Fill your code here*/

/*Make sure you handle non-existing
 neighbors appropriately/
MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);

//*****

```

Έχω ήδη διακρίνουμε και ξεχωρίζουμε τις διεργασίες σε τρεις κατηγορίες, ανάλογα με τη θέση τους στο καρτεσιανό πλέγμα. Πιο συγκεκριμένα, διακρίνουμε τις παρακάτω κατηγορίες:

- οι διεργασίες που βρίσκονται εσωτερικά του καρτεσιανού πλέγματος,
- οι διεργασίες που βρίσκονται περιμετρικά του καρτεσιανού πλέγματος και
- οι διεργασίες που βρίσκονται περιμετρικά του καρτεσιανού πλέγματος και χρειάζονται padding.

Σημειώνουμε ότι κάποιες διεργασίες που βρίσκονται περιμετρικά του πλέγματος ενδέχεται να χρειάζεται να παραγεμιστούν (padding) είτε με μία στήλη στα δεξιά (right padding) είτε με μία γραμμή στο τέλος (bottom padding), ώστε τα χωρία που διαθέτουν οι διεργασίες να είναι ομοιόμορφα και ίδιων διαστάσεων.

Παρακάτω φαίνονται αναλυτικά τα βήματα που ακολουθούμε σε αυτό το στάδιο:

```
-----Define the iteration ranges per process-----
//*****TODO*****//

int i_min,i_max,j_min,j_max;
int padding[2];

padding[1]=global_padded[1]-global[1];
padding[0]=global_padded[0]-global[0];

/*Fill your code here*/

/*Three types of ranges:
 -internal processes
 -boundary processes
 -boundary processes and padded global array
 */

bool is_internal(int x, int y){
    return x!=0 && y!=0 && x!=(grid[0]-1) && y!=(grid[1]-1);
}

bool is_right_padded(){
    return global[1]%grid[1]!=0;
}

bool is_bottom_padded(){
    return global[0]%grid[0]!=0;
}

bool is_boundary_not_padded(int x, int y){
    return !is_internal(x,y) && !is_right_padded() && !is_bottom_padded();
}

if(is_internal(rank_grid[0],rank_grid[1])){
    i_min=1;
    j_min=1;
    i_max=local[0]+1;
    j_max=local[1]+1;
}

else if(is_boundary_not_padded(rank_grid[0],rank_grid[1])){
    if(north<0 && west<0 && south>=0 && east>=0){ //north-west border
        i_min=2;
        j_min=2;
        i_max=local[0]+1;
        j_max=local[1]+1;
    }
    else if(west<0 && north>=0 && east>=0 && south>=0){ //middle west borders
        i_min=2;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1];
    }
}
```

```

        i_min=1;
        j_min=2;
        i_max=local[0]+1;
        j_max=local[1]+1;
    }
    else if(west<0 && south<0 && east>=0 && north>=0){ // south west border
        i_min=1;
        j_min=2;
        i_max=local[0];
        j_max=local[1]+1;
    }
    else if(south<0 && west>=0 && north>=0 && east>=0){ // south middle borders
        i_min=1;
        j_min=1;
        i_max=local[0];
        j_max=local[1]+1;
    }
    else if(east<0 && south<0 && north>=0 && west>=0){ // south east border
        i_min=1;
        j_min=1;
        i_max=local[0];
        j_max=local[1];
    }
    else if(east<0 && north>=0 && west>=0 && south>=0){ //east middle borders
        i_min=1;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1];
    }
    else if(north<0 && east<0 && west>=0 && south>=0){ //north east border
        i_min=2;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1];
    }
    else if(north<0 && east>=0 && south>=0 && west>=0){ //middle north borders
        i_min=2;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1]+1;
    }
    else if(west<0 && north<0 && south<0 && east>=0){//we have only east neighbor
        i_min=2;
        j_min=2;
        i_max=local[0];
        j_max=local[1]+1;
    }
    else if(east<0 && north<0 && south<0 && west>=0){ //only west neighbor
        i_min=2;
        j_min=1;
        i_max=local[0];
        j_max=local[1];
    }
    else if(east<0 && north<0 && south<0 && west<0){ //no neighbors
        i_min=2;
        j_min=2;
        i_max=local[0];
        j_max=local[1];
    }
}
}

else if(is_right_padded() && is_bottom_padded()){
    if(north<0 && east<0 && south>=0 && west>=0){ //north-east border
        i_min=2;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1]-padding[1];
    }
    else if(east<0 && north>=0 && south>=0 && west>=0){ //middle east borders
        i_min=1;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1]-padding[1];
    }
    else if(east<0 && south<0 && north>=0 && west>=0){ //south east border

```

```

        i_min=1;
        j_min=1;
        i_max=local[0]-padding[0];
        j_max=local[1]-padding[1];
    }
    else if(south<0 && north>=0 && east>=0 && west>=0){ //middle south borders
        i_min=1;
        j_min=1;
        i_max=local[0]-padding[0];
        j_max=local[1]+1;
    }
    else if(south<0 && west<0 && north>=0 && east>=0){ //south west border
        i_min=1;
        j_min=2;
        i_max=local[0]-padding[0];
        j_max=local[1]+1;
    }
}

