ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Συστήματα Παράλληλης Επεξεργασίας
Ακ. έτος 2022-2023, 9ο Εξάμηνο

Εξαμηνιαία Εργασία

Ομάδα 8
Κωνσταντίνος Σιδέρης, Α.Μ.:03118134
Δέσποινα Μουσάδη, Α.Μ.:03118108
Ελισάβετ Λυδία Αλβανάκη, Α.Μ.:03118167

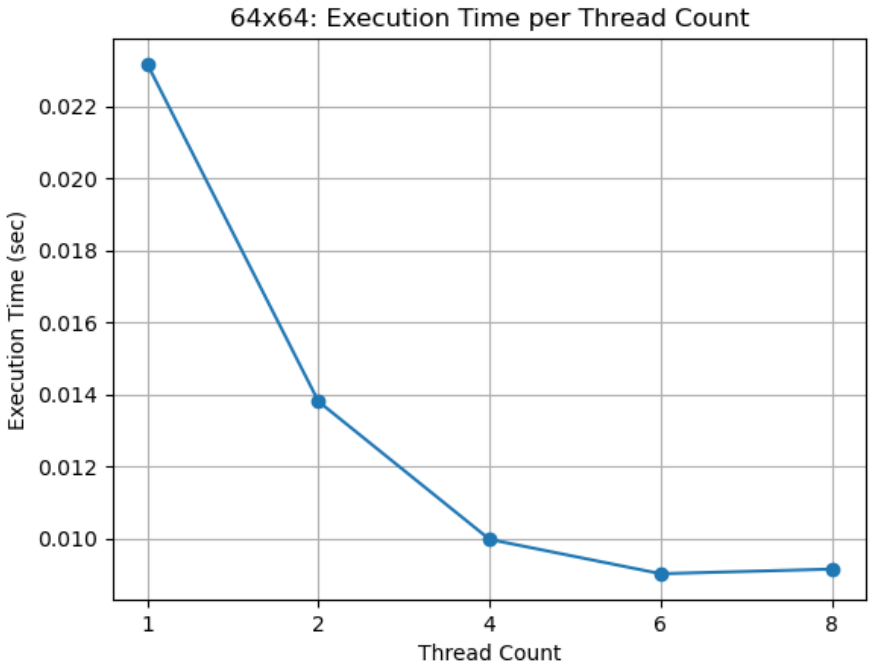
Περιεχόμενα

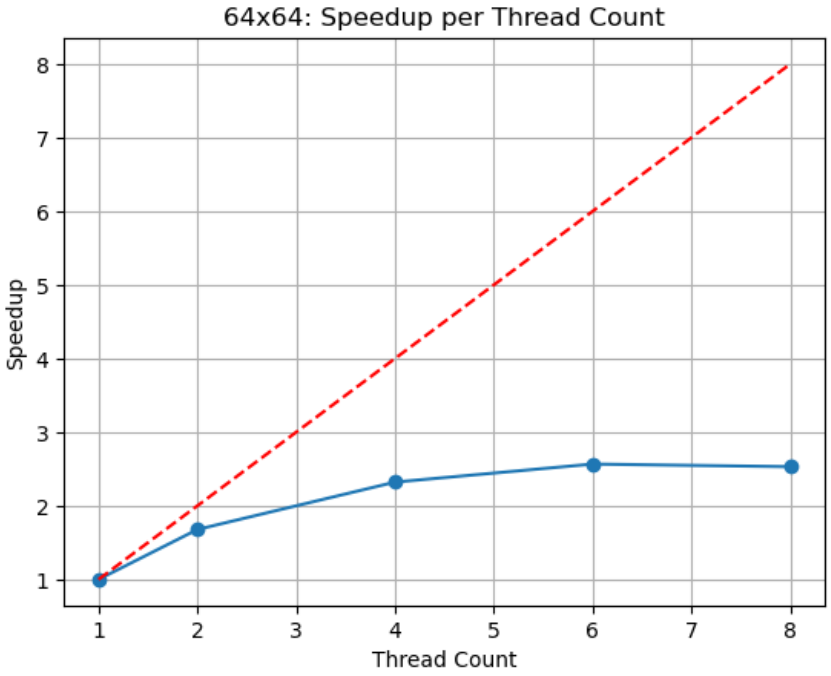
1 Εξοικείωση με το περιβάλλον προγραμματισμού

1.1 Σκοπός της Άσκησης

Σε αυτή την άσκηση καλούμαστε να παραλληλοποιήσουμε, σε αρχιτεκτονική κοινής μνήμης, το Παιχνίδι της Ζωής του Conway (Conway's Game of Life), με τη χρήση του OpenMP, ώστε να εξοικειωθούμε με τις υποδομές του εργαστηρίου. Λαμβάνουμε μετρήσεις επίδοσής για 1000 γενιές του παιχνιδιού χρησιμοποιώντας 1, 2, 4, 6 και 8 πυρήνες (νήματα) σε μεγέθη ταμπλό 64×64, 1024×1024 και 4096×4096. Ακολουθούν τα διαγράμματα χρόνου και speedup ως προς τους πυρήνες (νήματα) για όλα τα μεγέθη ταμπλό.

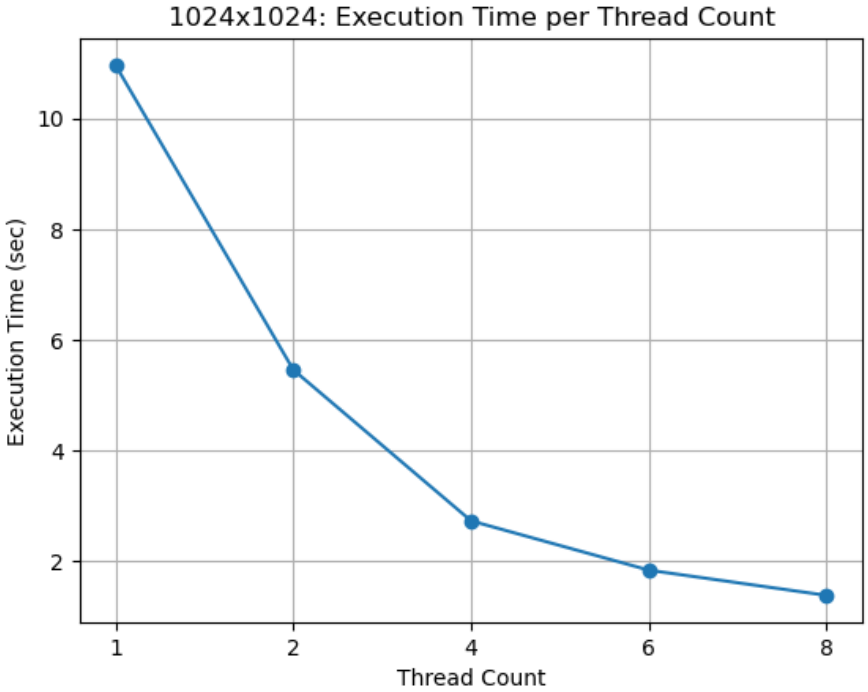
1.2 Ταμπλό 64x64

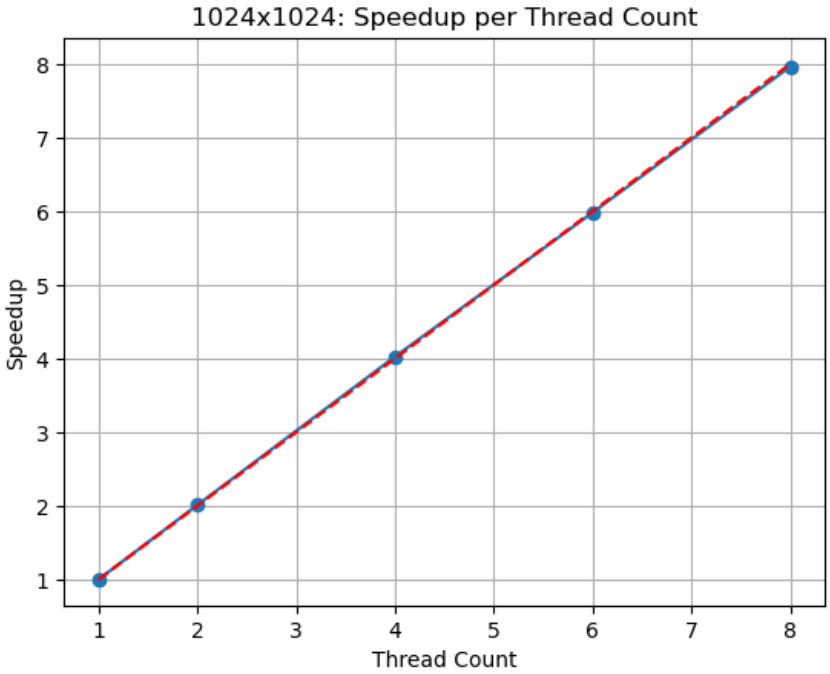




Παρατηρούμε ότι το speedup αυξάνεται σε μικρό βαθμό με την αύξηση των πυρήνων και σταθεροποιείται για 6 νήματα και άνω, είναι δηλαδή μη ιδανικό (typical speedup). Αυτό οφείλεται στο γεγονός ότι το μέγεθος ταμπλό δεν είναι αρκετά μεγάλο ώστε το κέρδος από την παραλληλοποίηση να υπερτερεί του διαχειριστικού κόστους των νημάτων και κατανομής εργασιών.

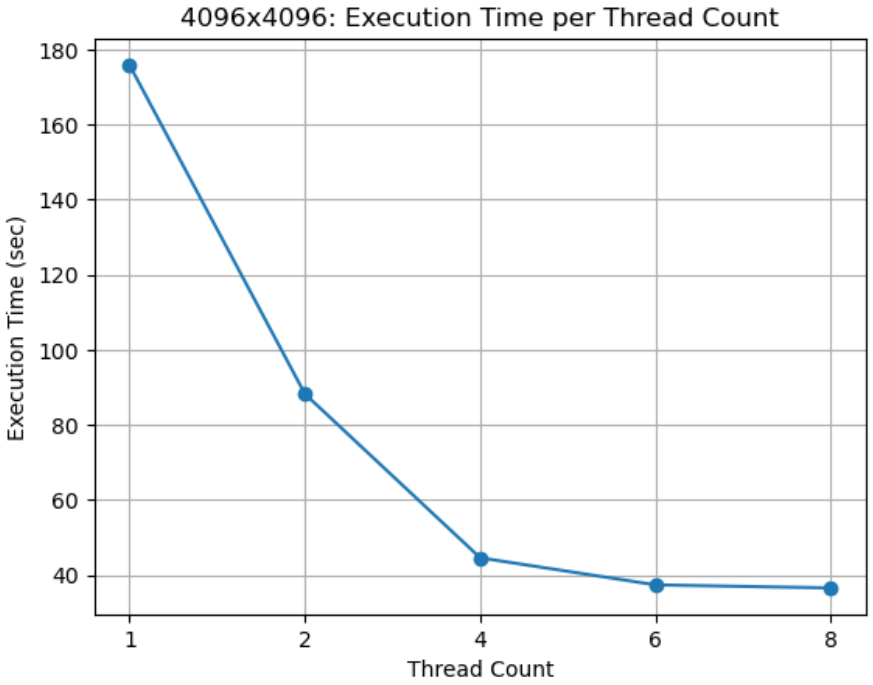
1.3 Ταμπλό 1024x1024

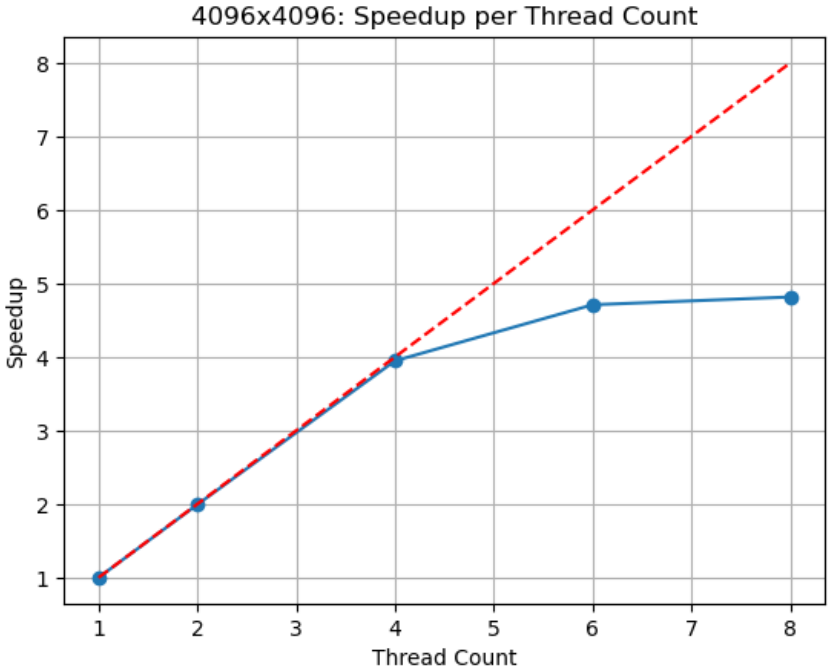




Σε αυτή την περίπτωση το speedup είναι ιδανικό (linear speedup) συνεπώς, συμπεραίνουμε ότι το μέγεθος ταμπλό 1024x1024 είναι επαρκώς μεγάλο ώστε το κέρδος από την παραλληλοποίηση να υπερτερεί του διαχειριστικού κόστους των νημάτων και κατανομής εργασιών καθώς και ότι τα δεδομένα που απαιτούνται κατά την εκτέλεση χωράνε με αποδοτικό τρόπο στην μνήμη cache.

1.4 Ταμπλό 4096x4096





Στην περίπτωση ταμπλό 4096×4096 παρατηρούμε ότι αρχικά το speedup είναι ιδανικό (linear speedup) ενώ στην συνέχεια η κλιμακωσιμότητα ”σπάει” (scalability break) για 4 νήματα και άνω. Καθώς εργαζόμαστε σε μοντέλο κοινού χώρου διευθύνσεων με ιεραρχία μνήμης, κάθε πυρήνας χρησιμοποιεί κοινά με άλλους πυρήνες blocks μνήμης. Δεδομένου ότι τα δεδομένα που επεξεργάζεται κάθε πυρήνας ενδέχεται να μην χωράνε σε ένα block μνήμης και ότι ο συνολικός αριθμός των blocks αυξάνεται λόγω της αύξησης του μεγέθους ταμπλό, δημιουργείται συμφόρηση στο διάδρομο μνήμης.

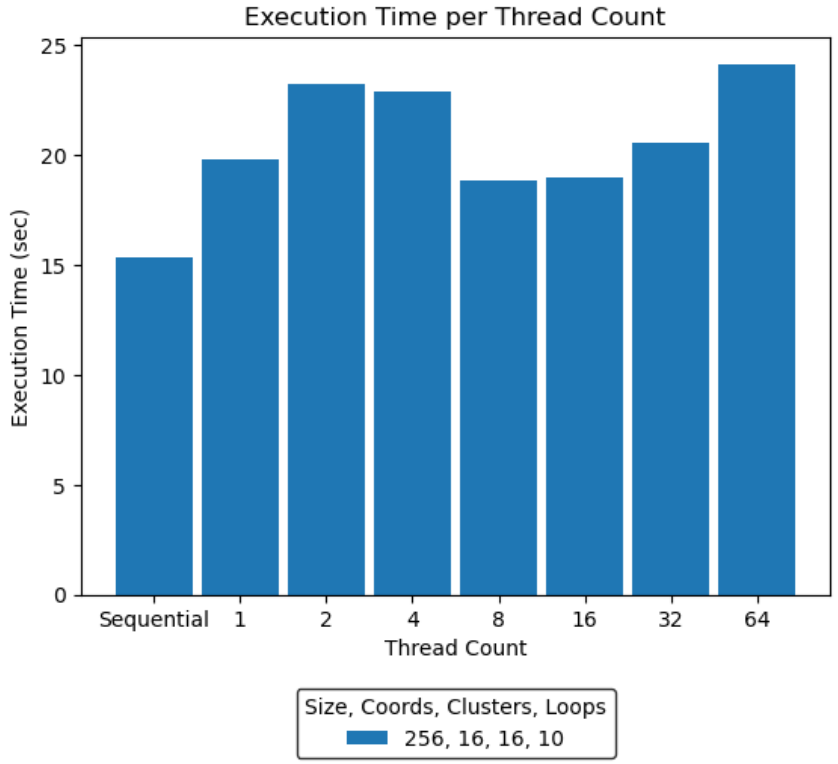
2 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

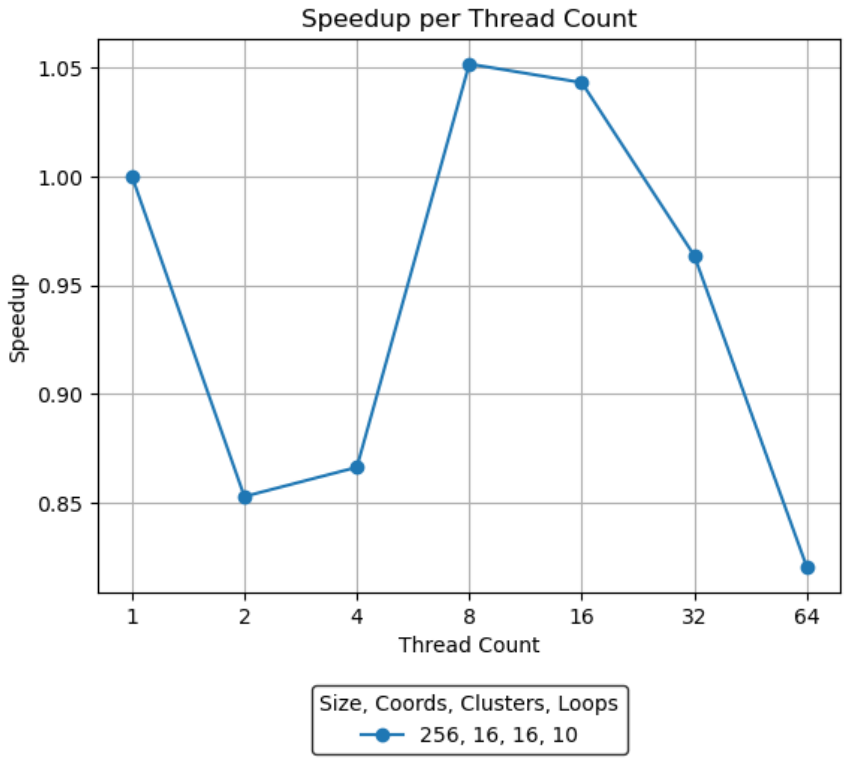
2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Σε αυτή την άσκηση παραλληλοποιούμε δύο εκδόσεις του αλγορίθμου K-means. Στην πρώτη έκδοση (Shared Clusters) τα δεδομένα είναι μοιραζόμενα δηλαδή, οι συστάδες (clusters) καταγράφονται σε κοινή για όλα τα νήματα δομή δεδομένων. Στην δεύτερη έκδοση (Copied Clusters) τα δεδομένα είναι αντεγραμμένα δηλαδή, οι συστάδες (clusters) καταγράφονται σε ξεχωριστή δομή δεδομένων για κάθε νήμα και στο τέλος γίνεται reduction των δεδομένων που έχουν συγκεντρωθεί. Ακολουθούν ο κώδικας και τα διαγράμματα χρόνου και speedup ως προς τα νήματα για τις δύο εκδόσεις.

2.1.1 Shared Clusters

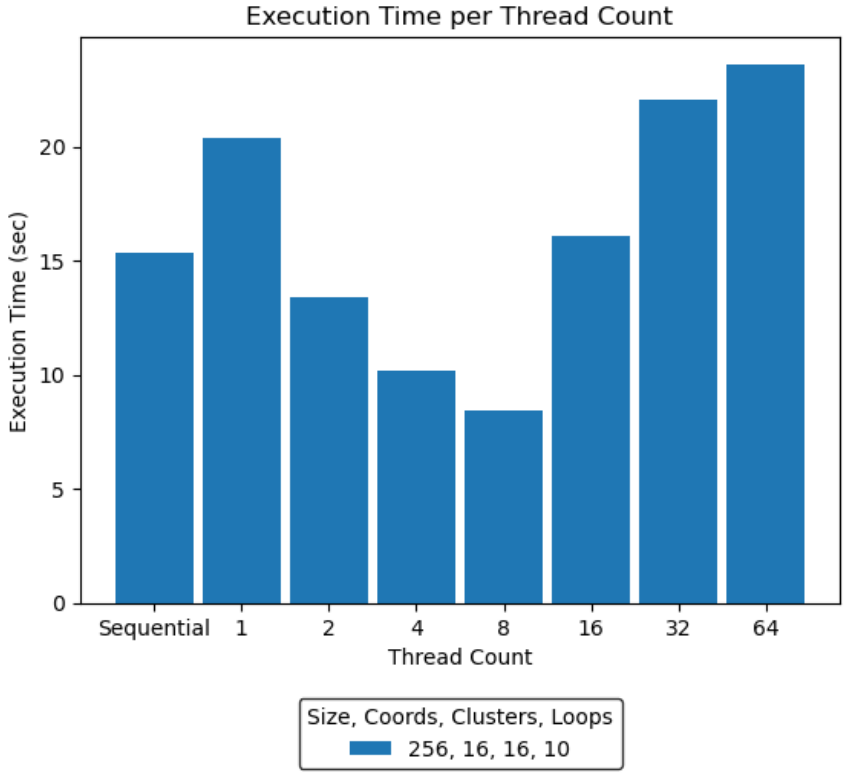
```
void kmeans(...) {

    ...

    do {

        ...

        delta = 0.0;

        /* DONE: Detect parallelizable region and use OpenMP pragmas */
        #pragma omp parallel for shared(newClusters, newClusterSize)
    private(i, j, index) reduction(+:delta)
        for (i=0; i<numObjs; i++) {

            ...

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

        ...

    } while (delta > threshold && loop < 10);

    ...

}
```

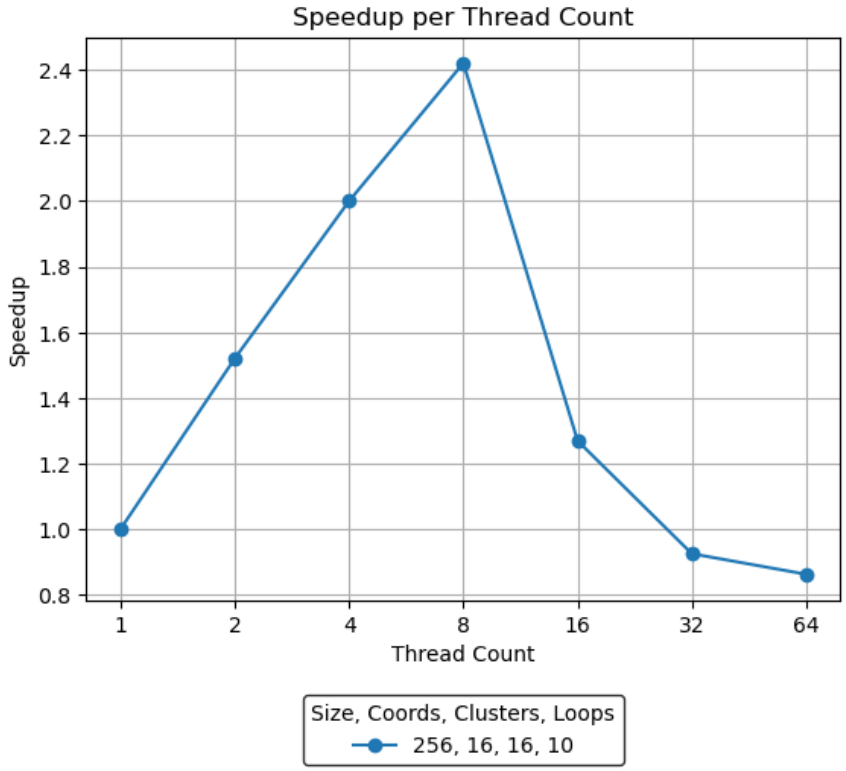




Παρατηρούμε ότι ο χρόνος εκτέλεσης είναι μεγαλύτερος σε σχέση με αυτόν του σειριακού αλγορίθμου ανεξάρτητα από τον αριθμό νημάτων. Για 8 και 16 νήματα υπάρχει μία μικρή βελτίωση στους χρόνους λόγω παραλληλισμού αλλά για περαιτέρω αύξηση των νημάτων αυξάνεται και ο χρόνος εκτέλεσης. Αυτό συμβαίνει καθώς, τα οφέλη του παραλληλισμού δεν υπερτερούν του χρονικού overhead που προκαλείται από την χρήση κλειδωμάτων για πρόσβαση στους διαμοιραζόμενους πίνακες newClusters και newClusterSize. Κάθε νήμα πριν γράψει σε μία θέση των πινάκων ”κλειδώνει” ολόκληρους τους πίνακες οπότε, αν πολλά νήματα προσπαθούν να γράψουν ταυτόχρονα εισάγεται σειριακότητα στο πρόγραμμα και η επίδοση μειώνεται.

