ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΜΕΝΩΝ ΣΥΣΤΗΜΑΤΩΝ – 7ο ΕΞΑΜΗΝΟ

- Ομάδα 7 -

Μαρκέτος Νικόδημος – ΑΜ:03117095

Τζομάκα Αφροδίτη – ΑΜ: 03117107

1η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

ΖΗΤΟΥΜΕΝΟ 1ο

1) Κάνοντας χρήση της εντολής *uname -a* παίρνουμε τις κάτωθι πληροφορίες:
Έκδοση λειτουργικού: Ubuntu 16.04.1 LTS
Έκδοση πυρήνα: 4.15.0
Ύστερα εκτελώντας *lscpu* και *sudo lshw -short* καταγράφουμε:
L1d cache: 32KB, ***L1i cache***: 32KB, ***L2 cache***: 256KB, ***L3 cache***: 3072KB (lscpu)
RAM: 8GB (sudo lshw -short)
Αριθμός πυρήνων: 2 / CPU socket (lscpu)
Συχνότητα ρολογιού: 495.334 MHz (lscpu)

2) Κάνοντας χρήση της συνάρτησης gettimeofday() όπως φαίνεται στο αρχείο phods2.c και εκτελώντας το runscript1_23 έχουμε τα εξής αποτελέσματα:
Max: 39839 usec, Min: 6646 usec, Average: 7633.81 usec.

3) Ο μετασχηματισμένος κώδικας υπάρχει στο αρχείο phods3.c. Σε αυτόν εφαρμόσαμε τους εξής μετασχηματισμούς:

- *Loop Fussion:* Εξετάζοντας την συνάρτηση phods_motion_estimation παρατηρούμε ότι ο κώδικας του forloop που διαχειρίζεται τα pixel του x-dimension είναι ο ίδιος με αυτόν του y-dimension επομένως συγχωνεύουμε αυτά τα δύο loop.
- *Loop Unrolling:* Στο επόμενο βήμα παρατηρούμε ότι ο κώδικας που βρίσκεται εντός του while με όρισμα το S εκτελείται σταθερά και ανεξαρτήτως συνθήκης 3 φορές (S=4, S=2, S=1). Όμοια, ο κώδικας εντός του forloop με όρισμα το i εκτελείται σταθερά και ανεξαρτήτως συνθήκης 3 φορές σε κάθε S (i = -S, i = 0, i= S). Επομένως φτιάχνουμε δύο macros, τις S_FORLOOP και I_FORLOOP με σκοπό την μείωση των branch penalties και των εντολών που εκτελούνται.
- *Data Reuse:* Στο τελικό στάδιο, αποτιμήσαμε σε μεταβλητές στοιχεία πίνακα (vectors_x[x][y], vectors_y[x][y]) και πολλαπλασιασμούς (B*x, B*y) που γίνονται πολλές φορές στον κώδικα ώστε να μειώσουμε στην

πρώτη περίπτωση τις προσπελάσεις στους πίνακες και στην δεύτερη τις πράξεις που εκτελούνται.

Ξανά λοιπόν, τρέχοντας τον κώδικα phods3.c μέσω του runscript1_23.sh παίρνουμε τις ακόλουθες μετρήσεις χρόνων εκτέλεσης:

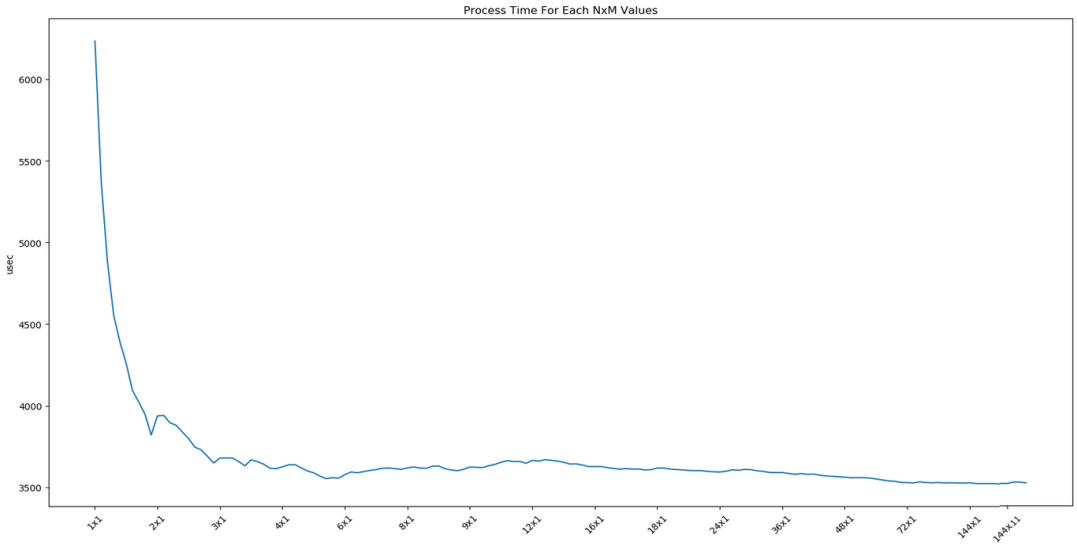
Max: 21019 usec, Min: 3085 usec, Average: 3747.26 usec

4) Μετασχηματίζουμε τον κώδικα ώστε να δέχεται ως όρισμα την τιμή του B στο αρχείο phods4.c.

