ΤΜΉΜΑ ΗΛΕΚΤΡΟΛΌΓΩΝ ΜΗΧΑΝΙΚΏΝ ΚΑΙ ΜΗΧΑΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΏΝ

ΠΑΝΕΠΙΣΤΉΜΙΟ ΘΕΣΣΑΛΊΑΣ

CE421 ΣΥΣΤΉΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΎ ΥΨΗΛΏΝ ΕΠΙΔΌΣΕΩΝ

Πουρναρόπουλος Φοίβος, ΑΕΜ: 2614

Ηλιάδης Γρηγόρης, ΑΕΜ: 2522

Στη συγκεκριμένη εργασία καλούμαστε να παραλληλοποιήσουμε με τη χρήση της OpenMP τον κώδικα που υλοποιεί τον αλγόριθμο k-means για clustering. Έπειτα, αξιολογούμε πειραματικά την επίδοση της εφαρμογής σε σύστημα με πολυπύρηνους επεξεργαστές και καταγράφουμε τα αποτελέσματα για διαφορετικές τιμές threads (1,2,4,8,16,32 και 64).

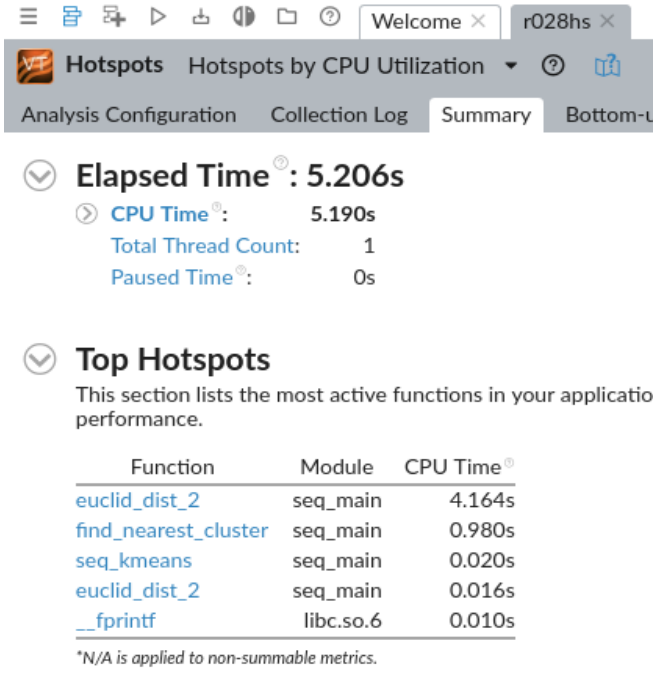
- ***Flags Intel C compiler: -O3***

Εκτελέσαμε τον ακολουθιακό κώδικα για 12 φορές με αρχείο εισόδου το texture17695.bin(2000 clusters και default threshold) και κρατήσαμε τις 10 (αφαιρώντας το max και min χρόνο εκτέλεσης). Η ίδια προσέγγιση ακολουθείται και για όλα τα υπόλοιπα πειράματα της εργασίας.

Μέσος χρόνος: 5.747260 sec

Τυπική απόκλιση: 0.079763 sec

Εφαρμόζοντας profiling μέσω του vtune παρατηρήσαμε ότι τα loops που έχουν το μεγαλύτερο ποσοστό του CPU time είναι στις συναρτήσεις euclid_dist και find_nearest_cluster, όπως φαίνεται και από το στιγμιότυπο που ακολουθεί.



Εικόνα 1: Profiling μέσω vtune

Για το λόγο αυτό δοκιμάσαμε να παραλληλοποιήσουμε αρχικά το loop της συνάρτησης euclid_dist_2, προστατεύοντας τη μεταβλητή ans, διότι γράφεται από όλα τα threads.

```
31  float euclid_dist_2(int    numdims,  /* no. dimensions */
32                      float *coord1,   /* [numdims] */
33                      float *coord2)   /* [numdims] */
34  {
35      int i;
36      float ans=0.0;
37      #pragma omp parallel for reduction(+:ans)
38      for (i=0; i<numdims; i++)
39          ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
40
41      return(ans);
42  }
43
```

Εικόνα 2: Παραλληλοποίηση loop euclid_dist_2

Όμως, ο χρόνος εκτέλεσης μετά από την συγκεκριμένη τροποποίηση αυξάνεται σημαντικά σε σχέση με τον αρχικό, πράγμα που οφείλεται στο γεγονός ότι το κόστος για τη δημιουργία και καταστροφή και τον συγχρονισμό των threads είναι πολύ μεγαλύτερο σε σχέση με το workload που αναλαμβάνει το κάθε thread. Επίσης, σε αυτό συμβάλλει και το μεγάλο πλήθος των επαναλήψεων του αλγορίθμου. Πιο συγκεκριμένα, η euclid_dist_2 καλείται συνολικά loops * numObjs * numClusters φορές, όπου loops είναι ο αριθμός επαναλήψεων μέχρι να συγκλίνει ο αλγόριθμος k-means, numObjs το πλήθος των σημείων του αρχείου εισόδου και numClusters ο αριθμός των clusters που δίνεται ως όρισμα του προγράμματος. Κατά συνέπεια, για κάθε βήμα του αλγορίθμου προστίθεται σημαντικό overhead για τη δημιουργία/καταστροφή/συγχρονισμό των threads, ενώ στην πραγματικότητα το loop που παραλληλοποιούμε εκτελείται για numdims φορές. Επομένως, δε συμφέρει να παραλληλοποιήσουμε το συγκεκριμένο loop.

Με την ίδια λογική, δοκιμάσαμε να παραλληλοποιήσουμε το loop της συνάρτησης find_nearest_cluster, όπως φαίνεται στην Εικόνα 3.

```
46  int find_nearest_cluster(int     numClusters, /* no. clusters */
47                           int     numCoords,   /* no. coordinates */
48                           float  *object,      /* [numCoords] */
49                           float **clusters)    /* [numClusters][numCoords] */
50  {
51      int   index, i;
52      float dist, min_dist;
53
54      /* find the cluster id that has min distance to object */
55      index    = 0;
56      min_dist = euclid_dist_2(numCoords, object, clusters[0]);
57      #pragma omp parallel for private(dist)
58      for (i=1; i<numClusters; i++) {
59          dist = euclid_dist_2(numCoords, object, clusters[i]);
60          /* no need square root */
61          #pragma omp critical {
62              if (dist < min_dist) { /* find the min and its array index */
63                  min_dist = dist;
64                  index    = i;
65              }
66          }
67      }
68      return(index);
69  }
```