Χρήση μεταβλητής περιβάλλοντος GOMP_CPU_AFFINITY

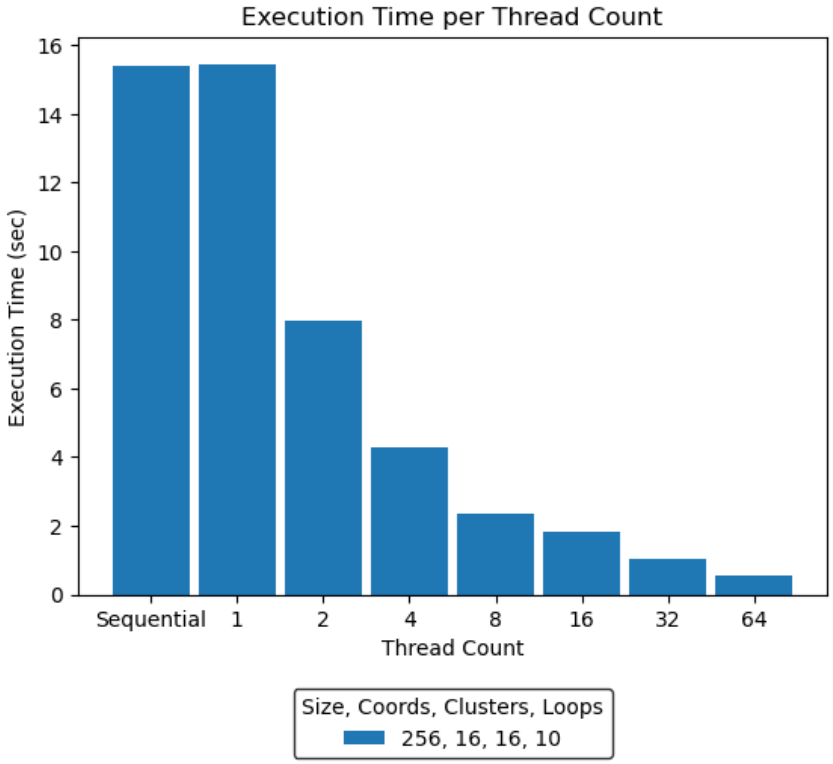


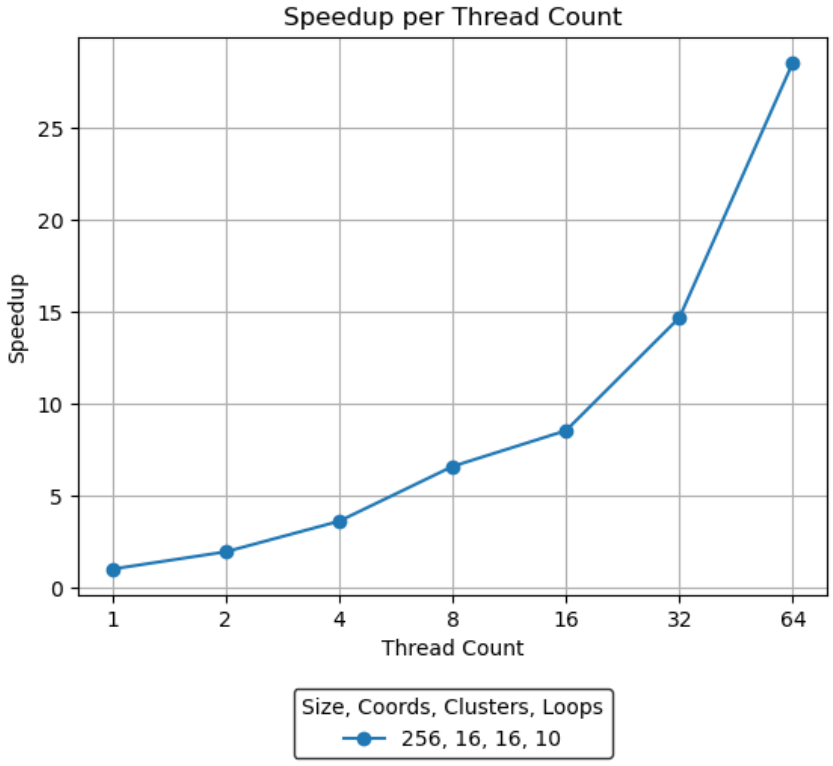


Με τη χρήση της μεταβλητής περιβάλλοντος GOMP_CPU_AFFINITY γίνεται πρόσδεση νημάτων σε συγκεκριμένα cores. Η επίδοση βελτιώνεται καθώς, ενισχύεται το locality αφού κάθε νήμα έχει την ίδια cache για κάθε iteration του for loop που εκτελεί. Συνεπώς, έχουμε περισσότερα cache hits από ότι θα είχαμε με τυχαίο allocation νημάτων σε πυρήνες κάθε φορά. Για 16 νήματα και άνω η επίδοση πάλι υποφέρει από την εισαγωγή σειριακότητας λόγω των κλειδωμάτων.

2.1.2 Copied Clusters

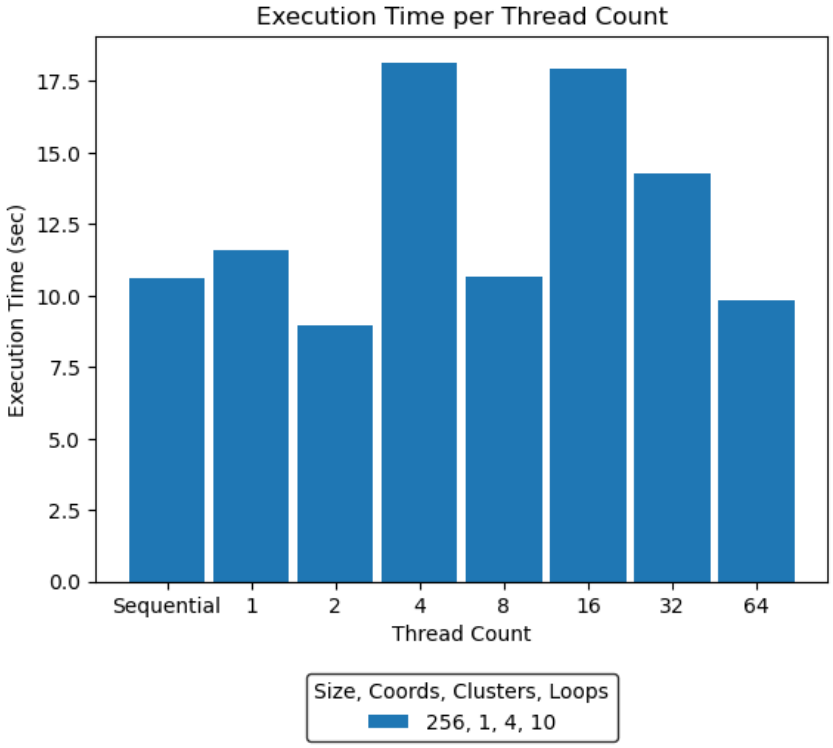
```
void kmeans(...) {

    ...

    do {

        ...

        /* DONE: Initiliaze local cluster data to zero (separate for each
↪   thread) */
        #pragma omp parallel for shared(local_newClusters,
↪   local_newClusterSize) private(i, j, k)
        for (k=0; k<nthreads; k++)
        {
            ...
        }

        #pragma omp parallel for shared(local_newClusters,
↪   local_newClusterSize) private(i, j, k, index) reduction(+:delta)
        for (i=0; i<numObjs; i++) {

            ...

            // update new cluster centers : sum of all objects within
↪   (average will be performed later)
            /* DONE: Collect cluster data in local to each thread arrays */
            k = omp_get_thread_num();
            local_newClusterSize[k][index]++;
            for (j=0; j<numCoords; j++)
                local_newClusters[k][index*numCoords + j] +=
↪   objects[i*numCoords + j];
        }

        /* DONE: Reduction of cluster data from local arrays to shared. */
        for (k=0; k<nthreads; k++)
        {
            for (i=0; i<numClusters; i++) {
                for (j=0; j<numCoords; j++)
                    newClusters[i*numCoords + j] +=
↪   local_newClusters[k][i*numCoords + j];
                newClusterSize[i] += local_newClusterSize[k][i];
            }
        }

        ...

    } while (delta > threshold && loop < 10);

    ...

}
```

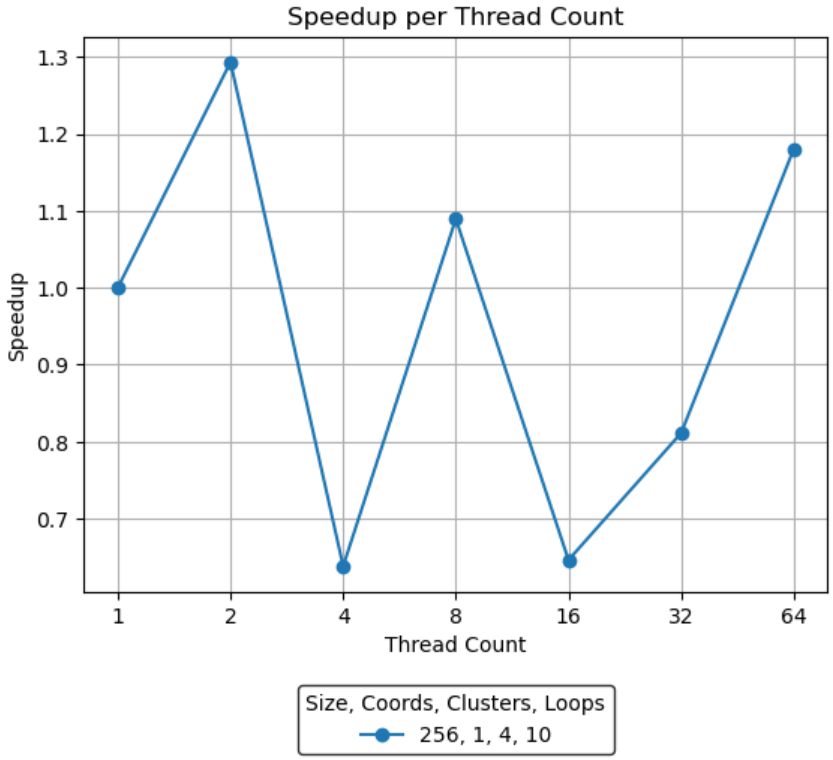




Η συγκεκριμένη έκδοση του αλγορίθμου επιτυγχάνει πολύ ικανοποιητικούς χρόνους και scaling σε σχέση με την έκδοση που χρησιμοποιεί shared clusters. Όπως έχει προαναφερθεί ο λόγος που η προηγούμενη έκδοση δεν κάνει καλό scaling είναι διότι το overhead από την χρήση κλειωμάτων είναι πολύ μεγάλο καθώς αυξάνεται ο αριθμός των νημάτων. Σε αυτή την έκδοση το κάθε νήμα γράφει σε μία διαφορετική σειρά των πινάκων local_newClusters και local_newClusterSize, δηλαδή ουσιαστικά το κάθε νήμα έχει ένα αντίγραφο των αντίστοιχων πινάκων newClusters και newClusterSize του προηγούμενου ερωτήματος. Συνεπώς, αφού δεν υπάρχει ανάγκη για την χρήση κλειδωμάτων επιλύεται το πρόβλημα της προηγούμενης έκδοσης.

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



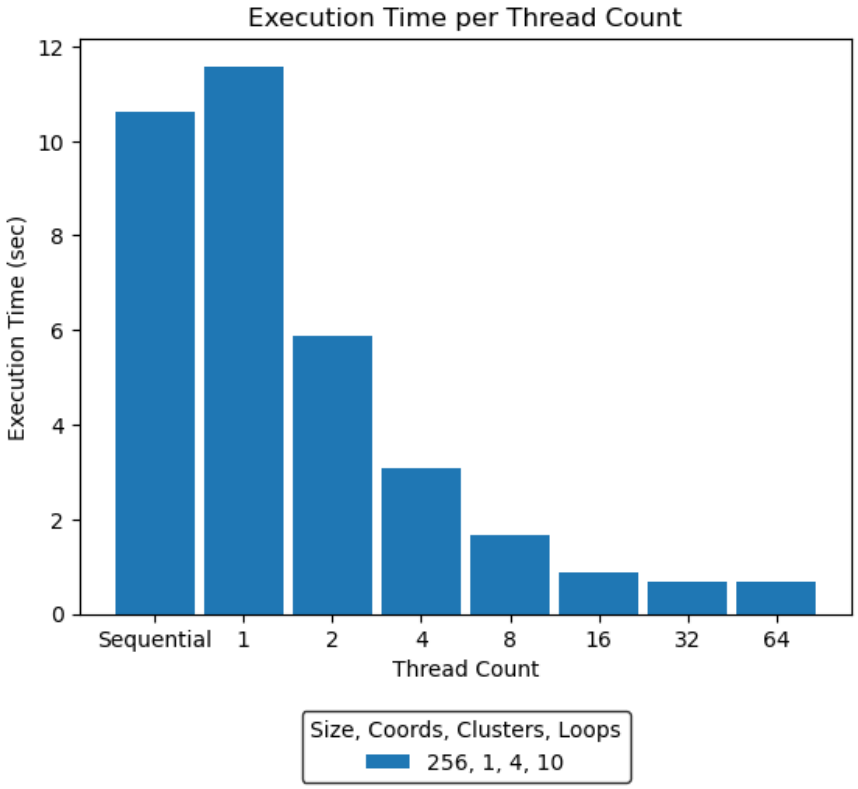


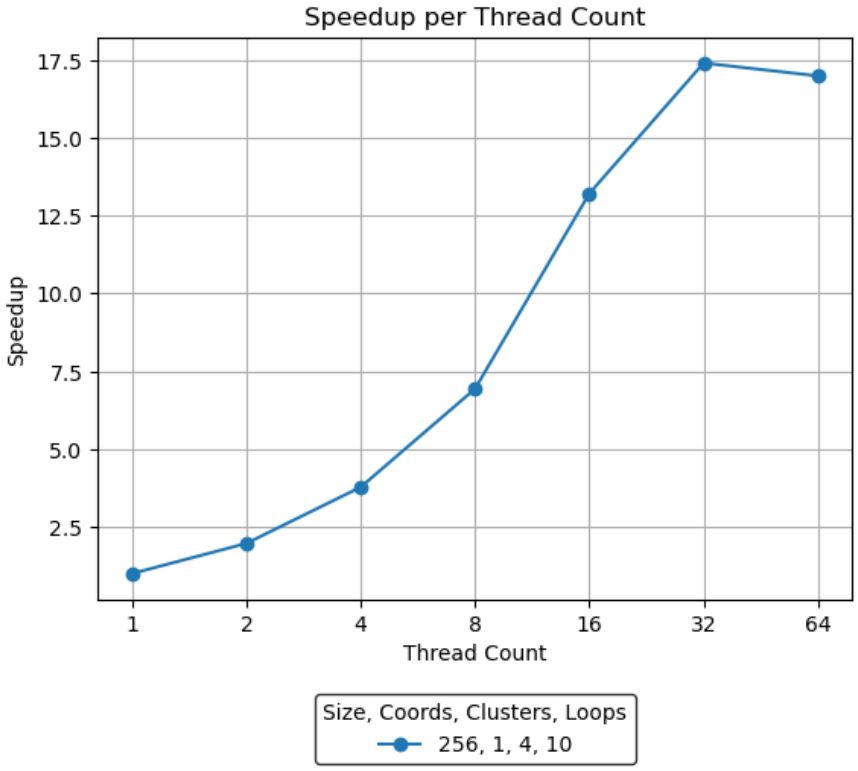
Για το configuration Size, Coords, Clusters, Loops = 256, 1, 4, 10 παρατηρούμε μη ικανοποιητικούς χρόνους και scaling, καθώς αφού έχουμε μικρότερο αριθμό clusters περισσότερες γραμμές των πινάκων local_newClusters και local_newClusterSize βρίσκονται στο ίδιο cache block. Συνεπώς, παρατηρείται το φαινόμενο False Sharing, όπου όταν ένα νήμα γράφει διαφορετικά δεδομένα από άλλα νήματα, τα οποία όμως βρίσκονται στο ίδιο cache block, τότε το block θα γίνει invalid για όλα τα άλλα νήματα και θα αναγκαστούν να το ξανά φορτώσουν ολόκληρο.

Βέλτιστη Υλοποίηση

```
void kmeans(...) {

    ...

    // Initialize local (per-thread) arrays (and later collect result on
↪   global arrays)
    #pragma omp parallel for shared(local_newClusters, local_newClusterSize)
↪   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));
    }

    timing = wtime();
    do {
        // before each loop, set cluster data to 0
        #pragma omp parallel for shared(newClusters, newClusterSize)
↪   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;

        /* DONE: Initiliaze local cluster data to zero (separate for each
↪   thread) */
        #pragma omp parallel for shared(local_newClusters,
↪   local_newClusterSize) private(i, j, k)
        for (k=0; k<nthreads; k++)
        {
            for (i=0; i<numClusters; i++) {
                for (j=0; j<numCoords; j++)
                    local_newClusters[k][i*numCoords + j] = 0.0;
                local_newClusterSize[k][i] = 0;
            }
        }

        #pragma omp parallel for shared(local_newClusters,
↪   local_newClusterSize) private(i, j, k, index) reduction(+:delta)
        for (i=0; i<numObjs; i++)
        {
            ...

            // update new cluster centers : sum of all objects located within
↪   (average will be performed later)
```

```
            /* DONE: Collect cluster data in local arrays (local to each
    thread) */
            k = omp_get_thread_num();
            local_newClusterSize[k][index]++;
            for (j=0; j<numCoords; j++)
                local_newClusters[k][index*numCoords + j] +=
    objects[i*numCoords + j];

        }

        /* DONE: Reduction of cluster data from local arrays to shared. */
        for (k=0; k<nthreads; k++)
        {
            for (i=0; i<numClusters; i++) {
                for (j=0; j<numCoords; j++)
                    newClusters[i*numCoords + j] +=
    local_newClusters[k][i*numCoords + j];
                newClusterSize[i] += local_newClusterSize[k][i];
            }
        }

        ...

    } while (delta > threshold && loop < loop_threshold);

    ...

}
```

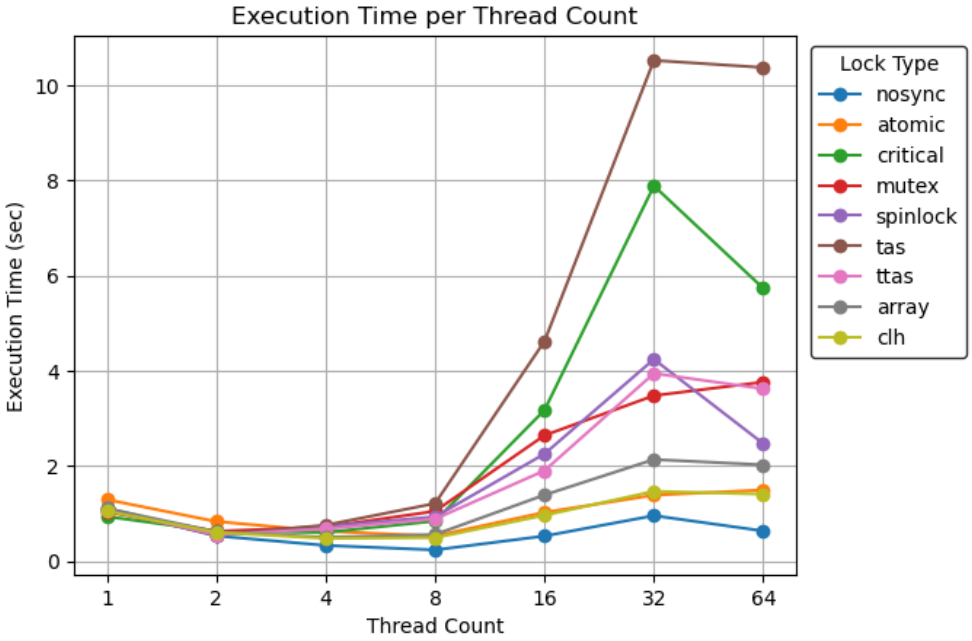


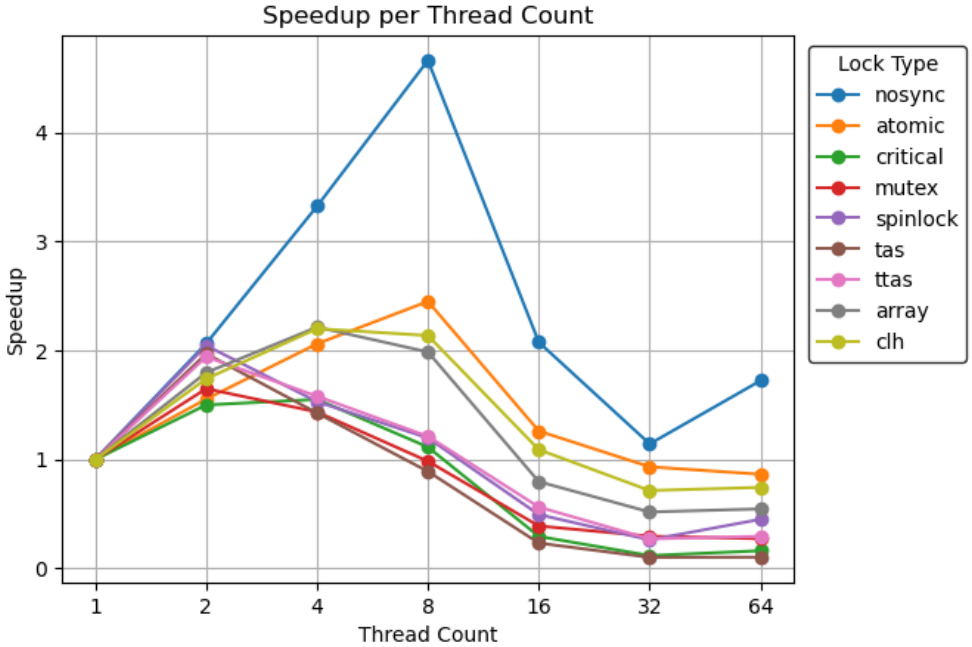


Παρατηρούμε ότι όταν κάθε νήμα κάνει calloc() τα μέρη των πινάκων local_newClusters και local_newClusterSize στα οποία γράφει έχουμε καλύτερο performance από όταν ένα νήμα κάνει calloc() ολόκληρους τους πίνακες. Αυτό συμβαίνει λόγω της πολιτικής first touch του Linux, σύμφωνα με την οποία, σε NUMA αρχιτεκτονικές, το memory allocation μίας σελίδας μνήμης γίνεται στο node στο οποίο βρίσκεται το thread που έχει πρόσβαη σε αυτή για πρώτη φορά. Έτσι, αν κάθε thread κάνει first touch στις σελίδες μνήμης όπου γράφει, διασφαλίζουμε ότι η εκχώρηση μνήμης γίνεται στη RAM του node που βρισκόμαστε, ελαχιστοποιώντας έτσι τα remote memory accesses (προσβάσεις σε RAM άλλου node) τα οποία είναι πιο χρονοβόρα. Οι καλύτεροι χρόνοι που πετυχαίνουμε είναι 0.6755sec (0.0675sec per loop) για το configuration Size, Coords, Clusters, Loops = 256, 1, 4, 10 με 32 threads και 0.4577sec (0.0458sec per loop) για το configuration Size, Coords, Clusters, Loops = 256, 16, 16, 10 με 64 threads.