else if(is_right_padded()){
    if(north<0 && east<0 && south>=0 && west>=0){ //north-east border
        i_min=2;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1]-padding[1];
    }
    else if(east<0 && north>=0 && south>=0 && west>=0){ //middle east borders
        i_min=1;
        j_min=1;
        i_max=local[0]+1;
        j_max=local[1]-padding[1];
    }
    else if(east<0 && south<0 && north>=0 && west>=0){ //south east border
        i_min=1;
        j_min=1;
        i_max=local[0];
        j_max=local[1]-padding[1];
    }
    else if(west<0 && north<0 && south<0 && east>=0){//we have only east neighbor
        i_min=2;
        j_min=2;
        i_max=local[0];
        j_max=local[1]+1;
    }
    else if(east<0 && north<0 && south<0 && west>=0){ //only west neighbor
        i_min=2;
        j_min=1;
        i_max=local[0];
        j_max=local[1]-padding[1];
    }
}

else if(is_bottom_padded()){
    if(east<0 && south<0 && north>=0 && west>=0){ //south east border
        i_min=1;
        j_min=1;
        i_max=local[0]-padding[0];
        j_max=local[1];
    }
    else if(south<0 && north>=0 && east>=0 && west>=0){ //middle south borders
        i_min=1;
        j_min=1;
        i_max=local[0]-padding[0];
        j_max=local[1]+1;
    }
    else if(south<0 && west<0 && north>=0 && east>=0){ //south west border
        i_min=1;
        j_min=2;
        i_max=local[0]-padding[0];
        j_max=local[1]+1;
    }
}

MPI_Barrier(MPI_COMM_WORLD);

```

```
//*****
```

Τώρα, υλοποιούμε το computational core του προγράμματος. Κάθε διεργασία επικοινωνεί με γειτονικές διεργασίες και ανταλλάζουν τα περιμετρικά του rows και columns. Εδώ, γίνεται και έλεγχος για αρνητικά ή μη αρνητικά ranks γειτόνων, όπως εξηγήσαμε και νωρίτερα και, αναλόγως, η κάθε διεργασία στέλνει και λαμβάνει τα κατάλληλα δεδομένα στους κατάλληλους γείτονες. Πραγματοποιούνται αποστολές και λήψεις μηνυμάτων μέσω των εντολών MPI_Isend και MPI_Irecv και, στο τέλος, προσθέτουμε την εντολή MPI_Waitall, ώστε οι διεργασίες να περιμένουν μέχρι να ολοκληρωθούν όλες οι ανταλλαγές μηνυμάτων. Έπειτα, κάθε διεργασία, διαθέτοντας πλέον όλα τα δεδομένα που χρειάζεται, καλεί τη συνάρτηση Jacobi().

Προσθέτουμε και τους κατάλληλους timers, ώστε να λάβουμε μετρήσεις για το χρόνο εκτέλεσης του αλγορίθμου.

```
//----Computational core----//
gettimeofday(&ttc, 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
    //*****TODO*****
    /*Fill your code here*/
    /*Compute and Communicate*/
    /*Add appropriate timers for computation*/
    MPI_Request requests[8];
    int counter = 0;

    if(north>=0){

        MPI_Isend(*u_current+local[1]+3,1,row,north,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;
        MPI_Irecv(*u_current+1,1,row,north,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;
    }
    if(south>=0){

MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;

MPI_Irecv(*u_current+((local[1]+2)*(local[0]+1))+1,1,row,south,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;
    }
    if(east>=0){

        MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;
        MPI_Irecv(*u_current+2*(local[1]+1)+1,1,column,east,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;
    }
    if(west>=0){

MPI_Isend(*u_current+local[1]+3,1,column,west,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;

MPI_Irecv(*u_current+local[1]+2,1,column,west,0,MPI_COMM_WORLD,&requests[counter]);
        counter+=1;
    }

    MPI_Waitall(8, requests, MPI_STATUS_IGNORE);
}
#endif
```

```

MPI_Status status[8];
MPI_Waitall(counter,requests,status);

swap=u_previous;
u_previous=u_current;
u_current=swap;

gettimeofday(&tcs,NULL);
Jacobi(u_previous,u_current,i_min,i_max,j_min,j_max);
gettimeofday(&tcf,NULL);
tcomp+=(tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;

```

Έπειτα, συμπληρώνουμε και τον κώδικα, με τον οποίο πραγματοποιείται ο έλεγχος σύγκλισης. Πρέπει να ελέγχουμε ότι ο αλγόριθμος συγκλίνει για όλες τις MPI διεργασίες και, γι' αυτό το λόγο, χρησιμοποιούμε την εντολή MPI_Allreduce().

```

#ifndef TEST_CONV
if (t%c==0) {
    //*****TODO*****
    /*Test convergence*/
    converged=converge(u_previous,u_current,i_min,i_max-1,j_min,j_max-1);
    MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);
}
#endif
//*****

```

Στο τέλος, η διεργασία με rank 0 αναλαμβάνει να συλλέξει τα μερικά αποτελέσματα όλων των διεργασιών και να συνθέσει το συνολικό global πίνακα U με τα τελικά αποτελέσματα.

```

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);

//---Rank 0 gathers local matrices back to the global matrix---//
if (rank==0) {
    U=allocate2d(global_padded[0],global_padded[1]);
}

