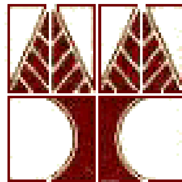
Ατομική Διπλωματική Εργασία

ΠΡΟΣΟΜΙΩΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΠΑΡΑΛΛΗΛΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΥΝΕΚΤΙΚΩΝ ΣΥΝΙΣΤΩΣΩΝ ΣΤΗΝ ΠΛΑΤΦΟΡΜΑ XMT

Χαρίκλεια Ζαχαριάδη

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2014

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΡΟΣΟΜΙΩΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΠΑΡΑΛΛΗΛΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΥΝΕΚΤΙΚΩΝ ΣΥΝΙΣΤΩΣΩΝ ΣΤΗΝ ΠΛΑΤΦΟΡΜΑ XMT

Χαρίκλεια Ζαχαριάδη

Επιβλέπων Καθηγητής

Χρύσης Γεωργίου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2014

Ευχαριστίες

Ιδιαίτερες ευχαριστίες θα ήθελα να εκφράσω στον υπεύθυνο καθηγητή και επιβλέπον της πτυχιακής εργασίας αυτής, κύριο Χρύση Γεωργίου που για ένα ολόκληρο χρόνο με τη βοήθεια και τη συμβολή του ήρθε σε πέρας η εργασία αυτή. Επίσης θα ήθελα να τον ευχαριστήσω για τον χρόνο του και την υπομονή που έδειξε όλο αυτό το διάστημα. Ευχαριστίες θέλω να δώσω επιπλέον και στο οικογενειακό και φιλικό μου περιβάλλον που ήταν δίπλα μου όλη την περίοδο.

Περίληψη

Το κύριο θέμα με το οποίο ασχολείται η παρούσα διπλωματική εργασία είναι η παράλληλη επεξεργασία, δηλαδή εν συντομία η συνεργασία πολλών επεξεργαστών ταυτόχρονα για την επίλυση κάποιου συγκεκριμένου προβλήματος και την εξαγωγή του επιθυμητού αποτελέσματος. Στα αρχικά βήματα της Ατομικής Διπλωματικής Εργασίας αυτής μελετήθηκε η έννοια του παραλληλισμού και όλων των σχετικών ορισμών γύρο από αυτόν αλλά και του προσομοιωτή που με τη βοήθεια του γίνεται η υλοποίηση. Σε μετέπιπτα βήματα μελετήθηκε συγκεκριμένο πρόβλημα, όπως είναι το πρόβλημα των συνεκτικών συνιστωσών και πως λύνεται σειριακά αλλά και παράλληλα που είναι το κύριο θέμα. Αφού κατανοήθηκε ακολούθησε η υλοποίηση στον προσομοιωτή XMT με την βοήθεια της γλώσσας προγραμματισμού XMTC. Ακολούθησαν διάφορα πειράματα με διαφορετικά δεδομένα στον συγκεκριμένο αλγόριθμο όπου και πάρθηκαν οι μετρήσεις που αφορούσαν κυρίως τον χρόνου και το κόστος που χρειάζεται ο αλγόριθμος να λύσει το πρόβλημα. Δεν αρκεί να λυθεί παράλληλα αλλά και να εξάγει σωστά αποτελέσματα.

Μεγάλη βαρύτητα στην Διπλωματική Εργασία αυτή δίνεται στη σύγκριση των αναμενόμενων αποτελεσμάτων με τα αποτελέσματα που λαμβάνουμε. Με τον όρο αναμενόμενα αποτελέσματα εννοούμαι την θεωρητική αξιολόγηση του αλγορίθμου που εκφράζει τι θα περιμέναμε να δούμε από τον αλγόριθμο όσον αφορά τον χρόνο αλλά και το κόστος του. Τα αποτελέσματα που λαμβάνουμε από την άλλη προκύπτουν από τα πειράματα που έγιναν στον υλοποιημένο αλγόριθμο. Επίσης ακόμα μια σύγκριση που γίνεται είναι η σύγκριση του εδραιωμένου παράλληλου αλγόριθμου συνεκτικών συνιστωσών και μιας παραλλαγής του αλγορίθμου αυτού. Έχει μεγάλο ενδιαφέρον να παρατηρήσουμε τα αποτελέσματα που παίρνουμε, έτσι ώστε να εξάγουμε και να οδηγηθούμε στα ανάλογα συμπεράσματα που προκύπτουν.

Περιεχόμενα

Κεφάλαιο 1 **Εισαγωγή**………………………………………………………………… **1**

1.1 Στόχος-Κίνητρο 1

1.2 Μεθοδολογία 2

1.3 Δομή Κειμένου 3

Κεφάλαιο 2 **Παράλληλος Υπολογισμός**……………………………………………. **5**

2.1 Ιστορικά 5

2.2 Παραλληλισμός 6

2.3 Παράλληλη Επεξεργασία 7

2.4 Γιατί Παράλληλη Επεξεργασία 9

2.5 Παράλληλος Αλγόριθμος 10

2.6 Κριτήρια Αξιολόγησης Παράλληλων Αλγορίθμων 11

2.7 Μοντέλα Παράλληλου Αλγόριθμου 14

2.8 Μοντέλα Παράλληλου Υπολογισμού PRAM 17

Κεφάλαιο 3 **Πλατφόρμα XMT** ……..………………………………………….. **22**

3.1 Πλατφόρμα XMT 22

3.2 Γλώσσα Προγραμματισμού XMTC 25

3.3 Περιορισμοί 34

Κεφάλαιο 4 **Παράλληλη Αναζήτηση**…………………………….……………. **36**

4.1 Πρόβλημα 37

4.2 Σειριακός Αλγόριθμος 37

4.3 Παράλληλος Αλγόριθμος 40

4.4 Πειραματική Αξιολόγηση 45

Κεφάλαιο 5 **Συνεκτικές Συνιστώσες**…………………………………………… **52**

5.1 Πρόβλημα 52

5.2 Σειριακός Αλγόριθμος 53

5.3 Παράλληλοι Αλγόριθμοι 57

5.4 Παραλλαγή Αλγορίθμου Συνεκτικών Συνιστωσών 66

5.5 Σύγκριση Αλγορίθμου 70

ΚΕΦΑΛΑΙΟ **6 Συμπεράσματα** …………………………………………………….. **74**

6.1 Τελικά Συμπεράσματα 74

6.2 Προβλήματα Που Συναντήθηκαν 75

6.3 Οφέλη Ατομικής Διπλωματικής Εργασίας 76

6.4 Μελλοντική Εργασία 77

Β ι β λ ι ο γ ρ α φ ί α ……………………………………………………………. **79**

Π α ρ ά ρ τ η μ α Α……………………….……..…………………………….………… **A-1**

Κεφάλαιο 1

Εισαγωγή

1.1 Στόχος-Κίνητρο 1

1.2 Μεθοδολογία 2

1.3 Δομή Κειμένου 3

1.1 Στόχος-Κίνητρο

Αναμφίβολα εδώ και κάποιες δεκαετίες βασικό και κύριο κομμάτι της Επιστήμης της Πληροφορικής είναι οι πολυπύρηνες, δηλαδή πολλοί πυρήνες μαζί που συνεργάζονται με κάποιο συγκεκριμένο τρόπο επικοινωνώντας είτε με ανταλλαγή μηνύματα είτε μέσω κοινόχρηστης μνήμης. Αυτό το κομμάτι δεν είναι κάτι που πρέπει να αγνοήσουμε αν θέλουμε να ακολουθούμε τα βήματα της Πληροφορικής και να προοδεύουμε, αλλά αντιθέτως με μελέτη και έρευνα στον τομέα αυτό μπορούμε να εξάγουμε νέα συμπεράσματα και δεδομένα που θα είναι βοηθητικά για εξέλιξη και πρόοδο γενικότερα στην Επιστήμη της Πληροφορικής.

Σε προσωπικό επίπεδο η πρώτη μου επαφή που ήταν και η αρχή του ενδιαφέροντος μου για την παράλληλη επεξεργασία και υπολογισμό ήταν η παρακολούθηση του μαθήματος ¨ΕΠΛ431:Σύνθεση Παράλληλων Αλγορίθμων¨, όπου πήρα και τα πρώτα εφόδια για τον παραλληλισμό, κατανοώντας σε μεγάλο βαθμό όλες τις έννοιες που περικλείει στην οικογένεια της η παράλληλη επεξεργασία.

Ένα κομμάτι της εργασίας αυτής που αποτελεί και την αρχή της, όπως ανέφερα είναι η κατανόηση του παραλληλισμού και η εξοικείωση με την πλατφόρμα XMT (eXplicit Multi Threading)[2] και γλώσσα XMTC[2] που είναι τα μέσα για μελέτη και υλοποίηση παράλληλων αλγορίθμων. Μεγάλη έμφαση όμως δίνεται ίσως στο σημαντικότερο κομμάτι που ασχολείται η πτυχιακή εργασία αυτή, που δεν είναι άλλο από την σύγκριση που θέλουμε ανάμεσα στην θεωρητική αξιολόγηση και στην πειραματική αξιολόγηση. Μας ενδιαφέρει σε μεγάλο βαθμό να δούμε αν το μοντέλο μας συμπεριφέρεται έτσι όπως αναμένουμε ή υπάρχουν κάποιοι παράγοντες που το επηρεάζουν και αντιδρά διαφορετικά. Ασχολούμαστε

επίσης με την σύγκριση του διαδεδομένου αλγορίθμου Συνεκτικών Συνιστωσών[4,8] και κάποιας παραλλαγής του αλγορίθμου όπου επιλύει το ίδιο πρόβλημα, με τις πληροφορίες που εξάγουμε από τα ακατέργαστα δεδομένα, παίρνουμε πάλι κάποια συμπεράσματα μέσα και από αυτή την σύγκριση.
Με την πάροδο του χρόνου και καθώς έτρεχε η παρούσα διπλωματική εργασία δεν θα μπορούσε να μην με απασχολήσει ιδιαίτερα και η γλώσσα XMTC. Διαπίστωσα ότι υστερεί σε αρκετά σημεία λόγω των περιορισμών της, κάτι που δυσκολεύει στην πρόοδο και ανέλιξη του παράλληλου υπολογισμού και επεξεργασίας. Η βελτίωση της είναι αναγκαίο κακό θα μπορούσαμε να πούμε.

1.2 Μεθοδολογία

Η μεθοδολογία και ο προγραμματισμός κατά την διάρκεια της διεξαγωγής μιας Διπλωματικής Εργασίας είναι πολύ σημαντικά κομμάτια έτσι ώστε να αναπτυχθεί ομαλά και με τον σωστό τρόπο χωρίς να παραλείπεται κάποιο κομμάτι. Από την αρχή είχε δοθεί από τον κύριο Χρύση Γεωργίου σε εμένα ένα χρονοδιάγραμμα με όλα τα κομμάτια που έπρεπε να γίνουν με την σειρά είτε αυτά ήταν μελέτη, είτε υλοποίηση στην πλατφόρμα.
Η αρχή της Διπλωματικής Εργασίας έγινε θέτοντας τις βάσεις και τα θεμέλια. Ο μόνος τρόπος για να γίνει αυτό σωστά είναι η κατανόηση τις ευρύτερης έννοιας του παραλληλισμού και όλων των ορισμών που συσχετίζονται με αυτόν. Μόνο με την πλήρη κατανόηση των ορισμών θα μπορούσα να προχωρήσω, γιατί θα με βοηθούσε να καταλαβαίνω τι ακριβώς σημαίνουν, οι περιορισμοί και απαιτήσεις που έχει ο κάθε ορισμός. Σε αυτό το κομμάτι ήταν πολύτιμα τα εφόδια που πήρα όπως ανέφερα και ποιο πάνω, από το μάθημα που παρακολούθησα στο οποίο αναλύθηκαν όλοι οι ορισμοί αλλά σημαντικά βοήθησε και η προσωπική μελέτη που έκανα.
Αφού υπήρχε μια πλήρη κατανόηση των ορισμών το επόμενο βήμα έπρεπε να ήταν η κατανόηση του μέσου που θα μας εξυπηρετούσε με τον παραλληλισμό. Το μέσο αυτό δεν είναι άλλο από την πλατφόρμα-προσομοιωτή XMT (eXplicit Multi Threading). Αρχικά έγινε η εγκατάσταση της πλατφόρμα XMT και της γλώσσας που το συνοδεύει την XMTC, γλώσσα που είναι πολύ παρόμοια με την ήδη γνωστή και πολυδουλεμένη γλώσσα προγραμματισμού την C. Για την κατανόηση τόσο της πλατφόρμας, όσο και του μέσου χρειάστηκε προσεκτική μελέτη του εγχειριδίου (manual) που προσφέρεται στον διαδικτυακό χώρο[2]. Αφού μελετήθηκε το εγχειρίδιο, ακολούθησε η υλοποίηση κάποιων μικρών, εύκολων και απλών προγραμμάτων μέσα από την πλατφόρμα στην γλώσσα XMTC που στόχευε στην εξοικείωση με την γλώσσα και το περιβάλλον γιατί αν και μοιάζει πολύ με την γλώσσα C, υπάρχουν πολλά σημεία που διαφοροποιούνται λόγω των περιορισμών και πρέπει να είμαστε ιδιαίτερα

προσεκτικοί με τη χρήση της. Ένα πρόβλημα που υλοποιήθηκε και με βοήθησε πολύ για κατανόηση και εξάσκηση ήταν το πρόβλημα της ¨Αναζήτησης¨. Υλοποιήθηκε όπως διδάχτηκε στο μάθημα και ήταν η απαρχή για τον παράλληλο προγραμματισμό για εμένα.
Ακολούθησε η μελέτη και κατανόηση κάποιων συγκεκριμένων προβλημάτων, δηλαδή ποιο είναι το πρόβλημα και ποιο το ζητούμενο αποτέλεσμα. Αφού κατανοήθηκε γενικότερα το πρόβλημα ακολούθησε η κατανόηση συγκεκριμένου παράλληλου αλγορίθμου που λύνει αυτό το πρόβλημα. Συγκεκριμένα το πρόβλημα που μελέτησα είναι το πρόβλημα των συνεκτικών συνιστωσών, πρόβλημα που αναλύεται με λεπτομέρεια σε μεταγενέστερο σημείο.
Επόμενο βήμα ήταν η υλοποίηση του παράλληλου αλγορίθμου που μελετήθηκε αλλά και μιας παραλλαγή του αλγορίθμου γιατί ο αρχικός αλγόριθμος αντιμετώπιζε κάποια προβλήματα με τον αριθμό των επεξεργαστών και δεν θα μπορούσαμε να έχουμε αρκετές μετρήσεις που είναι και ο τελικός μας στόχος. Έτσι προτιμήθηκε να γίνει και η υλοποίηση της παραλλαγής έτσι ώστε να έχουμε περισσότερες μετρήσεις και μια γενικότερη εικόνα αν όντως παίρνουμε τα αναμενόμενα αποτελέσματα.
Όλη αυτή η διαδικασία στόχευε με το πέρας της υλοποίησης να πάρουμε μετρήσεις του χρόνου που χρειάζονται οι επεξεργαστές μας για να συνεργαστούν και να λύσουν το πρόβλημα που έχουμε. Αλλάζοντας τον αριθμό των δεδομένων πήραμε τις μετρήσεις μας όπου και μαζεύοντας τα δημιουργήσαμε τις γραφικές μας. Παράλληλα δημιουργήθηκαν και οι γραφικές με τα αναμενόμενα αποτελέσματα. Επίσης αφού έγιναν οι μετρήσεις του χρόνου, με απασχόλησε και το κόστος των αλγορίθμων. Εν κατακλείδι φτάνοντας στο τέλος όλης αυτής της διαδικασίας ήρθε και το πιο σημαντικό κομμάτι, που δεν είναι άλλο από την σύγκριση της θεωρητικής και πειραματικής αξιολόγησης. Επίσης έγινε σύγκριση και των δύο αλγορίθμων που υλοποιήθηκαν μεταξύ τους, όπου προκύπτει και ο βέλτιστος παράλληλος αλγόριθμος. Μέσα από τις συγκρίσεις αυτές βγάζουμε τα συμπεράσματα που προκύπτουν και αναφέρονται στα τελευταία σημεία της παρούσας εργασίας.
Γενικότερα θα μπορούσαμε να πούμε ότι γινόταν παράλληλη δουλεία τόσο στο πρακτικό κομμάτι, όσο και στο θεωρητικό, αφού το ένα κομμάτι συμπλήρωνε το άλλο έτσι ώστε να υπάρχει και να προκύψει μια ολοκληρωμένη δουλεία.

1.3 Δομή Κειμένου

Η Διπλωματική Εργασία αυτή αποτελείται συνολικά από έξι (6) κεφάλαια. Στο πρώτο κεφάλαιο αναπτύχτηκε ο στόχος της εργασία αυτής και μερικά λόγια για το πώς θα εξελιχτεί. Στο δεύτερο κεφάλαιο αναλύουμε την έννοια του παραλληλισμού και όλες τις σχετικές λεπτομέρειες που πρέπει να γνωρίζουμε για αυτόν έτσι ώστε να ενταχτούμε στην ιδέα του και να μπορούμε να κατανοήσουμε τα μετέπειτα στάδια. Το τρίτο κεφάλαιο μας εισάγει την

πλατφόρμα αλλά και την γλώσσα που στηρίζεται ο παραλληλισμός, στοιχεία που θα εργαστούμε και γι' αυτό είναι πολύ σημαντική η αναφορά και μελέτη τους. Το προηγούμενο κεφάλαιο ακολουθεί το κεφάλαιο τέσσερα όπου μελετάται και αναλύεται ο πρώτος αλγόριθμος που δεν είναι άλλος από τον αλγόριθμο αναζήτησης. Σε αυτό το κεφάλαιο μελετάται τόσο ο σειριακός όσο και ο παράλληλος. Στο κεφάλαιο πέντε αναλύεται ένας άλλος αλγόριθμος, αλγόριθμος συνεκτικών συνιστωσών. Αρχικά μελετάται ο σειριακός αλγόριθμος και στη συνέχεια δύο παράλληλοι αλγόριθμοι, ο ένας είναι ο κανονικός που ορίζεται και έπειτα μια παραλλαγή αυτού. Το κεφάλαιο έξι όπου είναι και το τελευταίο, γίνονται τα τελικά σχόλια και παρατηρήσεις που προέκυψαν μέσα από την εργασία.

Κεφάλαιο 2

Παράλληλος Υπολογισμός

2.1 Ιστορικά	5
2.2 Παραλληλισμός	6
2.3 Παράλληλη Επεξεργασία	7
2.4 Γιατί Παράλληλη Επεξεργασία	9
2.5 Παράλληλος Αλγόριθμος	10
2.6 Κριτήρια Αξιολόγησης Παράλληλων Αλγορίθμων	11
2.7 Μοντέλα Παράλληλου Υπολογισμού	14
2.8Μοντέλο Παράλληλου Υπολογισμού PRAM	17

2.1 Ιστορικά

Είναι κοινός αποδεκτό ότι η Επιστήμη της Πληροφορικής είναι μια πρόσφατη επιστήμη αν την συγκρίνουμε με την Επιστήμη των Μαθηματικών, της Φιλολογίας, της Ιστορίας, της Βιολογίας, της Φυσικής. Η απαρχή της επιστήμης αυτής χρονολογείται μέσα στον 20^{ο} αιώνα και πιο συγκεκριμένα θα μπορούσαμε να πούμε ότι κρίσιμη περίοδος για την Πληροφορική ήταν κατά την διάρκεια της Βιομηχανικής Επανάστασης (1750-1850), μια περίοδος που σημάδεψε ολόκληρο τον κλάδο της Πληροφορικής. Αποτελεί όμως, αν και η πιο πρόσφατη, την επιστήμη που σημείωσε τα μεγαλύτερα και γρηγορότερα άλματα-προόδους τα τελευταία χρόνια. Η εξέλιξη που σημείωσε ήταν ραγδαία και καθοριστική στην ζωή όλων των ανθρώπων ανεξαρτήτως σε ποια χώρα ζούσαν. Το μέλλον διαγράφεται να κινείται με τους ίδιους ρυθμούς, επιφυλάσσοντας σημαντικές προόδους που θα μας αλλάζουν της ζωή προς το καλύτερο, έτσι, το μόνο που έχουμε να κάνουμε είναι να προχωρούμε μαζί και συνοδοιπόροι προσφέροντας και εμείς ότι καλύτερο μπορούμε.

Το μεγαλύτερο άλμα που σημειώθηκε στον κλάδο της Πληροφορικής μέχρι τώρα υποστηρίζεται ότι είναι η εύρεση και η εισαγωγή πολλών πυρήνων σε μόνο ένα τσιπ, δηλαδή την χρήση των πολυεπεξεργαστών. Η εταιρία INTEL ήταν αυτή του έφτιαξε και εδραίωσε πρώτη τον μικροεπεξεργαστή αρχές του 1970 και δεν ήταν άλλος από ένα επεξεργαστή 4-bit(4004). Θεωρείται ότι ο πρώτος ψηφιακός υπολογιστής ENIAC1 με εικοσιπέντε (25) ανεξάρτητες μονάδες υπολογιστών να συνεργάζονται για την επίλυση ενός και μόνο

προβλήματος. Ακολούθησαν πολλά σημαντικά και μεγάλα βήματα που οδήγησαν στην κατάληξη το 2013 ένας απλός υπολογιστής να έχει την δυνατότητα να έχει πολλαπλούς πολυεπεξεργαστές με 3,6GHz. Το πολύ μικρό μέγεθος που έχει ένας μικροεπεξεργαστής και το σχετικά μικρό κόστος του λόγο της χαμηλής τιμής, δίνουν την ευχέρεια σε όλους εμάς να έχουμε στην διαθεσιμότητα μας μέσο του υπολογιστή μας τέτοιους μικροεπεξεργαστές. Αφού έχουμε την δυνατότητα από πλευράς υλικού (hardware) να έχουμε στα χέρια μας αυτά τα μέσα το μόνο που απομένει και είναι το βασικότερο κομμάτι είναι να τα χρησιμοποιούμε σωστά, στοχεύοντας στην μέγιστη αποτελεσματικότητα και κέρδος μέσα από την χρήση τους. Κάπου εδώ εισάγεται και η έννοια του παραλληλισμού, αφού η παράλληλη επεξεργασία έχει να κάνει με τον συντονισμό περισσότερων από ένα επεξεργαστή που δουλεύουν ταυτόχρονα για την διεκπεραίωση κάποιας εργασίας, έννοια που εξηγείται με λεπτομέρεια στο επόμενο υποκεφάλαιο.

2.2 Παραλληλισμός

Η παράλληλη επεξεργασία αποτελεί αναπόσπαστο κομμάτι της καθημερινής μας ζωής εδώ και πολλά χρόνια· ίσως θα μπορούσαμε να πούμε από τον καιρό που υπάρχει η ανθρώπινη ζωή χωρίς όμως να το αντιλαμβανόμαστε και να το συνειδητοποιούμε. Ιστορικά στοιχεία αποδεικνύουν ότι δεν είναι νέοι ορισμοί, όπως πολλοί τοποθετούνται, αλλά ότι οι Έλληνες εδώ και περίπου 2000 χρόνια πριν, συνέλαβαν τον ορισμό και την έννοια του παραλληλισμού. Πολλές φορές λαμβάνουμε μέρος σε τέτοιες παράλληλες δουλειές, υπάρχουν όμως και οι περιπτώσεις που γινόμαστε δέκτες των αποτελεσμάτων παράλληλων εργασιών. Για να γίνει αντιληπτό όμως, πρέπει να σκεφτούμε ότι εάν στην καθημερινότητα μας έρθουμε αντιμέτωποι με ένα δύσκολο πρόβλημα ή χρονοβόρο ζητάμε την βοήθεια από κάποιους που έχουν και αυτοί την ειδική γνώση, δηλαδή είναι όμοιοι με εμάς, έτσι ώστε να το λύσουμε, είτε ζητάμε την βοήθεια από μια ομάδα ανθρώπων που μας δίνουν λύση στο πρόβλημα μας. Ένα παράδειγμα που μπορούμε να δώσουμε από την καθημερινότητα μας έτσι ώστε να γίνει πιο κατανοητό είναι η εργασία στις οικοδομές. Για παράδειγμα στις οικοδομές όπου κτίζεται ένα καινούριο κτήριο βλέπουμε ότι εργάζονται πέρα του ενός κτίστη, όπου ο καθένας εργάζεται με τον ίδιο τρόπο σε διαφορετικό όμως σημείο της οικοδομής. Με το να δουλεύουν μαζί, ταυτόχρονα, κερδίζουμε χρόνο, που θα ολοκληρωνόταν η δουλειά αν κτιζόταν από ένα και μόνο κτίστη. Αναμφίβολα η ολοκλήρωση της εργασίας γίνεται σε μικρότερο χρονικό διάστημα με την παράλληλη εργασία και συνεργασία των εργατών της οικοδομής. Αν για παράδειγμα είχαμε να κάνουμε μια εργασία και δεδομένου ότι μπορούσαμε να την σπάσουμε σε 100 ανεξάρτητα κομμάτια και βάζαμε

100 εργάτες να κάνουν ένα από τα ανεξάρτητα κομμάτια της, η εργασία μας θα γινόταν 100 φορές γρηγορότερα παρά η αρχική εργασία με μόνο ένα εργάτη. Ο χρόνος που κερδίζουμε είναι φανερά μικρότερος. Ένα σημείο που πρέπει να τονίσουμε επίσης είναι ότι οι εργάτες έχουν κοινά χαρακτηριστικά, αφού είναι όλοι γνώστες του αντικειμένου.

Παράλληλη εργασία θα μπορούσαμε να ορίσουμε την ταυτόχρονη εργασία ατόμων/μηχανών που έχουν κάποια κοινά χαρακτηριστικά έτσι ώστε να διεκπεραιώνουν και να δώσουν λύση σε ένα κοινό τους πρόβλημα πιο γρήγορα, είτε σε ένα πρόβλημα που δεν θα ήταν εφικτό να λυθεί με τη χρήση ενός και μόνου ατόμου/μηχανής.

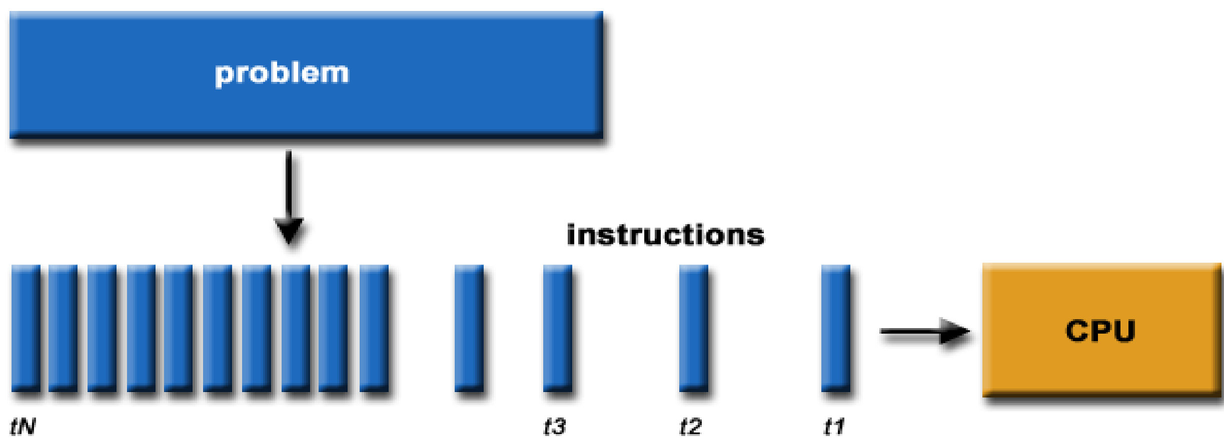
2.3 Παράλληλη Επεξεργασία

Θέλοντας να μιλήσουμε πιο συγκεκριμένα για το κομμάτι που μας αφορά θα ασχοληθούμε και θα αναλύσουμε την παράλληλη επεξεργασία που αναφέρεται σε υπολογιστικά συστήματα όπου είναι και ο τομέας που ασχολούμαστε.

Με τον όρο παράλληλη επεξεργασία εννοούμαι την ανάπτυξη προγραμμάτων τα οποία εκμεταλλεύονται την ύπαρξη πολλαπλών μονάδων σε ένα πολυεπεξεργαστή ή πολυυπολογιστή, δηλαδή η ύπαρξη περισσότερων από ένα επεξεργαστή, για να επιτύχουν αύξηση υπολογιστικών επιδόσεων, μείωση του απαιτούμενου χρόνου εκτέλεσης του προγράμματος και επίλυση ενός προβλήματος. Εδώ πλέον μπορούμε να μιλάμε για επεξεργαστές που δουλεύουν παράλληλα αντί για εργάτες και διεργασίες που αντιστοιχούν στις εργασίες σε ποιο πάνω αναφορά/παράδειγμα. Για την υλοποίηση μιας παράλληλης επεξεργασίας απαιτείται ένας παράλληλος υπολογιστής, όπου είναι μια συλλογή από επεξεργαστές του ίδιου τύπου, συνδεδεμένοι μεταξύ τους με τέτοιο τρόπο που επιτρέπουν τον συντονισμό των ενεργειών τους και την ανταλλαγή δεδομένων μεταξύ τους. Οι επεξεργαστές που αποτελούν τον παράλληλο υπολογιστή απαιτείται να βρίσκονται/τοποθετούνται κοντά ο ένας από τον άλλο, δηλαδή να έχουν μικρή διαφορά στην απόσταση. Ο τρόπος ταξινόμησης των παράλληλων επεξεργαστών εξαρτάται από τα αρχιτεκτονικά χαρακτηριστικά και λειτουργείς του υπολογιστή. Συγκεκριμένα αυτά τα χαρακτηριστικά περιλαμβάνουν τον τύπο και τον αριθμό των επεξεργαστών. Επίσης ρόλο παίζουν οι διασυνδέσεις μεταξύ των επεξεργαστών αλλά και ο αντίστοιχος τρόπος επικοινωνίας μεταξύ τους που υποστηρίζει ο παράλληλος υπολογιστής. Επιπλέον χαρακτηριστικά είναι ο ολικός έλεγχος, ο τρόπος που συγχρονίζονται, καθώς επίσης και ο τρόπος που εισάγονται και εξάγονται τα δεδομένα μας. Επίσης πρέπει να είναι του ίδιου τύπου, να έχουν κοινά χαρακτηριστικά και να ανήκουν στην ίδια οικογένεια. Κοινός τους στόχος είναι να λύσουν από κοινού ένα πρόβλημα όσο το δυνατό γρηγορότερα. Ο άλλος

τρόπος που προσφέρεται για επεξεργασία εκτός από τον παράλληλο είναι η κατανεμημένη επεξεργασία που την βρίσκουμε σε κατανεμημένους υπολογιστές με κατανεμημένα συστήματα. Τα συστήματα αυτά είναι ένα σύνολο από επεξεργαστές που μπορεί να αποτελείται από διαφορετικού τύπου επεξεργαστές (ετερογενής), που κατανέμονται σε μια σχετικά μεγάλη γεωγραφική περιοχή. Πρωταρχικός στόχος είναι να χρησιμοποιηθεί κατανεμημένη πηγή για να μαζευτούν πληροφορίες και δεδομένα για να διαβιβαστούν πάνω σε ένα συνδεδεμένο δίκτυο με πολλούς επεξεργαστές. Μια εργασία μπορεί να ανατεθεί σε κάποιο επεξεργαστή καθώς άλλοι επεξεργαστές ανέλαβαν να διεκπεραιώσουν κάποια άλλη εργασία και στο τέλος να επικοινωνήσουν δίνοντας το σωστό αποτέλεσμα. Παρατηρούμε ότι και στα δυο συστήματα έχουμε την ύπαρξη πολλών επεξεργαστών με την διαφορά ότι στα παράλληλα συστήματα τρέχουν όλοι την ίδια διεργασία, ενώ στα κατανεμημένα συστήματα μπορεί ο κάθε επεξεργαστής να τρέχει διαφορετική διεργασία. Το μεγάλο άλμα όμως δεν ήταν από τα κατανεμημένα συστήματα στα παράλληλα που οι διαφορές τους αναφέρονται πιο πάνω, αλλά από τα σειριακά συστήματα στα παράλληλα.[3]

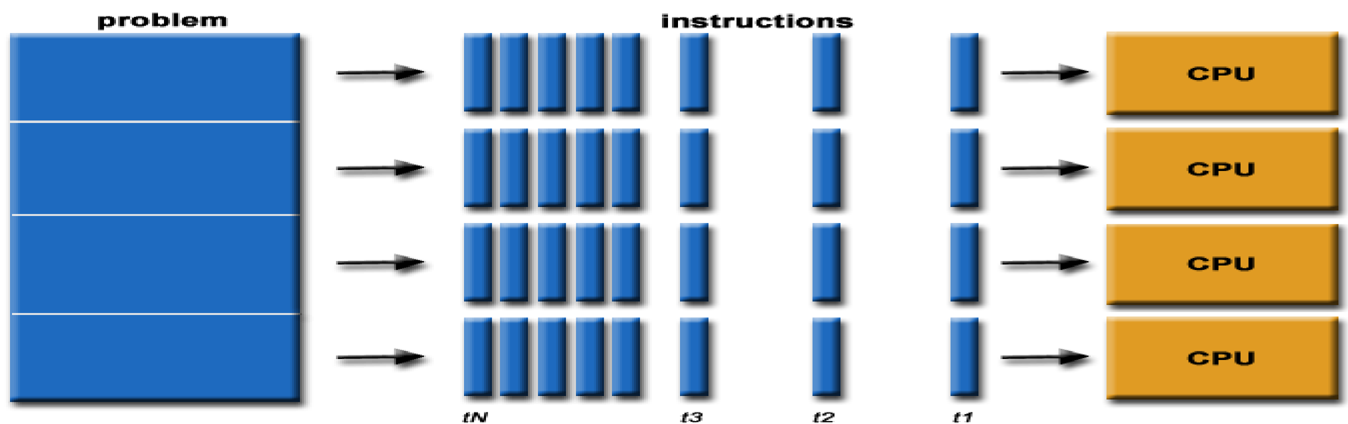
Παραδοσιακά από τον αρχή της Επιστήμης αυτής, όταν ξεκίνησαν να συλλαμβάνονται οι πρώτοι ορισμοί και να γίνονται οι πρώτες απόπειρες για συγγραφή λογισμικών και αλγορίθμων η μόνη ιδέα που κυριαρχούσε ήταν η ακολουθιακή δομή (σειριακή), όπου ο υπολογιστής θεωρείτο ότι είχε μια Κεντρική Μονάδα Επεξεργασίας (CPU) και το πρόβλημα λυνόταν με την εκτέλεση μιας σειράς από εντολές όπου αυτές οι εντολές προϋπόθεταν την εκτέλεση και τερματισμό της προηγούμενης εντολής. Σχηματική επεξήγηση στο Σχήμα 2.1.



Σχήμα 2.1[3]

Αν θέλουμε να συγκρίνουμε την σειριακή με την παράλληλη επεξεργασία θα προσπαθήσουμε να εξηγήσουμε την παράλληλη με τον απλούστερο τρόπο όπου θεωρείται ότι ο υπολογιστής διαθέτει πολλές Κεντρικές Μονάδες Επεξεργασίας(CPU), το πρόβλημα διασπάτε σε διακριτά τμήματα που μπορούν να δουλεύουν ταυτόχρονα και κάθε τμήμα

επιλύει ακολουθιακά το κομμάτι του.[3] Σχηματική Αναπαράσταση της παράλληλης επεξεργασίας παρουσιάζεται στο Σχήμα2.2 πιο κάτω, έτσι ώστε να γίνει πιο κατανοητό.



