ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΕ ΠΡΟΒΛΗΜΑΤΑ ΜΗΧΑΝΙΚΗΣ ΜΑΘΗΣΗΣ

2η Εργαστηριακή Άσκηση

Ο αλγόριθμος των κ-μέσων «K-MEANS με openMP»

Ον/μο: Γεώργιος Σκόνδρας

Τμήμα: Μηχανικών Η/Υ και Πληροφορικής

Α.Μ: 1020408

Σημείωση

Για την υλοποίηση της άσκησης χρησιμοποιήθηκε προσωπικός φορητός υπολογιστής Huawei Matebook d14 με τα παρακάτω χαρακτηριστικά

- *Επεξεργαστής: AMD Ryzen 3500U*
- *Λειτουργικό Σύστημα: (μέσω Oracle VM Virtual Box 6.1) Ubuntu 20.04.1 LTS*
- *Compiler: gcc 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)*

Εφόσον χρησιμοποιήθηκε Virtual Machine για την εκπόνηση της άσκησης, παραθέτω τις ρυθμίσεις που επιλέχτηκαν

Βαλε εικονα εδώ

Δώσαμε 2 φυσικούς πυρήνες που αντιστοιχούν σε 4 thread(Simultaneous Multi Threading)

*Οι κώδικες για την κάθε εργασία βρίσκονται στα αρχεία *.c στους φακέλους ex1,ex2 κλπ*

Εργασία 1

Πήραμε το αρχείο σειριακού κώδικα από την προηγούμενη άσκηση και η μοναδική προσθήκη που έγινε ήταν η προσθήκη της παρακάτω εντολής, ώστε να τυπώνονται οι αθροιστικές αποστάσεις των σημείων από τα κέντρα τους σε κάθε επανάληψη.

```
printf("Iteration %d distance: %f\n",i,newDistance);
```

Αρχικά θα κάνουμε profiling του σειριακού κώδικα. Για να γίνει αυτό θα κάνουμε compile χωρίς optimizations στον compiler, ώστε να μην γίνει inlining και να μπορούμε να δούμε πόσο χρόνο καταναλώνει η κάθε συνάρτηση. Οπότε κάνουμε compile με παράμετρο ***-pg*** για να ενεργοποιήσουμε τη λειτουργία του profiling όπως φαίνεται παρακάτω και εκτελούμε το πρόγραμμα.

```
osboxes@osboxes:~/Desktop/parProg/lab2/ex1$ gcc -pg kmeans_serial.c -o kmeans_serial
osboxes@osboxes:~/Desktop/parProg/lab2/ex1$ time ./kmeans_serial
Iteration 1 distance: 60934692.000000
Iteration 2 distance: 33077994.000000
Iteration 3 distance: 33048914.000000
Iteration 4 distance: 33035394.000000
Iteration 5 distance: 33027598.000000
Iteration 6 distance: 33022234.000000
Iteration 7 distance: 33018904.000000
Iteration 8 distance: 33016474.000000
Iteration 9 distance: 33014706.000000
Iteration 10 distance: 33013304.000000
Iteration 11 distance: 33012220.000000
Iteration 12 distance: 33011290.000000
Iteration 13 distance: 33010534.000000
Iteration 14 distance: 33010042.000000
Iteration 15 distance: 33009362.000000
Iteration 16 distance: 33008800.000000

real    14m44,627s
user    14m44,084s
```

Στη συνέχεια τρέχουμε τον ***gprof*** και αποθηκεύουμε τα αποτελέσματα στο αρχείο ***analysis.txt***, το οποίο ανοίγουμε, όπως φαίνεται παρακάτω.

```
osboxes@osboxes:~/Desktop/parProg/lab2/ex1$ gprof kmeans_serial gmon.out > analysis.txt
osboxes@osboxes:~/Desktop/parProg/lab2/ex1$ cat analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.36    544.07   544.07       16    34.00    34.00  Classification
  1.01    549.60     5.54       16     0.35     0.35  UpdateCenters
  0.11    550.21     0.60        1     0.60     0.60  InitializeVectors
  0.00    550.21     0.00        1     0.00     0.00  InitializeCenters
```