2.1.3 Αμοιβαίος Αποκλεισμός - Κλειδώματα

Παρατίθενται τα συνολικά διαγράμματα χρόνου εκτέλεσης και speedup για τα ζητούμενα κλειδώματα:



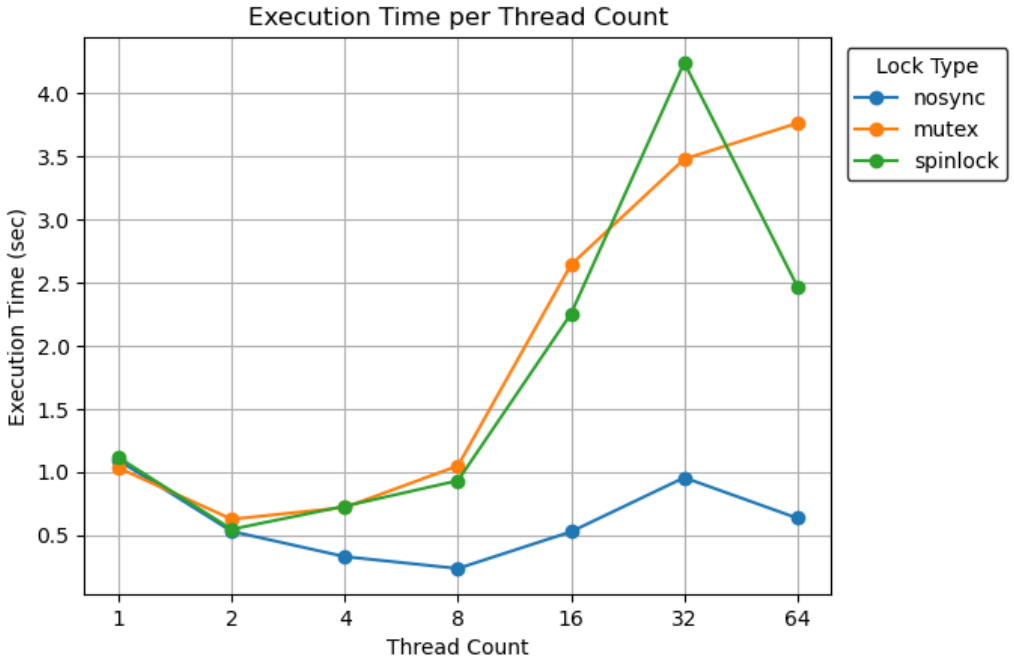


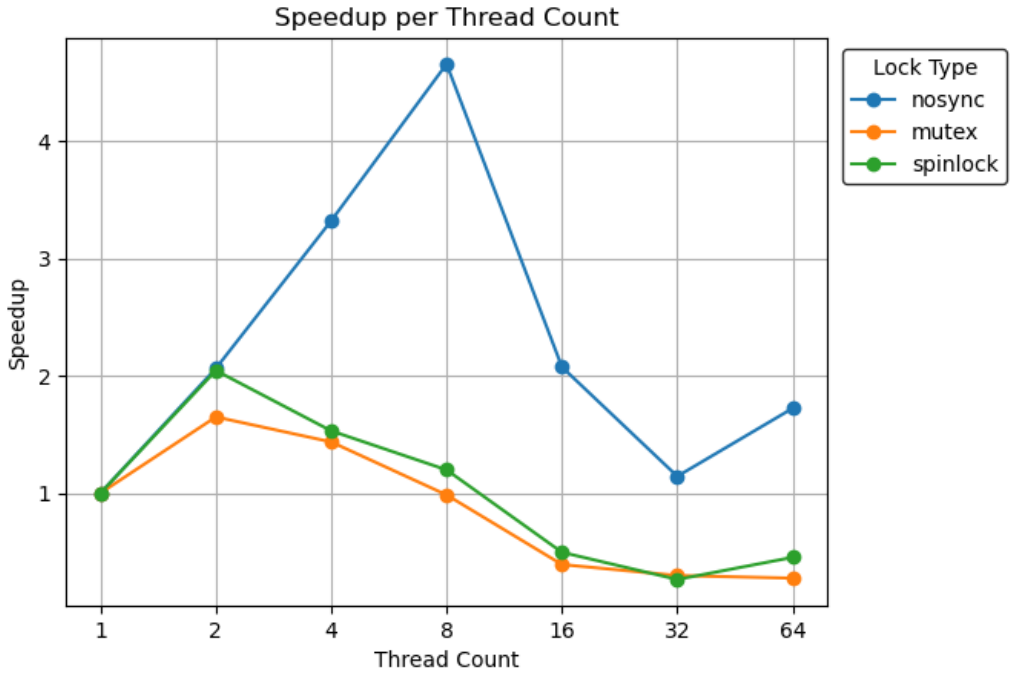
Παρατηρούμε ότι λαμβάνουμε τις καλύτερες επιδόσεις και κλιμακωσιμότητα με τα κλειδώματα Array, CLH και με το OMP Atomic. Στην περίπτωση μας, το κρίσιμο τμήμα περιέχει μόνο προσθέσεις και συνεπώς η χρήση ατομικών ενοτλών είναι πιο γρήγορη αφού παρακάμπτεται η χρήση κλειδωμάτων και γίνεται απευθείας εκτέλεση εντολών στον επεξεργαστή. Τα κλειδώματα Array και CLH είναι κλειδώματα με μορφή ουράς, δηλαδή κάθε νήμα ελέγχει τον προκάτοχο του ώστε να μάθει εάν είναι η σειρά του να μπει στο κρίσιμο τμήμα. Συνεπώς, μειώνεται σημαντικά η συμφόρηση του διαδρόμου δεδομένων που προκαλείται από cache invalidations αφού κάθε νήμα κάνει spin σε διαφορετική τοποθεσία. Ακόμη, το κρίσιμο τμήμα αξιοποιείται σε μεγαλύτερο βαθμό αφού, καθώς υπάρχει σειρά προτεραιότητας First Come First Serve, δεν χάνεται χρόνος στην διεκδίκηση του κλειδώματος. Στην συνέχεια θα εξετάσουμε τα κλειδώματα σε μεγαλύτερη λεπτομέρεια ανά ζευγάρια. Ο διαχωρισμός σε ζευγάρια έγινε με βάση την ομοιότητα μεταξύ των κλειδωμάτων.

nosync_lock: Αυτή η υλοποίηση κλειδώματος δεν παρέχει συγχρονισμό και συνεπώς χρησιμοποιείται ως άνω όριο για τη μελέτη της επίδοσης των υπόλοιπων κλειδωμάτων.

pthread_mutex_lock: Χρησιμοποιούμε το pthread_mutex_t που παρέχει η βιβλιοθήκη POSIX Pthreads. Τα mutexes χρησιμοποιούν μία ατομική μεταβλητή η οποία λαμβάνει την τιμή 1 όταν κάποιο νήμα έχει πάρει το κλείδωμα και την τιμή 0 όταν είναι διαθέσιμο. Σε περίπτωση που ένα νήμα ζητήσει το κλείδωμα ενώ αυτό δεν είναι διαθέσιμο το λειτουργικό σύστημα θα κοιμίσει (sleep) το νήμα και θα το ξυπνήσει πάλι όταν το κλείδωμα είναι διαθέσιμο. Τα mutexes ενδείκνυνται για μεγάλα τμήματα κώδικα μέσα στο κρίσιμο τμήμα καθώς η διαδικασία κοίμησης ενός νήματος από το λειτουργικό καταναλώνει πόρους της CPU. **pthread_spin_lock:** Χρησιμοποιούμε το pthread_spinlock_t που παρέχει η βιβλιοθήκη POSIX Pthreads. Τα spinlocks επίσης χρησιμοποιούν μία ατομική μεταβλητή η οποία λαμβάνει τιμές 1 και 0 με την διαφορά ότι σε περίπτωση που ένα νήμα ζητήσει το κλείδωμα ενώ αυτό δεν είναι διαθέσιμο τότε το νήμα περιμένει σε ένα busy wait loop, κάνει δηλαδή spin στον πυρήνα. Η ενεργή αναμονή καταναλώνει πόρους της CPU και συνεπώς ενδείκνυται για μικρά τμήματα κώδικα μέσα στο κρίσιμο τμήμα.

Ακολουθούν διαγράμματα σύγκρισης των δύο κλειδωμάτων:

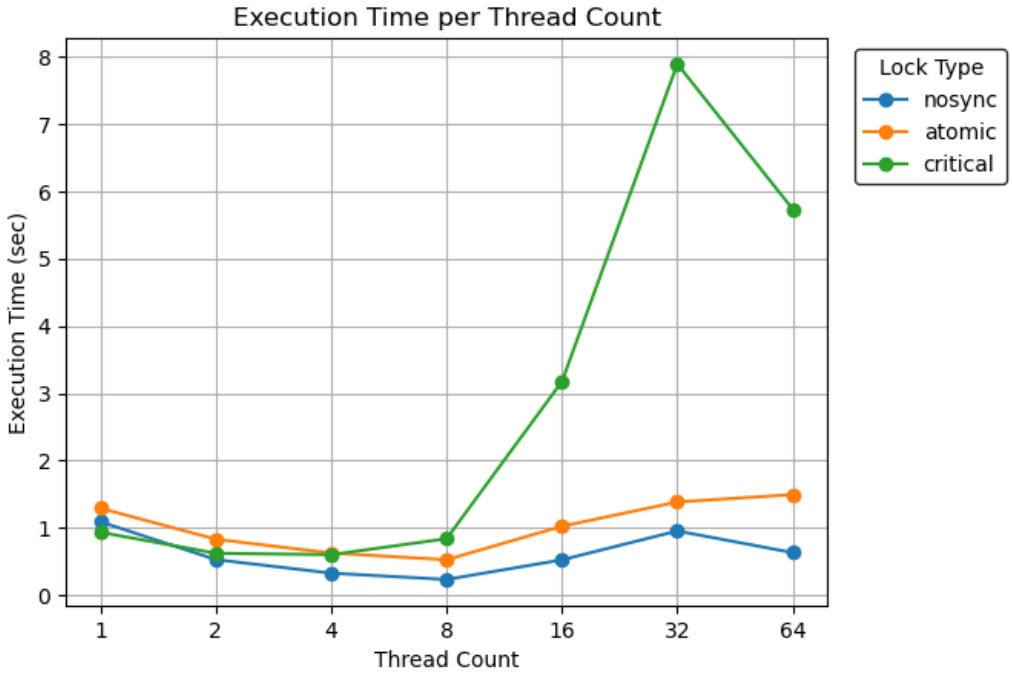


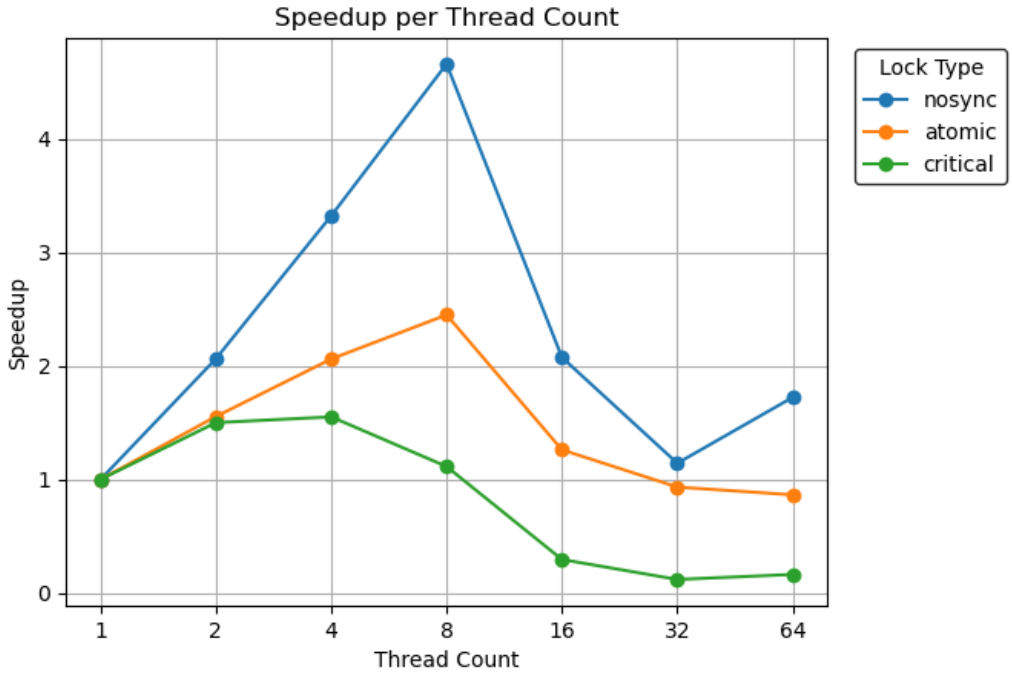


Η αύξηση των νημάτων από 1 σε 2 προκαλεί μείωση του χρόνου εκτέλεσης και αύξηση του speedup. Περαιτέρω αύξηση των νημάτων προκαλεί αύξηση του χρόνου και μείωση του speedup, δηλαδή τα κλειδώματα δεν κλιμακώνουν. Με την αύξηση των νημάτων χάνεται το όφελος του παραλληλισμού αφού μεγαλύτερος αριθμός νημάτων ανταγωνίζονται για ένα κλείδωμα και συνεπώς έχουμε μεγάλες καθυστερήσεις λόγο αναμονής, ενώ ταυτόχρονα καταναλώνονται περισσότεροι πόροι της CPU από το spin και την κοίμηση εργασιών για το spinlock και το mutex αντίστοιχα. Λαμβάνουμε καλύτερες επιδόσεις από τα spinlocks σε σχέση με τα mutexes για όλες τις τιμές νημάτων εκτός από 32, το οποίο ενδέχεται να οφείλεται σε ατυχή εκτέλεση. Αυτό είναι αναμενόμενο αφού το κρίσιμο τμήμα είναι μικρό (αύξηση counter/αλλαγή τιμής πίνακα).

OMP Atomic: Το OMP atomic είναι η υλοποίηση ατομικών εντολών του OpenMP. Οι ατομικές εντολές εκτελούνται στον πυρήνα, δεν εμπλέκεται δηλαδή το λειτουργικό σύστημα. **OMP Critical Section:** Το OMP critical section είναι η υλοποίηση κρίσιμου τμήματος του OpenMP. Ο κώδικας που περιέχεται στο κρίσιμο τμήμα μπορεί να εκτελεστεί από ένα νήμα τη φορά.

Ακολουθούν διαγράμματα σύγκρισης των δύο κλειδωμάτων:

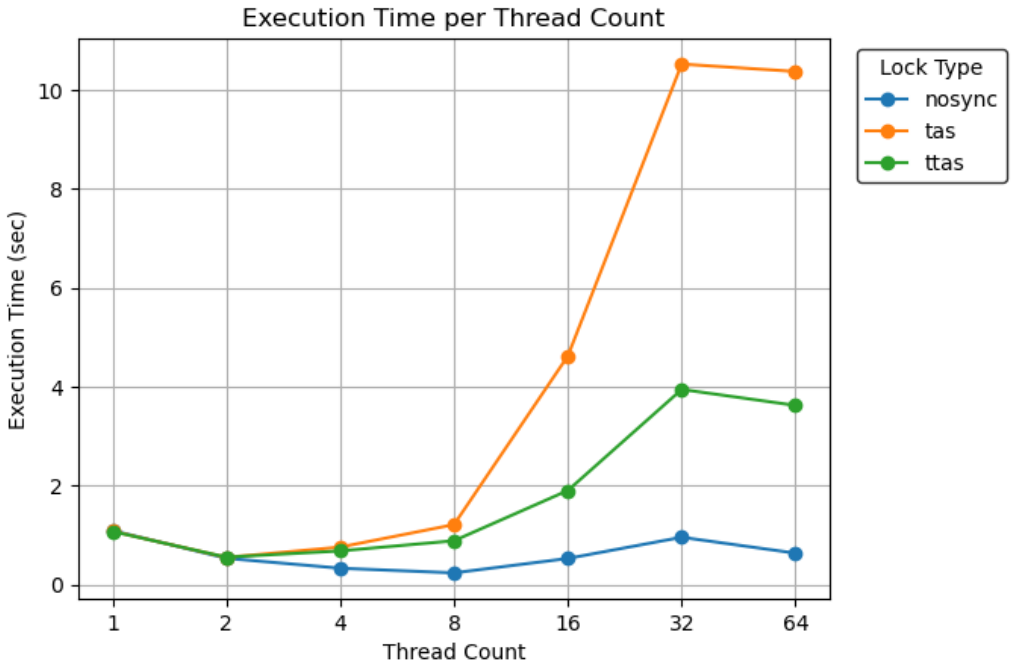


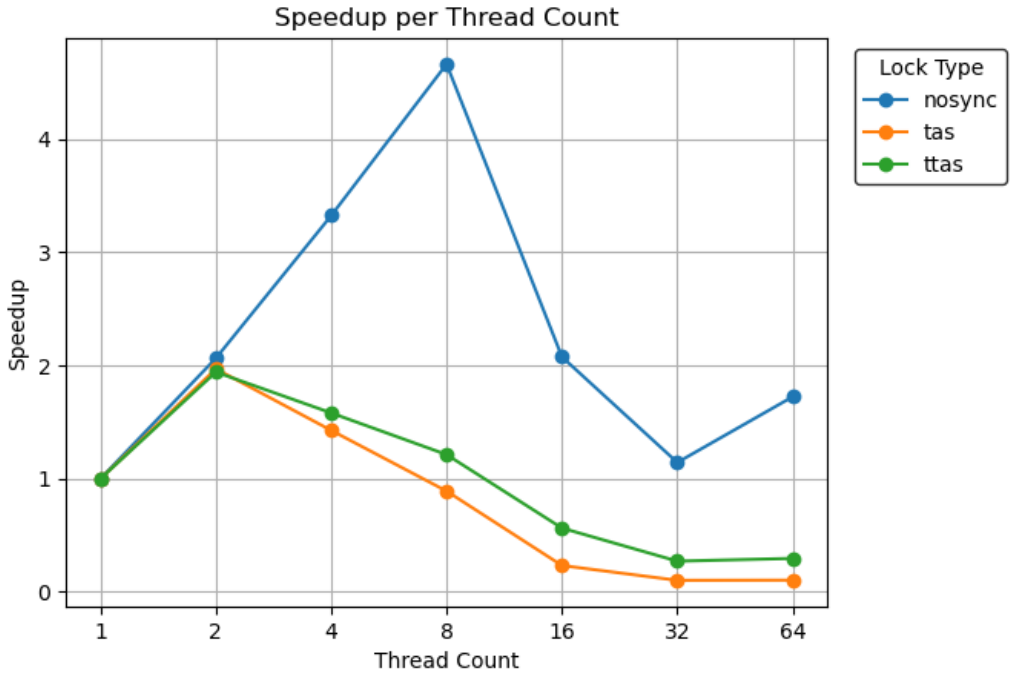


Το OMP critical με την αύξηση των νημάτων παρουσιάζει μεγάλη αύξηση του χρόνου εκτέλεσης, σε αντίθεση με το OMP atomic το οποίο διατηρεί σχεδόν σταθερούς χρόνους με μικρές αυξήσεις. Αυτό είναι αναμενόμενο αφού το OMP critical υλοποιείται με την χρήση κλειδωμάτων συνεπώς, με την αύξηση των νημάτων χάνεται το όφελος του παραλληλισμού αφού μεγαλύτερος αριθμός νημάτων ανταγωνίζονται για ένα κλείδωμα και συνεπώς έχουμε μεγάλες καθυστερήσεις λόγω αναμονής. Σε αντίθεση το OMP atomic χρησιμοποιεί ατομικές εντολές για τις οποίες υπάρχει υποστήριξη από το υλικό (hardware) και συνεπώς μειώνεται η καθυστέρηση αφού δεν παρεμβάλλεται κάποιο κλείδωμα, και συνεπώς το λειτουργικό σύστημα, πριν την εκτέλεση των εντολών.

TAS lock: Το κλείδωμα Test and Set υλοποιείται με την ατομική μεταβλητή state. Για να λάβει ένα νήμα το κλείδωμα πρέπει να θέσει το state σε 1. Αν η μεταβλητή προηγουμένως είχε τιμή 0 σημαίνει ότι το κλείδωμα ήταν ελεύθερο και συνεπώς δίνεται στο νήμα. Διαφορετικά το νήμα περιμένει σε busy wait loop μέχρι το state να πάρει τιμή 0.
TTAS lock: Το κλείδωμα Test and Test and Set υλοποιείται όπως το TAS με την διαφορά ότι διαβάζει συνεχώς την μεταβλητή state ώστε να βεβαιωθεί ότι το κλείδωμα είναι διαθέσιμο πριν το διεκδικήσει, σε αντίθεση με το TAS που διεκδικεί συνεχώς το lock.

Ακολουθούν διαγράμματα σύγκρισης των δύο κλειδωμάτων:

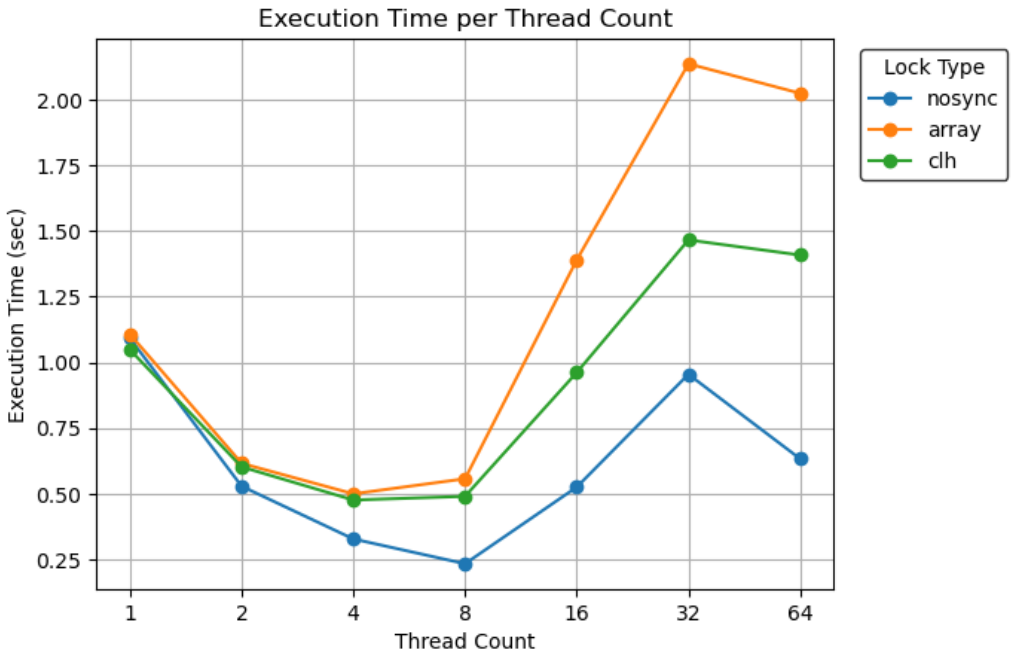


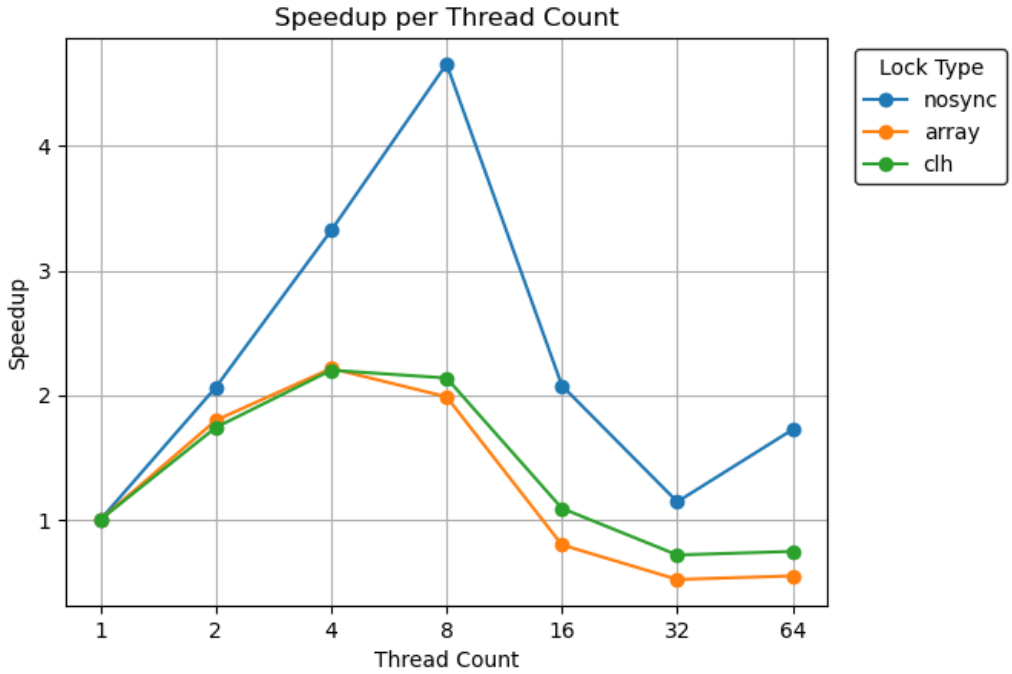


Παρατηρούμε ότι με την αύξηση των νημάτων το TAS έχει χειρότερη επίδοση και κλιμάκωση σε σχέση με το TTAS. Αυτό οφείλεται στο γεγονός ότι το TAS μαρκάρει την μεταβλητή state ως αλλαγμένη ακόμα και αν δεν κατάφερε να αλλάξει την τιμή της, προκαλεί δηλαδή cache invalidation σε όλα τα νήματα που περιμένουν το κλείδωμα, το οποίο με τη σειρά του προκαλεί μεγάλη συμφόρηση στον διάδρομο δεδομένων. Στην περίπτωση του TTAS πριν προσπαθήσουμε να αλλάξουμε την τιμή του πεδίου state διαβάζουμε συνεχώς την τιμή του ώστε να βεβαιωθούμε ότι το κλείδωμα είναι διαθέσιμο, συνεπώς θα προκληθούν λιγότερα cache invalidations. Και στις δύο περιπτώσεις τα κλειδώματα δεν είναι ”δίκαια”, δηλαδή μπορεί κάποιο νήμα να μην μπει ποτέ στο κρίσιμο τμήμα (starvation).