Εικόνα 3: Παραλληλοποίηση loop find_nearest_cluster

Όπως και στην προηγούμενη δοκιμή, η συγκεκριμένη παραλληλοποίηση εξακολουθεί να αυξάνει δραματικά το συνολικό overhead του προγράμματος. Αυτό οφείλεται στο γεγονός ότι το workload για κάθε thread εξακολουθεί να είναι μικρό σε σχέση με το overhead δημιουργίας/καταστροφής/συγχρονισμού τους. Οι γραμμές 62-65 αποτελούν κρίσιμο τμήμα, οπότε χρησιμοποιήσαμε επιπλέον κώδικα για τον συγχρονισμό των νημάτων έτσι ώστε να αποφύγουμε τυχόν race conditions, κάτι που επιβαρύνει επιπλέον την απόδοση, δεδομένου ότι μόνο 1 thread μπορεί να βρίσκεται στο κρίσιμο τμήμα κάθε στιγμή. Σε συνδυασμό και με το overhead δημιουργίας/ καταστροφής/συγχρονισμού των threads (για loops*numObjs φορές) , κάνει το πρόγραμμα πολύ πιο αργό σε σχέση με το ακολουθιακό, γι' αυτό και δεν συμφέρει ούτε η παραπάνω παραλληλοποίηση.

Με στόχο να μοιράσουμε μεγαλύτερο workload στα threads, ώστε να ελαχιστοποιήσουμε το overhead δημιουργίας και καταστροφής τους, πειραματιστήκαμε με την παραλληλοποίηση των loops της συνάρτησης seq_kmeans, δεδομένου ότι το πλήθος των σημείων του αρχείου texture17695.bin είναι πολύ μεγαλύτερο σε σχέση με τα threads που μπορούμε να εκμεταλλευτούμε ταυτόχρονα (1,2,…,64).

Αρχικά, παραλληλοποιήσαμε το for loop στη συνάρτηση seq_kmeans στη γραμμή 107 της παρακάτω εικόνας. Οι μεταβλητές index και j δηλώνονται ως private, καθώς πρέπει να είναι ανεξάρτητες για το κάθε thread. Επιπλέον, το σενάριο ταυτόχρονης εγγραφής της κοινής μεταβλητής delta αποφεύγεται μέσω του reduction clause. Εφόσον η εκτέλεση του for loop γίνεται παράλληλα, υπάρχει κίνδυνος 2 ή και περισσότερα threads να έχουν ίδια τιμή του index μετά από κλήση της find_nearest_cluster, γεγονός που μας αναγκάζει να εκτελέσουμε ατομικά τις εντολές των γραμμών 121 και 124(με χρήση του #pragma omp atomic), αυξάνοντας το προστιθέμενο overhead για τον συγχρονισμό των threads. Παρόλα αυτά, με την συγκεκριμένη παραλληλοποίηση παρατηρούμε σημαντική βελτίωση στο χρόνο εκτέλεσης του προγράμματος που επιτυγχάνει **speedup x23.64** φορές με χρήση 64 νημάτων (όπου επιτυγχάνεται η καλύτερη επίδοση) σε σχέση με τον ακολουθιακό κώδικα .

Τέλος, η πολιτική scheduling που χρησιμοποιείται στις επόμενες παραλληλοποιήσεις είναι η dynamic , καθώς ήταν αυτή με την μεγαλύτερη βελτίωση στο χρόνο εκτέλεσης του προγράμματος, κάτι που θα εξηγηθεί περεταίρω στη συνέχεια.

```
do {
    delta = 0.0;
    #pragma omp parallel for private(index,j) reduction(+:delta) \
        schedule(dynamic)
    for (i=0; i<numObjs; i++) {
        /* find the array index of nestest cluster center */
        index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                     clusters);

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

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

        /* update new cluster center : sum of objects located within */
        #pragma omp atomic
             newClusterSize[index]++;
        for (j=0; j<numCoords; j++){
            #pragma omp atomic
                 newClusters[index][j] += objects[i][j];
        }
    }
    /* average the sum and replace old cluster center with newClusters */

    for (i=0; i<numClusters; i++) {

        for (j=0; j<numCoords; j++) {

                            if (newClusterSize[i] > 0)
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
                newClusters[i][j] = 0.0;   /* set back to 0 */
            }
            newClusterSize[i] = 0;   /* set back to 0 */
    }
    delta /= numObjs;
} while (delta > threshold && loop++ < 500);
```

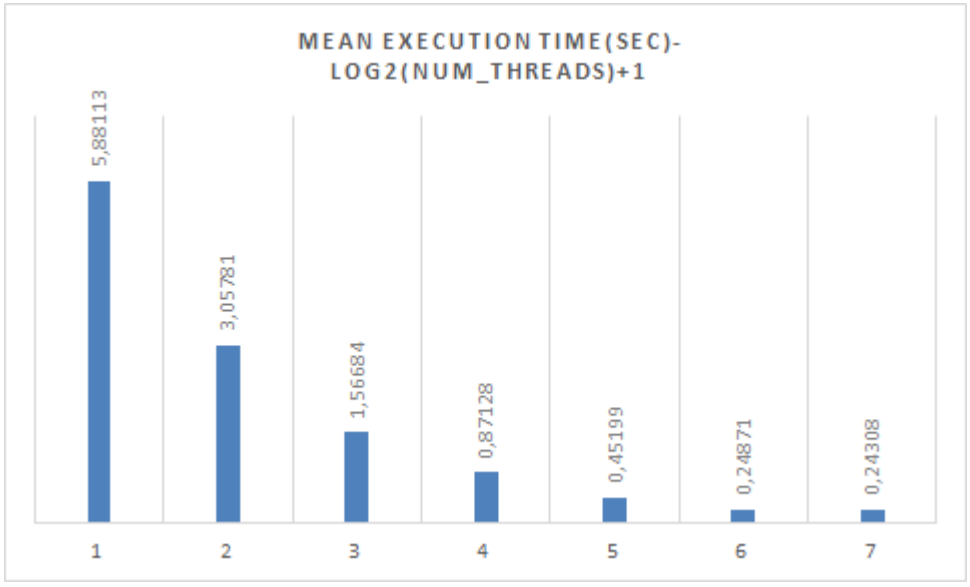
Εικόνα 4: Παραλληλοποίηση 1ου loop της seq_kmeans

