Clustering K-Means Algorithm

Parallelisation with OpenMP

Γεωργακίδης Χρήστος (Georgakidis Chris)
Πανεπιστήμιο Θεσσαλίας (University of Thessaly)
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών (ECE Department)
Ελλάδα (Greece)
email: cgeorgakidis@uth.gr
AEM: 1964

Αυτή η εργασία είναι μέρος του μαθήματος "Συστήματα Υπολογισμού Υψηλών Επιδόσεων" (High Performance Computing - HPC) και πιο συγκεκριμένα στην εξοικείωση με το εργαλείο OpenMP

Keywords—HPC; OpenMP; Clustering K-Means; Parallelisation

I. INTRODUCTION

Ο αλγόριθμος Clustering K-Means ομαδοποιεί n παρατηρήσεις και k clusters. Κάθε cluster χαρακτηρίζεται από τις συντεταγμένες του "κέντρου" του (centroid). Κάθε μία από τις n παρατηρήσεις ανατίθεται στο cluster από το centroid του οποίου έχει τη μικρότερη απόσταση. Η όλη διαδικασία σταματά μόλις κριθεί ότι δεν χρειάζεται κάποια παρατήρηση να αλλάξει cluster. Γίνεται αντιληπτό για τον αλγόριθμο αυτό ότι όσο αυξάνεται το πλήθος των clusters θα αυξάνεται και ο χρόνος εκτέλεσής του. Έτσι κρίνεται καλή λύση η παραλληλοποίησή του. Προς επίτευξη αυτού του στόχου επιλέχθηκε το μοντέλο προγραμματισμού OpenMP, του οποίου βασικό πλεονέκτημα είναι ότι επιτρέπει την ένκολη, σταδιακή παραλληλοποίηση ακολουθιακών εφαρμογών.

II. SEQUENTIAL VERSION OF THE ALGORITHM

Έστω X = {x1, x2, x3, …, xn} είναι το σύνολο από n παρατηρήσεις και V = {v1, v2, v3, vk} είναι το σύνολο από τα κέντρα. Αρχικά, επιλέγονται k τυχαία κέντρα των clusters. Έπειτα, υπολογίζουμε την απόσταση ανάμεσα σε κάθε παρατήρηση και στα κέντρα των clusters, και αναθέτουμε την κάθε παρατήρηση στο cluster, από το κέντρο του οποίου απέχει την μικρότερη απόσταση. Σε επόμενο βήμα, υπολογίζονται τα νέα κέντρα του κάθε cluster χρησιμοποιώντας τον τύπο:

$$v_i = (1/c_i) \sum_{j=1}^{c_i} x_i$$

, όπου 'ci' αναπαρηστά το πλήθος των παρατηρήσεων στο i-στό cluster. Στη συνέχεια, επανυπολογίζουμε την απόσταση κάθε παρατήρησης από τα νέα κέντρα. Η όλη διαδικασία επαναλαμβάνεται μέχρι καμία από τις παρατηρήσεις δεν ανατέθηκε σε άλλο cluster σε μία επανάληψη.

III. PARALLEL VERSION OF THE ALGORITHM

Από την περιγραφή του αλγορίθμου γίνεται φανερό ότι ο αλγόριθμος είναι παραλληλοποιήσιμος. Προς επίτευξη αυτού του στόχου ακολουθήθηκε η ακόλουθη στρατηγική:

1. Προκειμένου να εντοπιστεί το σημείο του κώδικα στο οποίο καταναλώνεται ο περισσότερος χρόνος χρησιμοποιήθηκε το εργαλείο της Intel, Vtune Amplifier. Από την εκτέλεσή του εντοπίστηκε ότι το πιο χρονοβόρο κομμάτι κώδικα είναι η συνάρτηση euclid_dist_2(). Οπότε, ως πρώτο βήμα κρίθηκε η παραλληλοποίηση της συνάρτησης αυτής. Παρακάτω φαίνεται η παραλληλοποιημένη εκδοχή της συνάρτησης αυτής.

```
int i;
float ans=0.0;
#pragma omp parallel for reduction(+:ans)
for (i=0; i<numdims; i++)
   ans = (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
return(ans);
```

Ωστόσο, παρατηρήθηκε ότι ο χρόνος αυξήθηκε. Ο λόγος ήταν ότι η euclid_dist_2d() καλείται numClusters φορές από την find_nearest_cluster() με αποτέλεσμα μέγαλο overhead λόγο της συνεχής δημιουργίας και καταστροφής threads. Έτσι αυτό το βήμα ήταν ανεπιτυχές.