Οι κοινοί διαιρέτες του N=144 και M=176 είναι οι 1, 2, 4, 8, 16, επομένως αυτές είναι και οι τιμές του B τις οποίες δοκιμάζουμε. Για τον σκοπό αυτό τρέχουμε το runscript1_4.sh το οποίο χρησιμοποιεί το best.py και ελέγχει εξαντλητικά τις περιπτώσεις αυτές εμφανίζοντας το καλύτερο B. Για τα δικά μας δεδομένα αυτό είναι το ***B = 16*** (εκείνο δηλαδή που είχε αρχικά defined ο κώδικας που δώθηκε) με μέσο χρόνο: ***3681.38*** **usec.** Το optimal B (OB), το μέγεθος δηλαδή του μπλοκ που διαχειρίζεται αποτελεσματικότερα (πιο γρήγορα) το κυρίως τμήμα κώδικα, εξαρτάται από ποικίλους παράγοντες. Αρχικά ένας από αυτούς είναι το λειτουργικό σύστημα (και οι εκδόσεις του), το οποίο, ναι, βρέθηκε στο πρώτο ερώτημα. Επίσης εκείνο που θα θέλαμε είναι να μεγιστοποιήσουμε το locality φέρνοντας στην cache τόσα δεδομένα όσα αυτή μπορεί να χωρέσει και να αποφύγουμε την σύγκρουση δεδομένων. Αυτός ο εύληπτος κανόνας έχει φυσικά διάφορες παραμέτρους που τον επηρεάζουν και έχουν να κάνουν με τα χαρακτηριστικά της cache (size, associativity, replacement policy). Από αυτά λοιπόν, το μέγεθος της cache έχει προσδιοριστεί στο ερώτημα 1.

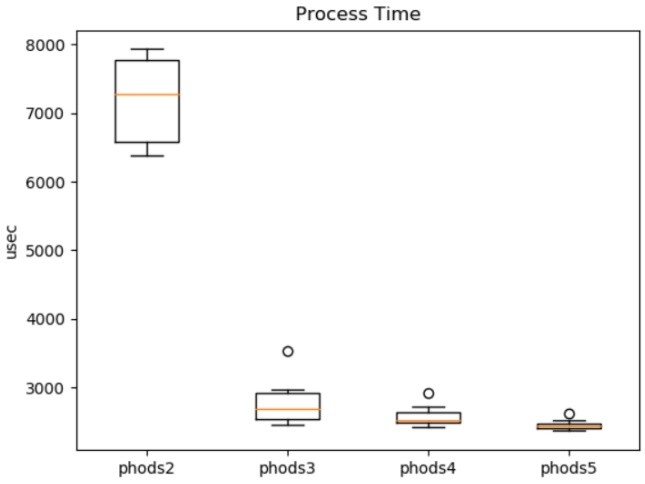
5) Όμοια με πριν, μετασχηματίζουμε τον κώδικα έτσι ώστε πρώτον να δέχεται δύο, αυτήν την φορά, ορίσματα, το Bx και το By και δεύτερον να χρησιμοποιεί αυτά κατάλληλα στους υπολογισμούς των συναρτήσεων του κυρίως κώδικα. Το μετασχηματισμένο αυτό αρχείο είναι το phods5.c.

Για Bx = διαιρέτες του N και By = διαιρέτες του M έχουμε συνολικά 15*10 = 150 συνδυασμούς. Τρέχουμε το runscript1_5.sh το οποίο χρησιμοποιεί το best.py και ελέγχει εξαντλητικά τις περιπτώσεις αυτές, βρίσκει το καλύτερο ζεύγος Bx, By και δημιουργεί το παρακάτω διάγραμμα αποτελεσμάτων:



Στην παραπάνω εικόνα βλέπουμε πως κυμαίνεται ο χρόνος εκτέλεσης για τους διαφορετικούς συνδυασμούς NxM. Ο άξονας των x είναι αριθμημένος ενδεικτικά με τιμές ix1, όπου i = (διαιρέτες του N) και με τον συνδυασμό που ελαχιστοποιεί το αποτέλεσμα. Για τα δικά μας δεδομένα αυτός είναι για ***Bx = 144, By = 11*** με μέσο χρόνο εκτέλεσης ***3498.3954667*** **usec**.
Είναι φανερό πως για να εγγυηθούμε την βελτιστότητα η εξαντλητική αναζήτηση αποτελεί αναγκαίο κακό. Ωστόσο βασιζόμενοι στο γεγονός πως η καλυτέρευση που παρατηρούμε στον χρόνο για το βέλτιστο B είναι αρκετά μικρή, θα μπορούσαμε να αρκεστούμε σε μια καλή προσέγγιση εισάγοντας κάποιον ευριστικό μηχανισμό. Ενδεικτικά αναφέρεται η δυαδική αναζήτηση.

