Συστήματα Παράλληλης Επεξεργασίας

Εργαστηριακή Άσκηση
(Full Version)

[Το LaTeX Project εδώ]

Ομάδα 17

Δημήτριος-Δαυίδ Γεροκωνσταντής (ΑΜ : 03119209)
Αθανάσιος Τσουκλείδης-Καρυδάκης (ΑΜ : 03119009)

Χειμερινό 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή ΗΜΜΥ

Περιεχόμενα

ΜΕΡΟΣ Α

Εισαγωγική Εργαστηριακή Άσκηση

Εξοικείωση με το περιβάλλον προγραμματισμού
Παραλληλοποίηση του αλγορίθμου Game of Life

1 Εισαγωγή – Τρόπος Εργασίας

Σκοπός της παρούσας εργαστηριακής άσκησης του Μέρους Α είναι η εξοικείωση με τη χρήση των πόρων του εργαστηρίου κατασκευάζοντας, μεταγλωττίζοντας και εκτελώντας ένα παράλληλο πρόγραμμα. Αρχικά καλούμαστε να παραλληλοποιήσουμε το δοθέν σειριακό πρόγραμμα για το πρόβλημα Game of Life και στη συνέχεια να μελετήσουμε την επίδοση (χρόνο εκτέλεσης και επιτάχυνση με χρήση πολλών πυρήνων) για διάφορα configurations του προβλήματος και των χρησιμοποιού-μενων πόρων (συγκεκριμένα για 3 μεγέθη εισόδου $N = \{64, 1024, 4096\}$ και για αριθμό πυρήνων $\{1, 2, 4, 6, 8\}$).

1.1 Εντοπισμός Παραλληλισμού στο Game of Life

Συνοπτικά το πρόγραμμα καλείται να διατρέξει έναν διδιάστατο πίνακα ελέγχοντας τους 8 γείτονες κάθε κελιού. Στόχος είναι η κατανομή αυτής της εργασίας σε πολλούς workers (νήματα) ώστε να απαιτηθεί μικρότερος συνολικός χρόνος εκτέλεσης. Το πρόβλημα είναι με τέτοιο τρόπο δομημένο, ώστε η κατάσταση του πίνακα κάποια χρονική στιγμή t (δηλαδή η κατάσταση – alive or dead - καθενός κελιού την t) να εξαρτάται από την κατάστασή του την ακριβώς προηγούμενη χρονική στιγμή (δηλαδή την κατάσταση των 8 γειτόνων κάθε κελιού την προηγούμενη χρονική στιγμή). Αυτή η εξάρτηση μεταξύ διαδοχικών χρονικών στιγμών καθιστά μη ωφέλιμη την παραλληλοποίηση της συνολικής εργασίας σε επίπεδο χρονικών βημάτων (δηλαδή ανάθεση του υπολογισμού της κατάστασης του πίνακα διαφορετικών χρονικών στιγμών σε διαφορετικά νήματα) καθώς το υποτιθέμενο νήμα που θα επιφορτιζόταν με την χρονική στιγμή t δεν θα μπορούσε να παραλληλοποιήσει την εργασία του με αυτή του νήματος που αναλαμβάνει την στιγμή t-1 καθώς το πρώτο νήμα θα έπρεπε να περιμένει το αποτέλεσμα του δεύτερου νήματος. Συνεπώς δεν υπάρχει παραλληλισμός σε επίπεδο χρόνου. Από την άλλη, δεν υπάρχει καμία εξάρτηση μεταξύ των καταστάσεων κελιών του πίνακα την ίδια χρονική στιγμή. Δηλαδή κάθε χρονική στιγμή, ο υπολογισμός της κατάστασης ενός κελιού είναι μια εντελώς ανεξάρτητη εργασία (ως προς τις αντίστοιχες εργασίες την ίδια χρονική στιγμή). Αυτό μας οδηγεί, σε κάθε χρονική στιγμή, να μπορούμε να παραλληλοποιήσουμε τους υπολογισμούς κατάστασης κάθε κελιού μεταξύ τους. Κάποια κελιά θα ανατεθούν σε κάποια νήματα ενώ κάποια άλλα κελιά σε κάποια άλλα νήματα.

1.2 Παραλληλοποίηση Προγράμματος - Κώδικας

Σε επίπεδο κώδικα, αυτό που θέλουμε είναι η παραλληλοποίηση των δύο τελευταίων for loops (που διατρέχουν με δείκτες i, j τον $N \times N$ πίνακα). Έτσι προσθέτουμε πάνω από το δεύτερο (από τα 3 συνολικά) for loop την γραμμή κώδικα η οποία παραλληλοποιεί και τα δύο τελευταία for loops :

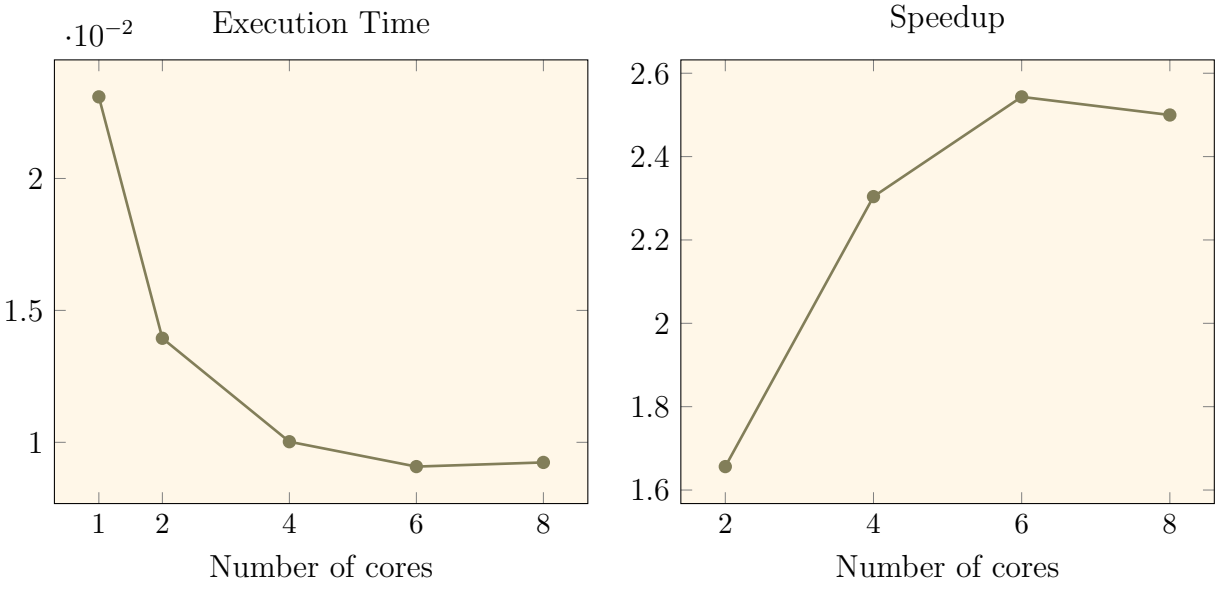
```
1    #pragma omp parallel for private(nbrs,i,j) shared(previous,current,N)
```

Επιπλέον επισημαίνουμε ότι για κάθε νήμα θα υπάρχει ένα ιδιωτικό τοπικό αντίγραφο της δυάδας (i, j) του κελιού που ανήκει σε ένα νήμα όπως και ένα τοπικό αντίγραφο της μεταβλητής nbrs ώστε κάθε νήμα να υπολογίζει τον αριθμό των γειτόνων του δικού του κελιού. Διαμοιραζόμενες θα είναι οι δομές previous, current που κρατούν την κατάσταση του πίνακα δύο διαδοχικές χρονικές στιγμές και θέλουμε όλα τα νήματα να μπορούν να διαβάσουν και να γράψουν σε αυτές όπως και η μεταβλητή N της διάστασης του πίνακα.

2 Εκτέλεση Προγράμματος – Επιδόσεις - Διαγράμματα

Στη συνέχεια υποβάλλουμε στον Torque τις εργασίες μεταγλώττισης και εκτέλεσης του προγράμματος για τα διάφορα configurations (προβλήματος και πόρων) και διαβάζουμε από τα παραγόμενα αρχεία *.out* τον απαιτούμενο χρόνο εκτέλεσης. Παρακάτω φαίνονται τα διαγράμματα που παρουσιάζουν αυτές τις επιδόσεις. Συγκεκριμένα για κάθε N (μέγεθος πίνακαπλέγματος), παρουσιάζουμε τον απαιτούμενο χρόνο εκτέλεσης καθώς και το SpeedUp[1] με χρήση 1, 2, 4, 6 ή 8 υπολογιστικών πυρήνων.

Για $N = 64$ (πλέγμα 64×64)



(a) Execution Time for N=64 (64×64)

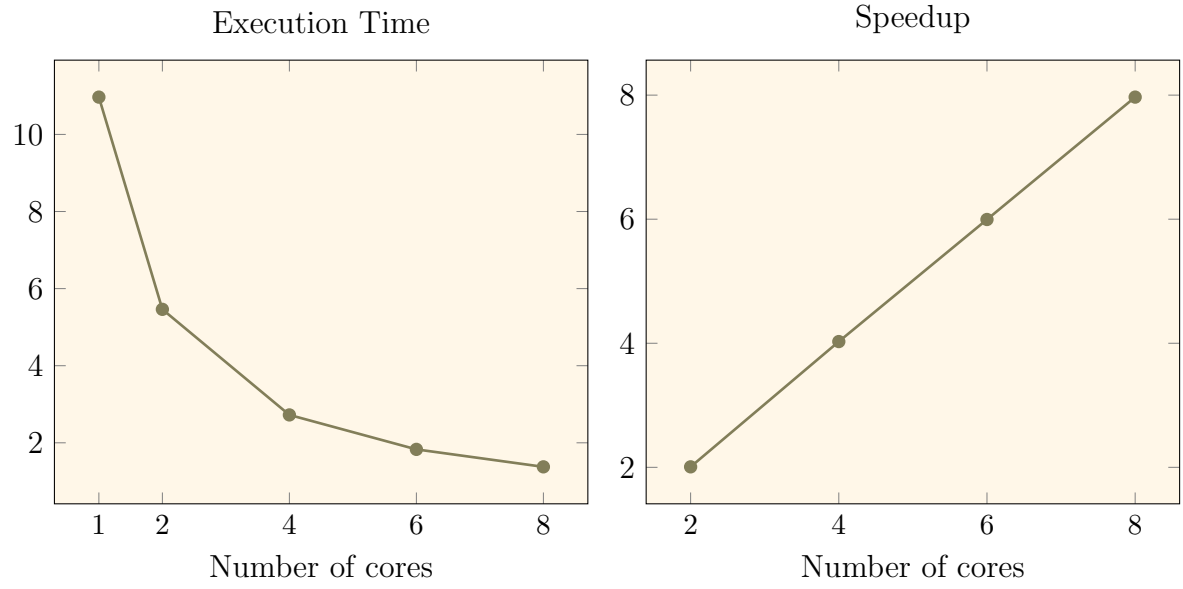
(b) Speedup for N=64 (64×64)

Εικόνα 1: Χρόνος εκτέλεσης και Speedup για $N = 64$.

Παρατηρούμε ότι εν γένει (μέχρι τα 6 cores), η επίδοση του προγράμματος βελτιώνεται με χρήση περισσότερων πυρήνων, αλλά με φθίνοντα ρυθμό. Δηλαδή, η βελτίωση από τους 4 στους 6 πυρήνες είναι αρκετά μικρότερη από την βελτίωση από τους 2 στους 4 (φαίνεται και από την κλίση στο διάγραμμα speedup). Με αύξηση από τα 6 στα 8 cores παρατηρούμε ότι η επίδοση επιδεινώνεται. Το μέγεθος του προβλήματος είναι αρκετά μικρό με N=64 και τελικά το διαχειρι-στικό overhead (δημιουργίας/join κλπ) των νημάτων είναι αρκετά μεγάλο συγκριτικά με τον υπολογιστικό φόρτο του προγράμματος. Έτσι με 8 νήματα παύουμε να έχουμε κλιμακωσιμότητα (scalability breaks).

[1] Δηλαδή τον λόγο $\frac{Execution\_Time\_of\_the\_serial\_program(1core)}{Execution\_Time\_using\_more\_cores}$ που εκφράζει το όφελος της χρήσης πολλών πυρήνων σε χρονική επιτάχυνση

Για $N = 1024$ (πλέγμα 1024×1024)



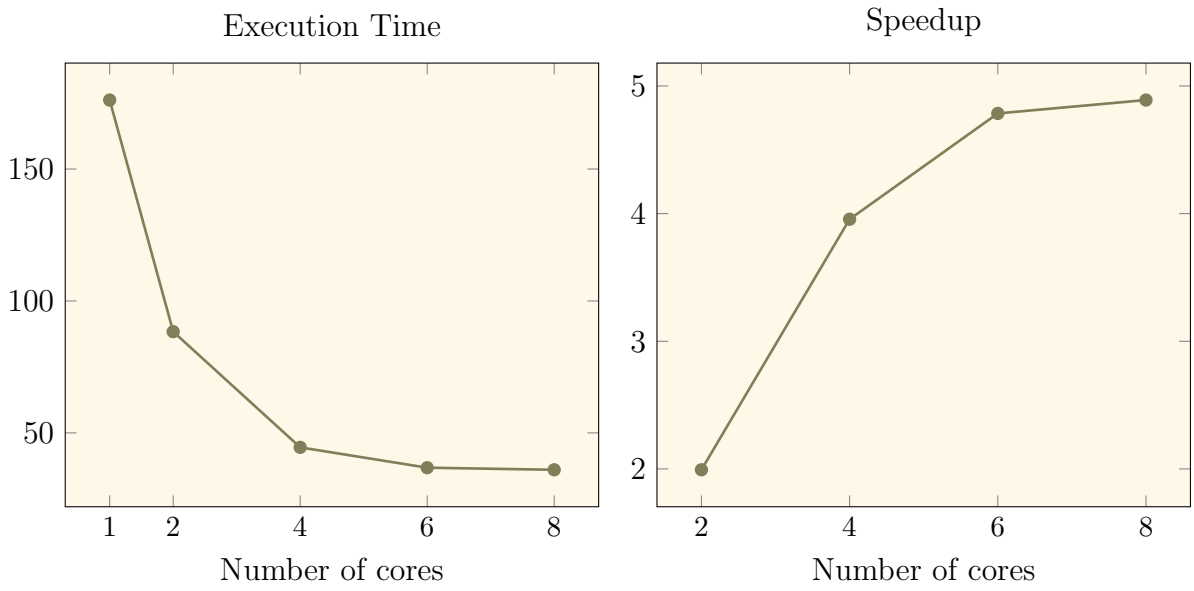
(a) Execution Time for N=1024 (1024×1024) (b) Speedup for N=1024 (1024×1024)

Εικόνα 2: Χρόνος εκτέλεσης και Speedup για $N = 1024$.

Σε αυτή την περίπτωση παρατηρούμε ότι έχουμε ιδανικό-γραμμικό SpeedUp ίσο με το πλήθος των cores και το executions time μειώνεται p φορές με χρήση p περισσότερων cores. Το μέγεθος του προβλήματος είναι τέτοιο ώστε να θεωρούνται αμελητέα τα overheads των νημάτων (και τα overheads πρόσβασης στη μνήμη) και τελικά να αξιοποιούμε πλήρως τον παραλληλισμό του προβλήματος. Τα περισσότερα νήματα που προστίθενται έχουν τον μέγιστο βαθμό χρησιμοποί-ησης επιτυγχάνοντας έτσι ιδανική συμπεριφορά (και επιτάχυνση) χωρίς να προστίθενται καθυ-στερήσεις λόγω overheads ή εξαρτήσεων.

Για $N = 4096$ (πλέγμα 4096×4096)

Τα διαγράμματα φαίνονται στην εικόνα 3 στην επόμενη σελίδα. Όπως και στην πρώτη περίπτωση με N=64, ο χρόνος εκτέλεσης μειώνεται με αύξηση του πλήθους των cores (και εδώ μάλιστα χωρίς κάποια εξαίρεση). Ωστόσο πάλι, ο ρυθμός βελτίωσης είναι φθίνων και έτσι η βελτίωση ναι μεν υπάρχει αλλά αυξανόμενων των cores γίνεται λιγότερο έντονη (και φυσικά όχι ανάλογη του πλήθους των cores). Σε αντίθεση με την πρώτη περίπτωση όπου αυτό αιτιολογήθηκε από τα overheads διαχείρισης των νημάτων (αυτά υπερίσχυαν λόγω του μικρού μεγέθους του προβλήματος), εδώ (που το πρόβλημα δεν έχει μικρό μέγεθος) δεν μπορούμε να πούμε ότι το overhead που επιφέρει μικρότερες βελτιώσεις είναι αυτό των νημά-των αλλά αυτό της πρόσβασης στην κοινή μνήμη. Συγκεκριμένα, λόγω του μεγέθους του προ-βλήματος, συχνά διαφορετικά νήματα θα πρέπει να προσπελάσουν είτε το ίδιο block μνήμης (π.χ. όταν οι πληροφορίες για την κατάσταση των γειτόνων βρίσκονται στο ίδιο block – false ή true sharing) είτε και διαφορετικά blocks περιοριζόμενα όμως από το bandwidth της μνήμης και δημιουργώντας συμφόρηση. Η συμφόρηση στη μνήμη είναι ένα overhead που για μεγάλο μέγεθος προβλήματος (άρα για συχνές προσπελάσεις μνήμης) και μεγάλο αριθμό ανταγωνιζόμενων (ως προς τη χρήση



(a) Execution Time for N=4096 (4096 × 4096)

(b) Speedup for N=4096 (4096 × 4096)

Εικόνα 3: Χρόνος εκτέλεσης και Speedup για $N = 4096$.

της μνήμης) threads θα επιφέρει καθυστερήσεις. Σημείωση : Η συμπεριφορά κατά την οποία παρατηρείται φθίνουσα βελτίωση με αύξηση του παραλληλισμού (cores) προβλέπεται και από τον νόμο του Amdahl. Συγκεκριμένα, το Game of Life ανήκει στην κατηγορία εφαρμογών Stencil οι οποίες κάνουν επαναλαμβανόμενα τον ίδιο υπολογισμό αφότου προσπελάσουν την μνήμη (επαναλαμβανόμενα : προσπέλαση μνήμης και υπολογισμός). Το Game of Life έχει μικρό operational intensity (περίπου 1/2) που πρακτικά σημαίνει ότι απαιτείται ανάγνωση σημαντικού κομματιού μνήμης για την εκτέλεση κάποιου operation. Έτσι, το πρόβλημα αυτό κατατάσσεται στα memory bound προβλήματα (ανάλογα και με τα χαρακτη-ριστικά του χρησιμοποιούμενου μηχανήματος, βλ. roofline plots). Ως εκ τούτου, η εφαρμογή αυτή έχει σημαντικό σειριακό κομμάτι εκτέλεσης, αυτό της προσπέλασης της μνήμης (το οποίο οφείλει να γίνει σειριακά και δεν παραλληλοποιείται). Αυτό, σύμφωνα και με τον νόμο του Amdahl οδηγεί σε μη ικανοποιητική κλιμακωσιμότητα (και άρα μη ικανοποιητική επίδοση με αύξηση του αριθμού των νημάτων).

ΜΕΡΟΣ Β

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

Παραλληλοποίηση των αλγορίθμων K-means και Floyd-Warshall

Εισαγωγή

Στην παρούσα εργαστηριακή άσκηση καλούμαστε να παραλληλοποιήσουμε και να βελτιστοποιήσουμε δύο προγράμματα με χρήση διάφορων τεχνικών και στη συνέχεια να εξάγουμε συμπεράσματα για την επίδραση αυτών των τεχνικών στην επίδοση των προγραμμάτων και για το βαθμό κατά τον οποίο επηρεάζουν την κλιμακωσιμότητά τους. Θεωρούμε αρχιτεκτονική κοινής μνήμης.

3 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Ο πρώτος αλγόριθμος τον οποίο επιχειρούμε να παραλληλοποιήσουμε είναι ο αλγόριθμος K-means. Ο αλγόριθμος αυτός προσπαθεί να ομαδοποιήσει N αντικείμενα σε K συστάδες (clusters) κοντινών μεταξύ τους στοιχείων. Συγκεκριμένα, μέχρι να επιτευχθεί κάποιο πλήθος επαναλήψεων του αλγορίθμου ή ο αλγόριθμος να συγκλίνει[2], για κάθε αντικείμενο υπολογίζεται η απόστασή του (ευκλείδια απόσταση) από το κέντρο όλων των συστάδων ώστε να βρεθεί η κοντινότερη, το αντικείμενο αυτό προστίθεται εν συνεχεία στην συστάδα αυτή (με κατάλληλη ενημέρωση των αντίστοιχων δομών του προγράμματος) και τέλος με βάση την ομαδοποίηση που προκύπτει επανυπολογίζονται τα κέντρα των συστάδων. Θα χρησιμοποιηθούν δύο μέθοδοι παραλληλοποίησης:

1. Η μέθοδος των Shared Clusters, όπου οι κρίσιμες δομές του προγράμματος θα αντιμετωπιστούν ως διαμοιραζόμενες με συνέπεια την ανάγκη για συγχρονισμό.

2. Η μέθοδος των Copied Clusters and Reduce, όπου θα δημιουργηθούν αντίγραφα των κρίσιμων δομών του προγράμματος για κάθε νήμα και στο τέλος τα επιμέρους αποτελέσματα θα πρέπει να συναθροιστούν (reduction).

Ο παραλληλισμός σε αυτό το πρόγραμμα, έγκειται στην διαδικασία που περιεγράφηκε παραπάνω. Διαφορετικά νήματα αναλαμβάνουν τους απαραίτητους υπολογισμούς για διαφορετικές ομάδες από αντικείμενα, για τα οποία δηλαδή θα υπολογίσουν την κοντινότερη συστάδα, θα ενημερώσουν τις κατάλληλες δομές και θα επανυπολογίσουν τα κέντρα.

3.1 Shared Clusters

Παρακάτω απαντώνται τα επιμέρους ερωτήματα για αυτή τη μέθοδο παραλληλισμού.

3.1.1 1ο Ζητούμενο : Παραλληλοποίηση και σχολιασμός

Αρχικά, παρατίθεται ο κώδικας της άσκησης με τις απαραίτητες προσθήκες για την παραλληλοποίηση (συγκεκριμένα, το μέρος του κώδικα στο οποίο εφαρμόζεται παραλληλισμός).

```
1     do {
2     // before each loop, set cluster data to 0
3     for (i=0; i<numClusters; i++) {
4     for (j=0; j<numCoords; j++)
```

[2]δηλαδή να μην αλλάζει πολύ (αυτό καθορίζεται από την μεταβλητή delta του προγράμματος) ο τρόπος κατανομής των αντικειμένων στις συστάδες

```
5     newClusters[i*numCoords + j] = 0.0;
6     newClusterSize[i] = 0;
7     }
8
9     delta = 0.0;
10
11    /*
12    * TODO: Detect parallelizable region and use appropriate OpenMP
          pragmas
13    */
14    #pragma omp parallel for private(i,j,index) default(shared) reduction
          (+:delta)
15    for (i=0; i<numObjs; i++) {
16    // find the array index of nearest cluster center
17    index = find_nearest_cluster(numClusters, numCoords, &objects[i*
          numCoords], clusters);
18
19    // if membership changes, increase delta by 1
20    if (membership[i] != index)
21    delta += 1.0;
22
23    // assign the membership to object i
24    membership[i] = index;
25
26    // update new cluster centers : sum of objects located within
27    /*
28    * TODO: protect update on shared "newClusterSize" array
29    */
30    #pragma omp atomic update
31    newClusterSize[index]++;
32    for (j=0; j<numCoords; j++)
33    /*
34    * TODO: protect update on shared "newClusters" array
35    */
36    #pragma omp atomic update
37    newClusters[index*numCoords + j] += objects[i*numCoords + j];
38    }
39
40    // average the sum and replace old cluster centers with newClusters
41    for (i=0; i<numClusters; i++) {
42    if (newClusterSize[i] > 0) {
43    for (j=0; j<numCoords; j++) {
44    clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
          newClusterSize[i];
45    }
46    }
47    }
48
49    // Get fraction of objects whose membership changed during this loop.
          This is used as a convergence criterion.
50    delta /= numObjs;
51
52    loop++;
53    printf("\r\tcompleted loop %d", loop);
54    fflush(stdout);
55    } while (delta > threshold && loop < loop_threshold);
```

View the .cpp file here *(Dropbox link)*

Οι προσθήκες που απαιτούνται ώστε ο κώδικας να παραλληλοποιηθεί είναι οι εξής :

1. **Γραμμή 14:** Με την εντολή αυτή παραλληλοποιούμε το βασικό for loop του κώδικά μας που υλοποιεί τον πυρήνα του αλγορίθμου. Συγκεκριμένα, δημιουργούνται με αυτή την εντολή πολλαπλά νήματα τα οποία αναλαμβάνουν να εκτελέσουν διαφορετικά iterations. Σημαντικός εδώ είναι ο χαρακτηρισμός των δεδομένων (data access patterns). Συγκεκριμένα, θα πρέπει για τις μεταβλητές i, j και index να δημιουργηθούν τοπικά αντίγραφα για κάθε διαφορετικό νήμα, καθώς καθένα από αυτά εκτελεί διαφορετική επανάληψη του βρόχου (και δεν θα πρέπει αυτές να περιπλέκονται για διαφορετικά νήματα) και υπολογίζει διαφορετικό index για τα αντικείμενα που έχει αναλάβει. Κατά τα άλλα, by default, οι υπόλοιπες δομές θα είναι διαμοιραζόμενες (shared), ενώ ειδική μέριμνα θα πρέπει να υπάρξει για την μεταβλητή delta, που καθορίζει την σύγκλιση του αλγορίθμου. Συγκεκριμένα, αυτή θα πρέπει να είναι τοπική για κάθε νήμα και στο τέλος οι τοπικές τιμές όλων των νημάτων να αθροιστούν, κάτι που παραπέμπει σε reduction με operation την πρόσθεση (+).[3]

2. **Γραμμές 30 και 36:** Αφού ορίσαμε την παράλληλη περιοχή με την προηγούμενη εντολή, τώρα καλούμαστε για τις κοινές διαμοιραζόμενες δομές που διαβάζουν και τροποποιούν όλα τα νήματα, να επιλύσουμε τα race conditions υλοποιώντας συγχρονισμό. Αυτές οι δομές είναι οι newClusterSize και newClusters. Για αυτές τις δύο δομές, θα πρέπει να εξασφαλιστεί η ατομική προσπέλασή τους (τροποποίησή τους) με ορισμό κατάλληλης κρίσιμης περιοχής. Ως εκ τούτου, η εντολή που προστέθηκε στις δύο αυτές γραμμές ορίζει ότι η ενημέρωση των δύο δομών θα πραγματοποιηθεί από atomic instruction, το οποίο γίνεται ατομικά από το hardware.

Σημειώνουμε ότι οι γραμμές 41 έως 55 θα εκτελεστούν ατομικά από το initial thread (χωρίς να χρειάζεται να οριστεί ρητά αυτό με εντολή, καθώς αυτό το τμήμα είναι έτσι κι αλλιώς εκτός της παράλληλης περιοχής που ορίσαμε και θα γίνει ατομικά). Στην επόμενη σελίδα, στην εικόνα 4,παραθέτουμε τα διαγράμματα χρόνου και speedup.

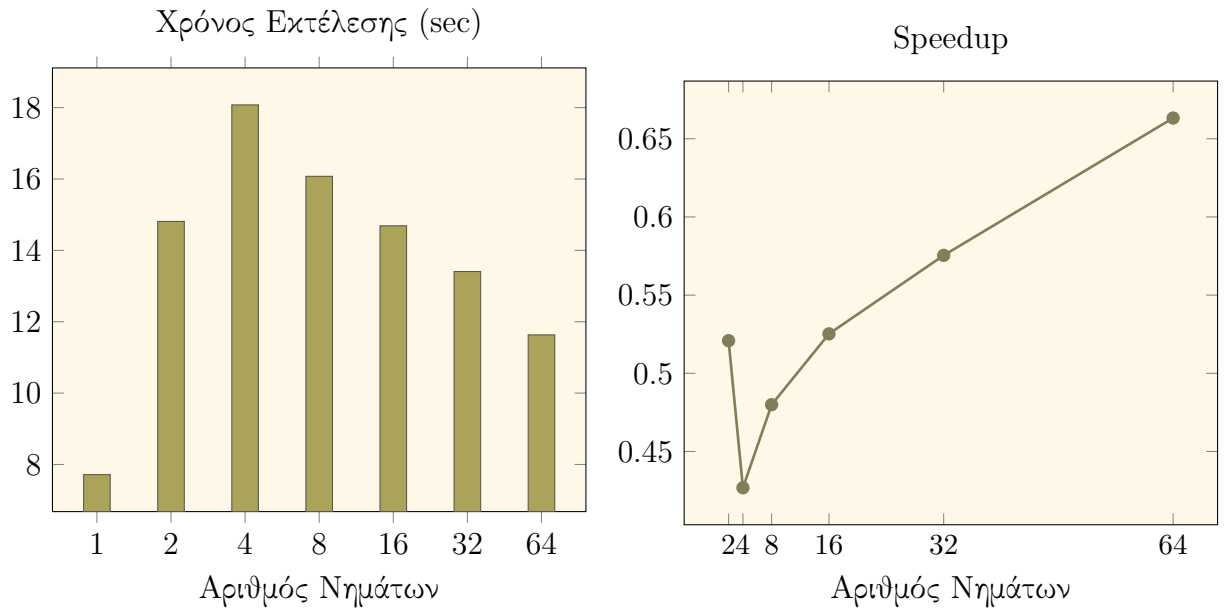
Παρατηρήσεις

Από τα διαγράμματα της εικόνας 4, παρατηρούμε καταρχάς ότι σε κάθε περίπτωση η χρήση πολυνηματισμού επιφέρει χειρότερες επιδόσεις συγκριτικά με τον σειριακό αλγόριθμο. Γενικά, δύο είναι οι τρόποι με τους οποίους ο συγκεκριμένος πολυνηματισμός (με διαμοιραζόμενες δομές) επηρεάζει την επίδοση αυτού του αλγορίθμου:

- η χρήση πολυνηματισμού κατανέμει τον υπολογιστικό φόρτο σε πολλούς εργάτες κάτι που τείνει να βελτιώσει την επίδοση

- η ανάγκη για συγχρονισμό ακριβώς λόγω των διαμοιραζόμενων δομών οδηγεί σε δημιουργία ανταγωνισμού μεταξύ των νημάτων για την δέσμευση του κρίσιμου τμήματος, με τον ανταγωνισμό να γίνεται εντονότερος με την αύξηση των νημάτων επιδεινώνοντας την επίδοση

[3]Συγκεκριμένα, ο αλγόριθμος συγκλίνει όταν το delta είναι μικρότερο από κάποιο threshhold, οπότε κάθε νήμα αυξάνει την τιμή του delta και έτσι απομακρύνει τον αλγόριθμο από την σύγκλιση όταν κάποιο αντικείμενο αλλάζει συστάδα. Γίνεται λοιπόν σαφές, ότι τελικά οι τοπικές τιμές των delta πρέπει να προστεθούν.



(a) Χρόνος Εκτέλεσης - Πλήθος Νημάτων (b) Speedup - Πλήθος Νημάτων

Εικόνα 4: Χρόνος εκτέλεσης και Speedup για διαφορετικό πλήθος νημάτων

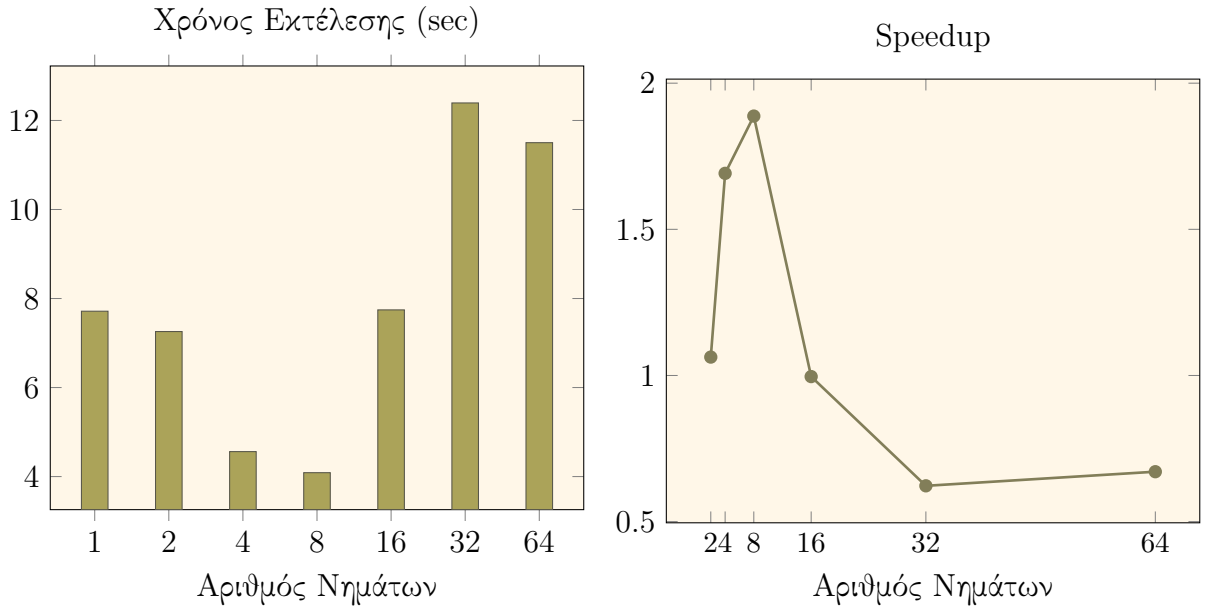
Όπως φαίνεται όμως, στην δική μας περίπτωση η αρνητική επίδραση του ανταγωνισμού αυτού υπερκαλύπτει την θετική συνεισφορά του πολυνηματισμού στην κατανομή των υπολογισμών και έτσι βλέπουμε τα παραπάνω αποτελέσματα. Το Speedup παραμένει σταθερά μικρότερο του 1. Οι προσβάσεις στην κρίσιμη περιοχή δημιουργούν ένα σημαντικό σειριακό μέρος στον αλγόριθμο που τελικά είναι καθοριστικό για την κακή κλιμακωσιμότητα (βλ. Amdhal). Ταυτόχρονα, το γεγονός ότι όλα τα νήματα καταλήγουν να γράφουν (modify) στις κοινές αυτές δομές, αφενός δημιουργεί πολλή κίνηση στο δίαυλο για invalidations του αντίστοιχου coherence protocol και αφετέρου γίνονται invalid ενδεχομένως χρήσιμα δεδομένα στις caches των νημάτων. Τέλος ένα ακόμη φαινόμενο το οποίο επηρεάζει την επίδοση είναι το γεγονός ότι ένα νήμα μπορεί κατά τη διάρκεια της ζωής του να εκτελεστεί κατά περιόδους από διαφορετικά cores του συστήματος (όταν π.χ. ένα από αυτα αποφασίζει να κάνει context switch και να ασχοληθεί με κάποιο άλλο νήμα) με συνέπεια, κάθε φορά που αυτό συμβαίνει, το νέο αυτό core να μην έχει ήδη αποθηκευμένη την κατάσταση του εν λόγω νήματος και να καταναλώσει χρόνο στο να διαβάσει πληροφορίες από την αρχή (κάτι που δε θα συνέβαινε αν το νήμα αυτό επέστρεφε στο αρχικό core στο οποίο εκτελούνταν). Αυτό μας οδηγεί στο επόμενο ζητούμενο παρακάτω.

3.1.2 2ο Ζητούμενο : Thread Binding

Ακριβώς αυτό το ζήτημα που αναφέρθηκε τελευταίο παραπάνω, επιλύεται με χρήση της συγκεκριμένης μεταβλητής περιβάλλοντος, η οποία επιβάλλει σε ένα νήμα να ανατίθεται σε μια συγκεκριμένη υπολογιστική μονάδα σε όλη τη διάρκεια της ζωής του και ως εκ τούτου να μην δημιουργούνται περιττές καθυστερήσεις όταν ένα νήμα ανατίθεται σε ένα νέο core το οποίο θα πρέπει να ενημερωθεί για την κατάσταση αυτού του νήματος (π.χ. δεν έχει ήδη στην κρυφή του μνήμη τιμές χρήσιμων μεταβλητών που χρησιμοποιούσε αυτό το νήμα). Συγκεκριμένα, στο αρχείο run_on_queue.sh προσθέτουμε τη γραμμή κώδικα :

```
1     export GOMP_CPU_AFFINITY="0 - (N-1)"
```

όπου N είναι ο αριθμός των νημάτων. Με αυτό τον τρόπο, κυκλικά ανατίθεται στον πυρήνα 0 το νήμα 0, στον πυρήνα 1 το νήμα 1 κ.ο.κ. Τα διαγράμματα χρόνου και speedup φαίνονται παρακάτω :



(a) Χρόνος Εκτέλεσης - Πλήθος Νημάτων

(b) Speedup - Πλήθος Νημάτων

Εικόνα 5: Χρόνος εκτέλεσης και Speedup για διαφορετικό πλήθος νημάτων με χρήση thread binding

Παρατηρήσεις

Από τα παραπάνω διαγράμματα, παρατηρούμε βασικά ότι οι επιδόσεις έχουν βελτιωθεί συνολικά και μάλιστα σε βαθμό που σε κάποιες περιπτώσεις έχουμε speedup μεγαλύτερο της μονάδας (οπότε το παράλληλο πρόγραμμα είναι καλύτερο από το σειριακό - αν και το speedup σε όλες τις περιπτώσεις είναι πολύ χειρότερο από το γραμμικό). Όμως, με αύξηση του πλήθους των νημάτων από ένα όριο και πάνω, η επίδοση επιδεινώνεται ενώ σε κάθε περίπτωση γενικά η επίδοση είναι μη ικανοποιητική και η κλιμακωσιμότητα σχεδόν ανύπαρκτη. Αυτό μας οδηγεί στην αναζήτηση καλύτερου τρόπου παραλληλοποίησης.

3.2 Copied Clusters and Reduce

Παρακάτω παρατίθενται οι απαντήσεις μας στα ζητούμενα ερωτήματα.

3.2.1 1ο Ζητούμενο : Παραλληλοποίηση κώδικα και σχολιασμός

Γενικά, θα μπορούσαμε να πούμε ότι υπάρχουν δύο τρόποι προσέγγισης αυτής της υλοποίησης:

1. Κάθε νήμα, όταν μπαίνει στην παράλληλη περιοχή, δημιουργεί για τον εαυτό του μια τοπική ιδιωτική δομή (για κάθε μια από τις 2 διαμοιραζόμενες δομές που μας απασχολούν), ενημερώνει αυτήν με τους υπολογισμούς του (χωρίς να χρειάζεται να οριστεί ως private

καθώς κάθε νήμα δημιουργεί την δική του) και όταν ολοκληρώσει την εργασία του, ατομικά (με ορισμό κρίσιμού τμήματος - oops!) αναλαμβάνει να ενημερώσει την αντίστοιχη global δομή προσθέτοντάς της τον δικό του υπολογισμό που έκανε στην ιδιωτική του δομή. Αυτό βέβαια συνοδεύεται με όλα τα άσχημα της προηγούμενης υλοποίησης λόγω της ανάγκης για συγχρονισμό (έστω και μικρότερης κλίμακας μόνο μετά το τέλος των ατομικών υπολογισμών και όχι συνέχεια όπως στην προηγούμενη υλοποίηση).