//*****TODO*****
/*fill your code here*/
MPI_Gatherv(*u_current+local[1]+3,1,local_block,*U,scattercounts,scatteroffset,global_block,0,CART_COMM);
//*****

```

Μέθοδος Gauss – Seidel SOR

Αν και η μέθοδος Jacobi προσφέρει λύση στο πρόβλημα της διάδοσης θερμότητας σε δύο διαστάσεις, συγκλίνει με πολύ αργό ρυθμό. Γι' αυτό το λόγο, αποφασίζουμε να χρησιμοποιήσουμε τη μέθοδο Gauss – Seidel SOR, η οποία συγκλίνει πολύ πιο γρήγορα από τη μέθοδο Jacobi. Η μέθοδος Gauss – Seidel SOR υπολογίζει τη θερμότητα σε ένα σημείο του πλέγματος, χρησιμοποιώντας την παρακάτω εξίσωση:

$$u_{x,y}^{t+1} = u_{x,y}^t + \omega \cdot \frac{u_{x-1,y}^{t+1} + u_{x,y-1}^{t+1} + u_{x+1,y}^t + u_{x,y+1}^t}{4}, \quad \omega \in (0,2)$$

Σημειώνουμε ότι όταν υπολογίζουμε τη θερμότητα, τότε έχει προηγηθεί και ολοκληρωθεί ήδη ο υπολογισμός των θερμοτήτων και . Η μέθοδος Gauss – Seidel SOR εκμεταλλεύεται αυτό το γεγονός και, σε αντίθεση με τη μέθοδο Jacobi, όταν έρχεται η στιγμή που πρέπει να υπολογιστεί η θερμότητα, τότε χρησιμοποιεί τις ανανεωμένες τιμές και και όχι τις θερμότητες της προηγούμενης χρονικής στιγμής. Έτσι, επιτυγχάνεται σύγκλιση με πιο γρήγορο ρυθμό.

Όπως και προηγουμένως, έτσι και τώρα, θα παραλληλοποιήσουμε τη μέθοδος Gauss – Seidel SOR με τη βοήθεια του αρχείου `mpi_skeleton.c`. Ο κώδικας που γράφουμε έχει πολλές ομοιότητες με αυτόν που αναλύσαμε νωρίτερα για τη μέθοδο Jacobi. Γι' αυτό το λόγο, θα παραθέσουμε και θα αναλύσουμε μονάχα τα τμήματα κώδικα που χρειάστηκε να ανανεώσουμε.

Αρχικά, στην αρχή του αρχείου τοποθετούμε τη συνάρτηση `GaussSeidel()`, η οποία υλοποιεί την εξίσωση της μεθόδου Gauss – Seidel SOR.

```
void GaussSeidel(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*u_previous[i][j])*omega/4.0;
}
```

Έπειτα, πρέπει να τροποποιήσουμε τον κώδικα που αφορά το computational core της εφαρμογής μας. Πιο συγκεκριμένα, όπως εξηγήσαμε, στη μέθοδο Gauss – Seidel SOR, οι θερμότητες του βόρειου και του δυτικού στοιχείου λαμβάνονται από την τρέχουσα και όχι από την προηγούμενη χρονική στιγμή. Για αυτό το λόγο, πραγματοποιούμε τις παρακάτω αλλαγές:

```
//----Computational core----/
gettimeofday(&tts, NULL);
#ifndef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
#define T 256
for (t=0;t<T;t++) {
#endif

//*****TODO*****
/*Fill your code here*/
/*Compute and Communicate*/
/*Add appropriate timers for computation*/
MPI_Request requests[8];
int counter = 0;

if(north>=0){
    MPI_Isend(*u_current+local[1]+3,1,row,north,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
    MPI_Irecv(*u_current+1,1,row,north,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
}
if(south>=0){

MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;

MPI_Irecv(*u_current+((local[1]+2)*(local[0]+1))+1,1,row,south,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
}
if(east>=0){

MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;

MPI_Irecv(*u_current+2*(local[1]+1)+1,1,column,east,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
}
if(west>=0){
    MPI_Isend(*u_current+local[1]+3,1,column,west,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
    MPI_Irecv(*u_current+local[1]+2,1,column,west,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
}

MPI_Status status[8];
MPI_Waitall(counter,requests,status);
```

```

swap=u_previous;
u_previous=u_current;
u_current=swap;

MPI_Request RecRequests[8];
int RecCounter = 0;

if(north>=0){
    MPI_Recv(*u_current+1,1,row,north,0,MPI_COMM_WORLD,&RecRequests[RecCounter]);
    RecCounter+=1;
}

if(west>=0){

MPI_Recv(*u_current+local[1]+2,1,column,west,0,MPI_COMM_WORLD,&RecRequests[RecCounter]);
    RecCounter+=1;
}

MPI_Waitall(RecCounter,RecRequests,status);

gettimeofday(&tcs,NULL);

GaussSeidel(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;

MPI_Request SendRequests[8];
int SendCounter = 0;

if(south>=0){

MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,MPI_COMM_WORLD,&SendRequests[SendCounter]);
    SendCounter+=1;
}
if(east>=0){

MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&SendRequests[SendCounter]);
    SendCounter+=1;
}

#ifndef TEST_CONV
if (t%C==0) {
    //*****TODO*****
    /*Test convergence*/
    converged=converge(u_previous,u_current,i_min,i_max-1,j_min,j_max-1);
    MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);
}
#endif
MPI_Waitall(SendCounter,SendRequests,status);
//*****
}

```

Μέθοδος Red – Black SOR

Σκοπός μας είναι να επιτύχουμε ακόμη ταχύτερο ρυθμό σύγκλισης. Γι' αυτό το λόγο, θα πειραματιστούμε και με τη μέθοδο Red – Black SOR.

Η μέθοδος Red – Black SOR χωρίζει τα στοιχεία του πλέγματος σε άρτια ($i + j \bmod 2 = 0$) (red) και σε περιττά ($i + j \bmod 2 = 1$) (black). Έπειτα, πραγματοποιεί δύο φάσεις υπολογισμών:

$$u_{x,y}^{t+1} = u_{x,y}^t + \omega \cdot \frac{u_{x-1,y}^t + u_{x,y-1}^t + u_{x+1,y}^t + u_{x,y+1}^t - 4u_{x,y}^t}{4}, \quad \omega \in (0,2) \text{ when } (x+y) \bmod 2 = 0$$

και

$$u_{x,y}^{t+1} = u_{x,y}^t + \omega \cdot \frac{u_{x-1,y}^{t+1} + u_{x,y-1}^{t+1} + u_{x+1,y}^{t+1} + u_{x,y+1}^{t+1} - 4u_{x,y}^t}{4}, \quad \omega \in (0,2) \text{ when } (x+y) \bmod 2 = 1$$

Παραλληλοποιούμε και αυτή τη μέθοδο με τη βοήθεια του αρχείου `mpi_skeleton.c`.

Αρχικά, ορίζουμε τις δύο συναρτήσεις που υλοποιούν τις red και black φάσεις της μεθόδου.