6) Τρέχοντας το αρχείο boxplot.py παίρνουμε την εξής γραφική παράσταση:



Το boxplot διάγραμμα επαληθεύει τα δεδομένα που συλλέξαμε πρότινος τρέχοντας τα προαναφερθέντα script για τις διαφορετικές υλοποιήσεις του αλγορίθμου phods. Πιο σγκεκριμένα, βλέπουμε ότι για κάθε επόμενη υλοποίηση έχουμε και καλύτερη απόδοση τόσο στις min/max τιμές όσο και στο median αυτών. Τέλος, θα θέλαμε να σημειώσουμε την εξής παρατήρηση: οι μετρήσεις που κάναμε δεν ήταν αρκετά συνεπείς, με την έννοια του ότι διαδοχικές εκτελέσεις κώδικα υπολογισμού των επιδόσεων, έδιναν διαφορετικά αποτελέσματα. Στην πλειοψηφία η διακύμανση ήταν μικρή αλλά κάποιες φορές –για παράδειγμα στην πρώτη εκτέλεση σε σχέση με τις επόμενες– υπήρχαν μεγάλες διαφορές από τα θεωρητικά αναμενόμενα αποτελέσματα (σε αυτές τις μετρήσεις οφείλονται και οι ακραίες τιμές που σημειώνονται με κυκλάκια στο boxplot). Μια ερμηνεία για αυτό είναι ότι μετά την πρώτη εκτέλεση γεμίζει η instruction cache και έτσι, λόγω των μειωμένων fetches μειώνεται και ο χρόνος εκτέλεσης.

ΖΗΤΟΥΜΕΝΟ 2ο

1) Ο τροποποιημένος κώδικας περιέχεται στο αρχείο tables1.c. Τρέχοντας το run2_1.sh παίρνουμε τα εξής αποτελέσματα:
Max = 449837, Min = 437749 , Average = 441239.7.

2) Οι αλλαγές στον κώδικα tables_orio.c για κάθε ευριστικό αλγόριθμο φαίνονται ως σχόλια στο κομμάτι def search. Για κάθε τρόπο αναζήτησης έχουμε τα εξής αποτελέσματα:

- ***Exhaustive:*** **UF = 13**
- ***Random:*** **UF = 4**
- ***Simplex:*** **UF = 2**

Παρατηρούμε ότι τα αποτελέσματα που πήραμε διαφέρουν μεταξύ τους. Αυτό αποτελεί αναμενόμενο γεγονός που οφείλεται στην διαφορετικότητα των αλγορίθμων αναζήτησης που χρησιμοποιήθηκαν. Ο Exhaustive αλγόριθμος προφανώς βρίσκει την βέλτιστη τιμή. Ο Simplex χρησιμοποιεί ευριστικό μηχανισμό που αυξάνει μεν την απόδοση σε σχέση με τον Exhaustive, ωστόσο δεν εγγυάται την εύρεση της βέλτιστης τιμής. Ο Random επιστρέφει τυχαία τιμή επομένως με μεγάλη πιθανότητα διαφορετική από εκείνη των Exhaustive και Simplex και σίγουρα όχι εγγυημένα την βέλτιστη ή κάποια καλή προσέγγισή της.

3) Οι τροποποιημένοι κώδικες του εν λόγω ερωτήματος περιέχονται στα αρχεία tables_exhaustive.c, tables_random.c και tables_simplex.c. Για κάθε τρόπο αναζήτησης έχουμε τα εξής αποτελέσματα:

- *Exhaustive:*
  Max = 412006usecs, Min = 404942usecs, Average = 407898,1usecs.
- *Random:*
  Max = 704393usecs, Min = 403540usecs, Average = 445337,2usecs.
- *Simplex:*
  Max = 456239usecs, Min = 417519usecs, Average = 424863,1usecs.

Τα άνωθεν αποτελέσματα επιβεβαιώνουν την θεωρητική ανάλυση που κάναμε στο προηγούμενο ερώτημα. Ταυτόχρονα βλέπουμε και βελτίωση σε σχέση με τον αρχικό, μη βελτιστοποιημένο κώδικα.