Βλέποντας την έκθεση του profiler, παρατηρούμε πως η συνάρτηση που εκτελεί τους πιο χρονοβόρους υπολογισμούς είναι η συνάρτηση ***Classification***, δηλαδή η συνάρτηση που υπολογίζει τις αποστάσεις των διανυσμάτων από τα κέντρα. Πιο συγκεκριμένα καταναλώνει το **99.36%** του χρόνου του προγράμματος. Συνεπώς η συνάρτηση αυτή είναι απαραίτητο να παραλληλοποιηθεί. Παρακάτω φαίνονται αναλυτικά οι συναρτήσεις που καταναλώνουν τον περισσότερο χρόνο

- ***Classification*** 99.36%
- ***UpdateCenters*** 1.01%
- ***InitializeVectors*** 0.11%

Συνεπώς σε αρχικό στάδιο αυτές είναι οι συναρτήσεις που θα προσπαθήσουμε να παραλληλοποιήσουμε.

Σε αυτό το σημείο θα εκτελέσουμε τον σειριακό κώδικα και με optimizations στον compiler, ώστε να έχουμε μία εικόνα για τον χρόνο που καταναλώνει ο σειριακός κώδικας.

Προσθέτουμε στον κώδικα την παρακάτω γραμμή, δίνοντας οδηγίες στον compiler.

```
#pragma GCC optimize("O3","unroll-loops","omit-frame-pointer","inline", "unsafe-math-optimizations")
```

Κάνουμε compile και τρέχουμε το πρόγραμμα, όπως φαίνεται παρακάτω:

```
osboxes@osboxes:~/Desktop/parProg/lab2/ex1$ gcc kmeans_serial.c -o kmeans_serial
osboxes@osboxes:~/Desktop/parProg/lab2/ex1$ time ./kmeans_serial
Iteration 1 distance: 60934656.000000
Iteration 2 distance: 33077976.000000
Iteration 3 distance: 33048906.000000
Iteration 4 distance: 33035310.000000
Iteration 5 distance: 33027438.000000
Iteration 6 distance: 33022304.000000
Iteration 7 distance: 33018978.000000
Iteration 8 distance: 33016260.000000
Iteration 9 distance: 33014594.000000
Iteration 10 distance: 33013316.000000
Iteration 11 distance: 33012052.000000
Iteration 12 distance: 33011180.000000
Iteration 13 distance: 33010442.000000
Iteration 14 distance: 33009870.000000
Iteration 15 distance: 33009214.000000
Iteration 16 distance: 33008626.000000

real    0m41,398s
user    0m41,105s
sys     0m0,287s
```

Παίρνουμε χρόνο **41.4sec**.

Εργασία 2

Σε πρώτη φάση παραλληλοποιώ την συνάρτηση ***Classification***. Η συνάρτηση αυτή περιέχει ένα τριπλό for loop το οποίο υπολογίζει για κάθε διάνυσμα την απόστασή του από κάθε κέντρο. Αυτό γίνεται προσθέτοντας τα παρακάτω OpenMP directive πριν το τριπλό for loop που γίνεται ο υπολογισμός των αποστάσεων. Να σημειωθεί ότι μετά από αρκετές απόπειρες να παραλληλοποιηθούν και άλλα σημεία του κώδικα, εκτός από το τριπλό for loop, το αποτέλεσμα ήταν σε χρόνο ή ίδιο ή ελαφρώς χειρότερο. Αυτό οφείλεται στο γεγονός ότι η συνάρτηση ***Classification*** που περιέχει μόνο αυτό το τριπλό for loop καταναλώνει το 99,36% του χρόνου του αλγορίθμου σύμφωνα με τον profiler. Συνεπώς όλα τα άλλα τμήματα του κώδικα εκτελούνται αρκετά γρήγορα και σειριακά, οπότε το overhead για create και join των thread για αυτά τα τμήματα μπορεί να είναι μεγαλύτερο από το να εκτελούνταν απλά σειριακά.

```
#pragma omp parallel
```

```
#pragma omp for private(j,k,dist,euclDist) reduction(+:cumDist) reduction(min:minDist)
for( i = 0; i < N; i++ ){
    minDist= FLT_MAX;
    for( j = 0; j < Nc; j++ ){
        dist = 0;
        for (k = 0; k < Nv; k++) {
            euclDist = ( Vec[i][k] - Center[j][k]) * ( Vec[i][k] - Center[j][k] );
            dist += euclDist;
        }
        if ( dist < minDist ) { //Always enters here
            minDist = dist;
            Classes[i] = j;
        }
    }
    cumDist += minDist;
}
```