Ακολουθούν οι μετρήσεις για τον κώδικα της εικόνας 4

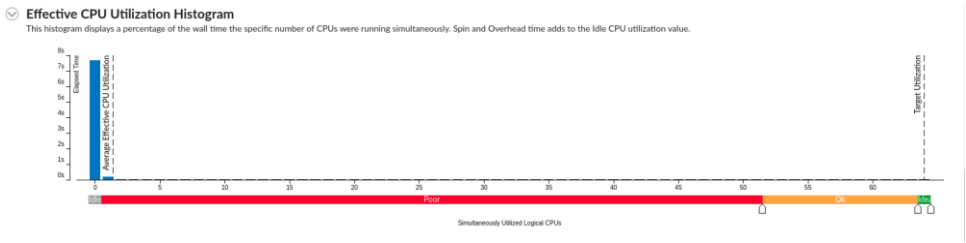
#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.88113	3.05781	1.56684	0.87128	0.45199	0.24871	0.24308
Τυπική απόκλιση (sec)	0.095462	0.065665	0.027284	0.014617	0.002547	0.002342	0.019548

Σχολιασμός αποτελεσμάτων: Παρατηρούμε ότι ο μέσος χρόνος εκτέλεσης σχεδόν υποδιπλασιάζεται όταν διπλασιάζουμε τον αριθμό των νημάτων, εκτός από την περίπτωση των 64 threads (εικόνα 5). Αυτό συμβαίνει διότι όσο αυξάνουμε τα νήματα, επιτυγχάνουμε καλύτερο παραλληλισμό. Όμως, στην περίπτωση των 64 ο χρόνος είναι σχεδόν ίδιος με τον χρόνο για 32 νήματα και ενδεχομένως σε ορισμένες εκτελέσεις να είναι και μεγαλύτερος(λόγω τυπικής απόκλισης). Στην πραγματικότητα, παρόλο που ο βαθμός παραλληλοποίησης αυξάνεται και το workload για το κάθε thread μειώνεται, αυξάνεται ταυτόχρονα το overhead του συγχρονισμού των νημάτων στις ατομικές εντολές και το συνολικό overhead για τη δημιουργία και την καταστροφή τους. Επίσης, όσο περισσότερα νήματα δημιουργούνται, τόσο πιο δύσκολο είναι να δουλεύουν όλα

ταυτόχρονα μεταξύ τους, πράγμα που φαίνεται από το παρακάτω στιγμιότυπο από το profiling μέσω του vtune(όλα τα threads δουλεύουν για ελάχιστο χρόνο ταυτόχρονα).

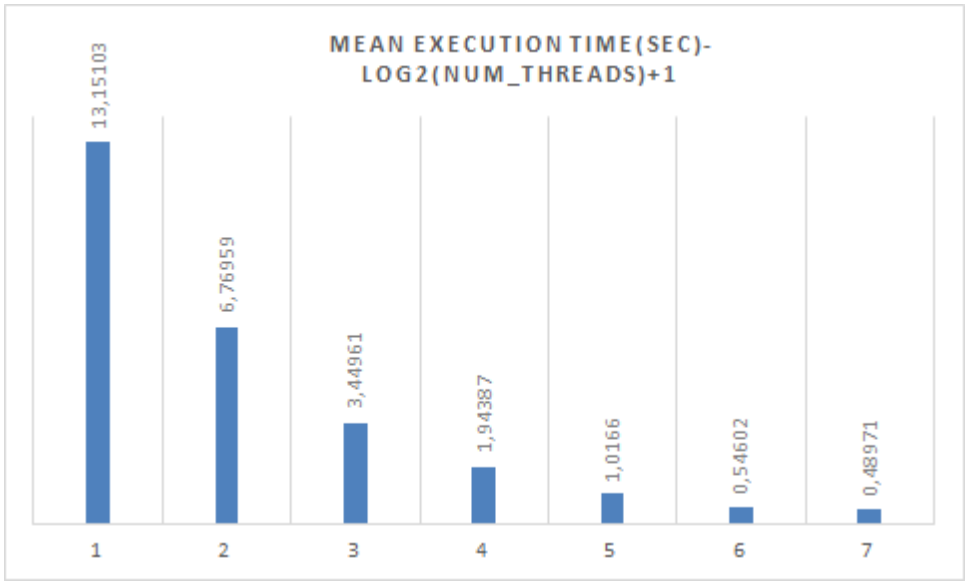


Εικόνα 5: Μέσοι χρόνοι εκτέλεσης για 1, 2, 4, 8, 16, 32, 64 threads.



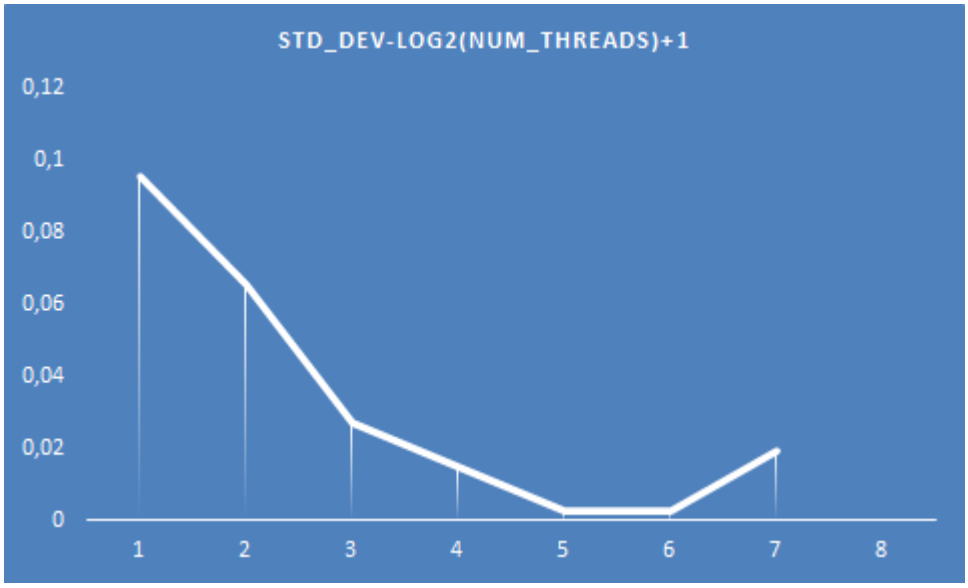
Εικόνα 6: Screenshot vtune. Ο μεγάλος χρόνος που εμφανίζεται να μην χρισιμοποιείται καμία CPU οφείλεται στο ότι τρέχουμε remotely το vtune από το artemis με script που καθορίζει τις μεταβλητές περιβάλλοντος.