2. Το αρχικό νήμα (initial thread) δημιουργεί για κάθε ένα από τα νήματα μια private δομή (για κάθε μια από τις 2 δομές που μας απασχολούν) δίνοντάς της επιπρόσθετα - ως πρώτο όρισμα - το νήμα στο οποίο ανήκει. Στη συνέχεια, κάθε νήμα τροποποιεί μόνο το αντίγραφο που του ανήκει (το οποίο έχει δηλαδή ως πρώτο όρισμα το δικό του thread_id). Στο τέλος, αφότου κάθε νήμα έχει κάνει υπολογισμούς στις δικές του δομές, το initial thread αναλαμβάνει να ενημερώσει την global δομή προσθέτοντας[4] τους επιμέρους υπολογισμούς από κάθε νήμα. Αυτή η υλοποίηση δεν απαιτεί συγχρονισμό (η ενημέρωση της global δομής γίνεται μόνο από το αρχικό νήμα) και το μόνο overhead είναι η δημιουργία των τοπικών δομών και η τελική συνάθροισή τους (τα οποία γίνονται από το αρχικό νήμα και δεν παραλληλοποιούνται).

Παρακάτω, υλοποιούμε στον κώδικα την δεύτερη προσέγγιση που έχει απαλλαγεί από την ανάγκη συγχρονισμού. Σημειώνεται ότι διατηρούμε σε αυτήν την υλοποίηση τον μηχανισμό του thread binding ώστε να εκμεταλλευτούμε τα οφέλη του.

```
1     do {
2     // before each loop, set cluster data to 0
3     for (i=0; i<numClusters; i++) {
4     for (j=0; j<numCoords; j++)
5     newClusters[i*numCoords + j] = 0.0;
6     newClusterSize[i] = 0;
7     }
8     delta = 0.0;
9     /*
10    * TODO: Initiliaze local cluster data to zero (separate for each
          thread)
11    */
12    int n;
13    for(n=0 ; n<nthreads ; n++){
14    for (i=0; i<numClusters; i++) {
15    for (j=0; j<numCoords; j++)
16    local_newClusters[n][i*numCoords + j] = 0.0;
17    local_newClusterSize[n][i] = 0;
18    }
19    }
20    #pragma omp parallel for private(i, j, index) default(shared)
          reduction(+:delta)
21    for (i=0; i<numObjs; i++)
22    {
23    // find the array index of nearest cluster center
24    index = find_nearest_cluster(numClusters, numCoords, &objects[i*
          numCoords], clusters);
```

[4] οι πράξεις που κάνει κάθε νήμα στις ιδιωτικές του δομές είναι προσθετικές (είτε προσθέτει 1 στη δομή μεγέθους της συστάδας, είτε προσθέτει τις συντεταγμένες του στις συντεταγμένες του κέντρου της συστάδας ώστε μετά να υπολογιστεί ο μέσος όρος), οπότε είναι λογικό ότι το reduction πρεπει να είναι προσθετικό

```
25
26      // if membership changes, increase delta by 1
27      if (membership[i] != index)
28      delta += 1.0;
29
30      // assign the membership to object i
31      membership[i] = index;
32
33      // update new cluster centers : sum of all objects located within (
           average will be performed later)
34      /*
35      * TODO: Collect cluster data in local arrays (local to each thread)
36      *       Replace global arrays with local per-thread
37      */
38      local_newClusterSize[omp_get_thread_num()][index]++;
39      for (j=0; j<numCoords; j++)
40      local_newClusters[omp_get_thread_num()][index*numCoords + j] +=
           objects[i*numCoords + j];
41      }
42      /*
43      * TODO: Reduction of cluster data from local arrays to shared.
44      *       This operation will be performed by one thread
45      */
46      int t, index, j;
47      for(t=0 ; t<nthreads ; t++){
48      for(index=0 ; index<numClusters ; index++){
49      for(j=0 ; j<numCoords ; j++){
50      newClusters[index*numCoords + j]+=local_newClusters[t][index*numCoords
            + j];
51      }
52      newClusterSize[index]+=local_newClusterSize[t][index];
53      }
54      }
55      // average the sum and replace old cluster centers with newClusters
56      for (i=0; i<numClusters; i++) {
57      if (newClusterSize[i] > 0) {
58      for (j=0; j<numCoords; j++) {
59      clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
           newClusterSize[i];
60      }
61      }
62      }
63      // Get fraction of objects whose membership changed during this loop.
           This is used as a convergence criterion.
64      delta /= numObjs;
65      loop++;
66      printf("\r\tcompleted loop %d", loop);
67      fflush(stdout);
68      } while (delta > threshold && loop < loop_threshold);
```

View the .cpp file here *(Dropbox link)*

Οι προσθήκες που απαιτούνται ώστε ο κώδικας να παραλληλοποιηθεί είναι οι εξής :

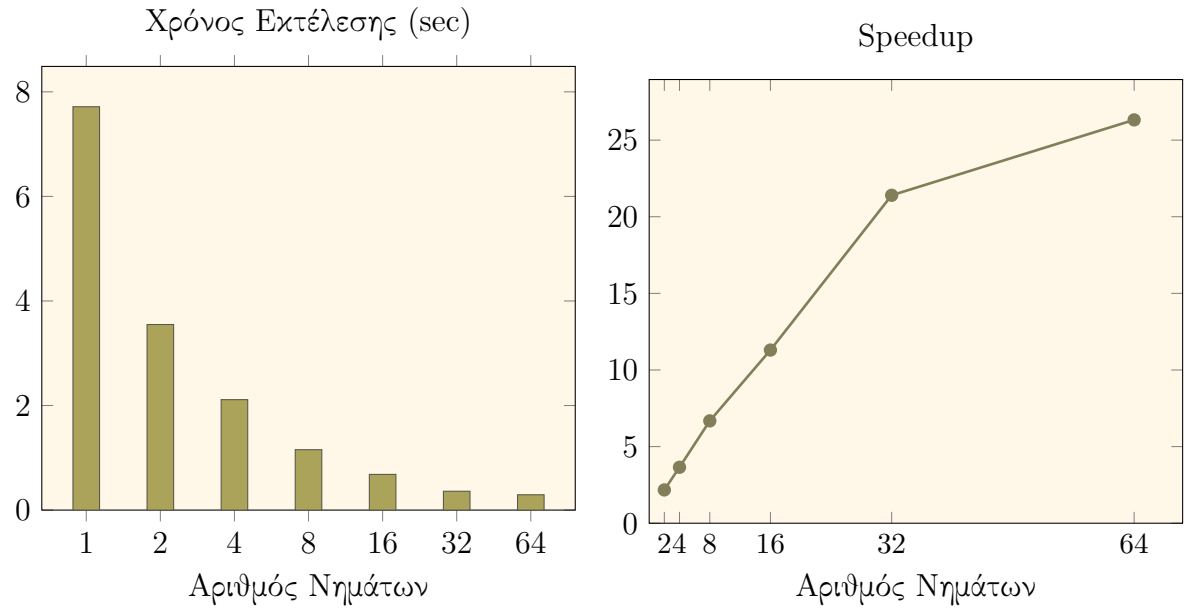
1. **Γραμμές 14-21:** Η δημιουργία από το αρχικό νήμα όλων των ιδιωτικών δομών που θα χρησιμοποιήσουν τα νήματα.

2. **Γραμμή 23:** Ο ορισμός του παράλληλου τμήματος όπως ακριβώς και στην προηγούμενη υλοποίηση.

3. **Γραμμές 41-43:** Το κάθε νήμα τροποποιεί μόνο τις δομές που του ανήκουν (προσθήκη "local" και χρήση της omp_get_thread_num()).

4. **Γραμμές 51-59:** Το αρχικό νήμα ενημερώνει την global δομή προσθέτοντας τα επιμέρους αποτελέσματα των τοπικών δομών (συγκεκριμένα για κάθε νήμα και για κάθε cluster προστίθενται τα τοπικά αποτελέσματα για κάθε coordinate).

Τα διαγράμματα χρόνου και speedup φαίνονται παρακάτω :



(a) Χρόνος Εκτέλεσης - Πλήθος Νημάτων

(b) Speedup - Πλήθος Νημάτων

Εικόνα 6: Χρόνος εκτέλεσης και Speedup για διαφορετικό πλήθος νημάτων με χρήση copied clusters and reduction

Παρατηρήσεις

Από τα παραπάνω διαγράμματα, παρατηρούμε ότι η επίδοση είναι εμφανώς καλύτερη με αυτή την υλοποίηση. Συγκεκριμένα, το speedup είναι σχεδόν γραμμικό μέχρι τα 8 νήματα και από εκεί και έπειτα παραμένει ικανοποιητικό (αν και μικρότερο του γραμμικού). Η επίδοση συνεχώς βελτιώνεται με αύξηση του πλήθους των νημάτων. Το αποτέλεσμα είναι αναμενόμενο καθώς αποφύγαμε το overhead του συγχρονισμού. Ωστόσο, να σημειωθεί ότι αυτή η υλοποίηση επιβαρύνει περισσότερο τη μνήμη, καθώς θα πρέπει να δεσμευθεί χώρος για κάθε ένα από τα τοπικά αντίγραφα των δομών.

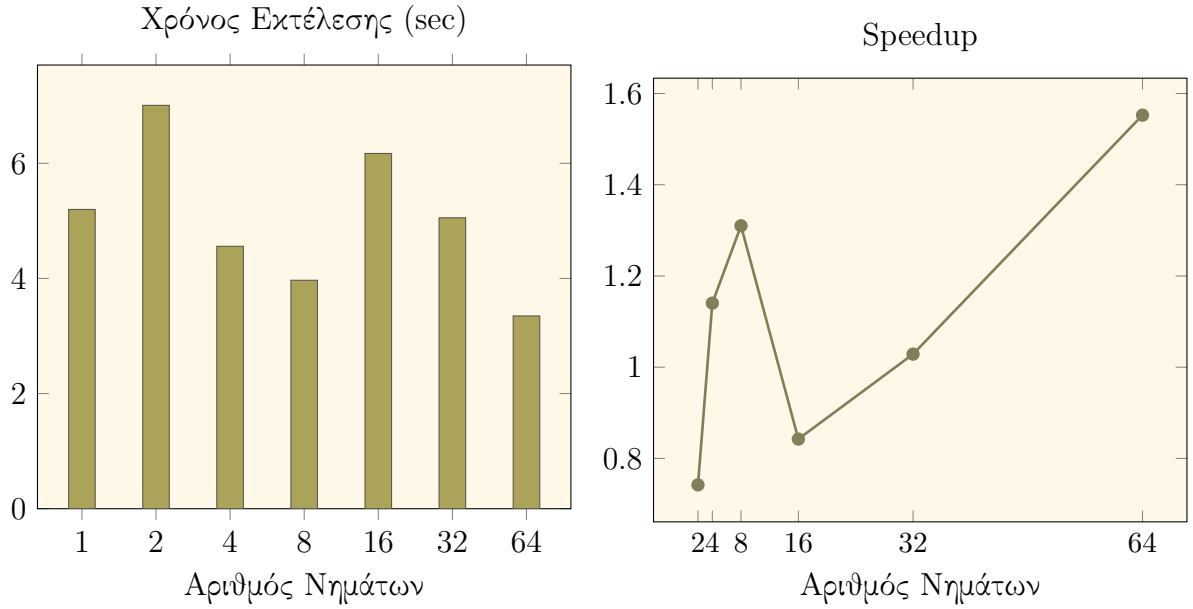
Operational Intensity

Στο πρόγραμμα αυτό, ακολουθείται η λογική ότι πρέπει να διαβαστεί ένας μεγάλος όγκος πληροφοριών για την εκτέλεση λίγων πράξεων. Συγκεκριμένα, διαβάζεται ολόκληρος ο πίνακας

objects (που περιέχει τις 16 συντεταγμένες των > 2 εκ. αντικειμένων) και ο πίνακας newClusters ώστε να διαβαστούν οι συντεταγμένες των κέντρων όλων των συστάδων και στη συνέχεια γίνονται κάποιες απλές πράξεις υπολογισμού αποστάσεων και σύγκριση αυτών. Αυτό ανάγει το πρόβλημα σε memory bound, γεγονός που σημαίνει ότι οι δυνατότητες της μνήμης περιορίζουν την κλιμακωσιμότητα. Σε αυτό εν μέρει θα μπορούσε να οφείλεται το γεγονός ότι το σχεδόν γραμμικό speedup που παρατηρούμε για μικρό αριθμό νημάτων, στη συνέχεια γίνεται εμφανώς sublinear.

3.2.2 2ο Ζητούμενο : Χρήση διαφορετικού configuration - first touch policy - false sharing

Αρχικά, χρησιμοποιώντας το configuration που υποδεικνύεται, ξανατρέχουμε το πρόγραμμα (χρησιμοποιώντας την προηγούμενη υλοποίηση παραλληλισμού - copied clusters and reduction) και λαμβάνουμε τα παρακάτω αποτελέσματα[5].



(a) Χρόνος Εκτέλεσης - Πλήθος Νημάτων

(b) Speedup - Πλήθος Νημάτων

Εικόνα 7: Χρόνος εκτέλεσης και Speedup με Config=$\{256, 1, 4, 10\}$

Παρατηρούμε ότι οι επιδόσεις είναι πολύ χειρότερες συγκριτικά με το προηγούμενο configuration και παρακάτω θα εξηγήσουμε ότι αυτό οφείλεται στα χαρακτηριστικά της NUMA αρχιτεκτονικής του πολυεπεξεργαστικού συστήματος που συναντάται στο Linux. Συγκεκριμένα, βασική πολιτική που χρησιμοποιείται σε τέτοια συστήματα είναι η first touch policy. Σύμφωνα με αυτή, το memory allocation για ένα νήμα γίνεται με τέτοιο τρόπο ώστε η μνήμη που δεσμεύεται για το νήμα προκειμένου να αποθηκεύσει ένα αντικείμενο που δημιούργησε, να βρίσκεται κατά το δυνατόν εγγύτερα σε αυτό. Στο δικό μας πρόγραμμα, υπενθυμίζουμε ότι οι δημιουργία

[5]Χρησιμοποιήθηκε το configuration που υποδείχθηκε εκ των υστέρων ως διόρθωση όπου τα clusters είναι 4 και όχι 8. Για παράδειγμα, με 8 clusters το μέγεθος του πίνακα newClusters είναι $8clusters \times 8bytes(double) = 64bytes$, ενώ με 4 clusters το μέγεθος είναι 32 bytes. Υποθέτουμε ότι το μέγεθος των memory blocks είναι τέτοιο ώστε να δημιουργεί false sharing μεταξύ πινάκων των 32 και όχι των 64 bytes.

(δηλαδή η δέσμευση μνήμης) και η αρχικοποίηση των τοπικών δομών των νημάτων πραγματοποιείται ατομικά από το αρχικό νήμα. Όμως αυτό, με βάση όσα αναφέρθηκαν για την first touch policy, έχει ως συνέπεια οι τοπικές δομές όλων των νηματων να αποθηκεύονται στην μνήμη που αντιστοιχεί στο αρχικό νήμα. Ως εκ τούτου, η προσπέλαση αυτών των τοπικών δομών από τα νήματα στα οποία ανήκουν αφενός απαιτεί περισσότερο χρόνο (καθώς τα νήματα θα πρέπει να αναζητήσουν τα δεδομένα τους σε μια απομακρυσμένη από αυτά μνήμη - αυξημένο memory latency) και αφετέρου δημιουργεί συμφόρηση στον memory controller ο οποίος θα πρέπει να εξυπηρετεί αιτήματα που προορίζονται για την ίδια πάντα μνήμη (αυτή του αρχικού νήματος).

Ακριβώς τα παραπάνω έχουν επίσης ως συνέπεια την επιδείνωση του φαινομένου false sharing. Συγκεκριμένα, το γεγονός ότι τα πάντα (όλες οι τοπικές δομές) βρίσκονται στο κομμάτι μνήμης του αρχικού νήματος, έχει ως συνέπεια στοιχεία διαφορετικών τοπικών δομών να ανήκουν στο ίδιο block αυτής της μοναδικής μνήμης. Μάλιστα, το παραπάνω επιδεινώνεται ακόμη περισσότερο από το παρόν configuration, εξαιτίας του οποίου οι τοπικές δομές είναι μικρότερου μεγέθους (ενώ ταυτόχρονα το μέγεθος των blocks παραμένει σταθερό). Ως εκ τούτου, μεγαλύτερο ποσοστό των στοιχείων δύο γειτονικών τοπικών δομών (δηλαδή δύο διαδοχικών νημάτων) θα έχει τοποθετηθεί στο ίδιο memory block οδηγώντας σε false sharing. Αυτό θα οδηγήσει σε ανεπιθύμητα invalidations του coherence protocol τα οποία θα διαγράφουν χρήσιμα αντίγραφα δεδομένων στα νήματα, δημιουργώντας ταυτόχρονα μη χρήσιμη κίνηση στον δίαυλο δεδομένων.

Αυτό επιλύεται με παραλληλοποίηση τόσο της δέσμευσης μνήμης για τις τοπικές δομές όσο και της αρχικοποίησής τους, η οποία υλοποιείται στις γραμμές κώδικα 7 και 29 όπως φαίνεται παρακάτω. Με αυτό τον τρόπο, το κάθε νήμα τοποθετεί τις χρήσιμες για αυτό τοπικές δομές στην κοντινή του μνήμη (και η προσπέλαση γίνεται γρηγορότερα), ενώ ταυτόχρονα ο διαχωρισμός των τοπικών δομών σε πολλαπλά διαφορετικά κομμάτια μνήμης (άρα σε διαφορετικά blocks τα οποία ανήκουν σε διαφορετικές μνήμες) επιλύει το φαινόμενο του false sharing.

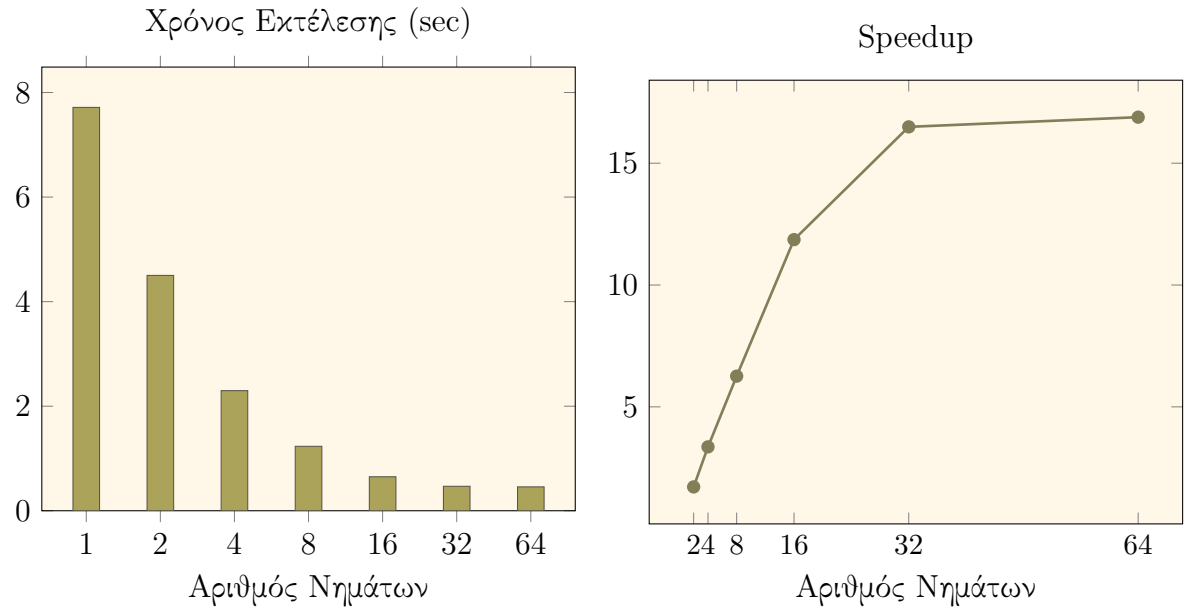
```
1    /*
2    * Hint for false-sharing
3    * This is noticed when numCoords is low (and neighboring
        local_newClusters exist close to each other).
4    * Allocate local cluster data with a "first-touch" policy.
5    */
6    // Initialize local (per-thread) arrays (and later collect result on
        global arrays)
7    #pragma omp parallel for private(k)
8    for (k=0; k<nthreads; k++)
9    {
10   local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(
        numClusters, sizeof(**local_newClusterSize));
11   local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters
        * numCoords, sizeof(**local_newClusters));
12   }
13   timing = wtime();
14   do {
15   // before each loop, set cluster data to 0
16   for (i=0; i<numClusters; i++) {
17   for (j=0; j<numCoords; j++)
18   newClusters[i*numCoords + j] = 0.0;
19   newClusterSize[i] = 0;
20   }
21   delta = 0.0;
```

```
22      /*
23      * TODO: Initiliaze local cluster data to zero (separate for each
            thread)
24      */
25      int n;
26      #pragma omp parallel for private(n, i, j) default(shared)
27      for(n=0 ; n<nthreads ; n++){
28      for (i=0; i<numClusters; i++) {
29      for (j=0; j<numCoords; j++)
30      local_newClusters[n][i*numCoords + j] = 0.0;
31      local_newClusterSize[n][i] = 0;
32      }
33      }
34      #pragma omp parallel for private(i, j, index) default(shared)
            reduction(+:delta)
35      for (i=0; i<numObjs; i++)
36      {
37      // find the array index of nearest cluster center
38      index = find_nearest_cluster(numClusters, numCoords, &objects[i*
            numCoords], clusters);
39      // if membership changes, increase delta by 1
40      if (membership[i] != index)
41      delta += 1.0;
42      // assign the membership to object i
43      membership[i] = index;
44      // update new cluster centers : sum of all objects located within (
            average will be performed later)
45      /*
46      * TODO: Collect cluster data in local arrays (local to each thread)
47      *       Replace global arrays with local per-thread
48      */
49      local_newClusterSize[omp_get_thread_num()][index]++;
50      for (j=0; j<numCoords; j++)
51      local_newClusters[omp_get_thread_num()][index*numCoords + j] +=
            objects[i*numCoords + j];
52      }
53      /*
54      * TODO: Reduction of cluster data from local arrays to shared.
55      *       This operation will be performed by one thread
56      */
57      int t, index, j;
58      for(t=0 ; t<nthreads ; t++){
59      for(index=0 ; index<numClusters ; index++){
60      for(j=0 ; j<numCoords ; j++){
61      newClusters[index*numCoords + j]+=local_newClusters[t][index*numCoords
            + j];
62      }
63      newClusterSize[index]+=local_newClusterSize[t][index];
64      }
65      }
66      // average the sum and replace old cluster centers with newClusters
67      for (i=0; i<numClusters; i++) {
68      if (newClusterSize[i] > 0) {
69      for (j=0; j<numCoords; j++) {
70      clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
            newClusterSize[i];
```

```
71      }
72      }
73      }
74      // Get fraction of objects whose membership changed during this loop.
            This is used as a convergence criterion.
75      delta /= numObjs;
76      loop++;
77      printf("\r\tcompleted loop %d", loop);
78      fflush(stdout);
79      } while (delta > threshold && loop < loop_threshold);
```

View the .cpp file here *(Dropbox link)*

Παρακάτω παρατίθεται τα διαγράμματα χρόνου και speedup για αυτή την υλοποίηση.



(a) Χρόνος Εκτέλεσης - Πλήθος Νημάτων

(b) Speedup - Πλήθος Νημάτων

Εικόνα 8: Χρόνος εκτέλεσης και Speedup με Config=$\{256, 1, 8, 10\}$ και επίλυση της αρνητικής επίδρασης των first touch policy και false sharing.

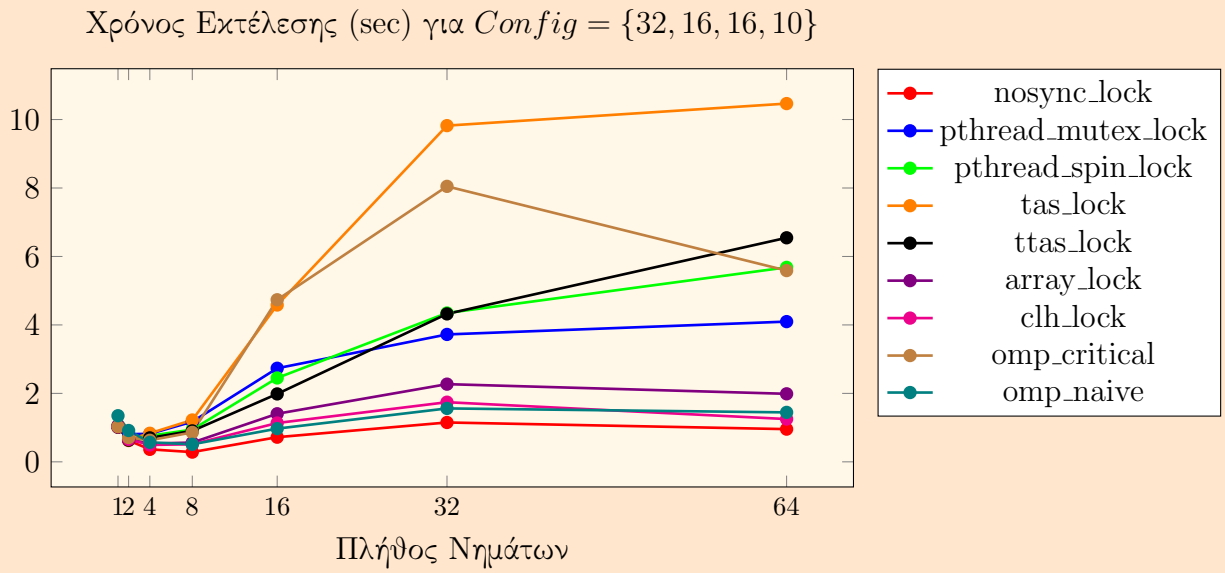
Παρατηρούμε ότι η επίδοση είναι εμφανώς καλύτερη. Το speedup είναι κοντά στο γραμμικό για μικρό αριθμό νημάτων ενώ για μεγαλύτερο αριθμό νημάτων γίνεται χειρότερο (κάτι που είναι βέβαια αναμενόμενο σε τυπικά προγράμματα). Η κλιμακωσιμότητα είναι αρκετά καλή και πλέον ικανοποιητική (συγκριτικά με τις προηγούμενες υλοποιήσεις).

3.2.3 3ο Ζητούμενο (bonus)

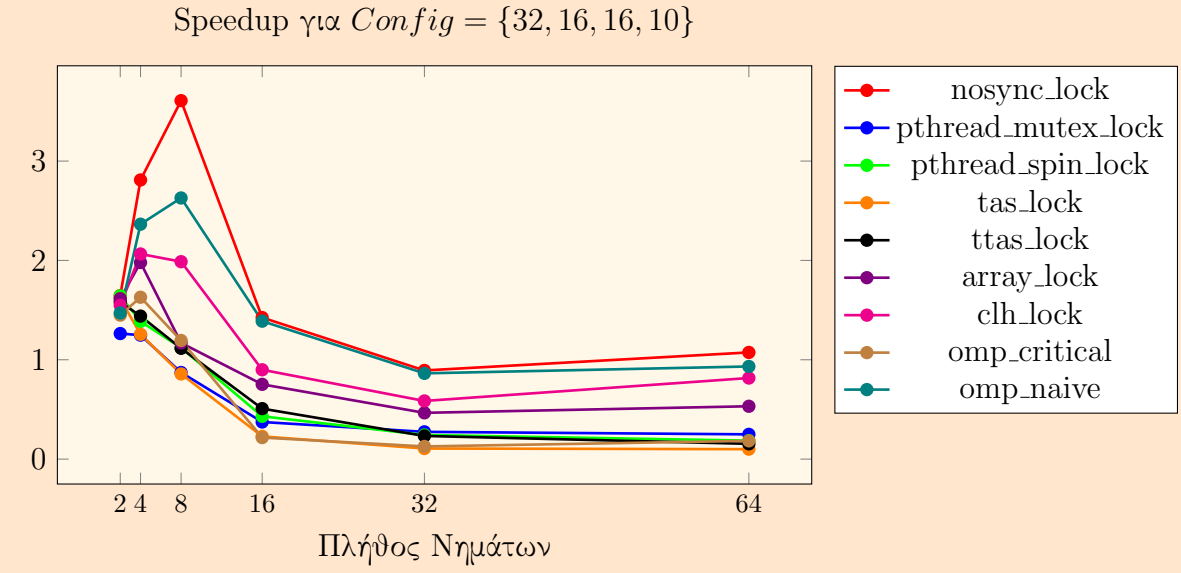
Δεν επιλύθηκε προς το παρόν!

3.3 Αμοιβαίος Αποκλεισμός - Κλειδώματα

Για κάθε είδος κλειδώματος, με κατάλληλη τροποποίηση του *run_on_queue.sh*, τρέχουμε τον K-means καταγράφοντας την επίδοσή του. Παρακάτω, παρουσιάζονται οι χρόνοι εκτέλεσης και το Speedup για κάθε διαφορετική επιλογή κλειδώματος (η δεύτερη εικόνα παρουσιάζει μια μεγέθυνση της πρώτης για μικρό αριθμό νημάτων, ώστε να είναι ευδιάκριτη η συμπεριφορά των κλειδωμάτων).

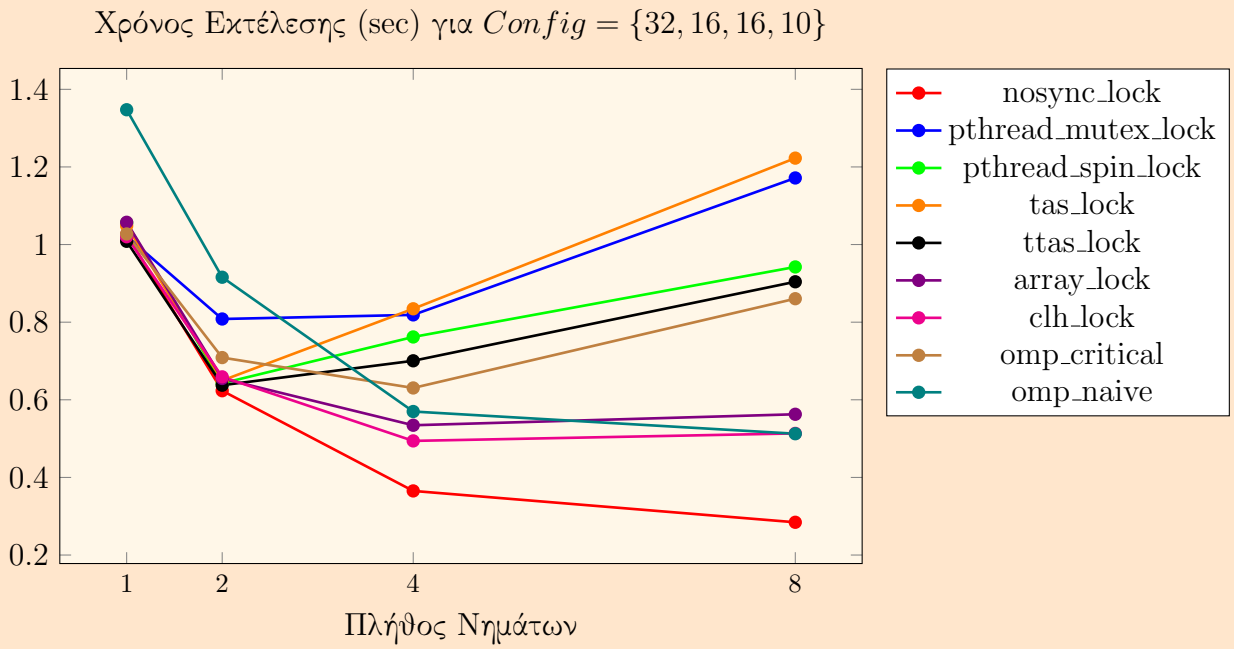


(a) Χρόνος εκτέλεσης ανά τύπο κλειδώματος

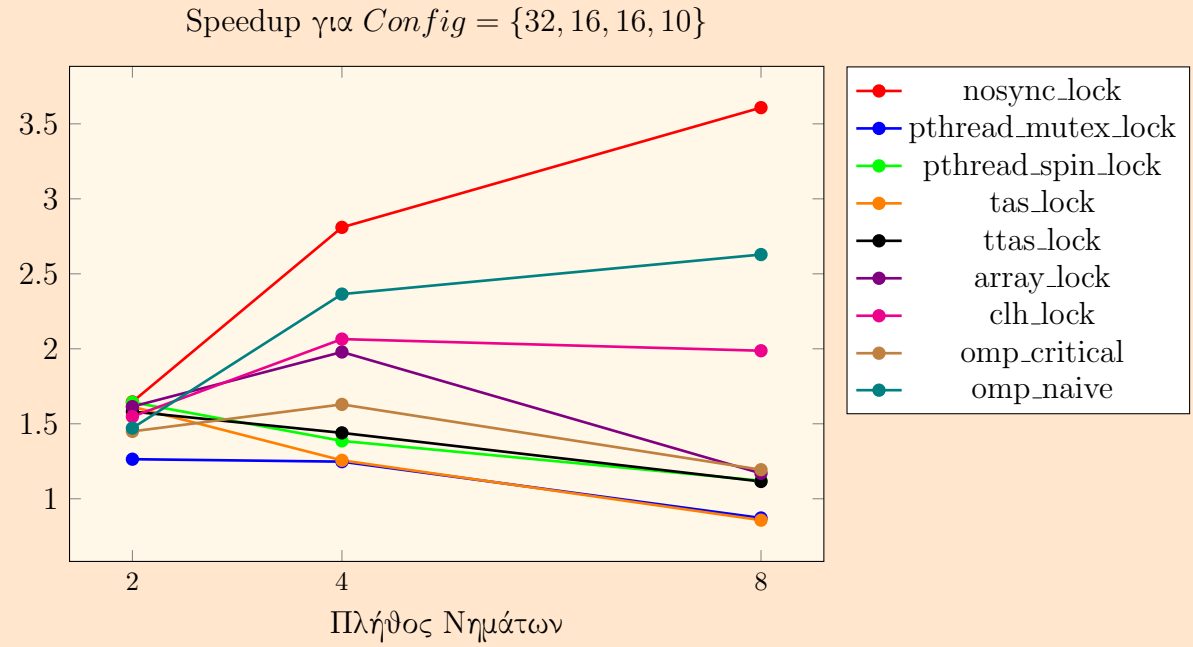


(b) Speedup ανα τύπο κλειδώματος

Εικόνα 9: Speedup και χρόνος εκτέλεσης για $Config = \{32, 16, 16, 10\}$ συναρτήσει του πλήθους νημάτων για κάθε διαφορετικό τύπο κλειδώματος.



(a) Χρόνος εκτέλεσης ανά τύπο κλειδώματος



(b) Speedup ανα τύπο κλειδώματος

Εικόνα 10: Μεγέθυνση : Speedup και χρόνος εκτέλεσης για $Config = \{32, 16, 16, 10\}$ και με 1, 2, 4, 8 νήματα για κάθε διαφορετικό τύπο κλειδώματος.