Παρακάτω αιτιολογούνται οι επιλογές της παραλληλοποίησης:

- **#pragma omp parallel** Ανοίγουμε παράλληλη περιοχή με αριθμό threads όσα δηλώσαμε με την εντολή ***omp_set_num_threads()*** .Στη συγκεκριμένη εργασία, για το σύστημα που εργαζόμαστε χρησιμοποιήθηκαν 4 thread.
- **#pragma omp for**: Δίνουμε οδηγία το παρακάτω for loop να εκτελεστεί παράλληλα
- **private(j,k,dist,euclDist)**: επειδή αυτές οι μεταβλητές χρησιμοποιούνται στα εμφωλευμένα for loop, τις δηλώνουμε ως private για κάθε thread, ώστε να αποφύγουμε τα race conditions.
- **reduction(+:cumDist)**: Η μεταβλητή cumDist υπολογίζει την αθροιστική απόσταση των διανυσμάτων από τα κέντρα τους. Το συγκεκριμένο directive δίνει οδηγία ώστε το κάθε thread να υπολογίσει το δικό του cumDist σύμφωνα με τα δεδομένα που επεξεργάζεται αυτό και όταν κάνουν join όλα τα thread αθροίζονται όλα τα επιμέρους cumDist, ώστε να υπολογιστεί το συνολικό.

- **reduction(min:minDist):** Η μεταβλητή minDist υπολογίζει την ελάχιστη απόσταση ενός διανύσματος από τα κέντρα. Εδώ έχουμε την ίδια λογική με την παραπάνω εντολή, αλλά για τον υπολογισμό ελαχίστου.

Στη συνέχεια κάνουμε compile και εκτελούμε την παράλληλη έκδοση όπως φαίνεται στην επόμενη εικόνα

```
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ gcc kmeans_omp.c -o kmeans_omp -fopenmp
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ time ./kmeans_omp
Number of available threads:4
Iteration 1 distance: 60935012.000000
Iteration 2 distance: 33077876.000000
Iteration 3 distance: 33049044.000000
Iteration 4 distance: 33035256.000000
Iteration 5 distance: 33027508.000000
Iteration 6 distance: 33022474.000000
Iteration 7 distance: 33018928.000000
Iteration 8 distance: 33016474.000000
Iteration 9 distance: 33014600.000000
Iteration 10 distance: 33013142.000000
Iteration 11 distance: 33011996.000000
Iteration 12 distance: 33011000.000000
Iteration 13 distance: 33010230.000000
Iteration 14 distance: 33009600.000000
Iteration 15 distance: 33009032.000000
Iteration 16 distance: 33008604.000000

real    0m12,531s
user    0m42,777s
sys     0m0,232s
```

Παρατηρούμε σημαντική βελτίωση του χρόνου εκτέλεσης. Είμαστε κοντά σε υποτετραπλασιαμό του χρόνου εκτέλεσης με χρήση 4 threads(**σειριακό 41,4sec**, **παράλληλο 12,5 sec**). Παρόλα αυτά παρατηρούμε αλλαγή των αποστάσεων σε κάθε iteration σε σχέση με τη σειριακή εκτέλεση. Επιπλέον τα σφάλματα είναι διαφορετικά σε κάθε παράλληλη εκτέλεση.