```
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*u_previous[i][j]);
}

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*u_previous[i][j]);
}
```

Έπειτα, πρέπει να τροποποιήσουμε και το computational core της εφαρμογής μας. Καθώς η `RedSOR()` απαιτεί τιμές της προηγούμενης χρονικής στιγμής, θα καλείται αμέσως μετά τις αρχικές επικοινωνίες, όπως συνέβαινε και με τη μέθοδο Jacobi. Αμέσως μετά, οι διεργασίες πρέπει να ανταλλάξουν ξανά μηνύματα, ώστε όλι να διαθέτουν τις τρέχουσες τιμές Θερμότητας. Μόλις ολοκληρωθούν οι ανταλλαγές μηνυμάτων, κάθε διεργασία μπορεί να καλέσει τη συνάρτηση `BlackSOR()`. Παρακάτω φαίνεται ο τροποποιημένος κώδικας.

```
//----Computational core----//
gettimeofday(&pts, NULL);
#ifndef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
#define T 256
for (t=0;t<T;t++) {
#endif

//*****TODO*****
/*Fill your code here*/
/*Compute and Communicate*/
/*Add appropriate timers for computation*/
MPI_Request requests[8];
int counter = 0;

if(north>=0){
    MPI_Isend(*u_current+local[1]+3,1,row,north,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
    MPI_Irecv(*u_current+1,1,row,north,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
}
if(south>=0){

MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,MPI_COMM_WORLD,&requests[counter]);
counter+=1;

MPI_Irecv(*u_current+((local[1]+2)*(local[0]+1))+1,1,row,south,0,MPI_COMM_WORLD,&requests[counter]);
counter+=1;
}
if(east>=0){

MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&requests[counter]);
counter+=1;

MPI_Irecv(*u_current+2*(local[1]+1)+1,1,column,east,0,MPI_COMM_WORLD,&requests[counter]);
counter+=1;
}
if(west>=0{
    MPI_Isend(*u_current+local[1]+3,1,column,west,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
    MPI_Irecv(*u_current+local[1]+2,1,column,west,0,MPI_COMM_WORLD,&requests[counter]);
    counter+=1;
}

MPI_Status status[8];
MPI_Waitall(counter,requests,status);
```

```

swap=u_previous;
u_previous=u_current;
u_current=swap;

gettimeofday(&tcs,NULL);
RedSOR(u_previous,u_current,i_min,i_max,j_min,j_max,omega);
gettimeofday(&tcf,NULL);
tcomp+=(tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
MPI_Request Brequests[8];
int Bcounter = 0;

if(north>=0){
    MPI_Isend(*u_current+local[1]+3,1,row,north,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;
    MPI_Irecv(*u_current+1,1,row,north,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;
}
if(south>=0){

MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;

MPI_Irecv(*u_current+((local[1]+2)*(local[0]+1))+1,1,row,south,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
);
    Bcounter+=1;
}
if(east>=0){

MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;

MPI_Irecv(*u_current+2*(local[1]+1)+1,1,column,east,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;
}
if(west>=0){

MPI_Isend(*u_current+local[1]+3,1,column,west,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;

MPI_Irecv(*u_current+local[1]+2,1,column,west,0,MPI_COMM_WORLD,&Brequests[Bcounter]);
    Bcounter+=1;
}

MPI_Waitall(Bcounter,Brequests,status);
gettimeofday(&tcs,NULL);
BlackSOR(u_previous,u_current,i_min,i_max,j_min,j_max,omega);
gettimeofday(&tcf,NULL);
tcomp+=(tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;

#ifndef TEST_CONV
if (t%C==0) {
    //*****TODO*****
    /*Test convergence*/
    converged=converge(u_previous,u_current,i_min,i_max-1,j_min,j_max-1);
    MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);