Οι παρατηρήσεις που εξάγουμε για κάθε έναν από τους διαφορετικούς τρόπους κλειδώματος σχετικά με τον τρόπο λειτουργίας του και την επίδρασή του στην επίδοση του K-means είναι οι παρακάτω :

no_sync_lock Το συγκεκριμένο "κλείδωμα" ουσιαστικά δεν υλοποιεί αμοιβαίο αποκλεισμό καθώς οι υλοποιήσεις των lock_free, lock_aqcuire, lock_release είναι κενές. Ως εκ τούτου, δεν υπάρχει συγχρονισμός και τα νήματα μπορούν ταυτόχρονα να προσπελαύνουν τις κρίσιμες κοινές δομές. Τα race conditions για αυτές τις δομές δεν επιλύονται και για αυτό το λόγο τα αποτελέσματα του προγράμματος είναι εν γένει λανθασμένα. Ωστόσο, η έλλειψη συγχρονισμού και επομένως η απουσία των overheads που προκαλεί, οδηγούν σε πολύ καλούς χρόνους εκτέλεσης. Η υλοποίηση αυτή έχει την καλύτερη επίδοση όπως φαίνεται και στα διαγράμματα και χρησιμοποιείται ως ένα άνω όριο για τις συγκρίσεις μας.

pthread_mutex_lock Για το κλείδωμα αυτό, το νήμα που θέλει να το αποκτήσει προσπαθεί να κάνει ένα atomic test and increment operation προκειμένου να ελέγξει αν το κλειδί είναι δεσμευμένο και αν όχι, να το δεσμεύσει (lock=1). Στην περίπτωση που το κλειδί είναι μη δεσμευμένο, το νήμα θα το δεσμεύσει πολύ γρήγορα χωρίς καθυστερήσεις. Ωστόσο, αν το κλειδί δεν είναι ελεύθερο, θα κληθεί ένα system call (futex) το οποίο θα βάλει το νήμα σε κατάσταση sleep (wait). Το system call αυτό καλεί μια μέθοδο που κάνει το νήμα να περιμένει σε μια μεταβλητή έως ότου αυτή αλλάξει (στην συγκεκριμένη περίπτωση αυτή η μεταβλητή είναι το lock). Όταν το lock αλλάξει (ελευθερωθεί), με ευθύνη του λειτουργικού συστήματος το νήμα θα "ξυπνήσει" καταφέρνοντας πλέον να δεσμεύσει το κλειδί. Να σημειώσούμε εδώ, ότι οι αλλαγές της μεταβλητής lock παρακολουθούνται από το λειτουργικό σύστημα (σε kernel mode) και όχι από το ίδιο το νήμα (user mode), δηλαδή δεν πρόκειται για ένα κλείδωμα τύπου spinlock. Το βασικό μειονέκτημα αυτού του κλειδώματος έγκειται στο γεγονός ότι κάθε φορά που το κλειδί είναι δεσμευμένο και λόγω contention το νήμα θα πρέπει να μπει σε κατάσταση wait, καλείται αναγκαστικά ένα system call, δηλαδή απαιτείται kernel assistance το οποίο όμως εισάγει σημαντικές καθυστερήσεις. Ειδικά αν η εργασία που επιθυμεί να κάνει ένα νήμα δεσμεύοντας το κλειδί είναι μικρή (όπως εν γένει στην περίπτωσή μας), τελικά το overhead του system call δεν συμφέρει και επιφέρει δυσανάλογες καθυστερήσεις σε σχέση με την ποσότητα δουλειάς που το νήμα θέλει να πραγματοποιήσει. Συνεπώς, όπως φαίνεται και στα επόμενα σχήματα, το συγκεκριμένο κλείδωμα επιτυγχάνει αρκετά κακή επίδοση η οποία χειροτερεύει μάλιστα με την αύξηση του πλήθους των νημάτων η οποία προκαλεί αύξηση του contention, περισσότερες αποτυχίες δέσμευσης του κλειδιού, περισσότερα system calls futex κ.ο.κ.

pthread_spin_lock Στη συγκεκριμένη περίπτωση, για την δέσμευση του κλειδιού, ελέγχεται η κατάστασή του και αν είναι ελεύθερο τότε δεσμεύεται γρήγορα και αποδοτικά, ωστόσο αν είναι δεσμευμένο το νήμα σκανάρει με τρόπο busy wait την κατάσταση του lock ώστε να ελέγξει πότε αυτό θα ελευθερωθεί. Ως εκ τούτου, από την μία δεν χρησιμοποιούνται system calls και έτσι δεν εμπλέκεται το λειτουργικό σύστημα το οποίο θα προσέθετε καθυστερήσεις, από την άλλη η διαδικασία του busy wait δεσμεύει συνεχώς τον επεξεργαστή σε όλη τη διάρκεια ελέγχου της τιμής του κλειδιού μέχρι αυτό να ελευθερωθεί. Στην περίπτωσή μας, θα έλεγε κανείς ότι αυτό δεν αποτελεί πρόβλημα καθώς στον sandman κάθε νήμα δεσμεύει αποκλειστικά έναν επεξεργαστή και κάθε επεξεργαστής αφιερώνεται αποκλειστικά σε ένα νήμα (αφού δεν τρέχουμε ταυτόχρονα άλλες εργασίες εκτός του K-means στον sandman). Πράγματι, αυτό συμβαίνει μέχρι τα 32 νήματα όπου το busy wait των νημάτων δεν είναι τόσο άσχημο (και οι επιδόσεις είναι καλύτερες από αυτές του pthread_mutex_lock) καθώς κάθε νήμα δεσμεύει ένα από τα 32 cores του sandman (τα οποία δεν επιθυμεί να τα χρησιμοποιήσει άλλο νήμα) και κάνει busy wait σε αυτό χωρίς να εμποδίζει τη λειτουργία άλλων νημάτων. Ωστοσο, δεν πρέπει να παραλείψουμε ότι ο sandman ενώ έχει 64 λογικά cores λόγω hyperthreading, διαθέτει 32

φυσικούς πυρήνες. Επομένως, όταν χρησιμοποιούμε 64 νήματα, σε κάθε πυρήνα βρίσκονται δύο νήματα, κάθε ένα από τα οποία όταν κάνει busy wait εμποδίζει το άλλο νήμα του core. Εξαιτίας αυτού, για 64 νήματα παρατηρούμε χειρότερες επιδόσεις.

tas_lock Αρχικά, εξηγούμε τη λειτουργία αυτού του μηχανισμού κλειδώματος, το οποίο αποτελεί atomic instruction υλοπιημένο σε επίπεδο hardware. Μέσω του μηχανισμού Test And Set όταν ένας επεξεργαστής (στον οποίο εκτελείται κάποιο νήμα) επιθυμεί να πάρει το κλειδί για είσοδο στην κρίσιμη περιοχή, αρχικά κάνει έναν έλεγχο (Test) για το αν το κλειδί είναι ή όχι δεσμευμένο. Αν δεν είναι δεσμευμένο, το δεσμεύει (ατομικά) θέτοντάς το ως δεσμευμένο (Set), διαφορετικά επαναλαμβάνει τη διαδικασία, δηλαδή επιμένει στο να ελέγχει διαρκώς το κλειδί (spinlock) έως ότου αποδεσμευτεί από τον επε- ξεργαστή που το χρησιμοποιεί. Ωστόσο, υποθέτοντας χρήση invalidation πρωτοκόλλου συνάφειας, η Test And Set είναι μια λειτουργία με πρόθεση τροποποίησης του διαμοιραζόμενου πόρου (κλειδιού) και ως εκ τούτου ο επεξεργαστής που την χρησιμοποιεί μεταβαίνει σε κατάσταση Modified με ταυτόχρονη διαγραφή όλων των αντιγράφων του κλειδιού στις caches όλων των άλλων επεξεργαστών οι οποίοι ως εκ τούτου μεταβαίνουν σε κατάσταση Invalid, η οποία στη συνέχεια θα έχει ως αποτέλεσμα cache miss σε όλους αυτούς τους επεξεργαστές που θα θελήσουν αργότερα να έχουν πρόσβαση στην τιμή του κοινού κλειδιού και ως εκ τούτου για να ξαναλάβουν αυτήν την τιμή θα δημιουργήσουν επιπλέον κίνηση στο διάδρομο. Όλοι οι επεξεργαστές προσπαθούν με πρόθεση εγγραφής (Modified) να θέσουν ένα (ενδεχομένως ήδη δεσμευμένο) κλειδί, οπότε και αποτυγχάνουν να το δεσμεύσουν και σκοτώνουν όλα τα άλλα αντίγραφα δημιουργώντας πολλή κίνηση (και μάλιστα μη χρήσιμη κίνηση) στον διάδρομο (bus) με αρνητική επίπτωση στην επίδοση. Κατόπιν όλων αυτών, εξηγείται και η συμπεριφορά αυτού του κλειδώματος όπως φαίνεται στα διαγράμματα. Το Test and Set, οδηγεί σε πολύ κακές επιδόσεις και συγκεκριμένα με χρήση άνω των 2 νημάτων, αυτό το κλείδωμα επιτυγχάνει τις χειρότερες επιδόσεις σχετικά με κάθε άλλο κλείδωμα. Για 2 νήματα, αν και έχει επίσης μη ικανοποιητική επίδοση, δεν είναι το χειρότερο όλων καθώς τα νήματα είναι πολύ λίγα και τα overheads του test and set (όπως αυτά εξηγήθηκαν π.χ. κίνηση στο διάδρομο, invalidations κλπ) δεν καταφέρνουν ακόμη να επικρατήσουν.

ttas_lock Ο μηχανισμός (hardware atomic instruction επίσης) Test and Test And Set (TTAS) εισάγει μια βελτίωση στον TAS. Αντί οι επεξεργαστές να προσπαθούν με πρόθεση εγγραφής να τροποποιήσουν το κλειδί, πρώτα με πρόθεση ανάγνωσης (το πρώτο Test) ελέγχουν αν το κλειδί είναι δεσμευμένο. Αν δεν είναι δεσμευμένο τότε προχωρούν στην γνωστή διαδικασία Test And Set, διαφορετικά επαναλαμβάνουν τη διαδικασία από την αρχή, δηλαδή κάνουν spin πάνω στην εντολή Test (το πρώτο Test) με πρόθεση ανάγνωσης του κλειδιού. Δηλαδή αντί να προσπαθούν επαναλαμβανόμενα να αποκτήσουν πρόσβαση με πρόθεση τροποποίησης του κλειδιού, προσπαθούν απλά με πρόθεση ανάγνωσης (κατάσταση Shared του πρωτοκόλλου συνάφειας) να το διαβάσουν και μόνο αν δεν είναι δεσμευμένο προχωρούν στην προσπάθεια τροποποίησής του. Ως εκ τούτου, δεν κάνουν Invalid τα αντίγραφα όλων των άλλων επεξεργαστών παρά μόνο όταν είναι σίγουροι [6] πως το κλειδί είναι διαθέσιμο και μπορούν πράγματι να το τροποποιήσουν. Ως εκ τούτου παράγεται αρκετά λιγότερη κίνηση στον διάδρομο με αποτέλεσμα οι επιδόσεις να είναι καλύτερες σχετικά με το Test and Set. Βέβαια, από τα διαγράμματα είναι εμφανές πως το κλείδωμα αυτό εν γένει δεν επιτυγχάνει καλές επιδόσεις. Το

[6] Προφανώς, βέβαια, είναι πιθανό αρκετά νήματα να δουν ταυτόχρονα ότι το κλειδί είναι ελεύθερο, να συμπεράνουν ότι μπορούν να το δεσμεύσουν και όλα μαζί να προχωρήσουν στην διαδικασία Test and Set με όλα τα αρνητικά που μπορεί αυτό να έχει (και που εξηγήθηκαν στην παραπάνω παράγραφο για το Test and Set)

γεγονός ότι απαλλαχθήκαμε από τα πολλά invalidations του TAS, δεν σημαίνει πως απαλλαχθήκαμε και απο τα αρνητικά της spinlock λογικής που ακολουθεί αυτό το κλείδωμα. Μάλιστα, παρατηρούμε ότι για αυτό το λόγο, το TTAS επιτυγχάνει πολύ παρόμοιες επιδόσεις με το pthread_spin_lock (με τη διαφορά ότι το spinlock γίνεται στο TTAS σε επίπεδο hardware και όχι software).

array_lock Τα προηγούμενα κλειδώματα είχαν το βασικό μειονέκτημα ότι οδηγούσαν όλα τα νήματα να συνωστίζονται σε μια κοινή θέση μνήμης (αυτή του μοναδικού κλειδιού), κάτι το οποίο μπορεί να γίνει προβληματικό σε αρχιτεκτονικές κοινής μνήμης. Η ιδεά αυτού του κλειδώματος είναι να λύσει αυτό το πρόβλημα. Κάθε νήμα θα εργάζεται και θα κάνει spinlock σε μια δική του θέση μνήμης (στον δικό του χώρο), ενώ η πρόσβαση σε κοινές θέσεις μνήμης θα είναι σπάνια και όταν συμβαίνει θα συμμετέχουν σε αυτήν λίγα νήματα. Συγκεκριμένα, για το κλείδωμα αυτό χρησιμοποιείται μια κυκλική δομή array μεγέθους ίσου με το πλήθος των νημάτων και μια κοινή μεταβλητή tail που υποδεικνύει την θέση στην οποία θα μπει το επόμενο νήμα που θα επιχειρήσει να κάνει lock. Κάθε θέση του array είναι ένα flag (true ή false) που δηλώνει αν το νήμα που κατέχει την εκάστοτε θέση μπορεί ή όχι να μπει στο κρίσιμο τμήμα. Αρχικά, το tail είναι 0 και η πρώτη θέση του array ίση με true, ώστε το πρώτο νήμα που θα ζητήσει πρόσβαση στο κρίσιμο τμήμα να τα καταφέρει. Κάθε φορά που ένα νήμα κάνει lock, διαβάζει την τιμή του tail ώστε να καταλάβει μια συγκεκριμένη θέση του array και αυξάνει την μεταβλητή αυτή κατά 1. Αυτή η λειτουργία οφείλει να γίνει ατομικά, καθώς πολλά νήματα ταυτόχρονα ενδέχεται να κάνουν lock. Ωστόσο, αυτό δεν γίνεται τόσο συχνά. Αφού το νήμα, τοποθετήθηκε σε μια συγκεκριμένη θέση του array, κάνει spinlock σε αυτήν έως ότου το flag του γίνει true. Αυτό το spinlock γίνεται απρόσκοπτα από το κάθε νήμα στον δικό του χώρο, χωρίς να δημιουργείται contention. Όταν ένα νήμα κάνει unlock, θέτει το δικό του flag ίσο με false και το flag της επόμενης κυκλικά θέσης ίσο με true, παραχωρώντας στο επόμενο νήμα το δικαίωμα πρόσβασης στο κρίσιμο τμήμα. Σε αυτό το σημείο, ένα νήμα αποκτά πρόσβαση (με πρόθεση εγγραφής) στο χώρο ενός άλλου νήματος στο array. Ωστόσο, αυτό δεν γίνεται συχνά και όταν γίνεται συμμετέχουν σε αυτό δύο νήματα, αυτό που κάνει unlock και αυτό που κάνει spin σε αυτή τη θέση. Επίσης, να σημειωθεί ότι με αυτό το κλείδωμα επιτυγχάνεται δικαιοσύνη, καθώς κάθε νήμα θα αποκτήσει δικαίωμα πρόσβαση στο κρίσιμο τμήμα σύμφωνα με τη σειρά με την οποία ζήτησε πρόσβαση (λόγω αυτής της κυκλικής δομής) και αποφυγή του starvation καθώς σίγουρα κάποια στιγμή το δικαίωμα πρόσβασης στο κρίσιμο τμήμα θα φτάσει μέσω της κυκλικής αυτής διαδικασίας σε οποιοδήποτε νήμα συμμετέχει σε αυτήν. Από τα διαγράμματα, παρατηρούμε ότι το συγκεκριμένο κλείδωμα επιτυγχάνει συγκριτικά καλές επιδόσεις (είναι μάλιστα τρίτο σε σειρά μετά τα omp_naive και clh_lock) που οφείλονται στα παραπάνω πλεονεκτήματα της αποφυγής συμφόρησης σε μια μοναδική θέση μνήμης. Ωστόσο, όσο αυξάνεται το πλήθος των νημάτων και δεδομένου ότι κάθε νήμα εκτελεί μια σχετικά μικρή εργασία στο κρίσιμο τμήμα, τα overheads πρόσβασης στην μεταβλητή tail και ενημέρωση του επόμενου slot στον array κατά τη διαδικασία του unlock προσθέτουν δυσανάλογα overheads συγκριτικά με την ποσότητα χρήσιμης δουλειάς που εκτελούν τα νήματα. Μάλιστα, πρέπει να συνυπολογίσουμε ότι όσο περισσότερα είναι τα νήματα, τόσο μεγαλύτερη είναι και η κυκλική δομή με συνέπεια μεγαλύτερες καθυστερήσεις στην λήψη πρόσβασης στο κρίσιμο τμήμα (στην χειρότερη περίπτωση, ένα νήμα θα πρέπει να περιμένει όλα τα υπόλοιπα N-1 νήματα να ολοκληρώσουν την δουλειά τους προκειμένου να αποκτήσει πρόσβαση στο κρίσιμο τμήμα).

clh_lock Το κλείδωμα αυτό ακολουθεί ως προς τα βασικά σημεία την λογική του array_lock χρησιμοποιώντας όμως τη δομή μιας συνδεδεμένης λίστας. Κάθε κόμβος (Qnode) αποτελείται από μια τριπλέτα my_node, pred_node (δείκτης στον predecessor) και locked (μια boolean μεταβλητή που είναι true όταν ο συγκεκριμένο κόμβος είτε αναμένει να λάβει το κλειδί είτε βρίσκεται ήδη στο κρίσιμο τμήμα και false όταν έχει κάνει release το lock). Μια μεταβλητή tail δείχνει δείχνει στον πιο πρόσφατα added κόμβο στην linked list. Κάθε νήμα-κόμβος της λίστας σκανάρει την μεταβλητή locked του predecessor του έως ότου αυτή γίνει false, οπότε και μπορεί πλέον να εισέλθει στο κρίσιμο τμήμα. Όταν ένας κόμβος εξέλθει του κρίσιμου τμήματος έχει ευθύνη μόνο να ενημερώσει την locked μεταβλητή του σε false. Για λόγους αποφυγής σπατάλης χώρου και ως εναλλακτική του garbage collection (όπου αυτό υπάρχει ήδη), υπάρχει και πρόβλεψη το νήμα που καταφέρνει να εισέλθει στο κρίσιμο τμήμα, να χρησιμοποιεί πλέον ως κόμβο αυτόν του predecessor ο οποίος πλέον δεν χρησιμοποιείται (αφού αυτός απελευθέρωσε το κλείδωμα). Οι δύο βασικές διαφορές του clh από το array_lock είναι οι εξής :

- η χρησιμοποιούμενη δομή είναι πλέον δυναμική και δεν δημιουργείται εξαρχής με μέγεθος ίσο με το πλήθος νημάτων. Μάλιστα, με διάφορες τεχνικές (όπως αυτή που αναφέρθηκε παραπάνω) μπορεί να αποφευχθεί ακόμη περισσότερο η σπατάλη χώρου που γινόταν στο array_lock. Το κάθε νήμα που συμμετέχει στη διαδικασία δεν χρειάζεται να γνωρίζει το πλήθος των νημάτων που επίσης συμμετέχουν, μειώνοντας έτσι την πολυπλοκότητα υπολογισμών του κάθε νήματος.

- πλέον, το ενδιαφερόμενο (για πρόσβαση στο κρίσιμο τμήμα) νήμα είναι αυτό που κάνει spin ελέγχοντας την απελευθέρωση του κρίσιμου τμήματος. Στο array_lock το εξερχόμενο από το κρίσιμο τμήμα νήμα ήταν υπεύθυνο για την ενημέρωση του επόμενου ενδιαφερόμενου νήματος γεγονός που ενδέχεται να δημιουργούσε καθυστέρηση στην περίπτωση που το εξερχόμενο νήμα γινόταν schedule out αμέσως μετά την έξοδο από το κρίσιμο τμήμα και πριν προλάβει να ειδοποιήσει το επόμενο ενδιαφερόμενο νήμα. Τώρα το ενδιαφερόμενο νήμα, αμέσως μόλις καταλάβει ότι το κρίσιμο τμήμα είναι ελεύθερο, μπορεί να μπει σε αυτό.

Από τα διαγράμματα, παρατηρούμε ότι το clh_lock επιτυγχάνει πολύ καλές επιδόσεις συγκριτικά με τα υπόλοιπα κλειδώματα (καλύτερες από το array_lock) αφού έχει όλα τα πλεονεκτήματα του array_lock στα οποία προστίθενται όμως και τα παραπάνω.

Συνολικά για τα παραπάνω κλειδώματα Όλα τα προηγούμενα κλειδώματα εν γένει δεν επιτυγχάνουν ιδιαίτερα καλές επιδόσεις (ειδικά για μεγάλο αριθμό νημάτων) καθώς το Speedup στις περισσότερες περιπτώσεις είναι χαμηλό και μάλιστα για περισσότερα των 16 νημάτων το speedup γίνεται μικρότερο του 1 (επίδοση χειρότερη από το σειριακό). Αυτό οφείλεται στα δημιουργούμενα overheads εξαιτίας της χρήσης των locks ειδικά όταν για την απόκτησή τους ανταγωνίζονται πολλά νήματα.

omp_critical Στη συγκεκριμένη περίπτωση, ορίζουμε μέσω του OpenMP ως critical section όλο εκείνο το τμήμα κώδικα που ενημερώνει τις δύο κρίσιμες διαμοιραζόμενες δομές. Ωστόσο με τον τρόπο αυτό, μόνο ένα νήμα κάθε φορά μπορεί να τις ενημερώνει, μειώνοντας δραματικά τον βαθμό παραλληλισμού, καθώς δεν επιτρέπουμε ακόμη και το να ενημερώνουν διαφορετικά νήματα, διαφορετικές θέσεις των διαμοιραζόμενων δομών. Ως εκ τούτου, η επίδοση αυτού του κλειδώματος είναι ιδαίτερα κακή, ενδεχομένως και εξαιτίας του ειδικού τρόπου κλειδώματος

που υλοποιεί το #pragma omp critical. Θα μποτούσαμε να πούμε ότι η επίδοσή του μοιάζει (βλ. Speedup) με αυτήν του tas_lock, ιδιαίτερα για περισσότερα από 8 νήματα.

omp_naive Πρόκειται για την δική μας υλοποίηση στο section 3.1. Σε αυτήν την περίπτωση, εκτός του ότι χρησιμοποιούνται hardware implemented atomic instructions για την εντολή που περικλείει το κάθε #pragma omp atomic (επιτυγχάνοντας καλύτερες επιδόσεις), δεν ορίζουμε ένα μεγάλο κρίσιμο τμήμα που περιλαμβάνει την πλήρη ενημέρωση των δύο διαμοιραζόμενων δομών αλλά ουσιαστικά χωρίζουμε την συνολική δουλειά των νημάτων σε μικρότερα κομμάτια δημιουργώντας περισσότερο παραλληλισμό (για παράδειγμα όταν ένα νήμα ενημερώνει την μια δομή, ένα άλλο νήμα μπορεί ταυτόχρονα να ενημερώνει την άλλη και όσο ένα νήμα βρίσκεται σε ένα iteration ενημερώνοντας ένα συγκεκριμένο τμήμα του new_Clusters ταυτόχρονα ένα άλλο νήμα μπορεί να βρίσκεται σε κάποιο άλλο iteration ενημερώνοντας ένα άλλο τμήμα του πίνακα). Για τους λόγους αυτούς, παρατηρούμε στα διαγράμματα την υλοποίηση αυτή να επιτυγχάνει τις καλύτερες επιδόσεις συγκριτικά με τις υπόλοιπες υλοποιήσεις.

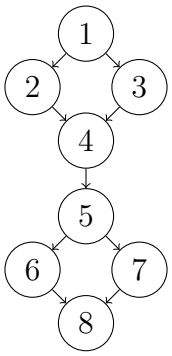
4 Παραλληλοποίηση της αναδρομικής εκδοχής του αλγορίθμου Floyd-Warshall

Εισαγωγή

Στο δεύτερο μέρος αυτής της εργαστηριακής άσκησης, καλούμαστε να παραλληλοποιήσουμε την αναδρομική εκδοχή του αλγορίθμου Floyd-Warshall με χρήση OpenMP tasks. Αρχικά, προκειμένου να ακολουθήσουμε μια ανάλυση βασιμένη σε tasks, απαιτείται η γνώση της λειτουργίας του αλγορίθμου ώστε να εντοπιστούν οι εργασίες που τον αποτελούν όπως επίσης και οι εξαρτήσεις μεταξύ τους (και ως εκ τούτου οι δυνατότητες παραλληλισμού).

4.1 Γράφος εξαρτήσεων, σχεδιασμός και υλοποίηση παραλληλοποίησης

Ο γράφος εξαρτήσεων για τον αναδρομικό Floyd Warshall παρατίθεται στην εικόνα 11 όπου κάθε κόμβος αναπαριστά μια αναδρομική κλήση FW_SR του αλγορίθμου όπως θα φανεί στον κώδικα στη συνέχεια.



Εικόνα 11: Γράφος Εξαρτήσεων Αναδρομικού Floyd-Warshall

Παρατηρούμε ότι ο μέγιστος βαθμός παραλληλισμού είναι δύο tasks. Συγκεκριμένα, από τον παραπάνω γράφο, συμπεραίνουμε ότι η δομή του αλγορίθμου μας θα πρέπει να μοιάζει όπως παρακάτω :

- Εκτελείται από ένα νήμα (το αρχικό νήμα) η αναδρομική κλήση 1.
- Το αρχικό νήμα δημιουργεί (προσθέτει στο task pool) την εργασία αναδρομικής κλήσης 2 (*#pragma omp task*), ενώ το ίδιο εκτελεί την εργασία 3 (όσο η εργασία 2 εκτελείται παράλληλα από κάποιο άλλο νήμα που την πήρε από το task pool).
- Το αρχικό νήμα που εκτελούσε την εργασία 3 θα πρέπει να περιμένει την ολοκλήρωση της εργασίας 2 (*#pragma omp taskwait*), ώστε στη συνέχεια να προχωρήσει εκτελώντας (το ίδιο) την εργασία 4 και έπειτα την εργασία 5 σειριακά, όπως απαιτούν οι εξαρτήσεις.
- Το αρχικό νήμα δημιουργεί (προσθέτει στο task pool) την εργασία αναδρομικής κλήσης 6 (*#pragma omp task*), ενώ το ίδιο εκτελεί την εργασία 7 (όσο η εργασία 6 εκτελείται παράλληλα από κάποιο άλλο νήμα που την πήρε από το task pool).

- Το αρχικό νήμα περιμένει την ολοκλήρωση της εργασίας 6, ώστε στη συνέχεια να προχωρήσει στην εκτέλεση της εργασίας 8.

Σημειώνουμε ότι το γεγονός ότι οι εργασίες (1, 3) και (5, 7) εκτελούνται από το ίδιο νήμα (το αρχικό νήμα) συμβάλλει στην αξιοποίηση της τοπικότητας. Για παράδειγμα, η εργασία 3 χρειάζεται δεδομένα λόγω εξάρτησης από την εργασία 1. Η εκτέλεσή τους από το ίδιο νήμα (αν υποθέσουμε ότι κάθε νήμα εκτελείται πάντα από τον ίδιο επεξεργαστή[7])εξασφαλίζει την διατήρηση στην cache δεδομένων της εργασίας 1 που θα χρειαστεί η εργασία 3. Ως εκ τούτου, δεν θα απαιτηθεί επικοινωνία μεταξύ διαφορετικών cores καθώς οι πληροφορίες και για τις δύο εργασίες θα βρίσκονται στην cache του ίδιου core (όπου τρέχει το αρχικό νήμα). Επιπλέον, οι δομές A, B, C δεν χρειάζεται να γίνουν private για κάθε διαφορετική εργασία καθώς κάθε μία από αυτές προσπελαύνει διαφορετικό τμήμα αυτών των πινάκων. Οι πίνακες αυτοί θέλουμε επομένως να ειναι shared (που είναι by default οπότε δεν σημειώνεται ρητά στα αντίστοιχα directives).

Μια σημαντική επισήμανση για τον κώδικα που ακολουθεί είναι η τοποθέτηση των directives *#pragma omp parallel* και *#pragma omp single* με τέτοιο τρόπο ώστε να περικλείουν την αρχική κλήση της αναδρομικής συνάρτησης και όχι το σώμα της αναδρομικής συνάρτησης. Η χρήση των δύο αυτών directives αποσκοπεί αντίστοιχα στον ορισμό μιας παράλληλης περιοχής που θα εκτελεστεί από πολλά νήματα (καθώς και στην δημιουργία τους) και στον ορισμό μιας περιοχής που θα ελέγχει/διαχειρίζεται (δηλαδή θα δημιουργεί ή θα αναλαμβάνει εργασίες) ένα νήμα (το αρχικό). Η τοποθέτησή τους με τον τρόπο αυτό, σημαίνει την συνολική παράλληλη εκτέλεση της αναδρομικής κλήσης με τρόπο τέτοιο ώστε πολλαπλά νήματα να αναλαμβάνουν και να εκτελούν εργασίες από το task pool. Αν τα δύο αυτά directives τοποθετούνταν στο σώμα της αναδρομικής συνάρτησης, θα καλούνταν σε κάθε αναδρομική κλήση, κάτι που ούτως ή άλλως φαίνεται λογικά παράδοξο αλλά επιβεβαιώθηκε κιόλας πρακτικά. Παρατηρήσαμε ότι μια τέτοια τοποθέτηση των directives οδηγεί σε παράξενη συμπεριφορά κατά την οποία με χρήση άνω των δύο νημάτων η επίδοση σταθεροποιείται σαν να εξακολουθούν να υπάρχουν και να λειτουργούν μόνο δύο νήματα, παρά τον ορισμό παραπάνω νημάτων.

Κώδικας

Παρακάτω παρατίθεται ο κώδικας, στον οποίο φαίνονται όλα τα σημεία που εξηγήθηκαν παραπάνω. Αρχικά, παρατίθεται το τμήμα κώδικα όπου καλείται η αναδρομική συνάρτηση :

```
1       #pragma omp parallel
2       {
3               #pragma omp single
4               {
5                       FW_SR(A,0,0, A,0,0,A,0,0,N,B);
6               }
7       }
```

Εν συνεχεία, παρουσιάζονται οι τροποποιήσεις στο σώμα της αναδρομικής συνάρτησης όπου έχει προστεθεί ο παραλληλισμός σε επίπεδο tasks:

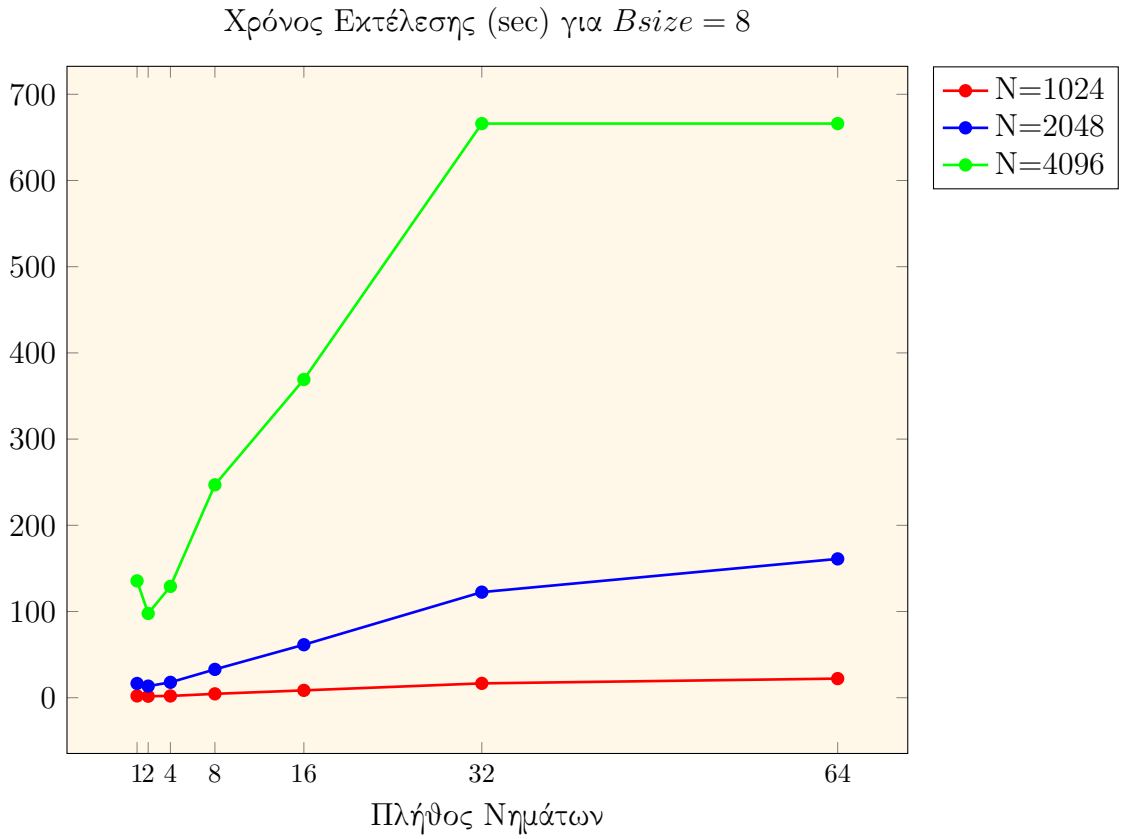
[7]...δηλαδή ότι ένα νήμα είναι προσδεδεμένο σε έναν επεξεργαστή μέσω της μεθόδου thread binding και της χρήσης της αντίστοιχης μεταβλητής περιβάλλοντος. Στην δική μας υλοποίηση ενώ αρχικά επιχειρήθηκε η χρήση trhead binding, παρατηρήσαμε ότι σε κάποιες περιπτώσεις αντί να βελτιώνει, τελικά επιδεινώνει ελαφρά την επίδοση.

```
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;
    /*
     * The base case (when recursion stops) is not allowed to be edited!
    * What you can do is try different block sizes.
    */
    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 {
            \\ task 1 of task graph
            FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
            #pragma omp task if(0)
            {
            #pragma omp task
                    // task 2 of task graph
                    FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN
                       /2, myN/2, bsize);
            \\ task 3
            FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN
               /2, bsize);
            #pragma omp taskwait
            }
            // tasks 4 and 5
            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 if(0)
            {
            #pragma omp task
                    // task 6 of task graph
                    FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,
                       crow+myN/2, ccol, myN/2, bsize);
            // task 7
            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
            }
            // task 8
            FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN
               /2, bsize);
        }
}
```

View the .cpp file here *(Dropbox link)*

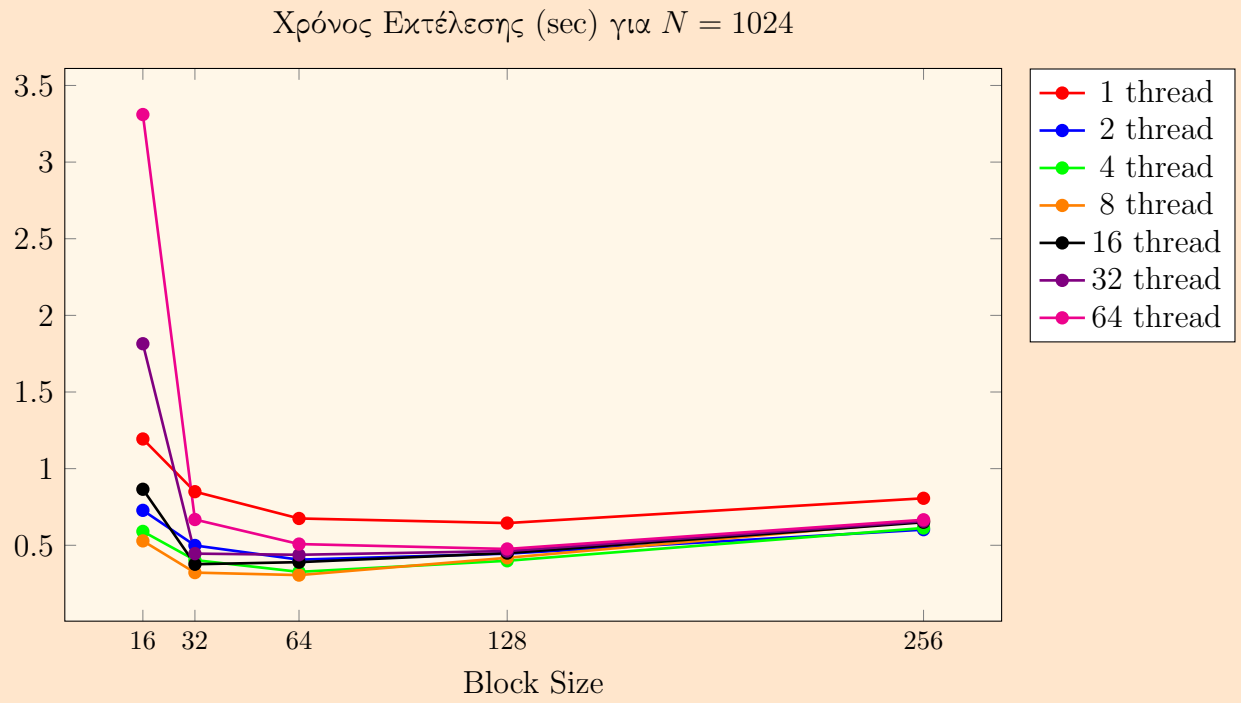
4.2 Διαγράμματα-Παρατηρήσεις

Στη συνέχεια, τρέχουμε τον παραπάνω κώδικα στον sandman για κάθε διαφορετικό configuration, δηλαδή για 1, 2, 4, 8, 16, 32 και 64 νήματα, για μέγεθος γράφου 1024, 2048 και 4096 κόμβων και για block size 8, 16, 32, 64, 128 και 256. Αρχικά παραθέτουμε ξεχωριστά τους χρόνους εκτέλεσης για block size ίσο με 8, καθώς οι χρόνοι για αυτό είναι πολύ μεγάλοι και θα εκμηδένιζαν οπτικά όλες τις άλλες μετρήσεις αν προστίθεντο στα επόμενα συνολικά διαγράμματα. Επιπλέον, επειδή από ένα σημείο και μετά, οι χρόνοι ήταν υπερβολικά μεγάλοι, δεν έγιναν κάποιες εκτελέσεις (για N = 4096, για 32 και 64 νήματα όπου φαίνονται καποιες λάθος τιμές). Για τον ίδιο λόγο, στο επόμενο διάγραμμα Speedup για N=4096, οι τιμές για 32 και 64 νήματα στην γραμμή του block size = 8 (κόκκινη) είναι επίσης αυθαίρετες.

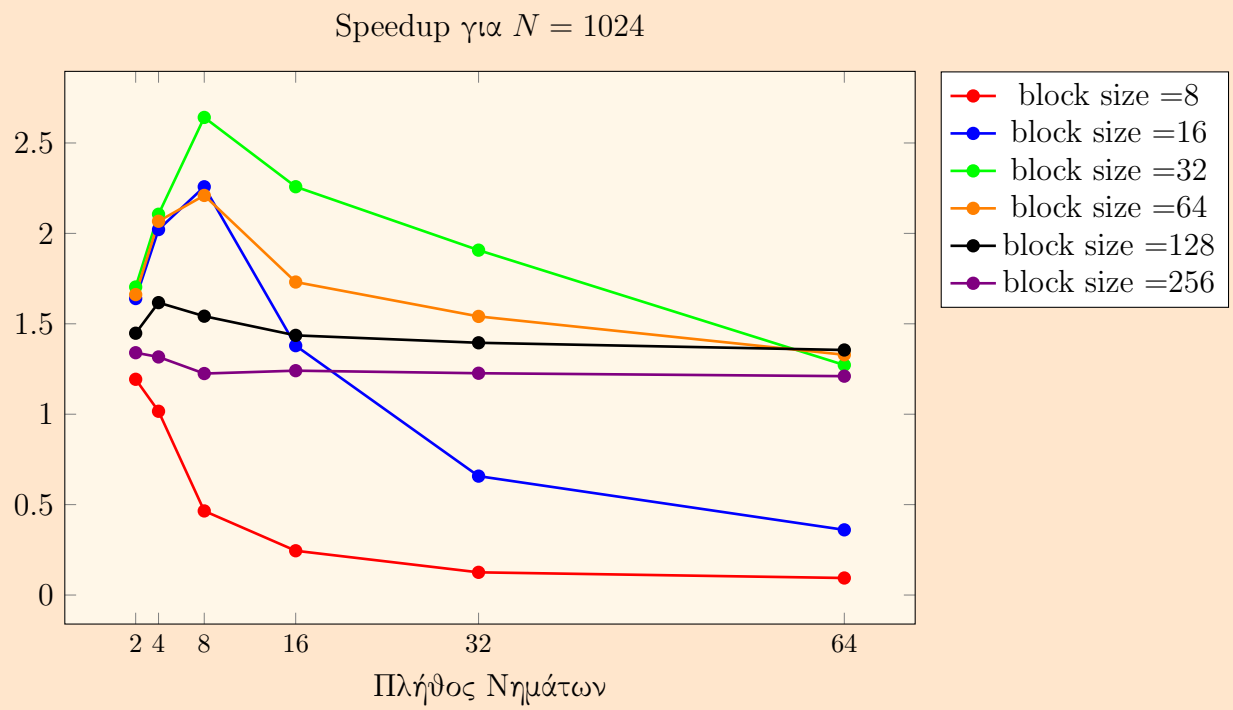


Εικόνα 12: Χρόνος εκτέλεσης για block size ίσο με 8 συναρτήσει του πλήθους νημάτων και του μεγέθους του γράφου.

Στη σύνεχεια παρατίθεται τα υπόλοιπα συνολικά διαγράμματα χρόνου και Speedup στις επόμενες σελίδες.