Επιπλέον, αν αυξήσουμε των αριθμό των clusters η μειώσουμε το threshold επιτυγχάνουμε καλύτερη κλιμάκωση του workload για 64 threads. Αυτό έχει ως συνέπεια να φαίνεται η βελτίωση στον μέσο χρόνο εκτέλεσης μεταξύ 32 και 64 threads, σε αντίθεση με το προηγούμενο πείραμα όπου οι χρόνοι ήταν παρόμοιοι. Στην παρακάτω εικόνα φαίνονται τα αποτελέσματα της εκτέλεσης για 3000 clusters και threshold = 0.00001 τα οποία επιβεβαιώνουν τα παραπάνω συμπεράσματα (στο προηγούμενο πείραμα είχαμε 2000 clusters και threshold=0.001).



Εικόνα 7: Μέσοι χρόνοι εκτέλεσης για 1,2,4,8,16,32,64 threads, 3000 clusters, threshold=0.00001.

Από τον πίνακα των μετρήσεων συμπεραίνουμε ότι όσο αυξάνουμε τα threads τόσο μειώνεται η τυπική απόκλιση, που είναι λογικό να συμβαίνει καθώς μειώνεται σημαντικά και ο χρόνος εκτέλεσης, εκτός από την περίπτωση των 64 που αυξάνεται σε σχέση με τα 32 (εικόνα 8). Αυτό μπορεί να οφείλεται στο γεγονός ότι όσο αυξάνουμε το πλήθος των νημάτων, τόσο μεγαλύτερο overhead έχουμε για τον συγχρονισμό τους, επομένως σε ένα κακό σενάριο εκτέλεσης που πολλά threads θέλουν να εκτελέσουν τις ίδιες ατομικές εντολές, ο χρόνος εκτέλεσης θα αυξηθεί σημαντικά. Επίσης, η μεγάλη τυπική απόκλιση των 64 threads, μπορεί να οφείλεται και στο γεγονός ότι είναι δύσκολο να επιτύχουμε αποκλειστική χρήση όλων των πόρων του συστήματος, οπότε το πλήθος των cores που χρησιμοποιούνται ταυτόχρονα μπορεί να αλλάξει από μία εκτέλεση σε μία άλλη, καθώς υπάρχει τυχαιότητα στον τρόπο που θα εκτελεστεί ο παράλληλος κώδικας.



Εικόνα 8: Τυπική απόκλιση για 1, 2, 4, 8, 16, 32, 64 threads.

Έπειτα, δοκιμάσαμε να παραλληλοποιήσουμε και τα 2 for loops της do-while, όπως φαίνεται στην εικόνα 9.

```
        delta = 0.0;
        #pragma omp parallel
        {
            #pragma omp for private(index,j) reduction(+:delta) \
            schedule(dynamic)
                for (i=0; i<numObjs; i++) {
                    /* find the array index of nestest cluster center */
                    index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                                 clusters);

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

                        delta += 1.0;

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

                    /* update new cluster center : sum of objects located within */
                    #pragma omp atomic
                        newClusterSize[index]++;
                    for (j=0; j<numCoords; j++){
                        #pragma omp atomic
                            newClusters[index][j] += objects[i][j];
                    }
                }
            //#pragma omp barrier
            /* average the sum and replace old cluster center with newClusters */
            #pragma omp for private(j)\
            schedule(dynamic) nowait
            for (i=0; i<numClusters; i++) {
                for (j=0; j<numCoords; j++) {
                    if (newClusterSize[i] > 0)
                        clusters[i][j] = newClusters[i][j] / newClusterSize[i];
                        newClusters[i][j] = 0.0;   /* set back to 0 */
                    }
                    newClusterSize[i] = 0;   /* set back to 0 */
                }
            }
            delta /= numObjs;
        } while (delta > threshold &&  loop++ < 500);
```

Εικόνα 9: Παραλληλοποίηση των 2 loops της seq_kmeans

Πιο συγκεκριμένα, ο μετρητής j γίνεται private, καθώς το κάθε thread πρέπει να έχει δικό του αντίγραφο. Επίσης, προσθέσαμε το nowait clause με σκοπό να αποφύγουμε χρήση implicit barrier στο τέλος της 2ης for και τα threads που τερματίζουν να μη χρειάζεται να περιμένουν άσκοπα τα υπόλοιπα. Δεν είναι εφικτό να βάλουμε nowait και στην πρώτη for, καθώς υπάρχει εξάρτηση δεδομένων μεταξύ της 1ης και της 2ης (στους πίνακες clusters, newClusters και newClusterSize) , επομένως πρέπει να έχει ολοκληρωθεί η πρώτη για να μπορέσουμε να εκτελέσουμε την δεύτερη.

Ακολουθούν οι μετρήσεις για τον κώδικα της εικόνας 9:

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.968380	3.090480	1.614010	0.895400	0.456490	0.254840	0.262480
Τυπική απόκλιση (sec)	0.098464	0.072648	0.031255	0.012244	0.009686	0.001773	0.021324

Σχολιασμός αποτελεσμάτων: Παρατηρούμε παρόμοια συμπεριφορά σε σχέση με την προηγούμενη παραλληλοποίηση όσον αφορά τους μέσους χρόνους και τις τυπικές αποκλήσεις, και επιπλέον μία μικρή αύξηση του μέσου χρόνου εκτέλεσης για όλους τους πιθανούς αριθμούς διαθέσιμων threads. Αυτό συμβαίνει διότι το workload που αντιστοιχεί σε κάθε νήμα της 2ης for είναι συγκρίσιμο με το overhead της παραλληλοποίησης (δημιουργία/καταστροφή και συγχρονισμός των threads), γεγονός που επιβαρύνει την απόδοση της εφαρμογής. Μέγιστο **speedup** σε σχέση με τον ακολουθιακό κώδικα: **x22.5**(για 32 threads)