Συνεπώς είναι απαραίτητο να ελέγξουμε την ορθότητα του αλγορίθμου. Για να γίνει αυτό θα εκτελέσουμε τον αλγόριθμο τυπώνοντας τα διανύσματα, καθώς και τα αποτελέσματα που προκύπτουν(κέντρα, κλάσεις αποστάσεις κλπ.). Αυτό θα το κάνουμε αρχικά για ένα μικρό αριθμό διανυσμάτων(kmeans_omp_test1.c, ώστε τα να βεβαιωθούμε ότι τα κέντρα είναι σωστά. Στη συνέχεια θα τον εκτελέσουμε και για τα μεγέθη που ζητούνται και θα

παρατηρήσουμε αν προκύπτουν μη έγκυρες τιμές(kmeans_omp_test2.c). Τα αποτελέσματα των εκτελέσεων βρίσκονται στα αρχεία test1.txt και test2.txt.

```
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ gcc kmeans_omp_test1.c -o kmeans_omp_test1 -fopenmp
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ time ./kmeans_omp_test1 > test1.txt

real    0m0,004s
user    0m0,003s
sys     0m0,003s
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ cat test1.txt
Initialized Vectors!
0.000000 0.000000
0.000000 1.000000
1.000000 0.000000
10.000000 10.000000
10.000000 11.000000
11.000000 10.000000

Initialized Centers!
0.000000 0.000000
0.000000 1.000000

0 0 0 0 0 0
Number of available threads:4
Iteration 1 distance: 584.000000
Iteration 2 distance: 39.437500
Iteration 3 distance: 2.666667
Iteration 4 distance: 2.666667
Iteration 5 distance: 2.666667
Iteration 6 distance: 2.666667
Iteration 7 distance: 2.666667
Iteration 8 distance: 2.666667
Iteration 9 distance: 2.666667
Iteration 10 distance: 2.666667
Iteration 11 distance: 2.666667
Iteration 12 distance: 2.666667
Iteration 13 distance: 2.666667
Iteration 14 distance: 2.666667
Iteration 15 distance: 2.666667
Iteration 16 distance: 2.666667
0.333333 0.333333
10.333333 10.333333

0 0 0 1 1 1
Center 0 has 3 children
Center 1 has 3 children
Finished at 16 iterations
```

```
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ gcc kmeans_omp_test2.c -o kmeans_omp_test2 -fopenmp
osboxes@osboxes:~/Desktop/parProg/lab2/ex2$ time ./kmeans_omp_test2 >test2.txt

real    0m13,545s
user    0m46,399s
sys     0m0,245s
```

Παρατηρώντας τα αποτελέσματα στο test1.txt, βλέπουμε ότι για 6 διανύσματα έγινε σωστή συσταδοποίση και στο test2.txt ότι οι τιμές των κέντρων είναι έγκυρες και κάθε διάνυσμα ανήκει σε ένα κέντρο.

Συνεπώς βεβαιωθήκαμε για την ορθότητα της παράλληλης υλοποίησης. Ο λόγος που βλέπουμε διαφορετικές αποστάσεις σε κάθε εκτέλεση, καθώς και σε σχέση με το σειριακό κώδικα οφείλεται σε σφάλματα στις πράξεις μεταξύ floating point αριθμών. Καθώς η εκτέλεση γίνεται παράλληλα κάθε φορά τα threads εκτελούν τους υπολογισμούς με διαφορετική σειρά. Συνεπώς οι πράξεις μεταξύ των floating point αριθμών γίνονται με διαφορετική σειρά και οδηγούν σε διαφορετικά floating point σφάλματα. Έτσι προκύπτουν διαφορετικές συσταδοποιήσεις, αλλά αυτές είναι ορθές, όπως αποδείξαμε προηγουμένως.