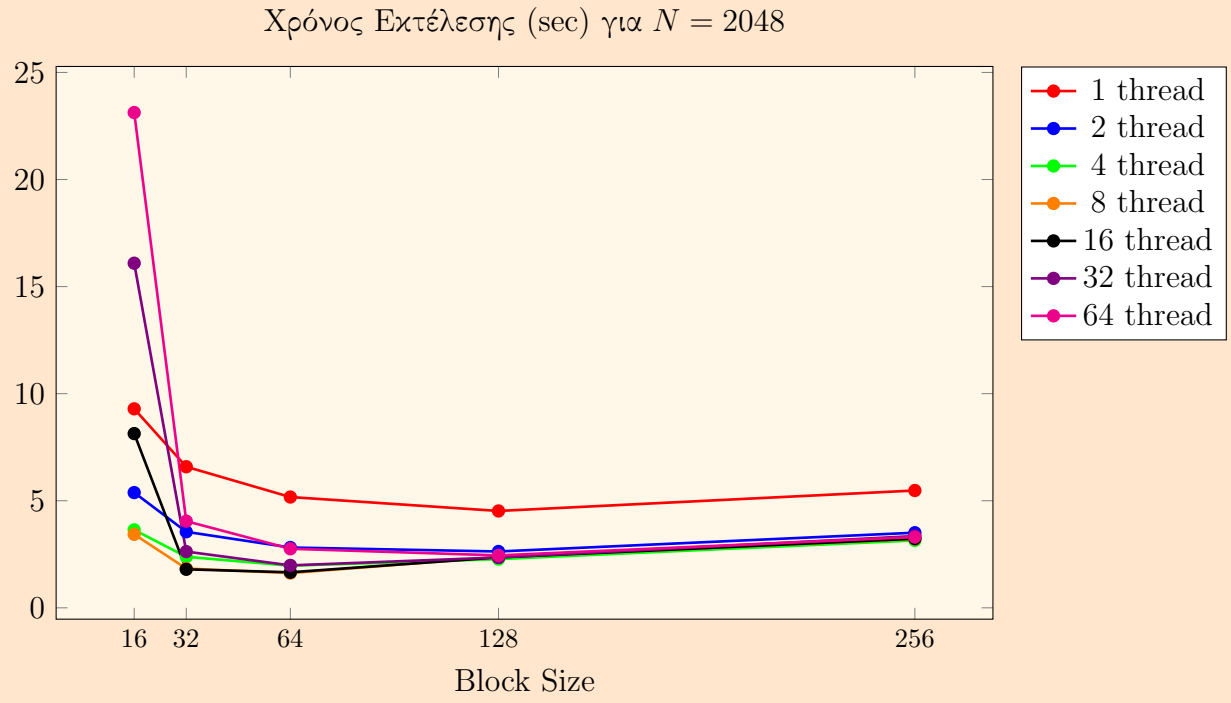


(a) Χρόνος εκτέλεσης ($N = 1024$)

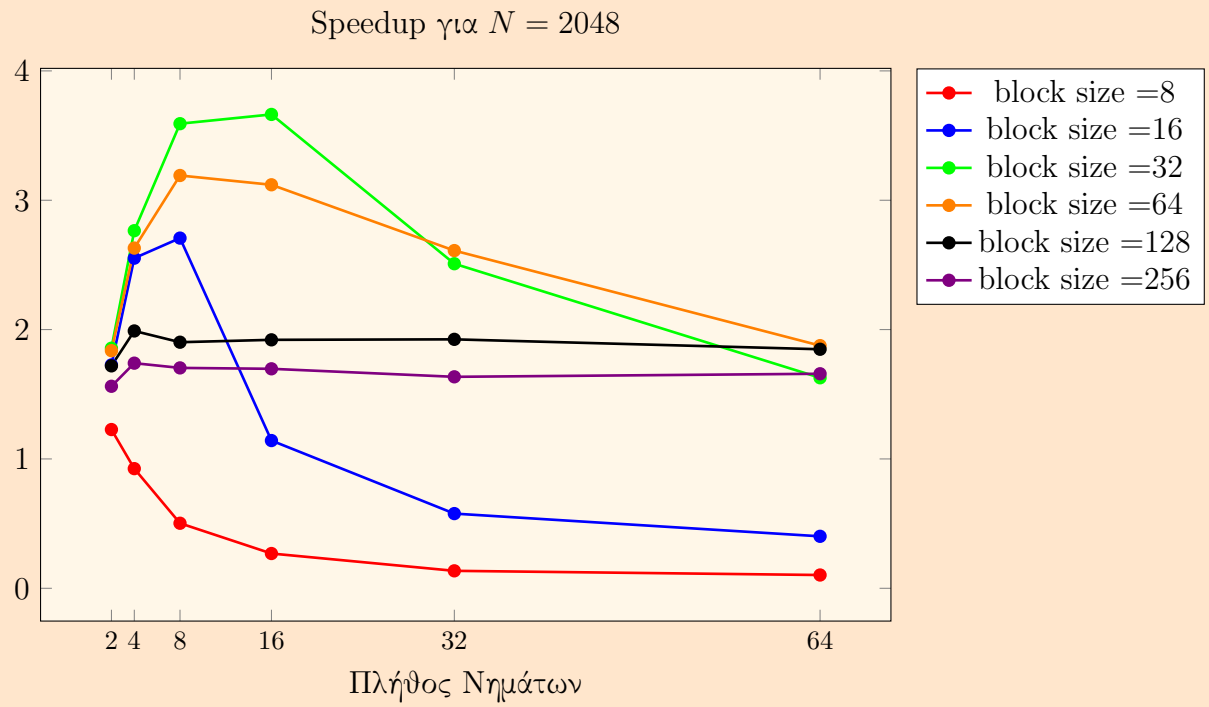


(b) Speedup ($N = 1024$)

Εικόνα 13: Speedup και χρόνος εκτέλεσης για γράφο με $N = 1024$ κόμβους συναρτήσει του πλήθους νημάτων και του χρησιμοποιούμενου block size.

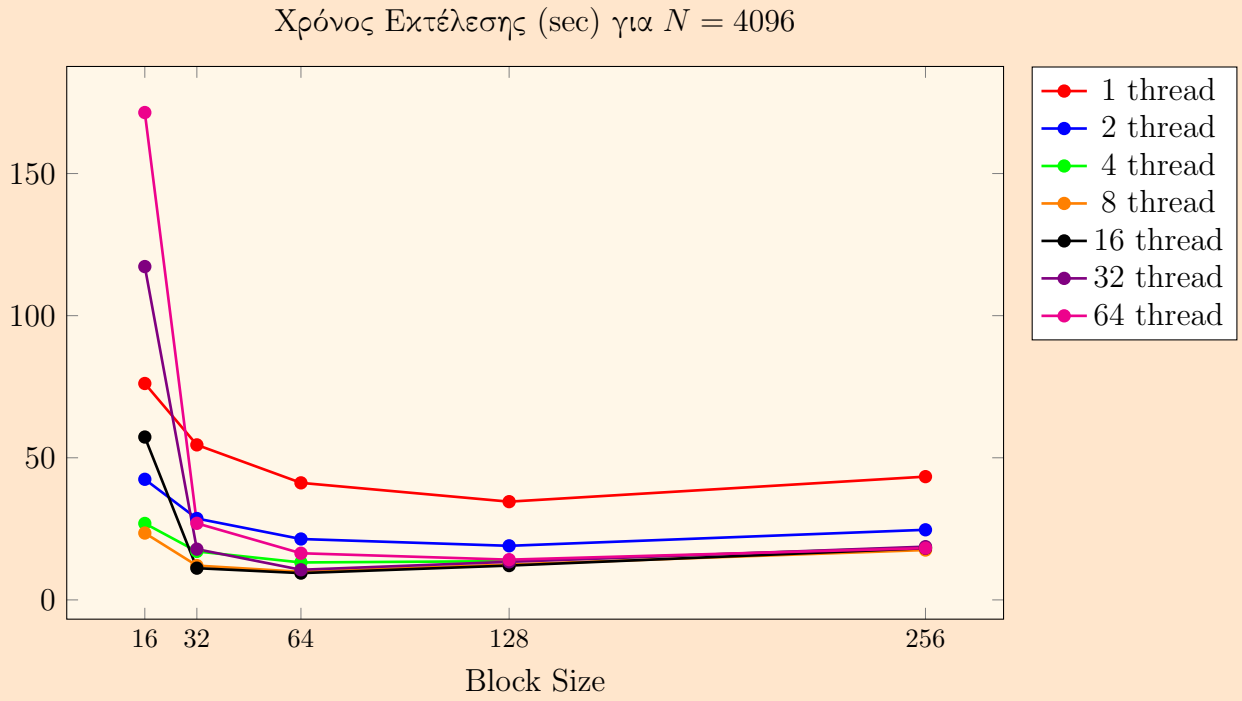


(a) Χρόνος εκτέλεσης ($N = 2048$)

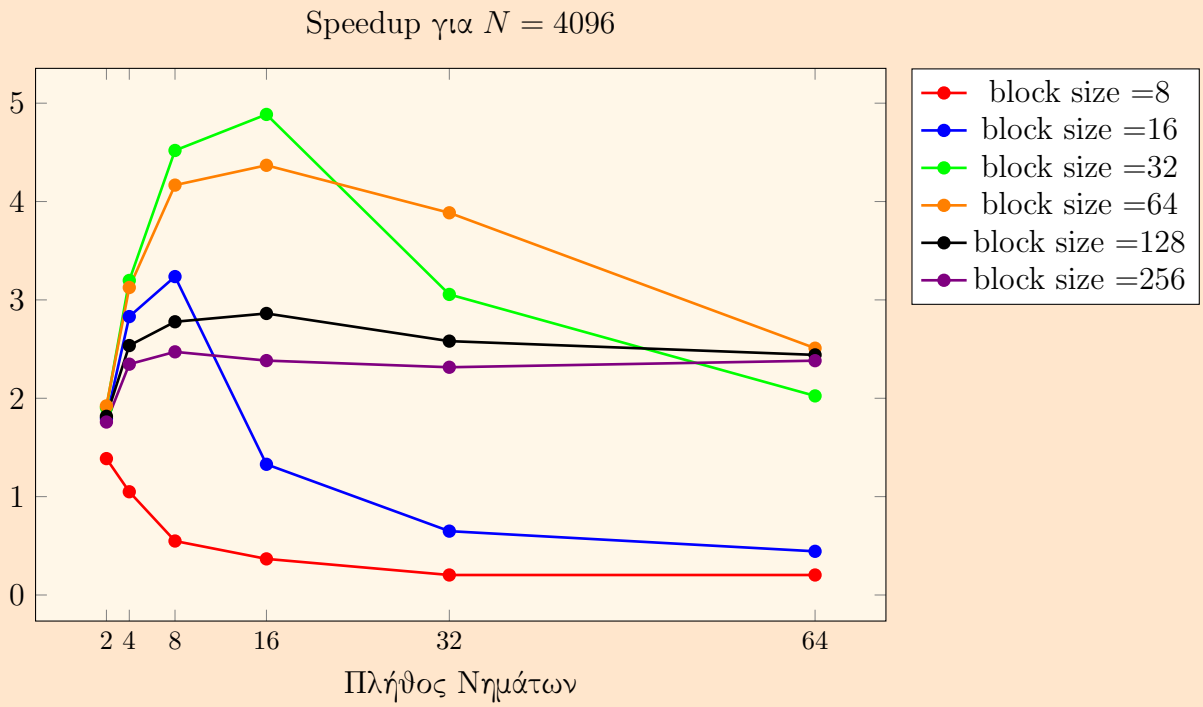


(b) Speedup ($N = 2048$)

Εικόνα 14: Speedup και χρόνος εκτέλεσης για γράφο με $N = 2048$ κόμβους συναρτήσει του πλήθους νημάτων και του χρησιμοποιούμενου block size.



(a) Χρόνος εκτέλεσης ($N = 4096$)



(b) Speedup ($N = 4096$)

Εικόνα 15: Speedup και χρόνος εκτέλεσης για γράφο με $N = 4096$ κόμβους συναρτήσει του πλήθους νημάτων και του χρησιμοποιούμενου block size.

Καταρχάς, είναι προφανές πως οι επιδόσεις και η κλιμακωσιμότητα είναι κάθε άλλο παρά ικανοποιητικά. Στις περισσότερες περιπτώσεις, το Speedup είναι μόνο λίγο μεγαλύτερο της μονάδας (δηλαδή η βελτίωση ως προς το σειριακό είναι πολύ μικρή). Αυτό οφείλεται στο βαθμό παραλληλισμού του αλγορίθμου. Όπως φάνηκε και στον γράφο εξαρτήσεων, μέχρι 2 εργασίες μπορούν να εκτελεστούν παράλληλα και για αυτό το λόγο παρατηρούμε πραγματικά ικανοποιητική επίδοση κυρίως με χρήση 2 νημάτων, ενώ τα περισσότερα νήματα ακόμη κι αν προκαλούν κάποια βελτίωση, αυτή δεν είναι αρκετή (απέχει αρκετά από το γραμμκό Speedup). Η επίδραση δε του block size σχετίζεται με την βάση της αναδρομής. Όταν το block size είναι πολύ μεγάλο (128, 256) δεν παρατηρούμε ιδιαίτερες βελτιώσεις για διαφορετικό αριθμό νημάτων καθώς η αναδρομή φτάνει στην βάση της αρκετά νωρίς (για μεγάλο μέγεθος υποπινάκων) με αποτέλεσμα ένα μεγάλο μέρος του χρόνου εκτέλεσης του προγράμματος να οφείλεται στην εκτέλεση του σειριακού τριπλού for loop της βάσης της αναδρομής (που βασικά αποτελεί τον σειριακό αλγόριθμο). Αντιστρόφως, για πολύ μικρό block size (π.χ. 8), η αναδρομή σπάει τον πίνακα γειτνίασης σε πάρα πολλά και πολύ μικρά (π.χ. $8x8$) κομμάτια (κάθε ένα από τα οποία ανήκει σε ένα task), με συνέπεια να έχουμε μεγάλο overhead για την δημιουργία πολλών εργασιών οι οποίες τελικά είναι πάρα πολύ μικρές (και ως εκ τούτου δεν άξιζε η δημιουργία task στο task pool για αυτές).

Γενικά, τα μικρά block sizes επωφελούνται από την χρήση περισσότερων νημάτων καθώς σχετίζονται με την δημιουργία περισσότερων (και μικρότερων) εργασιών μέχρι να φθάσουμε στην βάση της αναδρομής (οι οποίες μπορούν να εκτελούνται παράλληλα από τα πολλαπλά νήματα). Αντιστρόφως, μεγάλα block sizes επωφελούνται από τη χρήση λιγότερων νημάτων καθώς δεν δημιουργείται αρκετός παραλληλισμός ώστε το όφελος της χρήσης περισσότερων νημάτων να υπερκεράζει το διαχειριστικό τους overhead.

Ωστόσο, για πολύ μικρά block sizes (π.χ. 16) η χρήση πολλών νημάτων οδηγεί σε χειρότερες επιδόσεις από την χρήση λιγότερων νημάτων, καθώς το μικρό block size ισοδυναμεί με την δημιουργία πολλών μικρών **απλών** εργασιών με τέτοιο τρόπο ώστε η χρήση υπέρμετρου πολυνηματισμού τελικά να μην συμφέρει λόγω της επικράτησης των overheads δημιουργίας αυτών των νημάτων, ανάθεσης (μικρών/**εύκολων**) εργασιών σε αυτά και βέβαια overheads ταυτόχρονης πρόσβασης στη μνήμη για το διάβασμα ενός πολύ μικρού αντικειμένου (κάτι που δημιουργεί μη παραγωγική και συμφέρουσα συμφόρηση στην μνήμη καθώς πολλά νήματα απευθύνονται στην μνήμη για να διαβάσουν και να επεξεργαστούν ένα πολύ μικρό αντικείμενο).

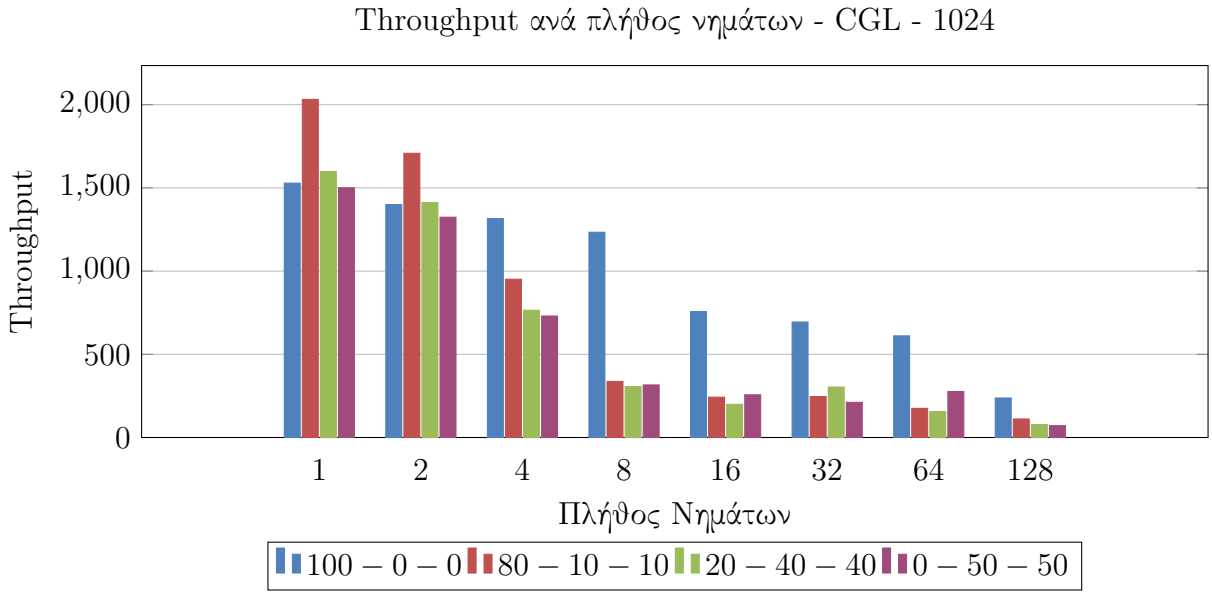
Οπότε, ενώ θα σκεφτόμασταν πως θα ήταν καλή ιδέα να χρησιμοποιήσουμε ένα κατά το δυνατόν μικρότερο block size ώστε να δημιουργήσουμε αρκετό παραλληλισμό από μικρές εργασίες και να μειώσουμε το μέρος του χρόνου εκτέλεσης που αφορά τον σειριακό τριπλό βρόχο (εκμεταλλευόμενοι αυτόν τον παραλληλισμό με πολλά νήματα και αποκτώντας καλύτερη κλιμακωσιμότητα), τελικά η χρήση πολύ μικρού block size οδηγεί στα προηγούμενα προβλήματα. Εν τέλει, η καλύτερη επίδοση επιτυγχάνεται για ένα μεσαίο μέγεθος block size (π.χ. 32 ή 64) και ένα επίσης ενδιάμεσο πλήθος νημάτων (8 ή 16). Σε κάθε περίπτωση βέβαια, όπως αναφέρθηκε, δεν θα πρέπει να έχουμε υψηλές προσδοκίες ως προς την κλιμακωσιμότητα που μπορεί να επιτευχθεί λόγω του εγγενώς μικρού βαθμού παραλληλισμού του αλγορίθμου.

Σε ό,τι αφορά το μέγεθος του γράφου, παρατηρούμε ότι οι μέγιστες τιμές Speedup που μπορούμε να επιτύχουμε αυξάνονται με την αύξηση του μεγέθους του γράφου καθώς για ίδιο block size και μεγαλύτερο γράφο, γίνονται περισσότερες αναδρομικές κλήσεις μέχρι να φθάσουμε στην βάση της αναδρομής και εκτελείται για περισσότερο χρόνο το παραλληλοποιήσιμο κομμάτι του προγράμματος.

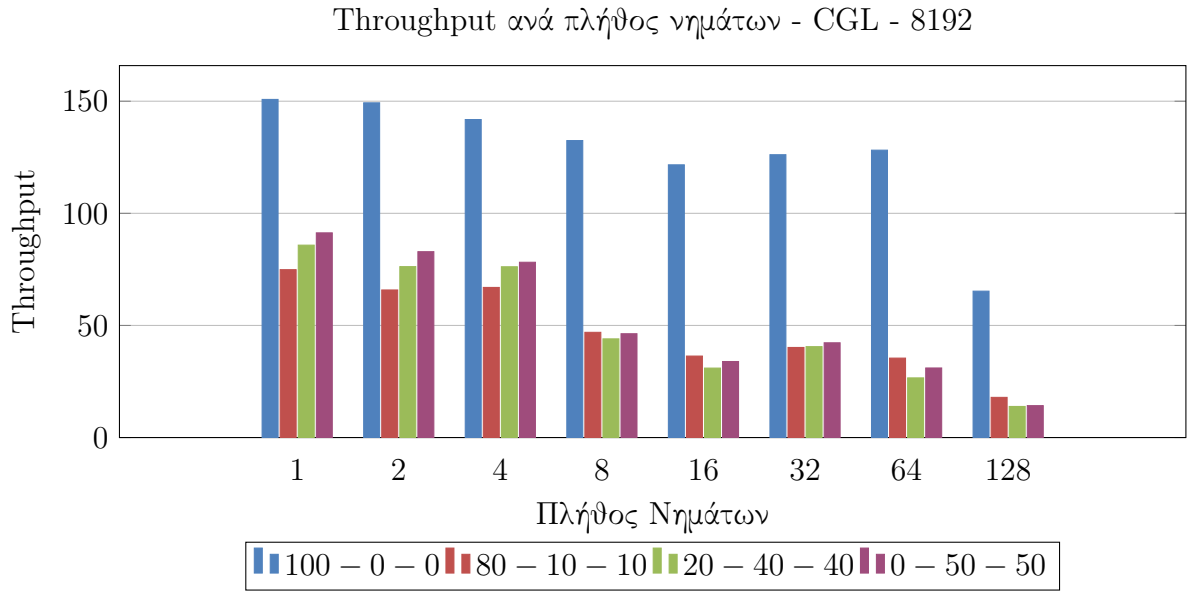
5 Ταυτόχρονες Δομές Δεδομένων

Σε αυτό το μέρος της εργαστηριακής άσκησης καλούμαστε να χρησιμοποιήσουμε διαφορετικούς τύπους κλειδώματος κατά ταυτόχρονη πρόσβαση σε μια συνδεδεμένη λίστα από πολλά νήματα.

Coarse Grain Lock Σε αυτή την περίπτωση, για ολόκληρη τη δομή της συνδεδεμένης λίστας χρησιμοποιείται ένα μοναδικό κοινό κλειδί, με συνέπεια οποιαδήποτε πρόσβαση (για οποιοδήποτε λόγο) σε οποιοδήποτε σημείο της συνδεδεμένης λίστας να γίνεται ατομικά από κάθε νήμα. Το προφανές μειονέκτημα αυτής της μεθόδου είναι ότι προκαλεί σημαντικές καθυστερήσεις, καθώς



(a) Throughput per number of threads for CGL lock and list length=1024



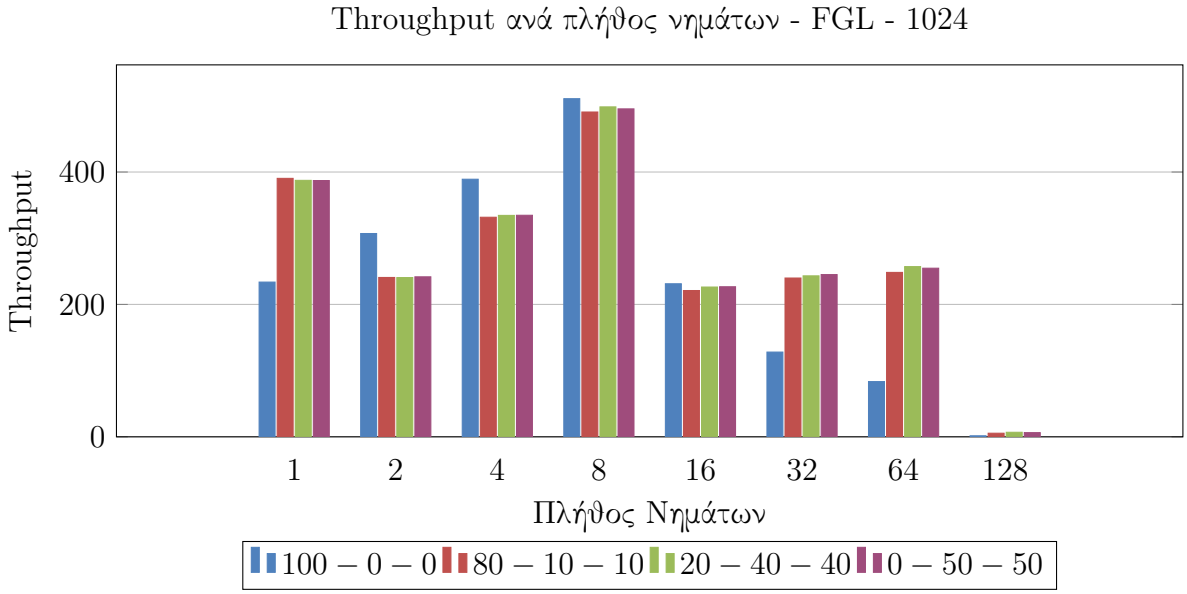
(b) Throughput per number of threads for CGL lock and list length=8192

Εικόνα 16: Throughput per number of threads and length list for CGL lock

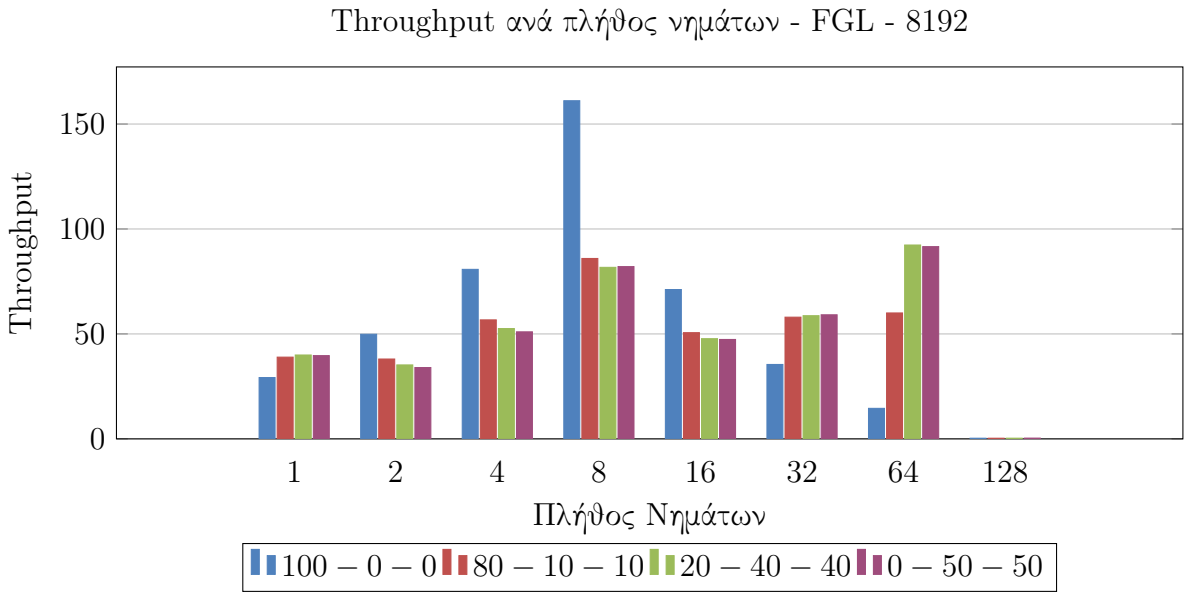
ακόμα και σε περιπτώσεις κατά τις οποίες δύο νήματα προσπελαύνουν την λίστα σε διαφορετικά σημεία (και επομένως δεν υπάρχει race condition μεταξύ τους), θα πρέπει να εκτελέσουν τις προσβάσεις τους σειριακά. Στα παραπάνω διαγράμματα της εικόνας 16, παρατηρούμε ότι σε κάθε περίπτωση το throughput γίνεται χειρότερο σε αντίστοιχα workloads με τη χρήση πολυνημάτισμού. Το γεγονός ότι όλες οι εργασίες πάνω στη λίστα θα γίνουν έτσι κι αλλιώς σειριακά έχει ως συνέπεια να μην προκύπτει βελτίωση με τη χρήση του πολυνηματισμού και μάλιστα οι επιδόσεις να γίνονται και χειρότερες λόγω του διαχειριστικού overhead των νημάτων και των locks. Δεδομένου ότι και τα τρία είδη εργασιών στην λίστα (contain, add, remove) χρησιμοποιούν lock και θα εκτελεστούν σειριακά από τα διάφορα νήματα (λόγω cgl), οι επι μέρους διαφορές στο throughput για διαφορετικά workloads οφείλονται στον απαιτούμενο χρόνο εκτέλεσης της κάθε εργασίας (π.χ. ένα add στην αρχή της λίστα θα είναι πολύ φθηνότερο από ένα add στο τέλος της λίστας λόγω της γραμμικής αναζήτησης του σημείου εισαγωγής). Να σημειωθεί επίσης ότι οι επιδόσεις όταν το μέγεθος της λίστας είναι μεγάλο (8192) γίνονται πολύ χειρότερες καθώς χρησιμοποιώντας ένα μεγάλο κλείδωμα για όλη τη λίστα, αποτρέπουμε τον παραλληλισμό που διαφορετικά θα ήταν ενδεχομένως εντονότερος στην μεγαλύτερη λίστα.

Fine Grain Lock Σε αυτή την περίπτωση, προσπαθούμε να επιτρεέψουμε σε διαφορετικά νήματα να προσπελαύνουν ταυτόχρονα τη λίστα όταν αυτές οι προσπελάσεις αφορούν διαφορετικά (επαρκώς μακρινά) σημεία της λίστας. Επομένως, ένα νήμα όταν εκτελεί κάποια εργασία σε κάποιο σημείο της λίστας αρκεί να κλειδώνει μόνο την γειτονιά του. Συγκεκριμένα, όταν εκτελείται διαγραφή ενός κόμβου, αρκεί το κλείδωμα του προς διαγραφή κόμβου (ώστε να μπλοκάρει το πεδίο του next και να μην χρησιμοποιηθεί για κάποια άλλη ταυτόχρονη διαγραφή που θα οδηγήσει σε αποτυχία της μιας εκ των δύο διαγραφών - βλέπε διαφάνειες μαθήματος) και του προηγούμενού του (καθώς το πεδίο του next είναι αυτό που πρέπει να μεταβληθεί για την διαγραφη). Όταν εκτελείται εισαγωγή κόμβου αντίστοιχα, κλειδώνεται ο προηγούμενος κόμβος του σημείο εισαγωγής (πρόκειται να μεταβληθεί το πεδίο του next) και ο επόμενος (ώστε η αναζήτηση του σημείου εισαγωγής να γίνει σωστά και να μην αλλοιωθεί από άλλες ταυτόχρονες εργασίες). Ομοίως για την contains που κάνει αναζήτηση. Βασικό συστατικό αυτού του κλειδώματος είναι η τακτική αναζήτησης στην λίστα μέσω του hand over hand locking. Κάθε νήμα που θέλει να προσπελάσει την λίστα, ξεκινά μια διαδικασία απόκτησης δυάδας κλειδιών από την αρχή της λιστάς έως ότου να φτάσει στο σημείο της λιστας όπου επιθυμεί να κάνει την εργασία. Ωστόσο, τα νήματα σειριοποιούνται κατά τη διάρκεια αυτής της διαδικασίας καθώς όλα ξεκινούν από την αρχή να λαμβάνουν κλειδιά διατρέχοντας τη λίστα. Ως εκ τούτου, νήματα που έπονται στην διαδικασία hand over hand locking οφείλουν να περιμένουν νήματα τα οποία προηγούνται. Για παράδειγμα, ένα νήμα που επιθυμεί να κάνει remove στη θέση 1000 θα πρέπει να περιμένει όλα τα νήματα που προσπελαύνουν προηγούμενες θέσεις της λίστας αν αυτά πρηγούνται στο hand over hand. Αυτό ωστόσο μπορεί να οδηγήσει σε σημαντικές καθυστερήσεις, ειδικά αν τα νήματα που καταλήγουν να περιμένουν έχουν να κάνουν μια πολύ σύντομη εργασία, η οποία τελικά καθυστερεί πολύ λόγω του hand over hand locking. Όπως φαίνεται στα επόμενα διαγράμματα της εικόνας 17, για μικρό αριθμό νημάτων παρατηρούμε κατά κανόνα βελτίωση του throughput μέχρι τα 8 νήματα για όλα τα workloads, ενώ για περισσότερα από 8 νήματα το throughput μειώνεται εξαιτίας του hand over hand locking μεταξύ πολλών νημάτων, το οποίο οδηγεί σε καθυστερήσεις οι οποίες γίνονται πιο έντονες όταν πολλά νήματα συνωστίζονται κατά τη δέσμευση των κλειδιών περιμένοντας άλλα νήματα. Μαλιστα, σχετικά με το workload $100 - 0 - 0$ που αποτελείται μόνο από εργασίες contains, παρατηρούμε ότι όσο αυξάνεται ο αριθμός των νημάτων γίνεται χειρότερο από τα υπόλοιπα workloads και οι επιπτώσεις του hand

over hand locking είναι εντονότερες σε αυτό. Αυτό οφείλεται στο γεγονός ότι η contains είναι μια απλή εργασία αναζήτησης που δεν τροποποιεί την λίστα (κάνει απλώς read), η οποία αναγκάζεται να περιμένει άλλες εργασίες contains (οι οποίες επίσης δεν τροποποιούν την λίστα). Δεδομένου ότι η contains κάνει μόνο read, μας επιτρέπει να την υλοποιήσουμε χωρίς locking επιτυγχάνοντας έτσι πολύ καλύτερες επιδόσεις (ωστόσο η υλοποίηση που μας δίνεται έτοιμη χρησιμοποιεί locking). Τέλος, συγκριτικά με την coarse grain υλοποίηση, επιτυγχάνονται χειρότερες επιδόσεις διότι αν και γλιτώσαμε το κλείδωμα όλης της δομής σε κάθε προσπέλαση, τελικά το hand over hand locking εισάγει σημαντικές καθυστερήσεις κυρίως λόγω του ότι ακόμη και μικρές εργασίες θα πρέπει ενδεχομένως να αναμένουν πολύ.



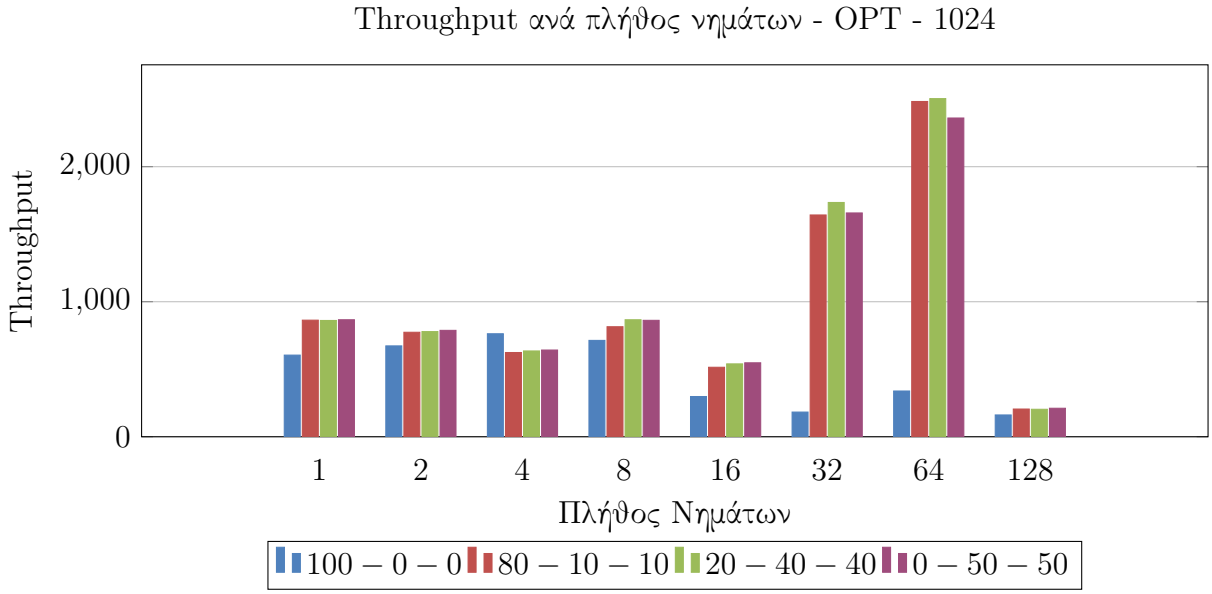
(a) Throughput per number of threads for FGL lock and list length=1024



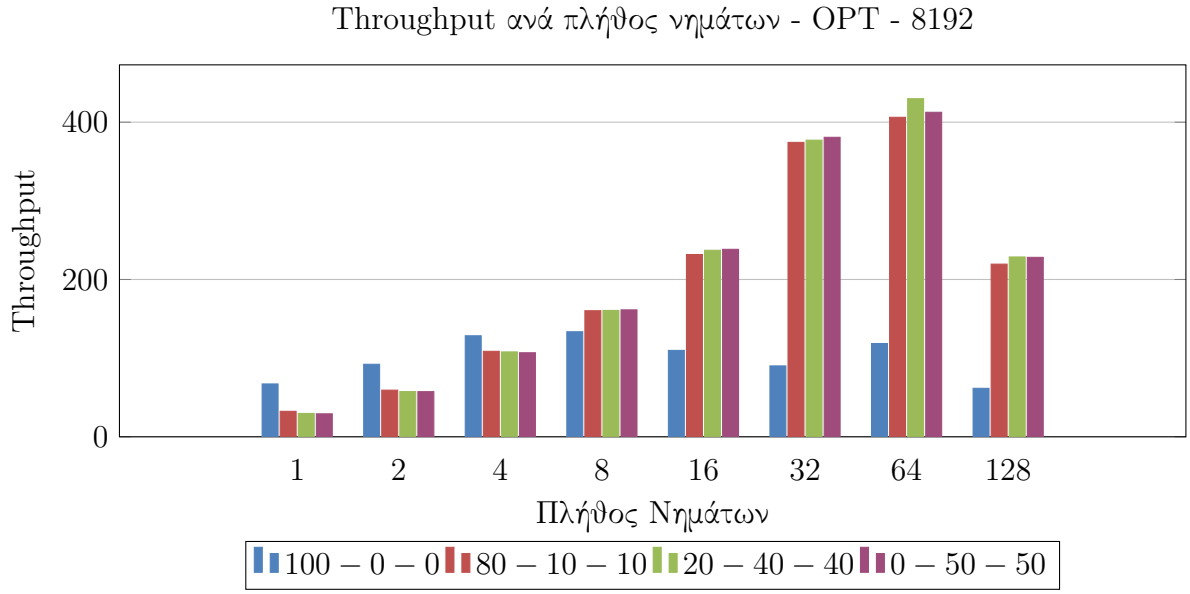
(b) Throughput per number of threads for FGL lock and list length=8192

Εικόνα 17: Throughput per number of threads and length list for FGL lock

Optimistic Synchronization Σε αυτή την υλοποίηση, η ιδέα βασιζεται στο γεγονός ότι δεν γίνονται τόσο συχνά conflicts μεταξύ νημάτων και για αυτό για την αναζήτηση των κρίσιμων κόμβων (π.χ. μεταξύ των οποίων θα εισαχθεί ένα νέο στοιχείο) δεν χρησιμοποιούνται κλειδώματα. Όταν βρεθούν αυτοί οι κόμβοι (χωρίς locks), στη συνέχεια κλειδώνονται και ελέγχεται αν εξακολουθούν να είναι στην κατάσταση στην οποία τους βρήκε το νήμα (δηλαδή ελέγχεται αν είναι προσπελάσιμοι από την αρχή της λίστας και ότι είναι διαδοχικοί). Αν αυτή η φάση ελέγου



(a) Throughput per number of threads for OPT lock and list length=1024



(b) Throughput per number of threads for OPT lock and list length=8192

Εικόνα 18: Throughput per number of threads and length list for OPT lock

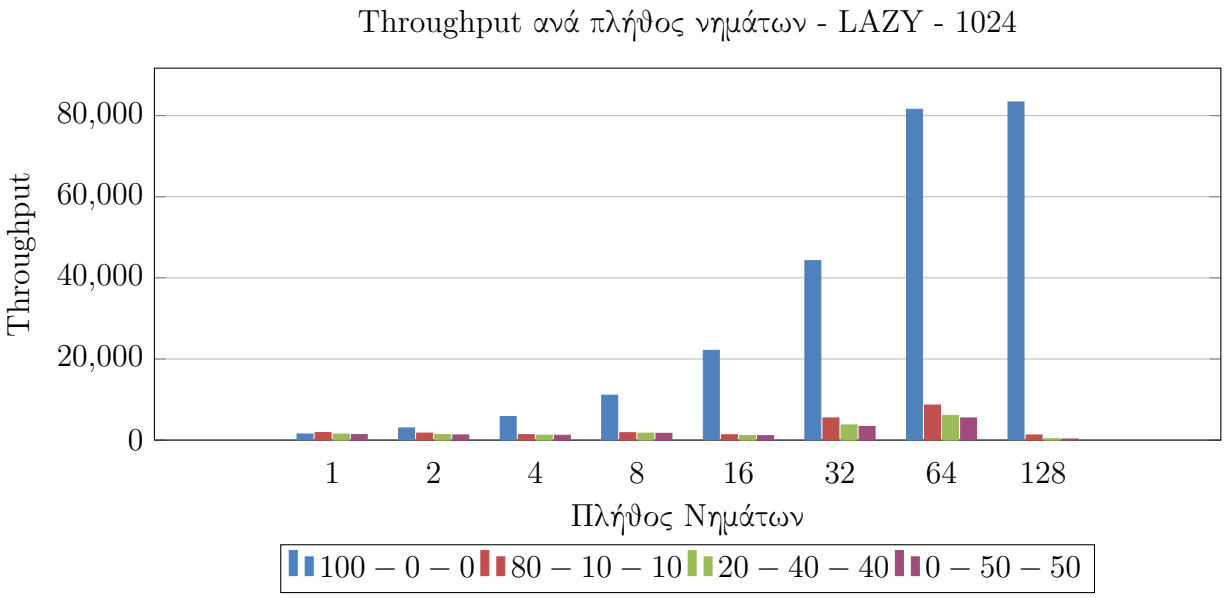
(validation) επιτύχει, τότε εκτελείται επιτυχώς η εργασίας του νήματος, διαφορετικά το νήμα ξανά προσπαθεί από την αρχή. Το βασικό μειονέκτημα αυτής της μεθόδου σε αυτήν την μορφή, είναι ο τρόπος με τον οποίο γίνεται ο έλεγχος προσπελασιμότητας των κόμβων στην validation

φάση. Σε αυτή τη φάση, η λίστα διατρέχεται πάλι από την αρχή, ώστε να επιβεβαιωθεί ότι οι κόμβοι είναι προσπελάσιμοι (και άρα εξακολυθούν να βρίσκονται στην λίστα). Επομένως, για κάθε εργασία χρειάζονται δύο προσπελάσεις της λίστα (μια αρχικά για την αναζήτηση των επιθυμητών κόμβων και μια στη συνέχεια για το validation). Αυτό καθιστά την μέθοδο προβληματική. Από τα διαγράμματα της εικόνας 18, παρατηρούμε ότι ειδικά για μεγάλο μέγεθος λίστας, η υλοποίηση αυτή προσφέρει καλή κλιμακωσιμότητα (μέχρι και τα 64 νήματα). Αφενός στην περίπτωση της λίστας των 1024, είμαστε τυχεροί και δεν εμφανίζονται πολλά conflicts μεταξύ των νημάτων (μάλιστα οι επιδόσεις είναι αρκετά καλές για 32 και 64 νήματα για τα περισσότερα workloads), αφετέρου για τη λίστα των 8192, λόγω του μεγάλου μεγέθους της πράγματι δεν είναι πολύ πιθανό δύο νήματα ταυτόχρονα να προσπελάσουν γειτονικές θέσεις. Εξαίρεση αποτελούν τα 128 νήματα στα οποία η επίδοση επιδεινώνεται καθώς τα νήματα είναι αρκετά ώστε να δημιουργούν conflicts. Το workload που δεν βελτιώνεται σχεδόν καθόλου είναι το $100 - 0 - 0$, όπου εκτελούνται μόνο contains (γεγονός που σημαίνει ότι η φάση validation θα είναι πάντα επιτυχής αφού κανένα νήμα δεν τροποποιεί την δομή). Η χρήση κλειδώματος και μετέπειτα η φάση του validation (που έχει προβλέψιμο αποτέλεσμα όπως εξηγήθηκε) αποτελούν δυσανάλογου μεγέθους overheads συγκριτικά με την δουλειά που η contains θέλει να κάνει. Τέλος, παρά τα μειονεκτήματα της μεθόδου, είναι σαφώς καλύτερη από την FGL καθώς έχει απαλλαγεί από την ανάγκη των κλειδωμάτων κατά την αναζήτηση.