2. Σαν δεύτερο βήμα παραλληλοποίησης επιλέχθηκε η παραλληλοποίηση της find_nearest_cluster(), μιας και

```
#pragma omp parallel for private (dist)
for (…) {
      dist = euclid_dist_2d(…);
      #pragma omp critical {
         if (dist < min_dist){ //make dist the new min}
      }
}
```

αυτή καλεί numClusters φορές την euclid_dist_2d(). Ωστόσο και αυτή η προσπάθεια παραλληλοποίησης είχε αρνητική επίπτωση στο χρόνο. Και σε αυτή και στη προηγούμενη περίπτωση ο λόγος της καθυστέρησης αυτής σε χρόνος οφείλεται σε μεγάλο βαθμό στο ότι και η find_nearest_cluster() και η euclid_dist_2d() καλούνται πολλές φορές από άλλη συνάρτηση, η euclid_dist_2d() από την find_nearest_cluster() η οποία με τη σειρά της καλείται πολλές φορές από την seq_kmeans(). Αυτό έχει ως αποτέλεσμα να δημιουργείται τεράστιο overhead λόγω της συνεχούς δημιουργίας και καταστροφής threads.

3. Η αποτυχία των δύο παραπάνω προσπαθειών παραλληλοποίησης είχε ως μονόδρομο την παραλληλοποίηση της seq_kmeans(), ή οποία καλή όλες τις άλλες. Σε αυτό το βήμα έγιναν πολλές ενέργειες οι οποίες παρουσιάζονται παρακάτω:

a. Αρχικά, έγινε παραλληλοποίηση του for loop που αρχικοποιεί τον πίνακα με τα memberships. Δεν υπήρχε κανένα είδος

```
#pragma omp parallel for schedule(static)
for (…){
        membership[i] = -1;
}
```

εξάρτησης δεδομένων, οπότε ήταν εύκολο βήμα.

b. Στη συνέχεια, έγινε προσπάθεια για την παραλληλοποίηση του επόμενου for loop που αρχικοποιεί τον πίνακα newClusters.

```
#pragma omp parallel for schedule(static)
for (…){
        newClusters[i] = newClusters[0] +
        i*numCoords;
}
```

Ωστόσο, σε αυτό το for loop υπάρχουν data depedencies καθώς κάθε i-οστό στοιχείο του πίνακα newCluster είναι ίσο με το προηγούμενό του αυξημένο κατά numCoords. Αυτό πρακτικά είναι ισοδύναμο με το να ισούται το i-οστό στοιχείο του πίνακα newClusters με το αρχικό στοιχείο του πίνακα (newClusters[0]) αυξημένο κατά i*numCoords. Αυτό έχει ως συνέπεια την εξάλειψη των data depedencies και ως εκ τούτου την δυνατότητα παραλληλοποίησης αυτού του for loop.

c. Τα παραπάνω βελτίωσαν τον χρόνο όμως παρατηρήθηκε ότι εφικτή θα ήταν επίτευξη καλύτερου χρόνου με το να συγχωνευθούν αυτά τα δύο parallel regions, καθώς έτσι κάθε thread θα δημιουργόταν και θα καταστρεφόταν μία φορά, σε αντίθεση με τώρα που κάθε νήμα δημιουργείται και καταστρέφεται σε καθένα από τα δύο for loops. Μεταξύ των δύο for loop υπάρχει ένα κομμάτι κώδικα που πραγματοποιεί δεσμεύσεις μνήμης και αρχικοποιήσεις. Αυτό είναι τελείως ανεξάρτητο από τον κώδικα του προηγούμενου for loop, με αποτέλεσμα αυτό να μπορεί να μετατοπιστεί πάνω από το πρώτο for loop. Έτσι τώρα τα δύο parallel regions μπορούν να ενωθούν σε ένα. Επίσης, λόγω της ένωσής τους σε κάθε ένα omp for προστίθεται και το nowait καθώς δεν υπάρχει εξάρτηση από το ένα for loop στο άλλο. Έτσι κάθε thread θα περιμένει τα υπόλοιπα μία και καλή στο barrier στο τέλος του parallel region.

```
#pragma omp parallel
{
        #pragma omp for nowait schedule
(static)
        for (…){
           membership[i] = -1;
        }
        #pragma omp for nowait schedule
(static)
        for(…){
           newClusters[i]= newClusters[0] +
                  i*numCoords;
        }
} //end of parallel region
```