Εργασία 3

Σε αυτό το σημείο με το directive schedule θα προσπαθήσουμε να βελτιώσουμε τους χρόνους εκτέλεσης του προγράμματος. Με αυτή την οδηγία καθορίζουμε τον τρόπο που θα διαιρεθούν τα loop iterations σε chunk sizes. Έγινε χρήση όλων των scheduling types που μας παρέχει η OpenMp(static, dynamic, guided, auto, runtime) και για αρκετά chunk sizes. Ο καλύτερος χρόνος επιτεύχθηκε με τη χρήση του τύπου dynamic chunk size 50, δηλαδή schedule(runtime,50). Με αυτή την επιλογή δημιουργούνται κομμάτια των loop με 50 iterations. Κάθε thread εκτελεί το κομμάτι του και μόλις το εκτελέσει, παίρνει δυναμικά ένα

άλλο. Με αυτή την οδηγία ο χρόνος εκτέλεσης μειώθηκε από τα 12,5s στα 11,6s όπως φαίνεται παρακάτω.

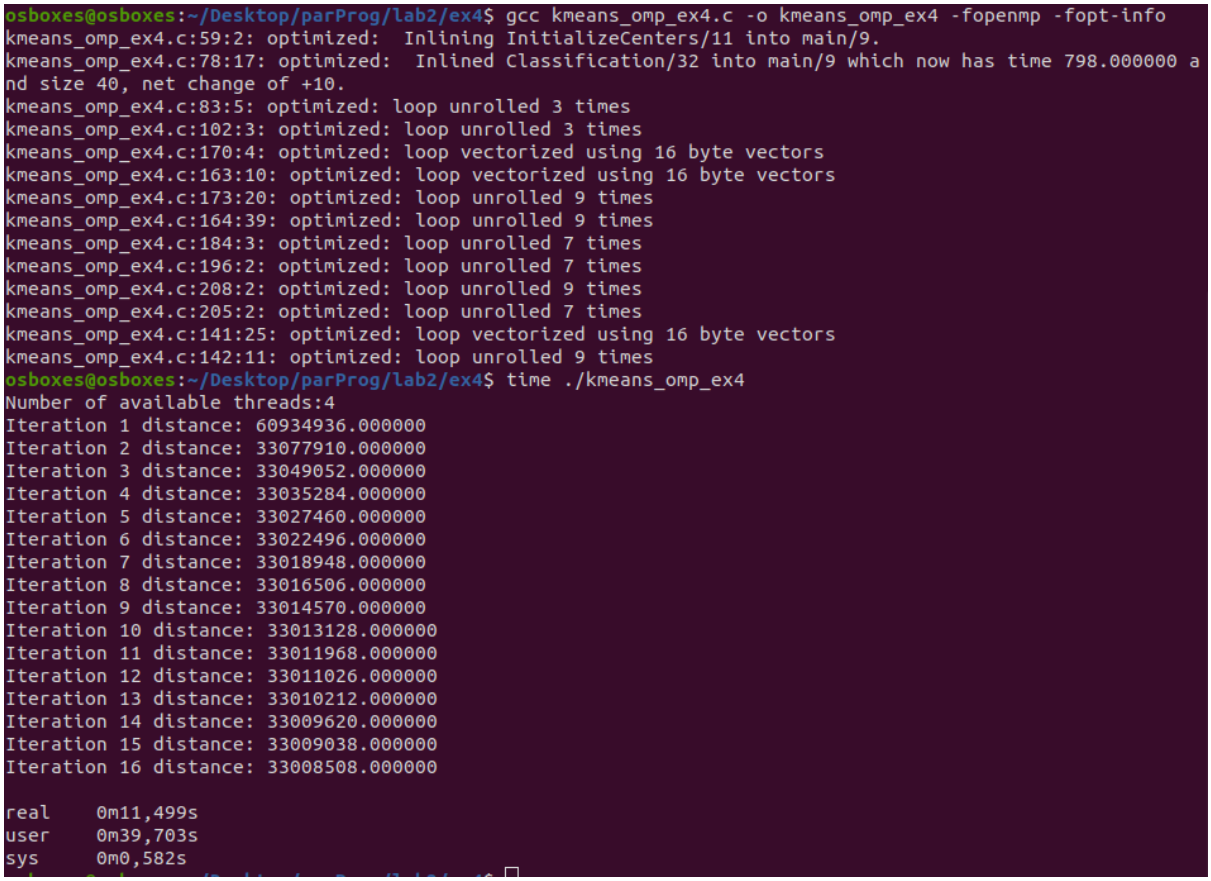
```
osboxes@osboxes:~/Desktop/parProg/lab2/ex3$ gcc kmeans_omp_ex3.c -o kmeans_omp_ex3 -fopenmp
osboxes@osboxes:~/Desktop/parProg/lab2/ex3$ time ./kmeans_omp_ex3
Number of available threads:4
Iteration 1 distance: 60935012.000000
Iteration 2 distance: 33077872.000000
Iteration 3 distance: 33049044.000000
Iteration 4 distance: 33035256.000000
Iteration 5 distance: 33027508.000000
Iteration 6 distance: 33022474.000000
Iteration 7 distance: 33018928.000000
Iteration 8 distance: 33016476.000000
Iteration 9 distance: 33014600.000000
Iteration 10 distance: 33013140.000000
Iteration 11 distance: 33011996.000000
Iteration 12 distance: 33010998.000000
Iteration 13 distance: 33010230.000000
Iteration 14 distance: 33009600.000000
Iteration 15 distance: 33009032.000000
Iteration 16 distance: 33008604.000000

real    0m11,590s
user    0m40,247s
sys     0m0,233s
osboxes@osboxes:~/Desktop/parProg/lab2/ex3$
```