Μία ακόμη πιθανή παραλληλοποίηση είναι να δώσουμε ακόμα μεγαλύτερο workload στα threads, ξεκινώντας το παράλληλο τμήμα έξω από την do-while, όπως φαίνεται στην εικόνα 10.

```
106     #pragma omp parallel
107     {
108         do{
109             delta = 0.0;
110 
111             #pragma omp for private(index,j) reduction(+:delta) \
112             schedule(dynamic)
113             for (i=0; i<numObjs; i++) {
114                 /* find the array index of nestest cluster center */
115                 index = find_nearest_cluster(numClusters, numCoords, objects[i],clusters);
116                 /* if membership changes, increase delta by 1 */
117                 if (membership[i] != index)
118                     delta += 1.0;
119                 /* assign the membership to object i */
120                 membership[i] = index;
121                 /* update new cluster center : sum of objects located within */
122                 #pragma omp atomic
123                     newClusterSize[index]++;
124                 for (j=0; j<numCoords; j++){
125                     #pragma omp atomic
126                         newClusters[index][j] += objects[i][j];
127                 }
128             }
129             /* average the sum and replace old cluster center with newClusters */
130             #pragma omp for private(j)\
131             schedule(dynamic) nowait
132             for (i=0; i<numClusters; i++) {
133                 for (j=0; j<numCoords; j++) {
134                     if (newClusterSize[i] > 0)
135                         clusters[i][j] = newClusters[i][j] / newClusterSize[i];
136                         newClusters[i][j] = 0.0;   /* set back to 0 */
137                     }
138                     newClusterSize[i] = 0;   /* set back to 0 */
139             }
140 
141             #pragma omp master
142             {
143                 delta /= numObjs;
144                 if(delta <= threshold || loop++ >= 500){
145                     flag = 1;
146                 }
147             }
148             #pragma omp barrier
149         }while(flag != 1);
150     }
```

Εικόνα 10: Παραλληλοποίηση της do-while

Στην συγκεκριμένη παραλληλοποίηση, δημιουργούμε τα threads έξω από την do-while, με σκοπό να τα δημιουργήσουμε και να τα καταστρέψουμε μόνο μία φορά, με κάποιο επιπλέον κόστος για τον συγχρονισμό τους (flag, #pragma omp barrier, #pragma omp master). Πιο συγκεκριμένα, το master thread εκτελεί ειδικό τμήμα κώδικα έτσι ώστε να δει εάν έχει συγκλίνει ο αλγόριθμος, και έπειτα να αλλάξει το flag από 0 σε 1, ώστε οι workers να βγουν από την do-while σε αυτή την περίπτωση. Επιπλέον, για να εξασφαλίσουμε ότι όλα τα threads εξετάζουν τη σωστή τιμή του flag και ότι το τρέχον βήμα έχει ολοκληρωθεί πριν πάμε στο επόμενο, προστίθεται ένα barrier πριν τον έλεγχο της τιμής του flag. Όπως φαίνεται και από τις παρακάτω μετρήσεις, η συγκεκριμένη έκδοση

του κώδικα φαίνεται λιγότερο αποδοτική σε σχέση με την πρώτη προσέγγιση(παραλληλοποίηση 1ης for μέσα στην do-while). Αυτό πιστεύουμε ότι συμβαίνει διότι στην παραλληλοποίηση της do-while , χρησιμοποιούμε περισσότερους μηχανισμούς για συγχρονισμό των threads, όπως προαναφέρθηκε. Επίσης, πιστεύουμε ότι πιθανόν ο compiler στην παραλληλοποίηση μόνο της 1ης for, αντιλαμβάνεται ότι τα threads επαναχρησιμοποιούνται για πολλές επαναλήψεις, με αποτλέλεσμα στην πραγματικότητα να μην σκοτώνει τα threads που δημιουργούμε σε κάθε επανάληψη της do-while, αλλά αντιθέτως τα κρατάει και τα επαναχρησιμοποιεί. Δεν μπορούμε να αποδείξουμε πειραματικά ότι συμβαίνει η συγκεκριμένη βελτιστοποίηση, αλλά ακόμα και αν όντως συμβαίνει δε μπορούμε να είμαστε σίγουροι ότι θα γίνει εάν χρησιμοποιούσαμε κάποιον άλλο compiler ή άλλη έκδοσή του.

Παρόλα αυτά υπό αυτές τις συνθήκες παράγει τον πιο γρήγορο κώδικα, συνεπώς συνεχίζουμε την υλοποίηση μας με βάση μόνο την παραλληλοποίηση της 1ης for (κώδικας της εικόνας 4).

Ακολουθούν οι μετρήσεις για τον κώδικα της εικόνας 10.

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.71698	3.04742	1.56802	0.87261	0.44921	0.25072	0.25895
Τυπική απόκλιση (sec)	0.129507	0.097543	0.036871	0.016323	0.005002	0.002541	0.018188

Σχολιασμός αποτελεσμάτων: Παρόμοια συμπεριφορά παρατηρείται και στις παραπάνω μετρήσεις, όπου ο χρόνος σχεδόν υποδιπλασιάζεται όταν διπλασιάζουμε τον αριθμό των νημάτων, εκτός από την περίπτωση των 64 όπου παρατηρείται και μία μικρή αύξηση. Μέγιστο **speedup** σε σχέση με τον ακολουθιακό κώδικα: **x22.92** (32 threads).

Έπειτα, έχοντας κρατήσει τον κώδικα με την καλύτερη επίδοση, κάναμε κάποιες περαιτέρω παραλληλοποιήσεις στις αρχικοποιήσεις των πινάκων της seq_kmeans. Πιο συγκεκριμένα ομαδοποιούμε την αρχικοποίηση για τους πίνακες membership και newClusters και παραλληλοποιούμε αυτά τα δύο loops. Μάλιστα, αφού δεν υπάρχει εξάρτηση δεδομένων μεταξύ τους, στο πρώτο loop προσθέτουμε το nowait clause. Το ίδιο κάνουμε και για το δεύτερο loop ώστε να αποφύγουμε το implicit barrier της δεύτερης παράλληλης for. Στην παρακάτω εικόνα 11 φαίνεται ο αλλαγμένος κώδικας της αρχικοποίησης.

```
 /* initialize membership[] */
#pragma omp parallel
{
    #pragma omp for nowait schedule(dynamic)
            for (i=0; i<numObjs; i++)
                    membership[i] = -1;
    #pragma omp for nowait schedule(dynamic)
            for (i=1; i<numClusters; i++)
                    newClusters[i] = newClusters[0] +(i * numCoords);
}
```

Εικόνα 11: Παραλληλοποίηση των 2 loops αρχικοποίησης.

Πίνακας με μετρήσεις για τον κώδικα της εικόνας 11.