d. Επίσης στα παραπάνω προτιμήθηκε η χρησιμοποίηση ως schedule το static καθώς το σώμα των for loops είναι μικρό και απλό (αναθέσεις, πρόσθεση & πολλαπλασιασμοί). Έτσι κάθε thread παίρνει ένα συγκεκριμένο κομμάτι επαναλήψεων, το default που είναι number_of_iterations/number_of_threads και το εκτελεί χωρίς να συμβαίνουν πολλές σύντομες εναλλαγές στα threads, συμβάλοντας στην καλύτερη επίτευξη χρόνου σε σχέση με τα άλλα είδη schedule (όχι όμως αισθητά).

e. Στη συνέχεια, ήταν η σειρά για την παραλληλοποίηση του do while. Ωστόσο αυτό δεν ήταν δυνατό καθώς δεν υπάρχει κάποιο construct του OpenMP για την παραλληλοποίηση του do while, εκτός και αν αυτό μετατραπεί σε for loop. Στον συγκεκριμένο κώδικα δεν κατέστη δυνατή μία τέτοια μετατροπή.

f. Το παραπάνω είχε ως αποτέλεσμα την παραλληλοποίηση του σώματος του do while.

```
do{
    delta = 0.0;
    #pragma omp parallel
    {
        for (…) {
            …
        }
        for (…) {
            …
        }
    }//end of parallel region
    delta /= numObjs;
}while (delta>threshold && loop++<500);
```

g. Το σώμα του parallel region αποτελείται από δύο for loops. Αρχικά, έγινε η παραλληλοποίηση του δεύτερου, καθώς είναι πιο απλό. Σε αυτό το for loop επιλέχθηκε ως schedule το static εφόσον καθεμία από αυτές τις επαναλήψεις είναι ανεξάρτητη από τις υπόλοιπες. Έτσι κάθε thread λαμβάνει ίσο αριθμό επαναλήψεων και μιας και όλα τα κομμάτια επαναλήψεων έχουν ίδιο βάρος υπολογισμών κανένα

```
do{
    delta = 0.0;
    #pragma omp parallel private(j)
    {
        for (…) {
            …
        }
        #pragma omp for schedule(static)
        for (…) {
            for (…){
                …
            }
        }
    }//end of parallel region
    delta /= numObjs;
}while (delta>threshold && loop++<500);
```

thread δεν θα περιμένει σε κατάσταση idle για πολύ χρόνο, όπως αυτό θα συνέβαινε αν επιλεγόταν ως schedule dynamic ή guided. Επίσης, θα πρέπει η μεταβλητή j να δηλωθεί ως private στο omp parallel καθώς όλα τα threads θα έχουν την ίδια μεταβλητή j, οπότε αυτή θα πρέπει να γίνει ιδιοτική για κάθε ένα από τα threads. Επίσης, δεν προστέθηκε nowait. Στην συγκεκριμένη περίπτωση επειή δεν υπάρχει κάποια άλλη εντολή μετά στο parallel region, το barrier του omp for και το barrier του omp parallel συγχωνεύονται σε ένα.