}
#endif
//*****
}

```

Μετρήσεις με έλεγχο σύγκλισης

Θα εκτελέσουμε και τους τρεις αλγόριθμους και θα ενεργοποιήσουμε τον έλεγχο σύγκλισης. Θα χρησιμοποιήσουμε πλέγμα μεγέθους 512x512 και 64 MPI διεργασίες.

Εκτελούμε κάθε έναν αλγόριθμο 3 φορές. Οι μετρήσεις που παρουσιάζονται, για κάθε αλγόριθμο, προκύπτουν από το μέσο όρο των τριών αυτών πειραμάτων.

Αρχικά, σχεδιάζουμε τον παρακάτω πίνακα, στον οποίο φαίνεται πόσες επαναλήψεις απαιτούνται προκειμένου να συγκλίνει και, τελικά, να τερματίσει κάθε αλγόριθμος:

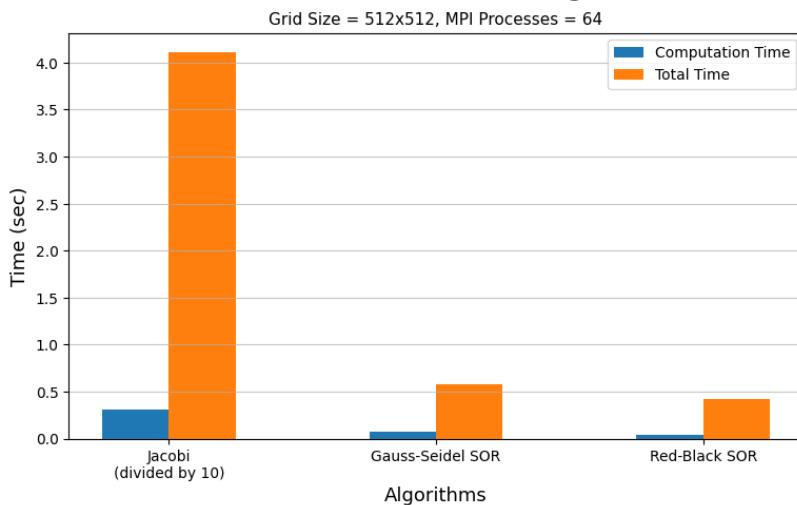
Αλγόριθμος	Πλήθος επαναλήψεων μέχρι να επιτευχθεί σύγκλιση
Jacobi	236001
Gauss – Seidel SOR	1501
Red – Black SOR	1201

Παρατηρούμε ότι η μέθοδος Jacobi απαιτεί τεράστιο πλήθος επαναλήψεων μέχρι να συγκλίνει και, άρα, πράγματι, παρουσιάζει πάρα πολύ αργό ρυθμό σύγκλισης. Έπειτα, η μέθοδος Gauss – Seidel SOR χρειάζεται αισθητά λιγότερες επαναλήψεις για να συγκλίνει. Συνεπώς, όπως αναμέναμε και από τη θεωρία η μέθοδος Gauss – Seidel SOR συγκλίνει πιο γρήγορα από τη μέθοδο Jacobi. Με ακόμη ταχύτερο ρυθμό, όμως, συγκλίνει η μέθοδος Red – Black SOR. Η Red – Black SOR απαιτεί τις λιγότερες επαναλήψεις από όλες τις μεθόδους για να συγκλίνει.

Στη συνέχεια, δημιουργούμε και το παρακάτω διάγραμμα, όπου φαίνονται οι χρόνοι εκτέλεσης κάθε αλγορίθμου. Παρουσιάζουμε δύο είδη μετρήσεων:

- *computation time*: ο χρόνος που απαιτείται για τον υπολογισμό των στοιχείων βάσει της εξίσωσης κάθε αλγορίθμου και
- *total time*: ο χρόνος που απαιτείται για τις επικοινωνίες μεταξύ των διεργασιών και για την εκτέλεση των υπόλοιπων εργασιών που εμφανίζονται στον κώδικα.

2D Heat Transfer Problem with Convergence Checks



Όπως βλέπουμε, η μέθοδος Jacobi παρουσιάζει το μεγαλύτερο χρόνο εκτέλεσης και, γι' αυτό το λόγο, χρειάστηκε να διαιρέσουμε το χρόνο εκτέλεσής της δια 10, ώστε να μπορούν να αναπαρασταθούν όλες οι μετρήσεις στο ίδιο διάγραμμα. Έπειτα, παρατηρούμε ότι η μέθοδος Gauss – Seidel SOR είναι ελαφρώς πιο χρονοβόρα από τη μέθοδο Red – Black SOR. Εάν και οι μέθοδοι Gauss – Seidel SOR και Red – Black SOR απαιτούν περισσότερες ανταλλαγές μηνυμάτων απ' ότι η μέθοδος Jacobi, παρουσιάζουν τόσο μικρότερο ρυθμό σύγκλισης που, τελικά, εκτελούνται πολύ ταχύτερα από τη μέθοδο Jacobi.