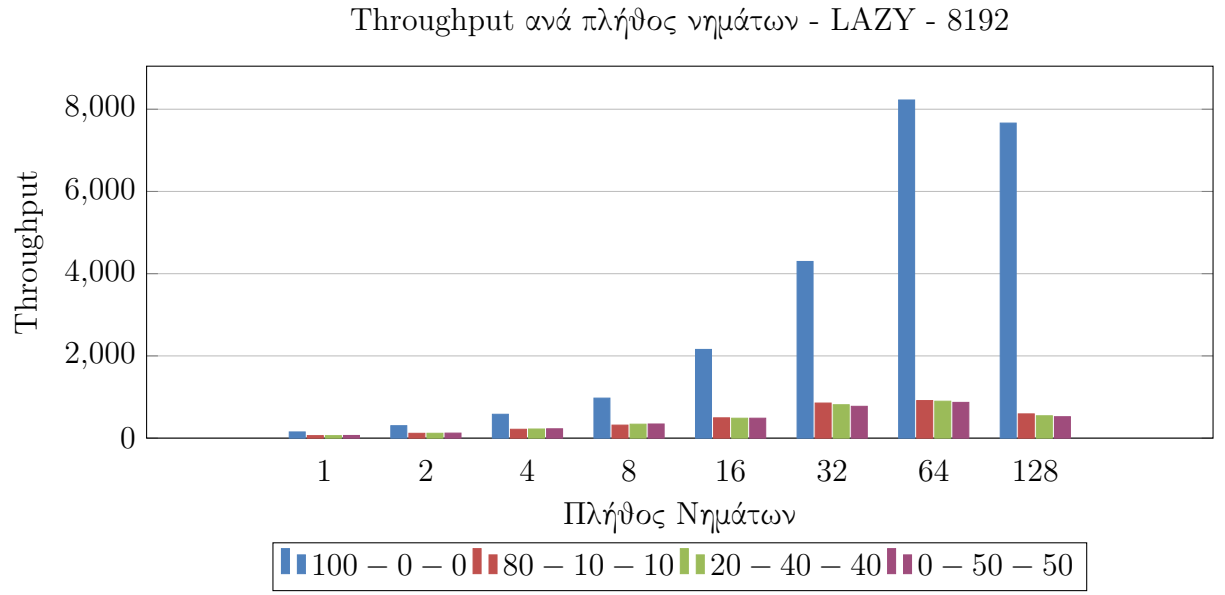
Lazy synchronization Σε αυτήν την υλοποίηση προσπαθούμε να απαλλαγούμε από την ανάγκη να διατρέξουμε για δεύτερη φορά τη λίστα κατά την validation φάση. Διατηρούμε ένα επιπλέον bit που δηλώνει για κάθε κόμβο αν βρίσκεται ή όχι στη λίστα και έτσι η validation φάση πλέον ελέγχει απλά αυτό το bit για τους κόμβους που την ενδιαφέρουν. Κατά τη διαγραφή ενός κόμβου, αυτός πρώτα διαγράφεται λογικά (με αλλαγή του bit) και έπειτα φυσικά (με αλλαγή των δεικτών). Από τα διαγράμματα της εικόνας 19, παρατηρούμε ότι αν εξαιρέσουμε το πρώτο workload $(100 - 0 - 0)$ η κλιμακωσιμότητα για τα υπόλοιπα workloads έχει τα ίδια χαρακτηριστικά. Αυτό οφείλεται βέβαια στο γεγονός ότι η μέθοδος αυτή διατηρεί όλα τα καλά της OPT μεθόδου (δηλαδή την αποφυγή συνεχών κλειδωμάτων κατά την αναζήτηση) εισάγοντας μια επιπλέον βελτιστοποίηση, αυτήν του αποδοτικού validation. Το validation πλέον από ένα πρόβλημα O(N) ανάγεται σε πρόβλημα O(1) (αν N το μέγεθος της λίστας). Ως εκ τούτου, κατά απόλυτες τιμές το throughput που επιτυγχάνεται είναι πολύ καλύτερο. Η διαφοροποίηση στο workload $100 - 0 - 0$ συγκριτικά με την OPT μέθοδο οφείλεται στο γεγονός ότι αποφεύγεται το validation στην contains (το οποίο είναι πάντα επιτυχές έτσι κι αλλιώς στο workload $100-0-0$). Επιπλέον, η contains σε αυτή την υλοποίηση (εκτός του ότι δεν κάνει validation) δεν χρησιμοποιεί καθόλου κλειδιά. Μόλις βρει το στοιχείο που αναζητά, δηλώνει επιτυχία χωρίς καμία επιπρόσθετη διαδικασία. Κατά συνέπεια, πλέον επιτυγχάνει πολύ καλές επιδόσεις.

Non-Blocking Synchronization Σε αυτή τη μέθοδο, προσπαθούμε να απαλλαγούμε πλήρως από την χρήση κλειδωμάτων μέσω της χρήσης ατομικών εντολών επιπέδου hardware. Συνοπτικά, οι υλοποιήσεις των τριών μεθόδων (add, remove, contains) έχουν ως εξής :

- **remove** : Γίνεται αναζήτηση του αντίστοιχου κόμβου χωρίς κλειδώματα και όταν βρεθεί, διαγράφεται λογικά (αλλαγή του marker με ατομική εντολή) και γίνεται μια προσπάθεια να διαγραφεί και φυσικά (επίσης μέσω ατομικής εντολής). Το γεγονός αυτό σημαίνει ότι από κάποιες remove που θα αποτύχουν στην φυσική διαγραφή θα δημιουργηθούν σκουπίδια (λογικά διαγεγραμμένοι κόμβοι που δεν είναι φυσικά διαγεγραμμένοι και παραμένουν



(a) Throughput per number of threads for LAZY lock and list length=1024

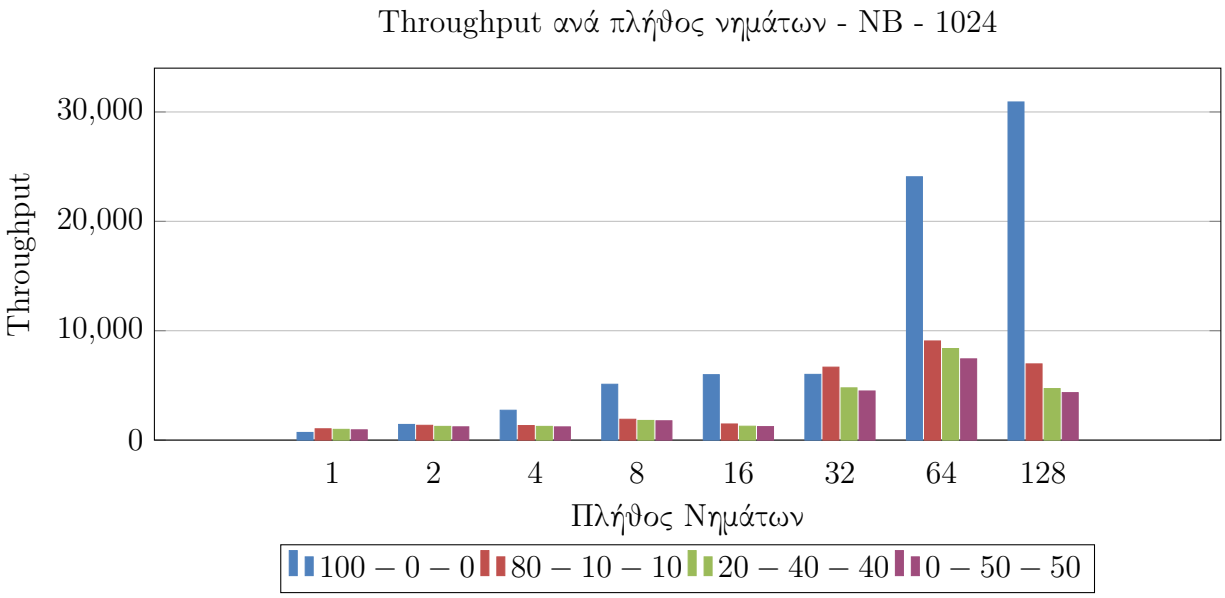


(b) Throughput per number of threads for LAZY lock and list length=8192

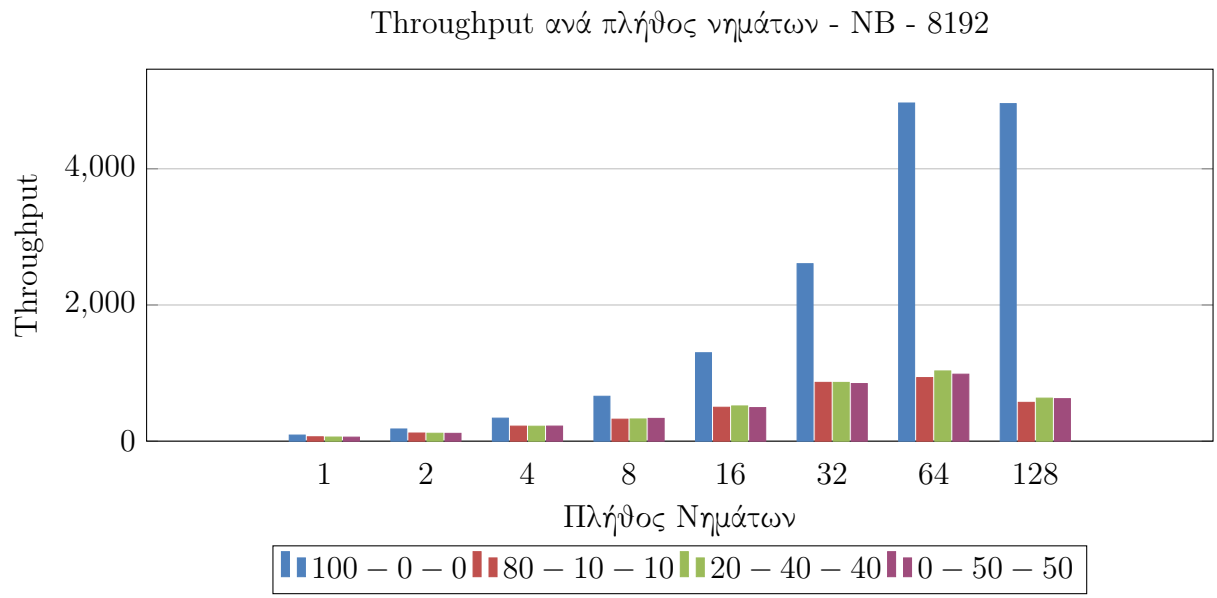
Εικόνα 19: Throughput per number of threads and length list for LAZY lock

προσπελάσιμοι). Αυτά τα σκουπίδια θα αναλαμβάνουν να μαζέψουν όσα νήματα κάνουν add και remove (ως δευτερέον έργο τους).

- **add** : Αφού βρεθεί η περιοχή εισαγωγής νέου στοιχείου, αυτό εισάγεται με κατάλληλη αλλαγή των δεικτών (η οποία γίνεται με ατομική εντολή και πρέπει να επιτύχει για την επιτυχία της add).
- **contains** : Διατρέχει απλώς τη λίστα (χωρίς κλειδώματα) και όταν βρει το επιθυμητό στοιχείο ελέγχει τον marker του. Ήδη φαίνεται το πόσο αποδοτική και απλή είναι αυτή η διαδικασία.



(a) Throughput per number of threads for NB lock and list length=1024



(b) Throughput per number of threads for NB lock and list length=8192

Εικόνα 20: Throughput per number of threads and length list for NB lock

Τα βασικά μειονεκτήματα της μεθόδου είναι τα παρακάτω :

- τα νήματα που κάνουν add και remove επιφορτίζονται με το σκούπισμα άχρηστων κόμβων το οποίο αν αποτύχει (κατά την φυσική διαγραφή για παράδειγμα, ο εν λόγω κόμβος προλαβαίνει να διαγραφεί από άλλο νήμα) οδηγεί στα εν λόγω νήματα να προσπαθήσουν πάλι από την αρχή (δηλαδή αν αποτύχουν στο δευτερεύον έργο τους που είναι το σκούπισμα αναγκάζονται να αρχίσουν από την αρχή)

- το γεγονός ότι επιτρέπεται να αφήνονται σκουπίδια από όσους κάνουν remove αλλά αποτυγχάνουν στη φυσική διαγραφή έχει ως συνέπεια να μένουν άχρηστα αλλά προσπελάσιμα

στοιχεία στην λίστα καθιστώντας την αναζήτηση σε αυτήν πιο χρονοβόρα.

Όπως φαίνεται στα διαγράμματα της εικόνας 23, επιτυγχάνεται καλή κλιμακωσιμότητα για όλα τα workloads (με εξαίρεση τα 128 νήματα που δημιουργούν αρκετό συνωστισμό και πιθανές αποτυχίες σε φυσικές διαγραφές). Οι επιδόσεις είναι παρόμοιες με αυτές του lazy synchronization (διότι αν και απαλλαχθήκαμε από τα locks, η διαχείριση των σκουπιδιών είναι ακριβή), με εξαίρεση το πρώτο workload $(100 - 0 - 0)$, στο οποίο ενώ έχουμε πολύ καλή κλιμακωσιμότητα το throughput είναι αρκετά μικρότερο συγκριτικά με την lazy synchronization υλοποίηση. Αυτό οφείλεται στο γεγονός ότι ενώ η υλοποίηση της contains στο LAZY και NB synchronization είναι παρόμοια, στην δεύτερη περίπτωση η διάσχιση της λίστας συναντάει αρκετά σκουπίδια σπαταλώντας ανούσια χρόνο.

Σημείωση Ως προς τη χρήση της μεταβλητής περιβάλλοντος *MT_CONF* που απαιτείται για την ανάθεση νημάτων σε λογικούς πυρήνες κατά την εκτέλεση των παραπάνω στον sandman, σημειώνουμε τα εξής : Ο sandman διαθέτει 32 φυσικούς πυρήνες (αρίθμηση 0 έως 31), καθένας εκ των οποίων διαθέτει 2 λογικούς πυρήνες (άρα συνολικά υπάρχουν 64 λογικοί πυρήνες), υποστηρίζοντας έτσι hyperthreading. Η αρίθμηση των λογικών πυρήνων είναι τέτοια ώστε οι λογικοί πυρήνες 0 και 32 να ανήκουν στον φυσικό πυρήνα 0, οι λογικοί πυρήνες 1 και 33 στον φυσικό πυρήνα 1 κ.ο.κ. Συνεπώς, για να χρησιμοποιήσουμε έως και N=32 νήματα για την εκτέλεση των παραπάνω, αρκεί να αναθέσουμε ένα νήμα σε κάθε φυσικό πυρήνα (αξιοποιώντας τον έναν από τους δύο λογικούς πυρήνες του), δηλαδή *MT_CONF=0,1,...,N-1*. Για N=64 νήματα, θα αναθέσουμε 2 νήματα ανά φυσικό πυρήνα, χρησιμοποιώντας και τα 2 hyperthreads του, δηλαδή *MT_CONF=0,1,...,31,32,33,...,63* (και οι 2 λογικοί πυρήνες κάθε φυσικού πυρήνα αναλαμβάνουν από ένα νήμα). Για N=128 νήματα, θα αναθέσουμε 4 νήματα ανά φυσικό πυρήνα και επομένως 2 νήματα ανά λογικό πυρήνα κάθε φυσικού πυρήνα, δηλαδή *MT_CONF=0,1,2,...,63,0,1,2,...,63*.

ΜΕΡΟΣ Γ

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

6 Παραλληλοποίηση και βελτιστοποίηση του K-means σε GPU

Εισαγωγή

Σε αυτό το μέρος της εργαστηριακής άσκησης, καλούμαστε να παραλληλοποιήσουμε τον αλγόριθμο K-means χρησιμοποιώντας την CUDA GPU του μηχανήματος silver1 της συστοιχίας serial. Πρόκειται να υλοποιήσουμε τρεις διαφορετικές εκδοχές παραλληλοποίησης (naive, transpose, shared), οι οποίες εξηγούνται στη συνέχεια. Αρχικά, για κάθε υλοποίηση παρουσιάζονται και εξηγούνται οι τροποποιήσεις μας και στο τέλος συγκεντρωτικά παρουσιάζονται τα διαγράμματα και οι συνολικές παρατηρήσεις-συγκρίσεις.

6.1 Υλοποίηση Naive - Κώδικας

Σε αυτή την υλοποίηση, σκοπός μας είναι η ανάθεση στην GPU του μέρους εκείνου του αλγορίθμου κατά το οποίο για κάθε object υπολογίζεται το κοντινότερο cluster (δηλαδή τελικά ο υπολογισμός του πίνακα membership που αποθηκεύει το index του κοντινότερου cluster για κάθε object και δευτερευόντος της μεταβλητής delta για τη σύγκλιση του αλγορίθμου). Συγκεκριμένα, στο πρόγραμμα (που παρατίθεται παρακάτω), μεταφέρουμε (δεσμεύοντας κατάλληλα μνήμη στη GPU) τα δεδομένα πάνω στα οποία θα δουλέψει η GPU, ορίζουμε τις συναρτήσεις εύρεσης ευκλείδιας απόστασης object από cluster και εύρεσης του κοντινότερου cluster και τελικά μεταφέρουμε τα αποτελέσματα πίσω στην CPU. Παρακάτω παρουσιάζεται ο κώδικας ανά συνάρτηση με τις αντίστοιχες επεξηγήσεις.Ο συνολικός κώδικας δίνεται με Dropbox link στο τέλος.

kmeans_gpu Για αυτή τη συνάρτηση (που ουσιαστικά είναι η main του προγράμματος) παρατίθενται τα βασικά σημεία που τροποποιήσαμε (συνολικά ο κώδικας στο Dropbox).
Αρχικά, ορίζουμε το μέγεθος του thread block, του grid και της shared memory όπως παρακάτω. Κάθε thread block έχει εν γένει μέγεθος ίσο με το block size που ρυθμίζεται στο run_on_queue.sh. Στην υλοποίησή μας, κάθε νήμα αναλαμβάνει τον υπολογισμό κοντινότερου cluster για ένα object. Άρα, συνολικά θέλουμε συνολικά numObjs νήματα, ομαδοποιημένα σε blocks των numThreadsPerClusterBlock νημάτων. Συνεπώς χρειαζόμαστε blocks πλήθους ίσου με το άνω όριο της διαίρεσης numObjs με το numThreadsPerClusterBlock. Το οποίο διαφορετικά υλοποιείται όπως φαίνεται παρακάτω (έτσι, αν έχουμε 128 νήματα ανά block αλλά το συνολικό πλήθος αντικειμένων είναι 250, θα δημιουργηθούν 2 blocks, με το δεύτερο να περιλαμβάνει 6 μη χρήσιμα νήματα). Ωστόσο, με αυτό τον τρόπο, δημιουργούνται επιπλέον νήματα από αυτά που χρειαζόμαστε (κάτι που θα πρέπει να ληφθεί υπόψη στην find_nearest_cluster παρακάτω), χωρίς όμως ιδιαίτερο κόστος, καθώς στη GPU η δημιουργία νημάτων γίνεται σχεδόν με μηδενικό κόστος σε επίπεδο υλικού. Τέλος, δημιουργούμε επαρκή χώρο στη shared memory για την αποθήκευση των partial deltas (ίσος με μηδέν στην περίπτωση της υλοποίησης χωρίς reduction).

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

```
3     const unsigned int clusterBlockSharedDataSize = sizeof(double)*
          numThreadsPerClusterBlock; // added partial_deltas into shared mem
4   // const unsigned int clusterBlockSharedDataSize = 0;
```

Στη συνέχεια, πρέπει να γίνουν οι κατάλληλες μεταφορές δεδομένων και η ανάθεση εργασίας στην GPU από την CPU. Συγκεκριμένα, αρχικά μεταφέρεται από την CPU στην GPU ο πίνακας clusters (με τα κέντρα των clusters) - ενώ προηγουμένως στον κώδικα έχουν ήδη μεταφερθεί οι πίνακες με τα objects και ο πίνακας membership και έχει δεσμευθεί ο κατάλληλος χώρος στην μνήμη της GPU για όλα αυτά. Έπειτα, καλείται ο gpu kernel (η find_nearest_cluster), τοποθετείται ένας barrier για να εξασφαλιστεί η ολοκλήρωση της δουλειάς της GPU και ο πίνακας membership όπως και το delta μεταφέρονται πίσω στην CPU, ώστε αυτή στη συνέχεια να υπολογίσει τα νέα κέντρα των clusters.

```
1                 /* GPU part: calculate new memberships */
2
3         /* TODO: Copy clusters to deviceClusters*/
4         checkCuda(cudaMemcpy(deviceClusters,clusters,numClusters*numCoords
              *sizeof(double),cudaMemcpyHostToDevice));
5
6         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
7
8                 //printf("Launching find_nearest_cluster Kernel with
                      grid_size = %d, block_size = %d, shared_mem = %d KB\n",
                       numClusterBlocks, numThreadsPerClusterBlock,
                      clusterBlockSharedDataSize/1000);
9         find_nearest_cluster
10            <<< numClusterBlocks, numThreadsPerClusterBlock,
                  clusterBlockSharedDataSize >>>
11            (numCoords, numObjs, numClusters,
12             deviceObjects, deviceClusters, deviceMembership,
                  dev_delta_ptr);
13
14        cudaDeviceSynchronize(); checkLastCudaError();
15                //printf("Kernels complete for itter %d, updating data in
                      CPU\n", loop);
16
17                /* TODO: Copy deviceMembership to membership*/
18        checkCuda(cudaMemcpy(membership,deviceMembership,numObjs*sizeof(
              int),cudaMemcpyDeviceToHost));
19
20        /* TODO: Copy dev_delta_ptr to &delta*/
21        checkCuda(cudaMemcpy(&delta,dev_delta_ptr,sizeof(double),
              cudaMemcpyDeviceToHost));
```

get_tid() Αρχικά, ορίζεται η βοηθητική συνάρτηση με την οποία υπολογίζεται το global id ενός νήματος στο μονοδιάστατο grid.

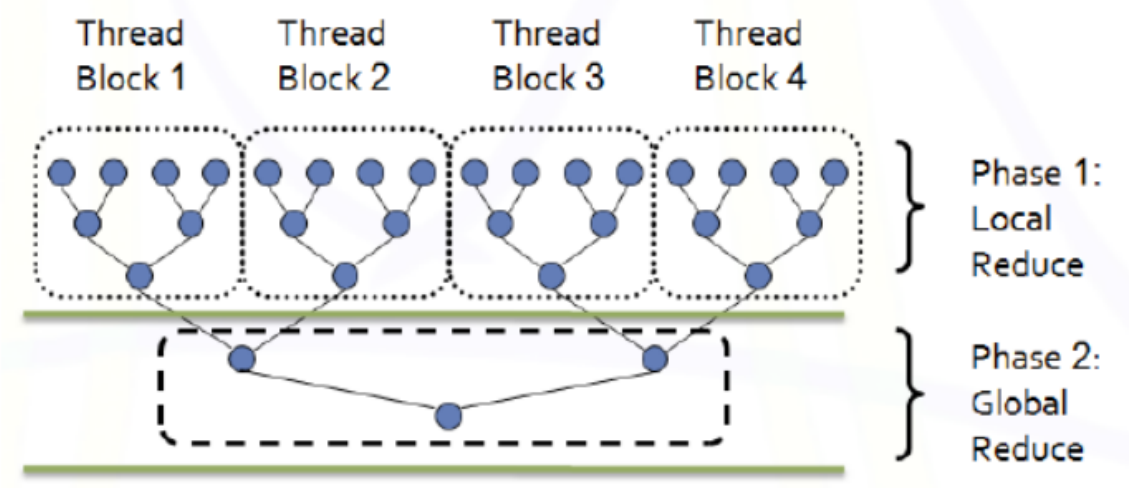
```
1 __device__ int get_tid(){
2     return blockDim.x*blockIdx.x+threadIdx.x; /* TODO: Calculate 1-Dim
          global ID of a thread */
3 }
```

euclid_dist_2 Στη συνέχεια, ορίζεται η *euclid_dist_2* μέσω της οποίας υπολογίζεται η ευκλείδια απόσταση (χωρίς τη ρίζα) του objectId από το κέντρο του clusterId έχοντας τα coordinates τους.

```
1  /* square of Euclid distance between two multi-dimensional points */
2  __host__ __device__ inline static
3  double euclid_dist_2(int    numCoords,
4                       int    numObjs,
5                       int    numClusters,
6                       double *objects,     // [numObjs][numCoords]
7                       double *clusters,    // [numClusters][numCoords]
8                       int    objectId,
9                       int    clusterId)
10 {
11     int i;
12     double ans=0.0;
13     double partial_ans=0.0;
14     /* TODO: Calculate the euclid_dist of elem=objectId of objects from
          elem=clusterId from clusters*/
15     for(i=0 ; i<numCoords ; i++){
16     partial_ans=(objects[objectId*numCoords+i]-clusters[clusterId*
          numCoords+i]);
17     ans+=partial_ans*partial_ans;
18     }
19     return(ans);
20 }
```

Σημαντική εδώ είναι η προσπέλαση των πινάκων objects και clusters στους οποίους οι δύο διαστάσεις (object/cluster και coordinate) έχουν αναχθεί σε μια, οπότε το i-οστό coordinate του objectId αντιστοιχεί στο στοιχείο $(objectId \times numCoords + i)$ και το i-οστό coordinate του clusterId αντιστοιχεί στο στοιχείο $(clusterId \times numCoords + i)$

find_nearest_cluster Η συνάρτηση αυτή βρίσκει για κάθε object το κοντινότερο cluster (και τελικά επιστρέφει τα συνολικά αποτελέσματα μέσω του πίνακα membership). Ουσιαστικά, υπολογίζεται η ευκλείδια απόσταση (όπως ορίστηκε παραπάνω) από κάθε cluster και βρίσκεται η ελάχιστη. Επειδή στην υλοποίησή μας θεωρούμε ότι κάθε νήμα λαμβάνει την εύρεση του κοντινότερου cluster για ένα object και επειδή λόγω του padding (που εξηγήθηκε παραπάνω) δημιουργούνται επιπλέον αδρανή νήματα (ώστε να γεμίσει το αντίστοιχο thread block), θα πρέπει να ληφθεί υπόψη ότι σε αυτά τα αδρανή νήματα δεν θα ανατεθεί κάποια εργασία. Το κύριο μέρος της συνάρτησης εκτελείται από νήματα με id μικρότερο του πλήθους των αντικειμένων (διότι αφού ένα νήμα αναλαμβάνει ένα object, τα χρήσιμα νήματα είναι τόσα όσο και το πλήθος των objects). Στη συνέχεια, καλούμε κατάλληλα την συνάρτηση ευκλείδιας απόστασης ώστε να ευρεθεί η ελάχιστη. Το τελευταίο σημαντικό συστατικό της υλοποίησης είναι ο υπολογισμός του delta, το οποιο είναι κοινή μεταβλητή για όλα τα νήματα και θα πρέπει να προσπελαύνεται ατομικά. Μια πρώτη σκέψη είναι η ενημέρωσή του ατομικά από κάθε νήμα μέσω της atomicAdd (όπως παρατίθεται σε σχόλιο στη γραμμή 37 του κώδικα). Ωστόσο αυτό δημιουργεί ένα σημαντικό bottleneck στην εκτέλεση του κώδικά, καθώς οι προσπελάσεις στην delta γίνονται σεριακά από όλα τα νήματα (αυτό μπορεί να οδηγήσει σε πολύ κακές επιδόσεις όπως θα φανεί παρακάτω). Η καλύτερη υλοποίηση περιλαμβάνει ένα δενδρικού τύπου reduction στην μεταβλητή delta. Όπως φαίνεται και στο παράδειγμα της εικόνας 21 αρχικά κάθε νήμα συναθροίζει το delta του με αυτό του διπλανού του νήματος (συναθροίσεις ανά δύο), οι επιμέρους



Εικόνα 21: Δενδρικό Reduction για τη μεταβλητή delta

συναθροίσεις αθροίζονται πάλι ανά δύο κ.ο.κ έως ότου ένα νήμα (το 0 κάθε block) να έχει το συνολικό delta από όλα τα threads του block. Η διαφοροποίηση έγκειται στο global reduce, όπου σε αντίθεση με την εικόνα, στην υλοποίησή μας το νήμα 0 κάθε thread block προσθέτει ατομικά το partial αποτελέσμά του στην συνολική μεταβλητή delta (και δεν συνεχίζεται το δενδρικού τύπου reduction όπως στην εικόνα). Έτσι μόνο ένα νήμα-εκπρόσωπος από κάθε block θα πρέπει να σειριοποιηθεί μέσω της ατομικής πρόσθεσης στο delta. Για το σκοπό αυτό, ορίζεται ένας πίνακας partial_deltas ο οποίος αποθηκεύεται στη shared memory που αντιστοιχεί στο εκάστοτε thread block. Η διαφορά στην επίδοση μεταξύ των δύο υλοποιήσεων για το delta φαίνεται στην εικόνα 22.Ο κώδικας φαίνεται παρακάτω.

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

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

    /* TODO: Maybe something is missing here... should all threads run this? */
    if (tid<numObjs) { /*because of padding in the grid size*/
        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 */
```

```
22      min_dist=euclid_dist_2(numCoords,numObjs,numClusters, objects,
            deviceClusters,tid,index);
23          for (i=1; i<numClusters; i++) {
24              /* TODO: call dist = euclid_dist_2(...) with correct objectId/
                    clusterId */
25          dist=euclid_dist_2(numCoords,numObjs,numClusters, objects,
                deviceClusters,tid,i);
26              /* no need square root */
27              if (dist < min_dist) { /* find the min and its array index */
28                  min_dist = dist;
29                  index    = i;
30              }
31          }
32      extern __shared__ double partial_deltas[];
33      partial_deltas[threadIdx.x]=0.0;
34          if (deviceMembership[tid] != index) {
35              /* TODO: Maybe something is missing here... is this write safe
                    ? */
36             // (*devdelta)+= 1.0; (not safe - race condition)
37            // atomicAdd(devdelta, 1.0); and this is also bad
38           partial_deltas[threadIdx.x]+=1.0;
39          }
40
41          /* assign the deviceMembership to object objectId */
42          deviceMembership[tid] = index;
43      __syncthreads();
44      int j = blockDim.x/2;
45      while(j!=0){
46          if(threadIdx.x<j)partial_deltas[threadIdx.x]+=partial_deltas[
                threadIdx.x+j];
47          __syncthreads();
48          j /= 2;
49      }
50      if(threadIdx.x==0){
51          atomicAdd(devdelta,partial_deltas[0]);
52      }
53
54      }
55  }
```

Συνολικά, ολόκληρος ο κώδικας υπάρχει **εδώ** *(Dropbox link)*

6.2 Υλοποίηση Transpose - Κώδικας

Σε αυτή την υλοποίηση σκοπός μας είναι η αντιστροφή των πινάκων objects και clusters από row-oriented σε column-oriented και η προσπέλασή τους με αντίστοιχο τρόπο. Παρακάτω παρατίθενται τα τμήματα κώδικα που άλλαξαν σε σχέση με την naive υλοποίηση.

kmeans_gpu Στο τμήμα αυτό, ορίζουμε τους ανεστραμμένους πίνακες objects και clusters με χρήση της συνάρτησης calloc_2d. Φροντίζουμε τα πρώτα δύο ορίσματα της calloc να είναι αντίστροφα αυτή τη φορά (πρώτα τα coordinates και μετά τα objectId/clusterId). Στη συνέχεια, αντιγράφονται τα δεδομένα του πίνακα objects στο ανεστραμμένο dimObjects, όπως παρακάτω.