Array Lock: Το κλείδωμα Array υλοποιείται με χρήση μίας διαμοιραζόμενης ατομικής (atomic) μεταβλητής tail και ενός διαμοιραζόμενου πίνακα flag. Κάθε θέση του πίνακα αντιστοιχεί σε ένα νήμα. Για να πάρει ένα νήμα το κλείδωμα αυξάνει την τιμή του tail και εάν το flag του έχει τιμή true τότε το παίρνει. Διαφορετικά κάνει spin μέχρι το flag να πάρει τιμή true. Κατά την απελευθέρωση του κλειδώματος το νήμα θέτει το flag του σε false και θέτει το flag της επόμενης θέσης του πίνακα σε true. Το κλείδωμα ουσιαστικά λειτουργεί σαν σκυτάλη, δηλαδή δίνεται από κάθε νήμα στο επόμενο με πολιτική First Come First Serve.
CLH Lock: Το κλείδωμα CLH υλοποιείται με χρήση συνδεδεμένης λίστας όπου στην μνήμη κάθε νήματος περιέχεται ο κόμβος του και ο κόμβος του predecessor του. Επίσης υπάρχει μία διαμοιραζόμενη ατομική (atomic) μεταβλητή tail η οποία είναι δείκτης στον τελευταίο κόμβο που προστέθηκε στην λίστα. Κάθε νήμα που επιθυμεί να πάρει το κλείδωμα δημιουργεί τον κόμβο του και θέτει το πεδίο locked του κόμβου σε true. Ο κόμβος του προστίθεται στο τέλος της λίστας με βάση το tail και κάνει spin πάνω στο πεδίο locked του predecessor του, δηλαδή του κόμβου που προηγείται στην λίστα από αυτόν. Όταν αυτό το πεδίο λάβει τιμή false τότε το νήμα μπορεί να πάρει το κλείδωμα. Για να αφήσει ένα νήμα το κλείδωμα αρκεί να αλλάξει την τιμή του πεδίου locked του κόμβου του σε false. Και σε αυτή την περίπτωση τα νήματα εξυπηρετούνται με πολιτική First Come First Serve.

Ακολουθούν διαγράμματα σύγκρισης των δύο κλειδωμάτων:





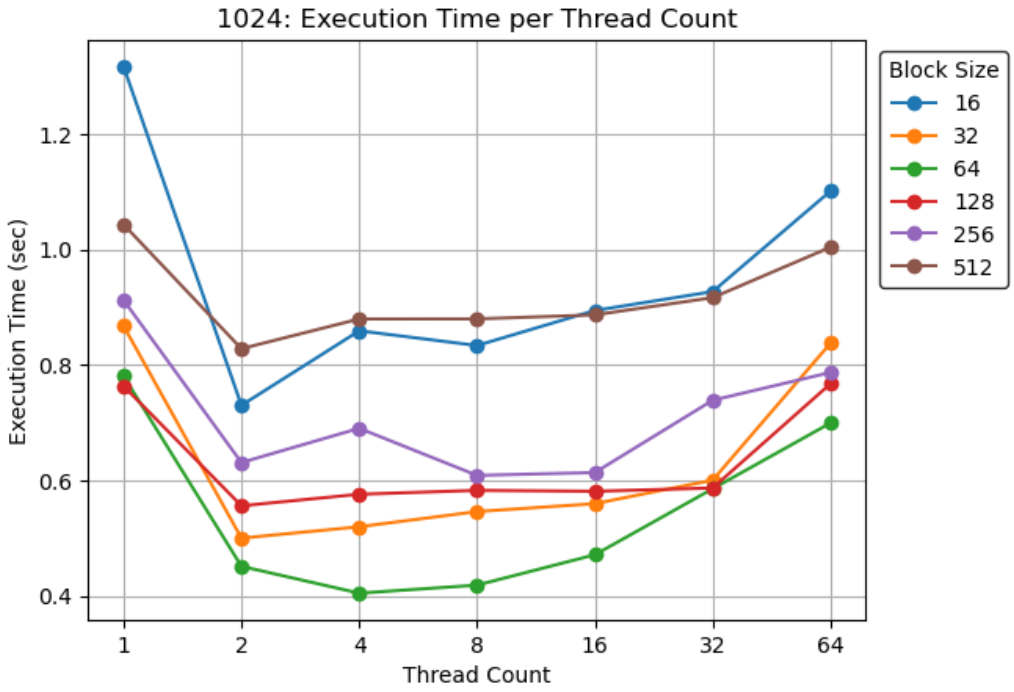
Και στις δύο περιπτώσεις παρατηρούμε ότι, με την αύξηση του αριθμού νημάτων από 8 και άνω, αυξάνεται και ο χρόνος εκτέλεσης του αλγορίθμου. Στην περίπτωση του κλειδώματος CLH, όταν κάποιο νήμα αφήνει το κλείδωμα γίνεται invalidation στην cache του successor του διότι αυτός κάνει συνεχώς spin στο πεδίο locked του predecessor του. Στην περίπτωση του κλειδώματος Array, υλοποιούμε το κλείδωμα με padding, δηλαδή προσθέτουμε χώρο ανάμεσα στις θέσεις διαδοχικών νημάτων στον πίνακα flag ώστε να μην βρίσκονται στην ίδια cache line και συνεπώς μειώνουμε τα cache invalidations λόγω false sharing χωρίς όμως να τα εξαλείψουμε. Τα παραπάνω έχουν ως αποτέλεσμα την μείωση της επίδοσης με την αύξηση των νημάτων. Η καλύτερη επίδοση του CLH σε σχέση με το Array για αριθμό threads πάνω από 8, οφείλετε στο γεγονός ότι το Array υλοποιείται με διαμοιραζόμενο πίνακα ενώ στο CLH το κομμάτι της συνδεδεμένης λίστας που αφορά κάθε νήμα είναι τοπικό σε αυτό. Ως αποτέλεσμα παρατηρείται μεγαλύτερη συμφόρηση στον διάδρομο δεδομένων και συνεπώς πιο κακή επίδοση με την χρήση κλειδώματος Array.

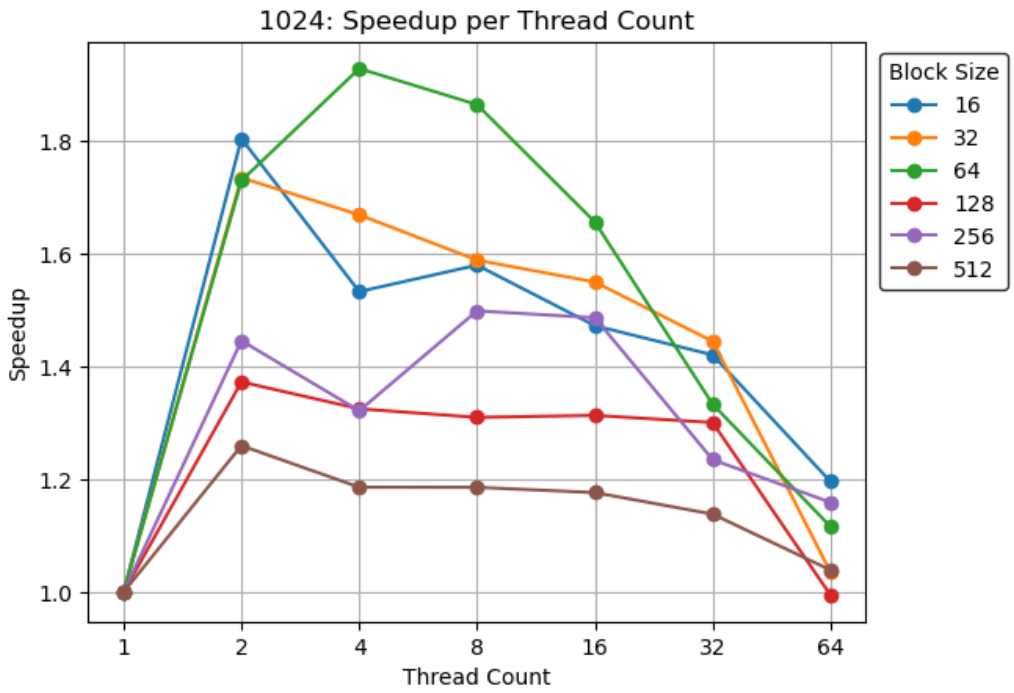
2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Σε αυτή την άσκηση παραλληλοποιούμε την recursive έκδοση του αλγορίθμου Floyd-Warshall με OpenMP tasks και τους πίνακες A,B,C διαμοιραζόμενους. Αναθέτουμε κλήσεις της συνάρτησης FW_SR που δεν αφορούν τα ίδια σημεία του πίνακα σε διαφορετικά tasks. Ακολουθούν ο κώδικας και τα διαγράμματα χρόνου και speedup ως προς τα νήματα για τις δύο εκδόσεις.

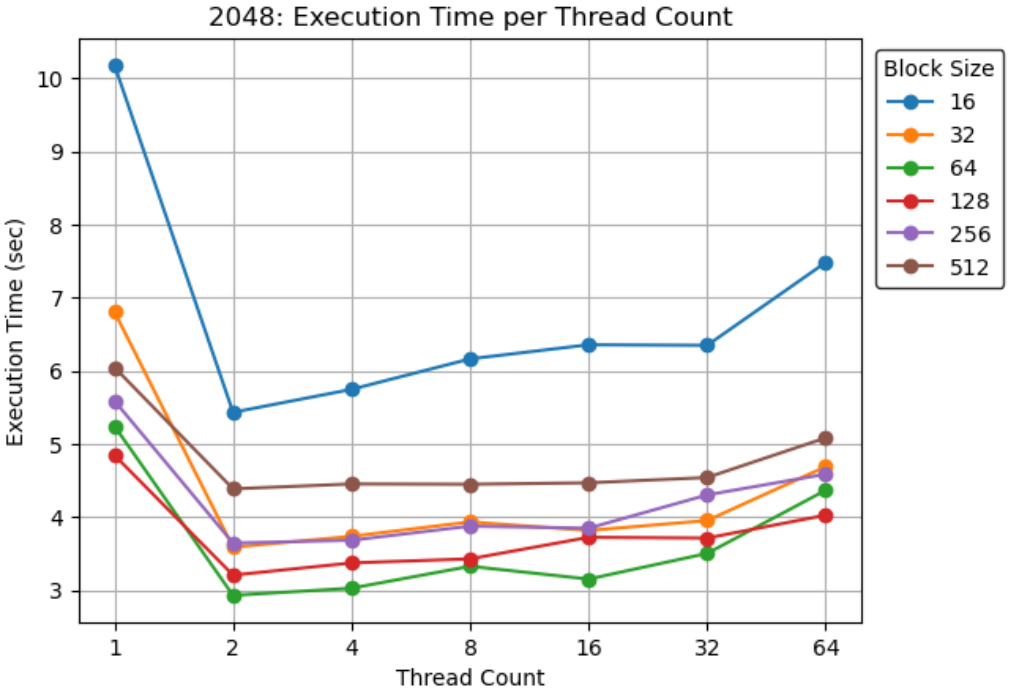
```
void FW_SR (int **A, int arow, int acol,int **B, int brow, int bcol,
            int **C, int crow, int ccol, int myN, int bsize)
{
    int k,i,j;

    if(myN<=bsize)
        for(k=0; k<myN; k++)
            for(i=0; i<myN; i++)
                for(j=0; j<myN; j++)
                    A[arow+i][acol+j]=min(A[arow+i][acol+j],
↪   B[brow+i][bcol+k]+C[crow+k][ccol+j]);
    else {
        FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
        #pragma omp parallel private(k, i, j)
        {
            #pragma omp single
            {
                #pragma omp task shared(A, B, C)
                    FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2,
↪   myN/2, bsize);
                FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol,
↪   myN/2, bsize);
                #pragma omp taskwait
            }
        }
        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
↪   myN/2, bsize);
        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
↪   ccol+myN/2, myN/2, bsize);
        #pragma omp parallel private(k, i, j)
                {
                        #pragma omp single
                        {
                                #pragma omp task shared(A, B, C)
                    FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
↪   bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
                FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2,
↪   ccol+myN/2, myN/2, bsize);
                #pragma omp taskwait
            }
        }
        FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2,
↪   bsize);
    }
}
```

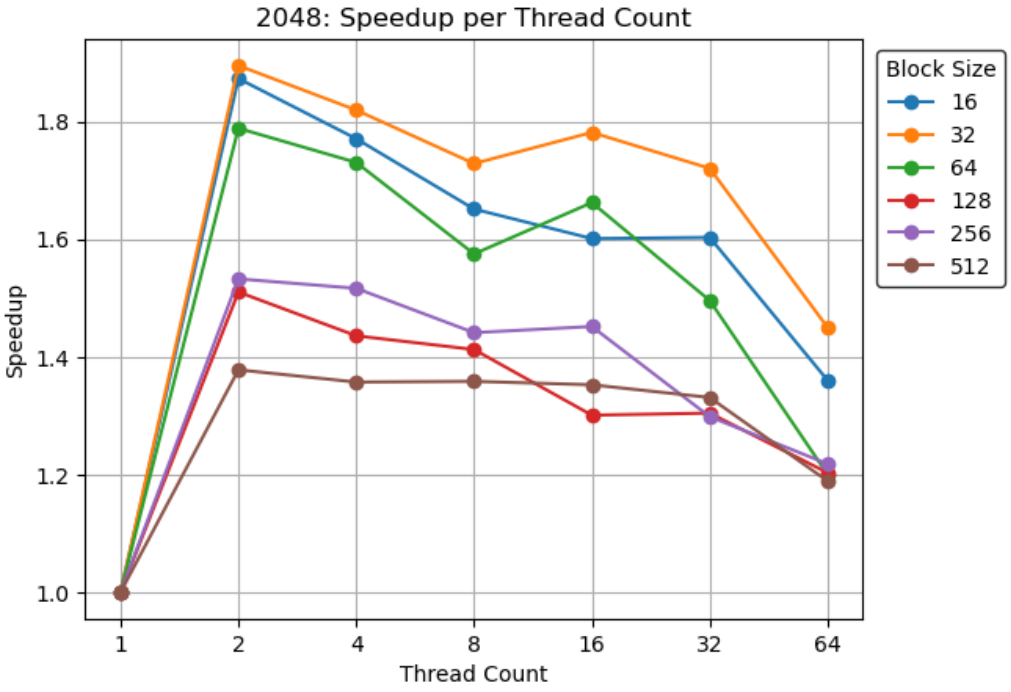
2.2.1 Πίνακας 1024x1024



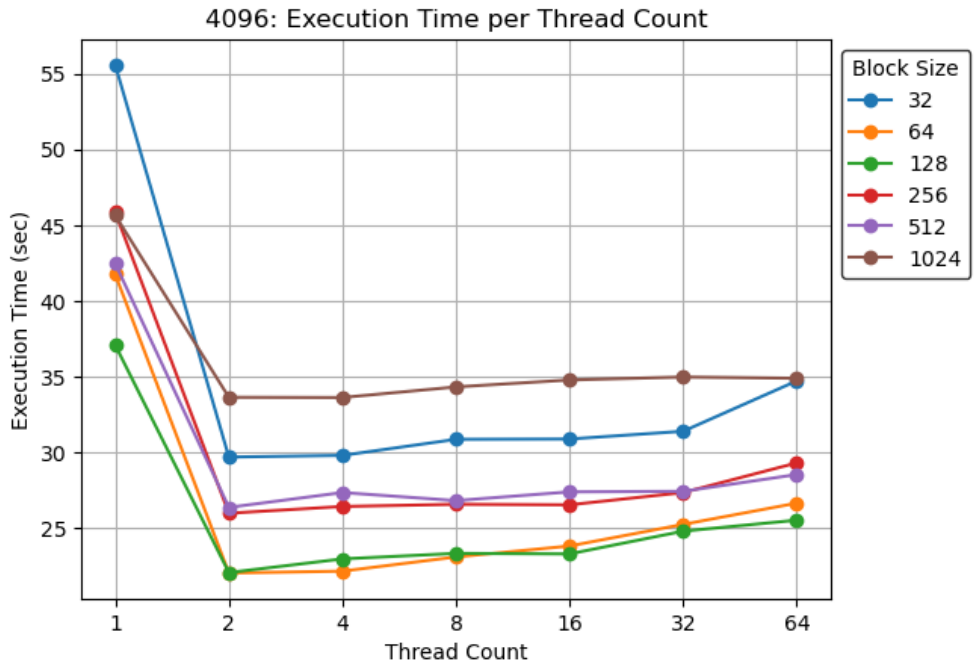


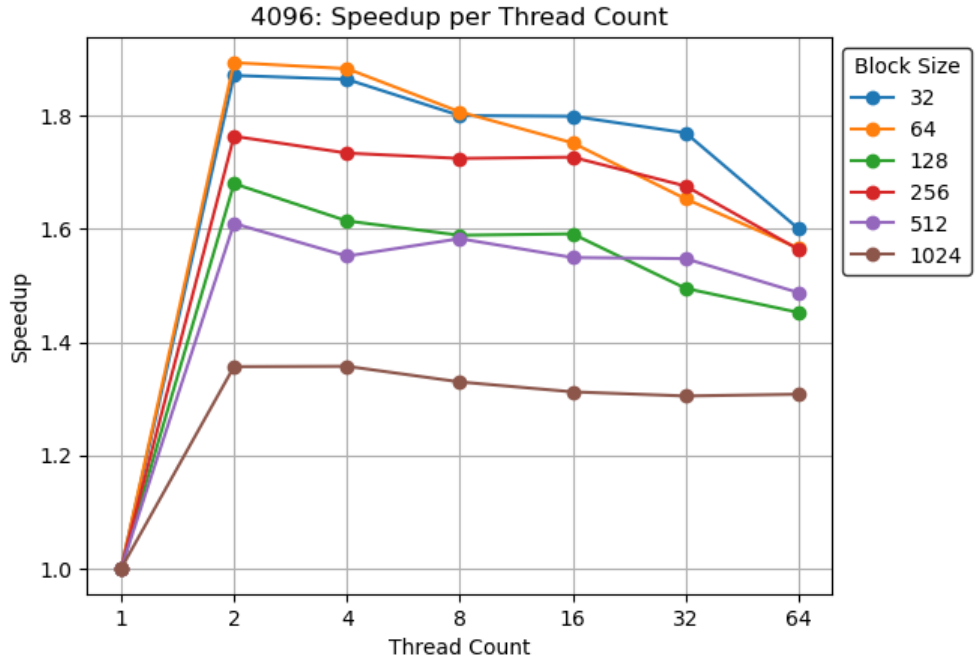
2.2.2 Πίνακας 2048x2048





2.2.3 Πίνακας 4096x4096



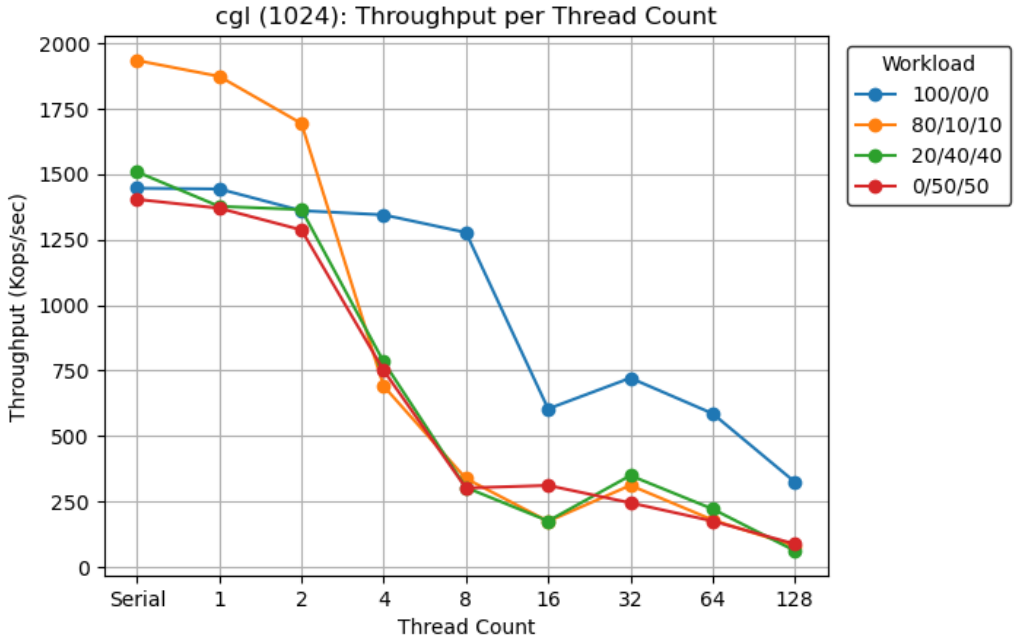


Παρατηρούμε για κάθε μέγεθος πίνακα σημειώνουμε καλύτερες επιδόσεις για block size 64 ή 128. Στην περίπτωση μικρότερων block size αποθηκεύονται περισσότερα από ένα block σε κάθε σελίδα της cache με αποτέλεσμα να παρατηρείται το φαινόμενο false sharing. Για μεγαλύτερα block size ένα block αποθηκεύεται σε πολλές σελίδες συνεπώς, πρέπει να φορτωθούν περισσότερες σελίδες της cache από κάθε νήμα. Και στις δύο περιπτώσεις προκαλείται συμφόρηση στο διάδρομο μνήμης με αποτέλεσμα την αύξηση των χρόνων εκτέλεσης. Συμφόρηση στο διάδρομο μνήμης προκαλείται επίσης και από την έλλειψη τοπικότητας των OpenMP tasks, σε αντίθεση με το παράλληλο for. Συνεπώς, όπως είναι φανερό και από στα διαγράμματα, έχουμε κακό scaling. Τέλος, ο συγκεκριμένος αλγόριθμος επιτρέπει την παράλληλη εκτέλεση του δεύτερου βήματος της αναδρομής με το τρίτο και του έκτου με το έβδομο, έχουμε δηλαδή μεγάλο κόστος συγχρονισμού αφού τέσσερα από τα οκτώ βήματα εκτελούνται σειριακά και ο μέγιστος αριθμός παράλληλων βημάτων είναι δύο.