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.462790	2.899940	1.529070	0.854310	0.434990	0.246780	0.258860
Τυπική απόκλιση (sec)	0.004432	0.050934	0.024814	0.016211	0.007633	0.002511	0.023132

Πίνακας με μετρήσεις του κώδικα της εικόνας 4(παραλληλοποίηση 1ης for με threshold 0.00001 και 2000 clusters)

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	8.229930	4.261950	2.178290	1.224740	0.652760	0.354530	0.349180
Τυπική απόκλιση (sec)	0.097430	0.091463	0.041098	0.017263	0.009676	0.004286	0.032288

Πίνακας με μετρήσεις του κώδικα της εικόνας 11 (παραλληλοποίηση 1ης for με threshold 0.00001 και 2000 clusters)

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	8.211590	4.288570	2.202450	1.240540	0.655090	0.358010	0.338380
Τυπική απόκλιση (sec)	0.087570	0.120518	0.059038	0.030933	0.007683	0.002953	0.020302

Σχολιασμός αποτελεσμάτων: Από τις παραπάνω μετρήσεις, βγάζουμε το συμπέρασμα ότι η παραλληλοποίηση της αρχικοποίησης δεν βελτιώνει το μέσο χρόνο εκτέλεσης στο σενάριο του threshold = 0.001 για 32 και 64 νήματα. Σε όλες τις υπόλοιπες τιμές νημάτων η παραλληλοποιημένη αρχικοποίηση είναι πιο γρήγορη. Επιπλέον, αυξάνοντας το workload (μικραίνοντας το threshold), παρατηρούμε ότι είναι πιο αργό για τιμές νημάτων 2,…, 32 και πιο γρήγορο για τις τιμες 1 και 64. Κατά συνέπεια, προτιμούμε να κρατήσουμε την έκδοση κώδικα που εκμεταλλεύεται καλύτερα τα 64 νήματα και με το μεγαλύτερο workload. Για το λόγο αυτό, κρατάμε την παραλληλοποίηση των αρχικοποιήσεων. Επίσης, αν και θεωρητικά το workload είναι παρόμοιο για το κάθε νήμα στην αρχικοποίηση, στην πράξη μέσα από τα πειράματα φαίνεται ότι η πολιτική dynamic είναι πιο γρήγορη.

Σε αυτό το στάδιο, έχοντας τελειώσει με τις παραλληλοποιήσεις των loops, επιχειρήσαμε να κάνουμε κάποιες γενικές βελτιστοποιήσεις στον κώδικα. Πιο συγκεκριμένα, δοκιμάσαμε να εφαρμόσουμε padding στον πίνακα newClusters της seq_kmeans, ώστε να αποφύγουμε το

φαινόμενο του false sharing. Το φαινόμενο αυτό συμβαίνει όταν threads χρησιμοποιούν διαφορετικές θέσεις μνήμης που όμως τυχαίνει να είναι στο ίδιο cache line. Αν κάποια από αυτές αλλάξει από ένα thread ολόκληρο το cache line γίνεται modified με αποτέλεσμα να έχουμε άσκοπα cache misses για τα υπόλοιπα threads που προσπελαύνουν το cache line αυτό . Για το λόγο αυτό, κάναμε padding έτσι ώστε να μην υπάρχουν ποτέ στοιχεία 2 διαφορετικών γραμμών του πίνακα newClusters στο ίδιο cache line. Με αυτό τον τρόπο, τα threads μπορούν να διαβάζουν/γράφουν διαφορετικές γραμμές του πίνακα ανεξάρτητα μεταξύ τους(χωρίς cache misses λόγω false sharing). Δεν επιλέξαμε να εφαρμόσουμε την ίδια τεχνική για τους πίνακες newClusterSize και membership γιατί έχουν μικρό μέγεθος γραμμής (1 int) σε σχέση με το chach line size. Οπότε το μέγεθος του padding θα ήταν μεγάλο και θα χαλούσε την τοπικότητα των δεδομένων στην μνήμη με αποτέλεσμα να έχει αρνητική επίδραση στην επίδοση. Στην παρακάτω εικόνα 12 φαίνεται ο κώδικας για το padding.

```
    //find in how many cache lines fits one line o newClusters array
    diff = (double)(numCoords*sizeof(float))/(double)cache_line;
    for(padd = 0; (double)padd < diff; padd++);

newClusters    = (float**) malloc(numClusters * sizeof(float*));
assert(newClusters != NULL);

    newClusters[0] = (float*)  calloc(numClusters * padd * cache_line, sizeof(char));
    assert(newClusters[0] != NULL);

    for (i=1; i<numClusters; i++)
    newClusters[i] = newClusters[i-1] + ((padd * cache_line * sizeof(char))/sizeof(float));
```

Εικόνα 12: Padding στον πίνακα newClusters.

Πριν την δέσμευση του πίνακα newClustes υπολογίζεται το μέγεθος του padding. Αρχικά με την εντολή sysconf βρίσκουμε το μέγεθες της cache line. Δυστυχώς η συγκεκριμένη εντολή είναι portable μόνο για επεξεργαστές x86, αλλά την χρησιμοποιούμε εφόσον τα πειράματα εκτελούνται σε τέτοια συστήματα. Στην συνέχεια υπολογίζουμε σε πόσα cache lines χωράει μία γραμμή του πίνακα newCluster και δεσμεύουμε την κατάλληλη εξτρά μνήμη.

Πίνακας με μετρήσεις για τον κώδικα της εικόνας 12.

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.629740	3.000990	1.545270	0.867740	0.445280	0.244090	0.254570
Τυπική απόκλιση (sec)	0.089763	0.092945	0.043992	0.016952	0.007078	0.002338	0.022992

Σχολιασμός αποτελεσμάτων: Οι χρόνοι του padding είναι παρόμοιοι με αυτούς της παραλληλοποίησης της 1ης for στην do-while, παρόλα αυτά για τα 64 νήματα φαίνεται να έχει μεγαλύτερο μέσο χρόνο εκτέλεσης. Αυτό πιστεύουμε ότι συμβαίνει διότι με το padding χαλάμε την τοπικότητα των δεδομένων στη μνήμη, εφόσον προσθέτουμε κενές θέσεις στον πίνακα newClusters, ενώ ταυτόχρονα αυξάνουμε την πιθανότητα να έχουμε προσπέλαση στην ίδια θέση μνήμης λόγω των περισσότερων threads. Αυτό το φαινόμενο είναι διαφορετικό από το φαινόμενο του false sharing, οπότε το padding δε μπορεί να το καταπολεμήσει.