```
    /* TODO: Transpose dims */
    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]

        //  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];
                }
        }
```

Ορίζουμε τα μεγέθη των blocks, grid, shared memory ακριβώς όπως και στην naive περίπτωση:

```
        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 = sizeof(double)*
       numThreadsPerClusterBlock;
```

Στη συνέχεια, γίνονται οι μεταφορές δεδομένων και η κλήση του gpu kernel από την cpu. Προσέχουμε αυτή τη φορά να μεταφέρουμε τον πίνακα dimObjects και όχι το objects στην GPU. Μάλιστα, επειδή ο dimObjects είναι διδιάστατος, φροντίζουμε ως δεύτερο όρισμα της cudaMemcpy να προσθέσουμε το dimObjects[0] το οποίο ως εκ τούτου είναι ένας δείκτης στο πρώτο στοιχείο της πρώτης γραμμής του dimObjects. Η μεταφορά των δεδομένων από τη GPU πίσω στη CPU γίνονται ομοίως με την naive περίπτωση.

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

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

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

                //printf("Launching find_nearest_cluster Kernel with
                   grid_size = %d, block_size = %d, shared_mem = %d KB\n",
                    numClusterBlocks, numThreadsPerClusterBlock,
                   clusterBlockSharedDataSize/1000);
        find_nearest_cluster
            <<< numClusterBlocks, numThreadsPerClusterBlock,
               clusterBlockSharedDataSize >>>
            (numCoords, numObjs, numClusters,
             deviceObjects, deviceClusters, deviceMembership,
                dev_delta_ptr);

        cudaDeviceSynchronize(); checkLastCudaError();
                //printf("Kernels complete for itter %d, updating data in
                   CPU\n", loop);

```

```
17                  /* TODO: Copy deviceMembership to membership*/
18          checkCuda(cudaMemcpy(membership,deviceMembership,numObjs*sizeof(
                int),cudaMemcpyDeviceToHost));
19
20          /* TODO: Copy dev_delta_ptr to &delta*/
21          checkCuda(cudaMemcpy(&delta,dev_delta_ptr,sizeof(double),
                cudaMemcpyDeviceToHost));
```

Στη συνέχεια, η CPU υπολογίζει τα νέα κέντρα των clusters αποθηκεύοντά τα στον πίνακα dimClusters (ο οποίος έχει ανεστραμμένη column-based μορφή). Υπολογίζουμε τον clusters (που είναι row-based) αντιστρέφοντας όπως στη συνέχεια τον dimClusters.

```
1     /*TODO: Update clusters using dimClusters. Be carefull of layout!!!
          clusters[numClusters][numCoords] vs dimClusters[numCoords][
          numClusters] */
2
3          for (i=0;i<numClusters;i++){
4                  for(j=0;j<numCoords;j++){
5                          clusters[i*numCoords+j]=dimClusters[j][i];
6                  }
7          }
```

Η συνάρτηση get_tid δεν αλλάζει, ενώ στην find_nearest_cluster αλλάζει μόνο η κλήση της euclidean συνάρτησης η οποία έχει αλλάξει όνομα.

euclid_dist_2_transpose Η βασική ιδέα υλοποίησης αυτής της συνάρτησης δεν αλλάζει σε σχέση με την naive. Αλλάζει μόνο ο τρόπος προσπέλασης στους πίνακες objects, clusters οι οποίοι πλέον είναι column-based. Με λίγη καλή οπτικοποίηση της αναπαράστασης, γίνεται εύκολα αντιληπτός ο τρόπος προσπέλασης που φαίνεται παρακάτω.

```
1  /* square of Euclid distance between two multi-dimensional points using
       column-base format */
2  __host__ __device__ inline static
3  double euclid_dist_2_transpose(int numCoords, int    numObjs,
4                      int    numClusters,
5                      double *objects,     // [numCoords][numObjs]
6                      double *clusters,    // [numCoords][numClusters]
7                      int    objectId,
8                      int    clusterId)
9  {
10     int i;
11     double ans=0.0;
12     double partial_ans=0.0;
13         /* TODO: Calculate the euclid_dist of elem=objectId of objects
               from elem=clusterId from clusters, but for column-base format
               !!! */
14     for(i=0 ; i<numCoords ; i++){
15         partial_ans=(objects[i*numObjs+objectId]-clusters[i*numClusters+
               clusterId]);
16         ans+=partial_ans*partial_ans;
17     }
18     return(ans);
19 }
```

Συνολικά, ολόκληρος ο κώδικας υπάρχει **εδώ** *(Dropbox link)*

6.3 Υλοποίηση Shared - Κώδικας

Στην υλοποίηση αυτή, ο πίνακας clusters με τα κέντρα των clusters μεταφέρεται στην shared memory που αντιστοιχει σε κάθε thread block με σκοπό η προσπέλασή του να μη γίνεται από την device memory αλλά από την κοντινή σε κάθε νήμα ενός thread block shared memory (δεδομένου ότι κάθε νήμα για τον υπολογισμό του χρειάζεται να διαβάσει όλο τον πίνακα clusters και να βρει το κοντινότερο κέντρο). Η υλοποίηση αυτή θα είναι ίδια με την transpose, με τη μόνη διαφορά στον ορισμό του μεγέθους της shared μνήμης και στη διαχείρισή της στην find_nearest_cluster.

kmeans_gpu - shared mem Η shared memory πλέον δεν αποθηκεύει μόνο τα partial deltas όπως πριν αλλά και τον πίνακα clusters που περιέχει double στοιχεία πλήθους $numClusters \times numCoords$. Οπότε το μέγεθος της shared μνήμης γίνεται :

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

find_nearest_cluster Στη συνέχεια, παρατίθεται από την find_nearest_cluster μόνο το μέρος εκείνο που αλλάζει (χωρίς το delta reduction). Ουσιαστικά, η διαφορά εδώ είναι ότι πρέπει ο πίνακας clusters να φορτωθεί (με συνεργασία των νημάτων ενός thread block) στην shared memory, στον __shared__ πίνακα shmemClusters. Μια σκέψη θα ήταν η ανάθεση σε κάθε νήμα του thread block να φέρει με round robin τρόπο ένα cluster (με όλα τα coordinates του) στην shared μνήμη. Στην περίπτωσή μας, όπου τα clusters είναι 16 ενώ τα νηματα ανά block είναι τουλάχιστον 32 (μέχρι και 1024 μάλιστα), αυτό θα σήμαινε ότι ένα σημαντικό ποσοστό νημάτων ενός block παραμένει αδρανές και δε συνεισφέρει στην μεταφορά. Μια βελτίωση είναι η ανάθεση σε κάθε νήμα (επίσης κυκλικά με round robin τρόπο) ένα στοιχείο του πίνακα clusters (που αντιστοιχεί σε ένα coordinate ενός cluster). Στην περίπτωση των 16 clusters με 2 coordinates, αυτό θα αναθέσει σε 32 νήματα να μεταφέρουν από ένα στοιχείο (κάτι που στην περίπτωση block size=32 αξιοποιεί πλήρως όλα τα νήματα). Στον κώδικα παρακάτω, φαίνεται αυτή η υλοποίηση. Σε κάθε νήμα ανατίθεται κυκλικά η μεταφορά ενός στοιχείου του πίνακα. Αν ο πίνακας είναι μεγαλύτερου μεγέθους από το πλήθος νημάτων των block η ανάθεση ξαναναρχίζει από την αρχή (αυτό αναπαρίσταται από το stride κατά blockDim.x). Προφανώς, στη συνέχεια, η συνάρτηση euclid_dist_2_transpose χρησιμοποιεί ως όρισμα τον πίνακα shmemClusters και όχι τον deviceClusters, που είναι αποθηκευμένος στην device memory.

```
1    extern __shared__ double shmemClusters[];
2
3        /* TODO: Copy deviceClusters to shmemClusters so they can be
            accessed faster.
4                BEWARE: Make sure operations is complete before any thread
                    continues... */
5
6        /* Get the global ID of the thread. */
7    int tid = get_tid();
8
9    int ind;
10   for(ind=threadIdx.x ; ind<numClusters*numCoords ; ind+=blockDim.x){
11       shmemClusters[ind]=deviceClusters[ind];
12   }
13   __syncthreads();
```

```
14          /* TODO: Maybe something is missing here... should all threads run
                this? */
15      if (tid<numObjs) {
16          int   index, i;
17          double dist, min_dist;
18
19          /* find the cluster id that has min distance to object */
20          index = 0;
21          /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/
               clusterId using clusters in shmem*/
22          min_dist=euclid_dist_2_transpose(numCoords,numObjs,numClusters,
               objects, shmemClusters,tid,index);
23          for (i=1; i<numClusters; i++) {
24              /* TODO: call dist = euclid_dist_2(...) with correct objectId/
                   clusterId using clusters in shmem*/
25              dist=euclid_dist_2_transpose(numCoords,numObjs,numClusters,
                   objects, shmemClusters,tid,i);
26              /* no need square root */
27              if (dist < min_dist) { /* find the min and its array index */
28                  min_dist = dist;
29                  index    = i;
30              }
31          }
```

Συνολικά, ολόκληρος ο κώδικας υπάρχει **εδώ** *(Dropbox link)*

6.4 Διαγράμματα - Συγκρίσεις/Παρατηρήσεις

Τα συνολικά διαγράμματα χρόνου εκτέλεσης και Speedup φαίνονται στο τέλος, ενώ προηγουμένως παρατίθενται κάποια στοιχεία της GPU που χρησιμοποιούμε τα οποία λήφθηκαν με χρήση της συνάρτησης cudaGetDeviceProperties και παρατηρήσεις-συγκρίσεις για τις διαφορετικές μεθόδους.

naive Σε ό,τι αφορα την naive υλοποίηση, παρατηρούμε ότι με χρήση GPU ακόμη και αυτή η υλοποίηση επιτυγχάνει σχεδόν δέκα φορές καλύτερη επίδοση σε σχέση με την σειριακή υλοποίηση στην CPU. Προφανώς, η GPU που δρα ως accelerator καταφέρνει να παραλληλοποιήσει πολύ αποδοτικά τον Kmeans (και συγκεκριμένα τον υπολογισμό των κοντινότερων clusters, που αποτελεί μια διαδικασία με αρκετό παραλληλισμό). Με αύξηση του block size από 32 σε 64 παρατηρείται βελτίωση στην επίδοση (το speedup αυξάνεται κατά 1.3), ενώ με περαιτέρω αύξηση του block size η επίδοση είναι στάσιμη. Η GPU υποστηρίζει ένα συγκεκριμένο μέγιστο αριθμό active blocks per SM, έστω B. Με μικρό block size (π.χ. 32), θα υπάρχουν το πολύ $32 \times B$ threads ανά SM ενώ το μέγιστο πλήθος threads ανά SM είναι 2048, όπως φαίνεται παραπάνω. Το occupancy της GPU (ο λόγος $32 \times B/2048$ στο παράδειγμά μας) θα είναι μικρό, με αποτέλεσμα την υποχρησιμοποίηση της GPU. Με αύξηση του block size (64 αντί για 32), το occupancy αυξάνεται καθώς αποφεύγουμε την υποχρησιμοποίηση των πόρων. Ωστόσο, από ένα σημείο και μετά αυτή η βελτίωση διακόπτεται. Εδώ, αξίζει να σημειώσουμε ότι ο τρόπος με τον οποίο η GPU καταφέρνει να κρύψει το latency της μνήμης επιτυγχάνοντας καλές επιδόσεις, είναι η ταυτόχρονη ανάθεση εργασιών σε warps όσο άλλα warps εκτελούν εργασίες μνήμης.

Αυτό ωστόσο προϋποθέτει ότι υπάρχουν αρκετές πράξεις να γίνουν στα ήδη υπάρχοντα δεδομένα ώστε η GPU να απασχολείται με αυτές όσο ταυτόχρονα γίνεται κάποιο load από τη μνήμη. Ο K-means όμως είναι memory bound, δηλαδή γίνονται λίγες πράξεις σε πολλά δεδομένα και ως εκ τούτου τα loads από τη μνήμη (και η καθυστέρηση που ενέχουν) υπερκαλύπτουν το όφελος της ταυτόχρονης εκτέλεσης πράξεων από άλλα νήματα. Κατά συνέπεια, με μεγάλο block size ενώ υπάρχει η προοπτική αξιοποίησης μεγαλύτερου παραλληλισμού, τελικά οι προσβάσεις μνήμης (αν και πολύ γρήγορες στην GPU) εξακολουθούν να επιβάλουν το κόστος τους. Σημαντικό επίσης είναι να σημειώσουμε ότι στην συγκεκριμένη υλοποίηση όλες οι αναγνώσεις μνήμης γίνονται από την device memory καθώς δεν χρησιμοποιείται η shared memory του κάθε SM (στην οποία οι προσβάσεις είναι γρηγορότερες). Μάλιστα, δεν πρέπει να παραληφθεί το γεγονός ότι εξαιτίας του επαρκούς compute capability ($6.1 >= 2.0$), η παρούσα GPU διαθέτει cache, η χρησιμότητα της οποίας θα φανεί περισσότερο παρακάτω. Χωρίς αυτήν, οι επιδόσεις θα ήταν σημαντικά χειρότερες στην naive υλοποίηση. Η επίδοση, τέλος, θα μπορούσε να επηρεαστεί από warp execution divergence (με συνέπεια την υποχρησιμοποίηση των warps), αλλά αυτό δεν συναντάται στην περίπτωσή μας.

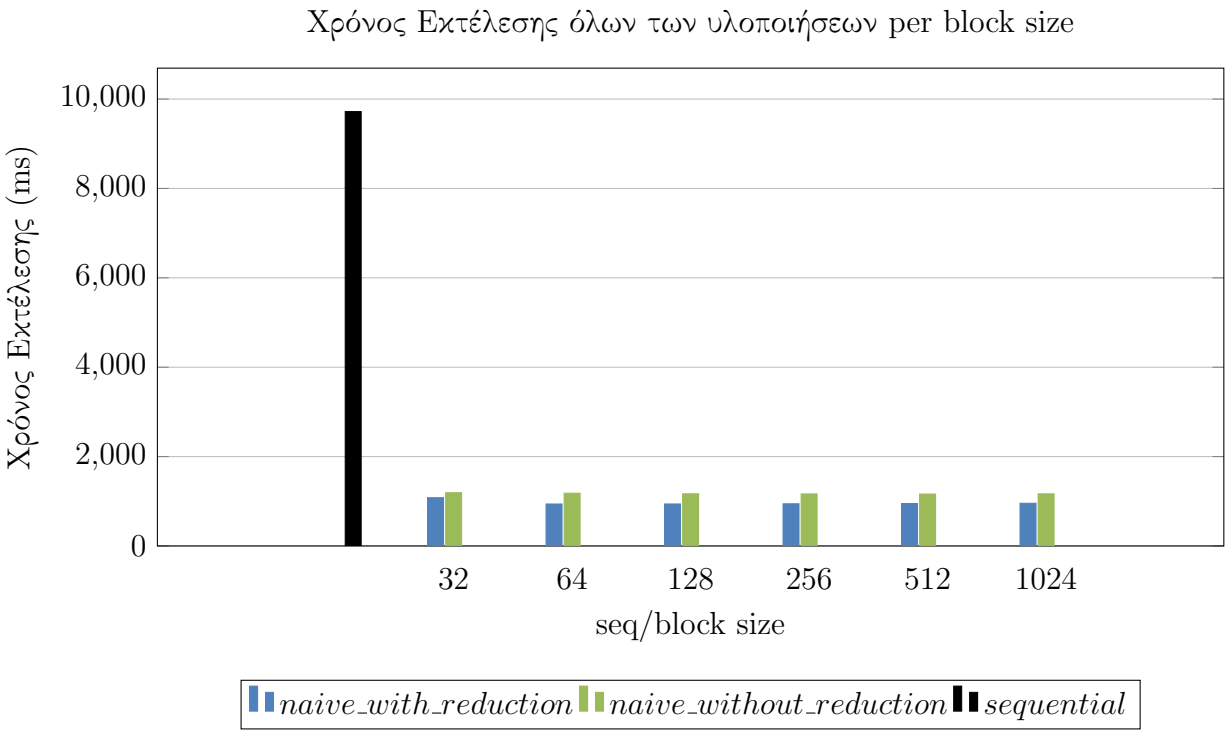
transpose Για την επεξήγηση του κέρδους σε επίδοση που μπορούμε να αποκτήσουμε αντιστρέφοντας τους πίνακες objects και clusters, αρχικά εξηγούμε τον τρόπο με τον οποίο γίνονται οι προσβάσεις στη μνήμη από τα νήματα. Οι προσβάσεις στην device memory από τα νήματα γίνονται ανά half-warp (ανά 16 νήματα). Κάθε ένα από αυτά τα νήματα, διαβάζει μια λέξη, την επόμενη στην μνήμη από αυτήν που διάβασε το προηγούμενο νήμα. Αν οι πίνακες objects και clusters δεν ήταν transposed, τότε στη μνήμη θα ήταν αποθηκευμένοι ως εξής (έστω για τα objects): πρώτα όλα τα coordinates κατά σειρά του object 0, δίπλα τους τα coordinates του object 1 κ.ο.κ. Κατά συνέπεια, το νήμα 0 θα διαάβαζε την πρώτη λέξη, δηλαδή το coordinate 0 του object 0, το νήμα 1 θα διάβαζε το coordinate 1 του object 0 κ.ο.κ. Ωστόσο, ενώ το νήμα 0 δουλεύει στο object 0 (και επομένως θα φέρει κάτι χρήσιμο για τον εαυτό του), το νήμα 1 δεν δουλεύει στο object 0 και δεν χρειάζεται για τον εαυτό του το coordinate 1. Στην περίπτωση που δεν είχαμε cache, αυτό θα σήμαινε ότι στο παρόν memory transaction το νήμα 1 έκανε μη χρήσιμη δουλειά και θα έπρεπε να ακολουθήσει νέο transaction ώστε να φέρει χρήσιμα για τον εαυτό του δεδομένα. Στην περίπτωσή μας, η cache έρχεται να σώσει αυτή τη λανθασμένη πρόσβαση στη μνήμη, καθώς μπορεί το νήμα να μην έφερε χρήσιμο δεδομένο για τον εαυτό του, αλλά έφερε χρήσιμο δεδομένο για το νήμα 0 και το έσωσε στην cache. Ως εκ τούτου, το νήμα 0 θα το βρει έτοιμο και δεν θα χρειαστεί νέο memory transaction για αυτό. Αν ο πίνακας objects (αντίστοιχα ο clusters) είναι transposed τότε στη μνήμη θα ήταν αποθηκευμένος ως εξής: πρώτα το coordinate 0 για όλα τα objects κατά σειρά, δίπλα τους το coordinate 1 για όλα τα objects κατά σειρά κ.ο.κ. Ως εκ τούτου, με ένα memory transaction, το νήμα 0 θα φέρει το coordinate 0 του object 0 και το νήμα 1 το coordinate 0 του object 1. Και τα δύο νήματα θα διαβάσουν κάτι χρήσιμο για τον εαυτό τους! Αυτή η διαδικασία συνεχίζεται με τρόπο ώστε κάθε νήμα να χρειάζεται να συμμετέχει τελικά σε numCoords transactions ώστε να φέρει όλα τα coordinates του object στο οποίο δουλεύει (1 transaction για κάθε coordinate). Με αυτό τον τρόπο, η transpose υλοποίηση επιτυγχάνει καλύτερες επιδόσεις από την naive, όπως φαίνεται και στα διαγράμματα, ενώ για παρόμοιους λόγους (που σχετίζονται με τη memory bound φύση του Kmeans), από ένα σημείο και έπειτα οι επιδόσεις σταθεροποιούνται και η GPU αδυνατεί να κρύψει το latency της μνήμης. Τέλος, να σημειωθεί ότι ενώ λόγω του τρόπου πρόσβασης στη μνήμη, η transpose υλοποίηση είναι πολύ καλή στην GPU, μια αντίστοιχη υλοποίηση σε CPU θα ήταν εξαιρετικά μη αποδοτική, καθώς εκεί το κάθε νήμα αποκτά ανεξάρτητα και ατομικά

πρόσβαση στη μνήμη και θα θέλαμε για κάθε νήμα να υπάρχει τοπικότητα στις αναφορές του στην μνήμη (δηλαδή τελικά οι αντίστοιχοι πίνακες να μην είναι transposed ώστε σε συνεχόμενες θέσεις μνήμης να υπάρχουν κατά σειρά τα coordinates ενός object στο οποίο δουλεύει το νήμα).

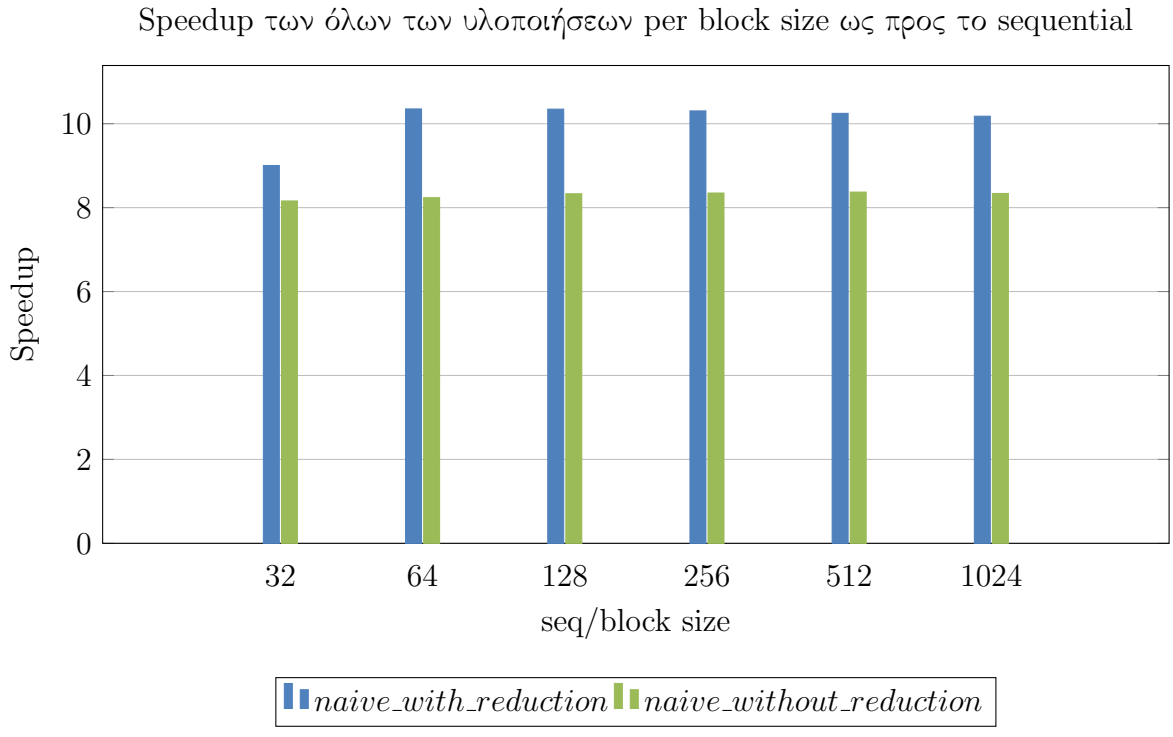
shared Στην υλοποίηση αυτή, το προφανές όφελος είναι ότι κάθε νήμα μπορεί πλέον να βρει τον πίνακα clusters στην κοντινή του shared memory, αντί να χρειαστεί πρόσβαση στην device memory. Δεδομένου ότι κάθε νήμα χρειάζεται να διαβάσει ολόκληρο τον πίνακα clusters για τον υπολογισμό του κοντινότερου cluster, η χρήση της shared memory μπορεί να βελτιώσει την επίδοση, όπως φαίνεται και στα διαγράμματα όπου η υλοποίηση αυτή επιτυγχάνει τις καλύτερες επιδόσεις. Με χρήση μικρού block size τα thread blocks θα είναι περισσότερα, δεδομένου ότι το συνολικό πλήθος νημάτων είναι συγκεκριμένο (σχεδόν ίσο με numObjs). Ως εκ τούτου, θα πρέπει να γίνουν περισσότερες μεταφορές του πίνακα cluster από την device στην shared memory, μια για κάθε thread block, επιδεινώνοντας τη συμφόρηση στην device memory. Από την άλλη με μεγάλο block size, ενώ οι μεταφορές που θα απαιτηθούν θα είναι λιγότερες, εντός ενός thread block τα πολλά νήματα που το απαρτίζουν θα δημιουργούν συμφόρηση στην shared memory για τα αιτήματα προσπέλασης του πίνακα clusters.

Τελικά : η shared υλοποίηση είναι καλύτερη από όλες τις άλλες υλοποιήσεις όπως εξηγήθηκε παραπάνω, ενώ συνολικά το ιδανικό block size είναι το 64 ή 128, όπου η shared υλοποίηση μπορεί να επιτύχει speedup ίσο με 11.22.

Διαγράμματα Στις επόμενες σελίδες παρατίθενται τα διαγράμματα επίδοσης στα οποία αναφερθήκαμε προηγουμένως.

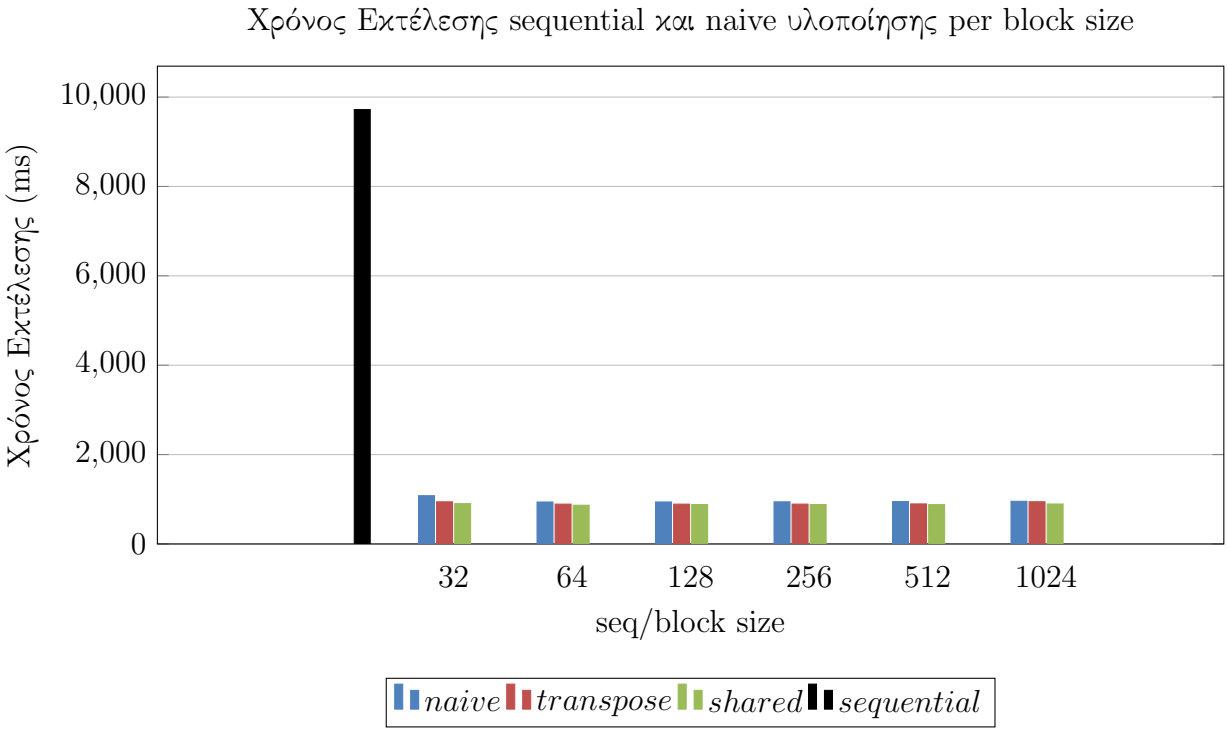


(a) Χρόνος Εκτέλεσης

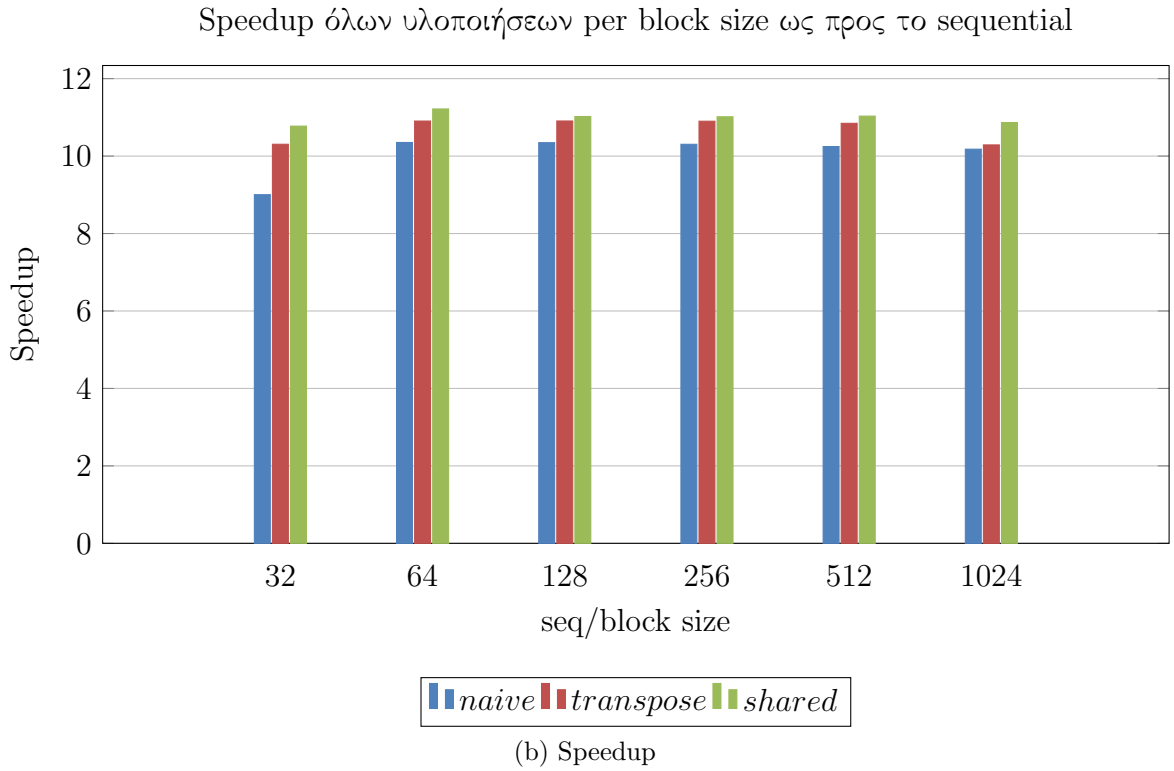


(b) Speedup

Εικόνα 22: Χρόνος Εκτέλεσης και Speedup ως προς το sequential των naive υλοποιήσεων per block size (χωρίς και με delta reduction)



(a) Χρόνος Εκτέλεσης



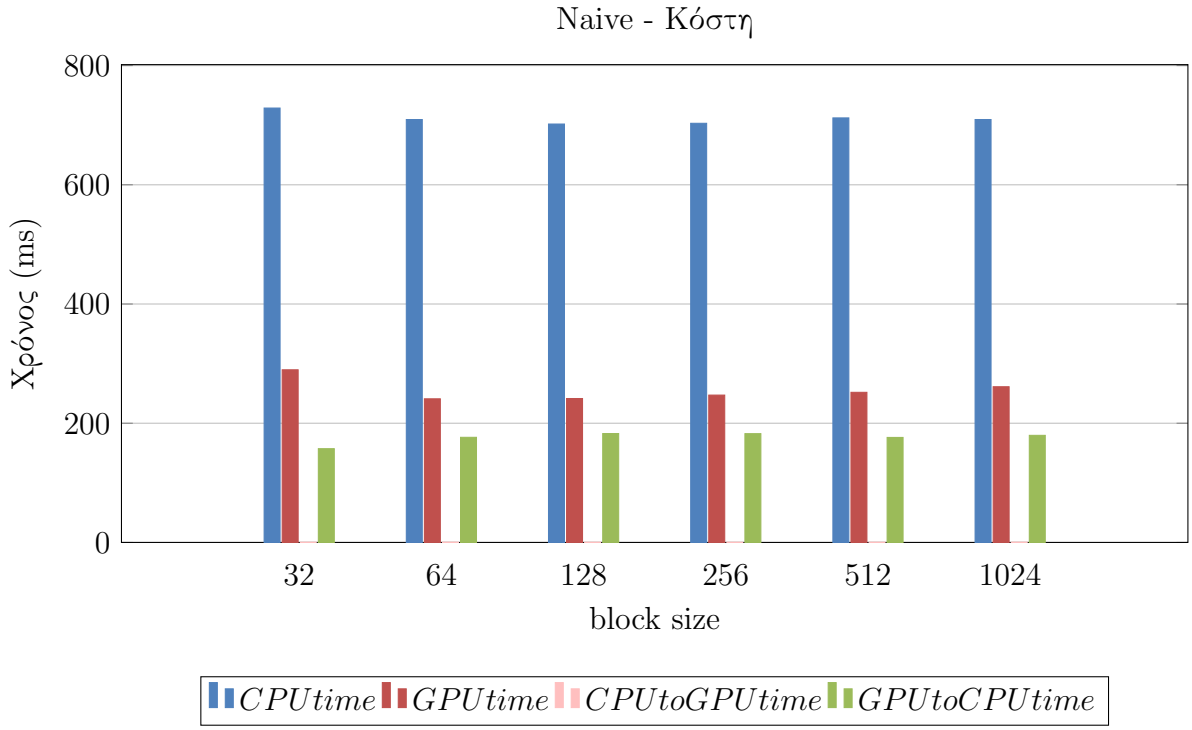
(b) Speedup

Εικόνα 23: Χρόνος Εκτέλεσης και Speedup ως προς το sequential όλων των υλοποιήσεων per block size (με delta reduction)

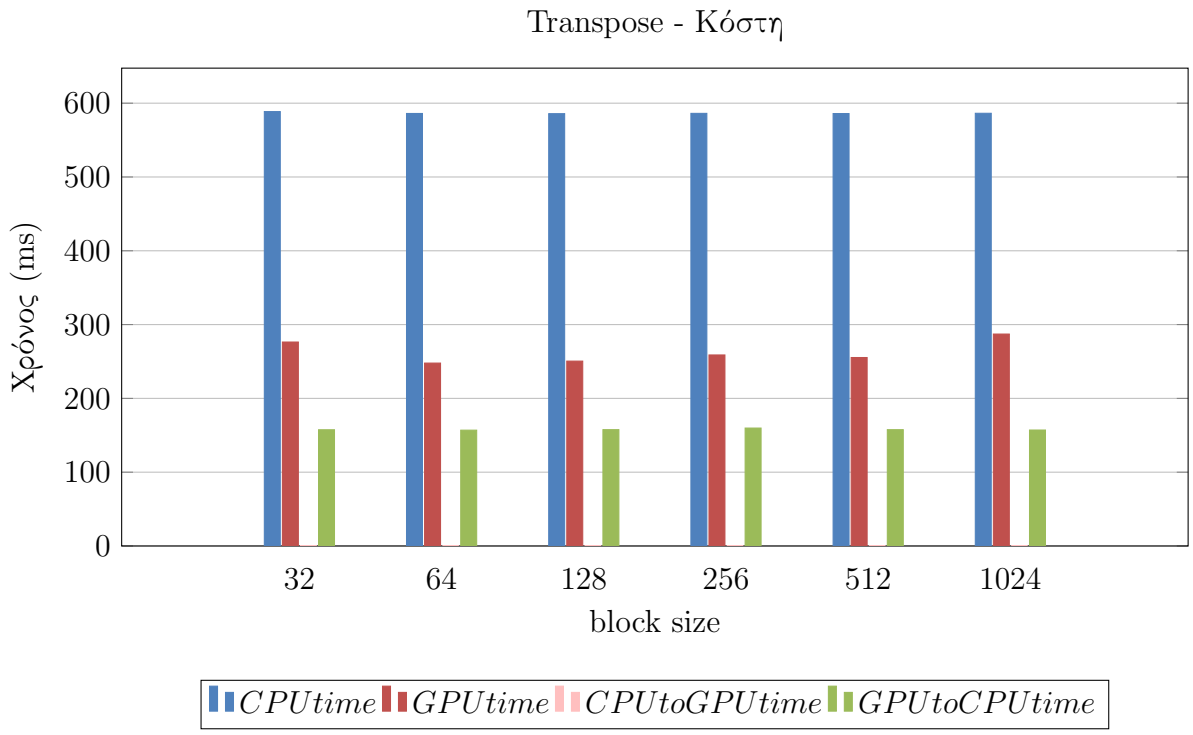
6.5 Σύγκριση Υλοποιήσεων - Bottleneck Analysis

6.5.1 Προσθήκη Timers

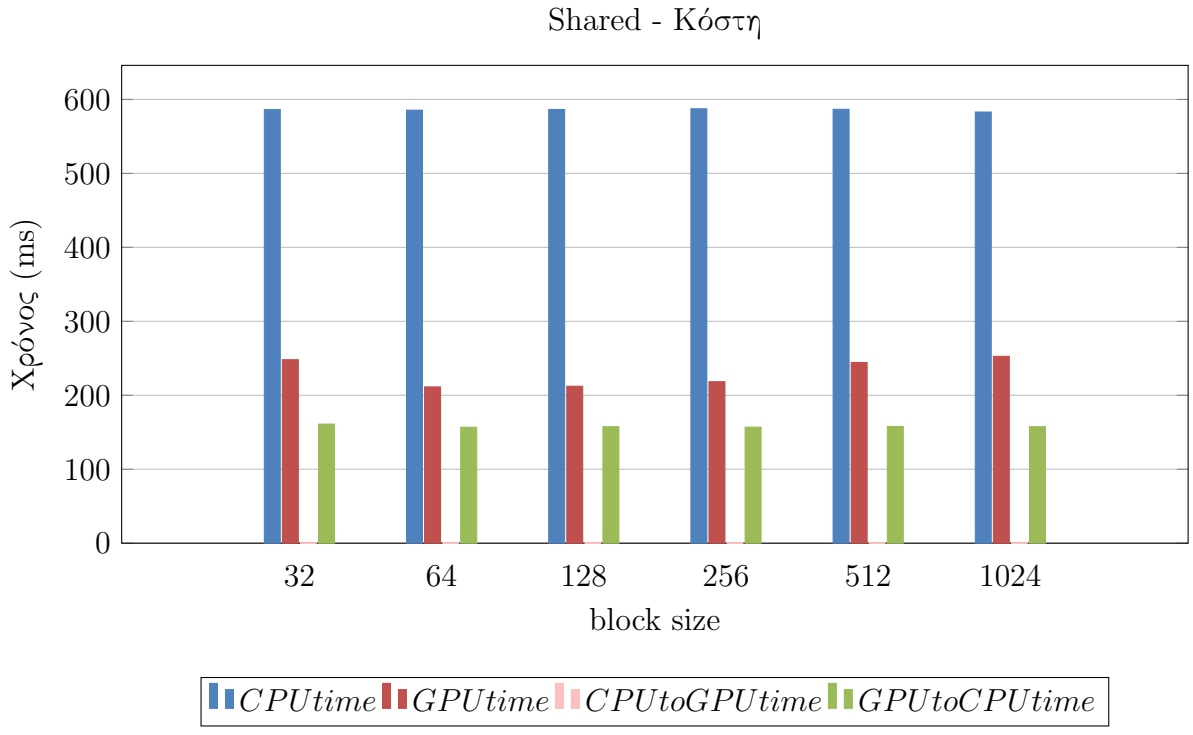
Στη συνέχεια, χρησιμοποιούμε κατάλληλους timers για τη μέτρηση του χρόνου που δαπανάται στην GPU, στην CPU και στην μεταφορά δεδομένων μεταξύ τους. Χρησιμοποιούμε τη συνάρτηση wtime() πριν και μετά κάθε τέτοια διαδικασία και με τη διαφορά των μετρήσεων αυτών υπολογίζουμε την αντίστοιχη χρονική διάρκεια (π.χ. τοποθετώντας wtime() πριν και μετά την κλήση του kernel, συμπεριλαμβάνοντας βέβαια και τον συγχρονισμό, μετράμε τον χρόνου που δαπανήθηκε στην GPU κ.ο.κ.). Τα αποτελέσματα που λαμβάνουμε είναι αυτά που φαίνονται στην εικόνα 24 παρακάτω. Αρχικά, παρατηρούμε ότι γενικά οι χρόνοι CPU είναι μεγαλύτεροι από τους αντίστοιχους χρόνους της GPU, οι μεταφορές από την GPU στη CPU (δηλαδή ο πίνακας membership, ο οποίος είναι αρκετά μεγάλος, μεγέθους ίσου με το πλήθος των αντικειμένων) αφενός δεν κυριαρχούν, ωστόσο είναι αρκετά σημαντικές (περίπου το 50% του χρόνου GPU) και δεν πρέπει να υποτιμώνται ενώ οι μεταφορές από την CPU στην GPU (η μεταβλητή delta και ο πίνακας clusters που είναι μικρός καθώς έχουμε λίγα clusters με λίγα coordinates) έχουν αμελητέα διάρκεια. Οι χρόνοι GPU αντικατοπτρίζουν τα συμπεράσματα που εξάγαμε προηγουμένως σχετικά με την επίδραση του τρόπου υλοποίησης στην λειτουργία της GPU (έτσι βλέπουμε βελτίωση του χρόνου GPU από naive στις άλλες δύο μεθόδους). Οι χρόνοι CPU δεν μεταβάλλονται ιδιαίτερα καθώς μπορεί μεν οι πίνακες να αντιστρέφονται στις δύο τελευταίες μεθόδους αλλά το πρόγραμμα CPU είναι σειριακό και έτσι δεν επηρεάζεται το locality των αναφορών που θα υπήρχε σε περίπτωση πολυνηματισμού στη CPU. Τέλος, οι χρόνοι μεταφορών είναι άμεσα συνδεδεμένοι με την ποσότητα δεδομένων που μεταφέρονται όπως φάνηκε και προηγουμένως.