Σχήμα 2.2[3]

2.4 Γιατί Παράλληλη Επεξεργασία

Το ερώτημα που εύστοχα δημιουργείται είναι πως δημιουργήθηκε η ανάγκη για παράλληλη επεξεργασία και τι ήταν αυτό που μας βοήθησε σε αυτό το άλμα;
Με το πέρασμα του χρόνου ο όγκος των δεδομένων και πληροφοριών αυξάνεται με αστραπιαίους ρυθμούς, μαζί με αυτά και οι ανάγκες και απαιτήσεις από τον Πληροφοριακό κόσμο. Ζούμε σε μια ανατρεπτική εποχή που δεν συμβαδίζει με την στασιμότητα αλλά αντιθέτως, η πρόοδος είναι η μόνη λέξη που την εκφράζει απόλυτα, έτσι αναγκαστικά ο σειριακός υπολογισμός έπρεπε να σημειώσει εξέλιξη και οδηγηθήκαμε στον παραλληλισμό. Η μεγάλη ποσότητα δεδομένων και πληροφοριών σε συνδυασμό με την απαίτηση να έχουμε τα αποτελέσματα όσο το δυνατό γρηγορότερα(χρόνος) σχετίζονται με τον παράλληλο υπολογισμό. Η ανάγκη για γρήγορες λύσεις που κάνουν χρήση μεγαλύτερου όγκου δεδομένα προκύπτει από ένα μεγάλο εύρος εφαρμογών, όπως για παράδειγμα την ρευστοδυναμική, πρόβλεψη καιρού, σεισμολογία, γενετική, σχεδιασμός και προσομοίωση μεγάλων συστημάτων, επεξεργασία και εξαγωγή δεδομένων, επεξεργασία εικόνας, τεχνητή νοημοσύνη ,αυτοματοποιημένη παραγωγή, κοσμολογία-αστρονομία και πολλές άλλες. Μια νέα εφαρμογή που έχει πολύ ενδιαφέρον και απαιτεί τεράστια υπολογιστική δύναμη, πράγμα που προσφέρεται από την παράλληλη επεξεργασία είναι το SETI@home, μια υποσχόμενη έρευνα και μελέτη που εξελίσσεται καθημερινά. Επίσης οι οικονομικές και επιχειρηματικές εφαρμογές είναι εξίσου σημαντικές και βρίσκουν και αυτές εφαρμογή στην παράλληλη επεξεργασία αφού απαιτούν διαχείριση μεγάλου όγκου δεδομένων.[8]
Δεν αρκεί μόνο όμως η ανάγκη να έχουμε παράλληλη επεξεργασία αλλά αν διαθέτουμε και τα μέσα που συμβάλουν υπέρ της ισχυρής δύναμης που θέλουμε να έχει. Κατά κύριο λόγο

μπορούμε να ορίσουμε τρεις (3) βασικούς παράγοντες. Αρχικά το κόστος του υλικού που απαιτείται έχει σημειώσει δραματική μείωση, γι' αυτό το λόγο έχουμε πλέον την δυνατότητα να κτίζουμε συστήματα με πολλούς επεξεργαστές σε ένα λογικό κόστος. Το υλικό γίνεται συνεχώς φθηνότερο αλλά και ισχυρότερο ενώ το λογισμικό είναι δημόσια διαθέσιμο και βελτιώνεται συνεχώς. Η παράλληλη επεξεργασία προσπαθεί να εκμεταλλευτεί όσο γίνεται πιο αποδοτικά το άφθονο υλικό, έτσι ώστε με τη βοήθεια ειδικού λογισμικού να επιτύχει μέγιστη επεξεργασία δεδομένων στον ελάχιστο χρόνο. Ένας δεύτερος λόγος που συμβάλει στην πρόοδο τέτοιων συστημάτων είναι η βελτίωση που σημειώθηκε στα κυκλώματα VLSI. Με την βοήθεια της τεχνολογίας είναι πλέον δυνατό να σχεδιάζουμε τσιπς με πολύπλοκα συστήματα που απαιτούν εκατομμύρια τρανζίστορ. Ο τελευταίος παράγοντας που μας βοηθά στην εξέλιξη της παράλληλης επεξεργασίας είναι οι γρηγορότεροι κύκλοι που μπορούν να σημειωθούν από τους φυσικούς περιορισμούς, δηλαδή πόσο γρήγορα μπορούν να κινηθούν τα δεδομένα μέσα από το υλικό μας, ένας τομέας που σημείωσε επίσης μεγάλη πρόοδο με το πέρας του χρόνου

Συμπερασματικά μπορούμε να πούμε η ανάγκη για διαχείριση μεγάλων όγκων δεδομένων σε μικρό χρονικό διάστημα σε συνδυασμό με την ευχέρεια που έχουμε όσο αφορά την τεχνολογική ανάπτυξη, οδήγησαν στην ανάγκη και πρόοδο για παράλληλη επεξεργασία. Όλοι αυτοί οι παράγοντες έχουν ωθήσει τους ερευνητές στο να μελετήσουν τον παραλληλισμό και την δυνατότητα χρήσης του στην καθημερινότητα μας σε μεγάλο βαθμό.[4]

2.5 Παράλληλος Αλγόριθμος

Ο κύριος στόχος όλων όσων ασχολούνται με τον παράλληλο υπολογισμό είναι να βρίσκουν και να παρουσιάζουν παράλληλους αλγόριθμους που λύνουν κάποια συγκεκριμένα προβλήματα πάνω σε παράλληλες μηχανές. Η εύρεση παράλληλων αλγορίθμων είναι μια δουλειά που δεν σταματά ποτέ, αφού συνεχώς οι ερευνητές προσπαθούν ολοένα και περισσότερα προβλήματα που έχουν ήδη κάποια λύση σε σειριακό αλγόριθμο να βρουν κάποια αντίστοιχη λύση που να λύνει το πρόβλημα σωστά αλλά και παράλληλα. Πιο κάτω ορίζεται ο ορισμός του μέχρι τώρα γνωστού μας αλγόριθμου αλλά και ο νέος ορισμός που εισάγαμε, ο παράλληλος αλγόριθμος.

Αλγόριθμος είναι μια πεπερασμένη ακολουθία εντολών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο σε σειριακό υπολογιστή, που οδηγούν στην επιτυχή επίλυση ενός προβλήματος.[9]

Παράλληλος Αλγόριθμος αναφέρεται σε μια ακολουθία πεπερασμένου συνόλου εντολών, οι οποίες ακολουθούν συντακτικούς κανόνες ειδικά για παράλληλα συστήματα, των οποίων η

εκτέλεση σε παράλληλο υπολογιστικό σύστημα σε πεπερασμένο χρόνο οδηγεί σε επιτυχή επίλυση ενός υπολογιστικού προβλήματος.[8]

Πιο συγκεκριμένα μπορούμε να πούμε ότι στο σειριακό αλγόριθμο ο κώδικας εκτελείται από ένα και μόνο επεξεργαστή και κάνει ακολουθιακά, δηλαδή με την σειρά τις εντολές που αποτελούν τον κώδικα, ενώ στον παράλληλο αλγόριθμο οι εντολές του κώδικα εκτελούνται από όλους τις επεξεργαστές την ίδια στιγμή, δηλαδή κάνουν όλοι την ίδια δουλειά.

Παράδειγμα:

Θέλουμε να αρχικοποιήσουμε ένα πίνακα n θέσεων με την τιμή 1 σε όλες τις κυψελίδες του.

Σειριακή Λύση:

```
for(i=0; i<n; i++){
        table[i]=1;
}
```

Παράλληλη Λύση:

```
Processors i=1 to n do in parallel
        table[i]=1;
end do
```

Στη σειριακή λύση ένας επεξεργαστής θα κάνει n φορές την ίδια δουλειά, δηλαδή θα αρχικοποιήσει και τις n θέσεις ένας και μόνο επεξεργαστής, ενώ στην παράλληλη λύση θα συνεργαστούν n επεξεργαστές και ο καθένας θα γράψει σε μια από τις n θέσεις του πίνακα την τιμή 1.

Ένα αποδεκτό μοντέλο σχεδιασμού και ανάλυσης σειριακών αλγορίθμων περιλαμβάνει μια Κεντρική Μονάδα Επεξεργασίας με Μνήμη Τυχαίας Προσπέλασης να συμπεριλαμβάνεται. Η επιτυχία αυτού του μοντέλου κυρίως οφείλεται στην απλότητα και στην ικανότητα να συλλαμβάνει σε μεγάλο βαθμό την συμπεριφορά σειριακών αλγορίθμων. Δυστυχώς, ο παράλληλος υπολογισμός υποφέρει από έλλειψη κοινά αποδεκτών μοντέλων-αλγορίθμων εξαιτίας της επιπλέων πολυπλοκότητας που εμφανίζεται λόγω της παρουσίας ενός συνόλου από επεξεργαστές.

2.6 Κριτήρια Αξιολόγησης Παράλληλων Αλγορίθμων

Όπως στην περίπτωση του σειριακού αλγορίθμου, υπάρχουν αρκετά σημαντικά κριτήρια, σαν ο χρόνος, ο χώρος και ο προγραμματισμός έτσι και στους παράλληλους. Η περίπτωση με τους παράλληλους αλγορίθμους είναι πιο περίπλοκη λόγω της παρουσίας επιπλέον

παραμέτρων. Μερικές από τις παραμέτρους είναι ο αριθμός των επεξεργαστών, η χωρητικότητα της τοπικής μνήμης, το επικοινωνιακό σύστημα αλλά και το πρωτόκολλο συγχρονισμού. Για να ελέγξουμε την πολυπλοκότητα κάποιου αλγορίθμου θα ξεκινήσουμε εισάγοντας τους ορισμούς κάποιων μετρικών.

Αριθμός Επεξεργαστών: **P(n) = p**

Όπου p είναι ο συνολικός αριθμός των επεξεργαστών που συνεργάζονται για να λύσουν το πρόβλημα P μεγέθους n.[4,8]

Σειριακός Χρόνος Εκτέλεσης: **T*(n)**

Δηλώνει τον χρόνο που χρειάζεται να λυθεί ένα πρόβλημα σειριακά. Επίσης εκφράζεται και η σειριακή πολυπλοκότητα του αλγορίθμου και πρέπει να αποδεικνύεται ότι κανένας άλλος σειριακός αλγόριθμος δεν λύνει το πρόβλημα πιο γρήγορα.[4,8]

Παράλληλος Χρόνος Εκτέλεσης: **Tp(n)**

Είναι ο συνολικός χρόνος που απαιτείται από p επεξεργαστές για να επιλύσουν ένα πρόβλημα μεγέθους n.[4,8]

Κόστος Παράλληλου Αλγορίθμου: **Cp(n) = Tp(n) * P(n)**

Το κόστος προκύπτει από το γινόμενο του πλήθους των επεξεργαστών και του παράλληλου χρόνου εκτέλεσης του αλγορίθμου.[4,8]

Επιτάχυνση: **Sp(n) = T*(n) / Tp(n)**

Ο λόγος του χρόνου εκτέλεσης του καλύτερου σειριακού αλγορίθμου που επιλύει κάποιο συγκεκριμένο πρόβλημα A δια το χρόνο εκτέλεσης του παράλληλου αλγορίθμου που επιλύει το πρόβλημα A μεγέθους n.[4,8]

Επιδιώκουμε να σχεδιάσουμε αλγορίθμους όπου Sp(n) ~ p αλλά στην πραγματικότητα υπάρχουν πολλοί παράγοντες που εισάγουν αναποτελεσματικότητα και ο κύριος λόγος στη συγκεκριμένη περίπτωση είναι η ανεπαρκής ταυτοχρονία, οι υπολογίστηκες καθυστερήσεις που προκύπτουν από τις καθυστερήσεις επικοινωνίας και το συντονισμό του συστήματος. Θεωρητικά τώρα αντιλαμβανόμαστε ότι αναπόφευκτα υπάρχει ένα κομμάτι που είναι σειριακό δηλαδή μια εργασία χρειάζεται το αποτέλεσμα μιας άλλης εργασίας, δηλαδή δεν είναι τελείως ανεξάρτητες. Έστω $0 \leq k \leq 1$ το κλάσμα αυτό. Τότε η μέγιστη δύναμη επιτάχυνσης που μπορεί να επιτευχθεί από οποιοδήποτε παράλληλο αλγόριθμο που χρησιμοποιεί p επεξεργαστές είναι $s \leq 1 / (k+(1-k)/p)$ σύμφωνα με τον νόμο του Amdahl.[8]

Απόδοση Κόστους: $\mathbf{e_p(n) = T^*(n) / C_p(n)}$

Ο λόγος του χρόνου εκτέλεσης του καλύτερου σειριακού αλγορίθμου που επιλύει κάποιο συγκεκριμένο πρόβλημα Α δια το κόστος του παράλληλου αλγορίθμου που λύει το συγκεκριμένο πρόβλημα Α.[4,8]

Αυτό το μέτρο προβάλει μια ένδειξη της αποτελεσματικής χρησιμοποίησης των p επεξεργαστών με βάση τον αλγόριθμο, δηλαδή πόσο καλά αξιοποιούμε τους επεξεργαστές κατά την εκτέλεση του παράλληλου αλγορίθμου. Ουσιαστικά μετρά αν κάθε επεξεργαστής κάνει ωφέλιμο έργο σχετικά με το συνολικό ποσοστό δουλειάς που απαιτείται από τον αλγόριθμο Α. Αν το αποτέλεσμα που παίρνουμε από την απόδοση κόστους είναι κοντά στο 1 για p επεξεργαστές τότε σημαίνει ότι ο αλγόριθμος Α τρέχει περίπου p φορές γρηγορότερα παρά από το να χρησιμοποιούσε ένα επεξεργαστή.

Όλα τα παραπάνω είναι μετρικές που μπορούμε να πάρουμε για αξιολόγηση της καταλληλότητας ενός παράλληλου αλγόριθμου. Πρέπει όμως να ορίσουμε πότε ένας παράλληλος αλγόριθμος είναι βέλτιστος και αποδεκτά μπορεί να χρησιμοποιείται αφού δεν μας επιβαρύνει με την εκτέλεση του.

Βέλτιστος Αλγόριθμος: $\mathbf{C(n) = O(T^*(n))}$

Ένας παράλληλος αλγόριθμος, σχεδιασμένος να επιλύει ένα πρόβλημα μεγέθους n, λέμε ότι είναι κόστους βέλτιστος αλγόριθμος για αυτό το πρόβλημα αν $C(n) = O(T^*(n))$. Δηλαδή το κόστος του παράλληλου αλγορίθμου είναι της τάξεως του καλύτερου σειριακού αλγόριθμου.

Αν κάποιος αλγόριθμος είναι βέλτιστος καθορίζεται όπως παρατηρούμε από το κόστος του που είναι το βασικό μέτρο σύγκρισης μεταξύ παράλληλων αλγορίθμων που επιλύουν το ίδιο πρόβλημα, ίδιου μεγέθους. Αν τώρα έχουν το ίδιο κόστος θα κοιτάξουμε ποιος αλγόριθμος επιλύει το πρόβλημα πιο γρήγορα, δηλαδή τον παράλληλο χρόνο που χρειάζεται να εκτελεστεί ο αλγόριθμος.[4,8]

Επιπλέον ένας αλγόριθμος είναι χρόνου βέλτιστος αν δεν μπορεί να λυθεί το πρόβλημα πιο γρήγορα από κάποιον άλλο αλγόριθμο, δηλαδή το $T^*(n)$ δεν μπορεί να βελτιωθεί, μειωθεί στη συγκεκριμένη περίπτωση.

Στοχεύουμε στη χρήση τόσου κόστους βέλτιστων αλγορίθμων αλλά και χρόνου βέλτιστων αλγορίθμων.[4]

2.7 Μοντέλα Παράλληλου Υπολογισμού

Το RAM (Random Access Machine)[4] έχει χρησιμοποιηθεί επιτυχώς για την εκτέλεση σειριακών αλγορίθμων, αλλά όπως αναφέρθηκε η μοντελοποίηση παράλληλων αλγορίθμων είναι συγκριτικά πιο ενδιαφέρουσα αφού έχουμε επιπλέον μια νέα διάσταση που είναι το σύνολο των αλληλένδετων μεταξύ τους επεξεργαστών. Πρέπει να εστιάσουμε σε αλγοριθμικά μοντέλα που μπορούν να περιγραφούν, αναλυθούν και υλοποιηθούν σε ένα γενικό μοντέλο. Ιδανικά ζητούμε το μοντέλο αυτό να ικανοποιεί δύο απαιτήσεις, απλότητα και δυνατότητα εκτέλεσης.

Η απλότητα καθορίζει ότι το μοντέλο πρέπει να είναι αρκετά απλό έτσι ώστε να έχουμε μεγαλύτερη ευχέρεια και δυνατότητα να περιγράφουμε παράλληλους αλγορίθμους πιο εύκολα. Επίσης ένα απλό μοντέλο θα μας διευκολύνει και στην ανάλυση του αλγορίθμου γύρω από μαθηματικά αποτελέσματα που μας ενδιαφέρουν όπως την ταχύτητα, επικοινωνία και διαχείριση μνήμης, Επιπλέον για να θεωρείται ένα μοντέλο απλό πρέπει αν είναι γενικά ανεξάρτητο, δηλαδή όχι συνδεδεμένο με κάποιο συγκεκριμένο τύπο αρχιτεκτονικής. Σημαντικό είναι να έχουμε στην διάθεση μας ανεξάρτητο μοντέλο από κάποιο υλικό(hardware) όσο το δυνατό περισσότερο.

Η δεύτερη απαίτηση που είναι σημαντική και λαμβάνουμε υπόψη μας είναι η δυνατότητα εκτέλεσης, δηλαδή οι παράλληλοι αλγόριθμοι αναπτύσσονται για μοντέλα που έχουν την δυνατότητα να εκτελούνται σε παράλληλους υπολογιστές. Αν δεν ισχύει αυτή η απαίτηση τότε δεν υπάρχει και λόγος σχεδιασμού παράλληλων αλγορίθμων αφού δεν μπορούν να χρησιμοποιηθούν.

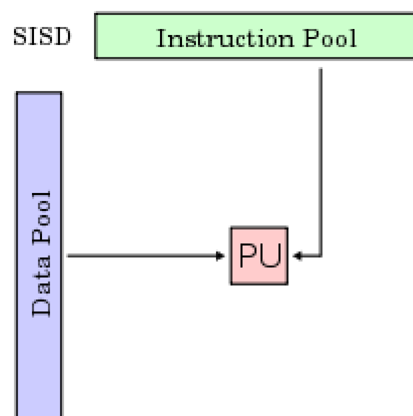
Η σχέση λογισμικού και υλικού είναι αναπόφευκτη και ένας σχεδιαστής παράλληλου αλγορίθμου θα πρέπει να έχει υπόψη του το συγκεκριμένο μοντέλο που θα υλοποιηθεί ο αλγόριθμος του. Για αυτό το λόγο αναφέρονται πιο κάτω τα μοντέλα που έχουμε στην διάθεση μας και αναπτύσσεται με περισσότερες λεπτομέρειες το μοντέλο που χρησιμοποιείται για υλοποίηση των αλγορίθμων που μελέτησα. Η κατηγοριοποίηση των μοντέλων έγινε το 1966 από τον Michael J. Flynn, τον αρχιτέκτονα των υπολογιστών όπως πολύ εύστοχα τον αποκάλεσαν.

Κάθε υπολογιστής λειτουργεί εκτελώντας εντολές πάνω σε δεδομένα. Από αυτό καταλαβαίνουμε ότι έχουμε στη διάθεση μας να διαχειριστούμε δύο παραμέτρους, τις εντολές και τα δεδομένα. Ένα ρεύμα εντολών (instructions) λέει στον υπολογιστή τι να κάνει σε κάθε βήμα, ενώ ένα ρεύμα δεδομένων (data) επηρεάζεται από το ρεύμα των εντολών και ανάλογα αλλάζουν ή παραμένουν οι ίδιες οι τιμές των δεδομένων μας. Η κατηγοριοποίηση γίνεται με βάση αυτές τις δύο παραμέτρους και οδηγούμαστε να έχουμε τέσσερις κατηγορίες.

Single Instruction Single Data (SISD)

Μέσα από το τίτλο του μοντέλου καταλαβαίνουμε ότι δεχόμαστε μόνο ένα ρεύμα εντολών, δηλαδή μόνο μια εντολή και δουλεύουμε σε ένα δεδομένο μόνο. Αλλιώς θα μπορούσαμε να πούμε ότι είναι η κατηγορία των σειριακών υπολογιστών με μόνο ένα επεξεργαστή όπου δουλεύει σε μόνο ένα ρεύμα εντολών κάθε φορά από την μνήμη και με βάση αυτού του ρεύματος τρέχει σε ένα ρεύμα δεδομένων. Εμείς δεν θα ασχοληθούμε καθόλου με αυτό το μοντέλο αφού αποτελεί κομμάτι του σειριακού υπολογιστή. Μπορούμε να δούμε Σχηματική Αναπαράσταση στο Σχήμα2.3

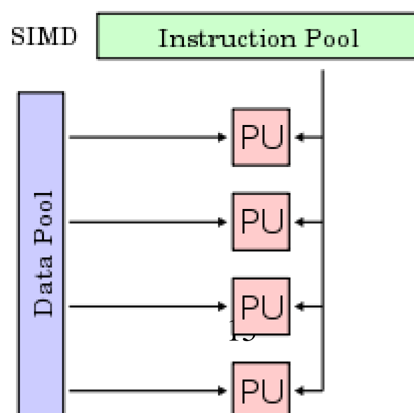
Σχήμα 2.3[11]



Single Instruction Multiple Data (SIMD)

Το μοντέλο αυτό σύμφωνα και με το όνομα του έχει το δικαίωμα να εκτελεί το ίδιο ρεύμα εντολών πάνω σε διαφορετικά δεδομένα. Σε αυτή την περίπτωση έχουμε στη διάθεση μας n πανομοιότυπους(ομογενής) επεξεργαστές, όπου ο καθένας έχει την δική του τοπική μνήμη άρα και μπορεί να έχει ότι θέλει αποθηκευμένο, ανεξάρτητα με το τι έχουν ή τι βλέπουν οι άλλοι επεξεργαστές, αυτό ουσιαστικά ορίζει το διαφορετικό ρεύμα δεδομένων(multiple data). Αν και οι επεξεργαστές μας έχουν διαφορετικά δεδομένα δεν έχουν όμως την ευχέρεια να κάνει ο καθένας αυθαίρετα όποια εντολή θέλει. Πρέπει να είναι συντονισμένοι-συγχρονισμένοι και να εκτελούν όλοι την ίδια εντολή ανά πάσα χρονική στιγμή, σε κάθε clock-cycle. Το μοντέλο αυτό είναι που θα μας απασχολήσει κατά την διάρκεια της Διπλωματικής Εργασίας αυτής. Σχηματική Αναπαράσταση του μοντέλου αυτού στο Σχήμα 2.4.

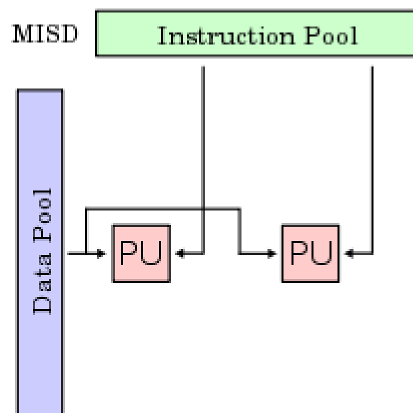
Σχήμα 2.4[12]



Multiple Instruction Single Data (MISD)

Αν ασχοληθούμε με το όνομα της κατηγορίας αυτής θα καταλάβουμε ότι αναφέρεται σε ένα ρεύμα με πολλαπλές εντολές που δουλεύουν σε ένα ρεύμα δεδομένων μονάχα. Έχουμε n ομογενής επεξεργαστές όπου ο καθένας έχει την δική του μονάδα ελέγχου, άρα αυτό μας δίνει το δικαίωμα ο κάθε επεξεργαστής να μπορεί να εκτελεί διαφορετική εντολή από κάποιον άλλο, έχουμε την απουσία του περιορισμού όπου πρέπει να εκτελούν όλοι την ίδια εντολή σε κάθε στιγμή(multiple instruction). Σε αυτό το μοντέλο πρέπει όμως οι επεξεργαστές να μοιράζονται τα δεδομένα που βρίσκονται στην κοινόχρηστη μνήμη και δεν έχει ο κάθε επεξεργαστής ξεχωριστή μνήμη. Για το μοντέλο αυτό μέχρις στιγμής δεν υπάρχει μηχανή που να το υποστηρίζει λόγω του ότι υπερκαλύπτεται από το μοντέλο MIMD και το μοντέλο SIMD είναι πιο προσιτό και πιο κατανοητό, άρα δεν θεωρείται χρήσιμο το MISD. Σχηματική Αναπαράσταση 2.5.

Σχήμα 2.5[13]

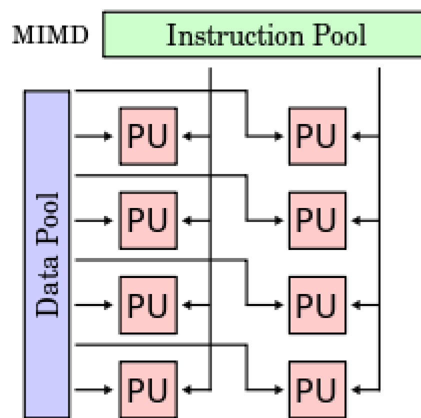


Multiple Instruction Multiple Data (MIMD)

Το μοντέλο αυτό είναι το πιο ισχυρό αλλά και γενικό μοντέλο και από τα τέσσερα(4) που υπάρχουν. Δέχεται πολλαπλά ρεύματα σε πολλαπλά δεδομένα. Πάλι έχουμε στην διάθεση μας n ομογενείς επεξεργαστές οι οποίοι δουλεύουν σε n διαφορετικά ρεύματα εντολών και n διαφορετικά ρεύματα δεδομένων. Ο κάθε επεξεργαστής έχει την δική του μνήμη και είναι ανεξάρτητη από την μνήμη κάποιου άλλου επεξεργαστή, άρα έχει τα δικά του δεδομένα που δεν επηρεάζονται από τις εντολές κάποιου άλλου επεξεργαστή αφού μόνο αυτός έχει δικαίωμα πρόσβασης. Επίσης οι επεξεργαστές έχουν το δικαίωμα να λειτουργούν αυθαίρετα χωρίς να πρέπει να είναι συγχρονισμένοι, μπορούν να εκτελούν οποιαδήποτε εντολή/πρόγραμμα θέλουν ανά πάσα στιγμή, λύνοντας διαφορετικά υπό-προβλήματα του ίδιου προβλήματος ή και διαφορετικού προβλήματος ίσως. Το μοντέλο αυτό φαίνεται πολύ προσαρμόσιμο αλλά αντιλαμβανόμαστε ότι είναι πολύ πολύπλοκο για να σχεδιαστεί και να αναλυθεί κάποιος παράλληλος αλγόριθμος πάνω σε αυτό, οποιοσδήποτε αλγόριθμος κτιστεί

με βάση αυτό το μοντέλο μπορεί να προσαρμοστεί και στα άλλα μοντέλα που αναφέρθηκαν πιο πάνω. Το μοντέλο αυτό μπορούμε να το δούμε στο σχήμα 2.6.[3,4]

Σχήμα 2.6[14]



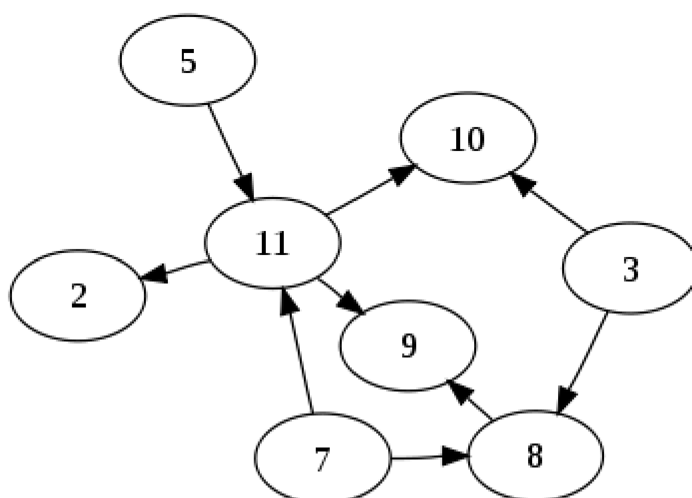
2.8 Μοντέλο Παράλληλου Υπολογισμού PRAM

Για αυτή την Διπλωματική Εργασία δουλέψαμε στην παράλληλη αρχιτεκτονική SIMD, δηλαδή ένα ρεύμα εντολών σε πολλά ρεύματα δεδομένων, από n επεξεργαστές που ο καθένας έχει την δική του μνήμη. Πιο συγκεκριμένα έγινε χρήση μιας κατηγορίας του μοντέλου αυτού, το PRAM (Parallel Random Access Machine)[4] όπου είναι μια μηχανή τυχαίας προσπέλαση που χρησιμοποιείται για παράλληλο υπολογισμό. Για το PRAM μοντέλο έχουμε p πανομοιότυπους, συγχρονισμένους επεξεργαστές που διαθέτουν όλοι τοπική μνήμη. Η επιλογή αυτού του μοντέλου γίνεται λόγω της πληθώρας πλεονεκτημάτων που μας προσφέρει. Αρχικά έχει ένα καλά ανεπτυγμένο τρόπο από τεχνικές και μεθόδους για να επιλύουν διάφορες κλάσεις υπολογιστικών προβλημάτων. Επίσης το PRAM μοντέλο αποφεύγει αλγοριθμικές λεπτομέρειες που περιλαμβάνουν συγχρονισμό και επικοινωνία και γι' αυτό επιτρέπει στους σχεδιαστές αλγορίθμων να συγκεντρωθούν σε άλλα θέματα του προβλήματος. Ένα τρίτο σημείο που μας κάνει να κατατασσόμαστε υπέρ του μοντέλου αυτού είναι ότι συλλαμβάνεται σαφής κατανομή των επεξεργαστών στη δουλεία που πρέπει να κάνουν έτσι δεν υπάρχει κάποια σύγχυση. Επιπλέον πλεονέκτημα είναι, ότι μπορεί το PRAM να αναλυθεί σε πιο γενικά επίπεδα αλλά και να προσαρμοστεί σε ποίο ιδικά.
Για την επικοινωνία μεταξύ των επεξεργαστών υπάρχουν τρείς διαφορετικοί τύποι που μπορούν να χρησιμοποιηθούν και αναφέρονται πιο κάτω.

DAGs (Directed Acyclic Graphs)

Κάθε δεδομένο παρουσιάζεται σαν ένας κόμβος που δεν έχει εισερχόμενες ακμές. Κάθε λειτουργία παρουσιάζεται σαν ένας κόμβος με εισερχόμενες ακμές από τους κόμβους που αντιπροσωπεύουν τα δεδομένα, όπου ο αριθμός των ακμών είναι τουλάχιστο δύο. Ένας

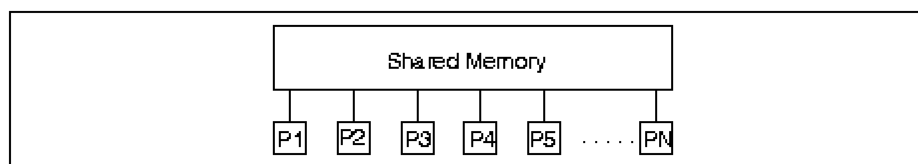
κόμβος χωρίς εξερχόμενες ακμές παρουσιάζετε ως μια έξοδος. Σχηματική Αναπαράσταση στο σχήμα 2.7[5]



Σχήμα 2.7[5]

Shared Memory Model

Η συγκεκριμένη υποκατηγορία είναι προέκταση του σειριακού μοντέλου. Πολλοί επεξεργαστές έχουν πρόσβαση σε single shared memory unit. Πιο συγκεκριμένα η κοινόχρηστη μνήμη περιλαμβάνει ένα αριθμό από επεξεργαστές όπου ο καθένας από αυτούς έχει την δική του τοπική μνήμη και μπορεί να τρέχει το δικό του πρόγραμμα αλλά επειδή εδώ έχουμε μόνο ένα ρεύμα εντολών αρκούμαστε στο να τρέχουν όλοι την ίδια εντολή. Όλοι αυτοί οι επεξεργαστές επικοινωνούν με το να αλλάζουν τα δεδομένα μέσω μιας κοινόχρηστης μνήμης. Κάθε επεξεργαστής είναι μοναδικά ορισμένος με ένα χαρακτηριστικό που ονομάζεται ταυτότητα(processor number or processor id) η οποία είναι διαθέσιμη τοπικά. Μπορούμε να δούμε στο Σχήμα 2.8.



Σχήμα 2.8[5]

Η κατηγορία κοινόχρηστης μνήμης πού δίνεται για επικοινωνία των επεξεργαστών χωρίζεται σε δύο άλλες υπό-κατηγορίες, την συγχρονισμένη και την ασύγχρονη. Στη συγχρονισμένη όλοι οι επεξεργαστές χωρίζονται κάτω από τον έλεγχο ενός κοινού ρολογιού που έχουν. Στην ασύγχρονη, κάθε επεξεργαστής δρα με διαφορετικό ρολόι. Στη συγκεκριμένη περίπτωση είναι ευθύνη του προγραμματιστή να θέσει σημεία συγχρονισμού όπου θεωρεί απαραίτητα

και άρα την ευθύνη έχει και πάλι ο προγραμματιστής για να διαβαστούν τα σωστά δεδομένα από τους επεξεργαστές.[4]

Network Model

Το δίκτυο επικοινωνίας ως μοντέλο καθορίζει ότι είναι ένας γράφος G=(N,E) όπου κάθε κόμβος i϶N αντιπροσωπεύει ένα επεξεργαστή και κάθε ακμή (i,j)Єe παρουσιάζει διπλής κατεύθυνσης επικοινωνία ανάμεσα στους επεξεργαστές i και j. Δηλαδή μπορεί να επικοινωνήσει ο i με τον j αλλά και αντίστροφα. Κάθε επεξεργαστής έχει την δική του τοπική μνήμη αλλά παρατηρείται η απουσία κοινόχρηστης μνήμης αφού επικοινωνούν με τις εντολές send() και receive().

Εμείς θα χρησιμοποιήσουμε στους αλγόριθμους μας κοινόχρηστη μνήμη. Γιατί έγινε όμως αυτή η επιλογή; Παρά την απλότητα του το DAGs μοντέλο εφαρμόζεται σε μια συγκεκριμένη κατηγορία προβλημάτων και υποφέρει από πολλές ελλείψεις, έτσι ακόμα και σε ένα απλό σχετικά αλγόριθμο με το μοντέλο αυτό θα ήταν δύσκολο και περίπλοκο να εφαρμοστεί και αναλυθεί. Επιπλέον, δεν έχει φυσικούς μηχανισμούς για να λάβει επικοινωνία μεταξύ των επεξεργαστών ή να λάβει κατανομή μνήμης και πρόσβαση στη μνήμη. Το δίκτυο επικοινωνίας μοιάζει να είναι λίγο καλύτερο για την επίλυση υπολογιστικού και επικοινωνιακού προβλήματος συγκριτικά με το DAGs αλλά το δίκτυο επικοινωνίας έχει δυο κύρια μειονεκτήματα και αυτό. Αρχικά είναι πιο περίπλοκο να βρεθούν και να αναλυθούν αλγόριθμοι σε αυτό το μοντέλο. Δεύτερον, το μοντέλο αυτό εξαρτάται σε μεγάλο βαθμό στην τοπολογία που είναι υπό περιορισμένο. Διαφορετικές τοπολογίες ίσως απαιτούν εντελώς διαφορετικούς αλγορίθμους για να λύσουν το ίδιο πρόβλημα. Αυτοί οι λόγοι καθορίζουν την συγχρονισμένη κοινόχρηστη μνήμη ως το πιο ικανό μοντέλο, όπου είναι και αυτό με το οποίο θα ασχοληθούμε για τους δικούς μας αλγορίθμους. Αξίζει επίσης να σημειώσουμε ότι ο χρόνος πρόσβασης σε μια κοινόχρηστη μήνη είναι Θ(1) και ότι τα δεδομένα εισόδου, τα ενδιάμεσα αποτελέσματα και τα δεδομένα εξόδου αποθηκεύονται στην κοινόχρηστη μνήμη μέσω κοινόχρηστων μεταβλητών.

Το μοντέλο κοινόχρηστης μνήμης του PRAM αυτό μας δίνει την δυνατότητα να κάνουμε ομαδοποίηση με βάση ακόμα ένα κριτήριο, την ταυτόχρονη πρόσβαση στην μνήμη, όπου πρόσβαση εννοούμε είτε γράψιμό, είτε διάβασμα δεδομένων. Προκύπτουν τέσσερεις υπό-κατηγορίες οι οποίες είναι οι ακόλουθες.[4]

Exclusive Read Exclusive Write (EREW)

Σε αυτή την κατηγορία συμπεριλαμβάνονται οι αλγόριθμοι που δεν υποστηρίζουν την ταυτόχρονη ανάγνωση ή γραφή από/στην ίδια κυψελίδα κοινόχρηστης μνήμης από

περισσότερους από ένα επεξεργαστές. Δηλαδή σε αυτή την ομάδα ανήκουν τα μοντέλα όπου μόνος ένας επεξεργαστής μπορεί να διαβάσει/γράψει σε μια κυψελίδα ταυτόχρονα, ανά πάσα στιγμή.