Επιπλέον, θέλαμε να παρατηρήσουμε την επίπτωση των διάφορων πολιτικών scheduling στην επίδοση του προγράμματος. Για το λόγο αυτό, δοκιμάσαμε τις πολιτικές static, dynamic και guided στο 1ο for-loop της while. Ακολουθούν οι μετρήσεις για την κάθε μία από αυτές:

Guided

Πίνακας με μετρήσεις:

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.841180	2.983750	1.559910	0.859300	0.450160	0.246270	0.255830
Τυπική απόκλιση (sec)	0.080646	0.063497	0.027165	0.004932	0.002214	0.002547	0.014477

Static

Πίνακας με μετρήσεις:

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.765720	2.978200	1.561820	0.876570	0.458670	0.388080	0.283390
Τυπική απόκλιση (sec)	0.150873	0.116473	0.053042	0.013756	0.002371	0.000240	0.033343

Dynamic

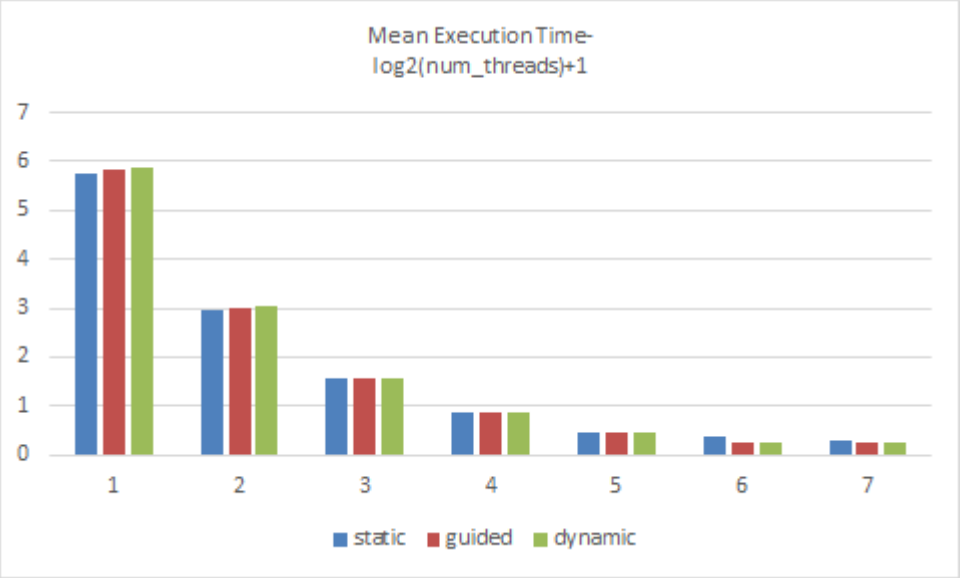
Πίνακας με μετρήσεις:

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.88113	3.05781	1.56684	0.87128	0.45199	0.24871	0.24308
Τυπική απόκλιση (sec)	0.095462	0.065665	0.027284	0.014617	0.002547	0.002342	0.019548

Από τις παραπάνω μετρήσεις συμπεραίνουμε ότι η πολιτική dynamic είναι η καλύτερη όσον αφορά την επίδοση του πρώτου for-loop της do-while. Θεωρητικά, για περίπου ίδιο workload σε κάθε επανάληψη συμφέρει η static, καθώς έχει μικρότερο κόστος σε σχέση με τις υπόλοιπες οι οποίες έχουν ανάγκη για συγχρονισμό των νημάτων. Όμως, όταν το workload διαφέρει από τη μία επανάληψη σε σχέση με μία άλλη, τότε υπερτερούν οι πολιτικές dynamic και guided. Στην περίπτωσή μας, βλέπουμε ότι η πολιτική dynamic επιτυγχάνει το μεγαλύτερο speedup. Αυτό μπορεί να οφείλεται και στο γεγονός ότι υπάρχουν οι ατομικές εντολές, με αποτέλεσμα τα threads να κάνουν spin σε αυτές, οπότε θα θέλαμε ιδανικά εάν κάποιο από τα υπόλοιπα νήματα έχει

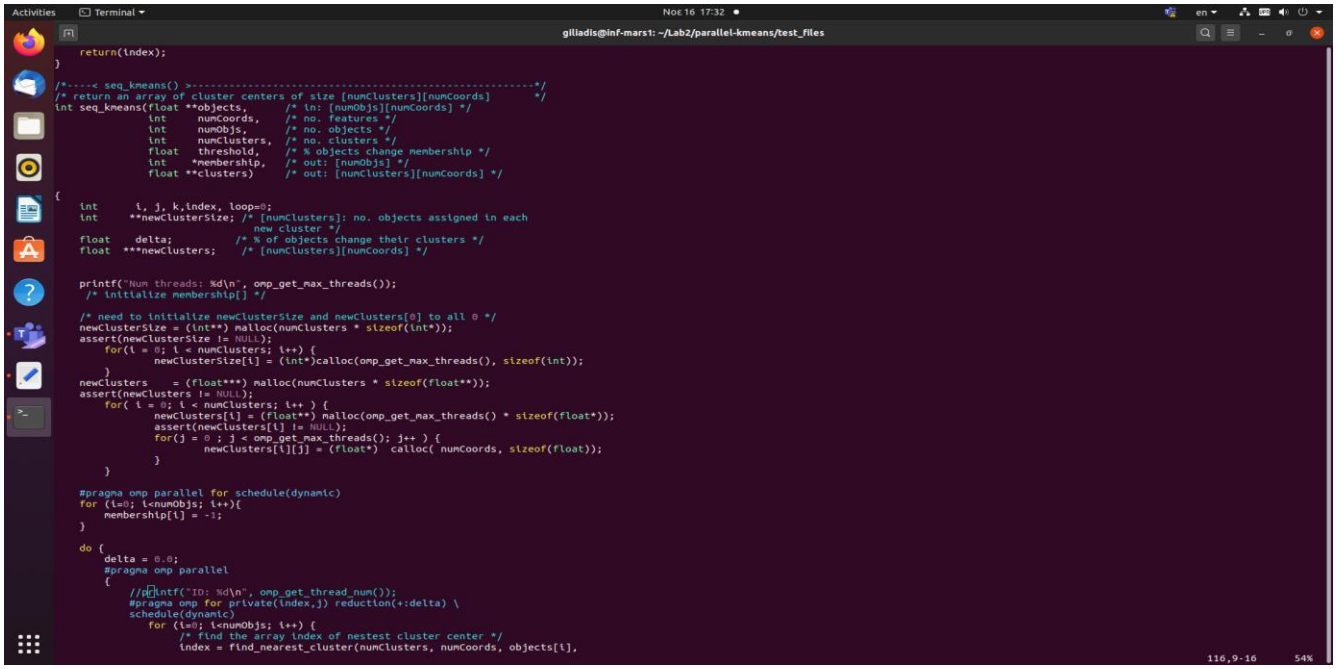
τελειώσει τη δουλειά που του ανατέθηκε, να αναλάβει να εκτελέσει και άλλες επαναλήψεις. Ανάμεσα στις guided και dynamic, δεν υπάρχουν μεγάλες διαφορές όσον αφορά το χρόνο, όμως η dynamic έχει γενικά πιο γρήγορο χρόνο εκτέλεσης, επομένως κρατάμε αυτή την πολιτική για το συγκεκριμένο loop.