Επίσης αξιοσημείωτο είναι το γεγονός ότι και στις τρεις μεθόδους το μεγαλύτερο ποσοστό του χρόνου εκτέλεσης δε δαπανάται στην εκτέλεση του computation core. Αντιθέτως, φαίνεται πως η επικοινωνία και η μεταφορά δεδομένων μεταξύ των διεργασιών είναι αυτές που επιβαρύνουν σημαντικά την εκτέλεση και των τριών αλγορίθμων. Υπενθυμίζουμε ότι κατά τον έλεγχο σύγκλισης, σε κάθε επανάληψη, πραγματοποιείται MPI_Allreduce() μεταξύ

των διεργασιών, μία διαδικασία πολύ χρονοβόρα, η οποία οφείλεται, εν μέρει, για το μικρό ποσοστό που λαμβάνει το computation time επί του συνολικού χρόνο εκτέλεσης.

Τελικά, για την επίλυση του προβλήματος σε ένα σύστημα κατανεμημένης μνήμης θα επιλέγαμε τη μέθοδο Red – Black SOR, η οποία παρουσιάζει το μικρότερο ρυθμό σύγκλισης, αλλά και το μικρότερο χρόνο εκτέλεσης.

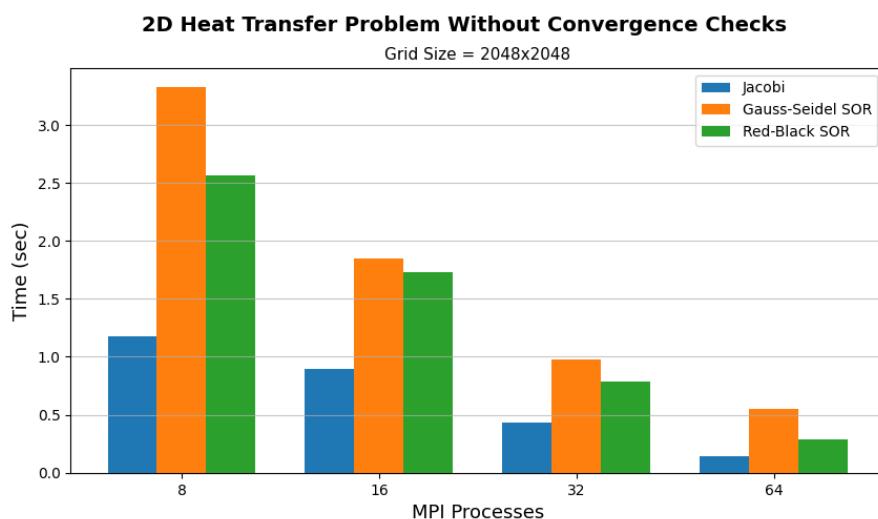
Μετρήσεις χωρίς έλεγχο σύγκλισης

Τώρα, θα απενεργοποιήσουμε τον έλεγχο σύγκλισης και θα εκτελέσουμε κάθε έναν αλγόριθμο για σταθερό αριθμό επαναλήψεων $T = 256$. Θα συλλέξουμε μετρήσεις για μεγέθη πίνακα 2048x2048, 4096x4096 και 6144x6144 και για πλήθος MPI διεργασιών 1, 2, 4, 8, 16, 32 και 64. Όπως και προηγουμένως, έτσι και τώρα, θα εκτελέσουμε 3 φορές κάθε αλγόριθμο και για κάθε συνδυασμό των configurations. Οι μετρήσεις που παρουσιάζουμε προκύπτουν από το μέσο όρο των 3 sets μετρήσεων που λαμβάνουμε.

Σημειώνουμε ότι αυτή τη φορά το computation time ήταν σχεδόν ίσο με το total time. Αυτό πιθανώς συμβαίνει, διότι τώρα απενεργοποιήσαμε τον έλεγχο σύγκλισης και, πλέον, δεν πραγματοποιούνται σε κάθε επανάληψη reductions από όλες τις διεργασίες. Άρα, πράγματι, στις μετρήσεις με έλεγχο σύγκλισης που συλλέξαμε προηγουμένως, το κόστος των MPI_Allreduce() επιβάρυναν σημαντικά την εκτέλεση των αλγορίθμων.

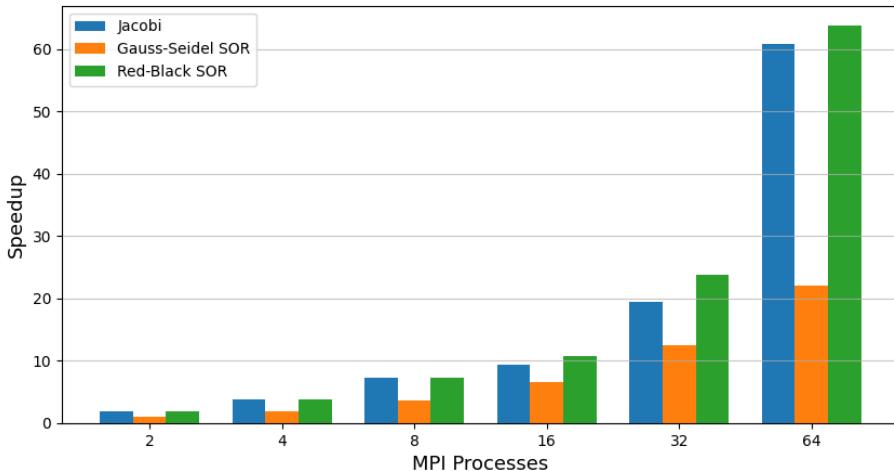
Δημιουργούμε διαγράμματα επιτάχυνσης και εκτέλεσης χρόνου για κάθε ένα μέγεθος πίνακα. Χρησιμοποιούμε μονάχα τις μετρήσεις του total time.

Πίνακας 2048x2048