h. Έπειτα σειρά έχει το πρώτο for loop. Σε αυτό το for loop χρειάζεται προσοχή καθώς πραγματοποιούνται πράξεις με floating points. Αυτό είναι ένα πρόβλημα στον παραλληλισμό καθώς στις πράξεις με floating points μπορεί να διαφέρει το αποτέλεσμα ανάλογα με τη σειρά που πραγματοποιούνται οι πράξεις. Στην συγκεκριμένη περίπτωση, η πρόσθεση newClusters[index][j] += objects[i][j] στο εμφωλευμένο for loop αντιμετωπίζει αυτό το πρόβλημα κατά την παραλληλοποίηση του εξωτερικού for loop, καθώς η πρόσθεση floating points δεν είναι αντιμεταθετική. Γι' αυτό το λόγο επιλέχθηκε ως schedule το dynamic με μέγεθος chunk 1. Αυτό έχει ως αποτέλεσμα να υπάρχει μεγαλύτερη πιθανότητα τα threads να εκτελέσουν την πρόσθεση αυτή με τη σειρά που αυτή εκτελείται στον ακολουθιακό αλγόριθμο σε σχέση με το static. Το guided στις πρώτες επαναλήψεις δίνει μεγάλα κομμάτια και όσο περνάνε οι επαναλήψεις αυτά μικραίνουν. Αυτό έχει ως αποτέλεσμα η διαφορά του αποτελέσματος να είναι καλύτερη από αυτή του static, ωστόσο όμως δεν είναι τόσο καλό όσο το dynamic. Ίσως κάνει μία μικρή βελτίωση στο χρόνο σε σχέση με το dynamic αλλά αυτό είναι ένα trade off μεταξύ accurance και timing, που προσωπικά επιλέχθηκε να δωθεί μεγαλύτερη σημασία στο accurance. Δοκιμάστηκε και η επιλογή ordered έτσι ώστε να γίνουν οι πράξεις ακριβώς με τη σειρά που θα έτρεχαν στον ακολουθιακό κώδικα, όμως αυτό είχε ως αποτέλεσμα την εκτόξευση του χρόνου, το οποίο είναι λογικό καθώς απαιτείται τεράστιος χρόνος για locks. Επίσης το index, στο οποίο επιστρέφεται το αποτέλεσμα από την find_nearest_cluster(), πρέπει να είναι ιδιωτικό για κάθε thread. οπότε πρέπει να προστέθει στην λίστα των private μεταβλητών στον ορισμό του parallel region. Επιπλέον, η συσσωρευτική αύξηση του newClusterSize[index] πρέπει να είναι ατομική καθώς μπορεί να τύχει δύο νήματα να έχουν το ίδιο index και να πάνε να αυξήσουν το ίδιο στοιχείο του πίνακα δύο φορές, οπότε πρέπει να χαρακτηριστεί atomic. Για τον ίδίο ακριβώς λόγο πρέπει να είναι atomic και η πράξη newClusters[index][j] += objects[i][j]. Ο λόγος που επιλέχθηκε