Στο παρακάτω γράφημα εμφανίζονται οι μέσοι χρόνοι εκτελέσεις για τις τρεις αυτές πολιτικές.

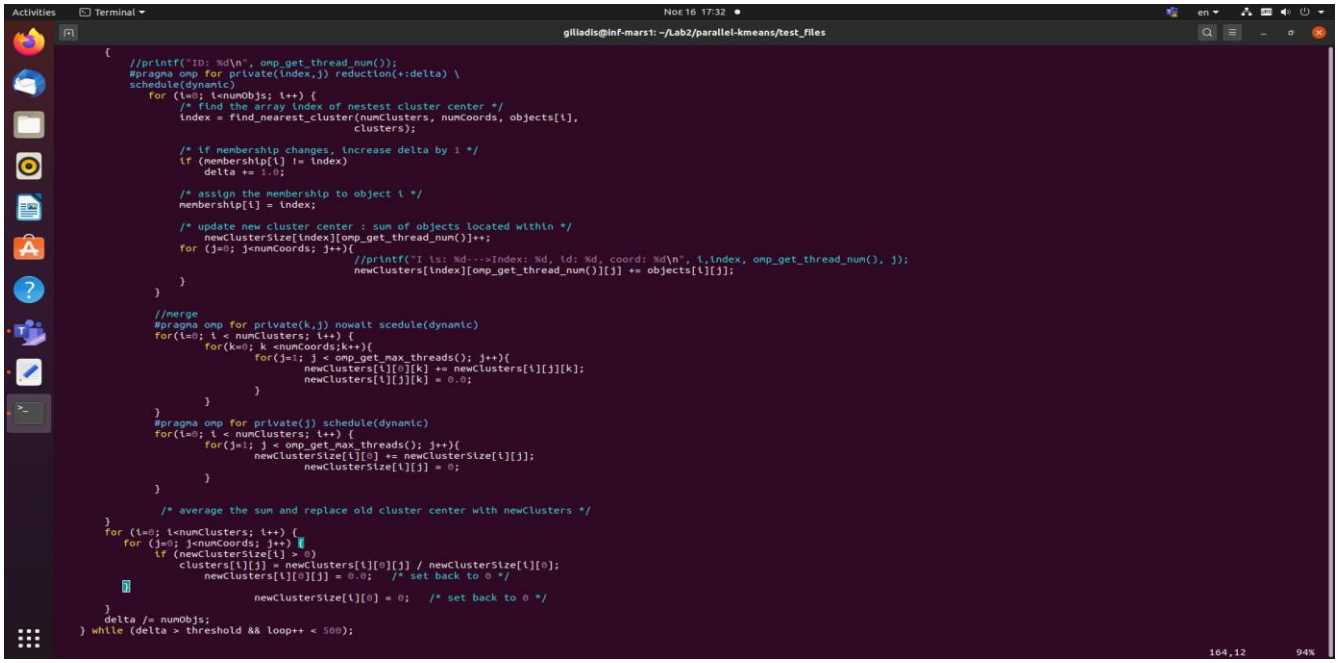


Εικόνα 13: Scheduling dynamic-guided-static.

Μία τελευταία προσπάθεια για βελτιστοποίηση του κώδικα, έγινε θέλοντας να εξαλείψουμε τις ατομικές εντολές στην 1η for της while, με σκοπό να γλιτώσουμε το overhead του συγχρονισμού σε αυτά τα 2 σημεία. Για να το κάνουμε αυτό, δημιουργήσαμε έναν τρισδιάστατο πίνακα ώστε το κάθε νήμα να λειτουργεί ανεξάρτητα από τα υπόλοιπα, δηλαδή να έχει πρόσβαση στις δικές του μεταβλητές που αφορούν τις συντεταγμένες ενός δεδομένου newCluster. Μόλις το κάθε νήμα τελειώσει τους υπολογισμούς, τότε εκτελείται παράλληλος κώδικας ώστε να συγκεντρώσει όλα τα επιμέρους αποτελέσματα, αποθηκεύοντάς τα στις θέσεις του πρώτου νήματος, ώστε τελικά να χρησιμοποιηθούν τα σωστά αποτελέσματα. Αυτό, όπως φαίνεται και από τις μετρήσεις, δεν βοήθησε από άποψη επίδοσης, παρόλο που εξαλείφθηκαν οι ατομικές εντολές. Αυτό θεωρούμε ότι οφείλεται στο κόστος του merge των επιμέρους αποτελεσμάτων, αλλά και στο ότι χαλάμε την τοπικότητα στη μνήμη βάζοντας τρισδιάστατο πίνακα.



Εικόνα 14.α : Υλοποίηση merge



Εικόνα 14.β: Συνέχεια υλοποίησης merge

Πίνακας με μετρήσεις:

#Threads	1	2	4	8	16	32	64
Μέσος χρόνος εκτέλεσης(sec)	5.526620	2.965700	1.544300	0.863760	0.457310	0.262110	0.253600
Τυπική απόκλιση (sec)	0.105990	0.113195	0.039654	0.011374	0.004706	0.002494	0.013885

Σχολιασμός αποτελεσμάτων: Παρατηρούμε βελτίωση για μικρό αριθμό νημάτων σε σχέση με την προηγούμενη καλύτερη έκδοση κώδικα. Όμως, για μεγάλο αριθμό νημάτων ο χρόνος γίνεται χειρότερος. Αυτό θεωρούμε ότι οφείλεται διότι το κόστος του merge των επιμέρους αποτελεσμάτων αυξάνεται όσο αυξάνεται και ο αριθμός των νημάτων (αυξάνεται το μέγεθος του τρισδιάστατου πίνακα).

Γενική παρατήρηση: Ο χρόνος για τους παράλληλους κώδικες με χρήση 1 thread κάποιες φορές βγαίνει λίγο μεγαλύτερος από τον ακολουθιακό. Αυτό οφείλεται στο overhead της παραλληλοποίησης.

Τέλος, δεδομένου ότι υπάρχει πιθανότητα ο compiler να εφάρμοσε την βελτιστοποίηση που προαναφέρθηκε, ο βέλτιστος κώδικας από άποψη speedup σε σχέση με τον ακολουθιακό σε μεγάλο workload, είναι αυτός στον οποίο παραλληλοποιήσαμε μόνο την 1η for της while με dynamic πολιτική για scheduling, σε συνδυασμό με την παραλληλοποίηση των αρχικοποιήσεων στην seq_kmeans με την ίδια πολιτική.