2D Heat Transfer Problem Without Convergence Checks

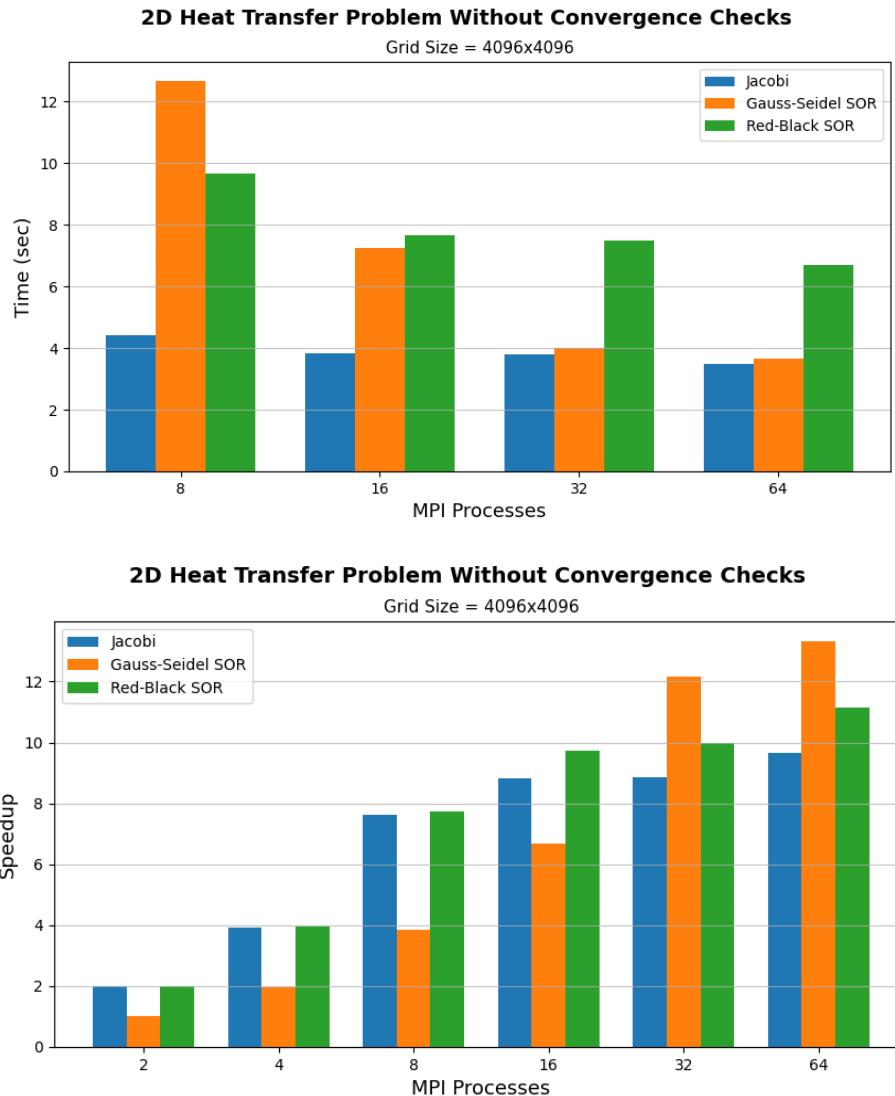
Grid Size = 2048x2048



Αρχικά, παρατηρούμε ότι, πλέον, η μέθοδος Jacobi εκτελείται ταχύτερα από τις άλλες δύο μεθόδους. Αυτό συμβαίνει, διότι απενεργοποιώντας τους ελέγχους σύγκλισης, επιβάλλουμε και στις τρεις μεθόδους το ίδιο πλήθος επαναλήψεων. Επομένως, οι μέθοδοι Gauss – Seidel SOR και Red – Black SOR δε διαθέτουν, πλέον, το πλεονέκτημα του ταχύτερου ρυθμού σύγκλισης έναντι της μεθόδου Jacobi. Η αυξημένη ανάγκη για επικοινωνία μεταξύ των διεργασιών στις μεθόδους Gauss – Seidel SOR και Red – Black SOR, τελικά, καθιστά τις μεθόδους πιο χρονοβόρες από τη μέθοδο Jacobi.

Αυτό που διαφοροποιεί το χρόνο εκτέλεσης κάθε μεθόδου είναι ο τρόπος με τον οποίο πραγματοποιούνται οι επικοινωνίες μεταξύ των διεργασιών. Πιο συγκεκριμένα, στη μέθοδο Jacobi, σε κάθε επανάληψη, όλες οι διεργασίες ανταλλάζουν αρχικά δεδομένα και μόλις ολοκληρωθεί αυτό το στάδιο επικοινωνιών, εισέρχονται όλες ταυτόχρονα στο στάδιο υπολογισμών και επίλυσης της εξίσωσης. Στη μέθοδο Red – Black SOR, αρχικά, όλες οι διεργασίες ανταλλάσσουν δεδομένα, έπειτα, εκτελούν την red φάση υπολογισμών, ύστερα, ανταλλάσσουν πάλι δεδομένα και, τέλος, εισέρχονται στη black φάση υπολογισμών. Ωστόσο, στη μέθοδο Gauss – Seidel SOR, παρατηρούνται εξαρτήσεις μεταξύ των διεργασιών που δεν επιτρέπουν συμμετρία μεταξύ φάσεων επικοινωνίας και φάσεων υπολογισμών ανάλογη με αυτήν που παρατηρήσαμε στις μεθόδους Jacobi και Red – Black SOR. Πιο συγκεκριμένα, στη μέθοδο Gauss – Seidel SOR, για τον υπολογισμό κάθε στοιχείου, χρειαζόμαστε το βόρειο και δυτικό γείτονα της τρέχουσας χρονικής στιγμής. Αυτό έχει ως συνέπεια κάθε διεργασία να πρέπει να περιμένει το βόρειο και δυτικό γείτονά της για να εισέλθει στη φάση των υπολογισμών. Και έπειτα, μόλις η ίδια διεργασία ολοκληρώσει τους υπολογισμούς της, πρέπει να στέλνει τα αποτελέσματά της σε κάθε νότιο και ανατολικό γείτονά της, προκειμένου αυτοί να μπορέσουν να ξεκινήσουν τους υπολογισμούς. Βλέπουμε, λοιπόν, ότι αναπτύσσονται εξαρτήσεις που περιορίζουν την παραλληλία του αλγορίθμου. Γι' αυτό το λόγο, οι μέθοδοι Jacobi και Red – Black SOR κλιμακώνουν καλύτερα και εμφανίζουν μεγαλύτερο speedup από τη μέθοδο Gauss – Seidel SOR.

