ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΗΛΕΚΤΡΟΝΙΚΗΣ

Σχεδιασμός Ενσωματωμένων Συστημάτων

9ο εξάμηνο

1η Εργαστηριακή Άσκηση:

ΒΕΛΤΙΣΤΟΠΟΊΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΧΑΜΗΛΗ ΚΑΤΑΝΑΛΩΣΗ ΕΝΕΡΓΕΙΑΣ ΚΑΙ ΥΨΗΛΗ ΑΠΟΔΟΣΗ

03114681: Maliganis Nikolaos
03116112: Gezekelian Viken

Ζητούμενο 1ο – Loop Optimizations & Design Space Exploration

Ερώτημα 1ο:

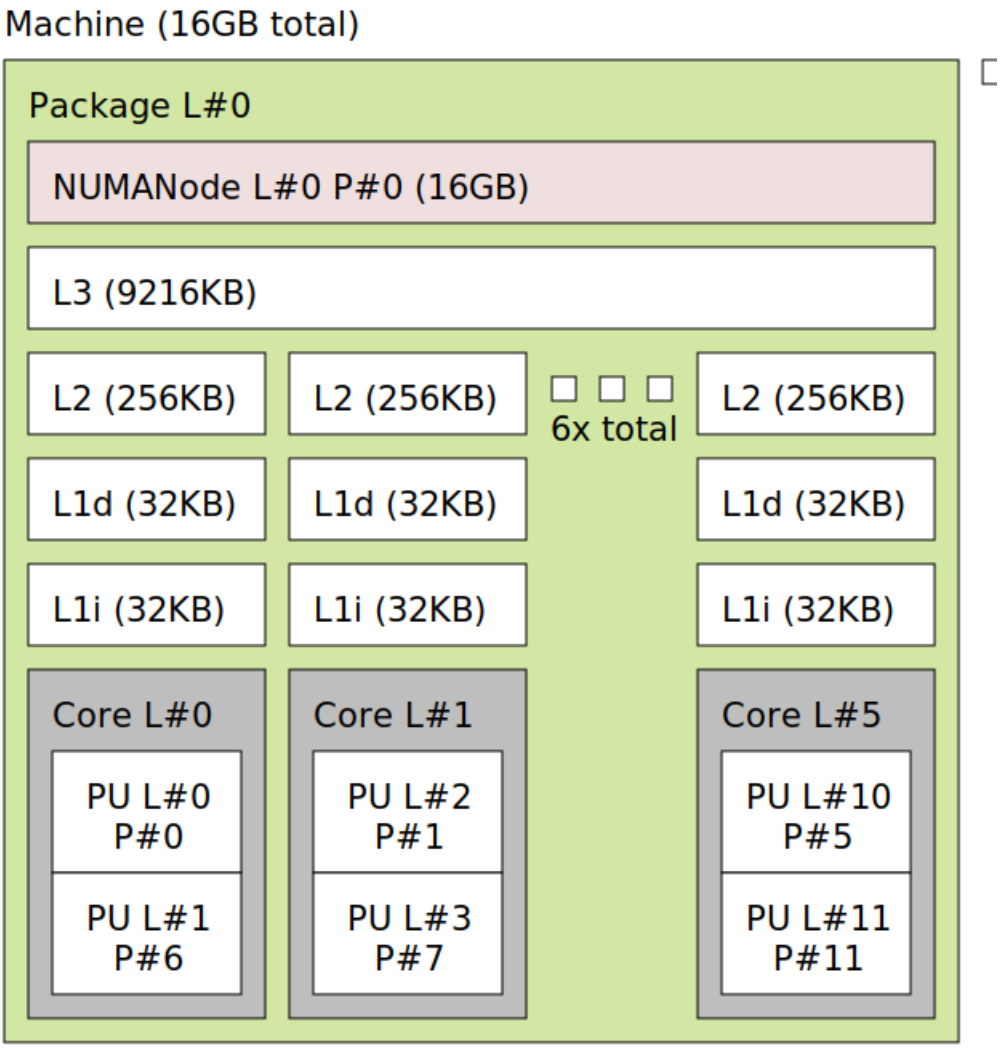
Έκδοση Λειτουργικού: Ubuntu 20.04.1 LTS (lsb_release -a)
Έκδοση πυρήνα: Linux 5.4.0-53-generic (uname -a)
Με lscpu βρίσκω:

CPU(s): 12
On-line CPU(s) list: 0-11
Thread(s) per core: 2
Core(s) per socket: 6
Socket(s): 1

L1i cache: 192 KiB
L1d cache: 192 KiB
L2 cache: 1,5 MiB
L3 cache: 9 MiB
CPU MHz: 800.406
CPU max MHz: 4100,0000
CPU min MHz: 800,0000

Από την lscpu λοιπόν, βρίσκουμε πως ο υπολογιστής μας διαθέτει 6 πυρήνες, όπου ο καθένας μπορεί να εξυπηρετεί ταυτόχρονα 2 threads. Επίσης, βλέπουμε πως (συνολικά) διαθέτει: 192 Kib L1 instruction και 192Kib L1 data cache, 1,5 Mib L2 cache και 9Mib L3 cache. Επίσης, μπορούμε να δούμε πως η μέγιστη συχνότητα του επεξεργαστή είναι τα 4,1 MHz.

Με χρήση της εντολής lstopo(από το πακέτο hwloc), μπορούμε να δούμε με μεγαλύτερη λεπτομέρεια την τοπολογία του επεξεργαστή μας. Όπως φαίνεται και παρακάτω, βλέπουμε πως υπάρχουν:
6 L1 instruction caches, μεγέθους 32 KB, μία για κάθε πυρήνα του επεξεργαστή
6 L1 data caches, μεγέθους 32 KB, μία για κάθε πυρήνα του επεξεργαστή
6 L2 caches, μεγέθους 256 KB, μία για κάθε πυρήνα του επεξεργαστή
6 L3 caches, μεγέθους 9216 KB, την οποία μοιράζονται και οι 6 πυρήνες του επεξεργαστή



Μέσω της εντολής sudo dmidecode --type 17, μπορούμε να βρούμε πληροφορίες και για τις RAM του υπολογιστή μας. Βρίσκουμε λοιπόν πως ο υπολογιστής μας περιέχει 2 πανομοιότυπα sticks RAM, των 8 GB το καθένα, τεχνολογίας DDR4 και ταχύτητας 2667 MT/s(πρακτικά δηλαδή 1333*2=2667 MHz λόγω ddr)

Total Width: 64 bits
Data Width: 64 bits
Size: 8192 MB
Type: DDR4
Type Detail: Synchronous
Speed: 2667 MT/s

Τέλος, μέσω της εντολής getconf -a | grep CACHE, θα πάρουμε και πιο συγκεκριμένες πληροφορίες που αφορούν το associativity και το μέγεθος γραμμής(block) της κάθε cache:

```
LEVEL1_ICACHE_SIZE          32768
LEVEL1_ICACHE_ASSOC         8
LEVEL1_ICACHE_LINESIZE      64
LEVEL1_DCACHE_SIZE          32768
LEVEL1_DCACHE_ASSOC         8
LEVEL1_DCACHE_LINESIZE      64
LEVEL2_CACHE_SIZE           262144
LEVEL2_CACHE_ASSOC          4
LEVEL2_CACHE_LINESIZE       64
LEVEL3_CACHE_SIZE           9437184
LEVEL3_CACHE_ASSOC          12
LEVEL3_CACHE_LINESIZE       64
```

Ερώτημα 2ο:

Αρχικά, όσον αφορά τη λειτουργία του κώδικα που έχουμε γράψει για τα παρακάτω ερωτήματα:

Έχουμε δημιουργήσει το βασικό script για την λειτουργία του κώδικα, το main.sh, δουλειά του οποίου είναι να δέχεται ως argument το script(ή υποερώτημα) που θέλουμε να εκτελέσουμε κάθε φορά. Πέραν του main.sh, τά υπόλοιπα scripts βρίσκονται εντός του φακέλου Scripts, και το καθένα(πέραν του 1.6.sh που δημιουργεί το boxplot που ζητείται) δέχεται ως argument πόσες φορές θέλουμε να τρέξουμε τον κώδικα του phods(το argument αυτό το περνάμε μέσω του main.sh). Ακόμη, στον ίδιο φάκελο υπάρχουν και μερικές βοηθητικές συναρτήσεις σε python, μία για τον υπολογισμό του μέσου όρου των Execution Times, μία για την εύρεση των βέλτιστων συνδυασμών Bx, By στο ερώτημα 1.5 και μία για τη δημιουργία του Boxplot. Λ.χ. για το συγκεκριμένο ερώτημα τρέχουμε ./main.sh 1.2 10

Τα αποτελέσματα που προκύπτουν από τη μέτρηση, είναι τα εξής(1.2_mean.txt):

Minimum: 5146
Maximum: 6704
Mean: 5425.0

Βλέπουμε λοιπόν πως ο ελάχιστος χρόνος εκτέλεσης του phods είναι τα 5.146 μsecs, ο μέγιστος τα 6.704 μsecs, ενώ ο μέσος όρος ανέρχεται στα 5.425 μsecs. Βέβαια, τρέχοντας τον κώδικα για μερικές χιλιάδες επαναλήψεις, βλέπουμε ακόμη μεγαλύτερες αποκλίσεις ελάχιστου και μέγιστου χρόνου εκτέλεσης, ενώ ο μέσος όρος του χρόνου έρχεται πιο κοντά στα 5.300 μsecs.

Ερώτημα 3ο:

Στο ερώτημα αυτό, ζητούνται μετατροπές επί του αρχικού κώδικα του phods. Το τροποποιημένο αρχείο είναι το phods.c.

Ξεκινώντας από τη σημαντικότερη μετατροπή που έχουμε κάνει στον κώδικα, έχουμε το loop merging των δύο τριπλών loops, που στον αρχικό κώδικα ξεκινούσαν στις γραμμές 90 και 123

αντίστοιχα. Το όφελος από την κίνηση αυτή είναι προφανές, και ανέρχεται στα (κατά μέσο όρο) 500 μsecs, γεγονός λογικό αφού μειώνουμε κατά το ήμισυ τον αριθμό των επαναλήψεων. Οι υπόλοιπες αλλαγές που έχουμε πραγματοποιήσει, καταφέρνουν να μειώσουν τον μέσο χρόνο εκτέλεσης κατα μερικά μονάχα μsec(τρέξαμε τον κώδικα μερικές χιλιάδες φορές κάθε φορά, ώστε να μειώσουμε την επιρροή των εξωτερικών παραγόντων στους μέσους χρόνους εκτέλεσης). Συγκεκριμένα, δηλώσαμε μια μεταβλητή εντός της void phods_motion_estimation, στην οποία αναθέσαμε την τιμή 255*B*B, καθώς η συγκεκριμένη τιμή υπολογιζόταν 2 φορές εντός 3 loops, και ανατειθόταν στα min1 και min2. Στον κώδικα μας λοιπόν, αντικαταστήσαμε τους υπολογισμούς αυτούς με αναθέσεις. Ακόμη, εφόσον η τιμή της μεταβλητής p1 δεν μεταβάλλεται εντός κάθε ξεχωριστού loop, χρησιμοποιήσαμε την ίδια μεταβλητή, για την εκτέλεση και των 2 loop που κάναμε merge προηγουμένως.
Πέραν των αλλαγών αυτών, δοκιμάσαμε και κάποιες μετατροπές οι οποίες είτε δεν φάνηκαν να επηρεάζουν τον χρόνο εκτέλεσης(loop reversal στο for(i=-S; i<S+1; i+=S)), είτε αύξησαν τελικά τον χρόνο εκτέλεσης του κώδικα(loop reversal στα for(x=0; x<N/B; x++) και for(y=0; y<M/B; y++)).
Παρακάτω, φαίνονται τα αποτελέσματα που προκύπτουν απο την μέτρηση(1.3_mean.txt):
Minimum: 4685
Maximum: 4883
Mean: 4784.5

Βλέπουμε λοιπόν πως ο ελάχιστος χρόνος εκτέλεσης του phods_motion_estimation είναι τα 4.685 μsecs, ο μέγιστος τα 4.883 μsecs, ενώ ο μέσος όρος ανέρχεται στα 4.784 μsecs. Παρατηρούμε αρχικά λοιπόν πως ο χρόνος εκτέλεσης του κώδικα παρουσιάζει μια μείωση της τάξεως του ~12%, ποσοστό σημαντικό εάν αναλογιστούμε πως δεν χρειάστηκε να εφαρμόσουμε κάποια από τις πιο σύνθετες τεχνικές για την επίτευξη του αποτελέσματος αυτού.

Ερώτημα 4ο:
Στο ερώτημα αυτό, έχουμε αρχικά δημιουργήσει το αρχείο 1.4.py το οποίο είναι υπεύθυνο για την αντικατάσταση της γραμμής #define B 16 του phods.c με την κατάλληλη τιμή. Καλώντας λοιπόν το 1.4.sh μέσω της main, το script αυτόματα ορίζει τις κατάλληλες τιμές στον κώδικα, κάνει compile, το τρέχει και έπειτα υπολογίζει τον μέσο όρο των επαναλήψεων που ορίσαμε. Εφόσον ζητήθηκε να θέσουμε στο B τιμές που συνιστούν κοινό διαιρέτη των M και N, δώσαμε τις τιμές 1, 2, 4, 8, 16. Τα πλήρη αποτελέσματα υπάρχουν εντός των αρχείων 1.4_XX_mean.txt, όπου XX οι πιθανές τιμές του B. Ακόμη, για να είναι πιο αντιπροσωπευτικά τα αποτελέσματα, τρέξαμε τον κώδικα μερικές εκατοντάδες φορές(στα αρχεία που υποβάλαμε, υπάρχουν, όπως ζητούνται, 10 επαναλήψεις). Με βάση τα αποτελέσματα, βλέπουμε πως ο ορισμούς του B σε 1 ή 2, παρουσιάζει δραματική αύξηση στον χρόνο εκτέλεσης του κώδικα. Αντιθέτως, τα αποτελέσματα όταν θέσαμε την τιμή του B ίση με 4 ή 16(όπως ήταν δηλαδή αρχικά) είναι σε μεγάλο βαθμό συγκρίσιμα. Τέλος, το καλύτερο αποτέλεσμα παρουσιάζεται για B=8, που φαίνεται και παρακάτω:
Minimum: 4761
Maximum: 4845
Mean: 4784.2

Όσον αφορά το κοινό σημείο μεταξύ Hardware και της τιμής 8, θα δούμε λίγο πιο προσεκτικά την L1d cache του επεξεργαστή μας. Παρατηρούμε, πως έχουμε line size ίσο με 64 bytes, και 8-way associative μνήμη. Πρακτικά, ο κώδικας του phods, χρησιμοποιεί το μέγεθος του B για να ορίσει τους πίνακες που χρησιμοποιούνται(προκύπτουν πίνακες μεγέθους 18x22) και έπειτα τους διατρέχει για B επαναλήψεις. Πιθανώς λοιπόν, η βελτίωση αυτή που βλέπουμε να οφείλεται στο γεγονός πως χωράνε 16 στοιχεία ανά γραμμή της cache, μειώνοντας τον αριθμό των blocks που πρέπει να έρθουν από τα μεγαλύτερα επίπεδα της cache L2 και L3.

Ερώτημα 5ο:

Στο ερώτημα αυτό, έχουμε εργαστεί ομοίως με προηγουμένως, όσον αφορά τον τρόπο εκτέλεσης του κώδικα, με τη διαφορά πως έχουμε δημιουργήσει το αρχείο phods_1.5.c και χρησιμοποιούμε τον κώδικα 1.5.py για τις απαραίτητες μετατροπές. Ακόμη, έχουμε και το αρχείο 1.5_best_search.py, το οποίο αναζητά όλα τα αρχεία τα οποία δημιουργούνται και εμφανίζει τους 3 καλύτερους χρόνους, καθώς και τις τιμές των Bx, By για τις οποίες οι τιμές αυτές εμφανίζονται. Για διάφορες τιμές των Bx και By, παρατηρούμε τραγικές αυξήσεις στον χρόνο εκτέλεση του κώδικα, οι βέλτιστες όμως τιμές Bx, By είναι 1 και 88 αντίστοιχα, για τις οποίες οι χρόνοι εκτέλεσης είναι:
Minimum: 3626
Maximum: 3950
Mean: 3737.51

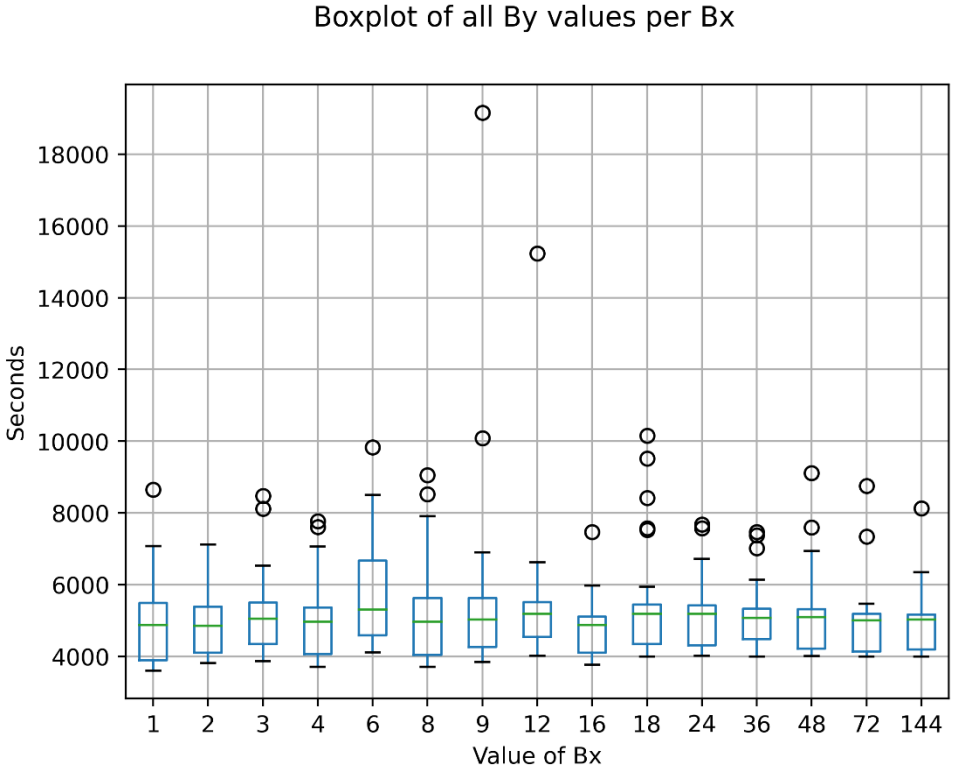
Ακολουθούν οι τιμές Bx=1, By=176, με μέσο χρόνο εκτέλεσης 3742.92 seconds, και Bx=8, By=176, με χρόνο εκτέλεσης 3832.04 seconds
Για τον περιορισμό της αναζήτησης, μπορούμε να παρατηρήσουμε τις τιμές που εμφανίζουν τις μέγιστες τιμές(1.5_worst_search.py), οι οποίες είναι για:
Ο χειρότερος συνδυασμός είναι για Bx=1 και By=1, με μέσο χρόνο εκτέλεσης 6516.92 seconds, ενώ ακολουθούν:
Bx=2, By=1, με μέσο χρόνο εκτέλεσης 5774.89 seconds
Bx=3, By=1, με μέσος χρόνο εκτέλεσης 5749.78 seconds

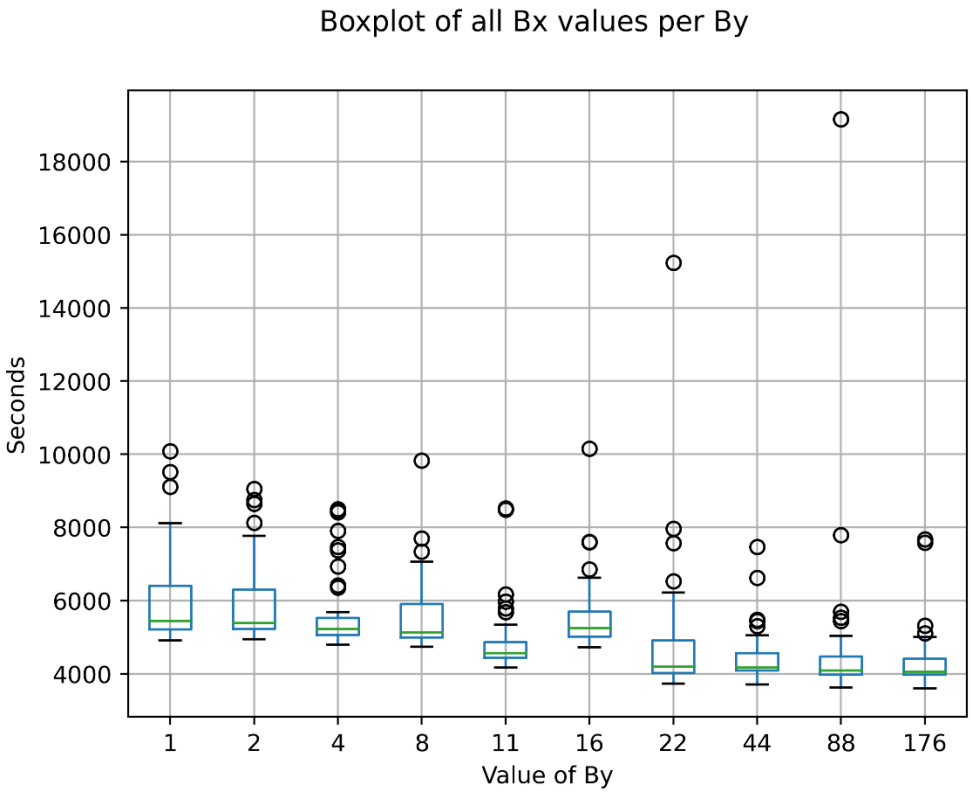
Παρατηρούμε πως φαίνεται οι χειρότερες επιδόσεις να επιτυγχάνονται στις τιμές όπου το Bx και By έχουν μικρή τιμή, όπως γινόταν και στο παραπάνω ερώτημα, δημιουργώντας μικρά σε μέγεθος blocks. Ακόμη, βλέποντας και τα υπόλοιπα αποτελέσματα, παρατηρούμε πως καλύτερες, γενικότερα, επιδόσεις επιτυγχάνονται για τους συνδυασμούς όπου η τιμή του By είναι μεγαλύτερη της τιμής του Bx. Καθώς είναι δύσκολο όμως να συγκριθούν σωστά όλα τα αποτελέσματα καθώς υπάρχουν πολλά αρχεία, ακολουθούν παρακάτω διαγράμματα που θα βοηθήσουν στην ανάλυση.

Ερώτημα 6ο:

Στο ερώτημα αυτό, έχουμε φτιάξει 4 διαφορετικά boxplots, που θα μας βοηθήσουν στην παρουσίαση των παραπάνω αποτελεσμάτων. Για τα boxplots μας λοιπόν, η δομή του κώδικα είναι σε μεγάλο βαθμό παρόμοια. Αρχικά, διαβάζουμε από τα αρχεία που περιέχουν τις ελάχιστες, μέσες και μέγιστες τιμές runtime τα δεδομένα, και τα βάζουμε κατάλληλα σε πίνακες που περιέχουν τον χρόνο εκτέλεσης, τον τύπο του(μέσος όρος, ελάχιστη τιμή, μέγιστη τιμή), το υποερώτημα στο οποίο ανήκουν και συγκεκριμένα για τα boxplots που αφορούν το υποερώτημα 1.5 τις τιμές των Bx και By.

Παραθέτουμε λοιπόν αρχικά τα δύο διαγράμματα που αφορούν το ερώτημα 1.5:

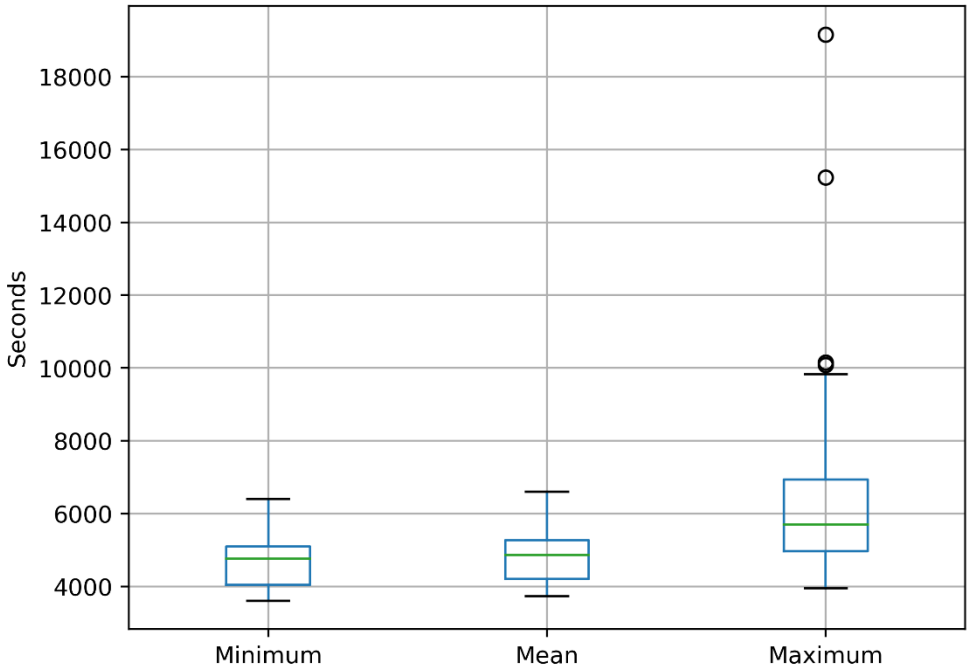




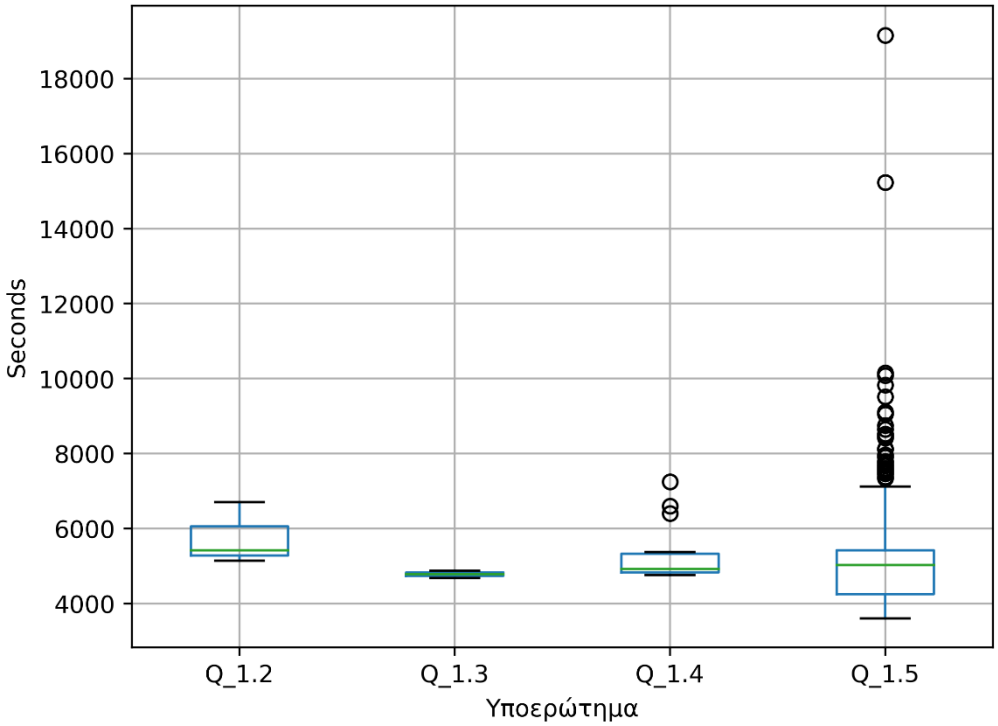
Τα Boxplots αυτά, κάνουν group τα δεδομένα αναλόγως με τις τιμές των Bx και By, περιλαμβάνοντας για την κάθε τιμή όλες τις δυνατές τιμές της άλλης μεταβλητής. Από αυτά λοιπόν, βλέπουμε ξεκάθαρα πως ο χρόνος εκτέλεσης φαίνεται να βελτιώνεται με την αύξηση του By($2^{ο}$ διάγραμμα), με μοναδική εξαίρεση την τιμή 11, η οποία παρουσιάζει καλύτερους μέσους χρόνους εν σχέσει με την ακριβώς μικρότερη και μεγαλύτερη. Και πάλι όμως, οι βέλτιστες φαίνονται να είναι οι τιμές 88 και 176. Όσον αφορά την τιμή του Bx, βλέπουμε πως τα αποτελέσματα δεν είναι τόσο ξεκάθαρα, καθώς βλέπουμε σημαντικές αυξομειώσεις σε κάθε τιμή, με τις τιμές που να φαίνονται πως έχουν τα βέλτιστα αποτελέσματα(κατά μέσο όρο) να είναι οι τιμές 1, 4, 8 και 16. Τέλος, να σημειώσουμε πως οι κουκκίδες που βλέπουμε στα διαγράμματα αφορούν τιμές οι οποίες δεν συμπεριλαμβάνονται στην πραγματικότητα στον υπολογισμό του Boxplot, καθώς θεωρούνται «εξωτερικά» σημεία, λόγω των μεγάλων αποκλίσεων τους σε σχέση με τα υπόλοιπα δεδομένα. Άλλωστε, γνωρίζουμε πως και στην πραγματικότητα είναι πιθανό να αφορούν κάποιο συγκεκριμένο, πολύ μεγάλο χρόνο εκτέλεσης που καταγράφηκε ως μέγιστος, και πιθανώς το μέγεθος να οφείλεται σε κάποιο εξωτερικό παράγοντα που καθυστέρησε την εκτέλεση.

Στη συνέχεια, ακολουθούν άλλα δύο boxplots, τα οποία λαμβάνουν υπόψιν το σύνολο των δεδομένων, και κάνουν group των δεδομένων είτε με βάση τον τύπο του χρόνου εκτέλεσης που καταγράφηκε, είτε το υποερώτημα το οποίο αφορούν. Τα διαγράμματα φαίνονται παρακάτω:

Boxplot of Minimum, Mean and Maximum Values



Boxplot of all values per Question



Στο πρώτο διάγραμμα, έχουμε πάρει τιμές από τα αποτελέσματα όλων των υποερωτημάτων. Στο διάγραμμα αυτό, μπορούμε να δούμε αρχικά πως για όλα τα ερωτήματα, οι ελάχιστοι χρόνοι εκτέλεσης φαίνεται να είναι πολύ κοντά στους πραγματικούς. Ακόμη, χάριν στο boxplot μπορούμε να δούμε πως οι μέσοι χρόνοι εκτέλεσης, οι οποίοι βέβαια επηρεάζονται σε πολύ μεγάλο βαθμό από τα ερωτήματα 1.4 και 1.5 λόγω του όγκου των δεδομένων που παίρνουμε από αυτά, είναι ισότιμα κατανεμημένοι, με τον μέσο όρο να βρίσκεται λίγο πάνω από τη μέση του box. Αντιθέτως, στο boxplot των ελάχιστων τιμών βλέπουμε πως η μέση τιμή βρίσκεται στο άνω τέταρτο του box, γεγονός που σημαίνει πως υπάρχουν κάποιοι συνδυασμοί που παρουσιάζουν σημαντικά μικρότερους ελάχιστους χρόνος εκτέλεσης σε σχέση με τους υπόλοιπους.

Στο δεύτερο διάγραμμα, μπορούμε να δούμε τους μέσους χρόνους εκτέλεσης ανά υποερώτημα. Αρχικά, παρατηρούμε πως το box του ερωτήματος 1.3 φαίνεται να είναι συμπιεσμένο εν σχέσει με τα υπόλοιπα. Το γεγονός αυτό, σηματοδοτεί πως δεν υπήρξαν σημαντικές αποκλίσεις της ελάχιστης και μέγιστης τιμής κατά την εκτέλεση του κώδικα, πράγμα που πιθανώς οφείλεται στην βελτιστοποίηση του κώδικα που πραγματοποιήσαμε. Ακόμη, σημαντικό είναι το γεγονός πως το box του υποερωτήματος 1.5, πέραν ορισμένων μέγιστων τιμών που δεν προσμετρώνται στο διάγραμμα, παρουσιάζει μέγιστο, περίπου στο ίδιο σημείο με το box του ερωτήματος 1.2. Το γεγονός αυτό σημαίνει πως ακόμη και με τον χειρότερο δυνατό συνδυασμό blocks, και πάλι φτάνουμε στην χειρότερη περίπτωση να έχουμε χρόνους συγκρίσιμους με αυτούς του αρχικού κώδικα, προτού βελτιστοποιηθεί, και μας δείχνει τη σημασία της βελτιστοποίησης που πραγματοποιήσαμε. Τέλος, στο box του ερωτήματος 1.4 βλέπουμε πως η αλλαγή του μεγέθους του block size δεν φαίνεται να έφερε σημαντική βελτίωση στον χρόνο εκτέλεσης, γεγονός όμως που οφείλεται στο ότι το αρχικό block size 16, ήταν μια ήδη καλή επιλογή.

Ζητούμενο 2ο – Αυτοματοποιημένη βελτιστοποίηση κώδικα

Ερώτημα 1ο:

Στο ερώτημα αυτό του ζητήματος 2ο της εργαστηριακής άσκησης θα πρέπει να χρησιμοποιήσουμε το εργαλείο Orio προκειμένου να πραγματοποιήσουμε αυτοματοποιημένη βελτιστοποίηση κώδικα. Αναλυτικότερα, αναζητούμε το βέλτιστο Unroll Factor για δεδομένα for loop 1.000.000 επαναλήψεων, εφαρμόζοντας διαφορετικές μεθόδους αναζήτησης με τη βοήθεια του Orio.

Στο ερώτημα αυτό, όπως και προηγούμενος, θα χρησιμοποιήσουμε εκ-νέου την συνάντηση **gettimeofday** η οποία εμπεριέχεται στο `<sys/time.h>`

```
    /* Added gettimeofday support  ---> */
    gettimeofday(&tf,NULL);
    time=((tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec)) *
0.000001;
    printf("%lf\n", time);
    /* <--- Added gettimeofday support */
```

1. Snippet Code

Η time.c έχει εμπλουτιστεί με το παραπάνω snippet code έτσι ώστε να μας παρέχει στην έξοδο.

Εκτελώντας τον κώδικα το Make και τρέχοντας το κατάλληλο Python Script, έχουμε ως αποτέλεσμα:

Minimum	Maximum	Average
0.233782 sec	0.24755 sec	0.242166 sec

Παρατήρηση: το Makefile έχει προστεθεί ο όρος -mcmodel=large, έτσι ώστε να δεσμεύσουμε μνήμη για τον πίνακα των 100.000.000 θέσεων που δίνεται στον προς βελτιστοποίηση κώδικα.

Ερώτημα 2ο:

Για το exhaustive search, το Unrolling Factor είναι 3, για το random search το Unrolling Factor είναι 5 και για το simplex search, 4 αντίστοιχα.

Συνεπώς έχουμε:
Για exhaustive:

```
$ sudo orcc tables_orio_exhaustive.c
```

```
  for (i=0; i<=n-3; i=i+3) {
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i];
    y[(i+1)]=y[(i+1)]+a1*x1[(i+1)]+a2*x2[(i+1)]+a3*x3[(i+1)];
    y[(i+2)]=y[(i+2)]+a1*x1[(i+2)]+a2*x2[(i+2)]+a3*x3[(i+2)];
  }
  for (i=n-((n-(0))%3); i<=n-1; i=i+1)
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i];
}
```

2.exhaustive UF = 3

Για random:

```
$ sudo orcc tables_orio_random.c
```

```
   for (i=0; i<=n-5; i=i+5) {
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i];
    y[(i+1)]=y[(i+1)]+a1*x1[(i+1)]+a2*x2[(i+1)]+a3*x3[(i+1)];
    y[(i+2)]=y[(i+2)]+a1*x1[(i+2)]+a2*x2[(i+2)]+a3*x3[(i+2)];
    y[(i+3)]=y[(i+3)]+a1*x1[(i+3)]+a2*x2[(i+3)]+a3*x3[(i+3)];
    y[(i+4)]=y[(i+4)]+a1*x1[(i+4)]+a2*x2[(i+4)]+a3*x3[(i+4)];
  }
  for (i=n-((n-(0))%5); i<=n-1; i=i+1)
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i];
```

3.random UF = 5

Για simplex:

```
$ sudo orcc tables_orio_simplex.c
```

```
  for (i=0; i<=n-4; i=i+4) {
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i];
    y[(i+1)]=y[(i+1)]+a1*x1[(i+1)]+a2*x2[(i+1)]+a3*x3[(i+1)];
    y[(i+2)]=y[(i+2)]+a1*x1[(i+2)]+a2*x2[(i+2)]+a3*x3[(i+2)];
    y[(i+3)]=y[(i+3)]+a1*x1[(i+3)]+a2*x2[(i+3)]+a3*x3[(i+3)];
  }
  for (i=n-((n-(0))%4); i<=n-1; i=i+1)
    y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i];
```

4.simplex UF = 4

Ενσωματώνοντας το snippet code στο αρχείο tables.c παίρνουμε τα αρχεία tables_exhaustive.c, tables_random.c και tables_simplex.c

Πιο συγκεκριμένα και αναφορικά με τις 3 διαφορετικές αλγοριθμικές προσεγγίσεις βελτιστοποίησης:

- **Random:** Χρειάζεται λιγότερο χρόνο ολοκλήρωσης της αναζήτησης, εξαιτίας του τυχαίου τρόπου επιλογής factor.
- Simplex: Χρειάζεται έναν ενδιάμεσο χρόνο εκτέλεσης αναζήτησης, εξαιτίας του ευριστικού τρόπου λύσης για την επιλογή απόφασης
- Exhaustive: Εκτελεί εξαντλητικό τρόπο αναζήτησης και για αυτό θεωρείται καλύτερη επιλογή, εξαιτίας της επιλογής του βέλτιστου unroll.

Ερώτημα 3ο:

Εκτελώντας τον κώδικα το Make και τρέχοντας το κατάλληλο Python Script, έχουμε ως αποτέλεσμα:

Operation	Minimum	Maximum	Average
Normal	0.233782 sec	0.24755 sec	0.242166 sec
Exhaustive	0.140655 sec	0.149229 sec	0.143717 sec
Simplex	0.162471 sec	0.173756 sec	0.168284 sec
Random	0.173902 sec	0.181233 sec	0.177554 sec

Όπως παρατηρούμε υπάρχει μια βελτίωση εξαιτίας της χρήσης του Loop Unrolling, κυρίως στη χρήση του exhaustive ωστόσο όχι σηματική η διαφορά μεταξύ του UF =3, UF=4 και UF=5. Random ορθά έχει τον χειρότερο χρόνο εξαιτίας της τυχαιότητας του αλγορίθμου και από τις 3 και Simplex αποτελεί μια μέση επιλογή βελτιστοποίησης από τις 3. Η Exhaustive Search, εξαιτίας της εξαντλητικής αναζήτηση που εφαρμόζει ο αλγοριθμος ελέγχει όλους τους δυνατούς unroll factors δύνοντας έτσι το βέλτιστο unroll για τον αρχικό βρόχο.