Concurrent Read Exclusive Write (CREW)

Στην ομάδα αυτή ανήκουν οι αλγόριθμοι που επιτρέπουν την ταυτόχρονη ανάγνωση, δηλαδή πέραν του ενός επεξεργαστή να διαβάζει από μια κυψελίδα μνήμης την ίδια στιγμή, αλλά μονάχα ένας επεξεργαστής να γράφει σε μια μονάδα χρόνου.

Exclusive Read Concurrent Write (ERCE)

Αυτός ο τύπος αλγορίθμων επιτρέπει την ταυτόχρονη γραφή στην ίδια κυψελίδα μνήμης από όσους επεξεργαστές θέλουν, δεν υπάρχει δηλαδή κάποιος περιορισμός όσον αφορά το γράψιμο, αλλά υπάρχει περιορισμός στην ταυτόχρονη πρόσβαση για διάβασμα από κάποια κυψελίδα μνήμης, αφού μόνο ένας επεξεργαστής μπορεί να διαβάσει ανά πάσα στιγμή από κάποια θέση της κοινόχρηστης μνήμης. Αξίζει να σημειωθεί ότι το μοντέλο αυτό υπάρχει μόνο σε θεωρητικό επίπεδο αφού κανένας αλγόριθμος δεν βρίσκει εφαρμογή πάνω του.

Concurrent Read Concurrent Write (CRCW)

Η κατηγορία αυτή είναι και η πιο ελεύθερη κατηγορία αφού δεν υπάρχει απολύτως κανένας περιορισμός, ούτε για την ανάγνωση αλλά ούτε για την γραφή στην κοινόχρηστη μνήμη. Μπορεί όποιος επεξεργαστής θέλει σε οποιοδήποτε σημείο της εκτέλεσης του αλγορίθμου να διαβάσει ή να γράψει από/στην κοινόχρηστη μνήμη ανεξάρτητα με το τι κάνει κάποιος άλλος επεξεργαστής.

Η τόση μεγάλη ελευθερία όσον αφορά το θέμα της ταυτόχρονης γραφής δημιουργεί κάποια επιπλέον προβλήματα όπου πρέπει να είμαστε ιδιαίτερα προσεκτικοί, για να μπορούμε να τα διαχειριστούμε σε αντίθεση με την ταυτόχρονη ανάγνωση που δεν αντιμετωπίζουμε κάποιο πρόβλημα γιατί ο κάθε επεξεργαστής βλέπει την τιμή της κοινόχρηστης κυψελίδας και την αποθηκεύει σε τοπική του μεταβλητή. Το πρόβλημα δημιουργείται όταν πέραν του ενός επεξεργαστή θέλουν να γράψουν στην ίδια κυψελίδα μνήμης την ίδια στιγμή, όπου πρέπει να καθοριστεί ποιος θα επικρατήσει και πια η τιμή θα γραφτεί τελικά στην συγκεκριμένη θέση, αφού μονάχα μια μπορεί να είναι γραμμένη. Έτσι έχουμε την δημιουργία τριών(3) νέων κατηγοριών που βρίσκονται κάτω από την ομπρέλα του ταυτόχρονης εγγραφής.

- Common CRCW

  Το μοντέλο αυτό βάζει ένα επιπλέον περιορισμό, όλοι οι επεξεργαστές που προσπαθούν να γράψουν την ίδια στιγμή στην ίδια θέση να γράφουν την ίδια τιμή, δηλαδή για παράδειγμα να γράφουν όλοι την τιμή 1.

- Arbitrary CRCW

  Σε αυτή την έκδοση του μοντέλου δίνεται μεγάλη σημασία στην τυχαιότητα. Ένας τυχαίος από τους επεξεργαστές που προσπαθούν την ίδια στιγμή να γράψουν στην

ίδια κυψελίδα θα επικρατήσει, δεν υπάρχει κριτήριο ποιος είναι αυτός, αλλά θα μπορούσαμε να πούμε ο πιο τυχερός που είναι αυτός που θα αποθηκεύσει την τιμή του.

- Priority CRCW

  Σε πιο πάνω αναφορά μου επισήμανα ότι ο κάθε επεξεργαστής έχει μια μοναδική ταυτότητα. Σε αυτό το μοντέλο η ταυτότητα του επεξεργαστή όπου είναι και μοναδική, είναι σημαντική και την λαμβάνουμε υπόψη μας αφού είναι αυτή που μας καθορίζει ποιος επεξεργαστής έχει προτεραιότητα όταν υπάρχει ταυτόχρονο γράψιμο σε μια κυψελίδα μνήμης. Η μεγαλύτερη προτεραιότητα καθορίζεται ανάλογα, ίσως να έχει μεγαλύτερη προτεραιότητα ο επεξεργαστής με την μεγαλύτερη ταυτότητα, άρα η σειρά θα ήταν $P_n, P_{n-1},\ldots,P_2,P_1$, ενώ αν προτεραιότητα είχε ο επεξεργαστής με την μικρότερη ταυτότητα η προτεραιότητα θα ήταν $P_1,P_2,\ldots,P_{n-1},P_n$.

  Είναι εμφανές ότι δημιουργούνται πολλές κατηγορίες με τους περιορισμούς που αναγκαστικά έχουμε και έτσι δημιουργείται και η ανάγκη να τις ιεραρχήσουμε ανάλογα με την δύναμη τους .

EREW→CREW→Common CRCW→Arbitrary CRCW→Priority CRCW

Weak(Αδύνατο) Strong(Δυνατό)

Το EREW μοντέλο είναι το πιο αδύνατο μοντέλο αλλά και το πιο περιοριστικό, ενώ το Priority CRCW είναι το πιο δυνατό αλλά και πιο ελαστικό συνάμα. Ένας αλγόριθμος που είναι σχεδιασμένος σε ένα πιο αδύνατο μοντέλο μπορεί να προσαρμοστεί και να εκτελεστεί σε ένα πιο δυνατό μοντέλο χωρίς κανένα πρόβλημα, μπορεί να γίνει και το αντίθετο αλλά θα χρειαστεί μεγάλο κόστος στο χρόνο και στον αριθμό των επεξεργαστών. Επιδιώκουμε να κτίζουμε αλγορίθμους σε αδύνατα μοντέλα έτσι ώστε να μπορούν να χρησιμοποιηθούν σε γενική κλίμακα ανεξαρτήτου μοντέλου.

Εν κατακλείδι θα ήθελα να αναφέρω ότι στην συγκεκριμένη Διπλωματική Εργασία θα μας απασχολήσει το SIMD, κατηγορία PRAM, κοινόχρηστη μνήμη και πιο συγκεκριμένα Arbitrary CRCW.[4]

Κεφάλαιο 3

Πλατφόρμα XMT

3.1 Πλατφόρμα XMT 22

3.2 Γλώσσα Προγραμματισμού XMTC 25

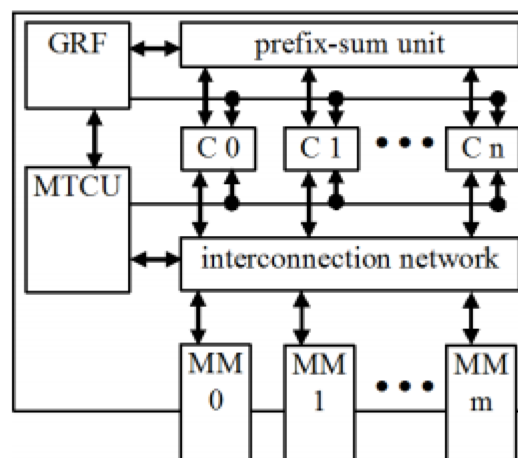
3.3 Περιορισμοί 34

3.1 Πλατφόρμα XMT

Η πλατφόρμα XMT όπως αναφέρθηκε αρκετές φορές είναι η πλατφόρμα όπου ασχολήθηκα για την υλοποίηση των παράλληλων αλγορίθμων. Η ιδέα της πλατφόρμας αυτή εφευρέθηκε από τον καθηγητή Uzi Vishkin το 1998 και μετά από πολύχρονη έρευνα και προσπάθεια ο καθηγητής και η ομάδα του κατάφεραν το 2007 να παρουσιάσουν την υλοποίηση ενός PRAM-On-Chip 64-Core 75MHz FPGA. Μια προσπάθεια που συνεχίζεται από την ίδια ομάδα και με συνεργασίες με άλλα πανεπιστήμια από ολόκληρο τον κόσμο για περαιτέρω βελτίωση και ανάπτυξη της πλατφόρμας αυτής έτσι ώστε να έχουμε περισσότερες δυνατότητες. Η πλατφόρμα XMT υλοποιεί το μοντέλο παράλληλου υπολογισμού Arbitary CRCW SIMD.

Η πλατφόρμα αυτή είναι η μοναδική που υπάρχει η οποία προσημειώνει το αλγοριθμικό μοντέλο PRAM και κατ' επέκταση τους αλγορίθμους που είναι σχεδιασμένοι για αυτό το μοντέλο. Όπως αναφέρθηκε η υλοποίησης της πλατφόρμας XMT οδηγήθηκε από το όραμα PRAM-On-Chip όπου επιδίωκε στη δημιουργία ενός εύκολου και εύχρηστου μοντέλου που θα μπορούσε να προγραμματίζει παράλληλα ενώ θα είχαμε στην διάθεση μας ένα τσιπ που θα αποτελείτο από χιλιάδες επεξεργαστές. Η δημιουργία μιας τέτοιας πλατφόρμας ήταν κάτι πολύ ενδιαφέρον και πρόκληση για τους ερευνητές γιατί ήταν φανερό πως δεν ήταν κάτι εύκολο και απλό αφού θα έπρεπε να βρουν τον τρόπο να μετατρέπουν ένα αλγόριθμο που είχε την σκεπτική PRAM σε έμπρακτο αλγόριθμο XMT. Πρέπει να αναφερθεί ότι στους αλγορίθμους που είναι προγραμματισμένοι για PRAM θεωρούμαι αυτονόητο ότι ακολουθούν την δομή "Work-Depth". Η δομή αυτή ουσιαστικά καθορίζει ότι οι επεξεργαστές είναι συγχρονισμένοι και λειτουργούν όλοι με τον ίδιο ρυθμό ανά γύρους, κάτι

που δεν ισχύει έμπρακτα αφού ο καθένας έχει την δική του ταχύτητα. Τελικά, αυτό ήταν το στοίχημα που θα έπρεπε να κερδίσουν οι ερευνητές και να καταφέρουν για την πλατφόρμα XMT έτσι ώστε να λειτουργεί όπως το PRAM μοντέλο, δηλαδή να συγχρονίσουν τους επεξεργαστές μεταξύ τους. Η μεθοδολογία Work-Depth ήταν μια έννοια που ορίστηκε από τους Shiloach και Vishkin το 1982. Η χρήση αυτής της μεθόδου καθιστά εύκολα τον εντοπισμό της ορθότητας και της ανάλυσης του αλγορίθμου αφού γίνονται με βάση τον συνολικό αριθμό των λειτουργιών/ενεργειών (work) και τον αριθμό των γύρων/επαναλήψεων (depth). Η γλώσσα που αναμένεται να χρησιμοποιηθεί από τον προγραμματιστή είναι η XMTC, θέμα που αναλύεται σε μεταγενέστερο επίπεδο και καθορίζεται πως πετυχαίνουμε την μεθοδολογία work-depth. Θέλοντας να αναφέρουμε πιο έμπρακτα στοιχεία για την πλατφόρμα XMT, πρέπει να πούμε ότι είναι μια μηχανή μια 64 επεξεργαστές και βασισμένη στη τεχνολογία FPGA. Είναι υλοποιημένη με τρία FPGA τσιπς και μία κάρτα MTCU(Master Thread Control Unit),παράλληλα και πολλαπλά TCU, ένα διαχειριστή μνήμης(MC), μια prefix sum unit και καθολικούς καταχωρητές. Απεικόνιση των κομματιών που απαρτίζουν ένα XMT φαίνεται στην εικόνα πιο κάτω. Σχηματική Αναπαράσταση με το Σχήμα 3.1.



Σχήμα 3.1

Όσο περνά ο καιρός οι ερευνητές που ασχολούνται εντατικά με την παράλληλη επεξεργασία, κάνουν προσπάθειες για αναβάθμισης της πλατφόρμας έτσι ώστε να γίνει όσο το δυνατό γρηγορότερη και πιο ευέλικτη, δηλαδή να μπορούν οι χρήστες να προσομοιώνουν περισσότερα προβλήματα και αλγορίθμους στην πλατφόρμα. Στόχος τους είναι να την αναπτύξουν έτσι ώστε να μπορεί να υποστηρίζει περισσότερες εντολές και όσο το δυνατό πιο γρήγορα με μικρότερο κόστος. Η καινούρια πλατφόρμα του PRAM On-Chip που κατάφερε να υλοποιηθεί αποτελείται από 128 επεξεργαστές δηλαδή είναι πλατφόρμα PRAM-On-Chip 128-Core. Αποτελείται από τρία τσιπς FPGA. Τα δύο από αυτά είναι Virtex-4 LX200 και το ένα που απομένει Virtex-4 FX100. Όλα τα άλλα στοιχεί παραμένουν τα ίδια, δηλαδή

χρησιμοποιούνται τα ίδια χαρακτηριστικά. Η επέκταση που έγινε ήταν μια πολύ σημαντική βελτίωση αφού πλέον έχουμε τα αποτελέσματα που θέλουμε σε πολύ πιο μικρό χρονικό διάστημα απ' ότι τα είχαμε με την παλιά πλατφόρμα. Ελπίζουμε σε ακόμη πιο ανεπτυγμένες πλατφόρμες στο προσεχώς μέλλον που θα μας δίνουν μεγαλύτερη ευχέρεια.

Όπως αναφέρθηκα στον παράλληλο υπολογισμό και στους παράλληλους αλγορίθμους γίνεται χρήση πολλαπλών νημάτων μέσα σε παράλληλα κομμάτια υπολογισμού, δηλαδή δουλεύουν μαζί πολλοί επεξεργαστές. Ο κάθε επεξεργαστής όμως μέσα στα παράλληλα κομμάτια έχει την δική του ταχύτητα, δεν εκτελούν όλοι οι επεξεργαστές ακριβώς την ίδια στιγμή κάποια εντολή, είναι ασύγχρονοι όμως. Οι επεξεργαστές όμως έχουν το δικαίωμα πρόσβασης σε κοινόχρηστες μεταβλητές, μεταβλητές που όλοι οι επεξεργαστές έχουν πρόσβαση επάνω τους. Ο συνδυασμός την κοινόχρηστης πρόσβασης με τον μη συγχρονισμό των επεξεργαστών δημιουργούν ένα πολύ σημαντικό πρόβλημα. Το πρόβλημα είναι ότι εάν δυο επεξεργαστές ή περισσότεροι προσπαθούν να έχουν πρόσβαση στην ίδια κοινόχρηστη μεταβλητή και ο ένας είναι πιο γρήγορος από τον άλλο τότε ο πιο αργός ίσως να διαβάσει διαφορετική τιμή από αυτή που πρέπει στη μεταβλητή ή μπορεί να συμβεί και το αντιθέτω δηλαδή να διαβάσει ο πιο γρήγορος μια τιμή που δεν πρέπει γιατί δεν πρόλαβε να γράψει ο άλλος που είναι πιο αργός. Δηλαδή οι ταχύτητες των επεξεργαστών δημιουργούν πρόβλημα και ίσως να έχουμε λανθασμένο αποτέλεσμα με τον τερματισμό του προγράμματος. Για το πρόβλημα αυτό βρέθηκαν κάποιες λύσεις έτσι ώστε να μπορεί να αντιμετωπίζεται. Η μια λύση που δίνεται να χρησιμοποιείται περισσότερο είναι η χρήση διπλών δομών, δηλαδή για παράδειγμα διπλό πίνακα και γράφεται και διαβάζεται ανά γύρο σε μια από τις στήλες του πίνακα, με αυτό τον τρόπο είμαστε σίγουροι ότι διαβάζεται η σωστή τιμή κάθε φορά.

Οι ερευνητές που κατάφεραν να υλοποιήσουν την πλατφόρμα XMT είχαν στο πίσω μέρος του μυαλού τους το αλγοριθμικό μοντέλο PRAM, αφού είναι και η μόνη πλατφόρμα που το υποστηρίζει. Ναι μεν είχαν υπόψη τους το αλγοριθμικό μοντέλο αυτό αλλά σίγουρα δεν μπόρεσαν να το προσομοιώσουν αυτό κάθε αυτό. Θα ήθελα να τονίσουμε τις διαφορές που έχουν οι αλγόριθμοι στην πλατφόρμα και στο θεωρητικό μοντέλο PRAM. Η μια διαφορά είναι ότι όπως ανέφερα πιο πριν στην θεωρία οι επεξεργαστές θεωρούνται ότι εκτελούν τις εντολές τους με την ίδια ταχύτητα κάτι που δεν συμβαίνει έμπρακτα και χρειαζόμαστε μεθόδους για να επιλύουμε αυτό το πρόβλημα. Μια ακόμη διαφορά που έχουμε είναι όσον αφορά τον χρόνο που υπολογίζουμε. Όταν υπολογίζουμε την πολυπλοκότητα στους θεωρητικούς αλγορίθμους δεν υπολογίζουμε τον χρόνο που χρειάζεται για αρχικοποίηση κάποιας μεταβλητής ή για κάποιες άλλες προσβάσεις ενώ σε έμπρακτους αλγορίθμους οι προσβάσεις αυτές παίρνουν χρόνο και υπολογίζεται. Επίσης ακόμη μια διαφορά που έχουν

είναι ότι στους θεωρητικούς αλγορίθμους χρησιμοποιούνται πολλές εντολές που δεν μπορούν να χρησιμοποιηθούν στην πλατφόρμα XMT και χρειάζεται να βρεθούν άλλες μέθοδοι πιο πολύπλοκοι ώστε να αντικατασταθούν αυτές οι εντολές. Μια τελευταία διαφορά που εντοπίζεται είναι ότι στους θεωρητικούς αλγορίθμους δεν υπολογίζεται ο χρόνος που χρειάζεται για συγχρονισμό σε αντίθεση όμως με την πλατφόρμα XMT που μετρά και προστίθεται ο χρόνος που χρειάζεται για να γίνει ο συγχρονισμός των επεξεργαστών και αυτό συμβαίνει γιατί οι παράλληλοι αλγόριθμοι στο PRAM θεωρείται ότι γίνεται χρήση συγχρονισμένων επεξεργαστών. Ο χρόνος συγχρονισμού που χρειάζεται η πλατφόρμα XMT είναι σταθερός και θέλοντας να μιλήσουμε πιο συγκεκριμένα μπορούμε να πούμε ότι χρειάζεται 0,2 μικροδευτερόλεπτα για να γίνει, δηλαδή ότι για να επικοινωνήσουν και να εντοπίσουν ότι είναι εντάξει και τερμάτισαν όλοι την δουλειά που είχαν αναλάβει.[15]

Γενικά η πλατφόρμα XMT αποτελείται από XMTC language, XMTC compiler, XMT simulator και XMT FPGA. Τα προγράμματα που υπάρχουν στη γλώσσα XMTC μπορούν να εκτελεστούν είτε απευθείας στη μηχανή που είναι βασισμένη στο FPGA, είτε σε ένα ¨cycle accurate¨ προσομοιωτή, τον XMTC simulator δηλαδή.[7]

3.2 Γλώσσα Προγραμματισμού XMTC

Η γλώσσα προγραμματισμού XMTC της πλατφόρμας XMT είναι το μέσο μας για να μπορούμε να προγραμματίζουμε τους παράλληλους αλγόριθμους που έχουμε στον μοντέλο PRAM. Είναι μια πρόσφατη γλώσσα προγραμματισμού που έχει όμως την βάση της σε μια πολυχρησιμοποιημένη και πασίγνωστη γλώσσα προγραμματισμού, την C. Ουσιαστικά η XMTC αποτελεί προέκταση της γλώσσας C, έτσι για την εκμάθηση της δεν θα έπρεπε να ξεκινήσουμε από την αρχή, αφού προϋπήρχαν ήδη οι βασικές γνώσεις στην γλώσσα, αλλά θα πρέπει να μελετήσουμε τυχόν προεκτάσεις και περιορισμούς που ίσως εμφανίζονται. Με λίγα λόγια αρχικά θα μπορούσαμε να πούμε ότι προσθέτονται δύο(2) επιπλέον εντολές έτσι ώστε να πετυχαίνουμε τον παράλληλο υπολογισμό, εντολές που θα αναφερθούν με μεγαλύτερη λεπτομέρεια σε μεταγενέστερο επίπεδο αυτού του κεφαλαίου. Η τόση ομοιότητα των δύο προγραμματιστικών γλωσσών μας επιτρέπει ένα πρόγραμμα που είναι γραμμένο στη γλώσσα C να μπορούμε να το τρέξουμε σε προσομοιωτή της γλώσσας XMTC, αλλά όχι το αντίθετο λόγω των επιπλέων εντολών που προστέθηκαν. Η γλώσσα XMTC χωρίζεται σε δυο κύρια κομμάτια, το σειριακό κομμάτι που είναι όπως το γνωρίζουμε χωρίς καμία αλλαγή, αλλά και

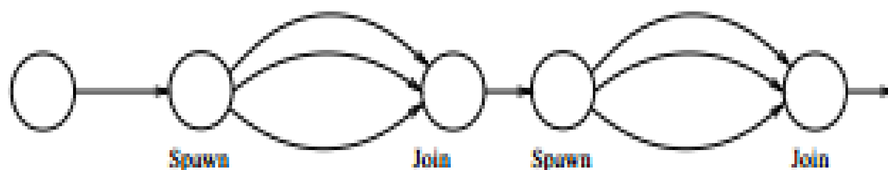
το παράλληλο κομμάτι που είναι το καινούριο μέρος που πρέπει να μελετήσουμε. Το παράλληλο μέρος εισάγεται με την εντολή *spawn* η οποία είναι η μια από τις δύο νέες εντολές. Δυστυχώς η γλώσσα αυτή είναι υπό κατασκευή και χρειάζεται πολλά βήματα ακόμη για να είναι πραγματικά εύχρηστη και να μπορεί να χρησιμοποιείται σε γενικό επίπεδο παραλληλισμού. Η ερευνητική ομάδα που ασχολείται με αυτό το θέμα προσπαθεί για να καλύπτονται περισσότερες προγραμματιστικές ανάγκες. Περιορισμούς και προβλήματα προκύπτουν και αναλύονται κατά την διάρκεια της Διπλωματικής Εργασίας.[6]

3.3.1 Main()

Όπως και στον σειριακό προγραμματισμό είναι απαραίτητη η κλήση της συνάρτησης main(), όπου καθορίζεται και η αρχή του προγράμματος μας, είτε αυτό είναι σειριακό, είτε παράλληλο. Έξω από την συνάρτηση main() μπορούμε μόνο να ορίσουμε κάποιες μεταβλητές και συναρτήσεις, που θα τους δοθεί περισσότερη σημασία σε πιο κάτω υπό-κεφάλαιο. Είναι απαραίτητη η χρήση της συνάρτησης αυτής για να ορίζεται το πρόγραμμα μας. Μέσα στη συνάρτηση αυτή ορίζουμε ότι θέλουμε, αρκεί πάντα να είναι στα επιτρεπτά πλαίσια της γλώσσας, δηλαδή αυτά που δέχεται και υποστηρίζει η XMTC.

3.3.2 Spawn()

Η μια καινούρια εντολή που προστέθηκε στην γλώσσα C είναι η εντολή spawn(), που είναι και η εντολή που καθορίζει το παράλληλο κομμάτι του αλγορίθμου και το ξεχωρίζει από το σειριακό. Σε σύντομη αναφορά μου ποιο πάνω καθόρισα ότι ένας παράλληλος αλγόριθμος εναλλάσσεται ανάμεσα σε σειριακά και παράλληλα κομμάτια/λειτουργίες. Η εντολή spawn() δηλώνει ότι σταματά η σειριακή λειτουργία εκείνη την χρονική στιγμή και τα πολλαπλά νήματα/επεξεργαστές ξεκινούν να δουλεύουν παράλληλα για να λύσουν το υπό-πρόβλημα που βρίσκεται μέσα στο μπλοκ της εντολής spawn. Όταν τερματίσει η εντολή spawn(), και αυτό γίνεται με την εντολή *join* τότε επανερχόμαστε και πάλι στο σειριακό κομμάτι του αλγορίθμου. Στην πιο κάτω εικόνα φαίνεται πως σειριακά και παράλληλα κομμάτια μπορούν να εναλλάσσονται κατά την διάρκεια που τρέχει κάποιο πρόγραμμα.



Κάθε επεξεργαστής μέσα στο παράλληλο μπλοκ έχει τους δικούς του ρυθμούς και είναι οι ίδιοι μέχρι το τέλος του παράλληλου μπλοκ, επίσης οι επεξεργαστές δεν ακολουθούν όλοι τους ίδιους ρυθμούς, αφού έχουν διαφορετικές ταχύτητες. Ο κάθε επεξεργαστής φροντίζει να εκτελέσει όλες τις εντολές που βρίσκεται μέσα στο μπλοκ. Όταν όλοι οι επεξεργαστές τελειώσουν όλες τις εντολές που έχουν να κάνουν συγχρονίζονται στο τέλος του μπλοκ και προχωρούν μαζί. Ουσιαστικά η εντολή αυτή εκτός από το να ορίζει το παράλληλο κομμάτι έχει και την ευθύνη να συγχρονίζει όλους τους επεξεργαστές, δηλαδή να κάνει την δουλειά του work-depth. Η εντολή αυτή ορίζεται ως

```
spawn(start thread, end thread){
        parallel section;
}
```

Όπως αναφέραμε οι επεξεργαστές ξεχωρίζουν με την μοναδική τους ταυτότητα όπου στη συγκεκριμένη περίπτωση είναι ακέραιοι αριθμοί έτσι ώστε να μπορούμε να τους χειριζόμαστε με μεγαλύτερη ευκολία. Άρα *start thread* και *end thread* είναι ακέραιες μεταβλητές που μας ορίζουν ποιοι επεξεργαστές θα εργαστούν παράλληλα για το μπλοκ που ακολουθεί. Ο αριθμός των επεξεργαστών που θα δουλεύουν παράλληλα ορίζεται από την πράξη (end thread-start thread)+1. Με αυτή την δήλωση σημαίνει ότι θα εργαστούν παράλληλα οι επεξεργαστές όπου ο μοναδικός αριθμός ταυτότητας τους είναι ανάμεσα στο start thread και end thread. Εργάζονται δηλαδή start thread ≤ id ≤ end thread. Για παράδειγμα αν είχαμε spawn(1,20) σημαίνει ότι θα εργαστούν 20 επεξεργαστές παράλληλα, αυτοί με ταυτότητες από 1 μέχρι 20. Είναι ευθύνη του προγραμματιστή να συντονίσει τους επεξεργαστές του και να διαχειριστεί την μνήμη του.

3.3.3 Single Spawn

Η δομή Single Spawn ορίζεται στον κώδικα μας ως **sspawn**() και βρίσκεται ορισμένη μέσα σε μια δομή/μπλοκ spawn(). Παράδειγμα από κομμάτι του κώδικα που περιέχει μια τέτοια δομή είναι φαίνεται στο πιο κάτω κομμάτι κώδικα:

```
spawn(start thread, end thread){
        int child_id;
        parallel code;
        sspawn(child_id){
                initialization block;
                code for initialization the child thread;
        }
}
```

Με την χρήση αυτής της δομής, ο αρχικός επεξεργαστής, ας τον ονομάσουμε πατέρα, παράγει/δημιουργεί ακόμα έναν νήμα/επεξεργαστή που το ονομάζουμε παιδί. Ο πατέρας εκτελεί τον κώδικα από το σημείο initialization block, δηλαδή το κομμάτι που είναι μέσα στο sspawn μπλοκ και το παιδί νήμα ξεκινά να εκτελεί εντολές από την πρώτη γραμμή του μπλοκ που είναι ο πατέρας του, δηλαδή ξεκινά από την αρχή. Αυτή η κατάσταση αρχίζει έναν αγώνα από το παιδί, όπου το παιδί μπορεί να ξεκινήσει να εκτελεί τον κώδικα που του ανατέθηκε, ενώ μπορεί ο πατέρας του να μην έχει φτάσει να αρχικοποιήσει το παιδί του, έτσι ώστε να πάρει τις σωστές τιμές και αυτό μπορεί να γίνει γιατί έχουν διαφορετικές ταχύτητες και ίσως το παιδί να είναι πιο γρήγορο από τον πατέρα του. Είναι ευθύνη του προγραμματιστή να συγχρονίσει τους επεξεργαστές όταν χρησιμοποιήσει αυτή την δομή έτσι ώστε να μην αντιμετωπίζει προβλήματα και μια λύση είναι η χρήση της συνάρτησης psm() που εξηγείται παρακάτω. Η νέα τιμή που δίνεται στο initialization block αντιγράφεται στην τοπική μεταβλητή child_id και είναι ένας ακέραιος αριθμός. Ο αριθμός αυτός είναι ο ακριβώς επόμενος ελεύθερος αριθμός, δηλαδή ο αριθμός που δεν ανήκει σε κάποιον άλλο επεξεργαστή, για παράδειγμα αν χρησιμοποιώ τους επεξεργαστές 1 μέχρι 9, τότε το child_id θα πάρει την τιμή 10. Αφού ο πατέρας αρχικοποιήσει το παιδί του τότε συνεχίζει να εκτελεί τον κώδικα του, από εκεί που έμεινε κανονικά. Το παιδί νήμα όμως παραβλέπει το κομμάτι του sspawn και δεν δημιουργεί δικό του παιδί.

3.3.4 Βιβλιοθήκες

Οι βιβλιοθήκες είναι επίσης ένα σημαντικό κομμάτι γιατί χωρίς την χρήση της βιβλιοθήκης <xmtc.h> στον κώδικα μας δεν μπορεί να μεταγλωττιστεί ο κώδικας μας, έτσι είναι απαραίτητο να εισάγουμε στην αρχή του κώδικα μας την εντολή αυτή. Με την εντολή αυτή ουσιαστικά εισάγονται όλες οι επιπλέον εντολές και πληροφορίες που χρειαζόμαστε για την παράλληλη μεταγλώττιση. Επίσης εισάγεται αυτόματα η εντολή για τύπωμα και δεν πρέπει να ανησυχεί ο προγραμματιστής. Επίσης επιτρέπεται η εισαγωγή βιβλιοθηκών από τον προγραμματιστή όπου έχει αποθηκευμένες τις δικές του μεταβλητές σε κάποιο αρχείο και αυτό γίνεται με το #include. Για παράδειγμα θέλω να ενσωματώσω το αρχείο με τίτλο variables, τότε θα εισάγω στον πάνω μέρος του κώδικα μου #include “variables.h”.

3.3.5 Μεταβλητές

Χωρίζουμε τις μεταβλητές σε δυο μεγάλες κατηγορίες όπως και στην γλώσσα C, τις καθολικές και τοπικές μεταβλητές. Η μια κατηγορία είναι οι καθολικές μεταβλητές οι οποίες έχουν πρόσβαση όλοι οι επεξεργαστές από οποιοδήποτε σημείο του κώδικα μας, δηλαδή οι μεταβλητές αυτές μοιράζονται σε όλους τους επεξεργαστές. Η άλλη κατηγορία που

μπορούμε να έχουμε είναι οι τοπικές μεταβλητές, που για τον παράλληλο υπολογισμό χωρίζονται σε δύο(2) υπό-κατηγορίες. Η μια υπό-κατηγορία είναι αυτή όπου ορίζεται και πάλι όπως στην C, δηλαδή το εύρος της μεταβλητής είναι μονάχα μέσα στο μπλοκ που ορίζεται. Για παράδειγμα αν ορίζεται μέσα σε μια συνάρτηση τότε μπορώ να την χρησιμοποιώ, δηλαδή να γράφω ή να διαβάζω από/σε αυτήν μόνο μέσα στη συνάρτηση, δεν μπορώ οπουδήποτε μέσα στον κώδικα μου. Η νέα υπό-κατηγορία που δημιουργείται έχει να κάνει με τα παράλληλα κομμάτια και είναι οι μεταβλητές που ορίζονται μέσα σε αυτά. Ονομάζονται thread-local μεταβλητές και ορίζονται μέσα σε ένα spawn μπλοκ. Τέτοιες μεταβλητές είναι ιδιωτικές του κάθε επεξεργαστή και αποθηκεύονται στην τοπική τους μνήμη. Κανένας άλλος επεξεργαστής δεν μπορεί να έχει πρόσβαση σε τοπική μεταβλητή κάποιου άλλου επεξεργαστή. Πρέπει να είμαστε προσεκτικοί και πάλι σε αυτή την περίπτωση, να μην έχουμε ορισμένες δύο μεταβλητές με το ίδιο όνομα και η μια να είναι καθολική και η άλλη τοπική γιατί θα δημιουργείται σύγχυση μεταξύ των μεταβλητών από τους επεξεργαστές. Επίσης γενικά μιλώντας πρέπει να είμαστε πολύ προσεκτικοί στον τρόπο που διαχειριζόμαστε τις μεταβλητές μας στον παράλληλο προγραμματισμό, γιατί στις καθολικές μεταβλητές μπορεί οποιοσδήποτε επεξεργαστής να τις χειρίζεται και ίσως να υπάρχουν μπερδέματα αν κάποιος γράφει σε κάποια μεταβλητή και δεν είναι υπό την αντίληψη μας. Μια επιπλέον αδυναμία της γλώσσας παρουσιάζεται και στον τύπο των μεταβλητών που επιτρέπεται. Στη γλώσσα XMTC δικαιούμαστε να ορίσουμε μόνο μεταβλητές τύπου int και long μεγέθους 32 bit δηλαδή κάνουμε χρήση μόνο ακέραιων αριθμών. Επιπρόσθετα μπορούμε να δημιουργήσουμε δομή structure που να περιέχει όμως μόνο τους τύπους που μας επιτρέπονται, δηλαδή int και long. Ακόμη μια χρήση που μπορούμε να κάνουμε είναι η χρήση πινάκων, τόσο μονοδιάστατου όσο και δισδιάστατου. Επιτρέπεται και η χρήση δεικτών αλλά αποθαρρύνεται γιατί αποτελεί μια περίπλοκη δομή για το XMTC. Το νέο στοιχείο που εισάγεται σε αυτή την γλώσσα όσον αφορά τις μεταβλητές και δεν υπάρχει στην C, είναι η μεταβλητή psBaseReg. Μεταβλητές τέτοιου τύπου μπορούν να χρησιμοποιούνται μόνο για prefix-sum συναρτήσεις και είναι η δεύτερη παράμετρος που καλείται με αυτή την συνάρτηση. Ορίζεται ως καθολική μεταβλητή πριν καλεστεί με την συνάρτηση. Σε σειριακό κώδικα μπορώ να την χρησιμοποιώ κανονικά, ενώ σε παράλληλο κομμάτι κώδικα μόνο από την συνάρτηση ps(). Υπάρχει και εδώ κάποιος περιορισμός, αφού μόνο έξι(6) μεταβλητές τέτοιου τύπου δικαιούμαστε να χρησιμοποιούμε σε κάθε αλγόριθμο. Αναμένεται ότι σε μεταγενέστερα επίπεδα θα μπορεί να γίνεται χρήση περισσοτέρων τύπων μεταβλητών.

3.3.6 Σύμβολο $

Το σύμβολο αυτό χρησιμοποιείται από τα παράλληλα κομμάτια του προγράμματος μας και ορίζει την τιμή του κάθε επεξεργαστή. Το σύμβολο αυτό δίνει την δυνατότητα στον προγραμματιστή να έχει πρόσβαση στον επεξεργαστή και να εκτελεί εντολές με βάση το δικό του αναγνωριστικό. Η τιμή του συμβόλου αυτού δεν μπορεί να αλλάξει από τον χρήστη, παρά μόνο να χρησιμοποιείται για κάλεσμα. Οι τιμές που έχουν τα σύμβολα $ είναι από το πεδίο των παραμέτρων που δίνεται στη συνάρτηση spawn(). Ένα παράδειγμα με το σύμβολο $ είναι το ακόλουθο:

```
Table[n];
spawn(1,n){
        Table[$-1]=1;
}
```

Σε αυτό το κομμάτι κώδικα ο κάθε επεξεργαστής έχει την ευθύνη μιας κυψελίδας του πίνακα και την αρχικοποιεί με 1. Η τιμή της κυψελίδας ορίζεται από το $-1 γιατί οι θέσεις του πίνακα μας ξεκινούν από μηδέν(0) ενώ ο αριθμός των επεξεργαστών από 1.
Το σύμβολο αυτό χρησιμοποιείται πάρα πολύ στους κώδικες μου αλλά και γενικά σε όλους τους κώδικες παράλληλων αλγορίθμων.