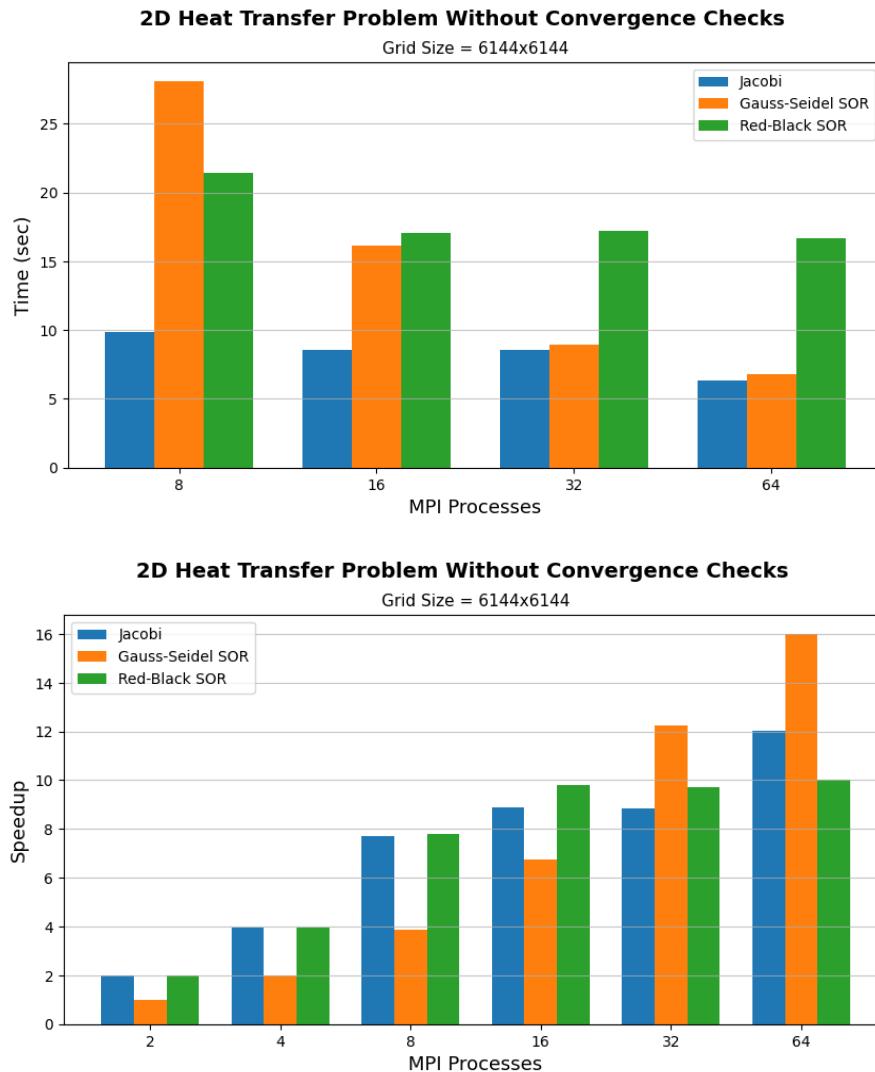
Πίνακας 4096x4096



Αρχικά, παρατηρούμε και, πάλι, ότι η μέθοδος Jacobi εκτελείται πιο γρήγορα από τις άλλες δύο μεθόδους για τους λόγους που έχουμε ήδη αναφέρει. Ωστόσο, σε αυτόν τον πίνακα, παρατηρούμε ότι οι μέθοδοι Jacobi και Red – Black SOR παύουν να κλιμακώνουν για 32 και 64 MPI διεργασίες. Συνεπώς, για 32 και 64 MPI διεργασίες, η μέθοδος Gauss – Seidel SOR εμφανίζει μεγαλύτερο speedup και εκτελείται και ταχύτερα από τις άλλες δύο μεθόδους. Η διαφορά αυτή οφείλεται στο μεγαλύτερο μέγεθος πίνακα και στις περισσότερες προσβάσεις στη μνήμη. Πιο συγκεκριμένα, όπως γνωρίζουμε, η ουρά parlab διαθέτει 8 κόμβους, με κάθε κόμβο να διαθέτει 2 επεξεργαστές και κάθε επεξεργαστή να διαθέτει 4 πυρήνες. Με αυτόν τον τρόπο, η ουρά parlab μπορεί να φιλοξενήσει το πολύ 64 MPI διεργασίες. Όταν εμείς εκτελούμε το πρόγραμμά μας για 32 και 64 MPI διεργασίες, τότε χρησιμοποιούμε μεγάλο πλήθος πυρήνων και, άρα, σε κάθε MPI διεργασία αναλογεί μικρότερο μέγεθος cache μνήμης. Όμως, τώρα, χρησιμοποιούμε μεγαλύτερο μέγεθος πίνακα, με αποτέλεσμα να παρουσιάζεται resource contention μεταξύ των πολλών MPI διεργασιών. Παρατηρούνται περισσότερα cache line updates, με αποτέλεσμα οι αλγόριθμοι Jacobi και Red – Black SOR να μην κλιμακώνουν. Αντιθέτως, η μέθοδος Gauss – Seidel SOR δε φαίνεται να επηρεάζεται τόσο από το φαινόμενο αυτό. Αυτό συμβαίνει, διότι, όπως εξηγήσαμε και νωρίτερα, οι διεργασίες

στη μέθοδο Gauss – Seidel SOR δεν εισέρχονται ταυτόχρονα στη φάση υπολογισμών, όπως συμβαίνει με τις μεθόδους Jacobi και Red – Black SOR. Συνεπώς, οι διεργασίες της μεθόδου Gauss – Seidel SOR δε θα προκαλούν ταυτόχρονα και πολλά cache line updates. Αυτός είναι ο λόγος για τον οποίο η μέθοδος Gauss – Seidel SOR εκτελείται ταχύτερα και με μεγαλύτερο speedup από τις άλλες δύο μεθόδους για 32 και 64 MPI διεργασίες.

Πίνακας 6144x6144



Οι παρατηρήσεις μας είναι ίδιες με τις παρατηρήσεις που καταγράψαμε για τον πίνακα 4096x4096.