Εργασία 4

Σε αυτό το σημείο θα κάνουμε χρήση της οδηγίας #pragma omp simd για να χρησιμοποιήσουμε τις δυνατότητες vector processing που μας παρέχει το OpenMP. Προσθέτουμε την παραπάνω οδηγία στο εσωτερικό for του τριπλού for που έχουμε παραλληλοποιήσει. Κάνοντας compile με την παράμετρο -fopt-info, όπως βλέπουμε παρακάτω, μπορούμε να δούμε ποιες βελτιστοποιήσεις έγιναν από τον compiler. Βλέπουμε πως σύμφωνα με το μήνυμα του compiler:

```
kmeans_omp_ex4.c:142:25: optimized: loop vectorized using 16 byte vectors
```

Στην γραμμή που μας ενδιαφέρει έγινε χρήση των wide registers. Εκτελούμε το πρόγραμμα όπως φαίενται στην παρακάτω εικόνα. Ο χρόνος δεν άλλαξε σε μεγάλο βαθμό, καθώς όπως ζητείται από την εκφώνηση κάνουμε compile με παράμετρο -O3 η οποία κάνει loop vectorization by default.



Σχόλιο: Καταληκτικά σε τη χρήση παράλληλης επεξεργασίας και του interface της OpenMp καταφέραμε να μειώσουμε σημαντικά το χρόνο εκτέλεσης της μορφής του αλγορίθμου **K-means** που υλοποιήσαμε σειριακά. Πιο συγκεκριμένα με τη χρήση **4 thread** μειώσαμε τα **41,4sec** της σειριακής σε εκτέλεσης σε περίπου **11,5sec** (η έκδοση της Εργασίας 3). Δηλαδή με τη χρήση 4 thread κάναμε το πρόγραμμα περίπου 3,6 φορές πιο γρήγορο.

ΤΕΛΟΣ