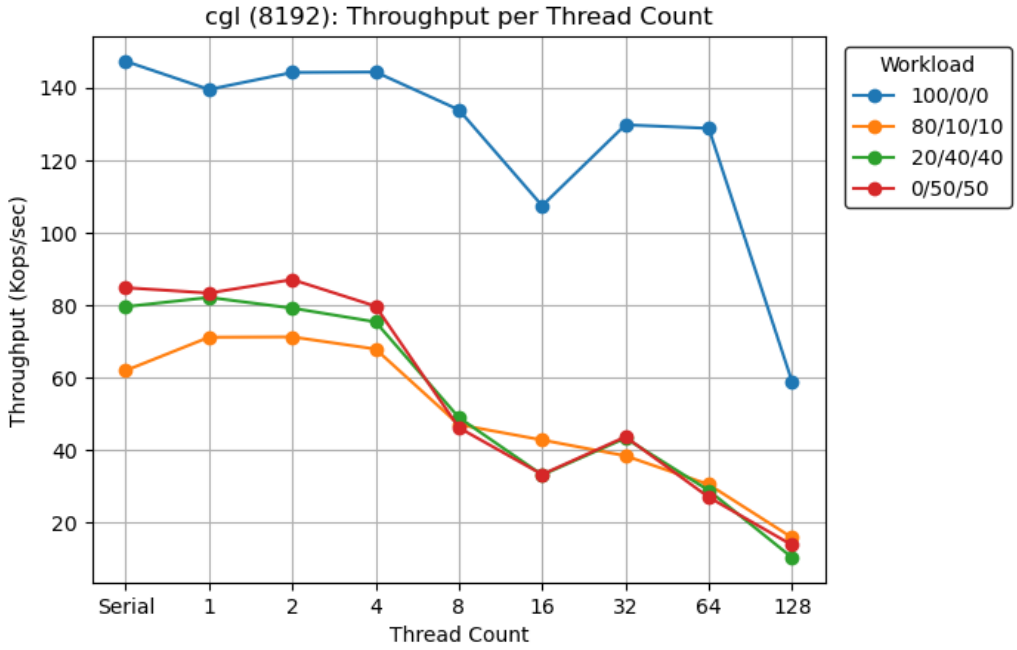
2.3 Ταυτόχρονες Δομές Δεδομένων

Σκοπός αυτής της άσκησης είναι η αξιολόγηση της επίδοσής διάφορων ταυτόχρονων υλοποιήσεων απλής συνδεδεμένης λίστας υπό διαφορετικές συνθήκες.
Γενικές Παρατηρήσεις: Σε όλα τα διαγράμματα ότι για μεγαλύτερο μέγεθος λίστας έχουμε μικρότερο throughput. Αυτό συμβαίνει επειδή οι μέθοδοι contains, add και remove πρέπει να διατρέξουν την λίστα.
Ακολουθούν τα διαγράμματα της κάθε υλοποίησης με σχολιασμό:

2.3.1 Coarse-Grain Locking

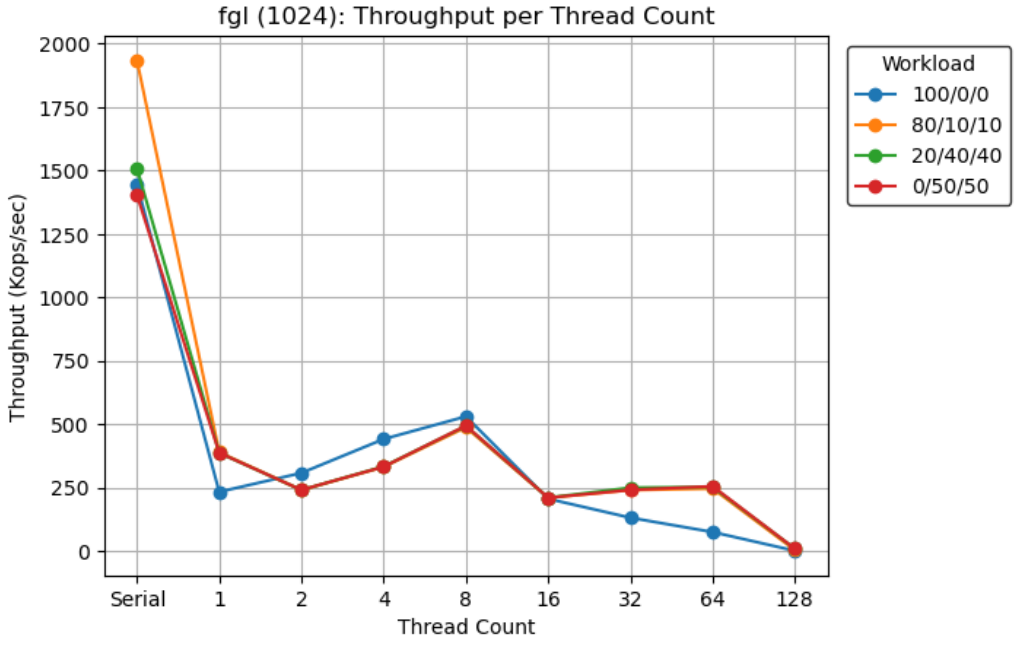


Το throughput σε όλα τα worload, με εξαίρεση το 100/0/0 όπου παραμένει περίπου σταθερό μέχρι τα 4 νήματα, μειώνεται καθώς αυξάνονται τα νήματα. Αυτό είναι αναμενόμενο αφού κλειδώνουμε ολόκληρη τη λίστα και επομένως έχουμε σειριακή συμπεριφορά με επιπρόσθετο overhead από τη διαχείριση των νημάτων. Το workload 100/0/0 έχει υψηλότερο throughput καθώς το contains διατρέχει την λίστα χωρίς όμως να αλλάζει τους pointers όπως τα operations add και remove.

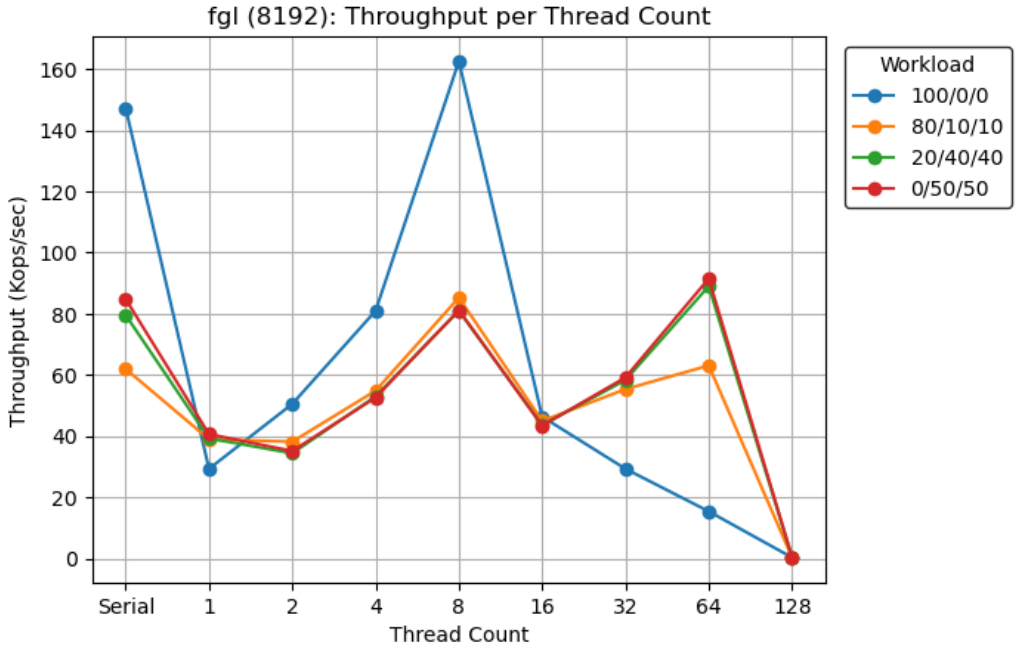


Το workload 100/0/0 έχει πάλι υψηλότερο throughput ενώ τα υπόλοιπα workload έχουν παρόμοια επίδοση. Παρατηρούμε πάλι πτωτική τάση στο throughput για τους ίδιους λόγους που προαναφέρθηκαν.

2.3.2 Fine-Grain Locking

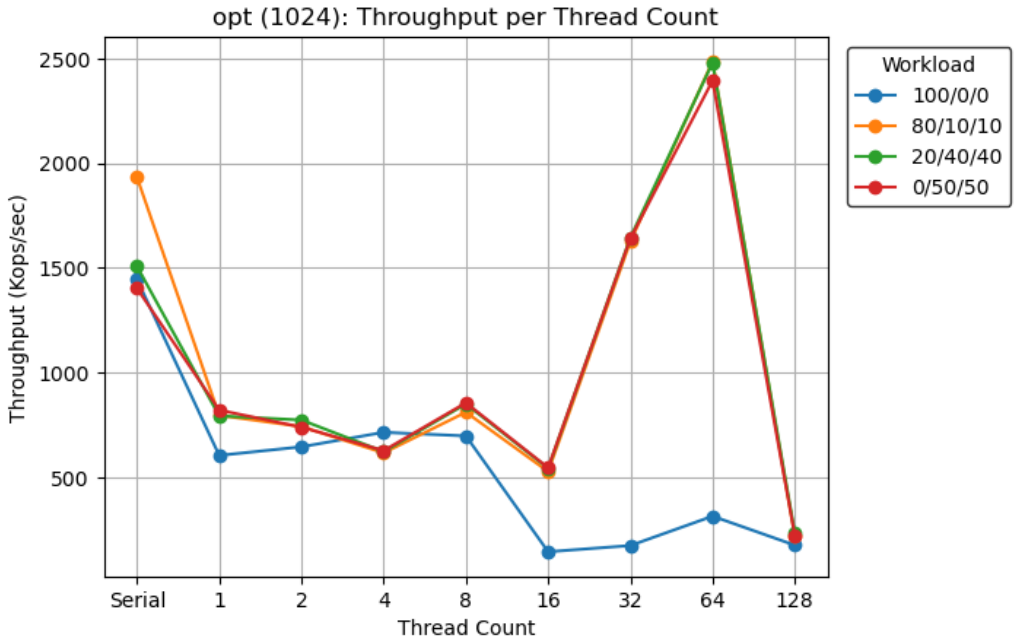


Παρατηρούμε ότι πάλι έχουμε κακό scaling. Η επίδοση μειώνεται με την αύξηση των νημάτων, με εξαίρεση μια μικρή βελτίωση για 8 νήματα η οποία όμως και πάλι δεν ξεπερνά την επίδοση του σειριακού κώδικα. Αυτό οφείλεται στο γεγονός ότι, στην υλοποίηση Fine-Grain, νήματα που προσπελαύνουν διακριτά μέρη της λίστας μπορεί να αδυνατούν να εργαστούν ταυτόχρονα δηλαδή, εάν ένα νήμα εργάζεται στην αρχή της λίστας τα άλλα νήματα θα πρέπει να το περιμένουν πριν προχωρήσουν.

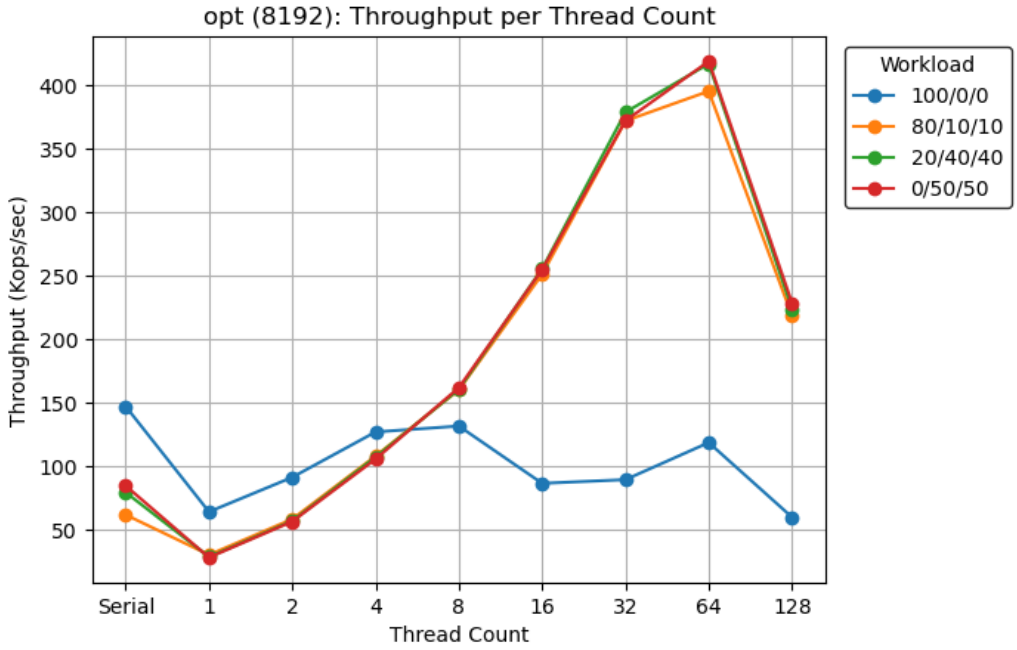


Το throughput μειώνεται σε σχέση με αυτό για μέγεθος λίστας 1024 λόγω του αυξημένου μεγέθους της λίστας όπως έχει προαναφερθεί. Παρόλα αυτά, το αυξημένο μέγεθος λίστας επωφελείται περισσότερο από τον παραλληλισμό, το οποίο είναι φανερό από της αυξήσεις επίδοσης για 8 και 64 νήματα. Τα νήματα εργάζονται σε μεγαλύτερους χώρους και συνεπώς έχουμε λιγότερα instances κατά τα οποία ένα νήμα θα πρέπει να περιμένει ένα άλλο

2.3.3 Optimistic Synchronisation

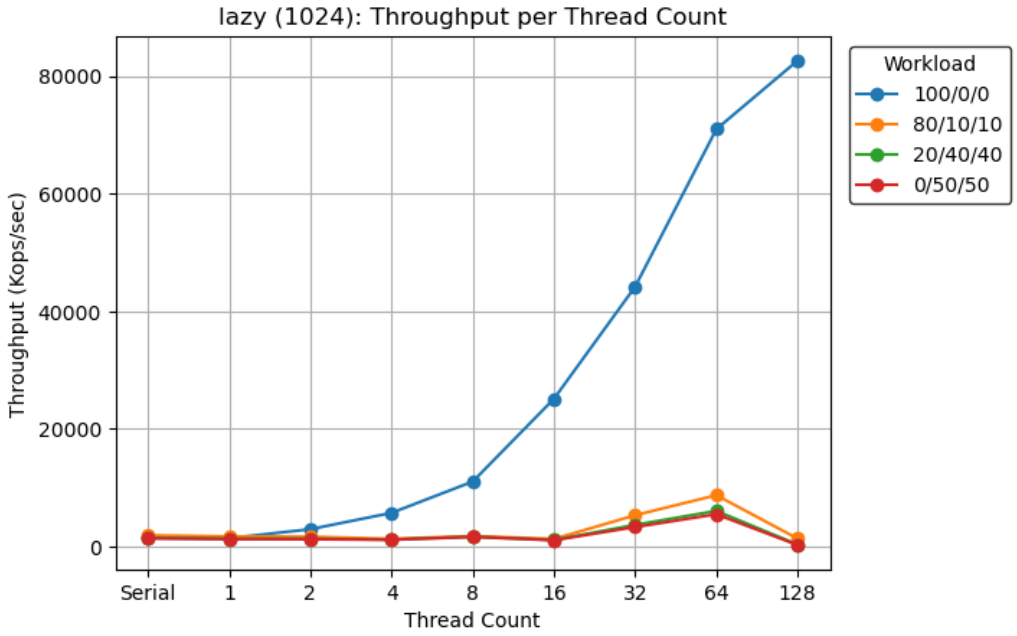


Παρατηρούμε ότι η υλοποίηση με Optimistic Synchronisation εμφανίζει μεγαλύτερο throughput σε σχέση με τις Coarse-Grain και Fine-Grain υλοποιήσεις, καθώς η αναζήτηση γίνεται χωρίς την χρήση κλειδωμάτων. Παρατηρούμε ότι δεν έχουμε καλό scaling, με εξαίρεση τα 64 νήματα όπου όλα τα workload εκτός του 100/0/0 σημειώνουν σημαντική βελτίωση της επίδοσης. Για μικρό μέγεθος λίστας υπάρχει μεγαλύτερη συγκέντρωση νημάτων στην ίδια περιοχή και συνεπώς περισσότερες συγκρούσεις (αποτυχίες της validate) οπότε διατρέχουμε περισσότερες φορές την λίστα.

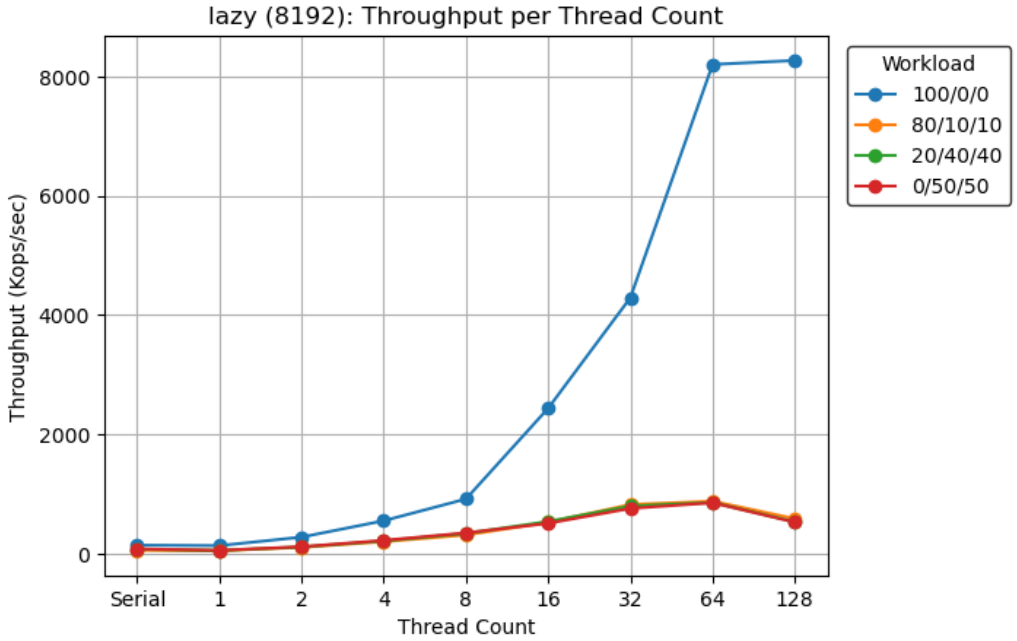


Για μέγεθος λίστας 8192 στοιχείων παρατηρούμε χαμηλότερο throughput, όπως είναι αναμενόμενο για μεγαλύτερη λίστα, αλλά καλύτερο scaling καθώς, όπως προαναφέρθηκε, για μεγαλύτερο μέγεθος λίστας έχουμε λιγότερες συγκρούσεις.

2.3.4 Lazy Synchronisation

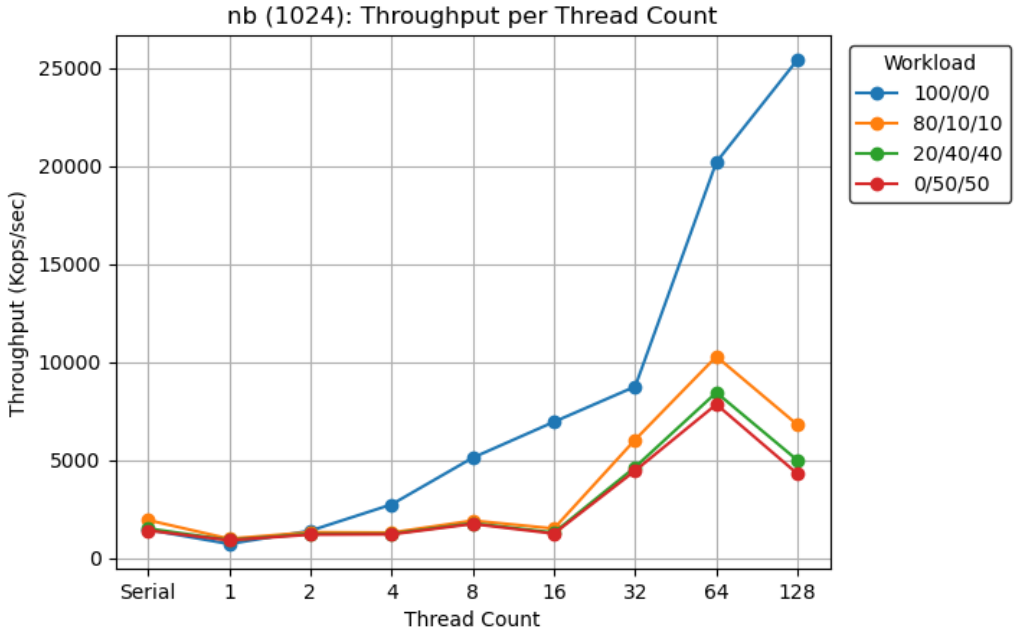


Με την υλοποίηση Lazy Synchronisation καταγράφουμε καλύτερο throughput και scaling σε σχέση με την υλοποίηση Optimistic, ιδιαίτερα για το workload 100/0/0, καθώς η μέθοδος contains υλοποιείται χωρίς κλειδώματα και η validate ελέγχει τους κόμβους τοπικά, δηλαδή δεν διατρέχει όλη την λίστα.

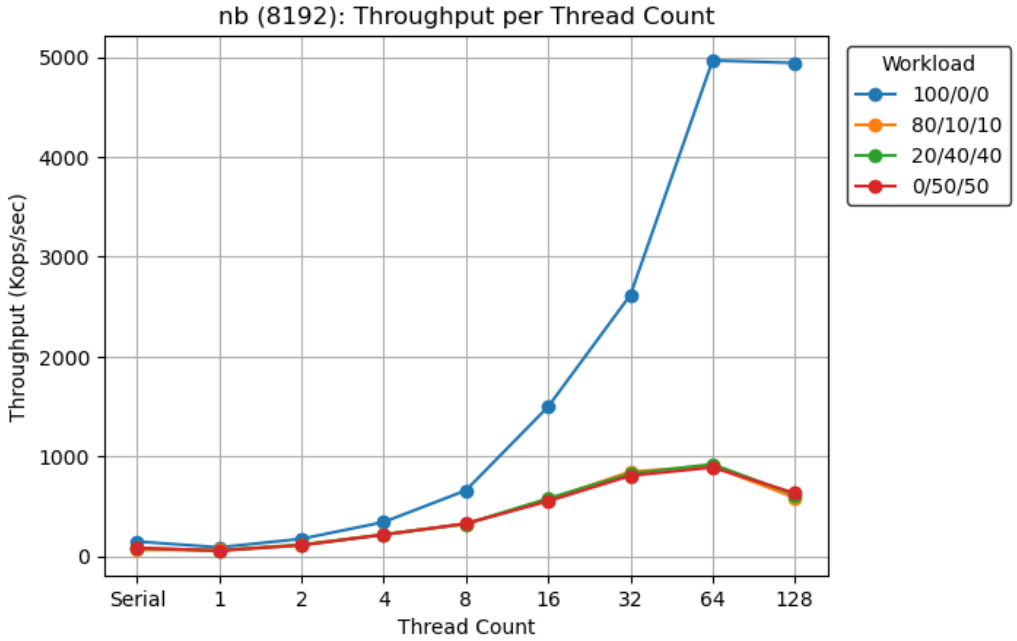


Παρατηρούμε παρόμοιο scaling με αυτό που εμφανίζεται για μέγεθος λίστας 1024 αλλά χαμηλότερο throughput καθώς αυξάνεται το μέγεθος της λίστας και συνεπώς ο χρόνος αναζήτησης στοιχείων.

2.3.5 Non-Blocking Synchronisation



Παρατηρούμε για workload 100/0/0 ότι η υλοποίηση Non-Blocking έχει χαμηλότερο throughput σε σχέση με την Lazy. Αυτό οφείλεται στο γεγονός ότι η μέθοδος contains δεν χρησιμοποεί κλειδώματα στην περίπτωση της Lazy ενώ στην περίπτωση της Non-Blocking χρησιμοποιεί ατομικές εντολές. Το πλεονέκτημα της χρήσης ατομικών εντολών είναι ότι η contains είναι wait-free, δηλαδή όλα τα νήματα προοδεύουν. Στα υπόλοιπα workload παρατηρούμε παρόμοιους χρόνους και scaling με την Lazy διότι ενώ το operation contains παρουσιάζει χαμηλότερο throughput, για τα operations add, remove η χρήση ατομικών εντολών αντί κλειδωμάτων έχει καλύτερη επίδοση.



Παρατηρούμε παρόμοιο scaling με αυτό που εμφανίζεται για μέγεθος λίστας 1024, με εξαίρεση την αύξηση της επίδοση για 64 νήματα. Το throughput πάλι μειώνεται καθώς αυξάνεται το μέγεθος της λίστας.

3 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

3.1 Σκοπός της Άσκησης

Σε αυτή την ενότητα καλούμαστε να υλοποιήσουμε και να βελτιστοποιήσουμε τον αλγόριθμο K-means σε επεξεργαστές γραφικών NVIDIA με χρήση του εργαλείου CUDA. Από τους σκελετούς των τριών υλοποιήσεων Naive, Transpose και Shared συμπληρώθηκαν οι συναρτήσεις euclid_dist_2, find_nearest_cluster και kmeans_gpu.