3.3.7 Συναρτήσεις Κλήσεις Συστήματος

Ακόμα ένας περιορισμός εμφανίζεται στην κλήση συναρτήσεων, αφού δεν επιτρέπεται να καλούμε συναρτήσεις μέσα σε παράλληλα κομμάτια, δηλαδή μέσα spawn μπλοκ. Αν καλέσουμε συνάρτηση μέσα σε παράλληλο κομμάτι δεν θα δημιουργηθεί κάποιο πρόβλημα κατά την μεταγλώττιση αλλά θα προκύψει πρόβλημα κατά την εκτέλεση του προγράμματος. Όσον αφορά τώρα την συγγραφή συναρτήσεων, επιτρέπεται να δημιουργούμαι νέες συναρτήσεις σύμφωνα πάντα με τους κανόνες της γλώσσας C και να τις καλούμε οπουδήποτε μέσα στο πρόγραμμα εκτός από παράλληλα κομμάτια. Προς το παρόν η γλώσσα XMTC δεν υποστηρίζει κλήσεις συστήματος, που για παράδειγμα είναι I/O συναρτήσεις ή μαθηματικές συναρτήσεις από την βιβλιοθήκη math.h όχι μόνο στο παράλληλο κομμάτι αλλά και στο σειριακό. Μια συνάρτηση που προσωπικά θεωρώ σημαντική αλλά και γενικά είναι αποδεκτή είναι η συνάρτηση τυπώματος, printf(), που είναι πολύ χρήσιμη, αλλά δεν μπορούμε να την χρησιμοποιήσουμε ελεύθερα μέσα σε παράλληλα κομμάτια. Επίσης για την χρήση της δεν είναι απαραίτητη η βιβλιοθήκη #include stdio.h αφού εισάγεται αυτόματα η βιβλιοθήκη xmtio.h που ευθύνεται για την είσοδο και έξοδο αποτελεσμάτων. Η χρήση της συνάρτησης printf() μπορεί να έχει μόνο μέχρι 3 παραμέτρους και αυτοί να είναι ακέραιοι αριθμοί αφού μόνο αυτοί υποστηρίζονται ως μεταβλητές. Ακόμα μια σημαντική απώλεια

συνάρτησης είναι η συνάρτηση διαβάσματος από το πληκτρολόγιο scanf(), που δεν μας επιτρέπεται η χρήση της.

3.3.8 Εντολή "Prefix Sum" και "Prefix Sum to Memory"

Οι δύο αυτές εντολές αποτελούν νέα στοιχεία που έχει η γλώσσα XMTC και δεν παρουσιάζονται στην C. Η χρήση τους μέσα στον κώδικα γίνεται ως ακολούθως:

ps(int local_integer,psBaseReg ps_base);

και

psm(int local_integer, int variable);

Η χρήση και των δύο αυτών συναρτήσεων είναι να προσθέτουν την τιμή της μεταβλητής local_integer, που είναι η πρώτη παράμετρος στην τιμή της δεύτερης παραμέτρου όπου στη συγκεκριμένη περίπτωση είναι είτε η μεταβλητή ps_base είτε η variable. Επίσης οι δύο αυτές συναρτήσεις αντιγράφουν αυτόματα την παλιά τιμή της δεύτερης παραμέτρου στην πρώτη παράμετρο, δηλαδή την μεταβλητή local_integer στη συγκεκριμένη περίπτωση. Ουσιαστικά η εντολή ps προσθέτει την τιμή της πρώτης και δεύτερης παραμέτρου και η δεύτερη παράμετρος κρατά το αποτέλεσμα του αθροίσματος. Την δουλειά αυτή την ονομάζουμε προθεματικό άθροισμα. Όπως αναφέρθηκε ξανά η εντολή αυτή μπορεί να εμφανιστεί μόνο σε σειριακό κομμάτι και οι παράμετροι μπορούν να είναι μόνο ακέραιοι αριθμοί. Συγκρίνοντας τις δύο αυτές εντολές που κάνουν την ίδια δουλειά παρατηρούμε ότι η συνάρτηση ps είναι πιο αποδοτική από την psm και αυτό οφείλεται στην αρχιτεκτονική. Οι μεταβλητές ps_base είναι καταχωρητές ενσωματωμένοι στην αρχιτεκτονική με καλύτερη πρόσβαση, ενώ οι διευθύνσεις μνήμης για μεταβλητές τύπου int είναι πιο δύσκολη η πρόσβαση σε αυτά. Επίσης αξίζει να αναφερθεί ότι η τιμή της πρώτης παραμέτρου πρέπει να ισούται με 0 ή 1 πριν την εκτέλεση του προγράμματος. Σε αυτή την Διπλωματική Εργασία δεν γίνεται χρήση τέτοιων συναρτήσεων.

3.3.9 Επιπρόσθετα Σύνολα Δεδομένων και Μνήμη

Όπως έχουμε δει σε ποιο πάνω περιορισμούς καθώς παρουσιάζαμε στοιχεία της XMTC η XMT FPGA αλλά και ο προσομοιωτής δεν είναι δυνατό να λάβουν καλέσματα από το σύστημα(system calls). Δυο συχνά και χρήσιμα καλέσματα είναι το διάβασμα δεδομένων από αρχεία και η δυναμική δέσμευση μνήμης που δυστυχώς δεν μπορούμε να κάνουμε χρήση τους λόγο των περιορισμών που παρουσιάζονται. Για αυτό το λόγο έχει δημιουργηθεί μια προσωρινή μέθοδος που μπορούμε να δουλεύουμε σε επιπλέον δεδομένα και να γίνεται δυναμική δέσμευση μνήμης επίσης, με την ελπίδα όμως ότι θα βελτιωθεί σύντομα η πλατφόρμα και θα μπορούν να υποστηρίζονται και αυτές οι λειτουργίες. Οι χρήστες για να πετύχουν αυτές τις λειτουργίες θα πρέπει να ακολουθήσουν κάποια βήματα. Αρχικά θα

πρέπει να ξέρουν ποιες και τι τύπου είναι οι μεταβλητές που χρειάζονται και να προετοιμάζουν ένα αρχείο που θα τις περιέχει μέσα. Επίσης θα πρέπει να γίνεται χρήση του εργαλείου memMapCreate που είναι εργαλείο μνήμης. Για το εργαλείο αυτό θα πρέπει επίσης ο χρήστης να ετοιμάσει και τα ακόλουθα τρία(3) αρχεία:

- Ένα header file(.h) που θα χρησιμοποιηθεί με το κώδικα του προγράμματος. Το αρχείο αυτό περιλαμβάνει τις δηλώσεις των κοινόχρηστων μεταβλητών όπου και δεσμεύεται η απαραίτητη μνήμη. Δηλώνουμε μια κοινόχρηστη μεταβλητή με το να ορίσουμε το όνομα και το μέγεθος της, Το αρχείο αυτό θα μπορούσαμε να πούμε ότι είναι μια βιβλιοθήκη και για αυτόν το λόγο θα πρέπει να ενσωματώνεται με τον κώδικα με τον ίδιο τρόπο όπως οι βιβλιοθήκες, δηλαδή #include file.h
- Ένα δυαδικό αρχείο(.xbo) που χρησιμοποιείται σαν είσοδος στον μεταγλωττιστή. Στο αρχείο αυτό υπάρχουν οι αρχικές τιμές των μεταβλητών που ορίζονται μέσα από το αρχείο header(.h). Οι τιμές στις μεταβλητές δίνονται από τον χρήστη μέσω του εργαλείου μνήμης MemMapCreator. Τα στοιχεία αυτού του αρχείου έχουν δυαδική μορφή και εισάγεται στο πρόγραμμα μας το αρχείο αυτό από την κονσόλα που εργαζόμαστε.
- Προαιρετικά ένα αρχείο κειμένου(.txt) που χρησιμοποιείται για δική μας χρήση για να ελέγχουμε και να διορθώνουμε πιο εύκολα λάθη, αφού είναι δύσκολο να το κάνουμε με το δυαδικό αρχείο. Το αρχείο αυτό περιέχει ότι περιέχει και το δυαδικό αρχείο παρουσιάζοντας όμως τα στοιχεία μας σε δεκαδική μορφή. Αυτό γίνεται για δική μας διευκόλυνση.

Σε μελλοντικές εκδοχές της πλατφόρμας XMT όπου θα υποστηρίζεται η κλήση συνάρτησης για είσοδο και έξοδο δεδομένων το εργαλείο αυτό θα αντικατασταθεί και δεν θα χρησιμοποιείται καθόλου.

3.3.10 XMT Serializer

Το XMT όπως αναφέρθηκε πολλές φορές είναι μια πλατφόρμα με πολλές αδυναμίες και λάθη. Οι αδυναμίες αυτές δημιουργούν προβλήματα όπου δεν μπορούν να ανιχνευτούν από τον μεταγλωττιστή, αλλά ούτε από τον προσομοιωτή. Επίσης τις περισσότερες φορές δεν γίνονται αντιληπτά ούτε από τον ίδιο τον προγραμματιστή, έτσι υπάρχει η ανάγκη να γίνεται περισσότερος έλεγχος για να είμαστε σίγουρη για το αποτέλεσμα που έχουμε. Ουσιαστικά την αδυναμία ελέγχου και αποτελμάτωσης παράλληλου κώδικα την συμπληρώνει μια σειριακή λύση. Για αυτό το πρόβλημα χρησιμοποιούμε το εργαλείο σειριοποίησης(XMTC Serializer), το οποίο μετατρέπει τα παράλληλα κομμάτια σε σειριακά και τρέχει το πρόγραμμα μας σαν κανονικό πρόγραμμα στη γλώσσα

προγραμματισμού C με ένα κανονικό μεταγλωττιστή, για παράδειγμα τον γνωστό μας gcc. Αξίζει να σημειωθεί ότι γίνεται ένα αντίγραφο του νέου μας κώδικα και ο υπάρχον κώδικας παραμένει όπως είχε. Για να γίνει αυτός ο έλεγχος θα πρέπει ο χρήστης να κάνει τις δύο πιο κάτω εντολές στην κονσόλα:

xmtcser mycode.c

gcc mycode.serialized.c –o mycode

./mycode

3.3.11 Μεταγλώττιση και Προσομοίωση ενός Προγράμματος XMTC

Όπως και στην σειριακή επεξεργασία πρώτα πρέπει να γίνει μεταγλώττιση (compile) ενός προγράμματος και την μεταγλώττιση να ακολουθήσει η προσομοίωση (simulate) του κώδικα.

- **Μεταγλώττιση Κώδικα**

  Για την μεταγλώττιση του παράλληλου κώδικα που είναι γραμμένος στη γλώσσα προγραμματισμού XMTC, πρέπει ο χρήστης να καλέσει τον μεταγλωττιστή xmtcc και αυτό γίνεται μέσα από την κονσόλα με την πιο κάτω εντολή.

  xmtcc program.c

- **Προσομοίωση Κώδικα**

  Μετά από την μεταγλώττιση προκύπτει ένας assembly κώδικας ο οποίος είναι αυτός που χρειάζεται και θα πάρει ο προσομοιωτής για να κάνει την προσομοίωση με χρήση της εντολής xmtsim. Η προσομοίωση μπορεί να γίνει με δύο(2) τρόπους. Ο ένας χρησιμοποιεί assembly simulation όπου εκτελείται σειριακή προσομοίωση, κάτι που δεν ενδείκνυται. Η εντολή για αυτό τον τρόπο είναι: *xmtsim –binload a.b a.sim*

  Ο δεύτερος τρόπος που υπάρχει είναι cycle-accurate Simulation όπου είναι και πιο ρεαλιστικός τρόπος αφού γίνεται παράλληλη εκτέλεση. Εμείς χρησιμοποιούμε αυτό τον τρόπο ώστε να μπορούμε να παίρνουμε και τον συνολικό αριθμό κύκλων που γίνονται, άλλωστε είναι αυτό που μας ενδιαφέρει. Η προσομοίωση γίνεται με την εξής εντολή:

  xmtsim –cycle –biloaad a.b a.sim

Και στις δύο περιπτώσεις η εντολή γράφεται από την κονσόλα.

3.3 Περιορισμοί

Με το πέρασμα του χρόνο τόσο μέσα από την μελέτη της γλώσσας XMTC αλλά και πιο έμπρακτα μέσα από τις προσπάθειες υλοποίησης των αλγορίθμων που μου ανατέθηκαν παρατήρησα τόσο ότι δυστυχώς η γλώσσα αυτή αντιμετωπίζει αρκετά προβλήματα και υστερεί σε αρκετά σημεία επίσης. Πολλά από αυτά αναφέρονται πιο πάνω μέσα από την μελέτη κάθε υπό-κεφαλαίου ξεχωριστά, αλλά θεώρησα ότι θα ήταν καλό να μαζευτούν και να αναφερθούν ως ένα ξεχωριστώ κεφάλαιο όλα μαζί. Έτσι πιο κάτω αναφέρω τους περιορισμούς και απαγορεύσεις που εντόπισα.

- Ο μέγιστος αριθμός εντολών που μπορούν να οριστούν μέσα σε ένα spawn μπλοκ είναι 1000 εντολές.
- Αν το όριο των virtual thread ξεπεράσει το όριο των 64 για FPGA και 1024 για τον προσομοιωτή της πλατφόρμας τότε και στις δύο περιπτώσεις αρχίζει να μεταχειρίζεται και τα επιπλέον με τυχαία μεθοδολογία.
- Ο αριθμός των τοπικών μεταβλητών μέσα σε ένα παράλληλο κομμάτι δεν πρέπει να είναι πολύ μεγάλος λόγω περιορισμού μνήμης.
- Δεν επιτρέπεται η χρήση της συνάρτησης random() για δημιουργία τυχαίων αριθμών.
- Δεν επιτρέπεται το κάλεσμα συναρτήσεων μέσα σε παράλληλα κομμάτια και κατ' επέκταση απαγορεύεται και η κλήση αναδρομικής συνάρτησης αλλά και η συνάρτηση printf() που χρησιμεύει για τύπωμα.
- Δεν επιτρέπονται φωλιασμένα spawn().
- Οι παράμετροι της συνάρτησης spawn() πρέπει να είναι ακέραιοι αριθμοί.
- Αν υπάρχει σημείο που έχω ταυτόχρονη γραφή, πολύ πιθανόν να οδηγηθώ σε ατέρμονα βρόχο και να μην τερματίσω ποτέ.
- Μέσα σε single spawn (sspawn) δεν δικαιούμαι να ορίσω νέες μεταβλητές.
- Δεν δικαιούμαι να εισάγω την βιβλιοθήκη math.h και κατ' επέκταση ούτε να χρησιμοποιώ τις συναρτήσεις της.
- Δεν επιτρέπεται να κάνω δυναμική δέσμευση δισδιάστατου πίνακα, δηλαδή απαγορεύεται ο ορισμό int table[n][n].
- Μπορώ να έχω μόνο μεταβλητές τύπου int, ακέραιοι δηλαδή.
- Δεν διακαούμε να χρησιμοποιώ τον ορισμό unsigned.
- Δεν επιτρέπεται η χρήση της βιβλιοθήκης stdio.h
- Με το κάλεσμα της εντολής printf() μπορώ να περνώ μόνο τρείς παραμέτρους, δηλαδή δικαιούμαι να τυπώνω μέχρι τρεις αριθμούς κάθε φορά.

- Όταν τυπώνω σε FPGA προσομοιωτή μου επιτρέπεται να τυπώνω μέχρι 114,687 χαρακτήρες.

Πολλοί από αυτούς τους περιορισμούς ήδη μελετούνται και γίνονται προσπάθειες βελτίωσης τους από την ερευνητική ομάδα που ασχολείται με το θέμα αυτό. Ελπίζουμε σε μια καλύτερη πλατφόρμα έτσι ώστε να γίνονται πολύ περισσότερα πράγματα σε παράλληλους υπολογισμούς.[6,7]

Κεφάλαιο 4

Παράλληλη Αναζήτηση

4.1 Πρόβλημα 37

4.2 Σειριακός Αλγόριθμος 37

4.3 Παράλληλος Αλγόριθμος 40

4.4 Πειραματική Αξιολόγηση 45

Το πρόβλημα της αναζήτησης έχει υλοποιηθεί για δύο κύριους λόγους. Ο πρώτος λόγος που στην πορεία φάνηκε και πολύ βοηθητικός είναι ότι υλοποιήθηκε ο Αλγόριθμος Αναζήτησης για εξάσκηση με την πλατφόρμα και τον προσομοιωτή. Ήταν ένας τρόπος για να μάθω και σε πρακτικό επίπεδο πως λειτουργεί η πλατφόρμα XMT, ποίες οι απαιτήσεις και περιορισμοί της. Αποτελούσε μια πολύ καλή ευκαιρία για τον λόγο ότι είναι ένας σχετικά εύκολος αλγόριθμος και η βάση του είναι ο γνωστός αλγόριθμος Δυαδικής Αναζήτησης που γνωρίζω από τις εμπειρίες μου στον προγραμματισμό μέσα από το πέρασμα των χρόνων στην Επιστήμη της Πληροφορικής. Η υλοποίηση αυτή ήταν ένας πολύ καλός τρόπος εξάσκησης μου, έτσι ώστε σε μεταγενέστερο επίπεδο να μπορούσα να ανταπεξέλθω σε ποιο δύσκολους αλγορίθμους που τυχόν θα μου ανατίθενται. Ο δεύτερος λόγος που υλοποιήθηκε ο αλγόριθμος αναζήτησης είναι γιατί επρόκειτο να χρησιμοποιηθεί στον Αλγόριθμο Κυρτού Περιβλήματος που θα ακολουθούσε, όπου ο Αλγόριθμος Αναζήτησης αποτελούσε κομμάτι του αλγορίθμου αυτού. Δυστυχώς στην πορεία διαπιστώσαμε ότι δεν θα μπορούσε να υλοποιηθεί παράλληλα ο Αλγόριθμους Κυρτού Περιβλήματος λόγω του περιορισμού της XMTC να καλεί συναρτήσεις μέσα σε παράλληλα κομμάτια, πράγμα απαραίτητο για τον παράλληλο αλγόριθμο. Η αδυναμία κλήσης συνάρτησης απέκλειε και την αναδρομή και κατ' επέκταση την υλοποίηση του, αφού ήταν απαραίτητη η αναδρομική κλήση συνάρτησης. Η υλοποίηση του Αλγορίθμου Κυρτού Περιβλήματος χωρίς παράλληλη αναδρομή θα έφτανε στα όρια και επίπεδα του σειριακού αλγορίθμου και δεν υπήρχε κανένας λόγος μελέτης, υλοποίησης και πειραματικής αξιολόγησης του, αφού στοχεύουμε σε παράλληλους αλγόριθμους στην παρούσα Διπλωματική Εργασία.

4.1 Πρόβλημα

Το πρόβλημα Αναζήτησης έχει ως δεδομένα εισόδου ένα σύνολο από στοιχεία, έστω $X=\{X_1,X_2,X_3,\ldots,X_n\}$ όπου X_i ένας τυχαίος αριθμός. Επίσης είσοδο αποτελεί και ένα στοιχείο y όπου στόχος μας είναι να αναζητήσουμε μέσα από το σύνολο των αριθμών αν το στοιχείο y υπάρχει στο σύνολο X όπως δηλώνει και το όνομα του αλγορίθμου. Αν το στοιχείο που ψάχνουμε βρίσκετε στο σύνολο τότε επιστρέφουμε την θέση που βρέθηκε το συγκεκριμένο στοιχείο, αλλιώς επιστρέφουμε κάποιο μήνυμα ότι δεν υπάρχει το στοιχείο y στην λίστα με τους αριθμούς που έχουμε στην διάθεση μας, μπορούν να συμβούν και τα δύο ενδεχόμενα αλλά όχι με τα ίδια δεδομένα. Τελικός στόχος είναι η επιστροφή της θέσης που είναι αποθηκευμένο το στοιχείο αν υπάρχει.

4.2 Σειριακός Αλγόριθμος

Όπως είναι γνωστό η σειριακή λύση γίνεται με την χρήση ενός και μόνο επεξεργαστή, ο οποίος είναι αυτός που χειρίζεται ολόκληρο το σύνολο των στοιχείων και κάνει τους ανάλογους υπολογισμούς και μετατοπίσεις που αναφέρονται πιο κάτω με μεγαλύτερη λεπτομέρεια. Το πρόβλημα της αναζήτησης βρίσκει λύση με περισσότερους από ένα αλγορίθμους που είναι στην διάθεση μας, όπως για παράδειγμα η Γραμμική Αναζήτηση (Linear Search), η Δυαδική Αναζήτηση (Binary Search) και ο αλγόριθμος Αυτοοργάνωσης (Self Organizing Search) καθώς και κάποιες παραλλαγές τους. Επικρατέστερος και πιο διαδεδομένος αλγόριθμος, αφού είναι αυτός που οι περισσότεροι χρησιμοποιούν, από τους υπάρχοντες, είναι ο Αλγόριθμος Δυαδικής Αναζήτησης. Ο συγκεκριμένος αλγόριθμος αποτελεί κλασικό παράδειγμα της τεχνικής "Διαίρει και Βασίλευε" (divide and conquer). Τεχνική κατά την οποία το αρχικό πρόβλημα διαιρείται σε μικρότερα υπό-προβλήματα, όπου και λύνονται ξεχωριστά και στην συνέχεια οι λύσεις τους συνδυάζονται για να προκύψει η λύση του αρχικού προβλήματος. Η μέθοδος αυτή χρησιμοποιείται σε πολλούς αλγορίθμους, όπως αλγορίθμους ταξινόμησης (merge sort, quick sort). Μιλώντας πιο συγκεκριμένα η μέθοδος χρησιμοποιείται σε τρία στάδια. Στο πρώτο βήμα το πρόβλημα διαιρείται σε ένα αριθμό από μικρότερα προβλήματα, στο δεύτερο βήμα ακολουθεί αναδρομική επίλυση των υπό-προβλημάτων εξαιρώντας τις περιπτώσεις όπου είναι εύκολα υπό-προβλήματα και λύνονται απευθείας χωρίς αναδρομή και το τρίτο και τελευταίο πρόβλημα είναι η διαδικασία που συνδυάζονται οι λύσεις και προκύπτει μια μόνο λύσης που αντιστοιχεί στο αρχικό πρόβλημα.

Πιο συγκεκριμένα έχουμε σαν είσοδο δεδομένων ένα μονοδιάστατο πίνακα με ακέραιους αριθμούς μεγέθους n. Ο πίνακας αυτός προαπαιτείται να είναι ταξινομημένος, δηλαδή να ικανοποιούνται οι συνθήκες $X_1<X_2<X_3....<X_n$. Η δυαδική αναζήτηση στηρίζεται στο γεγονός ότι ο πίνακας είναι ταξινομημένος, μειώνοντας έτσι σημαντικά τον αριθμό των συγκρίσεων που απαιτούνται. Επίσης κάθε στοιχείο του πίνακα είναι μοναδικό, δηλαδή δεν υπάρχουν δύο θέσεις του πίνακα με το ίδιο στοιχείο. Στην διάθεση μας έχουμε επίσης έναν ακέραιο αριθμό y όπου είναι το στοιχείο που αναζητούμε στον πίνακα. Ο αλγόριθμος ξεκινά με τη σύγκριση του στοιχείου y με το μεσαίο στοιχείο του πίνακα, δηλαδή floor(n/2) είναι το μεσαίο στοιχείο. Βασισμένο στην έξοδο αυτής της σύγκρισης, η αναζήτηση είτε μπορεί να τερματίσει με επιτυχία, εάν y=Xn/2, είτε μπορεί να περιοριστεί στο μισό αριστερό, είτε μισό δεξιό κομμάτι του πίνακα για το λόγο ότι επειδή το διάνυσμα είναι σε αύξουσα σειρά, η τιμή του στοιχείου στη θέση i, όπου i η μέση του πίνακα, θα είναι μικρότερη ή ίση από όλα τα στοιχεία δεξιά και μεγαλύτερη ή ίση από όλα τα στοιχεία αριστερά. Έτσι, αν σε μια σύγκριση, το y είναι μεγαλύτερο από το στοιχείο στη θέση i, δεν έχει νόημα να κάνουμε συγκρίσεις με τα στοιχεία που βρίσκονται σε θέση y<i (δηλαδή, αριστερά από τη θέση i).Από αυτό το βήμα και έπειτα αναζητούμε σε ένα μικρότερο κομμάτι του πίνακα μας όπου πιθανόν θα βρίσκεται το στοιχείο που ψάχνουμε. Αυτήν ακριβώς την ιδιότητα εκμεταλλεύεται η δυαδική αναζήτηση, ώστε να μειώσει κατά το δυνατό τις συγκρίσεις που κάνει και να τερματίσει πιο γρήγορα. Η συνάρτηση επαναλαμβάνεται για πεπερασμένο αριθμό φορών, μέχρις ότου είτε το στοιχείο Xi υπολογιστεί τέτοιο ώστε Xi=y, είτε το μέγεθος του υπό-πίνακα που είναι υπό μελέτη είναι ίσο με 1. Οι δύο περιπτώσεις αυτές είναι τερματικές και ο αλγόριθμος τερματίζει να εκτελείται. Στην πρώτη περίπτωση λύση βρέθηκε και επιστρέφεται η θέση που βρίσκεται το Xi που ισούται με y, ενώ στην δεύτερη περίπτωση, δεν υπάρχει το στοιχείο στον πίνακα μας και επιστρέφω μήνυμα λάθους. Ακολουθεί ψευδοκώδικας που επιλύει το πρόβλημα με Δυαδική Αναζήτηση.

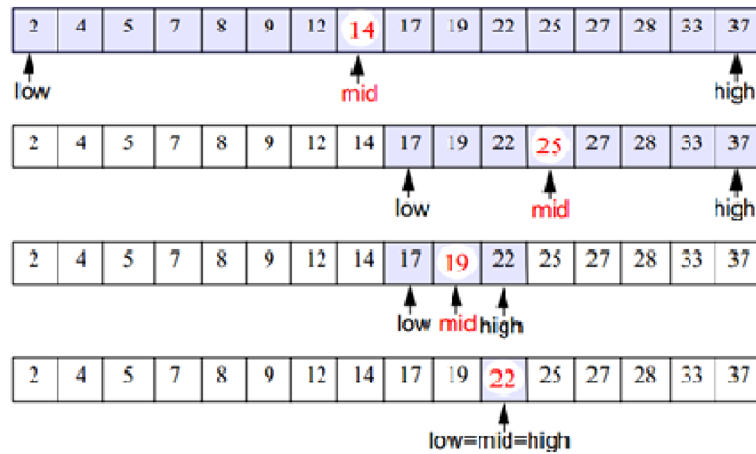
```
BinarySearch(A[0..n-1],Value){
    low=0;
    high=N-1;
    while(low<=high){
        mid=(low+higt)/2;
        if(A[mid]>value)
            high=mid-1;
        else if (A[mid<value])
            low=mid+1;
        else return mid;
    }
```

return not_found;

}

Ακολουθεί παράδειγμα εκτέλεσης(Σχήμα 4.1):

BinarySearch(A[16],22)



Σχήμα 4.1[4]

Η ορθότητα του αλγορίθμου βασίζεται στο ότι σε κάθε βήμα η ταξινομημένη λίστα διαχωρίζεται σε συνεχόμενες υπό-λίστες, και ότι κάθε υπό-λίστα είναι επίσης ταξινομημένη. Αυτό εξασφαλίζει, σε κάθε βήμα, τον ορθό προσδιορισμό της υπό-λίστας που δύναται το y να ανήκει.

Πολυπλοκότητα Χρόνου:

Σκοπός είναι τώρα να ασχοληθούμε με την πολυπλοκότητα του αλγορίθμου αφού είναι και αυτή που θα μας φανεί χρήσιμη για να τον συγκρίνουμε με τον παράλληλο αλγόριθμο μας. Η χρονική πολυπλοκότητα του αλγορίθμου είναι λογαριθμική ως προς το μέγεθος της εισόδου, έστω μέγεθος πίνακα n και σειριακός χρόνος T(n), τότε, λύνει το πρόβλημα σε T(n)=O(logn). Το μέγεθος του προβλήματος μειώνεται σε κάθε επανάληψη στο μισό, άρα έχουμε O(logn) επαναλήψεις στη χειρότερη περίπτωση που είναι να μην βρεθεί το στοιχείο στην λίστα αλλιώς έχουμε πιο λίγες επαναλήψεις. Η κάθε επανάληψη παίρνει χρόνο Θ(1) αφού εκτελούνται απλές πράξεις σε κάθε επανάληψη και έτσι προκύπτει ο συνολικός σειριακός χρόνος. Ο σειριακός χρόνος αντιπροσωπεύει την χειρότερη περίπτωση όπου το στοιχείο δεν βρίσκεται στον πίνακα και αναγκαστικά θα πρέπει να κοιτάξουμε σε ¨όλο¨ το διάνυσμα. Υποθέτουμε επίσης ότι η σύγκριση παίρνει χρόνο σταθερό χρόνο c, για να ελέγξει που βρίσκεται το στοιχείο.

Έτσι έχουμε:

$$T(n) = \begin{cases} c & \text{if } n = 0 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + c & \text{if } n > 0 \end{cases}$$

Έχουμε στην χειρότερη περίπτωση logn επαναλήψεις και έτσι προκύπτει:

$$T(n) = c \cdot \log n + c$$

Όπου υποθέτουμε ότι c=Θ(1), δηλαδή ότι παίρνει μόνο μια μονάδα χρόνου και καταλήγουμε στο συμπέρασμα ότι:

$$T(n) = O(\log n)$$

Η λύση είναι βέλτιστη ως προς το χρόνο αφού είναι ο καλύτερος χρόνος που μπορούμε να πετύχουμε για την επίλυση του προβλήματος Αναζήτησης και ο χρόνος που θα συγκρίνουμε για τον βέλτιστο παράλληλο αλγόριθμο μας. Το κόστος αξίζει να σημειωθεί ότι είναι ίσο με τον σειριακό χρόνο ολοκλήρωσης.

4.3 Παράλληλος Αλγόριθμος

Για τον παράλληλο αλγόριθμο εξακολουθούν να ισχύουν τα δεδομένα εισόδου που είχαμε και στον σειριακό αλγόριθμο με επιπλέον μια μεταβλητή, την p. Δηλαδή, ένας μονοδιάστατος πίνακας μεγέθους n, όπου n είναι και το μέγεθος του προβλήματος. Ο πίνακας μας πρέπει να είναι γραμμικά ταξινομημένος έτσι ώστε $X_1<X_2<X_3\ldots<Xn$. Δοθέντος ενός στοιχείου y ενδιαφερόμαστε να λύσουμε το πρόβλημα της αναζήτησης, δηλαδή να δούμε αν στον μονοδιάστατο πίνακα υπάρχει το στοιχείο κλειδί, το y, και αν υπάρχει να ορίσουμε την θέση στην οποία βρίσκεται, όταν περισσότεροι από ένας επεξεργαστές συνεργάζονται ταυτόχρονα. Ο πίνακας και το στοιχείο που αναζητούμε είναι αποθηκευμένα στην κοινόχρηστη μνήμη που διαθέτουμε έτσι ώστε όλοι οι επεξεργαστές να μπορούν να έχουν πρόσβαση πάνω τους. Βάση και θεμέλια του παράλληλου αλγόριθμου αποτελεί ο σειριακός Δυαδικός Αλγόριθμος Αναζήτησης που αναλύθηκε με μεγάλη λεπτομέρεια πιο πάνω. Εύστοχα, θα μπορούσε να πει κανείς ότι ο εν λόγω αλγόριθμος αναζήτησης είναι μια φυσική γενίκευση της Δυαδικής Αναζήτησης στα μέτρα του παραλληλισμού όπου εκτελούνται p+1 αναζητήσεις, όπου p ο αριθμός των επεξεργαστών, αντί δυαδική αναζήτηση. Ακολουθείται η ίδια σκεπτική πορεία του ¨Διαίρει και Βασίλευε¨, προσαρμοσμένη στα πλεονεκτήματα και περιορισμούς της παράλληλης επεξεργασίας, δηλαδή τα τρία βήματα που αποτελείται η τεχνική αυτή. Ουσιαστικά, αφού έχουμε στην διάθεση μας περισσότερους από ένα επεξεργαστές μπορούμε να συγκρίνουμε το y ταυτόχρονα με διάφορα στοιχεία από τον πίνακα μας. Η σκεπτική του αλγορίθμου έχει ως εξής: Σε κάθε βήμα, η λίστα χωρίζεται σε p+1 ταξινομημένες υπό-λίστες ίσου μεγέθους, υπάρχει περίπτωση η τελευταία υπό-λίστα να μην έχει το ίδιο μέγεθος, αλλά αυτό δεν αποτελεί πρόβλημα, και ο κάθε επεξεργαστής ελέγχει παράλληλα το τελευταίο στοιχείο της υπό-λίστας που του ανατέθηκε να είναι υπεύθυνος. Το μέγεθος της κάθε υπό-λίστας είναι n/p+1. Η καινούριος παράμετρος p που προσθέσαμε σαν μεταβλητή στον παράλληλο

αλγόριθμο μας, πρέπει να πούμε ότι είναι ο αριθμός των επεξεργαστών που είναι διαθέσιμος στον PRAM μοντέλο μας. Η παράλληλη αναζήτηση όπου p_j ορίζεται ο επεξεργαστής που είναι υπεύθυνος για την υπό-λίστα j για να ελέγξει τα όρια και να κάνει τους ανάλογους ελέγχους. Ακολούθως αφού ανατεθεί στον κάθε επεξεργαστή το κομμάτι του, κάνει τους ακόλουθους ελέγχους, όπου Xi το στοιχείο που ελέγχει ο επεξεργαστής Pi και είναι το τελευταίο στοιχείο της κάθε υπό-λίστας. Αν Xi > y, τότε όλα τα στοιχεία που ακολουθούν το Xi δεν λαμβάνονται πλέον υπόψη. Αν Xi < y, τότε συμβαίνει το αντίθετο. Το αποτέλεσμα αυτής της παράλληλης αναζήτησης είναι είτε να βρεθεί το y, είτε να επικεντρωθεί η αναζήτηση σε μια από τις p+1 υπό-λίστες. Η διαδικασία επαναλαμβάνεται μέχρις ότου να βρεθεί το y,ή το μέγεθος της μιας και μοναδικής λίστας που μένει να μην ξεπερνά το p. Στην πρώτη περίπτωση το ζητούμενο βρέθηκε, ενώ στην δεύτερη περίπτωση η λύση μπορεί να τερματίσει με t συγκρίσεις να γίνονται παράλληλα, όπου $t \leq y$ σε ένα συγκεκριμένο πεδίο στοιχείων της υπό-λίστας. Στο συγκεκριμένο αλγόριθμο έχουν προστεθεί κάποιες επιπλέον κοινόχρηστες μεταβλητές, όπως ο πίνακας c[ ] που χρησιμοποιείται για να δείξει την έξοδο της σύγκρισης που εκτελείται στο κάθε βήμα, επίσης οι ακέραιες μεταβλητές l και r οι οποίες είναι δείκτες που υποδεικνύουν τα όρια, αριστερό και δεξιό αντίστοιχα, της υπό-λίστας που είναι υπό επεξεργασία την συγκεκριμένη χρονική στιγμή.[4,8]

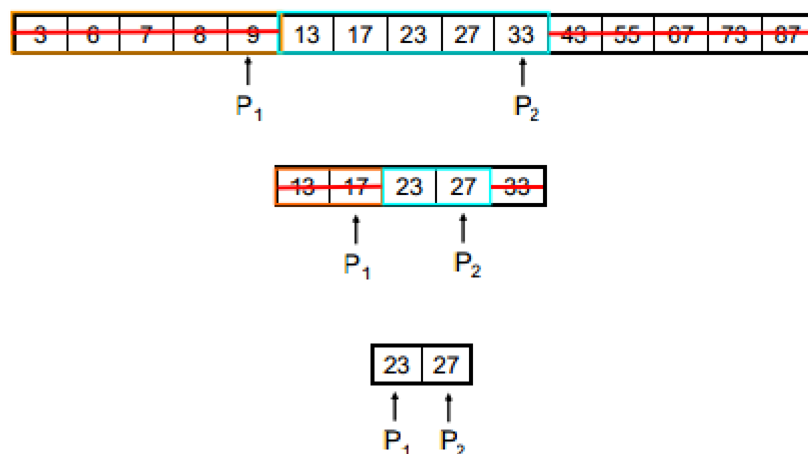
Ακολουθεί ψευδοκώδικας του αλγορίθμου:

```
Begin
        if (j=1){
                l=0; r=n+1; x[0]=-oo; x[n+1]=+oo;
                c[0]=0; c[p+1]=1;
        }
        while(r-l>p){
                2.1 if(j=1){
                        q[0]=l;
                        q[p+1]=r;
                }
                2.2 q[j]=l+j*floor((r-l)/(p+1));
                2.3 if(y=x[qj]){
                        return qj; exit;
                }
                else{
                        if(y>x[qj] && c[j]=1)
                                c[j]=0;
                        if(y<x[qj])
```

```
                    c[j]=1;
            }
            2.4 if(c[j]<c[j+1]){
                    l=q[j];
                    r=q[j+1];
            }
            2.5 if(j=1 && c[0]<c[1]){
                    l=q[0];
                    r=q[1];
            }
    }
    If(j≤r-l){
            3.1Case statement:
                    y=x[l+j] then return l+j;exit;
                    y>x[l+j] then c[j]=0;
                    y<x[l+j] then c[j]=1
            3.2 if(c[j-1]<c[j]) {
                    return l+j-1;
            }
    }
End[4]
```

Στον ψευδοκώδικα όπως παρατηρούμαι οι επεξεργαστές μας είναι συγχρονισμένοι κάτι που δεν συμβαίνει στην πραγματικότητα όπως ανέφερα πολλές φορές και για να λυθεί αυτό το πρόβλημα και να είμαστε πιο σίγουροι για τα αποτελέσματα που έχουμε χρησιμοποιούμε διπλή δομή. Δηλαδή διπλό πίνακα και ανά γύρους διαβάζουν από διαφορετική στήλη έτσι ώστε να ξέρουμε ότι διαβάζουμε τις σωστές τιμές.

Ακολουθεί παράδειγμα εκτέλεσης όπου n=15, y=23 και p=2(Σχηματική Αναπαράσταση 4.2)



Σχήμα 4.2 [8]

Στο σχηματικό παράδειγμα που δίνεται είναι μια εκτέλεση του αλγορίθμου, όπου στο βήμα 1, έχουμε ολόκληρο τον πίνακα μας και 2 επεξεργαστές, άρα χωρίζεται ο πίνακας μας σε 3 ίσα μέρη και οι επεξεργαστές αναλαμβάνουν ο καθένας από ένα κομμάτι. Ο επεξεργαστής P1 πηγαίνει στη θέση 5 και ο επεξεργαστής P2 στη 10. Γίνονται οι απαραίτητοι έλεγχοι και συγκεντρώνουμε την προσοχή μας στις θέσεις 5 μέχρι 10. Επαναλαμβάνεται η διαδικασία στις θέσεις από 5 μέχρι 10 και χωρίζεται σε 3 κομμάτια, οι δυο επεξεργαστές P1 και P2 αναλαμβάνουν τις θέσεις 7 και 9 αντίστοιχα. Η προσοχή μας επικεντρώνεται στις θέσεις 8 και 9 και στο τέλος επιστρέφεται η τιμή 8, γιατί το 23 βρίσκεται στην θέση 8 του πίνακα.

Η ορθότητα του αλγορίθμου βασίζεται στο ότι σε κάθε βήμα η ταξινομημένη λίστα διαχωρίζεται σε συνεχόμενες υπό-λίστες, όπου κάθε υπό-λίστα είναι επίσης ταξινομημένη κατά αύξουσα σειρά. Η ορθότητα στον παράλληλο αλγόριθμο όπως και στον σειριακό πηγάζει από το ίδιο σημείο, την ταξινόμηση της αρχικής λίστας αλλά και των μικρότερων υπό-λιστών που δημιουργούνται κατά την εκτέλεση του αλγορίθμου.
Ο αλγόριθμος μπορεί να εφαρμοστεί σε CREW PRAM μοντέλο με p επεξεργαστές. Εντοπίζουμε την απαίτηση για ταυτόχρονη ανάγνωση (concurrent read) στις κοινόχρηστες μεταβλητές l,r,y όπου χρειάζεται να είναι προσβάσιμα από όλους τους επεξεργαστές. Απλά για αναφορά, μπορούμε να πούμε ότι Ω(logn-logp) παράλληλα βήματα μπορούν να εκτελεστούν και σε EREW μοντέλο όταν το στοιχείο y είναι διαθέσιμο στους επεξεργαστές.

Πολυπλοκότητα Χρόνου-Κόστος:

Για να βρούμε αρχικά τον μέγιστο χρόνο που απαιτείται για την επίλυση του συγκεκριμένου προβλήματος με αυτό τον αλγόριθμο και κατ’ επέκταση και την πολυπλοκότητα που προκύπτει μέσα από τον χρόνο, θα χρειαστεί να αναλύσουμε τον χρόνο που χρειάζεται μια επανάληψη και τον αριθμό των επαναλήψεων που τυχόν να χρειαστεί να γίνουν μέχρι να βρεθεί λύση και το γινόμενο τους αποτελεί τον παράλληλο χρόνο που χρειάζεται να λυθεί το πρόβλημα. Πρέπει να πάρουμε ένα προς ένα τα βήματα που γίνονται σε κάθε επανάληψη. Το πρώτο βήμα που είναι ο διαχωρισμός του πίνακα και η ανάθεση σε διάφορους επεξεργαστές τα όρια που είναι υπεύθυνοι να εργαστούν παίρνει χρόνο Θ(1). Επίσης η σύγκριση του τελευταίου στοιχείου κάθε υπό-λίστας με τον ζητούμενο αριθμό αλλά και η εύρεση αν δεν υπάρχει το στοιχείο στην λίστα γίνεται και αυτή σε Θ(1) χρόνο υποθέτουμε, σύμφωνα πάντα με τις αρχές που ορίσαμε. Τώρα πρέπει να υπολογίσουμε τον αριθμό των επαναλήψεων που χρειάζονται στην χειρότερη περίπτωση έτσι ώστε να έχουμε και τον χειρότερο χρόνο. Υπάρχει και η περίπτωση όπου βρεθεί από την πρώτη σύγκριση, άρα έχω σταθερό αριθμό, αλλά δεν θα μας απασχολήσει αυτή η περίπτωση. Στόχος μας είναι το μέγεθος της υπό-λίστας να είναι ίση με τον αριθμό των επεξεργαστών, αλλά για να γίνει αυτό σε κάθε επανάληψη το μέγεθος της λίστας είναι 1/(p+1) φορές μικρότερο από το μέγεθος της λίστας

στην προηγούμενη επανάληψη και αυτό συμβαίνει γιατί μοιράζουμε την ήδη μοιρασμένη λίστα σε p+1 πιο μικρά κομμάτια. Αυτό αποτελεί μια σειρά όπου στην πρώτη επανάληψη έχουμε n στοιχεία, στην δεύτερη επανάληψη έχουμε n/(p+1) στην τρίτη επανάληψη έχουμε n/(p+1)^2, στην i-οστή επανάληψη έχουμε n/(p+1)^i-1. Το ζητούμενο είναι να έχουμε p+1 στοιχεία στην τελευταία επανάληψη και έστω ότι αυτό συμβαίνει σε λ επαναλήψεις, τότε ακολουθούν κάποιες μαθηματικές πράξεις:

n / (p+1)^λ-1 = p+1

n = (p+1) / (p+1)^λ-1

n = (p+1)^-λ

n = 1/(p+1)^λ

λ = O(logn/logp)

Μέσα από κάποιες απλές μαθηματικές πράξεις προκύπτει ότι ο μέγιστος αριθμός των επαναλήψεων που μπορούμε να έχουμε είναι logn/logp και αφού σε πιο πάνω ανάλυση είδαμε ότι κάθε επανάληψη παίρνει χρόνο Θ(1), μπορούμε να πούμε ότι ο παράλληλος χρόνος είναι O(logn/logp).

Tp(n) = O(logn/logp)

Όπως ορίστηκε η πολυπλοκότητα κάποιου παράλληλου αλγορίθμου είναι το γινόμενο του παράλληλου χρόνου και ο αριθμός των επεξεργαστών που χρησιμοποιούνται. Στο συγκεκριμένο αλγόριθμο χρησιμοποιούμε p επεξεργαστές και άρα η πολυπλοκότητα πολύ απλά είναι:

Cp(n) = O(plogn/logp)

Μέσα από τα αποτελέσματα που παίρνουμε από τις συναρτήσεις που υπολογίζουν τόσο το χρόνο όσο και το κόστος παρατηρούμε ότι είναι συναρτήσει όχι μόνο του n που είναι ο αριθμός των στοιχείων αλλά και το p που είναι το πλήθος των επεξεργαστών.

Όσον αφορά τον χρόνο που χρειάζεται να λυθεί το πρόβλημα παράλληλα μπορούμε να πούμε ότι είναι ο βέλτιστος χρόνος που λύνει το πρόβλημα αναζήτησης, δηλαδή ο μικρότερος χρόνος. Το πρόβλημα δεν μπορεί να λυθεί πιο γρήγορα από O(logn/logp) μονάδες χρόνου, είναι ότι καλύτερο μπορούμε να πετύχουμε σε θέμα χρόνου από οποιοδήποτε παράλληλο αλγόριθμο που επιλύει αυτό το πρόβλημα, αφού αποτελεί και το κάτω φράγμα. Η πολυπλοκότητα του κόστους όπου είναι και αυτό που μας ενδιαφέρει περισσότερο, όπως αναφέραμε σε θεωρητικό επίπεδο πρέπει να είναι της τάξης του καλύτερου σειριακού χρόνου εάν θέλουμε να τον θεωρούμε βέλτιστο. Στην σειριακή επίλυση ο σειριακός χρόνος είναι T*(n)=O(logn). Βλέποντας τα φαίνεται ότι δεν είναι της ίδιας τάξης αφού το κόστος είναι μεγαλύτερο από τον σειριακό χρόνο και άρα υποθέτουμε ότι δεν είναι βέλτιστος, εντούτοις όμως ο αλγόριθμος μπορεί νε θεωρηθεί βέλτιστος κάτω από κάποια συνθήκη. Ο αλγόριθμος

θεωρείται βέλτιστος εάν ο αριθμός των επεξεργαστών(p) είναι σταθερά, δηλαδή δεν είναι κάποια μεταβλητή. Αν p=c όπου c σταθερά τότε έχουμε: O(clogn/logc). Το c και logc θεωρούνται αμελητέα και είναι ασήμαντα, έτσι έχουμε Cp(n) = O(logn) της τάξης του καλύτερου σειριακού χρόνου και είναι βέλτιστος.[4,8]

4.4 Πειραματική Αξιολόγηση

Το σημείο αυτό είναι αυτό που θα μας απασχολήσει περισσότερο και μας ενδιαφέρει περισσότερο επίσης. Είναι η πειραματική αξιολόγηση όπου μέσα από διάφορες μετρήσεις που πάρθηκαν από τα πειράματα που έγιναν κατά την διάρκεια της εξέλιξης της Διπλωματικής Εργασίας αυτής στον προσομοιωτή XMT, καταλήξαμε στην δημιουργία γραφικών παραστάσεων. Οι γραφικές παραστάσεις αυτές συγκρίνονται επίσης με τις αναμενόμενες γραφικές παραστάσεις, δηλαδή τις παραστάσεις που προκύπτουν από την συνάρτηση που έχουμε στα χέρια μας τόσο για το χρόνο όσο και για το κόστος που προέκυψαν από την θεωρητική ανάλυση.

Μέσα από την προσομοίωση τα αποτελέσματα που παίρναμε αντιστοιχούσαν σε μονάδες χρόνου του υπολογιστή και όχι σε δευτερόλεπτα. Αυτό δεν μας απασχολεί ιδιαίτερα αφού είναι η ίδια μονάδα μέτρησης σε όλες τις μετρήσεις που πήραμε.

Επίσης στον συγκεκριμένο αλγόριθμο όπως διαπιστώσαμε πιο πάνω έχουμε δύο μεταβλητές όπου πρέπει να ελέγξουμε, τον αριθμό των επεξεργαστών (p) και τον αριθμό των στοιχείων (n). Αρχικά, θα μεταβάλλουμε τον αριθμό των επεξεργαστών και θα διατηρούμε σταθερό τον αριθμό των στοιχείων του πίνακα, και στην άλλη περίπτωση θα μεταβάλουμε τον αριθμό των στοιχείων και θα κρατάμε σταθερό τον αριθμό των επεξεργαστών που θα εργάζονται. Επίσης επιχειρήσαμε ένα τρίτο πείραμα όπου μεταβάλλαμε και τις δύο μεταβλητές ταυτόχρονα. Τα στοιχεία μας και στις τρείς περιπτώσεις ήταν ταξινομημένα.

Για να έχουμε αξιόπιστες μετρήσεις, χωρίς τυχόν ατασθαλίες για την κάθε περίπτωση, παίρναμε τρείς μετρήσεις με διαφορετικά δεδομένα, όπου σε κάποια το στοιχείο y δεν υπήρχε και υπολογιζόταν ο μέσος χρόνος τους, ο οποίος είναι αυτός που χρησιμοποιείται τόσο στους πίνακες όσο και στις γραφικές.

Για την πρώτη περίπτωση διατηρούμε σταθερό τον αριθμό των στοιχείων που μελετούμε και πιο συγκεκριμένα έχουμε 1024 στοιχεία (n=1024), όπου το 1024 είναι δύναμη του 2 και ίσως να μας διευκόλυνε στις πράξεις για την θεωρητική μέτρηση, γι’ αυτό προτιμήθηκε. Αλλάζουμε όμως τον αριθμό των επεξεργαστών που εργάζονται ταυτόχρονα για να λύσουν το πρόβλημα και να μας δώσουν το επιθυμητό αποτέλεσμα και είναι η παράμετρος που

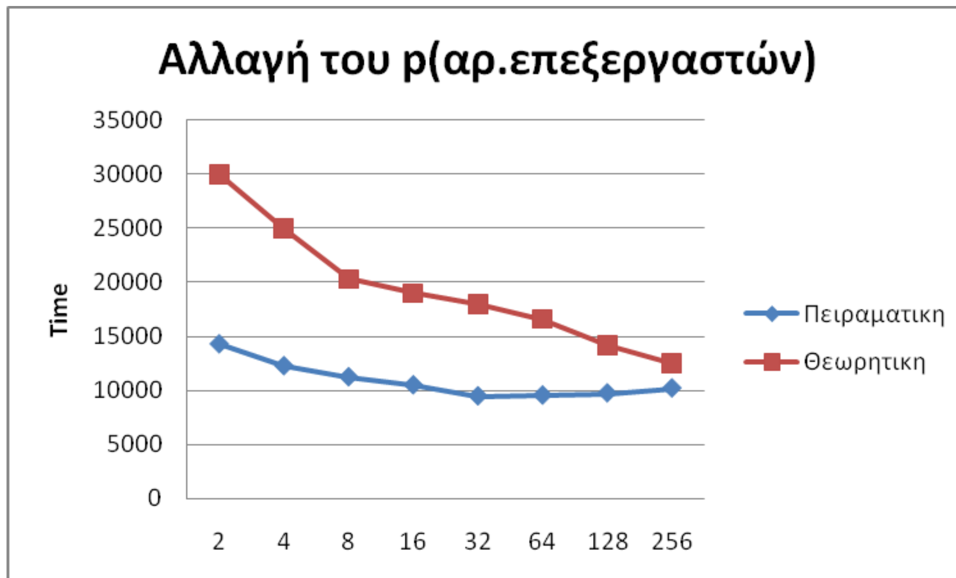
θέλουμε να ελέγξουμε στη συγκεκριμένη περίπτωση. Συγκεκριμένα δίνουμε στο p τιμές από 2 μέχρι 256, 2,8,16,32,64,128,256 αφού η πλατφόρμα μας δύσκολα μπορεί να ανταπεξέλθει σε πιο πολλούς επεξεργαστές. Σίγουρα θα προτιμούσαμε περισσότερες μετρήσεις για να είμαστε πιο βέβαιοι για τα αποτελέσματα μας αλλά δυστυχώς δεν ήταν εφικτό. Αφού πήραμε τις πειραματικές μετρήσεις μας, τότε υπολογίσαμε για τις αντίστοιχες τιμές τα θεωρητικά αποτελέσματα σύμφωνα με την συνάρτηση logn/logp που προέκυψε από την ανάλυση. Πιο κάτω φαίνονται τόσο οι πειραματικές όσο οι θεωρητικές τιμές μας μαζεμένες στο πίνακα Πίνακας 4.1.

Αριθμός Επεξεργαστών	Πειραματικός Χρόνος	Θεωρητικός Χρόνος
2	14292	10000
4	12274	5000
8	11248	3330
16	10504	2500
32	9479	2000
64	9557	1600
128	9753	1400
256	10211	1250

Πίνακας 4.1

Αξίζει να σημειωθεί ότι ο θεωρητικός χρόνος που παρουσιάζεται στον πίνακα μας είναι όλες οι τιμές πολλαπλάσια του 1000. Δηλαδή όλες οι τιμές που προέκυψαν από logn/logp είναι x1000 μόνο και μόνο για πρακτικούς λόγους, έτσι ώστε να μπορούν να μπουν και τα δύο γραφήματα σε μια γραφική για να γίνει η σύγκριση γιατί τα αποτελέσματα από την συνάρτηση ήταν πάρα πολύ μικρά και σε σύγκριση με τα πειραματικά δεν θα διακρίνονταν πάνω στην γραφική μας.

Η πιο κάτω γραφική παράσταση (Γραφική 4.1) αναπαριστά την μεταβολή του χρόνου εκτέλεσης στην περίπτωση που μεταβάλλεται ο αριθμός των επεξεργαστών και διατηρείται ίδιος ο αριθμός των στοιχείων.



Γραφική 4.1

Παρατηρήσεις

Αρχικά παρατηρούμε ότι τόσο στο θεωρητικό όσο και στο πειραματικό γράφημα όσο αυξάνεται ο αριθμός των επεξεργαστών που δουλεύουν ταυτόχρονα, δηλαδή μεγαλώνει το p τόσο μειώνεται ο χρόνος που χρειάζεται για να λυθεί το πρόβλημα μας. Αυτή η παρατήρηση προκύπτει τόσο από τις γραφικές αλλά και από τον πίνακα που δημιουργήθηκε. Αυτό το φαινόμενο ήταν αναμενόμενο όσον αφορά την θεωρητική συμπεριφορά της παράστασης γιατί ο ασυμπωτικός χρόνος εκτέλεσης του αλγορίθμου είναι λογαριθμικά αντιστρόφως ανάλογος του αριθμού των επεξεργαστών. Έτσι αναμένουμε την πτώση του χρόνου όσο αυξάνεται το p. Είναι αναμενόμενο και κοινή λογική θα μπορούσε να πει κανείς ότι για μεγαλύτερο αριθμό επεξεργαστών να χρειάζεται λιγότερος χρόνος για την εκτέλεση του αλγορίθμου.

Μια δεύτερη και εξίσου σημαντική παρατήρηση που πρέπει να σημειώσουμε είναι ότι το πειραματικό γράφημα έχει την μορφή και καμπύλη που έχει το θεωρητικό γράφημα. Ακολουθούν και τα δύο την ίδια μορφή και αυτό είναι πολύ σημαντικό γιατί συμπεράνουμε ότι ο αλγόριθμος που υλοποιήσαμε ικανοποιεί τις προσδοκίες που έχουμε όσον αφορά τον χρόνο που αναμέναμε να πάρουμε. Βλέπουμε ότι και τα δύο γραφήματα έχουν μια μείωση στις τιμές όσον αυξάνεται ο αριθμός p η οποία δεν είναι γραμμική αλλά με καμπύλη, δηλαδή δεν μειώνεται με σταθερό ρυθμό.

Μια τελευταία παρατήρηση που μπορούμε να κάνουμε είναι ότι σε καμία περίπτωση δεν παίρνουμε ακριβώς τις ίδιες τιμές που μας δίνει η αναμενόμενη συνάρτηση. Τα νούμερα στην πειραματική αξιολόγηση είναι πολύ μεγαλύτερα και αυτό οφείλεται όπως θα αναλυθεί και πιο κάτω με μεγαλύτερη λεπτομέρεια στο ότι στον προσομοιωτή γίνονται και κάποιοι

υπολογισμοί επιπλέον για την επίλυση του αλγορίθμου και σε καμία περίπτωση δεν μπορούν να θεωρηθούν ως μια μονάδα χρόνου και να αγνοηθούν ως σταθερά.

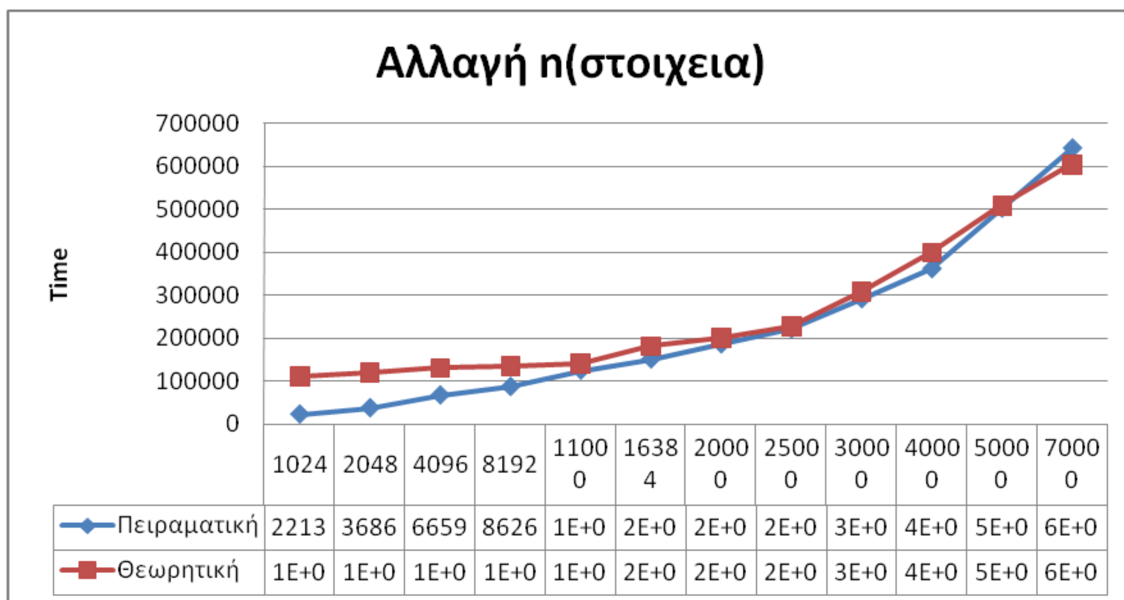
Για την δεύτερη περίπτωση που έγιναν πειραματικές μετρήσεις είναι όταν ο αριθμός των επεξεργαστών μένει σταθερός και δεν αλλάζεται σε καμία περίπτωση μέτρησης και συγκεκριμένα έγιναν μετρήσεις με p=2, δηλαδή δύο επεξεργαστές, δούλευαν ταυτόχρονα για να λυθεί το πρόβλημα. Άλλαζε όμως ο αριθμός των στοιχείων σε κάθε μέτρηση, και αυτό γίνεται για να παρατηρήσουμε πως διαμορφώνεται ο χρόνος καθώς αυξάνονται τα στοιχεία που πρέπει να μελετήσουμε. Στη συγκεκριμένη περίπτωση δεν μας ενδιέφεραν ο περιορισμός να μην χρησιμοποιούμε πολλούς επεξεργαστές και έτσι δοκιμάσαμε τα όρια της μνήμης που έχουμε στην διάθεση μας αφού πειραματιστήκαμε για μεγάλους αριθμούς δεδομένων. Συγκεκριμένα είχαμε πίνακα που είχε αποθηκευμένους 1024 ακέραιους αριθμούς, μέχρι πίνακα με 90000 ακέραιους που έπρεπε να διερευνηθούν και να εντοπίσουν την θέση ενός συγκεκριμένου στοιχείου. Και πάλι τις πειραματικές μετρήσεις ακολούθησαν οι αντίστοιχες θεωρητικές που προέκυψαν και πάλι από την συνάρτηση logn/logp. Μετά από την συλλογή των δεδομένων, σειρά είχε η δημιουργία του ακόλουθου πίνακα που παρουσιάζει και τα δύο αποτελέσματα. Ο πίνακας 4.2 παρουσιάζεται πιο κάτω:

Αριθμός στοιχείων(n)	Πειραματικές Μετρήσεις	Θεωρητικές Μετρήσεις
1024	14292	1000000
2048	22138	1100000
4096	36862	1200000
8192	66599	1300000
11000	86262	1342000
16384	124269	1399000
20000	150331	1428000
25000	185328	1460000
30000	220340	1487000
40000	290651	1528000
50000	360648	1560000
70000	501395	1609000
90000	641395	1645000

Πίνακας 4.2

Και πάλι πρέπει να σημειώσουμε ότι τα θεωρητικά αποτελέσματα δεν είναι ακριβώς αυτά που πήραμε αλλά έγινε κάποια επεξεργασία σε όλα όμως η ίδια, έτσι ώστε να έχουμε την ευκαιρία να μπορούμε να τα συγκρίνουμε στην ίδια γραφική παράσταση. Τα αποτελέσματα που πήραμε έχουν πολλαπλασιαστεί x100000 και είναι αυτά που φαίνονται στον πίνακα μας για τις θεωρητικές μετρήσεις.

Τελευταίο βήμα αυτής της διαδικασίας είναι και πάλι η δημιουργία της γραφικής παράστασης Γραφική 4.2 με τα δύο γραφήματα, θεωρητικό και πειραματικό αντίστοιχα που μας παρουσιάζει πως μεταβάλλεται ο χρόνος όσο αυξάνεται ο αριθμός των στοιχείων που έχουμε να μελετήσουμε.



Γραφική 4.2

Παρατηρήσεις

Αρχικά και πάλι παρατηρώντας τόσο τον πίνακα όσο και τη γραφική που περιλαμβάνει τα δυο γραφήματα βλέπουμε ότι την τιμή του χρόνου για κάθε αριθμού n. Παρατηρούμε ότι για ίδιο αριθμό επεξεργαστών ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται με μια μικρή λογαριθμική τάση καθώς αυξάνεται ο αριθμός των στοιχείων, πλήθος δεδομένων. Η λογαριθμική αυτή αύξηση και πάλι δικαιολογείται και είναι αναμενόμενη, γιατί σύμφωνα με

την συνάρτηση logn/logp ο ασυμπωτικός χρόνος εκτέλεσης του αλγορίθμου αναζήτησης εξαρτάται ευθέως ανάλογα από το πλήθος των δεδομένων εισόδου. Έτσι παίρνουμε αναμενόμενα αποτελέσματα. Επίσης, είναι λογική η αύξηση του χρόνου που παρατηρούμε και στο πειραματικό και στο θεωρητικό γράφημα γιατί, όσο αυξάνεται ο αριθμός των δεδομένων είναι λογικό να χρειάζεται περισσότερος χρόνος να ολοκληρωθεί ο αλγόριθμος αφού διατηρούμε σταθερό τον αριθμό των επεξεργαστών.

Επιπλέον, η παρατήρηση που έχουμε είναι ότι και πάλι τα δύο γραφήματα έχουν παρόμοια μορφή, δηλαδή συμπεριφέρονται με τον ίδιο τρόπο. Αυτό σημαίνει ότι ο αλγόριθμος που υλοποιήσαμε είναι στα βήματα του θεωρητικού αλγορίθμου που αναλύσαμε. Αυτό είναι πολύ θετικό και επιβεβαιώνει τις υποψίες μας.

Η τελευταία παρατήρηση που έχουμε να κάνουμε και πάλι είναι για τον θεωρητικό και πειραματικό χρόνο και πάλι. Όπως μπορούμε να παρατηρήσουμε ο πραγματικός χρόνος που χρειάζεται για να ολοκληρωθεί η εκτέλεση του προγράμματος, είναι πολύ μεγαλύτερος από τον θεωρητικό χρόνο. Το φαινόμενο αυτό που παρατηρείται είναι δικαιολογημένο αφού ο θεωρητικός χρόνος είναι ασυμπωτικός, δηλαδή δεν λαμβάνει υπόψη του ένα σύνολο από σταθερές, Αν και σε αυτές οι σταθερές δεν επηρεάζουν τον ασυμπτωτικό χρόνο εκτέλεσης, παίζουν ρόλο στον καθορισμό του πραγματικού χρόνου εκτέλεσης. Επιπλέον όπως αναφέρθηκε πιο πάνω η υλοποίηση του αλγορίθμου γίνεται σε CREW PRAM μοντέλο. Το γεγονός αυτό έχει ως αποτέλεσμα μια επιπλέον καθυστέρηση στην εκτέλεση του προγράμματος, αφού υπάρχει ταυτόχρονη ανάγνωση της ίδιας κυψελίδας που απαιτεί κάποιον επιπλέον χρόνο.

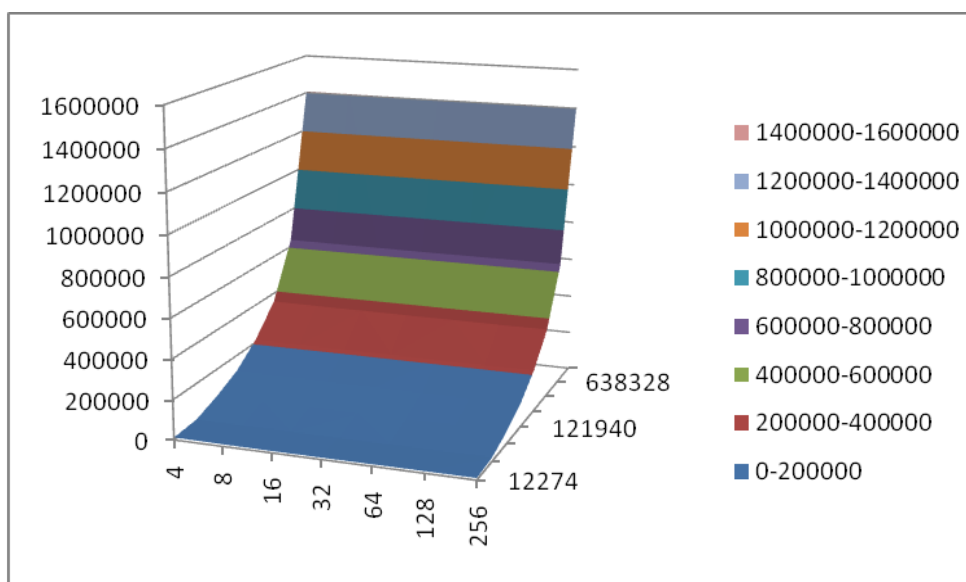
Το τρίτο και τελευταίο πείραμα που έγινε για τον συγκεκριμένο αλγόριθμο είχε την ιδιαιτερότητα ότι τίποτα δεν ήταν σταθερό αλλά σε κάθε μέτρηση μεταβαλλόταν τόσο ο αριθμός των επεξεργαστών όσο και ο αριθμός των στοιχείων. Και στις δύο περιπτώσεις αυξάναμε ταυτόχρονα τις δύο μεταβλητές μας, χωρίς κανένα σταθερό σημείο. Συγκεκριμένα ο αριθμός των επεξεργαστών που κυμανθήκαμε είναι από 4 μέχρι 512 και ποιο συγκεκριμένα, 4,8,16,32,64,128,256 και 512 λόγω του περιορισμού και πάλι να μην μπορούμε να χρησιμοποιούμε πιο πολλούς επεξεργαστές. Ο αριθμός των δεδομένων που ελέγξαμε ήταν από 1024 μέχρι 200000 και συγκεκριμένα για 8 διαφορετικά που είναι: 1024,2048,8192,16384,20000,50000,90000,200000. Στόχος μας ήταν να δούμε πως θα συμπεριφερθεί σε αυτή την περίπτωση ο αλγόριθμος μας. Πιο κάτω ακολουθά ο πίνακας Πίνακας4.3 με τις μετρήσεις που πήραμε:

Αριθμός επεξεργαστών	N=1024	N=2048	N=8192	N=16384	N=20000	N=50000	N=90000	N=200000

4	12274	19762	63847	121940	217258	357576	638328	1408653
8	12248	18737	62490	120200	215465	355475	636280	1406550
16	10504	17702	61469	119172	214423	354468	635235	1405566
32	9479	17643	60830	118140	213476	354261	634627	1404647
64	9557	16756	60621	118286	213630	353642	633709	1403655
128	9753	16931	59970	117308	213558	354085	634000	1404159
256	10211	17438	60453	117808	213096	353106	634366	1404885
512	10477	18548	61538	118881	214154	354191	634231	1404162

Πίνακας 4.3

Τα δεδομένα στη συγκεκριμένη περίπτωση είναι όπως πάρθηκαν από τα πειράματα χωρίς να γίνει καμία τροποποίηση για τις διάφορες τιμές που ελέγχθηκαν, δηλαδή δεν έγινε κάποιος πολλαπλασιασμός για σκοπούς ευκολίας. Πιο κάτω παρουσιάζεται η γραφική Γραφική4.3 παράσταση που προέκυψε μέσα από τις ποικίλες μετρήσεις που έγιναν.



Γραφική 4.3

Παρατηρήσεις

Τα πειράματα χωρίς κανένα σταθερό σημείο και η δημιουργία της τρισδιάστατης γραφικής παράστασης που προέκυψε, απλά μας επιβεβαιώνουν τα αποτελέσματα που είχαμε και παρατηρήσεις που κάναμε σε πιο πάνω πειράματα και γραφικές. Δηλαδή ότι όσον αυξάνεται ο αριθμός των δεδομένων τόσο αυξάνεται και ο χρόνος που χρειάζεται να λυθεί το πρόβλημα για τον ίδιο αριθμό επεξεργαστών. Επίσης ότι για το ίδιο πλήθος στοιχείων αν δουλεύουν περισσότεροι επεξεργαστές μαζί τότε το πρόβλημα λύνεται πιο γρήγορα. Οι μικρότερες τιμές στο χρόνο παρατηρούνται κυρίως πιο εύκολα και από τον πίνακα όταν έχουμε μικρό αριθμό δεδομένων και μεγαλύτερο αριθμό επεξεργαστών που φτάνει σχεδόν το σταθερό χρόνο για να ολοκληρωθεί ο αλγόριθμος μας και να δοθεί το αποτέλεσμα που αναμέναμε.

Κεφάλαιο 5

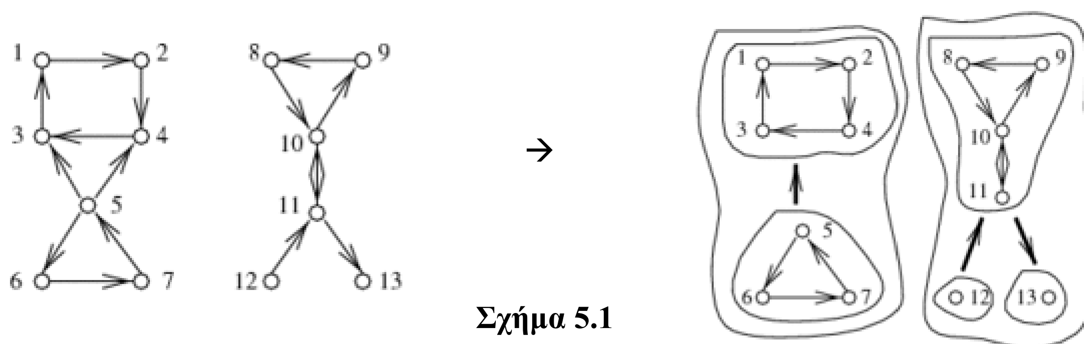
Συνεκτικές Συνιστώσες

5.1 Πρόβλημα	52
5.2 Σειριακός Αλγόριθμος	53
5.3 Παράλληλος Αλγόριθμος	57
5.4 Παραλλαγή Αλγορίθμου Συνεκτικών Συνιστωσών	66
5.5 Σύγκριση Αλγορίθμων	70

Το πρόβλημα των συνεκτικών συνιστωσών που ασχολείται με γράφους αποτελεί ένα ήδη διαδομένο και δομημένο πρόβλημα στην Επιστήμη της Πληροφορικής, αλλά δεν υπάρχουν πολλοί τρόποι υλοποίησης του, στον σειριακό υπολογισμό. Αυτό και μόνο καθιστά ακόμη μεγαλύτερη δυσκολία την δημιουργία και υλοποίηση του στον παράλληλο υπολογισμό και αυτό είναι λόγω του ότι ασχολείται με γράφους και ο τομέας αυτός στην παράλληλη επεξεργασία χρειάζεται ακόμη μεγάλα βήματα για να είμαστε ικανοποιημένοι. Το πρόβλημα αυτό είναι το κύριο κομμάτι της Διπλωματικής Εργασίας αυτής, αφού είναι αυτό που μελέτησα και στην συνέχεια υλοποίησα παράλληλα στην πλατφόρμα XMT με την βοήθεια της γλώσσας XMTC. Στόχος στη συγκεκριμένη περίπτωση δεν ήταν η εξοικείωση με την πλατφόρμα αφού αυτό φροντίσαμε να γίνει πιο πριν για να μπορώ να ανταπεξέλθω σε ένα απαιτητικό πρόβλημα σαν και αυτό, αλλά στόχος στην προκειμένη περίπτωση είναι να δούμε πως συμπεριφέρεται ο αλγόριθμος και να τον συγκρίνουμε με μια διαφορετική παραλλαγή που επίσης υλοποιήθηκε. Με την υλοποίηση και των δύο θα μπορούμε να συγκρίνουμε τόσο τον χρόνο αλλά και το κόστος που μας ενδιαφέρει ιδιαίτερα.

5.2 Πρόβλημα

Το πρόβλημα των συνεκτικών συνιστωσών ενός γράφου, είτε αυτός είναι κατευθυνόμενος είτε όχι, αποτελεί ένα από τα πιο θεμελιώδη προβλήματα στη θεωρία γράφων. Ουσιαστικά αυτό που έχουμε να κάνουμε είναι δοθέντος ως είσοδο έναν γράφο G=(V,E) όπου V οι κορυφές του γράφου, |V|=n, και E οι ακμές του, |E|=m, είναι να δημιουργήσουμε ένα υπό-γράφο όπου οποιεσδήποτε δύο κορυφές είναι συνδεδεμένες η μια με την άλλη μέσα από μονοπάτια και η οποία δεν είναι ενωμένη με κάποια επιπλέον κορυφή του υπό-γραφήματος. Για να γίνει περισσότερο κατανοητό, παρακάτω παρουσιάζεται ένα παράδειγμα στο σχήμα 5.1 όπου στα αριστερά έχουμε την είσοδο του προβλήματος και δεξιά την έξοδο που αναμένουμε.



Σχήμα 5.1

Αυτός ο γράφος αποτελείται από δύο συνεκτικές συνιστώσες που φαίνονται στους κύκλους. Εμείς στο πρόβλημα μας θα ασχοληθούμε με μη-κατευθυνόμενους γράφους και πιο συγκεκριμένα ορίζουμε ότι η συνεκτική συνιστώσα που προκύπτει από μη κατευθυνόμενους γράφους, είναι αν μια κορυφή v είναι προσβάσιμη από μια κορυφή u αν υπάρχει μονοπάτι από το v στο u μήκους 0. Δηλαδή κάποιος κόμβος αποτελεί τον αρχηγό ενός συνόλου από κόμβους και είναι πολλοί αυτοί που έχουν κοινό και όχι ο καθένας ξεχωριστό. Ουσιαστικά θα μπορούσαμε να πούμε ότι είναι ένα σύνολο από κόμβους που είναι όλοι συνδεδεμένοι μεταξύ τους και δημιουργούνται νέα υπό-γραφήματα που λύνουν το πρόβλημα των συνεκτικών συνιστωσών.

5.2 Σειριακός Αλγόριθμος

Το πρόβλημα των συνεκτικών συνιστωσών αποτελεί ένα πρόβλημα γράφων και όσον αφορά τον σειριακό υπολογισμό το συναντούμε σχεδόν καθημερινά, θα μπορούσε να πει κανείς χωρίς να το γνωρίζουμε όμως. Το πρόβλημα αυτό είναι βασικό κομμάτι, στην καρδία, πολλών εφαρμογών που ασχολούνται με τους γράφους καθώς και παιχνιδιών. Για

παράδειγμα σε παιχνίδια και εφαρμογές που γίνεται ταξινόμηση αντικειμένων με κάποια κοινά χαρακτηριστικά που μπορούμε να τα ονομάσουμε ¨όμοια¨. Το πρόβλημα της συνεκτικής συνιστώσας στην προκειμένη περίπτωση εντοπίζεται όταν θα πρέπει να κατηγοριοποιήσουμε διάφορα αντικείμενα σε μια κατηγορία, έτσι και εμείς θα πρέπει να ταξινομήσουμε τις κορυφές μας σε κάποια άλλη κορυφή που θα είναι ο αρχηγός. Επίσης προβλήματα που περιέχουν τον αλγόριθμο συνεκτικών συνιστωσών είναι προβλήματα 2-satisfiability τα οποία είναι δυαδικά προβλήματα, δηλαδή οι μεταβλητές τους παίρνουν τιμές 0 ή 1 αναλόγως. Τέτοια προβλήματα ελέγχουν σε γραμμικό χρόνο αν είναι αληθές κάποιες ποσοτικές συναρτήσεις που δέχονται μόνο τιμές 0 και 1. Ένα άλλο πρόβλημα που λύνεται με την βοήθεια των συνεκτικών συνιστωσών, είναι προβλήματα τα οποία μας λένε αν ναι ή όχι το ταίριασμα των ακμών μπορούν να είναι μέρος ενός τέλειου ταιριάσματος. Επίσης το πρόβλημα συνεκτικών συνιστωσών τα τελευταία χρόνια βρίσκει μεγάλη εφαρμογή και σε προβλήματα ανάλυσης εικόνας. Ένας τομέας που αναπτύσσεται γρήγορα τα τελευταία χρόνια με την γενική πρόοδο της τεχνολογίας, άρα είναι πολύ σημαντικό να έχουμε καλούς αλγόριθμους συνεκτικών συνιστωσών για να είναι πιο εύκολη η πρόοδος του τομέα αυτού. Μια μικρή εφαρμογή του αλγορίθμου αυτού εντοπίζεται και στον μέσο κοινωνικής δικτύωσης ¨Facebook¨.

Όπως αναφέρθηκε υπάρχουν κάποιοι αλγόριθμοι που μπορούν και λύνουν το πρόβλημα αυτό αλλά οι τρεις δημοφιλέστεροι που μας δίνουν λύση είναι ο Kosaraju's αλγόριθμος[17], ο Tarjan's συνεκτικών συνιστωσών αλγόριθμος[16] και ο path-based συνιστωσών αλγόριθμος[18]. Και οι τρεις αυτοί αλγόριθμου λύνουν το πρόβλημα σε γραμμικό χρόνο ως προς το πλήθος των κορυφών και το πλήθος των ακμών που είναι ότι καλύτερο μπορούμε να επιτύχουμε. Εμείς θα προσπαθήσουμε να κατανοήσουμε και να εξηγήσουμε τον πρώτο, Kosaraju αλγόριθμο αλλά θα γίνει μια μικρή αναφορά και στους υπόλοιπους δύο.

Ο Tarjan's αλγόριθμος δημοσιεύθηκε από τον Robert Tarjan το 1972 και έχει την βάση του στην αναζήτηση depth first η οποία γίνεται μια και μόνο φορά. Διατηρεί μια στοίβα των κορυφών που έχουν διερευνηθεί από την αναζήτηση, αλλά δεν έχουν εκχωρηθεί σε ένα στοιχείο, και υπολογίζει μια μεταβλητή για κάθε κορυφή που είναι ο αριθμός του δείκτη με το υψηλότερο προσβάσιμο πρόγονο σε ένα βήμα από έναν απόγονο της κορυφής. Η μεταβλητή αυτή χρησιμοποιείται για να διαπιστωθεί πότε ένα σύνολο κορυφών πρέπει να μετακινηθεί από τη κορυφή σε μια νέα συνιστώσα. Ο path-based συνιστωσών αλγόριθμος έχει την βάση του στον Tarjan αλγόριθμο αφού γίνεται αναζήτηση, αλλά χρησιμοποιούνται δύο στοίβες σε αυτόν. Η μια στοίβα χρησιμοποιείται για να αποθηκεύονται οι κορυφές που δεν ταξινομήθηκαν κάπου ακόμα ενώ η άλλη για να αποθηκεύει την ροή όσων έχουν ενταχθεί σε μια ομάδα. Ο γραμμικός αυτός αλγόριθμος δόθηκε πρώτη φορά στην δημοσιότητα από τον Edsger W. Dijkstra το 1976. Ο Kosaraju αλγόριθμος ο οποίος είναι και

αυτός που θα ασχοληθούμε στηρίζεται σε δύο αναζητήσεις depth first. Η πρώτη αναζήτηση γίνεται στο πρώτο γράφημα και χρησιμοποιείται για να επιλεχτεί τη σειρά με την οποία η δεύτερη depth-first αναζήτηση έχει επισπευτεί τις κορυφές και αναδρομικά τους διερευνά εάν όχι. Η δεύτερη depth first αναζήτηση γίνεται στο αρχικό γράφημα και γίνεται για να οριστεί ανάποδα το γράφημα και σε κάθε επανάληψη βρίσκεται η ισχυρότερη κορυφή. Ο αλγόριθμος αυτός αναλύεται με μεγαλύτερη λεπτομέρεια πιο κάτω. Ο S. Rao Kosaraju ήταν ο δημιουργός του το 1978 αλλά δεν τον δημοσίευσε ποτέ και το 1981 δημοσιεύτηκε από τον Micha Sharir.

Ας εξηγήσουμε με μεγαλύτερη λεπτομέρεια τον σειριακό αλγόριθμο που είναι πιο διαδεδομένος. Ο αλγόριθμος βασίζεται στην τεχνική depth first αναζήτησης, έτσι πρώτα πρέπει να κατανοήσουμε αυτό. Η τεχνική depth first ενός γράφου G=(V,E) είναι μια συμμετρική μέθοδος που επισκέπτεται όλες τις ακμές του γράφου G, ξεκινώντας με την κορυφή v϶V και κάνει χρήση ακμών από το G.Πιο συγκεκριμένα περιγράφεται ως μια μεταβλητή k που υποδηλώνει την τιμή των συνεκτικών συνιστωσών κάθε φορά. Αρχικά ξεκινάμε με k=1. Η διαδικασία αναζήτησης εφαρμόζεται στην κορυφή v ξεκινώντας με το να επισπευτώ την v και να της ορίσω την τιμή που έχει η μεταβλητή k. Προχωρούμε με το να επιλέξουμε μια ακμή w από το σύνολο των ακμών που δεν έχει επισπευτεί ακόμη και συνδέεται με την v και εφαρμόζουμε και σε αυτή την μέθοδο αναζήτησης. Όταν τερματίσει σε κάποια στιγμή η μέθοδος αναζήτησης που γίνεται στην w, επιστρέφουμε πίσω στην κορυφή v και προχωρούμε με το να εφαρμόσουμε την μέθοδο αναζήτησης σε μια κορυφή που δεν επισκέφθηκε ακόμη και είναι συνδεδεμένη επίσης με την v. Εάν δεν υπάρχει καμία τέτοια κορυφή η διαδικασία αναζήτησης τερματίζει στο v. Τότε αυξάνουμε την μεταβλητή μας k σε k+1 και συνεχίζουμε περνώντας μια τυχαία κορυφή v’ που δεν έχει επισκεφθεί ακόμη και ανήκει στο σύνολο των κορυφών. Η ίδια διαδικασία επαναλαμβάνεται τώρα στην κορυφή v’. Ο αλγόριθμος τερματίζει όταν όλες οι κορυφές έχουν επισκεφθεί. Μετά από αυτή την διαδικασία οι κορυφές που ανήκουν στην ίδια συνεκτική συνιστώσα έχουν την ίδια τιμή. Πιο κάτω ακολουθεί ο ψευδοκώδικας του αλγορίθμου αυτού όπου οι κορυφές που έχουν επισκεφθεί σηματοδοτούνται με την βοήθεια μιας στοίβας S .

Let G be a directed graph and S be an empty stack.

1. While S does not contain all vertices:

Choose an arbitrary vertex v not in S.

Perform a depth-first search starting at v.

Each time that depth-first search finishes expanding a vertex u, push u onto S.

Reverse the directions of all arcs to obtain the transpose graph.

2.While S is nonempty:

Pop the top vertex v from S.

Perform a depth-first search starting at v in the transpose graph.

The set of visited vertices will give the strongly connected component containing v; record this and remove all these vertices from the graph G and the stack S.

Αξίζει να σημειωθεί επίσης ότι μπορούμε να χρησιμοποιήσουμε breadth-first search (BFS) αλγόριθμο στην εντολή 2 αντί του depth-first που χρησιμοποιείται που δεν επηρεάζει σε κάτι. Η ορθότητα του αλγορίθμου πηγάζει από την συνάρτηση αναζήτησης που χρησιμοποιούμε δηλαδή την depth-first. Η συνάρτηση αυτή μας δίνει την ευκαιρία να είμαστε σίγουροι ότι το πρόβλημα μας λύνεται σωστά γιατί επισκέπτονται όλες οι ακμές του γραφήματος και έτσι δίνεται η ανάλογη τιμή του υπερκόμβου που πρέπει. Έτσι είμαστε σίγουροι ότι δεν θα μείνει κάποιος κόμβος χωρίς να ανήκει σε κάποια κατηγορία-οικογένεια.

Χρόνος-Πολυπλοκότητα:

Μας ενδιαφέρει όπως και σε προηγούμενο αλγόριθμο ο σειριακός χρόνος που επιλύεται το πρόβλημα μας γιατί αποτελεί δείκτη και μέτρο σύγκρισης και για τον παράλληλο χρόνο και πιο ειδικά για το παράλληλο κόστος. Έτσι πρέπει μέσα από την ανάλυση του αλγορίθμου με εξηγήσαμε πιο πάνω να εντοπίσουμε την χρονική πολυπλοκότητα του αλγορίθμου που δεν είναι άλλος από τον χειρότερο χρόνο που χρειάζεται να λυθεί το πρόβλημα. Ο αλγόριθμος αυτός λύνει το πρόβλημα βέλτιστα σε χρόνο O(n+m), όπου |V|=n το πλήθος των κορυφών και |E|=m το πλήθος των ακμών. Η χρονική πολυπλοκότητα έχει να κάνει και πάλι με τον αλγόριθμο αναζήτησης που χρησιμοποιούμε. Ο αλγόριθμος αυτός παίρνει χρόνο O(n+m) για νε εκτελεστεί, εμείς τον καλούμε δύο φορές σύμφωνα με τον ψευδοκώδικα μας και άρα έχουμε O(2*(n+m)) το οποίο ισούται με O(n+m) και άρα κρατάμε αυτό μέχρι στιγμής. Επίσης παρατηρούμε ότι σε κάθε εκτέλεση της επανάληψης γίνονται απλές πράξεις όπως την σηματοδότηση ότι επισκέφτηκε κάποια κορυφή στο πρώτο βρόχο και στον δεύτερο την έξοδο από την στοίβα κάποιας κορυφής επίσης. Οι λειτουργίες αυτές παίρνουν χρόνο Θ(1) και άρα μέσα από το γινόμενο προκύπτει ότι ο σειριακός χρόνος είναι της τάξεως

T*(n) = O(|V|+|E|)

δηλαδή O(n+m).

Το πρόβλημα μας λύνεται γραμμικά συνάρτηση του αριθμού των ακμών και των κορυφών.

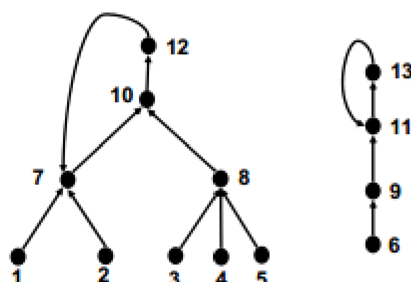
Αποτελεί επίσης κάτω φράγμα για την επίλυση του αλγορίθμου αυτού αφού οποιαδήποτε λύση πρέπει να ελέγξει όλο το σύνολο των ακμών και κορυφών για να δοθεί σωστό αποτέλεσμα. Άρα, μπορούμε να πούμε με βεβαιότητα ότι ο χρόνος αυτός είναι βέλτιστος και ότι καλύτερο μπορούμε να επιτύχουμε σε θέμα χρόνο μιλώντας πάντα για την σειριακή λύση που μας επιστρέφει τις συνεκτικές συνιστώσες ενός αρχικού γράφου. Θεωρητικά είναι ότι

καλύτερο μπορούμε να έχουμε αλλά στην πράξη τα αποτελέσματα δεν είναι και τόσο καλά όσο τα αναμένουμε ειδικά όταν έχουμε ισχυρά συνδεδεμένους γράφους που απαιτείται πολύ δουλειά και μεγαλύτερος χρόνος. Εμείς όμως θα ασχοληθούμε με την θεωρητική ανάλυση στη συγκεκριμένη περίπτωση και λέμε ότι καλύτερος σειριακός χρόνος είναι O(|V|+|E|).

5.3 Παράλληλος Αλγόριθμος

Στόχος μας γενικά σε αυτή την Διπλωματική Εργασία είναι η κατανόηση και υλοποίηση παράλληλων αλγορίθμων. Το πρόβλημα των συνεκτικών συνιστωσών είναι ένα αρκετά περίπλοκο και απαιτητικό πρόβλημα και ειδικά ο παράλληλος υπολογισμός του. Βάση του παράλληλου αλγορίθμου συνεκτικών συνιστωσών είναι και σε αυτή την περίπτωση ο σειριακός αλγόριθμος και πιο συγκεκριμένα η αναζήτηση και γι΄ αυτό τον λόγο αναλύθηκαν και επεξηγήθηκαν με μεγάλη λεπτομέρεια πιο πάνω. Είναι σημαντικό να κατανοήσουμε τον σειριακό αλγόριθμο για να μπορούμε να ανταπεξέλθουμε πιο εύκολα σε αυτό, το παράλληλο. Είσοδο και αυτού του προβλήματος αποτελεί και πάλι ένας γράφος G=(V,E) όπου V το σύνολο των κορυφών και E το σύνολο των ακμών. Υπάρχουν δύο κύριοι τρόποι που μπορούμε να αναπαραστήσουμε ένα γράφο. Είτε ως λίστα γειτνίασης, είτε ως πίνακα γειτνίασης. Εμείς στο παρόν πρόβλημα μας θα αναπαριστούμε τον γράφο μας σαν πίνακα γειτνίασης όπου είναι ένας δισδιάστατος πίνακας A μεγέθους n*n,όπου n ο αριθμός των κόμβων. Ο πίνακας αυτός παίρνει τις τιμές 0 ή 1 αναλόγως αν συνδέει ακμή της δύο κορυφές που αντιστοιχούν σε κάθε σειρά ή στήλη. A[u,v]=1 αν και μόνο αν υπάρχει ακμή που ενώνει τις δύο κορυφές, και επειδή ασχολούμαστε με μη κατευθυνόμενα γραφήματα τότε αναμένουμε αν A[u,v]=1τότε και A[v,u]=1. Δηλαδή έχουμε μια συμμετρία στον πίνακα μας όσον αφορά τις τιμές που έχει. Πριν να ξεκινήσουμε να δίνουμε τα λεπτομερή βήματα του αλγορίθμου θα εισάγουμε κάποιες έννοιες οι οποίες θα μας φανούν χρήσιμες.

Ψευδοδάσος: Στη θεωρία γραφημάτων, ένα Ψευδοδάσος είναι ένα μη-κατευθυνόμενο γράφημα στο οποίο κάθε συνεκτική συνιστώσα έχει το πολύ ένα κύκλο. Είναι ένα σύστημα από κορυφές και ακμές που συνδέουν ζεύγη κορυφών, τέτοιο ώστε δεν υπάρχουν δύο κύκλοι διαδοχικών ακμών να μοιράζονται οποιαδήποτε κορυφή με κάποια άλλη ακμή. Ουσιαστικά είναι ένας κατευθυνόμενος γράφος όπου κάθε κόμβος έχει εξερχόμενο βαθμό μικρότερο ή ίσο με 1. Πιο κάτω ακλουθεί παράδειγμα στο



Σχήμα5.2 για περισσότερη κατανόηση.

Σχήμα 5.2[8]

Το ερώτημα είναι πως μπορούμε να δημιουργούμαι ένα ψευδοδάσος. Για αυτό υπάρχει μια αυθαίρετη συνάρτηση D:V→V ορίζει ένα ψευδοδάσος (V,F) όπου F={<v,D(v)>|vєV}.

Ριζωμένα Κατευθυνόμενα Δέντρα: Ένας γράφος δεν αποτελεί δέντρο αλλά κάποιες κορυφές ενωμένες μεταξύ τους. Το ριζωμένο κατευθυνόμενο δέντρο είναι ένα γράφος όπου υπάρχει μια ρίζα που ορίζουμε εμείς και είναι το μόνο στοιχείο που δεν έχει πατέρα αλλά έχει παιδιά. Οι υπόλοιπες κορυφές έχουν κάποιον απόγονο όμως.

Ριζωμένος Αστέρας: Ένα δέντρο είναι ριζωμένος αστέρας όταν κάθε ακμή είναι ευθέως συνδεδεμένη στη ρίζα του δέντρου. Πιο συγκριμένα όταν κάθε κόμβος έχει πατέρα τη ρίζα του δέντρου.

Αυτές οι δομές είναι χρήσιμες και θα μας φανούν βοηθητικές αφού θα αναφερθούμε σε αυτές στον αλγόριθμο μας.

Η έξοδος που παίρνουμε μετά από την ολοκλήρωση του αλγορίθμου είναι ένα διάνυσμα D μεγέθους n, όπου |V|=n, τέτοιο ώστε κάθε θέση του πίνακα D να έχει τον εκπρόσωπο του γράφου συνεκτικής συνιστώσας. Για παράδειγμα η μικρότερη τιμή ενός connected component γράφου πιθανό να αντιπροσωπεύει τη συνιστώσα και στο συγκεκριμένο αυτό θα χρησιμοποιείται. Όταν έχουμε αυτό το πίνακα θα μπορούμε να απαντούμε σε O(1) χρόνο αν δυο κορυφές ανήκουν στην ίδια συνεκτική συνιστώσα. Ο αλγόριθμος αποτελείται από μια επαναληπτική διαδικασία όπου στην αρχή της κάθε επανάληψης, οι διαθέσιμες κορυφές είναι διαχωρισμένες σε γκρουπ τέτοια ώστε όλες οι κορυφές του γκρουπ να έχουν την ίδια συνεκτική συνιστώσα και κατά την διάρκεια της κάθε επανάληψης κάποια γκρουπ ταξινομούνται σε μεγαλύτερα γκρουπ όπου υπάρχει αυτή η δυνατότητα. Ο αλγόριθμος τερματίζει όταν δεν μπορούν να ενωθούν κάποια άλλα γκρουπ. Τα τελικά γκρουπ είναι ένα ριζωμένο κατευθυνόμενο δέντρο, με την ρίζα να είναι ο εκπρόσωπος της συνιστώσας.

5.3.1 Βέλτιστος Αλγόριθμος

Έστω A είναι ένας δισδιάστατος πίνακα nxn ενός μη κατευθυνόμενου γράφου G=(V,E) όπου V={1,2,…,n}. Ως εκ τούτου A[i,j]=1 εάν και μόνο εάν (i,j)єE. Ορίζουμε την ακόλουθη

συνάρτηση C έτσι ώστε V:C(v) = min{u|A(u,v)=1} και αν το v δεν έχει κάποιο γείτονα τότε C(v)=v. Η συνάρτηση αυτή ουσιαστικά μας βρίσκει τον γείτονα μας με την μικρότερη ταυτότητα και αν δεν έχουμε κάποιο γείτονα τότε δίνουμε την δική μας τιμή σαν το γείτονα με την μικρότερη ταυτότητα. Μετά από την εφαρμογή της συνάρτησης αυτής μπορούμε να πούμε ότι δημιουργείται ένα ψευδοδάσος που αποτελείται από κορυφές που αποτελούν κατευθυνόμενα δέντρα.

Ο αλγόριθμος ξεκινά με το να υπολογίσουμε την συνάρτηση C πάνω σε όλες τις κορυφές του γράφου στο σύνολο V. Οι κορυφές που ανήκουν στο ίδιο κατευθυνόμενο δέντρο του ψευδοδάσους που προκαλείται από την C είναι στην ίδια συνεκτική συνιστώσα. Ως, εκ τούτου μπορούμε να συρρικνώσουμε κάθε σύνολο Vi σε μια κορυφή, που ονομάζεται υπερκόμβος και η διαδικασία επαναλαμβάνεται στο γράφημα με τους υπερκόμβους όπου τώρα κόμβοι, είναι μόνο οι υπερκόμβοι και εκπροσωπούν τους υπόλοιπους κόμβους. Δηλαδή έχουμε υπερκόμβους Vi τέτοια ώστε (Vi,Vj) είναι μια ακμή εάν και μόνο αν υπάρχει v϶Vi και w϶Vj με την προϋπόθεση ότι (v,w)϶E. Ο αλγόριθμος τερματίζει όταν κανένας υπερκόμβος δεν μπορεί να συρρικνωθεί σε κάποια άλλη συνιστώσα. Με τον όρο συρρικνώσουμε εννοούμε ένα σύνολο Vi από κορυφές ενός συνδεδεμένου δέντρου Ti αναγνωρίζουμε ένα ειδικό vi από το Vi και τον ορίζουμε εκπρόσωπο, αυτόν με την μικρότερη ταυτότητα. Ακολούθως ορίζουμε κάθε κορυφή ποιος είναι ο εκπρόσωπος της.

Ένα άλλο θέμα που πρέπει να λύσουμε είναι πως θα ορίσουμε ποιος είναι ο εκπρόσωπος του κάθε κόμβου. Για αυτό το θέμα θα χρησιμοποιήσουμε την τεχνική του διπλασιασμού του δείκτη, χρησιμοποιώντας πάλι την συνάρτηση C, όπου ni=|Vi|. Για κάθε v϶Vi, νέα τιμή είναι ίση με μια από τις δύο ακμές για το δέντρο μας. Η τιμή που προκύπτει μέσα από την συνάρτηση min{C(v),C(C(v))} και γίνεται η ρίζα του δέντρου και διαφοροποιείται αναλόγως η συνάρτηση C έτσι ώστε ορίζεται τώρα ένα σύνολο από ριζωμένους αστέρες. Η ρίζα του κάθε δέντρου ορίζεται ως ο εκπρόσωπος του συνόλου και είναι ο υπερκόμβος. Έστω vi και vj είναι δύο υπερκόμβοι που εκπροσωπούν τα δέντρα Ti και Tj αντίστοιχα. Τότε vi και vj συρρικνώνονται σε ένα γράφημα εάν και μόνο εάν υπάρχει v϶Vi και w϶Vj τέτοιο ώστε (v,w)϶E. Το τελευταίο βήμα του αλγορίθμου είναι να αντιστραφεί η διαδικασία και κάθε κόμβος να ορίσει την συνεκτική συνιστώσα της γιατί μέχρι τώρα μόνο οι υπερκόμβοι γνώριζαν και προχωρούσαν. Αυτό γίνεται με το να επεκτείνουμε κάθε ρίζα ρ στο δικό της δέντρο σύμφωνα πάντα με την προηγούμενη επανάληψη και αναθέτουμε την τιμή στο D(v) για όλες τις κορυφές όπου είναι και η μικρότερη τιμή. Στο τέλος όλης τις διαδικασίας ο πίνακας D έχει σε κάθε θέση του τη συνεκτική συνιστώσα που του αναλογεί.

Begin

1. *A0[]=A[];*
   N0[]=N;

k=0;

2. *While(nk[]>0){*

    2.1 k=k+1;

    2.2 C(v)=min{u|Ak-1(u,v)=1, u!=v}

    If no one then C(v)=v;

    2.3 Shrink each tree of the pseudoforest defined by C to a rooted star. The root of each star containing more than one vertex defines new supervertex

    2.4 Set nk to be equal to the number of the new supervertex, and set Ak to be the nk x nk adjacency matrix of the new supervertex graph.

    }

3. *For each vertex v, determine D(v) as follows. If, at the end of step 2, C(v)=v, then set D(v)=v otherwise, reverse the process performed at step 2 by expanding each supervertex r int the set Vr of vertices of its directed tree, and making the assignment D(v)=D(r) for each vϵVr.*

End[4]

Ο αλγόριθμος που δόθηκε πιο πάνω για επίλυση του προβλήματος των συνεκτικών συνιστωσών είναι ένας από τους δύο αλγορίθμους που έχουν βρεθεί και λύνουν παράλληλα το πρόβλημα μας. Ο συγκεκριμένος αλγόριθμος όμως που παρουσιάσαμε πρέπει να αναφέρουμε ότι λύνει τι πρόβλημα μας σε common CRCW PRAM μοντέλο, δηλαδή απαιτεί τόσο ταυτόχρονο διάβασμα όσο και ταυτόχρονη γραφή από περισσότερους από ένα επεξεργαστές στην ίδια κυψελίδα μνήμης την ίδια χρονική στιγμή. Πιο συγκεκριμένα ανήκουν στη κατηγορία common δηλαδή επιτρέπεται ταυτόχρονο γράψιμο μόνο αν οι επεξεργαστές προσπαθούν να γράψουν την ίδια τιμή την ίδια στιγμή. Κάνοντας αναφορά στον αλγόριθμο αν θέλουμε να μιλήσουμε πιο συγκεκριμένα τα βήματα 1, 2.1 και 2.2 δεν έχουν κάποια ιδιαίτερη απαίτηση όσον αφορά την μνήμη γιατί ο κάθε επεξεργαστής διαβάζει από ξεχωριστή θέση στη μνήμη και γράφει επίσης σε ξεχωριστή θέση έτσι δεν μας στοιχίζει κάτι. Προχωρώντας όμως στο βήμα 2.3 παρατηρούμε ότι απαιτείται ταυτόχρονο διάβασμα στη κατασκευής του αστέρα μας που χρησιμοποιείται τεχνική διπλασιασμού του δείκτη. Στο βήμα 2.4 είναι το σημείο όπου χρειάζεται το ταυτόχρονο γράψιμο στην ίδια κυψελίδα μνήμης(concurrent write) για το κατασκεύασμα του νέου πίνακα γειτνίασης. Κλείνοντας το βήμα 3 μπορεί να γίνει χωρίς κάποια ταυτόχρονη πρόσβαση στη μνήμη, αλλά έχει ήδη οριστεί το μοντέλο που πρέπει να εφαρμοστεί για να ικανοποιείται ο αλγόριθμος.

Πολυπλοκότητα Χρόνου-Κόστους:

Θα βασιστούμε στον αλγόριθμο μας και θα προσπαθήσουμε να αναλύσουμε τόσο τον χρόνο που χρειάζεται όσο και το κόστος που υπολογίζεται μέχρι να ολοκληρωθεί ο αλγόριθμος αυτός. Στον αλγόριθμο εντοπίζουμε ότι υπάρχει μια επαναληπτική δομή, άρα δεν μπορούμε να είμαστε σίγουροι πότε θα τερματίσει, έτσι στόχος μας είναι σε αρχικό βήμα να εντοπίσουμε τον μέγιστο αριθμό επαναλήψεων που μπορούν να προκύψουν στην χειρότερη περίπτωση. Παρατηρούμε ότι το μέγεθος του προβλήματος μας σε κάθε επανάληψη μειώνεται λογαριθμικά και άρα μπορούμε να πούμε ότι στην χειρότερη περίπτωση θα χρειαστεί να κάνουμε logn επαναλήψεις, αφού ο αριθμός των αστέρων μειώνεται κατά το ήμισυ σε κάθε επανάληψη. Αφού εντοπίσαμε τον αριθμό των επαναλήψεων που απαιτείται τώρα χρειάζεται να βρούμε πόσο χρόνο παίρνει η κάθε επανάληψη για να εκτελεστεί και να υπολογίσουμε το γινόμενο τους, όπου θα είναι και ο συνολικός χρόνος που χρειάζεται ο αλγόριθμος μας. Για να υπολογίσουμε τον χρόνο που παίρνει για να γίνει μια επανάληψη πρέπει να πάρουμε ένα προς ένα τα βήματα του βρόχου επανάληψης και στην συγκεκριμένη περίπτωση του while. Θα εξετάσουμε την κοστή επανάληψη, το βήμα 2.1 παίρνει χρόνο Θ(1) και δεν το υπολογίζουμε, προχωρούμε στο βήμα 2.2 όπου υπολογίζουμε την συνάρτηση C που υπολογίζει το μικρότερο και γίνεται με την χρήση παράλληλου αλγορίθμου όπου για κάθε γραμμή παίρνει χρόνο Θ(lognk) και χρησιμοποιεί Θ(nk) επεξεργαστές, άρα συνολικά για όλες τις γραμμές χρειάζεται Θ(lognk) χρόνο και Θ(nk^2) επεξεργαστές. Προχωρώντας στο βήμα 2.3 το δέντρο μετατρέπεται σε αστέρα και χρησιμοποιείται η τεχνική διπλασιασμού του δείκτη. Η τεχνική αυτή για $\log n_{k-1}$ επαναλήψεις χρειαζόμαστε χρόνο $O(\log n_{k-1})$ και χρησιμοποιούμε $O(n\log n_{k-1})$ επεξεργαστές. Τερματίζοντας τα βήματα 2.3 έχουμε τον αριθμό n_k που είναι οι μη τετριμμένοι υπερκόμβοι και χρησιμοποιώ τεχνική prefix sum που υπολογίζεται σε $O(\log n_k)$ χρόνο χρησιμοποιώντας $O(n_k)$ επεξεργαστές. Μας απομένει να δημιουργήσουμε τον πίνακα γειτνίασης όπου ένας επεξεργαστής αναλαμβάνει μια θέση του πίνακα της επανάληψης κ-1 και γράφει 1 στον πίνακα της επανάληψη κ ανάλογος. Αφού επιτρέπουμε το ταυτόχρονο γράψιμο η διαδικασία αυτή παίρνει χρόνο O(1) και χρειάζονται $O((n_{k-1})^2)$ επεξεργαστές. Από αυτό προκύπτει ότι για κάθε επανάληψη χρειαζόμαστε O(logn) χρόνο για να ολοκληρωθεί μια επανάληψη και $O(n^2)$ επεξεργαστές. Αφού έχουμε και O(logn) επαναλήψεις τότε μπορούμε να πούμε ότι το δεύτερο βήμα μας παίρνει χρόνο $O(\log^2 n)$. Το βήμα 1 θεωρούμαι ότι παίρνει χρόνο O(1) αφού είναι απλές αρχικοποιήσεις και το βήμα 3 μπορούμε να πούμε ότι γίνεται μέσα στα όρια του χρόνου που χρειάζεται το βήμα 2 και άρα δεν προσδίδει κάποιαν επιπλέον πολυπλοκότητα στον χρόνο μας. Στο σύνολο παρατηρούμε ότι χρειάζεται $O(\log^2 n)$ χρόνος για να λυθεί το πρόβλημα.

$T_{n^2}(n) = O(\log n^2)$

Όσον αφορά το κόστος τώρα είπαμε ότι προκύπτει από το γινόμενο του παράλληλου χρόνου και του αριθμού των επεξεργαστών που εργάστηκαν ταυτόχρονα για να λυθεί το πρόβλημα μας άρα έχουμε : O(log^2(n))*O(n^2) και συνολικά.

$$C_{n^2}(n) = O(n^2 \log^2 n)$$

Ένας παράλληλος αλγόριθμος για να είναι βέλτιστος πρέπει το κόστος του να είναι της τάξεως του χρόνο του καλύτερου σειριακού αλγορίθμου. Πιο πάνω ορίσαμε το χρόνο του σειριακού σε O(|V|+|E|), αλλά στην συγκεκριμένη περίπτωση έχουμε πίνακες γειτνίασης και άρα ο χρόνος του καλύτερου σειριακού είναι $O(n^2)$. Με την πρώτη ματιά μπορούμε να πούμε ότι δεν είναι τις ίδιας τάξεως αφού ο παράλληλος έχει $\log^2 n$ περισσότερες πράξεις. Ορίζουμε τον αλγόριθμο μας βέλτιστο αφού είναι ότι καλύτερο μπορούμε να πετύχουμε για αυτό το πρόβλημα συνεκτικών συνιστωσών έτσι και αλλιώς δεν απέχει πολύ από την σειριακό χρόνο που επιλύει το πρόβλημα.[4,8]

Πειραματική Αξιολόγηση

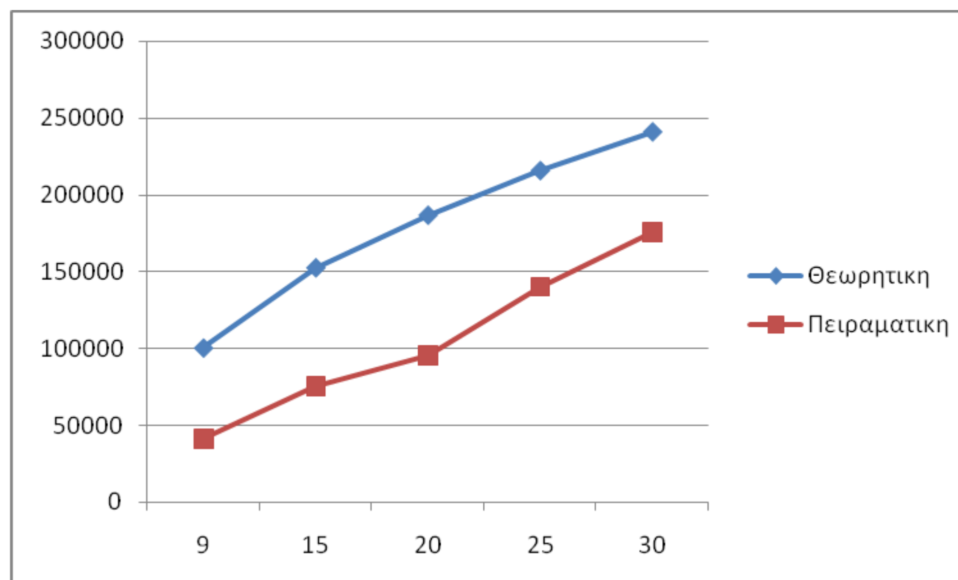
Την θεωρητική ανάλυση που έγινε πιο πάνω ακολουθεί η πειραματική αξιολόγηση όπου είναι και αυτή που μας ενδιαφέρει περισσότερο στο κεφάλαιο αυτό. Ο αλγόριθμος υλοποιήθηκε στην πλατφόρμα XMT και πάλι, όπου είναι και ο προσομοιωτής που μας βοήθησε να κάνουμε τα πειράματα μας και να πάρουμε τα αποτελέσματα. Ο λόγος που έγιναν τα πιο κάτω πειράματα είναι για να δούμε αν πράγματι από την υλοποίηση του αλγορίθμου παίρνουμε τα αναμενόμενα αποτελέσματα και αν συμπεριφέρονται με τον ίδιο τρόπο η θεωρητική και πειραματική ανάλυση. Η θεωρητική ανάλυση έγινε δίνοντας στις συναρτήσεις τόσο του χρόνο όσο και του κόστους διάφορες τιμές για τον αριθμό των δεδομένων, αφού είναι η μόνη μεταβλητή σε αυτή την περίπτωση, ενώ τα πειράματα έγιναν αυξάνοντας κάθε φορά το μέγεθος του προβλήματος. Δόθηκαν διάφορες τιμές και τρέχαμε τον αλγόριθμο που υλοποιήθηκε. Για ένα συγκεκριμένο αριθμό δεδομένων(πχ n=16) γίνονταν τρείς μετρήσεις με διαφορετικά δεδομένα, έπειτα υπολογιζόταν ο μέσος όρος από αυτές και το αποτέλεσμα είναι η τιμή που χρησιμοποιείται στην ανάλυση. Τις μετρήσεις ακολουθούσε η δημιουργία πίνακα που περιείχε τα δεδομένα και στη συνέχεια η δημιουργία των γραφικών παραστάσεων που περιλάμβανε το θεωρητικό αλλά και πειραματικό γράφημα. Ξεκινώντας πήραμε πέντε διαφορετικές μετρήσεις για τον αλγόριθμο μας, δηλαδή δώσαμε πέντε διαφορετικά πλήθη δεδομένων από 9 μέχρι 30 και πιο συγκεκριμένα δώσαμε τις τιμές 9,15,20,25,30. Δυστυχώς δεν μπορούσαμε να πάρουμε για μεγαλύτερο αριθμό δεδομένων για τον λόγο ότι ο αριθμός των επεξεργαστών που δουλεύουν στον συγκεκριμένο αλγόριθμο είναι συνάρτησης της δύναμης του n. Επίσης δεν πρέπει να ξεχνούμε τον περιορισμό όσον αφορά τον αριθμό των επεξεργαστών, όπου δεν μπορούν να χρησιμοποιούνται περισσότεροι από 512 ή 1024 σε κάποιες περιπτώσεις. Εμείς δοκιμάσαμε για n=30 όπου είναι ένας πολύ μικρός αριθμός στοιχείων και άρα δούλευαν 900 επεξεργαστές ταυτόχρονα, αριθμός πολύ

κοντά στο 1024. Αφού πήραμε τις πειραματικές μετρήσεις μας, τότε υπολογίσαμε για τις αντίστοιχες τιμές τα θεωρητικά αποτελέσματα σύμφωνα με την συνάρτηση log^2n όπου =logn*logn που προέκυψε από την ανάλυση. Πιο κάτω ακολουθεί ο πίνακας Πινάκας5.1 με τα δεδομένα μας όπου περιέχει τόσο τα πειραματικά αλλά και θεωρητικά αποτελέσματα.

Αριθμός στοιχείων(n)	Πειραματικές Μετρήσεις	Θεωρητικές Μετρήσεις
9	41263,33	10,048
15	75520,33	15,26
20	95685	18,67
25	140085,7	21,565
30	175589	24,0775

Πίνακας 5.1

Λόγω της μεγάλης διαφοράς στον αριθμό της πειραματικής και θεωρητική μέτρησης θεώρησα καλύτερο τα δύο γραφήματα να γίνουν σε διαφορετική γραφική παράσταση, έτσι ώστε να είναι πιο ευδιάκριτα. Ακριβώς πιο κάτω ακολουθούν οι δύο πίνακες που προέκυψαν μέσα από τα δεδομένα μας που μας δείχνουν πως μεταβάλλεται ο χρόνος με την αλλαγή τους. Η γραφική 5.1 παρουσιάζει τα δύο γραφήματα, θεωρητικής και πειραματικής αξιολόγησης.



Γραφική 5.1

Παρατηρήσεις

Βλέποντας τις γραφικές είναι εύκολο να δούμε ότι και στις δύο υπάρχει μια αύξηση στο χρόνο που χρειάζεται ο προσομοιωτής να λύσει το πρόβλημα μας καθώς αυξάνονται τα δεδομένα που χρησιμοποιούμε. Δηλαδή όσο δίνουμε περισσότερα δεδομένα τόσο μεγαλώνει ο χρόνος και αυτό είναι λογικό να συμβαίνει γιατί ο χρόνος σύμφωνα με την συνάρτηση που προέκυψε από την θεωρητική ανάλυση μας είναι ευθέως ανάλογος με τον αριθμό των δεδομένων και κατ’ επέκταση με τον αριθμό των επεξεργαστών.

Μια δεύτερη παρατήρηση που μπορούμε να κάνουμε είναι ότι το πειραματικό γράφημα ακολουθεί την μορφή και τον τρόπο συμπεριφοράς του θεωρητικού γραφήματος. Αυτό είναι πολύ σημαντικό γιατί είναι ένδειξη ότι πράγματι στην υλοποίηση τα βήματα που χρειάζονται για να ολοκληρωθεί ο αλγόριθμος μας, είναι αυτά που πρέπει και αναμενόμενα σύμφωνα πάντα με την θεωρητική ανάλυση. Το ότι ακολουθούν σχεδόν την ίδια μορφή ή παρόμια θα μπορούσε να πει κανείς είναι πολύ ενθαρρυντικό. Για τον λόγο ότι δεν ελέγξαμε πολλά στοιχεία λόγο του περιορισμού μας υπάρχει μια μικρή αμφιβολία για την καμπύλη του γραφήματος που προέκυψε αφού θα έπρεπε να είναι λίγο πιο έντονη η καμπύλη γιατί είναι συνάρτηση της δύναμης του αλγορίθμου. Δεν μπορούμε όμως να δώσουμε περισσότερη έμφαση σε αυτό γιατί δεν μας το επιτρέπει ο περιορισμός και οι συνθήκες που ασχολούμαστε. Έτσι απλά μένουμε στο ότι οι δυο γραφικές συμπεριφέρονται με τον ίδιο τρόπο.

Η τελευταία παρατήρηση που έχουμε να κάνουμε είναι ότι τα αποτελέσματα στην πειραματική αξιολόγηση είναι πολύ μεγαλύτερα, από τα αποτελέσματα που πήραμε στην θεωρητική αξιολόγηση, δηλαδή, οι αριθμοί. Αυτό οφείλεται στο γεγονός ότι στον πρόγραμμα μας παίρνουν επιπλέον χρόνο οι διάφορες αρχικοποιήσεις που γίνονται, οι έλεγχοι, το πρώτο και τρίτο βήμα που στην θεωρητική ανάλυση θεωρούνται αμελητέα και μπαίνουν κάτω από την ομπρέλα των επαναλήψεων ενώ στην πραγματικότητα χρειάζονται και αυτά το χρόνο τους να γίνουν. Αυτός είναι ο λόγος που υπάρχει μια τέτοια διαφορά στα αποτελέσματα που έχουμε.

Αφού πάρθηκαν οι μετρήσεις του χρόνου, σειρά έχει να υπολογισθεί το κόστος του αλγορίθμου, τόσο το θεωρητικό όσο και το πειραματικό. Το θεωρητικό κόστος υπολογίζεται μέσα από την συνάρτηση που προέκυψε από την ανάλυση που είναι O(n^2log^2(n)) και χρησιμοποιήθηκαν τα πέντε πλήθη στοιχείων που ελέγχθηκαν πιο πάνω για τον χρόνο. Το πειραματικό τώρα κόστος προέκυψε πολλαπλασιάζοντας το χρόνο που χρειάζεται να ολοκληρωθεί ο αλγόριθμος επί των αριθμό των επεξεργαστών που χρησιμοποιήθηκαν, που δεν ήταν άλλος από την δύναμη του πλήθους των δεδομένων. Ο χρόνος χρησιμοποιείται όπως πάρθηκε στα πιο πάνω πειράματα. Δηλαδή, ακολουθήσαμε τον τρόπο που ορίστηκε στην αρχή και είναι αριθμός επεξεργαστών*χρόνος εκτέλεσης. Τα δεδομένα παρέμειναν τα ίδια στη συγκεκριμένη περίπτωση.

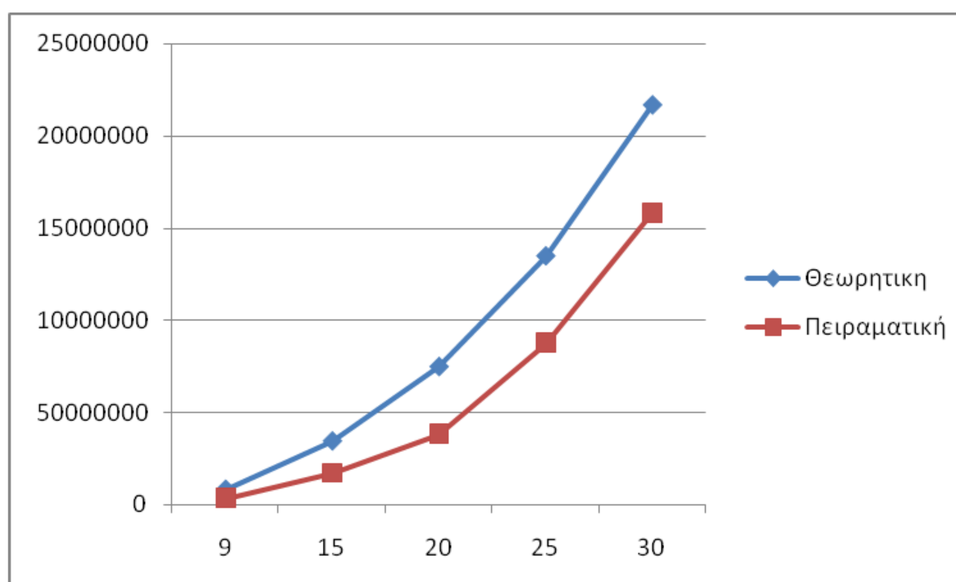
Ο πίνακας που ακολουθεί Πίνακας 5.2 παρουσιάζει τα αποτελέσματα που προέκυψαν.

Αριθμός Δεδομένων(n)	Πειραματική Αξιολόγηση	Θεωρητική Αξιολόγηση
9	3342329	813.92
15	16992074	3434.35
20	38274000	7471.62
25	87553563	13478.37

30	158030100	21669.81

Πίνακας 5.2

Μετά από τον πίνακα προκύπτουν οι γραφικές μας που γίνονται βάση τον πίνακα που έχουμε. Και πάλι τα γραφήματα μας θα αναπαρασταθούν σε δυο γραφικές για να μπορούμε να κάνουμε την σύγκριση μας. Προκύπτει η γραφική παράσταση που αναπαριστάται στην Γραφική 5.2.



Γραφική 5.2

Παρατηρήσεις

Σύμφωνα και με τις δύο γραφικές που έχουμε στην διάθεση μας παρατηρούμε ότι το κόστος αυξάνεται όσο αυξάνεται ο αριθμός των στοιχείων που κάνουμε τα πειράματα μας. Και πάλι αυτό είναι μια συμπεριφορά που αναμέναμε γιατί βάση της συνάρτησης του κόστους από την θεωρητική ανάλυση έχουμε ότι είναι ευθέως ανάλογα των επεξεργαστών που χρησιμοποιούνται και πιο συγκεκριμένα της δύναμης τους. Άρα είναι λογικό όσο αυξάνεται ο αριθμός των δεδομένων τόσο να αυξάνεται και το κόστος που χρειάζεται για να ολοκληρωθεί.

Πολύ εύστοχα παρατηρούμαι επίσης ότι τα δύο γραφήματα ακολουθούν την ίδια τάση, και έχουν την ίδια συμπεριφορά. Η μορφή τους είναι πολύ παρόμοια. Και τα δύο βλέπουμε ότι ακολουθούν μια τάση η οποία είναι της μορφής n^2 όπου αφού σε κάθε πείραμα χρησιμοποιούνται n^2 επεξεργαστές. Η μορφή είναι σχεδόν γραμμική αλλά όχι απόλυτα γιατί

την μορφή καθορίζει επίσης ο πολλαπλασιασμός που γίνεται με τον χρόνο ο οποίος είναι λογαριθμικός και έτσι παρατηρούμαι την καμπύλη που έχουν τα δύο γραφήματα. Επίσης παρατηρούμε ότι η πειραματική γραφική ακολουθεί την τάση της θεωρητικής και αυτό μας καθορίζει ότι σωστά η υλοποίηση μας γίνεται με βάση τον θεωρητικό αλγόριθμο.

Φυσικό επακόλουθο είναι και πάλι οι τιμές στην πειραματική αξιολόγηση να είναι πιο μεγάλες από τις αντίστοιχες της θεωρητικής και αυτό προκύπτει από την τελευταία παρατήρηση στις γραφικές με τον χρόνο, γιατί αφού είχαμε και εκεί μεγάλη διαφορά είναι λογικό να έχουμε και εδώ αφού πολλαπλασιάζουμε αυτή την τιμή.

5.4 Παραλλαγή Αλγορίθμου Συνεκτικών Συνιστωσών

Ο περιορισμός που έχουμε να αντιμετωπίσουμε, ότι δηλαδή δεν μπορούμε να χρησιμοποιούμε πάνω από 512 ή 1024 επεξεργαστές, αναλόγως, μας οδήγησε στην παραλλαγή του αλγορίθμου αυτού. Με την χρήση αυτών των επεξεργαστών μπορούσαμε να κάνουμε πειράματα όπου το πλήθος των δεδομένων ήταν μέχρι 30 στοιχεία. Αυτό μας δυσκόλεψε και μας στέρησε την δυνατότητα να δοκιμάσουμε περισσότερα στοιχεία και να ελέγξουμε πως συμπεριφέρεται ο αλγόριθμος μας. Έτσι γεννήθηκε η ιδέα να υλοποιηθεί ένας παράλληλος αλγόριθμος διαφορετικός όπου θα κάνει χρήση και απαιτεί λιγότερους επεξεργαστές. Ο αλγόριθμος αυτός αποτελεί την παραλλαγή και πιο κάτω εξηγείται πως έγινε.

Η παραλλαγή του κανονικού αλγορίθμου συνεκτικών συνιστωσών έχει την βάση της και μοιάζει πάρα πολύ με τον κανονικό αλγόριθμο εκτός κάποια σημεία που διαφοροποιείται. Ακολουθεί όμως την ίδια δομή, τακτική και κινείται κατά κύριο λόγο στα βήματα του προηγούμενου. Έτσι εμείς θα επικεντρωθούμε στα σημεία που έγιναν οι αλλαγές και όχι στην επεξήγηση τους από το μηδέν. Στόχος λοιπόν ήταν να δούμε πως αλλιώς μπορούν να υλοποιηθούν τα σημεία που απαιτούν n^2 επεξεργαστές. Έτσι παίρνουμε τα σημεία με την σειρά και αναλόγως κάνουμε αλλαγές ή παραμένουν ίδια τα σημεία. Αρχικά το βήμα 1 του αλγορίθμου δεν χρειάζεται κάποια αλλαγή αφού είναι απλές αρχικοποιήσεις. Το βήμα 2 είναι αυτό που μας ενδιαφέρει κυρίως. Παρατηρούμε ότι για το πρώτο βήμα του, δηλαδή 2.1 δεν χρειάζεται κάποια αλλαγή αφού είναι μια απλή πράξη και δεν μας επηρεάζει κάτι. Αντιθέτως το βήμα 2.2 παρατηρούμε ότι κάνει χρήση n^2 επεξεργαστών και έτσι εμείς στην παραλλαγή του αλγορίθμου δοκιμάζουμε άλλη τεχνική όπου κάνει χρήση n επεξεργαστών και χρειάζεται O(n) χρόνο για να ολοκληρωθεί. Παρατηρούμε ότι παίρνει περισσότερο χρόνο αλλά κάνει χρήση πιο λίγων επεξεργαστών που είναι και το ζητούμενο μας σε αυτή την περίπτωση. Το βήμα που ακολουθεί δηλαδή 2.3 παρατηρούμε ότι οι επεξεργαστές μας είναι μέσα στα επιθυμητά όρια μας και έτσι δεν κάνουμε κάποια αλλαγή αφού δεν μας επηρεάζει. Το βήμα

όμως 2.4 που ασχολείται με την δημιουργία του νέου πίνακα γειτνίασης βάση του προηγουμένου που έχουμε στην διάθεση μας χρειάζεται τροποποίηση για να χρησιμοποιούνται ο επιθυμητός αριθμός επεξεργαστών. Η τροποποίηση που έγινε είναι να δουλεύουν n επεξεργαστές αλλά να χρειάζονται O(n) χρόνο για να ολοκληρώσουν την δουλεία τους και αυτό γιατί ο κάθε επεξεργαστής είναι υπεύθυνος για μια σειρά του πίνακα και όχι για ένα στοιχείο, μόνο όπως στον κανονικό αλγόριθμων συνεκτικών συνιστωσών. Το βήμα 3 δεν εντοπίζει κάποια αλλαγή αφού είναι βασισμένο και στις δύο περιπτώσεις στο βήμα 2 των αλγορίθμων μας.

Παρατηρούμε ότι κάθε επανάληψη χρειάζεται O(n) χρόνο για να ολοκληρωθεί. Ο αριθμός των επαναλήψεων που απαιτούνται δεν αντιμετωπίζει κάποια διαφορά και πάλι είναι logn. Άρα συνολικός χρόνος είναι το γινόμενο τους και άρα O(nlogn). Επίσης ο στόχος μας επιτεύχθηκε να χρησιμοποιούμε δηλαδή n επεξεργαστές παρά n^2.

Μέσα από αυτές τις αλλαγές που έγιναν συνοψίζοντας έχουμε συνολικό χρόνο εκτέλεσης O(nlogn) με τη χρήση O(n) επεξεργαστών. Το κόστος όπως και στις προηγούμενες περιπτώσεις έρχεται μέσα από τον πολλαπλασιασμό του χρόνου και του αριθμού των επεξεργαστών. Άρα το κόστος για την παραλλαγή αυτή είναι $O(n^2logn)$.

Πειραματική Αξιολόγηση

Αφού έγινε η υλοποίηση και αυτού του αλγορίθμου ακολούθησε η πειραματική αξιολόγηση του για να μπορούμε να τον συγκρίνουμε με τον κανονικό αλγόριθμο που υλοποιήθηκε. Η πειραματική αξιολόγηση ακολούθησε τα βήματα που έγιναν και στις προηγούμενες πειραματικές αξιολογήσεις. Αυτή την φορά είχαμε την ευκαιρία να δοκιμάσουμε και να πειραματιστούμε σε μεγαλύτερο πλήθος δεδομένων αφού έχουμε στην διάθεση μας μέχρι 1024 επεξεργαστές άρα μπορούμε να ελέγξουμε μέχρι 1024 στοιχεία κάτι το οποίο κάναμε. Κάναμε πειράματα από 8 μέχρι 1024 δεδομένα και συγκεκριμένα ελέγξαμε τις τιμές 8,16,32,64,128,256,512 και 1024. Για αυτές τις τιμές τρέξαμε τον αλγόριθμο μας τρείς(3) φορές με διαφορετικά δεδομένα και τα αποτελέσματα που προκύπτουν είναι ο μέσος όρος τους. Για την θεωρητική αξιολόγηση εφαρμόσαμε τον ίδιο αριθμό δεδομένων στις συναρτήσεις τόσο του χρόνου όσο και του κόστους.

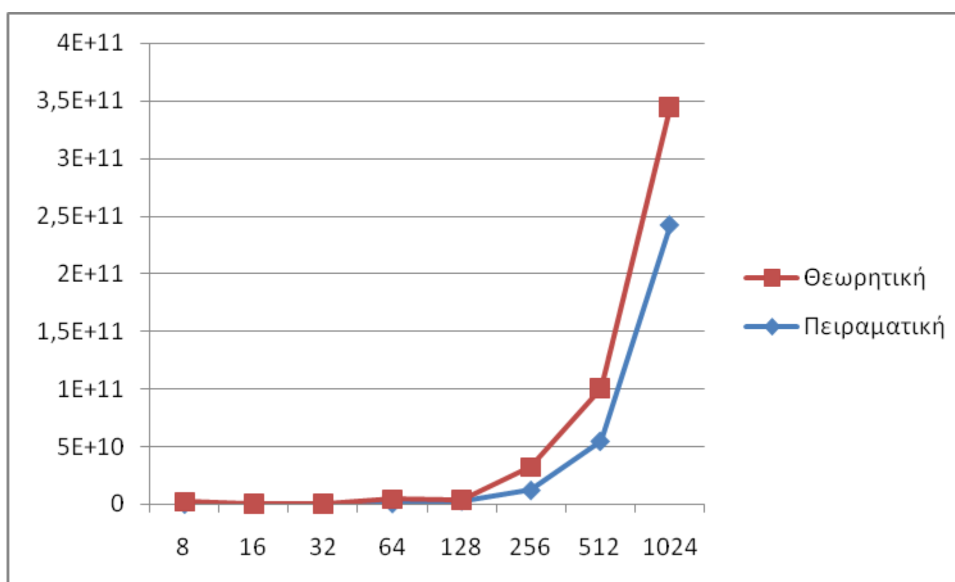
Αρχίζουμε με τον χρόνο και πιο κάτω είναι ο πίνακας Πίνακας5.3 με τα αποτελέσματα για τα δεδομένα μας, πειραματική και θεωρητική αντίστοιχα.

Αριθμός Δεδομένων(n)	Πειραματική Αξιολόγηση	Θεωρητική Αξιολόγηση
8	554858,5	24
16	1479623	64
32	3699057	160
64	9001130	384
128	20714717	896

256	46575089	2048
512	1,07E+08	4608
1024	2,37E+08	10240

Πίνακας 5.3

Τον πίνακα ακολούθησε η δημιουργία των γραφικών παραστάσεων για το πώς εξελίσσεται ο χρόνος όπου και πάλι είναι δύο διαφορετικές παραστάσεις, μια για την πειραματική αξιολόγηση και μια για την θεωρητική έτσι ώστε να μπορούμε να συγκρίνουμε με μεγαλύτερη ευκολία για το λόγο ότι έχουν μεγάλη διαφορά στις τιμές. Στην Γραφική Αναπαράσταση Γραφική 5.3 παρουσιάζεται η γραφική που προκύπτει από τον πίνακα.



Γραφική 5.3

Παρατηρήσεις

Η πρώτη παρατήρηση που γίνεται έχει να κάνει πάλι με την αύξηση του χρόνου. Όσο περισσότερα είναι τα δεδομένα τόσο περισσότερος χρόνος χρειάζεται να εκτελεστεί ο αλγόριθμος μας. Αυτό συμβαίνει γιατί ο χρόνος είναι ανάλογος του πλήθους των δεδομένων και έτσι όταν αυξάνεται ο ένας παράγοντας αυξάνεται και ο άλλος. Η αύξηση αυτή παρατηρείται και στις δύο γραφικές παραστάσεις.

Μια δεύτερη παρατήρηση που γίνεται και πολύ σημαντική είναι ότι τα δύο γραφήματα έχουν την ίδια μορφή, ακολουθούν την ίδια τάση. Αυτό μας επιβεβαιώνει ότι ο αλγόριθμος που υλοποιήσαμε ακολουθεί σωστά τα βήματα του αλγόριθμου που είναι δοσμένου σε θεωρητικό επίπεδο. Αν θέλουμε να μιλήσουμε πιο συγκεκριμένα και τα δύο γραφήματα ακολουθούν λογαριθμική αύξηση με το πέρασμα του χρόνου αφού έχουν την μορφή λογαριθμικής αναπαράστασης. Αυτό είναι αναμενόμενο γιατί συμφώνα με την θεωρητική ανάλυση χρειαζόμαστε O(nlogn) χρόνο.

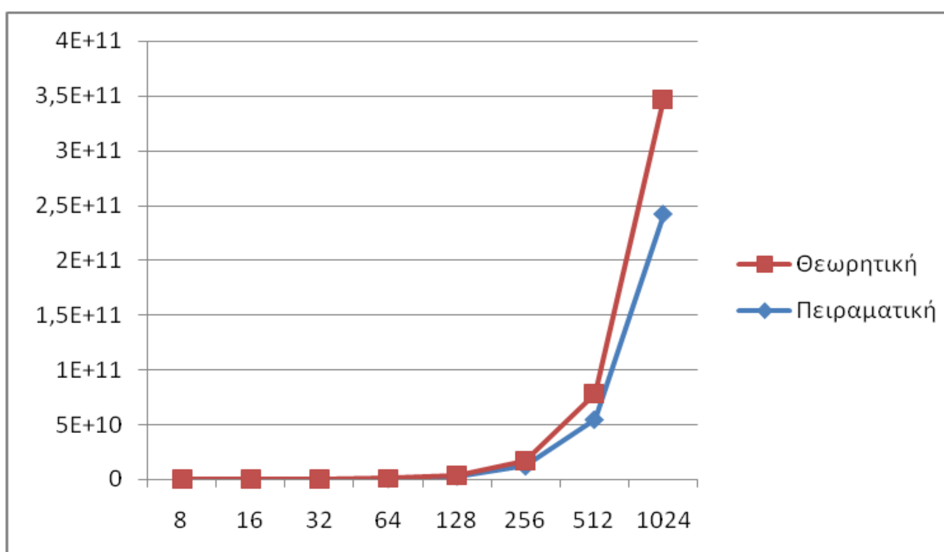
Η τελευταία παρατήρηση έχει να κάνει και πάλι με την σύγκριση των τιμών της θεωρητικής και της πειραματικής αξιολόγησης. Τα αποτελέσματα στην πειραματική είναι πολύ πιο μεγάλα και αυτό συμβαίνει γιατί κατά την υλοποίηση γίνονται κάποιες επιπλέον πράξεις όπως εξηγήθηκε και πιο πάνω που παίρνουν χρόνο, ενώ κατά την θεωρητική ανάλυση δεν τις λαμβάνουμε υπόψη και δίνουμε τον ασυμπωτικό χρόνο του αλγορίθμου.

Στη συνέχεια θα ασχοληθούμε με το κόστος του νέου αλγορίθμου που εξηγήσαμε. Όπως προέκυψε από την ανάλυση της παραλλαγής το κόστος είναι της τάξεως O(n^2(logn)). Αφού χρησιμοποιούνται n επεξεργαστές και ο συνολικός χρόνος εκτέλεσης του προγράμματος είναι nlogn. Οι πειραματικές μετρήσεις προέκυψαν από το γινόμενο του πλήθους των επεξεργαστών όπου είναι n και του χρόνου όπου χρησιμοποιούνται οι μετρήσεις που πάρθηκαν στα πιο πάνω πειράματα. Πιο κάτω παρουσιάζεται ο πίνακας Πίνακας5.4 που προκύπτει για το κόστος του αλγορίθμου όπου φαίνονται και τα πειραματικά αποτελέσματα αλλά και να θεωρητικά.

Αριθμός Δεδομένων(n)	Πειραματική Αξιολόγηση	Θεωρητική Αξιολόγηση
8	4438868	192
16	23673962	1024
32	1,18E+08	5120
64	5,12E+08	24576
128	2,65E+09	114688
256	1,19E+10	524288
512	5,45E+10	2359296
1024	2,42E+11	10485760

Πίνακας5.4

Σειρά έχουν και πάλι η δημιουργία των γραφικών έτσι ώστε να μπορούμε να συγκρίνουμε τα αποτελέσματα που έχουμε από τα κόστη. Το γράφημα που προκύπτει από τα στοιχεία του πίνακα παρουσιάζεται στην Γραφική 5.4.



Γραφική5.4

Παρατηρήσεις

Όπως σε όλες τις περιπτώσεις έτσι και σε αυτή όσο αυξάνεται ο αριθμός των δεδομένων τόσο περισσότερο κόστος έχουμε για την επίλυση του αλγορίθμου μας. Αυτό το παρατηρούμε με μεγάλη ευκολία από τις γραφικές μας, αφού έχουμε μια δραματική αύξηση του κόστους που χρειαζόμαστε ο αλγόριθμος με το πέρας των δεδομένων και στις δύο γραφικές μας. Αυτό είναι και πάλι αναμενόμενο γιατί το κόστος είναι ανάλογο του n, των στοιχείων που έχουμε να επεξεργαστούμε και να δώσουμε το αναμενόμενο αποτέλεσμα. Άρα όσο περισσότερα στοιχεία έχουμε, τόσο περισσότερο κόστος χρειάζεται να λυθεί το πρόβλημα των συνεκτικών συνιστωσών.

Αν συγκρίνουμε τα δύο γραφήματα μπορούμε να παρατηρήσουμε με μεγάλη ευκολία ότι έχουν την ίδια μορφή, κυμαίνονται στα ίδια επίπεδα και αυτό μας δίνει την ευχέρεια να είμαστε πιο ακριβείς ο λόγος ότι παίρνουμε περισσότερα δείγματα απ’ ότι είχαμε στον προηγούμενο κανονικό αλγόριθμο. Βλέπουμε ότι είναι λογαριθμική η τάση που έχει η γραφική μας και αυτό οφείλεται στο ότι είναι πολλαπλάσιο του λογαρίθμου και επίσης αυτή η απότομη αλλαγή και άνοδος που έχουν μεταξύ τους οι μετρικές οφείλεται στο ότι είναι ανάλογο της δύναμης του n όπου κάνει πολύ πιο έντονη και αισθητή την αλλαγή και άνοδο.

Τέλος παρατηρούμαι ότι οι αριθμοί στην πειραματική αξιολόγηση είναι κατά πολύ πιο μεγάλοι από τους αριθμούς που έχουμε στην θεωρητική ανάλυση και αυτή η παρατήρηση βασίζεται σε μια προηγούμενη παρατήρηση που έγινε για τον χρόνο όπου ισχύει το αντίστοιχο. Όπως ειπώθηκε, το κόστος πηγάζει από τον χρόνο και έτσι αφού υπάρχει αυτή η μεγάλη διαφορά στον χρόνο, έτσι προκύπτει και η μεγάλη διαφορά και στο κόστος όσον αφορά την σύγκριση με τις θεωρητικές τιμές μας.

5.5 Σύγκριση Αλγορίθμων

Σε αυτό το σημείο θα συγκρίνουμε τον χρόνο και το κόστος που λύφθηκαν από τα πειράματα μας κατά την προσομοίωση τους στην πλατφόρμα. Αξίζει να σημειωθεί ότι για τον αλγόριθμο της παραλλαγής έγιναν άλλες μετρήσεις για να είναι στα μέτρα του κανονικού αλγορίθμου και να ταιριάζει με τους περιορισμούς που έχουμε. Δηλαδή τρέξαμε τον αλγόριθμο της παραλλαγής για τις τιμές n=9,15,20,25 και 30, έτσι ώστε να μπορούμε να συγκρίνουμε για ίδιο αριθμό δεδομένων γιατί αλλιώς δεν θα ήταν εφικτό. Σε αυτό το σημείο δεν θα ασχοληθούμε με την θεωρητική ανάλυση αφού αναλύθηκε πιο πάνω,

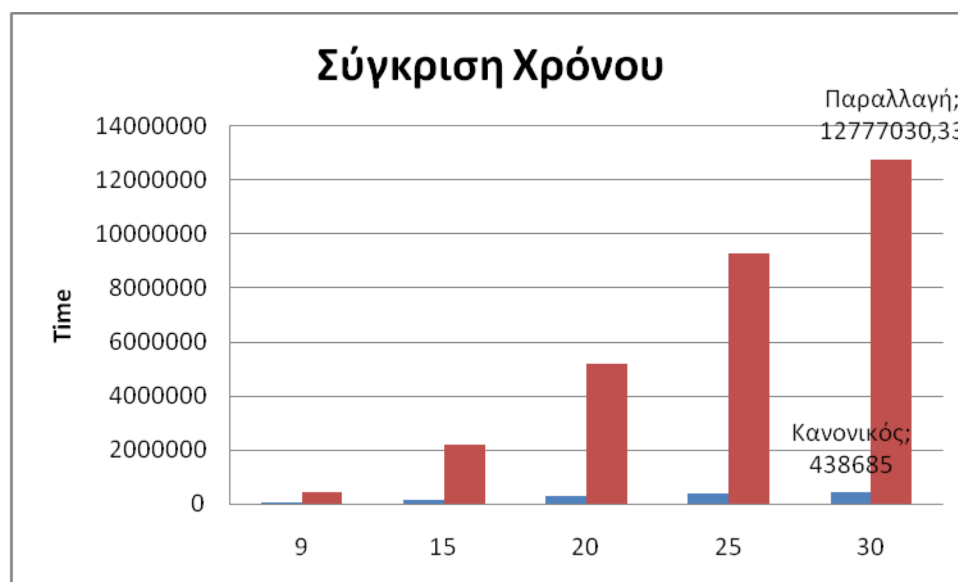
Πρώτα θα ασχοληθούμε με τον χρόνο και να συγκρίνουμε τους δύο αλγορίθμους πως συμπεριφέρεται ο χρόνος τους για τα ίδια δεδομένα. Ακολουθεί ο πίνακας Πίνακας5.5 με τα αποτελέσματα που έχουμε από την προσομοίωση τους.

Αριθμός Δεδομένων(n)	Κανονικός Αλγόριθμος	Παραλλαγή Αλγορίθμου
9	41263,33	405599
15	135205,3	2195079
20	256851	5175655
25	353450,7	9288921
30	438685	12777030

Πίνακας 5.5

Στον πίνακα μπορούμε να δούμε πριν μελετήσουμε την γραφική ότι η παραλλαγή του αλγορίθμου χρειάζεται περισσότερο χρόνο για να ολοκληρώσει τον αλγόριθμο της σε σύγκριση με τον κανονικό αλγόριθμο.

Ακολουθεί η γραφική Γραφική5.5 που περιέχει τα δύο γραφήματα έτσι ώστε να μπορούμε να κάνουμε μια πιο ξεκάθαρη σύγκριση των δύο αλγορίθμων.



Γραφική 5.5

Παρατηρήσεις

Αρχικά παρατηρούμε ότι και τα δύο γραφήματα αυξάνουν τον χρόνο τους καθώς αυξάνουμε τα δεδομένα μας, δηλαδή συμπεριφέρονται με τον ίδιο τρόπο, δεν υπάρχει κάποιο που να μειώνει τον χρόνο καθώς αυξάνεται το πλήθος των δεδομένων. Και οι δύο αλγόριθμοι είναι ανάλογοι του μεγέθους των δεδομένων και έτσι συμπεριφέρονται το ίδιο στην προκειμένη περίπτωση.

Σύμφωνα με τις συναρτήσεις που προέκυψαν από την θεωρητική ανάλυση ο κανονικός αλγόριθμος χρειάζεται $O(\log^2 n)$ χρόνο για να εκτελέσει τις $O(n\log n)$, άρα αναμένουμε ότι η παραλλαγή του αλγορίθμου θα χρειάζεται περισσότερο χρόνο να εκτελεστεί. Σύμφωνα με

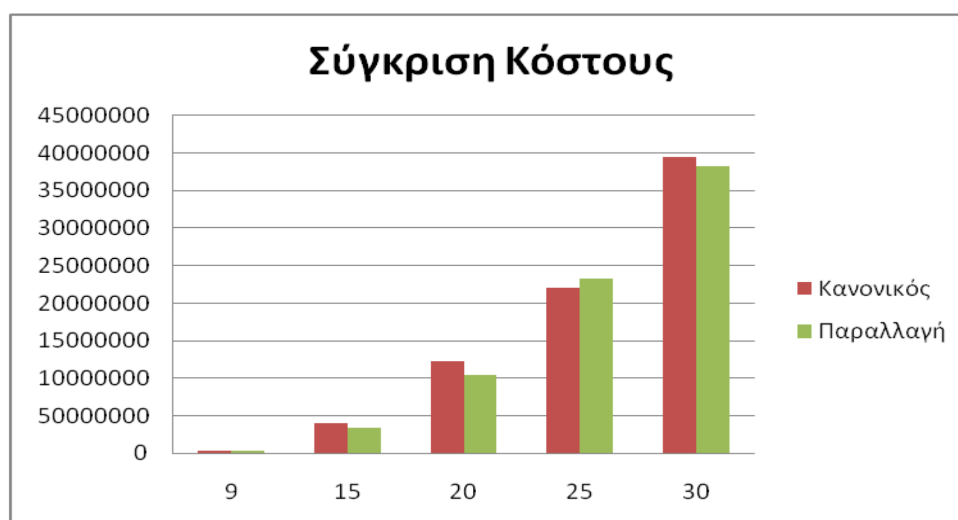
αυτά που προέκυψαν αυτό διαπιστώνεται αφού ο αλγόριθμος όπου είναι η παραλλαγή, βλέπουμε από την γραφική μας ότι για το ίδιο μέγεθος δεδομένων χρειάζεται περισσότερο χρόνο να εκτελεστεί και να δώσει το αναμενόμενο αποτέλεσμα, ενώ ο κανονικός χρόνος χρειάζεται πολύ λιγότερο χρόνο. Άρα μπορούμε να συμπεράνουμε ότι για το ίδιο μέγεθος δεδομένων ο κανονικός αλγόριθμος είναι πιο γρήγορος σε σχέση με την παραλλαγή του και αυτό οφείλεται στο ότι πολλές πράξεις γίνονται σε γραμμικό χρόνο στην παραλλαγή.
Ως συμπέρασμα από αυτή την σύγκριση μπορούμε να πούμε ότι ο κανονικός αλγόριθμος είναι πιο γρήγορος από την παραλλαγή του αλγορίθμου και ότι επίσης και οι δύο αλγόριθμοι συμπεριφέρονται με τον ίδιο τρόπο όσον αφορά τον αριθμό των δεδομένων, αυξάνεται ο χρόνος όσο αυξάνεται ο αριθμός των δεδομένων.

Στη συνέχεια θα ασχοληθούμε το κόστος των δύο αλγορίθμων. Κόστος το οποίο προκύπτει όταν πολλαπλασιάσουμε τον αριθμό των επεξεργαστών που δούλεψαν παράλληλα επί τον χρόνο που χρειάστηκε να λυθεί το πρόβλημα. Ό χρόνος όπως και όλες τις προηγούμενες φορές βασίζεται στις πιο πάνω μετρήσεις που έγιναν και αυτές χρησιμοποιούύνται.
Πιο κάτω είναι ο πίνακας Πίνακας 5.6 που περιέχει τα αποτελέσματα που προέκυψαν σύμφωνα με τις μετρήσεις μας.

Αριθμός Δεδομένων(n)	Κανονικός Αλγόριθμος	Παραλλαγή Αλγορίθμου
9	3342329,7	3650391
15	40421193	32926185
20	122740400	1,04E+08
25	220906250	2,32E+08
30	394816500	3,83E+08

Πίνακας 5.6

Τον πίνακα ακολουθεί η γραφική Γραφική5.6 όπου θα είναι μπορούμε να συγκρίνουμε με μεγαλύτερη ευκολία για το πώς συμπεριφέρονται οι αλγόριθμοι μας όσον αφορά το κόστος.



Γραφική5.6

Παρατηρήσεις

Αρχικά παρατηρούμε για άλλη μια φορά ότι όσο αυξάνεται ο αριθμός των δεδομένων τόσο αυξάνεται το κόστος και για τους δύο αλγόριθμους, γιατί είναι ανάλογη του n και οι δύο.

Από την θεωρητική ανάλυση και των δύο αλγορίθμων προέκυψε ότι ο κανονικός αλγόριθμος έχει κόστος O(n^2(log^2(n))) και η παραλλαγή O(n^2(logn)) κόστος. Θα μπορούσε να πει κανείς ότι έχουν σχεδόν το ίδιο κόστος αφού η διαφορά τους είναι ένα logn, όπου ο κανονικός αλγόριθμος απαιτεί λίγο μεγαλύτερο κόστος από τον αλγόριθμο της παραλλαγής. Στις μετρήσεις μας κατά κύριο λόγο παρατηρούμε ότι επιβεβαιώνεται αυτή η διαπίστωση αφού σχεδόν σε όλες τις μετρήσεις το κόστος του κανονικού αλγορίθμου εύρεσης συνεκτικών συνιστωσών είναι μεγαλύτερο από το κόστος της παραλλαγής. Είναι σχεδόν ίσα θα μπορούσαμε και αν αντιμετωπίζονται κάποιες αστάθειες αυτές προκύπτουν γιατί σε όλους τους υπολογισμούς του κόστους και χρόνου κατά την θεωρητική ανάλυση υπολογιζόταν ο ασυμπωτικός χρόνος και κόστος των αλγορίθμων αντίστοιχα.

Οι συγκρίσεις που έγιναν πιο πάνω τόσο του κόστους όσο και του χρόνου για τους δύο αλγορίθμους που μελετήσαμε δεν μπορούν να μας αφήσουν εκατό τις εκατό βέβαιους και σίγουρους για τα συμπεράσματα μας. Ο λόγος αυτού του ισχυρισμού πηγάζει από τον περιορισμό που αντιμετωπίσαμε μπροστά μας κατά την διάρκεια της πτυχιακής εργασίας αυτής, και φυσικά αναφέρομαι στον περιορισμό όπου δεν μπορούν να χρησιμοποιηθούν πέραν των 1024 επεξεργαστών στη συγκεκριμένη περίπτωση. Τα αποτελέσματα που πήραμε μέσα από τα πειράματα θεωρείται μια ένδειξη αποτελεσμάτων και όχι αποτελέσματα τα οποία θα μπορούσαν να εκπροσωπήσουν ένα γενικό πείραμα. Προσπαθήσαμε και προσαρμοστήκαμε όμως μέσα από τα υπάρχοντα δεδομένα και κάναμε τις συγκρίσεις μας.

Κεφάλαιο 6

Συμπεράσματα

6.1 Τελικά Συμπεράσματα 74
6.2 Προβλήματα Που Συναντήθηκαν 75
6.3 Οφέλη Ατομικής Διπλωματικής Εργασίας 76
6.4 Μελλοντική Εργασία 77

6.1 Τελικά Συμπεράσματα

Με την ολοκλήρωση της Διπλωματικής Εργασίας αυτής καλό θα ήταν να συνοψίσουμε τα συμπεράσματα που προέκυψαν κατά την διάρκεια διεξαγωγής της και τι αποκομίσαμε μέσα από αυτήν. Ένα πολύ σημαντικό κομμάτι για εμένα ήταν η εμβάθυνση στην παράλληλη σκέψη (parallel thinking). Μετά από την πρώτη επαφή με τον παραλληλισμό μέσα από την παρακολούθηση του μαθήματος ¨ΕΠΛ431:Παράλληλοι Αλγόριθμοι¨, οι γνώσεις και εμπειρίες που αποκόμισα από αυτή την εργασία θα μου είναι ιδιαίτερα χρήσιμες γιατί ο παραλληλισμός είναι το μέλλον της Επιστήμης αυτή. Με μεγάλη μου χαρά κατάλαβα πως είναι να σκέφτεσαι παράλληλα αλλά και να τα κάνεις στην πράξη. Δηλαδή η υλοποίηση των παράλληλων αλγορίθμων. Ένα πολύ σημαντικό κομμάτι της εργασίας αυτής είναι και η σύγκριση που έπρεπε να γίνει, κάτι που ανέπτυξε τις ικανότητες μου να μπορώ να συγκρίνω και να εξάγω συμπεράσματα. Κάποια από τα συμπεράσματα που προέκυψαν είναι τα ακόλουθα.

Σύμφωνα με όλους τους αλγορίθμους που υλοποιήθηκαν ο χρόνος όσο και το κόστος είναι ανάλογα του πλήθους των δεδομένων. Δηλαδή όσα περισσότερα στοιχεία έχουμε να επεξεργαστούμε τόσο περισσότερος χρόνος χρειάζεται να ολοκληρωθεί ο αλγόριθμος μας. Όπως επίσης ισχύει το αντιθέτω, όσο πιο λίγα στοιχεία είναι τόσο ποιος λίγος χρόνος χρειάζεται για να έχουμε το επιθυμητό αποτέλεσμα. Αυτό θα μπορούσε να αποτελεί και κοινή λογική αλλά αποδεικνύεται έμπρακτα μέσα από τα πειράματα μας και τις παρατηρήσεις στις γραφικές που προκύπτουν μέσα από αυτά. Άρα συμπερασματικά λέμε ότι ο χρόνος και κόστος εκτέλεσης είναι ευθέως ανάλογος του πλήθους των στοιχείων που ορίζονται σαν είσοδο για τον αλγόριθμο.

Ο αλγόριθμος αναζήτησης μας πρόσφερε την ευκαιρία τόσο μέσα από το θεωρητικό αλλά και μέσα από το πειραματικό κομμάτι του να εξάγουμε ακόμη ένα συμπέρασμα για τους παράλληλους αλγορίθμους γενικότερα. Το συμπέρασμα αυτό προέκυψε μέσα από την ανάγκη του αλγορίθμου να ορίζουμε τον αριθμό των επεξεργαστών που θα εργάζονται ταυτόχρονα, έτσι θα έπρεπε να ελέγξουμε και αυτή την παράμετρο. Για αυτό το λόγο κάναμε πειράματα όπου διατηρούσαμε σταθερό το πλήθος των δεδομένων και αλλάζαμε τον αριθμό των επεξεργαστών. Μέσα από αυτά τα πειράματα συμπεράναμε ότι όσο αυξάνεται ο αριθμός των επεξεργαστών τόσο μειώνεται ο χρόνος που απαιτείται ώστε να ολοκληρωθεί ο αλγόριθμος μας. Εν κατακλείδι συμπεράνουμε ότι εάν διατηρείται σταθερός ο αριθμός των δεδομένων τότε ο αριθμός των επεξεργαστών είναι αντίστροφος ανάλογος του χρόνου που χρειάζεται για να εκτελεστεί ο αλγόριθμος μας, δεν ισχύει όμως το ίδιο και για το κόστος.

Επίσης ένα ακόμη συμπέρασμα που πηγάζει από όλους τους αλγορίθμους αυτή την φορά είναι ότι κατά την σύγκριση της πρακτικής και θεωρητικής πολυπλοκότητας του χρόνου και του κόστους παίρναμε ασυμπτωτικά τα ίδια αποτελέσματα. Από τις συγκρίσεις αυτές συμπεράνουμε ότι η θεωρητική πολυπλοκότητα ταυτίζεται ασυμπτωτικά με την πειραματική. Εντοπίστηκε όμως σε όλους τους αλγόριθμους το γεγονός ότι τα αποτελέσματα από τα πειράματα είχαν πολύ μεγαλύτερους χρόνους και κατά συνέπεια και κόστη σε σύγκριση με αυτά των θεωρητικών πειραμάτων. Αυτό προκύπτει από το γεγονός ότι στην πραγματικότητα στον αλγόριθμο μας εκτελούμαι και άλλες εντολές όπως την δήλωση μεταβλητών, επιπλέον ελέγχους, αναθέσεις, κάτι που αγνοούμαι και δεν υπολογίζουμε στην θεωρητική ανάλυση.

Επίσης ένα τελευταίο συμπέρασμα που μπορούμε να αναφέρουμε είναι ότι κατά την υλοποίηση ενός βέλτιστου αλγορίθμου και μιας παραλλαγής του τα συμπεράσματα που προκύπτουν είναι ότι ο βέλτιστος αλγόριθμος είναι πιο αποδοτικός από την παραλλαγή και αυτός είναι και ο λόγος που είναι πιο διαδεδομένος. Οι βέλτιστοι αλγόριθμοι είναι πιο αποδοτικοί και προτιμούνται για χρήση.

Συνοψίζοντας, η αξιολόγηση των αλγορίθμων μας οδήγησε στην εξαγωγή και επιβεβαίωση συμπερασμάτων όπως την επιβεβαίωση της θεωρητικής αξιολόγησης.

6.2 Προβλήματα που Συναντήθηκαν

Δεν ήταν λίγα τα προβλήματα που συναντήθηκαν κατά την διάρκεια της διεξαγωγής της Διπλωματικής Εργασίας. Τα προβλήματα αυτά αφορούσαν κυρίως το πρακτικό κομμάτι και στην υλοποίηση των αλγορίθμων στην πλατφόρμα XMT και πιο συγκεκριμένα στη γλώσσα προγραμματισμού XMTC. Για το λόγο ότι η παράλληλη υλοποίηση είναι ένας πολύ καινούριος τομέας υπάρχουν αρκετά σημεία που υστερεί και έτσι συναντήσαμε προβλήματα.

Αρχικά θα ήθελα να αναφέρω το πρόβλημα όπου δεν μπορούν να χρησιμοποιηθούν πέρα τον 1024 επεξεργαστών και στερήθηκε το δικαίωμα να γίνει έλεγχος για μεγαλύτερο αριθμό

στοιχείων στον ένα αλγόριθμο που υλοποιήθηκε όπου θα είχαμε πιο ρεαλιστική άποψη. Το γεγονός αυτό είχε σαν αποτέλεσμα να υπάρχει άνω φράγμα και στο πλήθος των δεδομένων εισόδου αφού είναι πολλές οι περιπτώσεις που εξαρτώνται από τους επεξεργαστές. Επίσης ένα άλλο πρόβλημα που συναντήσαμε από την αρχή της εργασία αυτής είναι ότι δεν μπορεί να γίνει κάλεσμα συνάρτησης σε παράλληλο κομμάτι και άρα ούτε και αναδρομής έτσι ναυάγησε ο αρχικός στόχος της υλοποίησης του προβλήματος του κυρτού προβλήματος παράλληλα. Αν δεν γίνει κάποια επέκταση στην γλώσσα τότε ο αλγόριθμος αυτός δεν θα μπορεί να υλοποιηθεί παράλληλα ποτέ αν και υπάρχει ο θεωρητικός υπολογισμός του. Ένα άλλο πρόβλημα που αντιμετώπισα ήταν ότι δεν μπορούσε να γίνει δυναμική αρχικοποίηση και έτσι καταφεύγαμε σε άλλες μεθόδους όπως για παράδειγμα την στατική. Επίσης δεν μπορούσε να γίνεται ούτε εισαγωγή δεδομένων από το χρήστη κάτι δεν είναι ρεαλιστική αναπαράσταση της πραγματικότητα

6.3 Οφέλη Ατομικής Διπλωματικής Εργασίας

Σίγουρα δεν θα μπορούσα να παραβλέψω τα πολλαπλά οφέλη που αποκόμισα μέσα από την διαδικασία διεκπεραίωσης μιας Διπλωματικής Εργασίας και συγκεκριμένα αυτής. Ήταν η πρώτη φορά που μπήκα στην διαδικασία να φέρω εις πέρας μια τόση μεγάλη εργασία σαν και αυτή. Σε πρώτο επίπεδο δεν μπορώ να αγνοήσω το γεγονός ότι αποκόμισα πάρα πολλά από την διαδικασία συγγραφής της Διπλωματικής Εργασίας, όπως το να συντονίζω και να αναδιοργανώνω σωστά όσα θέλω να ειπωθούν στο χαρτί, επίσης το να ψάχνω για πληροφορίες από τον διαδικτυακό χώρο, σημειώσεις, άρθρα και να παίρνω όσα χρειάζομαι, όσα μου είναι χρήσιμα και πόσο μάλλον να διασταυρώνω κάποιες πληροφορίες για να είμαι σίγουρη για την εγκυρότητα των πηγών και ορθότητα των πληροφοριών. Επιπλέον το θέμα της Διπλωματικής Εργασίας όπου είχα αναλάβει, δηλαδή η παράλληλη επεξεργασία, πιστεύω πως μου πρόσφερε τα μέγιστα και είμαι πολύ χαρούμενη που ασχολήθηκα με αυτό. Όπως ανάφερα εμβάθυνα κατά πολύ στον παραλληλισμό, κατανόησα παράλληλες έννοιες και ορισμούς που δεν είχα την ευκαιρία παλαιότερα παρά μονό σε ένα επιφανειακό επίπεδο μέσα από την παρακολούθηση του σχετικού μαθήματος. Σημαντικό για εμένα ήταν και το γεγονός ότι μπόρεσα να κατανοήσω και σε πρακτικό επίπεδο τόσο τις δυσκολίες όσο τις ευκαιρίες που δίνονται μέσα από τους παράλληλους αλγόριθμους. Οι δυσκολίες με βοήθησαν στον προσπαθώ να βρω τρόπους να τις ξεπεράσω, κάτι που είναι μάθημα ζωής θα τολμούσα να έλεγα, ενώ οι ευκαιρείς με βοήθησαν στο να δω τα πλεονεκτήματα του παραλληλισμού. Επίσης ακόμα μια νέα γλώσσα η XMTC θα αποτελεί μέρος του ενεργητικού μου. Το τελευταίο κομμάτι της διπλωματικής που ήταν η σύγκριση και αξιολόγηση αλγορίθμων ήταν και αυτό ιδιαίτερα χρήσιμο και δεν θα μπορούσα να αγνοήσω τα οφέλη του, όπως της

ικανότητα να μπορώ να καθορίζω τις σωστές παραμέτρους και συνθήκες όπου θα γίνει ένα πείραμα και επίσης την ανάπτυξη κριτικής σκέψης και εξαγωγής συμπερασμάτων. Αυτό θα είναι ιδιαίτερα σημαντικό γιατί είναι οι βάσης και θεμέλια για ερευνητική μελέτη. Το σημαντικότερο που αποκόμισα όμως κατά την άποψη μου ήταν την παράλληλη σκέψη που θα κουβαλώ μαζί μου σε όλη τη συνέχεια που απλώνεται μπροστά μου, αφού ο παραλληλισμός όπως ανέφερα ξανά είναι το μέλλον της Επιστήμης Πληροφορικής, ένα μέλλον που διαγράφεται λαμπρό.

6.4 Μελλοντική Εργασία

Όπως αναφέρθηκε πολλές φορές η πλατφόρμα που ασχολούμαστε XMT και ποιο συγκεκριμένα η γλώσσα XTC αντιμετωπίζει πολλές δυσκολίες και περιορισμούς και ο παραλληλισμός είναι ένας τομέας πολύ ελπιδοφόρος και πολλά υποσχόμενος αφού μας προσφέρει δυνατότητες και ευκαιρίες που δεν έχουμε στην σειριακή επεξεργασία. Σίγουρα μια αναβάθμιση της πλατφόρμας αλλά και τις γλώσσας ώστε να είναι ποιο δεκτική και ευέλικτη θα ήταν ότι καλύτερο θα μπορούσαμε να έχουμε σε μελλοντική εργασία. Σίγουρα αυτό είναι δύσκολο και απαιτεί την συνεργασία πολλών μελετητών, αλλά αν γίνει θα αλλάξει τον τρόπο που λειτουργούν τα πάντα στην Επιστήμη της Πληροφορικής μέχρι τώρα. Στην εργασία αυτή έχουν μελετηθεί ήδη κάποιοι απαιτητικοί αλγόριθμοι όπως τις συνεκτικής συνιστώσας, σε μελλοντικές εργασίες θα μπορούσαν να μελετηθούν και να υλοποιηθούν ποιο πολύπλοκοι έτσι ώστε να δοκιμαστούν τα όρια και οι αντοχές της πλατφόρμας και της παράλληλης γλώσσας. Μια καλή ιδέα για μελλοντική εργασία θα ήταν να βρεθούν τρόποι να αντικαταστήσουμε την κλήση συναρτήσεων μέσα σε παράλληλα κομμάτια (spawn-join), έτσι ώστε να μπορούν να υλοποιηθούν αλγόριθμοι που χρειάζονται την κλήση αναδρομικής συνάρτησης, γιατί μέχρι τώρα δεν μας δίνεται αυτή η ευκαιρία. Ο περιορισμός του αριθμού των επεξεργαστών μας δίνει την ώθηση στο μέλλον να επεκταθεί η πλατφόρμα έτσι ώστε να γίνεται χρήση περισσότερων και να μπορούμε να μιλάμε με μεγαλύτερη ακρίβεια για τα συμπεράσματα που προκύπτουν. Τέλος θεωρώ ότι η μελέτη της πρωτότυπης μηχανής FPGA και σύγκριση της με την δική μας θα ήταν μια πολύ καλή δουλειά που θα καινοτομούσε στο κλάδο του παραλληλισμού.

Βιβλιογραφία

[1] Aydin Balkan, Xingzhi Wen, Fuat Keceli, George Caragea, Alex Tzannes, Mike Horak, Beliz Saybasili, Mary Kiemb, Explicit Multi-Threading (XMT): A PRAM-On-Chip Vision, February 2009

http://www.umiacs.umd.edu/~vishkin/XMT/index.shtml

[2] Aydin Balkan, Xingzhi Wen, Fuat Keceli, George Caragea, Alex Tzannes, Mike Horak, Beliz Saybasili, Mary Kiemb, Software Release of the XMT Environment

For Experimenting with XMT, and Teaching and Self-Studying Parallelism

http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html

[3] Blaise Barney, Introduction to Parallel Computing, Lawrence Livermore Laboratory, USA

[4] Joseph JáJá, An Introduction to Parallel Algorithms, Addision-Wesley Publishing Company, Inc,1992

[5] Tushar Mahapatra & Sanjay Mishra, Oracle Parallel Processing, O'Reilly Media, August 2000

[6] University of Illions, Using Simple Abstraction to Guide the Reinvention of Computing for Parallelism

[7] Uzi Vishkin, The eXplicit MultiThreading (XMT) Parallel Computer Architecture Next generation multi-core supercomputing, University of Maryland

http://www.umiacs.umd.edu/users/vishkin/XMT/talk-12-13-07.pdf

[8] Χρύσης Γεωργίου, Σημειώσεις Μαθήματος ΕΠΛ431-Σύνθεση Παράλληλων Αλγορίθμων, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου.

http://www.cs.ucy.ac.cy/~chryssis/EPL431/

[9] http://en.wikipedia.org/wiki/Algorithm

[10] http://www.umiacs.umd.edu/users/vishkin/index.shtml

[11] http://en.wikipedia.org/wiki/SISD

[12] http://en.wikipedia.org/wiki/SIMD

[13] http://en.wikipedia.org/wiki/MISD

[14] http://en.wikipedia.org/wiki/MIMD

[15] http://zonge.com.au/docs/transmitters/xmt_32_man.pdf

[16] http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

[17] http://en.wikipedia.org/wiki/Kosaraju's_algorithm

[18] http://en.wikipedia.org/wiki/Path-based_strong_component_algorithm

Παράρτημα Α

Στο σημείο αυτό θα εισάγω τους κώδικες που υλοποιήθηκαν στην παράλληλη γλώσσα προγραμματισμού XMTC

Α.1 Παράλληλος Αλγόριθμος Αναζήτησης

```
#include xmtc.h
int main(){
int n;
n=16;
int table[n+2];
int i=0;
int stoixeio;
int processors;
int l;
int r;
stoixeio=9;
processors=3;
int C[processors+2];
int Q[processors+2];
int help_table[processors];
int solution;
int flag=0;
int help2;
for(i=1;i<=n;i++){
table[i]=I;
}
//Bima 1
//Begin spawn1
spawn(1,processors){
if($==1){
l=0;
r=n+1;
table[0]=-10000000;
table[n+1]=10000000;
```

```
C[0]=0;
C[processors+1]=1;
}
}//End of spawn
//Telos vimatos

//Bima 2
//Begin spawn
while(r-l>processors){
flag=0;
spawn(1,processors){
help_table[$]=Q[$];
//find help2
//help2 einai katholiki metavliti
if($==1){
help2=((r-l)/(processors+1));
}
}

spawn(1,processors){
//2.1
if($==1){
Q[0]=1;
Q[processors+1]=r;
}

//2.2
Q[$]=1+($*help2);

//2.3
if(stoixeio==table[Q[$]]){
solution=Q[$];
flag=1;
}
if(flag==0){
if(stoixeio>table[Q[$]]){
```

```
C[$]=0;
}
if(stoixeio<table[Q[$]]){
C[$]=1;
}
}
}//end spawn

spawn(1,processors){
//2.4
if(C[$]<C[$+1]){
l=Q[$];
r=Q[$+1];
}
//2.5
if($==1 && C[0]<C[1]){
l=Q[0];
r=Q[1];
}
}//end of spawn
}//end of while-bima2
//End Step 2

//Step 3
if(flag==0){
spawn(1,processors){
if($<=r-l){
//3.1
//a
if(stoixeio==table[l+$]){
solution=l+$;
flag=1;
}
//b
if(flag==0){
if(stoixeio>table[l+$]){
```

```
C[$]=0;
}
//c
if(stoixeio<table[l+$]){
C[$]=1;
}
//3.2
if(C[$-1]<C[$]){
solution=l+$-1;
}
}//if esoteriko
}//if($<=r-l)
}//end of spawn
}//end of if
//printf("SOLUTION %d\n\n",solution);
}
```

A.2 Παράλληλος Κανονικός Αλγόριθμος Συνεκτικών Συνιστωσών

```
#include <xmtc.h>

int main(){
int n=30;
int A[30][30];
int A_voithitikos[30][30];
int A_iperkomvoi[30][30];
int table[900][2];
int D[n];
int k=0;
int help=0;
int isa=12;



int C[n];
int helpC[n];
int helptable[n];
int i=0;
int j=0;
int num_iperkomvoi=100;//posoi apo autous einai iperkmvoi

int f=1;
int extra[n];
int deutera;
deutera=n*n;

/*Anathesi se kathe epexergastei mia kipselida tou pinaka*/
spawn(1,deutera){
if($%n==0){
table[$-1][0]=($/n)-1;
table[$-1][1]=n-1;
}
if($%n!=0){
table[$-1][0]=($/n);
```

```
table[$-1][1]=($%n)-1;
}
}

A[1][2]=A[1][4]=A[2][1]=A[2][5]=A[12][5]=A[5][12]=A[10][0]=A[0][10]=A[22][21]=A[2
1][22]=1;
A[4][1]=A[5][2]=A[5][6]=A[6][5]=A[13][12]=A[12][13]=A[17][18]=A[18][17]=A[9][12]=
A[12][9]=1;
/*Telos arxikopoiisis distiastatou pinaka*/

/*Antigrafi pin;aka A Ston A_iperkomvoi*/
spawn(1,deutera){
int i=0;
int grammi,stili;
grammi=table[$-1][0];
stili=table[$-1][1];
A_iperkomvoi[grammi][stili]=A[grammi][stili];
}
/*Arxikopoiisi pinaka C[] gia min*/
spawn(1,n){
C[$-1]=100;
}
/*Telos Arxikopoiisis Pinaka C[]*/

/*Start Step 2*/
while(f==1){
/*Start Step 2.2*/
spawn(1,n){
int x=0;
for(x=0;x<n;x++){
if(A_iperkomvoi[$-1][x]==1){
if(C[$-1]>x){
C[$-1]=x+1;
}
}
}
```

```
}
/*End of Step 2.2*/

/*Start Step 2.3*/
spawn(1,n){
if(C[$-1]>$){
C[$-1]=$;
}
}
while(isa!=1){
spawn(1,n){
if (C[$-1]!=helpC[$-1]){
helptable[$-1]=0;
}
else {
helptable[$-1]=1;
}
}
isa=1;
for(i=0;i<n;i++){
if(helptable[i]==0){
isa=0;
}
}
spawn(1,n){
helpC[$-1]=C[$-1];
}
spawn(1,n){
int temp;
temp=C[$-1];
if(C[$-1]>C[temp-1]){
C[$-1]=C[temp-1];
}
}
}
/*End of STep 2.3*/
```

```
/*Start Step 2.4*/
//find the num of iperkomvoi
num_iperkomvoi=0;
for(i=0;i<n;i++){
if(C[i]==i+1){
num_iperkomvoi++;
}
}
int A1[num_iperkomvoi]; //leei poioi arithmoi einai iperkomvoi
int help_iperkomvoi=0;
for(i=0;i<n;i++){
if(C[i]==i+1){
A1[help_iperkomvoi]=i+1;
help_iperkomvoi++;
}
}

int voithitiki=0;

spawn(1,deutera){
int grammi,stili;
grammi=table[$-1][0];
stili=table[$-1][1];
A_voithitikos[grammi][stili]=0;
}

for(voithitiki=1;voithitiki<=num_iperkomvoi;voithitiki++){
int help45[n];
int help_num=0;
for(i=0;i<n;i++){
if(C[i]==A1[voithitiki-1]){
help45[help_num]=i+1;
help_num++;
}
}
```

```
int temp=0;
int xar=0;
int mar;
int ka=0;
ka=0;
xar=0;
mar=0;


for(i=0;i<help_num;i++){
for(j=0;j<n;j++){
if(A_iperkomvoi[help45[i]-1][j]==1){
ka=C[j];
for(mar=0;mar<num_iperkomvoi;mar++){
if(ka==A1[mar]){
xar=mar;
}
}
A_voithitikos[voithitiki-1][xar]=1;
}
}
}
}

/*MIdenismos tis Diagwniou*/
spawn(1,n){
A_voithitikos[$-1][$-1]=0;
}


spawn(1,deutera){
int grammi,stili;
grammi=table[$-1][0];
stili=table[$-1][1];
A_iperkomvoi[grammi][stili]=A_voithitikos[grammi][stili];
```

```
}

f=0;
spawn(1,deutera){
int grammi,stili;
grammi=table[$-1][0];
stili=table[$-1][1];
if(A_iperkomvoi[grammi][stili]==0){
}
if(A_iperkomvoi[grammi][stili]==1){
f=1;
}
}

/*End STep 2.4*/
//for(i=0;i<9;i++){
//printf("%d:%d \n",i+1,C[i]);
//}
//printf("\n");
}//End of while

/*Start of Step 3*/
spawn(1,n){
extra[$-1]=C[$-1];
}
spawn(1,n){
D[$-1]=extra[$-1];
}
spawn(1,n){
if(extra[$-1]==$){
D[$-1]=$;
}
}

int D1[n];
int pa=1;
```

```
int ka;
while(pa==1){
spawn(1,n){
D1[$-1]=D[$-1];
}
spawn(1,n){
if(extra[$-1]==$){
}
else{
ka=D[$-1];
D[$-1]=D[ka-1];
}
pa=0;
for(j=0;j<n;j++)
if(D1[$-1]==D[$-1]){
}
else{
pa=1;
}
}
}
//printf("\nPInakas D\n");
//for(i=0;i<n;i++){
//printf("%d",D[i]);
//}

/*End of Step 3*/
}
```

A.3 Παράλληλος Αλγόριθμος Παραλλαγής Συνεκτικών Συνιστωσών

```
#include <xmtc.h>

int main(){
int n=30;
int A[30][30];
int A_voithitikos[30][30];
int A_iperkomvoi[30][30];
int D[n];
int k=0;
int help=0;
int isa=12;

int C[n];
int helpC[n];
int helptable[n];
int i=0;
int j=0;
int num_iperkomvoi=100;//posoi apo autous einai iperkmvoi

int f=1;
int extra[n];

int voithitiki=0;

/*Arxikopoiisi Distiastatou Pinaka*/
spawn(1,n){
for(i=0;i<n;i++){
A[$-1][i]=0;
}
}
A[1][2]=A[1][4]=A[2][1]=A[2][5]=A[12][5]=A[5][12]=A[10][0]=A[0][10]=A[22][21]=A[2
1][22]=1;
```

```
A[4][1]=A[5][2]=A[5][6]=A[6][5]=A[13][12]=A[12][13]=A[17][18]=A[18][17]=A[9][12]=
A[12][9]=1;
/*Telos arxikopoiisis distiastatou pinaka*/

/*Antigrafi pin;aka A Ston A_iperkomvoi*/
spawn(1,n){
int i=0;
for(i=0;i<n;i++){
A_iperkomvoi[$-1][i]=A[$-1][i];
}
}
/*Arxikopoiisi pinaka C[] gia min*/
spawn(1,n){
C[$-1]=100;
}
/*Telos Arxikopoiisis Pinaka C[]*/

/*Start Step 2*/
while(f==1){

/*Start Step 2.2*/
spawn(1,n){
int x=0;
for(x=0;x<n;x++){
if(A_iperkomvoi[$-1][x]==1){
if(C[$-1]>x){
C[$-1]=x+1;

}

}
}
}
/*End of Step 2.2*/

/*Start Step 2.3*/
```

```
spawn(1,n){
if(C[$-1]>$){
C[$-1]=$;
}
}
while(isa!=1){
spawn(1,n){
if (C[$-1]!=helpC[$-1]){
helptable[$-1]=0;
}
else {
helptable[$-1]=1;
}
}
isa=1;
for(i=0;i<n;i++){
if(helptable[i]==0){
isa=0;
}
}
spawn(1,n){
helpC[$-1]=C[$-1];
}
spawn(1,n){
int temp;
temp=C[$-1];
if(C[$-1]>C[temp-1]){
C[$-1]=C[temp-1];
}
}
}
/*End of STep 2.3*/

/*Start Step 2.4*/
//find the num of iperkomvoi
num_iperkomvoi=0;
```

```
for(i=0;i<n;i++){
if(C[i]==i+1){
num_iperkomvoi++;
}
}
int A1[num_iperkomvoi]; //leei poioi arithmoi einai iperkomvoi
int help_iperkomvoi=0;
for(i=0;i<n;i++){
if(C[i]==i+1){
A1[help_iperkomvoi]=i+1;
help_iperkomvoi++;
}
}




spawn(1,n){
int i=0;
for(j=0;j<n;j++){
A_voithitikos[$-1][j]=0;
}
}

for(i=0;i<n;i++){
for(j=0;j<n;j++){
A_voithitikos[i][j]=0;
}
}
for(voithitiki=1;voithitiki<=num_iperkomvoi;voithitiki++){
int help45[n];
int help_num=0;
for(i=0;i<n;i++){
if(C[i]==A1[voithitiki-1]){
help45[help_num]=i+1;
help_num++;
}
```

```
}
int temp=0;
int xar=0;
int mar;
int ka=0;
ka=0;
xar=0;
mar=0;



for(i=0;i<help_num;i++){
for(j=0;j<n;j++){
if(A_iperkomvoi[help45[i]-1][j]==1){
ka=C[j];
for(mar=0;mar<num_iperkomvoi;mar++){
if(ka==A1[mar]){
xar=mar;
}
}
A_voithitikos[voithitiki-1][xar]=1;
}
}
}
}

/*MIdenismos tis Diagwniou*/
spawn(1,n){
A_voithitikos[$-1][$-1]=0;
}



spawn(1,n){
int i=0;
for(i=0;i<n;i++){
A_iperkomvoi[$-1][i]=A_voithitikos[$-1][i];
}
```

```
}

f=0;
//for(i=0;i<n;i++)
spawn(1,n){
int j;
for(j=0;j<n;j++){
if(A_iperkomvoi[$-1][j]==0){
}
if(A_iperkomvoi[$-1][j]==1){
f=1;
}
}
}
/*End STep 2.4*/
}//End of while

/*Start of Step 3*/
spawn(1,n){
extra[$-1]=C[$-1];
}
spawn(1,n){
D[$-1]=extra[$-1];
}
spawn(1,n){
if(extra[$-1]==$){
D[$-1]=$;
}
}

int D1[n];
int pa=1;
int ka;
while(pa==1){
spawn(1,n){
D1[$-1]=D[$-1];
```

```
}
for(i=0;i<n;i++){

if(extra[i]==i+1){
}
else{
ka=D[i];
D[i]=D[ka-1];
}
}
pa=0;
for(j=0;j<n;j++){
if(D1[i]==D[i]){
}
else{
pa=1;
}
}
}

/*End of Step 3*/
}
```