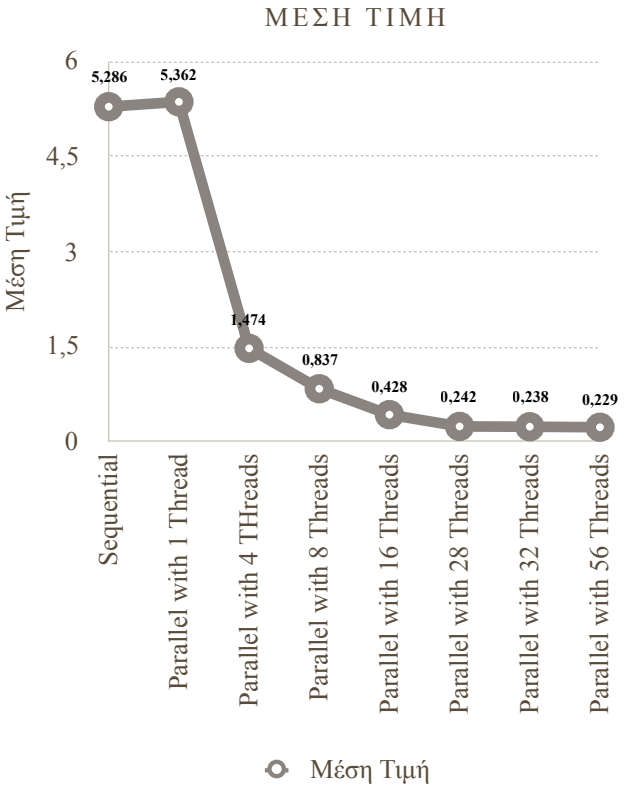
οι δύο αυτές εντολές να μπουν σε atomic και όχι σε ένα critical region είναι γιατί στα critical regions που είναι εκτενή χάνεται πολύς χρόνος περιμένοντας για locks. Επίσης υπάρχει μία μικρή διαφορά μετάξύ του critical και του atomic. Το critical προστατεύει μόνο σε επίπεδο εκτέλεσης του κώδικα, ενώ το atomic προστατεύει και από ταυτόχρονη προσπέλαση στην ίδια θέση μνήμης, κάτι που στην συγκεκριμένη περίπτωση το χρειαζόμαστε. Το ότι επιλέχθηκε διαφορετικό schedule για καθένα από τα δύο for loop οδηγεί στο να μην μπορούν να ενωθούν σε ένα. Επίσης αυτό δεν είναι και επιθυμητό καθώς στο πρώτο for loop αλλάζουν οι τιμές του πίνακα newClusters, ο οποίος στη συνέχεια χρησιμοποιείται στο επόμενο. Έτσι, θα πρέπει κάθε νήμα που τελειώνει το for loop να περιμένει όλα τα άλλα στο barrier, το οποίο είναι υλοποιημένο από το omp for και γι' αυτό το λόγο δεν προστέθηκε nowait, το οποίο το αφαιρεί.

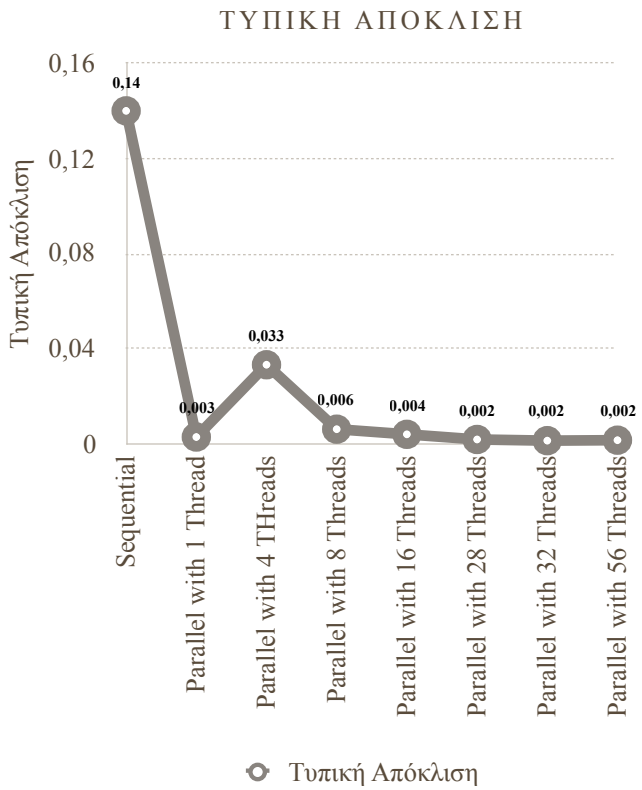
4. Τέλος, για την καλύτερη απόδοση χρησιμοποιήθηκαν δύο μεταβλητές περιβάλλοντος. Πιο συγκεκριμένα, τέθηκε OMP_PROC_BIND = true, η οποία αποτρέπει τον κίνδυνο τα threads να μεταπηδούν ανάμεσα στα cores. Αυτό έχει ως αποτέλεσμα να εξασφαλίζεται καλύτερη τοπικότητα Επίσης, τέθηκε OMP_DYNAMIC = false, έτσι ώστε να μην επιτρέπεται από το σύστημα να μεταφέρει λιγότερα threads από αυτά που ζητήθηκαν.