(a) Naive



(b) Transpose

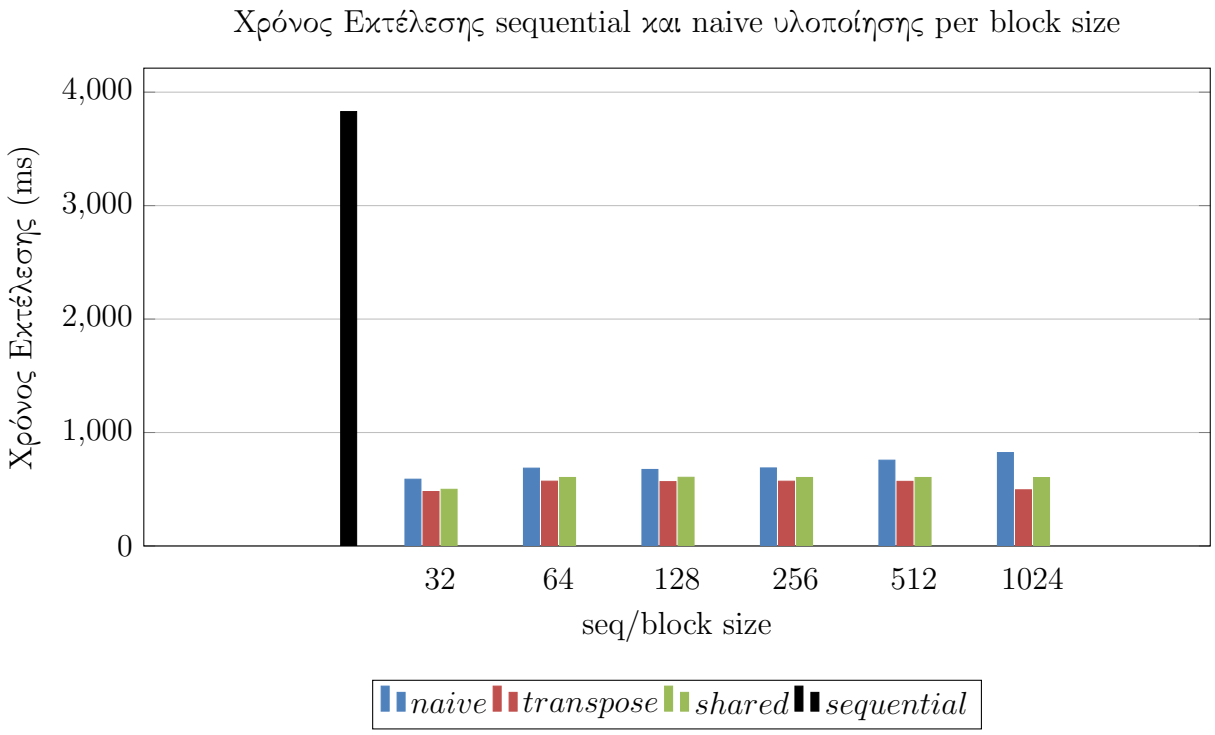


(c) Shared

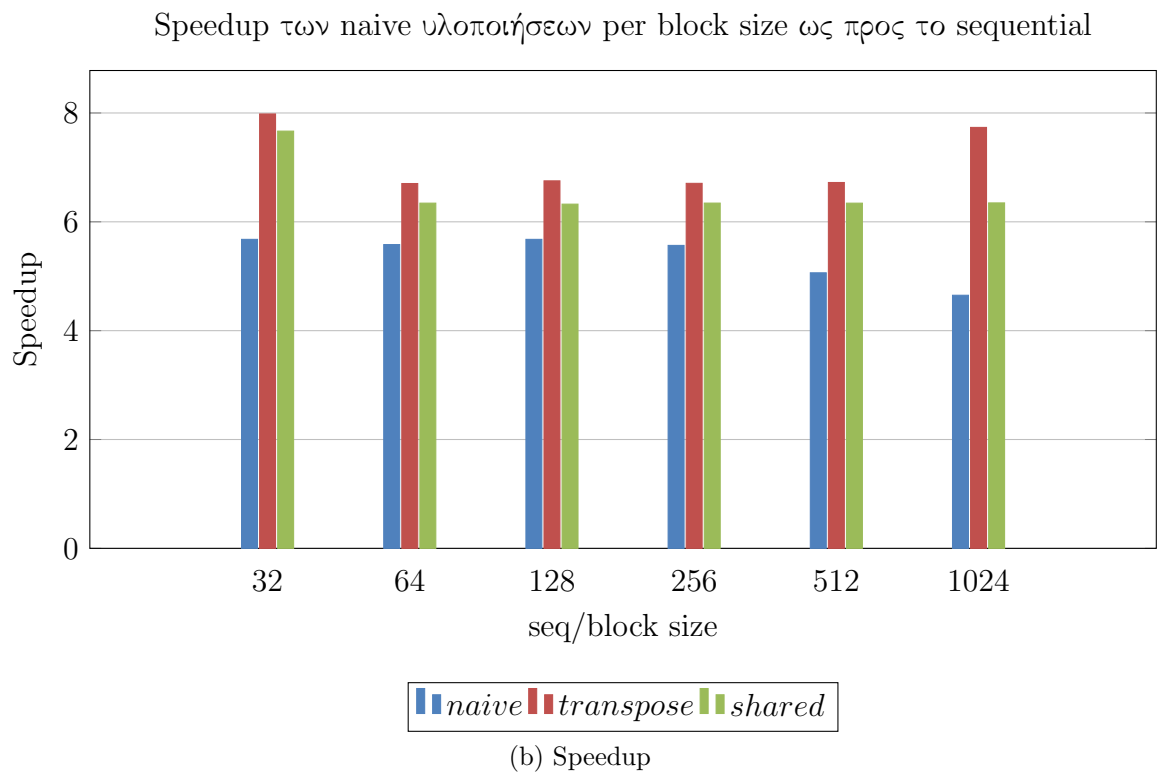
Εικόνα 24: Χρόνοι εκτέλεσης σε GPU και CPU και κόστη μεταφοράς δεδομένων μεταξύ τους για όλες τις υλοποιήσεις

6.5.2 Χρήση διαφορετικού configuration

Τρέχουμε πάλι τον K-means με όλες τις προηγούμενες υλοποιήσεις αλλά μεταβάλλοντας το configuration και συγκεκριμένα αυξάνοντας τα coordinates από 2 σε 16. Τα αποτελέσματα ως προς την επίδοση φαίνονται στην εικόνα 25 στην επόμενη σελίδα. Αρχικά, παρατηρούμε σημαντικά καλύτερη βελτίωση της επίδοσης από την naive προς την transpose υλοποίηση (αύξηση τουλάχιστον κατά 2 στο speedup). Αυτό οφείλεται στην δόμηση και στον τρόπο πρόσβασης στην shared memory, η οποία καθιστά την παρούσα naive υλοποίηση αρκετά χειρότερη σχετικά με το προηγούμενο configuration, καθιστώντας την transpose εκδοχή ευεργετική. Προηγουμένως στην naive υλοποίηση με δύο coordinates, με ένα memory transaction από ένα half-warp, ανά δύο νήματα (εκ των 16) διαβαζόταν μια τιμή χρήσιμη για ένα νήμα (για παράδειγμα τα πρώτα δύο νήματα διάβαζαν τα 2 coordinates του object 0 χρήσιμα για το νήμα 0, τα δύο επόμενα νήματα διάβαζαν τα 2 coordinates του object 1, χρήσιμα για το νήμα 1 κ.ο.κ). Συνολικά, με ένα memory transaction από 16 νήματα, διαβαζόταν χρήσιμη πληροφορία για 8 νήματα (και αυτό βέβαια δεδομένης της ύπαρξης cache!). Με 16 coordinates στην naive υλοποίηση, με ένα memory transaction (το πρώτο ας πούμε) από ένα half-warp, και τα 16 νήματα φέρνουν (ευτυχώς στην cache) όλα τα 16 coordinates του object 0, χρήσιμα για ένα μόνο νήμα, το νήμα 0. Επομένως, 16 νήματα διαβάζουν χρήσιμη πληροφορία για ένα μόνο νήμα. Έτσι λοιπόν, έρχεται στη συνέχεια η transpose υλοποίηση να βελτιώσει σημαντικά αυτό το πρόβλημα (καθώς σε κάθε memory transaction κάθε νήμα διαβάζει κάτι χρήσιμο για τον εαυτό του και μπορεί αμέσως να γίνει processing πάνω σε αυτή την πληροφορία). Σε ό,τι αφορά την shared υλοποίηση -σε αντίθεση με το προηγούμενο configuration- φαίνεται ότι επιδεινώνεται η επίδοση, με το αυξημένο μέγεθος της shared memory να παίζει σημαντικό ρόλο σε αυτό. Με μικρό block size, τα active thread blocks ανά SM μπορεί να είναι περισσότερα αυξάνοντας το contention πρόσβασης στην shared memory ενώ αντίστοιχα για μεγάλο block size, τα πολλά νήματα εντός ενός thread block που ταυτόχρονα θέλουν να διαβάσουν έναν αυξημένου μεγέθους πίνακα clusters επισης αυξάνουν το contention. Τέλος, δεδομένου του σταθερού συνολικού μεγέθους shared memory ανά SM, η απαίτηση για μεγαλύτερη δέσμευση shared memory ανά active thread block ενός SM περιορίζει το πλήθος των thread blocks που μπορούν να δρομολογηθούν ταυτόχρονα σε ένα SM, μειώνοντας με αυτό τον τρόπο το occupancy (καθώς θα υπάρξουν thread blocks που θα πρέπει να αναμένουν για τη δέσμευση των πόρων του SM). Συνεπώς, λόγω των παραπάνω, η shared υλοποίηση δεν είναι κατάλληλη για arbitrary configurations καθώς η επίδοσή της είναι συνδεδεμένη με το μέγεθος της shared memory, σε αντίθεση με την transpose της οποίας η επίδοση δεν εξαρτάται με τον ίδιο τρόπο από το μέγεθος του πίνακα clusters, καθώς αυτό δεν αποθηκεύεται στην shared memory (με ό,τι συνέπεια αυτό μπορεί να έχει).



(a) Χρόνος Εκτέλεσης



(b) Speedup

Εικόνα 25: Χρόνος Εκτέλεσης και Speedup ως προς το sequential όλων των υλοποιήσεων per block size (με delta reduction) με $Config = 256, 16, 16, 10$

ΜΕΡΟΣ Δ

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

7 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Εισαγωγή

Σε αυτό το μέρος της εργαστηριακής άσκησης, καλούμαστε να βελτιστοποιήσουμε/ παραλληλοποιήσουμε αλγορίθμους σε συστήματα κατανεμημένης μνήμης με χρήση του προγραμματιστικού μοντέλου ανταλλαγής μηνυμάτων, όπως αυτό υλοποιείται από το πρότυπο του MPI. Αρχικά ασχολούμαστε με τον αλγόριθμο K-means.

7.1 Υλοποίηση K-means - Κώδικας σε MPI

Αρχικά, εξηγούμε παρακάτω τον κώδικα που γράφτηκε σε MPI και κυρίως τα TODO tasks που καλούμασταν να υλοποιήσουμε. Ο κώδικας θα εξηγηθεί ανά αρχείο (main.c, kmeans.c, file-io.c) σε στοχευμένα σημεία.
Πολύ συνοπτικά, στον συνολικό κώδικα, κάθε MPI διεργασία αναλαμβάνει ένα μέρος των συνολικών αντικειμένων, τα αρχικά κέντρα των clusters γίνονται broadcast σε όλους ώστε έπειτα ο καθένας με κλήση της συνάρτησης kmeans να υπολογίσει τα κοντινότερα clusters για τα δικά του αντικείμενα, τα νέα κέντρα των clusters υπολογίζονται με ένα reduction μεταξύ των διεργασιών (εντός της συνάρτησης kmeans) και τελικά αφού ο αλγόριθμος ολοκληρωθεί φροντίζουμε από τα τοπικά αποτελέσματα membership των επιμέρους διεργασιών να παράξουμε τον συνολικό τελικό πίνακα tot_membership που για κάθε αντικείμενο αποθηκεύει το κοντινότερο σε αυτό cluster. Links για τους full .c κώδικες εδώ : **main.c**, **kmeans.c**, **file-io.c**.

main.c Εξηγούμε την υλοποίηση των TODO tasks στο βασικό αρχείο main.c.
Αρχικά κάνουμε broadcast τα initial clusters σε όλους. Η διεργασία 0, ως ο root του broadcast, μεταφέρει τον πίνακα clusters που περιέχει $numClusters \times numCoords$ doubles σε όλες τις διεργασίες στους τοπικούς πίνακες clusters.

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

Στη συνέχεια, καλείται ο kmeans για κάθε διεργασία. Πρέπει να προσαρμόσουμε το πλήθος των αντικειμένων για τα οποία κάθε διεργασία θα δουλέψει. Αυτό υπολογίζεται στην μεταβλητή rank_numObjs (από την συνάρτηση που δημιουργεί και κατανέμει τα objects) για κάθε διεργασία.

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

Έπειτα υπολογίζουμε τους πίνακες recvcounts, displs που θα χρησιμοποιηθούν για το Gather των τοπικών πινάκων membership. Ο recvcounts αναπαριστά το πλήθος των αντικειμένων που λαμβάνονται από κάθε διεργασία. Κάθε διεργασία στέλνει τον πίνακα membership που περιέχει rank_numObjs integers. Συνεπώς, ο recvcounts θα περιέχει τα rank_numObjs όλων των διεργασιών, τα οποία επομένως χρειάζεται να γίνουν gather από την διεργασία 0 ώστε

αυτή στη συνέχεια να υπολογίσει τον recvcounts. Η διεργασία 0 (root), κάνει gather 1 integer από κάθε διεργασία ο οποίος βρίσκεται στη μεταβλητή rank_numObjs και αποθηκεύει τους integers στον recvcounts όπως φαίνεται παρακάτω. Ο πίνακας displs αποθηκεύει τον index στον tot_membership όπου θα αποθηκευτούν τα τοπικά δεδομένα από κάθε διεργασία. Πρόκειται δηλαδή για ένα displacement (από την αρχή του πίνακα) όπου αποθηκεύονται τα δεδομένα που συλλέγονται από κάθε διεργασία. Υπολογίζεται όπως παρακάτω με τρόπο τέτοιο, ώστε να είναι:

$$displs[0] = 0$$

$$displs[1] = recvcounts[0]$$

$$displs[2] = recvcounts[0] + recvcounts[1]$$

```
int recvcounts[size], displs[size];

MPI_Gather(&rank_numObjs, 1, MPI_INT, recvcounts, 1, MPI_INT, 0,
    MPI_COMM_WORLD);

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
*/
displs[0]=0;
int ind1, ind2;
for(ind1=1;ind1<size;ind1++){
    int my_sum=0;
    for(ind2=0;ind2<=ind1-1;ind2++){
        my_sum+=recvcounts[ind2];
    }
    displs[ind1]=my_sum;
    }
}
```

Οι δύο παραπάνω πίνακες γίνονται broadcast από τον 0 (root) σε όλους ,οπως παρακάτω. Και οι δύο πίνακες περιέχουν size integers.

```
/*
* 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);
```

Τέλος, γίνονται gather οι τοπικοί πίνακες membership τον συνολικό πίνακα tot_membership. Χρησιμοποιείται η εκδοχή MPI_Gatherv που διότι οι διάφοροι τοπικοί πίνακες περιέχουν (και στέλνουν προς συλλογή) διαφορετικό πλήθος αντικειμένων. Χρησιμοποιούνται οι πίνακες recvcounts, displs ου ορίστηκαν παραπάνω.

```
/*
* 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 Εξηγούμε την υλοποίηση των TODO tasks στο αρχείο kmeans.c μέσω του οποίου κάθε διεργασία εκτελεί τοπικά τον kmeans αλγόριθμο στα objects που της αναλογούν. Η λειτουργία του αλγορίθμου kmeans για κάθε διεργασία είναι η γνωστή, απλώς θα πρέπει να προστεθεί η επικοινωνία μεταξύ των διεργασιών ώστε μετά τους επιμέρους υπολογισμούς σε κάθε iteration να γίνεται reduction των επιμέρους αποτελεσμάτων (πίνακες *rank_newClusters*, *rank_newClusterSize* και μεταβλητή σύγκλιση *rank_delta*) στα συναθροισμένα αποτελέσματα. Όλα τα reductions θα είναι προσθετικά, λόγω του τρόπου υπολογισμού των τοπικών αποτελέσμάτων σε κάθε διεργασία που είναι προσθετικός. Χρησιμοποιείται η *MPI_Allreduce* ώστε το reduction να γίνεται σε όλους τους κόμβους, καθώς όλοι θέλουν να έχουν την συναθροισμένη πληροφορία για να εκτελέσουν το επόμενο iteration τους επί των ανανεωμένων δεδομένων. Το reduction γίνεται όπως παρακάτω. Σημασία έχει να προστεθούν σωστά τα μεγέθη counts που αναφέρονται στο πλήθος των elements στον send buffer (πρώτο όρισμα). Ο *rank_newClusters* περιλαμβάνει όλα τα numCoords coordinates για όλα τα numClusters clusters, ο *rank_newClustersize* περιλαμβάνει το μέγεθος για κάθε cluster (από τα *numClusters*) και το *rank_delta* είναι ένας αριθμός.

```
        /*
* TODO: Perform reduction of cluster data (rank_newClusters,
    rank_newClusterSize) from local arrays to shared.
*/
MPI_Allreduce(rank_newClusters, newClusters, numClusters*numCoords,
    MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT,
    MPI_SUM, MPI_COMM_WORLD);
 /*
* TODO: Perform reduction from rank_delta variable to delta variable, that
     will be used for convergence check.
*/
MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
    ;
```

file-io.c Σε αυτό το αρχείο περιλαμβάνεται η συνάρτηση *dataset_generation* που καλεί κάθε διεργασία για να της ανατεθούν αντικείμενα. Αρχικά πρέπει να υπολογιστεί πόσα αντικείμενα θα λάβει κάθε διεργασία (πιθανώς αυτό το πλήθος δεν θα είναι ίδιο για όλες τις διεργασίες αν το πλήθος τους δεν είναι διαιρέτης του συνολικού πλήθους αντικειμένων). Η κατανομή που υλοποιήσαμε (με σκοπό να είναι όσο πιο ομοιόμορφη γίνεται) βασίζεται στη λογική round-robin. Κυκλικά, δίνεται σε κάθε διεργασία από ένα αντικείμενο, μέχρι αυτά να τελειώσουν. Εδώ γίνεται και ένα μέρος της υλοποίησης του επόμενου TODO. Ο υπολογισμός του sendcounts (που περιέχει πόσα αντικείμενα θα σταλούν σε κάθε διεργασία) μπορεί να υπολογιστεί από τώρα, καθώς μέσω του round robin τρόπου, κάθε φορά που σε μια διεργασία ανατίθεται ένα νέο αντικείμενο αρκεί να αυξάνεται η συγκεκριμένη καταχώριση στον πίνακα. Τελικά, το *rank_numObjs* της τρέχουσας διεργασίας θα είναι η καταχώριση του sendcounts στην θέση rank.

```
/*
* TODO: Calculate number of objects that each rank will examine (*
    rank_numObjs)
*/
// Round Robin Implementation
int sendcounts[size];
int q;
for(q=0;q<size;q++){
```

```
8       sendcounts[q]=0;
9   }
10  int proc=0;
11  int remaining_objects=numObjs;
12  while(remaining_objects>0){
13      sendcounts[proc]+=1;
14      remaining_objects-=1;
15      proc=(proc+1)%size;
16  }
17  *rank_numObjs = sendcounts[rank];
```

Τα displacements υπολογίζονται ακριβώς με τον ίδιο τρόπο, όπως προηγουμένως και για αυτό η διαδικασία δεν παρουσιάζεται πάλι. Οι δύο υπολογισθέντες πίνακες γίνονται ομοίως με πριν broadcast σε όλους.

```
1  /*
2  * TODO: Broadcast the sendcounts and displs arrays to other ranks
3  */
4  MPI_Bcast(sendcounts,size,MPI_INT,0,MPI_COMM_WORLD);
5  MPI_Bcast(displs,size,MPI_INT,0,MPI_COMM_WORLD);
```

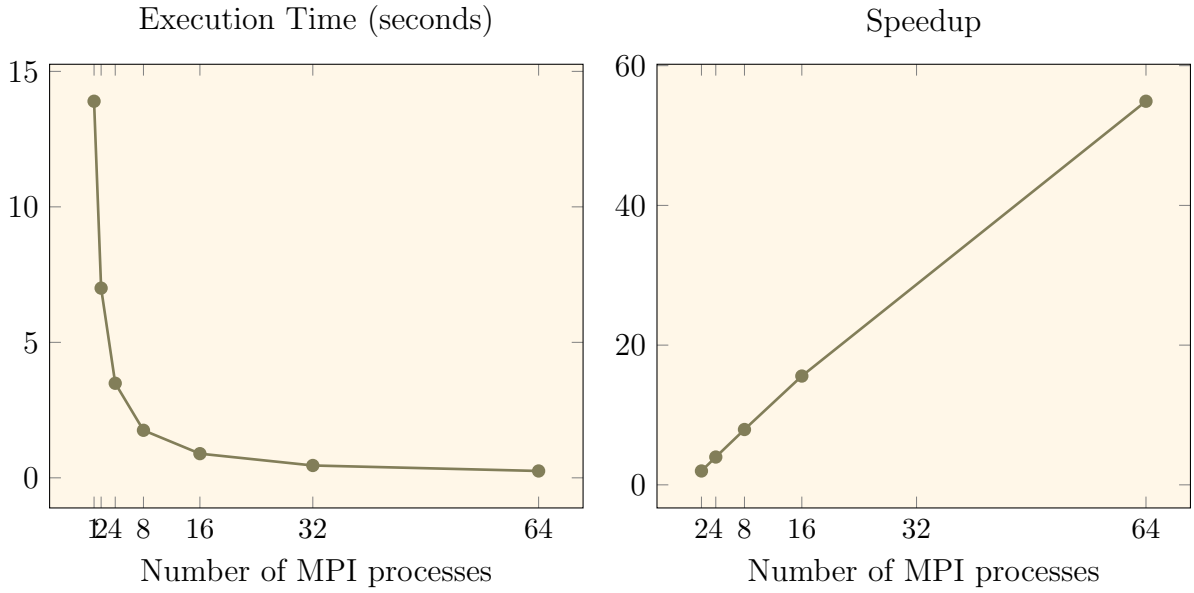
Τέλος, αφού παραχθούν για κάθε διεργασία τα αντικείμενα που της αντιστοιχούν, αυτά γίνονται scatter (ομοίως, επιλογή του Scatterv καθώς σε κάθε διεργασία ενδεχομένως να δοθεί διαφορετικό πλήθος από αντικείμενα). Ο συνολικός πίνακας objects γίνεται scatters στους τοπικούς *rank_objects* με χρήση των sendcounts και displs (που έχουν την ίδια χρησιμότητα με τους recvcounts, dislpls που παρουσιάστηκαν στην main). Η παράμετρος recvcount είναι $rank_numObjs \times numCoords$ καθώς ο πίνακας *rank_objects* περιλαμβάνει όλα τα coordinates των αντικειμένων της εκάστοτε διεργασίας.

```
1  /*
2  * TODO: Scatter objects to every rank. (hint: each rank may receive
       different number of objects)
3  */
4  MPI_Scatterv(objects,sendcounts,displs,MPI_DOUBLE,rank_objects,(*
       rank_numObjs)*numCoords,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

7.2 Επιδόσεις-Σύγκρισεις για διαφορετικό πλήθος MPI διεργασιών

Στη συνέχεια, παράγουμε τα διαγράμματα χρόνου εκτέλεσης και speedup (εικόνα 26) από τα οποία φαίνεται συγκριτικά η επίδοση του K-means όταν χρησιμοποιείται διαφορετικός αριθμός από MPI διεργασίες. Παρατηρούμε ότι ο χρόνος εκτέλεσης μειώνεται σημαντικά με χρήση όλο και περισσότερων MPI διεργασιών και μάλιστα το speedup είναι πάρα πολύ κοντά στο γραμμικό. Αυτό οφείλεται στις μειωμένες ανάγκες επικοινωνίας μεταξύ των διεργασιών κατά το κύριο υπολογιστικό μέρος (εκτέλεση της συνάρτησης kmeans()). Σε κάθε iteration του kmeans, κάθε διεργασία δουλεύει απρόσκοπτα στα δικά της αντικείμενα χωρίς να χριεάζεται καθόλου επικοινωνία με άλλες διεργασίες και μόνο στο τέλος κάθε iteration χρειάζεται ένα συνολικό collective communication (Allreduce) με το οποίο οι διεργασίες ενημερώνουν βασικές δομές δεδομένων. Ωστόσο, αυτή η επικοινωνία δεν είναι point to point με πολλαπλούς γείτονες (όπως θα δούμε στην επόμενη εφαρμογή του heat equation) αλλά συνολική για όλους, κάτι που την καθιστά ταχύτερη. Με μια συνολική coolective επικοινωνία όλες οι διεργασίες αποκτούν ό,τι

χρειάζονται για το επόμενο iteration. Σε ό,τι αφορά το κόστος των αρχικών και τελικών επικοινωνιών (πριν και μετά το κύριο υπολογιστικό μέρος της συνάρτησης kmeans()), αυτό αφορά διαδικασίες που γίνονται μόνο μια φορά στην αρχή ή στο τέλος του αλγορίθμου (π.χ. η κατανομή των objects στις διεργασίες η το συνολικό reduce για τον *tot_membership*) και έτσι δεν επιβαρύνουν σημαντικά τον αλγόριθμο. Τελικά το όφελος της χρήσης πολλαπλών διεργασιών (καταμερισμός της εργασίας μεταξύ των clones) υπερκαλύπτει τα overheads επικοινωνίας καθώς αυτά είναι μικρά και τελικά έχουμε πολύ καλή επίδοση.



(a) Execution Time

(b) Speedup

Εικόνα 26: Χρόνος εκτέλεσης και Speedup για την υλοποίηση του K-means με χρήση πολλαπλών MPI διεργασιών

7.3 Σύγκριση της MPI με την OpenMP υλοποίηση (bonus)

Συκρίνουμε την επίδοση της MPI υλοποίησης όπως καταγράφηκε παραπάνω με την επίδοση της OpenMP υλοποίησης όπως καταγράφηκε προηγουμένως τόσο στην Shared Clusters όσο και στην Copied Clusters and Reduce υλοποίηση. Σε ό,τι αφορά την πρώτη υλοποίηση στο OpenMP είναι σαφές ότι η επίδοσή της είναι πολύ χειρότερη από αυτή του MPI καθώς χρησιμοποιεί μια κοινή μνήμη στην οποία υπάρχει πολύ σημαντικό contention overhead. Από την μία, λόγω της κοινής μνήμης, το bandwidth ανά OpenMP core είναι μειωμένο, από την άλλη λόγω της αποθήκευσης των clusters στην κοινή μνήμη, για την προσπέλασή τους δημιουργείται πολύ μεγάλο contention. Η δεύτερη υλοποίηση προσπαθεί να διορθώσει αυτό το πρόβλημα "μοιράζοντας" την κοινή μνήμη μεταξύ των cores και δημιουργώντας αντίγραφα των clusters. Κάθε core έχει πρόσβαση σε ένα αντίγραφο των clusters το οποίο βρίσκει σε ένα τμήμα της κοινής μνήμης στο οποίο έχει πρόσβαση μόνο αυτό. Αυτό λύνει το πρόβλημα του contention (και έτσι οι επιδόσεις γίνονται εξαιρετικά καλύτερες) αλλά και πάλι η ύπραξη μιας μοναδικής κοινής μνήμης με συγκεκριμένο συνολικό bandwidth, μειώνει το bandwidth που είναι διαθέσιμο ανά κόμβο ενώ συγχρόνως δημιουργείται και επιπλέον overhead της διαχείρισης των πολλαπλών copies

των clusters εντός της κοινης μνήμης. Αντιθέτως στο MPI τα αντίγραφα αυτά αποθηκεύονται σε πολλαπλές μνήμες (στην μνήμη του κάθε clone), κάτι που προφανώς κάνει το contention μηδενικό, αφετέρου εξασφαλίζει πολύ μεγαλύτερο bandwidth μνήμης σε κάθε κόμβο. Το μόνο επιπρόσθετο κόστος στο MPI είναι ότι τα τελικά reductions απαιτούν δικτυακή επικοινωνία μεταξύ των κόμβων ενώ στο OpenMP το reduction γίνεται με χρήση της κοινής μνήμης εντός του ενός κόμβου αρκετά ταχύτερα. Ωστόσο, λόγω του μικρού overhead επικοινωνίας (όπως εξηγήθηκε παραπάνω), τελικά το MPI καταλήγει να είναι ακόμα καλύτερο και από την δεύτερη υλοποίηση στο OpenMP.

Συνολικά, σχετικά με την σύγκριση του OpenMP και του MPI, το πρώτο ενδείκνυται για χρήση με λίγα cores καθώς έτσι συνδυάζεται η εγγενής στο OpenMP μικρού κόστους επικοινωνία (συγκέντρωση δεδομένων από την κοινή μνήμη) με το μικρό contention στην μνήμη που εξασφαλίζεται από τα λίγα cores. Αντιθέτως, το MPI προτιμάται για χρήση με αρκετές MPI διεργασίες καθώς το εγγενές πλεονέκτημα της απουσίας contention (λόγω μη κοινής μνήμης) να συνδυάζεται με το σχετικά μικρό κόστος επικοινωνίας (καθώς με πολλές διεργασίες, τελικά το όφελος της παραλληλίας λόγω των πολλών διεργασιών υπερνικά τα overheads δικτυακής επικοινωνίας).

Τέλος, μια σκέψη συνδυασμού των επιμέρους πλεονεκτημάτων θα ήταν η χρήση MPI όπως υλοποιήθηκε παραπάνω, με τη διαφορά ότι κάθε MPI διεργασία εσωτερικά θα μπορούσε να χρησιμοποιεί OpenMP για την εκτέλεση της υποεργασίας που έχει ανάλάβει με χρήση πολλαπλών συνεργαζόμενων (μέσω κοινής μνήμης) cores.

8 Διάδοση Θερμότητας σε δύο Διαστάσεις

Εισαγωγή

Στη συνέχεια της παρύσας εργαστηριακής άσκησης, υλοποιούμε σε MPI το πρόβλημα της εξίσωσης θερμότητας, ενός αλγορίθμου τύπου stencil που θυμίζει το Game of Life. Δίνονται 3 μέθοδοι υλοποίησης. Υλοποιείται (προς το παρόν) η μέθοδος Jacobi.

8.1 Εντοπισμός Παραλληλισμού-Σχεδιασμός

Η ιδέα για την παράλληλη εκτέλεση του αλγορίθμου heat-transfer έγκειται στην κατανομή του υπολογογιστικού χωρίου στις MPI διεργασίες, ώστε αυτές να υπολογίσουν την θερμότητα στο δικό τους χωρίο. Συγκεκριμένα, θα χρησιμοποιηθεί cartesian grid από MPI processes (ορισμός του καρτεσιανού communicator $CART\_COMM$ για διευκόλυνση της επικοινωνίας διεργασιών εντός του 2διάστατου grid), μέ τέτοιο τρόπο ώστε κάθε διεργασία να αναλάβει να υπολογίσει ένα συγκεκριμένο διδιάστατο χωρίο. Για αυτό το σκοπό, ωστόσο, θα χρειαστεί κάθε διεργασία να επικοινωνήσει με γειτονικές της προκειμένου να αντάλλάξει χρήσιμες τιμές θερμότητας. Αυτή η διαδικασία περιγράφεται στο επόμενο section όπου εξηγείται η καθεμία από τις 3 μεθόδους αναλυτικά.

8.2 Υλοποίηση σε MPI - Ανάπτυξη παράλληλου προγράμματος

Μέθοδος Jacobi - Κώδικας σε MPI [Link για full κώδικα : εδώ]

Με τη μέθοδο αυτή, η θερμότητα σε κάθε σημείο του grid μια χρονική στιγμή υπολογίζεται ως ο μέσος όρος των θερμοτήτων των γειτονικών σημείων (σε σχήμα σταυρού) την προηγούμενη χρονική στιγμή. Οι διεργασίες θα συγκροτήσουν ένα διδιάστατο πλέγμα μέσω ενός ορισθέντος καρτεσιανού communicator ($CART\_COMM$), ώστε η επικοινωνία τους να υλοποιηθεί πιο έυκολα (μιας και το grid του χωρίου προς εξέταση είναι διδιάστατο και κάθε διεργασία θα χρειάζεται να επικοινωνεί με γειτονικές της σε ένα διδιάστατο χωρίο). Διαισθητικά, το χωρίο προς εξέταση χωρίζεται σε τμήματα (κουτάκια) και κάθε διεργασία υπολογίζει την θερμότητα στο δικό της κουτί. Για αυτό όμως ενδέχεται κατά περιπτώσεις να χρειαστεί να επικοινωνήσει με γειτονικές της διεργασίες. Παρακάτω παρουσιάζεται ο κώδικας στα διάφορα TODO tasks που έπρεπε να υλοποιηθούν.
Αρχικά, ο global πίνακας U με την θερμότητα σε όλο το χωρίο, θα πρέπει να μοιραστεί στις διεργασίες. Κάθε διεργασία κρατάει δύο πίνακες *u_current*, *u_previous* για την θερμότητα των σημείων του δικού της χωρίου. Οι δύο αυτοί πίνακες περιέχουν περιμετρικά τους ένα extra στρώμα σημείων ώστε εκεί να αποθηκεύονται γραμμές και στήλες που λήφθηκαν κατόπιν επικοινωνίας από γειτονικές διεργασίες. Αρχικά, οι δύο πίνακες αρχικοποιούνται στο 0 με χρήση της zero2d που βρίσκεται στο αρχείο utils.c υλοποιημένη. Οι πίνακες αυτοί έχουν διαστάσεις $local[0] \times local[1]$ για το χρήσιμο κομμάτι του χωρίου τους και οι προσθέσεις κατά 2 αφορούν το extra περιμετρικό πλέγμα. Για το scatter γίνεται χρήση της Scatterv (καθώς κάθε διεργασία μπορεί να μην πάρει ίσου μεγέθους χωρίο αν το συνολικό χωρίο δεν διαιρείται ακριβώς στις διεργασίες). Γίνεται scatter ενός global block και οι διεργασίες λαμβάνουν ένα local block (datatypes που έχουν ήδη οριστεί). Κάθε διεργασία λαμβάνει το αντίστοιχο local block στο σημείο του πίνακα *u_current* όπου αρχίζει το χρήσιμο τμήμα του χωρίου (προσπερνώντας το extra layer περιμετρικά). Τέλος, ο *u_previous* γίνεται ίσος με τον *u_current* (αρχικοποίηση).

```
1  //----Rank 0 scatters the global matrix----//
2
3  //*************TODO*******************//
4
5  /*Fill your code here*/
6
7  /*Make sure u_current and u_previous are
8  both initialized*/
9  /*  int ind_x, ind_y;
10 for(ind_x=0;ind_x<local[0]+2;ind_x++){
11     for(ind_y=0;ind_y<local[1]+2;ind_y++){
12         u_current[ind_x][ind_y]=0;
13         //u_previous[ind_x][ind_y]=0;
14     }
15 }
16 */
17
18 zero2d(u_current,local[0]+2,local[1]+2);
19 zero2d(u_previous,local[0]+2,local[1]+2);
20
21
22 MPI_Scatterv(*U,scattercounts,scatteroffset,global_block,(*u_current+local
       [1]+3),1,local_block,0,CART_COMM);
23
```

```
int ind_x, ind_y;
for(ind_x=0;ind_x<local[0]+2;ind_x++){
    for(ind_y=0;ind_y<local[1]+2;ind_y++){
        u_previous[ind_x][ind_y]=u_current[ind_x][ind_y];
    }
}
```

Όπως αφέθηκε να εννοηθεί παραπάνω, οι διεργασίες θα ανταλλάσσουν τις περιμετρικές του γραμμές και στήλες καθώς θα τους είναι χρήσιμες για τον υπολογισμό θερμότητας στα συνοριακά τους σημεία. Για αυτό το λόγο ορίζονται κατάλληλοι τύποι δεδομένων όπως παρακάτω (το row ως contiguous ενώ το column ως vector καθώς δεν περιλαμβάνει συνεχόμενα στοιχεία αλλά στοιχεία με κατάλληλο 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, για κάθε διεργασία στο καρτεσιανό πλέγμα υπολογίζεται το rank των γειτονικών της διεργασιών με χρήση της MPI_Cart_shift. Η εντολή προσαρμόζεται κατάλληλα για την εύρεση των 4 χρήσιμων γειτόνων north,south,west,east. Παρατηρήθηκε ότι όταν ένας εκ των γειτόνων δεν υπάρχει (για διεργασίες περιμετρικές του καρτεσιανού πλέγματος) η επιστρεφόμενη τιμή rank είναι αρνητική. Έτσι έγιναν handle και αυτές οι περιπτώσεις.

```
//----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*/
// printf("%d ",rank);
MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1,1, &west, &east);
```

Στη συνέχεια, υπολογίζουμε το χωρίο στο οποίο κάθε διεργασία θα εκτελέσει τα Jacobi iterations της. Ορίζουμε κατάλληλες boolean συναρτήσεις για τον διαχωρισμό των τριών περιπτώσεων (εσωτερικές διεργασίες, συνοριακές αλλά χωρίς padding και συνοριακές με padding) και για κάθε μια από τις περιπτώσεις εξαντλητικά ορίστηκαν τα iteration ranges για όλες τις δυνατές περιπτώσεις όπως φαίνεται στον κώδικα που παρατίθεται με link στο dropbox. Το grid

μπορεί να είναι right padded, δηλαδή να χρειαστεί συμπλήρωση σημείων στην δεξιά πλευρά του grid ώστε ο χωρισμός του στις διεργασίες να είναι ομοιόμορφος, μπορεί να είναι bottom padded, δηλαδή αντίστοιχα να χρειάζεται padding στο κάτω μέρος του grid ή να είναι και τα δύο, κάτι που υπολογίζεται με κάποια απλά modulo operations στον κώδικα. Αντίστοιχα υπολογίζεται το ποσό του padding στον ομώνυμο πίνακα προς τις δύο κατευθύνσεις.
Έπειτα οι διεργασίες εισάγονται στο computational core στο οποίο πρέπει να επικοινωνήσουν με τις γειτονικές διεργασίες ώστε να ανταλλάξουν περιμετρικά rows και columns και στη συνέχεια (αφού θα έχει πλέον γεμίσει το extra layer του *u_current* με τις απαραίτητες πληροφορίες) να κληθεί η βασική συνάρτηση Jacobi που λήφθηκε αυτούσια από την σειριακή έκδοση. Ανάλογα με τι είδους γείτονες έχει κάθε διεργασία (έλεγχος για αρνητικά ή μη ranks γειτόνων όπως εξηγήθηκε), θα πρέπει να επικοινωνήσει με αυτούς. Θα κάνει ένα send και ένα receive σε καθέναν από αυτούς χρησιμοποιώντας κατάλληλες διευθύνσεις στα ορίσματα πηγής προορισμού με βάση όσα εξηγήθηκαν για το πού αποθηκεύονται τα δεδομένα που ανταλλάσσονται. Χρειάζεται στο τέλος να γίνει Waitall όλων των αιτημάτων send και receive ώστε όλες οι διεργασίες να καλέσουν την Jacobi έχοντας αποκτήσει όλα τα απαραίτητα δεδομένα. Γίνεται επίσης swap μεταξύ των current και previous πινάκων ώστε ο previous να γίνει πλέον ο current και ο current να δείχνει εκεί που έδειχνε ο previous και να κάνει overwrite το συγκεκριμένο τμήμα μνήμης. Στον έλεγχο σύγκλισης φροντίζουμε - για να συγκλίνει συνολικά ο αλγόριθμος - να έχουν συγκλίνει όλες οι MPI διεργασίες. Δεν απαιτείται παρά μόνο ένα reduction (φανερό σε όλους - Allreduce) που να υπολογίζει το λογικό AND των αντίστοιχων τοπικών μεταβλητών σύγκλισης.

```
1  //*************TODO*******************//
2
3  /*Fill your code here*/
4
5  /*Compute and Communicate*/
6
7  /*Add appropriate timers for computation*/
8  MPI_Request requests[8];
9  int counter = 0;
10
11 if(north>=0){
12     //MPI_Sendrecv(*u_current+local[1]+3,1,row,north,0,*u_current+1,1,row,
           north,0,MPI_COMM_WORLD,&requests[counter]);
13
14     MPI_Isend(*u_current+local[1]+3,1,row,north,0,MPI_COMM_WORLD,&requests
           [counter]);
15     counter+=1;
16     MPI_Irecv(*u_current+1,1,row,north,0,MPI_COMM_WORLD,&requests[counter
           ]);
17     counter+=1;
18 }
19 if(south>=0){
20     //MPI_Sendrecv(*u_current+local[0]*(local[1]+2)+1,1,row,south,1,*
           u_current+((local[1]+2)*(local[0]+1))+1,1,row,south,1,
           MPI_COMM_WORLD,&requests[counter]);
21
22     MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,
           MPI_COMM_WORLD,&requests[counter]);
23     counter+=1;
24     MPI_Irecv(*u_current+((local[1]+2)*(local[0]+1))+1,1,row,south,0,
```

```
        MPI_COMM_WORLD,&requests[counter]);
25      counter+=1;
26  }
27  if(east>=0){
28      //MPI_Sendrecv(*u_current+2*(local[1]+1),1,column,east,3,*u_current
            +2*(local[1]+1)+1,1,column,east,3,MPI_COMM_WORLD,&requests[counter
            ]);
29
30      MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&
            requests[counter]);
31      counter+=1;
32      MPI_Irecv(*u_current+2*(local[1]+1)+1,1,column,east,0,MPI_COMM_WORLD,&
            requests[counter]);
33      counter+=1;
34  }
35  if(west>=0){
36      //MPI_Sendrecv(*u_current+local[1]+3,1,column,west,4,*u_current+local
            [1]+2,1,column,west,4,MPI_COMM_WORLD,&requests[counter]);
37
38      MPI_Isend(*u_current+local[1]+3,1,column,west,0,MPI_COMM_WORLD,&
            requests[counter]);
39      counter+=1;
40      MPI_Irecv(*u_current+local[1]+2,1,column,west,0,MPI_COMM_WORLD,&
            requests[counter]);
41      counter+=1;
42  }
43
44  MPI_Status status[8];
45  MPI_Waitall(counter,requests,status);
46
47  swap=u_previous;
48  u_previous=u_current;
49  u_current=swap;
50
51  gettimeofday(&tcs,NULL);
52
53  Jacobi(u_previous,u_current,i_min,i_max,j_min,j_max);
54  gettimeofday(&tcf,NULL);
55  tcomp+=(tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
56
57
58  #ifdef TEST_CONV
59  if (t%C==0) {
60  //*************TODO*************//
61  /*Test convergence*/
62      converged=converge(u_previous,u_current,i_min,i_max-1,j_min,j_max-1);


63
64      MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND,
            MPI_COMM_WORLD);
65  }
66  #endif
67
68  //************************************//
69  }
```

Στο τέλος ο global matrix U θα πρέπει να υπολογιστεί συλλέγοντας τα περιεχόμενα των επιμέρους πινάκων *u_current* των διεργασιών με χρήση της Gatherv.

```
//----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 / Κώδικας σε MPI [Link για full κώδικα : εδώ]

Στη συνέχεια, υλοποιούμε τον heat-transfer αλγόριθμο με τη μέθοδο Gauss-Seidel, της οποίας οι διαφοροποιήσεις έγκειται αφενός στον κύριο υπολογιστικό πυρήνα (συνάρτηση Gauss-Seidel αντί Jacobi), αφετέρου στον τρόπο επικοινωνίας μεταξύ των διεργασιών. Η συνάρτηση Gauss-Seidel χρησιμοποιήθηκε αυτούσια από τον σειριακό κώδικα και ο ορισμός της παραμέτρου omega αφέθηκε όπως δίνεται στον κώδικα. Στη μέθοδο αυτή για λόγους ταχύτερης σύγκλισης, οι τιμές θερμότητας του βόρειου και δυτικού στοιχείου λαμβάνονται από την παρούσα και όχι από την προηγούμενη χρονική στιγμή. Για να επιτευχθεί αυτό, χρειάζονται κάποιες επιπλέον επικοινωνίες σε σχέση με την μέθοδο Jacobi. Η διαφορά είναι ότι πριν τον υπολογιστικό πυρήνα (κλήση της GaussSeidel()), οι διεργασίες με βόρειο και δυτικό γείτονα θα πρέπει να κάνουν receive από αυτούς και αυτοί (οι οποίοι προφανώς θα έχουν νότιο και ανατολικό γείτονα) να κάνουν send μετά την κλήση του πυρήνα (οπότε και θα έχουν υπολογίσει τις θερμότητες της παρούσας χρονικής στιγμής). Αυτές οι προσθήκες φαίνονται παρακάτω.

```
MPI_Request RecRequests[8];
int RecCounter = 0;

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

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

MPI_Waitall(RecCounter,RecRequests,status);


```

```
18 gettimeofday(&tcs,NULL);
19 GaussSeidel(u_previous, u_current,i_min, i_max, j_min, j_max, omega);
20 gettimeofday(&tcf,NULL);
21 tcomp+=(tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
22
23 MPI_Request SendRequests[8];
24 int SendCounter = 0;
25
26 if(south>=0){
27     MPI_Isend(*u_current+local[0]*(local[1]+2)+1,1,row,south,0,
          MPI_COMM_WORLD,&SendRequests[SendCounter]);
28     SendCounter+=1;
29 }
30 if(east>=0){
31     MPI_Isend(*u_current+2*(local[1]+1),1,column,east,0,MPI_COMM_WORLD,&
          SendRequests[SendCounter]);
32     SendCounter+=1;
33 }
34
35
36 #ifdef TEST_CONV
37 if (t%C==0) {
38     //*************TODO**************//
39     /*Test convergence*/
40     converged=converge(u_previous,u_current,i_min,i_max-1,j_min,j_max-1);
41
42     MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND,
          MPI_COMM_WORLD);
43
44 }
45 #endif
46 MPI_Waitall(SendCounter,SendRequests,status);
47
48 //************************************//
49 }
```

Μέθοδος Red Black / Κώδικας σε MPI [Link για full κώδικα : εδώ]

Στη συνέχεια, υλοποιούμε τον heat-transfer αλγόριθμο με τη μέθοδο Red Blackl, μέσω τις οποίας (για λόγους ταχύτερης σύγκλισης) τα στοιχεία του υπολογιστικού χωρίου χωρίζονται σε άρτια με $(i+j)\%2 = 0$ (Red) και σε περιττά με $(i+j)\%2 = 1$ (Black). Ο υπολογισμός των Red στοιχείων γίνεται βάσει της προηγούμενης χρονικής στιγμής εξ' ολοκλήρου (όπως ο Jacobi) ενώ ο υπολογισμός των Black στοιχείων γίνεται βάσει της τρέχουσας χρονικής στιγμής εξ' ολοκλήρου και για αυτό απαιτεί κάποια επιπλέον επικοινωνία μετά τον υπολογιστικό πυρήνα, οπότε και έχουν υπολογιστεί οι θερότητες της τρέχουσας χρονικής στιγμής. Ειδικότερα, διαχωρίζουμε τον υπολογιστικό πυρήνα σε δύο συναρτήσεις RedSOR(), BlackSOR() (οι οποίες λαμβάνονται αυτούσιες από τη σειριακή υλοποίηση). Η RedSOR() απαιτεί τιμές μόνο της προηγούμενης χρονικής στιγμής και έτσι θα κληθεί αμέσως μετά τις αρχικές επικοινωνίες (όπως έκανε και η Jacobi δηλαδή). Μετά την κλήση RedSOR() και πριν την BlackSOR() απαιτείται εκ νέου επικοινωνία όπου όλοι ανταλλάσουν με όλους τις τωρινές τους τιμές θερμότητας, ώστε η BlackSOR() να κληθεί με αυτές. Να σημειωθεί ότι κατά τη διάρκεια αυτής της διαδικα-

σίας πιθανόν κάποιες τιμές να γίνουν overwrite (για παράδειγμα αν μια διεργασία δεν εισαχθεί στην RedSOR() γιατί είναι περιττή, οι τιμές που απέκτησε από την αρχική επικοινωνία -πριν την RedSOR()- θα γίνουν overwrite από τις τιμές που αποκτά μετά ώστε να εισαχθεί στην BlackSOR()).

```
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;
```

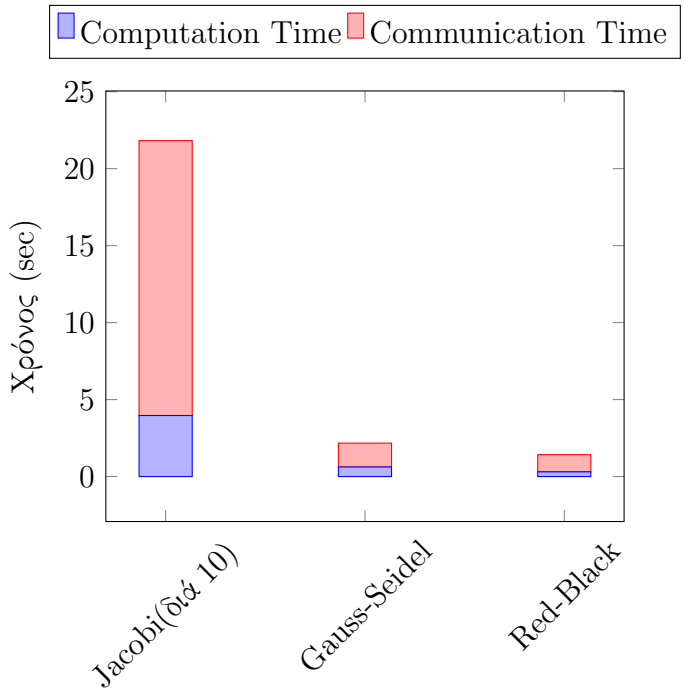
```
44
45
46
47 #ifdef TEST_CONV
48 if (t%C==0) {
49     //*************TODO**************//
50     /*Test convergence*/
51     converged=converge(u_previous,u_current,i_min,i_max-1,j_min,j_max-1);
52
53     MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND,
           MPI_COMM_WORLD);
54 }
55 #endif
56
57 //************************************//
58 }
```

8.3 Μετρήσεις Επίδοσης - Συγκρίσεις Μεθόδων

Παρακάτω για κάθε ένα από τα δύο σενάρια μετρήσεων, παρατίθενται οι ζητούμενες μετρήσεις επίδοσης.

8.3.1 Με χρήση ελέγχου σύγκλισης

Στο σενάριο αυτό, για μέγεθος grid 1024×1024 και για 64 MPI διεργασίες οι οποίες χρησιμοποιούν έλεγχο σύγκλισης, μετράται για κάθε μια από τις 3 μεθόδους ο χρόνος εκτέλεσης του αλγορίθμου heat-transfer και η κατανομή αυτού του χρόνου σε υπολογιστικό χρόνο και χρόνο επικοινωνίας(στον οποίο θεωρούμε ότι συμπεριλαμβάνεται και ο πραγματικά αμελητέος χρόνος αρχικοποιήσεων). Τα αποτελέσματα που λαμβάνουμε φαίνονται στο παρακάτω διάγραμμα της εικόνας 27 . Ο χρόνος για την μέθοδο Jacobi προέκυπτε πολύ μεγαλύτερος από αυτόν των άλλων δύο μεθόδων ($\times 100$ περίπου) και έτσι στο διάγραμμα παρουσιάζεται (για λόγους ευκρίνειας) υπό κλίμακα (διαιρεμένος διά του 10). Παρατηρούμε ότι η μέθοδος Red Black είναι καλύτερη από όλες, ακολουθεί με μικρή διαφορά η Gauss Seidel και με πολύ μεγάλη διαφορά έπειτα ακολουθεί η Jacobi. Δεδομένου ότι το σενάριο αυτό περιλαμβάνει έλεγχο σύγκλισης, είναι αναμενόμενο η Jacobi να έχει την χειρότερη επίδοση (καθώς χρησμοποιεί τον trivial τρόπο υλοποίησης). Οι άλλες δύο μέθοδοι είναι με τέτοιο τρόπο προσαρμοσμένες ακριβώς για να συγκλίνουν ταχύτερα και έτσι επιτυγχάνουν πολύ καλύτερες επιδόσεις. Οι δύο αυτές μέθοδοι χρησιμοποιούν μεν επιπρόσθετες επικοινωνίες (οι οποίες σαφώς είναι χρονοβόρες) αλλά μέσω αυτών καταφέρνουν να συγκλίνουν πολύ ταχύτερα. Έτσι η ταχύτητα σύγκλισης επικρατεί των overheads επικοινωνίας και τελικά έχουν πολύ καλές επιδόσεις (αυτή η συμπεριφορά δεν θα επαναληφθεί με τον ίδιο τρόπο παρακάτω όπου η σύγκλιση δεν χρησιμοποιείται). Τέλος επισημαίνουμε ότι στο σενάριο αυτό, το κύριο μέρος του χρόνου εκτέλεσης καταναλώνεται σε επικοινωνίες, αφενός γιατί λόγω ταχύτερης σύγκλιση το computation time καταλήγει να είναι μικρό, αφετέρου γιατί σε κάθε iteration όλες οι διεργασίες θα πρέπει να κάνουν Allreduce για τον συνολικό έλεγχο σύγκλισης (κάτι που αποτελεί σημαντικό overhead επικοινωνίας συγκριτικά με το μειωμένο computation time). Μαλιστα, το τμήμα χρήσιμου computation time κυμαίνεται μεταξύ 18% και 28% του συνολικού χρόνου για τις διάφορες μεθόδους.

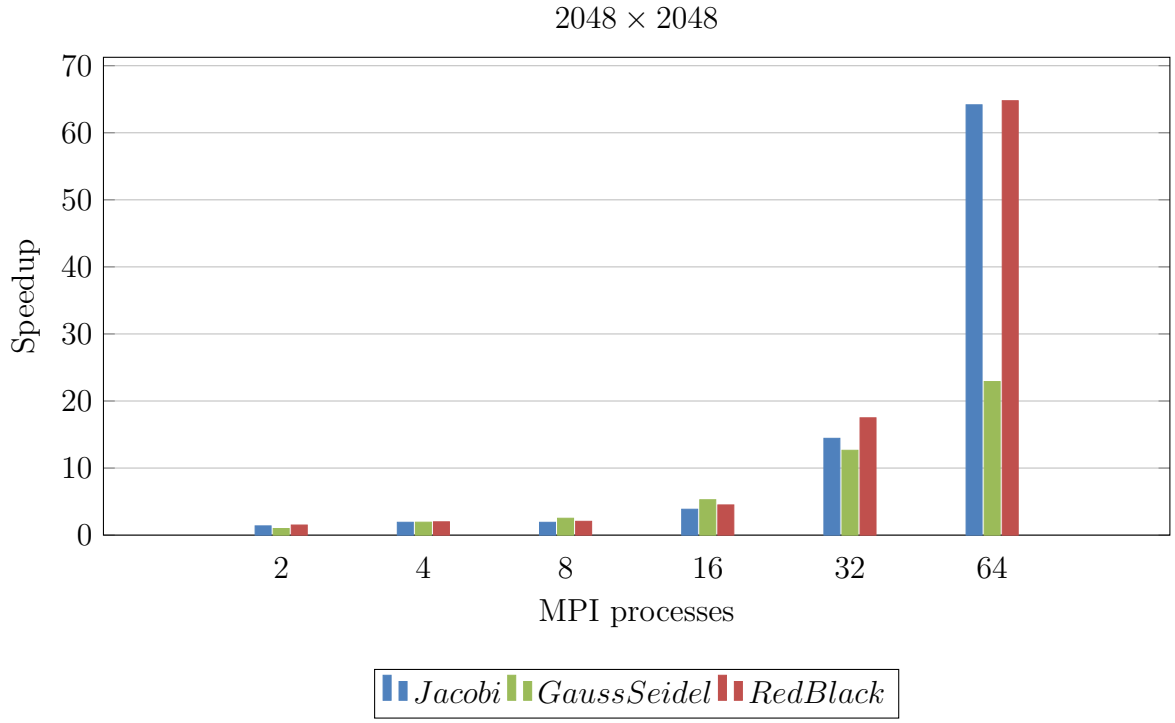


Εικόνα 27: Κατανομή χρόνου εκτέλεσης του heat-transfer αλγορίθμου με χρήση διαφορετικών μεθόδων υλοποίησης (με χρήση του ελέγχου σύγκλισης)

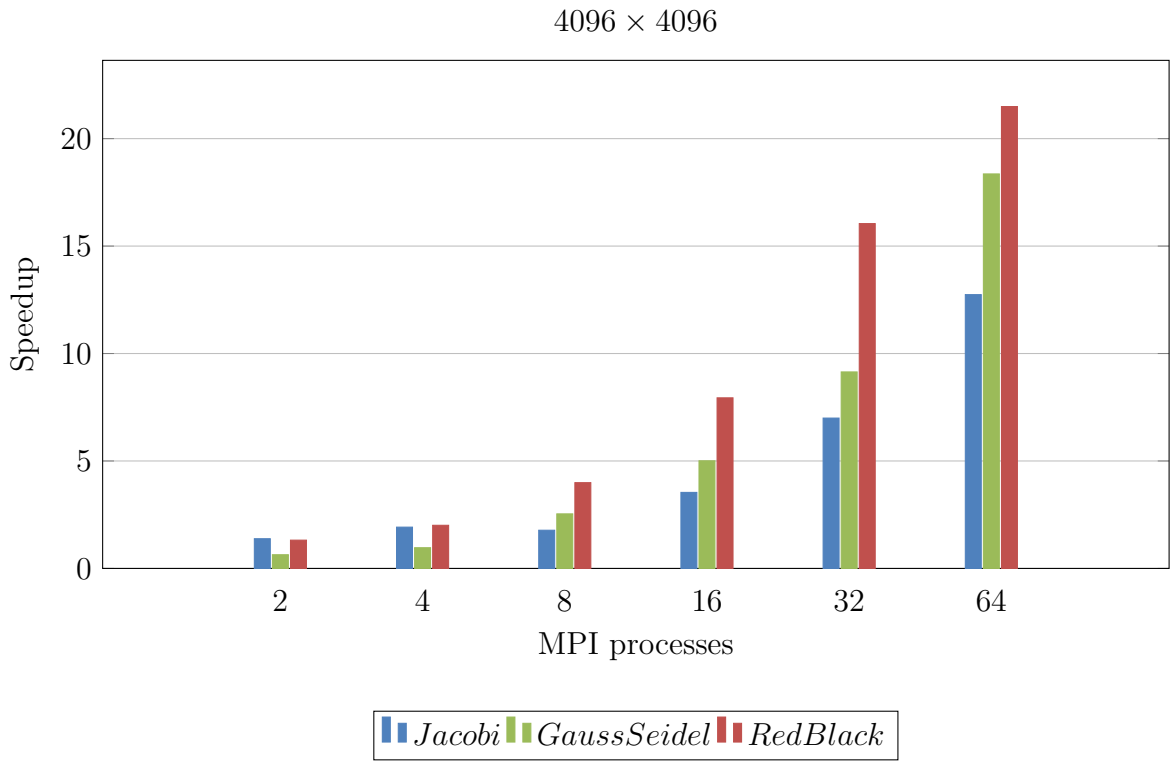
8.3.2 Χωρίς χρήση ελέγχου σύγκλισης

Στη συνέχεια, απενεργοποιούμε τον έλεγχο σύγκλισης και κάνουμε μετρήσεις για 3 διαφορετικά μεγέθη υπολογιστικού χωρίου και διαφορετικό πλήθος MPI διεργασιών. Υπολογίζουμε και σχεδιάζουμε το speedup καθεμιάς από τις μεθόδους συγκριτικά με την σειριακή υλοποίηση (με 1 MPI διεργασία) [εικόνα 28] και στη συνέχεια υπολογίζουμε και απεικονίζουμε τον χρόνο εκτέλεσης (Total και computation) για 8, 16, 32 και 64 MPI διεργασίες [εικόνα 29]. Εν γένει, το γεγονός ότι σε αυτό το σενάριο δεν γίνεται έλεγχος σύγκλισης με συνέπεια να μην αξιοποιείται το δυνατό σημείο των Gauss-Seidel, Red Black μεθόδων, η μέθοδος Jacobi καταλήγει να είναι στις περισσότερες περιπτώσεις καλύτερη (είτε ως προς το speedup είτε ως προς τον απόλυτο χρόνο εκτέλεσης). Οι άλλες δύο μέθοδοι προσθέτουν overheads επικοινωνίας τα οποία τελικά δεν μετουσιώνονται σε πλεονέκτημα ταχύτερης σύγκλισης (και τελικά ταχύτερης εκτέλεσης), αφού αυτό το setting δεν χρησιμοποιείται εδώ. Επιπλέον, παρατηρούμε ότι ο Gauss-Seidel για μικρό μέγεθος grid (2048×2048), επιτυγχάνει το χειρότερο scaling σε αντίθεση με τις άλλες δύο μεθόδους οι οποίες ειδικά για μεγάλο αριθμό από MPI processes επιτυγχάνουν ιδανικό speedup. Αυτό οφείλεται στο γεγονός ότι ο Gauss Seidel είναι "ασύγχρονος" μεταξύ διαφορετικών διεργασιών. Υπάρχουν διεργασίες που προχωρούν στον υπολογιστικό πυρήνα και προοδεύουν ενώ άλλες αναμένουν να τους σταλλούν πληροφορίες και δεν εισέρχονται ακόμη στον υπολογιστικό πυρήνα. Όταν οι διεργασίες που εισέρχονται αμέσως στον υπολογιστικό τους πυρήνα ολοκληρώσουν την εργασία τους, υποχρεούνται (αφότου κάνουν send κατάλληλες πληροφορίες) να αναμένουν άλλες διεργασίες οι οποίες τώρα θα μπουν στον υπολογιστικό τους πυρήνα (καθώς βρίσκονταν υπό αναμονή για receive πληροφοριών). Με λίγα λόγια υπάρχουν διεργασίες δύο ταχυτήτων. Αυτές που εισέρχονται αμέσως στο υπολογιστικό τους χωρίο προοδεύοντας και αυτές που πρέπει να περιμένουν, κάνοντας και τις υπόλοιπες να καθυστερούν.

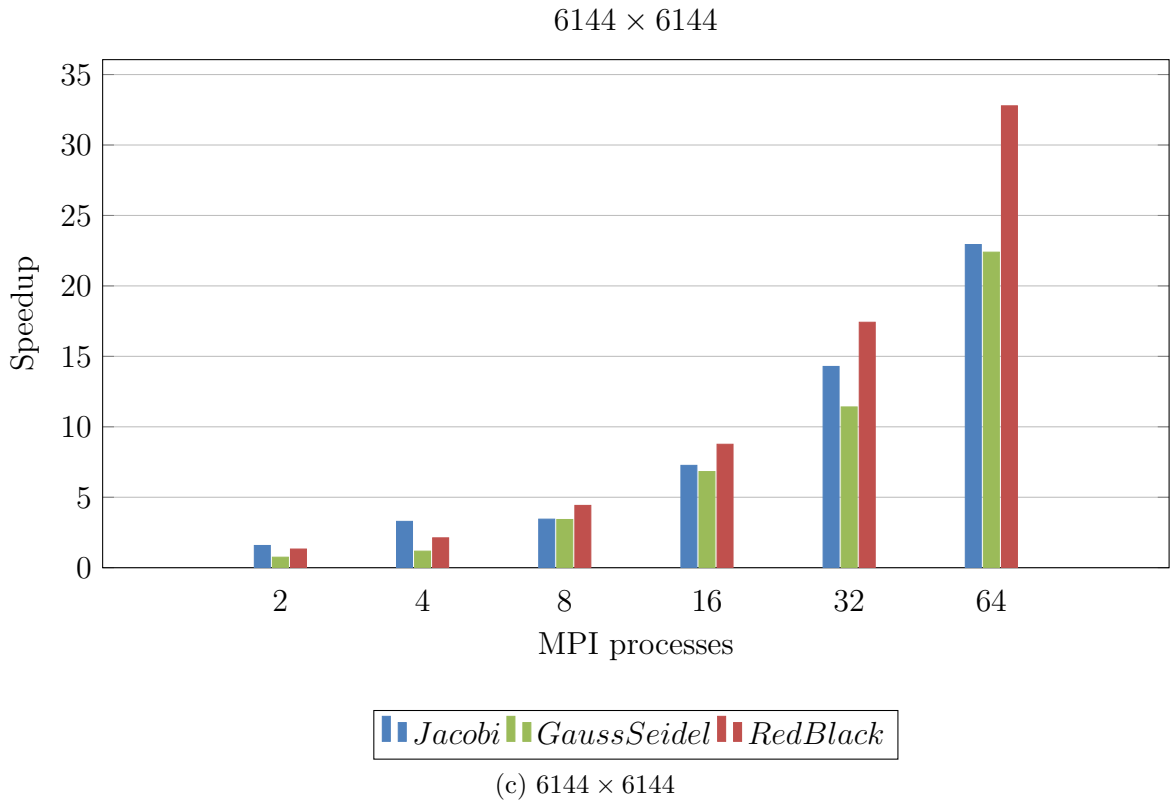
Αυτό φαίνεται και στο διάγραμμα χρόνων εκτέλεσης στο οποίο είναι εμφανές ότι η Gauss Seidel καταναλώνει σημαντικό μερίδιο χρόνου στις αναμονές για επικοινωνία μεταξύ των διεργασιών (σχεδόν το ήμισυ του χρόνου εκτέλεσης αφορά επικοινωνίες).Σημαντικό εδώ είναι να σημειωθεί ότι το scaling του Gauss-Seidel παραμένει σχεδόν ίδιο για όλα τα μεγέθη υπολογιστικού grid. Αντιθέτως, ειδικά για μεγάλο αριθμό από MPI processes οι άλλες δύο μέθοδοι παρουσιάζουν αρκετα χειρότερο scaling. Για να εξηγηθεί αυτό το φαινόμενο θα πρέπει να αναφέρουμε ότι υπάρχουν αρκετές περιπτώσεις στην εκτέλεση του αλγορίθμου στη σειρά μηχανημάτων parlab κατά τις οποίες διαφορετικές MPI διεργασίες ανατίθενται στο ίδιο clone, διαμοιραζόμενες έτσι τους πόρους του. Κατά αυτόν τον τρόπο, για μεγάλο υπολογιστικό χωρίο ενδέχεται να υπάρξουν συγκρούσεις και contention σε αυτούς τους πόρους εάν πολλές MPI διεργασίες τους διεκδικήσουν ταυτόχρονα. Αυτή η ταυτόχρονη διεκδίκηση συμβαίνει πολύ περισσότερο στην Jacobi και Red Black μέθοδο συγκριτικά με την Gauss-Seidel. Στις δύο πρώτες μεθόδους, οι υπολογισμοί γίνονται εξ ολοκλήρου είτε βάσει της προηγούμενης είτε βάσει της τρέχουσας χρονικής στιγμής. Ως εκ τούτου, οι διάφορες διεργασίες εμφανίζουν περισσότερο ταυτοχρονισμό στην εκτέλεσή τους. Αντιθέτως, στην Gauss Seidel, το φαινόμενο των διεργασιών δύο ταχυτήτων που περιγράφηκε προηγουμένως έχει ως αποτέλεσμα να μην εκτελούν όλες οι διεργασίες ταυτόχρονα τα ίδια στάδια και έτσι να μην διεκδικούν ταυτόχρονα τους ίδιους πόρους. Τέλος, αναφέρουμε ότι τα διαγράμματα χρόνου εκτέλεσης για διάφορα μεγέθη grid παρουσιάζουν παρόμοια ποιοτική συμπεριφορά μεταξύ των διαφόρων μεθόδων, ενώ αξιοσημείωτο είναι ότι οι Jacobi και Red Black καταναλώνουν το μεγαλύτερο μέρος του χρόνου εκτέλεσής τους σε χρήσιμο computation, ενώ το communication time είναι σχτικά αμελητέο.



(a) 2048 × 2048

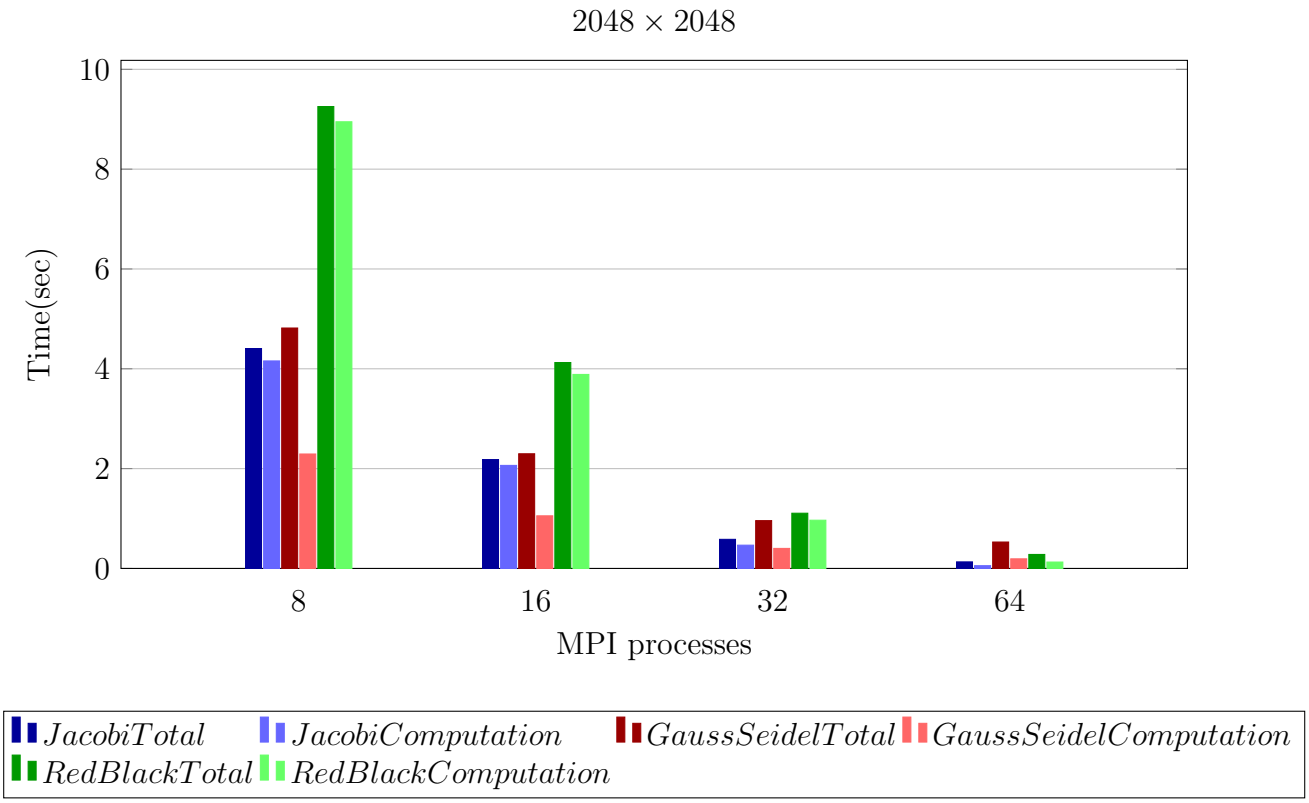


(b) 4096 × 4096

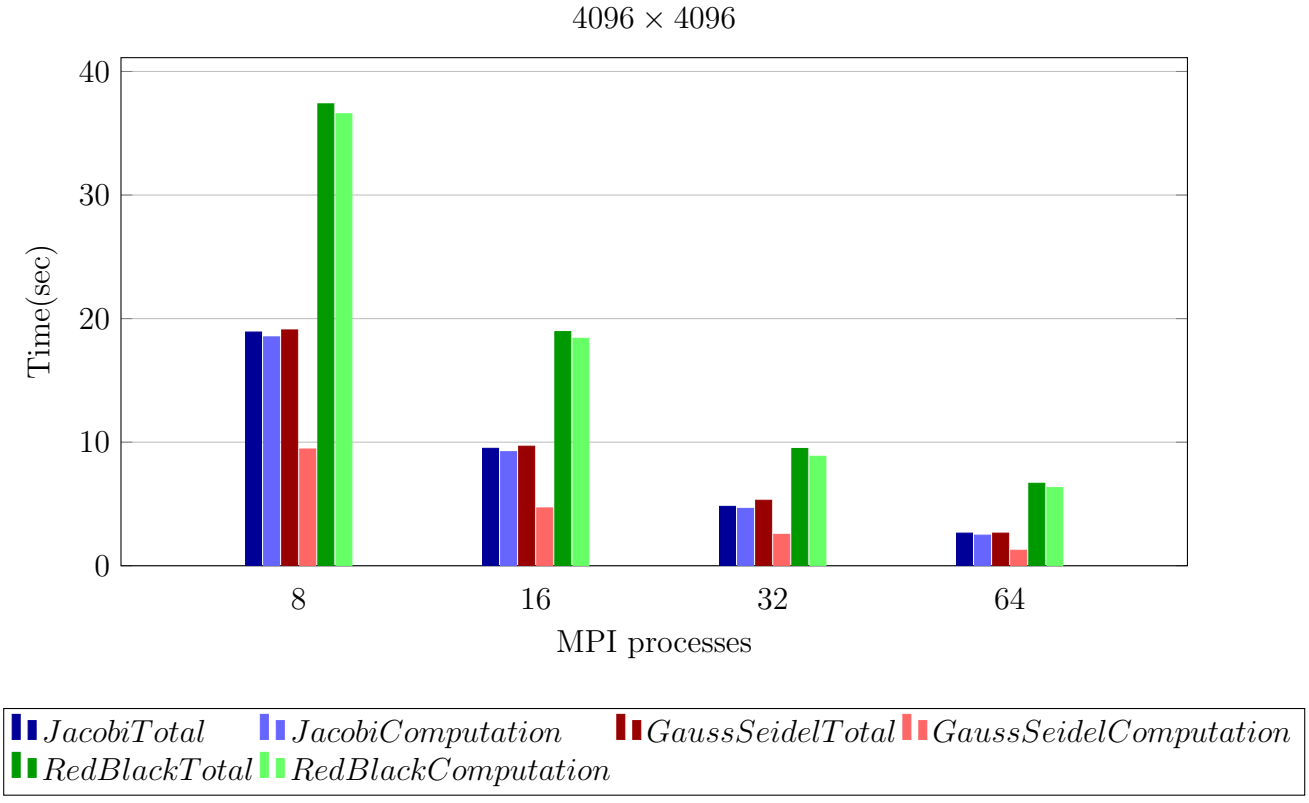


(c) 6144 × 6144

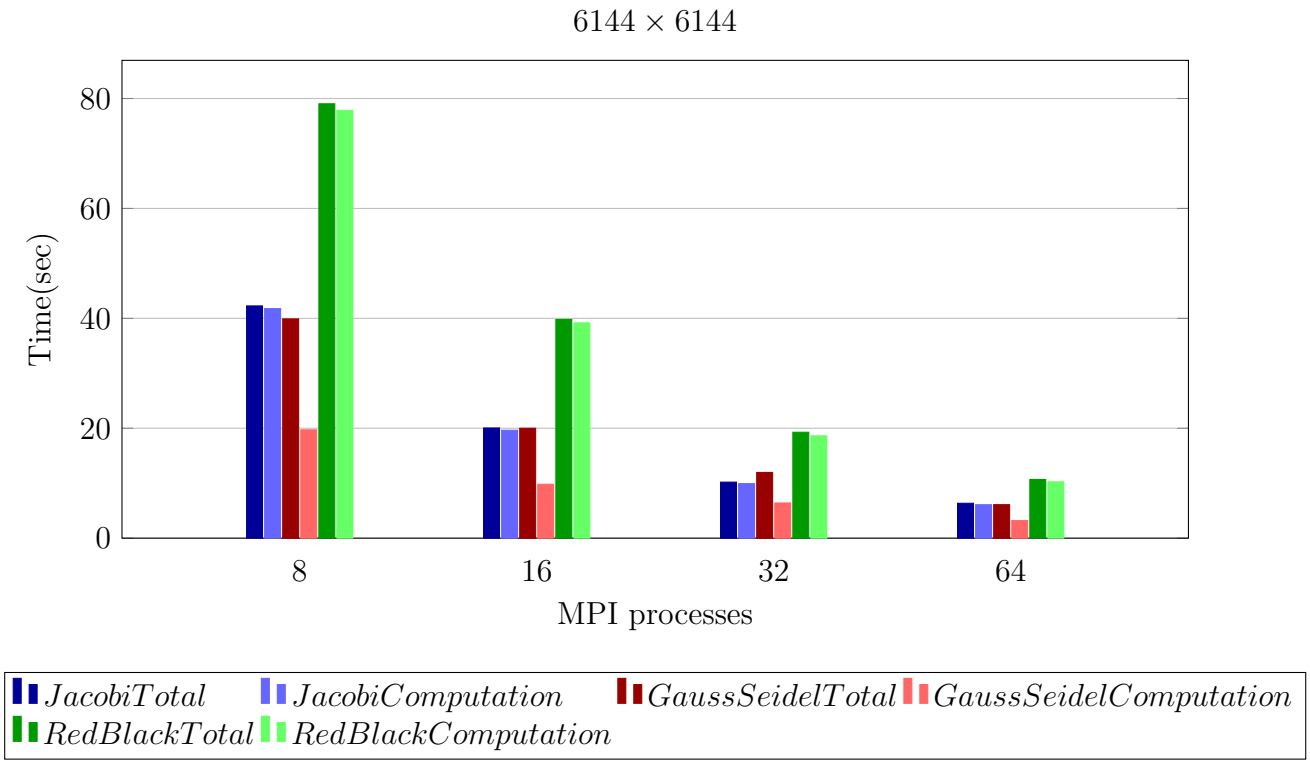
Εικόνα 28: Χρόνοι εκτέλεσης σε GPU και CPU και κόστη μεταφοράς δεδομένων μεταξύ τους για όλες τις υλοποιήσεις



(a) 2048 × 2048



(b) 4096 × 4096



(c) 6144 × 6144

Εικόνα 29: Total και computation χρόνοι εκτέλεσης του αλγορίθμου heat-transfer με χρήση 8, 16, 32 και 64 MPI διεργασιών για διάφορα μεγέθη υπολογιστικού χωρίου