3.2 Naive Version

Σε αυτή την έκδοση αναθέτουμε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι του αλγορίθμου: τον υπολογισμό των nearest clusters σε κάθε iteration. Ακολουθεί ο κώδικας των συναρτήσεων euclid_dist_2, find_nearest_cluster και kmeans_gpu:

euclid_dist_2

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

    /* DONE: 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);
}
```

find_nearest_cluster

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

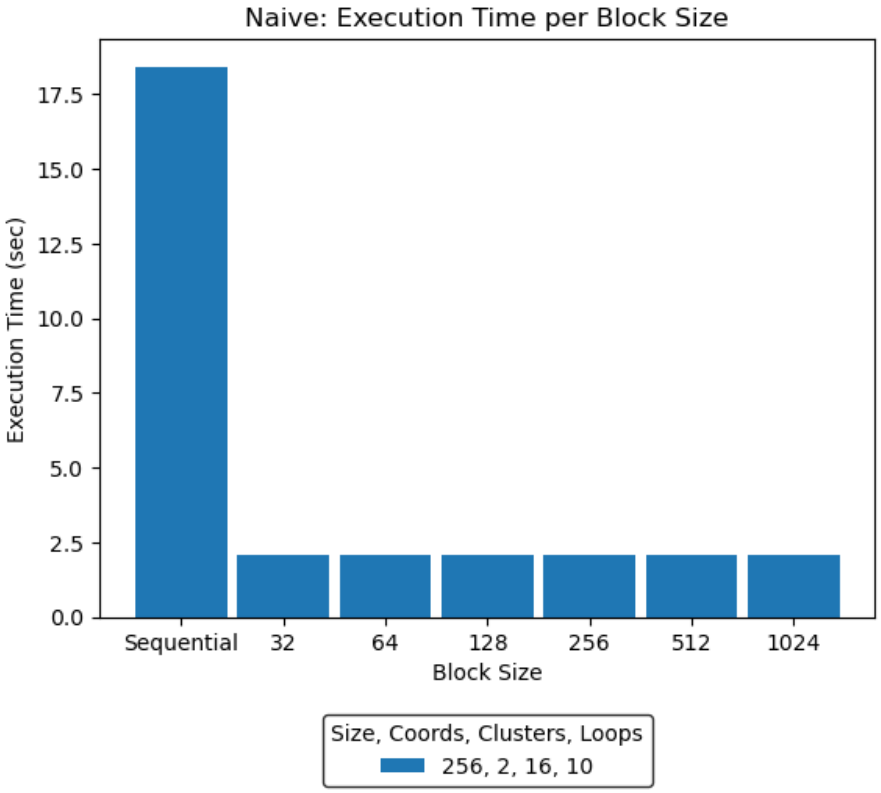
```
{
    /* Get the global ID of the thread. */
    int tid = get_tid();

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

        /* find the cluster id that has min distance to object */
        index = 0;
        /* DONE: 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++) {
            /* DONE: 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) {
            /* DONE: Maybe something is missing here... is this write safe? */
            atomicAdd(devdelta, 1.0);
        }

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

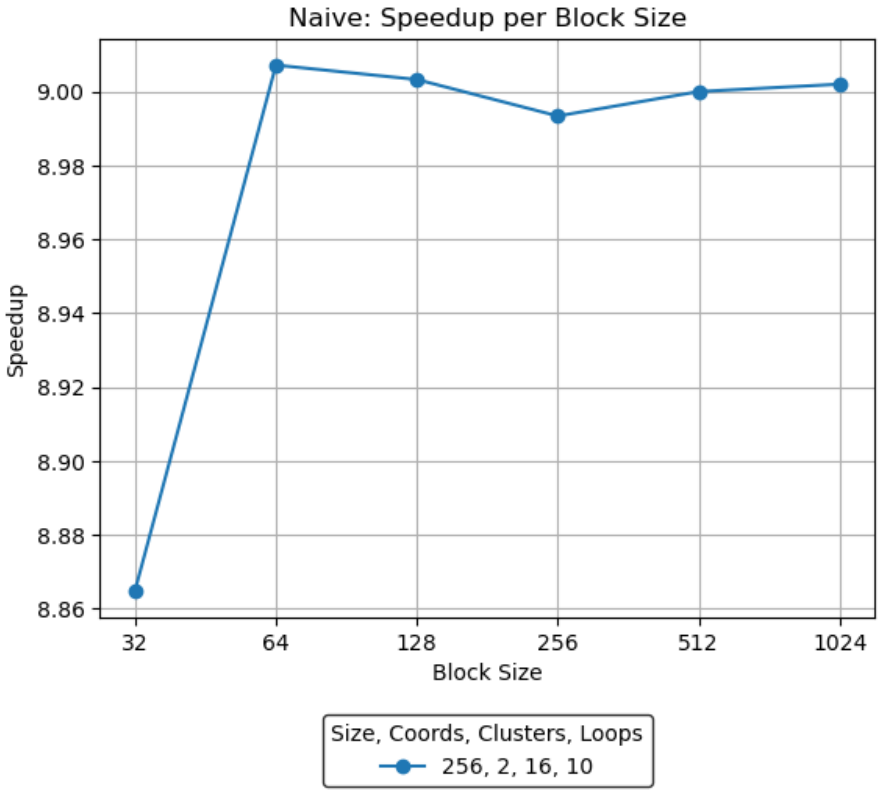
kmeans_gpu

```
void kmeans_gpu(float *objects,      /* in: [numObjs][numCoords] */
                int     numCoords,    /* no. features */
                int     numObjs,      /* no. objects */
                int     numClusters,  /* no. clusters */
                float   threshold,    /* % objects change membership */
                long    loop_threshold,   /* maximum number of iterations */
                int    *membership,   /* out: [numObjs] */
                float * clusters,   /* out: [numClusters][numCoords] */
                int blockSize)
{

    ...
```

```
    printf("\n|-------------Naive GPU Kmeans--------------|\n\n");

    ...

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

    ...

    do {
        timing_internal = wtime();

        /* GPU part: calculate new memberships */

        /* DONE: Copy clusters to deviceClusters */
        checkCuda(cudaMemcpy(deviceClusters, clusters,
        numClusters*numCoords*sizeof(float), cudaMemcpyHostToDevice));

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

        ...

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

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

        /* CPU part: Update cluster centers*/

        ...

    } while (delta > threshold && loop < loop_threshold);

    timing = wtime() - timing;

    ...

    return;
}
```

Ακολουθεί το διάγραμμα χρόνου εκτέλεσης (bar plot) και το αντίστοιχο διάγραμμα speedup για το configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10}:





Παρατηρούμε ότι έχουμε βελτίωση στον χρόνο εκτέλεσης σε σχέση με τον σειριακό αλγόριθμο, αφού οι πιο βαριοί υπολογισμοί γίνονται από τη GPU. Ωστόσο, βλέπουμε ότι το μέγεθος block δεν επηρεάζει τον χρόνο εκτέλεσης και το speedup (έχουμε πολύ μικρή διακύμανση σε τιμές). Αυτό συμβαίνει γιατί ο αριθμός των νημάτων είναι πάντα ίσος με τον αριθμό των αντικειμένων για τα συγκεκριμένα μεγέθη block, τα οποία είναι μικρότερα από τον αριθμό των αντικειμένων (με βάση τη συνθήκη στη συνάρτηση kmeans_gpu).

3.3 Transpose Version

Σε αυτή την έκδοση αλλάζουμε την δομή των δεδομένων από row-based σε column-based indexing. Ακολουθεί ο κώδικας των συναρτήσεων euclid_dist_2, find_nearest_cluster και kmeans_gpu:

euclid_dist_2

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

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

    return(ans);
}
```

find_nearest_cluster

```
__global__ static
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters,
                          float *objects,           //  [numCoords][numObjs]
                          float *deviceClusters,    //  [numCoords]
                          [numClusters]
                          int *membership,          //  [numObjs]
                          float *devdelta)
{

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

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

```
        /* find the cluster id that has min distance to object */
        index = 0;
        /* DONE: call min_dist = euclid_dist_2(...) with correct
        objectId/clusterId */
        min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
        objects, deviceClusters, tid, 0);

        for (i=1; i<numClusters; i++) {
            /* DONE: call dist = euclid_dist_2(...) with correct
            objectId/clusterId */
            dist = euclid_dist_2_transpose(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 (membership[tid] != index) {
            /* DONE: Maybe something is missing here... is this write safe? */
            atomicAdd(devdelta, 1.0);
        }

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

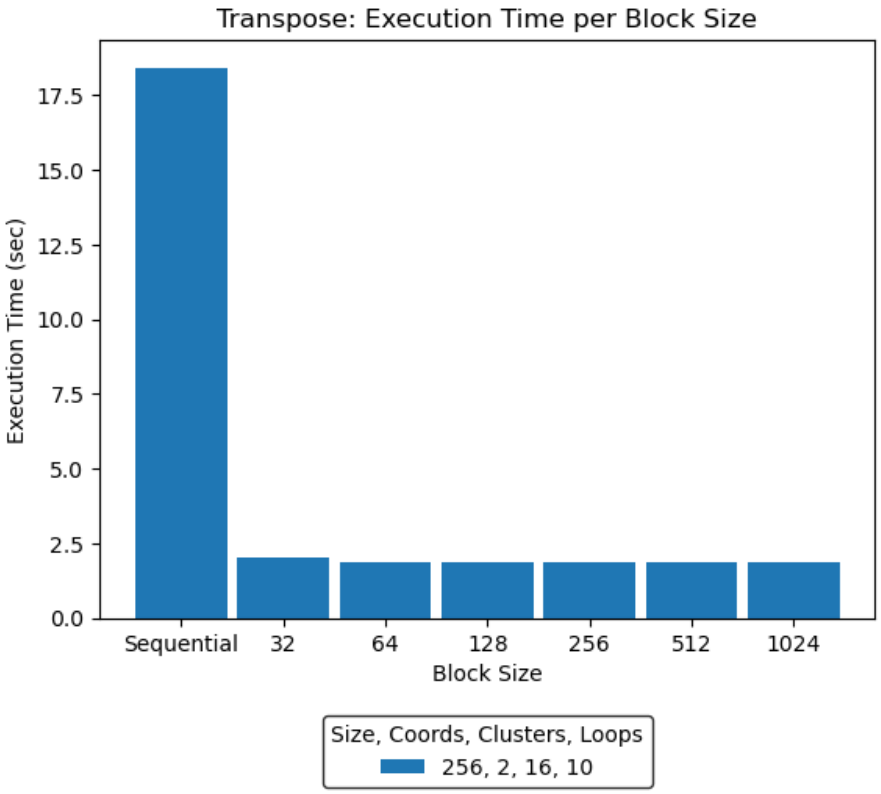
kmeans_gpu

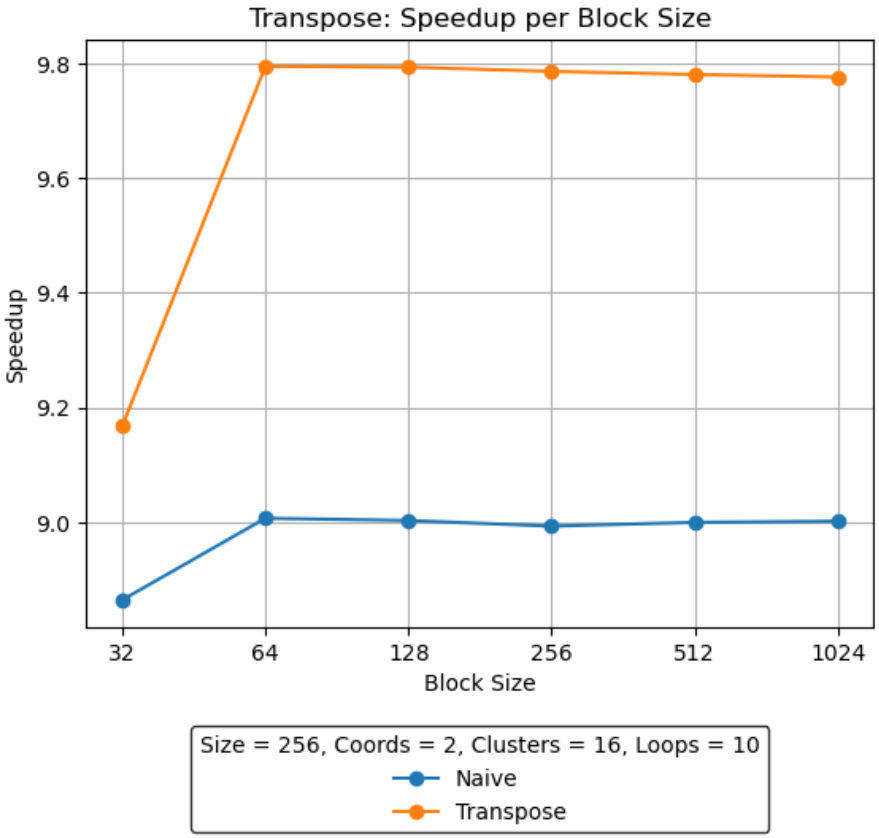
```
void kmeans_gpu(float *objects, /* in: [numObjs][numCoords] */
                int numCoords, /* no. features */
                int numObjs, /* no. objects */
                int numClusters, /* no. clusters */
                float threshold, /* % objects change membership */
                long loop_threshold, /* maximum number of iterations */
                int *membership, /* out: [numObjs] */
                float * clusters,   /* out: [numClusters][numCoords] */
                int blockSize)
{

    ...

    /* DONE: Transpose dims */
    float  **dimObjects = (float**) calloc_2d(numCoords, numObjs,
    sizeof(float));
    float  **dimClusters = (float**) calloc_2d(numCoords, numClusters,
    sizeof(float));
    float  **newClusters = (float**) calloc_2d(numCoords, numClusters,
    sizeof(float));
```

```
...

printf("\n|-----------Transpose GPU Kmeans-----------|\n\n");

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

...

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

...

do {
    timing_internal = wtime();

    /* GPU part: calculate new memberships */

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

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

     ...

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

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

    /* CPU part: Update cluster centers*/

   ...

} while (delta > threshold && loop < loop_threshold);
/* DONE: 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[numCoords * i + j] = dimClusters[j][i];
        }
    }

    ...

    return;
}
```

Ακολουθεί το διάγραμμα χρόνου εκτέλεσης (bar plot) και το αντίστοιχο διάγραμμα speedup για το configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10}:





Και πάλι παρατηρούμε ότι το speedup δεν επηρεάζεται από το block size για τους ίδιους λόγους που προαναφέρθηκαν και στην έκδοση Naive. Η μεγαλύτερη απόκλιση παρουσιάζεται για block size 32 αλλά και πάλι είναι αμελητέα (διαφορά τάξεως 0.6). Συγκριτικά με την έκδοση Naive οι χρόνοι εκτέλεσης και συνεπώς το speedup παρουσιάζουν βελτίωση. Αυτό οφείλεται στο γεγονός ότι τα νήματα οργανώνονται σε blocks με κοινή μνήμη συνεπώς, η οργάνωση των δεδομένων σε στήλες έχει ως αποτέλεσμα να περιέχονται σε μία cache line περισσότερα στοιχεία τα οποία επεξεργάζονται τα νήματα ενός block. Δηλαδή, ενώ ενδέχεται να μην περιέχονται όλες οι συντεταγμένες ενός στοιχείου στην ίδια cache line, θα περιέχονται δεδομένα για διαδοχικά στοιχεία και εφόσον διαδοχικά νήματα επεξεργάζονται και διαδοχικά σημεία είναι προφανές ότι θα έχουμε περισσότερα cache hits και συνεπώς καλύτερη επίδοση με column-based indexing.

3.4 Shared Version

Σε αυτή την έκδοση τοποθετούμε τα clusters στην διαμοιραζόμενη μνήμη της GPU ώστε να μπορούν τα στοιχεία κάθε block να τα κάνουν access πιο γρήγορα. Ακολουθεί ο κώδικας των συναρτήσεων euclid_dist_2, find_nearest_cluster και kmeans_gpu:

euclid_dist_2

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

```
{
    int i;
    float ans=0.0;

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

    return(ans);
}
```

find_nearest_cluster

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

    /* DONE: Copy deviceClusters to shmemClusters so they can be accessed faster.
        BEWARE: Make sure operations is complete before any thread continues... */
    int j;
    if (threadIdx.x == 0) {
        for (j = 0; j < numClusters * numCoords; j++)
            shmemClusters[j] = deviceClusters[j];
    }
    __syncthreads();

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

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

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

        /* DONE: 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++) {
            /* DONE: 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) {
        /* DONE: Maybe something is missing here... is this write safe? */
            atomicAdd(devdelta, 1.0);
        }

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

        }
}
```

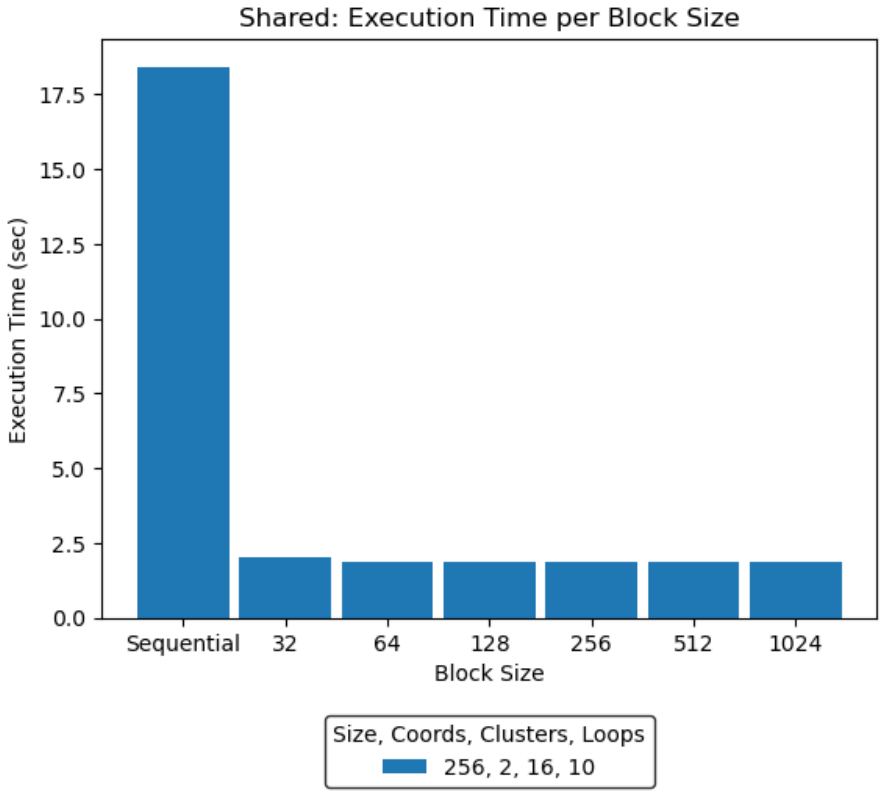
kmeans_gpu

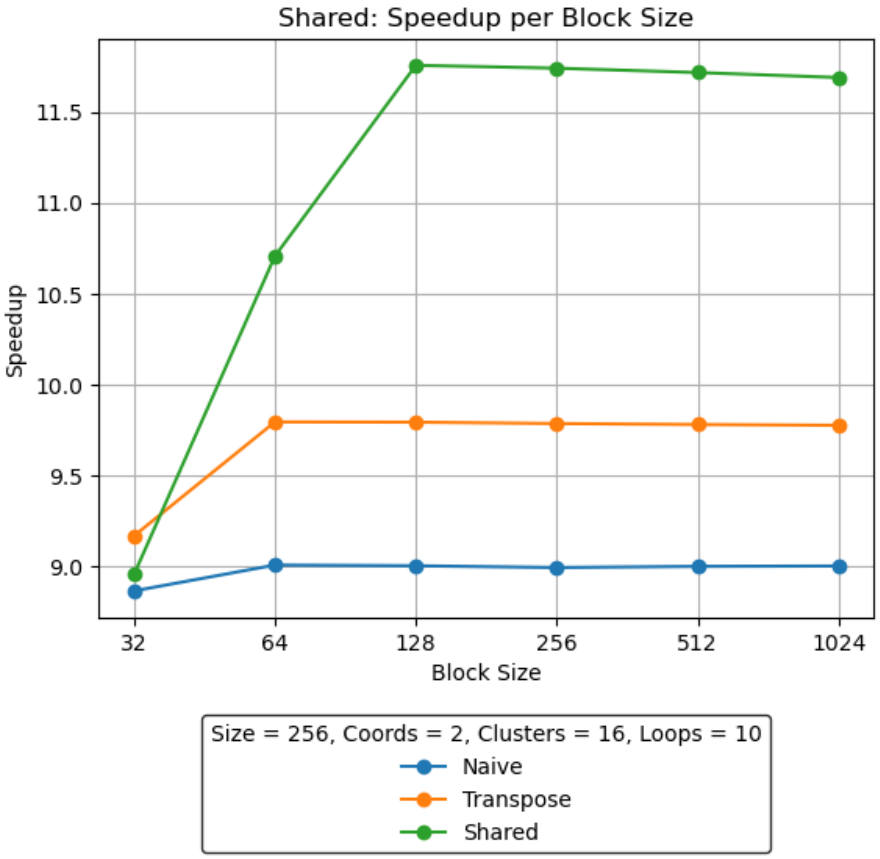
```
void kmeans_gpu(float *objects,      /* in: [numObjs][numCoords] */
                int numCoords,    /* no. features */
                int numObjs,      /* no. objects */
                int numClusters,  /* no. clusters */
                float threshold,    /* % objects change membership */
                long loop_threshold,   /* maximum number of iterations */
                int *membership,   /* out: [numObjs] */
                float * clusters,   /* out: [numClusters][numCoords] */
                int blockSize)
{

    ...

    /* DONE: Transpose dims */
    float  **dimObjects = (float**) calloc_2d(numCoords, numObjs,
    sizeof(float));
    float  **dimClusters = (float**) calloc_2d(numCoords, numClusters,
    sizeof(float));
    float  **newClusters = (float**) calloc_2d(numCoords, numClusters,
    sizeof(float));

    ...

    printf("\n|-----------Shared GPU Kmeans------------|\n\n");
```

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

...

const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize)?
blockSize: numObjs;
/* DONE: 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 = numCoords *
numClusters * sizeof(float);

...

do {
    timing_internal = wtime();

    /* GPU part: calculate new memberships */

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

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

     ...

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

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

    /* CPU part: Update cluster centers*/

    ...
```

```
        /* DONE: 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[numCoords * i + j] = dimClusters[j][i];
            }
        }

    ...

    return;
}
```

Ακολουθεί το διάγραμμα χρόνου εκτέλεσης (bar plot) και το αντίστοιχο διάγραμμα speedup για το configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10}:

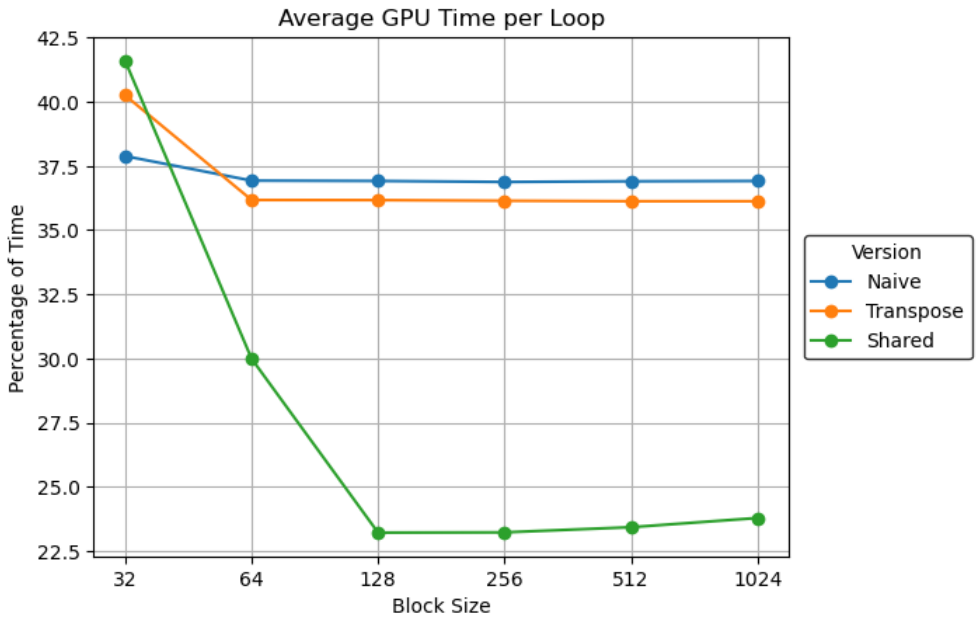




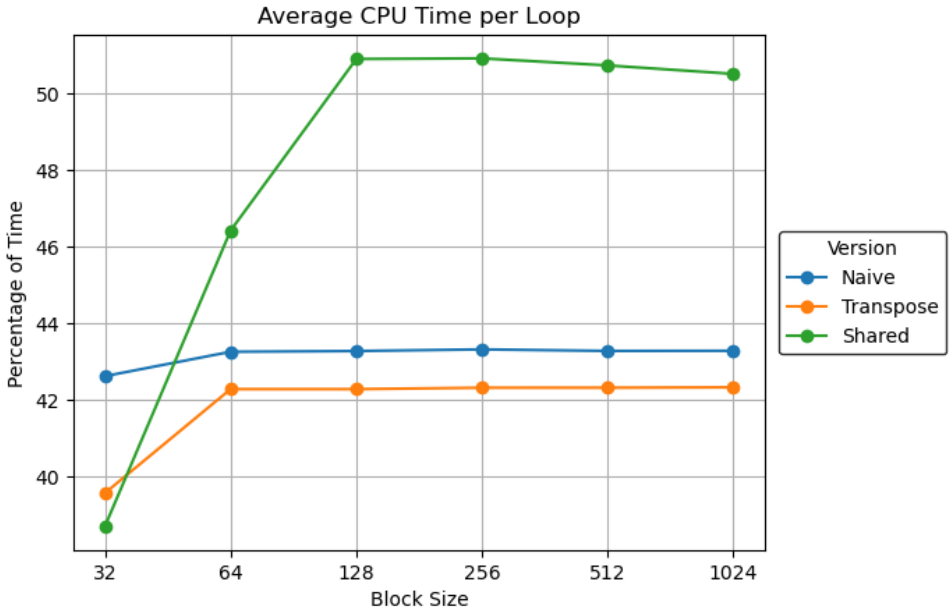
Στην υλοποίηση Shared παρατηρούμε βελτίωση της επίδοσης και συνεπώς του speedup σε σχέση με τις εκδόσεις Naive, Transpose καθώς, οι συστάδες τοποθετούνται στην διαμοιραζόμενη μνήμη της GPU και συνεπώς λόγω της εγγύτητας τους οι πρόσβαση στα δεδομένα είναι πιο γρήγορη. Παρατηρούμε ότι το block size έχει μεγαλύτερη επίδραση στην επίδοση σε σχέση με τις προηγούμενες υλοποιήσεις. Το speedup αυξάνεται με την αύξηση του μεγέθους block από 32 σε 64 και 128 και μετά σταθεροποιείται με ελαφριά μειωτική τάση. Για μικρά μεγέθη block δεν αξιοποιείται επαρκώς η κοινή μνήμη καθώς, αφού διαδοχικά σημεία του πίνακα επεξεργάζονται από διαδοχικά νήματα, υπάρχει ανταγωνισμός στην κοινή μνήμη μεταξύ των νημάτων διότι είναι πιο πιθανό να εργάζονται στην ίδια σελίδα μνήμης. Συνεπώς, εισάγονται καθυστερήσεις λόγω κλειδωμάτων και cache invalidations. Η μικρή μειωτική τάση που παρατηρούμε για μεγαλύτερο μέγεθος block οφείλεται στο γεγονός ότι η αύξηση των threads ανά block χωρίς να αυξάνεται και το μέγεθος της μνήμης δημιουργεί και πάλι ανταγωνισμό.

3.5 Σύγκριση Υλοποίησεων (Bottleneck Analysis)

3.5.1 Μέσος Χρόνος GPU ανά Επανάληψη

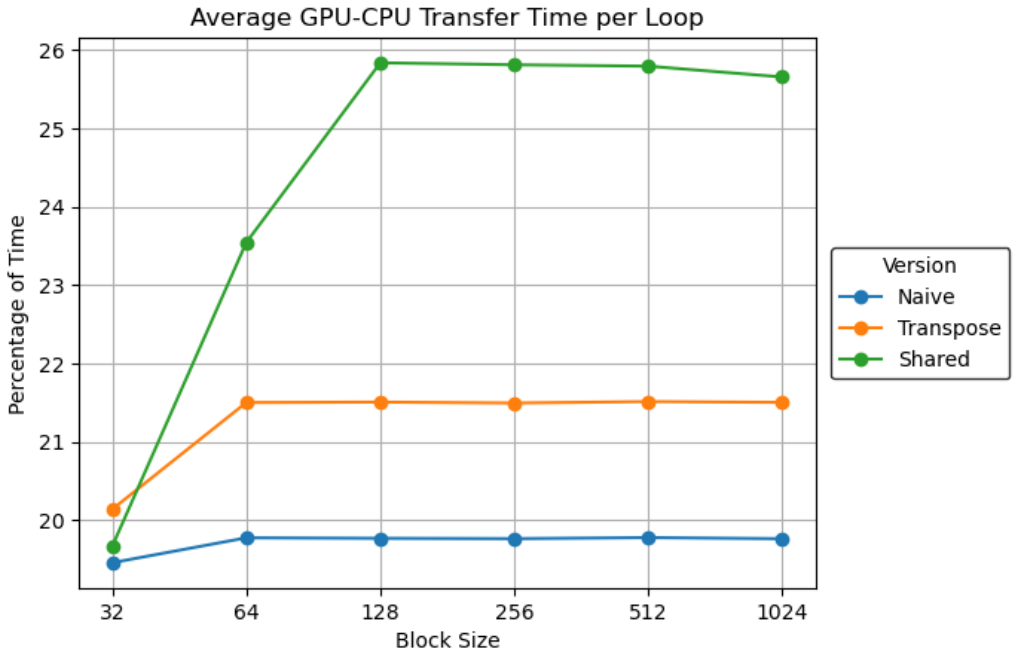


3.5.2 Μέσος Χρόνος CPU ανά Επανάληψη

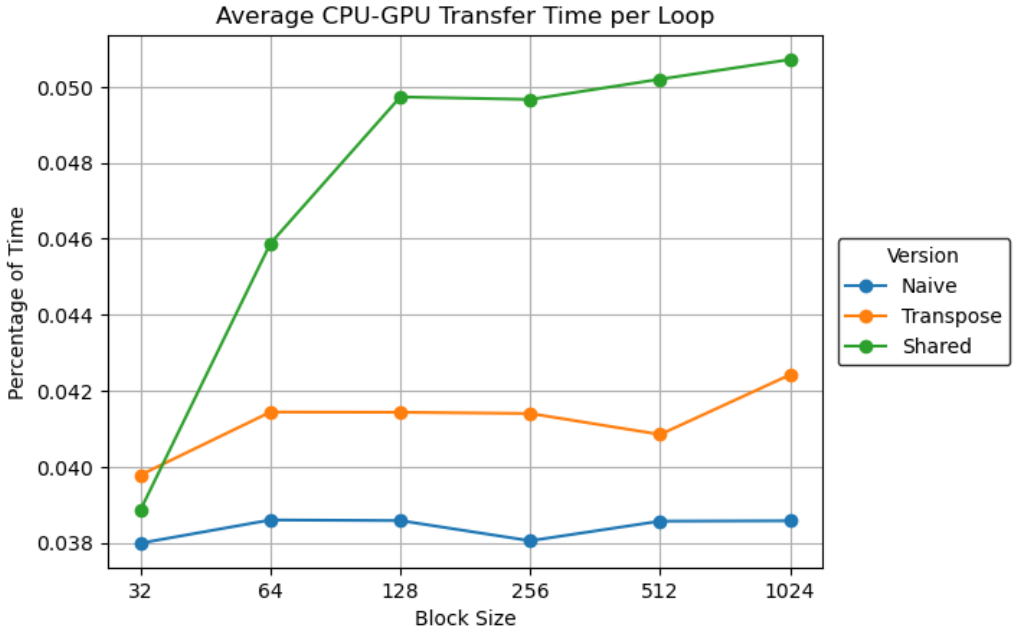


Για τις εκδόσεις Naive, Transpose παρατηρούμε ότι το block size δεν επηρεάζει σημαντικά το ποσοστό του υπολογιστικού χρόνου στην CPU ή την GPU καθώς, ο αριθμός των νημάτων παραμένει σταθερός. Στην έκδοση Shared, το ποσοστό χρόνου στην GPU μειώνεται με την αύξηση του block size από 32 καθώς, όπως έχει προαναφερθεί, με μεγαλύτερο block size γίνεται πιο αποδοτική χρήση της μοιραζόμενης μνήμης αφού σε μία cache line θα περιέχεται χρήσιμη πληροφορία για περισσότερα νήματα. Η αύξηση του block size σε 512, 1024 έχει ως αποτέλεσμα μικρή αύξηση του ποσοστού χρόνου σε σχέση με αυτό των block size 128, 256. Αυτό οφείλεται στο γεγονός ότι η αύξηση των νημάτων ανά block προκαλεί συμφόρηση στον δίαυλο δεδομένων της διαμοιραζόμενης μνήμης του κάθε block της GPU. Ο χρόνος επεξεργασίας στην CPU δεν επηρεάζεται με την αλλαγή του block size, μειώνεται όμως ο συνολικός χρόνος επεξεργασίας συνεπώς το ποσοστό χρόνου στην CPU αυξάνεται.

3.5.3 Μέσος Χρόνος Μεταφορών GPU-CPU ανά Επανάληψη

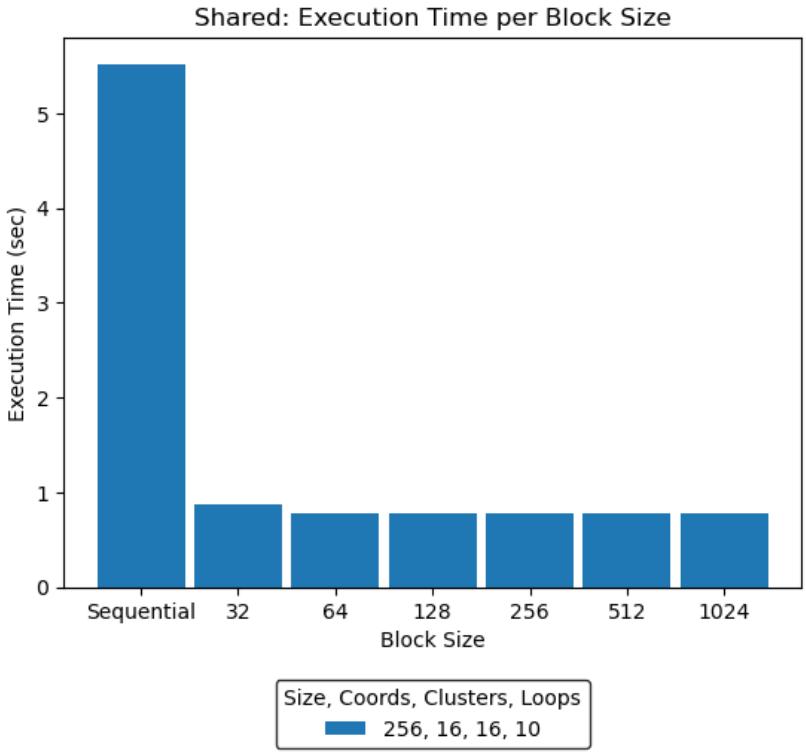


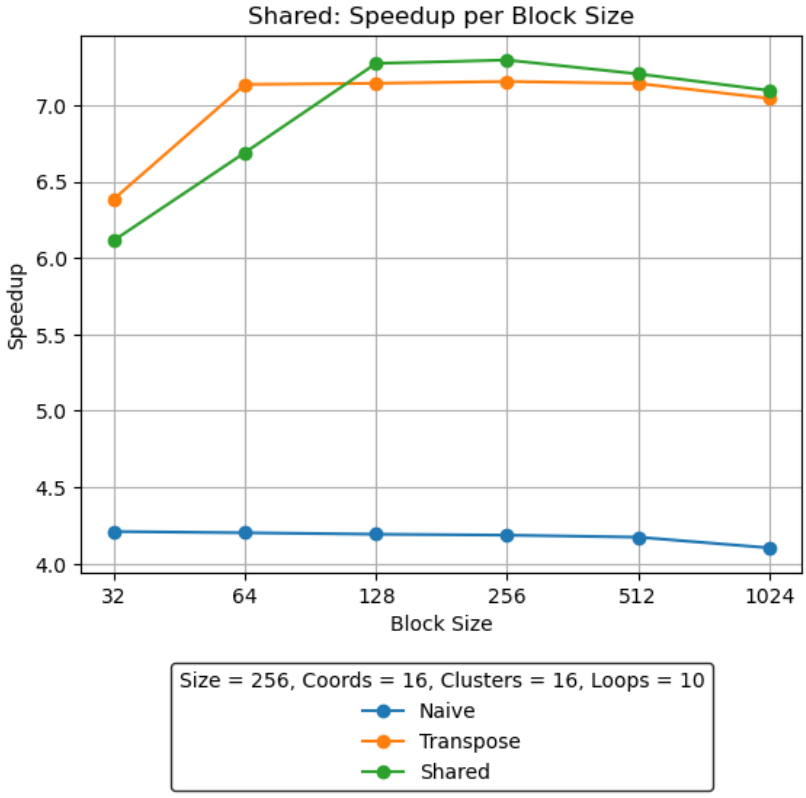
3.5.4 Μέσος Χρόνος Μεταφορών CPU-GPU ανά Επανάληψη



Σε όλες τις εκδόσεις ο χρόνος μεταφορών CPU-GPU και GPU-CPU παραμένει σταθερός. Στην έκδοση Shared αυξάνεται το ποσοστό του χρόνου που καταναλώνεται σε μεταφορές καθώς ο χρόνος επεξεργασίας στην GPU, και συνεπώς ο συνολικός χρόνος ανά επανάληψη μειώνεται. Οι μεταφορές CPU-GPU καταναλώνουν πολύ μικρότερο ποσοστό χρόνου καθώς αντιγράφεται μόνο ο πίνακας συστάδων σε αντίθεση με τις μεταφορές GPU-CPU όπου αντιγράφεται ο πολύ μεγαλύτερων διαστάσεων πίνακας αντικειμένων.
Από τις παραπάνω παρατηρήσεις συμπεραίνουμε ότι η συνολική επίδοση του iterative μέρους εμποδίζεται κυρίως από τον χρόνο επεξεργασίας CPU ο οποίος παραμένει σταθερός και από τις μεταφορές GPU-CPU του πίνακα αντικειμένων.

3.5.5 Configuration {256, 16, 16, 10}





Παρατηρούμε ότι το speedup είναι αρκετά χαμηλότερο για τη shared εκδοχή του αλγορίθμου. Αυτό συμβαίνει γιατί στην υλοποίησή μας κάθε νήμα υπολογίζει την ευκλείδεια απόσταση ανάμεσα σε δύο σημεία κάνοντας loop σε όλες τις συντεταγμένες. Στην αρχική εκτέλεση είχαμε μόνο 2 συντεταγμένες, ενώ τώρα 10, γεγονός που προκαλεί τη μείωση του speedup. Άρα, η παρούσα shared υλοποίηση δεν είναι κατάλληλη για την επίλυση του kmeans για arbitrary configurations.

4 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

4.1 Σκοπός της Άσκησης

Σκοπός της άσκησης είναι η παραλληλοποίηση των αλγορίθμων εξίσωσης θερμότητας Jacobi και Gauss-Seidel SOR σε αρχιτεκτονική κατανεμημένης μνήμης, χρησιμοποιώντας το προγραμματιστικό μοντέλο ανταλλαγής μηνυμάτων MPI. Ακολουθεί ο κώδικας των δύο υλοποιήσεων.

4.1.1 Jacobi

Στον αλγόριθμο Jacobi, για τον υπολογισμό ενός στοιχείου απαιτούνται τα γειτονικά στοιχεία του προηγούμενου χρονικού βήματος, που βρίσκονται πάνω, κάτω, δεξιά και αριστερά.

```
...

int main(int argc, char ** argv) {

    ...

    //----Rank 0 defines positions and counts of local blocks
    ↪ (2D-subdomains) on global matrix----//
    int * scatteroffset, * scattercounts;
    if (rank==0) {
        ...
    }

    //----Rank 0 scatters the global matrix----//
    //*************DONE*******************//
    MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block,
    ↪ &(u_previous[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
    MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block,
    ↪ &(u_current[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
    //************************************//

    if (rank==0)
        free2d(U);

    //----Define datatypes or allocate buffers for message passing----//
    //*************DONE*******************//
    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);
    //************************************//


    //----Find the 4 neighbors with which a process exchanges messages----//
```

```
    //************DONE*****************//
    int north, south, east, west;

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

    //---Define the iteration ranges per process-----//
    //************DONE*****************//
    int i_min,i_max,j_min,j_max;

    /* internal processes */
    i_min = 1;
    i_max = local[0];

    j_min = 1;
    j_max = local[1];

    /* boundary processes */
    if (north == MPI_PROC_NULL) {
        i_min = 2;
    }

    if (south == MPI_PROC_NULL) {
        i_max -= 1 + (global_padded[0]-global[0]);
        /* if there is no padding global_padded[0]-global[0] = 0 */
    }

    if (west == MPI_PROC_NULL) {
        j_min = 2;
    }

    if (east == MPI_PROC_NULL) {
        j_max -= 1 + (global_padded[1]-global[1]);
        /* if there is no padding global_padded[1]-global[1] = 0 */
    }
    //**********************************//

    //----Computational core----//
    gettimeofday(&tts, NULL);
    #ifdef 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

        //************DONE*****************//
        int requests_count = 0;
        MPI_Request requests[8];
```

```
        MPI_Status statuses[8];

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

        /* Communicate */
        gettimeofday(&tms,NULL); /* Messaging timer */
        if (north != MPI_PROC_NULL) {
            MPI_Isend(&(u_previous[1][1]), 1, row, north, 0, MPI_COMM_WORLD,
↪   &requests[requests_count++]);
            MPI_Irecv(&(u_previous[0][1]), 1, row, north, 1, MPI_COMM_WORLD,
↪   &requests[requests_count++]);
        }

        if (south != MPI_PROC_NULL) {
            MPI_Isend(&(u_previous[local[0]][1]), 1, row, south, 1,
↪   MPI_COMM_WORLD, &requests[requests_count++]);
            MPI_Irecv(&(u_previous[local[0]+1][1]), 1, row, south, 0,
↪   MPI_COMM_WORLD, &requests[requests_count++]);
        }

        if (west != MPI_PROC_NULL) {
            MPI_Isend(&(u_previous[1][1]), 1, column, west, 2, MPI_COMM_WORLD,
↪   &requests[requests_count++]);
            MPI_Irecv(&(u_previous[1][0]), 1, column, west, 3, MPI_COMM_WORLD,
↪   &requests[requests_count++]);
        }

        if (east != MPI_PROC_NULL) {
            MPI_Isend(&(u_previous[1][local[1]]), 1, column, east, 3,
↪   MPI_COMM_WORLD, &requests[requests_count++]);
            MPI_Irecv(&(u_previous[1][local[1]+1]), 1, column, east, 2,
↪   MPI_COMM_WORLD, &requests[requests_count++]);
        }

        MPI_Waitall(requests_count, requests, statuses);
        gettimeofday(&tmf,NULL);
        tmsg+=(tmf.tv_sec-tms.tv_sec)+(tmf.tv_usec-tms.tv_usec)*0.000001;

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

        #ifdef TEST_CONV
        if (t%C==0) {
            //*************DONE**************//
```

```
            /*Test convergence*/
            converged = converge(u_previous, u_current, i_min, i_max, j_min,
↪   j_max);
            MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN,
↪   MPI_COMM_WORLD);
        }
        #endif
        //*************************************//

    }
    gettimeofday(&ttf,NULL);

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

    MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    MPI_Reduce(&tmsg,&msg_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]);
    }

    //*************DONE*******************//
    MPI_Gatherv(&(u_previous[1][1]), 1, local_block, &(U[0][0]),
↪   scattercounts, scatteroffset, global_block, 0, MPI_COMM_WORLD);
    //*************************************//

    ...

}
```

4.1.2 Gauss-Seidel SOR

Στον αλγόριθμο Gauss-Seidel SOR, για τον υπολογισμό ενός στοιχείου απαιτούνται τα γειτονικά στοιχεία του προηγούμενου χρονικού βήματος, που βρίσκονται κάτω, δεξιά και τα γειτονικά στοιχεία του τρέχοντος χρονικού βήματος, που βρίσκονται πάνω, αριστερά. Ακόμη, απαιτείται η τιμή από το προηγούμενο χρονικό βήμα του ίδιου του στοιχείου.

```
...

int main(int argc, char ** argv) {

    ...

    //----Rank 0 defines positions and counts of local blocks
↪   (2D-subdomains) on global matrix----//
    int * scatteroffset, * scattercounts;
    if (rank==0) {
        ...
    }

    //----Rank 0 scatters the global matrix----//
```

```
    //************DONE*****************//
    MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block,
    &(u_previous[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
    MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block,
    &(u_current[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
    //************************************//

    if (rank==0)
        free2d(U);

    //----Define datatypes or allocate buffers for message passing----//
    //************DONE*****************//
    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);
    //************************************//


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

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

    //---Define the iteration ranges per process-----//
    //************DONE*****************//
    int i_min,i_max,j_min,j_max;

    /* internal processes */
    i_min = 1;
    i_max = local[0];

    j_min = 1;
    j_max = local[1];

    /* boundary processes */
    if (north == MPI_PROC_NULL) {
        i_min = 2;
    }

    if (south == MPI_PROC_NULL) {
        i_max -= 1 + (global_padded[0]-global[0]);
        /* if there is no padding global_padded[0]-global[0] = 0 */
    }
```

```
    if (west == MPI_PROC_NULL) {
        j_min = 2;
    }

    if (east == MPI_PROC_NULL) {
        j_max -= 1 + (global_padded[1]-global[1]);
        /* if there is no padding global_padded[1]-global[1] = 0 */
    }
    //*************************************//

    //----Computational core----//
    gettimeofday(&tts, NULL);
    #ifdef 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

        //*************DONE*******************//
        int prev_req_count = 0, curr_req_count = 0;
        MPI_Request prev_requests[6], curr_requests[2];
        MPI_Status prev_statuses[6], curr_statuses[2];

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

        /* Communicate */
        gettimeofday(&tms,NULL); /* Messaging timer */
        if (north != MPI_PROC_NULL) {
            MPI_Isend(&(u_previous[1][1]), 1, row, north, 0, MPI_COMM_WORLD,
↪   &prev_requests[prev_req_count++]);
            MPI_Irecv(&(u_current[0][1]), 1, row, north, 1, MPI_COMM_WORLD,
↪   &prev_requests[prev_req_count++]);
        }

        if (south != MPI_PROC_NULL) {
            MPI_Irecv(&(u_previous[local[0]+1][1]), 1, row, south, 0,
↪   MPI_COMM_WORLD, &prev_requests[prev_req_count++]);
        }

        if (west != MPI_PROC_NULL) {
            MPI_Isend(&(u_previous[1][1]), 1, column, west, 2, MPI_COMM_WORLD,
↪   &prev_requests[prev_req_count++]);
            MPI_Irecv(&(u_current[1][0]), 1, column, west, 3, MPI_COMM_WORLD,
↪   &prev_requests[prev_req_count++]);
        }
```

```
        if (east != MPI_PROC_NULL) {
            MPI_Irecv(&(u_previous[1][local[1]+1]), 1, column, east, 2,
↪   MPI_COMM_WORLD, &prev_requests[prev_req_count++]);
        }

        MPI_Waitall(prev_req_count, prev_requests, prev_statuses);
        gettimeofday(&tmf,NULL);
        tmsg+=(tmf.tv_sec-tms.tv_sec)+(tmf.tv_usec-tms.tv_usec)*0.000001;


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


        /* Communicate */
        gettimeofday(&tms,NULL); /* Messaging timer */
        if (south != MPI_PROC_NULL) {
            MPI_Isend(&(u_current[local[0]][1]), 1, row, south, 1,
↪   MPI_COMM_WORLD, &curr_requests[curr_req_count++]);
        }

        if (east != MPI_PROC_NULL) {
            MPI_Isend(&(u_current[1][local[1]]), 1, column, east, 3,
↪   MPI_COMM_WORLD, &curr_requests[curr_req_count++]);
        }

        MPI_Waitall(curr_req_count, curr_requests, curr_statuses);
        gettimeofday(&tmf,NULL);
        tmsg+=(tmf.tv_sec-tms.tv_sec)+(tmf.tv_usec-tms.tv_usec)*0.000001;

        #ifdef TEST_CONV
        if (t%C==0) {
            //*************DONE**************//
            /*Test convergence*/
            converged = converge(u_previous, u_current, i_min, i_max, j_min,
↪   j_max);
            MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN,
↪   MPI_COMM_WORLD);
        }
        #endif
        //************************************//

    }
    gettimeofday(&ttf,NULL);
    ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;
```

```
    MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    MPI_Reduce(&tmsg,&msg_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]);
    }

    //*************DONE******************//
    MPI_Gatherv(&(u_previous[1][1]), 1, local_block, &(U[0][0]),
    scattercounts, scatteroffset, global_block, 0, MPI_COMM_WORLD);
    //************************************//

    ...

}
```

4.2 Μετρήσεις με Έλεγχο Σύγκλισης

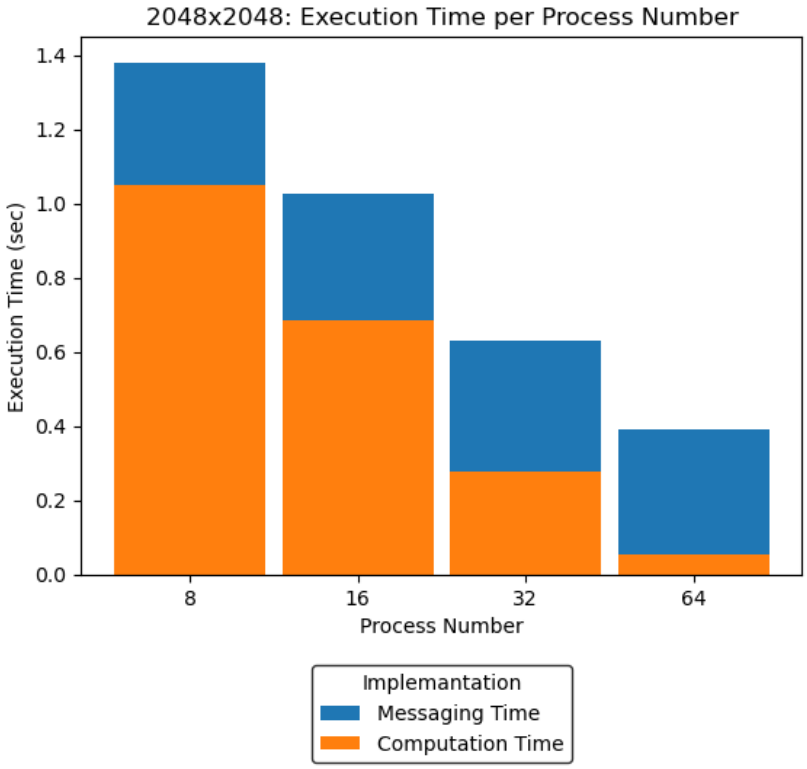
Λαμβάνουμε μετρήσεις με έλεγχο σύγκλισης για τα παράλληλα προγράμματα, για μέγεθος πίνακα 1024x1024. Όλες οι χρονικές μετήσεις είναι σε sec.

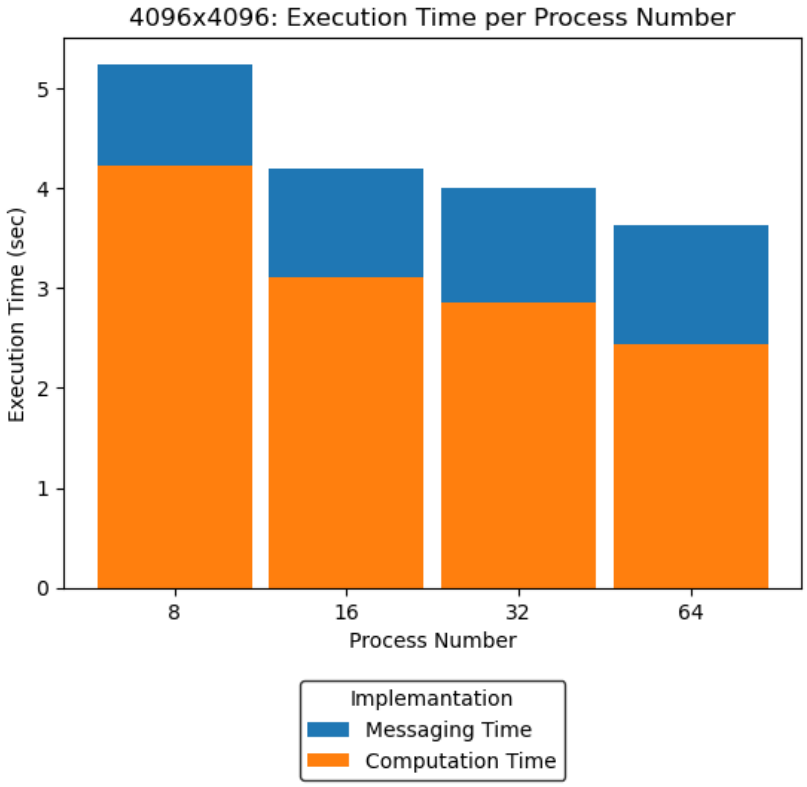
	Jacobi	**Gauss-Seidel SOR**
Επαναλήψεις	798201	3201
Συνολικός Χρόνος	222.299	2.102
Χρόνος Υπολογισμών	39.007	0.551
Χρόνος Ελέγχου Σύγκλισης	4.713	0.084
Συνολικός Χρόνος ανά Επανάληψη	0.000278501	0.000656889

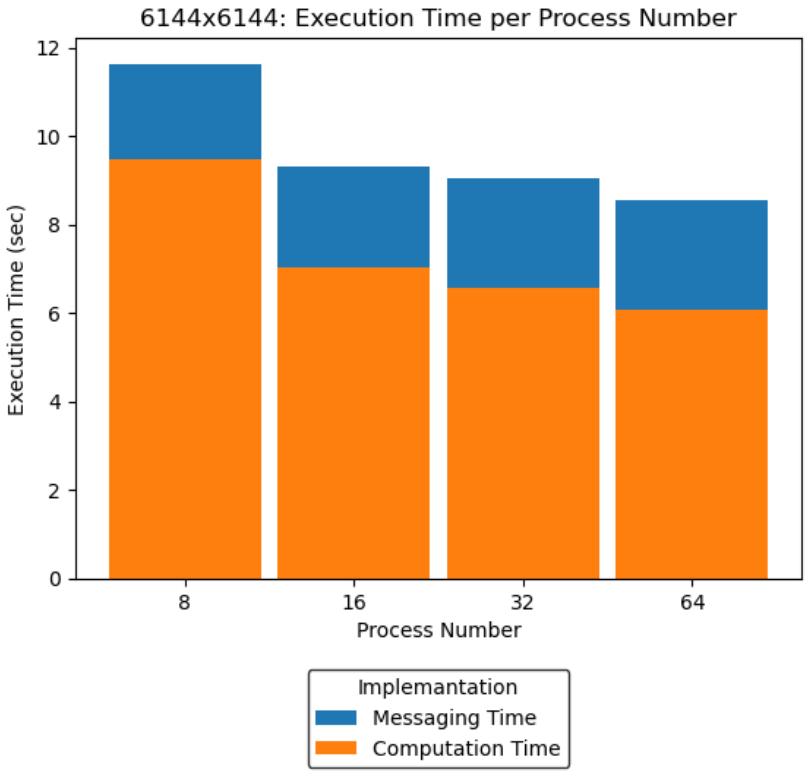
Από τις παραπάνω μετρήσεις είναι φανερό ότι, παρόλο που ο αλγόριθμος Gauss-Seidel SOR απαιτεί περισσότερο χρόνο ανά επανάληψη, συγκλίνει πολύ πιο γρήγορα από τον αλγόριθμο Jacobi και συνεπώς, θα τον επιλέγαμε για την επίλυση του προβλήματος σε ένα σύστημα κατανεμημένης μνήμης.

4.3 Μετρήσεις χωρίς Έλεγχο Σύγκλισης

4.3.1 Χρόνοι εκτέλεσης Jacobi

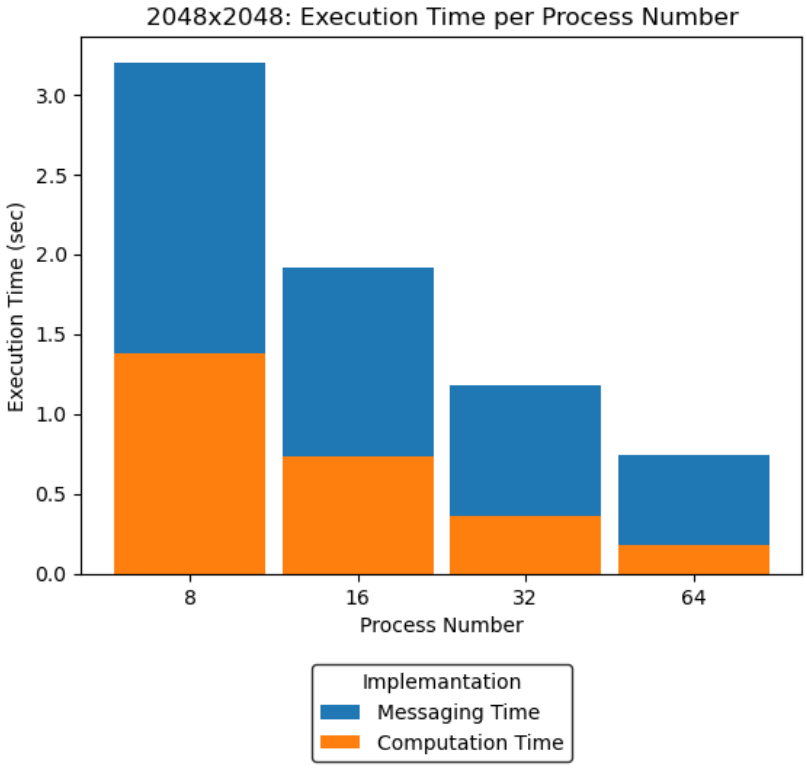


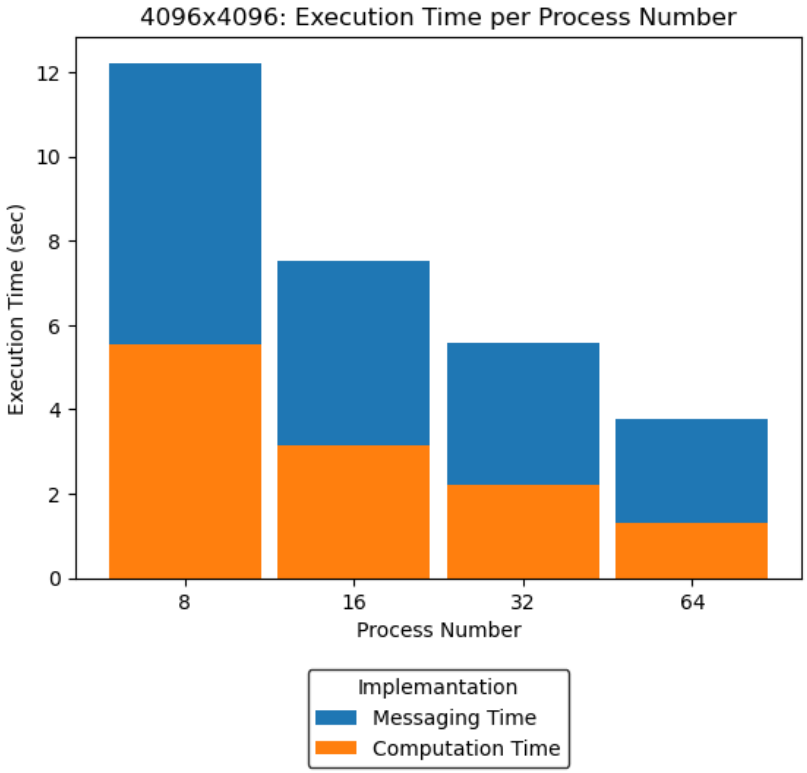


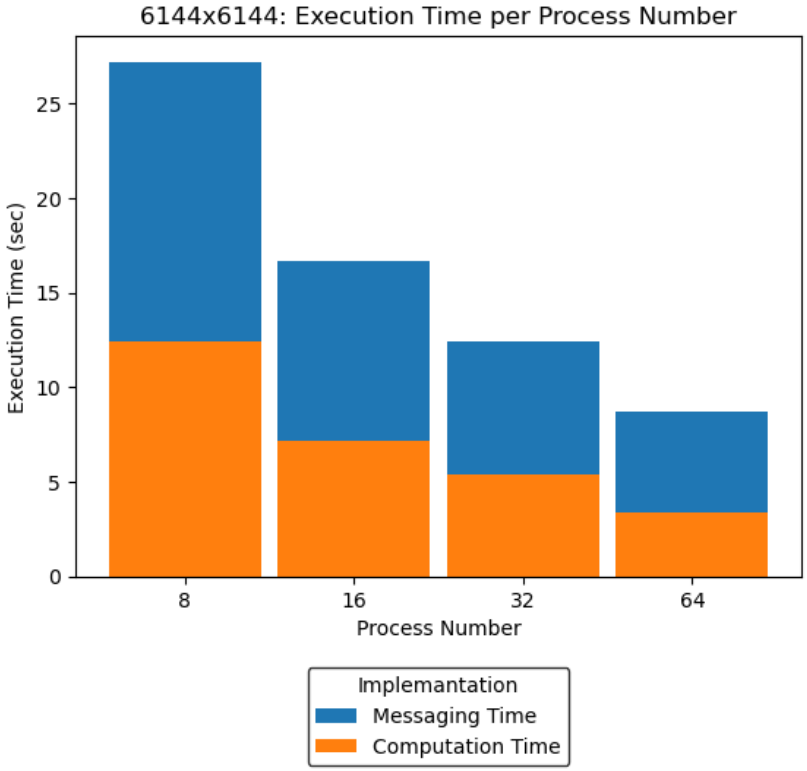


Παρατηρούμε και στα 3 διαγράμματα ότι με την αύξηση των διεργασιών μειώνεται ο χρόνος υπολογισμών και αυξάνεται ο χρόνος επικοινωνίας. Το μέγεθος πίνακα παραμένει σταθερό συνεπώς, με την αύξηση των διεργασιών, μειώνεται το μέγεθος του local block πάνω στο οποίο εργάζεται κάθε διεργασία, αυξάνεται όμως η ανάγκη για επικοινωνία αφού ο πίνακας κατακερματίζεται σε μεγαλύτερο βαθμό. Για τα μεγέθη πίνακα 4096 και 6144, με την αύξηση των διεργασιών παρατηρούμε μικρότερες μειώσεις μεταξύ των χρόνων. Αυτό οφείλεται στο γεγονός ότι για μεγαλύτερα μεγέθη πίνακα πρέπει να αυξηθούν και οι διεργασίες ώστε να δούμε πιο ουσιαστικές μειώσεις του χρόνου εκτέλεσης.

4.3.2 Χρόνοι εκτέλεσης Gauss-Seidel SOR

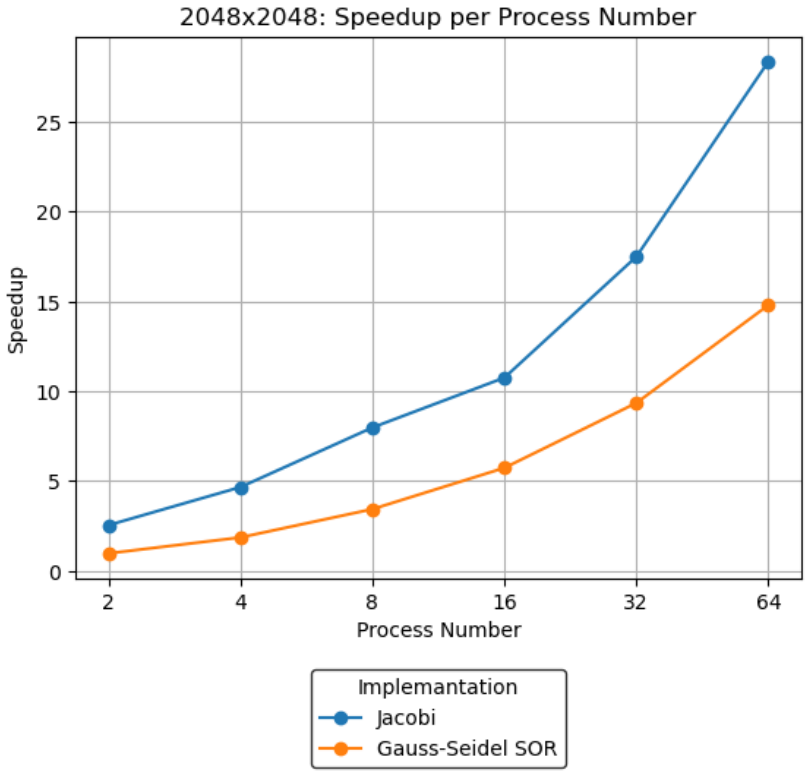


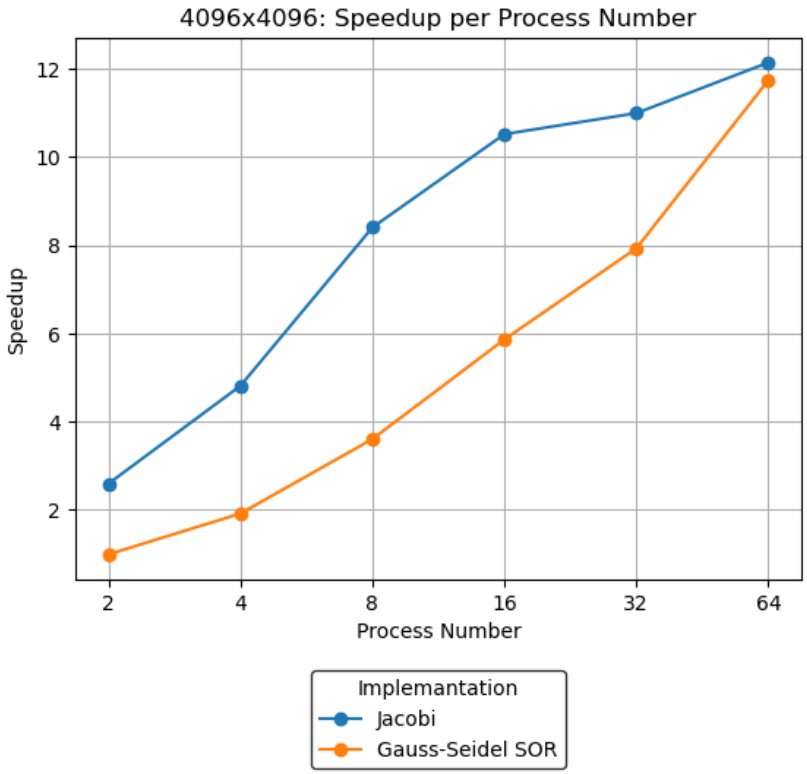


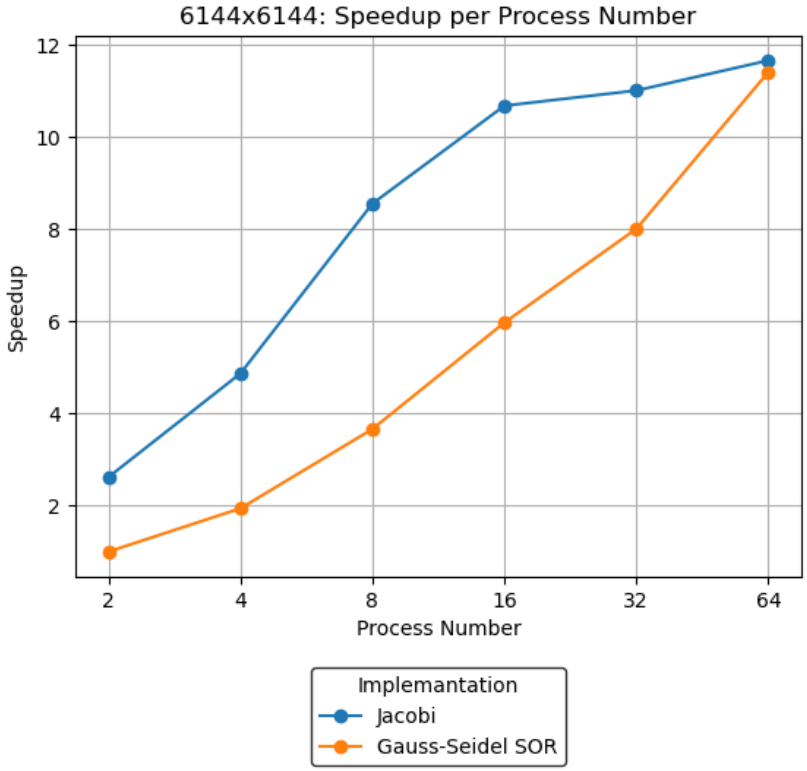


Σε αντίθεση με την υλοποίηση Jacobi, παρατηρούμε ότι για μικρότερο αριθμό εργασιών έχουμε μεγαλύτερο χρόνο επικοινωνίας. Ο αλγόριθμος Gauss-Seidel SOR εξαρτάται από τιμές του προηγούμενου αλλά και του τρέχοντος χρονικού βήματος. Ως συνέπεια, ο χρόνος επικοινωνίας μίας διεργασίας που περιμένει να λάβει δεδομένα από το τρέχων χρονικό βήμα επιβαρύνεται και από τον υπολογιστικό χρόνο της διεργασίας αποστολέα. Όπως προαναφέρθηκε, για χαμηλότερο αριθμό διεργασιών αυξάνεται το μέγεθος του local block που επεξεργάζεται κάθε διεργασία και συνεπώς και ο υπολογιστικός χρόνος που απαιτεί οπότε, με βάση τα παραπάνω αυξάνεται και ο χρόνος επικοινωνίας. Η εξάρτηση του χρόνου επικοινωνίας από τον χρόνο υπολογισμού έχει ως αποτέλεσμα πιο αισθητές μειώσεις του χρόνου εκτέλεσης ακόμα για μικρές αυξήσεις των διεργασιών συγκριτικά με το μέγεθος πίνακα, σε αντίθεση με τον αλγόριθμο Jacobi.

4.3.3 Σύγκριση speedup των δύο εκδόσεων







Παρατηρούμε ότι ο αλγόριθμος Jacobi εμφανίζει μεγαλύτερες τιμές speedup σε σχέση με τον αλγόριθμο Gauss-Seidel SOR. Με την αύξηση του μεγέθους πίνακα το scaling του Jacobi ”σπάει” (scalability break) το οποίο είναι αναμενόμενο με βάση την ανάλυση χρόνων, όπου παρατηρήσαμε ότι για μεγαλύτερα μεγέθη πίνακα χρειάζεται και μεγαλύτερη αύξηση των διεργασιών ώστε να παρατηρηθούν ουσιαστικές μειώσεις του χρόνου εκτέλεσης. Σε αντίθεση, ο αλγόριθμος Gauss-Siedel παρουσιάζει καλύτερο scaling για μεγαλύτερα μεγέθη πίνακα, το οποίο οφείλεται στην εξάρτηση του χρόνου επικοινωνίας από τον χρόνο υπολογισμού που προαναφέρθηκε. Παρόλο που οι τιμές speedup του αλγόριθμου Jacobi είναι καλύτερες, με την αύξηση του μεγέθους πίνακα μειώνονται σε αντίθεση με τον αλγόριθμο Gauss-Seidel όπου παραμένουν σχεδόν σταθερές. Συνεπώς, συμπεραίνουμε ότι ο αλγόριθμος Gauss-Siedel SOR ενδέχεται να προσφέρει καλύτερη κλιμακωσιμότητα ως προς το μέγεθος των δεδομένων εισόδου.

4.4 Γενικά συμπεράσματα

Οι παραπάνω αναλύσεις των χρόνων εκτέλεσης και του speedup των δύο αλγορίθμων επιβεβαιώνει τα αρχικά μας συμπεράσματα από τις μετρήσεις με έλεγχο σύγκλιση. Ο αλγόριθμος Jacobi εμφανίζει καλύτερους χρόνους και υψηλότερο speedup καθώς ο υπολογιστικός του πυρήνας περιέχει λιγότερες πράξεις και η επικοινωνία είναι πιο αποτελεσματική σε σχέση με τον αλγόριθμο Gauss-Seidel SOR, όπου σε ορισμένες διεργασίες παρεμβάλλονται υπολογισμοί πριν την λήψη των δεδομένων. Παρόλα αυτά, δεν μπορούμε να αγνοήσουμε το γεγονός ότι ο αλγόριθμος Gauss-Seidel SOR φτάνει σε ορθό αποτέλεσμα σε πολύ λιγότερες επαναλήψεις από ότι ο αλγόριθμος Jacobi και συνεπώς, παρά την χειρότερη επίδοση του ανά επανάληψη, αποτελεί την καλύτερη επιλογή.