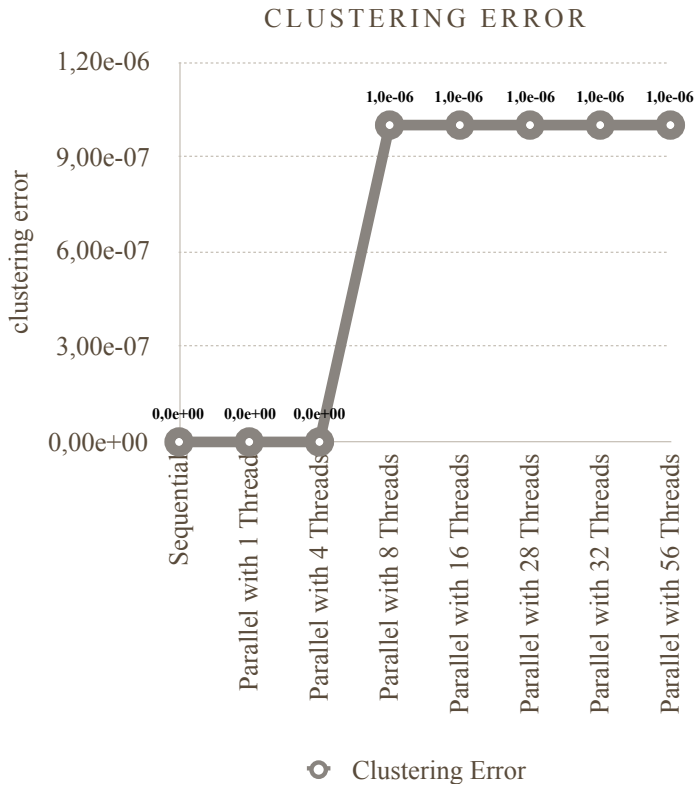
```
    do{
            delta = 0.0;
            #pragma omp parallel private(index, j)
            {
             #pragma omp for schedule(dynamic, 1)
                for (…) {
                   …
                   #pragma omp atomic
                   newClusterSize[index]++;
                      for (j=0; j<numCoords;
j++) {
                       #pragma omp atomic
                     newClusters[index][j] +=
objects[i][j];
                     }
                } //end of omp for


                          #pragma omp for
schedule(static)
                for (…) {
                   for (…){
                      …
                   }
                } //end of omp for
            } //end of parallel region
            delta /= numObjs;
    }while (delta>threshold && loop+
+<500);
```

IV. CONCLUSION

Η παραλληλοποίηση του κώδικα είναι ένα πολύ σημαντικό κομμάτι στην βελτιστοποίηση προγραμμάτων. Το OpenMP έιναι ένα εργαλείο που βοηθάει προς αυτήν την κατεύθυνση. Ωστόσο το ποσοστό βελτίωσης εξαρτάται και από το εκάστοτε σύστημα. Επίσης μπορεί να παρατηρηθεί το φαινόμενο να χειροτερέψει η απόδοση του προγράμματος όσο αυξάνεις τον αριθμό τον threads που παρέχεις. Για τη συγκεκριμένη εργασία πραγματοποιήθηκαν οι μετρήσεις με flags στον μεταγλωττιστή -fast -qopenmp -DNDEBUG και παρακάτω δίδονται τα διαγράμματα μέσης τιμής, τυπικής απόκλισης καθώς και σφάλματος αποτελέσματος λόγω των floating point operations.







References

Η παραπάνω στρατηγική παραλληλοποίησης που ακολουθήθηκε πρόεκυψε μετά από κατανόηση της διάλεξης του μαθήματος για το OpenMP καθώς και από πηγές στο Internet που παρατήθονται παρακάτω:

1. OpenMP Related Tips by INTEL [https://software.intel.com/en-us/articles/openmp-related-tips].
2. OpenMP Tips, Tricks and Gotchas by EPCC, University of Edinburgh [https://www.archer.ac.uk/training/course-material/2017/08/openmp-ox/Slides/L09-TipsTricksGotchas.pdf].
3. OpenMP by Wikipedia [https://en.wikipedia.org/wiki/OpenMP].
4. Common Mistakes in OpenMP and How to Avoid Them, Paper by Michael Su¨ ß and Claudia Leopold of University of Kassel [http://michaelsuess.net/publications/suess_leopold_common_mistakes_06.pdf].
5. How to Get Good Performance by Using OpenMp [http://www.akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf].