ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Επανασχεδίαση, επεκτασιμότητα και ανάπτυξη βιβλιοθήκης επίλυσης προβλημάτων ψηφιακής συνδυαστικής λογικής

ΓΕΩΡΓΙΑΔΗΣ ΧΡΗΣΤΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

ΔΑΔΑΛΙΑΡΗΣ ΑΝΤΩΝΙΟΣ
Επίκουρος Καθηγητής

Λαμία 14 Ιουλίου έτος 2023

UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Redesign, expandability and development of library that solves digital combinational logic problems

GEORGIADIS CHRISTOS

FINAL THESIS

ADVISOR

DADALIARIS ANTONIOS
Assistant Professor

Lamia July 14 year 2023

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: 14/7/2023

Ο – Η Δηλ.

Γεωργιάδης Χρήστος

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Η παρούσα διπλωματική εργασία έχει ως στόχο την μεταφορά βιβλιοθήκης επίλυσης προβλημάτων συνδυαστικής λογικής, από την γλώσσα Python στην C. Τα εργαλεία που δημιουργήθηκαν ως αποτέλεσμα της εργασίας μπορούν να βοηθήσουν κάποιον/ποια που έχει ανάγκη να καταλάβει τα προβλήματα της ψηφιακής συνδυαστικής λογικής δίνοντας τα αποτελέσματα στα αναφερόμενα προβλήματα είτε είναι φοιτητής/τρια είτε έχει άπλα ενδιαφέρον για τα περιεχόμενα του μαθήματος της Λογικής Σχεδίασης. Μετά την δημιουργία της βιβλιοθήκης γίνεται ανάλυση των δύο γλωσσών Python και C για τις διαφορές που έχουν σε επιδόσεις και φιλοσοφία γραφής κώδικα. Η εργασία πέρα από το ότι περιλαμβάνει λύσεις για τα προβλήματα της συνδυαστικής λογικής, αποτελεί και μια εξερεύνηση στο κόσμο του παράλληλου κώδικα με εισαγωγή στις έννοιες και τεχνικές που απαιτούνται και μετέπειτα με πρακτικά παραδείγματα πάνω στην βιβλιοθήκη.

ABSTRACT

The present thesis has the aim to transfer a library that solves digital combinational problems written in Python to one written in C. The tools created as a result of this thesis aim to help those that try to understand the problems presented by the course Logical Design if they are students following a course at any university or if they are just interested to the specific problems by giving them the results to said problems. After the creation of the library analysis is given between the difference in the two languages Python and C, in regard to their speed and way of writing code. Besides solving the problems presented by Logical Design this thesis also delves into the world of parallel computing by explaining all the concepts and techniques necessary to implement parallelism then incorporating the knowledge gained to try and parallelize the code that makes the library.

Table of Contents

1.Εισαγωγή

(1.1 Στόχος)

Η παρούσα διπλωματική εργασία έχει στόχο την μεταφορά βιβλιοθήκης επίλυσης προβλημάτων συνδυαστικής λογικής, από την προγραμματιστική γλώσσα Python στην C.

Καθώς η νέα βιβλιοθήκη πρέπει να παρουσιάζει τις ίδιες λειτουργικότητες σε σχέση με προηγούμενες παραπλήσιες προσπάθειες, και να γίνει καταγραφή των διαφορών των δύο γλωσσών, αλλαγές θα γίνουν όπου γίνεται για να βελτιωθεί η βιβλιοθήκη σε επιδόσεις, λειτουργικότητα, χρησιμότητα και ακρίβεια.

Οι μέθοδοι που υλοποιήθηκαν στην παρούσα βιβλιοθήκη λύνουν τα ίδια προβλήματα που αντιμετωπίζουν όλοι οι φοιτητές της Πληροφορικής στις αρχές της φοίτησης τους, σε μαθήματα όπως η Λογική Σχεδίαση/Ψηφιακή Σχεδίαση. Το όνομα του μαθήματος δεν έχει σημασία γιατί τα προβλήματα που υλοποιήθηκαν μέθοδοι και θα συζητήσουμε είναι βασικά για την κατανόηση της λειτουργίας των υπολογιστών και είναι απαραίτητο προαπαιτούμενο για όποιον/οια ασχοληθεί με την Πληροφορική.

Τα προβλήματα που παρέχονται λύσεις είναι **:**

- Ο υπολογισμός αναπτύγματος για δυαδικούς, οκταδικούς, δεκαδικούς, δεκαεξαδικούς αριθμούς.
- Ο υπολογισμών των βασικών πυλών (**AND, NAND, OR, XOR, XNOR, NOT)**. υπολογισμός συμπληρώματος ως προς ένα και ως προς δύο.
- Η μετατροπή δυαδικών αριθμών σε κώδικες (**2-4-2-1, BCD, Biquinary, Gray).**
- υπολογισμός πινάκων αληθείας μέχρι 4 μεταβλητές.
- Η μετατροπή από άθροισμα ελαχιστόρων σε γινόμενο μεγιστόρων και το αντίστροφο.
- Η ελαχιστοποίηση λογικών εκφράσεων με την χρήση του **χάρτη Karnaugh** και την μέθοδο **Quine McCluskey.**

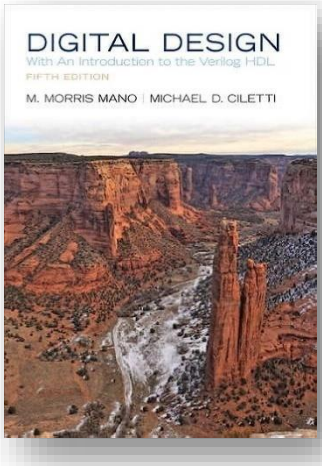
Αφού υλοποιήθηκαν οι μέθοδοι εξερευνήθηκε η πιθανότητα για την παραλληλοποίηση της μεθόδων της βιβλιοθήκης για την επιτάχυνση της.

ΚΕΦΑΛΑΙΟ 2 Βιβλιογραφική Επισκόπηση

Για την μελέτη της θεωρίας της Λογικής Σχεδίασης και για την χρήση της γλώσσας C χρειάστηκε η χρήση βιβλιογραφίας που μελετούσε σε βάθος το περιεχόμενο της Λογικής σχεδίασης και τον προγραμματισμό της γλώσσας C.

- **Ψηφιακή Σχεδίαση**

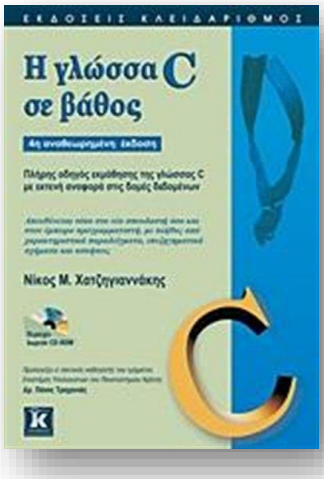
  Συγγραφείς: M. Morris Mano, Michael D. Ciletti
  Εκδοτικός Οίκος: Παπασωτηρίου
  Έκδοση: 5η έκδοση
  Έτος: 2014
  ISBN: 978-960-491-084-7



Το βιβλίο **Ψηφιακή Σχεδίαση** περιέχει όλο το θεωρητικό κομμάτι της συνδυαστικής λογικής που καλύπτει όλο το μάθημα της Λογικής Σχεδίασης το οποίο και η βιβλιοθήκη που δημιουργήθηκε προσπαθεί να δώσει λύσεις.

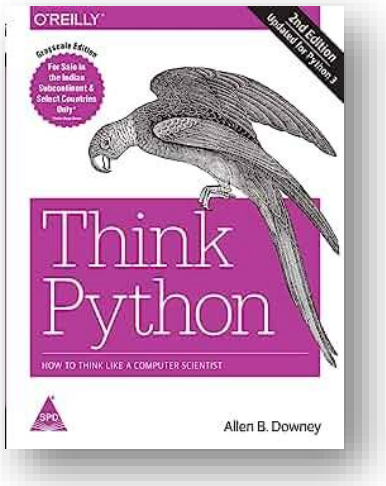
- **Η γλώσσα C σε βάθος**

  Συγγραφέας: Νίκος Μ. Χατζηγιαννάκης
  Εκδοτικός οίκος: Κλειδάριθμος
  Έκδοση: 4η αναθεωρημένη έκδοση
  ISBN: 978-960-461-498-1



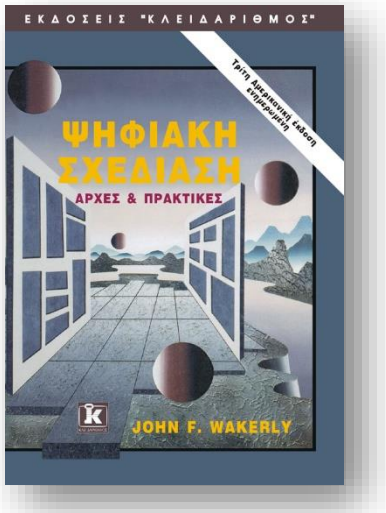
Το βιβλίο **Η γλώσσα C σε βάθος** είναι ένας οδηγός εκμάθησης της γλώσσας C και αποτελείτε από όλες τις εύκολες και δύσκολες έννοιες που χρειάζεται να γνωρίζει ο/η χρήστης της γλώσσας και είναι ένα εξαιρετικό εργαλείο για την εκμάθηση της. Το συγκεκριμένο βιβλίο περιλαμβάνει όλη την γνώση και τις τεχνικές που χρειάστηκαν για να δημιουργηθεί η βιβλιοθήκη.

- **<u>Σκέψου σε Python</u>**
  Συγγραφέας: Downy B. Allen
  Εκδοτικός οίκος: Κλειδάριθμος
  Έτος: 2020
  ISBN: 9789606450907



Το βιβλίο **Σκέψου σε Python** είναι ένας οδηγός εκμάθησης της γλώσσας Python και είναι χρήσιμο τόσο σε άτομα που μαθαίνουν για πρώτη φορά Python όσο και σε έμπειρους προγραμματιστές που δεν είναι οικείοι με την γλώσσα. Το περιεχόμενο του κυμαίνεται από εύκολες έννοιες όπως οι βρόγχοι επαναλήψεις όσο μέχρι και πιο δύσκολες έννοιες όπως ο αντικειμενοστραφής προγραμματισμός και η ανάλυση αλγορίθμων.

- **<u>Ψηφιακή Σχεδίαση: Αρχές και πρακτικές</u>**
  Συγγραφέας: John F. Wakerly
  Εκδοτικός οίκος: Κλειδάριθμος
  Έτος: 2004
  ISBN: 960-209-728-0



Το βιβλίο **Ψηφιακή Σχεδίαση: Αρχές και πρακτικές** καλύπτει όλα τα βασικά υλικά της Ψηφιακής Σχεδίασης όπως τις λογικές πύλες, τους αποκωδικοποιητές, τους πολυπλέκτες και τα κυκλώματα μνήμης flip-flop.

ΚΕΦΑΛΑΙΟ 3 Λογική Σχεδίαση

(3.1 Εισαγωγή)

Από την δημιουργία του πρώτου λειτουργήσιμου transistor το 1947 από τους Αμερικανούς φυσικούς John Bardeen και Walter Brattain κατά την διάρκεια εργασίας τους για την Αμερικάνικη ερευνητική εταιρία Bell Labs μέχρι και σήμερα [1], τα ψηφιακά συστήματα είναι ενσωματωμένα στην καθημερινή ζωή μας. Διάφορα πεδία που τα ψηφιακά συστήματα έχουν γίνει απαραίτητα περιλαμβάνουν τις επικοινωνίες, την ιατρική, την εξερεύνηση του διαστήματος, την πρόβλεψη του καιρού, την ύπαρξη του Διαδικτύου και των συσκευών που μπορούν να αλληλοεπιδράσουν με αυτό όπως οι προσωπικοί υπολογιστές και τα έξυπνα κινητά(smartphone) … και η λίστα συνεχίζεται. Είναι τόσο κυρίαρχος ο ρόλος των ψηφιακών συστημάτων στην καθημερινότητα μας, που μπορούμε να χαρακτηρίσουμε την τωρινή τεχνολογική εποχή ως **ψηφιακή**. Είτε κοιτάξουμε υπέρ-υπολογιστές σε ερευνητικά κέντρα που εκτείνονται σε μεγάλες αποστάσεις και καταναλώνουν μεγάλες ποσότητες ενέργειας, είτε δούμε κάτι συνήθης που χρησιμοποιούμε καθημερινά όπως ένα smartphone, τα συστατικά είναι ίδια. Αυτά τα κοινά συστατικά αναλύει η Λογική Σχεδίαση στην επιστήμη των υπολογιστών και για αυτά θα δημιουργούν προγραμματιστικές λύσεις μετέπειτα.

(3.2 Δυαδικοί Αριθμοί)

(3.2.1 Γενικά)

Το πρώτο από αυτά τα συστατικά και μια πολύ θεμελιώδης έννοια για οποιοδήποτε ψηφιακό σύστημα είναι το δυαδικό σύστημα αναπαράστασης αριθμών. Έννοια που έχει ρίζες στην Αρχαία Αίγυπτο, Ινδία και Κίνα, μελετήθηκε αργότερα από τον Βρετανό Μαθηματικό George Boole το 1854 όταν δημοσίευσέ ένα σημαντικό επιστημονικό άρθρο, το οποίο περιέγραφε ένα αλγεβρικό σύστημα λογικής γνωστό ως Άλγεβρα Boole. Η Άλγεβρα Boole είναι ουσιαστικής σημασίας στον σχεδιασμό ψηφιακών κυκλωμάτων. [2]

(3.2.2 Αναπαράσταση Αριθμών)

Σύμφωνα με τους M.Mano & D. Ciletti(2014) «Ένα δεκαδικός αριθμός όπως ο 7392 αναπαριστά μία ποσότητα ίση με 7 χιλιάδες συν 3 εκατοντάδες συν 9 δεκάδες συν 2 μονάδες. Οι χιλιάδες, εκατοντάδες κ.λπ.. είναι δυνάμεις του 10 που προσδιορίζονται από την θέση των συντελεστών (ψηφίων) του αριθμού»(σελ.3) Πιο ακριβέστερα το 7392 μπορεί να γραφτεί με την εξής μαθηματική έκφραση

$$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

Παρατηρούμε στην παραπάνω έκφραση ότι όλα τα ψηφία του αριθμού 4192 πολλαπλασιάζονται με κάποια δύναμη ενός κοινού αριθμού, στην συγκεκριμένη περίπτωση το 10, επειδή στο δεκαδικό σύστημα έχουμε 10 ψηφία που μπορούμε να χρησιμοποιήσουμε, από το 0 έως το 9. Αλλάζοντας το 10 μπορούμε να αναπαραστήσουμε αριθμούς και σε άλλες βάσεις όπως για παράδειγμα το 2 ή το 8. Το 2 μας ενδιαφέρει ιδιαίτερα καθώς είναι απαραίτητη βάση για τα ψηφιακά συστήματα. Γενικεύοντας την αναπαράσταση έχουμε

$$an * r^n + an - 1 * r^{n-1} + \cdots + a2 * r2 + a1 * r + \cdots \text{ [3]}$$

Όπου οι συντελεστές a μπορούν να πάρουν τιμές από 0 έως $r - 1$. Για παράδειγμα ο αριθμός 256 στο οκταδικό σύστημα θα έχει αναπαράσταση $2 * 8^2 + 5 * 8^1 + 6 * 8^0 = 174_{(10)}$.

(3.2.3 Δυαδική Αναπαράσταση)

Στην δυαδική αναπαράσταση έχουμε μόνο 2 ψηφία να χρησιμοποιήσουμε, το 0 και το 1. Για να αναπαραστήσουμε τον αριθμό 1110 στο δεκαδικό έχουμε το εξής

$$1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 1 * 8 + 1 * 4 + 1 * 2 + 0 * 1 = 14$$

Το 14 σε αυτήν την περίπτωση είναι η δεκαδική αναπαράσταση του δυαδικού αριθμού 1110.

(3.2.4 Μετατροπή Δεκαδικού Αριθμού σε Δυαδικό)

Η πρώτη συνάρτηση που υλοποιήθηκε στην βιβλιοθήκη λέγεται **10to2.c** υλοποιεί την εξής μεθοδολογία για την μετατροπή ενός αριθμού από το δεκαδικό σύστημα. Αν θέλουμε να μετατρέψουμε τον αριθμό 47 σε δυαδικό θα κάνουμε το εξής:

- Ο αριθμός 47 θα διαιρεθεί με το 2, αν υπάρχει υπόλοιπο το bit είναι 1 αλλιώς αν δεν υπάρχει υπόλοιπο και είναι τέλια διαίρεση θα είναι 0.
  $47/2 = 23 \quad \text{bit}=1$
- Η διαδικασία αυτή θα επαναληφθεί μέχρι ο αριθμός να γίνει 0.
  $23/2 = 11 \quad \text{bit}=1$
  $11/2 = 5 \quad \text{bit}=1$
  $5/2 = 2 \quad \text{bit}=1$
  $2/2 = 1 \quad \text{bit}=0$
  $1/2 = 0 \quad \text{bit}=1$
- Το τελικό αποτέλεσμα θα είναι με αντίστροφη σειρά:
  $= 101111_2$ [4]

Για μικρούς αριθμούς εύκολος τρόπος για να βρεθεί ο δυαδικός αριθμός χωρίς πράξεις είναι να γράψουμε τις δυνάμεις του δύο ανάποδα και να βάλουμε 1 στους αριθμούς που αθροιστικά μας δίνουν τον δεκαδικό αριθμό. Για παράδειγμα ο αριθμός 57:

$64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1$
$0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 = 111001_2$ [5]

(3.3 Άλγεβρα Boole)

Η άλγεβρα Boole είναι ένα σύνολο από κανόνες και θεωρήματα που εφαρμόζετε πάνω στις συναρτήσεις Boole. Οι τιμές που μπορούν να πάρουν οι μεταβλητές τις είναι το 0 και το 1 ή αλλιώς **αλήθεια** ή **ψεύδος** και οι επιτρεπτοί λογικοί Boolean τελεστές είναι το ΚΑΙ(AND)(∧), το Η(OR)(∨) και το ΟΧΙ(NOT)(¬). Περιγράφει την χρήση των συναρτήσεων Boole και χρησιμοποιείται για την ανάλυση και απλοποίηση τους.

Για παράδειγμα η συνάρτηση $(xy + x'z + yz)$ [6] μπορεί να απλοποιηθεί με τον εξής τρόπο:

- Αξίωμα 5,$(x + x' = 1)$ η συνάρτηση γίνεται $xy + x'z + yz(x + x') = xy + x'z + yzx + yzx'$
- Θεώρημα 6, απορρόφηση$(x + xy = x)$ η συνάρτηση γίνεται $xy(1 + y) + x'z(1 + y)$
- Θεώρημα 2,$(x + 1 = 1), 1 + y = 1$, η συνάρτηση γίνεται $xy + x'z$

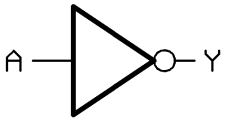
Η συνάρτηση πήγε από την αρχική της μορφή $xy + x'z + yz$ στην συνάρτηση $xy + x'z$ η οποία στην υλική της υλοποίηση θα χρειαστεί λιγότερα καλώδιά και λιγότερες λογικές πύλες, και οι δύο συναρτήσεις έχουν τον ίδιο πίνακα αληθείας, για τις ίδιες εισόδους έχουμε την ίδια έξοδο άρα είναι ταυτόσημες. Για μεγαλύτερες συναρτήσεις δεν είναι πρακτικό να απλοποιηθούν με το χέρι, σε τέτοιες περιπτώσεις χρησιμοποιείτε ο χάρτης Karnaugh και η μέθοδος Quine-McCluskey.

(3.4 Λογικές Πύλες)

Λογική πύλη είναι μια συσκευή είτε θεωρητική είτε υλική ή οποία εφαρμόζει μία συνάρτηση Boolean. Μια τέτοια συνάρτηση είναι μία λογική πράξη όπου μία ή περισσότεροι είσοδοι παράγουν μία έξοδο. Συνήθως υλοποιούνται σε διόδους και transistors αν και στο παρελθόν έχουν φτιαχτεί και με άλλους τρόπους όπως με λυχνίες κενού. Σήμερα οι περισσότερες λογικές πύλες φτιάχνονται από **MOSFETs.**

Με τις λογικές πύλες μας δίνεται η δυνατότητα να υλοποιήσουμε υλικά όλη την Άλγεβρα Boole καθώς και τα μαθηματικά και τους αλγόριθμους που την περιγράφει. Αν και οι πύλες που θα μελετηθούν παρακάτω και θα υλοποιηθούν μετέπειτα στην **C** είναι απλές, αποτελούν την βάση πιο πολύπλοκων κυκλωμάτων όπως πολυπλέκτες, αθροιστές, flip-flops τα οποία με την σειρά τους χρησιμοποιούνται για μνήμες υπολογιστή και μικροεπεξεργαστές, όπου υλοποιούνται εκατοντάδες εκατομμύρια λογικές πύλες.

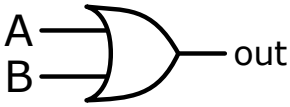
NOT GATE



Εικόνα 1 πύλη NOT,(Wikimedia Commons)

A	B
0	1
1	0

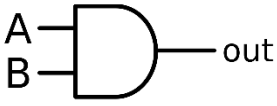
OR GATE



Εικόνα 2 Πύλη OR,(Wikipedia)

A	B	Z
0	0	0
1	0	1
0	1	1
1	1	1

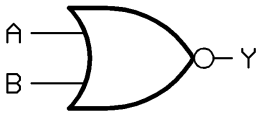
AND GATE



Εικόνα 3 πύλη AND,(Wikimedia Commons)

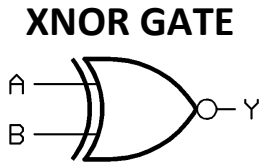
A	B	Z
0	0	0
1	0	0
0	1	0
1	1	1

NOR GATE



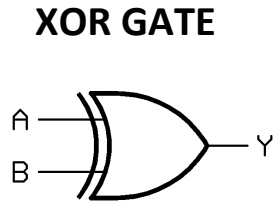
Εικόνα 4 Πύλη NOR,(Wikimedia Commons)

A	B	Z
0	0	1
1	0	0
0	1	0
1	1	0



Εικόνα 5 Πύλη XNOR,(Wikimedia commons)

A	B	Z
0	0	1
1	0	0
0	1	0
1	1	1



Εικόνα 6 Πύλη XOR,(Wikimedia Commons)

A	B	Z
0	0	0
1	0	1
0	1	1
1	1	0

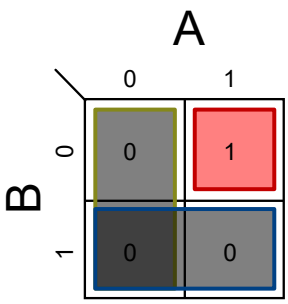
Στις παραπάνω εικόνες απεικονίζονται οι βασικές λογικές πύλες NOT, OR, AND, NOR, XNOR, XOR που χρησιμοποιούνται για την δημιουργία ψηφιακών κυκλωμάτων, η δημιουργία των πυλών NOR, XNOR, XOR μπορεί να επιτευχθεί με τον συνδυασμό της πύλης NOT και είτε της πύλης AND είτε τής πύλης OR. [7]

(3.5 Ελαχιστοποίηση Συναρτήσεων)

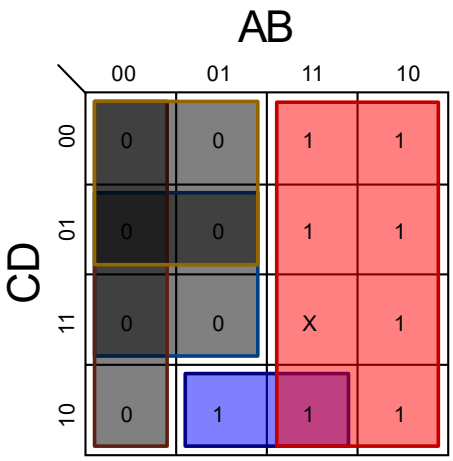
(3.5.1 Χάρτες Karnaugh)

Σε ένα ψηφιακό κύκλωμα φτιαγμένο με λογικές πύλες, η πολυπλοκότητα είναι ανάλογη με την πολυπλοκότητα της αλγεβρικής συνάρτησης που την περιγράφει. Το τελικό αποτέλεσμα η έξοδος στο πίνακα αληθείας, μπορεί να μας δοθεί από διάφορες αλγεβρικές εκφράσεις. Για να κατανοηθεί καλύτερα ένα κύκλωμα και να χρησιμοποιηθούν λιγότερες λογικές πύλες είναι προτιμότερο να ελαχιστοποιήσουμε την λογική έκφραση που έχουμε.

Για την ελαχιστοποίηση της λογικής έκφρασης μπορεί να χρησιμοποιηθεί ο Χάρτης Karnaugh ή αλλιώς Κ-χάρτης, αν και είναι δυνατό να ελαχιστοποιηθεί η λογική έκφραση με αλγεβρικό τρόπο, ο χάρτης προτιμάται λόγο της απλότητας και ευκολίας του. Ο Κ-χάρτης είναι ένα γράφημα που αποτελείται από τετράγωνα, όπου το κάθε τετράγωνο αναπαριστά έναν ελαχιστόρο της συνάρτησης(εικόνα 1.). Δεδομένου ότι κάθε λογική έκφραση μπορεί να αναπαρασταθεί ως άθροισμα ελαχιστόρων, κάθε συνάρτηση μπορεί να αναπαρασταθεί γραφικά στον Κ-χάρτη. Ο χάρτης ουσιαστικά μας παρέχει ένα οπτικό μέσο όλων των πιθανών τρόπων έκφρασης μιας συνάρτησης [8]. Επιλέγοντας διαφορετικές διατάξεις τετραγώνων στον χάρτη, ο χρήστης μπορεί να δημιουργήσει εναλλακτικές αλγεβρικές εκφράσεις για την συνάρτηση. Μέσα από αυτές τις εκφράσεις ο χρήστης μπορεί να επιλέξει την πιο απλουστευμένη. [73]



Εικ.1 Χάρτης Karnaugh 2 μεταβλητών,(Wikimedia Commons)



Εικ.2 Χάρτης Karnaugh 4 μεταβλητών,(Wikimedia Commons)

Στην εικ.1 και εικ.2 απεικονίζονται 2 χάρτες Karnaugh με 2 και 4 μεταβλητές,

Η απλουστευμένη αλγεβρική μορφή που θα παράγει ο χάρτης θα είναι αυτή που θα έχει τον ελάχιστο αριθμό πυλών και τον ελάχιστο αριθμό εισόδων σε κάθε πύλη. Στην περίπτωση που υπάρχουν παραπάνω από μία ελαχιστοποιημένες εκφράσεις που συμπληρώνουν τα κριτήρια ελαχιστοποίησης τότε όλες αυτές οι λύσεις είναι αποδεκτές.

(3.5.2 Μέθοδος Quine-McCluskey)

Η μέθοδος Quine-McCluskey ή αλλιώς μέθοδος των prime implicants δημιουργήθηκε το 1952 από τον Willard V. Quine και είναι μέθοδος ελαχιστοποίησης συναρτήσεων Boolean [9]. Η λειτουργία της είναι παρόμοια με αυτή του χάρτη Karnaugh καθώς και οι δύο χρησιμοποιούν ελαχιστόρους για να βρουν την ελαχιστοποιημένη μορφή της συνάρτησης. Η μέθοδος Quine-McCluskey βρίσκει χρήση στην ελαχιστοποίηση συναρτήσεων με περισσότερες από 4 μεταβλητές καθώς η χρήση του χάρτη Karnaugh γίνεται δύσκολη σε τέτοιες περιπτώσεις.

Σε παρόμοιο τρόπο με τον χάρτη K η QM βρίσκει όλες τις δυνατές διατάξεις στον χάρτη που ονομάζονται **prime implicants** και τους βάζει σε έναν ξεχωριστό πίνακα για να βρει τους **ουσιώδης(essential) prime implicants,** αυτούς δηλαδή που χρειαζόμαστε για την ελαχιστοποιημένη συνάρτηση. Με είσοδο τους ελαχιστόρους `2, 4, 6, 8, 9, 10, 12, 13, 15` και τις μεταβλητές `A,B,C,D` η QM εκτελείτε με τον εξής τρόπο:

- Ομαδοποιούμε τους ελαχιστόρους σε σετ των οποίων υπάρχει διαφορά μόνο ενός bit, ελέγχοντας κάθε ελαχιστόρο με τους όλους υπόλοιπους μέχρι να μην υπάρχουν άλλες ομαδοποιήσεις. Κάθε σετ που δεν μπορεί να ομαδοποιηθεί με κάποιο άλλο είναι prime implicant και το σημειώνουμε.

```
INITIAL QUINE-MCLUSKEY TABLE
       A B C D
     2|0 0 1 0  |1 one's
     4|0 1 0 0  |1 one's
     6|0 1 1 0  |2 one's
     8|1 0 0 0  |1 one's
     9|1 0 0 1  |2 one's
    10|1 0 1 0  |2 one's
    12|1 1 0 0  |2 one's
    13|1 1 0 1  |3 one's
    15|1 1 1 1  |4 one's
-----------------------------------
      1st GROUP TABLE
      A B C D
  2, 6|0 - 1 0  |1 one's
  2,10|- 0 1 0  |1 one's
  4, 6|0 1 - 0  |1 one's
  4,12|- 1 0 0  |1 one's
  8, 9|1 0 0 -  |1 one's
  8,10|1 0 - 0  |1 one's
  8,12|1 - 0 0  |1 one's
  9,13|1 - 0 1  |2 one's
 12,13|1 1 0 -  |2 one's
 13,15|1 1 - 1  |3 one's
-----------------------------------
      2st GROUP TABLE
          A B C D
 8, 9,12,13|1 - 0 -  |1 one's
 8,12, 9,13|1 - 0 -  |1 one's
```

- Παίρνουμε όλους τους prime implicants που βρήκαμε και τους ταξινομούμε στον παρακάτω πίνακα όπου στις στήλες είναι όλοι οι ελαχιστόροι που χρησιμοποιούν οι prime implicants και στις γραμμές είναι οι συναρτήσεις που τους αντιστοιχούν. Αφού έχει φτιαχτεί ο prime implicant πίνακας αφαιρούμε τους essential prime implicants και για να γίνει αυτό προσέχουμε ποιες στήλες έχουν μόνο ένα 1 και τις αφαιρούμε μαζί με τους ελαχιστόρους που την αντιστοιχούν καθώς αυτές οι στήλες ανήκουν σε **essential prime implicants**, αυτό ονομάζεται **κυριαρχία στήλης(column dominance)**.
  Αν δεν γίνεται να αφαιρεθούν prime implicants γιατί δεν υπάρχει στήλη που να έχει μόνο ένα 1 τότε πρέπει να δούμε αν υπάρχει γραμμή που να έχει ένα 1 και αφαιρείτε με τον ίδιο τρόπο ο essential prime implicant, αυτό ονομάζεται **κυριαρχία γραμμής(row dominance)**. Χρησιμοποιώντας είτε column είτε row dominance η διαδικασία επαναλαμβάνεται μέχρι να εξαντληθούν όλοι οι ελαχιστόροι από τον πίνακα.

```
        PRIME IMPLICANT TABLE
          2   4   6   8   9  10  12  13  15
=========================================
A'CD'|  1   -   1   -   -   -   -   -   -
B'CD'|  1   -   -   -   -   1   -   -   -
A'BD'|  -   1   1   -   -   -   -   -   -
BC'D'|  -   1   -   -   -   -   1   -   -
AB'D'|  -   -   -   1   -   1   -   -   -
  ABD|  -   -   -   -   -   -   -   1   1
  AC'|  -   -   -   1   1   -   1   1   -

      PRIME IMPLICANT TABLE |column
dominance|
          2   4   6
=========================================
A'CD'|  1   -   1
B'CD'|  1   -   -
A'BD'|  -   1   1
BC'D'|  -   1   -

      PRIME IMPLICANT TABLE |row dominance|
          4   6
=========================================
A'CD'|  -   1 | 0power
A'BD'|  1   1 | 2power
BC'D'|  1   - | 0power
```

- Η τελική ελαχιστοποιημένη συνάρτηση θα είναι το άθροισμα όλων των essential prime implicants και θα είναι η $AC' + ABD + B'CD' + A'CD' + BC'D' + AB'D'$.

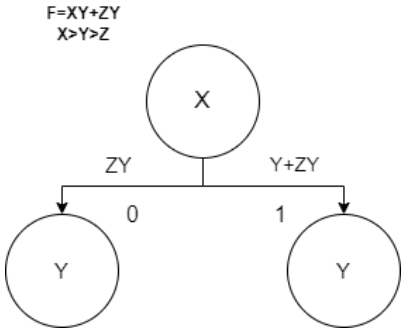
(3.6 Binary Decision Diagrams – BDD's)

Δυαδικό διάγραμμα απόφασης ή αλλιώς BDD είναι ένας κατευθυνόμενος ακυκλικός γράφος που αναπαριστά μια συνάρτηση Boole. Τα BDDs είναι ένας τρόπος να απεικονίσουμε μια συνάρτηση Boole και μπορεί να γίνει συμπαγής με την μετατροπή σε ελαχιστοποιημένο διατεταγμένο BDD ή αλλιώς **ROBDD** με τους αντίστοιχους κανόνες. Επειδή η χρήση των κανόνων θεωρείτε απαραίτητη όταν τα χρησιμοποιούμε τα ονομάζουμε BDD αλλά εννοούμε τα **ROBDD [10]**.

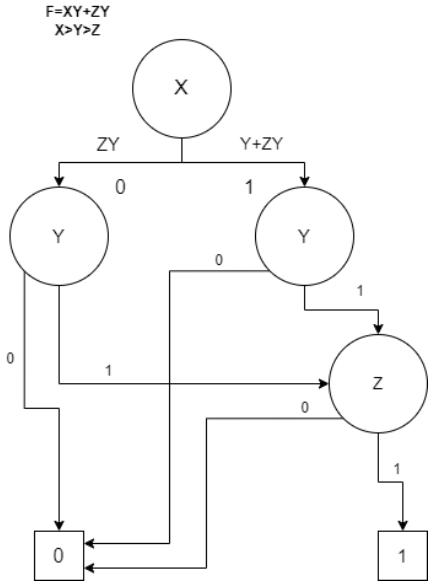
Ένα BDD έχει δύο τερματικούς κόμβους όπου ο ένας αντιστοιχεί στο 0 και ο άλλος στο 1, διαθέτει επίσης και ένα σύνολο από μη τερματικούς κόμβους όπου ο κάθε μη τερματικός κόμβος έχει μία μεταβλητή και δύο ακμές, με κάθε ακμή να έχει μια συνάρτηση όπου η μεταβλητή είναι 0 στην μία και 1 στην άλλη. Η σειρά των μεταβλητών θα ορίζεται από εμάς και θα είναι ίδια για αυτό το BDD ονομάζεται διατεταγμένο. Το BDD θεωρείτε ελαχιστοποιημένο όταν έχουν εφαρμοστεί οι εξής κανόνες:

- Συγχώνευση των ισομορφικών υπό-γράφων.
- Διαγραφή κόμβων όπου δύο κόμβοι-παιδιά τους είναι ισομορφικά.

[1]



Εικόνα 3 Αρχικό στάδιο δημιουργίας BDD



Εικόνα 4 Τελικό στάδιο δημιουργίας BDD

[1] Τα bdd των εικόνων 3 και 4 φτιάχτηκαν με το online εργαλείο draw.io

(3.7 Άλλες αναπαραστάσεις αριθμών)

Ανάλογα με το ψηφιακό κύκλωμα που σχεδιάζουμε μπορεί να εμφανιστεί η ανάγκη να αναπαραστήσουμε δεκαδικούς αριθμούς με διαφορετικό τρόπο από αυτόν του κανονικού δυαδικού που χρησιμοποιούμε συνήθως, να χρησιμοποιήσουμε δηλαδή έναν κώδικα. Οι πιο διάσημοι από τους δυαδικούς κώδικες είναι ο **BCD(8421)**, **2421**, **BIQUINARY** και **GRAY [11]**.

Δεκαδικό	BCD	2421	Biquinary	Gray
0	0000	0000	0100001	0000
1	0001	0111	0100010	0001
2	0010	0110	0100100	0011
3	0011	0101	0101000	0010
4	0010	0100	0110000	0110
5	0101	1011	1000001	0111
6	0110	1010	1000010	0101
7	0111	1001	1000100	0100
8	1000	1000	1001000	1100
9	1001	1111	1010000	1101

Ο κώδικας **BCD** χρησιμοποιεί μια τετράδα από bits για να αντιπροσωπεύσει ένα ψηφίο ενώ το δυαδικό σύστημα χρησιμοποιεί δυνάμεις του δύο, αυτό σημαίνει ότι ο κώδικας BCD είναι πιο συμπαγής και ευανάγνωστος από τον χρήστη. Χρησιμοποιεί λιγότερο χώρο από τους δυαδικούς αριθμούς αλλά χρειάζεται μια μικρή πρόσθεση στο κύκλωμα για να μπορεί να γίνει βασική αριθμητική με αυτόν [12]. Αν και δεν έχει διαδεδομένη χρήση σήμερα είναι αναγκαίος σε συστήματα όπου η δεκαδική ακρίβεια είναι απαραίτητη. Για παράδειγμα:

Decimal: 234
Binary: 11101010
BCD: 001000110100

Ο κώδικας **2421** ή αλλιώς Aiken κώδικας είναι συμπληρωματικός του BCD και χρησιμοποιείται σε ηλεκτρονικά ρολόγια και αριθμομηχανές [13]. Για παράδειγμα:

Decimal: 1527
Binary: 10111110111
2421: 0001101100101101

Ο κώδικας **Biquinary** χωρίζεται σε δύο τμήματα, ένα είναι το “bi” και αποτελείτε από 2 bits και το άλλο είναι το “quinary” που αποτελείτε από 5 bits. Η λειτουργία του μοιάζει με αυτή του άβακα με το “quinary” τμήμα να έχει εύρος τιμών 0-4 ή 5-9 και το “bi” τμήμα να αλλάζει το εύρος από το ένα στο άλλο. Ο συγκεκριμένος κώδικας χρησιμοποιήθηκε σε άβακες και σε μερικούς από τους πρώτους υπολογιστές, η χρήση του σε υπολογιστές ήταν ιδιαίτερα χρήσιμη καθώς οι καταστάσεις bit των υπολογιστών της εποχής κολλούσαν συχνά λόγω χρήσης μηχανικών διακόπτων και το τμήμα “bi” μπορούσε να λειτουργήσει ως bit ασφάλειας για ανίχνευση λαθών επιβεβαιώνοντας αν ένας αριθμός είναι έγκυρος [14]. Για παράδειγμα:

Decimal: 476
Binary: 111011100
Biquinary: 011000010001001000010

Ο κώδικας **Gray** ταξινομεί τα bits με τέτοιο τρόπο ώστε διαδοχικές τιμές διαφέρουν μόνο σε ένα bit. Επειδή αλλάζει μόνο ένα bit κάθε φορά χρησιμοποιείται για την ανίχνευση λαθών σε ψηφιακά συστήματα και επικοινωνίες αλλά και στους χάρτες Karnaugh [15]. Για παράδειγμα:

Decimal: 1234
Binary: 10011010010
Gray: 0001001100100110

ΚΕΦΑΛΑΙΟ 4 Σύγκριση C - Python

(4.1 Ιστορική Αναδρομή των Γλωσσών Προγραμματισμού C και Python)

Η γλώσσα προγραμματισμού C(22) αναπτύχθηκε την δεκαετία του '70 από τον Dennis Ritchie, με πρώτη εμφάνιση της το 1972. Αρχικά φτιαγμένη στα Bell Labs από τον Dennis Ritchie ως διάδοχος της γλώσσας B, η C ήταν μια προσπάθεια βελτίωσης της B με σκοπό χρήσης της στην δημιουργία του λειτουργικού συστήματος UNIX(31) [16].

Εικ. 1 Το επίσημο λογότυπο της C, (Wikimedia Commons)

Χαρακτηριστικά της C είναι ο μινιμαλισμός της, η φορητότητα της, η ταχύτητα της καθώς και η δυνατότητα της να έχει πρόσβαση χαμηλού επιπέδου στην μνήμη την κάνουν ιδιαίτερα χρήσιμη στην δημιουργία λειτουργικών συστημάτων όπως το UNIX(και κατ επέκταση το Linux), μεταγλωττιστών όπως ο GCC, ενσωματωμένων συστημάτων και γλωσσών προγραμματισμού όπως η Python.

Η Python(23) δημιουργήθηκε το 1989 στο ερευνητικό κέντρο Centrum Wiskunde & Informatica από τον Guido van Rossum [17]. Η Python είναι μια υψηλού επιπέδου γλώσσα προγραμματισμού το ίδιο με την C, με την διαφορά όμως ότι η C είναι μεταγλωττισμένη και διαδικαστική γλώσσα ενώ η Python είναι διερμηνευόμενη. Η Python επίσης μαζί με διαδικαστική λογική επιτρέπει και αντικειμενοστραφές προγραμματισμό, τρόπος που έγινε πιο διάσημος στα χρόνια μετά την κορυφή της διασημότητας της C, γλώσσες όπως η **C++**(24) το 1979, η **Java**(25) το 1996 και η **Javascript** το 1995.

Εικ.2 Το επίσημο λογότυπο της Python, (Wikimedia Commons)

Η φιλοσοφία της ήταν και είναι η απλοποίηση και η μείωση του κώδικα που χρειάζεται να γράψει ο προγραμματιστής καθώς και την βελτίωση της ορατότητας του. Το καταφέρνει αυτό απλοποιώντας το συντακτικό σε σχέση με αντίστοιχες γλώσσες όπως η **C,** και κρύβοντας την διαχείριση μνήμης πίσω από ένα συλλέκτη σκουπιδιών(garbage collector) ο οποίος διαχειρίζεται μόνος την μνήμη διώχνει αντικείμενα από την μνήμη που δεν χρειάζονται πλέον.

Η Python σήμερα χρησιμοποιείτε κυρίως στην αυτοματοποίηση, στην ανάλυση δεδομένων και στο μηχανική μάθηση(machine learning). Ως γενική γλώσσα προγραμματισμού όμως αυτές η χρήσεις τις δεν είναι οι μοναδικές και έχει πολλές άλλες χρήσεις, μερικές από αυτές είναι η δημιουργία ηλεκτρονικών παιχνιδιών(video games) και η δημιουργία σεναρίων(scripting).

(4.2 Χρήσεις και Πλεονεκτήματα)

Ως γλώσσες γενικού σκοπού και η **C** και η **Python** καλύπτουν μεγάλη γκάμα δυνατοτήτων και είναι και οι δύο ικανές να χρησιμοποιηθούν σε πολύπλοκα συστήματα και προγράμματα. Η κάθε γλώσσα όμως ταιριάζει σε διαφορετικές χρήσεις καλύτερα και έχει διαφορετικά πλεονεκτήματα σε σχέση με την άλλην.

Αρχικά όσο αφορά το εκπαιδευτικό κομμάτι η Python ταιριάζει καλύτερα σε κάποιον/ποια που έρχεται πρώτη φορά σε επαφή με τον προγραμματισμό. Το εύκολο συντακτικό και η πρόσθεση βασικών δομών όπως οι λίστες και μια μεγάλη ποικιλία ενσωματωμένων συναρτήσεων μέσα στην ίδια την γλώσσα κάνουν την εμπειρία πιο γρήγορη και πιο εύκολη.

Κάτι άλλο που μπορεί να βοηθήσει είναι η ευκολία της εγκατάστασης και χρήσης της Python, καθώς η γλώσσα εγκαθίσταται μόνο με ένα **.exe** αρχείο και μπορεί και περιλαμβάνει το directory path στις μεταβλητές περιβάλλοντος από την εγκατάσταση της. Επειδή είναι και **γλώσσα scripting** έρχεται με το δικό της τερματικό που επιτρέπει την εκτέλεση εντολών 'ζωντανά' μία-μία, ικανότητα που είναι δυνατή χάρη στον διερμηνευτή της Python. Στην C το πιο δύσκολο συντακτικό και πιο δύσκολες έννοιες όπως η διαχείριση μνήμης και οι δείκτες καθώς και η διαδικασία μεταγλώττισης για την εκτέλεση του προγράμματος, την κάνουν πιο δύσκολη γλώσσα για την χρήση της στην εκπαίδευση.

Το γεγονός όμως ότι η Python είναι διερμηνευμένη και έχει **garbage collector** την κάνει πιο αργή από την C. Ανάλογα το πρόγραμμα η Python είναι είτε πιο αργή είτε πολύ πιο αργή από την C, κάτι που δεν την κάνει την καλύτερη επιλογή για χρήση σε προγράμματα υψηλών επιδόσεων με ανάγκη για μεγάλους υπολογισμούς, ούτε στην δημιουργία λειτουργικών συστημάτων ή μεταγλωττιστών όπου η ταχύτητα είναι απαραίτητο προαπαιτούμενο. Η Python θυσιάζει ταχύτητα για διευκόλυνση προγραμματισμού με πιο ορατό και μικρό κώδικα, και την δυνατότητα να μπορεί να χρησιμοποιηθεί σε μεγάλο εύρος προβλημάτων.

(4.3 Τα βασικά)

(4.3.1 Μεταβλητές)

Στην C οι μεταβλητές είναι στατικά ορισμένες και πρέπει να οριστούν με ακρίβεια από τον χρήση με το χρήση ειδικών λέξεων-κλειδιών όπως το `int` και το `double`. Στην Python δεν γίνεται αυτό και οι μεταβλητές είναι δυναμικά ορισμένες και δεν χρειάζεται η χρήση λέξεων-κλειδιών.

(4.3.2 Δηλώσεις if)

Οι συνθήκες αν(if) είναι παρόμοιες ανάμεσα στις δύο γλώσσες καθώς και οι δύο χρησιμοποιούνε τις ίδιες συνθήκες:

- **a == b**: αληθείς αν το **a** ίσο με το **b**
- **a != b**: αληθείς αν το **a** διάφορο με το **b**
- **a > b**: αληθείς αν το **a** μεγαλύτερο από το **b**
- **a >= b**: αληθείς αν το **a** μεγαλύτερο **ή** ίσο με το **b** (**a => b** στην Python)
- **a < b**: αληθείς αν το **a** μικρότερο από το **b**
- **a <= b**: αληθείς αν το **a** μικρότερο **ή** ίσο με το **b** (**a <= b** στην Python)

```
#Python
if x == 1:
    x = x + 5
    a[x] = 1
//C
if(x == 1)
{
    x = x + 5;
    a[x] = 1;
}
```

Εικόνα 3 Απλή διαφορά σύνταξης

Διαφορά υπάρχει στην σύνταξη των δηλώσεων με την Python να μην χρησιμοποιεί παρενθέσεις και να συμβολίζει το τέλος με διπλή τελεία. Πιο σημαντική διαφορά όμως είναι το γεγονός ότι η Python χρησιμοποιεί εσοχές για να διακρίνει τη είναι μέσα στην περιοχή της με διάστημα 4 κενούς χαρακτήρες ή αλλιώς ένα **tab**, ενώ η C χρησιμοποιεί αγκύλες για να γνωρίζει το περιεχόμενο των δηλώσεων της. Παράδειγμα φαίνεται στην εικόνα 3 αλλά και στην εικόνα 4 όπου οι βρόγχοι δηλώνονται παρόμοια.

(4.3.3 For Loops)

```
# adding numbers 1-N in python
N = 100000000
sum = 0
for i in range(N):
    sum = sum + i;
print(sum, "in", (end - start), "s");

// adding numbers 1-N in C
long long int N = 100000000;
long long int sum = 0;
for(int i = 0; i < N; i++)
{
    sum = sum + i;
}
printf("%lld in %fs", sum, (end - start));
```

Εικόνα 4 Άθροισμα αριθμών 1-N

Οι βρόγχοι επανάληψης(**for** loops) και στις δύο γλώσσες είναι παρόμοια με την διαφορά ότι η Python χρησιμοποιεί εσοχές(indetation) για να διαχωρίσει τα περιεχόμενα του **for** από την συνέχεια του κώδικα, ότι είναι μέσα στο **for** δηλαδή πρέπει να είναι 1 tab μέσα στην γραμμή ή αλλιώς 4 κενά. Η μεταβλητή i και στις δύο εκδοχές είναι τοπική μεταβλητή μέσα στο **for**, κάτι που είναι δυνατό στην C από την έκδοση C99. Αν χρησιμοποιηθεί η έκδοση C89 το αντίστοιχο for θα έπρεπε να γραφτεί έτσι `int i = 0; for(i, ...).`

Εκτελώντας τους παραπάνω κώδικες φαίνεται ότι τα **for** loops είναι πολύ πιο αργά στην Python, με $\mathbf{0.002\,sec}$ $\boldsymbol{χρόνο\ εκτέλεσης(runtime)}$ για την C και $\mathbf{11.84}\boldsymbol{sec}$ για την Python. Η C καταλήγει να είναι $\mathbf{5920}$ φορές πιο γρήγορη σε αυτήν την περίπτωση. Η Python είναι αργή γιατί στην πραγματικότητα δεν έχει μεταβλητές αλλά ονόματα, κάθε φορά που δηλώνουμε μια μεταβλητή στην Python δεν γνωρίζει τι τύπος είναι. Πρέπει να δημιουργήσει ένα καινούργιο ***PyObject** και να θέσει τον τύπο μεταβλητής του όπως καταλάβει ο διερμηνευτής από την τιμή που του δώσαμε, να δημιουργήσει ένα όνομα που να με το όνομα που να δώσαμε στην μεταβλητή και να το δείξει στο ***PyObject** που δημιούργησε και τέλος να ανεβάσει τον μετρητή αναφοράς του κατά ένα. Αυτό επιτρέπει την Python να έχει δυναμικές δηλώσεις μεταβλητών. Στην C δεν γίνεται αυτό, ο προγραμματιστής πρέπει ρητά να δηλώσει τις μεταβλητές.

Αυτό δεν σημαίνει όμως ότι πρέπει να είναι αργή η Python, απλά χρειάζονται μερικά παραπάνω βήματα για να επιταχυνθεί το παραπάνω πρόγραμμα. Με την βιβλιοθήκη **NumPy(26)** η οποία η ίδια είναι γραμμένη σε C ο παραπάνω κώδικας μπορεί να ξαναγραφτεί έτσι:

```
import numpy as np
# adding numbers 1-N
N = 100000000
sum = 0
start = time.time()
sum =
np.sum(np.arange(N,dtype=np.int64))
end = time.time();
print(sum, "in", (end - start), "s");
```

Εικόνα 5 Άθροισμα αριθμών 1-N με NumPy

Με αυτήν την έκδοση(εικόνα 5) το αποτέλεσμα είναι ίδιο αλλά ο χρόνος εκτέλεσης πήγε στα $\mathbf{0.3}\boldsymbol{sec}$, χρόνος **40** φορές πιο γρήγορος από την αρχική έκδοση και **150** φορές πιο αργός από την εκδοχή της C, μεγάλη βελτίωση σε σχέση με την απλή Python.

(4.3.4 Εκτυπώσεις)

Αφού υπολογιστεί το πρόβλημα που θέλουμε να λύσουμε πρέπει κάπως να πάρουμε το αποτέλεσμα για να το δούμε. Ο πιο απλός τρόπος που μπορεί να επιτευχθεί αυτό είναι με εκτυπώνοντας το αποτέλεσμα στο τερματικό. Κάθε γλώσσα έχει την δικιά της συνάρτηση `print("stuff to print")` για να εκτυπώσει με τους δικούς της κανόνες.

Στην C υπάρχει η συνάρτηση `printf( const char * format, ... );` όπου `format` ένα ειδικό αλφαριθμητικό με την μορφή "`%[flags][width][.precision][length]specifier[ref]`"(27), και με αυτές τις παραμέτρους επικοινωνούμε στην `printf` πως και τι να εκτυπώσει. Οι παράμετροι αντιστοιχούν ένα προς ένα με την ίδια σειρά τις δεύτερες τιμές εισόδου και έχουν τις εξής επιδράσεις:

- `specifier` είναι ο τύπος μεταβλητής που θέλουμε να εκτυπώσουμε π.χ. %d για ακέραιους και %f για δεκαδικούς.
- `flags` είναι προαιρετικές επιλογές όπως π.χ. να ορίσουμε δικό μας πρόσημο + ή – στην τιμή ότι πρόσημο και να έχει ή ίδια.
- `width` είναι ή προαιρετική επιλογή να ορίσουμε ελάχιστο αριθμό χαρακτήρων για εκτύπωση, αν εκτυπωθούν λιγότεροι αριθμοί από αυτόν τότε το αποτέλεσμα γεμίζεται με κενά. Είναι χρήσιμο για να ευθυγραμμιστούν τιμές που δεν έχουν τον ίδιο αριθμό ψηφίων.
- `.precision` είναι προαιρετικός αριθμός που ορίζει τον μέγιστο αριθμό χαρακτήρων ή ψηφίων για εκτύπωση.
    - Αν είναι ακέραιος τότε ορίζει ελάχιστο αριθμό ψηφίων και αν είναι μικρότερος συμπληρώνει το αποτέλεσμα με μηδενικά.
    - Αν είναι δεκαδικός τότε ορίζει μέγιστο αριθμό δεκαδικών ψηφίων
    - Αν είναι αλφαριθμητικό τότε ορίζει μέγιστο αριθμό γραμμάτων
- `length` είναι προαιρετική επιλογή που προσδιορίζει το μήκος των μεταβλητών π.χ. **ll** για μεγάλους ακεραίους ή **h** για μικρούς.

```
printf("z = %.4d\ny = %.2lf\n%.7s\n", z, y, str);
    /*
        z = 0001
        y = 123.12
        hello w
       */
```

Εικόνα 6 Παράδειγμα χρήσης printf() στην C

Η Python διαθέτει δική της υλοποίηση με μορφή την `print(*objects, sep=, end=).`

- Η **`print`** της Python παίρνει `*objects` τα οποία μπορεί να είναι οποιουδήποτε τύπου χωρίς δήλωση και χωρίζονται με κόμματα(αυτόματα βάζει κενό χαρακτήρα), αν χωρίζονται με τον τελεστή **+** τότε η **`print`** προσθέτει όλα τα `*objects` σε ένα αλφαριθμητικό και μετά τα εκτυπώνει.
- Ο χαρακτήρας που βάζει η **`print`** είναι ο κενός χαρακτήρας αλλά αν θέλουμε να χρησιμοποιηθεί άλλος χαρακτήρας μπορεί να γίνει η χρήση του `sep=` όπου του δίνουμε τιμή τον χαρακτήρα ή το αλφαριθμητικό που θέλουμε να βάλει ανάμεσα στα `*objects`.
- Αν χρειαστεί η εκτύπωση μας να τελειώνει πάντα με έναν συγκεκριμένο χαρακτήρα μπορεί να γίνει χρήση του `end=` όπου του δίνουμε τιμή τον χαρακτήρα ή αλφαριθμητικό αυτό που θέλουμε να μπει στο τέλος.

```
str = "an"
x = 12.353535
print("This is",str,"example number","{:.2f}".format(x),sep='--',end="!!!!")
   #This is--an--example number--12.35!!!!
```

Εικόνα 7 Παράδειγμα χρήσης print() στην Python

- Αν θέλουμε να σχεδιάσουμε τις μεταβλητές μας όπως στην υλοποίηση της C μπορεί χα χρησιμοποιηθεί με τον ίδιο τρόπο με την εξής μορφή όπου φαίνεται και στο παράδειγμα της εικόνα 7:

```
{:.%[flags][width][.precision][length]specifier}.format(var)
```

Οι εκτυπώσεις διαφέρουν ανάμεσα στις δύο γλώσσες με την C να χρειάζεται πιο ακριβής ορισμούς και αυστηρό τρόπο αντιστοιχίας ένα προς ένα ορισμού και μεταβλητής. Η Python αντίθετα είναι λιγότερο αυστηρή χωρίς ανάγκη ορισμού του τύπου μεταβλητής με εύκολη προσθήκη μεταβλητών διαφορετικών τύπων, εύκολο ξεχώρισμά αντικειμένων και την δυνατότητα να χρησιμοποιεί τον τρόπο σχεδιασμού της C.

(4.4 Μετατροπές Μεταβλητών)

Στην Python η μετατροπή από έναν τύπο δεδομένου σε έναν άλλον είναι απλή και γίνεται χρησιμοποιώντας τις ενσωματωμένες συναρτήσεις που μας παρέχει η ίδια όπως οι παρακάτω: `int(),float(),str()`. Οι εξής συναρτήσεις χρησιμοποιούνται κανονικά όπως όλες οι συναρτήσεις και παίρνουν ως είσοδο την μεταβλητή που θέλουμε να γίνει μετατροπή, π.χ αν έχουμε `x = 5` ακέραια μεταβλητή και θέλουμε να γίνει τύπου string τότε μπορούμε να καλέσουμε το `str(x)` που θα μας γυρίσει το `x` ως `'5'` .

The ASCII code

American Standard Code for Information Interchange

ASCII control characters			
DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters								
DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Εικόνα 8 Πίνακας ASCII (Wikimedia Commons)

Στην C μπορούν γίνουν με τον ίδιο τρόπο οι μετατροπές όσο αφορά τα `int` και τα `float` αλλά όχι τα `str` και τα `chr`.Για να μετατρέψουμε ακέραιους και δεκαδικούς σε χαρακτήρες ή αλφαριθμητικά και ανάποδα πρέπει να υλοποιήσουμε δικούς μας τρόπους. Για την μετατροπή ακεραίων-χαρακτήρων μπορούμε να προσθέσουμε στον ακέραιο τον χαρακτήρα `'0'` όπως στην εικόνα 9. Το παραπάνω δουλεύει γιατί στην C οι `char` είναι στην πραγματικότητα αριθμοί, πιο συγκεκριμένα είναι ακέραιοι των **8 bit** ή αλλιώς **1 byte** με εύρος τιμών από το -128 μέχρι το 127. Από το 0 μέχρι το 128 κάθε αριθμός μας δίνει έναν χαρακτήρα και αυτό είναι ο γνωστός **ascii πίνακας.** Στον πίνακα ascii (εικόνα 8) φένετε ότι ο χαρακτήρας `'0'` έχει την τιμή 48 και από κάτω του ακολουθούν όλοι οι υπόλοιποι αριθμοί, αν ο ακέραιος μας είναι το 4 και του προσθέσουμε το `'0'` το αποτέλεσμα θα είναι ο τιμή 52 με χαρακτήρα το `'4'` .

Η παραπάνω μέθοδος λειτουργεί για ακεραίους από το 0 μέχρι και το 9 καθώς σε έναν `char` μπορεί να χωρέσει μόνο ένας χαρακτήρας, τι αν θέλουμε να μετατρέψουμε μεγαλύτερους αριθμούς? Αρχικά θα χρειαστούμε ένα αλφαριθμητικό(**string)** το οποίο θα είναι ένας πίνακας από `char` αρκετά μεγάλος ώστε να χωράει όλα τα ψηφία μας όπως: `char str[128] = {0};` για αρχικοποίηση του πίνακα). Μετά μπορούμε να βγάλουμε όλα τα ψηφία και να τα μετατρέψουμε ψηφίο-ψηφίο με τον τρόπο της προηγούμενης παραγράφου όπως στην εικόνα 9.

```
char str[128] = {0};
int i = 0;
while(number > 0)
{
    str[i++] = (number % 10)+'0';
    number = number / 10;
}
// ή αλλιώς
sprintf(str, "%d", number);
```

Εικόνα 9 Μετατροπή ακεραίου σε χαρακτήρα

Σε αυτήν την περίπτωση όμως μπορεί να μας βοηθήσει η C και να χρησιμοποιήσουμε την συνάρτηση **sprintf**, παίρνει για είσοδο στο string που φτιάξαμε, ένα string που δηλώνουμε από τι τύπο θα γίνει η μετατροπή, και τον αριθμό που θέλουμε να μετατρέψουμε. Με αυτόν τον τρόπο μπορούμε να μετατρέψουμε και άλλους τύπους δεδομένων όπως δεκαδικούς, δυαδικούς, δεκαεξαδικούς. Για αυτό και είναι ο πιο εύχρηστος τρόπος.

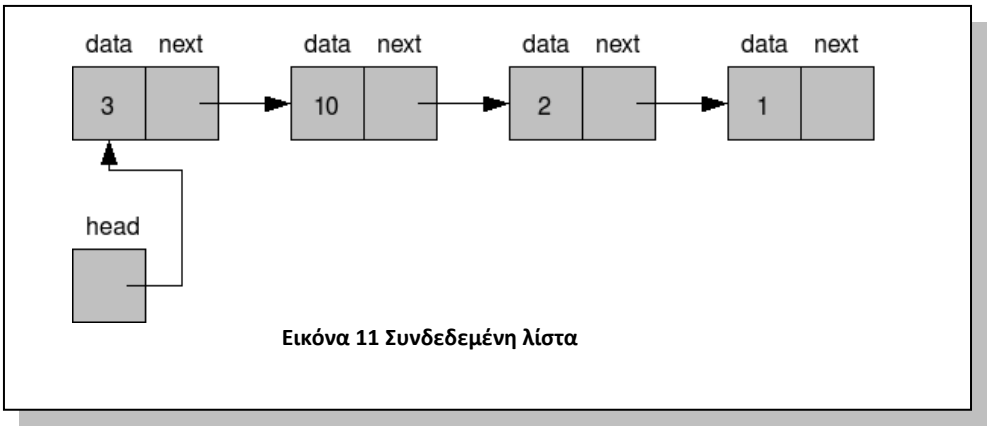
(4.5 Λίστες)

Οι λίστες της Python υλοποιούν δυναμικούς πίνακες, με έτοιμες συναρτήσεις για την επεξεργασία τους όπως στην εικόνα 10. Μια λίστα δηλώνεται δυναμικά χωρίς να χρειαστεί να δηλώσουμε τον τύπο των στοιχείων της και μέσα στην λίστα μπορούμε να έχουμε διαφορετικού τύπου στοιχεία όπως π.χ. στην λίστα `M = [ 1, 7.6, "hello"]` έχουμε έναν int, έναν float και ένα string.

Στην C ότι περιεγράφηκε πιο πάνω δεν υπάρχει και πρέπει ο/η προγραμματιστής/στρια να δημιουργήσει λίστες και συναρτήσεις που να ικανοποιούν τις ανάγκες του προγράμματος.

```
M = [1, 2, 4, 1, 3, 8, 6]
M.append(2);
# M = [1, 2, 4, 1, 3, 8, 6, 2]
M.remove(3)
# M = [1, 2, 4, 1, 8, 6, 2]
M.sort()
# M = [1, 1, 2, 2, 4, 6, 8]
M.reverse()
# M = [8, 6, 4, 2, 2, 1, 1]
M.clear()
# M = []
```

Εικόνα 10 Απλή χρήση λίστας στην Python



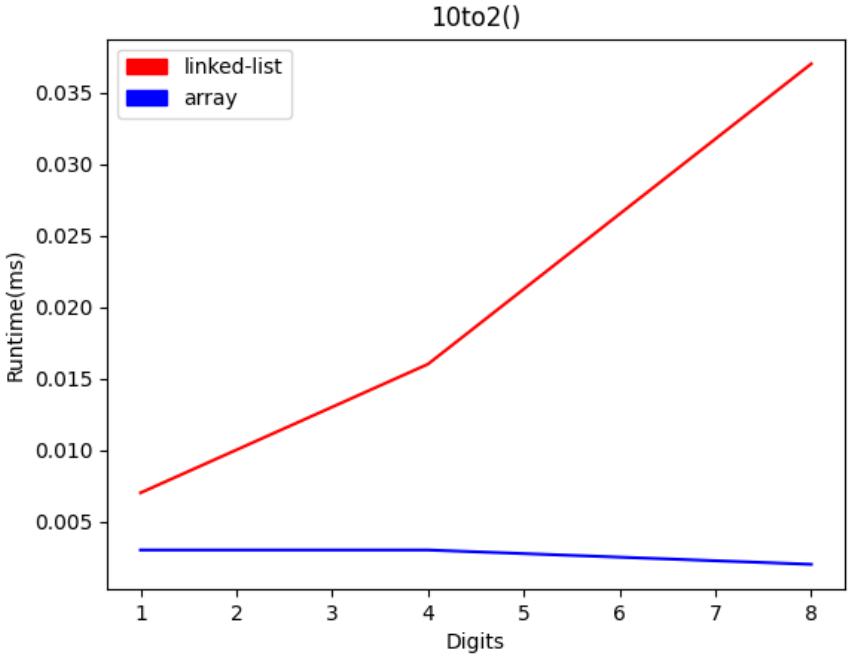
Εικόνα 11 Συνδεδεμένη λίστα

Μια προσέγγιση που να ικανοποιεί τις λειτουργίες της εικόνα 10 είναι οι συνδεδεμένες λίστες. Σε μια συνδεδεμένη λίστα υπάρχει ένας pointer που δείχνει την αρχή της λίστας και κάθε στοιχείο της λίστας έχει δύο τιμές, μια μεταβλητή **data** που μπορεί να είναι ότι τύπος δεδομένου χρειάζεται και ένας pointer **next** που δείχνει το επόμενο στοιχείο στην λίστα [18]. Αυτή η λύση είναι ικανοποιητική σε σχέση με στατικούς πίνακες όταν χρειαζόμαστε τα ακόλουθα πλεονεκτήματα:

- **Χρήση Μνήμης.** Αν δεν είναι γνωστό το μέγεθος του πίνακα από πριν συνήθως πρέπει να κάνουμε τον πίνακα αρκετά μεγάλο ώστε να χωράει όλα τα στοιχεία που θα βάλουμε. Αυτό μπορεί να αφήσει περιττό χώρο που δεν χρησιμοποιείτε και στην περίπτωση που ο χώρος δεν είναι αρκετός πρέπει να γίνει resize του πίνακα κάτι που είναι αργό. Η συνδεδεμένη λίστα δεν έχει τέτοιο πρόβλημα γιατί χρησιμοποιεί ακριβώς όσα **nodes** όσα υπάρχουν στοιχεία, με το τελευταίο node να είναι το τελευταίο στοιχείο. Στην περίπτωση που έχουμε έναν πλήρη στατικό πίνακα και δεν σπαταλάμε χώρο, η αντίστοιχη λίστα θα χρησιμοποιεί περισσότερο χώρο στην μνήμη γιατί κάθε node καταναλώνει περισσότερο χώρο γιατί χρειάζεται και έναν pointer.
- **Insertions και Deletions.** Όταν χρειάζεται να γίνουν πολλά **insertions** και **deletions** κάτι που δεν είναι χρονοβόρο στις συνδεδεμένες λίστες γιατί το μόνο που χρειάζεται και στις δύο περιπτώσεις είναι να δείξουμε τους αντίστοιχους **pointers** στα κατάλληλα σημεία, κάνοντας **traverse** την λίστα μέχρι τον κόμβο που ψάχνουμε όπου δημιουργούμε νέο κόμβο και ενημερώνουμε τους pointers στην περίπτωση του insertion και απλά ενημερώνοντας τους pointers στο deletion. Όταν γίνεται **insert** και **delete** στην αρχή και το τέλος της λίστας το κόστος είναι $\boldsymbol{O(1)}$, όταν όμως ο κόμβος δεν βρίσκεται στην αρχή ή στο τέλος το κόστος είναι γραμμικό με το μέγεθος της λίστας $\boldsymbol{O(n)}$ γιατί πρέπει να γίνει traverse της λίστας μέχρι να βρούμε τον κόμβο που αναζητούμε.

Οι συνδεδεμένες λίστες έχουν και τα ακόλουθα μειονεκτήματα σε σχέση με τους στατικούς πίνακες:

- **Αργή αναζήτηση**. Σε έναν στατικό πίνακα μπορούμε πολύ εύκολα να πάρουμε κάποιο στοιχείο του όποιο και να είναι αυτό παρέχοντας τον αριθμό. Στις λίστες δεν γίνεται αυτό γιατί πρέπει να γίνει προσπέλαση όλης της λίστας πριν φτάσουμε στο στοιχείο που επιθυμούμε, για παράδειγμα αν θέλουμε το στοιχείο 1024 τις λίστας θα πρέπει να διασχίσουμε τα όλα τα προηγούμενα 1023 στοιχεία της.



Εικόνα 11 Σύγκριση ταχύτητας στατικού πίνακα με συνδεδεμένη λίστα στην μετατροπή δεκαδικού σε δυαδικό αριθμό

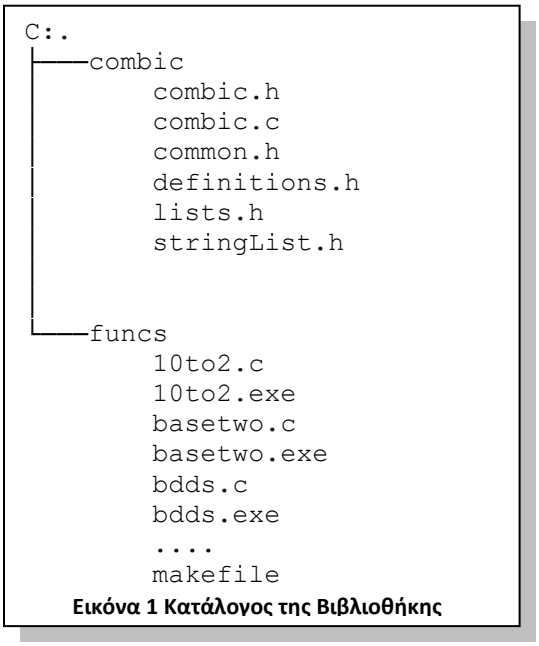
Στην περίπτωση της βιβλιοθήκης Combic ελέγχθηκε η διαφορά λιστών-πινάκων με την συνάρτηση μετατροπής ακεραίων-δυαδικών **10to2()**. Στην εικόνα 11 παρουσιάζονται δύο διαφορετικές εκδοχές τις συνάρτησης, μία με συνδεδεμένες λίστα και μία με στατικό πίνακα. Ο στατικός πίνακας είναι έως **18.5** φορές πιο γρήγορος και χρησιμοποιεί **2.016kb** αντί για **3516kb** .

(4.6 Συμπεράσματα)

Στην σημερινή εποχή και οι δύο γλώσσες θεμελιώνουν την θέση τους ως σημαντικά και δυνατά εργαλεία άξια αρκετά ώστε να χρησιμοποιούνται ευρέως. Η C διατηρεί την θέση της με την ταχύτητα της και τον κλασσικό δομημένο προγραμματισμό τον οποίο προτιμούν πολλοί/λες από τον αντικειμενοστραφή προγραμματισμό και βρίσκει χρήση στην δημιουργία λειτουργικών συστημάτων και μεταγλωττιστών. Η Python έχει ότι κάνει την C ελκυστική συστήνοντας ταυτόχρονα τις έννοιες του αντικειμενοστραφή προγραμματισμού με το μόνο αρνητικό την ταχύτητα της, το οποίο αν ο/η προγραμματιστής/τρια είναι αρκετά ικανός/νη γίνεται να ξεπεραστεί, βρίσκει ευρέως χρήση ως εκπαιδευτικό μέσο, γλώσσα scripting και για την δημιουργία γενικών εφαρμογών.

ΚΕΦΑΛΑΙΟ 5 Προγραμματισμός σε C

(5.1 Ο Κατάλογος Combic)



Εικόνα 1 Κατάλογος της Βιβλιοθήκης

Η βιβλιοθήκη Combic είναι το αρχείο που περιέχει όλες τις συναρτήσεις που δημιουργήθηκαν στην C και είναι ο συνδυασμός του Combinational και το Logic καθώς και τα προβλήματα που λύνει η βιβλιοθήκη ανήκουν στην **Συνδυαστική λογική**. Ο κατάλογος συμπεριλαμβάνει έναν φάκελο με τα αρχεία κεφαλίδας που χρειάζονται για την βιβλιοθήκη και ένα αρχείο κεφαλίδας το `combic.h` που περιλαμβάνει όλες τις συναρτήσεις σε ένα αρχείο για εύκολη ενσωμάτωση και χρήση σε άλλα προγράμματα, έναν φάκελο με ξεχωριστά αρχεία `.c` για κάθε συνάρτηση μαζί με τα εκτελέσιμα τους και τα αρχεία επαλήθευσης και δοκιμών. Παρακάτω αναφέρονται όλες οι συναρτήσεις, τι παίρνουν ως είσοδο και τι επιστρέφουν ως αποτέλεσμα:

- `char *detobi(unsigned long long int num)`
  Παίρνει ως είσοδο έναν δεκαδικό αριθμό και τον επιστρέφει σε δυαδική μορφή.
- `char *basetwo(char *num)`
  Ελέγχει αν η είσοδος είναι οκταδικός ή δεκαεξαδικός αριθμός και επιστρέφει την δυαδική μορφή.
- `void bdds(char *f, int x, char *y)`
  Παίρνει ως είσοδο μια συνάρτηση, τον αριθμό των μεταβλητών, τα σύμβολα των μεταβλητών και επιστρέφει το bdd.
- `int caland(int a, int b, int c, int d, int e)`
  Παίρνει ως είσοδο 5 μεταβλητές και υπολογίζει την **AND** πύλη τους.
- `int calnand(int a, int b, int c, int d, int e)`
  Παίρνει ως είσοδο 5 μεταβλητές και υπολογίζει την **NAND** πύλη τους.
- `int calnot(int number)`
  Παίρνει ως είσοδο 1 μεταβλητή και υπολογίζει την **NOT** πύλη της.
- `int calor(int a, int b, int c, int d, int e)`
  Παίρνει ως είσοδο 5 μεταβλητές και υπολογίζει την **OR** πύλη τους.

- `int calxnor(int a, int b, int c, int d, int e)`
  Παίρνει ως είσοδο 5 μεταβλητές και υπολογίζει την **XNOR** πύλη τους.
- `int calxor(int a, int b, int c, int d, int e)`
  Παίρνει ως είσοδο 5 μεταβλητές και υπολογίζει την **XOR** πύλη τους.
- `int calculateGate(int arg[], int argLen, char *gate)`
  Παίρνει ως είσοδο έναν πίνακα μεταβλητών, το μήκος του, τον τύπο πύλης και υπολογίζει την πύλη που πήρε ως είσοδο. Οι πύλες που μπορεί να υπολογίσει είναι οι :**AND, NAND, OR, XNOR, XOR**.
- `void conmulmaxsuminima(char *function)`
  Παίρνει ως είσοδο μια συνάρτηση σε γινόμενο αθροισμάτων και την μετατρέπει σε άθροισμα γινομένων.
- `void consuminimamulmax(char *s)`
  Παίρνει ως είσοδο μια συνάρτηση σε άθροισμα γινομένων και την μετατρέπει σε γινόμενο αθροισμάτων.
- `char *dedtobi(double number)`
  Παίρνει ως είσοδο έναν ακέραιο με δεκαδικό μέρος και τον επιστρέφει σε δυαδική μορφή.
- `int devbin(char *number)`
  Παίρνει ως είσοδο έναν δυαδικό αριθμό και επιστρέφει την δεκαδική αναπαράσταση του.
- `void devdem(char *number)`
  Παίρνει ως είσοδο έναν ακέραιο αριθμό και επιστρέφει την δεκαδική αναπαράσταση του.
- `int devhex(char *number)`
  Παίρνει ως είσοδο έναν δεκαεξαδικό αριθμό και επιστρέφει την δεκαδική αναπαράσταση του.
- `int devoct(char *number)`
  Παίρνει ως είσοδο έναν οκταδικό αριθμό και επιστρέφει την δεκαδική αναπαράσταση του.
- `char *karnfour(int minterms[16])`
  Παίρνει ως είσοδο 16 ελαχιστόρους και λύνει χάρτη Karnaugh 4 μεταβλητών.
- `char *karnfoursitu(int minterms[16], char *donts)`
  Παίρνει ως είσοδο 16 ελαχιστόρους και λύνει χάρτη Karnaugh 4 μεταβλητών με συνθήκες αδιαφορίας.
- `char *karnthree(int a, int b, int c, int d, int e, int f, int g, int h)`
  Παίρνει ως είσοδο 8 ελαχιστόρους και λύνει χάρτη Karnaugh 3 μεταβλητών.
- `char *karnthreesitu(int a, int b, int c, int d, int e, int f, int g, int h, char *kat)`
  Παίρνει ως είσοδο 8 ελαχιστόρους και λύνει χάρτη Karnaugh 3 μεταβλητών με συνθήκες αδιαφορίας.
- `char *karntwo(int a, int b, int c, int d)`
  Παίρνει ως είσοδο 4 ελαχιστόρους και λύνει χάρτη Karnaugh 2 μεταβλητών.
- `char *karntwositu(int a, int b, int c, int d, char *kat)`
  Παίρνει ως είσοδο 4 ελαχιστόρους και λύνει χάρτη Karnaugh 2 μεταβλητών με συνθήκες αδιαφορίας.
- `char *mcluskey(int *minterms, int num_minterms, int num_vars, int *donts, int num_donts)`
  Παίρνει ως είσοδο την λίστα με τους ελαχιστόρους, τις συνθήκες αδιαφορίας, των αριθμό μεταβλητών και επιστρέφει την ελαχιστοποιημένη συνάρτηση με την μέθοδο Quine-McCluskey.
- `char *mulmax(char *s)`
  Παίρνει ως είσοδο μια συνάρτηση και την μετατρέπει σε γινόμενο αθροισμάτων.
- `void num2421(unsigned long long int num)`
  Παίρνει ως είσοδο έναν ακέραιο και επιστρέφει την δυαδική αναπαράσταση του σε κώδικα 2421.

- `char *numbcd(char *number)`
  Παίρνει ως είσοδο έναν ακέραιο και επιστρέφει την δυαδική αναπαράσταση του σε κώδικα BCD.
- `char *numbiqu(char *number)`
  Παίρνει ως είσοδο έναν ακέραιο και επιστρέφει την δυαδική αναπαράσταση του σε κώδικα Biquinary.
- `char *numgray(char *number)`
  Παίρνει ως είσοδο έναν ακέραιο και επιστρέφει την δυαδική αναπαράσταση του σε κώδικα Gray.
- `char *quine_mccluskeysoi(char *vars, int *minterms, int *donts)`
  Παίρνει ως είσοδο τα σύμβολα των μεταβλητών και λίστα με ελαχιστόρους και επιστρέφει την ελαχιστοποιημένη συνάρτηση με την μέθοδο Quine-McCluskey με συνθήκες αδιαφορίας.
- `int retwo(char *number)`
  Παίρνει ως είσοδο έναν αριθμό και ελέγχει αν έχει δεκαδικό μέρος και τον επιστρέφει ως δυαδικό.
- `char *subase(char *a, char *b)`
  Παίρνει ως είσοδο δύο δυαδικούς αριθμούς και τους αφαιρεί με την μέθοδο συμπληρώματος βάσης.
- `char *suminima(char *s)`
  Παίρνει ως είσοδο μια συνάρτηση και την μετατρέπει σε άθροισμα ελαχιστόρων.
- `char *supone(char *number)`
  Παίρνει ως είσοδο έναν δυαδικό αριθμό και επιστρέφει το συμπλήρωμα ως προς ένα.
- `char *suptwo(char *number)`
  Παίρνει ως είσοδο έναν δυαδικό αριθμό και επιστρέφει το συμπλήρωμα ως προς δύο.
- `void truthtable(char *s, int y)`
  Παίρνει ως είσοδο μια συνάρτηση και το πλήθος μεταβλητών και επιστρέφει τον πίνακα αληθείας. Επιτρεπόμενοι αριθμοί μεταβλητών είναι οι 2, 3, 4.

(5.2 Όλα τα εργαλεία που χρησιμοποιήθηκαν)

Κατά την διάρκεια ανάπτυξης της βιβλιοθήκη χρησιμοποιήθηκαν διάφορα εργαλεία και βιβλιοθήκες για την δημιουργία, επαλήθευση και βελτίωση κώδικα. Χρησιμοποιήθηκαν οι εξής βιβλιοθήκες:

- `stdio.h`
  Βασική βιβλιοθήκη του GCC περιέχει βασικές συναρτήσεις όπως την `printf()`.
- `stdlib.h`
  Βασική βιβλιοθήκη του GCC περιέχει συναρτήσεις για την μετατροπή τύπων, διαχείριση μνήμης και βασικές συνηθής συναρτήσεις όπως η `rand()`.
- `string.h`
  Βασική βιβλιοθήκη του GCC περιέχει συναρτήσεις για τον χειρισμό αλφαριθμητικών όπως οι `strcpy()`, `strlen()`, `strcat()` και `strcmp()`.
- `math.h`
  Βασική βιβλιοθήκη του GCC περιέχει συναρτήσεις για μαθηματικές πράξεις όπως τριγωνομετρικές πράξεις και δυνάμεις όπως με `cos()`, `sin()`, `pow()`. Χρειάζεται το **-lm** ως argument για μεταγλώττιση.
- `ctype.h`
  Βασική βιβλιοθήκη του GCC περιέχει συναρτήσεις για τον έλεγχο χαρακτήρων με συναρτήσεις όπως `isupper()`, `islower()`, `isdigit()`.
- `stdbool.h`
  Βασική βιβλιοθήκη του GCC περιέχει μακροεντολές για τον ορισμό τύπου δεδομένων Boolean.

- `omp.h`
  Βιβλιοθήκη ανεπτυγμένη από την Intel(28) για την παραλληλοποίηση κώδικα.

Από εργαλεία χρησιμοποιήθηκαν τα εξής:

- `Valgrind(47)`
  Προγραμματιστικό εργαλείο για την αποσφαλμάτωση μνήμης, την εύρεση διαρροών μνήμης και για profiling.
- `Cppcheck(30)`
  Στατικό εργαλείο ανάλυσης για τις γλώσσες C/C++, χρησιμοποιείται για την εύρεση λαθών σε κώδικα και την επικίνδυνου κώδικα.
- `Gdb(31)`
  Χρησιμοποιείτε για την αποσφαλμάτωση κώδικα με την χρήση breakpoints που επιτρέπει την εκτέλεση κώδικα γραμμή-γραμμή.
- `Gcc(22)`
  Ένας από τους πιο διάσημους μεταγλωττιστές για την γλώσσα C.
- `Makefile(32)`
  Πρόγραμμα για την κατασκευή και εγκατάσταση προγραμμάτων, βοηθάει στην αυτοματοποίηση μεταγλώττισης πολύπλοκων προγραμμάτων τα οποία έχουν πολλά αρχεία πηγής.

(5.3 Επαλήθευση)

Σε απλό κώδικα όπου τα πιθανά αποτελέσματα είναι λίγα, είναι εύκολο να ελεγχθεί αν δεν έχει σφάλματα αλλά αν έχουμε πολύπλοκο κώδικα με πολλούς συνδυασμούς εισόδων-εξόδων τότε γίνεται ιδιαίτερα δύσκολο να επαληθευτεί η ακρίβεια του.

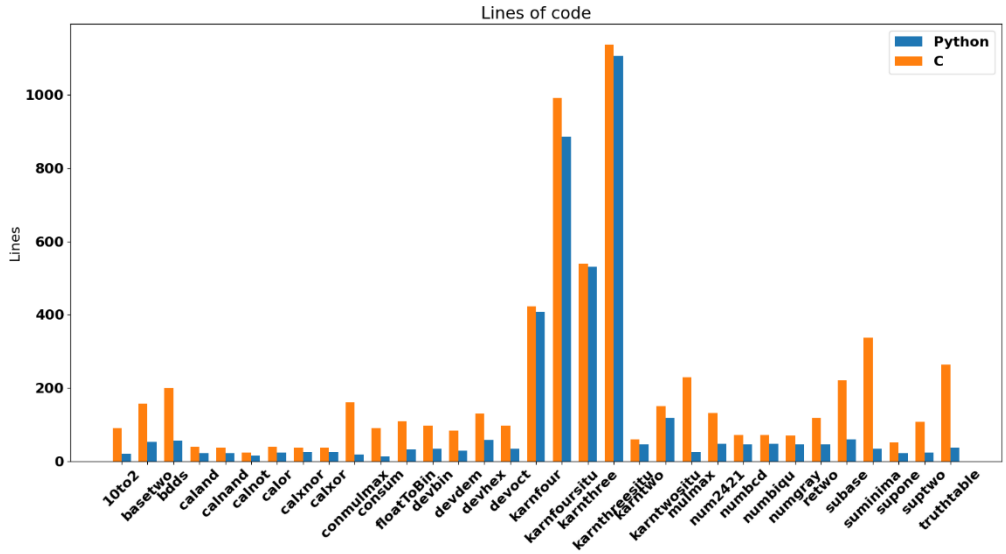
Για να επαληθεύσουμε τέτοιο κώδικα πρέπει να ελεγχθεί σε μεγάλο εύρος εισόδων και αυτό μπορεί να γίνει με έναν από δυο τρόπους. Ένας τρόπος είναι η χρήση μιας βιβλιοθήκης δοκιμασιών (unit testing framework) όπως το Unity(33) ή το GoogleTest(34), αυτή η μέθοδος είναι πιο χρήσιμη για μεγαλύτερα έργα και είναι πιο διατηρήσιμη και αυτοματοποιημένη. Ο άλλος τρόπος είναι να δημιουργήσουμε εμείς κώδικα που δοκιμάζει το πρόγραμμα μας.

Στην περίπτωση της Combic υπάρχει η έκδοση της Python που μπορούμε να συγκρίνουμε το αποτέλεσμα για να επιβεβαιωθεί ότι το αποτέλεσμα είναι σωστό. Για να χρησιμοποιήσουμε τις ήδη υλοποιημένες εκδόσεις της Python μπορούμε να εκτελέσουμε και τις δυο υλοποιήσεις με την ίδια είσοδο και να παρατηρήσουμε αν έχουν την ίδια έξοδο. Αυτή την λειτουργεία επιτρέπει το αρχείο κεφαλίδας `valid.h` το οποίο επιτρέπει με την συνάρτηση `isValid()` να συγκρίνουμε το αποτέλεσμα των δύο εκδόσεων. Ο τρόπος που λειτουργεί η συνάρτηση είναι:

- Παίρνει ως είσοδο δύο αλφαριθμητικά ένα η εντολή για εκτέλεση και το άλλο το αποτέλεσμα που θέλουμε να συγκρίνουμε, η εντολή στην σε αυτήν την περίπτωση θα είναι για να εκτελεστεί η έκδοση της Python και θα έχει την μορφή `"python3 ../python/10to2.py <args>"` και το αποτέλεσμα θα είναι η εκδοχή της C.
- Αφού εκτελέσει την εκδοχή της Python θα πάρει πίσω το αποτέλεσμα και θα το συγκρίνει με το αποτέλεσμα της C, αν διαφέρουν θα εκτυπώσει ότι διαφέρουν και αν είναι ίδια θα εκτυπώσει αντίστοιχα.

Με αυτή τη διαδικασία έγινε η επαλήθευση των αποτελεσμάτων σε συναρτήσεις όπου υπήρχαν αμφιβολίες για αποτελέσματα με πολλές και διάφορες εισόδους όπου η εξέταση τους μια-μια είναι αδύνατη.

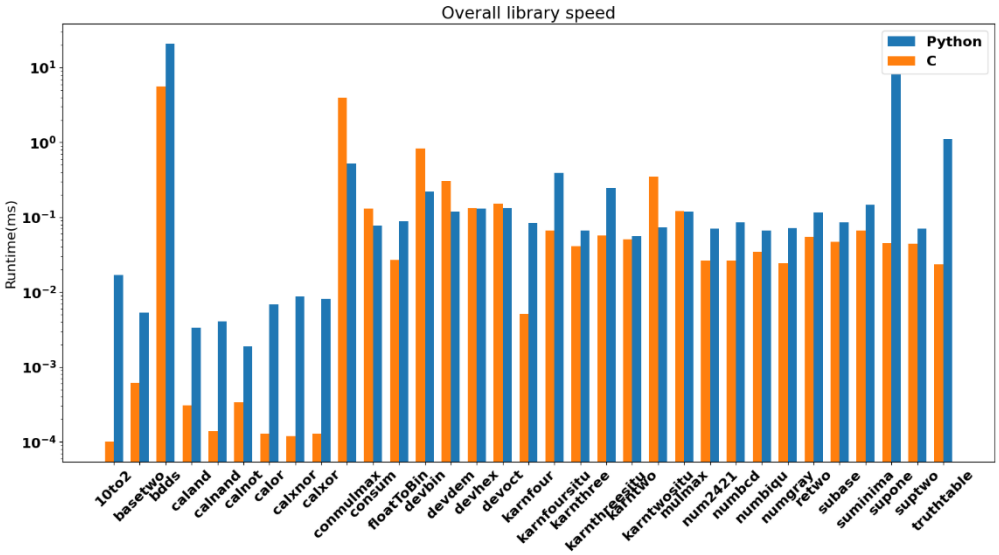
(5.4 Γραμμές κώδικα)



Εικόνα 2 Γράφημα που δείχνει πόσες γραμμές κώδικα χρειάστηκαν για κάθε γλώσσα

Αναφέρθηκε στο κεφ.4 στα πλεονεκτήματα και μειονεκτήματα κάθε γλώσσας ότι η το συντακτικό της Python είναι πιο απλό και καταλήγει σε μικρότερο μήκους κώδικα. Αυτό είναι κάτι που μπορούμε να δούμε με το γράφημα της εικόνας 2 που δείχνει τις γραμμές που χρησιμοποιήθηκαν. Η υλοποίηση των συγκεκριμένων συναρτήσεων είναι παρόμοια και κάνει την σύγκριση ανάμεσα στις γλώσσες έγκυρη, ενώ χρειάστηκε να αφαιρεθούν από το γράφημα οι συναρτήσεις qm.c, gatecal.c, mcmluskey.py και quine_mccluskeysoi.py καθώς έχουν διαφορετική λειτουργικότητα και θα αναλυθούν περεταίρω.

(5.5 Επιδόσεις)



Εικόνα 3 Διαφορά ταχύτητας ανάμεσα στην C και στην Python, λογαριθμική κλίμακα

Στην εικόνα 3 το γράφημα δείχνει την διαφορά στον χρόνο εκτέλεσης στην C και στην Python, η κλίμακα είναι λογαριθμική και οι χρόνοι είναι μετρημένοι σε **ms.** Η C για τις περισσότερες

συναρτήσεις είναι πιο γρήγορη κάτι που είναι αναμενόμενο αλλά υπάρχουν και περιπτώσεις που η Python είναι ίση ή πιο γρήγορη όπως των conmulmaxsuminima και devbin. Η πιο πιθανή εξήγηση για την διαφορά είναι διαφορετική υλοποίηση στον τρόπο επίλυσης. Επίσης όλες οι συναρτήσεις κυμαίνονται σε χρόνο εκτέλεσης της τάξης των ms κάτι που δεν είναι αρκετό για να δοκιμάσουμε την ταχύτητα των δύο γλωσσών.

(5.6 Βελτιώσεις)

Πέρα από τις βελτιώσεις σε ταχύτητα και χρήση μνήμης έγιναν μερικές ακόμα βελτιώσεις στην χρηστικότητα μερικών συναρτήσεων.

- Οι συναρτήσεις `caland.c, calnand.c, calnot.c, calor.c, calxnor.c, calxor.c` παίρνουν ως είσοδο 5 μεταβλητές και δίνουν ως έξοδο την λογική πύλη που αντιστοιχεί στο όνομα της συνάρτησης που καλούμε. Για απλοποίηση και διευκόλυνση της χρήσης αυτών των συναρτήσεων δημιουργήθηκε μια νέα συνάρτηση `gatecal.c` η οποία έχει τις εξής εισόδους:
  - μεταβλητό μήκος μεταβλητών ως int array
  - το μέγεθος του πίνακα
  - την πύλη που θέλουμε να χρησιμοποιήσουμε ως string

  Η νέα συνάρτηση συνδυάζει τις προηγούμενες 6 συναρτήσεις και αφαιρεί τον περιορισμό 5 μεταβλητών.
- Οι συναρτήσεις `quine_mcluskey.c` και `quine_mcluskeysoi.c` υπολογίζουν την ελαχιστοποιημένη λογική έκφραση με δεδομένο τους ελαχιστόρους που δίνονται ως είσοδο μαζί με τις συνθήκες αδιαφορίας αν υπάρχουν. Επειδή ο υπολογισμός της μεθόδου Quine McCluskey με συνθήκες αδιαφορίας γίνεται εύκολα με μία αλλαγή στην μέθοδο `quine_mcluskey.c` δεν χρειάζεται να υπάρχει η `quine_mcluskeysoi.c` και μπορούν να συνδυαστούν σε μία συνάρτηση `qm.c` η οποία ανάλογα με την είσοδο μπορεί να υπολογίσει και τις δύο εκδοχές.

  Ο συνδυασμός των δύο συναρτήσεων όμως δεν είναι η μόνη βελτίωση της `qm.c` καθώς εκτυπώνει και τα ενδιάμεσα βήματα της μεθόδου Quine McCluskey με λεπτομέρεια ενώ η αρχική συνάρτηση στην Python δίνει μόνο την τελική έκφραση, παράδειγμα για την εκτύπωση που κάνει η `qm.c` φαίνεται στην εικ.[3] του κεφ.3.5 όπου εξηγείτε η αντίστοιχη μέθοδος θεωρητικά στα πλαίσια της Λογικής Σχεδίασης.

(5.7 Χρήσιμα Εργαλεία)

(5.7.1 Κανονικές Εκφράσεις)

Κατά την ανάπτυξη της βιβλιοθήκης υπήρξαν περιπτώσεις όπου η αλλαγή εκφράσεων από μία γλώσσα στην άλλη όπως με τις δηλώσεις **`if`** `(if statements)` έκαναν την επιλογή αλλαγής με το χέρι αδύνατη καθώς ο αριθμός είναι πολύ μεγάλος. Τέτοιες περιπτώσεις βρέθηκαν στις ακόλουθες συναρτήσεις: `karnfour(), karnfoursitu(), karnthree(), karnthreesitu(), karntwo(), karntwositu(), num2421(), numbcd(), numbiqu(), numgray(), truthtable()`

Όλες οι παραπάνω συναρτήσεις έχουν τμήματα κώδικα της μορφής(εικ.[2]):

```
1   if   a==0 and b==0 and c==0 and d==0:
2        k = "0"
3   elif a==0 and b==0 and c==0 and d==0:
4        k = "xyz"
5   elif a==0 and b==0 and c==0 and d==0:
6        k = "xyz'"
7   ....
    ....
500 elif a==1 and b==1 and c==1 and d==1:
501      k = "x' + y' + z'"
502 elif a==1 and b==1 and c==1 and d==1:
503      k = "1"
504 return k
```

Εικόνα 4 Παράδειγμα πολλών if statements σε σειρά

Με τέτοιο αριθμό είναι αδύνατο να μετατραπούν στην μορφή που χρειάζεται η C κάνοντας αλλαγή κάθε `if statement` ένα-ένα. Μια λύση στο πρόβλημα είναι να δημιουργήσουμε ένα πρόγραμμα που να διαβάζει ένα .txt αρχείο που να έχει το τμήμα κώδικα που θέλουμε να αλλάξουμε και να επεξεργαστούμε το τμήμα σαν αλφαριθμητικό. Μια άλλη λύση είναι να προσπαθήσουμε να αλλάξουμε τον υπολογισμό του κώδικα της εικόνας 4 ώστε να μην χρειάζεται τόσα `if statements` για να βρει το αποτέλεσμα. Η πιο ευλύγιστη και καλή σε αυτήν την περίπτωση λύση όμως είναι να χρησιμοποιήσουμε μια **Κανονική Έκφραση(RegX)**.

Οι κανονικές εκφράσεις ή αλλιώς RegX είναι μια σειρά από χαρακτήρες οι οποία έχει μια ή περισσότερες αντιστοιχίες σε ένα κείμενο. Τα RegX χρησιμοποιούνται ευρέως σε μηχανές αναζήτησης, επεξεργαστές κειμένου και λεκτικές αναλύσεις [19], αλλά μπορούν να βρουν απλή εφαρμογή σε κείμενο όπως της εικόνας 4.

Για να μπορέσουμε να μετατρέψουμε το κείμενο της εικόνας 4 με RegX πρέπει να επιβεβαιώσουμε πρώτα ότι ο επεξεργαστής κώδικα που χρησιμοποιούμε υποστηρίζει τις κανονικές εκφράσεις. Στην περίπτωση του Visual Studio Code τα RegX υποστηρίζονται και βρίσκονται μαζί με την κανονική ‘’βρες και αντικατέστησε’’(find and replace) δυνατότητα.

Τα RegX έχουν δικό τους συντακτικό με φράσεις που βρίσκουν συγκεκριμένα αλφαριθμητικά όπως υποδεικνύεται παρακάτω:

- `.` η τελεία αντιστοιχεί οποιοδήποτε χαρακτήρα με εξαίρεση την νέα γραμμή.
- `\w \d \s` αντιστοιχούν γράμμα, ψηφίο, κενό αντίστοιχα.
- `*` αντιστοιχεί την προηγούμενη φράση 0 ή περισσότερες φορές.
- `+` αντιστοιχεί την προηγούμενη φράση 1 η περισσότερες φορές.
- `()` με τις παρενθέσεις ορίζουμε μια υπό-φράση την οποία μπορούμε να καλέσουμε το περιεχόμενο μετέπειτα.

Στην περίπτωση της εικόνας 4 μπορούμε για να βρούμε τα `if statements` να βάλουμε στο πεδίο ‘’βρες’’(find) το RegX:

```
(elif a==)(\d)( and b==)(\d)( and c==)(\d)( and d==)(\d)( and e==)(\d)( and f==)(\d)( and g==)(\d)( and h==)(\d)(:)
```

Επειδή το μόνο που αλλάζει είναι ο αριθμός που συγκρίνονται οι μεταβλητές ξεχωρίζουμε με παρενθέσεις σε ομάδες το `if statement` για να διατηρηθούν οι αριθμοί όταν αλλαχθούν τα `if statements`.Για να μετατραπεί στην μορφή της C μπορούμε να χρησιμοποιήσουμε στο πεδίο ‘’αντικατάσταση’’(replace) πεδίο το RegX:

```
else if(a==$2 && b==$4 && c==$6 && d==$8 && e==$10 && f==$12 && g==$14 && h==$16)
```

Γράφουμε το elif με την αντίστοιχη μορφή της C και όπου χρειάζεται να διατηρηθεί το αρχικό περιεχόμενο κάνουμε αναφορά με το $ και τον αριθμό ομάδας.

Οι κανονικές εκφράσεις είναι ένα χρήσιμο εργαλείο για την επεξεργασία μεγάλων κειμένων όπως τον κώδικα ενός προγράμματος αλλά και οποιοδήποτε άλλο κείμενο μπορεί να χρειαστεί εύρεση και αλλαγή, σε κείμενα όπου η απλή εύρεση αυτούσιας έκφρασης δεν είναι αρκετή και πρέπει να εισάγουμε προγραμματιστική λογική τα RegX είναι η λύση.

(5.7.2 Windows Subsystem for Linux)

Το Windows Subsystem for Linux ή αλλιώς WSL(35) είναι μια δυνατότητα των Windows(36) να εκτελεί το περιβάλλον Linux(37) μέσα τους χωρίς την ανάγκη για εγκατάσταση κάποιας διανομής των Linux και την χρήση εικονικού υπολογιστή. Η πρώτη έκδοση του WSL βγήκε 2 Αυγούστου 2016 και εφάρμοζε τα καλέσματα συστήματος του Linux μέσα από τον kernel των Windows.

Τον Ιούνιο του 2019 βγήκε η δεύτερη έκδοση WSL2 η οποία βελτίωσε σημαντικά τις δυνατότητες του WSL εφαρμόζοντας τον πραγματικό kernel των Linux μέσα από ένα εικονικό μηχάνημα επιτρέποντας το κάλεσμα πραγματικών Linux κλήσεων συστήματος και βελτιώνοντας την ταχύτητα εκτέλεσης προγραμμάτων και πρόσβασης αρχείων.

Υπάρχουν περιπτώσεις στην ανάπτυξη λογισμικού όπου είναι αναγκαία η πρόσβαση σε άλλο λειτουργικό σύστημα όπως το Linux καθώς η χρήση συγκεκριμένων βιβλιοθηκών μπορεί να γίνει μόνο σε αυτό. Στην περίπτωση της βιβλιοθήκης Combic η πρώτη προσπάθεια παραλληλοποίησης έγινε με τα **pthreads**, τα οποία είναι UNIX και δεν είναι διαθέσιμα στα Windows. Οι συνηθής επιλογές μέχρι και τον ερχομό του WSL ήταν είτε ή χρήση εικονικού υπολογιστή είτε της ''διπλής εκκίνησης''(dual booting) κάτι που είναι δύσχρηστο και δεν είναι καλή εμπειρία καθώς είτε η ταχύτητα δεν είναι ίδια στην πρώτη περίπτωση είτε το περιβάλλον εργασίας χωρίζεται στα δύο στην δεύτερη περίπτωση. Με την χρήση όμως του WSL λύνονται και τα δύο προβλήματα. Για να χρησιμοποιηθεί το WSL ακολουθούνται τα εξής βήματα:

- Εγκατάσταση ανοίγοντας ένα **powershell**(38) παράθυρο με δικαιώματα διαχειριστή και πληκτρολογώντας την εντολή `wsl -install` ,με τις αυτόματες επιλογές θα γίνει εγκατάσταση του WSL2 και του **Ubuntu**(39) αλλά δίνεται η επιλογή να γίνει εγκατάσταση του WSL1 και άλλων διανομών.
- Αφού γίνει η εγκατάσταση θα ζητηθεί η δημιουργία λογαριασμού UNIX με την εισαγωγή ονόματος χρήστη και κωδικού.
- Μετά είναι καλή πρακτική να γίνει ενημέρωση των πακέτων του Ubuntu με την εντολή `sudo apt update && sudo apt upgrade` .
- Τώρα είναι δυνατή η χρήση του τερματικού Linux εκτελώντας το **wsl.exe** με την ίδια εμπειρία που θα είχαμε αν ήταν εγκατεστημένα τα Ubuntu αντί τα Windows στον υπολογιστή μας.

Για την καλύτερη χρήση του WSL προτείνεται η χρήση του Visual Studio Code το οποίο προσφέρει την εκτέλεση κώδικα μέσα στο WSL με την ίδια ταχύτητα και συγχώνευση του περιβάλλοντος WSL για εύκολη πρόσβαση αρχείων.

ΚΕΦΑΛΑΙΟ 6 Πολυπρογραμματισμός με OpenMP[2]

(6.1 Εισαγωγή)

Μια προσπάθεια που έγινε να βελτιωθεί ο κώδικας που δόθηκε στην Python είναι η εξερεύνηση παράλληλου κώδικα και πως μπορεί να μας επιταχύνει την εκτέλεση σε προβλήματα όπου μπορεί να εφαρμοστεί. Ο αρχικός κώδικας σε Python φυσικά είναι σειριακός, καθώς και είναι ο κώδικας στην C όπου έγινε η μεταφορά. Σε μια εφαρμογή όπως αυτές που φτιάχτηκαν για την επίλυση των προβλημάτων της Λογικής Σχεδίασης, ο κώδικας εκτελείτε σε ένα νήμα εκτέλεσης.

(6.2 Σειριακός και Πολυνηματικός Κώδικας)

Είναι σημαντική να διακριθεί ο σειριακός και ο πολυνηματικός κώδικας, καθώς έχει μεγάλη σημασία που πρέπει να χρησιμοποιηθούν και πια είναι τα πλεονεκτήματα του καθενός. Σειριακός κώδικας είναι αυτός που έχει μόνο ένα νήμα το κεντρικό ή αλλιώς το νήμα γονέας όπου έχει την δικιά του στοίβα και τρέχει σειριακά από την αρχή μέχρι το τέλος. Οι μοντέρνοι επεξεργαστές όμως

[2] Η εξήγηση της βιβλιοθήκης OpenMP και γιατί είναι χρήσιμη καθώς και τα παραδείγματα ακολουθούν την καταπληκτική σειρά από βίντεο tutorial του Tim Mattson ο οποίος είναι ένας από τους δημιουργούς της OpenMP στην Intel(https://www.youtube.com/@OpenMPARB)

έρχονται με περισσότερους από έναν πυρήνες και μπορούμε να τους χρησιμοποιήσουμε για να επιταχύνουμε τον κώδικα μας.

Ένα νήμα μπορεί να τρέξει μόνο σε έναν πυρήνα[3] αλλά αν το πρόβλημα που προσπαθούμε να επιλύσουμε παίρνει ώρα να επιλυθεί δηλαδή λεπτά ή ώρες γίνεται να προσπαθήσουμε να το επιταχύνουμε δημιουργώντας επιπλέον νήματα. Για να έχουμε παράλληλο κώδικα θα πρέπει να σπάσουμε το πρόβλημα σε κομμάτια και να τα δώσουμε στα νήματα, τα οποία θα τρέχουν στον ίδιο χρόνο, αλλιώς αν τα νήματα δεν τρέχουν στο ίδιο χρονικό περιθώριο αυτό που καταφέρνουμε λέγεται concurrency και δεν βοηθάει στην επιτάχυνση της εύρεσης λύσης του προβλήματος.

(6.3 Ιστορική Αναδρομή)

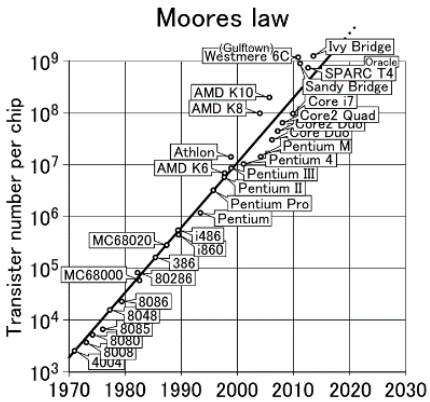
Η OpenMP(46) είναι ένα σετ από οδηγίες μεταγλωττιστή, αλλά και ένα API το οποίο παρέχει υποστήριξη για πολύ-επεξεργασία σε περιβάλλοντα με κοινή μνήμη. Είναι διαθέσιμη σε **FORTRAN**(40), **C** και **C++** και υποστηρίζεται από μια μεγάλη λίστα Αρχιτεκτονικών Υπολογιστών και Λειτουργικών Συστημάτων όπως **Solaris**(41), **AIX**(42), **FreeBSD**(43), **HP-UX**(44), **Linux, macOS**(45), **Windows**.

Εικόνα 1 Το επίσημο λογότυπο της OpenMP

Ουσιαστικά η OpenMP μας παρέχει τα εργαλεία που χρειαζόμαστε για να δημιουργήσουμε παράλληλο κώδικα. Τι είναι όμως παράλληλος κώδικας και γιατί είναι κάτι που πρέπει να μας ενδιαφέρει?

(6.4 Moore's Law)

Για να δούμε την χρησιμότητα της OpenMP πρέπει πρώτα να μιλήσουμε για τον Νόμο του Moore, οποίος διαπιστώνει ότι **κάθε 2 χρόνια ο αριθμός των transistor σε ένα ενσωματωμένο κύκλωμα(όπως ένας CPU) διπλασιάζεται**. Έχοντας πάρει το όνομα του από τον συνιδρυτή της Intel, τον Gordon Moore, ο νόμος του Moore είναι μια εμπειρική παρατήρηση και δεν είναι βασισμένος σε κάποιο νόμο της Φυσικής, αλλά βασίζεται σε απόκτηση εμπειρίας του Gordon Moore, από την παραγωγή επεξεργαστών στην Intel. [20]



Εικόνα 2 Ο νόμος του Moore 1970-2011 (Wikimedia commons)

Αρχικά όταν διαπιστώθηκε αυτή η παρατήρηση το 1965, έλεγε ότι ο θα γίνεται διπλασιασμός των transistor κάθε χρόνο για την επόμενη δεκαετία μέχρι το 1975. Το 1975 ο Gordon Moore ανανέωσε τον νόμο όπως περιεγράφηκε πιο πάνω, όπου και παρέμεινε έτσι για τα τελευταία 47 χρόνια. Ο νόμος ισχύει ακόμα γιατί με κάθε καινούργια γενιά επεξεργαστών, βελτιώσεις στην αρχιτεκτονική και η σμίκρυνση των transistor, μας δίνει το αποτέλεσμα που περιμένουμε, δηλαδή τον διπλασιασμό των transistor και με αποτέλεσμα την βελτίωση στις επιδώσεις.

Τα τελευταία χρόνια όμως όσο τα transistor γίνονται πιο μικρά και πυκνά και μπορούμε να βάλουμε περισσότερα σε ένα κύκλωμα, ένα νέο πρόβλημα εμφανίστηκε, ένα πρόβλημα το οποίο μας δείχνει ότι δεν θα μπορούμε πλέον να βασιστούμε στον νόμο του Moore και την ραγδαία αύξηση στις επιδόσεις του υλικού και αυτό είναι η **κατανάλωση ενέργειας.**

[3] Επειδή μια διαδικασία έχει ένα νήμα και τρέχει σειριακά δεν σημαίνει ότι θα τρέχει στον ίδιο πυρήνα που άρχισε, το λειτουργικό σύστημα θα την αναθέτει σε διαφορετικούς πυρήνες ανάλογα με τις ανάγκες του συστήματος, αυτό ονομάζεται concurrency.

(6.5 To Power Wall)

Όπως περιεγράφηκε στην προηγούμενη παράγραφο ο νόμος του Moore δεν είναι κάτι που μπορούμε να βασιστούμε πλέον. Για να φτιαχτούν γρηγορότεροι επεξεργαστές πρέπει να δημιουργηθούν καλύτερες σωληνώσεις επεξεργαστών, το οποίο μπορούμε να κάνουμε βάζοντας επιπλέον transistor στο κύκλωμα, αλλά **μεγαλώνοντας τον αριθμό των transistor, μεγαλώνουμε και την κατανάλωση ενέργειας**. Πιο απλά καθώς οι αρχιτεκτονικές υπολογιστών εξελίσσονται και μας δίνουν περισσότερες δυνατότητες και επιδόσεις, τόσο αυξάνεται και η ενέργεια που χρειάζεται να καταναλώσουμε. Επειδή όμως οι λύσεις μας για την ψύξη των επεξεργαστών είναι περιορισμένες, αυτό το σχέδιο δεν μπορεί να συντηρηθεί. Ότι είπαμε μέχρι τώρα ισχύει για σειριακά προγράμματα, πάμε να δούμε τι μπορεί να γίνει αν μπορούμε να κάνουμε ένα πρόγραμμα πολύ-νηματικό και πως αυτό μας βοηθάει με την κατανάλωση ενέργειας.

Σύμφωνα με τον Tim Mattson[21] από την Intel αν έχουμε ένα κύκλωμα που παίρνει κάποια είσοδο και δίνει κάποια έξοδο με $\boldsymbol{C = Χωρητικότητα}$ την ικανότητα του να κρατάει ενέργεια και είναι ίσο με $\boldsymbol{C = \frac{q}{V}}$ όπου **q** η φόρτιση και $\boldsymbol{V}$ η τάση. Η δουλεία σε ενέργεια που χρησιμοποιεί το κύκλωμα είναι $\boldsymbol{V * q = W}$ το οποίο σε συνδυασμό με την χωρητικότητα μας δίνει $\boldsymbol{W = CV^2}$**.** Επειδή θέλουμε να δούμε την δουλεία στο πέρασμα του χρόνου, ή αλλιώς στο κύκλωμα μας όταν πόσες φορές ανεβάζουμε και κατεβάζουμε το ρολόι, πολλαπλασιάζουμε με την συχνότητα και παίρνουμε $\boldsymbol{Power = W * f = CV^2f}$**.** Αν το ένα κύκλωμα έχει ένα πυρήνα και για κάποιες εισόδους μας δώσει κάποιες εξόδους με δεδομένη την χωρητικότητα $\boldsymbol{C}$, την τάση $\boldsymbol{V}$, και την συχνότητα $\boldsymbol{f}$ τότε ή κατανάλωση θα είναι $\boldsymbol{Power_1 = CV^2f}$**.** Αν αντίθετα έχουμε ένα κύκλωμα με δύο πυρήνες οι οποίοι τρέχουν με μισή συχνότητα $\boldsymbol{0.5f}$ τότε με τα περισσότερα καλώδια η χωρητικότητα γίνεται λίγο παραπάνω από διπλή $\boldsymbol{2.2C}$ και η τάση επειδή είναι κάπως γραμμική με την συχνότητα γίνεται $\boldsymbol{0.6V}$, η νέα κατανάλωση για το διπύρηνο κύκλωμα θα είναι $\boldsymbol{Power_2 = 0.396CV^2f}$, μόλις **40%** της αρχικής κατανάλωσης. Τα δύο παραπάνω κυκλώματα για την ίδια είσοδο μας δίνουν την ίδια έξοδο, αλλά αυτό με τους δύο πυρήνες καταναλώνει **60%** λιγότερη ενέργεια και για αυτό ο πολυπρογραμματισμός είναι πολύ σημαντικός καθώς δεν γίνεται να βασιστούμε στους επεξεργαστές να γίνουν πιο γρήγοροι σειριακά γιατί ή κατανάλωση ανεβαίνει με τέτοιο ρυθμό που δεν είναι εφικτό και χτυπάμε έναν τοίχο ενέργειας(**power wall)**.

(6.6 Το “hello world” του Παράλληλου κώδικα)

(6.6.1 Σειριακό π)

```
#include <stdio.h>
static long num_of_steps = 1000000000;
double step;

int main()
{
    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_of_steps;

    for(int i = 0; i < num_of_steps; i++)
    {
        x = (i+0.5) * step;
        sum = sum + 4.0/(1+x*x);
    }
    pi = step * sum;
    printf("PI=%lf\n", pi);

}
```

Εικόνα 3 Σειριακό π

Ένα καλό πρόγραμμα που μπορούμε να χρησιμοποιήσουμε για την εξήγηση του παραλληλισμού είναι ο υπολογισμός του **π** γιατί μπορούμε εύκολα να φτιάξουμε μια παράλληλη έκδοση του. Για να υπολογίσουμε το π πρέπει να λύσουμε το ακόλουθο ολοκλήρωμα $\int_0^1 \frac{4.0}{1+x^2} dx$ το οποίο υπολογίζεται όπως φαίνεται στην εικόνα 3. Ο αριθμός των βημάτων πρέπει να είναι μεγάλος για να βρούμε σωστά το π και αν τα μεγαλώσουμε περισσότερο μας βοηθάει γιατί μεγαλώνει ο χρόνος εκτέλεσης του προγράμματος, κάτι που μας βοηθάει να δείξουμε τον παραλληλισμό. Ο κώδικας της εικόνας 3 χρειάζεται **1.4191sec** για να μας δώσει το π και είναι σημαντικό αυτό για τον παραλληλισμό γιατί ο κώδικας που θέλουμε να επιταχύνουμε πρέπει να είναι αρκετά μεγάλο φορτίο και να παίρνει σημαντικό χρόνο στον χρόνο εκτέλεσης. Αν ο κώδικας μας είναι κάτι που τρέχει είδη πολύ γρήγορα π.χ. σε μs ή ms δεν έχει νόημα να προσπαθήσουμε να το παραλληλοποιήσουμε γιατί το overhead που έρχεται από την δημιουργία των **threads** θα κάνει την εκτέλεση πιο αργή.

Αρχικά για να φτιάξουμε μια παράλληλη εκδοχή του προγράμματος της εικόνας 3 πρέπει να εντοπίσουμε το σημείο στον κώδικα που παίρνει την περισσότερη ώρα να εκτελεστεί καθώς και ο συνολικός χρόνος εκτέλεσης βασίζεται σε αυτό. Για να εντοπιστεί το σημείο μπορούμε να χρησιμοποιήσουμε προγράμματα μελέτης εκτέλεσης(profiling programs) όπως το callgrind(25), το οποίο θα μας πει που πέρασε τον περισσότερο χρόνο το πρόγραμμα κατά την εκτέλεση, εδώ όμως μπορούμε εύκολα να δούμε που είναι γιατί μπορεί να είναι μόνο το **for loop** που κάνει πολλές πράξεις και παίρνει χρόνο να τελειώσει και συνήθως τα σημεία που μπορούμε και ψάχνουμε να κάνουμε πιο γρήγορα είναι οι επαναλήψεις.

Αφού βρήκαμε το σημείο πρέπει να μοιράσουμε το φορτίο ανάμεσα στα νήματα ώστε κάθε νήμα να κάνει ένα ποσοστό της συνολικής δουλειάς. Μέσα στο for loop έχουμε **`x=(i+0.5)*step;`** το οποίο βασίζεται στον δείκτη **i** και ένα άθροισμα **`sum=sum+4.0/(1+x*x);`** το sum μας βοηθάει που χρησιμοποιείτε γιατί μετά για να πάρουμε το τελικό αποτέλεσμα μπορεί να σπάσει σε μικρότερα sums που μπορούμε να δώσουμε σε νήματα ώστε να υπολογιστούν παράλληλα και να επιταχύνουμε τον κώδικα, είναι σαν να μοιράζουμε χαρτιά τράπουλας με το κάθε νήμα να παίρνει μία δική του επανάληψη.

(6.6.2 Παράλληλο π)

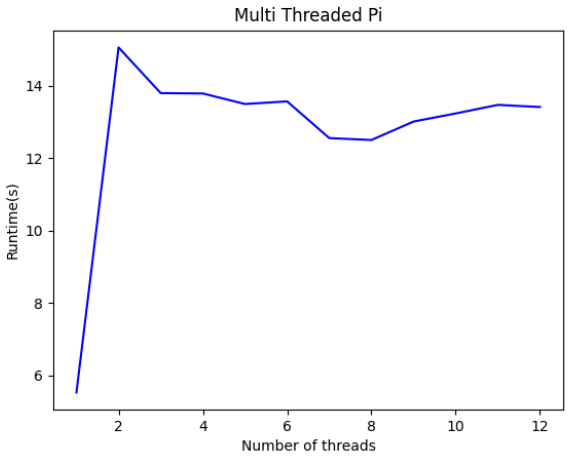
```
static long num_of_steps = 1000000000;
double step;

int main()
{
    int i, nthreads;
    double pi, sum[NUM_THREADS];
    step = 1.0 / (double)num_of_steps;
    double start_time, run_time;
    omp_set_num_threads(NUM_THREADS);
    start_time = omp_get_wtime();
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if(id == 0)
        {
            nthreads = nthrds;
        }
        for(i = id, sum[id] = 0.0; i < num_of_steps; i = i + nthrds)
        {
            x = (i+0.5) * step;
            sum[id] = sum[id] + 4.0/(1+x*x);
        }
    }
    for(i = 0, pi = 0.0; i < nthreads; i++)
    {
        pi += sum[i] * step;
    }
    run_time = omp_get_wtime();
    printf("PI=%lf in %.4fs\n", pi, (run_time-start_time));
    }
```

Εικόνα 4 Πρώτη εκδοχή του παράλληλου π

Στην εικόνα 4 φαίνεται η λύση και υπάρχουν λεπτομέρειες που χρειάζονται περεταίρω εξήγηση:

- Για να χρησιμοποιήσουμε την OpenMP πρέπει συμπεριλάβουμε το αρχείο **κεφαλίδας(header file)** `omp.h`, η συγκεκριμένη βιβλιοθήκη συνήθως έρχεται μαζί με τους περισσότερους compilers αλλά αν δεν την έχουμε πρέπει να την εγκαταστήσουμε και αυτό γίνεται διαφορετικά ανάλογα με το τι λειτουργικό σύστημα έχουμε για παράδειγμα όμως στα **Ubuntu** γίνεται με την εντολή "**sudo apt install libomp-dev"** και στα windows γίνεται με το πρόγραμμα **Mingw**(48)**.** Τέλος πρέπει να διευκρινίσουμε στον compiler χρησιμοποιούμε την OpenMP με την επιλογή
  -fopenmp.
- Δηλώνουμε `NUM_THREADS` τον αριθμό των νημάτων που θέλουμε να δημιουργήσουμε και καλούμε την συνάρτηση `omp_set_num_threads(NUM_THREADS)` για πούμε στην OpenMP πόσα νήματα να δημιουργήσει.
- Οι μεταβλητές **start_time** και **run_time** είναι εδώ για να αξιολογήσουμε την απόδοση του κώδικα και να δούμε αν υπάρχουν βελτιώσεις στον χρόνο εκτέλεσης. Καλούν την συνάρτηση `omp_get_wtime()` στην αρχή και στο τέλος των πράξεων και αφαιρώντας το τέλος από την αρχή έχουμε τον χρόνο που πήρε ο κώδικας να εκτελεστεί. Μετράμε τον χρόνο μόνο στο σημείο όπου το πρόγραμμα κάνει πράξεις γιατί είναι χρονοβόρο και είναι το ουσιαστικό κομμάτι του κώδικα που κάνει δουλειά ενώ οι δηλώσεις μεταβλητών, οι δηλώσεις αρχείων κεφαλίδας και οι εκτυπώσεις στο τερματικό δεν μας ενδιαφέρουν για αξιολόγηση.
- Χρησιμοποιούμε την οδηγία `#pragma omp parallel` για να δηλώσουμε την περιοχή όπου θέλουμε να δημιουργήσουμε νήματα και να παραλληλοποιήσουμε κώδικα. Σημαντική πληροφορία είναι το γεγονός ότι τα νήματα έχουν δική τους τοπική μνήμη και όταν κάθε νήμα τελειώσει την δουλειά του ότι πληροφορία έχει στις δικές του μεταβλητές θα χαθεί όταν γυρίσει τον έλεγχο στο κύριο νήμα. Επειδή όμως θέλουμε να αποθηκεύσουμε την δουλειά που κάνει κάθε νήμα για να την ενώσουμε και να πάρουμε το αποτέλεσμα μας στο τέλος χρειαζόμαστε μοιραζόμενη μνήμη και για να γίνει αυτό το **sum** είναι δηλωμένο έξω από το παράλληλο κομμάτι ώστε να το βλέπουν όλα τα νήματα και ενώ ήταν μεταβλητή τώρα είναι πίνακας με στοιχεία ίσα τον αριθμό νημάτων που έχουμε για να έχει κάθε νήμα δικό του μερικό **sum**. Αφού τα νήματα τελειώσουν και έχουμε όλα τα μερικά sum θα τα προσθέσουμε και θα πάρουμε το ολικό sum που θέλουμε για να βρούμε το π.
- Μέσα στο παράλληλο κομμάτι όμως κάθε νήμα πρέπει να έχει και δικές του τοπικές μεταβλητές για να μπορεί να κάνει την δουλειά του. Καλώντας το `omp_get_thread_num()` παίρνουμε τον αριθμό του νήματος που το καλεί, αν π.χ. αφού ξεκινήσουμε το παράλληλο σημείο και δημιουργηθούν και μπουν τα νήματα, το νήμα 1 θα καλέσει το `omp_get_thread_num()` και θα γυρίσει πίσω τον αριθμό του νήματος που σε αυτήν την περίπτωση είναι 1 και πάει στην μεταβλητή **id.** Χρειαζόμαστε το id για να διαιρέσουμε το for loop πιο μετά.
- Έξω από το παράλληλο σημείο έχουμε μια καθολική μεταβλητή **`nthreads`** και μέσα έχουμε το **`nthrds`** που είναι τοπική και καλούμε `nthrds = omp_get_num_threads();` για να δούμε πόσα νήματα έχουν φτιαχτεί, μετά παίρναμε τον αριθμό νημάτων στο καθολικό nthreads γιατί γίνεται αυτό? Εμείς μπορεί να ζητήσαμε όταν δηλώσαμε τον αριθμό νημάτων πιο πάνω με το `omp_set_num_threads(NUM_THREADS);` το σύστημα όμως μπορεί να μας δώσει λιγότερα νήματα από αυτά που ζητήσαμε και στο τέλος που προσθέτουμε τα sums πρέπει να ξέρουμε πόσα νήματα δημιουργήθηκαν.
- Μέσα στο for loop φαίνεται η διανομή των νημάτων. Ο δείκτης **i** ξεκινάει από τον αριθμό του νήματος **id** και μεγαλώνει με τον συνολικό αριθμό νημάτων, το νήμα 0 για παράδειγμα θα έχει της επαναλήψεις 0, 2, 4, 6, 8, … ενώ το νήμα 1 θα έχει τις επαναλήψεις 1, 3, 5, 7, 9, ... Το **id** χρησιμοποιείται ξανά ως δείκτης με το sum ώστε κάθε νήμα να έχει το δικό του sum.

(6.6.3 False Sharing)



Εικόνα 5 Σύγκριση χρόνου με αριθμό νημάτων, χωρίς βελτιστοποιήσεις μεταγλωττιστή για να φανεί το πρόβλημα του false sharing.

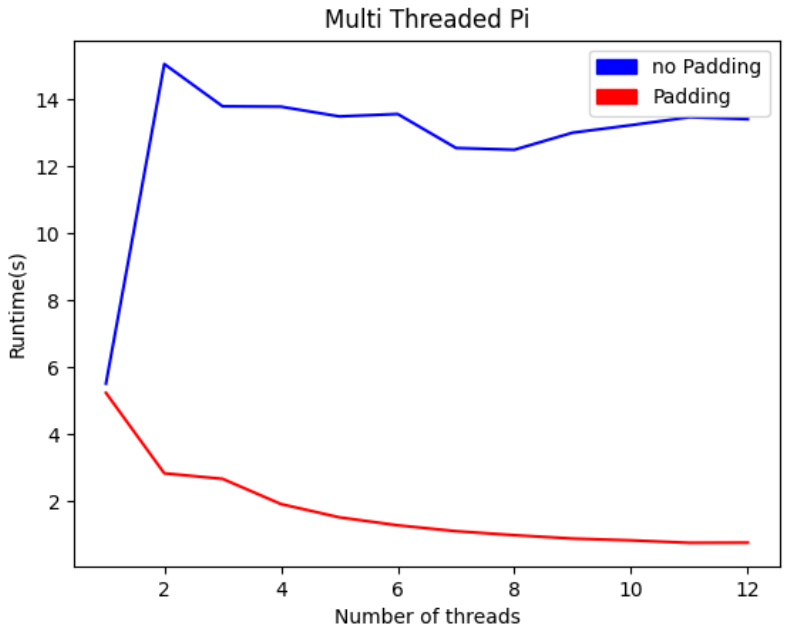
Στην εικόνα 5 το μπλε γράφημα δείχνει τα αποτελέσματα του κώδικα που συζητήθηκε πιο πάνω, αρχικά με ένα νήμα να τρέχει στα **5.52sec** και με την πρόσθεση νημάτων να χειροτερεύει τον χρόνο με τυχαίο τρόπο. Περιμέναμε ο χρόνος εκτέλεσης να βελτιωθεί αλλά έγινε πολύ χειρότερος πιο είναι το πρόβλημα? Το πρόβλημα που αντιμετωπίζουμε λέγεται **ψευδής κοινή χρήση(false sharing)** και πρέπει να το επιλύσουμε για να πάρουμε όλες τις επιδόσεις που μπορεί να μας προσφέρει ο επεξεργαστής μας.

Όπως υποδεικνύεται και από το όνομα false sharing σημαίνει ότι δεν υπάρχει πραγματική διανομή μνήμης ανάμεσα στους πυρήνες. Κάθε thread χρησιμοποιεί ένα μέρος του πίνακα sum για να βρει το μερικό άθροισμα, ο sum όμως θα είναι αποθηκευμένος στην μνήμη cache και εδώ είναι το πρόβλημα που δημιουργείτε. Η μνήμη cache είναι χωρισμένη σε γραμμές που ονομάζονται **cache lines** και οι γραμμές δεν είναι πολύ μεγάλες συνήθως 64 bytes ή 128 bytes για αυτό ο sum χωράει μέσα σε μία γραμμή και αυτό είναι πρόβλημα επειδή αυτός κάθεται στην ίδια γραμμή και αν έρθει ένας πυρήνας και αποκτήσει πρόσβαση σε ένα στοιχείο του sum και ένας άλλος πυρήνας προσπαθήσει να αποκτήσει και αυτός πρόσβαση δεν θα μπορεί και θα πρέπει να περιμένει. Τα νήματα δηλαδή θα παλεύουν για πρόσβαση στην ίδια γραμμή cache δημιουργώντας **αστοχίες cache(cache misses)** και μειώνοντας τις επιδόσεις μας.

```
#define PAD 8
int main()
{
    int i, nthreads;
    double pi, sum[NUM_THREADS][PAD];
    #pragma omp parallel
    {
        for(i = id, sum[id][0] = 0.0; i < num_of_steps; i += nthrds)
        {
            x = (i+0.5) * step;
            sum[id][0] = sum[id][0] + 4.0/(1+x*x);
        }
    }
    for(i = 0, pi = 0.0; i < nthreads; i++)
    {
        pi += sum[i][0] * step;
    }
   }
```

Εικόνα 6 Παράλληλο π που λύνει το false sharing με την χρήση array padding

Για να λυθεί το πρόβλημα του false sharing θα πρέπει να εγγυηθούμε ότι κάθε στοιχείο του sum θα βρίσκεται σε διαφορετικό cache line και για να γίνει αυτό μια από τις λύσεις που μπορούμε να υλοποιήσουμε είναι να γίνει **pad** ή αλλιώς «φούσκωμα» του πίνακα. Στους σύγχρονους επεξεργαστές το μέγεθος των γραμμών cache είναι συνήθως 64 bytes αλλά μπορεί να διαφέρει ανάλογα με τον επεξεργαστή, στην περίπτωση του επεξεργαστή που έτρεξε ο κώδικας ελέγχθηκε το μέγεθος της γραμμής cache και είναι 64 bytes. Ένας ακέραιος είναι 4 bytes το οποίο σημαίνει ότι το pad θα πρέπει να είναι $4 * x = 64\ bytes => x = 8\ bytes$, στην συνέχεια για να γίνει pad του sum θα αλλαχτεί σε δισδιάστατο και η δεύτερη διάσταση θα έχει το μέγεθος της γραμμή cache δηλαδή 8 θέσεις στον πίνακα και έτσι σιγουρεύουμε ότι κάθε μερικό sum βρίσκεται σε ξεχωριστό cache line και δεν έχουμε false sharing.



Εικόνα 7 Παράλληλο π, σύγκριση εκδοχής χωρίς padding και με padding

Στην εικόνα 7[4] φαίνεται η διαφορά του array padding με κάθε νέο νήμα που δίνουμε στο πρόγραμμα να βελτιώνει την ταχύτητα, με ένα νήμα να χρειάζεται **5.24sec** να ολοκληρωθεί και με 12 νήματα να χρειάζεται μόνο **0.77sec, 6.8** φορές πιο γρήγορο και για αυτό ο παραλληλισμός είναι σημαντικός και ένα δυνατό εργαλείο στα χέρια κάθε προγραμματιστή.

[4] Η δημιουργία του γραφήματος της εικόνας 7 αλλά και των όλων των υπόλοιπων στην πτυχιακή έγιναν με την χρήση του εργαλείου matplotlib(50)

(6.6.4 Φορητότητα)

Το padding όντως είναι μια λύση του false sharing αλλά για να δουλέψει έπρεπε πρώτα να γνωρίζουμε το μέγεθος της cache line για τον συγκεκριμένο επεξεργαστή κάτι που δεν είναι ίδιο σε όλους τους επεξεργαστές. Η συγκεκριμένη λύση δηλαδή δεν είναι φορητή και δεν προτιμάτε αλλά χρειάζεται να την γνωρίζει ο προγραμματιστής για την κατανόηση της μνήμης και το πως επιδράει με τα νήματα. Είναι προτιμότερο να γραφτεί το πρόγραμμα με τέτοιο τρόπο ώστε να είναι φορητό και το μόνο που χρειάζεται είναι μια μικρή αλλαγή στον κώδικα και μια νέα OpenMP οδηγία.

```
int nthreads;
double pi = 0.0;
step = 1.0 / (double)num_of_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i, id, nthrds;
    double x, sum;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if(id == 0)
    {
        nthreads = nthrds;
    }
    for(i = id, sum = 0.0; i < num_of_steps; i = i + nthrds)
    {
        x = (i+0.5) * step;
        sum += 4.0/(1+x*x);
    }
    #pragma omp critical
        pi += sum * step;
}
```

Εικόνα 8 Παράλληλο π με την χρήση κρίσιμου σημείου για συγχρονισμό νημάτων

Οι αλλαγές που χρειάζονται να γίνουν φαίνονται στην εικόνα 8 και είναι οι εξής:

- Ο πίνακας sum δεν υπάρχει πλέον, τώρα το κάθε νήμα έχει δικό του αντίγραφο της μεταβλητής sum για βρει το μερικό άθροισμα.
- Όταν φτάσει στο τέλος της οδηγίας `#pragma omp parallel` ο κώδικας όλα τα νήματα φεύγουν με εξαίρεση το κυρίαρχο νήμα, κάτι που σημαίνει ότι και όλα τα αθροίσματα που είχαν τα νήματα τοπικά για τον εαυτό τους θα χαθούν, για αυτό πρέπει κάπως να τα αθροίσουμε πριν φύγουν τα νήματα.
- Για να αθροιστούν τα sum δημιουργούμε την μεταβλητή pi έξω από το παράλληλο τμήμα και αθροίζουμε σε αυτήν τα μερικά αθροίσματα, ένα πρόβλημα που μπορεί να δημιουργηθεί σε αυτήν την περίπτωση επειδή όλα τα νήματα θα τελειώσουν την δουλειά τους και θα φτάσουν στο τέλος όπου θα πρέπει να προσπελάσουν το pi ένα νήμα μπορεί να προσπαθεί να διαβάσει ενώ ένα άλλο γράφει ή και το ανάποδο, αυτό ονομάζεται **συνθήκη συναγωνισμού(race condition)** και χρειάζεται προσοχή.
- Σε σημεία που δημιουργούνται race conditions πρέπει να σιγουρευτούμε ότι κάθε νήμα θα έχει τελειώσει πριν μπορέσει να έρθει άλλο νήμα και γράψει στο pi, αυτό ονομάζεται **αμοιβαίος αποκλεισμός(mutual exclusion)** και μπορεί να επιτευχθεί χρησιμοποιώντας τα

omp οδηγήματα `#pragma omp critical` και `#pragma omp atomic` . Με αυτά τα οδηγήματα διευκρινίζουμε στην OpenMP ότι η πράξη `pi += sum * step;` πρέπει να έχει mutual exclusion.

- Αφού μπορούμε να χρησιμοποιήσουμε το `#pragma omp critical` για να συγχρονίσουμε τα νήματα στο τέλος δεν γίνεται να μην χρησιμοποιήσουμε το sum και να προσθέσουμε απευθείας στο pi αποφεύγοντας τα race conditions? Δεν γίνεται γιατί τα εργαλεία συγχρονισμού έχουν μεγάλο κόστος και η πράξη που κάνουμε για να βρούμε το `x` δεν παίρνει σημαντικό χρόνο άρα το αποτέλεσμα θα είναι τα νήματα να τελειώνουν τον υπολογισμό του `x` και θα περιμένουν για την σειρά τους. Το αποτέλεσμα θα είναι να χειροτερέψουμε πολύ την ταχύτητα μας, για αυτό χρειάζεται προσοχή που και πως χρησιμοποιούνται τα εργαλεία συγχρονισμού.
- Με τον κώδικα της εικόνα 8 λύνουμε το πρόβλημα του false sharing διατηρώντας το πρόγραμμα μας φορητό χωρίς να χάνουμε ταχύτητα.

(6.6.5 Απλοποίηση)

Περιπτώσεις κώδικα όπως της εικόνας 8 θα βρίσκουμε συνέχεια μπροστά μας, δηλαδή να έχουμε ένα μεγάλο for loop με πολλές επαναλήψεις που να παίρνει αρκετό χρόνο να τελειώσει και να θέλουμε να σπάσουμε τις επαναλήψεις σε κομμάτια, να τα δώσουμε σε νήματα ώστε να τα δουλέψουνε παράλληλα και στο τέλος να προσθέσουμε το αποτέλεσμα. Επειδή είναι τόσο συχνά αυτά τα **`for`** loops με την OpenMP υπάρχει τρόπος να απλοποιήσουμε τον κώδικα χωρίς να γράφουμε κάθε φορά το παράδειγμα τις εικόνας 8. Για να απλοποιηθεί μπορεί να γίνει η χρήση του `#pragma omp for`. Η συγκεκριμένη εντολή μπαίνει πριν από έναν βρόγχο επανάληψης και κάνει ότι σημειώθηκε παραπάνω μόνο του.

```
#pragma omp parallel private(x, sum)
{
    #pragma omp for
    for(int i = 0; i < num_of_steps; i++)
    {
        x = (i+0.5) * step;
        sum += 4.0/(1+x*x);
    }
    #pragma omp critical
    pi += sum * step;
}
```

Εικόνα 9 Απλοποιημένο παράλληλο π με την χρήση του pragma omp for

Πάλι πρέπει να υπάρχει το `#pragma omp parallel` αλλιώς δεν δημιουργούνται νήματα και πρέπει να δηλώσουμε τις μεταβλητές `x` και `sum` ως ιδιωτικές για να έχει κάθε νήμα δικό του αντίγραφο. Το `#pragma omp for` θα χωρίσει τον βρόγχο που βρίσκεται από κάτω του και θα μοιράσει τις επαναλήψεις στα νήματα αλλά στο τέλος δεν θα συνδυάσει τα αποτελέσματα στην καθολική μεταβλητή, αυτό γίνεται με το `reduction` γίνεται. Επίσης γίνεται να ενωθεί το `for` με το `parallel` και να γίνει `#pragma omp parallel for` σε περιπτώσεις που υπάρχει μόνο ένας βρόγχος για παραλληλοποίηση.

(6.6.6 Reduction)

Υπάρχουν περιπτώσεις που ο βρόγχος επανάληψης που θέλουμε να παραλληλοποιήσουμε έχει πράξεις μέσα που βασίζονται στο αποτέλεσμα των προηγούμενων επαναλήψεων όπως στην περίπτωση του προγράμματος π. Αυτές οι περιπτώσεις ονομάζονται **εξαρτήσεις βρόγχου** (loop carried dependence) και χρειάζεται να τροποποιήσουμε τον κώδικα μας πριν προσπαθήσουμε την παραλληλοποίηση και να αφαιρέσουμε τις εξαρτήσεις βρόγχου αλλιώς το αποτέλεσμα που θα πάρουμε θα είναι λάθος.

```
int j = 10;
for(int i = 0; i < N; i++)
{
        j += 2; //loop carried dependency
        a[i] = b[j] + 2;
}
```

Εικόνα 10 Παράδειγμα εξάρτησης βρόγχου

Το αποτέλεσμα της εικόνα 10 θα είναι λάθος γιατί τα νήματα δεν εκτελούνται με την ίδια σειρά, το νήμα 1 για παράδειγμα μπορεί να είναι στην επανάληψη 5 και να περιμένει το `j` να είναι έχει την τιμή 20 αλλά το νήμα 2 μπορεί να είναι στην επανάληψη 8 `j` να είναι έχει την τιμή 36 και έτσι το αποτέλεσμα μας βασίζεται στην προηγούμενη επανάληψη για τον υπολογισμό του και πρέπει να τον τροποποιηθεί ώστε η πράξη να μπορεί να γίνει εκτός σειράς .

```
for(int i = 0; i < N; i++)
{
        //loop carried dependency solved
        int j = 10 + 2*(i+1)
        a[i] = b[j] + 2;
}
```

Εικόνα 11 Παράδειγμα επίλυσης εξάρτησης βρόγχου

Για να λυθεί η εξάρτηση βρόγχου πρέπει το `j` να υπολογίζεται από το νήμα που το χρειάζεται σε οποιαδήποτε επανάληψη και να είναι. Η λύση φαίνεται στην εικόνα 11, το `j` είναι τοπική μεταβλητή σε κάθε νήμα και αντί να προστίθεται κάθε φορά το 2 σε αυτό τώρα υπολογίζεται από την επανάληψη που βρισκόμαστε.

Παρόμοια το σειριακό π είναι πρόβλημα εξάρτησης βρόγχου όπου προσθέτουμε στο sum και βασιζόμαστε στα αποτελέσματα των προηγουμένων αποτελεσμάτων. Προηγουμένως λύσαμε το πρόβλημα δίνοντας σε κάθε νήμα το δικό του sum, επειδή όμως αυτά τα προβλήματα είναι συνηθής η OpenMP μας δίνει δικό της εργαλείο για να τα αντιμετωπίσουμε και το εργαλείο αυτό είναι η φράση `reduction` .

```
double sum, x, pi;
#pragma omp parallel for reduction(+:sum, x) schedule(static)
for(int i = 0; i < num_of_steps; i++)
{
    x = (i+0.5) * step;
    sum += 4.0/(1+x*x);
}
pi += sum * step;
```

Εικόνα 11 Παράλληλο π με χρήση reduction

Αντί να δημιουργήσουμε εμείς το sum τοπικά και να το προσθέσουμε στο pi με mutual exclusion στο τέλος `#pragma omp critical pi += sum * step;` μπορούμε να χρησιμοποιήσουμε το `reduction`. Τα reductions έχουν την μορφή `reduction(op:list)` όπου `op` ο τύπος πράξης που θέλουμε να κάνουμε π.χ.+,-,/,* και `list` η λίστα μεταβλητών που θέλουμε να είναι τοπικά στο παράλληλο τμήμα.

Το `reduction` θα κάνει το εξής, θα δημιουργήσει τοπικές μεταβλητές για τις μεταβλητές που του προσδιορίσαμε(σε αυτήν την περίπτωση το `sum` και το `x`) και θα τις αρχικοποιήσει. Αφού τελειώσει το παράλληλο τμήμα θα προσθέσει τα αποτελέσματα σε μια μεταβλητή και θα την επιστρέψει στην καθολική μεταβλητή.

(6.6.7 Schedule)

Στην εικόνα 11 υπάρχει μια ακόμα φράση που δεν εξηγήθηκε και αυτή είναι η `schedule`. Η `schedule` είναι ένας τρόπος να επικοινωνήσουμε στον μεταγλωττιστή τον τρόπο με τον οποίο να σπάσει τον βρόγχο. Η μορφή που έχει είναι `schedule(type, chunk_size)` με `type` τον τύπο διαμοιρασμού των επαναλήψεων και `chunk_size` το μέγεθος των επαναλήψεων που δίνει σε κάθε νήμα. Υπάρχουν οι εξής τύποι:

- `schedule(static, chunk_size)`
  Δίνει σε κάθε νήμα αριθμό επαναλήψεων ίσο με το `chunk_size` αν δεν προσδιορίσουμε αριθμό δίνει σε κάθε νήμα αριθμό `NUM_THREADS/NUM_ITERATIONS`
- `schedule(dynamic, chunk_size)`
  Κάθε νήμα παίρνει από μια ουρά αριθμό επαναλήψεων ίσο με το `chunk_size` μέχρι να τελειώσει ο αριθμός επαναλήψεων, χρήσιμο όταν στην εκτέλεση δεν γνωρίζουμε κάθε νήμα πόση ώρα θα κάνει και κάθε νήμα παίρνει διαφορετικό χρόνο να τελειώσει
- `schedule(guided, chunk_size)`
  Τα νήματα ξεκινούν αρχικά με μεγάλα blocks από επαναλήψεις και όσο φτάνουν στο τέλος παίρνουν όλο και μικρότερα, δεν χρησιμοποιείτε συνήθως
- `schedule(runtime)`
  Βλέπει στην στιγμή της εκτέλεσης από την περιβαλλοντική μεταβλητή τον τύπο, χρησιμοποιείται για δοκιμές χωρίς να χρειαστεί να ξανά μεταγλωττιστεί ο κώδικας
- `schedule(auto)`
  Δίνει τον έλεγχο στον μεταγλωττιστή να κάνει ότι νομίζει ότι είναι καλύτερο

Προτείνεται στον/στην προγραμματιστή/στρια να πειραματιστεί με τις διάφορες επιλογές τύπων αλλά και μεγεθών `chunk_size` για να πάρει το καλύτερο αποτέλεσμα.

(6.7 Παράλληλος κώδικας στην Combic)

(6.7.1 Επιλογή συναρτήσεων)

Για να βρεθούν υποψήφιες μέθοδοι για μετατροπή σε παράλληλο κώδικα ακολούθησαν τα εξής βήματα:

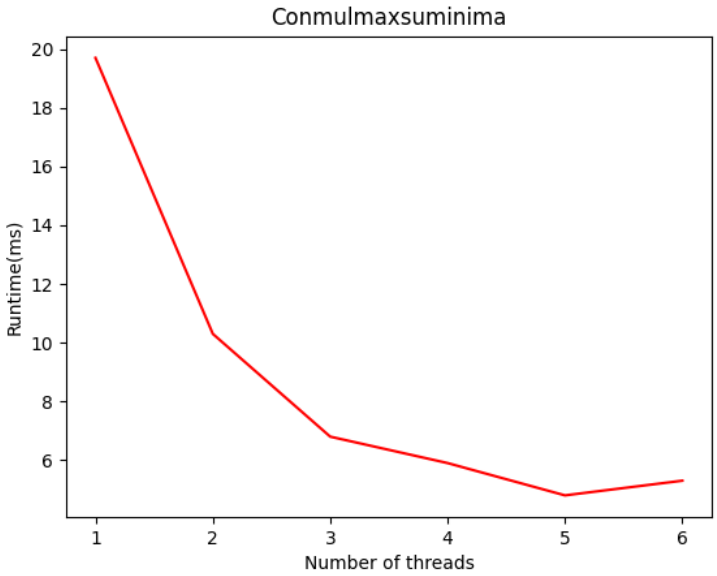
- Αρχικά αναλύθηκε ο χρόνος εκτέλεσης όλων των μεθόδων της βιβλιοθήκης με τις μέγιστες εισόδους όπου γινόταν, μέθοδοι οι οποίες χρειάστηκαν μόνο ms για να ολοκληρωθούν αποκλείστηκαν καθώς η χρήση νημάτων θα έκανε την εκτέλεση πιο αργή.
- Μετά από αυτή την διαδικασία έμειναν μόνο μέθοδοι οι οποίες δεν είχαν όριο στο μέγεθος εισόδου τους και η μέθοδος qm.c. Οι μέθοδοι εκτός του qm.c στην είσοδο επιλέχτηκαν γιατί χρησιμοποιούν strings χωρίς όριο στο μήκος τους για είσοδο και .το qm.c επιλέχτηκε γιατί για πολλούς ελαχιστόρους η διαδικασία ένωσης ελαχιστόρων που γίνεται μπορεί να πάρει σημαντικό χρόνο εκτέλεσης από λεπτά μέχρι ώρες.

Η πρώτη προσπάθεια έγινε στην μέθοδο conmulmaxsuminima η οποία μετατρέπει εκφράσεις από multiplication of magistrates σε sum of minima και αφού κάνει την μετατροπή ταξινομεί τα ξεχωριστά strings που αποτελούν την έκφραση. Αν η είσοδος είναι αρκετά μεγάλη η ταξινόμηση μπορεί να γίνει παράλληλα και να δούμε διαφορά στον χρόνο εκτέλεσης.

```
start_time = omp_get_wtime();
#pragma omp parallel
{

    int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if(id == 0) nthreads = nthrds;
    for(i = id; i < length; i = i + nthrds){
        sort(list[i]);
    }

}
run_time = omp_get_wtime();
```

Εικόνα 12 Παράλληλο sorting στην conmulmaxsuminima

Το αποτέλεσμα φένετε στην εικόνα 12 όπου γίνεται παράλληλα η ταξινόμηση σε κάθε string της λίστας. Παρακάτω στην εικόνα 13 φένετε και η επιτάχυνση που γίνεται όταν προσθέτουμε νήματα:



Εικόνα 13 Thread scaling στην conmulmaxsuminima

Στην εικόνα 13 φένετε το πώς μειώνεται ο χρόνος εκτέλεσης όσο προσθέτουμε νήματα οπότε ο κώδικας είναι επιτυχία? Δυστυχώς όχι γιατί για να παραχθεί αυτό το αποτέλεσμα χρειάστηκε να χρησιμοποιηθεί ένα ειδικό string μήκους 400.000 το οποίο δεν είναι ρεαλιστική είσοδος για την συνάρτηση. Η αλήθεια είναι ότι αν ο πίνακας ή το struct object που έχουμε έχει μεγάλο μήκος θα πάρουμε πίσω καλύτερο χρόνο εκτέλεσης και δεν έχει νόημα να δημιουργήσουμε εμείς τις καταστάσεις που χρειάζονται για να τα καταφέρει. Για αυτό απορρίπτονται και οι υπόλοιπες συναρτήσεις με παρόμοια λογική.

Το ίδιο δεν ισχύει για την επόμενη και μόνη συνάρτηση που κατάφερε να γίνει παράλληλη την qm.c η οποία έχει δύο στάδια:

- Στο πρώτο στάδιο συγκεντρώνει σε ομάδες ελαχιστόρους ανάλογα με το αν διαφέρουν κατά ένα bit.
- Αφού έχει τελειώσει μια νέα ομάδα τότε σημειώνει ποιοι συνδυασμοί από ελαχιστόρους δεν μπορούν να συνδυαστούν άλλο και τους σημειώνει ως **prime implicants.**
- Όταν δεν μπορεί να δημιουργήσει άλλη ομάδα σταματάει και βλέπει σε έναν πίνακα όπου βρίσκονται όλοι οι prime implicants και οι ελαχιστόροι που καλύπτουν, ποιοι καλύπτουν μόνο ένα ελαχιστόρο οριζόντια ή κάθετα και τον αφαιρεί.
- Όσοι prime implicants αφαιρούνται από τον πίνακα λέγονται essentials και το άθροισμα τους είναι η τελική ελαχιστοποιημένη συνάρτηση.

Στην qm.c ανάλογα με το πόσους ελαχιστόρους έχει η συνάρτηση ως είσοδο η διαδικασίες της 1)δημιουργίας ομάδων, 2)σημείωση prime implicants, 3)εύρεση των essential prime implicants μπορούν να πάρουν αρκετό χρόνο για να ολοκληρωθούν, για παράδειγμα με ελαχιστόρους 0-8192 τα πρώτα δύο στάδια χρειάζονται **57 λεπτά**[5] για να τελειώσουν.

[5] Για τον υπολογισμό 0-2^n ελαχιστόρων το αποτέλεσμα είναι A+A' το οποίο όσο αφορά τον χρόνο εκτέλεσης του 3ου σταδίου(την εύρεση των essentials) είναι αμελητέος. Η επιτάχυνση που θα παρατηρηθεί αφορά τα πρώτα δύο στάδια.

(6.7.2 Quine McCluskey πρώτη προσπάθεια)

```
#pragma omp parallel for private(j, k, bit_pos, temp_binary) shared(cant_merge)
schedule(dynamic)
for (i = 0; i < local_num_groups - 1; i++) {
    for (j = i + 1; j < local_num_groups; j++) {
        bit_pos = bit_diff((*groups)[i]->binary, (*groups)[j]->binary);
        if (bit_pos != 0) {
            // Code that checks for duplicates
            if (!isDuplicate) {
                MintermSet *merged_set = createMintermSet();
                // Code that appends to merged_set
                #pragma omp critical
                {
                    (*new_groups)[(*num_new_groups)++] = merged_set;
                }
            }
         }
     }
}
```

Εικόνα 14 Παράλληλος κώδικας δημιουργίας ομάδων.

Για το πρώτο στάδιο η αλλαγή του κώδικα φένετε στην εικόνα 14. Ο κώδικας αποτελείτε από δύο βρόγχους με το εξωτερικό να έχει εύρος $0 - N$ και τον εσωτερικό να έχει εύρος $N - i - 1$ άρα η συνολική πολυπλοκότητα είναι $O(n^2)$ το οποίο εξηγεί και γιατί χρειάζεται τόσο χρόνο για να ολοκληρωθεί. Επειδή δεν έχουμε κάποιο τρόπο να βελτιώσουμε την πολυπλοκότητα του αλγορίθμου μπορούμε να δοκιμάσουμε αν θα βοηθήσει ο παραλληλισμός και αυτές είναι η αλλαγές που έγιναν για να επιτευκτεί αυτό:

- Με την χρήση του `#pragma omp parallel for` δημιουργούμε νήματα και τα δίνουμε μέρος των επαναλήψεων του εξωτερικού βρόγχου.
- Οι μετρητές **i** και **j** ορίζονται ως `private` για να έχει το κάθε νήμα δικό του αντίγραφο και να μην υπάρχουν data races, το ίδιο ισχύει και για τις μεταβλητές **bit_pos** και **temp_binary**.
- Η μεταβλητή **cant_merge** ορίζεται ως καθολική καθώς δεν μας ενδιαφέρει πιο η πόσα νήματα θα αλλάξουν την τιμή της.
- Το σημείο όπου γίνεται πρόσθεση του ενωμένου σετ στην ομάδα είναι κρίσιμο σημείο και πρέπει να συγχρονιστεί με το `#pragma omp critical` γιατί μόνο ένα νήμα μπορεί να έχει πρόσβαση εκείνη την στιγμή.
- Χρησιμοποιείται `schedule(dynamic)` γιατί με αυτό βρέθηκαν τα καλύτερα αποτελέσματα.

Επειδή όμως τα νήματα δεν εκτελούνται με την ίδια σειρά που θα εκτελούσε ένα νήμα δημιουργούνται πολλά διπλότυπα σετ στην νέα ομάδα που πρέπει να αφαιρεθούν πριν πάμε στην νέα ομάδα. Αυτό θα γίνει με ένα νέο στάδιο μετά την ένωση που θα ελέγχει κάθε σετ με τα υπόλοιπα σετ στην ομάδα και θα το αφαιρεί αν δεν είναι μοναδικό.

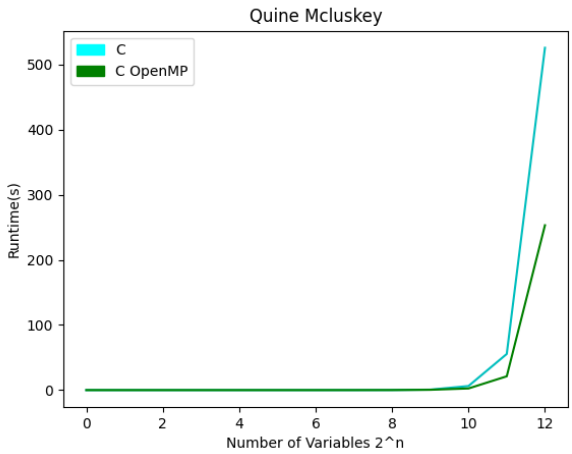
```
#pragma omp parallel for private(j) schedule(dynamic)
for (i = 0; i < num_groups; i++) {
    int isPrime = 1;

    for (j = 0; j < num_new_groups; j++) {
        if (bit_diff(groups[i]->binary, new_groups[j]->binary))
{
            isPrime = 0;
            break;
        }
    }
    groups[i]->isPrimeImplicant = isPrime;
   }
```

Εικόνα 15 Παράλληλος κώδικας εύρεσης των prime implicants

Για το δεύτερο στάδιο υπάρχει μεγαλύτερη πολυπλοκότητα καθώς ο εσωτερικός βρόγχος έχει εύρος $\mathbf{0 - N2}$ όπου $\mathbf{N2}$ είναι ο αριθμός των σετ της καινούργιας ομάδας η οποία έχει περισσότερα σετ, για αυτόν τον λόγο αξίζει να γίνει αυτό το στάδιο παράλληλο. Οι αλλαγές φαίνονται στην εικόνα 15 και θα γίνουν με τα εξής βήματα:

- Πάλι δημιουργούμε νήματα με το `#pragma omp parallel for` και μοιράζουμε τις επαναλήψεις του i.
- Η μεταβλητή **isPrime** δηλώνεται μέσα στο εξωτερικό βρόγχο και λειτουργεί σαν είναι **private(isPrime)** τοπικό σε κάθε νήμα.
- Το `schedule(dynamic)` πάλι έδωσε τα καλύτερα αποτελέσματα.

Με όλες αυτές τις αλλαγές όμως πόσο πιο γρήγορα εκτελείτε το qm.c σε σχέση με την αρχική εκδοχή?

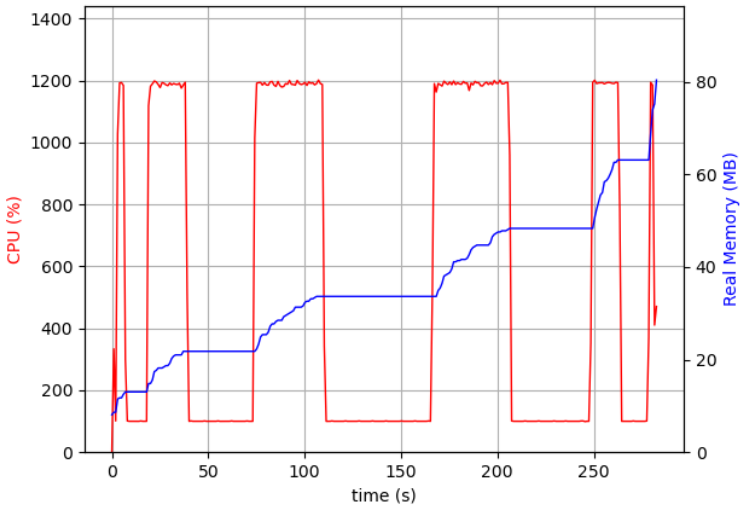


Εικόνα 16 Η μέθοδος qm σε 1 και 12 νήματα

Στην εικόνα 16 φαίνεται η διαφορά στον χρόνο εκτέλεσης και μπορούν να βγουν τα εξής συμπεράσματα:

- Για λίγες μεταβλητές μέχρι 9 το πρόγραμμα τελειώνει πολύ γρήγορα για να κάνουν οποιαδήποτε διαφορά περισσότερα νήματα και δεν θα αναλυθούν περαιτέρω.
- Από 9 μεταβλητές και μετά το πρόγραμμα που τρέχει παράλληλα είναι πιο γρήγορο και για 12 μεταβλητές παίρνει **253sec** ενώ με ένα νήμα χρειάζεται **525sec**, **2.07 φορές** πιο γρήγορο. Είναι όμως καλό το αποτέλεσμα όταν χρησιμοποιούμε 12 νήματα? Όχι δεν είναι και ο λόγος που το scaling μας είναι τόσο χαμηλό θα εξηγηθεί με τις επόμενες εικόνες.

(6.7.3 Χρήση όλου του επεξεργαστή)



Εικόνα 17 Χρήση CPU και μνήμης της qm σε CPU με 12 πυρήνες

Με της εικόνα 17[6] γίνεται αντιληπτό το πρόβλημα. Το γράφημα δείχνει την χρήση του επεξεργαστή και την μνήμης της qm.c όσο περνάει ο χρόνος και αυτό που μπορούμε να δούμε είναι ότι αφού δημιουργηθεί η νέα ομάδα(η δημιουργία φαίνεται στα σημεία που ο επεξεργαστής χρησιμοποιείτε πλήρως και δεσμεύει νέα μνήμη για τα νέα σετ) το πρόγραμμα περνάει περισσότερο χρόνο στην αφαίρεση αντίγραφων(φαίνεται και από την δέσμευση μνήμης που σταματάει γιατί κάνει ολίσθηση τα αντίγραφα στο τέλος της ομάδας). Καταφέραμε και κάναμε μέρος του προγράμματος σειριακό κάτι που δεν θέλουμε, αυτό που θέλουμε είναι να χρησιμοποιήσουμε όσο περισσότερο γίνεται όλους τους πυρήνες του επεξεργαστή.

```
for( i = 0; i < num_new_groups - 1; i++) {
        for( j = 0; j < num_new_groups - 1; j++) {
                duplicate_found = strcmp(new_groups[i]->binary,
                                         new_groups[j]->binary);
                if(!duplicate_found) {
                        // Shift elements
                        for(k = i; k < num_new_groups - 1; k++)
                                new_groups[k] = new_groups[k+1];
                        num_new_groups--;
                        i--;
                }
        }
}
```

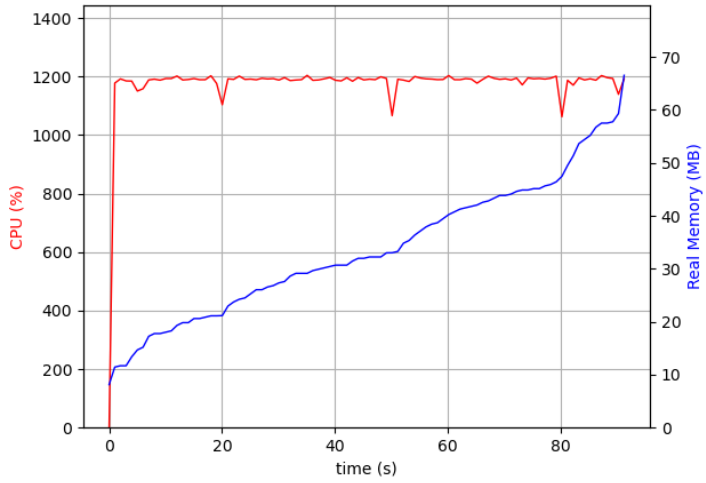
Εικόνα 18 Πρώτος τρόπος αφαίρεσης αντιγράφων από την ομάδα O(n^2)

[6] Η εικόνα 17 φτιάχτηκε με την εφαρμογή psrecord(49)

```
// Sort comparing the binaries so duplicates will be next to each other
qsort(new_groups, num_new_groups, sizeof(MintermSet *), compareBinary);

// Remove adjacent duplicates
for( i = 0; i < num_new_groups - 1; i++) {
    duplicate_found = strcmp(new_groups[i]->binary,
                             new_groups[i+1]->binary);
    if(!duplicate_found) {
        // Shift elements
        for(k = i; k < num_new_groups - 1; k++)
            new_groups[k] = new_groups[k+1];
        num_new_groups--;
        i--;
    }
}
```

Εικόνα 19 Δεύτερος τρόπος αφαίρεσης αντιγράφων με QuickSort O(nlogn)

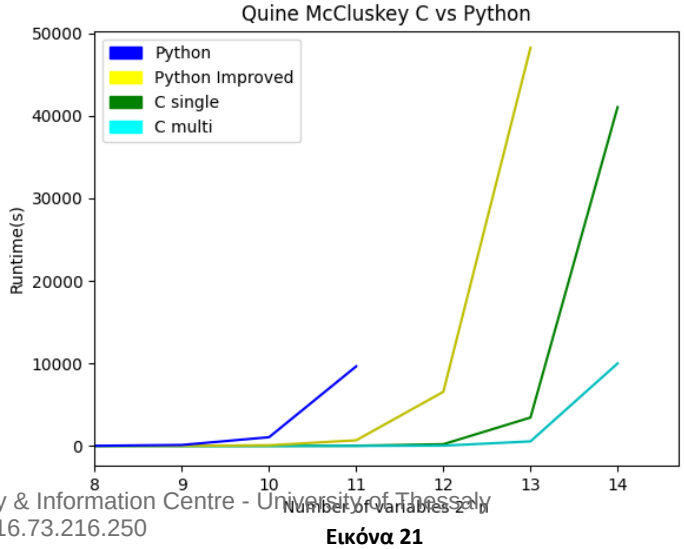
Στην εικόνα 18 είναι ο αρχικός τρόπος που πήρε περισσότερο χρόνο να τελειώσει από το παράλληλο τμήμα και είναι λογικό γιατί είναι **O(n^2)** ενώ στην εικόνα 19 είναι ο νέος τρόπος που κάνει πρώτα ταξινόμηση την ομάδα με **quicksort** και αφαιρεί γειτονικά αντίγραφα, ο νέος τρόπος είναι **O(nlogn).** Με την νέα προσθήκη το qm.c θα είναι πολύ πιο γρήγορο κάτι που θα φανεί στο επόμενο γράφημα.



Εικόνα 20 Βελτιωμένο qm.c

Το βελτιωμένο που δείχνει η εικόνα 20 είναι 3 φορές πιο γρήγορο(6 φορές σε σχέση με το 1 νήμα) και χρησιμοποιεί σχεδόν αποκλειστικά όλο το επεξεργαστή. Πως συγκρίνεται αυτή η νέα έκδοση με την μονονηματική και την Python?

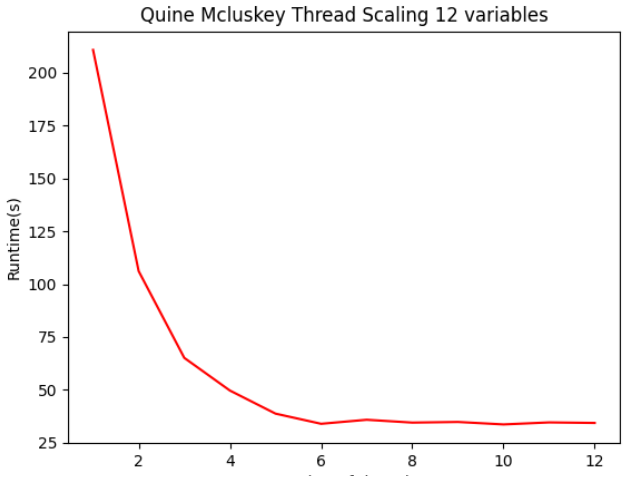
(6.7.4 Σύγκριση με Python)



Εικόνα 21

Η Python έχει δύο εκδόσεις του αλγορίθμου με την μία να είναι αρκετή αργή που ονομάζεται απλά Python στην εικόνα 21 και την άλλη που είναι σημαντικά καλύτερη και ονομάζεται Python Improved. Από την εικόνα 21 μπορούν να βγουν τα εξής συμπεράσματα:

- Η **αργή εκδοχή της Python** είναι πολύ αργή, καταφέρνει μόνο να φτάσει τις **11 μεταβλητές** μετά από **2.68 ώρες** ενώ η μονονηματική εκδοχή της C καταφέρνει **13 μεταβλητές** μετά από μόλις **57 λεπτά.**
- Η βελτιωμένη εκδοχή της Python(η αλλιώς **Python Improved**) είναι πολύ καλύτερη και καταφέρνει **13 μεταβλητές** μετά από **13.40 ώρες** (η αργή εκδοχή θα έπαιρνε μέρες για τόσες μεταβλητές).
- Η **μονονηματική εκδοχή της C** νικάει και τις 2 εκδοχές της Python καταφέρνοντας **14 μεταβλητές** μετά από **11.40 ώρες**.
- Η πιο γρήγορη από όλες τις εκδοχές όμως είναι η παράλληλη εκδοχή της C που καταφέρνει **14 μεταβλητές** σε **2.77 ώρες**, **4.1 φορές** πιο γρήγορη από την μονονηματική εκδοχή.



Εικόνα 22

Στην εικόνα 22 είναι το thread scaling της μεθόδου. Η αλλαγή από 1 νήμα σε 2 νήματα μας επιταχύνει 2 φόρες και είναι τέλειο scaling, μετά όμως έχουμε μικρότερες βελτιώσεις μέχρι τα 6 νήματα και μετά από τα 6 νήματα δεν υπάρχει καμία βελτίωση. Επειδή όμως 12 μεταβλητές δεν είναι πολλές, είναι πιθανό για μεγαλύτερες εισόδους να βοηθήσουν και περισσότερα από 6 νήματα.

ΚΕΦΑΛΑΙΟ 7 Συμπεράσματα και μελλοντικές Προεκτάσεις

Η βιβλιοθήκη που δημιουργήθηκε είναι ένα έτοιμο εργαλείο που μπορεί να χρησιμοποιηθεί εύκολα από οποιοδήποτε άτομο που κατέχει ελάχιστη γνώση με το τερματικό. Τα προβλήματα που επιλύει κυμαίνονται από εύκολα(όπως η μετατροπή ακεραίου σε δυαδικό και ο υπολογισμός λογικών πυλών) μέχρι δύσκολα(όπως η εύρεση BDD και η ελαχιστοποίηση με την μέθοδο Quine McCluskey) και είναι χρήσιμα για την επαλήθευση λύσεων τόσο σε μάθημα πανεπιστημίου όσο και στον σχεδιασμό κυκλωμάτων.

Η σύγκριση των δύο προγραμματιστικών γλωσσών C και Python δείχνει ότι εκτός από μερικές περιπτώσεις όπου μία προτιμάται από την άλλη, σε χρήση για γενικούς υπολογισμούς η ικανότητα του προγραμματιστή να υλοποιήσει μια λύση γρήγορα για κάποιο πρόβλημα και με σωστά αποτελέσματα είναι μεγαλύτερη από την ταχύτητα της ίδιας της γλώσσας(εκτός και αν η γλώσσα που χρησιμοποιείται έχει κάποια ιδιαιτερότητα και είναι εξαιρετικά αργή).

Η ανάπτυξη παράλληλου λογισμικού φανέρωσε το γεγονός ότι δεν γίνεται και δεν χρειάζεται να εκτελεστούν παράλληλα όλα τα προγράμματα. Αλλά για προγράμματα που η λύση είναι ο παραλληλισμός όπως στην περίπτωση της qm τα αποτελέσματα μπορεί να είναι πολύ πιο γρήγορα. Η ανάγκη για την ανάπτυξη παράλληλου κώδικα φαίνεται και στην τωρινή εποχή όπου η τάση είναι να γίνεται όλο και πιο αργή η ανάπτυξη της ταχύτητας του επεξεργαστή καθώς ο νόμος του Moore χάνει την ισχύ του και οι κατασκευαστές παρουσιάζουν όλο και περισσότερους πυρήνες μέσα στην σιλικόνη που αποτελεί το πακέτο του επεξεργαστή. Ο παραλληλισμός μάλιστα έχει αρχίσει να κάνει τους κλασσικούς επεξεργαστές να δείχνουν απαρχαιωμένους καθώς πολλά προβλήματα σήμερα λύνονται σε κάρτες γραφικών(GPU) όπου μικροί και αδύναμοι πυρήνες αλλά πολλοί περισσότεροι σε σχέση με μια κεντρική μονάδα επεξεργασίας(CPU) μπορούν να λύσουν προβλήματα πολύ πιο γρήγορα.

Παρακάτω δίνονται ενδεικτικά μερικές βελτιώσεις που θα βελτίωναν την χρηστικότητα της βιβλιοθήκης:

- **Γραφικά:** Οι περισσότερες από τις συναρτήσεις δίνουν άμεσα αποτελέσματα χωρίς ενδιάμεσα βήματα και εικόνες, κάτι που δεν είναι πρόβλημα για τις περισσότερες. Υπάρχουν όμως συναρτήσεις που ή δημιουργία εικόνας είτε ως κάποιο κοινό μέσο(.jpg, .png, …) είτε ως πρόγραμμα με γραφικά θα ήταν καλή πρόσθεση. Οι συναρτήσεις που θα επωφελούνταν από αυτό είναι η BDDs και η όποια συνάρτηση δίνει αποτέλεσμα λογική έκφραση. Η BDDs θα μπορούσε να μετατρέψει τα αποτελέσματα τα τις σε ένα γράφο και ένα ξεχωριστό πρόγραμμα θα μπορούσε να φτιαχτεί με την ικανότητα να δείξει τις πύλες των λογικών εκφράσεων ενωμένες.
- **Αναλυτής(Parser)**: Θα ήταν χρήσιμο να ενοποιηθεί ο τρόπος έκφρασης τον μεταβλητών και των αριθμών της βιβλιοθήκης και με έναν parser και αντίστοιχη γραμματική να αναγνωρίζει το πρόβλημα από την είσοδο και να επιστρέφει την σωστή απάντηση χωρίς περισσότερες πληροφορίες για το πρόβλημα.
- **Εφαρμογή διαδικτύου(Webapp)**: Η αλήθεια είναι ότι η περισσότερες εφαρμογές βρίσκονται στον περιηγητή και στις φορητές συσκευές πλέον. Για αυτό η αλλαγή της βιβλιοθήκης σε webapp σε συνδυασμό με τον parser που αναφέρθηκε πιο πάνω θα ήταν μια πολύ καλή προσθήκη.

ΒΙΒΛΙΟΓΡΑΦΙΑ

1) https://en.wikipedia.org/wiki/Transistor
2) https://en.wikipedia.org/wiki/Binary_number
3) M. Morris Mano, Michael D .Ciletti, Ψηφιακή Σχεδίαση, 5η έκδοση(2014),σελ.4
4) M. Morris Mano, Michael D .Ciletti, Ψηφιακή Σχεδίαση, 5η έκδοση(2014),σελ.6
5) M. Morris Mano, Michael D .Ciletti, Ψηφιακή Σχεδίαση, 5η έκδοση(2014),σελ.5
6) M. Morris Mano, Michael D .Ciletti, Ψηφιακή Σχεδίαση, 5η έκδοση(2014),σελ.49
7) M. Morris Mano, Michael D .Ciletti, Ψηφιακή Σχεδίαση, 5η έκδοση(2014),σελ.61
8) M. Morris Mano, Michael D .Ciletti, Ψηφιακή Σχεδίαση, 5η έκδοση(2014),σελ.73
9) https://en.wikipedia.org/wiki/Quine%E2%80%93McCluskey_algorithm
10) https://en.wikipedia.org/wiki/Binary_decision_diagram
11) https://en.wikipedia.org/wiki/Binary_decision_diagram
12) https://en.wikipedia.org/wiki/BCD_(character_encoding)
13) https://en.wikipedia.org/wiki/Aiken_code
14) https://en.wikipedia.org/wiki/Bi-quinary_coded_decimal
15) https://en.wikipedia.org/wiki/Gray_code
16) https://en.wikipedia.org/wiki/C_(programming_language)
17) https://en.wikipedia.org/wiki/Python_(programming_language)
18) Νίκος Μ. Χατζηγιαννάκης, Η γλώσσα C σε βάθος, 4η έκδοση, σελ. 525
19) https://en.wikipedia.org/wiki/Regular_expression
20) https://en.wikipedia.org/wiki/Moore%27s_law
21) https://www.youtube.com/watch?v=cMWGeJyrc9w
22) https://gcc.gnu.org/
23) https://www.python.org/
24) https://isocpp.org/
25) https://www.java.com/en/
26) https://numpy.org/
27) https://cplusplus.com/reference/cstdio/printf/
28) https://www.intel.com/content/www/us/en/homepage.html
29) https://valgrind.org/
30) https://cppcheck.sourceforge.io/
31) https://www.sourceware.org/gdb/
32) https://www.gnu.org/software/make/
33) http://www.throwtheswitch.org/unity
34) https://github.com/google/googletest
35) https://learn.microsoft.com/en-us/windows/wsl/install
36) https://www.microsoft.com/en-us/windows?r=1
37) https://en.wikipedia.org/wiki/Linux
38) https://learn.microsoft.com/en-us/powershell/
39) https://ubuntu.com/
40) https://fortran-lang.org/
41) https://www.oracle.com/solaris/solaris11/
42) https://www.ibm.com/products/aix
43) https://www.freebsd.org/
44) https://www.hpe.com/us/en/servers/hp-ux.html
45) https://support.apple.com/macos
46) https://www.openmp.org/

47) https://valgrind.org/docs/manual/cl-manual.html
48) https://www.mingw-w64.org/
49) https://github.com/astrofrog/psrecord
50) https://matplotlib.org/